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。

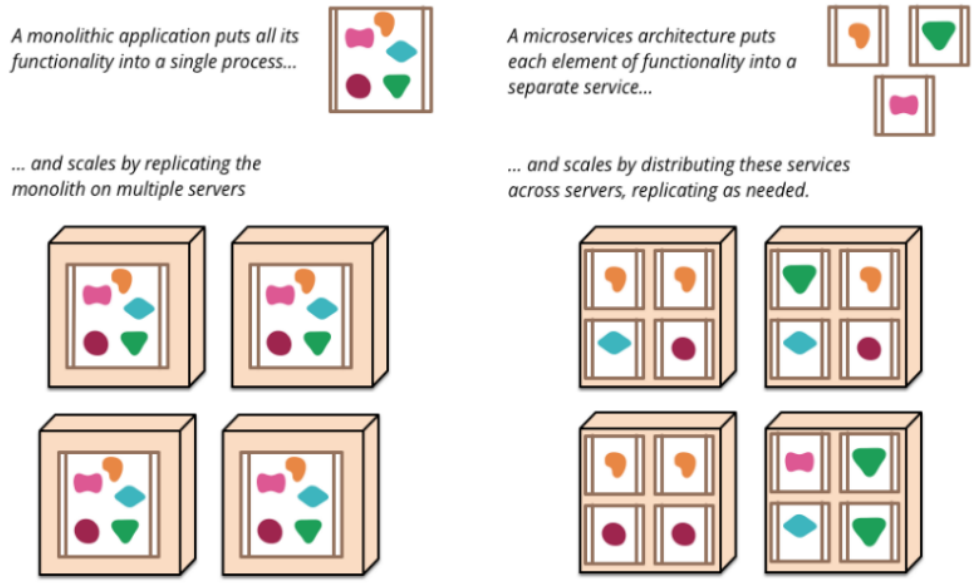
单体应用架构优点

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



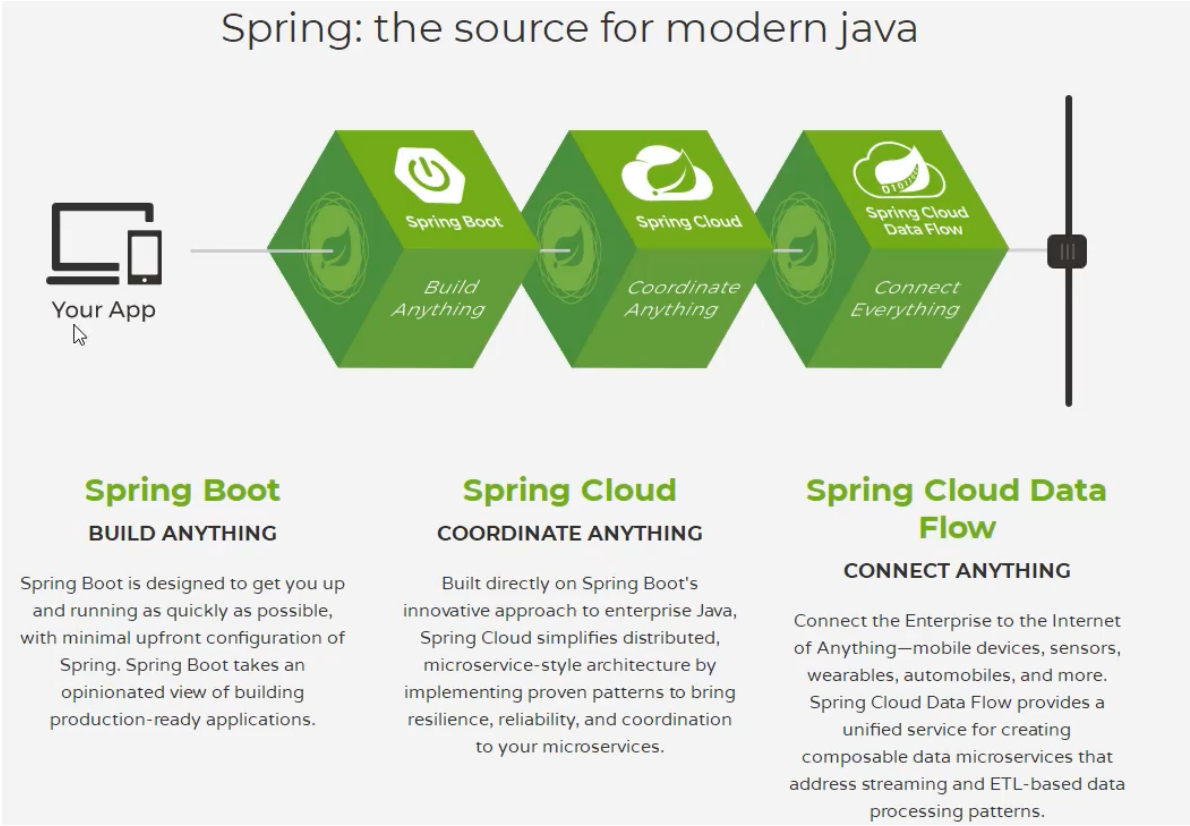
所谓微服务架构，就是要解决单体应用架构的方式，将每个功能元素独立出来，把独立出来的功能元素的动态组合，需要的功能元素才去拿来组合，需要将多一些时间可以整合多个功能元素。所以微服务架构只是对功能元素进行复制，而没有对整个应用进行复制。

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



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

```
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的第三方库配置（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

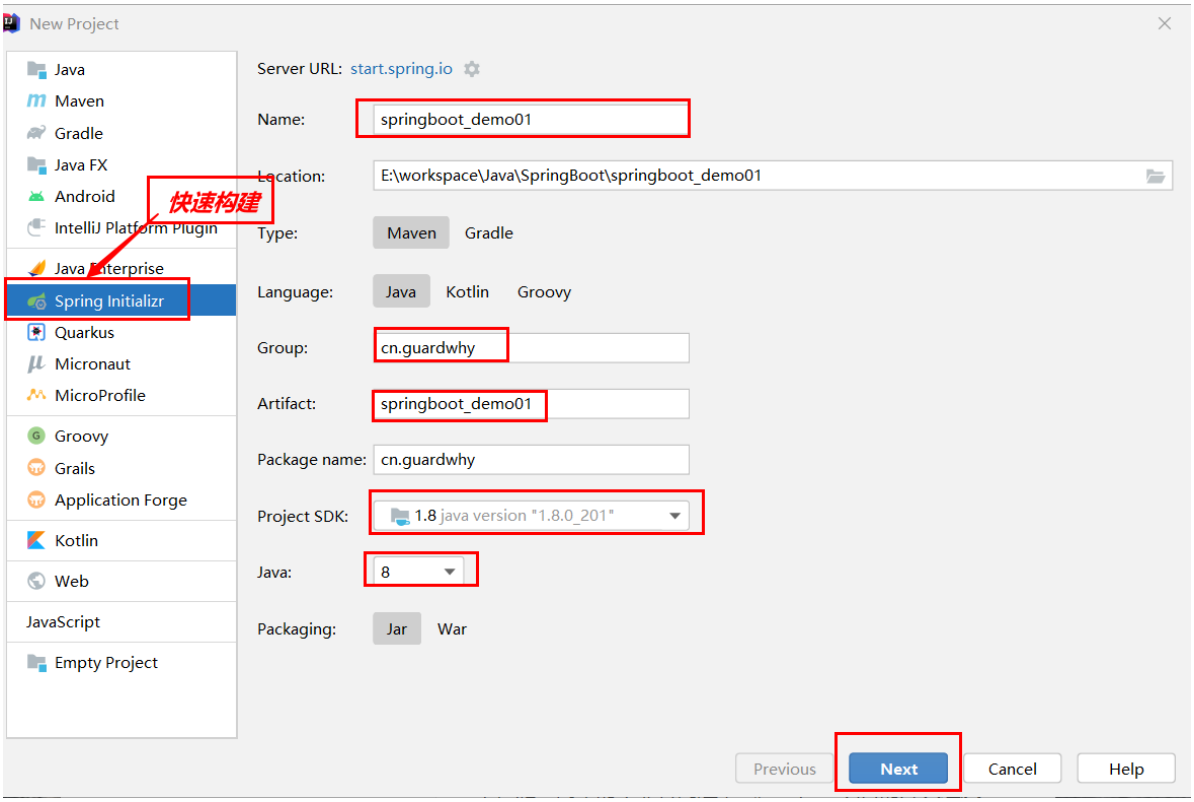
- IDEA

3.2 SpringBoot 快速构建

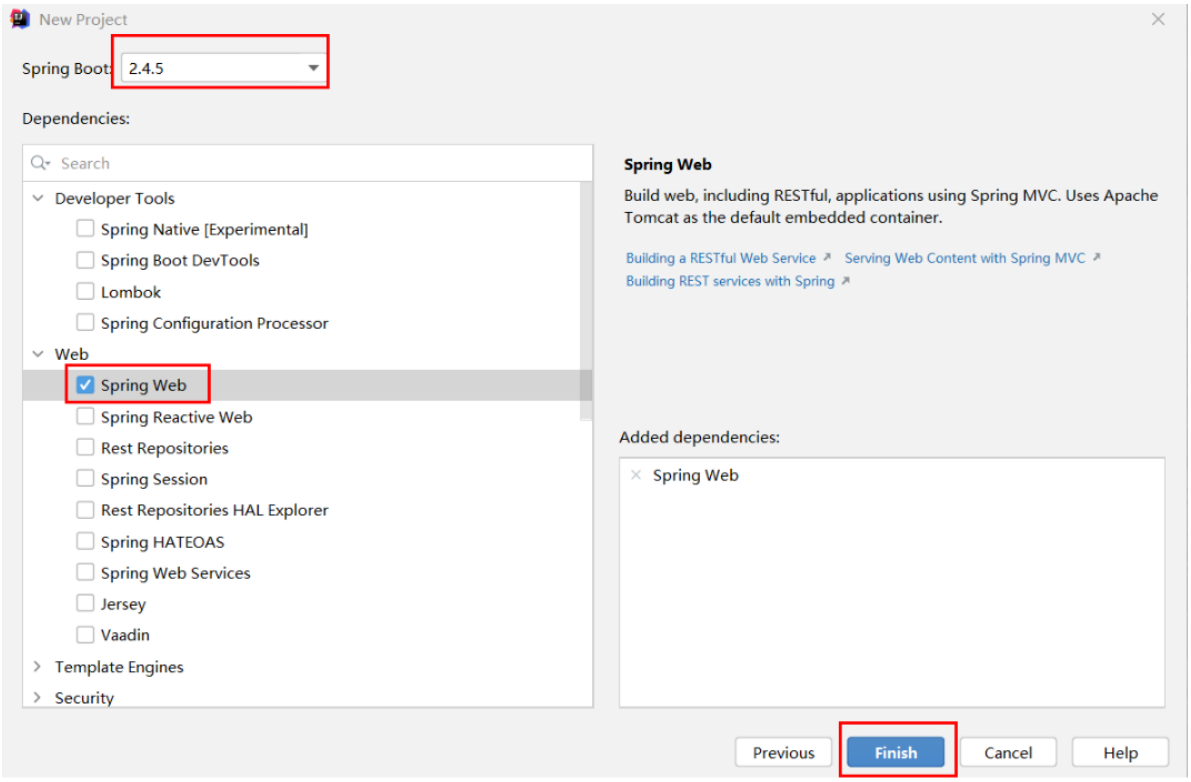
案例需求：请求Controller中的方法，并将返回值响应到页面。

使用Spring Initializr方式构建Spring Boot项目

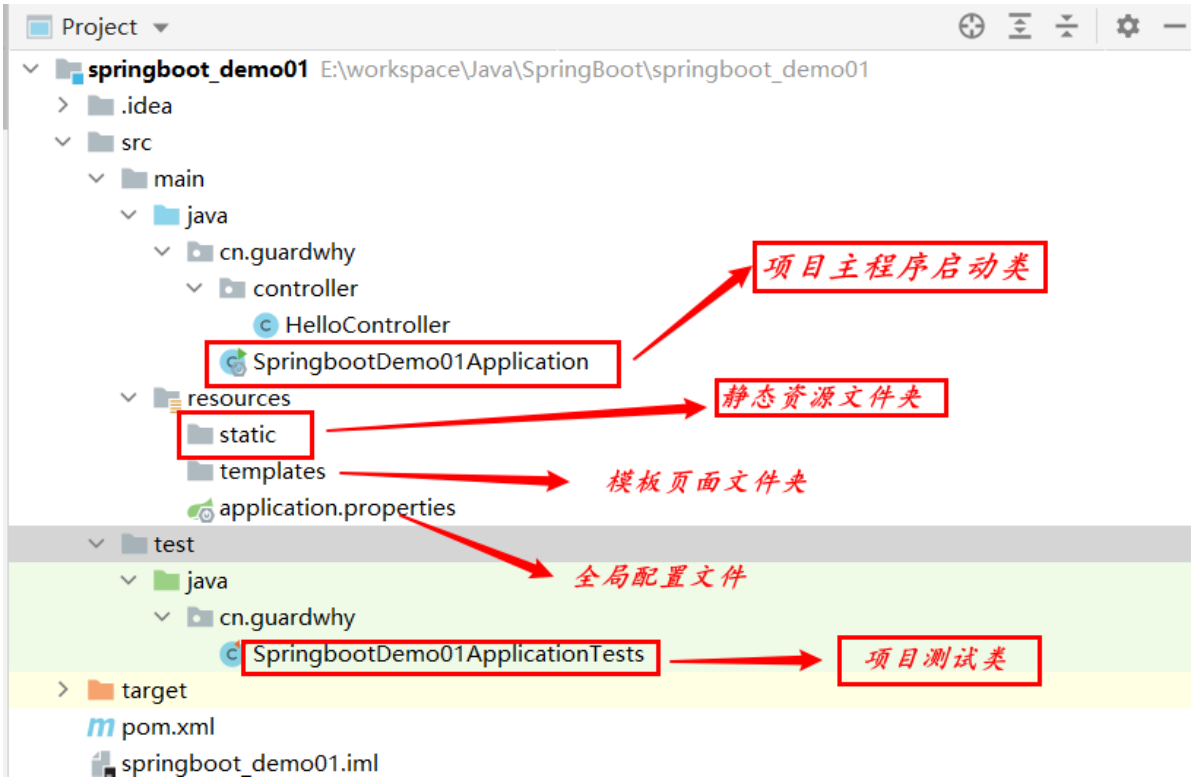
本质上说，Spring Initializr是一个Web应用，它提供了一个基本的项目结构，能够帮助我们快速构建一个基础的Spring Boot项目。



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"?>
```

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <!--父依赖-->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.4.5</version>
        <relativePath/>
    </parent>
    <groupId>cn.guardwhy</groupId>
    <artifactId>springboot_demo01</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springboot_demo01</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <!--web场景启动器-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!--Springboot单元测试-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!--打包插件-->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

创建一个用于Web访问的Controller

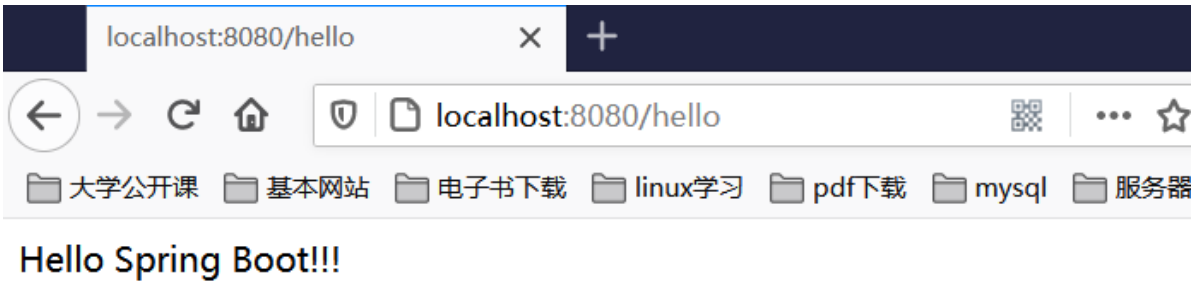
在主程序的同级目录下，新建一个controller包，一定要在同级目录下，否则识别不到。
在包中新建一个HelloController类

```java
package cn.guardwhy.controller;

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

@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String hello(){
        return "Hello Spring Boot!!!";
    }
}
```

运行项目

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

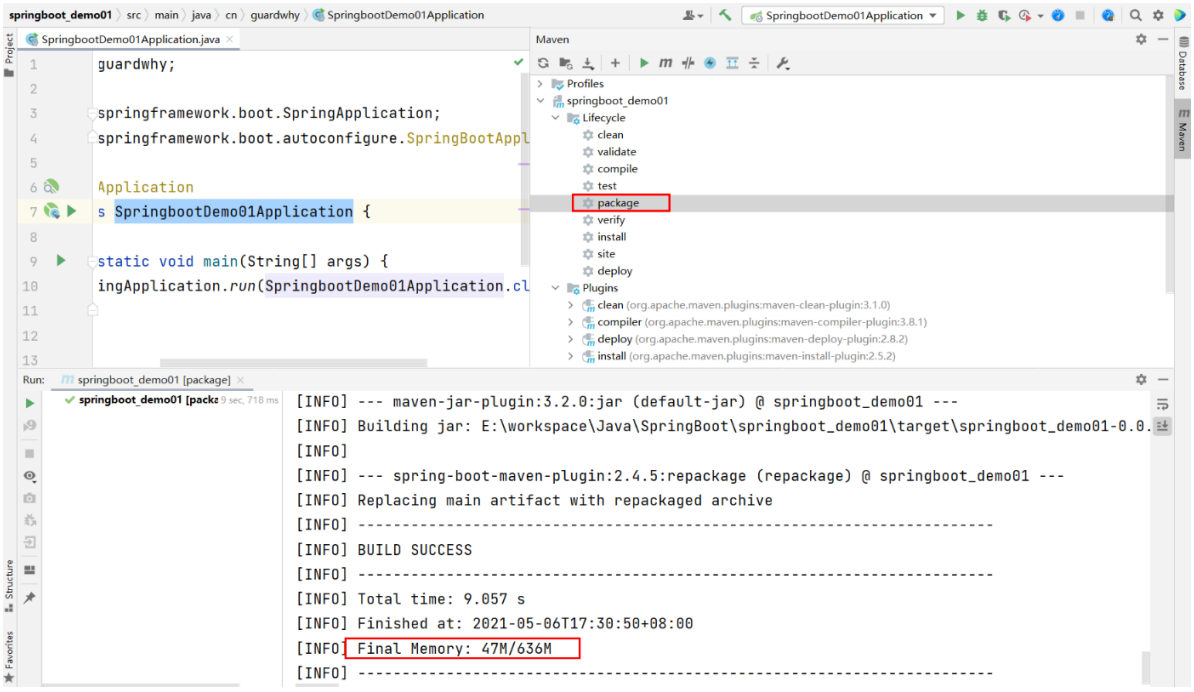
在控制台上会发现SpringBoot项目默认启动的端口号为8080，此时，可以在浏览器上访问 http://localhost:8080/hello



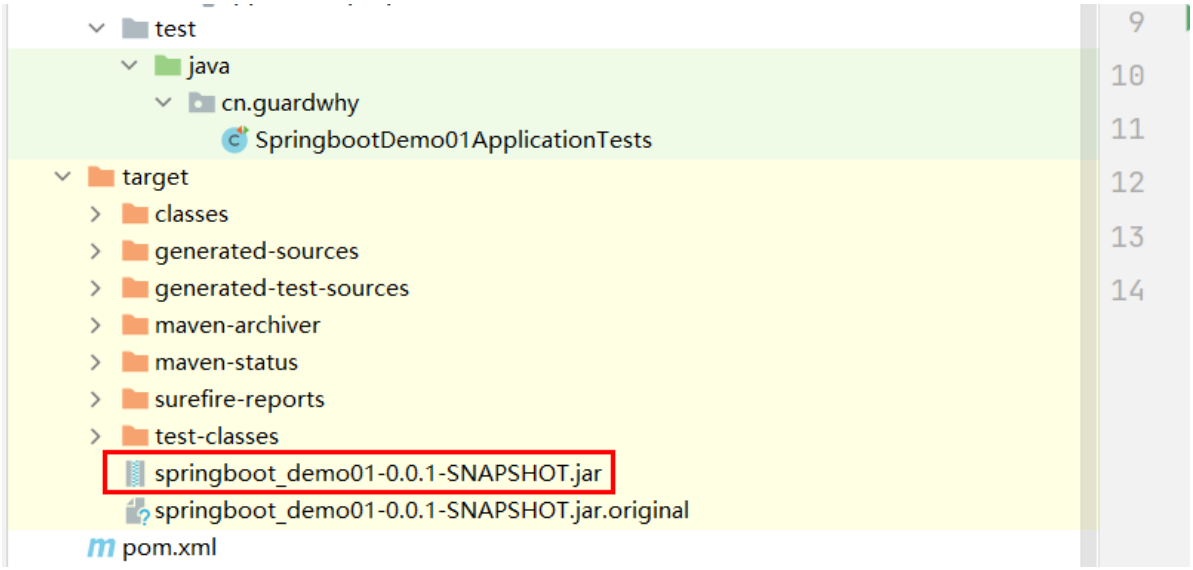
页面输出的内容是“Hello Spring Boot!!!”，至此，构建Spring Boot项目就完成了。

3.3 项目打包

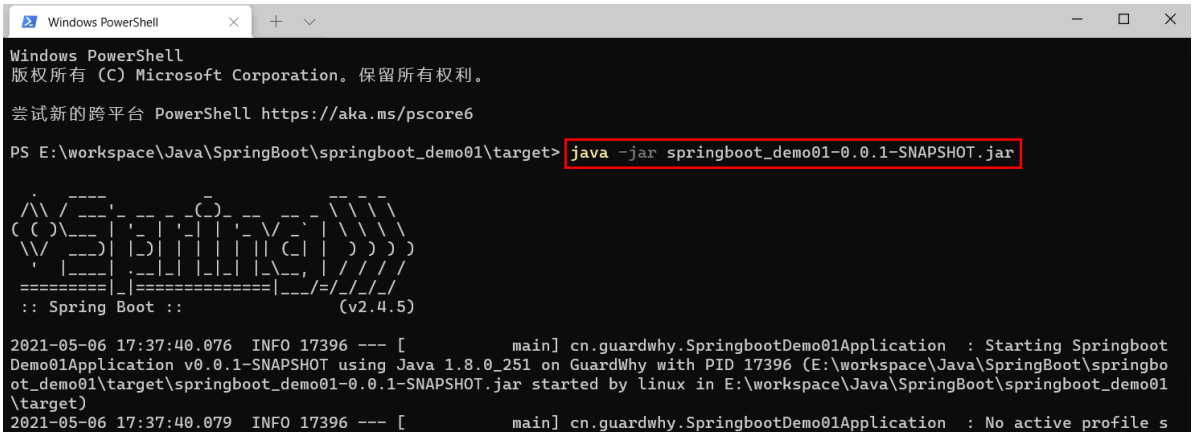
将项目打成jar包，点击 maven的 package！！！



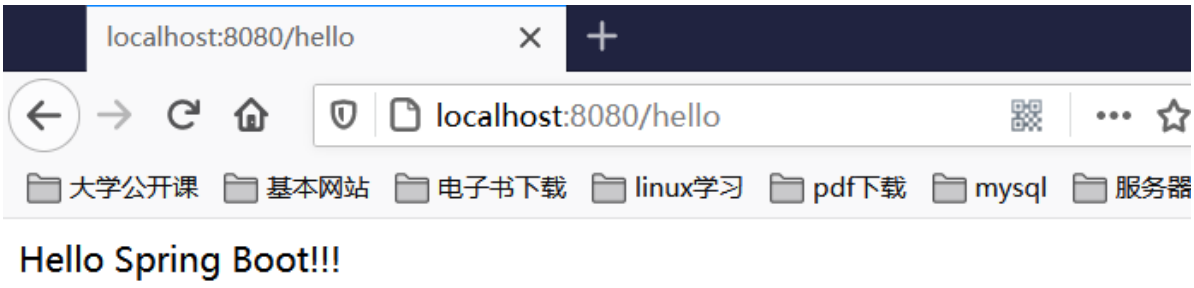
如果打包成功，则会在target目录下生成一个 jar包。



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！！！



执行结果



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！

```xml
<!--父依赖-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.5</version>
</parent>
```

点进去，发现还有一个父依赖。

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>2.4.5</version>
</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

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

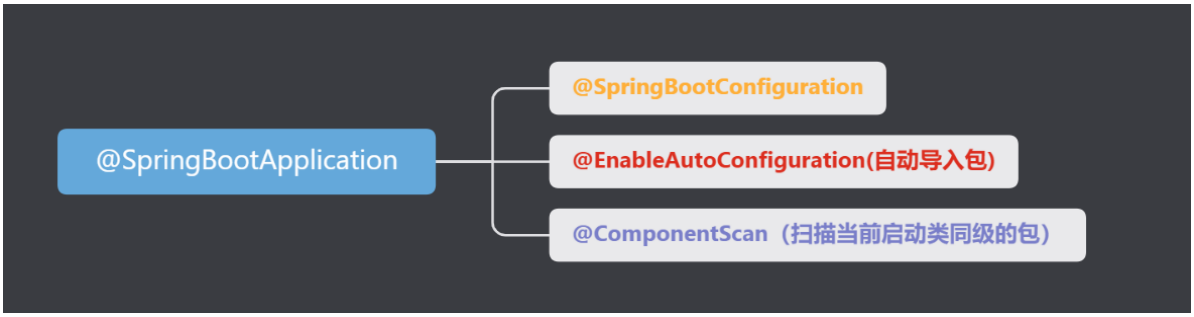
4.2 主启动类自动装配原理

默认启动类

```java
package cn.guardwhy;

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

@SpringBootApplication // 标注了一个主程序类，说明这是一个Spring Boot应用.
public class SpringbootDemo01Application {

    public static void main(String[] args) {
        // 启动了一个服务
        SpringApplication.run(SpringbootDemo01Application.class, args);
    }
}
```

@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
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

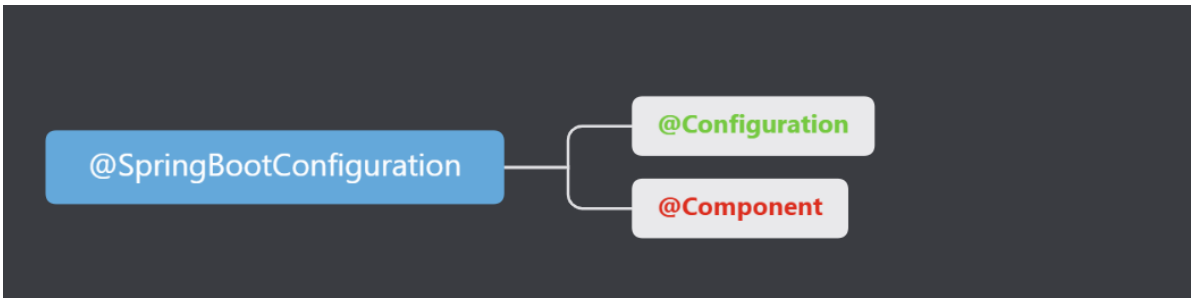
```
1  public @interface SpringBootApplication {
2    // 源码示例
3  }
```

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

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {
    @AliasFor(
        annotation = Component.class
    )
    String value() default "";

    boolean proxyBeanMethods() default true;
}
```

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。

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

    /**
     * Environment property that can be used to override when auto-configuration is
     * enabled.
     */
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    /**
     * Exclude specific auto-configuration classes such that they will never be applied.
     * @return the classes to exclude
     */
    Class<?>[] exclude() default {};

    /**
```

```
     * Exclude specific auto-configuration class names such that they will never be
     * applied.
     * @return the class names to exclude
     * @since 1.3.0
     */
    String[] excludeName() default {};

}
```

@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。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {

    /**
     * Base packages that should be registered with {@link AutoConfigurationPackages}.
     * <p>
     * Use {@link #basePackageClasses} for a type-safe alternative to String-based package
     * names.
     * @return the back package names
     * @since 2.3.0
     */
    String[] basePackages() default {};

    /**
     * Type-safe alternative to {@link #basePackages} for specifying the packages to be
     * registered with {@link AutoConfigurationPackages}.
     * <p>
     * Consider creating a special no-op marker class or interface in each package that
     * serves no purpose other than being referenced by this attribute.
     * @return the base package classes
     * @since 2.3.0
     */
    Class<?>[] basePackageClasses() default {};
}
```

@Import(AutoConfigurationImportSelector.class)

给容器导入组件，自动导入包的核心！！！

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY =
"spring.boot.enableautoconfiguration";
    Class<?>[] exclude() default {};

    String[] excludeName() default {};
}
```

AutoConfigurationImportSelector

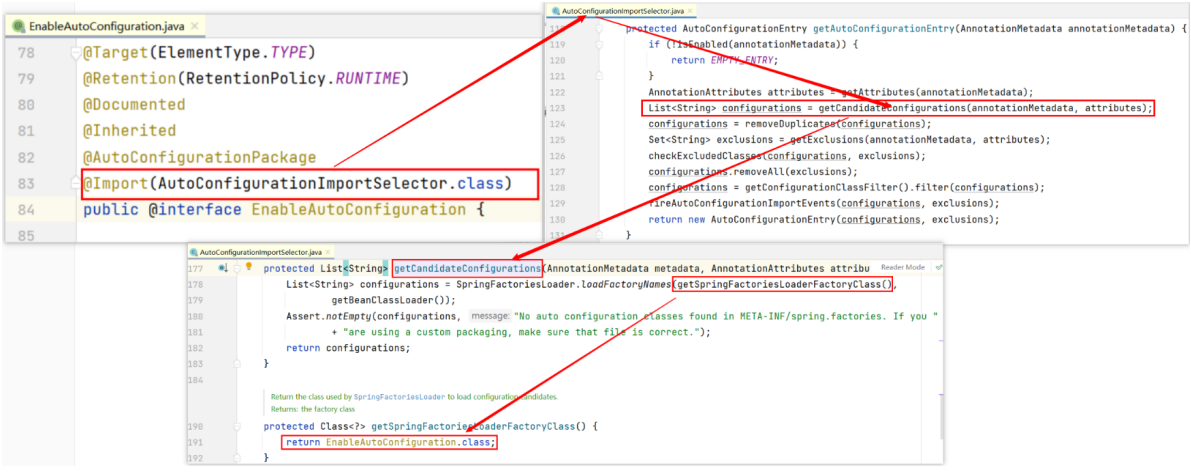
自动导入选择器，选择了啥？

获取所有的配置，继续点击。

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

获取候选的配置

```
// 方法
protected List<String> getCandidateConfigurations(AnnotationMetadata
metadata, AnnotationAttributes attributes) {
    List<String> configurations =
SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass()
,

getBeanClassLoader());
    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.");
    return configurations;
}
```



getAutoConfigurationEntry( )

