

10分钟实现Spring Boot发生邮件功能

yizhiwazi Java后端 2019-11-25

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | yizhiwazi

链接 | www.jianshu.com/p/5eb000544dd7

基础知识

什么是SMTP?

SMTP全称为Simple Mail Transfer Protocol（简单邮件传输协议），它是一组用于从源地址到目的地址传输邮件的规范，通过它来控制邮件的中转方式。SMTP认证要求必须提供账号和密码才能登陆服务器，其设计目的在于避免用户受到垃圾邮件的侵扰。

什么是IMAP?

IMAP全称为Internet Message Access Protocol（互联网邮件访问协议），IMAP允许从邮件服务器上获取邮件的信息、下载邮件等。IMAP与POP类似，都是一种邮件获取协议。

什么是POP3?

POP3全称为Post Office Protocol 3（邮局协议），POP3支持客户端远程管理服务器端的邮件。POP3常用于“离线”邮件处理，即允许客户端下载服务器邮件，然后服务器上的邮件将会被删除。目前很多POP3的邮件服务器只提供下载邮件功能，服务器本身并不删除邮件，这种属于改进版的POP3协议。

IMAP和POP3协议有什么不同呢？

两者最大的区别在于，IMAP允许双向通信，即在客户端的操作会反馈到服务器上，例如在客户端收取邮件、标记已读等操作，服务器会跟着同步这些操作。而对于POP协议虽然也允许客户端下载服务器邮件，但是在客户端的操作并不会同步到服务器上面的，例如在客户端收取或标记已读邮件，服务器不会同步这些操作。

进阶知识

什么是JavaMailSender和JavaMailSenderImpl?

JavaMailSender和JavaMailSenderImpl是Spring官方提供的集成邮件服务的接口和实现类，以简单高效的设计著称，目前是Java后端发送邮件和集成邮件服务的主流工具。

如何通过JavaMailSenderImpl发送邮件？

非常简单，直接在业务类注入JavaMailSenderImpl并调用send方法发送邮件。其中简单邮件可以通过SimpleMailMessage来发送邮件，而复杂的邮件（例如添加附件）可以借助MimeMessageHelper来构建MimeMessage发送邮件。例如：

```
@Autowired  
private JavaMailSenderImpl mailSender;
```

```
public void sendMail() throws MessagingException {  
    //简单邮件  
    SimpleMailMessage simpleMailMessage = new SimpleMailMessage();  
    simpleMailMessage.setFrom("admin@163.com");  
    simpleMailMessage.setTo("socks@qq.com");  
    simpleMailMessage.setSubject("Happy New Year");  
    simpleMailMessage.setText("新年快乐！");  
    mailSender.send(simpleMailMessage);  
  
    //复杂邮件  
    MimeMessage mimeMessage = mailSender.createMimeMessage();  
    MimeMessageHelper messageHelper = new MimeMessageHelper(mimeMessage);  
    messageHelper.setFrom("admin@163.com");  
    messageHelper.setTo("socks@qq.com");  
    messageHelper.setSubject("Happy New Year");  
    messageHelper.setText("新年快乐！");  
    messageHelper.addInline("doge.gif", new File("xx/xx/doge.gif"));  
    messageHelper.addAttachment("work.docx", new File("xx/xx/work.docx"));  
    mailSender.send(mimeMessage);  
}
```

为什么JavaMailSenderImpl 能够开箱即用？

所谓开箱即用其实就是基于官方内置的自动配置，翻看源码可知邮件自动配置类(MailSenderPropertiesConfiguration) 为上文提供了邮件服务实例(JavaMailSenderImpl)。具体源码如下：

```
@Configuration  
@ConditionalOnProperty(prefix = "spring.mail", name = "host")  
class MailSenderPropertiesConfiguration {  
    private final MailProperties properties;  
    MailSenderPropertiesConfiguration(MailProperties properties) {  
        this.properties = properties;  
    }  
    @Bean  
    @ConditionalOnMissingBean  
    public JavaMailSenderImpl mailSender() {  
        JavaMailSenderImpl sender = new JavaMailSenderImpl();  
        applyProperties(sender);  
        return sender;  
    }
```

其中MailProperties是关于邮件服务器的配置信息，具体源码如下：

```
@ConfigurationProperties(prefix = "spring.mail")  
public class MailProperties {  
    private static final Charset DEFAULT_CHARSET = StandardCharsets.UTF_8;  
    private String host;  
    private Integer port;  
    private String username;  
    private String password;  
    private String protocol = "smtp";  
    private Charset defaultEncoding = DEFAULT_CHARSET;  
    private Map<String, String> properties = new HashMap<>();  
}
```

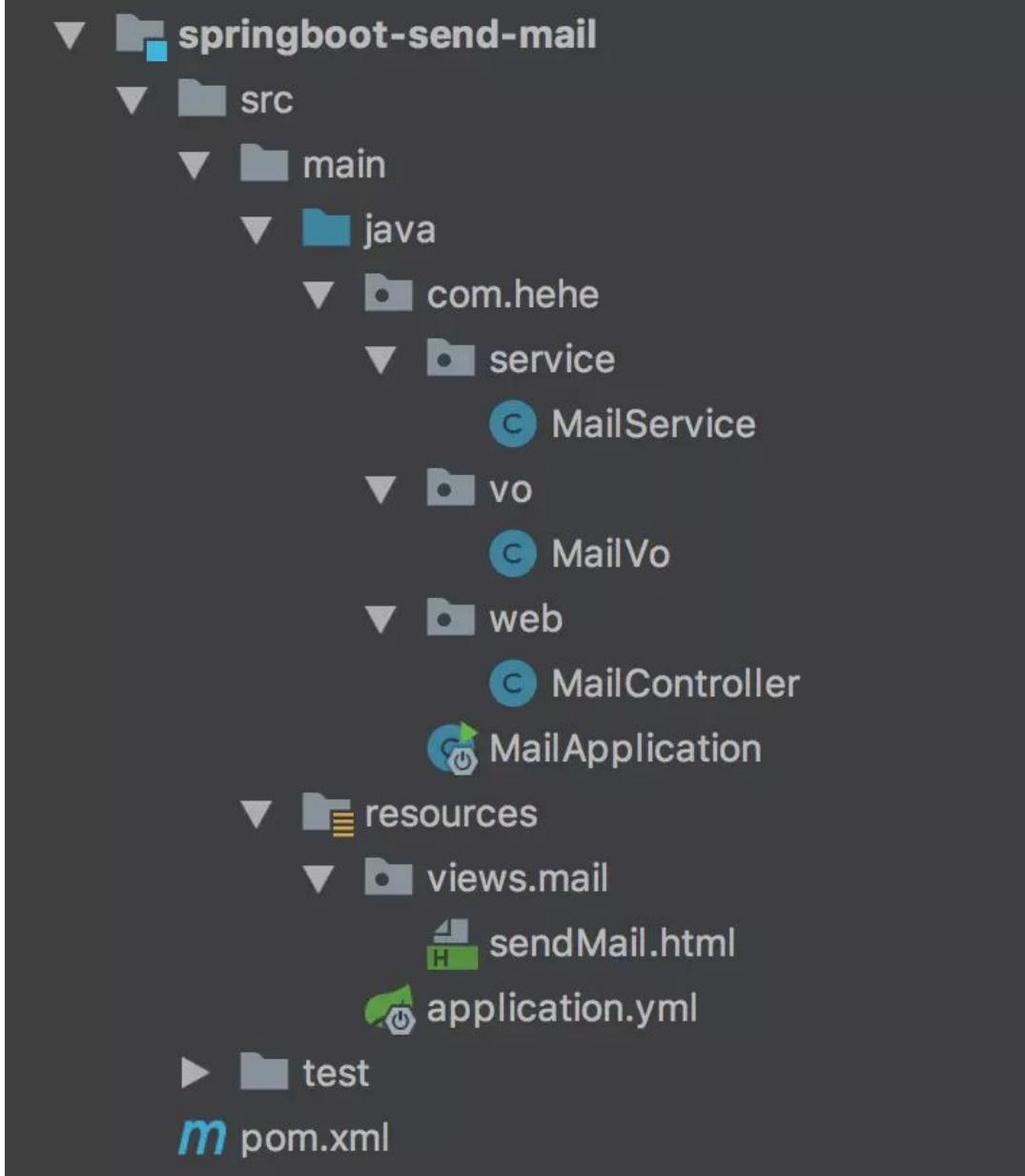
一、开启邮件服务

登陆网易邮箱163，在设置中打开并勾选POP3/SMTP/IMAP服务，然后会得到一个授权码，这个邮箱和授权码将用作登陆认证。



二、配置邮件服务

首先咱们通过 Spring Initializr 创建工程springboot-send-mail，如图所示：



然后在pom.xml 引入web、 thymeleaf 和spring-boot-starter-mail等相关依赖。例如：

```

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-mail</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>jquery</artifactId>
        <version>3.3.1</version>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>bootstrap</artifactId>
        <version>3.3.7</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

```

根据前面提到的配置项(MailProperties)填写相关配置信息，其中spring.mail.username 表示连接邮件服务器时认证的登陆账号，可以是普通的手机号或者登陆账号，并非一定是邮箱，为了解决这个问题，推荐大家在spring.mail.properties.from填写邮件发信人即真实邮箱。

Tips：关注微信公众号：Java后端，每日提送技术博文。

然后在application.yml添加如下配置：

```

spring:
  mail:
    host: smtp.163.com #SMTP服务器地址
    username: socks #登陆账号
    password: 123456 #登陆密码（或授权码）
    properties:
      from: socks@163.com #邮件发信人（即真实邮箱）
    thymeleaf:
      cache: false
      prefix: classpath:/views/
    servlet:
      multipart:
        max-file-size: 10MB #限制单个文件大小
        max-request-size: 50MB #限制请求总量

```

透过前面的进阶知识，我们知道在发送邮件前，需要先构建 SimpleMailMessage 或 MimeMessage 邮件信息类来填写邮件标题、邮件内容等信息，最后提交给 JavaMailSenderImpl 发送邮件，这样看起来没什么问题，也能实现既定目标，但在实际使用中会出现大量零散和重复的代码，还不便于保存邮件到数据库。

那么优雅的发送邮件应该是如何的呢？应该屏蔽掉这些构建信息和发送邮件的细节，不管是简单还是复杂邮件，都可以通过统一的 API 来发送邮件。例如： mailService.send(mailVo) 。

例如通过邮件信息类(MailVo) 来保存发送邮件时的邮件主题、邮件内容等信息：

```
package com.hehe.vo;

public class MailVo {
    private String id;//邮件id
    private String from;//邮件发送人
    private String to;//邮件接收人 (多个邮箱则用逗号","隔开)
    private String subject;//邮件主题
    private String text;//邮件内容
    private Date sentDate;//发送时间
    private String cc;//抄送 (多个邮箱则用逗号","隔开)
    private String bcc;//密送 (多个邮箱则用逗号","隔开)
    private String status;//状态
    private String error;//报错信息
    @JsonIgnore
    private MultipartFile[] multipartFiles;//邮件附件
    //省略GET&SET方法
}
```

三、发送邮件和附件

除了发送邮件之外，还包括检测邮件和保存邮件等操作，例如：

- 检测邮件 checkMail(); 首先校验邮件收信人、邮件主题和邮件内容这些必填项，若为空则拒绝发送。
- 发送邮件 sendMimeMail(); 其次通过 MimeMessageHelper 来解析 MailVo 并构建 MimeMessage 传输邮件。
- 保存邮件 sendMimeMail(); 最后将邮件保存到数据库，便于统计和追查邮件问题。

本案例邮件业务类 MailService 的具体源码如下：

```
package com.hehe.service;

/**
 * 邮件业务类 MailService
 */
@Service
public class MailService {

    private Logger logger = LoggerFactory.getLogger(getClass());//提供日志类

    @Autowired
    private JavaMailSenderImpl mailSender;//注入邮件工具类

    /**
     * 发送邮件
     */
    public MailVo sendMail(MailVo mailVo) {
        try {
            checkMail(mailVo); //1.检测邮件
            sendMimeMail(mailVo); //2.发送邮件
        } catch (Exception e) {
            logger.error("发送邮件失败：" + e.getMessage());
        }
    }
}
```

```
return saveMail(mailVo); //3.保存邮件
} catch (Exception e) {
    logger.error("发送邮件失败:", e); //打印错误信息
    mailVo.setStatus("fail");
    mailVo.setError(e.getMessage());
    return mailVo;
}

}

//检测邮件信息类
private void checkMail(MailVo mailVo) {
    if (StringUtils.isEmpty(mailVo.getTo())) {
        throw new RuntimeException("邮件收信人不能为空");
    }
    if (StringUtils.isEmpty(mailVo.getSubject())) {
        throw new RuntimeException("邮件主题不能为空");
    }
    if (StringUtils.isEmpty(mailVo.getText())) {
        throw new RuntimeException("邮件内容不能为空");
    }
}

//构建复杂邮件信息类
private void sendMimeMail(MailVo mailVo) {
    try {
        MimeMessageHelper messageHelper = new MimeMessageHelper(mailSender.createMimeMessage(), true); //true表示支持复杂类型
        mailVo.setFrom(getMailSendFrom()); //邮件发信人从配置项读取
        messageHelper.setFrom(mailVo.getFrom()); //邮件发信人
        messageHelper.setTo(mailVo.getTo().split(", ")); //邮件收信人
        messageHelper.setSubject(mailVo.getSubject()); //邮件主题
        messageHelper.setText(mailVo.getText()); //邮件内容
        if (!StringUtils.isEmpty(mailVo.getCc())) { //抄送
            messageHelper.setCc(mailVo.getCc().split(", "));
        }
        if (!StringUtils.isEmpty(mailVo.getBcc())) { //密送
            messageHelper.setCc(mailVo.getBcc().split(", "));
        }
        if (mailVo.getMultipartFiles() != null) { //添加邮件附件
            for (MultipartFile multipartFile : mailVo.getMultipartFiles()) {
                messageHelper.addAttachment(multipartFile.getOriginalFilename(), multipartFile);
            }
        }
        if (StringUtils.isEmpty(mailVo.getSentDate())) { //发送时间
            mailVo.setSentDate(new Date());
            messageHelper.setSentDate(mailVo.getSentDate());
        }
        mailSender.send(messageHelper.getMimeMessage()); //正式发送邮件
        mailVo.setStatus("ok");
        logger.info("发送邮件成功: {}->{}", mailVo.getFrom(), mailVo.getTo());
    } catch (Exception e) {
        throw new RuntimeException(e); //发送失败
    }
}

//保存邮件
private MailVo saveMail(MailVo mailVo) {
    //将邮件保存到数据库..
    return mailVo;
}

//获取邮件发信人
public String getMailSendFrom() {
    return mailSender.getJavaMailProperties().getProperty("from");
}
```

```
    return mailService.getMailFrom().getProperties().getProperty("from");
}
```

搞定了发送邮件最核心的业务逻辑，接下来咱们写一个简单页面用来发送邮件。

首先写好跟页面交互的控制器 MailController，具体源码如下：

```
@RestController
public class MailController {
    @Autowired
    private MailService mailService;

    /**
     * 发送邮件的主界面
     */
    @GetMapping("/")
    public ModelAndView index() {
        ModelAndView mv = new ModelAndView("mail/sendMail");//打开发送邮件的页面
        mv.addObject("from", mailService.getMailSendFrom());//邮件发信人
        return mv;
    }

    /**
     * 发送邮件
     */
    @PostMapping("/mail/send")
    public MailVo sendMail(MailVo mailVo, MultipartFile[] files) {
        mailVo.setMultipartFiles(files);
        return mailService.sendMail(mailVo);//发送邮件和附件
    }
}
```

然后在/resources/views/mail目录新建sendMail.html，具体源码如下：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<head>
    <meta charset="UTF-8"/>
    <title>发送邮件</title>
    <link th:href="@{/webjars/bootstrap/css/bootstrap.min.css}" rel="stylesheet" type="text/css"/>
    <script th:src="@{/webjars/jquery/jquery.min.js}"></script>
    <script th:href="@{/webjars/bootstrap/js/bootstrap.min.js}"></script>

</head>

<body>
<div class="col-md-6" style="margin:20px;padding:20px;border: #E0E0E0 1px solid;">
    <marquee behavior="alternate" onfinish="alert(12)" id="mq"
        onMouseOut="this.start();$('#egg').text('嗯 真听话！ ');"
        onMouseOver="this.stop();$('#egg').text('有本事放开我呀！ ');"
        <h5 id="egg">祝大家新年快乐！ </h5>
    </marquee>

    <form class="form-horizontal" id="mailForm">
        <div class="form-group">
            <label class="col-md-2 control-label">邮件发信人:</label>
            <div class="col-md-6">
                <input class="form-control" id="from" name="from" th:value="${from}" readonly="readonly">
            </div>
        </div>
        <div class="form-group">
            <label class="col-md-2 control-label">邮件收件人:</label>
            <div class="col-md-6">
                <input class="form-control" id="to" name="to" th:value="${to}" type="text">
            </div>
        </div>
        <div class="form-group">
            <label class="col-md-2 control-label">邮件主题:</label>
            <div class="col-md-6">
                <input class="form-control" id="subject" name="subject" type="text" value="新年快乐！" />
            </div>
        </div>
        <div class="form-group">
            <label class="col-md-2 control-label">邮件内容:</label>
            <div class="col-md-6">
                <input class="form-control" id="content" name="content" type="text" value="祝大家新年快乐！" />
            </div>
        </div>
        <div class="form-group">
            <label class="col-md-2 control-label">附件:</label>
            <div class="col-md-6">
                <input type="file" multiple="multiple" name="files" />
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-6 col-md-offset-2">
                <button type="button" class="btn btn-primary" onclick="submitForm();">发送邮件
            </div>
        </div>
    </form>
</div>

```

```

<label class="col-md-2 control-label">邮件收信人:</label>
<div class="col-md-6">
    <input class="form-control" id="to" name="to" title="多个邮箱使用,隔开">
</div>
</div>
<div class="form-group">
    <label class="col-md-2 control-label">邮件主题:</label>
    <div class="col-md-6">
        <input class="form-control" id="subject" name="subject">
    </div>
</div>
<div class="form-group">
    <label class="col-md-2 control-label">邮件内容:</label>
    <div class="col-md-6">
        <textarea class="form-control" id="text" name="text" rows="5"></textarea>
    </div>
</div>
<div class="form-group">
    <label class="col-md-2 control-label">邮件附件:</label>
    <div class="col-md-6">
        <input class="form-control" id="files" name="files" type="file" multiple="multiple">
    </div>
</div>
<div class="form-group">
    <label class="col-md-2 control-label">邮件操作:</label>
    <div class="col-md-3">
        <a class="form-control btn btn-primary" onclick="sendMail()">发送邮件</a>
    </div>
    <div class="col-md-3">
        <a class="form-control btn btn-default" onclick="clearForm()">清空</a>
    </div>
</div>
</div>
</form>

```

```

<script th:inline="javascript">
    var appCtx = [[${#request.getContextPath()}]];

    function sendMail() {
        var formData = new FormData($('#mailForm')[0]);
        $.ajax({
            url: appCtx + '/mail/send',
            type: "POST",
            data: formData,
            contentType: false,
            processData: false,
            success: function (result) {
                alert(result.status === 'ok' ? "发送成功！" : "你被Doge嘲讽了：" + result.error);
            },
            error: function () {
                alert("发送失败！");
            }
        });
    }

    function clearForm() {
        $('#mailForm')[0].reset();
    }

    setInterval(function () {
        var total = $('#mq').width();
        var width = $('#doge').width();
        var left = $('#doge').offset().left;
        if (left <= width / 2 + 20) {

```

```
$('#doge').css('transform', 'rotateY(180deg)')  
}  
if(left >= total - width / 2 - 40 {  
    $('#doge').css('transform', 'rotateY(-360deg)')  
}  
});  
</script>  
</div>  
</body>  
</html>
```

四、测试发送邮件

如果是初学者，建议大家先下载源码，修改配置后运行工程，成功后再自己重新写一遍代码，这样有助于加深记忆。

启动工程并访问：<http://localhost:8080> 然后可以看到发送邮件的主界面如下：

嗯 真听话！



邮件发信人:

liver_r@mail@163.com

邮件收信人:

████████@qq.com

邮件主题:

Happy New Yeah 兄dei !

邮件内容:

2019 新年快乐!

邮件附件:

Choose Files doge.gif

邮件操作:

发送邮件

清空

然后填写你的小号邮箱，点击发送邮件，若成功则可以登陆小号邮箱查看邮件和刚才上传的附件。

« 返回 回复 回复全部 转发 删除 彻底删除 举报 拒收 标记为... 移动到... »

Happy New Yeah 兄dei ! ☆

发件人: [一只袜子 <251111111@qq.com>](#)

时间: 2019年1月6日(星期天)凌晨1:50

收件人: 一只袜子 <251111111@qq.com>

附件: 1 个 (doge.gif)

这不是腾讯公司的官方邮件。请勿轻信密保、汇款、中奖信息，勿轻易拨打陌生电话。 举报垃圾邮件

2019 新年快乐！

附件(1 个)

普通附件

doge.gif (132.17K)  邮件附件发送成功！

预览 下载 收藏 转存

至此发送邮件代码全部完成，欢迎大家下载并关注Github 源码。

五、常见失败编码

如果企业定制了邮件服务器，自然会记录邮件日志，根据错误编码存储日志有利于日常维护。

例如这些由网易邮箱提供的错误编码标识：

421

- 421 HL:REP 该IP发送行为异常，存在接收者大量不存在情况，被临时禁止连接。请检查是否有用户发送病毒或者垃圾邮件，并核对发送列表有效性；
- 421 HL:ICC 该IP同时并发连接数过大，超过了网易的限制，被临时禁止连接。请检查是否有用户发送病毒或者垃圾邮件，并降低IP并发连接数量；
- 421 HL:IFC 该IP短期内发送了大量信件，超过了网易的限制，被临时禁止连接。请检查是否有用户发送病毒或者垃圾邮件，并降低发送频率；
- 421 HL:MEP 该IP发送行为异常，存在大量伪造发送域域名行为，被临时禁止连接。请检查是否有用户发送病毒或者垃圾邮件，并使用真实有效的域名发送；

450

- 450 MI:CEL 发送方出现过多的错误指令。请检查发信程序；
- 450 MI:DMC 当前连接发送的邮件数量超出限制。请减少每次连接中投递的邮件数量；
- 450 MI:CCL 发送方发送超出正常的指令数量。请检查发信程序；
- 450 RP:DRC 当前连接发送的收件人数量超出限制。请控制每次连接投递的邮件数量；
- 450 RP:CCL 发送方发送超出正常的指令数量。请检查发信程序；
- 450 DT:RBL 发信IP位于一个或多个RBL里。请参考<http://www.rbls.org/>关于RBL的相关信息；
- 450 WM:BLI 该IP不在网易允许的发送地址列表里；
- 450 WM:BLU 此用户不在网易允许的发信用户列表里；

451

- 451 DT:SPM ,please try again 邮件正文带有垃圾邮件特征或发送环境缺乏规范性,被临时拒收。请保持邮件队列,两分钟
后重投邮件。需调整邮件内容或优化发送环境;
- 451 Requested mail action not taken: too much fail authentication 登录失败次数过多,被临时禁止登录。请检查密码
与帐号验证设置;
- 451 RP:CEL 发送方出现过多的错误指令。请检查发信程序;
- 451 MI:DMC 当前连接发送的邮件数量超出限制。请控制每次连接中投递的邮件数量;
- 451 MI:SFQ 发信人在15分钟内的发信数量超过限制,请控制发信频率;
- 451 RP:QRC 发信方短期内累计的收件人数量超过限制,该发件人被临时禁止发信。请降低该用户发信频率;
- 451 Requested action aborted: local error in processing 系统暂时出现故障,请稍后再次尝试发送;

500

- 500 Error: bad syntaxU 发送的smtp命令语法有误;
- 550 MI:NHD HELO命令不允许为空;
- 550 MI:IMF 发信人电子邮件地址不合规范。请参考<http://www.rfc-editor.org/>关于电子邮件规范的定义;
- 550 MI:SPF 发信IP未被发送域的SPF许可。请参考<http://www.openspf.org/>关于SPF规范的定义;
- 550 MI:DMA 该邮件未被发信域的DMARC许可。请参考<http://dmarc.org/>关于DMARC规范的定义;
- 550 MI:STC 发件人当天的连接数量超出了限定数量,当天不再接受该发件人的邮件。请控制连接次数;
- 550 RP:FRL 网易邮箱不开放匿名转发 (Open relay) ;
- 550 RP:RCL 群发收件人数量超过了限额,请减少每封邮件的收件人数量;
- 550 RP:TRC 发件人当天内累计的收件人数量超过限制,当天不再接受该发件人的邮件。请降低该用户发信频率;
- 550 DT:SPM 邮件正文带有很多垃圾邮件特征或发送环境缺乏规范性。需调整邮件内容或优化发送环境;
- 550 Invalid User 请求的用户不存在;
- 550 User in blacklist 该用户不被允许给网易用户发信;
- 550 User suspended 请求的用户处于禁用或者冻结状态;
- 550 Requested mail action not taken: too much recipient 群发数量超过了限额;

552

- 552 Illegal Attachment 不允许发送该类型的附件,包括以.uu .pif .scr .mim .hqx .bhx .cmd .vbs .bat .com .vbe .vb .js
.wsh等结尾的附件;
- 552 Requested mail action aborted: exceeded mailsize limit 发送的信件大小超过了网易邮箱允许接收的最大限制;

553

- 553 Requested action not taken: NULL sender is not allowed 不允许发件人为空,请使用真实发件人发送;
- 553 Requested action not taken: Local user only SMTP类型的机器只允许发信人是本站用户;
- 553 Requested action not taken: no smtp MX only MX类型的机器不允许发信人是本站用户;
- 553 authentication is required SMTP需要身份验证,请检查客户端设置;

- 554 DT:SPM 发送的邮件内容包含了未被许可的信息，或被系统识别为垃圾邮件。请检查是否有用户发送病毒或者垃圾邮件；
- 554 DT:SUM 信封发件人和信头发件人不匹配；
- 554 IP is rejected, smtp auth error limit exceed 该IP验证失败次数过多，被临时禁止连接。请检查验证信息设置；
- 554 HL:IHU 发信IP因发送垃圾邮件或存在异常的连接行为，被暂时挂起。请检测发信IP在历史上的发信情况和发信程序是否存在异常；
- 554 HL:IPB 该IP不在网易允许的发送地址列表里；
- 554 MI:STC 发件人当天内累计邮件数量超过限制，当天不再接受该发件人的投信。请降低发信频率；
- 554 MI:SPB 此用户不在网易允许的发信用户列表里；
- 554 IP in blacklist 该IP不在网易允许的发送地址列表里。

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. 前后端分离开发, RESTful 接口如何设计
2. 面试官:讲一下 Mybatis 初始化原理
3. 我们再来聊一聊 Java 的单例吧

4. 我采访了一位 Pornhub 工程师

5. 团队开发中 Git 最佳实践



喜欢文章, 点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

41道 Spring Boot 面试题，帮你整理好了！

Java后端 5天前

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

今天跟大家分享下SpringBoot 常见面试题的知识。

1 什么是springboot ?

用来简化spring应用的初始搭建以及开发过程 使用特定的方式来进行配置 (properties或yml文件)

创建独立的spring引用程序 main方法运行

嵌入的Tomcat 无需部署war文件

简化maven配置

自动配置spring添加对应功能starter自动化配置

答:spring boot来简化spring应用开发, 约定大于配置, 去繁从简, just run就能创建一个独立的, 产品级别的应用

2 Springboot 有哪些优点?

-快速创建独立运行的spring项目与主流框架集成

-使用嵌入式的servlet容器, 应用无需打包成war包

-starters自动依赖与版本控制

-大量的自动配置, 简化开发, 也可修改默认值

-准生产环境的运行应用监控

-与云计算的天然集成

3 如何重新加载Spring Boot上的更改, 而无需重新启动服务器?

这可以使用DEV工具来实现。通过这种依赖关系, 您可以节省任何更改, 嵌入式tomcat将重新启动。

Spring Boot有一个开发工具(DevTools)模块, 它有助于提高开发人员的生产力。Java开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。

开发人员可以重新加载Spring Boot上的更改, 而无需重新启动服务器。这将消除每次手动部署更改的需要。

Spring Boot在发布它的第一个版本时没有这个功能。

这是开发人员最需要的功能。DevTools模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供H2数据库控制台以更好地测试应用程序。

`org.springframework.boot`

`spring-boot-devtools`

`true`

4 Spring Boot、Spring MVC 和 Spring 有什么区别？

1、Spring

Spring最重要的特征是依赖注入。所有 Spring Modules 不是依赖注入就是 IOC 控制反转。

当我们恰当的使用 DI 或者是 IOC 的时候，我们可以开发松耦合应用。松耦合应用的单元测试可以很容易的进行。

2、Spring MVC

Spring MVC 提供了一种分离式的方法来开发 Web 应用。通过运用像 DispatcherServlet, ModelAndView 和 ViewResolver 等一些简单的概念，开发 Web 应用将会变得非常简单。

3、SpringBoot

Spring 和 SpringMVC 的问题在于需要配置大量的参数。

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">
        <value>/WEB-INF/views/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>

<mvc:resources mapping="/webjars/**" location="/webjars/" />
```

https://blog.csdn.net/Dome_

Spring Boot 通过一个自动配置和启动的项来解决这个问题。为了更快的构建产品就绪应用程序，Spring Boot 提供了一些非功能性特征。

5 什么是自动配置？

Spring 和 SpringMVC 的问题在于需要配置大量的参数。

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">
        <value>/WEB-INF/views/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>

<mvc:resources mapping="/webjars/**" location="/webjars/" />
```

https://blog.csdn.net/Dome_

我们能否带来更多的智能？当一个 MVC JAR 添加到应用程序中的时候，我们能否自动配置一些 beans？

Spring 查看(CLASSPATH 上可用的框架)已存在的应用程序的配置。在此基础上，Spring Boot 提供了配置应用程序和框架所需要的基本配置。这就是自动配置。

6 什么是 Spring Boot Starter ?

启动器是一套方便的依赖描述符，它可以放在自己的程序中。你可以一站式的获取你所需要的 Spring 和相关技术，而不需要依赖描述符的通过示例代码搜索和复制黏贴的负载。

例如，如果你想使用 Sping 和 JPA 访问数据库，只需要你的项目包含 spring-boot-starter-data-jpa 依赖项，

7 能否举一个例子来解释更多 Staters 的内容？

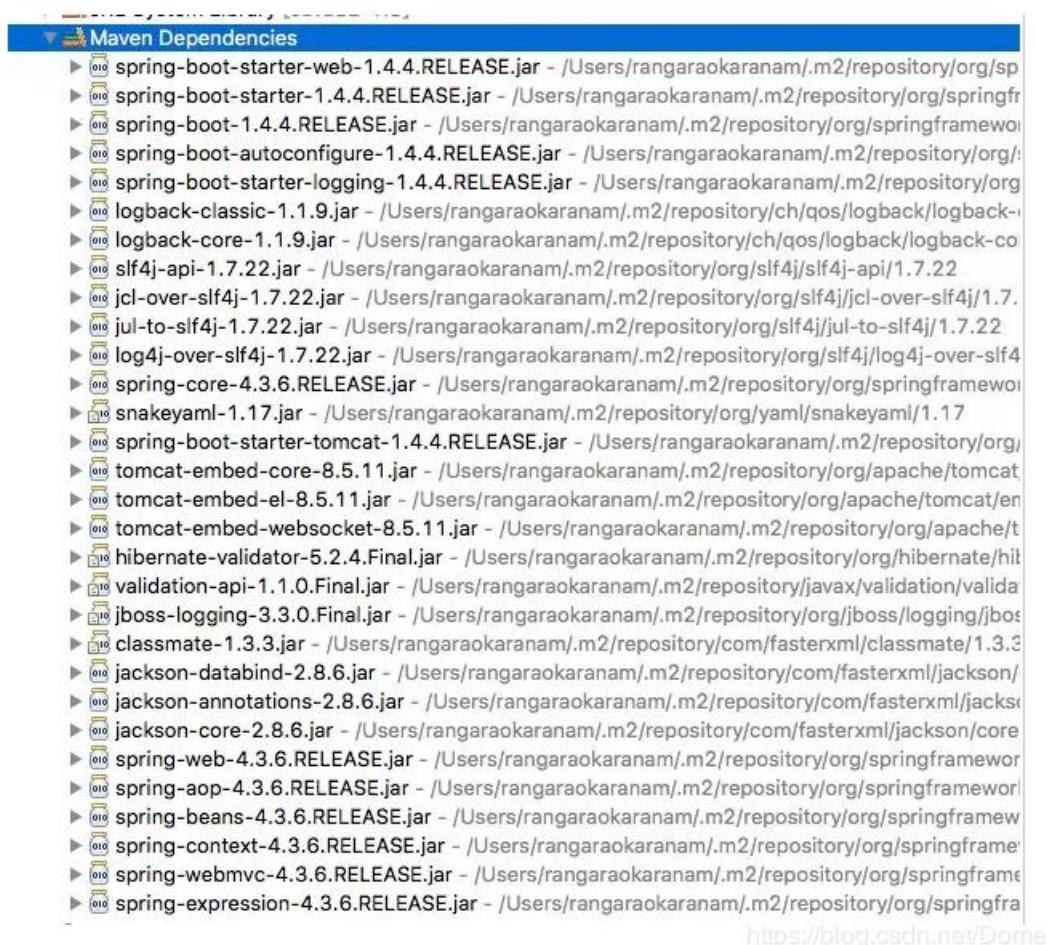
让我们来思考一个 Stater 的例子 -Spring Boot Starter Web。

如果你想开发一个 web 应用程序或者是公开 REST 服务的应用程序。Spring Boot Start Web 是首选。让我们使用 Spring Initializr 创建一个 Spring Boot Start Web 的快速项目。

Spring Boot Start Web 的依赖项

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

下面的截图是添加进我们应用程序的不同的依赖项



https://blog.csdn.net/Dome_

依赖项可以被分为：

- Spring - core, beans, context, aop
- Web MVC - (Spring MVC)
- Jackson - for JSON Binding
- Validation - Hibernate,Validation API
- Embedded Servlet Container - Tomcat
- Logging - logback,slf4j

任何经典的 Web 应用程序都会使用所有这些依赖项。Spring Boot Starter Web 预先打包了这些依赖项。

作为一个开发者，我不需要再担心这些依赖项和它们的兼容版本。

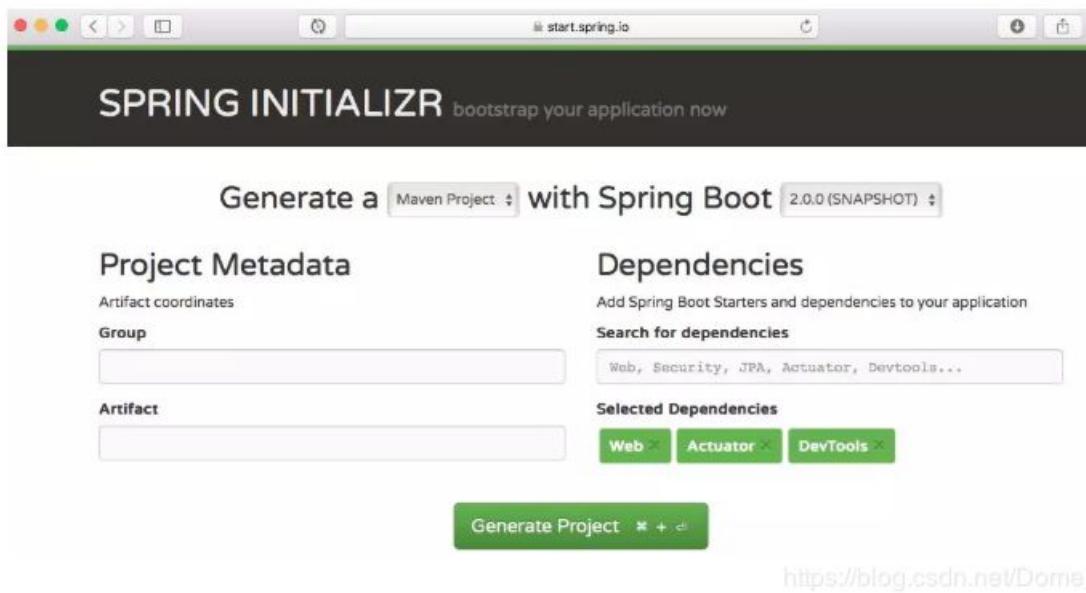
8 Spring Boot 还提供了其它的哪些 Starter Project Options?

Spring Boot 也提供了其它的启动器项目包括，包括用于开发特定类型应用程序的典型依赖项。

- spring-boot-starter-web-services - SOAP Web Services;
- spring-boot-starter-web - Web 和 RESTful 应用程序；
- spring-boot-starter-test - 单元测试和集成测试；
- spring-boot-starter-jdbc - 传统的 JDBC；
- spring-boot-starter-hateoas - 为服务添加 HATEOAS 功能；
- spring-boot-starter-security - 使用 SpringSecurity 进行身份验证和授权；
- spring-boot-starter-data-jpa - 带有 Hibernate 的 Spring Data JPA；
- spring-boot-starter-data-rest - 使用 Spring Data REST 公布简单的 REST 服务；

9 创建一个 Spring Boot Project 的最简单的方法是什么？

Spring Initializr是启动 Spring Boot Projects 的一个很好的工具。



就像上图中所展示的一样，我们需要做一下几步：

1、登录 Spring Initializr，按照以下方式进行选择：

2、选择 com.in28minutes.springboot 为组

3、选择 student-services 为组件

4、选择下面的依赖项

Web

Actuator

DevTools

5、点击生 GenerateProject

10 Spring Initializr 是创建 Spring Boot Projects 的唯一方法吗？

不是的。

Spring Initializr 让创建 Spring Boot 项目变得很容易，但是，你也可以通过设置一个 maven 项目并添加正确的依赖项来开始一个项目。

在我们的 Spring 课程中，我们使用两种方法来创建项目。

第一种方法是 start.spring.io。

另外一种方法是在项目的标题为“Basic Web Application”处进行手动设置。

手动设置一个 maven 项目

这里有以下几个重要的步骤：

1、在 Eclipse 中，使用文件 - 新建 Maven 项目来创建一个新项目

2、添加依赖项。

3、添加 maven 插件。

4、添加 Spring Boot 应用程序类。

到这里，准备工作已经做好！

11 为什么我们需要 spring-boot-maven-plugin？

spring-boot-maven-plugin 提供了一些像 jar 一样打包或者运行应用程序的命令。

1、spring-boot:run 运行你的 SpringBooty 应用程序。

2、spring-boot:repackage 重新打包你的 jar 包或者是 war 包使其可执行

3、spring-boot:start 和 spring-boot:stop 管理 Spring Boot 应用程序的生命周期（也可以说是为了集成测试）。

4、spring-boot:build-info 生成执行器可以使用的构造信息。

12 如何使用 SpringBoot 自动重装我的应用程序？

使用 Spring Boot 开发工具。

把 Spring Boot 开发工具添加进入你的项目是简单的。

把下面的依赖项添加至你的 Spring Boot Project pom.xml 中

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
```

重启应用程序，然后就可以了。

同样的，如果你想自动装载页面，有可以看看 FiveReload

```
1 http://www.logicbig.com/tutorials/spring-framework/spring-boot/boot-live-reload/.
```

在我测试的时候，发现了 LiveReload 漏洞，如果你测试时也发现了，请一定要告诉我们。

13 Spring Boot中的监视器是什么？

Spring boot actuator是spring启动框架中的重要功能之一。Spring boot监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。

有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为HTTP URL访问的REST端点来检查状态。

14 什么是YAML？

YAML是一种人类可读的数据序列化语言。它通常用于配置文件。

与属性文件相比，如果我们想要在配置文件中添加复杂的属性，YAML文件就更加结构化，而且更少混淆。可以看出 YAML具有分层配置数据。

15 springboot自动配置的原理

在spring程序main方法中 添加@SpringBootApplication或者@EnableAutoConfiguration

会自动去maven中读取每个starter中的spring.factories文件 该文件里配置了所有需要被创建spring容器中的bean

16 springboot读取配置文件的方式

springboot默认读取配置文件为application.properties或者是application.yml

17 springboot集成mybatis的过程

添加mybatis的starter maven依赖

```
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>1.3.2</version>
</dependency>
```

在mybatis的接口中 添加@Mapper注解

在application.yml配置数据源信息

18 什么是嵌入式服务器?我们为什么要使用嵌入式服务器呢?

思考一下在你的虚拟机上部署应用程序需要些什么。

第一步：安装 Java

第二步：安装 Web 或者是应用程序的服务器(Tomcat/WebSphere/Weblogic 等等)

第三步：部署应用程序 war 包

如果我们想简化这些步骤，应该如何做呢？

让我们来思考如何使服务器成为应用程序的一部分？

你只需要一个安装了 Java 的虚拟机，就可以直接在上面部署应用程序了，

这个想法是嵌入式服务器的起源。

当我们创建一个可以部署的应用程序的时候，我们将会把服务器（例如，tomcat）嵌入到可部署的服务器中。

例如，对于一个 Spring Boot 应用程序来说，你可以生成一个包含 Embedded Tomcat 的应用程序 jar。你就可想而知运行正常 Java 应用程序一样来运行 web 应用程序了。

嵌入式服务器就是我们的可执行单元包含服务器的二进制文件（例如，tomcat.jar）。

19 如何在 Spring Boot 中添加通用的 JS 代码？

在源文件夹下，创建一个名为 static 的文件夹。然后，你可以把你的静态的内容放在这里面。

例如，myapp.js 的路径是 resources\static\js\myapp.js

你可以参考它在 jsp 中的使用方法：

```
<script src="/js/myapp.js"></script>
```

错误：HAL browser gives me unauthorized error - Full authentication is required to access this resource.

该如何来修复这个错误呢？

```
{
  "timestamp": 1488656019562,
  "status": 401,
  "error": "Unauthorized",
  "message": "Full authentication is required to access this resource.",
  "path": "/beans"
}
```

两种方法：

方法 1：关闭安全验证

application.properties

```
management.security.enabled:FALSE
```

方法二：在日志中搜索密码并传递至请求标头中

20 什么是 Spring Data？

来自：//projects.spring.io/spring-data/

Spring Data 的使命是在保证底层数据存储特殊性的前提下，为数据访问提供一个熟悉的、一致性的，基于

Spring 的编程模型。这使得使用数据访问技术，关系数据库和非关系数据库，map-reduce 框架以及基于云的数据服务变得很容易。

为了让它更简单一些，Spring Data 提供了不受底层数据源限制的 Abstractions 接口。

下面来举一个例子：

```
interface TodoRepository extends CrudRepository<Todo, Long> {
```

你可以定义一个简单的库，用来插入，更新，删除和检索代办事项，而不需要编写大量的代码。

21 什么是 Spring Data REST?

Spring Data TEST 可以用来发布关于 Spring 数据库的 HATEOAS RESTful 资源。

下面是一个使用 JPA 的例子：

```
@RepositoryRestResource(collectionResourceRel = "todos", path = "todos")
public interface TodoRepository extends PagingAndSortingRepository<Todo, Long> {
```

不需要写太多代码，我们可以发布关于 Spring 数据库的 RESTful API。

下面展示的是一些关于 TEST 服务器的例子

```
1 POST:
2 URL: http://localhost:8080/todos
3 Use Header: Content-Type: application/json
4 Request Content
```

代码如下：

```
{
  "user": "Jill",
  "desc": "Learn Hibernate",
  "done": false
}
```

响应内容：

```
{  
    "user": "Jill",  
    "desc": "Learn Hibernate",  
    "done": false,  
    "_links": {  
        "self": {  
            "href": "http://localhost:8080/todos/1"  
        },  
        "todo": {  
            "href": "http://localhost:8080/todos/1"  
        }  
    }  
}
```

https://blog.csdn.net/Dome_

响应包含新创建资源的 href。

22 path="users", collectionResourceRel="users" 如何与 Spring Data Rest 一起使用?

```
@RepositoryRestResource(collectionResourceRel = "users", path = "users")  
  
public interface UserRestRepository extends PagingAndSortingRepository<User, Long>{  
}
```

path- 这个资源要导出的路径段。

collectionResourceRel- 生成指向集合资源的链接时使用的 rel 值。在生成 HATEOAS 链接时使用。

23 当 Spring Boot 应用程序作为 Java 应用程序运行时,后台会发生什么?

如果你使用 Eclipse IDE, Eclipse maven 插件确保依赖项或者类文件的改变一经添加, 就会被编译并在目标文件中准备好!在这之后, 就和其它的 Java 应用程序一样了。

当你启动 java 应用程序的时候, spring boot 自动配置文件就会魔法般的启用了。

当 Spring Boot 应用程序检测到你正在开发一个 web 应用程序的时候, 它就会启动 tomcat。

24 我们能否在 spring-boot-starter-web 中用 jetty 代替 tomcat?

在 spring-boot-starter-web 移除现有的依赖项, 并把下面这些添加进去。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

https://blog.csdn.net/Dome_

25 如何使用 Spring Boot 生成一个 WAR 文件？

推荐阅读：

<https://spring.io/guides/gs/convert-jar-to-war/>

下面有 spring 说明文档直接的链接地址：

1 <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-maven-packaging>

26 如何使用 Spring Boot 部署到不同的服务器？

你需要做下面两个步骤：

在一个项目中生成一个 war 文件。

将它部署到你最喜欢的服务器 (websphere 或者 Weblogic 或者 Tomcat and so on)。

第一步：这本入门指南应该有所帮助：

<https://spring.io/guides/gs/convert-jar-to-war/>

第二步：取决于你的服务器。

27 RequestMapping 和 GetMapping 的不同之处在哪里？

RequestMapping 具有类属性的，可以进行 GET,POST,PUT 或者其它的注释中具有的请求方法。GetMapping 是 GET 请求方法中的一个特例。它只是 ResquestMapping 的一个延伸，目的是为了提高清晰度。

28 为什么我们不建议在实际的应用程序中使用 Spring Data Rest？

我们认为 Spring Data Rest 很适合快速原型制造！在大型应用程序中使用需要谨慎。

通过 Spring Data REST 你可以把你的数据实体作为 RESTful 服务直接发布。

当你设计 RESTful 服务器的时候，最佳实践表明，你的接口应该考虑到两件重要的事情：

你的模型范围。

你的客户。

通过 With Spring Data REST, 你不需要再考虑这两个方面, 只需要作为 TEST 服务发布实体。

这就是为什么我们建议使用 Spring Data Rest 在快速原型构造上面, 或者作为项目的初始解决方法。对于完整演变项目来说, 这并不是一个好的注意。

29 在 Spring Initializer 中, 如何改变一个项目的包名字?

好消息是你可以定制它。点击链接“转到完整版本”。你可以配置你想要修改的包名称!

30 JPA 和 Hibernate 有哪些区别?

简而言之

JPA 是一个规范或者接口

Hibernate 是 JPA 的一个实现

当我们使用 JPA 的时候, 我们使用 javax.persistence 包中的注释和接口时, 不需要使用 hibernate 的导入包。

我们建议使用 JPA 注释, 因为哦我们没有将其绑定到 Hibernate 作为实现。后来(我知道 - 小于百分之一的几率), 我们可以使用另一种 JPA 实现。

31 使用 Spring Boot 启动连接到内存数据库 H2 的 JPA 应用程序需要哪些依赖项?

在 Spring Boot 项目中, 当你确保下面的依赖项都在类路里面的时候, 你可以加载 H2 控制台。

web 启动器

h2

jpa 数据启动器

其它的依赖项在下面:

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

https://blog.csdn.net/Dome_

需要注意的一些地方:

一个内部数据内存只在应用程序执行期间存在。这是学习框架的有效方式。

这不是你希望的真是世界应用程序的方式。

在问题“如何连接一个外部数据库?”中，我们解释了如何连接一个你所选择的数据库。

32 如何不通过任何配置来选择 Hibernate 作为 JPA 的默认实现？

因为 Spring Boot 是自动配置的。

下面是我们添加的依赖项：

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

spring-boot-starter-data-jpa 对于 Hibernate 和 JPA 有过渡依赖性。

当 Spring Boot 在类路径中检测到 Hibernate 中，将会自动配置它为默认的 JPA 实现。

33 我们如何连接一个像 MySQL 或者 Oracle 一样的外部数据库？

让我们以 MySQL 为例来思考这个问题：

第一步 - 把 mysql 连接器的依赖项添加至 pom.xml

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

第二步 - 从 pom.xml 中移除 H2 的依赖项

或者至少把它作为测试的范围。

```
<!--
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>test</scope>
    </dependency>
-->
```

https://blog.csdn.net/Dome_

第三步 - 安装你的 MySQL 数据库

更多的来看看这里 -<https://github.com/in28minutes/jpa-with-hibernate#installing-and-setting-up-mysql>

第四步 - 配置你的 MySQL 数据库连接

配置 application.properties

```
1 spring.jpa.hibernate.ddl-auto=none
2 spring.datasource.url=jdbc:mysql://localhost:3306/todo_example
```

```
- spring.jpa.hibernate.ddl-auto: none
2 spring.datasource.username=todouser
spring.datasource.password=YOUR_PASSWORD
```

第五步 - 重新启动, 你就准备好了!

就是这么简单!

34 你能否举一个以 `ReadOnly` 为事务管理的例子?

当你从数据库读取内容的时候, 你想把事物中的用户描述或者是其它描述设置为只读模式, 以便于 Heberate 不需要再次检查实体的变化。这是非常高效的。

35 Spring Boot 的核心注解是哪个? 它主要由哪几个注解组成的?

启动类上面的注解是`@SpringBootApplication`, 它也是 Spring Boot 的核心注解, 主要组合包含了以下 3 个注解:

`@SpringBootConfiguration`: 组合了 `@Configuration` 注解, 实现配置文件的功能。

`@EnableAutoConfiguration`: 打开自动配置的功能, 也可以关闭某个自动配置的选项, 如关闭数据源自动配置功能:

`@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })`。

`@ComponentScan`: Spring 组件扫描。

36 开启 Spring Boot 特性有哪几种方式?

1) 继承`spring-boot-starter-parent`项目

2) 导入`spring-boot-dependencies`项目依赖

37 Spring Boot 需要独立的容器运行吗?

可以不需要, 内置了 Tomcat/ Jetty 等容器。

38 运行 Spring Boot 有哪几种方式?

1) 打包用命令或者放到容器中运行

2) 用 Maven/ Gradle 插件运行

3) 直接执行 `main` 方法运行

39 你如何理解 Spring Boot 中的 Starters?

Starters 可以理解为启动器, 它包含了一系列可以集成到应用里面的依赖包, 你可以一站式集成 Spring 及其他技术, 而不需要到处找示例代码和依赖包。如你想使用 Spring JPA 访问数据库, 只要加入`spring-boot-starter-data-jpa` 启动器依赖就能使用了。

40 Spring Boot 支持哪些日志框架? 推荐和默认的日志框架是哪个?

Spring Boot 支持 Java Util Logging, Log4j2, Lockback 作为日志框架, 如果你使用 Starters 启动器, Spring Boot 将使用 Logback 作为默认日志框架.

4.1 SpringBoot 实现热部署有哪几种方式?

主要有两种方式:

- 1、Spring Loaded
- 2、Spring-boot-devtools

参考文献: https://blog.csdn.net/Dome_/article/details/90339363

来源:阿凯的帽子反戴

链接:blog.csdn.net/Kevin_Gu6/article/details/88547424

- E N D -

如果看到这里,说明你喜欢这篇文章,请[转发](#)、[点赞](#)。同时标星(置顶)本公众号可以第一时间接受到博文推送。

推荐阅读

1. 女程序员做了个梦。。。
2. Spring 和 Spring Boot 之间到底有啥区别?
3. IDEA 新特性: 提前知道代码怎么走
4. ping 命令还能这么玩?



微信搜一搜

Java后端

声明: pdf仅供学习使用,一切版权归原创公众号所有;建议持续关注原创公众号获取最新文章,学习愉快!

Spring Boot + MyBatis + Druid + PageHelper 实现多数据源并分页

虚无境 Java后端 2019-11-12

点击上方 Java后端, 选择 [设为星标](#)

接收文章, 更加及时

作者 | 虚无境

链接 | cnblogs.com/xuwujing/p/8964927.html

前言

本篇文章主要讲述的是 Spring Boot整合Mybatis、Druid和PageHelper 并实现多数据源和分页。其中Spring Boot整合Mybatis这块, 在之前的的一篇文章中已经讲述了, 这里就不过多说明了。重点是讲述在多数据源下的如何配置使用Druid和PageHelper。

<http://www.cnblogs.com/xuwujing/p/8260935.html>

Druid介绍和使用

在使用Druid之前, 先来简单的了解下Druid。

Druid是一个数据库连接池。Druid可以说是目前最好的数据库连接池! 因其优秀功能、性能和扩展性方面, 深受开发人员的青睐。

Druid已经在阿里巴巴部署了超过600个应用, 经过一年多生产环境大规模部署的严苛考验。Druid是阿里巴巴开发的号称“为监控而生”的数据库连接池!

同时Druid不仅仅是一个数据库连接池, Druid 核心主要包括三部分:

- 基于Filter-Chain模式的插件体系。
- DruidDataSource 高效可管理的数据库连接池。
- SQLParser

Druid的主要功能如下:

- 是一个高效、功能强大、可扩展性好的数据库连接池。
- 可以监控数据库访问性能。
- 数据库密码加密
- 获得SQL执行日志
- 扩展JDBC

介绍方面这块就不再多说, 具体的可以看官方文档。那么开始介绍Druid如何使用。

首先是Maven依赖, 只需要添加druid这一个jar就行了。

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.8</version>
</dependency>
```

Tips: 可以关注微信公众号: Java后端, 获取Maven教程和每日技术博文推送。

配置方面, 主要的只需要在application.properties或application.yml添加如下就可以了。

说明: 因为这里我是用来两个数据源, 所以稍微有些不同而已。Druid 配置的说明在下面中已经说的很详细了, 这里我就不在说明了。

```
## 默认的数据源
master.datasource.url=jdbc:mysql://localhost:3306/springBoot?useUnicode=true&characterEncoding=utf8&allowMultiQueries=true
master.datasource.username=root
master.datasource.password=123456
master.datasource.driverClassName=com.mysql.jdbc.Driver

## 另一个的数据源
cluster.datasource.url=jdbc:mysql://localhost:3306/springBoot_test?useUnicode=true&characterEncoding=utf8
cluster.datasource.username=root
cluster.datasource.password=123456
cluster.datasource.driverClassName=com.mysql.jdbc.Driver

# 连接池的配置信息
# 初始化大小, 最小, 最大
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.initialSize=5
spring.datasource.minIdle=5
spring.datasource.maxActive=20
# 配置获取连接等待超时的时间
spring.datasource.maxWait=60000
# 配置间隔多久才进行一次检测, 检测需要关闭的空闲连接, 单位是毫秒
spring.datasource.timeBetweenEvictionRunsMillis=60000
# 配置一个连接在池中最小生存的时间, 单位是毫秒
spring.datasource.minEvictableIdleTimeMillis=300000
spring.datasource.validationQuery=SELECT 1 FROM DUAL
spring.datasource.testWhileIdle=true
spring.datasource.testOnBorrow=false
spring.datasource.testOnReturn=false
# 打开PSCache, 并且指定每个连接上PSCache的大小
spring.datasource.poolPreparedStatements=true
spring.datasource.maxPoolPreparedStatementPerConnectionSize=20
# 配置监控统计拦截的filters, 去掉后监控界面sql无法统计, 'wall'用于防火墙
spring.datasource.filters=stat,wall,log4j
# 通过connectProperties属性来打开mergeSql功能; 慢SQL记录
spring.datasource.connectionProperties=druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000
```

成功添加了配置文件之后, 我们再来编写Druid相关的类。

首先是MasterDataSourceConfig.java这个类, 这个是默认的数据源配置类。

```
@Configuration
@MapperScan(basePackages = MasterDataSourceConfig.PACKAGE, sqlSessionFactoryRef = "masterSqlSessionFactory")
public class MasterDataSourceConfig {

    static final String PACKAGE = "com.pancm.dao.master";
    static final String MAPPER_LOCATION = "classpath:mapper/master/*.xml";
```

```
@Value("${master.datasource.url}")
private String url;

@Value("${master.datasource.username}")
private String username;

@Value("${master.datasource.password}")
private String password;

@Value("${master.datasource.driverClassName}")
private String driverClassName;

@Value("${spring.datasource.initialSize}")
private int initialSize;

@Value("${spring.datasource.minIdle}")
private int minIdle;

@Value("${spring.datasource.maxActive}")
private int maxActive;

@Value("${spring.datasource.maxWait}")
private int maxWait;

@Value("${spring.datasource.timeBetweenEvictionRunsMillis}")
private int timeBetweenEvictionRunsMillis;

@Value("${spring.datasource.minEvictableIdleTimeMillis}")
private int minEvictableIdleTimeMillis;

@Value("${spring.datasource.validationQuery}")
private String validationQuery;

@Value("${spring.datasource.testWhileIdle}")
private boolean testWhileIdle;

@Value("${spring.datasource.testOnBorrow}")
private boolean testOnBorrow;

@Value("${spring.datasource.testOnReturn}")
private boolean testOnReturn;

@Value("${spring.datasource.poolPreparedStatements}")
private boolean poolPreparedStatements;

@Value("${spring.datasource.maxPoolPreparedStatementPerConnectionSize}")
private int maxPoolPreparedStatementPerConnectionSize;

@Value("${spring.datasource.filters}")
private String filters;

@Value("{spring.datasource.connectionProperties}")
private String connectionProperties;

@Bean(name = "masterDataSource")
@Primary
public DataSource masterDataSource() {
    DruidDataSource dataSource = new DruidDataSource();
    dataSource.setUrl(url);
    dataSource.setUsername(username);
    dataSource.setPassword(password);
    dataSource.setDriverClassName(driverClassName);
    dataSource.setInitialSize(initialSize);
    dataSource.setMinIdle(minIdle);
    dataSource.setMaxActive(maxActive);
    dataSource.setMaxWait(maxWait);
    dataSource.setTimeBetweenEvictionRunsMillis(timeBetweenEvictionRunsMillis);
    dataSource.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);
    dataSource.setValidationQuery(validationQuery);
    dataSource.setTestWhileIdle(testWhileIdle);
    dataSource.setTestOnBorrow(testOnBorrow);
    dataSource.setTestOnReturn(testOnReturn);
    dataSource.setPoolPreparedStatements(poolPreparedStatements);
    dataSource.setMaxPoolPreparedStatementPerConnectionSize(maxPoolPreparedStatementPerConnectionSize);
    dataSource.setFilters(filters);
    return dataSource;
}
```

```

dataSource.setUsername(username);
dataSource.setPassword(password);
dataSource.setDriverClassName(driverClassName);

//具体配置
dataSource.setInitialSize(initialSize);
dataSource.setMinIdle(minIdle);
dataSource.setMaxActive(maxActive);
dataSource.setMaxWait(maxWait);
dataSource.setTimeBetweenEvictionRunsMillis(timeBetweenEvictionRunsMillis);
dataSource.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);
dataSource.setValidationQuery(validationQuery);
dataSource.setTestWhileIdle(testWhileIdle);
dataSource.setTestOnBorrow(testOnBorrow);
dataSource.setTestOnReturn(testOnReturn);
dataSource.setPoolPreparedStatements(poolPreparedStatements);
dataSource.setMaxPoolPreparedStatementPerConnectionSize(maxPoolPreparedStatementPerConnectionSize);
try {
    dataSource.setFilters(filters);
} catch (SQLException e) {
    e.printStackTrace();
}
dataSource.setConnectionProperties(connectionProperties);
return dataSource;
}

@Bean(name = "masterTransactionManager")
@Primary
public DataSourceTransactionManager masterTransactionManager() {
    return new DataSourceTransactionManager(masterDataSource());
}

@Bean(name = "masterSqlSessionFactory")
@Primary
public SqlSessionFactory masterSqlSessionFactory(@Qualifier("masterDataSource") DataSource masterDataSource)
    throws Exception {
    final SqlSessionFactoryBean sessionFactory = new SqlSessionFactoryBean();
    sessionFactory.setDataSource(masterDataSource);
    sessionFactory.setMapperLocations(new PathMatchingResourcePatternResolver()
        .getResources(MasterDataSourceConfig.MAPPER_LOCATION));
    return sessionFactory.getObject();
}
}

```

其中这两个注解说明下：

- **@Primary** : 标志这个 Bean 如果在多个同类 Bean 候选时, 该 Bean 优先被考虑。多数据源配置的时候注意, 必须要有一个主数据源, 用 @Primary 标志该 Bean。
- **@MapperScan**: 扫描 Mapper 接口并容器管理。

需要注意的是sqlSessionFactoryRef 表示定义一个唯一 SqlSessionFactory 实例。

上面的配置完之后, 就可以将Druid作为连接池使用了。但是Druid并不简简单单的是个连接池, 它也可以说是一个监控应用, 它自带了web监控界面, 可以很清晰的看到SQL相关信息。

在SpringBoot中运用Druid的监控作用, 只需要编写StatViewServlet和WebStatFilter类, 实现注册服务和过滤规则。这里我们可以将这两个写在一起, 使用@Configuration和@Bean。

为了方便理解，相关的配置说明也写在代码中了，这里就不再过多赘述了。

代码如下：

```
@Configuration
public class DruidConfiguration {

    @Bean
    public ServletRegistrationBean druidStatViewServlet() {
        //注册服务
        ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean(
            new StatViewServlet(), "/druid/*");
        //白名单(为空表示,所有的都可以访问,多个IP的时候用逗号隔开)
        servletRegistrationBean.addInitParameter("allow", "127.0.0.1");
        //IP黑名单(存在共同时, deny优先于allow)
        servletRegistrationBean.addInitParameter("deny", "127.0.0.2");
        //设置登录的用户名和密码
        servletRegistrationBean.addInitParameter("loginUsername", "pancm");
        servletRegistrationBean.addInitParameter("loginPassword", "123456");
        //是否能够重置数据.
        servletRegistrationBean.addInitParameter("resetEnable", "false");
        return servletRegistrationBean;
    }

    @Bean
    public FilterRegistrationBean druidStatFilter() {
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(
            new WebStatFilter());
        //添加过滤规则
        filterRegistrationBean.addUrlPatterns("/*");
        //添加不需要忽略的格式信息
        filterRegistrationBean.addInitParameter("exclusions",
            "*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*");
        System.out.println("druid初始化成功!");
        return filterRegistrationBean;
    }
}
```

编写完之后，启动程序，在浏览器输入：<http://127.0.0.1:8084/druid/index.html>，然后输入设置的用户名和密码，便可以访问Web界面了。

多数据源配置

在进行多数据源配置之前，先分别在springBoot和springBoot_test的mysql数据库中执行如下脚本。

```
-- springBoot库的脚本
```

```
CREATE TABLE `t_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '自增id',
  `name` varchar(10) DEFAULT NULL COMMENT '姓名',
  `age` int(2) DEFAULT NULL COMMENT '年龄',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=15 DEFAULT CHARSET=utf8
```

```
-- springBoot_test库的脚本
```

```
CREATE TABLE `t_student` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(16) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8
```

注:为了偷懒,将两张表的结构弄成一样了!不过不影响测试!

在application.properties中已经配置这两个数据源的信息,上面已经贴出了一次配置,这里就不再贴了。

这里重点说下 第二个数据源的配置。和上面的MasterDataSourceConfig.java差不多,区别在于没有使用@Primary注解和名称不同而已。需要注意的是MasterDataSourceConfig.java对package和mapper的扫描是精确到目录的,这里的第二个数据源也是如此。

那么代码如下:

```

@Configuration
@MapperScan(basePackages = ClusterDataSourceConfig.PACKAGE, sqlSessionFactoryRef = "clusterSqlSessionFactory")
public class ClusterDataSourceConfig {

    static final String PACKAGE = "com.pancm.dao.cluster";
    static final String MAPPER_LOCATION = "classpath:mapper/cluster/*.xml";

    @Value("${cluster.datasource.url}")
    private String url;

    @Value("${cluster.datasource.username}")
    private String username;

    @Value("${cluster.datasource.password}")
    private String password;

    @Value("${cluster.datasource.driverClassName}")
    private String driverClass;

    // 和MasterDataSourceConfig一样，这里略

    @Bean(name = "clusterDataSource")
    public DataSource clusterDataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        dataSource.setDriverClassName(driverClass);

        // 和MasterDataSourceConfig一样，这里略 ...
        return dataSource;
    }

    @Bean(name = "clusterTransactionManager")
    public DataSourceTransactionManager clusterTransactionManager() {
        return new DataSourceTransactionManager(clusterDataSource());
    }

    @Bean(name = "clusterSqlSessionFactory")
    public SqlSessionFactory clusterSqlSessionFactory(@Qualifier("clusterDataSource") DataSource clusterDataSource)
        throws Exception {
        final SqlSessionFactoryBean sessionFactory = new SqlSessionFactoryBean();
        sessionFactory.setDataSource(clusterDataSource);
        sessionFactory.setMapperLocations(new PathMatchingResourcePatternResolver().getResources(ClusterDataSourceConfig.M/return sessionFactory.getObject();
    }
}

```

成功写完配置之后，启动程序，进行测试。

分别在springBoot和springBoot_test库中使用接口进行添加数据。

t_user

```

POST http://localhost:8084/api/user
{"name":"张三","age":25}
{"name":"李四","age":25}
{"name":"王五","age":25}

```

t_student

```
POST http://localhost:8084/api/student
```

```
{"name":"学生A","age":16}  
 {"name":"学生B","age":17}  
 {"name":"学生C","age":18}
```

成功添加数据之后，然后进行调用不同的接口进行查询。

请求:

```
GET http://localhost:8084/api/user?name=李四
```

返回:

```
{  
    "id": 2,  
    "name": "李四",  
    "age": 25  
}
```

请求:

```
GET http://localhost:8084/api/student?name=学生C
```

返回:

```
{  
    "id": 1,  
    "name": "学生C",  
    "age": 16  
}
```

通过数据可以看出，成功配置了多数据源了。

PageHelper 分页实现

PageHelper是Mybatis的一个分页插件，非常的好用！这里强烈推荐！！！

PageHelper的使用很简单，只需要在Maven中添加pagehelper这个依赖就可以了。

Maven的依赖如下：

```
<dependency>  
    <groupId>com.github.pagehelper</groupId>  
    <artifactId>pagehelper-spring-boot-starter</artifactId>  
    <version>1.2.3</version>  
</dependency>
```

注：这里我是用springBoot版的！也可以使用其它版本的。

添加依赖之后，只需要添加如下配置或代码就可以了。

第一种，在application.properties或application.yml添加

```
pagehelper:  
    helperDialect: mysql  
    offsetAspageNum: true  
    rowBoundsWithCount: true  
    reasonable: false
```

第二种，在mybatis.xml配置中添加

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
    <property name="dataSource" ref="dataSource" />  
    <!-- 扫描mapping.xml文件 -->  
    <property name="mapperLocations" value="classpath:mapper/*.xml"></property>  
    <!-- 配置分页插件 -->  
    <property name="plugins">  
        <array>  
            <bean class="com.github.pagehelper.PageHelper">  
                <property name="properties">  
                    <value>  
                        helperDialect=mysql  
                        offsetAspageNum=true  
                        rowBoundsWithCount=true  
                        reasonable=false  
                    </value>  
                </property>  
            </bean>  
        </array>  
    </property>  
</bean>
```

第三种，在代码中添加，使用@Bean注解在启动程序的时候初始化。

```
@Bean  
public PageHelper pageHelper(){  
    PageHelper pageHelper = new PageHelper();  
    Properties properties = new Properties();  
    //数据库  
    properties.setProperty("helperDialect", "mysql");  
    //是否将参数offset作为pageNum使用  
    properties.setProperty("offsetAspageNum", "true");  
    //是否进行count查询  
    properties.setProperty("rowBoundsWithCount", "true");  
    //是否分页合理化  
    properties.setProperty("reasonable", "false");  
    pageHelper.setProperties(properties);  
}
```

因为这里我们使用的是多数据源，所以这里的配置稍微有些不同。我们需要在sessionFactory这里配置。这里就对MasterDataSourceConfig.java进行相应的修改。

在masterSqlSessionFactory方法中，添加如下代码。

```

@Bean(name = "masterSqlSessionFactory")
@Primary
public SqlSessionFactory masterSqlSessionFactory(@Qualifier("masterDataSource") DataSource masterDataSource)
    throws Exception {
    final SqlSessionFactoryBean sessionFactory = new SqlSessionFactoryBean();
    sessionFactory.setDataSource(masterDataSource);
    sessionFactory.setMapperLocations(new PathMatchingResourcePatternResolver()
        .getResources(MasterDataSourceConfig.MAPPER_LOCATION));
    //分页插件
    Interceptor interceptor = new PageInterceptor();
    Properties properties = new Properties();
    //数据库
    properties.setProperty("helperDialect", "mysql");
    //是否将参数offset作为PageNum使用
    properties.setProperty("offsetAsPageNum", "true");
    //是否进行count查询
    properties.setProperty("rowBoundsWithCount", "true");
    //是否分页合理化
    properties.setProperty("reasonable", "false");
    interceptor.setProperties(properties);
    sessionFactory.setPlugins(new Interceptor[] {interceptor});

    return sessionFactory.getObject();
}

```

注：其它的数据源也想进行分页的时候，参照上面的代码即可。

这里需要注意的是reasonable参数，表示分页合理化，默认值为false。如果该参数设置为true时，pageNum<=0时会查询第一页，pageNum>pages(超过总数时)，会查询最后一页。默认false时，直接根据参数进行查询。

设置完PageHelper之后，使用的话，只需要在查询的sql前面添加PageHelper.startPage(pageNum,pageSize)；如果是想知道总数的话，在查询的sql语句后买呢添加page.getTotal()就可以了。

代码示例：

```

public List<T> findByListEntity(T entity) {
    List<T> list = null;
    try {
        Page<?> page = PageHelper.startPage(1,2);
        System.out.println(getClass().getName()+"设置第一页两条数据!");
        list = getMapper().findByListEntity(entity);
        System.out.println("总共有:"+page.getTotal()+"条数据,实际返回:"+list.size()+"两条数据!");
    } catch (Exception e) {
        logger.error("查询"+getClass().getName()+"失败!原因是:",e);
    }
    return list;
}

```

代码编写完毕之后，开始进行最后的测试。

查询t_user表的所有数据，并进行分页。

请求：

```
GET http://localhost:8084/api/user
```

返回：

```
[  
 {  
   "id": 1,  
   "name": "张三",  
   "age": 25  
 },  
 {  
   "id": 2,  
   "name": "李四",  
   "age": 25  
 }  
]
```

控制台打印:

```
开始查询...  
User设置第一页两条数据!  
2018-04-27 19:55:50.769 DEBUG 6152 --- [io-8084-exec-10] c.p.d.m.UserDao.findByListEntity_COUNT :==> Preparing: SELECT cou  
2018-04-27 19:55:50.770 DEBUG 6152 --- [io-8084-exec-10] c.p.d.m.UserDao.findByListEntity_COUNT :==> Parameters:  
2018-04-27 19:55:50.771 DEBUG 6152 --- [io-8084-exec-10] c.p.d.m.UserDao.findByListEntity_COUNT :<= Total: 1  
2018-04-27 19:55:50.772 DEBUG 6152 --- [io-8084-exec-10] c.p.dao.master.UserDao.findByListEntity :==> Preparing: select id, nar  
2018-04-27 19:55:50.773 DEBUG 6152 --- [io-8084-exec-10] c.p.dao.master.UserDao.findByListEntity :==> Parameters: 2(Integer)  
2018-04-27 19:55:50.774 DEBUG 6152 --- [io-8084-exec-10] c.p.dao.master.UserDao.findByListEntity :<= Total: 2  
总共有:3条数据,实际返回:2两条数据!
```

查询t_student表的所有的数据，并进行分页。

请求:

```
GET http://localhost:8084/api/student
```

返回:

```
[  
 {  
   "id": 1,  
   "name": "学生A",  
   "age": 16  
 },  
 {  
   "id": 2,  
   "name": "学生B",  
   "age": 17  
 }  
]
```

控制台打印:

```
开始查询...  
Studnet设置第一页两条数据!  
2018-04-27 19:54:56.155 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.S.findByListEntity_COUNT :==> Preparing: SELECT count(0) FR  
2018-04-27 19:54:56.155 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.S.findByListEntity_COUNT :==> Parameters:  
2018-04-27 19:54:56.156 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.S.findByListEntity_COUNT :<= Total: 1  
2018-04-27 19:54:56.157 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.StudentDao.findByListEntity :==> Preparing: select id, name, a  
2018-04-27 19:54:56.157 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.StudentDao.findByListEntity :==> Parameters: 2(Integer)  
2018-04-27 19:54:56.157 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.StudentDao.findByListEntity :<= Total: 2  
总共有:3条数据,实际返回:2两条数据!
```

查询完毕之后，我们再来看Druid 的监控界面。

在浏览器输入:<http://127.0.0.1:8084/druid/index.html>

The screenshot shows the 'SQL Stat View JSON API' section of the Druid Monitor. It displays a table of executed SQL statements with the following columns:

N	SQL	执行数	执行时间	慢慢	事务执行	错误数	更新行数	试做行数	执行中	最大并发	执行时间分布	执行+RS时间分布	读取行分布	更新行分布
1	SELECT id, name, ag...	1					2		1	[1.0.0.0.0.0.0]	[1.0.0.0.0.0.0]	[0,1.0.0.0]	[1.0.0.0.0]	
2	select id, name, ag...	2					4		1	[2.0.0.0.0.0.0]	[2.0.0.0.0.0.0]	[0,2.0.0.0]	[2.0.0.0.0]	
3	SELECT count(0) FROM t_us...	1					1		1	[1.0.0.0.0.0.0]	[1.0.0.0.0.0.0]	[0,1.0.0.0]	[1.0.0.0.0]	
4	SELECT count(0) FROM t_st...	2	1	1			2		1	[1.1.0.0.0.0.0]	[2.0.0.0.0.0.0]	[0,2.0.0.0]	[2.0.0.0.0]	

可以很清晰的看到操作记录！

如果想知道更多的Druid相关知识，可以查看官方文档！

结语

这篇终于写完了，在进行代码编写的时候，碰到过很多问题，然后慢慢的尝试和找资料解决了。本篇文章只是很浅的介绍了这些相关的使用，在实际的应用可能会更复杂。如果有有更好的想法和建议，欢迎留言进行讨论！

参考文章：

<https://www.bysocket.com/?p=1712>

Durid官方地址：

<https://github.com/alibaba/druid>

PageHelper官方地址：

<https://github.com/pagehelper/Mybatis-PageHelper>

文中源码：

<https://github.com/xuwujing/springBoot>

- END -



喜欢文章, 点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

Spring Boot + MyBatis 多模块项目搭建教程

枫本非凡 Java后端 2019-09-30

点击上方Java后端, 选择“[设为星标](#)”

优质文章, 及时送达

作者 | 枫本非凡

链接 | cnblogs.com/orzlin/p/9717399.html

上篇 | IDEA 远程一键部署 Spring Boot 到 Docker

一、前言

最近公司项目准备开始重构，框架选定为SpringBoot+Mybatis，本篇主要记录了在IDEA中搭建SpringBoot多模块项目的过程。

1、开发工具及系统环境

- IDE:

IntelliJ IDEA 2018.2

- 系统环境:

mac OSX

2、项目目录结构

- biz层:

业务逻辑层

- dao层:

数据持久层

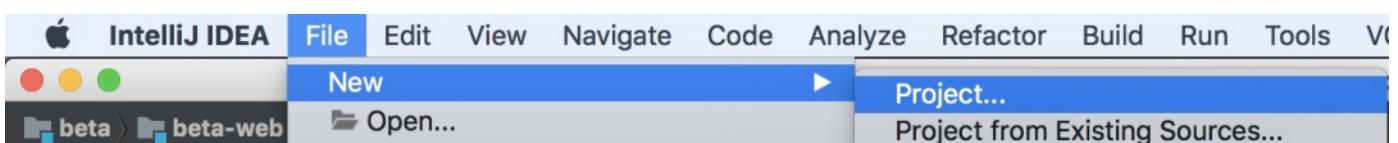
- web层:

请求处理层

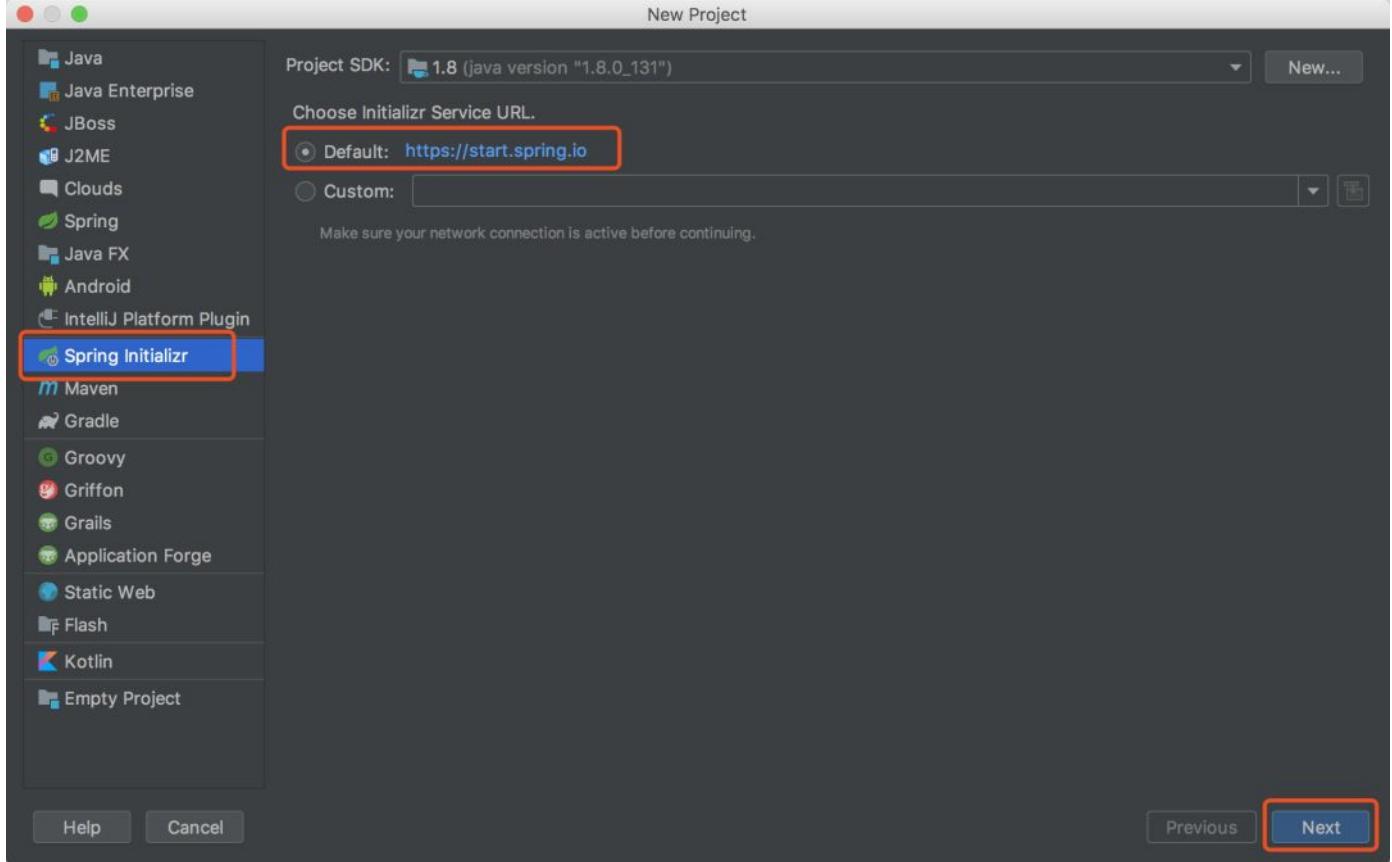
二、搭建步骤

1、创建父工程

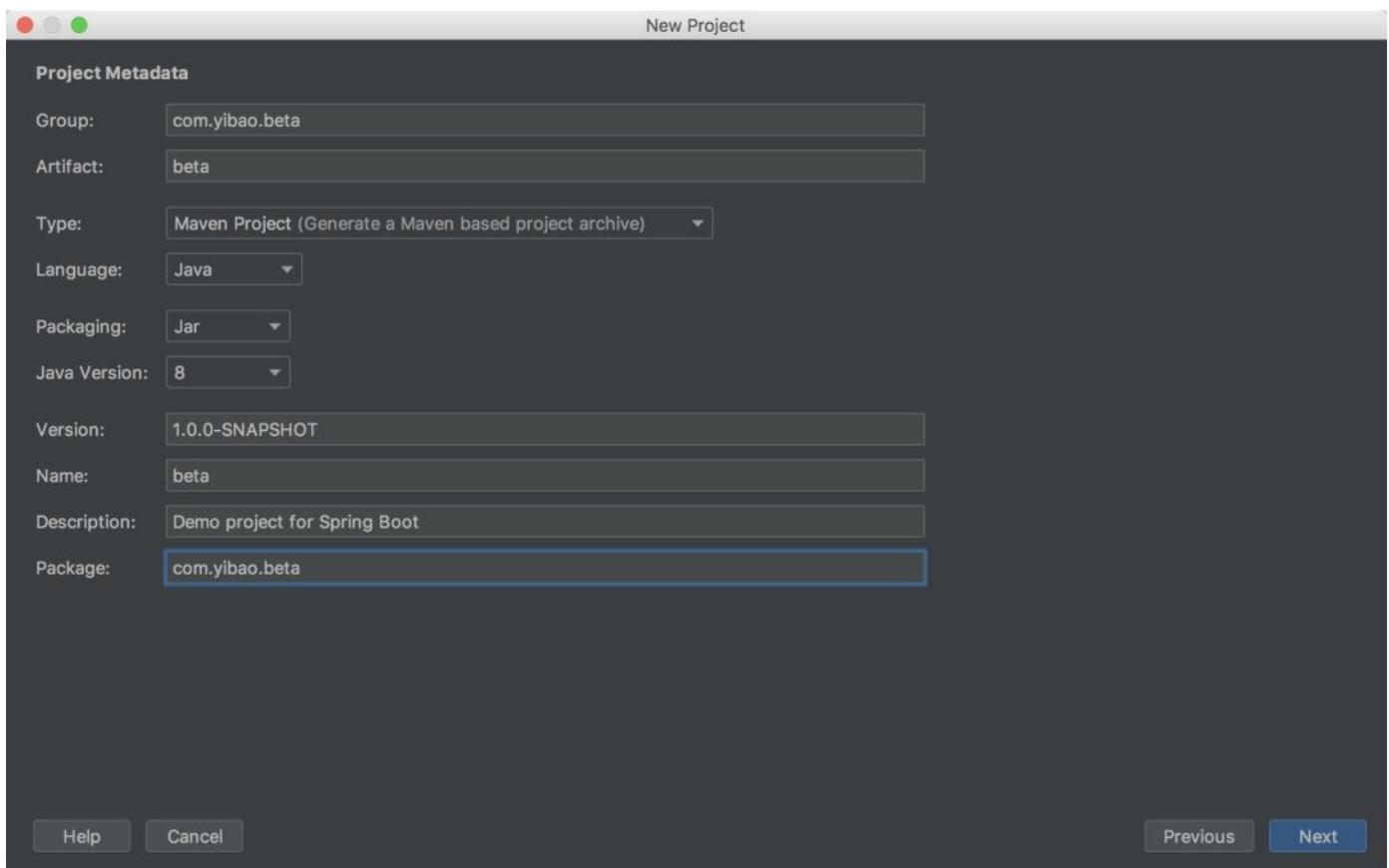
IDEA 工具栏选择菜单 File -> New -> Project...



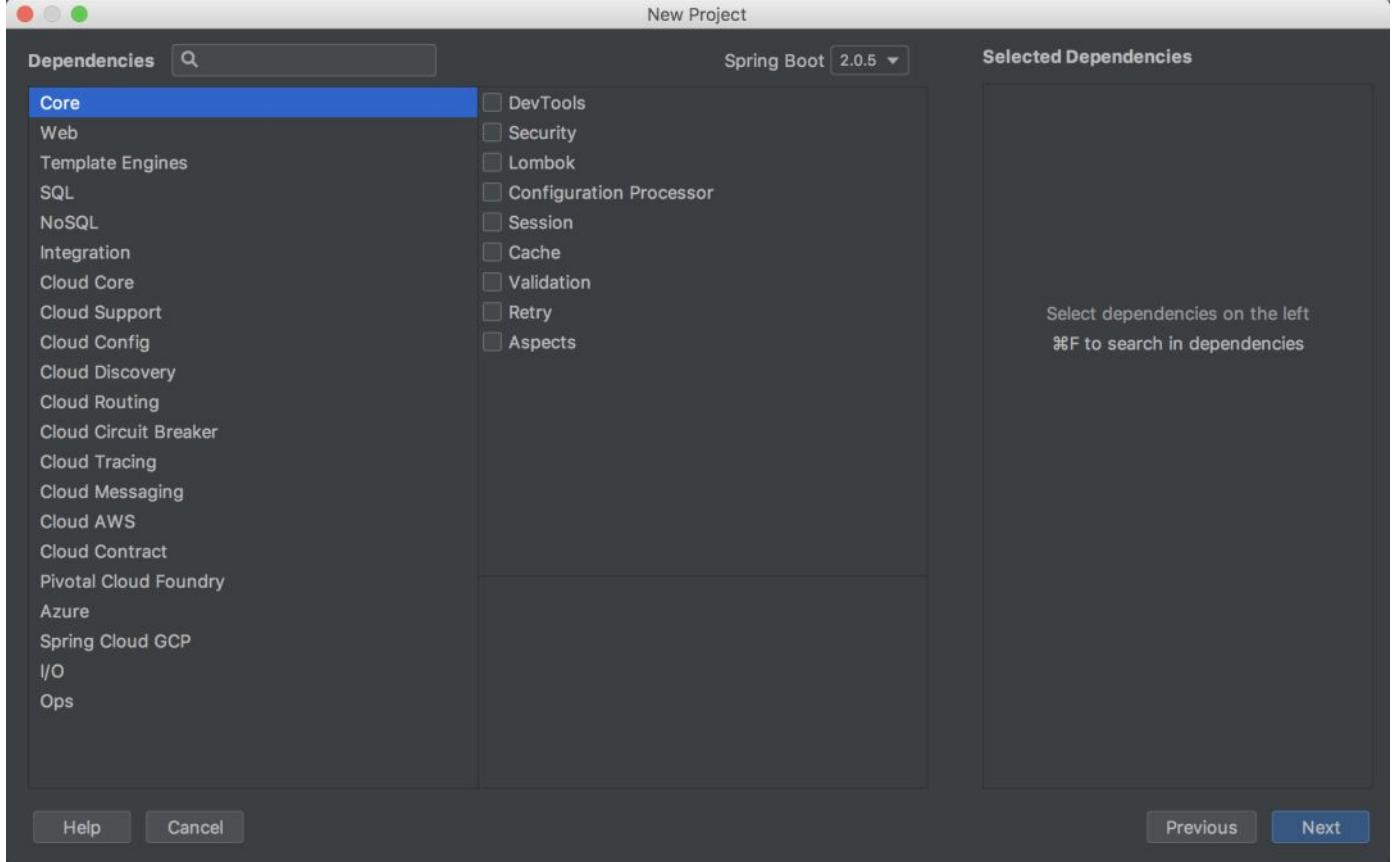
选择Spring Initializr, Initializr默认选择Default, 点击Next



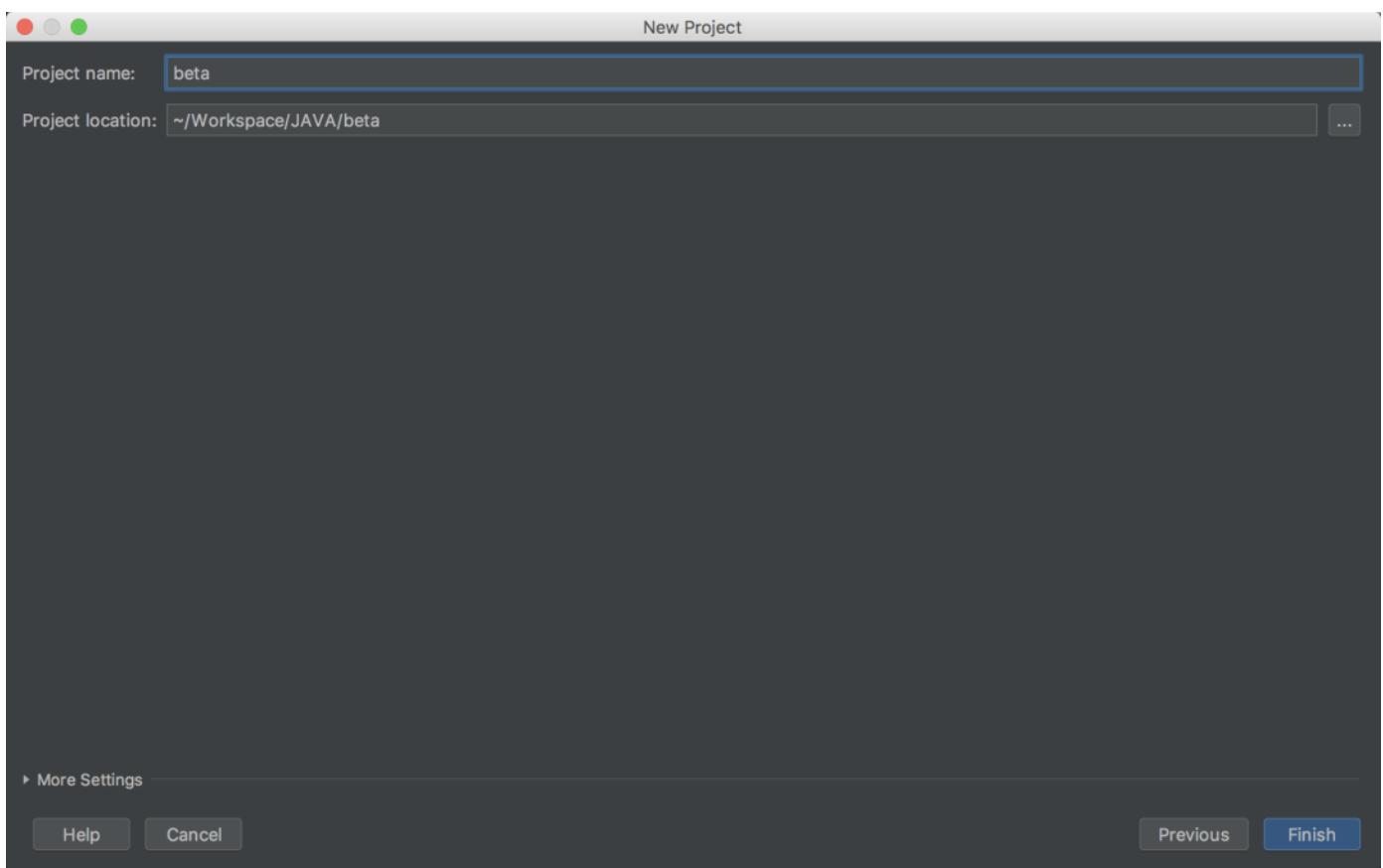
填写输入框，点击Next



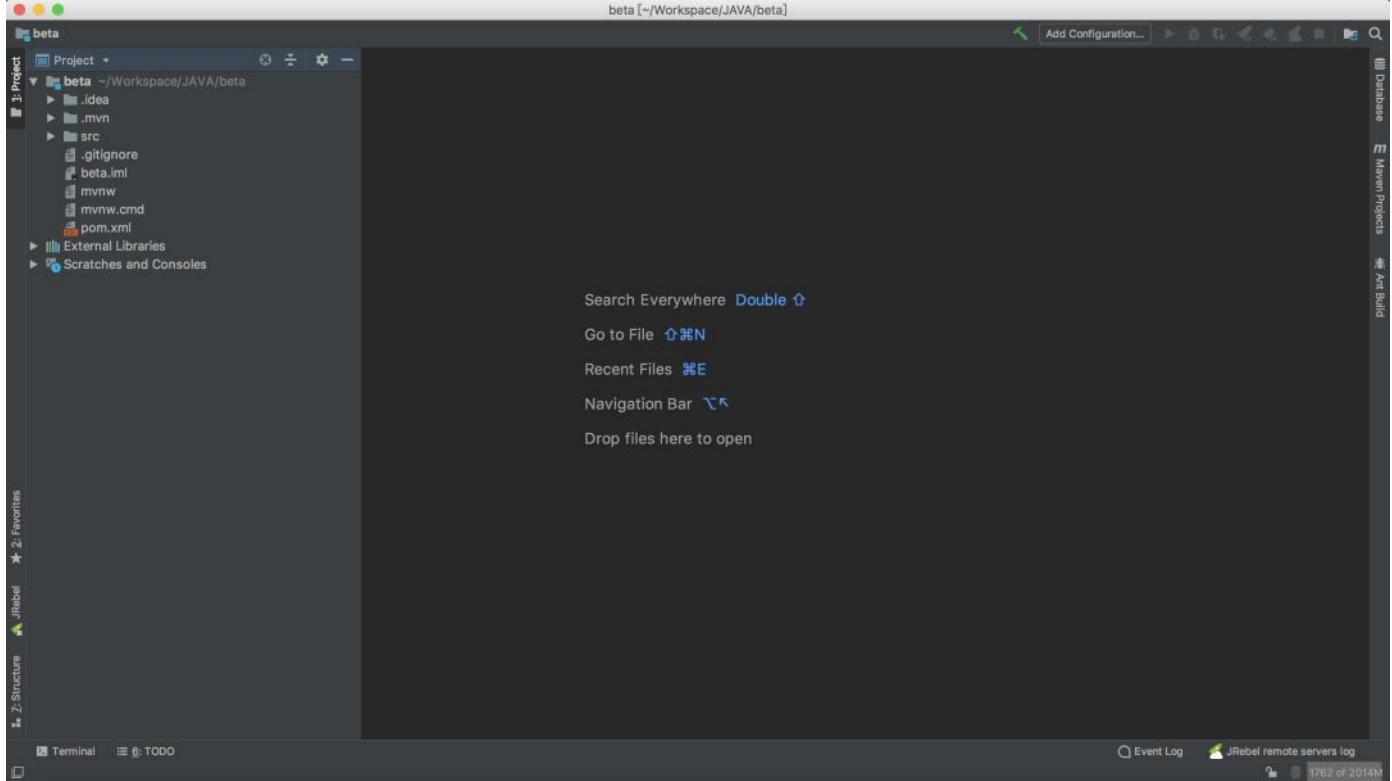
这步不需要选择直接点Next



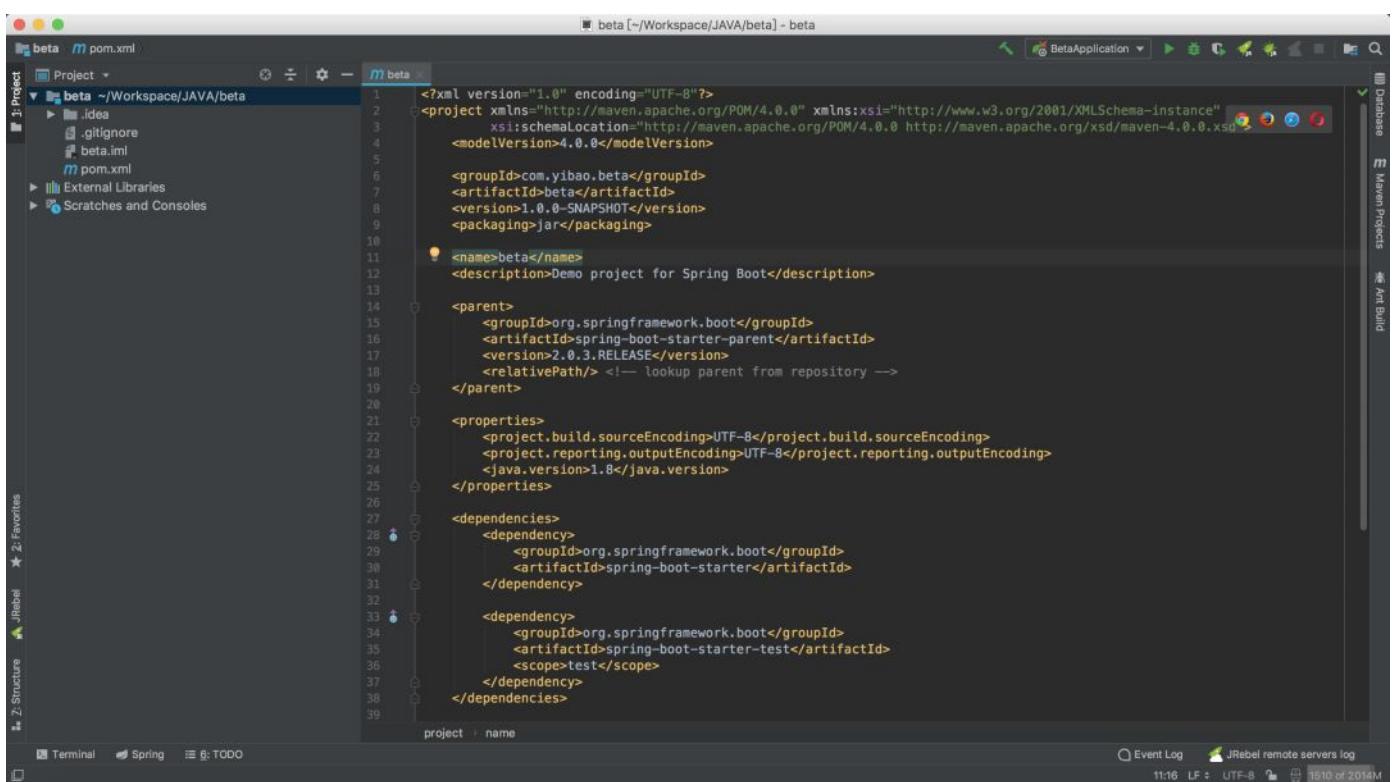
点击Finish创建项目



最终得到的项目目录结构如下

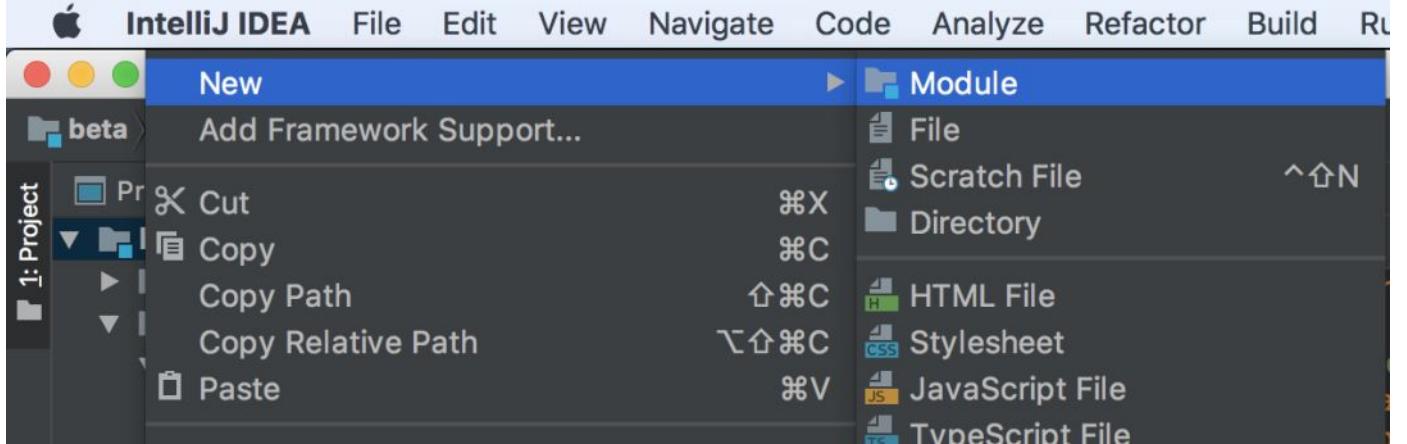


删除无用的.mvn目录、src目录、mvnw及mvnw.cmd文件，最终只留.gitignore和pom.xml

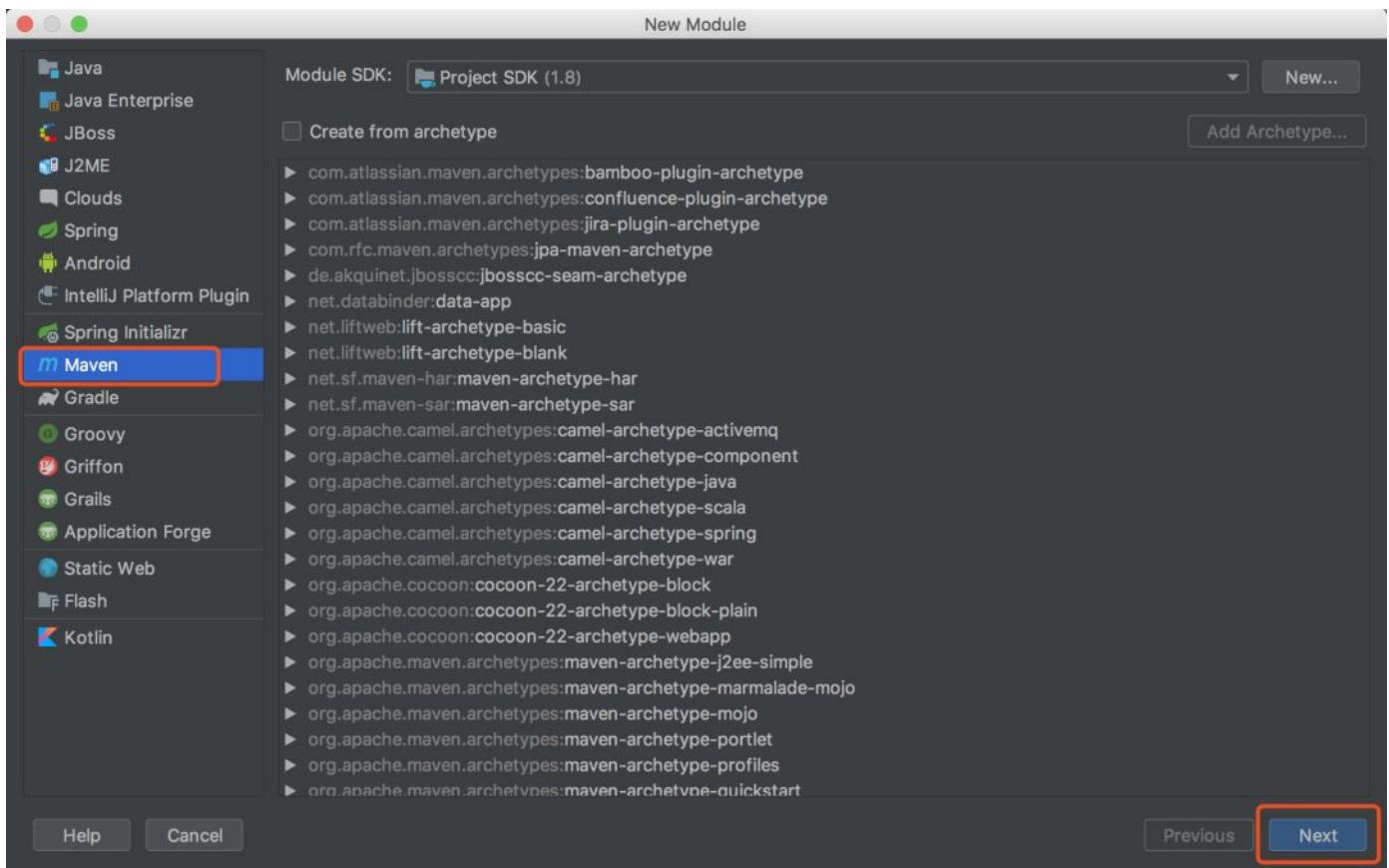


2、创建子模块

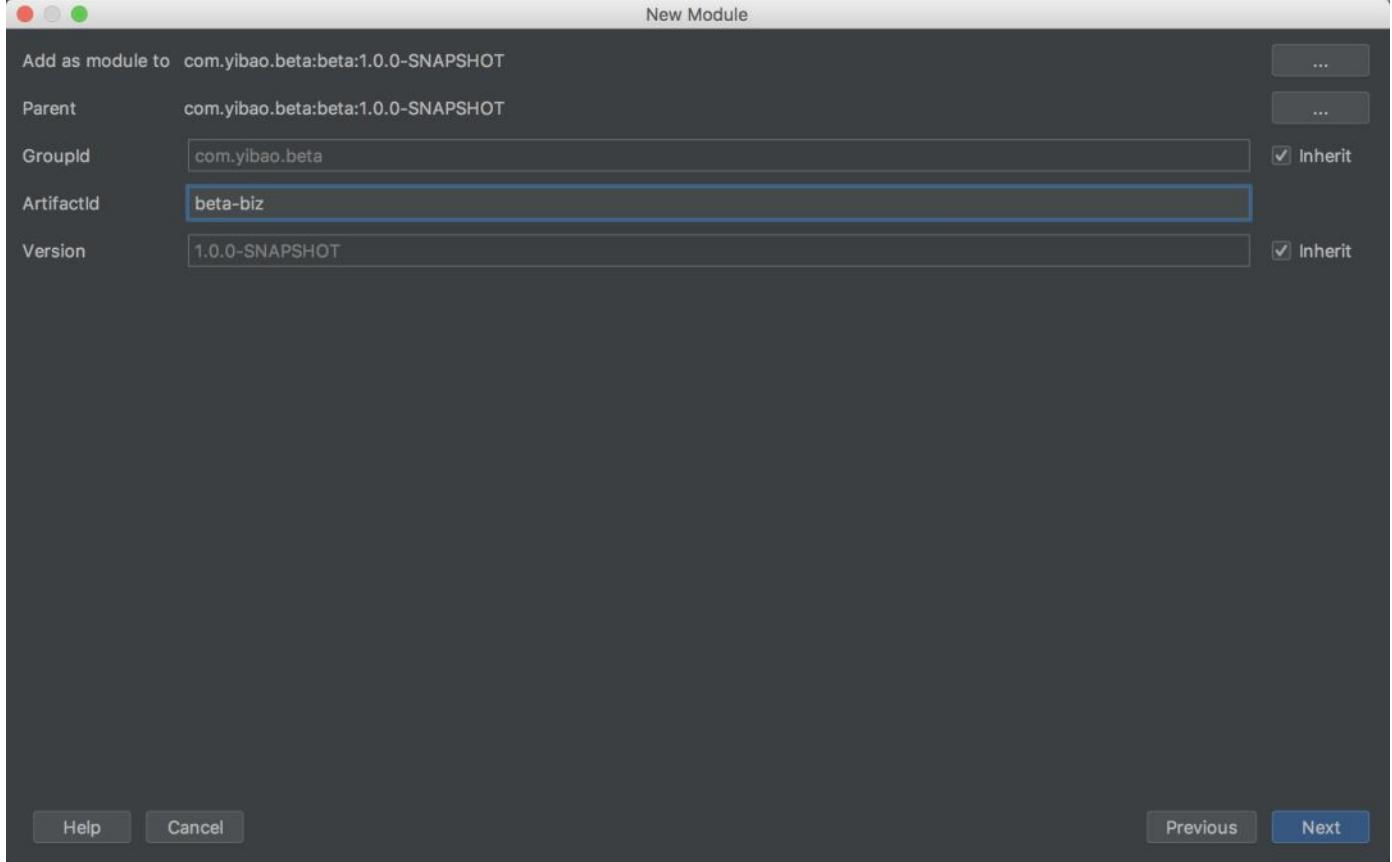
选择项目根目录beta右键呼出菜单，选择New -> Module



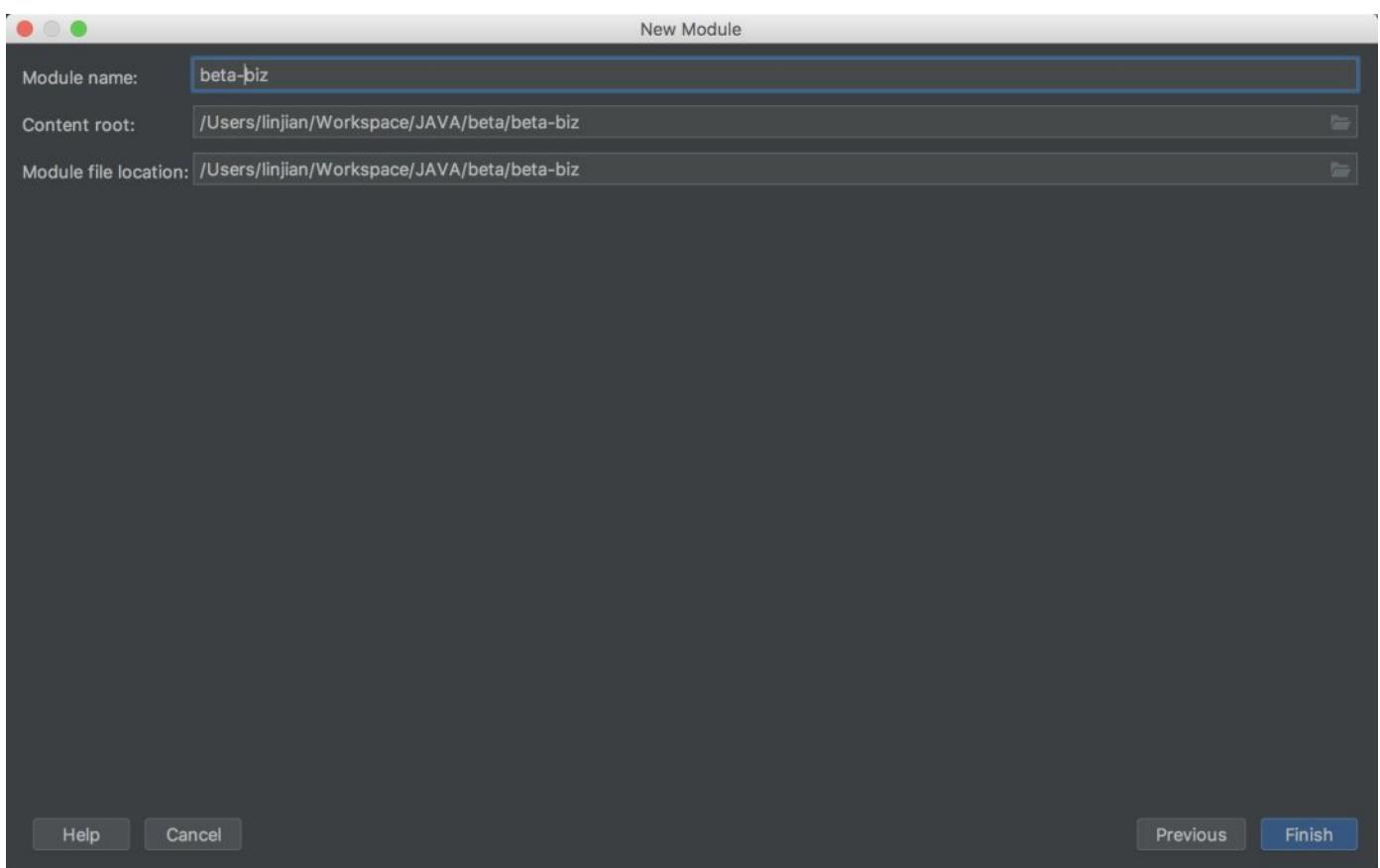
选择Maven，点击Next



填写ArtifactId，点击Next



修改Module name增加横杠提升可读性，点击Finish



同理添加beta-dao、beta-web子模块，最终得到项目目录结构如下图

```
<?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.yibao.beta</groupId>
  <artifactId>beta</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <modules>
    <module>beta-biz</module>
    <module>beta-dao</module>
    <module>beta-web</module>
  </modules>
  <packaging>pom</packaging>
  <name>beta</name>
  <description>Demo project for Spring Boot</description>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.3.RELEASE</version>
    <relativePath/> 
  </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</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
```

3、运行项目

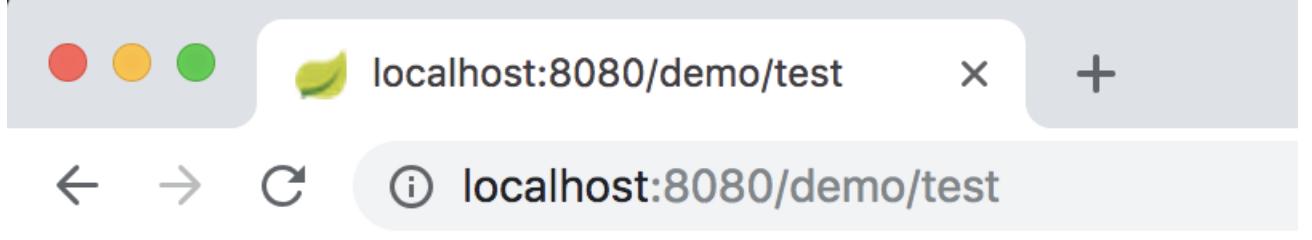
在beta-web层创建com.yibao.beta.web包(注意:这是多层目录结构并非单个目录名,com >> yibao >> beta >> web)并添加入口类BetaWebApplication.java

```
1 @SpringBootApplication
2 public class BetaWebApplication {
3
4     public static void main(String[] args)
5     {
6         SpringApplication.run(BetaWebApplication.class, args);
7     }
8 }
```

在com.yibao.beta.web包中添加controller目录并新建一个controller，添加test方法测试接口是否可以正常访问

```
1 @RestController
2 @RequestMapping("demo")
3 public class DemoController {
4
5     @GetMapping("test")
6     public String test() {
7         return "Hello World!";
8     }
9 }
```

运行BetaWebApplication类中的main方法启动项目，默认端口为8080，访问<http://localhost:8080/demo/test>得到如下效果



Hello World!

