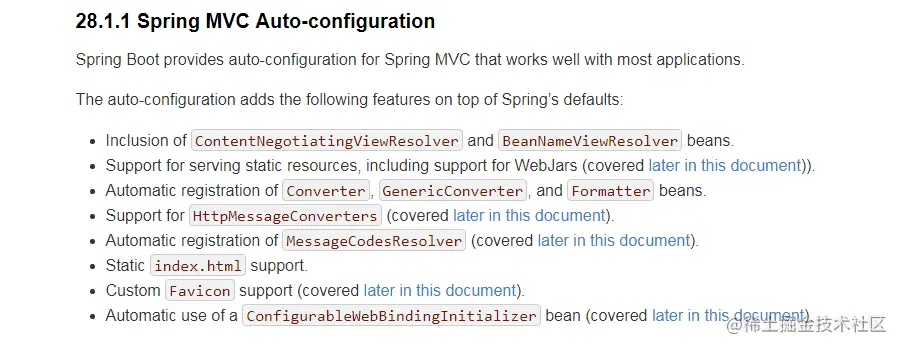
在使用新蜂商城时，很多人会问到关于参数接收和处理的问题，其实这是一个非常简单的知识点，不过有些同学可能只学习了 Spring Boot 框架，而对于其基础框架 Spring 和 Spring MVC 框架的知识点掌握得并不是很牢靠，所以这里我也会简单的介绍一下，首先需要达成一个共识：Spring Boot 框架中的 web 开发模块就是使用了 Spring MVC 框架。

不过与普通的 web 项目开发相比，在 Spring Boot 项目中我们并不需要对 Spring MVC 框架进行配置即可使用该框架的相关功能，这种体验在第 7 讲《Spring Boot 项目初体验》中都有领略到。只是在项目中引用了 web starter，之后就可以进行 Controller 层代码的编写和 web 请求，这是因为 Spring Boot 自动配置机制，Spring MVC 所需的相关类在项目启动过程汇总已经自动配置生效了，下图就是 Spring Boot 官方文档中关于 Spring MVC 自动配置的简介：



通过官方文档的介绍我们可以发现，Spring Boot 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

以上自动配置都是在 WebMvcAutoConfiguration 自动配置类中操作的，接下来我们将结合源码和小案例对以上知识点进行介绍和讲解。

## 自动注册 Converter 、Formatter

在 WebMvcAutoConfigurationAdapter 内部类中，含有 addFormatters() 方法，该方法会向 FormatterRegistry 添加 IOC 容器中所有的 Converter、GenericConverter、Formatter 类型的 bean。

**@Override**

**public** **void** **addFormatters**(FormatterRegistry registry) {

**for** (Converter<?, ?> converter : getBeansOfType(Converter.class)) {

registry.addConverter(converter);

}

**for** (GenericConverter converter : getBeansOfType(GenericConverter.class)) {

registry.addConverter(converter);

}

**for** (Formatter<?> formatter : getBeansOfType(Formatter.class)) {

registry.addFormatter(formatter);

}

}

为了方便大家理解，我们简单的举一个小例子，新建 TestController 类并新增 typeConversionTest() 方法，参数分别为 goodsName(String 类型)、weight(float类型)、type(int类型)、onSale(Boolean类型)，代码如下：

**@RestControllerpublic** **class** **TestController** {

**@RequestMapping("/test/type/conversion")**

**public** **void** **typeConversionTest**(String goodsName, **float** weight, **int** type, Boolean onSale) {

System.out.println("goodsName:" + goodsName);

System.out.println("weight:" + weight);

System.out.println("type:" + type);

System.out.println("onSale:" + onSale);

}

}

之后我们重启项目并在浏览器中输入地址进行请求，看一下打印结果是怎么样的。

第一次请求：http://localhost:8080/test/type/conversion?goodsName=iPhoneX&weight=174.5&type=1&onSale=true

