

快速开发框架 Spring Boot

课程讲义

主讲: Reythor 雷

2019

快速开发框架 Spring Boot

第1章 Spring Boot 基础

1.1 Spring Boot 简介

Spring Boot 是由 Pivotal^[ˈpɪvətl]团队（一家做大数据的公司）提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot 致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。

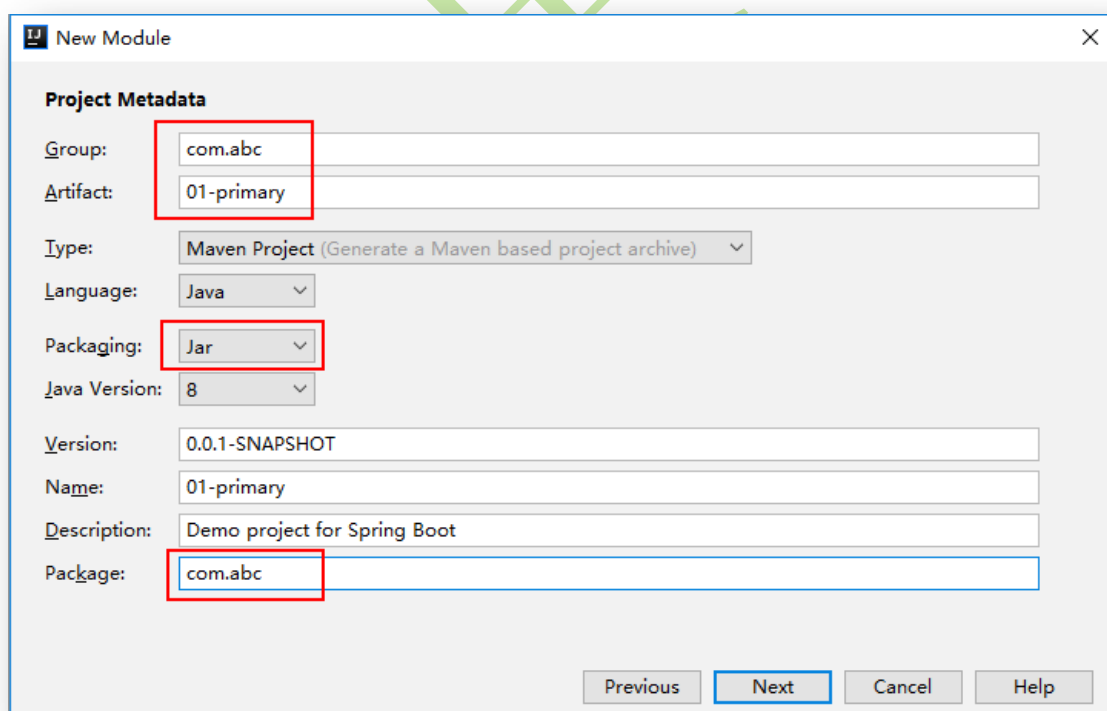
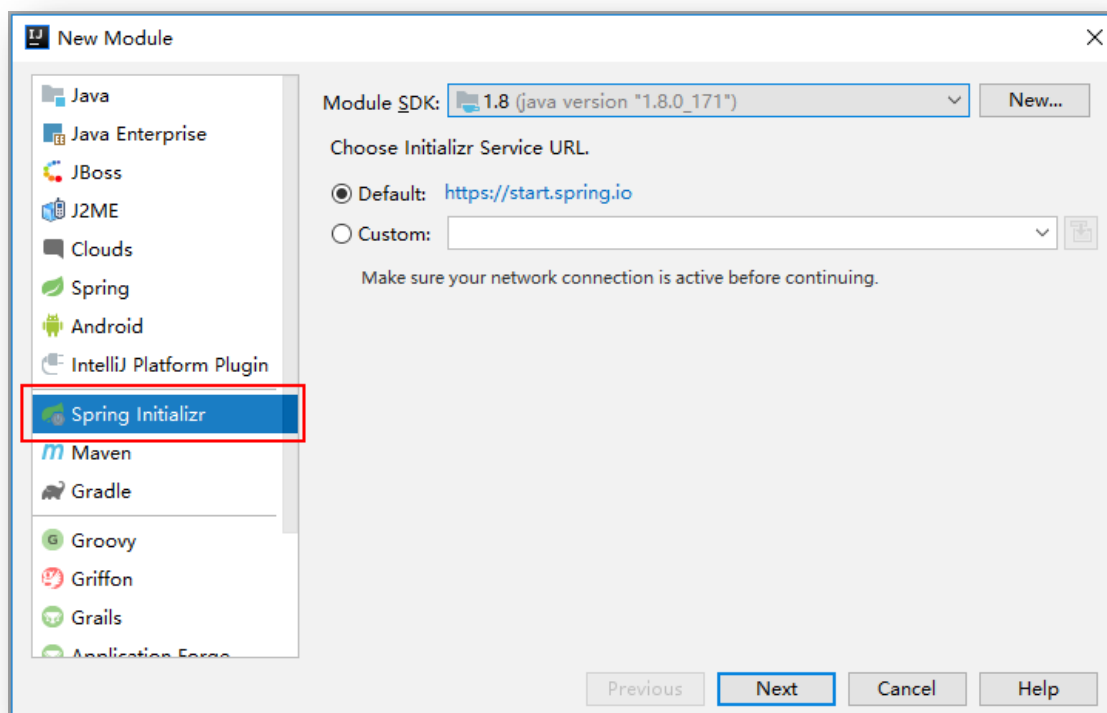
简单来说，SpringBoot 可以简化 Spring 应用程序的开发，使我们不再需要 Spring 配置文件及 web.xml 文件。

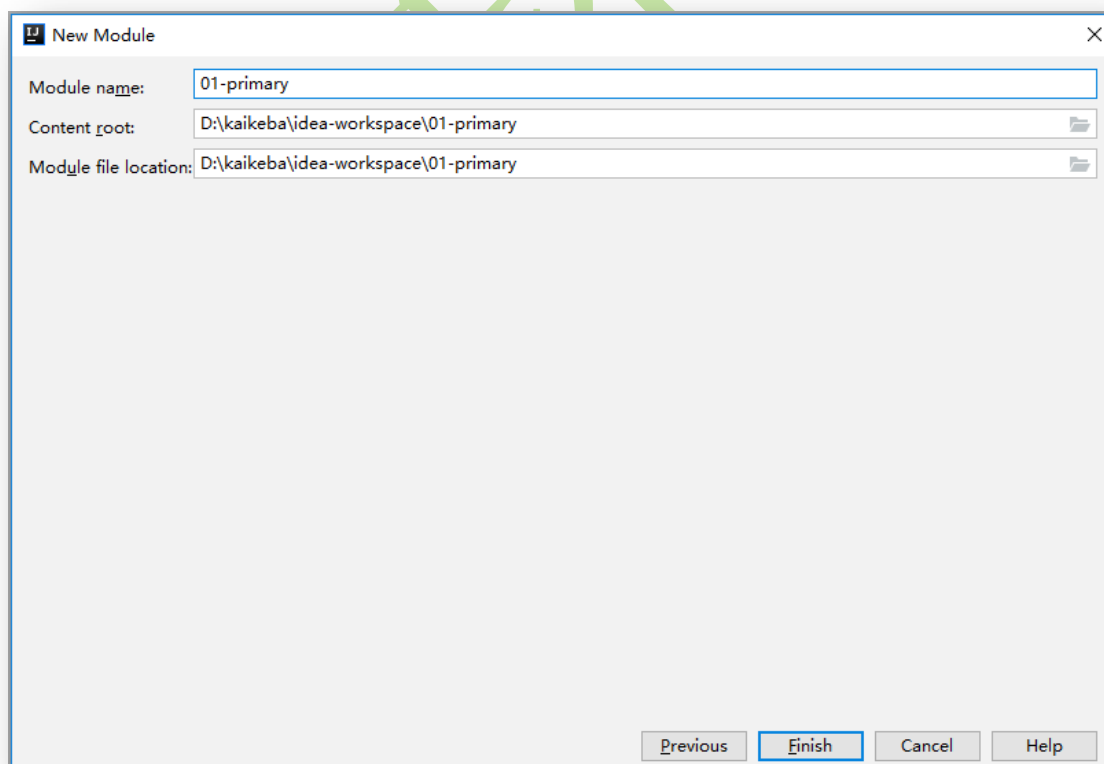
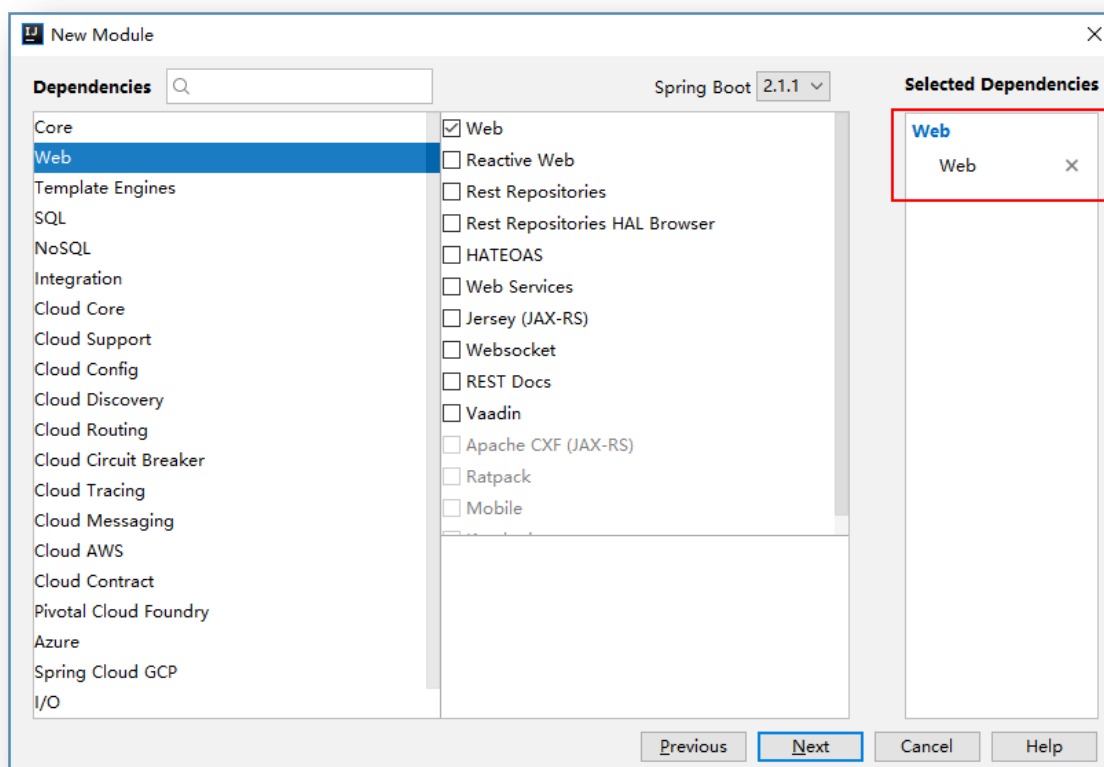
1.2 Spring Boot 工程的创建

1.2.1 在 Idea 中创建

（1）工程创建

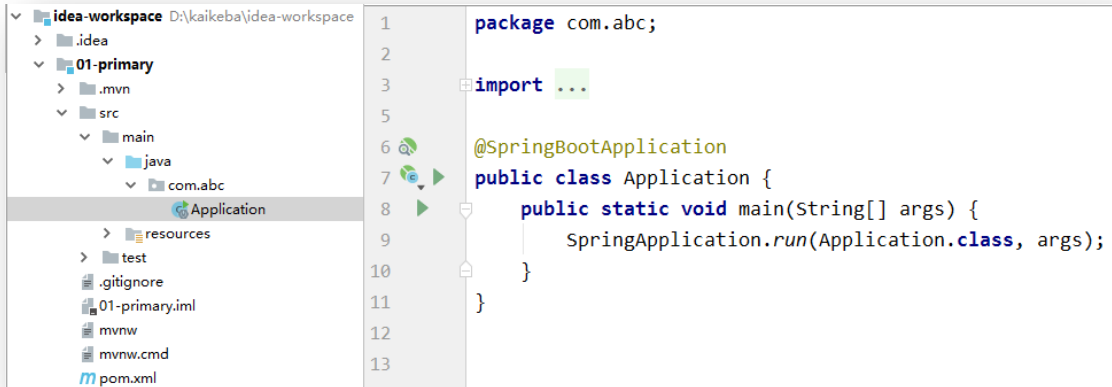
这里要创建一个 Spring Boot 的 Web 工程。首先新建 Spring Initializr。



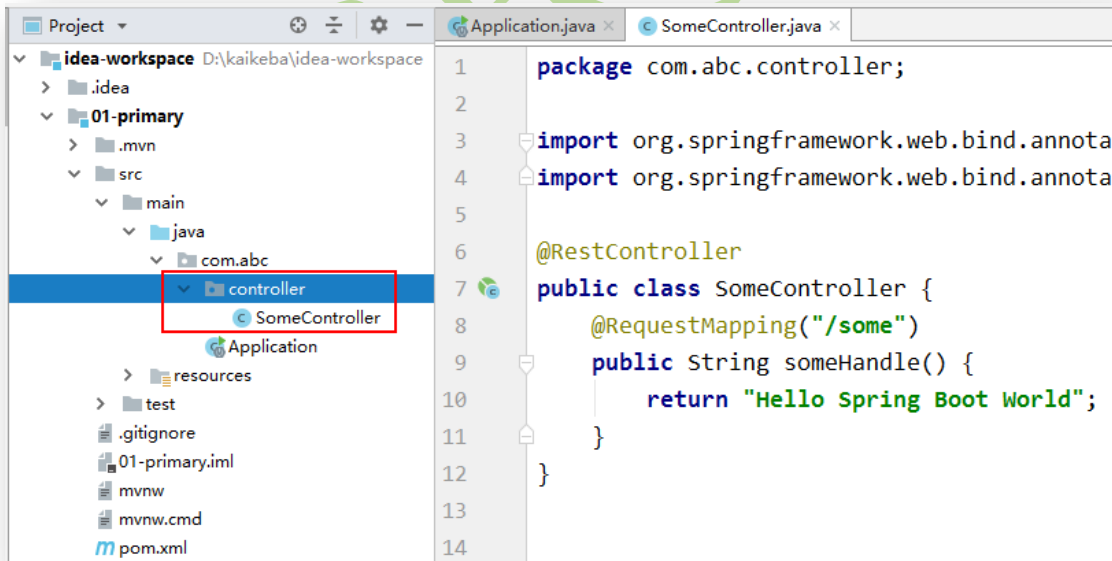


(2) 工程编辑

系统会在前面设置的包中自动生成一个启动类。



在启动类所在的包下再创建一个子包，在其中编写 SpringMVC 的处理器类。注意，要求代码所在的包必须是启动类所在包的子孙包，不能是同级包。对于本例而言，要求代码必须出现在 `com.abc` 包的子孙包中。



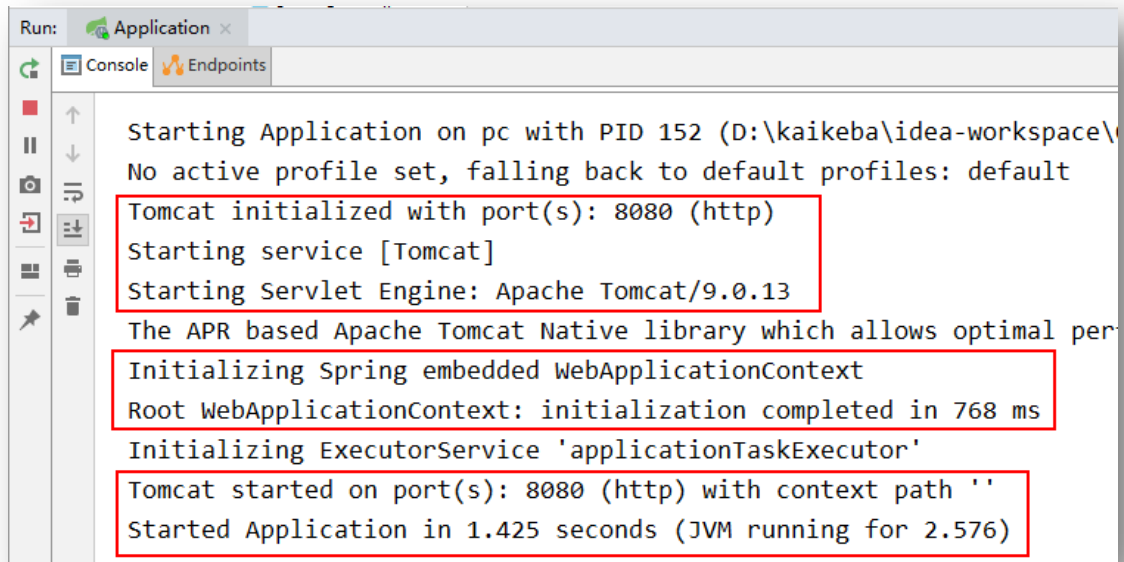
(3) 工程运行

对于 SpringBoot 程序的运行，若是在 Idea 环境下运行，比较简单，直接运行 main 类即可；若是没有 Idea 环境，则可打包后直接通过 java 命令运行。

A、Idea 下的运行

a、运行

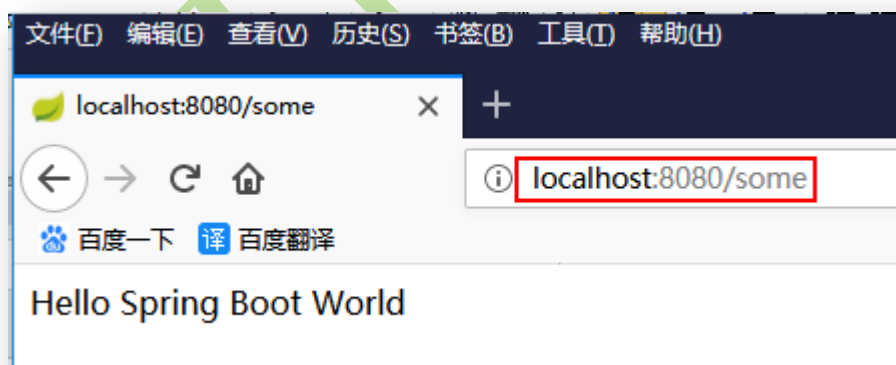
打开启动类并直接运行即可启动 SpringBoot 框架。在控制台查看启动信息可知：



- Tomcat 已启动，且为内置 Tomcat，端口号为 8080。
- 项目的上下文路径 Context Path，即访问该项目时的项目路径为空，即浏览器访问时无需项目名称。

b、访问

在浏览器地址栏中直接输入“主机 + 端口 + URI”即可访问该项目，无需项目名称。

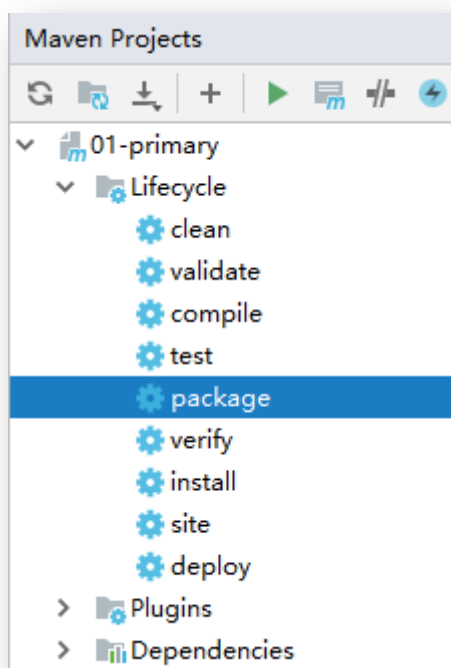


B、打包后运行

将 SpringBoot 工程打包后即可脱离 Idea 环境运行。

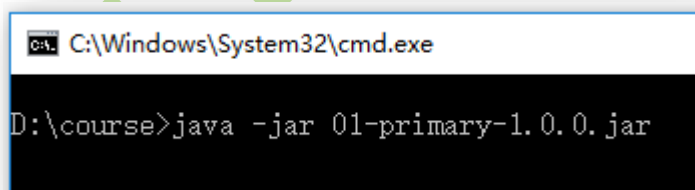
a、打包

使用 `package` 命令将工程打为 Jar 包。



b、运行

将打好的 Jar 包移动到任意目录，当然，也可在原来的 `target` 目录，在命令行即可通过 `java` 命令直接运行。例如，将 jar 包移动到 `D:/course` 目录中。