以上虽然项目能正常启动，但是模块间的依赖关系却还未添加，下面继续完善。微信搜索 `web_resource` 获取更多推送

4、配置模块间的依赖关系

各个子模块的依赖关系：biz层依赖dao层，web层依赖biz层

父pom文件中声明所有子模块依赖（dependencyManagement及dependencies的区别自行查阅文档）

```
1 <dependencyManagement>
2   <dependencies>
3   >
4     <dependency>
5   >
6       <groupId>com.yibao.beta</groupId>
7   >
8       <artifactId>beta-biz</artifactId>
9   >
10      <version>${beta.version}</version>
11  >
12      </dependency>
13  >
14      <dependency>
15  >
16        <groupId>com.yibao.beta</groupId>
17  >
18        <artifactId>beta-dao</artifactId>
19  >
20        <version>${beta.version}</version>
21  >
22      </dependency>
23  >
24      <dependency>
25  >
26        <groupId>com.yibao.beta</groupId>
27  >
28        <artifactId>beta-web</artifactId>
29  >
30        <version>${beta.version}</version>
31  >
32      </dependency>
33  >
34  </dependencies>
```

```
</dependencyManagement>
```

其中\${beta.version}定义在properties标签中

在beta-web层中的pom文件中添加beta-biz依赖

```
1 <dependencies>
2     <dependency>
3     >
4         <groupId>com.yibao.beta</groupId>
5     >
6         <artifactId>beta-biz</artifactId>
7     >
8     </dependency>
9 >
</dependencies>
```

在beta-biz层中的pom文件中添加beta-dao依赖

```
1 <dependencies>
2     <dependency>
3     >
4         <groupId>com.yibao.beta</groupId>
5     >
6         <artifactId>beta-dao</artifactId>
7     >
8     </dependency>
9 >
</dependencies>
```

4. web层调用biz层接口测试

在beta-biz层创建com.yibao.beta.biz包，添加service目录并在其中创建DemoService接口类，[微信搜索 web_resource 获取更多推送](#)

```
1 public interface DemoService {
2     String test()
3 }
4 }
```

```
1 @Service
2 public class DemoServiceImpl implements DemoService {
3
4     @Override
5     public String test()
6     {
7         return "test"
8     }
9 }
```

```
8     }
9 }
```

DemoController通过@.Autowired注解注入DemoService，修改DemoController的test方法使之调用DemoService的test方法，最终如下所示：

```
1 package com.yibao.beta.web.controller;@RestController
2 @RequestMapping("demo")
3 public class DemoController {
4
5     @Autowired
6     private DemoService demoService;
7
8     @GetMapping("test"
9     )
10    public String test() {
11        return demoService.test();
12    }
13}
```

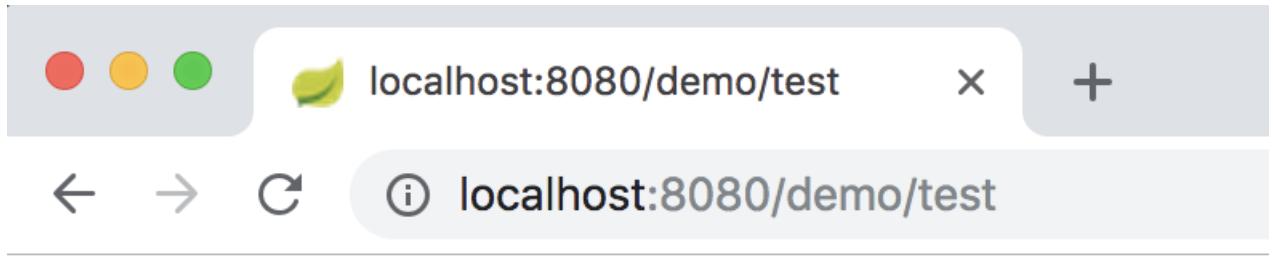
再次运行BetaWebApplication类中的main方法启动项目，发现如下报错

```
1 ****
2 APPLICATION FAILED TO START
3 ****
4
5 Description:
6 Field demoService in com.yibao.beta.web.controller.DemoController required a bean of type 'com.yibao.beta.biz.service.DemoService' which could not be found.
7
8 Action:
9 Consider defining a bean of type 'com.yibao.beta.biz.service.DemoService' in your configuration.
```

原因是找不到DemoService类，此时需要在BetaWebApplication入口类中增加包扫描，设置@SpringBootApplication注解中的scanBasePackages值为com.yibao.beta，最终如下所示

```
1 package com.yibao.beta.web;
2
3 @SpringBootApplication(scanBasePackages = "com.yibao.beta"
4 )
5 @MapperScan("com.yibao.beta.dao.mapper")
6 public class BetaWebApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(BetaWebApplication.class, args)
10    ;
11    }
12}
```

设置完后重新运行main方法，项目正常启动，访问http://localhost:8080/demo/test得到如下效果



6. 集成Mybatis

父pom文件中声明mybatis-spring-boot-starter及lombok依赖

```
1 <dependencyManagement>
2   <dependencies>
3   >
4     <dependency>
5     >
6       <groupId>org.mybatis.spring.boot</groupId>
7     >
8       <artifactId>mybatis-spring-boot-starter</artifactId>
9     >
10      <version>1.3.2</version>
11    >
12    </dependency>
13  >
14  <dependency>
15  >
16    <groupId>org.projectlombok</groupId>
17  >
18    <artifactId>lombok</artifactId>
19  >
20    <version>1.16.22</version>
21  >
22    </dependency>
23  >
24  </dependencies>
25 >
26 </dependencyManagement>
```

在beta-dao层中的pom文件中添加上述依赖

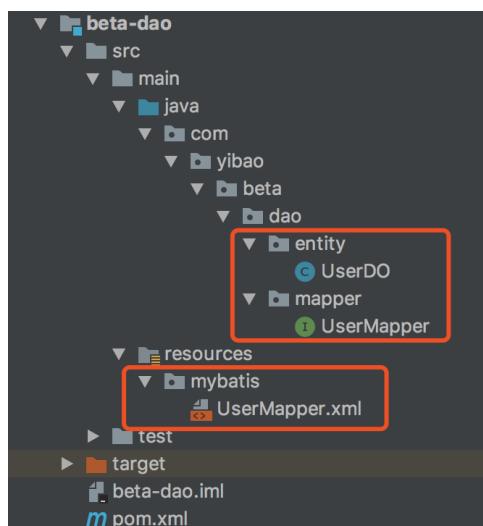
```
1 <dependencies>
2   <dependency>
3   >
4     <groupId>org.mybatis.spring.boot</groupId>
5   >
```

```

6   <artifactId>mybatis-spring-boot-starter</artifactId>
7   >
8     </dependency>
9   >
10    <dependency>
11      >
12        <groupId>mysql</groupId>
13      >
14        <artifactId>mysql-connector-java</artifactId>
15      >
16        </dependency>
17      >
18        <dependency>
19          >
20            <groupId>org.projectlombok</groupId>
21          >
22            <artifactId>lombok</artifactId>
23          >
24            </dependency>
25          >
26        </dependencies>

```

在beta-dao层创建com.yibao.beta.dao包，通过mybatis-generator工具生成dao层相关文件（DO、Mapper、xml），存放目录如下



application.properties文件添加jdbc及mybatis相应配置项

```

1 spring.datasource.driverClassName = com.mysql.jdbc.Driver
2 spring.datasource.url = jdbc:mysql://192.168.1.1/test?useUnicode=true&characterEncoding=utf-8
3 spring.datasource.username = test
4 spring.datasource.password = 123456
5
6 mybatis.mapper-locations = classpath:mybatis/*.xml
7 mybatis.type-aliases-package = com.yibao.beta.dao.entity
8

```

DemoService通过@Autowired注解注入UserMapper，修改DemoService的test方法使之调用UserMapper的

selectByPrimaryKey方法，最终如下所示

```
1 package com.yibao.beta.biz.service.impl;
2
3 @Service
4 public class DemoServiceImpl implements DemoService {
5
6     @Autowired
7     private UserMapper userMapper;
8
9     @Override
10    public String test()
11    {
12        UserDO user = userMapper.selectByPrimaryKey(1)
13    ;
14        return user.toString();
15    }
16}
```

再次运行BetaWebApplication类中的main方法启动项目，发现如下报错

```
1 APPLICATION FAILED TO START
*****
2
3
4 Description:
5 Field userMapper in com.yibao.beta.biz.service.impl.DemoServiceImpl required a bean of type 'com.y
6 .
7
8
9 Action:
10 Consider defining a bean of type 'com.yibao.beta.dao.mapper.UserMapper' in your configuration.
```

原因是找不到UserMapper类，此时需要在BetaWebApplication入口类中增加dao层包扫描，添加@MapperScan注解并设置其值为com.yibao.beta.dao.mapper，最终如下所示

```
1 package com.yibao.beta.web;
2
3 @SpringBootApplication(scanBasePackages = "com.yibao.beta"
4 )
5 @MapperScan("com.yibao.beta.dao.mapper")
6 public class BetaWebApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(BetaWebApplication.class, args)
10    ;
11        }
12 }
```

设置完后重新运行main方法，项目正常启动，访问http://localhost:8080/demo/test得到如下效果



localhost:8080/demo/test

UserDO(id=1, userName=linjian)

至此，一个简单的SpringBoot+Mybatis多模块项目已经搭建完毕，我们也通过启动项目调用接口验证其正确性。

四、总结

一个层次分明的多模块工程结构不仅方便维护，而且有利于后续微服务化。在此结构的基础上还可以扩展common层（公共组件）、server层（如dubbo对外提供的服务）[微信搜索 web_resource 获取更多推送](#)

此为项目重构的第一步，后续还会在框架中集成logback、disconf、redis、dubbo等组件

五、未提到的坑

在搭建过程中还遇到一个maven私服的问题，原因是公司内部的maven私服配置的中央仓库为阿里的远程仓库，它与maven自带的远程仓库相比有些jar包版本并不全，导致在搭建过程中好几次因为没拉到相应jar包导致项目启动不了。

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. Java后端优质文章整理
2. IDEA 远程一键部署 Spring Boot 到 Docker
3. 这 26 条，你赞同几个？
4. 7 个开源的 Spring Boot 前后端分离项目

5. 如何设计 API 接口, 实现统一格式返回?



Java后端

长按识别二维码, 关注我的公众号

喜欢文章, 点个在看 

[阅读原文](#)

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

Spring Boot + Mybatis 配合 AOP 和注解实现动态数据源切换配置

韩数 Java后端 2019-10-25

点击上方 Java后端, 选择 [设为星标](#)

技术博文, 及时送达

来自 | 韩数

链接 | juejin.im/post/5d830944f265da03963bd153

0、前言

随着应用用户数量的增加，相应的并发请求的数量也会跟着不断增加，慢慢地，单个数据库已经没有办法满足我们频繁的数据库操作请求了。

在某些场景下，我们可能会需要配置多个数据源，使用多个数据源(例如实现数据库的读写分离)来缓解系统的压力等，同样的，Springboot官方提供了相应的实现来帮助开发者们配置多数据源，一般分为两种方式(目前我所了解到的)，分包和AOP。

而在Springboot +Mybatis实现多数据源配置中，我们实现了静态多数据源的配置，但是这种方式怎么说呢，在实际的使用中不够灵活，为了解决这个问题，我们可以使用上文提到的第二种方法,即使用AOP面向切面编程的方式配合我们的自定义注解来实现不同数据源之间动态切换的目的。

1、数据库准备

数据库准备仍然和之前的例子相同，具体建表sql语句则不再详细说明，表格如下：

数据库	testdatasource1	testdatasource2
数据表	sys_user	sys_user2
字段	user_id(int), user_name(varchar) user_age (int)	同

并分别插入两条记录，为了方便对比，其中testdatasource1为芳年25岁的张三， testdatasource2为芳年30岁的李四。

2、环境准备

首先新建一个Springboot项目，我这里版本是2.1.7.RELEASE，并在pom文件中引入相关依赖，和上次相比，这次主要额外新增了aop相关的依赖，如下：

```
1 <dependency>
2   <groupId>mysql</groupId>
3   <artifactId>mysql-connector-java</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.boot</groupId>
7   <artifactId>spring-boot-starter-jdbc</artifactId>
8 </dependency>
9 <dependency>
10  <groupId>org.springframework</groupId>
```

```
11 <artifactId>spring-aop</artifactId>
12 <version>5.1.5.RELEASE</version>
13 </dependency>
14 <dependency>
15   <groupId>junit</groupId>
16   <artifactId>junit</artifactId>
17 </dependency>
18 <dependency>
19   <groupId>org.aspectj</groupId>
20   <artifactId>aspectjweaver</artifactId>
21   <version>1.9.2</version>
22 </dependency>
```

3、代码部分

首先呢，在我们Springboot的配置文件中配置我们的datasource，和以往不一样的是，因为我们有两个数据源，所以要指定相关数据库的名称，其中主数据源为primary，次数据源为secondary如下：

```
1 #配置主数据库
2 spring.datasource.primary.jdbc-url=jdbc:mysql://localhost:3306/testdatasource1?useUnicode=true&cha
3 spring.datasource.primary.username=root
4 spring.datasource.primary.password=root
5 spring.datasource.primary.driver-class-name=com.mysql.cj.jdbc.Driver
6
7 ##配置次数据库
8 spring.datasource.secondary.jdbc-url=jdbc:mysql://localhost:3306/testdatasource2?useUnicode=true&c
9 spring.datasource.secondary.username=root
10 spring.datasource.secondary.password=root
11 spring.datasource.secondary.driver-class-name=com.mysql.cj.jdbc.Driver
12
13
14 spring.http.encoding.charset=UTF-8
15 spring.http.encoding.enabled=true
16 spring.http.encoding.force=true
```

需要我们注意的是，Springboot2.0 在配置数据库连接的时候需要使用jdbc-url，如果只使用url的话会报 **jdbcUrl is required with driverClassName.** 错误。

新建一个配置文件，DynamicDataSourceConfig 用来配置我们相关的bean,代码如下

```
1 @Configuration
2 @MapperScan(basePackages = "com.mzd.multipledatasources.mapper", sqlSessionFactoryRef = "SqlSession"
3 public class DynamicDataSourceConfig {
4
5     // 将这个对象放入Spring容器中
6     @Bean(name = "PrimaryDataSource")
7     // 表示这个数据源是默认数据源
8     @Primary
9     // 读取application.properties中的配置参数映射成为一个对象
10    // prefix表示参数的前缀
11    @ConfigurationProperties(prefix = "spring.datasource.primary")
```

```

12     public DataSource getDateSource1() {
13         return DataSourceBuilder.create().build();
14     }
15
16
17     @Bean(name = "SecondaryDataSource")
18     @ConfigurationProperties(prefix = "spring.datasource.secondary")
19     public DataSource getDateSource2() {
20         return DataSourceBuilder.create().build();
21     }
22
23
24     @Bean(name = "dynamicDataSource")
25     public DynamicDataSource DataSource(@Qualifier("PrimaryDataSource") DataSource primaryDataSour
26                                         @Qualifier("SecondaryDataSource") DataSource secondaryDataS
27
28         //这个地方是比较核心的targetDataSource 集合是我们数据库和名字之间的映射
29         Map<Object, Object> targetDataSource = new HashMap<>();
30         targetDataSource.put(DataSourceType.DataBaseType.Primary, primaryDataSource);
31         targetDataSource.put(DataSourceType.DataBaseType.Secondary, secondaryDataSource);
32         DynamicDataSource dataSource = new DynamicDataSource();
33         dataSource.setTargetDataSources(targetDataSource);
34         dataSource.setDefaultTargetDataSource(primaryDataSource); //设置默认对象
35         return dataSource;
36     }
37
38
39     @Bean(name = "SqlSessionFactory")
40     public SqlSessionFactory SqlSessionFactory(@Qualifier("dynamicDataSource") DataSource dynamicD
41             throws Exception {
42         SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
43         bean.setDataSource(dynamicDataSource);
44         bean.setMapperLocations(
45             new PathMatchingResourcePatternResolver().getResources("classpath*:mapping/*/*.xml"));
46         return bean.getObject();
47     }
48 }
```

而在这所有的配置中，最核心的地方就是DynamicDataSource这个类了，DynamicDataSource是我们自定义的动态切换数据源的类，该类继承了AbstractRoutingDataSource 类并重写了它的determineCurrentLookupKey()方法。

AbstractRoutingDataSource 类内部维护了一个名为targetDataSources的Map，并提供的setter方法用于设置数据源关键字与数据源的关系，实现类被要求实现其determineCurrentLookupKey()方法，由此方法的返回值决定具体从哪个数据源中获取连接。同时AbstractRoutingDataSource类提供了程序运行时动态切换数据源的方法，在dao类或方法上标注需要访问数据源的关键字，路由到指定数据源，获取连接。

DynamicDataSource代码如下：

```

1  public class DynamicDataSource extends AbstractRoutingDataSource {
2
3     @Override
4     protected Object determineCurrentLookupKey() {
5         DataBaseType DataBaseType = DataBaseType.getDataBaseType();
```

```
5     return DataBaseType;
6 }
7
8 }
9
```

DataSourceType类的代码如下：

```
1 public class DataSourceType {
2
3     //内部枚举类，用于选择特定的数据类型
4     public enum DataBaseType {
5         Primary, Secondary
6     }
7
8     // 使用ThreadLocal保证线程安全
9     private static final ThreadLocal<DataBaseType> TYPE = new ThreadLocal<DataBaseType>();
10
11    // 往当前线程里设置数据源类型
12    public static void set DataBaseType(DataBaseType DataBaseType) {
13        if (dataBaseType == null) {
14            throw new NullPointerException();
15        }
16        TYPE.set(dataBaseType);
17    }
18
19    // 获取数据源类型
20    public static DataBaseType get DataBaseType() {
21        DataBaseType DataBaseType = TYPE.get() == null ? DataBaseType.Primary : TYPE.get();
22        return DataBaseType;
23    }
24
25    // 清空数据类型
26    public static void clear DataBaseType() {
27        TYPE.remove();
28    }
29
30 }
```

接下来编写我们相关的Mapper和xml文件，代码如下：

```
1 @Component
2 @Mapper
3 public interface PrimaryUserMapper {
4
5     List<User> findAll();
6 }
7
8 @Component
9 @Mapper
10 public interface SecondaryUserMapper {
11
12     List<User> findAll();
13 }
```

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5  <mapper namespace="com.jdkcb.mybatisstuday.mapper.one.PrimaryUserMapper">
6
7      <select id="findAll" resultType="com.jdkcb.mybatisstuday.pojo.User">
8          select * from sys_user;
9      </select>
10
11 </mapper>
12
13
14 <?xml version="1.0" encoding="UTF-8" ?>
15 <!DOCTYPE mapper
16     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
17     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
18 <mapper namespace="com.jdkcb.mybatisstuday.mapper.two.SecondaryUserMapper">
19
20     <select id="findAll" resultType="com.jdkcb.mybatisstuday.pojo.User">
21         select * from sys_user2;
22     </select>
23
24
25 </mapper>

```

相关接口文件编写好之后，就可以编写我们的aop代码了：

```

1 @Aspect
2 @Component
3 public class DataSourceAop {
4     //在primary方法前执行
5     @Before("execution(* com.jdkcb.mybatisstuday.controller.UserController.primary(..))")
6     public void setDataSource2test01() {
7         System.out.println("Primary业务");
8         DataSourceType.setDatabaseType(DataSourceType.DatabaseType.Primary);
9     }
10
11     //在secondary方法前执行
12     @Before("execution(* com.jdkcb.mybatisstuday.controller.UserController.secondary(..))")
13     public void setDataSource2test02() {
14         System.out.println("Secondary业务");
15         DataSourceType.setDatabaseType(DataSourceType.DatabaseType.Secondary);
16     }
17 }

```

编写我们的测试 UserController，代码如下：

```

1 @RestController

```

```
2 public class UserController {  
3  
4     @Autowired  
5     private PrimaryUserMapper primaryUserMapper;  
6     @Autowired  
7     private SecondaryUserMapper secondaryUserMapper;  
8  
9  
10    @RequestMapping("primary")  
11    public Object primary(){  
12        List<User> list = primaryUserMapper.findAll();  
13        return list;  
14    }  
15    @RequestMapping("secondary")  
16    public Object secondary(){  
17        List<User> list = secondaryUserMapper.findAll();  
18        return list;  
19    }  
20  
21 }
```

4、测试

启动项目，在浏览器中分别输入http://127.0.0.1:8080/primary 和http://127.0.0.1:8080/secondary，结果如下：

```
1 [{"user_id":1,"user_name":"张三","user_age":25}]  
2 [{"user_id":1,"user_name":"李四","user_age":30}]
```

5、等等

等等，啧啧啧，我看你这不行啊，还不够灵活，几个菜啊，喝成这样，这就算灵活了？

那肯定不能的，aop也有aop的好处，比如两个包下的代码分别用两个不同的数据源，就可以直接用aop表达式就可以完成了，但是，如果想本例中方法级别的拦截，就显得优点不太灵活了，这个适合就需要我们的注解上场了。

6、配合注解实现

首先自定义我们的注解 @DataSource

```
1 /**  
2  * 切换数据注解 可以用于类或者方法级别 方法级别优先级 > 类级别  
3  */  
4 @Target({ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER})  
5 @Retention(RetentionPolicy.RUNTIME)  
6 @Documented  
7 public @interface DataSource {  
8     String value() default "primary"; //该值即key值， 默认使用默认数据库  
9 }
```

通过使用aop拦截，获取注解的属性value的值。如果value的值并没有在我们DataBaseType里面，则使用我们默认的数据源，如果有的话，则切换为相应的数据源。

```
1 @Aspect
2 @Component
3 public class DynamicDataSourceAspect {
4     private static final Logger logger = LoggerFactory.getLogger(DynamicDataSourceAspect.class);
5
6     @Before("@annotation(dataSource)");//拦截我们的注解
7     public void changeDataSource(JoinPoint point, DataSource dataSource) throws Throwable {
8         String value = dataSource.value();
9         if (value.equals("primary")){
10             DataSourceType.setDatabaseType(DataSourceType.DatabaseType.Primary);
11         }else if (value.equals("secondary")){
12             DataSourceType.setDatabaseType(DataSourceType.DatabaseType.Secondary);
13         }else {
14             DataSourceType.setDatabaseType(DataSourceType.DatabaseType.Primary);//默认使用主数据库
15         }
16
17     }
18
19     @After("@annotation(dataSource) ") //清除数据源的配置
20     public void restoreDataSource(JoinPoint point, DataSource dataSource) {
21         DataSourceType.clearDatabaseType();
22
23
24     }
25 }
```

7、测试

修改我们的mapper，添加我们的自定义的@DataSource注解，并注解掉我们DataSourceAop类里面的内容。如下：

```
1 @Component
2 @Mapper
3 public interface PrimaryUserMapper {
4
5     @DataSource
6     List<User> findAll();
7 }
8
9 @Component
10 @Mapper
11 public interface SecondaryUserMapper {
12
13     @DataSource("secondary")//指定数据源为：secondary
14     List<User> findAll();
15 }
```

启动项目，在浏览器中分别输入http://127.0.0.1:8080/primary 和http://127.0.0.1:8080/secondary，结果如下：

```
1 [{"user_id":1,"user_name":"张三", "user_age":25}]\n2 [{"user_id":1,"user_name":"李四", "user_age":30}]
```

到此，就算真正的大功告成啦。

8、源码

github.com/hanshuaiKang/HanShu-Note

学Java，请关注公众号：Java后端



Java后端

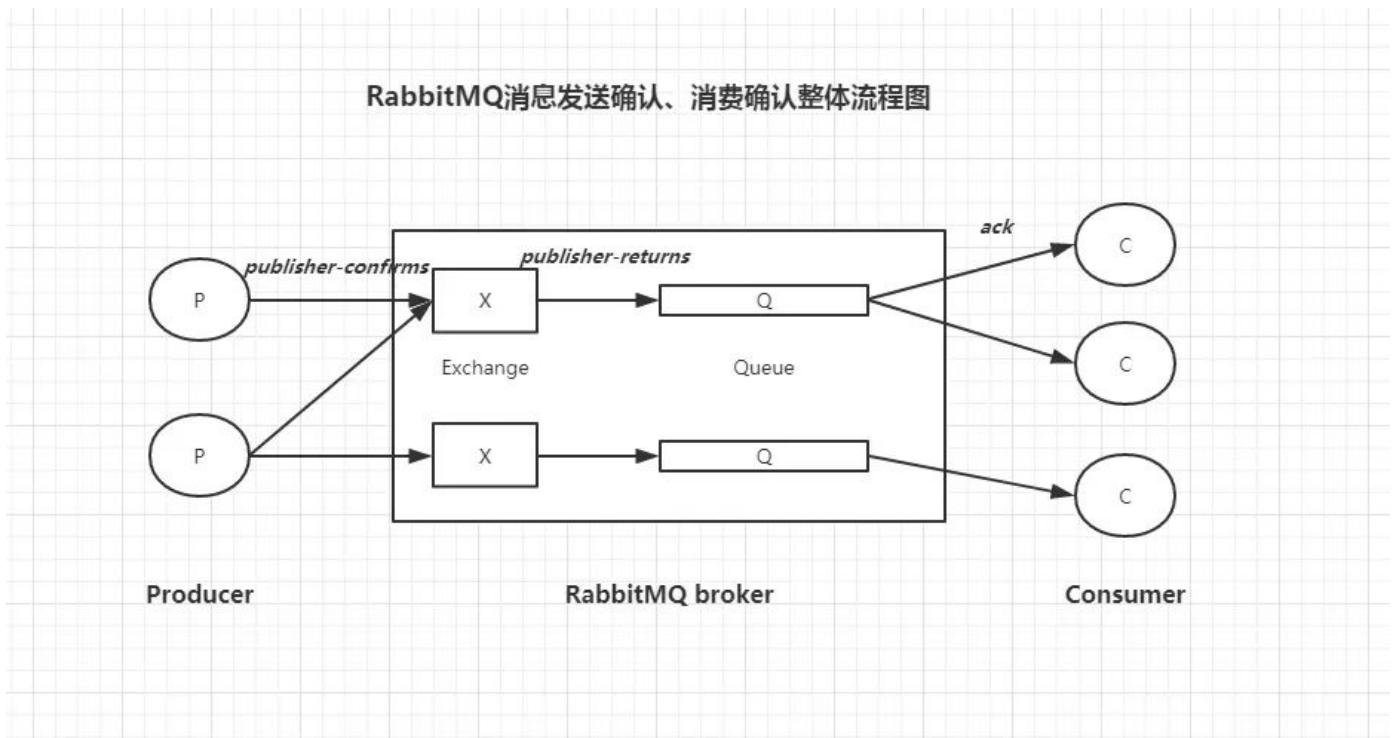
长按识别二维码，关注我的公众号

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

Spring Boot + RabbitMQ发送邮件(保证消息 100% 投递成功并被消费)

Java后端 2019-12-29

一、先扔一张图



说明:

本文涵盖了关于RabbitMQ很多方面的知识点, 如:

- 消息发送确认机制
- 消费确认机制
- 消息的重新投递
- 消费幂等性, 等等

这些都是围绕上面那张整体流程图展开的, 所以有必要先贴出来, 见图知意

二、实现思路

- 简略介绍163邮箱授权码的获取
- 编写发送邮件工具类
- 编写RabbitMQ配置文件
- 生产者发起调用
- 消费者发送邮件
- 定时任务定时拉取投递失败的消息, 重新投递
- 各种异常情况的测试验证
- 拓展: 使用动态代理实现消费端幂等性验证和消息确认(ack)

三、项目介绍

- Spring Boot版本2.1.5.RELEASE, 旧版本可能有些配置属性不能使用, 需要以代码形式进行配置
- RabbitMQ版本3.7.15
- MailUtil: 发送邮件工具类

- RabbitConfig: rabbitmq相关配置
- TestServiceImpl: 生产者, 发送消息
- MailConsumer: 消费者, 消费消息, 发送邮件
- ResendMsg: 定时任务, 重新投递发送失败的消息

说明: 上面是核心代码, MsgLogService mapper xml等均未贴出, 完整代码可以参考我的GitHub, 欢迎fork, <https://github.com/wangzaiplus/springboot/tree/wxw>

四、代码实现

163邮箱授权码的获取, 如图:



该授权码就是配置文件spring.mail.password需要的密码

2. pom

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

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

3. rabbitmq、邮箱配置

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest

spring.rabbitmq.publisher-confirms=true

spring.rabbitmq.publisher-returns=true

spring.rabbitmq.listener.simple.acknowledge-mode=manual
spring.rabbitmq.listener.simple.prefetch=100

spring.mail.host=smtp.163.com
spring.mail.username=18621142249@163.com
spring.mail.password=123456wangzai
spring.mail.from=18621142249@163.com
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true
```

说明: password即授权码, username和from要一致

4. 表结构

```
CREATE TABLE `msg_log` (
  `msg_id` varchar(255) NOT NULL DEFAULT '' COMMENT '消息唯一标识',
  `msg` text COMMENT '消息体, json格式化',
  `exchange` varchar(255) NOT NULL DEFAULT '' COMMENT '交换机',
  `routing_key` varchar(255) NOT NULL DEFAULT '' COMMENT '路由键',
  `status` int(11) NOT NULL DEFAULT '0' COMMENT '状态: 0投递中 1投递成功 2投递失败 3已消费',
  `try_count` int(11) NOT NULL DEFAULT '0' COMMENT '重试次数',
  `next_try_time` datetime DEFAULT NULL COMMENT '下一次重试时间',
  `create_time` datetime DEFAULT NULL COMMENT '创建时间',
  `update_time` datetime DEFAULT NULL COMMENT '更新时间',
  PRIMARY KEY (`msg_id`),
  UNIQUE KEY `unq_msg_id` (`msg_id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='消息投递日志';
```

说明: exchange routing_key字段是在定时任务重新投递消息时需要用到的

5. MailUtil

```
@Component
@Slf4j
public class MailUtil {

    @Value("${spring.mail.from}")
    private String from;

    @Autowired
    private JavaMailSender mailSender;

    public boolean send(Mail mail) {
        String to = mail.getTo();
        String title = mail.getTitle();
        String content = mail.getContent();

        SimpleMailMessage message = new SimpleMailMessage();
        message.setFrom(from);
        message.setTo(to);
        message.setSubject(title);
        message.setText(content);

        try {
            mailSender.send(message);
            log.info("邮件发送成功");
            return true;
        } catch (MailException e) {
            log.error("邮件发送失败, to: {}, title: {}", to, title, e);
            return false;
        }
    }
}
```

6. RabbitConfig

```
@Configuration
@Slf4j
public class RabbitConfig {

    @Autowired
    private CachingConnectionFactory connectionFactory;

    @Autowired
    private MsgLogService msgLogService;

    @Bean
    public RabbitTemplate rabbitTemplate() {
        RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(converter());

        rabbitTemplate.setConfirmCallback((correlationData, ack, cause) -> {
            if (ack) {
                log.info("消息成功发送到Exchange");
                String msgId = correlationData.getId();
                msgLogService.updateStatus(msgId, Constant.MsgLogStatus.DELIVER_SUCCESS);
            } else {
                log.info("消息发送到Exchange失败, {}, cause: {}", correlationData, cause);
            }
        });
    }

    rabbitTemplate.setMandatory(true);

    rabbitTemplate.setReturnCallback((message, replyCode, replyText, exchange, routingKey) -> {
        log.info("消息从Exchange路由到Queue失败: exchange: {}, route: {}, replyCode: {}, replyText: {}, message: {}", exchange, routingKey, replyCode, replyText, message);
    });

    return rabbitTemplate;
}

@Bean
public Jackson2JsonMessageConverter converter() {
    return new Jackson2JsonMessageConverter();
}

public static final String MAIL_QUEUE_NAME = "mail.queue";
public static final String MAIL_EXCHANGE_NAME = "mail.exchange";
public static final String MAIL_ROUTING_KEY_NAME = "mail.routing.key";

@Bean
public Queue mailQueue() {
    return new Queue(MAIL_QUEUE_NAME, true);
}

@Bean
public DirectExchange mailExchange() {
    return new DirectExchange(MAIL_EXCHANGE_NAME, true, false);
}

@Bean
public Binding mailBinding() {
    return BindingBuilder.bind(mailQueue()).to(mailExchange()).with(MAIL_ROUTING_KEY_NAME);
}

}
```

7. TestServiceImpl生产消息

```
@Service
public class TestServiceImpl implements TestService {

    @Autowired
    private MsgLogMapper msgLogMapper;

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Override
    public ServerResponse send(Mail mail) {
        String msgId = RandomUtil.UUID32();
        mail.setMsgId(msgId);

        MsgLog msgLog = new MsgLog(msgId, mail, RabbitConfig.MAIL_EXCHANGE_NAME, RabbitConfig.MAIL_ROUTING_KEY_NAME);
        msgLogMapper.insert(msgLog);

        CorrelationData correlationData = new CorrelationData(msgId);
        rabbitTemplate.convertAndSend(RabbitConfig.MAIL_EXCHANGE_NAME, RabbitConfig.MAIL_ROUTING_KEY_NAME, MessageHelper.o

        return ServerResponse.success(ResponseCode.MAIL_SEND_SUCCESS.getMsg());
    }

}
```

8. MailConsumer消费消息, 发送邮件

```
@Component
@Slf4j
public class MailConsumer {

    @Autowired
    private MsgLogService msgLogService;

    @Autowired
    private MailUtil mailUtil;

    @RabbitListener(queues = RabbitConfig.MAIL_QUEUE_NAME)
    public void consume(Message message, Channel channel) throws IOException {
        Mail mail = MessageHelper.msgToObj(message, Mail.class);
        log.info("收到消息: {}", mail.toString());

        String msgId = mail.getId();

        MsgLog msgLog = msgLogService.selectByMsgId(msgId);
        if (null == msgLog || msgLog.getStatus().equals(Constant.MsgLogStatus.CONSUMED_SUCCESS)) {
            log.info("重复消费, msgId: {}", msgId);
            return;
        }

        MessageProperties properties = message.getMessageProperties();
        long tag = properties.getDeliveryTag();

        boolean success = mailUtil.send(mail);
        if (success) {
            msgLogService.updateStatus(msgId, Constant.MsgLogStatus.CONSUMED_SUCCESS);
            channel.basicAck(tag, false);
        } else {
            channel.basicNack(tag, false, true);
        }
    }
}
```

说明: 其实就完成了3件事: 1.保证消费幂等性, 2.发送邮件, 3.更新消息状态, 手动ack

9. ResendMsg定时任务重新投递发送失败的消息

```
@Component
@Slf4j
public class ResendMsg {

    @Autowired
    private MsgLogService msgLogService;

    @Autowired
    private RabbitTemplate rabbitTemplate;

    private static final int MAX_TRY_COUNT = 3;

    @Scheduled(cron = "0/30 * * * ?")
    public void resend() {
        log.info("开始执行定时任务(重新投递消息)");

        List<MsgLog> msgLogs = msgLogService.selectTimeoutMsg();
        msgLogs.forEach(msgLog -> {
            String msgId = msgLog.getId();
            if (msgLog.getTryCount() >= MAX_TRY_COUNT) {
                msgLogService.updateStatus(msgId, Constant.MsgLogStatus.DELIVER_FAIL);
                log.info("超过最大重试次数, 消息投递失败, msgId: {}", msgId);
            } else {
                msgLogService.updateTryCount(msgId, msgLog.getNextTryTime());

                CorrelationData correlationData = new CorrelationData(msgId);
                rabbitTemplate.convertAndSend(msgLog.getExchange(), msgLog.getRoutingKey(), MessageHelper.objToMsg(msgLog.getMsg()));

                log.info("第 {} 次重新投递消息");
            }
        });

        log.info("定时任务执行结束(重新投递消息)");
    }
}
```

说明: 每一条消息都和exchange routingKey绑定, 所有消息重投共用这一个定时任务即可

五、基本测试

OK, 目前为止, 代码准备就绪, 现在进行正常流程的测试

发送请求:

搜索... 全局设置 项目设置 历史记录 | 编辑项目 主页

个人	基本信息
▶ 测试	接口名称: sendMail 请求方法: POST 请求地址: http://localhost:11111/test/send 数据类型: X-WWW-FORM-URLENCODED 响应类型: JSON
▶ jwt	
▶ 分布式锁	
▼ springboot测试项目	
getAll	
getOne	
add	
update	
delete	
getByUsername	
jedis	
getToken	
testIdempotence	
testJedisUtil	
testAccessLimit	
testRedisson	
login	
sendMail	演示 请求地址: http://localhost:11111/test/send 请求参数: to: 18621142249@163.com title: 标题 content: 正文 代理请求 直接运行
	Body Headers { "status": 0, "msg": "邮件发送成功", "data": null }

后台日志:

```
springboot [F:\projects\springboot - ...src\main\java\com\wangzaiplus\test\service] mfh\TestServiceImpl.java [Test] - IntelliJ IDEA
File Edit View Navigate Code Analyze Build Run Tools VCS Window Help
Project TestApplication TestController TestService TestServiceImpl TestServiceImpl.java SimpleMailConsumer MailConsumer MailListener BaseConsumer MailMapper MailInteractor MailLogService MailLogServiceImpl BeanSendMapper MailUtil MailUtilImpl RabbitConfig JavaMailSender
RabbitConfig.java
String msgId = RandomUtil.randomUUID();
mail.setMsgId(msgId);

MsgLog msgLog = new MsgLog(msgId, mail, RabbitConfig.MAIL_EXCHANGE_NAME, RabbitConfig.MAIL_ROUTING_KEY_NAME);
msgLogMapper.insert(msgLog); // 消息入库

CorrelationData correlationData = new CorrelationData(msgId);
rabbitTemplate.convertAndSend(RabbitConfig.MAIL_EXCHANGE_NAME, RabbitConfig.MAIL_ROUTING_KEY_NAME, MessageHelper.objToMsg(mail), correlationData); // 发送消息

return ServerResponse.success(ResponseCode.MAIL_SEND_SUCCESS.getMsg());
TestServiceImpl.java
send()

2019-07-18 18:22:13.335 INFO 6348 --- [io-11111-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2019-07-18 18:22:13.335 INFO 6348 --- [io-11111-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2019-07-18 18:22:13.340 INFO 6348 --- [io-11111-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 5 ms
2019-07-18 18:22:13.472 INFO 6348 --- [io-11111-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2019-07-18 18:22:13.572 INFO 6348 --- [io-11111-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2019-07-18 18:22:13.577 DEBUG 6348 --- [io-11111-exec-1] c.w.t.mapper.MsgLogMapper.insert : ==> Preparing: INSERT INTO msg_log(msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, update_time) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
2019-07-18 18:22:13.598 DEBUG 6348 --- [io-11111-exec-1] c.w.t.mapper.MsgLogMapper.insert : ==> Parameters: 746ada2ece324d86952b1418b8d5ced2(String), {"to": "18621142249@163.com", "title": "标题"}, ?, ?, ?, ?, ?, ?, ?, ?
2019-07-18 18:22:13.603 DEBUG 6348 --- [io-11111-exec-1] c.w.t.mapper.MsgLogMapper.insert : <== Updates: 1
2019-07-18 18:22:13.622 INFO 6348 --- [127.0.0.1:5672] c.wangzaiplus.test.config.RabbitConfig : 消息成功发送到Exchange
2019-07-18 18:22:13.622 DEBUG 6348 --- [127.0.0.1:5672] c.w.t.mapper.MsgLogMapper.updateStatus : ==> Preparing: update msg_log set status = ?, update_time = now() where msg_id = ?
2019-07-18 18:22:13.623 DEBUG 6348 --- [127.0.0.1:5672] c.w.t.mapper.MsgLogMapper.updateStatus : ==> Parameters: 1(Integer), 746ada2ece324d86952b1418b8d5ced2(String)
2019-07-18 18:22:13.627 DEBUG 6348 --- [127.0.0.1:5672] c.w.t.mapper.MsgLogMapper.updateStatus : <== Updates: 1
2019-07-18 18:22:13.639 INFO 6348 --- [ntContainer@0-1] c.w.test.mq.consumer.SimpleMailConsumer : 收到消息: com.wangzaiplus.test.pojo.Mail@31d0a5fe
2019-07-18 18:22:13.641 DEBUG 6348 --- [ntContainer@0-1] c.w.t.m.MsgLogMapper.selectByPrimarykey : ==> Preparing: select msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, update_time from msg_log where msg_id = ?
2019-07-18 18:22:13.641 DEBUG 6348 --- [ntContainer@0-1] c.w.t.m.MsgLogMapper.selectByPrimarykey : ==> Parameters: 746ada2ece324d86952b1418b8d5ced2(String)
2019-07-18 18:22:13.666 DEBUG 6348 --- [ntContainer@0-1] c.w.t.m.MsgLogMapper.selectByPrimarykey : <== Total: 1
2019-07-18 18:22:14.367 INFO 6348 --- [ntContainer@0-1] com.wangzaiplus.test.util.MailUtil : 邮件发送成功
2019-07-18 18:22:14.367 DEBUG 6348 --- [ntContainer@0-1] c.w.t.mapper.MsgLogMapper.updateStatus : ==> Preparing: update msg_log set status = ?, update_time = now() where msg_id = ?
2019-07-18 18:22:14.368 DEBUG 6348 --- [ntContainer@0-1] c.w.t.mapper.MsgLogMapper.updateStatus : ==> Parameters: 3(Integer), 746ada2ece324d86952b1418b8d5ced2(String)
2019-07-18 18:22:14.373 DEBUG 6348 --- [ntContainer@0-1] c.w.t.mapper.MsgLogMapper.updateStatus : <== Updates: 1
```

数据库消息记录:

视图	帮助	函数	事件	查询	报表	计划	模型
对象 msg_log @test (localhost) - ...							
开始任务	备注	筛选	排序	导入	导出		
msg_id	msg	exchange	routing_key	status	try_count	next_try_time	create_time
746ada2ece324d86952b1418b1d5ced2	{"to": "18621142249@163.com", "title": "标题", "content": "正文", "msgId": "mail.exchange"}	mail.routing.key	3	0	2019-07-18 18:23:13	2019-07-18 18:22:13	2019-07-18 18:22:14

状态为3, 表明已消费, 消息重试次数为0, 表明一次投递就成功了

查看邮箱

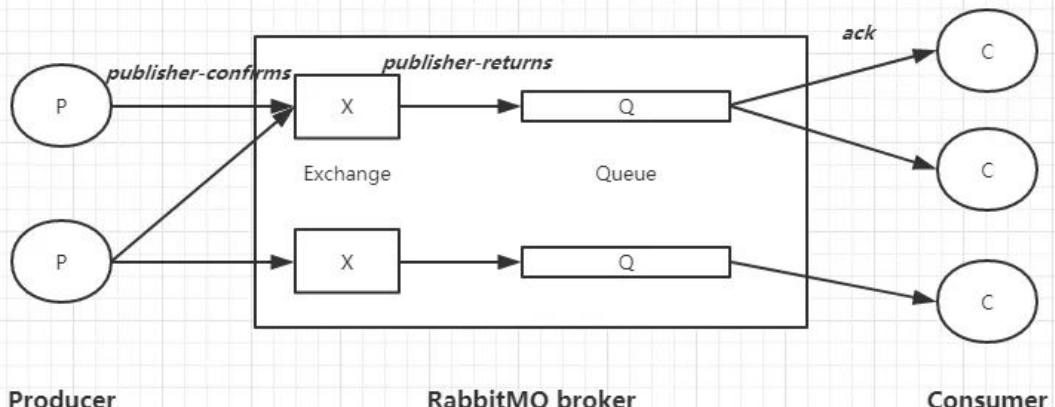
The screenshot shows a 163 webmail inbox. On the left sidebar, there are categories like '收件箱', '待办邮件', '智能标签', '星标联系人邮件', '草稿箱', '已发送', '订阅邮件 (3)', '其他3个文件夹', '邮件标签', '邮箱中心', and '文件中心'. The main area displays an incoming email from '我 <18621142249@163.com>' with the subject '标题'. The email body contains promotional text about upgrading to VIP email. The '正文' (Body) section is highlighted with a red box.

发送成功

六、各种异常情况测试

步骤一罗列了很多关于RabbitMQ的知识点, 很重要, 很核心, 而本文也涉及到了这些知识点的实现, 接下来就通过异常测试进行验证(这些验证都是围绕本文开头扔的那张流程图展开的, 很重要, 所以, 再贴一遍)

RabbitMQ消息发送确认、消费确认整体流程图



验证消息发送到Exchange失败情况下的回调, 对应上图P->X

如何验证? 可以随便指定一个不存在的交换机名称, 请求接口, 看是否会触发回调

```
public ServerResponse accessLimit() {
    return ServerResponse.success("accessLimit: success");
}

@Override
public ServerResponse send(Mail mail) {
    String msgId = RandomUtil.randomUUID();
    mail.setMsgId(msgId);

    MsgLog msgLog = new MsgLog(msgId, mail, RabbitConfig.MAIL_EXCHANGE_NAME, RabbitConfig.MAIL_ROUTING_KEY_NAME);
    msgLogMapper.insert(msgLog); // 消息入库

    CorrelationData correlationData = new CorrelationData(msgId);
    rabbitTemplate.convertAndSend(exchange, RabbitConfig.MAIL_EXCHANGE_NAME + "abcd", RabbitConfig.MAIL_ROUTING_KEY_NAME, MessageHelper.objToMsg(mail), correlationData);
    return ServerResponse.success(ResponseCode.MAIL_SEND_SUCCESS.getMsg());
}
```

```
INFO 10868 --- [io-11111-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
INFO 10868 --- [io-11111-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
INFO 10868 --- [io-11111-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 6 ms
INFO 10868 --- [io-11111-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
INFO 10868 --- [io-11111-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
DEBUG 10868 --- [io-11111-exec-1] c.w.t.mapper.MsgLogMapper.insert : ==> Preparing: INSERT INTO msg_log(msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, updated_time) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
DEBUG 10868 --- [io-11111-exec-1] c.w.t.mapper.MsgLogMapper.insert : ==> Parameters: bb97elcba9064d188f2701ff85e1ecd(String), {"to":"18621142249@163.com","title":"标题","content":"正文","msgId":String}
DEBUG 10868 --- [io-11111-exec-1] c.w.t.mapper.MsgLogMapper.insert : <== Updates: 1
INFO 10868 --- [ 127.0.0.1:5672] o.s.a.r.c.CachingConnectionFactory : Channel shutdown: channel error; protocol method: #method#channel.close(reply-code=404, reply-text=NOT_FOUND - no exchange 'mail.exchangeabcd' in vhost '/', cause: channel error; protocol method: #method#
: 消息发送到Exchange失败, CorrelationData [id=bb97elcba9064d188f2701ff85e1ecd], cause: channel error; protocol method: #method#
```

发送失败, 原因:reply-code=404, reply-text=NOT_FOUND - no exchange 'mail.exchangeabcd' in vhost '/', 该回调能够保证消息正确发送到Exchange, 测试完成

验证消息从Exchange路由到Queue失败情况下的回调, 对应上图X->Q

同理, 修改一下路由键为不存在的即可, 路由失败, 触发回调

```
public ServerResponse accessLimit() {
    return ServerResponse.success("accessLimit: success");
}

@Override
public ServerResponse send(Mail mail) {
    String msgId = RandomUtil.randomUUID();
    mail.setMsgId(msgId);

    MsgLog msgLog = new MsgLog(msgId, mail, RabbitConfig.MAIL_EXCHANGE_NAME, RabbitConfig.MAIL_ROUTING_KEY_NAME);
    msgLogMapper.insert(msgLog); // 消息入库

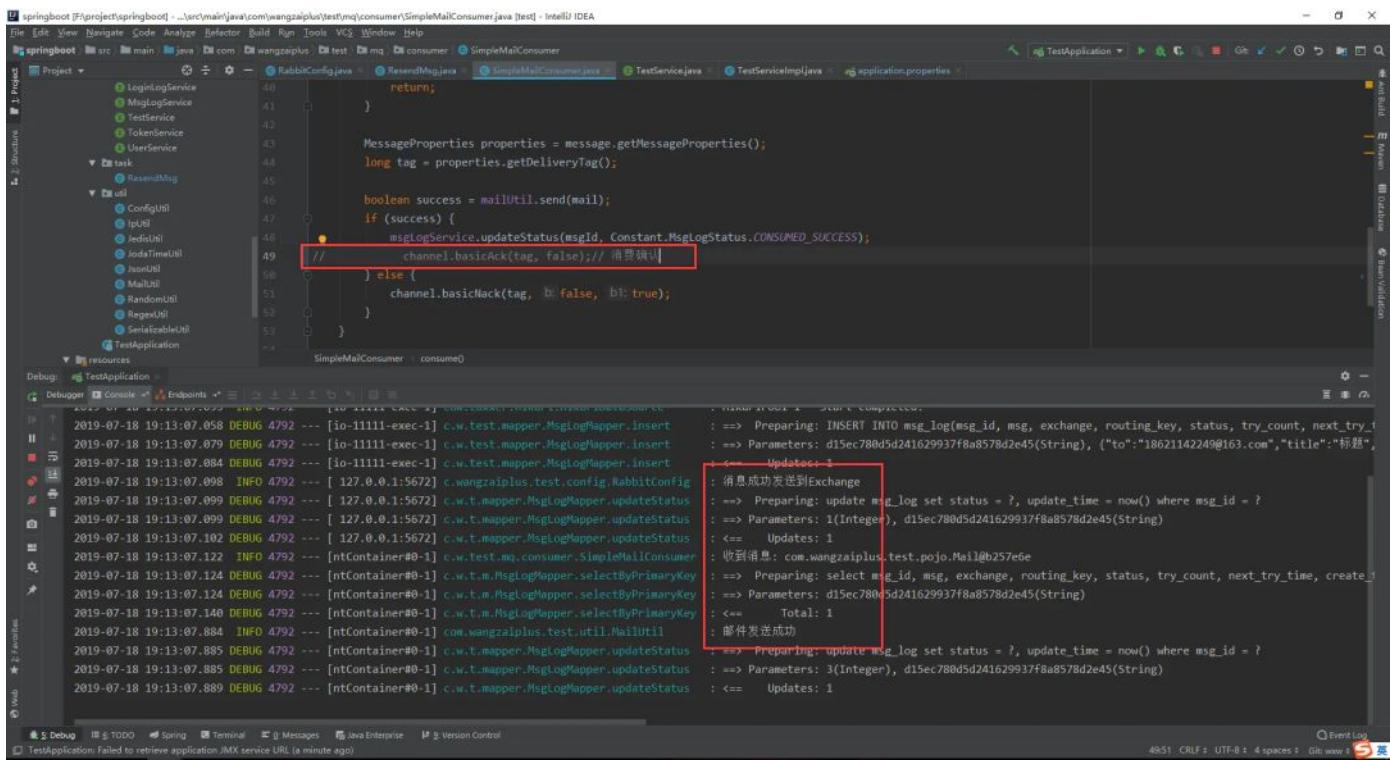
    CorrelationData correlationData = new CorrelationData(msgId);
    rabbitTemplate.convertAndSend(RabbitConfig.MAIL_EXCHANGE_NAME, routingKey, RabbitConfig.MAIL_ROUTING_KEY_NAME + "abcd", MessageHelper.objToMsg(mail), correlationData);
    return ServerResponse.success(ResponseCode.MAIL_SEND_SUCCESS.getMsg());
}
```

```
2019-07-18 19:04:15.966 INFO 9424 --- [io-11111-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2019-07-18 19:04:15.972 INFO 9424 --- [io-11111-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 6 ms
2019-07-18 19:04:16.110 INFO 9424 --- [io-11111-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2019-07-18 19:04:16.206 INFO 9424 --- [io-11111-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2019-07-18 19:04:16.211 DEBUG 9424 --- [io-11111-exec-1] c.w.t.mapper.MsgLogMapper.insert : ==> Preparing: INSERT INTO msg_log(msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, updated_time) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
2019-07-18 19:04:16.231 DEBUG 9424 --- [io-11111-exec-1] c.w.t.mapper.MsgLogMapper.insert : ==> Parameters: 40e5540700394100a68189130a53d747(String), {"to":"18621142249@163.com","title":"标题","content":"正文","msgId":String}
2019-07-18 19:04:16.247 INFO 9424 --- [ 127.0.0.1:5672] c.wangzaiplus.test.config.RabbitConfig : 消息从Exchange路由到Queue失败: exchange: mail.exchange, route: mail.routing.keyabcd, replyCode: 312, replyText: NO_ROUTE
2019-07-18 19:04:16.248 INFO 9424 --- [ 127.0.0.1:5672] c.wangzaiplus.test.config.RabbitConfig : 消息成功发送到Exchange
2019-07-18 19:04:16.248 DEBUG 9424 --- [ 127.0.0.1:5672] c.w.t.mapper.MsgLogMapper.updateStatus : ==> Preparing: update msg_log set status = ?, update_time = now() where msg_id = ?
2019-07-18 19:04:16.248 DEBUG 9424 --- [ 127.0.0.1:5672] c.w.t.mapper.MsgLogMapper.updateStatus : ==> Parameters: 1(Integer), 40e5540700394100a68189130a53d747(String)
2019-07-18 19:04:16.253 DEBUG 9424 --- [ 127.0.0.1:5672] c.w.t.mapper.MsgLogMapper.updateStatus : <== Updates: 1
```

发送失败, 原因:route: mail.routing.keyabcd, replyCode: 312, replyText: NO_ROUTE

验证在手动ack模式下, 消费端必须进行手动确认(ack), 否则消息会一直保存在队列中, 直到被消费, 对应上图Q->C

将消费端代码channel.basicAck(tag, false); // 消费确认注释掉, 查看控制台和rabbitmq管控台

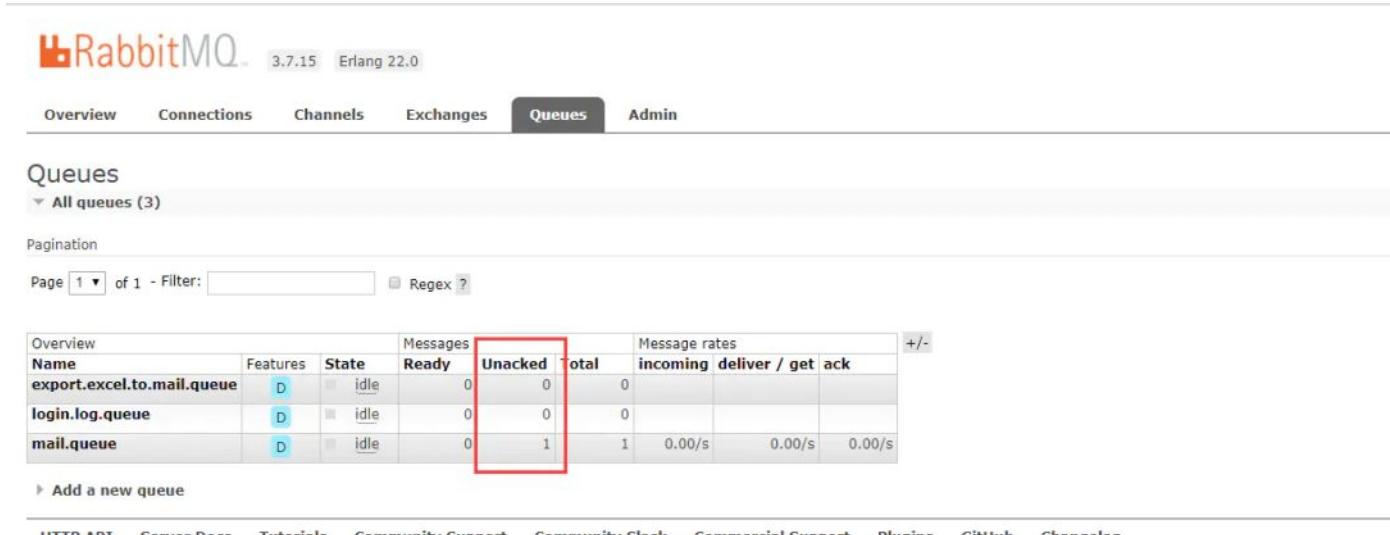


```
    return;
}
MessageProperties properties = message.getMessageProperties();
long tag = properties.getDeliveryTag();
boolean success = mailUtil.send(mail);
if (success) {
    msgLogService.updateStatus(msgId, Constant.MsgLogStatus.CONSUMED_SUCCESS);
} else {
    channel.basicNack(tag, b: false, b1: true);
}
}

SimpleMailConsumer -> consume()
```

IDEA Debug Console output:

```
2019-07-18 19:13:07.058 DEBUG 4792 --- [io-11111-exec-1] c.w.test.mapper.MsgLogMapper.insert :==> Preparing: INSERT INTO msg_log(msg_id, msg, exchange, routing_key, status, try_count, next_try_time) VALUES(?, ?, ?, ?, ?, ?, ?)
: ==> Parameters: d15ec780d5d241629937f8a8578d2e45(String), ("to":"18621142249@163.com","title":"标题"), 1, 1, null, null, null
: <== Updates: 1
: 消息成功发送到Exchange
: ==> Preparing: update msg_log set status = ?, update_time = now() where msg_id = ?
: ==> Parameters: 1(Integer), d15ec780d5d241629937f8a8578d2e45(String)
: <== Updates: 1
: 收到消息: com.wangzaipius.test.pojo.Mail@b257e6e
: ==> Preparing: select msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, update_time from msg_log where msg_id = ?
: ==> Parameters: d15ec780d5d241629937f8a8578d2e45(String)
: <== Total: 1
: 邮件发送成功
: ==> Preparing: update msg_log set status = ?, update_time = now() where msg_id = ?
: ==> Parameters: 3(Integer), d15ec780d5d241629937f8a8578d2e45(String)
: <== Updates: 1
```



RabbitMQ 3.7.15 Erlang 22.0

Overview Connections Channels Exchanges Queues Admin

Queues

All queues (3)

Pagination

Page 1 of 1 - Filter: Regex ?

Name	Features	State	Ready	Unacked	Total	Message rates
export.excel.to.mail.queue	D	idle	0	0	0	+/-
login.log.queue	D	idle	0	0	0	
mail.queue	D	idle	0	1	1	0.00/s 0.00/s 0.00/s

Add a new queue

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

可以看到, 虽然消息确实被消费了, 但是由于是手动确认模式, 而最后又没手动确认, 所以, 消息仍被rabbitmq保存, 所以, 手动ack能保证消息一定被消费, 但一定要记得basicAck

验证消费端幂等性

接着上一步, 去掉注释, 重启服务器, 由于有一条未被ack的消息, 所以重启后监听到消息, 进行消费, 但是由于消费前会判断该消息的状态是否未被消费, 发现status=3, 即已消费, 所以, 直接return, 这样就保证了消费端的幂等性, 即使由于网络等原因投递成功而未触发回调, 从而多次投递, 也不会重复消费进而发生业务异常

```

    @Autowired
    private MailUtil mailUtil;

    @RabbitListener(queues = RabbitConfig.MAIL_QUEUE_NAME)
    public void consume(Message message, Channel channel) throws IOException {
        Mail mail = MessageHelper.msgToObj(message, Mail.class);
        Log.info("收到消息: {}", mail.toString());

        String msgId = mail.getId();

        MsgLog msgLog = msgLogService.selectByMsgId(msgId);
        if (null == msgLog || msgLog.getStatus().equals(Constant.MsgLogStatus.CONSUMED_SUCCESS)) { // 消费幂等性
            Log.info("重复消费, msgId: {}", msgId);
            return;
        }

        MessageProperties properties = message.getMessageProperties();
    }

```

Execution log (部分输出):

```

2019-07-18 19:19:44.353 INFO 10132 --- [main] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
2019-07-18 19:19:44.386 INFO 10132 --- [main] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory@544e8149:0/SimpleConnection@7e0f9528 [delegate=amqp]
2019-07-18 19:19:44.473 INFO 10132 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 11111 (http) with context path ''
2019-07-18 19:19:44.476 INFO 10132 --- [main] com.wangzaiplus.test.TestApplication : Started TestApplication in 2.737 seconds (JVM running for 3.318)
2019-07-18 19:19:44.488 INFO 10132 --- [ntContainer#0-1] c.w.t.m.consumer.SimpleMailConsumer : 收到消息: com.wangzaiplus.test.pojo.Mail@f3ba3d
2019-07-18 19:19:44.492 INFO 10132 --- [ntContainer#0-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2019-07-18 19:19:44.586 INFO 10132 --- [ntContainer#0-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2019-07-18 19:19:44.592 DEBUG 10132 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Preparing: select msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, update_time from msg_log where msg_id = ?
2019-07-18 19:19:44.618 DEBUG 10132 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Parameters: d15ec780d5d241629937f8a8578d2e45(String)
2019-07-18 19:19:44.628 DEBUG 10132 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : << Total: 1
2019-07-18 19:19:44.638 INFO 10132 --- [ntContainer#0-1] c.w.t.m.consumer.SimpleMailConsumer : 重复消费, msgId: d15ec780d5d241629937f8a8578d2e45

```

验证消费端发生异常消息也不会丢失

很显然, 消费端代码可能发生异常, 如果不做处理, 业务没正确执行, 消息却不见了, 给我们感觉就是消息丢失了, 由于我们消费端代码做了异常捕获, 业务异常时, 会触发: `channel.basicNack(tag, false, true);`, 这样会告诉rabbitmq该消息消费失败, 需要重新入队, 可以重新投递到其他正常的消费端进行消费, 从而保证消息不被丢失

测试: `send`方法直接返回`false`即可(这里跟抛出异常一个意思)

```

private JavaMailSender mailSender;

/**
 * 发送简单邮件
 */
public boolean send(Mail mail) {
    if (true) return false;

    String to = mail.getTo(); // 目标邮箱
    String title = mail.getTitle(); // 邮件标题
}

```

Execution log (部分输出):

```

2019-07-18 19:33:07.303 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Preparing: select msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, update_time from msg_log where msg_id = ?
2019-07-18 19:35:07.303 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Parameters: 21a696312894428fa33a2a20676e3ffa(String)
2019-07-18 19:35:07.303 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : << Total: 1
2019-07-18 19:35:07.304 INFO 4732 --- [ntContainer#0-1] c.w.t.m.consumer.SimpleMailConsumer : 收到消息: com.wangzaiplus.test.pojo.Mail@2d8b1d6a
2019-07-18 19:35:07.304 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Preparing: select msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, update_time from msg_log where msg_id = ?
2019-07-18 19:35:07.304 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Parameters: 21a696312894428fa33a2a20676e3ffa(String)
2019-07-18 19:35:07.304 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : << Total: 1
2019-07-18 19:35:07.304 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : 收到消息: com.wangzaiplus.test.pojo.Mail@57c4bd1
2019-07-18 19:35:07.304 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Preparing: select msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, update_time from msg_log where msg_id = ?
2019-07-18 19:35:07.304 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Parameters: 21a696312894428fa33a2a20676e3ffa(String)
2019-07-18 19:35:07.304 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : << Total: 1
2019-07-18 19:35:07.305 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : 收到消息: com.wangzaiplus.test.pojo.Mail@89ab03c
2019-07-18 19:35:07.305 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.consumer.SimpleMailConsumer : >>> Preparing: select msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, update_time from msg_log where msg_id = ?
2019-07-18 19:35:07.305 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Parameters: 21a696312894428fa33a2a20676e3ffa(String)
2019-07-18 19:35:07.305 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : << Total: 1
2019-07-18 19:35:07.305 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.consumer.SimpleMailConsumer : 收到消息: com.wangzaiplus.test.pojo.Mail@79e656cc
2019-07-18 19:35:07.305 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Preparing: select msg_id, msg, exchange, routing_key, status, try_count, next_try_time, create_time, update_time from msg_log where msg_id = ?
2019-07-18 19:35:07.305 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : >>> Parameters: 21a696312894428fa33a2a20676e3ffa(String)
2019-07-18 19:35:07.305 DEBUG 4732 --- [ntContainer#0-1] c.w.t.m.MsgLogMapper.selectByPrimaryKey : << Total: 1

```

可以看到, 由于`channel.basicNack(tag, false, true)`, 未被ack的消息(unacked)会重新入队并被消费, 这样就保证了消息不会丢失

验证定时任务的消息重投

实际应用场景中, 可能由于网络原因, 或者消息未被持久化MQ就宕机了, 使得投递确认的回调方法`ConfirmCallback`没有被执行, 从而导致数据库该消息状态一直是投递中的状态, 此时就需要进行消息重投, 即使也许消息已经被消费了

定时任务只是保证消息100%投递成功, 而多次投递的消费幂等性需要消费端自己保证

我们可以将回调和消费成功后更新消息状态的代码注释掉, 开启定时任务, 查看是否重投

The screenshot shows the IntelliJ IDEA interface with the code editor open to a file named `RabbitConfig.java`. The code contains logic for handling message delivery attempts. A red box highlights a comment in the code: `// 消息是否成功发送到Exchange`. Below the code editor is a terminal window showing application logs. The logs show multiple DEBUG-level entries for the year 2019-07-19 at 10:20:00.000. One entry indicates a failed delivery attempt: `INFO 4076 --- [scheduling-1] com.wangzaiplus.test.task.ResendMsg : 超过最大重试次数, 消息投递失败, msgId: 8cede5aae3d31452eb55413eba32427c2`. Another entry shows a successful delivery attempt: `INFO 4076 --- [scheduling-1] com.wangzaiplus.test.task.ResendMsg : 定时任务执行结束(重新投递消息)`.

The screenshot shows the Oracle Database SQL Developer interface with a table named `msg_log` selected. The table has columns: `msg_id`, `msg`, `exchange`, `routing_key`, `status`, `try_count`, `next_try_time`, `create_time`, and `update_time`. A single row is visible in the results grid, with the `try_count` column highlighted by a red box. The value in this column is 2.

对象	msg_log @test (localhost) - ...							
开始事务	回滚	筛选	排序	导入	导出			
msg_id	msg	exchange	routing_key	status	try_count	next_try_time	create_time	update_time
8cede5aae3d31452eb55413eba32427c2	{"to":"18621142249@163.com","title":"标题","content":"正文","msgId":mail.exchange}	mail.routing.key		2	3	2019-07-19 10:20:38	2019-07-19 10:16:38	2019-07-19 10:21:00

可以看到, 消息会重投3次, 超过3次放弃, 将消息状态置为投递失败状态, 出现这种非正常情况, 就需要人工介入排查原因

七、拓展: 使用动态代理实现消费端幂等性验证和消费确认(ack)

不知道大家发现没有, 在MailConsumer中, 真正的业务逻辑其实只是发送邮件`mailUtil.send(mail)`而已, 但我们又不得不在调用`send`方法之前校验消费幂等性, 发送后, 还要更新消息状态为"已消费"状态, 并手动`ack`, 实际项目中, 可能还有很多生产者-消费者的应用场景, 如记录日志, 发送短信等等, 都需要rabbitmq, 如果每次都写这些重复的公用代码, 没必要, 也难以维护, 所以, 我们可以将公共代码抽离出来, 让核心业务逻辑只关心自己的实现, 而不用做其他操作, 其实就是AOP

为达到这个目的, 有很多方法, 可以用spring aop, 可以用拦截器, 可以用静态代理, 也可以用动态代理, 在这里, 我用的是动态代理

目录结构如下:

```
public Object getProxy() {
    ClassLoader classLoader = target.getClass().getClassLoader();
    Class[] interfaces = target.getClass().getInterfaces();

    Object proxy = Proxy.newProxyInstance(classLoader, interfaces, (proxy1, method, args) -> {
        Message message = (Message) args[0];
        Channel channel = (Channel) args[1];

        String correlationId = getCorrelationId(message);

        if (!isConsumed(correlationId)) { // 消费幂等性，防止消息被重复消费
            log.info("重复消费, correlationId: {}", correlationId);
            return null;
        }

        MessageProperties properties = message.getMessageProperties();
        long tag = properties.getDeliveryTag();

        try {
            Object result = method.invoke(target, args); // 真正消费的业务逻辑
            msgLogService.updateStatus(correlationId, Constant.MsgLogStatus.CONSUMED_SUCCESS);
            channel.basicAck(tag, b: false); // 消费确认
            return result;
        } catch (Exception e) {
            log.error("getProxy error", e);
            channel.basicNack(tag, b: false, b1: true);
            return null;
        }
    });
}
```

核心代码就是代理的实现, 这里就不把所有代码贴出来了, 只是提供一个思路, 我们要尽可能地把代码写的更简洁更优雅

八、总结

发送邮件其实很简单, 但深究起来其实有很多需要注意和完善的地方, 一个看似很小的知识点, 也可以引申出很多问题, 甚至涉及到方方面面, 这些都需要自己踩坑, 当然我这代码肯定还有很多不完善和需要优化的点, 希望小伙伴多多提意见和建议

我的代码都是经过自测验证过的, 图也都是一点一点自己画的或认真截的, 希望小伙伴能学到一点东西, 路过的点个赞或点个关注呗, 谢谢

项目**Github**, 欢迎fork

<https://github.com/wangzaiplus/springboot/tree/wxw>

- END -

推荐阅读

1. Java 线程有哪些不太为人所知的技巧与用法?
2. 关于 CPU 的一些基本知识总结
3. 发布没有答案的面试题, 都是耍流氓
4. 什么是一致性 Hash 算法?
5. 团队开发中 Git 最佳实践



喜欢文章, 点个在看 

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

Spring Boot + Redis + 注解 + 拦截器来实现接口幂等性校验

Java后端 1月6日

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | wangzaiplus

链接 | www.jianshu.com/p/6189275403ed

一、概念

幂等性, 通俗的说就是一个接口, 多次发起同一个请求, 必须保证操作只能执行一次

比如:

- 订单接口, 不能多次创建订单
 - 支付接口, 重复支付同一笔订单只能扣一次钱
 - 支付宝回调接口, 可能会多次回调, 必须处理重复回调
 - 普通表单提交接口, 因为网络超时等原因多次点击提交, 只能成功一次
- 等等

二、常见解决方案

- 唯一索引 -- 防止新增脏数据
- token机制 -- 防止页面重复提交
- 悲观锁 -- 获取数据的时候加锁(锁表或锁行)
- 乐观锁 -- 基于版本号version实现, 在更新数据那一刻校验数据
- 分布式锁 -- redis(jedis、redisson)或zookeeper实现
- 状态机 -- 状态变更, 更新数据时判断状态

三、本文实现

本文采用第2种方式实现, 即通过redis + token机制实现接口幂等性校验

四、实现思路

为需要保证幂等性的每一次请求创建一个唯一标识token, 先获取token, 并将此token存入redis, 请求接口时, 将此token放到header或者作为请求参数请求接口, 后端接口判断redis中是否存在此token:

- 如果存在, 正常处理业务逻辑, 并从redis中删除此token, 那么, 如果是重复请求, 由于token已被删除, 则不能通过校验, 返回
请勿重复操作提示
- 如果不存在, 说明参数不合法或者是重复请求, 返回提示即可

五、项目简介

- springboot
- redis

- @Apildempotent注解 + 拦截器对请求进行拦截
- @ControllerAdvice全局异常处理
- 压测工具: jmeter

说明:

本文重点介绍幂等性核心实现, 关于springboot如何集成redis、ServerResponse、ResponseCode等细枝末节不在本文讨论范围之内, 有兴趣的小伙伴可以查看我的Github项目:

```
https://github.com/wangzaiplus/springboot/tree/wxw
```

六、代码实现

pom

```
<!-- Redis-Jedis -->
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>2.9.0</version>
</dependency>

<!--lombok 本文用到@Slf4j注解, 也可不引用, 自定义log即可-->
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.16.10</version>
</dependency>
```

JedisUtil

```
@Component
@Slf4j
public class JedisUtil {

    @Autowired
    private JedisPool jedisPool;

    private Jedis getJedis() {
        return jedisPool.getResource();
    }

    /**
     * 设值
     *
     * @param key
     * @param value
     * @return
     */
    public String set(String key, String value) {
        Jedis jedis = null;
        try {
            jedis = getJedis();
            return jedis.set(key, value);
        } catch (Exception e) {
            log.error("set key:{} value:{} error", key, value, e);
            return null;
        } finally {
            close(jedis);
        }
    }
}
```

```
}

/**
 * 设置
 *
 * @param key
 * @param value
 * @param expireTime 过期时间, 单位:s
 * @return
 */
public String set(String key, String value, int expireTime) {
    Jedis jedis = null;
    try {
        jedis = getJedis();
        return jedis.setex(key, expireTime, value);
    } catch (Exception e) {
        log.error("set key:{} value:{} expireTime:{} error", key, value, expireTime, e);
        return null;
    } finally {
        close(jedis);
    }
}

/**
 * 取值
 *
 * @param key
 * @return
 */
public String get(String key) {
    Jedis jedis = null;
    try {
        jedis = getJedis();
        return jedis.get(key);
    } catch (Exception e) {
        log.error("get key:{} error", key, e);
        return null;
    } finally {
        close(jedis);
    }
}

/**
 * 删除key
 *
 * @param key
 * @return
 */
public Long del(String key) {
    Jedis jedis = null;
    try {
        jedis = getJedis();
        return jedis.del(key.getBytes());
    } catch (Exception e) {
        log.error("del key:{} error", key, e);
        return null;
    } finally {
        close(jedis);
    }
}

/**
 * 判断key是否存在
 */

```

```
*  
* @param key  
* @return  
*/  
public Boolean exists(String key) {  
    jedis = null;  
    try {  
        jedis = getJedis();  
        return jedis.exists(key.getBytes());  
    } catch (Exception e) {  
        log.error("exists key:{} error", key, e);  
        return null;  
    } finally {  
        close(jedis);  
    }  
}  
  
/**  
 * 设值key过期时间  
 *  
 * @param key  
 * @param expireTime 过期时间, 单位: s  
 * @return  
 */  
public Long expire(String key, int expireTime) {  
    jedis = null;  
    try {  
        jedis = getJedis();  
        return jedis.expire(key.getBytes(), expireTime);  
    } catch (Exception e) {  
        log.error("expire key:{} error", key, e);  
        return null;  
    } finally {  
        close(jedis);  
    }  
}  
  
/**  
 * 获取剩余时间  
 *  
 * @param key  
 * @return  
 */  
public Long ttl(String key) {  
    jedis = null;  
    try {  
        jedis = getJedis();  
        return jedis.ttl(key);  
    } catch (Exception e) {  
        log.error("ttl key:{} error", key, e);  
        return null;  
    } finally {  
        close(jedis);  
    }  
}  
  
private void close(Jedis jedis) {  
    if (null != jedis) {  
        jedis.close();  
    }  
}  
}
```

```
/**  
 * 在需要保证接口幂等性的Controller的方法上使用此注解  
 */  
@Target{ElementType.METHOD}  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Apildempotent {  
}
```

ApildempotentInterceptor拦截器

```
/**  
 * 接口幂等性拦截器  
 */  
public class ApildempotentInterceptor implements HandlerInterceptor {  
  
    @Autowired  
    private TokenService tokenService;  
  
    @Override  
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {  
        if (!(handler instanceof HandlerMethod)) {  
            return true;  
        }  
  
        HandlerMethod handlerMethod = (HandlerMethod) handler;  
        Method method = handlerMethod.getMethod();  
  
        Apildempotent methodAnnotation = method.getAnnotation(Apildempotent.class);  
        if (methodAnnotation != null) {  
            check(request); // 幂等性校验，校验通过则放行，校验失败则抛出异常，并通过统一异常处理返回友好提示  
        }  
  
        return true;  
    }  
  
    private void check(HttpServletRequest request) {  
        tokenService.checkToken(request);  
    }  
  
    @Override  
    public void postHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, ModelAndView  
    )  
    }  
    @Override  
    public void afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception  
    )  
    }  
}
```

TokenServiceImpl

```
@Service
public class TokenServiceImpl implements TokenService {

    private static final String TOKEN_NAME = "token";

    @Autowired
    private JedisUtil jedisUtil;

    @Override
    public ServerResponse createToken() {
        String str = RandomUtil.UUID32();
        StrBuilder token = new StrBuilder();
        token.append(Constant.Redis.TOKEN_PREFIX).append(str);

        jedisUtil.set(token.toString(), token.toString(), Constant.Redis.EXPIRE_TIME_MINUTE);

        return ServerResponse.success(token.toString());
    }

    @Override
    public void checkToken(HttpServletRequest request) {
        String token = request.getHeader(TOKEN_NAME);
        if (StringUtils.isBlank(token)) { // header中不存在token
            token = request.getParameter(TOKEN_NAME);
            if (StringUtils.isBlank(token)) { // parameter中也不存在token
                throw new ServiceException(ResponseCode.ILLEGAL_ARGUMENT.getMsg());
            }
        }

        if (!jedisUtil.exists(token)) {
            throw new ServiceException(ResponseCode.REPETITIVE_OPERATION.getMsg());
        }

        Long del = jedisUtil.del(token);
        if (del <= 0) {
            throw new ServiceException(ResponseCode.REPETITIVE_OPERATION.getMsg());
        }
    }
}
```

TestApplication

```

@SpringBootApplication
@MapperScan("com.wangzaiplus.test.mapper")
public class TestApplication extends WebMvcConfigurerAdapter {

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

    /**
     * 跨域
     * @return
     */
    @Bean
    public CorsFilter corsFilter() {
        final UrlBasedCorsConfigurationSource urlBasedCorsConfigurationSource = new UrlBasedCorsConfigurationSource();
        final CorsConfiguration corsConfiguration = new CorsConfiguration();
        corsConfiguration.setAllowCredentials(true);
        corsConfiguration.addAllowedOrigin("*");
        corsConfiguration.addAllowedHeader("*");
        corsConfiguration.addAllowedMethod("*");
        urlBasedCorsConfigurationSource.registerCorsConfiguration("/**", corsConfiguration);
        return new CorsFilter(urlBasedCorsConfigurationSource);
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // 接口幂等性拦截器
        registry.addInterceptor(apildempotentInterceptor());
        super.addInterceptors(registry);
    }

    @Bean
    public ApildempotentInterceptor apildempotentInterceptor() {
        return new ApildempotentInterceptor();
    }
}

```

OK, 目前为止, 校验代码准备就绪, 接下来测试验证

七、测试验证

1、获取token的控制器TokenController

```

@RestController
@RequestMapping("/token")
public class TokenController {

    @Autowired
    private TokenService tokenService;

    @GetMapping
    public ServerResponse token() {
        return tokenService.createToken();
    }
}

```

2、TestController, 注意@Apildempotent注解, 在需要幂等性校验的方法上声明此注解即可, 不需要校验的无影响

```

@RestController
@RequestMapping("/test")
@Slf4j
public class TestController {

    @Autowired
    private TestService testService;

    @ApiDempotent
    @PostMapping("testIdempotence")
    public ServerResponse testIdempotence() {
        return testService.testIdempotence();
    }

}

```

3、获取token

The screenshot shows a REST client interface with a sidebar containing various API endpoints. The 'getToken' endpoint is selected and highlighted in blue. The main panel displays the configuration for this endpoint: the URL is `http://localhost:8080/token`, the method is GET, and the response type is JSON. Below this, there is a preview area showing the JSON response body:

```

{
    "status": 0,
    "msg": "token:gQ6FJleuJ3J7MF55yoY7el",
    "data": null
}

```

查看redis

The screenshot shows the Redis Desktop Manager interface. A key named 'token' is expanded, revealing a single string value: 'token:gQ6FJleuJ3J7MF55yoY7el'. A red arrow points from the left side of the Redis interface towards this token entry.

4、测试接口安全性: 利用jmeter测试工具模拟50个并发请求, 将上一步获取到的token作为参数

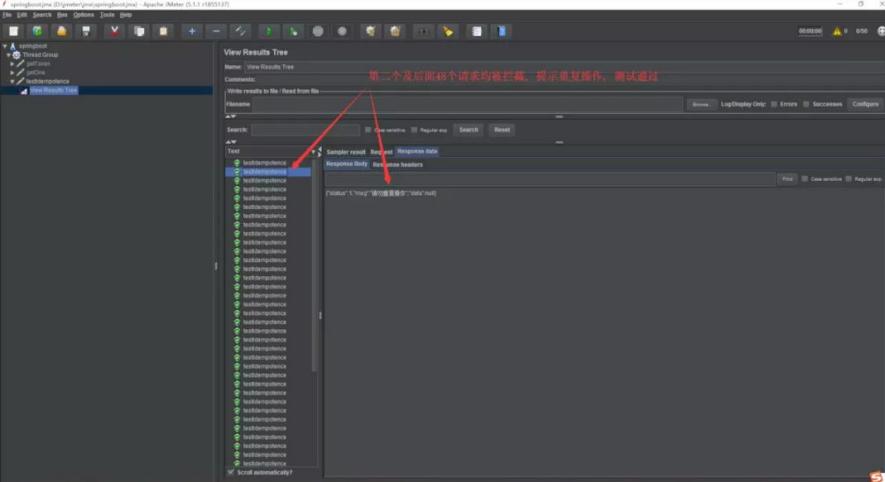
The screenshot shows the JMeter 'View Results Tree' listener. The results pane displays the response for the first request, which is highlighted in red. The response body is shown as:

```

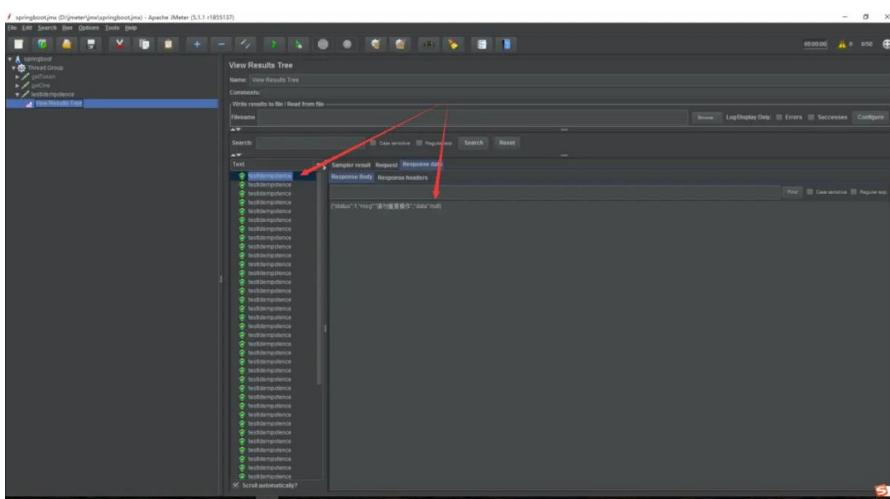
{
    "status": 0,
    "msg": "token:gQ6FJleuJ3J7MF55yoY7el",
    "data": null
}

```

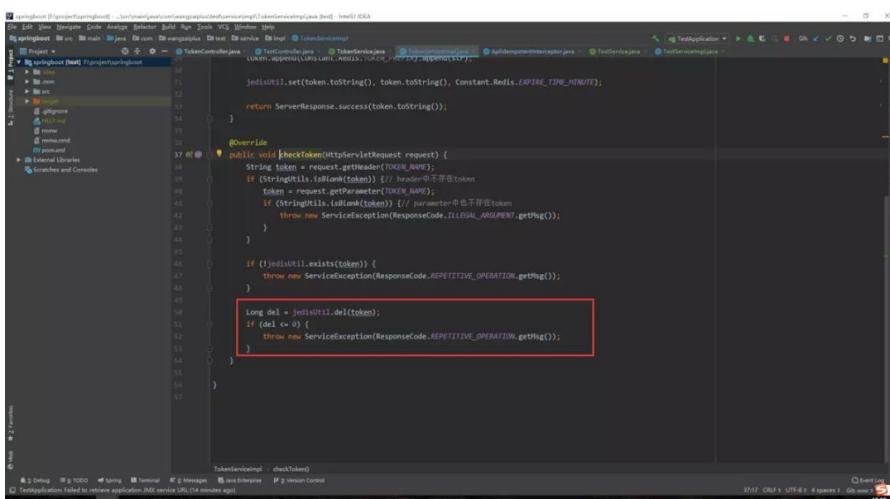
A red arrow points from the right side of the JMeter interface towards the successful response message.



5、header或参数均不传token, 或者token值为空, 或者token值乱填, 均无法通过校验, 如token值为"abcd"



八、注意点(非常重要)



上图中,不能单纯的直接删除token而不校验是否删除成功,会出现并发安全性问题,因为,有可能多个线程同时走到第46行,此时token还未被删除,所以继续往下执行,如果不校验`redisUtil.del(token)`的删除结果而直接放行,那么还是会出现重复提交问题,即使实际上只有一次真正的删除操作,下面重现一下

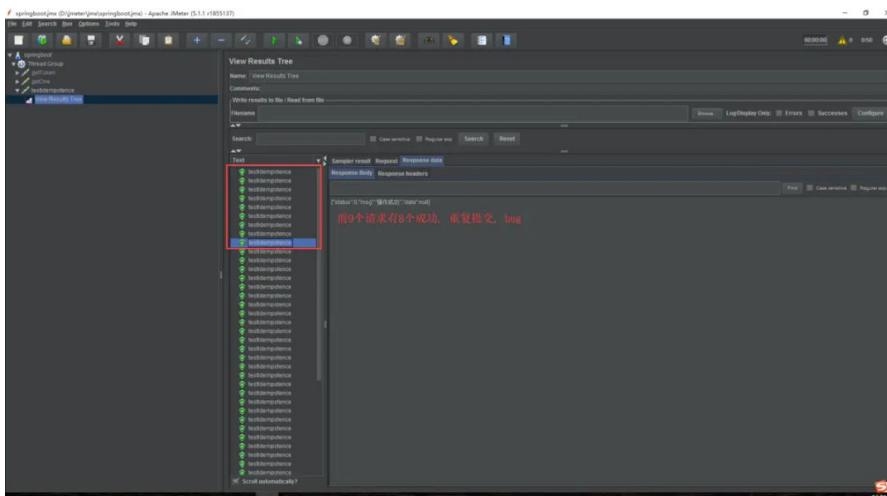
稍微修改一下代码:

```
@Override
public void checkToken(HttpServletRequest request) {
    String token = request.getHeader(TOKEN_NAME);
    if (StringUtils.isBlank(token)) {// header中不存在token
        token = request.getParameter(TOKEN_NAME);
        if (StringUtils.isBlank(token)) {// parameter中也不存在token
            throw new ServiceException(ResponseCode.ILLEGAL_ARGUMENT.getMsg());
        }
    }

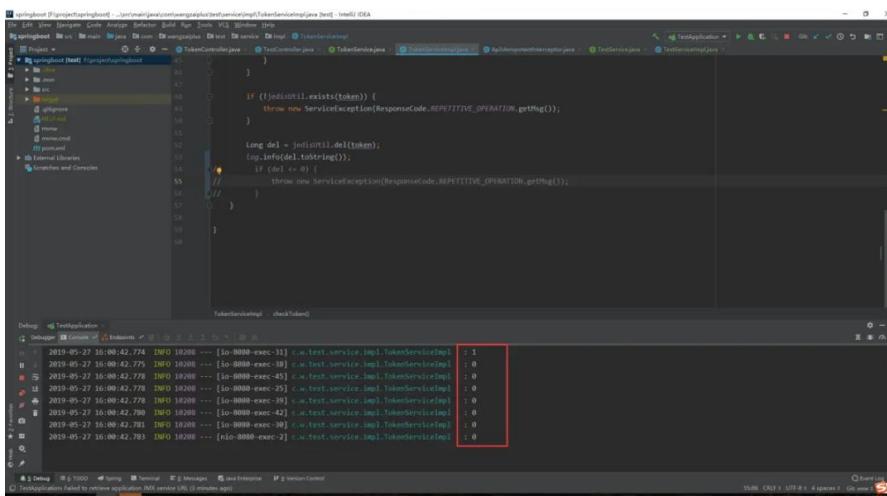
    if (!jedisUtil.exists(token)) {
        throw new ServiceException(ResponseCode.REPETITIVE_OPERATION.getMsg());
    }

    Long del = jedisUtil.del(token);
    log.info(del.toString());
    if (del <= 0) {
        throw new ServiceException(ResponseCode.REPETITIVE_OPERATION.getMsg());
    }
}
```

再次请求



再看看控制台



虽然只有一个真正删除掉token,但由于没有对删除结果进行校验,所以还是有并发问题,因此,必须校验

九、总结

其实思路很简单,就是每次请求保证唯一性,从而保证幂等性,通过拦截器+注解,就不用每次请求都写重复代码,其实也可以利用spring aop实现,无所谓。

Github

- END -

推荐阅读

- [1. Github标星10.8K!Java 实战博客项目分享](#)
- [2. 浅析VO、DTO、DO、PO的概念、区别和用处](#)
- [3. 为什么年终奖是一个彻头彻尾的职场圈套？](#)
- [4. 什么是一致性 Hash 算法？](#)
- [5. 团队开发中 Git 最佳实践](#)

↓ 公众号推荐,方向:机器学习、深度学习 ↓



喜欢文章, 点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

Spring Boot+Redis 分布式锁：模拟抢单

神牛003 Java后端 2月17日

来源：cnblogs.com/wangrudong003/p/10627539.html

本篇内容主要讲解的是redis分布式锁，这个在各大厂面试几乎都是必备的，下面结合模拟抢单的场景来使用她；本篇不涉及到的redis环境搭建，快速搭建个人测试环境，这里建议使用docker；本篇内容节点如下：

jedis的nx生成锁

- 如何删除锁
- 模拟抢单动作(10w个人开抢)
- jedis的nx生成锁

对于java中想操作redis，好的方式是使用jedis，首先pom中引入依赖：

```
<dependency>
<groupId>redis.clients</groupId>
<artifactId>jedis</artifactId>
</dependency>
```

对于分布式锁的生成通常需要注意如下几个方面：

- **创建锁的策略：** redis的普通key一般都允许覆盖，A用户set某个key后，B在set相同的key时同样能成功，如果是锁场景，那就无法知道到底是哪个用户set成功的；这里jedis的setnx方式为我们解决了这个问题，简单原理是：当A用户先set成功了，那B用户set的时候就返回失败，满足了某个时间点只允许一个用户拿到锁。
- **锁过期时间：** 某个抢购场景时候，如果没有过期的概念，当A用户生成了锁，但是后面的流程被阻塞了一直无法释放锁，那其他用户此时获取锁就会一直失败，无法完成抢购的活动；当然正常情况一般都不会阻塞，A用户流程会正常释放锁；过期时间只是为了更有保障。

下面来上段setnx操作的代码：

```
public boolean setnx(String key, String val) {
    Jedis jedis = null;
    try {
        jedis = jedisPool.getResource();
        if (jedis == null) {
            return false;
        }
        return jedis.set(key, val, "NX", "PX", 1000 * 60).
            equalsIgnoreCase("ok");
    } catch (Exception ex) {
    } finally {
        if (jedis != null) {
            jedis.close();
        }
    }
    return false;
}
```

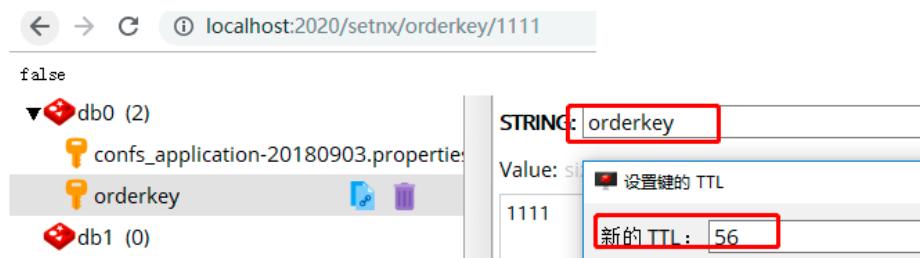
这里注意点在于jedis的set方法，其参数的说明如：

- NX: 是否存在key, 存在就不set成功
- PX: key过期时间单位设置为毫秒 (EX: 单位秒)

setnx如果失败直接封装返回false即可, 下面我们通过一个get方式的api来调用下这个setnx方法:

```
@GetMapping("/setnx/{key}/{val}")
public boolean setnx(@PathVariable String key, @PathVariable String val) {
    return jedisCom.setnx(key, val);
}
```

访问如下测试url, 正常来说第一次返回了true, 第二次返回了false, 由于第二次请求的时候redis的key已存在, 所以无法set成功



由上图能够看到只有一次set成功, 并key具有一个有效时间, 此时已到达了分布式锁的条件。

如何删除锁

上面是创建锁, 同样的具有有效时间, 但是我们不能完全依赖这个有效时间, 场景如: 有效时间设置1分钟, 本身用户A获取锁后, 没遇到什么特殊情况正常生成了抢购订单后, 此时其他用户应该能正常下单了才对, 但是由于有个1分钟后锁才能自动释放, 那其他用户在这1分钟无法正常下单 (因为锁还是A用户的), 因此我们需要A用户操作完后, 主动去解锁:

```
public int delnx(String key, String val) {
    Jedis jedis = null;
    try {
        jedis = jedisPool.getResource();
        if (jedis == null) {
            return 0;
        }

        //if redis.call('get','orderkey')=='1111' then return redis.call('del','orderkey') else return 0 end
        StringBuilder sbScript = new StringBuilder();
        sbScript.append("if redis.call('get','").append(key).append("')=='").append(val).append("').append(" then ").
            append(" return redis.call('del','").append(key).append("')").
            append(" else ").
            append(" return 0").
            append(" end");

        return Integer.valueOf(jedis.eval(sbScript.toString()));
    } catch (Exception ex) {
    } finally {
        if (jedis != null) {
            jedis.close();
        }
    }
    return 0;
}
```

这里也使用了jedis方式, 直接执行lua脚本: 根据val判断其是否存在, 如果存在就del;

其实个人认为通过jedis的get方式获取val后，然后再比较value是否是当前持有锁的用户，如果是那最后再删除，效果其实相同；只不过直接通过eval执行脚本，这样避免多一次操作了redis而已，缩短了原子操作的间隔。(如有不同见解请留言探讨)；同样这里创建个get方式的api来测试：

```
@GetMapping("/deInx/{key}/{val}")
public int deInx(@PathVariable String key, @PathVariable String val) {
    return jedisCom.deInx(key, val);
}
```

注意的是deInx时，需要传递创建锁时的value，因为通过et的value与deInx的value来判断是否是持有锁的操作请求，只有value一样才允许del；

模拟抢单动作(10w个人开抢)

有了上面对分布式锁的粗略基础，我们模拟下10w人抢单的场景，其实就是一个并发操作请求而已，由于环境有限，只能如此测试；如下初始化10w个用户，并初始化库存，商品等信息，如下代码：

```
//总库存
private long nKuCuen = 0;
//商品key名字
private String shangpingKey = "computer_key";
//获取锁的超时时间 秒
private int timeout = 30 * 1000;

@GetMapping("/qiangdan")
public List<String> qiangdan() {

    //抢到商品的用户
    List<String> shopUsers = new ArrayList<>();

    //构造很多用户
    List<String> users = new ArrayList<>();
    IntStream.range(0, 100000).parallel().forEach(b -> {
        users.add("神牛-" + b);
    });

    //初始化库存
    nKuCuen = 10;

    //模拟开抢
    users.parallelStream().forEach(b ->{
        String shopUser = qiang(b);
        if (!StringUtils.isEmpty(shopUser)) {
            shopUsers.add(shopUser);
        }
    });
}

return shopUsers;
}
```

有了上面10w个不同用户，我们设定商品只有10个库存，然后通过并行流的方式来模拟抢购，如下抢购的实现：

```

/**
 * 模拟抢单动作
 *
 * @param b
 * @return
 */
private String qiang(String b) {
    //用户开抢时间
    long startTime = System.currentTimeMillis();

    //未抢到的情况下，30秒内继续获取锁
    while ((startTime + timeout) >= System.currentTimeMillis()) {
        //商品是否剩余
        if (nKuCuen <= 0) {
            break;
        }
        if (jedisCom.setnx(shangpingKey, b)) {
            //用户b拿到锁
            logger.info("用户{}拿到锁...", b);
            try {
                //商品是否剩余
                if (nKuCuen <= 0) {
                    break;
                }
            }

            //模拟生成订单耗时操作，方便查看：神牛-50 多次获取锁记录
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            //抢购成功，商品递减，记录用户
            nKuCuen -= 1;

            //抢单成功跳出
            logger.info("用户{}抢单成功跳出...所剩库存：{}", b, nKuCuen);

            return b + "抢单成功，所剩库存：" + nKuCuen;
        } finally {
            logger.info("用户{}释放锁...", b);
            //释放锁
            jedisCom.delnx(shangpingKey, b);
        }
    } else {
        //用户b没拿到锁，在超时范围内继续请求锁，不需要处理
        //if(b.equals("神牛-50") || b.equals("神牛-69")) {
        //logger.info("用户{}等待获取锁...", b);
        //}
        //
    }
    return "";
}

```

这里实现的逻辑是：

- parallelStream()：并行流模拟多用户抢单
- (startTime + timeout) >= System.currentTimeMillis()：判断未抢成功的用户，timeout秒内继续获取锁
- 获取锁前和后都判断库存是否还足够

- jedisCom.setnx(shangpingKey, b): 用户获取抢购锁
- 获取锁后并下单成功，最后释放锁：jedisCom.delnx(shangpingKey, b)

再来看下记录的日志结果：

```
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-53182拿到锁...
[onPool-worker-3] [com.mn.controller.UserController] : 用户神牛-98000拿到锁...
[onPool-worker-3] [com.mn.controller.UserController] : 用户神牛-98000抢单成功跳出...所剩库存: 7
[onPool-worker-3] [com.mn.controller.UserController] : 用户神牛-98000释放锁...
[onPool-worker-3] [com.mn.controller.UserController] : 用户神牛-12878拿到锁...
[onPool-worker-3] [com.mn.controller.UserController] : 用户神牛-12878抢单成功跳出...所剩库存: 6
[onPool-worker-3] [com.mn.controller.UserController] : 用户神牛-12878释放锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-28036拿到锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-28036抢单成功跳出...所剩库存: 5
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-28036释放锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-53183拿到锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-53183抢单成功跳出...所剩库存: 4
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-53183释放锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-12879拿到锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-12879抢单成功跳出...所剩库存: 3
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-12879释放锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-12880拿到锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-12880抢单成功跳出...所剩库存: 2
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-12880释放锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-28037拿到锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-28037抢单成功跳出...所剩库存: 1
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-28037释放锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-12881拿到锁...
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-12881抢单成功跳出...所剩库存: 0
[onPool-worker-2] [com.mn.controller.UserController] : 用户神牛-12881释放锁...
```

最终返回抢购成功的用户：

```
← → ⌂ localhost:2020/qiangdan
["神牛-3120抢单成功，所剩库存: 9","神牛-53182抢单成功，所剩库存: 8","神牛-98000抢单成功，所剩库存: 7","神牛-12878抢单成功，所剩库存: 6","神牛-12879抢单成功，所剩库存: 3","神牛-12880抢单成功，所剩库存: 2","神牛-28037抢单成功，所剩库存: 1","神牛-12881抢单成功，所剩库存: 0"]
```

-END-

如果看到这里，说明你喜欢这篇文章，请[转发](#)、[点赞](#)。微信搜索「web_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓ 扫描二维码添加小编 ↓



推荐阅读

1. 揭秘阿里、腾讯、字节跳动在家办公的区别
2. 前后端分离开发，RESTful 接口应该这样设计
3. Spring Boot + Vue 如此强大？
4. 如何获取靠谱的新型冠状病毒疫情



微信搜一搜

Q Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！



微信搜一搜

Java后端

作者 | xiangzhihong

segmentfault.com/a/1190000021376934

前言

虽然 B/S 是目前开发的主流，但是 C/S 仍然有很大的市场需求。受限于浏览器的沙盒限制，网页应用无法满足某些场景下的使用需求，而桌面应用可以读写本地文件、调用更多系统资源，再加上 Web 开发的低成本、高效率的优势，这种跨平台方式越来越受到开发者的喜爱。

Electron 是一个基于 Chromium 和 Node.js，使用 HTML、CSS 和 JavaScript 来构建跨平台应用的跨平台开发框架，兼容 Mac、Windows 和 Linux。目前，Electron 已经创建了包括 VScode 和 Atom 在内的大量应用。

环境搭建

创建 Electron 跨平台应用之前，需要先安装一些常用的工具，如 Node、vue 和 Electron 等。

安装 Node

进入 Node 官网下载页 <http://nodejs.cn/download/>，然后下载对应的版本即可，下载时建议下载稳定版本。如果安装 Node 使用 Homebrew 方式，建议安装时将 npm 仓库镜像改为淘宝镜像，如下所示。

```
npm config set registry http://registry.npm.taobao.org/  
或者  
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

安装/升级 vue-cli

先执行以下命令，确认下本地安装的 vue-cli 版本。

```
vue -V
```

如果没有安装或者不是最新版，可以执行以下命令安装/升级。

```
npm install @vue/cli -g
```

安装 Electron

使用如下命令安装 Electron 插件。

```
npm install -g electron  
或者  
cnpm install -g electron
```

为了验证是否安装成功，可以使用如下的命令。

```
electron --version
```

创建运行项目

Electron 官方提供了一个简单的项目，可以执行以下命令将项目克隆到本地。

```
git clone https://github.com/electron/electron-quick-start
```

然后在项目中执行如下命令即可启动项目。

```
cd electron-quick-start  
npm install  
npm start
```

启动后项目的效果如下图。

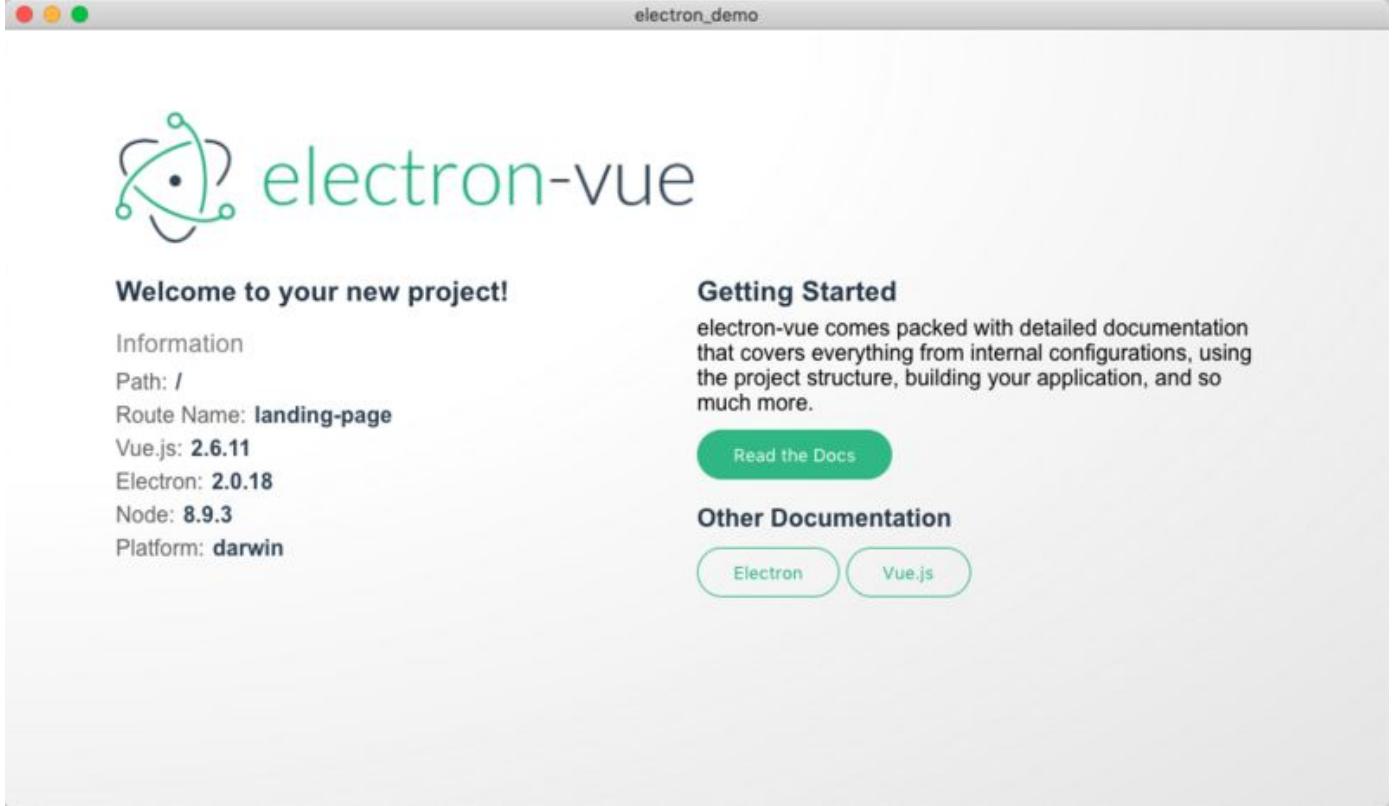


除此之外，我们可以使用 vue-cli 脚手架工具来创建项目。

```
vue init simulatedgreg/electron-vue
```

```
xzh:electron_demo bilibili$ vue init simulatedgreg/electron-vue  
? Generate project in current directory? Yes  
? Application Name electron_demo  
? Application Id com.example.yourapp  
? Application Version 0.0.1  
? Project description An electron-vue project  
? Use Sass / Scss? No  
? Select which Vue plugins to install (Press <space> to select, <a> to toggle all, <i>  
to invert selection) axios, vue-electron, vue-router, vuex, vuex-electron  
? Use linting with ESLint? No  
? Set up unit testing with Karma + Mocha? No  
? Set up end-to-end testing with Spectron + Mocha? No  
? What build tool would you like to use? builder  
? author zhihong xiang <xiangzhihong@bilibili.com>  
  
vue-cli · Generated "electron_demo".
```

然后，使用 npm install 命令安装项目所需要的依赖包，安装完成之后，可以使用 npm run dev 或 npm run build 命令运行 electron-vue 模版应用程序，运行效果如下图所示。



Electron 源码目录

Electron 的源代码主要依据 Chromium 的拆分约定被拆成了许多部分。为了更好地理解源代码，您可能需要了解一下 Chromium 的多进程架构。

Electron 源码目录结构和含义具体如下：

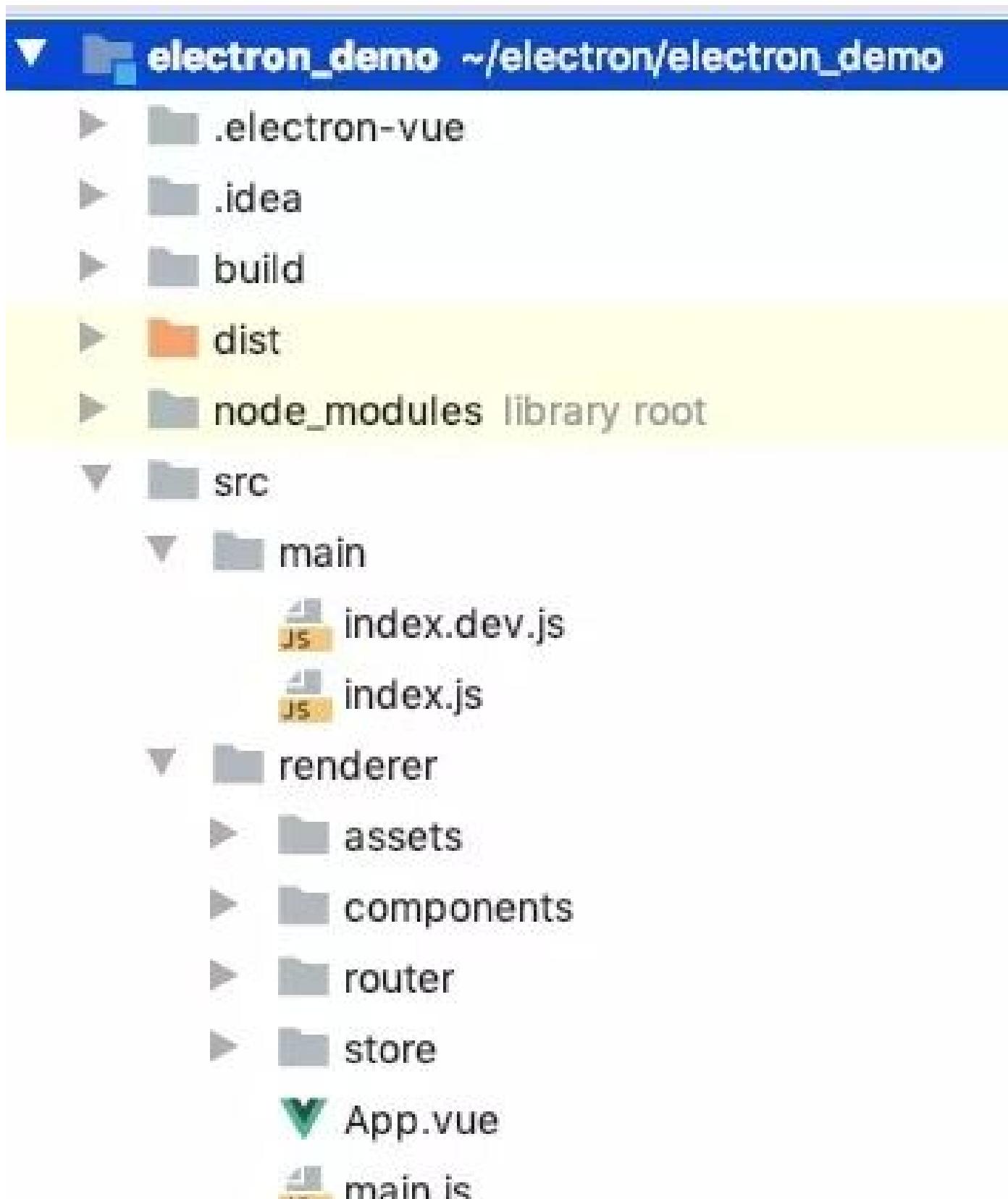
```
Electron
├── atom - Electron 的源代码
│   ├── app - 系统入口代码
│   ├── browser - 包含了主窗口、UI 和其他所有与主进程有关的东西，它会告诉渲染进程如何管理页面
│   │   ├── lib - 主进程初始化代码中 JavaScript 部分的代码
│   │   ├── ui - 不同平台上 UI 部分的实现
│   │   │   ├── cocoa - Cocoa 部分的源代码
│   │   │   ├── gtk - GTK+ 部分的源代码
│   │   │   └── win - Windows GUI 部分的源代码
│   │   ├── default_app - 在没有指定 app 的情况下 Electron 启动时默认显示的页面
│   │   ├── api - 主进程 API 的实现
│   │   │   └── lib - API 实现中 Javascript 部分的代码
│   │   ├── net - 网络相关的代码
│   │   ├── mac - 与 Mac 有关的 Objective-C 代码
│   │   └── resources - 图标，平台相关的文件等
│   ├── renderer - 运行在渲染进程中的代码
│   │   ├── lib - 渲染进程初始化代码中 JavaScript 部分的代码
│   │   └── api - 渲染进程 API 的实现
│   │       └── lib - API 实现中 Javascript 部分的代码
│   └── common - 同时被主进程和渲染进程用到的代码，包括了一些用来将 node 的事件循环
│       整合到 Chromium 的事件循环中时用到的工具函数和代码
├── lib - 同时被主进程和渲染进程使用到的 Javascript 初始化代码
└── api - 同时被主进程和渲染进程使用到的 API 的实现以及 Electron 内置模块的基础设施
    └── lib - API 实现中 Javascript 部分的代码
├── chromium_src - 从 Chromium 项目中拷贝来的代码
├── docs - 英语版本的文档
├── docs-translations - 各种语言版本的文档翻译
├── spec - 自动化测试
├── atom.gyp - Electron 的构建规则
└── common.gypi - 为诸如 `node` 和 `breakpad` 等其他组件准备的编译设置和构建规则
```

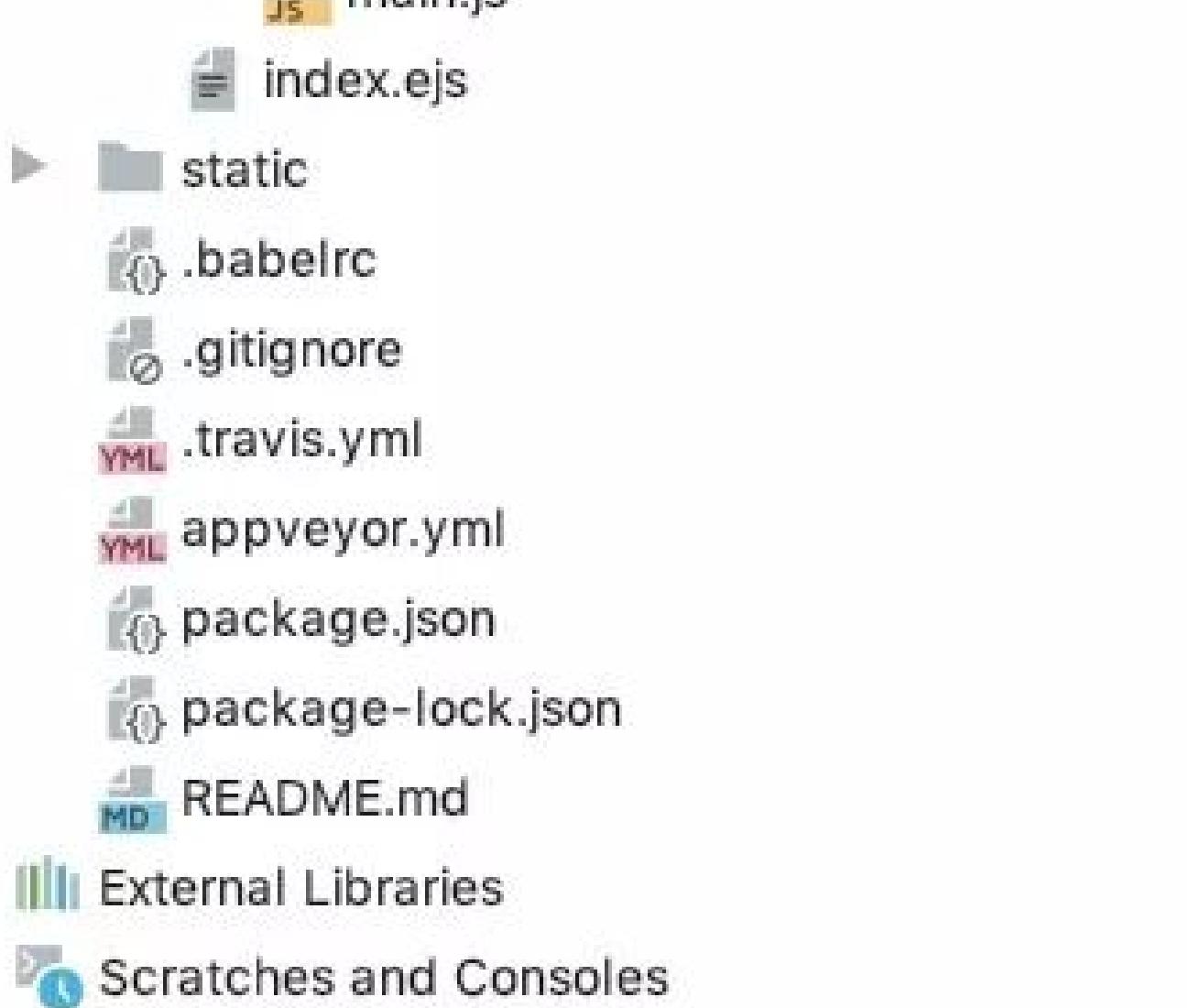
平时开发时，需要重点关注的就是 src、package.json 和 appveyor.yml 目录。除此之外，其他需要注意的目录如下：

- script - 用于诸如构建、打包、测试等开发用途的脚本
- tools - 在 gyp 文件中用到的工具脚本，但与 script 目录不同，该目录中的脚本不应该被用户直接调用
- vendor - 第三方依赖项的源代码，为了防止人们将它与 Chromium 源码中的同名目录相混淆，在这里我们不使用 third_party 作为目录名
- node_modules - 在构建中用到的第三方 node 模块
- out - ninja 的临时输出目录
- dist - 由脚本 script/create-dist.py 创建的临时发布目录
- external_binaries - 下载的不支持通过 gyp 构建的预编译第三方框架

应用工程目录

使用 electron-vue 模版创建的 Electron 工程结构如下图。





和前端工程的项目结构类似，Electron项目的目录结构如下所示：

- electron-vue：Electron模版配置。
- build：文件夹，用来存放项目构建脚本。
- config：中存放项目的一些基本配置信息，最常用的就是端口转发。
- node_modules：这个目录存放的是项目的所有依赖，即 npm install 命令下载下来的文件。
- src：这个目录下存放项目的源码，即开发者写的代码放在这里。
- static：用来存放静态资源。
- index.html：则是项目的首页、入口页，也是整个项目唯一的HTML页面。
- package.json：中定义了项目的所有依赖，包括开发时依赖和发布时依赖。

对于开发者来说，90%的工作都是在 src 中完成，src 中的文件目录如下。



1、主进程

Electron 运行 package.json 的 main 脚本 (background.js) 的进程称为主进程。在主进程中运行的脚本通过创建 web 页面来展示用户界面。一个 Electron 应用总是有且只有一个主进程。

2、渲染进程

由于 Electron 使用了 Chromium 来展示 Web 页面，所以 Chromium 的多进程架构也被使用到。每个 Electron 中的 Web 页面运行在它自己的渲染进程中。在普通的浏览器中，Web 页面通常在一个沙盒环境中运行，不被允许去接触原生的资源。然而 Electron 允许用户在 Node.js 的 API 支持下可以在页面中和操作系统进行一些底层交互。

3、主进程与渲染进程通信

主进程使用 BrowserWindow 实例创建页面。每个 BrowserWindow 实例都在自己的渲染进程中运行页面。当一个 BrowserWindow 实例被销毁后，相应的渲染进程也会被终止。主进程管理所有的 Web 页面和它们对应的渲染进程。每个渲染进程都是独立的，它只关心它所运行的 Web 页面。

src 目录结构

在 Electron 目录中，src 会包含 main 和 renderer 两个目录。

main 目录

main 目录会包含 index.js 和 index.dev.js 两个文件。

- index.js：应用程序的主文件，electron 也从这里启动的，它也被用作 webpack 产品构建的入口文件，所有的 main 进程工作都应该从这里开始。
- index.dev.js：此文件专门用于开发阶段，因为它会安装 electron-debug 和 vue-devtools。一般不需要修改此文件，但它可以扩展开发的需求。

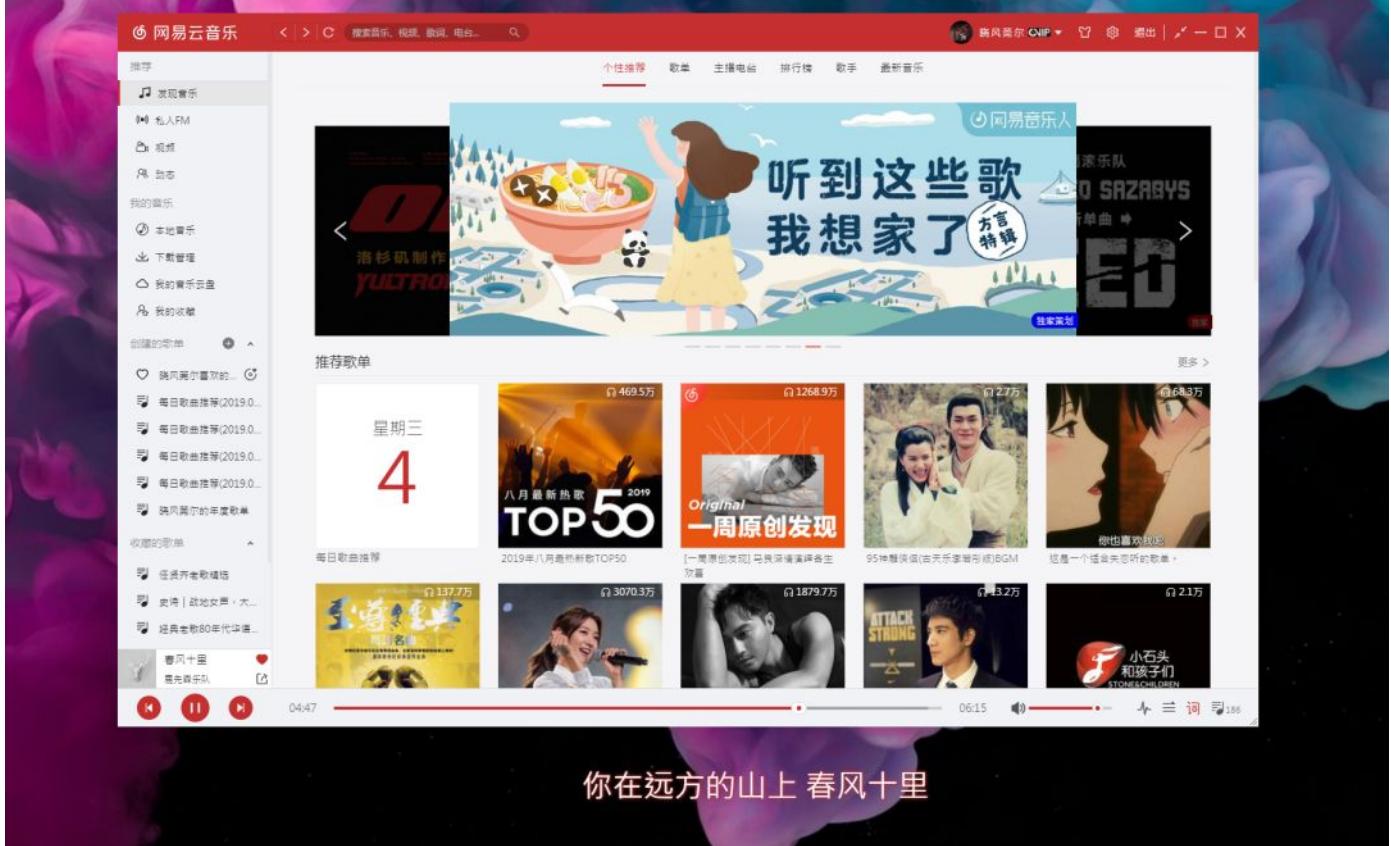
渲染进程

renderer 是渲染进程目录，平时项目开发源码的存放目录，包含 assets、components、router、store、App.vue 和 main.js。

assets：assets 下的文件如 (js、css) 都会在 dist 文件夹下面的项目目录分别合并到一个文件里面去。components：此文件用于存放应用开发的组件，可以是自定义的组件。router：如果你了解 vue-router，那么 Electron 项目的路由的使用方式和 vue-router 的使用方式类似。modules：electron-vue 利用 vuex 的模块结构创建多个数据存储，并保存在 src/renderer/store/modules 中。

相关案例

- <https://github.com/xiaozhu188/electron-vue-cloud-music>



- <https://github.com/SmallRuralDog/electron-vue-music>

推荐阅读

1. 为了去阿里,我准备了一年,没想到竟是这样的结果
2. 大白话带你梳理一下 Dubbo 的那些事儿
3. 安利一款 IDEA 中强大的代码生成利器



微信搜一搜

Q Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

Spring Boot一个依赖搞定 Session 共享

江南一点雨 Java后端 2019-11-25

点击上方 Java后端, 选择 [设为星标](#)

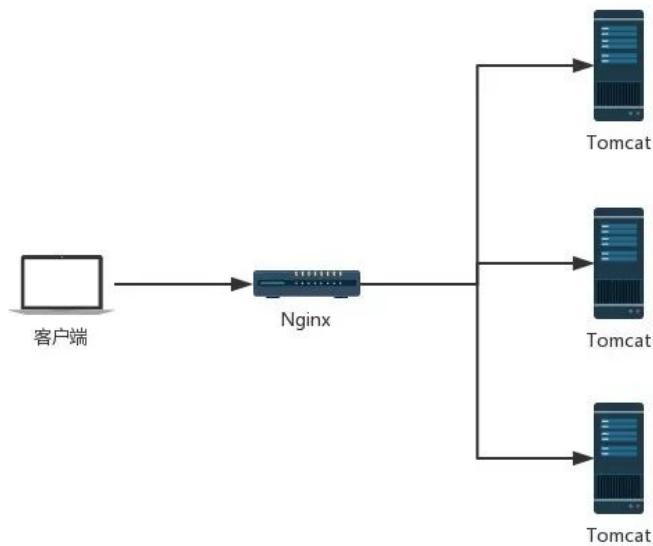
优质文章, 及时送达

作者：江南一点雨

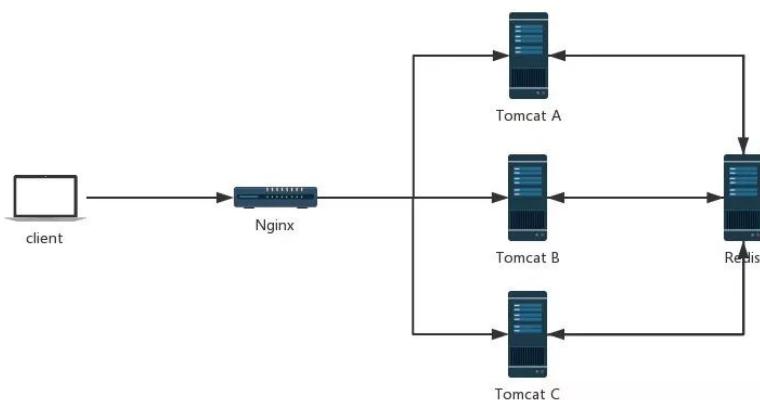
微信公众号：牧码小子 (ID: a_javaboy)

有的人可能会觉得题目有点夸张，其实不夸张，题目没有使用任何修辞手法！认真读完本文，你就知道说的是对的了！

在传统的单服务架构中，一般来说，只有一个服务器，那么不存在 Session 共享问题，但是在分布式/集群项目中，Session 共享则是一个必须面对的问题，先看一个简单的架构图：



在这样的架构中，会出现一些单服务中不存在的问题，例如客户端发起一个请求，这个请求到达 Nginx 上之后，被 Nginx 转发到 Tomcat A 上，然后在 Tomcat A 上往 session 中保存了一份数据，下次又来一个请求，这个请求被转发到 Tomcat B 上，此时再去 Session 中获取数据，发现没有之前的数据。对于这一类问题的解决，思路很简单，就是将各个服务之间需要共享的数据，保存到一个公共的地方（主流方案就是 Redis）：



当所有 Tomcat 需要往 Session 中写数据时，都往 Redis 中写，当所有 Tomcat 需要读数据时，都从 Redis 中读。这样，不同的服务就可以使用相同的 Session 数据了。

这样的方案，可以由开发者手动实现，即手动往 Redis 中存储数据，手动从 Redis 中读取数据，相当于使用一些 Redis 客户端工具来实现这样的功能，毫无疑问，手动实现工作量还是蛮大的。

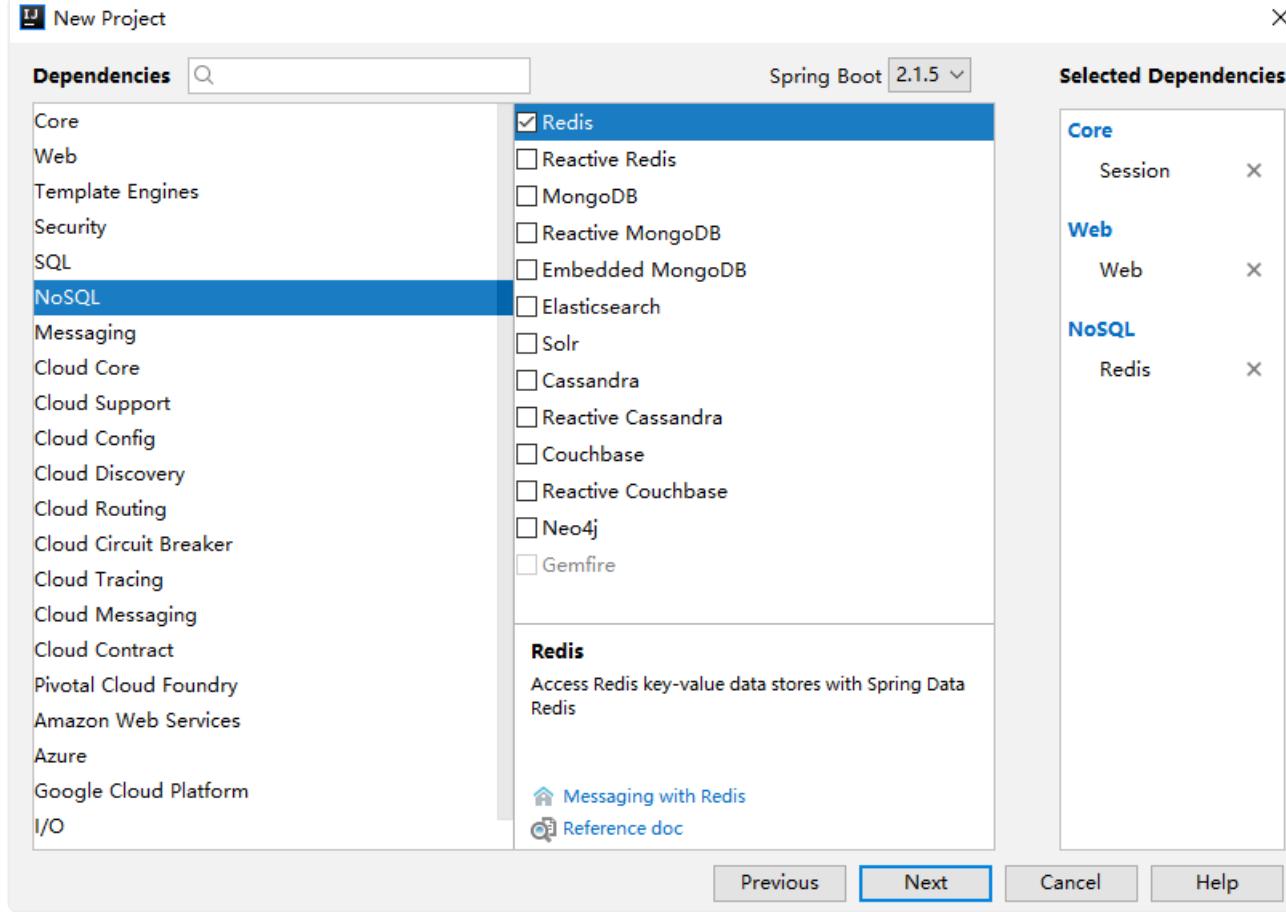
一个简化的方案就是使用 Spring Session 来实现这一功能，Spring Session 就是使用 Spring 中的代理过滤器，将所有的 Session 操作拦截下来，自动的将数据同步到 Redis 中，或者自动的从 Redis 中读取数据。

对于开发者来说，所有关于 Session 同步的操作都是透明的，开发者使用 Spring Session，一旦配置完成后，具体的用法就像使用一个普通的 Session 一样。

一、实战

1.1、创建工程

首先创建一个 Spring Boot 工程，引入 Web、Spring Session 以及 Redis：



创建成功之后，pom.xml 文件如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.session</groupId>
        <artifactId>spring-session-data-redis</artifactId>
    </dependency>
</dependencies>
```

