

1. Spring Boot 框架

1.1 第一章 Xml 和 JavaConfig

Spring 使用 Xml 作为容器配置文件，在 3.0 以后加入了 JavaConfig. 使用 java 类做配置文件使用。

1.1.1 什么是 JavaConfig

JavaConfig: 是 Spring 提供的使用 java 类配置容器。配置 Spring IOC 容器的纯 Java 方法。

优点:

1. 可以使用面向对象的方式，一个配置类可以继承配置类，可以重写方法
2. 避免繁琐的 xml 配置

1.1.2 Xml 配置容器

创建 001-pre-boot 项目

pom.xml

```
<dependencies>

    <dependency>

        <groupId>org.springframework</groupId>

        <artifactId>spring-context</artifactId>

        <version>5.3.1</version>

    </dependency>
```

```
<dependency>

    <groupId>junit</groupId>

    <artifactId>junit</artifactId>

    <version>4.12</version>

</dependency>

</dependencies>

<build>

    <plugins>

        <!-- 编译插件 -->

        <plugin>

            <artifactId>maven-compiler-plugin</artifactId>

            <!-- 插件的版本 -->

            <version>3.5.1</version>

            <!-- 编译级别 -->

            <configuration>

                <source>1.8</source>

                <target>1.8</target>

                <!-- 编码格式 -->

                <encoding>UTF-8</encoding>

            </configuration>

        </plugin>

    </plugins>

</build>

</project>
```

```
        </configuration>

    </plugin>

</plugins>

</build>
```

创建数据类 Student

```
package com.bjpowernode.vo

public class Student {

    private Integer id;

    private String name;

    private Integer age;

    // set | get

}
```

resources 目录下创建 Spring 的配置文件 applicationContext.xml

```
<bean id="myStudent" class="com.bjpowernode.vo.Student">

    <property name="id" value="1001" />

    <property name="name" value="李强国" />

    <property name="age" value="20" />

</bean>
```

单元测试：

```
@Test

public void test01() {

    String config="applicationContext.xml";

    ApplicationContext ctx = new
ClassPathXmlApplicationContext(config);

    Student student = (Student) ctx.getBean("myStudent");

    System.out.println("student="+student);

}
```

1.1.3 JavaConfig 配置容器

JavaConfig 主要使用的注解

@Configuration:放在类的上面， 这个类相当于 xml 配置文件，可以在其中声明 bean

@Bean:放在方法的上面， 方法的返回值是对象类型， 这个对象注入到 spring ioc 容器

创建配置类（等同于 xml 配置文件）

```
package com.bjpowernode.config;

import com.bjpowernode.vo.Persion;

import com.bjpowernode.vo.Student;

import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Import;

import java.util.Date;

/**
 * @Configuration: 表示当前类是 xml 配置文件的作用
 *
 *      在这个类中有很多方法， 方法的返回值是对象。
 *
 *      在这个方法的上面加入@Bean， 表示方法返回值的对象放入到容器中。
 *
 *      @Bean == <bean></bean>
 */
@Configuration
public class SpringConfig {

    /**
     * 定义方法， 方法的返回值是对象。
     *
     * @Bean: 表示把对象注入到容器中。
     *
     * 位置: 方法的上面
     *
     * @Bean 没有使用属性， 默认对象名称是方法名
     */
    @Bean
```

```
public Student createStudent() {  
  
    Student student = new Student();  
  
    student.setId(1002);  
  
    student.setName("周仓");  
  
    student.setAge(29);  
  
    return student;  
  
}
```

// name : 指定对象的名称

```
@Bean(name = "myStudent2")
```

```
public Student makeStudent() {  
  
    Student student = new Student();  
  
    student.setId(1003);  
  
    student.setName("诸葛亮");  
  
    student.setAge(30);  
  
    return student;  
  
}
```

```
@Bean
```

```
public Date myDate() {  
    return new Date();  
}  
  
}  
  
}
```

测试方法：

```
//使用 JavaConfig  
  
@Test  
  
public void test02() {  
    //没有 xml 配置文件，使用 java 类代替 xml 配置文件 的作用  
  
    ApplicationContext ctx = new  
AnnotationConfigApplicationContext(SpringConfig.class);  
  
    Student student = (Student) ctx.getBean("createStudent");  
  
    System.out.println("student==="+student);  
}  
  
@Test  
  
public void test03() {  
    //没有 xml 配置文件，使用 java 类代替 xml 配置文件 的作用  
  
    ApplicationContext ctx = new  
AnnotationConfigApplicationContext(SpringConfig.class);
```

```
Student student = (Student) ctx.getBean("myStudent2");

System.out.println("myStudent2===" + student);

Object myDate = ctx.getBean("myDate");

System.out.println("myDate = " + myDate);

}
```

1.1.4 @ImportResource

@ImportResource 是导入 xml 配置，等同于 xml 文件的 resources
创建数据类：

```
public class Cat {

    private String cardId;

    private String name;

    private Integer age;

    // set | get

}
```

创建配置文件 beans.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">

    <bean id="myCat" class="com.bjpowernode.vo.Cat">

        <property name="cardId" value="XSET29001" />

        <property name="name" value="tom 猫"/>

        <property name="age" value="3" />

    </bean>

</beans>
```

创建配置类：

```
@Configuration

@ImportResource(value =

{"classpath:beans.xml","classpath:applicationContext.xml"})

public class SystemConfig {

    //使用@Bean 声明 bean

}
```

创建测试方法：

```
@Test
```

```
public void test04() {  
    //没有 xml 配置文件，使用 java 类代替 xml 配置文件 的作用  
  
    ApplicationContext ctx =  
        new AnnotationConfigApplicationContext(SystemConfig.class);  
  
    Cat cat = (Cat) ctx.getBean("myCat");  
  
    System.out.println("cat===" + cat);  
}
```

1.1.5 @PropertyResource

@PropertyResource 是读取 properties 属性配置文件

在 resources 目录下创建 config.properties

```
tiger.name=东北老虎  
tiger.age=6
```

创建数据类 Tiger

```
@Component("tiger")  
  
public class Tiger {  
    @Value("${tiger.name}")  
  
    private String name;
```

```
@Value("${tiger.age}")

private Integer age;

@Override

public String toString() {

    return "Tiger{" +

        "name=' " + name + ' \' ' +

        ", age=" + age +

        ' } ' ;

}

}
```

修改 SystemConfig 文件

```
@Configuration

@PropertySource(value = "classpath:config.properties")

@ComponentScan(value = "com.bjpowernode.vo")

public class SystemConfig {

    //使用@Bean 声明 bean

}
```

创建测试方法

```
@Test

public void test05() {

    //没有 xml 配置文件，使用 java 类代替 xml 配置文件 的作用

    ApplicationContext ctx =

        new AnnotationConfigApplicationContext(SystemConfig.class);

    Tiger tiger = (Tiger) ctx.getBean("tiger");

    System.out.println("Tiger==="+tiger);