获得自动配置的实体！！！

EnableAutoConfiguration.java

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

点击

AutoConfigurationImportSelector.java

```
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！！！

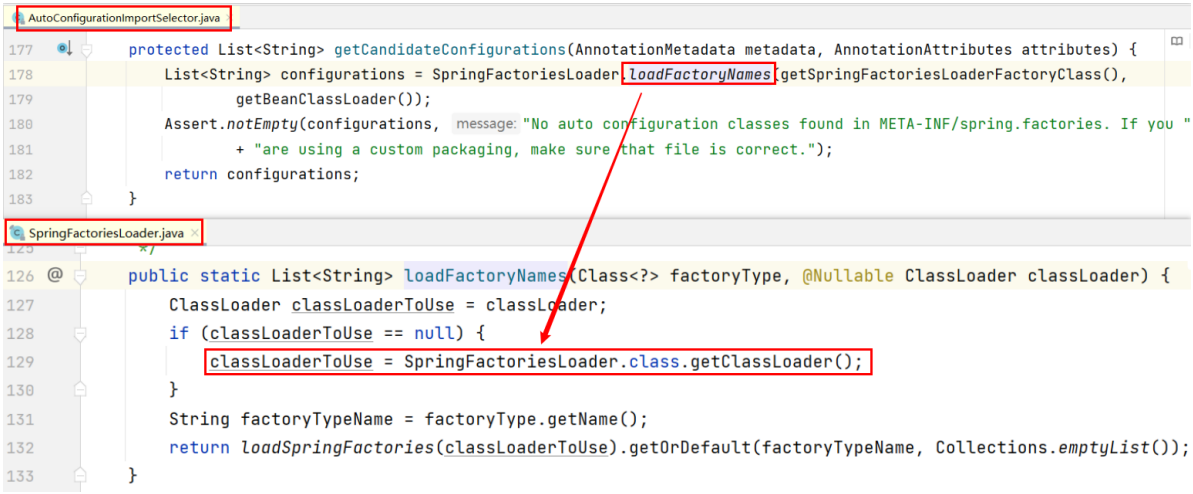
AutoConfigurationImportSelector.java

```
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     /**
186      * Return the class used by {@link SpringFactoriesLoader} to load configuration
187      * candidates.
188      * @return the factory class
189      */
190     protected Class<?> getSpringFactoriesLoaderFactoryClass() {
191         return EnableAutoConfiguration.class;
192     }
```

标志了EnableAutoConfiguration注解的类

public static List loadFactoryNames( )

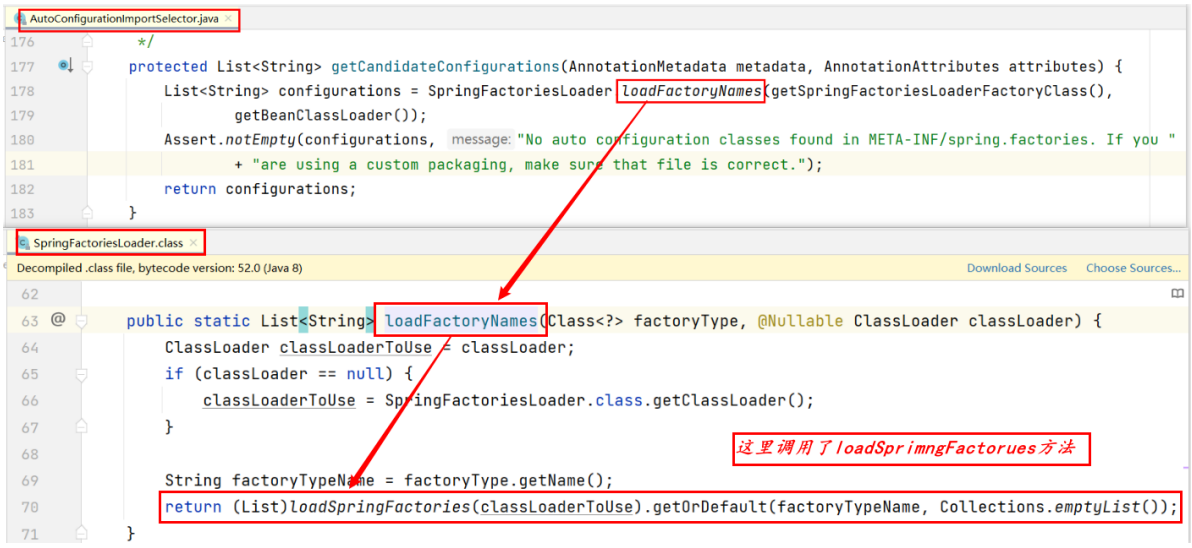
获取所有的加载配置！！！



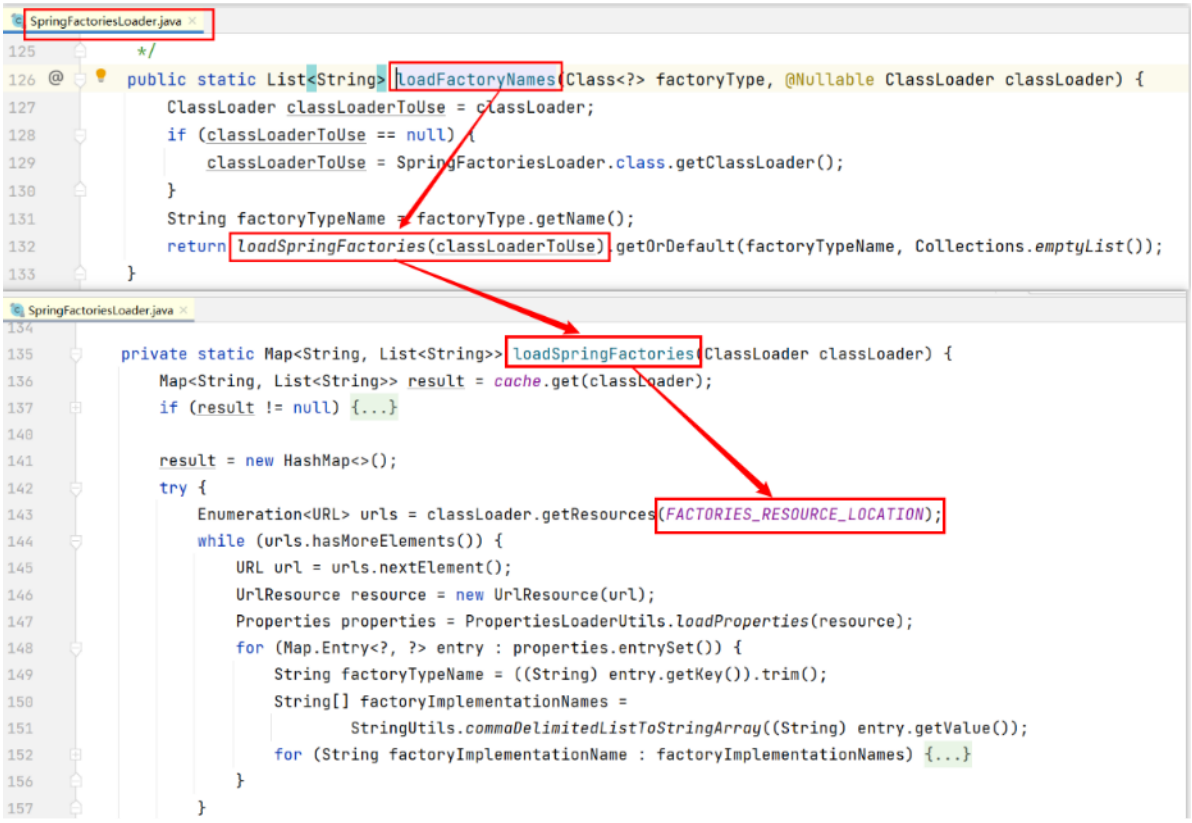
4.2.4 loadSpringFactories

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

```
 public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable ClassLoader classLoader) {
     ClassLoader classLoaderToUse = classLoader;
     if (classLoader == null) {
         classLoaderToUse = SpringFactoriesLoader.class.getClassLoader();
     }

     String factoryTypeName = factoryType.getName();
     return (List)loadSpringFactories(classLoaderToUse).getOrDefault(factoryTypeName, Collections.emptyList());
 }
```



继续点击查看 loadSpringFactories 方法



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

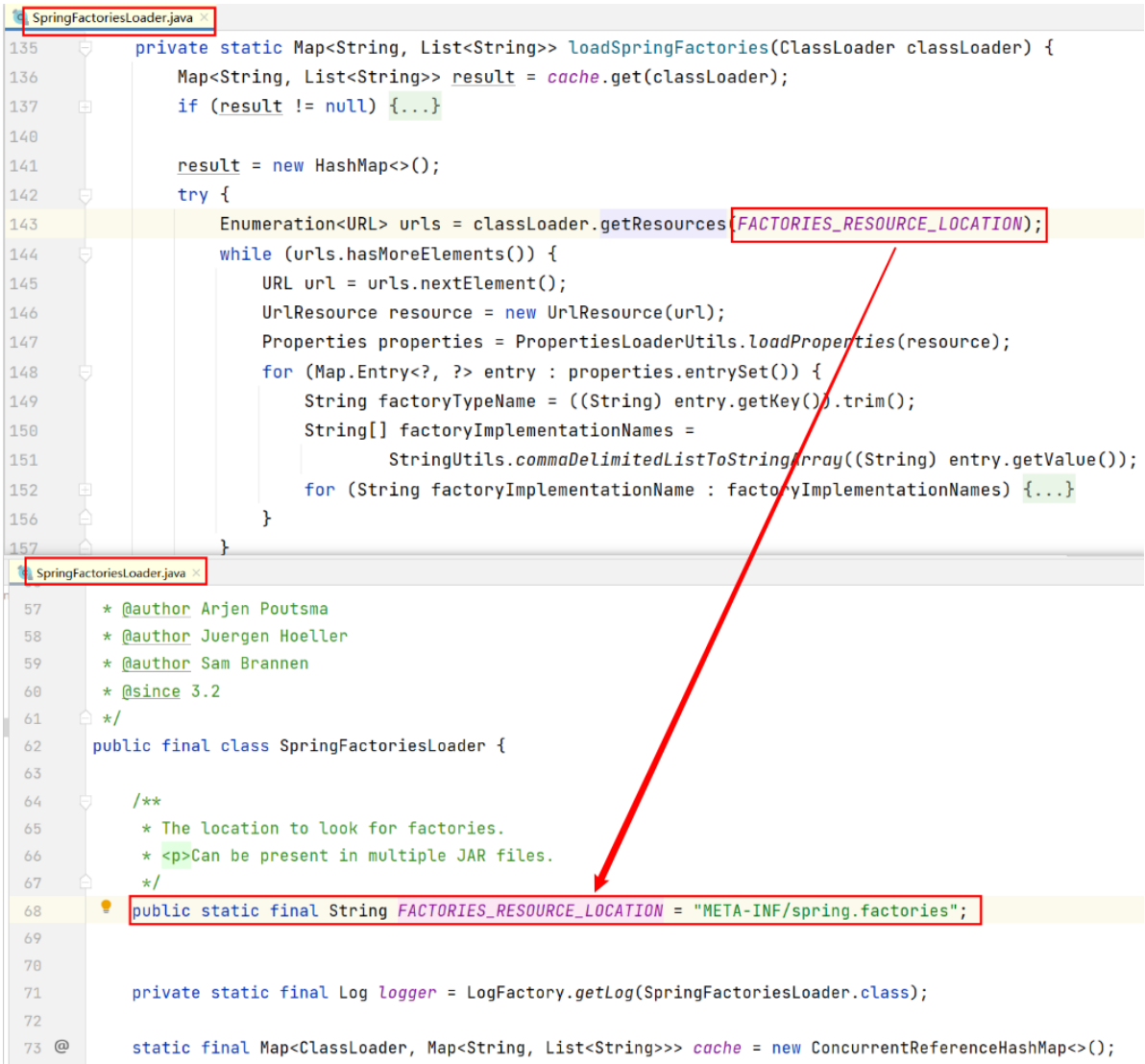
```
private static Map<String, List<String>> loadSpringFactories(ClassLoader classLoader) {
    Map<String, List<String>> result = cache.get(classLoader);
    // 获得classLoader，返回可以看到这里得到的EnableAutoConfiguration标注的类本身
    if (result != null) {
        return result;
    }

    result = new HashMap<>();
    try {
        Enumeration<URL> urls = classLoader.getResources(FACTORIES_RESOURCE_LOCATION);
        // 判断是否存在更多的元素
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            UrlResource resource = new UrlResource(url);
            // 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
            Properties properties = PropertiesLoaderUtils.loadProperties(resource);
            for (Map.Entry<?, ?> entry : properties.entrySet()) {
                String factoryTypeName = ((String) entry.getKey()).trim();
                String[] factoryImplementationNames =
                        StringUtils.commaDelimitedListToStringArray((String) entry.getValue());
                for (String factoryImplementationName : factoryImplementationNames) {
                    result.computeIfAbsent(factoryTypeName, key -> new ArrayList<>())
                            .add(factoryImplementationName.trim());
                }
            }
```

```
        }

        // Replace all lists with unmodifiable lists containing unique elements
        result.replaceAll((factoryType, implementations) -> implementations.stream().distinct()
.collect(Collectors.collectingAndThen(Collectors.toList(), Collections::unmodifiableList)));
        cache.put(classLoader, result);
    }
    catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load factories from location [" +
                                           FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
    return result;
}
```

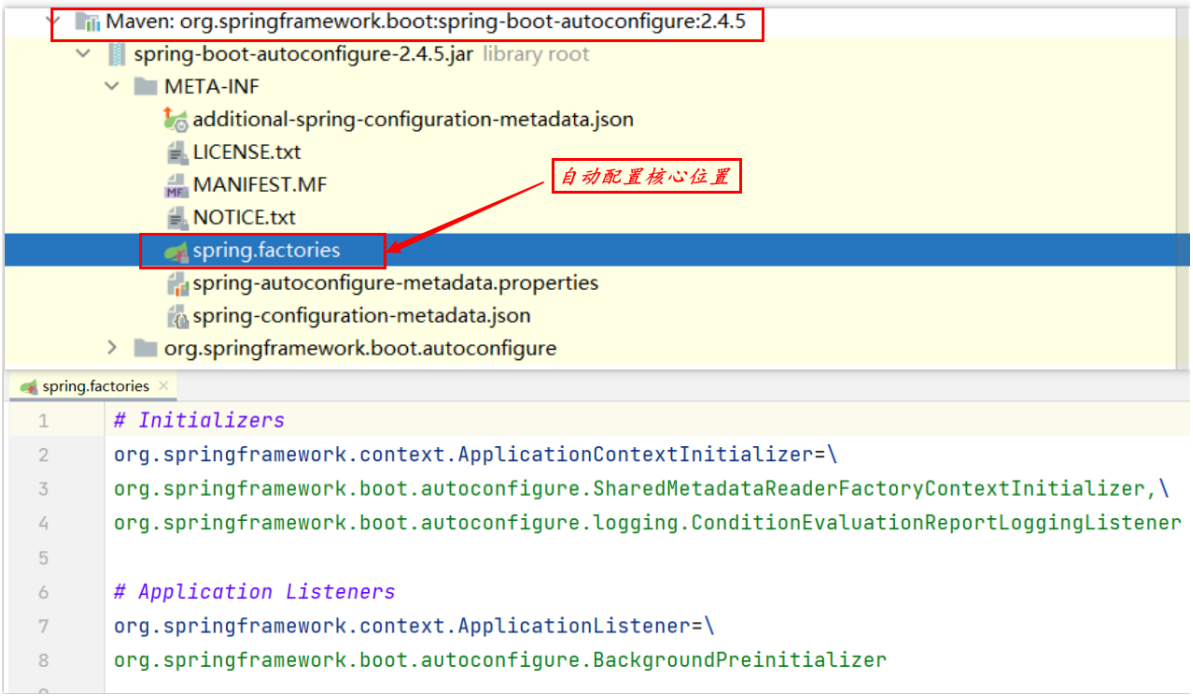
spring.factories

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!

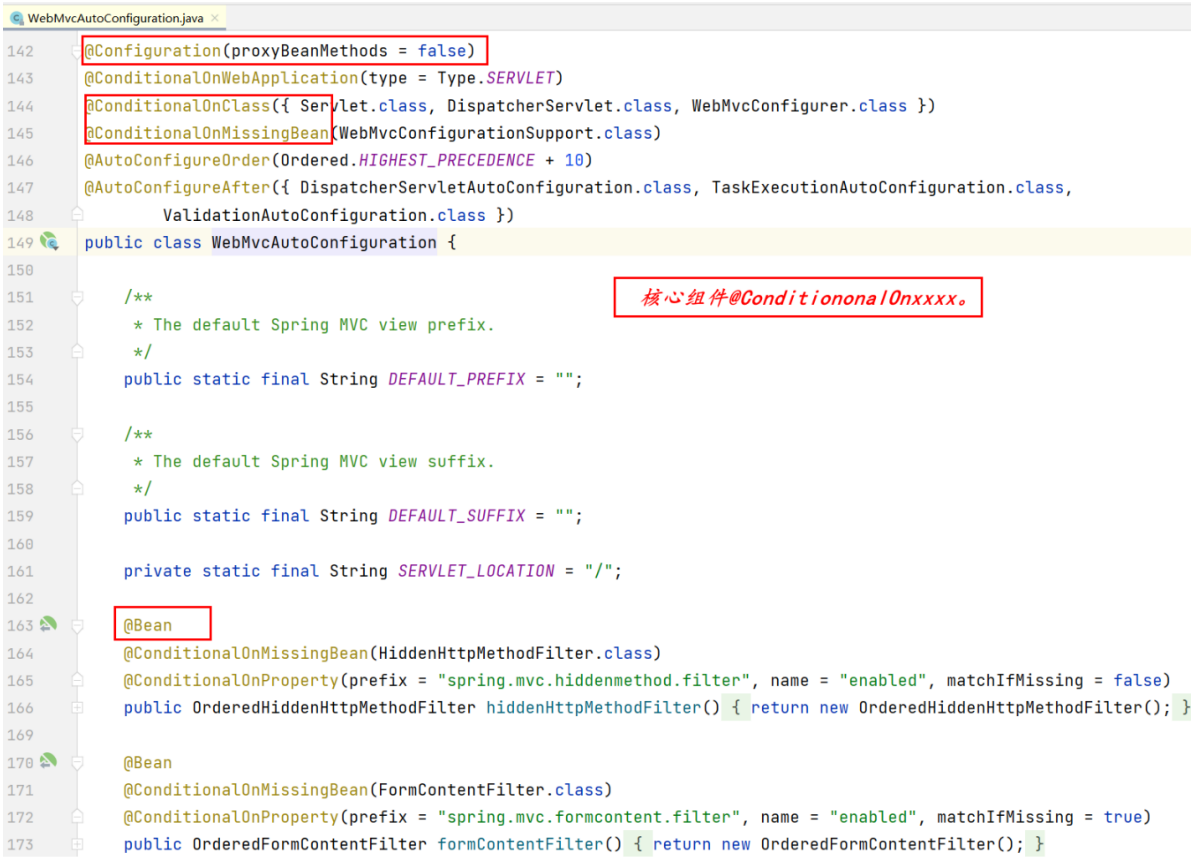


通过该文件找到相对应的jar包，从jar包中找到Springboot自动配置核心文件！！！



4.2.5 WebMvcAutoConfiguration

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



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一个服务。

```
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

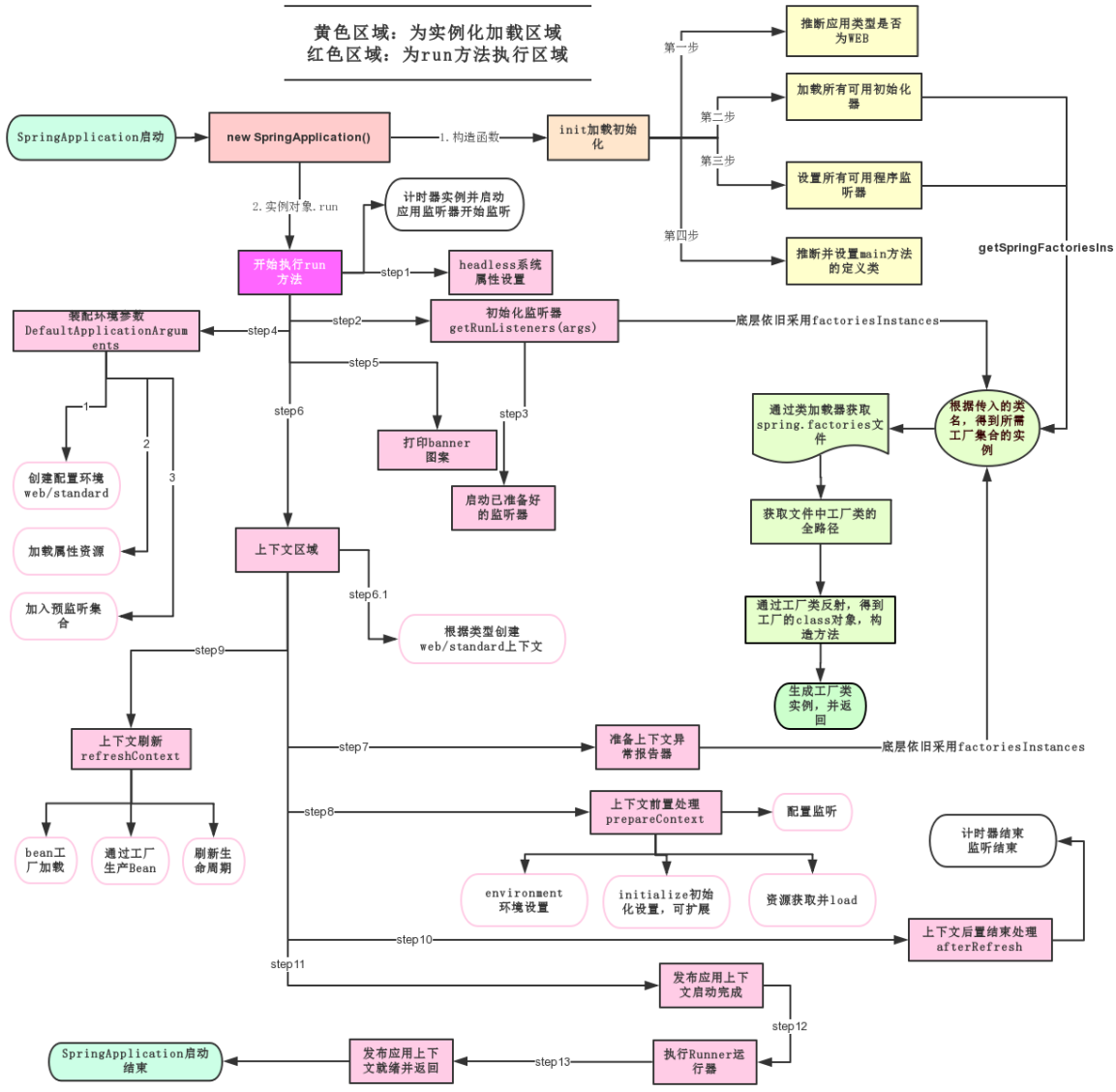
- 推断应用的类型是普通的项目还是Web项目。
  查找并加载所有可用初始化器， 设置到initializers属性中。
- 找出所有的应用程序监听器，设置到listeners属性中。
- 推断并设置main方法的定义类，找到运行的主类。

查看SpringApplication的构造器

```
1  public SpringApplication(ResourceLoader resourceLoader, Class<?>...
   primarySources) {
2      this.sources = new LinkedHashSet();
3      this.bannerMode = Mode.CONSOLE;
4      this.logStartupInfo = true;
5      this.addCommandLineProperties = true;
6      this.addConversionService = true;
7      this.headless = true;
8      this.registerShutdownHook = true;
9      this.additionalProfiles = Collections.emptySet();
10     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 =
    this.getBootstrapRegistryInitializersFromSpringFactories();
19
     this.setInitializers(this.getSpringFactoriesInstances(ApplicationContextIni
    tializer.class));
20
     this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.clas
    s));
21      this.mainApplicationClass = this.deduceMainApplicationClass();
22  }
```

4.3.2 run方法的执行



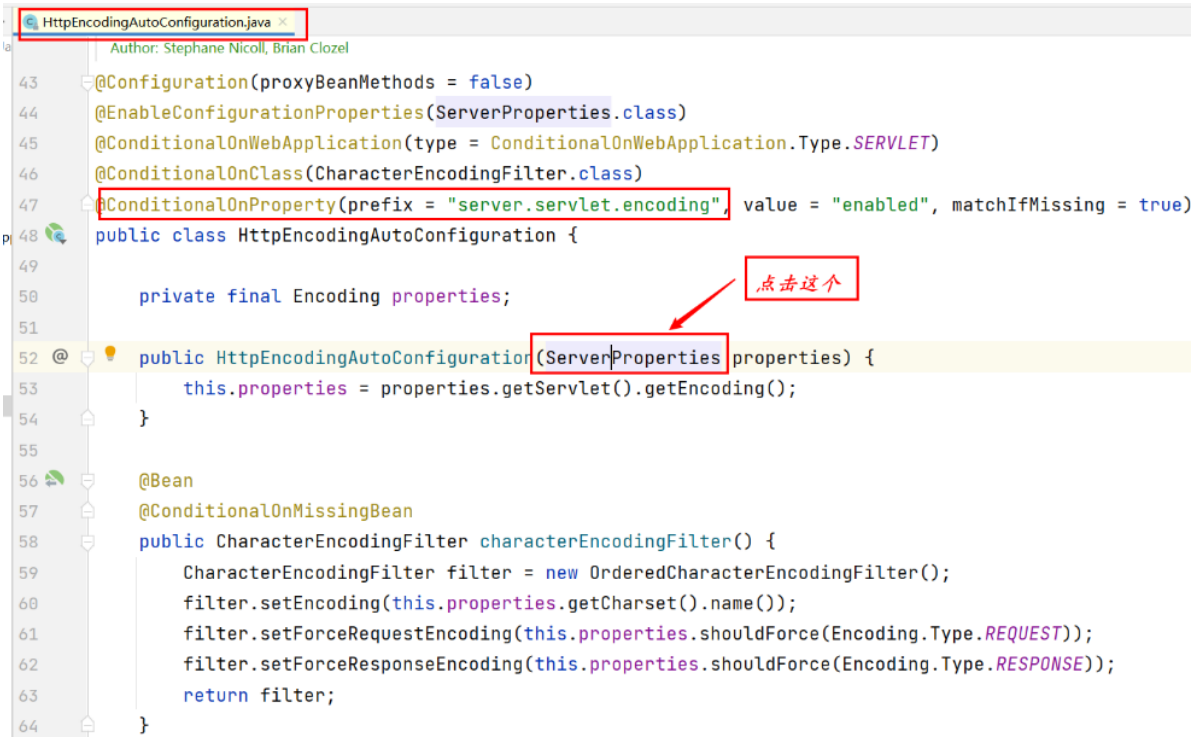
4.4 分析自动配置原理

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

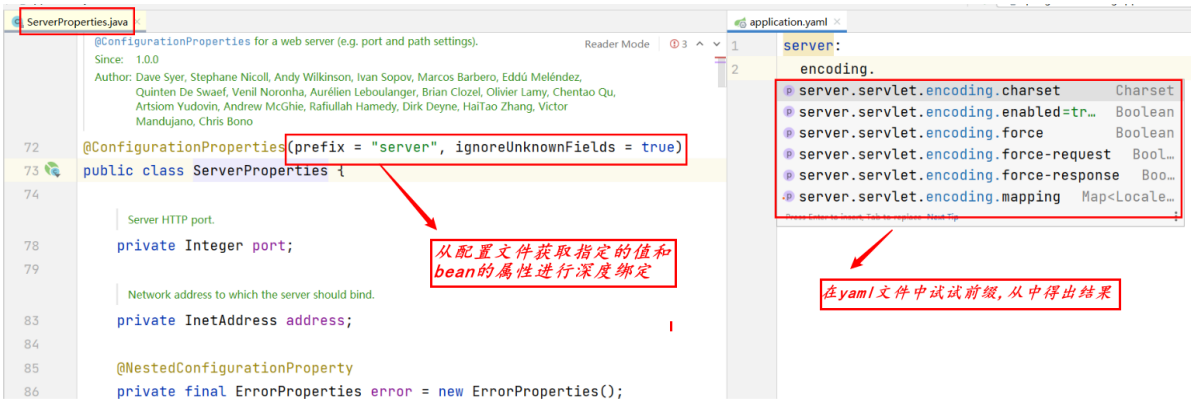
```
package org.springframework.boot.autoconfigure.web.servlet;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnWebApplication
;
import org.springframework.boot.autoconfigure.web.ServerProperties;
import
org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import
org.springframework.boot.web.servlet.filter.OrderedCharacterEncodingFilter;
import
org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFact
ory;
import org.springframework.boot.web.servlet.server.Encoding;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;
import org.springframework.web.filter.CharacterEncodingFilter;

// 表示这是一个配置类， 也可以给容器中添加组件。
@Configuration(proxyBeanMethods = false)
// 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
@EnableConfigurationProperties(ServerProperties.class)

// 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。

// 判断当前应用是否是web应用，如果是当前配置类生效。
@ConditionalOnWebApplication(type =
ConditionalOnWebApplication.Type.SERVLET)

//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。
@ConditionalOnClass(CharacterEncodingFilter.class)

/*
    判断配置文件中是否存在某个配置: server.servlet.encoding
    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 =
true，也是默认生效的。
*/
@ConditionalOnProperty(prefix = "server.servlet.encoding", value =
"enabled", matchIfMissing = true)
public class HttpEncodingAutoConfiguration {
    //它已经和SpringBoot的配置文件映射了
    private final Encoding properties;
    //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
    OrderedCharacterEncodingFilter();
50          filter.setEncoding(this.properties.getCharset().name());
51
    filter.setForceRequestEncoding(this.properties.shouldForce(Encoding.Type.REQ
    UEST));
52
    filter.setForceResponseEncoding(this.properties.shouldForce(Encoding.Type.RE
    SPONSE));
53          return filter;
54      }
55
56      @Bean
57      public LocaleCharsetMappingsCustomizer localeCharsetMappingsCustomizer()
    {
58          return new LocaleCharsetMappingsCustomizer(this.properties);
59      }
60
61      static class LocaleCharsetMappingsCustomizer
62              implements
    WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>, Ordered {
63
64          private final Encoding properties;
65
66          LocaleCharsetMappingsCustomizer(Encoding properties) {
67              this.properties = properties;
68          }
69
70          @Override
71          public void customize(ConfigurableServletWebServerFactory factory) {
72              if (this.properties.getMapping() != null) {
73
    factory.setLocaleCharsetMappings(this.properties.getMapping());
74              }
75          }
76
77          @Override
78          public int getOrder() {
79              return 0;
80          }
81      }
82  }
```

