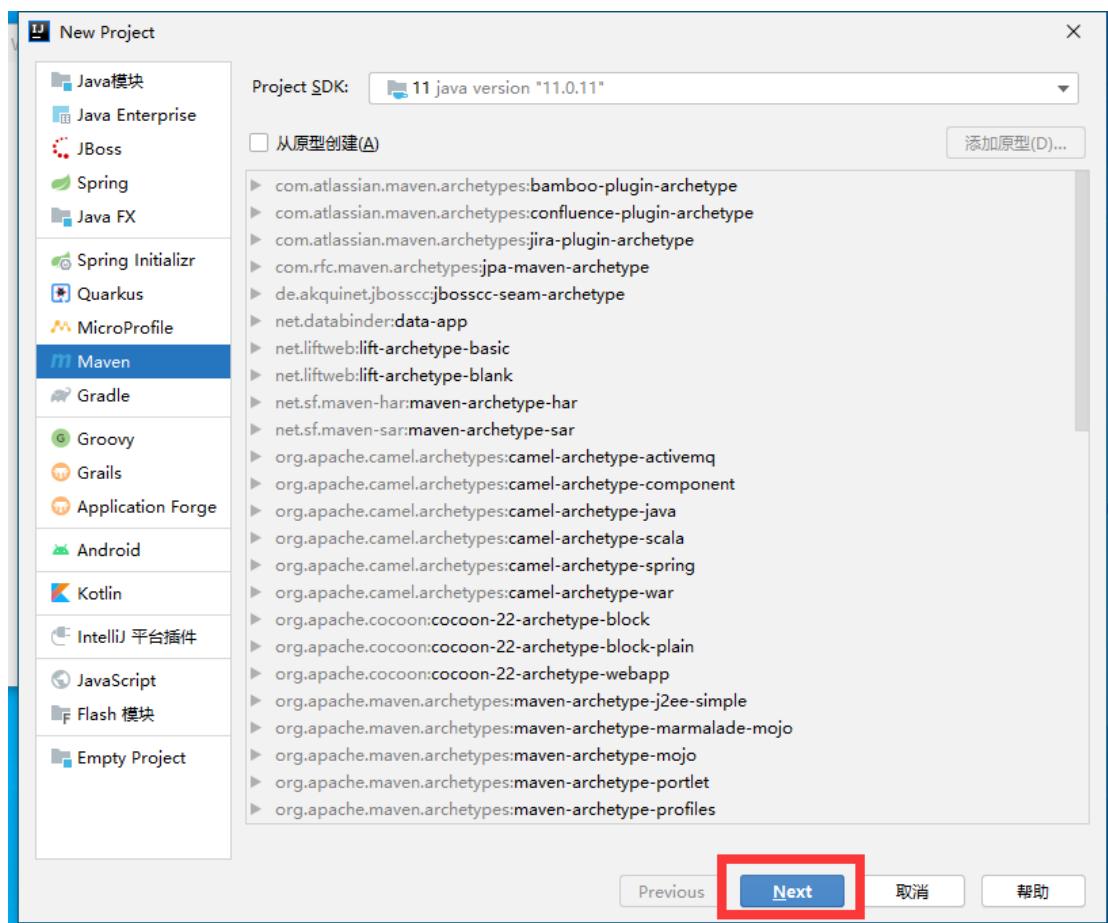
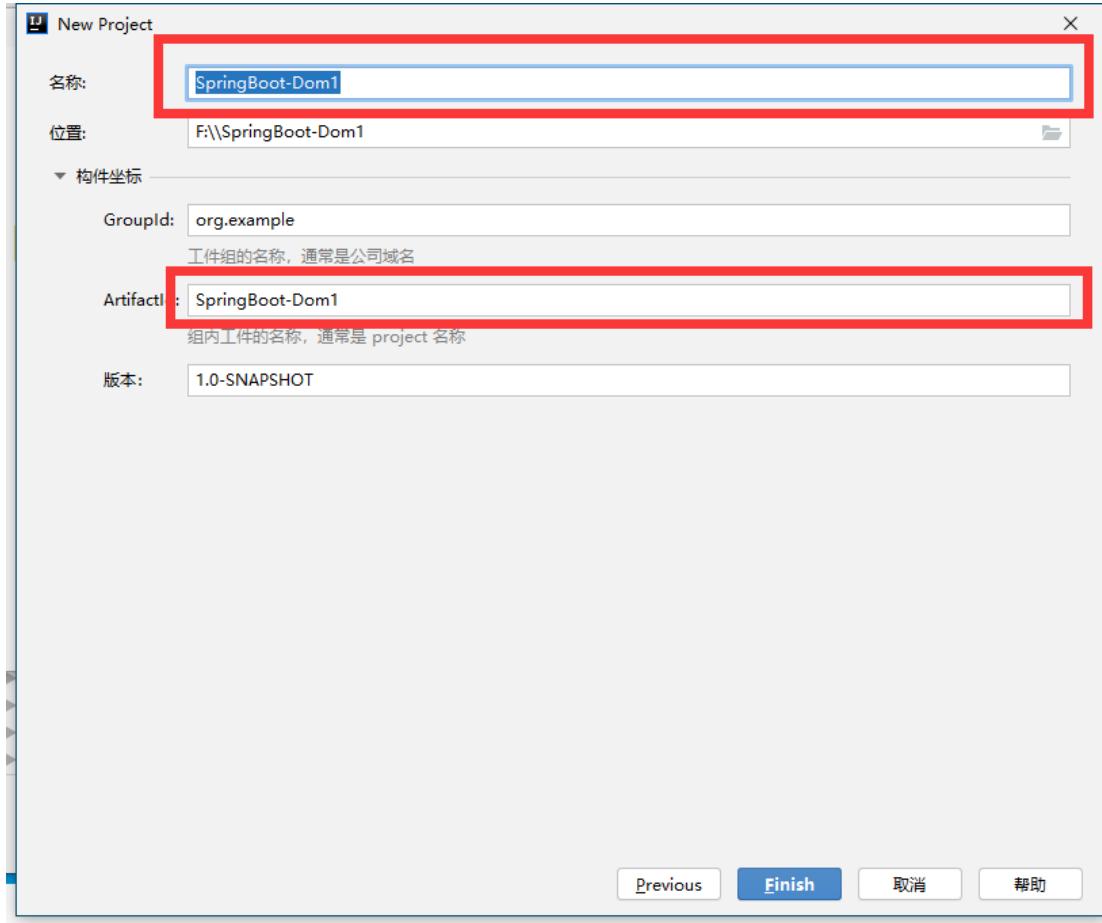


SpringBoot2

一、创建一个简单的工程

(一)、HelloWored



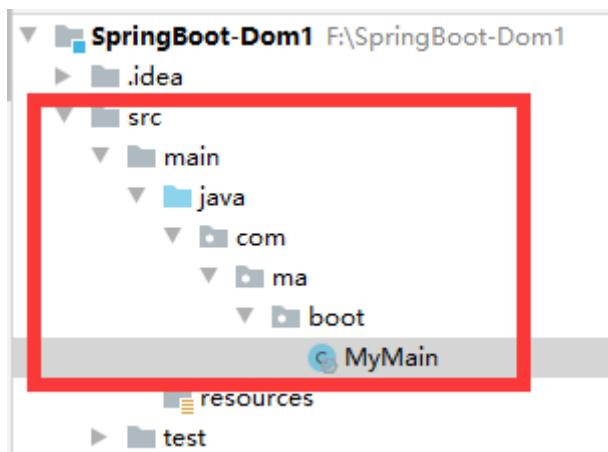


(二)、在 POM 中添加依赖

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.4.RELEASE</version>
</parent>

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

(三)、创建主程序



(四)、使用 @SpringBootApplication 注解声明这是一个 SpringBoot 程序

```
@SpringBootApplication  
public class MyMain {  
}
```

(五)、创建 main 方法让主程序跑起来

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

(六)、创建控制层类

```
//Controller 和 RequestBody 的合体
@RestController
public class MyController {

    /**
     * 创建一个控制器
     */
    @RequestMapping("/hello")
    public String test1(){
        return "hello";
    }
}
```

(七)、创建配置文件 application.properties

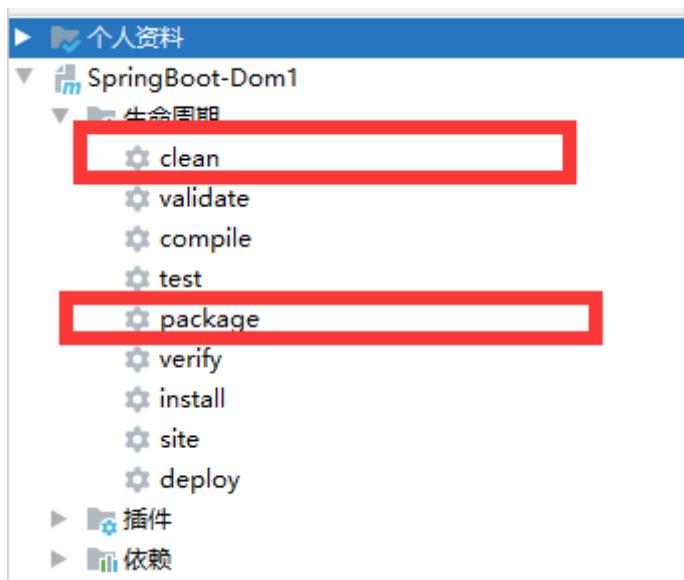
```
#设置端口号
server.port=8888
```

(八)、一键打包设置

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

```
</plugins>  
</build>
```

(九)、打包



(十)、修改自己需要的版本依赖（如 mysql）

```
<properties>  
    <mysql.version>8.0.21</mysql.version>  
</properties>
```

(十一)、扩展包扫描范围

```
@SpringBootApplication(scanBasePackages = "com.ma")  
public class MyMain {  
  
    public static void main(String[] args) {  
        SpringApplication.run(MyMain.class,args);  
    }  
}
```

```
}
```

二、注解

(一)、@configuration 注解

说明：

该注解用于类上，表明该类是一个 Spring 管理器
proxyBeanMethods 属性：

当 proxyBeanMethods 为 false 的时候就不是单例模式了

当为 true 的时候都是从容器中获取的一个对象

```
@Configuration(proxyBeanMethods = true)
public class MyController {

    /**
     * 当@Bean 后面每值的时候 ioc 中容器的名字为方法名
     * 当@Bean 中有名字的时候 ioc 中容器名字为 value 值
     * @return
     */
    @Bean("user")
    public User user1(){
        return new User(18,"小马");
    }

    @Bean
    public Pat cat(){
        return new Pat("小猫","吃老鼠");
    }
}
```

使用：

```
//调用启动器获取对象  
  
ConfigurableApplicationContext run = SpringApplication.run(MyMain.class, args);  
  
  
  
//获得 ioc 容器中 user 对象  
//测试@Bean 中带名字  
  
User user = run.getBean("user", User.class);  
System.out.println(user);  
  
  
  
//获得 ioc 容器中 cat  
//测试@Bean 中不带名字  
  
Pat cat = run.getBean("cat", Pat.class);  
System.out.println(cat);
```

(二)、@import 注解 (“User.class”)

说明：

导入特定的组件， 默认是类的全类名

```
@Import({User.class})
```

测试：

```
//获取全部组件  
  
String[] beanNamesForType = run.getBeanNamesForType(User.class);  
for (String s : beanNamesForType) {  
    System.out.println(s);  
}
```

结果：

```
com.ma.boot.pojo.User  
user
```

(三)、@conditional 注解

说明：

当达成某个条件的时候才向 ioc 中注册组件

使用：

```
//当容器中有 cat 组件的时候才注册 user  
  
@ConditionalOnBean(name = "user")  
@Bean("user")  
public User user1(){  
    return new User(18,"小马");  
}  
  
  
//当容器中有 cat 的时候注册 MyController 中的所有方法  
  
@Import({User.class})  
@Configuration  
@ConditionalOnBean(name = "user")  
public class MyController {
```

(四)、@ImportResource 注解

说明：

将 XML 中的所有 Bean 注册到容器中

XML：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <beans>
        <bean id="haha" class="com.ma.boot.pojo.User">
            <property name="name" value="zhangsan"/>
            <property name="age" value="19"/>
        </bean>
    </beans>
</beans>
```

Spring 配置类

```
@ImportResource(classPath:beans.xml)

Public class MyConfig
```

(五)、@ConfigurationProperties(prefix = "person")

说明:

读取配置文件注入属性,但 pojo 类必须在容器中

(六)、第一种方式:pojo 加入容器

配置文件:

```
# person 属性注入
person.age=18
```

```
person.name=马佳盛  
person.gender=男
```

Pojo 类:

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
@Component           //加入容器  
@ConfigurationProperties(prefix = "person") //开启文件配置注入前缀为 person  
public class Person {  
    private int age;  
    private String name;  
    private String gender;  
}
```

控制器测试:

```
@RestController  
public class MyController {  
  
    @Autowired           //自动装配  
    private Person person;  
  
    /**  
     * 创建一个控制器  
     */  
    @RequestMapping("/hello")  
    public String test1(){  
        return "hello";  
    }  
}
```

```
@RequestMapping("/hello1")
public Person test2(){
    return person;
}
}
```

(七)、第二种方式:使用 **EnableConfigurationProperties**

控制器测试:

```
@RestController
@EnableConfigurationProperties(Person.class)      //开启配置文件注入并加入容器
public class MyController {

    @Autowired
    private Person person;

    /**
     * 创建一个控制器
     */
    @RequestMapping("/hello")
    public String test1(){
        return "hello";
    }

    @RequestMapping("/hello1")
    public Person test2(){
        return person;
    }
}
```

三、SpringBoot 的自动配置

(一)、自动配置【源码分析】-自动包规则原理

1、Spring Boot 启动类

```
@SpringBootApplication          //声明该类为 SpringBoot 配置类  
public class MainApplication {  
  
    public static void main(String[] args) {  
  
        SpringApplication.run(MainApplication.class, args);  
  
    }  
  
}
```

2、SpringBootApplication 底层

```
@Target(ElementType.TYPE)  
  
@Retention(RetentionPolicy.RUNTIME)  
  
@Documented  
  
@Inherited  
  
@SpringBootConfiguration      //配置类  
  
@EnableAutoConfiguration     //开启自动装配  
  
@ComponentScan(               //包扫描  
  
    excludeFilters = {@Filter(
```

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

重 点 分 析 `@SpringBootConfiguration` , `@EnableAutoConfiguration` , `@ComponentScan`

3、`@SpringBootConfiguration`

只是一个配置类

4、`@ComponentScan`

只是一个扫描类

5、`@EnableAutoConfiguration`

```
@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 {};  
  
}
```

重点分析：

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

6、**@AutoConfigurationPackage**

自动配置包，指定了默认的包规则。

```
@Target(ElementType.TYPE)  
  
@Retention(RetentionPolicy.RUNTIME)  
  
@Documented  
  
@Inherited  
  
@Import(AutoConfigurationPackages.Registrar.class)//给容器中导入一个组件  
  
public @interface AutoConfigurationPackage {  
  
    String[] basePackages() default {};
```

```
Class<?>[] basePackageClasses() default {};  
}
```

1. 利用 Registrar 给容器中导入一系列组件
2. 将指定的一个包下的所有组件导入进 MainApplication 所在包下。

(二)、自动配置【源码分析】 - 初始加载自动配置类

1、@Import(AutoConfigurationImportSelector.class)

(1)、利用 getAutoConfigurationEntry(annotationMetadata);给容器中批量导入一些组件

(2) 、调用 List<String> configurations =
getCandidateConfigurations(annotationMetadata, attributes)获取
到所有需要导入到容器中的配置类

(3) 、利用工厂加载 Map<String, List<String>>
loadSpringFactories(@Nullable ClassLoader classLoader);得到所有的
组件

(4) 、从 META-INF/spring.factories 位置来加载一个文件。

4.1 默认扫描我们当前系统里面所有 META-
INF/spring.factories 位置的文件

4.2 、spring-boot-autoconfigure-2.3.4.RELEASE.jar 包里面
也有 META-INF/spring.factories

```
# 文件里面写死了 spring-boot 一启动就要给容器中加载的所有配置类  
  
# spring-boot-autoconfigure-2.3.4.RELEASE.jar/META-INF/spring.factories  
  
# Auto Configure
```

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\  
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\  
...  
...
```

虽然我们 127 个场景的所有自动配置启动的时候默认全部加载，但是
`xxxxAutoConfiguration` 按照条件装配规则（`@Conditional`），最终会按需配置。

(三)、总结:

1. SpringBoot 先加载所有的自动配置类 `xxxxxAutoConfiguration`
2. 每个自动配置类按照条件进行生效， 默认都会绑定配置文件指定的值。（`xxxxProperties` 里面读取，`xxxProperties` 和配置文件进行了绑定）
3. 生效的配置类就会给容器中装配很多组件
4. 只要容器中有这些组件， 相当于这些功能就有了
5. 定制化配置
 - 5.1 用户直接自己`@Bean` 替换底层的组件
 - 5.2 用户去看这个组件是获取的配置文件什么值就去修改。

四、yaml 的用法

(一)、基本语法

- `key: value;` kv 之间有空格

- 大小写敏感
- 使用缩进表示层级关系
- 缩进不允许使用 `tab`, 只允许空格
- 缩进的空格数不重要, 只要相同层级的元素左对齐即可
- '#'表示注释
- 字符串无需加引号, 如果要加, 单引号''、双引号""表示字符串内容会被 转义、不转义

(二)、数据类型

1、基础类型

K: v 如:name: 张三

2、对象类型

2.1、行内写法

K: {k1:k1,k2:k2,k3:k3}

2.2、多行写法

K:

K1:k1

K2:k2

K3:k3

3、数组

3.1、行内写法

K: [k1,k2,k3]

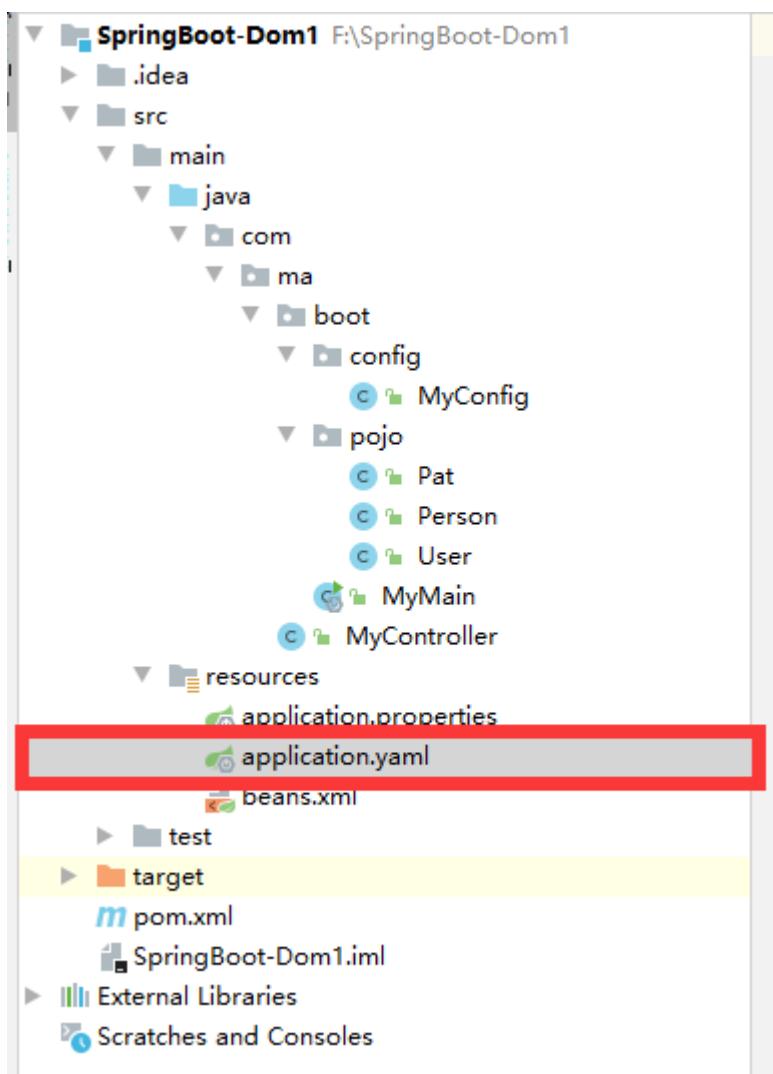
3.2、多行写法

K:

- K1
- K2
- K3

(三)、实例:

1、创建一个 yaml 文件



2、创建实体类

2.1、创建 person 类

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
@Component  
@ConfigurationProperties(prefix = "person")  
public class Person {  
  
    private String userName;  
    private Boolean boss;  
    private Date birth;  
    private Integer age;  
    private Pet pet;  
    private String[] interests;  
    private List<String> animal;  
    private Map<String, Object> score;  
    private Set<Double> salarys;  
    private Map<String, List<Pet>> allPets;  
}
```

2.2、创建 Pet 类

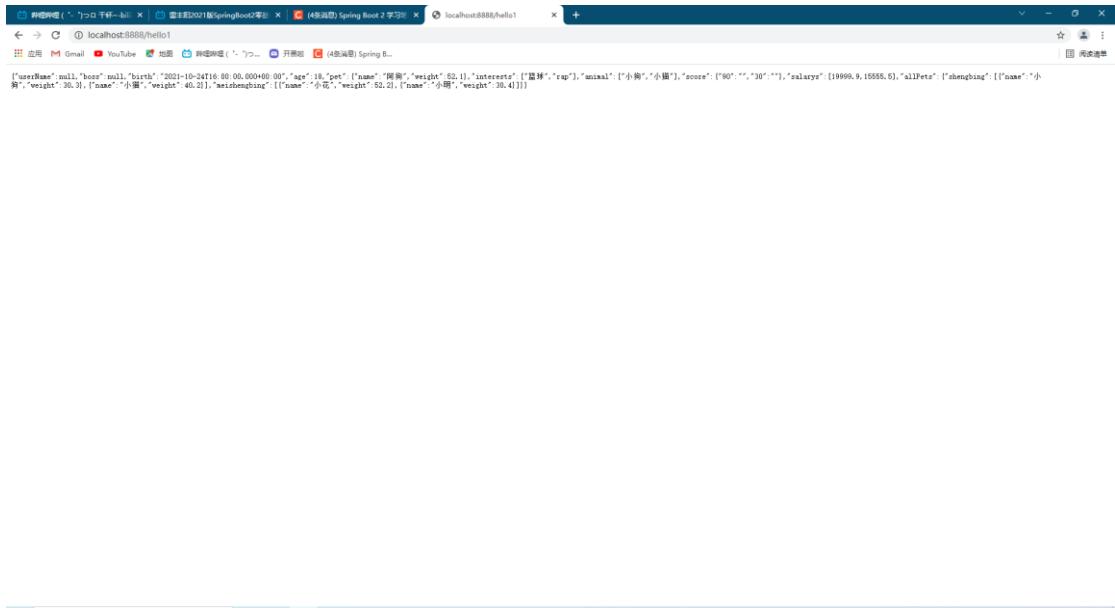
```
@Data  
@AllArgsConstructor  
public class Pet {  
  
    private String name;  
    private Double weight;  
}
```

3、注入属性

```
person:  
    userName: zhangsan  
    boss: true  
    birth: 2021/10/25  
    age: 18  
  
pet:  
    name: 阿狗  
    weight: 52.1  
  
interests: [篮球,rap]  
animal: [小狗,小猫]  
score: {语文:90,数学:30}  
salarys: [19999.9,15555.5]  
  
allPets:  
    shengbing:  
        - {name: 小狗,weight: 30.3}  
        - {name: 小猫,weight: 40.2}  
  
meishengbing:  
    - {name: 小花,weight: 52.2}  
    - {name: 小明,weight: 30.4}
```

4、测试:

```
@RequestMapping("/hello1")  
public Person test2(){  
    return person;  
}
```



附加:Yaml 中单引号和双引号的区别

单引号:纯字符串

双引号:\n 代表换行

(四)、提示插件

1、添加依赖

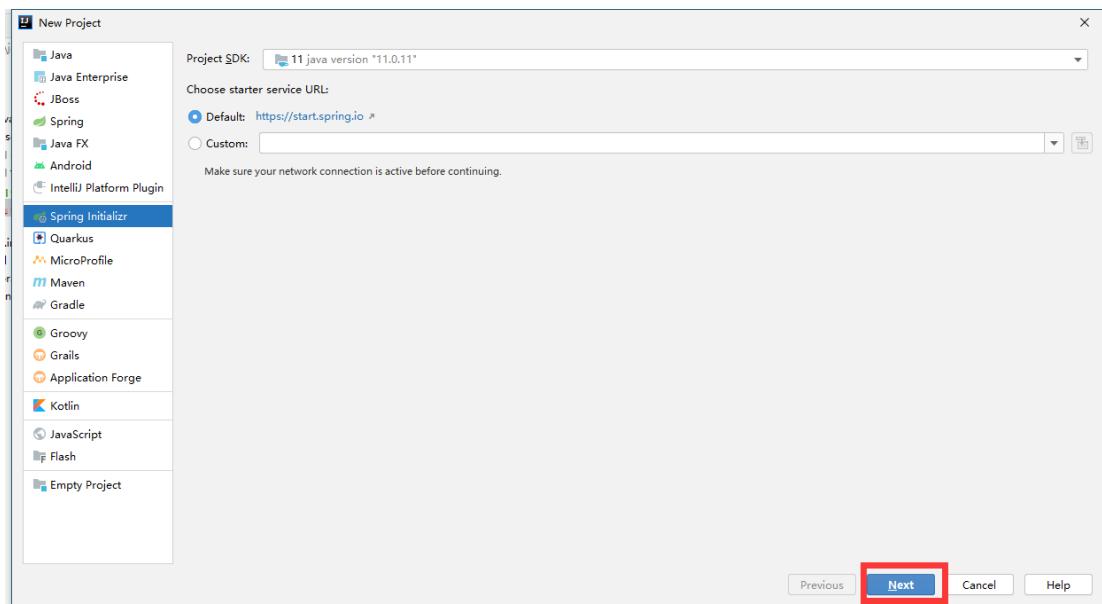
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