}
```

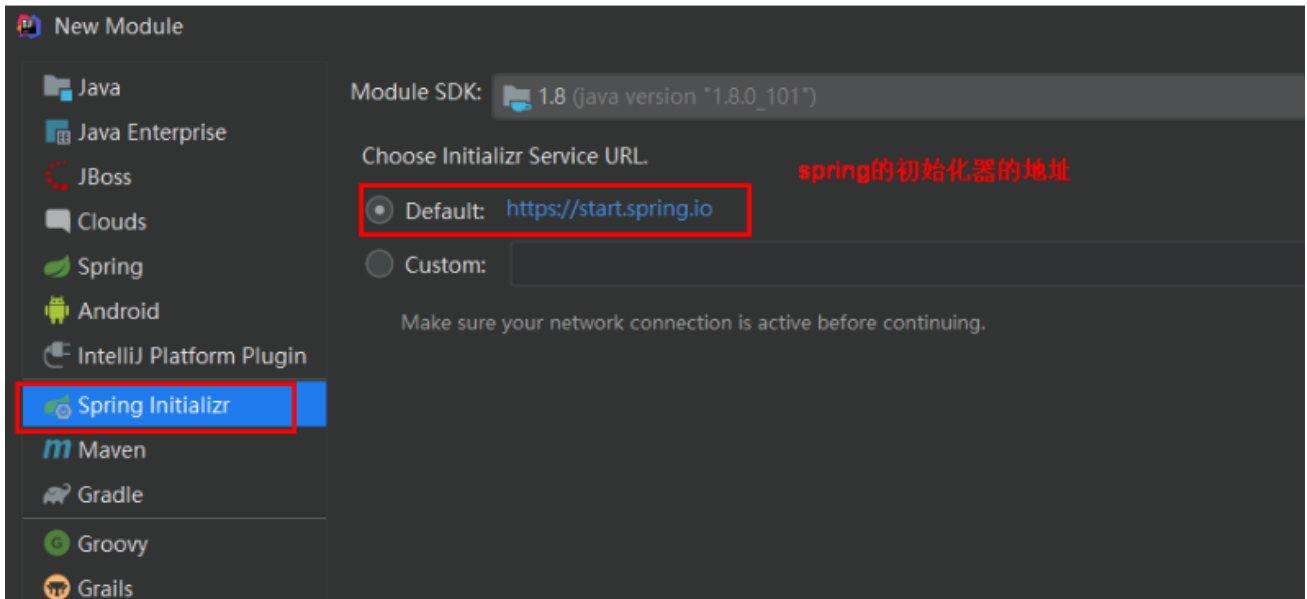
1.2 第二章 Spring Boot 入门

1.2.1 第一种方式: <https://start.spring.io>

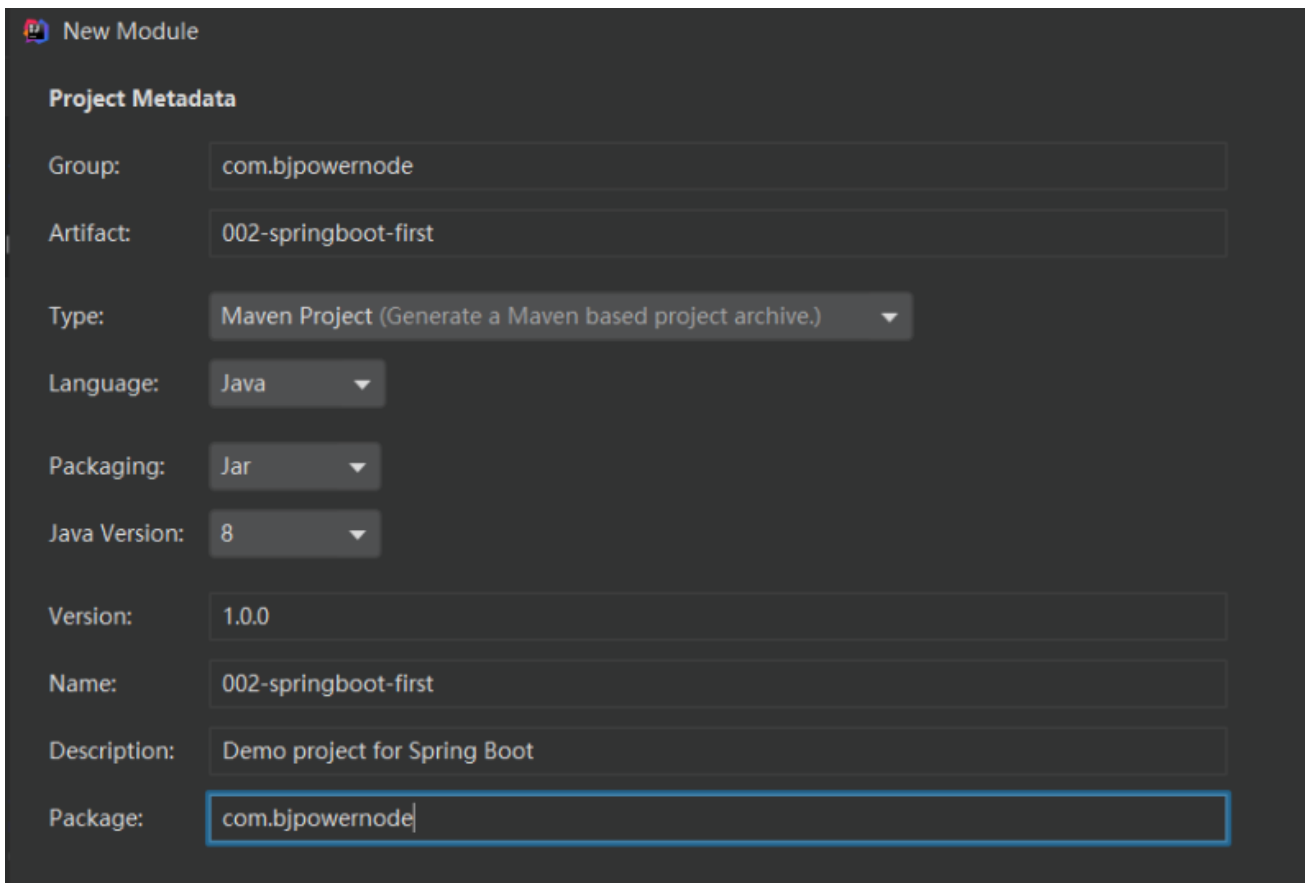
使用 spring boot 提供的初始化器。 向导的方式，完成 spring boot 项目的创建： 使用方便。

1.2.1.1 创建项目步骤

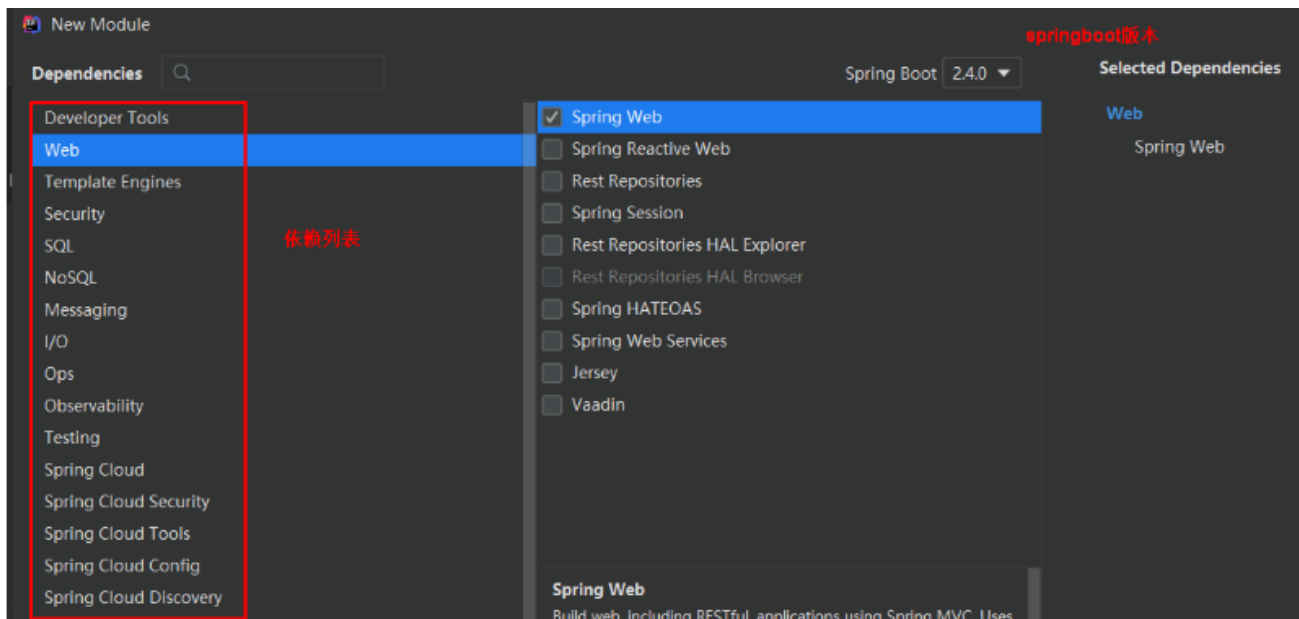
step 1: 新建项目



step 2

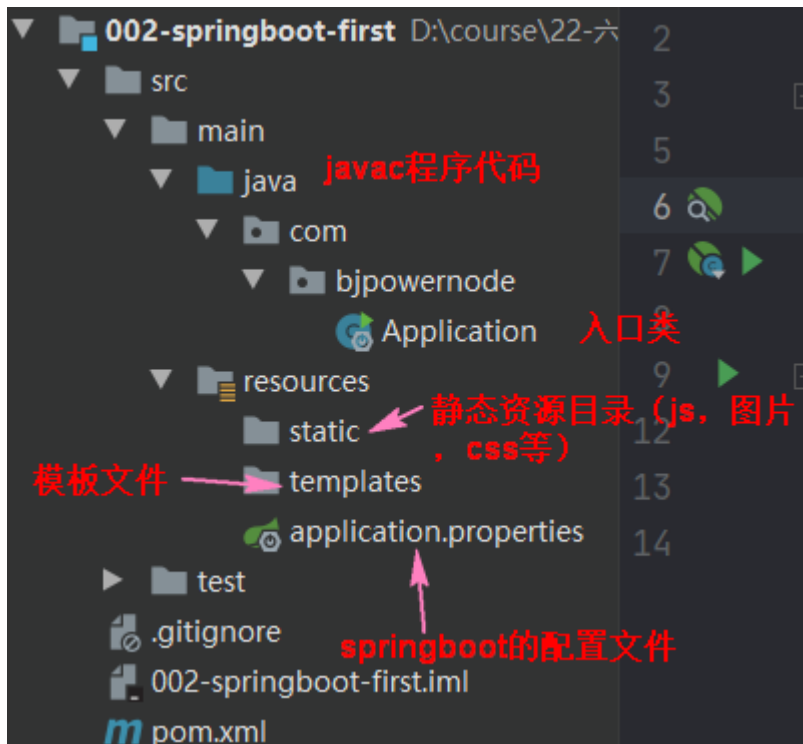


step 3 选择依赖

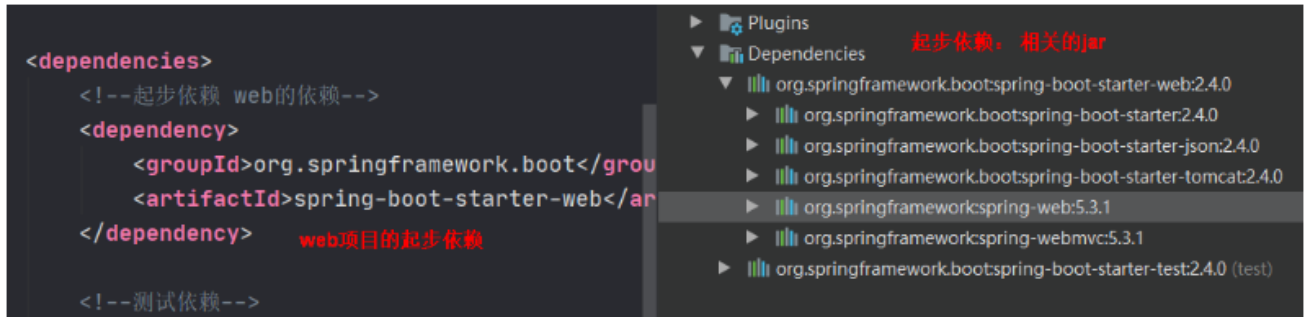


step 4 : 最后创建项目，设置项目的目录位置

step 5: Spring Boot 项目目录结构



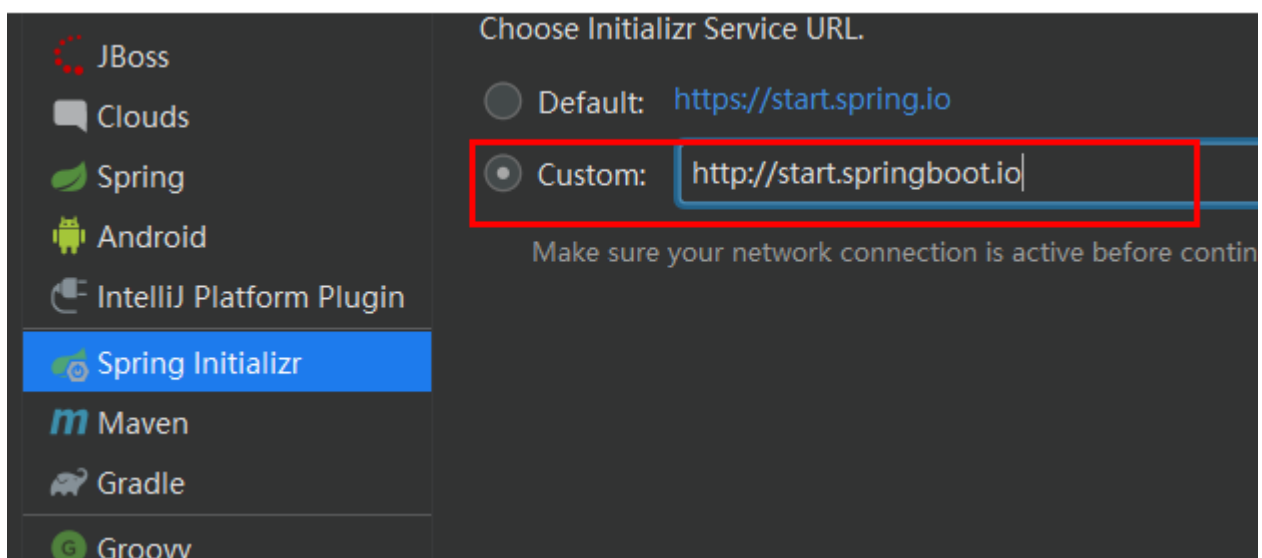
1.2.1.2 起步依赖



1.2.2 第二种方式，使用 springboot 提供的初始化器，使用的国内的地址

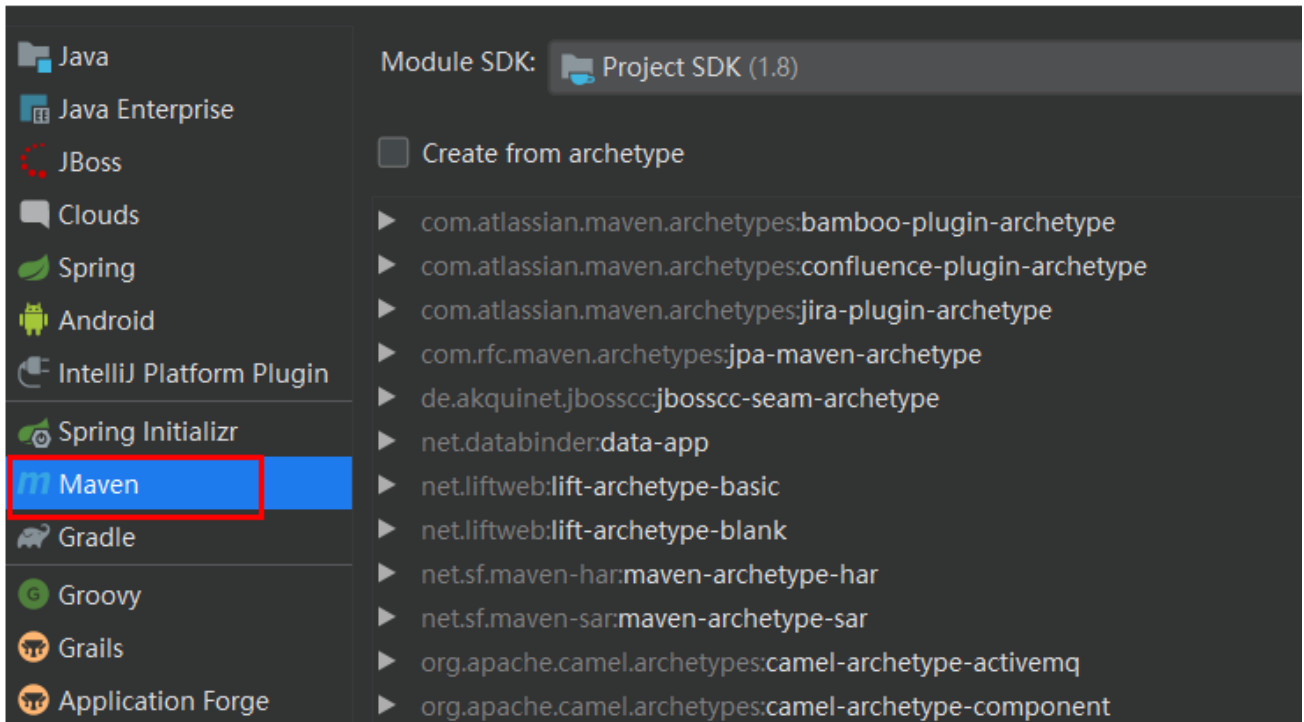
国内地址: <https://start.springboot.io>

创建项目的步骤同上

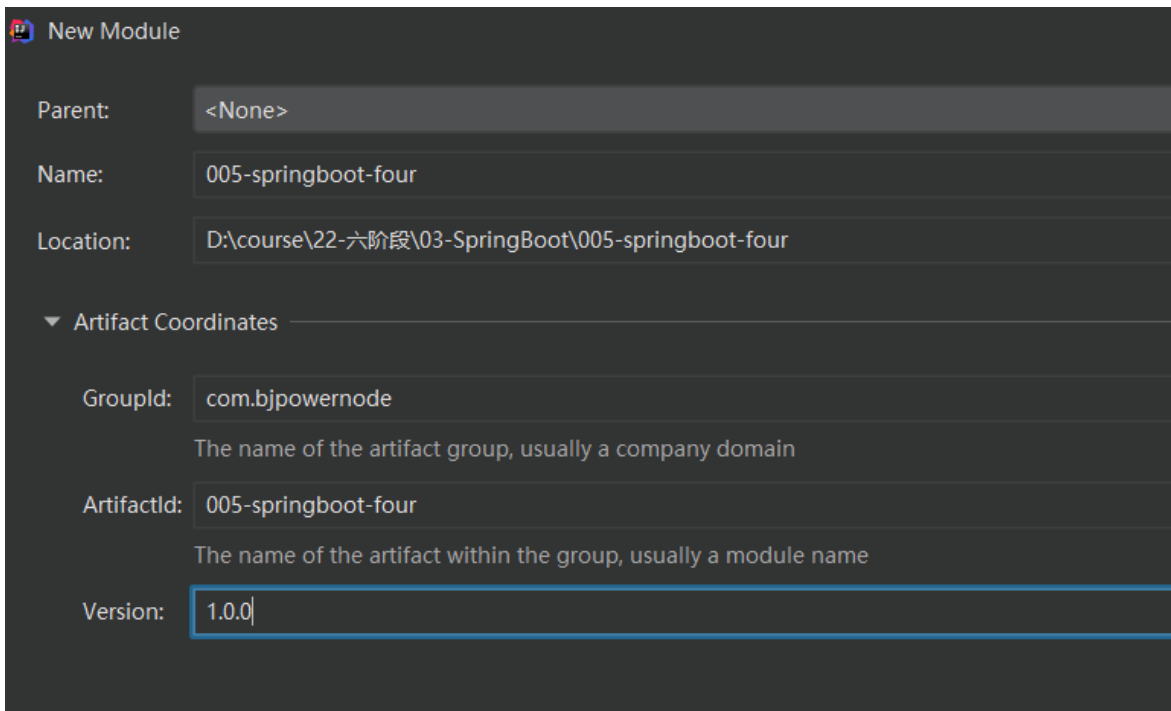


1.2.3 第三种方式 使用 maven 向导创建项目

创建一个普通 maven 项目

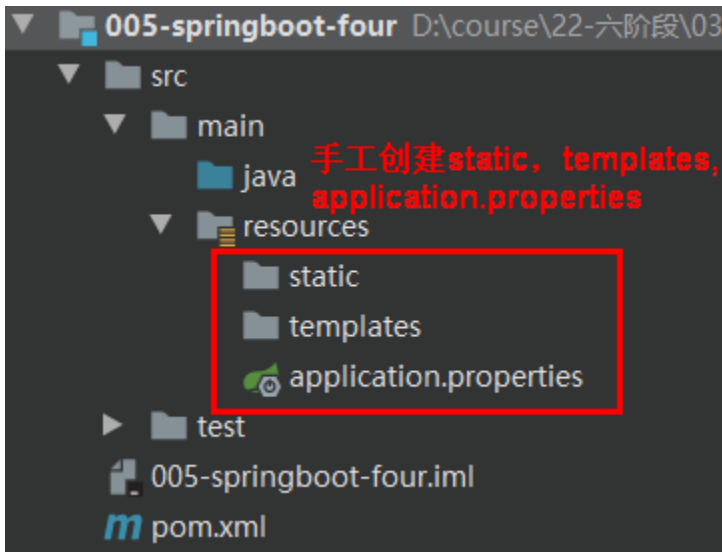


添加 gav



点击 finish 创建，完成项目创建。

修改项目的目录



添加 Spring Boot 依赖

```
<dependencies>

  <dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-web</artifactId>

  </dependency>

  <dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-test</artifactId>

    <scope>test</scope>

  </dependency>

</dependencies>
```

```
</dependencies>

<build>

    <plugins>

        <plugin>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-maven-plugin</artifactId>

        </plugin>

    </plugins>

</build>
```

创建启动类：加入@SpringBootApplication 注解

```
package com.bjpowernode;

@SpringBootApplication

public class MySpringBootMain {

    public static void main(String[] args) {

        SpringApplication.run(MySpringBootMain.class, args);

    }

}
```

1.2.3.1 入门案例

1.2.3.2 入门案例分析

1.2.3.3 重要注解

`@SpringBootApplication` : `@SpringBootApplication` 是一个复合注解，是由 `@SpringBootConfiguration`，`@EnableAutoConfiguration`，`@ComponentScan` 联合在一起组成的。

`@SpringBootConfiguration` : 就是 `@Configuration` 这个注解的功能，使用 `@SpringBootConfiguration` 这个注解的类就是配置文件的作用。

`@EnableAutoConfiguration`: 开启自动配置，把一些对象加入到 spring 容器中。

`@ComponentScan`: 组件扫描器，扫描注解，根据注解的功能，创建 java bean，给属性赋值等等。组件扫描器默认扫描的是 `@ComponentScan` 注解所在的类，类所在的包和子包。

1.2.4 Spring Boot 核心配置文件

Spring Boot 的核心配置文件用于配置 Spring Boot 程序，名字必须以 application 开始

1.2.4.1 .properties 文件（默认采用该文件）

在 002-springboot-springmvc 项目基础上，进行修改

项目名称: 003-springboot-port-context-path

通过修改 application.properties 配置文件,在修改默认 tomcat 端口号及项目上下文件根键值对的 properties 属性文件配置方式

```
#设置访问端口号
```

```
server.port=9092

#应用的 contextpath

server.servlet.context-path=/boot
```

启动应用， 在浏览器访问 <http://localhost:9092/boot/>

1.2.4.2 .yaml 文件

项目名称：005-springboot-yml，在之前项目基础之上

yaml 是一种 yaml 格式的配置文件，主要采用一定的空格、换行等格式排版进行配置。

yaml 是一种直观的能够被计算机识别的数据序列化格式，容易被人类阅读，yaml 类似于 xml，但是语法比 xml 简洁很多，值与前面的冒号配置项必须要有一个空格，yaml 缀也可以使用 yaml 后缀

```
server:

  port: 9091

  servlet:

    context-path: /boot
```

注意：当两种格式配置文件同时存在，在 SpringBoot2.4 开始，使用的是 yaml 配置文件。修改配置名称都为 application。

重新运行 Application，查看启动的端口及上下文根

推荐使用 yaml 格式配置文件

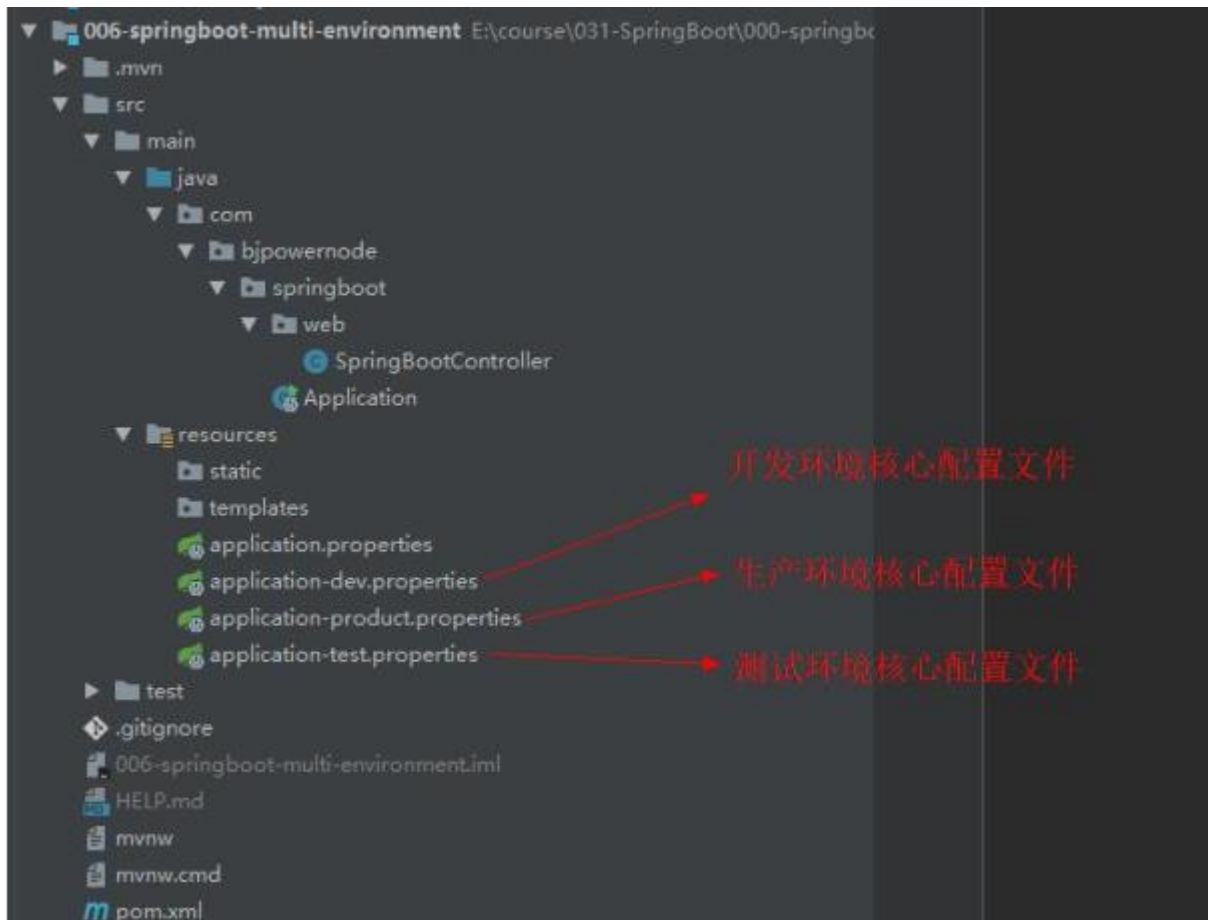
1.2.4.3 多环境配置

在实际开发的过程中，我们的项目会经历很多的阶段（开发->测试->上线），每个阶段

的配置也会不同，例如：端口、上下文根、数据库等，那么这个时候为了方便在不同的环境之间切换，SpringBoot 提供了多环境配置，具体步骤如下

项目名称：006-springboot-multi-environment

为每个环境创建一个配置文件，命名必须以 application-环境标识.properties|yaml



application.properties

