Sprint boot

从根本上看，是各种库的集合，可以被构建系统（Maven）使用

核心：

1. 自动配置，不用手动配置xml文件 2.起步依赖3.命令行界面4.Actuator，监控运行状态

的应用

解决了问题

1. 搭建简单2.配置简单3.编码简单4.部署简单，内置tomcat服务，不用在配置了；可包jar包，直接用java运行5.监控简单

能干什么

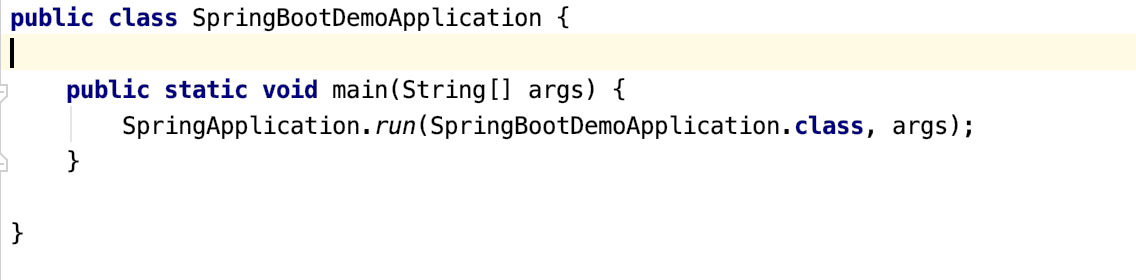
1. 基于模版的Web应用开发 2.RESTful服务开发，前后端分离3.微服务

**Spring Boot 入门**

## 一、项目框架构建

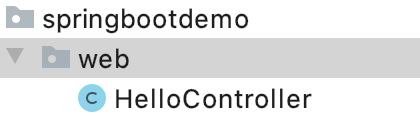
### 1.Spring boot 三种启动方式

1. java类文件启动，直接在运行下图所示的类



1. 在命令行输入mvn spring-boot:run启动
2. mvn package 命令将项目打包，然后进入到target目录，输入java -jar 对应的jar包启动

## 2.hello world

在新建一个web包，在这个包内新建一个java类作为控制器

**package** com.zph.springbootdemo.web;  
  
**import** org.springframework.web.bind.annotation.RequestMapping;  
**import** org.springframework.web.bind.annotation.RestController;  
  
@RestController  
**public class** HelloController {  
 @RequestMapping(**"/say"**)  
 **public** String hello(){  
 **return "Hello Spring Boot"**;  
 }  
}

## URL路由

### 1、RESTful URL

一个URL，代表一个资源的状态转换

GET htp://localhost:8080/api/v1/books 获取读书清单列表

POST htp://localhost:8080/api/v1/books 新增一个清单

GET htp://localhost:8080/api/v1/books/{id} 获取一条读书清单

PUT htp://localhost:8080/api/v1/books 更新一个读书清单

DELETE htp://localhost:8080/api/v1/books/{id} 删除一个读书清单

阮一峰 了解RESTful的相关信息

### 2、@RestController 注解

### 3、@RequestMapping 注解

完成了一个请求映射的作用，使对应的方法/类能处理url的请求

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

**public** String hello(){  
 **return "Hello Spring Boot"**;  
}

然后hello方法只能处理GET的请求，如果是其他类型如POST等的话就无法接收到

### 4、@RequestMapping 简写形式

如果想要接受得失post请求，直接注解可以直接改成@PostMapping(“/say”),其他类同

### 5、@Controller 注解

报错，显示没有定义视图（View），返回的字符串应该要对应一个模版，如果没找到这个模版就会抛出异常,这时候需要去pom.xml添加项目名spring-boot-starter-thymeleaf，然后在resources的模版里创建该视图对应的模版，可以是html的形式的。

和RestController控制器不同，用于模版开发，返回html页面。但是如果有的映射想返回字符串或者实体，可以在方法上添加@ResponseBody 效果和@RestController一样了。

