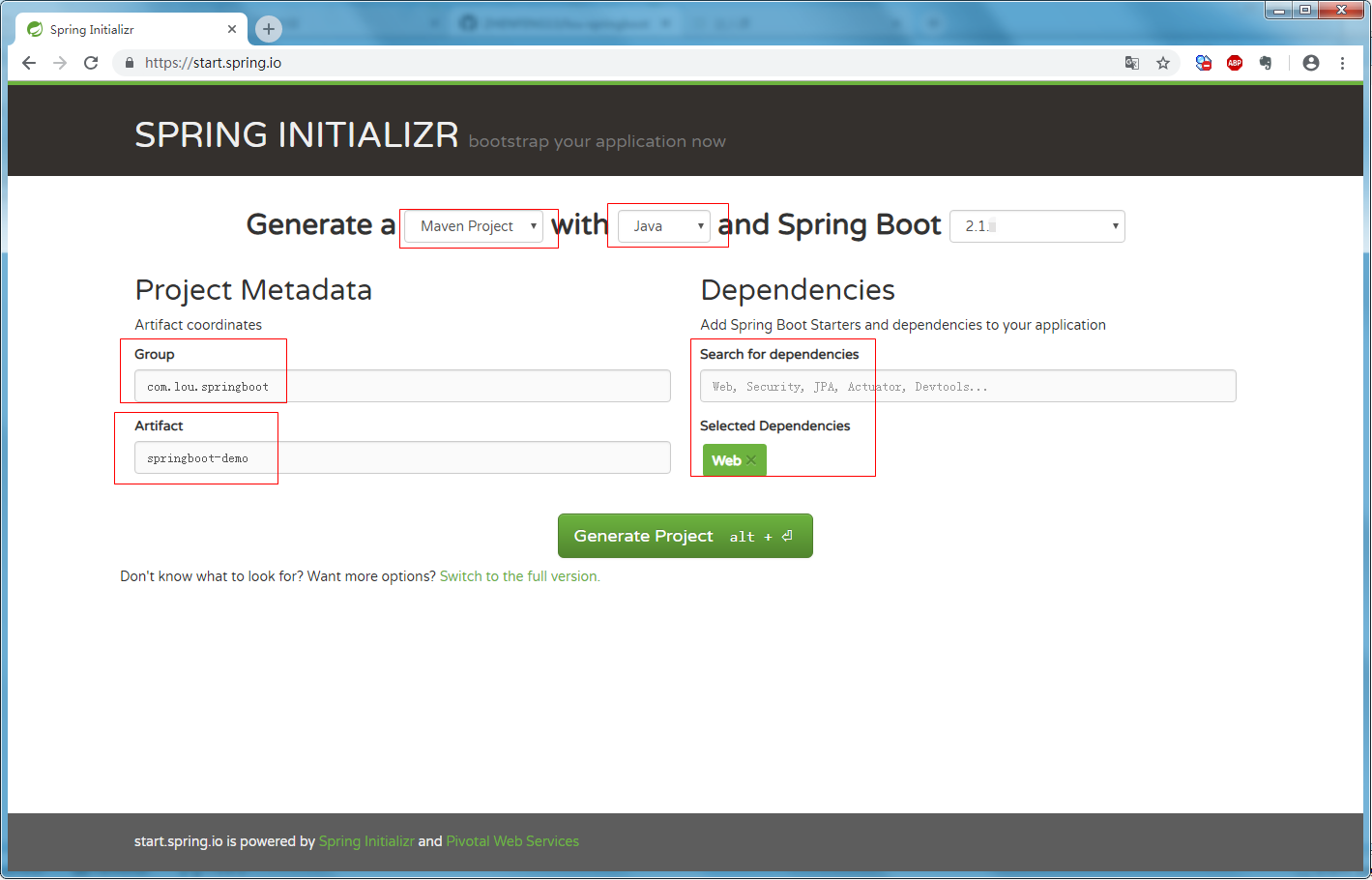
**Spring Boot 项目构建**

Spring 官方提供了 Spring Initializr 来进行 Spring Boot 的快速构建，这是一个在线生成 Spring Boot 基础项目的工具，我们可以将其理解为 Spring Boot 的“创建向导”，接下来我们使用这个在线向导来快速的创建一个 Spring Boot 骨架工程。