打印结果：

goodsName:iPhoneXweight:174.5type:1onSale:true

第二次请求：http://localhost:8080/test/type/conversion?goodsName=iPhone8&weight=174.5&type=2&onSale=0 打印结果：

goodsName:iPhone8weight:174.5type:2onSale:false

其实这就是 SpringMVC 中的类型转换，Http 请求传递的数据都是字符串 String 类型的，上面这个方法在 Controller 中定义，如果该方法对应的地址接收到到浏览器的请求的话，并且请求中含有 goodsName(String 类型)、weight(float类型)、type(int类型)、onSale(Boolean类型) 参数且都已经被进行正确的类型转换了，如果参数无法通过 String 强转的话也会报错，这就是文章中提到的 MessageCodesResolver 了，朋友们可以自行多测试几次。

以上是简单的类型转换，如果业务需要的话也可以进行自定义类型转换器添加到项目中。

## 消息转换器 HttpMessageConverter

HttpMessageConverter 的设置也是通过 WebMvcAutoConfigurationAdapter 完成的，源码如下：

**@Override**

**public** **void** **configureMessageConverters**(List<HttpMessageConverter<?>> converters) {

this.messageConvertersProvider.ifAvailable((customConverters) -> converters

.addAll(customConverters.getConverters()));

}

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，这两个注解的作用分别可以将请求中的数据解析成 json 并绑定为实体对象以及将响应结果以 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：

*// 实体类***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 +

'}';