## 三、参数处理

### @PathVariable

获取一条资源，或者改变一个资源

要通过如下URL的方式获取一条读书清单

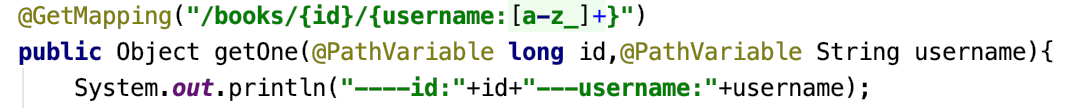
GET htp://localhost:8080/api/v1/books/{id} 获取一条读书清单

前面的类需要映射/api/v1，此方法就用如下方式，如果不加@PathVariable 会提示找不到id，如果方法里使用的id和GetMapping中的id名字不一样，那么需要在PathVariable里传入参数id，那么后面的id都可以改成其他名字，因为形成了映射。

@GetMapping(**"/books/{id}"**)  
**public** Object getOne(@PathVariable(**"id"**) **long** id){  
 System.***out***.println(**"----id:"**+id);  
 Map<String,Object> book=**new** HashMap<>();  
 book.put(**"name"**,**"nihao"**);  
 book.put(**"author"**,**"傻子"**);  
 **return** book;  
}

**注意**

1.如果前端有多个参数，那么，如下图所示，在那个参数前也要加@PathVariable



2.参数的顺序可以前后改变{id}的位置可以放到books的前面，当然对应的URL也要改变

3.可以使用正则表达式过滤无效的URL

### @RequestParam

提交一个表单，过滤查询

## 自定义配置

### 配置文件

Application.yml和application.properties都可以配置项目的相关属性，其中.yml是json格式的超类

### 自定义属性配置

在配置文件中直接制定属性的内容，然后在Controller类文件中通过注解@Value(${})直接获取

@Value(**"${book.name}"**)  
**private** String **name**; *//这个name和上面的name可以不一样*

如果属性很多，可以用实体对应属性

创建一个实体类，类里包括application.proprieties的自定义属性，记住要设置这些属性的setter，否则会报错，提示无法绑定。在类前加入下面两个注解

@Component  
@ConfigurationProperties(prefix = **"book"**)

分别用于激活ConfigurationProprieties 和，通过prefix定义外部属性进行配置。

### 多环境配置

自定义的配置文件，（生产环境，测试环境，开发环境等）

application-{}.yml

在application.yml中可以定义让哪个环境配置处于active状态，那么之后就配置那个环境配置（如端口号，logging的级别等）。

**spring**:  
 **profiles**:  
 **active**: dev *#可以指定哪个application是活跃的，直接写-后面的就可以，不需要写全程*

也可以将整个项目打包 然后使用 java -jar {jar包的名字} --spring.profile.active={要配置的环境名字}

## JPA数据库操作

### 配置

JPA（Java Persistence API），定义了ORM以及实体对象持久化的标准接口。Hibernate实现了JPA的一个ORM框架

JPA是一个接口规范，不是框架

Spring Data JPA 是Spring基于ORM框架、JPA规范的基础上封装的一套JPA应用框架

前提是在pom.xml中添加依赖jpa和jdbc（mysql-connector-java)，在开发环境的配置文件中，配置jdbc和jpa，  
**spring**:  
 **datasource**:  
 **driver-class-name**: com.mysql.jdbc.Driver  
 **url**: jdbc:mysql://localhost:3306/book?useUnicode=true&characterEncoding=utf-8  
 **username**: root  
 **password**: zhupeihao123  
 **jpa**:  
 **hibernate**:  
 **ddl-auto**: update #每次部署的时候是在原来数据库的基础上更新，create的话就每次都新创建表  
 **show-sql**: **true**

### 基本查询

在domain中新增一个Repository接口，该接口继承JpaRepository，然后创建一个service包，在该类中有一个刚才创建的Repository接口的对象，通过这个对象实现各种增删改查的服务。通过Controller中的类来调用服务层

注意：

这里在Controller中如果要新增一个书单，或者更新一个书单，需要

**public** Book update(@RequestParam **long** id,  
 @RequestParam String name,  
 @RequestParam String author,  
 @RequestParam String description,  
 @RequestParam **int** status)

