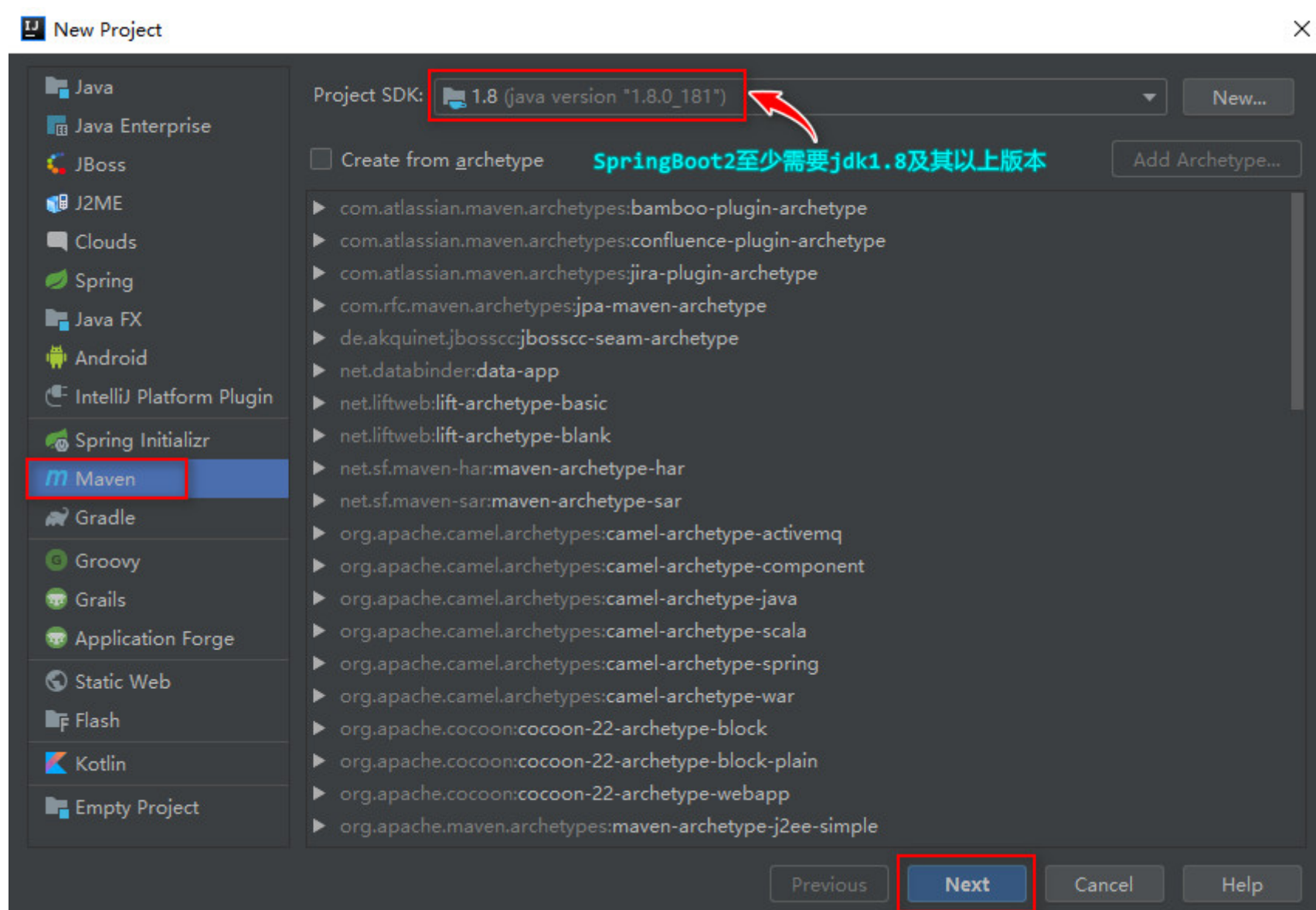
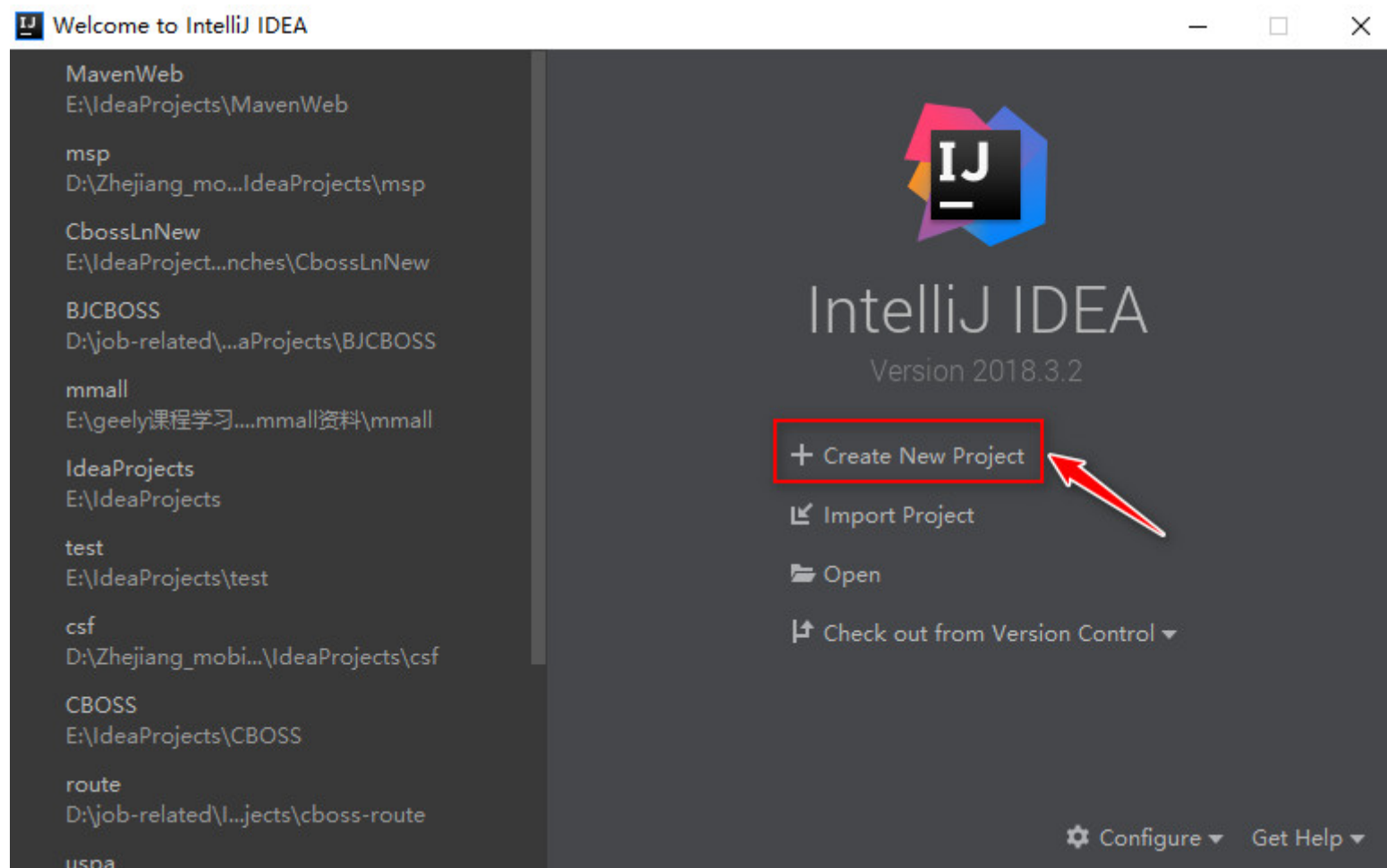


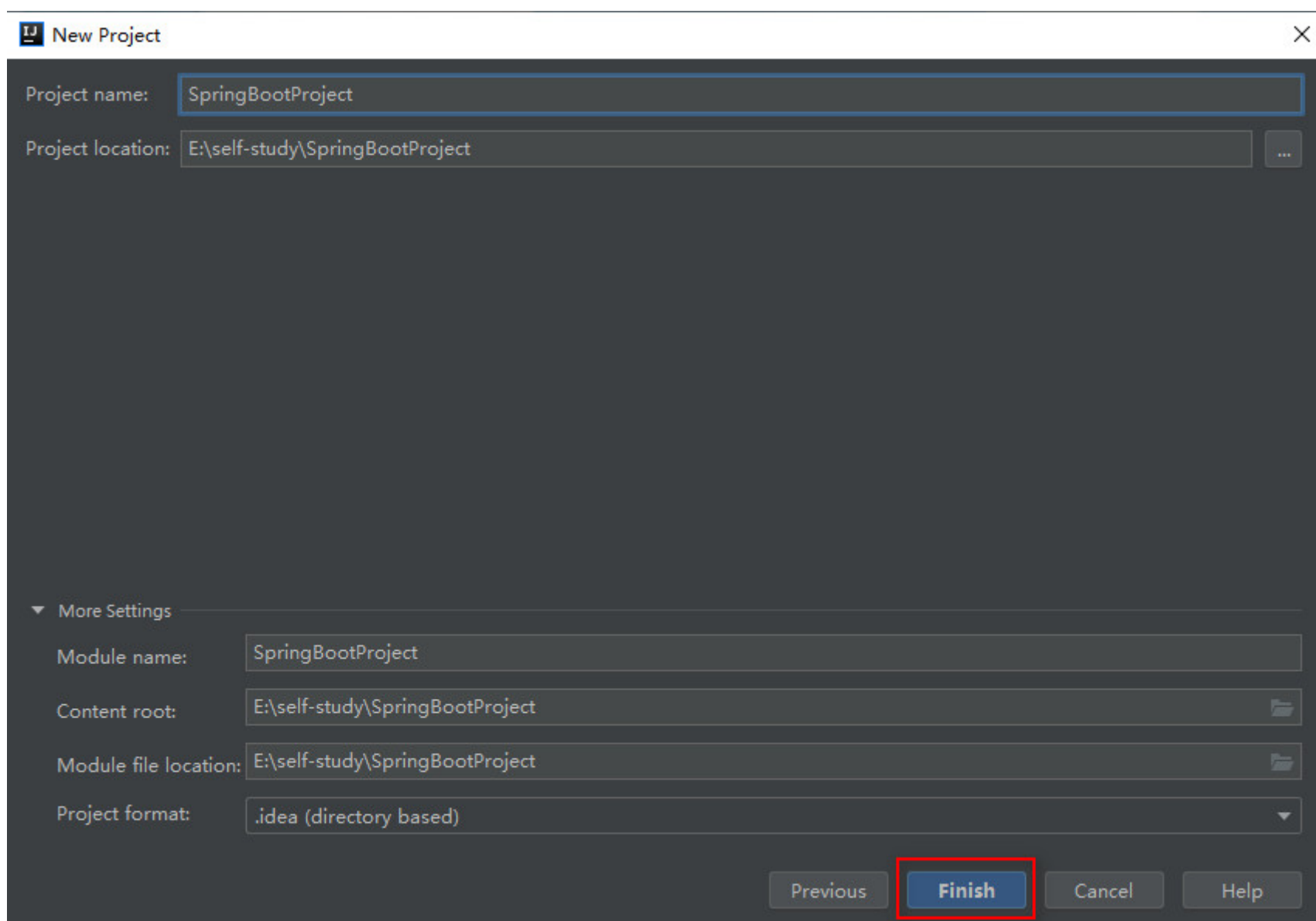
1.编写一个HelloWorld

1.1 版本问题

SpringBoot2要求jdk至少要是1.8或者以上版本，maven至少3.3以上版本。

1.2 新建一个maven项目





1.3 配置maven的settings.xml文件

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
5      http://maven.apache.org/xsd/settings-1.0.0.xsd">
6
7      <localRepository>E:\self-study\mvn_repository</localRepository>
8
9      <mirrors>
10         <mirror>
11             <id>nexus-aliyun</id>
12             <name>aliyun maven</name>
13             <url>http://maven.aliyun.com/nexus/content/groups/public</url>
14             <mirrorOf>central</mirrorOf>
15         </mirror>
16     </mirrors>
17
18     <profiles>
19         <profile>
20             <id>jdk-1.8</id>
21             <activation>
22                 <activeByDefault>true</activeByDefault>
23                 <jdk>1.8</jdk>
24             </activation>
25             <properties>
```

```

25         <maven.compiler.source>1.8</maven.compiler.source>
26         <maven.compiler.target>1.8</maven.compiler.target>
27         <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
28     </properties>
29 </profile>
30 </profiles>
31 </settings>

```

1.4 pom.xml文件中引入依赖

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.gongsl</groupId>
8     <artifactId>SpringBootProject</artifactId>
9     <version>1.0-SNAPSHOT</version>
10
11     <parent>
12         <groupId>org.springframework.boot</groupId>
13         <artifactId>spring-boot-starter-parent</artifactId>
14         <version>2.3.7.RELEASE</version>
15     </parent>
16
17     <dependencies>
18         <dependency>
19             <groupId>org.springframework.boot</groupId>
20             <artifactId>spring-boot-starter-web</artifactId>
21         </dependency>
22     </dependencies>
23
24     <!-- spring-boot-maven-plugin可以为我们创建一个可执行的jar -->
25     <build>
26         <plugins>
27             <plugin>
28                 <groupId>org.springframework.boot</groupId>
29                 <artifactId>spring-boot-maven-plugin</artifactId>
30             </plugin>
31         </plugins>
32     </build>
33
34 </project>

```

1.5 创建启动类

```

1 package com.gongsl;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 /**
7  * 主程序类
8  * @Author: gongsl
9  * @Date: 2021-01-08 18:18
10 */
11 @SpringBootApplication

```

```
12 public class MainApplication {
13     public static void main(String[] args) {
14         SpringApplication.run(MainApplication.class, args);
15     }
16 }
```

启动类中的run方法是有返回值的，返回值就是IOC容器。

1.6 编写业务代码

```
1 package com.gongsl.controller;
2
3 import org.springframework.web.bind.annotation.RequestMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 /**
7  * @Author: gongsl
8  * @Date: 2021-01-08 18:19
9  */
10 @RestController
11 public class HelloController {
12
13     @RequestMapping("/hello")
14     public String home(){
15         return "Hello SpringBoot2!";
16     }
17 }
```

1.7 运行测试

执行 `MainApplication` 类的 `main` 方法，然后浏览器地址栏输入 `http://localhost:8080/hello` 进行访问即可。

1.8 修改端口启动

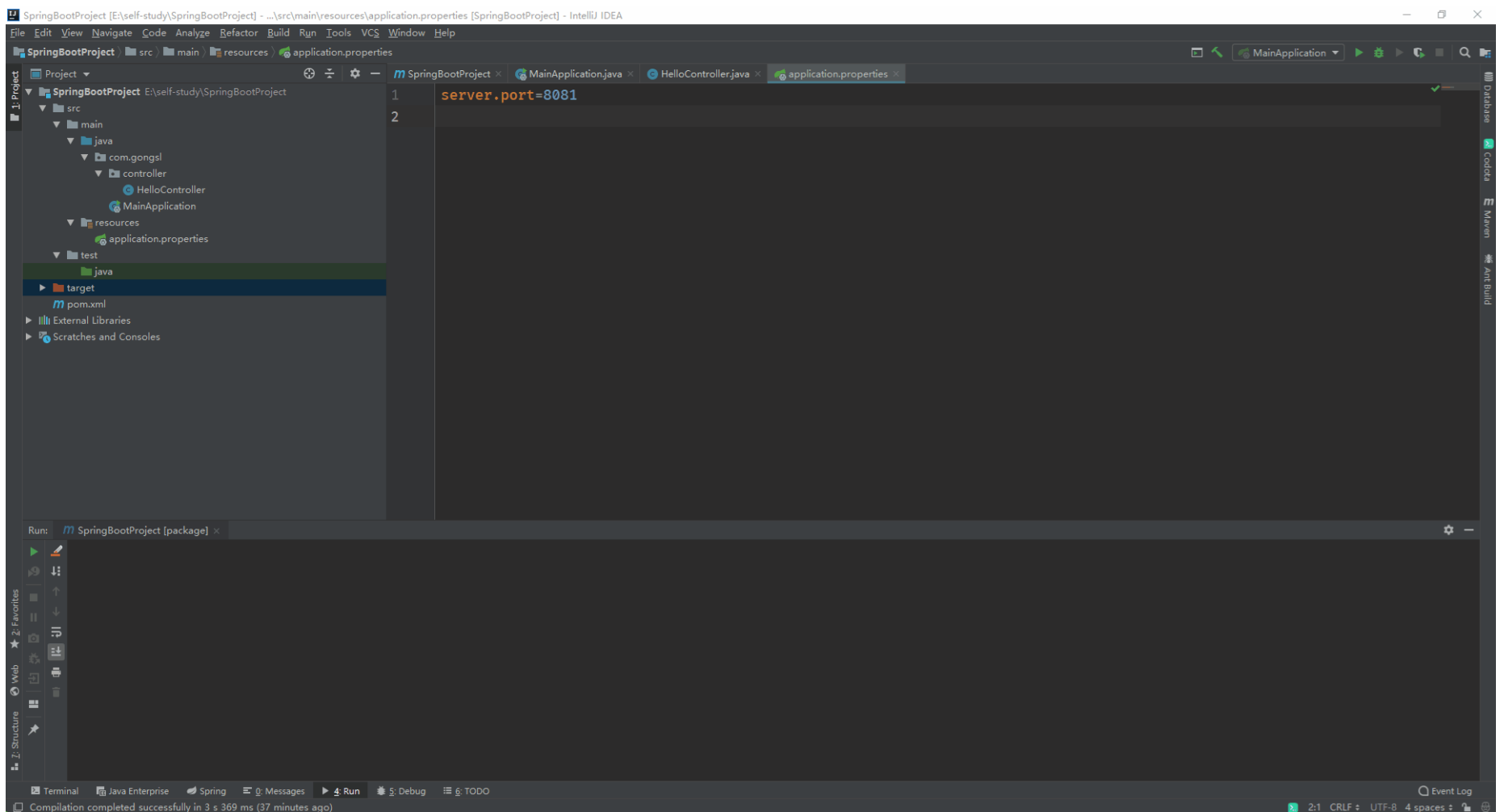
在maven项目的 `src/main/resources` 目录下新增 `application.properties` 配置文件，然后在文件中新增以下内容：