```
#指定使用的环境文件
```

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

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

application-dev.properties

```
#开发环境的信息
```

```
server.port=9091
```

```
server.servlet.context-path=/mydev
```

```
application-product.properties
```

```
server.port=9093
```

```
server.servlet.context-path=/myprod
```

```
application-test.properties
```

```
#开发环境的信息
```

```
server.port=9091
```

```
server.servlet.context-path=/mydev
```

1.2.4.4 Spring Boot 自定义配置

SpringBoot 的核心配置文件中，除了使用内置的配置项之外，我们还可以在自定义配置，然后采用如下注解去读取配置的属性值

1.2.4.4.1 @Value 注解

@Value("\${key}") ， key 来自 application.properties(yml)

application.properties: 添加两个自定义配置项 school.name 和 school.website。在 IDEA 中可以看到这两个属性不能被 SpringBoot 识别，背景是桔色的

```
server.port=9090

server.servlet.context-path=/myboot


school.name=动力节点

school.address=北京大兴区

school.website=www.bjpowernode.com


site=www.bjpowernode.com
```

读取配置文件数据

```
@Controller

public class HelloController {

    @Value("${server.port}")

    private String port;

    @Value("${school.name}")

    private String name;


    @Value("${site}")

    private String site;
```

```
@RequestMapping("/hello")

@ResponseBody

public String doHello() {

    return "hello, port:" + port + "学校: "+name+", 网站: "+site;

}

}
```

启动应用 Application ， 访问浏览器

1.2.4.4.2 @ConfigurationProperties

项目名称：008-springboot-custom-configuration

将整个文件映射成一个对象，用于自定义配置项比较多的情况

案例演示

在 com.bjpowernode.springboot.config 包下创建 SchoolInfo 类，并为该 类加上 Component 和 ConfigurationProperties 注解，prefix 可以不指定，如果不指定，那么会去配置文件中寻找与该类的属性名一致的配置，prefix 的作用可以区分同名配置

```
@Component

@ConfigurationProperties(prefix = "school")

public class SchoolInfo {

    private String name;
```



```
private String address;

private String website;

// set | get 方法

}
```

创建 SchoolController

```
@Controller

public class SchoolController {

    @Resource

    private SchoolInfo schoolInfo;

    @RequestMapping("/myschool")

    @ResponseBody

    public String doSchool() {

        return "学校信息: "+schoolInfo.toString();

    }

}
```

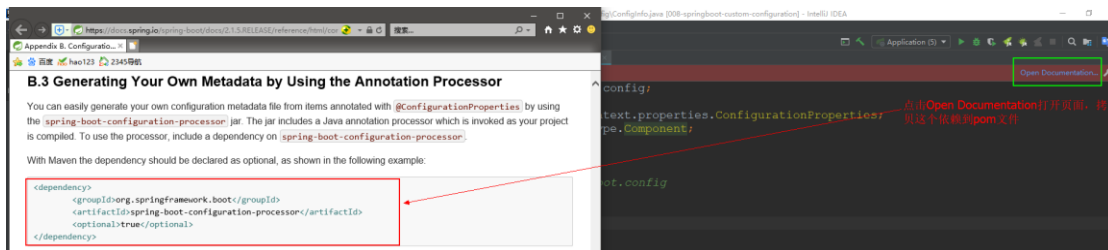
执行 Application ， 访问浏览器查看数据

1.2.4.4.3 警告解决

- 在 SchoolInfo 类中使用了 ConfigurationProperties 注解，IDEA 会出现一个警告，不影响程序的执行



- 点击 open documentnaton 跳转到网页，在网页中提示需要加一个依赖，我们将这个依赖拷贝，粘贴到 pom.xml 文件中



【解决使用@ConfigurationProperties 注解出现警告问题】

```
<dependency>

  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-configuration-processor</artifactId>

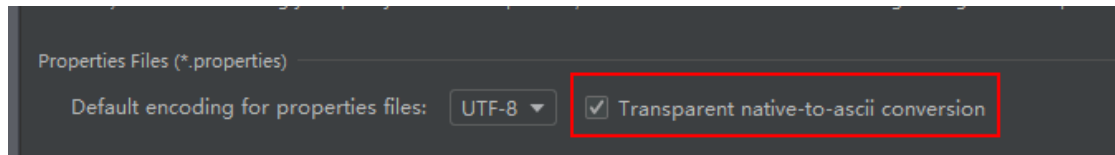
  <optional>true</optional>

</dependency>
```

1.2.4.4.4 中文乱码

如果在 SpringBoot 核心配置文件中存在中文信息，会出现乱码：

- 一般在配置文件中，不建议出现中文（注释除外）
- 如果有，可以先转化为 ASCII 码



1.2.4.4.5 提示

大家如果是从其它地方拷贝的配置文件，一定要将里面的空格删干净

1.2.5 Spring Boot 中使用 JSP

1.2.5.1 在 pom.xml 文件中配置以下依赖项

```
<!--引入 Spring Boot 内嵌的 Tomcat 对 JSP 的解析包，不加解析不  
了 jsp 页面-->  
  
<!--如果只是使用 JSP 页面，可以只添加该依赖-->  
  
<dependency>  
    <groupId>org.apache.tomcat.embed</groupId>  
    <artifactId>tomcat-embed-jasper</artifactId>  
</dependency>  
  
<!--如果要使用 servlet 必须添加该以下两个依赖-->  
<!-- servlet 依赖的 jar 包-->  
  
<dependency>
```

```
<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

</dependency>

<!-- jsp 依赖 jar 包-->

<dependency>

    <groupId>javax.servlet.jsp</groupId>

    <artifactId>javax.servlet.jsp-api</artifactId>

    <version>2.3.1</version>

</dependency>

<!--如果使用 JSTL 必须添加该依赖-->

<!--jstl 标签依赖的 jar 包 start-->

<dependency>

    <groupId>javax.servlet</groupId>

    <artifactId>jstl</artifactId>

</dependency>
```

1.2.5.2 在 pom.xml 的 build 标签中要配置以下信息

SpringBoot 要求 jsp 文件必须编译到指定的 META-INF/resources 目录下才能访问，否则

访问不到。其实官方已经更建议使用模板技术（后面会讲模板技术）

```
<!--  
    SpringBoot 要求 jsp 文件必须编译到指定的  
META-INF/resources 目录下才能访问，否则访问不到。  
    其它官方已经建议使用模版技术（后面会课程会单独讲解模版技  
术）  
-->  
<resources>  
    <resource>  
        <!--源文件位置-->  
        <directory>src/main/webapp</directory>  
        <!--指定编译到 META-INF/resource, 该目录不能随便写-->  
        <targetPath>META-INF/resources</targetPath>  
        <!--指定要把哪些文件编译进去，**表示 webapp 目录及子  
目录，*.*表示所有文件-->  
        <includes>  
            <include>**/*.*</include>  
        </includes>  
    </resource>
```

```
</resources>
```

1.2.5.3 在 application.properties 文件配置 Spring MVC 的视图展示为 jsp，这里相当于 Spring MVC 的配置

```
#配置 SpringMVC 的视图解析器  
  
#其中：/相当于 src/main/webapp 目录  
  
spring.mvc.view.prefix=/  
spring.mvc.view.suffix=.jsp
```

集成完毕之后，剩下的步骤和我们使用 Spring MVC 一样

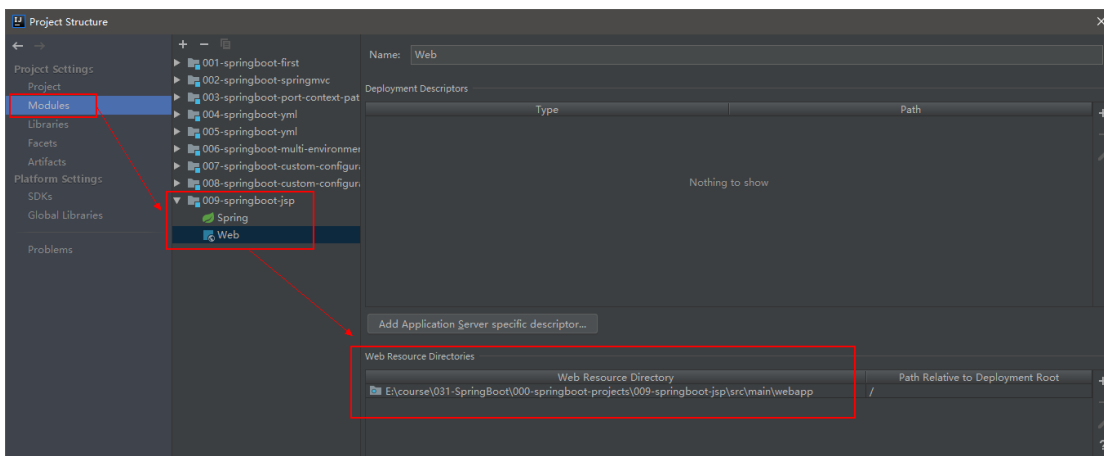
1.2.5.4 在 com.bjpowernode.springboot.controller 包下创建 JspController 类，并编写代码

```
@Controller  
  
public class SpringBootController {  
  
    @RequestMapping(value = "/springBoot/jsp")  
    public String jsp(Model model) {  
  
        model.addAttribute("data", "SpringBoot 前端使用 JSP")  
    }  
}
```

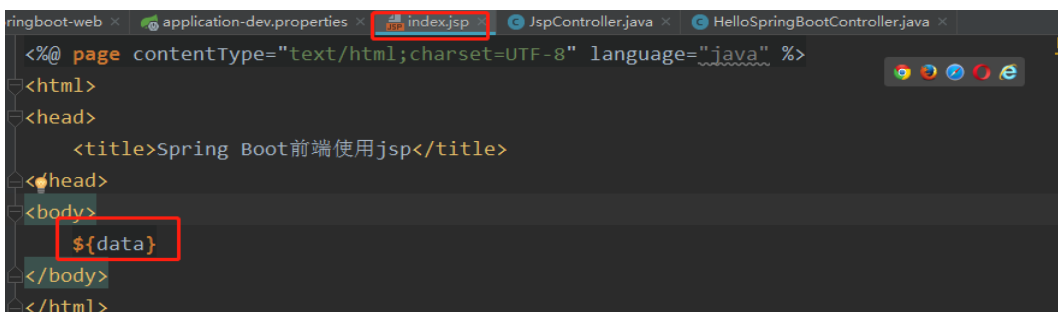
```
页面!");  
  
    return "index";  
  
}  
  
}
```

1.2.5.5 在 src/main 下创建一个 webapp 目录，然后在该目录下新建 index.jsp 页面

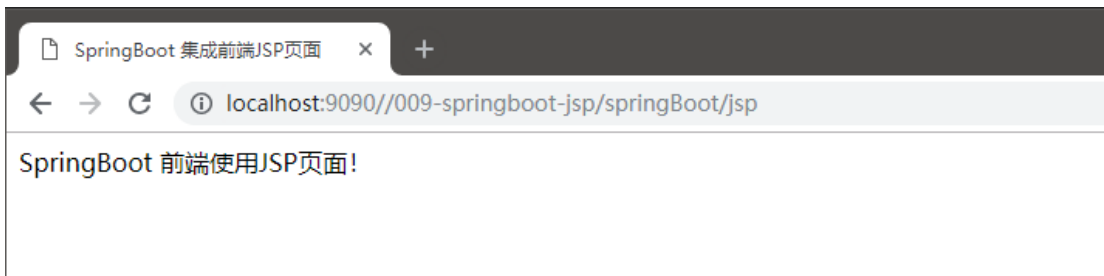
如果在 webapp 目录下右键，没有创建 jsp 的选项，可以在 Project Structure 中指定 webapp 为 Web Resource Directory



1.2.5.6 在 jsp 中获取 Controller 传递过来的数据



1.2.5.7 重新运行 Application，通过浏览器访问测试



1.2.6 Spring Boot 中使用 ApplicationContext

在 main 方法中 SpringApplication.run() 方法获取返回的 Spring 容器对象，再获取业务 bean 进行调用。

创建 Spring Boot 项目：010-springboot-container

指定项目的 gav 和版本等信息

Project Metadata

Group:

Artifact:

Type:

Language:

Packaging:

Java Version:

Version:

Name:

Description:

Package:

选择依赖:

New Module

Dependencies

Developer Tools

Web

Template Engines

Security

SQL

NoSQL

Messaging

I/O

Ops

Spring Boot 2.4.1

Spring Boot DevTools

Lombok

Spring Configuration Processor

不用选择依赖

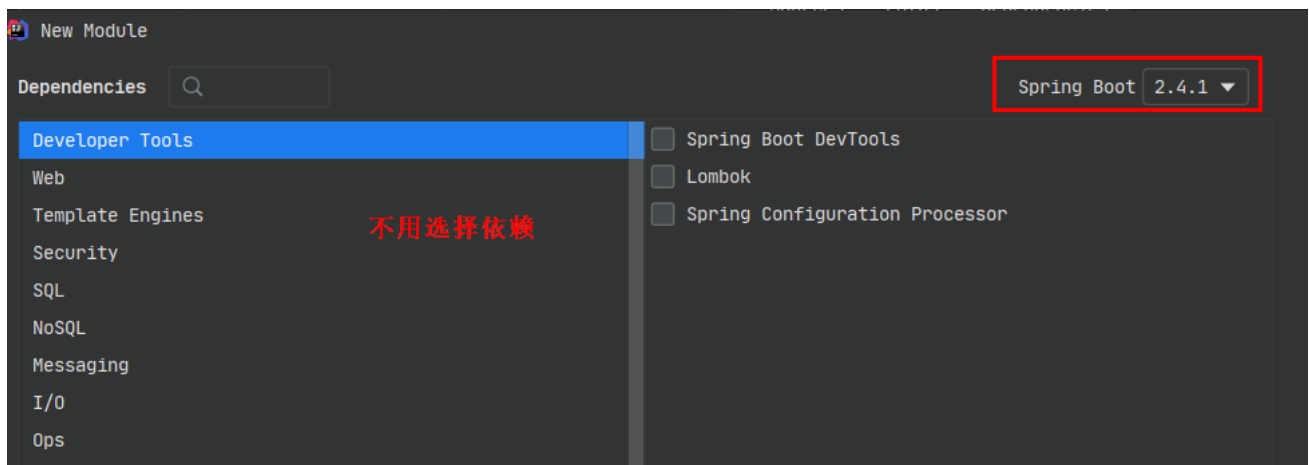
创建一个接口 UserService 和他的实现类

创建启动类， main 方法中获取容器对象

1.2.7 CommandLineRunner 接口

开发中可能会有这样的情景。需要在容器启动后执行一些内容。比如读取配置文件，数据库连接之类的。SpringBoot 给我们提供了两个接口来帮助我们实现这种需求。这两个接口分别为 CommandLineRunner 和 ApplicationRunner。他们的执行时机为容器启动完成的时候。这两个接口中有一个 run 方法，我们只需要实现这个方法即可。这两个接口的不同之处在于：ApplicationRunner 中 run 方法的参数为 ApplicationArguments，而 CommandLineRunner 接口中 run 方法的参数为 String 数组

创建 Spring Boot 项目，不用选依赖，或者修改 010-springboot-container



创建 SomeService 接口和实现类，定义 sayHello() 方法

```
public interface SomeService {  
  
    void sayHello(String name);  
  
}
```

```
@Service("someService")
```

```
public class SomeServiceImpl implements SomeService {

    @Override

    public void sayHello(String name) {

        System.out.println("欢迎:"+name);

    }

}
```

实现 CommandLineRunner 接口

```
@SpringBootApplication

public class Application implements CommandLineRunner {

    public static void main(String[] args) {

        //run 方法返回值是容器对象

        //1. 创建容器对象

        ConfigurableApplicationContext ctx =

SpringApplication.run(Application.class, args);

        SomeService service = (SomeService) ctx.getBean("someService");

        service.sayHello("李四");

    }

    @Override
```

```
public void run(String... args) throws Exception {  
  
    //2. 容器对象创建好，执行 run  
  
    System.out.println("输出， 在容器对象创建好后执行的代码");  
  
}  
  
}
```

运行主类，查看输出结果

1.3 第三章 Spring Boot 和 web 组件

1.3.1 SpringBoot 中拦截器

SpringMVC 使用拦截器

- 1) 自定义拦截器类，实现 HandlerInterceptor 接口
- 2) 注册拦截器类