如果觉得参数过多很麻烦，可以直接在括号里传入对象update(Book book) ，jpa会自动注入对象？？？

### 复杂查询

除了通过主键id，也可以通过其它方式进行查询，可以直接在Repository类里进行定义，JPA会自动创建相应的SQL语句，如：

List<Book> findByAuthorAndStatus(String author,int status);

List<Book> findByDescriptionEndsWith(String des);

等。

### JPA自定义查询

可以在Repository接口中使用如下语句，实现sql中的查询效果。

@Query(value = **"select** *\** **from book where LENGTH(name)>?1"**,nativeQuery =**true**)  
*// Book是实体类的名称 ?是占位符，后面的1是下面函数的第一个参数  
//@Query("select b from Book b where length(b.name) >?1 ")*

List<Book> findByJPQL(**int** len);

可以实现多表的关联查询

### 自定义更新

实现更新和删除时，也要使用JPQL的注释

@Modifying  
@Query(**"update Book b set b.status=?1 where b.id=?2"**)  
**int** updateByJPQL(**int** statue, **long** id);

除此之外，还要添加@Transactional ，如果是加在Repository中的某个update，那么就是作为原子操作，如果是加在service中的某个更新函数里，就是作为事务操作。如果不加这个注解就会有报错。

### 事务

service中如果某个方法是事务，那么，只有在这个方法都成功执行时，才能在数据库中执行，否则，会回退。

## 使用Thymeleaf模版

### 使用Thymeleaf3

### 取值

在Controller器中要将待显示的对象传到model中

Book book=**bookService**.getOne(id);  
**if**(book==**null**)  
 book=**new** Book();

*//把对象传到模版对象上*model.addAttribute(**"book"**,book);

在Thymeleaf3的模版中,在<span th:text=EL表达式>

<**p**>  
 <**strong**>书名：</**strong**><**span th:text="${book.name}"**>我们三</**span**>  
 </**p**>  
 <**p**>  
 <**strong**>作者：</**strong**><**span th:text="${book.author}"**>杨绛</**span**>  
 </**p**>  
 <**p**>  
 <**strong**>说明：</**strong**><**span th:text="${book.description}"**>不错</**span**>  
 </**p**>  
 <**p**>  
 <**strong**>状态：</**strong**><**span th:text="${book.status}"**>0</**span**>  
 </**p**>

### 静态资源处理

### 判断

**switch的判断，当状态为0，1，2时分别显示不同的内容**

<**p th:switch="\*{status}"**>  
 <**strong**>状态：</**strong**>  
 <**span th:case="0"**>想读</**span**>  
 <**span th:case="1"**>在读</**span**>  
 <**span th:case="2"**>已读</**span**>  
</**p**>

**If，unless判断，这两个时相反的，unless是if取非。**

</**div**>  
 <**div class="alert alert-success" th:unless="\*{status==0}"**>  
 <**strong**>不错</**strong**>,你已经开始行动啦  
 </**div**>  
 <**div class="alert alert-warning" th:if="\*{status==0}"**>  
 <**strong**>注意</**strong**>,你还没开始行动  
 </**div**>  
</**div**>

### 迭代

<**tr th:each="book,iterStat:${books}" th:class="bg-success" th:if="\*{iterStat.even}"**>  
 <**td th:text="${iterStat.count}"**>1</**td**>  
 <**td th:text="${book.name}"**>书名</**td**>  
 <**td th:text="${book.author}"**>作者</**td**>  
 <**td th:text="${book.description}"**>说明</**td**>  
 <**td th:text="${book.status}"**>状态</**td**>  
</**tr**>



<**table class=" table table-bordered"**>  
 <**thead**>  
 <**tr**>  
 <**th**>#</**th**>  
 <**th**>书名</**th**>  
 <**th**>作者</**th**>  
 <**th**>说明</**th**>  
 <**th**>状态</**th**>  
 </**tr**>  
 </**thead**>  
 <**tbody**>  
 *<!--  
 count:计数功能  
 index:从0开始的索引  
 size:当前获取的列表所有的元素条数  
 even/odd :  
 -->* <**tr th:each="book,iterStat:${books}" th:class="\*{iterStat.even}?'bg-success':'bg-danger'"**>  
 <**td th:text="${iterStat.count}"**>1</**td**>  
 <**td th:text="${book.name}"**>书名</**td**>  
 <**td th:text="${book.author}"**>作者</**td**>  
 <**td th:text="${book.description}"**>说明</**td**>  
 <**td th:text="${book.status}"**>状态</**td**>  
 </**tr**>  
 </**tbody**>  
</**table**>