* 首先，打开在浏览器中输入 Spring Initializr 的网站地址：[https://start.spring.io](https://start.spring.io/" \t "_blank)
* 之后可以看到页面上需要我们填写和选择项目的基础信息，依次填写即可
* 最后点击“Generate Project”按钮即可获取到一个 Spring Boot 基础项目的代码压缩包



如图所示，Spring Boot 版本我们选择的是最新稳定版本 2.1.0 ，当然也可以选择其他稳定版本，视项目要求而定，“ dependencies ” 表示添加到项目所依赖的 Spring Boot 组件，根据项目要求来选择，需要哪些场景就直接选择相应模块即可，与 SpringBoot Initializr 构建方式类似，也可以多选，本次演示选择了 Web 模块。

**mvn 命令行创建 Spring Boot 项目**

打开命令行并将目录切换到对应的文件夹中，之后运行以下命令：

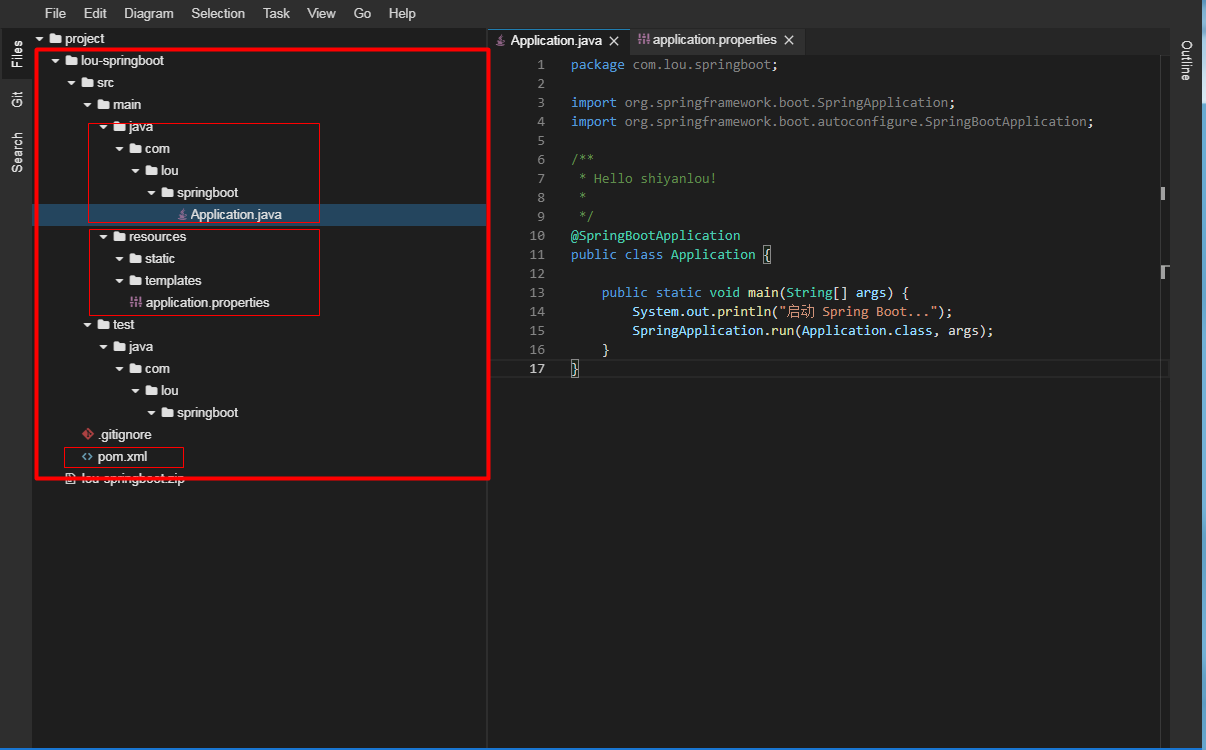
mvn archetype:generate -DinteractiveMode=false -DgroupId=com.lou.springboot -DartifactId=springboot-demo -Dversion=0.0.1-SNAPSHOT

在构建成功后可以生成骨架项目，但是由于生成的项目仅仅是骨架项目，因此 pom.xml 文件中需要自己添加依赖，主方法启动类也需要自行添加，并不是很方便，因此不是特别推荐。