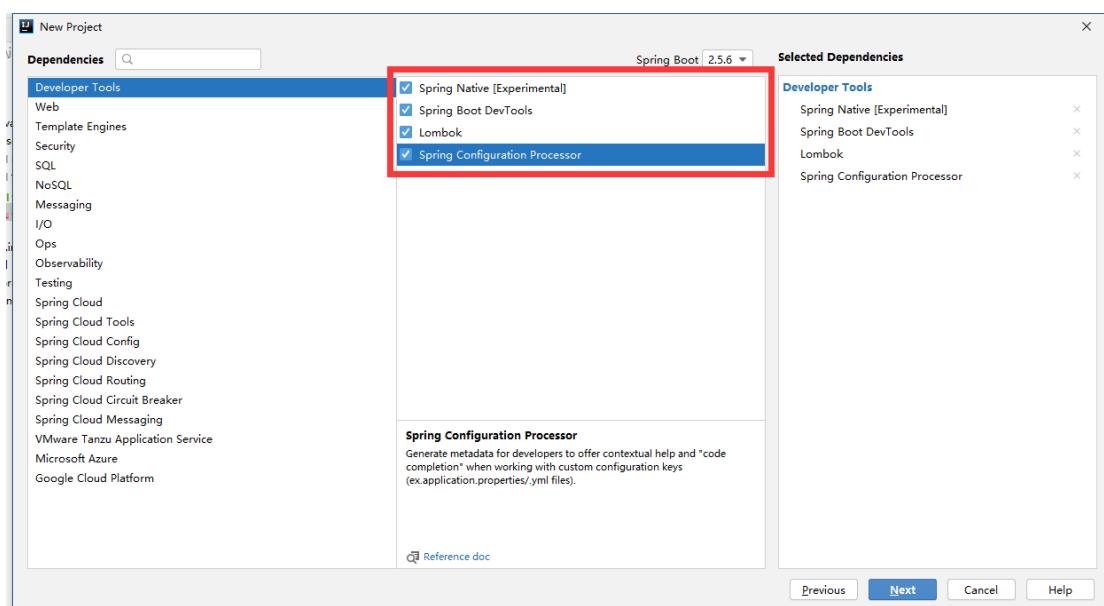
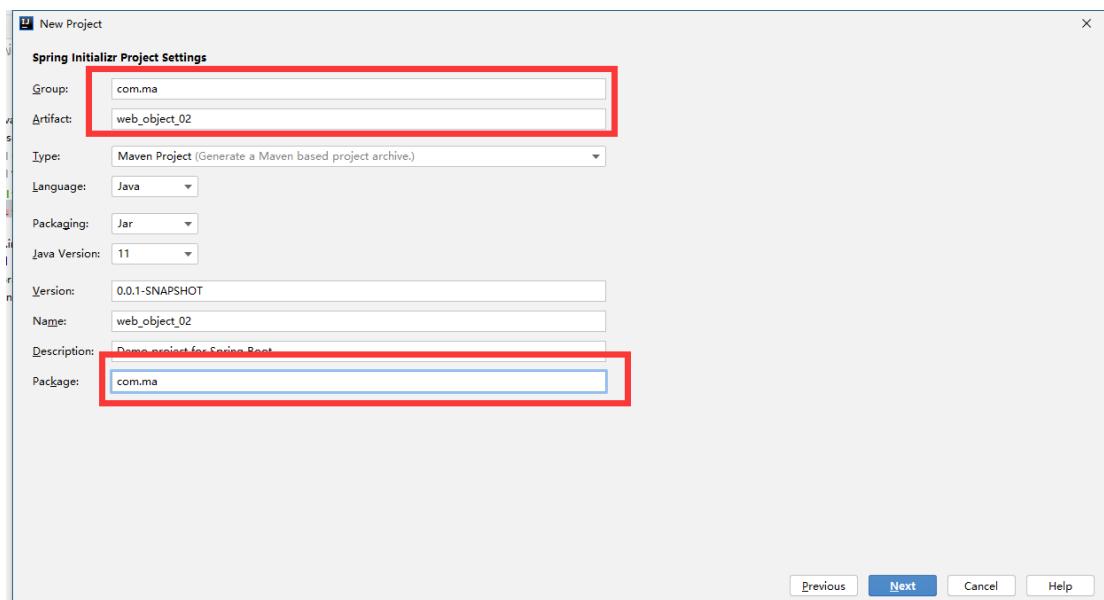
1、 打包时不打包

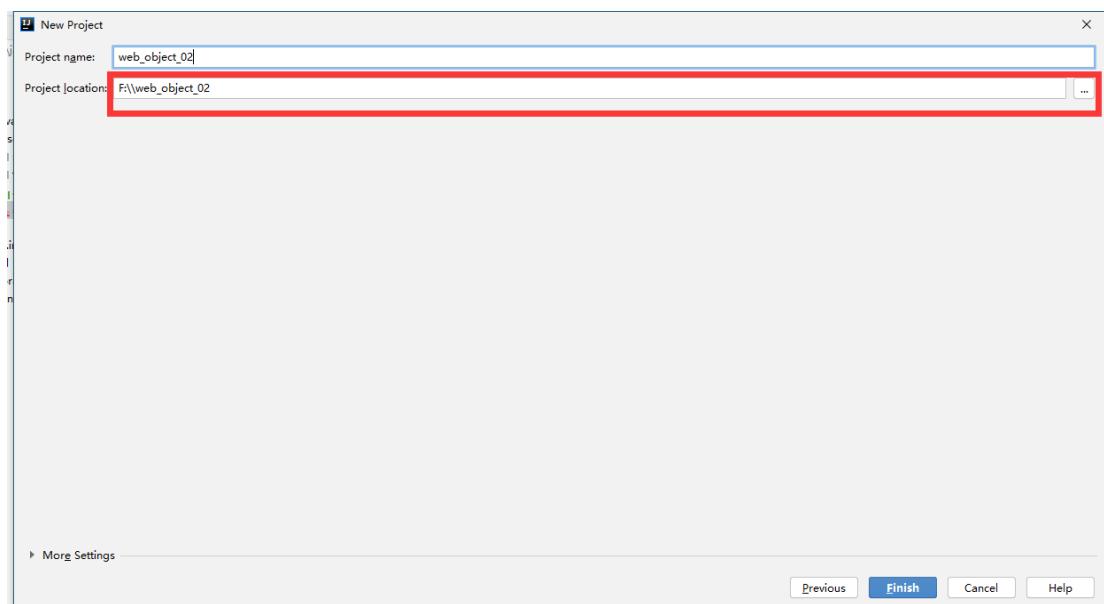
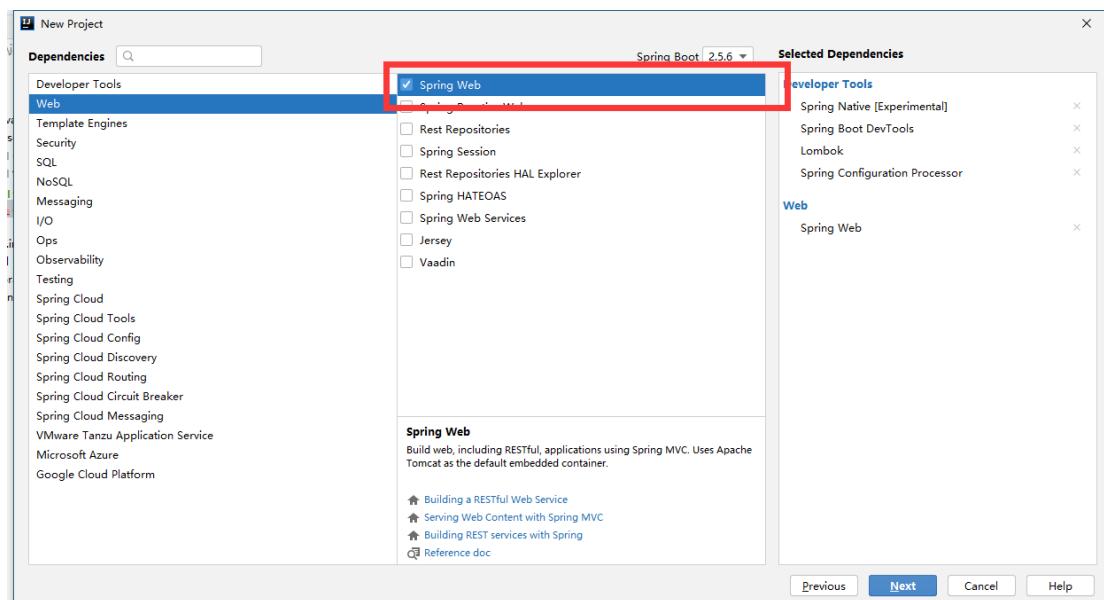
```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <excludes>
          <exclude>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
          </exclude>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

五、创建一个简单的 web 工程

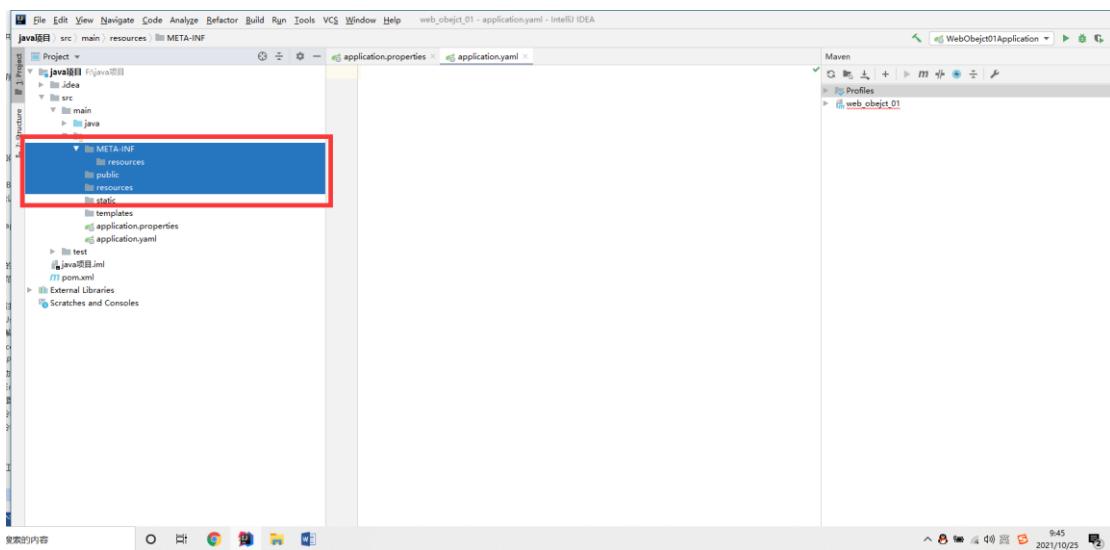
(一)、创建项目



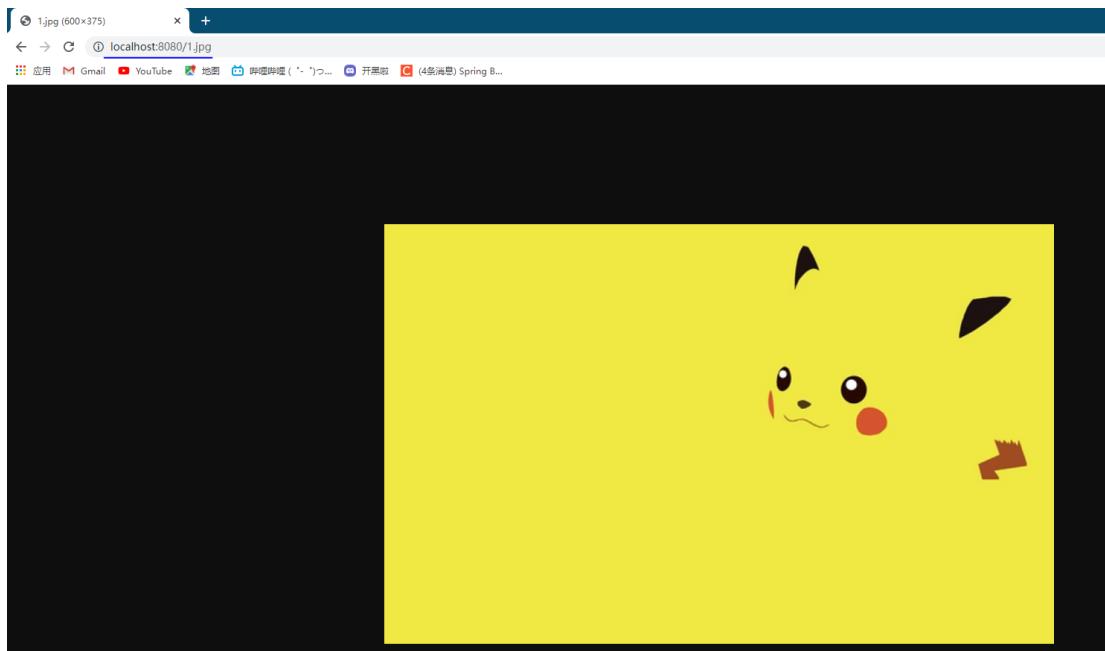




(二)、创建文件夹



(三)、访问直接网址/资源名



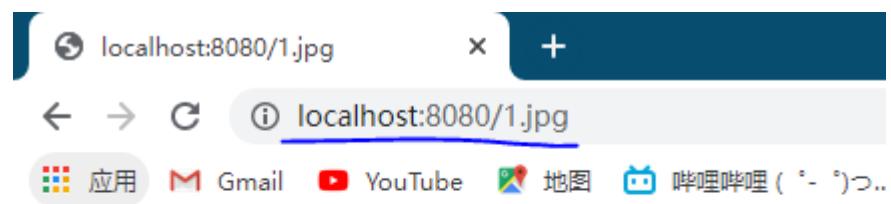
六、静态资源的处理

(一)、更改访问位置

1、在控制类中添加名字相同的资源名

```
@org.springframework.stereotype.Controller  
@ResponseBody  
public class Controller {  
  
    @RequestMapping("1.jpg")  
    public String test1(){  
        return "aaa";  
    }  
}
```

2、测试



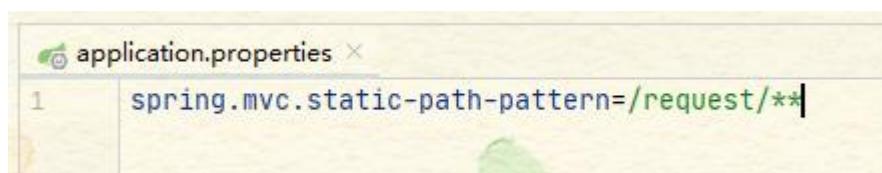
aaa

3、原理

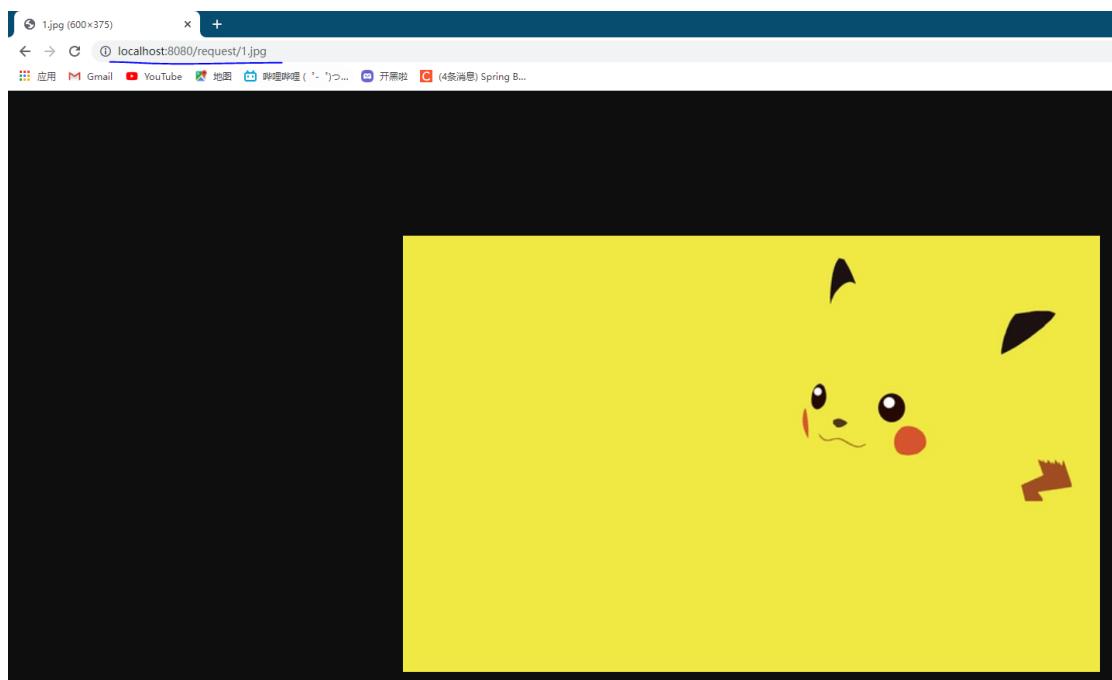
默认访问的资源名拦截请求是`/**`,默认在全部位置查找
`RequestMapping` 也是`/**`,默认全部查找
所以先找动态资源(`RequestMapping`),再找静态资源

(二)、更改默认资源访问位置

在 `application.properties` 中添加
`Spring.mvc.static-path-pattern=访问路径`

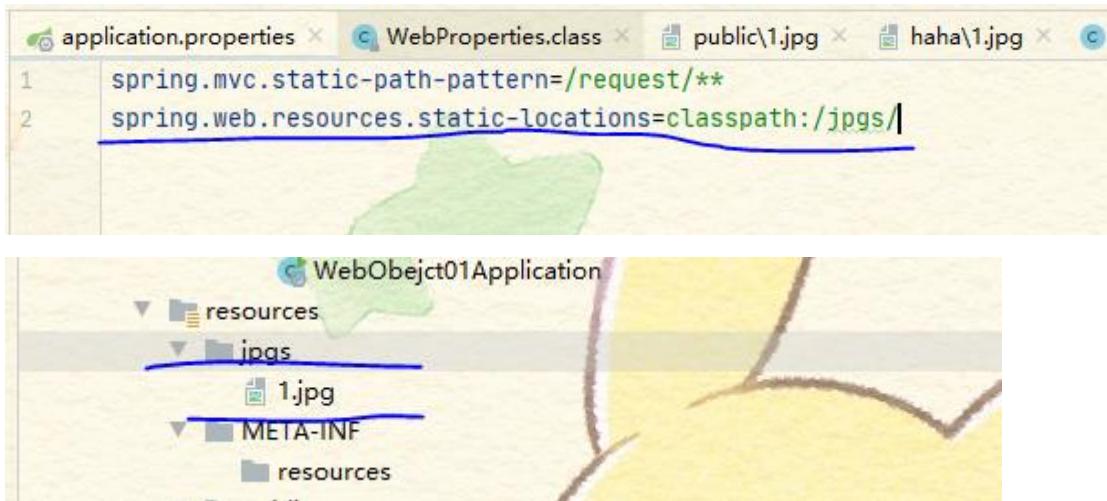


测试

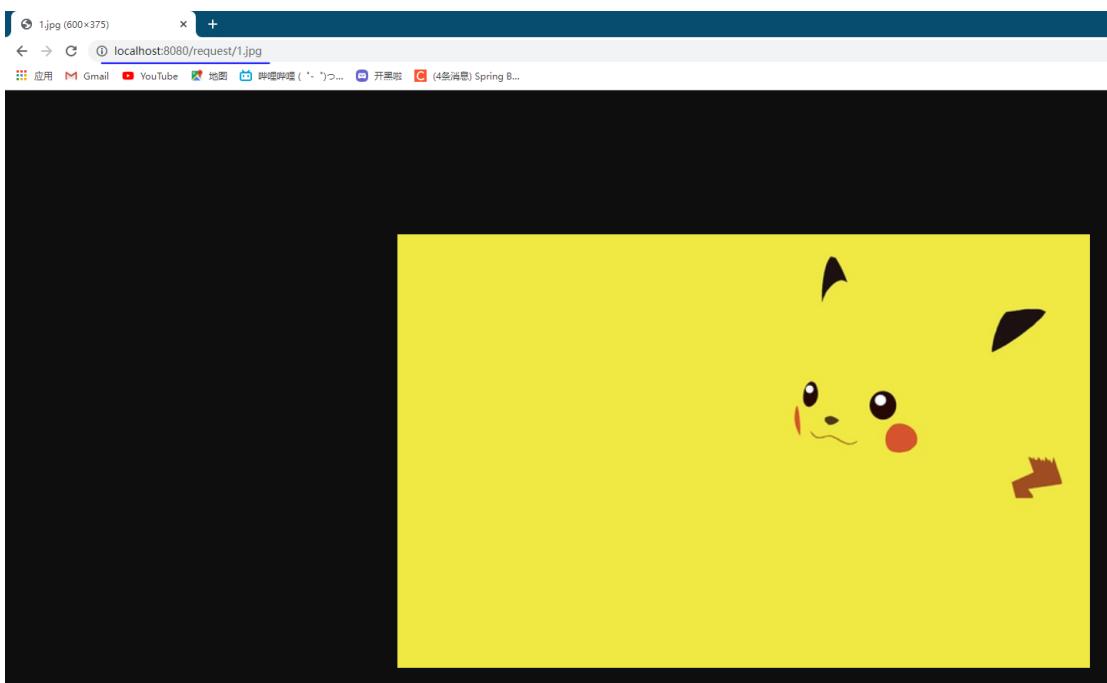


(三)、更改静态资源访问路径

在 application.properties 中添加
Spring.web.resources.static-locations=路径名



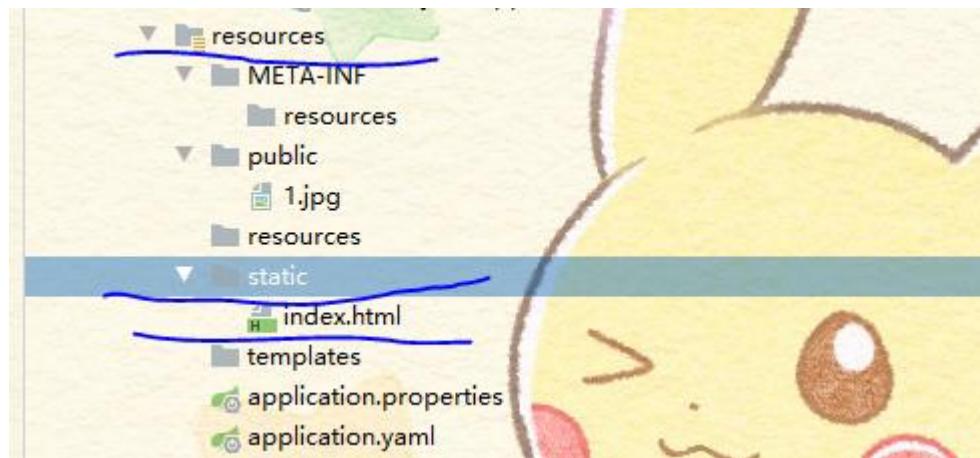
测试



七、欢迎页

(一)、静态资源模式

直接在几个静态资源目录中存放 index.html



然后进行网页访问



欢迎来到我的网页

(二)、控制层方法模式

在控制层类中添加 index 的映射,访问主目录的时候就会自动查找

(三)、小图标显示

在静态资源下添加 Favicon.ico 就会默认将网页小图标替换为 Favicon.ico 图标,没有网,凑和看,修改的就是这东西



八、PUT 和 DELETE 请求的处理

(一)、HTML 网页

```
<h1>欢迎来到我的网页</h1>
<form action="/user" method="get">
    <input type="submit" value="get 方式提交"/>
</form>
<form action="/user" method="post">
    <input type="submit" value="post 提交"/>
</form>
<form action="/user" method="post">
    <input type="hidden" name="_method" value="DELETE">
    <input type="submit" value="delete 提交"/>
</form>
<form action="/user" method="post">
    <input type="hidden" name="_method" value="put">
    <input type="submit" value="put 提交"/>
</form>
```

(二)、Controller

```
@org.springframework.stereotype.Controller  
@ResponseBody  
public class Controller {  
    @RequestMapping("2.jpg")  
    public String test1(){  
        return "aaa";  
    }  
    @RequestMapping(value = "/user",method = RequestMethod.GET)  
    public String getUser(){  
        return "GET-张三";  
    }  
    @RequestMapping(value = "/user",method = RequestMethod.POST)  
    public String saveUser(){  
        return "POST-张三";  
    }  
    @RequestMapping(value = "/user",method = RequestMethod.PUT)  
    public String putUser(){  
        return "PUT-张三";  
    }  
    @RequestMapping(value = "/user",method = RequestMethod.DELETE)  
    public String deleteUser(){  
        return "DELETE-张三";  
    }  
}
```

(三)、开启拦截器拦截请求更改提交方式

```
spring:  
  mvc:  
    hiddenmethod:  
      filter:  
        enabled: true
```

(四)、Rest 原理

- 表单提交会带上`_method=PUT`
- **请求过来被 HiddenHttpMethodFilter 拦截**
 - 请求是否正常， 并且是 POST
 - 获取到`_method` 的值。
 - 兼容以下请求： PUT. DELETE. PATCH
- 原生 `request (post)`， 包装模式 `RequestWrapper` 重写了 `getMethod` 方法， 返回的是传入的值。
- 过滤器链放行的时候用 `wrapper`。以后的方法调用 `getMethod` 是调用 `RequestWrapper` 的。

(五)、修改默认 Rest 请求 name

1. 创建一个 web 配置类



2. 添加注解,代理模式为 false

```
@Configuration(proxyBeanMethods = false)  
public class WebConfig {
```

3. 创建自己的 HiddenHttpMethodFilter

```
@Bean  
public HiddenHttpMethodFilter hiddenHttpMethodFilter(){  
    HiddenHttpMethodFilter methodFilter = new HiddenHttpMethodFilter();  
    methodFilter.setMethodParam("_met");      //返回自己识别的 name  
    return methodFilter;  
}
```

九、接受参数

@PathVariable(restful 风格获取参数)
@RequestHeader(获取请求头)
@RequestParam(获取请求参数 ? 的形式)

@CookieValue(获取 cookie 的值)
@RequestBody(获取请求体 非 get 请求)
@RequestAttribute(获取 request 域的值)
@MatrixVariable(矩阵变量)

(一)、接受简单的参数

```
@GetMapping("/person/{id}/zhangsan/{xuhao}")
public Map<String, Object> getPerson(@PathVariable("id") String id,
                                      @PathVariable("xuhao") Integer xuhao,
                                      @PathVariable Map<String, Object> parMap,
                                      @RequestHeader("Host") String hosts,
                                      @RequestHeader Map<String, Object> mapHead){

    Map<String, Object> map = new HashMap<>();
    map.put("id", id);
    map.put("xuhao", xuhao);
    map.put("parMap", parMap);
    map.put("Host", hosts);
    map.put("mapHead", mapHead);
    return map;
}
```

(二)、接受请求域的值

```
@RequestMapping("/textAtbt")
public String setRequestAtbt(HttpServletRequest request){
    request.setAttribute("name", "张三");
    request.setAttribute("age", 18);
    return "forward:success";
}
```

```
@ResponseBody  
 @RequestMapping("/success")  
 public Map<String, Object> getRequestAtbt(@RequestAttribute("name") String name,  
 @RequestAttribute("age") Integer age,  
 HttpServletRequest request){  
     Map<String, Object> map = new HashMap<>();  
     map.put("name", name);  
     map.put("age", age);  
     String name1 = request.getParameter("name");  
     System.out.println(name1);  
     return map;  
 }
```

(三)、接受矩阵变量

1、语法：

请求路径： /cars/sell;low=34;brand=byd,audi,yd

变量以;隔开,如果有多个值使用,隔开

2、开启矩阵变量功能

实现 `WebMvcConfigurer` 接口：手动开启：原理。对于路径的处理。

`UrlPathHelper` 的 `removeSemicolonContent` 设置为 `false`，让其支持矩阵变量的。

手动开启矩阵变量

```
@Configuration(proxyBeanMethods = false)  
 public class WebConfig implements WebMvcConfigurer {  
     @Override
```

```
public void configurePathMatch(PathMatchConfigurer configurer) {  
  
    UrlPathHelper urlPathHelper = new UrlPathHelper();  
  
    // 不移除；后面的内容。矩阵变量功能就可以生效  
  
    urlPathHelper.setRemoveSemicolonContent(false);  
  
    configurer.setUrlPathHelper(urlPathHelper);  
  
}  
}
```

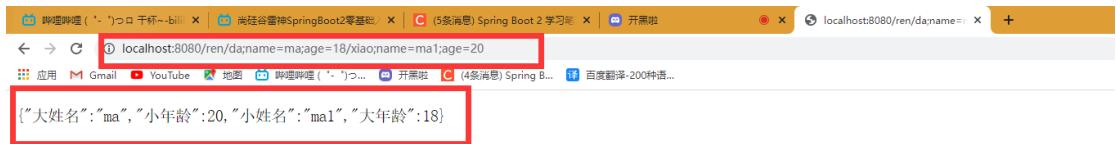
3、矩阵变量必须有 url 路径才能解析

4、案例：

映射器

```
@GetMapping("ren/{da}/{xiao}")  
public Map<String, Object> ren(@MatrixVariable(value = "name", pathVar = "da") String  
daName,  
                               @MatrixVariable(value = "name", pathVar = "xiao")String xiaoName,  
                               @MatrixVariable(value = "age",pathVar = "da")Integer daAge,  
                               @MatrixVariable(value = "age",pathVar = "xiao")Integer xiaoAge){  
  
    Map<String, Object> map = new HashMap<>();  
    map.put("大姓名",daName);  
    map.put("小姓名",xiaoName);  
    map.put("大年龄",daAge);  
    map.put("小年龄",xiaoAge);  
    return map;  
}
```

访问地址+结果



(四)、Map、Model 参数原理

Map 和 Model 的数据会存放在 request 请求与中

案例：

```
@org.springframework.stereotype.Controller
public class Controller {

    @GetMapping("/params")
    public String testParams(Map<String, Object> map,
                           Model model,
                           HttpServletRequest request,
                           HttpServletResponse response){

        map.put("putMap", "putMap");
        model.addAttribute("putModel", "putModel");
        request.setAttribute("putRequest", "putRequest");
        return "forward:/success";
    }