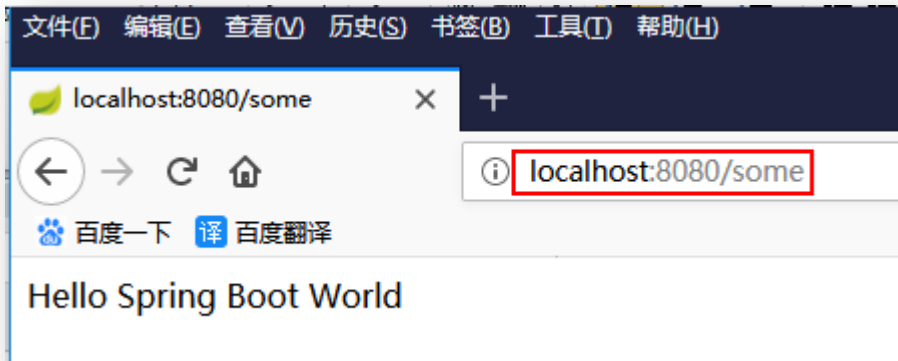


当看到如下提示时，表示应用启动成功。

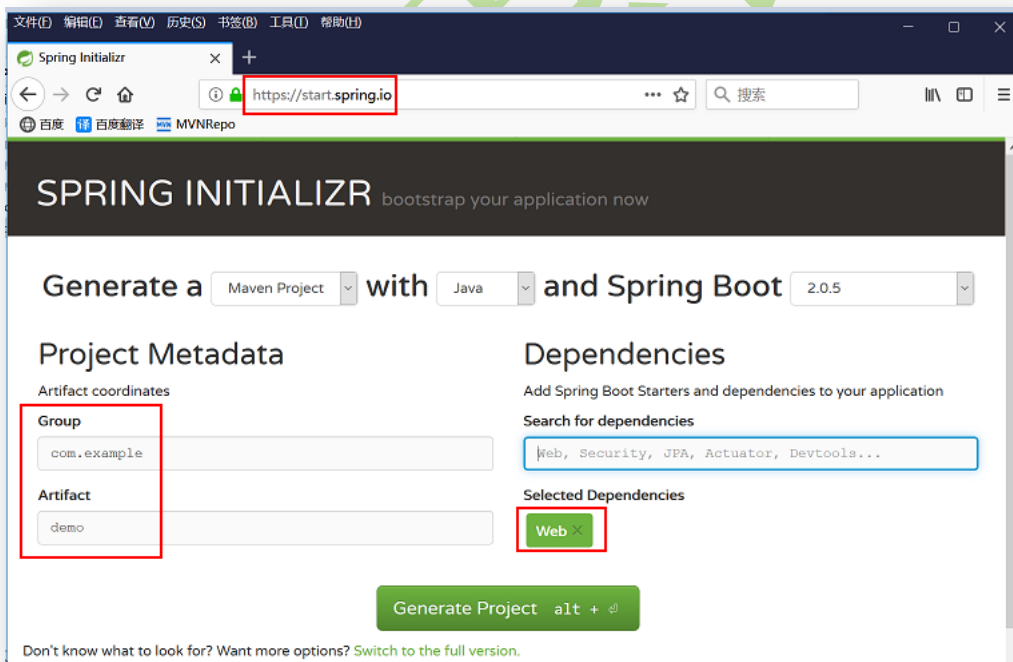
```
ping : Mapped URL path [/**/favicon.ico] onto handler of type [class
      : Registering beans for TMX exposure on startup
rver : Tomcat started on port(s): 8080 (http) with context path ''
      : Started Application in 1.901 seconds (JVM running for 2.243)
```

c、访问

在浏览器地址栏中直接输入“主机 + 端口 + URI”即可访问该项目，无需项目名称。



1.2.2 官网创建



在点击了 **Generate Project** 按钮后，即可打开一个下载对话框。官网将配置好的 **Spring Boot** 工程生成了一个 **zip** 压缩文件，只要我们将其下载到本地即可。

下载后，将其解压到 **idea-workspace** 中，在 **idea** 中即可马上看到该工程。注意，此时该工程是作为一个 **Module** 出现的。然后，再通过“导入外部 **Module** 方式”将该工程导入为 **Maven** 工程即可。

1.3 基于 war 的 Spring Boot 工程

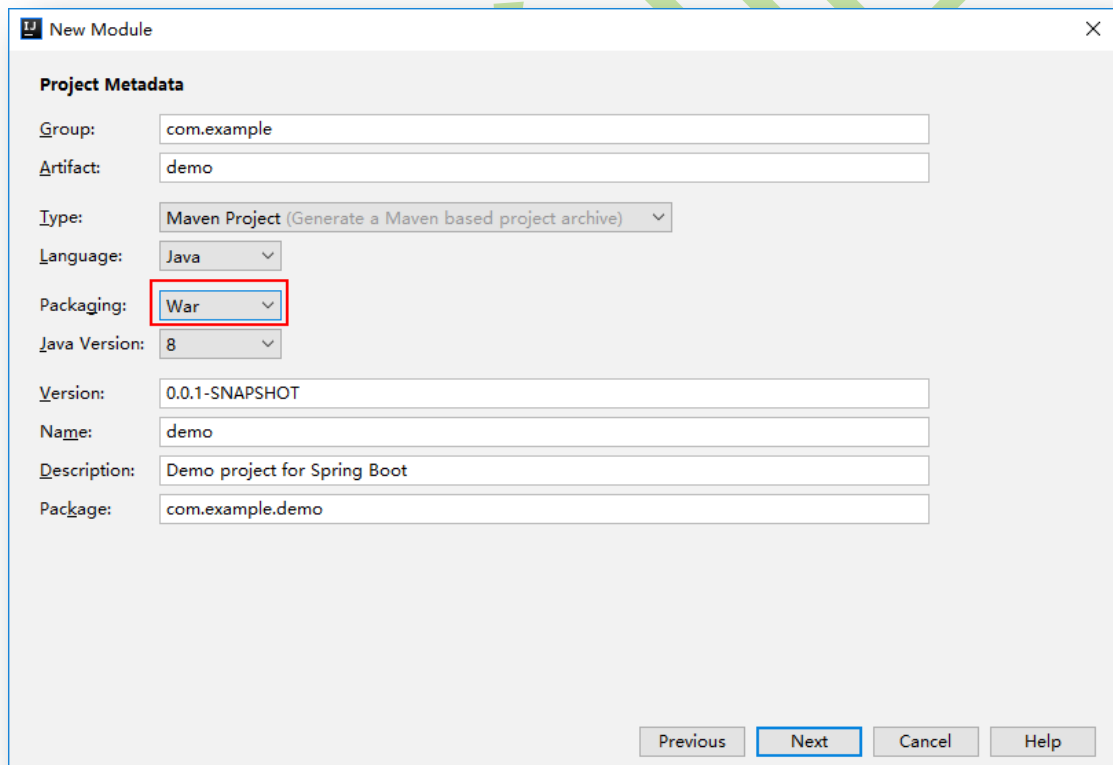
前面创建的 Spring Boot 工程最终被打为了 Jar 包，是以可执行文件的形式出现的，其使用了 Spring Boot 内嵌的 Tomcat 作为 Web 服务器来运行 web 应用的。新版 Dubbo 的监控中心工程就是典型的应用。

但在实际生产环境下，对于访问量不大的应用，直接以 Jar 包的形式出现，使用起来是非常方便的，不用部署了。但对于访问量较大的 Web 工程，我们不能使用 Tomcat，而要使用更为高效的商业 web 容器，例如 JBOSS、WebLogic 等，此时我们需要的是 war 包而非 jar 包。

下面我们来看一下如何使用 Spring Boot 将工程打为 war 包。

1.3.1 工程创建

其创建过程与前面打为 jar 包的相似，只以下面的窗口中选择 Packaging 为 War 即可。



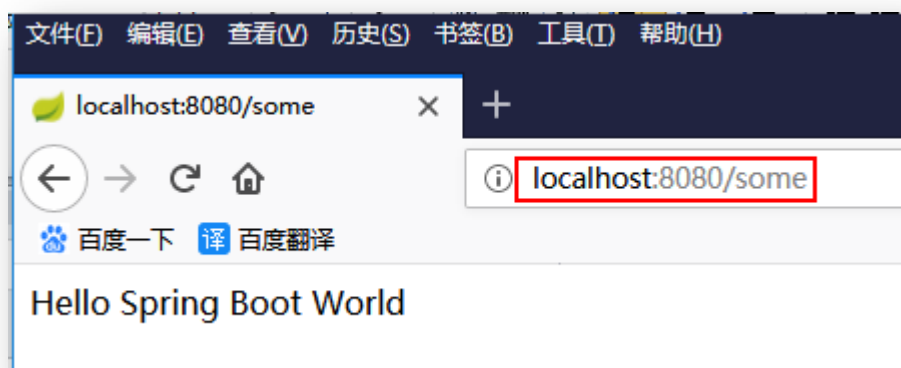
1.3.2 工程编辑

将第一个工程中的处理器复制到当前工程中即可

1.3.3 工程运行

(1) Idea 下运行与访问

打开启动类并直接运行即可启动 SpringBoot 工程，访问方式与之前的也相同。



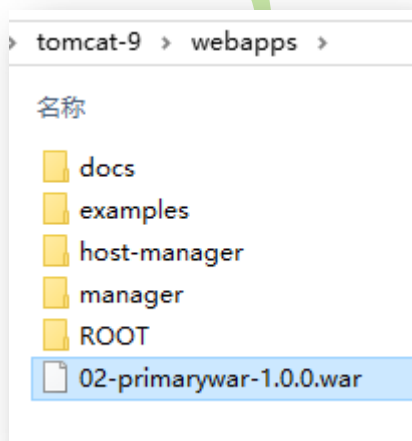
(2) 打包部署

A、打包

运行 `package` 命令，其会打为 war 包。

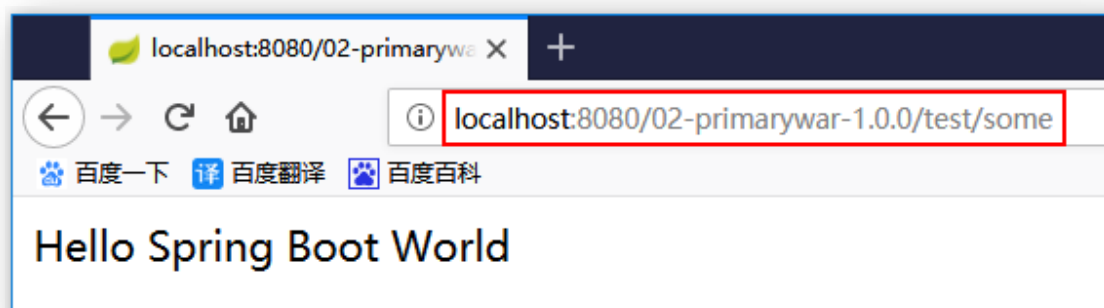
B、部署

找到该 war 包，将其部署到 Tomcat 的 `webapps` 目录中，启动 Tomcat。



C、访问

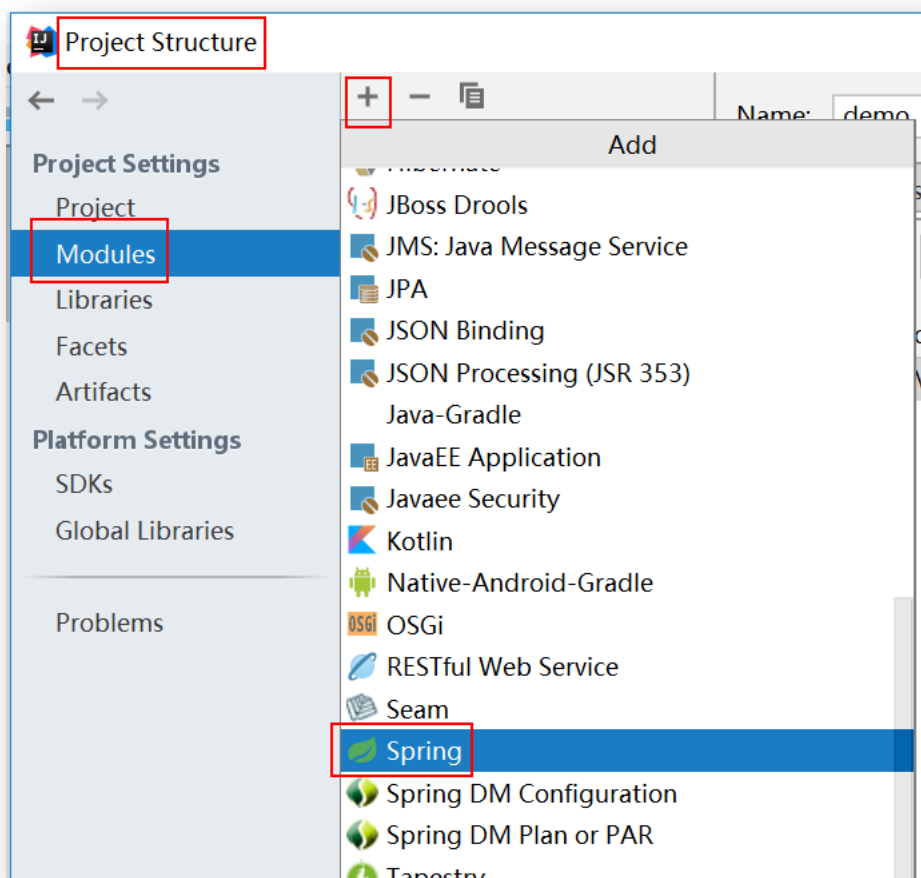
在浏览器中可以访问到该工程。注意，由于工程是部署到了 Tomcat 的 webapps 中，不是部署到 webapps/ROOT 中，所以在访问时需要指定工程名。



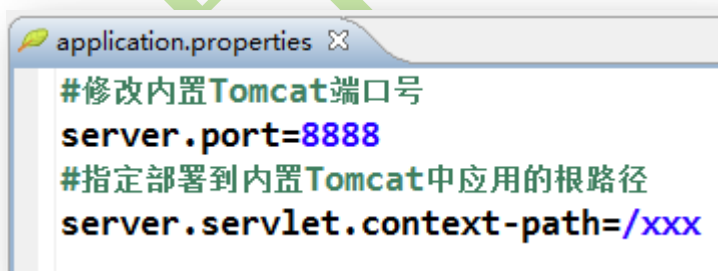
1.4 SpringBoot 的主配置文件

1.4.1 编辑器

Spring Boot 的主配置文件是 `src/main/resources` 中默认创建的 `spring.properties` 文件。该文件打开后是没有自动提示功能的。此时可以打开 Project Structure 窗口，在 Modules 中选中没有自动提示的工程，点击+号，找到 Spring，将其添加可以。此时的配置文件就有了自动提示功能，包括后面的 `yml` 文件也有了自动提示。



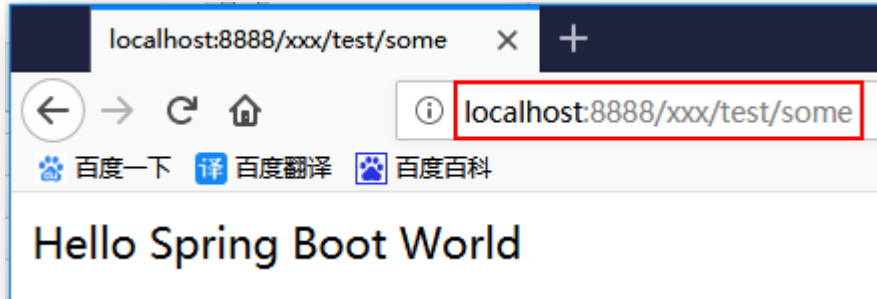
1.4.2 简单尝试



在 Eclipse 中运行工程后，查看日志文件可以看到端口号与应用的确发生的变化。

```
: Mapped URL path [/**/favicon.ico] onto handler of type [class org
: Registering beans for JMX exposure on startup
: Tomcat started on port(s): 8888 (http) with context path '/xxx'
: Started Application in 1.921 seconds (JVM running for 2.259)
```

在地址栏中需要键入新的端口号与应用的根名称。



不过需要注意,这里指定的 Tomcat 的端口号及应用的根路径,仅仅是针对于内置 Tomcat 的,是测试时使用的。将工程打为 war 包后部署到真正的 Tomcat,这些配置是不起作用的,即 Tomcat 的端口号为真正 Tomcat 的端口号,而项目的根路径为 war 包名称。

1.4.3 yml 文件

Spring Boot 的主配置文件也可使用 application.yml 文件。yml, 也可写为 yaml。

在开发之初 YAML 的本意是 Yet Another Markup Language (仍是一种标记语言)。后来为了强调这种语言是以数据为中心,而不是以标记为中心,所以将 YAML 解释为 Yaml Ain't Markup Language (Yaml 不是一种标记语言)。它是一种直观的能够被电脑识别的数据序列化格式,是一个可读性高并且容易被人阅读,容易和脚本语言交互,用来表达多级资源序列的编程语言。

yml 与 properties 文件的主要区别是对于多级属性,即 key 的显示方式不同。yml 文件在输入时,只需按照点(.)的方式输出 key 即可,输入完毕后回车即出现了如下形式。该形式要求冒号后与值之间要有一个空格。不同级别的属性间要有两个空格的缩进。

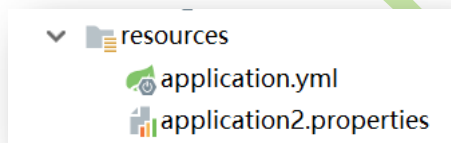
需要注意,很多脚本中的空格都是作为无效字符出现的,但 yml 脚本则是作为有效字符出现的,必须要保证空格的数量。

```

application.yml x
1  # 配置当前应用的端口号与上下文路径
2  server:
3    port: 9999      # 设置当前应用的端口号
4    servlet:
5      context-path: /xxx  # 设置当前应用的上下文路径
6      compression: # 设置Tomcat的响应压缩
7        enabled: true  # 开启响应内容压缩
8        min-response-size: 2048 # 设置开启响应压缩的最小文件大小, 默认2k
9        # mime-types: # 指定要压缩的文件类型, 默认已经包含了常见文本类型
10     tomcat:
11       uri-encoding: utf-8 # 设置Tomcat编码, 默认utf-8
12

```

在演示时需要注意，application.properties 与 application.yml 这两个文件只能有一个。要求文件名必须为 application。所以，此时可以将 application.properties 文件重命名为其它名字即可。



1.5 Actuator 监控器

Actuator[ˈæktʃʊˌeɪtə]（激励者;执行器）是 Spring Boot 提供的一个可挺拔模块，用于对工程进行监控。其通过不同的监控终端实现不同的监控功能。其功能与 Dubbo 的监控中心类似，不同的是，Dubbo 的监控中心是需要专门部署的，而 Spring Boot 的 Actuator 是存在于每一个工程中的。

1.5.1 基本环境搭建

随便一个 Spring Boot 工程中都可以使用 Actuator 对其进行监控。本例使用 01-primary 工程作为被监控对象，复制该工程并重命名为 03-actuatorTest，在该工程中进行修改。

(1) 导入依赖

```

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

```

```
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

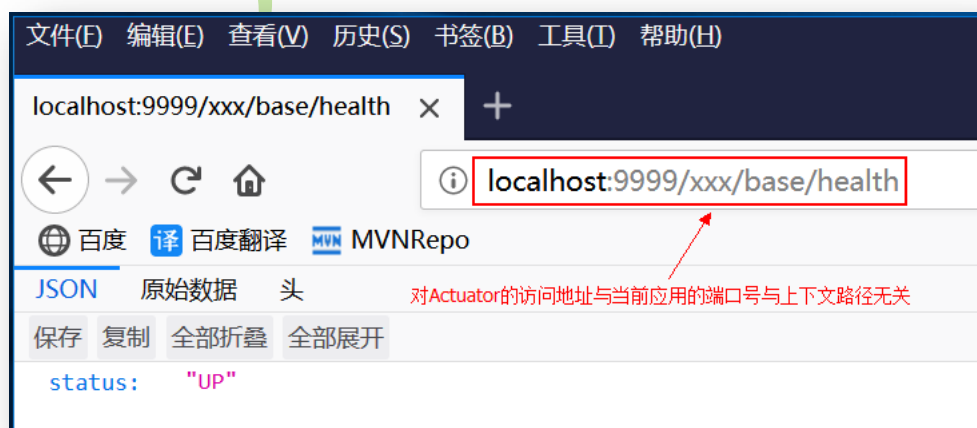
(2) 修改配置文件

A screenshot of a code editor showing the 'application.properties' file. The file contains configuration for the application's server and the Actuator. The configuration is as follows:

```
1 # 当前应用的端口号与上下文路径
2 server.port=8888
3 server.servlet.context-path=/first
4
5 # Actuator 监控的端口号与上下文路径
6 management.server.port=9999
7 management.server.servlet.context-path=/xxx
8 # 指定监控终端的基本路径，默认为actuator
9 management.endpoints.web.base-path=/base
10
```

(3) 访问测试—health 监控终端

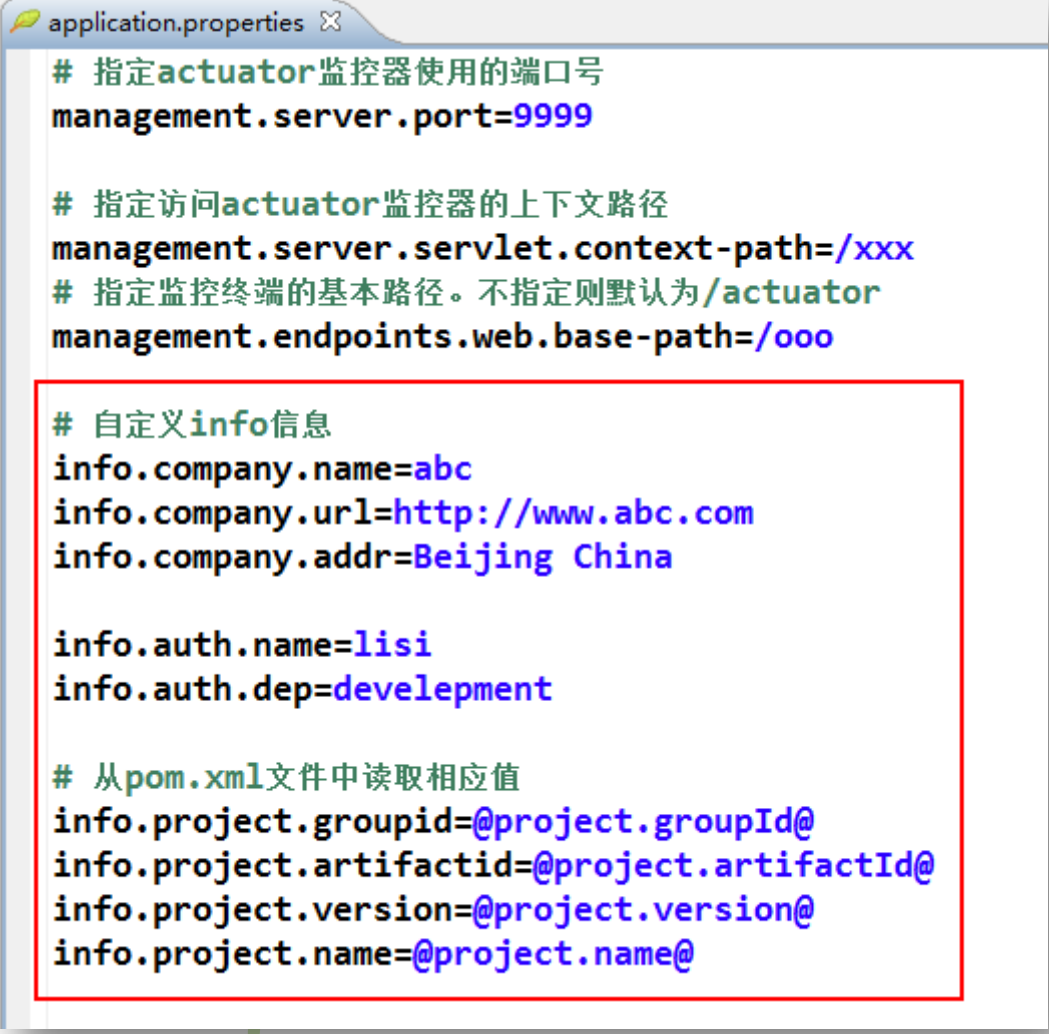
启动该工程后在地址栏访问，health 是一种监控终端，用于查看当前应用程序（微服务）的健康状况。



1.5.2 添加 Info 信息

(1) 修改配置文件

在配置文件中添加如下 Info 信息，则可以通过 info 监控终端查看到。