根据当前不同的条件判断，决定这个配置类是否生效。



- 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。



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。

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

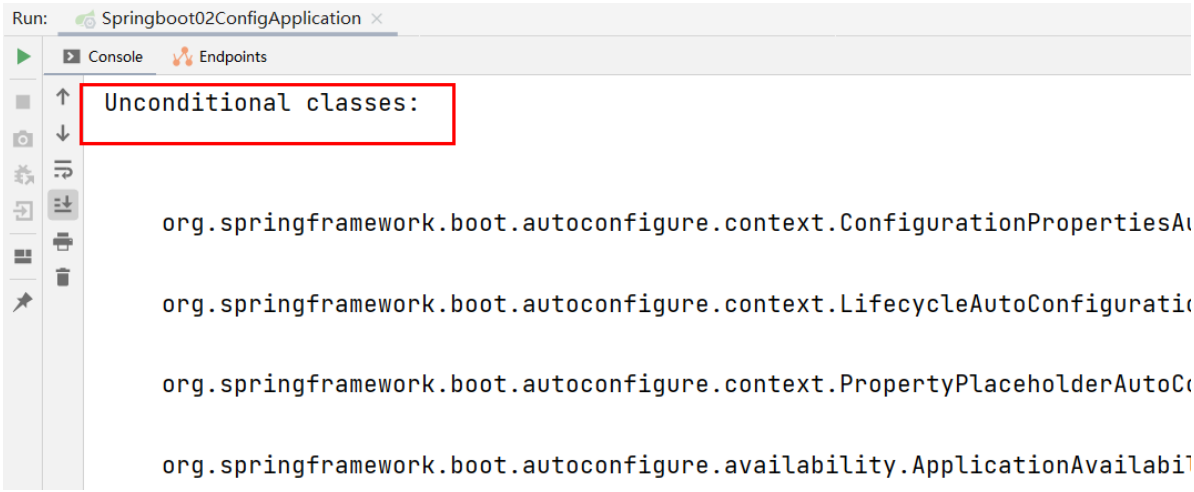
Positive matches:（自动配置类启用的：正匹配）

```
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）



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

```
语法结构 ： key=value
```

- application.yml

```
语法结构 ：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.yml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：

```
<server>
  <port>8089<port>
</server>
```

yaml配置：

```
server:
    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。

```
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。

```
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。

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

对象、map(键值对)

语法格式

```
#对象、集合格式
k:
 v1:
 v2:
```

案例说明

```
# 对象写法
stu:
  name: guardwhy
  age: 28
```

行内写法

```
# 行内写法
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

```
# 数组
start:
  - kobe
  - james
  - curry
```

行内写法

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

修改SpringBoot端口号

```
server:
    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

```
package cn.guardwhy.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.stereotype.Component;

@Data
@AllArgsConstructor
@NoArgsConstructor
// 学生类
@Component
public class Student {
    private String username;
    private Integer userage;
}
```

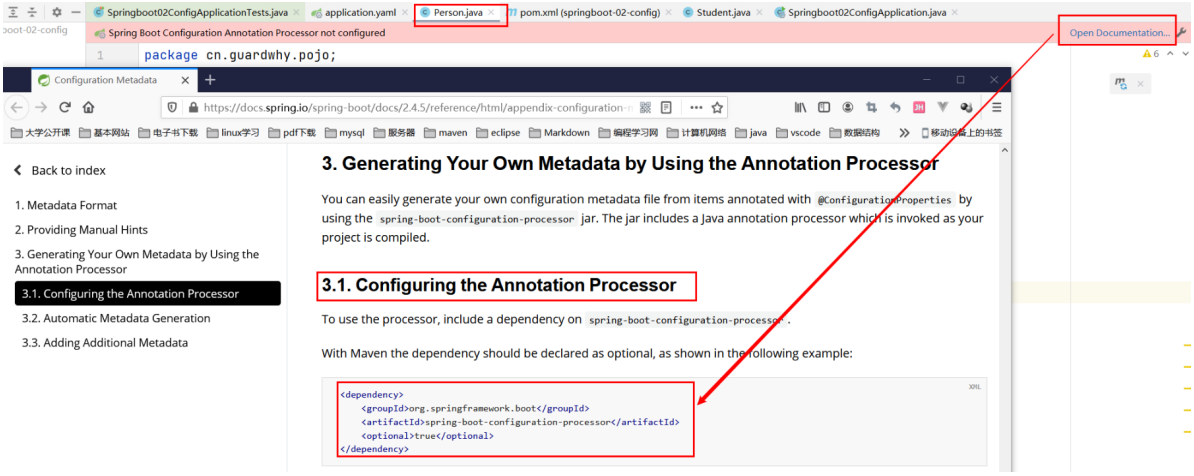
编写实体类 Person

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

```
package cn.guardwhy.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.List;
import java.util.Map;

// Person类
@Component
@ConfigurationProperties(prefix = "person")
@Data
@AllArgsConstructor
```

```java
@NoArgsConstructor
public class Person {
    private String name;
    private Integer age;
    private Date birthday;
    private Map<String, Object> maps;
    private List<Object> lists;
    // 导入学生类
    private Student student;
}
```

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！！！




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

application.yaml

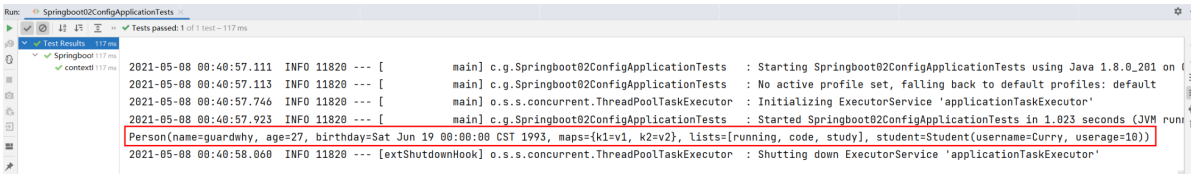
```yaml
person:
  name: guardwhy
  age: 27
  birthday: 1993/06/19
  maps: {k1: v1, k2: v2}
  lists:
    - running
    - code
    - study
  student:
    username: Curry
    userage: 10
```

测试案例

```java
package cn.guardwhy;

import cn.guardwhy.domain.Person;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
```

```java

@SpringBootTest
class Springboot02ConfigApplicationTests {

    // 注入数据
    @Autowired
    private Person person;

    @Test
    void contextLoads() {
        System.out.println(person);
    }
}
```

5.3.2 执行结果



5.3.2 properties方式

编写实体类 Person

```java
package cn.guardwhy.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.PropertySource;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.List;
import java.util.Map;

// Person类
@Component
@Data
@AllArgsConstructor
@NoArgsConstructor
// 加载指定的配置文件
@PropertySource(value = "classpath:Person.properties")
public class Person {
    // SPEL表达式取出配置文件的值
    @Value("${name}")
    private String name;
    @Value("${age}")
    private Integer age;
    private Date birthday;
    private Map<String, Object> maps;
    private List<Object> lists;
    // 导入学生类
    private Student student;
```

```
33 }
```

加载指定配置文件

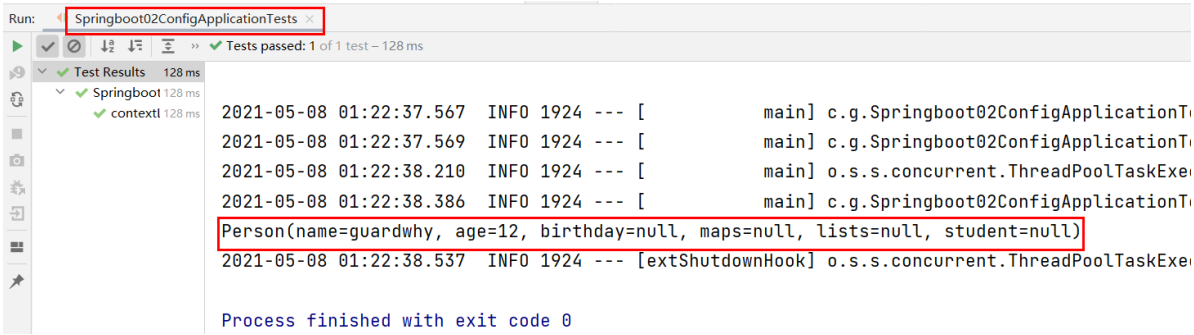
新建一个person.properties文件

```
name=guardwhy
age=12
```

测试案例

```
package cn.guardwhy;

import cn.guardwhy.domain.Person;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class Springboot02ConfigApplicationTests {

    // 注入数据
    @Autowired
    private Person person;

    @Test
    void contextLoads() {
        System.out.println(person);
    }
}
```

5.3.3 执行结果

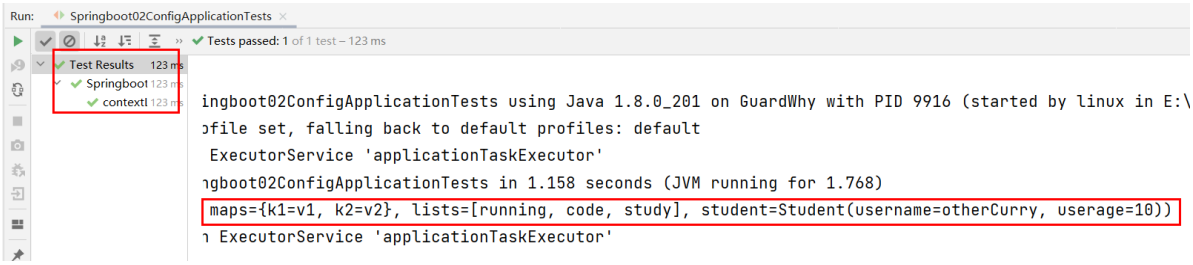


5.3.4 配置文件占位符

application.yaml

```yaml
person:
  name: guardwhy${random.uuid} # 随机uuid
  age: ${random.int} # 随机int
  birthday: 1993/06/19
  maps: {k1: v1, k2: v2}
  lists:
    - running
    - code
    - study
  student:
    # 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
    username: ${student.test:other}Curry
    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

```java
package cn.guardwhy.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.PropertySource;
import org.springframework.stereotype.Component;
import org.springframework.validation.annotation.Validated;

import javax.validation.constraints.Email;
import java.util.Date;
import java.util.List;
import java.util.Map;

// Person类
@Component
@ConfigurationProperties(prefix = "person")
@Data
```

```java
@AllArgsConstructor
@NoArgsConstructor
// 数据校验
@Validated
public class Person {
    @Email(message="用户名格式不合法!!!")
    private String name;
    private Integer age;
    private Date birthday;
    private Map<String, Object> maps;
    private List<Object> lists;
    // 导入学生类
    private Student student;
}
```

5.4.2 pom.xml

添加Validated相关依赖

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

5.4.3 运行结果

运行结果： default message [用户名格式不合法!!!];

```
.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.springfram
oot.context.properties.bind.validation.ValidationBindHandler.validateAndPush(ValidationBindHandler.java:141)
```

使用数据校验，可以保证数据的正确性

```java
@NotNull(message="名字不能为空")
private String userName;
@Max(value=120,message="年龄最大不能查过120")
private int age;
@Email(message="邮箱格式错误")
private String email;
空检查
@Null    验证对象是否为null
@NotNull    验证对象是否不为null, 无法查检长度为0的字符串
@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.
@NotEmpty  检查约束元素是否为NULL或者是EMPTY.

Booelan检查
@AssertTrue   验证 Boolean 对象是否为 true
@AssertFalse    验证 Boolean 对象是否为 false

长度检查
@Size(min=, max=) 验证对象（Array,Collection,Map,String）长度是否在给定的范围之内
@Length(min=, max=) string is between min and max included.
日期检查
@Past    验证 Date 和 Calendar 对象是否在当前时间之前
@Future    验证 Date 和 Calendar 对象是否在当前时间之后
```

```
@Pattern    验证 String 对象是否符合正则表达式的规则
.......等等
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

```properties
#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。
#启动SpringBoot，就可以看到已经切换到test下的配置。
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

```yaml
server:
  port: 8081
# 选择要激活哪个环境
spring:
  profiles:
    active: dev

---
server:
  port: 8082
spring:
  # 配置开发环境名称
  profiles: dev

---
server:
  # 配置测试环境名称
  port: 8083
spring:
  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

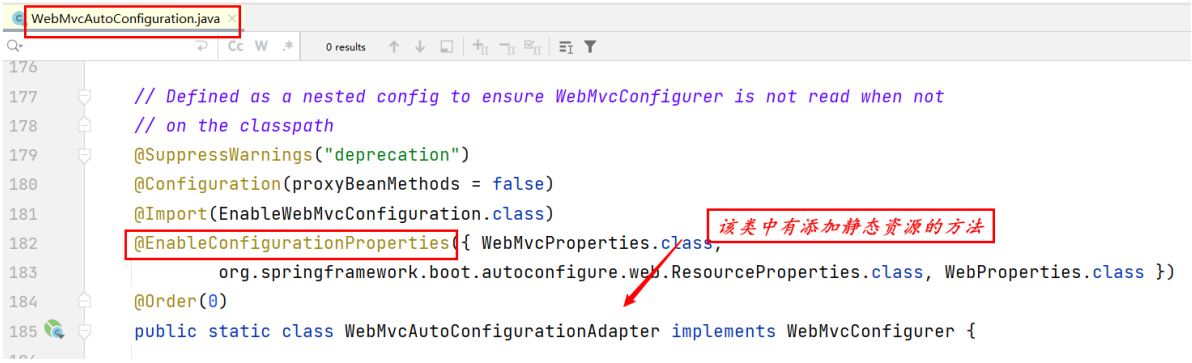
6.1 静态资源处理

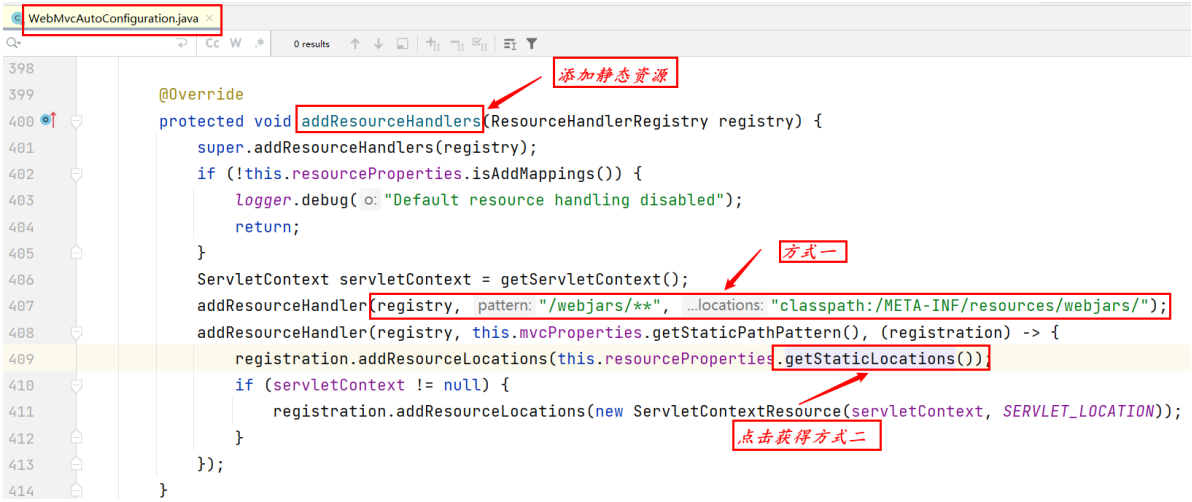
SpringBoot中，SpringMVC的web配置都在 WebMvcAutoConfiguration 这个配置类里面，其中有一个方法：`addResourceHandlers` 添加资源处理

源码分析

Springboot主要帮我们配置了什么?

[(?)AutoConfigurartion: 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！！]





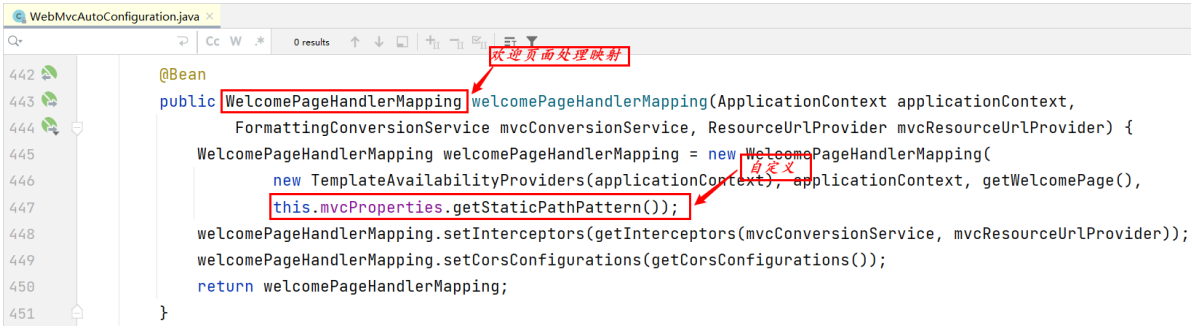
小结

- 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

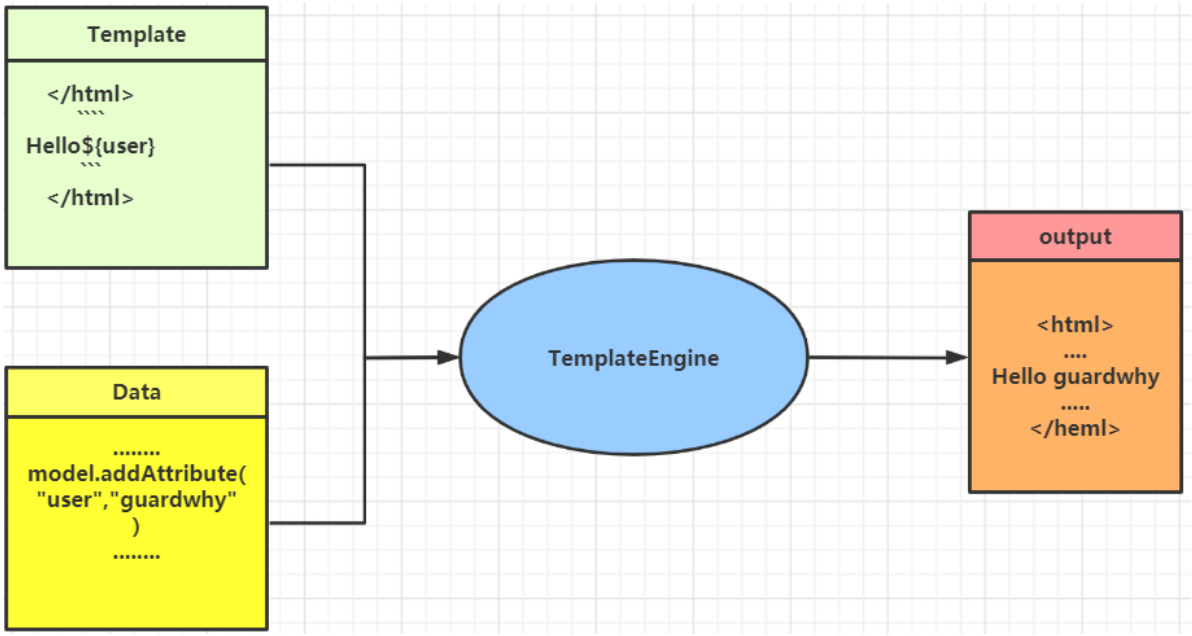


欢迎页，静态资源文件夹下的所有 index.html 页面；被 /** 映射。比如访问 http://localhost:8080/ ，就会找静态资源文件夹下的 index.html
新建一个 index.html ，在上面的3个目录【public 、static、resource】中任意一个；然后访问测试 http://localhost:8080/ 看结果！

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

```
<!--thymeleaf版本-->
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring5</artifactId>
</dependency>
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-java8time</artifactId>
</dependency>
```

源码分析

Thymeleaf的自动配置类：ThymeleafProperties

```
@ConfigurationProperties(prefix = "spring.thymeleaf")
public class ThymeleafProperties {

    private static final Charset DEFAULT_ENCODING = StandardCharsets.UTF_8;
    // 前缀
    public static final String DEFAULT_PREFIX = "classpath:/templates/";
    // 后缀
    public static final String DEFAULT_SUFFIX = ".html";

    /**
     * Whether to check that the template exists before rendering it.
```

```
     */
    private boolean checkTemplate = true;

    /**
     * Whether to check that the templates location exists.
     */
    private boolean checkTemplateLocation = true;

    /**
     * Prefix that gets prepended to view names when building a URL.
     */
    private String prefix = DEFAULT_PREFIX;

    /**
     * Suffix that gets appended to view names when building a URL.
     */
    private String suffix = DEFAULT_SUFFIX;

    /**
     * Template mode to be applied to templates. See also Thymeleaf's
TemplateMode enum.
     */
    private String mode = "HTML";

    /**
     * Template files encoding.
     */
    private Charset encoding = DEFAULT_ENCODING;

    /**
     * Whether to enable template caching.
     */
    private boolean cache = true;

    /**
     * Whether to enable Thymeleaf view resolution for Web frameworks.
     */
    private boolean enabled = true;
}
```

- 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

```
package cn.guardwhy.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class TestController {
    @RequestMapping("/t1")
    public String test01(Model model){
```

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

使用thymeleaf，需要在html文件中导入命名空间的约束。

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

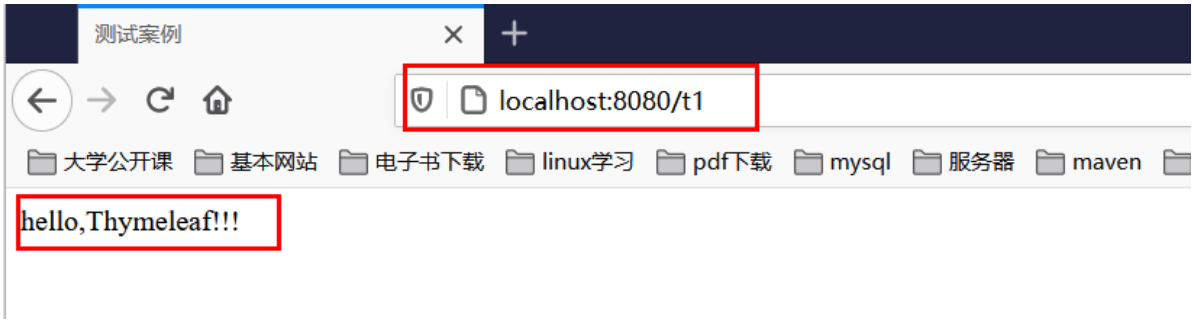
编写前端test.html页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>测试案例</title>
</head>
<body>

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

  <!--所有的html元素都可以被thymeleaf替换接管; th:元素名-->
  <div th:text="${message}"></div>
</body>
</html>
```

要在页面内容添加以下格式,不然会报错！！！

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

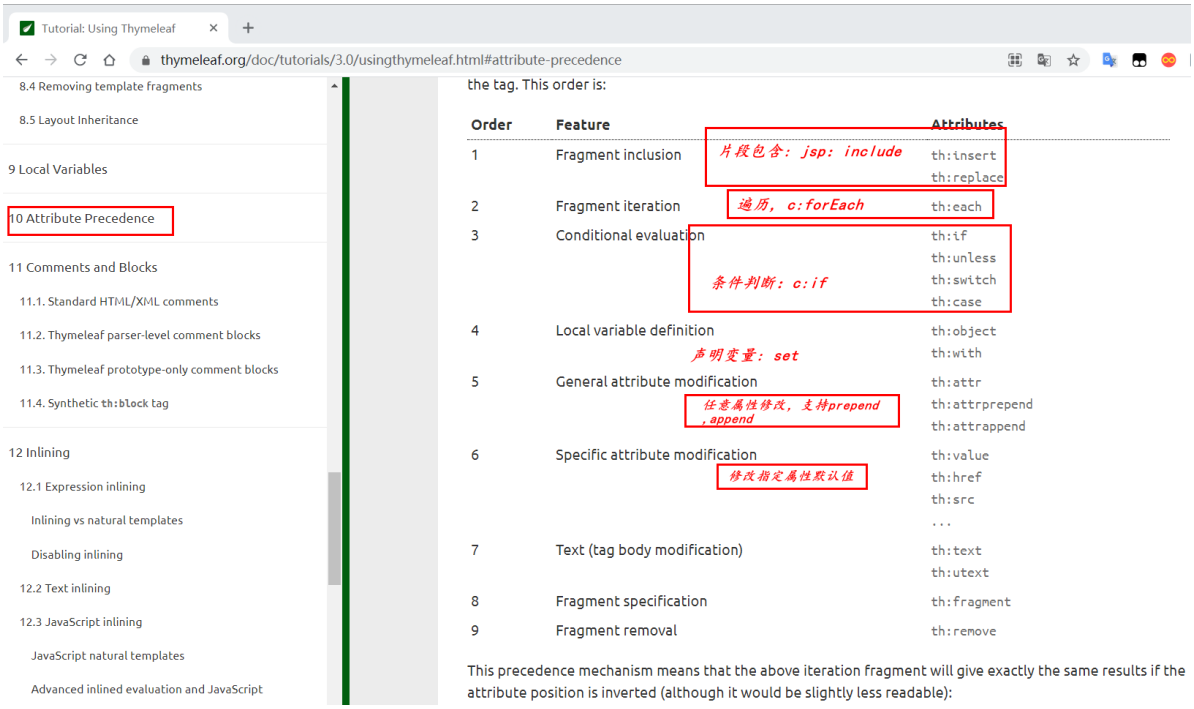
测试结果



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



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	${...}
选择变量表达式	*{...}
消息表达式	#{...}
链接URL表达式	@{...}
片段表达式	~{...}

6.3.4 案例测试

编写一个Controller，放一些数据。

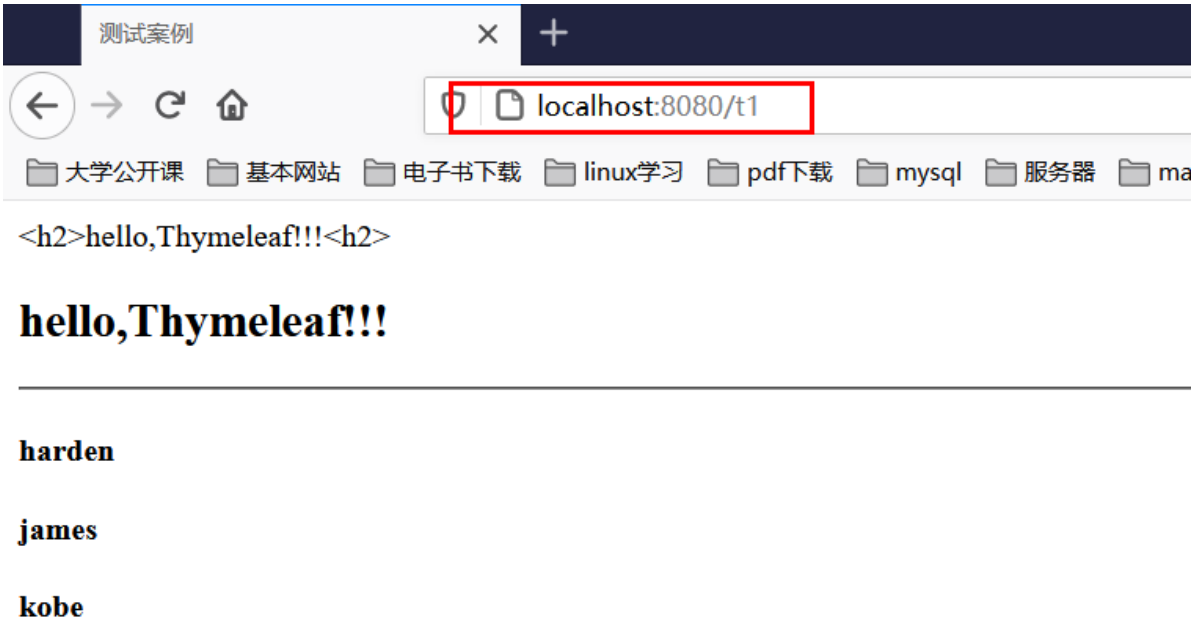
```
package cn.guardwhy.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.Arrays;

@Controller
public class TestController {
    @RequestMapping("/t1")
    public String test01(Model model){
        model.addAttribute("message", "<h2>hello,Thymeleaf!!!<h2>");
        model.addAttribute("lists", Arrays.asList("harden",
"james","kobe"));
        return "test";
    }
}
```