**直接打开**

最后一种方式是直接导入 Spring Boot 项目，如果已经存在 Spring Boot 项目则直接打开，与引入普通的 Maven 工程类似，解压并通过 IDE 打开项目即可(比如 Eclipse、IDEA 或者实验楼的 WebIDE )，导入成功就可以进行 Spring Boot 项目开发了。

打开项目之后可以看到 Spring Boot 项目的目录结构如下：



如上图所示，Spring Boot 的目录结构主要由以下部分组成：

example-springboot

├── src/main/java

├── src/main/resources

├── src/test/java

└── pom.xml

其中 src/main/java 表示 Java 程序开发目录，这个目录大家应该都比较熟悉，唯一的区别是 Spring Boot 项目中还有一个主程序类。

src/main/resources 表示资源文件目录，与普通的 Spring 项目相比有些区别，如上图所示该目录下有 static 和 templates 两个目录，是 Spring Boot 项目默认的静态资源文件目录和模板文件目录，在 Spring Boot 项目中是没有 webapp 目录的，默认是使用 static 和 templates 两个文件夹。

src/test/java 表示测试类文件夹，与普通的 Spring 项目差别不大。

pom.xml 用于配置项目依赖。

以上即为 Spring Boot 项目的目录结构，与普通的 Spring 项目存在一些差异，不过在平常开发过程中，这个差异的影响并不大，说到差别较大的地方可能是部署和启动方式的差异，接下来十三讲详细介绍 Spring Boot 项目的启动方式。

**Spring Boot 项目启动**

**Main() 方法启动**

与普通的 Web 项目相比，Spring Boot 启动项目减少了几个中间步骤，不用去配置 Servlet 容器，也不用打包并且发布到 Servlet 容器再去启动，而是直接运行主方法即可启动项目，开发调试都十分方便也节省开发时间。在你的本机上开发项目时，可以直接在 Eclipse 或者 IDEA 中运行 Spring Boot 主程序类即可，比如现在的项目中有 Application 类，可以直接运行它的 run() 方法，项目就能够正常启动了。

**Maven 插件启动**

由于 pom.xml 文件中引入了 spring-boot-maven-plugin 插件依赖，也可以直接使用 Maven 命令来启动 Spring Boot 项目，插件配置如下。如果 pom.xml 文件中没有该 Maven 插件，是无法通过这种方式启动 Spring Boot 项目的，这一点需要注意。

<build>

<plugins>

<plugin>

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

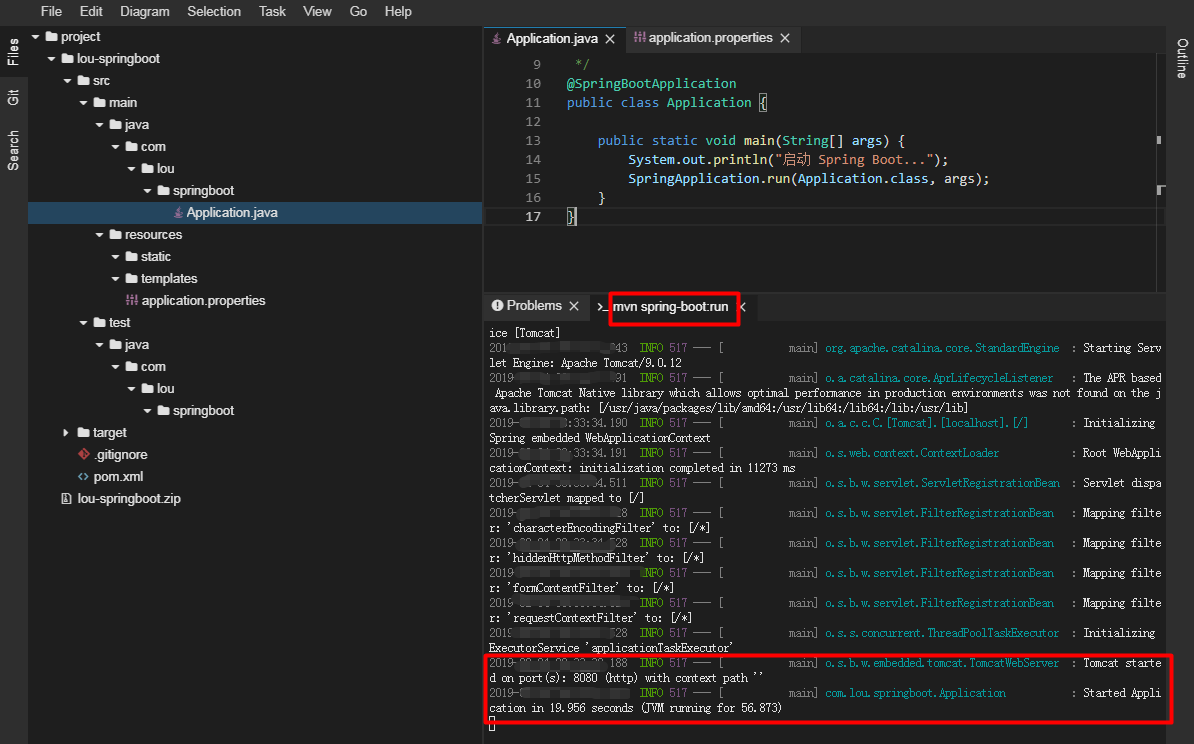
<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