    @ResponseBody
    @GetMapping("/success")
    public Map<String, Object> success (@RequestAttribute(value = "putMap", required =
false) String msg,
```

```
        @RequestAttribute(value =
"putRequest",required = false)String myRequest,
        @RequestAttribute(value =
"putModel",required = false)String myModel,
HttpServletResponse request){

    Map<String, Object> map = new HashMap<>();
    map.put("getMap",msg);
    map.put("getRequest",myRequest);
    map.put("getModel",myModel);

    Object map1 = request.getAttribute("putMap");
    Object request1 = request.getAttribute("putRequest");
    Object model1 = request.getAttribute("putModel");
    map.put("map1",map1);
    map.put("request1",request1);
    map.put("model1",model1);
    return map;
}
```

(五)、使用自己的 JavaBean 接收参数并赋值

1、自定义 JavaBean

Person

```
@Data
public class Person {

    private String name;
    private Integer age;
```

```
private Pat pat;  
}
```

Pat

```
@Data  
public class Pat {  
    private String name;  
    private Integer age;  
}
```

2、HTML

```
<form action="/person" method="get">  
    姓名:<input type="text" value="张三" name="name">  
    年龄:<input type="text" value="15" name="age">  
    宠物名:<input type="text" value="小猫" name="pat.name">  
    宠物年龄:<input type="text" value="14" name="pat.age">  
    <input type="submit" value="提交">  
</form>
```

3、控制器

```
@ResponseBody  
@GetMapping("/person")  
public Person getPerson(Person person){  
    return person;  
}
```

(六)、使用逗号隔开参数接收(如:猫,15)

1、自定义 JavaBean

Person

```
@Data  
public class Person {  
  
    private String name;  
    private Integer age;  
    private Pat pat;  
}
```

Pat

```
@Data  
public class Pat {  
  
    private String name;  
    private Integer age;  
}
```

2、HTML

```
<form action="/person" method="get">  
    姓名:<input type="text" value="张三" name="name">  
    年龄:<input type="text" value="15" name="age">  
    宠物:<input type="text" value="小猫,15" name="pat">  
    <input type="submit" value="提交">  
</form>
```

3、开启分割参数设置

在 WebConfiguration 注解下的控制器中添加

```
@Bean
public WebMvcConfigurer webMvcConfigurer(){
    return new WebMvcConfigurer() {
        @Override
        public void addFormatters(FormatterRegistry registry) {
            registry.addConverter(new Converter<String, Pat>() {
                @Override
                public Pat convert(String source) {
                    if(!StringUtils.isEmpty(source)){
                        Pat pat = new Pat();
                        String [] split = source.split(",");
                        pat.setName(split[0]);
                        pat.setAge(Integer.valueOf(split[1]));
                        return pat;
                    }
                    return null;
                }
            });
        }
    };
}
```

4、控制器

```
@ResponseBody  
 @GetMapping("/person")  
 public Person getPerson(Person person){  
     return person;  
 }
```

十、响应数据

(一)、基于请求头的内容协商

1、导入支持返回 JSON 包

```
<dependency>  
  
    <groupId>org.springframework.boot</groupId>  
  
    <artifactId>spring-boot-starter-json</artifactId>  
  
    <version>2.3.4.RELEASE</version>  
  
    <scope>compile</scope>  
  
</dependency>
```

2、导入支持返回 XML 包

```
<dependency>  
  
    <groupId>com.fasterxml.jackson.dataformat</groupId>  
  
    <artifactId>jackson-dataformat-xml</artifactId>
```

```
</dependency>
```

3、控制层返回对象

```
@Controller

public class ResponseTestController {

    @ResponseBody //利用返回值处理器里面的消息转换器进行处理

    @GetMapping(value = "/test/person")

    public Person getPerson(){

        Person person = new Person();

        person.setAge(28);

        person.setBirth(new Date());

        person.setUserName("zhangsan");

        return person;

    }

}
```

1、内容协商

客户端发送数据的时候会发送请求头 `Accept`,告诉服务器自己能接受什么的类型(类型根据权重判断优先级),然后服务器根据权重向客户端发送数据,如果不支持大的权重会往下走,知道匹配到最佳的返回方式

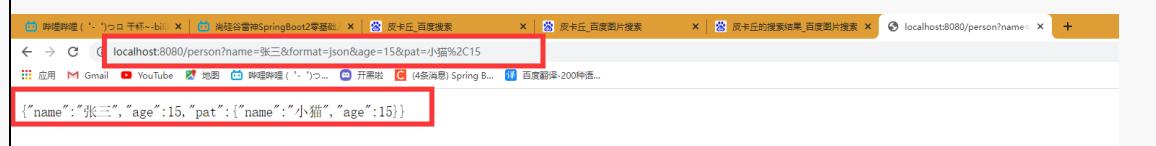
(二)、基于请求参数的内容协商

1、在 yaml 中添加配置

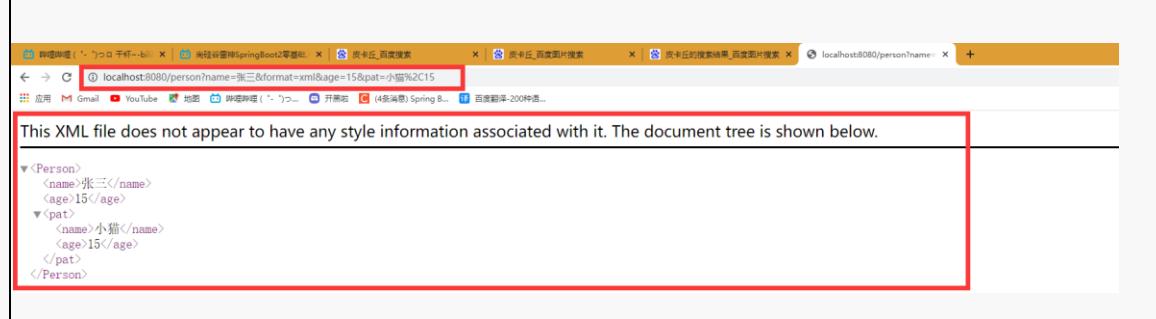
```
spring:  
  mvc:  
    contentnegotiation:  
      favor-parameter: true
```

2、请求参数中携带 format 请求=返回方式

<http://localhost:8080/test/person?format=json> //返回 JSON



<http://localhost:8080/test/person?format=xml> //返回 XML



(三)、自定义请求头 And 参数内容协商

1、创建自己的 MessageConverters

```
public class MyConverter implements HttpMessageConverter<Person> {

    //是否支持读
    @Override
    public boolean canRead(Class<?> clazz, MediaType mediaType) {
        return false;
    }

    //是否支持写
    @Override
    public boolean canWrite(Class<?> clazz, MediaType mediaType) {
        return clazz.isAssignableFrom(Person.class);
    }

    //返回支持的格式名
    @Override
    public List<MediaType> getSupportedMediaTypes() {
        return MediaType.parseMediaTypes("application/x-ma");
    }

    @Override
    public Person read(Class<? extends Person> clazz, HttpInputMessage inputMessage) throws
    IOException, HttpMessageNotReadableException {
        return null;
    }
}
```

```

//发送客户端数据形式

@Override
public void write(Person person, MediaType contentType, HttpOutputMessage outputMessage) throws
IOException, HttpResponseMessageNotWritableException {
    String date = person.getName() + ";" + person.getAge();
    OutputStream body = outputMessage.getBody();
    body.write(date.getBytes());
}
}

```

2、添加到底层 WebConfiguration 设置

```

@Configuration(proxyBeanMethods = false)
public class WebConfig implements WebMvcConfigurer {

    @Bean
    public WebMvcConfigurer webMvcConfigurer() {
        return new WebMvcConfigurer() {

            //覆盖底层的 MessageMyConverter

            @Override
            public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {

                Map<String, MediaType> mediaType = new HashMap<>();
                mediaType.put("json", MediaType.APPLICATION_JSON);

                //json 方式
                mediaType.put("xml", MediaType.APPLICATION_XML);

                //xml 方式
                mediaType.put("aaa", MediaType.parseMediaType("application/x-ma"));           //将自己的装入底层
                ParameterContentNegotiationStrategy parameterContentNegotiationStrategy = new

```

```
ParameterContentNegotiationStrategy(mediaType);           //参数形式

        HeaderContentNegotiationStrategy headerContentNegotiationStrategy = new
HeaderContentNegotiationStrategy();           //请求头方式

        configurer.strategies(Arrays.asList(parameterContentNegotiationStrategy,
headerContentNegotiationStrategy));

    }

//参数可以以, 分开赋值

@Override

public void addFormatters(FormatterRegistry registry) {

    registry.addConverter(new Converter<String, Pat>() {

        @Override

        public Pat convert(String source) {

            if (!StringUtils.isEmpty(source)) {

                Pat pat = new Pat();

                String[] split = source.split(",");
                pat.setName(split[0]);
                pat.setAge(Integer.valueOf(split[1]));
                return pat;
            }
            return null;
        }
    });
}

//添加自己的方法

@Override

public void extendMessageConverters(List<HttpMessageConverter<?>> converters) {
    converters.add(new MyConverter());
}
```

```
        }
    );
}
}
```

十一、Thymeleaf

(一)、简单实用

1、基础语法

表达式名字	语法	用途
变量取值	<code>\${...}</code>	获取请求域、 <code>session</code> 域、对象等值
选择变量	<code>*{...}</code>	获取上下文对象值
消息	<code>#{...}</code>	获取国际化等值
链接	<code>@{...}</code>	生成链接
片段表达式	<code>~{...}</code>	<code>jsp:include</code> 作用，引入公共页面片段

2、设置属性值-th:attr

- 设置单个值

```
<input type="submit" value="Subscribe!" th:attr="value=#{"subscribe.submit}" />
```

- 设置多个值

```

```

3、循环

```
<tr th:each="prod : ${prods}">  
  
    <td th:text="${prod.name}">Onions</td>  
  
    <td th:text="${prod.price}">2.41</td>  
  
    <td th:text="${prod.inStock} ? #{true} : #{false}">yes</td>  
  
</tr>
```

4、条件

If

```
<a href="comments.html"  
  
th:href="@{/product/comments(prodId=${prod.id})}"  
  
th:if="${not #lists.isEmpty(prod.comments)}">view</a>
```

switch

```
<div th:switch="${user.role}">  
  
    <p th:case="admin">User is an administrator</p>  
  
    <p th:case="#{roles.manager}">User is a manager</p>  
  
    <p th:case="*"/>User is some other thing</p>  
  
</div>
```

(二)、Thymeleaf 初体验

1、导入 Thymeleaf 依赖

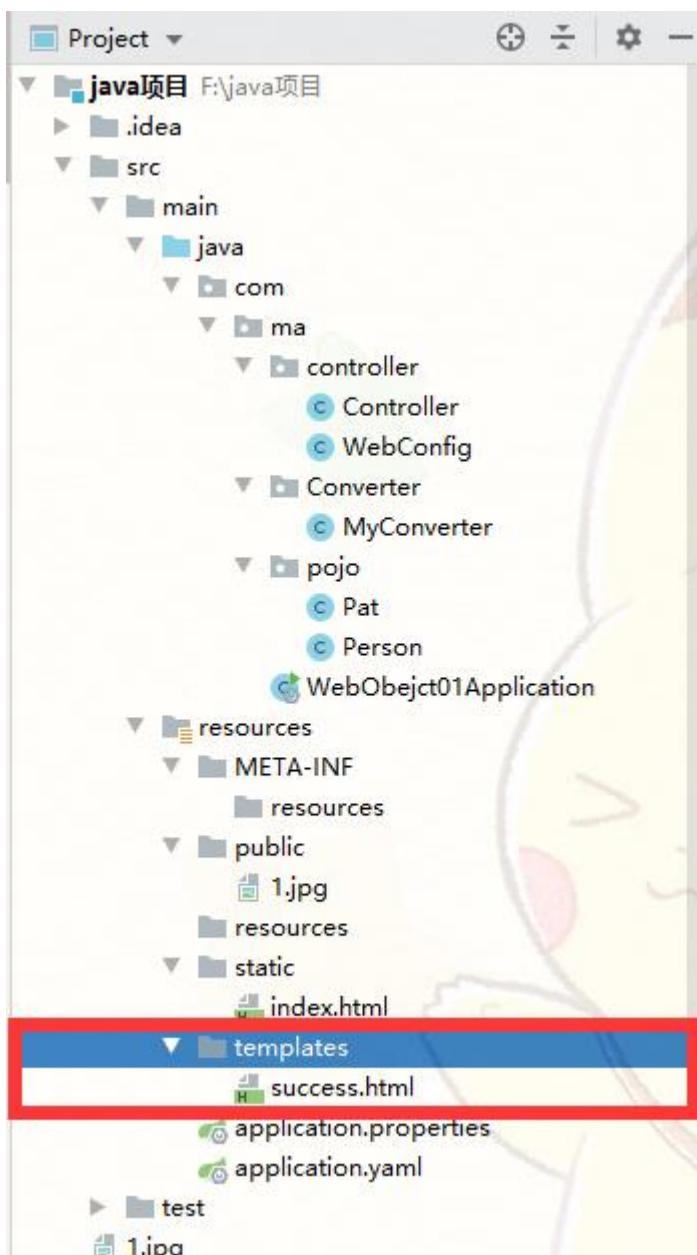
```
<dependency>  
  
    <groupId>org.springframework.boot</groupId>  
  
    <artifactId>spring-boot-starter-thymeleaf</artifactId>  
  
</dependency>
```

2、只需要根据底层的设置创建网页即可

底层

```
public static final String DEFAULT_PREFIX = "classpath:/templates/";           //模板放置处  
  
public static final String DEFAULT_SUFFIX = ".html";                           //文件的后缀名
```

创建网页文件



2、控制层链接

```
@org.springframework.stereotype.Controller  
public class Controller {  
  
    @GetMapping("/testThymeleaf")
```

```
public String testThymeleaf(Model model){  
    model.addAttribute("msg","欢迎使用 Thymeleaf");  
    model.addAttribute("url","www.fanyi.com");  
    return "success";  
}
```

4、网页代码

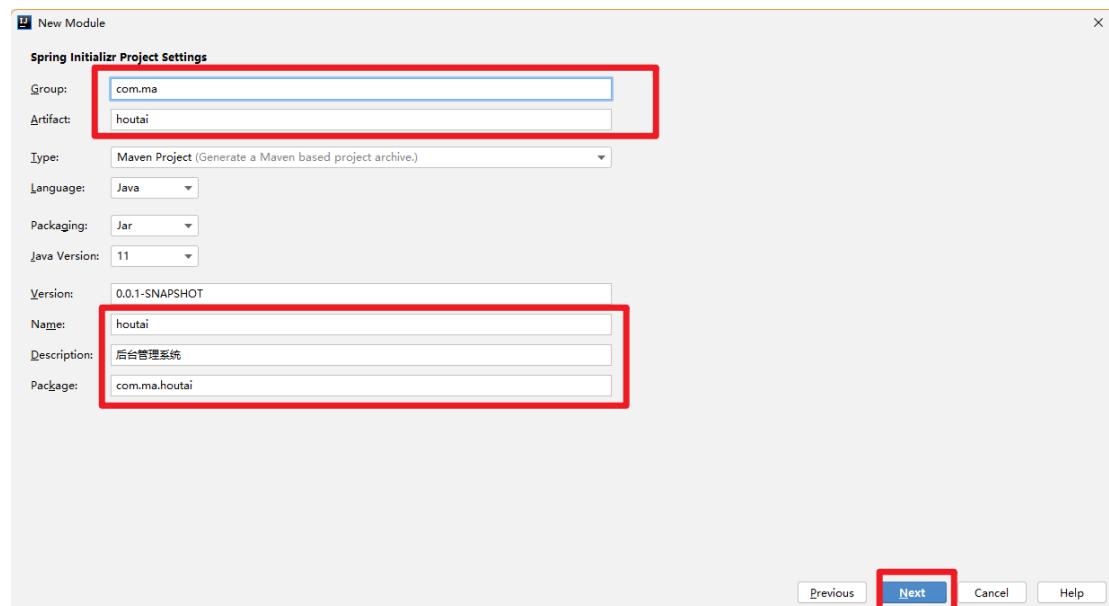
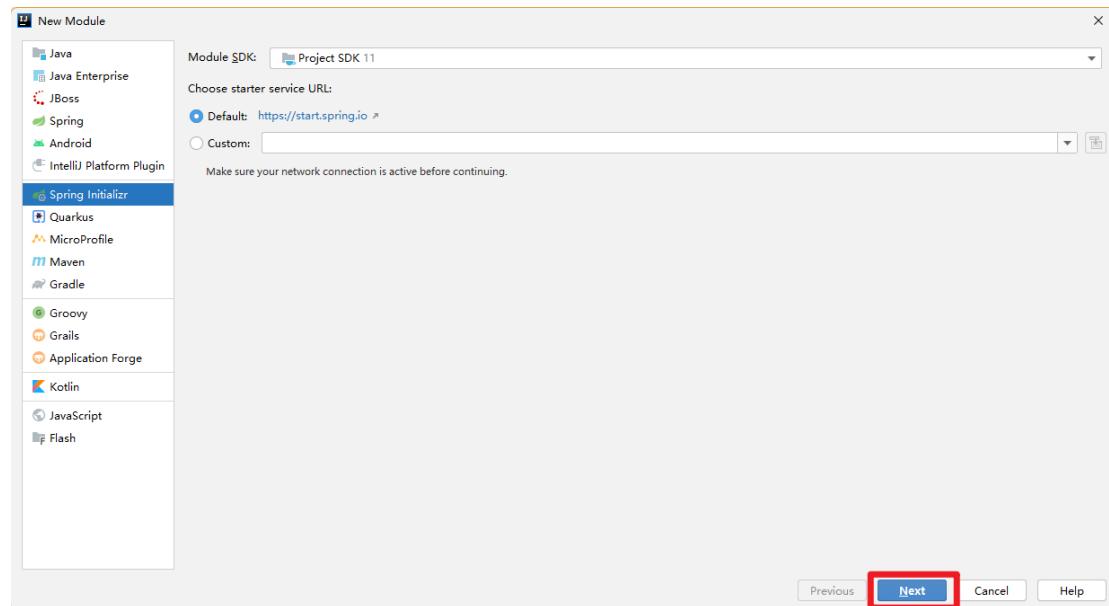
```
<body>  
<h1 th:text="${msg}">Hello World</h1>  
  
<a th:href="@{url}">www.baidu.com</a>  
</body>
```

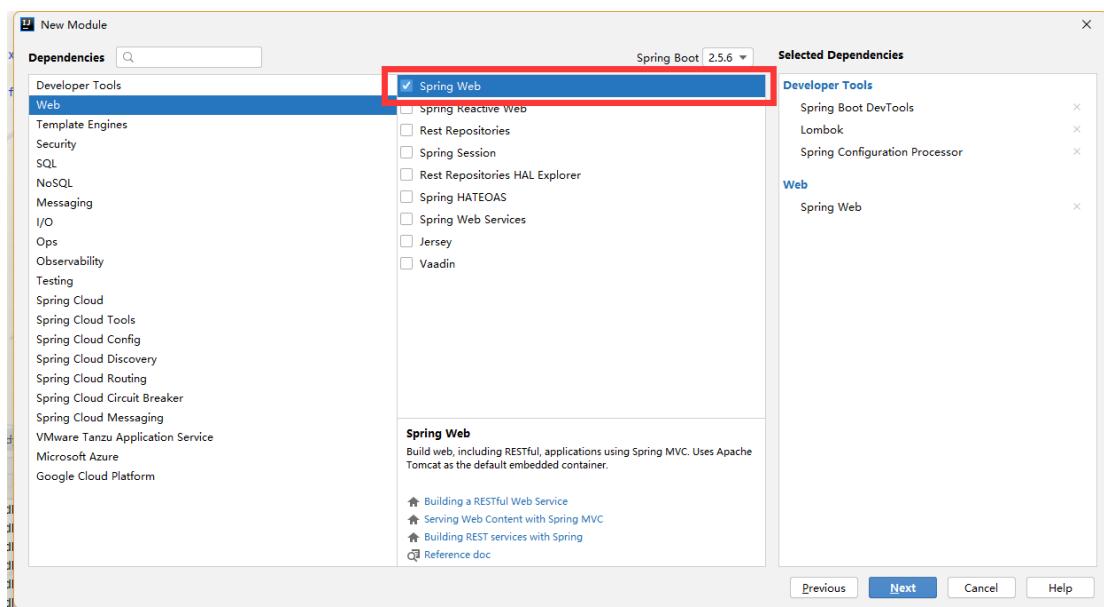
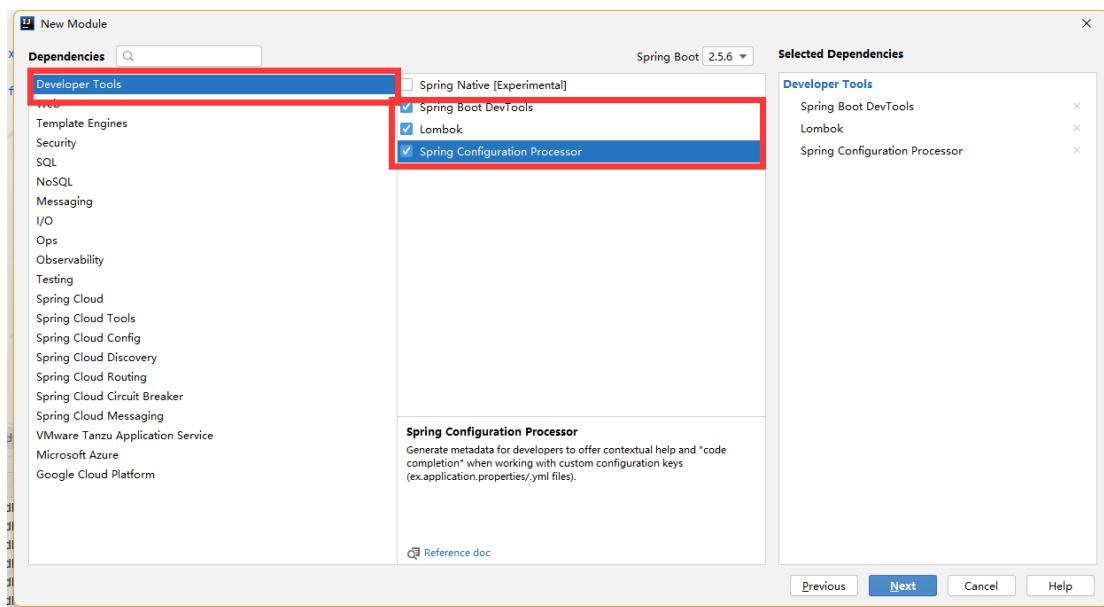
十二、后台管理系统

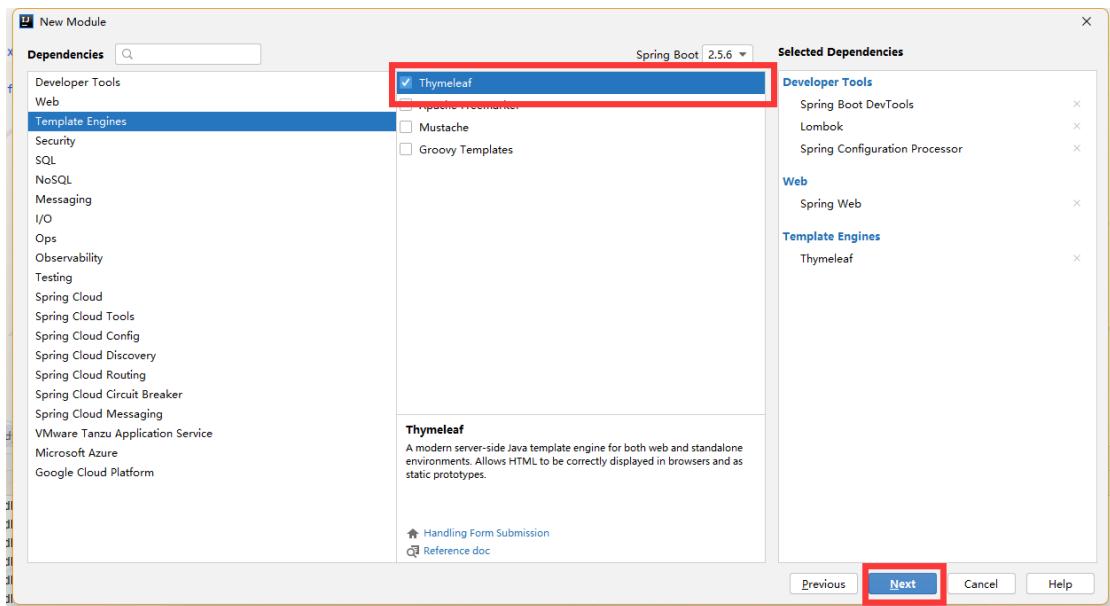


AdminEx - 响应式设计后台管理模版@www.java1234.com.zip

(一) 创建项目





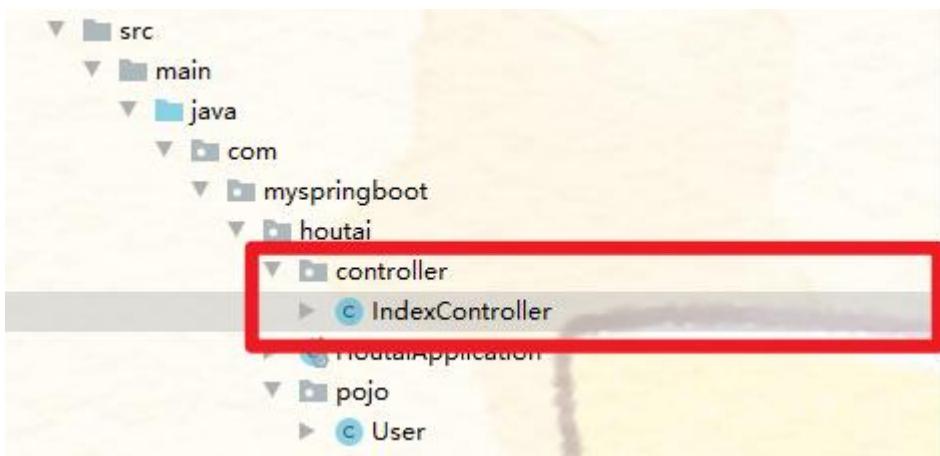


(二)、登录和主页名字配置

1、复制 login.html 和 index.html 到 templates 目录, index.html 改为 main.html



2、Controller 设置



3、Controller 代码

```
@Controller
public class IndexController {

    //设置访问目录,如果是/或/login 跳转到login 目录
    @GetMapping(value = {"/", "/login"})
    public String indexPage(){
        return "login";
    }

    //登录时判断是否登录成功,成功重定向主页,防止不断提交,不成功返回并提示
    @PostMapping("/login")
    public String mainPage(User user, Model model, HttpServletRequest request){
        if(StringUtils.hasLength(user.getUserName()) && "123456".equals(user.getPassWord())){
            request.getSession().setAttribute("loginUser", user);
            return "redirect:/main.html";
        }else {
            model.addAttribute("loginMsg", "登录失败");
            return "login";
        }
    }
}
```

```
    }

}

//判断是否登录过,登陆过跳转,没登录跳转到登录页面

@GetMapping(value ="main.html")

public String mainPage(HttpServletRequest request){

    Object loginUser = request.getSession().getAttribute("loginUser");

    if(loginUser != null){

        return "main";

    }else {

        request.setAttribute("loginMsg","请先登录");

        return "login";

    }

}

}
```

4、登录表单提交设置

```
<form class="form-signin" action="/login" method="post">
```

5、修改主页为登录的用户名

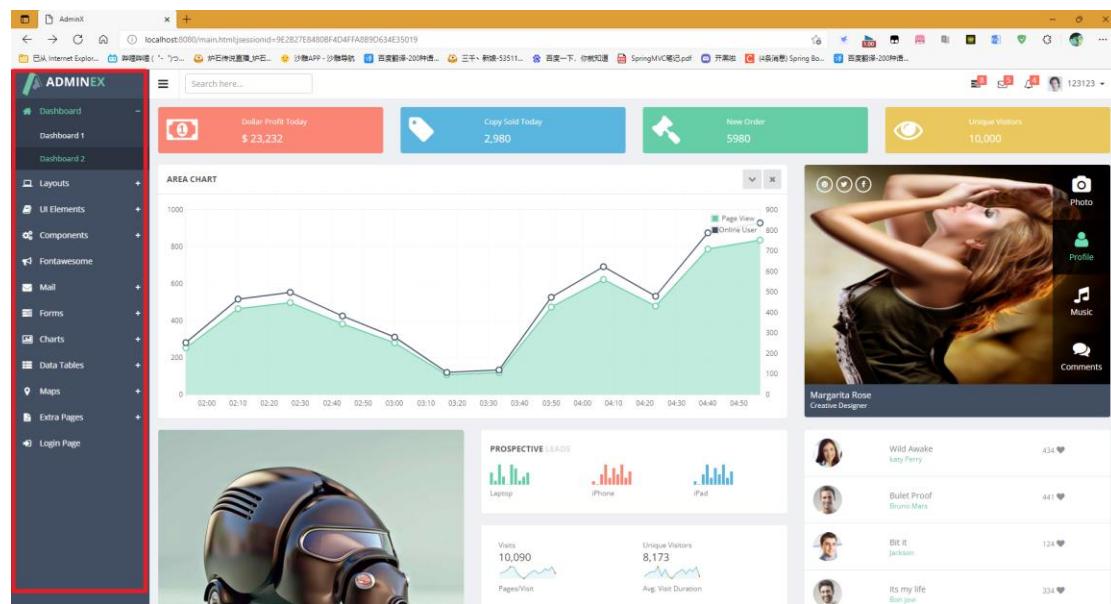
```


[[${session.loginUser.userName}]]

<span class="caret"></span>
```

(三)、抽取公共页面

1、抽取左菜单



The screenshot shows a web-based admin dashboard titled "ADMINEX". On the left, there is a dark sidebar menu with various options like Dashboard, Layouts, UI Elements, Components, Charts, Data Tables, Maps, Extra Pages, and Login Page. A red box highlights this sidebar area. The main content area features several cards: "Dollar Profit Today" (\$23,232), "Copy Sold Today" (2,980), "New Order" (5980), and "Unique Visitors" (10,000). Below these are two charts: an "AREA CHART" showing visitor trends from 02:00 to 04:50, and a "PROSPECTIVE LEADS" chart for Laptop, iPhone, and iPad. To the right, there's a profile section for "Margarita Rose" (Circuit Designer) with a photo, and a list of users with their activity metrics: Wild Awake (lady Perry), Bulet Proof (Bruno Mars), Bit R (Jackson), and Its my life (Bon jovi).