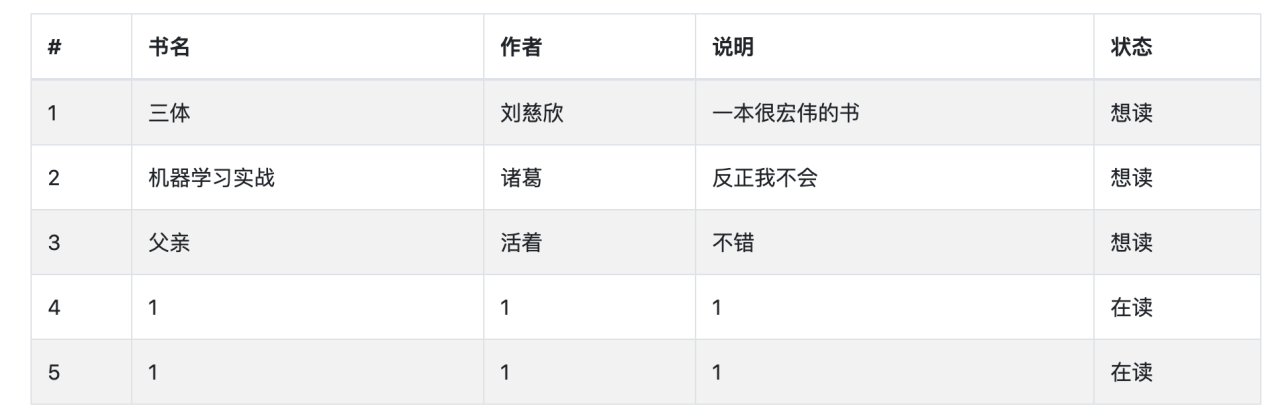


<**table class=" table table-bordered table-hover table-striped"** >

Table-hover是鼠标移动上去可以改变颜色

Table-striped 两行之间有颜色差异

Table-bordered 多了垂直的分割线和边框



### URL

实现在一个书单列表中点击书名，跳转到该书名id的页面中，所以要获得该书名的id。有两种方式：

<**td** ><**a href="#" th:text="${book.name}" th:href="@{/books/{id}(id=${book.id})}"**>书名</**a**></**td**>

<**td** ><**a href="#" th:text="${book.name}" th:href="@{'/books/'+${book.id}}"**>书名</**a**></**td**>

第二种是用字符串相加获得超链接地址

## 表单处理

### Post请求和Redirect返回

用post添加一个书单信息以后，需要直接显示所有书单列表，所有在客服端想服务器发送post请求以后，服务器在处理完请求后通知客服端访问重定向后的url。

**return "redirect:/books"**;

对于无参数传递的形式，可以如上面代码所示

提交的页面如下所示，<form> </form>是一个表单 <input>中的name 必须和实体类中的名称一致，

<**form action="#" th:action="@{/books}" th:object="${book}" method="post" style="background-color**: **rgba**(237,210,245,0.09)**"**>  
 <**input type="hidden" th:value="\*{id}" name="id"**>  
 <**div class="form-group "**>  
 <**label for="nameId"**>书名</**label**>  
 <**input type="text" name="name" id="nameId" class="form-group" size="60" th:field="\*{name}"**>  
 </**div**>  
 <**div class="form-group "** >  
 <**label for="authorId"**>作者</**label**>  
 <**input type="text" name="author" id="authorId" class="form-group" size="60"th:field="\*{author}"**>  
 </**div**>  
  
