

1- 微服务

1.1 微服务基本概念

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相沟通（通常是基于HTTP的RESTful API）。

1.2 单体应用架构

所谓单体应用架构只要是指，**将一个应用中的所有服务都封装在一个应用中**。常见的ERP、CMS系统，都是将数据库，web访问等所有功能放入到一个war包中，然后发布到tomcat等容器中的应用。

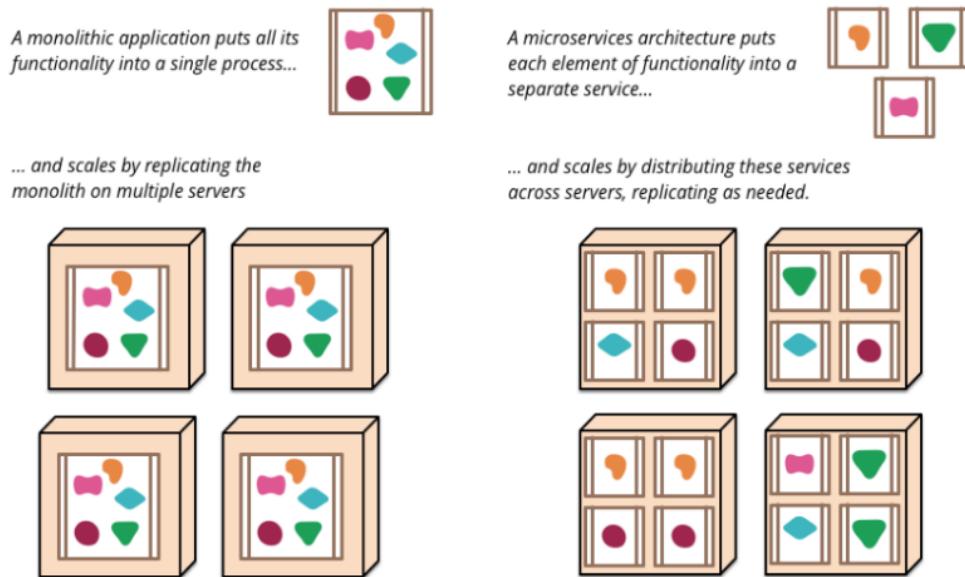
单体应用架构优点

易于开发和测试，也十分方便部署。当需要扩展时，只需要将war包多复制几份，将其放置服务器中，通过nginx做个负载均衡就可以了。

单体应用架构缺点

哪怕是需要修改非常小的地方，都要将整个服务停掉，然后重新打包，部署这个war包。当面临一个大型项目时，不可能将所有的内容都放在一个应用里面。如何维护和分工合作都是问题。

1.3 微服务架构



所谓微服务架构，就是要解决单体应用架构的方式，将每个功能元素独立出来，把独立出来的功能元素的动态组合，需要的功能元素才去拿来组合，需要将多一些时间可以整合多个功能元素。**所以微服务架构只是对功能元素进行复制，而没有对整个应用进行复制。**

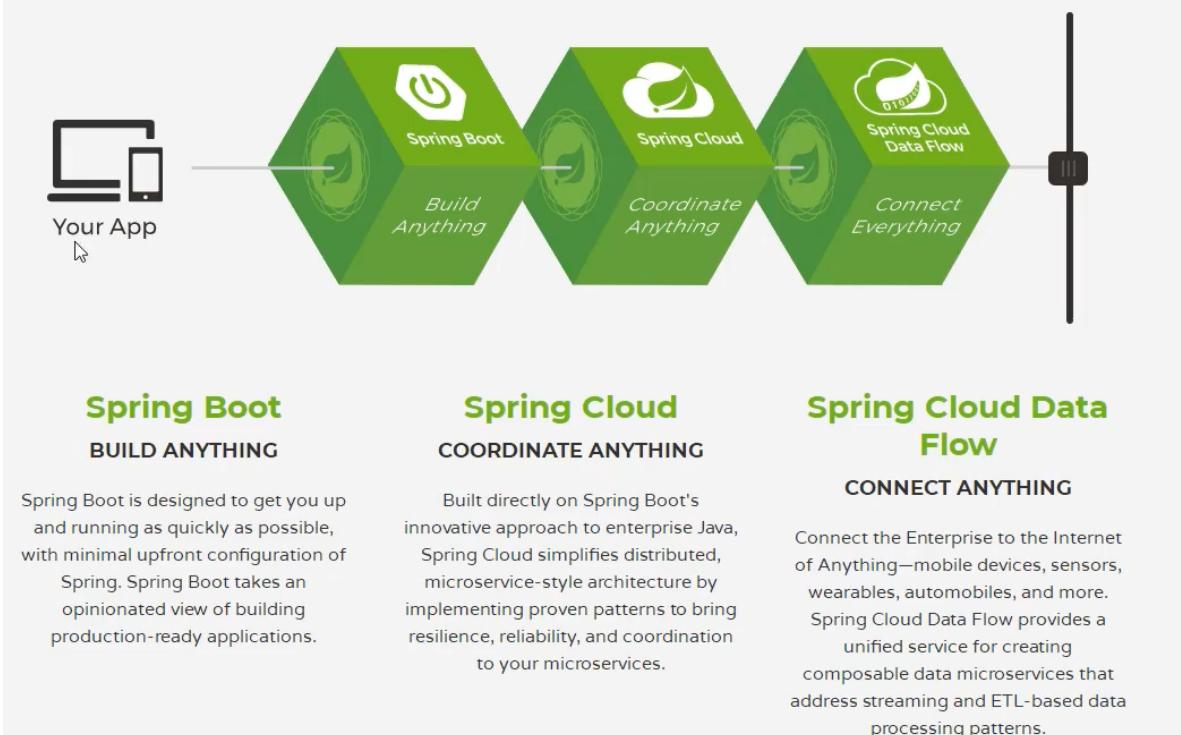
微服务架构好处

节省了调用资源，每个功能元素的服务都是一个可替换的，可独立升级的软件代码！！！

微服务的具体解释: <https://martinfowler.com/articles/microservices.html>

如何构建微服务

Spring: the source for modern java



一个大型系统的微服务架构，就像一个复杂交织的神经网络，每一个神经元就是一个功能元素，它们各自完成自己的功能，然后通过http相互请求调用。比如京东商城。缓存、数据库、浏览页面、结账、支付等服务都是一个个独立的功能服务，都被微化了。它们作为了一个个微服务共同构建一个庞大的系统，如果修改其中的一个功能，只需要更新升级其中一个功能服务单元即可。但是这种庞大的系统架构给部署和运维带来很大的难度，于是，Spring给我们带来了构建大型分布式微服务的全套、全程产品。

- 构建一个个功能独立的微服务应用单元，可以使用Spring Boot，可以帮助快速构建一个应用。
- 大型分布式网络服务的调用，这部分由Spring cloud来完成，实现分布式。
- Spring 为我们想清楚了整个开始构建应用到大型分布式应用全流程方案。

2- SpringBoot简介

2.1 约定优于配置

1 | Build Anything with Spring Boot: Spring Boot is the starting point for building all Spring-based applications. Spring Boot is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring.

上面是引自官网的一段话，大概是说：Spring Boot 是所有基于 Spring 开发的项目的起点，Spring Boot 的设计是为了让你尽可能快的跑起来 Spring 应用程序并且尽可能减少你的配置文件。

约定优于配置(Convention over Configuration)，又称按约定编程

本质上是说是一种软件设计范式，系统、类库或框架应该假定合理的默认值，而非要求提供不必要的配置。默认帮我们进行了很多设置，多数 Spring Boot 应用只需要很少的 Spring 配置。同时它集成了大量常用第三方库配置（例如 Redis、MongoDB、Jpa、RabbitMQ、Quartz 等等），SpringBoot 应用中这些第三方库几乎可以零配置的开箱即用。

2.2 Spring优缺点分析

优点

Spring是Java企业版 (Java Enterprise Edition, JEE, 也称J2EE) 的轻量级代替品。无需开发重量级的Enterprise Java Bean (EJB)，Spring为企业级Java开发提供了一种相对简单的方法，通过依赖注入和面向切面编程，用简单的Java对象 (Plain Old Java Object, POJO) 实现了EJB的功能。

缺点

虽然Spring的组件代码是轻量级的，但它的配置却是重量级的。一开始，Spring用XML配置，而且是很多XML配置。Spring 2.5引入了基于注解的组件扫描，这消除了大量针对应用程序自身组件的显式XML配置。

Spring 3.0引入了基于Java的配置，这是一种类型安全的可重构配置方式，可以代替XML。所有这些配置都代表了开发时的损耗。因为在思考Spring特性配置和解决业务问题之间需要进行思维切换，所以编写配置挤占了编写应用程序逻辑的时间。和所有框架一样，Spring实用，但与此同时它要求的回报也不少。

除此之外，项目的依赖管理也是一件耗时耗力的事情。在环境搭建时，需要分析要导入哪些库的坐标，而且还需要分析导入与之有依赖关系的其他库的坐标，一旦选错了依赖的版本，随之而来的不兼容问题就会严重阻碍项目的开发进度SSM整合：Spring、Spring MVC、Mybatis、Spring-Mybatis整合包、数据库驱动，引入依赖的数量繁多、容易存在版本冲突。

2.3 Spring Boot解决上述spring问题

SpringBoot对上述Spring的缺点进行的改善和优化，基于约定优于配置的思想，可以让开发人员不必在配置与逻辑业务之间进行思维的切换，全身心的投入到逻辑业务的代码编写中，从而大大提高了开发的效率，一定程度上缩短了项目周期。

起步依赖

起步依赖本质上是一个Maven项目对象模型(Project Object Model, POM)，定义了对其他库的传递依赖，这些东西加在一起即支持某项功能。

简单的说，起步依赖就是将具备某种功能的依赖坐标打包到一起，并提供一些默认的功能。

自动配置

springboot的自动配置，指的是springboot会自动将一些配置类的bean注册进ioc容器，我们可以需要的地方使用@Autowired或者@Resource等注解来使它。

“自动”的表现形式就是我们只需要引我们想用功能的包，相关的配置我们完全不用管，springboot会自动注入这些配置bean，我们直接使用这些bean即可。springboot: 简单、快速、方便地搭建项目；对主流开发框架的无配置集成；极大提高了开发、部署效率。

3- 入门案例

3.1 环境准备

我们将学习如何快速的创建一个Spring Boot应用，并且实现一个简单的Http请求处理。通过这个例子对Spring Boot有一个初步的了解，并体验其结构简单、开发快速的特性。

环境准备

- jdk: "1.8"
- Maven-3.6
- SpringBoot 2.x 最新版

开发工具

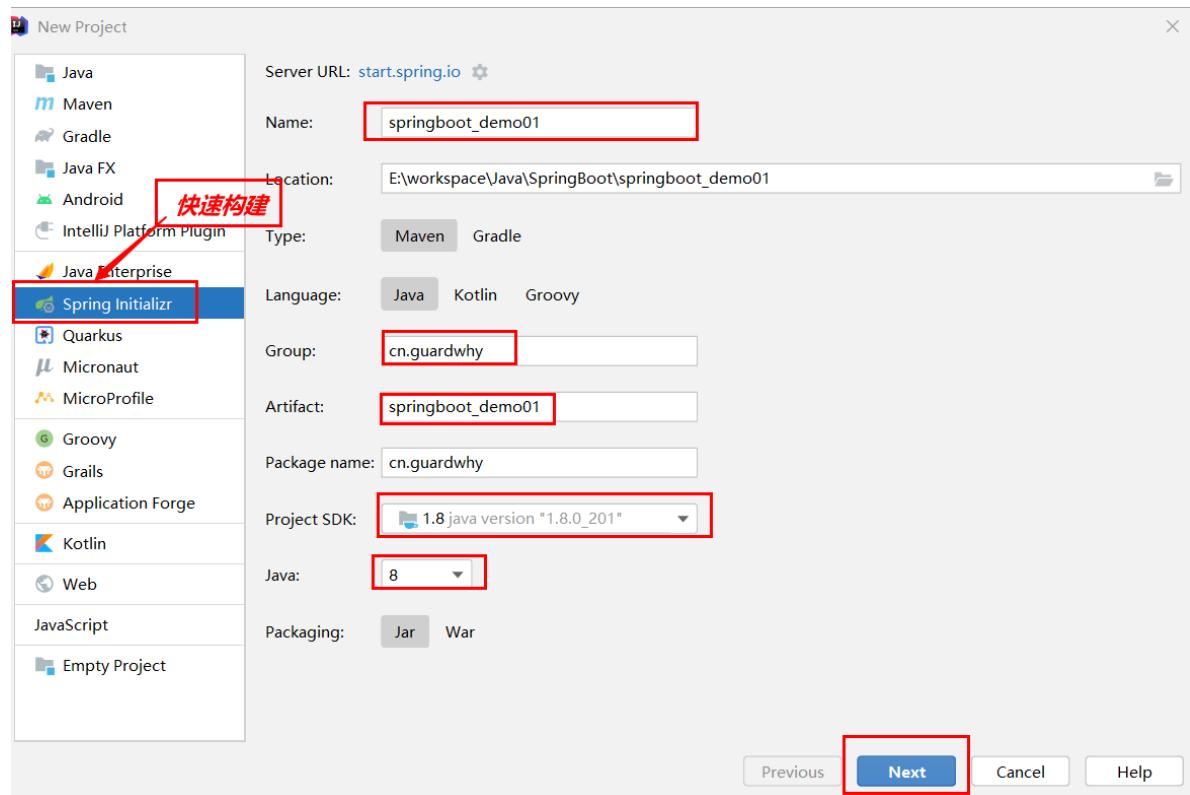
- IDEA

3.2 SpringBoot 快速构建

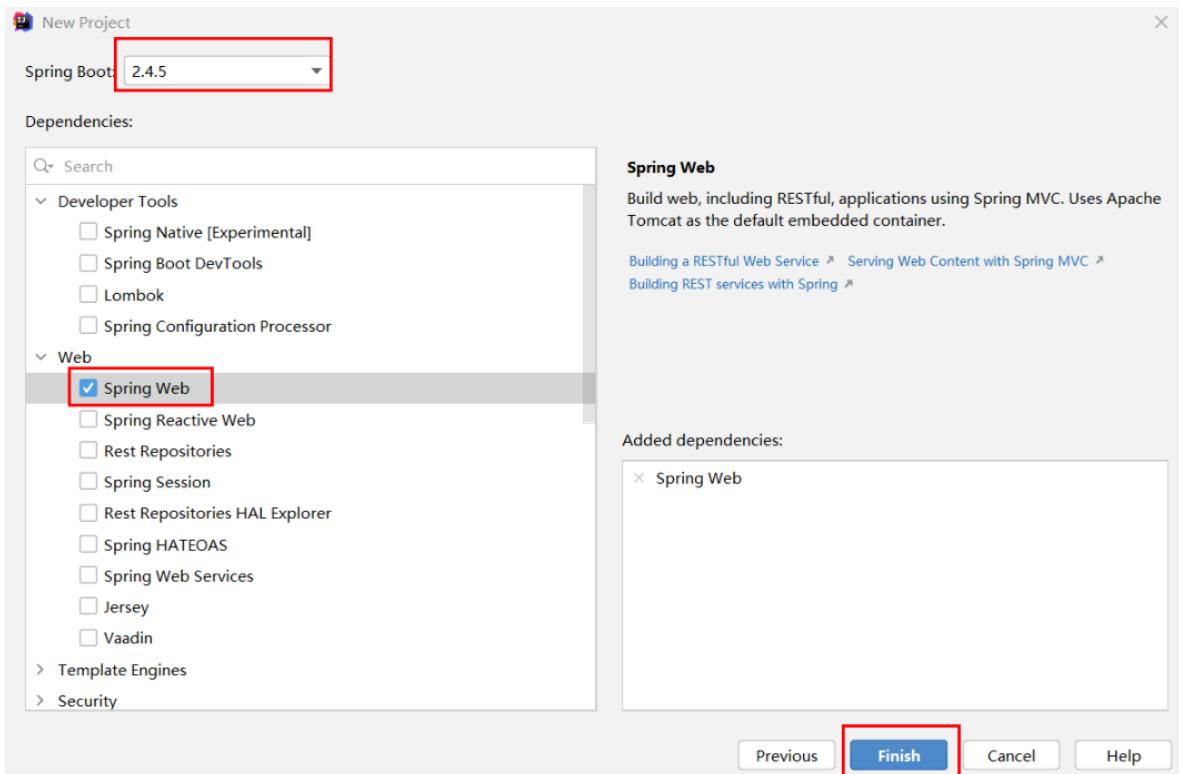
案例需求：请求Controller中的方法，并将返回值响应到页面。

使用Spring Initializr方式构建Spring Boot项目

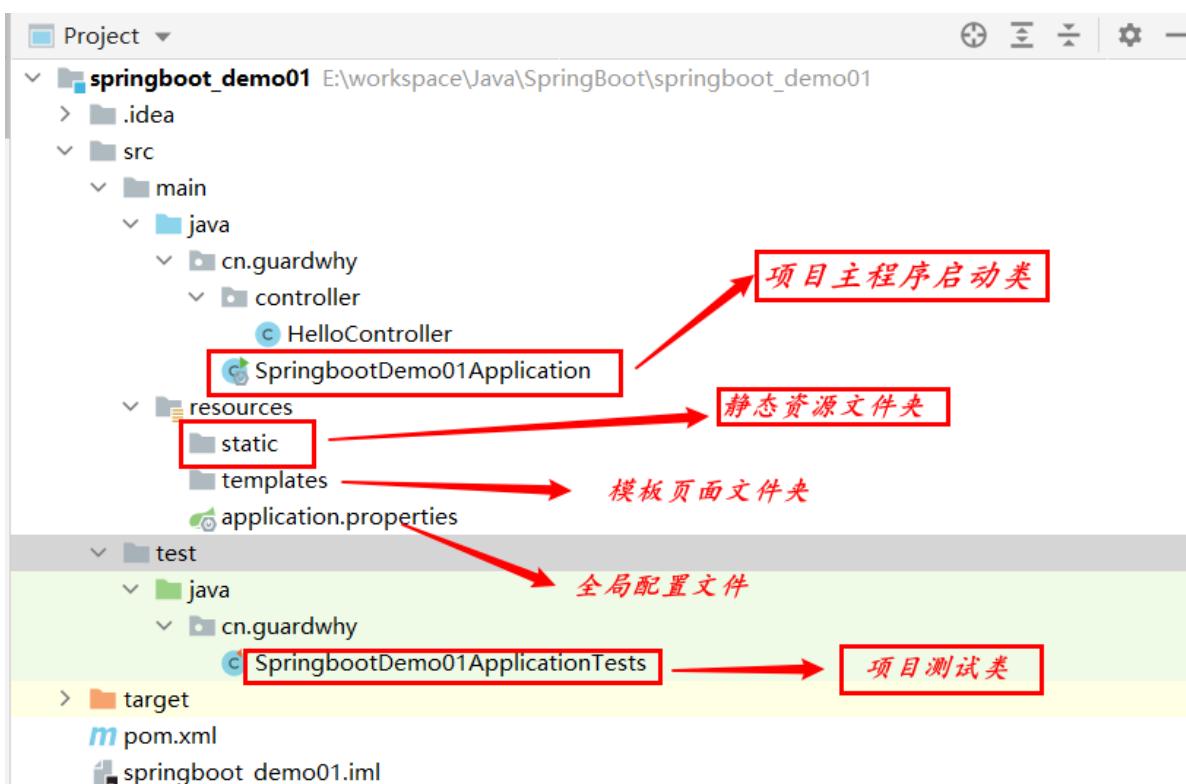
本质上说，Spring Initializr是一个Web应用，它提供了一个基本的项目结构，能够帮助我们快速构建一个基础的Spring Boot项目。



选择当前最新的稳定版本，创建工程时候选中web依赖。



Spring Boot项目就创建好了，创建好的Spring Boot项目结构如图。



使用Spring Initializr方式构建的Spring Boot项目会默认生成项目启动类；

存放前端静态资源和页面的文件夹、编写项目配置的配置文件以及进行项目单元测试的测试类。

pom.xml 分析

打开 `pom.xml`，看看Spring Boot项目的依赖

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

```
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5       https://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7   <!--父依赖-->
8   <parent>
9     <groupId>org.springframework.boot</groupId>
10    <artifactId>spring-boot-starter-parent</artifactId>
11    <version>2.4.5</version>
12    <relativePath/>
13  </parent>
14  <groupId>cn.guardwhy</groupId>
15  <artifactId>springboot_demo01</artifactId>
16  <version>0.0.1-SNAPSHOT</version>
17  <name>springboot_demo01</name>
18  <description>Demo project for Spring Boot</description>
19  <properties>
20    <java.version>1.8</java.version>
21  </properties>
22  <dependencies>
23    <!--web场景启动器-->
24    <dependency>
25      <groupId>org.springframework.boot</groupId>
26      <artifactId>spring-boot-starter-web</artifactId>
27    </dependency>
28    <!--Springboot单元测试-->
29    <dependency>
30      <groupId>org.springframework.boot</groupId>
31      <artifactId>spring-boot-starter-test</artifactId>
32      <scope>test</scope>
33    </dependency>
34  </dependencies>
35
36  <build>
37    <plugins>
38      <!--打包插件-->
39      <plugin>
40        <groupId>org.springframework.boot</groupId>
41        <artifactId>spring-boot-maven-plugin</artifactId>
42      </plugin>
43    </plugins>
44  </build>
45
46</project>
```

创建一个用于Web访问的Controller

在主程序的同级目录下，新建一个controller包，**一定要在同级目录下，否则识别不到。**
在包中新建一个HelloController类

```

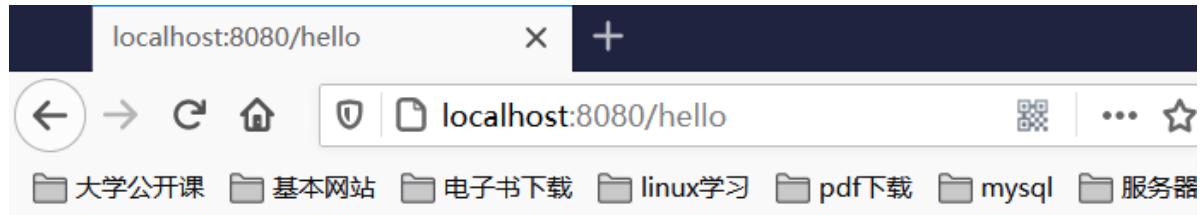
1 package cn.guardwhy.controller;
2
3 import org.springframework.web.bind.annotation.RequestMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 public class HelloController {
8     @RequestMapping("/hello")
9     public String hello(){
10         return "Hello Spring Boot!!!";
11     }
12 }

```

运行项目

运行主程序启动类SpringbootDemo01Application，项目启动成功后。

在控制台上会发现SpringBoot项目默认启动的端口号为8080，此时，可以在浏览器上访问
<http://localhost:8080/hello>

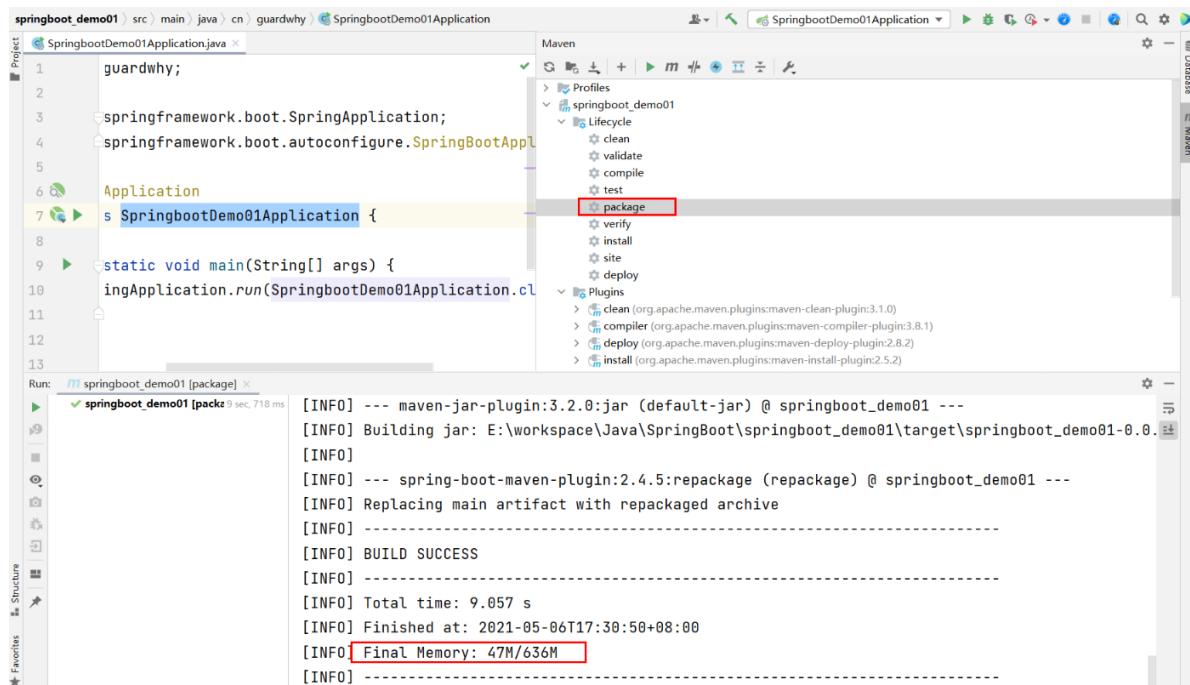


Hello Spring Boot!!!

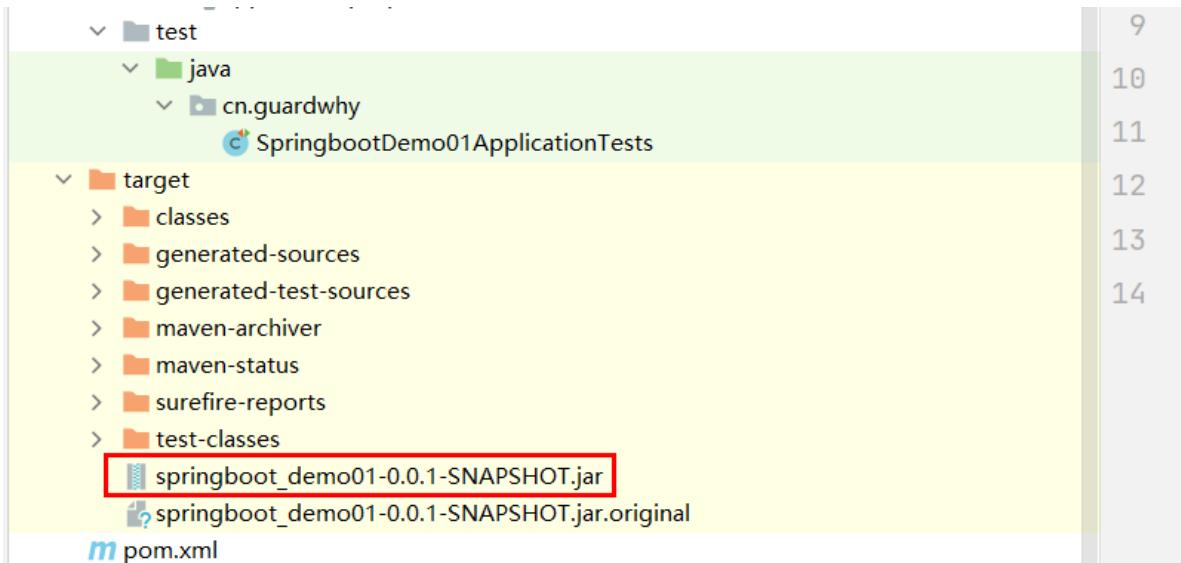
页面输出的内容是“Hello Spring Boot!!!”，至此，构建Spring Boot项目就完成了。

3.3 项目打包

将项目打成jar包，点击 maven的 package！！！



如果打包成功，则会在target目录下生成一个jar包。



打成了jar包后，就可以在任何服务器上运行了，执行 `java -jar xxx.jar` 命令！！！

```
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

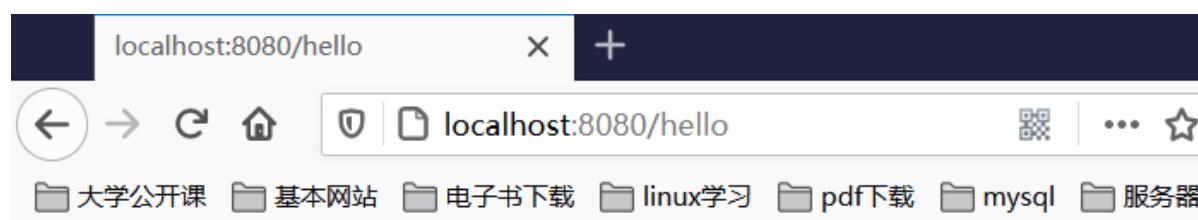
尝试新的跨平台 PowerShell https://aka.ms/pscore6

PS E:\workspace\Java\SpringBoot\springboot_demo01\target> java -jar springboot_demo01-0.0.1-SNAPSHOT.jar

  _/ \_ /----'_--'_--'_(_)_--'_ \ \ \ \ \
 ((_)_--| '_| '_| '_| '_/ \_ | \ \ \ \
 \| \_--| |_)| | | | | | | | | | ) ) ) )
   ' | _--| | .| | | | | | | | | | | | | |
=====| | =====| | | | | | | | | | | | | |
 :: Spring Boot ::          (v2.4.5)

2021-05-06 17:37:40.076 INFO 17396 --- [           main] cn.guardwhy.SpringbootDemo01Application : Starting Springboot
Demo01Application v0.0.1-SNAPSHOT using Java 1.8.0_251 on GuardWhy with PID 17396 (E:\workspace\Java\SpringBoot\springbo
ot_demo01\target\springboot_demo01-0.0.1-SNAPSHOT.jar started by linux in E:\workspace\Java\SpringBoot\springboot_demo01
\target)
2021-05-06 17:37:40.079 INFO 17396 --- [           main] cn.guardwhy.SpringbootDemo01Application : No active profile s
```

执行结果



Hello Spring Boot!!!

4- SpringBoot原理

4.1 相关依赖配置

Pom.xml

它主要是依赖一个父项目，主要是管理项目的资源过滤及插件！

```
1 <!--父依赖-->
2 <parent>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-starter-parent</artifactId>
5   <version>2.4.5</version>
6 </parent>
```

点进去，发现还有一个父依赖。

```
1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-dependencies</artifactId>
4   <version>2.4.5</version>
5 </parent>
```

这里才是真正管理SpringBoot应用里面所有依赖版本的地方，SpringBoot的版本控制中心，以后导入依赖默认是不需要写版本。

但是如果导入的包没有在依赖中管理着就需要手动配置版本了。

启动器 spring-boot-starter

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4 </dependency>
```

- springboot-boot-starter-xxx：就是spring-boot的场景启动器。
- spring-boot-starter-web：帮我们导入了web模块正常运行所依赖的组件。
- SpringBoot将所有的功能场景都抽取出来，做成一个个的starter 启动器，只需要在项目中引入这些starter即可，所有相关的依赖都会导入进来。
- 要用什么功能就导入什么样的场景启动器即可，我们未来也可以自己自定义 starter。

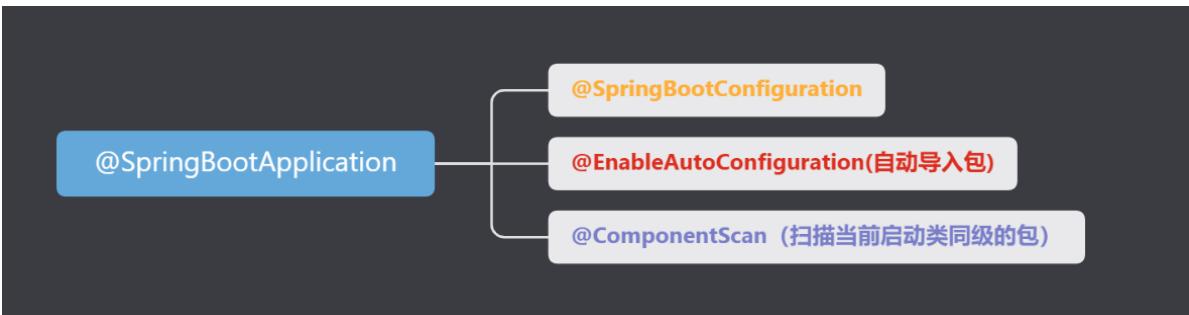
4.2 主启动类自动装配原理

默认启动类

```
1 package cn.guardwhy;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication // 标注了一个主程序类，说明这是一个Spring Boot应用。
7 public class SpringbootDemo01Application {
8
9     public static void main(String[] args) {
10         // 启动了一个服务
11         SpringApplication.run(SpringbootDemo01Application.class, args);
12     }
13 }
```

@SpringBootApplication

进入这个注解，可以看到上面还有很多其他注解。



作用：标注在某个类上说明这个类是SpringBoot的主配置类，SpringBoot就应该运行这个类的main方法来启动SpringBoot应用。

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration // 标注在类上
@EnableAutoConfiguration // 标注在类上
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
                                @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringApplication {
}

```

```

1 public @interface SpringApplication {
2     // 源码示例
3 }

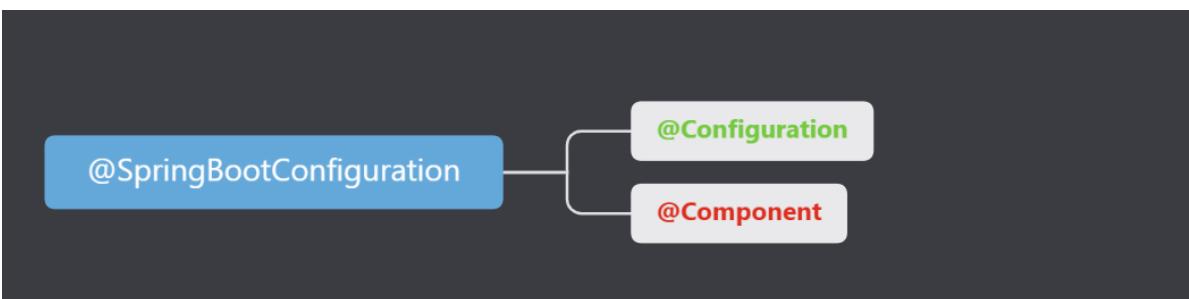
```

4.2.1 @ComponentScan

这个注解在Spring中很重要，它对应XML配置中的元素，扫描当前启动类同级的包。
自动扫描并加载符合条件的组件或者bean，将这个bean定义加载到IOC容器中。

4.2.2 @SpringBootConfiguration

标注在某个类上，表示这是一个SpringBoot的配置类，继续点击注解查看。



@Configuration

```

1 @Target({ElementType.TYPE})
2 @Retention(RetentionPolicy.RUNTIME)
3 @Documented
4 @Configuration
5 public @interface SpringBootConfiguration {
6     @AliasFor(
7         annotation = Configuration.class
8     )
9     boolean proxyBeanMethods() default true;
10 }

```

这里的 @Configuration, 说明这是一个配置类 , 配置类就是对应Spring的xml 配置文件。继续点击下去

@Component

```
1  @Target({ElementType.TYPE})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Component
5  public @interface Configuration {
6      @AliasFor(
7          annotation = Component.class
8      )
9      String value() default "";
10
11     boolean proxyBeanMethods() default true;
12 }
```

里面的 @Component 这就说明，启动类本身也是Spring中的一个组件而已，负责启动应用！ !!