}

}

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：

**@RestControllerpublic** **class** **TestController** {

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

**public** SaleGoods **httpMessageConverterTest**(**@RequestBody** SaleGoods saleGoods) {

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

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

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

**return** saleGoods;

}

}

代码中并没有 @ResponseBody 注解，是因为 @RestController 注解是一个组合注解，包含 @ResponseBody 注解。

编码完成后重启项目，并发送请求数据进行测试，请求数据如下：

{

"id":1,

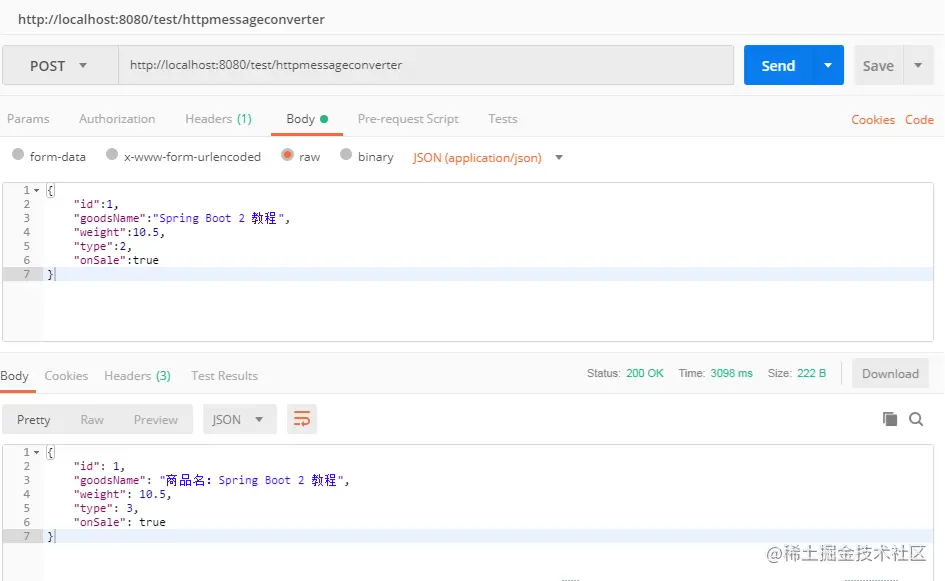
"goodsName":"Spring Boot 2 教程",

"weight":10.5,

"type":2,

"onSale":**true**}

由于是 POST 请求，因此没有直接使用浏览器访问，而是使用 postman 进行模拟请求，最终获得结果如下：

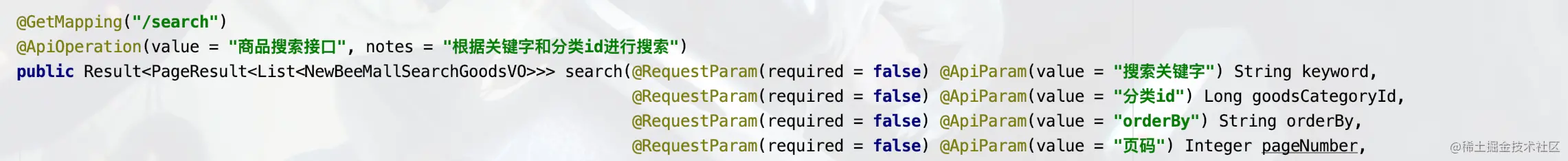


由于消息转换器的存在，对象数据的读取不仅简单而且完全正确，响应时也不用自行封装工具类，使得开发过程变得更加灵活和高效。

## 新蜂商城中的实际应用

讲完了 Spring MVC 对于参数和返回结果的处理，接下来大家跟着我的思路，我们一起来看一下在新蜂商城项目中又是怎样去运用这两个知识点，怎样去处理参数接收和结果返回的。

* 普通参数接收



截图为商品搜索功能，这里的参数接收方式和前文中讲到的 demo 类似，由于是 GET 请求方式，所以传参时直接在路径后拼接参数和参数值即可，格式为：

?key1=value1&key2=value2

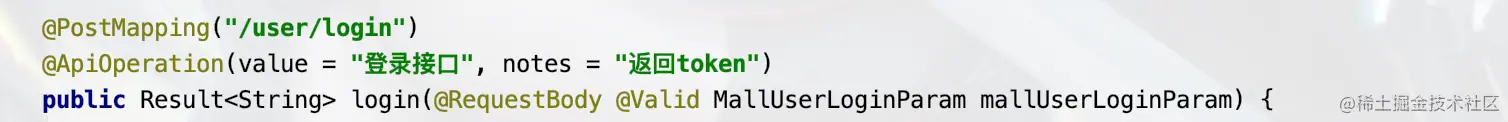
* 路径参数接收



新蜂商城项目的部分接口在设计时也采用了这种将参数拼入路径中的方式，比如订单详情接口，如果我们想要查询订单号为 2072843 的订单，则直接请求 /order/2072843 路径即可，代码中使用 @PathVariable 注解来进行接收。

当然也可以设计为普通参数接收的形式，比较类似，都是简单类型的参数。

* 对象参数接收



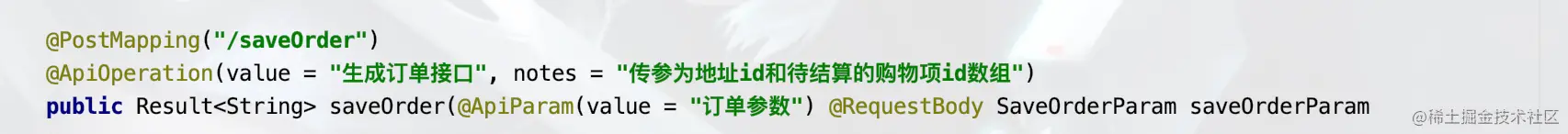
新蜂商城项目中的 POST 或者 PUT 类型的请求方式，基本都是以对象形式来接收参数，前端在请求 Body 中放入 json 格式的请求参数，后端则使用 @RequestBody 注解进行接收，并将这些参数转换为对应的实体类。

为了传参形式的统一，POST 或者 PUT 类型的请求参数，前端传过来的格式要求为 json 形式，Content-Type 统一设置为 application/json。