```
application.properties
# 指定actuator监控器使用的端口号
management.server.port=9999

# 指定访问actuator监控器的上下文路径
management.server.servlet.context-path=/xxx
# 指定监控终端的基本路径。不指定则默认为/actuator
management.endpoints.web.base-path=/ooo

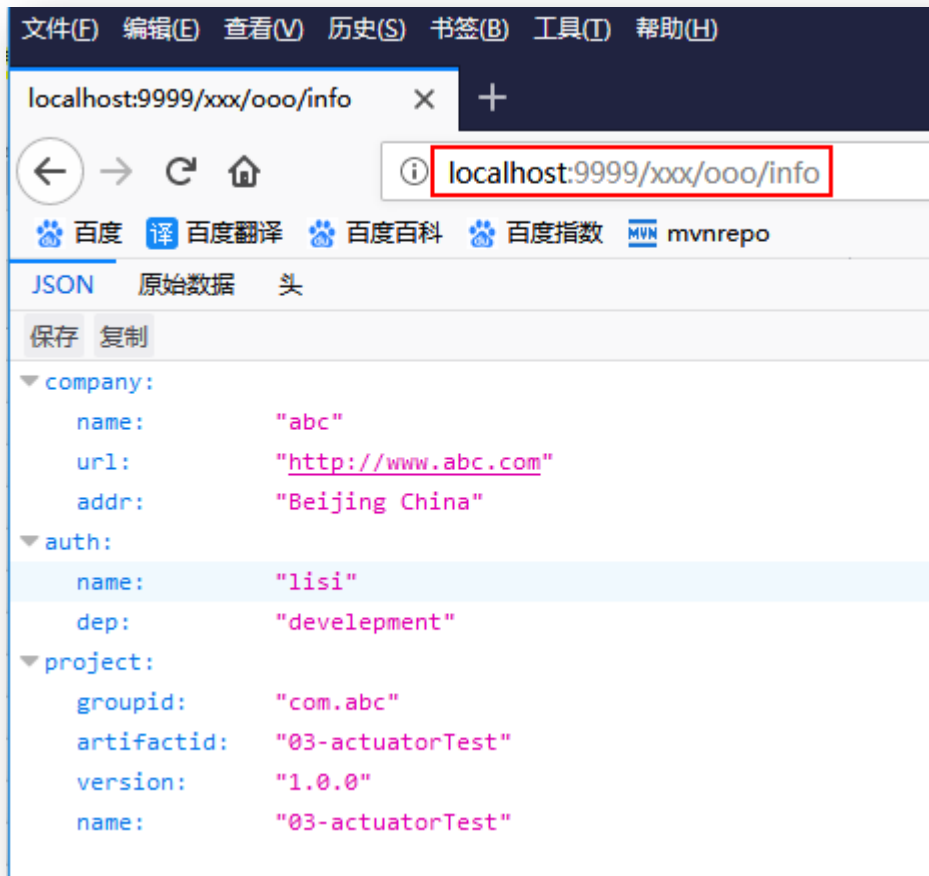
# 自定义info信息
info.company.name=abc
info.company.url=http://www.abc.com
info.company.addr=Beijing China

info.auth.name=lisi
info.auth.dep=develepment

# 从pom.xml文件中读取相应值
info.project.groupid=@project.groupId@
info.project.artifactid=@project.artifactId@
info.project.version=@project.version@
info.project.name=@project.name@
```

(2) 访问测试

这些信息都是以 JSON 的格式显示在浏览器的。



1.5.3 开放其它监控终端

默认情况下，Actuator 仅开放了 health 与 info 两个监控终端，但其还有很多终端可用，不过，需要手工开放。

(1) 修改配置文件

在配置文件中添加如下内容。

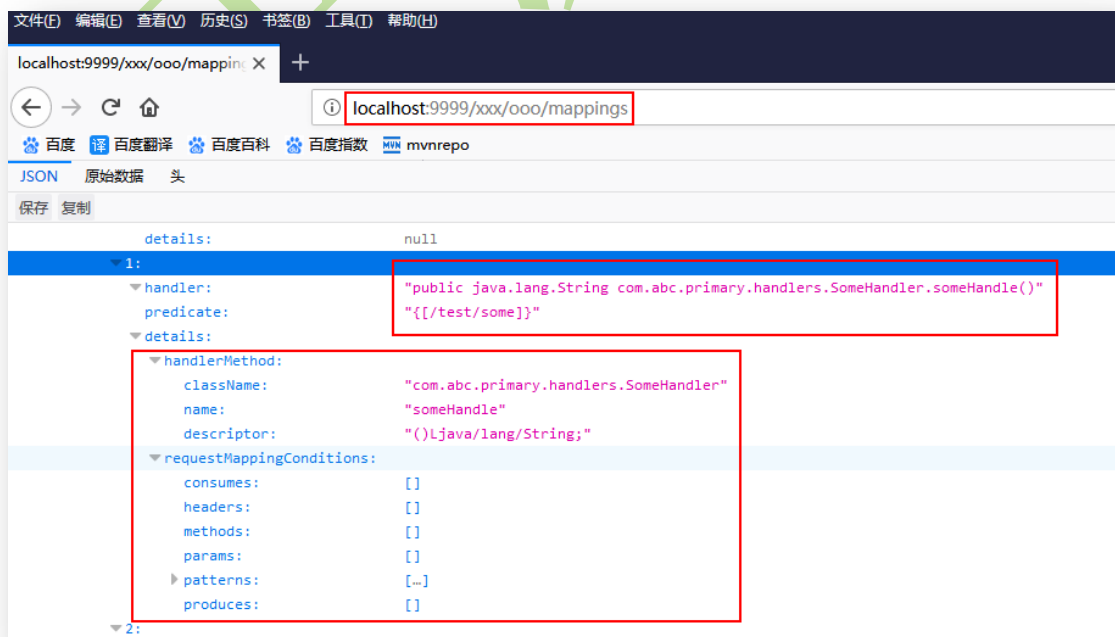
```
# 开放所有监控终端，默认只开启了health和info监控终端
# 在yaml中*号为关键字，需要将其使用双引号引起来 "*"
management.endpoints.web.exposure.include=*
```

```
# 设置actuator监控
management:
  server:
    port: 9999 # 指定actuator监控器使用的端口号
    servlet:
      context-path: /ooo # 指定访问actuator所要使用的上下文路径
  endpoints:
    web:
      base-path: /jjj # 指定监控终端的基本路径，默认为/actuator
      exposure:
        include: '*' # 开放所有监控终端
        # include: ['env', 'beans'] # 开放指定监控终端
```

(2) 访问测试

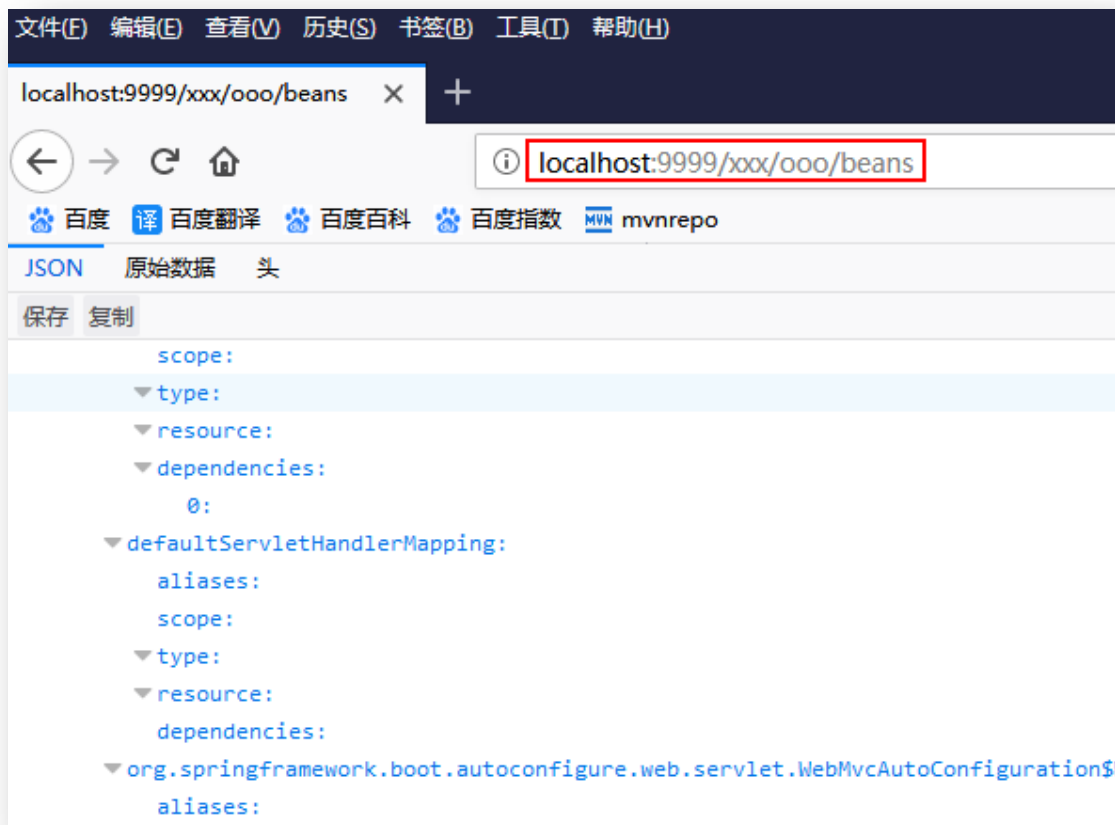
A、mappings 终端

通过 mappings 终端，可以看到当前工程中所有的 URI 与处理器的映射关系，及详细的处理器方法及其映射规则。很实用。



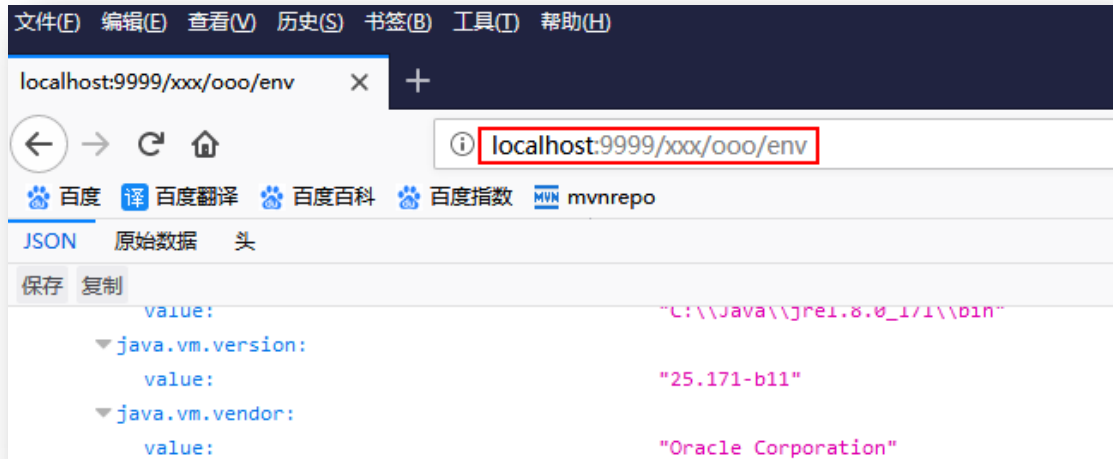
B、beans 终端

可以查看到当前应用中所有的对象信息。



C、env 终端

可以看到当前应用程序运行主机的所有软硬件环境信息。



1.5.4 单独关闭某些监控终端

在开放了所有监控终端的情况下，有些终端显示的信息并不想公开，此时可以单独关闭这些终端。

(1) 修改配置文件

在配置文件中添加如下内容。

```

# 开放所有监控终端，默认只开启了health和info监控终端
# 在yaml中*号为关键字，需要将其使用双引号引起来 "*"
management.endpoints.web.exposure.include=*

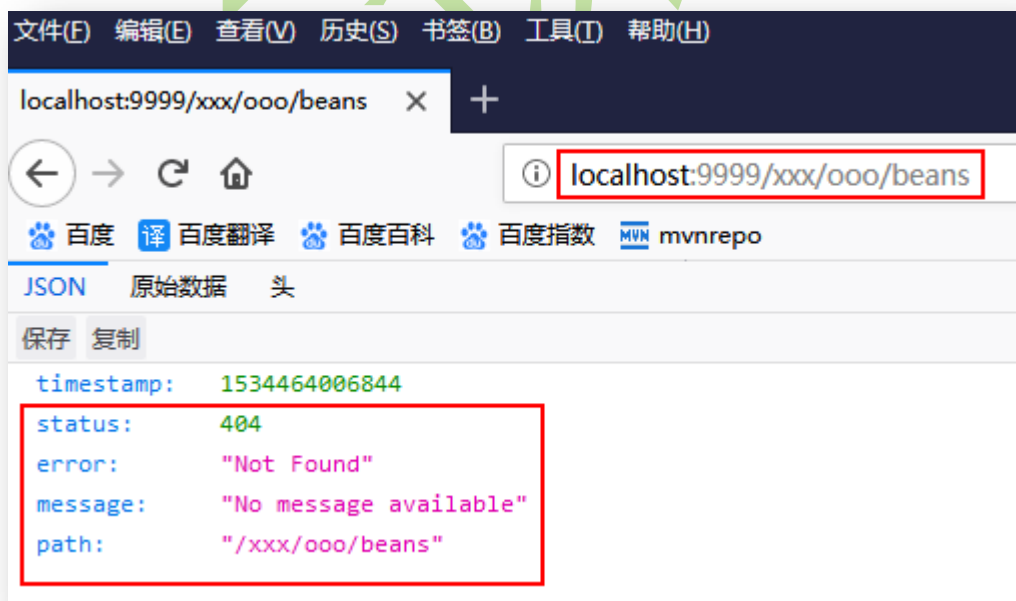
# 单独关闭env与beans监控终端
management.endpoints.web.exposure.exclude=env,beans

```

```
# 设置actuator监控
management:
  server:
    port: 9999 # 指定actuator监控器使用的端口号
    servlet:
      context-path: /ooo # 指定访问actuator所要使用的上下文路径
  endpoints:
    web:
      base-path: /jjj # 指定监控终端的基本路径，默认为/actuator
      exposure:
        include: '*' # 开放所有监控终端
        # include: ['env', 'beans'] # 开放指定监控终端
        exclude: ['env', 'beans'] # 关闭指定监控终端
```

(2) 访问测试

在关闭这些终端后，其它终端仍可继续使用。



1.5.5 常用的监控终端

在百度搜索“springboot actuator”即可找到如下表格。

HTTP 方法	监控终端	功能描述
GET	/autoconfig	提供了一份自动配置报告，记录哪些自动配置条件通过了，哪些没通过
GET	/configprops	描述配置属性(包含默认值)如何注入 Bean
GET	/beans	描述应用程序上下文里全部的 Bean，以及它们的关系
GET	/dump	获取线程活动的快照
GET	/env	获取全部环境属性
GET	/env/{name}	根据名称获取特定的环境属性值
GET	/health	报告应用程序的健康指标，这些值由 HealthIndicator 的实现类提供
GET	/info	获取应用程序的定制信息，这些信息由 info 打头的属性提供
GET	/mappings	描述全部的 URI 路径，以及它们和控制器(包含 Actuator 端点)的映射关系
GET	/metrics	报告各种应用程序度量信息，比如内存用量和 HTTP 请求计数
GET	/metrics/{name}	报告指定名称的应用程序度量值
POST	/shutdown	关闭应用程序，要求 endpoints.shutdown.enabled 设置为 true
GET	/trace	提供基本的 HTTP 请求跟踪信息(时间戳、HTTP 头等)

第2章 Spring Boot 重要用法

2.1 自定义异常页面

对于 404、405、500 等异常状态，服务器会给出默认的异常页面，而这些异常页面一般都是英文的，且非常不友好。我们可以通过简单的方式使用自定义异常页面，并将默认状态码页面进行替换。

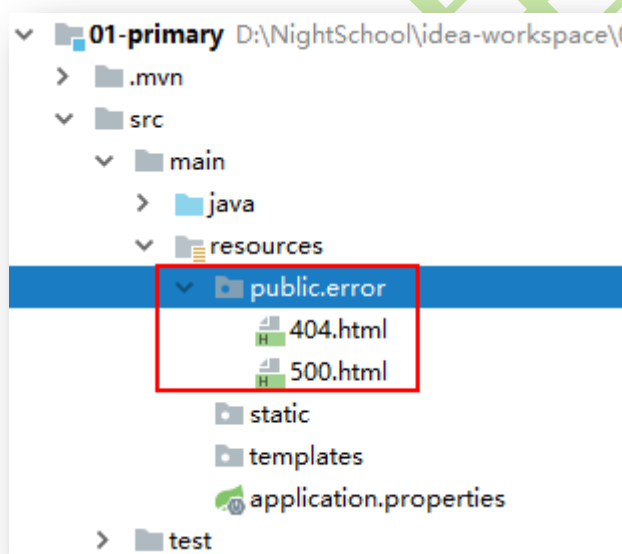
直接在前面程序上修改即可，无需创建新的工程。

2.1.1 定义目录

在 `src/main/resources` 目录下再定义新的目录 `public/error`，必须是这个目录名称。

2.1.2 定义异常页面

在 `error` 目录中定义异常页面。这些异常页面的名称必须为相应的状态码，扩展名为 `html`。



```

500.html x
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>500</title>
6  </head>
7  <body>
8      this is 500 error.
9  </body>
10 </html>
11

```

```

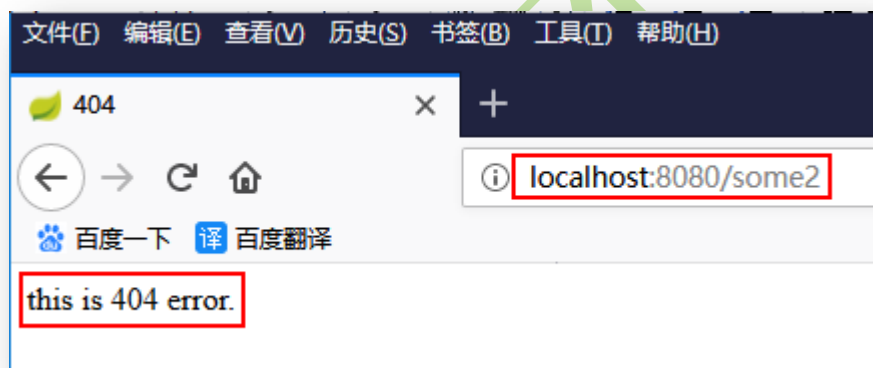
404.html x
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>404</title>
6  </head>
7  <body>
8      this is 404 error.
9  </body>
10 </html>
11

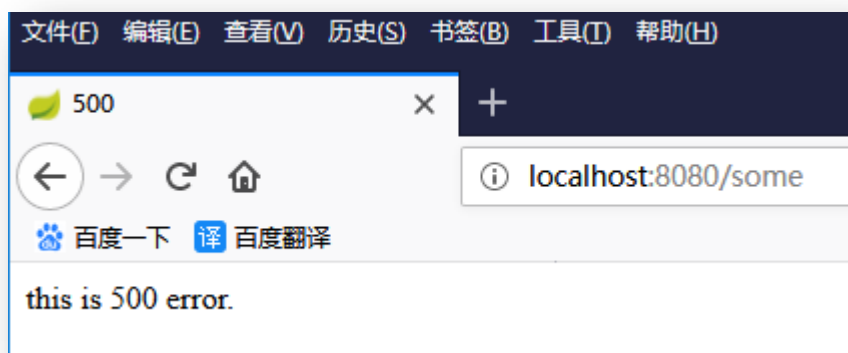
```


2.1.3 修改处理器模拟 500 错误

```
@RestController
public class SomeController {
    @RequestMapping("/some")
    public String someHandle() {
        int i = 3 / 0;
        return "Hello Spring Boot World";
    }
}
```

2.1.4 访问效果





2.2 单元测试

2.2.1 定义工程

直接在前面工程中修改即可。

2.2.2 定义 Service 接口

```
public interface SomeService {  
    void doSome();  
}
```

2.2.3 定义 Service 两个实现类

注意，实现类上要添加@Service 注解，以交给 Spring 容器来管理。

```
@Service
public class SomeServiceImpl implements SomeService {
    @Override
    public void doSome() {
        System.out.println("SomeServiceImpl的doSome()");
    }
}
```

```
@Service
public class OtherServiceImpl implements SomeService {
    @Override
    public void doSome() {
        System.out.println("OtherServiceImpl的doSome()");
    }
}
```

2.2.4 修改测试类

打开 src/test/java 中的测试类，在其中直接添加测试方法即可。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ApplicationTests {
    // 若接口只有一个实现类，则可以使用byType方式自动注入
    // 但现在存在两个实现类，只能使用byName方式自动注入
    @Autowired
    @Qualifier("someServiceImpl")
    private SomeService someService;

    @Autowired
    @Qualifier("otherServiceImpl")
    private SomeService otherService;

    @Test
    public void test01() {
        someService.doSome();
        otherService.doSome();
    }
}
```

2.3 多环境选择

2.3.1 什么是多环境选择

以下两种场景下需要进行“多环境选择”。

(1) 相同代码运行在不同环境

在开发应用时，通常同一套程序会被运行在多个不同的环境，例如，开发、测试、生产环境等。每个环境的数据库地址、服务器端口号等配置都会不同。若在不同环境下运行时将配置文件修改为不同内容，那么，这种做法不仅非常繁琐，而且很容易发生错误。