4.2.3 @EnableAutoConfiguration

单体应用架构需要自己配置文件，而现在SpringBoot可以自动帮我们配置。

@EnableAutoConfiguration(自动导入包)告诉SpringBoot开启自动配置功能，这样自动配置才能生效，点击注解继续查看。

@AutoConfigurationPackage

具体作用

自动配置包，继续点击下去。

```
1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Inherited
5  @AutoConfigurationPackage
6  @Import(AutoConfigurationImportSelector.class)
7  public @interface EnableAutoConfiguration {
8
9      /**
10       * Environment property that can be used to override when auto-
11       * configuration is
12       * enabled.
13       */
14      String ENABLED_OVERRIDE_PROPERTY =
15      "spring.boot.enableautoconfiguration";
16
17      /**
18       * Exclude specific auto-configuration classes such that they will never
19       * be applied.
20       * @return the classes to exclude
21       */
22      Class<?>[] exclude() default {};
23
24      /**
25       *
26       */
27 }
```

```

22     * Exclude specific auto-configuration class names such that they will
23     never be
24     * applied.
25     * @return the class names to exclude
26     * @since 1.3.0
27     */
28     String[] excludeName() default {};
29 }

```

@Import(AutoConfigurationPackages.Registrar.class)

基本作用

Spring底层注解@import，给容器中导入一个组件。Registrar.class将主启动类的所在包及包下面所有子包里面的所有组件扫描到Spring容器。

自动配置，自动注册包。

```

1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Inherited
5  @Import(AutoConfigurationPackages.Registrar.class)
6  public @interface AutoConfigurationPackage {
7
8      /**
9       * Base packages that should be registered with {@link
10      AutoConfigurationPackages}.
11      * <p>
12      * Use {@link #basePackageClasses} for a type-safe alternative to
13      String-based package
14      * names.
15      * @return the base package names
16      * @since 2.3.0
17      */
18      String[] basePackages() default {};
19
20      /**
21       * Type-safe alternative to {@link #basePackages} for specifying the
22       packages to be
23       * registered with {@link AutoConfigurationPackages}.
24       * <p>
25       * Consider creating a special no-op marker class or interface in each
26       package that
27       * serves no purpose other than being referenced by this attribute.
28       * @return the base package classes
29       * @since 2.3.0
30       */
31      Class<?>[] basePackageClasses() default {};
32 }

```

```
@Import(AutoConfigurationImportSelector.class)
```

给容器导入组件，自动导入包的核心！！！

```
1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Documented
4 @Inherited
5 @AutoConfigurationPackage
6 @Import(AutoConfigurationImportSelector.class)
7 public @interface EnableAutoConfiguration {
8     String ENABLED_OVERRIDE_PROPERTY =
9         "spring.boot.enableautoconfiguration";
10    Class<?>[] exclude() default {};
11    String[] excludeName() default {};
12 }
```

AutoConfigurationImportSelector

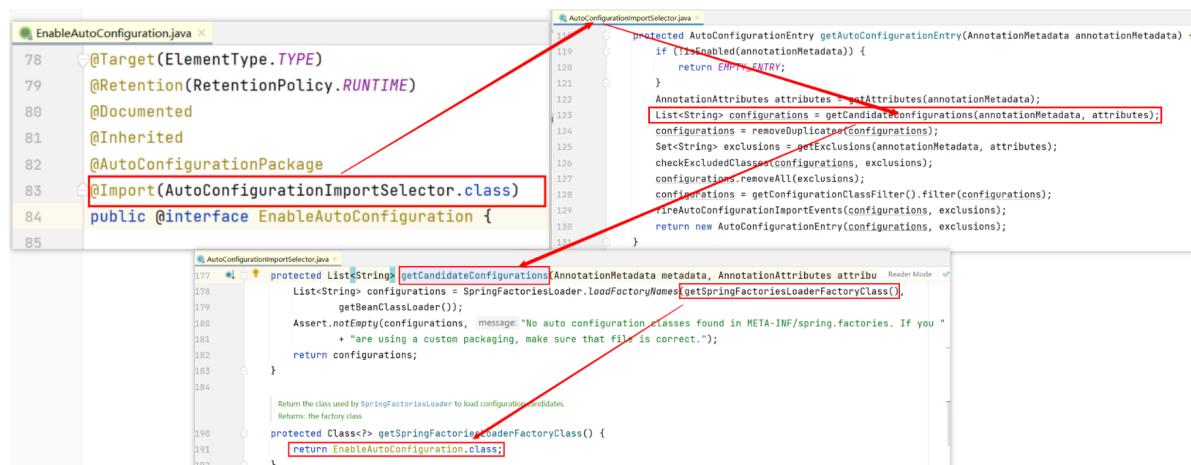
自动导入选择器，选择了啥？

获取所有的配置，继续点击。

```
1 List<String> configurations = getCandidateConfigurations(annotationMetadata,
attributes);
```

获取候选的配置

```
1 // 方法
2 protected List<String> getCandidateConfigurations(AnnotationMetadata
metadata, AnnotationAttributes attributes) {
3     List<String> configurations =
SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass()
,
4
getBeanClassLoader());
5     Assert.notEmpty(configurations, "No auto configuration classes found in
META-INF/spring.factories. If you "
+ "are using a custom packaging, make sure that file is
correct.");
6     return configurations;
7 }
8 }
```



getAutoConfigurationEntry()

获得自动配置的实体！！！

```
78     @Target(ElementType.TYPE)
79     @Retention(RetentionPolicy.RUNTIME)
80     @Documented
81     @Inherited
82     @AutoConfigurationPackage
83     @Import(AutoConfigurationImportSelector.class) 点击
84     public @interface EnableAutoConfiguration {
85 }
```

```
18     protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
19         if (!isEnabled(annotationMetadata)) {
20             return EMPTY_ENTRY;
21         }
22         AnnotationAttributes attributes = getAttributes(annotationMetadata);
23         List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
24         configurations = removeDuplicates(configurations);
25         Set<String> exclusions = getExclusions(annotationMetadata, attributes);
26         checkExcludedClasses(configurations, exclusions);
27         configurations.removeAll(exclusions);
28         configurations = getConfigurationClassFilter().filter(configurations);
29         fireAutoConfigurationImportEvents(configurations, exclusions);
30         return new AutoConfigurationEntry(configurations, exclusions);
31     }
```

getCandidateConfigurations()

获取候选的配置！！！

```
177     protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
178         List<String> configurations = SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),
179             getBeanClassLoader());
180         Assert.notEmpty(configurations, message: "No auto configuration classes found in META-INF/spring.factories. If you "
181             + "are using a custom packaging, make sure that file is correct.");
182         return configurations;
183     }
184     /**
185      * Return the class user by {@link SpringFactoriesLoader} to load configuration
186      * candidates.
187      * @return the factory class
188      */
189     protected Class<?> getSpringFactoriesLoaderFactoryClass() {
190         return EnableAutoConfiguration.class;
191     }
192 }
```

public static List loadFactoryNames()

获取所有的加载配置！！！

```

1 AutoConfigurationImportSelector.java
177     protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
178         List<String> configurations = SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderClass(),
179             getClassLoader());
180         Assert.notEmpty(configurations, message: "No auto configuration classes found in META-INF/spring.factories. If you "
181             + "are using a custom packaging, make sure that file is correct.");
182         return configurations;
183     }
184
185 SpringFactoriesLoader.java
125     */
126     public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable ClassLoader classLoader) {
127         ClassLoader classLoaderToUse = classLoader;
128         if (classLoaderToUse == null) {
129             classLoaderToUse = SpringFactoriesLoader.class.getClassLoader();
130         }
131         String factoryTypeName = factoryType.getName();
132         return loadSpringFactories(classLoaderToUse).getOrDefault(factoryTypeName, Collections.emptyList());
133     }

```

4.2.4 loadSpringFactories

继续点击源码，调用了 **SpringFactoriesLoader** 类的静态方法！进入 **SpringFactoriesLoader** 类 **loadFactoryNames()** 方法

```

1 public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable
2     ClassLoader classLoader) {
3     ClassLoader classLoaderToUse = classLoader;
4     if (classLoader == null) {
5         classLoaderToUse = SpringFactoriesLoader.class.getClassLoader();
6     }
7
8     String factoryTypeName = factoryType.getName();
9     return
10    (List)loadSpringFactories(classLoaderToUse).getOrDefault(factoryTypeName,
11        Collections.emptyList());
12}

```

```

1 AutoConfigurationImportSelector.java
176     */
177     protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
178         List<String> configurations = SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderClass(),
179             getClassLoader());
180         Assert.notEmpty(configurations, message: "No auto configuration classes found in META-INF/spring.factories. If you "
181             + "are using a custom packaging, make sure that file is correct.");
182         return configurations;
183     }
184
185 SpringFactoriesLoader.class
62
63     public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable ClassLoader classLoader) {
64         ClassLoader classLoaderToUse = classLoader;
65         if (classLoader == null) {
66             classLoaderToUse = SpringFactoriesLoader.class.getClassLoader();
67         }
68
69         String factoryTypeName = factoryType.getName();
70         return (List)loadSpringFactories(classLoaderToUse).getOrDefault(factoryTypeName, Collections.emptyList());
71     }

```

继续点击查看 **loadSpringFactories 方法**

获取项目资源: classLoader.getResources(FACTORIES_RESOURCE_LOCATION);

```
1 private static Map<String, List<String>> loadSpringFactories(ClassLoader
2 classLoader) {
3     Map<String, List<String>> result = cache.get(classLoader);
4     // 获得classLoader, 返回可以看到这里得到的EnableAutoConfiguration标注的类本身
5     if (result != null) {
6         return result;
7     }
8
8     result = new HashMap<>();
9     try {
10         Enumeration<URL> urls =
11             classLoader.getResources(FACTORIES_RESOURCE_LOCATION);
12         // 判断是否存在更多的元素
13         while (urls.hasMoreElements()) {
14             URL url = urls.nextElement();
15             UrlResource resource = new UrlResource(url);
16             // 将读取到的资源遍历, 封装成一个Properties, 所有的资源加载到配置类中
17             Properties properties =
18                 PropertiesLoaderUtils.loadProperties(resource);
19             for (Map.Entry<?, ?> entry : properties.entrySet()) {
20                 String factoryTypeName = ((String) entry.getKey()).trim();
21                 String[] factoryImplementationNames =
22                     StringUtils.commaDelimitedListToStringArray((String)
23                         entry.getValue());
24                 for (String factoryImplementationName :
25                     factoryImplementationNames) {
26                     result.computeIfAbsent(factoryTypeName, key -> new
27                         ArrayList<>())
28                         .add(factoryImplementationName.trim());
29                 }
30             }
31         }
32     }
```

```

26         }
27
28     // Replace all lists with unmodifiable lists containing unique
29     // elements
30     result.replaceAll((factoryType, implementations) ->
31     implementations.stream().distinct()
32
33     .collect(Collectors.collectingAndThen(Collectors.toList(),
34     Collections::unmodifiableList)));
35     cache.put(classLoader, result);
36   }
37   catch (IOException ex) {
38     throw new IllegalArgumentException("Unable to load factories from
location [" +
39                               FACTORIES_RESOURCE_LOCATION +
40                           "]", ex);
41   }
42   return result;
43 }

```

spring.factories

从这里获取系统文件(自动配置核心): `"META-INF/spring.factories"`

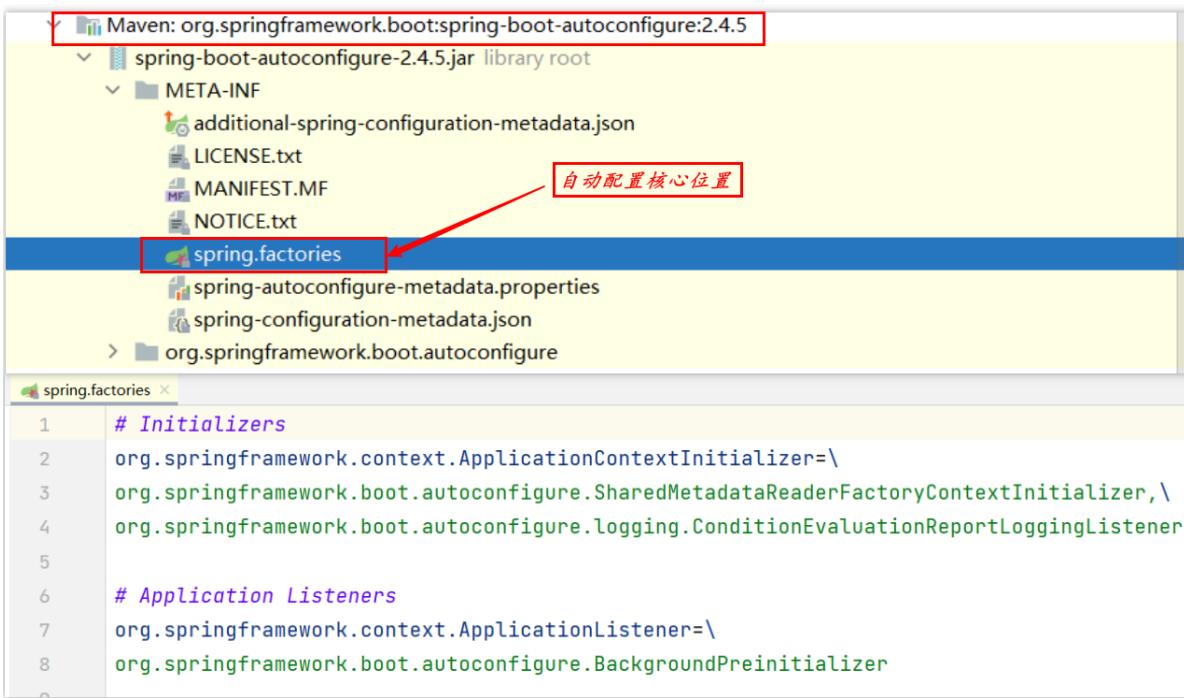
根据源码打开spring.factories, 看到了很多自动配置的文件, 这就是自动配置根源所在!

```

135     private static Map<String, List<String>> loadSpringFactories(ClassLoader classLoader) {
136       Map<String, List<String>> result = cache.get(classLoader);
137       if (result != null) {...}
138
139       result = new HashMap<>();
140       try {
141         Enumeration<URL> urls = classLoader.getResources(FACTORIES_RESOURCE_LOCATION);
142         while (urls.hasMoreElements()) {
143           URL url = urls.nextElement();
144           UrlResource resource = new UrlResource(url);
145           Properties properties = PropertiesLoaderUtils.loadProperties(resource);
146           for (Map.Entry<?, ?> entry : properties.entrySet()) {
147             String factoryTypeName = ((String) entry.getKey()).trim();
148             String[] factoryImplementationNames =
149               StringUtils.commaDelimitedListToStringArray((String) entry.getValue());
150             for (String factoryImplementationName : factoryImplementationNames) {...}
151           }
152         }
153       }
154     }
155
156     /**
157      * @author Arjen Poutsma
158      * @author Juergen Hoeller
159      * @author Sam Brannen
160      * @since 3.2
161     */
162     public final class SpringFactoriesLoader {
163
164       /**
165        * The location to look for factories.
166        * <p>Can be present in multiple JAR files.
167        */
168       public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";
169
170       private static final Log logger = LoggerFactory.getLog(SpringFactoriesLoader.class);
171
172       static final Map<ClassLoader, Map<String, List<String>>> cache = new ConcurrentHashMap<>();
173     }

```

通过该文件找到相对应的jar包，从jar包中找到Springboot自动配置核心文件！！！



4.2.5 WebMvcAutoConfiguration

随机在自动配置类打开看看，比如选择WebMvcAutoConfiguration类。

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
@ConditionalOnMissingBean(WebMvcConfigurerSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class, TaskExecutionAutoConfiguration.class,
        ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {

    /**
     * The default Spring MVC view prefix.
     */
    public static final String DEFAULT_PREFIX = "";

    /**
     * The default Spring MVC view suffix.
     */
    public static final String DEFAULT_SUFFIX = "";

    private static final String SERVLET_LOCATION = "/";

    @Bean
    @ConditionalOnMissingBean(HiddenHttpMethodFilter.class)
    @ConditionalOnProperty(prefix = "spring.mvc.hiddenmethod.filter", name = "enabled", matchIfMissing = false)
    public OrderedHiddenHttpMethodFilter hiddenHttpMethodFilter() { return new OrderedHiddenHttpMethodFilter(); }

    @Bean
    @ConditionalOnMissingBean(FormContentFilter.class)
    @ConditionalOnProperty(prefix = "spring.mvc.formcontent.filter", name = "enabled", matchIfMissing = true)
    public OrderedFormContentFilter formContentFilter() { return new OrderedFormContentFilter(); }
}
```

可以看到这些一个个的都是JavaConfig配置类，而且都注入了一些Bean，自动配置真正实现是从classpath中搜寻所有的 META-INF/spring.factories 配置文件，并将其中对应的 org.springframework.boot.autoconfigure 包下的配置项。

通过反射实例化为对应标注了@Configuration的JavaConfig形式的IOC容器配置类，然后将这些都汇总成为一个实例并加载到IOC容器中。

注意事项：有些自动配置为啥有的没有生效，需要导入相应的start才能有作用？

核心注解：@ConditionalOnXXX，如果这里面的条件都满足，才会生效！！！

4.2.6 小结

SpringBoot所自动配置都是在启动的时候扫描并且加载 `spring.factories` 所有的自动配置类都在这里面，但是不一定生效。要判断条件是否生效，只要导入了对应的start，就有对应的启动器了就有对应的启动器了，自动装配就会生效，然后就配置成功！！！

- Springboot在启动的时候，从类路径之下 `/META-INF/spring.factories` 获取指定的值，将这些自动配置的类导入容器，自动配置就会生效，帮我们进行自动配置！
- 以前我们需要自动配置的东西，现在由Springboot帮助我们做了，整合JavaEE解决方案和自动配置的东西都在 `spring-boot-autoconfigure-2.4.5.jar` 这个包下面。它会把所有的需要导入的组件，以类名的方式返回，这些组件就会被添加到容器。
- 容器中也会存在非常多的xxxAutoConfiguration的文件(@Bean)，就是这些类给容器中导入了这个场景需要的所有组件，并且自动配置(@Configuration, JavaConfig)！！有了自动配置类，免去了我们手动编写配置文件的工作。

4.3 主启动类运行

最初以为就是运行了一个main方法，但是真实的情况是开启了服务。

```
1 package cn.guardwhy;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class SpringbootDemo01Application {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringbootDemo01Application.class, args);
11     }
12 }
```

主启动类运行主要分为两个方面：一部分是 `SpringApplication` 的实例化，二是 `run` 方法的执行。

4.3.1 `SpringApplication` 实例化

- 推断应用的类型是普通的项目还是Web项目。
查找并加载所有可用初始化器，设置到 `initializers` 属性中。
- 找出所有的应用程序监听器，设置到 `listeners` 属性中。
- 推断并设置 `main` 方法的定义类，找到运行的主类。

查看 `SpringApplication` 的构造器

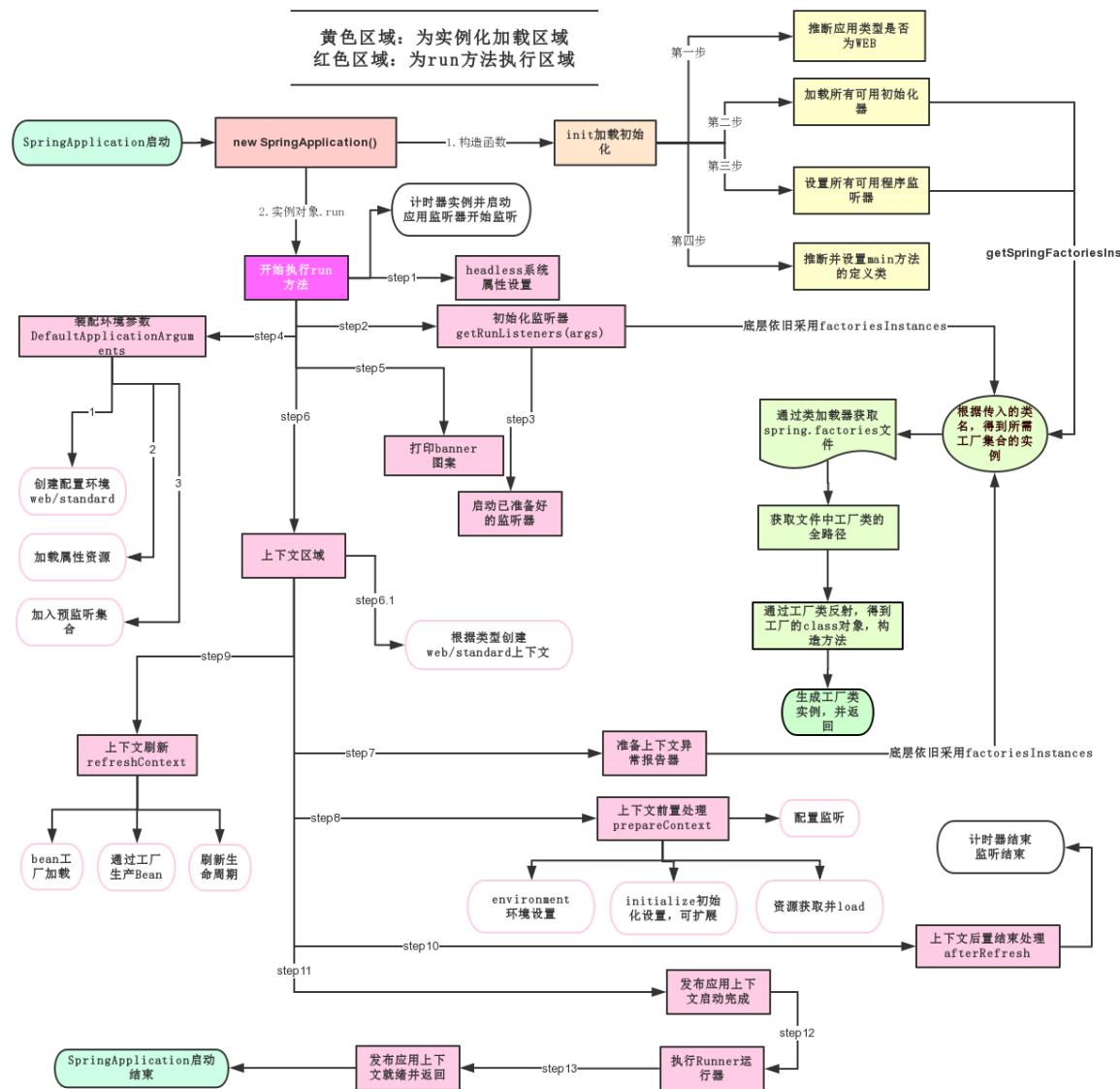
```
1 public SpringApplication(ResourceLoader resourceLoader, Class<?>...
2 primarySources) {
3     this.sources = new LinkedHashSet();
4     this.bannerMode = Mode.CONSOLE;
5     this.logStartupInfo = true;
6     this.addCommandLineProperties = true;
7     this.addConversionService = true;
8     this.headless = true;
9     this.registerShutdownHook = true;
10    this.additionalProfiles = Collections.emptySet();
11    this.isCustomEnvironment = false;
```

```

11     this.lazyInitialization = false;
12     this.applicationContextFactory = ApplicationContextFactory.DEFAULT;
13     this.applicationStartup = ApplicationStartup.DEFAULT;
14     this.resourceLoader = resourceLoader;
15     Assert.notNull(primarySources, "PrimarySources must not be null");
16     this.primarySources = new LinkedHashSet(Arrays.asList(primarySources));
17     this.webApplicationType = WebApplicationType.deduceFromClasspath();
18     this.bootstrapRegistryInitializers =
19       this.getBootstrapRegistryInitializersFromSpringFactories();
20
21     this.setInitializers(this.getSpringFactoriesInstances(ApplicationContextInitializer.class));
22     this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.class));
23     this.mainApplicationClass = this.deduceMainApplicationClass();
24   }

```

4.3.2 run方法的执行



4.4 分析自动配置原理

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

```
1 package org.springframework.boot.autoconfigure.web.servlet;
2
3 import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
4 import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
5 import
6 org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
7 import
8 org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
9 import
10 org.springframework.boot.autoconfigure.condition.ConditionalOnWebApplication
11 ;
12 import org.springframework.boot.autoconfigure.web.ServerProperties;
13 import
14 org.springframework.boot.context.properties.EnableConfigurationProperties;
15 import org.springframework.boot.web.server.WebServerFactoryCustomizer;
16 import
17 org.springframework.boot.web.servlet.filter.OrderedCharacterEncodingFilter;
18 import
19 org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
20 import org.springframework.boot.web.servlet.server.Encoding;
21 import org.springframework.context.annotation.Bean;
22 import org.springframework.context.annotation.Configuration;
23 import org.springframework.core.Ordered;
24 import org.springframework.web.filter.CharacterEncodingFilter;
25
26 // 表示这是一个配置类， 也可以给容器中添加组件。
27 @Configuration(proxyBeanMethods = false)
28 // 将配置文件中对应的值和ServerProperties绑定起来，并把ServerProperties加入到ioc容器中
29 @EnableConfigurationProperties(ServerProperties.class)
30
31 // Spring底层@Conditional注解：根据不同的条件判断，如果满足指定的条件，整个配置类里面的配置就会生效。
32 // 判断当前应用是否是web应用， 如果是当前配置类生效。
33 @ConditionalOnWebApplication(type =
34 ConditionalOnWebApplication.Type.SERVLET)
35
36 // 判断当前项目有没有这个类CharacterEncodingFilter，SpringMVC中进行乱码解决的过滤器。
37 @ConditionalOnClass(CharacterEncodingFilter.class)
38
39 /*
40     判断配置文件中是否存在某个配置： server.servlet.encoding
41     如果不存在，判断也是成立的，即使我们配置文件中不配置server.servlet.encoding =
42     true， 也是默认生效的。
43 */
44 @ConditionalOnProperty(prefix = "server.servlet.encoding", value =
45 "enabled", matchIfMissing = true)
46 public class HttpEncodingAutoConfiguration {
47     // 它已经和SpringBoot的配置文件映射了
48     private final Encoding properties;
49     // 只有一个有参构造器的情况下，参数的值就会从容器中拿
```

```
41     public HttpEncodingAutoConfiguration(ServerProperties properties) {
42         this.properties = properties.getServlet().getEncoding();
43     }
44
45     //给容器中添加一个组件，这个组件的某些值需要从properties中获取
46     @Bean
47     @ConditionalOnMissingBean // 判断容器有没有这个组件？
48     public CharacterEncodingFilter characterEncodingFilter() {
49         CharacterEncodingFilter filter = new
50         OrderedCharacterEncodingFilter();
51         filter.setEncoding(this.properties.getCharset().name());
52
53         filter.setForceRequestEncoding(this.properties.shouldForce(Encoding.Type.REQUEST));
54
55         filter.setForceResponseEncoding(this.properties.shouldForce(Encoding.Type.RESPONSE));
56         return filter;
57     }
58
59     @Bean
60     public LocaleCharsetMappingsCustomizer localeCharsetMappingsCustomizer() {
61         return new LocaleCharsetMappingsCustomizer(this.properties);
62     }
63
64     static class LocaleCharsetMappingsCustomizer
65         implements
66         WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>, Ordered {
67
68         private final Encoding properties;
69
70         LocaleCharsetMappingsCustomizer(Encoding properties) {
71             this.properties = properties;
72         }
73
74         @Override
75         public void customize(ConfigurableServletWebServerFactory factory) {
76             if (this.properties.getMapping() != null) {
77
78                 factory.setLocaleCharsetMappings(this.properties.getMapping());
79             }
80         }
81
82     }
83 }
```

根据当前不同的条件判断，决定这个配置类是否生效。

```

43     @Configuration(proxyBeanMethods = false)
44     @EnableConfigurationProperties(ServerProperties.class)
45     @ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.SERVLET)
46     @ConditionalOnClass(CharacterEncodingFilter.class)
47     @ConditionalOnProperty(prefix = "server.servlet.encoding", value = "enabled", matchIfMissing = true)
48     public class HttpEncodingAutoConfiguration {
49
50         private final Encoding properties;
51
52     @Bean
53     public HttpEncodingAutoConfiguration(ServerProperties properties) {
54         this.properties = properties.getServlet().getEncoding();
55     }
56
57     @Bean
58     @ConditionalOnMissingBean
59     public CharacterEncodingFilter characterEncodingFilter() {
60         CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();
61         filter.setEncoding(this.properties.getCharset().name());
62         filter.setForceRequestEncoding(this.properties.shouldForce(Encoding.Type.REQUEST));
63         filter.setForceResponseEncoding(this.properties.shouldForce(Encoding.Type.RESPONSE));
64         return filter;
65     }

```

- 当这个配置类生效，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是和配置文件进行深度绑定的。
- 所有在配置文件中能配置的属性都是在 (?)Properties 类中封装着，配置文件能配置什么就可以参照某个功能对应的这个属性类。

ServerProperties.java

```

    @ConfigurationProperties for a web server (e.g. port and path settings).
    Since: 1.0.0
    Author: Dave Syer, Stephane Nicoll, Andy Wilkinson, Ivan Sopov, Marcos Barbero, Eddú Meléndez,
    Quinten De Swae, Venil Noronha, Aurélien Leboulanger, Brian Clozel, Olivier Lamy, Chentao Qu,
    Artiom Yudovin, Andrew McGhie, Rafulullah Hamedy, Dirk Deyne, Haifao Zhang, Victor
    Mandujano, Chris Bono
    @ConfigurationProperties(prefix = "server", ignoreUnknownFields = true)
    public class ServerProperties {
        ...
    }

```

application.yaml

```

server:
  encoding:
    charset: Charset
    enabled: Boolean
    force: Boolean
    force-request: Boolean
    force-response: Boolean
    mapping: Map<Locale, String>

```

4.4.1 自动装配小结

- SpringBoot启动会加载大量的自动配置类，查看需要的功能有没有在SpringBoot默认写好的自动配置类当中。
- 查看自动配置类中到底配置了哪些组件(如果该组件存在要我们要用的组件存在，则无需手动配置)，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。只需要在配置文件中指定这些属性的值即可。
- (?)AutoConfigurartion 是自动配置类，给容器中添加组件。 (?) Properties封装配置文件中相关属性，可以通过Springboot配置 (.yaml) 修改属性！！

4.4.2 @Conditional

@Conditional派生注解 (Spring注解版原生的@Conditional作用)

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。

@Conditional扩展注解	作用 (判断是否满足当前指定条件)
@ConditionalOnJava	系统的java版本是否符合要求
@ConditionalOnBean	容器中存在指定Bean ;
@ConditionalOnMissingBean	容器中不存在指定Bean ;
@ConditionalOnExpression	满足SpEL表达式指定
@ConditionalOnClass	系统中有指定的类
@ConditionalOnMissingClass	系统中没有指定的类
@ConditionalOnSingleCandidate	容器中只有一个指定的Bean , 或者这个Bean是首选Bean
@ConditionalOnProperty	系统中指定的属性是否有指定的值
@ConditionalOnResource	类路径下是否存在指定资源文件
@ConditionalOnWebApplication	当前是web环境
@ConditionalOnNotWebApplication	当前不是web环境
@ConditionalOnJndi	JNDI存在指定项

注意点

自动配置类必须在一定的条件下才能生效，可以通过启用 `debug=true` 属性，让控制台打印自动配置报告，这样就可以知道哪些自动配置类生效。

```
1 # 可以通过 debug=true 来查看，哪些自动配置类生效，哪些没有生效
2 debug: true
```

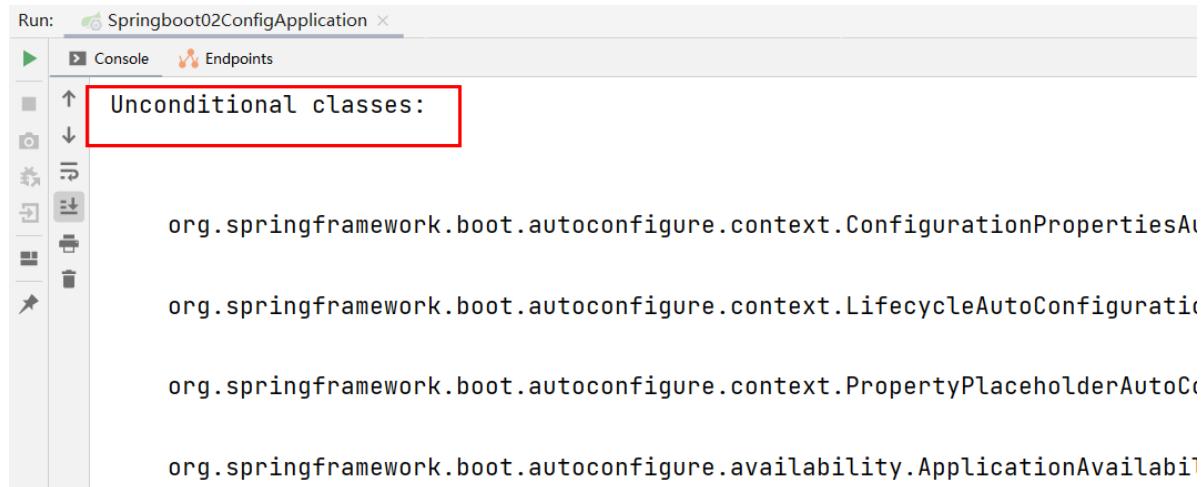
Positive matches: (自动配置类启用的：正匹配)

```
Positive matches:
-----
AopAutoConfiguration matched:
- @ConditionalOnProperty (spring.aop.auto=true) matched (OnPropertyCondition)
```

Negative matches: (没有启动，没有匹配成功的自动配置类：负匹配)

```
Negative matches:
-----
ActiveMQAutoConfiguration:
Did not match:
- @ConditionalOnClass did not find required class 'javax.jms.ConnectionFactory' (OnClassCondition)
```

Unconditional classes: (没有条件的类)



```
Run: Springboot02ConfigApplication <-->
  Console Endpoints
  ↑ Unconditional classes:
  ↓

org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration
org.springframework.boot.autoconfigure.context.LifecycleAutoConfiguration
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration
org.springframework.boot.autoconfigure.availability.ApplicationAvailabilityAutoConfiguration
```

5-注入配置文件

5.1 基本概述

SpringBoot使用一个全局的配置文件， 配置文件名称是固定的。

- application.properties