注意：

这里我使用的 Spring Boot 版本是 2.1.4，如果使用当前最新版 Spring Boot 2.1.5 的话，除了上面这些依赖之外，需要额外添加 Spring Security 依赖（其他操作不受影响，仅仅只是多了一个依赖，当然也多了 Spring Security 的一些默认认证流程）。

1.2、配置 Redis

```
spring.redis.host=192.168.66.128  
spring.redis.port=6379  
spring.redis.password=123  
spring.redis.database=0
```

这里的 Redis，我虽然配置了四行，但是考虑到端口默认就是 6379， database 默认就是 0，所以真正要配置的，其实就是两行。

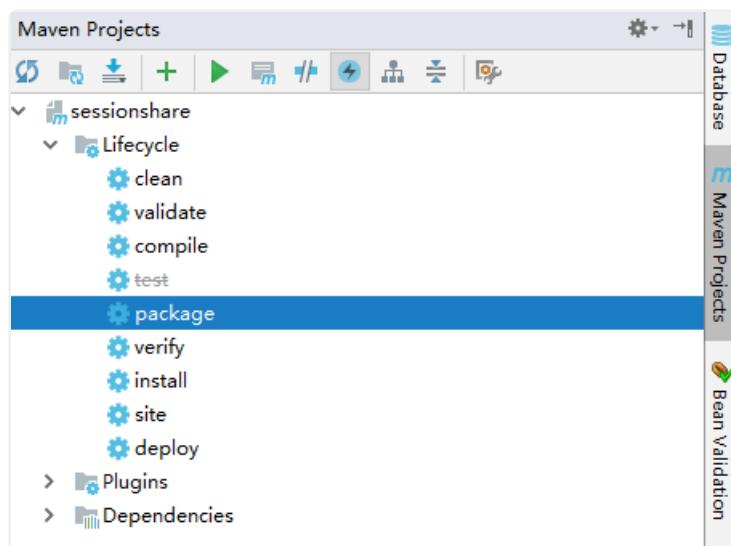
1.3、使用

配置完成后，就可以使用 Spring Session 了，其实也就是使用普通的 HttpSession，其他的 Session 同步到 Redis 等操作，框架已经自动帮你完成了：

```
@RestController  
public class HelloController {  
    @Value("${server.port}")  
    Integer port;  
    @GetMapping("/set")  
    public String set(HttpSession session) {  
        session.setAttribute("user", "javaboy");  
        return String.valueOf(port);  
    }  
    @GetMapping("/get")  
    public String get(HttpSession session) {  
        return session.getAttribute("user") + ":" + port;  
    }  
}
```

考虑到一会 Spring Boot 将以集群的方式启动，为了获取每一个请求到底是哪一个 Spring Boot 提供的服务，需要在每次请求时返回当前服务的端口号，因此这里我注入了 server.port。

接下来，项目打包：



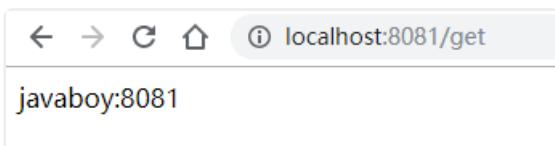
打包之后，启动项目的两个实例：

```
java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8080  
java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8081
```

然后先访问 `localhost:8080/set` 向 8080 这个服务的 Session 中保存一个变量，访问完成后，数据就已经自动同步到 Redis 中了：

```
127.0.0.1:6379> keys *
1) "spring:session:sessions:bla8fbad-8912-4d01-alcf-acf90765bab4"
2) "spring:session:sessions:expires:bla8fbad-8912-4d01-alcf-acf90765bab4"
3) "spring:session:expirations:1559551560000"
```

然后，再调用 `localhost:8081/get` 接口，就可以获取到 8080 服务的 session 中的数据：



此时关于 session 共享的配置就已经全部完成了，session 共享的效果我们已经看到了，但是每次访问都是我自己手动切换服务实例，因此，接下来我们来引入 Nginx，实现服务实例自动切换。

1.4、引入 Nginx

很简单，进入 Nginx 的安装目录的 `conf` 目录下（默认是在 `/usr/local/nginx/conf`），编辑 `nginx.conf` 文件：

```
upstream javaboy.org{
    server 127.0.0.1:8080 weight=1;
    server 127.0.0.1:8081 weight=2;
}
server {
    listen      80;
    server_name localhost;

    #charset koi8-r;

    #access_log  logs/host.access.log  main;

    location / {
        proxy_pass http://javaboy.org;
        proxy_redirect default;
        #root   html;
        #index  index.html index.htm;
    }
}
```

在这段配置中：

- 1、`upstream` 表示配置上游服务器
- 2、`javaboy.org` 表示服务器集群的名字，这个可以随意取名字
- 3、`upstream` 里边配置的是一个个的单独服务
- 4、`weight` 表示服务的权重，意味者将有多少比例的请求从 Nginx 上转发到该服务上
- 5、`location` 中的 `proxy_pass` 表示请求转发的地址，`/` 表示拦截到所有的请求，转发转发到刚刚配置好的服务集群中
- 6、`proxy_redirect` 表示设置当发生重定向请求时，nginx 自动修正响应头数据（默认是 Tomcat 返回重定向，此时重定向的地址是 Tomcat 的地址，我们需要将之修改使之成为 Nginx 的地址）。

Tips：关注微信公众号：Java后端，每日提送技术博文。

配置完成后，将本地的 Spring Boot 打包好的 jar 上传到 Linux，然后在 Linux 上分别启动两个 Spring Boot 实例：

```
nohup java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8080 &
nohup java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8081 &
```

其中

- nohup 表示当终端关闭时，Spring Boot 不要停止运行
- & 表示让 Spring Boot 在后台启动

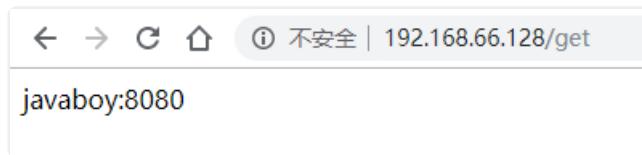
配置完成后，重启 Nginx：

```
/usr/local/nginx/sbin/nginx -s reload
```

Nginx 启动成功后，我们首先手动清除 Redis 上的数据，然后访问 192.168.66.128/set 表示向 session 中保存数据，这个请求首先会到达 Nginx 上，再由 Nginx 转发给某一个 SpringBoot 实例：



如上，表示端口为 8081 的 SpringBoot 处理了这个 /set 请求，再访问 /get 请求：



可以看到， /get 请求是被端口为 8080 的服务所处理的。

二、总结

本文主要向大家介绍了 Spring Session 的使用，另外也涉及到一些 Nginx 的使用，虽然本文较长，但是实际上 Spring Session 的配置没啥。

我们写了一些代码，也做了一些配置，但是全都和 Spring Session 无关，配置是配置 Redis，代码就是普通的 HttpSession，和 Spring Session 没有任何关系！

唯一和 Spring Session 相关的，可能就是我在一开始引入了 Spring Session 的依赖吧！

如果大家没有在 SSM 架构中用过 Spring Session，可能不太好理解我们在 Spring Boot 中使用 Spring Session 有多么方便，因为在 SSM 架构中，Spring Session 的使用要配置三个地方，一个是 web.xml 配置代理过滤器，然后在 Spring 容器中配置 Redis，最后再配置 Spring Session，步骤还是有些繁琐的，而 Spring Boot 中直接帮我们省去了这些繁琐的步骤！不用再去配置 Spring Session。

好了，本文就说到这里，有问题欢迎讨论，本文相关案例我已经上传到 GitHub，大家可以自行下载：

<https://github.com/lenve/javaboy-code-samples>

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



扫一扫上面的二维码，加我微信

推荐阅读

1. 前后端分离开发, RESTful 接口如何设计
2. 面试官:讲一下 Mybatis 初始化原理
3. 我们再来聊一聊 Java 的单例吧
4. 我采访了一位 Pornhub 工程师
5. 团队开发中 Git 最佳实践



喜欢文章，点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

Spring Boot 使用 AOP 实现 REST 接口简易灵活的安全认证

JeffWong Java后端 1月29日



微信搜一搜

Java后端

作者 | JeffWong

链接 | www.cnblogs.com/jeffwongishandsome

本文将通过AOP的方式实现一个相对更加简易灵活的API安全认证服务，我们先看实现，然后介绍和分析AOP基本原理和常用术语。

一、Authorized实现

1、定义注解

```
package com.power.demo.common;

import java.lang.annotation.*;

/*
 * 安全认证
 */
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Authorized {

    String value() default "";
}
```

这个注解看上去什么都没有，仅仅是一个占位符，用于标志是否需要安全认证。

2、表现层使用注解

```
@Authorized
@RequestMapping(value = "/getinfobyid", method = RequestMethod.POST)
@ApiOperation("根据商品Id查询商品信息")
@ApImplicitParams({
    @ApImplicitParam(paramType = "header", name = "authtoken", required = true, value = "authtoken", dataType =
    "String"),
})
public GetGoodsByGoodsIdResponse getGoodsByGoodsId(@RequestHeader String authtoken, @RequestBody GetGoodsByGoodsIdReq

    return _goods ApiService.getGoodsByGoodsId(request);

}

```

看上去就在一个方法上加了Authorized注解，其实它也可以作用于类上，也可以类和方法混合使用。

3、请求认证切面

下面的代码是实现灵活的安全认证的关键：

```
/*
 * 请求认证切面，验证自定义请求header的authtoken是否合法
 */
@Aspect
@Component
public class AuthorizedAspect {

    @Autowired
    private AuthTokenService authTokenService;

    @Pointcut("@annotation(org.springframework.web.bind.annotation.RequestMapping)")
    public void requestMapping() {
    }

    @Pointcut("execution(* com.power.demo.controller.*Controller.*(..))")
    public void methodPointCut() {
    }

    /**
     * 某个方法执行前进行请求合法性认证 注入Authorized注解（先）
     */
    @Before("requestMapping() && methodPointCut()&&@annotation(authorized)")
    public void doBefore(JoinPoint joinPoint, Authorized authorized) throws Exception {

        PowerLogger.info("方法认证开始...");

        Class type = joinPoint.getSignature().getDeclaringType();

        Annotation[] annotations = type.getAnnotationsByType(Authorized.class);

        if (annotations != null && annotations.length > 0) {
            PowerLogger.info("直接类认证");
            return;
        }

        //获取当前http请求
        ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();
        HttpServletRequest request = attributes.getRequest();

        String token = request.getHeader(AppConst.AUTH_TOKEN);
    }
}
```

```

BizResult<String> bizResult = authTokenService.powerCheck(token);

System.out.println(SerializeUtil.Serialize(bizResult));

if (bizResult.getIsOK() == true) {
    PowerLogger.info("方法认证通过");
} else {
    throw new Exception(bizResult.getMessage());
}

}

@Before("requestMapping() && methodPointCut()")
public void doBefore(JoinPoint joinPoint) throws Exception {

PowerLogger.info("类认证开始...");

Annotation[] annotations = joinPoint.getSignature().getDeclaringType().getAnnotationsByType(Authorized.class);

if (annotations == null || annotations.length == 0) {
    PowerLogger.info("类不需要认证");
    return;
}

//获取当前http请求
ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();
HttpServletRequest request = attributes.getRequest();

String token = request.getHeader(AppConst.AUTH_TOKEN);

BizResult<String> bizResult = authTokenService.powerCheck(token);

System.out.println(SerializeUtil.Serialize(bizResult));

if (bizResult.getIsOK() == true) {
    PowerLogger.info("类认证通过");
} else {
    throw new Exception(bizResult.getMessage());
}

}
}

```

需要注意的是，对类和方法上的Authorized处理，定义了重载的处理方法doBefore。AuthTokenService和上文介绍的处理逻辑一样，如果安全认证不通过，则抛出异常。

如果我们在类上或者方法上都加了Authorized注解，不会进行重复安全认证，请放心使用。

4、统一异常处理

上文已经提到过，对所有发生异常的API，都返回统一格式的报文至调用方。主要代码大致如下：

```

/**
 * 全局统一异常处理增强
 */
@ControllerAdvice
public class GlobalExceptionHandler {

    /**
     * API统一异常处理
     */
    @ExceptionHandler(value = Exception.class)
    @ResponseBody
    public ErrorInfo<Exception> jsonApiErrorHandler(HttpServletRequest request, Exception e) {
        ErrorInfo<Exception> errorInfo = new ErrorInfo<>();
        try {
            System.out.println("统一异常处理...");
            e.printStackTrace();

            Throwable innerEx = e.getCause();
            while (innerEx != null) {
                //innerEx.printStackTrace();
                if (innerEx.getCause() == null) {
                    break;
                }
                innerEx = innerEx.getCause();
            }

            if (innerEx == null) {
                errorInfo.setMessage(e.getMessage());
                errorInfo.setError(e.toString());
            } else {
                errorInfo.setMessage(innerEx.getMessage());
                errorInfo.setError(innerEx.toString());
            }

            errorInfo.setData(e);
            errorInfo.setTimestamp(new Date());
            errorInfo.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value()); //500错误
            errorInfo.setUrl(request.getRequestURL().toString());
            errorInfo.setPath(request.getServletPath());
        } catch (Exception ex) {
            ex.printStackTrace();

            errorInfo.setMessage(ex.getMessage());
            errorInfo.setError(ex.toString());
        }

        return errorInfo;
    }
}

```

认证不通过的API调用结果如下：

```
{  
    "timestamp": "2018-06-07T12:23:25.576+0000",  
    "status": 500,  
    "error": "java.lang.Exception: 不存在此authToken: cfad31547575483b8b78f554a27bd4a4",  
    "message": "不存在此authToken: cfad31547575483b8b78f554a27bd4a4",  
    "url": "http://localhost:9090/api/v1/goods/getbyid",  
    "path": "/api/v1/goods/getbyid",  
    "data": {  
        "cause": {  
            "cause": null,  
            "stackTrace": [  
                {  
                    "methodName": "doBefore",  
                    "fileName": "AuthorizedAspect.java",  
                    "lineNumber": 101,  
                    "className": "com.power.demo.controller.tool.AuthorizedAspect",  
                    "nativeMethod": false  
                },  
                {  
                    "methodName": "invoke0",  
                    "fileName": "AuthorizedAspect.java",  
                    "lineNumber": 101,  
                    "className": "com.power.demo.controller.tool.AuthorizedAspect",  
                    "nativeMethod": false  
                }  
            ]  
        }  
    }  
}
```

异常的整个堆栈可以非常非常方便地帮助我们排查到问题。

我们再结合上文来看安全认证的时间先后，根据理论分析和实践发现，**过滤器Filter先于拦截器Interceptor先于自定义Authorized方法认证先于Authorized类认证**。

到这里，我们发现通过AOP框架AspectJ，一个@Aspect注解外加几个方法几十行业务代码，就可以轻松实现对REST API的拦截处理。

那么为什么会有@Pointcut，既然有@Before，是否有@After？

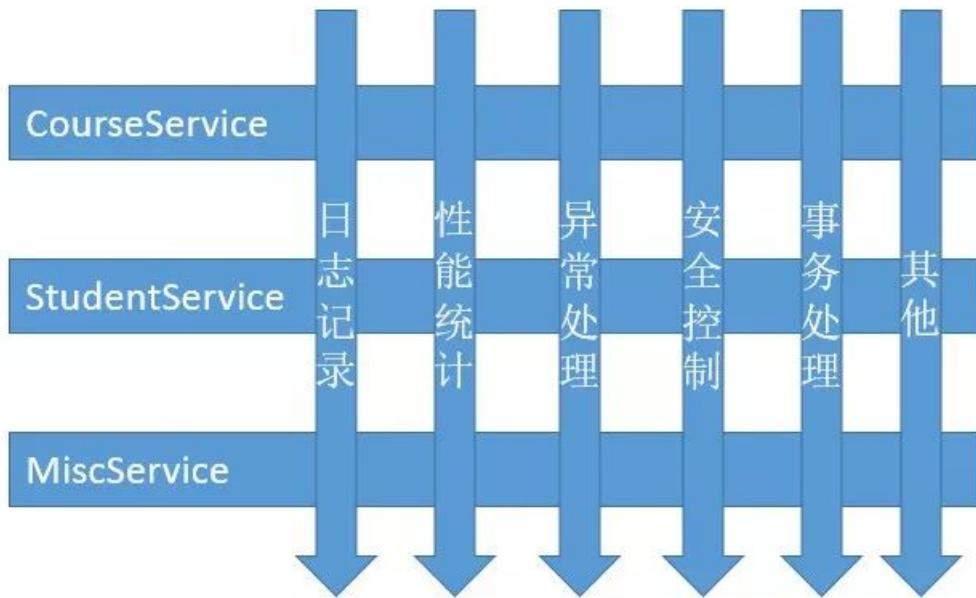
其实上述简易安全认证功能实现的过程主要利用了Spring的AOP特性。

下面再简单介绍下AOP常见概念（主要参考Spring实战），加深理解。AOP概念较多而且比较乏味，经验丰富的老鸟到此就可以忽略这一段了。

二、AOP

1、概述

AOP (Aspect Oriented Programming)，即面向切面编程，可以处理很多事情，常见的功能比如日志记录，性能统计，安全控制，事务处理，异常处理等。



AOP可以认为是一种更高级的“复用”技术，它是OOP（Object Oriented Programming，面向对象编程）的补充和完善。AOP的理念，就是将分散在各个业务逻辑代码中相同的代码通过横向切割的方式抽取到一个独立的模块中。将相同逻辑的重复代码横向抽取出来，使用动态代理技术将这些重复代码织入到目标对象方法中，实现和原来一样的功能。这样一来，我们在写业务逻辑时就只关心业务代码。

OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过OOP允许开发者定义纵向的关系，但并不适合定义横向的关系，例如日志功能。日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为**横切**（cross cutting），在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

AOP技术恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为“Aspect”，即切面。

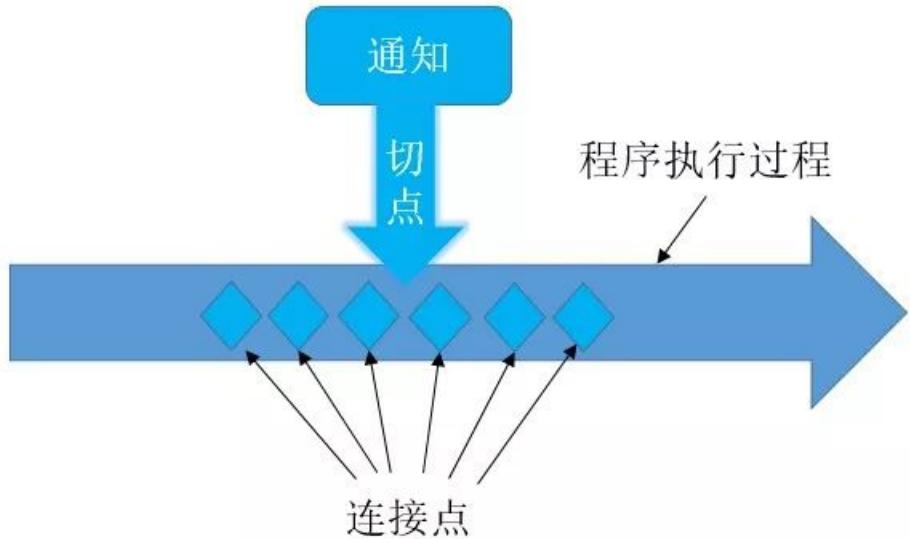
所谓“切面”，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

使用“横切”技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。

业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，它们经常发生在核心关注点的多处，而**各处基本相似**，比如权限认证、日志、事务。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

2、AOP术语

深刻理解AOP，要掌握的术语可真不少。



Target: 目标类，需要被代理的类，如：UserService

Advice: 通知，所要增强或增加的功能，定义了切面的“什么”和“何时”，模式有**Before**、**After**、**After-returning**、**After-throwing**和**Around**

Join Point: 连接点，应用执行过程中，能够插入切面的所有“点”（时机）

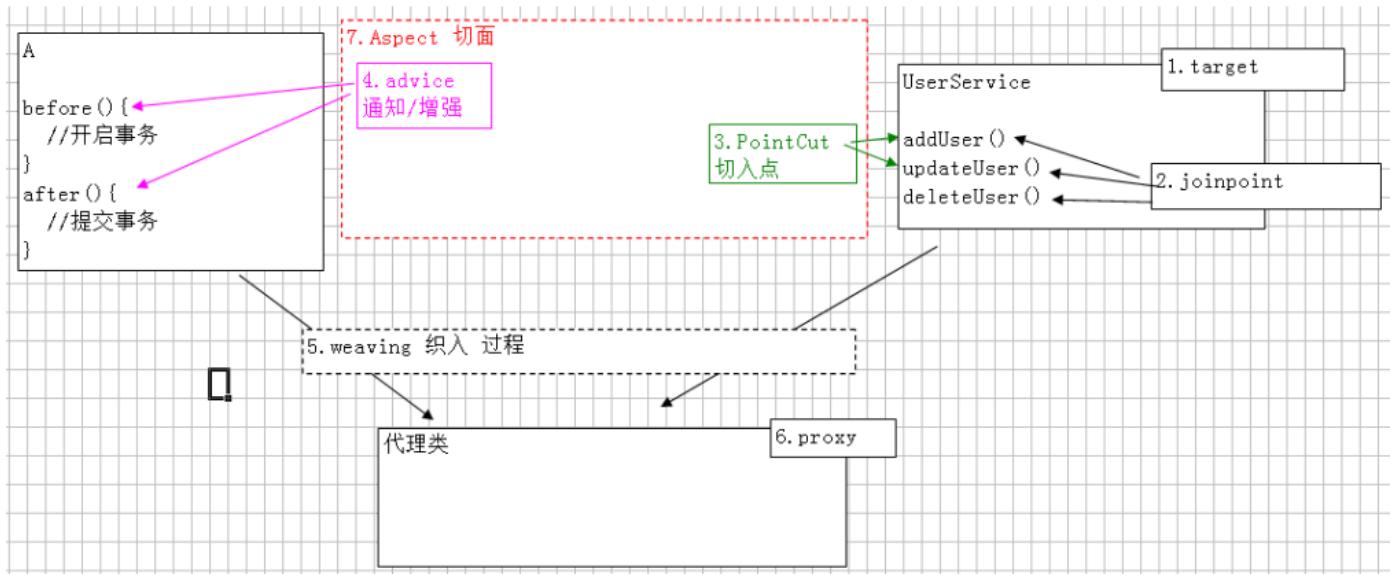
Pointcut: 切点，实际运行中，选择插入切面的连接点，即定义了哪些点得到了增强。切点定义了切面的“何处”。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。

Aspect: 切面，把横切关注点模块化为特殊的类，这些类称为切面，切面是通知和切点的结合。通知和切点共同定义了切面的全部内容：它是什么，在何时和何处完成其功能

Introduction: 引入，允许我们向现有的类添加新方法或属性

Weaving: 织入，把切面应用到目标对象并创建新的代理对象的过程，切面在指定的连接点被织入到目标对象中。在目标对象的生命周期里有多个点可以进行织入：编译期、类加载期、运行期

下面参考自网上图片，可以比较直观地理解上述这几个AOP术语和流转过程。



(1) 动态代理

使用动态代理可以为一个或多个接口在运行期动态生成实现对象，生成的对象中实现接口的方法时可以添加增强代码，从而实现

AOP：

```
/*
 * 动态代理类
 */
public class DynamicProxy implements InvocationHandler {

    /**
     * 需要代理的目标类
     */
    private Object target;

    /**
     * 写法固定, aop专用:绑定委托对象并返回一个代理类
     *
     * @param target
     * @return
     */
    public Object bind(Object target) {
        this.target = target;
        return Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(), this);
    }

    /**
     * 调用 InvocationHandler接口定义方法
     *
     * @param proxy 指被代理的对象。
     * @param method 要调用的方法
     * @param args 方法调用时所需要的参数
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Object result = null;
        // 切面之前执行
        System.out.println("[动态代理]切面之前执行");

        // 执行业务
        result = method.invoke(target, args);

        // 切面之后执行
        System.out.println("[动态代理]切面之后执行");

        return result;
    }
}
```

缺点是只能针对接口进行代理，同时由于动态代理是通过反射实现的，有时可能要考虑反射调用的开销，否则很容易引发性能问题。

(2) 字节码生成

动态字节码生成技术是指在运行时动态生成指定类的一个子类对象（注意是针对类），并覆盖其中特定方法，覆盖方法时可以添加增强代码，从而实现AOP。

最常用的工具是CGLib：

```

import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

/**
 * 使用cglib动态代理
 * <p>
 * JDK中的动态代理使用时，必须有业务接口，而cglib是针对类的
 */
public class CglibProxy implements MethodInterceptor {

    private Object target;

    /**
     * 创建代理对象
     *
     * @param target
     * @return
     */
    public Object getInstance(Object target) {
        this.target = target;
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(this.target.getClass());
        //回调方法
        enhancer.setCallback(this);
        //创建代理对象
        return enhancer.create();
    }

    @Override
    public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
        Object result = null;
        System.out.println("[cglib]切面之前执行");

        result = methodProxy.invokeSuper(proxy, args);

        System.out.println("[cglib]切面之后执行");

        return result;
    }
}

```

(3) 定制的类加载器

当需要对类的所有对象都添加增强，动态代理和字节码生成本质上都需要动态构造代理对象，即最终被增强的对象是由AOP框架生成，不是开发者new出来的。

解决的办法就是实现自定义的类加载器，在一个类被加载时对其进行增强。

JBoss就是采用这种方式实现AOP功能。

这种方式目前只是道听途说，本人没有在实际项目中实践过。

(4) 代码生成

利用工具在已有代码基础上生成新的代码，其中可以添加任何横切代码来实现AOP。

(5) 语言扩展

可以对构造方法和属性的赋值操作进行增强，AspectJ是采用这种方式实现AOP的一个常见的Java语言扩展。

比较：根据日志，上述流程的执行顺序依次为：过滤器、拦截器、AOP方法认证、AOP类认证

附：记录API日志

最后通过记录API日志，记录日志时加入API耗时统计（其实我们在开发.NET应用的过程中通过AOP这种记录日志的方式也已经是标配），加深上述AOP的几个核心概念的理解：

```
/*
 * 服务日志切面，主要记录接口日志及耗时
 */
@Aspect
@Component
public class SvcLogAspect {

    @Pointcut("@annotation(org.springframework.web.bind.annotation.RequestMapping)")
    public void requestMapping() {
    }

    @Pointcut("execution(* com.power.demo.controller.*Controller.*(..))")
    public void methodPointCut() {
    }

    @Around("requestMapping() && methodPointCut()")
    public Object around(ProceedingJoinPoint pjd) throws Throwable {

        System.out.println("Spring AOP方式记录服务日志");

        Object response = null;//定义返回信息

        BaseApiRequest baseApiRequest = null;//请求基类

        int index = 0;

        Signature curSignature = pjd.getSignature();

        String className = curSignature.getClass().getName();//类名

        String methodName = curSignature.getName();//方法名

        Logger logger = LoggerFactory.getLogger(className);//日志

        StopWatch watch = DateTimeUtil.StartNew();//用于统计调用耗时

        //获取方法参数
        Object[] reqParamArr = pjd.getArgs();
        StringBuffer sb = new StringBuffer();
        //获取请求参数集合并进行遍历拼接
        for (Object reqParam : reqParamArr) {
            if (reqParam == null) {
                index++;
                continue;
            }
            try {
                sb.append(SerializeUtil.Serialize(reqParam));
            }
            //获取继承自BaseApiRequest的请求实体
            if (baseApiRequest == null && reqParam instanceof BaseApiRequest) {

```

```

        index++;
        baseApiRequest = (BaseApiRequest) reqParam;
    }

} catch (Exception e) {
    sb.append(reqParam.toString());
}
sb.append(",");
}

String strParam = sb.toString();
if (strParam.length() > 0) {
    strParam = strParam.substring(0, strParam.length() - 1);
}

//记录请求
logger.info(String.format("【%s】类的【%s】方法，请求参数：%s", className, methodName, strParam));

response = pjd.proceed(); //执行服务方法

watch.stop();

//记录应答
logger.info(String.format("【%s】类的【%s】方法，应答参数：%s", className, methodName, SerializeUtil.Serialize(response)));

//获取执行完的时间
logger.info(String.format("接口【%s】总耗时(毫秒)：%s", methodName, watch.getTotalTimeMillis()));

//标准请求-应答模型

if (baseApiRequest == null) {

    return response;
}

if ((response != null && response instanceof BaseApiResponse) == false) {

    return response;
}

System.out.println("Spring AOP方式记录标准请求-应答模型服务日志");

Object request = reqParamArr[index];

BaseApiResponse bizResp = (BaseApiResponse) response;
//记录日志
String msg = String.format("请求：%s=====应答：%s=====总耗时(毫秒)：%s", SerializeUtil.Serialize(request),
    SerializeUtil.Serialize(response), watch.getTotalTimeMillis());

if (bizResp.getIsOK() == true) {
    logger.info(msg);
} else {
    logger.error(msg); //记录错误日志
}

return response;
}
}

```

标准的请求-应答模型，我们都会定义请求基类和应答基类，本文示例给到的是BaseApiRequest和BaseApiResponse，搜集日志时，可以对错误日志加以区分特殊处理。

注意上述代码中的@Around环绕通知，参数类型是ProceedingJoinPoint，而前面第一个示例的@Before前置通知，参数类型是JoinPoint。

下面是AspectJ通知和增强的5种模式：

@Before前置通知，在目标方法执行前实施增强，请求参数JoinPoint，用来连接当前连接点的连接细节，一般包括方法名和参数值。在方法执行前进行执行方法体，不能改变方法参数，也不能改变方法执行结果。

@After 后置通知，请求参数JoinPoint，在目标方法执行之后，无论是否发生异常，都进行执行的通知。在后置通知中，不能访问目标方法的执行结果(因为有可能发生异常)，不能改变方法执行结果。

@AfterReturning 返回通知，在目标方法执行后实施增强，请求参数JoinPoint，其能访问方法执行结果(因为正常执行)和方法的连接细节，但是不能改变方法执行结果。（注意和后置通知的区别）

@AfterThrowing 异常通知，在方法抛出异常后实施增强，请求参数JoinPoint，throwing属性代表方法体执行时候抛出的异常，其值一定与方法中Exception的值需要一致。

@Around 环绕通知，请求参数ProceedingJoinPoint，环绕通知类似于动态代理的全过程，ProceedingJoinPoint类型的参数可以决定是否执行目标方法，而且环绕通知必须有返回值，返回值即为目标方法的返回值。

参考：

<<Spring实战>>

<https://github.com/cglib/cglib>

<https://www.eclipse.org/aspectj/>

<https://blog.csdn.net/danchu/article/details/70146985>

<https://blog.csdn.net/danchu/article/details/70238002>

<https://www.cnblogs.com/zhaozihan/p/5953063.html>

推荐阅读

[1. Spring Boot 实现过滤器、拦截器与切片](#)

[2. 如何设计一个安全的对外接口](#)

[3. 互联网公司的抗疫情行动！](#)

[4. 如何获取靠谱的新型冠状病毒疫情](#)



微信搜一搜

Java后端

[阅读原文](#)

Spring Boot 全局异常处理整理

嘟嘟MD Java后端 2019-11-05

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 嘟嘟MD

来源 | tengj.top/2018/05/16/springboot13

前言

今天来一起学习一下Spring Boot中的异常处理，在日常web开发中发生了异常，往往是需要通过一个统一的异常处理来保证客户端能够收到友好的提示。

正文

本篇要点如下：

介绍Spring Boot默认的异常处理机制

如何自定义错误页面

通过@ControllerAdvice注解来处理异常

介绍Spring Boot默认的异常处理机制

默认情况下，Spring Boot为两种情况提供了不同的响应方式。

一种是浏览器客户端请求一个不存在的页面或服务端处理发生异常时，一般情况下浏览器默认发送的请求头中Accept: text/html，所以Spring Boot默认会响应一个html文档内容，称作“Whitelabel Error Page”

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Mar 21 15:46:43 CST 2018

There was an unexpected error (type=Not Found, status=404).

No message available

另一种是使用Postman等调试工具发送请求一个不存在的url或服务端处理发生异常时，Spring Boot会返回类似如下的Json格式字符串信息

```
{  
  "timestamp": "2018-05-12T06:11:45.209+0000",  
  "status": 404,  
  "error": "Not Found",  
  "message": "No message available",  
  "path": "/index.html"  
}
```

原理也很简单，Spring Boot 默认提供了程序出错的结果映射路径/error。这个/error请求会在BasicErrorController中处理，其内部是通过判断请求头中的Accept的内容是否为text/html来区分请求是来自客户端浏览器（浏览器通常默认自动发送请求头内

容Accept:text/html) 还是客户端接口的调用，以此来决定返回页面视图还是 JSON 消息内容。

相关BasicErrorController中代码如下：

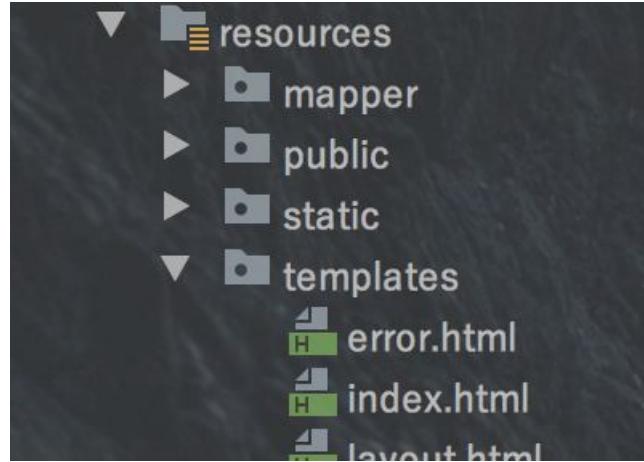
```
@RequestMapping(produces = "text/html") 浏览器请求
public ModelAndView errorHtml(HttpServletRequest request,
    HttpServletResponse response) {
    HttpStatus status = getStatus(request);
    Map<String, Object> model = Collections.unmodifiableMap(getErrorAttributes(
        request, isIncludeStackTrace(request, MediaType.TEXT_HTML)));
    response.setStatus(status.value());
    ModelAndView modelAndView = resolveErrorView(request, response, status, model);
    return (modelAndView == null ? new ModelAndView("error", model) : modelAndView);
}

@RequestMapping
@ResponseBody 客户端请求
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
    Map<String, Object> body = getErrorAttributes(request,
        isIncludeStackTrace(request, MediaType.ALL));
    HttpStatus status = getStatus(request);
    return new ResponseEntity<Map<String, Object>>(body, status);
}
```

如何自定义错误页面

好了，了解完Spring Boot默认的错误机制后，我们来点有意思的，浏览器端访问的话，任何错误Spring Boot返回的都是一个 Whitelabel Error Page 的错误页面，这个很不友好，所以我们可以自定义下错误页面。

1、先从最简单的开始，直接在 /resources/templates 下面创建error.html就可以覆盖默认的 Whitelabel Error Page 的错误页面，我项目用的是thymeleaf模板，对应的error.html代码如下：



```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    动态error错误页面
    <p th:text="${error}"></p>
    <p th:text="${status}"></p>
    <p th:text="${message}"></p>
</body>
</html>
```

这样运行的时候，请求一个不存在的页面或服务端处理发生异常时，展示的自定义错误界面如下：

动态error错误页面

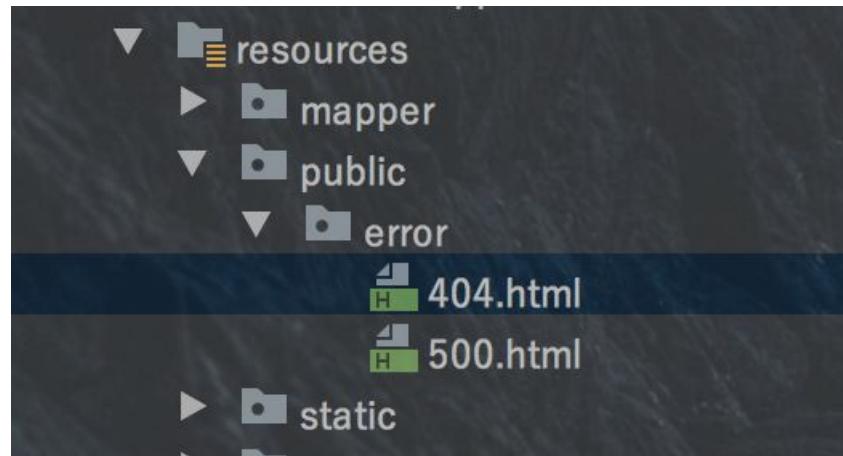
Not Found

404

No message available

2、此外，如果你想更精细一点，根据不同的状态码返回不同的视图页面，也就是对应的404，500等页面，这里分两种，错误页面可以是静态HTML（即，添加到任何静态资源文件夹下），也可以使用模板构建，文件的名称应该是确切的状态码。

如果只是静态HTML页面，不带错误信息的，在resources/public/下面创建error目录，在error目录下面创建对应的状态码html即可，例如，要将404映射到静态HTML文件，您的文件夹结构如下所示：



静态404.html简单页面如下：

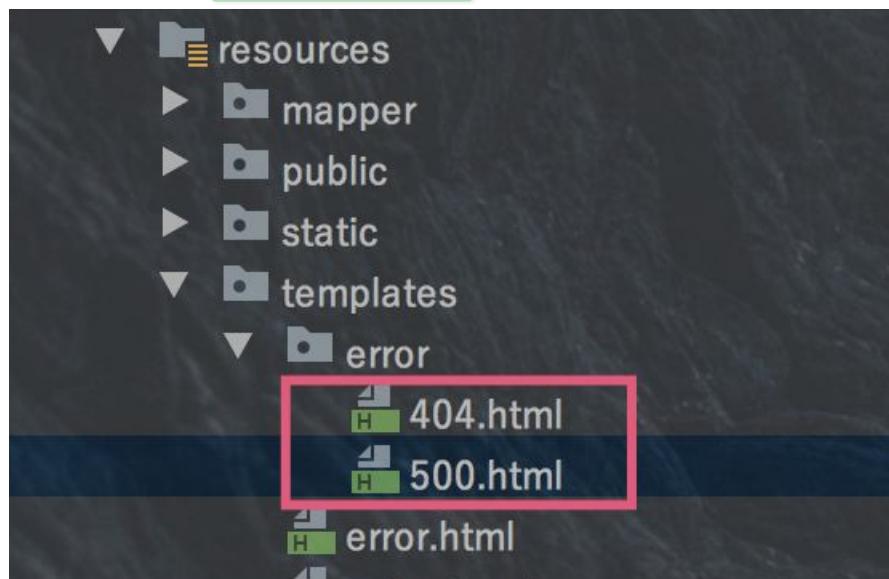
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
  静态404错误页面
</body>
</html>
```

这样访问一个错误路径的时候，就会显示 静态404错误页面 错误页面

静态404错误页面

注：这时候如果存在上面第一种介绍的error.html页面，则状态码错误页面将覆盖error.html，具体状态码错误页面优先级比较高。

如果是动态模板页面，可以带上错误信息，在 resources/templates/ 下面创建error目录，在error目录下面命名即可：



这里我们模拟下500错误，控制层代码,模拟一个除0的错误：

```
@Controller
public class BaseErrorController extends AbstractController{
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @RequestMapping(value="/ex")
    @ResponseBody
    public String error(){
        int i=5/0;
        return "ex";
    }
}
```

500.html代码:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>Title</title>
</head>
<body>
    动态500错误页面
    <p th:text="${error}"></p>
    <p th:text="${status}"></p>
    <p th:text="${message}"></p>
</body>
</html>
```

这时访问 <http://localhost:8080/spring/ex> 即可看到如下错误，说明确实映射到了500.html

动态500错误页面

Internal Server Error

500

/ by zero

注:如果同时存在静态页面500.html和动态模板的500.html, 则后者覆盖前者。即 `templates/error/` 这个的优先级比 `resources/public/error` 高。

整体概括上面几种情况, 如下:

`error.html`会覆盖默认的 whitelabel Error Page 错误提示

静态错误页面优先级别比`error.html`高

动态模板错误页面优先级比静态错误页面高

3、上面介绍的只是最简单的覆盖错误页面的方式来定义, 如果对于某些错误你可能想特殊对待, 则可以这样

```
@Configuration  
public class ContainerConfig {  
    @Bean  
    public EmbeddedServletContainerCustomizer containerCustomizer(){  
        return new EmbeddedServletContainerCustomizer(){  
            @Override  
            public void customize(ConfigurableEmbeddedServletContainer container){  
                container.addErrorPages(new ErrorPage(HttpStatus.INTERNAL_SERVER_ERROR, "/error/500"));  
            }  
        };  
    }  
}
```

上面这段代码中 `HttpStatus.INTERNAL_SERVER_ERROR` 就是对应500错误码, 也就是说程序如果发生500错误, 就会将请求转发到 `/error/500` 这个映射来, 那我们只要实现一个方法是对应这个 `/error/500` 映射即可捕获这个异常做出处理

```
@RequestMapping("/error/500")  
@ResponseBody  
public String showServerError() {  
    return "server error";  
}
```

这样, 我们再请求前面提到的异常请求 `http://localhost:8080/spring/ex` 的时候, 就会被我们这个方法捕获了。

server error

这里我们就只对500做了特殊处理，并且返还的是字符串，如果想要返回视图，去掉 @ResponseBody注解，并返回对应的视图页面。如果想要对其他状态码自定义映射，在customize方法中添加即可。

上面这种方法虽然我们重写了/500映射，但是有一个问题就是无法获取错误信息，想获取错误信息的话，我们可以继承 BasicErrorController或者干脆自己实现ErrorController接口，除了用来响应/error这个错误页面请求，可以提供更多类型的错误格式等（BasicErrorController在上面介绍SpringBoot默认异常机制的时候有提到）

这里博主选择直接继承BasicErrorController，然后把上面 /error/500 映射方法添加进来即可

```
@Controller
public class MyBasicErrorController extends BasicErrorController {

    public MyBasicErrorController() {
        super(new DefaultErrorAttributes(), new ErrorProperties());
    }

    /**
     * 定义500的 ModelAndView
     * @param request
     * @param response
     * @return
     */

    @RequestMapping(produces = "text/html", value = "/500")
    public ModelAndView errorHtml500(HttpServletRequest request, HttpServletResponse response) {
        response.setStatus(HttpStatus.valueOf(request.getAttribute("status").value()));
        Map<String, Object> model = getErrorAttributes(request, isIncludeStackTrace(request, MediaType.TEXT_HTML));
        model.put("msg", "自定义错误信息");
        return new ModelAndView("error/500", model);
    }

    /**
     * 定义500的错误JSON信息
     * @param request
     * @return
     */

    @RequestMapping(value = "/500")
    @ResponseBody

    public ResponseEntity<Map<String, Object>> error500(HttpServletRequest request) {
        Map<String, Object> body = getErrorAttributes(request, isIncludeStackTrace(request, MediaType.TEXT_HTML));
        HttpStatus status = HttpStatus.valueOf(request.getAttribute("status").value());
        return new ResponseEntity<Map<String, Object>>(body, status);
    }
}
```

代码也很简单，只是实现了自定义的500错误的映射解析，分别对浏览器请求以及json请求做了回应。

BasicErrorController默认对应的@RequestMapping是 /error，固我们方法里面对应的 @RequestMapping(produces = "text/html", value = "/500") 实际上完整的映射请求是 /error/500，这就跟上面 customize 方法自定义的映射路径对上了。

errorHtml500 方法中，我返回的是模板页面，对应/templates/error/500.html，这里顺便自定义了一个msg信息，在

500.html也输出这个信息 <p th:text="\${msg}"></p>，如果输出结果有这个信息，则表示我们配置正确了。

再次访问请求http://localhost:8080/spring/ex，结果如下



Tips：大家可以关注微信公众号：Java后端，获取更多推送。

通过@ControllerAdvice注解来处理异常

Spring Boot提供的ErrorController是一种全局性的容错机制。此外，你还可以用@ControllerAdvice注解和@ExceptionHandler注解实现对指定异常的特殊处理。

这里介绍两种情况：

局部异常处理 @Controller + @ExceptionHandler

全局异常处理 @ControllerAdvice + @ExceptionHandler

局部异常处理 @Controller + @ExceptionHandler

局部异常主要用到的是@ExceptionHandler注解，此注解注解到类的方法上，当此注解里定义的异常抛出时，此方法会被执行。如果@ExceptionHandler所在的类是@Controller，则此方法只作用在此类。如果@ExceptionHandler所在的类带有@ControllerAdvice注解，则此方法会作用在全局。

该注解用于标注处理方法处理那些特定的异常。被该注解标注的方法可以有以下任意顺序的参数类型：

Throwable、Exception 等异常对象；

ServletRequest、HttpServletRequest、ServletResponse、HttpServletResponse；

HttpSession 等会话对象；

org.springframework.web.context.request.WebRequest；

java.util.Locale；

java.io.InputStream、java.io.Reader；

java.io.OutputStream、java.io.Writer；

org.springframework.ui.Model；

并且被该注解标注的方法可以有以下的返回值类型可选：

```
ModelAndView;  
org.springframework.ui.Model;  
java.util.Map;  
org.springframework.web.servlet.View;  
@ResponseBody 注解标注的任意对象；  
HttpEntity<?> or ResponseEntity<?>;  
void;
```

以上罗列的不完全，更加详细的信息可参考：Spring ExceptionHandler。

举个简单例子，这里我们对除0异常用@ExceptionHandler来捕捉。

```
@Controller  
public class BaseErrorController extends AbstractController{  
    private Logger logger = LoggerFactory.getLogger(this.getClass());  
  
    @RequestMapping(value="/ex")  
    @ResponseBody  
    public String error(){  
        int i=5/0;  
        return "ex";  
    }  
  
    //局部异常处理  
    @ExceptionHandler(Exception.class)  
    @ResponseBody  
    public String exHandler(Exception e){  
        //判断发生异常的类型是除0异常则做出响应  
        if(e instanceof ArithmeticException){  
            return "发生了除0异常";  
        }  
        //未知的异常做出响应  
        return "发生了未知异常";  
    }  
}
```

← → ⏪ ⓘ localhost:8080/spring/ex

发生了除0异常

全局异常处理 @ControllerAdvice + @ExceptionHandler

在spring 3.2中，新增了@ControllerAdvice注解，可以用于定义@ExceptionHandler、@InitBinder、@ModelAttribute，并应用到所有@RequestMapping中。

简单的说，进入Controller层的错误才会由@ControllerAdvice处理，拦截器抛出的错误以及访问错误地址的情况@ControllerAdvice处理不了，由SpringBoot默认的异常处理机制处理。

我们实际开发中，如果是要实现RESTful API，那么默认的JSON错误信息就不是我们想要的，这时候就需要统一一下JSON格式，所以需要封装一下。

```
/*
 * 返回数据
 */
public class AjaxObject extends HashMap<String, Object> {
    private static final long serialVersionUID = 1L;

    public AjaxObject() {
        put("code", 0);
    }

    public static AjaxObject error() {
        return error(HttpStatus.SC_INTERNAL_SERVER_ERROR, "未知异常，请联系管理员");
    }

    public static AjaxObject error(String msg) {
        return error(HttpStatus.SC_INTERNAL_SERVER_ERROR, msg);
    }

    public static AjaxObject error(int code, String msg) {
        AjaxObject r = new AjaxObject();
        r.put("code", code);
        r.put("msg", msg);
        return r;
    }

    public static AjaxObject ok(String msg) {
        AjaxObject r = new AjaxObject();
        r.put("msg", msg);
        return r;
    }

    public static AjaxObject ok(Map<String, Object> map) {
        AjaxObject r = new AjaxObject();
        r.putAll(map);
        return r;
    }

    public static AjaxObject ok() {
        return new AjaxObject();
    }

    public AjaxObject put(String key, Object value) {
        super.put(key, value);
        return this;
    }

    public AjaxObject data(Object value) {
        super.put("data", value);
        return this;
    }

    public static AjaxObject apiError(String msg) {
        return error(1, msg);
    }
}
```

上面这个AjaxObject就是我平时用的，如果是正确情况返回的就是：

```
{  
    code: 0,  
    msg: "获取列表成功" ,  
    data: {  
        queryList :[]  
    }  
}
```

正确默认code返回0，data里面可以是集合，也可以是对象，如果是异常情况，返回的json则是：

```
{  
    code: 500,  
    msg: "未知异常，请联系管理员"  
}
```

然后创建一个自定义的异常类：

```
public class BusinessException extends RuntimeException implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
    private String msg;  
    private int code = 500;  
  
    public BusinessException(String msg) {  
        super(msg);  
        this.msg = msg;  
    }  
  
    public BusinessException(String msg, Throwable e) {  
        super(msg, e);  
        this.msg = msg;  
    }  
  
    public BusinessException(int code, String msg) {  
        super(msg);  
        this.msg = msg;  
        this.code = code;  
    }  
  
    public BusinessException(String msg, int code, Throwable e) {  
        super(msg, e);  
        this.msg = msg;  
        this.code = code;  
    }  
  
    public String getMsg() {  
        return msg;  
    }  
  
    public void setMsg(String msg) {  
        this.msg = msg;  
    }  
  
    public int getCode() {  
        return code;  
    }  
  
    public void setCode(int code) {  
        this.code = code;  
    }  
}
```

注：spring 对于 RuntimeException 异常才会进行事务回滚

Controller中添加一个json映射，用来处理这个异常

```
@Controller  
public class BaseErrorController{  
    @RequestMapping("/json")  
    public void json(ModelMap modelMap) {  
        System.out.println(modelMap.get("author"));  
        int i=5/0;  
    }  
}
```

最后创建这个全局异常处理类：

```

/**
 * 异常处理器
 */
@RestControllerAdvice
public class BusinessExceptionHandler {
    private Logger logger = LoggerFactory.getLogger(getClass());

    /**
     * 应用到所有@RequestMapping注解方法，在其执行之前初始化数据绑定器
     * @param binder
     */
    @InitBinder
    public void initBinder(WebDataBinder binder) {
        System.out.println("请求有参数才进来");
    }

    /**
     * 把值绑定到Model中，使全局@RequestMapping可以获取到该值
     * @param model
     */
    @ModelAttribute
    public void addAttributes(Model model) {
        model.addAttribute("author", "嘟嘟MD");
    }

    @ExceptionHandler(Exception.class)
    public Object handleException(Exception e, HttpServletRequest req){
        AjaxObject r = new AjaxObject();
        //业务异常
        if(e instanceof BusinessException){
            r.put("code", ((BusinessException) e).getCode());
            r.put("msg", ((BusinessException) e).getMsg());
        }else{//系统异常
            r.put("code", "500");
            r.put("msg", "未知异常，请联系管理员");
        }

        //使用HttpServletRequest中的header检测请求是否为ajax，如果是ajax则返回json，如果为非ajax则返回view(即 ModelAndView)
        String contentTypeHeader = req.getHeader("Content-Type");
        String acceptHeader = req.getHeader("Accept");
        String xRequestedWith = req.getHeader("X-Requested-With");
        if((contentTypeHeader != null && contentTypeHeader.contains("application/json"))
                || (acceptHeader != null && acceptHeader.contains("application/json"))
                || "XMLHttpRequest".equalsIgnoreCase(xRequestedWith)) {
            return r;
        } else {
            ModelAndView modelAndView = new ModelAndView();
            modelAndView.addObject("msg", e.getMessage());
            modelAndView.addObject("url", req.getRequestURL());
            modelAndView.addObject("stackTrace", e.getStackTrace());
            modelAndView.setViewName("error");
            return modelAndView;
        }
    }
}

```

@ExceptionHandler 拦截了异常，我们可以通过该注解实现自定义异常处理。其中，@ExceptionHandler 配置的 value 指定需要拦截的异常类型，上面我配置了拦截Exception，再根据不同异常类型返回不同的相应，最后添加判断，如果是Ajax请求，则返回json,如果是非ajax则返回view，这里是返回到error.html页面。

为了展示错误的时候更友好，我封装了下error.html,不仅展示了错误，还添加了跳转百度谷歌以及StackOverFlow的按钮，如下：

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org" layout:decorator="layout">
<head>
    <title>Spring Boot管理后台</title>
    <script type="text/javascript">
</script>
</head>
<body>
<div layout:fragment="content" th:remove="tag">
<div id="navbar">
    <h1>系统异常统一处理</h1>
    <h3 th:text=""错误信息：' + ${msg}"></h3>
    <h3 th:text=""请求地址：' + ${url}"></h3>

    <h2>Debug</h2>
    <a th:href="@{'https://www.google.com/webhp?hl=zh-CN#safe=strict&hl=zh-CN&q=' + ${msg}}"
        class="btn btn-primary btn-lg" target="_blank" id="Google">Google</a>
    <a th:href="@{'https://www.baidu.com/s?wd=' + ${msg}}" class="btn btn-info btn-lg" target="_blank" id="Baidu">Baidu</a>
    <a th:href="@{'http://stackoverflow.com/search?q=' + ${msg}}"
        class="btn btn-default btn-lg" target="_blank" id="StackOverFlow">StackOverFlow</a>
    <h2>异常堆栈跟踪日志StackTrace</h2>
    <div th:each="line:${stackTrace}">
        <div th:text="${line}"></div>
    </div>
</div>
<div layout:fragment="js" th:remove="tag">
</div>
</body>
</html>
```

访问`http://localhost:8080/json`的时候,因为是浏览器发起的，返回的是error界面：

系统异常统一处理

错误信息： / by zero

请求地址： http://localhost:8080/spring/json

Debug

[Google](#)

[Baidu](#)

[StackOverFlow](#)

异常堆栈跟踪日志StackTrace

```
com.dudu.controller.BaseErrorController.json(BaseErrorController.java:31)
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
java.lang.reflect.Method.invoke(Method.java:498)
org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:205)
org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:133)
org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:97)
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:827)
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:738)
org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:85)
org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:967)
org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:901)
org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:970)
org.springframework.web.servlet.FrameworkServlet doGet(FrameworkServlet.java:861)
```

如果是ajax请求，返回的就是错误：

```
{ "msg": "未知异常，请联系管理员", "code": 500 }
```

这里我给带@ModelAttribute注解的方法通过Model设置了author值，在json映射方法中通过 ModelMwap 获取到改值。

认真的你可能发现，全局异常类我用的是@RestControllerAdvice，而不是@ControllerAdvice，因为这里返回的主要是json格式，这样可以少写一个@ResponseBody。

总结

到此，SpringBoot中对异常的使用也差不多全了，本项目中处理异常的顺序会是这样，当发送一个请求：

- 1.拦截器那边先判断是否登录，没有则返回登录页。
- 2.在进入Controller之前，譬如请求一个不存在的地址，返回404错误界面。
- 3.在执行@RequestMapping时，发现的各种错误（譬如数据库报错、请求参数格式错误/缺失/值非法等）统一由@ControllerAdvice处理，根据是否Ajax返回json或者view。

- E N D -

如果看到这里，说明你喜欢这篇文章，请[转发](#)、[点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



扫一扫上面的二维码图案，加我微信

推荐阅读

1. [12306 的架构到底有多牛逼？](#)
2. [为什么我不建议你去外包公司？](#)
3. [Java 性能优化](#)
4. [Java 开发中常用的 4 种加密方法](#)
5. [团队开发中 Git 最佳实践](#)



学 Java，请关注公众号：Java后端

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

Spring Boot 创建定时任务（配合数据库动态执行）

yizhiwazi Java后端 2019-11-07

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | yizhiwazi

来源 | jianshu.com/p/d160f2536de7

序言：创建定时任务非常简单，主要有两种创建方式：一、基于注解(@Scheduled) 二、基于接口（SchedulingConfigurer）。前者相信大家都很熟悉，但是实际使用中我们往往想从数据库中读取指定时间来动态执行定时任务，这时候基于接口的定时任务就大派用场了。

一、静态定时任务（基于注解）

基于注解来创建定时任务非常简单，只需几行代码便可完成。

@Scheduled 除了支持灵活的参数表达式cron之外，还支持简单的延时操作，例如 fixedDelay，fixedRate 填写相应的毫秒数即可。

```
@Configuration //1.主要用于标记配置类，兼备Component的效果。  
@EnableScheduling //2.开启定时任务  
public class SimpleScheduleConfig {  
    //3.添加定时任务  
    @Scheduled(cron = "0/5 * * * * ?")  
    private void configureTasks() {  
        System.out.println("执行定时任务1: " + LocalDateTime.now());  
    }  
}
```

Cron表达式参数分别表示：

秒 (0~59) 例如0/5表示每5秒

分 (0~59)

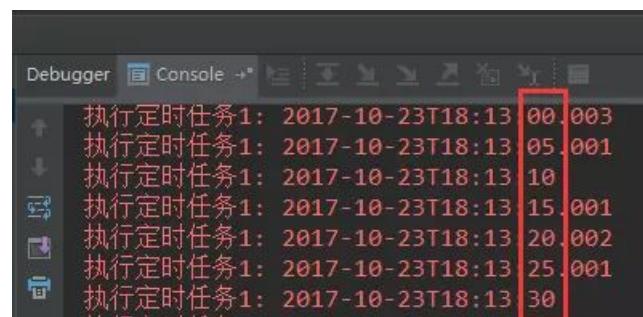
时 (0~23)

月的某天 (0~31) 需计算

月 (0~11)

周几 (可填1-7 或 SUN/MON/TUE/WED/THU/FRI/SAT)

启动应用，可以看到控制台的信息如下：



```
执行定时任务1: 2017-10-23T18:13:00.003
执行定时任务1: 2017-10-23T18:13:05.001
执行定时任务1: 2017-10-23T18:13:10
执行定时任务1: 2017-10-23T18:13:15.001
执行定时任务1: 2017-10-23T18:13:20.002
执行定时任务1: 2017-10-23T18:13:25.001
执行定时任务1: 2017-10-23T18:13:30
```

诚然，使用Scheduled 确实很方便，但缺点是当我们调整了执行周期的时候，需要重启应用才能生效，这多少有些不方便。为了

达到实时生效的效果，可以使用接口来完成定时任务。

二、动态定时任务（基于接口）

为了演示效果，这里选用 Mysql数据库 和 Mybatis 来查询和调整定时任务的执行周期，然后观察定时任务的执行情况。

Tips：关注微信公众号：Java后端，每日获取技术博文推送。

1.引入依赖

```
<!--依赖管理-->
<dependencies>
    <dependency><!--添加Web依赖-->
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency><!--添加Mybatis依赖-->
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>1.3.1</version>
    </dependency>
    <dependency><!--添加MySQL依赖-->
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency><!--添加Test依赖-->
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2.添加数据库记录

在Navicat 连接本地数据库，随便打开查询窗口，然后执行脚本内容，如下：

```
DROP DATABASE IF EXISTS `socks`;
CREATE DATABASE `socks`;
USE `SOCKS`;
DROP TABLE IF EXISTS `cron`;
CREATE TABLE `cron` (
    `cron_id` varchar(30),
    `cron` varchar(30)
);
INSERT INTO `cron` VALUES ('1','0/5 * * * * ?');
```

The screenshot shows the MySQL Workbench interface. On the left, the database tree is visible with the 'socks' database selected. Under 'socks', there is a 'cron' table highlighted with a red box. On the right, a query results window titled '* 无标题 @socks (root) - 查询' shows a single row in a table:

cron_id	cron
1	0/5 * * * *

Below the table, the text '数据库步骤:' is followed by three steps:

1. 创建数据库socks
2. 创建表cron
3. 插入数据.

然后在项目中的application.yml 添加数据源：

```
#application.yml 配置如下:  
spring:  
  datasource:  
    url: jdbc:mysql://localhost:3306/socks?useSSL=false  
    username: root  
    password: root
```

3.创建定时器

数据库准备好数据之后，我们编写定时任务，注意这里添加的是TriggerTask，目的是循环读取我们在数据库设置好的执行周期，以及执行相关定时任务的内容。具体代码如下：

```

@Configuration
@EnableScheduling
public class CompleteScheduleConfig implements SchedulingConfigurer {

    @Mapper
    public interface CronMapper {
        @Select("select cron from cron limit 1")
        String getCron();
    }

    @Autowired
    @SuppressWarnings("all")
    CronMapper cronMapper;

    /**
     * 执行定时任务.
     */
    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        taskRegistrar.addTriggerTask(
            //1.添加任务内容(Runnable)
            () -> System.out.println("执行定时任务2: " + LocalDateTime.now().toLocalTime()),
            //2.设置执行周期(Trigger)
            triggerContext -> {
                //2.1 从数据库获取执行周期
                String cron = cronMapper.getCron();
                //2.2 合法性校验.
                if (StringUtils.isEmpty(cron)) {
                    // Omitted Code ..
                }
                //2.3 返回执行周期(Date)
                return new CronTrigger(cron).nextExecutionTime(triggerContext);
            }
        );
    }
}

```

4. 动态修改执行周期

启动应用后，查看控制台，打印时间是我们预期的每5秒一次：

```

执行定时任务2: 18:39:00.001
执行定时任务2: 18:39:05
执行定时任务2: 18:39:10.001
执行定时任务2: 18:39:15.002
执行定时任务2: 18:39:20.001
执行定时任务2: 18:39:25.001
执行定时任务2: 18:39:30

```

然后打开Navicat，将执行周期修改为每1秒执行一次，如图：

cron_id	cron
1	0/1 * * * * ?

查看控制台，发现执行周期已经改变，并且不需要我们重启应用，十分方便。如图：

```
执行定时任务2: 18:39:00.001
执行定时任务2: 18:39:05
执行定时任务2: 18:39:10.001
执行定时任务2: 18:39:15.002
执行定时任务2: 18:39:20.001
执行定时任务2: 18:39:25.001
执行定时任务2: 18:39:30
执行定时任务2: 18:39:31.001
执行定时任务2: 18:39:32
执行定时任务2: 18:39:33.001
执行定时任务2: 18:39:34.001
执行定时任务2: 18:39:35
执行定时任务2: 18:39:36.001
执行定时任务2: 18:39:37.001
```

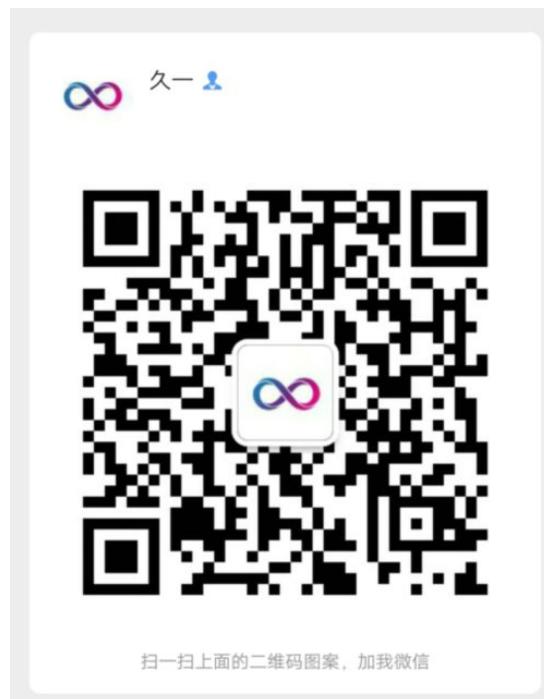
源码下载：

<https://github.com/yizhiwazi/springboot-socks/tree/master/springboot-schedule-task>

- END -

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. Spring Boot 全局异常处理整理
2. 细说 Java 主流日志工具库
3. 9 个爱不释手的 JSON 工具
4. 12306 的架构到底有多牛逼？
5. 团队开发中 Git 最佳实践



学Java, 请关注公众号: Java后端

喜欢文章, 点个在看 

[阅读原文](#)

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

Spring Boot 和 Vue 前后端分离教程(附源码)

Java后端 2019-10-26

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 梁小生0101

juejin.im/post/5c622fb5e51d457f9f2c2381

前端工具和环境：

- Node.js V10.15.0
- Vue.js V2.5.21
- yarn: V1.13.0
- IDE: VScode

后端工具和环境：

- Maven: 3.52
- jdk: 1.8
- MySql: 14.14
- IDE: IDEA
- Spring Boot: 2.0+
- Zookeeper: 3.4.13

前后端分离(服务端渲染、浏览器渲染)

实现真正的前后端解耦。

核心思想是前端html页面通过ajax调用后端的restful api接口并使用json数据进行交互。

前后端分离会为以后的大型分布式架构、弹性计算架构、微服务架构、多端化服务（多种客户端，例如：浏览器，安卓，IOS等）打下坚实的基础。

Vue.js

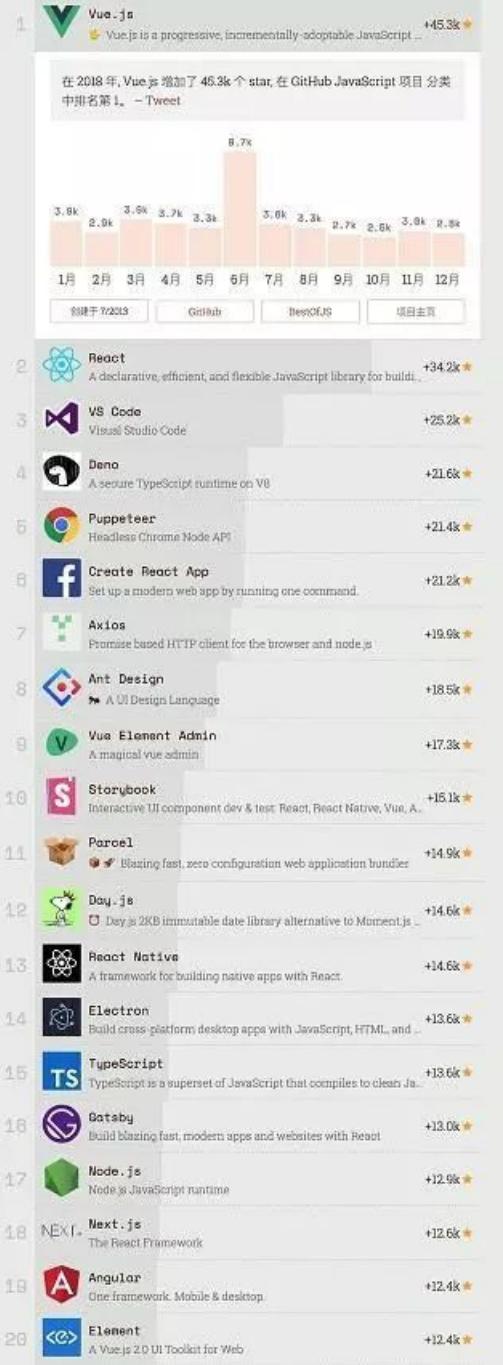
在讲Vue之前，需要大概了解下 HTML、CSS、JS是什么？

HTML是写标签的； CSS是写样式的； JS是给网页增加动态效果

Vue介绍

1,Vue是一套用于构建用户界面的渐进式框架，网址：cn.vuejs.org/

2,Vue在Github的欢迎度



3 不需要操作Dom，实现了MVVM

```

1 // jquery的操作
2 $("#test3").val("Dolly Duck");
3
4 // Vue的操作
5 MVVM，实现了双向绑定

```

4,学习成本低，文档浅显易懂

Vue 建项目

1,Vue 提供了一个官方的 CLI，为单页面应用 (SPA) 快速搭建繁杂的脚手架。基于Vue cli项目脚手架，网址：
cli.vuejs.org/zh/guide/

2,运行以下命令其一来创建一个新项目，有默认选默认即可

```
1 vue create vue-hello-world (命令行)
```

3,在创建好项目以后，运行以下命令将能看到初次项目创建的界面

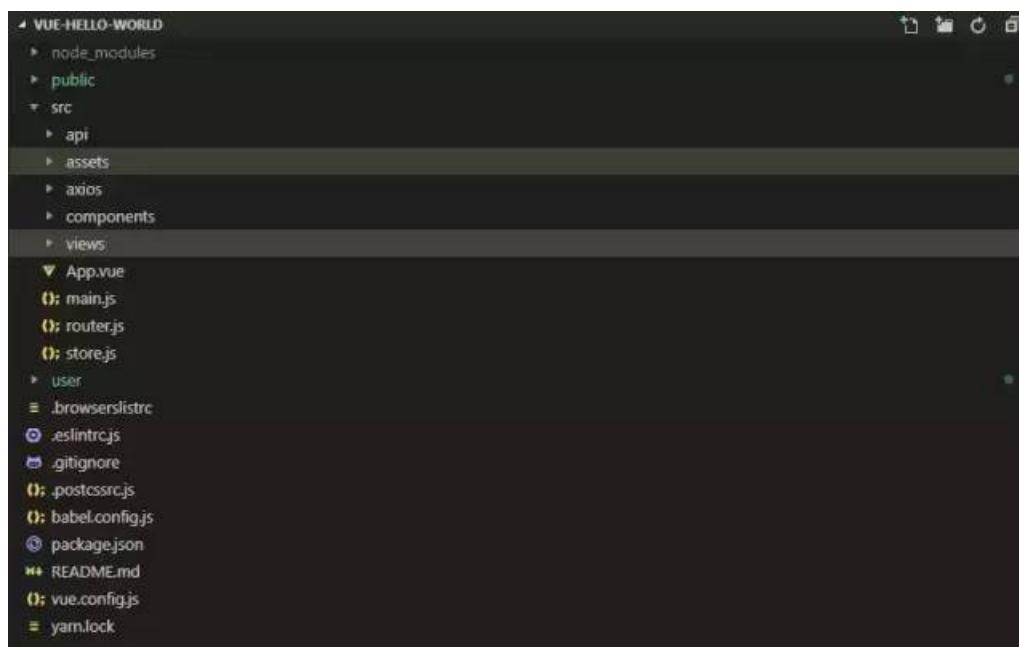
```
1 cd vue-hello-world
2 yarn serve
```

4,默认情况下，在浏览器访问 <http://localhost:8080/> 将能看到如下界面：



Vue 相关结构和生命周期介绍

1,目录结构如下图：

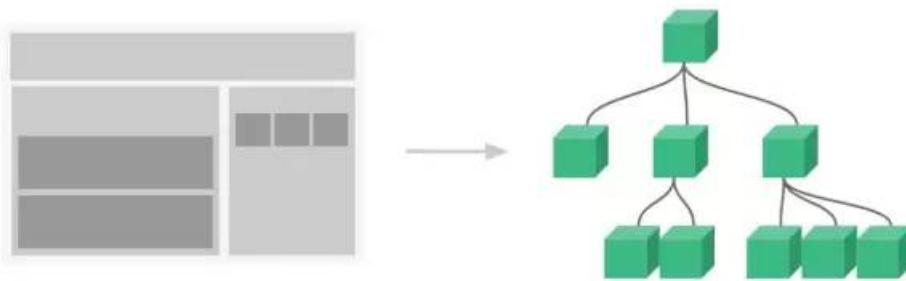


2,单个.vue文件的组成介绍

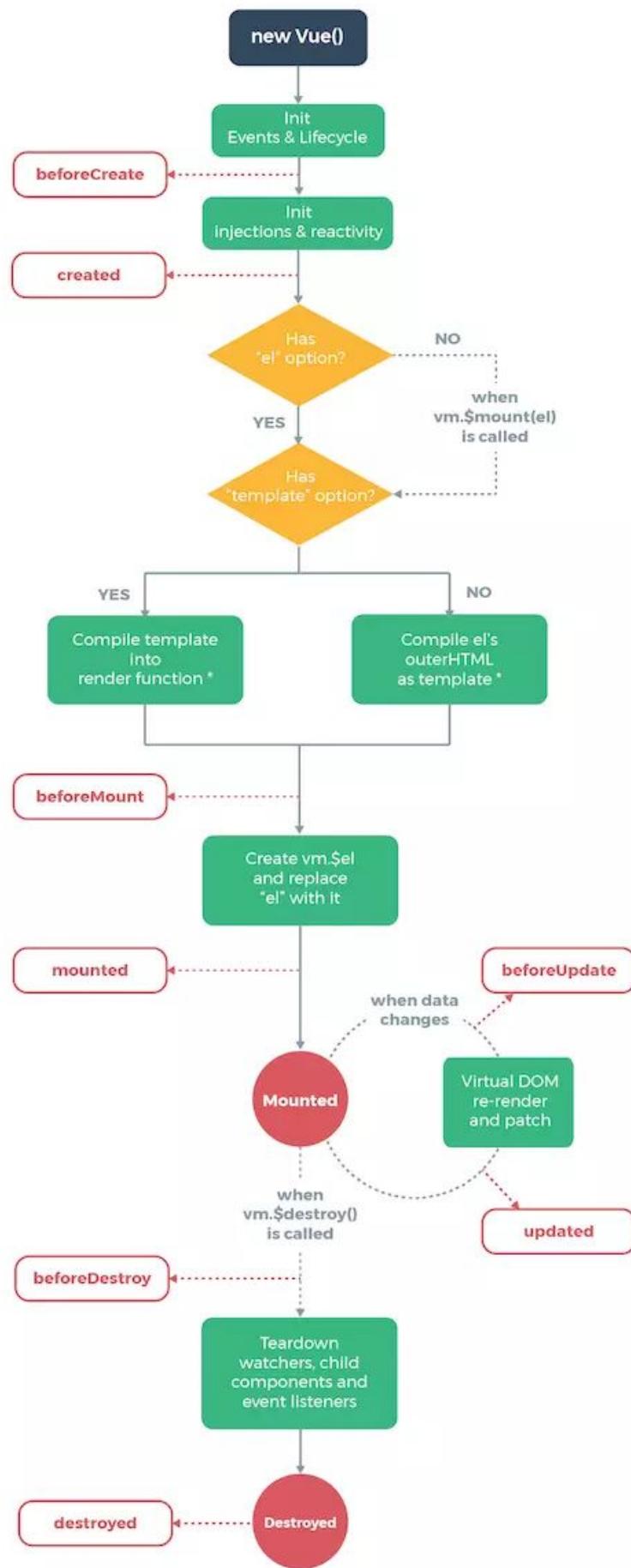
```
1 <template>
2 <!--html-->
3 </template>
4
5 <script>
6 //js
7 </script>
8
9 <style>
10 /* css style */
11 </style>
```

3,组件化应用构建

使用小型、独立和通常可复用的组件构建大型应用,一个页面如搭积木一样



4,Vue的生命周期如下图：



* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

钩子方法: 模板方法的执行结果, 该方法就叫做钩子方法, 个人理解: 影响了模板的执行, 把函数勾住了, 这个方法就是钩子函数。

钩子函数	解释
beforeCreate	实例初始化自动调用
created	实例创建后调用
beforeMount	在mount之前运行,元素已经加载,但是属性值没渲染
mounted	并挂载到实例上去之后调用该钩子
beforeDestroy	在开始销毁实例时调用
destroyed	在实例销毁后调用

Vue 常用指令

声明式渲染

```

1 <div id="app">
2   {{ message }}
3 </div>

```

```

1 data: {
2   message: 'Hello Vue!'
3 }

```

条件渲染

```

1 <div id="app-3">
2   <p v-if="seen">现在你看到我了</p>
3 </div>

```

```

1 data: {
2   seen: true
3 }

```

循环渲染

```

1 <div id="app-4">
2   <ol>
3     <li v-for="todo in todos">
4       {{ todo.text }}
5     </li>
6   </ol>

```

```
6 </div>
7
```

```
1 data: {
2   todos: [
3     { text: '学习 JavaScript' },
4     { text: '学习 Vue' },
5     { text: '整个牛项目' }
6   ]
7 }
```

监听事件

可以用 v-on 指令监听 DOM 事件，并在触发时运行一些 JavaScript 代码。

```
1 <div id="example-2">
2 <!-- `greet` 是在下面定义的方法名 -->
3 <button v-on:click="greet">Greet</button>
4 </div>
```

```
1 methods: {
2   greet: function () {
3     // `this` 在方法里指向当前 Vue 实例
4     alert('Hello ' + this.name + '!')
5   }
6 }
```

计算属性缓存 vs 方法

```
1 <div id="example">
2   <p>Original message: "{{ message }}"</p>
3   <p>Computed reversed message: "{{ reversedMessage }}"</p>
4 </div>
```

```
1 var vm = new Vue({
2   el: '#example',
3   data: {
4     message: 'Hello'
5   },
6   computed: {
7     // 计算属性的 getter
8     reversedMessage: function () {
9       // `this` 指向 vm 实例
10      return this.message.split('').reverse().join('')
11    }
12  },
13  methods: {
14    // 方法
15    reversedMessage: function () {
```

```
15     reverseMessage: function () {
16         return this.message.split('').reverse().join('')
17     }
18 }
19 )
```

数据变化，watch

```
1 var vm = new Vue({
2     el: '#demo',
3     data: {
4         firstName: 'Foo',
5         lastName: 'Bar'
6     },
7     computed: {
8         // 当两个值变化时，将会触发此函数
9         fullName: function () {
10             return this.firstName + ' ' + this.lastName
11         }
12     }
13 })
```

表单输入绑定

```
1 <input v-model="message" placeholder="edit me">
2 <p>Message is: {{ message }}</p>
```

缩写

v-bind 缩写

```
1 <!-- 完整语法 -->
2 <a v-bind:href="url">...</a>
3
4 <!-- 缩写 -->
5 <a :href="url">...</a>
```

v-on 缩写

```
1 <!-- 完整语法 -->
2 <a v-on:click="doSomething">...</a>
3
4 <!-- 缩写 -->
5 <a @click="doSomething">...</a>
```

路由

```
1 // 可提供加载
```

```
1 // 引入 VueRouter
2 const router = new VueRouter({
3   routes: [
4     {
5       path: '/user/:userId',
6       name: 'user',
7       component: User
8     }
9   ]
10 })
```

```
1 <!--html跳转-->
2 <router-link :to="{ name: 'user', params: { userId: 123 }}">User</router-link>
```

```
1 // js跳转
2 router.push({ name: 'user', params: { userId: 123 }})
```

使用 axios 访问 API

```
1 // get请求
2 axios.get('/user', {
3   params: {
4     ID: 12345
5   }
6 })
7 .then(function (response) {
8   console.log(response);
9 })
10 .catch(function (error) {
11   console.log(error);
12 });
```

```
1 // post 请求
2 axios.post('/user', {
3   firstName: 'Fred',
4   lastName: 'Flintstone'
5 })
6 .then(function (response) {
7   console.log(response);
8 })
9 .catch(function (error) {
10   console.log(error);
11 });
```

在学习完以上知识以后，将能使用Vue做出简单的页面运用

扩展：

TypeScript、Vue组件间传值、Mock、Vuex、调试、JavaScript的同步异步,作用域、ES6、部署(打包、优化、部署在静态服务器上、node中间层)、虚拟DOM、Http的get和post等。

tips: 欢迎关注微信公众号：Java后端，获取更多推送。

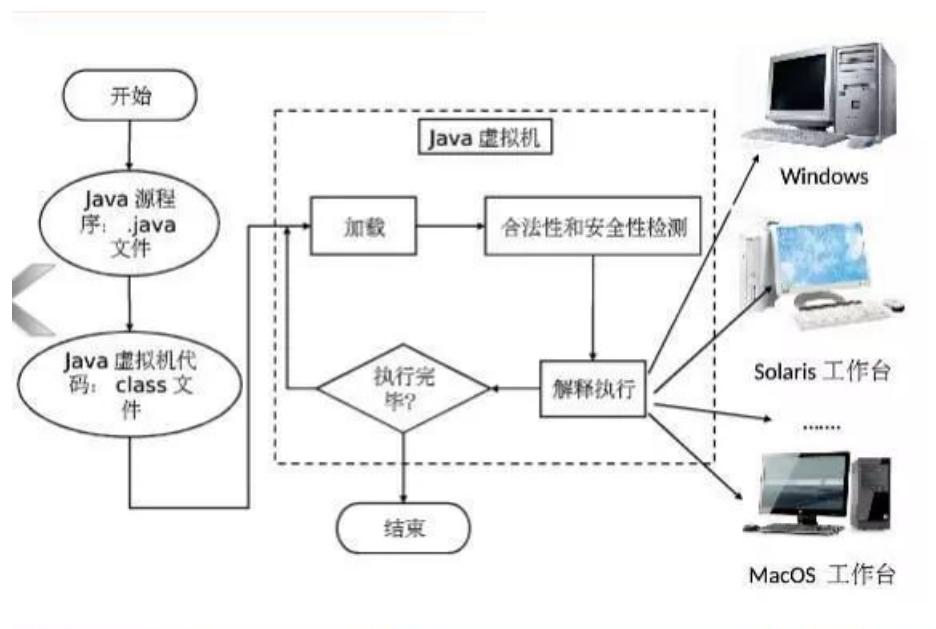
相关:

```
https://github.com/SimulatedREG/electron-vue  
https://muse-ui.org/#/zh-CN/list  
http://mpvue.com/  
https://www.iviewui.com/components/page
```

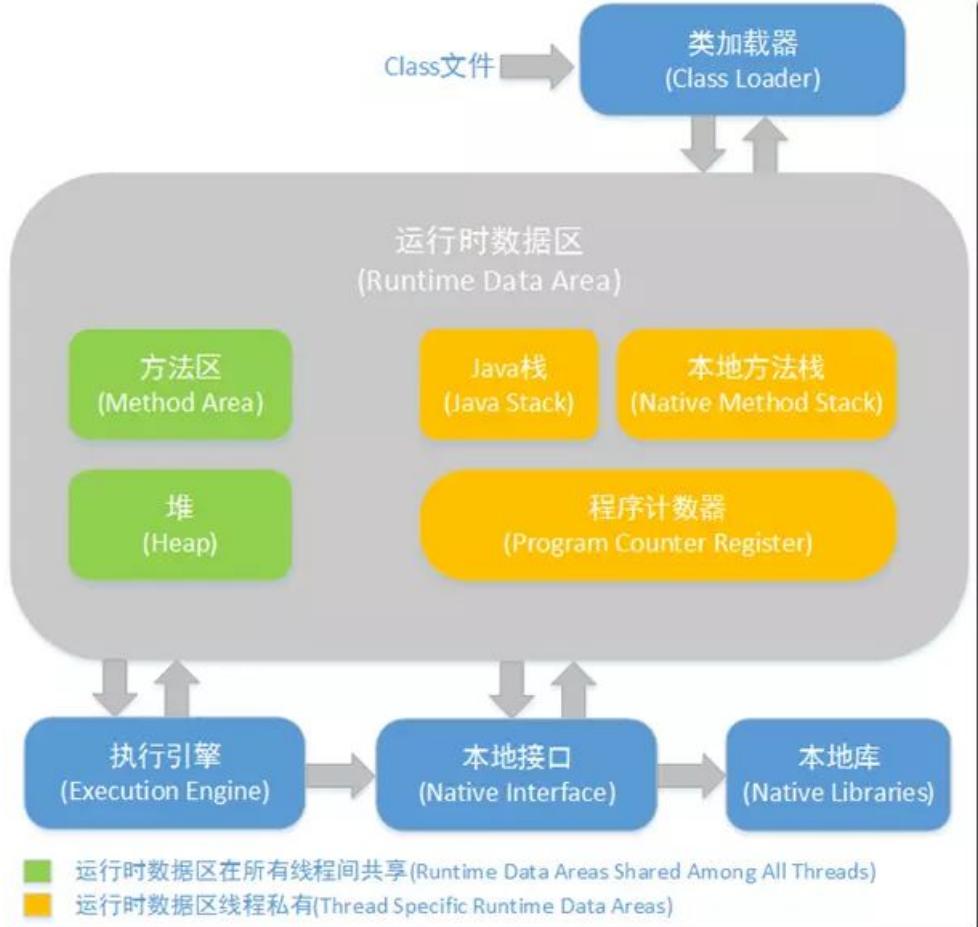
Spring Boot

在讲Spring Boot之前，需要大概了解下Java的一些相关

Java的工作原理



JVM 虚拟机



区域	解释
堆(heap)	由所有线程共享；new的对像的实例。
虚拟机栈(Stack)	每个线程拥有独立的栈；存放局部变量、对象引用。被调方法结束后，对应栈区变量等立即销毁
静态/方法区	方法区包含所有的class信息和static变量

介绍

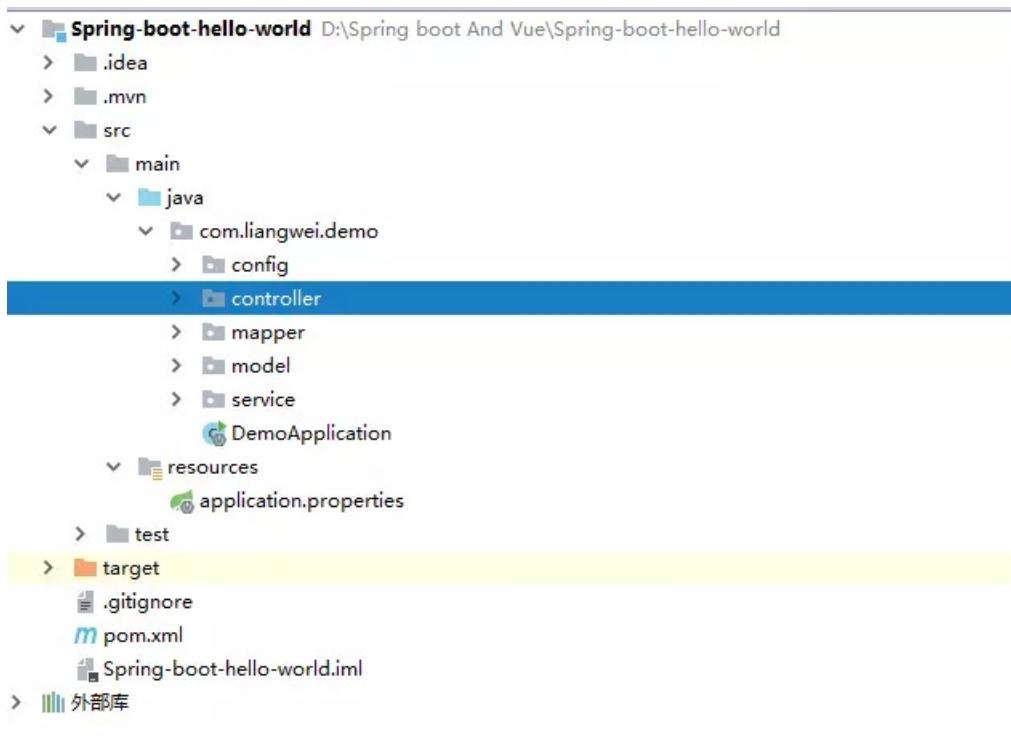
Spring Boot 是所有基于 Spring 开发的项目的。Spring Boot 的设计是为了让你尽可能快的跑起来 Spring 应用程序并且尽可能减少你的配置文件。系统学习springboot,可以在Java知音公众号回复关键字"Springboot聚合"，网罗优质教程。

使用Spring Boot开发单个RESTful服务

由于网上资源众多，就不详细编写创建步骤了。这里找了一个网上的教程，大家可以按这个步骤去创建一个项目，能用浏览器能访问就行。SpringBoot新建HelloWorld工程：

https://blog.csdn.net/small_mouse0/article/details/77800737

项目目录结构



和前端交互

1, 前端的Http请求会到controller这一层，而controller层根据相应路由信息注解会跳转到相应的类。

```
1 // 如: /api/user 的get请求将会被 UserQry() 函数处理
2
3 @RequestMapping("/api")
4 public class UserController {
5
6     @RequestMapping(value ="/user", method = RequestMethod.GET)
7     public List<User> UserQry() {
8         return userService.getUser();
9     }
10 }
```

2, 在框架经过处理以后，最终调用的是mapper层。

liang.user: 9 总记录 (大约)			
no	name	email	
454	wo我测试2	15@163.com	
1,024	vg	123@qq.com	
1,211	我是O泡	121221@qq.com	
9,527	意大利泡	45465@qq.com	
15,472	南乔峰	456789411@qq.com	
19,145	李云龙	1456244545@qq.com	
19,195	梁伟	272262983@qq.com	
23,265	胡斐	456784741@qq.com	
212,312,093	老司机	12345@sj.com	

```
1 @Select("select * from user")
2 List<User> getUser();
```

3,在执行相应的Sql以后，将会依次返回到controller层，然后在Http的返回中将会以Json串对象返回给前端的调用方。

4,前端在Http的response中拿到返回的值，然后再进行一些处理。

概念

- spring ioc容器：，主要用来管理对象和依赖，以及依赖的注入
- 依赖注入: 不用new，让Spring控制new过程
- 控制反转: 不是用new方式实例化对象,实质的控制权已经交由程序管理
- 面向切面：把一些功能抽离出来，再通过“动态”的方式掺入到业务中

Bean

bean是一个对象，由ioc容器生成的对象就是一个bean

配置VS注解

```
1 // Spring 的操作
2 package com.yiibai.common;
3
4 public class Customer
5 {
6     private Person person;
7
8     public Customer(Person person) {
9         this.person = person;
10    }
11
12    public void setPerson(Person person) {
13        this.person = person;
14    }
15    //...
16 }
17
18 package com.yiibai.common;
19
20 public class Person
21 {
22     //...
23 }
```

```
1 // Spring 的配置Bean的xml
2 <bean id="customer" class="com.yiibai.common.Customer">
3     <property name="person" ref="person" />
4 </bean>
5
6 <bean id="person" class="com.yiibai.common.Person" />
```

```
1 // Spring 的注解方式
```

```
1 // Spring 的注解方式  
2 public class Customer  
3 {  
4     @Autowired  
5     private Person person;  
6 }
```

注解

@SpringBootApplication

`@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan` 简化程序的配置。

@Configuration

注解在类上，表示这是一个IOC容器，相当于spring的配置文件，IOC容器的配置类。

@ComponentScan

如果扫描到有`@Component @Controller @Service`等这些注解的类，则把这些类注册为bean。`@Controller, @Service, @Repository`是`@Component`的细化，这三个注解比`@Component`带有更多的语义，它们分别对应了控制层、服务层、持久层的类。

@RestController

告诉Spring以JSON字符串的形式渲染结果，并直接返回给调用者。

@RequestMapping

告诉Spring这是一个用来处理请求地址映射的注解。

@Autowired

可以对类成员变量、方法及构造函数进行标注。从IoC容器中去查找，并自动装配。（去除`@Autowired`可以运行一下试试）

Mybatis的@Mapper

注解的接口生成一个实现类

跨域

浏览器从一个域名的网页去请求另一个域名的资源时，域名、端口、协议任一不同，都是跨域。

跨域资源共享(CORS) 是一种机制，它使用额外的 HTTP 头来告诉浏览器让运行的Web应用被准许访问来自不同源服务器上的指定的资源。

RESTful风格

Rest是web服务的一种架构风格，一种设计风格，URL只指定资源，以HTTP方法动词进行不同的操作。

```
1 // 非RESTful接口  
2 api/getfile.php - 获取文件信息，下载文件  
3 api/uploadfile.php - 上传创建文件
```

```
3 api/deletefile.php - 删除文件
4
5 // 只需要api/users这个接口
6 GET http://localhost:8080/api/users (查询用户)
7 POST http://localhost:8080/api/users (新增用户)
8 PUT http://localhost:8080/api/users (更新用户)
9 DELETE http://localhost:8080/api/users (删除用户)
10
```

Restful好处:

- URL具有很强可读性的，具有自描述性
- 规范化请求过程和返回结果
- 资源描述与视图的松耦合
- 可提供OpenAPI，便于第三方系统集成，提高互操作性
- 提供无状态的服务接口，降低复杂度，可提高应用的水平扩展性

扩展

JAVA的内存模型（非线程安全）、Linq、JWT、Redis、WebSocket、单点登录(SSO)、消息队列

Spring Cloud的分布式

其实在上面我们做的一个Spring Boot小的demo就是一个服务。若干个小的Spring Boot的模块，合在一起。使用一些分布式的套件，将模块集群化，让模块之间联系和管理起来，其实就是Spring Cloud的基本的微服务。

Spring Boot和 Spring Cloud的关系

基于Spring Boot 快速开发单个微服务，Spring Cloud是一个基于Spring Boot实现的开发工具；Spring Boot专注于快速、方便集成的单个微服务个体，Spring Cloud关注全局的服务治理框架；Spring Boot可以离开Spring Cloud独立使用开发项目，但是Spring Cloud离不开Spring Boot，属于依赖的关系。

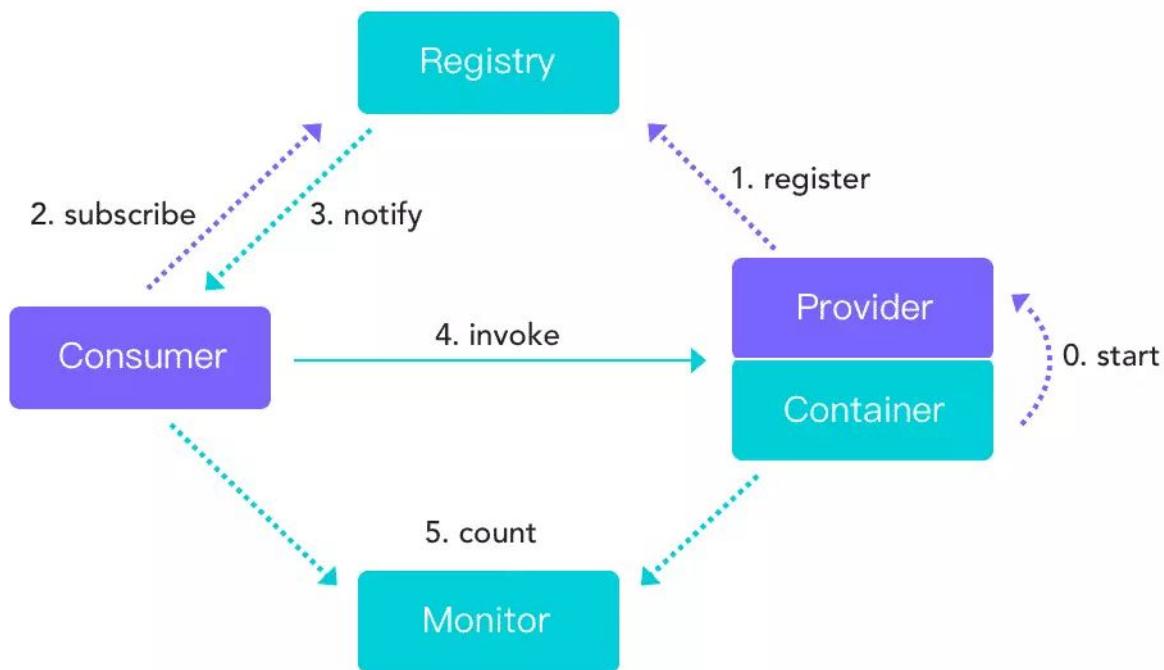
Dubbo

Dubbo 是一款高性能Java RPC框架,地址: dubbo.apache.org/zh-cn/

Dubbo的微服务的一些概念

Dubbo Architecture

..... init async → sync



- 生产者发布服务到服务注册中心中
- 消费者在服务注册中心中订阅服务
- 消费者调用已经注册的服务

Dubbo的实现单个微服务

```
1 // 定义服务接口标准
2 public interface DemoService {
3
4     String sayHello(String name);
5 }
```

```
1 // 生产者项目引用并实现
2 @Service
3 public class DemoServiceImpl implements DemoService {
4
5     @Override
6     public String sayHello(String name) {
7         return "Hello, " + name + " (from Spring Boot)";
8     }
9 }
```

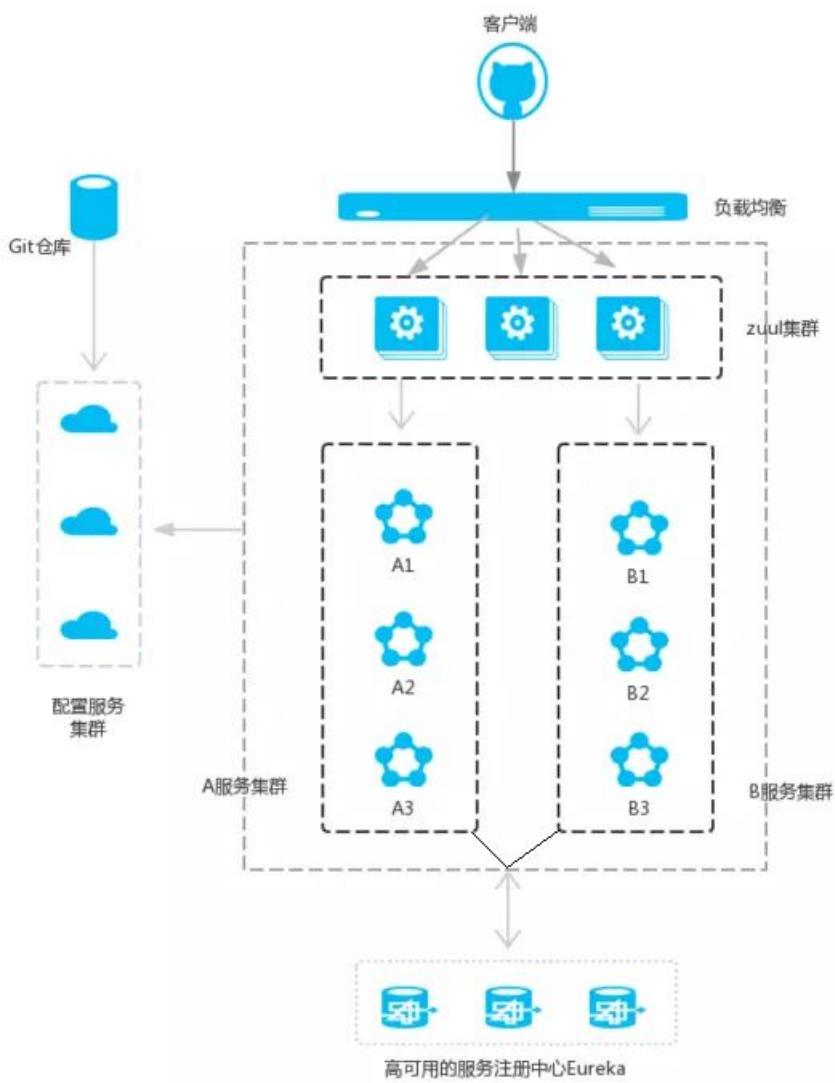
- ```
1 // 消费者引用然后调用
```

```

2 @RestController
3 public class DemoConsumerController {
4
5 @Reference
6 private DemoService demoService;
7
8 @RequestMapping("/sayHello/{name}")
9 public String sayHello(@PathVariable("name") String name) {
10 return demoService.sayHello(name);
11 }
12 }

```

## 分布式的基础套件介绍



**eureka、zookeeper** 服务注册和发现模块，服务注册在服务中心，提供给消费者使用。

**Hystrix** 断路器。为了保证其高可用，单个服务通常会集群部署。如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果。

**zuul** 路由网关。Zuul的主要功能是路由转发和过滤器。比如/api/user转发到到user服务，/api/shop转发到到shop服务

**Spring Cloud Config** 在服务数量巨多时，为了方便服务配置文件统一管理，实时更新，需要分布式配置中心组件。

**Spring Cloud Sleuth** 功能就是在分布式系统中提供追踪解决方案。

## Spring Cloud 和 Dubbo 对比

### 基础套件对比

|        | Dubbo     | Spring Cloud                 |
|--------|-----------|------------------------------|
| 服务注册中心 | Zookeeper | Spring Cloud Netflix Eureka  |
| 服务调用方式 | RPC       | REST API                     |
| 服务网关   | 无         | Spring Cloud Netflix Zuul    |
|        | 不完善       | Spring Cloud Netflix Hystrix |
| 分布式配置  | 无         | Spring Cloud Config          |
| 服务跟踪   | 无         | Spring Cloud Sleuth          |
| 消息总线   | 无         | Spring Cloud Bus             |
| 数据流    | 无         | Spring Cloud Stream          |
| 批量任务   | 无         | Spring Cloud Task            |
| .....  | .....     | .....                        |

Dubbo 只是实现了服务治理，而 Spring Cloud 子项目分别覆盖了微服务架构下的众多部件，而服务治理只是其中的一个方面。

Dubbo 提供了各种 Filter，对于上述中“无”的要素，可以通过扩展 Filter 来完善。例如：

- 分布式配置：可以使用淘宝的 diamond、百度的 disconf 来实现分布式配置管理；
- 服务跟踪：可以使用京东开源的 Hydra，或者扩展 Filter 用 Zippin 来做服务跟踪；
- 批量任务：可以使用当当开源的 Elastic-Job、tbschedule。

### 性能比较

| 线程数   | Dubbo | Spring Cloud |
|-------|-------|--------------|
| 10线程  | 2.75  | 6.52         |
| 20线程  | 4.18  | 10.03        |
| 50线程  | 10.3  | 28.14        |
| 100线程 | 20.13 | 55.23        |
| 200线程 | 42    | 110.21       |

### 服务依赖方式

**Dubbo**：服务提供方与消费方通过接口的方式依赖，因此需要为每个微服务定义了各自的 Interface 接口，并通过持续集成发布到私有仓库中，调用方应用对微服务提供的抽象接口存在强依赖关系，开发、测试、集成环境都需要严格的管理版本依赖。

**Spring Cloud**：服务提供方和服务消费方通过 JSON 方式交互，因此只需要定义好相关 JSON 字段即可，消费方和提供方无接口

依赖。

## 源代码地址

Spring Boot + Vue 代码地址：

<https://github.com/liangwei0101/Spring-Boot-And-Vue>

Dubbo-Spring-Boot 代码地址：

<https://github.com/liangwei0101/Dubbo-Spring-Boot>

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



## 推荐阅读

1. Java 开发中如何正确的踩坑
2. 当我遵循了这 16 条规范写代码
3. 理解 IntelliJ IDEA 的项目配置和 Web 部署
4. Java 开发中常用的 4 种加密方法
5. 团队开发中 Git 最佳实践



学Java, 请关注公众号: Java后端

喜欢文章, 点个在看

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

# Spring Boot 多模块项目实践（附打包方法）

yizhiwazi Java后端 2019-12-02

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | yizhiwazi

链接 | [www.jianshu.com/p/59ceea4f029d](http://www.jianshu.com/p/59ceea4f029d)

## 序言：

比起传统复杂的单体工程，使用Maven的多模块配置，可以帮助项目划分模块，鼓励重用，防止POM变得过于庞大，方便某个模块的构建，而不用每次都构建整个项目，并且使得针对某个模块的特殊控制更为方便。

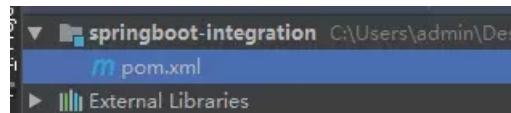
接下来，本文将重点阐述SpringBoot在Maven环境的多模块构建过程。

本项目传送门：

<https://github.com/yizhiwazi/springboot-socks/tree/master/springboot-integration>

## 一、创建聚合父工程

1.首先使用 Spring Initializr 来快速创建好一个Maven工程。然后删除无关的文件，只需保留pom.xml 文件。



2.然后在 pom.xml 里面声明该父工程包含的子模块。（其它信息就不逐一讲述了，诸如继承SpringBoot官方父工程以及统一依赖管理 请查看下面的注释说明）

```
<?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">
```

```
<!-- 基本信息 -->
<description>SpringBoot 多模块构建示例</description>
<modelVersion>4.0.0</modelVersion>
<name>springboot-integration</name>
<packaging>pom</packaging>
```

```
<!-- 项目说明：这里作为聚合工程的父工程 -->
<groupId>com.hehe</groupId>
<artifactId>springboot-integration</artifactId>
<version>1.0.0.RELEASE</version>
```

```
<!-- 继承说明：这里继承SpringBoot提供的父工程 -->
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>1.5.7.RELEASE</version>
 <relativePath/>
```

```

</parent>

<!-- 模块说明：这里声明多个子模块 -->
<modules>
 <module>mm-web</module>
 <module>mm-service</module>
 <module>mm-repo</module>
 <module>mm-entity</module>
</modules>

<!-- 版本说明：这里统一管理依赖的版本号 -->
<dependencyManagement>
 <dependencies>

 <dependency>
 <groupId>com.hehe</groupId>
 <artifactId>mm-web</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 </dependency>
 <dependency>
 <groupId>com.hehe</groupId>
 <artifactId>mm-service</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 </dependency>
 <dependency>
 <groupId>com.hehe</groupId>
 <artifactId>mm-repo</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 </dependency>
 <dependency>
 <groupId>com.hehe</groupId>
 <artifactId>mm-entity</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 </dependency>
 <dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <version>5.1.42</version>
 </dependency>
 </dependencies>
</dependencyManagement>

</project>
</pre>

```

## 二、创建子模块 (module)

注：这里是使用IDEA来创建子模块，使用Eclipse的小伙伴可通过 Spring Initializr 构建，然后复制去进去父工程根目录即可。

1. 对着父工程右键 - New - Module - > 输入 mm-web
2. 对着父工程右键 - New - Module - > 输入 mm-service
3. 对着父工程右键 - New - Module - > 输入 mm-repo
4. 对着父工程右键 - New - Module - > 输入 mm-entity