编写前端test.html页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>测试案例</title>
</head>
<body>

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

  <!--所有的html元素都可以被thymeleaf替换接管; th:元素名-->
  <div th:text="${message}"></div>
  <!--不转义-->
  <div th:utext="${message}"></div>

  <hr/>

    <!--遍历数据-->
    <h4 th:each="lists:${lists}" th:text="${lists}"></h4>
</body>
```

```
21 </html>
```

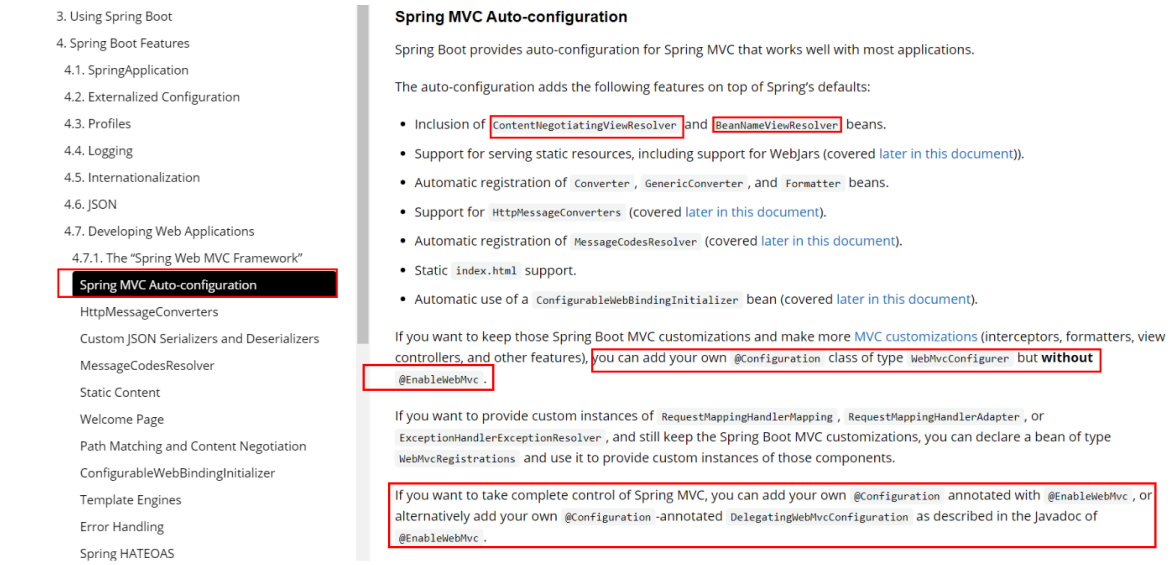
启动项目测试！！！



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



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
  WebApplicationObjectSupport implements ViewResolver, Ordered,
  InitializingBean {
2     @Nullable
3     private ContentNegotiationManager contentNegotiationManager;
```

```
    private final ContentNegotiationManagerFactoryBean cnmFactoryBean = new ContentNegotiationManagerFactoryBean();

    private boolean useNotAcceptableStatusCode = false;

    @Nullable
    private List<View> defaultViews;

    @Nullable
    private List<ViewResolver> viewResolvers;

    private int order = Ordered.HIGHEST_PRECEDENCE;
}
```

ViewResolver接口解析视图

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

方法重写

```
@Override
@Nullable
public View resolveViewName(String viewName, Locale locale) throws Exception {
    RequestAttributes attrs = RequestContextHolder.getRequestAttributes();
    Assert.state(attrs instanceof ServletRequestAttributes, "No current ServletRequestAttributes");
    List<MediaType> requestedMediaTypes = getMediaTypes(((ServletRequestAttributes) attrs).getRequest());
    if (requestedMediaTypes != null) {
        // 获取候选的视图
        List<View> candidateViews = getCandidateViews(viewName, locale, requestedMediaTypes);
        // 得到最好的视图
        View bestView = getBestView(candidateViews, requestedMediaTypes, attrs);
        if (bestView != null) {
            return bestView;
        }
    }
}
```

获取候选的视图

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

```
        for (ViewResolver viewResolver : this.viewResolvers) {
            // 封装成一个对象
            View view = viewResolver.resolveViewName(viewName, locale);
            if (view != null) {
                // 添加到候选的视图
                candidateViews.add(view);
            }
            for (MediaType requestedMediaType : requestedMediaTypes) {
                List<String> extensions =
this.contentNegotiationManager.resolveFileExtensions(requestedMediaType);
                for (String extension : extensions) {
                    String viewNameWithExtension = viewName + '.' +
extension;
                    view =
viewResolver.resolveViewName(viewNameWithExtension, locale);
                    if (view != null) {
                        candidateViews.add(view);
                    }
                }
            }
        }
    }
    if (!CollectionUtils.isEmpty(this.defaultViews)) {
        candidateViews.addAll(this.defaultViews);
    }
    // 返回出去
    return candidateViews;
}
```

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

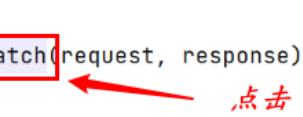
```
package cn.guardwhy.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.View;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import java.util.Locale;

// 拓展 springMVC
@Configuration
public class MyMvcConfig implements WebMvcConfigurer {
    // ViewResolver 实现了视图解析器接口类,可以将它看做视图解析器
    @Bean
    public ViewResolver myViewResolver(){
        return new MyViewResolver();
    }

    // 自定义了一个自己的视图解析器MyViewResolver
    public static class MyViewResolver implements ViewResolver{
```

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

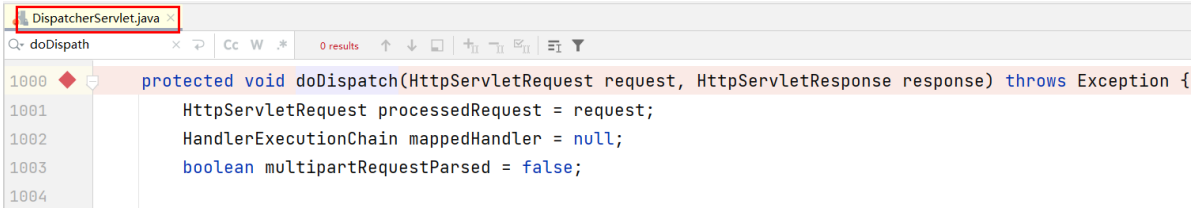
判断图解析器有没有起作用呢?

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

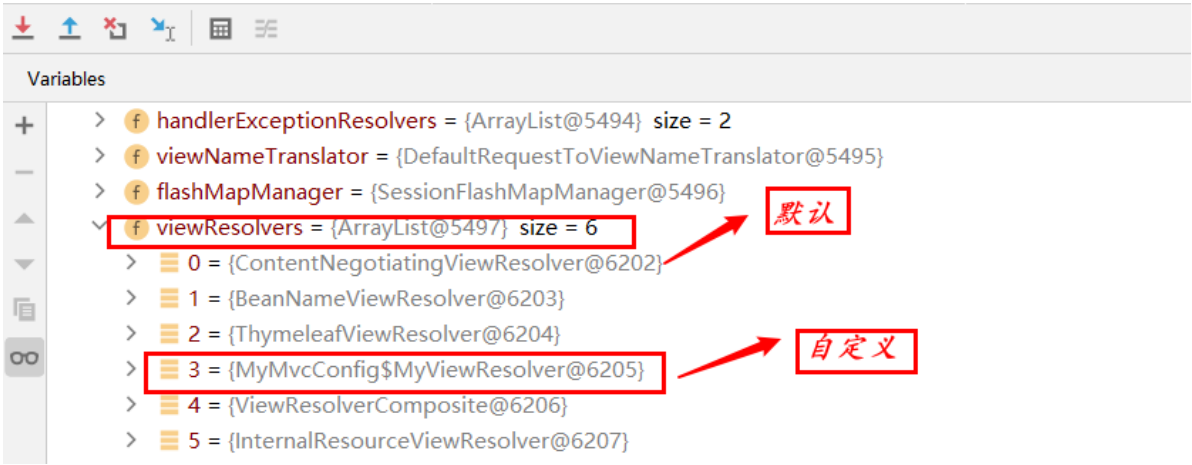
```
    try {
        doDispatch(request, response);
    }
    finally {
        if (!WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
            // Restore the original attribute snapshot, in case of an include.
            if (attributesSnapshot != null) {
                restoreAttributesAfterInclude(request,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

点击

```
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(WebProperties.class)
public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration implements ResourceLoaderAware {

    private static final Log logger = LogFactory.getLog(WebMvcConfigurer.class);

    private final Resources resourceProperties;

    private final WebMvcProperties mvcProperties;
```

点击

局部搜索`dateFormat`

```
@Deprecated
@DeprecatedConfigurationProperty(replacement = "spring.mvc.format.date")
public String getDateFormat() {
    return this.format.getDate();
}
```

点击

```
/**
 * 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

```
# 自定义的配置日期格式化
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。

```
package cn.guardwhy.config;

import org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

// 如果要扩展SpringMVC,官方建议这样做!!!
@Configuration
public class MyMvcConfig implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/user").setViewName("test");
    }
}
```

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

```
package cn.guardwhy.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import
org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
@EnableWebMvc   // 为什么加了一个注解，自动配置就失效了?
public class MyMvcConfig implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/user").setViewName("test");
    }
}
```

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。

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

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

```
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(WebProperties.class)
public static class EnableWebMvcConfiguration extends
DelegatingWebMvcConfiguration implements ResourceLoaderAware {

    private static final Log logger =
LogFactory.getLog(WebMvcConfigurer.class);

    private final Resources resourceProperties;

    private final WebMvcProperties mvcProperties;

    private final WebProperties webProperties;

    private final ListableBeanFactory beanFactory;

    private final WebMvcRegistrations mvcRegistrations;

    private final ResourceHandlerRegistrationCustomizer
resourceHandlerRegistrationCustomizer;

    private ResourceLoader resourceLoader;
}
```

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

```java
@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);
  }
}
```

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

```java
@Configuration
@EnableWebMvc   // 为什么加了一个注解，自动配置就失效了?
public class MyMvcConfig implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController( urlPathOrPattern: "/user").setViewName("test");
    }
}
```

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

点击，继续深入

5、`DelegatingWebMvcConfiguration`继承`WebMvcConfigurationSupport`。

```java
@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！！

```java
@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

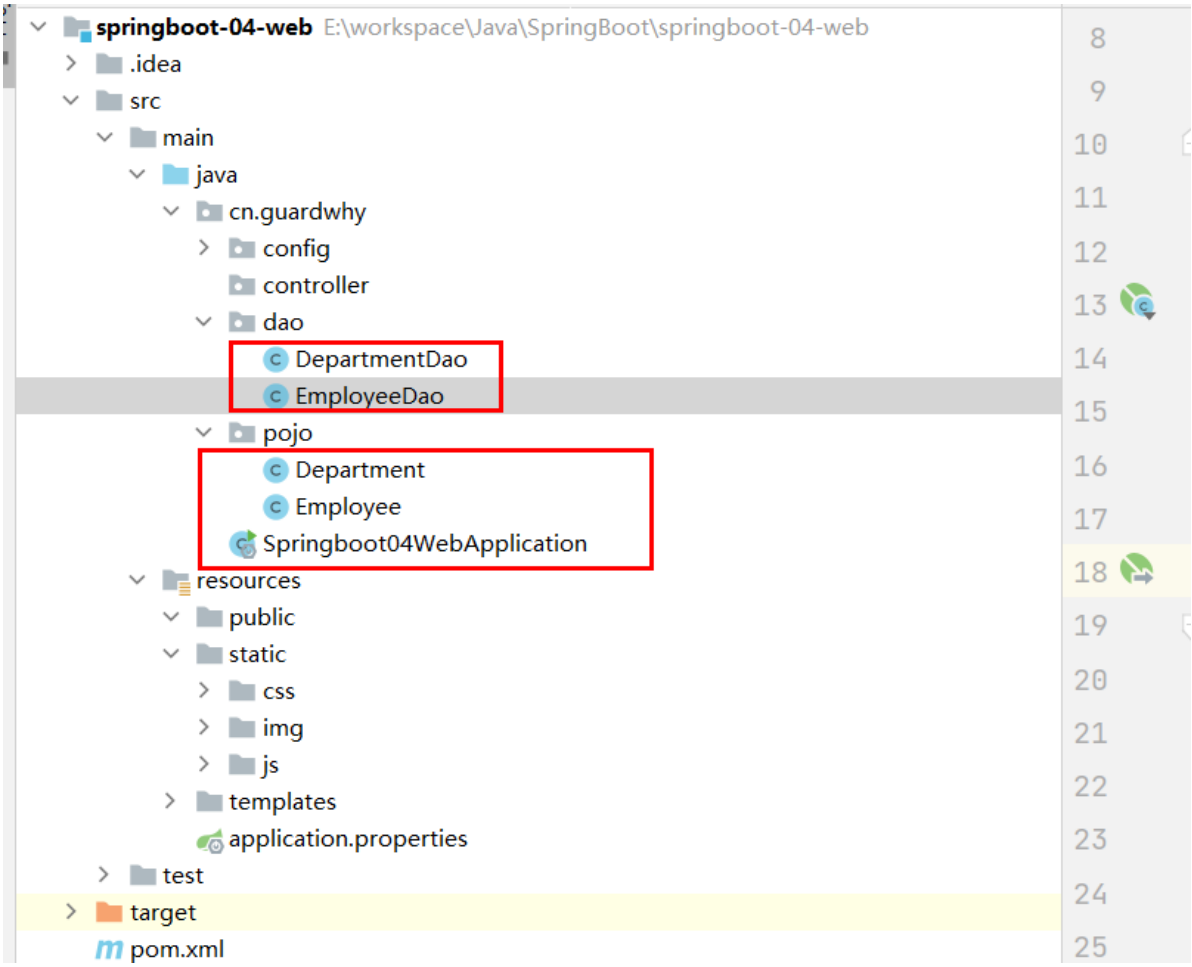
在Springboot中，有非常多的(???)Configuration帮助进行拓展配置，只要是看到了这个文件，需要注意！！！

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/1FBdaNfwlZtks7QlIbovhCw 提取码：h33l

项目目录



7.1.1 导入相关依赖

```xml
<!-- lombok -->
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
</dependency>
```

```xml
<!--thymeleaf版本-->
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring5</artifactId>
</dependency>
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-java8time</artifactId>
</dependency>
```

7.1.2 实体类实现

```java
package cn.guardwhy.pojo;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

// 部门表
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Department {
    private Integer id;
    private String departmentName;
}
```

```java
package cn.guardwhy.pojo;

import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;

// 员工表
@Data
@NoArgsConstructor
public class Employee {
    private Integer id;
    private String lastName;
    private String email;
    private Integer gender; //0: 女 1: 男
    private Department department;
    private Date birth;

    public Employee(Integer id, String lastName, String email, Integer
gender, Department department) {
        this.id = id;
        this.lastName = lastName;
        this.email = email;
        this.gender = gender;
        this.department = department;
        // 默认创建日期
        this.birth = new Date();
    }
}
```

7.1.3 持久层实现

DepartmentDao

```java
package cn.guardwhy.dao;

import cn.guardwhy.pojo.Department;
import org.springframework.stereotype.Repository;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

// 部门的dao
@Repository
public class DepartmentDao {
    // 模拟数据库中的数据
    private static Map<Integer, Department> departments = null;
    static {
        departments = new HashMap<Integer, Department>(); // 创建部门表
        departments.put(101, new Department(101, "教学部"));
        departments.put(102, new Department(102, "市场部"));
        departments.put(103, new Department(103, "教研部"));
        departments.put(104, new Department(104, "运营部"));
        departments.put(105, new Department(105, "后勤部"));
    }

    // 2.获取所有部门信息
    public Collection<Department> getDepartments(){
        return departments.values();
    }
    // 3.通过id得到部门
    public Department getDepartmentById(Integer id){
        return departments.get(id);
    }
}
```

EmployeeDao

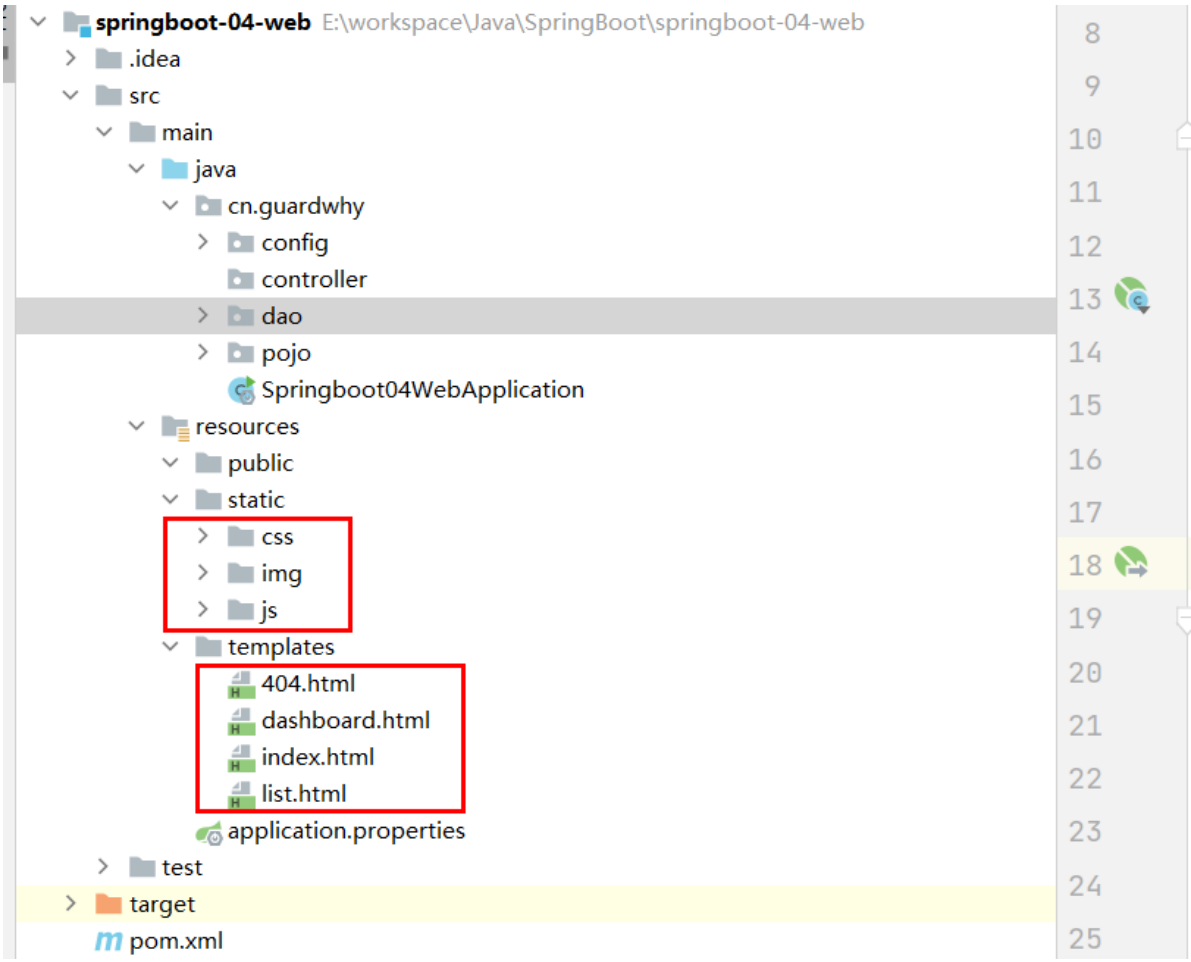
```java
package cn.guardwhy.dao;

import cn.guardwhy.pojo.Department;
import cn.guardwhy.pojo.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

@Repository
public class EmployeeDao {
    // 1.模拟数据库中的数据
    private static Map<Integer, Employee> employees = null;
    // 2.员工有所属的部门
    @Autowired
    private DepartmentDao departmentDao;
```

```
    static {
        // 3.创建一个员工表
        employees = new HashMap<Integer, Employee>();
        employees.put(1001, new Employee(1001, "kobe","1234678@qq.com", 1, new Department(101,"后勤部")));
        employees.put(1002, new Employee(1002, "james","1234678@qq.com", 1, new Department(102,"市场部")));
        employees.put(1003, new Employee(1003, "xiao hong","1234678@qq.com", 0, new Department(103,"教学部")));
        employees.put(1004, new Employee(1004, "harden","1234678@qq.com", 1, new Department(104,"运营部")));
        employees.put(1005, new Employee(1005, "guardwhy","1234678@qq.com", 1, new Department(105,"教研部")));
    }
    // 3.主键自增
    private static Integer initId = 1006;
    // 4.增加一个员工
    public void add(Employee employee){
        // 条件判断
        if(employee.getId() == null){
            employee.setId(initId++);
        }

 employee.setDepartment(departmentDao.getDepartmentById(employee.getDepartment().getId()));
        employees.put(employee.getId(), employee);
    }
    // 5.查询全部员工的信息
    public Collection<Employee> getAll(){
        return employees.values();
    }
    // 6.通过id查询员工
    public Employee getEmployeeById(Integer id){
        return employees.get(id);
    }
    // 7.通过id删除员工
    public void delete(Integer id){
        employees.remove(id);
    }
}
```

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

```java
package cn.guardwhy.config;

import org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
@Configuration
public class MyMvcConfig implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
        registry.addViewController("/index.html").setViewName("index");
    }
}
```

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

```properties
# 关闭模板引擎的缓存
spring.thymeleaf.cache=false

# 改变路径
server.servlet.context-path=/user
```

index.html

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">

<!-- Bootstrap core CSS -->
<link th:href="@{/css/bootstrap.min.css}" rel="stylesheet">
<!-- Custom styles for this template -->
<link th:href="@{/css/signin.css}" rel="stylesheet">

<img class="mb-4" th:src="@{/img/bootstrap-solid.svg}" alt="" width="72"
height="72">
```

404.html

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">

<!-- Bootstrap core CSS -->
<link th:href="@{/css/bootstrap.min.css}" rel="stylesheet">
<!-- Custom styles for this template -->
<link th:href="@{/css/dashboard.css}" rel="stylesheet">
```

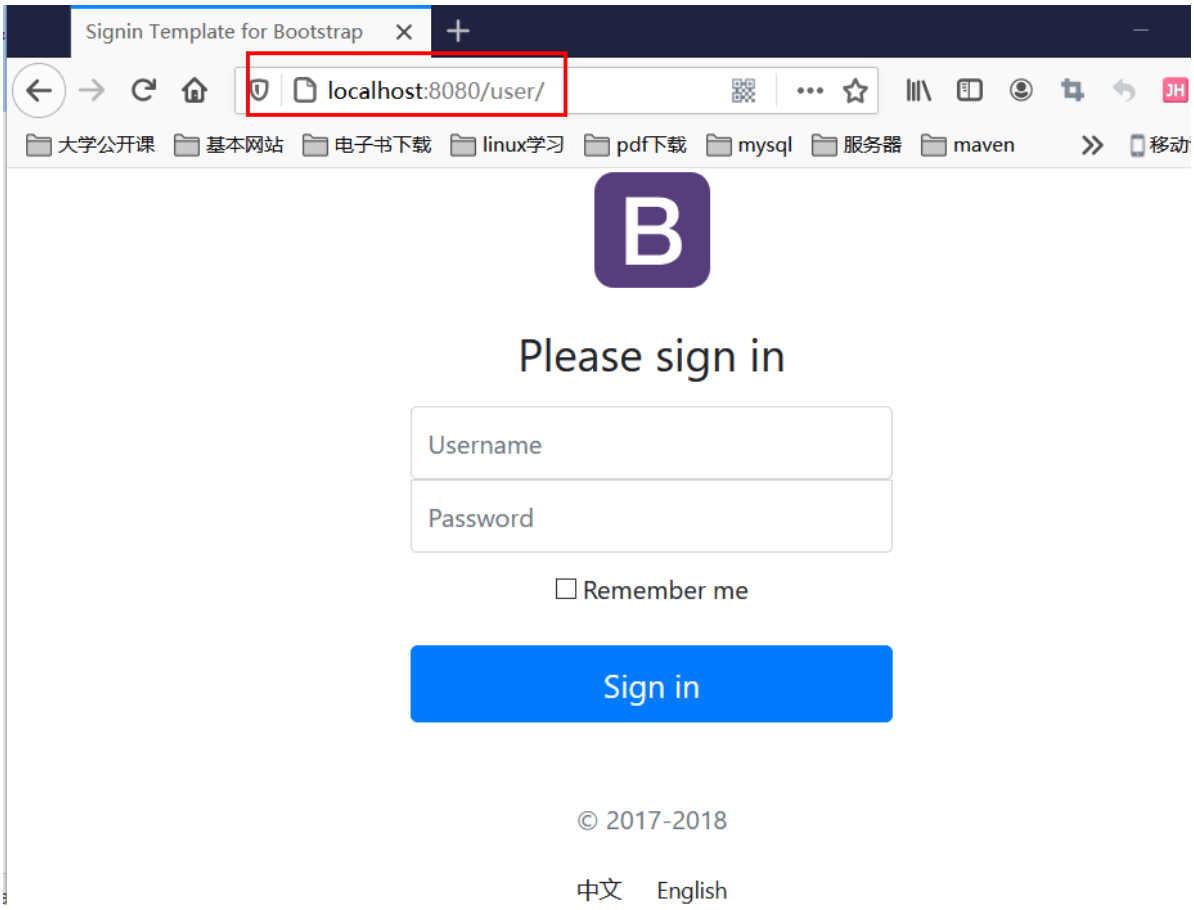
dashboard.html

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<!-- Bootstrap core CSS -->
<link th:href="@{/css/bootstrap.min.css}" rel="stylesheet">

<!-- Custom styles for this template -->
<link th:href="@{/css/dashboard.css}" rel="stylesheet">
```

list.html

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">

<!-- Bootstrap core CSS -->
<link th:href="@{/css/bootstrap.min.css}" rel="stylesheet">