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

这时端口就已经被修改成8081了，然后重新运行启动类的 `main` 方法，最后在浏览器地址栏输入 `http://localhost:8081/hello` 进行访问即可。

1.9 通过cmd窗口启动项目

如果我们在pom.xml文件中加了 `spring-boot-maven-plugin` 这个依赖的话，那么使用maven打成的包就是一个可执行的jar包。我们可以在cmd窗口中进行演示，这里需要用到 `java -jar` 命令，如下所示：



1.10 注意事项

1. 如果浏览器地址栏访问url出现问题，首先检查是不是 `MainApplication` 启动类的位置有问题。该类一定要放在最外侧，要保证能够扫描到所有的controller,因为SpringBoot会自动加载启动类所在包下以及其子包下的所有组件；
2. 如果我们不想把启动类放在最外侧，或者想要自定义被扫描的包及其下的controller类的话，也是可以实现的，只需要在启动类的 `@SpringBootApplication` 注解中增加 `scanBasePackages` 参数即可，比如扫描 `com.gongsl` 包下的所有内容，就配置为 `@SpringBootApplication(scanBasePackages = "com.gongsl")` 即可；

- 1 如果启动类的包路径是“com.gongsl”的话，那么
- 2 `@SpringBootApplication`
- 3 等同于
- 4 `@SpringBootConfiguration`
- 5 `@EnableAutoConfiguration`
- 6 `@ComponentScan("com.gongsl")`

3. 上面 `HelloController` 类上用的是 `@RestController` 注解，这个注解就是 `@Controller` 和 `@ResponseBody` 的组合注解；
4. 如果想知道 `application.properties` 配置文件中可以写哪些东西，那么我们可以参考[官方文档](#)。

2.SpringBoot简介

2.1 为什么使用SpringBoot

- SpringBoot能快速创建出生产级别的Spring应用。

2.2 SpringBoot的优点

- 可以创建独立的Spring应用；
- 内嵌了web服务器；
- 自动starter依赖，简化了构建配置；
- 自动配置Spring以及第三方功能；
- 提供生产级别的监控、健康检查及外部化配置；
- 无代码生成、无需编写XML文件等。

SpringBoot是整合Spring技术栈的一站式框架，也是简化Spring技术栈的快速开发脚手架。

2.3 SpringBoot的缺点

- 人称版本帝，迭代快，需要时刻关注变化；
- 封装太深，内部原理复杂，不容易精通。

2.4 SpringBoot的时代背景

2.4.1 微服务

- 微服务是一种架构风格；
- 微服务是一个应用拆分为一组小型服务；
- 每个服务运行在自己的进程内，也就是可独立部署和升级；
- 服务之间使用轻量级HTTP进行交互；
- 服务围绕业务功能拆分；
- 服务可以由全自动部署机制独立部署；
- 去中心化，服务自治。服务可以使用不同的语言、不同的存储技术。

2.4.2 分布式

2.4.2.1 分布式的困难

- 远程调用
- 服务发现
- 负载均衡
- 服务容错
- 配置管理
- 服务监控
- 链路追踪
- 日志管理
- 任务调度
-

2.4.2.2 分布式的解决方案

- SpringBoot + SpringCloud

2.4.3 云原生

2.4.3.1 上云的困难

- 服务自愈
- 弹性伸缩
- 服务隔离
- 自动化部署
- 灰度发布
- 流量治理
-

2.4.3.2 上云的解决方案

- Cloud Native

2.5 SpringBoot的特点

2.5.1 依赖管理

2.5.1.1 父项目做依赖管理

```

1  <!-- 通过spring-boot-starter-parent这个父项目做依赖管理 -->
2  <parent>
3      <groupId>org.springframework.boot</groupId>
4      <artifactId>spring-boot-starter-parent</artifactId>
5      <version>2.3.7.RELEASE</version>
6  </parent>
7
8  <!-- spring-boot-starter-parent这个父项目中还有下面这个父项目 -->
9  <parent>
10     <groupId>org.springframework.boot</groupId>
11     <artifactId>spring-boot-dependencies</artifactId>
12     <version>2.3.7.RELEASE</version>
13 </parent>

```

spring-boot-dependencies 这个父项目中几乎声明了所有开发中常用的依赖的版本号,这个就是自动版本仲裁机制。

2.5.1.2 导入starter作为场景启动器

- 我们以后在pom.xml文件中会见到很多 **spring-boot-starter-*** 这种配置,这个就是场景启动器,后面的*号就代表了某种场景;
- 只要引入了某个场景的starter,这个场景的所有常规需要的依赖都会被自动引入;
- 如果我们想知道SpringBoot支持哪些场景的starter,可以到[官方文档](#)中进行查阅;
- 像 ***-spring-boot-starter** 这种starter一般都是第三方为我们提供的简化开发的场景启动器;
- 所有场景启动器最底层的依赖就是下面这个。

```

1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter</artifactId>
4     <version>2.3.7.RELEASE</version>
5     <scope>compile</scope>
6 </dependency>

```

2.5.1.3 自动版本仲裁

- 自动版本仲裁功能可以让我们无需关注版本号,因为在 **spring-boot-dependencies** 这个父项目中,所有涉及的依赖的版本号都已经配置在了 **properties** 标签下了,所以我们引入依赖时默认可以不写版本号;
- 不过引入非版本仲裁的jar包时,要写版本号。

2.5.1.4 能够修改默认版本号

我们也可以修改默认版本号。首先要在 **spring-boot-dependencies** 这个父项目中查看需要修改版本号的依赖的标签名是什么,比如mysql版本号对应的标签名就是 **mysql.version**,假设我们想修改mysql的版本号为 **5.1.43**,那就直接在项目的pom.xml文件中加入如下内容即可。

```

1 <properties>
2     <mysql.version>5.1.43</mysql.version>
3 </properties>

```

2.5.2 自动配置

假设就以 **spring-boot-starter-web** 这个依赖为例。

- 只要引入了上面那个web依赖,SpringBoot就会为我们自动引入tomcat依赖,自动给我们配置好tomcat;
- 同样会引入SpringMVC的全套组件,并自动配好SpringMVC的常用功能;
- 还会自动配好Web常见功能,比如字符编码问题等,并帮我们配置好了所有web开发的常见场景;
- SpringBoot也会为我们配置一个默认的包扫描路径,不用再手动进行包扫描的配置;
- SpringBoot为我们提供的各种配置都会赋予一个默认值
 - 默认配置最终都是映射到某个类上,如: `MultipartProperties`;

- 配置文件的值最终会绑定某个类上，这个类会在容器中创建对象。
- SpringBoot是按需加载所有自动配置项，我们引入了哪些场景，这个场景的自动配置才会开启；
- SpringBoot所有的自动配置功能都在 `spring-boot-autoconfigure` 包里面，比如 `spring-boot-autoconfigure-2.3.7.RELEASE.jar` 包。

3. 常用注解的用法

3.1 @Bean注解和@Configuration注解

3.1.1 前置准备

3.1.1.1 引入依赖

```
1 <dependency>
2     <groupId>org.projectlombok</groupId>
3     <artifactId>lombok</artifactId>
4 </dependency>
```

这里引入lombok依赖时不用加版本号，因为SpringBoot的自动版本仲裁功能已经为我们设置好了对应的版本号，我们可以在 `spring-boot-dependencies` 这个父项目中根据 `lombok.version` 标签查看到对应的版本号是什么。

3.1.1.2 增加两个实体类

```
1 package com.gongsl.bean;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.ToString;
6
7 /**
8  * @Author: gongsl
9  * @Date: 2021-01-11 21:27
10  */
11 @Data
12 @ToString
13 @AllArgsConstructor
14 public class User {
15     private String name;
16     private Integer age;
17     private String gender;
18 }
```

```
1 package com.gongsl.bean;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.ToString;
6
7 /**
8  * @Author: gongsl
9  * @Date: 2021-01-11 21:33
10  */
11 @Data
12 @ToString
13 @AllArgsConstructor
14 public class Pet {
15     private String name;
16 }
```


3.1.1.3 新增配置类

```
1 package com.gongsl.config;
2
3 import com.gongsl.bean.Pet;
4 import com.gongsl.bean.User;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7
8 /**
9  * @Author: gongsl
10  * @Date: 2021-01-11 21:35
11  */
12 @Configuration
13 public class MyConfig {
14
15     @Bean
16     public User userTest(){
17         return new User("张三",18,"男");
18     }
19
20     @Bean
21     public Pet petCat(){
22         return new Pet("汤姆猫");
23     }
24 }
```

3.1.2 @Bean、@Configuration的用法

- 之前Spring想要将某个组件放到IOC容器中，一般会在一个xml配置文件中进行配置，而SpringBoot不再使用配置文件，而是使用配置类，我们可以新建一个类，比如 `MyConfig` 类，然后只要我们在这个类上加上 `@Configuration` 注解，那这个类就是配置类，这个类等同于之前我们使用的配置文件；
- 以前在配置文件中，我们可以使用 `bean` 标签来给容器添加组件，现在我们在配置类中的方法上使用 `@Bean` 注解可以完成同样的效果；
- 以上面的 `userTest` 方法为例，该方法使用了 `@Bean` 注解，就表示向容器中添加了一个组件，方法名就是组件id，方法的返回类型就是组件的类型，方法的返回值就是组件在容器中的实例；
- 如果我们不希望方法名作为容器中组件的id的话，也可以自定义，比如把上面配置类中 `userTest` 方法的 `@Bean` 注解改成 `@Bean("user")`，就把组件id改为"user"了，我们可以在启动类中进行验证，如下所示：

配置类：

```
1 package com.gongsl.config;
2
3 import com.gongsl.bean.Pet;
4 import com.gongsl.bean.User;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7
8 /**
9  * @Author: gongsl
10  * @Date: 2021-01-11 21:35
11  */
12 @Configuration
13 public class MyConfig {
14
15     @Bean("user")
16     public User userTest(){
17         return new User("张三",18,"男");
18     }
19 }
```

```

20     @Bean
21     public Pet petCat(){
22         return new Pet("汤姆猫");
23     }
24 }

```

启动类:

```

1  package com.gongsl;
2
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.SpringBootApplication;
5  import org.springframework.context.ConfigurableApplicationContext;
6
7  /**
8   * 主程序类
9   * @Author: gongsl
10  * @Date: 2021-01-08 18:18
11  */
12  @SpringBootApplication
13  public class MainApplication {
14      public static void main(String[] args) {
15          //返回的是IOC容器
16          ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class,
args);
17
18          //获取所有组件id的名称
19          String[] beanNames = run.getBeanDefinitionNames();
20          for (String name : beanNames) {
21              if("user".equals(name) || "petCat".equals(name)){
22                  System.out.print(name+" ");
23              }
24          }
25      }
26  }
27
28  //运行结果: user petCat

```

- 配置类中使用 `@Bean` 注解在方法上给容器注入的组件，默认是单实例的。由于是单实例，所以容器中同一个实例无论怎么获取，获取多少次，都是一样的，验证如下：

```

1  package com.gongsl;
2
3  import com.gongsl.bean.Pet;
4  import org.springframework.boot.SpringApplication;
5  import org.springframework.boot.autoconfigure.SpringBootApplication;
6  import org.springframework.context.ConfigurableApplicationContext;
7
8  /**
9   * 主程序类
10  * @Author: gongsl
11  * @Date: 2021-01-08 18:18
12  */
13  @SpringBootApplication
14  public class MainApplication {
15      public static void main(String[] args) {
16          //返回的是IOC容器
17          ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class,
args);

```

```

18
19     Pet pet1 = run.getBean(Pet.class);
20     Pet pet2 = run.getBean(Pet.class);
21     System.out.println(pet1==pet2); //运行结果: true
22
23     //通过组件id更精确地获取
24     Pet petCat1 = run.getBean("petCat", Pet.class);
25     Pet petCat2 = run.getBean("petCat", Pet.class);
26     System.out.println(petCat1==petCat2); //运行结果: true
27 }
28 }

```

- 配置类本身也是容器中的一个组件。如果不是的话，当我们从容器中获取的时候是会报错的，验证如下：

```

1 package com.gongsl;
2
3 import com.gongsl.config.MyConfig;
4 import org.springframework.boot.SpringApplication;
5 import org.springframework.boot.autoconfigure.SpringBootApplication;
6 import org.springframework.context.ConfigurableApplicationContext;
7
8 /**
9  * 主程序类
10  * @Author: gongsl
11  * @Date: 2021-01-08 18:18
12  */
13 @SpringBootApplication
14 public class MainApplication {
15     public static void main(String[] args) {
16         //返回的是IOC容器
17         ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class,
18             args);
19
20         MyConfig bean = run.getBean(MyConfig.class);
21         System.out.println(bean);
22     }
23 }
24 //运行结果: com.gongsl.config.MyConfig$$EnhancerBySpringCGLIB$$33e871dd@55f45b92

```

3.1.3 proxyBeanMethods属性的用法

1. 和SpringBoot1相比，SpringBoot2中的 `@Configuration` 注解新增了一个 `proxyBeanMethods` 属性(这个属性是Spring5.2版本以后才有的，SpringBoot2中Spring的版本就是5.2)，该属性的默认值是true；
2. 外部无论对配置类中已经使用 `@Bean` 注解注入到容器中的组件方法调用多少次，获取到的都是已经注入到容器中的单实例对象，之所以出现这种情况，就是因为配置类的 `@Configuration` 注解中 `proxyBeanMethods` 属性的默认值是true的原因；
3. 当配置类的 `@Configuration` 注解中 `proxyBeanMethods` 属性为true时，如果我们直接从容器中获取配置类，可以发现，获取到的结果是 `com.gongsl.config.MyConfig$$EnhancerBySpringCGLIB$$33e871dd@55f45b92`。从结果中的 `EnhancerBySpringCGLIB` 可以发现，`MyConfig` 类并不是一个普通的配置类，而是一个被SpringCGLIB增强了的代理对象；
4. 当 `proxyBeanMethods` 属性为true时，如果我们调用配置类中使用过 `@Bean` 注解的方法，比如 `petCat()` 方法，那么SpringBoot总会检查该方法是否已经被注入到容器中，如果发现容器中有该方法返回的组件，那么就直接从容器中获取，这时候无论我们调用多少次 `petCat()` 方法，返回的Pet对象都是同一个，即之前放入容器中的那一个；
5. 如果 `proxyBeanMethods` 属性为false，那么当我们调用 `petCat()` 方法时，SpringBoot就不会再检查该方法是否已经被注入到容器中，自然也就不会再从容器中获取组件，而是直接调用该方法，由于 `petCat()` 方法中的逻辑就是创建一个Pet对象，所以这时每调用一次 `petCat()` 方法就会创建一个新的Pet对象，这样每次调用 `petCat()` 方法返回的Pet对象就都不相同了，代码验证如下：