```
<mvc:interceptors>  
  <mvc:interceptor>  
    <mvc:mapping path="/loan/**"/>  
    <mvc:exclude-mapping path="/loan/loan"/>  
    <mvc:exclude-mapping path="/loan/loanInfo"/>  
    <mvc:exclude-mapping path="/loan/checkPhone"/>  
    <mvc:exclude-mapping path="/loan/checkCaptcha"/>  
    <mvc:exclude-mapping path="/loan/register"/>  
    <mvc:exclude-mapping path="/loan/login"/>  
    <mvc:exclude-mapping path="/loan/loadStat"/>  
    <mvc:exclude-mapping path="/loan/message/register"/>  
    <mvc:exclude-mapping path="/loan/messageCode"/>  
    <mvc:exclude-mapping path="/loan/wxpayNotify"/>  
    <bean class="com.bjpowernode.p2p.interceptor.UserInterceptor"/>  
  </mvc:interceptor>  
</mvc:interceptors>
```

Spring Boot 使用拦截器步骤:

1. 创建类实现 HandlerInterceptor 接口

```
package com.bjpowernode.web;

import org.springframework.web.servlet.HandlerInterceptor;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginInterceptor implements HandlerInterceptor {

    @Override

    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {

        System.out.println("执行了 LoginInterceptor, preHandle()");

        return true;

    }

}
```

2. 注册拦截器对象

```
//相当于 springmvc 配置文件

@Configuration

public class MyAppConfig implements WebMvcConfigurer {
```

```
//注册拦截器对象的

@Override

public void addInterceptors(InterceptorRegistry registry) {

    // InterceptorRegistry 登记系统中可以使用的拦截器

    // 指定拦截的地址

    String path [] = {"/user/**"};

    String excludePath [] = {"/user/login"};

    registry.addInterceptor( new LoginInterceptor())

                .addPathPatterns(path).excludePathPatterns(excludePath);

}

}
```

3. 创建测试使用的 Controller

```
@Controller

public class BootController {

    @RequestMapping("/user/account")

    @ResponseBody

    public String queryAccount() {
```

```
        return "user/account";
    }

    @RequestMapping("/user/login")
    @ResponseBody
    public String login() {
        return "/user/login";
    }
}
```

4. 主启动类

```
@SpringBootApplication
public class Application {

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

5. 启动主类， 运行浏览器

访问 user/account , user/login 观察拦截的输出语句

1.3.2 Spring Boot 中使用 Servlet

ServletRegistrationBean 用来做在 servlet 3.0+容器中注册 servlet 的功能，但更具有 SpringBean 友好性。

实现步骤：

1. 创建 Servlet

```
public class MyServlet extends HttpServlet {

    @Override

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException {

        doPost(req, resp);

    }

    @Override

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException {

        resp.setContentType("text/html;charset=utf-8");

        PrintWriter out = resp.getWriter();

        out.println("使用 Servlet 对象");

    }

}
```



```
        out.flush();

        out.close();

    }

}
```

2. 注册 Servlet

```
@Configuration

public class SystemConfig {

    @Bean

    public ServletRegistrationBean servletRegistrationBean() {

        /* ServletRegistrationBean reg = new ServletRegistrationBean();

        reg.setServlet( new MyServlet());

        reg.addUrlMappings("/myservlet");

        return reg;*/

        ServletRegistrationBean reg = new ServletRegistrationBean(

            new MyServlet(),

            "/loginServlet");
    }
}
```

```
        return reg;

    }

}
```

3. 主启动类

```
@SpringBootApplication

public class Application {

    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);

    }

}
```

4. 启动主类，在浏览器中访问 loginServlet

1.3.3 Spring Boot 中使用 Filter

FilterRegistrationBean 用来注册 Filter 对象

实现步骤：

1. 创建 Filter 对象

```
public class MyFilter implements Filter {
```

```
@Override

public void doFilter(ServletRequest servletRequest,

                    ServletResponse servletResponse,

                    FilterChain filterChain) throws IOException,

ServletException {

    System.out.println("使用了 Servlet 中的 Filter 对象");

    filterChain.doFilter(servletRequest, servletResponse);

}

}
```

2. 注册 Filter

```
@Configuration

public class SystemConfig {

    @Bean

    public FilterRegistrationBean filterRegistrationBean() {

        FilterRegistrationBean reg = new FilterRegistrationBean();

        //使用哪个过滤器

        reg.setFilter(new MyFilter());

        //过滤器的地址

    }

}
```

```
        reg.addUrlPatterns("/user/*");

        return reg;

    }

}
```

3. 创建 Controller

```
@Controller

public class FilterController {

    @RequestMapping("/user/account")

    @ResponseBody

    public String hello() {

        return "/user/account";

    }

    @RequestMapping("/query")

    @ResponseBody

    public String doSome() {

        return "/query";

    }

}
```

```
}  
  
}
```

4. 启动应用， 在浏览器访问 `user/account`， `/query` 查看浏览器运行结果

1.3.4 字符集过滤器的应用

创建项目：014-springboot-character-filter

实现步骤：

1. 创建 Servlet， 输出中文数据

```
public class MyServlet extends HttpServlet {  
  
    @Override  
  
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)  
throws ServletException, IOException {  
  
        resp.setContentType("text/html");  
  
        PrintWriter out = resp.getWriter();  
  
        out.println("学习 Hello SpringBoot 框架， 自动配置");  
  
        out.flush();  
  
        out.close();  
  
    }  
  
}
```

2) 注册 Servlet 和 Filter

```
package com.bjpowernode.config;

import com.bjpowernode.servlet.MyServlet;

import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.boot.web.servlet.ServletRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.filter.CharacterEncodingFilter;

@Configuration

public class SystemAppliationConfig {

    @Bean

    public ServletRegistrationBean servletRegistrationBean() {

        ServletRegistrationBean reg = new ServletRegistrationBean(new
MyServlet(), "/myservlet");

        return reg;
    }
}
```

```
}

//注册 Filter

@Bean

public FilterRegistrationBean filterRegistrationBean() {

    FilterRegistrationBean reg = new FilterRegistrationBean();

    //创建 CharacterEncodingFilter

    CharacterEncodingFilter filter = new CharacterEncodingFilter();

    //设置 request, response 使用编码的值

    filter.setEncoding("utf-8");

    //强制 request, response 使用 encoding 的值

    filter.setForceEncoding(true);

    reg.setFilter(filter);

    //请求先使用过滤器处理

    reg.addUrlPatterns("/*");

    return reg;
}
```

```
}  
  
}
```

3. 在 application.properties , 禁用 Spring Boot 中默认启用的过滤器

```
#启用自己的 CharacterEncodingFtitler 的设置
```

```
server.servlet.encoding.enabled=false
```

4. 启动主类，运行浏览器

1.3.5 在 application.properties 文件中设置过滤器

Spring Boot 项目默认启用了 CharacterEncodingFilter, 设置他的属性就可以

```
#设置 spring boot 中 CharacterEncodingFtitler 的属性值
```

```
server.servlet.encoding.enabled=true
```

```
server.servlet.encoding.charset=utf-8
```

```
#强制 request, response 使用 charset 他的值 utf-8
```

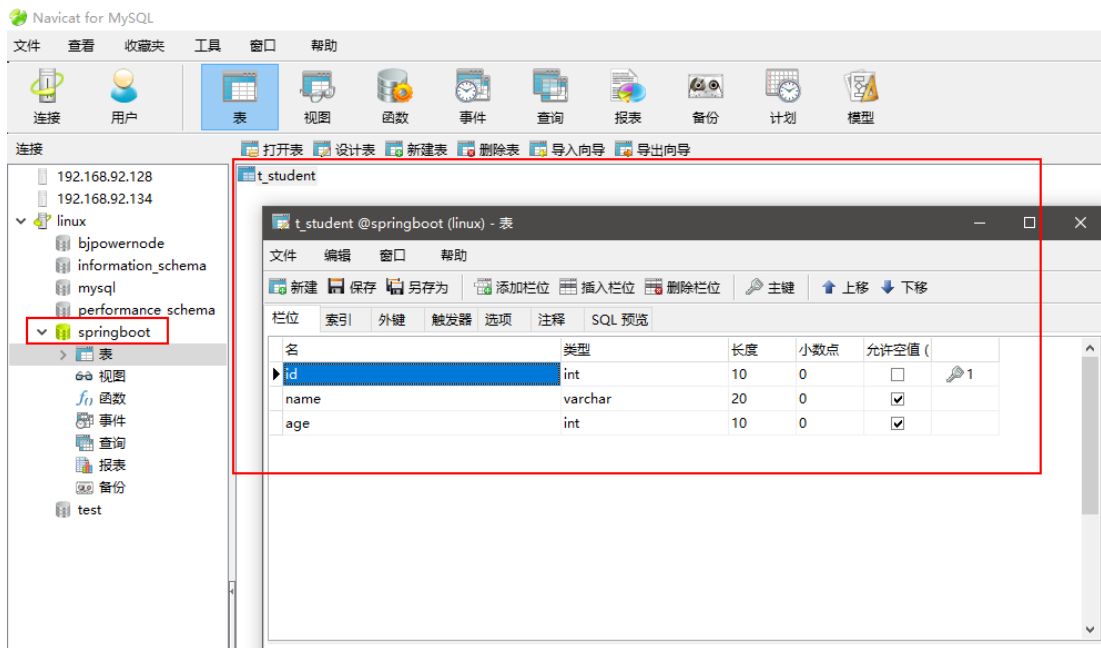
```
server.servlet.encoding.force=true
```

1.4 第四章 ORM 操作 MySQL

讲解 MyBatis 框架，读写 MySQL 数据。通过 SpringBoot +MyBatis 实现对数据库学生表的查询操作。

数据库参考：springboot.sql 脚本文件

创建数据库：数据库 springboot，指定数据库字符编码为 utf-8



插入数据

id	name	age
1	zs	15
2	ls	18
3	ww	20

1.4.1 创建 Spring Boot 项目

项目名称: 015-springboot-mybatis

使用@Mapper 注解

➤ pom.xml

```
<dependencies>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-web</artifactId>
```

```
</dependency>

<!--mybatis 起步依赖: mybatis 框架需要的依赖全部加入好-->

<dependency>

    <groupId>org.mybatis.spring.boot</groupId>

    <artifactId>mybatis-spring-boot-starter</artifactId>

    <version>2.1.4</version>

</dependency>

<!--mysql 驱动-->

<dependency>

    <groupId>mysql</groupId>

    <artifactId>mysql-connector-java</artifactId>

    <scope>runtime</scope>

</dependency>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-test</artifactId>

    <scope>test</scope>
```

```
</dependency>  
  
</dependencies>
```

加入 resources 插件

```
<!--加入 resource 插件-->  
  
<resources>  
  
  <resource>  
  
    <directory>src/main/java</directory>  
  
    <includes>  
  
      <include>**/*.xml</include>  
  
    </includes>  
  
  </resource>  
  
</resources>
```

➤ 配置数据源: application.properties

```
server.port=9090  
  
server.servlet.context-path=/myboot  
  
#连接数据库  
  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
  
spring.datasource.url=jdbc:mysql://localhost:3306/springdb?useUnicode=tr
```

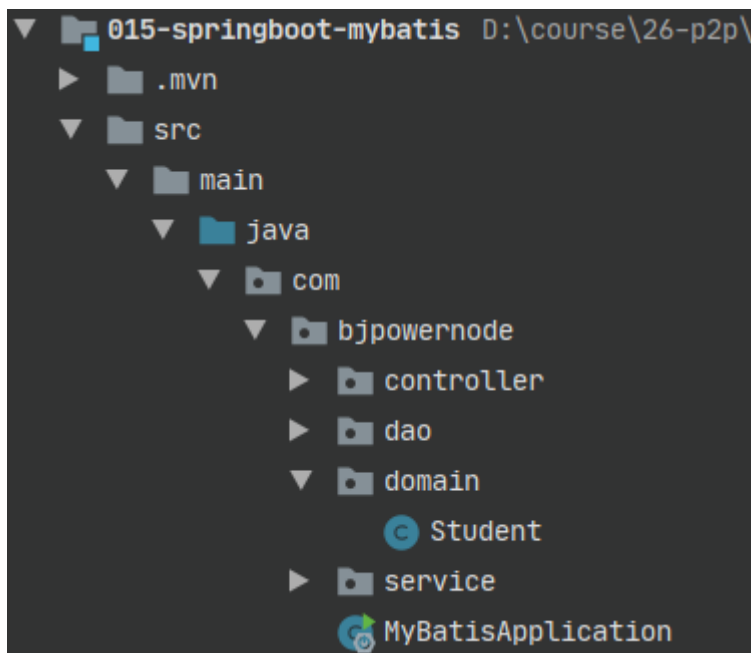
```
ue&characterEncoding=UTF-8&serverTimezone=GMT%2B8
```

```
#serverTimezone=Asia/Shanghai
```

```
spring.datasource.username=root
```

```
spring.datasource.password=123
```

- 创建数实体 bean， dao 接口， mapper 文件



- 实体类

```
public class Student {

    private Integer id;

    private String name;

    private Integer age;

    // set | get
```

```
}
```

➤ 创建 Dao 接口

```
package com.bjpowernode.dao;

import com.bjpowernode.domain.Student;

import org.apache.ibatis.annotations.Mapper;

import org.apache.ibatis.annotations.Param;

/**
 * @Mapper: 找到接口和他的 xml 文件
 *
 * 位置: 在接口的上面
 */
@Mapper
public interface StudentMapper {

    Student selectStudentById(@Param("id") Integer id);

}
```

➤ 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.bjpowernode.dao.StudentMapper">

    <select id="selectStudentById"

        resultType="com.bjpowernode.domain.Student">

        select * from student2 where id=#{id}

    </select>

</mapper>
```

➤ service 接口

```
public interface StudentService {

    Student queryStudent(Integer id);

}
```

➤ service 接口实现类

```
@Service

public class StudentServiceImpl implements StudentService {

    @Resource

    private StudentMapper studentDao;

    @Override

    public Student queryStudent(Integer id) {

        Student student = studentDao.selectStudentById(id);

    }

}
```

```
        return student;
    }
}
```

➤ controller 类

```
@Controller
public class StudentController {

    @Resource

    private StudentService studentService;

    @RequestMapping("/query")

    @ResponseBody

    public String queryStudent(Integer id) {

        Student student = studentService.queryStudent(id);

        return "查询结果 id 是"+id+", 学生="+student.toString();

    }

}
```

启动 Application 类，浏览器访问 <http://localhost:9090/myboot/query>

1.4.2 @MapperScan

在 Dao 接口上面加入@Mapper，需要在每个接口都加入注解。当 Dao 接口多的时候不方便。

可以使用如下的方式解决。

主类上添加注解包扫描: @MapperScan("com.bjpowernode.dao")

新建 Spring Boot 项目 : 016-springboot-mybatis2

项目的代码同上面的程序, 修改的位置:

1. 去掉 StudentMapper 接口的上面的 @Mapper 注解
2. 在主类上面加入 @MapperScan()

```
/**
 * @MapperScan: 扫描所有的 mybatis 的 dao 接口
 *
 * 位置: 在主类的上面
 *
 * 属性: basePackages: 指定 dao 接口的所在的包名。
 *
 * dao 接口和 mapper 文件依然在上一目录
 */

@SpringBootApplication
@MapperScan(basePackages = "com.bjpowernode.dao")
public class MyBatisApplication2 {

    public static void main(String[] args) {

        SpringApplication.run(MyBatisApplication2.class, args);

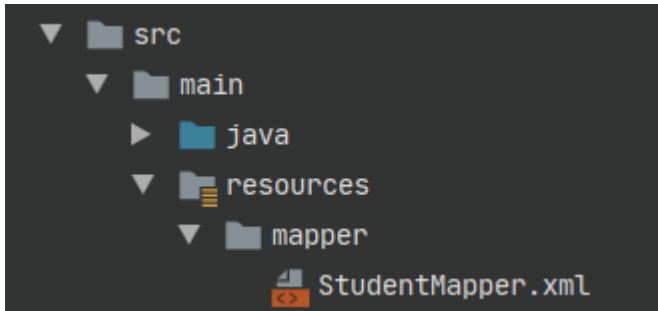
    }

}
```


1.4.3 mapper 文件和 java 代码分开管理

这种方式比较推荐，mapper 文件放在 resources 目录下， java 代码放在 src/main/java。实现步骤：

- 在 resources 创建自定义目录，例如 mapper， 存放 xml 文件



- 把原来的 xml 文件剪切并拷贝到 resources/mapper 目录
- 在 application.properties 配置文件中指定映射文件的位置，这个配置只有接口和映射文件不在同一个包的情况下，才需要指定。

#指定 mybatis 的配置， 相当于 mybatis 主配置文件的作用

mybatis:

mapper-locations: classpath:mapper/*.xml

configuration:

log-impl: org.apache.ibatis.logging.stdout.StdOutImpl

- 运行主类， 浏览器测试访问

1.4.4 事务支持

Spring Boot 使用事务非常简单，底层依然采用的是 Spring 本身提供的事务管理

- 在入口类中使用注解 @EnableTransactionManagement 开启事务支持
- 在访问数据库的 Service 方法上添加注解 @Transactional 即可

通过 SpringBoot +MyBatis 实现对数据库学生表的更新操作，在 service 层的方法中构建异常，查看事务是否生效。

创建项目：018-springboot-transaction

项目可以在 MyBatis 项目中修改。

实现步骤：

1. pom.xml