```
1 | 语义结构 : key=value
```

- application.yml

```
1 | 语义结构 : key: 空格 value
```

配置文件的作用

修改SpringBoot自动配置的默认值，因为SpringBoot在底层都自动配置好了。

5.2 yaml 基本概述

- YAML文件格式是Spring Boot支持的一种JSON文件格式，相较于传统的xml配置文件，YAML文件以数据为核心，是一种更为直观且容易被电脑识别的数据序列化格式。
- application.yaml配置文件的工作原理和application.properties是一样的，只不过yaml格式配置文件看起来更简洁一些。
- YAML文件的扩展名可以使用.yml或者.yaml。application.yaml文件使用“key: (空格) value”格式配置属性，使用缩进控制层级关系。

传统properties配置：

```
1 | <server>
2 |   <port>8089</port>
3 | </server>
```

yaml配置：

```
1 | server:
2 |   prot: 8081
```

5.2.1 yaml基础语法

注意：yaml语法要求极其严格！！！

1. 空格不能省略。
2. 以缩进来控制层级关系，只要是左边对齐的一列数据都是同一个层级的。
3. 属性和值的大小写都是十分敏感的。

普通值 [数字, 布尔值, 字符串]

字面量直接写在后面就可以，字符串默认不用加上双引号或者单引号。

```
1 | k: v
```

注意点：

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。

```
1 | 案例: name: "james \n harden" 输出: james 换行 harden
```

单引号('')会转义特殊字符，特殊字符最终会变成和普通字符一样输出。

```
1 | 案例: name: 'james \n harden' 输出: james \n harden
```

对象、map(键值对)

语法格式

```
1 # 对象、集合格式
2 k:
3   v1:
4   v2:
```

案例说明

```
1 # 对象写法
2 stu:
3   name: guardwhy
4   age: 28
```

行内写法

```
1 # 行内写法
2 student: {name: guardwhy, age: 21}
```

数组(List、set)

语法格式：用 - 值表示数组中的一个元素

```
1 # 数组
2 start:
3   - kobe
4   - james
5   - curry
```

行内写法

```
1 | starts: [kobe, harden, curry]
```

修改SpringBoot端口号

```
1 | server:  
2 |     port: 8081
```

5.3 注入配置文件

yaml文件更强大的地方在于，它可以给我们的实体类直接注入匹配值！！！

5.3.1 yaml方式

编写实体类 Student

```
1 | package cn.guardwhy.domain;  
2 |  
3 | import lombok.AllArgsConstructorConstructor;  
4 | import lombok.Data;  
5 | import lombok.NoArgsConstructorConstructor;  
6 | import org.springframework.stereotype.Component;  
7 |  
8 | @Data  
9 | @AllArgsConstructorConstructor  
10 | @NoArgsConstructorConstructor  
11 | // 学生类  
12 | @Component  
13 | public class Student {  
14 |     private String username;  
15 |     private Integer usage;  
16 | }
```

编写实体类 Person

@ConfigurationProperties：默认从全局配置文件中获取值

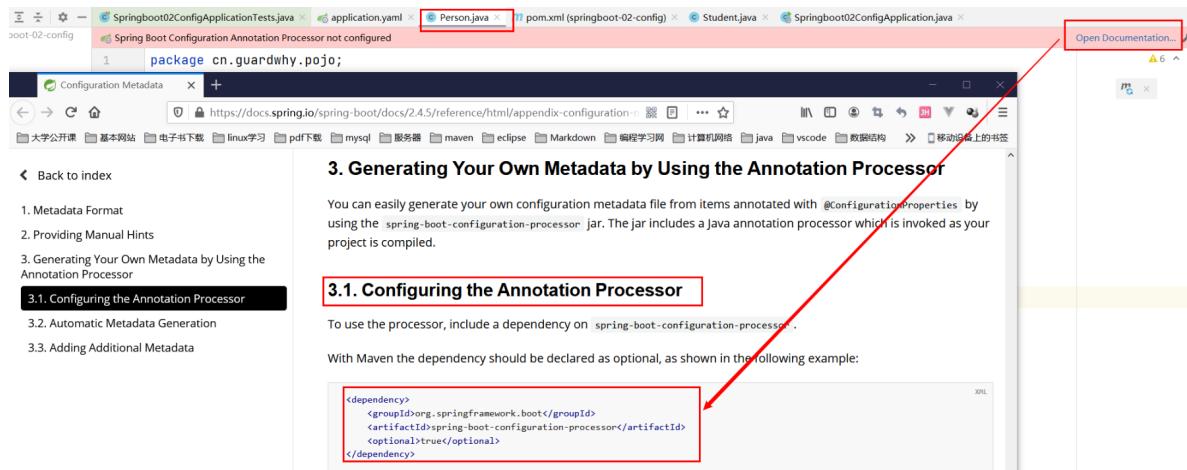
```
1 | package cn.guardwhy.domain;  
2 |  
3 | import lombok.AllArgsConstructorConstructor;  
4 | import lombok.Data;  
5 | import lombok.NoArgsConstructorConstructor;  
6 | import org.springframework.boot.context.properties.ConfigurationProperties;  
7 | import org.springframework.stereotype.Component;  
8 |  
9 | import java.util.Date;  
10 | import java.util.List;  
11 | import java.util.Map;  
12 |  
13 | // Person类  
14 | @Component  
15 | @ConfigurationProperties(prefix = "person")  
16 | @Data  
17 | @AllArgsConstructorConstructor
```

```

18 @NoArgsConstructor
19 public class Person {
20     private String name;
21     private Integer age;
22     private Date birthday;
23     private Map<String, Object> maps;
24     private List<Object> lists;
25     // 导入学生类
26     private Student student;
27 }

```

IDEA 提示，springboot配置注解处理器没有找到，在 pom.xml 中添加以下依赖！！！



```

1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-configuration-processor</artifactId>
4     <optional>true</optional>
5 </dependency>

```

application.yaml

```

1 person:
2     name: guardwhy
3     age: 27
4     birthday: 1993/06/19
5     maps: {k1: v1, k2: v2}
6     lists:
7         - running
8         - code
9         - study
10    student:
11        username: Curry
12        userage: 10

```

测试案例

```

1 package cn.guardwhy;
2
3 import cn.guardwhy.domain.Person;
4 import org.junit.jupiter.api.Test;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.boot.test.context.SpringBootTest;

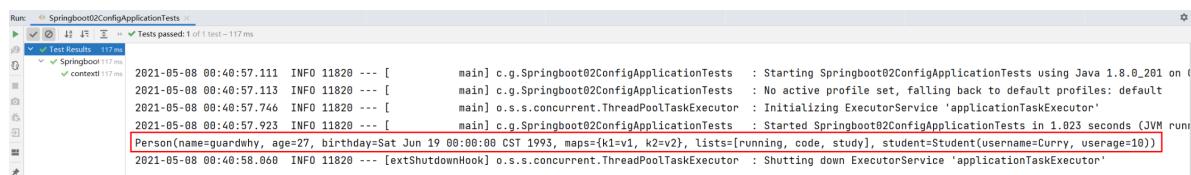
```

```

7
8 @SpringBootTest
9 class Springboot02ConfigApplicationTests {
10
11     // 注入数据
12     @Autowired
13     private Person person;
14
15     @Test
16     void contextLoads() {
17         System.out.println(person);
18     }
19 }

```

5.3.2 执行结果



5.3.2 properties方式

编写实体类 Person

```

1 package cn.guardwhy.domain;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6 import org.springframework.beans.factory.annotation.Value;
7 import org.springframework.boot.context.properties.ConfigurationProperties;
8 import org.springframework.context.annotation.PropertySource;
9 import org.springframework.stereotype.Component;
10
11 import java.util.Date;
12 import java.util.List;
13 import java.util.Map;
14
15 // Person类
16 @Component
17 @Data
18 @AllArgsConstructor
19 @NoArgsConstructor
20 // 加载指定的配置文件
21 @PropertySource(value = "classpath:Person.properties")
22 public class Person {
23     // SPEL表达式取出配置文件的值
24     @Value("${name}")
25     private String name;
26     @Value("${age}")
27     private Integer age;
28     private Date birthday;
29     private Map<String, Object> maps;
30     private List<Object> lists;
31     // 导入学生类
32     private Student student;

```

加载指定配置文件

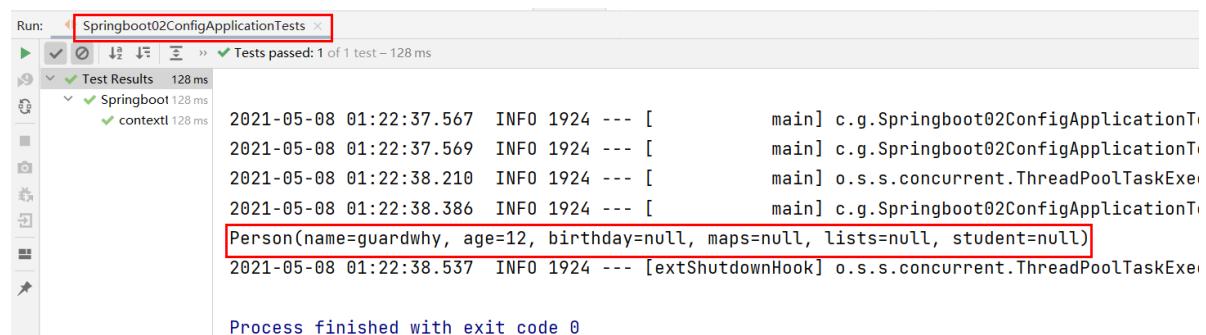
新建一个person.properties文件

```
1 | name=guardwhy
2 | age=12
```

测试案例

```
1 package cn.guardwhy;
2
3 import cn.guardwhy.domain.Person;
4 import org.junit.jupiter.api.Test;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.boot.test.context.SpringBootTest;
7
8 @SpringBootTest
9 class Springboot02ConfigApplicationTests {
10
11     // 注入数据
12     @Autowired
13     private Person person;
14
15     @Test
16     void contextLoads() {
17         System.out.println(person);
18     }
19 }
```

5.3.3 执行结果



5.3.4 配置文件占位符

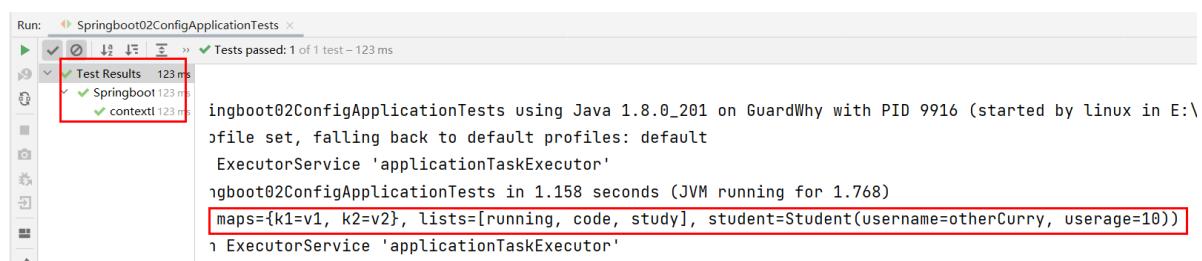
application.yaml

```

1 person:
2   name: guardwhy${random.uuid} # 随机uuid
3   age: ${random.int} # 随机int
4   birthday: 1993/06/19
5   maps: {k1: v1, k2: v2}
6   lists:
7     - running
8     - code
9     - study
10  student:
11    # 引用student.student的值, 如果不存在就用:后面的值, 即other, 然后拼接上curry
12    username: ${student.test:other}Curry
13    userage: 10

```

执行结果



小结

- 配置yaml和配置properties都可以获取到值，但是yaml更好使用。
- 如果在某个业务中，只需要获取配置文件中的某个值，可以使用一下 @value。
- 如果说专门编写了一个JavaBean来和配置文件进行一一映射，就直接@ConfigurationProperties。

5.4 JSR303数据校验

Springboot中可以用@validated来校验数据，如果数据异常则会统一抛出异常，方便异常中心统一处理。

5.4.1 实体类 Person

```

1 package cn.guardwhy.domain;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6 import org.springframework.beans.factory.annotation.Value;
7 import org.springframework.boot.context.properties.ConfigurationProperties;
8 import org.springframework.context.annotation.PropertySource;
9 import org.springframework.stereotype.Component;
10 import org.springframework.validation.annotation.Validated;
11
12 import javax.validation.constraints.Email;
13 import java.util.Date;
14 import java.util.List;
15 import java.util.Map;
16
17 // Person类
18 @Component
19 @ConfigurationProperties(prefix = "person")
20 @Data

```

```

21  @AllArgsConstructor
22  @NoArgsConstructor
23  // 数据校验
24  @Validated
25  public class Person {
26      @Email(message="用户名格式不合法!!!")
27      private String name;
28      private Integer age;
29      private Date birthday;
30      private Map<String, Object> maps;
31      private List<Object> lists;
32      // 导入学生类
33      private Student student;
34  }

```

5.4.2 pom.xml

添加Validated相关依赖

```

1  <dependency>
2      <groupId>org.springframework.boot</groupId>
3      <artifactId>spring-boot-starter-validation</artifactId>
4  </dependency>

```

5.4.3 运行结果

运行结果： default message [用户名格式不合法!!];

```

r.k.boot.context.properties.bind.validation.BindValidationException: Binding validation errors on person
person' on field 'name': rejected value [guardwhy]; codes [Email.person.name,Email.name,Email.java.lang.String,Email]; arguments [org.springframework.boot.context.properties.bind.ValidationBindHandler.validateAndPush(ValidationBindHandler.java:141)

```

使用数据校验，可以保证数据的正确性

```

1  @NotNull(message="名字不能为空")
2  private String userName;
3  @Max(value=120,message="年龄最大不能超过120")
4  private int age;
5  @Email(message="邮箱格式错误")
6  private String email;
7  空检查
8  @Null    验证对象是否为null
9  @NotNull  验证对象是否不为null，无法查检长度为0的字符串
10 @NotBlank 检查约束字符串是不是Null还有被Trim的长度是否大于0,只对字符串,且会去掉前后空格。
12 @NotEmpty 检查约束元素是否为NULL或者是EMPTY.
13
14 Boolean检查
15 @AssertTrue  验证 Boolean 对象是否为 true
16 @AssertFalse 验证 Boolean 对象是否为 false
17
18 长度检查
19 @Size(min=, max=) 验证对象 (Array,Collection,Map,String) 长度是否在给定的范围之内
20 @Length(min=, max=) string is between min and max included.
21 日期检查
22 @Past    验证 Date 和 Calendar 对象是否在当前时间之前
23 @Future  验证 Date 和 Calendar 对象是否在当前时间之后

```

```
24 @Pattern 验证 String 对象是否符合正则表达式的规则  
25 .....等等  
26 除此以外，我们还可以自定义一些数据校验规则
```

5.5 多环境切换

5.5.1 properties多配置文件

在主配置文件编写的时候，文件名可以是 `application-{profile}.properties/yaml`，用来指定多个环境版本。

比如说可以设定：`application-test.properties` 代表测试环境配置，`application-dev.properties` 代表开发环境配置。但是Springboot并不会直接启动这些配置文件，**它默认使用`application.properties`主配置文件。**

需要通过一个配置来选择需要激活的环境：

`application.properties`

```
1 #比如在配置文件中指定使用test环境，可以通过设置不同的端口号进行测试。  
2 #启动SpringBoot，就可以看到已经切换到test下的配置。  
3 spring.profiles.active=test
```

5.5.2 yml多配置文件

和properties配置文件中一样，但是使用yaml去实现不需要创建多个配置文件！

`application.yaml`

```
1 server:  
2   port: 8081  
3 # 选择要激活哪个环境  
4 spring:  
5   profiles:  
6     active: dev  
7  
8 ---  
9 server:  
10  port: 8082  
11 spring:  
12   # 配置开发环境名称  
13   profiles: dev  
14  
15 ---  
16 server:  
17   # 配置测试环境名称  
18   port: 8083  
19 spring:  
20   profiles: test
```

注意：如果yml和properties同时都配置了端口，并且没有激活其他环境， 默认会使用properties配置文件的！

5.5.3 配置文件加载位置

外部加载配置文件的方式十分多，我们选择最常用的即可，在开发的资源文件中进行配置！

参考文档: <https://docs.spring.io/spring-boot/docs/2.2.5.RELEASE/reference/htmlsingle>

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！！！

优先级由高到底，高优先级的配置会覆盖低优先级的配置，优先级排名顺序如下：

项目路径下的 config 文件夹配置文件 > 项目路径下配置文件 > 资源路径下的 config 文件夹配置文件 > 资源路径下配置文件

6- SpringBoot Web 开发

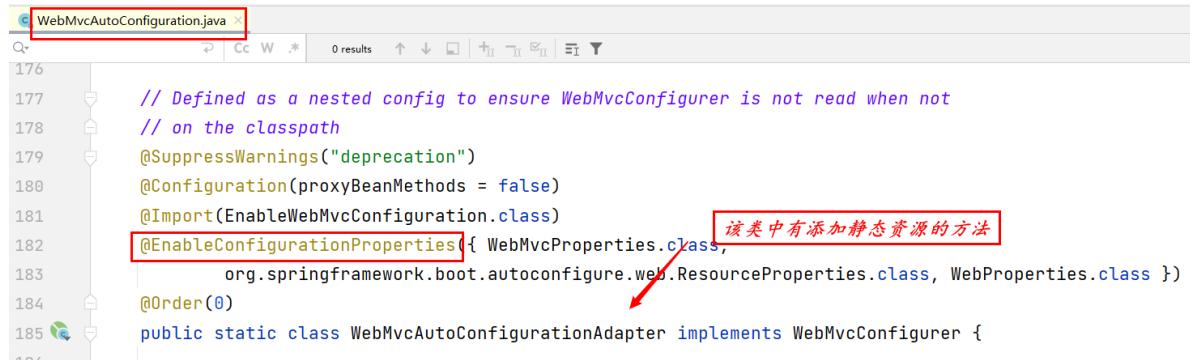
6.1 静态资源处理

SpringBoot 中，SpringMVC 的 web 配置都在 WebMvcAutoConfiguration 这个配置类里面，其中有一个方法： addResourceHandlers 添加资源处理

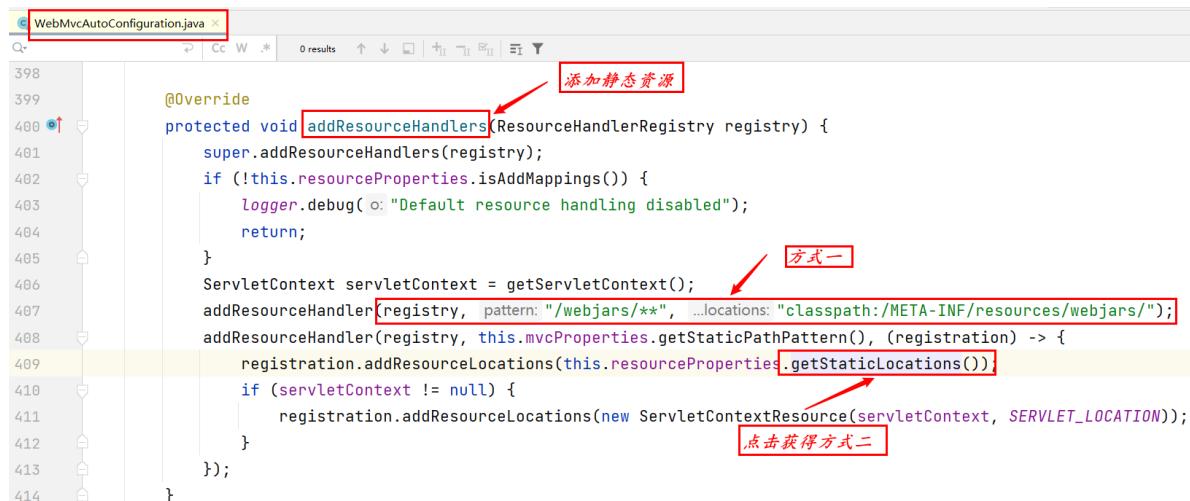
源码分析

Springboot 主要帮我们配置了什么？

[(?)AutoConfiguartion: 向容器中自动配置组件， (?) Properties: 自动配置类，装配配置文件中自定义的一些内容！！]



```
176
177     // Defined as a nested config to ensure WebMvcConfigurer is not read when not
178     // on the classpath
179     @SuppressWarnings("deprecation")
180     @Configuration(proxyBeanMethods = false)
181     @Import(EnableWebMvcConfiguration.class)
182     @EnableConfigurationProperties({ WebMvcProperties.class,
183                                     org.springframework.boot.autoconfigure.web.ResourceProperties.class, WebProperties.class })
184     @Order(0)
185     public static class WebMvcAutoConfigurationAdapter implements WebMvcConfigurer {
```



```
398
399
400     @Override
401     protected void addResourceHandlers(ResourceHandlerRegistry registry) {
402         super.addResourceHandlers(registry);
403         if (!this.resourceProperties.isAddMappings()) {
404             logger.debug("Default resource handling disabled");
405             return;
406         }
407         ServletContext servletContext = getServletContext();
408         addResourceHandler(registry, pattern: "/webjars/**", ...locations: "classpath:/META-INF/resources/webjars/");
409         addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(), (registration) -> {
410             registration.addResourceLocations(this.resourceProperties.getStaticLocations());
411             if (servletContext != null) {
412                 registration.addResourceLocations(new ServletContextResource(servletContext, SERVLET_LOCATION));
413             }
414         });
415     }
```

```
107
108     public String[] getStaticLocations() {
109         return this.staticLocations; 1
110     }
111

e WebProperties.java x
66
67     public Resources getResources() { return this.resources; }
68
69
70
71     public enum LocaleResolver {...} 3
72
73
74
75
76     public static class Resources {
77
78         private static final String[] CLASSPATH_RESOURCE_LOCATIONS = { "classpath:/META-INF/resources/",
79             "classpath:/resources/", "classpath:/static/", "classpath:/public/" };
80
81
82
83
84
85
86
87
88
89
90

Locations of static resources. Defaults to classpath:/META-INF/resources/, /resources/
/static/, /public/.

95
private String[] staticLocations = CLASSPATH_RESOURCE_LOCATIONS; 2
```

小结

- 在Springboot中可以使用以下方式处理静态资源。
 - webjars 访问方式: `localhost:8080/webjars/`
 - public、static、/**、resources 访问方式: `localhost:8080/`
 - 优先级: resources > static (默认) > public

6.2 首页处理

源码分析

可以看到一个欢迎页的映射，就是我们的首页

The screenshot shows the IntelliJ IDEA code editor with the file `WebMvcAutoConfiguration.java` open. The code defines a `@Bean` named `WelcomePageHandlerMapping`. A red box highlights the method name `WelcomePageHandlerMapping`, and a red arrow points from it to the text "欢迎页面处理映射". Another red box highlights the field reference `this.mvcProperties.getStaticPathPattern()`, and a red arrow points from it to the text "自定义".

```
442 @Bean  
443 public WelcomePageHandlerMapping welcomePageHandlerMapping(ApplicationContext applicationContext,  
444     FormattingConversionService mvcConversionService, ResourceUrlProvider mvcResourceUrlProvider) {  
445     WelcomePageHandlerMapping welcomePageHandlerMapping = new WelcomePageHandlerMapping(  
446         new TemplateAvailabilityProviders(applicationContext), applicationContext, getWelcomePage(),  
447         this.mvcProperties.getStaticPathPattern());  
448     welcomePageHandlerMapping.setInterceptors(getInterceptors(mvcConversionService, mvcResourceUrlProvider));  
449     welcomePageHandlerMapping.setCorsConfigurations(getCorsConfigurations());  
450     return welcomePageHandlerMapping;  
451 }
```

```
485     private Resource getWelcomePage() {
486         for (String location : this.resourceProperties.getStaticLocations()) {
487             Resource indexHtml = getIndexHtml(location);
488             if (indexHtml != null) {
489                 return indexHtml;
490             }
491         }
492         ServletContext servletContext = getServletContext();
493         if (servletContext != null) {
494             return getIndexHtml(new ServletContextResource(servletContext, SERVLET_LOCATION));
495         }
496         return null;
497     }
498
499     private Resource getIndexHtml(String location) {
500         return getIndexHtml(this.resourceLoader.getResource(location));
501     }
502
503     private Resource getIndexHtml(Resource location) {
504         try {
505             Resource resource = location.createRelative("index.html");
506             if (resource.exists() && (resource.getURL() != null)) {
507                 return resource;
508             }
509         } catch (Exception ex) {
510         }
511         return null;
512     }
513 }
```

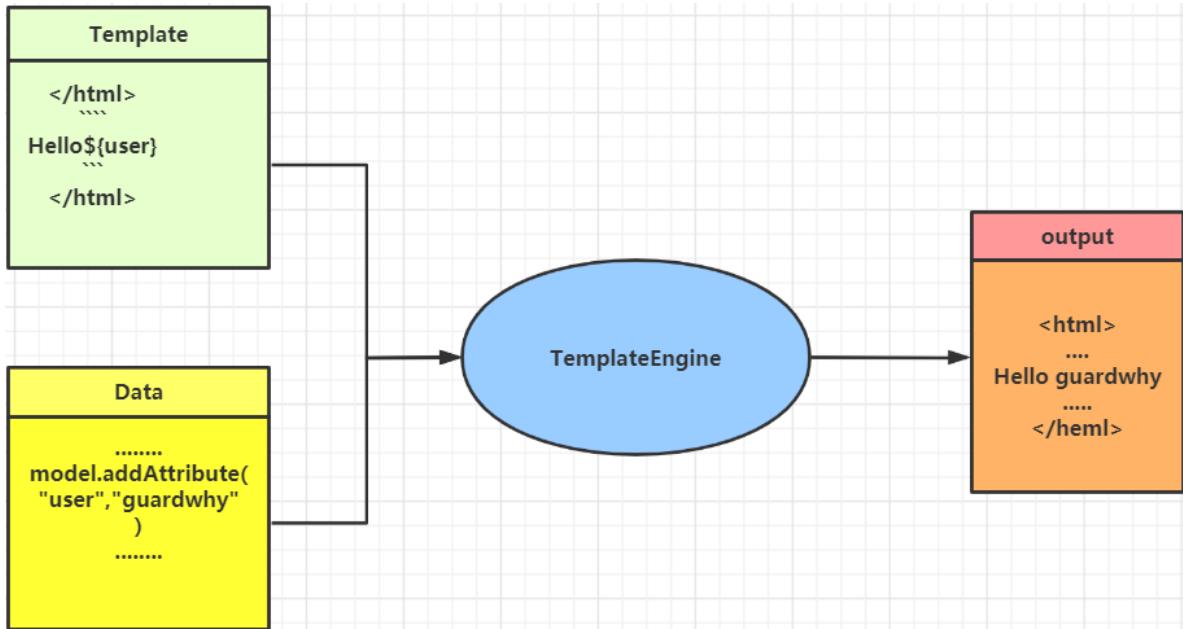
欢迎页，静态资源文件夹下的所有 index.html 页面；被 `/**` 映射。比如访问 <http://localhost:8080/>，就会找静态资源文件夹下的 index.html

新建一个 index.html，在上面的3个目录【public、static、resource】中任意一个；然后访问测试 <http://localhost:8080/> 看结果！

6.3 Thymeleaf

Thymeleaf是一种现代的基于服务器端的Java模板引擎技术，也是一个优秀的面向Java的XML、XHTML、HTML5页面模板，它具有丰富的标签语言、函数和表达式，在使用Spring Boot框架进行页面设计时，一般会选择Thymeleaf模板。

6.3.1 模板引擎



模板引擎的作用就是来写一个页面模板，比如有些值呢，是动态的，我们写一些表达式。而这些值，就是我们在后台封装一些数据。然后把这个模板和这个数据交给我们模板引擎，模板引擎按照我们这个数据帮你把这表达式解析、填充到我们指定的位置，然后把这个数据最终生成一个想要的内容给写出去。

引入Thymeleaf

- Thymeleaf 官网: <https://www.thymeleaf.org/>
- Thymeleaf 在Github 的主页: <https://github.com/thymeleaf/thymeleaf>
- Spring官方文档: 找到对应的版本<https://docs.spring.io/spring-boot/docs/2.4.5/reference/htmlsingle/#using-boot-starter>

找到对应的thymeleaf pom依赖

```

1 <!--thymeleaf版本-->
2 <dependency>
3     <groupId>org.thymeleaf</groupId>
4     <artifactId>thymeleaf-spring5</artifactId>
5 </dependency>
6 <dependency>
7     <groupId>org.thymeleaf.extras</groupId>
8     <artifactId>thymeleaf-extras-java8time</artifactId>
9 </dependency>

```

源码分析

Thymeleaf的自动配置类：ThymeleafProperties

```

1 @ConfigurationProperties(prefix = "spring.thymeleaf")
2 public class ThymeleafProperties {
3
4     private static final Charset DEFAULT_ENCODING = StandardCharsets.UTF_8;
5     // 前缀
6     public static final String DEFAULT_PREFIX = "classpath:/templates/";
7     // 后缀
8     public static final String DEFAULT_SUFFIX = ".html";
9
10    /**
11     * whether to check that the template exists before rendering it.
12
13 }

```

```

12     */
13     private boolean checkTemplate = true;
14
15     /**
16      * whether to check that the templates location exists.
17      */
18     private boolean checkTemplateLocation = true;
19
20     /**
21      * Prefix that gets prepended to view names when building a URL.
22      */
23     private String prefix = DEFAULT_PREFIX;
24
25     /**
26      * Suffix that gets appended to view names when building a URL.
27      */
28     private String suffix = DEFAULT_SUFFIX;
29
30     /**
31      * Template mode to be applied to templates. See also Thymeleaf's
32      * TemplateMode enum.
33      */
34     private String mode = "HTML";
35
36     /**
37      * Template files encoding.
38      */
39     private Charset encoding = DEFAULT_ENCODING;
40
41     /**
42      * Whether to enable template caching.
43      */
44     private boolean cache = true;
45
46     /**
47      * Whether to enable Thymeleaf view resolution for web frameworks.
48      */
49     private boolean enabled = true;
50 }
```

- 只需要把html页面放在类路径下的templates下，thymeleaf就可以自动渲染了。
- 使用thymeleaf什么都不需要配置，只需要将其放在指定的文件夹下。

6.3.2 案例分析

编写一个TestController

```

1 package cn.guardwhy.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.RequestMapping;
6
7 @Controller
8 public class TestController {
9     @RequestMapping("/t1")
10    public String test01(Model model){
```

```
11     model.addAttribute("message", "hello,Thymeleaf!!!");
12     return "test";
13 }
14 }
```

使用thymeleaf，需要在html文件中导入命名空间的约束。

```
1 | xmlns:th="http://www.thymeleaf.org"
```

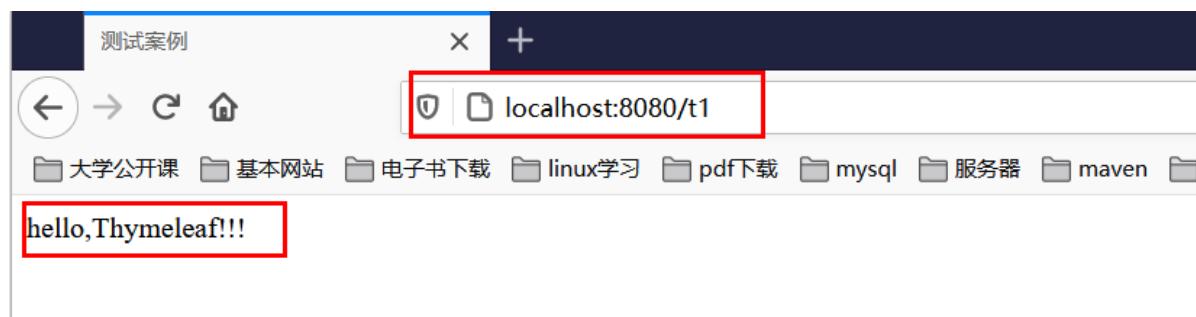
编写前端test.html页面

```
1 <!DOCTYPE html>
2 <html lang="en" xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <meta charset="UTF-8">
5     <title>测试案例</title>
6 </head>
7 <body>
8
9     <!--/*@thymesVar id="msg" type=""*/-->
10
11    <!--所有的html元素都可以被thymeleaf替换接管； th:元素名-->
12    <div th:text="${message}"></div>
13 </body>
14 </html>
```

要在页面内容添加以下格式,不然会报错！！！

```
1 | <!--/*@thymesVar id="msg" type=""*/-->
```

测试结果



6.3.3 Thymeleaf语法

在HTML页面上使用Thymeleaf标签，Thymeleaf 标签能够动态地替换掉静态内容，使页面动态展示。

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

The screenshot shows a browser window displaying the Thymeleaf tutorial at thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#attribute-precedence. The left sidebar contains a navigation menu with sections like '8.4 Removing template fragments', '8.5 Layout Inheritance', '9 Local Variables', '10 Attribute Precedence' (which is highlighted with a red box), '11 Comments and Blocks', '12 Inlining', and '13 JavaScript'. The main content area displays a table titled 'the tag. This order is:' with 9 rows, each representing a feature and its corresponding attributes. Red boxes highlight specific entries: 'Fragment inclusion' (th:insert, th:replace), 'Fragment iteration' (th:each), 'Conditional evaluation' (th:if, th:unless, th:switch, th:case), 'Local variable definition' (th:object, th:with), 'General attribute modification' (th:attr, th:attrprepend, th:attrappend), 'Specific attribute modification' (th:value, th:href, th:src, ...), 'Text (tag body modification)' (th:text, th:utext), 'Fragment specification' (th:fragment), and 'Fragment removal' (th:remove). A note below the table states: 'This precedence mechanism means that the above iteration fragment will give exactly the same results if the attribute position is inverted (although it would be slightly less readable):'.

Thymeleaf的常用标签

th: 标签	说明
th:insert	布局标签，替换内容到引入的文件
th:replace	页面片段包含（类似JSP中的include标签）
th:each	元素遍历（类似JSP中的c:forEach标签）
th:if	条件判断，如果为真
th:unless	条件判断，如果为假
th:switch	条件判断，进行选择性匹配
th:case	条件判断，进行选择性匹配
th:value	属性值修改，指定标签属性值
th:href	用于设定链接地址
th:src	用于设定链接地址
th:text	用于指定标签显示的文本内容

标准表达式

Thymeleaf模板引擎提供了多种标准表达式语法，在正式学习之前，先通过一张表来展示其主要语法及说明。

说明	表达式语法
变量表达式	<code> \${...}</code>
选择变量表达式	<code>*{...}</code>
消息表达式	<code>#{...}</code>
链接URL表达式	<code>@{...}</code>
片段表达式	<code>~{...}</code>

6.3.4 案例测试

编写一个Controller，放一些数据。