启动过程过程如下图所示，首先点击下方工具栏中的 Terminal 打开命令行窗口，之后在命令行中输入命令 mvn spring-boot：run 并执行该命令即可启动项目，如下图所示，Spring Boot 项目启动成功。



#### java -jar 命令启动

项目初始化时我们选择的打包方式为 Jar ，因此项目开发完成进行打包时的结果是一个 Jar 包， Java 运行 Jar 包的命令为 java -jar xxx.jar ，结合以上两个原因我们可以使用这种方式启动 Spring Boot 项目，接下来我们来演示这一过程。

* 首先，点击下方工具栏中的 Terminal 打开命令行窗口
* 之后使用 Maven 命令将项目打包，执行命令为:mvn clean package -Dmaven.test.skip=true，等待打包结果即可
* 打包成功后进入 target 目录，cd target
* 最后就是启动已经生成的 Jar 包，执行命令为java -jar springboot-demo-0.0.1-SNAPSHOT.jar

这种方式也是 Spring Boot 上线时常用的启动流程，希望不熟悉的朋友都按照以上过程练习几次。

**web项目开发讲解**

自行实现一个 Controller 来测试一下 Spring Boot 如何处理 web 请求，接下来使用 Spring Boot 做一个简单的接口实现。在 src/main/java 目录下新建 com.lou.springboot.controller 包，之后在包里新建一个 HelloController 类，编码如下:

因此我们自行实现一个 Controller 来测试一下 Spring Boot 如何处理 web 请求，接下来使用 Spring Boot 做一个简单的接口实现。在 src/main/java 目录下新建 com.lou.springboot.controller 包，之后在包里新建一个 HelloController 类，编码如下:

package com.lou.springboot.controller;

import org.springframework.stereotype.Controller;

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

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

@Controller

public class HelloController {

@GetMapping("/hello")

@ResponseBody

public String hello() {

return "hello,shiyanlou";

}

}

这段代码大家应该很熟悉，在 Spring Boot 项目中，大部分功能实现的写法与 Spring 项目开发的写法是相同的，这段代码的含义就是处理请求路径为 /hello 的 get 请求，之后返回一段字符串。 编码完成后重新启动项目，点击 WEB 服务按钮，并在浏览器地址栏地址末尾中添加 /hello，如下图所示，咱们的第一个 Spring Boot 项目实例就完成了！

**DispatchServlet 自动配置**

建了第一个 Spring Boot 项目，在初始化时我们选择了 web 模块(即在 pom 文件中引入了 spring-boot-starter-web 依赖)，该模块中包含了 SpringMVC 相关依赖，之后我们开发了第一个 web 功能，仅仅是在控制器类中实现了一个简单的字符串返回方法，并对该方法进行了请求地址映射设置，之后没有做其他配置操作，但是项目启动后就能够进行正常的 web 请求并正常返回数据。 咱们来回想一下在未使用 Spring Boot 开发 web 应用的场景，在平时的 Spring web 项目中，使用 SpringMVC 模块时需要在容器中 SpringMVC 相关的配置 bean，并且在 web.xml 中配置 DispatcherServlet 以及它的 进行请求地址的映射，配置文件如下:

<!--Start spring mvc Servlet-->

<servlet>

<servlet-name>Spring</Servlet-name>

<servletclass>org.springframework.web.Servlet.DispatcherServlet</Servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring-mvc.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<!--End spring mvc Servlet-->