```
<dependencies>

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-web</artifactId>

    </dependency>

    <dependency>

        <groupId>org.mybatis.spring.boot</groupId>

        <artifactId>mybatis-spring-boot-starter</artifactId>

        <version>2.1.4</version>

    </dependency>

    <dependency>

        <groupId>mysql</groupId>

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

```
<scope>runtime</scope>

</dependency>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-test</artifactId>

    <scope>test</scope>

</dependency>

</dependencies>
```

2. 修改 StudentService，在 addStudent() 方法中抛出异常

```
@Service

public class StudentServiceImpl {

    @Resource

    private StudentMapper studentDao;

    @Transactional

    public int addStudent(Student student) {

        int rows = studentDao.addStudent(student);

        System.out.println("addStudent 添加学生数据!!!");
    }
}
```

```
//在此构造一个除数为0的异常，测试事务是否起作用

    int i = 10/0;

    return rows;

}

}
```

3. 在Application主类上，@EnableTransactionManagement 开启事务支持
@EnableTransactionManagement 可选，但是@Service 必须添加事务才生效

```
@SpringBootApplication
@EnableTransactionManagement
@MapperScan(value="com.bjpowernode.dao")
public class Application {

    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);

    }

}
```

4. 测试应用， 数据没有添加成功
5. 注释掉 StudentServiceImpl 上的@Transactional 测试。数据添加成功

1.5 第五章 接口架构风格—RESTful

1.5.1 认识 REST

REST（英文：Representational State Transfer，简称 REST）

一种互联网软件架构设计的风格，但它并不是标准，它只是提出了一组客户端和服务端交互时的架构理念和设计原则，基于这种理念和原则设计的接口可以更简洁，更有层次，REST 这个词，是 Roy Thomas Fielding 在他 2000 年的博士论文中提出的。

任何的技术都可以实现这种理念，如果一个架构符合 REST 原则，就称它为 RESTful 架构
比如我们要访问一个 http 接口：http://localhost:8080/boot/order?id=1021&status=1
采用 RESTful 风格则 http 地址为：http://localhost:8080/boot/order/1021/1

1.5.2 RESTful 的注解

Spring Boot 开发 RESTful 主要是几个注解实现

（1）@PathVariable

获取 url 中的数据

该注解是实现 RESTful 最主要的一个注解

（2）@PostMapping

接收和处理 Post 方式的请求

（3）@DeleteMapping

接收 delete 方式的请求，可以使用 GetMapping 代替

接收 put 方式的请求，可以用 PostMapping 代替

接收 get 方式的请求

- 轻量，直接基于 http，不再需要任何别的诸如消息协议
 - get/post/put/delete 为 CRUD 操作
- 面向资源，一目了然，具有自解释性。
- 数据描述简单，一般以 xml，json 做数据交换。
- 无状态，在调用一个接口（访问、操作资源）的时候，可以不用考虑上下文，不用考虑当前状态，极大的降低了复杂度。
- 简单、低耦合

1.5.4.1 编写 Controller

```
package com.bjpowernode.controller;

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

```
@RestController
```

```
public class MyRestController {
```

```
    //get 请求
```

```
    /**
```

```
     * rest 中， url 要使用占位符，表示传递的数据。
```

```
     * 占位符叫做路径变量， 在 url 中的数据
```

```
     *
```

```
     * 格式： 在@RequestMapping 的 value 属性值中，使用 {自定义名称}
```

```
     * http://localhost:8080/myboot/student/1001/bj2009
```

```
     *
```

```
     *
```

```
     * @PathVariable: 路径变量注解，作用是获取 url 中的路径变量的值
```

```
     * 属性： value : 路径变量名称
```

```
     * 位置： 在逐个接收参数中，在形参定义的前面
```

```
     *
```

```
     * 注意：路径变量名和形参名一样， value 可以不写
```

```
     *
```

```
* /student/1001/bj2009

*/

@GetMapping(value = "/student/{studentId}/{classname}")

public String queryStudent(@PathVariable(value = "studentId") Integer

id,

        @PathVariable String classname) {

    return "get 请求, 查询学生 studentId: "+id+", 班级: "+classname;

}

///student/1001/bjpowernode

@GetMapping(value = "/student/{studentId}/school/{schoolname}")

public String queryStudentBySchool(@PathVariable(value = "studentId")

Integer id,

        @PathVariable String schoolname) {

    return "get 请求, 查询学生 studentId: "+id+", 班级: "+schoolname;

}
```



```
@PostMapping("/student/{stuId}")

public String createStudent(@PathVariable("stuId") Integer id,

                           String name,

                           Integer age) {

    return "post 创建学生, id="+id+", name="+name+", age="+age;

}


@PostMapping("/student/{stuId}")

public String modifyStudent(@PathVariable("stuId") Integer id,

                           String name) {

    System.out.println("=====put 请求方式 =====");

    return "put 修改学生, id="+id+", 修改的名称是: "+name;

}


@DeleteMapping("/student/{stuId}")

public String removeStudent(@PathVariable("stuId") Integer id) {

    System.out.println("=====delete 请求方式 =====");

    return "delete 删除学生, id="+id;
```

```
}  
  
}
```

application.properties 文件

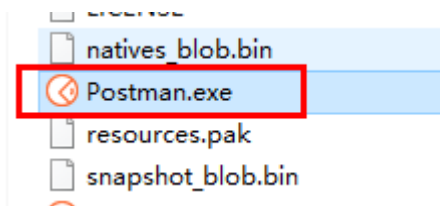
```
server.servlet.context-path=/myboot
```

启用 HiddenHttpMethodFilter 这个过滤器，支持 post 请求转为 put, delete

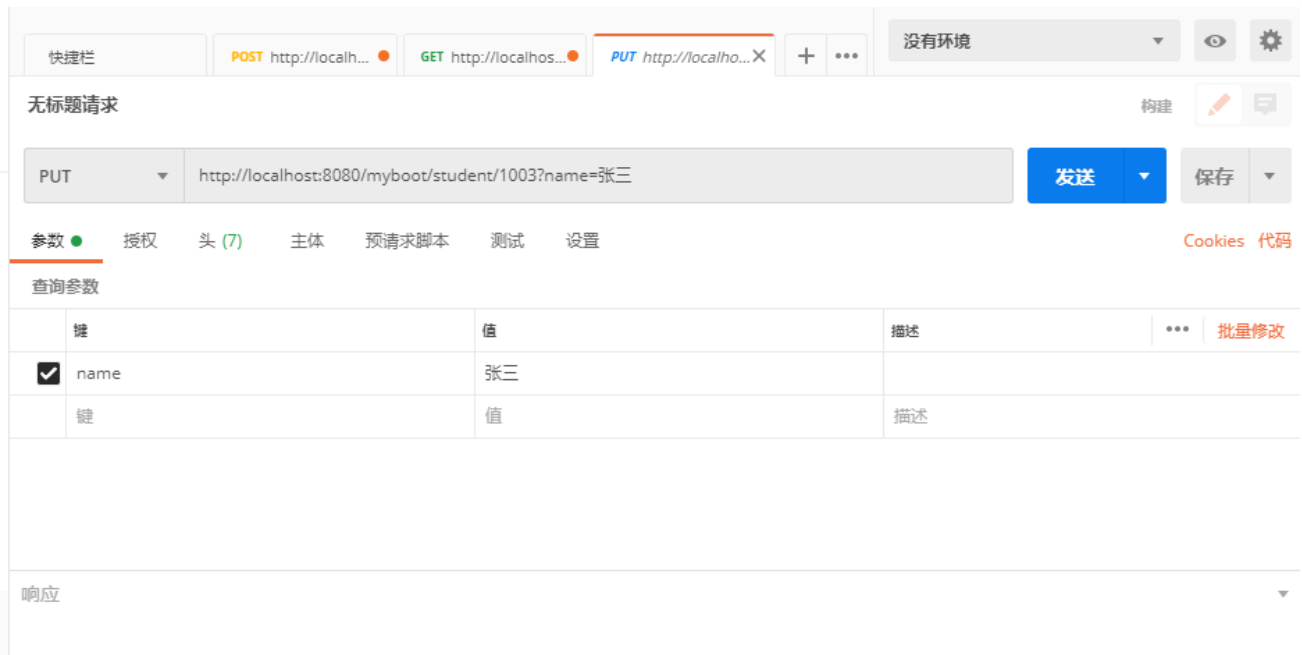
```
spring.mvc.hiddenmethod.filter.enabled=true
```

1.5.4.2 使用 Postman 模拟发送请求，进行测试

安装 Postman 测试软件，安装后执行 Postman.exe



使用方式，设置连接和参数，点击“发送”按钮



1.5.4.3 请求路径冲突

```
@GetMapping(value = "/student/{studentId}/{classname}")
@GetMapping(value = "/student/{studentId}/{schoolname}")
```

这样的路径访问会失败， 路径有冲突。

解决：设计路径，必须唯一， 路径 uri 和 请求方式必须唯一。

1.5.4.4 RESTful 总结

➤ 增 post 请求、删 delete 请求、改 put 请求、查 get 请求

➤ **请求路径不要出现动词**

例如：查询订单接口

/boot/order/1021/1（推荐）

/boot/queryOrder/1021/1（不推荐）

➤ **分页、排序等操作，不需要使用斜杠传参数**

例如：订单列表接口

/boot/orders?page=1&sort=desc

一般传的参数不是数据库表的字段，可以不采用斜杠

1.6 第六章 Spring Boot 集成 Redis

Redis 是一个 NoSQL 数据库，常作用缓存 Cache 使用。通过 Redis 客户端可以使用多种语言在程序中，访问 Redis 数据。java 语言中使用的客户端库有 Jedis, lettuce, Redisson 等。

Spring Boot 中使用 RedisTemplate 模版类操作 Redis 数据。

需求：完善根据学生 id 查询学生的功能，先从 redis 缓存中查找，如果找不到，再从数据库中查找，然后放到 redis 缓存中

1.6.1 项目之前需要安装配置好 Redis。

检查 Linux 中的 redis。 启动 redis，通过客户端访问数据

1.6.2 需求实现步骤

创建项目：020-springboot-redis

1.6.2.1 pom.xml

```
<!--redis 起步依赖

    spring boot 会在容器中创建两个对象 RedisTemplate ,StringRedisTemplate
-->

<dependency>

    <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>
```

1.6.2.2 核心配置文件 application.properties

```
#指定 redis  
  
spring.redis.host=localhost  
  
spring.redis.port=6379  
  
#spring.redis.password=123456
```

1.6.2.3 创建 RedisController

```
package com.bjpowernode.controller;  
  
import com.bjpowernode.vo.Student;  
import org.springframework.data.redis.core.RedisTemplate;  
import org.springframework.data.redis.core.StringRedisTemplate;  
import  
org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;  
import org.springframework.data.redis.serializer.StringRedisSerializer;  
import org.springframework.web.bind.annotation.GetMapping;
```

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

import org.springframework.web.bind.annotation.PostMapping;

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

import javax.annotation.Resource;

@RestController

public class RedisController {

    //注入 RedisTemplate

    //泛型 key, value 都是 String , 或者 Object, 不写

    @Resource

    private RedisTemplate redisTemplate;

    @GetMapping("/myname")

    public String getName() {

        redisTemplate.setKeySerializer( new StringRedisSerializer());

        //    redisTemplate.setValueSerializer( new StringRedisSerializer());

        String name=(String) redisTemplate.opsForValue().get("name");

        return name;

    }

}
```

```
@PostMapping("/name/{myname}")

public String addName(@PathVariable("myname") String name) {

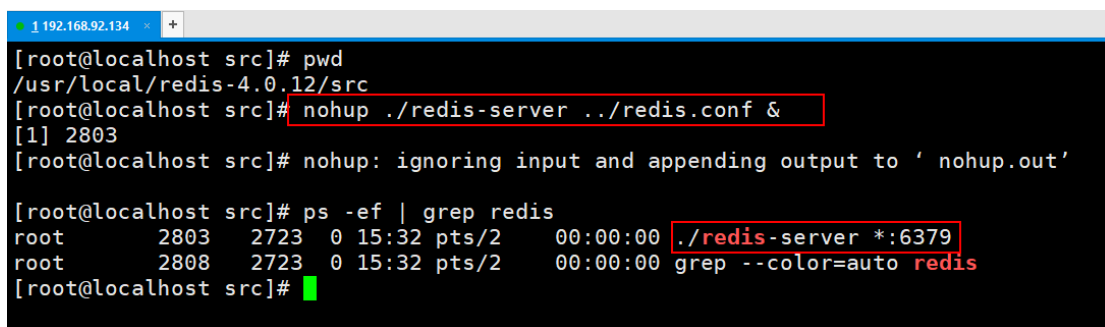
    redisTemplate.opsForValue().set("name", name);

    return "添加了学生:" + name;

}

}
```

1.6.2.4 启动 Redis 服务



```
1 192.168.92.134 +
[root@localhost src]# pwd
/usr/local/redis-4.0.12/src
[root@localhost src]# nohup ./redis-server ../redis.conf &
[1] 2803
[root@localhost src]# nohup: ignoring input and appending output to 'nohup.out'

[root@localhost src]# ps -ef | grep redis
root      2803    2723  0 15:32 pts/2    00:00:00 ./redis-server *:6379
root      2808    2723  0 15:32 pts/2    00:00:00 grep --color=auto redis
[root@localhost src]#
```

1.6.2.5 执行 Spring Boot Application 启动类

```
@SpringBootApplication

public class Application {

    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);

    }

}
```

```
}  
  
}
```

1.6.2.6 使用 Postman 发送请求

无标题请求 构建

POST ▼ http://localhost:8080/name/zhangsan 发送 ▼

参数 授权 头 (7) 主体 预请求脚本 测试 设置

查询参数

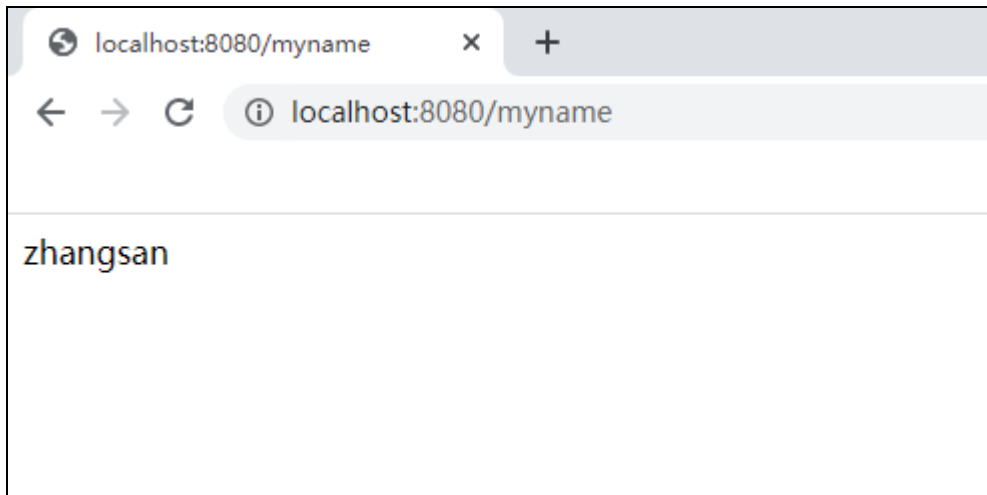
键	值	描述	..
键	值	描述	

主体 Cookies 头 (5) 测试结果 ⊕ 状态: 200 OK 时间: 44 ms 大小: 188 B

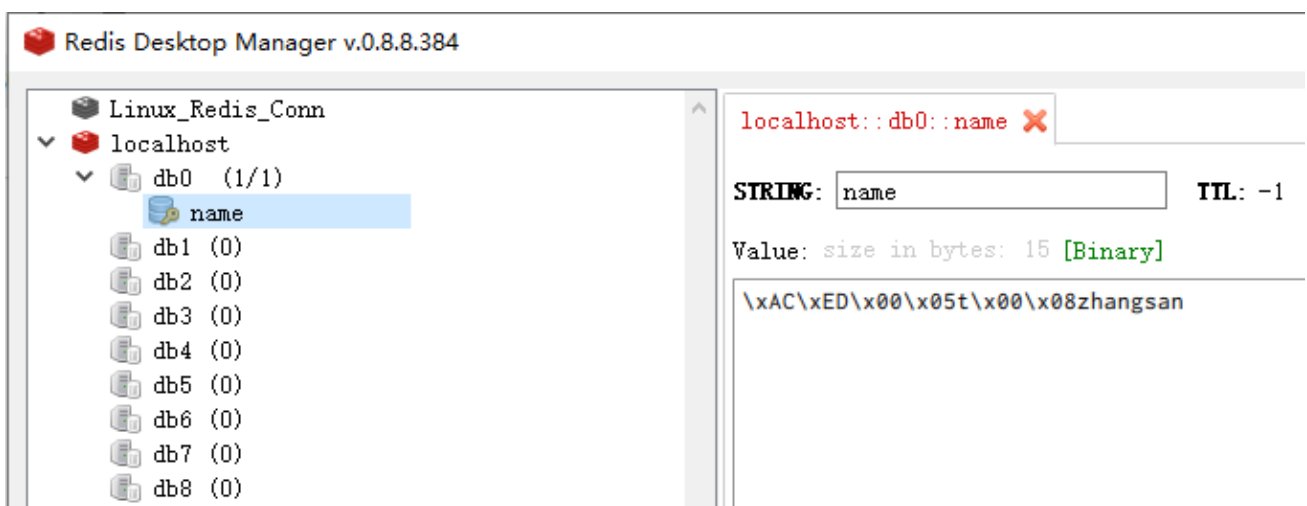
美化 原 预览 可视化 Text ▼ 🔄

1 添加了学生:zhangsan

1.6.2.7 启动浏览器访问 Controller



1.6.2.8 Redis Desktop Manager 桌面工具查看



1.7 第七章 Spring Boot 集成 Dubbo

阿里巴巴提供了 dubbo 集成 springBoot 开源项目，

可以到 GitHub 上 <https://github.com/apache/dubbo-spring-boot-project> 查看入门教程

1.7.1 创建接口 module

按照 Dubbo 官方开发建议，创建一个接口项目，该项目只定义接口和 model 类。

项目名称：021-interface-api

此项目就是一个普通的 maven 项目

gav:

```
<groupId>com.bjpowernode</groupId>

<artifactId>021-interface-api</artifactId>

<version>1.0.0</version>
```

model 类:

```
public class Student implements Serializable {

    private Integer id;

    private String name;

    private Integer age;

    //set|get 方法

}
```