```

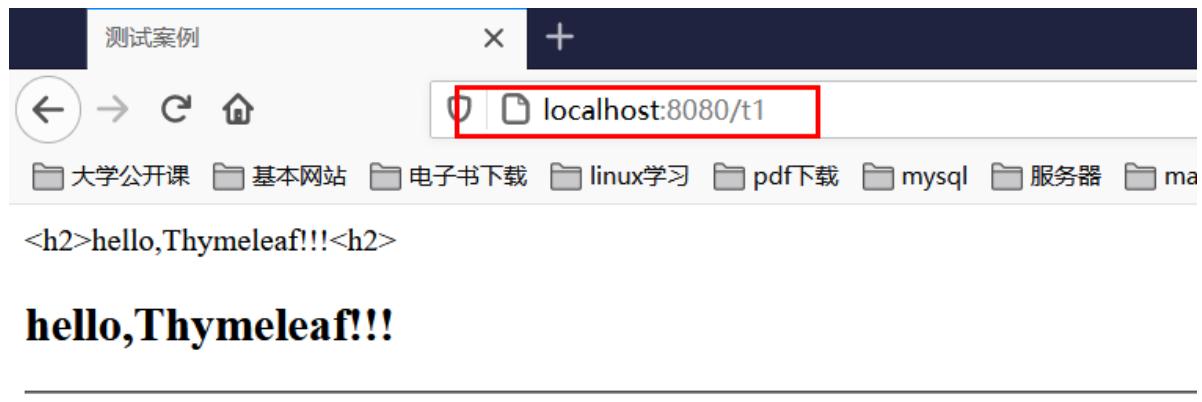
1 package cn.guardwhy.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.RequestMapping;
6
7 import java.util.Arrays;
8
9 @Controller
10 public class TestController {
11     @RequestMapping("/t1")
12     public String test01(Model model){
13         model.addAttribute("message", "<h2>hello,Thymeleaf!!!<h2>");
14         model.addAttribute("lists", Arrays.asList("harden",
15             "james", "kobe"));
16         return "test";
17     }
18 }
```

编写前端test.html页面

```

1 <!DOCTYPE html>
2 <html lang="en" xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <meta charset="UTF-8">
5     <title>测试案例</title>
6 </head>
7 <body>
8
9     <!--/*@thymesVar id="msg" type=""*/-->
10
11     <!--所有的html元素都可以被thymeleaf替换接管； th:元素名-->
12     <div th:text="${message}"></div>
13     <!--不转义-->
14     <div th:utext="${message}"></div>
15
16     <hr/>
17
18     <!--遍历数据-->
19     <h4 th:each="lists:${lists}" th:text="${lists}"></h4>
20 </body>
```

启动项目测试！！！



harden

james

kobe

6.4 SpringMVC自动配置原理

SpringBoot对SpringMVC还做了哪些配置，包括如何扩展，如何定制？

参考文档

3. Using Spring Boot 4. Spring Boot Features 4.1. SpringApplication 4.2. Externalized Configuration 4.3. Profiles 4.4. Logging 4.5. Internationalization 4.6. JSON 4.7. Developing Web Applications 4.7.1. The “Spring Web MVC Framework” Spring MVC Auto-configuration HttpMessageConverters Custom JSON Serializers and Deserializers MessageCodesResolver Static Content Welcome Page Path Matching and Content Negotiation ConfigurableWebBindingInitializer Template Engines Error Handling Spring HATEOAS	<p>Spring MVC Auto-configuration</p> <p>Spring Boot provides auto-configuration for Spring MVC that works well with most applications.</p> <p>The auto-configuration adds the following features on top of Spring's defaults:</p> <ul style="list-style-type: none"> • Inclusion of <code>ContentNegotiatingViewResolver</code> and <code>BeanNameViewResolver</code> beans. • Support for serving static resources, including support for Webjars (covered later in this document). • Automatic registration of <code>Converter</code>, <code>GenericConverter</code>, and <code>Formatter</code> beans. • Support for <code>HttpMessageConverters</code> (covered later in this document). • Automatic registration of <code>MessageCodesResolver</code> (covered later in this document). • Static <code>index.html</code> support. • Automatic use of a <code>ConfigurableWebBindingInitializer</code> bean (covered later in this document). <p>If you want to keep those Spring Boot MVC customizations and make more MVC customizations (interceptors, formatters, view controllers, and other features), you can add your own <code>@Configuration</code> class of type <code>WebMvcConfigurer</code> but without <code>@EnableWebMvc</code>.</p> <p>If you want to provide custom instances of <code>RequestMappingHandlerMapping</code>, <code>RequestMappingHandlerAdapter</code>, or <code>ExceptionHandlerExceptionResolver</code>, and still keep the Spring Boot MVC customizations, you can declare a bean of type <code>WebMvcRegistrations</code> and use it to provide custom instances of those components.</p> <p>If you want to take complete control of Spring MVC, you can add your own <code>@Configuration</code> annotated with <code>@EnableWebMvc</code>, or alternatively add your own <code>@Configuration</code>-annotated <code>DelegatingWebMvcConfiguration</code> as described in the Javadoc of <code>@EnableWebMvc</code>.</p>
--	---

6.4.1 源码分析

ContentNegotiatingViewResolver类

找到 `WebMvcAutoConfiguration`，然后搜索 `ContentNegotiatingViewResolver`，
`ContentNegotiatingViewResolver` 类实现了 `ViewResolver` 接口。

```

1 public class ContentNegotiatingViewResolver extends
2     webApplicationObjectSupport implements ViewResolver, Ordered,
3     InitializingBean {
4     @Nullable
5     private ContentNegotiationManager contentNegotiationManager;

```

```
4     private final ContentNegotiationManagerFactoryBean cnmFactoryBean = new
5     ContentNegotiationManagerFactoryBean();
6
7     private boolean useNotAcceptableStatusCode = false;
8
9     @Nullable
10    private List<View> defaultViews;
11
12    @Nullable
13    private List<ViewResolver> viewResolvers;
14
15    private int order = Ordered.HIGHEST_PRECEDENCE;
16 }
```

ViewResolver接口解析视图

```
1 public interface ViewResolver {
2     @Nullable
3     View resolveViewName(String viewName, Locale locale) throws Exception;
4 }
```

方法重写

```
1 @Override
2 @Nullable
3 public View resolveViewName(String viewName, Locale locale) throws Exception {
4     RequestAttributes attrs = RequestContextHolder.getRequestAttributes();
5     Assert.state(attrs instanceof ServletRequestAttributes, "No current
6     ServletRequestAttributes");
7     List<MediaType> requestedMediaTypes =
8     getMediaTypes(((ServletRequestAttributes) attrs).getRequest());
9     if (requestedMediaTypes != null) {
10         // 获取候选的视图
11         List<View> candidateViews = getCandidateViews(viewName, locale,
12             requestedMediaTypes);
13         // 得到最好的视图
14         View bestView = getBestView(candidateViews, requestedMediaTypes,
15             attrs);
16         if (bestView != null) {
17             return bestView;
18         }
19     }
20 }
```

获取候选的视图

```
1 private List<View> getCandidateViews(String viewName, Locale locale,
2     List<MediaType> requestedMediaTypes) throws Exception {
3     List<View> candidateViews = new ArrayList<>();
4     if (this.viewResolvers != null) {
5         Assert.state(this.contentNegotiationManager != null, "No
ContentNegotiationManager set");
6         // 遍历所有的视图解析器
7     }
8 }
```

```

6     for (ViewResolver viewResolver : this.viewResolvers) {
7         // 封装成一个对象
8         View view = viewResolver.resolveViewName(viewName, locale);
9         if (view != null) {
10             // 添加到候选的视图
11             candidateviews.add(view);
12         }
13         for (MediaType requestedMediaType : requestedMediaTypes) {
14             List<String> extensions =
15                 this.contentNegotiationManager.resolveFileExtensions(requestedMediaType);
16                 for (String extension : extensions) {
17                     String viewNameWithExtension = viewName + '.' +
18                     extension;
19                     view =
20                     viewResolver.resolveViewName(viewNameWithExtension, locale);
21                     if (view != null) {
22                         candidateviews.add(view);
23                     }
24                 }
25             if (!Collectionutils.isEmpty(this.defaultViews)) {
26                 candidateviews.addAll(this.defaultViews);
27             }
28             // 返回出去
29             return candidateViews;
30         }

```

所以得出结论：ContentNegotiatingViewResolver 这个视图解析器就是用来组合所有的视图解析器的！！！

6.4.2 自定义视图解析器

在主程序中去写一个视图解析器

```

1 package cn.guardwhy.config;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.web.servlet.View;
6 import org.springframework.web.servlet.ViewResolver;
7 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
8
9 import java.util.Locale;
10
11 // 拓展 springMVC
12 @Configuration
13 public class MyMvcConfig implements WebMvcConfigurer {
14     // ViewResolver 实现了视图解析器接口类,可以将它看做视图解析器
15     @Bean
16     public ViewResolver myViewResolver(){
17         return new MyViewResolver();
18     }
19
20     // 自定义了一个自己的视图解析器MyViewResolver
21     public static class MyViewResolver implements ViewResolver{

```

```

22     @Override
23     public View resolveViewName(String viewName, Locale locale) throws
Exception {
24         return null;
25     }
26 }
27 }
```

判断图解析器有没有起作用呢？

全局搜索 DispatcherServlet，DispatcherServlet 中有个给 doDispatch 方法。

```

try {
    doDispatch(request, response);
}
finally {
    if (!WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
        // Restore the original attribute snapshot, in case of an include.
        if (attributesSnapshot != null) {
            restoreAttributesAfterInclude(request, attributesSnapshot);
        }
    }
}
```

DispatcherServlet 中的 doDispatch 方法加个断点进行调试一下，因为所有的请求都会走到这个方法中。



启动项目，然后随便访问一个页面，看一下Debug信息，找到 this



小结

如果想自定义一些定制化的功能，只需要写这个自定义组件，然后将它交给Springboot，Springboot会自动的帮助我们自动装配！！！

6.4.3 转换器和格式化器

分析源码

全局搜索 `WebMvcAutoConfiguration`, 然后局部搜索 `formatters`

```
@Bean  
@Override  
public FormattingConversionService mvcConversionService() {  
    Format format = this.mvcProperties.getFormat();  
    WebConversionService conversionService = new WebConversionService(new DateTimeFormatters()  
        .dateFormat(format.getDate()).timeFormat(format.getTime()).dateTimeFormat(format.getDateTime()));  
    addFormatters(conversionService);  
    return conversionService;  
}  
  
@Configuration(proxyBeanMethods = false)  
@EnableConfigurationProperties(WebProperties.class)  
public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration implements ResourceLoaderAware {  
  
    private static final Log logger = LoggerFactory.getLog(WebMvcConfigurer.class);  
  
    private final Resources resourceProperties;  
    private final WebMvcProperties mvcProperties;
```

局部搜索 `dateFormat`

```
@Deprecated  
@DeprecatedConfigurationProperty(replacement = "spring.mvc.format.date")  
public String getDateFormat() {  
    return this.format.getDate();}  
/**  
 * Date-time format to use, for example `yyyy-MM-dd HH:mm:ss`.  
 */  
private String dateTime;  
  
public String getDate() {  
    return this.date;
```

结论

可以看到在Properties文件中，可以进行自动配置它。如果配置了自己的格式化方式，就会注册到Bean中生效，我们可以在配置文件中配置日期格式化的规则

```
1 # 自定义的配置日期格式化  
2 spring.mvc.date-format = 2001/12/06
```

6.4.4 修改SpringBoot的默认配置

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，如果有就用用户配置的，如果没有就用自动配置的。

如果有些组件可以存在多个，比如我们的视图解析器，就将用户配置的和自己默认的组合起来。

案例说明

编写一个@Configuration注解类，并且类型要为WebMvcConfigurer，**还不能标注@EnableWebMvc注解**。新建一个包叫config，写一个类MyMvcConfig。

```
1 package cn.guardwhy.config;
2
3 import org.springframework.context.annotation.Configuration;
4 import
5 org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
6 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
7
8 // 如果要扩展SpringMVC,官方建议这样做!!!
9 @Configuration
10 public class MyMvcConfig implements WebMvcConfigurer {
11     @Override
12     public void addViewControllers(ViewControllerRegistry registry) {
13         registry.addViewController("/user").setViewName("test");
14     }
15 }
```

执行结果



当要扩展SpringMVC，官方就推荐我们这么去使用，既保SpringBoot留所有的自动配置，也能用我们自己扩展的配置。

为什么加了一个注解(@EnableWebMvc)，自动配置就失效了

```
1 package cn.guardwhy.config;
2
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.web.servlet.config.annotation.EnableWebMvc;
5 import
6 org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
7 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
8
9 @Configuration
10 @EnableWebMvc // 为什么加了一个注解，自动配置就失效了？
11 public class MyMvcConfig implements WebMvcConfigurer {
12     @Override
13     public void addViewControllers(ViewControllerRegistry registry) {
14         registry.addViewController("/user").setViewName("test");
15     }
16 }
```

源码分析

1、`WebMvcAutoConfiguration`是SpringMVC的自动配置类，里面有一个类`WebMvcAutoConfigurationAdapter`。

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class, TaskExecutionAutoConfiguration.class,
    ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {

    @Configuration(proxyBeanMethods = false)
    @Import(EnableWebMvcConfiguration.class)
    @EnableConfigurationProperties({ WebMvcProperties.class,
        org.springframework.boot.autoconfigure.web.ResourceProperties.class, WebProperties.class })
    @Order(0)
    public static class WebMvcAutoConfigurationAdapter implements WebMvcConfigurer {

        private final WebMvcProperties mvcProperties;

        private final ListableBeanFactory beanFactory;
```

2、`WebMvcAutoConfigurationAdapter`上有一个注解，在做其他自动配置时会导入。

```
1 | @Import(EnableWebMvcConfiguration.class)
```

3、点击`EnableWebMvcConfiguration`这个类，它继承了一个父类：

`DelegatingWebMvcConfiguration`。

```
1 | @Configuration(proxyBeanMethods = false)
2 | @EnableConfigurationProperties(WebProperties.class)
3 | public static class EnableWebMvcConfiguration extends
4 | DelegatingWebMvcConfiguration implements ResourceLoaderAware {
5 |
6 |     private static final Log logger =
7 | LogFactory.getLog(WebMvcConfigurer.class);
8 |
9 |     private final Resources resourceProperties;
10 |
11 |     private final WebMvcProperties mvcProperties;
12 |
13 |     private final WebProperties webProperties;
14 |
15 |     private final ListableBeanFactory beanFactory;
16 |
17 |     private final ResourceHandlerRegistrationCustomizer
18 |     resourceHandlerRegistrationCustomizer;
19 |
20 |     private ResourceLoader resourceLoader;
21 | }
```

5、点击`DelegatingWebMvcConfiguration`，该类的作用是从容器中获取所有的webmvcconfig。

```

1  @Configuration(proxyBeanMethods = false)
2  public class DelegatingWebMvcConfiguration extends
3      webMvcConfigurationSupport {
4
5
6  @Autowired(required = false)
7  public void setConfigurers(List<WebMvcConfigurer> configurers) {
8      if (!CollectionUtils.isEmpty(configurers)) {
9          this.configurers.addWebMvcConfigurers(configurers);
10     }
11 }

```

4、点击 `@EnableWebMvc`，这个注解就是导入一个类：DelegatingWebMvcConfiguration。

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(DelegatingWebMvcConfiguration.class)
public @interface EnableWebMvc {
}

```

5、`DelegatingWebMvcConfiguration` 继承 `WebMvcConfigurationSupport`。

```

@Configuration(proxyBeanMethods = false)
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {

    private final WebMvcConfigurerComposite configurers = new WebMvcConfigurerComposite();

    @Autowired(required = false)
    public void setConfigurers(List<WebMvcConfigurer> configurers) {
        if (!CollectionUtils.isEmpty(configurers)) {
            this.configurers.addWebMvcConfigurers(configurers);
        }
    }
}

```

6、当该类存在时，`@ConditionalOnMissingBean(webMvcConfigurationSupport.class)` 会立马失效，所有自动配置会立马蹦掉！

```

    @Configuration(proxyBeanMethods = false)
    @ConditionalOnWebApplication(type = Type.SERVLET)
    @ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
    @ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
    @AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
    @AutoConfigureAfter({ DispatcherServletAutoConfiguration.class, TaskExecutionAutoConfiguration.class,
        ValidationAutoConfiguration.class })
    public class WebMvcAutoConfiguration {

```

该bean不存在的情况下，才会生效

小结

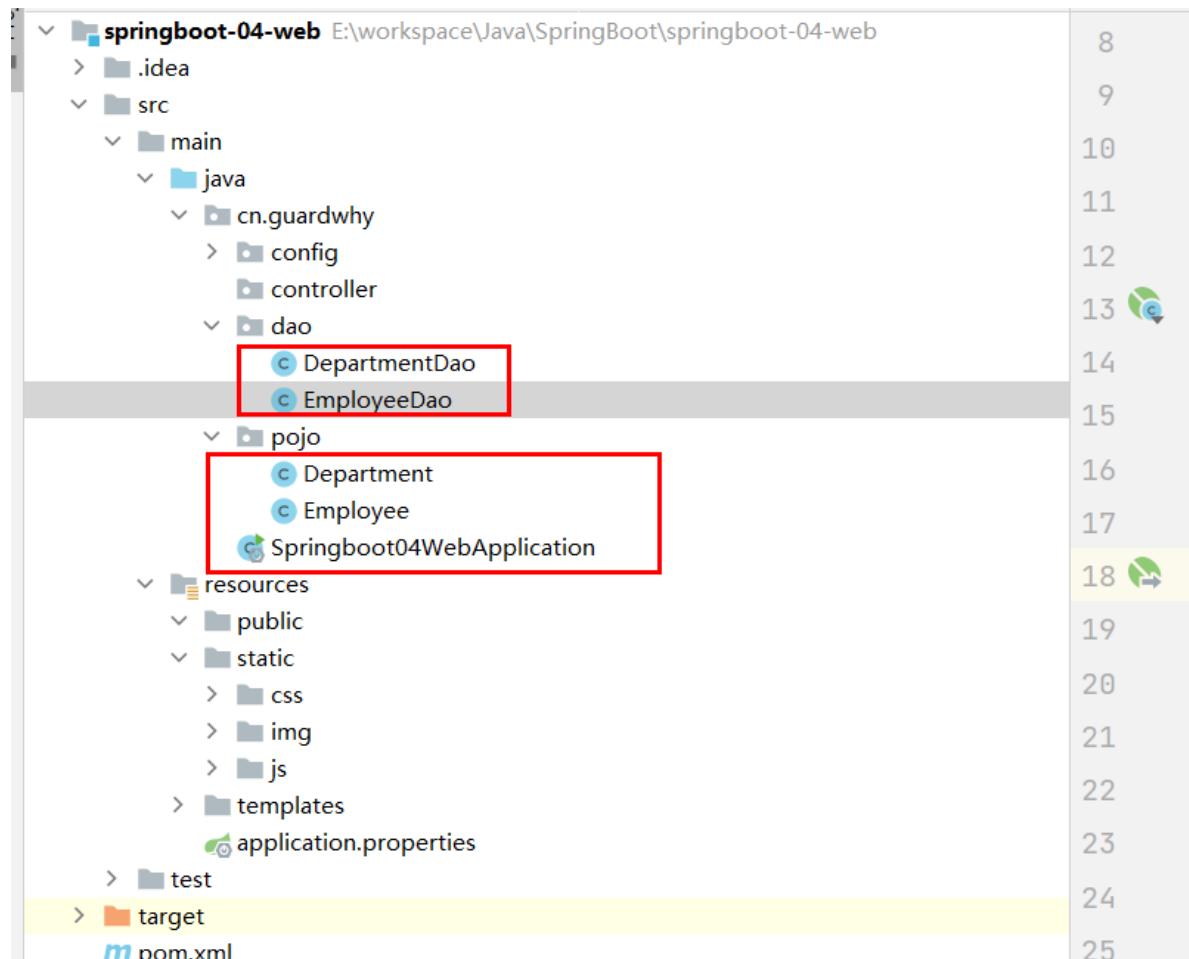
在Springboot中，有非常多的(???)Configuration帮助进行拓展配置，只要是看到了这个文件，需要注意！！！

7- 员工管理系统

7.1 准备环境

项目所需的素材：<https://pan.baidu.com/s/1FBdaNfwIZtks7QllbovhCw> 提取码：h33l

项目目录



7.1.1 导入相关依赖

```

1 <!-- Lombok -->
2 <dependency>
3   <groupId>org.projectlombok</groupId>
4   <artifactId>lombok</artifactId>
5 </dependency>
6

```

```
7 <!--thymeleaf版本-->
8 <dependency>
9   <groupId>org.thymeleaf</groupId>
10  <artifactId>thymeleaf-spring5</artifactId>
11 </dependency>
12 <dependency>
13   <groupId>org.thymeleaf.extras</groupId>
14   <artifactId>thymeleaf-extras-java8time</artifactId>
15 </dependency>
```

7.1.2 实体类实现

```
1 package cn.guardwhy.pojo;
2 import lombok.AllArgsConstructor;
3 import lombok.Data;
4 import lombok.NoArgsConstructor;
5
6 // 部门表
7 @Data
8@AllArgsConstructor
9@NoArgsConstructor
10 public class Department {
11     private Integer id;
12     private String departmentName;
13 }
```

```
1 package cn.guardwhy.pojo;
2
3 import lombok.Data;
4 import lombok.NoArgsConstructor;
5
6 import java.util.Date;
7
8 // 员工表
9 @Data
10@NoArgsConstructor
11 public class Employee {
12     private Integer id;
13     private String lastName;
14     private String email;
15     private Integer gender; //0: 女 1: 男
16     private Department department;
17     private Date birth;
18
19     public Employee(Integer id, String lastName, String email, Integer
gender, Department department) {
20         this.id = id;
21         this.lastName = lastName;
22         this.email = email;
23         this.gender = gender;
24         this.department = department;
25         // 默认创建日期
26         this.birth = new Date();
27     }
28 }
```

7.1.3 持久层实现

DepartmentDao

```
1 package cn.guardwhy.dao;
2
3 import cn.guardwhy.pojo.Department;
4 import org.springframework.stereotype.Repository;
5
6 import java.util.Collection;
7 import java.util.HashMap;
8 import java.util.Map;
9
10 // 部门的dao
11 @Repository
12 public class DepartmentDao {
13     // 模拟数据库中的数据
14     private static Map<Integer, Department> departments = null;
15     static {
16         departments = new HashMap<Integer, Department>(); // 创建部门表
17         departments.put(101, new Department(101, "教学部"));
18         departments.put(102, new Department(102, "市场部"));
19         departments.put(103, new Department(103, "教研部"));
20         departments.put(104, new Department(104, "运营部"));
21         departments.put(105, new Department(105, "后勤部"));
22     }
23
24     // 2.获取所有部门信息
25     public Collection<Department> getDepartments(){
26         return departments.values();
27     }
28     // 3.通过id得到部门
29     public Department getDepartmentById(Integer id){
30         return departments.get(id);
31     }
32 }
```

EmployeeDao

```
1 package cn.guardwhy.dao;
2
3 import cn.guardwhy.pojo.Department;
4 import cn.guardwhy.pojo.Employee;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Repository;
7
8 import java.util.Collection;
9 import java.util.HashMap;
10 import java.util.Map;
11
12 @Repository
13 public class EmployeeDao {
14     // 1.模拟数据库中的数据
15     private static Map<Integer, Employee> employees = null;
16     // 2.员工有所属的部门
17     @Autowired
18     private DepartmentDao departmentDao;
```

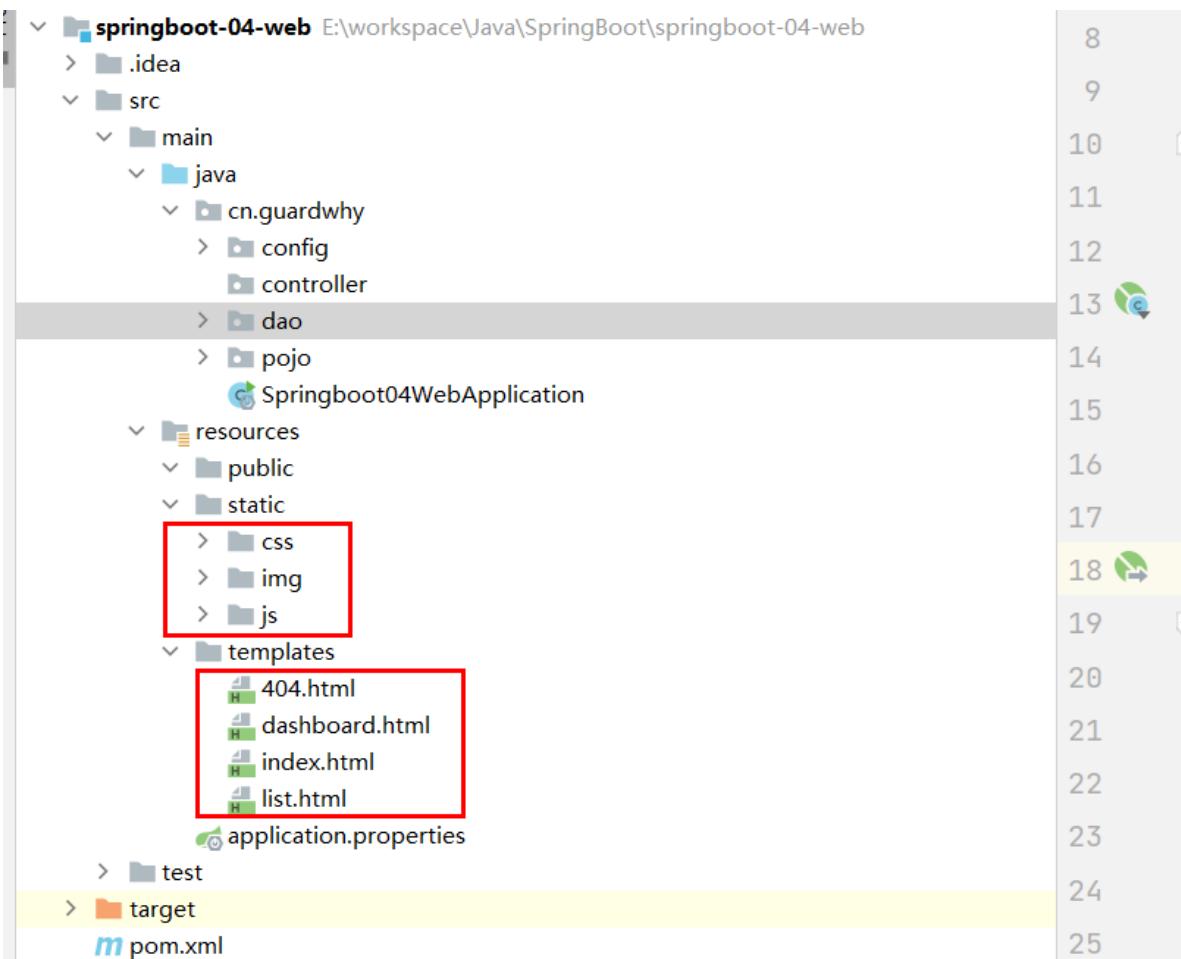
```

19     static {
20         // 3.创建一个员工表
21         employees = new HashMap<Integer, Employee>();
22         employees.put(1001, new Employee(1001, "kobe", "1234678@qq.com", 1,
23             new Department(101, "后勤部")));
24         employees.put(1002, new Employee(1002, "james", "1234678@qq.com", 1,
25             new Department(102, "市场部")));
26         employees.put(1003, new Employee(1003, "xiao hong", "1234678@qq.com",
27             0, new Department(103, "教学部")));
28         employees.put(1004, new Employee(1004, "harden", "1234678@qq.com", 1,
29             new Department(104, "运营部")));
30         employees.put(1005, new Employee(1005, "guardwhy", "1234678@qq.com",
31             1, new Department(105, "教研部)));
32     }
33     // 3.主键自增
34     private static Integer initId = 1006;
35     // 4.增加一个员工
36     public void add(Employee employee){
37         // 条件判断
38         if(employee.getId() == null){
39             employee.setId(initId++);
40         }
41
42         employee.setDepartment(departmentDao.getDepartmentById(employee.getDepartment().getId()));
43         employees.put(employee.getId(), employee);
44     }
45     // 5.查询全部员工的信息
46     public Collection<Employee> getAll(){
47         return employees.values();
48     }
49     // 6.通过id查询员工
50     public Employee getEmployeeById(Integer id){
51         return employees.get(id);
52     }
53     // 7.通过id删除员工
54     public void delete(Integer id){
55         employees.remove(id);
56     }
57 }
```

7.2 首页实现

7.2.1 导入静态资源

css, js等放在static文件夹下, html 放在 templates文件夹下。



7.2.2 编写MVC的扩展配置

```

1 package cn.guardwhy.config;
2
3 import org.springframework.context.annotation.Configuration;
4 import
5 org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
6 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
7 @Configuration
8 public class MyMvcConfig implements WebMvcConfigurer {
9     @Override
10    public void addViewControllers(ViewControllerRegistry registry) {
11        registry.addViewController("/").setViewName("index");
12        registry.addViewController("/index.html").setViewName("index");
13    }
14 }
```

2、解决资源导入的问题

首页配置: 注意点, 所有的页面静态资源都需要使用thymeleaf接管, @{}。

application.properties

```

1 # 关闭模板引擎的缓存
2 spring.thymeleaf.cache=false
3
4 # 改变路径
5 server.servlet.context-path=/user
```

index.html

```
1 <html lang="en" xmlns:th="http://www.thymeleaf.org">
2
3     <!-- Bootstrap core CSS --&gt;
4     &lt;link th:href="@{/css/bootstrap.min.css}" rel="stylesheet"&gt;
5     <!-- Custom styles for this template --&gt;
6     &lt;link th:href="@{/css/signin.css}" rel="stylesheet"&gt;
7
8     &lt;img class="mb-4" th:src="@{/img/bootstrap-solid.svg}" alt="" width="72"
9      height="72"&gt;</pre>
```

404.html

```
1 <html lang="en" xmlns:th="http://www.thymeleaf.org">
2
3     <!-- Bootstrap core CSS --&gt;
4     &lt;link th:href="@{/css/bootstrap.min.css}" rel="stylesheet"&gt;
5     <!-- Custom styles for this template --&gt;
6     &lt;link th:href="@{/css/dashboard.css}" rel="stylesheet"&gt;</pre>
```

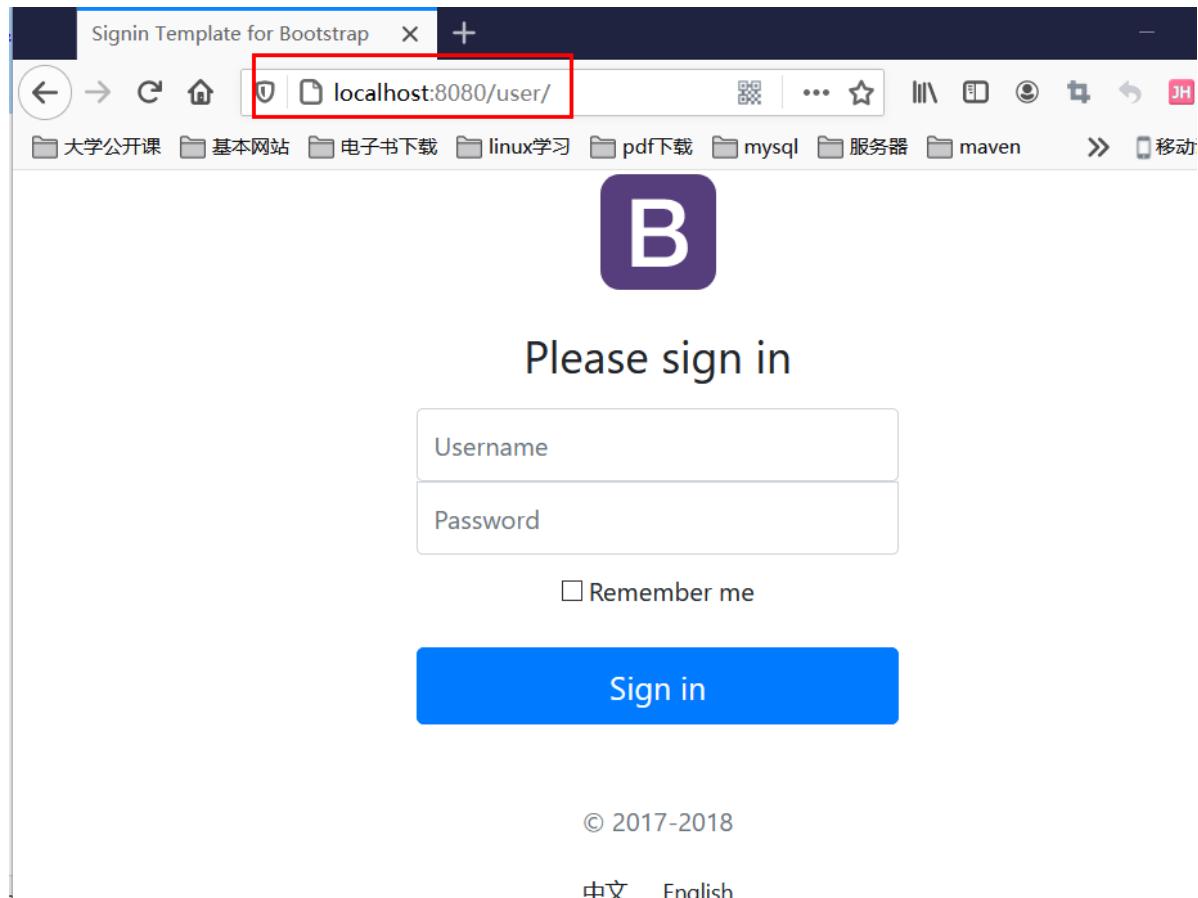
dashboard.html

```
1 <html lang="en" xmlns:th="http://www.thymeleaf.org">
2     <!-- Bootstrap core CSS --&gt;
3     &lt;link th:href="@{/css/bootstrap.min.css}" rel="stylesheet"&gt;
4
5     <!-- Custom styles for this template --&gt;
6     &lt;link th:href="@{/css/dashboard.css}" rel="stylesheet"&gt;</pre>
```

list.html