<!--Start Servlet-mapping -->

<servlet-mapping>

<Servlet-name>Spring</Servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<!--End Servlet-mapping -->

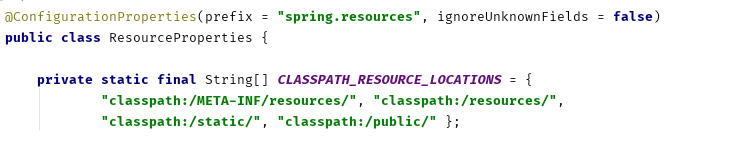
最后启动 Tomcat 服务器装载 DispatcherServlet，接着就可以进行基于 SpringMVC 框架的 web 项目开发和使用，但是 Spring Boot 开发 web 项目时，开发者只需导入 spring-boot-starter-web 场景启动器即可，根本无需再进行任何配置就能够使得 DispatherServlet 正常加载并使用，这是因为 **Spring Boot 在项目启动时已经将 DispatherServlet 自动配置到项目中**了，因此我们无需做多余的配置即可完成接口的开发。

不过呢，使用 Spring Boot 开发 web 应用时，有两个需要注意的配置项：

* **server.port** 表示启动后的端口号，默认是 8080
* **server.servlet.context-path** 项目路径，是构成 url 地址的一部分
* 如果想要修改这两个参数值，可以在 application.properties 配置文件中修改，示例如下：
* server.port=8082
* server.servlet.context-path=/shiyanlou/
* 这样的话，启动后的端口号就不再是 8080 而是 8082 了，在浏览器访问时也需要加上设置的项目路径，比如前面的 hello 请求，路径就变成了 <http://localhost:8082/shiyanlou/hello> ，这一点大家需要注意，如果想要对端口和项目访问路径做修改的话，修改这两个参数就可以了，由于实验楼线上环境只能访问 8080 端口，因此端口修改测试的访问需要在你本机上进行。

**静态资源处理**

静态资源一般指 Web 服务器对应目录中的资源文件，客户端发送请求到 Web 服务器后，Web 服务器（比如 Tomcat）直接从文件目录中获取文件并返回给客户端，客户端解析并渲染显示出来，比如 HTML、CSS、JavaScript、图片等文件，这些文件原样返回给客户端，并不会在 Web 服务器端有所改变，在进行 web 项目开发时，对于这些资源的处理也是十分重要的，这一小节我们将会讲解 Spring Boot 是如何处理这些文件的。



通过查看源码我们可以看到，Spring Boot 默认存放静态资源文件的目录有四个，这些目录都在类路径下，它们分别是：

* **/META-INF/resources/**
* **/resources/**
* **/static/**
* **/public/**

也就是说，在进行 web 功能开发时，如果项目中存在静态资源文件。如HTML、CSS、JavaScript、图片等文件，只要将这些文件放到以上四个文件目录即可正常访问到。接下来，我们新建四个目录并进行测试，类路径对应到项目路径中就是 **src/main/resources** ，新的目录结构如下：

* src/main/resources
* ├── application.properties
* ├── META-INF
* │ └── resources
* ├── public
* ├── resources
* ├── static
* └── templates
* 也就是说，在进行 web 功能开发时，如果项目中存在静态资源文件，如 HTML、CSS、JavaScript、图片等文件，只要将这些文件放到以上四个文件目录即可正常访问到。接下来，我们新建四个目录并进行测试，类路径对应到项目路径中就是 **src/main/resources** ，新的目录结构如下：
* src/main/resources
* ├── application.properties
* ├── META-INF
* │ └── resources
* ├── public
* ├── resources
* ├── static
* └── templates

为了验证，我们在分别在每个目录中新增一个静态资源文件，分别是 /META-INF/resources/test.html 、/resources/shiyanlou.png 、/static/test.css 、/public/test.js :

* 为 test.html 添加如下内容：
* <!DOCTYPE html>
* <html lang="en">
* <head>
* <meta charset="UTF-8" />
* <title>test</title>
* </head>
* <body>
* resource - html
* </body>
* </html
* 下载 shiyanlou.png 到对应目录
* 为 test.css 添加如下内容：
* . test {
* font-size: 14px;
* }
* 为 test.js 添加如下内容：
* function test() {
* console.log('resource - js');
* }