服务接口

```
public interface StudentService {
```

```
Student queryStudent(Integer studentId);  
  
}
```

1.7.2 服务提供者 module

实现 api 项目中的接口

项目名称: 022-service-prvider

使用 Spring Boot Initializer 创建项目, 不用选择依赖

step1) pom.xml 依赖

```
<dependency>  
  
    <groupId>org.apache.dubbo</groupId>  
  
    <artifactId>dubbo-spring-boot-starter</artifactId>  
  
    <version>2.7.8</version>  
  
</dependency>  
  
<dependency>  
  
    <groupId>org.apache.dubbo</groupId>  
  
    <artifactId>dubbo-dependencies-zookeeper</artifactId>  
  
    <version>2.7.8</version>  
  
    <type>pom</type>  
  
</dependency>
```

step2) application.properties

```
#服务名称

spring.application.name=service-provider

#zookeeper 注册中心

dubbo.registry.address=zookeeper://localhost:2181

#dubbo 注解所在的包名

dubbo.scan.base-packages=com.bjpowernode.service
```

step3) 创建接口的实现类

```
import org.apache.dubbo.config.annotation.DubboService;

import org.springframework.stereotype.Component;

@Component

@DubboService(interfaceClass = StudentService.class, version = "1.0")

public class StudentServiceImpl implements StudentService {

    @Override

    public Student queryStudent(Integer studentId) {

        Student student = new Student();

        student.setId(studentId);
```

```
        student.setName("张三");

        student.setAge(20);

        return student;
    }
}
```

step3) 应用主类 Application

```
@SpringBootApplication
@EnableDubbo

public class Application {

    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);

    }

}
```

1.7.3 创建消费者 module

项目名称: 023-consumer

使用 Spring Boot Initializer 创建项目， 选择 web 依赖

step1) pom.xml dubbo 依赖

```
<dependency>
```

```
<groupId>com.bjpowernode</groupId>

<artifactId>021-interface-api</artifactId>

<version>1.0.0</version>
</dependency>

<dependency>

  <groupId>org.apache.dubbo</groupId>

  <artifactId>dubbo-spring-boot-starter</artifactId>

  <version>2.7.8</version>
</dependency>

<dependency>

  <groupId>org.apache.dubbo</groupId>

  <artifactId>dubbo-dependencies-zookeeper</artifactId>

  <version>2.7.8</version>

  <type>pom</type>
</dependency>
```

step2) application.properties

```
#服务名称

spring.application.name=service-consumer

#dubbo 注解所在的包

dubbo.scan.base-packages=com.bjpowernode

#zookeeper

dubbo.registry.address=zookeeper://localhost:2181
```

step3) 创建 MyController

```
@RestController

public class MyController {

    @DubboReference(interfaceClass = StudentService.class, version =
"1.0", check = false)

    private StudentService studentService;

    @GetMapping("/query")

    public String searchStudent(Integer id) {

        Student student = studentService.queryStudent(id);

        return "查询学生: " + student.toString();
    }
}
```

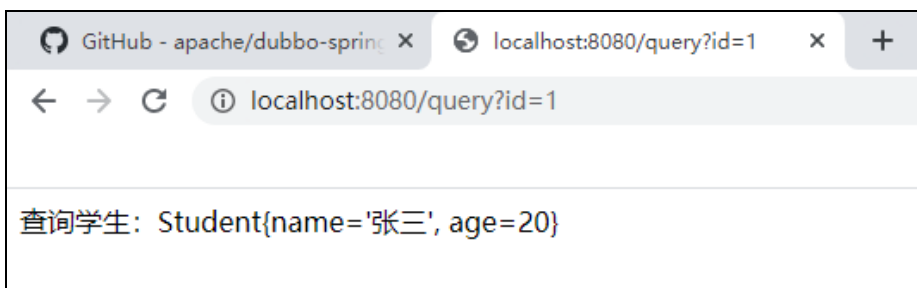
```
}  
  
}
```

step4) 主启动类型 Application

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

1.7.4 测试应用

- 1) 先启动 zookeeper
- 2) 运行服务提供者 022-service-provider
- 3) 运行消费者 023-consumer
- 4) 在浏览器执行 <http://localhost:8080/query?id=1>



1.7.5 SLF4j 依赖

dubbo 提供者或者消费者项目启动是，有日志的提示

```
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...  
SLF4J: Class path contains multiple SLF4J bindings.  
SLF4J: Found binding in [jar:file:/D:/course/26-p2p/p2p-repository/ch/qos/logback/logback-classic  
SLF4J: Found binding in [jar:file:/D:/course/26-p2p/p2p-repository/org/slf4j/slf4j-log4j12/1.7.30  
SLF4J: See http://www.slf4j.org/codes.html#multiple\_bindings for an explanation.  
SLF4J: Actual binding is of type [ch.qos.logback.classic.util.ContextSelectorStaticBinder]  
log4j:WARN No appenders could be found for logger (org.apache.dubbo.common.logger.LoggerFactory).  
log4j:WARN Please initialize the log4j system properly.
```

日志依赖 SLF4j 多次加入了。只需要依赖一次就可以。

解决方式：排除多余的 SLF4j 依赖，提供者和消费者项目都需要这样做

```
<dependency>  
  
  <groupId>org.apache.dubbo</groupId>  
  
  <artifactId>dubbo-dependencies-zookeeper</artifactId>  
  
  <version>2.7.8</version>  
  
  <type>pom</type>  
  
  <exclusions>  
  
    <exclusion>  
  
      <artifactId>slf4j-log4j12</artifactId>  
  
      <groupId>org.slf4j</groupId>  
  
    </exclusion>  
  
  </exclusions>  
</dependency>
```

```
</exclusions>

</dependency>
```

1.8 第八章 Spring Boot 打包

Spring Boot 可以打包为 war 或 jar 文件。 以两种方式发布应用

1.8.1 Spring Boot 打包为 war

创建 Spring Boot web 项目: 024-springboot-war

1.8.1.1 pom.xml

在 pom.xml 文件中配置内嵌 Tomcat 对 jsp 的解析包

```
<!--处理 jsp 的依赖-->

<dependency>

    <groupId>org.apache.tomcat.embed</groupId>

    <artifactId>tomcat-embed-jasper</artifactId>

</dependency>

<!--处理 web 的依赖-->

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-web</artifactId>

</dependency>
```

1.8.1.2 指定 jsp 文件编译目录

```
<resource>

  <directory>src/main/webapp</directory>

  <targetPath>META-INF/resources</targetPath>

  <includes>

    <include>*/*. *</include>

  </includes>

</resource>
```

1.8.1.3 打包后的 war 文件名称

```
<!--指定打包后的 war 文件名称-->

<finalName>myweb</finalName>
```

1.8.1.4 完整的 build 标签内容

```
<build>

  <!--指定打包后的 war 文件名称-->

  <finalName>myweb</finalName>
```

```
<resources>

<!--mybatis 他的 xml 文件放置 src/main/java 目录-->

<resource>

    <directory>src/main/java</directory>

    <includes>

        <include>**/*.xml</include>

    </includes>

</resource>

<!--指定 resources 下面的所有资源-->

<resource>

    <directory>src/main/resources</directory>

    <includes>

        <include>**/*. *</include>

    </includes>

</resource>

<!--指定 jsp 文件编译后目录-->

<resource>

    <directory>src/main/webapp</directory>

    <targetPath>META-INF/resources</targetPath>
```

```
        <includes>

            <include>**/*. *</include>

        </includes>

    </resource>

</resources>

<plugins>

    <plugin>

        <groupId>org.springframework.boot</groupId>

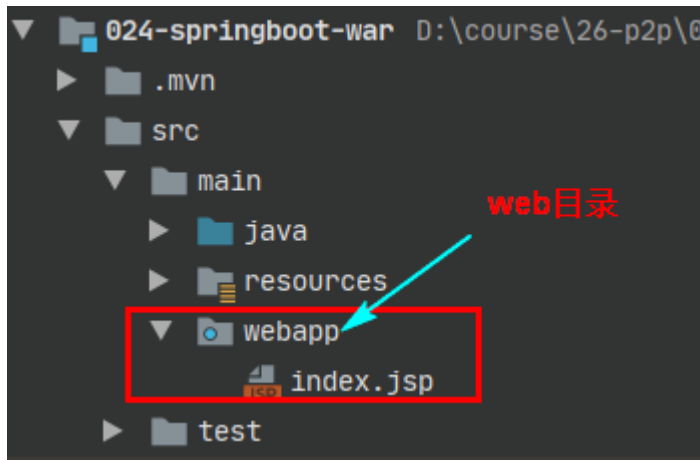
        <artifactId>spring-boot-maven-plugin</artifactId>

    </plugin>

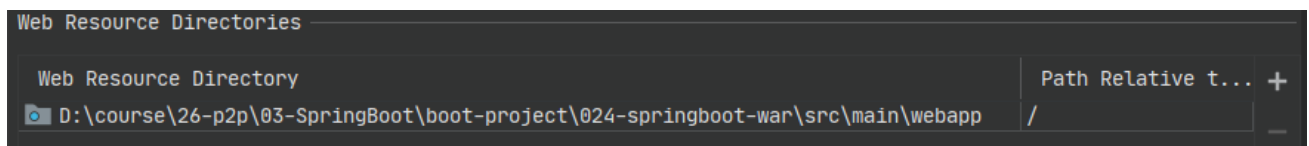
</plugins>

</build>
```

1.8.1.5 创建 webapp 目录



指定 webapp 是 web 应用目录



1.8.1.6 创建 jsp 文件

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>

<html>

<head>

    <title>jsp</title>

</head>

<body>

    显示 controller 中的数据 ${data}
```

```
</body>
```

```
</html>
```

1.8.1.7 创建 JspWarController

```
@Controller

public class JspWarController {

    @RequestMapping("/index")

    public ModelAndView index() {

        ModelAndView mv = new ModelAndView();

        mv.addObject("data", "SpringBoot web 应用打包为 war");

        mv.setViewName("index");

        return mv;

    }

}
```

1.8.1.8 设置视图解析器

```
application.properties
```

```
server.port=9090
```

```
server.servlet.context-path=/myboot

# webapp

spring.mvc.view.prefix=/

# .jsp

spring.mvc.view.suffix=.jsp
```

1.8.1.9 启动主类，在浏览器访问地址 index

访问浏览器，地址 index

1.8.1.10 主启动类继承 `SpringBootServletInitializer`

继承 `SpringBootServletInitializer` 可以使用外部 tomcat。

`SpringBootServletInitializer` 就是原有的 `web.xml` 文件的替代。使用了嵌入式 Servlet，默认是不支持 jsp。

```
@SpringBootApplication

public class JspApplication extends SpringBootServletInitializer {

    public static void main(String[] args) {

        SpringApplication.run(JspApplication.class, args);

    }

    @Override

    protected SpringApplicationBuilder configure(
```

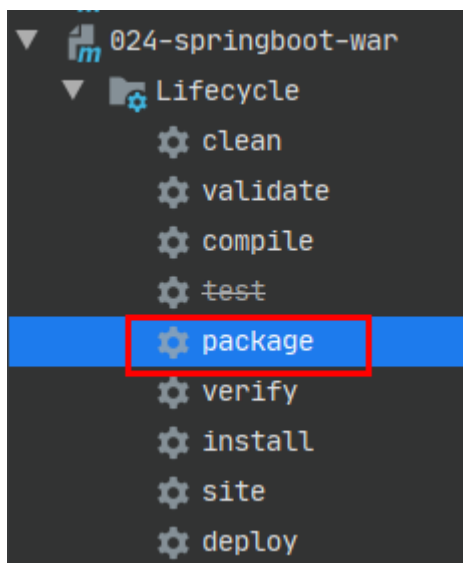


```
        SpringApplication builder) {  
  
        return builder.sources(JspApplication.class);  
  
    }  
  
}
```

1.8.1.11 指定项目 package 是 war

```
<!--指定 packing 为 war-->  
  
<packaging>war</packaging>
```

1.8.1.12 maven package 打包



1.8.1.13 发布打包后的 war 到 tomcat

target 目录下的 war 文件拷贝到 tomcat 服务器 webapps 目录中，启动 tomcat。

在浏览器访问 web 应用

1.8.2 Spring Boot 打包为 jar

创建项目 025-springboot-jar

1.8.2.1 pom.xml

```
<!--处理 jsp 的依赖-->

<dependency>

    <groupId>org.apache.tomcat.embed</groupId>

    <artifactId>tomcat-embed-jasper</artifactId>

</dependency>
```

```
<!--指定 jsp 文件编译后目录-->

<resource>

    <directory>src/main/webapp</directory>

    <targetPath>META-INF/resources</targetPath>

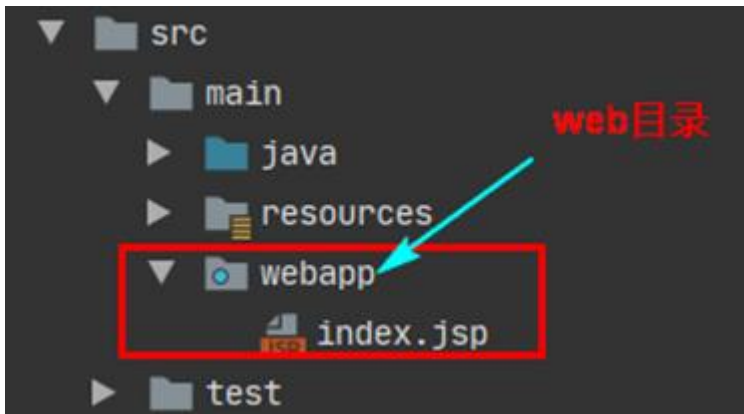
    <includes>

        <include>**/*. *</include>

    </includes>

</resource>
```

1.8.2.2 创建 webpp 目录



并指定他是 web 应用根目录

1.8.2.3 创建 jsp 文件

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>

<html>

<head>

    <title>hello</title>

</head>

<body>

    hello.jsp , 显示数据 ${data}

</body>

</html>
```

1.8.2.4 创建 HelloController

```
@Controller

public class HelloController {

    //ModelAndView

    //Model 存储数据 相当于 HttpServletRequest 的 setAttribute()

    @RequestMapping("/hello")

    public String doHello(Model model) {

        model.addAttribute("data", "SpringBoot 打包为 jar");

        return "hello";

    }

}
```

1.8.2.5 application.properties

```
server.port=9001

server.servlet.context-path=/myboot

spring.mvc.view.prefix=/

spring.mvc.view.suffix=.jsp
```

1.8.2.6 启动 Application，在浏览器访问应用

启动浏览器，访问 hello

1.8.2.7 修改 pom.xml 增加 resources 说明

以后为了保险起见，大家在打包的时候，建议把下面的配置都加上

```
<build>

  <finalName>myboot.jar</finalName>

  <resources>

    <!--mybatis 他的 xml 文件放置 src/main/java 目录-->

    <resource>

      <directory>src/main/java</directory>

      <includes>

        <include>**/*.xml</include>

      </includes>

    </resource>

    <!--指定 resources 下面的所有资源-->

    <resource>

      <directory>src/main/resources</directory>

      <includes>

        <include>**/*.xml</include>
```

```
</includes>

</resource>

<!--指定 jsp 文件编译后目录-->

<resource>

    <directory>src/main/webapp</directory>

    <targetPath>META-INF/resources</targetPath>

    <includes>

        <include>**/*. *</include>

    </includes>

</resource>

</resources>

<plugins>

    <plugin>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-maven-plugin</artifactId>

        <!--带有 jsp 的程序，打包为 jar-->

        <version>1.4.2.RELEASE</version>

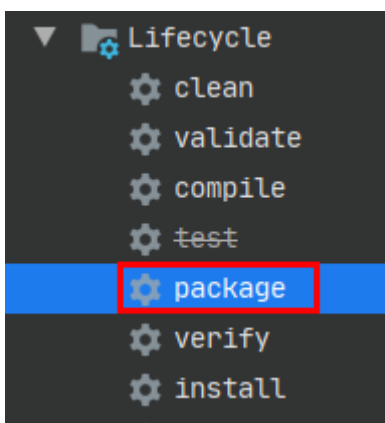
    </plugin>
```