<!-- Custom styles for this template -->
<link th:href="@{/css/dashboard.css}" rel="stylesheet">
```

7.2.3 执行结果

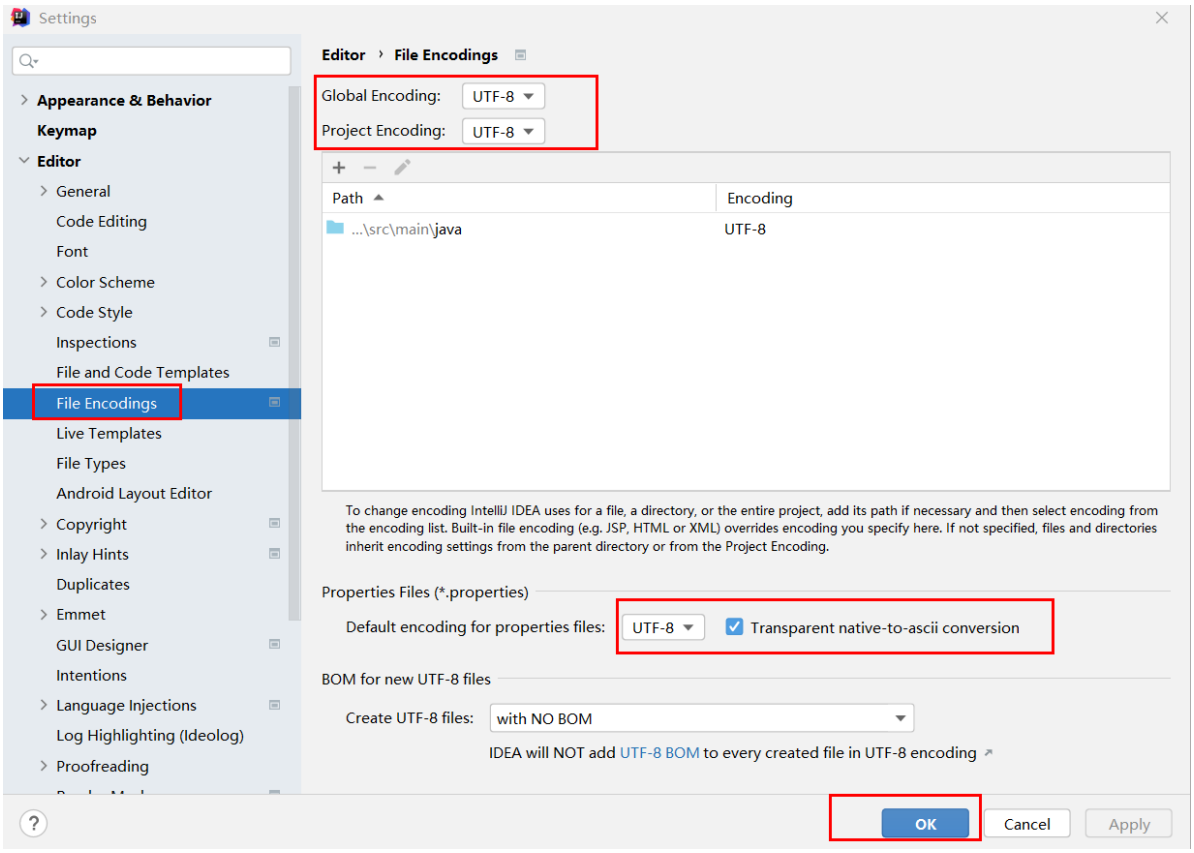


7.3 页面国际化

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

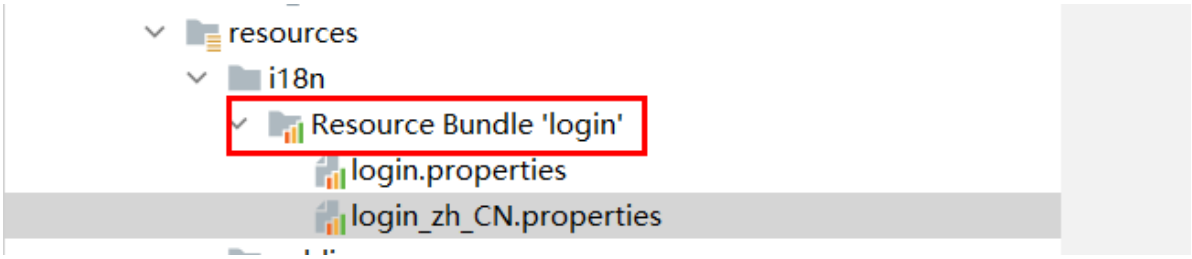
先在IDEA中统一设置`properties`的编码。



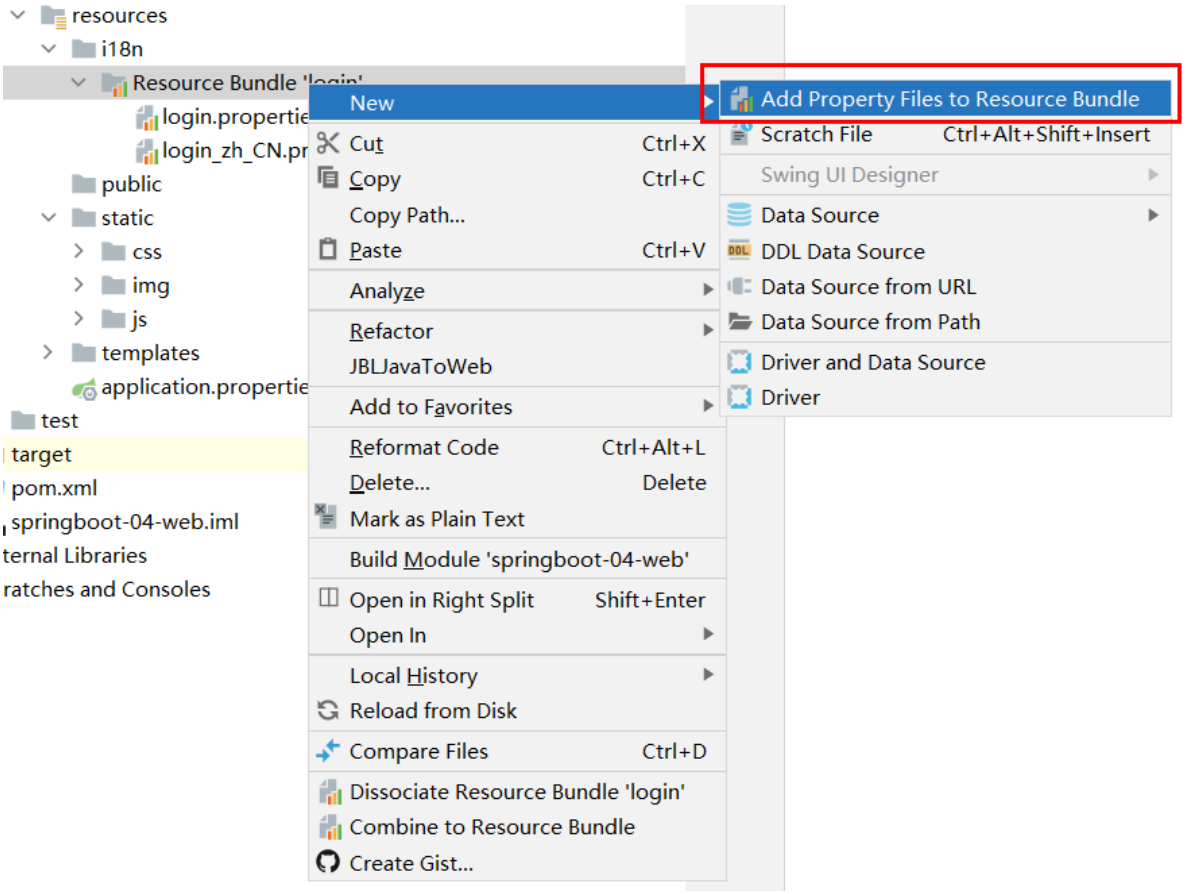
7.3.2 文件创建

1、在resources资源文件下新建一个i18n目录，存放国际化配置文件。

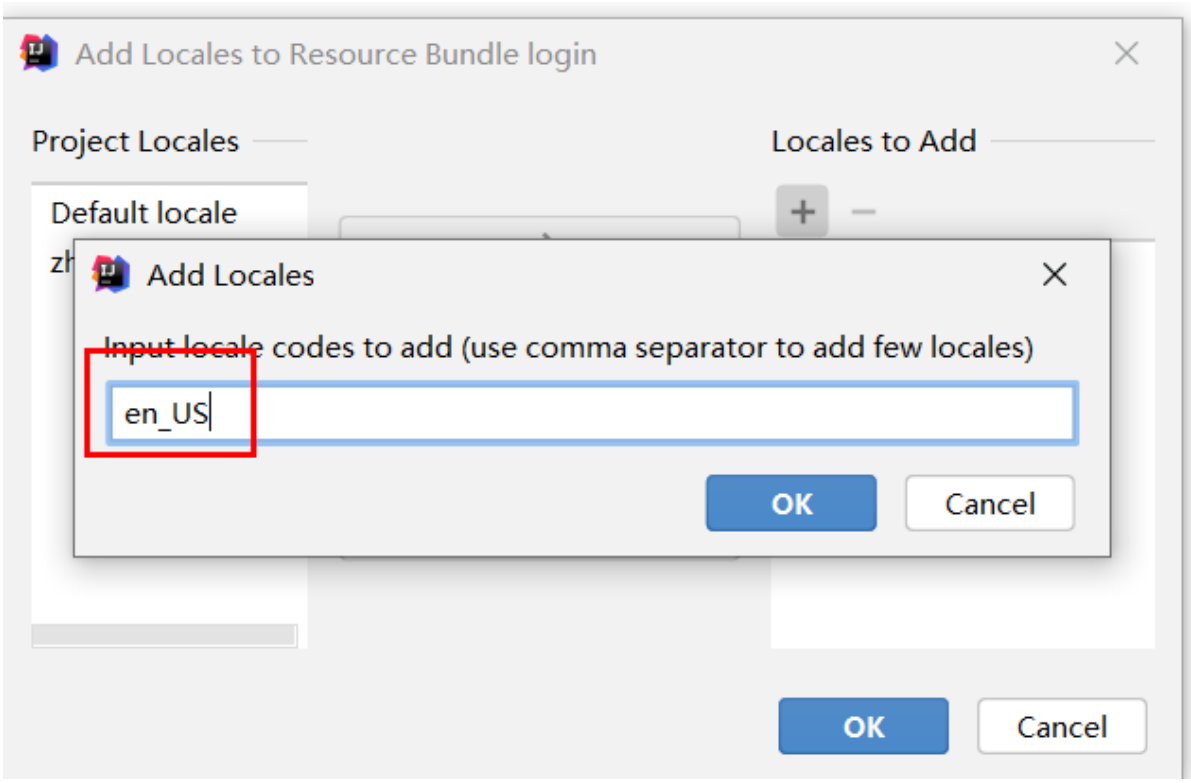
2、建立一个login.properties文件，还有一个login_zh_CN.properties。发现IDEA自动识别了要做国际化操作，文件夹变了。



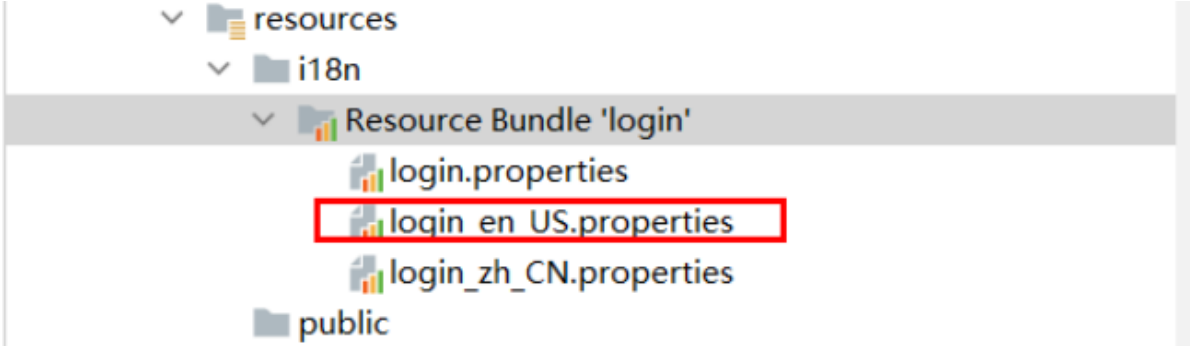
3、可以在这上面去新建一个文件



4、弹出如下页面：再添加一个英文的文件。



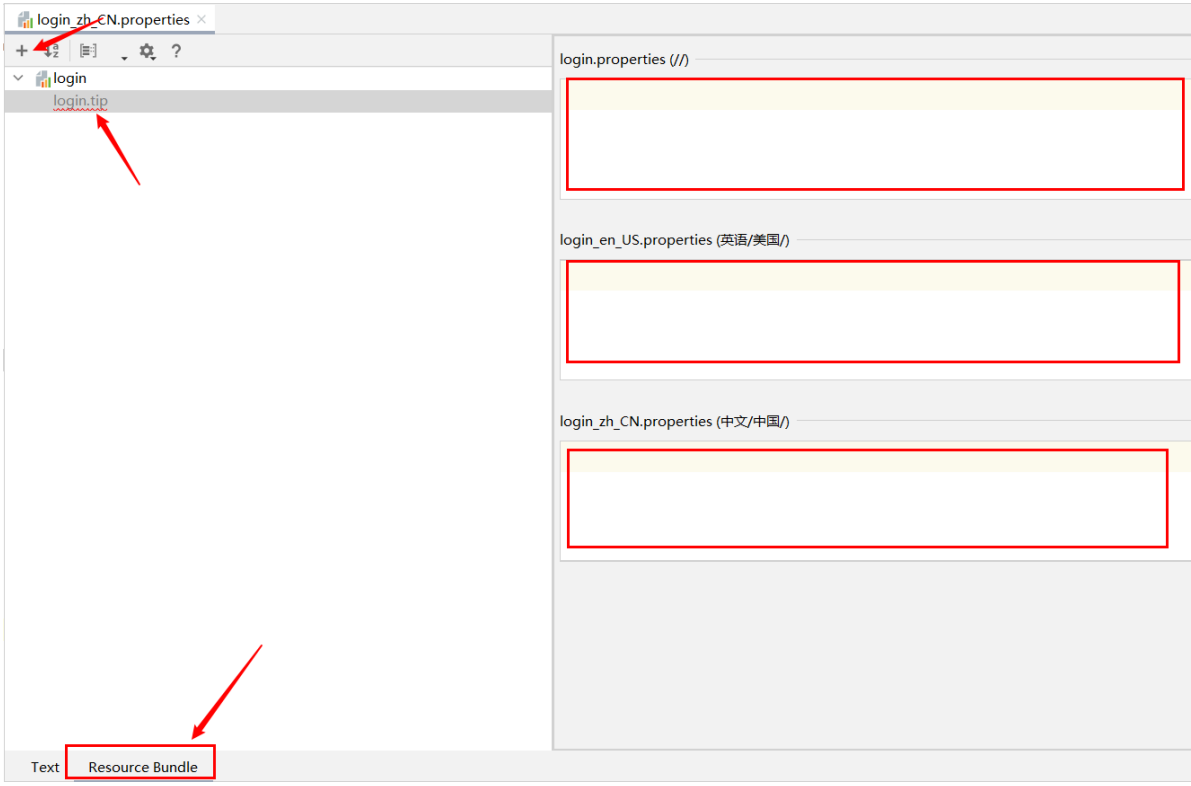
结果如下



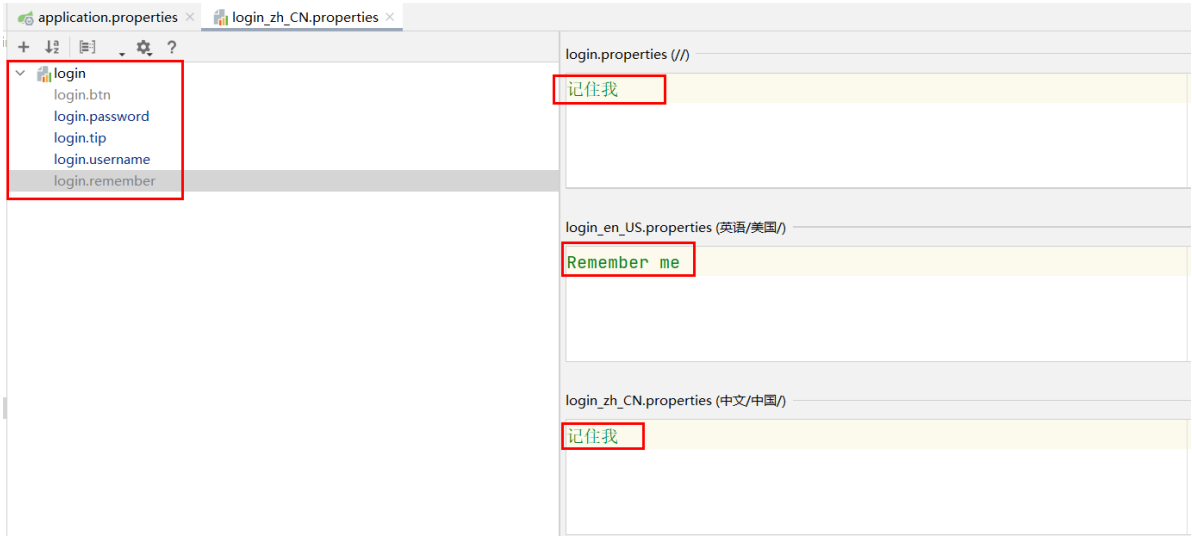
7.3.4 编写配置

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可以看到边上有三个文件框可以输入。



依次添加页面内容



然后去查看配置文件

login.properties: 默认

```
login.btn=登录
login.password=密码
login.remember=记住我
login.tip=请登录
login.username=用户名
```

英文:

```
login.btn=Sign in
login.password=Password
login.remember=Remember me
login.tip=Please sign in
login.username=Username
```

中文:

```
login.btn=登录
login.password=密码
login.remember=记住我
login.tip=请登录
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！！！

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingBean(name =
AbstractApplicationContext.MESSAGE_SOURCE_BEAN_NAME, search =
SearchStrategy.CURRENT)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Conditional(ResourceBundleCondition.class)
@EnableConfigurationProperties
public class MessageSourceAutoConfiguration {

    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.setBasenames(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.isUseCodeAsDefaultMessa
    ge());
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

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

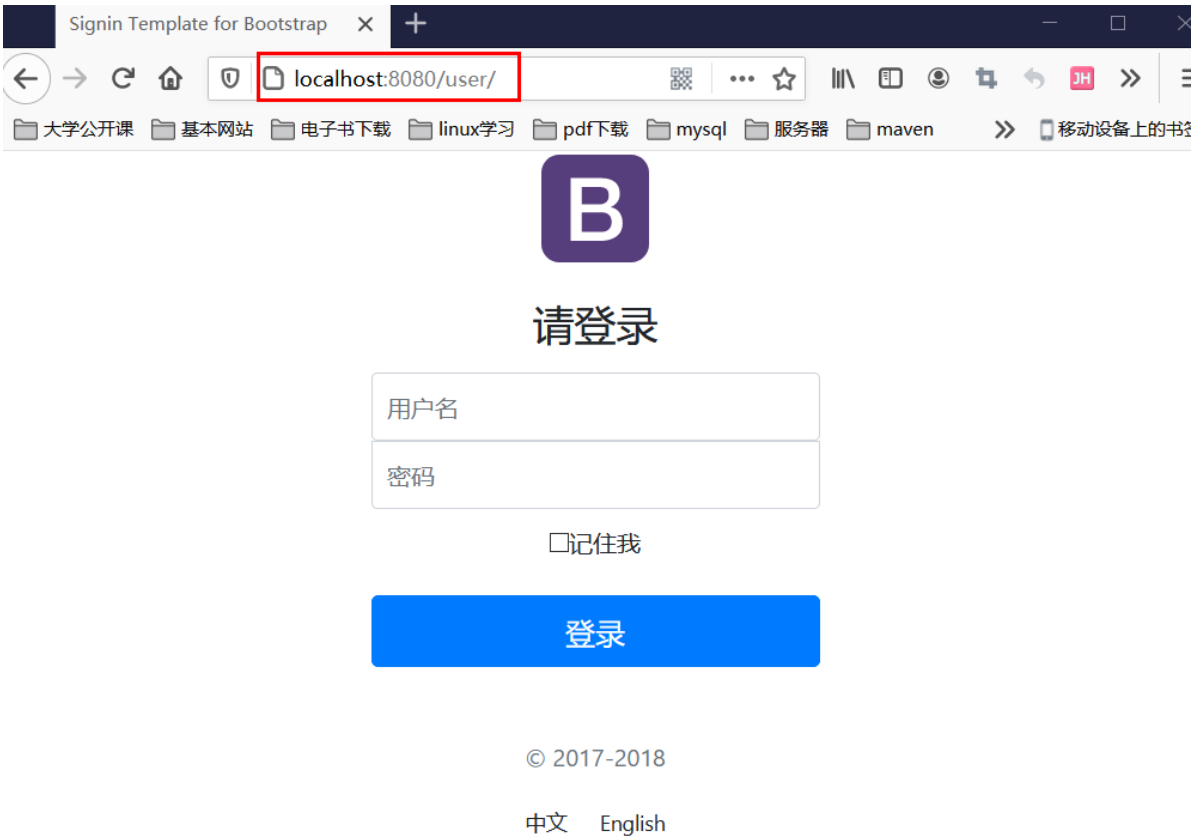
7.3.6 页面国际化值

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

```
<body class="text-center">
    <form class="form-signin" action="dashboard.html">
        <img class="mb-4" th:src="@{/img/bootstrap-solid.svg}" alt="" width="72" height="72">
        <h1 class="h3 mb-3 font-weight-normal" th:text="#{login.tip}">Please sign in</h1>
        <input type="text" class="form-control" th:placeholder="#{login.username}" required="" autofocus="">
        <input type="password" class="form-control" th:placeholder="#{login.password}" required="">
        <div class="checkbox mb-3">
            <label>
                <input type="checkbox" value="remember-me" th:text="#{login.remember}">
            </label>
        </div>
        <button class="btn btn-lg btn-primary btn-block" type="submit" th:text="#{login.btn}">Sign in</button>
        <p class="mt-5 mb-3 text-muted">© 2017-2018</p>
        <a class="btn btn-sm">中文</a>
        <a class="btn btn-sm">English</a>
    </form>

</body>
```

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。

```
@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 {

}
```

2、在WebMvcAutoConfiguration搜索LocaleResolver。

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

```
        return new FixedLocaleResolver(this.webProperties.getLocale());
    }
    if (this.mvcProperties.getLocaleResolver() ==
WebMvcProperties.LocaleResolver.FIXED) {
        return new FixedLocaleResolver(this.mvcProperties.getLocale());
    }
    // 接收头国际化分解
    AcceptHeaderLocaleResolver localeResolver = new
AcceptHeaderLocaleResolver();
    Locale locale = (this.webProperties.getLocale() != null) ?
this.webProperties.getLocale()
        : this.mvcProperties.getLocale();
    localeResolver.setDefaultLocale(locale);
    return localeResolver;
}
```

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

```
@Override
public Locale resolveLocale(HttpServletRequest request) {
    Locale defaultLocale = getDefaultLocale();
    // 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
    if (defaultLocale != null && request.getHeader("Accept-Language") ==
null) {
        return defaultLocale;
    }
    Locale requestLocale = request.getLocale();
    List<Locale> supportedLocales = getSupportedLocales();
    if (supportedLocales.isEmpty() ||
supportedLocales.contains(requestLocale)) {
        return requestLocale;
    }
    Locale supportedLocale = findSupportedLocale(request, supportedLocales);
    if (supportedLocale != null) {
        return supportedLocale;
    }
    return (defaultLocale != null ? defaultLocale : requestLocale);
}
```

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。

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

```
        // 如果请求链接不为空
        if(!StringUtils.isEmpty(language)){
            // 分割请求参数
            String[] split = language.split("_");
            // 国家，地区
            locale = new Locale(split[0],split[1]);
        }
        return locale;
    }

    @Override
    public void setLocale(HttpServletRequest request, HttpServletResponse
response, Locale locale) {

    }
}
```

将自己写的组件配置到Spring容器中

```
package cn.guardwhy.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.LocaleResolver;
import
org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
@Configuration
public class MyMvcConfig implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
        registry.addViewController("/index.html").setViewName("index");
    }
    @Bean
    public LocaleResolver localeResolver(){
        return new MyLocaleResolver();
    }
}
```

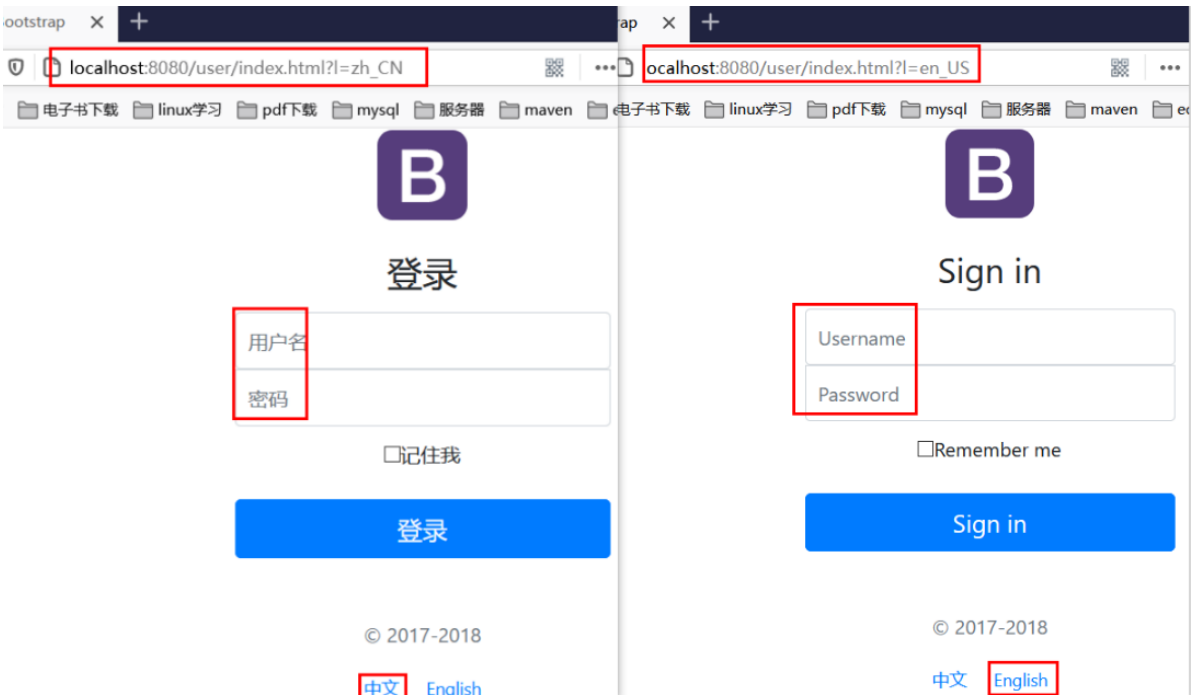
5、修改前端页面的跳转连接

点击链接让国际化资源生效，就需要让Locale生效。

index.html

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

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。

```
# 关闭模板引擎的缓存
spring.thymeleaf.cache=false
# 改变路径
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

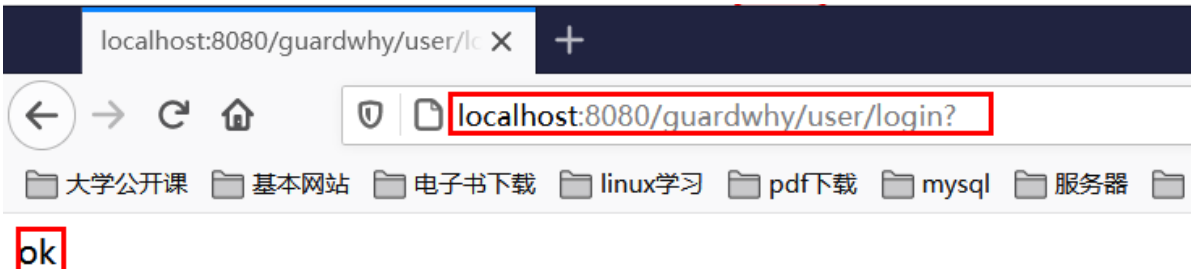
```
<form class="form-signin" th:action="@{/user/login}">
    <!--如果message的值为空，则不显示消息-->
    <h3 style="color: red" th:text="${message}" th:if="${not
#strings.isEmpty(message)}"></h3>
   <input type="text" name="username" class="form-control" th:placeholder="#
{login.username}" required="" autofocus="">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

2、测试controller

```
package cn.guardwhy.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class LoginController {
    @RequestMapping("/user/login")
```

```
    @ResponseBody
    public String login(){
        return "ok";
    }
}
```

测试结果



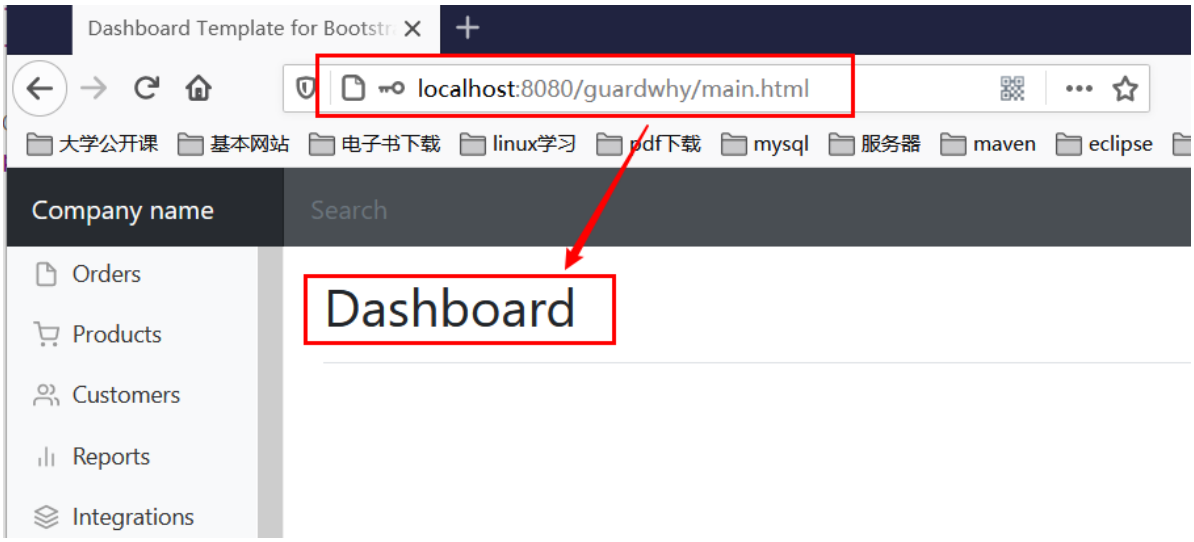
3、编写对应的controller

```
package cn.guardwhy.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.thymeleaf.util.StringUtils;

@Controller
public class LoginController {
    @RequestMapping("/user/login")
    public String login(@RequestParam("username") String username,
                        @RequestParam("password") String password,
                        Model model
                        ){
        // 具体业务
        if(!StringUtils.isEmpty(username) && "123666".equals(password)){
            // 登录成功！防止表单重复提交，我们重定向
            return "redirect:/main.html";
        }else{
            // 告诉用户，登录失败了
            model.addAttribute("msg","用户名或者密码错误!");
            return "index";
        }
    }
}
```

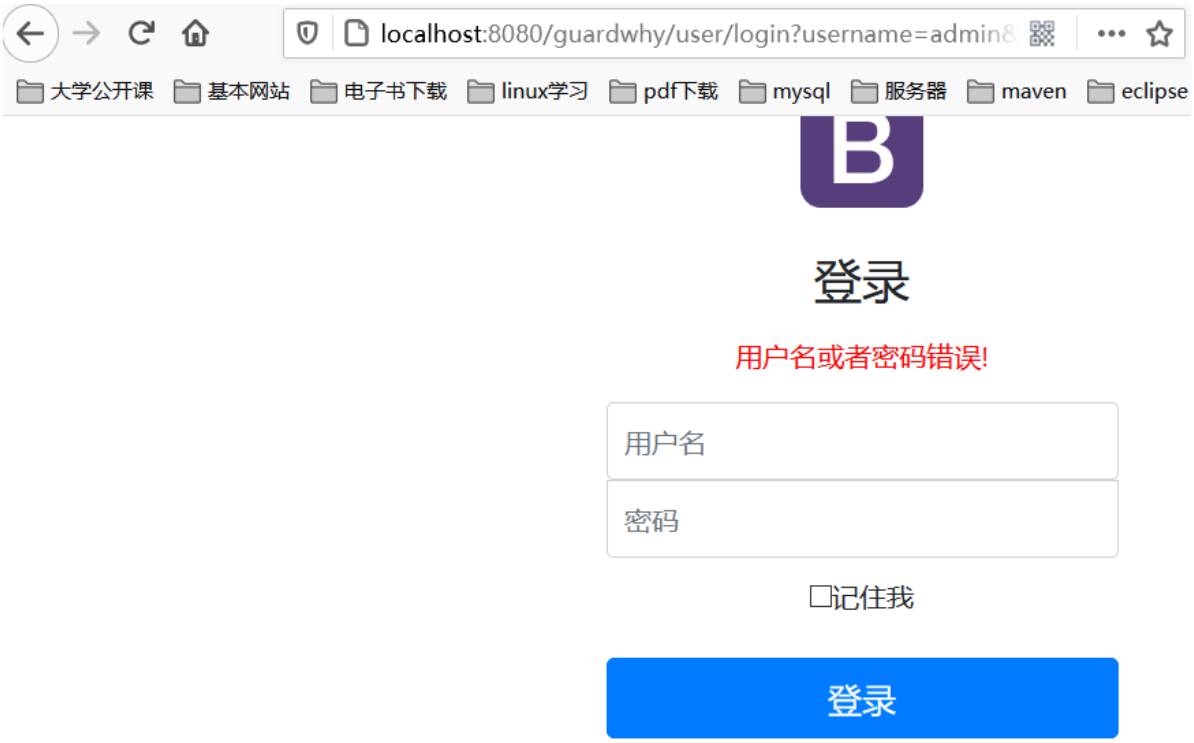
4、在自己的MyMvcConfig，添加一个视图控制映射。

```
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