1~4 步骤完成后，分别调整它们的pom.xml 以继承上面的父工程。

例如mm-web模块的pom.xml 需要改造成这样：

```

<?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>

<!-- 基本信息 -->
<groupId>com.hehe</groupId>
<artifactId>mm-web</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>mm-web</name>

<!-- 继承本项目的父工程 -->
<parent>
 <groupId>com.hehe</groupId>
 <artifactId>springboot-integration</artifactId>
 <version>1.0.0.RELEASE</version>
</parent>

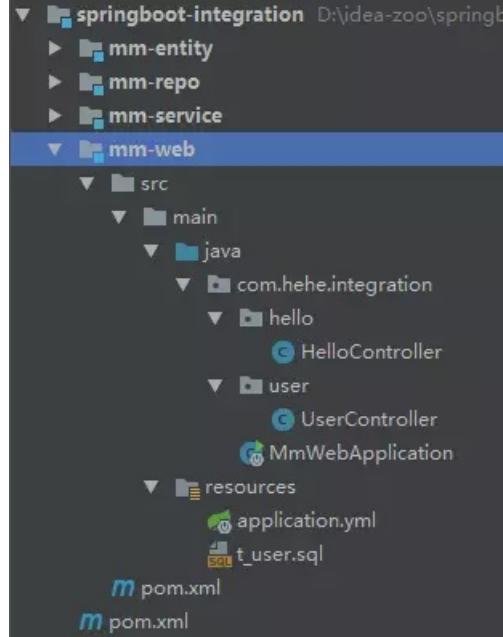
<!-- Web模块相关依赖 -->
<dependencies>
 <dependency>
 <groupId>com.hehe</groupId>
 <artifactId>mm-service</artifactId>
 </dependency>
 <dependency>
 <groupId>com.hehe</groupId>
 <artifactId>mm-entity</artifactId>
 </dependency>
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
 <dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 </dependency>
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-test</artifactId>
 <scope>test</scope>
 </dependency>
</dependencies>

</project>

```

### 三、编写子模块代码

#### 1. 控制层 (mm-web)



启动类：MmWebApplication.java (mm-web)

```
@SpringBootApplication
public class MmWebApplication {

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

控制器：UserController.java (mm-web)

```
@RestController
@RequestMapping("/user/*")
public class UserController {

 @Autowired
 UserService userService;

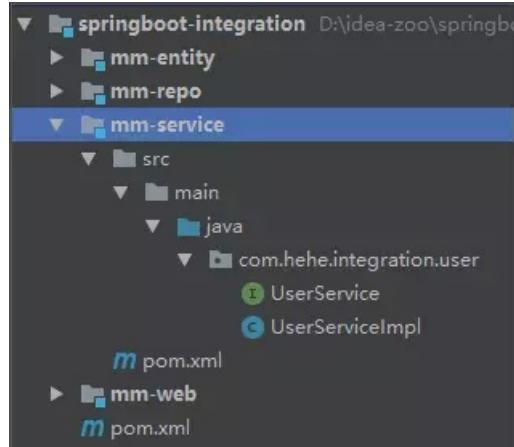
 @GetMapping("list")
 public R list() {
 try {
 return RisOk().data(userService.list());
 } catch (Exception e) {
 return R.isFail(e);
 }
 }

}
```

配置文件：application.yml (mm-web)

```
spring:
 datasource:
 url: jdbc:mysql://localhost:3306/socks?useSSL=false
 username: root
 password: root
 driver-class-name: com.mysql.jdbc.Driver
```

## 2.业务层 (mm-service)



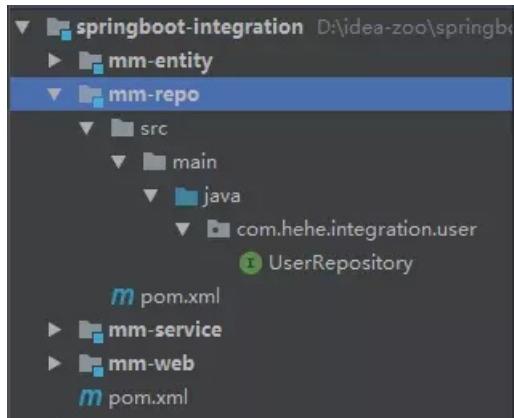
实现类：UserServiceImpl.java (mm-service)

```
@Service
public class UserServiceImpl implements UserService {

 @Autowired
 UserRepository userRepository;

 @Override
 public List<User> list() {
 return userRepository.findAll();
 }
}
```

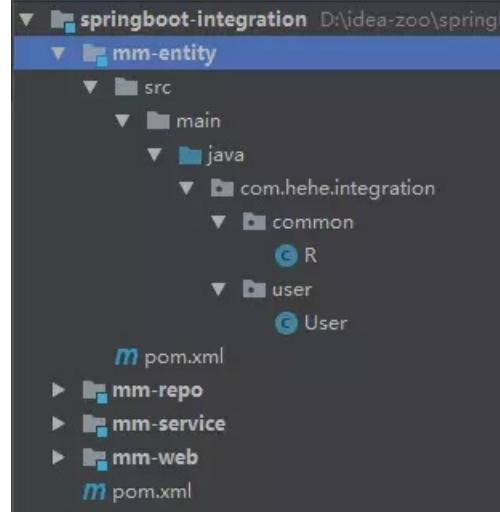
## 3.数据层 (mm-repo)



数据层代码：UserRepository.java (mm-repo)

```
public interface UserRepository extends JpaRepository<User, String> {
}
```

## 4.mm-entity (实体模型层)



R.java 作为统一返回的Bean对象

```
public class R<T> implements Serializable {

 private static final long serialVersionUID = -4577255781088498763L;
 private static final int OK = 0;
 private static final int FAIL = 1;
 private static final int UNAUTHORIZED = 2;

 private T data; //服务端数据
 private int status = OK; //状态码
 private String msg = ""; //描述信息

 //APIs
 public static R isOk(){
 return new R();
 }
 public static R isFail(){
 return new R().status(FAIL);
 }
 public static R isFail(Throwable e){
 return isFail().msg(e);
 }
 public R msg(Throwable e){
 this.setMsg(e.toString());
 return this;
 }
 public R data(T data){
 this.setData(data);
 return this;
 }
 public R status(int status){
 this.setStatus(status);
 return this;
 }

 //Constructors
 public R() {
 }

 //Getter&Setters
}
```

```

@Entity
@Table(name = "T_USER")
public class User {

 @Id
 @Column(name = "USERID")
 private String userId;
 @Column(name = "USERNAME")
 private String username;
 @Column(name = "PASSWORD")
 private String password;

 //Getter&Setters
}

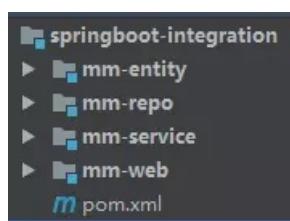
```

### 三、运行项目

为了更好的学习效果，建议先下载本项目，在IDE运行成功之后，然后再由自己手工敲一遍。

具体步骤：

1.首先下载好 springboot-socks，然后打开springboot-integration 工程。



2.安装Mysql数据库，然后创建数据库socks，并添加表t\_user，插入数据如图：

	对象	t_user @socks (root) - 表		
		开始事务	文本	筛选
		userid	username	password
		▶ 1	admin	admin
		▶ 2	jack	buzhidao

3.配置好整个项目之后，这里只需要运行mm-web模块下的MmWebApplication的启动类就可以了，如正常启动后，访问<http://localhost:8080> 可查询到用户列表信息。如下图：



### 四、运维部署（多模块打包）

## 1.添加打包插件

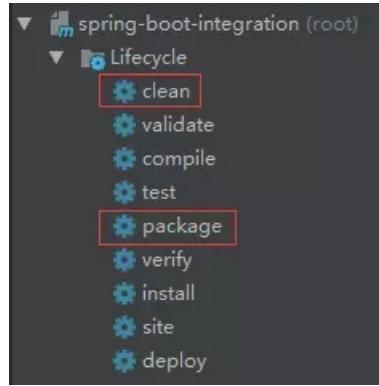
注意：多模块项目仅仅需要在启动类所在的模块添加打包插件即可！！不要在父类添加打包插件，因为那样会导致全部子模块都使用spring-boot-maven-plugin的方式来打包（例如BOOT-INF/com/hehe/xx），而mm-web模块引入mm-xx的jar需要的是裸露的类文件，即目录格式为（/com/hehe/xx）。

本案例的启动模块是mm-web，只需在它的pom.xml添加打包插件（spring-boot-maven-plugin）：

```
<!--多模块打包：只需在启动类所在模块的POM文件：指定打包插件-->
<build>
 <plugins>
 <plugin>
 <!--该插件主要用途：构建可执行的JAR-->
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-maven-plugin</artifactId>
 </plugin>
 </plugins>
</build>
```

## 2.打包工程

首先在IDE打开Maven插件，然后在聚合父工程spring-boot-integration中点击 clean，然后点击 package 进行打包。如图：



打包效果如下：

```
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] spring-boot-integration SUCCESS [0.000 s]
[INFO] mm-entity SUCCESS [1.915 s]
[INFO] mm-repo SUCCESS [0.235 s]
[INFO] mm-service SUCCESS [0.218 s]
[INFO] mm-web SUCCESS [0.891 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.798 s
[INFO] Finished at: 2017-10-18T17:17:02+08:00
[INFO] Final Memory: 35M/300M
[INFO] -----
```

打包地址默认在Target目录：

名称	修改日期
classes	2017/10/18 17:22
generated-sources 打包好的jar文件	2017/10/18 17:22
maven-archiver	2017/10/18 17:22
maven-status	2017/10/18 17:22
mm-web-0.0.1-SNAPSHOT.jar	2017/10/18 17:22
mm-web-0.0.1-SNAPSHOT.jar.original	2017/10/18 17:22

### 3.启动项目

通过命令行启动项目：

```
xx\mm-web\target>java -jar mm-web-0.0.1-SNAPSHOT.jar
```

启动效果如下：



```
C:\Windows\System32\cmd.exe - java -jar mm-web-0.0.1-SNAPSHOT.jar
Microsoft Windows [版本 10.0.15063]
(c) 2017 Microsoft Corporation. 保留所有权利。
D:\idea-zoo\springboot-socks\springboot-integration\mm-web\target>java -jar mm-web-0.0.1-SNAPSHOT.jar

:: Spring Boot :: (v1.5.6.RELEASE)

2017-10-18 17:27:29.448 INFO 8532 --- [main] com.hehe.integration.MmWebApplication : Starting MmWebApplication v0.0.1-SNAPSHOT on DESKTOP-5F0847F4 with PID 8532 (D:\idea-zoo\springboot-socks\springboot-integration\mm-web\target\mm-web-0.0.1-SNAPSHOT.jar started by admin in D:\idea-zoo\springboot-integration\mm-web\target)
2017-10-18 17:27:29.463 INFO 8532 --- [main] com.hehe.integration.MmWebApplication : No active profile set, falling back to default profiles: default
2017-10-18 17:27:29.541 INFO 8532 --- [main] atorConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@32a1bfc0: startup date [Wed Oct 18 17:27:29 CST 2017]; root of context hierarchy
2017-10-18 17:27:31.442 INFO 8532 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2017-10-18 17:27:31.458 INFO 8532 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2017-10-18 17:27:31.458 INFO 8532 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.16
2017-10-18 17:27:31.598 INFO 8532 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[] : Initializing Spring embedded WebApplicationContext
2017-10-18 17:27:31.598 INFO 8532 --- [ost-startStop-1] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2 057 ms
2017-10-18 17:27:31.786 INFO 8532 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean : Mapping servlet: 'dispatcherServlet' to [/]
2017-10-18 17:27:31.802 INFO 8532 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'characterEncodingFilter' to: [//*]
2017-10-18 17:27:31.802 INFO 8532 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [//*]
2017-10-18 17:27:31.802 INFO 8532 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'httpPutFormContentFilter' to: [//*]
2017-10-18 17:27:31.802 INFO 8532 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [//*]
2017-10-18 17:27:32.495 INFO 8532 --- [main] j.LocalContainerEntityManagerFactoryBean : Building JPA container EntityManagerFactory for persistence unit 'default'
2017-10-18 17:27:32.526 INFO 8532 --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [
 name: default
 ...
]
2017-10-18 17:27:32.620 INFO 8532 --- [main] org.hibernate.Version : HHH000412: Hibernate Core {5.0.12.Final}
2017-10-18 17:27:32.620 INFO 8532 --- [main] org.hibernate.cfg.Environment : HHH000206: hibernate.properties not found
2017-10-18 17:27:32.620 INFO 8532 --- [main] org.hibernate.cfg.Environment : HHH000021: Bytecode provider name : javassist
2017-10-18 17:27:32.682 INFO 8532 --- [main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
2017-10-18 17:27:32.943 INFO 8532 --- [main] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
2017-10-18 17:27:33.495 INFO 8532 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2017-10-18 17:27:34.152 INFO 8532 --- [main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@32a1bfc0: startup date [Wed Oct 18 17:27:29 CST 2017]; root of context hierarchy
2017-10-18 17:27:34.277 INFO 8532 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/],methods=[GET]}" onto public java.lang.String com.hehe.integration.hello.HelloController.index()
2017-10-18 17:27:34.277 INFO 8532 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/user/list],methods=[GET]}" onto public com.hehe.integration.common.R com.hehe.integration.user.UserController.list()
2017-10-18 17:27:34.293 INFO 8532 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequest)
2017-10-18 17:27:34.293 INFO 8532 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springframework.web.servlet.ModelAndView org.springframework.boot.autoconfigure.web.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest,java.lang.String)
2017-10-18 17:27:34.324 INFO 8532 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-10-18 17:27:34.324 INFO 8532 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
```

【END】

如果看到这里，说明你喜欢这篇文章，请[转发](#)、[点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



扫一扫上面的二维码图案，加我微信

### 推荐阅读

1. 你在公司项目里面看过哪些操蛋的代码？
2. 你知道 Spring Batch 吗？
3. 面试题：Lucene、Solr、ElasticSearch
4. 3 分钟带你彻底搞懂 Java 泛型背后的秘密
5. 团队开发中 Git 最佳实践



喜欢文章，点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot 如何优雅的解决 Ajax + 自定义 headers 的跨域请求

implements  
clan

Java后端 2月24日



微信搜一搜

Java后端

作者 | implements clan

链接 | [cnblogs.com/ukzq/p/10936310.html](http://cnblogs.com/ukzq/p/10936310.html)

## 1、什么是跨域

由于浏览器同源策略（同源策略，它是由Netscape提出的一个著名的安全策略。现在所有支持JavaScript的浏览器都会使用这个策略。所谓同源是指，域名，协议，端口相同。），凡是发送请求url的协议、域名、端口三者之间任意一与当前页面地址不同即为跨域。如果想要更多 Java 相关的技术博文可以本公众号「Java后端」回复「技术博文」获取。

具体可以查看下表：

Url	说明	是否允许通信
www.baidu.com/a.js	同一域下	允许通信
www.baidu.com/b.js		
www.baidu.com:9000/a.js	域相同端口不同	不允许通信（跨域）
www.baidu.com:9001/b.js		
www.baidu.com/a.js	不同域下	不允许通信（跨域）
www.wafer.com/b.js		

## 2、Spring Boot如何解决跨域问题

### 1.普通跨域请求解决方案：

①请求接口添加注解@CrossOrigin(origins = "http://127.0.0.1:8020", maxAge = 3600)

说明：origins = "http://127.0.0.1:8020" origins值为当前请求该接口的域

②通用配置（所有接口都允许跨域请求）

新增一个configuration类 或 在Application中加入CorsFilter和CorsConfiguration方法

```

@Configuration
public class CorsConfig {
 privateCorsConfiguration buildConfig() {
 CorsConfigurationcorsConfiguration= newCorsConfiguration();
 corsConfiguration.addAllowedOrigin("*");//1允许任何域名使用
 corsConfiguration.addAllowedHeader("*");//2允许任何头
 corsConfiguration.addAllowedMethod("*");//3允许任何方法(post、get等)
 returncorsConfiguration;
 }

 @Bean
 publicCorsFilter corsFilter() {
 UrlBasedCorsConfigurationSourcesource= newUrlBasedCorsConfigurationSource();
 source.registerCorsConfiguration("/*",buildConfig());//4
 return newCorsFilter(source);
 }
}

```

## 2.ajax自定义headers的跨域请求

```

$.ajax({
 type:"GET",
 url:"http://localhost:8766/main/currency/sginInState",
 dataType:"JSON",
 data:{
 uid:userId
 },
 beforeSend:function(XMLHttpRequest){
 XMLHttpRequest.setRequestHeader("Authorization",access_token);
 },
 success:function(res){
 console.log(res.code)
 }
})

```

此时请求http://localhost:8766/main/currency/sginInState接口发现OPTIONS

http://localhost:8766/main/currency/sginInState 500错误，普通跨域的解决方案已经无法解决这种问题，为什么会出现OPTIONS请求呢？

```

② ▶ OPTIONS http://localhost:8766/main/currency/sginInState?uid= 500
✖ Failed to load http://localhost:8766/main/currency/sginInState?uid=: Response for preflight does not have HTTP ok status.

```

## 原因

浏览器会在发送真正请求之前，先发送一个方法为OPTIONS的预检请求 Preflighted requests 这个请求是用来验证本次请求是否安全的，但是并不是所有请求都会发送，需要符合以下条件：

- 请求方法不是GET/HEAD/POST
- POST请求的Content-Type并非application/x-www-form-urlencoded, multipart/form-data, 或text/plain
- 请求设置了自定义的header字段

对于管理端的接口，我有对接口进行权限校验，每次请求需要在header中携带自定义的字段（token），所以浏览器会多发送一个OPTIONS请求去验证此次请求的安全性。

## 为何OPTIONS请求是500呢？

OPTIONS请求只会携带自定义的字段，并不会将相应的值带入进去，而后台校验token字段时 token为NULL，所以验证不通过，抛出了一个异常。

那么我们现在来解决这种问题：

### ① spring boot项目application.yml中添加

```
spring:
 mvc:
 dispatch-options-request:true
```

注意：这种解决方案可能在某些情况下并不能解决OPTIONS问题，原因可能是环境问题，也可能是复杂的自定义filter过滤器配置问题等。

### ②添加过滤器配置

第一步：手写RequestFilter请求过滤器配置类此类需要实现HandlerInterceptor类，HandlerInterceptor类是org.springframework.web.servlet.HandlerInterceptor下的。

具体代码实现：

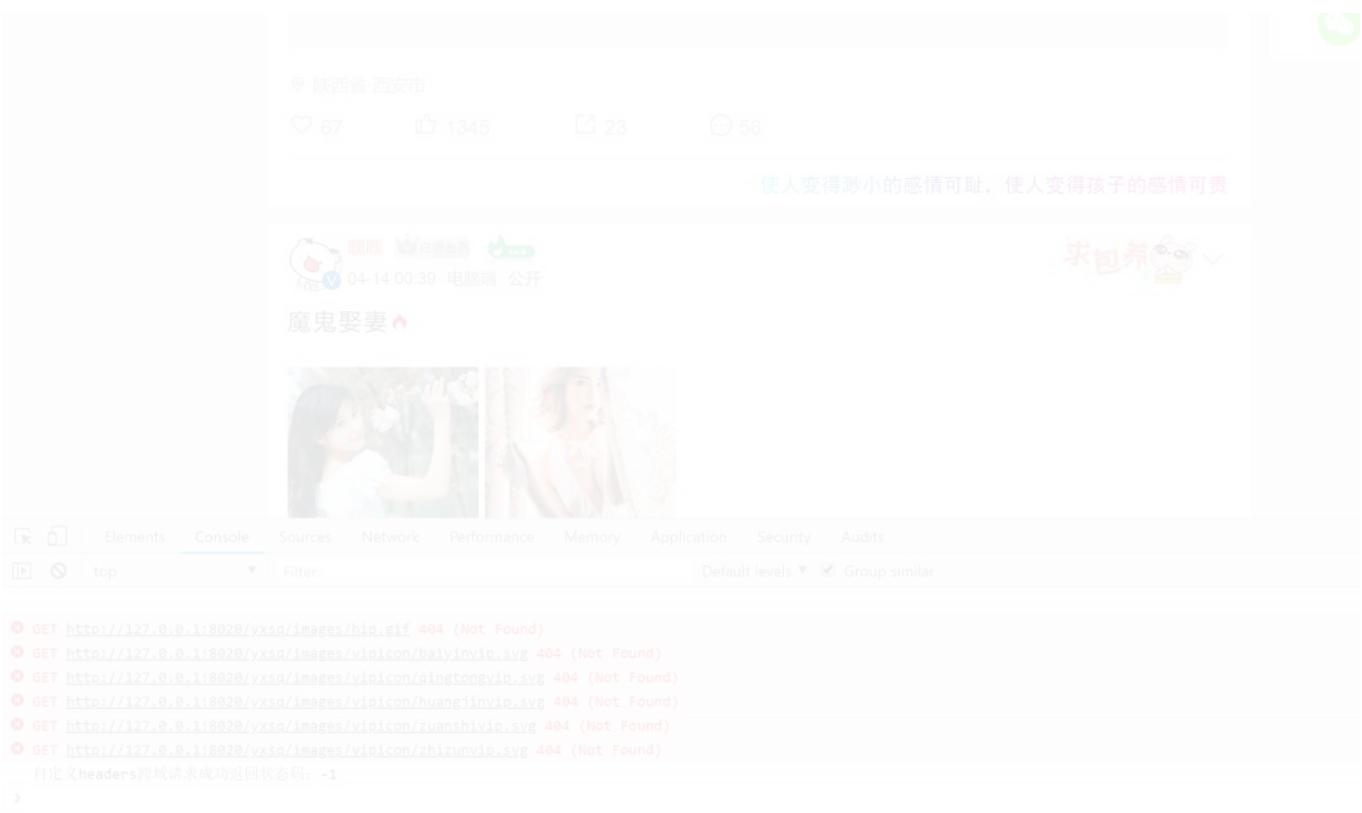
```
@Component
public class RequestFilter implements HandlerInterceptor {
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
 response.setHeader("Access-Control-Allow-Origin", "*");
 response.setHeader("Access-Control-Allow-Credentials", "true");
 response.setHeader("Access-Control-Allow-Methods", "GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS");
 response.setHeader("Access-Control-Max-Age", "86400");
 response.setHeader("Access-Control-Allow-Headers", "Authorization");
 //如果是OPTIONS请求则结束
 if (RequestMethod.OPTIONS.toString().equals(request.getMethod())) {
 response.setStatus(HttpStatus.NO_CONTENT.value());
 return false;
 }
 return true;
 }
}
```

第二步：手写MyWebConfiguration此类需要继承WebMvcConfigurerSupport。

注意：WebMvcConfigurerSupport是2.x版本以上的，1.x版本为WebMvcConfigurerAdapter。

具体代码实现：

```
@Component
public class MyWebConfiguration extends WebMvcConfigurationSupport{
 @Resource
 private RequestFilter requestFilter;
 @Override
 public void addInterceptors(InterceptorRegistry registry) {
 //跨域拦截器
 registry.addInterceptor(requestFilter).addPathPatterns("/**");
 }
}
```



此时我们就完美解决了ajax+自定义headers的跨域请求了，欢迎随时交流学习。

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓ 扫描二维码添加小编 ↓



### 推荐阅读

- [1. 10 分钟实现 Java 发送邮件功能](#)
- [2. Spring Boot 线程池的创建](#)
- [3. Spring Boot 整合 Redis](#)
- [4. 2020 年 9 大顶级 Java 框架](#)



微信搜一搜

Q Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot 实现过滤器、拦截器与切片

七印miss Java后端 1月28日



微信搜一搜

Java后端

作者 | 七印miss

链接 | [juejin.im/post/5c6901206fb9a049af6dcdf](https://juejin.im/post/5c6901206fb9a049af6dcdf)

Q: 使用过滤器、拦截器与切片实现每个请求耗时的统计，并比较三者的区别与联系

## 过滤器Filter

### 过滤器概念

Filter是J2E中来的，可以看做是Servlet的一种“加强版”，它主要用于对用户请求进行预处理和后处理，拥有一个典型的处理链。Filter也可以对用户请求生成响应，这一点与Servlet相同，但实际上很少会使用Filter向用户请求生成响应。

使用Filter完整的流程是：Filter对用户请求进行预处理，接着将请求交给Servlet进行预处理并生成响应，最后Filter再对服务器响应进行后处理。

### 过滤器作用

在JavaDoc中给出了几种过滤器的作用

- Examples that have been identified for this design are
  - 1) Authentication Filters, 即用户访问权限过滤
  - 2) Logging and Auditing Filters, 日志过滤，可以记录特殊用户的特殊请求的记录等
  - 3) Image conversion Filters
  - 4) Data compression Filters
  - 5) Encryption Filters
  - 6) Tokenizing Filters
  - 7) Filters that trigger resource access events
  - 8) XSL/T filters
  - 9) Mime-type chain Filter

对于第一条，即使用Filter作权限过滤，其可以这么实现：定义一个Filter，获取每个客户端发起的请求URL，与当前用户无权限访问的URL列表（可以是从DB中取出）作对比，起到权限过滤的作用。

### 过滤器实现方式

自定义的过滤器都必须实现javax.Servlet.Filter接口，并重写接口中定义的三个方法：

#### 1. void init(FilterConfig config)

用于完成Filter的初始化。

## 2.void destroy()

用于Filter销毁前，完成某些资源的回收。

## 3.void doFilter(ServletRequest request,ServletResponse response,FilterChain chain)

实现过滤功能，即对每个请求及响应增加的额外的预处理和后处理。,执行该方法之前，即对用户请求进行预处理；执行该方法之后，即对服务器响应进行后处理。

值得注意的是，chain.doFilter()方法执行之前为预处理阶段，该方法执行结束即代表用户的请求已经得到控制器处理。因此，如果在doFilter中忘记调用chain.doFilter()方法，则用户的请求将得不到处理。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

//必须添加注解，springmvc通过web.xml配置
@Component
public class TimeFilter implements Filter {
 private static final Logger LOG = LoggerFactory.getLogger(TimeFilter.class);

 @Override
 public void init(FilterConfig filterConfig) throws ServletException {
 LOG.info("初始化过滤器: {}", filterConfig.getFilterName());
 }

 @Override
 public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
 LOG.info("start to doFilter");
 long startTime = System.currentTimeMillis();
 chain.doFilter(request, response);
 long endTime = System.currentTimeMillis();
 LOG.info("the request of {} consumes {}ms.", getUrlFrom(request), (endTime - startTime));
 LOG.info("end to doFilter");
 }

 @Override
 public void destroy() {
 LOG.info("销毁过滤器");
 }

 private String getUrlFrom(ServletRequest servletRequest){
 if (servletRequest instanceof HttpServletRequest){
 return ((HttpServletRequest) servletRequest).getRequestURL().toString();
 }
 return "";
 }
}
```

从代码中可看出，类Filter是在javax.servlet.\*中，因此可以看出，过滤器的一个很大的局限性在于，其不能够知道当前用户的请求是被哪个控制器(Controller)处理的，因为后者是spring框架中定义的。

对于SpringMvc，可以通过在web.xml中注册过滤器。但在SpringBoot中不存在web.xml，此时如果引用的某个jar包中的过滤器，且这个过滤器在实现时没有使用@Component标识为Spring Bean，则这个过滤器将不会生效。

此时需要通过java代码去注册这个过滤器。以上面定义的TimeFilter为例，当去掉类注解@Component时，注册方式为：

```
@Configuration
public class WebConfig {
 /**
 * 注册第三方过滤器
 * 功能与spring mvc中通过配置web.xml相同
 * @return
 */
 @Bean
 public FilterRegistrationBean thirdFilter(){
 ThirdPartFilter thirdPartFilter = new ThirdPartFilter();
 FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean();

 filterRegistrationBean.setFilter(thirdPartFilter);
 List<String> urls = new ArrayList<>();
 //匹配所有请求路径
 urls.add("/*");
 filterRegistrationBean.setUrlPatterns(urls);

 return filterRegistrationBean;
 }
}
```

相比使用@Component注解，这种配置方式有个优点，即可以自由配置拦截的URL。

## 拦截器Interceptor

### 拦截器概念

拦截器，在AOP(Aspect-Oriented Programming)中用于在某个方法或字段被访问之前，进行拦截，然后在之前或之后加入某些操作。拦截是AOP的一种实现策略。

### 拦截器作用

- **日志记录：**记录请求信息的日志，以便进行信息监控、信息统计、计算PV（Page View）等
- **权限检查：**如登录检测，进入处理器检测是否登录
- **性能监控：**通过拦截器在进入处理器之前记录开始时间，在处理完后记录结束时间，从而得到该请求的处理时间。（反向代理，如apache也可以自动记录）；
- **通用行为：**读取cookie得到用户信息并将用户对象放入请求，从而方便后续流程使用，还有如提取Locale、Theme信息等，只要是多个处理器都需要的即可使用拦截器实现。

### 拦截器实现

通过实现HandlerInterceptor接口，并重写该接口的三个方法来实现拦截器的自定义：

1. preHandler(HttpServletRequest request, HttpServletResponse response, Object handler)

方法将在请求处理之前进行调用。SpringMVC中的Interceptor同Filter一样都是链式调用。每个Interceptor的调用会依据它的声明顺序依次执行，而且最先执行的都是Interceptor中的preHandle方法，所以可以在这个方法中进行一些前置初始化操作或者是对当前请求的一个预处理，也可以在这个方法中进行一些判断来决定请求是否要继续进行下去。

该方法的返回值是布尔值Boolean类型的，当它返回为false时，表示请求结束，后续的Interceptor和Controller都不会再执行；当返回值为true时就会继续调用下一个Interceptor的preHandle方法，如果已经是最后一个Interceptor的时候就会是调用当前请求的Controller方法。

## 2.postHandler(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)

在当前请求进行处理之后，也就是Controller方法调用之后执行，但是它会在DispatcherServlet进行视图返回渲染之前被调用，所以我们可以在的方法中对Controller处理之后的 ModelAndView 对象进行操作。

## 3.afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handle, Exception ex)

该方法也是需要当前对应的Interceptor的preHandle方法的返回值为true时才会执行。顾名思义，该方法将在整个请求结束之后，也就是在DispatcherServlet渲染了对应的视图之后执行。这个方法的主要作用是用于进行资源清理工作的。

```
@Component
public class TimeInterceptor implements HandlerInterceptor {
 private static final Logger LOG = LoggerFactory.getLogger(TimeInterceptor.class);
 @Override
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
 LOG.info("在请求处理之前进行调用（Controller方法调用之前）");
 request.setAttribute("startTime", System.currentTimeMillis());
 HandlerMethod handlerMethod = (HandlerMethod) handler;
 LOG.info("controller object is {}", handlerMethod.getBean().getClass().getName());
 LOG.info("controller method is {}", handlerMethod.getMethod());

 // 需要返回true，否则请求不会被控制器处理
 return true;
 }

 @Override
 public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
 LOG.info("请求处理之后进行调用，但是在视图被渲染之前（Controller方法调用之后），如果异常发生，则该方法不会被调用");
 }

 @Override
 public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
 LOG.info("在整个请求结束之后被调用，也就是在DispatcherServlet 渲染了对应的视图之后执行（主要是用于进行资源清理工作）");
 long startTime = (long) request.getAttribute("startTime");
 LOG.info("time consume is {}", System.currentTimeMillis() - startTime);
 }
}
```

与过滤器不同的是，拦截器使用@Component修饰后，在SpringBoot中还需要通过实现WebMvcConfigurer手动注册：

```
//java配置类
@Configuration
public class WebConfig implements WebMvcConfigurer {
 @Autowired
 private TimeInterceptor timeInterceptor;

 @Override
 public void addInterceptors(InterceptorRegistry registry){
 registry.addInterceptor(timeInterceptor);
 }
}
```

如果是在SpringMVC中，则需要通过xml文件配置 `<mvc:interceptors>` 节点信息。

## 切片Aspect

### 切片概述

相比过滤器，拦截器能够知道用户发出的请求最终被哪个控制器处理，但是拦截器还有一个明显的不足，即不能够获取request的参数以及控制器处理之后的response。所以就有了切片的用武之地了。

### 切片实现

切片的实现需要注意`@Aspect`,`@Component`以及`@Around`这三个注解的使用，详细查看官方文档：

<https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#aop>

```
@Aspect
@Component
public class TimeAspect {
 private static final Logger LOG = LoggerFactory.getLogger(TimeAspect.class);

 @Around("execution(* me.iftight.controller.*.*(..))")
 public Object handleControllerMethod(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{
 LOG.info("切片开始。。。");
 long startTime = System.currentTimeMillis();

 // 获取请求入参
 Object[] args = proceedingJoinPoint.getArgs();
 Arrays.stream(args).forEach(arg -> LOG.info("arg is {}", arg));

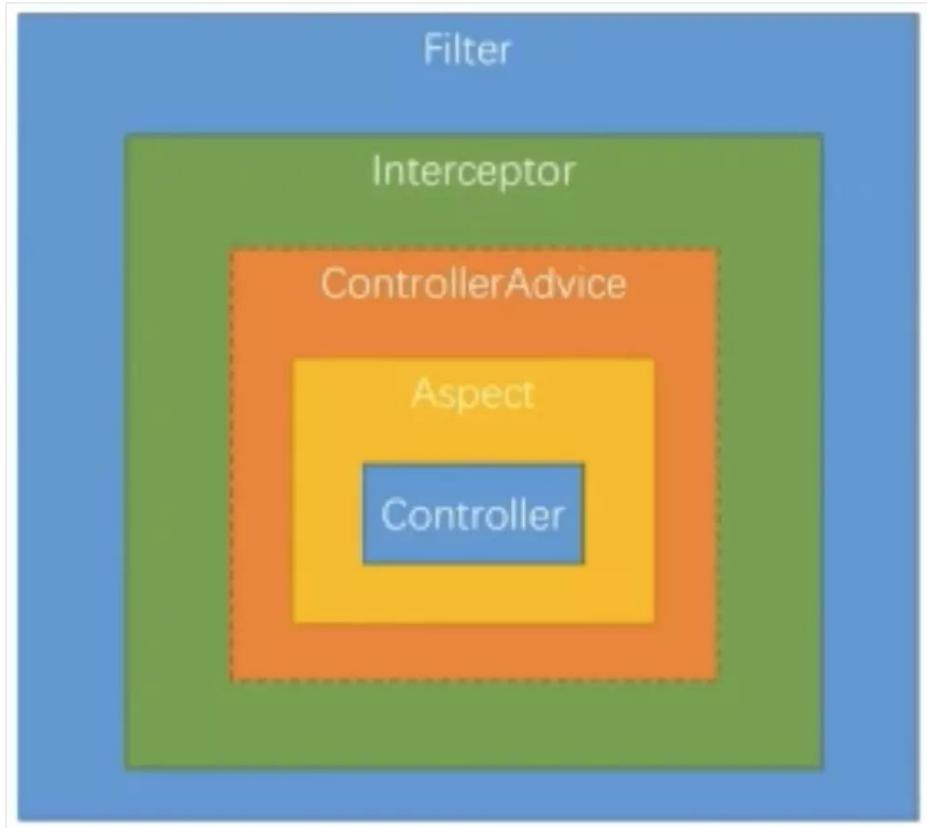
 // 获取相应
 Object response = proceedingJoinPoint.proceed();

 long endTime = System.currentTimeMillis();
 LOG.info("请求:{}, 耗时{}ms", proceedingJoinPoint.getSignature(), (endTime - startTime));
 LOG.info("切片结束。。。");
 return null;
 }
}
```

### 过滤器、拦截器以及切片的调用顺序

如下图，展示了三者的调用顺序Filter->Intercepto->Aspect->Controller。相反的是，当Controller抛出的异常的处理顺序则是从内到外的。因此我们总是定义一个注解`@ControllerAdvice`去统一处理控制器抛出的异常。

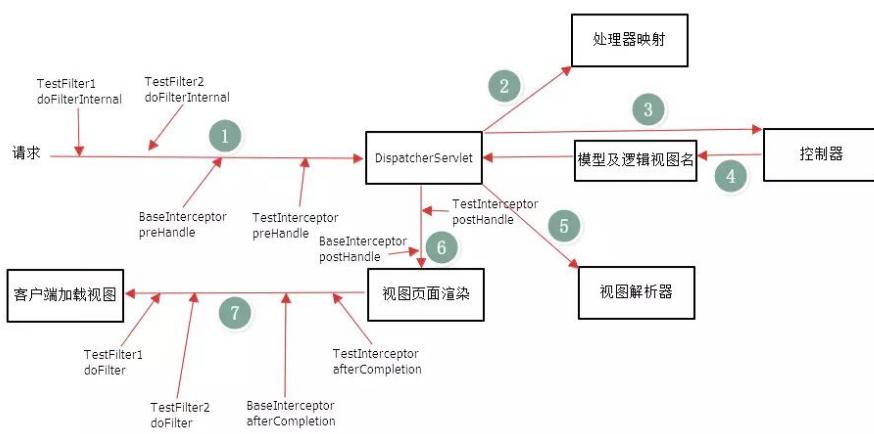
如果一旦异常被@ControllerAdvice处理了，则调用拦截器的afterCompletion方法的参数Exception ex就为空了。



实际执行的调用栈也说明了这一点：

```
o.s.w.s.DispatcherServlet
o.s.w.s.DispatcherServlet
m.i.c.f.ThirdPartFilter
m.i.c.f.TimeFilter
m.i.c.i.TimeInterceptor
m.i.c.i.TimeInterceptor
m.i.c.i.TimeInterceptor
m.i.c.a.TimeAspect
m.i.c.a.TimeAspect
m.i.c.a.TimeAspect
m.i.c.TestController
m.i.c.a.TimeAspect
m.i.c.a.TimeAspect
m.i.c.i.TimeInterceptor
m.i.c.i.TimeInterceptor
m.i.c.i.TimeInterceptor
m.i.c.f.TimeFilter
m.i.c.f.TimeFilter
m.i.c.f.ThirdPartFilter
```

而对于过滤器和拦截器详细的调用顺序如下图：



## 过滤器和拦截器的区别

最后有必要再说说过滤器和拦截器二者之间的区别：

	Filter	Interceptor
实现方式	过滤器是基于函数回调	基于Java的反射机制的
规范	Servlet规范	Spring规范
作用范围	对几乎所有的请求起作用	只对action请求起作用

除此之外，相比过滤器，拦截器能够“看到”用户的请求具体是被Spring框架的哪个控制器所处理。

## 参考

<https://blog.csdn.net/xiaodanjava/article/details/3212568>

## 推荐阅读

1. 武汉肺炎利好消息！
2. 如何设计一个安全的对外接口
3. 互联网公司的抗疫情行动！
4. 如何获取靠谱的新型冠状病毒疫情



微信搜一搜

Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot 常见错误及解决方法

Java后端 1月10日

以下文章来源于阿里巴巴中间件，作者方剑

Aliware 阿里巴巴中间件  
Aliware阿里巴巴中间件官方账号

## 导读

Spring Boot 作为 Java 生态中最流行的开发框架，意味着被数以万计的开发者所使用。下面根据我们自身遇到的问题，加上用户提供的一些反馈，来大致梳理下 Spring Boot 的常见错误及解决方法。

### 找不到配置？配置不对？配置被覆盖？

Spring Boot 配置加载过程解析：

1、Spring Boot 配置的加载有着约定俗成的步骤：从 resources 目录下加载 application.properties/application.yml；

再根据里面的

spring.profiles.active

来加载不同 profile 的配置文件

application-dev.properties/application-dev.yml

(比如加载 profile 为 dev 的配置文件)。

2、Spring Boot 所有的配置来源会被构造成 PropertySource，比如 -D 参数，-- 参数，系统参数，配置文件配置等等。这些 PropertySource 最终会被添加到 List 中，获取配置的时候会遍历这个 List，直到第一次获取对应 key 的配置，所以会存在优先级的问题。具体配置的优先级参考：

<https://stackoverflow.com/a/45822571>

配置覆盖案例：

Nacos 服务注册的 IP 可以通过 spring.cloud.nacos.discovery.ip 设置，当我们打成 JAR 包之后，如需修改注册 IP，可以通过 -Dspring.cloud.nacos.discovery.ip=xxx

(-D参数配置的优先级比配置文件要高)。

配置问题排查：

进入 <http://host:port/actuator/env> 这个 endpoint 查看具体的配置项属于哪个 PropertySource。

### Jar 包启动不了

执行 Spring Boot 构建的 jar 包后，返回 "my.jar中没有主清单属性" 错误。

错误分析：Spring Boot 的正常 jar 包运行方是通过 spring-boot-loader 这个模块里的 JarLauncher 完成的，该类内部提供了一套运行的规范。

解决方案：在 pom 里加上 spring-boot-maven-plugin 的 maven 插件配置(该插件会在 jar 里加入 spring-boot-loader 的代码，并在 MANIFEST.MF 中的 Main-Class 里写入 JarLauncher)：

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

## 自动化配置类没有被加载

条件注解是 Spring Boot 的核心特性之一, 第三方的 starter 或我们自定义的 starter 内部都会加载一些 AutoConfiguration, 有时候会存在一些 AutoConfiguration 没有被加载的情况。导致出现 NoSuchBeanDefinitionException, UnsatisfiedDependencyException 等异常

排查步骤(三种方式):

1、把 spring 的日志级别调到 debug 级别:

```
logging.level.org.springframework: debug.
```

2、从 ApplicationContext 中获取 ConditionEvaluationReport, 得到内部的 ConditionEvaluationReport.ConditionAndOutcomes 类中的输出信息。

3、进入 `http://host:port/actuator/conditions` 这个 endpoint 查看条件注解的 match 情况。

Tips: 欢迎关注微信公众号: Java后端

这是日志打印的不满足条件的 AutoConfiguratoin:

```
Unconditional classes:

```

```
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration
org.springframework.cloud.client.ReactiveCommonsClientAutoConfiguration
org.springframework.boot.actuate.autoconfigure.info.InfoContributorAutoConfiguration
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration
org.springframework.cloud.client.discovery.simple.SimpleDiscoveryClientAutoConfiguration
org.springframework.cloud.client.CommonsClientAutoConfiguration
org.springframework.cloud.commons.httpclient.HttpClientConfiguration
org.springframework.boot.actuate.autoconfigure.endpoint.EndpointAutoConfiguration
org.springframework.cloud.loadbalancer.config.BlockingLoadBalancerClientAutoConfiguration
```

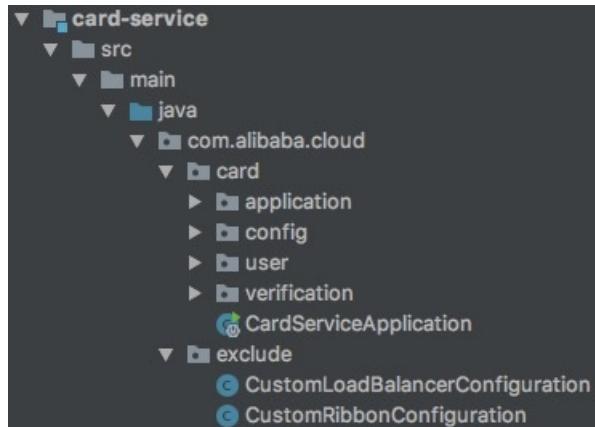
## 定义的 Component 没有被扫描到

@SpringBootApplication 注解内部也会使用 @ComponentScan 注解用于扫描 Component。默认情况下会扫描 @SpringBootApplication 注解修饰的入口类的包以及它下面的子包中所有的 Component。

@ComponentScan:

<https://github.com/StabilityMan/StabilityGuide/blob/master/ComponentScan>

这是推荐的包结构中项目的结构:



exclude 包下的类不会被扫描到， card 包下的类会被扫描到。

## Actuator Endpoint 访问不了

访问 Actuator，出现 404 错误。

解决方案:

1、Spring Boot 2.x 版本对 Actuator 做了大量的修改，其中访问的路径从

`http://host:port/endpointid`

变成了

`http://host:port/actuator/endpointid`。

确保访问的路径正确。

2、Endpoint 有 Security 要求，

在配置里加上 `management.endpoints.web.exposure.include=*` 即可。

- END -

## 推荐阅读

1. [扎心一问：分库分表就能无限扩容吗](#)
2. [全面了解 Nginx 主要应用场景](#)
3. [一场近乎完美基于 Dubbo 的微服务改造实践](#)
4. [什么是一致性 Hash 算法？](#)
5. [团队开发中 Git 最佳实践](#)



喜欢文章, 点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

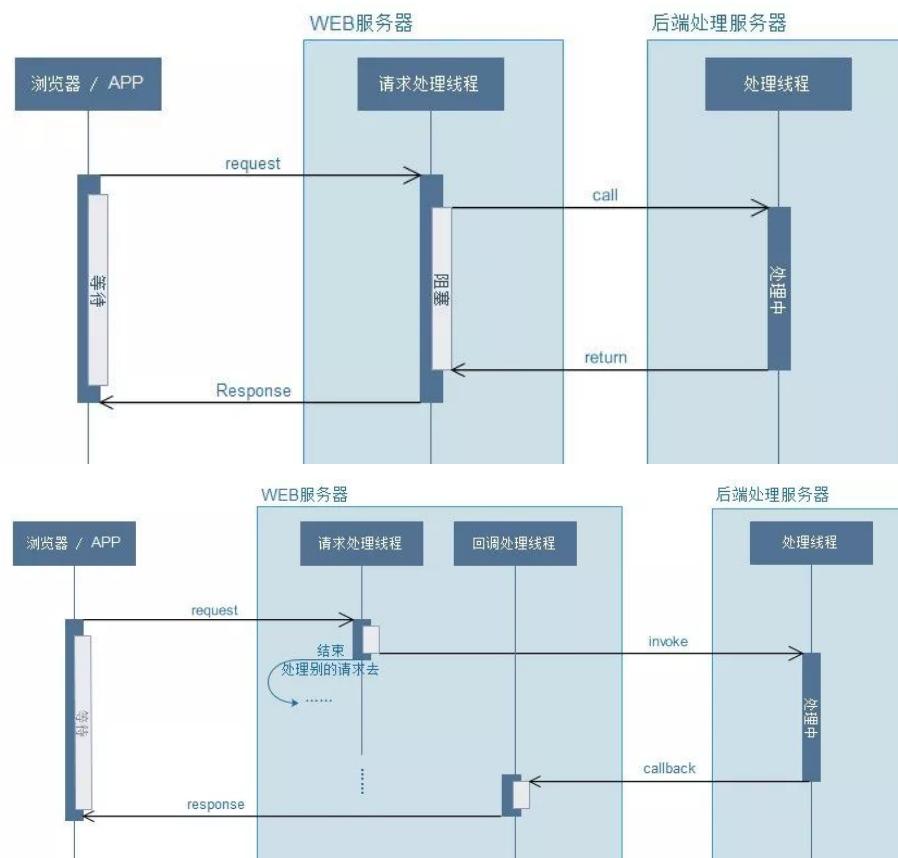
# Spring Boot 异步请求和异步调用

Java后端 3月1日



## 一、Spring Boot中异步请求的使用

### 1、异步请求与同步请求



### 特点：

可以先释放容器分配给请求的线程与相关资源，减轻系统负担，释放了容器所分配线程的请求，其响应将被延后，可以在耗时处理完成（例如长时间的运算）时再对客户端进行响应。

**一句话：增加了服务器对客户端请求的吞吐量**（实际生产上我们用的比较少，如果并发请求量很大的情况下，我们会通过nginx把请求负载到集群服务的各个节点上来分摊请求压力，当然还可以通过消息队列来做请求的缓冲）。

### 2、异步请求的实现

方式一：Servlet方式实现异步请求

```

@RequestMapping(value = "/email/servletReq", method = GET)
public void servletReq (HttpServletRequest request, HttpServletResponse response) {
 AsyncContext asyncContext = request.startAsync();
 //设置监听器:可设置其开始、完成、异常、超时等事件的回调处理
 asyncContext.addListener(new AsyncListener() {
 @Override
 public void onTimeout(AsyncEvent event) throws IOException {
 System.out.println("超时了...");
 //做一些超时后的相关操作...
 }
 @Override
 public void onStartAsync(AsyncEvent event) throws IOException {
 System.out.println("线程开始");
 }
 @Override
 public void onError(AsyncEvent event) throws IOException {
 System.out.println("发生错误: "+event.getThrowable());
 }
 @Override
 public void onComplete(AsyncEvent event) throws IOException {
 System.out.println("执行完成");
 //这里可以做一些清理资源的操作...
 }
 });
 //设置超时时间
 asyncContext.setTimeout(20000);
 asyncContext.start(new Runnable() {
 @Override
 public void run() {
 try {
 Thread.sleep(10000);
 System.out.println("内部线程: " + Thread.currentThread().getName());
 asyncContext.getResponse().setCharacterEncoding("utf-8");
 asyncContext.getResponse().setContentType("text/html;charset=UTF-8");
 asyncContext.getResponse().getWriter().println("这是异步的请求返回");
 } catch (Exception e) {
 System.out.println("异常: "+e);
 }
 //异步请求完成通知
 //此时整个请求才完成
 asyncContext.complete();
 }
 });
 //此时之类 request 的线程连接已经释放了
 System.out.println("主线程: " + Thread.currentThread().getName());
}

```

方式二：使用很简单，直接返回的参数包裹一层callable即可，可以继承WebMvcConfigurerAdapter类来设置默认线程池和超时处理

```
@RequestMapping(value = "/email/callableReq", method = GET)
@ResponseBody
public Callable<String> callableReq () {
 System.out.println("外部线程: " + Thread.currentThread().getName());

 return new Callable<String>() {

 @Override
 public String call() throws Exception {
 Thread.sleep(10000);
 System.out.println("内部线程: " + Thread.currentThread().getName());
 return "callable!";
 }
 };
}

@Configuration
public class RequestAsyncPoolConfig extends WebMvcConfigurerAdapter {

 @Resource
 private ThreadPoolTaskExecutor myThreadPoolTaskExecutor;

 @Override
 public void configureAsyncSupport(final AsyncSupportConfigurer configurer) {
 //处理 callable超时
 configurer.setDefaultTimeout(60*1000);
 configurer.setTaskExecutor(myThreadPoolTaskExecutor);
 configurer.registerCallableInterceptors(timeoutCallableProcessingInterceptor());
 }

 @Bean
 public TimeoutCallableProcessingInterceptor timeoutCallableProcessingInterceptor() {
 return new TimeoutCallableProcessingInterceptor();
 }
}
```

方式三：和方式二差不多，在Callable外包一层，给WebAsyncTask设置一个超时回调，即可实现超时处理

```
@RequestMapping(value = "/email/webAsyncReq", method = GET)
@ResponseBody
public WebAsyncTask<String> webAsyncReq () {
 System.out.println("外部线程: " + Thread.currentThread().getName());
 Callable<String> result = () -> {
 System.out.println("内部线程开始: " + Thread.currentThread().getName());
 try {
 TimeUnit.SECONDS.sleep(4);
 } catch (Exception e) {
 // TODO: handle exception
 }
 logger.info("副线程返回");
 System.out.println("内部线程返回: " + Thread.currentThread().getName());
 return "success";
 };
 WebAsyncTask<String> wat =new WebAsyncTask<String>(3000L, result);
 wat.onTimeout(new Callable<String>() {
 @Override
 public String call() throws Exception {
 // TODO Auto-generated method stub
 return "超时";
 }
 });
 return wat;
}
```

方式四：DeferredResult可以处理一些相对复杂一些的业务逻辑，最主要还是可以在另一个线程里面进行业务处理及返回，即可在两个完全不相干的线程间的通信。

```

@RequestMapping(value = "/email/deferredResultReq", method = GET)
@ResponseBody
public DeferredResult<String> deferredResultReq () {
 System.out.println("外部线程: " + Thread.currentThread().getName());
 //设置超时时间
 DeferredResult<String> result = new DeferredResult<String>(60*1000L);
 //处理超时事件 采用委托机制
 result.onTimeout(new Runnable() {

 @Override
 public void run() {
 System.out.println("DeferredResult超时");
 result.setResult("超时了!");
 }
 });
 result.onCompletion(new Runnable() {

 @Override
 public void run() {
 //完成后
 System.out.println("调用完成");
 }
 });
 myThreadPoolTaskExecutor.execute(new Runnable() {

 @Override
 public void run() {
 //处理业务逻辑
 System.out.println("内部线程: " + Thread.currentThread().getName());
 //返回结果
 result.setResult("DeferredResult!!!");
 }
 });
 return result;
}

```

## 二、Spring Boot中异步调用的使用

### 1、介绍

异步请求的处理。除了异步请求，一般上我们用的比较多的应该是异步调用。通常在开发过程中，会遇到一个方法是和实际业务无关的，没有紧密性的。比如记录日志信息等业务。这个时候正常就是启一个新线程去做一些业务处理，让主线程异步的执行其他业务。

小插曲：更多Spring Boot 相关文章可以本公众号（Java后端）回复 技术博文，获取~

### 2、使用方式（基于spring下）

需要在启动类加入@EnableAsync使异步调用@Async注解生效

在需要异步执行的方法上加入此注解即可@Async("threadPool"),threadPool为自定义线程池。

代码略。。。就俩标签，自己试一把就可以了

### 3、注意事项

在默认情况下，未设置TaskExecutor时， 默认是使用SimpleAsyncTaskExecutor这个线程池，但此线程不是真正意义上的线程池，因为线程不重用，每次调用都会创建一个新的线程。可通过控制台日志输出可以看出，每次输出线程名都是递增的。所以最好我们来自定义一个线程池。

调用的异步方法，不能为同一个类的方法（包括同一个类的内部类），简单来说，因为Spring在启动扫描时会为其创建一个代理类，而同类调用时，还是调用本身的代理类的，所以和平常调用是一样的。

其他的注解如@Cache等也是一样的道理，说白了，就是Spring的代理机制造成的。所以在开发中，最好把异步服务单独抽出一个类来管理。下面会重点讲述。。

#### 4、什么情况下会导致@Async异步方法会失效？

**调用同一个类下注有@Async异步方法：**

在spring中像@Async和@Transactional、cache等注解本质使用的是动态代理，其实Spring容器在初始化的时候Spring容器会将含有AOP注解的类对象“替换”为代理对象（简单这么理解），那么注解失效的原因就很明显了，就是因为调用方法的是对象本身而不是代理对象，因为没有经过Spring容器，那么解决方法也会沿着这个思路来解决。

**调用的是静态(static )方法**

**调用(private)私有化方法**

#### 5、解决4中问题1的方式（其它2,3两个问题自己注意下就可以了）

**将要异步执行的方法单独抽取成一个类**，原理就是当你把执行异步的方法单独抽取成一个类的时候，这个类肯定是被Spring管理的，其他Spring组件需要调用的时候肯定会注入进去，这时候实际上注入进去的就是代理类了。

其实我们的注入对象都是从Spring容器中给当前Spring组件进行成员变量的赋值，由于某些类使用了AOP注解，那么实际上在Spring容器中实际存在的是它的代理对象。那么我们就可以**通过上下文获取自己的代理对象调用异步方法**。

```
@Controller
@RequestMapping("/app")
public class EmailController {

 //获取ApplicationContext对象方式有多种,这种最简单,其它的大家自行了解一下
 @Autowired
 private ApplicationContext applicationContext;

 @RequestMapping(value = "/email/asyncCall", method = GET)
 @ResponseBody
 public Map<String, Object> asyncCall () {
 Map<String, Object> resMap = new HashMap<String, Object>();
 try{
 //这样调用同类下的异步方法是不起作用的
 //this.testAsyncTask();
 //通过上下文获取自己的代理对象调用异步方法
 EmailController emailController = (EmailController)applicationContext.getBean(EmailController.class);
 emailController.testAsyncTask();
 resMap.put("code",200);
 }catch (Exception e){
 resMap.put("code",400);
 logger.error("error!",e);
 }
 return resMap;
 }

 //注意一定是public,且是非static方法
 @Async
 public void testAsyncTask() throws InterruptedException {
 Thread.sleep(10000);
 System.out.println("异步任务执行完成! ");
 }
}
```

开启cglib代理，手动获取Spring代理类,从而调用同类下的异步方法。首先，在启动类上加

上@EnableAspectJAutoProxy(exposeProxy = true)注解。代码实现，如下：

```

@Service
@Transactional(value = "transactionManager", readOnly = false, propagation = Propagation.REQUIRED, rollbackFor = Throwable.class)
public class EmailService {

 @Autowired
 private ApplicationContext applicationContext;

 @Async
 public void testSyncTask() throws InterruptedException {
 Thread.sleep(1000);
 System.out.println("异步任务执行完成! ");
 }

 public void asyncCallTwo() throws InterruptedException {
 //this.testSyncTask();
 // EmailService emailService = (EmailService)applicationContext.getBean(EmailService.class);
 // emailService.testSyncTask();
 boolean isAop = AopUtils.isAopProxy(EmailController.class); //是否是代理对象;
 boolean isCglib = AopUtils.isCglibProxy(EmailController.class); //是否是CGLIB方式的代理对象;
 boolean isJdk = AopUtils.isJdkDynamicProxy(EmailController.class); //是否是JDK动态代理方式的代理对象;
 //以下才是重点!!!
 EmailService emailService = (EmailService)applicationContext.getBean(EmailService.class);
 EmailService proxy = (EmailService) AopContext.currentProxy();
 System.out.println(emailService == proxy ? true : false);
 proxy.testSyncTask();
 System.out.println("end!!!!");
 }
}

```

### 三、异步请求与异步调用的区别

两者的使用场景不同，异步请求用来解决并发请求对服务器造成压力，从而提高对请求的吞吐量；而异步调用是用来做一些非主线流程且不需要实时计算和响应的任务，比如同步日志到kafka中做日志分析等。

异步请求是会一直等待response相应的，需要返回结果给客户端的；而异步调用我们往往会马上返回给客户端响应，完成这次整个的请求，至于异步调用的任务后台自己慢慢跑就行，客户端不会关心。

### 四、总结

异步请求和异步调用的使用到这里基本就差不多了，有问题还希望大家多多指出。这边文章提到了动态代理，而spring中Aop的实现原理就是动态代理，后续会对动态代理做详细解读，还望多多支持哈。

作者 | 会炼钢的小白龙

链接 | [cnblogs.com/baixianlong/p/10661591.html](http://cnblogs.com/baixianlong/p/10661591.html)

- END -

如果看到这里，说明你喜欢这篇文章，请[转发](#)、[点赞](#)。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。



### 推荐阅读

1. [删库跑路！创始人回应了](#)
2. [高频使用的 Git 命令](#)
3. [一个依赖搞定 Session 共享](#)
4. [浅谈 Web 网站架构演变过程](#)



微信搜一搜

Q Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot 微信点餐系统

Tommmmm Java后端 2019-11-11

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | Tommmmm

链接 | [www.jianshu.com/p/ae14101989f2](http://www.jianshu.com/p/ae14101989f2)

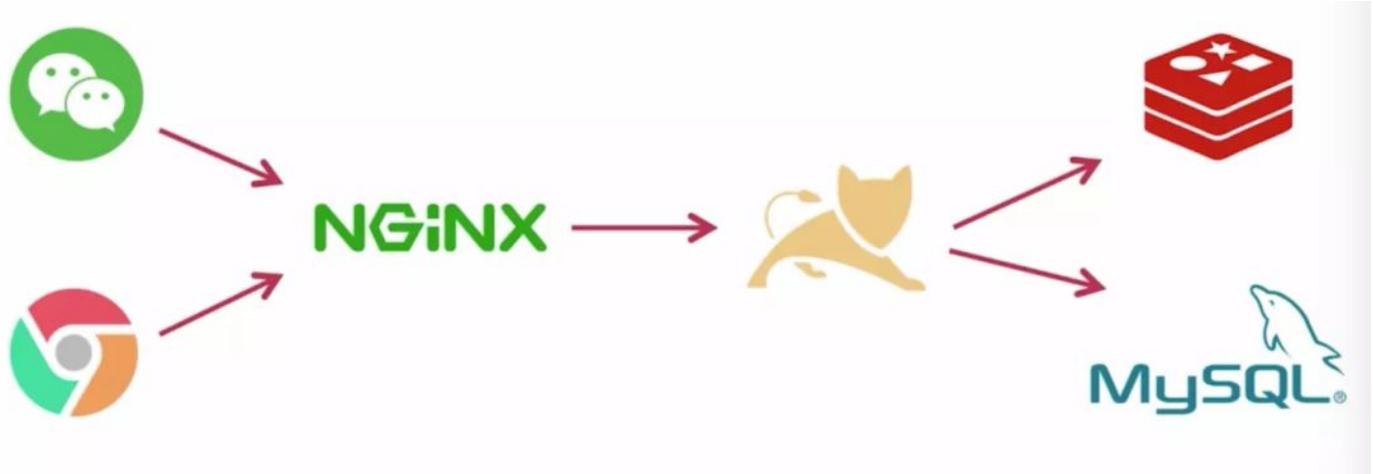
下文是对微信点餐系统项目的总结, 使用 Spring Boot 开发, 采用前后端分离架构, 针对此项目技术细节给出具体解释, 同样会分享源码给大家, 可以关注微信公众号: Java后端, 回复 点餐, 即可获取源码地址。本文作者 Tommmmm 欢迎点击阅读原文访问作者博客。

## 架构

前后端分离:



部署架构：



Nginx与Tomcat的关系在我的这篇文章，几分钟可以快速了解：

<https://www.jianshu.com/p/22dcb7ef9172>

补充：

setting.xml 文件的作用：**settings.xml是maven的全局配置文件**。而pom.xml文件是所在项目的局部配置。Settings.xml中包含类似本地仓储位置、修改远程仓储服务器、认证信息等配置。

maven的作用：借助Maven，**可将jar包仅仅保存在“仓库”中，有需要该文件时，就引用该文件接口，不需要复制文件过来占用空间。**

注：这个“仓库”应该就是本地安装maven的目录下的Repository的文件夹

## 分布式锁

线程锁：当某个方法或代码使用锁，在同一时刻仅有一个线程执行该方法或该代码段。**线程锁只在同一JVM中有效**，因为线程锁的实现在根本上是依靠线程之间**共享内存**实现的。如synchronized

进程锁：为了控制同一操作系统中多个进程访问某个共享资源。

分布式锁：当多个进程不在同一个系统中，用分布式锁控制多个进程对资源的访问。

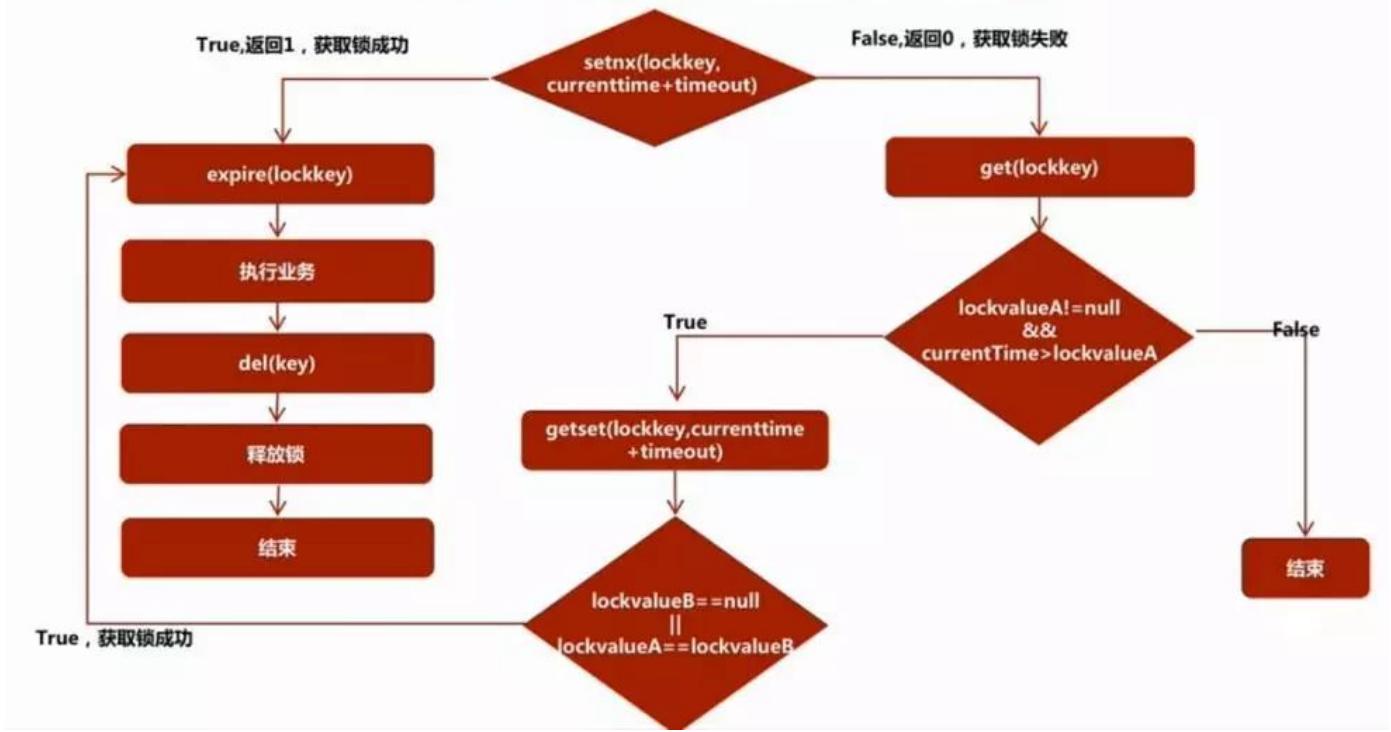
分布式锁一般有三种实现方式：1. 数据库乐观锁；2. 基于Redis的分布式锁；3. 基于ZooKeeper的分布式锁。

乐观锁的实现：使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。

CAS：可以阅读我的这篇文章：<https://www.jianshu.com/p/456bb1ea9627>

分布式锁基于Redis的实现：(本系统锁才用的)

# Redis分布式锁优化版流程图



## 基本命令：

1. SETNX (SET if Not exist) : 当且仅当 key 不存在, 将 key 的值设为 value , 并返回1; 若给定的 key 已经存在, 则 SETNX 不做任何动作, 并返回0。
2. GETSET: 将给定 key 的值设为 value , 并返回 key 的旧值。先根据key获取到旧的value, 再set新的value。
3. EXPIRE 为给定 key 设置生存时间,当 key 过期时, 它会被自动删除。

Tips：欢迎订阅公众号 Java后端，接受每日技术博文推送。

## 加锁方式：

这里的 jedis 是Java对Redis的集成

```
jedis.set(String key, String value, String nxxx, String expx, int time)
```

错误的加锁方式1：如果程序在执行完`setnx()`之后突然崩溃，导致锁没有设置过期时间。那么将会发生死锁。

```
Long result = jedis.setnx(Key, value);
if (result == 1) {
 jedis.expire(Key, expireTime);
}
```

错误的加锁方式2：分布式锁才用 (Key, 过期时间) 的方式, 如果锁存在, 那么获取它的过期时间, 如果锁的确已经过期了, 那么获得锁, 并且设置新的过期时间

错误分析：不同的客户端之间需要同步好时间。

```

long expires = System.currentTimeMillis() + expireTime;
String expiresStr = String.valueOf(expires);

if (jedis.setnx(lockKey, expiresStr) == 1) {
 return true;
}

String currentValueStr = jedis.get(lockKey);
if (currentValueStr != null && Long.parseLong(currentValueStr) < System.currentTimeMillis()) {

 String oldValueStr = jedis.getSet(lockKey, expiresStr);
 if (oldValueStr != null && oldValueStr.equals(currentValueStr)) {

 return true;
 }
}

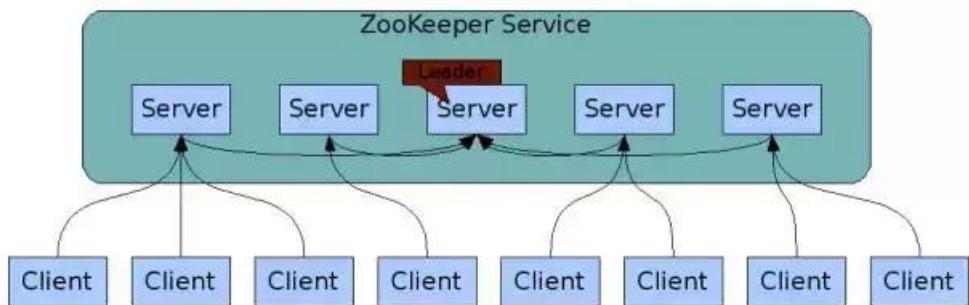
return false;

```

解锁：判断锁的拥有者后可以使用 `jedis.del(lockKey)` 来释放锁。

## 分布式锁基于Zookeeper的实现

Zookeeper简介：Zookeeper提供一个多层次的节点命名空间（节点称为znode），每个节点都用一个以斜杠（/）分隔的路径表示，而且每个节点都有父节点（根节点除外）。例如，/foo/doo这个表示一个znode，它的父节点为/foo，父父节点为/，而/为根节点没有父节点。



client不论连接到哪个Server，展示给它都是同一个视图，这是zookeeper最重要的性能。

Zookeeper 的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同系统状态。

为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务，实现中zxid是一个64位的数字。

## Zookeeper的分布式锁原理

获取分布式锁的流程：

1. 在获取分布式锁的时候在locker节点(locker节点是Zookeeper的指定节点)下创建临时顺序节点，释放锁的时候删除该临时节点。

2. 客户端调用createNode方法在locker下创建临时顺序节点，然后调用getChildren(“locker”)来获取locker下面的所有子节点，注意此时不用设置任何Watcher。
3. 客户端获取到所有的子节点path之后，如果发现自己创建的子节点序号最小，那么就认为该客户端获取到了锁。
4. 如果发现自己创建的节点并非locker所有子节点中最小的，说明自己还没有获取到锁，此时客户端需要找到比自己小的那个节点，然后对其调用exist()方法，同时对其注册事件监听器。
5. 之后，让这个被关注的节点删除，则客户端的Watcher会收到相应通知，此时再次判断自己创建的节点是否是locker子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。

我的解释：A在Locker下创建了Node\_n → 循环（每次获取Locker下的所有子节点 → 对这些节点按节点自增号排序顺序 → 判断自己创建的Node\_n是否是第一个节点 → 如果是则获得了分布式锁 → 如果不是监听上一个节点Node\_n-1等它释放掉分布式锁。）

---

@ControllerAdvice处理全局异常

Mybatis注解方式的使用：

@insert 用注解方式写SQL语句

## 分布式系统的Session

**1、分布式系统：**多节点，节点发送数据交互，不共享主内存，但通过网络发送消息合作。

分布式：不同功能模块的节点

集群：相同功能的节点

### 2、Session与token

服务端在HTTP头里设置SessionID而客户端将其保存在cookie

而使用Token时需要手动在HTTP头里设置，服务器收到请求后取出cookie进行验证。

都是一个用户一个标志

### 3、分布式系统中的Session问题：

高并发：通过设计保证系统能够同时并行处理很多请求。

当高并发量的请求到达服务端的时候通过负载均衡的方式分发到集群中的某个服务器，这样就有可能导致同一个用户的多次请求被分发到集群的不同服务器上，就会出现取不到session数据的情况

**根据访问不同的URL，负载到不同的服务器上去**

三台机器，A1部署类目，A2部署商品，A3部署单服务

通用方案：用Redis保存Session信息，服务器需要时都去找Redis要。登录时保存好key-value，登出时让他失效

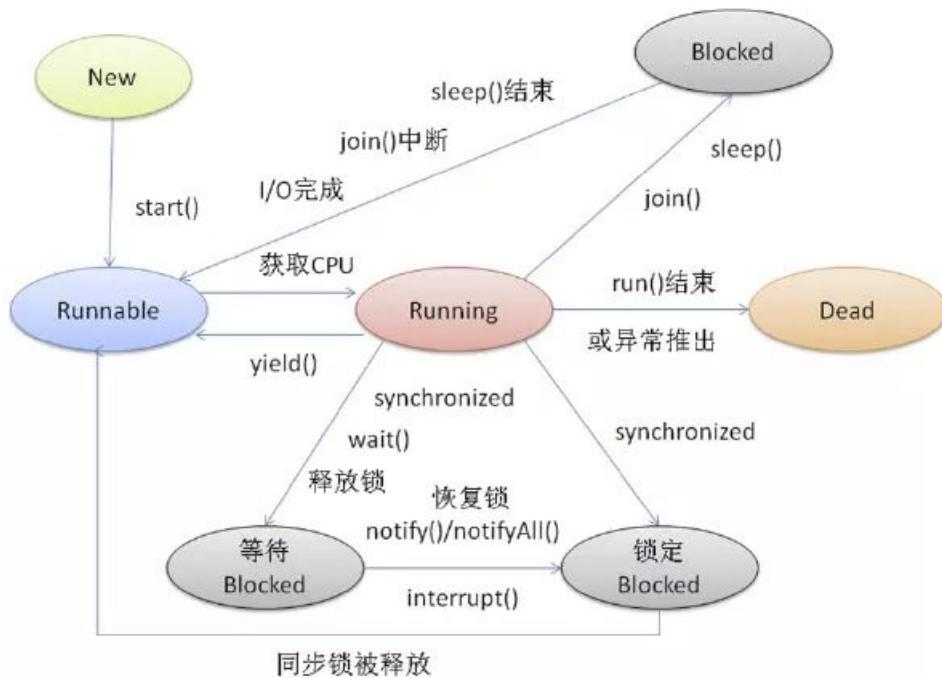
垂直扩展：IP哈希 IP的哈希值相同的访问同一台服务器

session的一致性：只要用户不重启浏览器，每次http短连接请求，理论上服务端都能定位到session，保持会话。

## Redis作为分布式锁

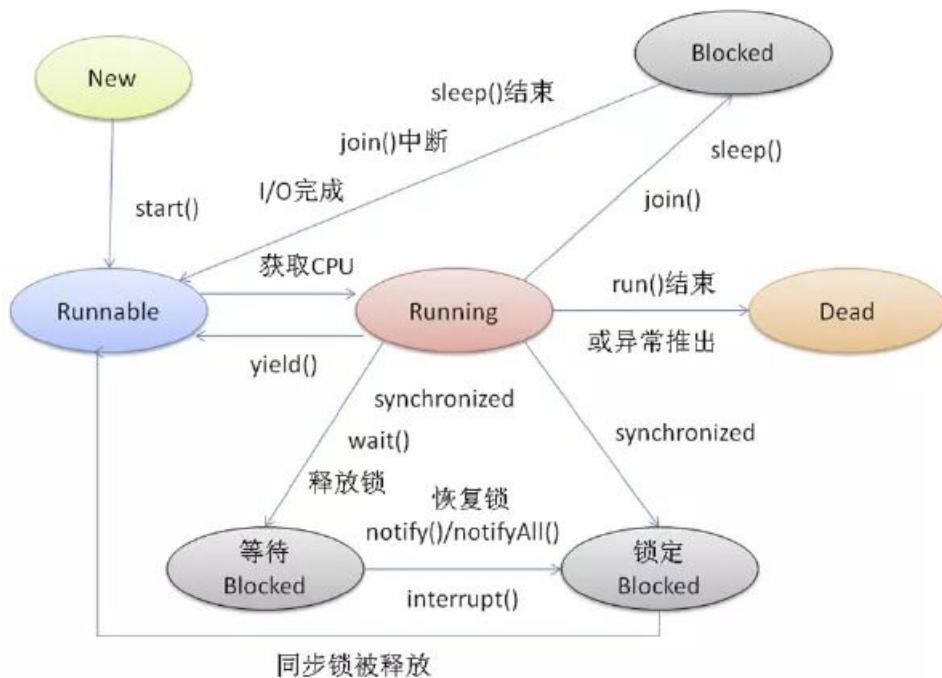
高并发：通过设计保证系统能够同时并行处理很多请求。

同步：Java中的同步指的是通过人为的控制和调度，保证共享资源的多线程访问成为线程安全。



线程的Block状态：

- 调用 **join()** 和 **sleep()** 方法， sleep() 时间结束或被打断
- wait()，使该线程处于等待池,直到notify()/notifyAll(): 不释放资源



此外，在runnable状态的线程是处于被调度的线程，Thread类中的yield方法可以让一个running状态的线程转入runnable。

Q：为什么wait,notify和notifyAll必须与synchronized一起使用？

Obj.wait()、Obj.notify必须在synchronized(Obj){…}语句块内。

A：wait就是说线程在获取对象锁后，主动释放对象锁，同时本线程休眠。

Q: Synchronized:

A: Synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。

公平和非公平锁的队列都基于锁内部维护的一个双向链表，表结点Node的值就是每一个请求当前锁的线程。公平锁则在于每次都是依次从队首取值。

ReentrantLock重入性：

重入锁可以看我的这两篇文章，都比较简单

<https://www.jianshu.com/p/587a4559442b>

<https://www.jianshu.com/p/1c52f17efaab>

Spring + Redis缓存的两个重要注解：

@cacheable 只会执行一次，当标记在一个方法上时表示该方法是支持缓存的，Spring会在其被调用后将其返回值缓存起来，以保证下次利用同样的参数来执行该方法时可以直接从缓存中获取结果。

@cacheput：与@Cacheable不同的是使用@CachePut标注的方法在执行前不会去检查缓存中是否存在之前执行过的结果，而是每次都会执行该方法，并将执行结果以键值对的形式存入指定的缓存中。

对数据库加锁

乐观锁与悲观锁

悲观锁依赖数据库实现：

```
select * from account where name="Erica" for update
```

这条sql语句锁定了account表中所有符合检索条件（name="Erica"）的记录，使该记录在修改期间其它线程不得占有

代码层加锁：

```
String hql ="from TUser as user where user.name='Erica'";
Query query = session.createQuery(hql);
query.setLockMode("user",LockMode.UPGRADE);
List userList = query.list();
```

## 其它

@Data类似于自动生成了Getter()、Setter()、ToString()等方法，这个可以参考历史文章：一份不可多得的Lombok学习指南

JAVA1.8的新特性StreamAPI：Collectors中提供了将流中的元素累积到汇聚结果的各种方式

```
List<Menu> menus=Menu.getMenus().stream().collect(Collectors.toList())
```

For - each写法：

for each语句是java5新增，在遍历数组、集合的时候，for each拥有不错的性能。

```
public static void main(String[] args) {
 String[] names = {"beibei", "jingjing"};
 for (String name : names) {
 System.out.println(name);
 }
}
```

for each虽然能遍历数组或者集合，但是只能用来遍历，无法在遍历的过程中对数组或者集合进行修改。

BindingResult：一个@Valid的参数后必须紧挨着一个BindingResult参数，否则spring会在校验不通过时直接抛出异常

```
@Data
public class OrderForm {

 @NotEmpty(message = "姓名必填")
 private String name;
}
```

后台：

```
@RequestMapping("save")
public String save(@Valid OrderForm order,BindingResult result) {

 if(result.hasErrors()){
 List<ObjectError> ls=result.getAllErrors();
 for (int i = 0; i < ls.size(); i++) {
 log.error("参数不正确, OrderForm={} ", order);
 throw new SellException(
 ,
 result.getFieldError().getDefaultMessage()
);
 System.out.println("error:"+ls.get(i));
 }
 }
 return "adduser";
}
```

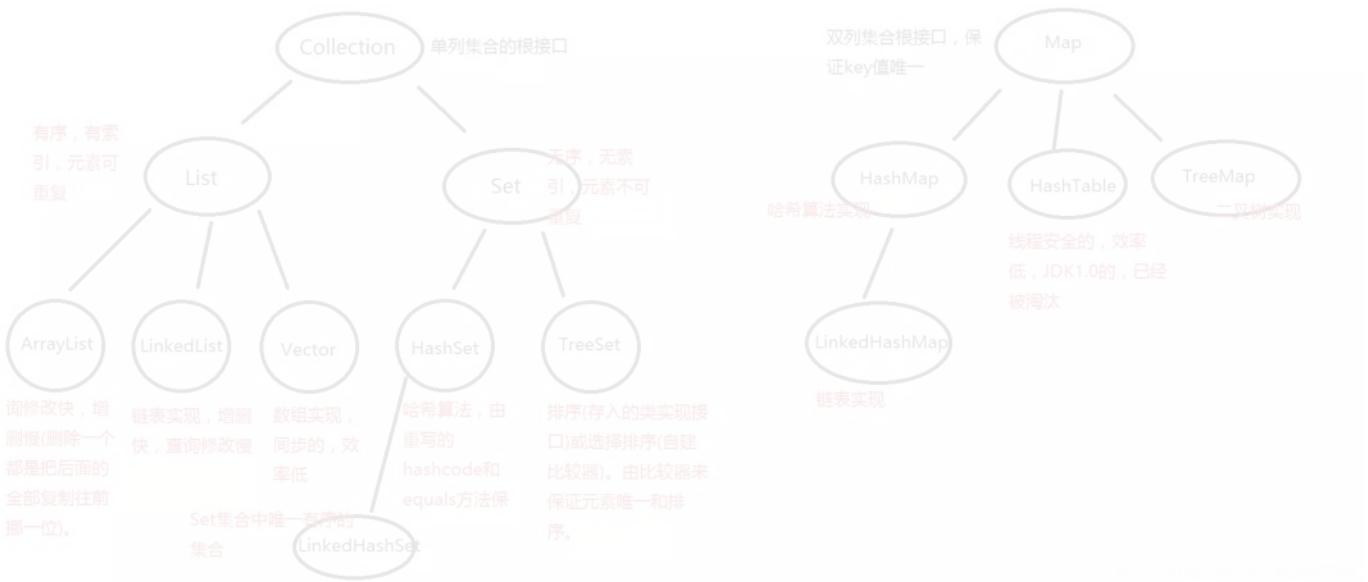
result.getFieldError().getDefaultMessage()可抛出“姓名必填”的异常。

#### 4、List转为Map

```
public class Apple {
 private Integer id;
 private String name;
 private BigDecimal money;
 private Integer num;

}
List<Apple> appleList = new ArrayList<>();
Apple apple1 = new Apple(1,"苹果1",new BigDecimal("3.25"),10);
Apple apple12 = new Apple(1,"苹果2",new BigDecimal("1.35"),20);
Apple apple2 = new Apple(2,"香蕉",new BigDecimal("2.89"),30);
Apple apple3 = new Apple(3,"荔枝",new BigDecimal("9.99"),40);
appleList.add(apple1);
appleList.add(apple12);
appleList.add(apple2);
appleList.add(apple3);
Map<Integer, Apple> appleMap = appleList.stream().collect(Collectors.toMap(Apple::getId, a -> a,(k1,k2)->k1));
```

#### 5、Collection的子类：List、Set



List: ArrayList、LinkedList、Vector

List: 有序容器, 允许null元素, 允许重复元素

Set: 元素是无序的, 不允许元素

最流行的是基于 HashMap 实现的 HashSet, 由 hashCode() 和 equals() 保证元素的唯一性。

可以用 set 帮助去掉 List 中的重复元素, set 的构造方法的参数可以是 List, 构造后是一个去重的 set

HashMap 的补充: 它不是 Collection 下的

Map 可以使用 containsKey() / containsValue() 来检查其中是否含有某个 key / value。

HashMap 会利用对象的 hashCode 来快速找到 key。

插入过程: 通过一个 hash 函数确定 Entry 的插入位置 index=hash(key), 但是数组的长度有限, 可能会发生 index 冲突, 当发生了冲突时, 会使用头插法, 即为新来的 Entry 指向旧的 Entry, 成为一个链表。

每次插入时依次遍历它的 index 下的单链表, 如果存在 Key 一致的节点, 那么直接替换, 并且返回新的值。

但是单链表不会一直增加元素, 当元素个数超过 8 个时, 会尝试将单链表转化为红黑树存储。

为何加载因子默认为 0.75? (0.75 开始扩容)

答: 通过源码里的 javadoc 注释看到, 元素在哈希表中分布的桶频率服从参数为 0.5 的泊松分布。

源码地址:

<https://github.com/923310233/wxOrder>



喜欢文章, 点个在看 

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！



微信搜一搜

Java后端

作者 | 艾神一不小心

<https://urlify.cn/zemY7z>

## 前言

实际开发中缓存处理是必须的，不可能我们每次客户端去请求一次服务器，服务器每次都要去数据库中进行查找，为什么要使用缓存？说到底是为了提高系统的运行速度。将用户频繁访问的内容存放在离用户最近，访问速度最快的地方，提高用户的响应速度，今天先来讲下在 Spring Boot 中整合 Redis 的详细步骤。

## 一、安装

Redis下载地址:

<https://redis.io/download>

首先要在本地安装一个redis程序，安装过程十分简单（略过），安装完成后进入到 Redis 文件夹中可以看到如下：

此电脑 > 开发工具 (G:) > redis			
名称	修改日期	类型	大小
dump.rdb	2018/6/12 10:11	RDB 文件	15 KB
Redis%20on%20Windows%20Releas...	2016/7/16 15:03	Office Open XM...	13 KB
Redis%20on%20Windows.docx	2016/7/16 15:03	Office Open XM...	17 KB
redis.windows.conf	2016/7/16 15:03	CONF 文件	43 KB
redis.windows-service.conf	2016/7/16 15:03	CONF 文件	43 KB
redis-64.3.0.503.zip	2018/5/15 19:06	好压 ZIP 压缩文件	5,846 KB
redis-benchmark.exe	2016/7/16 15:03	应用程序	405 KB
redis-benchmark.pdb	2016/7/16 15:03	PDB 文件	4,268 KB
redis-check-aof.exe	2016/7/16 15:03	应用程序	260 KB
redis-check-aof.pdb	2016/7/16 15:03	PDB 文件	3,436 KB
redis-check-dump.exe	2016/7/16 15:03	应用程序	271 KB
redis-check-dump.pdb	2016/7/16 15:03	PDB 文件	3,404 KB
redis-cli.exe	2016/7/16 15:03	应用程序	480 KB
redis-cli.pdb	2016/7/16 15:03	PDB 文件	4,412 KB
redis-server.exe	2016/7/16 15:03	应用程序	1,525 KB
redis-server.pdb	2016/7/16 15:03	PDB 文件	6,748 KB
Windows%20Service%20Documentati...	2016/7/16 15:03	Office Open XM...	14 KB

点击 redis-server.exe 开启 Redis 服务，可以看到如下图所示即代表开启 Redis 服务成功：

```

[17368] 12 Jun 09:11:48.303 # Warning: no config file specified, using the default config. In order to specify a config file use G:\redis\redis-server.exe /path/to/redis.conf

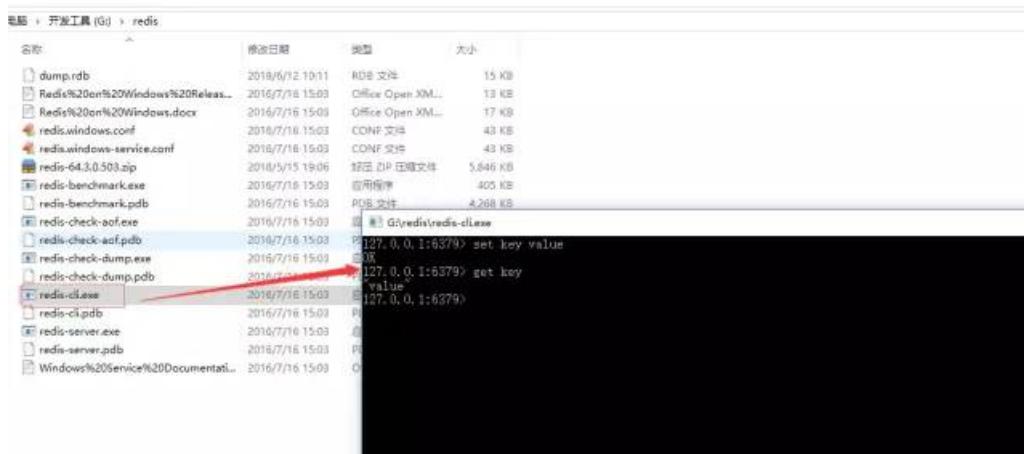
Redis 3.0.503 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 17368

http://redis.io

[17368] 12 Jun 09:11:48.313 # Server started, Redis version 3.0.503
[17368] 12 Jun 09:11:48.340 * DB loaded from disk: 0.027 seconds
[17368] 12 Jun 09:11:48.340 * The server is now ready to accept connections on port 6379
[17368] 12 Jun 10:11:49.045 * 1 changes in 3600 seconds. Saving...
[17368] 12 Jun 10:11:49.054 * Background saving started by pid 4152
[17368] 12 Jun 10:11:51.174 # fork operation complete
[17368] 12 Jun 10:11:51.174 * Background saving terminated with success

```

那么我们可以开启 Redis 客户端进行测试：



## 二、整合到 Spring Boot

1、在项目中加入 Redis 依赖，pom 文件中添加如下。

```

<!--整合Redis缓存支持-->
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

```

2、在 application.yml 中添加 Redis 配置。

```

##默认密码为空
redis:
 host:127.0.0.1
 #Redis服务器连接端口
 port:6379
 jedis:
 pool:
 #连接池最大连接数(使用负值表示没有限制)
 max-active:100
 #连接池中的最小空闲连接
 max-idle:10
 #连接池最大阻塞等待时间(使用负值表示没有限制)
 max-wait:10000
 #连接超时时间(毫秒)
 timeout:5000
 #默认是索引为0的数据库
 database:0

```

3、新建 RedisConfiguration 配置类，继承 CachingConfigurerSupport，@EnableCaching 开启注解。

```
@Configuration
@EnableCaching
public class RedisConfiguration extends CachingConfigurerSupport {
 /**
 *自定义生成key的规则
 */
 @Override
 public KeyGenerator keyGenerator() {
 return new KeyGenerator() {
 @Override
 public Object generate(Object o, Method method, Object... objects) {
 //格式化缓存key字符串
 StringBuildersb= new StringBuilder();
 //追加类名
 sb.append(o.getClass().getName());
 //追加方法名
 sb.append(method.getName());
 //遍历参数并且追加
 for(Objectobj:objects){
 sb.append(obj.toString());
 }
 System.out.println("调用Redis缓存Key:"+sb.toString());
 return sb.toString();
 }
 };
 }

 /**
 *采用RedisCacheManager作为缓存管理器
 * @param connectionFactory
 */
 @Bean
 public CacheManager cacheManager(RedisConnectionFactory connectionFactory) {
 RedisCacheManagerredisCacheManager=RedisCacheManager.create(connectionFactory);
 return redisCacheManager;
 }

 @Bean
 public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory factory) {
 //解决键、值序列化问题
 StringRedisTemplatetemplate= new StringRedisTemplate(factory);
 Jackson2JsonRedisSerializerjackson2JsonRedisSerializer= new Jackson2JsonRedisSerializer(Object.class);
 ObjectMapperom= new ObjectMapper();
 om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
 om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
 jackson2JsonRedisSerializer.setObjectMapper(om);
 template.setValueSerializer(jackson2JsonRedisSerializer);
 template.afterPropertiesSet();
 return template;
 }
}
```

4、创建自定义的接口来定义需要的 Redis 的功能。

```

/**
*K指以hash结构操作时键类型
*T为数据实体应实现序列化接口,并定义serialVersionUID*RedisTemplate提供了五种数据结构操作类型hash/list/set/zset/value
*方法命名格式为数据操作类型+操作如hashPut指以hash结构(也就是map)想key添加键值对
*/
public interface RedisHelper<HK,T>{
 /**
 Hash结构添加元素 @paramkeykey* @paramhashKeyhashKey* @paramdomain元素
 */
 void hashPut(Stringkey,HKhashKey,Tdomain);

 /**
 Hash结构获取指定key所有键值对 @paramkey* @return
 */
 Map<HK,T>hashFindAll(Stringkey);

 /**
 Hash结构获取单个元素 @paramkey* @paramhashKey* @return
 */
 ThashGet(Stringkey,HKhashKey);

 void hashRemove(Stringkey,HKhashKey);

 /**
 List结构向尾部(Right)添加元素 @paramkey* @paramdomain* @return
 */
 LonglistPush(Stringkey,Tdomain);

 /**
 List结构向头部(Left)添加元素 @paramkey* @paramdomain* @return
 */
 LonglistUnshift(Stringkey,Tdomain);

 /**
 List结构获取所有元素 @paramkey* @return
 */
 List<T>listFindAll(Stringkey);

 /**
 List结构移除并获取数组第一个元素 @paramkey* @return
 */
 TlistLPop(Stringkey);

 /**
 *对象的实体类
 * @paramkey
 * @paramdomain
 * @return
 */
 void valuePut(Stringkey,Tdomain);

 /**
 *获取对象实体类
 * @paramkey
 * @return
 */
 TValue(Stringkey);

 void remove(Stringkey);

 /**
 设置过期时间 @paramkey键* @paramtimeout时间* @paramtimeUnit时间单位
 */
 boolean expirse(Stringkey,longtimeout,TimeUnittimeUnit);
}

```

5、下面是创建 RedisHelperImpl 进行接口的实现。

```

@Service("RedisHelper")
public class RedisHelperImpl<HK,T> implements RedisHelper<HK,T>{
 //在构造器中获取redisTemplate实例,key(nothashKey)默认使用String类型
 private RedisTemplate<String,T> redisTemplate;
 //在构造器中通过redisTemplate的工厂方法实例化操作对象
 private HashOperations<String, T> hashOperations;
 private ListOperations<String, T> listOperations;
 private ZSetOperations<String, T> zSetOperations;
 private SetOperations<String, T> setOperations;
 private ValueOperations<String, T> valueOperations;

 //IDEA虽然报错,但是依然可以注入成功,实例化操作对象后就可以直接调用方法操作Redis数据库
 @Autowired
 public RedisHelperImpl(RedisTemplate<String, T> redisTemplate) {
 this.redisTemplate=redisTemplate;
 this.hashOperations=redisTemplate.opsForHash();
 this.listOperations=redisTemplate.opsForList();
 this.zSetOperations=redisTemplate.opsForZSet();
 this.setOperations=redisTemplate.opsForSet();
 this.valueOperations=redisTemplate.opsForValue();
 }

 @Override
 public void hashPut(String key, HKhashKey, T domain) {
 hashOperations.put(key, hashKey, domain);
 }

 @Override
 public Map<HK, T> hashFindAll(String key) {
 return hashOperations.entries(key);
 }

 @Override
 public T hashGet(String key, HKhashKey) {
 return hashOperations.get(key, hashKey);
 }

 @Override
 public void hashRemove(String key, HKhashKey) {
 hashOperations.delete(key, hashKey);
 }

 @Override
 public Long listPush(String key, T domain) {
 return listOperations.rightPush(key, domain);
 }

 @Override
 public Long listUnshift(String key, T domain) {
 return listOperations.leftPush(key, domain);
 }

 @Override
 public List<T> listFindAll(String key) {
 if(!redisTemplate.hasKey(key)){
 return null;
 }
 return listOperations.range(key, 0, listOperations.size(key));
 }

 @Override
 public T listLPop(String key) {
 return listOperations.leftPop(key);
 }

 @Override
 public void valuePut(String key, T domain) {
 valueOperations.set(key, domain);
 }

 @Override
 public T getValue(String key) {
 return valueOperations.get(key);
 }

 @Override
 public void remove(String key) {
 redisTemplate.delete(key);
 }

 @Override
 public boolean expire(String key, long timeout, TimeUnit timeUnit) {
 return redisTemplate.expire(key, timeout, timeUnit);
 }
}

```

### 三、测试

编写 TestRedis 类进行测试。

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class TestRedis {

 @Autowired
 private StringRedisTemplate stringRedisTemplate;

 @Autowired
 private RedisTemplate<String, Object> redisTemplate;

 @Autowired
 private RedisHelperImpl redisHelper;

 @Test
 public void test() throws Exception {
 // 基本写法
 // stringRedisTemplate.opsForValue().set("aaa","111");
 // Assert.assertEquals("111",stringRedisTemplate.opsForValue().get("aaa"));
 // System.out.println(stringRedisTemplate.opsForValue().get("aaa"));

 Author user= newAuthor();
 user.setName("Alex");
 user.setIntro_l("不会打篮球的程序不是好男人");
 redisHelper.valuePut("aaa",user);
 System.out.println(redisHelper.getValue("aaa"));
 }

 @Test
 public void testObj() throws Exception {
 Author user= newAuthor();
 user.setName("Jerry");
 user.setIntro_l("不会打篮球的程序不是好男人!");

 ValueOperations<String,Author> operations=redisTemplate.opsForValue();
 operations.set("502",user);
 Thread.sleep(500);
 boolean exists=redisTemplate.hasKey("502");
 if(exists){
 System.out.println(redisTemplate.opsForValue().get("502"));
 } else{
 System.out.println("existsisfalse");
 }
 //Assert.assertEquals("aa",operations.get("com.neo.f").getUserName());
 }
}

```

运行 TestRedis 测试类，结果如下。



注意：如果在 RedisConfiguration 中不配置redisTemplate(RedisConnectionFactory factory) 注解，会造成键、值的一个序列化问题，有兴趣的可以去试一下，序列化:序列化框架的选型和比对。

## 四、项目实战

首先需要在程序的入口处 Application 中添加 @EnableCaching 开启缓存的注解。

```

@EnableCaching //开启缓存
@SpringBootApplication
public class PoetryApplication {

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

```

上面的 Redis 相关写法是我们自定义设置并获取的，那么我们经常要在访问接口的地方去使用 Redis 进行缓存相关实体对象以及集合等，那么我们怎么实现呢？

比如我现在想在 AuthorController 中去缓存作者相关信息的缓存数据，该怎么办呢？如下：

```
@RestController
@RequestMapping(value="/poem")
public class AuthorController {

 private final static Logger logger= LoggerFactory.getLogger(AuthorController.class);

 @Autowired
 private AuthorRepository authorRepository;

 @Cacheable(value="poemInfo") //自动根据方法生成缓存
 @PostMapping(value="/poemInfo")
 public Result<Author> author(@RequestParam("author_id") int author_id, @RequestParam("author_name") String author_name){
 if(StringUtils.isEmpty(author_id)||StringUtils.isEmpty(author_name)){
 return ResultUtils.error(ResultCode.INVALID_PARAM_EMPTY);
 }
 Author author;
 Optional<Author> optional=authorRepository.getAuthorByIdAndName(author_id,author_name);
 if(optional.isPresent()){
 author=optional.get();
 //通过\n或者多个空格进行过滤去重
 if(!StringUtils.isEmpty(author.getIntro_l())){
 String s=author.getIntro_l();
 String intro=s.split("\\s+")[0];
 author.setIntro_l(intro);
 }
 } else{
 return ResultUtils.error(ResultCode.NO_FIND_THINGS);
 }
 return ResultUtils.ok(author);
 }
}
```

这里 @Cacheable(value="poemInfo") 这个注解的意思就是自动根据方法生成缓存，value 就是缓存下来的 key。到这里我们就已经把 Redis 整合到了 Spring Boot 中了。

- E N D -

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓ 扫描二维码添加小编 ↓



## 推荐阅读

1. [自费送新款 iPad，包邮！](#)
2. [18 个示例带你掌握 Java 8 日期时间处理！](#)
3. [安利一款 IDEA 中强大的代码生成利器](#)
4. [如何获取靠谱的新型冠状病毒疫情](#)



微信搜一搜

Q Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！



作者 | Sans

编辑 | 江南一点雨

地址 | [juejin.im/post/5d087d605188256de9779e64](https://juejin.im/post/5d087d605188256de9779e64)

## 一.说明

Shiro 是一个安全框架,项目中主要用它做认证,授权,加密,以及用户的会话管理,虽然 Shiro 没有 SpringSecurity 功能更丰富,但是它轻量,简单,在项目中通常业务需求 Shiro 也都能胜任.

## 二.项目环境

- MyBatis-Plus 版本: 3.1.0
- SpringBoot 版本:2.1.5
- JDK 版本:1.8
- Shiro 版本:1.4
- Shiro-redis 插件版本:3.1.0

数据表(SQL 文件在项目中):数据库中测试号的密码进行了加密,密码皆为 123456

数据表名	中文表名	备注说明
sys_user	系统用户表	基础表
sys_menu	权限表	基础表
sys_role	角色表	基础表
sys_role_menu	角色与权限关系表	中间表
sys_user_role	用户与角色关系表	中间表

Maven 依赖如下：

```
<dependencies>
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
 <dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <scope>runtime</scope>
 </dependency>
 <!-- AOP依赖,一定要加,否则权限拦截验证不生效 -->
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-aop</artifactId>
 </dependency>
 <!-- lombok插件 -->
 <dependency>
 <groupId>org.projectlombok</groupId>
 <artifactId>lombok</artifactId>
 <optional>true</optional>
 </dependency>
 <!-- Redis -->
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
 </dependency>
 <!-- mybatisPlus 核心库 -->
 <dependency>
 <groupId>com.baomidou</groupId>
 <artifactId>mybatis-plus-boot-starter</artifactId>
 <version>3.1.0</version>
 </dependency>
 <!-- 引入阿里数据库连接池 -->
 <dependency>
 <groupId>com.alibaba</groupId>
 <artifactId>druid</artifactId>
 <version>1.1.6</version>
 </dependency>
 <!-- Shiro 核心依赖 -->
 <dependency>
 <groupId>org.apache.shiro</groupId>
 <artifactId>shiro-spring</artifactId>
 <version>1.4.0</version>
 </dependency>
 <!-- Shiro-redis插件 -->
 <dependency>
 <groupId>org.crazycake</groupId>
 <artifactId>shiro-redis</artifactId>
 <version>3.1.0</version>
 </dependency>
 <!-- StringUtil工具 -->
 <dependency>
 <groupId>org.apache.commons</groupId>
 <artifactId>commons-lang3</artifactId>
 <version>3.5</version>
 </dependency>
</dependencies>
```

配置如下：

```
server:
 port: 8764
spring:
 # 配置数据源
 datasource:
 driver-class-name: com.mysql.cj.jdbc.Driver
 url: jdbc:mysql://localhost:3306/my.shiro?serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8&zeroDateTimeBehavior=convertToNull
 username: root
 password: root
 type: com.alibaba.druid.pool.DruidDataSource
 # Redis数据源
 redis:
 host: localhost
 port: 6379
 timeout: 6000
 password: 123456
 jedis:
 pool:
 max-active: 1000 # 连接池最大连接数 (使用负值表示没有限制)
 max-wait: -1 # 连接池最大阻塞等待时间 (使用负值表示没有限制)
 max-idle: 10 # 连接池中的最大空闲连接
 min-idle: 5 # 连接池中的最小空闲连接
 # mybatis-plus相关配置
 mybatis-plus:
 # xml扫描, 多个目录用逗号或者分号分隔 (告诉 Mapper 所对应的 XML 文件位置)
 mapper-locations:classpath:mapper/*.xml
 # 以下配置均有默认值,可以不设置
 global-config:
 db-config:
 # 主键类型 AUTO:"数据库ID自增" INPUT:"用户输入ID", ID_WORKER:"全局唯一ID (数字类型唯一ID)", UUID:"全局唯一ID UUID";
 id-type: auto
 # 字段策略 IGNORED:"忽略判断" NOT_NULL:"非NULL判断" NOT_EMPTY:"非空判断"
 field-strategy: NOT_EMPTY
 # 数据库类型
 db-type: MYSQL
 configuration:
 # 是否开启自动驼峰命名规则映射:从数据库列名到Java属性驼峰命名的类似映射
 map-underscore-to-camel-case: true
 # 返回map时true:当查询数据为空时字段返回为null,false:不加这个查询数据为空时, 字段将被隐藏
 call-setters-on-nulls: true
 # 这个配置会将执行的sql打印出来, 在开发或测试的时候可以用
 log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

### 三. 编写项目基础类

用户实体, Dao, Service 等在这里省略,请参考源码

编写 Exception 类来处理 Shiro 权限拦截异常

```

/**
 * @Description 自定义异常
 * @Author Sans
 * @CreateTime 2019/6/15 22:56
 */
@ControllerAdvice
public class MyShiroException {
 /**
 * 处理Shiro权限拦截异常
 * 如果返回JSON数据格式请加上 @ResponseBody注解
 * @Author Sans
 * @CreateTime 2019/6/15 13:35
 * @Return Map<Object> 返回结果集
 */
 @ResponseBody
 @ExceptionHandler(value = AuthorizationException.class)
 public Map<String, Object> defaultErrorHandler(){
 Map<String, Object> map = new HashMap<>();
 map.put("403", "权限不足");
 return map;
 }
}

```

## 创建 SHA256Util 加密工具

```

/**
 * @Description Sha-256加密工具
 * @Author Sans
 * @CreateTime 2019/6/12 9:27
 */
public class SHA256Util {
 /** 私有构造器 */
 private SHA256Util(){};
 /** 加密算法 */
 public final static String HASH_ALGORITHM_NAME = "SHA-256";
 /** 循环次数 */
 public final static int HASH_ITERATIONS = 15;
 /** 执行加密-采用SHA256和盐值加密 */
 public static String sha256(String password, String salt) {
 return new SimpleHash(HASH_ALGORITHM_NAME, password, salt, HASH_ITERATIONS).toString();
 }
}

```

## 创建 Spring 工具

```

/**
 * @Description Spring上下文工具类
 * @Author Sans
 * @CreateTime 2019/6/17 13:40
 */
@Component
public class SpringUtil implements ApplicationContextAware {
 private static ApplicationContext context;
 /**
 * Spring在bean初始化后会判断是不是ApplicationContextAware的子类
 * 如果该类是,setApplicationContext()方法,会将容器中ApplicationContext作为参数传入进去
 * @Author Sans
 * @CreateTime 2019/6/17 16:58
 */
 @Override
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
 context = applicationContext;
 }
 /**
 * 通过Name返回指定的Bean
 * @Author Sans
 * @CreateTime 2019/6/17 16:03
 */
 public static <T> T getBean(Class<T> beanClass) {
 return context.getBean(beanClass);
 }
}

```

## 创建 Shiro 工具

```

/**
 * @Description Shiro工具类
 * @Author Sans
 * @CreateTime 2019/6/15 16:11
 */
public class ShiroUtils {

 /** 私有构造器 */
 private ShiroUtils() {}

 private static RedisSessionDAO redisSessionDAO = SpringUtil.getBean(RedisSessionDAO.class);

 /**
 * 获取当前用户Session
 * @Author Sans
 * @CreateTime 2019/6/17 17:03
 * @Return SysUserEntity 用户信息
 */
 public static Session getSession() {
 return SecurityUtils.getSubject().getSession();
 }

 /**
 * 用户登出
 * @Author Sans
 * @CreateTime 2019/6/17 17:23
 */
 public static void logout() {
 SecurityUtils.getSubject().logout();
 }

 /**
 * 获取当前用户信息
 */
}

```

```

* @Author Sans
* @CreateTime 2019/6/17 17:03
* @Return SysUserEntity 用户信息
*/
public static SysUserEntity getUserInfo() {
 return (SysUserEntity) SecurityUtils.getSubject().getPrincipal();
}

/**
 * 删除用户缓存信息
* @Author Sans
* @CreateTime 2019/6/17 13:57
* @Param username 用户名称
* @Param isRemoveSession 是否删除Session
* @Return void
*/
public static void deleteCache(String username, boolean isRemoveSession){
 //从缓存中获取Session
 Session session = null;
 Collection<Session> sessions = redisSessionDAO.getActiveSessions();
 SysUserEntity sysUserEntity;
 Object attribute = null;
 for(Session sessionInfo : sessions){
 //遍历Session,找到该用户名称对应的Session
 attribute = sessionInfo.getAttribute(DefaultSubjectContext.PRINCIPALS_SESSION_KEY);
 if (attribute == null) {
 continue;
 }
 sysUserEntity = (SysUserEntity) ((SimplePrincipalCollection) attribute).getPrimaryPrincipal();
 if (sysUserEntity == null) {
 continue;
 }
 if (Objects.equals(sysUserEntity.getUsername(), username)) {
 session=sessionInfo;
 break;
 }
 }
 if (session == null||attribute == null) {
 return;
 }
 //删除session
 if (isRemoveSession) {
 redisSessionDAO.delete(session);
 }
 //删除Cache, 在访问受限接口时会重新授权
 DefaultWebSecurityManager securityManager = (DefaultWebSecurityManager) SecurityUtils.getSecurityManager();
 Authenticator authc = securityManager.getAuthenticator();
 ((LogoutAware) authc).onLogout((SimplePrincipalCollection) attribute);
}
}

```

创建 Shiro 的 SessionId 生成器

```

/**
 * @Description 自定义SessionId生成器
 * @Author Sans
 * @CreateTime 2019/6/11 11:48
 */
public class ShiroSessionIdGenerator implements SessionIdGenerator {
 /**
 * 实现SessionId生成
 * @Author Sans
 * @CreateTime 2019/6/11 11:54
 */
 @Override
 public Serializable generateId(Session session) {
 Serializable sessionId = new JavaUuidSessionIdGenerator().generateId(session);
 return String.format("login_token_%s", sessionId);
 }
}

```

## 四. 编写 Shiro 核心

### 类

创建 Realm 用于授权和认证

```

/**
 * @Description Shiro权限匹配和账号密码匹配
 * @Author Sans
 * @CreateTime 2019/6/15 11:27
 */
public class ShiroRealm extends AuthorizingRealm {
 @Autowired
 private SysUserService sysUserService;
 @Autowired
 private SysRoleService sysRoleService;
 @Autowired
 private SysMenuService sysMenuService;
 /**
 * 授权权限
 * 用户进行权限验证时候Shiro会去缓存中找,如果查不到数据,会执行这个方法去查权限,并放入缓存中
 * @Author Sans
 * @CreateTime 2019/6/12 11:44
 */
 @Override
 protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {
 SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
 SysUserEntity sysUserEntity = (SysUserEntity) principalCollection.getPrimaryPrincipal();
 //获取用户ID
 Long userId = sysUserEntity.getUserId();
 //这里可以进行授权和处理
 Set<String> rolesSet = new HashSet<>();
 Set<String> permsSet = new HashSet<>();
 //查询角色和权限(这里根据业务自行查询)
 List<SysRoleEntity> sysRoleEntityList = sysRoleService.selectSysRoleByUserId(userId);
 for (SysRoleEntity sysRoleEntity:sysRoleEntityList) {
 rolesSet.add(sysRoleEntity.getRoleName());
 List<SysMenuEntity> sysMenuEntityList = sysMenuService.selectSysMenuByRoleId(sysRoleEntity.getRoleId());
 for (SysMenuEntity sysMenuEntity :sysMenuEntityList) {
 permsSet.add(sysMenuEntity.getPerms());
 }
 }
 //将查到的权限和角色分别传入authorizationInfo中
 }
}

```

```
//设置权限和角色
authorizationInfo.setStringPermissions(permsSet);
authorizationInfo.setRoles(rolesSet);
return authorizationInfo;
}

/**
 *身份认证
 * @Author Sans
 * @CreateTime 2019/6/12 12:36
 */
@Override
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {
 //获取用户的输入的账号
 String username = (String) authenticationToken.getPrincipal();
 //通过username从数据库中查找 User对象，如果找到进行验证
 //实际项目中,这里可以根据实际情况做缓存,如果不做,Shiro自己也是有时间间隔机制,2分钟内不会重复执行该方法
 SysUserEntity user = sysUserService.selectUserByName(username);
 //判断账号是否存在
 if (user == null) {
 throw new AuthenticationException();
 }
 //判断账号是否被冻结
 if (user.getState() == null || user.getState().equals("PROHIBIT")) {
 throw new LockedAccountException();
 }
 //进行验证
 SimpleAuthenticationInfo authenticationInfo = new SimpleAuthenticationInfo(
 user, //用户名
 user.getPassword(), //密码
 ByteSource.Util.bytes(user.getSalt()), //设置盐值
 getName()
);
 //验证成功开始踢人(清除缓存和Session)
 ShiroUtils.deleteCache(username, true);
 return authenticationInfo;
}
}
```

创建 SessionManager 类

```

/**
 * @Description 自定义获取Token
 * @Author Sans
 * @CreateTime 2019/6/13 8:34
 */
public class ShiroSessionManager extends DefaultWebSessionManager {
 //定义常量
 private static final String AUTHORIZATION = "Authorization";
 private static final String REFERENCED_SESSION_ID_SOURCE = "Stateless request";
 //重写构造器
 public ShiroSessionManager() {
 super();
 this.setDeleteInvalidSessions(true);
 }
 /**
 * 重写方法实现从请求头获取Token便于接口统一
 * 每次请求进来,Shiro会去从请求头找Authorization这个key对应的Value(Token)
 * @Author Sans
 * @CreateTime 2019/6/13 8:47
 */
 @Override
 public Serializable getSessionId(ServletRequest request, ServletResponse response) {
 String token = WebUtils.toHttp(request).getHeader(AUTHORIZATION);
 //如果请求头中存在token 则从请求头中获取token
 if (!StringUtils.isEmpty(token)) {
 request.setAttribute(ShiroHttpServletRequest.REFERENCED_SESSION_ID_SOURCE, REFERENCED_SESSION_ID_SOURCE);
 request.setAttribute(ShiroHttpServletRequest.REFERENCED_SESSION_ID, token);
 request.setAttribute(ShiroHttpServletRequest.REFERENCED_SESSION_ID_IS_VALID, Boolean.TRUE);
 return token;
 } else {
 // 这里禁用掉Cookie获取方式
 // 按默认规则从Cookie取Token
 // return super.getSessionId(request, response);
 return null;
 }
 }
}

```

## 创建 ShiroConfig 配置类

```

/**
 * @Description Shiro配置类
 * @Author Sans
 * @CreateTime 2019/6/10 17:42
 */
@Configuration
public class ShiroConfig {

 private final String CACHE_KEY = "shiro:cache:";
 private final String SESSION_KEY = "shiro:session:";
 private final int EXPIRE = 1800;

 //Redis配置
 @Value("${spring.redis.host}")
 private String host;
 @Value("${spring.redis.port}")
 private int port;
 @Value("${spring.redis.timeout}")
 private int timeout;
 @Value("${spring.redis.password}")
 private String password;

 /**
 */
}

```

```
*开启Shiro-aop注解支持
* @Attention 使用代理方式所以需要开启代码支持
* @Author Sans
* @CreateTime 2019/6/12 8:38
*/
@Bean
public AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor(SecurityManager securityManager) {
 AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor = new AuthorizationAttributeSourceAdvisor();
 authorizationAttributeSourceAdvisor.setSecurityManager(securityManager);
 return authorizationAttributeSourceAdvisor;
}

/**
 * Shiro基础配置
 * @Author Sans
 * @CreateTime 2019/6/12 8:42
 */
@Bean
public ShiroFilterFactoryBean shiroFilterFactory(SecurityManager securityManager){
 ShiroFilterFactoryBean shiroFilterFactoryBean = new ShiroFilterFactoryBean();
 shiroFilterFactoryBean.setSecurityManager(securityManager);
 Map<String, String> filterChainDefinitionMap = new LinkedHashMap<>();
 // 注意过滤器配置顺序不能颠倒
 // 配置过滤:不会被拦截的链接
 filterChainDefinitionMap.put("/static/**", "anon");
 filterChainDefinitionMap.put("/userLogin/**", "anon");
 filterChainDefinitionMap.put("/**", "authc");
 // 配置shiro默认登录界面地址, 前后端分离中登录界面跳转应由前端路由控制, 后台仅返回json数据
 shiroFilterFactoryBean.setLoginUrl("/userLogin/unauth");
 shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefinitionMap);
 return shiroFilterFactoryBean;
}

/**
 * 安全管理器
 * @Author Sans
 * @CreateTime 2019/6/12 10:34
 */
@Bean
public SecurityManager securityManager() {
 DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();
 // 自定义Session管理
 securityManager.setSessionManager(sessionManager());
 // 自定义Cache实现
 securityManager.setCacheManager(cacheManager());
 // 自定义Realm验证
 securityManager.setRealm(shiroRealm());
 return securityManager;
}

/**
 * 身份验证器
 * @Author Sans
 * @CreateTime 2019/6/12 10:37
 */
@Bean
public ShiroRealm shiroRealm() {
 ShiroRealm shiroRealm = new ShiroRealm();
 shiroRealm.setCredentialsMatcher(hashedCredentialsMatcher());
 return shiroRealm;
}

/**
 * 凭证匹配器
 */
```

```
* 将密码校验交给Shiro的SimpleAuthenticationInfo进行处理,在这里做匹配配置
* @Author Sans
* @CreateTime 2019/6/12 10:48
*/
@Bean
public HashedCredentialsMatcher hashedCredentialsMatcher() {
 HashedCredentialsMatcher shaCredentialsMatcher = new HashedCredentialsMatcher();
 // 散列算法:这里使用SHA256算法;
 shaCredentialsMatcher.setHashAlgorithmName(SHA256Util.HASH_ALGORITHM_NAME);
 // 散列的次数, 比如散列两次, 相当于md5(md5(""));
 shaCredentialsMatcher.setHashIterations(SHA256Util.HASH_ITERATIONS);
 return shaCredentialsMatcher;
}

/***
 * 配置Redis管理器
 * @Attention 使用的是shiro-redis开源插件
 * @Author Sans
 * @CreateTime 2019/6/12 11:06
 */
@Bean
public RedisManager redisManager() {
 RedisManager redisManager = new RedisManager();
 redisManager.setHost(host);
 redisManager.setPort(port);
 redisManager.setTimeout(timeout);
 redisManager.setPassword(password);
 return redisManager;
}

/***
 * 配置Cache管理器
 * 用于往Redis存储权限和角色标识
 * @Attention 使用的是shiro-redis开源插件
 * @Author Sans
 * @CreateTime 2019/6/12 12:37
 */
@Bean
public RedisCacheManager cacheManager() {
 RedisCacheManager redisCacheManager = new RedisCacheManager();
 redisCacheManager.setRedisManager(redisManager());
 redisCacheManager.setKeyPrefix(CACHE_KEY);
 // 配置缓存的话要求放在session里面的实体类必须有个id标识
 redisCacheManager.setPrincipalIdFieldName("userId");
 return redisCacheManager;
}

/***
 * SessionID生成器
 * @Author Sans
 * @CreateTime 2019/6/12 13:12
 */
@Bean
public ShiroSessionIdGenerator sessionIdGenerator(){
 return new ShiroSessionIdGenerator();
}

/***
 * 配置RedisSessionDAO
 * @Attention 使用的是shiro-redis开源插件
 * @Author Sans
 * @CreateTime 2019/6/12 13:44
 */
@Bean
```

```

@Buill
public RedisSessionDAO redisSessionDAO() {
 RedisSessionDAO redisSessionDAO = new RedisSessionDAO();
 redisSessionDAO.setRedisManager(redisManager());
 redisSessionDAO.setSessionIdGenerator(sessionIdGenerator());
 redisSessionDAO.setKeyPrefix(SESSION_KEY);
 redisSessionDAO.setExpire(expire);
 return redisSessionDAO;
}

/**
 * 配置Session管理器
 * @Author Sans
 * @CreateTime 2019/6/12 14:25
 */
@Bean
public SessionManager sessionManager() {
 ShiroSessionManager shiroSessionManager = new ShiroSessionManager();
 shiroSessionManager.setSessionDAO(redisSessionDAO());
 return shiroSessionManager;
}
}

```

## 五.实现权限控制

Shiro 可以用代码或者注解来控制权限,通常我们使用注解控制,不仅简单方便,而且更加灵活.Shiro 注解一共有五个:

注解名称	说明
RequiresAuthentication	使用该注解标注的类,方法等在访问时,当前 Subject 必须在当前 session 中已经过认证.
RequiresGuest	使用该注解标注的类,方法等在访问时,当前 Subject 可以是"guest"身份,不需要经过认证或者在原先的 session 中存在记录.
RequiresUser	验证用户是否被记忆,有两种含义一种是成功登录的(subject.isAuthenticated()结果为 true);另外一种是被记忆的(subject.isRemembered()结果为 true).
RequiresPermissions	当前 Subject 需要拥有某些特定的权限时,才能执行被该注解标注的方法.如果没有权限,则方法不会执行还会抛出 AuthorizationException 异常.
RequiresRoles	当前 Subject 必须拥有所有指定的角色时,才能访问被该注解标注的方法.如果没有角色,则方法不会执行还会抛出 AuthorizationException 异常.