```
1 <html lang="en" xmlns:th="http://www.thymeleaf.org">
2
3     <!-- Bootstrap core CSS --&gt;
4     &lt;link th:href="@{/css/bootstrap.min.css}" rel="stylesheet"&gt;
5
6     <!-- Custom styles for this template --&gt;
7     &lt;link th:href="@{/css/dashboard.css}" rel="stylesheet"&gt;</pre>
```

7.2.3 执行结果

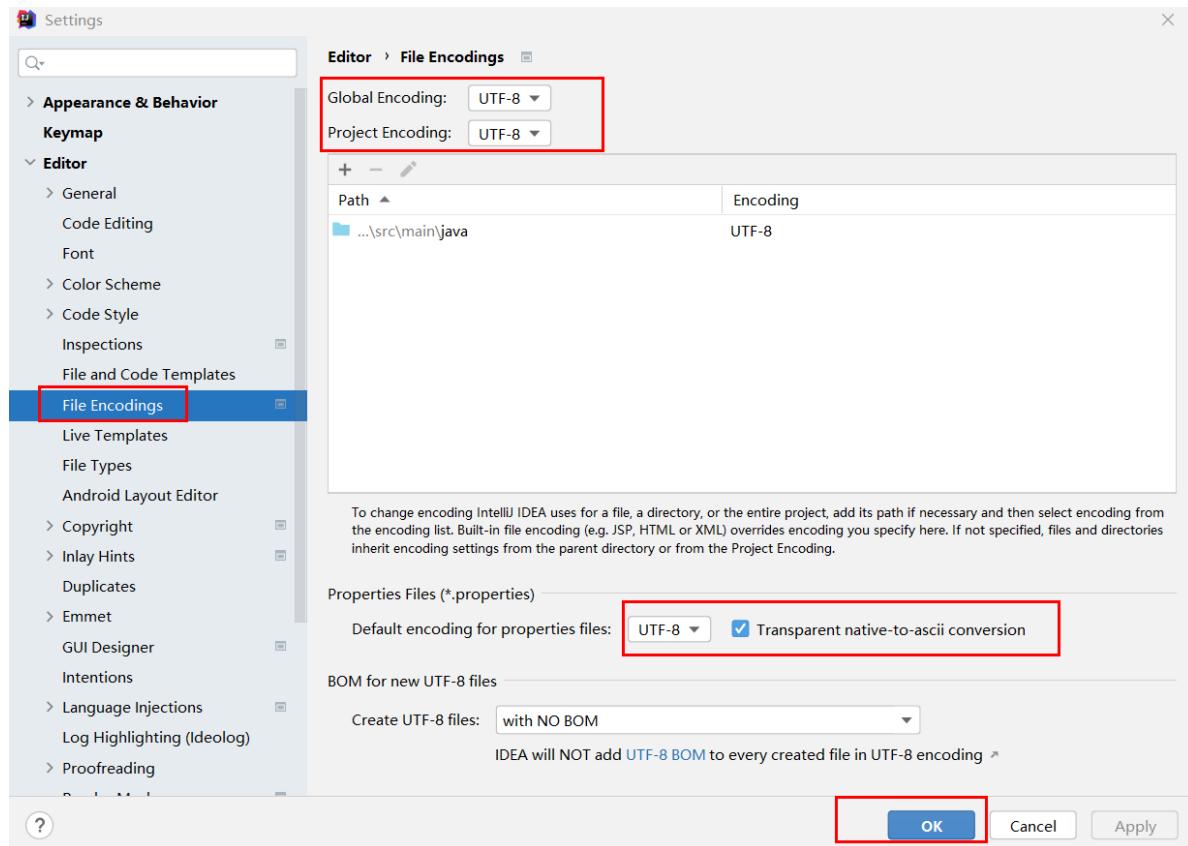


7.3 页面国际化

基本需求：有的时候，有些网站会去涉及中英文甚至多语言的切换，这时候就需要学习国际化。

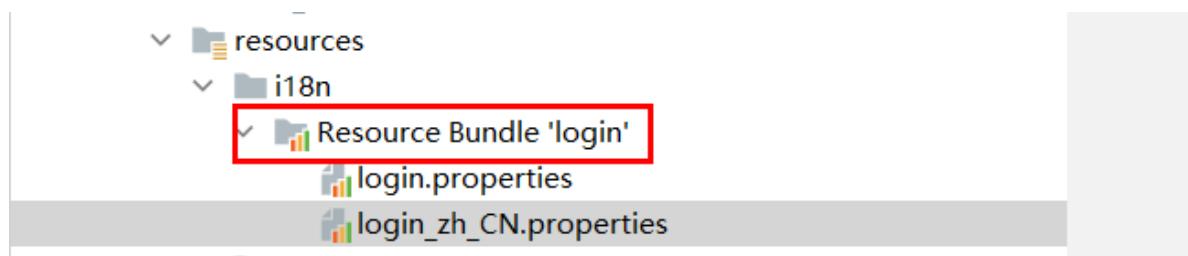
7.3.1 准备工作

先在IDEA中统一设置 properties 的编码。

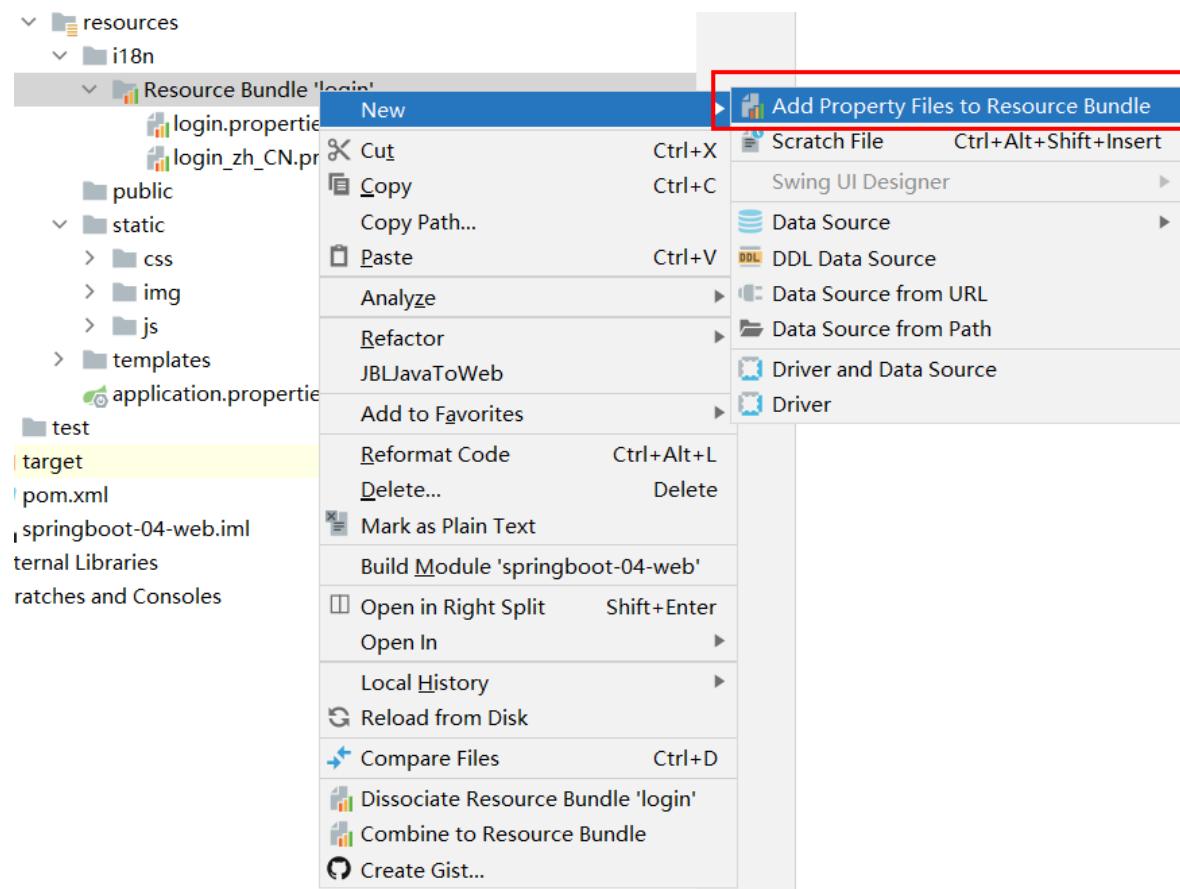


7.3.2 文件创建

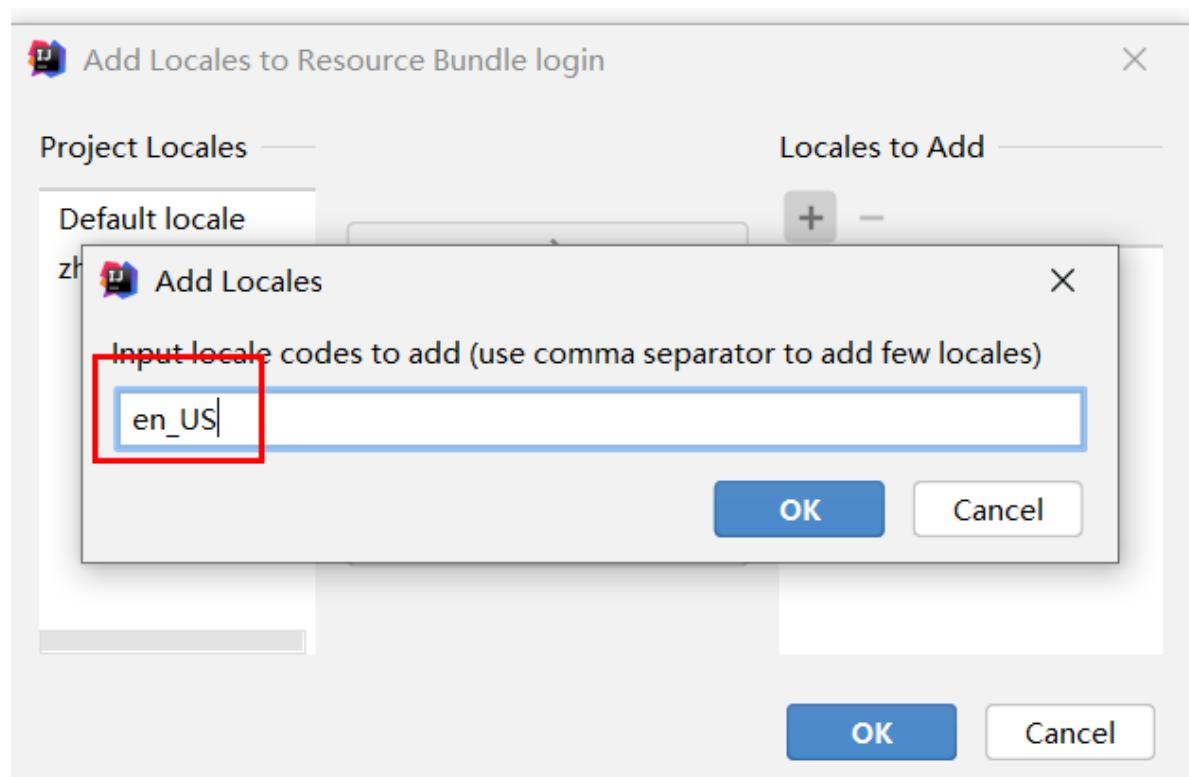
- 1、在resources资源文件下新建一个i18n目录，存放国际化配置文件。
- 2、建立一个login.properties文件，还有一个login_zh_CN.properties。发现IDEA自动识别了要做国际化操作，文件夹变了。



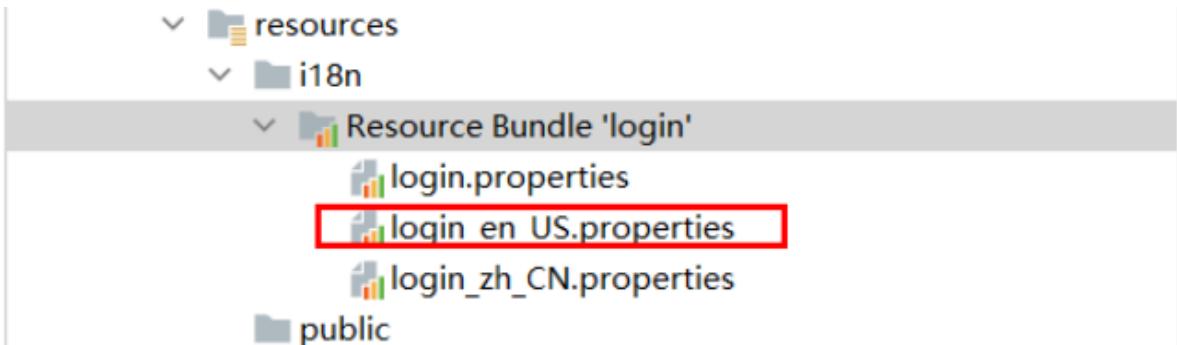
- 3、可以在这上面去新建一个文件



4、弹出如下页面：再添加一个英文的文件。



结果如下

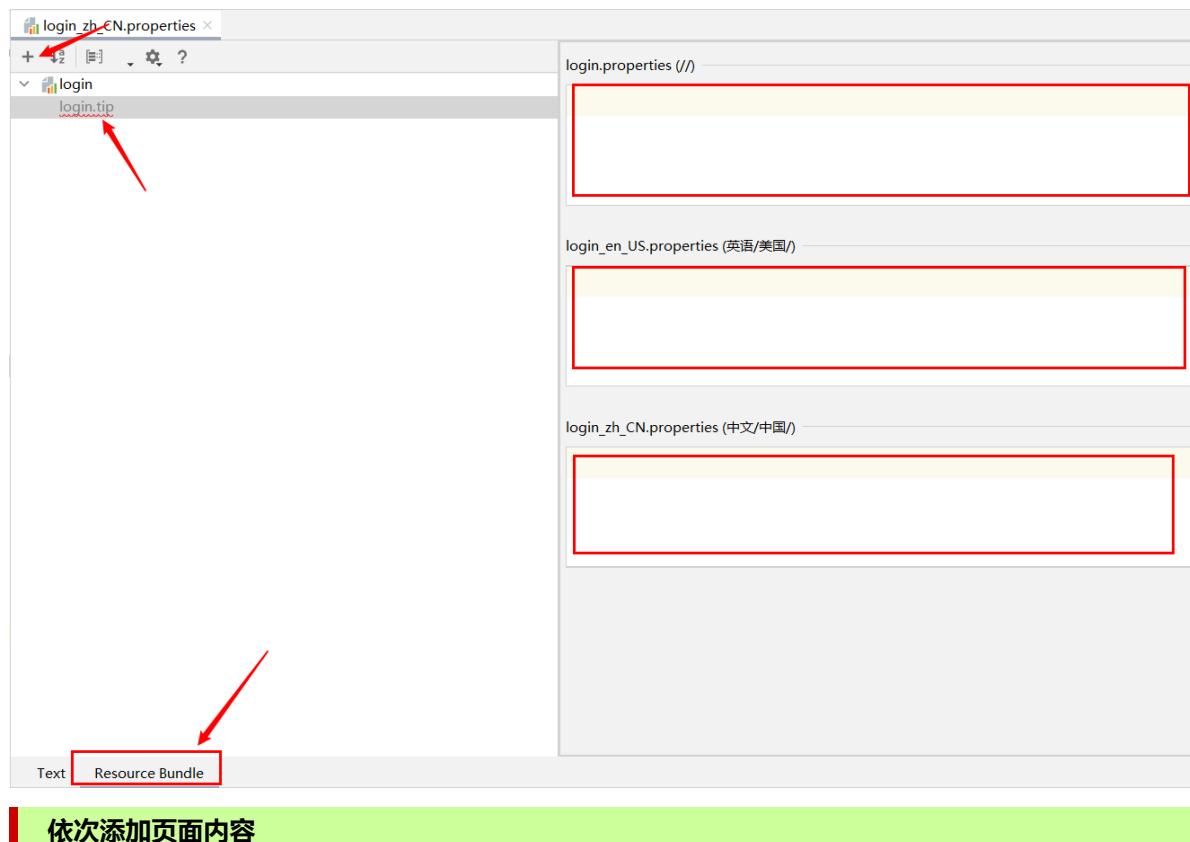


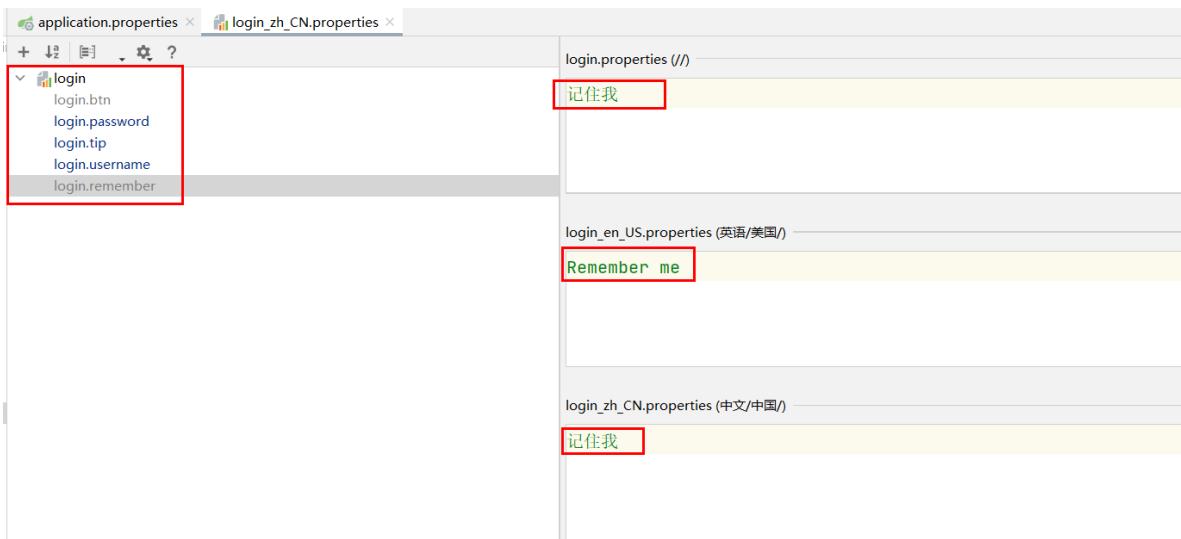
7.3.4 编写配置

可以看到idea下面有另外一个视图，通过它，我们可以编写配置文件。



这个视图点击 + 号就可以直接添加属性了，新建一个login.tip可以看到边上三个文件框可以输入。





然后去查看配置文件

login.properties: 默认

```

1 login.btn=登录
2 login.password=密码
3 login.remember=记住我
4 login.tip=请登录
5 login.username=用户名

```

英文:

```

1 login.btn=Sign in
2 login.password=Password
3 login.remember=Remember me
4 login.tip=Please sign in
5 login.username=Username

```

中文:

```

1 login.btn=登录
2 login.password=密码
3 login.remember=记住我
4 login.tip=请登录
5 login.username=用户名

```

7.3.5 文件源码分析

SpringBoot对国际化的自动配置: [MessageSourceAutoConfiguration](#), 点击源码进行分析! ! !

```

1 @Configuration(proxyBeanMethods = false)
2 @ConditionalOnMissingBean(name =
AbstractApplicationContext.MESSAGE_SOURCE_BEAN_NAME, search =
SearchStrategy.CURRENT)
3 @AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
4 @Conditional(ResourceBundleCondition.class)
5 @EnableConfigurationProperties
6 public class MessageSourceAutoConfiguration {
7
8     private static final Resource[] NO_RESOURCES = {};

```

```

9
10    @Bean
11    @ConfigurationProperties(prefix = "spring.messages")
12    public MessageSourceProperties messageSourceProperties() {
13        return new MessageSourceProperties();
14    }
15}

```

MessageSourceAutoConfiguration里面有一个方法，这里SpringBoot已经自动配置好了管理我们国际化资源文件的组件 ResourceBundleMessageSource。

```

1  @Bean // 获取 properties 传递过来的值进行判断
2  public MessageSource messageSource(MessageSourceProperties properties) {
3      ResourceBundleMessageSource messageSource = new
ResourceBundleMessageSource();
4      if (StringUtils.hasText(properties.getbasename())) {
5          // 设置国际化文件的基础名
6          messageSource.setBasename(StringUtils
7
.commaDelimitedListToStringArray(StringUtils.trimAllWhitespace(properties.ge
tBasename())));
8      }
9      if (properties.getEncoding() != null) {
10         messageSource.setDefaultEncoding(properties.getEncoding().name());
11     }
12
messageSource.setFallbackToSystemLocale(properties.isFallbackToSystemLocale
());
13     Duration cacheDuration = properties.getCacheDuration();
14     if (cacheDuration != null) {
15         messageSource.setCacheMillis(cacheDuration.toMillis());
16     }
17
messageSource.setAlwaysUseMessageFormat(properties.isAlwaysUseMessageFormat
());
18
messageSource.setUseCodeAsDefaultMessage(properties.isUseCodeAsDefaultMess
age());
19     return messageSource;
20 }

```

点击 MessageSourceProperties，查看配置文件！！！

```
public class MessageSourceProperties {
```

Comma-separated list of basenames (essentially a fully-qualified classpath location), each following the ResourceBundle convention with relaxed support for slash based locations. If it doesn't contain a package qualifier (such as "org.mypackage"), it will be resolved from the classpath root.

```
private String basename = "messages";
```

Message bundles encoding.

```
private Charset encoding = StandardCharsets.UTF_8;
```

配置文件的真实位置

```
1 | spring.messages.basename=i18n.login
```

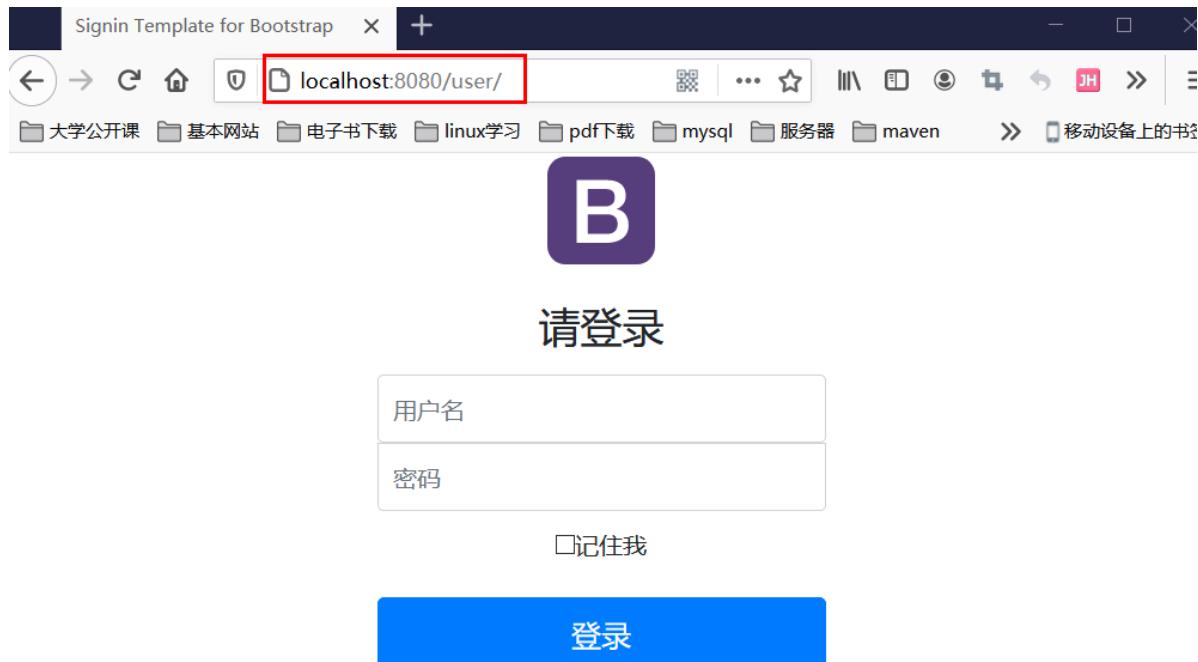
7.3.6 页面国际化值

去页面获取国际化的值，查看Thymeleaf的文档，找到message Expressions取值操作为：`#{{...}}`。

index.html

```
1 <body class="text-center">
2     <form class="form-signin" action="dashboard.html">
3         
5             <h1 class="h3 mb-3 font-weight-normal" th:text="#{login.tip}">Please
6             sign in</h1>
7             <input type="text" class="form-control" th:placeholder="#
8             {login.username}" required="" autofocus="">
9             <input type="password" class="form-control" th:placeholder="#
10            {login.password}" required="">
11            <div class="checkbox mb-3">
12                <label>
13                    <input type="checkbox" value="remember-me" th:text="#
14                    {login.remember}">
15                </label>
16            </div>
17            <button class="btn btn-lg btn-primary btn-block" type="submit"
18                th:text="#{login.btn}">Sign in</button>
19            <p class="mt-5 mb-3 text-muted">© 2017-2018</p>
20            <a class="btn btn-sm">中文</a>
21            <a class="btn btn-sm">English</a>
22        </form>
23    </body>
```

可以去启动项目，访问一下，发现已经自动识别为中文的了



7.3.7 配置国际化解析

1、全局搜索WebMvcAutoConfiguration (webmvc) 自动配置文件。

在Spring中有一个国际化的Locale (区域信息对象), **里面有一个叫做LocaleResolver(获取区域信息对象) 的解析器。**

```
1 @Configuration(proxyBeanMethods = false)
2 @ConditionalOnWebApplication(type = Type.SERVLET)
3 @ConditionalOnClass({ Servlet.class, DispatcherServlet.class,
4 webMvcConfigurer.class })
5 @ConditionalOnMissingBean(webMvcConfigurationSupport.class)
6 @AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
7 @AutoConfigureAfter({ DispatcherServletAutoConfiguration.class,
8 TaskExecutionAutoConfiguration.class,
9 validationAutoConfiguration.class })
10 public class WebMvcAutoConfiguration {
```

2、在WebMvcAutoConfiguration搜索LocaleResolver。

```
1 @Override
2 @Bean
3 @ConditionalOnMissingBean(name =
DispatcherServlet.LOCALE_RESOLVER_BEAN_NAME)
4 @SuppressWarnings("deprecation")
5 public LocaleResolver localeResolver() {
6     // 容器中没有就自己配, 有的话就用用户配置的
7     if (this.webProperties.getLocaleResolver() ==
WebProperties.LocaleResolver.FIXED) {
```

```

8         return new FixedLocaleResolver(this.webProperties.getLocale());
9     }
10    if (this.mvcProperties.getLocaleResolver() ==
11        webMvcProperties.LocaleResolver.FIXED) {
12        return new FixedLocaleResolver(this.mvcProperties.getLocale());
13    }
14    // 接收头国际化分解
15    AcceptHeaderLocaleResolver localeResolver = new
16    AcceptHeaderLocaleResolver();
17    Locale locale = (this.webProperties.getLocale() != null) ?
18        this.webProperties.getLocale()
19        : this.mvcProperties.getLocale();
20    localeResolver.setDefaultLocale(locale);
21    return localeResolver;
22 }
```

3、点击AcceptHeaderLocaleResolver，这个类中有一个方法。

```

1 @Override
2 public Locale resolveLocale(HttpServletRequest request) {
3     Locale defaultLocale = getDefaultLocale();
4     // 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
5     if (defaultLocale != null && request.getHeader("Accept-Language") == null) {
6         return defaultLocale;
7     }
8     Locale requestLocale = request.getLocale();
9     List<Locale> supportedLocales = getSupportedLocales();
10    if (supportedLocales.isEmpty() ||
11        supportedLocales.contains(requestLocale)) {
12        return requestLocale;
13    }
14    Locale supportedLocale = findSupportedLocale(request, supportedLocales);
15    if (supportedLocale != null) {
16        return supportedLocale;
17    }
18    return (defaultLocale != null ? defaultLocale : requestLocale);
19 }
```

4、处理的组件类(LocaleResolver)

自己写一个 MyLocaleResolver，可以在链接上携带区域信息。

```

1 import javax.servlet.http.HttpServletRequest;
2 import javax.servlet.http.HttpServletResponse;
3 import java.util.Locale;
4 // 可以在链接上携带区域信息
5 public class MyLocaleResolver implements LocaleResolver {
6     // 解析请求
7     @Override
8     public Locale resolveLocale(HttpServletRequest request) {
9         // 获得请求的语言参数
10        String language= request.getParameter("l");
11        System.out.println("Debug++>>" + language);
12        // 如果没有获取到就使用系统默认的
13        Locale locale = Locale.getDefault();
```

```

14     // 如果请求链接不为空
15     if(!StringUtil.isEmpty(language)){
16         // 分割请求参数
17         String[] split = language.split("_");
18         // 国家，地区
19         Locale = new Locale(split[0],split[1]);
20     }
21     return Locale;
22 }
23
24 @Override
25 public void setLocale(HttpServletRequest request, HttpServletResponse
response, Locale locale) {
26
27 }
28 }
```

将自己写的组件配置到Spring容器中

```

1 package cn.guardwhy.config;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.web.servlet.LocaleResolver;
6 import
7 org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
8 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
9 @Configuration
10 public class MyMvcConfig implements WebMvcConfigurer {
11     @Override
12     public void addViewControllers(ViewControllerRegistry registry) {
13         registry.addViewController("/").setViewName("index");
14         registry.addViewController("/index.html").setViewName("index");
15     }
16     @Bean
17     public LocaleResolver localeResolver(){
18         return new MyLocaleResolver();
19     }
20 }
```

5、修改前端页面的跳转连接

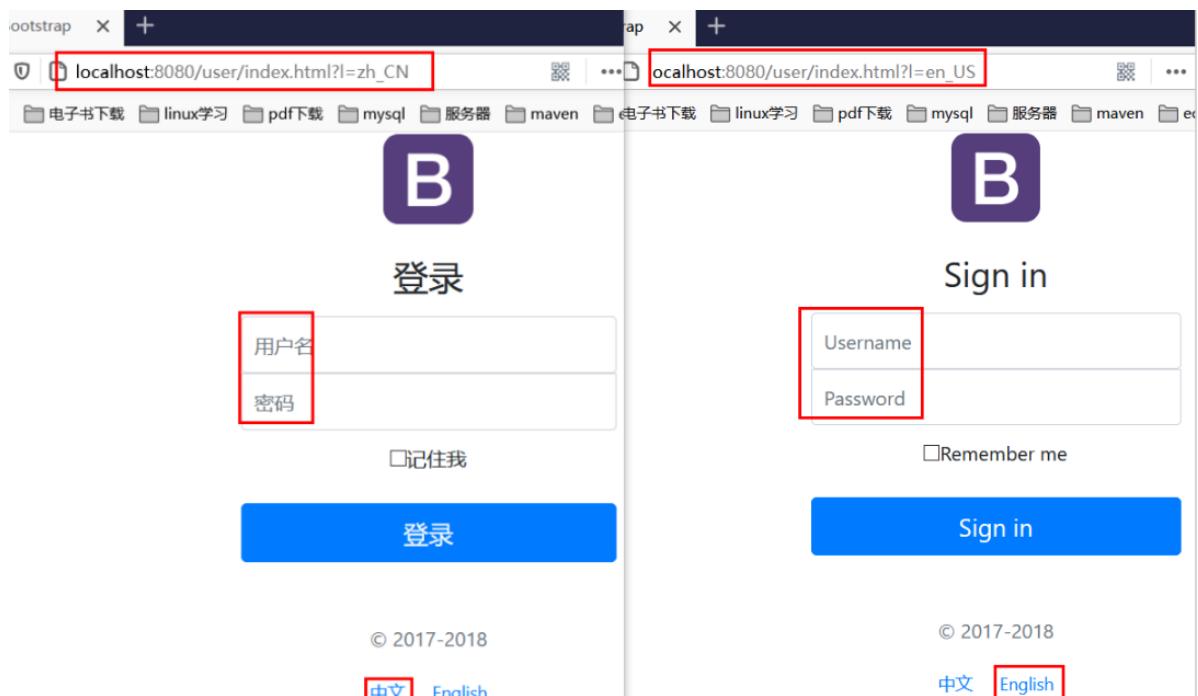
点击链接让国际化资源生效，就需要让Locale生效。

index.html

```

1 <a class="btn btn-sm" th:href="@{/index.html(l='zh_CN')}>中文</a>
2 <a class="btn btn-sm" th:href="@{/index.html(l='en_US')}>English</a>
```

执行结果



7.4 登录+拦截器(功能)

7.4.1 禁用模板缓存

页面存在缓存，所以需要禁用模板引擎的缓存。

```

1 # 关闭模板引擎的缓存
2 spring.thymeleaf.cache=false
3 # 改变路径
4 server.servlet.context-path=/guardwhy

```

7.4.2 登录操作

1、登录页面的表单提交地址写一个controller

```

1 <form class="form-signin" th:action="@{/user/login}">
2     <!--如果message的值为空，则不显示消息-->
3     <h3 style="color: red" th:text="${message}" th:if="${not
4         #strings.isEmpty(message)}"></h3>
5     <input type="text" name="username" class="form-control" th:placeholder="#
6         {login.username}" required="" autofocus="">
7     <input type="password" name="password" class="form-control"
8         th:placeholder="#{login.password}" required="">
9 </form>

```

2、测试controller

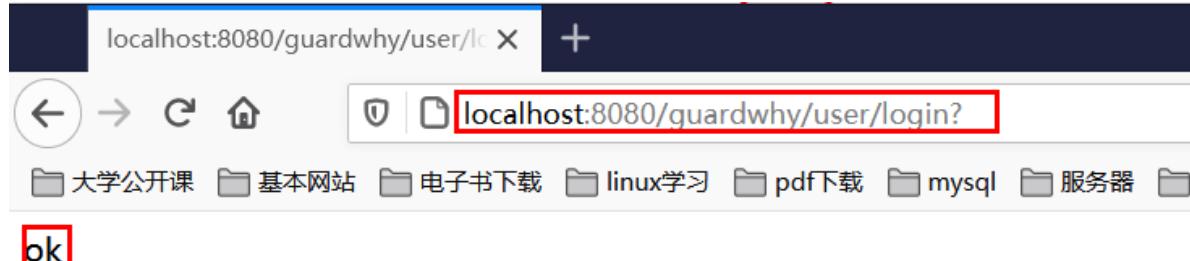
```

1 package cn.guardwhy.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.ResponseBody;
6
7 @Controller
8 public class LoginController {
9     @RequestMapping("/user/login")

```

```
10     @ResponseBody  
11     public String login(){  
12         return "ok";  
13     }  
14 }
```

测试结果



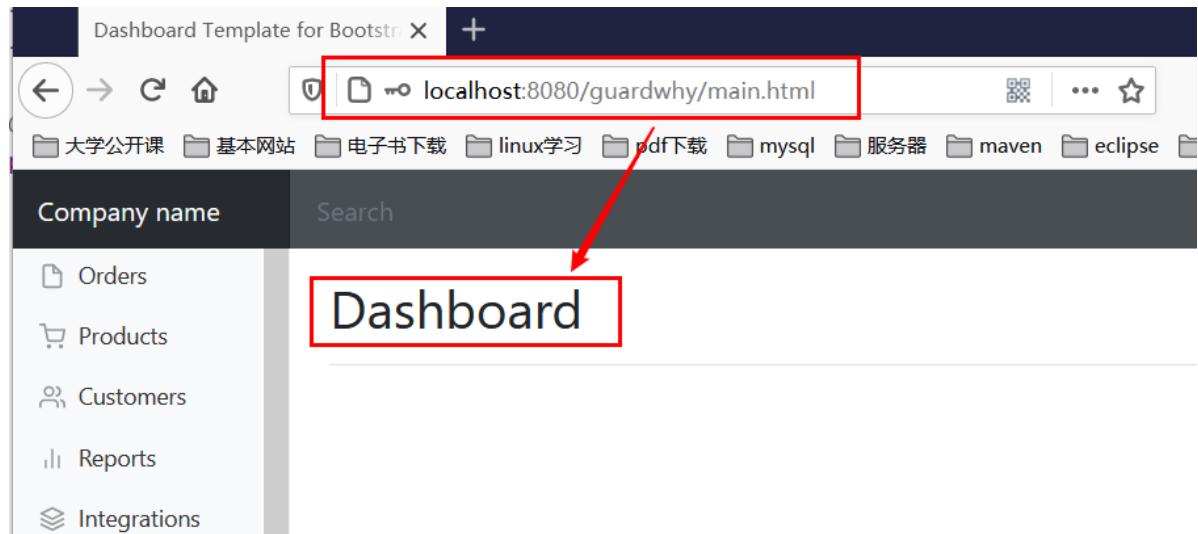
3、编写对应的controller