配置类:

```
1 package com.gongsl.config;
2
3 import com.gongsl.bean.Pet;
4 import com.gongsl.bean.User;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7
8 /**
9  * @Author: gongsl
10  * @Date: 2021-01-11 21:35
11  */
12 @Configuration(proxyBeanMethods = false)
13 public class MyConfig {
14
15     @Bean("user")
16     public User userTest(){
17         return new User("张三",18,"男");
18     }
19
20     @Bean
21     public Pet petCat(){
22         return new Pet("汤姆猫");
23     }
24 }
```

启动类:

```
1 package com.gongsl;
2
3 import com.gongsl.bean.Pet;
4 import com.gongsl.config.MyConfig;
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7 import org.springframework.context.ConfigurableApplicationContext;
8
9 /**
10  * 主程序类
11  * @Author: gongsl
12  * @Date: 2021-01-08 18:18
13  */
14 @SpringBootApplication
15 public class MainApplication {
16     public static void main(String[] args) {
17         //返回的是IOC容器
18         ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class,
19 args);
20
21         MyConfig bean = run.getBean(MyConfig.class);
22         System.out.println(bean); // 运行结果: com.gongsl.config.MyConfig@437ebf59
23
24         Pet pet1 = bean.petCat();
25         Pet pet2 = bean.petCat();
26         System.out.println(pet1==pet2); // 运行结果: false
27     }
28 }
```

当配置类中的 `proxyBeanMethods` 属性为false时，我们可以发现，pet1和pet2并不是同一个对象，而且从容器中获取的配置类也不带 `EnhancerBySpringCGLIB` 了，而变成了 `com.gongsl.config.MyConfig@437ebf59`。如果我们把配置类中的 `proxyBeanMethods` 属性设置成true的话，上面pet1和pet2就会是同一个对象，运行结果也自然就是true了。

6. 配置类的 `@Configuration` 注解中 `proxyBeanMethods` 属性可以用来解决组件依赖问题，这个属性也引申出了两种模式，分别是Full模式和Lite模式。当proxyBeanMethods属性值为true时，就是Full模式，这个是默认模式，属性值为false时就是Lite模式。如果组件之间存在依赖关系的话，我们要使用Full模式来保证单实例，无依赖关系则可以使用Lite模式。Lite模式可以减少SpringBoot的判断，加速容器启动的过程。

可以将Pet类设置为User类的一个属性来验证组件依赖：

User实体类：

```
1 package com.gongsl.bean;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.ToString;
6
7 /**
8  * @Author: gongsl
9  * @Date: 2021-01-11 21:27
10  */
11 @Data
12 @ToString
13 @AllArgsConstructor
14 public class User {
15     private String name;
16     private Integer age;
17     private String gender;
18     private Pet pet;
19 }
```

配置类：

```
1 package com.gongsl.config;
2
3 import com.gongsl.bean.Pet;
4 import com.gongsl.bean.User;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7
8 /**
9  * @Author: gongsl
10  * @Date: 2021-01-11 21:35
11  */
12 @Configuration(proxyBeanMethods = false)
13 public class MyConfig {
14
15     @Bean("user")
16     public User userTest(){
17         return new User("张三",18,"男",petCat());
18     }
19
20     @Bean
21     public Pet petCat(){
22         return new Pet("汤姆猫");
23     }
24 }
```


启动类：

```
1 package com.gongsl;
2
3 import com.gongsl.bean.Pet;
4 import com.gongsl.bean.User;
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7 import org.springframework.context.ConfigurableApplicationContext;
8
9 /**
10  * 主程序类
11  * @Author: gongsl
12  * @Date: 2021-01-08 18:18
13  */
14 @SpringBootApplication
15 public class MainApplication {
16     public static void main(String[] args) {
17         // 返回的是IOC容器
18         ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class,
19             args);
20         // 从容器中获取User对象
21         User user = run.getBean("user", User.class);
22         // 从容器中获取Pet对象
23         Pet pet = run.getBean("petCat", Pet.class);
24         System.out.println(user.getPet() == pet); // 运行结果: false
25     }
26 }
```

配置类中的 `proxyBeanMethods` 属性设置为了false，所以启动类中运行结果是false。也就是说，从容器中获取的User对象中的pet和容器中的不是一个，这样就无法保证pet的单实例了，那我们把pet注入到容器中就没有意义了，所以遇到这种组件依赖的情况，`proxyBeanMethods` 属性要设置为true才行。

3.2 @Import注解

- 该注解的值默认是一个数组，书写格式为 `@Import({User.class, Pet.class})`，如果数组中只有一个元素，也可以写成 `@Import(User.class)` 这种形式；
- 我们可以使用这个注解给容器中自动创建多个不同类型的组件，每个组件的组件id就是数组中各元素的全类名；
- 我们也可以使用 `@Import` 注解把jar包中的类给注入到容器中；
- 如果想要使用 `@Import` 注解把某个类注入到容器中，那么这个类一定要有无参的构造方法才行。

我们可以把User类、Pet类以及spring-webmvc这个jar包中的ModelAndView类都注入到容器中，演示代码如下：

User类：

```
1 package com.gongsl.bean;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6 import lombok.ToString;
7
8 /**
9  * @Author: gongsl
10  * @Date: 2021-01-11 21:27
11  */
12 @Data
13 @ToString
14 @NoArgsConstructor
```

```
15 @AllArgsConstructor
16 public class User {
17     private String name;
18     private Integer age;
19     private String gender;
20 }
```

Pet类:

```
1 package com.gongsl.bean;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6 import lombok.ToString;
7
8 /**
9  * @Author: gongsl
10  * @Date: 2021-01-11 21:33
11  */
12 @Data
13 @ToString
14 @NoArgsConstructor
15 @AllArgsConstructor
16 public class Pet {
17     private String name;
18 }
```

配置类:

```
1 package com.gongsl.config;
2
3 import com.gongsl.bean.Pet;
4 import com.gongsl.bean.User;
5 import org.springframework.context.annotation.Configuration;
6 import org.springframework.context.annotation.Import;
7 import org.springframework.web.servlet.ModelAndView;
8
9 /**
10  * @Author: gongsl
11  * @Date: 2021-01-11 21:35
12  */
13 @Import({User.class, Pet.class, ModelAndView.class})
14 @Configuration
15 public class MyConfig {
16
17     /*@Bean("user")
18     public User userTest(){
19         return new User("张三",18,"男");
20     }
21
22     @Bean
23     public Pet petCat(){
24         return new Pet("汤姆猫");
25     }*/
26 }
```

启动类:

```

1 package com.gongsl;
2
3 import com.gongsl.bean.Pet;
4 import com.gongsl.bean.User;
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7 import org.springframework.context.ConfigurableApplicationContext;
8 import org.springframework.web.servlet.ModelAndView;
9 import java.util.Arrays;
10
11 /**
12  * 主程序类
13  * @Author: gongsl
14  * @Date: 2021-01-08 18:18
15  */
16 @SpringBootApplication
17 public class MainApplication {
18     public static void main(String[] args) {
19         //返回的是IOC容器
20         ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class,
args);
21
22         //根据类型获取容器中的组件id
23         String[] userArr = run.getBeanNamesForType(User.class);
24         System.out.println(Arrays.toString(userArr)); //运行结果: [com.gongsl.bean.User]
25
26         String[] petArr = run.getBeanNamesForType(Pet.class);
27         System.out.println(Arrays.toString(petArr)); //运行结果: [com.gongsl.bean.Pet]
28
29         String[] servletArr = run.getBeanNamesForType(ModelAndView.class);
30         //运行结果: [org.springframework.web.servlet.ModelAndView]
31         System.out.println(Arrays.toString(servletArr));
32     }
33 }

```

通过启动类中的运行结果可以发现，这三个对象都已经注入到容器中了，而且组件id就是全类名。

3.3 @Conditional的子注解

- **@Conditional** 注解包含了很多的子注解，比如：

```

1 @ConditionalOnJava
2 @ConditionalOnProperty
3 @ConditionalOnResource
4 @ConditionalOnExpression
5 @ConditionalOnSingleCandidate
6 @ConditionalOnBean
7 @ConditionalOnMissingBean
8 @ConditionalOnClass
9 @ConditionalOnMissingClass
10 @ConditionalOnWebApplication
11 @ConditionalOnNotWebApplication

```

- **@Conditional** 及其子注解都是条件注解，即当我们满足这个注解指定的条件的时候，我们才给容器中注入相关的组件，或者干相应的事；
- **@Conditional** 注解及其子注解一般用在配置类中，即包含 **@Configuration** 注解的类。

以 **@ConditionalOnBean** 注解为例验证条件注解的用法：

配置类：

```

1 package com.gongsl.config;
2
3 import com.gongsl.bean.Pet;
4 import com.gongsl.bean.User;
5 import org.springframework.boot.autoconfigure.condition.ConditionalOnBean;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.context.annotation.Configuration;
8
9 /**
10  * @Author: gongsl
11  * @Date: 2021-01-11 21:35
12  */
13 @Configuration
14 public class MyConfig {
15
16     @Bean
17     public User user1(){
18         return new User("张三",18,"男");
19     }
20
21     @ConditionalOnBean(name = "user12")
22     @Bean
23     public User user2(){
24         return new User("李四",20,"女");
25     }
26
27     @Bean
28     public Pet petCat(){
29         return new Pet("汤姆猫");
30     }
31 }

```

启动类:

```

1 package com.gongsl;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.context.ConfigurableApplicationContext;
6 import java.util.ArrayList;
7 import java.util.List;
8
9 /**
10  * 主程序类
11  * @Author: gongsl
12  * @Date: 2021-01-08 18:18
13  */
14 @SpringBootApplication
15 public class MainApplication {
16     public static void main(String[] args) {
17         // 返回的是IOC容器
18         ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class,
19 args);
20
21         List<String> list = new ArrayList<String>();
22         // 获取容器中的所有组件id
23         String[] names = run.getBeanDefinitionNames();
24         for (String name : names) {
25             if("user1".equals(name) || "user2".equals(name) || "petCat".equals(name)){
26                 list.add(name);
27             }
28         }
29     }
30 }

```

```

26         }
27     }
28     System.out.println(list); //运行结果: [user1, petCat]
29 }
30 }

```

1. 配置类中 `user2` 方法上的 `@ConditionalOnBean(name = "user12")` 注解表示，只有当容器中有名叫“user12”的组件id时，`user2` 方法上的 `@Bean` 注解才会生效，即才会把user2注入到容器中；
2. 如果把配置类中的 `@ConditionalOnBean(name = "user12")` 改成 `@ConditionalOnBean(name = "user1")` 的话，启动类中的运行结果就是: [user1, user2, petCat]；
3. 类似 `@ConditionalOnBean` 这种条件注解也可以放到类上面(一般都是指配置类)，表示只有当条件注解中的条件满足时，类中的方法上把组件注入到容器中等操作才会生效；
4. 配置类上的 `@Configuration` 注解是会把该类作为一个组件注入到容器中的，但是如果类似 `@ConditionalOnBean` 这种条件注解放到配置类上面且条件注解中的条件不满足的话，配置类上的 `@Configuration` 注解就不会生效，配置类自然也不会再被注入到容器中了。

3.4 @ImportResource注解

在之前使用Spring开发项目的时候，我们想要把bean注入到容器中，一般都是使用的配置文件。如果现在使用 `@Bean` 注解了，但是之前使用的配置文件还想保留，也不想把配置文件中的bean标签一个个都转成 `@Bean` 注解的话，就可以使用这个 `@ImportResource` 注解来解决这个问题。

`@ImportResource`注解的用法案例如下：

在 `src/main/resources` 目录下新增了一个bean.xml文件，内容如下：

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://www.springframework.org/schema/beans
5                               http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <bean id="haha" class="com.gongsl.bean.User">
8         <property name="name" value="李四"></property>
9         <property name="age" value="20"></property>
10        <property name="gender" value="女"></property>
11    </bean>
12 </beans>

```

配置类：

```

1  package com.gongsl.config;
2
3  import com.gongsl.bean.Pet;
4  import com.gongsl.bean.User;
5  import org.springframework.context.annotation.Bean;
6  import org.springframework.context.annotation.Configuration;
7  import org.springframework.context.annotation.ImportResource;
8
9  /**
10   * @Author: gongsl
11   * @Date: 2021-01-11 21:35
12   */
13  @Configuration
14  @ImportResource("classpath:bean.xml")
15  public class MyConfig {
16
17      @Bean
18      public User user(){
19          return new User("张三", 18, "男");
20      }
21  }

```



```

20     }
21
22     @Bean
23     public Pet petCat(){
24         return new Pet("汤姆猫");
25     }
26 }

```

启动类：

```

1  package com.gongsl;
2
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.SpringBootApplication;
5  import org.springframework.context.ConfigurableApplicationContext;
6  import java.util.ArrayList;
7  import java.util.List;
8
9  /**
10   * 主程序类
11   * @Author: gongsl
12   * @Date: 2021-01-08 18:18
13   */
14  @SpringBootApplication
15  public class MainApplication {
16      public static void main(String[] args) {
17          //返回的是IOC容器
18          ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class,
args);
19
20          List<String> list = new ArrayList<String>();
21          //获取容器中的所有组件id
22          String[] names = run.getBeanDefinitionNames();
23          for (String name : names) {
24              if("haha".equals(name) || "user".equals(name) || "petCat".equals(name)){
25                  list.add(name);
26              }
27          }
28          System.out.println(list); //运行结果: [user, petCat, haha]
29      }
30  }

```

启动类的运行结果中包含了“haha”这个组件id，说明配置类中的 `@ImportResource("classpath:bean.xml")` 注解生效了，已经把配置文件中bean标签的内容注入到了容器中。

3.5 @ConfigurationProperties注解

- 该注解一般用于做配置绑定，这个注解可以将 `properties` 文件或者 `yaml` 文件中指定的数据绑定到javaBean中；
- `@ConfigurationProperties` 可以搭配 `@Component`、`@EnableConfigurationProperties` 或 `@Bean` 来使用；
- `@ConfigurationProperties` 如果不搭配 `@Bean` 的话，一般是放在实体类上的，然后可以使用该注解中的 `prefix` 属性来和配置文件中相同前缀的数据进行绑定。但是最好不要使用 `@ConfigurationProperties(prefix = "user")` 这种写法，因为前缀如果是“user”而实体类中正好也有一个name属性的话，那么在进行数据绑定的时候，不管配置文件中针对该属性配置的值是什么，这个name属性获取到的都是我们的电脑用户名；
- 由于只有容器中的组件才能使用类似配置绑定等功能，所以需要先把javaBean注入到容器中，比如搭配 `@Component` 注解进行注入。如果是搭配 `@EnableConfigurationProperties` 注解使用的话，该注解一般用在配置类上。假设我们对User类进行配置绑定，那么在配置类中使用 `@EnableConfigurationProperties(User.class)` 即可；

- `@ConfigurationProperties` 如果搭配 `@Bean` 使用的话，这两个注解直接标在配置类的方法上就可以了。假设我们对jar包中的实体类进行配置绑定的话，由于没法修改jar包中的类，所以是没法在类上标注解的，这时就可以采用搭配 `@Bean` 注解的这种方式进行配置绑定；
- 如果需要进行配置绑定的实体类是jar包中的，这时候一般会结合 `@EnableConfigurationProperties` 注解来进行配置绑定，毕竟jar包中的类我们无法修改，所以没法在类上加 `@Component` 注解，所以结合 `@Component` 注解实现配置绑定的方式就行不通了；
- 配置绑定时，会自动检测 `src/main/resources` 目录下的 `application.properties` 文件或 `application.yaml` 文件，所以需要绑定到javaBean中的内容要写在这俩配置文件的某一个里面才行。当然，写在 `application.yaml` 文件中也是可以的，因为它和 `application.yaml` 文件是一样的；
- 如果在 `src/main/resources` 目录下同时存在 `application.properties` 文件和 `application.yaml` 文件，那么在进行配置绑定的时候，`application.properties` 文件的优先级是要高于 `application.yaml` 文件的，言外之意就是，两个配置文件都针对某个实体类的某个属性配置值了的话，以 `application.properties` 文件中的配置为准。

`@ConfigurationProperties` 结合 `@Component` 演示配置绑定：

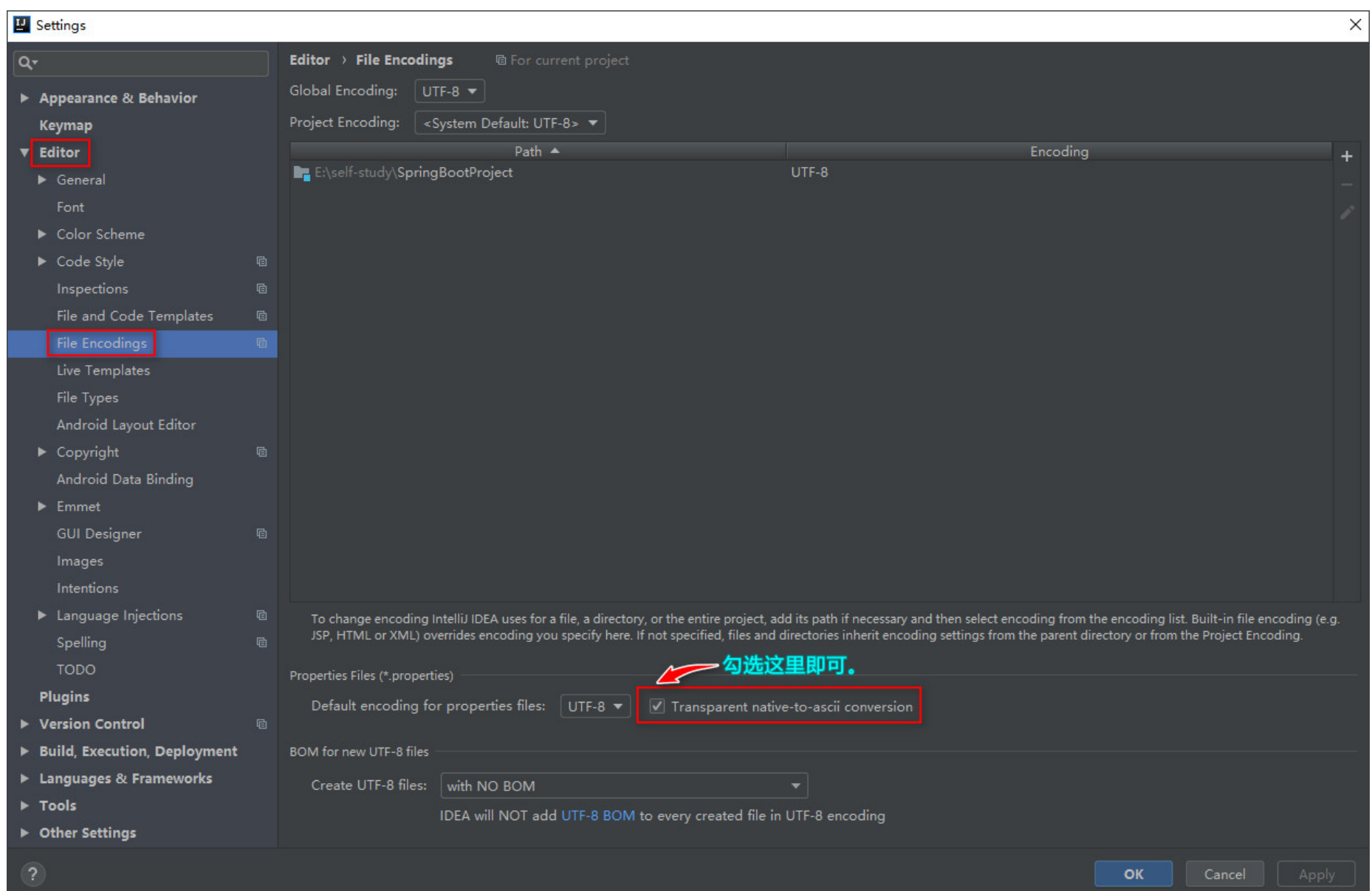
User实体类：