```
package cn.guardwhy.config;

import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginHandlerInterceptor implements HandlerInterceptor {
```

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

2、将拦截器注册到SpringMVC配置类当中

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

3、获取用户登录的信息

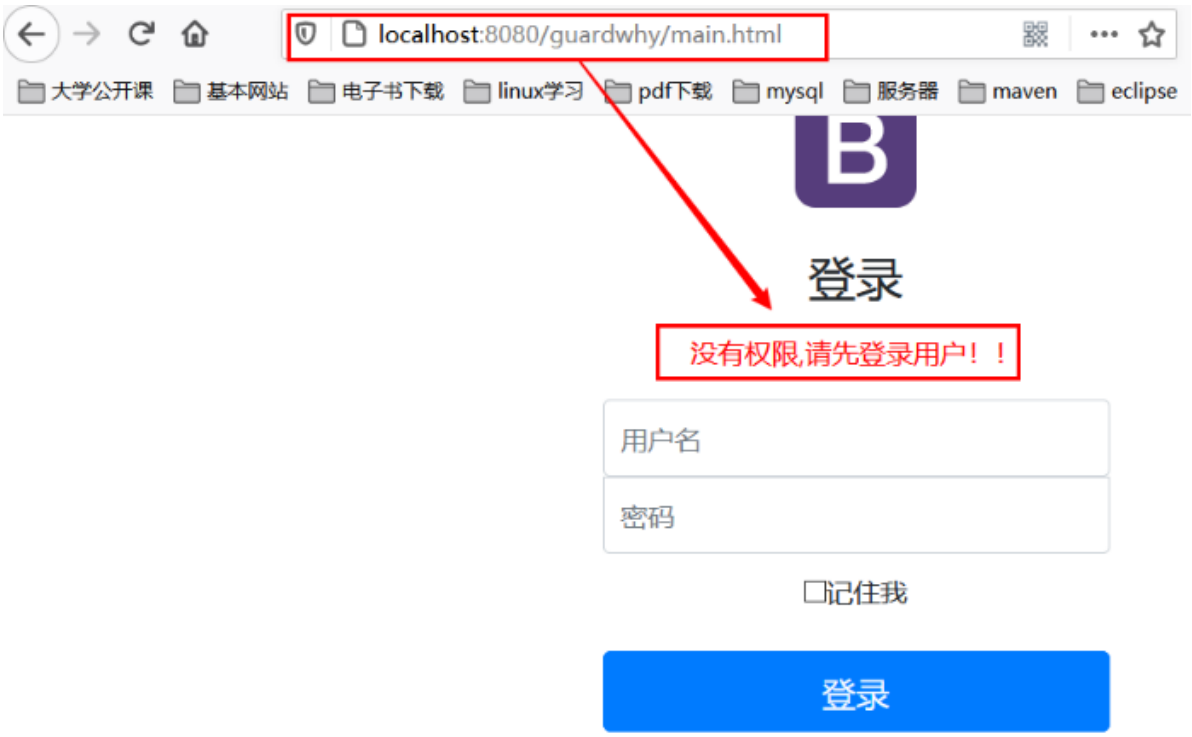
dashboard.html

dashboard.html

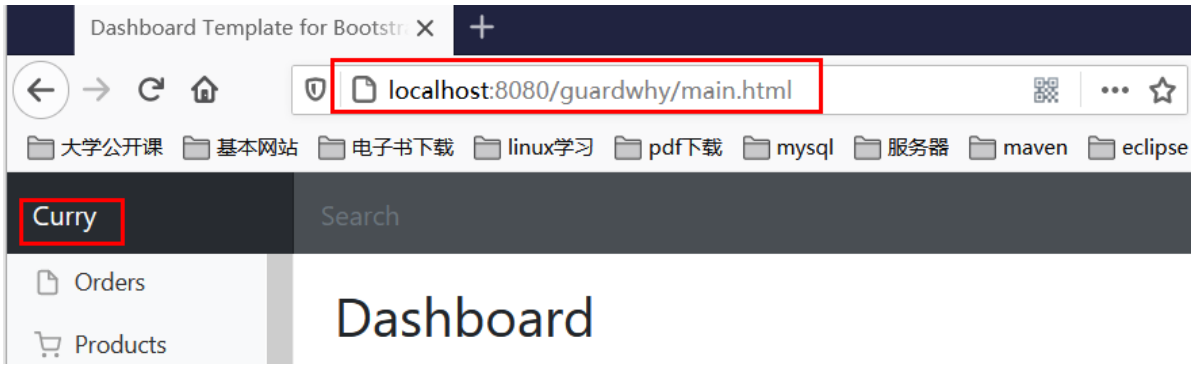
```
<body>
    <nav class="navbar navbar-dark sticky-top bg-dark flex-md-nowrap p-0">
        <a class="navbar-brand col-sm-3 col-md-2 mr-0" href="http://getbootstrap.com/docs/4.0/examples/dashboard/#">
            [[${session.loginUser}]]
        </a>
        <input class="form-control form-control-dark w-100" type="text" placeholder="Search" aria-label="Search">
        <ul class="navbar-nav px-3">
            <li class="nav-item text-nowrap">
                <a class="nav-link" href="http://getbootstrap.com/docs/4.0/examples/dashboard/#">Sign out</a>
            </li>
        </ul>
    </nav>
```

登录测试拦截

测试拦截成功！！！



获取用户登录的信息

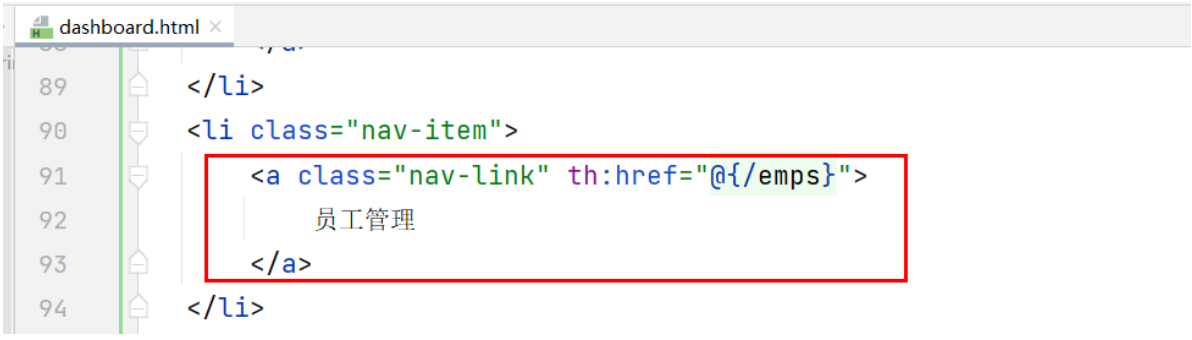


7.5 员工列表实现

7.5.1 员工列表页面跳转

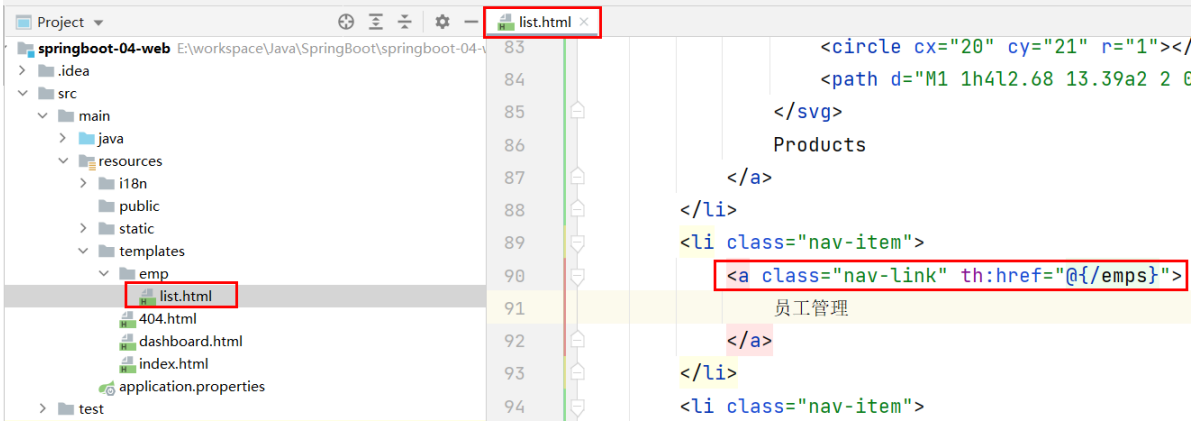
在主页点击Customers，就显示列表页面，将首页的侧边栏Customers改为员工管理，a链接添加请求。

dashboard.html



list.html

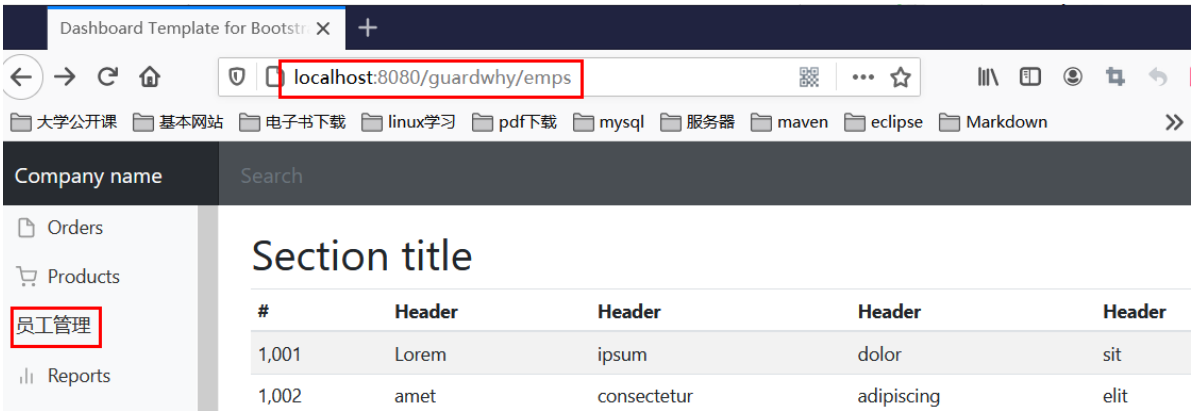
将`list.html`放在emp文件夹下。



处理请求controller

```java
package cn.guardwhy.controller;

import cn.guardwhy.dao.EmployeeDao;
import cn.guardwhy.pojo.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.Collection;

@Controller
public class EmployeeController {
    @Autowired
    EmployeeDao employeeDao;

    @RequestMapping("/emps")
    public String list(Model model){
    // 获得所有员工
        Collection<Employee> employees = employeeDao.getAll();
        model.addAttribute("emps", employees);
        return "emp/list";
    }
}
```

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

```
<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

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

```
list.html
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"...>
191             </div>
192         </div>
```

顶部导航抽取

dashboard.html

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

```
dashboard.html
45          <!--顶部导航栏-->
46          <nav class="navbar navbar-dark sticky-top bg-dark flex-md-nowrap 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/#
```

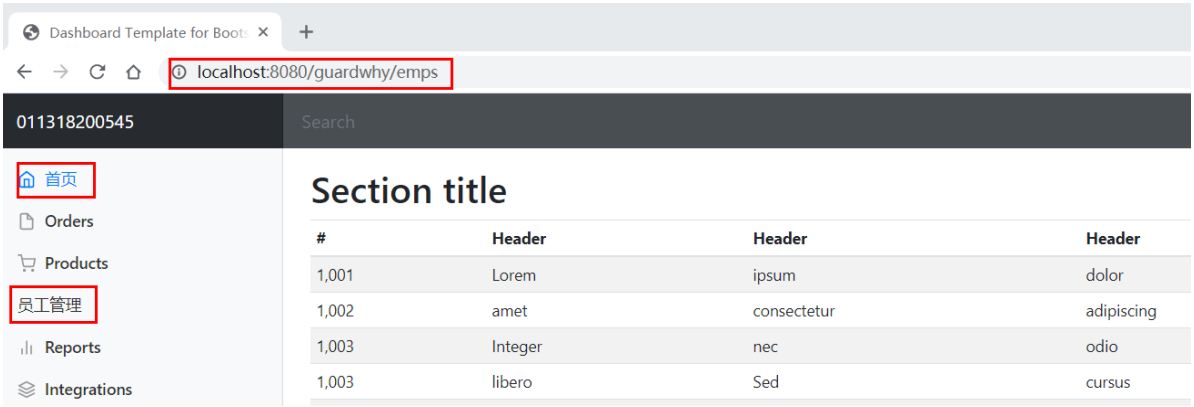
list.html

```
<div th:insert="~{dashboard::topbar}"></div>
<div class="container-fluid">
    <!--侧边栏-->
    <div class="row">
        <div th:insert="~{dashboard::sidebar}"></div>
    </div>
</div>
```

list.html

```
<!--顶部栏-->
<div th:insert="~{dashboard::topbar}"></div>
<div class="container-fluid">
    <!--侧边栏-->
    <div class="row">
        <div th:insert="~{dashboard::sidebar}"></div>

        <main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4"...>
    </div>
</div>
```

启动项目，看效果！！



建立一个commons文件夹，专门存放公共页面

commons.html

dashboard.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>
```

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

list.html

list.html

```
<body>
    <!--顶部栏-->
    <div th:replace="~{commons/commons::topbar}"></div>
    <div class="container-fluid">
        <!--侧边栏-->
        <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>
```

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

dashboard.html

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

    </div>
```

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

list.html

list.html

```
    <!--顶部栏-->
    <div th:replace="~{commons/commons::topbar}"></div>
    <div class="container-fluid">
        <!--侧边栏-->
        <div class="row">
            <div th:replace="~{commons/commons::sidebar(active='list.html')}"></div>
            <main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4"...>
        </div>
    </div>
```

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

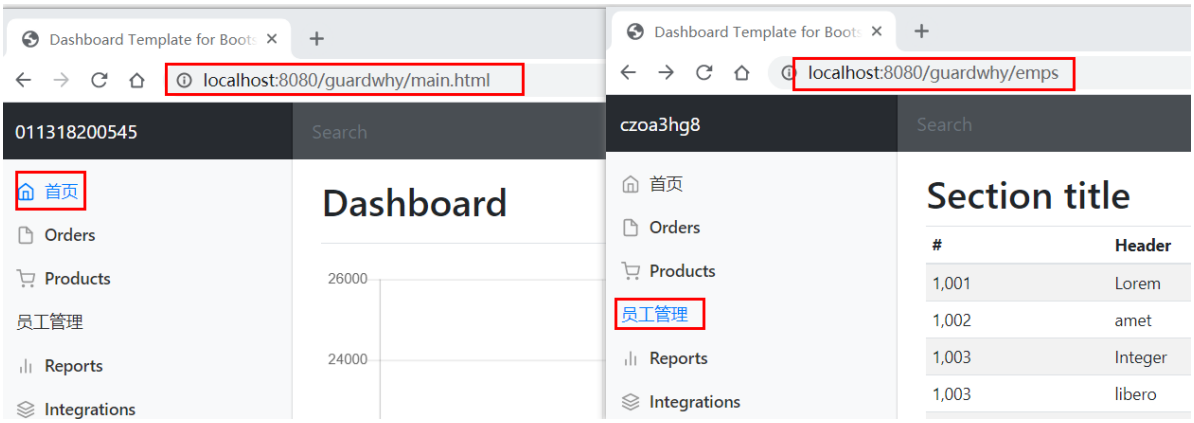
commons.html

commons.html

```
<!--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">
        <a th:class="${active=='main.html'? 'nav-link active':'nav-link'}" th:href="@{/index.html}">
          <svg xmlns="http://www.w3.org/2000/svg" width="24" height="24" viewBox="0 0 24 24" fill="nc
          首页 <span class="sr-only">(current)</span>
        </a>
      </li>
      <li class="nav-item"...>
      <li class="nav-item"...>
      <li class="nav-item">
        <a th:class="${active=='list.html'?'nav-link active':'nav-link'}" th:href="@{/emps}">
```

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

刷新页面，测试一下结果



员工信息页面展示

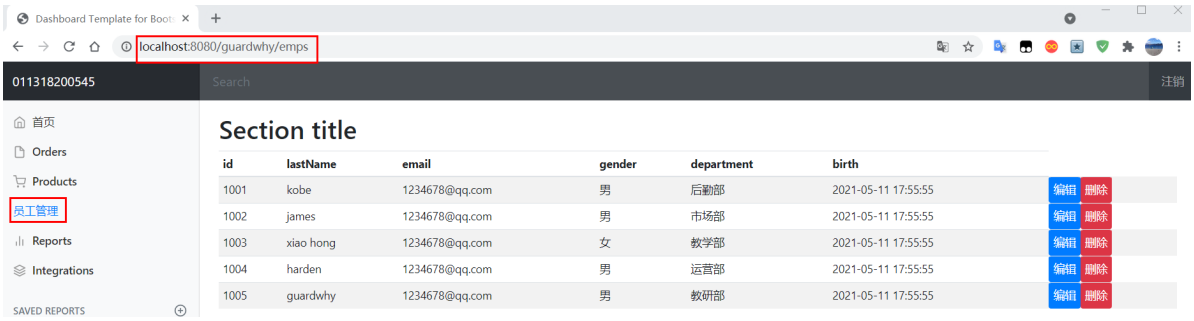
遍历员工信息

list.html

```
<main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-4">
    <h2>Section title</h2>
    <div class="table-responsive">
        <table class="table table-striped table-sm">
            <thead>
                <tr>
                    <th>id</th>
                    <th>lastName</th>
                    <th>email</th>
                    <th>gender</th>
                    <th>department</th>
                    <th>birth</th>
                </tr>
            </thead>
            <tbody>
                <tr th:each="emp:${emps}">
                    <td th:text="${emp.getId()}"></td>
                    <td th:text="${emp.getLastName()}"></td>
                    <td th:text="${emp.getEmail()}"></td>
```

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

执行结果



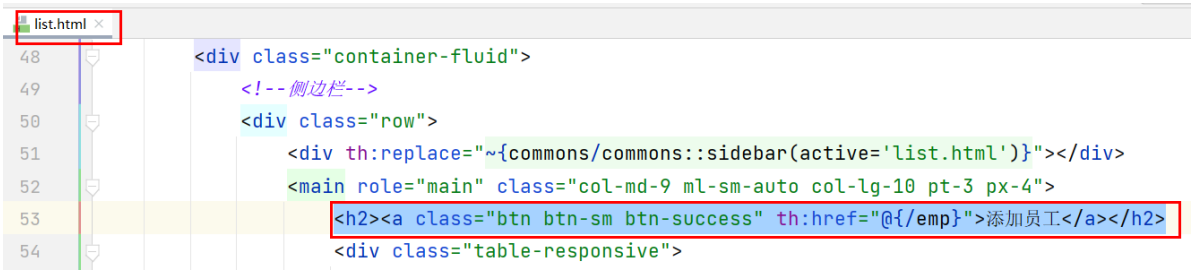
7.6 添加员工实现

7.6.1 表单优化

1、将添加员工信息改为超链接

list.html

```
<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！！！

```
<form>
    <div class="form-group">
        <label>LastName</label>
        <input type="text" name="lastName" class="form-control"
placeholder="guardwhy">
    </div>
    <div class="form-group">
        <label>Email</label>
        <input type="email" name="email" class="form-control"
placeholder="1625309592@qq.com">
    </div>
    <div class="form-group">
```

```
        <label>Gender</label><br/>
        <div class="form-check form-check-inline">
            <input class="form-check-input" type="radio" name="gender"
value="1">
            <label class="form-check-label">男</label>
        </div>
        <div class="form-check form-check-inline">
            <input class="form-check-input" type="radio" name="gender"
value="0">
            <label class="form-check-label">女</label>
        </div>
    </div>
    <div class="form-group">
        <label>department</label>
        <select class="form-control" name="department">
            <option>1</option>
            <option>2</option>
            <option>3</option>
            <option>4</option>
            <option>5</option>
        </select>
    </div>
    <div class="form-group">
        <label>Birth</label>
        <input type="text" name="birth" class="form-control"
placeholder="kuangstudy">
    </div>
    <button type="submit" class="btn btn-primary">添加</button>
</form>
```

7.6.2 添加员工

员工添加页面

1、EmployeeController

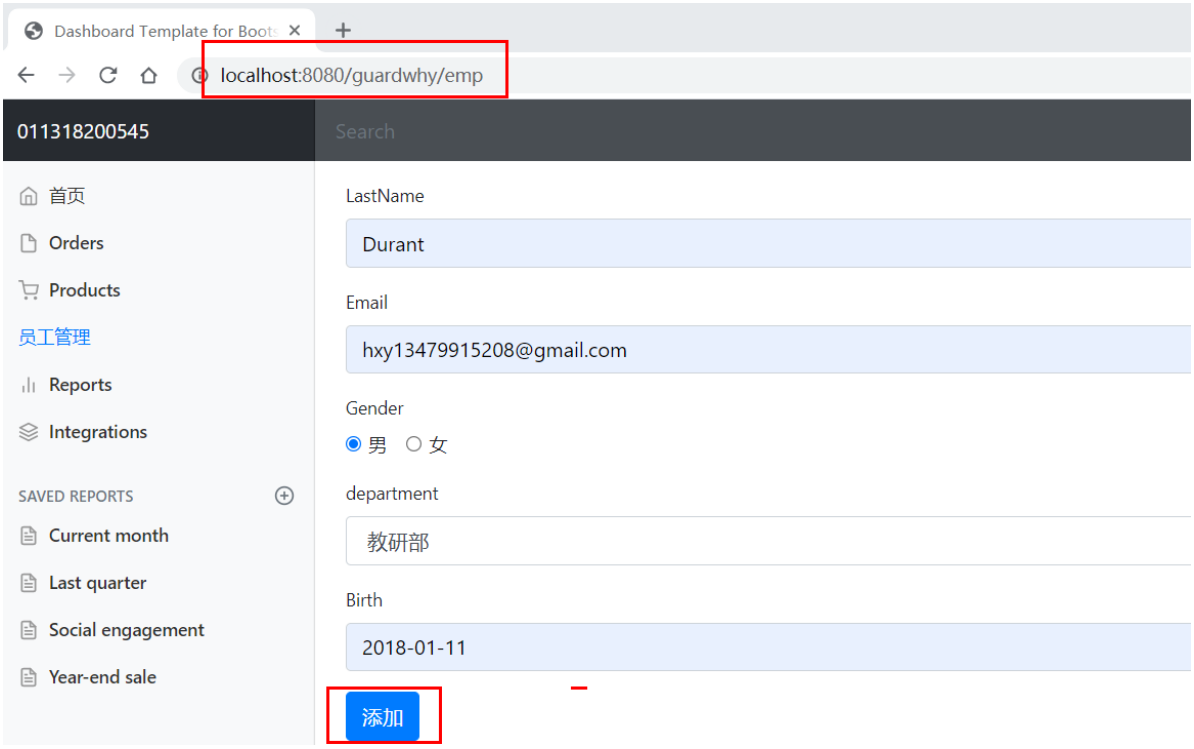
```
@GetMapping("/emp")
public String toAddpage(Model model){
    // 查询所有部门的信息
    Collection<Department> departments = departmentDao.getDepartments();
    model.addAttribute("departments", departments);
    return "emp/add";
}
```

2、前端页面修改

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

```
add.html
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()}"></option>
77            </select>
78        </div>
```

执行结果



员工添加功能

3、EmployeeController

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

4、修改aform表单提交方式

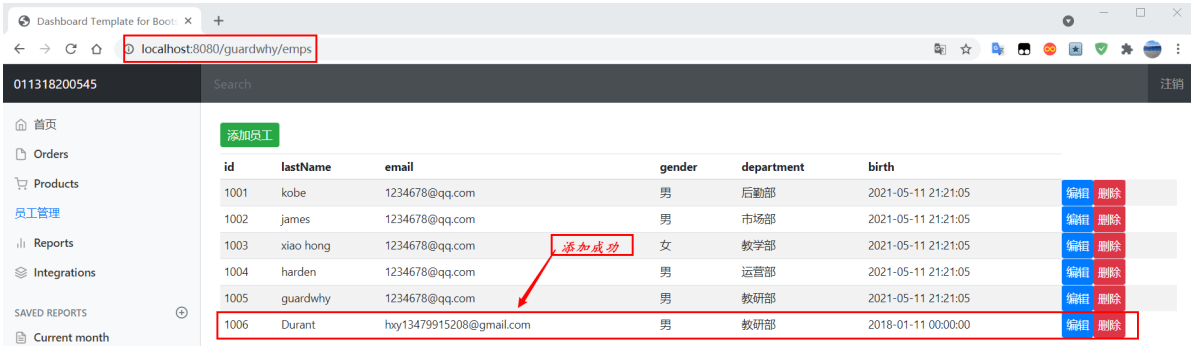
```
<form th:action="@{/emp}" method="post"></form>
```

自定义日期格式化

5、application.properties

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

执行结果



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

```
<td>
    <a class="btn btn-sm btn-primary" th:href="@{/emp/}+${emp.id}">编辑</a>
</td>
```

2、去员工修改页面controller

EmployeeController

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

7.7.2 查询数据回显

update.html

```
<!--提交方式-->
<form th:action="@{/updateEmp}" method="post">
    <div class="form-group">
        <label>LastName</label>
        <input th:value="${emp.getLastName()}" type="text" name="lastName"
class="form-control" placeholder="guardwhy">
    </div>
    <div class="form-group">
        <label>Email</label>
        <input th:value="${emp.getEmail()}" type="email" name="email"
class="form-control" placeholder="1625309592@qq.com">
    </div>
    <div class="form-group">
```

```
        <label>Gender</label><br/>
        <div class="form-check form-check-inline">
            <input th:checked="${emp.getGender()==1}"  class="form-check-input" type="radio" name="gender" value="1">
            <label class="form-check-label">男</label>
        </div>
        <div class="form-check form-check-inline">
            <input th:checked="${emp.getGender()==0}" class="form-check-input" type="radio" name="gender" value="0">
            <label class="form-check-label">女</label>
        </div>
    </div>
    <div class="form-group">
        <label>department</label>
        <!--控制器Controller接收的是Employee(员工)，所以需要提交其中的一个属性！！-->
        <select class="form-control" name="department.id">
            <option th:selected="${dept.getId()==emp.getDepartment().getId()}" th:each="dept:${departments}" th:text="${dept.getDepartmentName()}" th:value="${dept.getId()}"></option>
        </select>
    </div>
    <div class="form-group">
        <label>Birth</label>
        <!--日期格式化-->
        <input th:value="${#dates.format(emp.getBirth(),'yyyy-MM-dd HH:mm')}" type="text" name="birth" class="form-control" placeholder="guardwhuy">
    </div>
    <button type="submit" class="btn btn-primary">修改</button>
</form>
```

3、回到员工列表页面

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

执行结果

4、发现页面提交的没有id，在前端加一个隐藏域，提交id。

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

7.8 删除员工

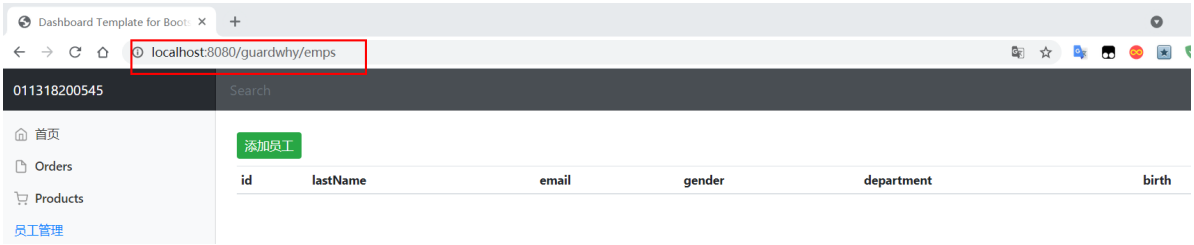
1、`list.html`页面，编写提交地址

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

2、编写删除员工Controller

```
// 删除员工
@GetMapping("/delemp/{id}")
public String deleteEmp(@PathVariable("id")int id){
    // 调用底层
    employeeDao.delete(id);
    return "redirect:/emps";
}
```

执行结果

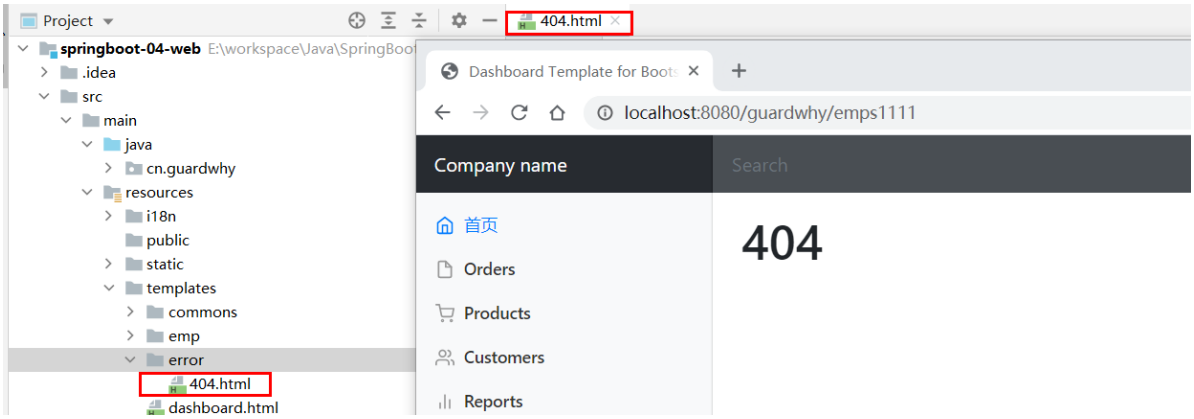


7.9 404 or 注销用户

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

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

2、注销用户controller