接下来启动项目，启动成功后可以点击页面上方的 Web 服务直接在显示查看网站效果。

结果都是正确的，文件虽然在不同的路径中，但是四个路径都是默认的静态资源目录，因此都能够被访问到。当然，有的同学可能会说了，我就想要访问 shiyanlou 目录下的静态资源文件，这种情况该怎么办呢？其实，前面四个静态资源目录是 Spring Boot 默认设置的，如果想要更改，可以修改配置文件中的 spring.resources.static-locations 参数。比如，我们想要访问 shiyanlou 目录下的静态资源文件，首先需要在src/main/resources下新建 shiyanlou 目录，将静态资源复制到该目录中：

shiyanlou

├── shiyanlou.png

├── test.css

└── test.html

我们复制了三个静态资源文件，test.js 依然放在 static 目录中，之后修改配置文件，添加如下设置：

spring.resources.static-locations=classpath:/shiyanlou/

重启后，在浏览器中再次分别访问这些静态资源，你会发现 shiyanlou 目录下的三个文件都能够正常访问，但是 test.js 访问时报错 404 了，因为配置文件中静态资源路径并没有包含 static 目录，shiyanlou 目录下也没有 test.js。如果想要正常访问到该文件，有两个办法，一是将 test.js 文件放到 shiyanlou 目录下，二是修改配置文件为 ：

spring.resources.static-locations=classpath:/shiyanlou/,classpath:/static/

spring.resources.static-locations 的参数值可以设置为一个数组。这样修改后，重启后又能够访问所有的静态资源文件了。

**消息转换器 HttpMessageConverter**

以往在使用 SpringMVC 框架开发项目时，大家应该都使用过 @RequestBody、@ResponseBody 注解进行请求实体的转换和响应结果的格式化输出。

以普遍使用的 json 数据为例，

RequestBody注解的作用将请求中的数据解析成 json 并绑定为实体对象

ResponseBody 将响应结果以 json 格式返回给请求发起者

但 Http 请求和响应是基于文本的，也就是说在 SpringMVC 内部维护了一套转换机制，也就是我们通常所说的“将 json 格式的请求信息转换为一个对象，将对象转换为 json 格式并输出为响应信息 ”，这些就是 HttpMessageConverter 的作用。

举一个简单的例子，我们定义一个实体类，并通过 @RequestBody、@ResponseBody 注解进行参数的读取和响应，代码如下：

在 com.lou.springboot 下创建如下目录结构：

com.lou.springboot

├── Application.java

├── controller

│ ├── HelloController.java

│ └── TestController.java

├── entity

│ └── SaleGoods.java

└── service

└── HelloService.java

SaleGoods.java 的代码内容如下：

// 实体类

package com.lou.springboot.entity;

public class SaleGoods {

private Integer id;

private String goodsName;

private float weight;

private int type;

private Boolean onSale;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getGoodsName() {

return goodsName;

}

public void setGoodsName(String goodsName) {

this.goodsName = goodsName;

}

public float getWeight() {

return weight;

}

public void setWeight(float weight) {

this.weight = weight;

}

public Boolean getOnSale() {

return onSale;

}

public void setOnSale(Boolean onSale) {

this.onSale = onSale;

}

public int getType() {

return type;

}

public void setType(int type) {

this.type = type;

}

@Override

public String toString() {

return "SaleGoods{" +

"id=" + id +

", goodsName='" + goodsName + '\'' +

", weight=" + weight +

", type=" + type +

", onSale=" + onSale +

'}';

}

}

TestController.java 控制器方法如下，拿到参数数值后进行简单的修改并将对象数据返回：

package com.lou.springboot.controller;

import com.lou.springboot.entity.SaleGoods;

import org.springframework.stereotype.Controller;

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

@Controller

public class TestController {

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

@ResponseBody

public SaleGoods httpMessageConverterTest() {

SaleGoods saleGoods = new SaleGoods();

saleGoods.setGoodsName("华为手机");

saleGoods.setId(1);

saleGoods.setOnSale(true);

saleGoods.setType(1);

saleGoods.setWeight(300);

return saleGoods;

}

编码完成后重启项目，点击 WEB 服务，并在地址栏地址末尾添加 /get/httpmessageconverter 进行测试，结果如下：

返回数据如下：

{

"id": 1,

"goodsName": "华为手机",

"weight": 300.0,

"type": 1,

"onSale": true

}

添加 @ResponseBody 注解后，Spring Boot 会直接将对象转换为 json 格式并输出为响应信息，这是将对象作为响应数据的例子。客户端请求服务端，服务端返回一个实体类，通过reponsebody注解转换为json格式，然后返回给客户端。

