

SpringBoot 入门

第 23 章

SpringBoot 基础 1

优就业.JAVA 教研室

课程目标

目标 1: SpringBoot 入门演示

目标 2: 运用 SpringBoot 进行基本 web 应用开发--Json 处理

目标 3: 运用 SpringBoot 进行基本 web 应用开发--传递参数

目标 4: 运用 SpringBoot 进行基本 web 应用开发--静态资源

目标 5: 运用 SpringBoot 进行基本 web 应用开发--webJar

目标 6: 运用 SpringBoot 属性配置

目标 7: 运用 SpringBoot 构建 RestFulApi

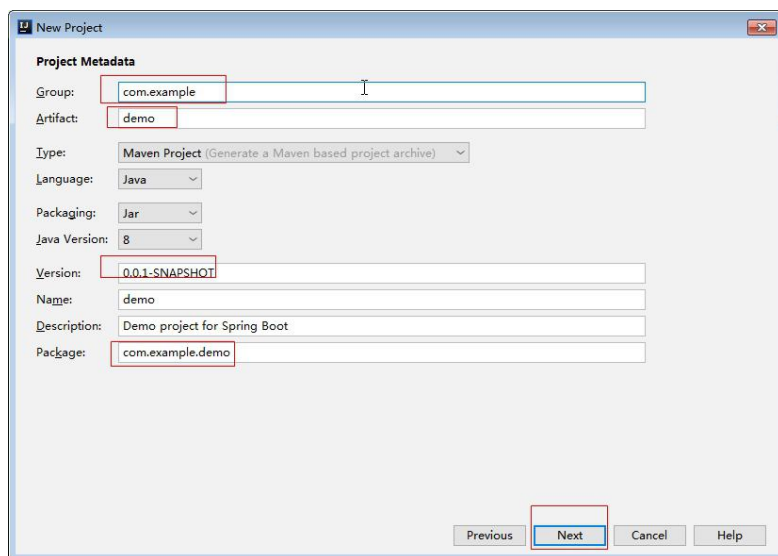
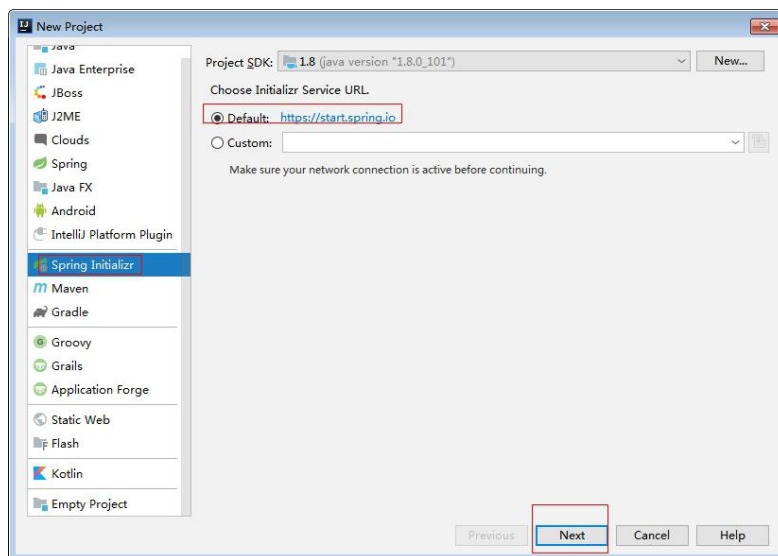
目标 8: 运用 SpringBoot 使用 Swagger2 构建 API 文档

目标 9: 运用 SpringBoot Jdbc 操作数据库

目标 10: 运用 SpringBoot 集成 Mybatis 操作数据库

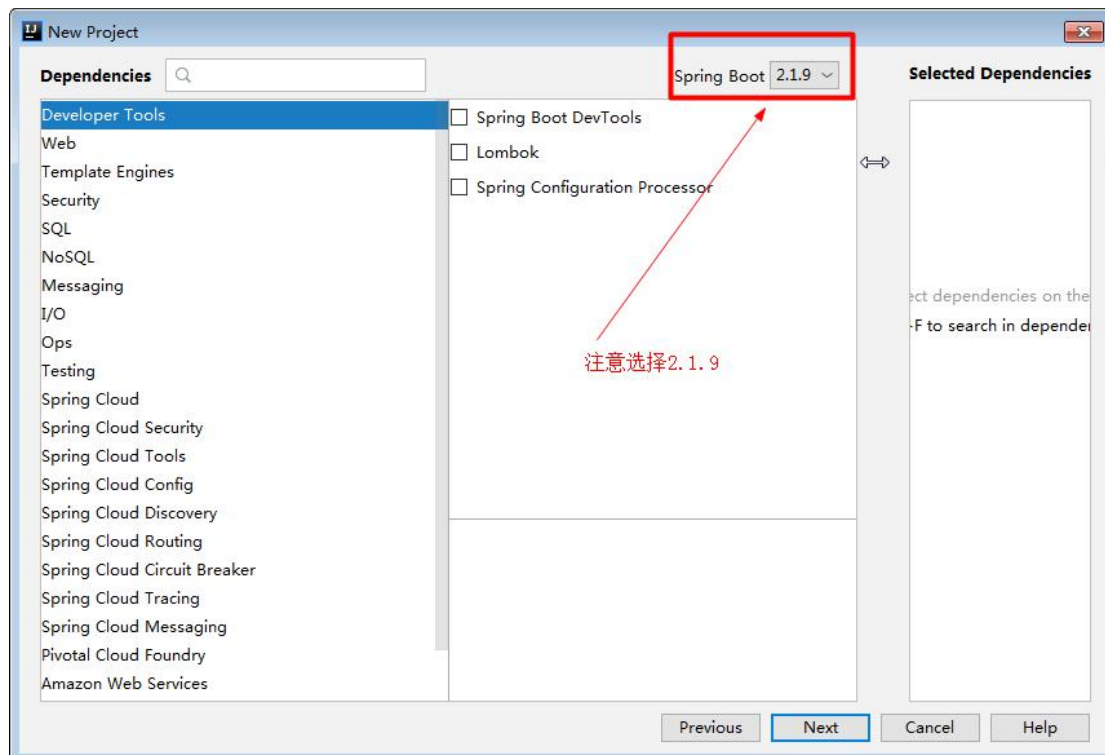
一、快速入门：创建第一个 SpringBoot 工程

1、点击 File--->New--->Project...



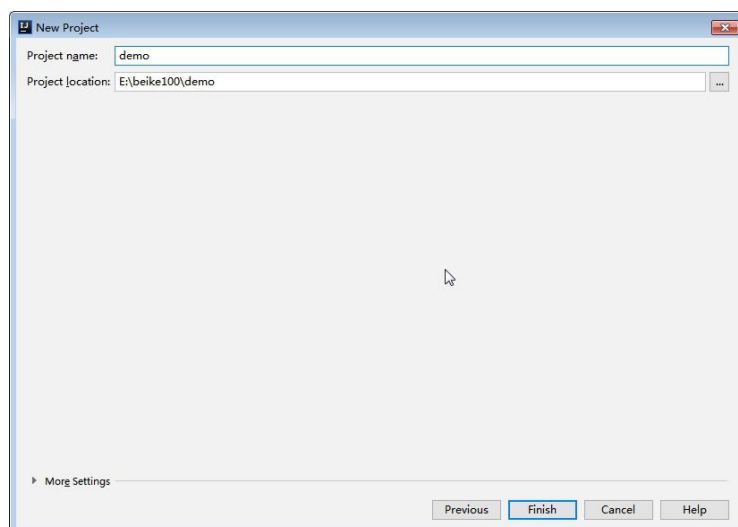
输入 maven 属性，注意 Artifact 不要出现大写、特殊字符。

2、选择 SpringBoot 版本，选择项目需要依赖的相关骨架包

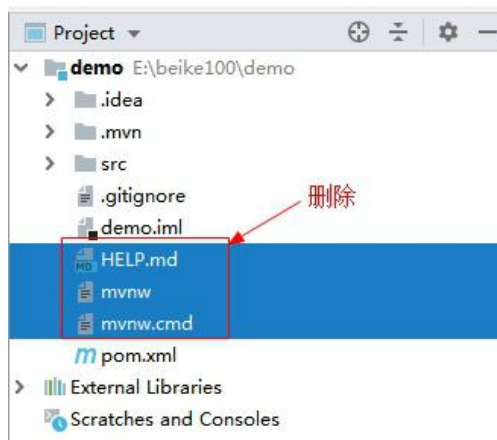


注意：一定要选择 **springBoot** 版本 **2.1.9**

3、设置项目保存目录：



4、项目创建完成，工程主界面如下：



5、项目说明

- (1)、默认有个 Demo001Application 类,里面是 spring boot 的载入函数
- (2)、resource 目录下有个 application.properties 文件,这个是 Spring boot 的配置文件
- (3)、test 目录下有个测试类 Demo001ApplicationTests, 这个是 spring boot 的单元测试
- (4)、pom.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.offcn</groupId>

  <artifactId>demo001</artifactId>

  <version>1.0</version>

  <packaging>jar</packaging>
```

```
<name>demo001</name>

<description>Demo project for Spring Boot</description>

<parent>

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

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

    <version>2.1.9.RELEASE</version>

    <relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

    <java.version>1.8</java.version>

</properties>

<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>


<build>

    <plugins>

        <plugin>

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

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

        </plugin>

    </plugins>

</build>

</project>
```

注意观察

一个继承 spring-boot-starter-parent,

两个依赖,

spring-boot-starter-web web 项目依赖必须,

spring-boot-starter-test spring boot 项目单元测试依赖

注意，很多同学配置的 maven 加速镜像，其中很多配置的阿里云 maven 镜像，在这会有找不到最新 Springboot 相关包的问题，请把加速镜像指向华为云：