此时就需要定义出不同的配置信息，在不同的环境中选择不同的配置。

(2) 不同环境执行不同实现类

在开发应用时，有时不同的环境，需要运行的接口的实现类也是不同的。例如，若要开发一个具有短信发送功能的应用，开发环境中要执行的 `send()` 方法仅需调用短信模拟器即可，而生产环境中要执行的 `send()` 则需要调用短信运营商所提供的短信发送接口。

此时就需要开发两个相关接口的实现类去实现 `send()` 方法，然后在不同的环境中自动选择不同的实现类去执行。

2.3.2 需求

下面将实现如下功能：存在开发与生产两种环境，不同环境使用不同配置文件，不同环境调用不同接口实现类。使用不同端口号对不同的配置文件加以区分。

2.3.3 多配置文件实现方式

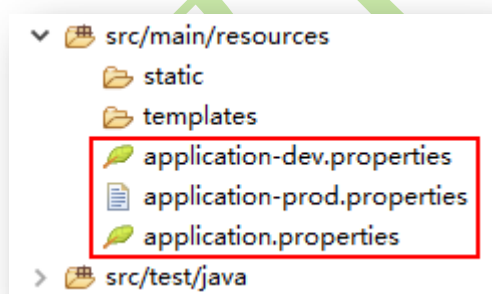
(1) 定义工程

复制前面打为 Jar 包的工程，并重命名为 03-multiEnv。

(2) 定义配置文件

A、定义多个配置文件

在 `src/main/resources` 中再定义两个配置文件，分别对应开发环境与生产环境。



```
application-dev.properties
server.port=8888
server.servlet.context-path=/ddd
```

```
application-prod.properties
server.port=9999
server.servlet.context-path=/ppp
```

```
application.properties
# Other Configurations
spring.profiles.active=dev
```

B、说明

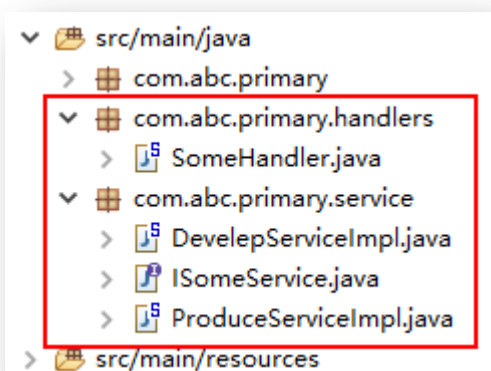
在 Spring Boot 中多环境配置文件名需要满足 `application-{profile}.properties` 的格式，其中 `{profile}` 为对应的环境标识，例如，

- `application-dev.properties`: 开发环境
- `application-test.properties`: 测试环境
- `application-prod.properties`: 生产环境

至于哪个配置文件会被加载，则需要在 `application.properties` 文件中通过 `spring.profiles.active` 属性来设置，其值对应 `{profile}` 值。例如，`spring.profiles.active=test` 就会加载 `application-test.properties` 配置文件内容。

在生产环境下，`application.properties` 中一般配置通用内容，并设置 `spring.profiles.active` 属性的值为 `dev`，即，直接指定要使用的配置文件为开发时的配置文件，而对于其它环境的选择，一般是通过命令行方式去激活。配置文件 `application-{profile}.properties` 中则配置各个环境的不同内容。

(3) 定义业务代码



A、定义业务接口

```
// 业务接口
public interface ISomeService {
    String send();
}
```

B、定义两个实现类

```
// 开发阶段使用的业务接口实现类
@Service
@Profile("dev")
public class DevelopServiceImpl implements ISomeService {

    @Override
    public String send() {
        return "开发阶段使用的业务接口实现类";
    }

}
```

```
// 生产环境下使用的业务接口实现类
@Service
@Profile("prod")
public class ProduceServiceImpl implements ISomeService {

    @Override
    public String send() {
        return "生产环境下使用的业务接口实现类";
    }

}
```

c、说明

在实现类上添加@Profile 注解，并在注解参数中指定前述配置文件中的{profile}值，用于指定该实现类所适用的环境。

(4) 定义处理器

```
@Controller
@RequestMapping("/test")
public class SomeHandler {

    @Autowired
    private ISomeService service;

    @RequestMapping("/some")
    public @ResponseBody String someHandle() {
        return service.send();
    }

}
```

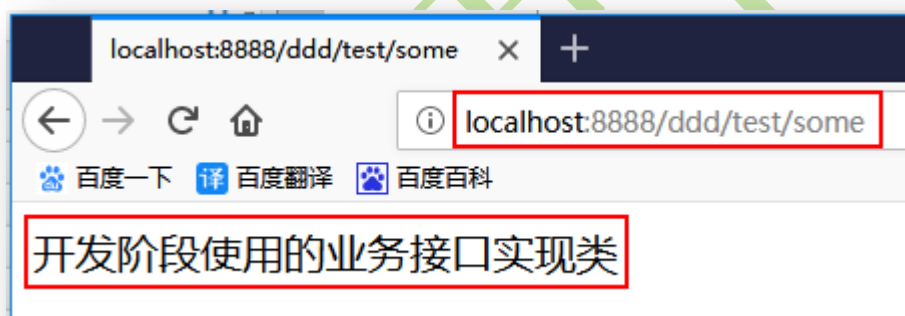

(5) Idea 下运行与访问

A、运行

打开主类在 Idea 中直接运行，即可在控制台看到默认使用的是开发环境，即端口号使用的是 8888，而工程的根路径为/ddd。

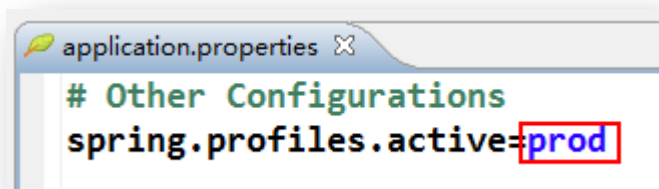
```
: mapped URL path [/***favicon.ico] onto handler of type [class org  
: Registering beans for JMX exposure on startup  
: Tomcat started on port(s): 8888 (http) with context path '/ddd'  
: Started Application in 2.099 seconds (JVM running for 2.429)
```

B、访问



从页面显示内容可知，其执行的业务接口实现类为 DevelopServiceImpl，即开发阶段应使用的实现类。

C、修改配置文件

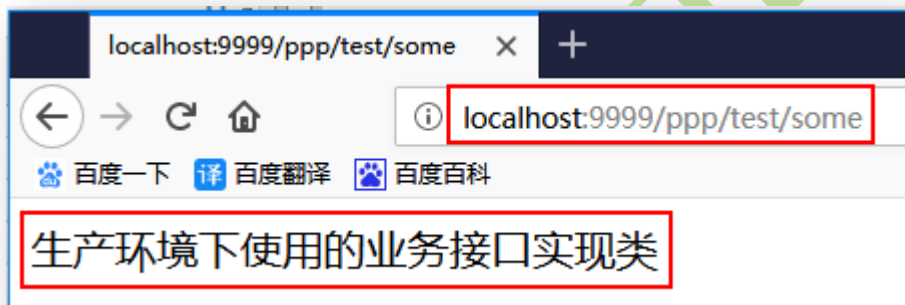


D、再运行

将上次运行停掉后再次运行主类，即可在控制台看到使用的是生产环境，即端口号使用的是 9999，而工程的根路径为 /ppp。

```
: Registering beans for JMX exposure on startup
: Tomcat started on port(s): 9999 (http) with context path '/ppp'
: Started Application in 1.958 seconds (JVM running for 2.269)
```

E、再访问



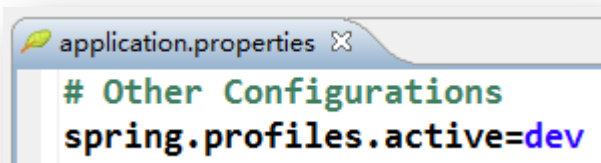
从页面显示内容可知，此次执行的业务接口实现类为 ProduceServiceImpl，即生产环境下应使用的实现类。

(6) 在命令行下选择环境

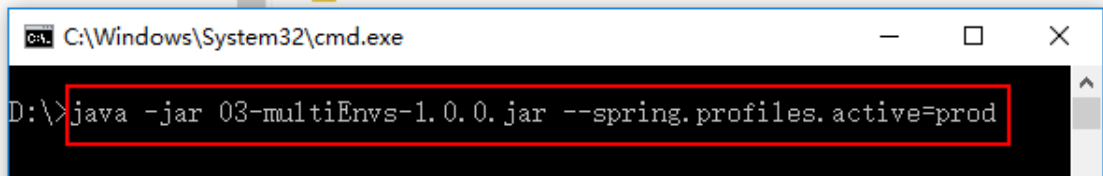
将工程打为 Jar 包后，在命令行运行。若要想切换运行环境，必须要修改主配置文件吗？答案是否定的。只需添加一个命令参数即可动态指定。

A、在命令行下运行 Jar 包

例如，现在的主配置文件中指定的是 dev 环境。

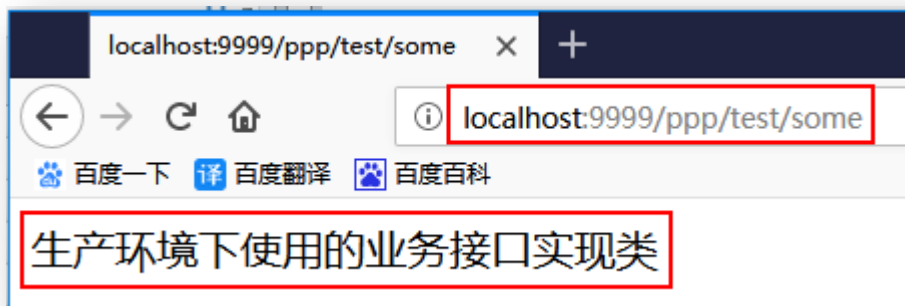


将当前工程打为 Jar 包后，在命令行运行时添加如下参数。



```
C:\Windows\System32\cmd.exe
D:\>java -jar 03-multiEnvs-1.0.0.jar --spring.profiles.active=prod
```

此时执行的就是生产环境，调用的就是 `ProduceServiceImpl` 类。



B、说明

在命令行中添加的参数可以是写在配置文件中的任意属性。其原理是命令行设置的属性值的优先级高于配置文件的。

2.3.4 单配置文件实现方式

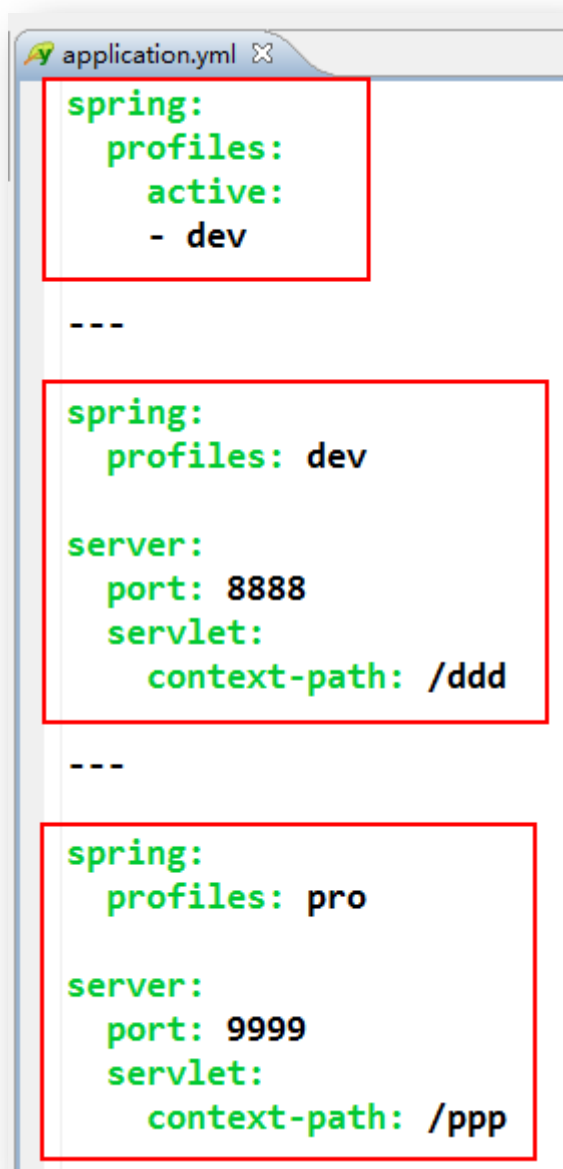
这种实现方式只能使用 `application.yml` 文件，使用 `application.properties` 文件好像文件本身就会出错。

(1) 定义工程

复制前面的 3-multiEnv 工程，并重命名为 03-multiEnv2。

(2) 修改配置文件

将原有的配置文件全部删除，然后定义 `application.yml` 文件。需要注意的是，这三部分之间是由三个减号 (-) 分隔的，必须是三个。而这三部分充当着之前的三个配置文件。



(3) 运行与访问

运行与访问方式与前面的多配置文件的完全相同。

2.4 读取自定义配置

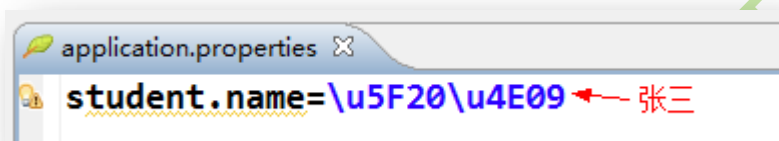
自定义配置，可以是定义在主配置文件 `application.properties` 中的自定义属性，也可以是自定义配置文件中的属性。

2.4.1 读取主配置文件中的属性

(1) 定义工程

复制前面打为 Jar 包的工程，并重命名为 04-customConfig。

(2) 修改主配置文件



(3) 修改 SomeHandler 类

```
@Controller
@RequestMapping("/test")
public class SomeHandler {

    @Value("${student.name}")
    private String name;

    @RequestMapping("/some")
    public @ResponseBody String someHandle() {
        return name;
    }
}
```

在@Value 注解中通过\${}符号可以读取指定的属性值。

2.4.2 读取指定配置文件中的属性

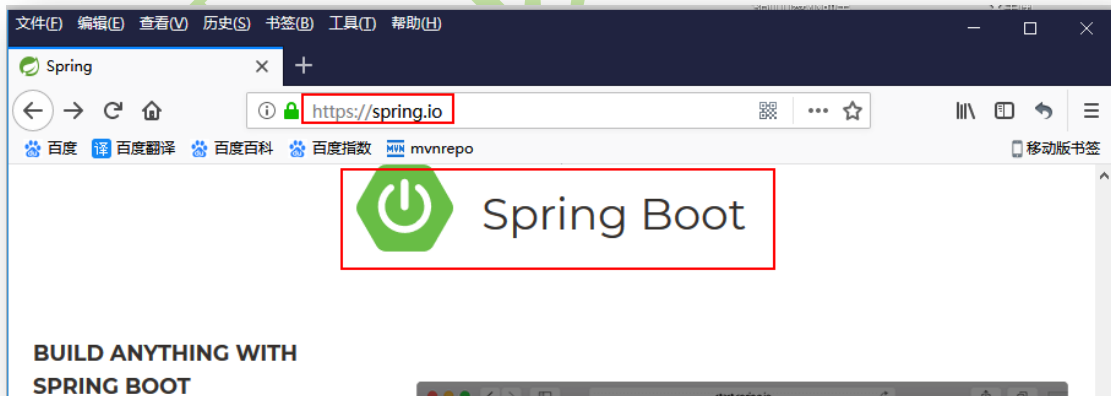
一般情况下，主配置文件中存放系统中定义好的属性设置，而自定义属性一般会写入自定义的配置文件中。也就是说，Java 代码除了可以读取主配置文件中的属性外，还可以读取指定配置文件中的属性，可以通过 `@PropertySource` 注解加载指定的配置文件。

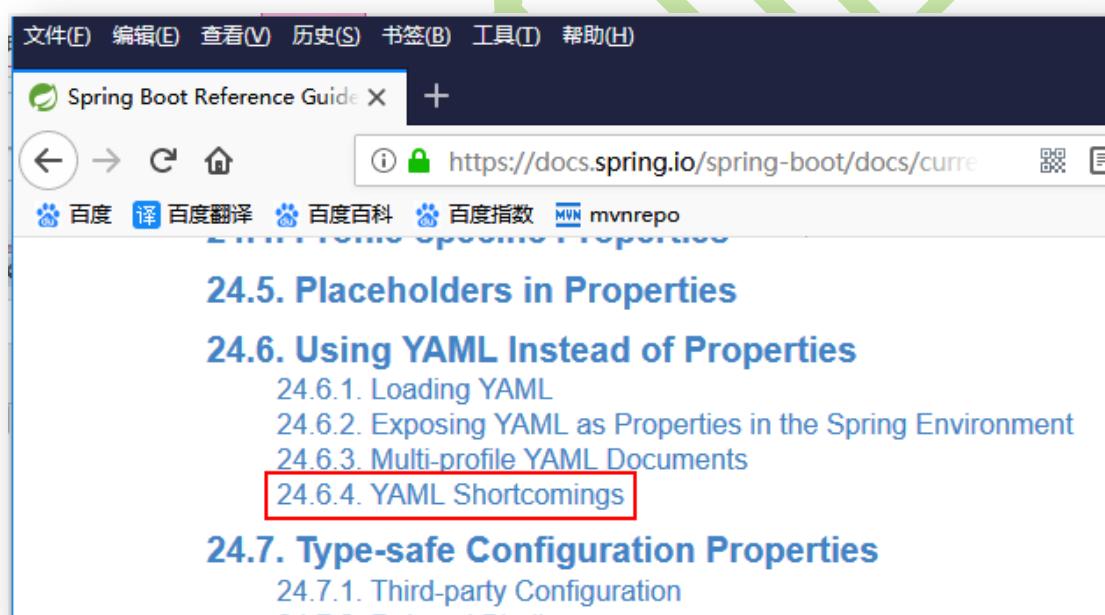
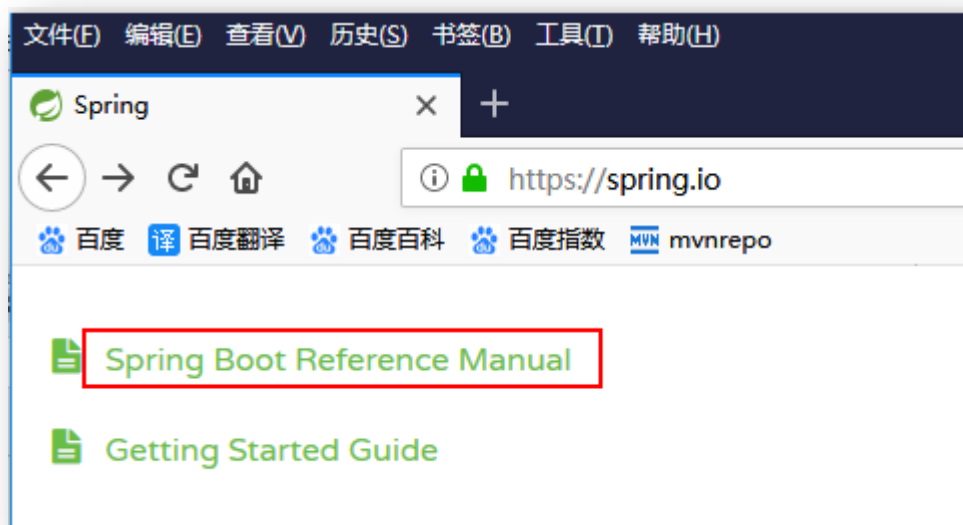
(1) 不能自定义 yml 文件



spring boot 官网给出说明，`@PropertySource` 注解不能加载 yml 文件。所以其建议自定义配置文件就使用属性文件。

官方文档地址：



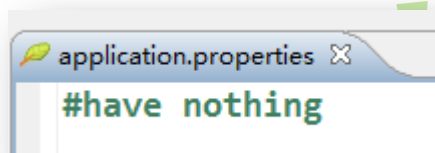




(2) 定义工程

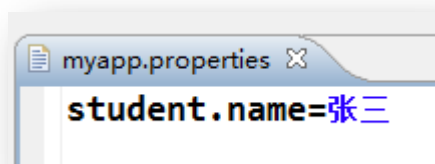
复制 04-customConfig 工程，并重命名为 04-customConfig2。

(3) 修改主配置文件



(4) 自定义配置文件

该配置文件为 properties 文件，文件名随意，存放在 src/main/resources 目录中。



(5) 修改 SomeHandler 类

若属性的值存在中文，则需要添加 encoding 属性。

```
@Controller
@RequestMapping("/test")
@PropertySource(value="classpath:myapp.properties", encoding="utf-8")
public class SomeHandler {

    @Value("${student.name}")
    private String name;

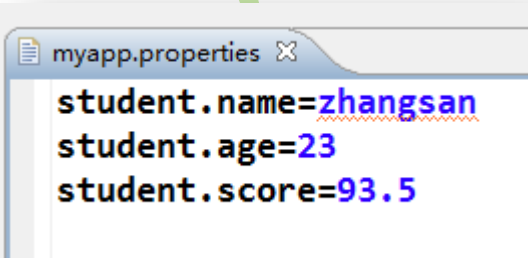
    @RequestMapping("/some")
    public @ResponseBody String someHandle() {
        return name;
    }
}
```

2.4.3 读取对象属性

(1) 定义工程

复制 04-customConfig2 工程，并重命名为 04-customConfig3。

(2) 修改自定义配置文件



```
myapp.properties
student.name=zhangsan
student.age=23
student.score=93.5
```

此时的 student 称为对象属性。

(3) 定义配置属性类

```
@Component
@PropertySource("classpath:myapp.properties")
@ConfigurationProperties("student")
public class Student {
    private String name;
    private int age;
    private double score;

    // getter and setter
    // toString()
```

说明：

- @PropertySource 用于指定要读取的配置文件
- @ConfigurationProperties 用于指定要读取配置文件中的对象属性，即指定要读取的配置文件属性的前缀
- @Component 表示当前从配置文件读取来的对象，由 Spring 容器创建
- 要保证类的属性名要与配置文件中的对象属性名相同

(4) 修改 SomeHandler 类

```
@Controller
@RequestMapping("/test")
public class SomeHandler {

    @Autowired
    private Student student;

    @RequestMapping("/some")
    public @ResponseBody String someHandle() {

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

        return student.getName();
    }
}
```