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


1.8.2.8 必须 maven 插件版本

```
<plugins>  
  
  <plugin>  
  
    <groupId>org.springframework.boot</groupId>  
  
    <artifactId>spring-boot-maven-plugin</artifactId>  
  
    <!--带有 jsp 的程序，打包为 jar-->  
  
    <version>1.4.2.RELEASE</version>  
  
  </plugin>  
  
</plugins>
```

1.8.2.9 执行 maven package



target 目录有 jar 文件：mybootjar.jar

 mybootjar.jar

1.8.2.10 执行 jar，启动内置的 tomcat

```
java -jar mybootjar.jar
```

浏览器访问 web 应用

1.8.3 Spring Boot 部署和运行方式总结

- 在 IDEA 中直接运行 Spring Boot 程序的 main 方法（开发阶段）
- 用 maven 将 Spring Boot 安装为一个 jar 包，使用 Java 命令运行

```
java -jar springboot-xxx.jar
```

可以将该命令封装到一个 Linux 的一个 shell 脚本中（上线部署）

- 写一个 shell 脚本：

```
#!/bin/sh
```

```
java -jar xxx.jar
```

- 赋权限 `chmod 777 run.sh`
- 启动 shell 脚本： `./run.sh`


```

-rw-r--r--. 1 root root 20239259 Jun 26 15:06 037-springboot-web-jar-1.0.0.jar
[root@localhost springboot-projects]#
[root@localhost springboot-projects]#
[root@localhost springboot-projects]#
[root@localhost springboot-projects]# vim run.sh
[root@localhost springboot-projects]# ll
total 19772
-rw-r--r--. 1 root root 20239259 Jun 26 15:06 037-springboot-web-jar-1.0.0.jar
-rw-r--r--. 1 root root      53 Jun 26 15:18 run.sh
[root@localhost springboot-projects]# clear
[root@localhost springboot-projects]# ll
total 19772
-rw-r--r--. 1 root root 20239259 Jun 26 15:06 037-springboot-web-jar-1.0.0.jar
-rw-r--r--. 1 root root      53 Jun 26 15:18 run.sh
[root@localhost springboot-projects]#
[root@localhost springboot-projects]#
[root@localhost springboot-projects]# chmod 777 run.sh
[root@localhost springboot-projects]# ll
total 19772
-rw-r--r--. 1 root root 20239259 Jun 26 15:06 037-springboot-web-jar-1.0.0.jar
-rwxrwxrwx. 1 root root      53 Jun 26 15:18 run.sh
[root@localhost springboot-projects]# ./run.sh

```

- 使用 Spring Boot 的 maven 插件将 Springboot 程序打成 war 包，单独部署在 tomcat 中运行（上线部署 常用）

```

      _ _ _ _ _
     /\ / _ ' _ _ _ _ ( ) _ _ _ _ _ \ \ \ \
    ( ( ) \ _ | ' _ | ' _ | | ' _ \ _ | \ \ \ \
     \ / _ _ | | | | | | | | | | ( | | ) ) )
      ' | _ _ | . _ | | | | | | | | | | / / / /
     =====|_|=====|_|/=/////
:: Spring Boot ::                (v2.4.1)

```

1.9 第九章 Thymeleaf 模版

1.9.1 认识 Thymeleaf

Thymeleaf 是一个流行的模板引擎，该模板引擎采用 Java 语言开发

模板引擎是一个技术名词，是跨领域跨平台的概念，在 Java 语言体系下有模板引擎，在 C#、PHP 语言体系下也有模板引擎，甚至在 JavaScript 中也会用到模板引擎技术，Java 生态下的模板引擎有 Thymeleaf、Freemaker、Velocity、Beetl（国产）等。

Thymeleaf 对网络环境不存在严格的要求，既能用于 Web 环境下，也能用于非 Web 环境下。在非 Web 环境下，他能直接显示模板上的静态数据；在 Web 环境下，它能像 Jsp 一样从后台接收数据并替换掉模板上的静态数据。它是基于 HTML 的，以 HTML 标签为载体，Thymeleaf 要寄托在 HTML 标签下实现。

Spring Boot 集成了 Thymeleaf 模板技术，并且 Spring Boot 官方也推荐使用 Thymeleaf 来替代 JSP 技术，Thymeleaf 是另外一种模板技术，它本身并不属于 Spring Boot，Spring Boot 只是很好地集成这种模板技术，作为前端页面的数据展示，在过去的 Java Web 开发中，我们往往会选择使用 Jsp 去完成页面的动态渲染，但是 jsp 需要翻译编译运行，效率低

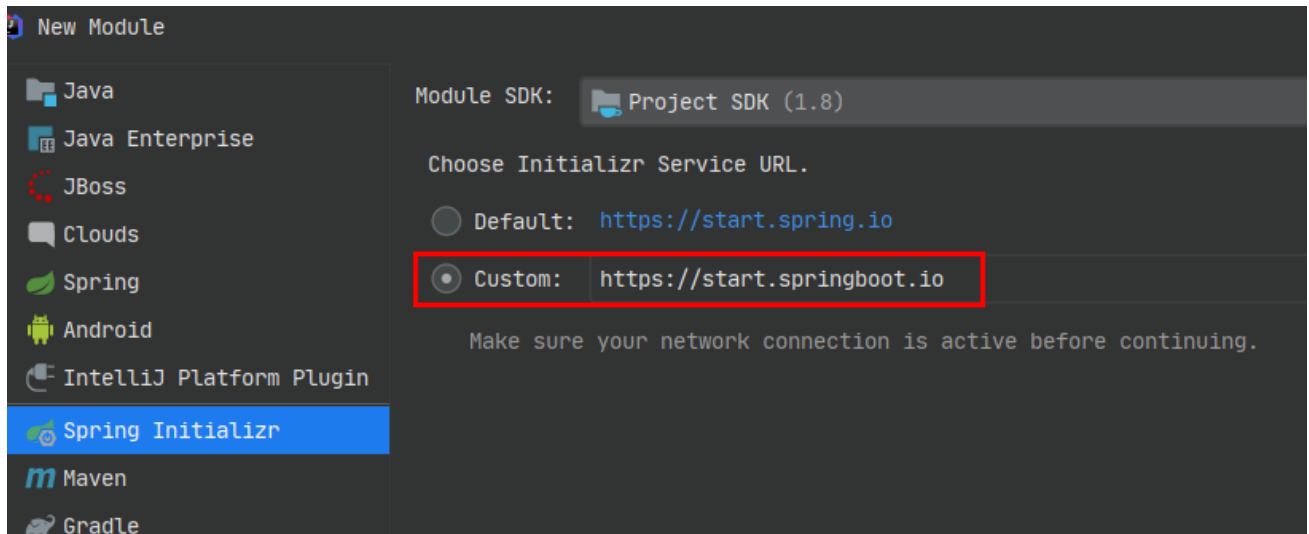
Thymeleaf 的官方网站：<http://www.thymeleaf.org>

Thymeleaf 官方手册：<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>

1.9.2 第一个例子

创建 Spring Boot 项目：026-springboot0-thymeleaf

1.9.2.1 创建 module 步骤



指定 gav

New Module

Project Metadata

Group: com.bjpowernode

Artifact: 026-springboot0-thymeleaf

Type: Maven Project (Generate a Maven based project archive.)

Language: Java

Packaging: Jar

Java Version: 8

Version: 1.0.0

Name: 026-springboot0-thymeleaf

Description: Demo project for Spring Boot

Package: com.bjpowernode

选择依赖:

Dependencies

Spring Boot 2.4.1

Developer Tools

Web

Template Engines

Spring Web

Spring Reactive Web

Rest Repositories

Dependencies

Developer Tools

Web

Template Engines

Security

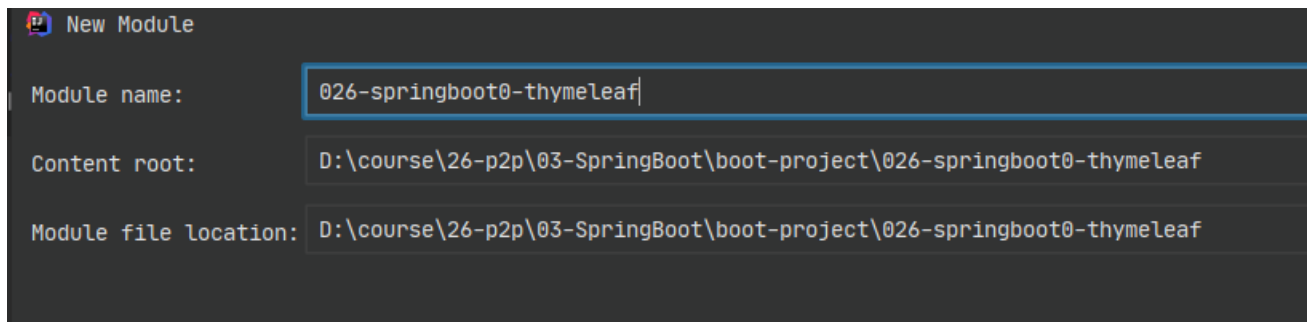
Thymeleaf

Apache Freemarker

Mustache

Groovy Templates

设置项目存储路径



1.9.2.2 pom.xml 主要依赖

```
<!--模版引擎依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

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

1.9.2.3 创建模版文件

在 resources/templates/目录下创建 demo.html

```
<!DOCTYPE html>

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

<head>

    <meta charset="UTF-8">

    <title>第一个例子</title>

</head>

<body>

    <table>

        <tr><td>测试数据</td></tr>

        <tr><td th:text="${name}"></td></tr>

    </table>

</body>

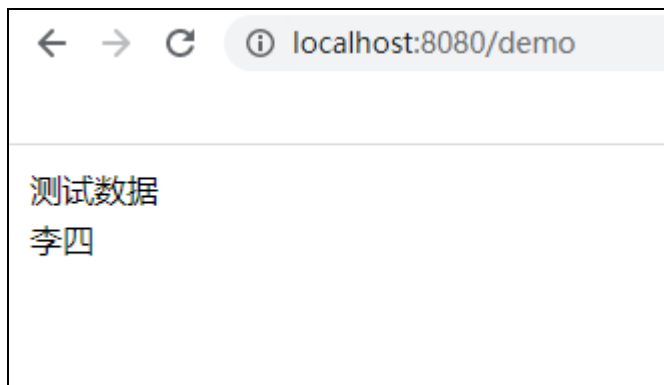
</html>
```

1.9.2.4 创建 ThymeleafController

```
@Controller
```

```
public class ThymeleafController {  
  
    @RequestMapping("/demo")  
  
    public String demo(HttpServletRequest request) {  
  
        request.setAttribute("name", "李四");  
  
        //模版文件名称  
  
        return "demo";  
  
    }  
  
}
```

1.9.2.5 运行 Application 类，在浏览器访问 demo



1.9.2.6 application.properties 设置

```
#配置 Thymeleaf
```

```
#开发阶段设置为 false ， 上线后设置为 true
spring.thymeleaf.cache=false
# 设置模版文件的路径，前缀
spring.thymeleaf.prefix=classpath:/templates/
# 设置后缀
spring.thymeleaf.suffix=.html
# 设置模版的类型
spring.thymeleaf.mode=HTML
```

1.9.3 表达式

表达式是在页面获取数据的一种 thymeleaf 语法。类似 `${key}`

1.9.3.1 标准变量表达式

注意：`th:text=""` 是 Thymeleaf 的一个属性，用于文本的显示

语法: `${key}`

说明：标准变量表达式用于访问容器（tomcat）上下文环境中的变量，功能和 EL 中的 `${}` 相同。Thymeleaf 中的变量表达式使用 `${变量名}` 的方式获取 Controller 中 model 其中的数据。也就是 request 作用域中的数据。

step1) 准备工作，创建 SysUser 类

```
public class SysUser {

    private Integer id;

    private String name;
```



```
private String sex; // m : 男 ; f: 女

private Integer age;

//set | get 方法

}
```

step2) Controller 增加方法

```
//标准表达式

@GetMapping("/thy/expression")

public String expresssion(Model model) {

    //添加简单类型的数据

    model.addAttribute("site", "www.bjpowernode.com");

    //对象类型

    model.addAttribute("myuser", new SysUser(1001, "李思", "f", 20));

    return "01-expression";

}
```

step3) 创建模版文件 01-expression

```
<body>

    <h3>标准表达式${key}</h3>

    <p th:text="${site}">xxx 动力网站</p>

    <p th:text="${myuser.id}">1000</p>
```

```
<p th:text="{myuser.name}">张力</p>

<p th:text="{myuser.sex}">未知</p>

<p th:text="{myuser.getAge()}">0</p>

</body>
```

模版文件（html）修改后，可以使用 idea—Build 菜单-Recompile 编译文件。重新 Recompile 即可生效。

1.9.3.2 1 选择变量表达式

语法：*{key}

说明：需要配和 th:object 一起使用。选择变量表达式，也叫星号变量表达式，使用 th:object 属性来绑定对象，选择表达式首先使用 th:object 来绑定后台传来的对象，然后使用 * 来代表这个对象，后面 {} 中的值是此对象中的属性。

选择变量表达式 *{...} 是另一种类似于标准变量表达式 \${...} 表示变量的方法

选择变量表达式在执行时是在选择的对象上求解，而 \${...} 是在上下文的变量 model 上求解

step1) Controller 增加方法

```
//选择表达式

@GetMapping("/thy/expression2")

public String expresssion2(Model model) {

    //添加简单类型的数据

    model.addAttribute("site", "www.bjpowernode.com");
```

```
//对象类型

model.addAttribute("myuser", new SysUser(1002, "周峰", "m", 30));

return "02-expression2";

}
```

step2) 创建模版文件 02-expression2

```
<div style="margin-left: 350px">

    <p>学习选择表达式</p>

    <div th:object="{myuser}">

        <p th:text="{id}">id</p>

        <p th:text="{name}">name</p>

        <p th:text="{sex}">sex</p>

        <p th:text="{age}">age</p>

    </div>

    <p th:text="{myuser.name}"></p>

</div>
```

1.9.3.3 链接表达式（URL 表达式）

语法：@{链接 url}

说明：主要用于链接、地址的展示，可用于

<script src="...">、<link href="...">、、<form action="...">、等，可以在 URL 路径中动态获取数据

step1) Controller 增加方法

```
//链接表达式

@GetMapping("/thy/link")

public String link(Model model) {

    model.addAttribute("stuId", 1001);

    return "03-link";

}

@GetMapping("/query/student")

@ResponseBody

public String query(Integer id) {

    return "查询学生 id="+id;

}

@GetMapping("/find/school")

@ResponseBody

public String query2(Integer id, String name) {

    return "查询 2, id="+id+", 姓名="+name;

}
```

step2)创建模版文件 03-link

```
<div style="margin-left: 350px">

    <p>链接表达式</p>

    <p>链接到绝对地址</p>

    <a th:href="@{http://www.baidu.com}">百度</a>

    <br/>

    <br/>

    <p>链接到相对地址</p>

    <a th:href="@{/query/student}">相对地址没有传参数</a>

    <br/>

    <br/>

    <p>链接到相对地址, 传参数方式 1</p>

    <a th:href="@{' /query/student?id=' + ${stuId} }">相对地址传参数方式
1</a>

    <br/>

    <br/>

    <p>链接到相对地址, 传参数方式 2, 推荐方式</p>

    <a th:href="@{/find/school(id=${stuId},name='lisi')}">相对地址传参数
```

```
方式 2</a>
```

```
</div>
```

1.9.4 Thymeleaf 属性

大部分属性和 html 的一样，只不过前面加了一个 th 前缀。加了 th 前缀的属性，是经过模版引擎处理的。

step1)Controller 增加方法

```
//模版属性

@GetMapping("/thy/property")

public String htmlProperty(Model model) {

    model.addAttribute("myname", "丽思");

    return "04-htmlproperty";

}
```

step2)创建模版文件 04-hhtmlproperty

```
<!DOCTYPE html>

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

<head>

    <meta charset="UTF-8">

    <title>html 属性</title>
```

```
<script type="text/javascript"
th:src="@{/js/jquery-3.4.1.js}"></script>

<script type="text/javascript">

    function clickFun() {

        alert("button click");

    }

</script>

</head>

<body>

    <div th:style="'margin-left: 350px'">

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

            <input type="text" th:id="username" th:name="username"
th:value="$ {myname}" />

            <input type="button" th:id="btn" th:onclick="clickFun()"
th:attr="value=' 按钮 click' " />

        </form>

        <br/>

        <br/>

        <p th:attr="name=$ {myname}">我是使用了自定义属性名称</p>
```

```
</div>

</body>

</html>
```

1.9.4.1 th:action

定义后台控制器的路径，类似<form>标签的 action 属性，主要结合 URL 表达式,获取动态变量

```
<form id="login" th:action="@{/login}" th:method="post">.....</form>
```

1.9.4.2 th:method

设置请求方法

```
<form id="login" th:action="@{/login}" th:method="post">.....</form>
```

1.9.4.3 th:href

定义超链接，主要结合 URL 表达式,获取动态变量