```
<link href="css/style.css" th:href="@{css/style.css}" rel="stylesheet">
<link th:href="@{css/style-responsive.css}" href="css/style-responsive.css" rel="stylesheet">

<!-- HTML5 shim and Respond.js support of HTML5 elements and media queries -->
<!--[if lt IE 9]>
<script th:src="@{js/html5shiv.js}"></script>
<script th:src="@{js/respond.min.js}"></script>
<![endif]-->

</head>
<body>

<!-- left side start--&gt;
&lt;div class="left-side sticky-left-side" id="leftCommon" ...&gt;
&lt;!-- 左侧结束--&gt;

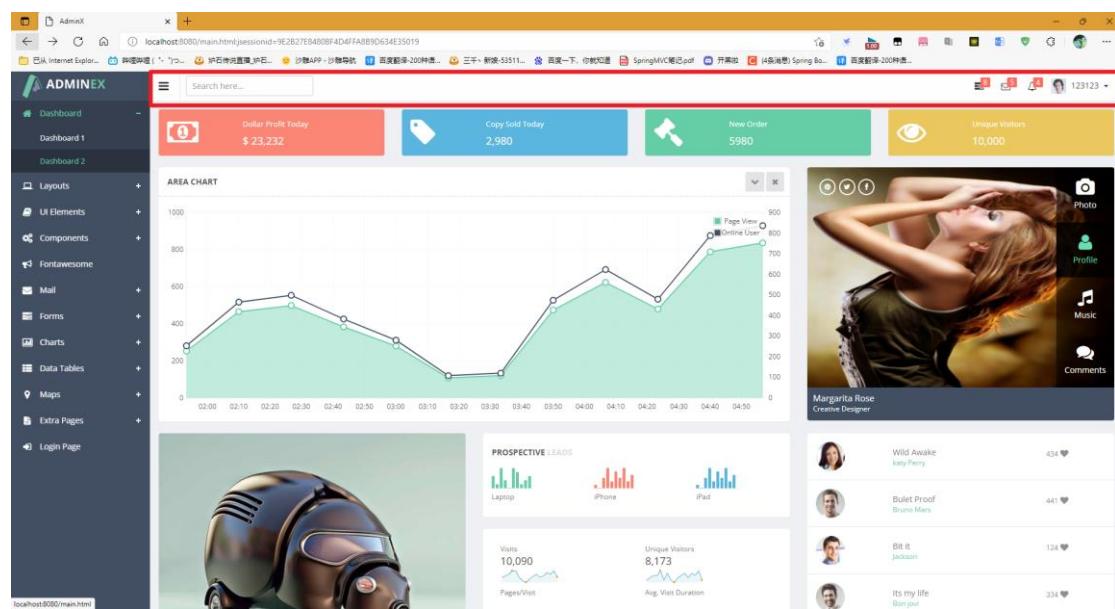
<!-- 头部分开始--&gt;
&lt;div class="header-section" id="header"&gt;

  <!-- 开关按钮开始--&gt;
  &lt;a class="toggle-btn"&gt;
    | &lt;i class="fa fa-bars"&gt;&lt;/i&gt;
  &lt;/a&gt;
  &lt;!-- 开关按钮结束--&gt;

  &lt;!-- 头部右侧内容 --&gt;
  &lt;div class="header-right"&gt;
    &lt;span&gt;123123 &lt;/span&gt;
    &lt;span&gt;123123 &lt;/span&gt;
  &lt;/div&gt;
&lt;/div&gt;

</pre>
```

2、抽取个人信息栏



The screenshot shows a web-based administration interface titled "AdminX". On the left is a dark sidebar menu with items like "Dashboard", "Dashboard 1", "Dashboard 2", "Layouts", "UI Elements", "Components", "Fontawesome", "Mail", "Forms", "Charts", "Data Tables", "Maps", "Extra Pages", and "Login Page". The main content area features a top navigation bar with links for "Dollar Profit Today" (\$23,232), "Copy Sold Today" (2,980), "New Order" (5980), and "Unique Visitors" (10,000). Below this is a large area chart showing data over time from 02:10 to 04:50. To the right of the chart is a profile section for "Margarita Rose" (Creative Designer) featuring a photo, profile, music, and comments sections. At the bottom of the page is a code editor window displaying the HTML and CSS code for the header section, which is highlighted with a red box.

```
<link href="css/style.css" th:href="@{css/style.css}" rel="stylesheet">
<link th:href="@{css/style-responsive.css}" href="css/style-responsive.css" rel="stylesheet">


<!--[if lt IE 9]>
<script th:src="@{js/html5shiv.js}"></script>
<script th:src="@{js/respond.min.js}"></script>
<!--[endif]-->

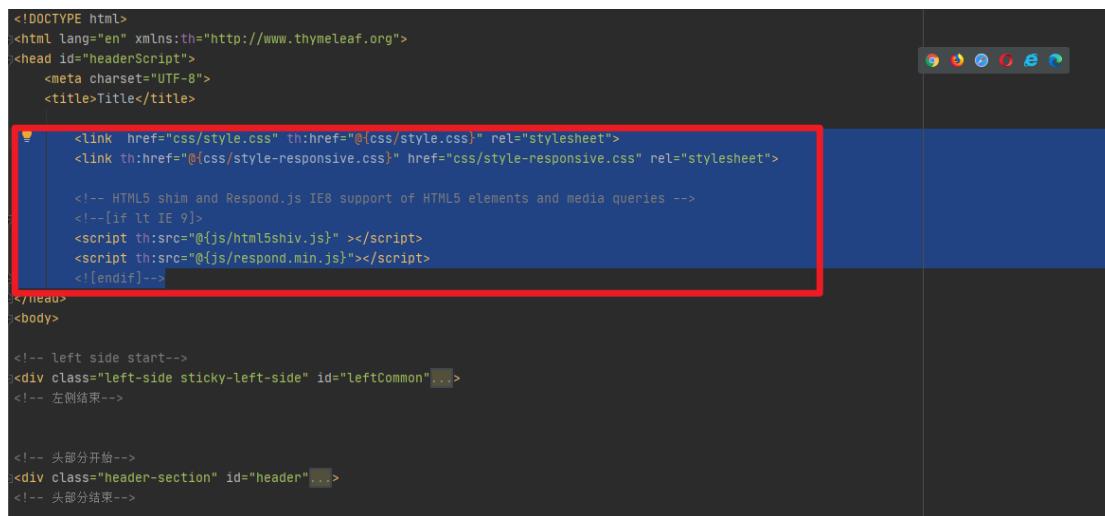
</head>
<body>

<!-- left side start--&gt;
&lt;div class="left-side sticky-left-side" id="leftCommon" ...&gt;
&lt;!-- 左侧结束--&gt;

<!-- 头部分开始--&gt;
&lt;div class="header-section" id="header" ...&gt;
&lt;!-- 头部分结束--&gt;

&lt;div id="dibuScript"&gt;
    &lt;!-- Placed js at the end of the document so the pages load faster --&gt;
    &lt;script th:src="@{js/jquery-1.10.2.min.js}"&gt;&lt;/script&gt;
    &lt;script th:src="@{js/jquery-ui-1.9.2.custom.min.js}"&gt;&lt;/script&gt;
    &lt;script th:src="@{js/jquery-migrate-1.2.1.min.js}"&gt;&lt;/script&gt;
    &lt;script th:src="@{js/bootstrap.min.js}"&gt;&lt;/script&gt;
&lt;/div&gt;</pre>
```

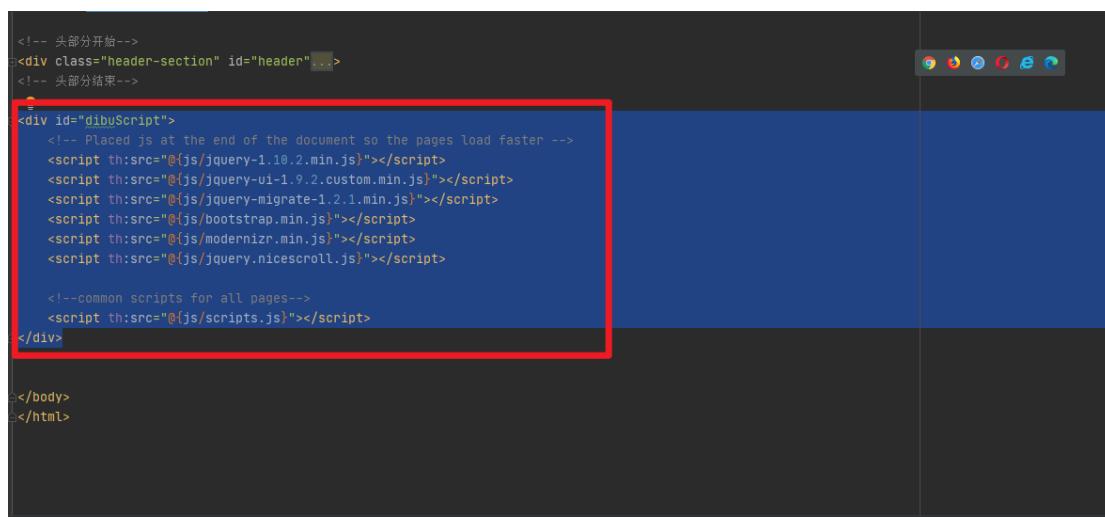
3、抽取头部 js.css 导入



```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head id="headerScript">
    <meta charset="UTF-8">
    <title>Title</title>
    <link href="css/style.css" th:href="@{css/style.css}" rel="stylesheet">
    <link th:href="@{css/style-responsive.css}" href="css/style-responsive.css" rel="stylesheet">
    <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media queries -->
    <!--[if lt IE 9]>
        <script th:src="@{js/html5shiv.js}"></script>
        <script th:src="@{js/respond.min.js}"></script>
    <![endif]-->
</head>
<body>
    <!-- left side start-->
    <div class="left-side sticky-left-side" id="leftCommon" ...>
    <!-- 左侧结束-->

    <!-- 头部分开始-->
    <div class="header-section" id="header" ...>
    <!-- 头部分结束-->
```

4、抽取底部 js.css 导入



```
<!-- 头部分开始-->
<div class="header-section" id="header" ...>
<!-- 头部分结束-->
<div id="dibuScript">
    <!-- Placed js at the end of the document so the pages load faster -->
    <script th:src="@{js/jquery-1.10.2.min.js}"></script>
    <script th:src="@{js/jquery-ui-1.9.2.custom.min.js}"></script>
    <script th:src="@{js/jquery-migrate-1.2.1.min.js}"></script>
    <script th:src="@{js/bootstrap.min.js}"></script>
    <script th:src="@{js/modernizr.min.js}"></script>
    <script th:src="@{js/jquery.nicescroll.js}"></script>
    <!--common scripts for all pages-->
    <script th:src="@{js/scripts.js}"></script>
</div>

</body>
</html>
```

5、网页加入抽取部分

```
<link rel="shortcut icon" href="#" type="image/png">

<title>Basic Table</title>

<div th:replace="common :: #headerScript"></div>
</head>

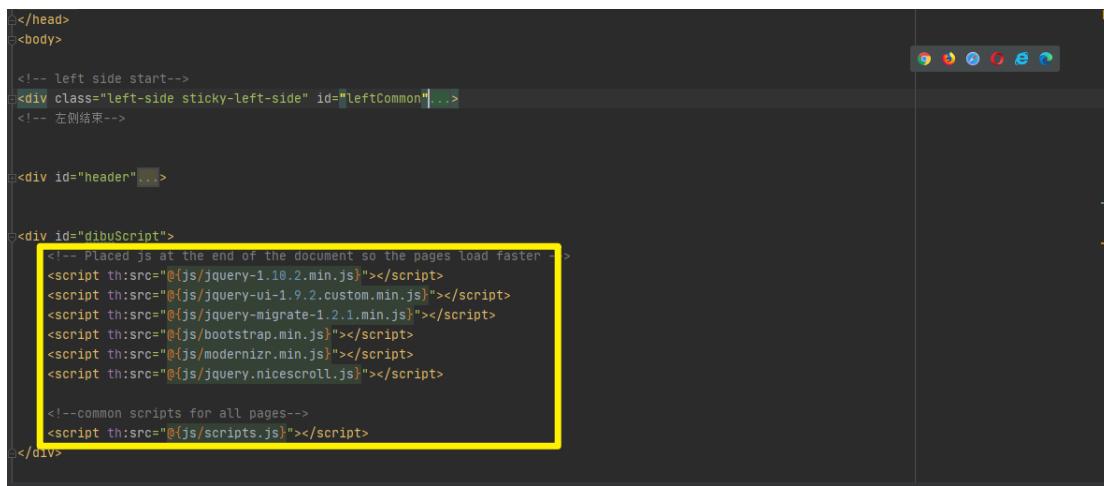
<body class="sticky-header">
    <section>

        <div th:replace="common :: #leftCommon"></div>

        <!-- main content start-->
        <div class="main-content">...
        <!-- main content end--> </section>

        <div th:include="common :: #dibuScript"></div>
    </body>
</html>
```

6、修改所有连接为类路径(根据需求改)



```
</head>
<body>

<!-- left side start-->
<div class="left-side sticky-left-side" id="leftCommon">...
<!-- 左侧结束-->

<div id="header">...

<div id="dibuScript">
    <!-- Placed js at the end of the document so the pages load faster -->
    <script th:src="@{js/jquery-1.10.2.min.js}"></script>
    <script th:src="@{js/jquery-ui-1.9.2.custom.min.js}"></script>
    <script th:src="@{js/jquery-migrate-1.2.1.min.js}"></script>
    <script th:src="@{js/bootstrap.min.js}"></script>
    <script th:src="@{js/modernizr.min.js}"></script>
    <script th:src="@{js/jquery.nicescroll.js}"></script>

    <!--common scripts for all pages-->
    <script th:src="@{js/scripts.js}"></script>
</div>
```

(四)、遍历网页数据

1、装入域数据

```
@GetMapping("/dynamic_table")
public String dynamic_table(Model model){
    List<User> users = Arrays.asList(new User("zhangsan","123456"),
                                    new User("lisi","222333"),
                                    new User("haha","333222"),
                                    new User("hehe","222222"));

    model.addAttribute("users",users);
    return "table/dynamic_table";
}
```

2、网页遍历域数据

```
<tr class="gradeX" th:each="user,stats : ${users} ">
    <td th:text="${ stats.count }" >Win 95+</td>
    <td th:text="${ user.getUserName() }" ></td>
```

```
<td th:text="${user.getPassWord()}"></td>  
</tr>
```

(五)、拦截器

1、创建一个拦截器类(实现 HandlerInterceptor)

```
@Slf4j  
public class LoginInterceptor implements HandlerInterceptor {  
  
    //访问数据处理业务之前  
  
    @Override  
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)  
throws Exception {  
  
    //获得 session  
    HttpSession session = request.getSession();  
    //取出 session 中用户的值  
    Object loginUser = session.getAttribute("loginUser");  
    //判断用户是否为空  
    if(loginUser != null){  
        //不为空放行  
        return true;  
    }  
  
    //为空添加域信息返回给客户端并显示  
    request.setAttribute("loginMsg","请先登录");  
    //转发到登录页面  
    request.getRequestDispatcher("/").forward(request,response);  
    return false;  
}
```

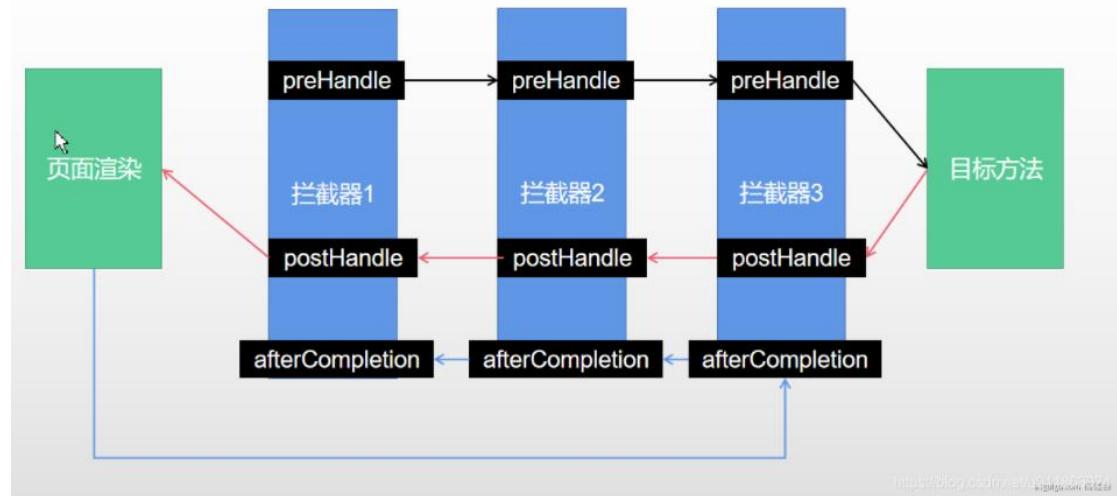
```
//访问数据处理业务之后  
@Override  
public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,  
ModelAndView modelAndView) throws Exception {  
  
}  
  
//视图渲染之后  
@Override  
public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object  
handler, Exception ex) throws Exception {  
  
}  
}
```

2、将自定义拦截器添加到容器中

```
@Configuration  
public class LoginInterceptorConfig implements WebMvcConfigurer {  
  
    @Override  
    public void addInterceptors(InterceptorRegistry registry) {  
        //添加的自定义拦截器  
        registry.addInterceptor(new LoginInterceptor())  
            //拦截什么规则  
            .addPathPatterns("/**")  
            //排除规则  
            .excludePathPatterns("/*,/login,/css/**,/fonts/**,/images/**,/js/**");  
    }  
}
```

3、拦截器机制原理

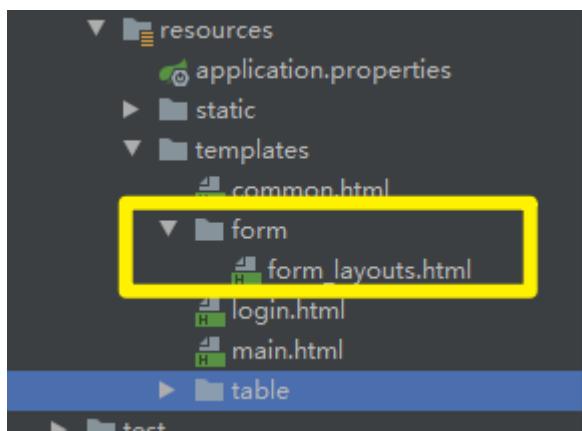
1. 根据当前请求，找到 HandlerExecutionChain（可以处理请求的 handler 以及 handler 的所有 拦截器）
2. 先来顺序执行 所有拦截器的 preHandle()方法。
3. 如果当前拦截器 preHandle()返回为 true。则执行下一个拦截器的 preHandle()
 - a) 如果当前拦截器返回为 false。直接倒序执行所有已经执行了的拦截器的 afterCompletion();
 - b) 如果任何一个拦截器返回 false，直接跳出不执行目标方法。
 - c) 所有拦截器都返回 true，才执行目标方法。
4. 倒序执行所有拦截器的 postHandle()方法。
5. 前面的步骤有任何异常都会直接倒序触发 afterCompletion()。
6. 页面成功渲染完成以后，也会倒序触发 afterCompletion()。



假如第二个 preHandle 错误,就不会执行第三个的 preHandle,直接执行第二个的 afterCompletion,再执行第一个 afterCompletion。

十三、文件上传

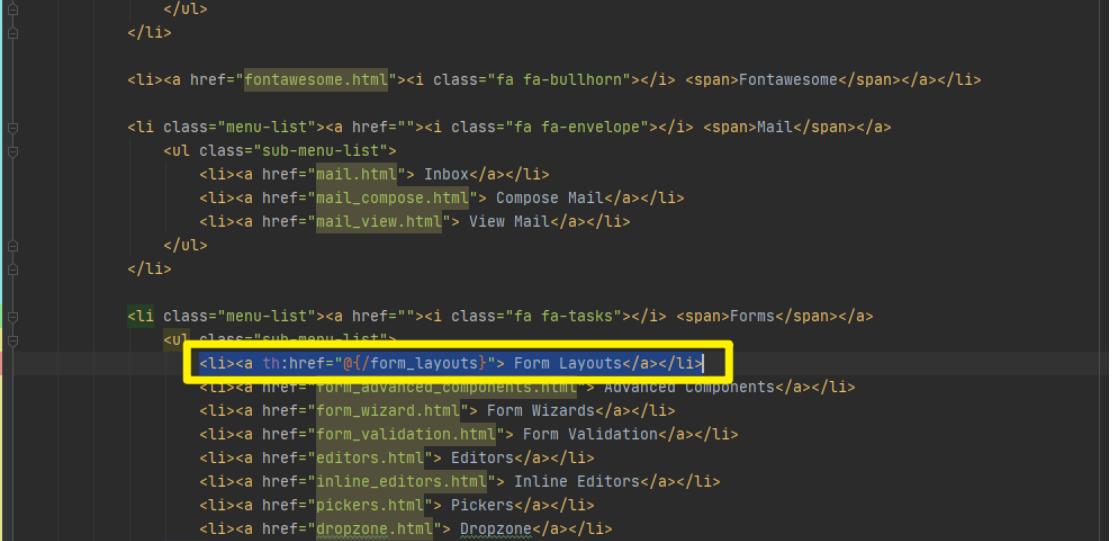
(一)、复制网页进 IDEA



(二)、添加访问的 Controller

```
@GetMapping("/form_layouts")
public String form_layouts(){
    return "form/form_layouts";
}
```

(三)、修改公共页的跳转路径



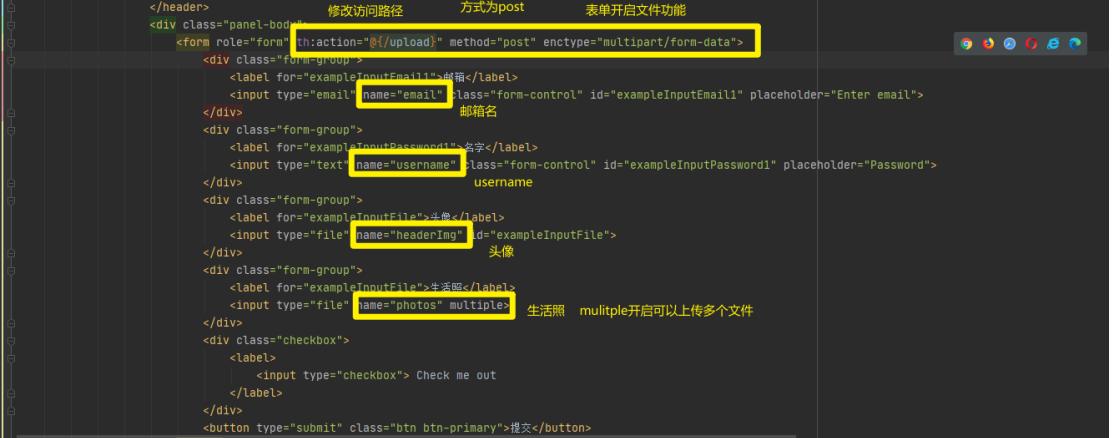
```
</ul>
</li>

<li><a href="#"> Fontawesome</a></li>

<li class="menu-list"><a href="#"> Mail</a>
    <ul class="sub-menu-list">
        <li><a href="#"> Inbox</a></li>
        <li><a href="#"> Compose Mail</a></li>
        <li><a href="#"> View Mail</a></li>
    </ul>
</li>

<li class="menu-list"><a href="#"> Forms</a>
    <ul class="sub-menu-list">
        <li><a th:href="@{/form_layouts}"> Form Layouts</a></li> (highlighted)
        <li><a href="#"> Advanced Components</a></li>
        <li><a href="#"> Form Wizards</a></li>
        <li><a href="#"> Form Validation</a></li>
        <li><a href="#"> Editors</a></li>
        <li><a href="#"> Inline Editors</a></li>
        <li><a href="#"> Pickers</a></li>
        <li><a href="#"> Dropzone</a></li>
    </ul>
</li>
```

(四)、修改表单提交为上传表单



```
</header>
<div class="panel-body" style="padding: 10px; border: 1px solid #ccc; border-radius: 5px; background-color: #fff; margin-bottom: 10px;">
    修改访问路径      方式为post      表单开启文件功能
    <form role="form" th:action="@{/upload}" method="post" enctype="multipart/form-data">
        <div class="form-group">
            <label for="exampleInputEmail1">邮箱</label>
            <input type="email" name="email" class="form-control" id="exampleInputEmail1" placeholder="Enter email">
            邮箱名
        </div>
        <div class="form-group">
            <label for="exampleInputPassword1">名字</label>
            <input type="text" name="username" class="form-control" id="exampleInputPassword1" placeholder="Password">
            username
        </div>
        <div class="form-group">
            <label for="exampleInputFile1">头像</label>
            <input type="file" name="headerImg" id="exampleInputFile1">
            头像
        </div>
        <div class="form-group">
            <label for="exampleInputFile2">生活照</label>
            <input type="file" name="photos" multiple="multiple" id="exampleInputFile2">
            生活照 multiple开启可以上传多个文件
        </div>
        <div class="checkbox">
            <label>
                <input type="checkbox"> Check me out
            </label>
        </div>
        <button type="submit" class="btn btn-primary">提交</button>
    </form>