```
<mirror>

    <id>huaweicloud</id>

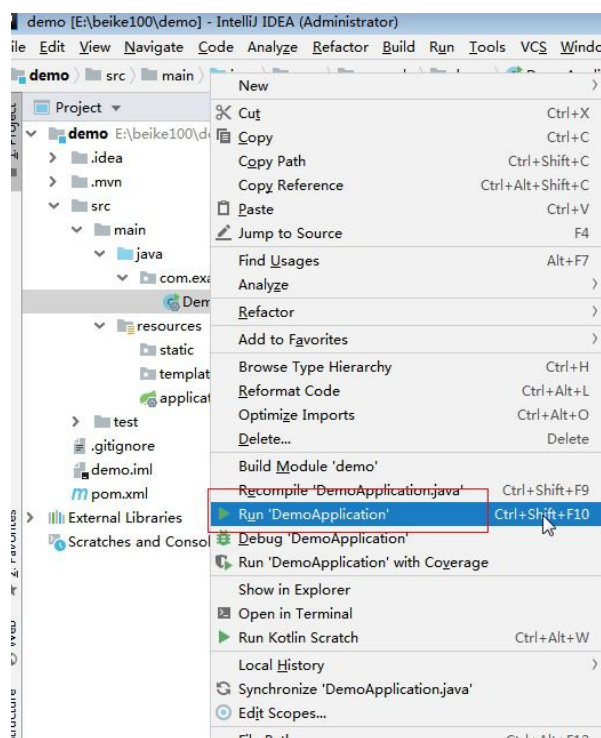
    <mirrorOf>*</mirrorOf>

    <url>https://mirrors.huaweicloud.com/repository/maven/</url>

</mirror>
```

6、启动项目

通过 spring boot 的启动类,这里是 Demo001Application，选中类，右键选择-->Run ‘DemoApplication’



在控制台出现如下输出：

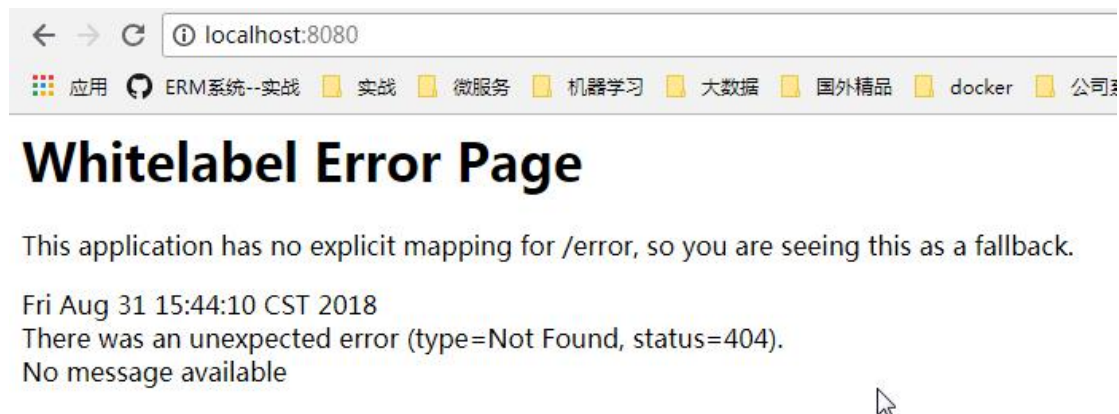
[illegible]

找到如下文字，表明 SpringBoot 已经成功启动：

```
ter      : Registering beans for JMX exposure on startup
WebServer : Tomcat started on port(s): 8080 (http) with context path ''
ion      : Started Demo001Application in 1.832 seconds (JVM running for 2.795)
```

tomcat 启动在 8080 端口，http 协议,启动花费了 1.832 秒

打开浏览器,输入地址:<http://localhost:8080>,出现如下画面



出现上图 404 错误是正常的,因为我们什么都没写。

7、编写 HelloController

```
package com.offcn.demo.controller;

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

@Controller

public class HelloController {

    @RequestMapping(value="/hello",method=RequestMethod.GET)

    @ResponseBody

    public String sayHello() {

        return "hello spring Boot!";

    }

}
```

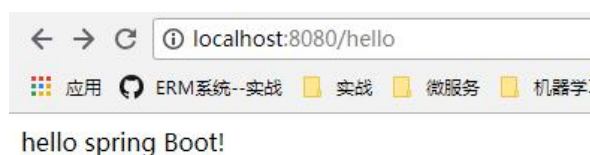
注意 HelloController 所在包，必须在 com.offcn.demo 包，或者子包下面。

重启应用，看控制台输出：

```
lerAdapter : Looking for @ControllerAdvice: org.springframework.boot.web.servlet.cont
lerMapping : Mapped "{[/hello],methods=[GET]}" onto public java.lang.String com.offcr
lerMapping : Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<
lerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springframework
erMapping : Mapped URL path [/webjars/**] onto handler of type [class org.springfram
erMapping : Mapped URL path [/**] onto handler of type [class org.springframework.we
ter : Registering beans for JMX exposure on startup
```

重启发现刚才写的 hello 已经映射出来了

访问 <http://localhost:8080/hello>



二、SpringBoot 基本 web 应用开发

1、SpringBoot json 支持

(1)、创建实体 bean Car

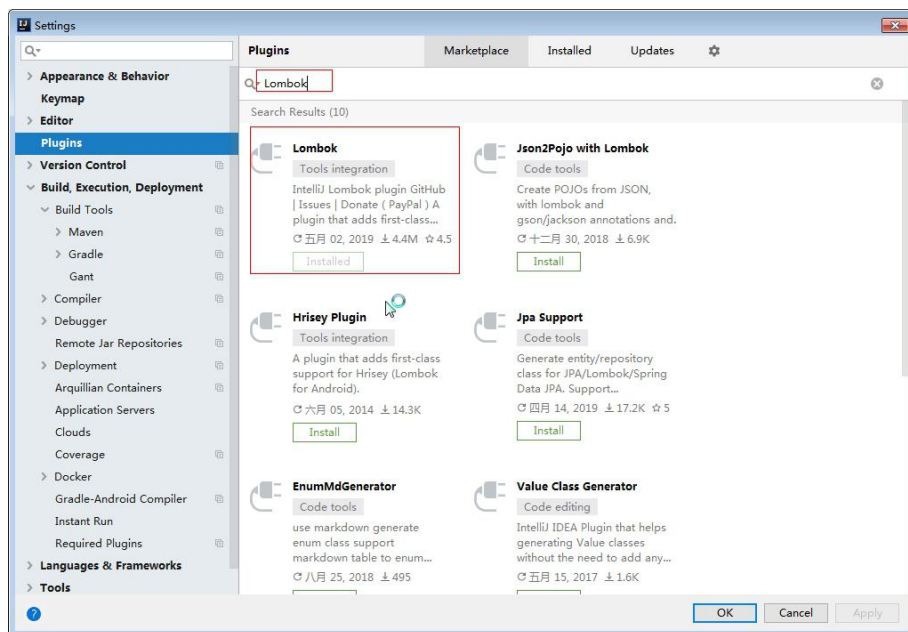
Lombok 使用

1、导入依赖库

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.6</version>
</dependency>
```

2、安装插件

Lombok



3、在实体 bean 使用

@Data //get 、 set

@AllArgsConstructor //所有参数的有参数构造函数

@NoArgsConstructor //无参数构造函数

```
package com.offcn.po;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
@Data //get 、 set
@AllArgsConstructor //所有参数的有参数构造函数
@NoArgsConstructor //无参数构造函数
public class Car {
    private Integer id;
    private String name;
    private Float price;
    @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss", timezone="GMT+8")
    private Date createdate;
}
```

(2)、创建 Controller CarController

```
package com.offcn.controller;

import com.offcn.po.Car;
import org.springframework.cache.CacheManager;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;
import java.util.List;

@Controller
@RequestMapping("/car")
public class CarController {

    @RequestMapping("/findone")
    public Car findOneCar() {
        Car car = new Car(1, "toyo", 1999.99F, new Date(), "13567890001");

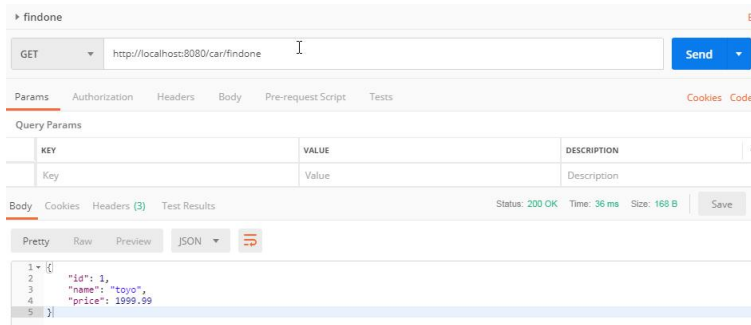
        return car;
    }
}
```