 <**div class="form-group"**>  
 <**label for="descriptionId"**>说明</**label**>  
 <**input type="text" name="description" id="descriptionId" class="form-group" size="60" th:field="\*{description}"**>  
 </**div**>  
 <**div class="form-group"**>  
 <**label for="statusId"**>状态</**label**>  
 <**select name="status" id="statusId"**>  
 <**option value="0" th:selected="(\*{status}=='0')"**>想读</**option**>  
 <**option value="1" th:selected="(\*{status}=='1')"**>在读</**option**>  
 <**option value="2" th:selected="(\*{status}=='2')"**>已读</**option**>  
 </**select**>  
 </**div**>  
  
 <**p class="text-center"**>  
 <**button type="submit" class="btn btn-primary"**>提交</**button**>  
 </**p**>  
</**form**>

### 初始化表单数据

### 3.flash attribute

*/\*\*  
 \* POST --->redirect--->GET  
 \* model只能在一个request中起作用，但是redirect会前后各产生一次request  
 \* 那么在post中添加的信息，在get的model是找不到的  
 \*  
 \*/*

@PostMapping(**"/books"**)  
**public** String post(Book book, RedirectAttributes redirectAttributes) {  
 Book book1 = **bookService**.postOne(book);  
 **if**(book1 != **null**) {  
 redirectAttributes.addFlashAttribute(**"message"**,**"《"**+book1.getName()+**"》获取成功"**);  
}  
 *//redirect* **return "redirect:/books"**;  
}

在book.html中获取这个message

<**div class=" alert alert-success " th:unless="${#strings.isEmpty(message)}"**>  
 <**a href="#" class="close" data-dismiss="alert"**>**&times;**</**a**>  
 <**strong**>恭喜,</**strong**>  
 <**span th:text="${message}"**>信息提交成功</**span**>  
</**div**>

## 分页处理

### 分页查询

*//进行分页*Page<Book>findAll(Pageable pageable);

在Repository中定义 返回Page形式的函数，

在service层中的findAll函数里定义每页的排序方式，分页的规则

Sort sort=Sort.*by*(Sort.Direction.***DESC***,**"id"**);  
Pageable pageable= PageRequest.*of*(0,10,sort);

**return bookRepository**.findAll(pageable);

这里分析一下使用page以后服务器返回到客户端的Json格式，page有content，pageable，totalPages,totalElements,,last,first,pageSize等参数。