```
1 package cn.guardwhy.controller;  
2  
3 import org.springframework.stereotype.Controller;  
4 import org.springframework.ui.Model;  
5 import org.springframework.web.bind.annotation.RequestMapping;  
6 import org.springframework.web.bind.annotation.RequestParam;  
7 import org.thymeleaf.util.StringUtils;  
8  
9 @Controller  
10 public class LoginController {  
11     @RequestMapping("/user/login")  
12     public String login(@RequestParam("username") String username,  
13                         @RequestParam("password") String password,  
14                         Model model  
15                         ){  
16         // 具体业务  
17         if(!StringUtils.isEmpty(username) && "123666".equals(password)){  
18             // 登录成功! 防止表单重复提交, 我们重定向  
19             return "redirect:/main.html";  
20         }else{  
21             // 告诉用户, 登录失败了  
22             model.addAttribute("msg","用户名或者密码错误!");  
23             return "index";  
24         }  
25     }  
26 }
```

4、在自己的MyMvcConfig，添加一个视图控制映射。

```
1 | registry.addViewController("/main.html").setViewName("dashboard");
```

5、测试登录成功！！！



6、测试登录失败(密码错误)



7.4.3 登录拦截器

存在问题

可以直接登录到后台主页，不用登录也可以实现！怎么处理这个问题呢？可以使用拦截器机制，然后实现登录检查！

1、自定义一个拦截器

```
1 package cn.guardwhy.config;
2
3 import org.springframework.web.servlet.HandlerInterceptor;
4
5 import javax.servlet.http.HttpServletRequest;
6 import javax.servlet.http.HttpServletResponse;
7
8 public class LoginHandlerInterceptor implements HandlerInterceptor {
```

```

9     @Override
10    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
11        // 1. 登录成功之后，应该有用户的session
12        Object loginUser = request.getSession().getAttribute("loginUser");
13        // 2. 条件判断
14        if(loginUser == null){ // 未登录，返回登录页面
15            request.setAttribute("msg", "没有权限，请先登录用户！！！");
16            request.getRequestDispatcher("/index.html").forward(request,
17            response);
17            return false;
18        }else {
19            return true;
20        }
21    }
22 }

```

2、将拦截器注册到SpringMVC配置类当中

```

1  @Override
2  public void addInterceptors(InterceptorRegistry registry) {
3      registry.addInterceptor(new LoginHandlerInterceptor())
4          .addPathPatterns("/**")
5          .excludePathPatterns("/index.html",
6          "/","/user/login","/css/*","/js/**", "/img/**");
6  }

```

3、获取用户登录的信息

dashboard.html



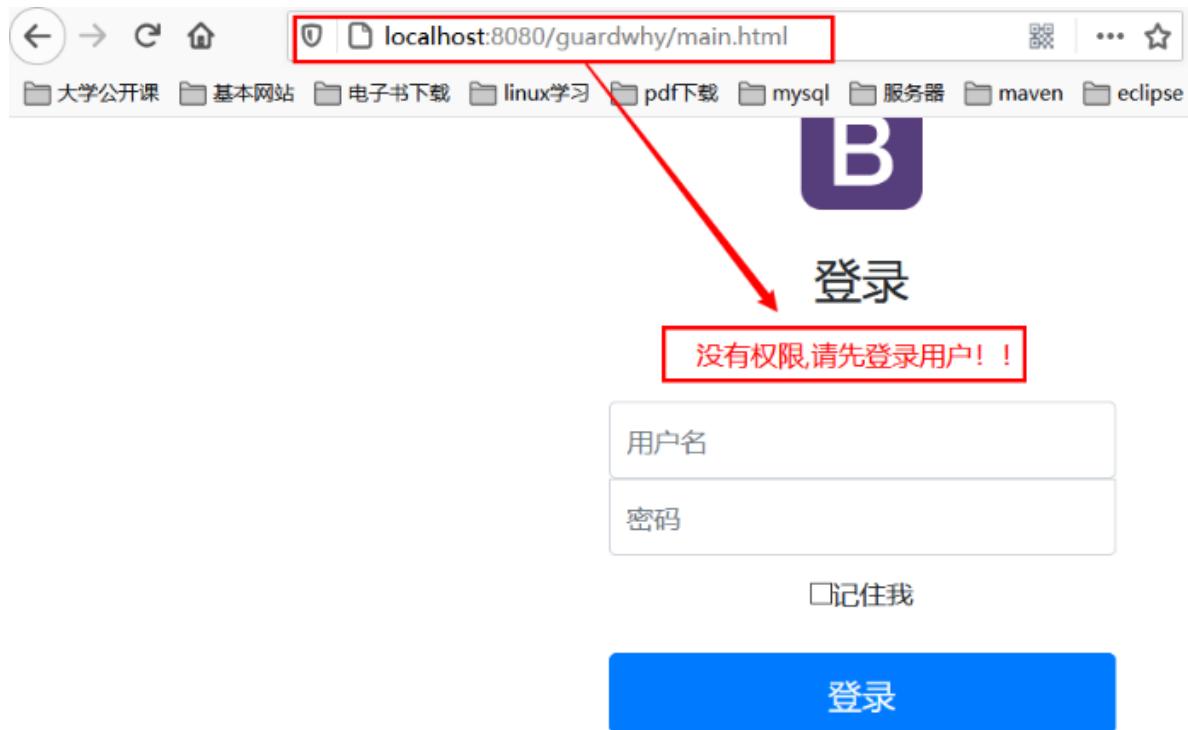
```

43
44     <body>
45         <nav class="navbar navbar-dark sticky-top bg-dark flex-md-nowrap p-0">
46             <a class="navbar-brand col-sm-3 col-md-2 mr-0" href="http://getbootstrap.com/docs/4.0/examples/dashboard/#">
47                 ${session.loginUser}
48             </a>
49             <input class="form-control form-control-dark w-100" type="text" placeholder="Search" aria-label="Search">
50             <ul class="navbar-nav px-3">
51                 <li class="nav-item text-nowrap">
52                     <a class="nav-link" href="http://getbootstrap.com/docs/4.0/examples/dashboard/#">Sign out</a>
53                 </li>
54             </ul>
55         </nav>

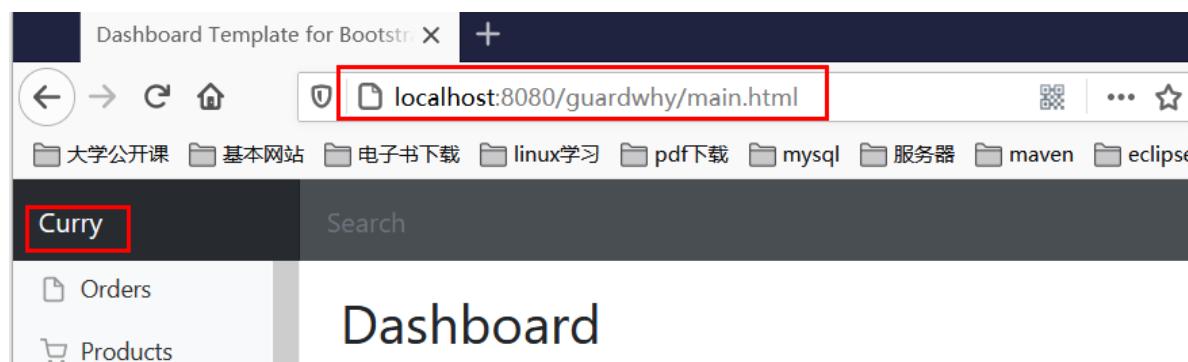
```

登录测试拦截

测试拦截成功！！！



获取用户登录的信息



7.5 员工列表实现

7.5.1 员工列表页面跳转

在主页点击Customers，就显示列表页面，将首页的侧边栏Customers改为员工管理，a链接添加请求。

dashboard.html

```
89 </li>
90 <li class="nav-item">
91 <a class="nav-link" th:href="@{/emps}">
92     员工管理
93 </a>
94 </li>
```

list.html

将list.html放在emp文件夹下。

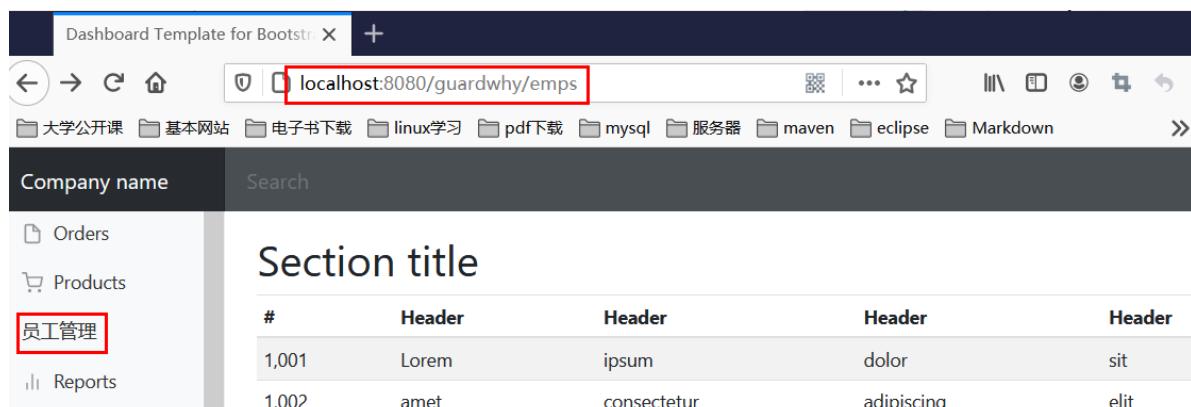
The screenshot shows the project structure in the left panel and the code editor in the right panel. The code editor is displaying the file 'list.html' under the 'src/main/resources/templates/emp' directory. The code contains an SVG logo and a navigation menu with an item for '员工管理' (Employee Management) which links to the '/emps' endpoint.

```
<circle cx="20" cy="21" r="1" />
<path d="M1 1h4l2.68 13.39a2 2 0
Products
</li>
<li class="nav-item">
<a class="nav-link" th:href="@{/emps}">
    员工管理
</a>
</li>
<li class="nav-item">
```

处理请求controller

```
1 package cn.guardwhy.controller;
2
3 import cn.guardwhy.dao.EmployeeDao;
4 import cn.guardwhy.pojo.Employee;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Controller;
7 import org.springframework.ui.Model;
8 import org.springframework.web.bind.annotation.RequestMapping;
9
10 import java.util.Collection;
11
12 @Controller
13 public class EmployeeController {
14     @Autowired
15     EmployeeDao employeeDao;
16
17     @RequestMapping("/emps")
18     public String list(Model model){
19         // 获得所有员工
20         Collection<Employee> employees = employeeDao.getAll();
21         model.addAttribute("emps", employees);
22         return "emp/list";
23     }
24 }
```

启动项目，项目能够跳转！！



7.5.2 公共页面元素抽取

侧边栏抽取

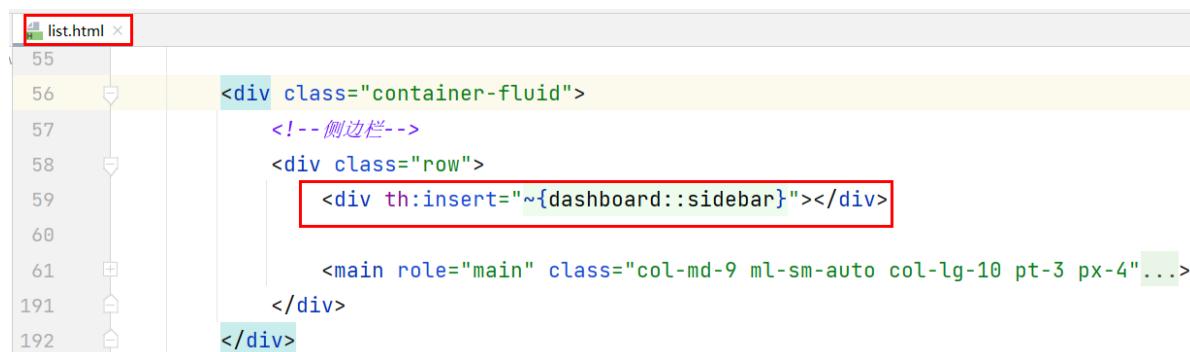
dashboard.html

```
1 <nav class="col-md-2 d-none d-md-block bg-light sidebar" th:fragment="sidebar"></nav>

class="container-fluid">
div class="row">
<!-- 侧边栏-->
<nav class="col-md-2 d-none d-md-block bg-light sidebar" th:fragment="sidebar">
    <div class="sidebar-sticky">
        <ul class="nav flex-column">
            <li class="nav-item">
```

list.html

```
1 <div class="row">
2     <div th:insert="~{dashboard::sidebar}"></div>
3 </div>
```



```
list.html x
55
56     <div class="container-fluid">
57         <!-- 侧边栏-->
58         <div class="row">
59             <div th:insert="~{dashboard::sidebar}"></div>
60
61             <main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4" ...>
62         </div>
63     </div>
```

顶部导航抽取

dashboard.html

```
1 <nav class="navbar navbar-dark sticky-top bg-dark flex-mdnowrap p-0" th:fragment="topbar">
2 </nav>
```



```
dashboard.html
45     <!-- 顶部导航栏-->
46     <nav class="navbar navbar-dark sticky-top bg-dark flex-mdnowrap p-0" th:fragment="topbar">
47         <a class="navbar-brand col-sm-3 col-md-2 mr-0" href="http://getbootstrap.com/docs/4.0/e
48             [[${session.loginUser}]]
49         </a>
50         <input class="form-control form-control-dark w-100" type="text" placeholder="Search" ar
51         <ul class="navbar-nav px-3">
52             <li class="nav-item text nowrap">
53                 <a class="nav-link" href="http://getbootstrap.com/docs/4.0/examples/dashboard/#"
```

list.html

```

1 <div th:insert="~{dashboard::topbar}"></div>
2 <div class="container-fluid">
3     <!--侧边栏-->
4     <div class="row">
5         <div th:insert="~{dashboard::sidebar}"></div>
6     </div>
7 </div>

```

```

46     <!--顶部栏-->
47     <div th:insert="~{dashboard::topbar}"></div>
48     <div class="container-fluid">
49         <!--侧边栏-->
50         <div class="row">
51             <div th:insert="~{dashboard::sidebar}"></div>
52
53         <main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4" ...>
54     </div>
183
184 </div>

```

启动项目，看效果！！

#	Header	Header	Header
1,001	Lorem	ipsum	dolor
1,002	amet	consectetur	adipiscing
1,003	Integer	nec	odio
1,003	libero	Sed	cursus

建立一个commons文件夹，专门存放公共页面

commons.html

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <!--顶部栏-->
    <nav class="navbar navbar-dark sticky-top bg-dark flex-mdnowrap p-0" th:fragment="topbar" ...>
        <!--侧边栏-->
        <nav class="col-md-2 d-none d-md-block bg-light sidebar" th:fragment="sidebar" ...>
    </html>

```

dashboard.html

```

<body>
    <div th:replace="~{commons/commons::topbar}"></div>
    <div class="container-fluid">
        <div class="row">
            <div th:replace="~{commons/commons::sidebar}"></div>
            <main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4" ...>
        </div>
    </div>

```

```
1 <div th:replace="~{commons/commons::topbar}"></div>
2 <div th:replace="~{commons/commons::sidebar}"></div>
```

list.html

```
44
45 <body>
46     <!-- 顶部栏-->
47     <div th:replace="~{commons/commons::topbar}"></div>
48     <div class="container-fluid">
49         <!-- 侧边栏-->
50         <div class="row">
51             <div th:replace="~{commons/commons::sidebar}"></div>
52             <main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4" ...>
53         </div>
54     </div>
55 
```

```
1 <div th:replace="~{commons/commons::topbar}"></div>
2 <div th:replace="~{commons/commons::sidebar}"></div>
```

侧边栏激活

存在问题: 侧边栏激活的问题, 总是激活第一个, 这应该是动态的。

dashboard.html

```
44 <body>
45     <div th:replace="~{commons/commons::topbar}"></div>
46     <div class="container-fluid">
47         <div class="row">
48             <!-- 传递参数给组件-->
49             <div th:replace="~{commons/commons::sidebar(active='main.html')}"></div>
50             <main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4" ...>
51         </div>
52     </div>
53 
```

```
1 <div th:replace="~{commons/commons::sidebar(active='main.html')}"></div>
```

list.html

```
44 <body>
45     <!-- 顶部栏-->
46     <div th:replace="~{commons/commons::topbar}"></div>
47     <div class="container-fluid">
48         <!-- 侧边栏-->
49         <div class="row">
50             <div th:replace="~{commons/commons::sidebar(active='list.html')}"></div>
51             <main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4" ...>
52         </div>
53     </div>
54 
```

```
1 <div th:replace="~{commons/commons::sidebar(active='list.html')}"></div>
```

commons.html

commons.html

```

18 <!-- 侧边栏 -->
19 <nav class="col-md-2 d-none d-md-block bg-light sidebar" th:fragment="sidebar">
20   <div class="sidebar-sticky">
21     <ul class="nav flex-column">
22       <li class="nav-item">
23         <a th:class="${active=='main.html'? 'nav-link active':'nav-link'}" th:href="@{/index.html}">
24           <svg xmlns="http://www.w3.org/2000/svg" width="24" height="24" viewBox="0 0 24 24" fill="none">
25             首页 <span class="sr-only">(current)</span>
26           </a>
27       </li>
28       <li class="nav-item" ...>
29       <li class="nav-item" ...>
30       <li class="nav-item">
31         <a th:class="${active=='list.html'? 'nav-link active':'nav-link'}" th:href="@{/emps}">
32           员工管理
33         </a>
34       </li>
35     </ul>
36   </div>
37 </nav>
38
39
40
41
42
43
44
45
46
47
48
49
50

```

```

1 <a th:class="${active=='main.html'? 'nav-link active':'nav-link'}"
2   th:href="@{/index.html}">
3 <a th:class="${active=='list.html'? 'nav-link active':'nav-link'}"
4   th:href="@{/emps}">

```

刷新页面，测试一下结果

#	Header
1,001	Lorem
1,002	amet
1,003	Integer
1,003	libero

员工信息页面展示

遍历员工信息

list.html

```

1 <main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4">
2   <h2>Section title</h2>
3   <div class="table-responsive">
4     <table class="table table-striped table-sm">
5       <thead>
6         <tr>
7           <th>id</th>
8           <th>lastName</th>
9           <th>email</th>
10          <th>gender</th>
11          <th>department</th>
12          <th>birth</th>
13        </tr>
14      </thead>
15      <tbody>
16        <tr th:each="emp:${emps}">
17          <td th:text="${emp.getId()}"></td>
18          <td th:text="${emp.getLastName()}"></td>
19          <td th:text="${emp.getEmail()}"></td>

```

```

20             <td th:text="${emp.getGender() == 0? '女': '男'}"></td>
21             <td th:text="${emp.department.getDepartmentName()}">
22             <td th:text="#${#dates.format(emp.getBirth(), 'yyyy-MM-dd
HH:mm:ss')}"></td>
23             <td class="btn btn-sm btn-primary">编辑</td>
24             <td class="btn btn-sm btn-danger">删除</td>
25         </tr>
26     </tbody>
27 </table>
28 </div>
29 </main>

```

执行结果

ID	lastName	email	gender	department	birth	操作
1001	kobe	1234678@qq.com	男	后勤部	2021-05-11 17:55:55	编辑 删除
1002	james	1234678@qq.com	男	市场部	2021-05-11 17:55:55	编辑 删除
1003	xiao hong	1234678@qq.com	女	教学部	2021-05-11 17:55:55	编辑 删除
1004	harden	1234678@qq.com	男	运营部	2021-05-11 17:55:55	编辑 删除
1005	guardwhy	1234678@qq.com	男	教研部	2021-05-11 17:55:55	编辑 删除

7.6 添加员工实现

7.6.1 表单优化

1、将添加员工信息改为超链接

list.html

```

1 | <h2><a class="btn btn-sm btn-success" th:href="@{/emp}">添加员工</a></h2>

```

2、添加 add.html 前端页面，复制 list.html 页面，修改即可！！！

```

1 | <form>
2 |   <div class="form-group">
3 |     <label>LastName</label>
4 |     <input type="text" name="lastName" class="form-control"
placeholder="guardwhy">
5 |   </div>
6 |   <div class="form-group">
7 |     <label>Email</label>
8 |     <input type="email" name="email" class="form-control"
placeholder="1625309592@qq.com">
9 |   </div>
10 |  <div class="form-group">

```

```

11     <label>Gender</label><br/>
12     <div class="form-check form-check-inline">
13         <input class="form-check-input" type="radio" name="gender"
14             value="1">
15         <label class="form-check-label">男</label>
16     </div>
17     <div class="form-check form-check-inline">
18         <input class="form-check-input" type="radio" name="gender"
19             value="0">
20         <label class="form-check-label">女</label>
21     </div>
22     <div class="form-group">
23         <label>department</label>
24         <select class="form-control" name="department">
25             <option>1</option>
26             <option>2</option>
27             <option>3</option>
28             <option>4</option>
29             <option>5</option>
30         </select>
31     </div>
32     <div class="form-group">
33         <label>Birth</label>
34         <input type="text" name="birth" class="form-control"
35             placeholder="kuangstudy">
36     </div>
37     <button type="submit" class="btn btn-primary">添加</button>
38 </form>

```

7.6.2 添加员工

员工添加页面

1、EmployeeController

```

1 @GetMapping("/emp")
2 public String toAddpage(Model model){
3     // 查询所有部门的信息
4     Collection<Department> departments = departmentDao.getDepartments();
5     model.addAttribute("departments", departments);
6     return "emp/add";
7 }

```

2、前端页面修改

```

1 <div class="form-group">
2     <label>department</label>
3     <!--控制器Controller接收的是Employee(员工), 所以需要提交其中的一个属性! ! -->
4     <select class="form-control" name="department.id">
5         <option th:each="dept:${departments}"
6             th:text="${dept.getDepartmentName()}" th:value="${dept.getId()}"/></option>
7     </select>
8 </div>

```

```
72 <div class="form-group">
73   <label>department</label>
74   <!-- 控制器Controller接收的是Employee(员工), 所以需要提交其中的一个属性! ! -->
75   <select class="form-control" name="department.id">
76     <option th:each="dept:${departments}" th:text="${dept.getDepartmentName()}" th:value="${dept.getId()}"/>
77   </select>
78 </div>
```

执行结果

Dashboard Template for Boot < +

localhost:8080/guardwhy/emp

011318200545 Search

首页 Orders Products 员工管理 Reports Integrations SAVED REPORTS + Current month Last quarter Social engagement Year-end sale

Last Name: Durant
Email: hxy13479915208@gmail.com
Gender: 男
department: 教研部
Birth: 2018-01-11

添加

员工添加功能

3、EmployeeController

```
1 // 3. 员工添加功能
2 @PostMapping("/emp")
3 public String addEmp(Employee employee){
4     // 3.1 调用底层方法保存员工信息
5     employeeDao.add(employee);
6     // 3.2 返回员工列表页面
7     return "redirect:/emps";
8 }
```

4、修改aform表单提交方式

```
1 <form th:action="@{/emp}" method="post"></form>
```

自定义日期格式化

5、application.properties

```
1 # 时间格式化
2 spring.mvc.format.date=yyyy-MM-dd
```

执行结果

The screenshot shows a web application interface for managing employees. On the left is a sidebar with navigation links like '首页', 'Orders', 'Products', '员工管理' (highlighted), 'Reports', 'Integrations', 'SAVED REPORTS', and 'Current month'. The main content area has a title '添加员工' (Add Employee) and a table listing employees. A red box highlights the last row of the table, which contains the new employee information: id=1006, lastName=Durant, email=hxy13479915208@gmail.com, gender=男 (Male), department=教研部, birth=2018-01-11 00:00:00. A red arrow points from this row to a red box containing the text '添加成功' (Added successfully).

ID	Last Name	Email	Gender	Department	Birth	操作
1001	kobe	1234678@qq.com	男	后勤部	2021-05-11 21:21:05	编辑 删除
1002	james	1234678@qq.com	男	市场部	2021-05-11 21:21:05	编辑 删除
1003	xiao hong	1234678@qq.com	女	教学部	2021-05-11 21:21:05	编辑 删除
1004	harden	1234678@qq.com	男	运营部	2021-05-11 21:21:05	编辑 删除
1005	guardwhy	1234678@qq.com	男	教研部	2021-05-11 21:21:05	编辑 删除
1006	Durant	hxy13479915208@gmail.com	男	教研部	2018-01-11 00:00:00	编辑 删除

7.7 员工信息修改

7.7.1 修改页面

1、首页编辑跳转链接

list.html

```

1 <td>
2     <a class="btn btn-sm btn-primary" th:href="@{/emp/{emp.id}}>编辑</a>
3 </td>

```

2、去员工修改页面controller

EmployeeController

```

1 // 员工修改页面
2 @GetMapping("/emp/{id}")
3 public String toUpdateEmp(@PathVariable("id") Integer id, Model model){
4     // 4.1 查出原来的数据
5     Employee employee = employeeDao.getEmployeeById(id);
6     model.addAttribute("emp", employee);
7     // 4.2 查出所有部门的信息
8     Collection<Department> departments = departmentDao.getDepartments();
9     model.addAttribute("departments", departments);
10    // 4.3 返回数据
11    return "emp/update";
12 }

```

7.7.2 查询数据回显

update.html

```

1 <!--提交方式-->
2 <form th:action="@{/updateEmp}" method="post">
3     <div class="form-group">
4         <label>LastName</label>
5         <input th:value="${emp.getLastname()}" type="text" name="lastName"
6 class="form-control" placeholder="guardwhy">
7     </div>
8     <div class="form-group">
9         <label>Email</label>
10        <input th:value="${emp.getEmail()}" type="email" name="email"
11 class="form-control" placeholder="1625309592@qq.com">
12     </div>
13     <div class="form-group">

```

```

12         <label>Gender</label><br/>
13         <div class="form-check form-check-inline">
14             <input th:checked="${emp.getGender() == 1}" class="form-check-input" type="radio" name="gender" value="1">
15                 <label class="form-check-label">男</label>
16             </div>
17             <div class="form-check form-check-inline">
18                 <input th:checked="${emp.getGender() == 0}" class="form-check-input" type="radio" name="gender" value="0">
19                     <label class="form-check-label">女</label>
20                 </div>
21         </div>
22         <div class="form-group">
23             <label>department</label>
24             <!--控制器Controller接收的是Employee(员工), 所以需要提交其中的一个属性! ! -->
25             <select class="form-control" name="department.id">
26                 <option
27                     th:selected="${dept.getId() == emp.getDepartment().getId()}"
28                     th:each="dept:${departments}" th:text="${dept.getDepartmentName()}"
29                     th:value="${dept.getId()}></option>
30                 </select>
31             </div>
32             <div class="form-group">
33                 <label>Birth</label>
34                 <!--日期格式化-->
35                 <input th:value="#{#dates.format(emp.getBirth(), 'yyyy-MM-dd HH:mm')}" type="text" name="birth" class="form-control"
placeholder="guardwhuy">
36             </div>
37             <button type="submit" class="btn btn-primary">修改</button>
38         </form>

```

3、回到员工列表页面

```

1 // 回到员工列表页面
2 @PostMapping("/updateEmp")
3 public String updateEmp(Employee employee){
4     // 5.1 添加操作
5     employeeDao.add(employee);
6     return "redirect:/emps";
7 }

```

执行结果

	id	lastName	email	gender	department	birth	编辑	删除
1001	kobe	1234678@qq.com	男	后勤部	2021-05-12 00:09:59	编辑	删除	
1002	james	1234678@qq.com	男	市场部	2021-05-12 00:09:59	编辑	删除	
1003	li lin	12335679@qq.com	女	后勤部	2017-03-12 00:00:00	编辑	删除	
1004	harden	1234678@qq.com	男	运营部	2021-05-12 00:09:59	编辑	删除	
1005	guardwhuy	1234678@qq.com	男	教研部	2021-05-12 00:09:59	编辑	删除	

4、发现页面提交的没有id，在前端加一个隐藏域，提交id。

```

1 | <!--隐藏域-->
2 | <input th:type="hidden" name="id" th:value="${emp.getId()}">

```

The screenshot shows a table of employees with columns: id, lastName, email, gender, department, and birth. A red box highlights the row for employee ID 1003, Jordan. An arrow points from the '修改成功' (Modification successful) message at the top right towards this row.

	id	lastName	email	gender	department	birth	编辑	删除
1001	kobe	1234678@qq.com	男	后勤部	2021-05-12 00:09:59	编辑	删除	
1002	james	1234678@qq.com	男	市场部	2021-05-12 00:09:59	编辑	删除	
1003	Jordan	12335679@qq.com	男	市场部	2007-06-11 00:00:00	编辑	删除	
1004	harden	1234678@qq.com	男	运营部	2021-05-12 00:09:59	编辑	删除	
1005	guardwhy	1234678@qq.com	男	教研部	2021-05-12 00:09:59	编辑	删除	

7.8 删除员工

1、list.html 页面，编写提交地址

```

1 | <!--删除链接-->
2 | <a class="btn btn-sm btn-danger" th:href="@{/delemp/}+${emp.getId()}">删除</a>

```

2、编写删除员工Controller

```

1 | // 删除员工
2 | @GetMapping("/delemp/{id}")
3 | public String deleteEmp(@PathVariable("id")int id){
4 |     // 调用底层
5 |     employeedao.delete(id);
6 |     return "redirect:/emps";
7 |
}

```

执行结果

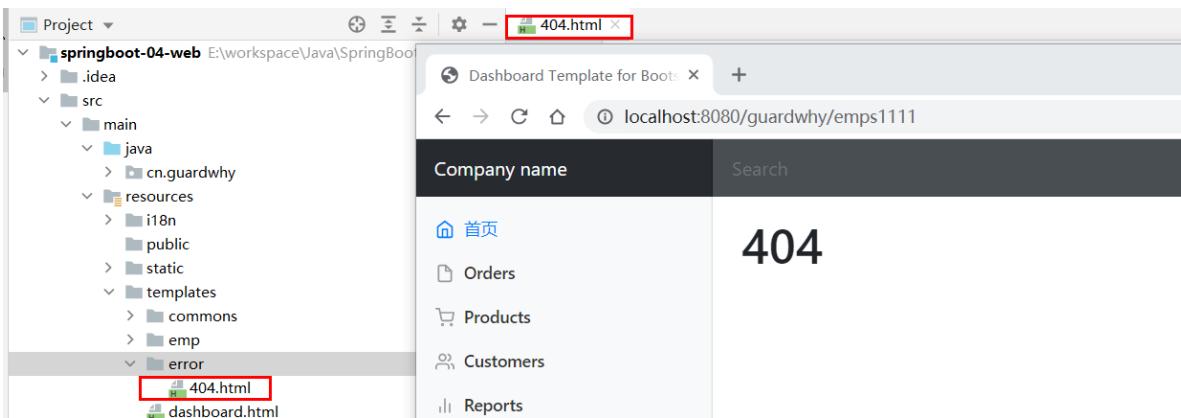
The screenshot shows the same employee list as before, but the row for employee ID 1003, Jordan, is no longer present, indicating it has been successfully deleted.