说明：@RestController 注解，等于@Controller 与 @ResponseBody 一起使用

(3)、测试获取单个对象 json

开启 postman，设置发出 get 请求，请求地址：

<http://localhost:8080/car/findone>



(4)、修改 Controller CarController 新增返回 list 集合方法

```
@RequestMapping("/getall")
public List<Car> getAll() {
    List<Car> list=new ArrayList<>();
    Car car1 = new Car(1, "toyota", 1999.99F, new Date(), "13567890001");
    Car car2= new Car(2, "dazhong", 2999.99F, new Date(), "13567890001");
    Car car3 = new Car(3, "fengtian", 3999.99F, new Date(), "13567890001");
    Car car4 = new Car(4, "benchi", 4999.99F, new Date(), "13567890001");

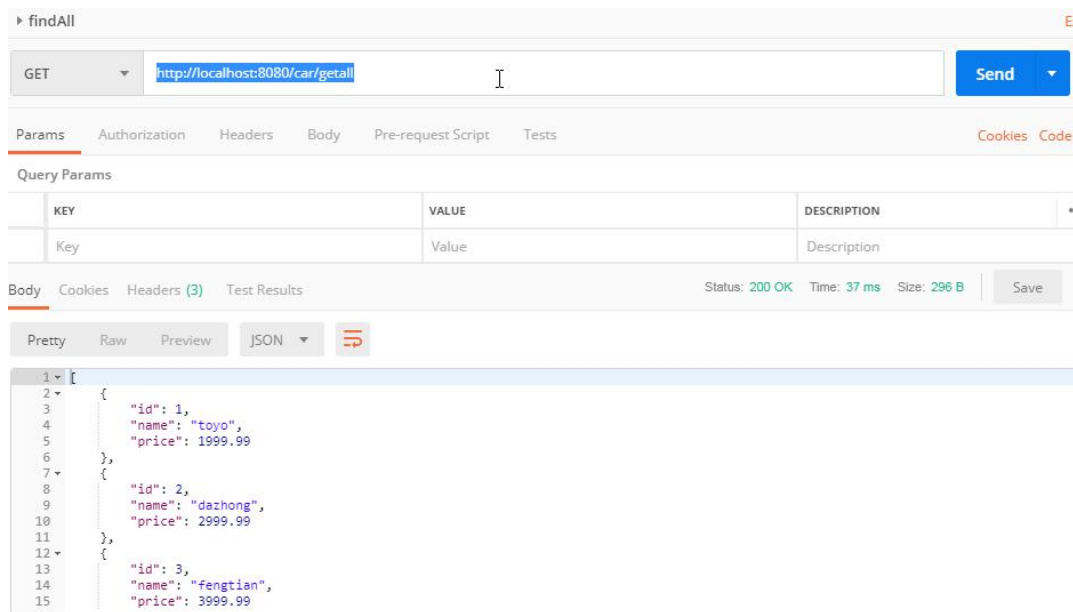
    list.add(car1);
    list.add(car2);
    list.add(car3);
    list.add(car4);

    return list;
}
```

(5)、测试获取集合多个对象 json

开启 postman，设置发出 get 请求，请求地址：

<http://localhost:8080/car/getall>



2、SpringBoot 请求传递参数

1.1. 第一类：请求路径传参

@RequestParam 获取查询参数。即 `url?name=value` 这种形式

@PathVariable 获取路径参数。即 `url/{id}` 这种形式

(1)、修改 Controller CarController 新增接收参数，返回单个对象方法

```
@RequestMapping("/getcar2/{name}")

public Car getCarById(@RequestParam(name="id") Integer
id,@PathVariable(name="name") String name) {

    Car car = new Car();

    car.setId(id);

    car.setName(name);

    car.setPrice(100000.99F);
}
```

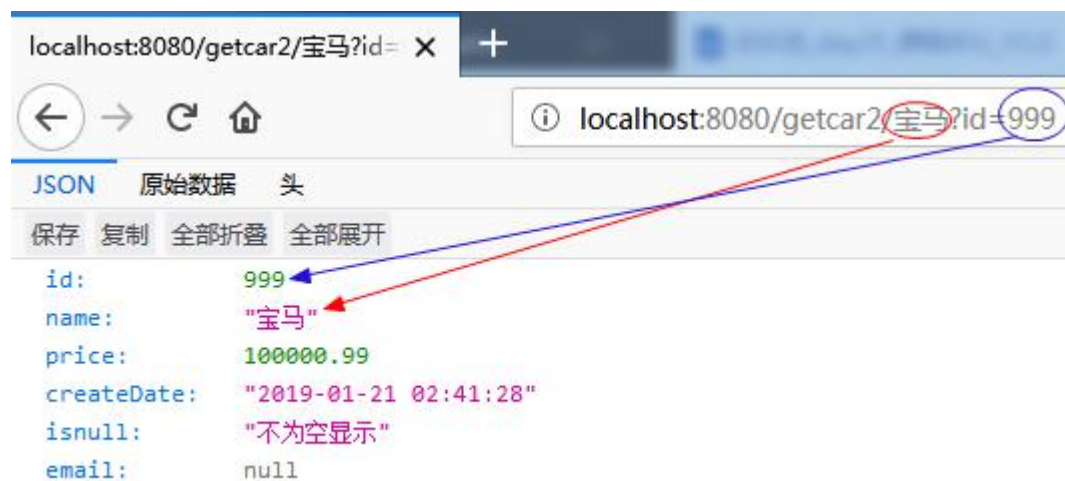
```
car.setCreateDate(new Date());

return car;

}
```

(2)、测试传递参数获取单个对象 json

请求地址: <http://localhost:8080/getcar2/宝马?id=999>



1.2. 第二类: @RequestBody 参数

(3)、修改 Controller CarController 新增接收封装对象参数, 返回单个及多个对象方法

```
@RequestMapping("/getcar3")

public Car getCarById(@RequestBody Car car) {

    return car;

}
```


@requestBody 注解常用来处理 content-type 不是默认的 application/x-www-form-urlencoded 编码的内容，比如说：application/json 或者是 application/xml 等。一般情况来说常用其来处理 application/json 类型。

(4)、测试传递 json 参数获取单个对象 json

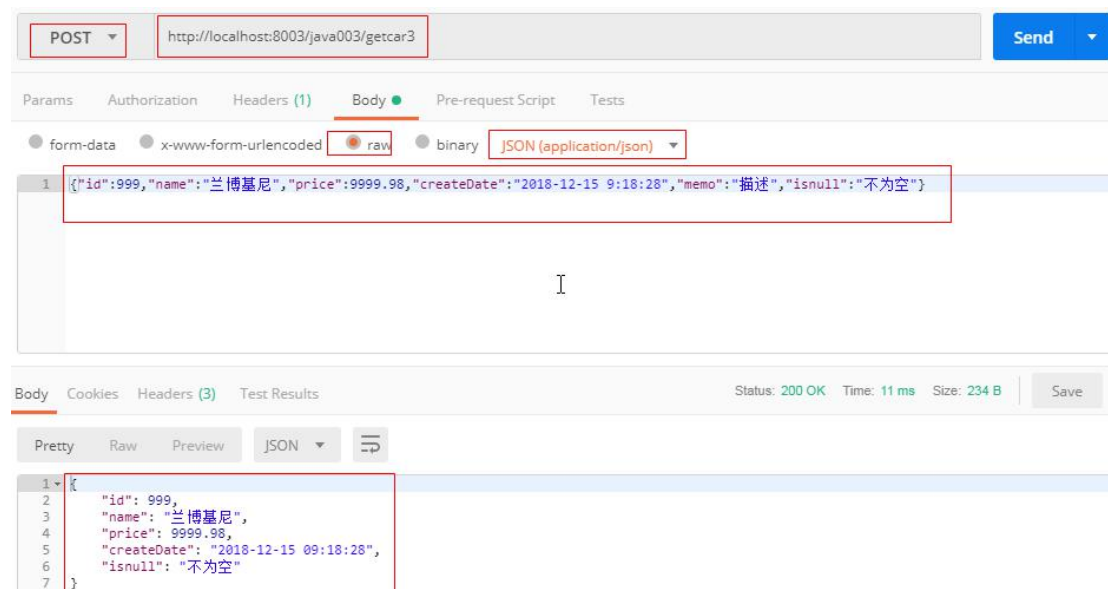
开启 postman，设置发出 post 请求，请求地址：

<http://localhost:8003/java003/getcar3>

请求参数，选择 body,选择 raw 方式，发送 JSON(application/json)请求

请求数据数据：

```
{ "id": 999, "name": "兰博基尼", "price": 9999.98, "createDate": "2018-12-15 9:18:28", "memo": "描述", "isnull": "不为空" }
```



1.3. 第三类：Body 参数，无注解

(5)、修改 Controller CarController 新增接收封装对象参数，返回单个及多个对象方法

```
@RequestMapping("/getcar4")

public Car getCarById(Car car) {

    return car;

}

@InitBinder

private void initBinder(WebDataBinder webDataBinder){
webDataBinder.addCustomFormatter(new DateFormatter("yyyy-MM-dd HH:mm:ss"));

}
```

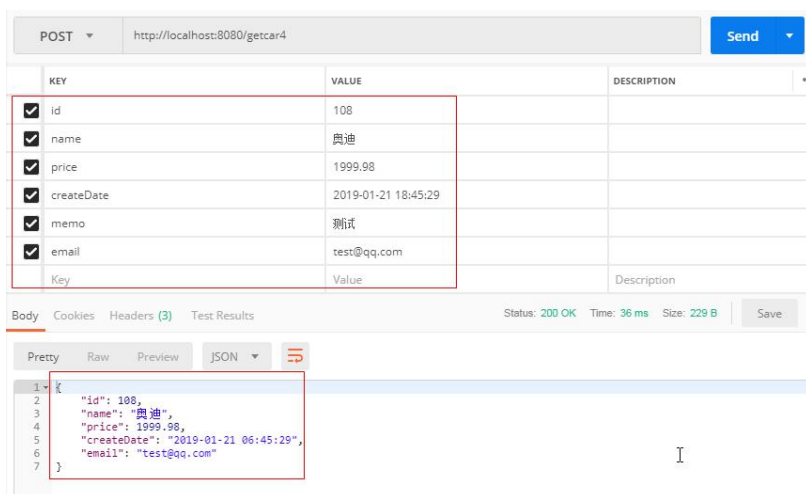
@InitBinder 用来格式化 request 请求中字符串（2018-01-19 17:25:29）为日期时间类型