一般情况下我们在项目中做权限控制,使用最多的是 **RequiresPermissions** 和 **RequiresRoles** 允许存在多个角色和权限,默认逻辑是 AND,也就是同时拥有这些才可以访问方法,可以在注解中以参数的形式设置成 OR

### 示例

```

//拥有一个角色就可以访问
@RequiresRoles(value={"ADMIN", "USER"},logical = Logical.OR)
//拥有所有权限才可以访问
@RequiresPermissions(value={"sys:user:info", "sys:role:info"},logical = Logical.AND)

```

使用顺序:Shiro 注解是存在顺序的,当多个注解在一个方法上的时候,会逐个检查,知道全部通过为止,默认拦截顺序是: **RequiresRoles->RequiresPermissions->RequiresAuthentication->RequiresUser->RequiresGuest**

## 示例

```
//拥有ADMIN角色同时还要有sys:role:info权限
@RequiresRoles(value={"ADMIN"})
@RequiresPermissions("sys:role:info")
```

## 创建 UserRoleController 角色拦截测试类

```
/**
 * @Description 角色测试
 * @Author Sans
 * @CreateTime 2019/6/19 11:38
 */
@RestController
@RequestMapping("/role")
public class UserRoleController {

 @Autowired
 private SysUserService sysUserService;
 @Autowired
 private SysRoleService sysRoleService;
 @Autowired
 private SysMenuService sysMenuService;
 @Autowired
 private SysRoleMenuService sysRoleMenuService;

 /**
 * 管理员角色测试接口
 * @Author Sans
 * @CreateTime 2019/6/19 10:38
 * @Return Map<String, Object> 返回结果
 */
 @RequestMapping("/getAdminInfo")
 @RequiresRoles("ADMIN")
 public Map<String, Object> getAdminInfo(){
 Map<String, Object> map = new HashMap<>();
 map.put("code", 200);
 map.put("msg", "这里是只有管理员角色能访问的接口");
 return map;
 }

 /**
 * 用户角色测试接口
 * @Author Sans
 * @CreateTime 2019/6/19 10:38
 * @Return Map<String, Object> 返回结果
 */
 @RequestMapping("/getUserInfo")
 @RequiresRoles("USER")
 public Map<String, Object> getUserInfo(){
 Map<String, Object> map = new HashMap<>();
 map.put("code", 200);
 map.put("msg", "这里是只有用户角色能访问的接口");
 return map;
 }

 /**
 * 角色测试接口
 * @Author Sans
 * @CreateTime 2019/6/19 10:38
 * @Return Map<String, Object> 返回结果
 */
 @RequestMapping("/getRoleInfo")
 @RequiresRoles(value={"ADMIN", "USER"}, logical = Logical.OR)
```

```

@RequiresUser
public Map<String, Object> getRoleInfo(){
 Map<String, Object> map = new HashMap<>();
 map.put("code", 200);
 map.put("msg", "这里是只要有ADMIN或者USER角色能访问的接口");
 return map;
}

/**
 * 登出(测试登出)
 * @Author Sans
 * @CreateTime 2019/6/19 10:38
 * @Return Map<String, Object> 返回结果
 */
@RequestMapping("/getLogout")
@RequiresUser
public Map<String, Object> getLogout(){
 ShiroUtils.logout();
 Map<String, Object> map = new HashMap<>();
 map.put("code", 200);
 map.put("msg", "登出");
 return map;
}
}

```

## 创建 UserMenuController 权限拦截测试类

```

/**
 * @Description 权限测试
 * @Author Sans
 * @CreateTime 2019/6/19 11:38
 */
@RestController
@RequestMapping("/menu")
public class UserMenuController {

 @Autowired
 private SysUserService sysUserService;
 @Autowired
 private SysRoleService sysRoleService;
 @Autowired
 private SysMenuService sysMenuService;
 @Autowired
 private SysRoleMenuService sysRoleMenuService;

 /**
 * 获取用户信息集合
 * @Author Sans
 * @CreateTime 2019/6/19 10:36
 * @Return Map<String, Object> 返回结果
 */
 @RequestMapping("/getUserInfoList")
 @RequiresPermissions("sys:user:info")
 public Map<String, Object> getUserInfoList(){
 Map<String, Object> map = new HashMap<>();
 List<SysUserEntity> sysUserEntityList = sysUserService.list();
 map.put("sysUserEntityList", sysUserEntityList);
 return map;
 }

 /**
 * 获取角色信息集合
 * @Author Sans
 * @CreateTime 2019/6/19 10:37
 */
 @RequestMapping("/getRoleInfoList")
 @RequiresPermissions("sys:role:info")
 public Map<String, Object> getRoleInfoList(){
 Map<String, Object> map = new HashMap<>();
 List<SysRoleEntity> sysRoleEntityList = sysRoleService.list();
 map.put("sysRoleEntityList", sysRoleEntityList);
 return map;
 }
}

```

```
* @CreateTime 2019/6/19 10:37
* @Return Map<String, Object> 返回结果
*/
@RequestMapping("/getRoleInfoList")
@RequiresPermissions("sys:role:info")
public Map<String, Object> getRoleInfoList(){
 Map<String, Object> map = new HashMap<>();
 List<SysRoleEntity> sysRoleEntityList = sysRoleService.list();
 map.put("sysRoleEntityList", sysRoleEntityList);
 return map;
}

/**
 * 获取权限信息集合
 * @Author Sans
 * @CreateTime 2019/6/19 10:38
 * @Return Map<String, Object> 返回结果
*/
@RequestMapping("/getMenuInfoList")
@RequiresPermissions("sys:menu:info")
public Map<String, Object> getMenuInfoList(){
 Map<String, Object> map = new HashMap<>();
 List<SysMenuEntity> sysMenuEntityList = sysMenuService.list();
 map.put("sysMenuEntityList", sysMenuEntityList);
 return map;
}

/**
 * 获取所有数据
 * @Author Sans
 * @CreateTime 2019/6/19 10:38
 * @Return Map<String, Object> 返回结果
*/
@RequestMapping("/ getInfoAll")
@RequiresPermissions("sys:info:all")
public Map<String, Object> getInfoAll(){
 Map<String, Object> map = new HashMap<>();
 List<SysUserEntity> sysUserEntityList = sysUserService.list();
 map.put("sysUserEntityList", sysUserEntityList);
 List<SysRoleEntity> sysRoleEntityList = sysRoleService.list();
 map.put("sysRoleEntityList", sysRoleEntityList);
 List<SysMenuEntity> sysMenuEntityList = sysMenuService.list();
 map.put("sysMenuEntityList", sysMenuEntityList);
 return map;
}

/**
 * 添加管理员角色权限(测试动态权限更新)
 * @Author Sans
 * @CreateTime 2019/6/19 10:39
 * @Param username 用户ID
 * @Return Map<String, Object> 返回结果
*/
@RequestMapping("/addMenu")
public Map<String, Object> addMenu(){
 //添加管理员角色权限
 SysRoleMenuEntity sysRoleMenuEntity = new SysRoleMenuEntity();
 sysRoleMenuEntity.setMenuId(4L);
 sysRoleMenuEntity.setRoleId(1L);
 sysRoleMenuService.save(sysRoleMenuEntity);
 //清除缓存
 String username = "admin";
 ShiroUtils.deleteCache(username, false);
 Map<String, Object> map = new HashMap<>();
}
```

```
 map.put("code",200);
 map.put("msg","权限添加成功");
 return map;
 }
}
```

创建 UserLoginController 登录类

```

/**
 * @Description 用户登录
 * @Author Sans
 * @CreateTime 2019/6/17 15:21
 */
@RestController
@RequestMapping("/userLogin")
public class UserLoginController {

 /**
 * 登录
 * @Author Sans
 * @CreateTime 2019/6/20 9:21
 */
 @RequestMapping("/login")
 public Map<String, Object> login(@RequestBody SysUserEntity sysUserEntity){
 Map<String, Object> map = new HashMap<>();
 //进行身份验证
 try{
 //验证身份和登陆
 Subject subject = SecurityUtils.getSubject();
 UsernamePasswordToken token = new UsernamePasswordToken(sysUserEntity.getUsername(), sysUserEntity.getPassword());
 //验证成功进行登录操作
 subject.login(token);
 }catch (IncorrectCredentialsException e) {
 map.put("code",500);
 map.put("msg","用户不存在或者密码错误");
 return map;
 }catch (LockedAccountException e) {
 map.put("code",500);
 map.put("msg","登录失败，该用户已被冻结");
 return map;
 }catch (AuthenticationException e) {
 map.put("code",500);
 map.put("msg","该用户不存在");
 return map;
 }catch (Exception e) {
 map.put("code",500);
 map.put("msg","未知异常");
 return map;
 }
 map.put("code",0);
 map.put("msg", "登录成功");
 map.put("token", ShiroUtils.getSession().getId().toString());
 return map;
 }

 /**
 * 未登录
 * @Author Sans
 * @CreateTime 2019/6/20 9:22
 */
 @RequestMapping("/unauth")
 public Map<String, Object> unauth(){
 Map<String, Object> map = new HashMap<>();
 map.put("code",500);
 map.put("msg", "未登录");
 return map;
 }
}

```

登录成功后会返回 TOKEN,因为是单点登录,再次登陆的话会返回新的 TOKEN,之前 Redis 的 TOKEN 就会失效了

The screenshot shows a POST request to `http://localhost:8764/userLogin/login`. The request body is JSON with the following content:

```
1 - {
2 "username": "admin",
3 "password": "123456"
4 }
```

The response status is 200 OK, with a response body containing:

```
1 - {
2 "msg": "登录成功",
3 "code": 0,
4 "token": "login_token_846d9f3f-adaf-48dd-b0ce-be223d37a187"
5 }
```

当第一次访问接口后我们可以看到缓存中已经有权限数据了,在次访问接口的时候,Shiro 会直接去缓存中拿取权限,注意访问接口时候要设置请求头.

The screenshot shows a GET request to `http://localhost:8764/menu/getUserInfoList`. The Headers tab shows the following configuration:

KEY	VALUE	DESCRIPTION
Content-Type	application/json	
Authorization	login_token_846d9f3f-adaf-48dd-b0ce-be223d37a187	
Key	Value	Description

The response status is 200 OK, with a response body containing:

```
1 - {
2 "sysUserEntityList": [
3 {
4 "userId": 1,
5 "username": "admin",
6 "password": "a1b0a08ad5de12e0f94762cb116c447e80c784d8aa2c6625263f7f3436cdd583",
7 "salt": "RvP3UDln3Q2QsyCZYH",
8 "state": "NORMAL"
9 },
10 {
11 "userId": 2,
12 "username": "user",
13 "password": "376eb5d2698c804ee83594fe8b0217f03ad138a046f7fa42b44c232c2e5e2b38",
14 "salt": "OV1rD37bDUK1cFRB10qG",
15 "state": "NORMAL"
16 }
17]
18 }
```

ADMIN 这个号现在没有 sys:info:all 这个权限的所以无法访问 `getInfoAll` 接口我们要动态分配权限后,要清掉缓存,在访问接口时候,Shiro 会去重新执行授权方法,之后再次把权限和角色数据放入缓存中

The screenshot shows the Redis browser interface. Under the key `我的redis`, there is a database `db0` which contains a `shiro` folder. Inside `shiro`, there is a `cache` folder which is currently empty. There is also a `session` folder which contains a single entry: `shiro:session:login_token_846d9f3f-adaf-48dd-b0ce-be223d37a187`.

访问添加权限测试接口,因为是测试,我把增加权限的用户 ADMIN 写死在里面了权限添加后,调用工具类清掉缓存,我们可以发现,Redis 中已经没有缓存了

http://localhost:8764/menu/addMenu

GET http://localhost:8764/menu/addMenu

Params Authorization Headers (8) Body Pre-request Script Tests Cookies Code Comments (0)

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
username	admin			
Key	Value	Description		

Body Cookies (1) Headers (3) Test Results Status: 200 OK Time: 1222 ms Size: 169 B Download

Pretty Raw Preview JSON

```

1 ↴ {
2 "msg": "权限添加成功",
3 "code": 200
4 }

```

我的redis

- db0 (1/1)
  - shiro (1)
    - session (1)
      - shiro:session:login\_token\_846d9f3f-adaf-48dd-b0ce-be...

再次访问 `getInfoAll` 接口因为缓存中没有数据,Shiro 会重新授权查询权限,拦截通过

GET http://localhost:8764/menu/getInfoAll

Params Authorization Headers (10) Body Pre-request Script Tests Cookies Code Comments (0)

Headers (2)

KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
Content-Type	application/json				
Authorization	login_token_846d9f3f-adaf-48dd-b0ce-be223d37a187				
Key	Value	Description			

Temporary Headers (8)

Body Cookies (1) Headers (3) Test Results Status: 200 OK Time: 111 ms Size: 832 B Download

Pretty Raw Preview JSON

```

5 {
6 "username": "admin",
7 "password": "a1bd09ad5dea12e0f94762cb116c447e80c784d8aa2c6625263f7f3436cdd583",
8 "salt": "RvP3UID2n3Q02sycZVvH",
9 "state": "NORMAL"
10 },
11 {
12 "userId": 2,
13 "username": "user",
14 "password": "376eb5d2698c004ee83594fe8b0217f03ad138a046f7fa42b44c232c2e5e2b38",
15 "salt": "OV1rD37bDUKNCFRB10qG",
16 "state": "NORMAL"
17 },
18 "sysMenuEntityList": [
19 {
20 "menuId": 1,
21 "name": "查看用户列表",
22 "perms": "sys:user:info"
23 }

```

## 六.项目源码

<https://github.com/xuyulong2017/my-java-demo>

谢谢大家阅读,如果喜欢,请收藏点赞,多给些 star,文章不足之处,也请给出宝贵意见.

- END -

## 推荐阅读

1. 实现接口幂等性校验
2. 全面了解 Nginx 主要应用场景
3. Github标星10.8K!Java 实战博客项目分享
4. 什么是一致性 Hash 算法?
5. 团队开发中 Git 最佳实践



喜欢文章, 点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot 整合 Spring-cache：让你的网站速度飞起来

Java后端 2019-11-27

以下文章来源于Java碎碎念，作者Java碎碎念



## Java碎碎念

专注但不限于Java，每天进步一点点！

计算机领域有人说过一句名言：“计算机科学领域的任何问题都可以通过增加一个中间层来解决”，今天我们就用 Spring-cache 给网站添加一层缓存，让你的网站速度飞起来。

### 本文目录

一、Spring Cache介绍  
二、缓存注解介绍  
三、Spring Boot+Cache实战  
    1、pom.xml引入jar包  
    2、启动类添加  
    3、@EnableCaching注解  
    4、配置数据库和redis连接  
    5、配置CacheManager  
    6、使用缓存注解  
    7、查看缓存效果  
    8、注意事项

### 一、Spring Cache介绍

Spring 3.1引入了基于注解的缓存(cache)技术，它本质上是一个对缓存使用的抽象，通过在既有代码中添加少量它定义的各种注解，就能够达到缓存方法的效果。

Spring Cache接口为缓存的组件规范定义，包含缓存的各种操作集合，并提供了各种xxxCache的实现，如 RedisCache, EhCacheCache, ConcurrentMapCache等；

项目整合Spring Cache后每次调用需要缓存功能的方法时，Spring会检查指定参数的指定的目标方法是否已经被调用过，如果有就直接从缓存中获取结果，没有就调用方法并把结果放到缓存。

### 二、缓存注解介绍

对于缓存声明，Spring的缓存提供了一组java注解：

- `@CacheConfig`:设置类级别上共享的一些常见缓存设置。
- `@Cacheable`:触发缓存写入。
- `@CacheEvict`:触发缓存清除。
- `@Caching` 将多种缓存操作分组

- `@CachePut`:更新缓存(不会影响到方法的运行)。

## `@CacheConfig`

该注解是可以将缓存分类，它是类级别的注解方式。我们可以这么使用它。

这样的话，`UserServiceImpl`的所有缓存注解例如`@Cacheable`的`value`值就都为`user`。

```
@CacheConfig(cacheNames = "user")
@Service
public class UserServiceImpl implements UserService {}
```

## `@Cacheable`

一般用于查询操作，根据key查询缓存。

1. 如果key不存在，查询db，并将结果更新到缓存中。
2. 如果key存在，直接查询缓存中的数据。

```
//查询数据库后 数据添加到缓存
 @Override
 @Cacheable(cacheNames = "cacheManager", key = "'USER:'+#id", unless = "#result == null")
 public User getUser(Integer id) {
 return repository.getUser(id);
 }
```

## `@CachePut`

`@CachePut`标注的方法在执行前不会去检查缓存中是否存在，而是每次都会执行该方法，并将执行结果以键值对的形式存入指定的缓存中。

```
//修改数据后更新缓存
 @Override
 @CachePut(cacheNames = "cacheManager", key = "'USER:'+#updateUser.id", unless = "#result == null")
 public User updateUser(User updateUser) {
 return repository.save(updateUser);
 }
```

## `@CacheEvict`

根据key删除缓存中的数据。`allEntries=true`表示删除缓存中的所有数据。

```
//清除一条缓存，key为要清空的数据
 @Override
 @CacheEvict(cacheNames = "cacheManager", key = "'USER:'+#id")
 public void deleteUser(Integer id) {
 repository.deleteByld(id);
 }
```

#### 1、pom.xml引入jar包

```
<!-- 引入缓存 starter -->
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<!-- 引入 redis -->
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-redis</artifactId>
 >
</dependency>
```

#### 2、启动类添加@EnableCaching注解

@EnableCaching注解是spring framework中的注解驱动的缓存管理功能，当你在配置类(@Configuration)上使用@EnableCaching注解时，会触发一个post processor，这会扫描每一个spring bean，查看是否已经存在注解对应的缓存。如果找到了，就会自动创建一个代理拦截方法调用，使用缓存的bean执行处理。

启动类部分代码如下：

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

#### 3、配置数据库和redis连接

application.properties部分配置如下：

```
#配置数据源信息
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://192.168.1.1:3306/test
spring.datasource.username=root
spring.datasource.password=1234

#配置jpa
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jackson.serialization.indent_output=true

Redis服务器地址
spring.redis.host=192.168.1.1

database
spring.redis.database = 1

Redis服务器连接端口 使用默认端口6379可以省略配置
spring.redis.port=6379

Redis服务器连接密码（默认为空）
spring.redis.password=1234

连接池最大连接数（如果配置<=0，则没有限制）
spring.redis.jedis.pool.max-active=8
```

## 4、配置CacheManager

WebConfig.java部分配置如下：

```
@Bean
public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {
 //缓存配置对象
 RedisCacheConfiguration redisCacheConfiguration = RedisCacheConfiguration.defaultCacheConfig();

 redisCacheConfiguration = redisCacheConfiguration.entryTtl(Duration.ofMinutes(30L)) //设置缓存的默认超时时间：30分钟
 .disableCachingNullValues() //如果是空值，不缓存
 .serializeKeysWith(RedisSerializationContext.SerializationPair.fromSerializer(keySerializer())) //设置key序列化器
 .serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerializer((valueSerializer()))); //设置value序列化器

 return RedisCacheManager
 .builder(RedisCacheWriter.nonLockingRedisCacheWriter(redisConnectionFactory))
 .cacheDefaults(redisCacheConfiguration).build();
}
```

## 5、使用缓存注解

UserServiceImpl.java中使用缓存注解示例如下：

```

//查询数据库后数据添加到缓存

@Override
@Cacheable(cacheNames = "cacheManager", key = "'USER:'+#id", unless = "#result == null")
public User getUser(Integer id) {
 return repository.getUser(id);
}

//清除一条缓存, key为要清空的数据

@Override
@CacheEvict(cacheNames = "cacheManager", key = "'USER:'+#id")
public void deleteUser(Integer id) {
 repository.deleteById(id);
}

//修改数据后更新缓存

@Override
@CachePut(cacheNames = "cacheManager", key = "'USER:'+#updateUser.id", unless = "#result == null")
public User updateUser(User updateUser) {
 return repository.save(updateUser);
}

```

## 6、查看缓存效果

启动服务后, 访问两次`http://localhost:8090/getUser/2`接口, 从打印日志可以看到, 第一次请求打印了sql说明查询了数据库, 耗时960, 而第二次直接查询的缓存耗时66, 增加缓存后速度提升非常明显。

postman访问截图

```

2019-08-21 21:03:02.861 INFO 13816 --- [nio-8090-exec-1] com.example.demo.aop.LogRecordAspect : ****
2019-08-21 21:03:02.861 INFO 13816 --- [nio-8090-exec-1] com.example.demo.aop.LogRecordAspect : 请求开始, URI: /getUser/2, method: GET
2019-08-21 21:03:02.980 INFO 13816 --- [nio-8090-exec-1] io.lettuce.core.EpollProvider : Starting without optional epoll library
2019-08-21 21:03:02.982 INFO 13816 --- [nio-8090-exec-1] io.lettuce.core.KqueueProvider : Starting without optional kqueue library
Hibernate: select user0_.id as id1_0_, user0_.name as name2_0_, user0_.password as password3_0_ from user user0_ where user0_.id=?
2019-08-21 21:03:03.821 INFO 13816 --- [nio-8090-exec-1] com.example.demo.aop.LogRecordAspect : ****
2019-08-21 21:03:08.934 INFO 13816 --- [nio-8090-exec-2] com.example.demo.aop.LogRecordAspect : 请求结束, URI: /getUser/2,耗时=960
2019-08-21 21:03:08.934 INFO 13816 --- [nio-8090-exec-2] com.example.demo.aop.LogRecordAspect : ****
2019-08-21 21:03:09.000 INFO 13816 --- [nio-8090-exec-2] com.example.demo.aop.LogRecordAspect : 请求开始, URI: /getUser/2, method: GET
2019-08-21 21:03:09.000 INFO 13816 --- [nio-8090-exec-2] com.example.demo.aop.LogRecordAspect : 请求结束, URI: /getUser/2,耗时=66

```

日志截图

## 7、注意事项

Spring cache是基于Spring Aop来动态代理机制来对方法的调用进行切面，这里关键点是对象的引用问题，如果对象的方法是内部调用（即 this 引用）而不是外部引用，则会导致 proxy 失效，那么我们的切面就失效，也就是说上面定义的各种注释包括 @Cacheable、@CachePut 和 @CacheEvict 都会失效。

到此Spring Boot 2.X中整合Spring-cache与Redis功能全部实现，有问题欢迎留言沟通哦！

完整源码地址：<https://github.com/suisui2019/springboot-study>

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot 核心知识，深入剖析！

CHEN川 Java后端 2019-10-24



原文 | [www.jianshu.com/p/83693d3d0a65](http://www.jianshu.com/p/83693d3d0a65)

在过去两三年的Spring生态圈，最让人兴奋的莫过于Spring Boot框架。或许从命名上就能看出这个框架的设计初衷：快速的启动Spring应用。因而Spring Boot应用本质上就是一个基于Spring框架的应用，它是Spring对“定优于配置”理念的最佳实践产物，它能够帮助开发者更快速高效地构建基于Spring生态圈的应用。

## 那Spring Boot有何魔法？

**自动配置、起步依赖、Actuator、命令行界面(CLI)**是Spring Boot最重要的4大核心特性，其中CLI是Spring Boot的可选特性，虽然它功能强大，但也引入了一套不太常规的开发模型，因而这个系列的文章仅关注其它3种特性。如文章标题，本文是这个系列的第一部分，将为你打开Spring Boot的大门，重点为你剖析其启动流程以及自动配置实现原理。要掌握这部分核心内容，理解一些Spring框架的基础知识，将会让你事半功倍。

### 一、抛砖引玉：探索Spring IoC容器

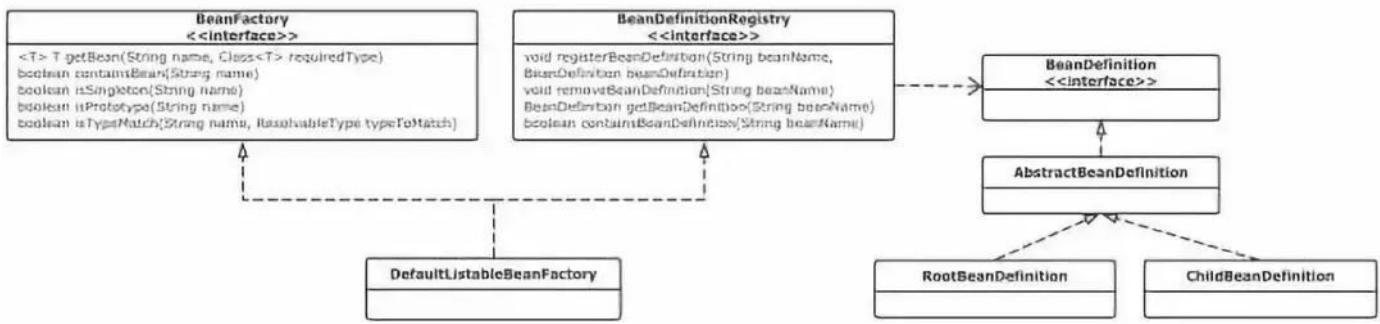
如果有看过 `SpringApplication.run()` 方法的源码，Spring Boot冗长无比的启动流程一定会让你抓狂，透过现象看本质，`SpringApplication`只是将一个典型的Spring应用的启动流程进行了扩展，因此，透彻理解Spring容器是打开Spring Boot大门的一把钥匙。

#### 1.1、Spring IoC容器

可以把Spring IoC容器比作一间餐馆，当你来到餐馆，通常会直接招呼服务员：点菜！至于菜的原料是什么？如何用原料把菜做出来？可能你根本就不关心。IoC容器也是一样，你只需要告诉它需要某个bean，它就把对应的实例（instance）扔给你，至于这个bean是否依赖其他组件，怎样完成它的初始化，根本就不需要你关心。

作为餐馆，想要做出菜肴，得知道菜的原料和菜谱，同样地，IoC容器想要管理各个业务对象以及它们之间的依赖关系，需要通过某种途径来记录和管理这些信息。`BeanDefinition`对象就承担了这个责任：容器中的每一个bean都会有一个对应的`BeanDefinition`实例，该实例负责保存bean对象的所有必要信息，包括bean对象的class类型、是否是抽象类、构造方法和参数、其它属性等等。当客户端向容器请求相对对象时，容器就会通过这些信息为客户端返回一个完整可用的bean实例。

原材料已经准备好（把`BeanDefinition`看着原料），开始做菜吧，等等，你还需要一份菜谱，`BeanDefinitionRegistry`和`BeanFactory`就是这份菜谱，`BeanDefinitionRegistry`抽象出bean的注册逻辑，而`BeanFactory`则抽象出了bean的管理逻辑，而各个`BeanFactory`的实现类就具体承担了bean的注册以及管理工作。它们之间的关系就如下图：



BeanFactory、BeanDefinitionRegistry关系图（来自：Spring揭秘）

DefaultListableBeanFactory 作为一个比较通用的BeanFactory实现，它同时也实现了BeanDefinitionRegistry接口，因此它就承担了 Bean的注册管理工作。从图中也可以看出，BeanFactory接口中主要包含getBean、containBean、getType、getAliases等管理bean的方法，而BeanDefinitionRegistry接口则包含registerBeanDefinition、removeBeanDefinition、getBeanDefinition等注册管理 BeanDefinition的方法。

下面通过一段简单的代码来模拟BeanFactory底层是如何工作的：

```

// 默认容器实现
DefaultListableBeanFactory beanRegistry = newDefaultListableBeanFactory();
// 根据业务对象构造相应的BeanDefinition
AbstractBeanDefinition definition = newRootBeanDefinition(Business.class,true);
// 将bean定义注册到容器中
beanRegistry.registerBeanDefinition("beanName",definition);
// 如果有多个bean，还可以指定各个bean之间的依赖关系
//
// 然后可以从容器中获取这个bean的实例
// 注意：这里的beanRegistry其实实现了BeanFactory接口，所以可以强转，
// 单纯的BeanDefinitionRegistry是无法强制转换到BeanFactory类型的
BeanFactory container = (BeanFactory)beanRegistry;
Business business = (Business)container.getBean("beanName");

```

这段代码仅为了说明BeanFactory底层的大致工作流程，实际情况会更加复杂，比如bean之间的依赖关系可能定义在外部配置文件 (XML/Properties)中、也可能是注解方式。Spring IoC容器的整个工作流程大致可以分为两个阶段：

## ①、容器启动阶段

容器启动时，会通过某种途径加载 ConfigurationMetaData。除了代码方式比较直接外，在大部分情况下，容器需要依赖某些工具类，比如：BeanDefinitionReader，BeanDefinitionReader会对加载的 ConfigurationMetaData 进行解析和分析，并将分析后的信息组装为相应的BeanDefinition，最后把这些保存了bean定义的BeanDefinition，注册到相应的BeanDefinitionRegistry，这样容器的启动工作就完成了。这个阶段主要完成一些准备性工作，更侧重于bean对象管理信息的收集，当然一些验证性或者辅助性的工作也在这一阶段完成。

来看一个简单的例子吧，过往，所有的bean都定义在XML配置文件中，下面的代码将模拟BeanFactory如何从配置文件中加载bean的定义以及依赖关系：

```

// 通常为BeanDefinitionRegistry的实现类，这里以DefaultListableBeanFactory为例
BeanDefinitionRegistry beanRegistry = newDefaultListableBeanFactory();
// XmlBeanDefinitionReader实现了 BeanDefinitionReader接口，用于解析XML文件
XmlBeanDefinitionReader beanDefinitionReader = newXmlBeanDefinitionReaderImpl(beanRegistry);
// 加载配置文件
beanDefinitionReader.loadBeanDefinitions("classpath:spring-bean.xml");
// 从容器中获取bean实例
BeanFactory container = (BeanFactory)beanRegistry;
Business business = (Business)container.getBean("beanName");

```

## ②、Bean的实例化阶段

经过第一阶段，所有bean定义都通过BeanDefinition的方式注册到BeanDefinitionRegistry中，当某个请求通过容器的getBean方法请求某个对象，或者因为依赖关系容器需要隐式的调用getBean时，就会触发第二阶段的活动：容器会首先检查所请求的对象之前是否已经实例化完成。如果没有，则会根据注册的BeanDefinition所提供的信息实例化被请求对象，并为其注入依赖。当该对象装配完毕后，容器会立即将其返回给请求方法使用。

BeanFactory只是Spring IoC容器的一种实现，如果没有特殊指定，它采用采用延迟初始化策略：只有当访问容器中的某个对象时，才对该对象进行初始化和依赖注入操作。而在实际场景下，我们更多的使用另外一种类型的容器：ApplicationContext，它构建在BeanFactory之上，属于更高级的容器，除了具有BeanFactory的所有能力之外，还提供对事件监听机制以及国际化的支持等。它管理的bean，在容器启动时全部完成初始化和依赖注入操作。

## 1.2、Spring容器扩展机制

IoC容器负责管理容器中所有bean的生命周期，而在bean生命周期的不同阶段，Spring提供了不同的扩展点来改变bean的命运。在容器的启动阶段，BeanFactoryPostProcessor允许我们在容器实例化相对对象之前，对注册到容器的BeanDefinition所保存的信息做一些额外的操作，比如修改bean定义的某些属性或者增加其他信息等。

如果要自定义扩展类，通常需要实现 org.springframework.beans.factory.config.BeanFactoryPostProcessor接口，与此同时，因为容器中可能有多个BeanFactoryPostProcessor，可能还需要实现 org.springframework.core.Ordered接口，以保证 BeanFactoryPostProcessor按照顺序执行。Spring提供了为数不多的BeanFactoryPostProcessor实现，我们以 PropertyPlaceholderConfigurer来说明其大致的工作流程。

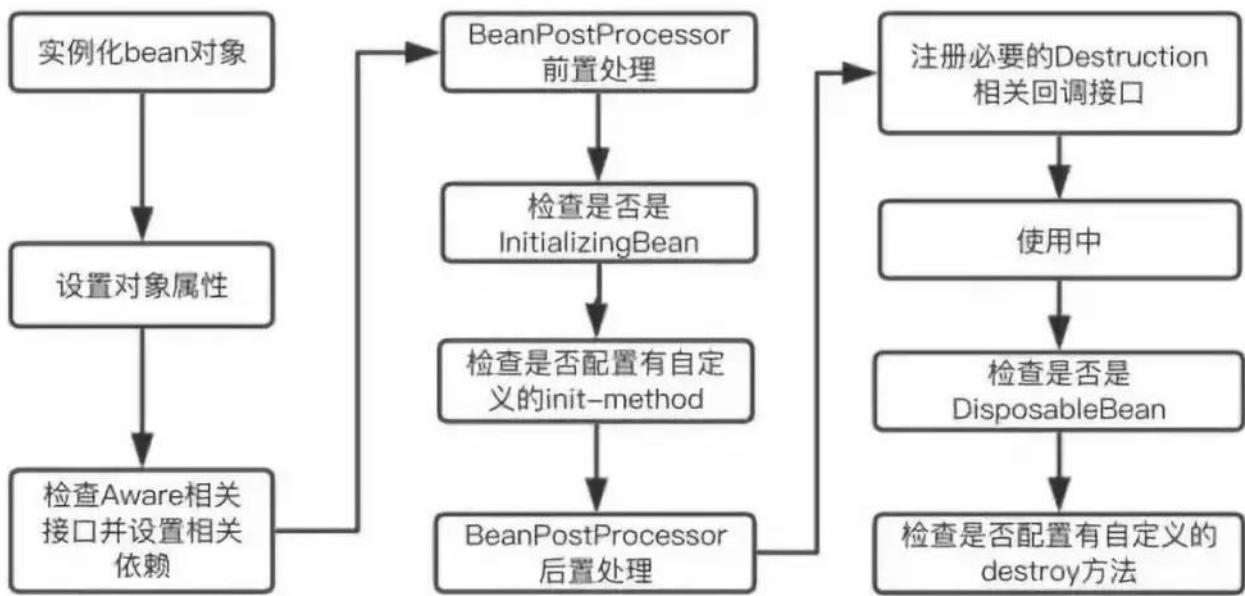
在Spring项目的XML配置文件中，经常可以看到许多配置项的值使用占位符，而将占位符所代表的值单独配置到独立的properties文件，这样可以将散落在不同XML文件中的配置集中管理，而且也方便运维根据不同的环境进行配置不同的值。这个非常实用的功能就是由 PropertyPlaceholderConfigurer负责实现的。

根据前文，当BeanFactory在第一阶段加载完所有配置信息时，BeanFactory中保存的对象的属性还是以占位符方式存在的，比如 \${jdbc.mysql.url}。当PropertyPlaceholderConfigurer作为BeanFactoryPostProcessor被应用时，它会使用properties配置文件中的值来替换相应的BeanDefinition中占位符所表示的属性值。当需要实例化bean时，bean定义中的属性值就已经被替换成我们配置的值。当然其实现比上面描述的要复杂一些，这里仅说明其大致工作原理，更详细的实现可以参考其源码。

与之相似的，还有 BeanPostProcessor，其存在于对象实例化阶段。跟BeanFactoryPostProcessor类似，它会处理容器内所有符合条件并且已经实例化后的对象。简单的对比，BeanFactoryPostProcessor处理bean的定义，而BeanPostProcessor则处理bean完成实例化后的对象。BeanPostProcessor定义了两个接口：

```
public interface BeanPostProcessor {
 // 前置处理
 Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;
 // 后置处理
 Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;
}
```

为了理解这两个方法执行的时机，简单的了解下bean的整个生命周期：



Bean的实例化过程（来自：Spring揭秘）

`postProcessBeforeInitialization()`方法与 `postProcessAfterInitialization()` 分别对应图中前置处理和后置处理两个步骤将执行的方法。这两个方法中都传入了bean对象实例的引用，为扩展容器的对象实例化过程提供了很大便利，在这儿几乎可以对传入的实例执行任何操作。注解、AOP等功能的实现均大量使用了 `BeanPostProcessor`，比如有一个自定义注解，你完全可以实现`BeanPostProcessor`的接口，在其中判断bean对象的脑袋上是否有该注解，如果有，你可以对这个bean实例执行任何操作，想想是不是非常的简单？

再来看一个更常见的例子，在Spring中经常能够看到各种各样的Aware接口，其作用就是在对象实例化完成以后将Aware接口定义中规定的依赖注入到当前实例中。比如最常见的 `ApplicationContextAware` 接口，实现了这个接口的类都可以获取到一个`ApplicationContext`对象。当容器中每个对象的实例化过程走到`BeanPostProcessor`前置处理这一步时，容器会检测到之前注册到容器的 `ApplicationContextAwareProcessor`，然后就会调用其`postProcessBeforeInitialization()`方法，检查并设置Aware相关依赖。看看代码吧，是不是很简单：

```

// 代码来自：org.springframework.context.support.ApplicationContextAwareProcessor
// 其postProcessBeforeInitialization方法调用了invokeAwareInterfaces方法
private void invokeAwareInterfaces(Object bean) {
 if(bean instanceof EnvironmentAware) {
 ((EnvironmentAware) bean).setEnvironment(this.applicationContext.getEnvironment());
 }
 if(bean instanceof ApplicationContextAware) {
 ((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);
 }
 //
}

```

最后总结一下，本小节内容和你一起回顾了Spring容器的部分核心内容，限于篇幅不能写更多，但理解这部分内容，足以让您轻松理解 Spring Boot的启动原理，如果在后续的学习过程中遇到一些晦涩难懂的知识，再回过头来看看Spring的核心知识，也许有意想不到的效果。也许Spring Boot的中文资料很少，但Spring的中文资料和书籍有太多太多，总有东西能给你启发。

## 二、夯实基础：JavaConfig与常见Annotation

### 2.1、JavaConfig

我们知道 `bean` 是Spring IOC中非常核心的概念，Spring容器负责bean的生命周期的管理。在最初，Spring使用XML配置文件的方式来描述bean的定义以及相互间的依赖关系，但随着Spring的发展，越来越多的人对这种方式表示不满，因为Spring项目的所有业务类均以bean的形式配置在XML文件中，造成了大量的XML文件，使项目变得复杂且难以管理。

后来，基于纯Java Annotation依赖注入框架 Guice 出世，其性能明显优于采用XML方式的Spring，甚至有部分人认为，Guice 可以完全取

代Spring（Guice仅是一个轻量级IOC框架，取代Spring还差的挺远）。正是这样的危机感，促使Spring及社区推出并持续完善了JavaConfig子项目，它基于Java代码和Annotation注解来描述bean之间的依赖绑定关系。比如，下面是使用XML配置方式来描述bean的定义：

```
<bean id="bookService" class="cn.moondev.service.BookServiceImpl"></bean>
```

而基于JavaConfig的配置形式是这样的：

```
@Configuration
public class MoonBookConfiguration {

 // 任何标注了@Bean的方法，其返回值将作为一个bean注册到Spring的IOC容器中
 // 方法名默认成为该bean定义的id

 @Bean
 public BookService bookService() {
 return new BookServiceImpl();
 }
}
```

如果两个bean之间有依赖关系的话，在XML配置中应该是这样：

```
<bean id="bookService" class="cn.moondev.service.BookServiceImpl">
 <property name="dependencyService" ref="dependencyService"/>
 </bean>

<bean id="otherService" class="cn.moondev.service.OtherServiceImpl">
 <property name="dependencyService" ref="dependencyService"/>
 </bean>

<bean id="dependencyService" class="DependencyServiceImpl"/>
```

而在JavaConfig中则是这样：

```
@Configuration
public class MoonBookConfiguration {

 // 如果一个bean依赖另一个bean，则直接调用对应JavaConfig类中依赖bean的创建方法即可
 // 这里直接调用dependencyService()

 @Bean
 public BookService bookService() {
 return new BookServiceImpl(dependencyService());
 }

 @Bean
 public OtherService otherService() {
 return new OtherServiceImpl(dependencyService());
 }

 @Bean
 public DependencyService dependencyService() {
 return new DependencyServiceImpl();
 }
}
```

你可能注意到这个示例中，有两个bean都依赖于dependencyService，也就是说当初始化bookService时会调用 dependencyService()，在初始化otherService时也会调用 dependencyService()，那么问题来了？这时候IOC容器中是有一个dependencyService实例还是两个？这个问题留着大家思考吧，这里不再赘述。

## 2.2、@ComponentScan

@ComponentScan注解对应XML配置形式中的<context:component-scan>元素，表示启用组件扫描，Spring会自动扫描所有通过注解配置的bean，然后将其注册到IOC容器中。我们可以通过basePackages等属性来指定@ComponentScan自动扫描的范围，如果不指

定，默认从声明 @ComponentScan 所在类的 package 进行扫描。正因为如此，SpringBoot 的启动类都默认在 src/main/java 下。

### 2.3、@Import

@Import 注解用于导入配置类，举个简单的例子：

```
@Configuration
public class MoonBookConfiguration{
 @Bean
 public BookService bookService() {
 return new BookServiceImpl();
 }
}
```

现在有另外一个配置类，比如： MoonUserConfiguration，这个配置类中有一个bean依赖于 MoonBookConfiguration 中的 bookService，如何将这两个bean组合在一起？借助 @Import 即可：

```
@Configuration
// 可以同时导入多个配置类，比如：@Import({A.class,B.class})
@Import(MoonBookConfiguration.class)
public class MoonUserConfiguration{
 @Bean
 public UserService userService(BookService bookService) {
 return new BookServiceImpl(bookService);
 }
}
```

需要注意的是，在4.2之前， @Import 注解只支持导入配置类，但是在4.2之后，它支持导入普通类，并将这个类作为一个bean的定义注册到IOC容器中。

### 2.4、@Conditional

@Conditional 注解表示在满足某种条件后才初始化一个bean或者启用某些配置。它一般用在由 @Component、 @Service、 @Configuration 等注解标识的类上面，或者由 @Bean 标记的方法上。如果一个 @Configuration 类标记了 @Conditional，则该类中所有标识了 @Bean 的方法和 @Import 注解导入的相关类将遵从这些条件。

在Spring里可以很方便的编写你自己的条件类，所要做的就是实现 Condition 接口，并覆盖它的 matches() 方法。举个例子，下面的简单条件类表示只有在 Classpath 里存在 JdbcTemplate 类时才生效：

```
public class JdbcTemplateCondition implements Condition{

 @Override
 public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata annotatedTypeMetadata) {
 try{
 conditionContext.getClassLoader().loadClass("org.springframework.jdbc.core.JdbcTemplate");
 return true;
 } catch(ClassNotFoundException e) {
 e.printStackTrace();
 }
 return false;
 }
}
```

当你用Java来声明bean的时候，可以使用这个自定义条件类：

```
@Conditional(JdbcTemplateCondition.class)
@Service
public MyService service() {

}
```

这个例子中只有当 `JdbcTemplateCondition` 类的条件成立时才会创建 `MyService` 这个 bean。也就是说 `MyService` 这个 bean 的创建条件是 `classpath` 里面包含 `JdbcTemplate`，否则这个 bean 的声明就会被忽略掉。

`SpringBoot` 定义了很多有趣的条件，并把他们运用到了配置类上，这些配置类构成了 `SpringBoot` 的自动配置的基础。`SpringBoot` 运用条件化配置的方法是：定义多个特殊的条件化注解，并将它们用到配置类上。下面列出了 `SpringBoot` 提供的部分条件化注解：

条件化注解	配置生效条件
<code>@ConditionalOnBean</code>	配置了某个特定 bean
<code>@ConditionalOnMissingBean</code>	没有配置特定的 bean
<code>@ConditionalOnClass</code>	Classpath 里有指定的类
<code>@ConditionalOnMissingClass</code>	Classpath 里没有指定的类
<code>@ConditionalOnExpression</code>	给定的 Spring Expression Language 表达式计算结果为 true
<code>@ConditionalOnJava</code>	Java 的版本匹配特定指或者一个范围值
<code>@ConditionalOnProperty</code>	指定的配置属性要有一个明确的值
<code>@ConditionalOnResource</code>	Classpath 里有指定的资源
<code>@ConditionalOnWebApplication</code>	这是一个 Web 应用程序
<code>@ConditionalOnNotWebApplication</code>	这不是一个 Web 应用程序

## 2.5、`@ConfigurationProperties` 与 `@EnableConfigurationProperties`

当某些属性的值需要配置的时候，我们一般会在 `application.properties` 文件中新建配置项，然后在 bean 中使用 `@Value` 注解来获取配置的值，比如下面配置数据源的代码。

```
// jdbc config
jdbc.mysql.url=jdbc:mysql://localhost:3306/sampledb
jdbc.mysql.username=root
jdbc.mysql.password=123456
.....
// 配置数据源
@Configuration
public class HikariDataSourceConfiguration{
 @Value("jdbc.mysql.url")
 public String url;
 @Value("jdbc.mysql.username")
 public String user;
 @Value("jdbc.mysql.password")
 public String password;

 @Bean
 public HikariDataSource dataSource() {
 HikariConfig hikariConfig = new HikariConfig();
 hikariConfig.setJdbcUrl(url);
 hikariConfig.setUsername(user);
 hikariConfig.setPassword(password);
 // 省略部分代码
 return new HikariDataSource(hikariConfig);
 }
}
```

使用 `@Value` 注解注入的属性通常都比较简单，如果同一个配置在多个地方使用，也存在不方便维护的问题（考虑下，如果有几十个地方在使用某个配置，而现在你想改下名字，你改怎么做？）。对于更为复杂的配置，`Spring Boot` 提供了更优雅的实现方式，那就是

@ConfigurationProperties注解。我们可以通过下面的方式来改写上面的代码：

```
@Component
// 还可以通过@PropertySource("classpath:jdbc.properties")来指定配置文件
@ConfigurationProperties("jdbc.mysql")
// 前缀=jdbc.mysql，会在配置文件中寻找jdbc.mysql.*的配置项
public class JdbcConfig{
 public String url;
 public String username;
 public String password;
}

@Configuration
public class HikariDataSourceConfiguration{

 @Autowired
 public JdbcConfig config;

 @Bean
 public HikariDataSource dataSource() {
 HikariConfig hikariConfig = new HikariConfig();
 hikariConfig.setJdbcUrl(config.url);
 hikariConfig.setUsername(config.username);
 hikariConfig.setPassword(config.password);
 // 省略部分代码
 return new HikariDataSource(hikariConfig);
 }
}
```

@ConfigurationProperties对于更为复杂的配置，处理起来也是得心应手，比如有如下配置文件：

```
#App
app.menus[0].title=Home
app.menus[0].name=Home
app.menus[0].path=/
app.menus[1].title=Login
app.menus[1].name=Login
app.menus[1].path=/login

app.compiler.timeout=5
app.compiler.output-folder=/temp/

app.error=/error/
```

可以定义如下配置类来接收这些属性

```
@Component
@ConfigurationProperties("app")
public class AppProperties{

 public String error;
 public List<Menu> menus = new ArrayList<>();
 public Compiler compiler = new Compiler();

 public static class Menu{
 public String name;
 public String path;
 public String title;
 }

 public static class Compiler{
 public String timeout;
 public String outputFolder;
 }
}
```

@EnableConfigurationProperties注解表示对@ConfigurationProperties的内嵌支持，默认会将对应Properties Class作为bean注入的IOC容器中，即在相应的Properties类上不用加@Component注解。

### 三、削铁如泥：SpringFactoriesLoader详解

JVM提供了3种类加载器：BootstrapClassLoader、ExtClassLoader、AppClassLoader分别加载Java核心类库、扩展类库以及应用的类路径(CLASSPATH)下的类库。JVM通过双亲委派模型进行类的加载，我们也可以通过继承java.lang.classloader实现自己的类加载器。

#### 何为双亲委派模型？

当一个类加载器收到类加载任务时，会先交给自己的父加载器去完成，因此最终加载任务都会传递到最顶层的BootstrapClassLoader，只有当父加载器无法完成加载任务时，才会尝试自己来加载。

采用双亲委派模型的一个好处是保证使用不同类加载器最终得到的都是同一个对象，这样就可以保证Java核心库的类型安全，比如，加载位于rt.jar包中的java.lang.Object类，不管是哪个加载器加载这个类，最终都是委托给顶层的BootstrapClassLoader来加载的，这样就可以保证任何的类加载器最终得到的都是同样一个Object对象。查看ClassLoader的源码，对双亲委派模型会有更直观的认识：

```
protectedClass<?> loadClass(String name, boolean resolve) {
 synchronized(getClassLoadingLock(name)) {
 // 首先，检查该类是否已经被加载，如果从JVM缓存中找到该类，则直接返回
 Class<?> c = findLoadedClass(name);
 if(c == null) {
 try{
 // 遵循双亲委派的模型，首先会通过递归从父加载器开始找，
 // 直到父类加载器是BootstrapClassLoader为止
 if(parent != null) {
 c = parent.loadClass(name, false);
 } else{
 c = findBootstrapClassOrNull(name);
 }
 } catch(ClassNotFoundException e) {}
 if(c == null) {
 // 如果还找不到，尝试通过findClass方法去寻找
 // findClass是留给开发者自己实现的，也就是说
 // 自定义类加载器时，重写此方法即可
 c = findClass(name);
 }
 }
 if(resolve) {
 resolveClass(c);
 }
 }
 return c;
}
```

但双亲委派模型并不能解决所有的类加载器问题，比如，Java提供了很多服务提供者接口(ServiceProviderInterface，SPI)，允许第三方为这些接口提供实现。常见的SPI有JDBC、JNDI、JAXP等，这些SPI的接口由核心类库提供，却由第三方实现，这样就存在一个问题：SPI的接口是Java核心库的一部分，是由BootstrapClassLoader加载的；SPI实现的Java类一般是由AppClassLoader来加载的。BootstrapClassLoader是无法找到SPI的实现类的，因为它只加载Java的核心库。它也不能代理给AppClassLoader，因为它是最顶层的类加载器。也就是说，双亲委派模型并不能解决这个问题。

线程上下文类加载器(ContextClassLoader)正好解决了这个问题。从名称上看，可能会误解为它是一种新的类加载器，实际上，它仅仅是Thread类的一个变量而已，可以通过setContextClassLoader(ClassLoadercl)和getContextClassLoader()来设置和获取该对象。如果不做任何的设置，Java应用的线程的上下文类加载器默认就是AppClassLoader。在核心类库使用SPI接口时，传递的类加载器使用线程上下文类加载器，就可以成功的加载到SPI实现的类。线程上下文类加载器在很多SPI的实现中都会用到。但在JDBC中，你可能会看到一种更直接的实现方式，比如，JDBC驱动管理java.sql.Driver中的loadInitialDrivers()方法中，你可以直接看到JDK是如何加载驱动的：

```
for(String aDriver : driversList) {
 try{
 // 直接使用AppClassLoader
 Class.forName(aDriver, true, ClassLoader.getSystemClassLoader());
 } catch(Exception ex) {
 println("DriverManager.Initialize: load failed: "+ ex);
 }
}
```

其实讲解线程上下文类加载器，最主要是让大家在看到 `Thread.currentThread().getClassLoader()` 和 `Thread.currentThread().getContextClassLoader()` 时不会一脸懵逼，这两者除了在许多底层框架中取得的 `ClassLoader` 可能会有所不同外，其他大多数业务场景下都是一样的，大家只要知道它是为了解决什么问题而存在的即可。

类加载器除了加载 `class` 外，还有一个非常重要功能，就是加载资源，它可以从 `JAR` 包中读取任何资源文件，比如，`ClassLoader.getResources(String name)` 方法就是用于读取 `JAR` 包中的资源文件，其代码如下：

```
public Enumeration<URL> getResources(String name) throws IOException{
 Enumeration<URL>[] tmp = (Enumeration<URL>[]) new Enumeration<?>[2];
 if(parent != null){
 tmp[0] = parent.getResources(name);
 } else{
 tmp[0] = getBootstrapResources(name);
 }
 tmp[1] = findResources(name);
 returnnewCompoundEnumeration<>(tmp);
}
```

是不是觉得有点眼熟，不错，它的逻辑其实跟类加载的逻辑是一样的，首先判断父类加载器是否为空，不为空则委托父类加载器执行资源查找任务，直到 `BootstrapClassLoader`，最后才轮到自己查找。而不同的类加载器负责扫描不同路径下的 `JAR` 包，就如同加载 `class` 一样，最后会扫描所有的 `JAR` 包，找到符合条件的资源文件。

类加载器的 `findResources(name)` 方法会遍历其负责加载的所有 `JAR` 包，找到 `JAR` 包中名称为 `name` 的资源文件，这里的资源可以是任何文件，甚至是 `.class` 文件，比如下面的示例，用于查找 `Array.class` 文件：

```
// 寻找Array.class文件
public static void main(String[] args) throws Exception{
 // Array.class的完整路径
 String name = "java/sql/Array.class";
 Enumeration<URL> urls = Thread.currentThread().getContextClassLoader().getResources(name);
 while(urls.hasMoreElements()) {
 URL url = urls.nextElement();
 System.out.println(url.toString());
 }
}
```

运行后可以得到如下结果：

```
$JAVA_HOME/jre/lib/rt.jar!/java/sql/Array.class
```

根据资源文件的 URL，可以构造相应的文件来读取资源内容。

看到这里，你可能会感到挺奇怪的，你不是要详解 `SpringFactoriesLoader` 吗？上来讲了一堆 `ClassLoader` 是几个意思？看下它的源码你就知道了：

```

public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";
// spring.factories文件的格式为：key=value1,value2,value3
// 从所有的jar包中找到META-INF/spring.factories文件
// 然后从文件中解析出key=factoryClass类名称的所有value值
public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {
 String factoryClassName = factoryClass.getName();
 // 取得资源文件的URL
 Enumeration<URL> urls = (classLoader != null ? classLoader.getResources(FACTORIES_RESOURCE_LOCATION) : ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
 List<String> result = newArrayList<String>();
 // 遍历所有的URL
 while(urls.hasMoreElements()) {
 URL url = urls.nextElement();
 // 根据资源文件URL解析properties文件
 Properties properties = PropertiesLoaderUtils.loadProperties(newUrlResource(url));
 String factoryClassNames = properties.getProperty(factoryClassName);
 // 组装数据，并返回
 result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(factoryClassNames)));
 }
 return result;
}

```

有了前面关于ClassLoader的知识，再来理解这段代码，是不是感觉豁然开朗：从 CLASSPATH 下的每个Jar包中搜寻所有 META-INF/spring.factories 配置文件，然后将解析properties文件，找到指定名称的配置后返回。需要注意的是，其实这里不仅仅是会去 ClassPath 路径下查找，会扫描所有路径下的Jar包，只不过这个文件只会在 Classpath 下的 jar 包中。来简单看下 spring.factories 文件的内容吧：

```

// 来自 org.springframework.boot.autoconfigure 下的 META-INF/spring.factories
// EnableAutoConfiguration 后文会讲到，它用于开启 Spring Boot 自动配置功能
org.springframework.boot.autoconfigure.EnableAutoConfiguration=
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration\

```

执行 loadFactoryNames(EnableAutoConfiguration.class, classLoader) 后，得到对应的一组 @Configuration 类，我们就可以通过反射实例化这些类然后注入到 IOC 容器中，最后容器里就有了一系列标注了 @Configuration 的 JavaConfig 形式的配置类。

这就是 SpringFactoriesLoader，它本质上属于 Spring 框架私有的一种扩展方案，类似于 SPI，Spring Boot 在 Spring 基础上的很多核心功能都是基于此，希望大家可以理解。

#### 四、另一件武器：Spring 容器的事件监听机制

过去，事件监听机制多用于图形界面编程，比如：点击按钮、在文本框输入内容等操作被称为事件，而当事件触发时，应用程序作出一定的响应则表示应用监听了这个事件，而在服务器端，事件的监听机制更多的用于异步通知以及监控和异常处理。Java 提供了实现事件监听机制的两个基础类：自定义事件类型扩展自 java.util.EventObject、事件的监听器扩展自 java.util.EventListener。来看一个简单的实例：简单的监控一个方法的耗时。

首先定义事件类型，通常的做法是扩展 EventObject，随着事件的发生，相应的状态通常都封装在此类中：

```

public class MethodMonitorEvent extends EventObject{
 // 时间戳，用于记录方法开始执行的时间
 public long timestamp;
 public MethodMonitorEvent(Object source) {
 super(source);
 }
}

```

事件发布之后，相应的监听器即可对该类型的事件进行处理，我们可以在方法开始执行之前发布一个 begin 事件，在方法执行结束之后发布一

个end事件，相应地，事件监听器需要提供方法对这两种情况下接收到的事件进行处理：

```
// 1、定义事件监听接口
public interface MethodMonitorEventListener extends EventListener{
 // 处理方法执行之前发布的事件
 public void onMethodBegin(MethodMonitorEvent event);
 // 处理方法结束时发布的事件
 public void onMethodEnd(MethodMonitorEvent event);
}

// 2、事件监听接口的实现：如何处理
public class AbstractMethodMonitorEventListener implements MethodMonitorEventListener{

 @Override
 public void onMethodBegin(MethodMonitorEvent event) {
 // 记录方法开始执行时的时间
 event.timestamp = System.currentTimeMillis();
 }

 @Override
 public void onMethodEnd(MethodMonitorEvent event) {
 // 计算方法耗时
 long duration = System.currentTimeMillis() - event.timestamp;
 System.out.println("耗时：" + duration);
 }
}
```

事件监听器接口针对不同的事件发布实际提供相应的处理方法定义，最重要的是，其方法只接收MethodMonitorEvent参数，说明这个监听器类只负责监听器对应的事件并进行处理。有了事件和监听器，剩下的就是发布事件，然后让相应的监听器监听并处理。通常情况，我们会有有一个事件发布者，它本身作为事件源，在合适的时机，将相应的事件发布给对应的事件监听器：

```
public class MethodMonitorEventPublisher {

 private List<MethodMonitorEventListener> listeners = new ArrayList<MethodMonitorEventListener>();

 public void methodMonitor() {
 MethodMonitorEvent eventObject = new MethodMonitorEvent(this);
 publishEvent("begin", eventObject);
 // 模拟方法执行：休眠5秒钟
 TimeUnit.SECONDS.sleep(5);
 publishEvent("end", eventObject);
 }

 private void publishEvent(String status, MethodMonitorEvent event) {
 // 避免在事件处理期间，监听器被移除，这里为了安全做一个复制操作
 List<MethodMonitorEventListener> copyListeners = new ArrayList<MethodMonitorEventListener>(listeners);
 for (MethodMonitorEventListener listener : copyListeners) {
 if ("begin".equals(status)) {
 listener.onMethodBegin(event);
 } else {
 listener.onMethodEnd(event);
 }
 }
 }

 public static void main(String[] args) {
 MethodMonitorEventPublisher publisher = new MethodMonitorEventPublisher();
 publisher.addEventListerner(new AbstractMethodMonitorEventListener());
 publisher.methodMonitor();
 }

 // 省略实现
 public void addEventListerner(MethodMonitorEventListener listener) {}
 public void removeEventListerner(MethodMonitorEventListener listener) {}
 public void removeAllListeners() {}
}
```

对于事件发布者（事件源）通常需要关注两点：

1、在合适的时机发布事件。此例中的methodMonitor()方法是事件发布的源头，其在方法执行之前和结束之后两个时间点发布MethodMonitorEvent事件，每个时间点发布的事件都会传给相应的监听器进行处理。在具体实现时需要注意的是，事件发布是顺序执行，为了不影响处理性能，事件监听器的处理逻辑应尽量简单。2、事件监听器的管理。publisher类中提供了事件监听器的注册与移除方法，这样客户端可以根据实际情况决定是否需要注册新的监听器或者移除某个监听器。如果这里没有提供remove方法，那么注册的监听器示例将一直被MethodMonitorEventPublisher引用，即使已经废弃不用了，也依然在发布者的监听器列表中，这会导致隐性的内存泄漏。

## Spring容器内的事件监听机制

Spring的ApplicationContext容器内部的所有事件类型均继承自 org.springframework.context.ApplicationEvent，容器中的所有监听器都实现 org.springframework.context.ApplicationListener 接口，并且以bean的形式注册在容器中。一旦在容器内发布ApplicationEvent及其子类型的事件，注册到容器的ApplicationListener就会对这些事件进行处理。

你应该已经猜到是怎么回事了。

ApplicationEvent继承自EventObject，Spring提供了一些默认的实现，比如： ContextClosedEvent 表示容器在即将关闭时发布的事件类型， ContextRefreshedEvent 表示容器在初始化或者刷新的时候发布的事件类型.....

容器内部使用ApplicationListener作为事件监听器接口定义，它继承自EventListener。ApplicationContext容器在启动时，会自动识别并加载EventListener类型的bean，一旦容器内有事件发布，将通知这些注册到容器的EventListener。

ApplicationContext接口继承了ApplicationEventPublisher接口，该接口提供了 voidpublishEvent(ApplicationEvent event) 方法定义，不难看出，ApplicationContext容器担当的就是事件发布者的角色。如果有兴趣可以查看 AbstractApplicationContext.publishEvent(ApplicationEvent event) 方法的源码： ApplicationContext 将事件的发布以及监听器的管理工作委托给 ApplicationEventMulticaster 接口的实现类。在容器启动时，会检查容器内是否存在名为 applicationEventMulticaster 的 ApplicationEventMulticaster 对象实例。如果有就使用其提供的实现，没有就默认初始化一个 SimpleApplicationEventMulticaster 作为实现。

最后，如果我们业务需要在容器内部发布事件，只需要为其注入ApplicationEventPublisher依赖即可：实现 ApplicationEventPublisherAware 接口或者 ApplicationContextAware 接口(Aware 接口相关内容请回顾上文)。

## 五、出神入化：揭秘自动配置原理

典型的Spring Boot应用的启动类一般均位于 src/main/java 根路径下，比如 MoonApplication 类：

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

其中 @SpringBootApplication 开启组件扫描和自动配置，而 SpringApplication.run 则负责启动引导应用程序。

@SpringBootApplication 是一个复合 Annotation，它将三个有用的注解组合在一起：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
 @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
 @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication{
 //
}
```

@SpringBootApplication 就是 @Configuration，它是Spring框架的注解，标明该类是一个 JavaConfig 配置类。而 @ComponentScan 启用组件扫描，前文已经详细讲解过，这里着重关注 @EnableAutoConfiguration。

@EnableAutoConfiguration 注解表示开启Spring Boot自动配置功能，Spring Boot会根据应用的依赖、自定义的bean、classpath下有没有某个类 等等因素来猜测你需要的bean，然后注册到IOC容器中。那 @EnableAutoConfiguration 是如何推算出你的需求？首先看下它的定义：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration{
 //
}
```

你的关注点应该在 @Import(EnableAutoConfigurationImportSelector.class) 上了，前文说过， @Import 注解用于导入类，并将这个类作为一个bean的定义注册到容器中，这里它将把 EnableAutoConfigurationImportSelector 作为bean注入到容器中，而这个类会将所有符合条件的@Configuration配置都加载到容器中，看看它的代码：

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
 // 省略了大部分代码，保留一句核心代码
 // 注意：SpringBoot最近版本中，这句代码被封装在一个单独的方法中
 // SpringFactoriesLoader相关知识请参考前文
 List<String> factories = new ArrayList<String>(new LinkedHashSet<String>(
 SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration.class, this.beanClassLoader)));
}
```

这个类会扫描所有的jar包，将所有符合条件的@Configuration配置类注入的容器中，何为符合条件，看看 META-INF/spring.factories 的文件内容：

```
// 来自 org.springframework.boot.autoconfigure下的META-INF/spring.factories
// 配置的key = EnableAutoConfiguration，与代码中一致
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration\
....
```

以 DataSourceAutoConfiguration 为例，看看Spring Boot是如何自动配置的：

```
@Configuration
 @ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
 @EnableConfigurationProperties(DataSourceProperties.class)
 @Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class })
 public class DataSourceAutoConfiguration {
 }
```

分别说一说：

`@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })`：当Classpath中存在DataSource或者EmbeddedDatabaseType类时才启用这个配置，否则这个配置将被忽略。

`@EnableConfigurationProperties(DataSourceProperties.class)`：将DataSource的默认配置类注入到IOC容器中，DataSourceproperties定义为：

```
// 提供对datasource配置信息的支持，所有的配置前缀为：spring.datasource
 @ConfigurationProperties(prefix = "spring.datasource")
 public class DataSourceProperties {
 private ClassLoader classLoader;
 private Environment environment;
 private String name = "testdb";

 }
```

`@Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class })`：导入其他额外的配置，就以 `DataSourcePoolMetadataProvidersConfiguration` 为例吧。

```
@Configuration
 public class DataSourcePoolMetadataProvidersConfiguration {
 @Configuration
 @ConditionalOnClass(org.apache.tomcat.jdbc.pool.DataSource.class)
 static class TomcatDataSourcePoolMetadataProviderConfiguration {
 @Bean
 public DataSourcePoolMetadataProvider tomcatPoolDataSourceMetadataProvider() {

 }
 }

 }
```

`DataSourcePoolMetadataProvidersConfiguration` 是数据库连接池提供者的一个配置类，即Classpath中存在 `org.apache.tomcat.jdbc.pool.DataSource.class`，则使用tomcat-jdbc连接池，如果Classpath中存在 `HikariDataSource.class` 则使用Hikari连接池。

这里仅描述了 `DataSourceAutoConfiguration` 的冰山一角，但足以说明 Spring Boot 如何利用条件话配置来实现自动配置的。回顾一下，`@EnableAutoConfiguration` 中导入了 `EnableAutoConfigurationImportSelector` 类，而这个类的 `selectImports()` 通过 `SpringFactoriesLoader` 得到了大量的配置类，而每一个配置类则根据条件化配置来做出决策，以实现自动配置。

整个流程很清晰，但漏了一个大问题：`EnableAutoConfigurationImportSelector.selectImports()` 是何时执行的？其实这个方法会在容器启动过程中执行：`AbstractApplicationContext.refresh()`，更多的细节在下一小节中说明。

## 六、启动引导：Spring Boot应用启动的秘密

### 6.1 SpringApplication初始化

SpringBoot整个启动流程分为两个步骤：初始化一个 `SpringApplication` 对象、执行该对象的 `run` 方法。看下 `SpringApplication` 的初始化流程，`SpringApplication` 的构造方法中调用 `initialize(Object[] sources)` 方法，其代码如下：

```
private void initialize(Object[] sources) {
 if(sources != null && sources.length > 0) {
 this.sources.addAll(Arrays.asList(sources));
 }
 // 判断是否是Web项目
 this.webEnvironment = deduceWebEnvironment();
 setInitializers((Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class));
 setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
 // 找到入口类
 this.mainApplicationClass = deduceMainApplicationClass();
}
```

初始化流程中最重要的就是通过SpringFactoriesLoader找到 spring.factories 文件中配置的 ApplicationContextInitializer 和 ApplicationListener 两个接口的实现类名称，以便后期构造相应的实例。 ApplicationContextInitializer 的主要目的是在 ConfigurableApplicationContext 做 refresh 之前，对 ConfigurableApplicationContext 实例做进一步的设置或处理。 ConfigurableApplicationContext 继承自 ApplicationContext，其主要提供了对 ApplicationContext 进行设置的能力。

实现一个 ApplicationContextInitializer 非常简单，因为它只有一个方法，但大多数情况下我们没有必要自定义一个 ApplicationContextInitializer，即便是 Spring Boot 框架，它默认也只是注册了两个实现，毕竟 Spring 的容器已经非常成熟和稳定，你没有必要来改变它。

而 ApplicationListener 的目的就没什么好说的了，它是 Spring 框架对 Java 事件监听机制的一种框架实现，具体内容在前文 Spring 事件监听机制这个小节有详细讲解。这里主要说说，如果你想为 Spring Boot 应用添加监听器，该如何实现？

## Spring Boot 提供两种方式来添加自定义监听器：

通过 SpringApplication.addListeners(ApplicationListener<?>... listeners) 或者 SpringApplication.setListeners(Collection<? extends ApplicationListener<?>> listeners)

两个方法来添加一个或者多个自定义监听器

既然 SpringApplication 的初始化流程中已经从 spring.factories 中获取到 ApplicationListener 的实现类，那么我们直接在自己的 jar 包的 META-INF/spring.factories 文件中新增配置即可：

```
org.springframework.context.ApplicationListener=\n cn.moondv.listeners.xxxxListener\
```

关于 SpringApplication 的初始化，我们就说这么多。

## 6.2 Spring Boot 启动流程

Spring Boot 应用的整个启动流程都封装在 SpringApplication.run 方法中，其整个流程真的是太长太长了，但本质上就是在 Spring 容器启动的基础上做了大量的扩展，按照这个思路来看看源码：

```

publicConfigurableApplicationContext run(String... args) {
 StopWatch stopWatch = newStopWatch();
 stopWatch.start();
 ConfigurableApplicationContext context = null;
 FailureAnalyzers analyzers = null;
 configureHeadlessProperty();
 //①
 SpringApplicationRunListeners listeners = getRunListeners(args);
 listeners.starting();
 try{
 //②
 ApplicationArguments applicationArguments = newDefaultApplicationArguments(args);
 ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);
 //③
 Banner printedBanner = printBanner(environment);
 //④
 context = createApplicationContext();
 //⑤
 analyzers = newFailureAnalyzers(context);
 //⑥
 prepareContext(context, environment, listeners, applicationArguments, printedBanner);
 //⑦
 refreshContext(context);
 //⑧
 afterRefresh(context, applicationArguments);
 //⑨
 listeners.finished(context, null);
 stopWatch.stop();
 return context;
 }
 catch(Throwable ex) {
 handleRunFailure(context, listeners, analyzers, ex);
 throw new IllegalStateException(ex);
 }
}

```

① 通过 SpringFactoriesLoader 查找并加载所有的 `SpringApplicationRunListeners`，通过调用 `starting()` 方法通知所有的 `SpringApplicationRunListeners`：应用开始启动了。`SpringApplicationRunListeners` 其本质上就是一个事件发布者，它在 SpringBoot 应用启动的不同时间点发布不同应用事件类型(`ApplicationEvent`)，如果有哪些事件监听者(`ApplicationListener`)对这些事件感兴趣，则可以接收并且处理。还记得初始化流程中，`SpringApplication` 加载了一系列 `ApplicationListener` 吗？这个启动流程中没有发现有发布事件的代码，其实都已经在 `SpringApplicationRunListeners` 这儿实现了。

简单的分析一下其实现流程，首先看下 `SpringApplicationRunListener` 的源码：

```

public interface SpringApplicationRunListener{
 // 运行run方法时立即调用此方法，可以用户非常早期的初始化工作
 void starting();

 // Environment准备好了后，并且ApplicationContext创建之前调用
 void environmentPrepared(ConfigurableEnvironment environment);

 // ApplicationContext创建好后立即调用
 void contextPrepared(ConfigurableApplicationContext context);

 // ApplicationContext加载完成，在refresh之前调用
 void contextLoaded(ConfigurableApplicationContext context);

 // 当run方法结束之前调用
 void finished(ConfigurableApplicationContext context, Throwable exception);
}

```

`SpringApplicationRunListener` 只有一个实现类：`EventPublishingRunListener`。①处的代码只会获取到一个 `EventPublishingRunListener` 的实例，我们来看看 `starting()` 方法的内容：

```
public void starting() {
 // 发布一个ApplicationStartedEvent
 this.initialMulticaster.multicastEvent(new ApplicationStartedEvent(this.application, this.args));
}
```

顺着这个逻辑，你可以在②处的 `prepareEnvironment()` 方法的源码中找到 `listeners.environmentPrepared(environment);` 即 `SpringApplicationRunListener` 接口的第二个方法，那不出你所料，`environmentPrepared()` 又发布了另外一个事件 `ApplicationEnvironmentPreparedEvent`。接下来会发生什么，就不用我多说了吧。

② 创建并配置当前应用将要使用的 `Environment`，`Environment` 用于描述应用程序当前的运行环境，其抽象了两个方面的内容：配置文件 (`profile`) 和属性 (`properties`)，开发经验丰富的同学对这两个东西一定不会陌生：不同的环境 (eg：生产环境、预发布环境) 可以使用不同的配置文件，而属性则可以从配置文件、环境变量、命令行参数等来源获取。因此，当 `Environment` 准备好后，在整个应用的任何时候，都可以从 `Environment` 中获取资源。

总结起来，②处的两句代码，主要完成以下几件事：

判断 `Environment` 是否存在，不存在就创建（如果是 web 项目就创建 `StandardServletEnvironment`，否则创建 `StandardEnvironment`）

配置 `Environment`：配置 `profile` 以及 `properties`

调用 `SpringApplicationRunListener` 的 `environmentPrepared()` 方法，通知事件监听者：应用的 `Environment` 已经准备好

③、`SpringBoot` 应用在启动时会输出这样的东西：

```
 .___.-' _--_
/\ /_ '-' _(_)_ __ _ _\ \ \
(()_ |'_|'_|'_V_`| \ \ \
\ \ _)|_| ||||| (|))
' |__| .__|_|_|_|_\ _| / / /
=====|_|=====|_|=/|_|/_/
:: Spring Boot :: (v1.5.6.RELEASE)
```

如果想把这个东西改成自己的涂鸦，你可以研究以下 `Banner` 的实现，这个任务就留给你们吧。

④、根据是否是 web 项目，来创建不同的 `ApplicationContext` 容器。

⑤、创建一系列 `FailureAnalyzer`，创建流程依然是通过 `SpringFactoriesLoader` 获取到所有实现 `FailureAnalyzer` 接口的 `class`，然后在创建对应的实例。`FailureAnalyzer` 用于分析故障并提供相关诊断信息。

⑥、初始化 `ApplicationContext`，主要完成以下工作：

将准备好的 `Environment` 设置给 `ApplicationContext`

遍历调用所有的 `ApplicationContextInitializer` 的 `initialize()` 方法来对已经创建好的 `ApplicationContext` 进行进一步的处理

调用 `SpringApplicationRunListener` 的 `contextPrepared()` 方法，通知所有的监听者：`ApplicationContext` 已经准备完毕

将所有的 bean 加载到容器中

调用 `SpringApplicationRunListener` 的 `contextLoaded()` 方法，通知所有的监听者：`ApplicationContext` 已经装载完毕

⑦、调用 `ApplicationContext` 的 `refresh()` 方法，完成 IoC 容器可用的最后一道工序。从名字上理解为刷新容器，那何为刷新？就是插手容器的启动，联系一下第一小节的内容。那如何刷新呢？且看下面代码：

```
// 摘自refresh()方法中一句代码
invokeBeanFactoryPostProcessors(beanFactory);
```

看看这个方法的实现：

```
protected void invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory) {
 PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(beanFactory, getBeanFactoryPostProcessors());

}
```

获取到所有的 BeanFactoryPostProcessor 来对容器做一些额外的操作。BeanFactoryPostProcessor 允许我们在容器实例化相应对象之前，对注册到容器的 BeanDefinition 所保存的信息做一些额外的操作。这里的 getBeanFactoryPostProcessors() 方法可以获取到 3 个 Processor：

```
ConfigurationWarningsApplicationContextInitializer$ConfigurationWarningsPostProcessor
SharedMetadataReaderFactoryContextInitializer$CachingMetadataReaderFactoryPostProcessor
ConfigFileApplicationListener$PropertySourceOrderingPostProcessor
```

不是有那么多 BeanFactoryPostProcessor 的实现类，为什么这儿只有这 3 个？因为在初始化流程获取到的各种 ApplicationContextInitializer 和 ApplicationListener 中，只有上文 3 个做了类似于如下操作：

```
public void initialize(ConfigurableApplicationContext context) {
 context.addBeanFactoryPostProcessor(new ConfigurationWarningsPostProcessor(getChecks()));
}
```

然后你就可以进入到 PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors() 方法了，这个方法除了会遍历上面的 3 个 BeanFactoryPostProcessor 处理外，还会获取类型为 BeanDefinitionRegistryPostProcessor 的 bean： org.springframework.context.annotation.internalConfigurationAnnotationProcessor，对应的 Class 为 ConfigurationClassPostProcessor。 ConfigurationClassPostProcessor 用于解析处理各种注解，包括： @Configuration、 @ComponentScan、 @Import、 @PropertySource、 @ImportResource、 @Bean。当处理 @import 注解的时候，就会调用<自动配置>这一小节中的 EnableAutoConfigurationImportSelector.selectImports() 来完成自动配置功能。其他的这里不再多讲，如果你有兴趣，可以查阅参考资料 6。

⑧、查找当前 context 中是否注册有 CommandLineRunner 和 ApplicationRunner，如果有则遍历执行它们。

⑨、执行所有 SpringApplicationRunListener 的 finished() 方法。

这就是 Spring Boot 的整个启动流程，其核心就是在 Spring 容器初始化并启动的基础上加入各种扩展点，这些扩展点包括： ApplicationContextInitializer、 ApplicationListener 以及各种 BeanFactoryPostProcessor 等等。你对整个流程的细节不必太过关注，甚至没弄明白也没有关系，你只要理解这些扩展点是在何时如何工作的，能让它们为你所用即可。

整个启动流程确实非常复杂，可以查询参考资料中的部分章节和内容，对照着源码，多看看，我想最终你都能弄清楚的。言而总之，Spring 才是核心，理解清楚 Spring 容器的启动流程，那 Spring Boot 启动流程就不在话下了。



Java后端

长按识别二维码，关注我的公众号

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot 的自动配置，到底是怎么做到？

Java后端 2019-12-16

点击上方 Java后端, 选择 **设为星标**

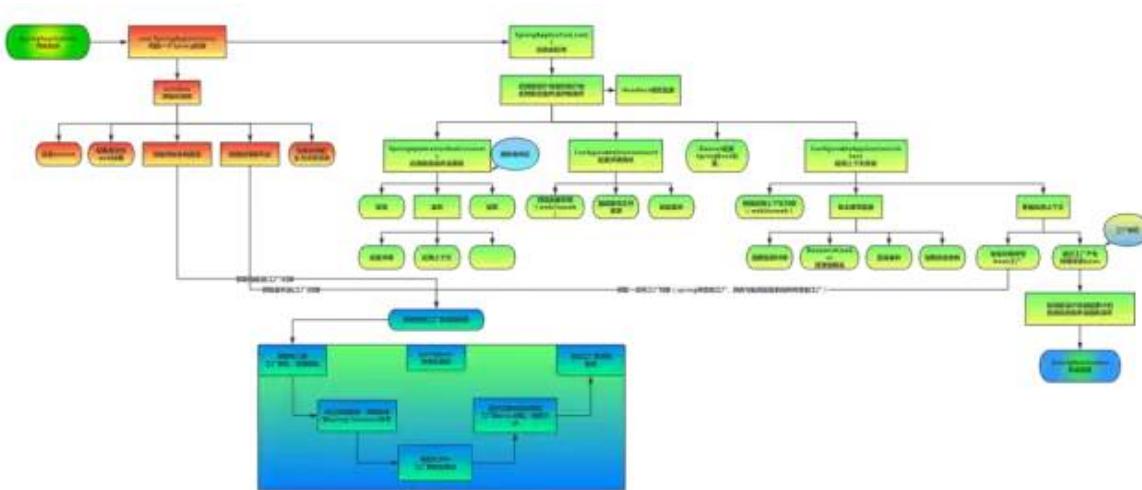
优质文章, 及时送达

作者 | 祖大帅

链接 | [juejin.im/post/5b679fbc5188251aad213110](https://juejin.im/post/5b679fbc5188251aad213110)

SpringBoot 的故事从一个面试题开始

## Spring Boot、Spring MVC 和 Spring 有什么区别？



先来个 SpringBoot 的启动结构图

## 分开描述各自的特征

Spring 框架就像一个家族，有众多衍生产品例如 boot、security、jpa 等等。但他们的基础都是 Spring 的 ioc 和 aop，ioc 提供了依赖注入的容器，aop 解决了面向横切面的编程，然后在此两者的基础上实现了其他延伸产品的高级功能。

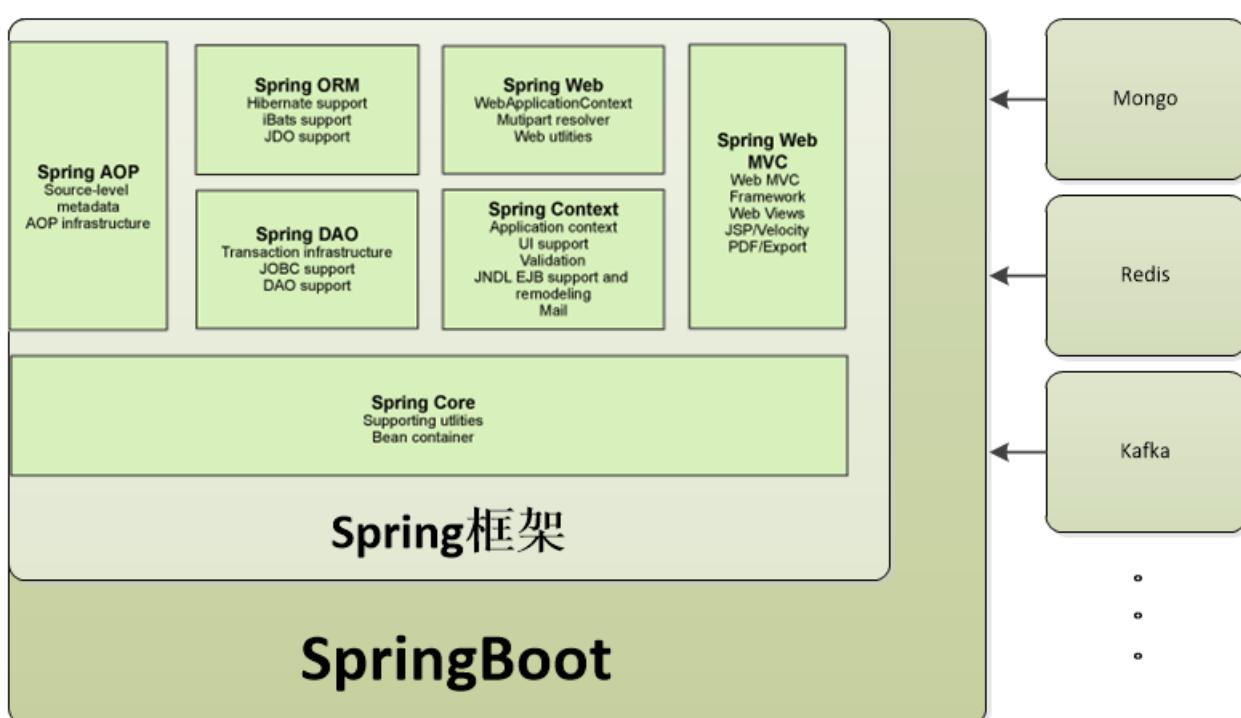
Spring MVC 提供了一种轻度耦合的方式来开发 web 应用。它是 Spring 的一个模块，是一个 web 框架。通过 DispatcherServlet, ModelAndView 和 View Resolver，开发 web 应用变得很容易。解决的问题领域是网站应用程序或者服务开发——URL 路由、Session、模板引擎、静态 Web 资源等等。

Spring Boot 实现了自动配置，降低了项目搭建的复杂度。它主要是为了解决使用 Spring 框架需要进行大量的配置太麻烦的问题，所以它并不是用来替代 Spring 的解决方案，而是和 Spring 框架紧密结合用于提升 Spring 开发者体验的工具。同时它集成了大量常用第三方库配置（例如 Jackson, JDBC, Mongo, Redis, Mail 等等），Spring Boot 应用中这些第三方库几乎可以零配置的开箱即用（out-of-the-box）。

Tips：关注微信公众号：Java后端，每日技术博文推送。

## 所以，用最简练的语言概括就是

- Spring 是一个“引擎”；
- Spring MVC 是基于 Spring 的一个 MVC 框架；
- Spring Boot 是基于 Spring 4 的条件注册的一套快速开发整合包。



## SpringBoot到底是怎么做到自动配置的？

从代码里看项目SpringBoot的项目启动类只有一个注解@SpringBootApplication和一个run方法。

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

直接看@SpringBootApplication的代码：

```

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
 excludeFilters = {@Filter(
 type = FilterType.CUSTOM,
 classes = {TypeExcludeFilter.class}
), @Filter(
 type = FilterType.CUSTOM,
 classes = {AutoConfigurationExcludeFilter.class}
)}
)
public @interface SpringBootApplication {
 @AliasFor(
 annotation = EnableAutoConfiguration.class
)
 Class<?>[] exclude() default {};
}

@AliasFor(
 annotation = EnableAutoConfiguration.class
)
String[] excludeName() default {};

@AliasFor(
 annotation = ComponentScan.class,
 attribute = "basePackages"
)
String[] scanBasePackages() default {};

@AliasFor(
 annotation = ComponentScan.class,
 attribute = "basePackageClasses"
)
Class<?>[] scanBasePackageClasses() default {};
}

```

@SpringBootApplication: 包含了@SpringBootConfiguration (打开是  
 @Configuration) , @EnableAutoConfiguration, @ComponentScan注解。

## [@Configuration](#)

JavaConfig形式的Spring IoC容器的配置类使用的那个@Configuration, SpringBoot社区推荐使用基于JavaConfig的配置形式, 所以, 这里的启动类标注了@Configuration之后, 本身其实也是一个IoC容器的配置类。

对比一下传统XML方式和config配置方式的区别:

## [XML声明和定义配置方式](#)

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"
 xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
 http://www.springframework.org/schema/aop
 http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context-3.0.xsd
 http://www.springframework.org/schema/tx
 http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
">
<bean id="app" class="com..." />
```

## 用一个过滤器举例，JavaConfig的配置方式是这样

```

@Configuration
public class DruidConfiguration {
 @Bean
 public FilterRegistrationBean statFilter(){
 //创建过滤器
 FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(new WebStatFilter());
 //设置过滤器过滤路径
 filterRegistrationBean.addUrlPatterns("/*");
 //忽略过滤的形式
 filterRegistrationBean.addInitParameter("exclusions","*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*");
 return filterRegistrationBean;
 }
}
```

任何一个标注了@Configuration的Java类定义都是一个JavaConfig配置类。

任何一个标注了@Bean的方法，其返回值将作为一个bean定义注册到Spring的IoC容器，方法名将默认成该bean定义的id。

## @ComponentScan

@ComponentScan对应XML配置中的元素，@ComponentScan的功能其实就是自动扫描并加载符合条件的组件（比如@Component和@Repository等）或者bean定义，最终将这些bean定义加载到IoC容器中。

我们可以通过basePackages等属性来细粒度的定制@ComponentScan自动扫描的范围，如果不指定，则默认Spring框架实现会从声明@ComponentScan所在类的package进行扫描。

注：所以SpringBoot的启动类最好是放在rootpackage下，因为默认不指定basePackages。

## @EnableAutoConfiguration

**(核心内容)** 看英文意思就是自动配置，概括一下就是，借助@Import的帮助，将所有符合自动配置条件的bean定义加载到IoC容器。

```

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
 String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

 Class<?>[] exclude() default {};
 String[] excludeName() default {};
}

```

里面最关键的是@Import(**EnableAutoConfigurationImportSelector.class**)，借助**EnableAutoConfigurationImportSelector**，**@EnableAutoConfiguration**可以帮助**SpringBoot**应用将所有符合条件的**@Configuration**配置都加载到当前**SpringBoot**创建并使用的**IoC**容器。该配置模块的主要使用到了**SpringFactoriesLoader**。

### SpringFactoriesLoader详解

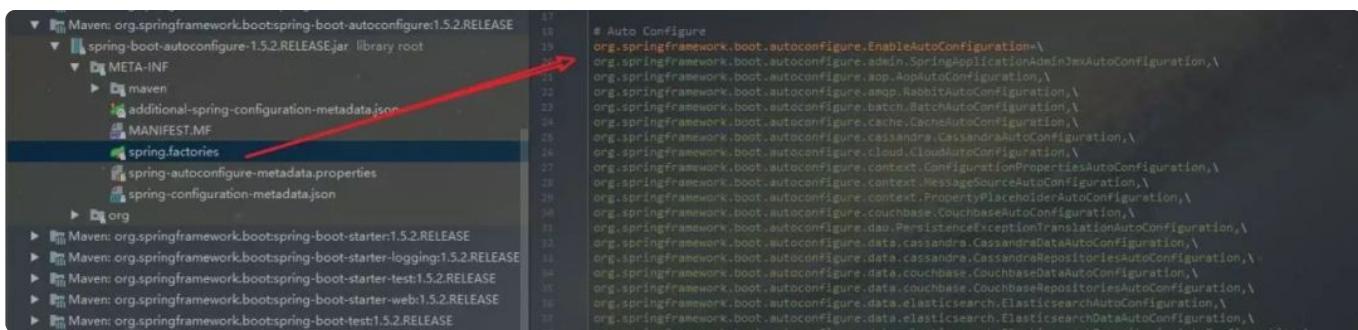
**SpringFactoriesLoader**为**Spring**工厂加载器，该对象提供了**loadFactoryNames**方法，入参为**factoryClass**和**classLoader**即需要传入工厂类名称和对应的类加载器，方法会根据指定的**classLoader**，加载该类加载器搜索路径下的指定文件，即**spring.factories**文件，传入的工厂类为接口，而文件中对应的类则是接口的实现类，或最终作为实现类。

```

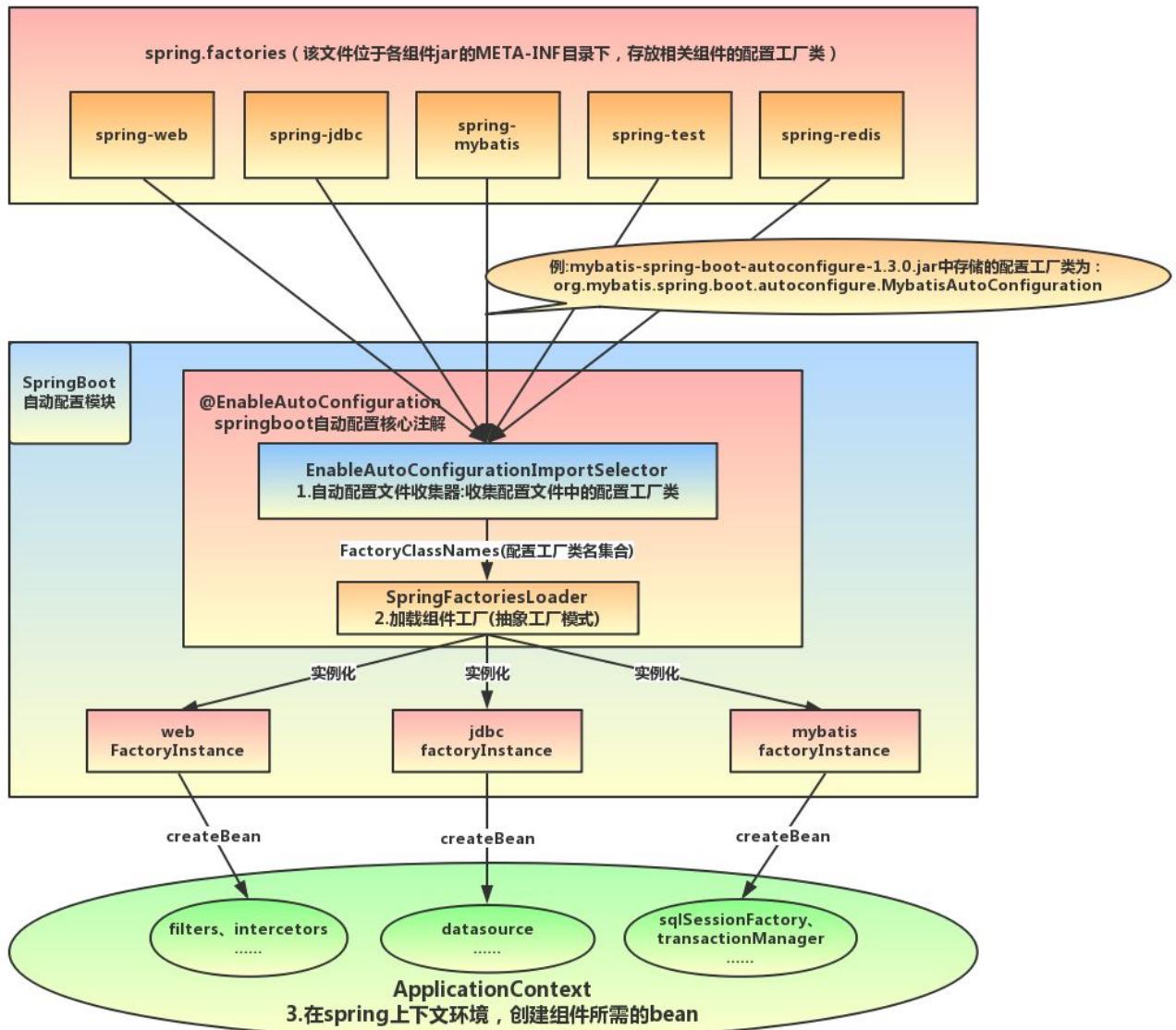
public abstract class SpringFactoriesLoader {
 ...
 public static <T> List<T> loadFactories(Class<T> factoryClass, ClassLoader classLoader) {
 ...
 }
 public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {
 ...
 }
}

```

所以文件中一般为如下图这种一对多的类名集合，获取到这些实现类的类名后，**loadFactoryNames**方法返回类名集合，方法调用方得到这些集合后，再通过**反射**获取这些类的类对象、构造方法，最终生成实例。



下图有助于我们形象理解自动配置流程（盗个图）



## AutoConfigurationImportSelector

继续上面讲的AutoConfigurationImportSelector.class。该类主要关注selectImports方法

```

public String[] selectImports(AnnotationMetadata annotationMetadata) {
 if (!this.isEnabled(annotationMetadata)) {
 return NO_IMPORTS;
 } else {
 AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
 AnnotationAttributes attributes = this.getAttributes(annotationMetadata);
 List<String> configurations = this.getCandidateConfigurations(annotationMetadata, attributes);
 configurations = this.removeDuplicates(configurations);
 Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);
 this.checkExcludedClasses(configurations, exclusions);
 configurations.removeAll(exclusions);
 configurations = this.filter(configurations, autoConfigurationMetadata);
 this.fireAutoConfigurationImportEvents(configurations, exclusions);
 return StringUtils.toStringArray(configurations);
 }
}

```

该方法在springboot启动流程——bean实例化前被执行，返回要实例化的类信息列表。如果获取到类信息，spring可以通过类加载器将类加载到jvm中，现在我们已经通过spring-boot的starter依赖方式依赖了我们需要的组件，那么这些组件的类信息在select方法中就可以被获取到。

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
 List<String> configurations = SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClass(), this.getBeanClas
 Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.factories. If you are using a custom packag
 return configurations;
}
```

方法中的getCandidateConfigurations方法，其返回一个自动配置类的类名列表，方法调用了loadFactoryNames方法，查看该方法

```
public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoader classLoader) {
 String factoryClassName = factoryClass.getName();
 return (List)loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collections.emptyList());
}
```

自动配置器会根据传入的factoryClass.getName()到项目系统路径下所有的spring.factories文件中找到相应的key，从而加载里面的类。我们就选取这个mybatis-spring-boot-autoconfigure下的spring.factories文件

```
Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.mybatis.spring.boot.autoconfigure.MybatisAutoConfiguration
```

进入org.mybatis.spring.boot.autoconfigure.MybatisAutoConfiguration中，又是一堆注解

```
@org.springframework.context.annotation.Configuration
@ConditionalOnClass({SqlSessionFactory.class, SqlSessionFactoryBean.class})
@ConditionalOnBean({DataSource.class})
@EnableConfigurationProperties({MybatisProperties.class})
@AutoConfigureAfter({DataSourceAutoConfiguration.class})
public class MybatisAutoConfiguration
{
 private static final Logger logger = LoggerFactory.getLogger(MybatisAutoConfiguration.class);
 private final MybatisProperties properties;
 private final Interceptor[] interceptors;
 private final ResourceLoader resourceLoader;
 private final DatabaseIdProvider databaseIdProvider;
 private final List<ConfigurationCustomizer> configurationCustomizers;
```

.@Spring的Configuration是一个通过注解标注的springBean，. @ConditionalOnClass({ SqlSessionFactory.class, SqlSessionFactoryBean.class})这个注解的意思是：当存在SqlSessionFactory.class, SqlSessionFactoryBean.class这两个类时才解析MybatisAutoConfiguration配置类,否则不解析这一个配置类。我们需要mybatis为我们返回会话对象，就必须有会话工厂相关类. @ConditionalOnBean(DataSource.class):只有处理已经被声明为bean的数据源。

@ConditionalOnMissingBean(MapperFactoryBean.class)这个注解的意思是如果容器中不存在name指定的bean则创建bean注入，否则不执行以上配置可以保证sqlSessionFactory、sqlSessionTemplate、dataSource等mybatis所需的组件均可被自动配置，@Configuration注解已经提供了Spring的上下文环境，所以以上组件的配置方式与Spring启动时通过mybatis.xml文件进行配置起到一个效果。

**只要一个基于SpringBoot项目的类路径下存在SqlSessionFactory.class, SqlSessionFactoryBean.class，并且容器中已经注册了dataSourceBean，就可以触发自动化配置**，意思说我们**只要在maven的项目中加入了mybatis所需要的若干依赖，就可以触发自动配置**，但引入mybatis原生依赖的话，每集成一个功能都要去修改其自动化配置类，那就得不到开箱即用的效果了。所以Spring-boot为我们提供了统一的starter可以直接配置好相关的类，触发自动配置所需的依赖(mybatis)如下：

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

因为maven依赖的传递性，我们只要依赖starter就可以依赖到所有需要自动配置的类，实现开箱即用的功能。也体现出Springboot简化了Spring框架带来的大量XML配置以及复杂的依赖管理，让开发人员可以更加关注业务逻辑的开发。

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



### 推荐阅读

1. [还搞不懂 Java NIO?快来读读这篇文章！](#)
2. [char 搞明白了吗？](#)
3. [使用 Redis 搭建电商秒杀系统](#)
4. [什么是一致性 Hash 算法？](#)
5. [团队开发中 Git 最佳实践](#)



喜欢文章, 点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot 线程池的创建、@Async 配置步骤及注意事项

Muscleheng Java后端 2月23日



微信搜一搜

Java后端

作者 | Muscleheng

[blog.csdn.net/Muscleheng/article/details/81409672](http://blog.csdn.net/Muscleheng/article/details/81409672)

## 前言

最近在做订单模块，用户购买服务类产品之后，需要进行预约，预约成功之后分别给商家和用户发送提醒短信。考虑发短信耗时的情况所以我想用异步的方法去执行，于是就在网上看见了Spring的@Async了。

但是遇到了许多问题，使得@Async无效，也一直没有找到很好的文章去详细的说明@Async的正确及错误的使用方法及需要注意的地方，这里简单整理了一下遇见的问题，Spring是以配置文件的形式来开启@Async，而SpringBoot则是以注解的方式开启。

我们可以使用springBoot默认的线程池，不过一般我们会自定义线程池（因为比较灵活），配置方式有：

1. 使用 xml 文件配置的方式
2. 使用Java代码结合@Configuration进行配置（推荐使用）

下面分别实现两种配置方式

## 第一步、配置@Async

### 一、Spring Boot启动类的配置：

在Spring Boot的主程序中配置@EnableAsync，如下所示：

```
@ServletComponentScan
@SpringBootApplication
@EnableAsync
public class ClubApiApplication {
 public static void main(String[] args) {
 SpringApplication.run(ClubApiApplication.class, args);
 }
}
```

### 二、Spring XML的配置方式：

1.applicationContext.xml同目录下创建文件threadPool.xml文件：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:task="http://www.springframework.org/schema/task"
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-task.xsd">

 <!-- 开启异步，并引入线程池 -->
 <task:annotation-driven executor="threadPool" />

 <!-- 定义线程池 -->
 <bean id="threadPool"
 class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
 <!-- 核心线程数， 默认为1 -->
 <property name="corePoolSize" value="10" />

 <!-- 最大线程数， 默认为Integer.MAX_VALUE -->
 <property name="maxPoolSize" value="50" />

 <!-- 队列最大长度， 一般需要设置值>=notifyScheduledMainExecutor.maxNum； 默认为Integer.MAX_VALUE -->
 <property name="queueCapacity" value="100" />

 <!-- 线程池维护线程所允许的空闲时间， 默认为60s -->
 <property name="keepAliveSeconds" value="30" />

 <!-- 完成任务自动关闭， 默认为false-->
 <property name="waitForTasksToCompleteOnShutdown" value="true" />

 <!-- 核心线程超时退出， 默认为false -->
 <property name="allowCoreThreadTimeOut" value="true" />

 <!-- 线程池对拒绝任务（无线程可用）的处理策略， 目前只支持AbortPolicy、CallerRunsPolicy； 默认为后者 -->
 <property name="rejectedExecutionHandler">
 <!-- AbortPolicy:直接抛出java.util.concurrent.RejectedExecutionException异常 -->
 <!-- CallerRunsPolicy:主线程直接执行该任务， 执行完之后尝试添加下一个任务到线程池中， 可以有效降低向线程池内添加任务的速度 -->
 <!-- DiscardOldestPolicy:抛弃旧的任务、暂不支持； 会导致被丢弃的任务无法再次被执行 -->
 <!-- DiscardPolicy:抛弃当前任务、暂不支持； 会导致被丢弃的任务无法再次被执行 -->
 <bean class="java.util.concurrent.ThreadPoolExecutor$CallerRunsPolicy" />
 </property>
 </bean>
</beans>

```

2.然后在applicationContext.xml中引入 `<import resource="threadPool.xml" />`

```

<!--如果不使用自定义线程池， 可以直接使用下面这段标签-->
<!--
<task:executor id="WhifExecutor" pool-size="10"/>
-->
<import resource="threadPool.xml" />
<task:annotation-driven executor="WhifExecutor" />

```

**第二步：创建两个异步方法的类，如下所示：**

第一个类（这里模拟取消订单后发短信，有两个发送短信的方法）：

```

@Service
public class TranTest2Service {

 // 发送提醒短信 1
 @Async
 public void sendMessage1() throws InterruptedException {

 System.out.println("发送短信方法---- 1 执行开始");
 Thread.sleep(5000); // 模拟耗时
 System.out.println("发送短信方法---- 1 执行结束");
 }

 // 发送提醒短信 2
 @Async
 public void sendMessage2() throws InterruptedException {

 System.out.println("发送短信方法---- 2 执行开始");
 Thread.sleep(2000); // 模拟耗时
 System.out.println("发送短信方法---- 2 执行结束");
 }
}

```

第二个类。调用发短信的方法（异步方法不能与被调用的异步方法在同一个类中，否则无效）：

```

@Service
public class OrderTaskService {
 @Autowired
 private TranTest2Service tranTest2Service;

 // 订单处理任务
 public void orderTask() throws InterruptedException {

 this.cancelOrder(); // 取消订单
 tranTest2Service.sendMessage1(); // 发短信的方法 1
 tranTest2Service.sendMessage2(); // 发短信的方法 2
 }

 // 取消订单
 public void cancelOrder() throws InterruptedException {
 System.out.println("取消订单的方法执行-----开始");
 System.out.println("取消订单的方法执行-----结束");
 }
}

```

经过测试得到如下结果：

1.没有使用@Async

```

取消订单的方法执行-----开始
取消订单的方法执行-----结束
发送短信方法---- 1 执行开始
发送短信方法---- 1 执行结束
发送短信方法---- 2 执行开始
发送短信方法---- 2 执行结束

```

<https://blog.csdn.net/Muscleheng>

2.使用了@Async

```
取消订单的方法执行-----开始
取消订单的方法执行-----结束
2018-08-03 23:27:46.165 INFO 10208 ---
发送短信方法---- 1 执行开始
发送短信方法---- 2 执行开始
发送短信方法---- 2 执行结束
发送短信方法---- 1 执行结束
https://blog.csdn.net/Muscleheng
```

可以看出，没有使用@Async方式实现的发送短信是同步执行的，意思就是说第一条发送之后再发送第二条，第二条发送成功之后再给用户提示，这样显然会影响用户体验，再看使用了@Async实现的，在执行第一个发送短信方法之后马上开启另一个线程执行第二个方法，显然这样我们的处理速度回快很多。

## 使用Java代码结合@Configuration注解的配置方式（推荐使用）

### 1. 新建一个配置类

```
/*
 * 线程池配置
 * @author zhh
 *
 */
@Configuration
@EnableAsync
public class ThreadPoolTaskConfig {

 /**
 * 默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，
 * 当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；
 * 当队列满了，就继续创建线程，当线程数量大于等于maxPoolSize后，开始使用拒绝策略拒绝
 */

 /** 核心线程数（默认线程数） */
 private static final int corePoolSize = 20;
 /** 最大线程数 */
 private static final int maxPoolSize = 100;
 /** 允许线程空闲时间（单位：默认为秒） */
 private static final int keepAliveTime = 10;
 /** 缓冲队列大小 */
 private static final int queueCapacity = 200;
 /** 线程池名前缀 */
 private static final String threadNamePrefix = "Async-Service-";

 @Bean("taskExecutor") //bean的名称，默认为首字母小写的方法名
 public ThreadPoolExecutor taskExecutor(){
 ThreadPoolExecutor executor = new ThreadPoolExecutor();
 executor.setCorePoolSize(corePoolSize);
 executor.setMaxPoolSize(maxPoolSize);
 executor.setQueueCapacity(queueCapacity);
 executor.setKeepAliveSeconds(keepAliveTime);
 executor.setThreadNamePrefix(threadNamePrefix);

 //线程池对拒绝任务的处理策略
 //CallerRunsPolicy：由调用线程（提交任务的线程）处理该任务
 executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
 //初始化
 executor.initialize();
 return executor;
 }
}
```

## 2. 创建两个异步方法的类（和之前的类类似仅仅是方法上注解不一样），如下所示：

第一个类（这里模拟取消订单后发短信，有两个发送短信的方法）：

```
@Service
public class TranTest2Service {
 Logger log = LoggerFactory.getLogger(TranTest2Service.class);

 // 发送提醒短信 1
 @PostConstruct // 加上该注解项目启动时就执行一次该方法
 @Async("taskExecutor")
 public void sendMessage1() throws InterruptedException {
 log.info("发送短信方法---- 1 执行开始");
 Thread.sleep(5000); // 模拟耗时
 log.info("发送短信方法---- 1 执行结束");
 }

 // 发送提醒短信 2
 @PostConstruct // 加上该注解项目启动时就执行一次该方法
 @Async("taskExecutor")
 public void sendMessage2() throws InterruptedException {

 log.info("发送短信方法---- 2 执行开始");
 Thread.sleep(2000); // 模拟耗时
 log.info("发送短信方法---- 2 执行结束");
 }
}
```

代码中的 `@Async("taskExecutor")` 对应我们自定义线程池中的 `@Bean("taskExecutor")`，表示使用我们自定义的线程池。

第二个类。调用发短信的方法（异步方法不能与被调用的异步方法在同一个类中，否则无效）：

```
@Service
public class OrderTaskService {
 @Autowired
 private TranTest2Service tranTest2Service;

 // 订单处理任务
 public void orderTask() throws InterruptedException {

 this.cancelOrder(); // 取消订单
 tranTest2Service.sendMessage1(); // 发短信的方法 1
 tranTest2Service.sendMessage2(); // 发短信的方法 2
 }

 // 取消订单
 public void cancelOrder() throws InterruptedException {
 System.out.println("取消订单的方法执行-----开始");
 System.out.println("取消订单的方法执行-----结束");
 }
}
```

运行截图：

```
2018-09-14 15:43:58.384 INFO 10256 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet'
2018-09-14 15:43:58.390 INFO 10256 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet'
2018-09-14 15:43:58.396 INFO 10256 --- [nio-8090-exec-1] c.b.test1.controller.BTest1Controller : id:10010, age:22
取消订单的方法执行-----开始
取消订单的方法执行-----结束
2018-09-14 15:43:58.433 INFO 10256 --- [Async-Service-2] com.boot.test1.service.TranTest2Service : 发送短信方法---- 2 执行开始
2018-09-14 15:43:58.435 INFO 10256 --- [Async-Service-1] com.boot.test1.service.TranTest2Service : 发送短信方法---- 1 执行开始
2018-09-14 15:44:00.434 INFO 10256 --- [Async-Service-2] com.boot.test1.service.TranTest2Service : 发送短信方法---- 2 执行结束
2018-09-14 15:44:03.437 INFO 10256 --- [Async-Service-1] com.boot.test1.service.TranTest2Service : 发送短信方法---- 1 执行结束
2018-09-14 15:51:43.326 INFO 10256 --- [nio-8090-exec-8] c.b.test1.controller.BTest1Controller : id:10010, age:22
```

注意看，截图中的 [nio-8090-exec-1] 是Tomcat的线程名称

[Async-Service-1]、[Async-Service-2]表示线程1和线程2，是我们自定义的线程池里面的线程名称，我们在配置类里面定义的线程池前缀：

```
private static final String threadNamePrefix = "Async-Service-"; // 线程池名前缀，说明我们自定义的线程池被使用了。
```

## 注意事项

如下方式会使@Async失效

- 异步方法使用static修饰
- 异步类没有使用@Component注解（或其他注解）导致spring无法扫描到异步类
- 异步方法不能与被调用的异步方法在同一个类中
- 类中需要使用@Autowired或@Resource等注解自动注入，不能自己手动new对象
- 如果使用SpringBoot框架必须在启动类中增加@EnableAsync注解

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓ 扫描二维码添加小编 ↓



## 推荐阅读

1. 6个接私活的网站，你有技术就有钱！
2. Spring Boot 整合 Redis
3. Java equals 和 hashCode 的这几个问题
4. 如何获取靠谱的新型冠状病毒疫情



微信搜一搜

Q Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot 集成 ElasticSearch

ZeroOne01 Java后端 2019-09-06

点击上方**蓝色字体**, 选择“标星公众号”

优质文章, 第一时间送达

作者 | ZeroOne01

链接 | [blog.51cto.com/zero01/2134718](http://blog.51cto.com/zero01/2134718)

## Spring Boot集成ElasticSearch

pom.xml文件中，依赖的各jar包版本如下：

```
1 <parent>
2 <groupId>org.springframework.boot</groupId>
3 >
4 <artifactId>spring-boot-starter-parent</artifactId>
5 >
6 <version>2.0.3.RELEASE</version>
7 >
8 <relativePath/> <!-- lookup parent from repository -->
9 </parent>
10
11 <properties>
12 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
13 >
14 <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
15 >
16 <java.version>1.8</java.version>
17 >
18 <elasticsearch.version>5.5.2</elasticsearch.version>
19 >
20 </properties>
21
22 <dependencies>
23 <dependency>
24 >
25 <groupId>org.springframework.boot</groupId>
26 >
27 <artifactId>spring-boot-starter-web</artifactId>
28 >
29 </dependency>
30 >
31 <dependency>
32 >
33 <groupId>org.projectlombok</groupId>
34 >
35 <artifactId>lombok</artifactId>
36 >
37 <optional>true</optional>
38 >
```

```
> </dependency>
>
> <dependency>
>
> <groupId>org.springframework.boot</groupId>
>
> <artifactId>spring-boot-starter-test</artifactId>
>
> <scope>test</scope>
>
> </dependency>
>
> <dependency>
>
> <groupId>org.elasticsearch.client</groupId>
>
> <artifactId>transport</artifactId>
>
> <version>${elasticsearch.version}</version>
>
> </dependency>
>
> </dependencies>
```

在工程中新建一个config包，在该包中创建一个ESConfig配置类，用于构造es的客户端实例对象。代码如下：

```
1 @Configuration
2 public class ESConfig {
3
4 @Bean
5 public TransportClient client() throws UnknownHostException
6 {
7 // 9300是es的tcp服务端口
8 InetSocketAddress node = new InetSocketAddress(
9 InetAddress.getByName("192.168.190.129")
10 ,
11 9300)
12 ;
13
14 // 设置es节点的配置信息
15 Settings settings = Settings.builder()
16 .put("cluster.name", "es"
17)
18 .build();
19
20
21 // 实例化es的客户端对象
22 TransportClient client = new PreBuiltTransportClient(settings);
23 client.addTransportAddress(node);
24
25 return client
```

```
 }
}
}
```

## 查询接口开发

我现在有一个结构化的索引如下：

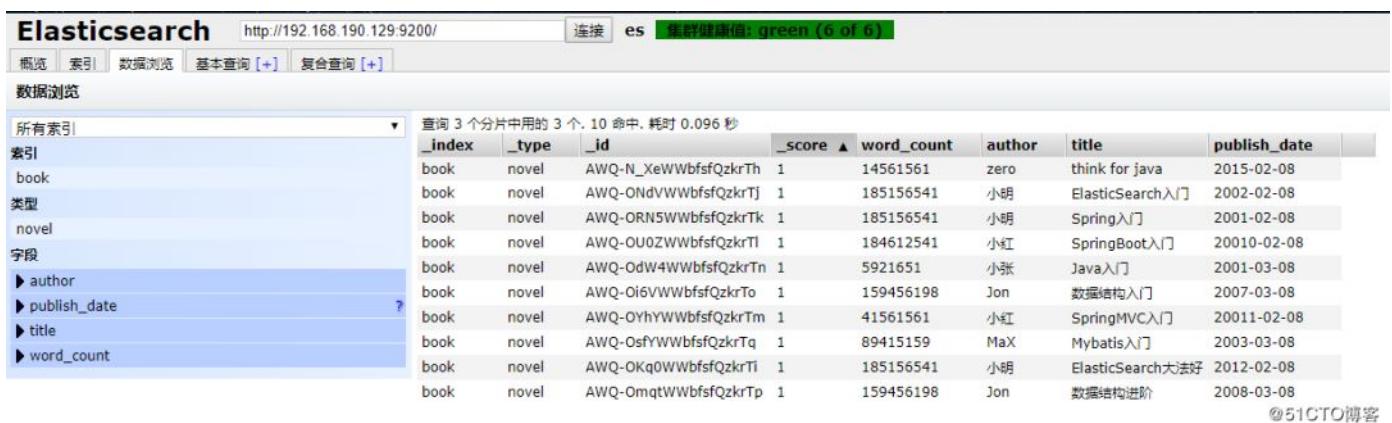


The screenshot shows the Elasticsearch Settings interface for the 'book' index. The configuration includes:

- state**: "open"
- settings**:
  - index**:
    - creation\_date**: "1530052030317"
    - number\_of\_shards**: "3"
    - number\_of\_replicas**: "1"
    - uuid**: "qg\_KG\_aNROCPEsPobaCDag"
    - version**:
      - created**: "5050299"
    - provided\_name**: "book"
  - mappings**:
    - novel**:
      - properties**:
        - word\_count**:
          - type**: "integer"
        - author**:
          - type**: "keyword"
        - title**:
          - type**: "text"
        - publish\_date**:
          - format**: "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd||epoch\_millis"
          - type**: "date"
  - aliases**: [ ]

@51CTO博客

该索引中有一些文档数据，如下：



The screenshot shows the Elasticsearch Data Browser displaying the 'book' index. The table lists 6 documents (hits) with the following details:

所有索引	_index	_type	_id	_score	word_count	author	title	publish_date
索引	book	novel	AWQ-N_XeWwbsfQzkrTh	1	14561561	zero	think for java	2015-02-08
类型	book	novel	AWQ-ONdVVWwbsfQzkrTj	1	185156541	小明	ElasticSearch入门	2002-02-08
novel	book	novel	AWQ-ORN5WWwbsfQzkrTk	1	185156541	小明	Spring入门	2001-02-08
字段	book	novel	AWQ-Ou0ZWWwbsfQzkrTl	1	184612541	小红	SpringBoot入门	20010-02-08
author	book	novel	AWQ-OdW4WWwbsfQzkrTn	1	5921651	小张	Java入门	2001-03-08
publish_date	book	novel	AWQ-Oi6VVWwbsfQzkrTo	1	159456198	Jon	数据结构入门	2007-03-08
title	book	novel	AWQ-OYhYWwbsfQzkrTm	1	41561561	小红	SpringMVC入门	20011-02-08
word_count	book	novel	AWQ-OsfYWwbsfQzkrTq	1	89415159	MaX	Mybatis入门	2003-03-08
	book	novel	AWQ-Okq0WWwbsfQzkrTt	1	185156541	小明	ElasticSearch大法好	2012-02-08
	book	novel	AWQ-QmqtWWwbsfQzkrTp	1	159456198	Jon	数据结构进阶	2008-03-08

@51CTO博客

在工程中新建一个controller包，在该包中新建一个 BookCrudController 类，用于演示es增删查改接口demo。我们首先来开发查询接口，代码如下：

```
1 @RestController
2 @RequestMapping("/es/demo")
3 public class BookCrudController {
4
 @Autowired
```

```

5 @Autowired
6
7 private TransportClient client;
8
9
10 /**
11 * 按id查询
12 * @param i
13 */
14 @GetMapping("/get/book/novel")
15
16 public ResponseEntity searchById(@RequestParam("id") String id) {
17 if (id.isEmpty()) {
18 return new ResponseEntity(HttpStatus.NOT_FOUND);
19 }
20
21 // 通过索引、类型、id向es进行查询数
22 GetResponse response = client.prepareGet("book", "novel", id).get()
23 ;
24
25 if (!response.exists()) {
26 return new ResponseEntity(HttpStatus.NOT_FOUND);
27 }
28
29 // 返回查询到的数据
30
31 return new ResponseEntity(response.getSource(), HttpStatus.OK);
32 }
33

```

启动SpringBoot工程，使用postman进行测试，查询结果如下：

The screenshot shows the Postman interface with the following details:

- URL:** localhost:8080/es/demo/get/book/novel?id=AWQ-N\_XeWWbfsfQzkrTh
- Method:** GET
- Response Body (Pretty JSON):**

```

1 [
2 "word_count": 14561561,
3 "author": "zero",
4 "title": "think for java",
5 "publish_date": "2015-02-08"
6]

```

## 新增接口开发

在 BookCrudController 类中开发新增接口，代码如下：

```

1 @PostMapping("/add/book/novel")
2 public ResponseEntity add(@RequestParam("title") String title,
3 @RequestParam("author") String author
4 ,
5 @RequestParam("word_count") int wordCount)
6

```

```
5
6 @RequestParam("publish_date"
7)
8 @DateTimeFormat(pattern = "yyy-MM-dd HH:mm:ss")
9)
10 Date publishDate) {
11
12 try {
13 // 将参数build成一个json对象
14 XContentBuilder content = XContentFactory.jsonBuilder()
15 .startObject()
16 .field("title", title
17)
18 .field("author", author
19)
20 .field("word_count", wordCount
21)
22 .field("publish_date", publishDate.getTime())
23 .endObject();
24
25
26 IndexResponse response = client.prepareIndex("book", "novel"
27)
28 .setSource(content)
29 .get()
30 ;
31
32
33 return new ResponseEntity(response.getId(), HttpStatus.OK);
34 } catch (IOException e) {
35 e.printStackTrace();
36 return new ResponseEntity(HttpStatus.INTERNAL_SERVER_ERROR);
37 }
38 }
```

重启SpringBoot工程，使用postman进行测试，测试结果如下：

The screenshot shows the Postman application interface. At the top, there are two tabs: 'localhost:8080/es/der' and 'localhost:8080/es/der'. To the right of these tabs are three buttons: a plus sign, three dots, and a gear icon. Further right is a dropdown menu labeled 'No Environment' with a downward arrow. On the far right are three icons: a magnifying glass, a user profile, and a gear.