7.9 404 or 注销用户

7.9.1 404 网页

- 在模板目录下添加一个error文件夹，文件夹中存放我们相应的错误页面
- 比如404.html 或者 4xx.html 等等，SpringBoot就会自动使用了。

执行结果



7.9.2 注销用户

1、注销请求

commons.html

```

1 <li class="nav-item text-nowrap">
2   <a class="nav-link" th:href="@{/user/logout}">注销</a>
3 </li>

```

2、注销用户controller

```

1 // 注销用户
2 @RequestMapping("/user/logout")
3 public String logout(HttpServletRequest session){
4   // 7.1 注销
5   session.invalidate();
6   // 7.2 返回到首页
7   return "redirect:/index.html";
8 }

```

3、注销成功，返回主页面！！！

8-SpringData

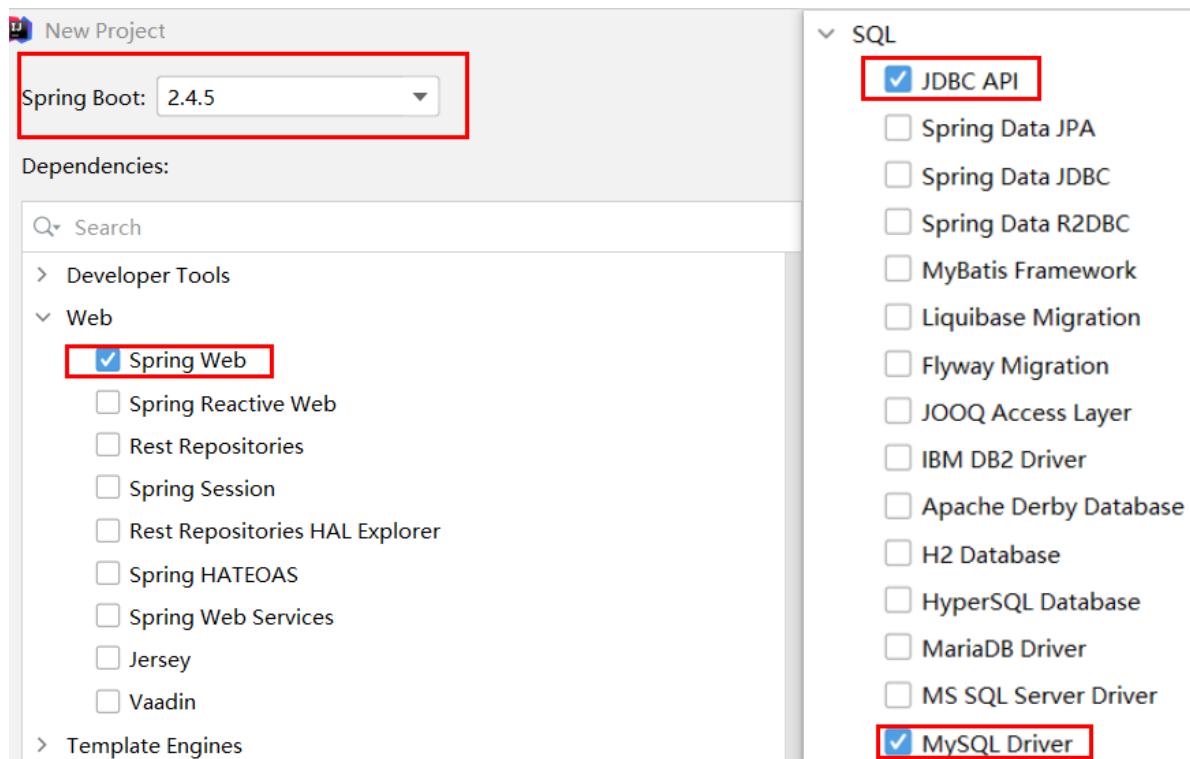
8.1 SpringBoot集成JDBC

对于数据访问层，无论是 SQL(关系型数据库) 还是 NOSQL(非关系型数据库)，Spring Boot 底层都是采用 Spring Data 的方式进行统一处理。

Spring Boot 底层都是采用 Spring Data 的方式进行统一处理各种数据库，Spring Data 也是 Spring 中与 Spring Boot 齐名的知名项目。

8.1.1 环境准备

1、新建一个项目测试：springboot-05-data引入相应的模块！基础模块



2、导入测试数据库

```
1 -- 创建数据库springboot_mysql
2 create database if not exists springboot_mybatis;
3
4 -- 使用数据库springboot_mybatis
5 use springboot_mybatis
6
7 -- 创建user数据表
8 create table user(
9     -- 字段名,字段类型
10    id int primary key,
11    name varchar(20),
12    pwd varchar(20)
13 );
14
15 -- 向表中插入所有字段
16 insert into user values(1, 'curry', '123667'),(2, 'kobe', '12366dd'),(3,
17 'harden', '666624');
18 -- 查询数据表
19 select * from user;
```

3、项目建好之后，发现自动导入了如下的启动器

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-jdbc</artifactId>
4 </dependency>
5 <dependency>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-web</artifactId>
8 </dependency>
9
10 <dependency>
11     <groupId>mysql</groupId>
```

```
12     <artifactId>mysql-connector-java</artifactId>
13     <scope>runtime</scope>
14 </dependency>
15 <dependency>
16     <groupId>org.springframework.boot</groupId>
17     <artifactId>spring-boot-starter-test</artifactId>
18     <scope>test</scope>
19 </dependency>
```

4、编写yaml配置文件连接数据库

```
1 spring:
2   datasource:
3     username: root
4     password: root
5     # 假如时区报错了，就增加了一个时区的配置就ok了serverTimezone=UTC
6     url: jdbc:mysql://localhost:3306/springboot_mybatis?
7       serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
8     driver-class-name: com.mysql.cj.jdbc.Driver
```

5、测试project，我们可以直接去使用，因为SpringBoot已经默认帮我们进行了自动配置。

```
1 package cn.guardwhy;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.boot.test.context.SpringBootTest;
6
7 import javax.sql.DataSource;
8 import java.sql.Connection;
9 import java.sql.SQLException;
10 import java.sql.SQLOutput;
11
12 @SpringBootTest
13 class Springboot05DataApplicationTests {
14     @Autowired
15     DataSource dataSource;
16
17     @Test
18     void contextLoads() throws SQLException {
19         // 1.查看一下默认的数据源: com.zaxxer.hikari.HikariDataSource
20         System.out.println(dataSource.getClass());
21         // 2.获得数据库连接
22         Connection connection = dataSource.getConnection();
23         System.out.println(connection);
24         // 3.关闭
25         connection.close();
26     }
27 }
```

执行结果

可以看到默认配置的数据源为：`class com.zaxxer.hikari.HikariDataSource`，我们并没有手动配置。

全局搜索找到数据源的自动配置：`DataSourceAutoConfiguration`文件。

```

1  @Configuration(proxyBeanMethods = false)
2  @ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
3  @ConditionalOnMissingBean(type = "io.r2dbc.spi.ConnectionFactory")
4  @EnableConfigurationProperties(DataSourceProperties.class)
5  @Import({ DataSourcePoolMetadataProvidersConfiguration.class,
6          DataSourceInitializationConfiguration.class })
7  public class DataSourceAutoConfiguration {
8
9      @Configuration(proxyBeanMethods = false)
10     @Conditional(EmbeddedDatabaseCondition.class)
11     @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
12     @Import(EmbeddedDataSourceConfiguration.class)
13     protected static class EmbeddedDatabaseConfiguration {
14
15
16         @Configuration(proxyBeanMethods = false)
17         @Conditional(PooledDataSourceCondition.class)
18         @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
19         @Import({ DataSourceConfiguration.Hikari.class,
20                  DataSourceConfiguration.Tomcat.class,
21                  DataSourceConfiguration.Dbcp2.class,
22                  DataSourceConfiguration.OracleUcp.class,
23                  DataSourceConfiguration.Generic.class,
24                  DataSourceJmxConfiguration.class })
25         protected static class PooledDataSourceConfiguration {

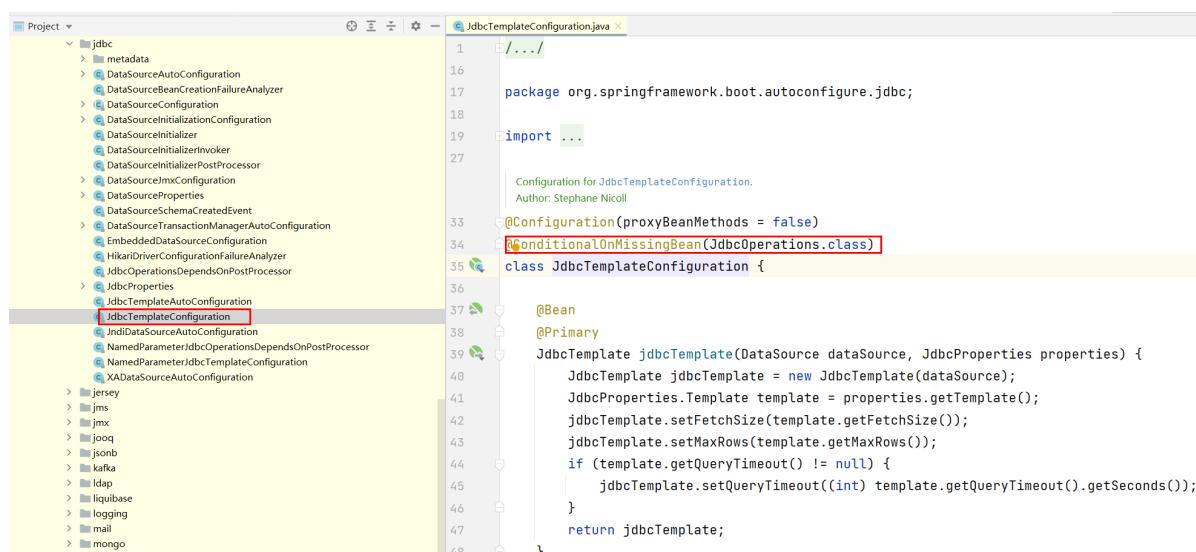
```

这里导入的类都在 `DataSourceConfiguration` 配置类下，可以看出 Spring Boot 2.4.5 默认使用 `HikariDataSource` 数据源。

`HikariDataSource` 号称 Java WEB 当前速度最快的数据源，相比于传统的 C3P0、`jdbc` 连接池更加优秀。

8.1.2 JdbcTemplate

基本特点



```

1  package org.springframework.boot.autoconfigure.jdbc;
2
3  import org.springframework.context.annotation.Configuration;
4  import org.springframework.context.annotation.Primary;
5  import org.springframework.jdbc.core.JdbcOperations;
6  import org.springframework.jdbc.core.JdbcTemplate;
7  import org.springframework.jdbc.core.namedparam.NamedParameterJdbcOperations;
8  import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
9  import org.springframework.jdbc.datasource.DataSourceUtils;
10 import org.springframework.jdbc.datasource.DriverManagerDataSource;
11 import org.springframework.jdbc.support.JdbcOperationsDependsOnPostProcessor;
12 import org.springframework.jdbc.support.JdbcProperties;
13 import org.springframework.jdbc.support.JdbcTemplate;
14 import org.springframework.jdbc.support.JdbcTemplateAutoConfiguration;
15 import org.springframework.jdbc.support.JndiDataSourceAutoConfiguration;
16 import org.springframework.jdbc.support.NamedParameterJdbcOperationsDependsOnPostProcessor;
17 import org.springframework.jdbc.support.NamedParameterJdbcTemplateConfiguration;
18 import org.springframework.jdbc.support.XADataSourceAutoConfiguration;
19 import org.springframework.transaction.TransactionManager;
20 import org.springframework.transaction.jta.XADataSource;
21
22 /**
23  * Configuration for JdbcTemplateConfiguration.
24  * Author: Stephane Nicoll
25  */
26 @Configuration(proxyBeanMethods = false)
27 @ConditionalOnMissingBean(JdbcOperations.class)
28 class JdbcTemplateConfiguration {
29
30     @Bean
31     @Primary
32     JdbcTemplate jdbcTemplate(DataSource dataSource, JdbcProperties properties) {
33         JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
34         JdbcProperties.Template template = properties.getTemplate();
35         jdbcTemplate.setFetchSize(template.getFetchSize());
36         jdbcTemplate.setMaxRows(template.getMaxRows());
37         if (template.getQueryTimeout() != null) {
38             jdbcTemplate.setQueryTimeout((int) template.getQueryTimeout().getSeconds());
39         }
40         return jdbcTemplate;
41     }
42 }

```

- 有了数据源(`com.zaxxer.hikari.HikariDataSource`)，可以直接拿到数据库连接对象(`java.sql.Connection`)，使用原生的 JDBC 语句来操作数据库。

- Spring 对原生的JDBC 做了轻量级的封装， JdbcTemplate就是封装后的产物， 数据库操作的所有CRUD 方法都在 JdbcTemplate 中。
- Spring Boot 不仅提供了默认的数据源， 同时已经配置好了 JdbcTemplate 放在了容器中。
- JdbcTemplate 的自动配置是依赖 ·org.springframework.boot.autoconfigure.jdbc 包下的 JdbcTemplateConfiguration 类。

案例测试

编写一个Controller，注入 jdbcTemplate，编写测试方法。

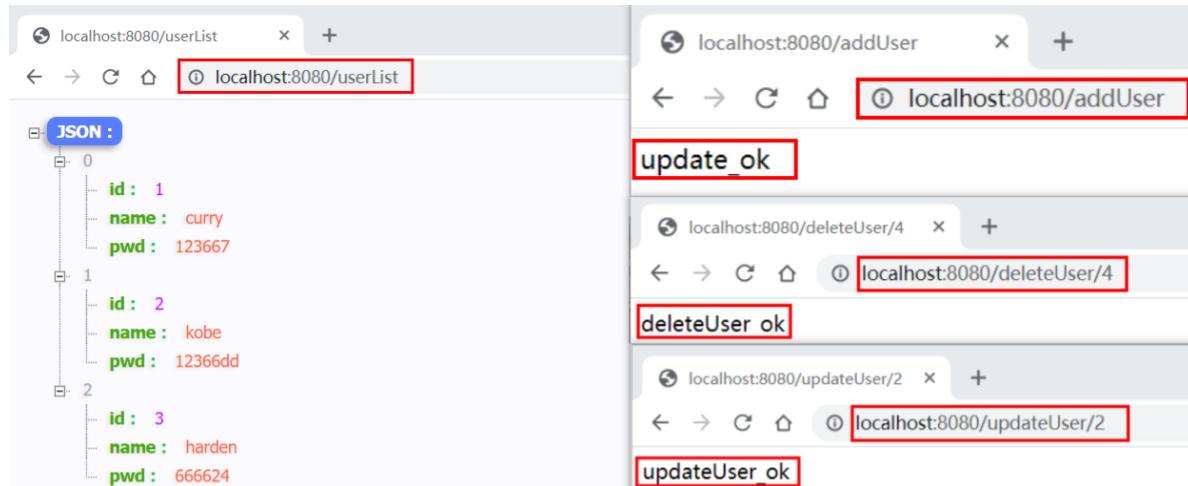
```

1 package cn.guardwhy.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.jdbc.core.JdbcTemplate;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.PathVariable;
7 import org.springframework.web.bind.annotation.RestController;
8
9 import java.util.List;
10 import java.util.Map;
11
12 @RestController
13 public class JDBCController {
14     // 注入jdbcTemplate
15     @Autowired
16     JdbcTemplate jdbcTemplate;
17     // 1.查询数据中的所有信息，没有实体类数据库的数据通过Map获取
18     @GetMapping("/userList")
19     public List<Map<String, Object>> userList(){
20         String sql = "select * from user";
21         List<Map<String, Object>> list_maps =
22         jdbcTemplate.queryForList(sql);
23         return list_maps;
24     }
25
26     // 2.添加数据
27     @GetMapping("/addUser")
28     public String addUser(){
29         String sql = "insert into user (id, name, pwd) values(4, 'guardwhy',
30         '123666')";
31         jdbcTemplate.update(sql);
32         return "update_ok";
33     }
34
35     // 3.通过id修改数据
36     @GetMapping("/updateUser/{id}")
37     public String updateUser(@PathVariable("id") int id){
38         String sql = "update user set name=?,pwd=? where id="+id;
39         // 3.1 封装数据
40         Object[] objects = new Object[2];
41         objects[0] = "james";
42         objects[1] = "mvp";
43         jdbcTemplate.update(sql, objects);
44         return "updateUser_ok";
45     }
46
47     // 4.通过id删除数据

```

```
46     @GetMapping("/deleteUser/{id}")
47     public String deleteUser(@PathVariable("id") int id){
48         String sql = "delete from user where id=?";
49         jdbcTemplate.update(sql, id);
50         return "deleteUser_ok";
51     }
52 }
```

执行结果



8.2 SpringBoo集成Druid

8.2.1 Druid 基本介绍

- Druid 是阿里巴巴开源平台上一个数据库连接池实现，结合了 C3P0、DBCP 等 DB 池的优点，同时加入了日志监控。
- Druid 可以很好的监控 DB 池连接和 SQL 的执行情况，天生就是针对监控而生的 DB 连接池。
- Spring Boot 2.0 以上默认使用 HikariDataSource 数据源，可以说 HikariDataSource 与 Druid 都是当前 Java Web 上最优秀的数据源。

Druid Github地址：<https://github.com/alibaba/druid/>

DruidDataSource 基本配置参数

配置	缺省值	具体作用
name		配置这个属性的意义在于，如果存在多个数据源，监控的 <br/时候可以通过名字来区分开来。如果没有配置，将会生成一个名字，格式是："DataSource-" +System.identityHashCode(this)
url		连接数据库的url，不同数据库不一样。例如： mysql :jdbc:mysql://10.20.153.104:3306/druid2 oracle :jdbc:oracle:thin:@10.20.149.85:1521:ocnauto
username		连接数据库的用户名
password		连接数据库的密码。如果你不希望密码直接写在配置文件中，可以使用ConfigFilter。
driverClassName	根据url自动识别	这一项可配可不配，如果不配置druid会根据url自动识别dbType，然后选择相应的driverClassName。
initialSize	0	初始化时建立物理连接的个数。初始化发生在显示调用init方法，或者第一次getConnection时。
maxActive	8	最大连接池数量。
maxIdle	8	已经不再使用，配置了也没效果。
minIdle		最小连接池数量。
maxWait		获取连接时最大等待时间，单位毫秒。配置了maxWait之后，缺省启用公平锁，并发效率会有所下降，如果需要可以通过配置useUnfairLock属性为true使用非公平锁。
poolPreparedStatements	false	是否缓存PreparedStatement，也就是PSCache。PSCache对支持游标的数据库性能提升巨大，比如说oracle。在mysql下建议关闭。
validationQuery		用来检测连接是否有效的sql，要求是一个查询语句。如果validationQuery为null，testOnBorrow、testOnReturn、testWhileIdle都不会起作用。
validationQueryTimeout		单位：秒，检测连接是否有效的超时时间。底层调用jdbc Statement对象的void setQueryTimeout(int seconds)方法。
testOnBorrow	true	申请连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。
maxOpenPreparedStatements	-1	要启用PSCache，必须配置大于0，当大于0时，poolPreparedStatements自动触发修改为true。在Druid中，不会存在Oracle下PSCache占用内存过多的问题，可以把这个数值配置大一些，比如说100
testOnReturn	false	归还连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。
testWhileIdle	false	建议配置为true，不影响性能，并且保证安全性。申请连接的时候检测，如果空闲时间大于timeBetweenEvictionRunsMillis，执行validationQuery检测连接是否有效。

配置	缺省值	具体作用
timeBetweenEvictionRunsMillis	1分钟 (1.0.14)	有两个含义： 1) Destroy线程会检测连接的间隔时间，如果连接空闲时间大于等于minEvictableIdleTimeMillis则关闭物理连接。 2) testWhileIdle的判断依据，详细看testWhileIdle属性的说明
numTestsPerEvictionRun		不再使用，一个DruidDataSource只支持一个EvictionRun
minEvictableIdleTimeMillis	30分钟 (1.0.14)	连接保持空闲而不被驱逐的最长时间。
connectionInitSqls		物理连接初始化的时候执行的sql
exceptionSorter	根据 dbType 自动识别	当数据库抛出一些不可恢复的异常时，抛弃连接。
filters		属性类型是字符串，通过别名的方式配置扩展插件，常用的插件有：监控统计用的filter:stat 日志用的filter:log4j，防御sql注入的filter:wall。
proxyFilters		类型是List<com.alibaba.druid.filter.Filter>，如果同时配置了filters和proxyFilters，是组合关系，并非替换关系。

配置数据源

1、添加上 Druid 数据源和log4j的依赖

```

1  <!-- https://mvnrepository.com/artifact/log4j/log4j -->
2  <dependency>
3      <groupId>log4j</groupId>
4      <artifactId>log4j</artifactId>
5      <version>1.2.17</version>
6  </dependency>
7  <!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
8  <dependency>
9      <groupId>com.alibaba</groupId>
10     <artifactId>druid</artifactId>
11     <version>1.2.6</version>
12 </dependency>
```

2、但可以通过 spring.datasource.type 指定数据源。

```

1  spring:
2      datasource:
3          username: root
4          password: root
5          url: jdbc:mysql://localhost:3306/springboot_mybatis?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
6          driver-class-name: com.mysql.cj.jdbc.Driver
7          <!--指定数据源-->
8          type: com.alibaba.druid.pool.DruidDataSource
```

执行结果

```

Run: Rerun Failed Tests
guardwhy (cn) 841 ms
  Springboot 841 ms
    context 841 ms
2021-05-13 15:38:43.139 INFO 8416 --- [main] c.g.Springboot05DataApplicationTests : No active profile set, falling back to default profiles: default
2021-05-13 15:38:44.125 INFO 8416 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService [pool-1]
2021-05-13 15:38:44.397 INFO 8416 --- [main] c.g.Springboot05DataApplicationTests : Started Springboot05DataApplication in 0.238 seconds (JVM: 0.238s)
class com.alibaba.druid.pool.DruidDataSource
2021-05-13 15:38:44.747 INFO 8416 --- [main] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} initied
com.mysql.cj.jdbc.ConnectionImpl@58f39564
2021-05-13 15:38:45.259 INFO 8416 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService [pool-1]
2021-05-13 15:38:45.268 INFO 8416 --- [extShutdownHook] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} closing
2021-05-13 15:38:45.263 INFO 8416 --- [extShutdownHook] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} closed

```

4、设置数据源连接初始化大小、最大连接数、等待时间、最小连接数等设置项。

```

1 spring:
2   datasource:
3     username: root
4     password: root
5     url: jdbc:mysql://localhost:3306/springboot_mybatis?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
6     driver-class-name: com.mysql.cj.jdbc.Driver
7     type: com.alibaba.druid.pool.DruidDataSource
8
9     #druid 数据源专有配置
10    initialSize: 5
11    minIdle: 5
12    maxActive: 20
13    maxWait: 60000
14    timeBetweenEvictionRunsMillis: 60000
15    minEvictableIdleTimeMillis: 300000
16    validationQuery: SELECT 1 FROM DUAL
17    testWhileIdle: true
18    testOnBorrow: false
19    testOnReturn: false
20    poolPreparedStatements: true
21    #配置监控统计拦截的filters, stat: 监控统计、log4j: 日志记录、wall: 防御sql注入
22    #如果允许时报错 java.lang.ClassNotFoundException:org.apache.log4j.Priority
23    #则导入 log4j 依赖即可, Maven 地址:
24    https://mvnrepository.com/artifact/log4j/log4j
25    filters: stat,wall,log4j
26    maxPoolPreparedStatementPerConnectionSize: 20
27    useGlobalDataSourceStat: true
      connectionProperties:
        druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500

```

8.2.2 具体使用

1、将DruidDataSource组件添加到容器中，并绑定属性。

```

1 package cn.guardwhy.config;
2
3 import com.alibaba.druid.pool.DruidDataSource;
4 import org.springframework.boot.context.properties.ConfigurationProperties;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7
8 import javax.sql.DataSource;
9
10 // 将自定义的Druid数据源添加到容器中
11 @Configuration
12 public class DruidConfig {

```

```

13     //将全局配置文件中前缀为 spring.datasource的属性值注入到
14     com.alibaba.druid.pool.DruidDataSource的同名参数中
15     @ConfigurationProperties(prefix = "spring.datasource")
16     @Bean
17     public DataSource druidDataSource(){
18         return new DruidDataSource();
19     }

```

2、测试代码

```

1 package cn.guardwhy;
2
3 import com.alibaba.druid.pool.DruidDataSource;
4 import org.junit.jupiter.api.Test;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.boot.test.context.SpringBootTest;
7
8 import javax.sql.DataSource;
9 import java.sql.Connection;
10 import java.sql.SQLException;
11 import java.sql.SQLOutput;
12
13 @SpringBootTest
14 class Springboot05DataApplicationTests {
15     @Autowired
16     DataSource dataSource;
17
18     @Test
19     void contextLoads() throws SQLException {
20         // 1.查看一下默认的数据源
21         System.out.println(dataSource.getClass());
22         // 2.获得数据库连接
23         Connection connection = dataSource.getConnection();
24         System.out.println(connection);
25         // 3.获得连接
26         DruidDataSource druidDataSource = (DruidDataSource) dataSource;
27         System.out.println("druidDataSource 数据源最大连接数:" +
28             druidDataSource.getMaxActive());
29         System.out.println("druidDataSource 数据源初始化连接数:" +
30             druidDataSource.getInitialSize());
31
32         // 4.关闭
33         connection.close();
34     }
35 }

```

执行结果



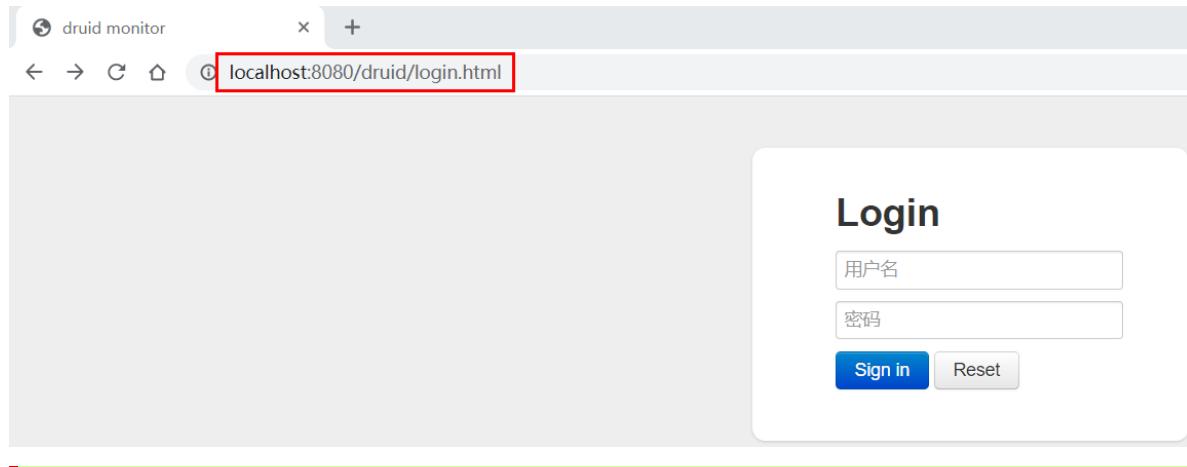
8.2.3 Druid 数据源监控

Druid 数据源具有监控的功能，并提供了一个 web 界面方便用户查看，Druid也提供了一个默认的页面。

设置 Druid 的后台管理页面，比如 登录账号、密码，配置后台管理。

```
1 // 后台监控
2 @Bean
3 public ServletRegistrationBean statViewServlet(){
4     ServletRegistrationBean bean = new ServletRegistrationBean(new
5
6     StatViewServlet(), "/druid/*");
7     // 后台需要有人登陆，账号密码配置
8     HashMap<String, String> initParameters = new HashMap<>();
9     // 3.增加配置
10    initParameters.put("loginUsername", "admin"); // 后台管理界面的登录账号
11    initParameters.put("loginPassword", "123456"); // 后台管理界面的登录密码
12    // 4.允许谁可以访问
13    initParameters.put("allow", "");
14
15    // 表示禁止该ip地址访问
16    // initParameters.put("guardwhy", "192.168.3.8");
17
18    // 5.设置初始化参数
19    bean.setInitParameters(initParameters);
20    return bean;
21 }
```

配置完毕后，直接访问: <http://localhost:8080/druid/index.html>



WebStatFilter：用于配置Web和Druid数据源之间的管理关联监控统计。

```

1 // 过滤器
2 @Bean
3 public FilterRegistrationBean webstatFilter() {
4     FilterRegistrationBean bean = new FilterRegistrationBean();
5     bean.setFilter(new WebStatFilter());
6     // 可以过滤哪些请求
7     Map<String, String> initParams = new HashMap<>();
8     // 这些东西不进行统计
9     initParams.put("exclusions", "*.js,*.css,/druid/*");
10    bean.setInitParameters(initParams);
11    return bean;
12 }

```

执行结果

The screenshot shows the 'Druid Monitor' interface with the 'SQL监控' tab selected. It displays a table of SQL statements with their execution details. Two specific queries are highlighted with red boxes: 'select * from user' at row 1 and 'INSERT INTO user(id, name)' at row 2.

N	SQL ▾	监控操作	执行数	执行时间	最慢	事务执行	错误数	更新行数	读取行数	执行中	最大并发	执行时间分布	执行+RS时分布	读取行分布	更新行分布
1	select * from user		1	13	13				4	1	[0,0,1,0,0,0,0]	[1,0,0,0,0,0,0]	[0,1,0,0,0,0]	[1,0,0,0,0,0]	
2	INSERT INTO user(id, name)		1	74	74			1		1	[0,0,1,0,0,0,0]	[0,0,1,0,0,0,0]	[1,0,0,0,0]	[0,1,0,0,0]	

8.3 SpringBoot整合MyBatis

8.3.1 整合测试

官方文档: <http://mybatis.org/spring-boot-starter/mybatis-spring-boot-autoconfigure/>

The screenshot shows the official documentation for 'mybatis-spring-boot-autoconfigure'. The left sidebar has a 'REFERENCE DOCUMENTATION' menu with sections like 'Introduction', 'Project Documentation', 'Project Information', 'CI Management', 'Dependencies', 'Dependency Information', 'Dependency Management', 'Distribution Management', and 'About'. The main content area starts with the 'Introduction' section, which includes a sub-section 'What is MyBatis-Spring-Boot-Starter?'. Below that is a paragraph about the module's purpose and a bulleted list of its benefits. Further down is the 'Requirements' section, which lists the required versions for the MyBatis-Spring-Boot-Starter, MyBatis-Spring, Spring Boot, and Java.

MyBatis-Spring-Boot-Starter	MyBatis-Spring	Spring Boot	Java
2.1	2.0 (need 2.0.2+ for enable all features)	2.1 or higher	8 or higher

Maven仓库地址: <https://mvnrepository.com/artifact/org.mybatis.spring.boot/mybatis-spring-boot-starter/2.1.4>

导入所需要的依赖

```

1 <!-- https://mvnrepository.com/artifact/org.mybatis.spring.boot/mybatis-
2   spring-boot-starter -->
3 <dependency>
4   <groupId>org.mybatis.spring.boot</groupId>
5   <artifactId>mybatis-spring-boot-starter</artifactId>

```

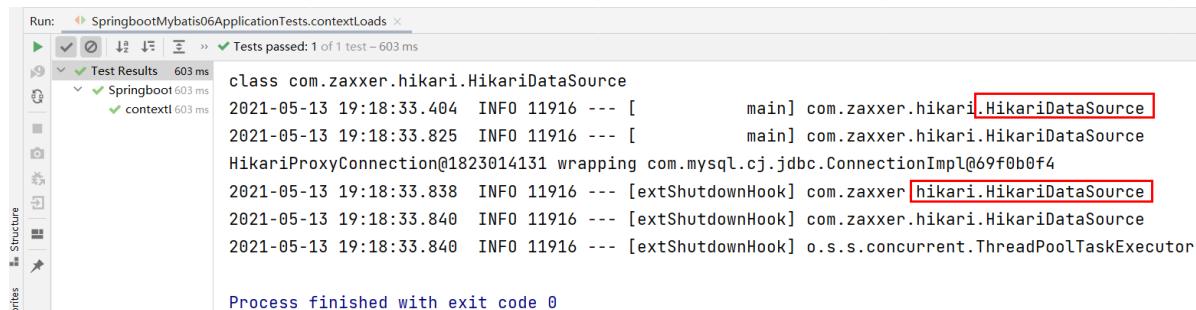
```
5 <version>2.1.4</version>
6 </dependency>
7 <dependency>
8     <groupId>org.mybatis</groupId>
9     <artifactId>mybatis</artifactId>
10    <version>3.5.7</version>
11 </dependency>
12 <dependency>
13     <groupId>org.springframework.boot</groupId>
14     <artifactId>spring-boot-starter-jdbc</artifactId>
15 </dependency>
```

配置数据库连接信息

application.yaml

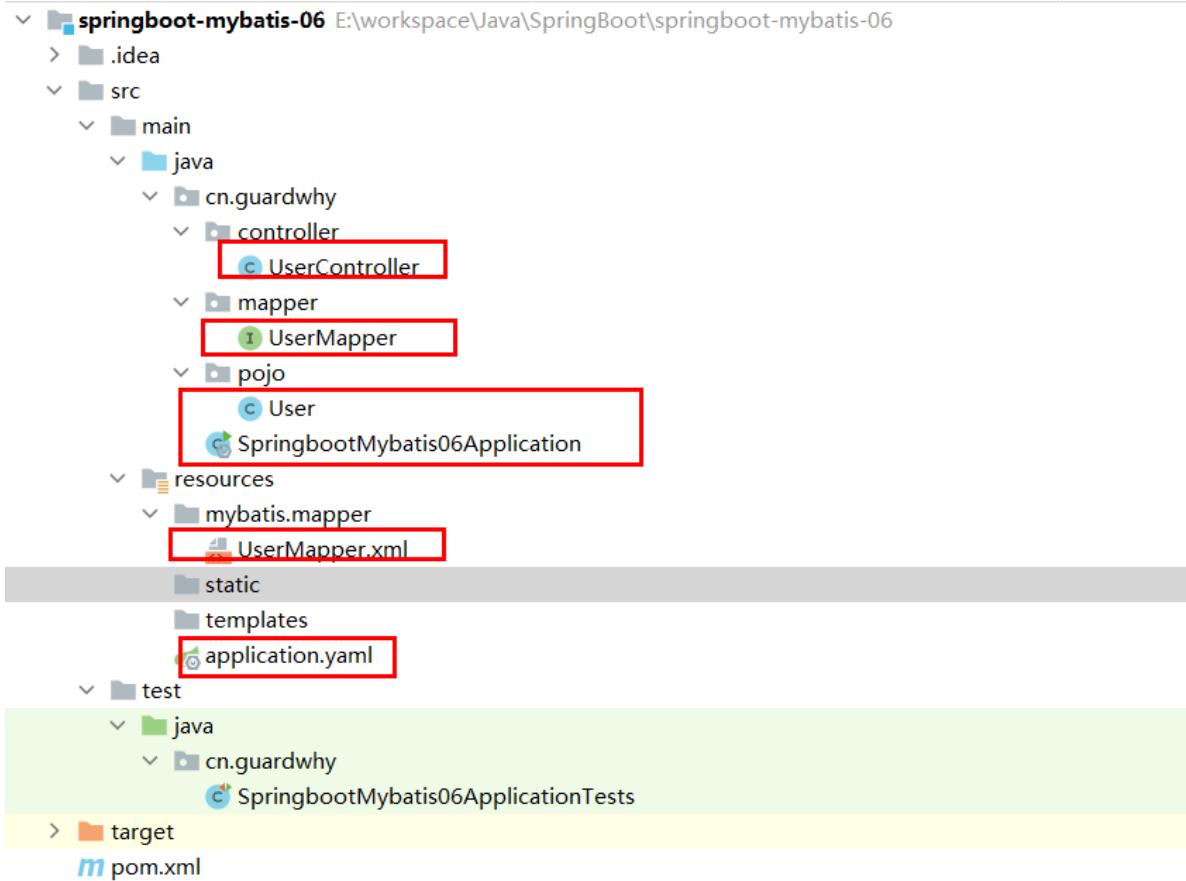
```
1 spring:
2   datasource:
3     username: root
4     password: root
5     url: jdbc:mysql://localhost:3306/springboot_mybatis?
serverTimezone=UTF&useUnicode=true&characterEncoding=utf-8
6     driver-class-name: com.mysql.cj.jdbc.Driver
```

启动工程，测试成功！！！



8.3.2 案例实现

1、工程目录



2、创建实体类User

```
1 package cn.guardwhy.pojo;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 @Data
8 @NoArgsConstructor
9 @AllArgsConstructor
10 public class User {
11     private int id;
12     private String name;
13     private String pwd;
14 }
```

3、创建mapper目录以及对应的 Mapper接口

```
1 package cn.guardwhy.mapper;
2
3 import cn.guardwhy.pojo.User;
4 import org.apache.ibatis.annotations.Mapper;
5 import org.springframework.stereotype.Repository;
6
7 import java.util.List;
8
9 @Mapper // 这个注解表示了这是一个mybatis的mapper类: dao
10 @Repository
11 public interface UserMapper {
```

```

12     // 1.查询所有的用户
13     List<User> queryUserList();
14     // 2.根据id查询用户
15     User queryUserById(Integer id);
16     // 3.添加用户
17     int addUser(User user);
18     // 4.修改用户
19     int updateUser(User user);
20
21     // 5.根据id删除用户
22     int deleteUser(int id);
23 }

```

4、配置Mapper映射文件

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3         PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4         "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <!--
6 实体类的映射文件：namespace 指定接口的类全名
7 -->
8 <mapper namespace="cn.guardwhy.mapper.UserMapper">
9
10    <select id="queryUserList" resultType="User">
11        select * from user
12    </select>
13
14    <select id="queryUserById" resultType="User">
15        select * from user where id = #{id}
16    </select>
17
18    <insert id="addUser" parameterType="User" >
19        insert into user (id, name, pwd) values(#{id},#{name},#{pwd})
20    </insert>
21
22    <update id="updateUser" parameterType="User" >
23        update user set name=#{name},pwd=#{pwd} where id=#{id}
24    </update>
25
26    <delete id="deleteUser" parameterType="int">
27        delete from user where id = #{id}
28    </delete>
29 </mapper>

```

5、myBatis 的映射配置文件

application.yaml

```

1 mybatis:
2   type-aliases-package: cn.guardwhy.pojo
3   mapper-locations: classpath:mybatis/mapper/*.xml

```

6、编写UserController进行测试

```

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

```

```

2
3 import java.util.List;
4
5 @RestController
6 public class UserController {
7     @Autowired
8     private UserMapper userMapper;
9
10    // 1.查询所有用户
11    @GetMapping("/queryUserList")
12    public List<User> queryUserList(){
13        List<User> list_user = userMapper.queryUserList();
14        for (User user : list_user) {
15            System.out.println(user);
16        }
17        return list_user;
18    }
19    // 2.查询根据id查询一个用户
20    @GetMapping("/queryUserById/{id}")
21    public User queryUserById(@PathVariable("id") Integer id){
22        return userMapper.queryUserById(id);
23    }
24
25    // 3.添加一个用户
26    @GetMapping("/adduser")
27    public String addUser(){
28        userMapper.addUser(new User(4, "guardwhy", "123"));
29        return "ok";
30    }
31
32    // 4.修改一个用户
33    @GetMapping("/updateuser")
34    public String updateUser(){
35        userMapper.updateUser(new User(3, "Papu1", "6667"));
36        return "ok_updateUser";
37    }
38
39    // 5.删除一个用户
40    @GetMapping("/deleteuser")
41    public String deleteUser(){
42        userMapper.deleteUser(6);
43        return "ok_deleteUser";
44    }
45}

```

测试成功！！！

9- Springboot 整合Swagger

9.1 Swagger基本概念

Swagger官网: <https://swagger.io/>

The screenshot shows the official website for Swagger. At the top, there's a navigation bar with icons for back, forward, search, and other browser functions. The address bar contains the URL "swagger.io". Below the address bar is the "Swagger" logo, which consists of a green circle with three dots and the word "Swagger" next to it, with the small text "Supported by SMARTBEAR" underneath. The main heading "API Development for Everyone" is centered above a large graphic of a 3D cube network. A prominent green button labeled "Explore Swagger Tools" is highlighted with a red box.