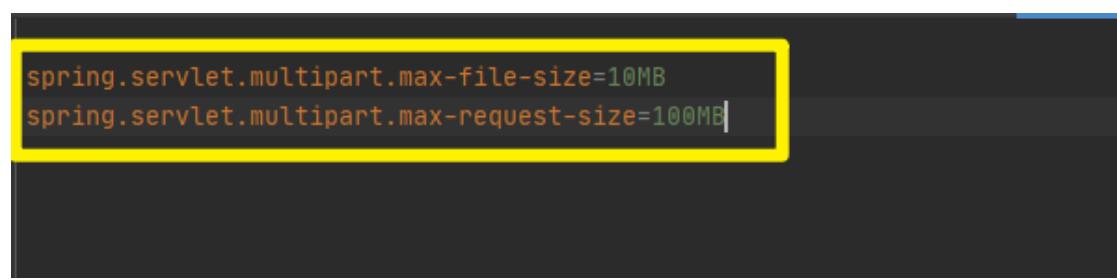
```

(五)、Controller 接受文件

```
@PostMapping("/upload")
public String requestFile(@RequestParam("email") String email,
                           @RequestParam("username") String username,
                           @RequestPart("headerImg") MultipartFile headerImg,
                           @RequestPart("photos") MultipartFile[] photos) throws IOException {
    //打印接受的参数信息
}
```

```
log.info("上传的信息: email={}, username={}, headerImg={}, photos={}",  
        email,username,headerImg.getSize(),photos.length);  
  
//判断是否为空  
  
if(!headerImg.isEmpty()) {  
    //获取上传的文件名  
  
    String originalFilename = headerImg.getOriginalFilename();  
  
    //储存到本地路径  
  
    headerImg.transferTo(new File("F:\\SpringBootFileTest\\\\"+originalFilename));  
}  
  
if(photos.length>0){  
    for (MultipartFile photo : photos) {  
        if(!photo.isEmpty()){  
            String originalFilename = photo.getOriginalFilename();  
  
            photo.transferTo(new File("F:\\SpringBootFileTest\\\\shengh\\\\"+originalFilename));  
        }  
    }  
}  
  
return "redirect:/form_layouts";  
}
```

(六)、设置 SpringBoot 接受文件的大小



```
spring.servlet.multipart.max-file-size=10MB  
spring.servlet.multipart.max-request-size=100MB
```

十四、异常处理

(一)、SpringBoot 默认机制

在 templates 下创建/error 放入错误页 4xx,5xx,底层自动处理 400,500 的错误跳转到 4xx,5xx 的错误页

(二)、SpringBoot 返回数据的方式

如果客户端使用浏览器发生错误,返回 SpringBoot 默认的错误页,如果客户端是机器访问,自动返回 json 数据

```
{  
    "timestamp": "2020-11-22T05:53:28.416+00:00",  
    "status": 404,  
    "error": "Not Found",  
    "message": "No message available",  
    "path": "/asadada"  
}
```

(三)、自定义异常处理

1.1、类头加@ControllerAdvice,方法头加@ExceptionHandler
@ExceptionHandler({添加预想的异常})

当服务器出现预想的异常的时候就会自动跳转到此方法处理

```
@ControllerAdvice  
  
public class GlobalExceptionHandler {  
  
    //当发生计算错误和空指针错误的时候抛出此异常  
  
    @ExceptionHandler({ArithException.class, NullPointerException.class})  
  
    public String handleArithException(){  
        //发生异常跳转到登录页面  
    }  
}
```

```
        return "login";
    }
}
```

1.2、自定义抛出的异常

类头添加@ResponseStatus 注解

```
//状态码异常注解  value = 状态码,报错信息
@ResponseStatus(value = HttpStatus.FORBIDDEN,reason = "用户数量过多")
public class UserTooManyException extends RuntimeException{

    public UserTooManyException(){}
}
```

当用户超过 3 个的时候抛出自定义异常

```
@GetMapping("/dynamic_table")
public String dynamic_table(Model model) {
    List<User> users = Arrays.asList(new User("zhangsan", "123456"),
        new User("lisi", "222333"),
        new User("haha", "333222"),
        new User("hehe", "222222"));
    model.addAttribute("users", users);

    if(users.size()>3){
        throw new UserTooManyException();
    }
    return "table/dynamic_table";
}
```

1.3、定义自己的异常状态码和信息

当发生错误的时候,由于优先级的问题,首先使用这个异常,这个中的方法返回 **ModelAndView** 的时候异常不再循环,直接返回方法中定义的错误

```
//优先级最小    越小越优先
@Order(value = Ordered.HIGHEST_PRECEDENCE)
```

```
//添加到容器中

@Component

public class CustomerHandlerExceptionResolver implements HandlerExceptionResolver {

    @Override

    public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response,
Object handler, Exception ex) {

        try {

            //定义状态码,错误信息
            response.sendError(333,"我喜欢的错误");

        } catch (IOException e) {

            e.printStackTrace();

        }

        return new ModelAndView();

    }

}
```

十五、原生组件注入

(一)、使用原生注解方式添加原生组件

使用原生组件都要在启动类中添加@ServletComponenScan 注解

```
//开启 Servlet 扫描

@WebServlet
```

1、添加原生的 Servlet

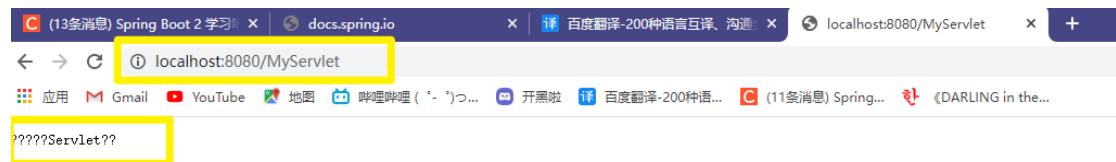
(1)、配置 Servlet 类继承 HttpServlet

```
//声明为原生的 Servlet(urlPattern = 配置路径)

@WebServlet(urlPatterns = "/MyServlet")

public class PrimordialServlet extends HttpServlet {
```

```
@Override  
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,  
IOException {  
    resp.getWriter().write("我是原生的 Servlet 组件");  
}  
}
```



2、添加原生的 filter

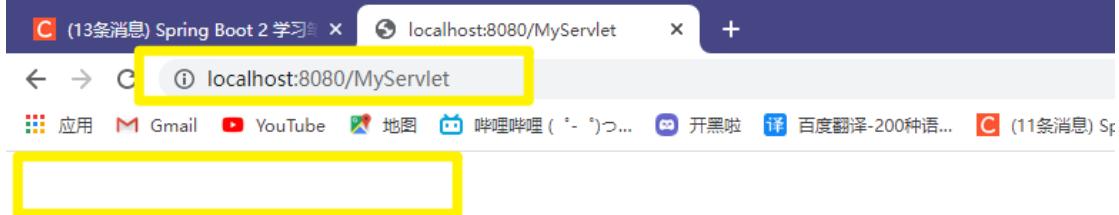
(1)、创建 Filter 类实现 Filter 接口

```
@Slf4j  
@WebFilter  
public class PrimordialFilter implements Filter {  
    @Override  
    public void init(FilterConfig filterConfig) throws ServletException {  
        log.info("Filter 初始化");  
    }  
  
    @Override  
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws  
IOException, ServletException {  
        log.info("Filter 处理业务");  
    }  
  
    @Override  
    public void destroy() {  
        log.info("Filter 销毁");  
    }  
}
```

```
}
```

```
}
```

访问不到,拦截器拦截



```
2021-11-18 08:48:10.156 INFO 3128 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.54]
2021-11-18 08:48:10.221 INFO 3128 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-11-18 08:48:10.221 INFO 3128 --- [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1 ms
2021-11-18 08:48:10.290 INFO 3128 --- [ restartedMain] c.m.h.NativeComponent.PrimordialFilter : Filter初胎化
2021-11-18 08:48:10.627 INFO 3128 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2021-11-18 08:48:10.682 INFO 3128 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-18 08:48:10.693 INFO 3128 --- [ restartedMain] c.mybatisplus.houtai.HoutaiApplication : Started HoutaiApplication in 1.945 seconds (JVM running since 2021-11-18 08:51:12.472)
2021-11-18 08:51:12.472 INFO 3128 --- [nio-8080-exec-1] c.m.h.NativeComponent.PrimordialFilter : Filter处理业务
2021-11-18 08:51:12.491 INFO 3128 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-11-18 08:51:12.491 INFO 3128 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2021-11-18 08:51:12.497 INFO 3128 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms
2021-11-18 08:51:12.493 INFO 3128 --- [nio-8080-exec-2] c.m.h.NativeComponent.PrimordialFilter : Filter处理业务
```

因在 IDEA 中停止程序相当于拔电源,所以无法测试销毁方法

3、添加原生的 Listener

(1)、创建类继承 ServletContextListener

```
@WebListener
@Slf4j
public class PrimordialListener implements ServletContextListener {

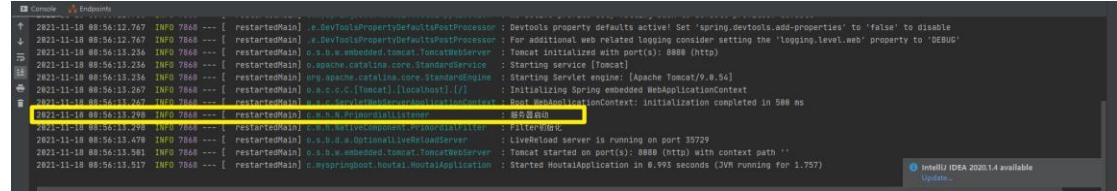
    //初始化方法
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        log.info("服务器启动");
    }

    //销毁方法
}
```

```

@Override
public void contextDestroyed(ServletContextEvent sce) {
    log.info("服务器关闭");
}

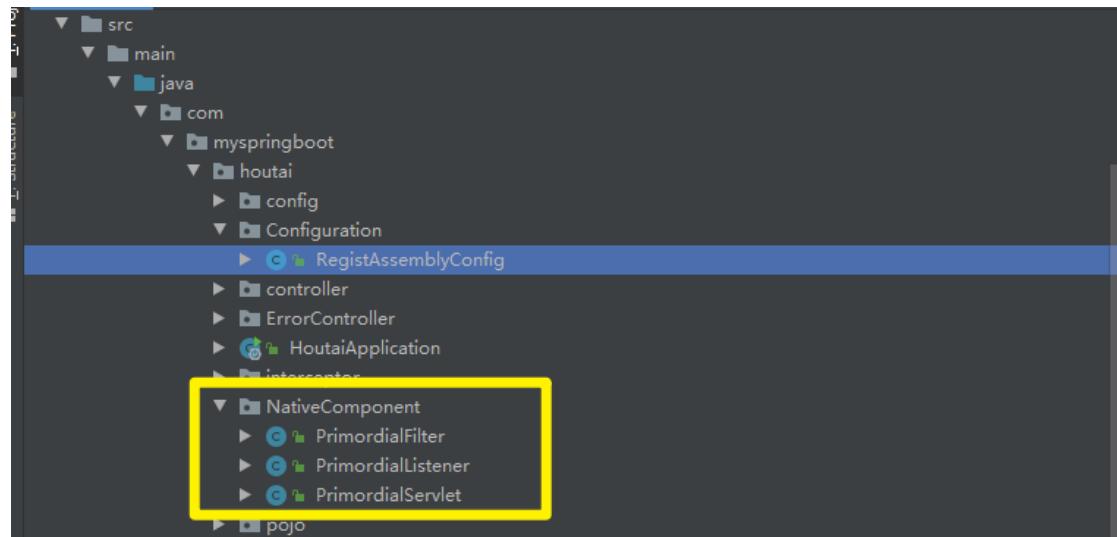
```



因在 IDEA 中停止程序相当于拔电源,所以无法测试销毁方法

(二)、使用 WebConfiguration 注册原生的 Servlet

1、创建三个基础类



(1)、Filter

```

@Slf4j
public class PrimordialFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        log.info("Filter 初始化");
    }
}

```

```
@Override  
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws  
IOException, ServletException {  
    log.info("Filter 处理业务");  
}  
  
@Override  
public void destroy() {  
    log.info("Filter 销毁");  
}  
}
```

(2)、Servlet

```
public class PrimordialServlet extends HttpServlet {  
  
    @Override  
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,  
    IOException {  
        resp.getWriter().write("我是原生的 Servlet 组件");  
    }  
}
```

(3)、Listener

```
@Slf4j  
public class PrimordialListener implements ServletContextListener {  
  
    //初始化方法  
    @Override  
    public void contextInitialized(ServletContextEvent sce) {  
        log.info("服务器启动");  
    }  
  
    //销毁方法
```

```
@Override  
public void contextDestroyed(ServletContextEvent sce) {  
    log.info("服务器关闭");  
}  
}
```

2、使用 WebConfiguration 注册三个组件

```
@Configuration  
public class ReginAssemblyConfig {  
  
    @Bean  
    public ServletRegistrationBean myServlet(){  
        PrimordialServlet primordialServlet = new PrimordialServlet();  
        return new ServletRegistrationBean(primordialServlet,"/my","/my2");  
    }  
  
    @Bean  
    public FilterRegistrationBean myFilter(){  
        PrimordialFilter primordialFilter = new PrimordialFilter();  
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(primordialFilter);  
        filterRegistrationBean.setUrlPatterns(Arrays.asList("/my","/my2"));  
        return filterRegistrationBean;  
    }  
  
    @Bean  
    public ServletListenerRegistrationBean myListener(){  
        PrimordialListener primordialListener = new PrimordialListener();  
        return new ServletListenerRegistrationBean(primordialListener);  
    }  
}
```

```
}
```

(三)定制化 Servlet 容器

1. `WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>`

把配置文件的值和 `ServletWebServerFactory` 进行绑定

2. 使用配置文件修改

3. 直接定义 `ConfigurableServletWebServerFactory`

`XxxCustomizer`:定制器,可以改变 xxx 的默认规则

```
1 import org.springframework.boot.web.server.WebServerFactoryCustomizer;
2 import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
3 import org.springframework.stereotype.Component;
4
5 @Component
6 public class CustomizationBean implements WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {
7
8     @Override
9     public void customize(ConfigurableServletWebServerFactory server) {
10         server.setPort(9000);
11     }
12 }
13 }
```

十七、定制化 SpringBoot 组件的几种方式

(1)、修改配置文件

(2)、编写自定义配置类 `xxxConfiguration+@Bean` 替换或添加容器的默认组件,视图解析器

(3)、Web 应用编写一个配置类实现 `WebMvcConfigurer` 即可定制化 web 功能+`@Bean` 扩展一些组件

```
1 @Configuration
2 public class AdminWebConfig implements WebMvcConfigurer{
3 }
```

(4)、`@EnableWebMvc+WebMvcConfigurer+@Bean` 可以完全接管 SpringMVC,所有的规则自己配置(新手劝退,我直接 tm 的不学)

- `@EnableWebMvc` + `WebMvcConfigurer` — `@Bean` 可以全面接管SpringMVC，所有规则全部自己重新配置；实现定制和扩展功能（高级功能，初学者退避三舍）。

• 原理：

1. `WebMvcAutoConfiguration` 默认的SpringMVC的自动配置功能类，如静态资源、欢迎页等。
2. 一旦使用 `@EnableWebMvc`，会 `@Import(DelegatingWebMvcConfiguration.class)`。
3. `DelegatingWebMvcConfiguration` 的作用，只保证SpringMVC最基本的使用
 1. 把所有系统中的 `WebMvcConfigurer` 拿过来，所有功能的定制都是这些 `WebMvcConfigurer` 合起来一起生效。
 2. 自动配置了一些非常底层的组件，如 `RequestMappingHandlerMapping`，这些组件依赖的组件都是从容器中获取如。
 3. `public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport`。
 4. `WebMvcAutoConfiguration` 里面的配置要能生效必须
`@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)`。
 5. `@EnableWebMvc` 导致了 `WebMvcAutoConfiguration` 没有生效。

十八、切换 WEB 服务器

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        //排除 tomcat 启动器
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
//添加 jetty 服务器
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

十九、数据库连接操作

(一)、数据库场景配置

1、导入 JDBC 依赖

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

2、导入 MySQL 驱动依赖

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

3、相关数据源配置类

DataSourceAutoConfiguration :

数据源的自动配置。

修改数据源相关的配置： `spring.datasource`。 数据库连接池的配置，是自己容器中没有 `DataSource` 才自动配置的。 底层配置好的连接池是： `HikariDataSource`。

DataSourceTransactionManagerAutoConfiguration:

事务管理器的自动配置。

JdbcTemplateAutoConfiguration:

`JdbcTemplate` 的自动配置，可以来对数据库进行 CRUD。

可以修改前缀为 `spring.jdbc` 的配置项来修改 `JdbcTemplate`。

`@Bean @Primary JdbcTemplate : Spring 容器中有这个 JdbcTemplate 组件，使用@Autowired。`

JndiDataSourceAutoConfiguration:

JNDI 的自动配置。

XADatasourceAutoConfiguration:

分布式事务相关的。

(二)、添加数据源

1、引入依赖

```
<dependency>
<groupId>com.alibaba</groupId>
<artifactId>druid-spring-boot-starter</artifactId>
<version>1.1.17</version>
</dependency>
```

2、yaml 文件添加配置(这样就能进入数据访问控制页)

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/db_account
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver

  druid:
    aop-patterns: com.atguigu.admin.* # 监控 SpringBean
    filters: stat,wall      # 底层开启功能, stat (sql 监控) , wall (防火墙)

    stat-view-servlet: # 配置监控页功能
      enabled: true
      login-username: admin
      login-password: admin
      resetEnable: false

    web-stat-filter: # 监控 web
      enabled: true
```

```
urlPattern: /*  
exclusions: '*.*.js, *.gif, *.jpg, *.png, *.css, *.ico, /druid/*'  
  
filter:  
    stat:      # 对上面filters 里面的stat 的详细配置  
        slow-sql-millis: 1000  
        logSlowSql: true  
        enabled: true  
  
    wall:  
        enabled: true  
        config:  
            drop-table-allow: false
```

3、进入 <http://localhost:8080/druid/sql.html> 进行访问

(三)、整合 MyBatis

1、导入 Mybatis 的依赖

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

2、设置 Mybatis 的 sql 映射文件

```
mybatis:  
    mapper-locations: classpath:mybatis/*.xml
```

3、创建数据库实体类

```
@Data  
@AllArgsConstructor  
public class User {  
  
    private String userName;  
    private String passWord;  
}
```

4、创建操作实体的接口

```
@Mapper  
public interface UserMapper {  
    public User getUserById(Integer id);  
}
```

5、接口对应的 mapper 文件

```
<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper  
      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
  
<mapper namespace="com.myspringboot.houtai.mapper.UserMapper">  
    <select id="getUserById" resultType="com.myspringboot.houtai.pojo.User">  
        select * from user where id=#{id}  
    </select>  
</mapper>
```

6、创建 Service 操作接口

```
@Service  
public class UserService {  
  
    @Autowired  
    private UserMapper userMapper;  
  
    public User getUserById(Integer id){  
        return userMapper.getUserById(1);  
    }  
}
```

7、设置 Controller 进行测试

```
@ResponseBody  
@GetMapping("/userTest")  
public User getUserTest(){  
    return userService.getUserById(1);  
}
```

8、测试结果



9、总结

导入 MyBatis 官方 Starter。
编写 Mapper 接口，需@Mapper 注解。
编写 SQL 映射文件并绑定 Mapper 接口。
在 application.yaml 中指定 Mapper 配置文件的所处位置，以及
指定全局配置文件的信息（建议：配置在 mybatis.configuration）。

(四)、整合 MybatisPlus

1、创建数据库

```
CREATE TABLE USER
(
    id BIGINT(20) NOT NULL COMMENT '主键 ID',
    NAME VARCHAR(30) NULL DEFAULT NULL COMMENT '姓名',
    age INT(11) NULL DEFAULT NULL COMMENT '年龄',
    email VARCHAR(50) NULL DEFAULT NULL COMMENT '邮箱',
    PRIMARY KEY (id)
);
```

注意创建完设置 id 为自增

2、添加数据

```
INSERT INTO USER (id, NAME, age, email) VALUES
(1, 'Jone', 18, 'test1@baomidou.com'),
(2, 'Jack', 20, 'test2@baomidou.com'),
(3, 'Tom', 28, 'test3@baomidou.com'),
(4, 'Sandy', 21, 'test4@baomidou.com'),
(5, 'Billie', 24, 'test5@baomidou.com');
```

3、导入依赖

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.1</version>
</dependency>
```

4、排除依赖(因为 MyBatisplus 里面有 jdbc 和 mybatis 所以前导入的就不要了)

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

5、MybatisPlus 中的自动配置

MybatisPlus 自动配置好了 mapper 的映射路径,默认为:classpath*:/*/*xml
在类路径所有子文件下只要存在 mapper 文件,就把 mapper 下的 xml 文件都变为映射文件
@Mapper 标注的接口也会被自动扫描, 直接 @MapperScan("com.lun.boot.mapper") 批量扫描。

6、MybatisPlus 的优点就是 mapper 接口直接继承 BaseMapper 就可以进行简单的增删查改

5、开启 SpringBoot 的 Mapper 注解扫描

```
//开启 Mapper 扫描
@MapperScan("com.myspringboot.houtai.mapper")
```

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

7、编写实体类 User

```
@Data  
@AllArgsConstructor  
public class User {  
  
    // @TableField 注解在添加数据库的时候排除的属性  
    @TableField(exist = false)  
    private String userName;  
  
    @TableField(exist = false)  
    private String passWord;  
  
    private Long id;  
    private String name;  
    private Integer age;  
    private String email;  
}
```

8、创建 mapper 类

```
@Mapper  
public interface UserMapper extends BaseMapper<User> {  
}
```

9、测试

```
@SpringBootTest
@Slf4j
class HoutaiApplicationTests {

    @Autowired
    private UserMapper userMapper;

    @Test
    void contextLoads() {

        List<User> userList = userMapper.selectList(null);
        userList.forEach(System.out::println);
    }
}

Tests passed: 1 of 1 test - 410 ms
2021-11-21 09:31:07.549 INFO 11172 --- [           main] c.m.houtai.HoutaiApplicationTests      : Started HoutaiApplicationTests in 2.632 seconds (JVM running for 3.454)
User(userName=null, password=null, id=1, name=Jone, age=18, email=test1@baomidou.com)
User(userName=null, password=null, id=2, name=Jack, age=20, email=test2@baomidou.com)
User(userName=null, password=null, id=3, name=Tom, age=28, email=test3@baomidou.com)
User(userName=null, password=null, id=4, name=Sandy, age=21, email=test4@baomidou.com)
User(userName=null, password=null, id=5, name=Billie, age=24, email=test5@baomidou.com)

2021-11-21 09:31:07.985 INFO 11172 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource      : HikariPool-1 - Shutdown initiated...
2021-11-21 09:31:07.993 INFO 11172 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource      : HikariPool-1 - Shutdown completed.

Process finished with exit code 0
```

(五)、CRUD

1、分页功能

Service 层

接口实现 `IService<操作的实体类>`(官方 `Service`)里面有很多数据库方法

```
public interface UserService extends IService<User> { }
```

```
public interface IService<T> {
    int DEFAULT_BATCH_SIZE = 1000;

    default boolean save(T entity) { return SqlHelper.retBool(this.getBaseMapper().insert(entity)); }

    @Transactional(
        rollbackFor = {Exception.class}
    )
    default boolean saveBatch(Collection<T> entityList) { return this.saveBatch(entityList, batchSize: 1000); }

    boolean saveBatch(Collection<T> entityList, int batchSize);

    @Transactional(
        rollbackFor = {Exception.class}
    )
    default boolean saveOrUpdateBatch(Collection<T> entityList) { return this.saveOrUpdateBatch(entityList, batchSize: 1000); }

    boolean saveOrUpdateBatch(Collection<T> entityList, int batchSize);

    default boolean removeById(Serializable id) { return SqlHelper.retBool(this.getBaseMapper().deleteById(id)); }

    default boolean removeByMap(Map<String, Object> columnMap) {
        Assert.notEmpty(columnMap, message: "error: columnMap must not be empty", new Object[]{});
        return SqlHelper.retBool(this.getBaseMapper().deleteByMap(columnMap));
    }

    default boolean remove(Wrapper<T> queryWrapper) {
        return SqlHelper.retBool(this.getBaseMapper().delete(queryWrapper));
    }

    default boolean removeByIds(Collection<? extends Serializable> idList) {
        return CollectionUtils.isEmpty(idList) ? false : SqlHelper.retBool(this.getBaseMapper().deleteBatchIds(idList));
    }

    default boolean updateById(T entity) { return SqlHelper.retBool(this.getBaseMapper().updateById(entity)); }

    default boolean update(Wrapper<T> updateWrapper) { return this.update((Object)null, updateWrapper); }

    default boolean update(T entity, Wrapper<T> updateWrapper) {
        return SqlHelper.retBool(this.getBaseMapper().update(entity, updateWrapper));
    }
}
```

Service 类继承 ServiceImpl<操作的实体类> 实现 UserService 接口,里面有调用 UserService 实现的 IService 接口中的方法

```
@Service

public class UserServiceImpl extends ServiceImpl<UserMapper,User> implements UserService {
```

```

public class ServiceImpl<M extends BaseMapper<T>, T> implements IService<T> {
    protected Log log = LogFactory.getLog(this.getClass());
    @Autowired
    protected M baseMapper;
    protected Class<T> entityClass = this.currentModelClass();
    protected Class<T> mapperClass = this.currentMapperClass();

    public ServiceImpl() {
    }

    public M getBaseMapper() { return this.baseMapper; }

    public Class<T> getEntityClass() { return this.entityClass; }

    /** @deprecated */
    @Deprecated
    protected boolean retBool(Integer result) { return SqlHelper.retBool(result); }

    protected Class<T> currentMapperClass() { return ReflectionKit.getSuperClassGenericType(this.getClass(), index: 0); }

    protected Class<T> currentModelClass() { return ReflectionKit.getSuperClassGenericType(this.getClass(), index: 1); }

    /** @deprecated */
    @Deprecated
    protected SqlSession sqlSessionBatch() { return SqlHelper.sqlSessionBatch(this.entityClass); }

    /** @deprecated */
    @Deprecated
    protected void closeSqlSession(SqlSession sqlSession) {
        SqlSessionUtils.closeSqlSession(sqlSession, GlobalConfigUtils.currentSessionFactory(this.entityClass));
    }

    /** @deprecated */
    @Deprecated
    protected String sqlStatement(SqlMethod sqlMethod) {
        return SqlHelper.table(this.entityClass).getSqlStatement(sqlMethod.getMethod());
    }

    @Transactional(
        rollbackFor = {Exception.class}
    )
    public boolean saveBatch(Collection<T> entityList, int batchSize) {

```

Controller 层

```

@.GetMapping("/dynamic_table")

public String dynamic_table(@RequestParam(value = "pg", defaultValue = "1")int pg ,Model model) {

    //从第几页开始,一页多少数据
    Page<User> page = new Page<>(pg,2);

    Page<User> userPage = userService.page(page, null);

    model.addAttribute("users", userPage);

    return "table/dynamic_table";
}

```

现在虽然分页了,分页了,但是不能用,需要开启

开启分页功能

```

// 最新版

@Bean

public MybatisPlusInterceptor mybatisPlusInterceptor() {
    MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();

```

```

    interceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.H2));
}

return interceptor;
}

```

HTML 设置

```

<ul>

    <li class="prev disabled"><a th:href="@{/dynamic_table(pg=users.current-1)}">← Prev</a></li>

    <li th:class="${num==users.current?'active':''}" th:each="num : ${#numbers.sequence(1,users.pages)}">
        <a th:href="@{/dynamic_table(pg=${num})}">[ ${num} ]</a></li>
    <li class="next disabled"><a th:href="@{/dynamic_table(pg=users.current+1)}">Next → </a></li>
</ul>

```

测试

The screenshot shows a responsive table interface. At the top, there's a search bar and a user profile for 'John Doe'. Below the header, the table has four columns: id, name, age, and email. Two rows of data are visible: one for 'Jone' (id 1, age 18, email test1@baomidou.com) and another for 'Jack' (id 2, age 20, email test2@baomidou.com). At the bottom left, it says '当前是第 1 页 总计 3 页 共 5 条记录'. At the bottom right, there's a pagination bar with buttons for '— Prev', '1', '2', '3', and 'Next —'.

This screenshot shows the same responsive table interface, but it's on the second page. It displays the same four columns and two rows of data for 'Sandy' and 'Billie'. The pagination bar at the bottom right shows '— Prev', '1', '2', '3', and 'Next —', with the number '2' highlighted in green to indicate the current page.

2、删除功能

Controller

```

@GetMapping("/user/delete/{id}")
public String deleteUser(@PathVariable("id") Long id,

```