```
1 package com.gongsl.bean;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6 import lombok.ToString;
7 import org.springframework.boot.context.properties.ConfigurationProperties;
8 import org.springframework.stereotype.Component;
9
10 /**
11  * @Author: gongsl
12  * @Date: 2021-01-11 21:27
13  */
14 @Data
15 @ToString
16 @NoArgsConstructor
17 @AllArgsConstructor
18 @Component
19 @ConfigurationProperties(prefix = "my.user")
20 public class User {
21     private String name;
22     private Integer age;
23     private String gender;
24 }
```

application.properties文件：

```
1 my.user.name=Tom
2 my.user.age=18
```

如果application.properties中有中文，在配置绑定时可能会乱码，只要勾选IDEA中的如下选项即可解决乱码问题：



如果是放到服务器上，没有办法像本地开发时这种改IDEA的设置的话，也可以把application.properties中的中文直接使用ASCII编码方式显示。比如 创建成功 就写成 `\u521B\u5EFA\u6210\u529F`。

测试类：

```
1 package com.gongsl.controller;
2
3 import com.gongsl.bean.User;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 /**
9  * @Author: gongsl
10  * @Date: 2021-01-08 18:19
11  */
12 @RestController
13 public class HelloController {
14
15     @Autowired
16     User user;
17
18     @RequestMapping("/user")
19     public User home(){
20         return user;
21     }
22 }
```

执行启动类中的main方法启动项目后浏览器访问：<http://localhost:8080/user> 会返回如下内容：

```
1 {"name":"Tom","age":18,"gender":null}
```

由返回的内容可知，配置绑定成功了。

`@ConfigurationProperties` 结合 `@EnableConfigurationProperties` 演示配置绑定：

User实体类:

```
1 package com.gongsl.bean;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6 import lombok.ToString;
7 import org.springframework.boot.context.properties.ConfigurationProperties;
8
9 /**
10  * @Author: gongsl
11  * @Date: 2021-01-11 21:27
12  */
13 @Data
14 @ToString
15 @NoArgsConstructor
16 @AllArgsConstructor
17 @ConfigurationProperties(prefix = "my.user")
18 public class User {
19     private String name;
20     private Integer age;
21     private String gender;
22 }
```

配置类:

```
1 package com.gongsl.config;
2
3 import com.gongsl.bean.User;
4 import org.springframework.boot.context.properties.EnableConfigurationProperties;
5 import org.springframework.context.annotation.Configuration;
6
7 /**
8  * @Author: gongsl
9  * @Date: 2021-01-11 21:35
10  */
11 @Configuration
12 @EnableConfigurationProperties(User.class)
13 public class MyConfig {
14
15 }
```

application.properties文件:

```
1 my.user.name=Tom
2 my.user.age=18
```

测试类:

```
1 package com.gongsl.controller;
2
3 import com.gongsl.bean.User;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 /**
9  * @Author: gongsl
```

```

10  * @Date: 2021-01-08 18:19
11  */
12  @RestController
13  public class HelloController {
14
15      @Autowired
16      User user;
17
18      @RequestMapping("/user")
19      public User home(){
20          return user;
21      }
22  }

```

启动项目进行测试时，测试结果和上面结合 `@Component` 注解演示的测试结果是一样的。与上面结合 `@Component` 注解相比，结合 `@EnableConfigurationProperties` 注解的区别就是去掉了实体类上的 `@Component` 注解，然后在配置类上增加了 `@EnableConfigurationProperties(User.class)` 注解。

`@ConfigurationProperties` 结合 `@Bean` 演示配置绑定：

User实体类：

```

1  package com.gongsl.bean;
2
3  import lombok.AllArgsConstructor;
4  import lombok.Data;
5  import lombok.NoArgsConstructor;
6  import lombok.ToString;
7  import org.springframework.boot.context.properties.ConfigurationProperties;
8
9  /**
10   * @Author: gongsl
11   * @Date: 2021-01-11 21:27
12   */
13  @Data
14  @ToString
15  @NoArgsConstructor
16  @AllArgsConstructor
17  public class User {
18      private String name;
19      private Integer age;
20      private String gender;
21  }

```

配置类：

```

1  package com.gongsl.config;
2
3  import com.gongsl.bean.User;
4  import org.springframework.boot.context.properties.ConfigurationProperties;
5  import org.springframework.context.annotation.Bean;
6  import org.springframework.context.annotation.Configuration;
7
8  /**
9   * @Author: gongsl
10   * @Date: 2021-01-11 21:35
11   */
12  @Configuration
13  public class MyConfig {
14

```



```

15     @Bean
16     @ConfigurationProperties(prefix = "my.user")
17     public User user(){
18         return new User();
19     }
20 }

```

application.properties文件:

```

1 my.user.name=Tom
2 my.user.age=18

```

测试类:

```

1 package com.gongsl.controller;
2
3 import com.gongsl.bean.User;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 /**
9  * @Author: gongsl
10  * @Date: 2021-01-08 18:19
11  */
12 @RestController
13 public class HelloController {
14
15     @Autowired
16     User user;
17
18     @RequestMapping("/user")
19     public User home(){
20         return user;
21     }
22 }

```

结合 `@Bean` 演示的测试结果和上面两种方式是一样的，这种配置绑定的方式，注解都标到了配置类的方法上，没有标到实体类上，所以即便实体类是jar包中的类，也可以实现配置绑定。

演示 `application.properties` 文件和 `application.yaml` 文件同时存在的场景:

application.properties文件:

```

1 my.user.name=Tom
2 my.user.age=18

```

application.yaml文件:

```

1 my.user:
2   name: Jerry
3   age: 20
4   gender: 男

```

执行启动类中的main方法启动项目后浏览器访问: <http://localhost:8080/user> 后返回如下内容:

```

1 {"name":"Tom","age":18,"gender":"男"}

```

通过返回结果发现，虽然name属性和age属性两个配置文件中都有配置值，但是最终使用的是 `application.properties` 文件中的，`application.properties` 文件中没有配置gender属性的值，所以最终使用的是 `application.yaml` 文件中的值。

3.6 @Value注解

- 该注解中可以使用 `${}` 这种方式默认从`application.properties`文件或者`application.yaml`文件中获取值来给实体类的属性赋值，比如 `@Value("${my.user.name}")` 就是获取文件中`my.user.name`对应的值。我们还可以在获取不到值的时候设置一个默认值，比如 `@Value("${my.user.name:Tom}")` 这种，就是在获取不到值的时候给实体类中对应属性设置默认值Tom；
- 该注解还可以使用SpEL表达式，SpEL表达式是可以进行运算的，比如在实体类的属性上添加 `@Value("#{10+2}")` 注解，就表示把 `12` 赋值给该属性。我们也可以在SpEL表达式中调用类的方法，下面案例中会有介绍；
- 我们也可以使用@Value注解直接给属性设置一个具体值，比如使用 `@Value("test")` 就是把 `test` 赋值给对应属性。

`@Value` 注解的用法案例如下：

User类：

```
1 package com.gongsl.bean;
2
3 import lombok.Data;
4 import org.springframework.beans.factory.annotation.Value;
5 import org.springframework.stereotype.Component;
6 import java.util.List;
7
8 @Data
9 @Component
10 public class User {
11     @Value("${my.user.name}")
12     private String name;
13
14     @Value("#{10+2}")
15     private Integer age;
16
17     @Value("男")
18     private String gender;
19
20     @Value("#{${my.user.friends}.split(',')}")
21     private List<String> friends;
22
23     @Value("${my.user.other:默认值}")
24     private String other;
25 }
```

测试类：

```
1 package com.gongsl.controller;
2
3 import com.gongsl.bean.User;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class HelloController {
10
11     @Autowired
12     User user;
13
14     @RequestMapping("/user")
```

```
15     public User home(){
16         return user;
17     }
18 }
```

application.properties文件:

```
1 my.user.name=Tom
2 my.user.friends=Jerry, Lucy, Mark
```

执行启动类中的main方法启动项目后浏览器访问: <http://localhost:8080/user> 后返回如下内容:

```
1 {"name":"Tom","age":12,"gender":"男","friends":["Jerry","Lucy","Mark"],"other":"默认值"}
```

3.7 @PropertySource注解

我们在进行配置绑定的时候, 无论是使用 `@Value` 注解还是使用 `@ConfigurationProperties` 注解, 默认情况下都只会从 application.properties 或者 application.yaml 这种 Spring 的主文件中获取数据。如果需要绑定的配置很多, 就会导致主配置文件内容过多, 不便于我们快速查找数据, 这时候就可以使用 `@PropertySource` 注解。我们可以针对某个实体类专门新增一个配置文件, 然后使用该注解进行导入, 再结合 `@Value` 注解或者 `@ConfigurationProperties` 注解进行配置绑定即可。

`@PropertySource` 结合 `@Value` 演示配置绑定:

User类:

```
1 package com.gongsl.bean;
2
3 import lombok.Data;
4 import org.springframework.beans.factory.annotation.Value;
5 import org.springframework.context.annotation.PropertySource;
6 import org.springframework.stereotype.Component;
7
8 @Data
9 @Component
10 @PropertySource(value = {"classpath:test.properties"})
11 public class User {
12     @Value("${my.user.name}")
13     private String name;
14
15     @Value("${my.user.age}")
16     private Integer age;
17 }
```

测试类:

```
1 package com.gongsl.controller;
2
3 import com.gongsl.bean.User;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class HelloController {
10
11     @Autowired
12     User user;
13
14     @RequestMapping("/user")
```

```

15     public User home(){
16         return user;
17     }
18 }

```

在 `src/main/resources` 目录下新增 `test.properties` 文件:

```

1 my.user.name=Jerry
2 my.user.age=18

```

执行启动类中的 `main` 方法启动项目后浏览器访问: <http://localhost:8080/user> 后返回如下内容:

```

1 {"name":"Jerry","age":18}

```

`@PropertySource` 结合 `@ConfigurationProperties` 演示配置绑定:

和结合 `@Value` 注解进行配置绑定相比, 其他都不变, 只需要把 `User` 类改成如下内容即可, 测试结果也是一样的。