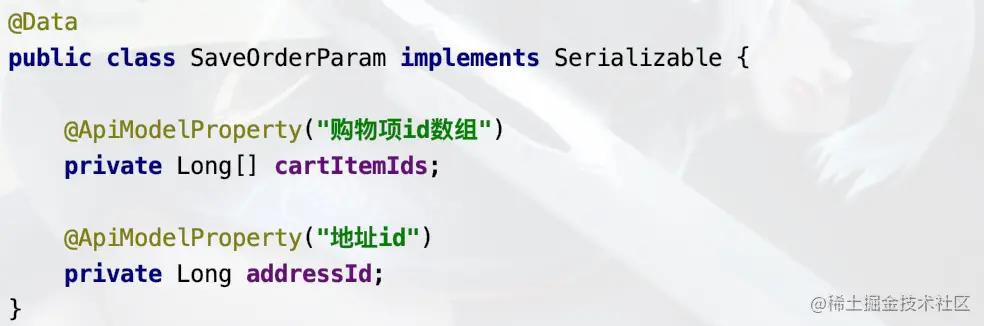
* 复杂对象接收

当然，有时也会出现复杂对象传参的处理，比如一个对象中包含另外一个对象，这种也与对象参数接收的方式一样，只是在 json 串中对加一层对象即可。

这里我以订单生成接口的传参来介绍：



前端需要将用户选勾选的购物项 id 数组和收货地址的 id 传过来，后端的处理逻辑类似，使用 @RequestBody 注解进行接收和对象转换即可。



前端传参时需要注意一下 json 格式，比如这个接口的传输参数就可以是：

{

"addressId": 0,

"cartItemIds": [

1,2,3

]}

## 统一结果响应

新蜂商城项目中使用统一的结果响应对象来处理请求的数据返回，这样做的好处是可以保证所有接口响应数据格式的统一，大大地减少接口响应的工作量和避免接口应答的不统一而造成的开发问题，以新蜂商城中的功能模块举例，有些接口需要返回简单的对象，比如字符串或者数字；有些接口需要返回一个复杂的对象，比如用户详情接口、商品详情接口，这些接口就需要返回不同的对象；有些接口又需要返回列表对象或者分页数据，这些对象又复杂了一些。

我们首先将返回结果进行抽象并封装。

新建 util 包，并封装 Result 结果类，代码如下（注：代码位于 ltd.newbee.mall.util）：

**package** ltd.newbee.mall.util;

**import** io.swagger.annotations.ApiModelProperty;

**import** java.io.Serializable;

**public** **class** **Result**<T> **implements** **Serializable** {

**private** **static** **final** **long** serialVersionUID = 1L;

*//业务码，比如成功、失败、权限不足等 code，可自行定义*

**@ApiModelProperty("返回码")**

**private** **int** resultCode;

*//返回信息，后端在进行业务处理后返回给前端一个提示信息，可自行定义*

**@ApiModelProperty("返回信息")**

**private** String message;

*//数据结果，泛型，可以是列表、单个对象、数字、布尔值等*

**@ApiModelProperty("返回数据")**

**private** T data;

**public** **Result**() {

}

**public** **Result**(**int** resultCode, String message) {

this.resultCode = resultCode;

this.message = message;

}

**public** **int** **getResultCode**() {

**return** resultCode;

}

**public** **void** **setResultCode**(**int** resultCode) {

this.resultCode = resultCode;

}

**public** String **getMessage**() {

**return** message;

}

**public** **void** **setMessage**(String message) {

this.message = message;

}

**public** T **getData**() {

**return** data;

}

**public** **void** **setData**(T data) {

this.data = data;

}

**@Override**

**public** String **toString**() {

**return** "Result{" +

"resultCode=" + resultCode +

", message='" + message + '\'' +

", data=" + data +

'}';

}

}

每一次后端数据返回都会根据以上格式进行数据封装，包括业务码、返回信息、实际的数据结果，而不是像前一个实验中的不确定格式，前端接受到该结果后对数据进行解析，并通过业务码进行相应的逻辑操作，之后再将 data 中的数据获取到并进行页面渲染或者进行信息提示。

实际返回的数据格式如下：

* 列表数据

{

"resultCode": 200,

"message": "SUCCESS",

"data": [{

"id": 2,

"name": "user1",

"password": "123456"

}, {

"id": 1,

"name": "13",

"password": "12345"

}]}

* 单条数据

{

"resultCode": 200,

"message": "SUCCESS",

"data": **true**}

如上两个分别是列表数据和单条数据的返回，后端进行业务处理后将会返回给前端一串 json 格式的数据，resultCode 等于 200 表示数据请求成功，该字段也可以自行定义，比如 0、1001、500 等等，message 值为 SUCCESS，也可以自行定义返回信息，比如“获取成功”、“列表数据查询成功”等，这些都需要与前端约定好，一个码只表示一种含义，而 data 中的数据可以是一个对象数组、也可以是一个字符串、数字等类型，根据不同的业务返回不同的结果，之后的实践内容里都会以这种方式返回数据。