接下来我们再写一个案例，使用 @RequestBody 接收前端请求并将参数转换为后端定义的对象，在 TestController 类中添加的方法如下，请求方法为 POST，并使用 @RequestBody 注解将前端传输的参数为json格式，然后转换为实体类。直接转换为 SaleGoods 对象。

@RequestMapping(value = "/test/httpmessageconverter", method = RequestMethod.POST)

@ResponseBody

public SaleGoods httpMessageConverterTest2(@RequestBody SaleGoods saleGoods) {

System.out.println(saleGoods.toString());

saleGoods.setType(saleGoods.getType() + 1);

saleGoods.setGoodsName("商品名：" + saleGoods.getGoodsName());

return saleGoods;

}

由于是 POST 请求，因此没有直接使用浏览器访问，而是在 static 中新增了 api-test.html 页面进行模拟请求,页面当中使用的是 applicaiton/json 格式进行 ajax 请求，resources/static/api-test.html 代码如下：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

<meta http-equiv="X-UA-Compatible" content="IE=edge" />

<title>lou.SpringBoot | 请求测试</title>

</head>

<body class="hold-transition login-page">

<div style="width:720px;margin:7% auto">

<div class="content">

<div class="container-fluid">

<div class="row">

<div class="col-lg-6">

<div class="card">

<div class="card-header">

<h5 class="m-0">接口测试</h5>

</div>

<div class="card-body">

<input id="id" type="text" placeholder="请输入id字段" />

<input

id="goodsName"

type="text"

placeholder="请输入goodsName字段"

/>

<input

id="weight"

type="number"

placeholder="请输入weight字段"

/>

<input id="type" type="number" placeholder="请输入type字段" />

<input

id="onSale"

type="number"

placeholder="请输入onSale字段(0或1)"

/>

<h6 class="card-title">接口返回数据如下：</h6>

<p class="card-text" id="result2"></p>

<a

href="#"

class="btn btn-primary"

onclick="requestPostTest()"

>发送Post请求</a

>

</div>

</div>

<br />

</div>

</div>

</div>

</div>

</div>

<!-- jQuery -->

<script src="https://cdn.staticfile.org/jquery/1.12.0/jquery.min.js"></script>

<script type="text/javascript">

function requestPostTest() {

var onSale = true;

var id = $('#id').val();

var weight = $('#weight').val();

var type = $('#type').val();

var goodsName = $('#goodsName').val();

var onSaleValue = $('#onSale').val();

if (onSaleValue !== 1) {

onSale = false;

}

var data = {

id: id,

goodsName: goodsName,

weight: weight,

type: type,

onSale: onSale,

};

$.ajax({

type: 'POST', //方法类型

dataType: 'json', //预期服务器返回的数据类型

url: '/test/httpmessageconverter',

contentType: 'application/json; charset=utf-8',

data: JSON.stringify(data),

success: function (result) {

$('#result2').html(JSON.stringify(result));

},

error: function () {

$('#result2').html('接口异常，请联系管理员！');

},

});

}

</script>

</body>

</html>

重启项目，在地址栏访问 /api-test.html 页面，并在页面输入框中输入对应的数据，点击请求按钮，最终获得结果如下：



前端 ajax 传输的数据是 5 个字段，到达后端后直接转换为 SaleGoods 对象。由于消息转换器的存在，对象数据的读取不仅简单而且完全正确，响应时也不用自行封装工具类，使得开发过程变得更加灵活和高效。

**总结**

首先对 DispatchServlet 自动配置做一下介绍，让大家知道，在做 web 功能开发时其实与原来使用 SpringMVC 框架开发时类似，只不过 Spring Boot 默认是已经将其配置好而减少我们的工作量，不仅如此，Spring Boot 还做了很多自动配置使得我们能够如此方便的进行 web 项目开发。

通过官方文档的介绍我们可以发现，Spring Boot 还做了如下的默认配置：

* 自动配置了 ViewResolver 视图解析器
* 静态资源文件夹处理
* 自动注册了大量的转换器和格式化器
* 提供了 HttpMessageConverter 对请求参数和返回结果进行处理
* 自动注册了 MessageCodesResolver
* 默认欢迎页配置
* favicon 自动配置