In the main header area, the method is set to 'POST' with a dropdown arrow, the URL is 'localhost:8080/es/demo/add/book/novel', and there are buttons for 'Params', 'Send' (highlighted in blue), 'Save' (with a dropdown arrow), and another 'Save' button.

Below the header, there are tabs for 'Authorization', 'Headers', 'Body' (which is selected and highlighted in orange), 'Pre-request Script', and 'Tests'. To the right of these tabs are 'Cookies' and 'Code' buttons.

The 'Body' tab has four radio button options: 'form-data' (selected and highlighted in orange), 'x-www-form-urlencoded', 'raw', and 'binary'. The 'raw' section displays the following JSON payload:

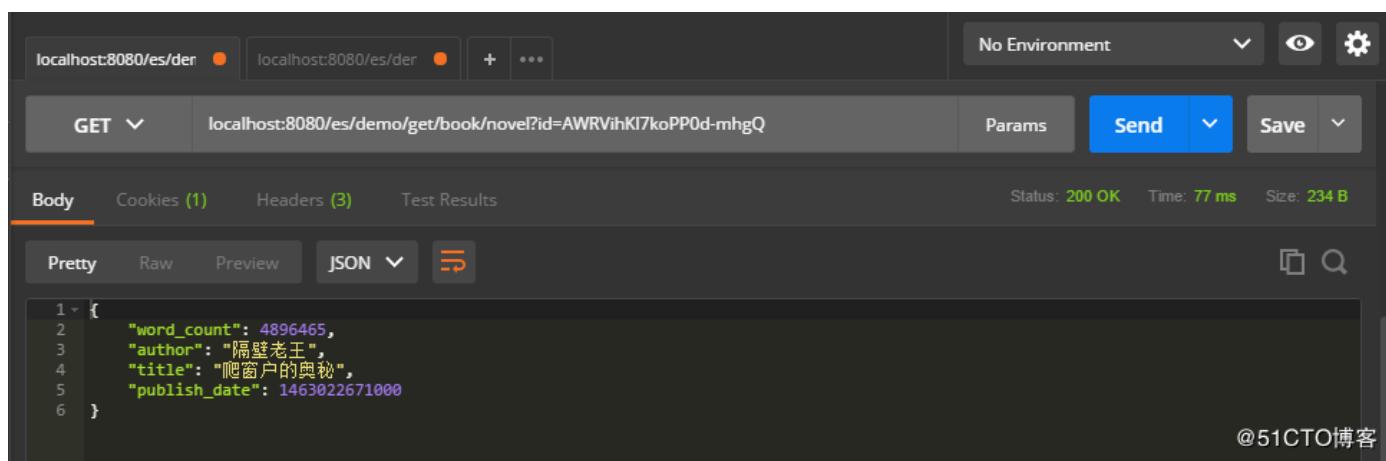
```
{
 "title": "爬窗户的奥秘",
 "author": "隔壁老王",
 "word_count": 4896465,
 "publish_date": "2016-05-12 11:11:11"
}
```

Below the raw payload, there is a row for adding new key-value pairs: 'New key' (Value) and 'Description'.

At the bottom, there are tabs for 'Body' (selected and highlighted in orange), 'Cookies (1)', 'Headers (3)', and 'Test Results'. To the right of these tabs are status metrics: 'Status: 200 OK', 'Time: 954 ms', and 'Size: 136 B'. Below these metrics are buttons for 'Pretty', 'Raw', 'Preview', and 'Text' (with a dropdown arrow). To the right of the text dropdown are two icons: a square with a double arrow and a magnifying glass.

The bottom-most part of the screenshot shows the response body, which contains a single line of text: '1 AWRVihK17koPP0d-mhgQ'.

使用返回的id去查询我们刚刚添加的数据，结果如下：



Postman screenshot showing a successful GET request to `localhost:8080/es/demo/get/book/novel?id=AWRVihKI7koPP0d-mhgQ`. The response status is 200 OK, time is 77 ms, and size is 234 B. The response body is a JSON object:

```
1 {
2 "word_count": 4896465,
3 "author": "隔壁老王",
4 "title": "爬窗户的奥秘",
5 "publish_date": 1463022671000
6 }
```

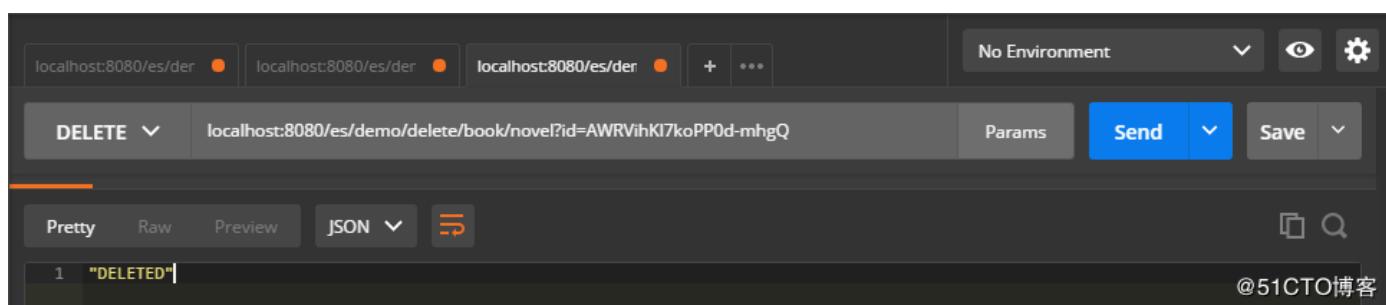
@51CTO博客

## 删除接口开发

代码如下：

```
1 @DeleteMapping("/delete/book/novel")
2 public ResponseEntity delete(@RequestParam("id") String id) {
3 DeleteResponse response = client.prepareDelete("book", "novel", id).get();
4
5 return new ResponseEntity(response.getResult(), HttpStatus.OK);
6 }
```

重启SpringBoot工程，使用postman进行测试，删除数据成功：



Postman screenshot showing a successful DELETE request to `localhost:8080/es/demo/delete/book/novel?id=AWRVihKI7koPP0d-mhgQ`. The response body is a JSON object containing the string "DELETED".

```
1 "DELETED"
```

@51CTO博客

## 更新接口开发

代码如下：

```
1 @PutMapping("/update/book/novel")
2 public ResponseEntity update(@RequestParam("id") String id,
3 @RequestParam(value = "title", required = false) String title
4 ,
5 @RequestParam(value = "author", required = false) String author
6 ,
7 @RequestParam(value = "word_count", required = false) Integer wordCount
8 ,
9 @RequestParam(value = "publish_date", required = false
10)
11 @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss"
12)
```

```

13 Date publishDate)
14 {
15 UpdateRequest update = new UpdateRequest("book", "novel", id)
16 ;
17 try
18 {
19 XContentBuilder builder = XContentFactory.jsonBuilder()
20 .startObject();
21
22 if (title != null)
23 {
24 builder.field("title", title)
25 ;
26 }
27 if (author != null)
28 {
29 builder.field("author", author);
30 }
31 if (wordCount != null)
32 {
33 builder.field("word_count", wordCount);
34 }
35 if (publishDate != null)
36 {
37 builder.field("publish_date", publishDate.getTime());
38 }
39
40 builder.endObject();
41 update.doc(builder);
42
43 UpdateResponse response = client.update(update).get();
44
45 return new ResponseEntity(response.getResult().toString(), HttpStatus.OK);
46 } catch (Exception e) {
47 e.printStackTrace();
48 return new ResponseEntity(HttpStatus.INTERNAL_SERVER_ERROR);
49 }
50 }

```

例如我们要修改文档id为AWQ-N\_XeWWbfsfQzkrTh的书籍数据：

The screenshot shows the Postman interface with the following details:

- URL:** `localhost:8080/es/demo/get/book/novel?id=AWQ-N_XeWWbfsfQzkrTh`
- Method:** GET
- Headers:** None
- Body:** None
- Params:** None
- Status:** 200 OK
- Time:** 44 ms
- Size:** 222 B
- JSON Response:**

```

1 { "word_count": 14561561,
2 "author": "zero",
3 "title": "think for java",
4 "publish_date": "2015-02-08"
5 }
```

修改书籍的标题和作者：

The screenshot shows a Postman interface with the following details:

- Method:** PUT
- URL:** localhost:8080/es/demo/update/book/novel
- Body (form-data):**
  - title: ElasticSearch从入门到放弃
  - author: zero菌
  - id: AWQ-N\_XeWWbfsfQzkrTh
- Status:** 200 OK
- Time:** 123 ms
- Size:** 122 B

修改成功：

The screenshot shows a Postman interface with the following details:

- Method:** GET
- URL:** localhost:8080/es/demo/get/book/novel?id=AWQ-N\_XeWWbfsfQzkrTh
- Body (JSON):**

```
1 {
2 "word_count": 14561561,
3 "author": "zero菌",
4 "title": "ElasticSearch从入门到放弃",
5 "publish_date": "2015-02-08"
6 }
```
- Status:** 200 OK
- Time:** 20 ms
- Size:** 242 B

## 复合查询接口开发

代码如下：

```
1 @PostMapping("/query/book/novel")
2 public ResponseEntity query(@RequestParam(value = "title", required = false) String title,
3 @RequestParam(value = "author", required = false) String author
4 ,
5 @RequestParam(value = "word_count", required = false) Integer wordCount
6 ,
7 @RequestParam(value = "publish_date", required = false
8)
9 @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss"
10)
11 Date publishDate
12 ,
13 @RequestParam(value = "gt_word_count", defaultValue = "0") Integer gtWordCoun
14 t,
15 @RequestParam(value = "lt_word_count", required = false) Integer ltWordCount)
```

```
16
17 // 组装查询条件
18
19 BoolQueryBuilder boolQuery = QueryBuilders.boolQuery();
20
21 if (title != null)
22 {
23 boolQuery.must(QueryBuilders.matchQuery("title", title));
24 }
25 if (author != null)
26 {
27 boolQuery.must(QueryBuilders.matchQuery("author", author));
28 }
29 if (wordCount != null)
30 {
31 boolQuery.must(QueryBuilders.matchQuery("word_count", wordCount));
32 }
33 if (publishDate != null)
34 {
35 boolQuery.must(QueryBuilders.matchQuery("publish_date", publishDate));
36 }
37 // 以word_count作为条件范围
38 RangeQueryBuilder rangeQuery = QueryBuilders.rangeQuery("word_count").from(gtWordCount);
39 if (ltWordCount != null && ltWordCount > 0)
40 {
41 rangeQuery.to(ltWordCount);
42 }
43 boolQuery.filter(rangeQuery);
44
45 // 组装查询请求
46
47 SearchRequestBuilder requestBuilder = client.prepareSearch("book");
48 .setTypes("novel")
49
50 .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
51 .setQuery(boolQuery)
52 .setFrom(0)
53 .setSize(10)
54;
55
56 // 发送查询请求
57
58 SearchResponse response = requestBuilder.get();
59
60 // 组装查询到的数据集
61
62 List<Map<String, Object>> result = new ArrayList<>();
63 for (SearchHit searchHitFields : response.getHits()) {
64 result.add(searchHitFields.getSource());
65 }
66
67 return new ResponseEntity(result, HttpStatus.OK);
68 }
```

重启SpringBoot工程，使用postman进行测试，测试结果如下：

POST localhost:8080/es/demo/query/book/novel

Body (1)

Key	Value	Description
author	小明	
title	ElasticSearch	
lt_word_count	185156541	

New key Value Description

Body Cookies (1) Headers (3) Test Results Status: 200 OK Time: 28 ms Size: 336 B

Pretty Raw Preview JSON

```
1 [[2 { 3 "word_count": 185156541, 4 "author": "小明", 5 "title": "ElasticSearch入门", 6 "publish_date": "2002-02-08" 7 }, 8 { 9 "word_count": 185156541, 10 "author": "小明", 11 "title": "ElasticSearch大法好", 12 "publish_date": "2012-02-08" 13 }]]
```

@51CTO博客

如果喜欢本篇文章，欢迎[转发](#)、[点赞](#)。关注订阅号「Web项目聚集地」，回复「进群」即可进入无广告技术交流。

## 推荐阅读

1. 基于 Spring Boot 的 Restful 风格实现增删改查
2. 如何使用牛逼的插件帮你规范代码
3. IntelliJ IDEA 构建maven多模块工程项目
4. 别在 Java 代码里乱打日志了，这才是正确姿势
5. 挑战 10 道超难 Java 面试题
6. 什么时候进行分库分表？



## Web项目聚集地

微信扫描二维码，关注我的公众号

喜欢文章，点个在看

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot：启动原理解析

平凡希 Java后端 2019-12-05

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 平凡希

来源 | [www.cnblogs.com/xiaoxi/p/7999885.html](http://www.cnblogs.com/xiaoxi/p/7999885.html)

我们开发任何一个Spring Boot项目，都会用到如下的启动类

```
1 @SpringBootApplication
2 public class Application {
3 public static void main(String[] args) {
4 SpringApplication.run(Application.class, args);
5 }
6 }
```

从上面代码可以看出，Annotation定义（@SpringBootApplication）和类定义（SpringApplication.run）最为耀眼，所以要揭开SpringBoot的神秘面纱，我们要从这两位开始就可以了。

## 一、**SpringBootApplication背后的秘密**

@SpringBootApplication注解是Spring Boot的核心注解，它其实是一个组合注解：

```
1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Documented
4 @Inherited
5 @SpringBootConfiguration
6 @EnableAutoConfiguration
7 @ComponentScan(excludeFilters = {
8 @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
9 @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
10 public @interface SpringApplication {
11 ...
12 }
```

虽然定义使用了多个Annotation进行了原信息标注，但实际上重要的只有三个Annotation：

- @Configuration（@SpringBootConfiguration点开查看发现里面还是应用了@Configuration）
- @EnableAutoConfiguration
- @ComponentScan

即 @SpringBootApplication = (默认属性)@Configuration + @EnableAutoConfiguration + @ComponentScan。

所以，如果我们使用如下的SpringBoot启动类，整个SpringBoot应用依然可以与之前的启动类功能对等：

```
1 @Configuration
2 @EnableAutoConfiguration
3 @ComponentScan
4 public class Application {
5 public static void main(String[] args) {
6 SpringApplication.run(Application.class, args);
7 }
8 }
```

每次写这3个比较累，所以写一个@SpringBootApplication方便点。接下来分别介绍这3个Annotation。

## 1、@Configuration

这里的@Configuration对我们来说不陌生，它就是JavaConfig形式的Spring  
loc容器的配置类使用的那个  
@Configuration，SpringBoot社区推荐使用基于JavaConfig的配置形式，所以，这里的启动类标注了@Configuration之后，  
本身其实也是一个IoC容器的配置类。

举几个简单例子回顾下，XML跟config配置方式的区别：

### (1) 表达形式层面

基于XML配置的方式是这样：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-bean.xsd">
5 default-lazy-init="true">
6 <!--bean定义-->
7 </beans>
```

而基于JavaConfig的配置方式是这样：

```
1 @Configuration
2 public class MockConfiguration{
3 //bean定义
4 }
```

任何一个标注了@Configuration的Java类定义都是一个JavaConfig配置类。

### (2) 注册bean定义层面

基于XML的配置形式是这样：

```
1 <bean id="mockService" class=".MockServiceImpl">
2 ...
3 </bean>
```

而基于JavaConfig的配置形式是这样的：

```
1 @Configuration
2 public class MockConfiguration{
3 @Bean
4 public MockService mockService(){
5 return new MockServiceImpl();
6 }
7 }
```

任何一个标注了@Bean的方法，其返回值将作为一个bean定义注册到Spring的IoC容器，方法名将默认成该bean定义的id。

### (3) 表达依赖注入关系层面

为了表达bean与bean之间的依赖关系，在XML形式中一般是这样：

```
1 <bean id="mockService" class=".MockServiceImpl">
2 <property name="dependencyService" ref="dependencyService" />
3 </bean>
4
5 <bean id="dependencyService" class="DependencyServiceImpl"></bean>
```

而基于JavaConfig的配置形式是这样的：

```
1 @Configuration
2 public class MockConfiguration{
3 @Bean
4 public MockService mockService(){
5 return new MockServiceImpl(dependencyService());
6 }
7
8 @Bean
9 public DependencyService dependencyService(){
10 return new DependencyServiceImpl();
11 }
12 }
```

如果一个bean的定义依赖其他bean，则直接调用对应的JavaConfig类中依赖bean的创建方法就可以了。

@Configuration：提到@Configuration就要提到他的搭档@Bean。使用这两个注解就可以创建一个简单的spring配置类，可以用来替代相应的xml配置文件。

```
1 <beans>
2 <bean id = "car" class="com.test.Car">
3 <property name="wheel" ref = "wheel"></property>
4 </bean>
5 <bean id = "wheel" class="com.test.Wheel"></bean>
6 </beans>
```

相当于：

```
1 @Configuration
2 public class Conf {
3 @Bean
4 public Car car() {
5 Car car = new Car();
6 car.setWheel(wheel());
7 return car;
8 }
9
10 @Bean
11 public Wheel wheel() {
12 return new Wheel();
13 }
14 }
```

@Configuration的注解类标识这个类可以使用Spring IoC容器作为bean定义的来源。

@Bean注解告诉Spring，一个带有@Bean的注解方法将返回一个对象，该对象应该被注册为在Spring应用程序上下文中的bean。

## 2、@ComponentScan

@ComponentScan这个注解在Spring中很重要，它对应XML配置中的元素，@ComponentScan的功能其实就是自动扫描并加载符合条件的组件（比如@Component和@Repository等）或者bean定义，最终将这些bean定义加载到IoC容器中。

我们可以通过basePackages等属性来细粒度的定制@ComponentScan自动扫描的范围，如果不指定，则默认Spring框架实现会从声明@ComponentScan所在类的package进行扫描。

注：所以SpringBoot的启动类最好是放在root package下，因为默认不指定basePackages。

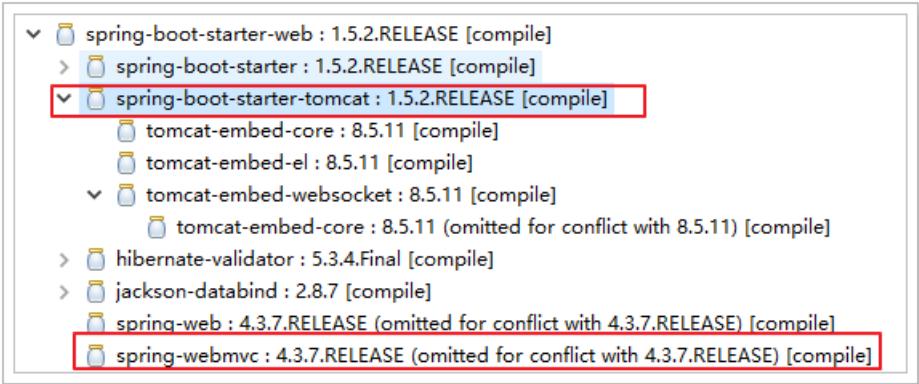
## 3、@EnableAutoConfiguration

个人感觉@EnableAutoConfiguration这个Annotation最为重要，所以放在最后来解读，大家是否还记得Spring框架提供的各种名字为@Enable开头的Annotation定义？比如@EnableScheduling、@EnableCaching、@EnableMBeanExport等，@EnableAutoConfiguration的理念和做事方式其实一脉相承，简单概括一下就是，**借助@Import的支持，收集和注册特定场景相关的bean定义。**

- @EnableScheduling是通过@Import将Spring调度框架相关的bean定义都加载到IoC容器。
- @EnableMBeanExport是通过@Import将JMX相关的bean定义加载到IoC容器。

而@EnableAutoConfiguration也是借助@Import的帮助，将所有符合自动配置条件的bean定义加载到IoC容器，仅此而已！

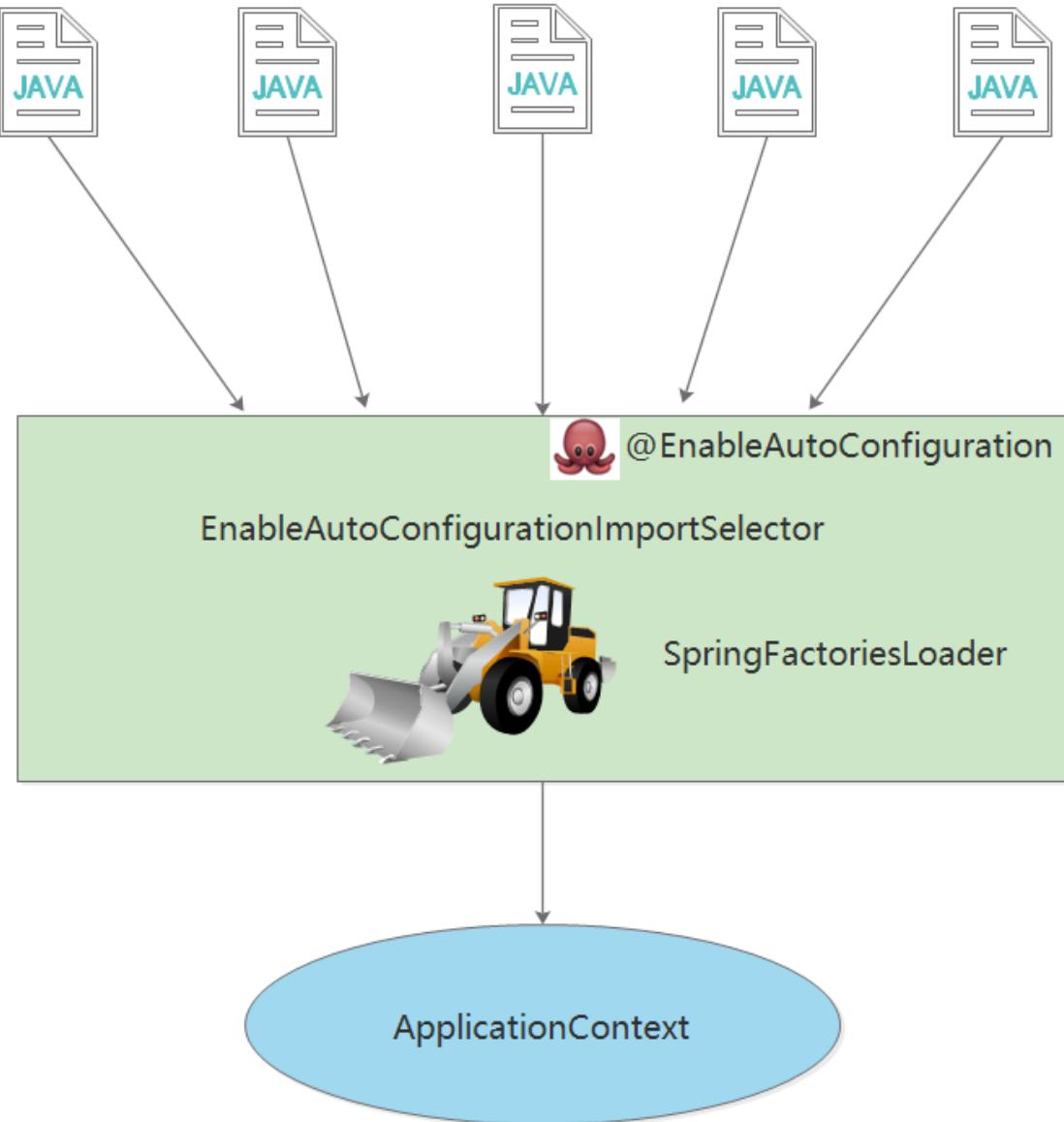
@EnableAutoConfiguration会根据类路径中的jar依赖为项目进行自动配置，如：添加了spring-boot-starter-web依赖，会自动添加Tomcat和Spring MVC的依赖，Spring Boot会对Tomcat和Spring MVC进行自动配置。



@EnableAutoConfiguration作为一个复合Annotation，其自身定义关键信息如下：

```
1 @SuppressWarnings("deprecation")
2 @Target(ElementType.TYPE)
3 @Retention(RetentionPolicy.RUNTIME)
4 @Documented
5 @Inherited
6 @AutoConfigurationPackage
7 @Import(EnableAutoConfigurationImportSelector.class)
8 public @interface EnableAutoConfiguration {
9 ...
10 }
```

其中，最关键的要属@Import(EnableAutoConfigurationImportSelector.class)，借助  
EnableAutoConfigurationImportSelector，@EnableAutoConfiguration可以帮助SpringBoot应用将所有符合条件的  
@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器。就像一只“八爪鱼”一样，借助于Spring框架原有的一个工具类：SpringFactoriesLoader的支持，@EnableAutoConfiguration可以智能的自动配置功效才得以大功告成！



EnableAutoConfiguration得以生效的关键组件关系图

### 自动配置幕后英雄：SpringFactoriesLoader详解

SpringFactoriesLoader属于Spring框架私有的一种扩展方案，其主要功能就是从指定的配置文件META-INF/spring.factories加载配置。

```

1 public abstract class SpringFactoriesLoader {
2 //...
3 public static <T> List<T> loadFactories(Class<T> factoryClass, ClassLoader classLoader) {
4 ...
5 }
6
7
8 public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {
9
10 }
11 }
```

配合`@EnableAutoConfiguration`使用的话，它更多是提供一种配置查找的功能支持，即根据`@EnableAutoConfiguration`的完整类名`org.springframework.boot.autoconfigure.EnableAutoConfiguration`作为查找的Key，获取对应的一组`@Configuration`类。

```

17 # Auto Configure
18 org.springframework.boot.autoconfigure.EnableAutoConfiguration=
19 org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,
20 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,
21 org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,
22 org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,
23 org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,
24 org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,
25 org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,
26 org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,
27 org.springframework.boot.autoconfigure.data.MessageSourceAutoConfiguration,
28 org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,
29 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseAutoConfiguration,
30 org.springframework.boot.autoconfigure.data.PersistenceExceptionTranslationAutoConfiguration,
31 org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,
32 org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,
33 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,
34 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration,
35 org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,
36 org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration,
37

```

上图就是从SpringBoot的autoconfigure依赖包中的META-INF/spring.factories配置文件中摘录的一段内容，可以很好地说明问题。

所以，`@EnableAutoConfiguration`自动配置的魔法骑士就变成了：**从classpath中搜寻所有的META-INF/spring.factories配置文件，并将其中org.springframework.boot.autoconfigure.EnableAutoConfiguration对应的配置项通过反射（Java Refletion）实例化为对应的标注了@Configuration的JavaConfig形式的IoC容器配置类，然后汇总为一个并加载到IoC容器。**

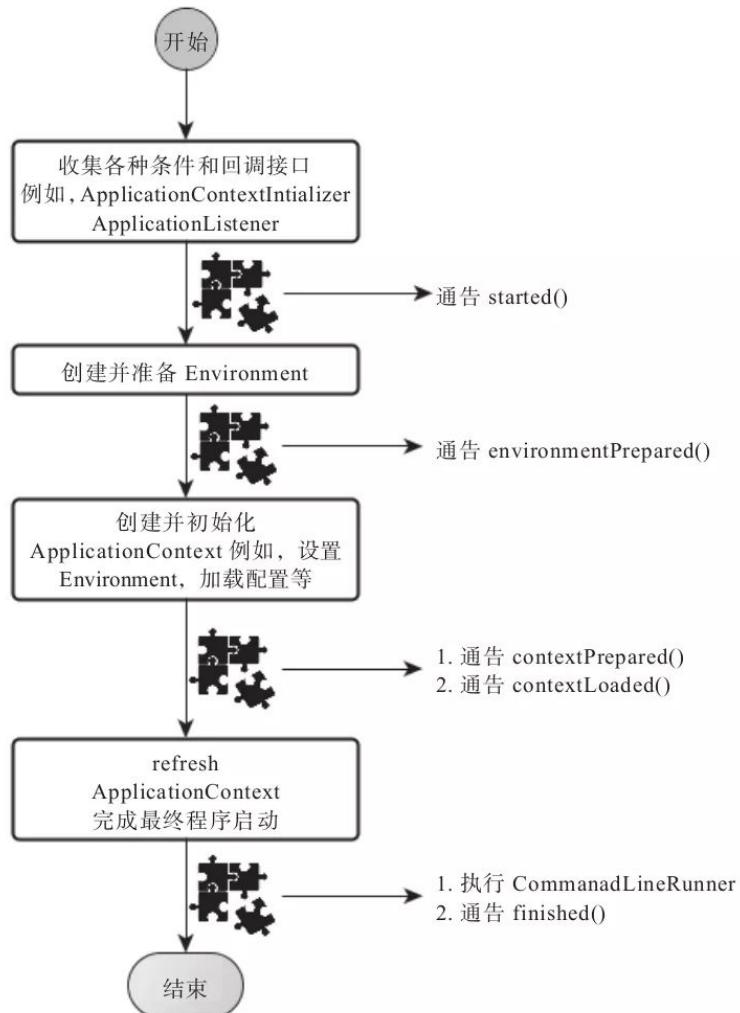
## 二、深入探索SpringApplication执行流程

SpringApplication的run方法的实现是我们本次旅程的主要线路，该方法的主要流程大体可以归纳如下：

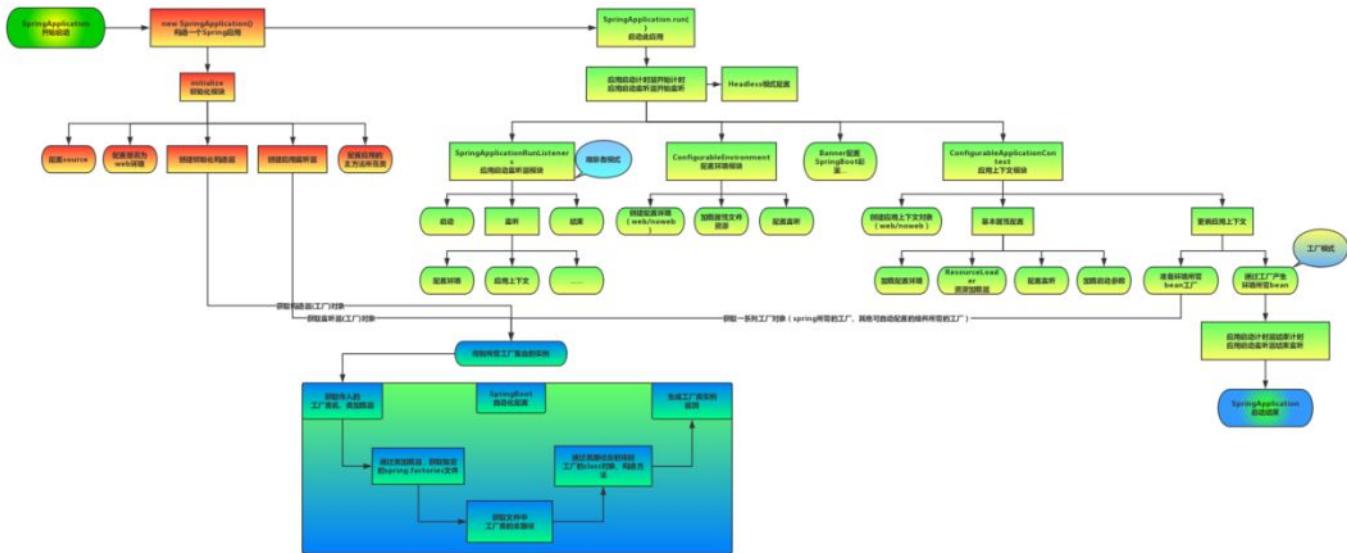
- 1) 如果我们使用的是SpringApplication的静态run方法，那么，这个方法里面首先要创建一个SpringApplication对象实例，然后调用这个创建好的SpringApplication的实例方法。在SpringApplication实例初始化的时候，它会提前做几件事情：
  - 根据classpath里面是否存在某个特征类（org.springframework.web.context.ConfigurableWebApplicationContext）来决定是否应该创建一个为Web应用使用的ApplicationContext类型。
  - 使用SpringFactoriesLoader在应用的classpath中查找并加载所有可用的ApplicationContextInitializer。
  - 使用SpringFactoriesLoader在应用的classpath中查找并加载所有可用的ApplicationListener。
  - 推断并设置main方法的定义类。
- 2) SpringApplication实例初始化完成并且完成设置后，就开始执行run方法的逻辑了，方法执行伊始，首先遍历执行所有通过SpringFactoriesLoader可以查找到并加载的SpringApplicationRunListener。调用它们的started()方法，告诉这些SpringApplicationRunListener，“嘿，SpringBoot应用要开始执行咯！”。
- 3) 创建并配置当前Spring Boot应用将要使用的Environment（包括配置要使用的PropertySource以及Profile）。
- 4) 遍历调用所有SpringApplicationRunListener的environmentPrepared()方法，告诉他们：“当前SpringBoot应用使用的Environment准备好了咯！”。
- 5) 如果SpringApplication的showBanner属性被设置为true，则打印banner。
- 6) 根据用户是否明确设置了applicationContextClass类型以及初始化阶段的推断结果，决定该为当前SpringBoot应用创建什么类型的ApplicationContext并创建完成，然后根据条件决定是否添加ShutdownHook，决定是否使用自定义的BeanNameGenerator，决定是否使用自定义的ResourceLoader，当然，最重要的，将之前准备好的Environment设置给创建好的ApplicationContext使用。
- 7) ApplicationContext创建好之后，SpringApplication会再次借助Spring-FactoriesLoader，查找并加载classpath中所有可用的ApplicationContext-Initializer，然后遍历调用这些ApplicationContextInitializer的initialize(applicationContext)方法来对已经创建好的ApplicationContext进行进一步的处理。

- 8) 遍历调用所有SpringApplicationRunListener的contextPrepared()方法。
- 9) 最核心的一步，将之前通过@EnableAutoConfiguration获取的所有配置以及其他形式的IoC容器配置加载到已经准备完毕的ApplicationContext。
- 10) 遍历调用所有SpringApplicationRunListener的contextLoaded()方法。
- 11) 调用ApplicationContext的refresh()方法，完成IoC容器可用的最后一道工序。
- 12) 查找当前ApplicationContext中是否注册有CommandLineRunner，如果有，则遍历执行它们。
- 13) 正常情况下，遍历执行SpringApplicationRunListener的finished()方法、(如果整个过程出现异常，则依然调用所有SpringApplicationRunListener的finished()方法，只不过这种情况下会将异常信息一并传入处理)

去除事件通知点后，整个流程如下：



本文以调试一个实际的SpringBoot启动程序为例，参考流程中主要类类图，来分析其启动逻辑和自动化配置原理。



## 总览：

上图为SpringBoot启动结构图，我们发现启动流程主要分为三个部分，第一部分进行`SpringApplication`的初始化模块，配置一些基本的环境变量、资源、构造器、监听器，第二部分实现了应用具体的启动方案，包括启动流程的监听模块、加载配置环境模块、及核心的创建上下文环境模块，第三部分是自动化配置模块，该模块作为springboot自动配置核心，在后面的分析中会详细讨论。在下面的启动程序中我们会串联起结构中的主要功能。

## 启动：

每个SpringBoot程序都有一个主入口，也就是`main`方法，`main`里面调用`SpringApplication.run()`启动整个`spring-boot`程序，该方法所在类需要使用`@SpringBootApplication`注解，以及`@ImportResource(if need)`，`@SpringBootApplication`包括三个注解，功能如下：

`@EnableAutoConfiguration`: SpringBoot根据应用所声明的依赖来对Spring框架进行自动配置。

`@SpringBootConfiguration(内部为@Configuration)`: 被标注的类等于在spring的XML配置文件中(`applicationContext.xml`)，装配所有bean事务，提供了一个spring的上下文环境。

`@ComponentScan`: 组件扫描，可自动发现和装配Bean，默认扫描`SpringApplication`的`run`方法里的`Booster.class`所在的包路径下文件，所以最好将该启动类放到根包路径下。

```

19 @SpringBootApplication
20 @ImportResource("classpath*:spring-context.xml")
21 public class Booter {
22
23 public static void main(String[] args) throws InterruptedException {
24 SpringApplication.run(Booster.class, args);
25 keepRunning();
26 }
27
28 /**
29 * 阻塞main方法
30 * @throws InterruptedException
31 */
32 private static void keepRunning() throws InterruptedException {
33 Thread t = Thread.currentThread();
34 synchronized (t) {
35 t.wait();
36 }
37 }
38 }
39

```

首先进入run方法

```
/**
 * Static helper that can be used to run a {@link SpringApplication} from the
 * specified sources using default settings and user supplied arguments.
 * @param sources the sources to load
 * @param args the application arguments (usually passed from a Java main method)
 * @return the running {@link ApplicationContext}
 */
public static ConfigurableApplicationContext |run|(Object[] sources, String[] args) {
 return new SpringApplication(sources).run(args);
}
```

run方法中去创建了一个SpringApplication实例，在该构造方法内，我们可以发现其调用了一个初始化的initialize方法

```
/**
 * Create a new {@link SpringApplication} instance. The application context will load
 * beans from the specified sources (see {@link SpringApplication class-level}
 * documentation for details. The instance can be customized before calling
 * {@link #run(String...)}.
 * @param sources the bean sources
 * @see #run(Object, String[])
 * @see #SpringApplication(ResourceLoader, Object...)
 */
@ public SpringApplication(Object... sources) {
 initialize(sources);
}
```

```
@ /unchecked, rawtypes/
private void initialize(Object[] sources) { sources: Object[1]@1138
 if (sources != null && sources.length > 0) {
 this.sources.addAll(Arrays.asList(sources)); sources: Object[1]@1138
 }
 this.webEnvironment = deduceWebEnvironment();
 setInitializers((Collection) getSpringFactoriesInstances(
 ApplicationContextInitializer.class));
 setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
 this.mainApplicationClass = deduceMainApplicationClass();
}
```

这里主要是为SpringApplication对象赋一些初值。构造函数执行完毕后，我们回到run方法

```

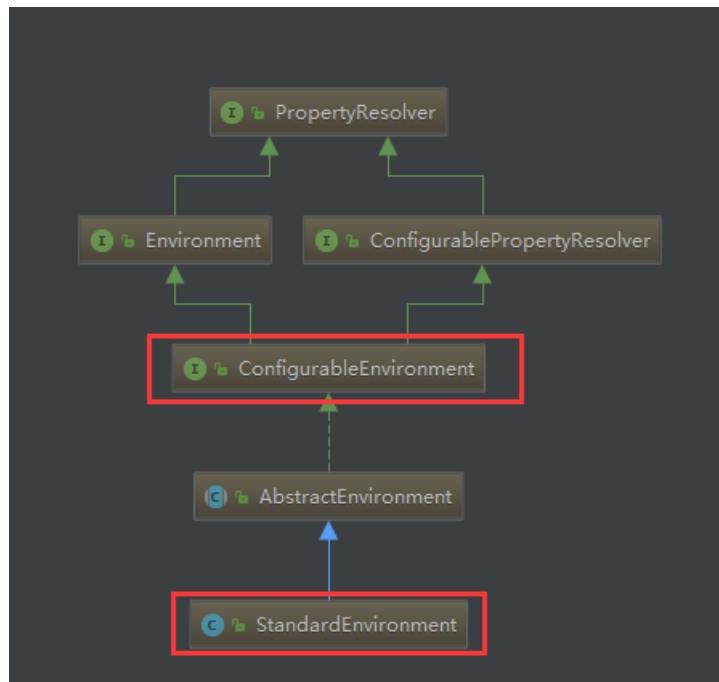
* Run the Spring application, creating and refreshing a new
* {@Link ApplicationContext}.
* @param args the application arguments (usually passed from a Java main method) args: {}
* @return a running {@Link ApplicationContext}
*/
public ConfigurableApplicationContext run(String... args) { args: {}
 StopWatch stopWatch = new StopWatch();
 stopWatch.start();
 ConfigurableApplicationContext context = null;
 FailureAnalyzers analyzers = null;
 configureHeadlessProperty();
 SpringApplicationRunListeners listeners = getRunListeners(args);
 listeners.starting();
 try {
 ApplicationArguments applicationArguments = new DefaultApplicationArguments(
 args);
 ConfigurableEnvironment environment = prepareEnvironment(listeners,
 applicationArguments);
 Banner printedBanner = printBanner(environment);
 context = createApplicationContext();
 analyzers = new FailureAnalyzers(context);
 prepareContext(context, environment, listeners, applicationArguments,
 printedBanner);
 refreshContext(context);
 afterRefresh(context, applicationArguments);
 listeners.finished(context, exception: null);
 stopWatch.stop();
 if (this.logStartupInfo) {
 new StartupInfoLogger(this.mainApplicationClass)
 .logStarted(getApplicationLog(), stopWatch);
 }
 }
 return context;
}

```

该方法中实现了如下几个关键步骤：

1. 创建了应用的监听器SpringApplicationRunListeners并开始监听

2. 加载SpringBoot配置环境(ConfigurableEnvironment)，如果是通过web容器发布，会加载StandardEnvironment，其最终也是继承了ConfigurableEnvironment，类图如下



可以看出，\*Environment最终都实现了PropertyResolver接口，我们平时通过environment对象获取配置文件中指定Key对应的value方法时，就是调用了propertyResolver接口的getProperty方法

3. 配置环境(Environment)加入到监听器对象中(SpringApplicationRunListeners)

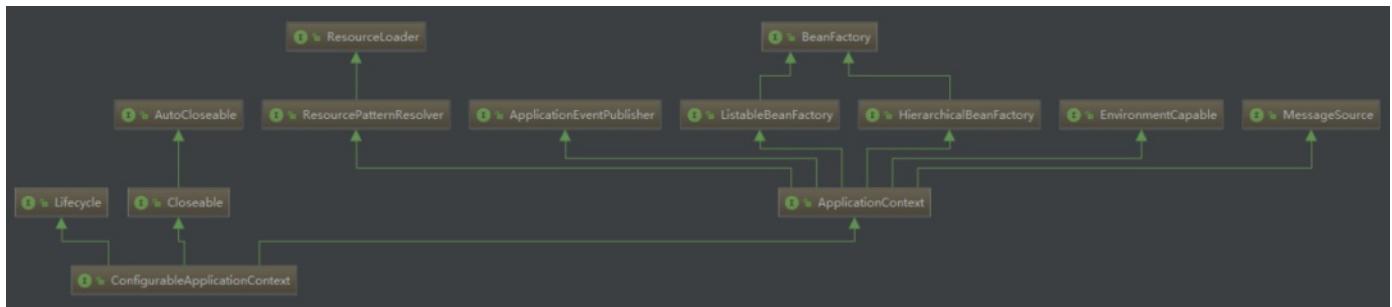
4. 创建run方法的返回对象：ConfigurableApplicationContext(应用配置上下文)，我们可以看一下创建方法：

```

551
552 /**
553 * Strategy method used to create the {@link ApplicationContext}. By default this
554 * method will respect any explicitly set application context or application context
555 * class before falling back to a suitable default.
556 * @return the application context (not yet refreshed)
557 * @see #setApplicationContextClass(Class)
558 */
559 protected ConfigurableApplicationContext createApplicationContext() {
560 Class<?> contextClass = this.applicationContextClass;
561 if (contextClass == null) {
562 try {
563 contextClass = Class.forName(this.webEnvironment
564 ? DEFAULT_WEB_CONTEXT_CLASS : DEFAULT_CONTEXT_CLASS);
565 }
566 catch (ClassNotFoundException ex) {
567 throw new IllegalStateException(
568 "Unable to create a default ApplicationContext, "
569 + "please specify an ApplicationContextClass",
570 ex);
571 }
572 }
573 return (ConfigurableApplicationContext) BeanUtils.instantiate(contextClass);
574 }
575
```

方法会先获取显式设置的应用上下文(applicationContextClass)，如果不存在，再加载默认的环境配置（通过是否是web environment判断），默认选择AnnotationConfigApplicationContext注解上下文（通过扫描所有注解类来加载bean），最后通过BeanUtils实例化上下文对象，并返回。

ConfigurableApplicationContext类图如下：



主要看其继承的两个方向：

LifeCycle：生命周期类，定义了start启动、stop结束、isRunning是否运行中等生命周期空值方法

ApplicationContext：应用上下文类，其主要继承了beanFactory(bean的工厂类)

5.回到run方法内，prepareContext方法将listeners、environment、applicationArguments、banner等重要组件与上下文对象关联

6.接下来的refreshContext(context)方法(初始化方法如下)将是实现spring-boot-starter-\*(mybatis、redis等)自动化配置的关键，包括spring.factories的加载，bean的实例化等核心工作。

```

AbstractApplicationContext refresh()
{
 prepareBeanFactory(beanFactory);

 try {
 // Allows post-processing of the bean factory in context subclasses.
 postProcessBeanFactory(beanFactory);

 // Invoke factory processors registered as beans in the context.
 invokeBeanFactoryPostProcessors(beanFactory);

 // Register bean processors that intercept bean creation.
 registerBeanPostProcessors(beanFactory);

 // Initialize message source for this context.
 initMessageSource();

 // Initialize event multicaster for this context.
 initApplicationEventMulticaster();

 // Initialize other special beans in specific context subclasses.
 onRefresh();

 // Check for listener beans and register them.
 registerListeners();

 // Instantiate all remaining (non-lazy-init) singletons.
 finishBeanFactoryInitialization(beanFactory); beanFactory: "org.sprin

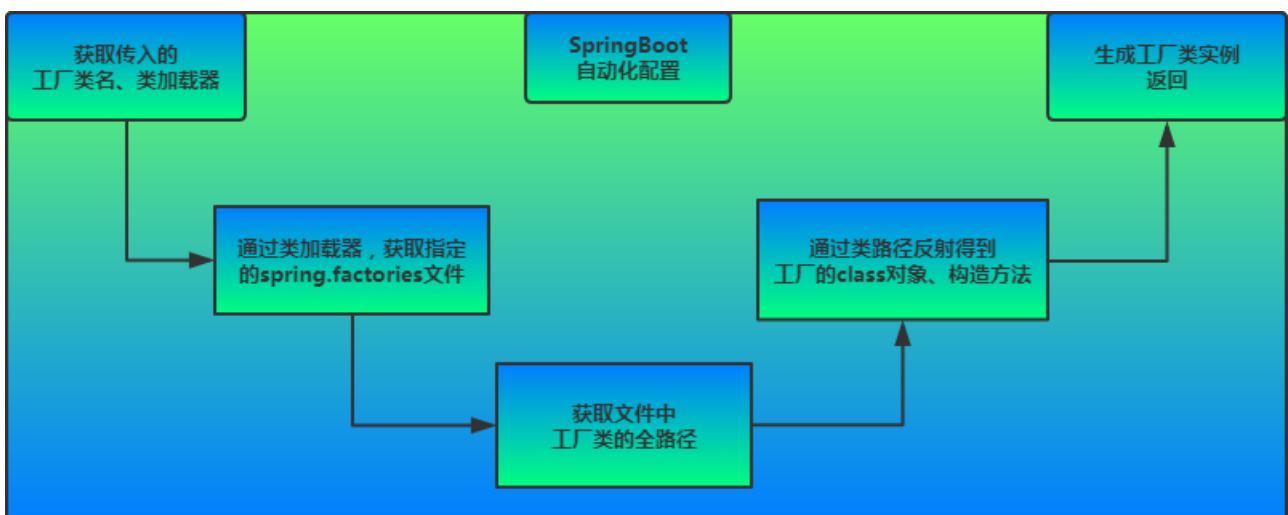
 // Last step: publish corresponding event.
 finishRefresh();
 }
}

```

配置结束后，Springboot做了一些基本的收尾工作，返回了应用环境上下文。回顾整体流程，Springboot的启动，主要创建了配置环境(environment)、事件监听(listeners)、应用上下文(applicationContext)，并基于以上条件，在容器中开始实例化我们需要的Bean，至此，通过SpringBoot启动的程序已经构造完成，接下来我们来探讨自动化配置是如何实现。

#### 自动化配置：

之前的启动结构图中，我们注意到无论是应用初始化还是具体的执行过程，都调用了SpringBoot自动配置模块。



该配置模块的主要使用到了SpringFactoriesLoader，即Spring工厂加载器，该对象提供了loadFactoryNames方法，入参为factoryClass和classLoader，即需要传入上图中的工厂类名称和对应的类加载器，方法会根据指定的classLoader，加载该类加载器搜索路径下的指定文件，即spring.factories文件，传入的工厂类为接口，而文件中对应的类则是接口的实现类，或最终作为实现类，所以文件中一般为如下图这种一对多的类名集合，获取到这些实现类的类名后，loadFactoryNames方法返回类名集合，方法调用方得到这些集合后，再通过反射获取这些类的类对象、构造方法，最终生成实例。

```

Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer

Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

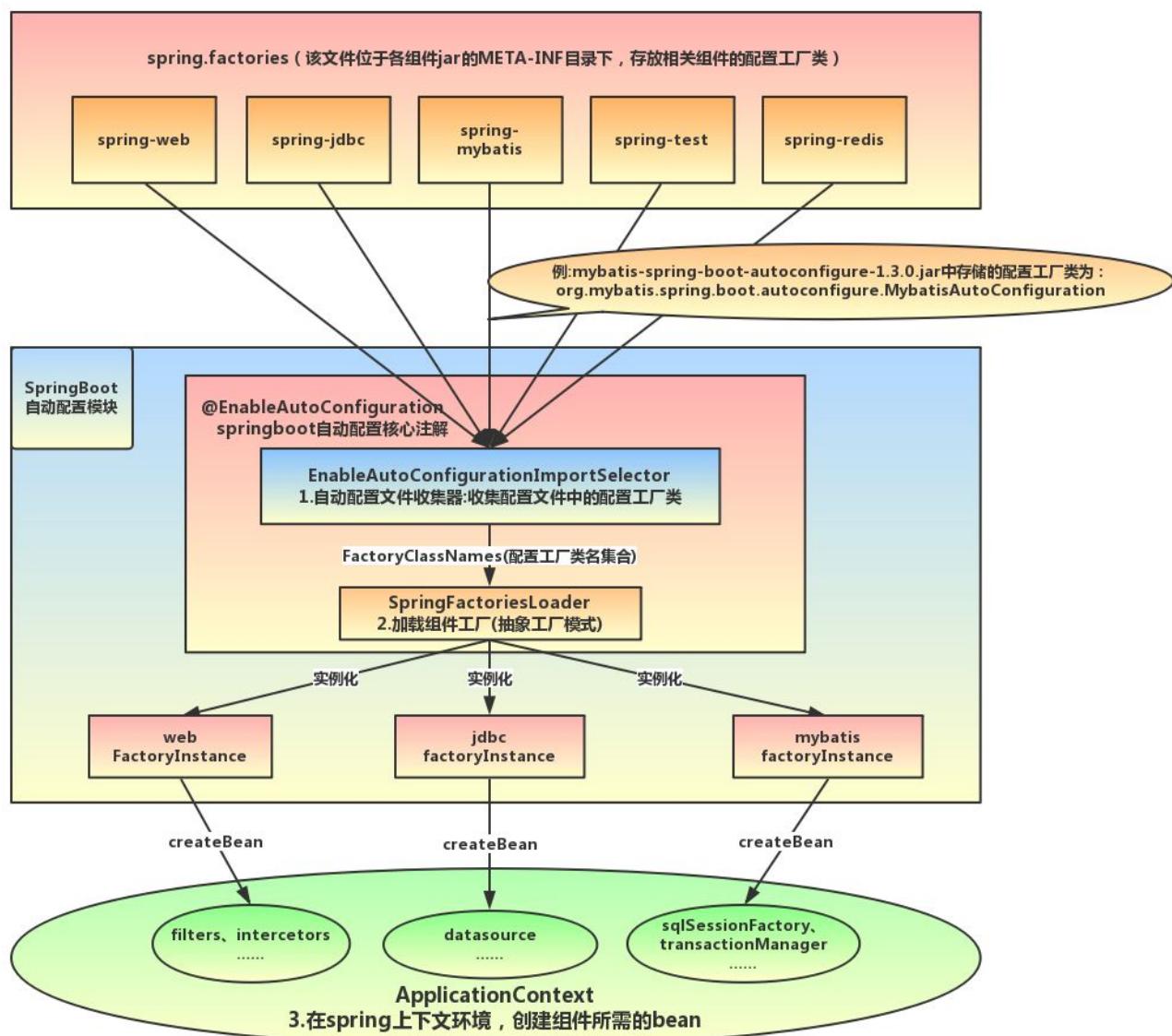
Auto Configuration Import Listeners
org.springframework.boot.autoconfigure.AutoConfigurationImportListener=\
org.springframework.boot.autoconfigure.condition.ConditionEvaluationReportAutoConfigurationImportListener

Auto Configuration Import Filters
org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\
org.springframework.boot.autoconfigure.condition.OnClassCondition

Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\

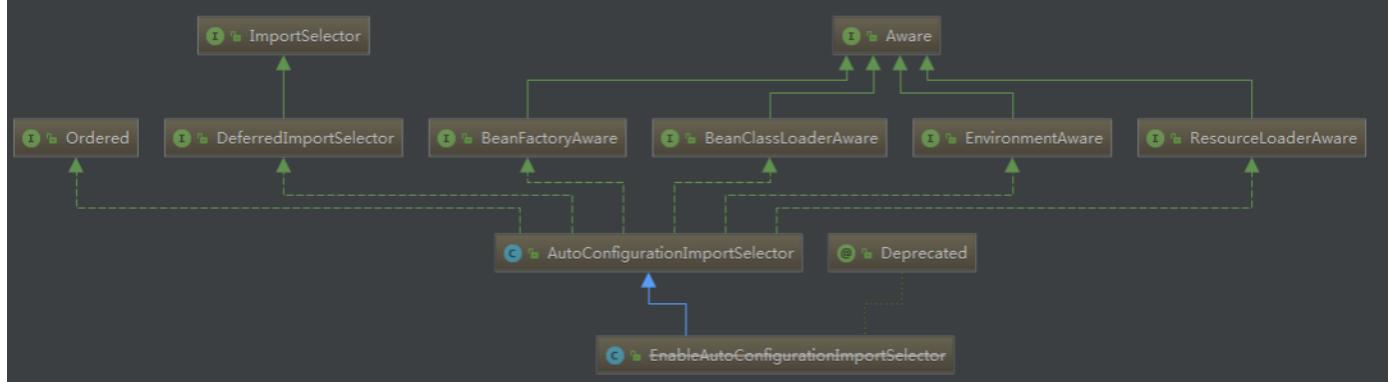

```

下图有助于我们形象理解自动配置流程。



mybatis-spring-boot-starter、spring-boot-starter-web等组件的META-INF文件下均含有spring.factories文件，自动配置模块中，SpringFactoriesLoader收集到文件中的类全名并返回一个类全名的数组，返回的类全名通过反射被实例化，就形成了具体的工厂实例，工厂实例来生成组件具体需要的bean。

之前我们提到了EnableAutoConfiguration注解，其类图如下：



可以发现其最终实现了ImportSelector(选择器)和BeanClassLoaderAware(bean类加载器中间件)，重点关注一下AutoConfigurationImportSelector的selectImports方法。

```

@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
 if (!isEnabled(annotationMetadata)) {
 return NO_IMPORTS;
 }
 try {
 AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
 .loadMetadata(this.beanClassLoader);
 AnnotationAttributes attributes = getAttributes(annotationMetadata);
 List<String> configurations = getCandidateConfigurations(annotationMetadata,
 attributes);
 configurations = removeDuplicates(configurations);
 configurations = sort(configurations, autoConfigurationMetadata);
 Set<String> exclusions = getExclusions(annotationMetadata, attributes);
 checkExcludedClasses(configurations, exclusions);
 configurations.removeAll(exclusions);
 configurations = filter(configurations, autoConfigurationMetadata);
 fireAutoConfigurationImportEvents(configurations, exclusions);
 return configurations.toArray(new String[configurations.size()]);
 }
 catch (IOException ex) {
 throw new IllegalStateException(ex);
 }
}

```

该方法在springboot启动流程——bean实例化前被执行，返回要实例化的类信息列表。我们知道，如果获取到类信息，spring自然可以通过类加载器将类加载到jvm中，现在我们已经通过spring-boot的starter依赖方式依赖了我们需要的组件，那么这些组建的类信息在select方法中也是可以被获取到的，不要急我们继续向下分析。

```

/**
 * Return the auto-configuration class names that should be considered. By default
 * this method will load candidates using {@link SpringFactoriesLoader} with
 * {@link #getSpringFactoriesLoaderFactoryClass()}.
 * @param metadata the source metadata
 * @param attributes the {@link #getAttributes(AnnotationMetadata)} annotation
 * attributes
 * @return a list of candidate configurations
 */
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
 AnnotationAttributes attributes) {
 List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
 getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
 Assert.notEmpty(configurations,
 message: "No auto configuration classes found in META-INF/spring.factories. If you "
 + "are using a custom packaging, make sure that file is correct.");
 return configurations;
}

```

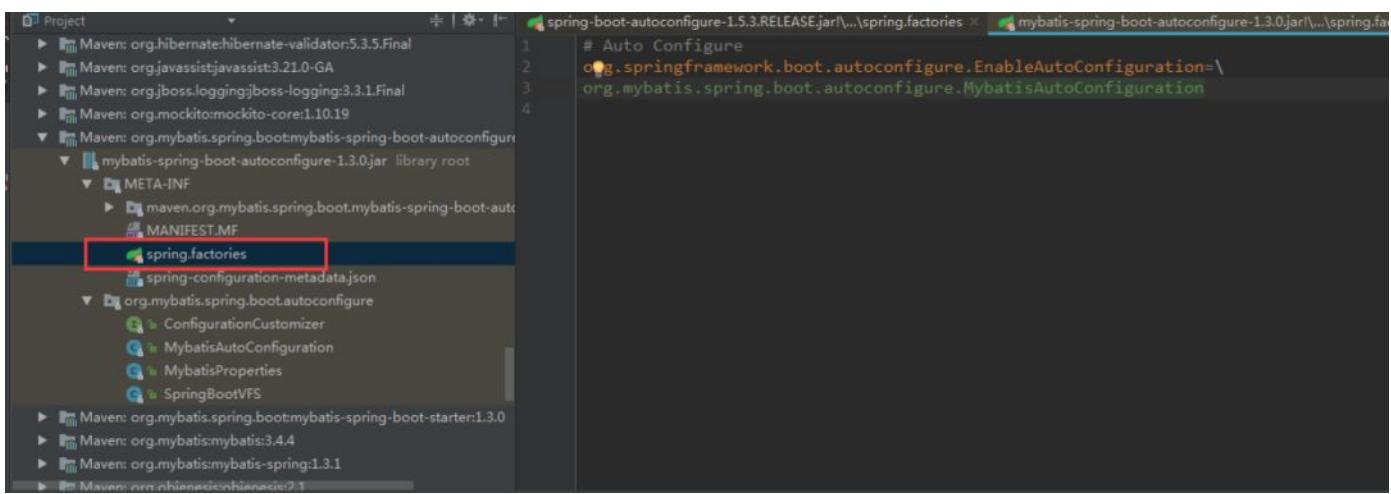
该方法中的getCandidateConfigurations方法，通过方法注释了解到，其返回一个自动配置类的类名列表，方法调用了loadFactoryNames方法，查看该方法

```

 public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {
 String factoryClassName = factoryClass.getName();
 try {
 Enumeration<URL> urls = (classLoader != null ? classLoader.getResources(FACTORIES_RESOURCE_LOCATION) : urls);
 ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
 List<String> result = new ArrayList<String>();
 result.size() = 0;
 while (urls.hasMoreElements()) {
 URL url = urls.nextElement();
 Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource(url));
 properties.size() = 1;
 url: "jar:file:/D:/work/repository/org/mybatis/spring/boot/mybatis-spring-boot-autoconfigure-1.3.0.jar!/META-INF/spring.factories"
 String factoryClassNames = properties.getProperty(factoryClassName);
 factoryClassNames: null
 properties.size() = 1
 result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(factoryClassNames)));
 result.size() = 0
 }
 } catch (IOException ex) {
 throw new IllegalArgumentException("Unable to load [" + factoryClass.getName() +
 "] factories from location [" + FACTORIES_RESOURCE_LOCATION + "]", ex);
 }
 }
}

```

在上面的代码可以看到自动配置器会根据传入的factoryClass.getName()到项目系统路径下所有的spring.factories文件中找到相应的key，从而加载里面的类。我们就选取这个mybatis-spring-boot-autoconfigure下的spring.factories文件



进入org.mybatis.spring.boot.autoconfigure.MybatisAutoConfiguration中，主要看一下类头：

```

1 /*
2 * package org.mybatis.spring.boot.autoconfigure;
3 *
4 * import ...
5 */
6 /**
7 * {@Link EnableAutoConfiguration Auto-Configuration} for Mybatis. Contributes a
8 * {@Link SqlSessionFactory} and a {@Link SqlSessionTemplate}.
9 *
10 * If {@Link org.mybatis.spring.annotation.MapperScan} is used, or a
11 * configuration file is specified as a property, those will be considered,
12 * otherwise this auto-configuration will attempt to register mappers based on
13 * the interface definitions in or under the root auto-configuration package.
14 *
15 * @author Eddú Meléndez
16 * @author Josh Long
17 * @author Kazuki Shimizu
18 * @author Eduardo Macarrón
19 */
20
21 @org.springframework.context.annotation.Configuration
22 @ConditionalOnClass({ SqlSessionFactory.class, SqlSessionFactoryBean.class })
23 @ConditionalOnBean(DataSource.class)
24 @EnableConfigurationProperties(MybatisProperties.class)
25 @AutoConfigureAfter(DataSourceAutoConfiguration.class)
26 public class MybatisAutoConfiguration {
27
28 private static final Logger logger = LoggerFactory.getLogger(MybatisAutoConfiguration.class);
29
30 private final MybatisProperties properties;
31
32 private final Interceptor[] interceptors;
33
34 private final ResourceLoader resourceLoader;

```

发现Spring的@Configuration，俨然是一个通过注解标注的springBean，继续向下看，

@ConditionalOnClass({  
    SqlSessionFactory.class,  
    SqlSessionFactoryBean.class})这个注解的意思是：当存在  
SqlSessionFactory.class, SqlSessionFactoryBean.class这两个类时才解析MybatisAutoConfiguration配置类，否则不解析这  
一个配置类，make sence，我们需要mybatis为我们返回会话对象，就必须有会话工厂相关类。

@ConditionalOnBean(DataSource.class): 只有处理已经被声明为bean的dataSource。

@ConditionalOnMissingBean(MapperFactoryBean.class)这个注解的意思是如果容器中不存在name指定的bean则创建bean  
注入，否则不执行（该类源码较长，篇幅限制不全粘贴）

以上配置可以保证sqlSessionFactory、sqlSessionTemplate、dataSource等mybatis所需的组件均可被自动配  
置，@Configuration注解已经提供了Spring的上下文环境，所以以上组件的配置方式与Spring启动时通过mybatis.xml文件进  
行配置起到一个效果。通过分析我们可以发现，只要一个基于SpringBoot项目的类路径下存在SqlSessionFactory.class,  
SqlSessionFactoryBean.class，并且容器中已经注册了dataSourceBean，就可以触发自动化配置，意思说我们只要在maven  
的项目中加入了mybatis所需要的若干依赖，就可以触发自动配置，但引入mybatis原生依赖的话，每集成一个功能都要去修改  
其自动化配置类，那就得不到开箱即用的效果了。所以Spring-boot为我们提供了统一的starter可以直接配置好相关的类，触发  
自动配置所需的依赖(mybatis)如下：

```

<!-- MyBatis -->
<dependency>
 <groupId>org.mybatis.spring.boot</groupId>
 <artifactId>mybatis-spring-boot-starter</artifactId>
 <version>${mybatis-springboot.version}</version>
</dependency>

```

这里是截取的mybatis-spring-boot-starter的源码中pom.xml文件中所有依赖：

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

因为maven依赖的传递性，我们只要依赖starter就可以依赖到所有需要自动配置的类，实现开箱即用的功能。也体现出Springboot简化了Spring框架带来的大量XML配置以及复杂的依赖管理，让开发人员可以更加关注业务逻辑的开发。

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



- [1. 一键下载 Pornhub 视频！](#)
- [2. 一个女生不主动联系你还有机会吗？](#)
- [3. 程序员开发了一款软件，完成了舔狗的绝地反杀](#)
- [4. Spring Boot 多模块项目实践\(附打包方法\)](#)
- [5. 团队开发中 Git 最佳实践](#)



喜欢文章，点个在看 

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring MVC 到 Spring Boot 的简化之路

beyondlicg Java后端 2019-11-12

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | beyondlicg

来源 | [juejin.im/post/5aa22d1f51882555677e2492](https://juejin.im/post/5aa22d1f51882555677e2492)

## 背景

从Servlet技术到Spring和Spring MVC，开发Web应用变得越来越简捷。但是Spring和Spring MVC的众多配置有时却让人望而却步，相信有过Spring MVC开发经验的朋友能深刻体会到这一痛苦。

因为即使是开发一个Hello-World的Web应用，都需要我们在pom文件中导入各种依赖，编写web.xml、spring.xml、springmvc.xml配置文件等。特别是需要导入大量的jar包依赖时，我们需要在网上查找各种jar包资源，各个jar间可能存在着各种依赖关系，这时候又得下载其依赖的jar包，有时候jar包间还存在着严格的版本要求，所以当我们只是想开发一个Hello-World的超简单的Web应用时，却把绝大部分的时间在花在了编写配置文件和导入jar包依赖上，极大地影响了我们的开发效率。

所以为了简化Spring繁杂的配置，Spring Boot应运而生。正如Spring Boot的名称一样，一键启动，Spring Boot提供了自动配置功能，为我们提供了开箱即用的功能，使我们将重心放在业务逻辑的开发上。那么Spring Boot又是怎么简化Spring MVC的呢？Spring Boot和Spring、Spring MVC间又是怎样的关系呢？Spring Boot又有什么新特点呢？接下来，让我们走进Spring MVC 到 Spring Boot的简化之路，或许你就能找到这些答案。

## Spring vs Spring MVC vs Spring Boot

- Spring Boot和Spring、Spring MVC不是竞争关系，Spring Boot使我们更加容易使用Spring和Spring MVC

## Spring FrameWork

- Spring FrameWork解决的核心问题是什么 Spring框架的最重要特性是依赖注入，所有的Spring模块的核心都是依赖注入（DI）或控制反转（IOC）。为什么很重要呢，因为当我们使用DI或IOC时，我们可以使应用得到解耦。我们来看一个简单的例子：

没有依赖注入的例子：

```
@RestController
public class WelcomeController {

 private WelcomeService service = new WelcomeService();

 @RequestMapping("/welcome")
 public String welcome() {
 return service.retrieveWelcomeMessage();
 }
}
```

WelcomeService service = new WelcomeService(); 意味着WelcomeController类与WelcomeService类紧密结合在一起，耦合度高。

使用依赖注入的例子：

```
@Component
public class WelcomeService {
 //Bla Bla Bla
}
```

```
@RestController
public class WelcomeController {

 @Autowired
 private WelcomeService service;

 @RequestMapping("/welcome")
 public String welcome() {
 return service.retrieveWelcomeMessage();
 }
}
```

依赖注入使世界看起来更简单，我们让**Spring**框架做了辛勤的工作：

**@Component**: 我们告诉**Spring**框架-嘿，这是一个你需要管理的**bean**

**@Autowired**: 我们告诉**Spring**框架-嘿，找到这个特定类型的正确匹配并自动装入它

## Spring 还能解决什么问题

### 1. 重复代码

Spring框架停止了依赖注入（DI）吗？没有，它在依赖注入（DI）的核心概念上开发了许多**Spring**模块：

- Spring JDBC
- Spring MVC
- Spring AOP
- Spring ORM
- Spring Test
- ...

考虑一下**Spring JDBC**，这些模块带来了新功能吗？并没有，我们完全可以使用Java代码完成这些工作。那么，它们给我们带来了什么？它们带来了简单的抽象，这些简单抽象的目的是：

1. 减少样板代码/减少重复
2. 促进解耦/增加单元可测性 例如：与传统的JDBC相比，我们使用**Spring JDBC**需要编写的代码减少了许多。

### 2. 与其他框架良好的集成

Spring框架并不尝试去解决已经解决了的问题，它所做的一切就是提供与提供出色解决方案的框架的完美集成。

- Hibernate
- iBatis
- JUnit
- ...

## Spring MVC

- **Spring MVC**框架解决的核心问题是什么 **Spring MVC**框架提供了开发Web应用的分离方式。通过DispatcherServlet、ModelAndView、View Resolver等简单概念，是Web应用开发变得更加简单。

## 为什么需要**Spring Boot**

基于Spring的应用程序有很多配置。当我们使用Spring MVC时，我们需要配置组件扫描，调度器servlet，视图解析器等：

视图解析器配置：

```
<bean
 class="org.springframework.web.servlet.view.InternalResourceViewResolver">
 <property name="prefix">
 <value>/WEB-INF/views/</value>
 </property>
 <property name="suffix">
 <value>.jsp</value>
 </property>
</bean>

<mvc:resources mapping="/webjars/**" location="/webjars/" />
```

前端调度器的典型配置：

```
<servlet>
 <servlet-name>dispatcher</servlet-name>
 <servlet-class>
 org.springframework.web.servlet.DispatcherServlet
 </servlet-class>
 <init-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>/WEB-INF/todo-servlet.xml</param-value>
 </init-param>
 <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
 <servlet-name>dispatcher</servlet-name>
 <url-pattern>/</url-pattern>
</servlet-mapping>
```

当我们使用Hibernate / JPA时，我们需要配置一个数据源，一个实体管理器工厂，一个事务管理器以及许多其他事物：

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
 destroy-method="close">
 <property name="driverClass" value="${db.driver}" />
 <property name="jdbcUrl" value="${db.url}" />
 <property name="user" value="${db.username}" />
 <property name="password" value="${db.password}" />
</bean>

<jdbc:initialize-database data-source="dataSource">
 <jdbc:script location="classpath:config/schema.sql" />
 <jdbc:script location="classpath:config/data.sql" />
</jdbc:initialize-database>

<bean
 class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
 id="entityManagerFactory">
 <property name="persistenceUnitName" value="hsqL_pu" />
 <property name="dataSource" ref="dataSource" />
</bean>

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
 <property name="entityManagerFactory" ref="entityManagerFactory" />
 <property name="dataSource" ref="dataSource" />
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>
```

**Spring Boot解决的问题**

## 1. Spring Boot 自动配置

Spring引入了新的思维过程：我们可以变得更加智能些吗？当一个spring mvc jar包被添加到应用程序时，我们是否可以自动配置一些bean？

1. 当Hibernate jar包在类路径时，自动配置数据源怎样？
  2. 当Spring MVC jar包在类路径时，自动配置Dispatcher Servlet怎样？
- Spring Boot查看CLASSPATH上对于本应用程序需要编写配置的框架，基于这些，Spring Boot提供了这些框架的基本配置-这就是自动配置。

## 2. Spring Boot Starter Projects

假设我们想开发一个Web应用程序。首先，我们需要确定我们想要使用的框架，使用哪个版本的框架以及如何将它们连接在一起。所有Web应用程序都有类似的需求。下面列出的是我们在Spring MVC中使用的一些依赖关系。这些包括Spring MVC，Jackson Databind（用于数据绑定），Hibernate-Validator（用于使用Java验证API的服务器端验证）和Log4j（用于日志记录）。在创建时，我们必须选择所有这些框架的兼容版本：

```
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-webmvc</artifactId>
 <version>4.2.2.RELEASE</version>
</dependency>

<dependency>
 <groupId>com.fasterxml.jackson.core</groupId>
 <artifactId>jackson-databind</artifactId>
 <version>2.5.3</version>
</dependency>

<dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-validator</artifactId>
 <version>5.0.2.Final</version>
</dependency>

<dependency>
 <groupId>log4j</groupId>
 <artifactId>log4j</artifactId>
 <version>1.2.17</version>
</dependency>
```

- 什么是Starter

Starter是一套方便的依赖描述符，可以包含在应用程序中。

你可以获得所需的所有Spring及相关技术的一站式服务，而无需搜索示例代码并复制依赖描述符的粘贴负载。

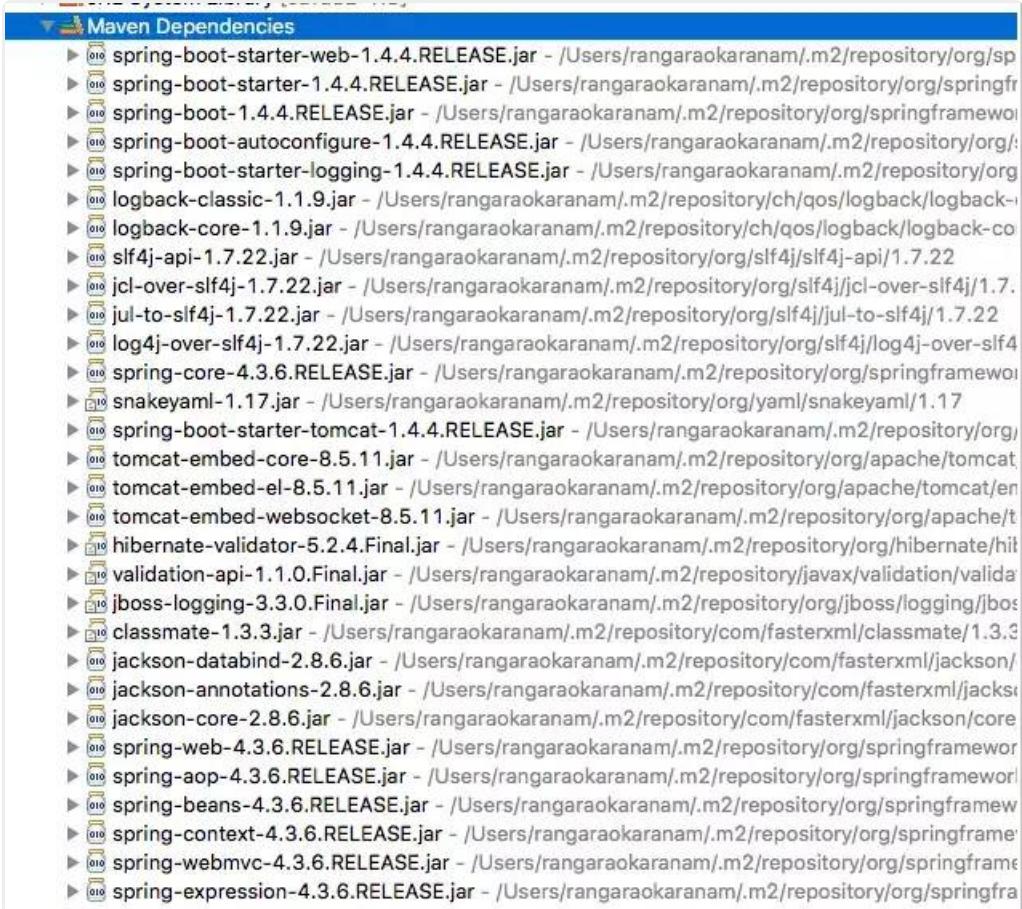
例如，如果你想开始使用Spring和JPA来访问数据库，只需在你的项目中包含spring-boot-starter-data-jpa依赖项就好。

我们来看Starter的一个示例 - Spring-Boot-Starter-Web

Spring-Boot-Starter-Web依赖：

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

以下屏幕截图显示了添加到我们的应用程序中的不同依赖关系：



img

任何典型的Web应用程序都会使用所有这些依赖项。Spring Boot Starter Web预先打包了这些。作为开发人员，我们不需要担心这些依赖关系或兼容版本。

### 3. Spring Boot Starter项目选项

正如Spring Boot Starter Web一样，Starter项目帮助我们快速入门开发特定类型的应用程序：

- spring-boot-starter-web-services - SOAP Web服务
- spring-boot-starter-web - Web和RESTful应用程序
- spring-boot-starter-test - 单元测试和集成测试
- spring-boot-starter-data-jpa - 带有Hibernate的Spring Data JPA
- spring-boot-starter-cache - 启用Spring Framework的缓存支持
- ...

### 什么是Spring Boot 自动配置

前面已经初步介绍过，在这里详细介绍一下。

当我们启动Spring Boot应用程序时，我们可以在日志中看到一些重要的消息。

```
Mapping servlet: 'dispatcherServlet' to [/]
```

```
Mapped "[/{error}]" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> or
```

```
Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHan
```

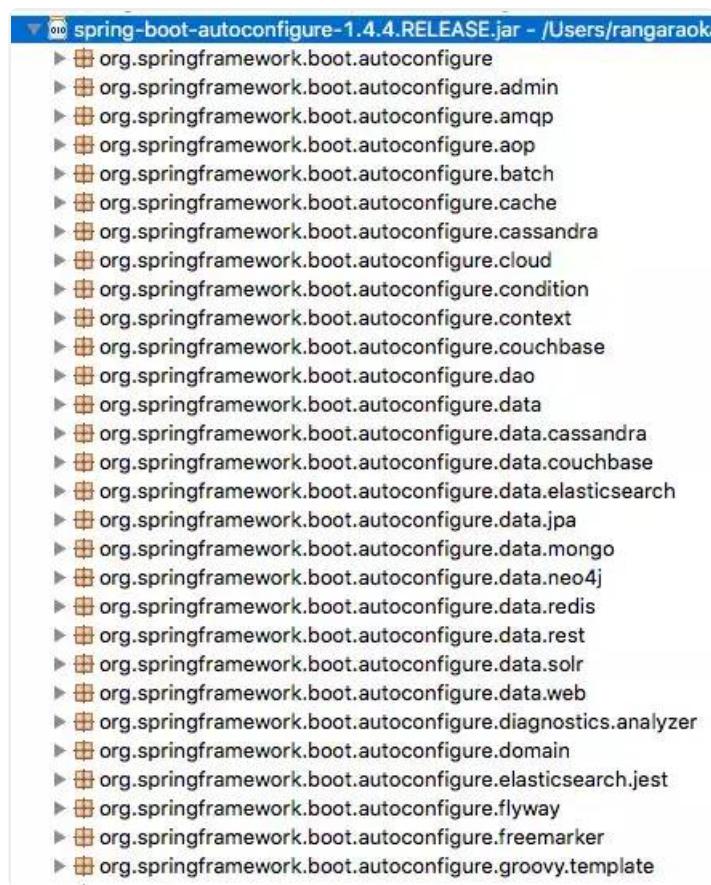
上面的日志语句显示了Spring Boot Auto Configuration的行为。

一当我们在应用中添加了Spring Boot Starter Web依赖，Spring Boot AutoConfiguration就会发现Spring MVC在类路径下，它会自动配置dispatcherServlet，一个默认的错误页面和webjars。

如果你添加了Spring Boot DataJPA Starter依赖，Spring Boot AutoConfiguration会自动配置数据源（datasource）和实体管理器（Entity Manager）

## Spring Boot Auto Configuration在哪里实现

所有的自动配置逻辑都在spring-boot-autoconfigure.jar中实现。mvc、data和其他框架的所有自动配置逻辑都存在与一个jar包中。



img

spring-boot-autoconfigure.jar中重要的文件是/META-INF/spring.factories，该文件；列出了在EnableAutoConfiguration key下启动的所有自动配置类。下面列出一些重要的配置类：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.SecurityFilterAutoConfiguration,\
org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration,\
```

## 查看自动配置

打开调试日志

在application.properties打开调试日志：

```
logging.level.org.springframework: DEBUG
```

## 总结

Spring Boot的出现本身就是为了减低Web开发的门槛，使开发人员能够专注于业务开发，而不需浪费时间在业务开发之外，至此Spring MVC到Spring Boot的简化之路到此结束。

- E N D -



喜欢文章，点个在看 

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring 和 Spring Boot 之间到底有啥区别？

乐傻驴 Java后端 1周前

# Java后端

作者：乐傻驴

链接：[jianshu.com/p/ffe5ebe17c3a](https://jianshu.com/p/ffe5ebe17c3a)

## 概述

对于 Spring 和 SpringBoot 到底有什么区别，我听到了很多答案，刚开始迈入学习 SpringBoot 的我当时也是一头雾水，随着经验的积累、我慢慢理解了这两个框架到底有什么区别，相信对于用了 SpringBoot 很久的同学来说，还不是很理解 SpringBoot 到底和 Spring 有什么区别，看完文章中的比较，或许你有了不同的答案和看法！

## 什么是Spring

作为 Java 开发人员，大家都 Spring 都不陌生，简而言之， Spring 框架为开发 Java 应用程序提供了全面的基础架构支持。它包含一些很好的功能，如依赖注入和开箱即用的模块，如：

SpringJDBC、SpringMVC、SpringSecurity、SpringAOP、SpringORM、SpringTest，这些模块缩短应用程序的开发时间，提高了应用开发的效率例如，在 JavaWeb 开发的早期阶段，我们需要编写大量的代码来将记录插入到数据库中。但是通过使用 SpringJDBC 模块的 JDBCTemplate，我们可以将操作简化为几行代码。

## 什么是Spring Boot

SpringBoot 基本上是 Spring 框架的扩展，它消除了设置 Spring 应用程序所需的 XML 配置，为更快，更高效的开发生态系统铺平了道路。

### SpringBoot 中的一些特征：

- 1、创建独立的 Spring 应用。
- 2、嵌入式 Tomcat、Jetty、Undertow 容器（无需部署 war 文件）。
- 3、提供的 starters 简化构建配置
- 4、尽可能自动配置 spring 应用。
- 5、提供生产指标，例如指标、健壮检查和外部化配置
- 6、完全没有代码生成和 XML 配置要求

## 从配置分析

### Maven 依赖

首先，让我们看一下使用 Spring 创建 Web 应用程序所需的最小依赖项

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-web</artifactId>
<version>5.1.0.RELEASE</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>5.1.0.RELEASE</version>
</dependency>
```

**与Spring不同，Spring Boot只需要一个依赖项来启动和运行Web应用程序：**

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<version>2.0.6.RELEASE</version>
</dependency>
```

**在进行构建期间，所有其他依赖项将自动添加到项目中。**

另一个很好的例子就是测试库。我们通常使用 SpringTest，JUnit，Hamcrest 和 Mockito 库。在 Spring 项目中，我们应该将所有这些库添加为依赖项。但是在 SpringBoot 中，我们只需要添加 spring-boot-starter-test 依赖项来自动包含这些库。

**Spring Boot 为不同的 Spring 模块提供了许多依赖项。一些最常用的是：**

spring-boot-starter-data-jpa spring-boot-starter-security spring-boot-starter-test spring-boot-starter-web  
spring-boot-starter-thymeleaf

有关 starter 的完整列表，请查看 Spring 文档。

## MVC 配置

**让我们来看一下 Spring 和 SpringBoot 创建 JSPWeb 应用程序所需的配置。**

Spring 需要定义调度程序 servlet，映射和其他支持配置。我们可以使用 web.xml 文件或 Initializer 类来完成此操作：

```
public class MyWebAppInitializer implements WebApplicationInitializer {
 @Override
 public void onStartup(ServletContext container) {
 AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
 context.setConfigLocation("com.pingfangushi");
 container.addListener(new ContextLoaderListener(context));
 ServletRegistration.Dynamic dispatcher = container
 .addServlet("dispatcher", new DispatcherServlet(context));
 dispatcher.setLoadOnStartup(1);
 dispatcher.addMapping("/");
 }
}
```

还需要将 @EnableWebMvc 注释添加到 @Configuration 类，并定义一个视图解析器来解析从控制器返回的视图：

```

@EnableWebMvc
@Configuration
public class ClientWebConfig implements WebMvcConfigurer {
 @Bean
 public ViewResolver viewResolver() {
 InternalResourceViewResolver bean
 = new InternalResourceViewResolver();
 bean.setViewClass(JstlView.class);
 bean.setPrefix("/WEB-INF/view/");
 bean.setSuffix(".jsp");
 return bean;
 }
}

```

再来看 SpringBoot 一旦我们添加了 Web 启动程序，SpringBoot 只需要在 application 配置文件中配置几个属性来完成如上操作：

```

spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp

```

**上面的所有 Spring 配置都是通过一个名为 auto-configuration 的过程添加 Bootweb starter 来自动包含的。**

这意味着 SpringBoot 将查看应用程序中存在的依赖项，属性和 bean，并根据这些依赖项，对属性和 bean 进行配置。当然，如果我们想要添加自己的自定义配置，那么 SpringBoot 自动配置将会退回。

## 配置模板引擎

现在我们来看下如何在 Spring 和 Spring Boot 中配置 Thymeleaf 模板引擎。

在 Spring 中，我们需要为视图解析器添加 thymeleaf-spring5 依赖项和一些配置：

```

@Configuration
@EnableWebMvc
public class MvcWebConfig implements WebMvcConfigurer {
 @Autowired
 private ApplicationContext applicationContext;

 @Bean
 public SpringResourceTemplateResolver templateResolver() {
 SpringResourceTemplateResolver templateResolver = new SpringResourceTemplateResolver();
 templateResolver.setApplicationContext(applicationContext);
 templateResolver.setPrefix("/WEB-INF/views/");
 templateResolver.setSuffix(".html");
 return templateResolver;
 }

 @Bean
 public SpringTemplateEngine templateEngine() {
 SpringTemplateEngine templateEngine = new SpringTemplateEngine();
 templateEngine.setTemplateResolver(templateResolver());
 templateEngine.setEnableSpringELCompiler(true);
 return templateEngine;
 }

 @Override
 public void configureViewResolvers(ViewResolverRegistry registry) {
 ThymeleafViewResolver resolver = new ThymeleafViewResolver();
 resolver.setTemplateEngine(templateEngine());
 registry.viewResolver(resolver);
 }
}

```

SpringBoot1X 只需要 spring-boot-starter-thymeleaf 的依赖项来启用 Web 应用程序中的 Thymeleaf 支持。但是由于 Thymeleaf3.0 中的新功能，我们必须将 thymeleaf-layout-dialect 添加为 SpringBoot2X Web 应用程序中的依赖项。配置好依

赖，我们就可以将模板添加到 src/main/resources/templates文件夹中， SpringBoot将自动显示它们。

## Spring Security 配置

为简单起见，我们使用框架默认的 HTTPBasic 身份验证。让我们首先看一下使用 Spring 启用 Security 所需的依赖关系和配置。

Spring 首先需要依赖 spring-security-web 和 spring-security-config 模块。接下来，我们需要添加一个扩展 WebSecurityConfigurerAdapter 的类，并使用 @EnableWebSecurity 注解：

```
@Configuration
@EnableWebSecurity
public class CustomWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter {

 @Autowired
 public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
 auth.inMemoryAuthentication()
 .withUser("admin")
 .password(passwordEncoder())
 .encode("password")
 .authorities("ROLE_ADMIN");
 }

 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http.authorizeRequests()
 .anyRequest().authenticated()
 .and()
 .httpBasic();
 }

 @Bean
 public PasswordEncoder passwordEncoder() {
 return new BCryptPasswordEncoder();
 }
}
```

这里我们使用 inMemoryAuthentication 来设置身份验证。同样，SpringBoot 也需要这些依赖项才能使其工作。但是我们只需要定义 spring-boot-starter-security 的依赖关系，因为这会自动将所有相关的依赖项添加到类路径中。

**SpringBoot 中的安全配置与上面的相同。**

## 应用程序启动引导配置

Spring 和 SpringBoot 中应用程序引导的基本区别在于 servlet。Spring 使用 web.xml 或 SpringServletContainerInitializer 作为其引导入口点。SpringBoot 仅使用 Servlet3 功能来引导应用程序，下面让我们详细了解下

## Spring 引导配置

Spring 支持传统的 web.xml 引导方式以及最新的 Servlet3+ 方法。

### 配置 web.xml 方法启动的步骤

Servlet 容器（服务器）读取 web.xml

web.xml 中定义的 DispatcherServlet 由容器实例化

DispatcherServlet 通过读取 WEB-INF/{servletName}-servlet.xml 来创建 WebApplicationContext。最后，DispatcherServlet 注册在应用程序上下文中定义的 bean

### 使用 Servlet3+ 方法的 Spring 启动步骤

容器搜索实现 ServletContainerInitializer 的类并执行 SpringServletContainerInitializer 找到实现所有类 WebApplicationInitializer `` WebApplicationInitializer 创建具有 XML 或上下文 @Configuration 类

## SpringBoot 引导配置

Spring Boot 应用程序的入口点是使用 @SpringBootApplication 注释的类

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

默认情况下，SpringBoot 使用嵌入式容器来运行应用程序。在这种情况下，SpringBoot 使用 publicstaticvoidmain 入口点来启动嵌入式 Web 服务器。此外，它还负责将 Servlet，Filter 和 ServletContextInitializerbean 从应用程序上下文绑定到嵌入式 servlet 容器。SpringBoot 的另一个特性是它会自动扫描同一个包中的所有类或 Main 类的子包中的组件。

SpringBoot 提供了将其部署到外部容器的方式。我们只需要扩展 SpringBootServletInitializer 即可：

```
/**
 * War 部署
 *
 * @author SanLi
 * Created by 2689170096@qq.com on 2018/4/15
 */
public class ServletInitializer extends SpringBootServletInitializer {

 @Override
 protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
 return application.sources(Application.class);
 }

 @Override
 public void onStartup(ServletContext servletContext) throws ServletException {
 super.onStartup(servletContext);
 servletContext.addListener(new HttpSessionEventPublisher());
 }
}
```

这里外部 servlet 容器查找在 war 包下的 META-INF 文件夹下 MANIFEST.MF 文件中定义的 Main-class，SpringBootServletInitializer 将负责绑定 Servlet，Filter 和 ServletContextInitializer。

## 打包和部署

最后，让我们看看如何打包和部署应用程序。这两个框架都支持 Maven 和 Gradle 等通用包管理技术。但是在部署方面，这些框架差异很大。例如，Spring Boot Maven 插件在 Maven 中提供 SpringBoot 支持。它还允许打包可执行 jar 或 war 包并就地运行应用程序。

在部署环境中 SpringBoot 对比 Spring 的一些优点包括：

- 1、提供嵌入式容器支持
- 2、使用命令 java -jar 独立运行 jar
- 3、在外部容器中部署时，可以选择排除依赖关系以避免潜在的 jar 冲突
- 4、部署时灵活指定配置文件的选项
- 5、用于集成测试的随机端口生成

## 结论

简而言之，我们可以说 SpringBoot 只是 Spring 本身的扩展，使开发，测试和部署更加方便。

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。同时标星（置顶）本公众号可以第一时间接受到博文推送。

#### 推荐阅读

[1. IDEA 新特性：提前知道代码怎么走](#)

[2. ping 命令还能这么玩？](#)

[3. String面试题](#)

[4. GitHub 移动端来了！](#)



微信搜一搜

Q Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Tomcat 在 Spring Boot 中是如何启动的

木木匠 Java后端 1月4日

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

---

作者 | 木木匠

链接 | [my.oschina.net/luozhou/blog/3088908](http://my.oschina.net/luozhou/blog/3088908)

## 前言

我们知道SpringBoot给我们带来了一个全新的开发体验，我们可以直接把web程序达成jar包，直接启动，这就得益于SpringBoot内置了容器，可以直接启动，本文将以Tomcat为例，来看看SpringBoot是如何启动Tomcat的，同时也将展开学习下Tomcat的源码，了解Tomcat的设计。

## 从 Main 方法说起

用过SpringBoot的人都知道，首先要写一个main方法来启动

我们直接点击run方法的源码，跟踪下来，发下最终的run方法是调用ConfigurableApplicationContext方法，源码如下：

```
public ConfigurableApplicationContext run(String... args) {
 StopWatch stopWatch = new StopWatch();
 stopWatch.start();
 ConfigurableApplicationContext context = null;
 Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();
 //设置系统属性『java.awt.headless』，为true则启用headless模式支持
 configureHeadlessProperty();
 //通过*SpringFactoriesLoader*检索*META-INF/spring.factories*,
 //找到声明的所有SpringApplicationRunListener的实现类并将其实例化,
 //之后逐个调用其started()方法，广播SpringBoot要开始执行了
 SpringApplicationRunListeners listeners = getRunListeners(args);
 //发布应用开始启动事件
 listeners.starting();
 try {
 //初始化参数
 ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
 //创建并配置当前SpringBoot应用将要使用的Environment (包括配置要使用的PropertySource以及Profile) ,
 //并遍历调用所有的SpringApplicationRunListener的environmentPrepared()方法，广播Environment准备完毕。
 ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);
 configureIgnoreBeanInfo(environment);
 //打印banner
 Banner printedBanner = printBanner(environment);
 //创建应用上下文
 context = createApplicationContext();
 //通过*SpringFactoriesLoader*检索*META-INF/spring.factories*，获取并实例化异常分析器
 exceptionReporters = getSpringFactoriesInstances(SpringBootExceptionReporter.class,
 new Class[] { ConfigurableApplicationContext.class }, context);
 //为ApplicationContext加载environment，之后逐个执行 ApplicationContextInitializer 的 initialize() 方法来进一步封装 ApplicationContext,
 //并调用所有的SpringApplicationRunListener的contextPrepared()方法，【EventPublishingRunListener只提供了一个空的contextPrepare
 //之后初始化IoC容器，并调用SpringApplicationRunListener的contextLoaded()方法，广播ApplicationContext的IoC加载完成,
 //这里就包括通过**@EnableAutoConfiguration**导入的各种自动配置类。
 prepareContext(context, environment, listeners, applicationArguments, printedBanner);
 //刷新上下文
 refreshContext(context);
 //再一次刷新上下文,其实是空方法，可能是为了后续扩展。
 afterRefresh(context, applicationArguments);
 stopWatch.stop();
 if (this.logStartupInfo) {
 new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), stopWatch);
 }
 //发布应用已经启动的事件
 listeners.started(context);
 //遍历所有注册的ApplicationRunner和CommandLineRunner，并执行其run()方法。
 //我们可以实现自己的ApplicationRunner或者CommandLineRunner，来对SpringBoot的启动过程进行扩展。
 callRunners(context, applicationArguments);
 }
 catch (Throwable ex) {
 handleRunFailure(context, ex, exceptionReporters, listeners);
 throw new IllegalStateException(ex);
 }

 try {
 //应用已经启动完成的监听事件
 listeners.running(context);
 }
 catch (Throwable ex) {
 handleRunFailure(context, ex, exceptionReporters, null);
 throw new IllegalStateException(ex);
 }
 return context;
}
```

其实这个方法我们可以简单的总结下步骤为

1. 配置属性
2. 获取监听器，发布应用开始启动事件
3. 初始化输入参数
4. 配置环境，输出banner
5. 创建上下文
6. 预处理上下文
7. 刷新上下文
8. 再刷新上下文
9. 发布应用已经启动事件
10. 发布应用启动完成事件

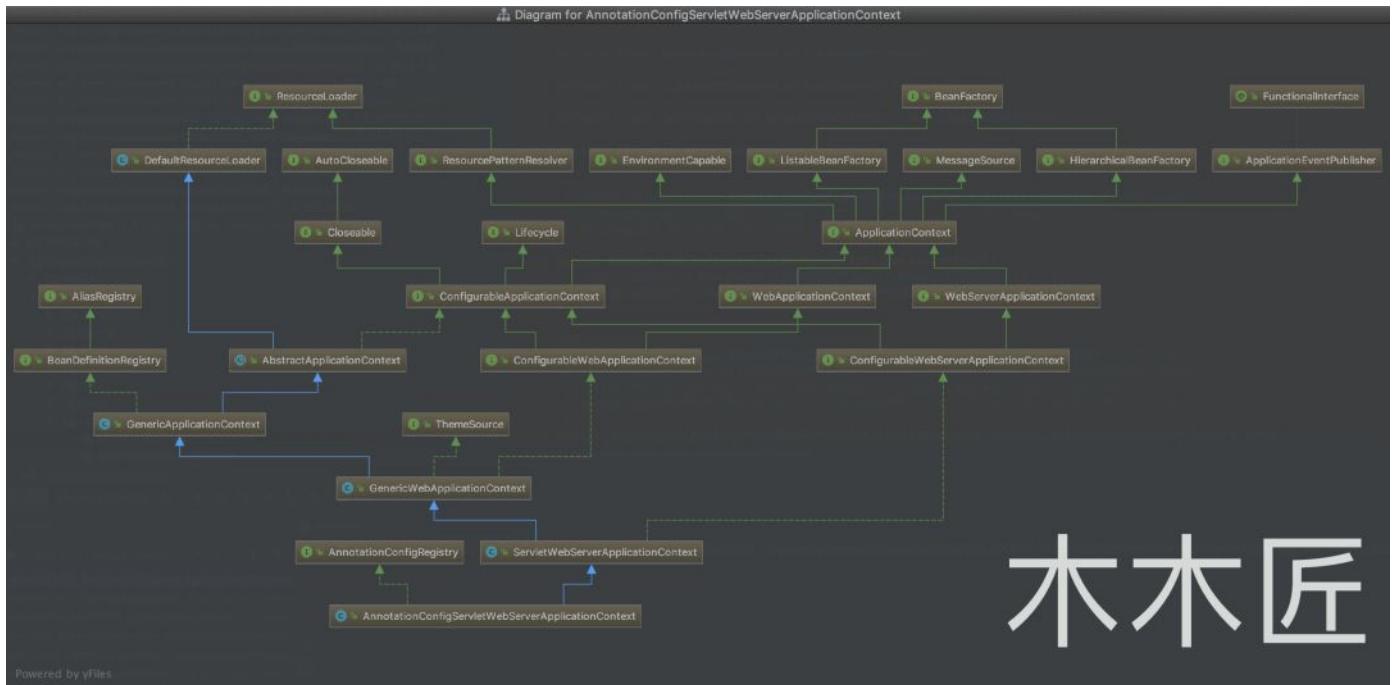
其实上面这段代码，如果只要分析tomcat内容的话，只需要关注两个内容即可，上下文是如何创建的，上下文是如何刷新的，分别对应的方法就是createApplicationContext() 和refreshContext(context)，接下来我们来看看这两个方法做了什么。

欢迎关注微信公众号：Java后端

```
protected ConfigurableApplicationContext createApplicationContext() {
 Class<!--?--> contextClass = this.applicationContextClass;
 if (contextClass == null) {
 try {
 switch (this.webApplicationType) {
 case SERVLET:
 contextClass = Class.forName(DEFAULT_SERVLET_WEB_CONTEXT_CLASS);
 break;
 case REACTIVE:
 contextClass = Class.forName(DEFAULT_REACTIVE_WEB_CONTEXT_CLASS);
 break;
 default:
 contextClass = Class.forName(DEFAULT_CONTEXT_CLASS);
 }
 } catch (ClassNotFoundException ex) {
 throw new IllegalStateException(
 "Unable create a default ApplicationContext, " + "please specify an ApplicationContextClass",
 ex);
 }
 }
 return (ConfigurableApplicationContext) BeanUtils.instantiateClass(contextClass);
}
```

这里就是根据我们的webApplicationType 来判断创建哪种类型的Servlet,代码中分别对应着Web类型(SERVLET),响应式Web类型 (REACTIVE),非Web类型 (default),我们建立的是Web类型，所以肯定实例化

DEFAULT\_SERVLET\_WEB\_CONTEXT\_CLASS指定的类，也就是AnnotationConfigServletWebServerApplicationContext类，我们来用图来说明下这个类的关系



# 木木匠

Powered by yFiles

通过这个类图我们可以知道，这个类继承的是`ServletWebServerApplicationContext`，这就是我们真正的主角，而这个类最终是继承了`AbstractApplicationContext`，了解完创建上下文的情况后，我们再来看看刷新上下文，相关代码如下：

```
//类: SpringApplication.java
private void refreshContext(ConfigurableApplicationContext context) {
 //直接调用刷新方法
 refresh(context);
 if (this.registerShutdownHook) {
 try {
 context.registerShutdownHook();
 }
 catch (AccessControlException ex) {
 // Not allowed in some environments.
 }
 }
}
```

```
//类: SpringApplication.java
protected void refresh(ApplicationContext applicationContext) {
 Assert.isInstanceOf(AbstractApplicationContext.class, applicationContext);
 ((AbstractApplicationContext) applicationContext).refresh();
}
```

这里还是直接传递调用本类的`refresh(context)`方法，最后是强转成父类`AbstractApplicationContext`调用其`refresh()`方法，该代码如下：

```
//类: AbstractApplicationContext
public void refresh() throws BeansException, IllegalStateException {
 synchronized (this.startupShutdownMonitor) {
 // Prepare this context for refreshing.
 prepareRefresh();

 // Tell the subclass to refresh the internal bean factory.
 ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

 // Prepare the bean factory for use in this context.
 prepareBeanFactory(beanFactory);

 try {
 // Allows post-processing of the bean factory in context subclasses.
 postProcessBeanFactory(beanFactory);
 }
```

```

// Invoke factory processors registered as beans in the context.
invokeBeanFactoryPostProcessors(beanFactory);

// Register bean processors that intercept bean creation.
registerBeanPostProcessors(beanFactory);

// Initialize message source for this context.
initMessageSource();

// Initialize event multicaster for this context.
initApplicationEventMulticaster();

// Initialize other special beans in specific context subclasses. 这里的意思就是调用各个子类的onRefresh()
onRefresh();

// Check for listener beans and register them.
registerListeners();

// Instantiate all remaining (non-lazy-init) singletons.
finishBeanFactoryInitialization(beanFactory);

// Last step: publish corresponding event.
finishRefresh();
}

catch (BeansException ex) {
 if (logger.isWarnEnabled()) {
 logger.warn("Exception encountered during context initialization - " +
 "cancelling refresh attempt: " + ex);
 }
}

// Destroy already created singletons to avoid dangling resources.
destroyBeans();

// Reset 'active' flag.
cancelRefresh(ex);

// Propagate exception to caller.
throw ex;
}

finally {
 // Reset common introspection caches in Spring's core, since we
 // might not ever need metadata for singleton beans anymore...
 resetCommonCaches();
}
}
}
}

```

这里我们看到onRefresh()方法是调用其子类的实现，根据我们上文的分析，我们这里的子类是ServletWebServerApplicationContext。

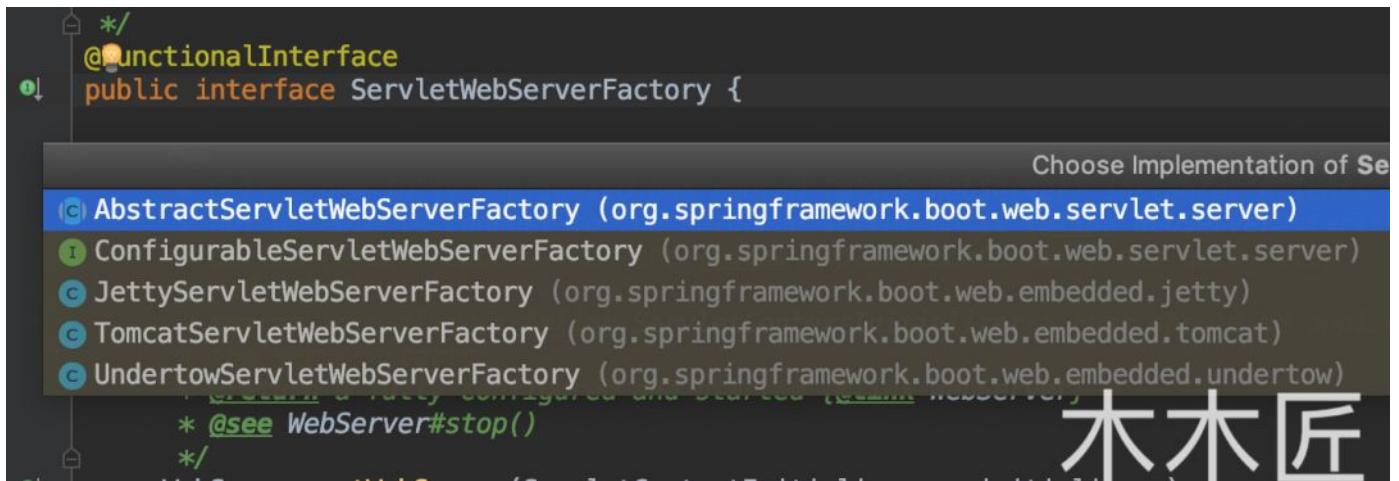
```

//类: ServletWebServerApplicationContext
protected void onRefresh() {
 super.onRefresh();
 try {
 createWebServer();
 }
 catch (Throwable ex) {
 throw new ApplicationContextException("Unable to start web server", ex);
 }
}

private void createWebServer() {
 WebServer webServer = this.webServer;
 ServletContext servletContext = getServletContext();
 if (webServer == null && servletContext == null) {
 ServletWebServerFactory factory = getWebServerFactory();
 this.webServer = factory.getWebServer(getSelfInitializer());
 }
 else if (servletContext != null) {
 try {
 getSelfInitializer().onStartup(servletContext);
 }
 catch (ServletException ex) {
 throw new ApplicationContextException("Cannot initialize servlet context", ex);
 }
 }
 initPropertySources();
}

```

到这里，其实庐山真面目已经出来了，createWebServer()就是启动web服务，但是还没有真正启动Tomcat，既然webServer是通过ServletWebServerFactory来获取的，我们就来看看这个工厂的真面目。



## 走进Tomcat内部

根据上图我们发现，工厂类是一个接口，各个具体服务的实现是由各个子类来实现的，所以我们就去看看TomcatServletWebServerFactory.getWebServer()的实现。

```

@Override
public WebServer getWebServer(ServletContextInitializer... initializers) {
 Tomcat tomcat = new Tomcat();
 File baseDir = (this.baseDirectory != null) ? this.baseDirectory : createTempDir("tomcat");
 tomcat.setBaseDir(baseDir.getAbsolutePath());
 Connector connector = new Connector(this.protocol);
 tomcat.getService().addConnector(connector);
 customizeConnector(connector);
 tomcat.setConnector(connector);
 tomcat.getHost().setAutoDeploy(false);
 configureEngine(tomcat.getEngine());
 for (Connector additionalConnector : this.additionalTomcatConnectors) {
 tomcat.getService().addConnector(additionalConnector);
 }
 prepareContext(tomcat.getHost(), initializers);
 return getTomcatWebServer(tomcat);
}

```

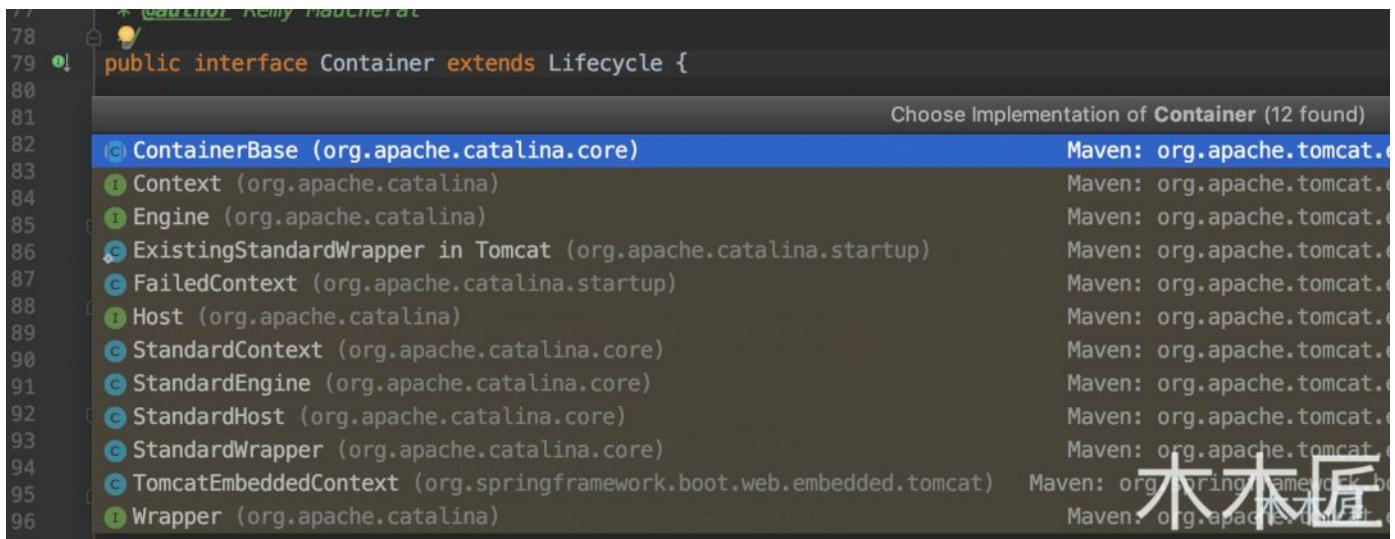
根据上面的代码，我们发现其主要做了两件事情，第一件事就是把Connnctor(我们称之为连接器)对象添加到Tomcat中，第二件事就是configureEngine,这连接器我们勉强能理解（不理解后面会述说），那这个Engine是什么呢？我们查看tomcat.getEngine()的源码：

```

public Engine getEngine() {
 Service service = getServer().findServices()[0];
 if (service.getContainer() != null) {
 return service.getContainer();
 }
 Engine engine = new StandardEngine();
 engine.setName("Tomcat");
 engine.setDefaultHost(hostname);
 engine.setRealm(createDefaultRealm());
 service.setContainer(engine);
 return engine;
}

```

根据上面的源码，我们发现，原来这个Engine是容器，我们继续跟踪源码，找到Container接口



上图中，我们看到了4个子接口，分别是Engine,Host,Context,Wrapper。我们从继承关系上可以知道他们都是容器，那么他们到底有啥区别呢？我看看他们的注释是怎么说的。

```

/**
If used, an Engine is always the top level Container in a Catalina
* hierarchy. Therefore, the implementation's <code>setParent()</code> method
* should throw <code>IllegalArgumentException</code>.
*
* @author Craig R. McClanahan
*/
public interface Engine extends Container {
 //省略代码
}

/**
* <p>
* The parent Container attached to a Host is generally an Engine, but may
* be some other implementation, or may be omitted if it is not necessary.
* </p><p>
* The child containers attached to a Host are generally implementations
* of Context (representing an individual servlet context).
*
* @author Craig R. McClanahan
*/
public interface Host extends Container {
 //省略代码

}

/**</p><p>
* The parent Container attached to a Context is generally a Host, but may
* be some other implementation, or may be omitted if it is not necessary.
* </p><p>
* The child containers attached to a Context are generally implementations
* of Wrapper (representing individual servlet definitions).
* </p><p>
* @author Craig R. McClanahan
*/
public interface Context extends Container, ContextBind {
 //省略代码
}

/**</p><p>
* The parent Container attached to a Wrapper will generally be an
* implementation of Context, representing the servlet context (and
* therefore the web application) within which this servlet executes.
* </p><p>
* Child Containers are not allowed on Wrapper implementations, so the
* <code>addChild()</code> method should throw an
* <code>IllegalArgumentException</code>.
*
* @author Craig R. McClanahan
*/
public interface Wrapper extends Container {