2.4.4 读取 List<String>属性

(1) 定义工程

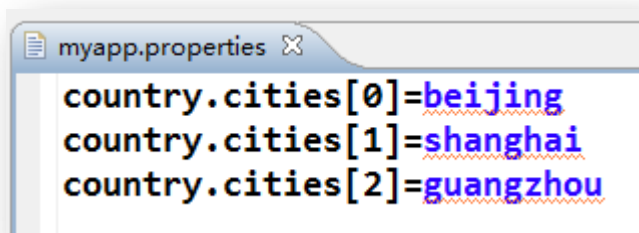
复制 04-customConfig3 工程，并重命名为 04-customConfig4。

(2) 导入依赖

注解@ConfigurationProperties 注解需要以下依赖。

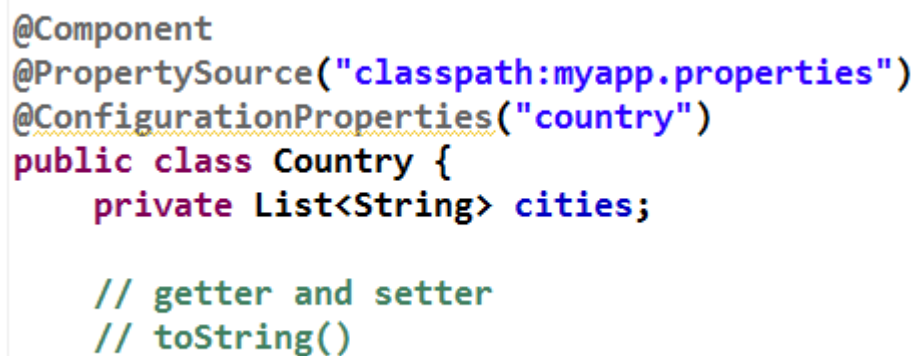
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

(3) 修改自定义配置文件



```
myapp.properties
country.cities[0]=beijing
country.cities[1]=shanghai
country.cities[2]=guangzhou
```

(4) 定义配置属性类



```
@Component
@PropertySource("classpath:myapp.properties")
@ConfigurationProperties("country")
public class Country {
    private List<String> cities;

    // getter and setter
    // toString()
```

(5) 修改 SomeHandler 类

```
@Controller
@RequestMapping("/test")
public class SomeHandler {

    @Autowired
    private Country country;

    @RequestMapping("/some")
    public @ResponseBody String someHandle() {

        for (String city : country.getCities()) {
            System.out.println(city);
        }

        return country.toString();
    }
}
```

2.4.5 读取 List<Object>属性

(1) 定义工程

复制 04-customConfig4 工程，并重命名为 04-customConfig5。

(2) 修改自定义配置文件

```
myapp.properties
group.students[0].name=zhangsan
group.students[0].age=23
group.students[0].score=93.5

group.students[1].name=lisi
group.students[1].age=24
group.students[1].score=94.5

group.students[2].name>wangwu
group.students[2].age=25
group.students[2].score=95.5
```

(3) 定义配置属性类

注意，Student 类无需任何注解。

```
public class Student {
    private String name;
    private int age;
    private double score;

    // getter and setter
    // toString()
```

```
@Component
@PropertySource("classpath:myapp.properties")
@ConfigurationProperties("group")
public class Group {
    private List<Student> students;

    // getter and setter
    // toString()
```

(4) 修改 SomeHandler 类

```
@Controller
@RequestMapping("/test")
public class SomeHandler {

    @Autowired
    private Group group;

    @RequestMapping("/some")
    public @ResponseBody String someHandle() {

        for (Student student : group.getStudents()) {
            System.out.println(student);
        }

        return group.toString();
    }
}
```

2.5 Spring Boot 下使用 JSP 页面

在 Spring Boot 下直接使用 JSP 文件，其是无法解析的，需要做专门的配置。

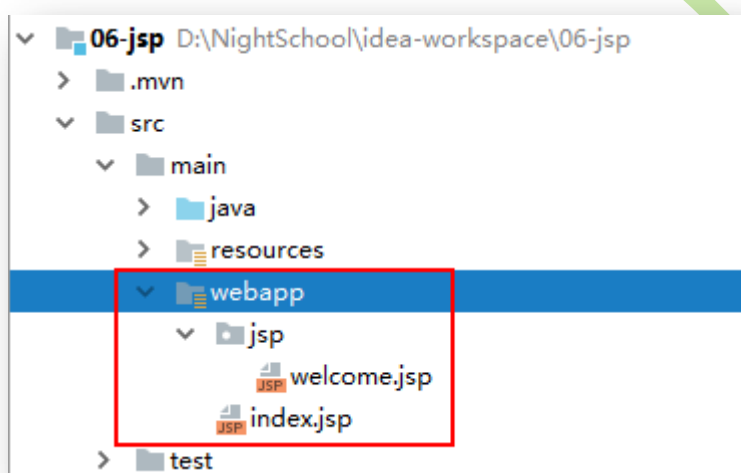
2.5.1 直接添加 JSP 文件

(1) 定义工程

复制第一个工程，并重命名为 06-jsp。

(2) 创建 webapp 目录

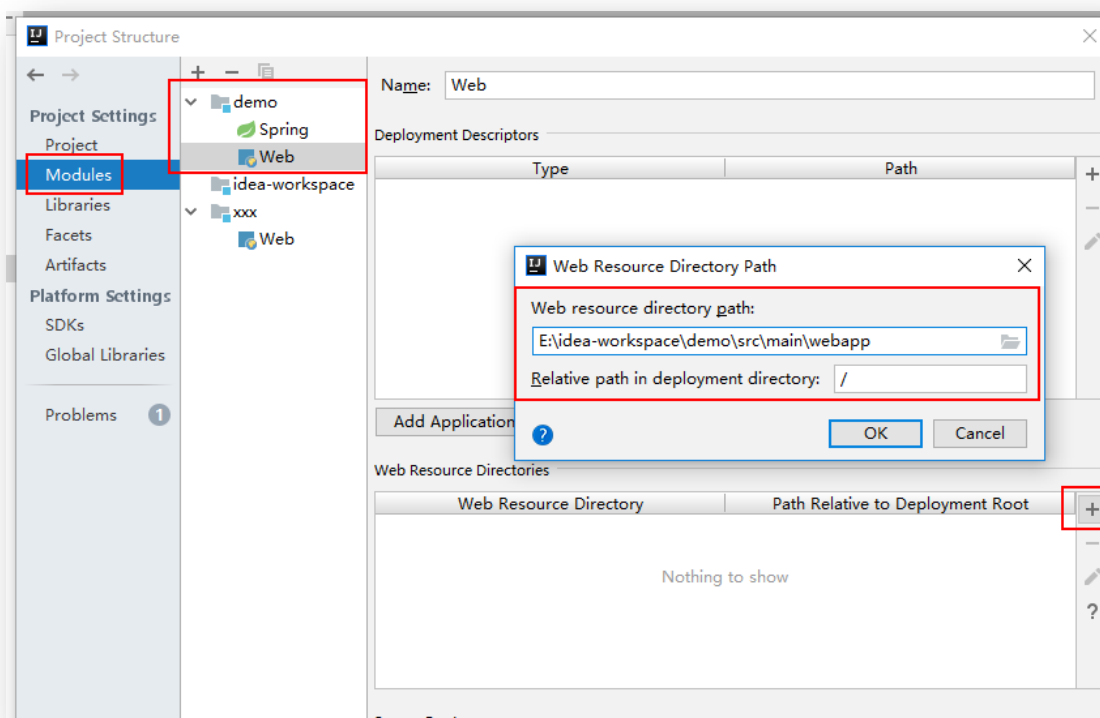
在 src/main 下创建 webapp 目录，用于存放 jsp 文件。这就是一个普通的目录，无需执行 Mark Directory As。



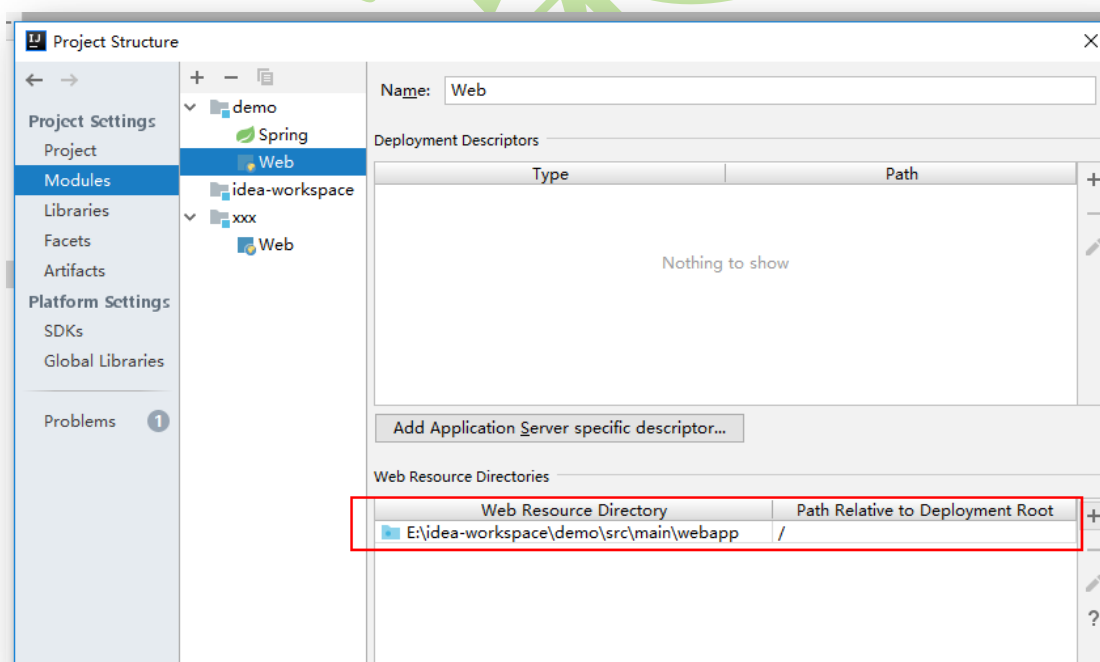
(3) 创建 index.jsp

A、指定 web 资源目录

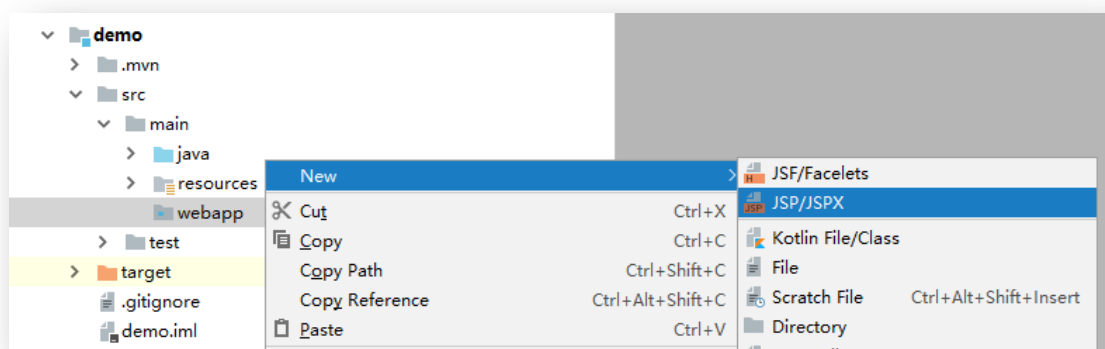
在 spring boot 工程中若要创建 jsp 文件，一般是需要在 src/main 下创建 webapp 目录，然后在该目录下创建 jsp 文件。但通过 Alt + Insert 发现没有创建 jsp 文件的选项。此时，需要打开 Project Structure 窗口，将 webapp 目录指定为 web 资源目录，然后才可以创建 jsp 文件。



指定后便可看到下面的窗口情况。

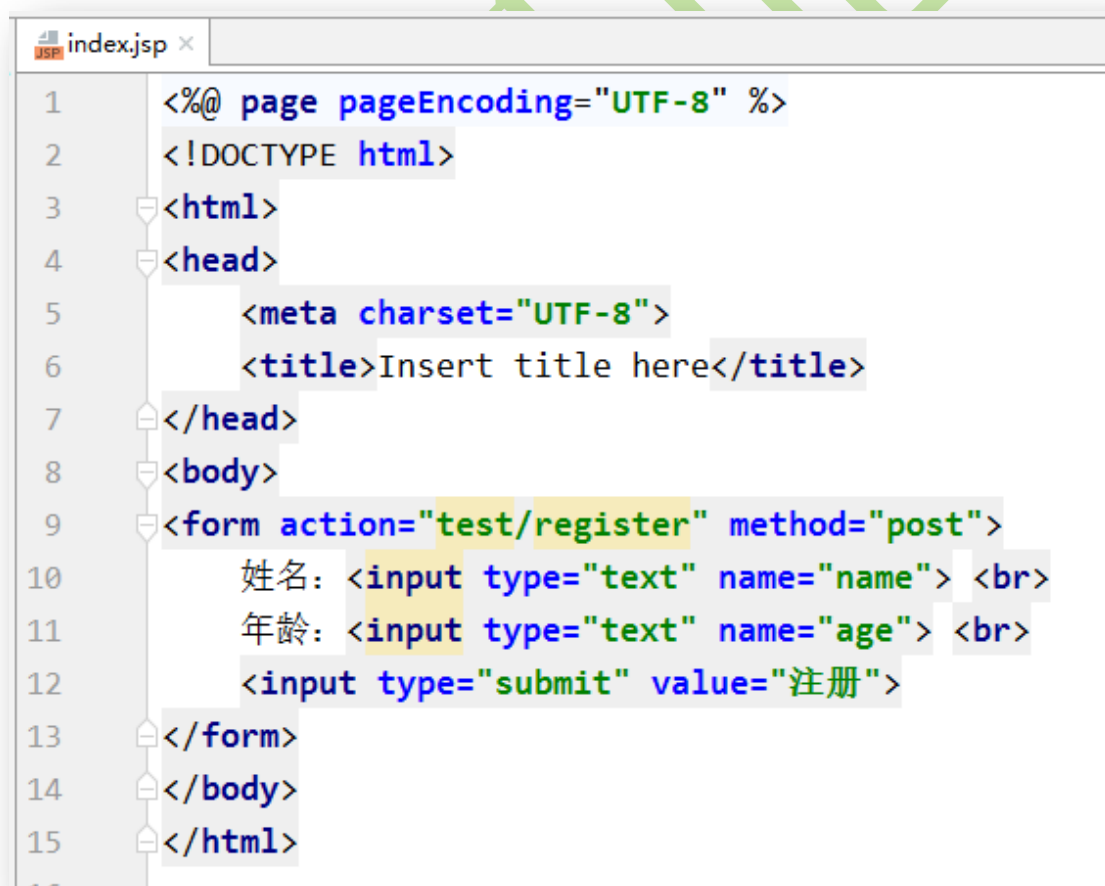


此时，便可在 webapp 中找到 jsp 的创建选项了。



B、创建 index 页面

在 src/main/webapp 下创建一个 html 文件，并命名为 index.html，创建完毕后再将其重命名为 index.jsp。因为 Idea 中是没有 JSP 页面模板的，不能直接创建 JSP 文件。



（4）启动后运行

此时启动工程后在浏览器直接访问，发现其并没有显示 index 页面。因为当前工程不能识别 jsp 文件。



2.5.2 使用物理视图

（1）添加 jasper 依赖

在 pom 中添加一个 Tomcat 内嵌的 jsp 引擎 jasper 依赖。jsp 引擎是用于解析 jsp 文件的，即将 jsp 文件解析为 Servlet 是由 jsp 引擎完成的。embed，嵌入。

```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

（2）注册资源目录

在 pom 文件中将 webapp 目录注册为资源目录。

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

  <resources>
    <resource>
      <directory>src/main/webapp</directory>
      <targetPath>META-INF/resources</targetPath>
      <includes>
        <include>**/*.*</include>
      </includes>
    </resource>
  </resources>
</build>
```

不过，我们一般会添加两个资源目录：

```
<resources>
  <!-- 注册dao包下mybatis映射文件为资源目录-->
  <resource>
    <directory>src/main/java</directory>
    <includes>
      <include>**/*.xml</include>
    </includes>
  </resource>
  <!-- 注册webapp目录为资源目录-->
  <resource>
    <directory>src/main/webapp</directory>
    <targetPath>META-INF/resources</targetPath>
    <includes>
      <include>**/*.*</include>
    </includes>
  </resource>
</resources>
```

(3) 创建 welcome.jsp

在 webapp 目录下再创建一个子目录 jsp，在其中创建 welcome.jsp 文件。

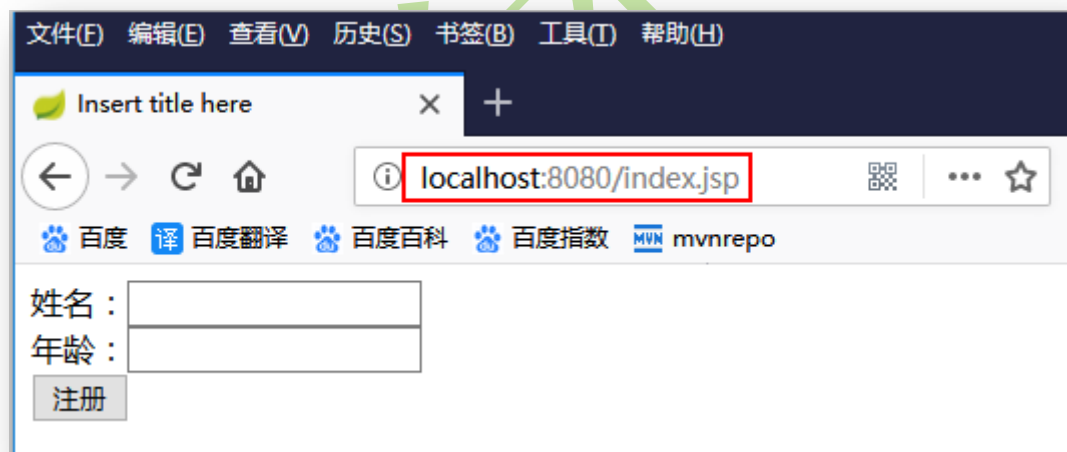
```
<body>
  name = ${pname } <br>
  age = ${page } <br>
</body>
```

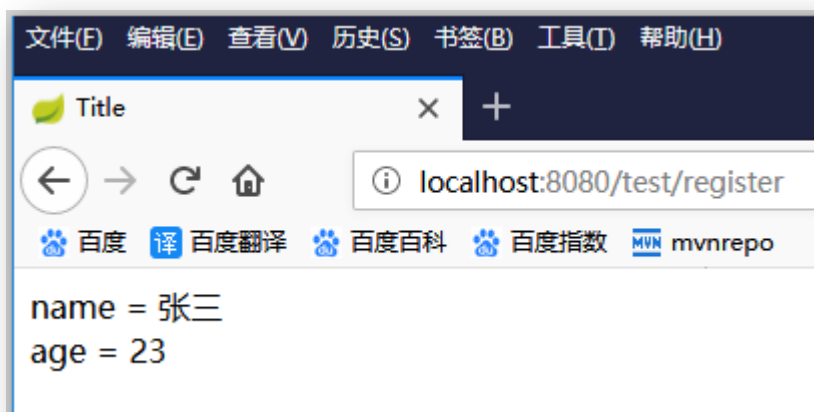
(4) 修改 SomeHandler 类

```
@Controller
@RequestMapping("/test")
public class SomeHandler {

    @RequestMapping("/register")
    public String registerHandle(String name, int age, Model model) {
        model.addAttribute("name", name);
        model.addAttribute("age", age);
        return "/jsp/welcome.jsp";
    }
}
```

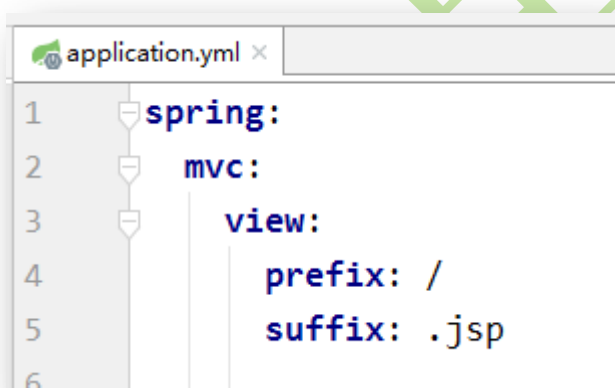
(5) 访问





2.5.3 使用逻辑视图

(1) 修改主配置文件



(2) 修改处理器

```
@Controller
@RequestMapping("/test")
public class SomeHandler {

    @RequestMapping("/register")
    public String registerHandle(String name, int age, Model model) {
        model.addAttribute("name", name);
        model.addAttribute("age", age);
        return "jsp/welcome";
    }
}
```

(3) 访问

执行效果与前面的相同。

2.6 Spring Boot 中使用 MyBatis

2.6.1 总步骤

- 导入依赖：MySQL 驱动、Druid 依赖、MyBatis 与 Spring Boot 整合依赖、Lombok 依赖
- 在 Service 接口实现类上添加@Service 注解
- 在 Dao 接口上添加@Mapper 注解
- 在 pom 中将 dao 目录注册为资源目录
- 在配置文件中注册映射文件、实体类别名，及数据源

2.6.2 需求

完成一个简单的注册功能。

2.6.3 定义工程

复制 06-jsp 工程，并重命名为 07-mybatis。

2.6.4 修改 pom 文件

导入三个依赖：mybatis 与 Spring Boot 整合依赖、mysql 驱动依赖与 Druid 数据源依赖。

```
<!--mybatis 与 spring boot 整合依赖-->
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>1.3.2</version>
</dependency>

<!--mysql 驱动-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.47</version>
</dependency>

<!-- druid 驱动 -->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.12</version>
</dependency>
```

2.6.5 修改 SomeHandler

```
@Controller
@RequestMapping("/test")
public class SomeHandler {

    @Autowired
    private ISomeService service;

    @RequestMapping("/register")
    public String registerHandle(Student student, Model model) {

        model.addAttribute("pname", student.getName());
        model.addAttribute("page", student.getAge());

        service.addStudent(student);

        // 逻辑视图名，其会与配置文件中的前缀与后缀自动拼接
        return "pages/welcome";
    }
}
```