```
    @RequestParam(value = "pg",defaultValue = "1")Integer pg,  
    RedirectAttributes ra){  
        //根据 id 删除用户  
        userService.removeById(id);  
        //重定向的时候往后添加数据  
        ra.addAttribute("pg",pg);  
        return "redirect:/dynamic_table";  
    }  
}
```

HTML

```
<td><a th:href="@{/user/delete/{id}(id=${user.id},pg=${users.current})}" class="btn btn-danger btn-sm"  
type="button">删除</a></td>
```

二十、 JUnit5

(一)、简介

1、 Spring Boot 2.2.0 版本开始引入 JUnit 5 作为单元测试默认库

2、作为最新版本的 JUnit 框架，JUnit5 与之前版本的 JUnit 框架有很大的不同。由三个不同子项目的几个不同模块组成。

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

2.1、 JUnit Platform: Junit Platform 是在 JVM 上启动测试框架的基础，不仅支持 Junit 自制的测试引擎，其他测试引擎也都可以接入。

2.2、 JUnit Jupiter: JUnit Jupiter 提供了 JUnit5 的新的编程模型，是 JUnit5 新特性的核心。内部包含了一个测试引擎，用于在 Junit Platform 上运行。

2.3、 JUnit Vintage: 由于 JUnit 已经发展多年，为了照顾老的项目，JUnit Vintage 提供了兼容 JUnit4.x, JUnit3.x 的测试引擎。

3、注意: SpringBoot 2.4 以上版本移除了默认对 Vintage 的依赖。

如果需要兼容 JUnit4 需要自行引入（不能使用 JUnit4 的功能
@Test）

(二)、导入依赖

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

(三)、想要使用 Junit4 功能就需要导入 Vintage

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

(四)、常用的注解

@Test: 表示方法是测试方法。但是与 JUnit4 的@Test 不同，他的职责非常单一不能声明任何属性，拓展的测试将会由 Jupiter 提供额外测试

@ParameterizedTest: 表示方法是参数化测试。

@RepeatedTest: 表示方法可重复执行。

@DisplayName: 为测试类或者测试方法设置展示名称。

@BeforeEach: 表示在每个单元测试之前执行。

@AfterEach: 表示在每个单元测试之后执行。

@BeforeAll: 表示在所有单元测试之前执行。

@AfterAll: 表示在所有单元测试之后执行。

@Tag: 表示单元测试类别，类似于 JUnit4 中的**@Categories**。

@Disabled: 表示测试类或测试方法不执行，类似于 JUnit4 中的**@Ignore**。

@Timeout: 表示测试方法运行如果超过了指定时间将会返回错误。

@ExtendWith: 为测试类或测试方法提供扩展类引用。

```
@Disabled  
 @DisplayName("测试方法2")  
 @Test  
 void test2() {  
     System.out.println(2);  
 }
```

(五)、断言

1、简单断言

方法	说明
assertEquals	判断两个对象或两个原始类型是否相等
assertNotEquals	判断两个对象或两个原始类型是否不相等
assertSame	判断两个对象引用是否指向同一个对象
assertNotSame	判断两个对象引用是否指向不同的对象
assertTrue	判断给定的布尔值是否为 true
assertFalse	判断给定的布尔值是否为 false
assertNull	判断给定的对象引用是否为 null
assertNotNull	判断给定的对象引用是否不为 null

```
1 | @Test
2 | @DisplayName("simple assertion")
3 | public void simple() {
4 |     assertEquals(3, 1 + 2, "simple math");
5 |     assertNotEquals(3, 1 + 1);
6 |
7 |     assertNotSame(new Object(), new Object());
8 |     Object obj = new Object();
9 |     assertEquals(obj, obj);
10 |
11 |     assertFalse(1 > 2);
12 |     assertTrue(1 < 2);
13 |
14 |     assertNull(null);
15 |     assertNotNull(new Object());
16 | }
```

2、数组断言

通过 `assertArrayEquals` 方法来判断两个对象或原始类型的数组是否相等。

```
1 | @Test
2 | @DisplayName("array assertion")
3 | public void array() {
4 |     assertArrayEquals(new int[]{1, 2}, new int[] {1, 2});
5 | }
```

3、组合断言

`assertAll()` 方法接受多个 `org.junit.jupiter.api.Executable` 函数式接口的实例作为要验证的断言，可以通过 `lambda` 表达式很容易的提供这些断言。

```
1 | @Test
2 | @DisplayName("assert all")
3 | public void all() {
4 |     assertAll("Math",
5 |             () -> assertEquals(2, 1 + 1),
6 |             () -> assertTrue(1 > 0)
7 |         );
8 | }
```

4、异常断言

方法中出异常顺利执行

```
1 | @Test
2 | @DisplayName("异常测试")
3 | public void exceptionTest() {
4 |     ArithmeticException exception = Assertions.assertThrows(
5 |         //扔出断言异常
6 |         ArithmeticException.class, () -> System.out.println(1 % 0));
7 | }
```

5、超时断言

Assertions.assertTimeout()为测试方法设置了超时时间。

```
1 | @Test
2 | @DisplayName("超时测试")
3 | public void timeoutTest() {
4 |     //如果测试方法时间超过1s将会异常
5 |     Assertions.assertTimeout(Duration.ofMillis(1000), () -> Thread.sleep(500));
6 | }
```

6、快速失败(直接退出程序)

```
1 | @Test
2 | @DisplayName("fail")
3 | public void shouldFail() {
4 |     fail("This should fail");
5 | }
```

(六)、前置条件

Unit 5 中的前置条件 (assumptions【假设】) 类似于断言，不同之处在于不满足的断言 **assertions** 会使得测试方法失败，而不满足的前置条件只会使得测试方法的执行终止。

前置条件可以看成是测试方法执行的前提，当该前提不满足时，就没有继续执行的必要。

```
1 @DisplayName("前置条件")
2 public class AssumptionsTest {
3     private final String environment = "DEV";
4
5     @Test
6     @DisplayName("simple")
7     public void simpleAssume() {
8         assumeTrue(Objects.equals(this.environment, "DEV"));
9         assumeFalse(() -> Objects.equals(this.environment, "PROD"));
10    }
11
12    @Test
13    @DisplayName("assume then do")
14    public void assumeThenDo() {
15        assumingThat(
16            Objects.equals(this.environment, "DEV"),
17            () -> System.out.println("In DEV")
18        );
19    }
20 }
```

`assumeTrue` 和 `assumeFalse` 确保给定的条件为 `true` 或 `false`，不满足条件会使得测试执行终止。

`assumingThat` 的参数是表示条件的布尔值和对应的 `Executable` 接口的实现对象。只有条件满足时，`Executable` 对象才会被执行；当条件不满足时，测试执行并不会终止。

(七)、嵌套测试

JUnit 5 可以通过 Java 中的内部类和`@Nested` 注解实现嵌套测试，从而可以更好的把相关的测试方法组织在一起。在内部类中可以使用`@BeforeEach` 和`@AfterEach` 注解，而且嵌套的层次没有限制。

类里面包装类

```
1 @DisplayName("A stack")
2 class TestingAStackDemo {
```

类中包括

```
12     @Nested
13     @DisplayName("when new")
14     class WhenNew {
```

(八)、参数化测试

利用 @ValueSource 等注解

@ValueSource: 为参数化测试指定入参来源，支持八大基础类以及 String 类型,Class 类型

@NullSource: 表示为参数化测试提供一个 null 的入参

@EnumSource: 表示为参数化测试提供一个枚举入参

@CsvFileSource: 表示读取指定 CSV 文件内容作为参数化测试入参

@MethodSource: 表示读取指定方法的返回值作为参数化测试入参(注意方法返回需要是一个流)

```
1  @ParameterizedTest
2  @ValueSource(strings = {"one", "two", "three"})
3  @DisplayName("参数化测试1")
4  public void parameterizedTest1(String string) {
5      System.out.println(string);
6      Assertions.assertTrue(StringUtils.isNotBlank(string));
7  }
8
9
10 @ParameterizedTest
11 @MethodSource("method")    //指定方法名
12 @DisplayName("方法来源参数")
13 public void testWithExplicitLocalMethodSource(String name) {
14     System.out.println(name);
15     Assertions.assertNotNull(name);
16 }
17
18 static Stream<String> method() {
19     return Stream.of("apple", "banana");
20 }
```

(九)、Junit4 升级

在进行迁移的时候需要注意如下的变化：

- 注解在 org.junit.jupiter.api 包中，断言在 org.junit.jupiter.api.Assertions 类中，前置条件在 org.junit.jupiter.api.Assumptions 类中。
- 把@Before 和@After 替换成@BeforeEach 和

`@AfterEach`。

- 把`@BeforeClass` 和`@AfterClass` 替换成`@BeforeAll` 和`@AfterAll`。
- 把`@Ignore` 替换成`@Disabled`。
- 把`@Category` 替换成`@Tag`。
- 把`@RunWith`、`@Rule` 和`@ClassRule` 替换成`@ExtendWith`。

二十一、监控功能

(一)、说明

未来每一个微服务在云上部署以后，我们都需要对其进行监控、追踪、审计、控制等。**SpringBoot** 就抽取了 **Actuator** 场景，使得我们每个微服务快速引用即可获得生产级别的应用监控、审计等功能。

(二)、**SpringBoot1** 和 **SpringBoot2** 监控的不同

1、**SpringBoot Actuator 1.x**

- 支持 **SpringMVC**
- 基于继承方式进行扩展
- 层级 **Metrics** 配置
- 自定义 **Metrics** 收集
- 默认较少的安全策略

2、**SpringBoot Actuator 2.x**

- 支持 **SpringMVC**、**JAX-RS** 以及 **Webflux**

- 注解驱动进行扩展
- 层级&名称空间 Metrics
- 底层使用 MicroMeter，强大、便捷默认丰富的安全策略

(三)、使用

1、导入依赖

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

默认只启动 shutdown 如果开启其他需要配置开启

2、开启全部功能

```
management:
  endpoints:
    enabled-by-default: true # 暴露所有端点信息
  web:
    exposure:
      include: '*' # 以 web 方式暴露
```

访问 <http://localhost:8080/actuator>



3、监控列表

有一些监控端点存在二级监控端点

测试例子

<http://localhost:8080/actuator/beans>

<http://localhost:8080/actuator/configprops>

<http://localhost:8080/actuator/metrics>

<http://localhost:8080/actuator/metrics/jvm.gc.pause>

<http://localhost:8080/actuator/metrics/endpointName/detailPath>

4、端点的开启与禁用

ID	JMX	Web
<code>auditevents</code>	Yes	No
<code>beans</code>	Yes	No
<code>caches</code>	Yes	No
<code>conditions</code>	Yes	No
<code>configprops</code>	Yes	No
<code>env</code>	Yes	No
<code>flyway</code>	Yes	No
<code>health</code>	Yes	Yes
<code>heapdump</code>	N/A	No
<code>httptrace</code>	Yes	No
<code>info</code>	Yes	Yes
<code>integrationgraph</code>	Yes	No
<code>jolokia</code>	N/A	No
<code>logfile</code>	N/A	No
<code>loggers</code>	Yes	No
<code>liquibase</code>	Yes	No
<code>metrics</code>	Yes	No
<code>mappings</code>	Yes	No
<code>prometheus</code>	N/A	No
<code>scheduledtasks</code>	Yes	No
<code>sessions</code>	Yes	No
<code>shutdown</code>	Yes	No
<code>startup</code>	Yes	No
<code>threaddump</code>	Yes	No

如果要开启单个功能需要将全部功能注释

5、定制 Endpoint(以 Health 为例)

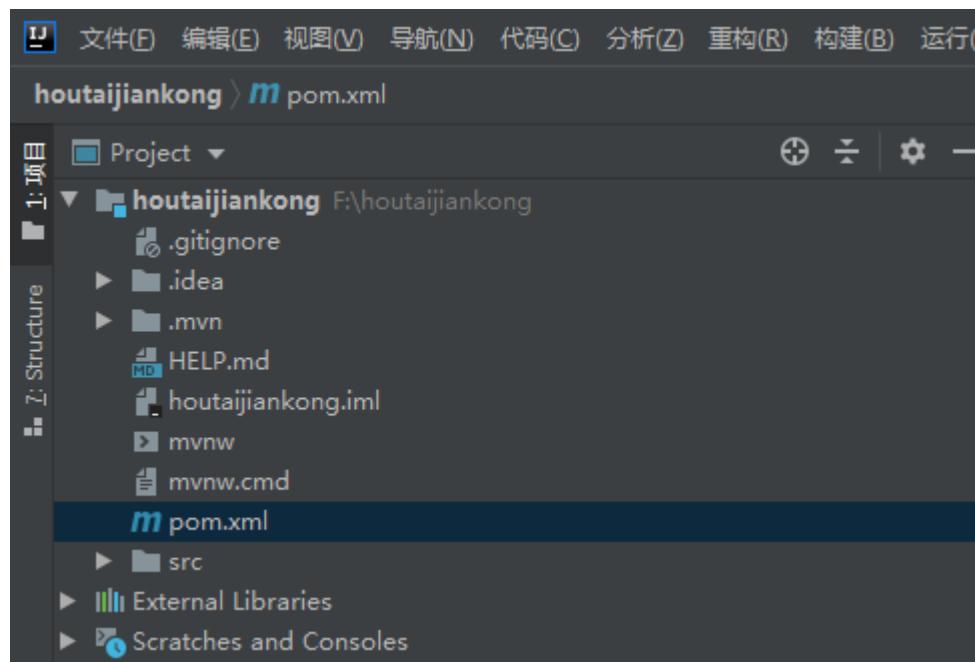
```
management:  
  endpoints:  
    enabled-by-default: false # 禁止暴露所有端点信息  
  web:  
    exposure:  
      include: '*' # 以 web 方式暴露  
  endpoint:  
    health:  
      enabled: true # 开启 health 端点  
      show-details: always # 显示信息为全部  
  info:  
    enabled: true # 开启 info 端点
```

现在只开启了 health 和 info 端点



(四)、Web 页面数据显示

1、首先建一个带 web 工程的项目



2、导入 Server 依赖

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
    <version>2.3.1</version>
</dependency>
```

3、启动类添加注解

```
@EnableAutoConfiguration
@EnableAdminServer
```

4、修改 tomcat 端口以免和主服务端口冲突

```
server.port=8888
```

5、主服务添加客户端依赖

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.3.1</version>
</dependency>
```

6、添加服务器地址

```
boot:
  admin:
    client:
      url: http://localhost:8888 # 访问监控 web 服务器地址
      instance:
        prefer-ip: true # 使用 ip 注册客户端
  application:
    name: houtai
```

二十二、profile 环境切换

(一)、说明

为了方便多环境适配，Spring Boot 简化了 profile 功能。

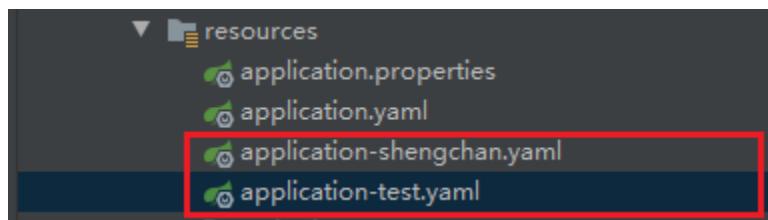
- 默认配置文件 `application.properites` 任何时候都会加载。
- 指定环境配置文件 `application-{xxx}.yaml`
- 激活指定环境

- 配置文件激活: `spring.profiles.active=xxx`
- 命令行激活: `java -jar sprigboot.jar --spring.profiles.active=xxx -- person.name=value` (修改配置文件的任意值, 命令行优先)
- 默认配置与环境配置同时生效
- 同名配置项, `profile` 配置优先

(二)、使用

1、准备环境

1.1、首先创建 2 个 `application-xxx.yaml` 的文件, 这里创建的是 `application-test.yaml` 和 `application-shengchan.yaml`



1.2、实体类 Person

```
//装入容器
@Component
//使用配置文件自动装入
@ConfigurationProperties("person")
//生成基础方法
@Data
public class Person {

    private String name;
    private Integer age;
}
```

1.3、设置配置文件 `application-shengchan.yaml`

```
person:
  name: shengchan-李四
```

```
age: 18
```

```
server:
```

```
port: 9999 #端口号
```

1.4、设置配置文件 application-test.yaml

```
person:
```

```
name: test-张三
```

```
age: 18
```

```
server:
```

```
port: 8989
```

2、配置文件模式切换

2.1、主配置文件,当前设置为 test

```
spring.profiles.active=test
```

2.2、启动测试

```
2021-11-24 14:46:30.727 INFO 13708 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2021-11-24 14:46:30.729 WARN 13708 --- [ restartedMain] com.zaxxer.hikari.util.DriverDataSource : Registered driver with driverClassName=com.mysql.jdbc.Driver was not found, trying direct instantiation.
2021-11-24 14:46:30.798 INFO 13708 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2021-11-24 14:46:30.844 INFO 13708 --- [ restartedMain] o.s.b.a.web.EndpointLinksResolver : LiveReload server is running on port 35729
2021-11-24 14:46:30.846 INFO 13708 --- [ restartedMain] o.s.b.a.web.EndpointLinksResolver : Exposing 14 endpoint(s) beneath base path '/actuator'
2021-11-24 14:46:30.858 INFO 13708 --- [ restartedMain] o.s.b.a.embedded.tomcat.TomcatWebServer : Tomcat started on port(s) [8989] with context path '' - 涵盖为test的资源配置文件
2021-11-24 14:46:30.858 INFO 13708 --- [ restartedMain] o.s.b.a.embedded.tomcat.TomcatWebServer : Started HouthApplication in 2.461 seconds (JVM running for 2.641)
2021-11-24 14:46:30.992 INFO 13708 --- [ restartedMain] o.s.b.c.r.ApplicationRegistration : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-11-24 14:46:31.095 INFO 13708 --- [ restartedMain] o.s.b.c.r.ApplicationRegistration : Initializing Servlet 'dispatcherServlet'
2021-11-24 14:46:31.095 INFO 13708 --- [ restartedMain] o.s.b.c.r.ApplicationRegistration : Completed initialization in 1 ms
2021-11-24 14:46:32.978 WARN 13708 --- [gistrationTask1] o.s.b.c.r.ApplicationRegistration : Failed to register application as Application(name=houtai, managementUrl=http://192.168.2.199:8989/actuator, healthUrl=http://192.168.2.199:8989/health)
```



2.3、主配置文件设置为 shengchan

```
spring.profiles.active=shengchan
```

2.4、启动测试

```
2021-11-24 14:48:53.513 INFO 16516 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2021-11-24 14:48:53.515 WARN 16516 --- [ restartedMain] com.zaxxer.hikari.util.DriverDataSource : Registered driver with driverClassName=com.mysql.jdbc.Driver was not found, trying direct instantiation.
2021-11-24 14:48:53.587 INFO 16516 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2021-11-24 14:48:53.635 INFO 16516 --- [ restartedMain] o.s.b.a.web.EndpointLinksResolver : LiveReload server is running on port 35729
2021-11-24 14:48:53.638 INFO 16516 --- [ restartedMain] o.s.b.a.embedded.EndpointLinksResolver : Exposing 14 endpoint(s) beneath base path '/actuator'
2021-11-24 14:48:53.678 INFO 16516 --- [ restartedMain] o.s.b.a.embedded.tomcat.TomcatWebServer : Tomcat started on port(s) [8999] (http) with context path '' - 涵盖为shengchan配置文件的8999
2021-11-24 14:48:53.682 INFO 16516 --- [ restartedMain] o.s.b.c.r.ApplicationRegistration : Started HouthApplication in 2.233 seconds (JVM running for 2.822)
2021-11-24 14:48:54.391 INFO 16516 --- [ restartedMain] o.s.b.c.r.ApplicationRegistration : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-11-24 14:48:54.409 INFO 16516 --- [ restartedMain] o.s.b.c.r.ApplicationRegistration : Initializing Servlet 'dispatcherServlet'
2021-11-24 14:48:54.409 INFO 16516 --- [ restartedMain] o.s.b.c.r.ApplicationRegistration : Completed initialization in 1 ms
2021-11-24 14:48:55.776 WARN 16516 --- [gistrationTask1] o.s.b.c.r.ApplicationRegistration : Failed to register application as Application(name=houtai, managementUrl=http://192.168.2.199:8999/actuator, healthUrl=http://192.168.2.199:8999/health)
```

3、命令行切换(在 active 后面加上环境名称就行)

```
F:\houmai\target>java -jar houmai-0.0.1-SNAPSHOT.jar --spring.profiles.active=test
:: Spring Boot ::

2023-11-24 14:54:18.101 INFO 5112 --- [           main] c.myspringboot.houmai.HoumaiApplication : Starting HoumaiApplication v0.0.1-SNAPSHOT using Java 11.0.11 on DESKTOP-94U9R11 with PID 5112 (F:\houmai\target\houmai-0.0.1-SNAPSHOT.jar)
2023-11-24 14:54:18.102 INFO 5112 --- [           main] o.s.b.a.e.EmbeddedTomcatWebServer   : The following profiles are active: test
2023-11-24 14:54:18.102 INFO 5112 --- [           main] o.s.b.a.e.EmbeddedTomcatWebServer   : Bootstrapping Spring Data JDBC repositories in DEFAULT mode.
2023-11-24 14:54:18.102 INFO 5112 --- [           main] o.s.b.a.e.EmbeddedTomcatWebServer   : Registering default repository in memory. Found 0 JDBC repository interfaces.
2023-11-24 14:54:18.497 INFO 5112 --- [           main] o.s.b.a.e.EmbeddedTomcatWebServer   : Tomcat initialized with port(s): 8989 (http)
2023-11-24 14:54:18.501 INFO 5112 --- [           main] o.apache.catalina.core.StandardService : Starting service [tomcat]
2023-11-24 14:54:18.501 INFO 5112 --- [           main] o.apache.catalina.core.StandardEngine : Starting Servlet Engine: [Apache Tomcat/9.0.54]
2023-11-24 14:54:18.550 INFO 5112 --- [           main] o.a.c.c.C.[Tomcat].[localhost].|/
2023-11-24 14:54:18.550 INFO 5112 --- [           main] w.a.c.Servlet#WebApplicationContext  : Initializing Spring embedded WebApplicationContext
2023-11-24 14:54:18.550 INFO 5112 --- [           main] w.a.c.Servlet#WebApplicationContext  : Root WebApplicationContext: initialization completed in 1395 ms
2023-11-24 14:54:18.690 INFO 5112 --- [           main] c.m.h.NativeComponents$PrimedaiFilter  : Filter初始化
2023-11-24 14:54:18.690 INFO 5112 --- [           main] com.mysql.jdbc.Driver                : Loading class com.mysql.jdbc.Driver. This is deprecated. The new driver class is com.mysql.cj.jdbc.Driver. The driver is automatically registered via the SPI and manual loading of the driver class is generally unnecessary.

SpringBoot启动日志
3.4.1
2023-11-24 14:54:20.626 INFO 5112 --- [           main] com.zaxxer.hikaridatasource          : HikariPool-1 - Starting...
2023-11-24 14:54:20.630 WARN 5112 --- [           main] com.zaxxer.hikaridatasource          : Registered driver with driverClassName=com.mysql.jdbc.Driver was not found, trying direct instantiation.
2023-11-24 14:54:20.630 WARN 5112 --- [           main] com.zaxxer.hikaridatasource          : HikariDataSource - Not instantiated.
2023-11-24 14:54:20.825 INFO 5112 --- [           main] o.s.b.a.e.web.EndpointLinksResolver  : Exposing 14 endpoint(s) beneath base path '/actuator'
2023-11-24 14:54:20.825 INFO 5112 --- [           main] o.s.b.a.e.web.EndpointLinksResolver  : exposing mappings for: [ManagementPortInfo, HealthEndpoint, InfoEndpoint, MetricsEndpoint, MBeanExporter, ProfileEndpoint, SessionEndpoint, ThymeleafEndpoint, AuditEndpoint, ConfigurationEndpoint, EndpointLinksEndpoint, ApplicationEndpoint, ApplicationRegistrationEndpoint, HealthEndpoint, ManagementPortInfo, SessionEndpoint, ThymeleafEndpoint, AuditEndpoint, ConfigurationEndpoint, EndpointLinksEndpoint, ApplicationEndpoint, ApplicationRegistrationEndpoint]
2023-11-24 14:54:20.871 INFO 5112 --- [           main] c.myspringboot.houmai.HoumaiApplication : Failed to register application as Application(name=houmai, managementUrl=http://192.168.2.199:8989/actuator, healthUrl=http://192.168.2.199:8989/actuator/health, serviceUrl=http://192.168.2.199:8989) at spring-boot-admin (http://localhost:8080/instances): 1/0 error on POST request for http://localhost:8080/instances ; Connection refused; connect, nested exception is: ]
```

(三)、@Profile 注解

@Profile 注解可以标注在类上,也可以标注在方法上

```
//装入容器
@Component
//使用配置文件自动装入
@ConfigurationProperties("person")
//生成基础方法
@Data
//当前环境为 test 的时候该类才生效
@Profile("test")
public class Person {

    private String name;
    private Integer age;

    //当前环境为 test 才生效此方法
    @Profile("test")
    void haha(){
        System.out.println("haha");
    }
}
```

(四)、环境组

主配置文件

将 huanjing 和 test 添加到 myHuanjing 组中

```
spring.profiles.group.myHuanjing[0]=test  
spring.profiles.group.myHuanjing[1]=shengchan
```

使用 myHuanjing 组

```
spring.profiles.active=myHuanjing
```

此时 application-test.yaml 和 application-shengchan.yaml 变为

Application-shengchan.yaml