```
// 注销用户
@RequestMapping("/user/logout")
public String logout(HttpSession session){
    // 7.1 注销
    session.invalidate();
    // 7.2 返回到首页
    return "redirect:/index.html";
}
```

3、注销成功，返回主页面！！！

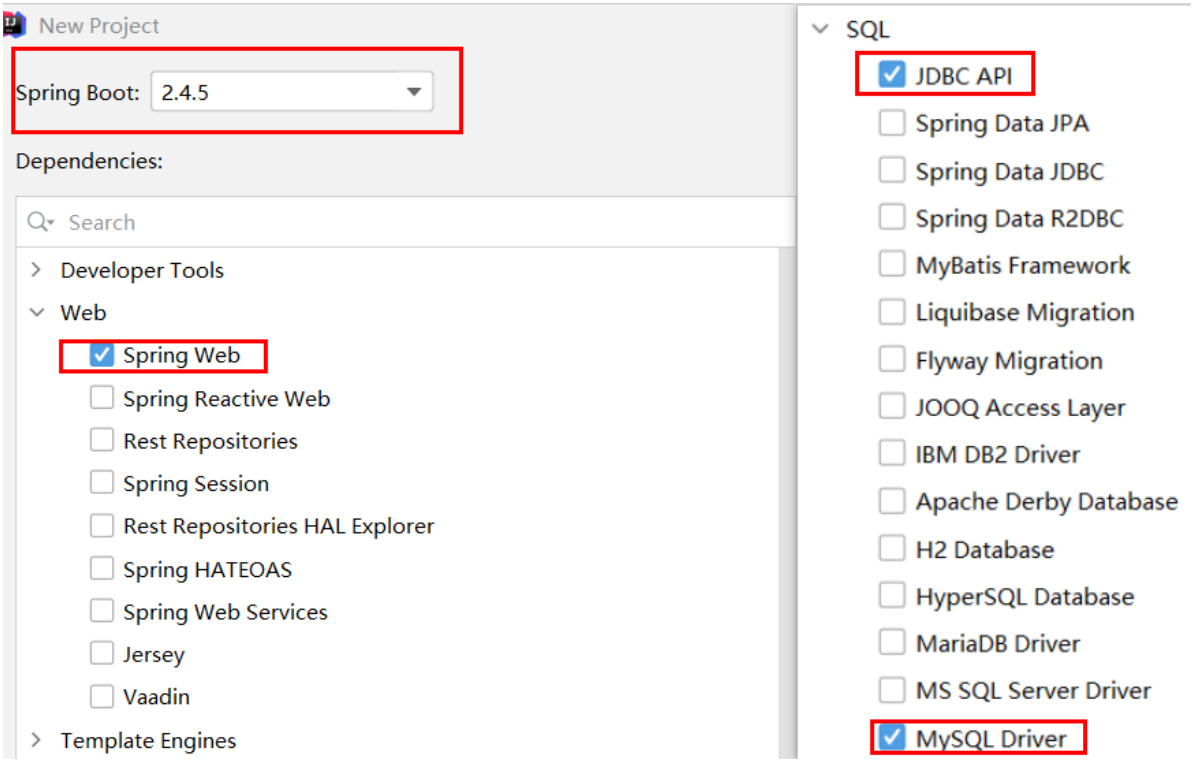
8-SpringData

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

```sql
-- 创建数据库springboot_mysql
create database if not exists springboot_mybatis;

-- 使用数据库springboot_mybatis
use springboot_mybatis

-- 创建user数据表
create table user(
    -- 字段名,字段类型
    id int primary key,
    name varchar(20),
    pwd varchar(20)
);

-- 向表中插入所有字段
insert into user values(1, 'curry', '123667'),(2, 'kobe', '12366dd'),(3,
'harden', '666624');
-- 查询数据表
select * from user;
```

3、项目建好之后，发现自动导入了如下的启动器

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

<dependency>
    <groupId>mysql</groupId>
```

```
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
```

4、编写yaml配置文件连接数据库

```
spring:
  datasource:
    username: root
    password: root
    # 假如时区报错了，就增加了一个时区的配置就ok了serverTimezone=UTC
    url: jdbc:mysql://localhost:3306/springboot_mybatis?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf-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。

```
package cn.guardwhy;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.SQLOutput;

@SpringBootTest
class Springboot05DataApplicationTests {
    @Autowired
    DataSource dataSource;

    @Test
    void contextLoads() throws SQLException {
        // 1.查看一下默认的数据源：com.zaxxer.hikari.HikariDataSource
        System.out.println(dataSource.getClass());
        // 2.获得数据库连接
        Connection connection = dataSource.getConnection();
        System.out.println(connection);
        // 3.关闭
        connection.close();
    }
}
```

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。

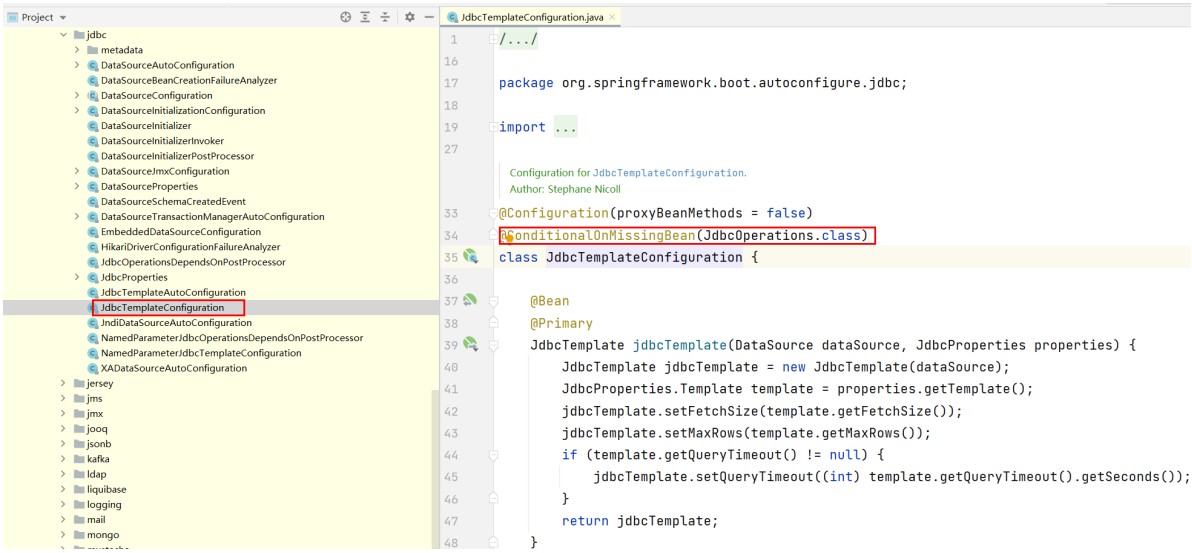
```java
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@ConditionalOnMissingBean(type = "io.r2dbc.spi.ConnectionFactory")
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ DataSourcePoolMetadataProvidersConfiguration.class, DataSourceInitializationConfiguration.class })
public class DataSourceAutoConfiguration {

    @Configuration(proxyBeanMethods = false)
    @Conditional(EmbeddedDatabaseCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import(EmbeddedDataSourceConfiguration.class)
    protected static class EmbeddedDatabaseConfiguration {

    }

    @Configuration(proxyBeanMethods = false)
    @Conditional(PooledDataSourceCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import({ DataSourceConfiguration.Hikari.class, DataSourceConfiguration.Tomcat.class,
            DataSourceConfiguration.Dbcp2.class, DataSourceConfiguration.OracleUcp.class,
            DataSourceConfiguration.Generic.class, DataSourceJmxConfiguration.class })
    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



- 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。

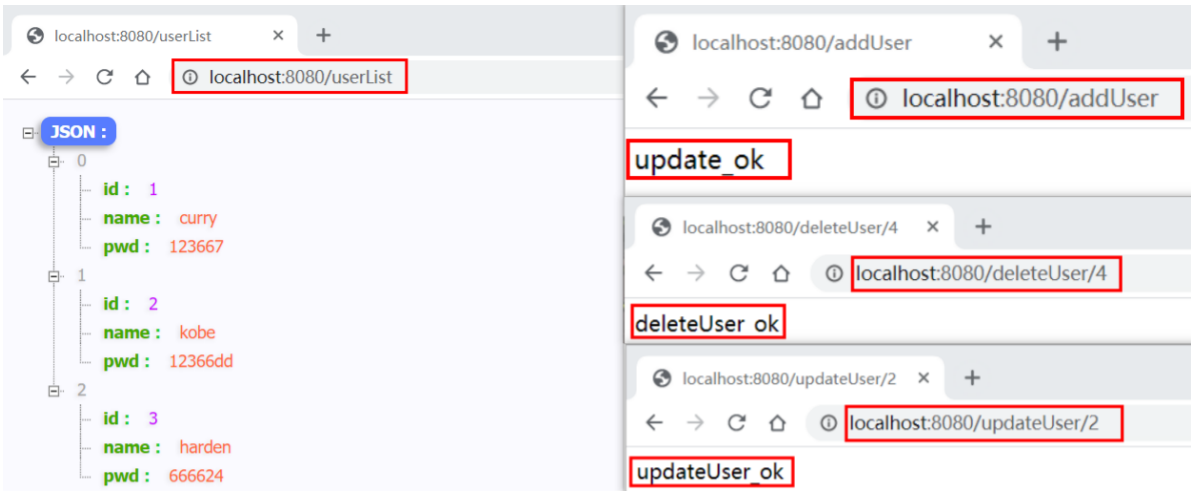
案例测试

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

```
package cn.guardwhy.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.Map;

@RestController
public class JDBCController {
    // 注入jdbcTemplate
    @Autowired
    JdbcTemplate jdbcTemplate;
    // 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
    @GetMapping("/userList")
    public List<Map<String, Object>> userList(){
        String sql = "select * from user";
        List<Map<String, Object>> list_maps =
jdbcTemplate.queryForList(sql);
        return list_maps;
    }

    // 2.添加数据
    @GetMapping("/addUser")
    public String addUser(){
        String sql = "insert into user (id, name, pwd) values(4, 'guardwhy',
'123666')";
        jdbcTemplate.update(sql);
        return "update_ok";
    }

    // 3.通过id修改数据
    @GetMapping("/updateUser/{id}")
    public String updateUser(@PathVariable("id") int id){
        String sql = "update user set name=?,pwd=? where id="+id;
        // 3.1 封装数据
        Object[] objects = new Object[2];
        objects[0] = "james";
        objects[1] = "mvp";
        jdbcTemplate.update(sql, objects);
        return "updateUser_ok";
    }

    // 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 Driud 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ur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都不会其作用。
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

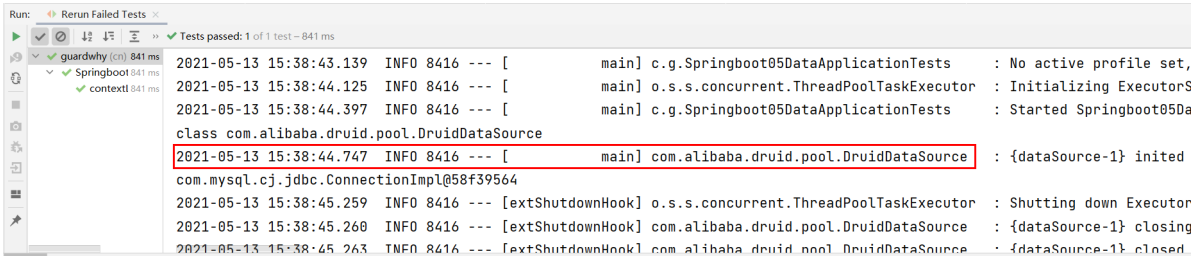
1、添加上 Druid 数据源和log4j的依赖

```
<!-- https://mvnrepository.com/artifact/log4j/log4j -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.6</version>
</dependency>
```

2、但可以 通过 spring.datasource.type 指定数据源。

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

执行结果



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。

```yaml
spring:
  datasource:
    username: root
    password: root
    url: jdbc:mysql://localhost:3306/springboot_mybatis?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
    driver-class-name: com.mysql.cj.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource

    #druid 数据源专有配置
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true
    #配置监控统计拦截的filters，stat:监控统计、log4j：日志记录、wall：防御sql注入
    #如果允许时报错 java.lang.ClassNotFoundException:org.apache.log4j.Priority
    #则导入 log4j 依赖即可，Maven 地址：
https://mvnrepository.com/artifact/log4j/log4j
    filters: stat,wall,log4j
    maxPoolPreparedStatementPerConnectionSize: 20
    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。

```java
package cn.guardwhy.config;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

// 将自定义的Druid数据源添加到容器中
@Configuration
public class DruidConfig {
```

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

2、测试代码

```
package cn.guardwhy;

import com.alibaba.druid.pool.DruidDataSource;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.SQLOutput;

@SpringBootTest
class Springboot05DataApplicationTests {
    @Autowired
    DataSource dataSource;

    @Test
    void contextLoads() throws SQLException {
        // 1.查看一下默认的数据源
        System.out.println(dataSource.getClass());
        // 2.获得数据库连接
        Connection connection = dataSource.getConnection();
        System.out.println(connection);
        // 3.获得连接
        DruidDataSource druidDataSource = (DruidDataSource) dataSource;
        System.out.println("druidDataSource 数据源最大连接数:" + druidDataSource.getMaxActive());
        System.out.println("druidDataSource 数据源初始化连接数:" + druidDataSource.getInitialSize());

        // 4.关闭
        connection.close();
    }
}
```

执行结果

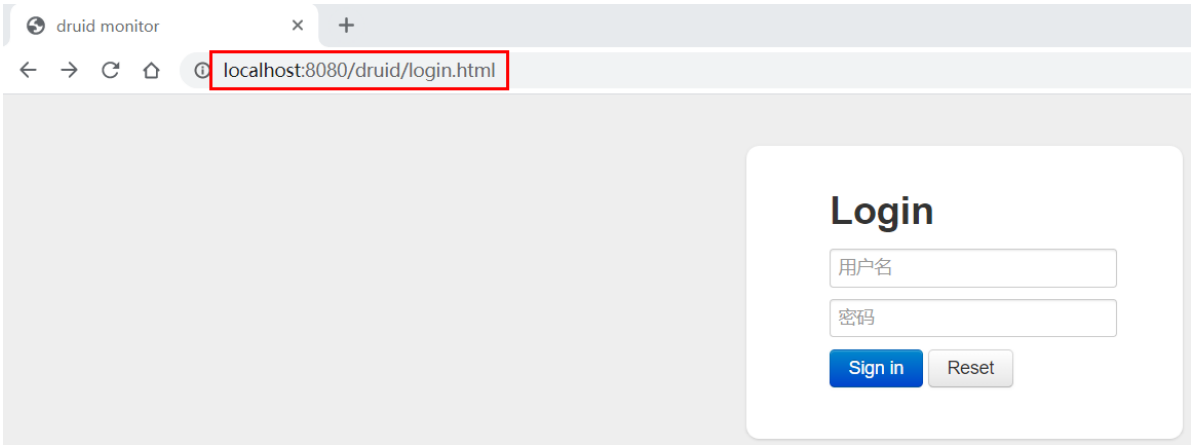
8.2.3 Druid 数据源监控

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。

```
// 后台监控
@Bean
public ServletRegistrationBean statViewServlet(){
    ServletRegistrationBean bean = new ServletRegistrationBean(new
StatViewServlet(), "/druid/*");
    // 后台需要有人登陆，账号密码配置
    HashMap<String, String> initParameters = new HashMap<>();
    // 3.增加配置
    initParameters.put("loginUsername","admin"); // 后台管理界面的登录账号
    initParameters.put("loginPassword", "123456"); // 后台管理界面的登录密码
    // 4.允许谁可以访问
    initParameters.put("allow", "");

    // 表示禁止该ip地址访问
    // initParameters.put("guardwhy","192.168.3.8");

    // 5.设置初始化参数
    bean.setInitParameters(initParameters);
    return bean;
}
```

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



监控 filter过滤器

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

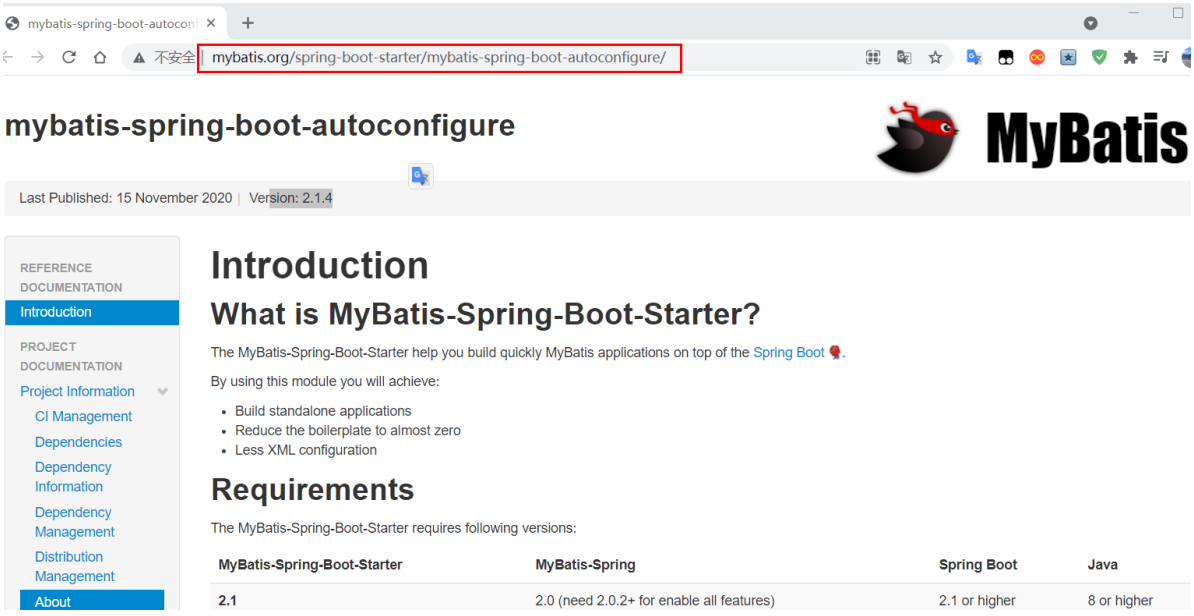
```java
// 过滤器
@Bean
public FilterRegistrationBean webStatFilter() {
    FilterRegistrationBean bean = new FilterRegistrationBean();
    bean.setFilter(new WebStatFilter());
    // 可以过滤哪些请求
    Map<String, String> initParams = new HashMap<>();
    // 这些东西不进行统计
    initParams.put("exclusions", "*.js,*.css,/druid/*");
    bean.setInitParameters(initParams);
    return bean;
}
```

执行结果

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/



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

```xml
<!-- https://mvnrepository.com/artifact/org.mybatis.spring.boot/mybatis-spring-boot-starter -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
```

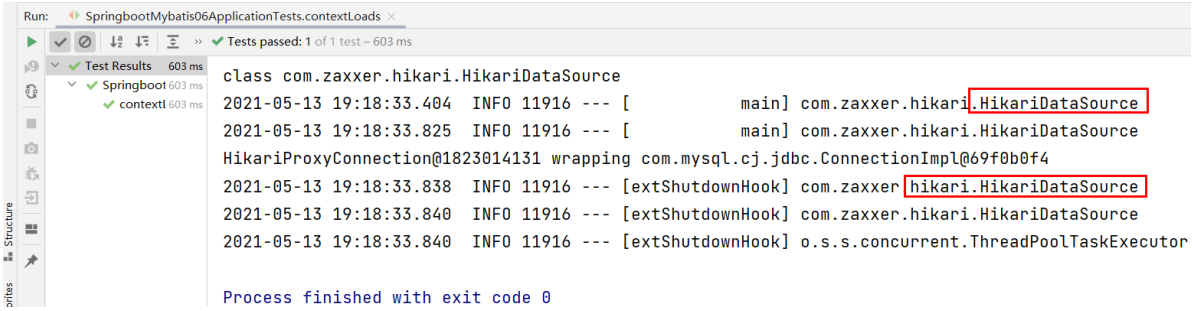
```xml
    <version>2.1.4</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.7</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

配置数据库连接信息

application.yaml

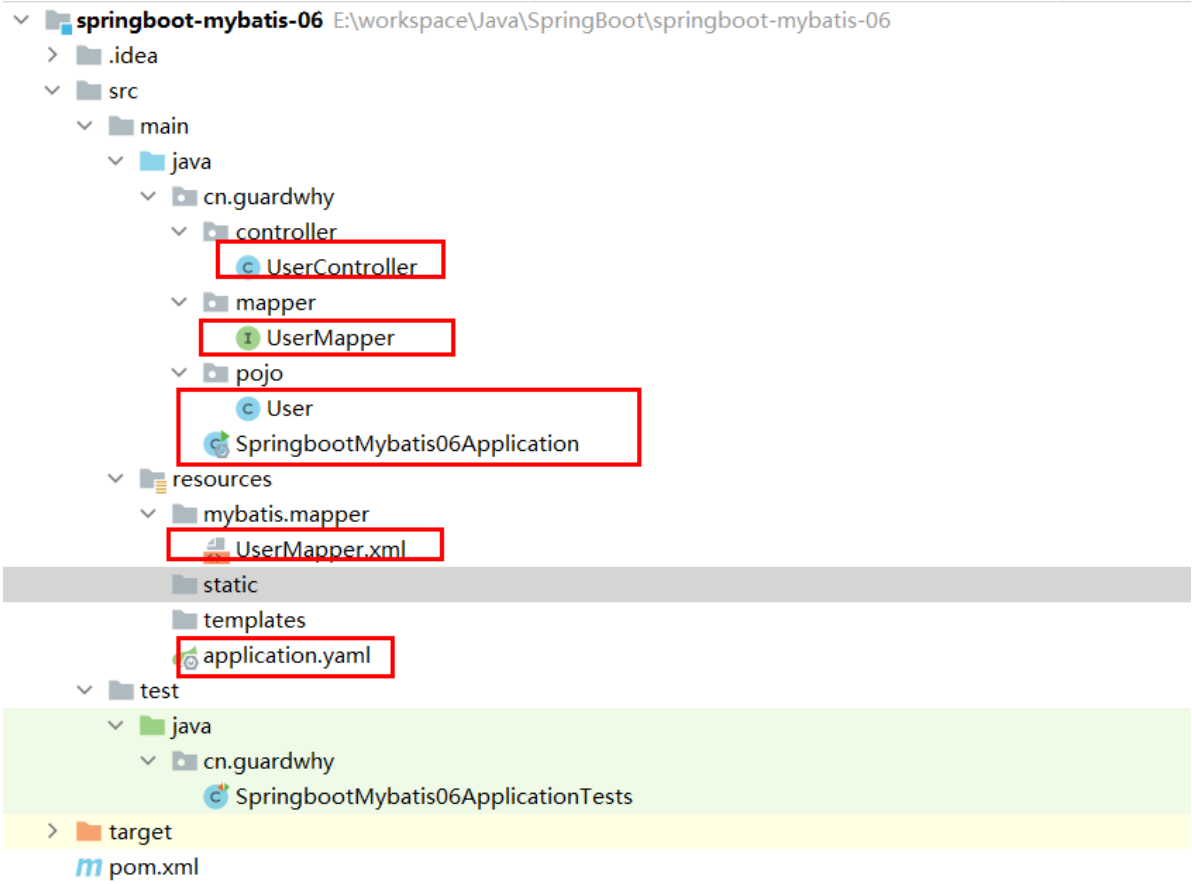
```yaml
spring:
  datasource:
    username: root
    password: root
    url: jdbc:mysql://localhost:3306/springboot_mybatis?serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
    driver-class-name: com.mysql.cj.jdbc.Driver
```

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

```java
package cn.guardwhy.pojo;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private int id;
    private String name;
    private String pwd;
}
```

3、创建mapper目录以及对应的 Mapper接口

```java
package cn.guardwhy.mapper;

import cn.guardwhy.pojo.User;
import org.apache.ibatis.annotations.Mapper;
import org.springframework.stereotype.Repository;

import java.util.List;

@Mapper // 这个注解表示了这是一个mybatis的mapper类: dao
@Repository
public interface UserMapper {
```

```
    // 1.查询所有的用户
    List<User> queryUserList();
    // 2.根据id查询用户
    User queryUserById(Integer id);
    // 3.添加用户
    int addUser(User user);
    // 4.修改用户
    int updateUser(User user);

    // 5.根据id删除用户
    int deleteUser(int id);
}
```

4、配置Mapper映射文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!--
实体类的映射文件: namespace 指定接口的类全名
-->
<mapper namespace="cn.guardwhy.mapper.UserMapper">

    <select id="queryUserList" resultType="User">
        select * from user
    </select>

    <select id="queryUserById" resultType="User">
        select * from user where id = #{id}
    </select>

    <insert id="addUser" parameterType="User" >
        insert into user (id, name, pwd) values(#{id},#{name},#{pwd})
    </insert>

    <update id="updateUser" parameterType="User">
        update user set name=#{name},pwd=#{pwd} where id=#{id}
    </update>

    <delete id="deleteUser" parameterType="int">
        delete from user where id = #{id}
    </delete>
</mapper>
```

5、myBatis 的映射配置文件

application.yaml

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

6、编写UserController进行测试

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

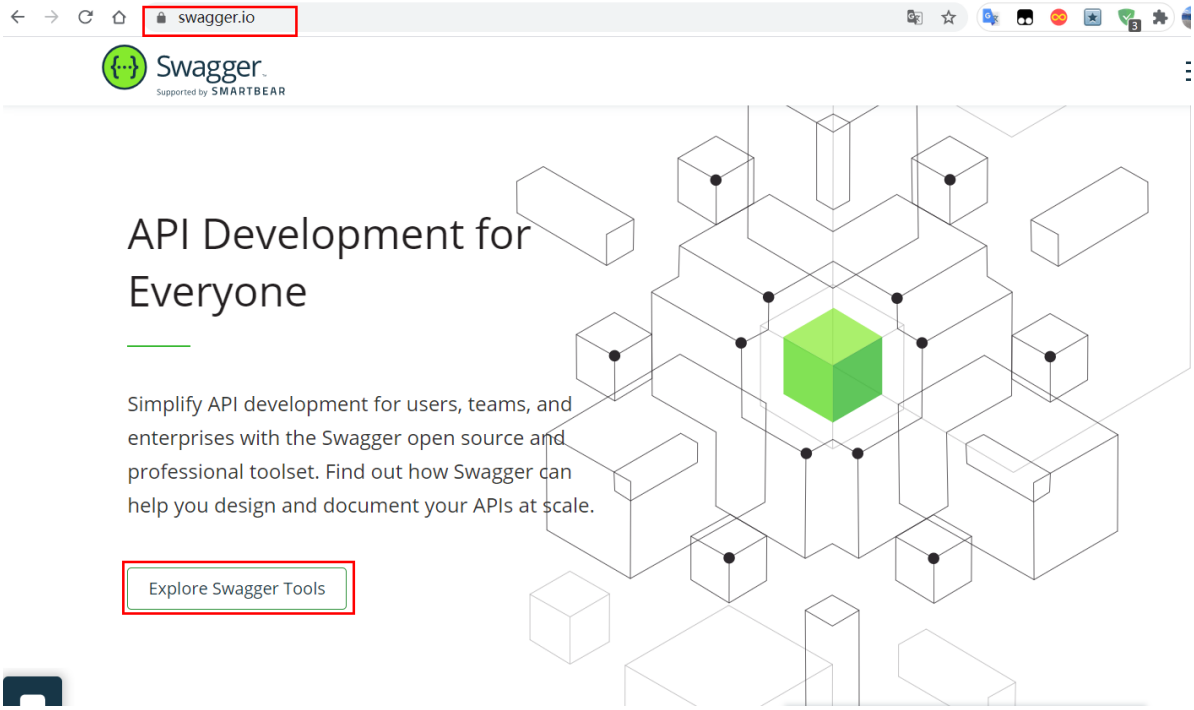
```java

import java.util.List;

@RestController
public class UserController {
    @Autowired
    private UserMapper userMapper;

    // 1.查询所有用户
    @GetMapping("/queryUserList")
    public List<User> queryUserList(){
        List<User> list_user = userMapper.queryUserList();
        for (User user : list_user) {
            System.out.println(user);
        }
        return list_user;
    }
    // 2.查询根据id查询一个用户
    @GetMapping("/queryUserById/{id}")
    public User queryUserById(@PathVariable("id") Integer id){
        return userMapper.queryUserById(id);
    }

    // 3.添加一个用户
    @GetMapping("/addUser")
    public String addUser(){
        userMapper.addUser(new User(4, "guardwhy", "123"));
        return "ok";
    }

    // 4.修改一个用户
    @GetMapping("/updateUser")
    public String updateUser(){
        userMapper.updateUser(new User(3, "Papul", "6667"));
        return "ok_updateUser";
    }

    // 5.删除一个用户
    @GetMapping("/deleteUser")
    public String deleteUser(){
        userMapper.deleteUser(6);
        return "ok_deleteUser";
    }
}
```

测试成功！！！

9- Springboot 整合Swagger

9.1 Swagger基本概念

Swagger官网：https://swagger.io/



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

```xml
<!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger2 -->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger-ui -->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

3.x.x版本