```

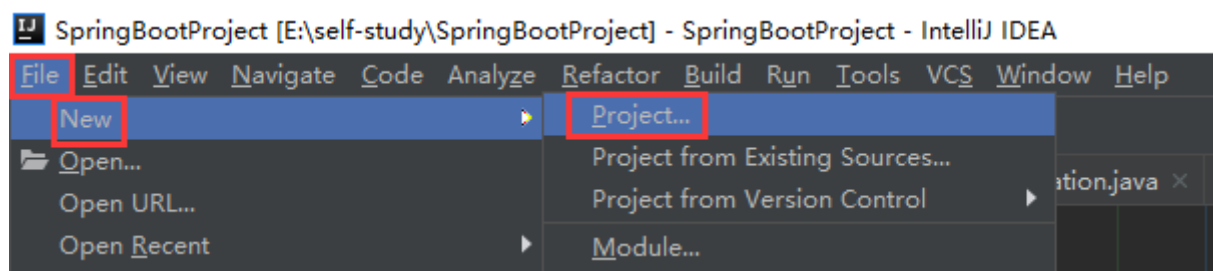
1 package com.gongsl.bean;
2
3 import lombok.Data;
4 import org.springframework.boot.context.properties.ConfigurationProperties;
5 import org.springframework.context.annotation.PropertySource;
6 import org.springframework.stereotype.Component;
7
8 @Data
9 @Component
10 @PropertySource(value = {"classpath:test.properties"})
11 @ConfigurationProperties(prefix = "my.user")
12 public class User {
13     private String name;
14     private Integer age;
15 }

```

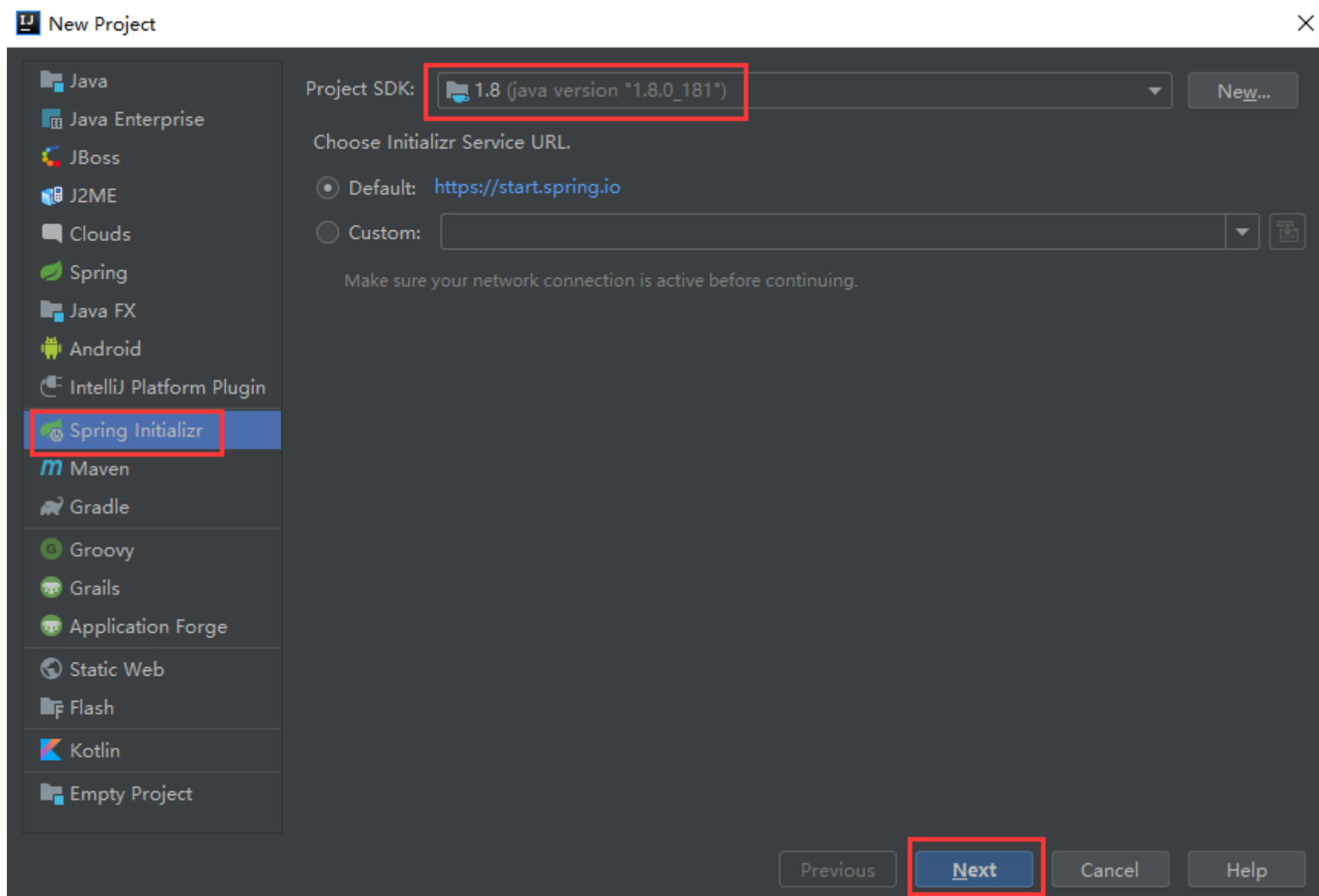
4.Spring Initializr

这是SpringBoot的一个项目初始化向导, 在IDEA中我们可以通过这个向导快速创建一个SpringBoot项目, 我们可以根据我们的开发场景选择不同的依赖, 步骤如下:

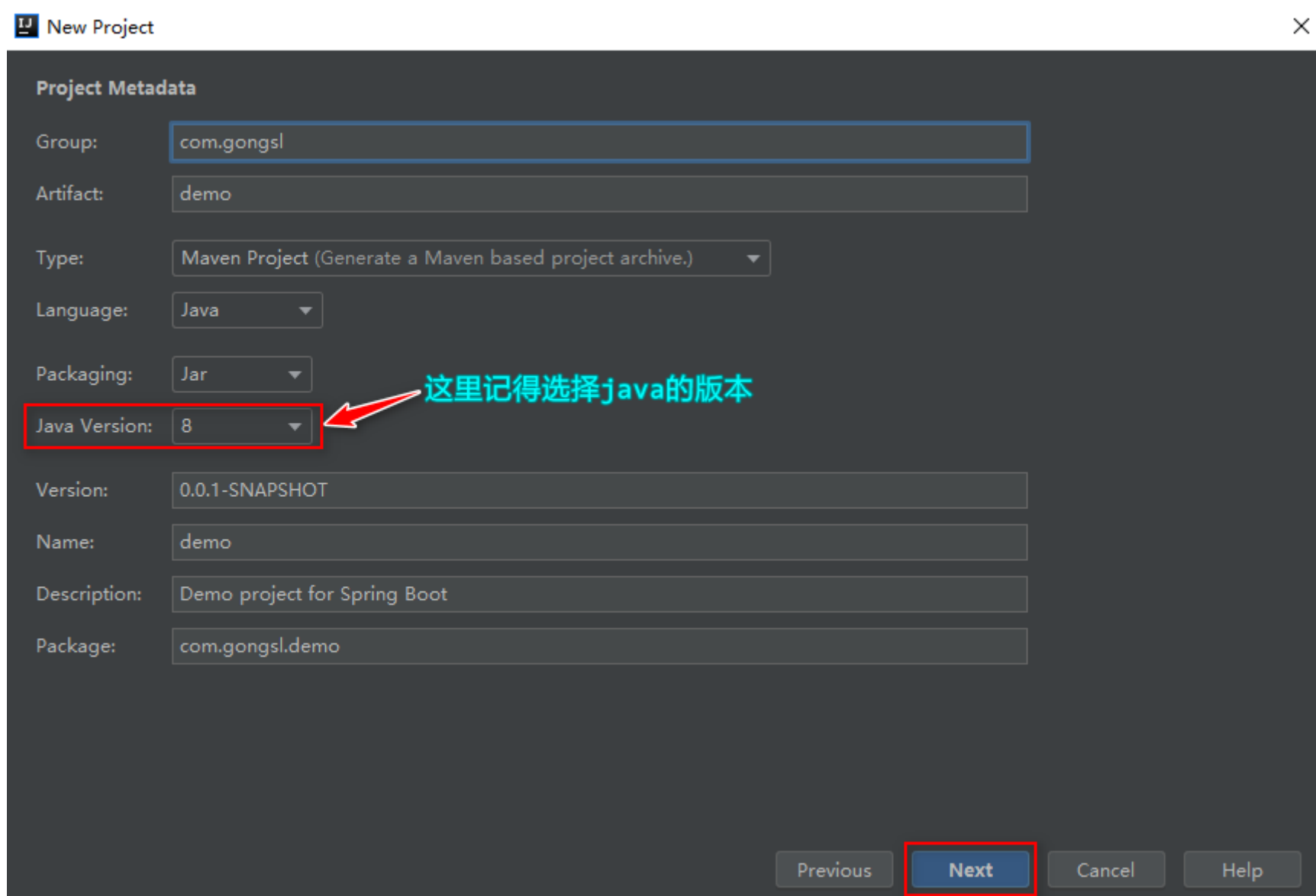
1. 依次选择File --->New --->Project;



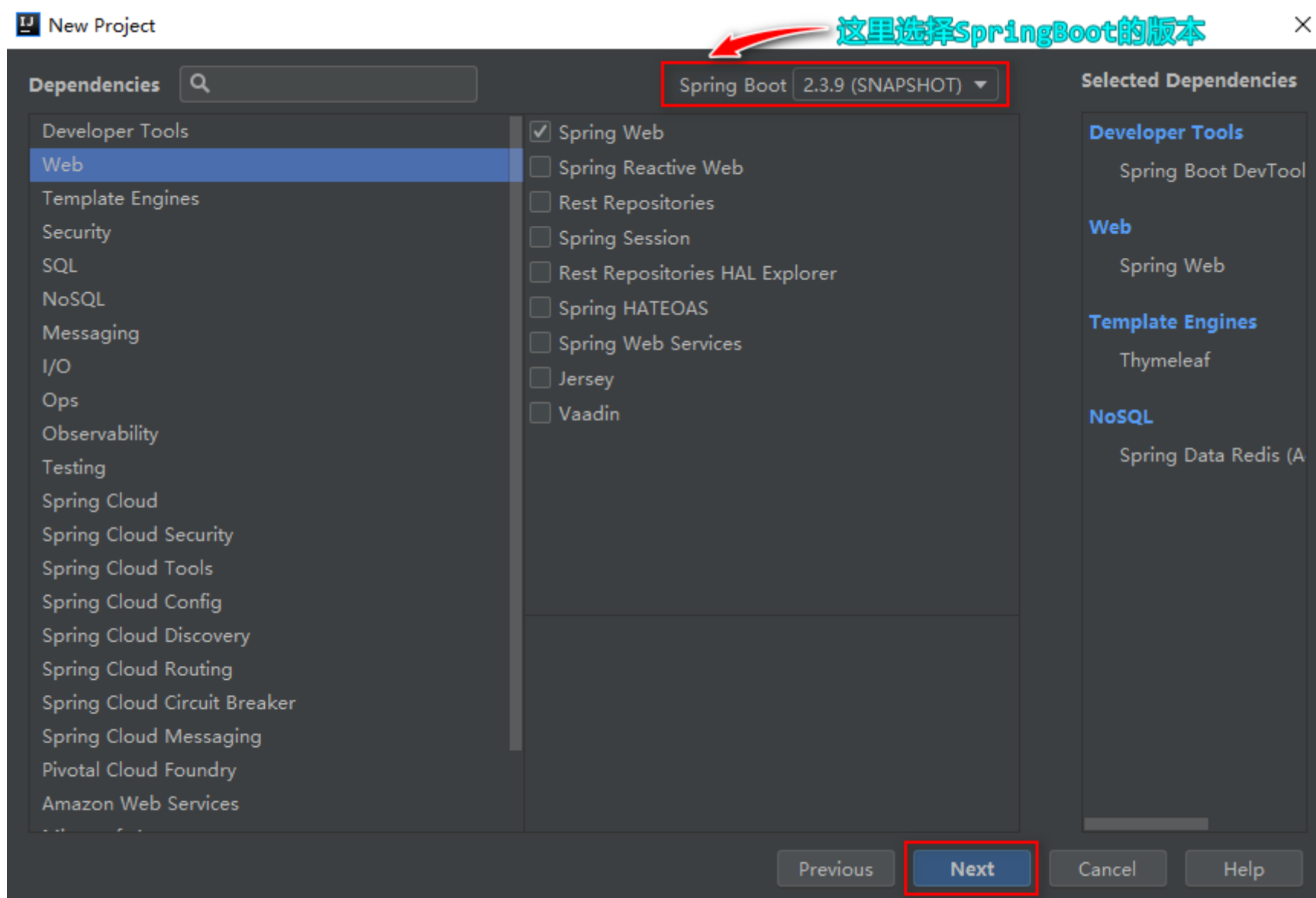
2. 选择Spring Initializr, 然后选择jdk版本;



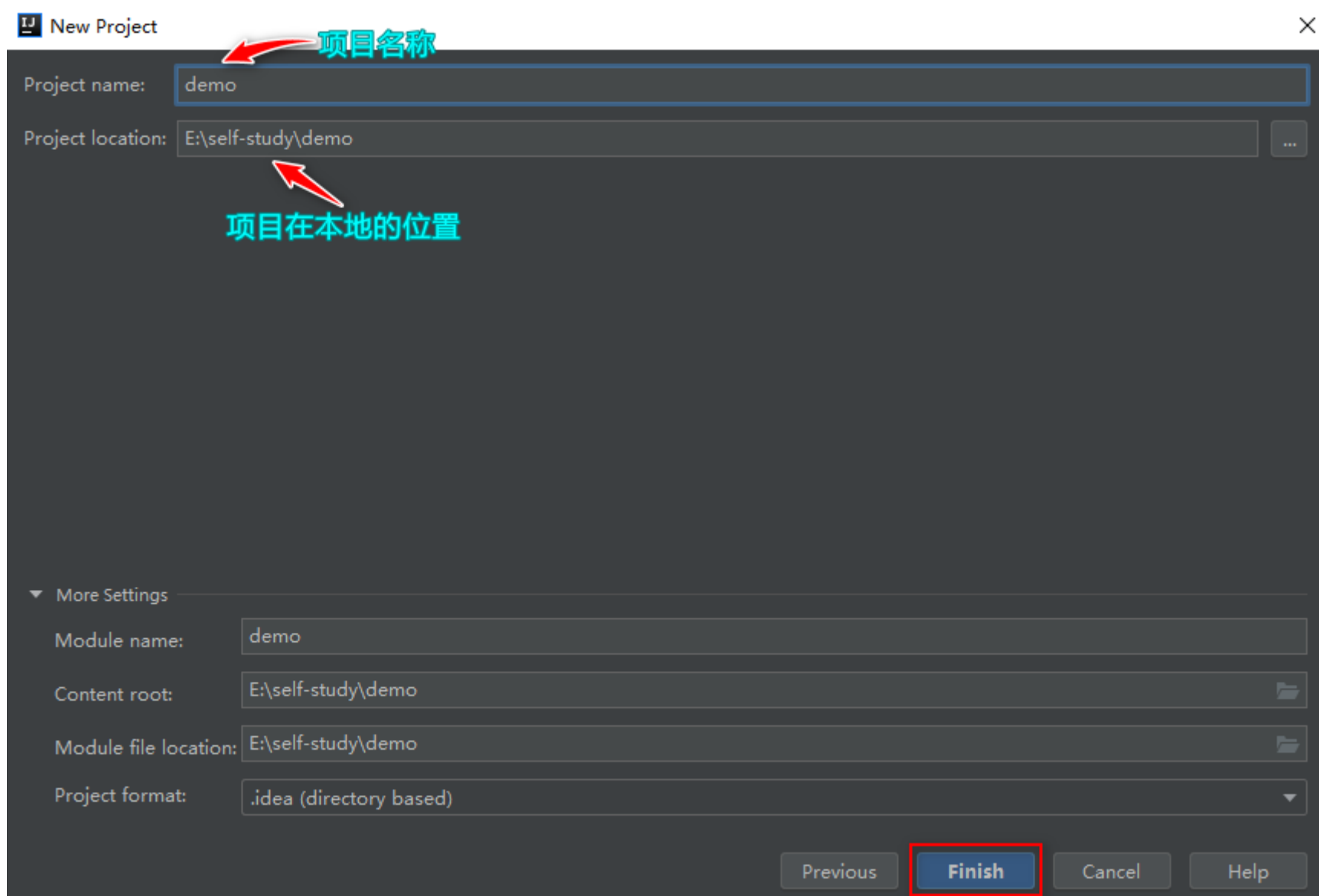
3. 填写基本信息;



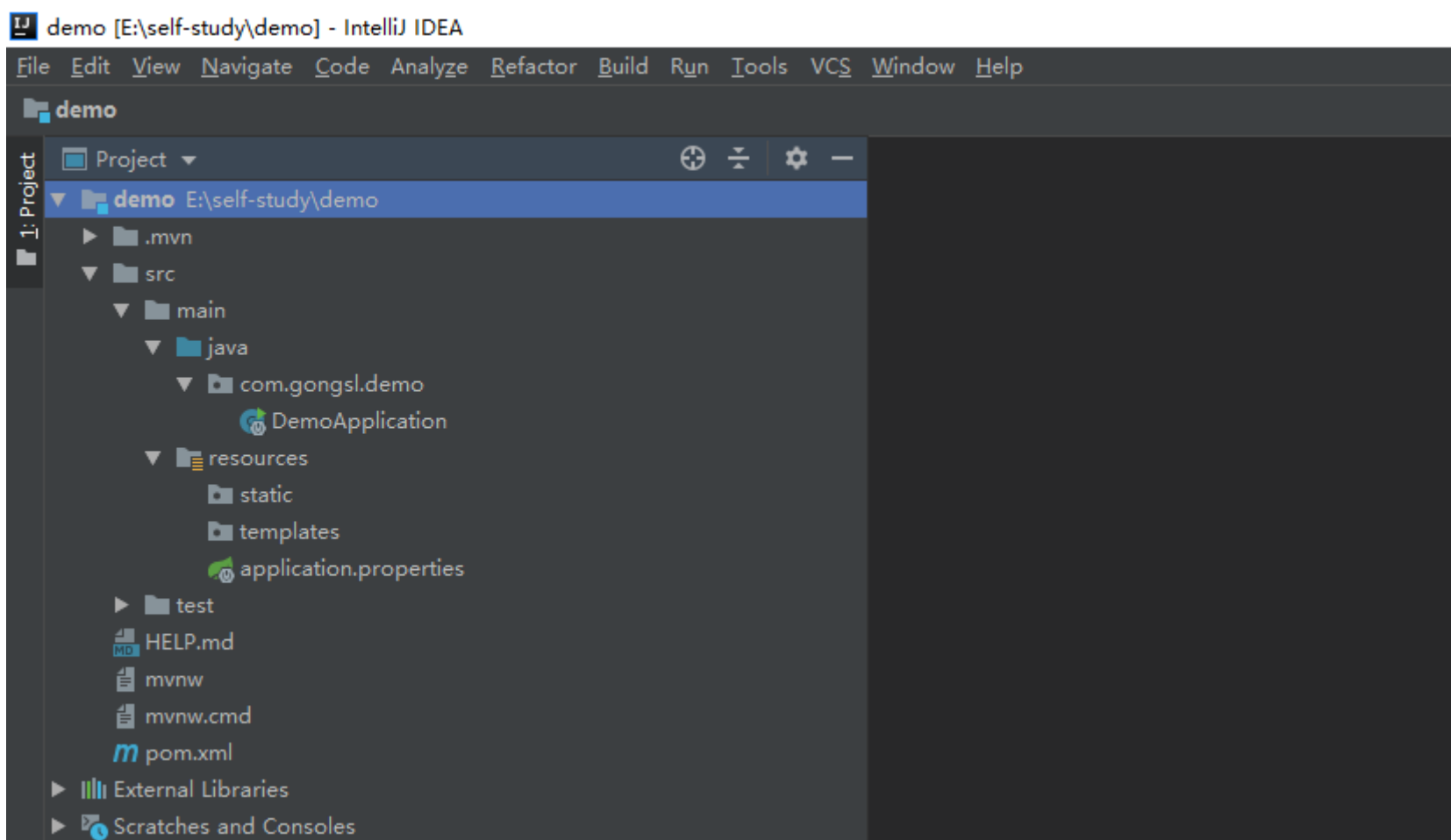
4. 选择依赖。我们需要什么场景下的依赖，就在这里进行选择就可以了;



5. 填写项目名称、项目位置等信息；



6. 最终生成的项目结构如下所示：



5. yaml配置文件的用法

5.1 基本语法

- 使用 **key: value** 这种表示形式，需要注意的是，冒号后面要有空格；
- 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，但是在IDEA中还是可以使用tab的，IDEA可以为我们自动转换成空格；
- 缩进的空格数不重要，只要相同层级的元素左对齐即可；
- 使用 **#** 来表示注释；
- 表示字符串时，是无需加引号的，如果要加，**'** 与 **"** 表示字符串内容会被转义/不转义。比如 **'test\t测试'** 最终结果是 **test\t测试**，而 **"test\t测试"** 最终结果是 **test 测试**。

5.2 不同数据类型的表示形式

- 字面量：单个的、不可再分的值。date、boolean、string、number、null