```
person:  
  name: shengchan-李四
```

application-test.yaml

```
person:  
  age: 18
```

组合来给 Person 实体类赋值

开始测试



二十三、配置文件加载优先级

(一)、外部配置源

- 1、java 属性文件(application.properties)
- 2、YAML 文件(application.yaml)
- 3、环境变量
- 4、命令行参数(java -jar xxx.jar -- 参数值)

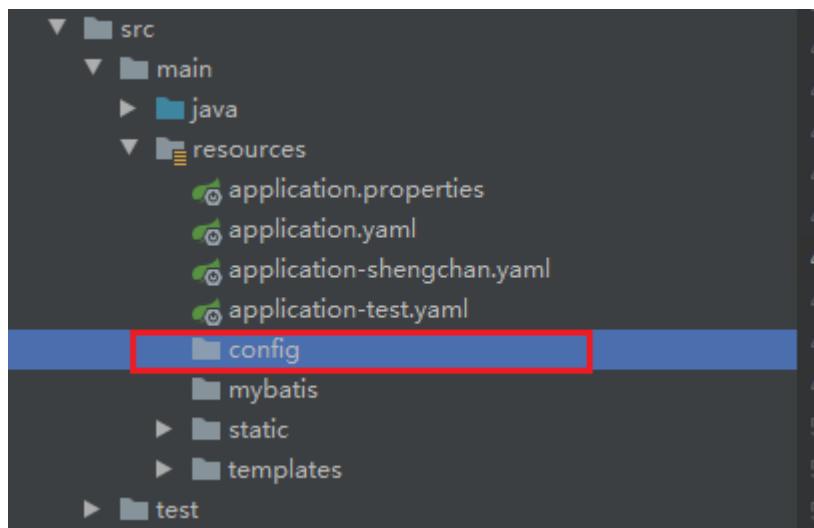
(二)、配置文件查找位置

1、越往下越优先

2、classpath 跟路径



3、根路径下的 config 目录



4、jar 包当前前目录

名称	修改日期	类型	大小
classes	2021/11/24 15:09	文件夹	
generated-sources	2021/11/24 14:50	文件夹	
generated-test-sources	2021/11/24 14:50	文件夹	
maven-archiver	2021/11/24 14:50	文件夹	
maven-status	2021/11/24 14:50	文件夹	
surefire-reports	2021/11/24 14:50	文件夹	
test-classes	2021/11/24 14:50	文件夹	
application.yaml	2021/11/24 12:40	YAML 文件	1 KB
houtai-0.0.1-SNAPSHOT.jar	2021/11/24 14:50	WinRAR 压缩文件	31,474 KB
houtai-0.0.1-SNAPSHOT.jar.original	2021/11/24 14:50	ORIGINAL 文件	2,929 KB

5、jar 包当前 config 目录



6、jar 包当前 config 目录下的字目录

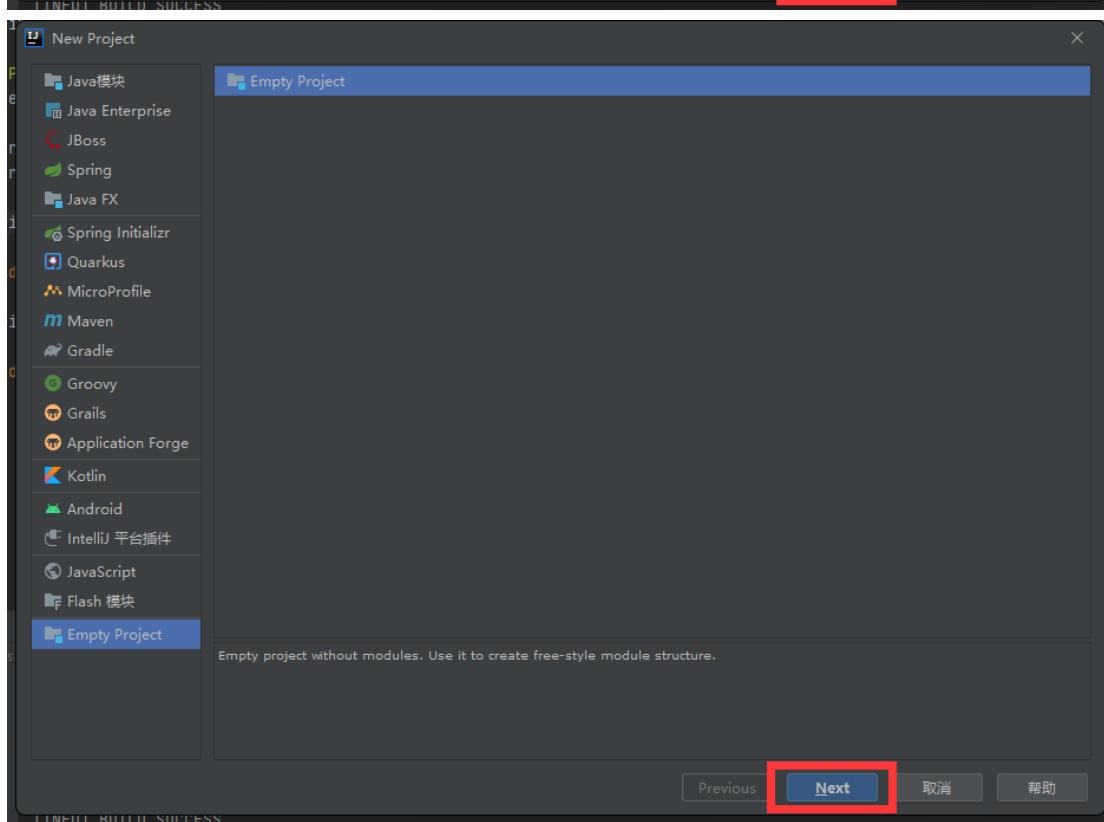
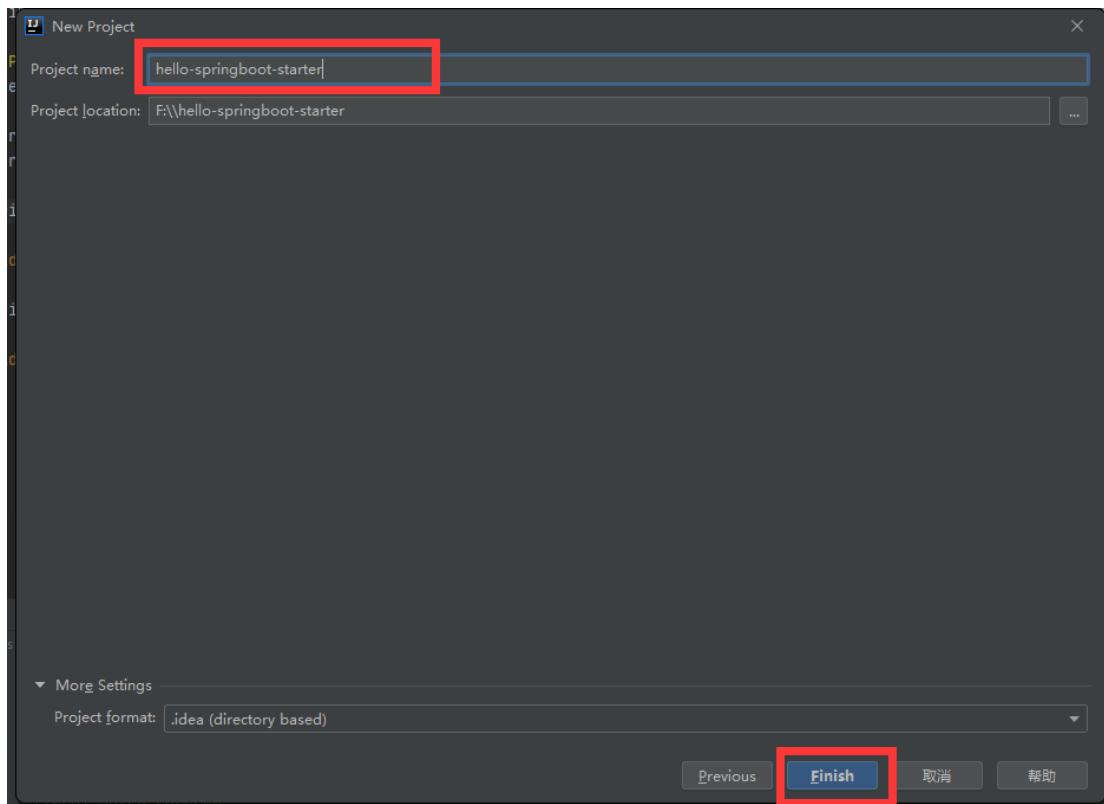


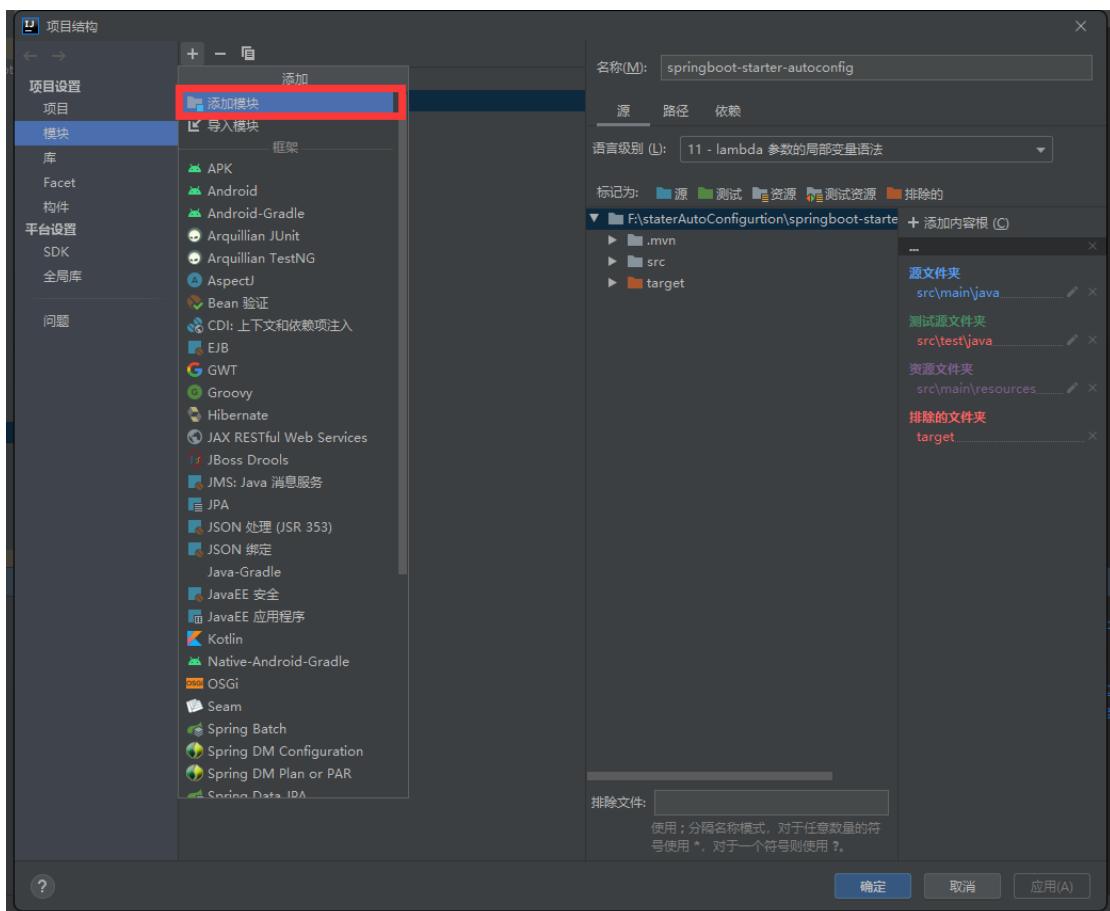
(三)、配置文件的加载顺序

- 1、当前 jar 包内部的 application.properties 和 application.yaml
- 2、当前 jar 包内部的 application-{profile}.properties 和 application-{profile}.yaml
- 3、打包 jar 包目录下的 application.properties 和 application.yaml
- 4、打包 jar 包目录下的 application-{profile}.properties 和 application-{profile}.yaml

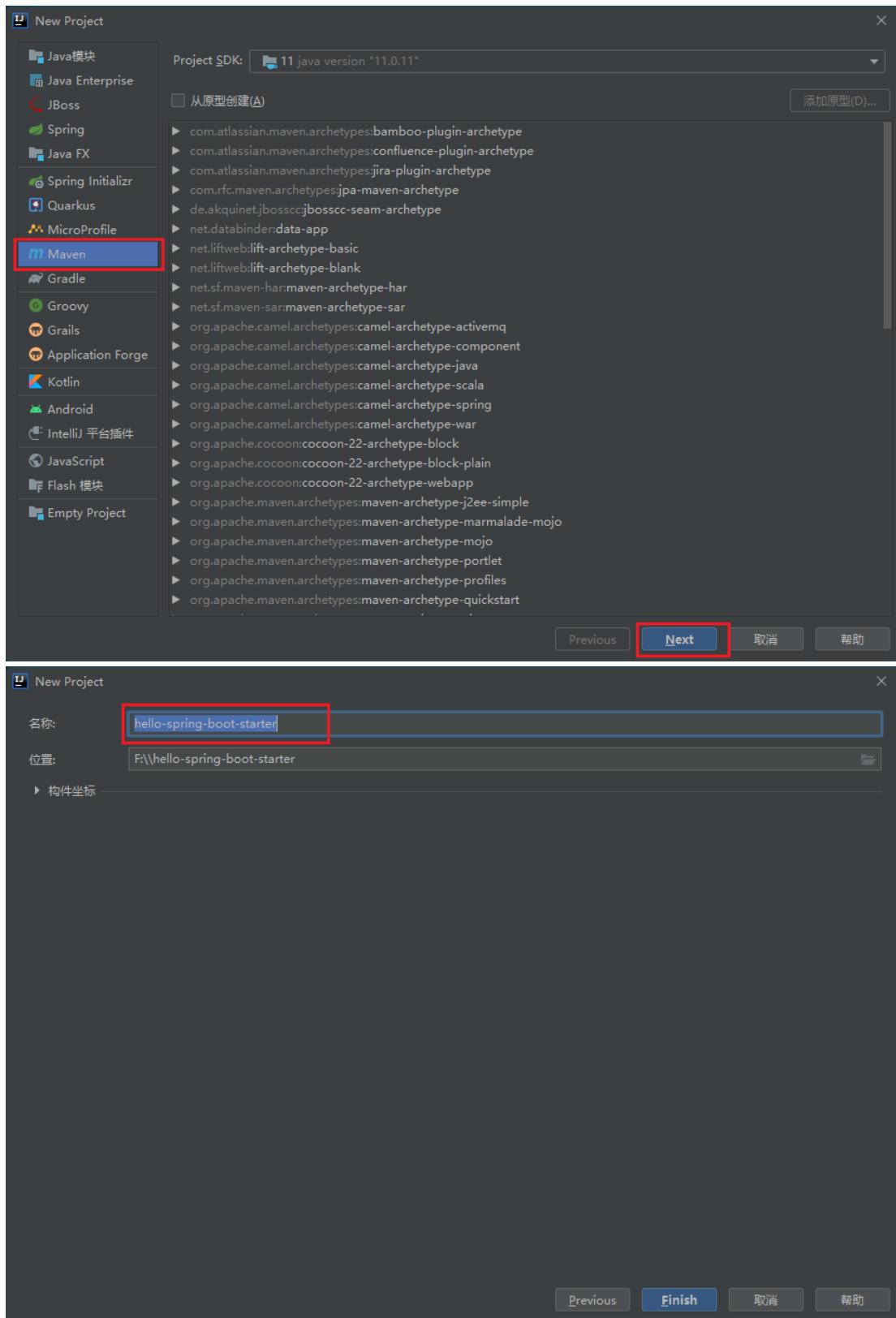
二十四、自定义 starter

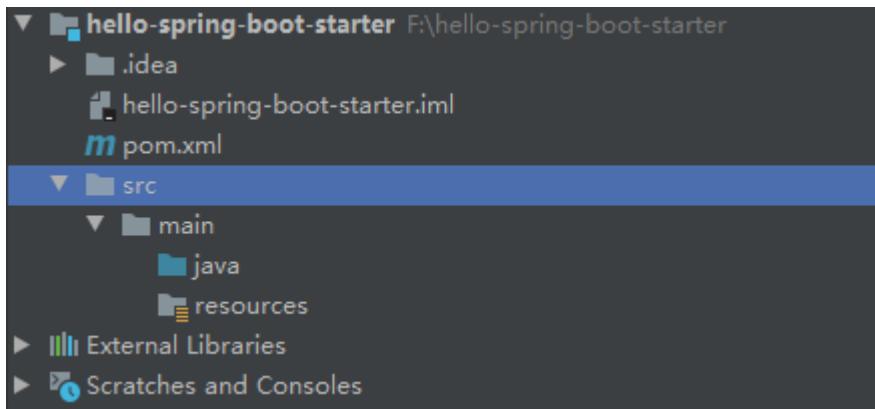
(一)、创建一个空的工程



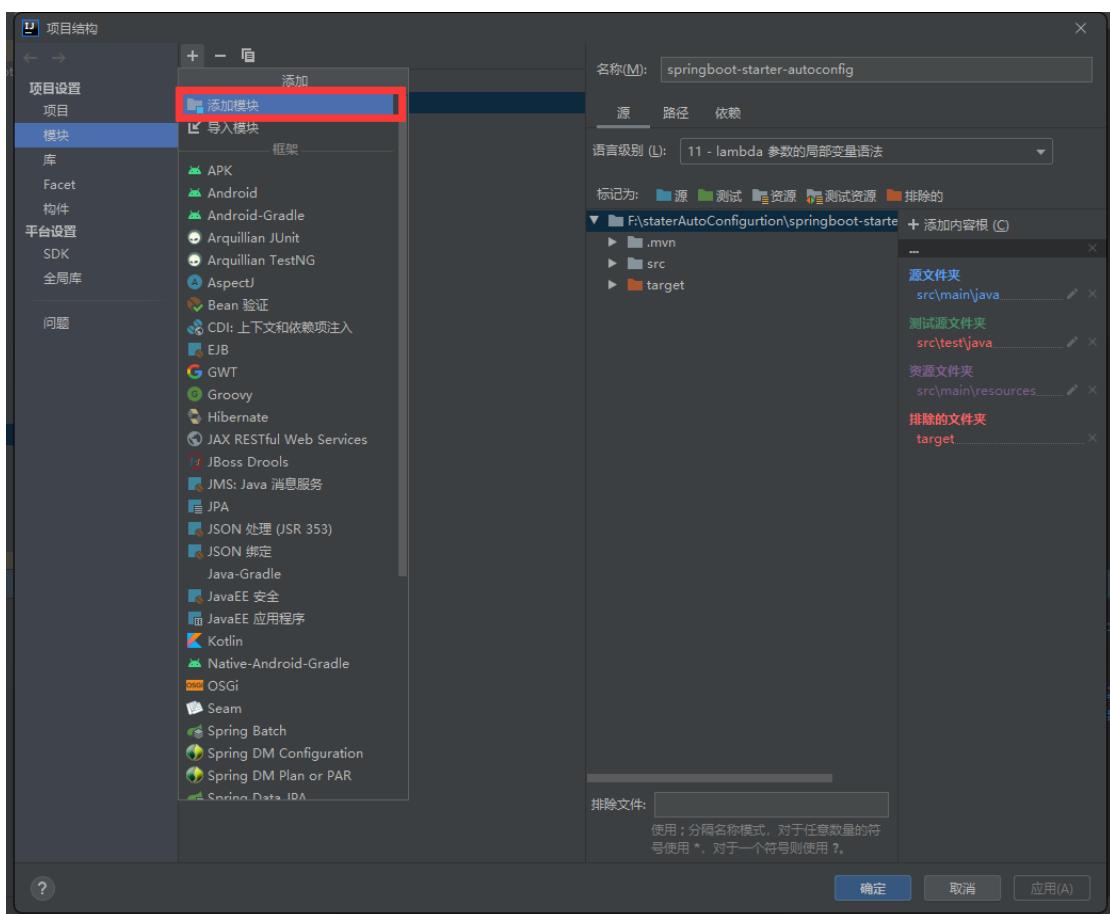


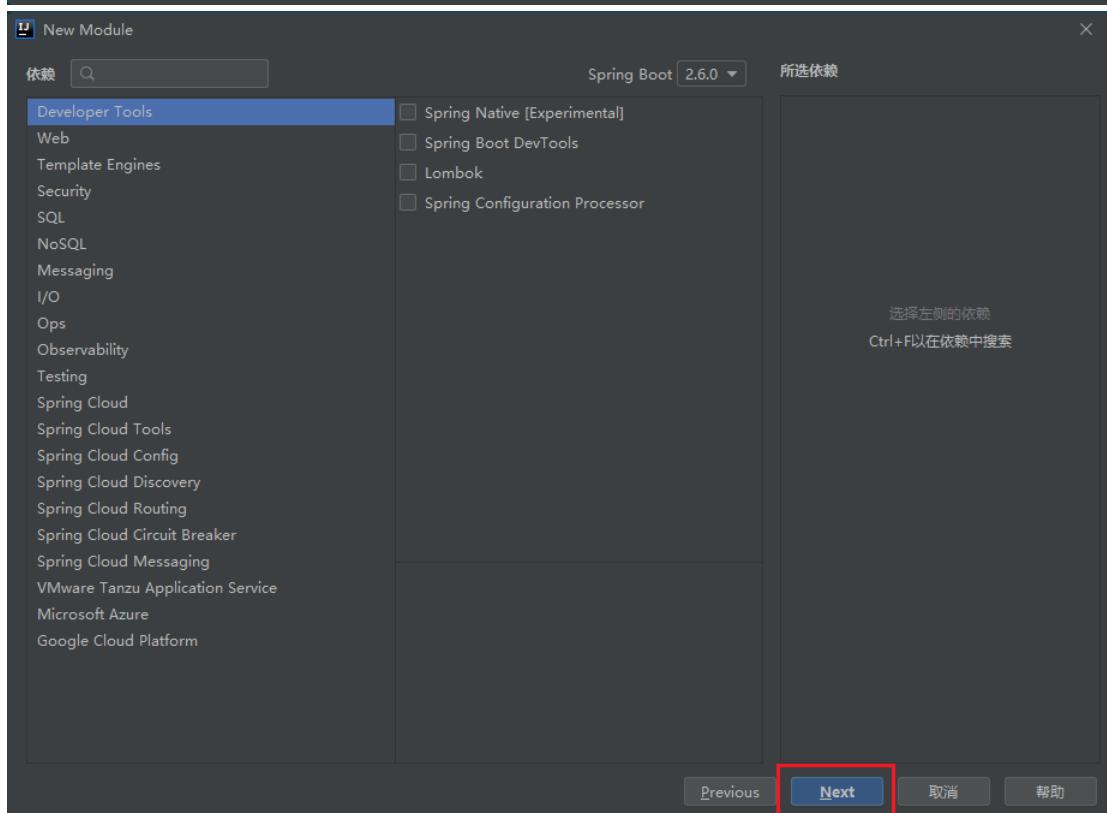
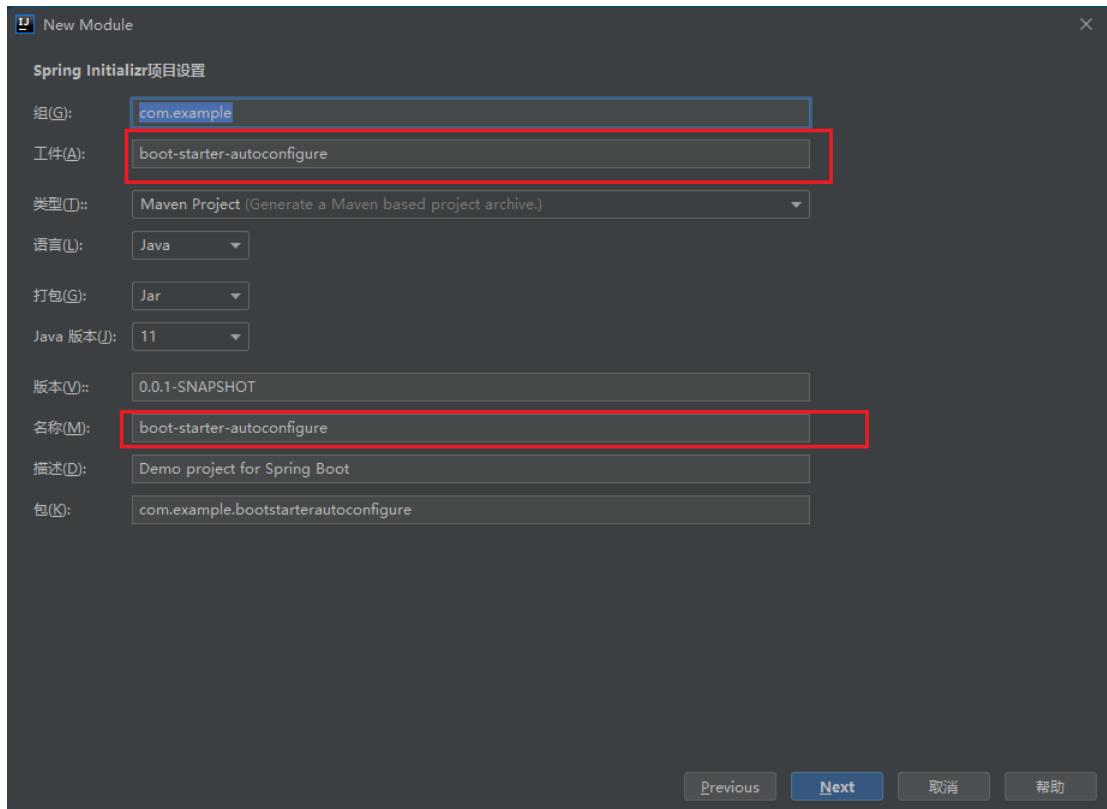
(二)、添加一个普通的 Maven

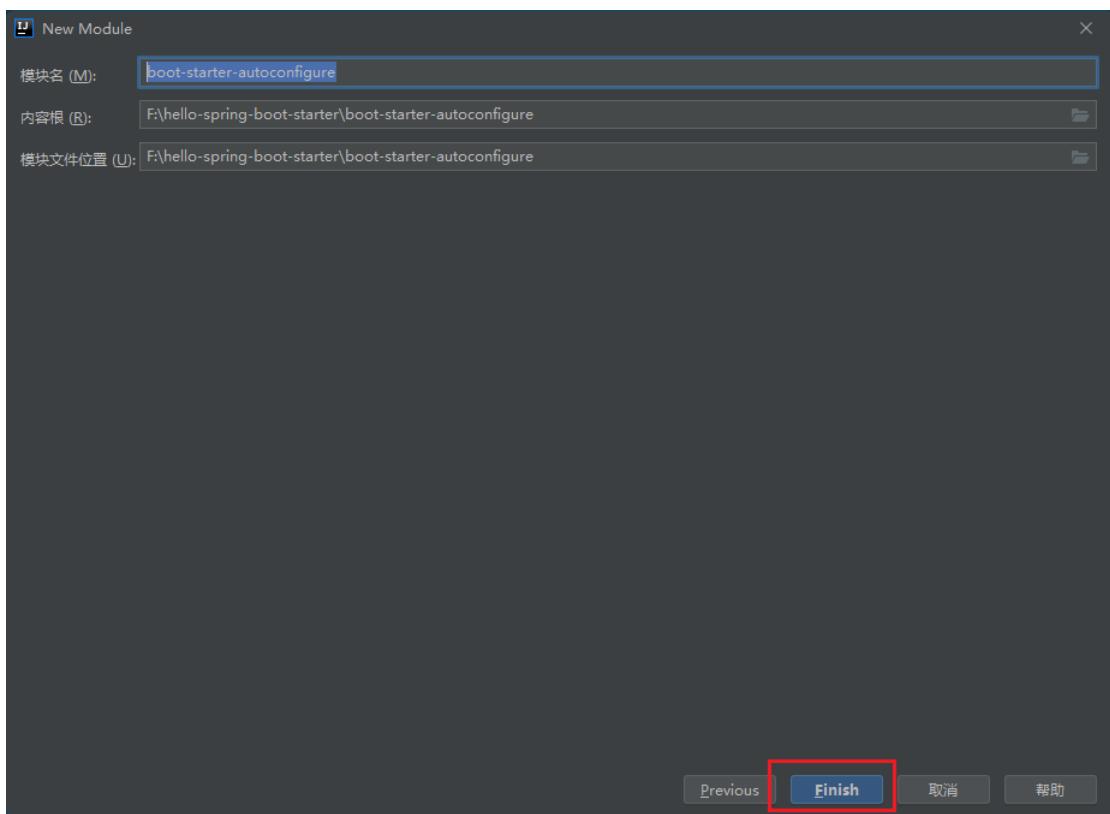




(二)、创建一个 SpringBoot 工程



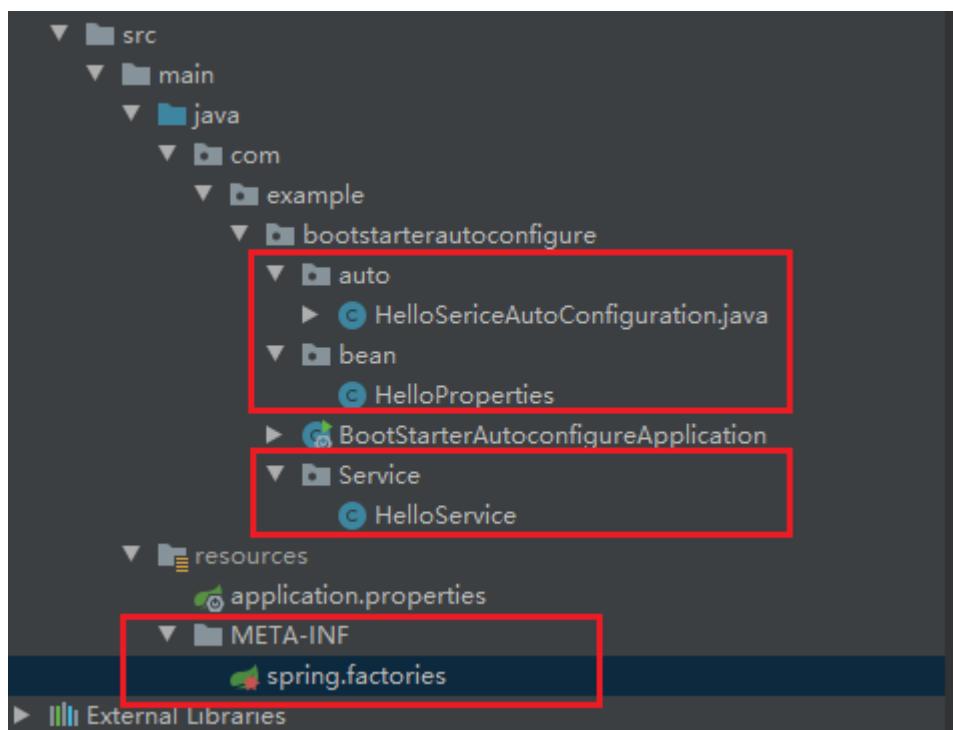




(三)、普通 Maven 引入 SpringBoot 的依赖使其导入 Maven 自动导入 SpringBoot

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>boot-starter-autoconfigure</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

(四)、AutoConfiguration 项目创建四个文件自动配置文件



(五)、配置 HelloServiceAutoConfiguration 类