 //省略代码
}

```

上面的注释翻译过来就是，Engine是最高级别的容器，其子容器是Host, Host的子容器是Context, Wrapper是Context的子容器，所以这4个容器的关系就是父子关系，也就是Engine>Host>Context>Wrapper。我们再看看Tomcat类的源码：

```

//部分源码，其余部分省略。
public class Tomcat {
 //设置连接器
 public void setConnector(Connector connector) {
 Service service = getService();
 boolean found = false;

```

```
boolean found = false;
for (Connector serviceConnector : service.findConnectors()) {
 if (connector == serviceConnector) {
 found = true;
 }
}
if (!found) {
 service.addConnector(connector);
}
}

//获取service
public Service getService() {
 return getServer().findServices()[0];
}

//设置Host容器
public void setHost(Host host) {
 Engine engine = getEngine();
 boolean found = false;
 for (Container engineHost : engine.findChildren()) {
 if (engineHost == host) {
 found = true;
 }
 }
 if (!found) {
 engine.addChild(host);
 }
}

//获取Engine容器
public Engine getEngine() {
 Service service = getServer().findServices()[0];
 if (service.getContainer() != null) {
 return service.getContainer();
 }
 Engine engine = new StandardEngine();
 engine.setName("Tomcat");
 engine.setDefaultHost(hostname);
 engine.setRealm(createDefaultRealm());
 service.setContainer(engine);
 return engine;
}

//获取server
public Server getServer() {

 if (server != null) {
 return server;
 }

 System.setProperty("catalina.useNaming", "false");

 server = new StandardServer();

 initBaseDir();

 // Set configuration source
 ConfigFileLoader.setSource(new CatalinaBaseConfigurationSource(new File(basedir), null));

 server.setPort(-1);

 Service service = new StandardService();
 service.setName("Tomcat");
 server.addService(service);
 return server;
}

//添加Context和监听器

```

```

//添加Context容器
public Context addContext(Host host, String contextPath, String contextName,
 String dir) {
 silence(host, contextName);
 Context ctx = createContext(host, contextPath);
 ctx.setName(contextName);
 ctx.setPath(contextPath);
 ctx.setDocBase(dir);
 ctx.addLifecycleListener(new FixContextListener());
}

if (host == null) {
 getHost().addChild(ctx);
} else {
 host.addChild(ctx);
}

//添加Wrapper容器
public static Wrapper addServlet(Context ctx,
 String servletName,
 Servlet servlet) {
 // will do class for name and set init params
 Wrapper sw = new ExistingStandardWrapper(servlet);
 sw.setName(servletName);
 ctx.addChild(sw);

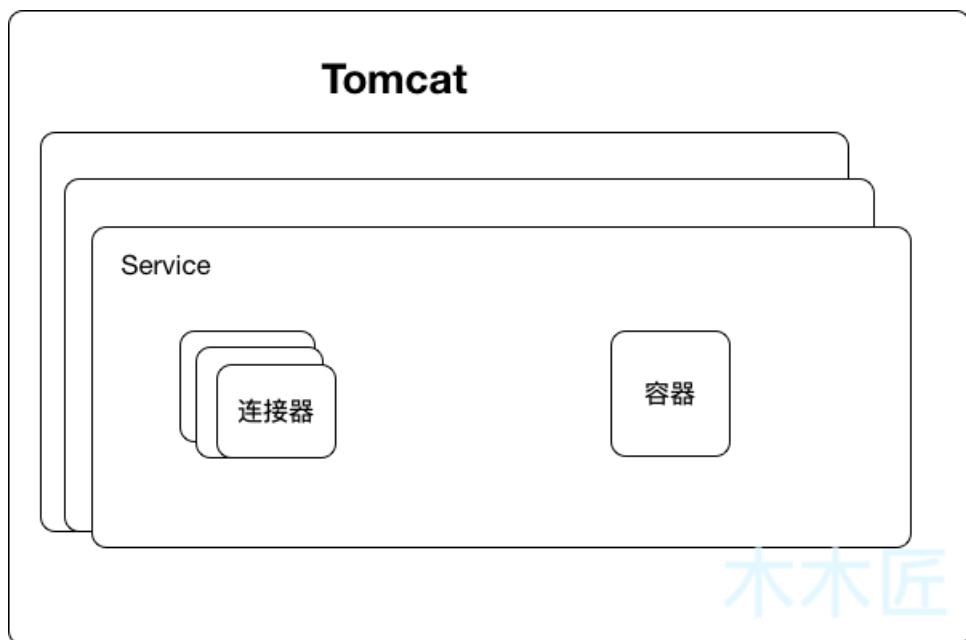
 return sw;
}

}

```

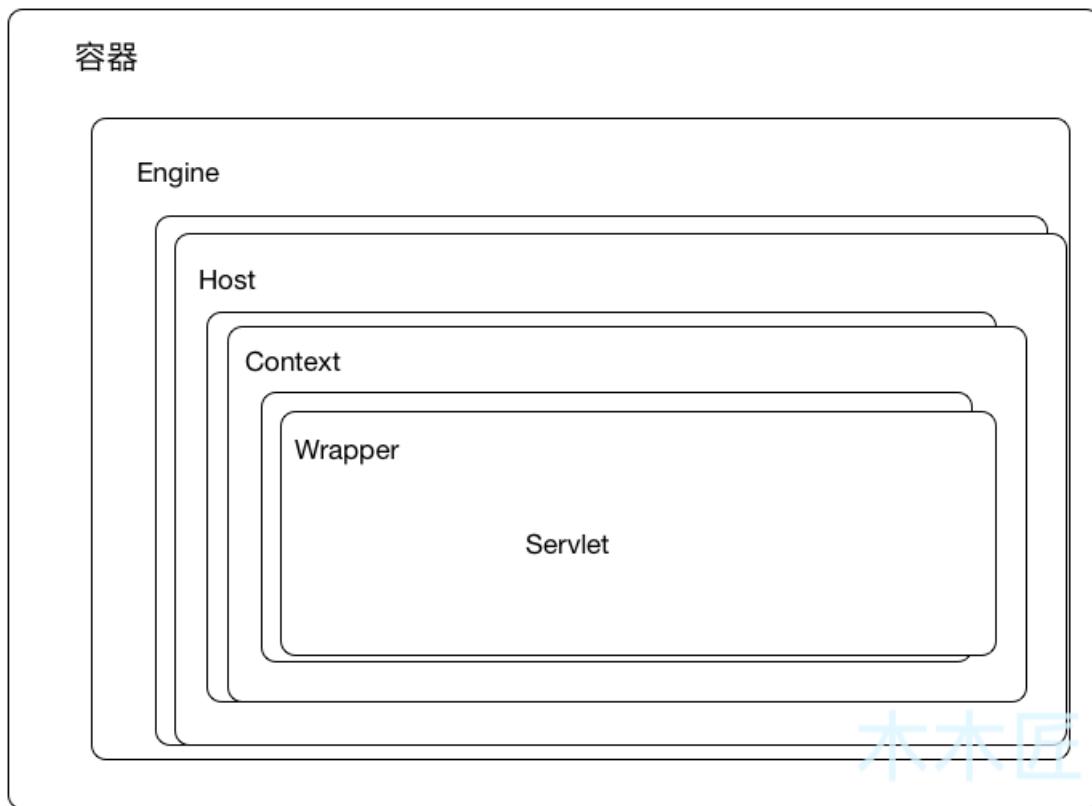
阅读Tomcat的getServer()我们可以知道，Tomcat的最顶层是Server, Server就是Tomcat的实例，一个Tomcat一个Server;通过getEngine()我们可以了解到Server下面是Service，而且是多个，一个Service代表我们部署的一个应用，而且我们还可以知道，Engine容器，一个service只有一个；根据父子关系，我们看setHost()源码可以知道，host容器有多个；同理，我们发现addContext()源码下，Context也是多个；addServlet()表明Wrapper容器也是多个，而且这段代码也暗示了，其实Wrapper和Servlet是一层意思。另外我们根据setConnector源码可以知道，连接器(Connector)是设置在service下的，而且是可以设置多个连接器(Connector)。

根据上面分析，我们可以小结下：Tomcat主要包含了2个核心组件，连接器(Connector)和容器(Container),用图表示如下：



一个Tomcat是一个Server,一个Server下有多个service，也就是我们部署的多个应用，一个应用下有多个连接器(Connector)和

一个容器（Container）,容器下有多个子容器，关系用图表示如下：



Engine下有多个Host子容器，Host下有多个Context子容器，Context下有多个Wrapper子容器。

## 总结

SpringBoot的启动是通过new SpringApplication()实例来启动的，启动过程主要做如下几件事情：> 1. 配置属性 > 2. 获取监听器，发布应用开始启动事件 > 3. 初始化输入参数 > 4. 配置环境，输出banner > 5. 创建上下文 > 6. 预处理上下文 > 7. 刷新上下文 > 8. 再刷新上下文 > 9. 发布应用已经启动事件 > 10. 发布应用启动完成事件

而启动Tomcat就是在第7步中“刷新上下文”；Tomcat的启动主要是初始化2个核心组件，连接器(Connector)和容器(Container)，一个Tomcat实例就是一个Server，一个Server包含多个Service，也就是多个应用程序，每个Service包含多个连接器 (Connetor) 和一个容器 (Container),而容器下又有多个子容器，按照父子关系分别为：Engine,Host,Context,Wrapper，其中除了Engine外，其余的容器都是可以有多个。

## 下期展望

本期文章通过SpringBoot的启动来窥探了Tomcat的内部结构，下一期，我们来分析下本次文章中的连接器(Connetor)和容器(Container)的作用，敬请期待。

- END -

## 推荐阅读

1. 今天,免费送一个 iPhone 11
2. 一场近乎完美基于 Dubbo 的微服务改造实践
3. 为什么年终奖是一个彻头彻尾的职场圈套?
4. 什么是一致性 Hash 算法?
5. 团队开发中 Git 最佳实践



喜欢文章, 点个在看 

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 【Spring Boot】一个依赖搞定 Session 共享

Java后端 2月28日



微信搜一搜

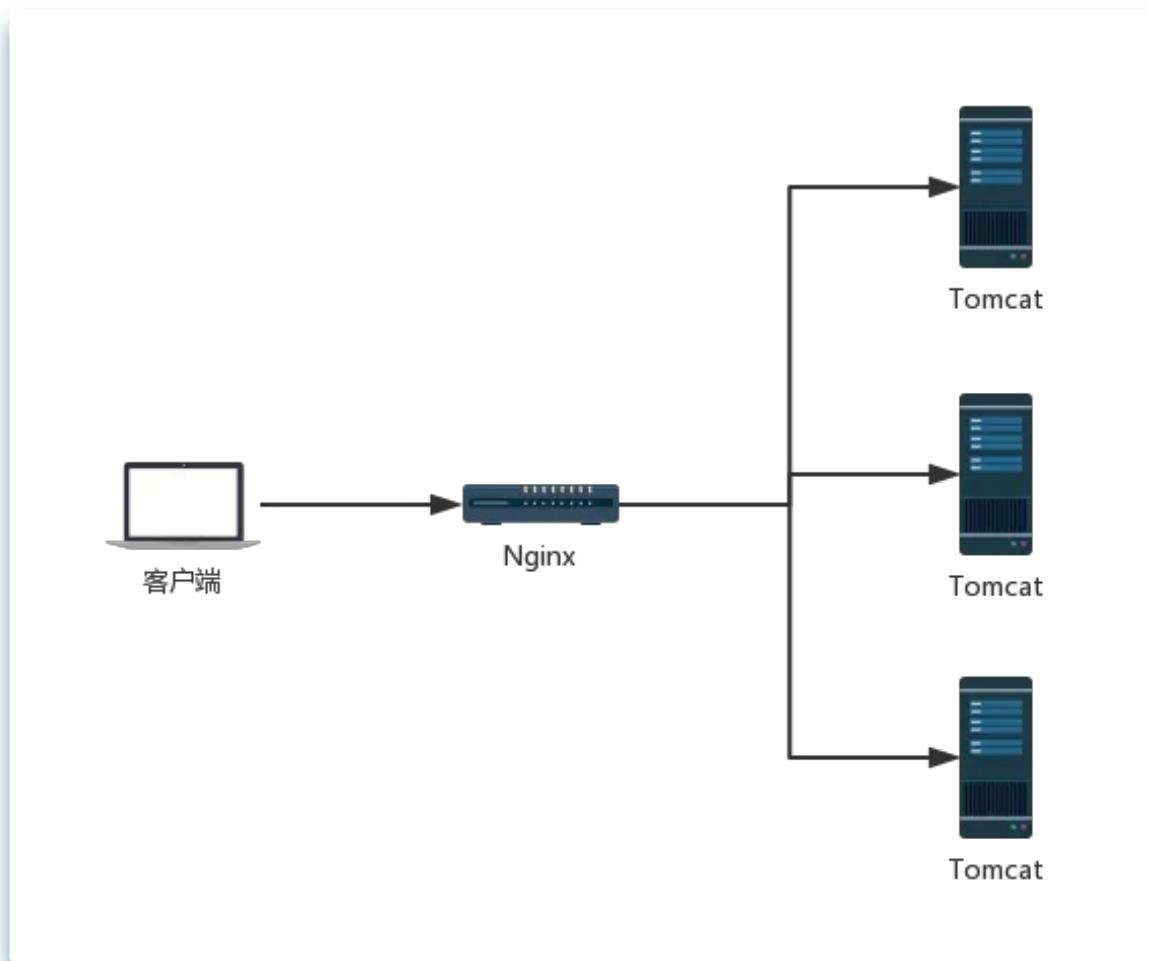
Java后端

来源：公众号【牧码小子】

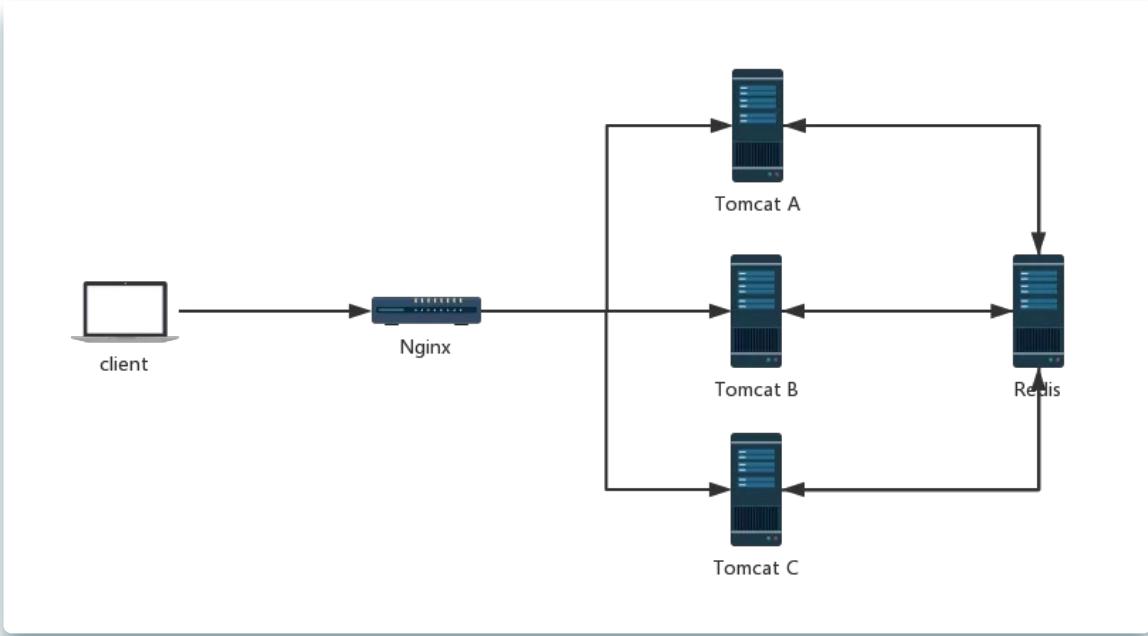
作者：江南一点雨

有的人可能会觉得题目有点夸张，其实不夸张，题目没有使用任何修辞手法！认真读完本文，你就知道松哥说的是对的了！

在传统的单服务架构中，一般来说，只有一个服务器，那么不存在 Session 共享问题，但是在分布式/集群项目中，Session 共享则是一个必须面对的问题，先看一个简单的架构图：



在这样的架构中，会出现一些单服务中不存在的问题，例如客户端发起一个请求，这个请求到达 Nginx 上之后，被 Nginx 转发到 Tomcat A 上，然后在 Tomcat A 上往 session 中保存了一份数据，下次又来一个请求，这个请求被转发到 Tomcat B 上，此时再去 Session 中获取数据，发现没有之前的数据。对于这一类问题的解决，思路很简单，就是将各个服务之间需要共享的数据，保存到一个公共的地方（主流方案就是 Redis）：



当所有 Tomcat 需要往 Session 中写数据时，都往 Redis 中写，当所有 Tomcat 需要读数据时，都从 Redis 中读。这样，不同的服务就可以使用相同的 Session 数据了。

这样的方案，可以由开发者手动实现，即手动往 Redis 中存储数据，手动从 Redis 中读取数据，相当于使用一些 Redis 客户端工具来实现这样的功能，毫无疑问，手动实现工作量还是蛮大的。

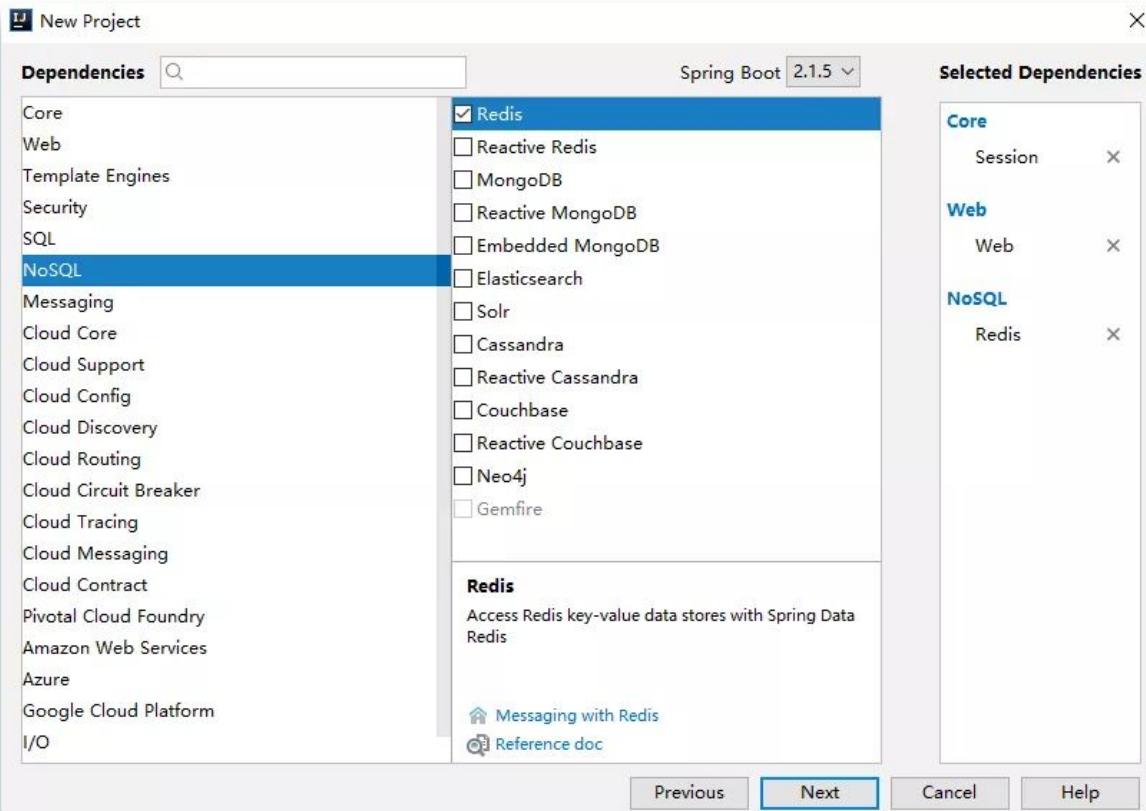
一个简化的方案就是使用 Spring Session 来实现这一功能，Spring Session 就是使用 Spring 中的代理过滤器，将所有的 Session 操作拦截下来，自动的将数据同步到 Redis 中，或者自动的从 Redis 中读取数据。

对于开发者来说，所有关于 Session 同步的操作都是透明的，开发者使用 Spring Session，一旦配置完成后，具体的用法就像使用一个普通的 Session 一样。

## 1 实战

### 1.1 创建工程

首先 创建一个 Spring Boot 工程，引入 Web、Spring Session 以及 Redis：



创建成功之后，pom.xml 文件如下：

```
<dependencies>
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-redis</artifactId>
 </dependency>
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
 <dependency>
 <groupId>org.springframework.session</groupId>
 <artifactId>spring-session-data-redis</artifactId>
 </dependency>
</dependencies>
```

### 注意：

这里我使用的 Spring Boot 版本是 2.1.4，如果使用当前最新版 Spring Boot 2.1.5 的话，除了上面这些依赖之外，需要额外添加 Spring Security 依赖（其他操作不受影响，仅仅只是多了一个依赖，当然也多了 Spring Security 的一些默认认证流程）。

## 1.2 配置 Redis

```
spring.redis.host=192.168.66.128
```

```
spring.redis.port=6379
```

```
spring.redis.password=123
```

```
spring.redis.database=0
```

这里的 Redis , 我虽然配置了四行 , 但是考虑到端口默认就是 6379 , database 默认就是 0 , 所以真正要配置的 , 其实就是两行。

### 1.3 使用

配置完成后 , 就可以使用 Spring Session 了 , 其实就是使用普通的 HttpSession , 其他的 Session 同步到 Redis 等操作 , 框架已经自动帮你完成了 :

```
@RestController

public class HelloController {

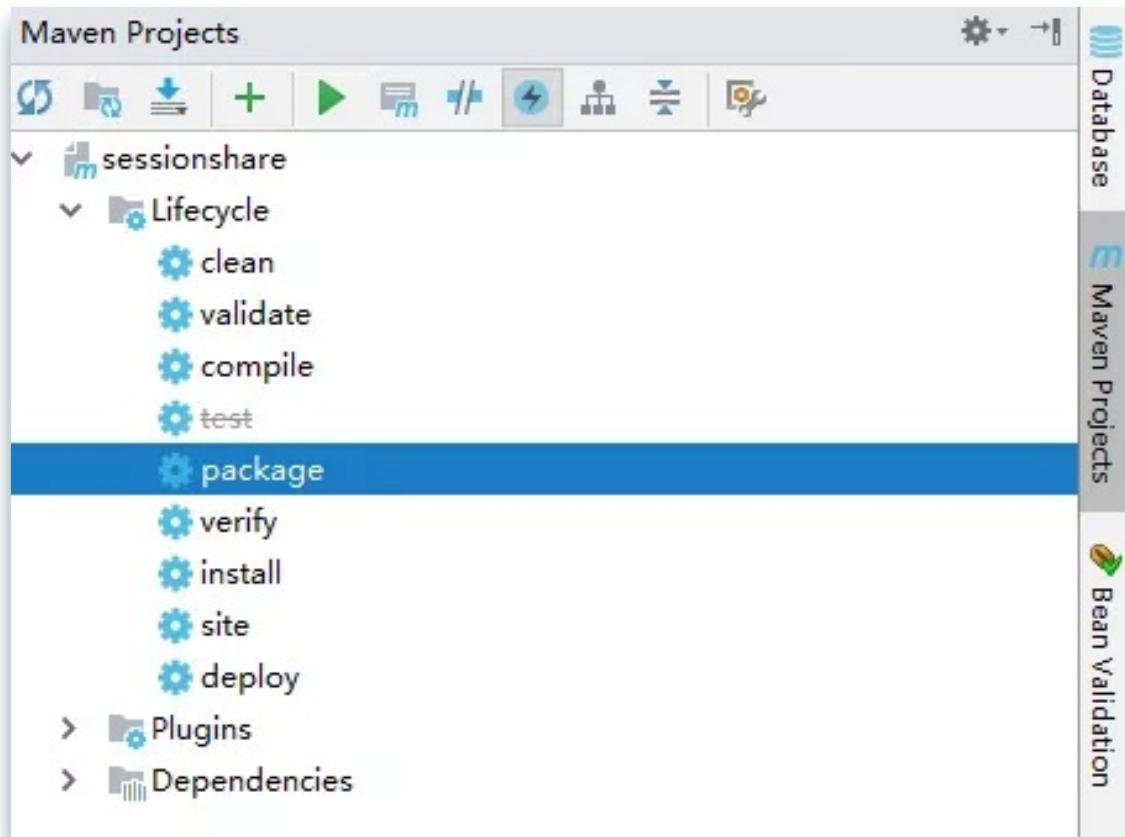
 @Value("${server.port}")
 Integer port;

 @GetMapping("/set")
 public String set(HttpSession session) {
 session.setAttribute("user", "javaboy");
 return String.valueOf(port);
 }

 @GetMapping("/get")
 public String get(HttpSession session) {
 return session.getAttribute("user") + ":" + port;
 }
}
```

考虑到一会 Spring Boot 将以集群的方式启动 , 为了获取每一个请求到底是哪一个 Spring Boot 提供的服务 , 需要在每次请求时返回当前服务的端口号 , 因此这里我注入了 server.port 。

接下来 , 项目打包 :



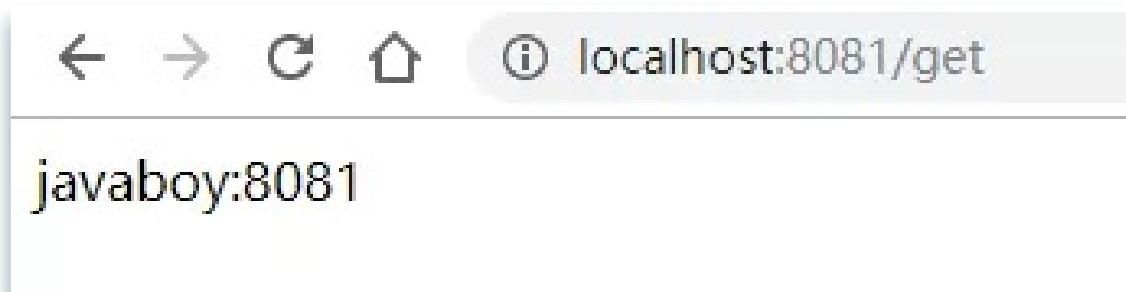
打包之后，启动项目的两个实例：

```
java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8080
java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8081
```

然后先访问 `localhost:8080/set` 向 8080 这个服务的Session 中保存一个变量，访问完成后，数据就已经自动同步到 Redis 中了：

```
127.0.0.1:6379> keys *
1) "spring:session:sessions:bla8fbad-8912-4d01-acf-acf90765bab4"
2) "spring:session:sessions:expires:bla8fbad-8912-4d01-acf-acf90765bab4"
3) "spring:session:expirations:1559551560000"
```

然后，再调用 `localhost:8081/get` 接口，就可以获取到 8080 服务的session 中的数据：



此时关于 session 共享的配置就已经全部完成了，session 共享的效果我们已经看到了，但是每次访问都是我自己手动切换服务实例，因此，接下来我们来引入 Nginx，实现服务实例自动切换。

## 1.4 引入 Nginx

很简单，进入 Nginx 的安装目录的 `conf` 目录下（默认是在 `/usr/local/nginx/conf`），编辑 `nginx.conf` 文件：

```
upstream javaboy.org{
 server 127.0.0.1:8080 weight=1;
 server 127.0.0.1:8081 weight=2;
}
server {
 listen 80;
 server_name localhost;

 #charset koi8-r;

 #access_log logs/host.access.log main;

 location / {
 proxy_pass http://javaboy.org;
 proxy_redirect default;
 #root html;
 #index index.html index.htm;
 }
}
```

在这段配置中：

1. `upstream` 表示配置上游服务器
2. `javaboy.org` 表示服务器集群的名字，这个可以随意取名字
3. `upstream` 里边配置的是一个个的单独服务
4. `weight` 表示服务的权重，意味者将有多少比例的请求从 `Nginx` 上转发到该服务上
5. `location` 中的 `proxy_pass` 表示请求转发的地址，`/` 表示拦截到所有的请求，转发转发到刚刚配置好的服务集群中
6. `proxy_redirect` 表示设置当发生重定向请求时，`nginx` 自动修正响应头数据（默认是 `Tomcat` 返回重定向，此时重定向的地址是 `Tomcat` 的地址，我们需要将之修改使之成为 `Nginx` 的地址）。

配置完成后，将本地的 `Spring Boot` 打包好的 `jar` 上传到 `Linux`，然后在 `Linux` 上分别启动两个 `Spring Boot` 实例：

```
nohup java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8080 &

nohup java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8081 &
```

其中

- `nohup` 表示当终端关闭时，`Spring Boot` 不要停止运行
- `&` 表示让 `Spring Boot` 在后台启动

配置完成后，重启 `Nginx`：

```
/usr/local/nginx/sbin/nginx -s reload
```

Nginx 启动成功后，我们首先手动清除 Redis 上的数据，然后访问 192.168.66.128/set 表示向 session 中保存数据，这个请求首先会到达 Nginx 上，再由 Nginx 转发给某一个 SpringBoot 实例：



如上，表示端口为 8081 的 SpringBoot 处理了这个 /set 请求，再访问 /get 请求：



可以看到，/get 请求是被端口为 8080 的服务所处理的。

## 2 总结

本文主要向大家介绍了 Spring Session 的使用，另外也涉及到一些 Nginx 的使用，虽然本文较长，但是实际上 Spring Session 的配置没啥。

我们写了一些代码，也做了一些配置，但是全都和 Spring Session 无关，配置是配置 Redis，代码就是普通的 HttpSession，和 Spring Session 没有任何关系！

唯一和 Spring Session 相关的，可能就是我在一开始引入了 Spring Session 的依赖吧！

如果大家没有在 SSM 架构中用过 Spring Session，可能不太好理解我们在 Spring Boot 中使用 Spring Session 有多么方便，因为在 SSM 架构中，Spring Session 的使用要配置三个地方，一个是 web.xml 配置代理过滤器，然后在 Spring 容器中配置 Redis，最后再配置 Spring Session，步骤还是有些繁琐的，而 Spring Boot 中直接帮我们省去了这些繁琐的步骤！不用再去配置 Spring Session。

好了，本文就说到这里，有问题欢迎留言讨论，本文相关案例我已经上传到 GitHub，大家可以自行下载：<https://github.com/lenve/javaboy-code-samples>

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓ 扫描二维码添加小编 ↓



### 推荐阅读

1. 浅谈 Web 网站架构演变过程
2. 一个非常实用的特性，很多人没用过
3. Spring Boot 2.x 操作缓存的新姿势
4. MyBatis 中 SQL 写法技巧小总结



微信搜一搜  
Q Java后端

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 【实战】Spring Boot 2.x 操作缓存的新姿势

涅槃重生 Java后端 2月27日



## 一、介绍

Spring Cache 是spring3版本之后引入的一项技术,可以简化对于缓存层的操作,spring cache与springcloud stream类似,都是基于抽象层,可以任意切换其实现。其核心是CacheManager、Cache这两个接口,所有由spring整合的cache都要实现这两个接口、Redis的实现类则是RedisCache 和 RedisManager。

往期关于Spring Boot实战文章可以关注微信公众号: Java后端, 后台回复 技术博文 获取。

## 二、使用

### I、查询

需要导入的依赖

```
1 <dependency>
2 <groupId>org.springframework.boot</groupId>
3 <artifactId>spring-boot-starter-cache</artifactId>
4 </dependency>
5 <dependency>
6 <groupId>org.springframework.boot</groupId>
7 <artifactId>spring-boot-starter-data-redis</artifactId>
8 </dependency>
```

编写对于cache的配置

```

1 @EnableCaching
2 @SpringBootConfiguration
3 public class CacheConfig {
4
5 @Autowired
6 private RedisConnectionFactory connectionFactory;
7
8 @Bean //如果有多个CacheManager的话需要使用@Primary直接指定那个是默认的
9 public RedisCacheManager cacheManager() {
10 RedisSerializer<String> redisSerializer = new StringRedisSerializer();
11 Jackson2JsonRedisSerializer<Object> jackson2JsonRedisSerializer = new Jackson2JsonRedisSerializer<>(Object.class);
12
13 ObjectMapper om = new ObjectMapper();
14 //防止在序列化的过程中丢失对象的属性
15 om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
16 //开启实体类和json的类型转换
17 om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
18 jackson2JsonRedisSerializer.setObjectMapper(om);
19
20 //配置序列化 (解决乱码的问题)
21 RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig().serializeKeysWith(RedisSerializationContext.Ser
22 //不缓存空值
23 .disableCachingNullValues()
24 //1分钟过期
25 .entryTtl(Duration.ofMinutes(1))
26 ;
27 RedisCacheManager cacheManager = RedisCacheManager.builder(connectionFactory)
28 .cacheDefaults(config)
29 .build();
30 return cacheManager;
31 }
32}

```

进行以上配置即可使用springboot cache了，还有一个key的生成策略的配置（可选）

```

1 @Bean
2 public KeyGenerator keyGenerator() {
3 return (target, method, params) -> {
4 StringBuffer key = new StringBuffer();
5 key.append(target.getClass().getSimpleName() + "#" + method.getName() + "(");
6 for (Object args : params) {
7 key.append(args + ",");
8 }
9 key.deleteCharAt(key.length() - 1);
10 key.append(")");
11 return key.toString();
12 };
13}

```

注意：如果配置了KeyGenerator

，在进行缓存的时候如果不指定key的话，最后会把生成的key缓存起来，如果同时配置

了KeyGenerator 和key则优先使用key。

在controller或者service的类上面添加 @CacheConfig，注解里面的参数详情见下表：

参数名	参数值	作用
cacheNames	可以随意填写，一般是一个模块或者一个很重要的功能名称	无具体作用，只是用来区分缓存，方便管理
keyGenerator	就是自己配置的KeyGenerator的名称	全局key都会以他的策略去生成
cacheManager	自己配置的CacheManager	用来操作Cache对象的，很多对于缓存的配置也由他去管理

在标有@CacheConfig的类里面编写一个查询单个对象的方法并添加 @Cacheable注解

```

1 @Cacheable(key = "#id", unless = "#result == null")
2 @PatchMapping("/course/{id}")
3 public Course courseInfo(@PathVariable Integer id) {
4 log.info("进来了.. ");
5 return courseService.getCourseInfo(id);
6 }
```

执行完该方法后，执行结果将会被缓存到Redis：



@Cacheable注解中参数详情见下表：

参数名	作用
cacheNames	被缓存的时候的命名空间
key	这里的key的优先级是最高的，可以覆盖掉全局配置的key，如果不配置的话使用的就是全局的key
keyGenerator	指定的缓存的key的生成器，默认没有
cacheManager	指定要使用哪个缓存管理器。默认是底层自动配置的管理器
condition	满足什么条件会进行缓存，里面可以写简单的表达式进行逻辑判断
unless	满足什么条件不进行缓存，里面可以写简单的表达式进行逻辑判断
sync	加入缓存的这个操作是否是同步的

## II、修改

编写一个修改的方法，参数传对象，返回值也改成这个对象

```
1 @PutMapping("/course")
2 public Course modifyCourse(@RequestBody Course course) {
3 courseService.updateCourse(course);
4 return course;
5 }
```

在方法上面添加 `@CachePut(key = "#course.id")` 注解，这个注解表示将方法的返回值更新到缓存中，注解中的参数和 `@Cacheable` 中的一样，这里就略过了。

## III、删除

编写删除方法，在方法上添加`@CacheEvict`注解

```
1 @CacheEvict(key = "#id")
2 @DeleteMapping("/course/{id}")
3 public void removeCourse(@PathVariable Integer id) {
4 courseService.remove(id);
5 }
```

`@CacheEvict` 的参数信息见下表：

参数名	描述
allEntries	是否删除该命名空间下面的全部缓存，默认是false
beforeInvocation	在执行删除方法前就执行清空缓存操作，默认是false，如果删除方法执行报错该注解则不执行

## 三、基于代码的Cache的使用

因为我们有配置的CacheManager，所以可以利用RedisCacheManager对象去手动操作cache，首先将CacheManager注入进来：

```
1 @Resource
2 private CacheManager cacheManager;
3
4 @PatchMapping("/course2/{id}")
5 public Course course2(@PathVariable Integer id) {
6 // 获取指定命名空间的cache
7 Cache cache = cacheManager.getCache("course");
8 // 通过key获取对应的value
9 Cache.ValueWrapper wrapper = cache.get(2);
10 if (wrapper == null) {
11 // 查询数据库
12 Course course = courseService.getCourseInfo(id);
13 // 加入缓存
14 cache.put(course.getId(), course);
15 return course;
16 } else {
17 // 将缓存的结果返回
18 // 因为配置了enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
19 // 所以在进行强转的时候不会报错
20 return (Course) wrapper.get();
21 }
22}
```

如果还看不明白，请去码云拉取源码 <https://gitee.com/tianmaolin/Spring-Boot-Cache.git>

作者 | 涅槃重生

链接 | [www.cnblogs.com/maolinjava/p/12335861.html](http://www.cnblogs.com/maolinjava/p/12335861.html)

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓ 扫描二维码添加小编 ↓



## 推荐阅读

1. 狠！删库跑路！
2. 分布式与集群的区别究竟是什么？
3. 看完这篇 HTTP，面试官就难不倒你了
4. 快速建站利器！



微信搜一搜

Q Java后端

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 一张图帮你记忆：Spring Boot 应用在启动阶段执行代码的几种方式

Java后端 2月24日

以下文章来源于日拱一兵，作者tan日拱一兵



## 日拱一兵

像读侦探小说一样趣读Java技术

作者 | 日拱一兵

来源 | 日拱一兵

### 前言

有时候我们需要在应用启动时执行一些代码片段，这些片段可能是仅仅是为了记录 log，也可能是在启动时检查与安装证书，诸如上述业务要求我们可能会经常碰到

Spring Boot 提供了至少 5 种方式用于在应用启动时执行代码。我们应该如何选择？本文将会逐步解释与分析这几种不同方式。

### CommandLineRunner

CommandLineRunner 是一个接口，通过实现它，我们可以在 Spring 应用成功启动之后 执行一些代码片段

```
@Slf4j
@Component
@Order(2)
public class MyCommandLineRunner implements CommandLineRunner {

 @Override
 public void run(String... args) throws Exception {
 log.info("MyCommandLineRunner order is 2");
 if (args.length > 0){
 for (int i = 0; i < args.length; i++) {
 log.info("MyCommandLineRunner current parameter is: {}", args[i]);
 }
 }
 }
}
```

当 Spring Boot 在应用上下文中找到CommandLineRunner bean，它将会在应用成功启动之后调用 run() 方法，并传递用于启动应用程序的命令行参数

通过如下 maven 命令生成 jar 包：

```
mvn clean package
```

通过终端命令启动应用，并传递参数：

```
java -jar springboot-application-startup-0.0.1-SNAPSHOT.jar --foo=bar --name=rgyb
```

查看运行结果：

```
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
main] .SpringbootApplicationStartupApplication : Started SpringbootApplicationStartupApplication in 2.11 seconds (JVM running for 2.551)
main] t.d.s.activity.MyCommandLineRunner
main] t.d.s.activity.MyCommandLineRunner
main] t.d.s.activity.MyCommandLineRunner
```

到这里我们可以看出几个问题：

1. 命令行传入的参数并没有被解析，而只是显示出我们传入的字符串内容 `--foo=bar, --name=rgyb`，我们可以通过 `CommandLineRunner` 解析，我们稍后看
2. 在重写的 `run()` 方法上有 `throws Exception` 标记，`Spring Boot` 会将 `CommandLineRunner` 作为应用启动的一部分，如果运行 `run()` 方法时抛出 `Exception`，应用将会终止启动
3. 我们在类上添加了 `@Order(2)` 注解，当有多个 `CommandLineRunner` 时，将会按照 `@Order` 注解中的数字从小到大排序（数字当然也可以用复数）

#### ⚠不要使用 `@Order` 太多

看到 `order` 这个 "黑科技" 我们会觉得它可以非常方便将启动逻辑按照指定顺序执行，但如果你这么写，说明多个代码片段是有相互依赖关系的，为了让我们的代码更好维护，我们应该减少这种依赖使用

### 小结

如果我们只是想简单的获取以空格分隔的命令行参数，那 `CommandLineRunner` 就足够使用了，对了如果想要更多 `Spring Boot` 相关的技术博文可以本公众号「Java后端」回复「技术博文」获取。

### ApplicationRunner

上面提到，通过命令行启动并传递参数，`CommandLineRunner` 不能解析参数，如果要解析参数，那我们就要用到 `ApplicationRunner` 参数了

```
@Component
@Slf4j
@Order(1)
public class MyApplicationRunner implements ApplicationRunner {

 @Override
 public void run(ApplicationArguments args) throws Exception {
 log.info("MyApplicationRunner order is 1");
 log.info("MyApplicationRunner Current parameter is {}", args.getOptionValues("foo"));
 }
}
```

重新打 jar 包，运行如下命令：

```
java -jar springboot-application-startup-0.0.1-SNAPSHOT.jar --foo=bar,rgyb
```

运行结果如下：

```
: Tomcat started on port(s): 8080 (http) with context path ''
Started SpringbootApplicationStartupApplication in 1.752 seconds (JVM running for 2.148)
MyApplicationRunner order is 1
MyApplicationRunner Current parameter is [bar,rgyb]
MyCommandLineRunner order is 2
MyCommandLineRunner current parameter is: --foo=bar,rgyb
```

到这里我们可以看出：

1. 同 MyCommandLineRunner 相似，但 ApplicationRunner 可以通过 run 方法的 ApplicationArguments 对象解析出命令行参数，并且每个参数可以有多个值在里面，因为 getOptionValues 方法返回 List 数组
2. 在重写的 run() 方法上有 throws Exception 标记，Spring Boot 会将 CommandLineRunner 作为应用启动的一部分，如果运行 run() 方法时抛出 Exception，应用将会终止启动
3. ApplicationRunner 也可以使用 @Order 注解进行排序，从启动结果来看，它与 CommandLineRunner 共享 order 的顺序，稍后我们通过源码来验证这个结论

## 小结

如果我们想获取复杂的命令行参数时，我们可以使用 ApplicationRunner

## ApplicationListener

如果我们不需要获取命令行参数时，我们可以将启动逻辑绑定到 Spring 的 ApplicationReadyEvent 上

```
@Slf4j
@Component
@Order(0)
public class MyApplicationListener implements ApplicationListener<ApplicationReadyEvent> {

 @Override
 public void onApplicationEvent(ApplicationReadyEvent applicationReadyEvent) {
 log.info("MyApplicationListener is started up");
 }
}
```

运行程序查看结果：

```
Tomcat started on port(s): 8080 (http) with context path ''
Started SpringbootApplicationStartupApplication in 1.351 seconds (JVM running for 1.811)
MyApplicationRunner order is 1
MyApplicationRunner Current parameter is null:
MyCommandLineRunner order is 2
MyApplicationListener is started up
```

到这我们可以看出：

1. ApplicationReadyEvent **当且仅当** 在应用程序就绪之后才被触发，甚至是说上面的 Listener 要在本文说的所有解决方案都执行了之后才会被触发，最终结论请稍后看
2. 代码中我用 Order(0) 来标记，显然 ApplicationListener 也是可以用该注解进行排序的，按数字大小排序，应该是最先执行。

但是,这个顺序仅用于同类型的

ApplicationListener

之间的排序,与前面提到的 ApplicationRunners 和

CommandLineRunners 的排序并不共享

## 小结

如果我们不需要获取命令行参数,我们可以通过 ApplicationListener<ApplicationReadyEvent> 创建一些全局的启动逻辑,我们还可以通过它获取 Spring Boot 支持的 configuration properties 环境变量参数

如果你看过我之前写的 **Spring Bean 生命周期三部曲**:

- Spring Bean 生命周期之缘起
- Spring Bean 生命周期之缘尽
- Spring Aware 到底是什么?

那么你会对下面两种方式非常熟悉了

### @PostConstruct

创建启动逻辑的另一种简单解决方案是提供一种在 bean 创建期间由 Spring 调用的初始化方法。我们要做的就只是将 @PostConstruct 注解添加到方法中:

```
@Component
@Slf4j
@DependsOn("myApplicationListener")
public class MyPostConstructBean {

 @PostConstruct
 public void testPostConstruct(){
 log.info("MyPostConstructBean");
 }
}
```

查看运行结果:

```
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 1802 ms
MyPostConstructBean
Initializing ExecutorService 'applicationTaskExecutor'
Tomcat started on port(s): 8080 (http) with context path ''
Started SpringbootApplicationStartupApplication in 3.443 seconds (JVM running for 4.745)
MyApplicationRunner order is 1
MyApplicationRunner Current parameter is null:
MyCommandLineRunner order is 2
MyApplicationListener is started up
```

从上面运行结果可以看出:

1. Spring 创建完 bean 之后 (在启动之前),便会立即调用 @PostConstruct 注解标记的方法,因此我们无法使用 @Order 注解对其进行自由排序,因为它可能依赖于 @Autowired 插入到我们 bean 中的其他 Spring bean。

2. 相反,它将在依赖于它的所有 bean 被初始化之后被调用,如果要添加人为的依赖关系并由此创建一个排序,则可以使用 @DependsOn 注解(虽然可以排序,但是不建议使用,理由和 @Order 一样)

## 小结

@PostConstruct 方法固有地绑定到现有的 Spring bean,因此应仅将其用于此单个 bean 的初始化逻辑;

## InitializingBean

与 @PostConstruct 解决方案非常相似,我们可以实现 InitializingBean 接口,并让 Spring 调用某个初始化方法:

```
@Component
@Slf4j
public class MyInitializingBean implements InitializingBean {

 @Override
 public void afterPropertiesSet() throws Exception {
 log.info("MyInitializingBean.afterPropertiesSet()");
 }
}
```

查看运行结果:

```
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 763 ms
MyInitializingBean.afterPropertiesSet()
MyPostConstructBean
Initializing ExecutorService 'applicationTaskExecutor'
Tomcat started on port(s): 8080 (http) with context path ''
Started SpringbootApplicationStartupApplication in 1.926 seconds (JVM running for 3.235)
MyApplicationRunner order is 1
MyApplicationRunner Current parameter is null:
MyCommandLineRunner order is 2
MyApplicationListener is started up
```

从上面的运行结果中,我们得到了和 @PostConstruct 一样的效果,但二者还是有差别的

### △ @PostConstruct 和 afterPropertiesSet 区别

1. afterPropertiesSet, 顾名思义「在属性设置之后」,调用该方法时,该 bean 的所有属性已经被 Spring 填充。

如果我们在某些属性上使用 @Autowired(常规操作应该使用构造函数注入),那么 Spring 将在调用 afterPropertiesSet 之前将 bean 注入这些属性。

但 @PostConstruct 并没有这些属性填充限制

2. 所以 InitializingBean.afterPropertiesSet 解决方案比使用 @PostConstruct 更安全,因为如果我们依赖尚未自动注入的

## 小结

如果我们使用构造函数注入，则这两种解决方案都是等效的

## 源码分析

请打开你的 IDE (重点代码已标记注释):

MyCommandLineRunner 和 ApplicationRunner 是在何时被调用的呢?

打开 SpringApplication.java 类，里面有 callRunners 方法

```
private void callRunners(ApplicationContext context, ApplicationArguments args) {
 List<Object> runners = new ArrayList<>();

 runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());

 runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());

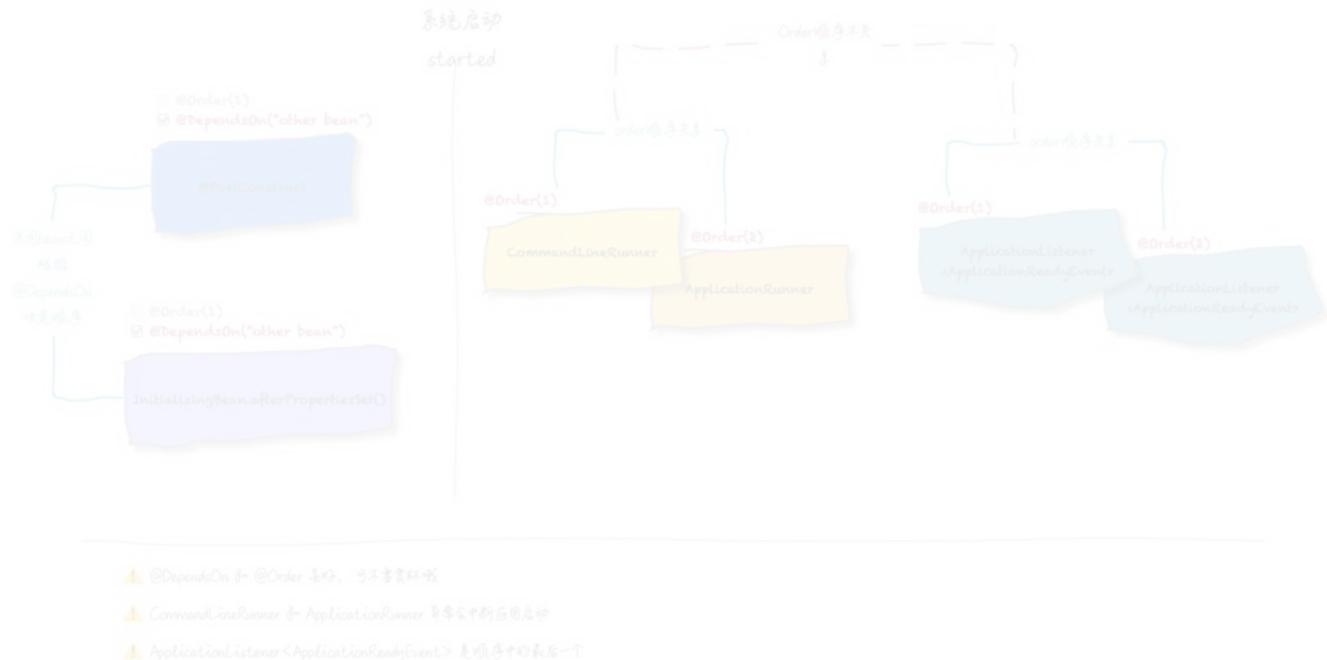
 AnnotationAwareOrderComparator.sort(runners);

 for (Object runner : new LinkedHashSet<>(runners)) {
 if (runner instanceof ApplicationRunner) {
 callRunner((ApplicationRunner) runner, args);
 }
 if (runner instanceof CommandLineRunner) {
 callRunner((CommandLineRunner) runner, args);
 }
 }
}
```

强烈建议完整看一下 SpringApplication.java 的全部代码，Spring Boot 启动过程及原理都可以从这个类中找到一些答案

## 总结

最后画一张图用来总结这几种方式（高清大图请查看原文：[dayarch.top/p/spring-boot-autowiring/](http://dayarch.top/p/spring-boot-autowiring/)）



- E N D -

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓ 扫描二维码添加小编 ↓



## 推荐阅读

1. 10 分钟实现 Java 发送邮件功能
2. Spring Boot 线程池的创建
3. Spring Boot 整合 Redis
4. 2020 年 9 大顶级 Java 框架



微信搜一搜

Q Java后端

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 为什么很多 Spring Boot 开发者放弃了 Tomcat ?

阿迈达聊技术 Java后端 2月20日



微信搜一搜

Java后端

作者 | 阿迈达聊技术

链接 | [toutiao.com/a6775476659416990212](https://toutiao.com/a6775476659416990212)

在SpringBoot框架中，我们使用最多的是Tomcat，这是SpringBoot默认的容器技术，而且是内嵌式的Tomcat。同时，**SpringBoot也支持Undertow容器，我们可以很方便的用Undertow替换Tomcat，而Undertow的性能和内存使用方面都优于Tomcat，那我们如何使用Undertow技术呢？本文将为大家细细讲解。**

SpringBoot可以说是目前最火的Java Web框架了。它将开发者从繁重的xml解救了出来，让开发者在几分钟内就可以创建一个完整的Web服务，极大的提高了开发者的工作效率。Web容器技术是Web项目必不可少的组成部分，因为任何Web项目都要借助容器技术来运行起来。在SpringBoot框架中，我们使用最多的是Tomcat，这是SpringBoot默认的容器技术，而且是内嵌式的Tomcat。



对于Tomcat技术，Java程序员应该都非常熟悉，它是Web应用最常用的容器技术。我们最早的开发的项目基本都是部署在Tomcat下运行，那除了Tomcat容器，SpringBoot中我们还可以使用什么容器技术呢？没错，就是题目中的**Undertow容器技术**。SpringBoot已经完全继承了Undertow技术，我们只需要引入Undertow的依赖即可，如下图所示。

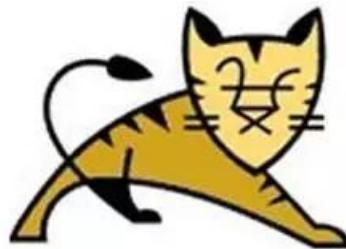
```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 <exclusions>
 <exclusion>
 <groupId>org.springframework</groupId>
 <artifactId>spring-webmvc</artifactId>
 </exclusion>
 <exclusion>
 <groupId>org.springframework</groupId>
 <artifactId>spring-web</artifactId>
 </exclusion>
 <exclusion>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter</artifactId>
 </exclusion>
 <exclusion>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-tomcat</artifactId>
 </exclusion>
 </exclusions>
</dependency>
```

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

配置好以后，我们启动应用程序，发现容器已经替换为Undertow。

### 那我们为什么需要替换Tomcat为Undertow技术呢？

Tomcat是Apache基金下的一个轻量级的Servlet容器，支持Servlet和JSP。Tomcat具有Web服务器特有的功能，包括Tomcat管理和控制平台、安全局管理和Tomcat阀等。**Tomcat本身包含了HTTP服务器，因此也可以视作单独的Web服务器**。但是，Tomcat和ApacheHTTP服务器不是一个东西，ApacheHTTP服务器是用C语言实现的HTTP Web服务器。Tomcat是完全免费的，深受开发者的喜爱。



Undertow是Red Hat公司的开源产品，它完全采用Java语言开发，是一款灵活的高性能Web服务器，支持阻塞IO和非阻塞IO。由于Undertow采用Java语言开发，可以直接嵌入到Java项目中使用。同时，Undertow完全支持Servlet和Web Socket，在高并发情况下表现非常出色。

# WildFly

undertow-servlet

undertow-core

undertow-websockets

## XNIO

我们在相同机器配置下压测Tomcat和Undertow，得到的测试结果如下所示：

- QPS测试结果对比：

### Tomcat

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Startup	3000	7	1	549	35.78374361	0	293.8583603	55.95935572	55.67238466	195
Others	3000	1	0	45	1.359661682	0	287.8802418	54.82094449	54.53981144	195
Others	3000	1	0	24	1.155032275	0	292.1129503	55.62697785	55.3417113	195

### Undertow

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Startup	3000	6	0	451	31.6188702	0	295.6830278	63.81440346	56.01807363	221
Others	3000	1	0	22	1.255447862	0	292.7400468	63.17924839	55.46051669	221
Others	3000	1	0	18	1.559477975	0	294.3773918	63.53262069	55.77071681	221

- 内存使用对比：

### Tomcat

Heap Size: 697,827,328 B

Used: 124,260,976 B

Max: 2,147,483,648 B

Threads: 17 Live, 22 Started

### Undertow

Heap Size: 630,718,464 B

Used: 114,599,536 B

Max: 2,147,483,648 B

Threads: 17 Live, 20 Started

通过测试发现，在高并发系统中，Tomcat相对来说比较弱。在相同的机器配置下，模拟相等的请求数，Undertow在性能和内存使用方面都是最优的。并且Undertow新版本默认使用持久连接，这将会进一步提高它的并发吞吐能力。所以，如果是高并发的业务系统，Undertow是最佳选择。

SpringBoot中我们既可以使用Tomcat作为Http服务，也可以用Undertow来代替。Undertow在高并发业务场景中，性能优于Tomcat。所以，如果我们的系统是高并发请求，不妨使用一下Undertow，你会发现你的系统性能会得到很大的提升。

- E N D -

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓ 扫描二维码添加小编 ↓



#### 推荐阅读

1. Chrome浏览器必知必会的小技巧
2. 100 多个免费 API 接口分享
3. 面试题：Class.forName 和 ClassLoader 有什么区别？
4. 一部全网最全的 JDK 发展历史轨迹图



微信搜一搜

Q Java后端

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 从零搭建一个 Spring Boot 开发环境！Spring Boot+Mybatis+Swagger2 环境搭建

calebman Java后端 2019-11-15

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | calebman

链接 | [www.jianshu.com/p/95946d6b0c7d](http://www.jianshu.com/p/95946d6b0c7d)

## 本文简介

- 为什么使用Spring Boot
- 搭建怎样一个环境
- 开发环境
- 导入快速启动项目
- 集成前准备
- 集成Mybatis
- 集成Swagger2
- 多环境配置
- 多环境下的日志配置
- 常用配置

## 为什么使用Spring Boot

Spring Boot 相对于传统的SSM框架的优点是提供了默认的样板化配置，简化了Spring应用的初始搭建过程，如果你不想被众多的xml配置文件困扰，可以考虑使用Spring Boot替代

## 搭建怎样一个环境

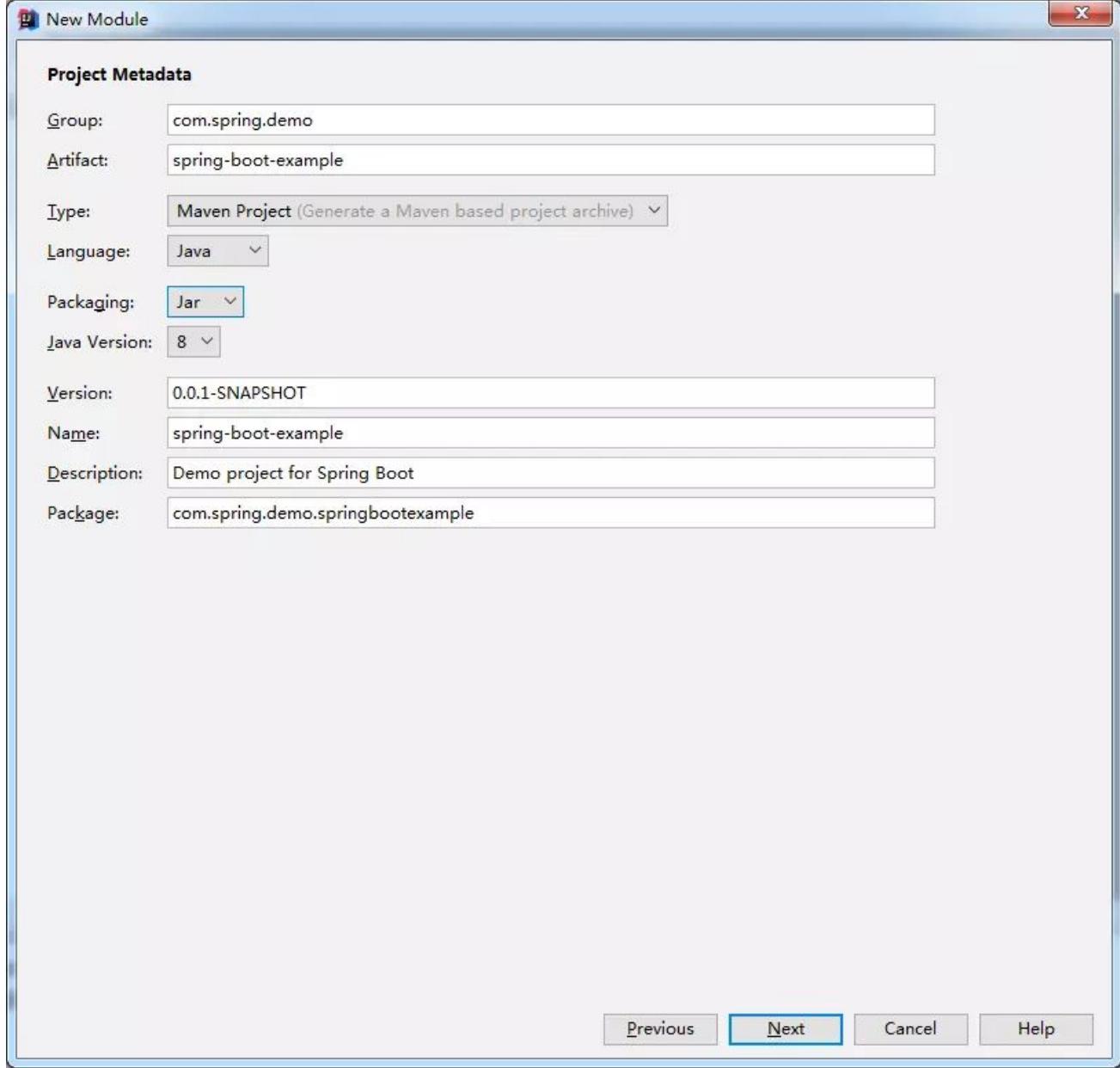
本文将基于Spring官方提供的快速启动项目模板集成Mybatis、Swagger2框架，并讲解mybatis generator一键生成代码插件、logback、一键生成文档以及多环境的配置方法，最后再介绍一下自定义配置的注解获取、全局异常处理等经常用到的东西。

## 开发环境

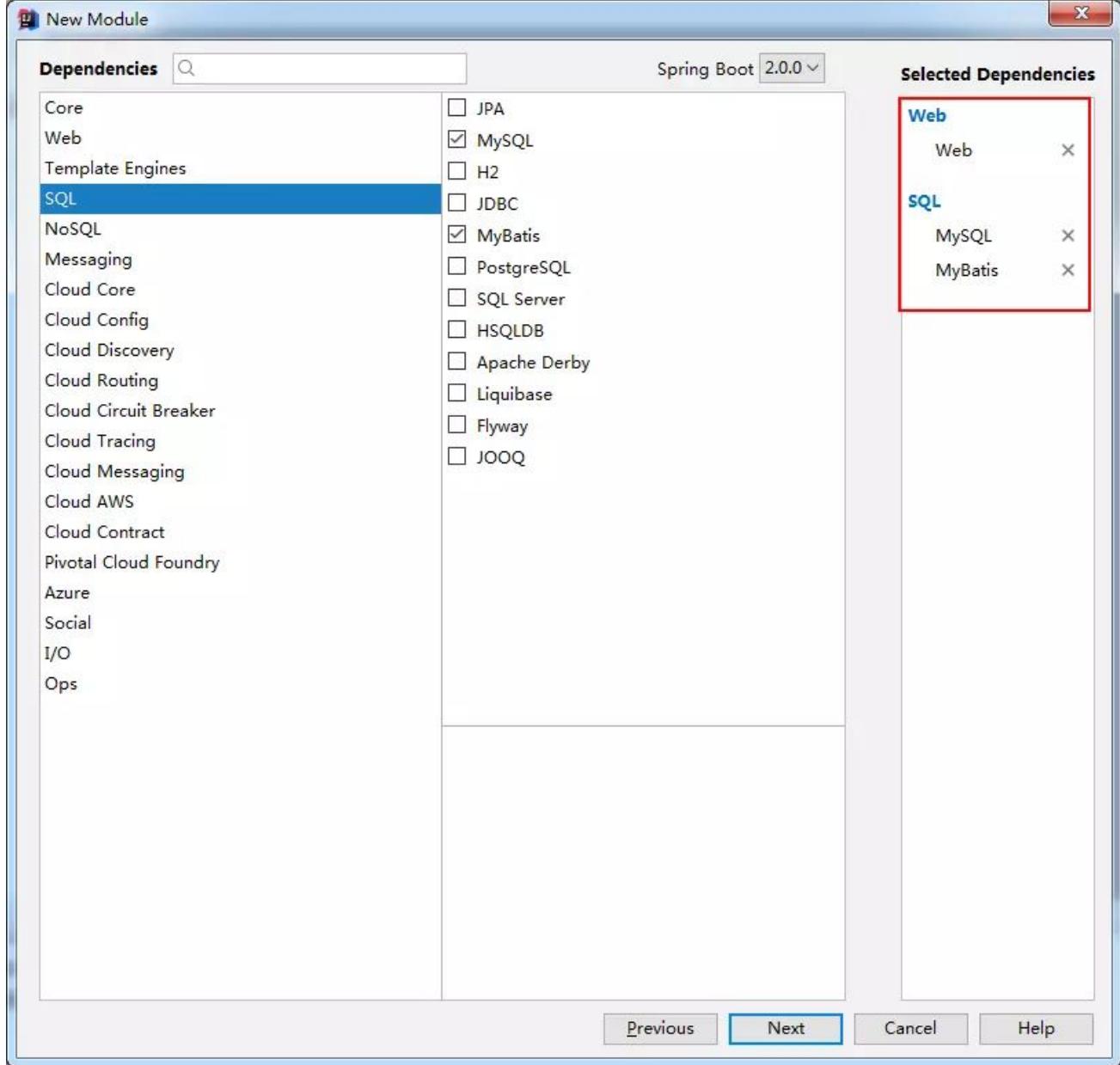
本人使用IDEA作为开发工具，IDEA下载时默认集成了SpringBoot的快速启动项目可以直接创建，如果使用Eclipse的同学可以考虑安装SpringBoot插件或者直接从这里配置并下载SpringBoot快速启动项目，需要注意的是本次环境搭建选择的是SpringBoot2.0的快速启动框架，Spring Boot2.0要求jdk版本必须要在1.8及以上。

## 导入快速启动项目

不管是通过IDEA导入还是通过命令行下载模板工程都需要初始化快速启动工程的配置，如果使用IDEA，在新建项目时选择Spring Initializr，主要配置如下图



IDEA新建SpringBoot项目-填写项目/包名

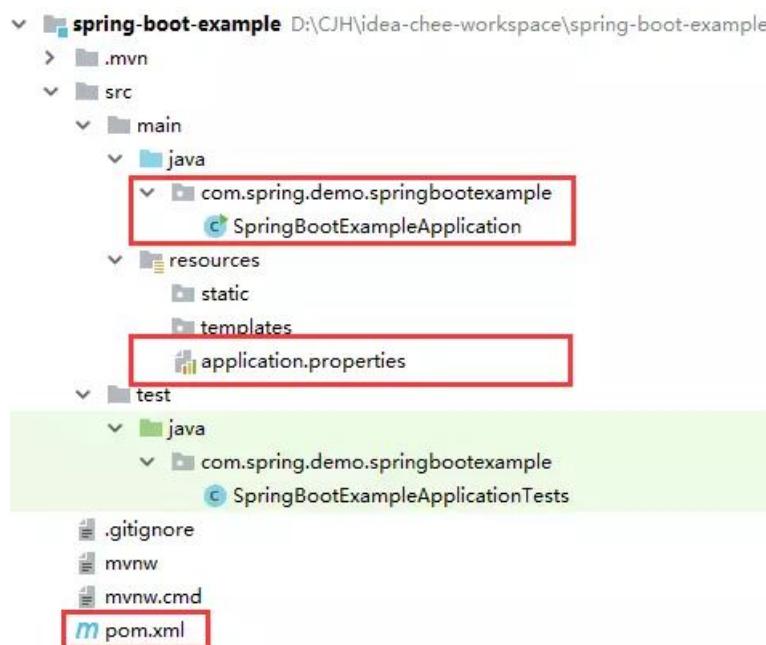


IDEA新建SpringBoot项目-选择依赖包

点击next之后finish之后IDEA显示正在下载模板工程，下载完成后会根据pom.xml下载包依赖，依赖下载完毕后模板项目就算创建成功了，如果是直接从官方网站配置下载快速启动项目可参考下图配置

The screenshot shows the Spring Initializr website interface. At the top, it says 'SPRING INITIALIZR bootstrap your application now'. Below that, there's a search bar with 'Generate a [Maven Project] with [Java] and Spring Boot 2.0.0'. The 'Project Metadata' section has 'Artifact coordinates' fields for 'Group' (set to 'com.spring.demo') and 'Artifact' (set to 'spring-boot-example'). The 'Dependencies' section has a 'Search for dependencies' input field containing 'Web, Security, JPA, Actuator, Devtools...'. Under 'Selected Dependencies', 'Web', 'MySQL', and 'MyBatis' are highlighted in green. At the bottom, there's a 'Generate Project' button and a note: 'Don't know what to look for? Want more options? Switch to the full version.'

从Search for dependencies 框中输入并选择Web、 Mysql、 Mybatis加入依赖，点击Generate Project下载快速启动项目，然后在IDE中选择导入Maven项目，项目导入完成后可见其目录结构如下图



快速启动项目-项目结构

需要关注红色方框圈起来的部分，由上往下第一个java类是用来启动项目的入口函数，第二个properties后缀的文件是项目的配置文件，第三个是项目的依赖包以及执行插件的配置

## 集成前准备

修改.properties为.yml

yml相对于properties更加精简而且很多官方给出的Demo都是yml的配置形式，在这里我们采用yml的形式代替properties，相对于properties形式主要有以下两点不同

1. 对于键的描述由原有的 "." 分割变成了树的形状
2. 对于所有的键的后面一个要跟一个空格，不然启动项目会报配置解析错误

```
properties式语法描述
spring.datasource.name = mysql
spring.datasource.url = jdbc:mysql://localhost:3306/db?characterEncoding=utf-8
spring.datasource.username = root
spring.datasource.password = 123
yml式语法描述
spring:
 datasource:
 name: mysql
 url: jdbc:mysql://localhost:3306/db?characterEncoding=utf-8
 username: root
 password: 123
```

## 配置所需依赖

快速启动项目创建成功后我们观察其pom.xml文件中的依赖如下图，包含了我们选择的Web、 Mybatis以及Mysql

```

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

<dependency>
 <groupId>org.mybatis.spring.boot</groupId>
 <artifactId>mybatis-spring-boot-starter</artifactId>
 <version>1.3.1</version>
</dependency>

<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <scope>runtime</scope>
</dependency>

<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-test</artifactId>
 <scope>test</scope>
</dependency>

```

但是我们使用ORM框架一般还会配合数据库连接池以及分页插件来使用，在这里我选择了阿里的druid以及pagehelper这个分页插件，再加上我们还需要整合swagger2文档自动化构建框架，所以增加了以下四个依赖项

```

<dependency>
 <groupId>com.github.pagehelper</groupId>
 <artifactId>pagehelper-spring-boot-starter</artifactId>
 <version>1.2.3</version>
</dependency>

<dependency>
 <groupId>com.alibaba</groupId>
 <artifactId>druid-spring-boot-starter</artifactId>
 <version>1.1.1</version>
</dependency>

<dependency>
 <groupId>com.alibaba</groupId>
 <artifactId>fastjson</artifactId>
 <version>1.2.31</version>
</dependency>

<dependency>
 <groupId>io.springfox</groupId>
 <artifactId>springfox-swagger2</artifactId>
 <version>2.5.0</version>
</dependency>

```

## 集成Mybatis

Mybatis的配置主要包括了druid数据库连接池、pagehelper分页插件、mybatis-generator代码逆向生成插件以及mapper、pojo扫描配置

### 配置druid数据库连接池

添加以下配置至application.yml文件中

```

spring:
 datasource:
 # 如果存在多个数据源，监控的时候可以通过名字来区分开来
 name: mysql
 # 连接数据库的url
 url: jdbc:mysql://localhost:3306/db?characterEncoding=utf-8
 # 连接数据库的账号
 username: root
 # 连接数据库的密码
 password: 123
 # 使用druid数据源
 type: com.alibaba.druid.pool.DruidDataSource
 # 扩展插件
 # 监控统计用的filter:stat 日志用的filter:log4j 防御sql注入的filter:wall
 filters: stat
 # 最大连接池数量
 maxActive: 20
 # 初始化时建立物理连接的个数。初始化发生在显示调用init方法，或者第一次getConnection时
 initialSize: 1
 # 获取连接时最大等待时间，单位毫秒
 maxWait: 60000
 # 最小连接池数量
 minIdle: 1
 timeBetweenEvictionRunsMillis: 60000
 # 连接保持空闲而不被驱逐的最长时间
 minEvictableIdleTimeMillis: 300000
 # 用来检测连接是否有效的sql，要求是一个查询语句
 # 如果validationQuery为null，testOnBorrow、testOnReturn、testWhileIdle都不会起作用
 validationQuery: select count(1) from 'table'
 # 申请连接的时候检测，如果空闲时间大于timeBetweenEvictionRunsMillis，执行validationQuery检测连接是否有效
 testWhileIdle: true
 # 申请连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能
 testOnBorrow: false
 # 归还连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能
 testOnReturn: false
 # 是否缓存PreparedStatement，即PSCache
 poolPreparedStatements: false
 # 要启用PSCache，必须配置大于0，当大于0时，poolPreparedStatements自动触发修改为true
 maxOpenPreparedStatements: -1

```

## 配置pagehelper分页插件

```

pagehelper分页插件
pagehelper:
 # 数据库的方言
 helperDialect: mysql
 # 启用合理化，如果pageNum < 1会查询第一页，如果pageNum > pages会查询最后一页
 reasonable: true

```

## 代码逆向生成插件mybatis-generator的配置及运行

mybatis-generator插件的使用主要分为以下三步

1. pom.xml中添加mybatis-generator插件

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

 <plugin>
 <groupId>org.mybatis.generator</groupId>
 <artifactId>mybatis-generator-maven-plugin</artifactId>
 <version>1.3.2</version>
 <configuration>
 <configurationFile>
 ${basedir}/src/main/resources/generator/generatorConfig.xml
 </configurationFile>
 <overwrite>true</overwrite>
 <verbose>true</verbose>
 </configuration>
 </plugin>
 </plugins>
</build>
```

## 2. 创建逆向代码生成配置文件generatorConfig.xml

参照pom.xml插件配置中的扫描位置，在resources目录下创建generator文件夹，在新建的文件夹中创建generatorConfig.xml配置文件，文件的详细配置信息如下

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
 PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
 "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>

 <properties resource="generator/generator.properties"/>
 <classPathEntry location="${classPathEntry}"/>
 <context id="DB2Tables" targetRuntime="MyBatis3">

 <jdbcConnection
 driverClass="com.mysql.jdbc.Driver"
 connectionURL="jdbc:mysql://localhost:3306/${db}?characterEncoding=utf-8"
 userId="${userId}"
 password="${password}">
 </jdbcConnection>
 <javaTypeResolver>
 <property name="forceBigDecimals" value="false"/>
 </javaTypeResolver>
 <javaModelGenerator targetPackage="${pojoTargetPackage}" targetProject="src/main/java">
 <property name="enableSubPackages" value="true"/>
 <property name="trimStrings" value="true"/>
 </javaModelGenerator>

 <sqlMapGenerator targetPackage="${mapperTargetPackage}" targetProject="src/main/resources">
 <property name="enableSubPackages" value="true"/>
 </sqlMapGenerator>

 <javaClientGenerator type="XMLMAPPER" targetPackage="${daoTargetPackage}" targetProject="src/main/java">
 <property name="enableSubPackages" value="true"/>
 </javaClientGenerator>

 <table tableName "%" schema="${db}"/>
 </context>
</generatorConfiguration>

```

为了将generatorConfig.xml配置模板化，在这里将变动性较大的配置项单独提取出来作为一个generatorConfig.xml的配置文件，然后通过properties标签读取此文件的配置，这样做的好处是当需要多处复用此xml时只需要关注少量的配置项。  
在generatorConfig.xml同级创建generator.properties文件，现只需要配置generator.properties文件即可，配置内容如下

```

请手动配置以下选项
数据库驱动:选择你的本地硬盘上面的数据库驱动包
classPathEntry = D:/CJH/maven-repository/mysql/mysql-connector-java/5.1.30/mysql-connector-java-5.1.30.jar
数据库名称、用户名、密码
db = db
userId = root
password = 123
生成pojo的包名位置 在src/main/java目录下
pojoTargetPackage = com.spring.demo.springbootexample.mybatis.po
生成DAO的包名位置 在src/main/java目录下
daoTargetPackage = com.spring.demo.springbootexample.mybatis.mapper
生成Mapper的包名位置 位于src/main/resources目录下
mapperTargetPackage = mapper

```

### 3. 运行mybatis-generator插件生成Dao、 Model、 Mapping

```

打开命令行cd到项目pom.xml同级目录运行以下命令
mvn mybatis-generator:generate -e

```

## mybatis扫描包配置

至此已经生成了指定数据库对应的实体、映射类，但是还不能直接使用，需要配置mybatis扫描地址后才能正常调用

### 1. 在application.yml配置mapper.xml以及pojo的包地址

```
mybatis:
 # mapper.xml包地址
 mapper-locations: classpath:mapper/*.xml
 # pojo生成包地址
 type-aliases-package: com.spring.demo.springbootexample.mybatis.po
```

### 2. 在SpringBootApplication.java中开启Mapper扫描注解

```
@SpringBootApplication
 @MapperScan("com.spring.demo.springbootexample.mybatis.mapper")
 public class SpringBootExampleApplication {

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

### 测试mapper的有效性

```
@Controller
public class TestController {

 @Autowired
 UserMapper userMapper;

 @RequestMapping("/test")
 @ResponseBody
 public Object test(){

 return userMapper.selectByExample(null);
 }
}
```

启动SpringBootApplication.java的main函数，如果没有在application.yml特意配置server.port那么springboot会采用默认的8080端口运行，运行成功将打印如下日志

```
Tomcat started on port(s): 8080 (http) with context path ''
```

在浏览器输入地址如果返回表格中的所有数据代表mybatis集成成功

```
http://localhost:8080/test
```

## 集成Swagger2

Swagger2是一个文档快速构建工具，能够通过注解自动生成一个Restful风格json形式的接口文档，并可以通过如swagger-ui等工具生成html网页形式的接口文档，swagger2的集成比较简单，使用需要稍微熟悉一下，集成、注解与使用分如下四步

### 1. 建立SwaggerConfig文件

```
@Configuration
public class SwaggerConfig {

 private final String version = "1.0";

 private final String title = "SpringBoot示例工程";

 private final String description = "API文档自动生成示例";

 private final String termsOfServiceUrl = "http://www.kingeid.com";

 private final String license = "MIT";

 private final String licenseUrl = "https://mit-license.org/";

 private final Contact contact = new Contact("calebman", "https://github.com/calebman", "chenjianhui0428@gmail.com");

 @Bean
 public Docket buildDocket() {
 return new Docket(DocumentationType.SWAGGER_2).apiInfo(buildApiInf())
 .select().build();
 }

 private ApiInfo buildApiInf() {
 return new ApiInfoBuilder().title(title).termsOfServiceUrl(termsOfServiceUrl).description(description)
 .version(version).license(license).licenseUrl(licenseUrl).contact(contact).build();
 }
}
```

## 2. 在SpringBootExampleApplication.java中启用Swagger2注解

在@SpringBootApplication注解下面加上@EnableSwagger2注解

## 3. 常用注解示例

```

@Controller
@RequestMapping("/v1/product")

@Api(value = "DocController", tags = {"restful api示例"})
public class DocController extends BaseController {

 @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
 @ResponseBody

 @ApiOperation(value = "修改指定产品", httpMethod = "PUT", produces = "application/json")

 @ApiImplicitParams({@ApiImplicitParam(name = "id", value = "产品ID", required = true, paramType = "path")})
 public WebResult update(@PathVariable("id") Integer id, @ModelAttribute Product product) {
 logger.debug("修改指定产品接收产品id与产品信息=>%d,{}", id, product);
 if (id == null || "" .equals(id)) {
 logger.debug("产品id不能为空");
 return WebResult.error(ERRORDetail.RC_0101001);
 }
 return WebResult.success();
 }
}

```

```

@ApiModel(value = "产品信息")
public class Product {

 @ApiModelProperty(required = true, name = "name", value = "产品名称", dataType = "query")
 private String name;
 @ApiModelProperty(name = "type", value = "产品类型", dataType = "query")
 private String type;
}

```

#### 4. 生成json形式的文档

集成成功后启动项目控制台会打印级别为INFO的日志，截取部分如下，表明可通过访问应用的v2/api-docs接口得到文档api的json格式数据，可在浏览器输入指定地址验证集成是否成功

```

Mapped "[/v2/api-docs],methods=[GET],produces=[application/json || application/hal+json]]"
http://localhost:8080/v2/api-docs

```

## 多环境配置

应用研发过程中多环境是不可避免的，假设我们现在有开发、演示、生产三个不同的环境其配置也不同，如果每次都在打包环节来进行配置难免出错，SpringBoot支持通过命令启动不同的环境，但是配置文件需要满足application-{profile}.properties的格式，profile代表对应环境的标识，加载时可通过不同命令加载不同环境。

```

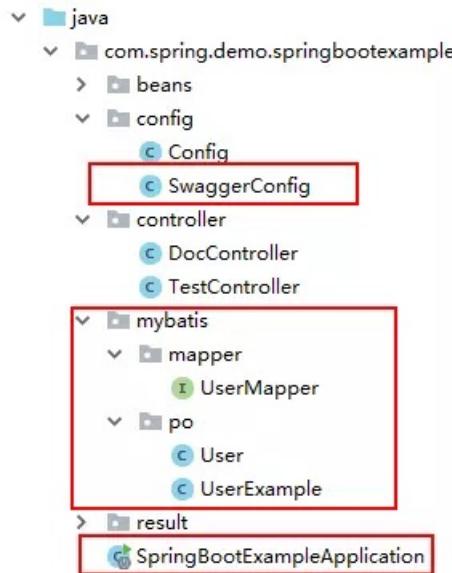
application-dev.properties: 开发环境
application-test.properties: 演示环境
application-prod.properties: 生产环境
运行演示环境命令
java -jar spring-boot-example-0.0.1-SNAPSHOT --spring.profiles.active=test

```

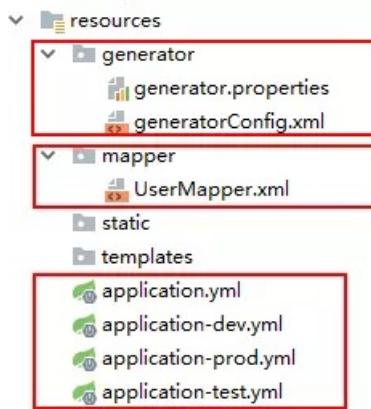
基于现在的项目实现多环境我们需要在application.yml同级目录新建application-dev.yml、application-test.yml、application-prod.yml三个不同环境的配置文件，将不变的公有配置如druid的大部分、pagehelper分页插件以及mybatis包扫描配置放置于application.yml中，并在application.yml中配置默认采用开发环境，那么如果不带--spring.profiles.active启动应用就默认为开发环境启动，变动较大的配置如数据库的账号密码分别写入不同环境的配置文件中

```
spring:
 profiles:
 # 默认使用开发环境
 active: dev
```

配置到这里我们的项目目录结构如下图所示



src/main/java目录结构



src/main/resources目录结构

至此我们分别完成了Mybatis、Swagger2以及多环境的集成，接下来我们配置多环境下的logger。对于logger我们总是希望在项目研发过程中越多越好，能够给予足够的信息定位bug，项目处于演示或者上线状态时为了不让日志打印影响程序性能我们只需要警告或者错误的日志，并且需要写入文件，那么接下来就基于logback实现多环境下的日志配置

## 多环境下的日志配置

创建logback-spring.xml在application.yml的同级目录，springboot推荐使用logback-spring.xml而不是logback.xml文件，logback-spring.xml的配置内容如下所示

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="60 seconds" debug="false">

<property name="maxsize" value="30MB" />

<property name="maxdays" value="90" />

<springProperty scope="context" name="logdir" source="resources.logdir"/>
```

```

<springProperty scope="context" name="appname" source="resources.appname"/>

<springProperty scope="context" name="basepackage" source="resources.basepackage"/>

<appender name="consoleLog" class="ch.qos.logback.core.ConsoleAppender">

<layout class="ch.qos.logback.classic.PatternLayout">
 <pattern>
 <pattern>%d{HH:mm:ss.SSS} [%-5level] %logger{36} - %msg%n</pattern>
 </pattern>
</layout>
</appender>

<appender name="fileLog" class="ch.qos.logback.core.rolling.RollingFileAppender">

<File>${logdir}/${appname}.log</File>

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

 <FileNamePattern>${logdir}/${appname}.%d{yyyy-MM-dd}.log</FileNamePattern>
 <maxHistory>${maxdays}</maxHistory>
 <totalSizeCap>${maxsize}</totalSizeCap>
</rollingPolicy>

<encoder>
 <charset>UTF-8</charset>
 <pattern>%d{HH:mm:ss.SSS} [%-5level] %logger{36} - %msg%n</pattern>
</encoder>
</appender>

<springProfile name="dev">
 <root level="INFO">
 <appender-ref ref="consoleLog"/>
 </root>

 <logger name="${basepackage}" level="DEBUG" additivity="false">
 <appender-ref ref="consoleLog"/>
 </logger>
</springProfile>

<springProfile name="test">
 <root level="WARN">
 <appender-ref ref="consoleLog"/>
 <appender-ref ref="fileLog"/>
 </root>
 <logger name="${basepackage}" level="INFO" additivity="false">
 <appender-ref ref="consoleLog"/>
 <appender-ref ref="fileLog"/>
 </logger>
</springProfile>

<springProfile name="prod">
 <root level="ERROR">
 <appender-ref ref="consoleLog"/>
 <appender-ref ref="fileLog"/>
 </root>
</springProfile>
</configuration>

```

日志配置中引用了application.yml的配置信息，主要有logdir、appname、basepackage三项，logdir是日志文件的写入地址，可以传入相对路径，appname是应用名称，引入这项是为了通过日志文件名称区分是哪个应该输出的，basepackage是包

过滤配置，比如开发环境中需要打印debug级别以上的日志，但是又想使除我写的logger之外的DEBUG不打印，可过滤到本项目的包名才用DEBUG打印，此外包名使用INFO级别打印，在application.yml中新建这三项配置，也可在不同环境配置不同属性

```
#应用配置
resources:
 # log文件写入地址
 logdir: logs/
 # 应用名称
 appname: spring-boot-example
 # 日志打印的基础扫描包
 basepackage: com.spring.demo.springbootexample
```

使用不同环境启动测试logger配置是否生效，在开发环境下将打印DEBUG级别以上的四条logger记录，在演示环境下降打印INFO级别以上的三条记录并写入文件，在生产环境下只打印ERROR级别以上的一条记录并写入文件

```
@RequestMapping("/logger")
@ResponseBody
public WebResult logger() {
 logger.trace("日志输出 {}", "trace");
 logger.debug("日志输出 {}", "debug");
 logger.info("日志输出 {}", "info");
 logger.warn("日志输出 {}", "warn");
 logger.error("日志输出 {}", "error");
 return "00";
}
```

## 常用配置

加载自定义配置

```
@Component
@PropertySource(value = {"classpath:application.yml"}, encoding = "utf-8")
public class Config {

 @Value("${resources.midpHost}")
 private String midpHost;

 public String getMidpHost() {
 return midpHost;
 }
}
```

全局异常处理器

```
@ControllerAdvice
public class GlobalExceptionResolver {

 Logger logger = LoggerFactory.getLogger(GlobalExceptionResolver.class);

 @ExceptionHandler(value = Exception.class)
 @ResponseBody
 public WebResult exceptionHandle(HttpServletRequest req, Exception ex) {
 ex.printStackTrace();
 logger.error("未知异常", ex);
 return WebResult.error(ERRORDetail.RC_0401001);
 }
}
```

[示例工程开源地址](#)

github:https://link.jianshu.com/?t=https%3A%2F%2Fgithub.com%2Fcalebman%2Fspring-boot-example

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 优化你的 Spring Boot

Java后端 2019-09-02

点击上方**蓝色字体**, 选择“标星公众号”

优质文章, 第一时间送达

来自 | Janti

链接 | [cnblogs.com/superfj/p/8667977.html](http://cnblogs.com/superfj/p/8667977.html)

## 介绍

在SpringBoot的Web项目中, 默认采用的是内置Tomcat, 当然也可以配置支持内置的jetty, 内置有什么好处呢?

1. 方便微服务部署。
2. 方便项目启动, 不需要下载Tomcat或者Jetty

针对目前的容器优化, 目前来说没有太多地方, 需要考虑如下几个点

- 线程数
- 超时时间
- jvm优化

针对上述的优化点来说, 首先线程数是一个重点, 初始线程数和最大线程数, 初始线程数保障启动的时候, 如果有大量用户访问, 能够很稳定的接受请求。

而最大线程数量用来保证系统的稳定性, 而超时时间用来保障连接数不容易被压垮, 如果大批量的请求过来, 延迟比较高, 不容易把线程打满。这种情况在生产中是比较常见的, 一旦网络不稳定, 宁愿丢包也不愿意把机器压垮。

jvm优化一般来说没有太多场景, 无非就是加大初始的堆, 和最大限制堆, 当然也不是无限增大, 根据的情况进快速开始

在 Spring Boot 配置文件中application.yml, 添加以下配置

```
1 server:
2 tomcat
3 :
4 min-spare-threads: 2
5 0
 max-threads: 10
 0
 connection-timeout: 5000
```

这块对tomcat进行了一个优化配置, 最大线程数是100, 初始化线程是20, 超时时间是5000ms

## Jvm优化

这块主要不是谈如何优化, jvm优化是一个需要场景化的, 没有什么太多特定参数, 一般来说在server端运行都会指定如下参数

初始内存和最大内存基本会设置成一样的，具体大小根据场景设置，-server是一个必须要用的参数，至于收集器这些使用默认的就可以了，除非有特定需求。

## 1. 使用-server模式

设置JVM使用server模式。64位JDK默认启动该模式

```
1 java -server -jar springboot-1.0.jar
r
```

## 2. 指定堆参数

这个根据服务器的内存大小，来设置堆参数。

- -Xms :设置Java堆栈的初始化大小
- -Xmx :设置最大的java堆大小

```
1 java -server -Xms512m -Xmx768m -jar springboot-1.0.jar
r
```

设置初始化堆内存为512MB，最大为768MB。

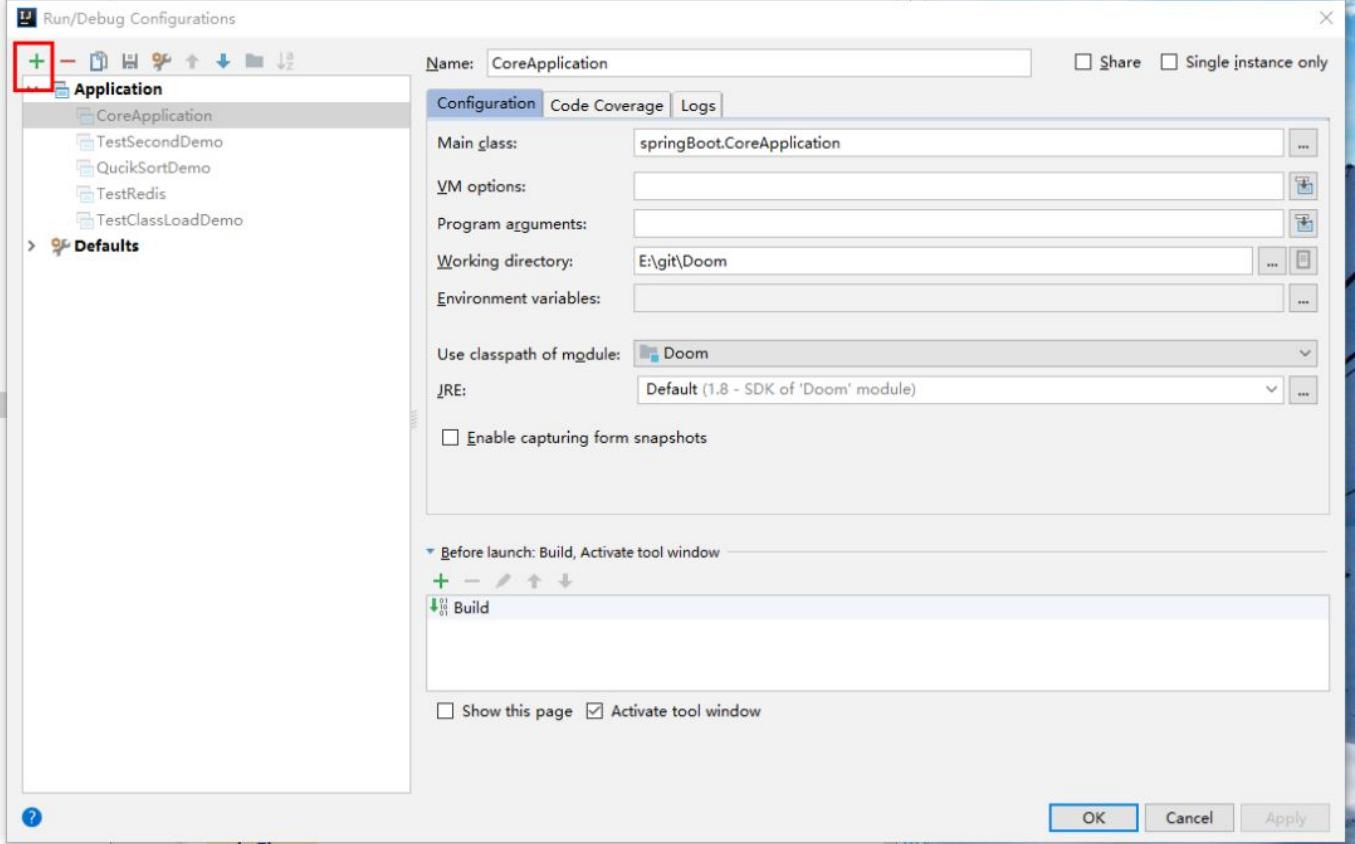
## 3. 远程Debug

在服务器上将启动参数修改为：

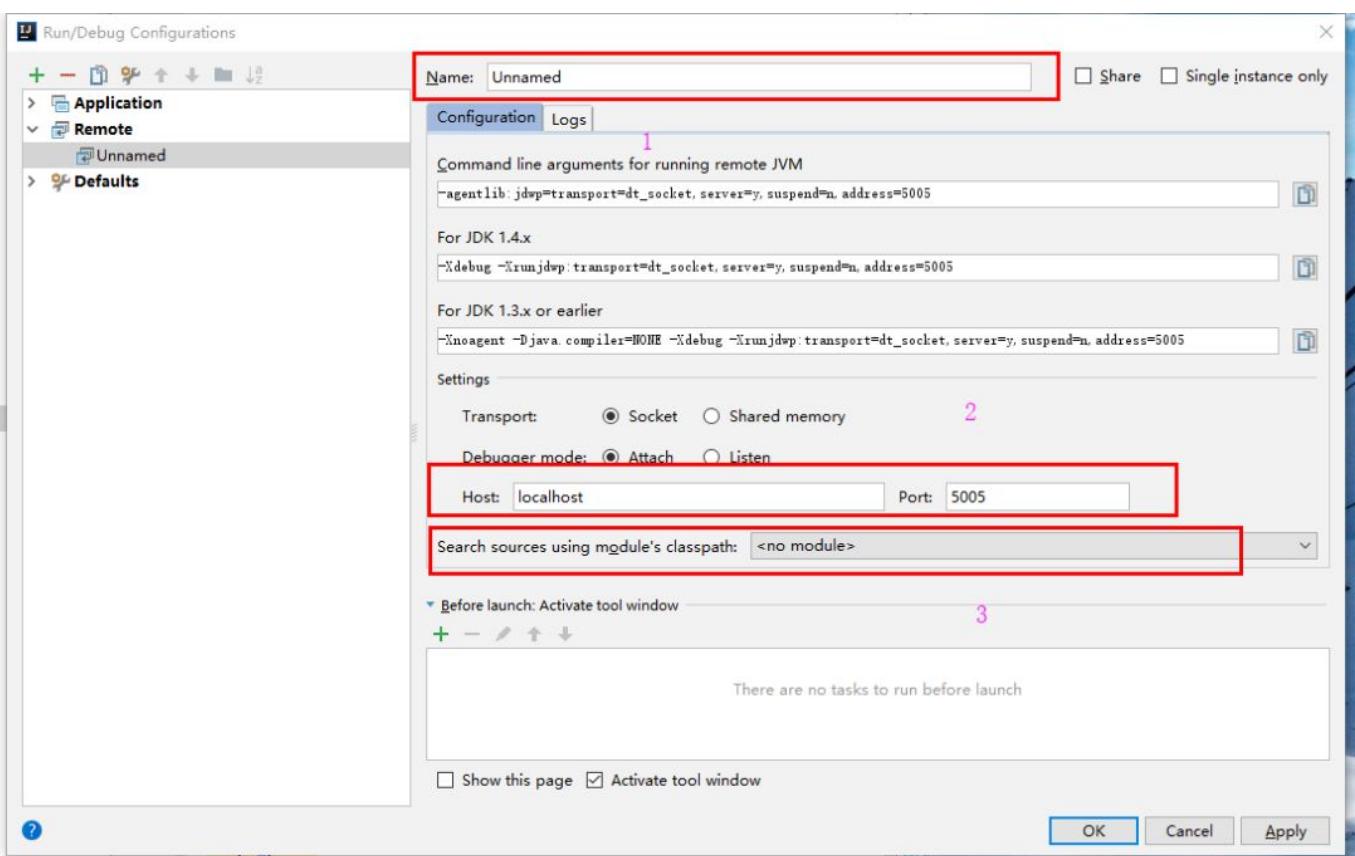
```
1 java -Djavax.net.debug
2 =
3 ssl -Xdebug -Xnoagent -Djava.compiler=
4 NONE -Xrunjdwp:transport=
5 dt_socket,server=y,suspend=
n,address=8888 -jar springboot-1.0.jar
```

这个时候服务端远程Debug模式开启，端口号为8888。

在IDEA中，点击Edit Configuration按钮。



出现弹窗，点击+按钮，找到Remote选项。

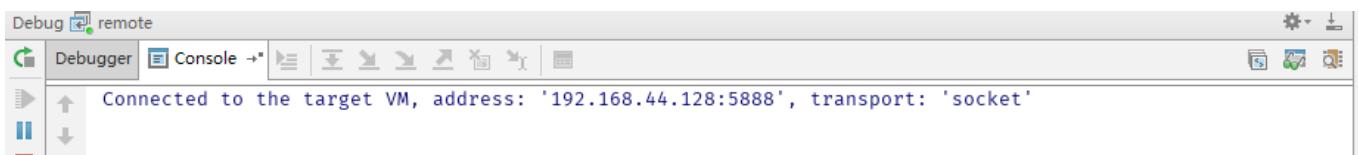


在【1】中填入Remote项目名称，在【2】中填IP地址和端口号，在【3】选择远程调试的项目module，配置完成后点击OK即可

如果碰到连接超时的情况，很有可能服务器的防火墙的问题，举例CentOs7，关闭防火墙

```
1 systemctl stop firewalld.service #停止firewall
2 systemctl disable firewalld.service #禁止firewall开机启动
```

点击debug按钮，IDEA控制台打印信息：



说明远程调试成功。

## JVM工具远程连接

### jconsole与Jvisualvm远程连接

通常我们的web服务都部署在服务器上的，在window使用jconsole是很方便的，相对于Linux就有一些麻烦了，需要进行一些设置。

#### 1.查看hostname,首先使用

```
1 hostname -i
```

查看，服务器的hostname为127.0.0.1，这个是不对的，需要进行修改

#### 2.修改hostname

修改/etc/hosts文件，将其第一行的“127.0.0.1 localhost.localdomain localhost”，修改为：“192.168.44.128 localhost.localdomain localhost”。“192.168.44.128”为实际的服务器的IP地址

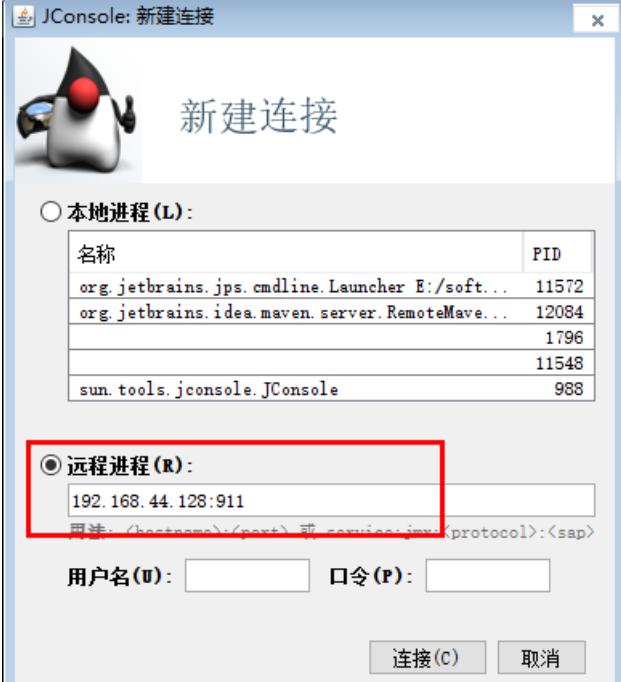
#### 3.重启Linux，在服务器上输入hostname -i，查看实际设置的IP地址是否为你设置的

#### 4.启动服务，参数为：

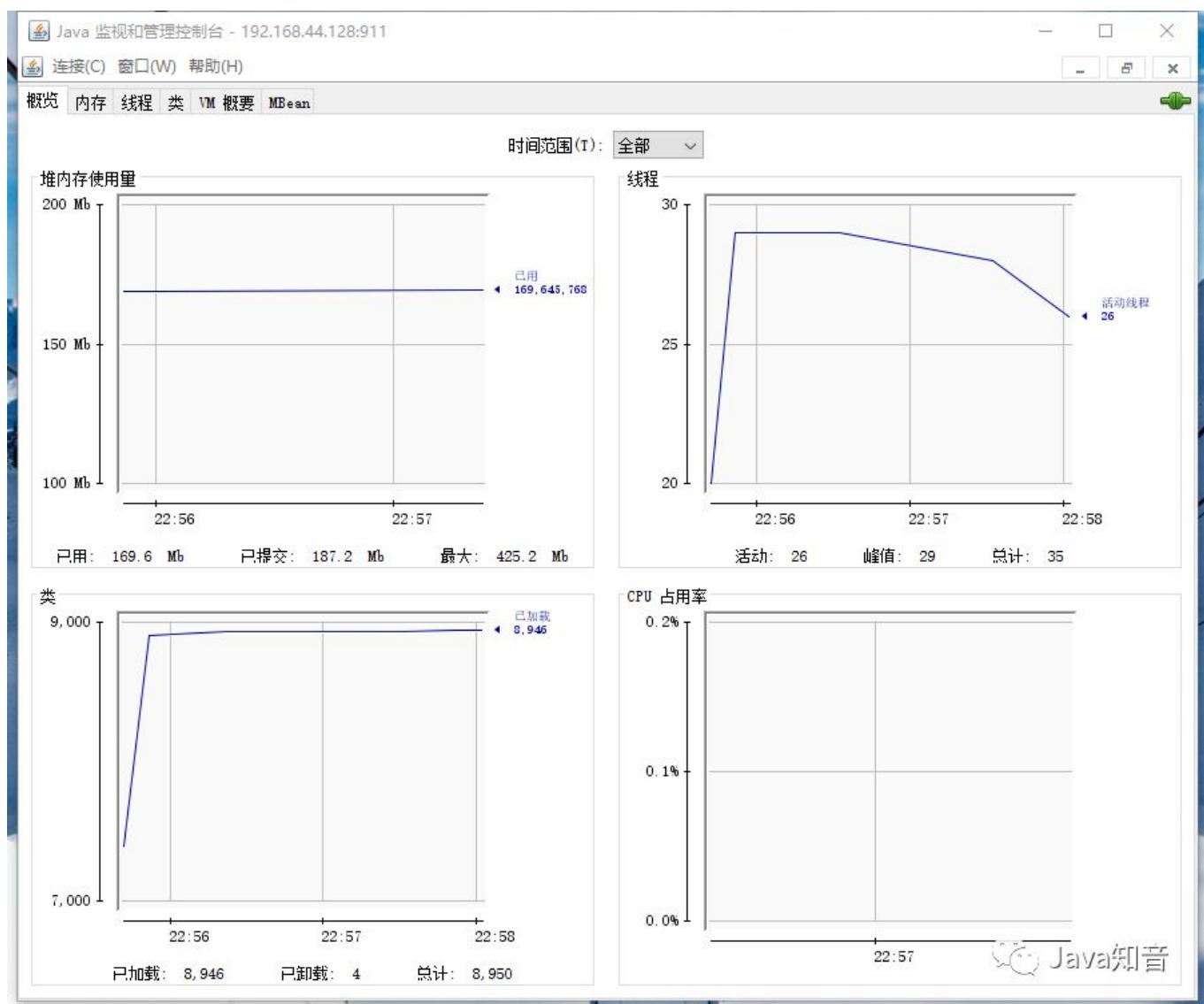
```
1 java -jar -Djava.rmi.server.hostname=192.168.44.128
2 -
3 Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=911
4 -
Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false jantent-1.0-SNAPSHOT.jar
```

ip为192.168.44.128，端口为911。

#### 5.打开Jconsole，进行远程连接，输入IP和端口即可

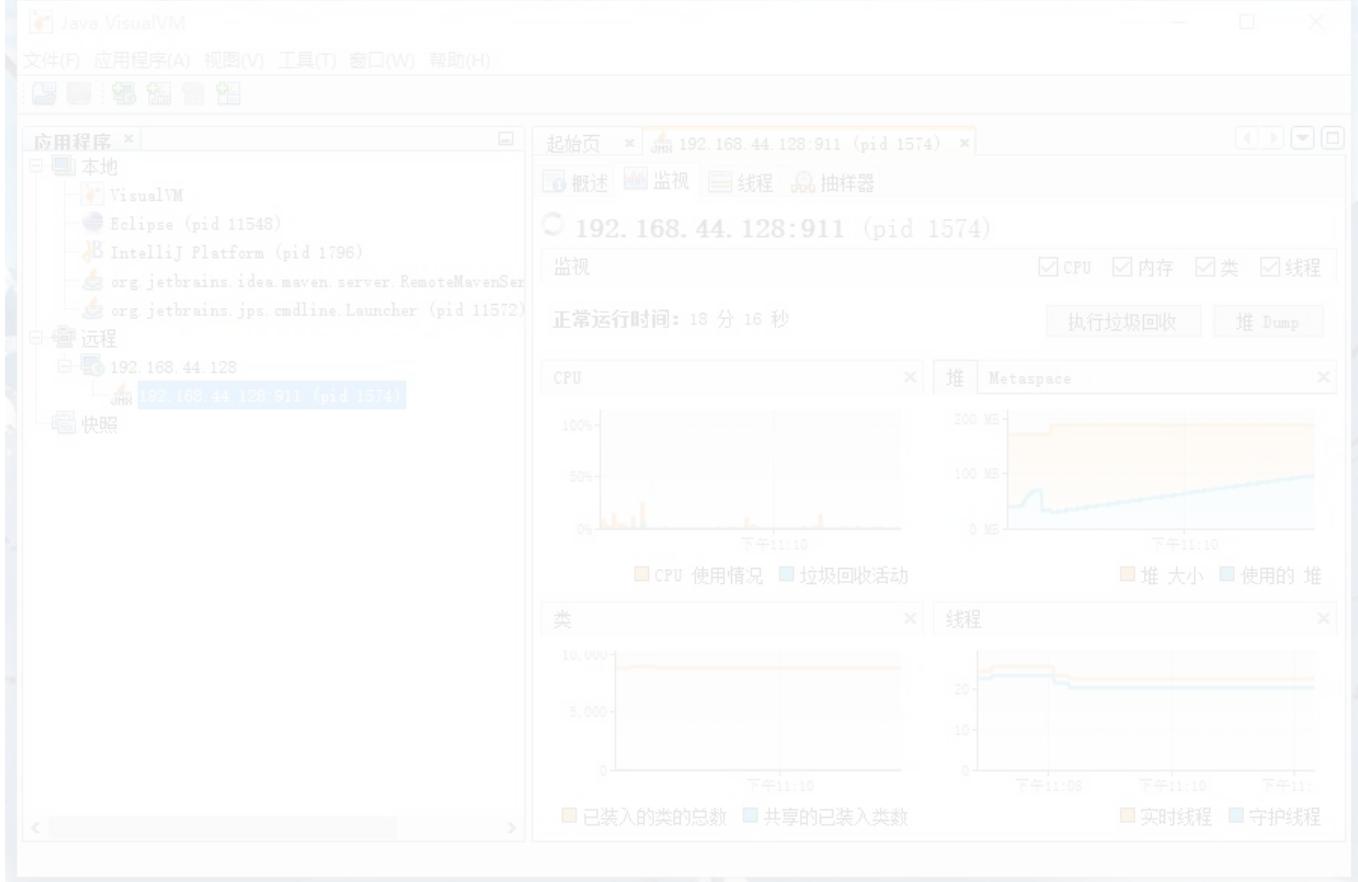


点击连接，经过稍稍等待之后，即可完成连接，如下图所示：



同理，JvisualVm的远程连接是同样的，启动参数也是一样。

然后在本机JvisualVm输入IP:PORT，即可进行远程连接：如下图所示：



相比较Jvisualvm功能更加强大一下，界面也更美观

如果喜欢本篇文章，欢迎[转发](#)、[点赞](#)。关注订阅号「Web项目聚集地」，回复「全栈」即可获取 2019 年最新 Java、Python、前端学习视频资源。

## 推荐阅读

- [1. 这代码写的，狗屎一样](#)
- [2. 这代码写的，狗屎一样（下）](#)
- [3. 除了负载均衡，Nginx 还可以做很多](#)
- [4. 快来薅当当的羊毛！](#)
- [5. 聊一聊 Java 泛型中的通配符](#)
- [6. 数据库不使用外键的 9 个理由](#)



Web项目聚集地

微信扫描二维码，关注我的公众号

喜欢文章，点个在看

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 在 Spring Boot 中，如何干掉 if else

cipher Java后端 2019-09-27

点击上方 Java后端, 选择“[设为星标](#)”

优质文章, 及时送达

来自 | [掘金](#) (作者: cipher)

原文 | [juejin.im/post/5c551122e51d457fcc5a9790](https://juejin.im/post/5c551122e51d457fcc5a9790)

推荐 | Java 8 的这个特性，用起来真的很爽 ([点击查看](#))

## 需求

这里虚拟一个业务需求，让大家容易理解。假设有一个订单系统，里面的一个功能是根据订单的不同类型作出不同的处理。

订单实体：

```
public class OrderDTO {

 private String code;

 private BigDecimal price;

 /**
 * 订单类型
 * 1: 普通订单;
 * 2: 团购订单;
 * 3: 促销订单;
 */
 private String type;

 // ... 省略 get / set ...