```
1 k: v
```

- 对象：键值对的集合。map、hash、set、object

```
1 #行内写法:
2 k: {k1: v1,k2: v2,k3: v3}
3 #或者:
4 k:
5   k1: v1
6   k2: v2
7   k3: v3
```

- 数组：一组按次序排列的值。array、list、queue

```
1 #行内写法:
2 k: [v1,v2,v3]
3 #或者:
4 k:
5   - v1
6   - v2
7   - v3
```

5.3 案例演示

User类:

```
1 package com.gongsl.bean;
2
3 import com.fasterxml.jackson.annotation.JsonFormat;
4 import lombok.Data;
5 import org.springframework.boot.context.properties.ConfigurationProperties;
6 import org.springframework.format.annotation.DateTimeFormat;
7 import org.springframework.stereotype.Component;
8 import java.util.Date;
9 import java.util.List;
10 import java.util.Map;
11
12 @Data
13 @Component
14 @ConfigurationProperties(prefix = "my.user")
15 public class User {
16
17     private String name;
18
19     private Integer age;
20
21     private String gender;
22
23     @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
24     @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss", timezone = "GMT+8")
25     private Date birth;
26
27     private Pet pet;
28
29     private List list;
30
31     private String[] strings;
32
33     private Map map;
34
35     private Map<String, Map<String, String>> score;
36 }
```

Pet类:

```
1 package com.gongsl.bean;
2
3 import lombok.Data;
4
5 @Data
6 public class Pet {
7     private String name;
8     private Double weight;
9 }
```

在 `src/main/resources` 目录下新增application.yaml文件:

```
1 my.user:
2   name: Jerry
3   age: 20
4   gender: 男
5   birth: 2020-12-28 15:36:25
```

```

6   pet:
7       name: cat
8       weight: 12.3
9   list: [abc,123,true,ABC]
10  strings: [张三,李四,Tom]
11  map: {a: 王五,b: test}
12  score:
13      english:
14          first: 30
15          second: 40
16          third: 50
17  math: [131,140,148]
18  chinese: {first: 128,second: 136}

```

测试类:

```

1  package com.gongsl.controller;
2
3  import com.gongsl.bean.User;
4  import org.springframework.beans.factory.annotation.Autowired;
5  import org.springframework.web.bind.annotation.RequestMapping;
6  import org.springframework.web.bind.annotation.RestController;
7
8  @RestController
9  public class HelloController {
10
11      @Autowired
12      User user;
13
14      @RequestMapping("/user")
15      public User home(){
16          return user;
17      }
18  }

```

执行启动类中的main方法启动项目后浏览器访问: <http://localhost:8080/user> 后返回如下内容:

```

1  {"name":"Jerry","age":20,"gender":"男","birth":"2020-12-28 15:36:25","pet":
   {"name":"cat","weight":12.3},"list":["abc",123,true,"ABC"],"strings":["张三","李
   四","Tom"],"map":{"a":"王五","b":"test"},"score":{"english":
   {"first":"30","second":"40","third":"50"},"math":{"0":"131","1":"140","2":"148"},"chinese":
   {"first":"128","second":"136"}}}

```

把返回的JSON串格式化后如下所示:

```

1  {
2      "name": "Jerry",
3      "age": 20,
4      "gender": "男",
5      "birth": "2020-12-28 15:36:25",
6      "pet": {
7          "name": "cat",
8          "weight": 12.3
9      },
10     "list": [
11         "abc",
12         123,
13         true,
14         "ABC"

```

```
15     ],
16     "strings": [
17         "张三",
18         "李四",
19         "Tom"
20     ],
21     "map": {
22         "a": "王五",
23         "b": "test"
24     },
25     "score": {
26         "english": {
27             "first": "30",
28             "second": "40",
29             "third": "50"
30         },
31         "math": {
32             "0": "131",
33             "1": "140",
34             "2": "148"
35         },
36         "chinese": {
37             "first": "128",
38             "second": "136"
39         }
40     }
41 }
```

5.4 配置文件的提示功能

自定义的类和配置文件绑定的时候，在application.yaml文件中写对应的配置是没有提示的，如果我们想要有提示，可以通过加入以下依赖来实现：

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

加入依赖以后，记得重新启动一下启动类，以便使依赖生效。

由于我们添加的这个依赖只是在代码开发时用于提示的，当项目打包部署后并没有什么实际性的用处，所以我们在项目打包时就不用把这个依赖对应的jar包打包进去了，可以使用 **exclude** 标签把这个依赖排除掉，如下所示：

```
1 <build>
2     <plugins>
3         <plugin>
4             <groupId>org.springframework.boot</groupId>
5             <artifactId>spring-boot-maven-plugin</artifactId>
6             <configuration>
7                 <excludes>
8                     <exclude>
9                         <groupId>org.springframework.boot</groupId>
10                        <artifactId>spring-boot-configuration-processor</artifactId>
11                    </exclude>
12                </excludes>
13            </configuration>
14        </plugin>
15    </plugins>
16 </build>
```

6.Web场景的开发

6.1 静态资源的访问

- 只要静态资源放在类路径下的 `/static`、`/public`、`/resources`、`/META-INF/resources` 任一文件夹中，就可以使用 **当前项目根路径/静态资源名** 的方式直接进行访问，比如：`http://localhost:8080/a.jpg`，如果访问不到，可以使用maven重新编译下再试试；
- 在maven项目中上面说的类路径就是指 `src/main/resources` 目录下，比如 `src/main/resources/static/a.jpg`；
- 如果我们想要改变这些静态资源的默认目录，可以使用 `spring.resources.static-locations` 配置进行修改，如果是在application.yaml文件中，可以写成下面这样：

```
1 spring:
2   resources:
3     static-locations: classpath:/test/
```

如果我们想要自定义多个默认的静态资源目录，可以使用如下写法：

```
1 spring:
2   resources:
3     static-locations: [classpath:/test1/,classpath:/test2/]
```

- 假设静态资源放在了默认目录下，但是我们想要给静态资源加一个访问前缀的话可以使用如下写法：

```
1 spring:
2   mvc:
3     static-path-pattern: /api/**
```

这时如果想要访问静态资源，就必须在url上加上 `api` 这个前缀了，比如`http://localhost:8080/api/a.jpg`。

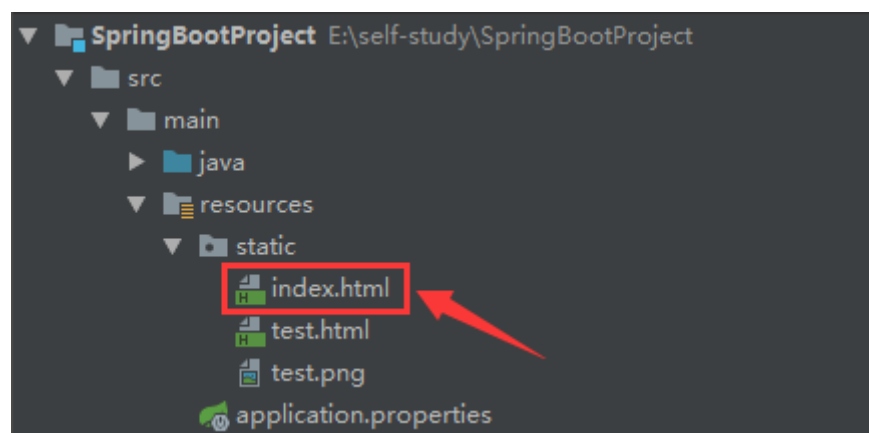
6.2 欢迎页的支持

我们如果将欢迎页 `index.html` 放到项目的静态资源目录下，比如 `static` 目录下，那么当我们在浏览器中进行访问时直接输入IP和端口就可以自动跳转到 `index.html` 页面，案例演示如下：

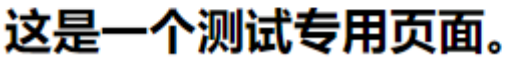
`index.html`中的内容：

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>测试使用</title>
6 </head>
7 <body>
8 <h2>这是一个测试专用页面。</h2>
9 </body>
10 </html>
```

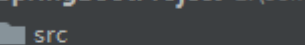
项目结构：



启动项目后测试结果展示：

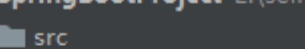


6.3 自定义favicon



SpringBootProject E:\self-study\SpringBootProject

- src
 - main
 - java
 - resources
 - static
 - favicon.ico
 - index.html



SpringBootProject E:\self-study\SpringBootProject

- src
 - main
 - java
 - resources
 - static
 - favicon.ico
 - index.html

测试使用

localhost:8080

看视频 破解 搜索 CSDN 壁纸网站

这是一个测试专用页面。

6.4 Web开发常用注解

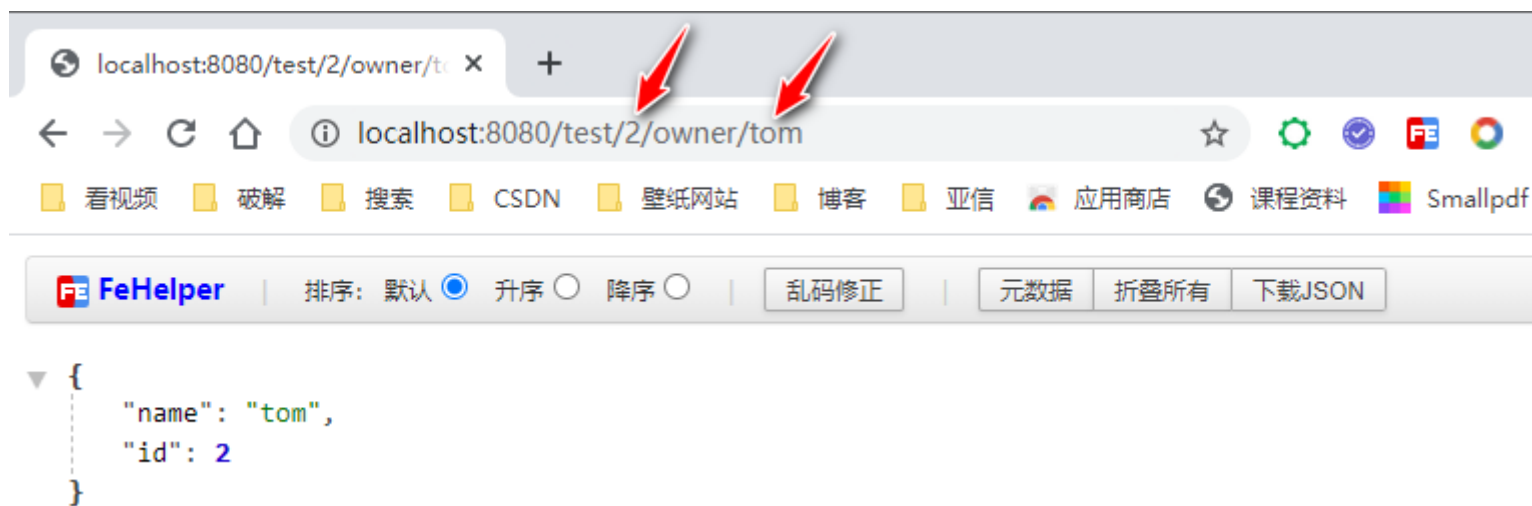
6.4.1 @PathVariable注解

[illegible]

```

15
16     Map<String, Object> map = new HashMap<String, Object>();
17     map.put("id",id);
18     map.put("name",name);
19     return map;
20 }
21

```



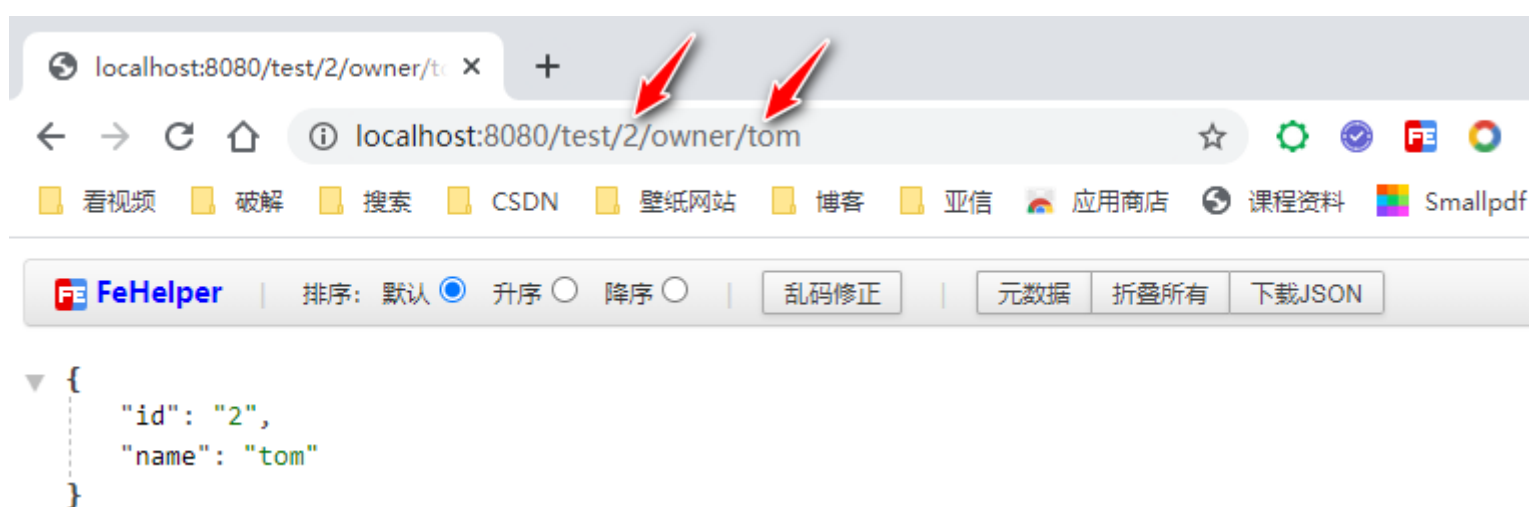
通过浏览器返回结果可知，已经通过 `@PathVariable` 注解获取到了请求路径中的变量啦。

除了以上方式外，还可以通过Map集合的方式获取请求路径中的所有变量，案例演示如下：

```

1 package com.gongsl.controller;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.PathVariable;
5 import org.springframework.web.bind.annotation.RestController;
6 import java.util.Map;
7
8 @RestController
9 public class HelloController {
10
11     @GetMapping("/test/{id}/owner/{name}")
12     public Map<String, Object> test(@PathVariable Map<String, Object> map){
13         return map;
14     }
15 }

```



6.4.2 @RequestHeader注解

可以使用该注解获取请求头中的信息，演示案例如下：

```

1 package com.gongsl.controller;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.PathVariable;
5 import org.springframework.web.bind.annotation.RequestHeader;
6 import org.springframework.web.bind.annotation.RestController;
7 import java.util.HashMap;

```

```

8  import java.util.Map;
9
10 @RestController
11 public class HelloController {
12
13     @GetMapping("/test/{id}")
14     public Map<String, Object> home(@PathVariable("id") Integer id,
15                                     @RequestHeader("connection") String connection,
16                                     @RequestHeader(name = "other", defaultValue = "未找到")
17                                     String other){
18         Map<String, Object> map = new HashMap<String, Object>();
19         map.put("id",id);
20         map.put("connection",connection);
21         map.put("other",other);
22         return map;
23     }
24 }