```
//Spring 配置类
@Configuration
//绑定类并注册到容器中
@EnableConfigurationProperties(HelloProperties.class)
public class HelloServiceAutoConfiguration {

    //如果容器中没有该类就添加
    @ConditionalOnMissingBean(HelloService.class)
    @Bean
    public HelloService helloService(){
        return new HelloService();
    }
}
```

(六)、HelloProperties 类(用于绑定配置文件装入值)

```
@ConfigurationProperties("hello")  
public class HelloProperties {  
  
    private String qianzhi;  
    private String houzhi;  
  
    public String getQianzhi() {  
        return qianzhi;  
    }  
  
    public HelloProperties(String qianzhi) {  
        this.qianzhi = qianzhi;  
    }  
  
    public String getHouzhi() {  
        return houzhi;  
    }  
  
    public void setHouzhi(String houzhi) {  
        this.houzhi = houzhi;  
    }  
}
```

(七)、HelloService(主要功能是给容器中添加组件实现业务)

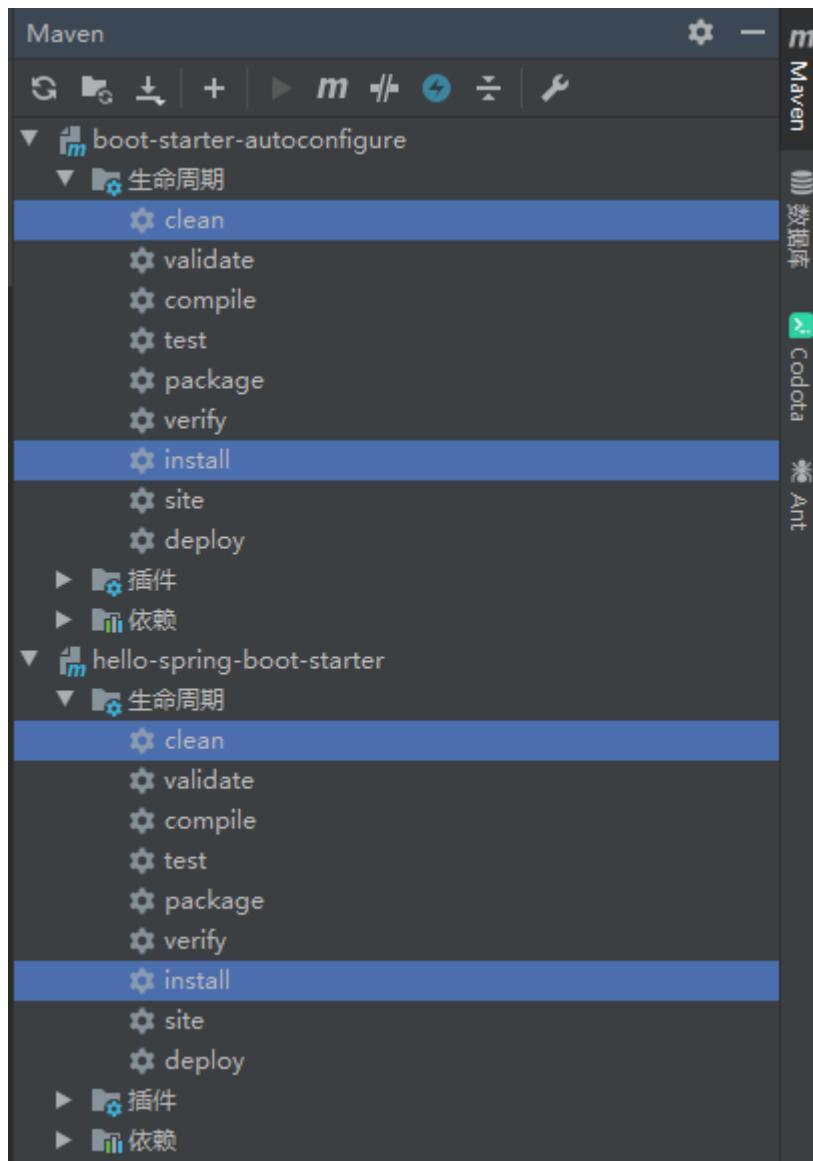
```
public class HelloService {  
  
    @Autowired  
    private HelloProperties helloProperties;
```

```
public String sayHello(String name){  
    return helloProperties.getQianzhi()+name+helloProperties.getHouzhi();  
}  
}
```

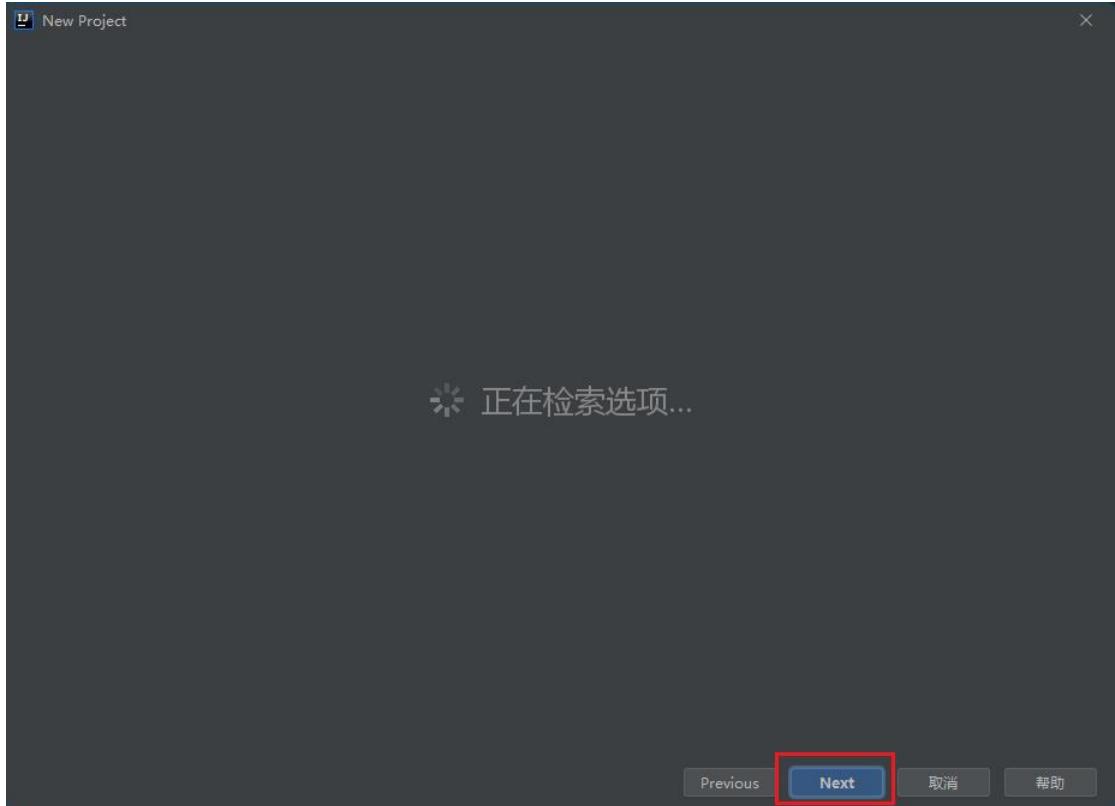
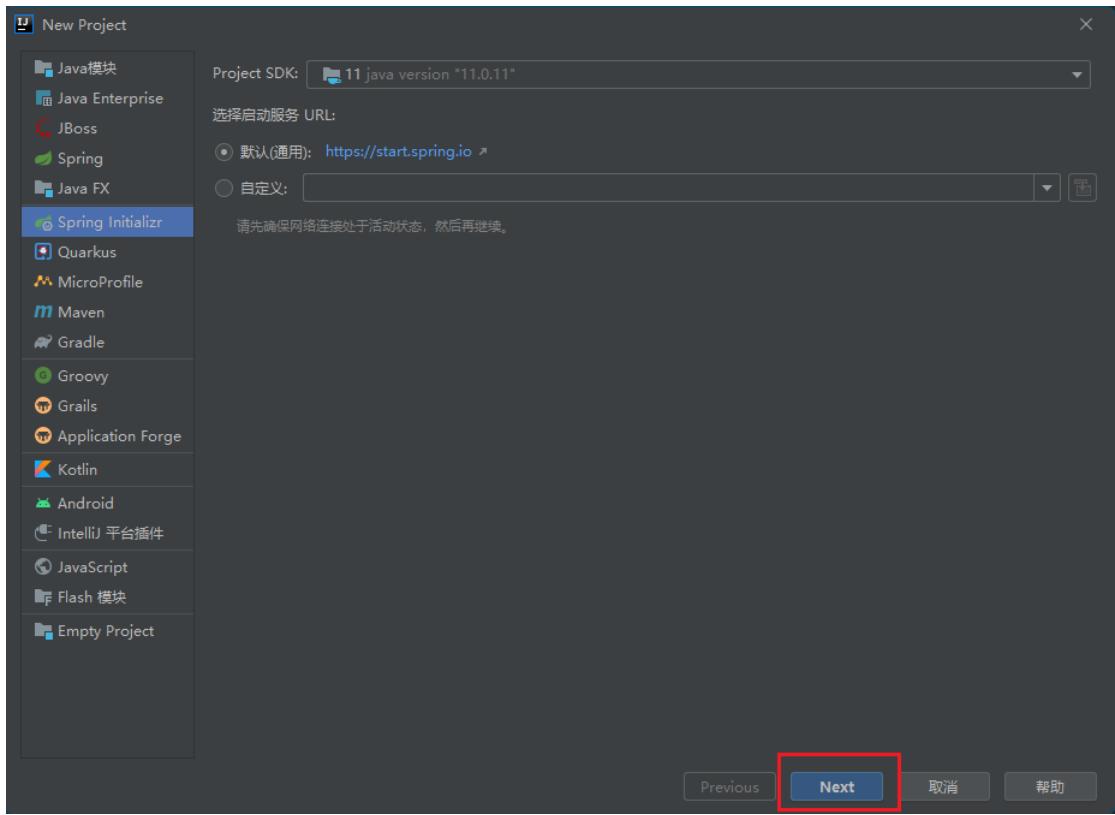
(八)、设置其他项目导入依赖自动启动 autoConfiguration 类
在 META-INF 下的 spring.factories 中添加

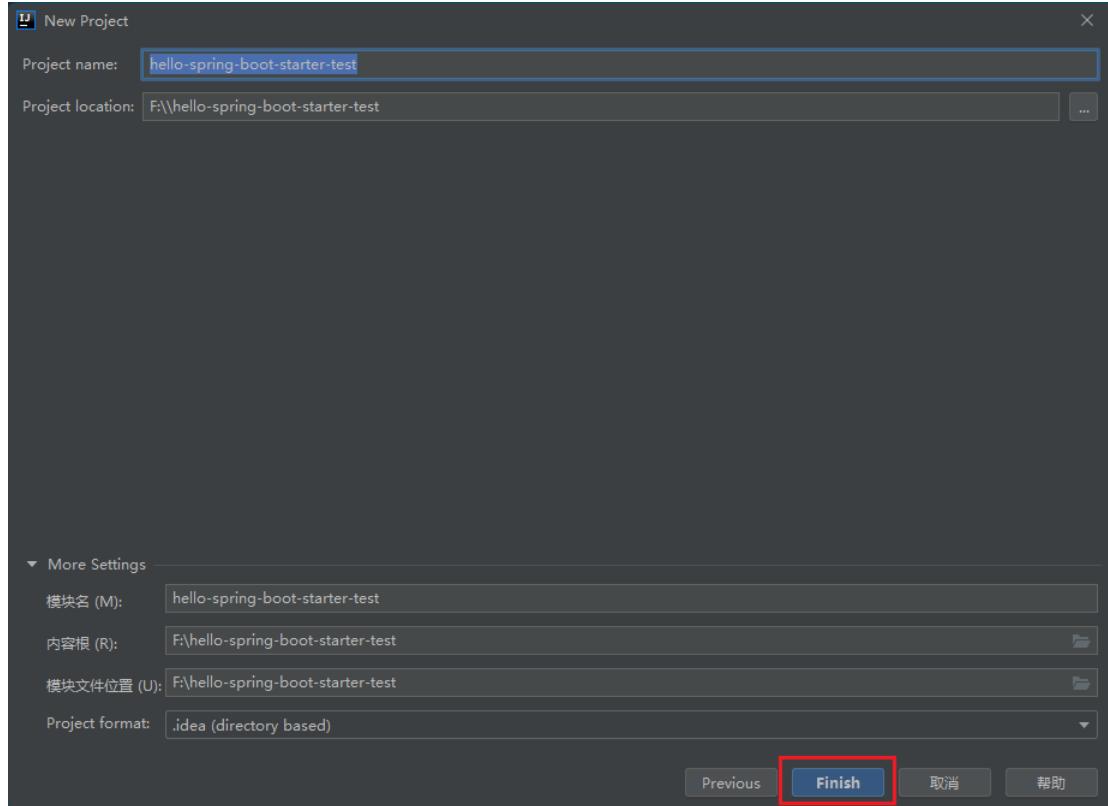
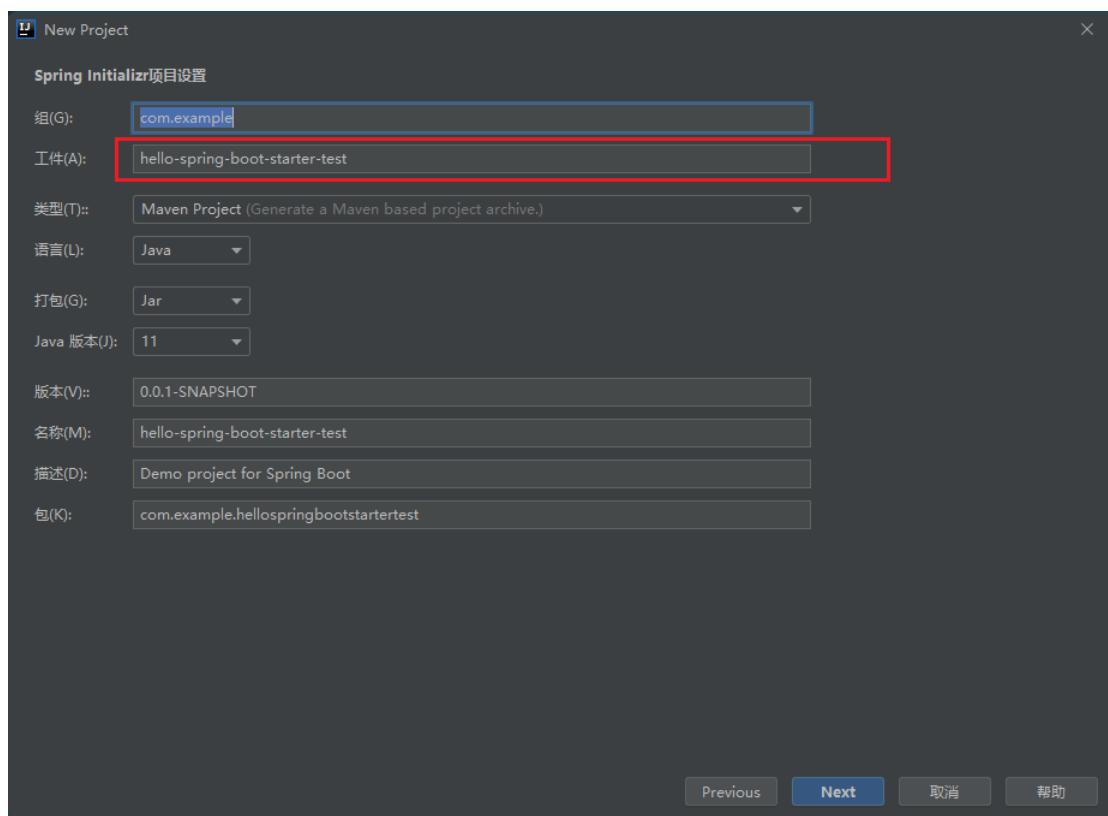
```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
com.example.bootstarterautoconfigure.auto.HelloServiceAutoConfiguration.java
```

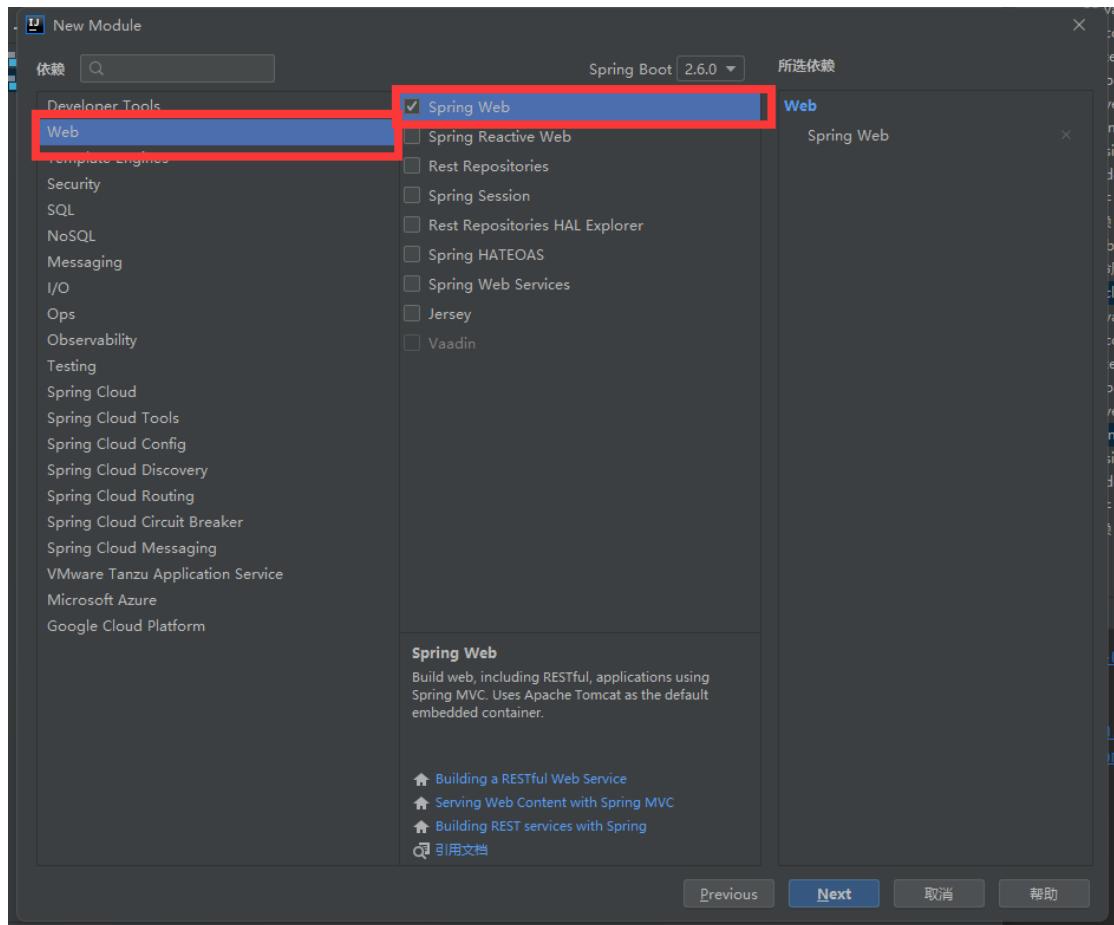
(九)、普通 Maven 和自动配置类打包在本地仓库



(八)、创建一个 SpringBoot 测试

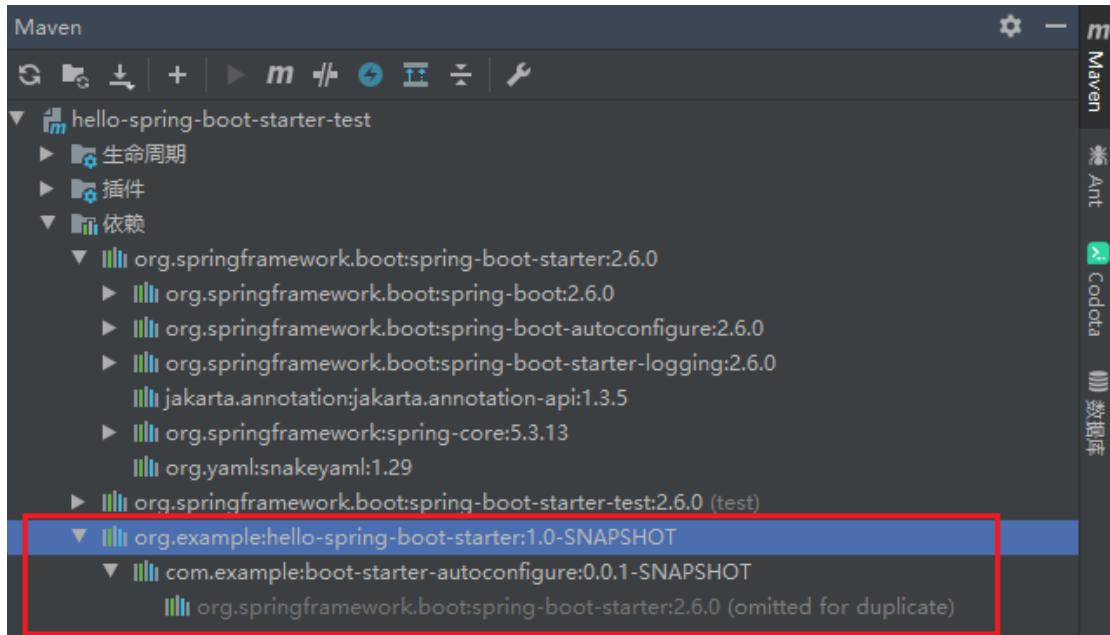






(八)、测试 SpringBoot 导入普通 Maven 依赖

```
<dependency>
    <groupId>org.example</groupId>
    <artifactId>hello-spring-boot-starter</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```



(九)、给自己的 starter 的 HelloProperties 赋值

```
hello.qianzhi=aaa
```

```
hello.houzhi=bbb
```

(十)、使用 Controller 测试是否成功

```
@Controller
```

```
public class StartTestController {
```

```
    @Autowired
```

```
    HelloService helloService;
```

```
    @ResponseBody
```

```
    @GetMapping("/startertest")
```

```
    public String login(){
```

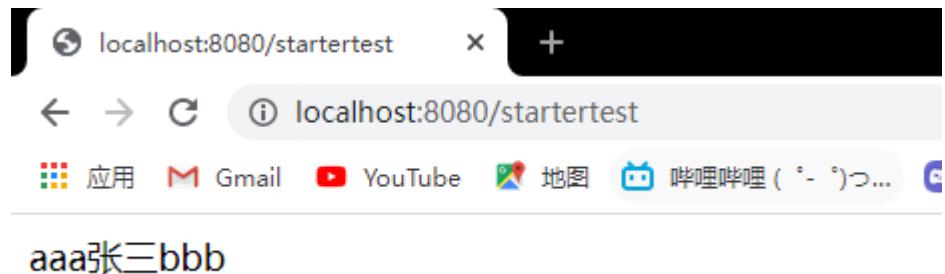
```
        String a = "aa";
```

```
        return helloService.sayHello("张三");
```

```
    }
```

```
}
```

(十一)、网页测试



二十五、Springboot2 创建启动流程

(一)、创建过程

底层的一个方法会去 `spring.factories` 文件找引导器, 初始化器 监听器, 然后通过

```
private Class<?> deduceMainApplicationClass() {  
    try {  
        StackTraceElement[] stackTrace = new RuntimeException().getStackTrace();  
        for (StackTraceElement stackTraceElement : stackTrace) {  
            if ("main".equals(stackTraceElement.getMethodName())) {  
                return Class.forName(stackTraceElement.getClassName());  
            }  
        }  
    }  
}
```

循环遍历栈找到第一个 main 方法找到启动器

二十六、Springboot 日志

(一)、SpringBoot 日志级别

trace	都输出
debug	调试中输出

info	一些想要的信息
warn	警告
error	报错

(二)、SpringBoot 默认级别

SpringBoot 默认级别为 info

```
2021-12-13 15:23:02.922 INFO 7676 --- [           main] c.m.houtai.HoutaiApplicationTests      : ws info
2021-12-13 15:23:02.922  WARN 7676 --- [           main] c.m.houtai.HoutaiApplicationTests      : ws warn
2021-12-13 15:23:02.922 ERROR 7676 --- [          main] c.m.houtai.HoutaiApplicationTests      : ws error

2021-12-13 15:23:02.940 INFO 7676 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Shutdown initiated...
2021-12-13 15:23:02.949 INFO 7676 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Shutdown completed.

Process finished with exit code 0
```

(三)、修改日志默认级别

在配置文件配置 com.myspringboot 下的类日志级别为 trace

```
logging.level.com.myspringboot=trace
```

```
2021-12-13 15:23:34.895 TRACE 15868 --- [           main] c.m.houtai.HoutaiApplicationTests      : ws trace
2021-12-13 15:23:34.895 DEBUG 15868 --- [          main] c.m.houtai.HoutaiApplicationTests     : ws debug
2021-12-13 15:23:34.895  INFO 15868 --- [          main] c.m.houtai.HoutaiApplicationTests     : ws info
2021-12-13 15:23:34.895  WARN 15868 --- [          main] c.m.houtai.HoutaiApplicationTests     : ws warn
2021-12-13 15:23:34.895 ERROR 15868 --- [          main] c.m.houtai.HoutaiApplicationTests     : ws error

2021-12-13 15:23:34.914 INFO 15868 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Shutdown initiated...
2021-12-13 15:23:34.922 INFO 15868 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Shutdown completed.
```

(四)、日志的其他设置

```
logging.level.com.myspringboot=trace
```

logging.pattern.console=控制台输出格式

logging.pattern.file=日志输出到文件设置

logging.file.name=指定日志输出的文件(可以绝对路径,也可以相对路径)

logging.file.path=和 file.name 差不多,同时存在使用 file.name

(五)、指定日志配置文件

在根路径下创建对应的文件名

Logging System	Customization
Logback	<code>logback-spring.xml</code> , <code>logback-spring.groovy</code> , <code>logback.xml</code> or <code>logback.groovy</code>
Log4j2	<code>log4j2-spring.xml</code> or <code>log4j2.xml</code>
JDK (Java Util Logging)	<code>logging.properties</code>

如果配置配置文件不加-spring 就被日志框架解析,没有 spring 的功能

比如说 :logback 日志 , 你配置文件是 `logback.xml` 就没有 `springboot` 的功能,如果配置文件名为:`logback-spring.xml` 就会被 `SpringBoot` 解析,就可以使用 `springboot` 的强大功能

二十七、集成 redis

(一)、添加依赖

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

(二)、配置连接源

```
#Redis 服务器地址
spring.redis.host=192.168.2.166

#Redis 服务器连接端口
spring.redis.port=6379

#Redis 数据库索引（默认为 0）
spring.redis.database=0

#连接超时时间（毫秒）
spring.redis.timeout=1800000

#连接池最大连接数（使用负值表示没有限制）
```

```
spring.redis.lettuce.pool.max-active=20
#最大阻塞等待时间(负数表示没限制)
spring.redis.lettuce.pool.max-wait=-1
#连接池中的最大空闲连接
spring.redis.lettuce.pool.max-idle=5
#连接池中的最小空闲连接
spring.redis.lettuce.pool.min-idle=0
```

(三) 存取值

```
@Controller
public class RedisController {

    @Autowired
    private RedisTemplate<Object, Object> redisTemplate;

    //装入值
    @GetMapping(value = "/put")
    @ResponseBody
    public String putDate(String key, String value){
        redisTemplate.opsForValue().set(key, value);
        return "值已存入 redis 中";
    }

    //取值
    @GetMapping(value = "/get")
    @ResponseBody
    public Object getDate(String key){
        return "值=" + key + "-----value=" + redisTemplate.opsForValue().get(key);
    }
}
```