前后端分离

- 后端：后端控制层，服务层，数据访问层。
- 前端：前端控制层、视图层。
- 前后端通过API进行交互，前后端相对独立且松耦合。
- 前后端可以部署在不同的服务器上....

产生的问题

前后端集成联调，前端人员和后端人员无法做到“即使协商，尽早解决”，最终导致问题集中爆发。

解决方案：

- 指定schema【计划的提纲】，实时更新最新的API，降低集成的风险。

Swagger

- 号称世界上最流行的API框架。
- Restful API 文档在线自动生成器，API 文档与 API 定义同步更新。
- 直接运行，在线测试API，支持多种语言（如：Java, PHP等）。

9.2 集成Swagger

- 1、新建一个Springboot项目。
- 2、添加相关的maven依赖

2.x.x 版本

```
1 <!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger2 -->
2 <dependency>
3   <groupId>io.springfox</groupId>
4   <artifactId>springfox-swagger2</artifactId>
5   <version>2.9.2</version>
6 </dependency>
7 <!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger-ui -->
8 <dependency>
9   <groupId>io.springfox</groupId>
10  <artifactId>springfox-swagger-ui</artifactId>
11  <version>2.9.2</version>
12 </dependency>
```

3.x.x版本

```
1 <dependency>
2   <groupId>io.springfox</groupId>
3   <artifactId>springfox-boot-starter</artifactId>
4   <version>3.0.0</version>
5 </dependency>
```

3、编写对应的controller

```
1 package cn.guardwhy.controller;
2
3 import org.springframework.web.bind.annotation.RequestMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 public class HelloController {
8     @RequestMapping(value = "/hello")
9     public String hello(){
10         return "hello Swagger!!!!";
11     }
12 }
```

4、编写一个配置类-SwaggerConfig来配置 Swagger

```
1 package cn.guardwhy.config;
2
3 import org.springframework.context.annotation.Configuration;
4 import springfox.documentation.swagger2.annotations.EnableSwagger2;
5
6 @Configuration // 配置类
7 @EnableSwagger2 // 开启Swagger2的自动配置
8 public class SwaggerConfig {
9
10 }
```

5、启动项目，访问测试。就可以看到swagger界面！！

2.x.x版本: <http://localhost:8080/swagger-ui.html>

The screenshot shows the Swagger UI 1.0 interface. At the top, it displays the URL localhost:8080/swagger-ui.html. The main header says "swagger" and "Select a spec default". Below this, the title "Api Documentation" is shown with a "1.0" badge. It includes links for "Base URL: localhost:8080/" and "<http://localhost:8080/v2/api-docs>". A red box highlights the "Swagger 信息" link. Under "Api Documentation", there are links for "Terms of service" and "Apache 2.0". The main content area lists two controllers: "basic-error-controller" (Basic Error Controller) and "hello-controller" (Hello Controller). A red box highlights the "接口信息" link under the "basic-error-controller" section. At the bottom, there are sections for "Models" and "实体类信息", each with a red border.

3.x.x版本: <http://localhost:8080/swagger-ui/>

The screenshot shows the Swagger UI 3.x.x interface. At the top, it displays the URL localhost:8080/swagger-ui/. The main header says "Swagger" and "Supported by SMARTBEAR". It includes a "Select a definition" dropdown set to "default". Below this, the title "Api Documentation" is shown with a "1.0 OAS3" badge. It includes links for "<http://localhost:8080/v3/api-docs>", "Api Documentation", "Terms of service", and "Apache 2.0". A red box highlights the "Servers" dropdown set to "http://localhost:8080 - Inferred Url". The main content area lists two controllers: "basic-error-controller" (Basic Error Controller) and "hello-controller" (Hello Controller). A red box highlights the "接口信息" link under the "basic-error-controller" section. At the bottom, there is a "Schemas" section with a red border.

9.3 配置Swagger

1、Swagger实例Bean是Docket，通过配置Docket实例来配置Swagger。

```
1 package cn.guardwhy.config;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import springfox.documentation.spi.DocumentationType;
6 import springfox.documentation.spring.web.plugins.Docket;
7 import springfox.documentation.swagger2.annotations.EnableSwagger2;
8
```

```

9  @Configuration // 配置类
10 @EnableSwagger2 // 开启Swagger2的自动配置
11 public class SwaggerConfig {
12     // 1.配置了Swagger的Docket的bean实例
13     @Bean
14     public Docket docket(){
15         return new Docket(DocumentationType.SWAGGER_2);
16     }
17 }

```

源码分析

The screenshot shows the Java code for the `SwaggerConfig`, `Docket`, and `ApiInfo` classes. The `SwaggerConfig` class contains a `@Bean` annotation for the `docket()` method. The `Docket` class has a private field `apiInfo` initialized to `ApiInfo.DEFAULT`. The `ApiInfo` class contains static final fields for `DEFAULT_CONTACT` and `DEFAULT`, which are `Contact` and `ApiInfo` objects respectively.

```

SwaggerConfig.java:
10 import java.util.ArrayList;
11 ...
13 @Configuration // 配置类
14 @EnableSwagger2 // 开启Swagger2的自动配置
15 public class SwaggerConfig {
16     // 1.配置了Swagger的Docket的bean实例
17     @Bean
18     public Docket docket(){
19         // 1.1 Docket实例关联上 apiInfo()
20         return new Docket(DocumentationType.SWAGGER_2).apiInfo(apiInfo());
21     }
22 }

Docket.java:
84 private List<? extends SecurityScheme> securitySchemes;
85 private Ordering<ApiListingReference> apiListingReferenceOrdering;
86 private Ordering<ApiDescription> apiDescriptionOrdering;
87 private Ordering<Operation> operationOrdering;
88
89 private ApiInfo apiInfo = ApiInfo.DEFAULT;
90 private String groupName = DEFAULT_GROUP_NAME;

ApiInfo.java:
1 package springfox.documentation.service;
2 ...
3 public class ApiInfo {
4     ...
5     public static final Contact DEFAULT_CONTACT = new Contact("", "", "");
6     public static final ApiInfo DEFAULT = new ApiInfo("Api Documentation", "Api Documentation", "1.0", "urn:tos", DEFAULT_CONTACT, "Apache 2.0", "http://www.apache.org/licenses/LICENSE-2.0", new ArrayList<VendorExtension>());
7 }


```

2、可以通过apiInfo()属性配置文档信息

```

1 // 2.配置Swagger信息=apiInfo
2 private ApiInfo apiInfo() {
3     // 2.1 作者信息
4     Contact contact = new Contact("guardwhy",
5         "https://home.cnblogs.com/u/Guard9/", "hxy1625309592@aliyun.com");
6     return new ApiInfo(
7         "学习记录总结!!!",
8         "好好学习，天天向上！！！",
9         "1.0",
10        "urn:tos",
11        contact,
12        "Apache 2.0",
13        "http://www.apache.org/licenses/LICENSE-2.0",
14        new ArrayList()
15    );
}

```

3、Docket 实例关联上 apiInfo()

```

1 package cn.guardwhy.config;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import springfox.documentation.service.ApiInfo;
6 import springfox.documentation.service.Contact;
7 import springfox.documentation.spi.DocumentationType;
8 import springfox.documentation.spring.web.plugins.Docket;
9 import springfox.documentation.swagger2.annotations.EnableSwagger2;
10

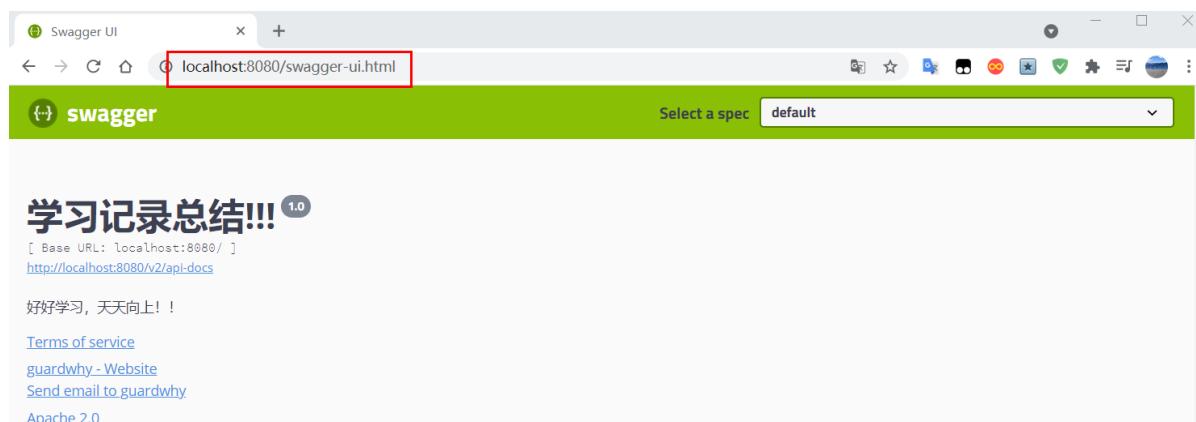
```

```

11 import java.util.ArrayList;
12
13 @Configuration // 配置类
14 @EnableSwagger2 // 开启Swagger2的自动配置
15 public class SwaggerConfig {
16     // 1.配置了Swagger的Docket的bean实例
17     @Bean
18     public Docket docket(){
19         // 1.1 Docket实例关联上 apiInfo()
20         return new Docket(DocumentationType.SWAGGER_2).apiInfo(apiInfo());
21     }
22
23     // 2.配置Swagger信息=apiInfo
24     private ApiInfo apiInfo() {
25         // 2.1 作者信息
26         Contact contact = new Contact("guardwhy",
27             "https://home.cnblogs.com/u/Guard9/", "hxy1625309592@aliyun.com");
28         return new ApiInfo(
29             "学习记录总结!!!",
30             "好好学习，天天向上！！",
31             "1.0",
32             "urn:tos",
33             contact,
34             "Apache 2.0",
35             "http://www.apache.org/licenses/LICENSE-2.0",
36             new ArrayList()
37         );
38     }

```

4、重启项目，访问测试！！



9.4 配置扫描接口

1、通过 `select()` 方法配置扫描接口

```

1  @Bean
2  public Docket docket(){
3      // 1.1 Docket实例关联上 apiInfo()
4      return new Docket(DocumentationType.SWAGGER_2)
5          .apiInfo(apiInfo())
6          .select()
7              // RequestHandlerSelectors,配置要扫描接口的方式
8              // basePackage:指定要扫描的包
9              // any():扫描全部

```

```
10     // none():不扫描
11     // withClassAnnotation: 扫描类上的注解，参数是一个注解的反射对象
12     // withMethodAnnotation: 扫描方法上的注解
13     .apis(RequestHandlerSelectors.basePackage("cn.guardwhy.controller"))
14     .build();
15 }
```

2、配置接口扫描过滤

```
1 @Bean
2 public Docket docket(){
3     // 1.1 Docket实例关联上 apiInfo()
4     return new Docket(DocumentationType.SWAGGER_2)
5         .apiInfo(apiInfo())
6         .select()
7         .apis(RequestHandlerSelectors.basePackage("cn.guardwhy.controller"))
8         // 配置如何通过path过滤，即这里只扫描请求以/guardwhy开头的接口
9         .paths(PathSelectors.ant("/guardwhy/**"))
10        .build();
11 }
```

9.5 配置开关Swagger

1、通过 enable() 方法配置是否启用swagger，如果是false，swagger将不能在浏览器中访问了。

```
1 @Bean
2 public Docket docket(){
3     // 1.1 Docket实例关联上 apiInfo()
4     return new Docket(DocumentationType.SWAGGER_2)
5         .apiInfo(apiInfo())
6         .enable(false) // enable是否启动Swagger，如果为False，则Swagger不能再浏览器中的访问！
7         .select()
8         .apis(RequestHandlerSelectors.basePackage("cn.guardwhy.controller"))
9         // 配置如何通过path过滤，即这里只扫描请求以/guardwhy开头的接口
10        .paths(PathSelectors.ant("/guardwhy/**"))
11        .build();
12 }
```

执行结果



2、希望Swagger在生产环境中使用，在发布的时候不使用？

application.properties

```
1 | spring.profiles.active=dev
```

application-dev.properties

```
1 | server.port=8081
```

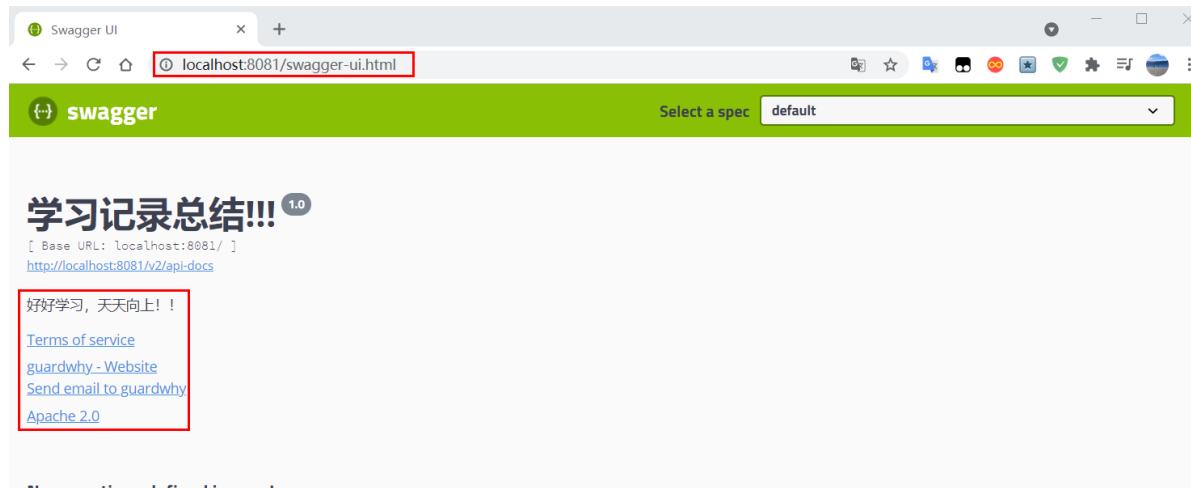
application-pro.properties

```
1 | server.port=8082
```

SwaggerConfig

```
1 @Bean
2 public Docket docket(Environment environment){
3     // 1.设置要显示Swagger的环境
4     Profiles profiles = Profiles.of("dev", "test");
5     // 2.通过environment.acceptsProfiles 判断是否处在自己设定的环境当中
6     boolean flag = environment.acceptsProfiles(profiles);
7     System.out.println(flag);
8
9     return new Docket(DocumentationType.SWAGGER_2)
10    .apiInfo(apiInfo())
11    .enable(flag) // enable是否启动Swagger, 如果为False, 则Swagger不能再浏览器
12    中的访问! !
13    .select()
14    .apis(RequestHandlerSelectors.basePackage("cn.guardwhy.controller"))
15    // 配置如何通过path过滤, 即这里只扫描请求以/guardwhy开头的接口
16    .paths(PathSelectors.ant("/guardwhy/**"))
17    .build();
}
```

3、启动项目，访问测试 <http://localhost:8081/swagger-ui.html>



9.6 分组和实体类配置

9.6.1 配置分组

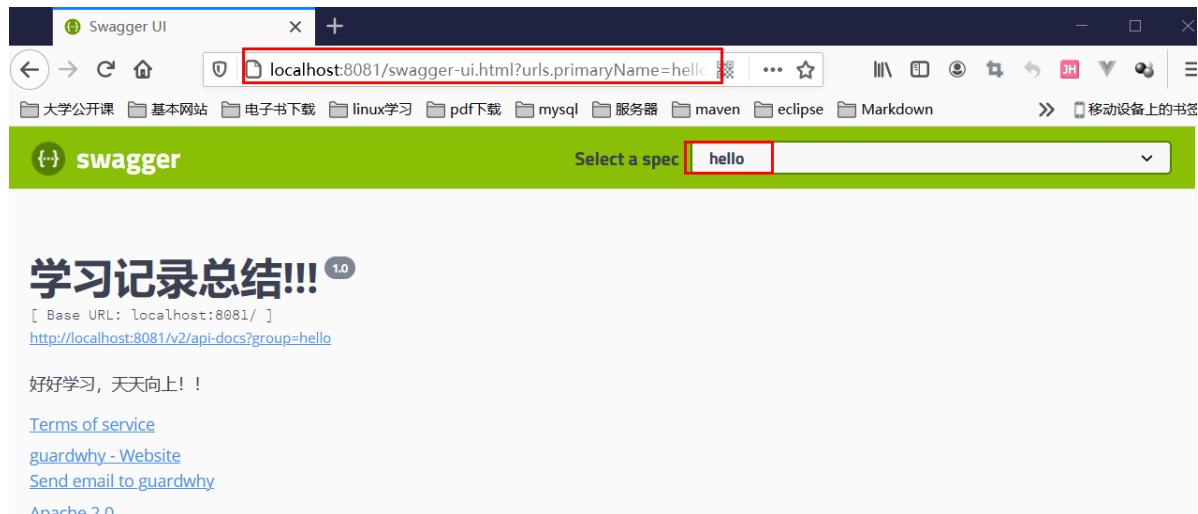
1、如果没有配置分组，默认是default。通过 `groupName()` 方法即可配置分组

```

1  @Bean
2  public Docket docket(Environment environment){
3      return new Docket(DocumentationType.SWAGGER_2)
4          .apiInfo(apiInfo())
5          .groupName("hello")
6          .enable(flag) // enable是否启动Swagger, 如果为False, 则Swagger不能再浏览器
7              中的访问! !
8          .select()
9          .apis(RequestHandlerSelectors.basePackage("cn.guardwhy.controller"))
10         // 配置如何通过path过滤, 即这里只扫描请求以/guardwhy开头的接口
11         .paths(PathSelectors.ant("/guardwhy/**"))
12         .build();

```

2、启动项目，查看结果！！！



3、配置多个分组只需要配置多个docket。

```

1 // 3. 配置多个分组
2 @Bean
3 public Docket docket1(){
4     return new Docket(DocumentationType.SWAGGER_2).groupName("group1");
5 }
6
7 @Bean
8 public Docket docket2(){
9     return new Docket(DocumentationType.SWAGGER_2).groupName("group2");
10 }
11
12 @Bean
13 public Docket docket3(){
14     return new Docket(DocumentationType.SWAGGER_2).groupName("group3");
15 }

```

4、重启项目，测试:<http://localhost:8080/swagger-ui.html>

The screenshot shows the Swagger UI interface. At the top, the URL bar contains 'localhost:8081/swagger-ui.html#/hello-controller'. Below the URL bar, there's a green header with the word 'swagger'. To the right of the header, a dropdown menu is open with the title 'Select a spec'. The dropdown menu lists several items: 'group1', 'group1' (which is highlighted in blue), 'group2', 'group3', and 'hello'. A red box surrounds the 'group1' item in the dropdown. Further down the page, there's a section titled 'Api Documentation' with a sub-section 'basic-error-controller' and another section 'hello-controller' which is currently expanded, showing its details.

9.6.2 实体配置

1、新建一个实体类 user类

@ApiModelProperty为类添加注释

@ApiModelPropertyProperty为类属性添加注释

```
1 package cn.guardwhy.pojo;
2
3 import io.swagger.annotations.ApiModel;
4 import io.swagger.annotations.ApiModelProperty;
5
6 @ApiModel("用户实体类")
7 public class User {
8     @ApiModelProperty("用户名")
9     public String username;
10    @ApiModelProperty("密码")
11    public String password;
12 }
```

2、只要这个实体在请求接口的返回值上，都能映射到实体项中

```
1 // 只要接口中，返回值存在实体类，它就会扫描到Swagger中
2 @PostMapping(value = "/user")
3 public User user(){
4     return new User();
5 }
```

3、重启项目，执行结果！！！

The screenshot shows the Swagger UI interface. In the top navigation bar, the URL is 'localhost:8081/swagger-ui.html?urls.primaryName=group'. Below the navigation, there's a sidebar titled 'Models' with sections for 'ModelAndView' and 'View'. The main content area displays a code block for a 'User Entity Class' (用户实体类) with two fields: 'password' (string, 密码) and 'username' (string, 用户名). The entire code block is highlighted with a red rectangle.

9.6.3 小结

Swagger是个优秀的工具，较于传统的Postman或Curl方式测试接口，使用swagger简直就是傻瓜式操作，不需要额外说明文档。

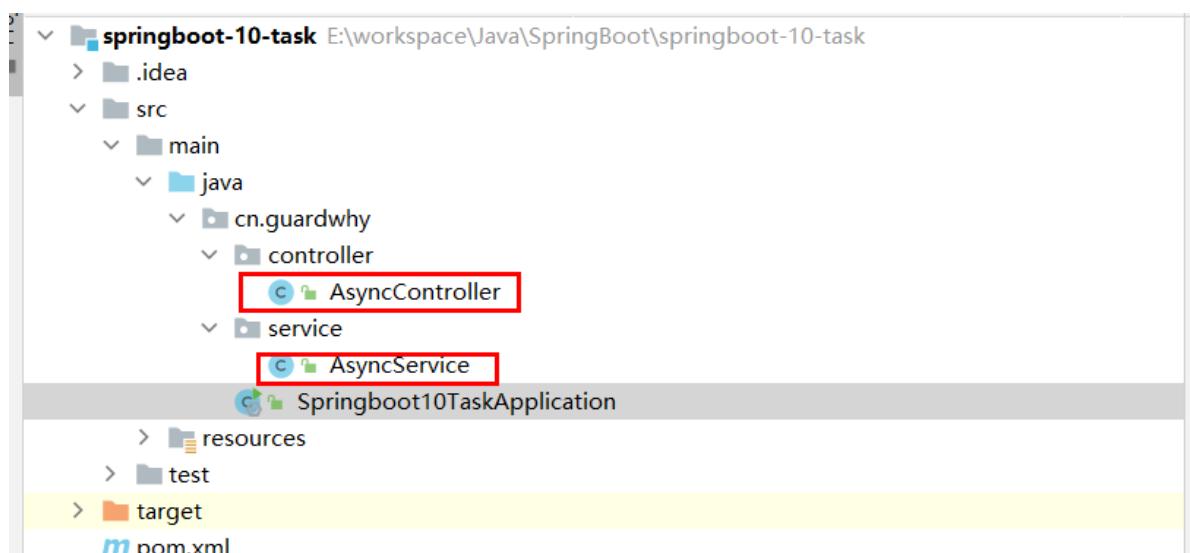
10- springboot整合任务

10.1 异步任务

在网站上发送邮件，后台会去发送邮件，此时前台会造成响应不动，直到邮件发送完毕，响应才会成功，所以一般会采用多线程的方式去处理这些任务。

编写方法，假装正在处理数据，使用线程设置一些延时，模拟同步等待的情况。

1、项目目录



2、编写AsyncService

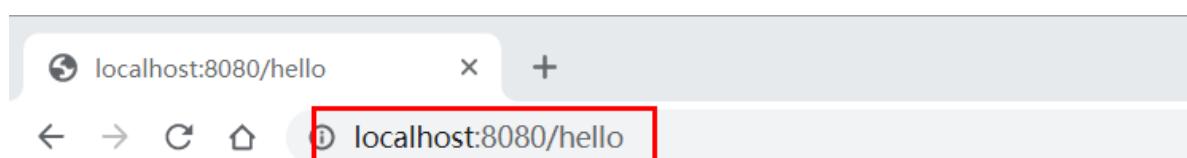
```
1 package cn.guardwhy.service;
2
3 import org.springframework.stereotype.Service;
4
5 @Service
```

```
6 public class AsyncService {
7     public void hello(){
8
9         try {
10             // 休眠3秒
11             Thread.sleep(30000);
12         } catch (InterruptedException e) {
13             e.printStackTrace();
14         }
15         // 输出结果
16         System.out.println("数据处理中.....");
17     }
18 }
```

3、编写AsyncController类

```
1 package cn.guardwhy.controller;
2
3 import cn.guardwhy.service.AsyncService;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class AsyncController {
10     @Autowired
11     AsyncService asyncService;
12
13     @RequestMapping("/hello")
14     public String hello(){
15         asyncService.hello(); // 停止3s, 转圈
16         return "success";
17     }
18 }
```

4、启动项目，访问<http://localhost:8080/hello>进行测试，3秒后出现了success，这是同步等待的情况。



success

5、给hello方法添加 @Async注解，可以在用户直接得到消息，在后台使用多线程的方式进行处理。

```
1 package cn.guardwhy.service;
2
3 import org.springframework.scheduling.annotation.Async;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class AsyncService {
8     @Async // 告诉Spring这是一个异步方法
9     public void hello(){
```

```

10
11     try {
12         // 休眠3秒
13         Thread.sleep(30000);
14     } catch (InterruptedException e) {
15         e.printStackTrace();
16     }
17     // 输出结果
18     System.out.println("数据处理中.....");
19 }
20 }
```

6、SpringBoot就会自己开一个线程池，进行调用！但是要让这个注解生效，需要在主程序上添加一个注解`@EnableAsync`，开启异步注解功能。

```

1 package cn.guardwhy;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.scheduling.annotation.EnableAsync;
6
7 @SpringBootApplication
8 @EnableAsync // 开启异步注解功能
9 public class Springboot10TaskApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(Springboot10TaskApplication.class, args);
13     }
14 }
15 }
```

7、重启项目，测试网页瞬间响应！！

10.2 定时任务

1、开发过程中经常需要执行一些定时任务，Spring为我们提供了异步执行任务调度的方式，提供了两个接口

- TaskExecutor接口
- TaskScheduler接口

2、创建一个ScheduledService

```

1 package cn.guardwhy.service;
2
3 import org.springframework.scheduling.annotation.Scheduled;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class ScheduledService {
8     //秒 分 时 日 月 周几
9     /*
10      * 30 15 10 * * 每天10点15分30 执行一次
11      * 30 0/5 10, 18 * *? 每天10点和18点，每隔五分钟执行一次
12      * 0 15 10 ? * 1-6 每个月的周一到周六 10.15分钟执行一次
13      */
14 }
```

```

14     @Scheduled(cron = "0 * * * * 0-7")
15     public void hello(){
16         System.out.println("hello Springboot! ! !");
17     }
18 }
```

3、**cron** 表达式是一个字符串，该字符串由 6 个空格分为 7 个域，每一个域代表一个时间含义。格式如下：

1 | [秒] [分] [时] [日] [月] [周] [年]

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

4、在主程序上增加 `@EnableScheduling` 开启定时任务功能。

```

1 package cn.guardwhy;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.scheduling.annotation.EnableAsync;
6 import org.springframework.scheduling.annotation.EnableScheduling;
7
8 @SpringBootApplication
9 @EnableAsync // 开启异步注解功能
10 @EnableScheduling // 开启定时功能的注解
11 public class Springboot10TaskApplication {
12
13     public static void main(String[] args) {
14         SpringApplication.run(Springboot10TaskApplication.class, args);
15     }
16 }
```

10.3 邮件任务

1、导入相关的依赖

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-mail</artifactId>
4 </dependency>
```

2、全局搜索自动配置类 `MailSenderAutoConfiguration`

```
1  ...
16
17 package org.springframework.boot.autoconfigure.mail;
18
19 import ...
33
34
35     Auto configuration for email support.
36     Since: 1.2.0
37     Author: Oliver Gierke, Stephane Nicoll, Eddú Meléndez
38
39     @Configuration(proxyBeanMethods = false)
40     @ConditionalOnClass({ MimeMessage.class, MimeType.class, MailSender.class })
41     @ConditionalOnMissingBean(MailSender.class)
42     @Conditional(MailSenderCondition.class)
43     @EnableConfigurationProperties(MailProperties.class)
44
45     @Import({ MailSenderJndiConfiguration.class, MailSenderPropertiesConfiguration.class })
46
47     public class MailSenderAutoConfiguration {
48
49
50         /**
51          * Condition to trigger the creation of a MailSender. This kicks in if either the host or jndi name
52          * property is set.
53         */
54         static class MailSenderCondition extends AnyNestedCondition {...}
55
56     }
57
58 }
```

3、查看自动配置文件 MailProperties

```
1  @ConfigurationProperties(prefix = "spring.mail")
2  public class MailProperties {
3
4      private static final Charset DEFAULT_CHARSET = StandardCharsets.UTF_8;
5
6      /**
7       * SMTP server host. For instance, `smtp.example.com`.
8       */
9      private String host;
10
11     /**
12      * SMTP server port.
13      */
14     private Integer port;
15
16     /**
17      * Login user of the SMTP server.
18      */
19     private String username;
20
21     /**
22      * Login password of the SMTP server.
23      */
24     private String password;
25
26     /**
27      * Protocol used by the SMTP server.
28      */
29     private String protocol = "smtp";
30
31     /**
32      * Default MimeMessage encoding.
```

```
33     */
34     private Charset defaultEncoding = DEFAULT_CHARSET;
35
36     /**
37      * Additional JavaMail Session properties.
38      */
39     private Map<String, String> properties = new HashMap<>();
40
41     /**
42      * Session JNDI name. When set, takes precedence over other Session
43      * settings.
44      */
45     private String jndiName;
46 }
```

4、配置文件

```
1 spring.mail.username=1625309592@qq.com
2 spring.mail.password=kommvnikudpsdgca
3 spring.mail.host=smtp.qq.com
4 # 开启加密验证
5 spring.mail.properties.mail.smtp.ssl.enable=true
```

5、单元测试

```
1 package cn.guardwhy;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.boot.test.context.SpringBootTest;
6 import org.springframework.mail.SimpleMailMessage;
7 import org.springframework.mail.javamail.JavaMailSenderImpl;
8 import org.springframework.mail.javamail.MimeMessageHelper;
9
10 import javax.mail.MessagingException;
11 import javax.mail.internet.MimeMessage;
12 import java.io.File;
13
14 @SpringBootTest
15 class Springboot10TaskApplicationTests {
16     @Autowired
17     JavaMailSenderImpl mailSender;
18
19     @Test
20     void contextLoads1() {
21         // 1.一个简单的邮件
22         SimpleMailMessage mailMessage = new SimpleMailMessage();
23         mailMessage.setSubject("hello Springboot!!!");
24         mailMessage.setText("Springboot源码分析");
25
26         // 发送
27         mailMessage.setTo("1625309592@qq.com");
28         mailMessage.setFrom("1625309592@qq.com");
29
30         mailSender.send(mailMessage);
31     }
32 }
```

```

32
33     @Test
34     void contextLoads2() throws MessagingException {
35         // 1.一个复杂的邮件
36         MimeMessage mimeMessage = mailSender.createMimeMessage();
37         // 2.组装
38         MimeMessageHelper helper = new MimeMessageHelper(mimeMessage, true);
39         // 正文
40         helper.setSubject("hello Springboot!!!!");
41         helper.setText("Springboot源码分析");
42
43         // 附件
44         helper.addAttachment("my.jpg", new
45             File("C:\\\\Users\\\\linux\\\\Pictures\\\\my.jpg"));
46         helper.addAttachment("timg.jpg", new
47             File("C:\\\\Users\\\\linux\\\\Pictures\\\\timg.jpg"));
48
49         // 发送
50         helper.setTo("1625309592@qq.com");
51         helper.setFrom("1625309592@qq.com");
52
53         mailSender.send(mimeMessage);
54     }

```

6、启动项目，执行结果！！！

The screenshot shows the QQ Mail inbox interface. The top navigation bar includes the QQ Mail logo, account information (Guard9<hxy1625309592@foxmail.com>), and links for Feedback, Help Center, and Logout. A search bar is also present.

The main content area displays an incoming email from Guard9. The email details are as follows:

- Subject: Springboot源码分析
- Date: 2021年5月16日 (星期一) 下午1:49
- To: 10号 <1625309592@qq.com>
- Attachments: 2个 (my.jpg...)

The email body contains the text "Springboot源码分析". Below the body, the attachment section shows two files:

- 普通附件:** my.jpg (224.20K) - Preview, Download,收藏, 转存
- 普通附件:** timg.jpg (18.93K) - Preview, Download,收藏, 转存

The left sidebar provides navigation links for writing, receiving, and managing contacts. It also lists various mailbox categories such as收件箱(1), 垃圾箱(2), and several other sections like 我的文件夹, 在线文档, and 日历.