```



通过浏览器返回结果可知，类似 `connection` 这种请求头中存在的属性就可以获取到对应的值，不存在的返回了我们自己设置的默认值。

我们还可以通过Map集合的方式，获取请求头中所有属性的值，演示案例如下：

```

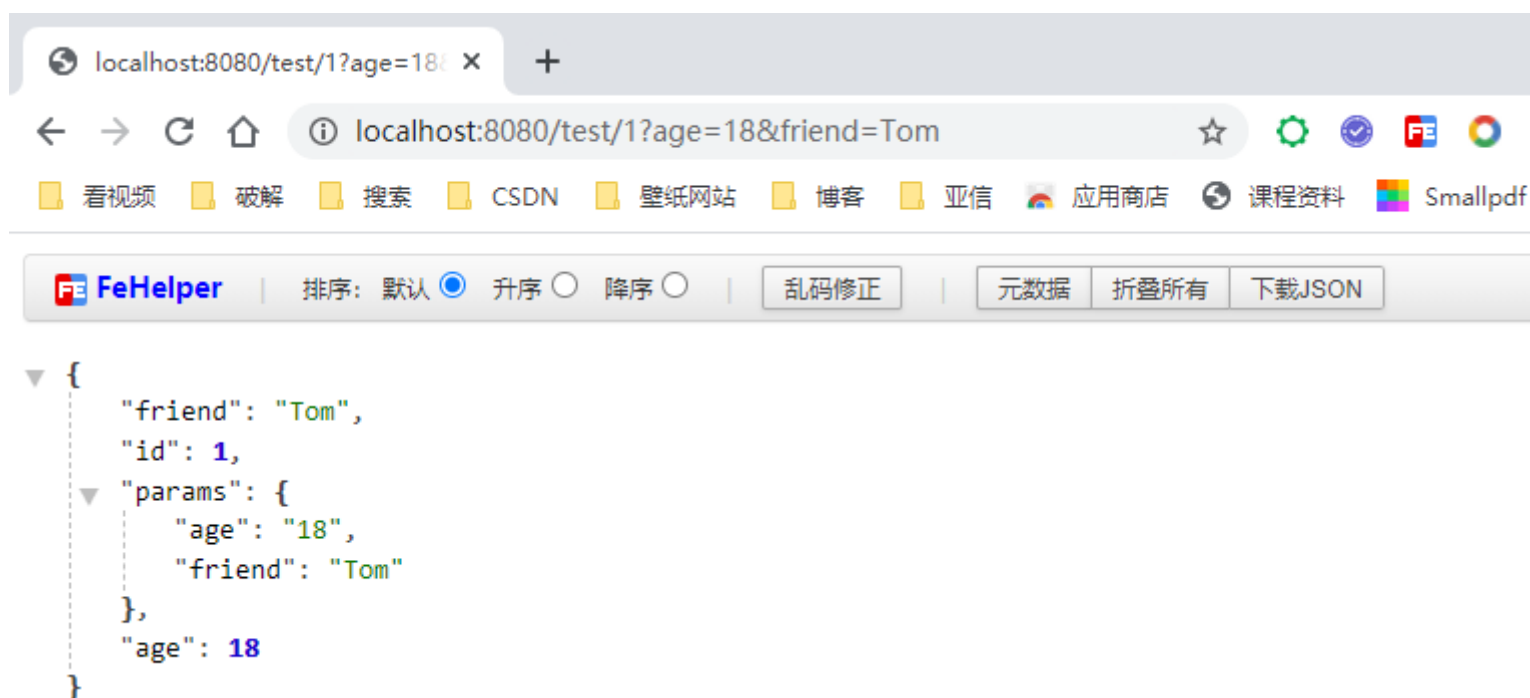
1  package com.gongsl.controller;
2
3  import org.springframework.web.bind.annotation.GetMapping;
4  import org.springframework.web.bind.annotation.PathVariable;
5  import org.springframework.web.bind.annotation.RequestHeader;
6  import org.springframework.web.bind.annotation.RestController;
7  import java.util.HashMap;
8  import java.util.Map;
9
10 @RestController
11 public class HelloController {
12
13     @GetMapping("/test/{id}")
14     public Map<String, Object> home(@PathVariable("id") Integer id,
15                                     @RequestHeader Map<String, Object> headers){
16         Map<String, Object> map = new HashMap<String, Object>();
17         map.put("id",id);
18         map.put("headers",headers);
19         return map;
20     }
21 }

```



6.4.3 @RequestParam注解

可以通过该注解获取请求参数，演示案例如下：

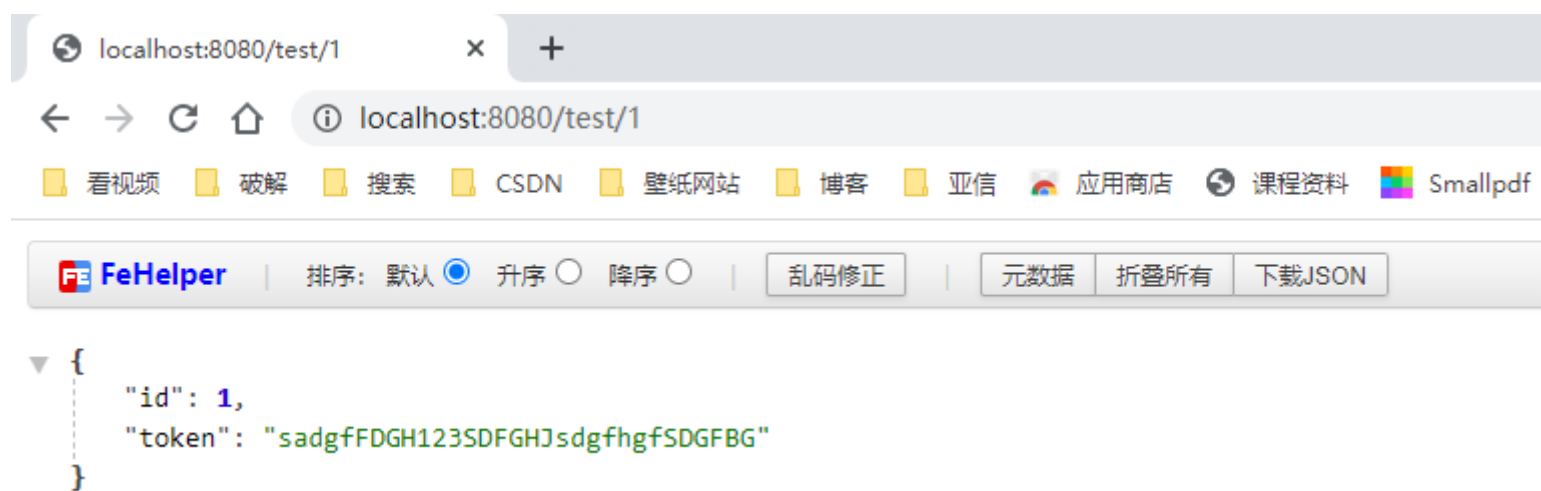


通过浏览器的返回结果可知，已经获取到了请求参数，而且该注解还支持通过Map集合的方式获取所有的请求参数。

6.4.4 @CookieValue注解

可以通过该注解获取cookie的值。由于我这边测试的时候浏览器中没有cookie，所有我自己手动加了一个，加上后演示案例如下：

```
1 package com.gongsl.controller;
2
3 import org.springframework.web.bind.annotation.*;
4 import java.util.HashMap;
5 import java.util.Map;
6
7 @RestController
8 public class HelloController {
9
10     @GetMapping("/test/{id}")
11     public Map<String, Object> home(@PathVariable("id") Integer id,
12                                   @CookieValue("token") String token){
13         Map<String, Object> map = new HashMap<String, Object>();
14         map.put("id", id);
15         map.put("token", token);
16         return map;
17     }
18 }
```

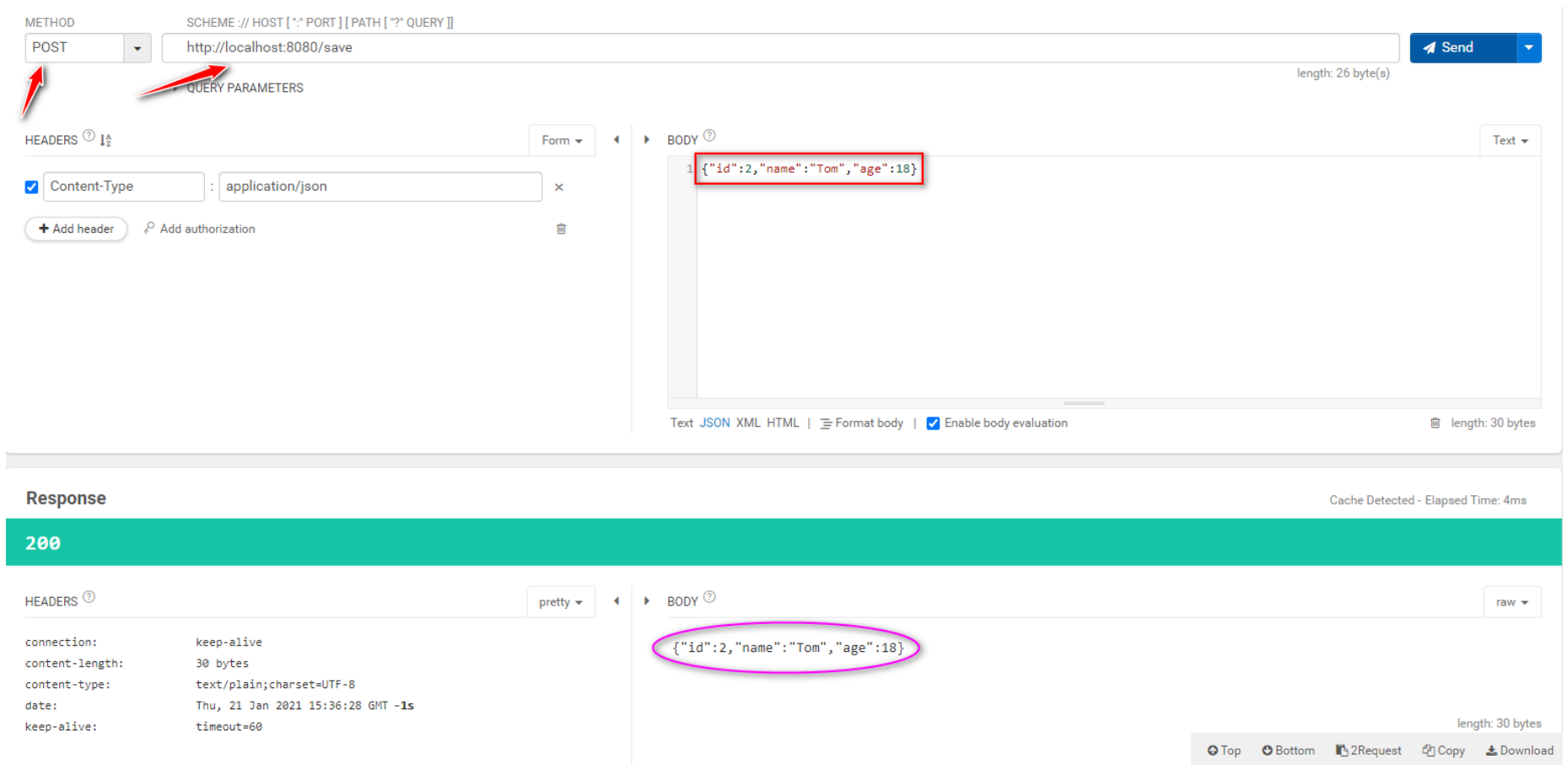


通过浏览器的返回结果可知，我们已经获取到浏览器中的cookie的值了。

6.4.5 @RequestBody注解

可以使用该注解获取请求体中的内容。这里使用工具发起一个post请求进行演示，演示案例如下：

```
1 package com.gongsl.controller;
2
3 import org.springframework.web.bind.annotation.*;
4
5 @RestController
6 public class HelloController {
7
8     @PostMapping("/save")
9     public String home(@RequestBody String content){
10         return content;
11     }
12 }
```



通过返回结果可以发现，已经获取到请求体中的内容了。

6.5 统一异常处理

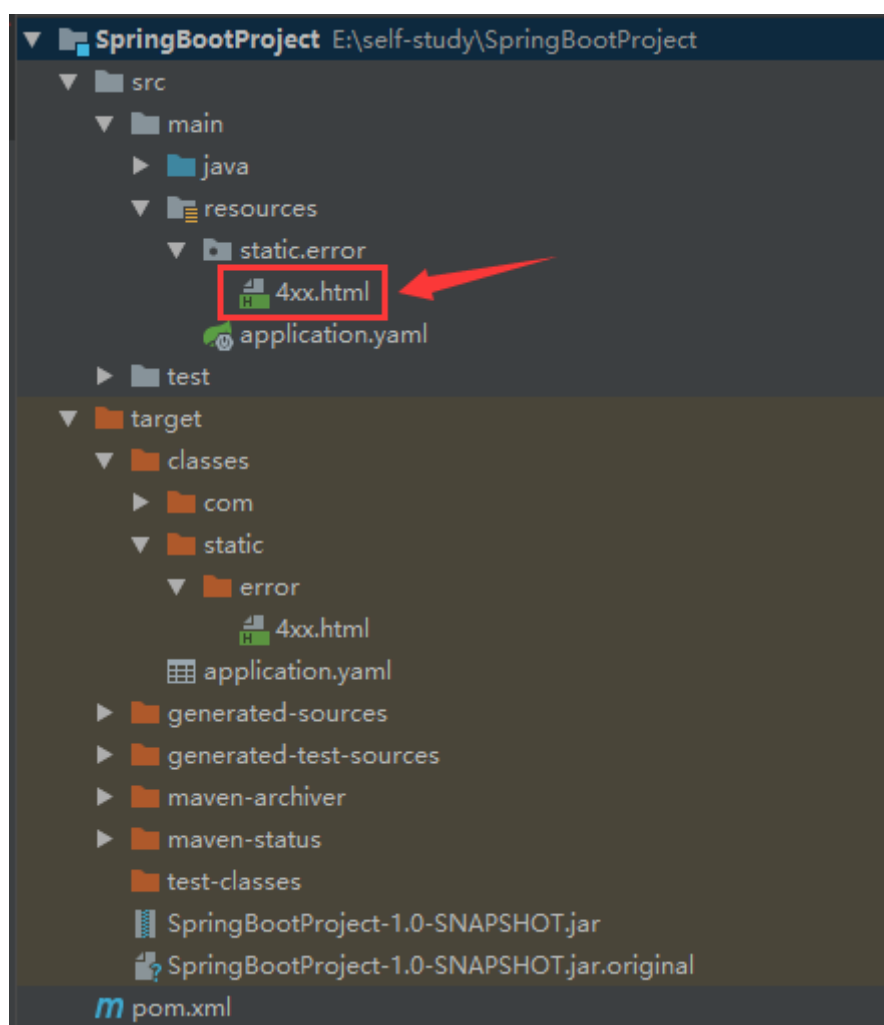
6.5.1 自定义错误页

在类路径下的 `/static`、`/public`、`/resources`、`/META-INF/resources` 任一文件夹中，新建一个 `error` 文件夹，然后在 `error` 文件夹下新增 `4xx.html` 或者 `5xx.html` 等页面，那么当出现错误的时候，错误状态码是4或者5开头的就会自动跳转到对应页面。比如出现404错误会跳转到 `4xx.html`，出现500错误会跳转到 `5xx.html`，演示如下：