(6)、测试传递普通 body 参数获取单个对象 json

开启 postman，设置发出 post 请求，请求地址：

<http://localhost:8080/getcar4>

请求参数，选择 body



3、SpringBoot 静态资源

(1)、默认静态资源映射

Spring Boot 对静态资源映射提供了默认配置

Spring Boot 默认将 /** 所有访问映射到以下目录：

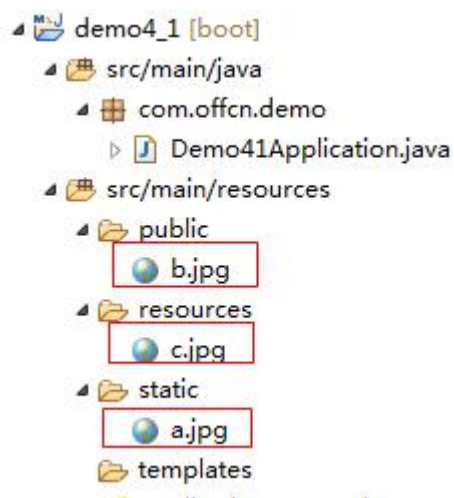
classpath:/static

classpath:/public

classpath:/resources

classpath:/META-INF/resources

如：在 resources 目录下新建 public、resources、static 三个目录，并分别放入 a.jpg b.jpg c.jpg 图片



浏览器分别访问：

<http://localhost:8080/a.jpg>

<http://localhost:8080/b.jpg>

<http://localhost:8080/c.jpg>

均能正常访问相应的图片资源。那么说明，Spring Boot 默认会挨个从 public resources static 里面找是否存在相应的资源，如果有则直接返回。

（2）、自定义静态资源访问

试想这样一种情况：一个网站有文件上传文件的功能，如果被上传的文件放在上述的那些文件夹中会有怎样的后果？

网站数据与程序代码不能有效分离；

当项目被打包成一个.jar 文件部署时，再将上传的文件放到这个.jar 文件中是有多么低的效率；

网站数据的备份将会很痛苦。

此时可能最佳的解决办法是将静态资源路径设置到磁盘的基本个目录。

第一种方式

1、配置类

```
package com.offcn.demo.config;

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

@Configuration

public class WebMvcConfig implements WebMvcConfigurer {

    @Override

    public void addResourceHandlers(ResourceHandlerRegistry registry) {

        //将所有 D:\springboot\pic\ 访问都映射到/myPic/** 路径下

        registry.addResourceHandler("/myPic/**").addResourceLocations("file:D:\\springboot\\pic\\");
    }
}
```

```
}  
  
}
```

上面的意思就是：将所有 D:/springboot/pic/ 访问都映射到/myPic/** 路径下

2、重启项目

例如，在 D:/springboot/pic/中有一张 logo.jpg 图片

在浏览器输入：http://localhost:8080/myPic/logo.jpg 即可访问。

第二种方式

首先，我们配置 application.properties

```
web.upload-path=D:/springboot/pic/  
  
spring.mvc.static-path-pattern=/**  
  
spring.resources.static-locations=classpath:/META-INF/resources/,classpath:  
/resources/,\  
  
classpath:/static/,classpath:/public/,file:${web.upload-path}
```

注意：

web.upload-path: 这个属于自定义的属性，指定了一个路径，注意要以/结尾；

spring.mvc.static-path-pattern=/**: 表示所有的访问都经过静态资源路径；

spring.resources.static-locations: 在这里配置静态资源路径，前面说了这里的配置是覆盖默认配置，所以需要将默认的也加上否则 static、public 等这些路径将不能被当作静态资源路径，在这个最末尾的 file:\${web.upload-path}之所有要加 file:是因为指定的是一个具体的硬盘路径，其他的使用 classpath 指的是系统环境变量。

然后，重启项目

例如，在 D:/springboot/pic/中有一张 8.png 图片

在浏览器输入：http://localhost:8080/8.png 即可访问。

4、WebJars

在 SpringBoot 中，允许我们直接访问 WEB-INF/lib 下的 jar 包中的 **META-INF/resources** 目录资源，即 WEB-INF/lib/{*.jar}/META-INF/resources 下的资源可以直接访问。

WebJars 也是利用了此功能，将所有 **前端的静态文件打包成一个 jar 包**，这样对于引用放而言，和普通的 jar 引入是一样的，还能很好的对前端静态资源进行管理。

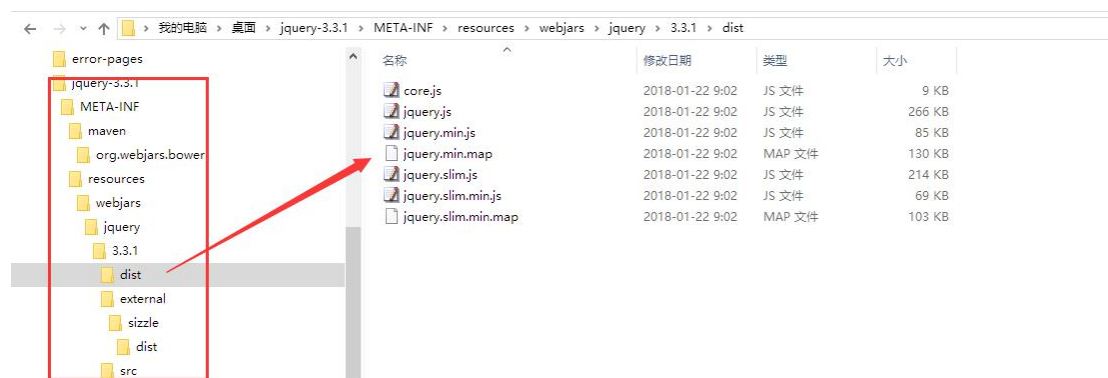
WebJars 是将 web 前端资源（如 jQuery & Bootstrap）打成 jar 包文件。借助版本管理工具 (Maven、gradle 等) 进行版本管理，保证这些 Web 资源版本唯一性。避免了文件混乱、版本不一致等问题。

（1）、WebJar 结构

开始使用前，我们看下 JQuery 的 webjars，借此来了解下 webjars 包的目录结构。以下是基于 jquery-3.3.1.jar:

```

META-INF
├── maven
│   ├── org.webjars.bower
│   │   └── jquery
│   │       ├── pom.properties
│   │       └── pom.xml
├── resources
│   └── webjars
│       └── jquery
│           └── 3.3.1
│               └── (静态文件及源码)
    
```



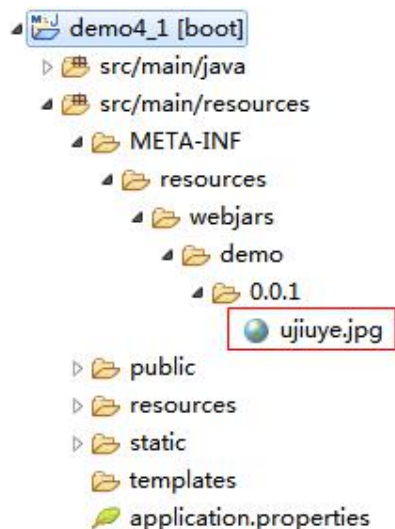
jquery-3.3.1 目录结构

所以可以看出，静态文件存放规则：META-INF/resources/webjars/\${name}/\${version}。

(2)、WebJars 实践

接下来我们以一个简单的示例，对 webjars 进行简单的实践下。

1、在 src/main/resources 路径下创建 META-INF/resources/webjars/demo/0.0.1 目录，同时为了演示效果，拷贝一个图片到此目录下。



2、编写一个简单的 html 页面，放在在 src/main/resources/static 下(当然也可以直接放在 webjar 下了，只需要后面加个映射关系即可)，内容如下：

```
<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Hello,WebJars</title>

</head>

<body>

    <h1>Hello,WebJars</h1>

</body>
```

```
</html>
```

3、编写配置类，添加一个资源映射关系.其实也可以不写，因为第 4 节也有说过，springboot 默认四个资源路径里面就包含了/META-INF/resources/了

```
@Configuration

public class WebMvcConfig implements WebMvcConfigurer {

    @Override

    public void addResourceHandlers(ResourceHandlerRegistry registry) {

        //配置映射关系
        registry.addResourceHandler("/webjars/**").addResourceLocations("classpath:
/META-INF/resources/webjars/");

    }

}
```

4、编写控制层，返回此页面地址。

```
/** webjar 示例

 * @author sunny

 * */

@Controller

public class WebJarsDemoController {

    @GetMapping("/")

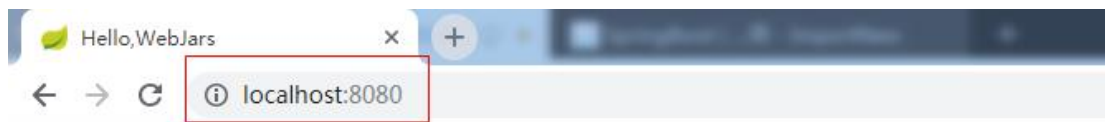
    public String index() {

        return "index.html";

    }

}
```