{

    "content": [

        {

            "id": 17,

            "name": "1",

            "author": "1",

            "description": "1",

            "status": 2

        },

        {

            "id": 16,

            "name": "时间佚名",

            "author": "",

            "description": "",

            "status": 2

        },

        {

            "id": 15,

            "name": "matlab",

            "author": "。。。",

            "description": "",

            "status": 2

        }

    ],

    "pageable": {

        "sort": {

            "sorted": true,

            "unsorted": false,

            "empty": false

        },

        "offset": 0,

        "pageNumber": 0,

        "pageSize": 3,

        "paged": true,

        "unpaged": false

    },

    "totalPages": 4,

    "totalElements": 12,

    "last": false,

    "size": 3,

    "number": 0,

    "sort": {

        "sorted": true,

        "unsorted": false,

        "empty": false

    },

    "numberOfElements": 3,

    "first": true,

    "empty": false

}

### Controller中进行分页查询

可以将在service中定义的pageable 放到Controller中，这样可以方便更改page,size

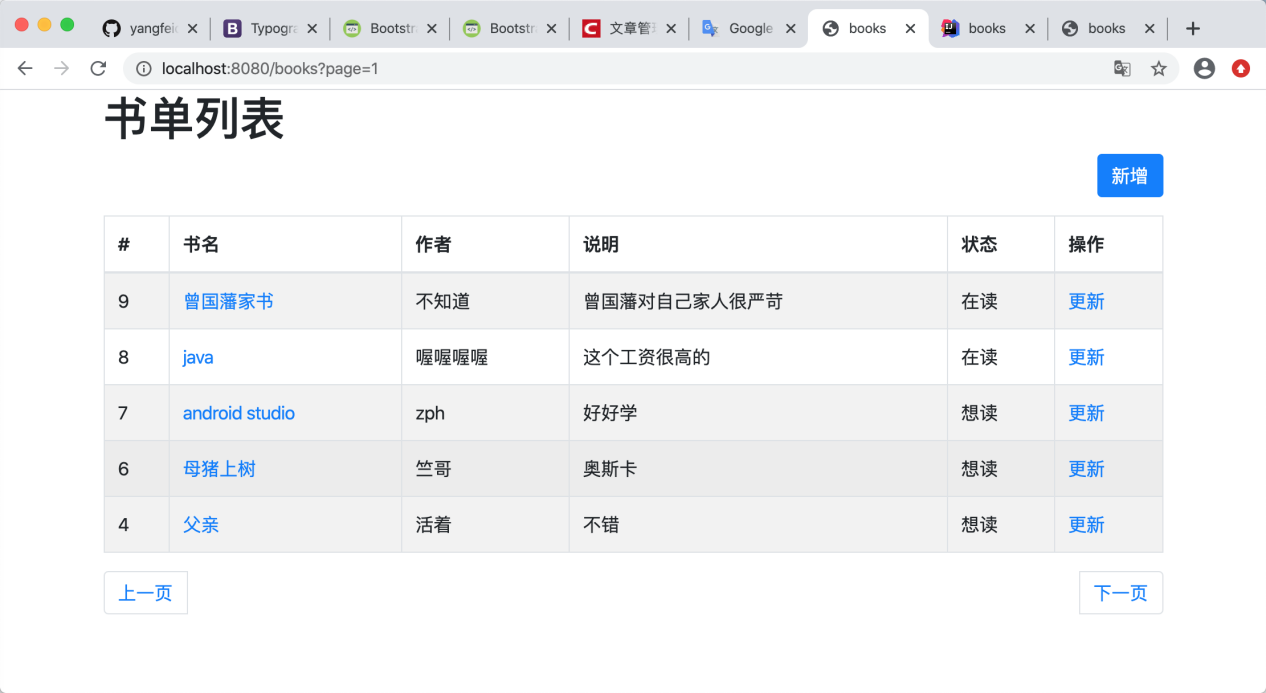
*/\*\*  
 \* 获取书单列表  
 \** ***@param model*** *\** ***@return*** *\*/*@GetMapping(**"/books"**)  
**public** String list(Model model, @RequestParam(defaultValue = **"0"**) **int** page, @RequestParam(defaultValue = **"5"**) **int** size){  
 Sort sort=Sort.*by*(Sort.Direction.***DESC***,**"id"**);  
 Pageable pageable= PageRequest.*of*(page,size,sort);  
 Page<Book> bookpage=**bookService**.findAllByPage(pageable);  
 model.addAttribute(**"page"**,bookpage);  
 **return "books"**;  
}

注意

这里将服务器返回的表单数据渲染到视图中时，表单的循环中，变量不再是books，而是page.content

<**tr th:each="book,iterStat:${page.content}"**>

*<!-- 定义上一页，下一页两个按钮 -->* <**nav**>  
 <**ul class="pagination justify-content-between"**>  
 <**li class="page-item"**><**a class="page-link" href="#" th:unless="${page.number==0}" th:href="@{'/books?page='+${page.number-1}}"**>上一页</**a**></**li**>  
 <**li class="page-item"** ><**a class="page-link" href="#" th:unless="${page.number==page.totalPages-1}" th:href="@{/books(page=${page.number+1})}"**>下一页</**a**></**li**>  
 </**ul**>  
 </**nav**>



### 自动绑定Page参数

@GetMapping(**"/books"**)  
**public** String list(Model model,  
 @PageableDefault(size=5,sort = **"id"**,direction = Sort.Direction.***DESC***) Pageable pageable  
 *//会自动将page，size,sort等参数封装到Pageable中  
 //如果输入的page为-1等，Pageable会自动将其设为0。这样就能防止输入错误，产生报错* ){  
 Page<Book> bookpage=**bookService**.findAllByPage(pageable);  
 model.addAttribute(**"page"**,bookpage);  
 **return "books"**;  
}

## 开发者工具

### 自动重启和自动刷新

<**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 </**dependency**>