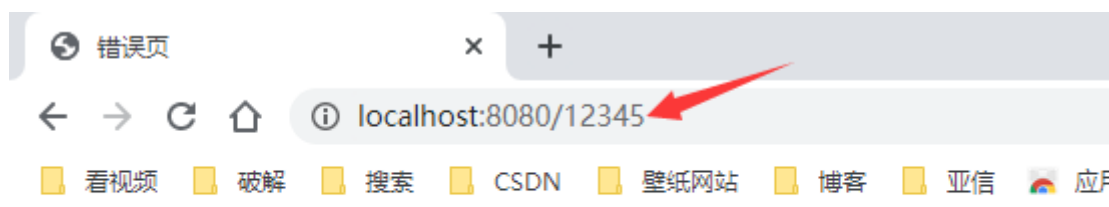
`4xx.html` 文件内容如下：

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>错误页</title>
6 </head>
7 <body>
8 <h3>错误页! </h3>
9 </body>
10 </html>
```

项目结构如下：



启动项目后，浏览器地址栏随便输入一个错误的url后缀，结果如下：



错误页!

通过浏览器结果可知，系统已经自动识别到了我们项目类路径下的 `static/error/4xx.html` 文件。

系统在识别错误页的时候，其实是根据错误状态码进行识别的，而且是先精确匹配再模糊匹配，好比出现了404错误，就会首先找 `404.html` 这个文件，如果找不到，才会去找 `4xx.html`，演示如下：

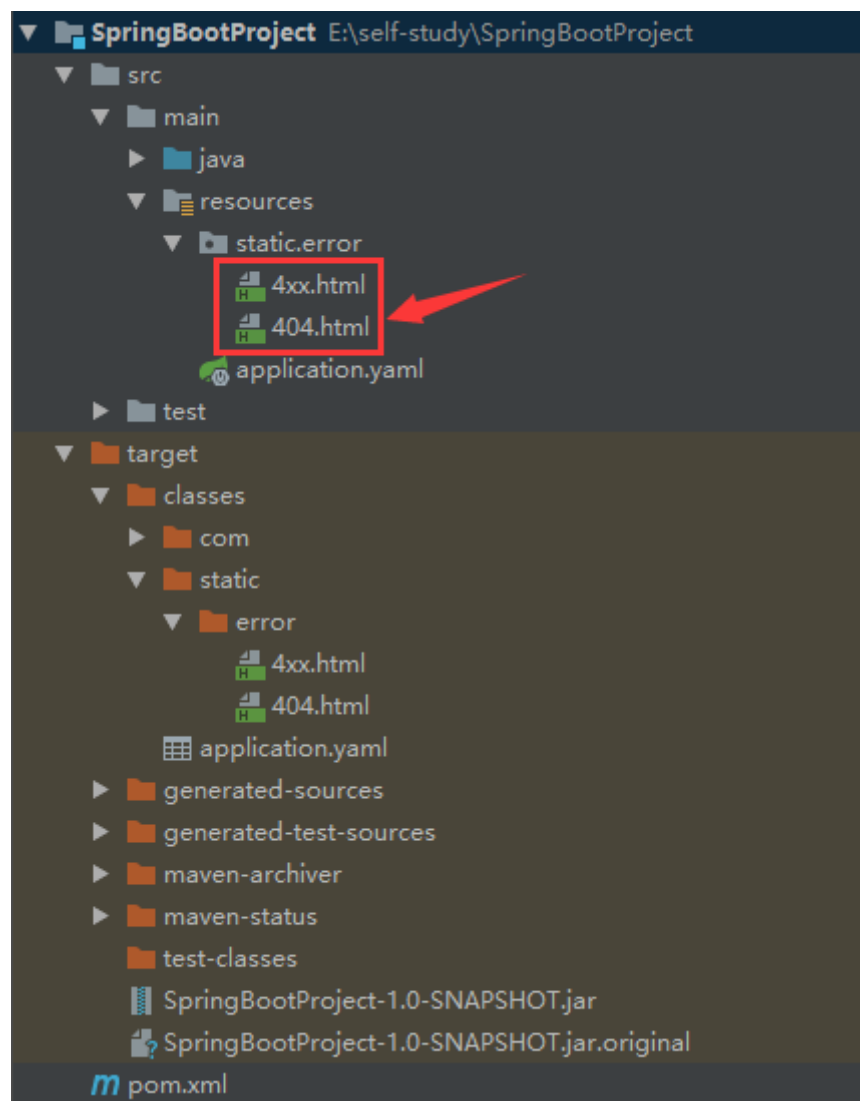
4xx.html文件内容如下：

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>错误页</title>
6 </head>
7 <body>
8 <h3>错误页! </h3>
9 </body>
10 </html>
```

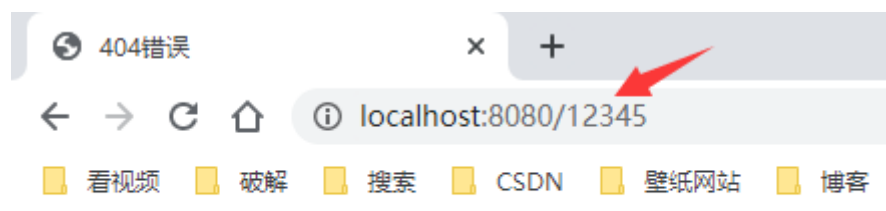
404.html文件内容如下：

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>404错误</title>
6 </head>
7 <body>
8 <h3>请求地址有误! </h3>
9 </body>
10 </html>
```

项目结构如下：



启动项目后，浏览器地址栏随便输入一个错误的url后缀，结果如下：



请求地址有误!

通过浏览器结果可知，当同时存在 `404.html` 和 `4xx.html`，如果出现了404错误，会优先跳转到 `404.html`。

如果修改页面内容不生效，可以使用maven重新编译后再试试。

6.5.2 注解方式实现异常处理

这里主要用到 `@ControllerAdvice`、`@ExceptionHandler`、`@ResponseStatus` 这三个注解。

6.5.2.1 处理指定异常

由于在测试的时候我们只想要返回具体值，而不是跳转页面，所以会用到 `@RestControllerAdvice` 注解，该注解其实就是 `@ControllerAdvice` 注解和 `@ResponseBody` 注解的组合注解，测试如下：

测试类：

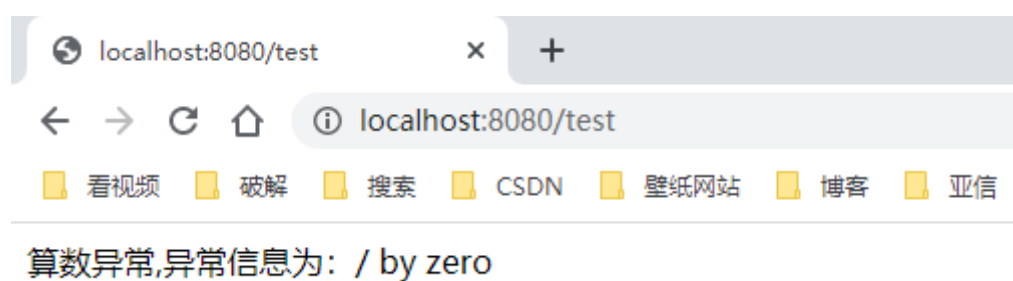
```
1 package com.gongsl.controller;
2
3 import org.springframework.web.bind.annotation.*;
4
5 @RestController
6 public class HelloController {
7
8     @GetMapping("/test")
9     public String home(String str){
10         int i = 10/0;
11         return str;
12     }
13 }
```

新增一个专门用于处理异常的全局异常处理类：

```
1 package com.gongsl.handler;
2
3 import org.springframework.web.bind.annotation.ExceptionHandler;
4 import org.springframework.web.bind.annotation.RestControllerAdvice;
5
6 @RestControllerAdvice
7 public class GlobalExceptionHandler {
8
9     @ExceptionHandler(ArithmeticException.class)
10    public String handleArithmeticException(ArithmeticException e){
11        return "算数异常,异常信息为: "+e.getMessage();
12    }
13 }
```

如果我们不需要获取异常中的信息的话，那么上面那个 `handleArithmeticException` 方法直接使用无参的即可。

启动项目后浏览完的测试结果：



我们在测试类中手动创造了一个 `ArithmeticException` 异常，由测试结果可知，已经被 `GlobalExceptionHandler` 类中专门用于处理该异常的 `handleArithmeticException(ArithmeticException e)` 方法拦截并处理了。

6.5.2.2 处理未指定的其他异常

我们可以在自定义的 `GlobalExceptionHandler` 类中针对常见的异常分别写一个对应的方法来进行针对性处理，对于不常见的其他异常，我们如果想要统一进行处理的话，只要使用 `@ExceptionHandler(Exception.class)` 注解即可，演示案例如下所示：

测试类：

```
1 package com.gongsl.controller;
2
3 import org.springframework.web.bind.annotation.*;
4
5 @RestController
6 public class HelloController {
7
8     @GetMapping("/test")
9     public String home(@RequestParam("str") String str){
10         int i = 10/0;
11         return str;
12     }
13 }
```

这里加一个 `@RequestParam` 注解，但是测试的时候不传参数，以便制造异常场景。

全局异常处理类：

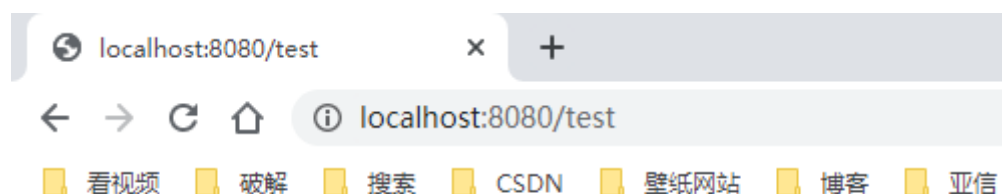
```
1 package com.gongsl.handler;
2
3 import org.springframework.web.bind.annotation.ExceptionHandler;
4 import org.springframework.web.bind.annotation.RestControllerAdvice;
5
6 @RestControllerAdvice
```

```

7 public class GlobalExceptionHandler {
8
9     @ExceptionHandler(ArithmeticException.class)
10    public String handleArithmeticException(ArithmeticException e){
11        return "算数异常,异常信息为: "+e.getMessage();
12    }
13
14    @ExceptionHandler(Exception.class)
15    public String handleException(Exception e){
16        return "其他异常,"+e.getMessage();
17    }
18 }

```

启动项目后浏览器的测试结果：



其他异常, Required String parameter 'str' is not present

针对某种异常，如果全局异常处理类中有专门处理该异常的方法，就会直接调用该方法进行处理。如果没有，就会使用标有 `@ExceptionHandler(Exception.class)` 注解的方法来进行统一处理。

6.5.2.3 @ResponseStatus注解的用法

我们可以使用 `@ResponseStatus` 注解来改变请求应答的状态码。

如果测试类和全局异常处理类都和上面6.5.2.2章节中的一样的话，由于异常已经被全局异常处理类处理了，所以请求返回的状态码是200。如果我们想要改变这个状态码的话，就可以使用 `@ResponseStatus` 注解，演示如下：

测试类：

```

1 package com.gongsl.controller;
2
3 import org.springframework.web.bind.annotation.*;
4
5 @RestController
6 public class HelloController {
7
8     @GetMapping("/test")
9     public String home(@RequestParam("str") String str){
10        int i = 10/0;
11        return str;
12    }
13 }

```

全局异常处理类：

```

1 package com.gongsl.handler;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.web.bind.annotation.ExceptionHandler;
5 import org.springframework.web.bind.annotation.ResponseStatus;
6 import org.springframework.web.bind.annotation.RestControllerAdvice;
7
8 @RestControllerAdvice
9 public class GlobalExceptionHandler {
10
11     @ExceptionHandler(ArithmeticException.class)
12     public String handleArithmeticException(ArithmeticException e){

```

```
13         return "算数异常,异常信息为: "+e.getMessage();
14     }
15
16     @ExceptionHandler(Exception.class)
17     @ResponseStatus(value = HttpStatus.NOT_FOUND)
18     public String handleException(Exception e){
19         return "其他异常,"+e.getMessage();
20     }
21 }
```

这里使用 `@ResponseStatus(value = HttpStatus.NOT_FOUND)` 注解的意思就是把状态码改为404。

启动项目后工具的测试结果：

The screenshot shows a web testing tool interface. The top section is labeled '测试' (Test) and contains a 'METHOD' dropdown set to 'GET' and a URL input field containing 'http://localhost:8080/test'. Below the URL field is a 'Send' button. The bottom section is labeled 'Response' and shows a red bar with the status code '404'. Below the status code is a 'BODY' tab showing the response message: '其他异常,Required String parameter 'str' is not present'.

使用工具进行测试，通过测试结果可知，状态码已经变成404了。

`@ResponseStatus` 注解还有一个 `reason` 属性，用法演示如下：

测试类：

```
1 package com.gongsl.controller;
2
3 import org.springframework.web.bind.annotation.*;
4
5 @RestController
6 public class HelloController {
7
8     @GetMapping("/test")
9     public String home(@RequestParam("str") String str){
10         int i = 10/0;
11         return str;
12     }
13 }
```

全局异常处理类：

```
1 package com.gongsl.handler;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.web.bind.annotation.ExceptionHandler;
5 import org.springframework.web.bind.annotation.ResponseStatus;
6 import org.springframework.web.bind.annotation.RestControllerAdvice;
7
8 @RestControllerAdvice
9 public class GlobalExceptionHandler {
10
```



```

11     @ExceptionHandler(ArithmeticException.class)
12     public String handleArithmeticException(ArithmeticException e){
13         return "算数异常,异常信息为: "+e.getMessage();
14     }
15
16     @ExceptionHandler(Exception.class)
17     @ResponseStatus(value = HttpStatus.GATEWAY_TIMEOUT, reason = "演示使用")
18     public String handleException(Exception e){
19         return "其他异常,"+e.getMessage();
20     }
21 }

```

上面的 `HttpStatus.GATEWAY_TIMEOUT` 指的是状态码设置为504的意思。

启动项目后浏览器的测试结果：



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Thu Jan 28 00:37:11 CST 2021

There was an unexpected error (type=Gateway Timeout, status=504).

演示使用

```

org.springframework.web.bind.MissingServletRequestParameterException: Required String parameter 'str' is not present
    at org.springframework.web.method.annotation.RequestParamMethodArgumentResolver.handleMissingValue(RequestParamMethodArgumentResolver.java:204)
    at org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver.resolveArgument(AbstractNamedValueMethodArgumentResolver.java:114)
    at org.springframework.web.method.support.HandlerMethodArgumentResolverComposite.resolveArgument(HandlerMethodArgumentResolverComposite.java:121)
    at org.springframework.web.method.support.InvocableHandlerMethod.getMethodArgumentValues(InvocableHandlerMethod.java:167)
    at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:134)
    at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:105)
    at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:878)
    at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:792)
    at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87)
    at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1040)
    at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:943)
    at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1006)
    at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:898)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:626)
    at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:883)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:733)
    at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
    at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
    at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
    at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
    at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)

```

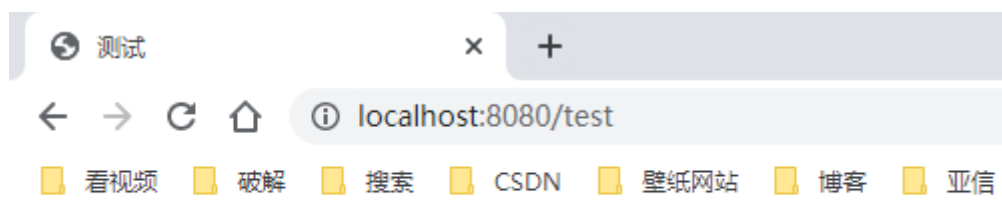
需要注意的是，当我们使用了 `reason` 属性后，默认情况下会到默认的静态资源路径下(比如 `static` 路径)的 `error` 目录中找错误页。由于我们把状态码设置成了504，所以会在 `error` 目录下找有没有 `504.html` 文件或者 `5xx.html` 文件。没找到就会返回上面那一堆报错。如果我们在 `error` 目录下新建了一个 `504.html` 文件，内容如下：

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>测试</title>
6  </head>
7  <body>
8  <h3>504页面测试! </h3>
9  </body>
10 </html>

```

那么用浏览器测试的时候，就会返回如下内容，这也印证了确实到 `error` 目录下找 `504.html` 文件或者 `5xx.html` 文件了。



504页面测试!

如果是使用类似postman等工具进行模拟调用的话，不管项目中有没有504.html或者5xx.html，返回的都是如下信息：

测试

Save

METHOD

SCHEME://HOST[:PORT][PATH[?QUERY]]

GET

http://localhost:8080/test

length: 26 byte(s)

Send

QUERY PARAMETERS

HEADERS

Form

+

Add header

Add authorization

BODY

XHR does not allow payloads for GET request.

Response

Cache Detected - Elapsed Time: 61ms

504

HEADERS

pretty

Content-Type: application/json

Transfer-Encoding: chunked

Date: Wed, 27 Jan 2021 16:44:27 GMT

Keep-Alive: timeout=60

Connection: keep-alive

COMPLETE REQUEST HEADERS

BODY

pretty

```
{
  timestamp : "2021-01-27T16:44:27.730+00:00",
  status : 504,
  error : "Gateway Timeout",
  trace : "org.springframework.web.bind.MissingServletRequestParameterException: Required String parameter 'st'",
  message : "演示使用",
  path : "/test"
}
```

length: 5 kilobytes

返回信息的应答报文体中 message 字段的值就是我们设置的 @ResponseStatus 注解中 reason 属性的值。