2.6.6 定义 Service 接口及实现类

(1) 定义 Service 接口

```
public interface ISomeService {

    void addStudent(Student student);

}
```

(2) 定义 Service 实现类

```
@Service
public class SomeServiceImpl implements ISomeService {

    @Autowired
    private IStudentDao dao;

    @Override
    public void addStudent(Student student) {
        dao.insertStudent(student);
    }

}
```

2.6.7 定义实体类及 DB 表

(1) 定义实体类

```
public class Student {
    private Integer id;
    private String name;
    private int age;

    // getter and setter
    // toString()
}
```

(2) 定义 DB 表

在 DB 的 test 数据库中定义 student 表。

[表] student @test (localhost_3306)

文件(F) 编辑(E) 查看(V) 窗口(W)

导入向导(I) 导出向导(O) 筛选向导 网格

id	name	age
(Null)	(Null)	(Null)

2.6.8 定义 Dao 接口

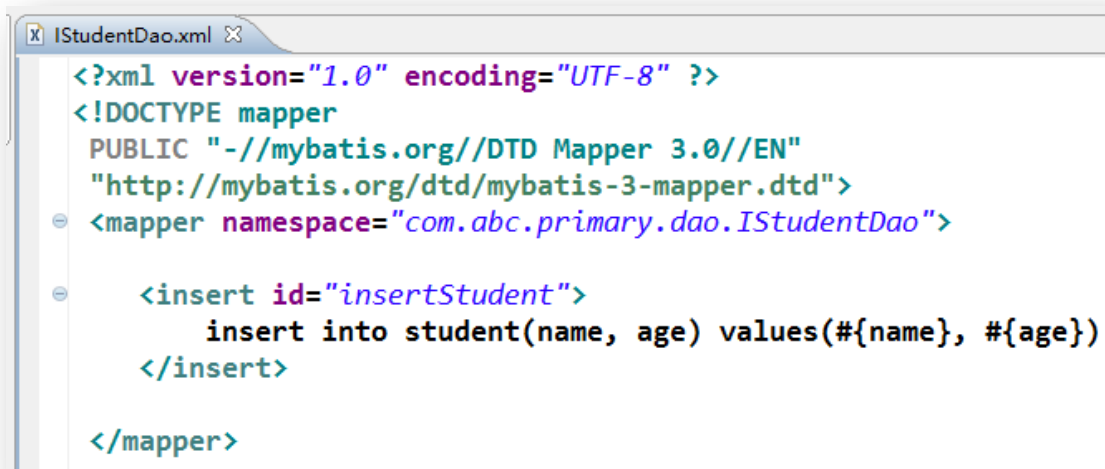
Dao 接口上要添加 @Mapper 注解。

```
@Mapper
public interface IStudentDao {

    void insertStudent(Student student);

}
```

2.6.9 定义映射文件



```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.abc.primary.dao.IStudentDao">

    <insert id="insertStudent">
        insert into student(name, age) values(#{name}, #{age})
    </insert>

</mapper>
```

2.6.10 注册资源目录

在 pom 文件中将 dao 目录注册为资源目录。

```
<resources>
  <!-- 注册dao包下mybatis映射文件为资源目录-->
  <resource>
    <directory>src/main/java</directory>
    <includes>
      <include>**/*.xml</include>
    </includes>
  </resource>
  <!-- 注册webapp目录为资源目录-->
  <resource>
    <directory>src/main/webapp</directory>
    <targetPath>META-INF/resources</targetPath>
    <includes>
      <include>**/*.*</include>
    </includes>
  </resource>
</resources>
```

2.6.11 修改主配置文件

在主配置文件中主要完成以下几件工作：

- 注册映射文件
- 注册实体类别名
- 注册数据源

```
application.properties
spring.mvc.view.prefix=/
spring.mvc.view.suffix=.jsp

mybatis.mapper-locations=classpath:com/abc/primary/dao/*.xml
mybatis.type-aliases-package=com.abc.primary.beans

spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql:///test
spring.datasource.username=root
spring.datasource.password=111

spring.datasource.url=jdbc:mysql:///test?useUnicode=true&characterEncoding=utf8
```

2.7 Spring Boot 的事务支持

若工程直接或间接依赖于 `spring-tx`，则框架会自动注入 `DataSourceTransactionManager` 事务管理器；若依赖于 `spring-boot-data-jpa`，则会自动注入 `JpaTransactionManager`。

2.7.1 定义工程

复制 07-mybatis 工程，并重命名为 08-transaction。

当前工程完成在对用户注册时一次插入到 DB 中两条注册信息。若在插入过程中有发生异常，则已完成插入的记录回滚。

2.7.2 修改启动类

```
@SpringBootApplication
@EnableTransactionManagement // 开启事务
public class Application {

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

2.7.3 修改 Service 实现类

```
@Service
public class StudentServiceImpl implements StudentService {
    @Autowired
    private StudentDao dao;

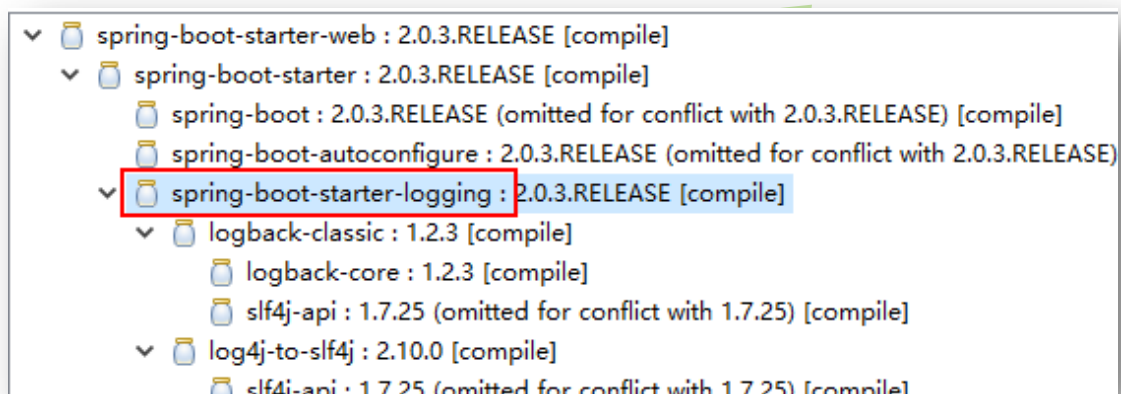
    // 采用Spring默认的事务提交方式：发生运行时异常回滚，发生受查异常提交
    @Transactional(rollbackFor = Exception.class)
    @Override
    public void addStudent(Student student) throws Exception {
        dao.insertStudent(student);
        // int i = 3 / 0;
        if (true) {
            throw new Exception("发生受查异常");
        }
        dao.insertStudent(student);
    }
}
```


2.8 Spring Boot 对日志的控制

2.8.1 logback 日志技术介绍

Spring Boot 中使用的日志技术为 logback。其与 Log4J 都出自同一人，性能要优于 Log4J，是 Log4J 的替代者。

在 Spring Boot 中若要使用 logback，则需要具有 spring-boot-starter-logging 依赖，而该依赖被 spring-boot-starter-web 所依赖，即不用直接导入 spring-boot-starter-logging 依赖。



2.8.2 spring boot 中使用 logback

在 Spring Boot 中使用 logback 日志，有两种方式。

(1) 添加配置属性

只需在核心配置文件中添加如下配置即可。

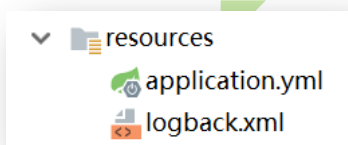
```
# Logback 日志控制
logging:
  # 指定日志显示的位置及格式
  pattern:
    console: logs-%level %msg%n

  level:
    root: warn      # 减少项目启动时的日志输出
    com.abc.dao: debug  # 显示指定dao包中类的执行日志
```

注意，在日志显示格式的属性值前面的 logs-是随意内容。在 yml 文件中的属性值若以 % 开头会报错，所以在 properties 文件中不存在该问题。

(2) 添加配置文件

该文件名为 logback.xml，且必须要放在 src/main/resources 类路径下。



内容如下：

```
logback.xml x
1  <?xml version="1.0" encoding="UTF-8"?>
2  <configuration>
3      <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
4          <encoder>
5              <pattern>%-5level - %msg%n</pattern>
6          </encoder>
7      </appender>
8
9      <root level="WARN">
10         <appender-ref ref="STDOUT" />
11     </root>
12
13     <logger name="com.abc.dao" level="DEBUG" />
14 </configuration>
```

2.9 Spring Boot 中使用 Redis

高并发下访问 Redis，存在什么问题？存在三个问题：

- 缓存穿透：为 DB 查询为 null 的数据预设一个值
- 缓存雪崩：提前规划好缓存到期时间
- 热点缓存：双重检测锁机制

2.9.1 定义工程

复制 08-transaction 工程，并重命名为 09-redisCache。

当前工程完成让用户在页面中输入要查询学生的 id，其首先会查看 Redis 缓存中是否存在，若存在，则直接从 Redis 中读取；若不存在，则先从 DB 中查询出来，然后再存放到 Redis 缓存中。但用户也可以通过页面注册学生，一旦有新的学生注册，则需要将缓存中的学生信息清空。根据 id 查询出的学生信息要求必须是实时性的，其适合使用注解方式的 Redis 缓存。

同时，通过页面还可以查看到总学生数，但对其要求是差不多就行，无需是实时性的。对于 Spring Boot 工程，其适合使用 API 方式的 Redis 缓存，该方式方便设置缓存的到期时限。

2.9.2 修改 pom 文件

在 pom 文件中添加 Spring Boot 与 Redis 整合依赖。

```
<!-- spring boot与redis整合依赖 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2.9.3 修改主配置文件

在主配置文件中添加如下内容：

```
# 连接单机Redis
spring.redis.host=osRedis
spring.redis.port=6379
spring.redis.password=111

# 连接Redis高可用集群
# spring.redis.sentinel.master=mymaster
# spring.redis.sentinel.nodes=sentinel1:26379,sentinel2:26379,sentinel3:26379

spring.cache.type=redis
spring.cache.cache-names=realTimeCache
```

2.9.4 修改实体类 Student

由于要将查询的实体类对象缓存到 Redis，Redis 要求实体类必须序列化。所以需要实体类实现序列化接口。

```
public class Student implements Serializable {
    private static final long serialVersionUID = 8998405589308132344L;

    private Integer id;
    private String name;
    private int age;

    // getter and setter
    // toString()
```

2.9.5 修改代码

(1) 修改 index 页面

```
<form action="test/register" method="post">
    姓名: <input type="text" name="name"> <br>
    年龄: <input type="text" name="age"> <br>
    <input type="submit" value="注册">
</form>

<hr>

<form action="test/find" method="post">
    学生id: <input type="text" name="id"> <br>
    <input type="submit" value="查询">
</form>

<hr>

<a href="test/studentsCount">查询学生总人数</a>
```

(2) 修改 SomeHandler 类

在其中添加两个处理器方法。

```
@RequestMapping("/find")
public @ResponseBody Student findHandle(int id) {
    return service.findStudentById(id);
}

@RequestMapping("/studentsCount")
public @ResponseBody Integer countHandle() {
    return service.findStudentsCount();
}
```

(3) 修改 Service 接口

在 Service 接口中添加一个业务方法。

```
public interface ISomeService {

    void addStudent(Student student);

    Student findStudentById(int id);
    Integer findStudentsCount();
}
```

(4) 修改 Service 接口实现类

```
@Service
public class SomeServiceImpl implements ISomeService {

    @Autowired
    private IStudentDao dao;

    @Autowired
    private RedisTemplate<Object, Object> redisTemplate;

    @CacheEvict(value="realTimeCache", allEntries=true)
    @Transactional
    @Override
    public void addStudent(Student student) {
        dao.insertStudent(student);
    }

    @Cacheable(value="realTimeCache", key="'student_'+#id")
    @Override
    public Student findStudentById(int id) {
        System.out.println("从DB中查询student");
        return dao.selectStudentById(id);
    }
}
```

```
// 使用双重检测锁解决热点缓存问题
@Override
public Integer findStudentsCount() {
    // 获取Redis操作对象
    BoundValueOperations<Object, Object> ops =
        redisTemplate.boundValueOps("count");
    // 从缓存中读取数据
    Object count = ops.get();
    if(count == null) {
        synchronized (this) {
            count = ops.get();
            if(count == null) {
                // 从DB中查询数据
                count = dao.selectStudentsCount();
                // 将数据写入到缓存，并设置到期时限
                ops.set(count, 10, TimeUnit.SECONDS);
            }
        }
    }
    return (Integer) count;
}
```

(5) 修改 Dao 接口

在其中添加两个方法。


```
@Mapper
public interface IStudentDao {

    void insertStudent(Student student);

    Student selectStudentById(int id);

    Integer selectStudentsCount();

}
```

(6) 修改映射文件

可以使用简单类名作为别名。

```
<insert id="insertStudent">
    insert into student(name, age) values("#{name}", #{age})
</insert>

<select id="selectStudentById" resultType="Student">
    select id,name,age from student where id=#{xxx}
</select>

<select id="selectStudentsCount" resultType="int">
    select count(id) from student
</select>
```

2.9.6 启动 Redis

```
redisos x
[root@redisos ~]# redis-server /usr/apps/redis-4.0.9/redis.conf
1422:C 20 Apr 19:11:25.097 # 0000000000000000 Redis is starting 000000
1422:C 20 Apr 19:11:25.097 # Redis version=4.0.9, bits=64, commit=0
1422:C 20 Apr 19:11:25.097 # Configuration loaded
[root@redisos ~]#
```

2.9.7 启动 Linux 主机

该示例需要启动三台 Redis 主机，三台 Redis 哨兵主机。

(1) 启动 Redis 集群

逐台启动三台 Redis 主机，即启动 Redis 集群。

```
redisos x redisos2 redisos3 sentinelos sentinelos2 sentinelos3
[root@redisos ~]# redis-server /usr/apps/redis-4.0.9/redis.conf
1400:C 20 Apr 18:32:29.336 # 0000000000000000 Redis is starting 000000
1400:C 20 Apr 18:32:29.336 # Redis version=4.0.9, bits=64, commit=0
1400:C 20 Apr 18:32:29.336 # Configuration loaded
[root@redisos ~]#
```

(2) 登录 Redis

逐台登录三台 Redis。

```
redisos x redisos2 redisos3 sentinelos sentinelos2 sentinelos3
[root@redisos ~]# redis-cli
127.0.0.1:6379>
```

(3) 查看 Redis 角色

逐台查看 Redis 的角色。

[illegible]

```
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:redisos
master_port:6379
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_repl_offset:462
```

```
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:redisos
master_port:6379
master_link_status:up
master_last_io_seconds_ago:0
master_sync_in_progress:0
slave_repl_offset:532
slave_priority:100
```

(4) 启动 sentinel 集群

逐台启动三台 Sentinel 主机，即启动 Sentinel 集群。

```
[root@sentinelos ~]# redis-sentinel /usr/apps/redis-4.0.9/sentinel.conf
1335:X 20 Apr 18:48:17.586 # 0000000000000000 Redis is starting 0000000000000000
1335:X 20 Apr 18:48:17.586 # Redis version=4.0.9, bits=64, commit=00000000, mo
1335:X 20 Apr 18:48:17.586 # Configuration loaded
1335:X 20 Apr 18:48:17.587 * Increased maximum number of open files to 10032 (

Redis 4.0.9 (00000000/0) 64 bit
Running in sentinel mode
Port: 26379
PID: 1335
```

2.10 Spring Boot 中使用 Dubbo

2.10.1 步骤

- 消费者与提供者工程均需要导入四个依赖
 - ◆ Dubbo 与 Spring Boot 整合依赖
 - ◆ zkClient 依赖
 - ◆ slf4j-log4j12 依赖
 - ◆ 自定义 commons 工程依赖
- 提供者工程
 - ◆ 将 Service 接口实现类的@Service 注解更换为阿里的注解，并添加@Component 注解
 - ◆ 在启动类上添加@EnableDubboConfiguration 与@EnableTransactionManager 注解
 - ◆ 修改配置文件：指定应用名称与注册中心地址
- 消费者工程
 - ◆ 将处理器中 Service 的声明上的@Autowired 注解更换为阿里的@Reference 注解
 - ◆ 在启动类上添加@EnableDubboConfiguration 注解
 - ◆ 修改配置文件：指定应用名称与注册中心地址

2.10.2 定义 commons 工程 11-dubboCommons

(1) 创建工程

这里就创建一个普通的 Maven 的 Java 工程，并命名为 11-dubboCommons。

(2) 定义 pom 文件

```
<groupId>com.abc</groupId>
<artifactId>11-dubboCommons</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.2</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

(3) 定义实体类

```
@Data
public class Employee implements Serializable {
    private Integer id;
    private String name;
    private int age;
}
```

（4）定义业务接口

```
public interface EmployeeService {  
    void addEmployee(Employee employee);  
    Employee findEmployeeById(int id);  
    Integer findEmployeeCount();  
}
```

（5）将工程安装到本地库

运行 Maven 的 install 命令，将工程安装到本地版本库，以备其它工程使用。

2.10.3 定义提供者 11-provider-springboot

（1）创建工程

创建一个 Spring Boot 工程，并重命名为 11-provider-springboot。

（2）定义 pom 文件

A、添加 dubbo 与 spring boot 整合依赖

B、添加 zkClient 依赖

C、其它依赖

- dubboCommons 依赖
- spring boot 与 redis 整合依赖
- mybatis 与 spring boot 整合依赖
- 数据源 Druid 依赖
- mysql 驱动依赖
- slf4j-log4j12 依赖
- spring-boot-starter-web 依赖

(3) 定义 Service 实现类

```
@Service // 阿里的注解
@Component
public class EmployeeServiceImpl implements EmployeeService {
    @Autowired
    private EmployeeDao dao;
    @Autowired
    private RedisTemplate<Object, Object> redisTemplate;

    // 插入操作会清空realTimeCache区域缓存
    @CacheEvict(value="realTimeCache", allEntries = true)
    // 指定发生受查异常回滚
    @Transactional(rollbackFor = Exception.class)
    @Override
    public void addEmployee(Employee employee) {
        dao.insertEmployee(employee);
    }

    // 将从DB中查询的数据存放到realTimeCache缓存区域，并为其指定
    // key为employee_id值
    @Cacheable(value="realTimeCache", key = "'employee_'+#id")
    @Override
    public Employee findEmployeeById(int id) {
        return dao.selectEmployeeById(id);
    }
}
```

```
// 使用双重检测锁解决热点缓存问题
@Override
public Integer findEmployeeCount() {
    // 获取Redis操作对象
    BoundValueOperations<Object, Object> ops =
        redisTemplate.boundValueOps("count");
    // 从缓存中读取数据
    Object count = ops.get();
    if(count == null) {
        synchronized (this) {
            count = ops.get();
            if(count == null) {
                // 从DB中查询
                count = dao.selectEmployeeCount();
                // 将查询的数据写入到Redis缓存, 并设置到期时限
                ops.set(count, 10, TimeUnit.SECONDS);
            }
        }
    }
    return (Integer) count;
}
```

(4) 定义 Dao 接口

```
@Mapper
public interface EmployeeDao {
    void insertEmployee(Employee employee);
    Integer selectEmployeeCount();
    Employee selectEmployeeById(int id);
}
```

(5) 定义映射文件

```
<mapper namespace="com.abc.dao.EmployeeDao">
```



```
<insert id="insertEmployee">
    insert into employee(name, age) values(#{name}, #{age})
</insert>
<select id="selectEmployeeCount" resultType="int">
    select count(id) from employee
</select>
<select id="selectEmployeeById" resultType="Employee">
    select id, name, age from employee where id=#{xxx}
</select>
</mapper>
```

(6) 修改启动类

在启动类上必须要添加@EnableDubboConfiguration注解，开启Dubbo的自动配置功能。

```
@EnableTransactionManagement    // 开启事务
@SpringBootApplication
@EnableCaching                  // 开启缓存
@EnableDubboConfiguration       // 开启Dubbo自动配置
public class ProviderApplication {

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

(7) 修改主配置文件

```
server:
  port: 8888

mybatis:
  # 注册mybatis中实体类的别名
  type-aliases-package: com.abc.bean
  # 注册映射文件
```

mapper-locations: classpath:com/abc/dao/*.xml

spring:

注册数据源

datasource:

指定数据源类型为Druid

type: com.alibaba.druid.pool.DruidDataSource

driver-class-name: com.mysql.jdbc.Driver

url: jdbc:mysql:///test?useUnicode=true&characterEncoding=utf8

username: root

password: 111

连接Redis 服务器

redis:

host: redis50S

port: 6379

password: 111

连接Redis 高可用集群

redis:

sentinel:

master: mymaster

nodes:

- sentinel0S1:26379

- sentinel0S2:26379

- sentinel0S3:26379

配置缓存

cache:

type: redis # 指定缓存类型

cache-names: realTimeCache # 指定缓存区域名称

功能等价于 spring-dubbo 配置文件中的<dubbo:application/>

该名称是由服务治理平台使用

application:

name: 11-provider-springboot

指定 zk 注册中心

dubbo:

registry: zookeeper://zk0S:2181

zk 集群作注册中心

registry: zookeeper://zk0S1:2181?backup=zk0S2:2181,zk0S3:2181

2.10.4 定义消费者 11-consumer-springboot

(1) 创建工程

创建一个 Spring Boot 工程，并重命名为 11-consumer-springboot。

(2) 定义 pom 文件

- dubbo 与 spring boot 整合依赖
- zkClient 依赖
- dubboCommons 依赖
- JSP 引擎 jasper 依赖
- slf4j-log4j12 依赖
- spring-boot-starter-web 依赖

(3) 修改主配置文件

spring:

功能等价于 spring-dubbo 配置文件中的<dubbo:application/>

application:

name: 11-consumer-springboot

指定 zk 注册中心

dubbo:

registry: zookeeper://zk0S:2181

zk 集群作注册中心

registry: zookeeper://zk0S1:2181?backup=zk0S2:2181,zk0S3:2181

(4) 创建 index.jsp 页面

在 src/main/webapp 目录下定义 index.jsp 文件。

```
<body>
  <form action="/consumer/employee/register" method="post">
    姓名: <input type="text" name="name"/> <br>
    年龄: <input type="text" name="age"/> <br>
    <input type="submit" value="注册"/>
  </form>
</body>
```