5、启动应用，访问地址即可：



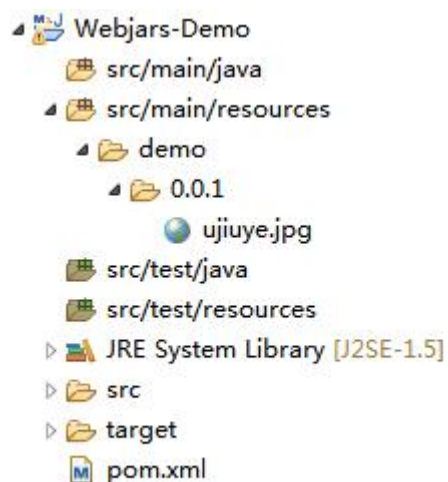
Hello, WebJars



可以看见图片已经正确显示出来了。

6、现在直接将 META-INF 下打成一个 jar，然后加入依赖进入。在来测试下。

这里直接创建一个新的工程，存在静态资源信息，工程结果如下：



然后对应的 pom 配置文件加入一个资源拷贝动作：

```
<build>

    <resources>

        <resource>

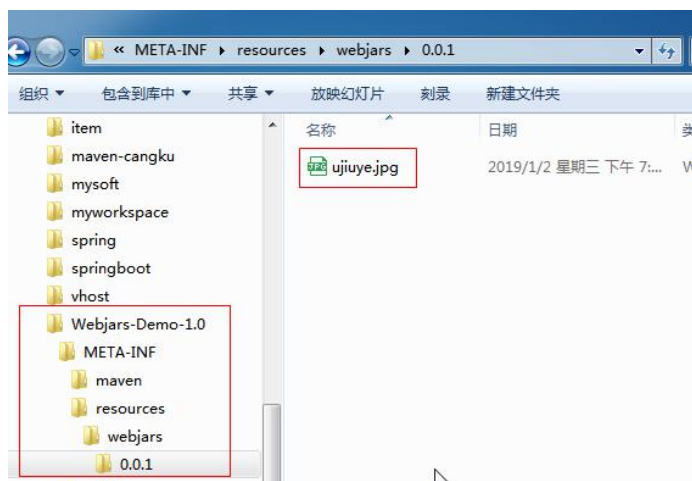
            <directory>${project.basedir}/src/main/resources</directory>
            <targetPath>${project.build.outputDirectory}/META-INF/resources/webjars</targetPath>

        </resource>

    </resources>

</build>
```

利用 maven 打包后，就可以看见 jar 包目录结构了：



7、然后我们删除了我们原先的资源文件或者图片重命名下，并引入依赖：

```
<dependency>

    <groupId>com.offcn</groupId>

    <artifactId>Webjars-Demo</artifactId>

    <version>1.0</version>

</dependency>
```

最后重新启动应用，再次访问下，依旧是正常显示的！

（1）、WebJars 常用引用

Jquery:

```
<dependency>

    <groupId>org.webjars</groupId>

    <artifactId>jquery</artifactId>

    <version>3.3.1-1</version>

</dependency>
```

Bootstrap:

```
<dependency>

    <groupId>org.webjars</groupId>

    <artifactId>bootstrap</artifactId>

    <version>4.2.1</version>

</dependency>
```

WebJars 查询: <https://www.webjars.org/all>

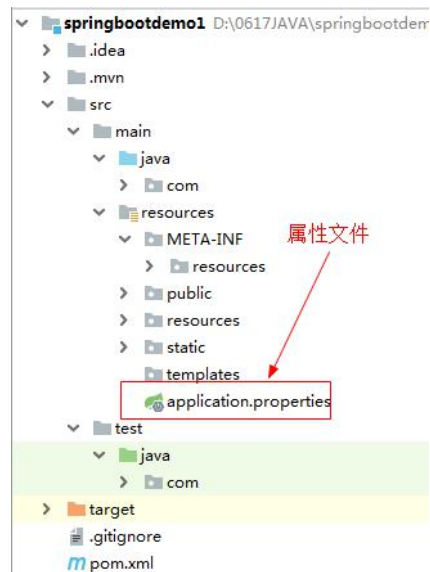
三、SpringBoot 属性配置

Spring Boot 并不真正是所谓的『零配置』，他的理念是“约定优于配置”采用了一些默认的习惯性配置，让你无需手动进行配置，从而让你的项目快速运行起来。所以要想玩转 Spring Boot，了解这些默认配置还是必不可少的。

1、项目默认属性配置文件所在位置及配置实例

创建 Spring Boot 项目时，会默认生成一个全局配置文件 application.properties(可以修改后缀

为.yml)，在 src/main/resources 目录下或者类路径的/config 下。我们可以通过修改该配置文件来对一些默认配置的配置值进行修改。



【修改默认配置】

1、spring boot 开发 web 应用的时候，默认 tomcat 的启动端口为 8080，如果需要修改默认的端口，则需要在 application.yml 添加以下记录：

```
server:

    port: 8888
```

重启项目，启动日志可以看到：Tomcat started on port(s): 8888 (http) 启动端口为 8888，浏览器中访问 http://localhost:8888 能正常访问。

2、spring boot 开发 web 应用的时候，访问路径为/，如果需要修改访问路径，则需要在 application.yml 添加以下记录：

```
server:

    port: 8888

servlet:

    context-path: /java001
```

重启项目，启动日志就可以看到：Tomcat started on port(s): 8888 (http) with context path '/java001'，浏览器中访问 <http://localhost:8888/java001> 能正常访问。

2、自定义属性及读取

我们可以在 application.yml 文件中，配置一些常量或者其他参数配置。读取的时候通过 Spring 的 @Value(“\${属性名}”) 注解即可。

(1)、在 application.yml 定义几个常量：

```
offcn_ip:

    1.1.1.1

offcn_port:

    9999
```

(2)、编写 Controller 类读取自定义属性

```
package com.offcn.demo.controller;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloConfigController {

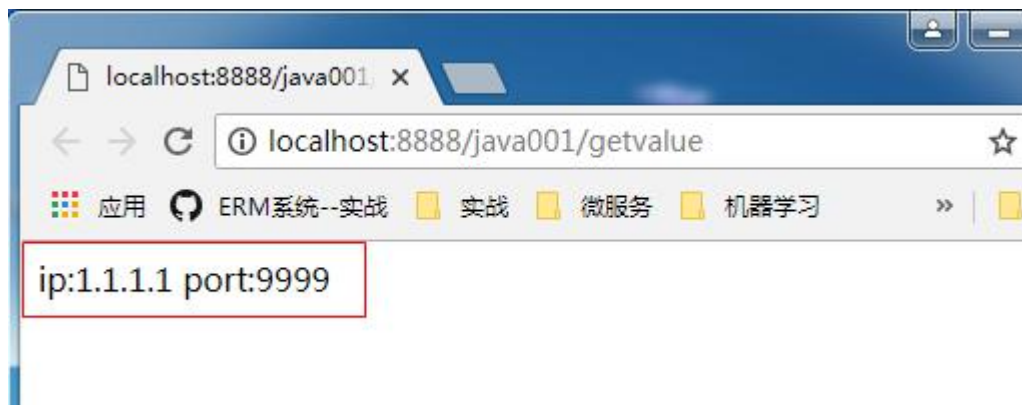
    @Value("${offcn_ip}")

    private String offcn_ip;
```

```
@Value("${offcn_port}")  
  
private String offcn_port;  
  
@GetMapping("/getvalue")  
  
public String getValue() {  
  
    return "ip:"+offcn_ip+" port:"+offcn_port;  
  
}  
  
}
```

访问 <http://localhost:8888/java001/getvalue>

显示结果如下：



3、实体类属性赋值

当属性参数变多的时候，我们习惯创建一个实体，用实体来统一接收赋值这些属性。

(1)、定义配置文件

```
userbody:
```

```
name: 优就业

password: 123456

birthday: 1992.10.28

mobile: 13802789765

address: 北京市朝阳区
```

(2)、创建实体类

```
package com.offcn.demo.bean;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix="userbody")

public class UserBody {

    private String name;

    private String password;

    private String birthday;

    private String mobile;

    private String address;

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

}
```

```
}

public String getPassword() {

    return password;

}

public void setPassword(String password) {

    this.password = password;

}

public String getBirthday() {

    return birthday;

}

public void setBirthday(String birthday) {

    this.birthday = birthday;

}

public String getMobile() {

    return mobile;

}

public void setMobile(String mobile) {

    this.mobile = mobile;

}

public String getAddress() {

    return address;

}

public void setAddress(String address) {
```



```
        this.address = address;

    }

    @Override

    public String toString() {

        return "UserBody [name=" + name + ", password=" + password + ",  
birthday=" + birthday + ", mobile=" + mobile  
        + ", address=" + address + "]";

    }

}
```

需要在实体类上增加注解@ConfigurationProperties，并指定 prrfix 前缀。

(3)、编写 Controller 调用属性 bean

```
package com.offcn.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.context.properties.EnableConfigurationProperties;

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

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

import com.offcn.demo.bean.UserBody;
```

```
@RestController

@EnableConfigurationProperties({UserBody.class})

public class HelloControllerBean {

    @Autowired

    UserBody userbody;

    @GetMapping("/getUser")

    public String getUser(){

        return userbody.toString();

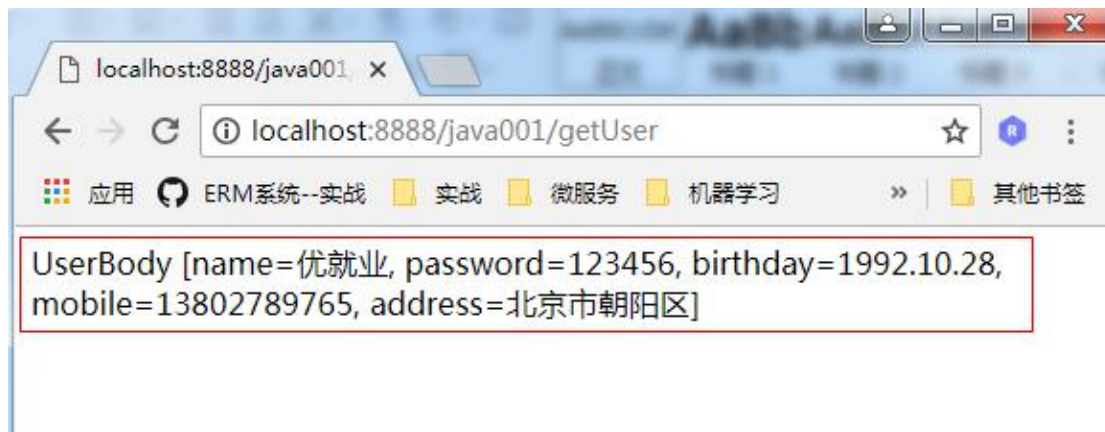
    }

}
```