```xml
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

3、编写对应的controller

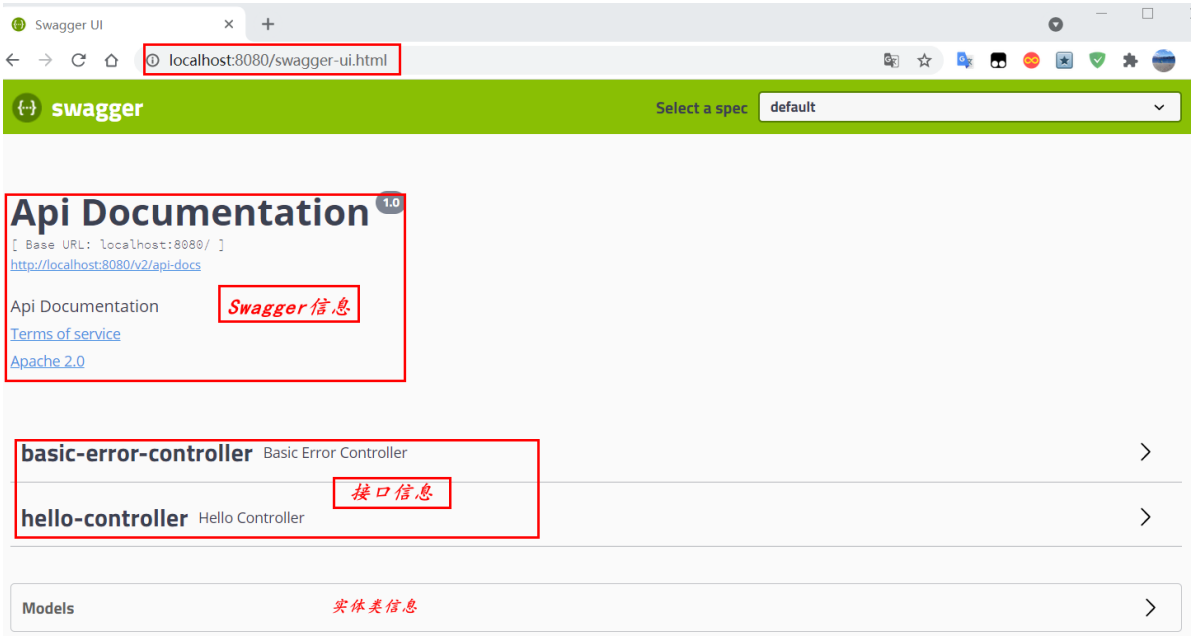
```java
package cn.guardwhy.controller;

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

@RestController
public class HelloController {
    @RequestMapping(value = "/hello")
    public String hello(){
        return "hello Swagger!!!";
    }
}
```

4、编写一个配置类-SwaggerConfig来配置 Swagger

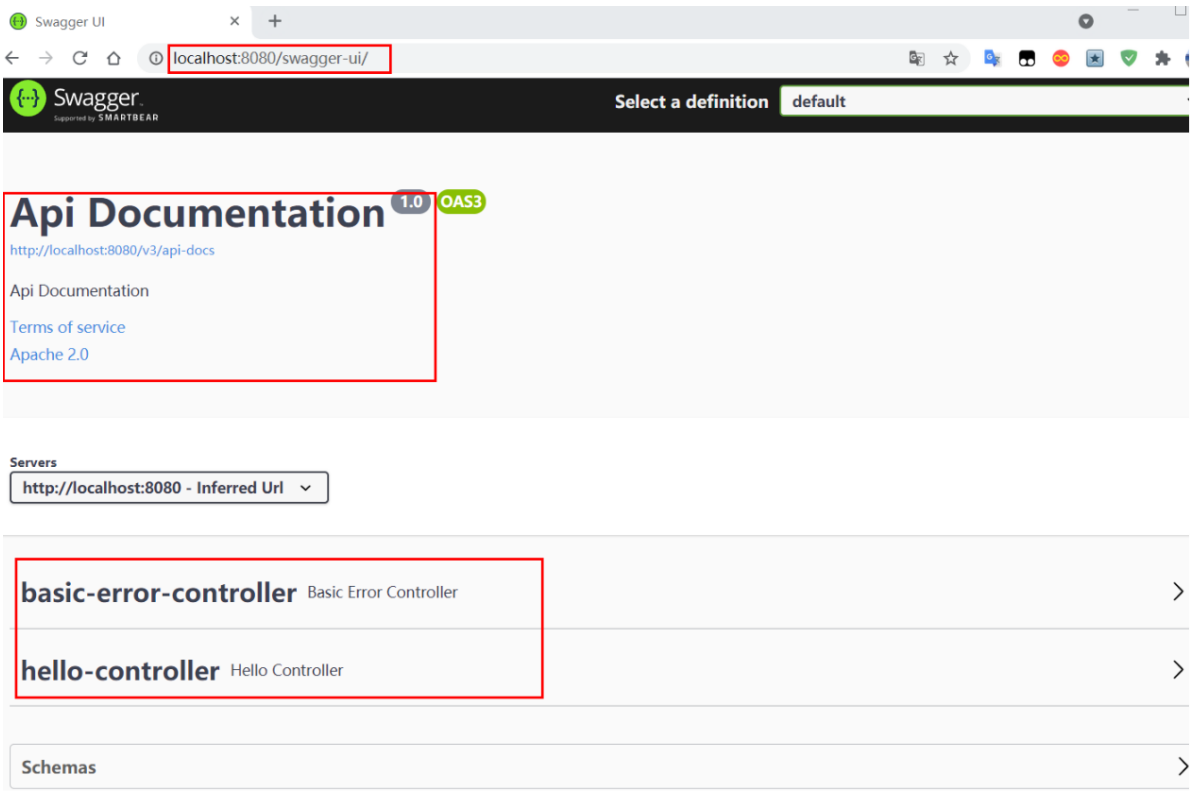
```java
package cn.guardwhy.config;

import org.springframework.context.annotation.Configuration;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration // 配置类
@EnableSwagger2 // 开启Swagger2的自动配置
public class SwaggerConfig {

}
```

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



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



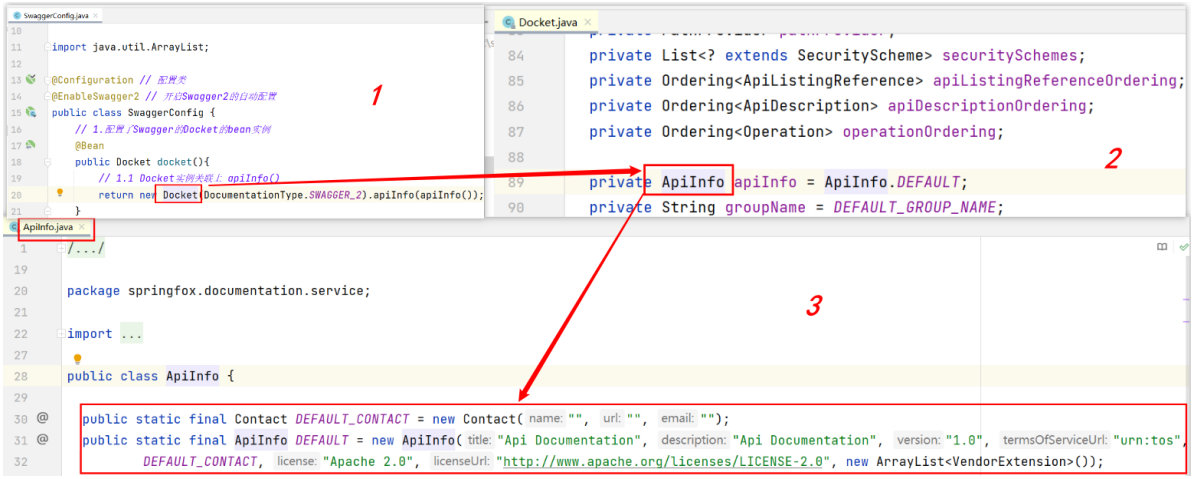
9.3 配置Swagger

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

```
package cn.guardwhy.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

```

```java
@Configuration // 配置类
@EnableSwagger2 // 开启Swagger2的自动配置
public class SwaggerConfig {
    // 1.配置了Swagger的Docket的bean实例
    @Bean
    public Docket docket(){
        return new Docket(DocumentationType.SWAGGER_2);
    }
}
```

源码分析



2、可以通过apiInfo()属性配置文档信息

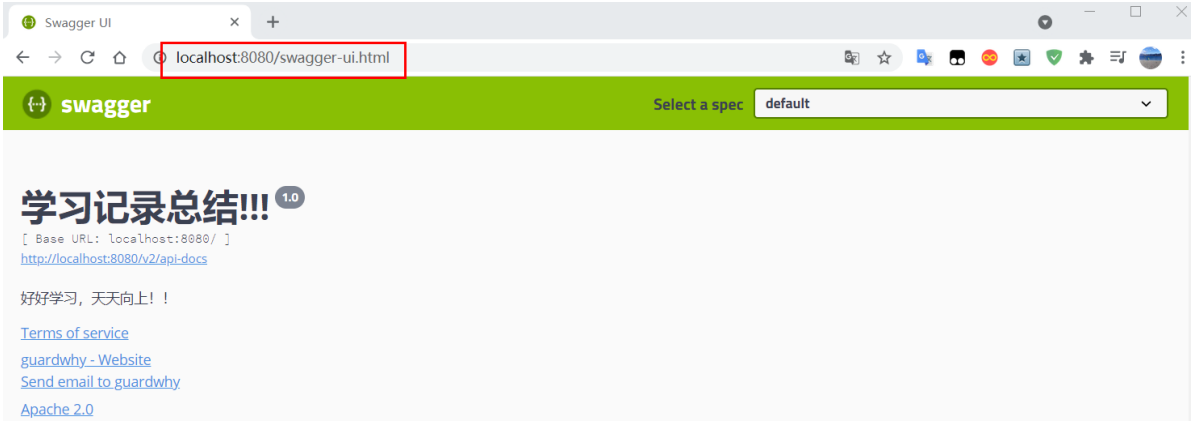
```java
// 2.配置Swagger信息=apiInfo
private ApiInfo apiInfo() {
    // 2.1 作者信息
    Contact contact = new Contact("guardwhy",
"https://home.cnblogs.com/u/Guard9/","hxy1625309592@aliyun.com");
    return new ApiInfo(
        "学习记录总结!!!",
        "好好学习，天天向上！！",
        "1.0",
        "urn:tos",
        contact,
        "Apache 2.0",
        "http://www.apache.org/licenses/LICENSE-2.0",
        new ArrayList()
    );
}
```

3、Docket 实例关联上 `apiInfo()`

```java
package cn.guardwhy.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

```

```java
import java.util.ArrayList;

@Configuration // 配置类
@EnableSwagger2 // 开启Swagger2的自动配置
public class SwaggerConfig {
    // 1.配置了Swagger的Docket的bean实例
    @Bean
    public Docket docket(){
        // 1.1 Docket实例关联上 apiInfo()
        return new Docket(DocumentationType.SWAGGER_2).apiInfo(apiInfo());
    }

    // 2.配置Swagger信息=apiInfo
    private ApiInfo apiInfo() {
        // 2.1 作者信息
        Contact contact = new Contact("guardwhy",
"https://home.cnblogs.com/u/Guard9/","hxy1625309592@aliyun.com");
        return new ApiInfo(
                "学习记录总结!!!",
                "好好学习，天天向上！！",
                "1.0",
                "urn:tos",
                 contact,
                "Apache 2.0",
                "http://www.apache.org/licenses/LICENSE-2.0",
                new ArrayList()
        );
    }
}
```

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

```java
@Bean
public Docket docket(){
    // 1.1 Docket实例关联上 apiInfo()
    return new Docket(DocumentationType.SWAGGER_2)
        .apiInfo(apiInfo())
        .select()
        // RequestHandlerSelectors,配置要扫描接口的方式
        // basePackage:指定要扫描的包
        // any():扫描全部
```

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

2、配置接口扫描过滤

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

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。

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

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

```
spring.profiles.active=dev
```

application-dev.properties

```
server.port=8081
```

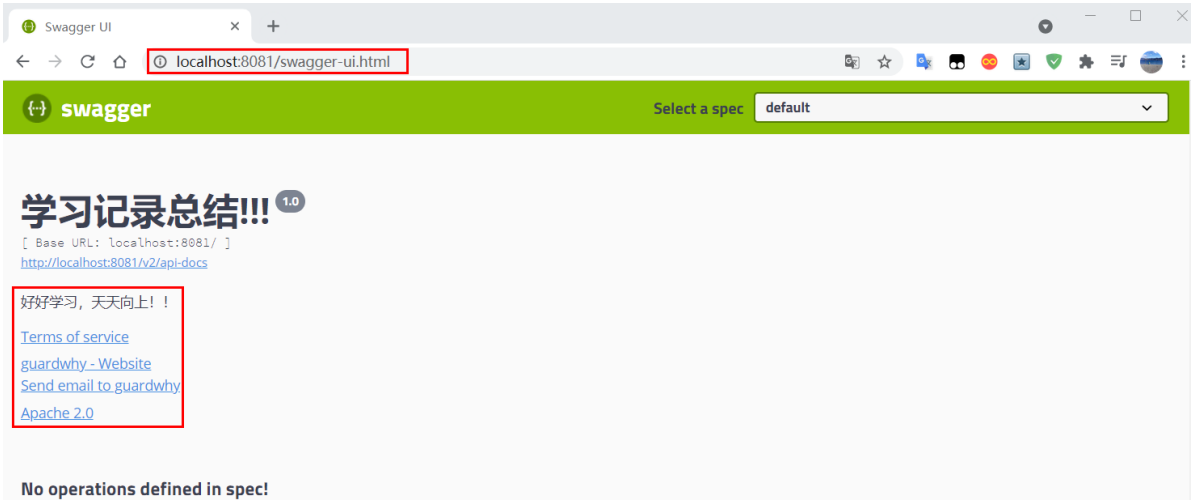
application-pro.properties

```
server.port=8082
```

SwaggerConfig

```
@Bean
public Docket docket(Environment environment){
    // 1.设置要显示Swagger的环境
    Profiles profiles = Profiles.of("dev","test");
    // 2.通过environment.acceptsProfiles 判断是否处在自己设定的环境当中
    boolean flag = environment.acceptsProfiles(profiles);
    System.out.println(flag);

    return new Docket(DocumentationType.SWAGGER_2)
        .apiInfo(apiInfo())
        .enable(flag) // enable是否启动Swagger，如果为False,则Swagger不能再浏览器中的访问！！
        .select()
        .apis(RequestHandlerSelectors.basePackage("cn.guardwhy.controller"))
        // 配置如何通过path过滤,即这里只扫描请求以/guardwhy开头的接口
        .paths(PathSelectors.ant("/guardwhy/**"))
        .build();
}
```

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



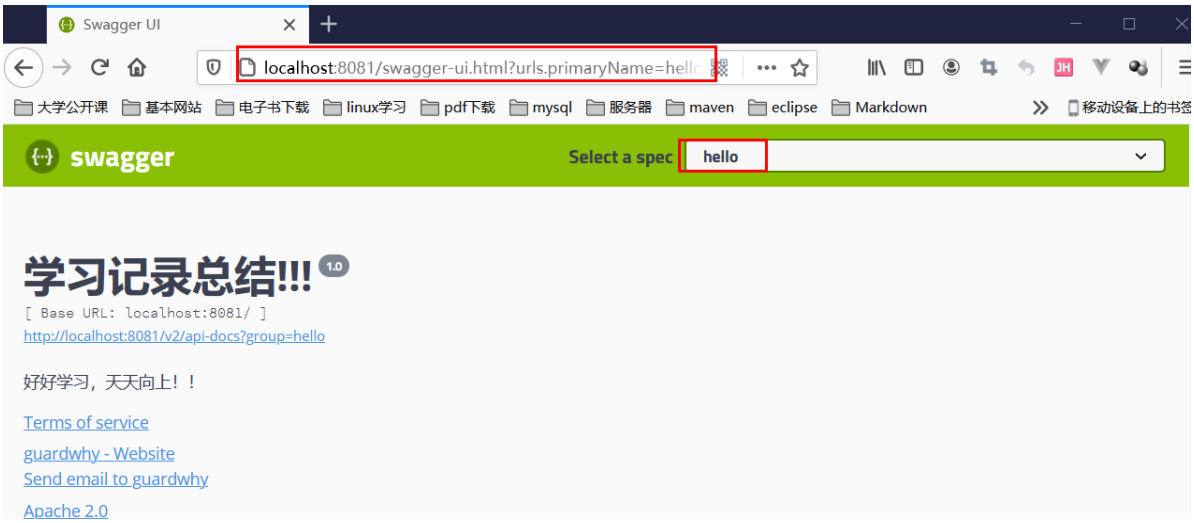
9.6 分组和实体类配置

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

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

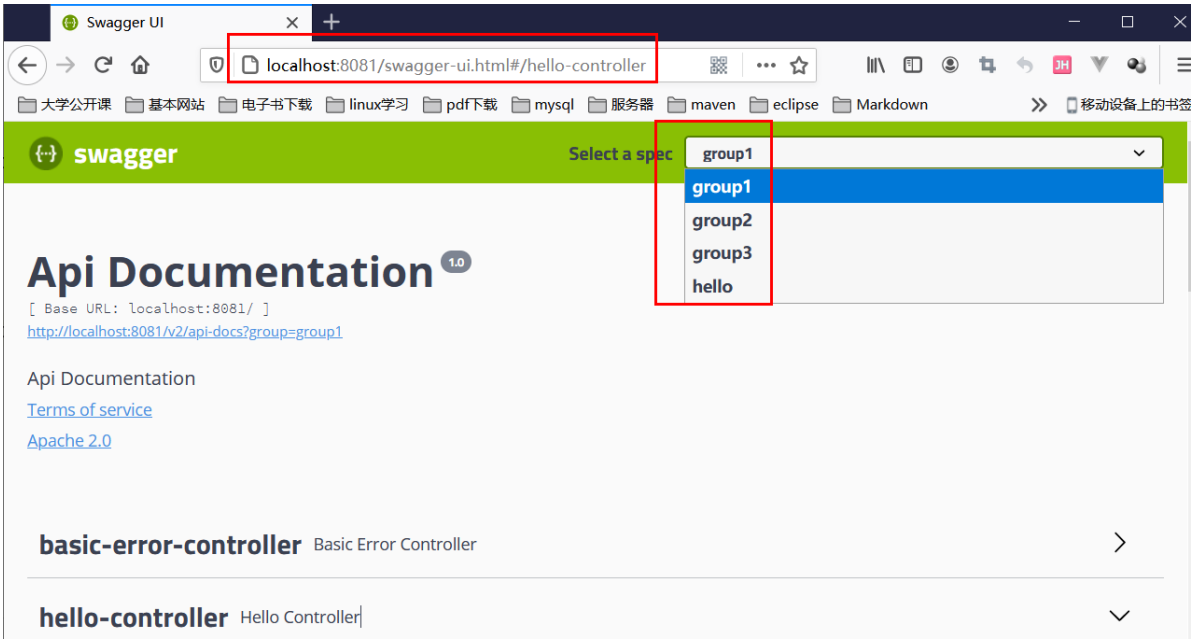
2、启动项目，查看结果！！！



3、配置多个分组只需要配置多个docket。

```java
// 3.配置多个分组
@Bean
public Docket docket1(){
    return new Docket(DocumentationType.SWAGGER_2).groupName("group1");
}

@Bean
public Docket docket2(){
    return new Docket(DocumentationType.SWAGGER_2).groupName("group2");
}

@Bean
public Docket docket3(){
    return new Docket(DocumentationType.SWAGGER_2).groupName("group3");
}
```

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



9.6.2 实体配置

1、新建一个实体类`User类`

@ApiModel为类添加注释

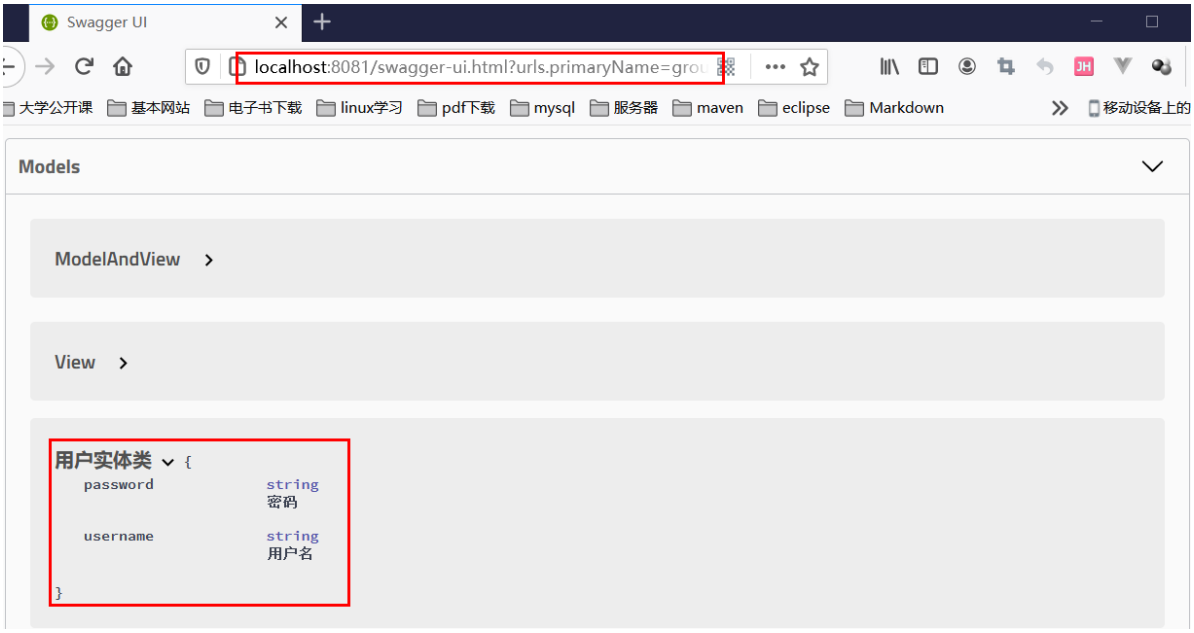
@ApiModelProperty为类属性添加注释

```
package cn.guardwhy.pojo;

import io.swagger.annotations.ApiModel;
import io.swagger.annotations.ApiModelProperty;

@ApiModel("用户实体类")
public class User {
    @ApiModelProperty("用户名")
    public String username;
    @ApiModelProperty("密码")
    public String password;
}
```

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

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

3、重启项目，执行结果！！！



9.6.3 小结

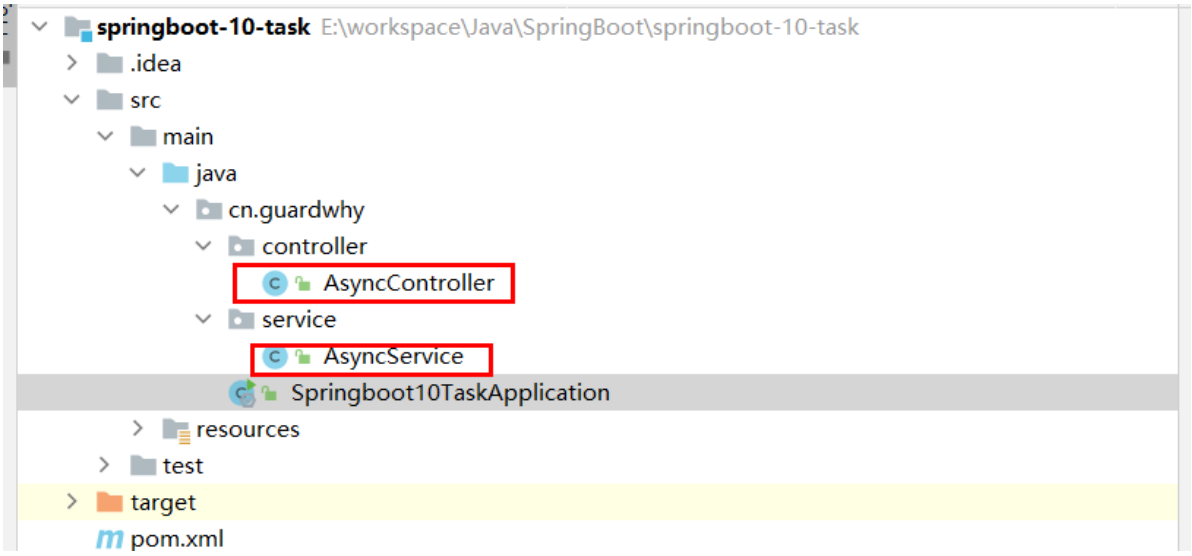
Swagger是个优秀的工具，较于传统的Postman或Curl方式测试接口，使用swagger简直就是傻瓜式操作，不需要额外说明文档。

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

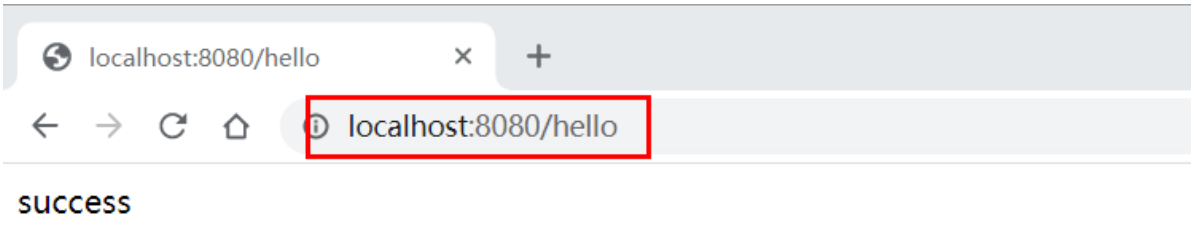
```
1  package cn.guardwhy.service;
2
3  import org.springframework.stereotype.Service;
4
5  @Service
```

```
public class AsyncService {
    public void hello(){

        try {
            // 休眠3秒
            Thread.sleep(30000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 输出结果
        System.out.println("数据处理中.....");
    }
}
```

3、编写AsyncController类

```
package cn.guardwhy.controller;

import cn.guardwhy.service.AsyncService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class AsyncController {
    @Autowired
    AsyncService asyncService;

    @RequestMapping("/hello")
    public String hello(){
        asyncService.hello(); // 停止3S，转圈
        return "success";
    }
}
```

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。

```
package cn.guardwhy.service;

import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;

@Service
public class AsyncService {
    @Async  // 告诉Spring这是一个异步方法
    public void hello(){
```

```
        try {
            // 休眠3秒
            Thread.sleep(30000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 输出结果
        System.out.println("数据处理中.....");
    }
}
```

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。

```
package cn.guardwhy;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableAsync;

@SpringBootApplication
@EnableAsync // 开启异步注解功能
public class Springboot10TaskApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot10TaskApplication.class, args);
    }

}
```

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

```
package cn.guardwhy.service;

import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Service;

@Service
public class ScheduledService {
    //秒   分   时     日   月   周几
    /*
    *  30 15 10 * *      每天10点15分30 执行一次
    *  30 0/5 10, 18 * *?   每天10点和18点，每隔五分钟执行一次
    * 0 15 10 ? * 1-6   每个月的周一到周六 10.15分钟执行一次
    */
```

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

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：

```
[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。

```
package cn.guardwhy;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.scheduling.annotation.EnableScheduling;

@SpringBootApplication
@EnableAsync // 开启异步注解功能
@EnableScheduling // 开启定时功能的注解
public class Springboot10TaskApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot10TaskApplication.class, args);
    }
}
```

10.3 邮件任务

1、 导入相关的依赖

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

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

MailSenderAutoConfiguration.java

```
1   /.../
16
17  package org.springframework.boot.autoconfigure.mail;
18
19  import ...
33
        Auto configuration for email support.
        Since:  1.2.0
        Author: Oliver Gierke, Stephane Nicoll, Eddú Meléndez
42  @Configuration(proxyBeanMethods = false)
43  @ConditionalOnClass({ MimeMessage.class, MimeType.class, MailSender.class })
44  @ConditionalOnMissingBean(MailSender.class)
45  @Conditional(MailSenderCondition.class)
46  @EnableConfigurationProperties(MailProperties.class)
47  @Import({ MailSenderJndiConfiguration.class, MailSenderPropertiesConfiguration.class })
48  public class MailSenderAutoConfiguration {
49
        Condition to trigger the creation of a MailSender. This kicks in if either the host or jndi name
        property is set.
54      static class MailSenderCondition extends AnyNestedCondition {...}
71
```

3、查看自动配置文件`MailProperties`

```java
@ConfigurationProperties(prefix = "spring.mail")
public class MailProperties {

    private static final Charset DEFAULT_CHARSET = StandardCharsets.UTF_8;

    /**
     * SMTP server host. For instance, `smtp.example.com`.
     */
    private String host;

    /**
     * SMTP server port.
     */
    private Integer port;

    /**
     * Login user of the SMTP server.
     */
    private String username;

    /**
     * Login password of the SMTP server.
     */
    private String password;

    /**
     * Protocol used by the SMTP server.
     */
    private String protocol = "smtp";

    /**
     * Default MimeMessage encoding.
```

```
     */
    private Charset defaultEncoding = DEFAULT_CHARSET;

    /**
     * Additional JavaMail Session properties.
     */
    private Map<String, String> properties = new HashMap<>();

    /**
     * Session JNDI name. When set, takes precedence over other Session
settings.
     */
    private String jndiName;
}
```

4、配置文件

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

5、 单元测试

```
package cn.guardwhy;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSenderImpl;
import org.springframework.mail.javamail.MimeMessageHelper;

import javax.mail.MessagingException;
import javax.mail.internet.MimeMessage;
import java.io.File;

@SpringBootTest
class Springboot10TaskApplicationTests {
    @Autowired
    JavaMailSenderImpl mailSender;

    @Test
    void contextLoads1() {
        // 1.一个简单的邮件
        SimpleMailMessage mailMessage = new SimpleMailMessage();
        mailMessage.setSubject("hello Springboot!!!");
        mailMessage.setText("Springboot源码分析");

        // 发送
        mailMessage.setTo("1625309592@qq.com");
        mailMessage.setFrom("1625309592@qq.com");

        mailSender.send(mailMessage);
    }
```

```
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
    File("C:\\Users\\linux\\Pictures\\my.jpg"));
45          helper.addAttachment("timg.jpg", new
    File("C:\\Users\\linux\\Pictures\\timg.jpg"));
46
47          // 发送
48          helper.setTo("1625309592@qq.com");
49          helper.setFrom("1625309592@qq.com");
50
51          mailSender.send(mimeMessage);
52      }
53
54  }
```

6、启动项目，执行结果！！！