(5) 定义处理器

```
@Controller
@RequestMapping("/consumer/employee")
public class SomeController {

    // @Autowired
    @Reference
    private EmployeeService employeeService;

    @PostMapping("/register")
    public String someHandle(Employee employee, Model model) {
        employeeService.addEmployee(employee);
        model.addAttribute("employee", employee);
        return "/welcome.jsp";
    }
}
```

```

    @RequestMapping("/find/{id}")
    @ResponseBody
    public Employee findHandle(@PathVariable("id") int id) {
        return employeeService.findEmployeeById(id);
    }

    @RequestMapping("/count")
    @ResponseBody
    public Integer countHandle() {
        return employeeService.findEmployeeCount();
    }
}

```

(6) 定义 welcome.jsp 页面

在 src/main/webapp 目录下定义 welcome.jsp 文件。

```

<body>
    welcome you 【 ${employee} 】 <br>
</body>

```

(7) 修改入口类

```
@SpringBootApplication
@EnableDubboConfiguration
public class ConsumerApplication {

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

2.11 Spring Boot 下使用拦截器

在非 Spring Boot 工程中若要使用 SpringMVC 的拦截器，在定义好拦截器后，需要在 Spring 配置文件中对其进行注册。但 Spring Boot 工程中没有了 Spring 配置文件，那么如何使用拦截器呢？

Spring Boot 对于原来在配置文件配置的内容，现在全部体现在一个类中，该类需要继承自 `WebMvcConfigurationSupport` 类，并使用 `@Configuration` 进行注解，表示该类为一个 `JavaConfig` 类，其充当配置文件的角色。

2.11.1 定义工程

复制 04-customConfig 工程，并重命名为 05-interceptor。

2.11.2 定义拦截器

```
public class SomeInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {

        System.out.println("执行拦截器" + request.getRequestURI());

        return true;
    }
}
```

2.11.3 定义处理器

```
@Controller
public class SomeHandler {

    @RequestMapping("/first/some")
    public @ResponseBody String someHandle() {
        return "/first/some";
    }

    @RequestMapping("/second/other")
    public @ResponseBody String otherHandle() {
        return "/second/other";
    }
}
```

2.11.4 定义配置文件类

```
@Configuration // 表示该文件充当配置文件
public class MyWebMvcConfiguration extends WebMvcConfigurationSupport {
    @Override
    protected void addInterceptors(InterceptorRegistry registry) {
        SomeInterceptor si = new SomeInterceptor();
        registry.addInterceptor(si)
            .addPathPatterns("/first/**") // 拦截first开头的路径
            .excludePathPatterns("/second/**"); // 不拦截second开头的路径
    }
}
```

2.12 Spring Boot 中使用 Servlet

在 Spring Boot 中使用 Servlet，根据 Servlet 注册方式的不同，有两种使用方式。若使用的是 Servlet3.0+版本，则两种方式均可使用；若使用的是 Servlet2.5 版本，则只能使用配置类方式。

2.12.1 注解方式

(1) 创建工程

创建一个 Spring Boot 工程，并命名为 11-servlet01。

(2) 创建 Servlet

```
@WebServlet("/some")
public class SomeServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello Spring Boot Servlet");
    }
}
```


(3) 修改入口类

在入口类中添加 Servlet 扫描注解。

```
@SpringBootApplication
@ServletComponentScan("com.abc.boot.servlets") // 开启Servlet扫描
public class Application {

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

2.12.2 配置类方式

(1) 创建工程

创建 spring boot 工程，并命名为 11-servlet02。

(2) 定义 Servlet

```
public class SomeServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello Spring Boot Servlet");
    }
}
```

(3) 定义配置类

```
@Configuration // 表示其为配置类，相当于applicationContext.xml文件
public class MyApplicationContext {

    @Bean // 表示该方法返回的对象即为Spring容器中的Bean，方法名随意
    public ServletRegistrationBean<SomeServlet> getServletBean() {
        // 创建Servlet
        SomeServlet servlet = new SomeServlet();
        // 注册Servlet
        return new ServletRegistrationBean<SomeServlet>(servlet, "/some");
    }
}
```

2.13 Spring Boot 中使用 Filter

在 Spring Boot 中使用 Filter 与前面的使用 Servlet 相似，根据 Filter 注册方式的不同，有两种使用方式。若使用的是 Servlet3.0+版本，则两种方式均可使用；若使用的是 Servlet2.5 版本，则只能使用配置类方式。

2.13.1 注解方式

(1) 使用工程

直接在 11-servlet01 工程上进行修改，不再创建新的工程。

(2) 创建 Filter

```
@WebFilter("/")
public class SomeFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        System.out.println("信息已被过滤");
        chain.doFilter(request, response);
    }

    @Override
    public void destroy() {
    }
}
```

(3) 修改入口类

在@ServletComponentScan 注解中注册 Filter 所在的包，当然，Spring Boot 支持通配符的使用。

```
@SpringBootApplication
@ServletComponentScan("com.abc.boot.*") // 开启Servlet扫描
public class Application {

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

2.13.2 配置方式

(1) 使用工程

直接在 11-servlet02 工程上进行修改，不再创建新的工程。

(2) 定义 Filter

```
public class SomeFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        System.out.println("信息已被过滤");
        chain.doFilter(request, response);
    }

    @Override
    public void destroy() {
    }

}
```

(3) 修改配置类

在配置类中添加如下方法。

```
@Bean
public FilterRegistrationBean<SomeFilter> getFilterBean() {
    // 创建Filter
    SomeFilter filter = new SomeFilter();
    // 创建注册对象
    FilterRegistrationBean<SomeFilter> registration = new FilterRegistrationBean<>(filter);
    // 添加过滤条件
    registration.addUrlPatterns("/");
    return registration;
}
```

第3章 模板引擎 Thymeleaf

3.1 Thymeleaf 简介

Thymeleaf[taɪm li:f], 百里香叶, 是一个流行的模板引擎, 该模板引擎采用 Java 语言开发。Java 中常见的模板引擎有 Velocity、Freemaker、Thymeleaf 等。不同的模板引擎都会具有自己的特定的标签体系, 而 Thymeleaf 以 HTML 标签为载体, 在 HTML 的标签下实现对数据的展示。

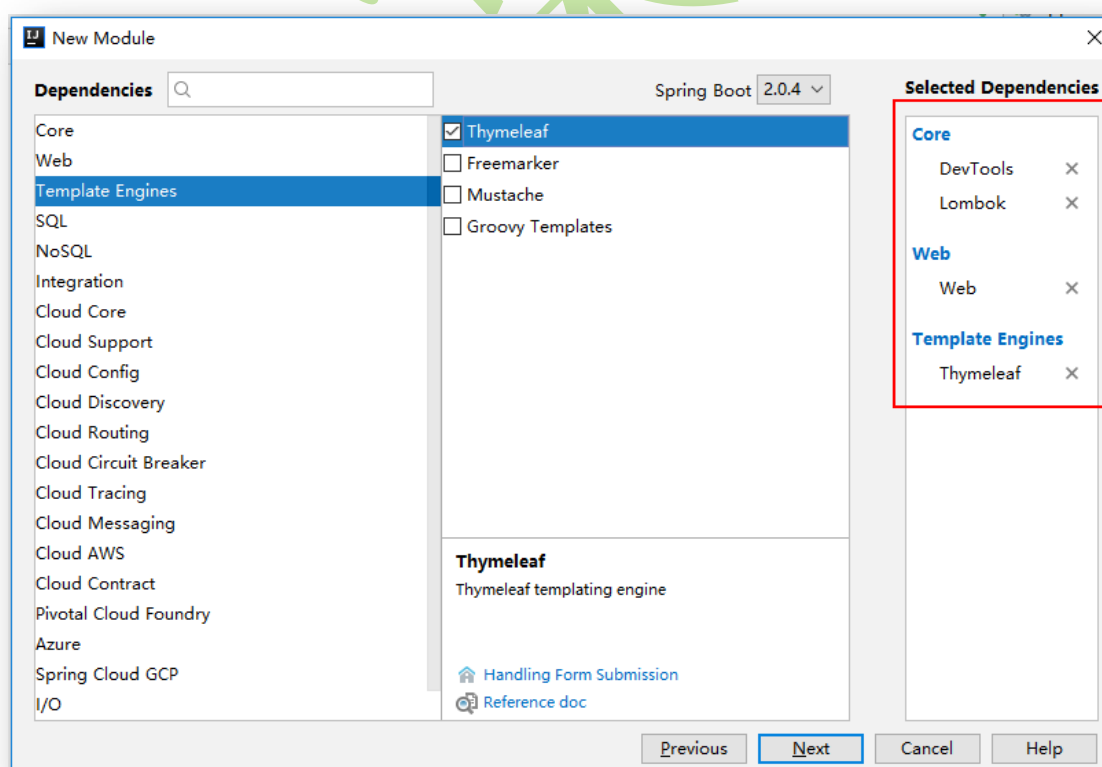
Thymeleaf 本身与 SpringBoot 是没有关系的, 但 SpringBoot 官方推荐使用 Thymeleaf 作为前端页面的数据展示技术, SpringBoot 很好地集成了这种模板技术。

Thymeleaf 的官网为: <http://www.thymeleaf.org>

3.2 Spring Boot 集成 Thymeleaf

3.2.1 创建工程

创建一个 Spring Boot 工程, 命名为 thymeleaf, 并在创建工程时导入如下依赖。



3.2.2 定义配置文件

```
application.properties x
1 # 开发阶段，建议关闭thymeleaf缓存，否则可能会出现数据未更新情况
2 spring.thymeleaf.cache=false
```

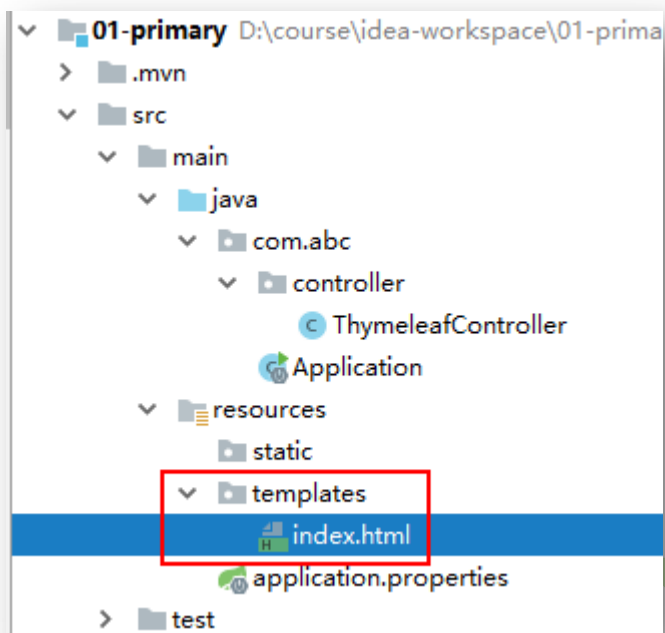
3.2.3 定义处理器

```
@Controller
public class ThymeleafController {

    @RequestMapping("/test/myindex")
    public String indexHandle(Model model) {
        model.addAttribute("welcome", "Hello Thymeleaf World");
        // 这里的index是Thymeleaf页面文件index.html，但不用写html后缀
        return "index";
    }
}
```

3.2.4 定义 index.html 页面

在 src/main/resources/templates 目录下定义 index.html 页面。



在页面的<html>标签中需要添加 Thymeleaf 的命名空间属性:

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



3.3 Thymeleaf 标准表达式

常用的 Thymeleaf 标准表达式有三种。标准表达式都是用于获取代码中存放到 Model

中的属性值的，只不过获取方式不同而已。

以下举例均在前面的 thymeleaf 工程基础上直接修改，无需再创建新的 Module。

3.3.1 变量表达式\${...}

使用\${...}括起来的表达式，称为变量表达式。该表达式的内容会显示在 HTML 标签文本处。

该表达式一般都是通过 th:text 标签属性进行展示的。

(1) 修改处理器类

```
@Controller
public class ThymeleafController {

    @RequestMapping("/test/myindex")
    public String indexHandle(Model model) {
        model.addAttribute("welcome", "Hello Thymeleaf World");

        Student student = new Student("张三", 23);
        model.addAttribute("student", student);

        // 这里的index是Thymeleaf页面文件index.html，但不用写html后缀
        return "index";
    }
}
```


(2) 创建 VO 类

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Student {
    private String name;
    private int age;
}
```

(3) 修改 index 页面

直接在页面中添加如下内容：

```
<body>
  <p th:text="${welcome}">这里会显示动态数据</p>
  <div th:text="${welcome}">这里会显示动态数据</div>
  <span th:text="${welcome}">这里会显示动态数据</span>
  <hr>
  <div th:text="${student}">这里会显示动态数据</div>
  <div th:text="${student.name}">这里会显示动态数据</div>
  <div th:text="${student.age}">这里会显示动态数据</div>
</body>
</html>
```

(4) 测试效果

```
Hello Thymeleaf World
Hello Thymeleaf World
Hello Thymeleaf World


---


Student(name=张三, age=23)
张三
23
```

3.3.2 选择表达式*{...}

选择表达式，也称为星号表达式，其是使用*{...}括起来的表达式。一般用于展示对象的属性。该表达式的内容会显示在HTML标签体文本处。但其需要与th:object标签属性联用，先使用th:object标签选择了对象，再使用*{...}选择要展示的对象属性。该表达式可以有效降低页面中代码的冗余。

不过，其也可以不与th:object标签联用，在*{...}中直接使用“对象.属性”方式，这种写法与变量表达式相同。

该表达式一般都是通过th:text标签属性进行展示的。

(1) 修改 index 页面

直接在页面中添加如下内容：

```
<hr>
<div th:object="${student}">
  <p>姓名: <span th:text="*{name}"></span> </p>
  <p>年龄: <span th:text="*{age}"></span> </p>
</div>

<hr>
<p>姓名: <span th:text="*{student.name}"></span> </p>
<p>年龄: <span th:text="*{student.age}"></span> </p>
```

(2) 测试效果

```
Hello Thymeleaf World
Hello Thymeleaf World
Hello Thymeleaf World


---


Student(name=张三, age=23)
张三
23


---


姓名：张三
年龄：23


---


姓名：张三
年龄：23
```

3.3.3 URL 表达式@{...}

使用@{...}括起来，并且其中只能写一个绝对 URL 或相对 URL 地址的表达式，称为 URL 表达式。这个绝对/相对 URL 地址中一般是包含有动态参数的，需要结合变量表达式\${...}进行字符串拼接。

@{...}中的 URL 地址具有三种写法。为了演示这三种写法的区别，先为当前工程添加一个上下文路径，然后直接在 index.html 文件中修改。

```
application.properties x
1  # 当前工程的上下文路径
2  server.servlet.context-path=/xxx
3
4  # 开发阶段，建议关闭thymeleaf缓存，否则可能会出现数据未更新情况
5  spring.thymeleaf.cache=false
6
```

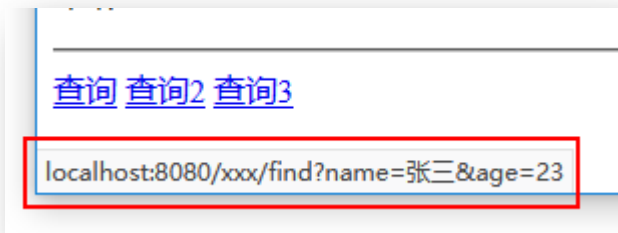
(1) 以 http 协议开头的绝对地址

在进行字符串拼接时使用加号(+)连接，容易出错。但使用双竖线则无需字符串拼接，简单易读。但是，Idea 会对其中的问号(?)报错，不过其不影响运行。

```
<hr>
<!-- 使用+进行字符串拼接，容易出错-->
<a th:href="@{'http://localhost:8080/xxx/find?name=' + ${student.name}}">查询</a>
<!-- 使用双竖线无需字符串拼接，简单易读-->
<a th:href="@{|http://localhost:8080/xxx/find?name=${student.name}|}">查询2</a>
<a th:href="@{|http://localhost:8080/xxx/find?name=${student.name}&age=${student.age}|}">查询3</a>
```

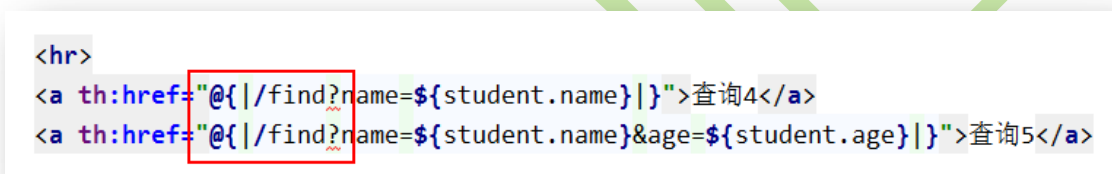
在页面通过查看源码可以看到其解析结果。当然，对于 and 符(&)Thymeleaf 会将其解析为实体形式(&#amp;#38;#38;#38;)，但浏览器会对(&#amp;#38;#38;#38;)进行正确解析。

```
<hr>
<a href="http://localhost:8080/xxx/find?name=张三">查询</a>
<a href="http://localhost:8080/xxx/find?name=张三">查询2</a>
<a href="http://localhost:8080/xxx/find?name=张三&#38;#38;#38;age=23">查询3</a>
```

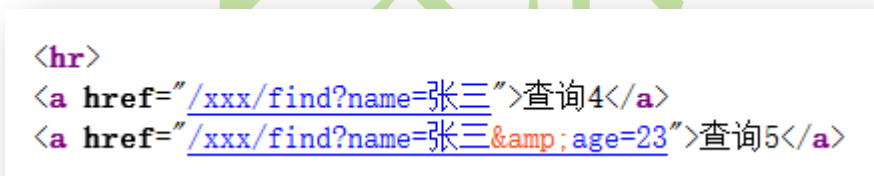


(2) 以/开头的相对地址

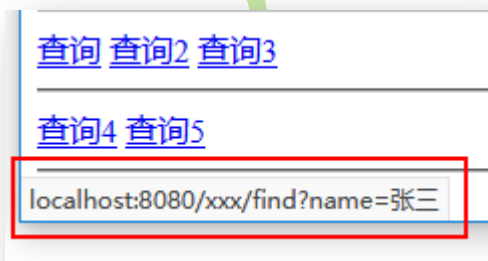
在 URL 表达式中，Thymeleaf 会将开头的斜杠(/)解析为当前工程的上下文路径 ContextPath，而浏览器会自动为其添加 “http://主机名:端口号”，即其即为一个绝对路径。



在页面通过查看源码可以看到其解析结果中已经添加了上下文路径。



而在页面则可以看到浏览器对其解析的结果已经添加了 <http://localhost:8080>。



(3) 不以/开头的相对地址

```
<hr>  
<a th:href="@{|find?name=${student.name}|}">查询6</a>
```

在页面通过查看源码可以看到其解析结果中并未添加任何东西，即没有上下文路径。也就是说，其是相对于当前请求路径的一个相对地址。

```
<hr>  
<a href="find?name=张三">查询6</a>
```

而在页面则可以看到浏览器对其解析的结果已经添加了 <http://localhost:8080/xxx/test>，这是相对于当前请求路径的一个相对地址的转换结果。

查询6

localhost:8080/xxx/test/find?name=张三

3.4 Thymeleaf 常见属性

3.4.1 逻辑运算相关属性

(1) th:if

该属性用于逻辑判断，类似于 JSTL 中的 `<c:if/>`。

A、修改处理器

在处理器中添加如下语句。

```
model.addAttribute("gender", "male");
```

B、修改 index 页面

在 index.html 文件中添加如下语句。

```
<hr>
<p th:if="${gender} == 'male'">
    男
</p>
```

C、效果

男

(2) th:switch/th:case

该属性用于多分支判断，类似于 Java 中的 Switch-Case 语句。

A、修改处理器

在处理器中添加如下语句。

```
model.addAttribute("age", 35);
```

B、修改 index 页面

在 index.html 文件中添加如下语句。

一旦某个 case 与 switch 的值相匹配了，剩余的 case 则不再比较。th:case="*" 表示默认 case，前面的 case 都不匹配时候执行该 case。

```
<hr>
<div th:switch="${age/10}">
  <p th:case="0">儿童</p>
  <p th:case="1">青少年</p>
  <p th:case="2">青年</p>
  <p th:case="3">中青年</p>
  <p th:case="4">中年</p>
  <p th:case="5">中老年</p>
  <p th:case="*">老年</p>
</div>
```

(3) th:each

该属性用于遍历数组、List、Set、Map，类似于 JSTL 中的<c:forEach/>。

A、遍历 List

遍历数组、Set 与遍历 List 方式是相同的。

a、修改处理器

在 Controller 中添加如下代码。


```
Student student1 = new Student("张三", 23);
Student student2 = new Student("李四", 24);
Student student3 = new Student("王五", 25);

// 创建List集合
List<Student> students = new ArrayList<>();
students.add(student1);
students.add(student2);
students.add(student3);

model.addAttribute("students", students);
```

b、修改 index 页面

前面的 stu 为当前遍历对象，而\${students}为遍历的集合。

```
<hr>
<p th:each="stu : ${students}">
    <span th:text="${stu.name}"></span>
    <span th:text="${stu.age}"></span>
</p>
```

c、效果

张三 23

李四 24

王五 25

B、遍历状态对象

```
<hr>
<p th:each="stu, xxx : ${students}">
    <!-- 声明状态对象为xxx -->
    <span th:text="${xxx.count}"></span>
    <span th:text="${stu.name}"></span>
    <span th:text="${stu.age}"></span>
</p>

<hr>
<p th:each="stu : ${students}">
    <!-- 状态对象为当前遍历对象加stat后缀 -->
    <span th:text="${stuStat.index}"></span>
    <span th:text="${stu.name}"></span>
    <span th:text="${stu.age}"></span>
</p>
```

a、效果

1 张三 23

2 李四 24

3 王五 25

0 张三 23

1 李四 24

2 王五 25

C、遍历 Map

需要清楚，Map 的键值对是一个 Map.Entry 对象。

a、修改处理器

```
// 创建Map
Map<String, Student> stuMap = new HashMap<>();
stuMap.put("stu1", student1);
stuMap.put("stu2", student2);
stuMap.put("stu3", student3);

model.addAttribute("stuMap", stuMap);
```

b、修改 index 页面

Idea 对这个遍历对象的 key、value 属性会报错，但不影响运行。

```
<hr>
<p th:each="entry : ${stuMap}">
    <!-- 状态对象为当前遍历对象加stat后缀 -->
    <span th:text="${entryStat.count}"></span>
    <span th:text="${entry.key}"></span>
    <span th:text="${entry.value}"></span>
    <span th:text="${entry.value.name}"></span>
    <span th:text="${entry.value.age}"></span>
</p>
```

c、效果

```
1 stu2 Student(name=李四, age=24) 李四 24
2 stu3 Student(name=王五, age=25) 王五 25
3 stu1 Student(name=张三, age=23) 张三 23
```

3.4.2 html 标签相关

(1) th:text/th:utext

这两个属性均用于在标签体中显示动态文本。但不同的是，th:utext 会解析文本中的 HTML 标签，而 th:text 则是原样显示。

A、修改处理器

```
model.addAttribute("welcome", "<h2>Thymeleaf,<br/>I'm learning.</h2>");
```

B、修改 index 页面

```
<hr>
<div th:text="${welcome}"></div>
<hr>
<div th:utext="${welcome}"></div>
```

C、效果

```
<h2>Thymeleaf,<br/>I'm learning.</h2>
```

Thymeleaf,
I'm learning.