EnableConfigurationProperties 注解需要加在调用类上，或者加在启动类 SpringbootSimpleApplication 上也可以。

访问地址：<http://localhost:8888/java001/getUser>

可以看到返回值：



4、自定义配置文件

application.yml 是系统默认的配置文件，当然我们也可以创建自定义配置文件，在路径 src/main/resources 下面创建文件 test.properties

注意：spring boot 1.5 版本后 @PropertySource 注解就不能加载自定义的 yml 配置文件了

(1)、定义 test.properties

```
testuser.name = "offcn"  
testuser.password = "123"  
testuser.birthday = "1978.10.28"
```

(2)、将配置赋值到 javabean

```
package com.offcn.demo.bean;  
  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.context.annotation.Configuration;
```

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

```
@Configuration
```

```
@PropertySource("classpath:test.properties")
```

```
@ConfigurationProperties(prefix = "testuser")
```

```
public class TestUser {
```

```
    private String name;
```

```
    private String password;
```

```
    private String birthday;
```

```
    public String getName() {
```

```
        return name;
```

```
    }
```

```
    public void setName(String name) {
```

```
        this.name = name;
```

```
    }
```

```
    public String getPassword() {
```

```
        return password;
```

```
    }
```

```
    public void setPassword(String password) {
```

```
        this.password = password;
```

```
    }
```

```
    public String getBirthday() {
```

```
        return birthday;
```

```
}

    public void setBirthday(String birthday) {

        this.birthday = birthday;

    }

    @Override

    public String toString() {

        return "testuser [name=" + name + ", password=" + password + ", birthday="
+ birthday + "]\n";

    }

}
```

@Configuration 注解包含@Component 注解

1.5 版本后需要通过@PropertySource(“classpath:test.properties”)指定配置文件

(3)、Controller 读取配置

```
package com.offcn.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.context.properties.EnableConfigurationProperties;

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

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

```
import com.offcn.demo.bean.TestUser;

import com.offcn.demo.bean.UserBody;

@RestController

@EnableConfigurationProperties({UserBody.class, TestUser.class})

public class HelloControllerBean {

    @Autowired

    UserBody userbody;

    @Autowired

    TestUser testUser;

    @GetMapping("/getUser")

    public String getUser(){

        return userbody.toString();

    }

    @GetMapping("/gettestuser")

    public String gettestUser() {

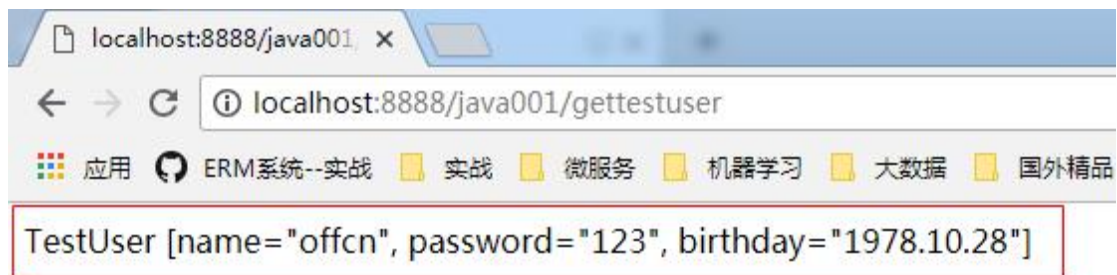
        return testUser.toString();

    }

}
```

访问地址: <http://localhost:8888/java001/gettestuser>

可以看到返回值:



5、多环境配置文件

使用多个 yml 配置文件进行配置属性文件

可以使用多个 yml 来配置属性, 将于环境无关的属性放置到 application.yml 文件里面; 通过与配置文件相同的命名规范, 创建 application-{profile}.yml 文件 存放不同环境特有的配置, 例如 application-test.yml 存放测试环境特有的配置属性, application-prod.yml 存放生产环境特有的配置属性。

通过这种形式来配置多个环境的属性文件, 在 application.yml 文件里面
spring.profiles.active=xxx 来指定加载不同环境的配置, 如果不指定, 则默认只使用
application.yml 属性文件, 不会加载其他的 profiles 的配置。

(1)、创建 application-dev.yml

```
server:  
  
  port: 8003  
  
  servlet:
```

```
context-path: /java003
```

(2)、创建 application-test.yml

```
server:  
  
port: 8001  
  
servlet:  
  
context-path: /java001
```

(3)、创建 application-prod.yml

```
server:  
  
port: 8002  
  
servlet:  
  
context-path: /java002
```

(4)、修改 application.yml

```
spring:  
  
  profiles:  
  
    active: test
```

通过设置，active: 的值对应不同的{profile}就可以使对应的配置文件生效。

四、SpringBoot 构建 RESTful API

1、RESTful 介绍

RESTful 是一种软件架构风格！

REST 就是指对同一个 URI 的资源的不同请求方式（GET，POST，PUT，DELETE）（表述）下的做出的不同的操作（查，增，改，删），改变的是资源的状态，即表述性状态转移。一个符合 REST 风格的 URI 就可以称之为一个 RESTful 的接口

2、RESTful 接口设计

在此我们以用户数据的基本操作来进行接口设计

HTTP 协议请求方法	SpringBoot 注解	URL	功能说明
POST	@PostMapping	/user/	创建一个用户
GET	@GetMapping	/user/	查询用户列表
GET	@GetMapping	/user/id	根据 id 查询一个用户
PUT	@PutMapping	/user/id	根据 id 更新一个用户
DELETE	@DeleteMapping	/user/id	根据 id 删除一个用户

3、用户实体 bean 创建

```
package com.offcn.po;

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

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private Long id;
    private String name;
    private Integer age;
}
```

4、创建 Controller UserController

```
package com.offcn.controllerold;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

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

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.PutMapping;

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

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

import com.offcn.po.User;
```

```
@RestController

@RequestMapping("/users-test")

public class UserController {

    private List<User> listUser=Collections.synchronizedList(new ArrayList<User>());

    /**
     * 获取全部用户信息
     * @return
     */
    @GetMapping("/")
    public List<User> getUserList(){

        return listUser;

    }

    /**
     * 新增用户
     * @param user
     * @return
     */
    @PostMapping("/")
    public String createUser(User user) {

        listUser.add(user);
    }
}
```

```
        return "success";

    }

    /**
     * 获取指定 id 用户信息
     *
     * @param id
     *
     * @return
     *
     */
    @GetMapping("/{id}")
    public User getUser(@PathVariable("id")Long id) {

        for (User user : listUser) {

            if(user.getId()==id) {

                return user;

            }

        }

        return null;

    }

    /**
     * 更新指定 id 用户信息
     *
     * @param id
     *
     * @param user
     *
     * @return
     *
     */
```

```
@PutMapping("/{id}")

public String updateUser(@PathVariable("id") Long id,User user) {

    for (User user2 : listUser) {

        if(user2.getId()==id) {

            user2.setName(user.getName());

            user2.setAge(user.getAge());

        }

    }

    return "success";

}

/**
 * 删除指定 id 用户
 *
 * @param id
 *
 * @return
 *
 */

@DeleteMapping("/{id}")

public String deleteUser(@PathVariable("id") Long id) {

    listUser.remove(getUser(id));

    return "success";