```
<a th:href="@{/query/student}">相对地址没有传参数</a>
```

1.9.4.4 th:src

用于外部资源引入，比如<script>标签的 src 属性，标签的 src 属性，常与@{}表达式结合使用，在 **SpringBoot 项目的静态资源都放到 resources 的 static 目录下，放到 static 路径下的内容，写路径时不需要写上 static**

```
<script type="text/javascript" th:src="@{/js/jquery-3.4.1.js}"></script>
```

1.9.4.5 th:text

用于文本的显示，该属性显示的文本在标签体中，如果是文本框，数据会在文本框外显示，要想显示在文本框内，使用 th:value

```
<input type="text" id="realName" name="reaName" th:text="${realName}">
```


1.9.4.6 th:style

设置样式

```
<a th:onclick="fun1('${user.id}+')" th:style="color:red">点击我</a>
```

1.9.4.7 th:each

这个属性非常常用，比如从后台传来一个对象集合那么就可以使用此属性遍历输出，它与JSTL 中的<c:forEach>类似，此属性既可以循环遍历集合，也可以循环遍历数组及 Map。

1.9.4.7.1 循环 List

Controller 增加方法

```
//循环 list

@GetMapping("/thy/eachlist")

public String eachList(Model model) {

    List<String> strList = new ArrayList<>();

    strList.add("三国");

    strList.add("三国志");

    strList.add("水浒");

    model.addAttribute("strlist", strList);

    //List<SysUser>

    List<SysUser> userList = new ArrayList<>();
```

```
userList.add(new SysUser(1001, "张飞", "m", 20));  
userList.add(new SysUser(1002, "刘备", "m", 29));  
userList.add(new SysUser(1003, "关羽", "m", 28));  
model.addAttribute("userList", userList);  
  
return "05-eachlist";  
}
```

创建模版文件 05-eachlist

```
<div th:style="'margin-left: 350px'">  
  
    <p>在一个 div 中循环 p 标签</p>  
  
    <div >  
  
        <p th:each="str, strSt:${strlist}" th:text="${str}"></p>  
  
    </div>  
  
    <br/>  
  
    <br/>  
  
    <div th:each="u:${userList}">  
  
        <p th:text="${u.id}"></p>  
  
        <p th:text="${u.name}"></p>  
  
        <p th:text="${u.sex}"></p>
```

```
<p th:text="${u.age}"></p>

</div>

</div>
```

语法说明:

th:each="user, iterStat : \${userlist}"中的 **\${userList}** 是后台传过来的集合

- **user**

定义变量，去接收遍历`${userList}`集合中的一个数据

- **iterStat**

`${userList}` 循环体的信息

- 其中 **user** 及 **iterStat** 自己可以随便取名

- **iterStat** 是循环体的信息，通过该变量可以获取如下信息

index: 当前迭代对象的 **index** (从 0 开始计算)

count: 当前迭代对象的个数 (从 1 开始计算) 这两个用的较多

size: 被迭代对象的大小

current: 当前迭代变量

even/odd: 布尔值，当前循环是否是偶数/奇数 (从 0 开始计算)

first: 布尔值，当前循环是否是第一个

last: 布尔值，当前循环是否是最后一个

注意: 循环体信息 **iterStat** 也可以不定义，则默认采用迭代变量加上 **Stat** 后缀，即 **userStat**

1.9.4.7.2 遍历数组 Array

th:each 语法同上

Controller 增加方法

```
//循环 Array
```

```
@GetMapping("/thy/eacharray")

public String eachArray(Model model) {

    SysUser[] users = new SysUser[3];

    users[0]= new SysUser(1001,"马超","m",22);

    users[1]= new SysUser(1002,"黄忠","m",26);

    users[2]= new SysUser(1003,"赵云","m",29);

    model.addAttribute("userarray",users);

    return "06-eacharray";

}
```

创建模版文件 06-eacharray

```
<div th:style="'margin-left: 350px'">

    <p>在一个 div 中循环 p 标签</p>

    <table border="1" cellpadding="0" cellspacing="0">

        <tr>

            <td>序号</td>

            <td>id</td>

            <td>姓名</td>

            <td>性别</td>

            <td>年龄</td>
```

```
        <td>年龄</td>

    </tr>

    <tr th:each="u:${userarray}" >

        <td th:text="${uStat.count}+'/' +${uStat.size}"></td>

        <td th:text="${u.id}"></td>

        <td th:text="${u.name}"></td>

        <td th:text="${u.sex}"></td>

        <td th:text="${u.age}"></td>

        <td th:text="${uStat.current.age}"></td>

    </tr>

</table>

</div>
```

1.9.4.7.3 遍历 map

Controller 增加方法

```
//循环 Map

@GetMapping("/thy/eachmap")

public String eachMap(Model model) {

    Map<String, SysUser> map = new HashMap<>();
```

```
map.put("user1", new SysUser(1001, "马超", "m", 22));  
map.put("user2", new SysUser(1002, "黄忠", "m", 26));  
map.put("user3", new SysUser(1003, "赵云", "m", 29));  
model.addAttribute("users", map);  
  
//List<Map<SysUser>  
  
List<Map<String, SysUser>> listmap = new ArrayList<>();  
listmap.add(map);  
  
map = new HashMap<>();  
map.put("sys1", new SysUser(2001, "曹操", "m", 22));  
map.put("sys2", new SysUser(2002, "孙权", "m", 26));  
map.put("sys3", new SysUser(2003, "刘备", "m", 29));  
listmap.add(map);  
model.addAttribute("listmap", listmap);  
return "07-eachmap";  
}
```

创建模版文件 07-eachmap

```
<div th:style="' margin-left: 350px' ">
```

```
<p>在一个 div 中循环 p 标签</p>

<div th:each="m, mSt:${users}">

    <p th:text="${mSt.count}"></p>

    <p th:text="${m.key}"> </p>

    <p th:text="${m.value.id}"></p>

    <p th:text="${m.value.name}"></p>

    <br/>

</div>


<br/>

<br/>

<p>list-map</p>

<div th:each="ul:${listmap}">

    <div th:each="um:${ul}">

        <p th:text="${um.key}"></p>

        <p th:text="${um.value.id}"></p>

        <p th:text="${um.value.name}">姓名是</p>

    </div>
```

```
</div>

</div>
```

1.9.4.8 条件判断 if

语法: `th:if="boolean 条件"` , 条件为 `true` 显示体内容

`th:unless` 是 `th:if` 的一个相反操作

Controller 增加方法

```
@GetMapping("/thy/ifunless")

public String ifUnless(Model model) {

    model.addAttribute("sex", "m");

    model.addAttribute("isLogin", true);

    model.addAttribute("age", 20);

    model.addAttribute("name", "");

    return "08-ifunless";

}
```

创建模版文件 08-ifunless

```
<p th:if="${sex == 'm'}">

    性别是 男

</p>
```



```
<p th:unless="{sex == 'f'}">
```

性别是女

```
</p>
```

```
<p th:if="{isLogin}">
```

用户已经登录

```
</p>
```

```
<p th:if="{age > 50}">
```

年龄是大于 50

```
</p>
```

```
<p th:if="{5>0}">
```

5>0

```
</p>
```

```
<!-- 空字符串是 true-->
```

```
<p th:if="{name}">
```

name 是 ‘’

```
</p>
```

1.9.4.9 switch, case 判断语句

语法：类似 java 中的 switch, case

```
<div th:switch="${sex}">

    <p th:case="m">显示男</p>

    <p th:case="f">显示女</p>

    <p th:case="*">未知</p>

</div>
```

一旦某个 case 判断值为 true，剩余的 case 则都当做 false，“*”表示默认的 case，前面的 case 都不匹配时候，执行默认的 case

1.9.4.10 th:inline

th:inline 有三个取值类型 (text, javascript 和 none)

1.9.4.10.1 内联 text

可以让 Thymeleaf 表达式不依赖于 html 标签，直接使用内联表达式[[表达式]]即可获取动态数据，要求在父级标签上加 **th:inline = “text”**属性

Controller 增加方法

```
//内联

@GetMapping("/thy/inline")

public String inline(Model model) {

    model.addAttribute("sex", "m");
```

```
model.addAttribute("isLogin", true);

model.addAttribute("age", 20);

model.addAttribute("name", "克里斯");

model.addAttribute("myuser", new SysUser(1005, "周向", "f", 23));

return "09-inline";
}
```

创建模版文件 09-inline

```
<div th:inline="text">

    姓名是: [[${name}]] <br/>

    登录了吗: [[${isLogin}]]

</div>

<br/>

<!--不用加入 th:inline='text'-->

<div>

    姓名是: [[${name}]] <br/>

    登录了吗: [[${isLogin}]]<br/>

    性别: [[${sex}]]

</div>
```

1.9.4.10.2 内联 javascript

可以在 js 中，获取模版中的数据。

在上面的模版页面中，增加

```
<button onclick="fun()">单击按钮</button>

<script type="text/javascript" th:inline="javascript">

    var name = [[${myuser.name}]];

    var id = [[${myuser.id}]];

    function fun() {

        alert("click 用户是"+name+", 他的 id 是"+id);

    }

</script>
```

1.9.5 字面量

Controller 增加方法

```
//字面量

@GetMapping("/thy/text")

public String text(Model model) {

    model.addAttribute("sex", "m");

    model.addAttribute("isLogin", true);

    model.addAttribute("age", 20);

}
```

```
model.addAttribute("name", null);

model.addAttribute("city", "北京");


model.addAttribute("myuser", new SysUser(1005, "周向", "f", 23));

return "10-text";

}
```

1.9.5.1.1 文本字面量

用单引号'...'包围的字符串为文本字面量

```
<p th:text="' 城市是' + ${city} + '    用户登录' + ${isLogin}"></p>
```

1.9.5.1.2 数字字面量

```
<p th:if="${age > 10}"> age > 10 </p>
```

```
<p th:if="20 > 5">20 大于 5</p>
```

1.9.5.1.3 boolean 字面量

```
<p th:if="${isLogin == true}">用户登录了</p>
```

1.9.5.1.4 null 字面量

```
<p th:if="{name == null}"> name 是 null </p>
```

1.9.6 字符串连接

<p>字符串的连接，1 使用'';2 使用|字符串内容|</p>

```
<p th:text="{城市是'+{city}+'      用户登录'+{isLogin}"}"></p>
```

```
<p th:text="{城市是${city}用户登录${isLogin} |"}"></p>
```

1.9.7 运算符

算术运算: + , - , * , / , %

关系比较: > , < , >= , <= (gt , lt , ge , le)

相等判断: == , != (eq , ne)

```
<p th:if="{age > 10}"> age > 10 </p>
```

```
<p th:if="{20 > 5}">20 大于 5</p>
```

```
<p th:text="{sex == 'm' ? '男' : '女'}"></p>
```

```
<p th:text="{sex == 'm' ? (isLogin?'男已经登录':'男的没有登录'):'女'}"></p>
```

1.9.8 Thymeleaf 基本对象

模板引擎提供了一组内置的对象，这些内置的对象可以直接在模板中使用，这些对象由#号开始引用，我们比较常用的内置对象

1.9.8.1 #request 表示 HttpServletRequest

1.9.8.2 #session 表示 HttpSession 对象

1.9.8.3 session 对象，表示 HttpSession 对象

//模版中基本对象

```
@GetMapping("/thy/baseObject")
public String baseObject(Model model, HttpServletRequest request,
HttpSession session ){
    request.setAttribute("reqdata","request 中的数据");
    request.getSession().setAttribute("sessdata","session 中数据");
    session.setAttribute("loginname","zhangsan");
    return "11-baseObject";
}
```

创建模版文件 11-baseObject

```
<div th:style="'margin-left: 350px'">
```

```
<p th:text="${#request.getAttribute('reqdata')}">request 作用域</p>

<p th:text="${#request.getServerName()}"></p>

<p th:text="${#request.getServerPort()}"></p>

<p th:text="${#request.getServletPath()}"></p>

<br/>

<p th:text="${#session.getAttribute('sessdata')}">session 中数据</p>

<p>处理#request, #session, session</p>

<p th:text="${#request.getServerName()}"></p>

<p th:text="${#request.getServerPort()}"></p>

<p th:text="${#request.getRequestURL()}"></p>

<p th:text="${#request.getRequestURI()}"></p>

<p th:text="${#request.getQueryString()}"></p>

<br/>

<br/>

<p th:text="${#session.getAttribute('loginname')}"></p>

<p th:text="${session.loginname}"></p>
```



```
</div>
```

1.9.9 Thymeleaf 内置工具类对象

1.9.9.1 内置工具类对象

模板引擎提供的一组功能性内置对象，可以在模板中直接使用这些对象提供的功能方法
工作中常使用的数据类型，如集合，时间，数值，可以使用 Thymeleaf 的提供的功能性对象来处理它们

内置功能对象前都需要加#号，内置对象一般都以 s 结尾

官方手册：<http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>

#dates: java.util.Date 对象的实用方法，``

#calendars: 和 dates 类似，但是 java.util.Calendar 对象；

#numbers: 格式化数字对象的实用方法；

#strings: 字符串对象的实用方法： contains, startsWith, prepending/appending 等；

#objects: 对 objects 操作的实用方法；

#booleans: 对布尔值求值的实用方法；

#arrays: 数组的实用方法；

#lists: list 的实用方法，比如``

#sets: set 的实用方法；

#maps: map 的实用方法；

#aggregates: 对数组或集合创建聚合的实用方法

1.9.9.2 例子

准备对象

```
package com.bjpowernode.vo;

public class Cat {

    private String name;

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

}
```

```
public class Dog {

    private String name;

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

}
```

```
public class Zoo {  
  
    private Dog dog;  
  
    private Cat cat;  
  
    public Dog getDog() {  
  
        return dog;  
  
    }  
  
    public void setDog(Dog dog) {  
  
        this.dog = dog;  
  
    }  
  
    public Cat getCat() {  
  
        return cat;  
  
    }  
  
    public void setCat(Cat cat) {  
  
        this.cat = cat;  
  
    }  
  
}
```

Controller 增加方法

```
//内置工具类对象
```

```
@GetMapping("/thy/utilObject")

public String utilObject(Model model, HttpSession session) {

    model.addAttribute("mydate", new Date());

    model.addAttribute("mynum", 26.695);

    model.addAttribute("mystr", "bjpowernode");


    List<String> mylist = Arrays.asList("a", "b", "c");

    model.addAttribute("mylist", mylist);

    //model.addAttribute("mylist", null);


    session.setAttribute("loginname", "zhangsan");


    Dog dog = new Dog();

    dog.setName("二哈");


    Cat cat = new Cat();

    cat.setName("英短");


    Zoo zoo = new Zoo();
```

```
zoo.setCat(cat);

//zoo.setDog(dog);

zoo.setDog(null);

//

model.addAttribute("zoo", zoo);

return "12-utilObject";
}
```

创建模版文件 12-utilObject

```
<p th:text="${#dates.format(mydate, 'yyyy-MM-dd')}"></p>

<p th:text="${#dates.format(mydate, 'yyyy-MM-dd HH:mm:ss')}"></p>

<p th:text="${#dates.year(mydate)}"></p>

<p th:text="${#dates.month(mydate)}"></p>

<p th:text="${#dates.createNow()}" />

<br/>

<br/>

<p th:text="${#numbers.formatCurrency(mynum)}"></p>

<p th:text="${#numbers.formatDecimal(mynum, 5, 2)}"></p>
```

```
<br/>

<br/>

<p>处理@#strings</p>

<p th:text="${#strings.toUpperCase(mystr)}"></p>

<p th:text="${#strings.indexOf(mystr,' power')}"></p>

<p th:text="${#strings.substring(mystr,2,5)}"></p>

<p th:text="${#strings.concat(mystr,' ----java 开发')}"></p>

<br/>

<br/>

<p>处理#lists</p>

<p th:text="${#lists.isEmpty(mylist)}"></p>

<p th:if="!${#lists.isEmpty(mylist)}"></p>

<p th:text="${#lists.size(mylist)}"></p>

<p>空值</p>

<p th:text="${zoo?.dog?.name}"></p>
```

1.9.10 内容复用

自定义模板是复用的行为。可以把一些内容，多次重复使用。

1.9.10.1 定义模板

语法: `th:fragment="top"` , 定义摸模板, 自定义名称是 `top`

例如:

```
<div th:fragment="top">
    <p>动力节点</p>
    <p>www.bjpowernode</p>
</div>
```

1.9.10.2 引用模板

语法: 引用模板 `~{ templatename :: selector }` 或者 `templatename :: selector`

例如:

```
<div th:insert="~{head :: top}"></div>
<div th:include="head :: top"></div>
```

1.9.10.3 模板例子

templates 目录下创建 head.html

```
<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
</head>
<body>
    <!--定义模板, 就是一段页面代码-->
    <div th:fragment="top">
        <p>动力节点</p>
```

```
<p>www.bjpowernode.com</p>
</div>
</body>
</html>
```

创建 footer.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
  <div>
    footer
    www.bjpowernode.com
    @copy; 动力节点 2020
  </div>
</body>
</html>
```

在其他文件中，使用模板内容

```
<p>使用模板</p>
<div th:insert=~{head :: top}>
  我是 div， insert 模板 top
</div>

<hr/>
<p>insert 模板 2</p>
<p th:insert="head :: top">
  insert
</p>
```



```
<hr/>
<br/>

<p>包含</p>

<div th:include="head :: top">
    我是当前的 div
</div>

<hr/>
<div th:include="footer :: html"></div>

<hr/>
<div th:include="footer" />
```