(2) th:name/th:value

该属性用于获取标签动态 name 属性值，及标签的默认 value 值。

A、修改处理器

```
model.addAttribute("attrName", "age");
```

B、修改 index 页面

```
<input type="text" th:name="${attrName}" th:value="${student.age}">
```

C、效果

在页面查看源码可以看到如下效果。

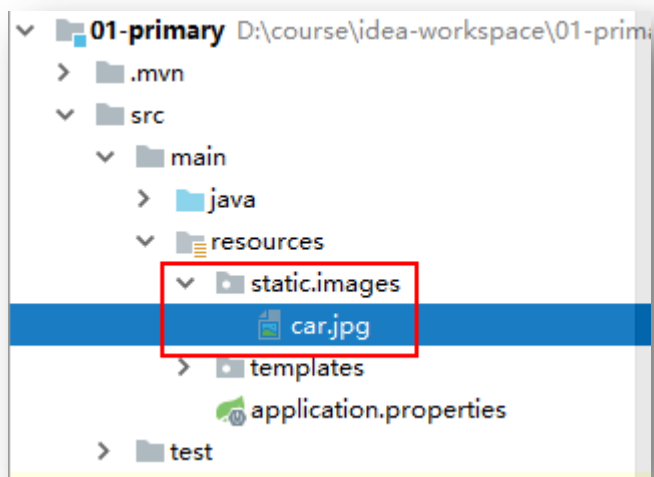
```
<input type="text" name="age" value="23">
```

(3) URL 路径相关

th:action、th:src、th:href，这三个都是与 URL 路径相关的属性。若这些 URL 中包含有动态参数，则它们的值需要 URL 表达式@{...}与变量表达式\${...}配合使用。下面以标签中的 th:src 为例演示用法。

A、创建目录放图片

在工程的 src/main/resources/static 目录下创建一个 Directory，命名为 images，并在其中放入一张图片 car.jpg。



B、修改处理器

```
model.addAttribute("photo", "car.jpg");
```

C、修改 index

```

```

3.4.3 css/js 相关

(1) th:id/th:name

这两个属性可以获取标签的动态 id 与 name 属性，以便在 js 中使用。

(2) th:style

该属性用于获取标签的 css 动态样式。

(3) th:onclick

该属性用于获取标签的单击事件所触发的动态事件，即单击事件所触发的方法名。这些 js 事件属性很多，都是以 th:on 开头。

th:onbeforeprint	th:onbeforeunload	th:onblur
th:oncanplay	th:oncanplaythrough	th:onChange
th:onclick	th:oncontextmenu	th:ondblclick
th:ondrag	th:ondragend	th:ondragenter
th:ondragleave	th:ondragover	th:ondragstart
th:ondrop	th:ondurationchange	th:onemptied
th:onended	th:onerror	th:onfocus
th:onformchange	th:onforminput	th:onhashchange
th:oninput	th:oninvalid	th:onkeydown
th:onkeypress	th:onkeyup	th:onload
th:onloadeddata	th:onloadedmetadata	th:onloadstart
th:onmessage	th:onmousedown	th:onmousemove

3.4.4 内联属性 th:inline

其应用场景是，在 HTML 某标签显示文本内部存在部分数据来自于动态数据，或 JS 内部需要使用动态数据的场景。在该场景中，可以使用`[[${...}]]`或`[( ${...}) ]`的方式将动态数据嵌入到指定的文本或脚本内部。

`th:inline` 的取值有四个：`text`、`javascript`、`css` 和 `non`。分别表示内联文本、内联脚本、内联 `css`，与禁止内联，即不解析`[[${...}]]`。不过，`th:inline="text"`可以省略不写。

(1) 内联文本

```
<hr>
```

```
<!-- 该方式会正确显示 -->
```

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

```
    他的名字是: [[${student.name}]]
```

```
</p>
```

```
<hr>
```

```
<!-- 该方式会正确显示: th:inline="text"可省略 -->
```

```
<p>
```

```
    我的名字也是: [[${student.name}]]
```

```
</p>
```

运行效果是:

张三

他的名字是 : 张三

我的名字也是 : 张三

(2) 内联脚本

```
<!--th:inline="javascript"不能省略-->
<script th:inline="javascript" type="text/javascript">
    // 无法获取
    // alert(${student.name});
    // 正解显示
    alert([[${student.name}]]);
</script>
```

(3) 内联 CSS

A、修改 index 页面

先在 index 页面中添加一个<div>，要使用内联 CSS 将其背景色设置为红色。

```
<div id="reddiv">
    我的背景色要变为红色
</div>
```

B、修改处理器

```
model.addAttribute("elementId", "reddiv");
model.addAttribute("bgColor", "red");
```

C、再修改 index 页面

瑞在 index 页面中添加如下代码，使用内联 CSS 定义<div>的样式。此内联的写法 Idea 不识别，但不影响运行。

```
<div id="reddiv">
    我的背景色要变为红色
</div>

<!-- 使背景色为红色 -->
<style th:inline="css">
    #[[${elementId}]] {
        width:100px;
        height:100px;
        background: [[${bgColor}]];
    }
</style>
```

3.4.5 万能属性 th:attr

该标签之所以称为万能属性，其主要具有两个应用场景，这两个场景在官方文档中均有详细讲解。

(1) 为无定义的属性赋予动态参数

很多 HTML 标签的属性都有其对应的 Thymeleaf 的 th 命名空间中属性，但并不是所有的都存在对应属性。若没有其对应的 th 命名空属性，但又想让其获取动态值，就需要使用该属性了。

5.1 标题为：为任意属性设置值。

- 4.13 The No-Operation token
- 4.14 Data Conversion / Formatting
- 4.15 Preprocessing
- 5 Setting Attribute Values
 - 5.1 Setting the value of any attribute**
 - 5.2 Setting value to specific attributes
 - 5.3 Setting more than one value at a time
 - 5.4 Appending and prepending
 - 5.5 Fixed-value boolean attributes
 - 5.6 Setting the value of any attribute (default attribute processor)
 - 5.7 Support for HTML5-friendly attribute and element names
- 6 Iteration
 - 6.1 Iteration basics
 - Using th:each

5.1 Setting the value of any attribute

Say our website publishes a newsletter, and we want our users to be able to subscribe. We'll create a `src/main/resources/templates/subscribe.html` template with a form:

```
<form action="subscribe.html">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe!" />
  </fieldset>
</form>
```

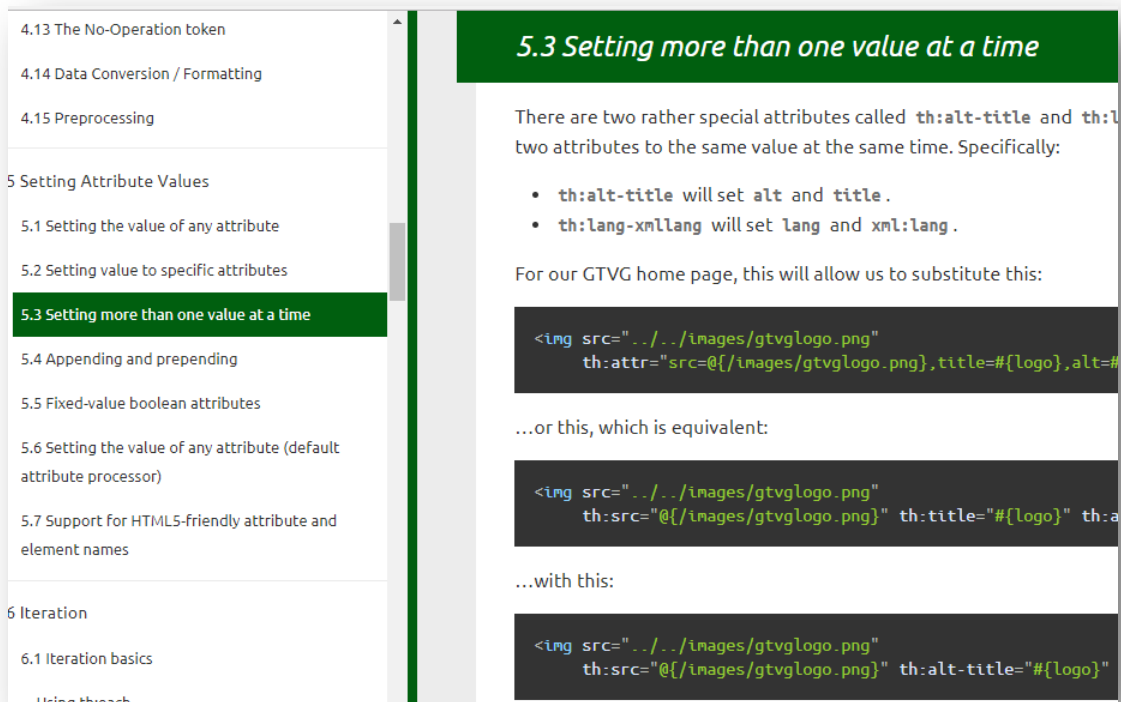
As with Thymeleaf, this template starts off more like a static prototype for an application. First, the `action` attribute in our form statically links to the place for useful URL rewriting. Second, the `value` attribute in the submit button but we'd like it to be internationalized.

Enter then the `th:attr` attribute, and its ability to change the value of

```
<form action="subscribe.html" th:attr="action=@{/subscribe}">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe!" th:attr="value=#{subscribe}" />
  </fieldset>
</form>
```

(2) 一次为多个属性赋予动态参数

若想一次为多个属性赋予动态参数，则可以使用该属性。



3.5 Thymeleaf 运算基础

3.5.1 Thymeleaf 字面量

字面量，即字面常量。Thymeleaf 中有四种字面量：文本、数字、布尔，及 null。

(1) 文本字面量

由单引号括起来的字符串，称为文本字面量。

```
<!-- 文本字面量 -->
<div>
    我爱你, <span th:text="'北京'"></span>
</div>
```

(2) 数字字面量

数字字面量就是数字，可以是整数，也可以是小数。

```
<!-- 数字字面量 -->
<div>
  3.14 + 6 = <span th:text="3.14 + 6"></span>
</div>
```

(3) 布尔字面量

布尔字面量就是 true、false，也可以写为 TRUE、FALSE、True、False。

A、修改处理器

```
model.addAttribute("isClose", false);
```

B、修改 index 页面

```
<!-- 布尔字面量 -->
<div>
    <span th:if="${isClose} == false">欢迎光临</span>
    <span th:if="${isClose} == true">关门歇业</span>
    <span th:if="${isClose} == FALSE">欢迎光临</span>
    <span th:if="${isClose} == TRUE">关门歇业</span>
    <span th:if="${isClose} == False">欢迎光临</span>
    <span th:if="${isClose} == True">关门歇业</span>
</div>
```

(4) null 字面量

表示空，其一般用于判断。若对象未定义，或对象已定义但其值为 null，则为 null 的判断结果为 true；若集合不为 null，但长度为 0，则为 null 的判断结果为 false。

A、修改处理器

```
// 对象为null
Student studentNull = null;
model.addAttribute("studentNull", studentNull);
```

```
// List的size为0
List<String> cities = new ArrayList<>();
model.addAttribute("cities", cities);
```

B、修改 index 页面

```
<!--null 字面量-->
<div>
    <span th:if="{user} == null">对象未定义</span>
    <span th:if="{studentNull} == null">学生为空</span>
    <span th:if="{cities} != null">集合长度为0，但不为null</span>
</div>
```

3.5.2 Thymeleaf 运算符

(1) 字符串拼接运算符

可以使用加号(+)进行字符串拼接，也可以将文本字面量与动态数据直接写入由双竖线括起来的字符串内，此时无需使用加号进行连接。

```
<hr>
<div th:text="|我的名字叫${student.name}|"></div>
```

(2) 算术运算符

+, -, *, /, %, 但不支持++与--运算。

(3) 关系运算符

以下两种写法均支持：

>, <, >=, <=, ==, !=

gt, lt, ge, le, eq, ne

（4）逻辑运算符

非：not 或者 !

与：and

或：or

（5）条件运行符

? :

```
<hr>
<!-- 以下两种写法均可-->
<div th:text="${gender} == 'male' ? '男' : '女'"></div>
<div th:text="${gender} == 'male' ? '男' : '女'"></div>
```

3.6 Thymeleaf 内置对象

为了方便使用，Thymeleaf 中内置了很多对象，程序员可以直接使用。

3.6.1 ServletAPI 对象

通过#request、#session、#servletContext 可以获取到 HttpServletRequest、HttpSession 及 ServletContext 对象，然后就可以调用其相应的方法了。

（1）修改处理器

在处理器中再定义一个处理器方法。

```
@RequestMapping("/test/attr")
public String attrHandle(HttpServletRequest request, HttpSession session) {
    request.setAttribute("req", "reqValue");
    session.setAttribute("ses", "sesValue");
    session.getServletContext().setAttribute("app", "appValue");

    return "index2";
}
```

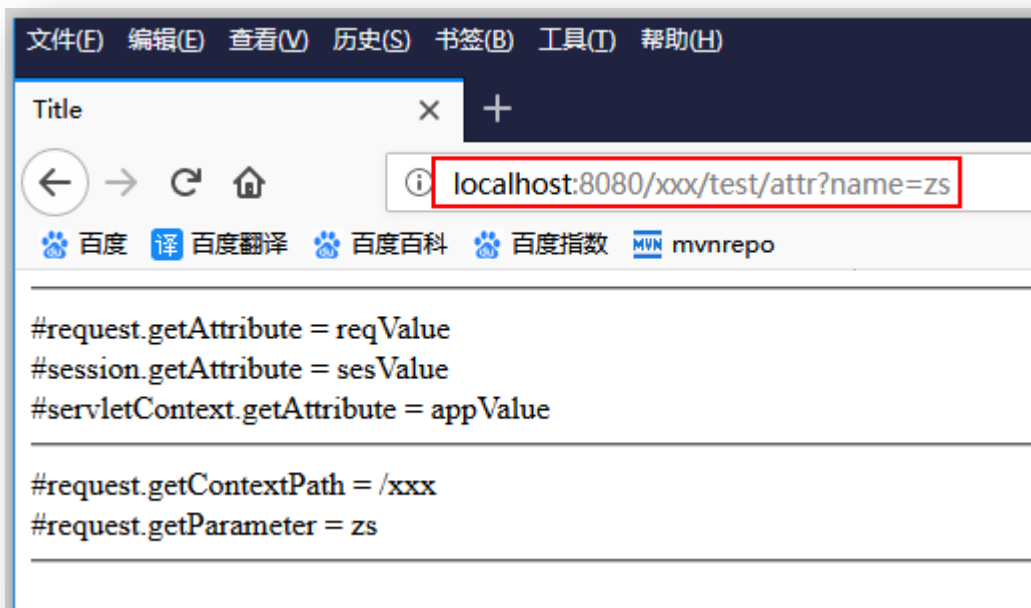
(2) 定义页面

再定义一个页面 index2.html。

```
<hr>
<!-- 获取域属性值-->
<span th:text="${#request.getAttribute('req')}"></span> <br>
<span th:text="${#session.getAttribute('ses')}"></span> <br>
<span th:text="${#servletContext.getAttribute('app')}"></span> <br>

<hr>
<!-- 获取上下文路径-->
<span th:text="${#request.getContextPath()}"></span> <br>
<!-- 获取请求参数-->
<span th:text="${#request.getParameter('name')}"></span> <br>
```

(3) 提交请求及运行效果



3.6.2 表达式实用对象

(1) 实用对象简介

这些实用对象都是以#开头，由于是对象，所以首字母是小写，且一般都是复数形式，即一般都是以s结尾。

- `#executionInfo`: 获取当前 Thymeleaf 模板对象的执行信息。
- `#messages`: 获取外部信息。
- `#uris/#urls`: URI/URL 处理工具。
- `#conversions`: 类型转换工具。
- `#dates`: 日期处理工具。
- `#calendars`: 日历处理工具。
- `#numbers`: 数字处理工具。
- `#strings`: 字符串处理工具。
- `#Objects`: 对象处理工具。
- `#booleans`: 布尔值处理工具。
- `#arrays`: 数组处理工具。
- `#lists`: List 处理工具。
- `#sets`: Set 处理工具。
- `#maps`: Map 处理工具。
- `#aggregates`: 聚合处理工具，例如，对数组、集合进行求和、平均值等。

- #ids: th:id 属性处理工具。

(2) 应用举例

下面就简单举几个例子，演示一下用法。

A、修改处理器

```
@RequestMapping("/test/util")
public String utilHandle(Model model) {

    int[] nums = {1,2,3,4,5};
    model.addAttribute("nums", nums);
    model.addAttribute("today", new Date());
    model.addAttribute("cardId", "325523200106256537");

    return "index3";
}
```

B、定义页面

再定义一个页面 index3.html。

```
<hr>
<!-- 格式化日期-->
<span th:text="${#dates.format(today, 'yyyy-MM-dd')}"></span> <br>
<!-- 从身份证中截取出生日-->
<span th:text="${#strings.substring(cardId, 6, 14)}"></span> <br>
<!-- 对数组元素求和-->
<span th:text="${#aggregates.sum(nums)}"></span> <br>
```

C、效果

2018-08-16
20010625
15

3.7 路径问题基础理论

3.7.1 路径的构成

路径由两部分构成：资源路径与资源名称。即

路径 = 资源路径 + 资源名称

资源路径与资源名称的分水岭为：路径中的最后一个斜杠。斜杠的前面部分称为资源路径，后面部分称为资源名称。

例如：

- 请求路径：<http://localhost:8080/xxx/test/index>
- 资源路径：<http://localhost:8080/xxx/test>
- 资源名称：index

3.7.2 路径的分类

根据是否可以唯一的定位一个资源，可以将路径划分为两类：绝对路径与相对路径。

- 绝对路径：可以唯一的定位一个资源的路径。在 Web 应用中，一般使用 URL 形式表示。
- 相对路径：仅依赖此路径无法唯一定位资源，但若为其指定一个参数路径，则可以将其转换为一个绝对路径，这样的路径称为相对路径。在 Web 应用中，一般使用 URI 形式表示。
- 转换关系：绝对路径 = 参照路径 + 相对路径

3.7.3 绝对路径分类

根据路径作用的不同，可以将绝对路径分为：资源定义路径，与资源请求路径。

- 资源定义路径：用于表示资源位置的路径。
- 资源请求路径：客户端所发出的对指定资源的请求路径。

3.7.4 相对路径分类

根据相对路径是否以斜杠开头，可以划分为两类：斜杠路径，与非杠路径。

- 斜杠路径：以斜杠开头的相对路径。
- 非杠路径：不以斜杠开头的相对路径。

3.7.5 斜杠路径分类

对于斜杠路径，根据其出现的位置的不同，可以划分为：前台路径与后台路径。

- 前台路径：出现在 HTML、JS、CSS，及 JSP 文件的静态部分的斜杠路径。例如，`<img src=""/>` `<a href=""> background:img("") window.location.href=""`。
- 后台路径：出现在 Java 代码、JSP 文件的动态部分（Java 代码块、JSP 动作等）、XML、properties 等配置文件中的斜杠路径。

3.7.6 路径解析器

相对路径，最终都会经过路径解析器，将其转换为绝对路径，以定义或定位一个资源。不同的相对路径，其路径解析器也是不同的。

- 前台路径：路径解析器为浏览器。
- 后台路径：路径解析器为服务器。
- 非杠路径：若非杠路径出现在前台路径位置，其路径解析器为浏览器；若非杠路径出现在后台路径位置，其路径解析器为服务器。

3.7.7 解析规则

不同的路径解析器，对同一个相对路径的解析结果是不同的。所谓解析结果，指的是将相对路径所转换的绝对路径。由于 $\text{绝对路径} = \text{参照路径} + \text{相对路径}$ ，所以，不同的解析器，会为相对路径匹配不同的参照路径。换句话说就是，我们学习的重点是，浏览器、服务器对于不同的相对路径所匹配的参照路径到底是谁。

（1）一般规则

- 前台路径：其参照路径为当前 web 服务器的根。
 - 后台路径：其参照路径为当前 web 应用的根。
 - 非杠路径：其参照路径为当前请求路径的资源路径。
- 例如，
- 请求路径：<http://localhost:8080/xxx/test/index>
 - 当前 web 服务器的根：<http://localhost:8080>
 - 当前 web 应用的根：<http://localhost:8080/xxx>
 - 资源路径：<http://localhost:8080/xxx/test>

（2）规则特例

- 在代码中使用 `HttpServletResponse` 的 `sendRedirect()`方法使用斜杠路径进行重定向时，其参照路径按照之前规则，应该是当前 web 应用的根，但实际情况是，当前 web 服务器的根。
- 以上规则对于一次跳转是没有问题的，若跳转次数超过一次，则有可能会存在问题。