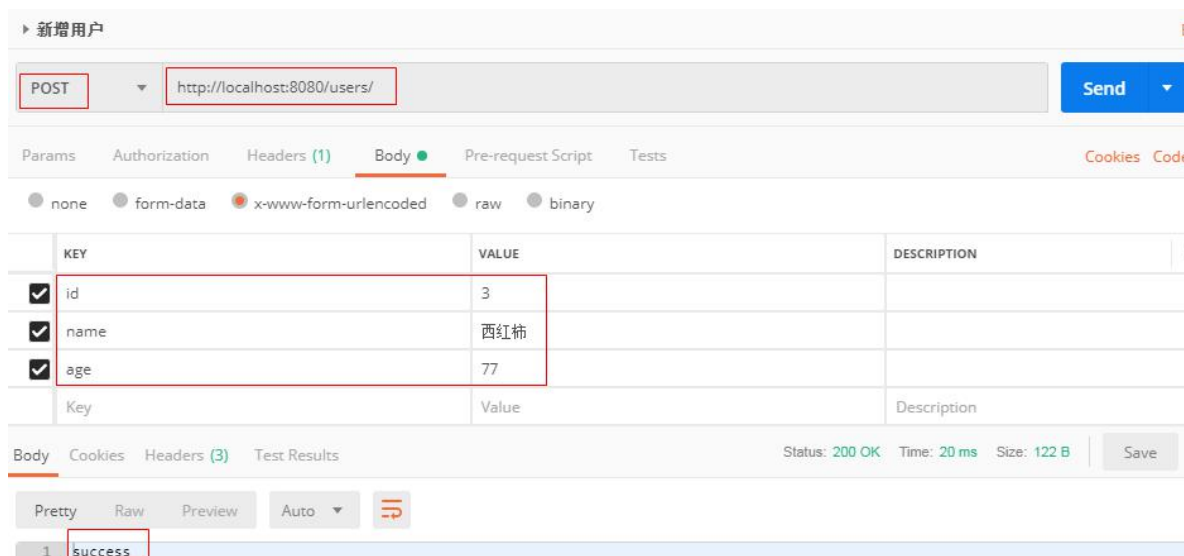
}
```

}

5、Postman 测试 RESTful 接口

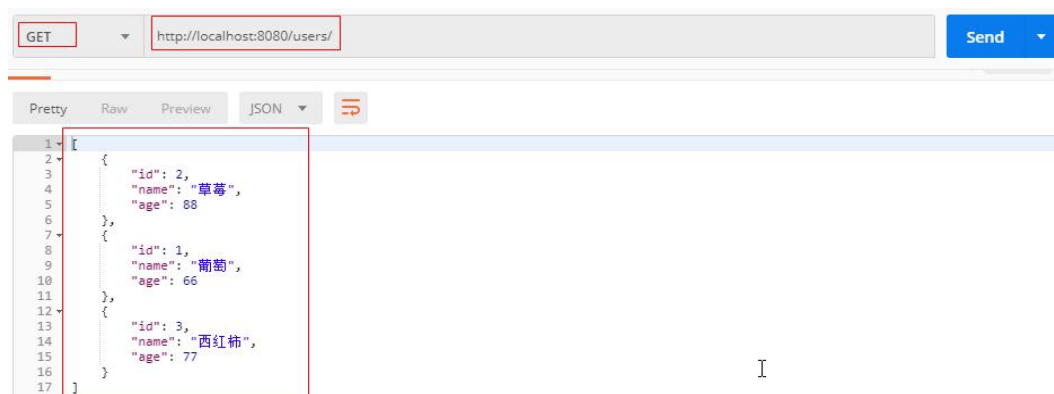
(1)、新增用户

post <http://localhost:8080/users/>



(2)、获取全部用户信息

get <http://localhost:8080/users/>



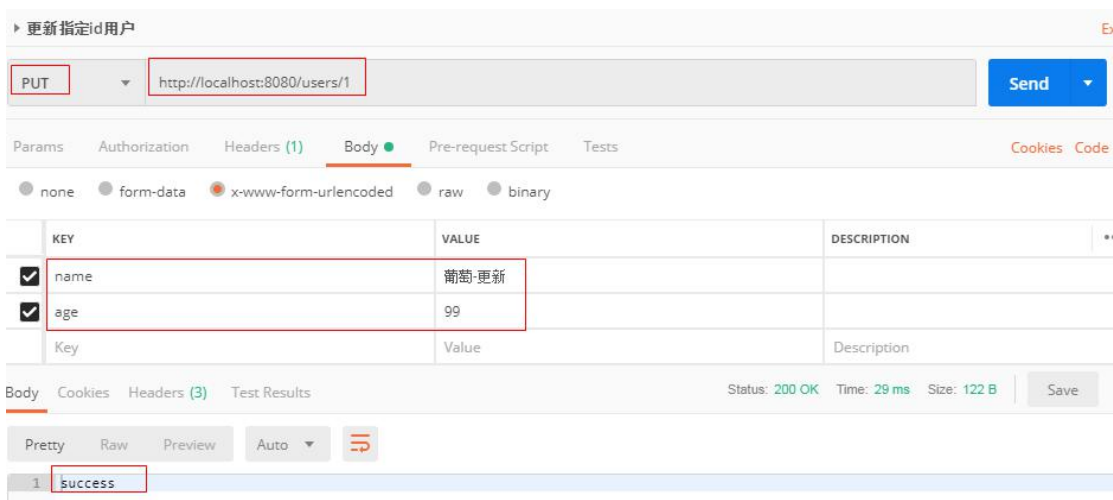
(3)、获取指定 id 用户信息

get <http://localhost:8080/users/id>



(4)、更新指定 id 用户信息

put <http://localhost:8080/users/id>



(5)、删除指定 id 用户信息

delete <http://localhost:8080/users/id>



五、SpringBoot 使用 Swagger2 构建 API 文档

1、Swagger2 介绍

编写和维护接口文档是每个程序员的职责，前面我们已经写好的接口

现在需要提供一份文档，这样才能方便调用者使用。

考虑到编写接口文档是一个非常枯燥的工作，我们采用 Swagger2 这套自动化文档工具来生成文档，它可以轻松的整合到 Spring Boot 中，并与 Spring MVC 程序配合组织出强大 RESTful API 文档。



2、SpringBoot 开启 Swagger2 支持

第一步：在 pom.xml 中加入 Swagger2 的依赖

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.2.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.2.2</version>
</dependency>
```

第二步：创建 Swagger2 配置类

```
package com.offcn.config;
```



```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .select()
            .apis(RequestHandlerSelectors.basePackage("com.offcn.controller"))
            .paths(PathSelectors.any())
            .build();
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            .title("Spring Boot中使用Swagger2构建RESTful APIs")
            .description("优就业")
            .termsOfServiceUrl("http://www.ujiuye.com/")
            .contact("Sunny")
            .version("1.0")
            .build();
    }
}
```

3、修改 Controller 增加文档注释

通过@Api 给类增加说明

通过@ApiOperation

注解来给方法增加说明

通过

@ApiImplicitParams

@ApiImplicitParam

注解来给参数增加说明

```
/**
 * 更新指定id用户信息
 * @param id
 * @param user
 * @return
 */
@PutMapping("/{id}")
@ApiOperation(value="更新指定id用户信息", notes="根据id更新用户信息")
@ApiImplicitParams({
    @ApiImplicitParam(name = "id", value = "用户ID", required = true,
dataType = "Long"),
    @ApiImplicitParam(name = "user", value = "用户详细实体user", required
= true, dataType = "User")
})
public String updateUser(@PathVariable("id") Long id,User user) {
    user.setId(id);
    userRepository.saveAndFlush(user);
    return "success";
}

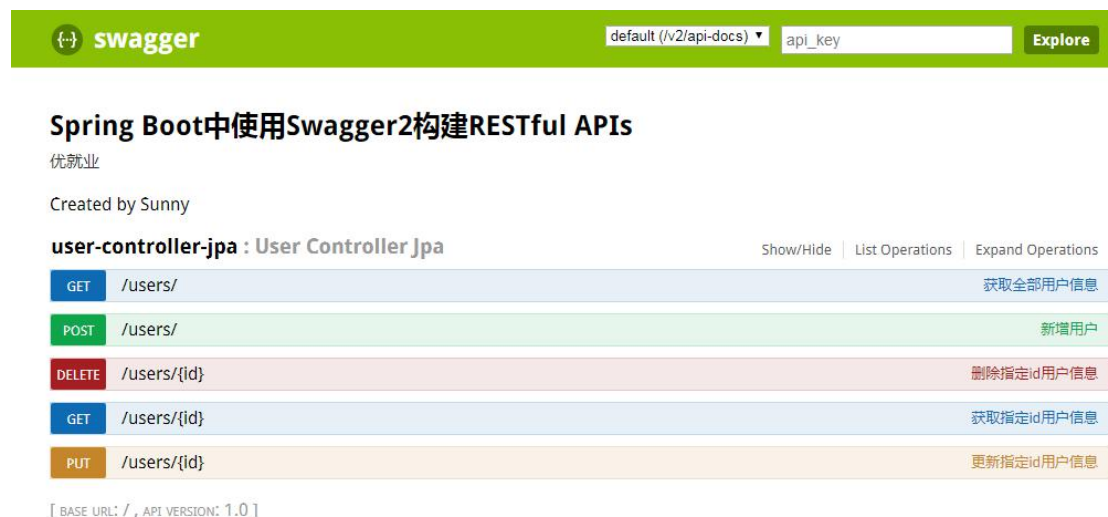
/**
 * 删除指定id用户
 * @param id
 * @return
 */
@DeleteMapping("/{id}")
@ApiOperation(value="删除指定id用户信息", notes="根据id删除用户信息")
@ApiImplicitParam(name = "id", value = "用户id", required = true, dataType
= "Long")
public String deleteUser(@PathVariable("id") Long id) {

    userRepository.deleteById(id);
    return "success";
}
```

4、查看 Swagger2 文档

重启应用

访问地址: <http://localhost:8080/swagger-ui.html>

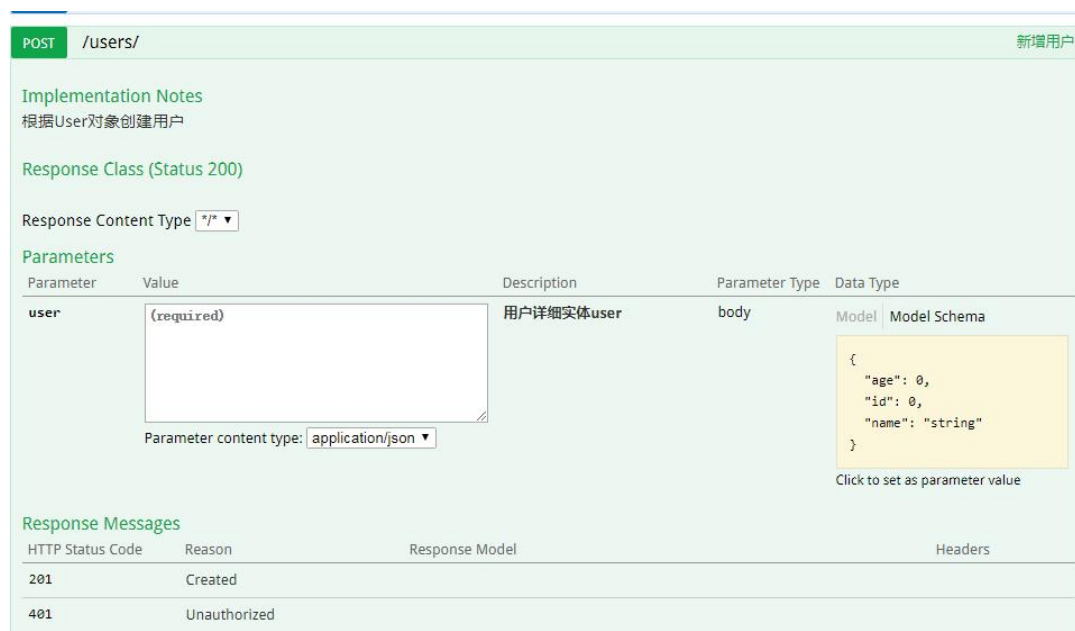


The image shows the Swagger UI interface. At the top, there's a green header with the Swagger logo, a dropdown menu set to 'default (/v2/api-docs)', an input field for 'api_key', and an 'Explore' button. Below the header, the title 'Spring Boot中使用Swagger2构建RESTful APIs' is displayed, followed by 'Created by Sunny'. The main section is titled 'user-controller-jpa : User Controller Jpa' and lists five API endpoints:

Method	Path	Description
GET	/users/	获取全部用户信息
POST	/users/	新增用户
DELETE	/users/{id}	删除指定id用户信息
GET	/users/{id}	获取指定id用户信息
PUT	/users/{id}	更新指定id用户信息

At the bottom, it shows '[BASE URL: / , API VERSION: 1.0]'.

点开每个接口，可以查看接口详情



The image shows the detailed view of the POST /users/ API endpoint. It includes the following sections:

- Implementation Notes:** 根据User对象创建用户
- Response Class (Status 200):** A dropdown menu for 'Response Content Type' set to '*/*'.
- Parameters:** A table with columns: Parameter, Value, Description, Parameter Type, and Data Type.

Parameter	Value	Description	Parameter Type	Data Type
user	(required)	用户详细实体user	body	Model Model Schema

Below the table, there's a 'Parameter content type' dropdown set to 'application/json'.
- Response Messages:** A table with columns: HTTP Status Code, Reason, Response Model, and Headers.

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		

六、SpringBoot Jdbc 操作数据库

1、SpringBoot 开启 jdbc 支持

刚才我们编写完成了针对用户数据的 RESTful API，但是我们是使用集合模拟的数据，在实际工作过程中需要把数据存储到数据库中，接下来我就带领大家来完成 SpringBoot 针对数据库的基本操作。

为了让 SpringBoot 支持 jdbc 数据操作，需要修改 pom.xml
增加所需的 jar

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

jdbc 连接数据库驱动（本次教学采用 MySQL 数据库），需要修改 pom.xml
增加所需的 jar

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

最后修改 src/main/resources/application.yml 中配置数据源信息

```
#数据库jdbc连接url地址,serverTimezone设置数据库时区东八区
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/test?serverTimezone=GMT%2B8
    username: root
    password: 123
    driver-class-name: com.mysql.cj.jdbc.Driver
```

2、编写数据库操作业务接口

```
package com.offcn.service;

import java.util.List;

import com.offcn.po.User;

//用户数据操作业务接口

public interface UserService {

    //获取全部用户数据

    public List<User> getUserList();

    //新增用户数据

    public void createUser(User user);

    //获取指定 id 用户信息

    public User getUser(Long id);

    //更新指定 id 用户信息

    public void updateUser(Long id,User user);

    //删除指定 id 用户

    public void deleteUser(Long id);

}
```

3、编写数据库操作业务实现类

```
package com.offcn.service.impl;

import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;
import org.springframework.stereotype.Service;

import com.offcn.po.User;
import com.offcn.service.UserService;

@Service
public class UserServiceImpl implements UserService {

    //SpringBoot提供的数据库操作类
    @Autowired
    JdbcTemplate jdbcTemplate;

    @Override
    public List<User> getUserList() {
        return jdbcTemplate.query("select * from users", new
        BeanPropertyRowMapper(User.class));
    }

    @Override
    public void createUser(User user) {
        jdbcTemplate.update("insert into
        users(name,age)values(?,?)",user.getName(),user.getAge());
    }

    @Override
    public User getUser(Long id) {
        return (User) jdbcTemplate.queryForObject("select * from user where
        id=?", new BeanPropertyRowMapper(User.class), id);
    }

    @Override
    public void updateUser(Long id, User user) {
        jdbcTemplate.update("update users set name=?,age=? where
        id=?",user.getName(),user.getAge(),id);
    }

    @Override
```

```
public void deleteUser(Long id) {  
    jdbcTemplate.update("delete from users where id=?",id);  
}  
  
}
```

4、编写 Controller

```
package com.offcn.controllerold;  
  
import java.util.List;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.DeleteMapping;  
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.PutMapping;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import com.offcn.po.User;  
  
import com.offcn.service.UserService;  
  
@RestController  
  
@RequestMapping("/users")  
  
public class UserControllerDb {
```

```
@Autowired

UserService userService;

/**
 * 获取全部用户信息
 * @return
 */

@GetMapping("/")

public List<User> getUserList(){

    return userService.getUserList();

}

/**
 * 新增用户
 * @param user
 * @return
 */

@PostMapping("/")

public String createUser(User user) {

    userService.createUser(user);

    return "success";

}

/**
```



```
* 获取指定 id 用户信息

* @param id

* @return

*/

@GetMapping("/{id}")

public User getUser(@PathVariable("id")Long id) {

    return userService.getUser(id);

}

/**

* 更新指定 id 用户信息

* @param id

* @param user

* @return

*/

@PutMapping("/{id}")

public String updateUser(@PathVariable("id") Long id,User user) {

    userService.updateUser(id, user);

    return "success";

}

/**

* 删除指定 id 用户
```

```
* @param id

* @return

*/

@DeleteMapping("/{id}")

public String deleteUser(@PathVariable("id") Long id) {

    userService.deleteUser(id);

    return "success";

}

}
```

5、Postman 测试 RESTful 接口

重新执行测试。

七、SpringBoot 整合 Mybatis 操作数据库

1、SpringBoot 开启 Mybatis 支持

添加 Mybatis 起步依赖以及 mysqljdbc 驱动、连接池 druid 驱动

```
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.0.0</version>
</dependency>
<dependency>
```

```
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.10</version>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
</dependency>
```

2、修改 SpringBoot 配置文件 application.yml

```
#配置数据源
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/test?serverTimezone=GMT%2B8
    type: com.alibaba.druid.pool.DruidDataSource
    username: root
    password: 123
    driver-class-name: com.mysql.jdbc.Driver
#springboot 整合 mybatis
mybatis:
  mapper-locations: classpath:mapper/*.xml
  type-aliases-package: com.offcn.po
```

3、创建实体 bean User

```
package com.offcn.po;

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

@Data
@NoArgsConstructor
@AllArgsConstructor
```

```
public class User {  
    private Long id;  
    private String name;  
    private Integer age;  
}
```

4、创建 Dao 接口,使用 mybatis 注解

```
package com.offcn.dao;  
  
import com.offcn.po.User;  
import org.apache.ibatis.annotations.*;  
  
import java.util.List;  
@Mapper  
public interface UserDao {  
  
    @Insert("insert into user(name, age) values(#{name},#{age})")  
    public void save(User user);  
  
    @Update("update user set name=#{name},age=#{age} where id=#{id}")  
    public void update(User user);  
  
    @Delete("delete from user where id=#{id}")  
    public void delete(Long id);  
  
    @Select("select * from user")  
    public List<User> getAll();  
  
    @Select("select * from user where id=#{id}")  
    public User findOne(Long id);  
}
```

5、创建 Service 接口

```
package com.offcn.service;  
  
import java.util.List;  
import com.offcn.po.User;  
//用户数据操作业务接口
```

```
public interface UserService {  
  
    //获取全部用户数据  
    public List<User> getUserList();  
    //新增用户数据  
    public void createUser(User user);  
    //获取指定 id 用户信息  
    public User getUser(Long id);  
    //更新指定 id 用户信息  
    public void updateUser(Long id, User user);  
    //删除指定 id 用户  
    public void deleteUser(Long id);  
}
```

6、创建 Service 实现类

```
package com.offcn.service.impl;  
  
import com.offcn.dao.UserDao;  
import com.offcn.po.User;  
import com.offcn.service.UserService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
@Service  
public class UserServiceImpl implements UserService {  
  
    @Autowired  
    private UserDao userDao;  
  
    @Override  
    public List<User> getUserList() {  
        return userDao.getAll();  
    }  
  
    @Override  
    public void createUser(User user) {  
        userDao.save(user);  
    }  
}
```

```
}

@Override
public User getUser(Long id) {
    return userDao.findOne(id);
}

@Override
public void updateUser(Long id, User user) {
    user.setId(id);
    userDao.update(user);
}

@Override
public void deleteUser(Long id) {
    userDao.delete(id);
}
}
```

7、创建 Controller

```
package com.offcn.controller;

import com.offcn.po.User;
import com.offcn.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    UserService userService;

    /**
     * 获取全部用户信息
     * @return
     */
}
```

```
*/  
@GetMapping("/")  
public List<User> getUserList() {  
    return userService.getUserList();  
}  
  
/**  
 * 新增用户  
 * @param user  
 * @return  
 */  
@PostMapping("/")  
public String createUser(User user) {  
    userService.createUser(user);  
    return "success";  
}  
  
/**  
 * 获取指定 id 用户信息  
 * @param id  
 * @return  
 */  
@GetMapping("/{id}")  
public User getUser(@PathVariable("id") Long id) {  
  
    return userService.getUser(id);  
}  
  
/**  
 * 更新指定 id 用户信息  
 * @param id  
 * @param user  
 * @return  
 */  
@PutMapping("/{id}")  
public String updateUser(@PathVariable("id") Long id, User user) {  
    userService.updateUser(id, user);  
    return "success";  
}  
  
/**  
 * 删除指定 id 用户  
 * @param id  
 * @return  
 */
```

```
@DeleteMapping("/{id}")
public String deleteUser(@PathVariable("id") Long id) {

    userService.deleteUser(id);
    return "success";

}
}
```

8、修改 SpringBoot 程序主启动类，增加扫描 dao 接口

```
@SpringBootApplication
@MapperScan("com.offcn.dao")
public class DemoApplication {

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

}
```

10、Postman 测试 RESTful 接口

重新执行测试。