}
```

service接口：

```
public interface IOrderService {

 /**
 * 根据订单的不同类型作出不同的处理
 *
 * @param dto 订单实体
 * @return 为了简单，返回字符串
 */
 String handle(OrderDTO dto);

}
```

## 传统实现

根据订单类型写一堆的if else：

```
public class OrderServiceImpl implements IOrderService {

 @Override
 public String handle(OrderDTO dto) {
 String type = dto.getType();
 if ("1".equals(type)) {
 return "处理普通订单";
 } else if ("2".equals(type)) {
 return "处理团购订单";
 } else if ("3".equals(type)) {
 return "处理促销订单";
 }
 return null;
 }
}
```

## 策略模式实现

利用策略模式，只需要两行即可实现业务逻辑：

```
@Service
public class OrderServiceV2Impl implements IOrderService {

 @Autowired
 private HandlerContext handlerContext;

 @Override
 public String handle(OrderDTO dto) {
 AbstractHandler handler = handlerContext.getInstance(dto.getType());
 return handler.handle(dto);
 }

}
```

可以看到上面的方法中注入了HandlerContext，这是一个处理器上下文，用来保存不同的业务处理器，具体在下文会讲解。我们从中获取一个抽象的处理器AbstractHandler，调用其方法实现业务逻辑。

现在可以了解到，我们主要的业务逻辑是在处理器中实现的，因此有多少个订单类型，就对应有多少个处理器。以后需求变化，增加了订单类型，只需要添加相应的处理器就可以，上述OrderServiceV2Impl完全不需改动。

我们先看看业务处理器的写法：

```
@Component
@HandlerType("1")
public class NormalHandler extends AbstractHandler {

 @Override
 public String handle(OrderDTO dto) {
 return "处理普通订单";
 }

}

@Component
@HandlerType("2")
public class GroupHandler extends AbstractHandler {

 @Override
 public String handle(OrderDTO dto) {
 return "处理团购订单";
 }

}
```

```
@Component
@HandlerType("3")
public class PromotionHandler extends AbstractHandler {

 @Override
 public String handle(OrderDTO dto) {
 return "处理促销订单";
 }

}
```

首先每个处理器都必须添加到spring容器中，因此需要加上@Component注解，其次需要加上一个自定义注解@HandlerType，用于标识该处理器对应哪个订单类型，最后就是继承AbstractHandler，实现自己的业务逻辑。

自定义注解 @HandlerType：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface HandlerType {

 String value();
}
```

抽象处理器 AbstractHandler：

```
public abstract class AbstractHandler {
 abstract public String handle(OrderDTO dto);
}
```

自定义注解和抽象处理器都很简单，那么如何将处理器注册到spring容器中呢？

具体思路是：

- 1、扫描指定包中标有@HandlerType的类；
- 2、将注解中的类型值作为key，对应的类作为value，保存在Map中；
- 3、以上面的map作为构造函数参数，初始化HandlerContext，将其注册到spring容器中；

我们将核心的功能封装在HandlerProcessor类中，完成上面的功能。

HandlerProcessor：

```
@Component
@SuppressWarnings("unchecked")
public class HandlerProcessor implements BeanFactoryPostProcessor {

 private static final String HANDLER_PACKAGE = "com.cipher.handler_demo.handler.biz";

 /**
 * 扫描@HandlerType, 初始化HandlerContext, 将其注册到spring容器
 *
 * @param beanFactory bean工厂
 * @see HandlerType
 * @see HandlerContext
 */
 @Override
 public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
 Map<String, Class> handlerMap = Maps.newHashMapWithExpectedSize(3);
 ClassScanner.scan(HANDLER_PACKAGE, HandlerType.class).forEach(clazz -> {
 // 获取注解中的类型值
 String type = clazz.getAnnotation(HandlerType.class).value();
 // 将注解中的类型值作为key, 对应的类作为value, 保存在Map中
 handlerMap.put(type, clazz);
 });
 // 初始化HandlerContext, 将其注册到spring容器中
 HandlerContext context = new HandlerContext(handlerMap);
 beanFactory.registerSingletonHandlerContext(context);
 }

}
```

### ClassScanner: 扫描工具类源码

HandlerProcessor需要实现BeanFactoryPostProcessor，在spring处理bean前，将自定义的bean注册到容器中。

核心工作已经完成，现在看看HandlerContext如何获取对应的处理器：

HandlerContext:

```
public class HandlerContext {

 private Map<String, Class> handlerMap;

 public HandlerContext(Map<String, Class> handlerMap) {
 this.handlerMap = handlerMap;
 }

 public AbstractHandler getInstance(String type) {
 Class clazz = handlerMap.get(type);
 if (clazz == null) {
 throw new IllegalArgumentException("not found handler for type: " + type);
 }
 return (AbstractHandler) BeanTool.getBean(clazz);
 }
}
```

BeanTool: 获取bean工具类

#getInstance 方法根据类型获取对应的class，然后根据class类型获取注册到spring中的bean。

最后请注意一点，HandlerProcessor和BeanTool必须能被扫描到，或者通过@Bean的方式显式的注册，才能在项目启动时发挥作用。

## 总结

利用策略模式可以简化繁杂的if else代码，方便维护，而利用自定义注解和自注册的方式，可以方便应对需求的变更。本文只是提供一个大致的思路，还有很多细节可以灵活变化，例如使用枚举类型、或者静态常量，作为订单的类型，相信你能想到更多更好的方法。

-END-

如果看到这里，说明你喜欢这篇文章，帮忙[转发](#)一下吧，感谢。微信搜索「web\_resource」，关注后回复「进群」即可进入无广告交流群。

↓ 扫描二维码进群 ↓



## 推荐阅读

- [1. Java后端优质文章整理](#)
- [2. 全网最牛掰的12306抢票神器](#)
- [3. 面试官:说一说 Spring Boot 自动配置原理](#)
- [4. 在浏览器输入 URL 回车之后发生了什么?](#)
- [5. 接私活必备的 10 个开源项目](#)



Java后端

长按识别二维码，关注我的公众号

喜欢文章，点个在看

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 在 Spring Boot 项目中使用 activiti

yawn Lau Java后端 2019-12-21

点击上方 Java后端, 选择 **设为星标**

优质文章, 及时送达

作者 | yawn Lau

来源 | [www.jvm123.com](http://www.jvm123.com)

链接 | [www.jvm123.com/2019/08/springboot-activiti/](http://www.jvm123.com/2019/08/springboot-activiti/)

新建springBoot项目时勾选activiti，或者在已建立的springBoot项目添加以下依赖：

```
<dependency>
<groupId>org.activiti</groupId>
<artifactId>activiti-spring-boot-starter-basic</artifactId>
<version>6.0.0</version>
</dependency>
```

数据源和activiti配置：

```
server:
port: 8081

spring:
datasource:
url: jdbc:mysql://localhost:3306/act5?useSSL=true
driver-class-name: com.mysql.jdbc.Driver
username: root
password: root

activiti default configuration
activiti:
database-schema-update: true
check-process-definitions: true
process-definition-location-prefix: classpath:/processes/
process-definition-location-suffixes:
- **.bpmn
- **.bpmn20.xml
history-level: full
```

在activiti的默认配置中, process-definition-location-prefix 是指定activiti流程描述文件的前缀(即路径), 启动时, activiti就会去寻找此路径下的流程描述文件, 并且自动部署; suffix 是一个String数组, 表示描述文件的默认后缀名, 默认以上两种。

springMVC配置：

```

package com.yawn.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.http.HttpStatus;
import org.springframework.web.servlet.config.annotation.*;

/**
 * Created by yawn on 2017/8/5.
 */
@Configuration
@EnableWebMvc
public class MvcConfig extends WebMvcConfigurerAdapter {

 @Override
 public void addResourceHandlers(ResourceHandlerRegistry registry) {
 registry.addResourceHandler("/static/**").addResourceLocations("classpath:/static/");
 registry.addResourceHandler("/templates/**").addResourceLocations("classpath:/templates/");
 super.addResourceHandlers(registry);
 }

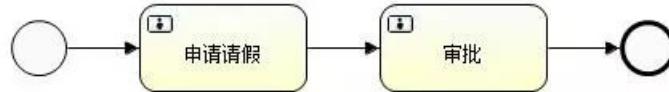
 @Override
 public void addViewControllers(ViewControllerRegistry registry) {
 registry.addViewController("/index");
 registry.addViewController("/user");
 registry.addRedirectViewController("/", "/templates/login.html");
 //registry.addStatusController("/403", HttpStatus.FORBIDDEN);
 super.addViewControllers(registry);
 }
}

```

这里配置静态资源和直接访问的页面：在本示例项目中，添加了thymeleaf依赖解析视图，主要采用异步方式获取数据，通过angularJS进行前端数据的处理和展示。

配置了数据源和activiti后，启动项目，activiti 的各个服务组件就已经被加入到spring容器中了，所以就可以直接注入使用了。如果在未自动配置的spring环境中，可以使用通过指定bean的init-method来配置activiti的服务组件。

以下请假流程为例：



- 开始流程并“申请请假”（员工）

```
private static final String PROCESS_DEFINE_KEY = "vacationProcess";

public Object startVac(String userName, Vacation vac) {

 identityService.setAuthenticatedUserId(userName);
 //开始流程
 ProcessInstance vacationInstance = runtimeService.startProcessInstanceByKey(PROCESS_DEFINE_KEY);
 //查询当前任务
 Task currentTask = taskService.createTaskQuery().processInstanceId(vacationInstance.getId()).singleResult();
 //申明任务
 taskService.claim(currentTask.getId(), userName);

 Map<String, Object> vars = new HashMap<>(4);
 vars.put("applyUser", userName);
 vars.put("days", vac.getDays());
 vars.put("reason", vac.getReason());
 //完成任务
 taskService.complete(currentTask.getId(), vars);

 return true;
}
```

在此方法中，Vacation 是申请时的具体信息，在完成“申请请假”任务时，可以将这些信息设置成参数。

## 2. 审批请假（老板）

### (1) 查询需要自己审批的请假

```

public Object myAudit(String userName) {
 List<Task> taskList = taskService.createTaskQuery().taskCandidateUser(userName)
 .orderByTaskCreateTime().desc().list();
 // 多此一举 taskList中包含了以下内容(用户的任务中包含了所在用户组的任务)
 // Group group = identityService.createGroupQuery().groupMember(userName).singleResult();
 // List<Task> list = taskService.createTaskQuery().taskCandidateGroup(group.getId()).list();
 // taskList.addAll(list);

 List<VacTask> vacTaskList = new ArrayList<>();
 for (Task task : taskList) {
 VacTask vacTask = new VacTask();
 vacTask.setId(task.getId());
 vacTask.setName(task.getName());
 vacTask.setCreateTime(task.getCreateTime());
 String instanceId = task.getProcessInstanceId();
 ProcessInstance instance = runtimeService.createProcessInstanceQuery().processInstanceId(instanceId).singleResult();
 Vacation vac = getVac(instance);
 vacTask.setVac(vac);
 vacTaskList.add(vacTask);
 }
 return vacTaskList;
}

private Vacation getVac(ProcessInstance instance) {
 Integer days = runtimeService.getVariable(instance.getId(), "days", Integer.class);
 String reason = runtimeService.getVariable(instance.getId(), "reason", String.class);
 Vacation vac = new Vacation();
 vac.setApplyUser(instance.getStartUserId());
 vac.setDays(days);
 vac.setReason(reason);
 Date startTime = instance.getStartTime(); // activiti 6 才有
 vac.setApplyTime(startTime);
 vac.setApplyStatus(instance.isEnded() ? "申请结束" : "等待审批");
 return vac;
}

```

package com.yawn.entity;

```

import java.util.Date;

/**
 * @author Created by yawn on 2018-01-09 14:31
 */
public class VacTask {

 private String id;
 private String name;
 private Vacation vac;
 private Date createTime;

 // getter setter ...
}

```

老板查询自己当前需要审批的任务，并且将任务和参数设置到一个VacTask对象，用于页面的展示。

## (2) 审批请假

```

public Object passAudit(String userName, VacTask vacTask) {
 String taskId = vacTask.getId();
 String result = vacTask.getVac().getResult();
 Map<String, Object> vars = new HashMap<>();
 vars.put("result", result);
 vars.put("auditor", userName);
 vars.put("auditTime", new Date());
 taskService.claim(taskId, userName);
 taskService.complete(taskId, vars);
 return true;
}

```

同理，result是审批的结果，也是在完成审批任务时需要传入的参数；taskId是刚才老板查询到的当前需要自己完成的审批任务ID。（如果流程在这里设置分支，可以通过判断result的值来跳转到不同的任务）

### 3. 查询记录

由于已完成的请假在数据库runtime表中查不到（runtime表只保存正在进行的流程示例信息），所以需要在history表中查询。

#### (1) 查询请假记录

```

public Object myVacRecord(String userName) {
 List<HistoricProcessInstance> hisProInstance = historyService.createHistoricProcessInstanceQuery()
 .processDefinitionKey(PROCESS_DEFINE_KEY).startedBy(userName).finished()
 .orderByProcessInstanceId().desc().list();

 List<Vacation> vacList = new ArrayList<>();
 for (HistoricProcessInstance hisInstance : hisProInstance) {
 Vacation vacation = new Vacation();
 vacation.setApplyUser(hisInstance.getStartTime());
 vacation.setApplyTime(hisInstance.getEndTime());
 vacation.setApplyStatus("申请结束");
 List<HistoricVariableInstance> varInstanceList = historyService.createHistoricVariableInstanceQuery()
 .processInstanceId(hisInstance.getId()).list();
 ActivitiUtil.setVars(vacation, varInstanceList);
 vacList.add(vacation);
 }
 return vacList;
}

```

请假记录即查出历史流程实例，再查出关联的历史参数，将历史流程实例和历史参数设置到Vacation对象（VO对象）中去，即可返回，用来展示。

```

package com.yawn.util;

import org.activiti.engine.history.HistoricVariableInstance;

import java.lang.reflect.Field;
import java.util.List;

/**
 * activiti中使用得到的工具方法
 * @author Created by yawn on 2018-01-10 16:32
 */
public class ActivitiUtil {

 /**
 * 将历史参数列表设置到实体中去
 * @param entity 实体
 * @param varInstanceList 历史参数列表
 */
 public static <T> void setVars(T entity, List<HistoricVariableInstance> varInstanceList) {
 Class<?> tClass = entity.getClass();
 try {
 for (HistoricVariableInstance varInstance : varInstanceList) {
 Field field = tClass.getDeclaredField(varInstance.getVariableName());
 if (field == null) {
 continue;
 }
 field.setAccessible(true);
 field.set(entity, varInstance.getValue());
 }
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
}

```

此外，以上是查询历史流程实例和历史参数后，设置VO对象的通用方法：可以根据参数列表中的参数，将与VO对象属性同名的参数设置到VO对象中去。

#### 4. 前端展示和操作

##### (1) 审批列表和审批操作示例

### 待我审核的请假

任务名称	任务时间	申请人	申请时间	天数	事由	操作
经理审批	2018-01-12 10:27:48	managea	2018-01-12 10:27:48	5	有紧急事情	<input type="button" value="审核通过"/> <input type="button" value="审核拒绝"/>

### 我的审核记录

时间	申请人	天数	事由	申请状态	审核人	审核结果	审核时间
2018-01-12 10:26:53	empa	4	有紧急事情	申请结束	dira	审核拒绝	2018-01-12 10:28:16
2018-01-12 10:27:06	empb	5	有紧急事情	申请结束	dira	审核通过	2018-01-12 10:28:12

```

<div ng-controller="myAudit">
 <h2 ng-init="myAudit()">待我审核的请假</h2>
 <table border="0">
 <tr>
 <td>任务名称</td>
 <td>任务时间</td>
 <td>申请人</td>
 <td>申请时间</td>
 <td>天数</td>
 <td>事由</td>
 <td>操作</td>
 </tr>
 <tr ng-repeat="vacTask in vacTaskList">
 <td>{{vacTask.name}}</td>
 <td>{{vacTask.createTime | date:'yyyy-MM-dd HH:mm:ss'}}</td>
 <td>{{vacTask.vac.applyUser}}</td>
 <td>{{vacTask.vac.applyTime | date:'yyyy-MM-dd HH:mm:ss'}}</td>
 <td>{{vacTask.vac.days}}</td>
 <td>{{vacTask.vac.reason}}</td>
 <td>
 <button type="button" ng-click="passAudit(vacTask.id, 1)">审核通过</button>
 <button type="button" ng-click="passAudit(vacTask.id, 0)">审核拒绝</button>
 </td>
 </tr>
 </table>
</div>

```

```

app.controller("myAudit", function ($scope, $http, $window) {
 $scope.vacTaskList = [];

```

```

 $scope.myAudit = function () {
 $http.get(
 "/myAudit"
).then(function (response) {
 $scope.vacTaskList = response.data;
 })
 };

```

```

 $scope.passAudit = function (taskId, result) {
 $http.post(
 "/passAudit",
 {
 "id": taskId,
 "vac": {
 "result": result >= 1 ? "审核通过" : "审核拒绝"
 }
 }
).then(function (response) {
 if (response.data === true) {
 alert("操作成功! ");
 $window.location.reload();
 } else {
 alert("操作失败! ");
 }
 })
 };
});
```

以上是一个springBoot 与 activiti 6.0 整合的示例项目的部分代码与说明，完整的项目代码在：

<https://gitee.com/yawensilence/activiti-demo6-springboot>。

原文请查看作者博客：

<http://www.jvm123.com/2019/08/springboot-activiti/>

-END-

#### 推荐阅读

1. 955 不加班的公司名单:955.WLB
2. 8 岁上海小学生B站教编程惊动苹果
3. 我在华为做外包的真实经历!
4. 什么是一致性 Hash 算法?
5. 团队开发中 Git 最佳实践



喜欢文章, 点个在看 

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 基于 Spring Boot 的 Restful 风格实现增删改查

Java后端 2019-09-05

点击上方**蓝色字体**, 选择“标星公众号”

优质文章, 第一时间送达

作者 | 虚无境

链接 | [cnblogs.com/xuwujing/p/8260935.html](http://cnblogs.com/xuwujing/p/8260935.html)

## 前言

在去年的时候, 在各种渠道中略微的了解了Spring Boot, 在开发web项目的时候是如何的方便、快捷。但是当时并没有认真的去学习下, 毕竟感觉自己在Struts和Spring MVC都用得不太熟练。不过在看了很多关于Spring Boot的介绍之后, 并没有想象中的那么难, 于是开始准备学习Spring Boot。

在闲暇之余的时候, 看了下Spring Boot实战以及一些大神关于Spring Boot的博客之后, 开始写起了我的第一个Spring Boot的项目。在能够对Spring Boot进行一些简单的开发Restful风格接口实现CRUD功能之后, 于是便有了本篇博文。

## Spring Boot介绍

Spring Boot是由Pivotal团队提供的全新框架, 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。

简单的来说就是, 只需几个jar和一些简单的配置, 就可以快速开发项目。

假如我就想简单的开发一个对外的接口, 那么只需要以下代码就可以了。

## 一个主程序启动springBoot

```
1 @SpringBootApplication
2 public class Application {
3 public static void main(String[] args)
4 {
5 SpringApplication.run(Application.class, args);
6 }
7 }
```

## 控制层

```
1 @RestController
2 public class HelloWorldController {
3 @RequestMapping("/hello"
4)
5 public String index() {
6 return "Hello World"
7 }
8 }
```

成功启动主程序之后,编写控制层,然后在浏览器输入 <http://localhost:8080//hello> 便可以查看信息。

感觉使用SpringBoot开发程序是不是非常的简单呢!

用SpringBoot实战的话来说:

这里没有配置,没有web.xml,没有构建说明,甚至没有应用服务器,但这就是整个应用程序了。SpringBoot会搞定执行应用程序所需的各种后勤工作,你只要搞定应用程序的代码就好。

基于SpringBoot开发一个Restful服务

## 一、开发准备

### 1.1 数据库和表

首先,我们需要在MySql中创建一个数据库和一张表

数据库的名称为 springboot,表名称为 t\_user

脚本如下:

```
1 CREATE DATABASE `springboot`;
2
3 USE `springboot`;
4
5 DROP TABLE IF EXISTS `t_user`;
6
7 CREATE TABLE `t_user` (
8 `id` int(11) NOT NULL AUTO_INCREMENT COMMENT 'id',
9 `name` varchar(10) DEFAULT NULL COMMENT '姓名',
10 `age` int(2) DEFAULT NULL COMMENT '年龄',
11 PRIMARY KEY (`id`)
12)
13) ENGINE=InnoDB AUTO_INCREMENT=12 DEFAULT CHARSET=utf8;
```

### 1.2 maven相关依赖

因为我们使用Maven创建的,所以需要添加SpringBoot的相关架包。

这里Maven的配置如下:

springBoot最核心的jar  
spring-boot-starter :核心模块,包括自动配置支持、日志和YAML;

```
1 <dependencies>
2 >
3 <dependency>
4 >
5 <groupId>org.springframework.boot</groupId>
```

```
6 >
7 <artifactId>spring-boot-starter-web</artifactId
8 >
9 </dependency>
10 >
11 <dependency>
12 >
13 <groupId>org.springframework.boot</groupId>
14 >
15 <artifactId>spring-boot-starter-thymeleaf</artifactId>
16 >
17 </dependency>
18 >
19 <dependency>
20 >
21 <groupId>org.springframework.boot</groupId>
22 >
23 <artifactId>spring-boot-starter-data-jpa</artifactId>
24 >
25 </dependency>
26 >
27
28
29 <dependency>
30 >
31 <groupId>org.springframework.boot</groupId>
32 >
33 <artifactId>spring-boot-devtools</artifactId>
34 >
35 <optional>true</optional>
36 >
37 </dependency>
38 >
39
40 <dependency>
41 >
42 <groupId>org.springframework.boot</groupId>
43 >
44 <artifactId>spring-boot-starter-test</artifactId>
45 >
46 <scope>test</scope>
47 >
48 </dependency>
49 >
```

```
<!-- Spring Boot Mybatis 依赖 -->
<dependency>
 <groupId>org.mybatis.spring.boot</groupId>
 <artifactId>mybatis-spring-boot-starter</artifactId>
 <version>${mybatis-spring-boot}</version>
```

```

>
 </dependency>
>

<!-- MySQL 连接驱动依赖 -->
<dependency>
>
 <groupId>mysql</groupId>
>
 <artifactId>mysql-connector-java</artifactId>
>
 <version>${mysql-connector}</version>
>
</dependency>
>

</dependencies>
>

```

## 二、工程说明

成功创建好数据库以及下载好相应架包之后。

我们来正式开发SpringBoot项目。

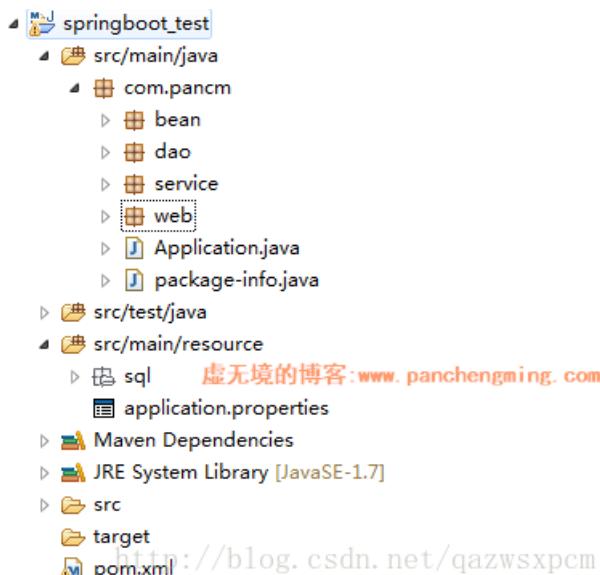
### 2.1 工程结构图：

首先确定工程结构，这里我就简单的说明下了。

```

com.pancm.web - Controller 层
com.pancm.dao - 数据操作层 DAO
com.pancm.bean - 实体类
com.pancm.service - 业务逻辑层
Application - 应用启动类
application.properties - 应用配置文件，应用启动会自动读取配置

```



## 2.2 自定义配置文件

一般我们需要一些自定义的配置，例如配置jdbc的连接配置，在这里我们可以用 `application.properties` 进行配置。数据源实际的配置以各位的为准。

```
1 ## 数据源配置
2 spring.datasource.url=jdbc:mysql://localhost:3306/springBoot?useUnicode=true&characterEncoding=utf8
3 spring.datasource.username=root
4 spring.datasource.password=123456
5 spring.datasource.driver-class-name=com.mysql.jdbc.Driver
6
7
8 ## Mybatis 配置
9 # 配置为 com.pancm.bean 指向实体类包路径。
10 mybatis.typeAliasesPackage=com.pancm.bean
11 # 配置为 classpath 路径下 mapper 包下，* 代表会扫描所有 xml 文件。
12 mybatis.mapperLocations=classpath\:mapper*.xml
```

## 三、代码编写

在创建好相关工程目录之后，我们开始来编写相应的代码。

### 3.1 实体类编写

由于我们这里只是用于测试，只在数据库中创建了一张`t_user`表，所以这里我们就只创建一个`User`实体类，里面的字段对应`t_user`表的字段。

示例代码如下：

```
1 public class User {
2 /** 编号 */
3 private int id;
4 /** 姓名 */
5 private String name;
6 /** 年龄 */
7 private int age;
8
9 public User()
10 {
11 }
12 public class User
13 {
14 /** 编号 */
15 private int id;
16 /** 姓名 */
17 private String name;
18 /** 年龄 */
19 private int age;
20 }
```

```
21 public User() {
22 }
23 // getter和 setter 略
24 }
```

### 3.2 Dao层编写

在以前的Dao层这块，hibernate和mybatis都可以使用注解或者使用mapper配置文件。在这里我们使用spring的JPA来完成基本的增删改查。

#### 说明：

一般有两种方式实现与数据库实现CRUD：

- 第一种是xml的mapper配置。
- 第二种是使用注解，@Insert、@Select、@Update、@Delete 这些来完成。本篇使用的是第二种。

```
1 @Mapper
2 public interface UserDao {
3
4 /**
5 * 用户数据新增
6 */
7 @Insert("insert into t_user(id,name,age) values (#{id},#{name},#{age})"
8)
9 void addUser(User user);
10
11 /**
12 * 用户数据修改
13 */
14 @Update("update t_user set name=#{name},age=#{age} where id=#{id}"
15)
16 void updateUser(User user);
17
18 /**
19 * 用户数据删除
20 */
21 @Delete("delete from t_user where id=#{id}"
22)
23 void deleteUser(int id);
24
25 /**
26 * 根据用户名查询用户信息
27 */
28 @Select("SELECT id,name,age FROM t_user where name=#{userName}"
29)
30 User findByName(@Param("userName") String userName);
31
32 /**
33 * 查询所有
```

```
35 */
36 @Select("SELECT id,name,age FROM t_user")

 List<User> findAll();

}
```

说明:

- mapper : 在接口上添加了这个注解表示这个接口是基于注解实现的CRUD。
- Results: 返回的map结果集, property 表示User类的字段, column 表示对应数据库的字段。
- Param:sql条件的字段。
- Insert、Select、Update、Delete: 对应数据库的增、查、改、删。

### 3.3 Service 业务逻辑层

这块和hibernate、mybatis的基本一样。

代码如下:

#### 接口

```
1 import com.pancm.bean.User;/** * * Title: UserService* Description:用户接口 * Version:1.0.0 * @author pancm */public
2 interface UserService {
3 User findUserById(int id);
4 void addUser(User user);
5 void updateUser(User user);
6 void deleteUser(int id);
7 }
8
```

#### 实现类

```
1 @Service
2 public class UserServiceImpl implements UserService {
3 @Autowired
4 private UserDao userDao;
5
6 @Override
7 public boolean addUser(User user)
8 {
9 boolean flag=false
10 ;
11 try
12 {
13 userDao.addUser(user);
14 flag=true
15 }
16 ;
17 }catch(Exception e){
18 e.printStackTrace();
19 }
20 return flag
21 }
22 }
```

```
22
23 @Override
24 public boolean updateUser(User user)
25 {
26 boolean flag=false
27 ;
28 try
29 {
30 userDao.updateUser(user);
31 flag=true
32 ;
33 }catch(Exception e){
34 e.printStackTrace();
35 }
36 return flag
37 ;
38 }
39
40 @Override
41 public boolean deleteUser(int id)
42 {
43 boolean flag=false
44 ;
45 try
46 {
47 userDao.deleteUser(id);
48 flag=true
49 ;
50 }catch(Exception e){
51 e.printStackTrace();
52 }
53 return flag
54 ;
55 }
56
57 @Override
58 public User findUserByName(String userName)
59 {
60 return userDao.findByName(userName);
61 }
62
63
64 @Override
65 public List<User> findAll()
66 {
67 return userDao.findAll();
68 }
69 }
```

### 3.4 Controller 控制层

控制层这块和springMVC很像，但是相比而言要简洁不少。

说明：

- **RestController**: 默认类中的方法都会以json的格式返回。
- **RequestMapping**: 接口路径配置。
- **method** : 请求格式。
- **RequestParam**: 请求参数。

具体实现如下：

```
1 @RestController
2 @RequestMapping(value = "/api/user"
3)
4 public class UserRestController {
5 @Autowired
6 private UserService userService;
7
8 @RequestMapping(value = "/user", method = RequestMethod.POST)
9 public boolean addUser(User user) {
10 System.out.println("开始新增...")
11 ;
12 return userService.addUser(user);
13 }
14
15 @RequestMapping(value = "/user", method = RequestMethod.PUT)
16 public boolean updateUser(User user) {
17 System.out.println("开始更新...")
18 ;
19 return userService.updateUser(user);
20 }
21
22 @RequestMapping(value = "/user", method = RequestMethod.DELETE)
23 public boolean delete(@RequestParam(value = "userName", required = true) int userId) {
24 System.out.println("开始删除...")
25 ;
26 return userService.deleteUser(userId);
27 }
28
29
30 @RequestMapping(value = "/user", method = RequestMethod.GET)
31 public User findByName(@RequestParam(value = "userName", required = true) String userName) {
32 System.out.println("开始查询...")
33 ;
34 return userService.findUserByName(userName);
35 }
36
37
38 @RequestMapping(value = "/userAll", method = RequestMethod.GET)
39 public List<User> findByUserAge() {
40 System.out.println("开始查询所有数据...")
41 ;
42 return userService.findAll();
43 }
```

```
 }
}
```

### 3.5 Application 主程序

SpringApplication 则是用于从main方法启动Spring应用的类。

默认，它会执行以下步骤：

1. 创建一个合适的ApplicationContext实例（取决于classpath）。
2. 注册一个CommandLinePropertySource，以便将命令行参数作为Spring properties。
3. 刷新application context，加载所有单例beans。
4. 激活所有CommandLineRunner beans。

直接使用main启动该类，SpringBoot便自动化配置了。

ps:即使是现在我依旧觉得这个实在是太厉害了。

该类的一些注解说明：

**SpringBootApplication**:开启组件扫描和自动配置。

**MapperScan**: mapper 接口类扫描包配置

代码如下：

```
1 @SpringBootApplication
2 @MapperScan("com.pancm.dao")
3 public class Application {
4 public static void main(String[] args)
5 {
6 // 启动嵌入式的 Tomcat 并初始化 Spring 环境及其各 Spring 组件
7 SpringApplication.run(Application.class, args);
8 System.out.println("程序正在运行...")
9 }
}
```

## 四、代码测试

代码编写完之后，我们进行代码的测试。

启动Application 之后，使用postman工具进行接口的测试。

postman的使用教程可以看这篇博客：

测试结果如下：

The screenshot shows two requests in Postman:

- POST Request:** URL: `http://localhost:8083/api/user`. Body: `{"id":1,"name":"xuwujing","age":18}`. Response status: 200 OK, Time: 85 ms, Size: 134 B. Body content: `true`.
- GET Request:** URL: `http://localhost:8083/api/user?userName=xuwujing`. Authorization: Inherit auth from parent. Response status: 200 OK, Time: 50 ms, Size: 165 B. Body content: `{ "id": 1, "name": "xuwujing", "age": 18 }`.

这里只使用了一个get和post测试，实际方法都测试过了。

项目放到github上面去了：

<https://github.com/xuwujing/springBoot>

如果喜欢本篇文章，欢迎[转发](#)、[点赞](#)。关注订阅号「Web项目聚集地」，回复「进群」即可进入无广告技术交流。

## 推荐阅读

[1. 什么时候进行分库分表？](#)

[2. 从 Java 程序员的角度理解加密](#)

[3. IDEA 中使用 Git 图文教程](#)

[4. 一文梳理 Redis 基础](#)

[5. 优化你的 Spring Boot](#)

[6. 数据库不使用外键的 9 个理由](#)



Web项目聚集地

微信扫描二维码，关注我的公众号

喜欢文章，点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 实战：SpringBoot & Restful API 构建示例

liuxiaopeng Java后端 2019-12-29

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | liuxiaopeng

链接 | [cnblogs.com/paddix/p/8215245.html](http://cnblogs.com/paddix/p/8215245.html)

在现在的开发流程中，为了最大程度实现前后端的分离，通常后端接口只提供数据接口，由前端通过Ajax请求从后端获取数据并进行渲染再展示给用户。我们用的最多的方式就是后端会返回给前端一个JSON字符串，前端解析JSON字符串生成JavaScript的对象，然后再做处理。

本文就来演示一下Spring boot如何实现这种模式，本文重点会讲解如何设计一个Restful的API，并通过Spring boot来实现相关的API。

不过，为了大家更好的了解Restful风格的API，我们先设计一个传统的数据返回接口，这样大家可以对比着来理解。

## 一、非Restful接口的支持

我们这里以文章列表为例，实现一个返回文章列表的接口，代码如下：

```
@Controller
@RequestMapping("/article")
public class ArticleController {

 @Autowired
 private ArticleService articleService;

 @RequestMapping("/list.json")
 @ResponseBody
 public List<Article> listArticles(String title, Integer pageSize, Integer pageNum) {
 if (pageSize == null) {
 pageSize = 10;
 }
 if (pageNum == null) {
 pageNum = 1;
 }
 int offset = (pageNum - 1) * pageSize;
 return articleService.getArticles(title, 1L, offset, pageSize);
 }
}
```

这个ArticleService的实现很简单，就是简单的封装了ArticleMapper的操作，ArticleService的实现类如下：

```

@Service
public class ArticleServiceImpl implements ArticleService {

 @Autowired
 private ArticleMapper articleMapper;

 @Override
 public Long saveArticle(@RequestBody Article article) {
 return articleMapper.insertArticle(article);
 }

 @Override
 public List<Article> getArticles(String title, Long userId, int offset, int pageSize) {
 Article article = new Article();
 article.setTitle(title);
 article.setUserId(userId);
 return articleMapper.queryArticlesByPage(article, offset, pageSize);
 }

 @Override
 public Article getById(Long id) {
 return articleMapper.queryById(id);
 }

 @Override
 public void updateArticle(Article article) {
 article.setUpdateTime(new Date());
 articleMapper.updateArticleById(article);
 }
}

```

运行Application.java这个类，然后访问：<http://localhost:8080/article/list.json>，就可以看到如下的结果：

```

{
 "id": 1,
 "title": "测试标题",
 "summary": "测试摘要",
 "createTime": "1514766300000",
 "publicTime": "1514886032000",
 "updateTime": "1514886032000",
 "userId": 1,
 "status": 2,
 "type": 0,
 "id": 2,
 "title": "测试标题2",
 "summary": "测试摘要2",
 "createTime": "1514885448000",
 "publicTime": "1514885448000",
 "updateTime": "1514885448000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 3,
 "title": "测试标题3",
 "summary": "测试摘要3",
 "createTime": "1514885452000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 4,
 "title": "测试标题4",
 "summary": "测试摘要4",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 5,
 "title": "测试标题5",
 "summary": "测试摘要5",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 6,
 "title": "测试标题6",
 "summary": "测试摘要6",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 7,
 "title": "测试标题7",
 "summary": "测试摘要7",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 8,
 "title": "测试标题8",
 "summary": "测试摘要8",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 9,
 "title": "测试标题9",
 "summary": "测试摘要9",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 10,
 "title": "测试标题10",
 "summary": "测试摘要10",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 11,
 "title": "测试标题11",
 "summary": "测试摘要11",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 12,
 "title": "测试标题12",
 "summary": "测试摘要12",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 13,
 "title": "测试标题13",
 "summary": "测试摘要13",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 14,
 "title": "测试标题14",
 "summary": "测试摘要14",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 15,
 "title": "测试标题15",
 "summary": "测试摘要15",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 16,
 "title": "测试标题16",
 "summary": "测试摘要16",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 17,
 "title": "测试标题17",
 "summary": "测试摘要17",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 18,
 "title": "测试标题18",
 "summary": "测试摘要18",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 19,
 "title": "测试标题19",
 "summary": "测试摘要19",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0,
 "id": 20,
 "title": "测试标题20",
 "summary": "测试摘要20",
 "createTime": "1514894512000",
 "publicTime": "1514894512000",
 "updateTime": "1514894512000",
 "userId": 1,
 "status": 1,
 "type": 0
}

```

ArticleServiceImpl这个类是一个很普通的类，只有一个Spring的注解@Service，标识为一个bean以便于通过Spring IoC容器来管理。我们再来看看ArticleController这个类，其实用过Spring MVC的人应该都熟悉这几个注解，这里简单解释一下：

- @Controller 标识一个类为控制器。
- @RequestMapping URL的映射。
- @ResponseBody 返回结果转换为JSON字符串。
- @RequestBody 表示接收JSON格式字符串参数。

通过这三个注解，我们就能轻松的实现通过URL给前端返回JSON格式数据的功能。不过大家肯定有点疑惑，这不都是Spring MVC的东西吗？跟Spring boot有什么关系？

其实Spring boot的作用就是为我们省去了配置的过程，其他功能确实都是Spring与Spring MVC来为我们提供的，大家应该记得Spring boot通过各种starter来为我们提供自动配置的服务，我们的工程里面之前引入过这个依赖：

```

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

```

这个是所有Spring boot的web工程都需要引入的jar包，也就是说只要是Spring boot的web的工程，都默认支持上述的功能。这

里我们进一步发现，通过Spring boot来开发web工程，确实为我们省了许多配置的工作。

## 二、Restful API设计

好了，我们现在再来看看如何实现Restful API。实际上Restful本身不是一项什么高深的技术，而只是一种编程风格，或者说是一种设计风格。

在传统的http接口设计中，我们一般只使用了get和post两个方法，然后用我们自己定义的词汇来表示不同的操作，比如上面查询文章的接口，我们定义了article/list.json来表示查询文章列表，可以通过get或者post方法来访问。而Restful API的设计则通过HTTP的方法来表示CRUD相关的操作。

因此，除了get和post方法外，还会用到其他的HTTP方法，如PUT、DELETE、HEAD等，通过不同的HTTP方法来表示不同含义的操作。下面是我设计的一组对文章的增删改查的Restful API：

接口URL	HTTP方法	接口说明
/article	POST	保存文章
/article/{id}	GET	查询文章列表
/article/{id}	DELETE	删除文章
/article/{id}	PUT	更新文章信息

这里可以看出，URL仅仅是标识资源的路劲，而具体的行为由HTTP方法来指定。

## 三、Restful API实现

现在我们再来看看如何实现上面的接口，其他就不多说，直接看代码：

```

@RestController
@RequestMapping("/rest")
public class ArticleRestController {

 @Autowired
 private ArticleService articleService;

 @RequestMapping(value = "/article", method = POST, produces = "application/json")
 public WebResponse<Map<String, Object>> saveArticle(@RequestBody Article article) {
 article.setUserId(1L);
 articleService.saveArticle(article);
 Map<String, Object> ret = new HashMap<>();
 ret.put("id", article.getId());
 WebResponse<Map<String, Object>> response = WebResponse.getSuccessResponse(ret);
 return response;
 }

 @RequestMapping(value = "/article/{id}", method = DELETE, produces = "application/json")
 public WebResponse<?> deleteArticle(@PathVariable Long id) {
 Article article = articleService.getById(id);
 article.setStatus(-1);
 articleService.updateArticle(article);
 WebResponse<Object> response = WebResponse.getSuccessResponse(null);
 return response;
 }

 @RequestMapping(value = "/article/{id}", method = PUT, produces = "application/json")
 public WebResponse<Object> updateArticle(@PathVariable Long id, @RequestBody Article article) {
 article.setId(id);
 articleService.updateArticle(article);
 WebResponse<Object> response = WebResponse.getSuccessResponse(null);
 return response;
 }

 @RequestMapping(value = "/article/{id}", method = GET, produces = "application/json")
 public WebResponse<Article> getArticle(@PathVariable Long id) {
 Article article = articleService.getById(id);
 WebResponse<Article> response = WebResponse.getSuccessResponse(article);
 return response;
 }
}

```

我们再来分析一下这段代码，这段代码和之前代码的区别在于：

- (1) 我们使用的是@RestController这个注解，而不是@Controller，不过这个注解同样不是Spring boot提供的，而是Spring MVC4中的提供的注解，表示一个支持Restful的控制器。
- (2) 这个类中有三个URL映射是相同的，即都是/article/{id}，这在@Controller标识的类中是不允许出现的。这里的可以通过method来进行区分，produces的作用是表示返回结果的类型是JSON。
- (3) @PathVariable这个注解，也是Spring MVC提供的，其作用是表示该变量的值是从访问路径中获取。

所以看来看去，这个代码还是跟Spring boot没太多的关系，Spring boot也仅仅是提供自动配置的功能，这也是Spring boot用起来很舒服的一个很重要的原因，因为它的侵入性非常非常小，你基本感觉不到它的存在。

#### 四、测试

代码写完了，怎么测试？除了GET的方法外，都不能直接通过浏览器来访问，当然，我们可以直接通过postman来发送各种http请求。不过我还是比较支持通过单元测试类来测试各个方法。这里我们就通过Junit来测试各个方法：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = Application.class)
public class ArticleControllerTest {

 @Autowired
 private ArticleRestController restController;

 private MockMvc mvc;

 @Before
 public void setUp() throws Exception {
 mvc = MockMvcBuilders.standaloneSetup(restController).build();
 }

 @Test
 public void testAddArticle() throws Exception {
 Article article = new Article();
 article.setTitle("测试文章000000");
 article.setType(1);
 article.setStatus(2);
 article.setSummary("这是一篇测试文章");
 Gson gosn = new Gson();
 RequestBuilder builder = MockMvcRequestBuilders
 .post("/rest/article")
 .accept(MediaType.APPLICATION_JSON)
 .contentType(MediaType.APPLICATION_JSON_UTF8)
 .content(gosn.toJson(article));

 MvcResult result = mvc.perform(builder).andReturn();
 System.out.println(result.getResponse().getContentAsString());
 }

 @Test
 public void testUpdateArticle() throws Exception {
 Article article = new Article();
 article.setTitle("更新测试文章");
 article.setType(1);
 article.setStatus(2);
 article.setSummary("这是一篇更新测试文章");
 Gson gosn = new Gson();
 RequestBuilder builder = MockMvcRequestBuilders
 .put("/rest/article/1")
 .accept(MediaType.APPLICATION_JSON)
 .contentType(MediaType.APPLICATION_JSON_UTF8)
 .content(gosn.toJson(article));

 MvcResult result = mvc.perform(builder).andReturn();
 }

 @Test
 public void testQueryArticle() throws Exception {
 RequestBuilder builder = MockMvcRequestBuilders
 .get("/rest/article/1")
 .accept(MediaType.APPLICATION_JSON)
 .contentType(MediaType.APPLICATION_JSON_UTF8);
 MvcResult result = mvc.perform(builder).andReturn();
 System.out.println(result.getResponse().getContentAsString());
 }

 @Test
 public void testDeleteArticle() throws Exception {
 RequestBuilder builder = MockMvcRequestBuilders
 .delete("/rest/article/1")
 .accept(MediaType.APPLICATION_JSON)
 .contentType(MediaType.APPLICATION_JSON_UTF8);
 }
}
```

```
.contentType(MediaType.APPLICATION_JSON_UTF8),
MvcResult result = mvc.perform(builder).andReturn();
}
}
```

执行结果这里就不给大家贴了，大家有兴趣的话可以自己实验一下。整个类要说明的点还是很少，主要这些东西都与Spring boot没关系，支持这些操作的原因还是上一篇文章中提到的引入对应的starter：

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

因为要执行HTTP请求，所以这里使用了MockMvc，ArticleRestController通过注入的方式实例化，不能直接new，否则ArticleRestController就不能通过Spring IoC容器来管理，因而其依赖的其他类也无法正常注入。通过MockMvc我们就可以轻松的实现HTTP的DELETE/PUT/POST等方法了。

## 五、总结

本文讲解了如果通过Spring boot来实现Restful的API，其实大部分东西都是Spring和Spring MVC提供的，Spring boot只是提供自动配置的功能。

但是，正是这种自动配置，为我们减少了很多的开发和维护工作，使我们能更加简单、高效的实现一个web工程，从而让我们能够更加专注于业务本身的开发，而不需要去关心框架的东西。

- END -

### 推荐阅读

1. Java 线程有哪些不太为人所知的技巧与用法？
2. 关于 CPU 的一些基本知识总结
3. 发布没有答案的面试题，都是耍流氓
4. 什么是一致性 Hash 算法？
5. 团队开发中 Git 最佳实践



声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 干货！没学过 Spring Boot ? 送你一份超详细的知识清单

CHENJII Java后端 2019-09-19



预警：本文非常长，建议先 mark 后看，预计阅读时间 81 min。

说明：前面有 4 个小节关于 Spring 的基础知识，分别是：IOC 容器、JavaConfig、事件监听、SpringFactoriesLoader 详解，它们占据了本文的大部分内容，虽然它们之间可能没有太多的联系，但这些知识对于理解 Spring Boot 的核心原理至关重要，如果你对 Spring 框架烂熟于心，完全可以跳过这 4 个小节。接下来便带你步入 Spring Boot 的学习之路，这个文章是由这些看似不相关的知识点组成，算是 Spring Boot 的一个知识清单。

在过去两三年的 Spring 生态圈，最让人兴奋的莫过于 Spring Boot 框架。或许从命名上就能看出这个框架的设计初衷：快速的启动 Spring 应用。因而 Spring Boot 应用本质上就是一个基于 Spring 框架的应用，它是 Spring 对“约定优先于配置”理念的最佳实践产物，它能够帮助开发者更快速高效地构建基于 Spring 生态圈的应用。[微信搜索 web\\_resource 关注后获取更多干货！](#)

那 Spring Boot 有何魔法？**自动配置、起步依赖、Actuator、命令行界面(CLI)**是 Spring Boot 最重要的 4 大核心特性，其中 CLI 是 Spring Boot 的可选特性，虽然它功能强大，但也引入了一套不太常规的开发模型，因而这个系列的文章仅关注其它 3 种特性。

如文章标题，本文是这个系列的第一部分，将为你打开 Spring Boot 的大门，重点为你剖析其启动流程以及自动配置实现原理。要掌握这部分核心内容，理解一些 Spring 框架的基础知识，将会让你事半功倍。

## 一、抛砖引玉：探索 Spring IoC 容器

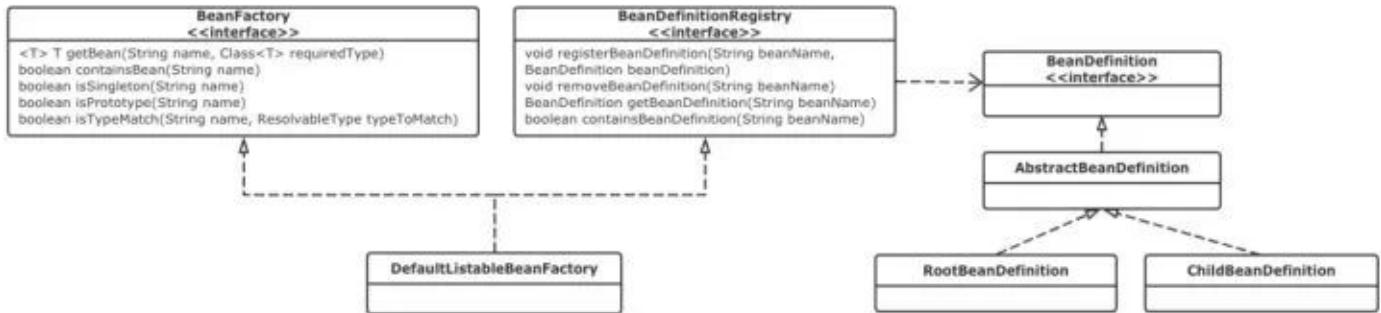
如果有看过 `SpringApplication.run()` 方法的源码，Spring Boot 冗长无比的启动流程一定会让你抓狂，透过现象看本质，`SpringApplication` 只是将一个典型的 Spring 应用的启动流程进行了扩展，因此，透彻理解 Spring 容器是打开 Spring Boot 大门的一把钥匙。

### 1.1、Spring IoC 容器

可以把 Spring IoC 容器比作一间餐馆，当你来到餐馆，通常会直接招呼服务员：点菜！至于菜的原料是什么？如何用原料把菜做出来？可能你根本就不关心。IoC 容器也是一样，你只需要告诉它需要某个 bean，它就把对应的实例（instance）扔给你，至于这个 bean 是否依赖其他组件，怎样完成它的初始化，根本就不需要你关心。

作为餐馆，想要做出菜肴，得知道菜的原料和菜谱，同样地，IoC 容器想要管理各个业务对象以及它们之间的依赖关系，需要通过某种途径来记录和管理这些信息。BeanDefinition 对象就承担了这个责任：容器中的每一个 bean 都会有一个对应的 BeanDefinition 实例，该实例负责保存 bean 对象的所有必要信息，包括 bean 对象的 class 类型、是否是抽象类、构造方法和参数、其它属性等等。当客户端向容器请求相对对象时，容器就会通过这些信息为客户端返回一个完整可用的 bean 实例。[微信搜索 web\\_resource 关注后获取更多干货！](#)

谱，BeanDefinitionRegistry和BeanFactory就是这份菜谱，BeanDefinitionRegistry 抽象出 bean 的注册逻辑，而 BeanFactory 则抽象出了 bean 的管理逻辑，而各个 BeanFactory 的实现类就具体承担了 bean 的注册以及管理工作。它们之间的关系就如下图：



DefaultListableBeanFactory作为一个比较通用的 BeanFactory 实现，它同时也实现了 BeanDefinitionRegistry 接口，因此它就承担了 Bean 的注册管理工作。从图中也可以看出，BeanFactory 接口中主要包含 `getBean`、`containBean`、`getType`、`getAliases` 等管理 bean 的方法，而 BeanDefinitionRegistry 接口则包含 `registerBeanDefinition`、`removeBeanDefinition`、`getBeanDefinition` 等注册管理 BeanDefinition 的方法。

下面通过一段简单的代码来模拟 BeanFactory 底层是如何工作的：

```
1 // 默认容器实现
2 DefaultListableBeanFactory beanRegistry = new DefaultListableBeanFactory();
3 // 根据业务对象构造相应的BeanDefinition
4 AbstractBeanDefinition definition = new RootBeanDefinition(Business.class, true)
5 ;
6 // 将bean定义注册到容器中
7 beanRegistry.registerBeanDefinition("beanName", definition);
8 // 如果有多个bean，还可以指定各个bean之间的依赖关系
//
```

```
1 // 然后可以从容器中获取这个bean的实例
2 // 注意：这里的beanRegistry其实实现了BeanFactory接口，所以可以强转，
3 // 单纯的BeanDefinitionRegistry是无法强制转换到BeanFactory类型的
4 BeanFactory container = (BeanFactory)beanRegistry;
5 Business business = (Business)container.getBean("beanName");
```

这段代码仅为了说明 BeanFactory 底层的大致工作流程，实际情况会更加复杂，比如 bean 之间的依赖关系可能定义在外部配置文件 (XML/Properties) 中、也可能是注解方式。Spring IoC 容器的整个工作流程大致可以分为两个阶段：

## 1. 容器启动阶段

容器启动时，会通过某种途径加载 ConfigurationMetaData。除了代码方式比较直接外，在大部分情况下，容器需要依赖某些工具类，比如：BeanDefinitionReader，BeanDefinitionReader 会对加载的 ConfigurationMetaData 进行解析和分析，并将分析后的信息组装为相对应的 BeanDefinition，最后把这些保存了 bean 定义的 BeanDefinition，注册到相应的 BeanDefinitionRegistry，这样容器的启动工作就完成了。这个阶段主要完成一些准备性工作，更侧重于 bean 对象管理信息的收集，当然一些验证性或者辅助性的工作也在这一阶段完

来看一个简单的例子吧,过往,所有的 bean 都定义在 XML 配置文件中,下面的代码将模拟 BeanFactory 如何从配置文件中加载 bean 的定义以及依赖关系:

```

1 // 通常为BeanDefinitionRegistry的实现类,这里以DefaultListableBeanFactory为例
2 BeanDefinitionRegistry beanRegistry = new DefaultListableBeanFactory();
3 // XmlBeanDefinitionReader实现了BeanDefinitionReader接口,用于解析XML文件
4 XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReaderImpl(beanRegistry);
5 // 加载配置文件
6 beanDefinitionReader.loadBeanDefinitions("classpath:spring-bean.xml");
7
8 // 从容器中获取bean实例
9 BeanFactory container = (BeanFactory)beanRegistry;
10 Business business = (Business)container.getBean("beanName");

```

## 2. Bean的实例化阶段

经过第一阶段,所有 bean 定义都通过 BeanDefinition 的方式注册到 BeanDefinitionRegistry 中,当某个请求通过容器的 getBean 方法请求某个对象,或者因为依赖关系容器需要隐式的调用 getBean 时,就会触发第二阶段的活动:容器会首先检查所请求的对象之前是否已经实例化完成。如果没有,则会根据注册的 BeanDefinition 所提供的信息实例化被请求对象,并为其注入依赖。当该对象装配完毕后,容器会立即将其返回给请求方法使用。

BeanFactory 只是 Spring IoC 容器的一种实现,如果没有特殊指定,它采用采用延迟初始化策略:只有当访问容器中的某个对象时,才对该对象进行初始化和依赖注入操作。而在实际场景下,我们更多的使用另外一种类型的容器: ApplicationContext, 它构建在 BeanFactory 之上,属于更高级的容器,除了具有 BeanFactory 的所有能力之外,还提供对事件监听机制以及国际化的支持等。它管理的 bean, 在容器启动时全部完成初始化和依赖注入操作。[微信搜索 web\\_resource 关注后获取更多干货](#)

### 1.2、Spring容器扩展机制

IoC 容器负责管理容器中所有bean的生命周期,而在 bean 生命周期的不同阶段, Spring 提供了不同的扩展点来改变 bean 的命运。在容器的启动阶段, BeanFactoryPostProcessor 允许我们在容器实例化相应对象之前,对注册到容器的 BeanDefinition 所保存的信息做一些额外的操作,比如修改 bean 定义的某些属性或者增加其他信息等。

如果要自定义扩展类,通常需要实现 org.springframework.beans.factory.config.BeanFactoryPostProcessor 接口,与此同时,因为容器中可能有多个 BeanFactoryPostProcessor,可能还需要实现 org.springframework.core.Ordered 接口,以保证 BeanFactoryPostProcessor 按照顺序执行。Spring 提供了为数不多的 BeanFactoryPostProcessor 实现,我们以 PropertyPlaceholderConfigurer 来说明其大致的工作流程。

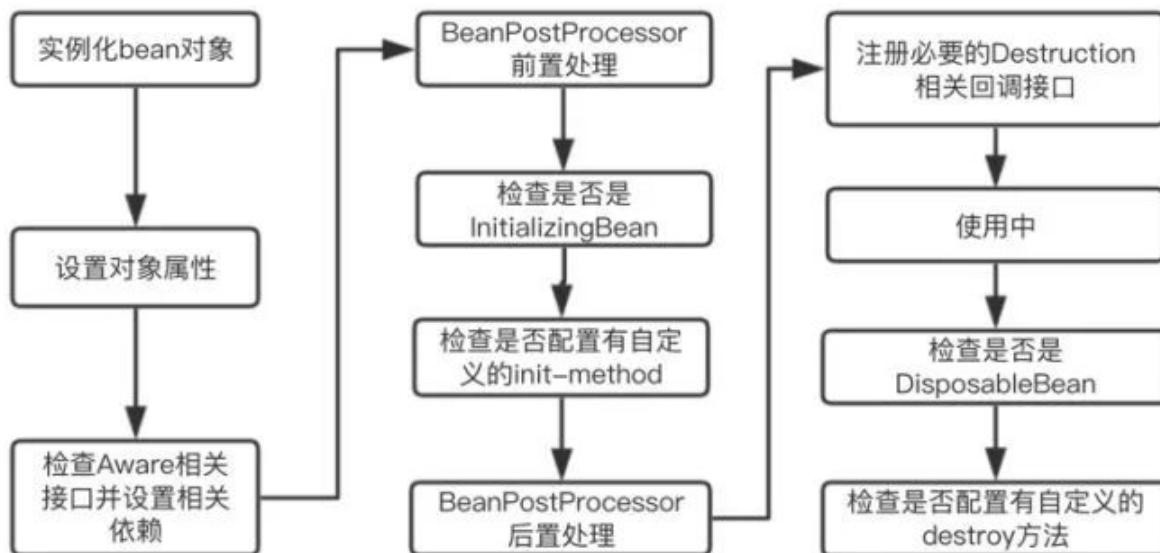
在 Spring 项目的 XML 配置文件中,经常可以看到许多配置项的值使用占位符,而将占位符所代表的值单独配置到独立的 properties 文件,这样可以将散落在不同 XML 文件中的配置集中管理,而且也方便运维根据不同的环境进行配置不同的值。这个非常实用的功能就是由 PropertyPlaceholderConfigurer 负责实现的。

根据前文,当 BeanFactory 在第一阶段加载完所有配置信息时,BeanFactory 中保存的对象的属性还是以占位符方式存在的,比如 \${jdbc.mysql.url}。当 PropertyPlaceholderConfigurer 作为 BeanFactoryPostProcessor 被应用时,它会使用 properties 配置文件中的值来替换相应的 BeanDefinition 中占位符所表示的属性值。当需要实例化 bean 时,bean 定义中的属性值就已经被替换成我们配置的值。当然其实现比上面描述的要复杂一些,这里仅说明其大致工作原理,更详细的实现可以参考其源码。[微信搜索 web\\_resource 关注后获取更多干货](#)

与之相似的，还有BeanPostProcessor，其存在于对象实例化阶段。跟BeanFactoryPostProcessor类似，它会处理容器内所有符合条件并且已经实例化后的对象。简单的对比，BeanFactoryPostProcessor处理bean的定义，而BeanPostProcessor则处理bean完成实例化后的对象。BeanPostProcessor定义了两个接口：

```
1 public interface BeanPostProcessor {
2 // 前置处
3 //
4 Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;
5 // 后置处
6 //
7 Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;
}
```

为了理解这两个方法执行的时机，简单的了解下bean的整个生命周期：



postProcessBeforeInitialization()方法与postProcessAfterInitialization()分别对应图中前置处理和后置处理两个步骤将执行的方法。这两个方法中都传入了bean对象实例的引用，为扩展容器的对象实例化过程提供了很大便利，在这儿几乎可以对传入的实例执行任何操作。注解、AOP等功能的实现均大量使用了BeanPostProcessor，比如有一个自定义注解，你完全可以实现BeanPostProcessor的接口，在其中判断bean对象的脑袋上是否有该注解，如果有，你可以对这个bean实例执行任何操作，想想是不是非常的简单？

再来看一个更常见的例子，在Spring中经常能够看到各种各样的Aware接口，其作用就是在对象实例化完成以后将Aware接口定义中规定的依赖注入到当前实例中。比如最常见的ApplicationContextAware接口，实现了这个接口的类都可以获取到一个ApplicationContext对象。当容器中每个对象的实例化过程走到BeanPostProcessor前置处理这一步时，容器会检测到之前注册到容器的ApplicationContextAwareProcessor，然后就会调用其postProcessBeforeInitialization()方法，检查并设置Aware相关依赖。看看代码吧，是不是很简单：

```
1 // 代码来自：org.springframework.context.support.ApplicationContextAwareProcessor
2 // 其postProcessBeforeInitialization方法调用了invokeAwareInterfaces方法
3 private void invokeAwareInterfaces(Object bean) {
```

```

4 if (bean instanceof EnvironmentAware) {
5 ((EnvironmentAware) bean).setEnvironment(this.applicationContext.getEnvironment());
6 }
7 if (bean instanceof ApplicationContextAware) {
8 ((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);
9 }
10 //
11 }

```

最后总结一下，本小节内容和你一起回顾了Spring容器的部分核心内容，限于篇幅不能写更多，但理解这部分内容，足以让您轻松理解Spring Boot的启动原理，如果在后续的学习过程中遇到一些晦涩难懂的知识，再回过头来看看Spring的核心知识，也许有意想不到的效果。也许Spring Boot的中文资料很少，但Spring的中文资料和书籍有太多太多，总有东西能给你启发。

## 二、夯实基础：JavaConfig与常见Annotation

### 2.1、JavaConfig

我们知道bean是Spring IOC中非常核心的概念，Spring容器负责bean的生命周期的管理。在最初，Spring使用XML配置文件的方式来描述bean的定义以及相互间的依赖关系，但随着Spring的发展，越来越多的人对这种方式表示不满，因为Spring项目的所有业务类均以bean的形式配置在XML文件中，造成了大量的XML文件，使项目变得复杂且难以管理。

后来，基于纯Java Annotation依赖注入框架 Guice出世，其性能明显优于采用XML方式的Spring，甚至有部分人认为，Guice可以完全取代Spring（Guice仅是一个轻量级IOC框架，取代Spring还差的挺远）。正是这样的危机感，促使Spring及社区推出并持续完善了JavaConfig子项目，它基于Java代码和Annotation注解来描述bean之间的依赖绑定关系。比如，下面是使用XML配置方式来描述bean的定义：

```

1 <bean id="bookService" class="cn.moondev.service.BookServiceImpl"></bean>

```

而基于JavaConfig的配置形式是这样的：

```

1 @Configuration
2 public class MoonBookConfiguration {
3
4 // 任何标志了@Bean的方法，其返回值将作为一个bean注册到Spring的IOC容器中
5 // 方法名默认成为该bean定义的id
6 @Bean
7 public BookService bookService() {
8
9 return new BookServiceImpl();
10 }
11 }

```

如果两个bean之间有依赖关系的话，在XML配置中应该是这样：

```

1 <bean id="bookService" class="cn.moondev.service.BookServiceImpl">
2 <property name="dependencyService" ref="dependencyService"/>
3

```

```

4 </bean>
5 <bean id="otherService" class="cn.moondev.service.OtherServiceImpl">
6 <property name="dependencyService" ref="dependencyService"/>
7 >
8 </bean>
9 <bean id="dependencyService" class="DependencyServiceImpl"/>

```

而在JavaConfig中则是这样：

```

1 @Configuration
2 public class MoonBookConfiguration {
3
4 // 如果一个bean依赖另一个bean，则直接调用对应JavaConfig类中依赖bean的创建方法即可
5 // 这里直接调用dependencyService()
6 @Bean
7 public BookService bookService() {
8
9 return new BookServiceImpl(dependencyService());
10 }
11
12 @Bean
13 public OtherService otherService() {
14
15 return new OtherServiceImpl(dependencyService());
16 }
17
18 @Bean
19 public DependencyService dependencyService() {
20
21 return new DependencyServiceImpl();
22 }
23
24 }}
```

你可能注意到这个示例中，有两个bean都依赖于dependencyService，也就是说当初始化bookService时会调用dependencyService()，在初始化otherService时也会调用dependencyService()，那么问题来了？这时候IOC容器中是有一个dependencyService实例还是两个？这个问题留着大家思考吧，这里不再赘述。

## 2.2、@ComponentScan

@ComponentScan注解对应XML配置形式中的<context:component-scan>元素，表示启用组件扫描，Spring会自动扫描所有通过注解配置的bean，然后将其注册到IOC容器中。我们可以通过basePackages等属性来指定@ComponentScan自动扫描的范围，如果不指定，默认从声明@ComponentScan所在类的package进行扫描。正因为如此，SpringBoot的启动类都默认在src/main/java下。

## 2.3、@Import

@Import注解用于导入配置类，举个简单的例子：

```

1 @Configuration
2 public class MoonBookConfiguration {
3 @Bean
4
5 public BookService bookService() {
6 return new BookServiceImpl(dependencyService());
7 }
8
9 @Bean
10 public OtherService otherService() {
11 return new OtherServiceImpl(dependencyService());
12 }
13 }
```

```
4 public BookService bookService()
5 {
6 return new BookServiceImpl();
7 }
}
```

现在有另外一个配置类，比如：MoonUserConfiguration，这个配置类中有一个bean依赖于MoonBookConfiguration中的bookService，如何将这两个bean组合在一起？借助@Import即可：

```
1 @Configuration
2 // 可以同时导入多个配置类，比如：@Import({A.class,B.class})
3 @Import(MoonBookConfiguration.class)
4 public class MoonUserConfiguration {
5 @Bean
6 public UserService userService(BookService bookService)
7 {
8 return new BookServiceImpl(bookService);
9 }
}
```

需要注意的是，在4.2之前，@Import注解只支持导入配置类，但是在4.2之后，它支持导入普通类，并将这个类作为一个bean的定义注册到IOC容器中。

## 2.4、@Conditional

@Conditional注解表示在满足某种条件后才初始化一个bean或者启用某些配置。它一般用在由@Component、@Service、@Configuration等注解标识的类上面，或者由@Bean标记的方法上。如果一个@Configuration类标记了@Conditional，则该类中所有标识了@Bean的方法和@Import注解导入的相关类将遵从这些条件。[微信搜索 web\\_resource 关注后获取更多干货](#)

在Spring里可以很方便的编写你自己的条件类，所要做的就是实现Condition接口，并覆盖它的matches()方法。举个例子，下面的简单条件类表示只有在Classpath里存在JdbcTemplate类时才生效：

```
1 public class JdbcTemplateCondition implements Condition {
2
3 @Override
4 public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata annotatedTypeMetadata)
5 {
6 try
7 {
8 conditionContext.getClassLoader().loadClass("org.springframework.jdbc.core.JdbcTemplate")
9 ;
10 return true
11 ;
12 } catch (ClassNotFoundException e) {
13 e.printStackTrace();
14 }
15 return false
16 ;
17 }
```

```
}
```

当你用Java来声明bean的时候，可以使用这个自定义条件类：

```
1 @Conditional(JdbcTemplateCondition.class)
2 @Service
3 public MyService service() {
4
5 }
```

这个例子中只有当JdbcTemplateCondition类的条件成立时才会创建MyService这个bean。也就是说MyService这个bean的创建条件是classpath里面包含JdbcTemplate，否则这个bean的声明就会被忽略掉。

SpringBoot定义了很多有趣的条件，并把他们运用到了配置类上，这些配置类构成了SpringBoot的自动配置的基础。SpringBoot运用条件化配置的方法是：定义多个特殊的条件化注解，并将它们用到配置类上。下面列出了SpringBoot提供的部分条件化注解：

条件化注解	配置生效条件
@ConditionalOnBean	配置了某个特定bean
@ConditionalOnMissingBean	没有配置特定的bean
@ConditionalOnClass	Classpath里有指定的类
@ConditionalOnMissingClass	Classpath里没有指定的类
@ConditionalOnExpression	给定的Spring Expression Language表达式计算结果为true
@ConditionalOnJava	Java的版本匹配特定指或者一个范围值
@ConditionalOnProperty	指定的配置属性要有一个明确的值
@ConditionalOnResource	Classpath里有指定的资源
@ConditionalOnWebApplication	这是一个Web应用程序
@ConditionalOnNotWebApplication	这不是一个Web应用程序

## 2.5、@ConfigurationProperties与@EnableConfigurationProperties

当某些属性的值需要配置的时候，我们一般会在application.properties文件中新建配置项，然后在bean中使用@Value注解来获取配置的值，比如下面配置数据源的代码。

```
1 // jdbc config
2 jdbc.mysql.url=jdbc:mysql://localhost:3306/sampledb
3 jdbc.mysql.username=root
4 jdbc.mysql.password=123456
5
```

```
1 // 配置数据源
2 @Configuration
3 public class HikariDataSourceConfiguration {
4
5 @Value("jdbc.mysql.url"
6 @Value("jdbc.mysql.username")
7 @Value("jdbc.mysql.password")}
```

```

6
7 public String url;
8 @Value("jdbc.mysql.username")
9)
10 public String user;
11 @Value("jdbc.mysql.password")
12)
13 public String password;
14
15 @Bean
16
17 public HikariDataSource dataSource() {
18 HikariConfig hikariConfig = new HikariConfig();
19 hikariConfig.setJdbcUrl(url);
20 hikariConfig.setUsername(user);
21 hikariConfig.setPassword(password);
22 // 省略部分代码
23
24 return new HikariDataSource(hikariConfig);
25 }
26

```

使用@Value注解注入的属性通常都比较简单，如果同一个配置在多个地方使用，也存在不方便维护的问题（考虑下，如果有几十个地方在使用某个配置，而现在你想改下名字，你改怎么做？）。对于更为复杂的配置，Spring Boot提供了更优雅的实现方式，那就是@ConfigurationProperties注解。我们可以通过下面的方式来改写上面的代码：

```

1 @Component
2 // 还可以通过@PropertySource("classpath:jdbc.properties")来指定配置文件
3 @ConfigurationProperties("jdbc.mysql")
4 // 前缀=jdbc.mysql，会在配置文件中寻找jdbc.mysql.*的配置项
5 public class JdbcConfig {
6 public String url;
7 public String username;
8 public String password;
9 }
10

```

```

1 @Configuration
2 public class HikariDataSourceConfiguration {
3
4 @Autowired
5 public JdbcConfig config;
6
7 @Bean
8 public HikariDataSource dataSource()
9 {
10 HikariConfig hikariConfig = new HikariConfig();
11 hikariConfig.setJdbcUrl(config.url);
12 hikariConfig.setUsername(config.username);
13 hikariConfig.setPassword(config.password);
14 // 省略部分代码
15 }

```

```
16 return new HikariDataSource(hikariConfig);
17 }
18 }
```

@ConfigurationProperties对于更为复杂的配置，处理起来也是得心应手，比如有如下配置文件：

```
1 #App
2 app.menus[0].title=Home
3 app.menus[0].name=Home
4 app.menus[0].path=/
5 app.menus[1].title=Login
6 app.menus[1].name=Login
7 app.menus[1].path=/login
8
9 app.compiler.timeout=5
10 app.compiler.output-folder=/temp/
11
12 app.error=/error/
```

可以定义如下配置类来接收这些属性

```
1 @Component
2 @ConfigurationProperties("app")
3 public class AppProperties {
4
5 public String error;
6 public List<Menu> menus = new ArrayList<>();
7 public Compiler compiler = new Compiler();
8
9 public static class Menu {
10 public String name
11 ;
12 public String path
13 ;
14 public String title
15 ;
16 }
17
18 public static class Compiler {
19 public String timeout;
20 public String outputFolder;
21 }
22 }
```

@EnableConfigurationProperties注解表示对@ConfigurationProperties的内嵌支持，默认会将对应Properties Class作为bean注入的IOC容器中，即在相应的Properties类上不用加@Component注解。

### 三、削铁如泥：SpringFactoriesLoader详解

JVM提供了3种类加载器：BootstrapClassLoader、ExtClassLoader、AppClassLoader分别加载Java核心类库、扩展类库以及应用的类路径(CLASSPATH)下的类库。JVM通过双亲委派模型进行类的加载，我们也可以通过继承java.lang.classloader实现自己的类加载器。

何为双亲委派模型？当一个类加载器收到类加载任务时，会先交给自己的父加载器去完成，因此最终加载任务都会传递到最顶层的BootstrapClassLoader，只有当父加载器无法完成加载任务时，才会尝试自己来加载。[微信搜索 web\\_resource 关注后获取更多干货](#)

采用双亲委派模型的一个好处是保证使用不同类加载器最终得到的都是同一个对象，这样就可以保证Java核心库的类型安全，比如，加载位于rt.jar包中的java.lang.Object类，不管是哪个加载器加载这个类，最终都是委托给顶层的BootstrapClassLoader来加载的，这样就可以保证任何的类加载器最终得到的都是同样一个Object对象。查看ClassLoader的源码，对双亲委派模型会有更直观的认识：

```
1 protected Class<?> loadClass(String name, boolean resolve) {
2 synchronized (getClassLoadingLock(name)) {
3 // 首先，检查该类是否已经被加载，如果从JVM缓存中找到该类，则直接返回
4 Class<?> c = findLoadedClass(name);
5 if (c == null) {
6 try {
7 // 遵循双亲委派的模型，首先会通过递归从父加载器开始找
8 ,
9 // 直到父类加载器是BootstrapClassLoader为止
10 if (parent != null) {
11 c = parent.loadClass(name, false)
12 ;
13 } else
14 {
15 c = findBootstrapClassOrNull(name);
16 }
17 } catch (ClassNotFoundException e) {
18 }
19 if (c == null)
20 {
21 // 如果还找不到，尝试通过findClass方法去寻找
22 // findClass是留给开发者自己实现的，也就是说
23 // 自定义类加载器时，重写此方法即
24 可
25 c = findClass(name);
26 }
27 }
28}
if (resolve) {
 resolveClass(c)
;
}
}
return c
;
}
```

但双亲委派模型并不能解决所有的类加载器问题，比如，Java 提供了很多服务提供者接口（ ServiceProviderInterface，SPI），允许第三方为这些接口提供实现。常见的 SPI 有 JDBC、JNDI、JAXP 等，这些SPI的接口由核心类库提供，却由第三方实现，这样就存在一个问题：SPI 的接口是 Java 核心库的一部分，是由BootstrapClassLoader加载的；SPI实现的Java类一般是由AppClassLoader来加载的。

BootstrapClassLoader是无法找到 SPI 的实现类的，因为它只加载Java的核心库。它也不能代理给AppClassLoader，因为它是最顶层的类加载器。也就是说，双亲委派模型并不能解决这个问题。

线程上下文类加载器(ContextClassLoader)正好解决了这个问题。从名称上看，可能会误解为它是一种新的类加载器，实际上，它仅仅是Thread类的一个变量而已，可以通过setContextClassLoader(ClassLoadercl)和getContextClassLoader()来设置和获取该对象。如果不做任何的设置，Java应用的线程的上下文类加载器默认就是AppClassLoader。在核心类库使用SPI接口时，传递的类加载器使用线程上下文类加载器，就可以成功的加载到SPI实现的类。线程上下文类加载器在很多SPI的实现中都会用到。但在JDBC中，你可能会看到一种更直接的实现方式，比如，JDBC驱动管理java.sql.Driver中的loadInitialDrivers()方法中，你可以直接看到JDK是如何加载驱动的：

```
1 for (String aDriver : driversList) {
2 try {
3 // 直接使用AppClassLoader
4 Class.forName(aDriver, true, ClassLoader.getSystemClassLoader());
5 } catch (Exception ex) {
6 println("DriverManager.Initialize: load failed: " + ex)
7 }
8 }
}
```

其实讲解线程上下文类加载器，最主要是让大家在看到Thread.currentThread().getClassLoader()和Thread.currentThread().getContextClassLoader()时不会一脸懵逼，这两者除了在许多底层框架中取得的ClassLoader可能会有所不同外，其他大多数业务场景下都是一样的，大家只要知道它是为了解决什么问题而存在的即可。

类加载器除了加载class外，还有一个非常重要功能，就是加载资源，它可以从jar包中读取任何资源文件，比如，ClassLoader.getResources(Stringname)方法就是用于读取jar包中的资源文件，其代码如下：

```
1 public Enumeration<URL> getResources(String name) throws IOException {
2 Enumeration<URL>[] tmp = (Enumeration<URL>[]) new Enumeration<?>[2]
3 ;
4 if (parent != null)
5 {
6 tmp[0] = parent.getResources(name);
7 } else
8 {
9 tmp[0] = getBootstrapResources(name);
10 }
11 tmp[1] = findResources(name);
12 return new CompoundEnumeration<>(tmp);
13 }
```

是不是觉得有点眼熟，不错，它的逻辑其实跟类加载的逻辑是一样的，首先判断父类加载器是否为空，不为空则委托父类加载器执行资源查找任务，直到BootstrapClassLoader，最后才轮到自己查找。而不同的类加载器负责扫描不同路径下的jar包，就如同加载class一样，最后会扫描所有的jar包，找到符合条件的资源文件。[微信搜索 web\\_resource 关注后获取更多干货](#)

类加载器的findResources(name)方法会遍历其负责加载的所有jar包，找到jar包中名称为name的资源文件，这里的资源可以是任何文件，甚至是.class文件，比如下面的示例，用于查找Array.class文件：

```
1 // 寻找Array.class文件
2 public static void main(String[] args) throws Exception{
3 // Array.class的完整路径
4 String name = "java/sql/Array.class"
5 ;
6 Enumeration<URL> urls = Thread.currentThread().getContextClassLoader().getResources(name);
7 while (urls.hasMoreElements()) {
8 URL url = urls.nextElement();
9 System.out.println(url.toString());
10 }
11 }
```

运行后可以得到如下结果：

```
1 $JAVA_HOME/jre/lib/rt.jar!/java/sql/Array.class
```

根据资源文件的URL，可以构造相应的文件来读取资源内容。

看到这里，你可能会感到挺奇怪的，你不是要详解SpringFactoriesLoader吗？上来讲了一堆ClassLoader是几个意思？看下它的源码你就知道了：

```
1 public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories"
2 ;
3 // spring.factories文件的格式为：key=value1,value2,value3
4 // 从所有的jar包中找到META-INF/spring.factories文件
5 // 然后从文件中解析出key=factoryClass类名称的所有value值
6 public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {
7 String factoryClassName = factoryClass.getName();
8 // 取得资源文件的URL
9 Enumeration<URL> urls = (classLoader != null ? classLoader.getResources(FACTORIES_RESOURCE_LOCATION) : ClassLo
10 List<String> result = new ArrayList<String>()
11 ;
12 // 遍历所有的URL
13 while (urls.hasMoreElements()) {
14 URL url = urls.nextElement();
15 // 根据资源文件URL解析properties文件
16 Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource(url));
17 String factoryClassNames = properties.getProperty(factoryClassName);
18 // 组装数据，并返
19 }
20 result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(factoryClassNames)));
21 }
22 return result;
23 }
```

有了前面关于ClassLoader的知识，再来理解这段代码，是不是感觉豁然开朗：从CLASSPATH下的每个Jar包中搜寻所有META-INF/spring.factories配置文件，然后将解析properties文件，找到指定名称的配置后返回。需要注意的是，其实这里不仅仅是会去ClassPath路径下查找，会扫描所有路径下的Jar包，只不过这个文件只会在Classpath下的jar包中。来简单看下spring.factories文件的内容吧：

```
1 // 来自 org.springframework.boot.autoconfigure下的META-INF/spring.factories
2 // EnableAutoConfiguration后文会讲到，它用于开启Spring Boot自动配置功能
3 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
4 org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\ \
5 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\ \
6 org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration\
```

执行loadFactoryNames(EnableAutoConfiguration.class,classLoader)后，得到对应的一组@Configuration类，我们就可以通过反射实例化这些类然后注入到IOC容器中，最后容器里就有了一系列标注了@Configuration的JavaConfig形式的配置类。

这就是SpringFactoriesLoader，它本质上属于Spring框架私有的一种扩展方案，类似于SPI，Spring Boot在Spring基础上的很多核心功能都是基于此，希望大家可以理解。[微信搜索 web\\_resource 关注后获取更多干货](#)

#### 四、另一件武器：Spring容器的事件监听机制

过去，事件监听机制多用于图形界面编程，比如：[点击按钮、在文本框输入内容等操作被称为事件](#)，而当事件触发时，应用程序作出一定的响应则表示应用监听了这个事件，而在服务器端，事件的监听机制更多的用于异步通知以及监控和异常处理。Java提供了实现事件监听机制的两个基础类：自定义事件类型扩展自java.util.EventObject、事件的监听器扩展自java.util.EventListener。来看一个简单的实例：简单的监控一个方法的耗时。

首先定义事件类型，通常的做法是扩展EventObject，随着事件的发生，相应状态通常都封装在此类中：

```
1 public class MethodMonitorEvent extends EventObject {
2 // 时间戳，用于记录方法开始执行的时
3 间
4 public long timestamp;
5
6 public MethodMonitorEvent(Object source)
7 {
8 super(source);
9 }
10 }
```

事件发布之后，相应的监听器即可对该类型的事件进行处理，我们可以在方法开始执行之前发布一个begin事件，在方法执行结束之后发布一个end事件，相应地，事件监听器需要提供方法对这两种情况下接收到的事件进行处理：

```
1 // 1、定义事件监听接口
2 public interface MethodMonitorEventListener extends EventListener {
3 // 处理方法执行之前发布的事
4 件
5 public void onMethodBegin(MethodMonitorEvent event)
6 ;
7 // 处理方法结束时发布的事
```

```

8 件
9 public void onMethodEnd(MethodMonitorEvent event)
10 ;
11 }
12 // 2、事件监听接口的实现：如何处理
13 public class AbstractMethodMonitorEventListener implements MethodMonitorEventListener {
14
15 @Override
16 public void onMethodBegin(MethodMonitorEvent event)
17 {
18 // 记录方法开始执行时的时
19 间
20 event.timestamp = System.currentTimeMillis();
21 }
22
23 @Override
24 public void onMethodEnd(MethodMonitorEvent event)
25 {
26 // 计算方法耗
27 时
28 long duration = System.currentTimeMillis() - event.timestamp
29 ;
30 System.out.println("耗时：" + duration);
31 }
32 }

```

事件监听器接口针对不同的事件发布实际提供相应的处理方法定义，最重要的是，其方法只接收MethodMonitorEvent参数，说明这个监听器类只负责监听器对应的事件并进行处理。有了事件和监听器，剩下的就是发布事件，然后让相应的监听器监听并处理。通常情况，我们都会有一个事件发布者，它本身作为事件源，在合适的时机，将相应的事件发布给对应的事件监听器：

```

1 public class MethodMonitorEventPublisher {
2
3 private List<MethodMonitorEventListener> listeners = new ArrayList<MethodMonitorEventListener>();
4
5 public void methodMonitor()
6 {
7 MethodMonitorEvent eventObject = new MethodMonitorEvent(this)
8 ;
9 publishEvent("begin",eventObject);
10 // 模拟方法执行：休眠5秒
11 钟
12 TimeUnit.SECONDS.sleep(5)
13 ;
14 publishEvent("end",eventObject);
15
16 }
17
18 private void publishEvent(String status,MethodMonitorEvent event)
19 {
20 // 避免在事件处理期间，监听器被移除，这里为了安全做一个复制操
21 作
22 List<MethodMonitorEventListener> copyListeners = new ArrayList<MethodMonitorEventListener>(listeners);

```

```

23 for (MethodMonitorEventListener listener : copyListeners) {
24 if ("begin".equals(status)) {
25 listener.onMethodBegin(event)
26 } else
27 {
28 listener.onMethodEnd(event)
29 }
30 }
31 }
32 }
33 }
34

public static void main(String[] args) {
 MethodMonitorEventPublisher publisher = new MethodMonitorEventPublisher();
 publisher.addEventListerner(new AbstractMethodMonitorEventListener());
 publisher.methodMonitor();
}
// 省略实
现
public void addEventListerner(MethodMonitorEventListener listener) {
}
public void removeEventListerner(MethodMonitorEventListener listener) {
}
public void removeAllListeners() {
}

```

对于事件发布者（事件源）通常需要关注两点：

1. 在合适的时机发布事件。此例中的methodMonitor()方法是事件发布的源头，其在方法执行之前和结束之后两个时间点发布MethodMonitorEvent事件，每个时间点发布的事件都会传给相应的监听器进行处理。在具体实现时需要注意的是，事件发布是顺序执行，为了不影响处理性能，事件监听器的处理逻辑应尽量简单。
2. 事件监听器的管理。publisher类中提供了事件监听器的注册与移除方法，这样客户端可以根据实际情况决定是否需要注册新的监听器或者移除某个监听器。如果这里没有提供remove方法，那么注册的监听器示例将一直被MethodMonitorEventPublisher引用，即使已经废弃不用了，也依然在发布者的监听器列表中，这会导致隐性的内存泄漏。

## Spring容器内的事件监听机制

Spring的ApplicationContext容器内部中的所有事件类型均继承自org.springframework.context.ApplicationEvent，容器中的所有监听器都实现org.springframework.context.ApplicationListener接口，并且以bean的形式注册在容器中。一旦在容器内发布ApplicationEvent及其子类型事件，注册到容器的ApplicationListener就会对这些事件进行处理。

你应该已经猜到是怎么回事了。

ApplicationEvent继承自EventObject，Spring提供了一些默认的实现，比如：ContextClosedEvent表示容器在即将关闭时发布的事件类型，ContextRefreshedEvent表示容器在初始化或者刷新的时候发布的事件类型.....

容器内部使用ApplicationListener作为事件监听器接口定义，它继承自EventListener。ApplicationContext容器在启动时，会自动识别并加载EventListener类型的bean，一旦容器内有事件发布，将通知这些注册到容器的EventListener。

ApplicationContext接口继承了ApplicationEventPublisher接口，该接口提供了voidpublishEvent(ApplicationEventevent)方法定义，不难看出，ApplicationContext容器担当的就是事件发布者的角色。如果有兴趣可以查看AbstractApplicationContext.publishEvent(ApplicationEventevent)方法的源码： ApplicationContext将事件的发布以及监听器的管理工作委托给ApplicationEventMulticaster接口的实现类。在容器启动时，会检查容器内是否存在名为applicationEventMulticaster的ApplicationEventMulticaster对象实例。如果有就使用其提供的实现，没有就默认初始化一个SimpleApplicationEventMulticaster作为实现。

最后，如果我们业务需要在容器内部发布事件，只需要为其注入ApplicationEventPublisher依赖即可：实现ApplicationEventPublisherAware接口或者ApplicationContextAware接口(Aware接口相关内容请回顾上文)。

## 五、出神入化：揭秘自动配置原理

典型的Spring Boot应用的启动类一般均位于src/main/java根路径下，比如MoonApplication类：

```
1 @SpringBootApplication
2 public class MoonApplication {
3
4 public static void main(String[] args)
5 {
6 SpringApplication.run(MoonApplication.class, args);
7 }
8 }
```

其中@SpringBootApplication开启组件扫描和自动配置，而SpringApplication.run则负责启动引导应用程序。@SpringBootApplication是一个复合Annotation，它将三个有用的注解组合在一起：

```
1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Documented
4 @Inherited
5 @SpringBootConfiguration
6 @EnableAutoConfiguration
7 @ComponentScan(excludeFilters = {
8 @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class)
9 ,
10 @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
11)
12 public @interface SpringBootApplication {
13 //
14 }
```

@SpringBootConfiguration就是@Configuration，它是Spring框架的注解，标明该类是一个JavaConfig配置类。

而@ComponentScan启用组件扫描，前文已经详细讲解过，这里着重关注@EnableAutoConfiguration。

@EnableAutoConfiguration注解表示开启Spring Boot自动配置功能，Spring Boot会根据应用的依赖、自定义的bean、classpath下有没有某个类 等等因素来猜测你需要的bean，然后注册到IOC容器中。那么@EnableAutoConfiguration是如何推算出你的需求？首先看下它的定义：

```

1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Documented
4 @Inherited
5 @AutoConfigurationPackage
6 @Import(EnableAutoConfigurationImportSelector.class)
7 public @interface EnableAutoConfiguration {
8 //
9 }

```

你的关注点应该在@Import(EnableAutoConfigurationImportSelector.class)上了，前文说过，@Import注解用于导入类，并将这个类作为一个bean的定义注册到容器中，这里它将把EnableAutoConfigurationImportSelector作为bean注入到容器中，而这个类会将所有符合条件的@Configuration配置都加载到容器中，看看它的代码：

```

1 public String[] selectImports(AnnotationMetadata annotationMetadata) {
2 // 省略了大部分代码，保留一句核心代码
3 // 注意：SpringBoot最近版本中，这句代码被封装在一个单独的方法中
4 // SpringFactoriesLoader相关知识请参考前文
5 List<String> factories = new ArrayList<String>(new LinkedHashSet<String>
6 (
7 SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration.class, this.beanClassLoader)));
}

```

这个类会扫描所有的jar包，将所有符合条件的@Configuration配置类注入的容器中，何为符合条件，看看META-INF/spring.factories的文件内容：

```

1 // 来自 org.springframework.boot.autoconfigure下的META-INF/spring.factories
2 // 配置的key = EnableAutoConfiguration，与代码中一致
3 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
4 org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
5 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
6 org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration\
7

```

以DataSourceAutoConfiguration为例，看看Spring Boot是如何自动配置的：

```

1 @Configuration
2 @ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
3 @EnableConfigurationProperties(DataSourceProperties.class)
4 @Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class })
5 public class DataSourceAutoConfiguration {
6 }

```

分别说一说：

@ConditionalOnClass({DataSource.class, EmbeddedDatabaseType.class})：当Classpath中存在DataSource或者

EmbeddedDatabaseType类时才启用这个配置，否则这个配置将被忽略。

@EnableConfigurationProperties(dataSourceProperties.class)：将DataSource的默认配置类注入到IOC容器

中，DataSourceProperties定义为：

```
1 // 提供对datasource配置信息的支持，所有的配置前缀为：spring.datasource
2 @ConfigurationProperties(prefix = "spring.datasource"
3)
4 public class DataSourceProperties {
5 private ClassLoader classLoader;
6 private Environment environment;
7 private String name = "testdb"
8 ;
9
10 }
```

@Import({Registrar.class,DataSourcePoolMetadataProvidersConfiguration.class})：导入其他额外的配置，就以

DataSourcePoolMetadataProvidersConfiguration为例吧。微信搜索 web\_resource 关注后获取更多干货

```
1 @Configuration
2 public class DataSourcePoolMetadataProvidersConfiguration {
3
4 @Configuration
5 @ConditionalOnClass(org.apache.tomcat.jdbc.pool.DataSource.class)
6 static class TomcatDataSourcePoolMetadataProviderConfiguration
7 {
8 @Bea
9 n
10 public DataSourcePoolMetadataProvider tomcatPoolDataSourceMetadataProvider()
11 {
12
13 }
14 }
15 }
```

DataSourcePoolMetadataProvidersConfiguration是数据库连接池提供者的一个配置类，即Classpath中存

在org.apache.tomcat.jdbc.pool.DataSource.class，则使用tomcat-jdbc连接池，如果Classpath中存在HikariDataSource.class则使用Hikari连接池。

这里仅描述了DataSourceAutoConfiguration的冰山一角，但足以说明Spring Boot如何利用条件话配置来实现自动配置的。回顾一下，@EnableAutoConfiguration中导入了EnableAutoConfigurationImportSelector类，而这个类的selectImports()通过SpringFactoriesLoader得到了大量的配置类，而每一个配置类则根据条件化配置来做出决策，以实现自动配置。

整个流程很清晰，但漏了一个大问题：EnableAutoConfigurationImportSelector.selectImports()是何时执行的？其实这个方法会在容器启动过程中执行：AbstractApplicationContext.refresh()，更多的细节在下一小节中说明。

## 六、启动引导：Spring Boot应用启动的秘密

## 6.1 SpringApplication初始化

SpringBoot整个启动流程分为两个步骤：初始化一个SpringApplication对象、执行该对象的run方法。看下SpringApplication的初始化流程，SpringApplication的构造方法中调用initialize(Object[] sources)方法，其代码如下：

```
1 private void initialize(Object[] sources) {
2 if (sources != null && sources.length > 0)
3 {
4 this.sources.addAll(Arrays.asList(sources));
5 }
6 // 判断是否是Web项目
7 this.webEnvironment = deduceWebEnvironment();
8 setInitializers((Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class));
9 setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
10 // 找到入口
11 类
12 this.mainApplicationClass = deduceMainApplicationClass();
13 }
```

初始化流程中最重要的就是通过SpringFactoriesLoader找到spring.factories文件中配置的ApplicationContextInitializer和ApplicationListener两个接口的实现类名称，以便后期构造相应的实例。ApplicationContextInitializer的主要目的是在ConfigurableApplicationContext做refresh之前，对ConfigurableApplicationContext实例做进一步的设置或处理。ConfigurableApplicationContext继承自ApplicationContext，其主要提供了对ApplicationContext进行设置的能力。

实现一个ApplicationContextInitializer非常简单，因为它只有一个方法，但大多数情况下我们没有必要自定义一个ApplicationContextInitializer，即便是Spring Boot框架，它默认也只是注册了两个实现，毕竟Spring的容器已经非常成熟和稳定，你没有必要来改变它。

而ApplicationListener的目的就没什么好说的了，它是Spring框架对Java事件监听机制的一种框架实现，具体内容在前文Spring事件监听机制这个小节有详细讲解。这里主要说说，如果你想为Spring Boot应用添加监听器，该如何实现？

Spring Boot提供两种方式来添加自定义监听器：

1. 通过SpringApplication.addListeners(ApplicationListener<?>...listeners)或者SpringApplication.setListeners(Collection<? extends ApplicationListener<?>>listeners)两个方法来添加一个或者多个自定义监听器
2. 既然SpringApplication的初始化流程中已经从spring.factories中获取到ApplicationListener的实现类，那么我们直接在自己的jar包的META-INF/spring.factories文件中新增配置即可：

```
org.springframework.context.ApplicationListener=\
cn.moondev.listeners.xxxxListener\
```

关于SpringApplication的初始化，我们就说这么多。

## 6.2 Spring Boot启动流程

Spring Boot应用的整个启动流程都封装在SpringApplication.run方法中，其整个流程真的是太长太长了，但本质上就是在Spring容器启

动的基础上做了大量的扩展，按照这个思路来看看源码：

```
1 public ConfigurableApplicationContext run(String... args) {
2 Stopwatch stopWatch = new Stopwatch();
3 stopWatch.start();
4 ConfigurableApplicationContext context = null
5 ;
6 FailureAnalyzers analyzers = null
7 ;
8 configureHeadlessProperty();
9 //
10 ①
11 SpringApplicationRunListeners listeners = getRunListeners(args);
12 listeners.starting();
13 try
14 {
15 //
16 ②
17 ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
18 ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);
19 //
20 ③
21 Banner printedBanner = printBanner(environment);
22 //
23 ④
24 context = createApplicationContext();
25 //
26 ⑤
27 analyzers = new FailureAnalyzers(context);
28 //
29 ⑥
30 prepareContext(context, environment, listeners, applicationArguments, printedBanner);
31 //
32 ⑦
33 refreshContext(context);
34 //
35 ⑧
36 afterRefresh(context, applicationArguments);
37 //
38
39 ⑨
40 listeners.finished(context, null)
41 ;
42 stopWatch.stop();
43 return context;
44 }
45 catch (Throwable ex) {
46 handleRunFailure(context, listeners, analyzers, ex);
47 throw new IllegalStateException(ex);
48 }
49 }
```

1. 通过SpringFactoriesLoader查找并加载所有的SpringApplicationRunListeners，通过调用starting()方法通知所有的

`SpringApplicationRunListeners`: 应用开始启动了。`SpringApplicationRunListeners`其本质上就是一个事件发布者，它在`SpringBoot`应用启动的不同时间点发布不同应用事件类型(`ApplicationEvent`)，如果有哪些事件监听者(`ApplicationListener`)对这些事件感兴趣，则可以接收并且处理。还记得初始化流程中，`SpringApplication`加载了一系列`ApplicationListener`吗？这个启动流程中没有发现有发布事件的代码，其实都已经在`SpringApplicationRunListeners`这儿实现了。

简单的分析一下其实现流程，首先看下`SpringApplicationRunListener`的源码：

```
1 public interface SpringApplicationRunListener {
2
3 // 运行run方法时立即调用此方法，可以用户非常早期的初始化工作
4 void starting()
5;
6
7 // Environment准备好后，并且ApplicationContext创建之前调用
8 void environmentPrepared(ConfigurableEnvironment environment)
9;
10
11 // ApplicationContext创建好后立即调用
12 void contextPrepared(ConfigurableApplicationContext context)
13;
14
15 // ApplicationContext加载完成，在refresh之前调用
16 void contextLoaded(ConfigurableApplicationContext context)
17;
18
19 // 当run方法结束之前调用
20 void finished(ConfigurableApplicationContext context, Throwable exception);
21
22}
```

`SpringApplicationRunListener`只有一个实现类：`EventPublishingRunListener`。①处的代码只会获取到一个`EventPublishingRunListener`的实例，我们来看看`starting()`方法的内容：

```
1 public void starting() {
2 // 发布一个ApplicationStartedEvent
3 this.initialMulticaster.multicastEvent(new ApplicationStartedEvent(this.application, this.args));
4 }
```

顺着这个逻辑，你可以在②处的`prepareEnvironment()`方法的源码中找到`listeners.environmentPrepared(environment)`；即`SpringApplicationRunListener`接口的第二个方法，那不出你所料，`environmentPrepared()`又发布了另外一个事件`ApplicationEnvironmentPreparedEvent`。接下来会发生什么，就不用我多说了吧。

2. 创建并配置当前应用将要使用的`Environment`，`Environment`用于描述应用程序当前的运行环境，其抽象了两个方面的内容：配置文件(`profile`)和属性(`properties`)，开发经验丰富的同学对这两个东西一定不会陌生：不同的环境(eg：生产环境、预发布环境)可以使用不同的配置文件，而属性则可以从配置文件、环境变量、命令行参数等来源获取。因此，当`Environment`准备好后，在整个应用的任何时候，都可以从`Environment`中获取资源。[微信搜索 web\\_resource 关注后获取更多干货](#)

总结起来，2处的两句代码，主要完成以下几件事：

- 判断Environment是否存在，不存在就创建（如果是web项目就创建StandardServletEnvironment，否则创建StandardEnvironment）
  - 配置Environment：配置profile以及properties
  - 调用SpringApplicationRunListener的environmentPrepared()方法，通知事件监听者：应用的Environment已经准备好了

3. SpringBoot应用在启动时会输出这样的东西：

如果想把这个东西改成自己的涂鸦，你可以研究以下Banner的实现，这个任务就留给你们吧。

4. 根据是否是web项目，来创建不同的ApplicationContext容器。

5. 创建一系列FailureAnalyzer，创建流程依然是通过SpringFactoriesLoader获取到所有实现FailureAnalyzer接口的class，然后在创建对应的实例。FailureAnalyzer用于分析故障并提供相关诊断信息。

6. 初始化ApplicationContext，主要完成以下工作：

- 将准备好的Environment设置给ApplicationContext
  - 遍历调用所有的 ApplicationContextInitializer 的 initialize() 方法来对已经创建好的 ApplicationContext 进行进一步的处理
  - 调用 SpringApplicationRunListener 的 contextPrepared() 方法，通知所有的监听者： ApplicationContext 已经准备完毕
  - 将所有的 bean 加载到容器中
  - 调用 SpringApplicationRunListener 的 contextLoaded() 方法，通知所有的监听者： ApplicationContext 已经装载完毕

7. 调用ApplicationContext的refresh()方法，完成IoC容器可用的最后一道工序。从名字上理解为刷新容器，那何为刷新？就是插手容器的启动，联系一下第一小节的内容。那如何刷新呢？且看下面代码：

```
1 // 摘自refresh()方法中一句代码
2 invokeBeanFactoryPostProcessors(beanFactory);
```

看看这个方法的实现：

```
1 protected void invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory) {
2 PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(beanFactory, getBeanFactoryPostProcessors());
3
4 }
```

获取到所有的 BeanFactoryPostProcessor来对容器做一些额外的操作。BeanFactoryPostProcessor允许我们在容器实例化相对象之前，对注册到容器的BeanDefinition所保存的信息做一些额外的操作。这里的getBeanFactoryPostProcessors()方法可以获取到3个Processor:

```
1 ConfigurationWarningsApplicationContextInitializer$ConfigurationWarningsPostProcessor
2 SharedMetadataReaderFactoryContextInitializer$CachingMetadataReaderFactoryPostProcessor
3 ConfigFileApplicationListener$PropertySourceOrderingPostProcessor
```

不是有那么多BeanFactoryPostProcessor的实现类，为什么这儿只有这3个？因为在初始化流程获取到的各种 ApplicationContextInitializer和ApplicationListener中，只有上文3个做了类似于如下操作：

```
1 public void initialize(ConfigurableApplicationContext context) {
2 context.addBeanFactoryPostProcessor(new ConfigurationWarningsPostProcessor(getChecks()));
3 }
```

然后你就可以进入到PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors()方法了，这个方法除了会遍历上面的3个BeanFactoryPostProcessor处理外，还会获取类型为BeanDefinitionRegistryPostProcessor的bean：org.springframework.context.annotation.internalConfigurationAnnotationProcessor，对应的Class为ConfigurationClassPostProcessor。ConfigurationClassPostProcessor用于解析处理各种注解，包括：@Configuration、@ComponentScan、@Import、@PropertySource、@ImportResource、@Bean。当处理@import注解的时候，就会调用<自动配置>这一小节中的EnableAutoConfigurationImportSelector.selectImports()来完成自动配置功能。其他的这里不再多讲，如果你有兴趣，可以查阅参考资料6。

8. 查找当前context中是否注册有CommandLineRunner和ApplicationRunner，如果有则遍历执行它们。

9. 执行所有SpringApplicationRunListener的finished()方法。

这就是Spring Boot的整个启动流程，其核心就是在Spring容器初始化并启动的基础上加入各种扩展点，这些扩展点包括：ApplicationContextInitializer、ApplicationListener以及各种BeanFactoryPostProcessor等等。你对整个流程的细节不必太过关注，甚至没弄明白也没有关系，你只要理解这些扩展点是在何时如何工作的，能让它们为你所用即可。

整个启动流程确实非常复杂，可以查询参考资料中的部分章节和内容，对照着源码，多看看，我想最终你都能弄清楚的。言而总之，Spring才是核心，理解清楚Spring容器的启动流程，那Spring Boot启动流程就不在话下了。

来源：51cto

作者：CHEN川

原文：<https://blog.51cto.com/luecsc/1964056>

- END -

[1] 王福强著；*SpringBoot揭秘：快速构建微服务体系*；机械工业出版社，2016

[2] 王福强著；*Spring揭秘*；人民邮电出版社，2009

[3] Craig Walls 著；丁雪丰译；*Spring Boot实战*；中国工信出版集团人民邮电出版社，2016

[4] 深入探讨 Java 类加载器

[5] spring boot实战：自动配置原理分析

[6] spring boot实战：Spring boot Bean加载源码分析

如果看到这里，说明你喜欢这篇文章，帮忙[转发](#)一下吧，感谢。微信搜索「web\_resource」，关注后即可获取每日一题的推送。

## 推荐阅读

1. 把 14 亿中国人拉到一个微信群？
2. Spring 中的 18 个注解，你会几个？
3. 寓教于乐，用玩游戏的方式学习 Git
4. 在浏览器输入 URL 回车之后发生了什么？
5. Java 最常见的 208 道面试题

[Java后端：专注于Java技术](#)



Java后端

长按识别二维码，关注我的公众号

喜欢文章，点个在看 

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 快速入手 Spring Boot 参数校验

Java后端 2019-12-31

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

来源 | 狂乱的贵公子

链接 | [cnblogs.com/cjsblog/p/8946768.html](http://cnblogs.com/cjsblog/p/8946768.html)

## 1、背景介绍

开发过程中, 后台的参数校验是必不可少的, 所以经常会看到类似下面这样的代码

```
public BaseResult save(CouponEntity coupon) {
 if (StringUtils.isBlank(coupon.getCouponName())) {
 return BaseResult.failure("优惠券名称不能为空");
 }
 if (null == coupon.getCouponType()) {
 return BaseResult.failure("优惠券类型不能为空");
 }
 if (coupon.getCouponType() == CouponTypeEnum.ZHE_KOU.getType() && null == coupon.getDiscountRate()) {
 return BaseResult.failure("折扣率不能为空");
 }
 if (coupon.getCouponType() != CouponTypeEnum.ZHE_KOU.getType() && null == coupon.getCouponAmount()) {
 return BaseResult.failure("优惠券面值不能为空");
 }
 if (null == coupon.getCouponNum()) {
 return BaseResult.failure("发放总量不能为空");
 }
 if (null == coupon.getReleaseStartTime() || null == coupon.getReleaseEndTime()) {
 return BaseResult.failure("发放起止时间不能为空");
 }
 if (coupon.getReleaseStartTime().compareTo(coupon.getReleaseEndTime()) > 0) {
 return BaseResult.failure("发放开始时间不能大于发放结束时间");
 }
 if (null == coupon.getAvailableType()) {
 return BaseResult.failure("有效期不能为空");
 }
 if (coupon.getAvailableType() == CouponValidityTypeEnum.FIXED_DATE.getType()) {
 if (null == coupon.getAvailableStartTime() || null == coupon.getAvailableEndTime()) {
 return BaseResult.failure("有效日期不能为空");
 }
 if (coupon.getAvailableStartTime().compareTo(coupon.getAvailableEndTime()) > 0) {
 return BaseResult.failure("有效日期开始时间不能大于结束时间");
 }
 if (coupon.getAvailableStartTime().compareTo(coupon.getReleaseStartTime()) < 0) {
 return BaseResult.failure("有效日期开始时间不能小于发放开始时间");
 }
 if (coupon.getAvailableEndTime().compareTo(coupon.getReleaseEndTime()) < 0) {
 return BaseResult.failure("发放结束时间不能大于有效期结束时间");
 }
 }
 if (coupon.getAvailableType() == CouponValidityTypeEnum.GET_DATE.getType() && null == coupon.getAvailableDay()) {
 return BaseResult.failure("领取之日不能为空");
 }
 if (null == coupon.getShopRange()) {
 return BaseResult.failure("适用门店不能为空");
 }
 if (coupon.getShopRange() == 2 && StringUtils.isBlank(coupon.getShopIds())) {
 return BaseResult.failure("请指定门店");
 }
 if (null == coupon.getProductRange()) {
 return BaseResult.failure("适用商品不能为空");
 }
 if (coupon.getProductRange() == 2 && StringUtils.isBlank(coupon.getProductTypeIds())) {
 return BaseResult.failure("请指定分类");
 }
}
```

这样写并没有什么错, 还挺工整的, 只是看起来不是很优雅而已。

接下来, 用Validation来改写这段

## 2、Spring Boot文档中的Validation

在 Spring Boot 的官网中, 关于Validation只是简单的提了一句, 如下

# 35. Validation

The method validation feature supported by Bean Validation 1.1 is automatically enabled as long as a JSR-303 implementation (such as Hibernate validator) is on the classpath. This lets bean methods be annotated with `javax.validation` constraints on their parameters and/or on their return value. Target classes with such annotated methods need to be annotated with the `@Validated` annotation at the type level for their methods to be searched for inline constraint annotations.

For instance, the following service triggers the validation of the first argument, making sure its size is between 8 and 10:

```
@Service
@Validated
public class MyBean {

 public Archive findByNameAndAuthor(@Size(min = 8, max = 10) String code,
 Author author) {
 ...
}
```

其实，**Spring Validator** 和Hibernate Validator是两套Validator，可以混着用，这里我们用Hibernate Validator

## 3、Hibernate Validator

[https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html\\_single/#preface](https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#preface)

## 4、Spring Validator

<https://docs.spring.io/spring/docs/5.0.5.RELEASE/spring-framework-reference/core.html#validation>

## 5、示例

### 5.1、引入spring-boot-starter-validation

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

### 5.2、定义一个对象

```
@Data
public class LoginForm {

 @NotBlank(message = "用户名不能为空")
 @Email
 private String username;

 @NotBlank(message = "密码不能为空")
 @Length(min = 6, message = "密码长度至少6位")
 private String password;
}
```

### 5.3、适用@Valid校验，并将校验结果放到BindingResult对象中

```
@Controller
@RequestMapping("/login")
public class LoginController extends BaseController {

 private ObjectError error;

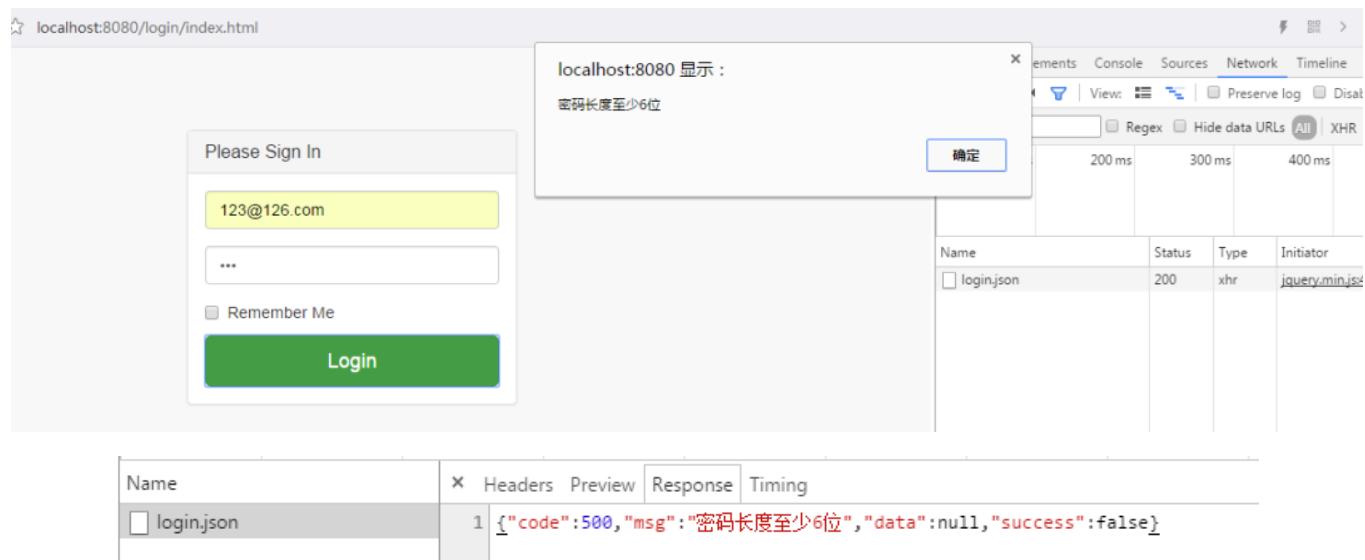
 @GetMapping("/index.html")
 public ModelAndView index() {
 return new ModelAndView("login");
 }

 @PostMapping("/login.json")
 @ResponseBody
 public RespResult login(@Valid LoginForm loginRequest, BindingResult bindingResult) {
 if (bindingResult.hasErrors()) {
 if (bindingResult.hasErrors()) {
 for (ObjectError error : bindingResult.getAllErrors()) {
 return RespResult.failure(error.getDefaultMessage());
 }
 }
 }
 return RespResult.success();
 }
}
```

注意：

- 默认情况下，如果校验失败会抛javax.validation.ConstraintViolationException异常，可以用统一异常处理去对这些异常做处理
- An Errors/BindingResult argument is expected to be declared immediately after the model attribute

### 5.4、看效果



如果在校验的对象后面再加上Model对象的话，如果返回的是 ModelAndView 就可以将这个 Model 设置到其中，这样在页面就可以取到错误消息了

```
@PostMapping("/login.json")
@ResponseBody
public RespResult login(@Valid LoginForm loginRequest, BindingResult bindingResult, Model model) {
 System.out.println(model);
 if (bindingResult.hasErrors()) {
 if (bindingResult.hasErrors()) {
 for (ObjectError error : bindingResult.getAllErrors()) {
 return RespResult.failure(error.getDefaultMessage());
 }
 }
 }
 return RespResult.success();
}
```

## 5.5、自定义校验规则

[https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html\\_single/#validator-customconstraints](https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#validator-customconstraints)

### 6.1. Creating a simple constraint

To create a custom constraint, the following three steps are required:

- Create a constraint annotation
- Implement a validator
- Define a default error message

这里，以优惠券创建为例来演示如何自定义校验规则

首先，优惠券表单如下（仅仅只是演示用）：

```
@Data
@CheckTimeInterval(startTime = "releaseStartTime", endTime = "releaseEndTime", message = "发放开始时间不能大于发放结束时间")
@CheckValidityMode(message = "领取后有效天数不能为空")
public class CouponForm {
 @NotNull(groups = {GroupCouponEdit.class}, message = "优惠券ID不能为空")
 private Long couponId; // 优惠券ID

 @NotNull(groups = {GroupCouponAdd.class, GroupCouponEdit.class}, message = "商家ID不能为空")
 private Integer merchantId; // 商家ID

 @NotBlank(groups = {GroupCouponAdd.class, GroupCouponEdit.class}, message = "优惠券名称不能为空")
 @Length(groups = {GroupCouponAdd.class, GroupCouponEdit.class}, max = 16, message = "优惠券名称最大长度为16")
 private String couponName; // 优惠券名称

 @NotNull(groups = {GroupCouponAdd.class, GroupCouponEdit.class}, message = "优惠券类型不能为空")
 private Integer couponType; // 优惠券类型

 @NotNull
 private Integer parValue; // 面值

 @NotNull
 private Integer quantity; // 发放数量

 private Date releaseStartTime; // 发放开始时间

 private Date releaseEndTime; // 发放结束时间

 @NotNull
 private Integer validityMode; // 有效期模式

 private Integer days; // 领取后多少天内有效

 private Integer limitType; // 限制领取类型

 private Integer limitNum; // 限制领取数量

 private Date validityStartTime; // 有效期开始时间

 private Date validityEndTime; // 有效期结束时间

 @Size(max = 200)
 private String remark; // 备注
}
```

这里除了自定义了两条校验规则之外，还用到了分组。

为什么要有分组这一说呢？因为，举个例子，添加的时候不需要校验id，而修改的时候id不能为空，有了分组以后，就可以添加的时候校验用组A，修改的时候校验用组B

下面重点看一下@CheckTimeInterval

**第一步、定义一个注解叫CheckTimeInterval**

```

@Target({TYPE, FIELD, METHOD, PARAMETER, ANNOTATION_TYPE, TYPE_USE})
@Retention(RUNTIME)
@Constraint(validatedBy = CheckTimeIntervalValidator.class)
@Documented
@Repeatable(CheckTimeInterval.List.class)
public @interface CheckTimeInterval {

 String startTime() default "from";

 String endTime() default "to";

 String message() default "{org.hibernate.validator.referenceguide.chapter06.CheckCase." +
 "message}";

 Class<?>[] groups() default { };

 Class<? extends Payload>[] payload() default { };

 @Target({TYPE, FIELD, METHOD, PARAMETER, ANNOTATION_TYPE})
 @Retention(RUNTIME)
 @Documented
 @interface List {
 CheckTimeInterval[] value();
 }
}

```

## 第二步、定义Validator去校验它

```

import org.springframework.beans.BeanWrapper;
import org.springframework.beans.BeanWrapperImpl;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import java.util.Date;

public class CheckTimeIntervalValidator implements ConstraintValidator<CheckTimeInterval, Object> {

 private String startTime;

 private String endTime;

 @Override
 public void initialize(CheckTimeInterval constraintAnnotation) {
 this.startTime = constraintAnnotation.startTime();
 this.endTime = constraintAnnotation.endTime();
 }

 @Override
 public boolean isValid(Object value, ConstraintValidatorContext context) {
 if (null == value) {
 return true;
 }

 BeanWrapper beanWrapper = new BeanWrapperImpl(value);
 Object start = beanWrapper.getPropertyValue(startTime);
 Object end = beanWrapper.getPropertyValue(endTime);

 if (null == start || null == end) {
 return true;
 }

 int result = ((Date) end).compareTo((Date) start);

 if (result >= 0) {
 return true;
 }

 return false;
 }
}

```

顺便提一句，这里BeanWrapper去取对象的属性值，我们稍微看一下BeanWrapper是做什么的

### 3.4. Bean manipulation and the BeanWrapper

The `org.springframework.beans` package adheres to the JavaBeans standard provided by Oracle. A JavaBean is simply a class with a default no-argument constructor, which follows a naming convention where (by way of an example) a property named `bingoMadness` would have a setter method `setBingoMadness(..)` and a getter method `getBingoMadness()`. For more information about JavaBeans and the specification, please refer to Oracle's website ([javabeans](#)).

One quite important class in the beans package is the `BeanWrapper` interface and its corresponding implementation (`BeanWrapperImpl`). As quoted from the javadocs, the `BeanWrapper` offers functionality to set and get property values (individually or in bulk), get property descriptors, and to query properties to determine if they are readable or writable. Also, the `BeanWrapper` offers support for nested properties, enabling the setting of properties on sub-properties to an unlimited depth. Then, the `BeanWrapper` supports the ability to add standard JavaBeans `PropertyChangeListeners` and `VetoableChangeListeners`, without the need for supporting code in the target class. Last but not least, the `BeanWrapper` provides support for the setting of indexed properties. The `BeanWrapper` usually isn't used by application code directly, but by the `DataBinder` and the `BeanFactory`.

The way the `BeanWrapper` works is partly indicated by its name: *it wraps a bean* to perform actions on that bean, like setting and retrieving properties.

The following code snippets show some examples of how to retrieve and manipulate some of the properties of instantiated `Companies` and `Employees`:

```
BeanWrapper company = new BeanWrapperImpl(new Company());
// setting the company name..
company.setPropertyValue("name", "Some Company Inc.");
// ... can also be done like this:
PropertyValue value = new PropertyValue("name", "Some Company Inc.");
company.setPropertyValue(value);

// ok, let's create the director and tie it to the company:
BeanWrapper jim = new BeanWrapperImpl(new Employee());
jim.setPropertyValue("name", "Jim Stravinsky");
company.setPropertyValue("managingDirector", jim.getWrappedInstance());

// retrieving the salary of the managingDirector through the company
Float salary = (Float) company.getPropertyValue("managingDirector.salary");
```

言归正传

### 第三步、验证

```
@PostMapping("/save.json")
@ResponseBody
public RespResult save(@Valid @Validated(GroupCouponAdd.class) CouponForm couponForm, BindingResult bindingResult) {
 if (bindingResult.hasErrors()) {
 for (ObjectError error : bindingResult.getAllErrors()) {
 System.out.println(error.getDefaultMessage());
 }
 }
 return RespResult.success();
}
```

x Headers Preview Response Timing

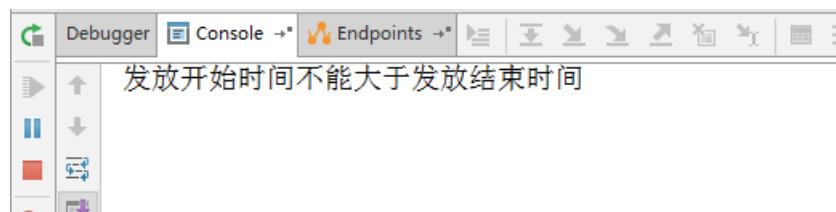
▼ General  
 Request URL: http://localhost:8080/coupon/save.json  
 Request Method: POST  
 Status Code: 200  
 Remote Address: [::1]:8080

▼ Response Headers view source  
 Content-Type: application/json; charset=UTF-8  
 Date: Thu, 26 Apr 2018 07:35:24 GMT  
 Transfer-Encoding: chunked

▼ Request Headers view source  
 Accept: \*/\*  
 Accept-Encoding: gzip, deflate  
 Accept-Language: zh-CN,zh;q=0.8  
 Connection: keep-alive  
 Content-Length: 188  
 Content-Type: application/x-www-form-urlencoded; charset=UTF-8  
 Host: localhost:8080  
 Origin: http://localhost:8080  
 Referer: http://localhost:8080/coupon/add.html  
 User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.36 Core/1.53.4882.400 QQBrowser/9.7.13059.400  
 X-Requested-With: XMLHttpRequest

▼ Form Data view source view URL encoded

```
merchantId: 123
couponName: 哈哈哈
couponType: 1
parValue: 100
quantity: 1000
releaseStartTime: 2018-04-25 00:00:00
releaseEndTime: 2018-04-21 23:59:59
validityMode: 2
```



看，自定义的校验生效了

## 6、补充

### 6.1、校验模式

[https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html\\_single/#section-fail-fast](https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#section-fail-fast)

下面补充一点，关于校验模式

默认会校验完所有属性，然后将错误信息一起返回，但很多时候不需要这样，一个校验失败了，其它就不必校验了

为此，需要这样设置

```

import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

@Configuration
public class ValidatorConfig {

 @Bean
 public Validator validator() {
 ValidatorFactory validatorFactory = Validation.byProvider(HibernateValidator.class)
 .configure()
 .failFast(true)
 .buildValidatorFactory();
 Validator validator = validatorFactory.getValidator();
 return validator;
 }
}

```

## 6.2、单个参数校验

```

@Controller
@RequestMapping("/coupon")
@Validated
public class CouponController extends BaseController {

 @GetMapping("/detail.html")
 public ModelAndView detail(@NotNull(message = "ID不能为空") Long id) {
 ModelAndView modelAndView = new ModelAndView(viewName: "coupon/detail");
 // TODO 查询
 return modelAndView;
 }
}

```

< > ⏪ ⏴ | ★ localhost:8080/coupon/detail.html?id=

500



如果是调整页面的时候参数校验失败的话，这时可以不做处理，让其调到错误页面。

如果是接口参数校验失败的话，可以在这里进行统一处理，并返回。例如：

```

@ControllerAdvice
@Component
public class GlobalExceptionHandlerAdvice {

 @ExceptionHandler(ConstraintViolationException.class)
 public RespResult handler(HttpServletRequest request, ConstraintViolationException ex) {
 if (WebUtils.isAjax(request)) {
 StringBuffer sb = new StringBuffer();
 for (ConstraintViolation violation : ex.getConstraintViolations()) {
 sb.append(violation.getMessage());
 }
 return RespResult.failure(sb.toString());
 }
 throw ex;
 }
}

```

## 6.3、错误页面

```

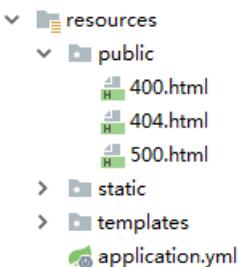
@SpringBootApplication
public class CjsSpringbootExampleApplication {

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

 @Bean
 public ErrorPageRegistrar errorPageRegistrar() {
 return new ErrorPageRegistrar() {
 @Override
 public void registerErrorPages(ErrorPageRegistry registry) {
 registry.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, path: "/400.html"));
 registry.addErrorPages(new ErrorPage(HttpStatus.FORBIDDEN, path: "/403.html"));
 registry.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND, path: "/404.html"));
 registry.addErrorPages(new ErrorPage(HttpStatus.INTERNAL_SERVER_ERROR, path: "/500.html"));
 }
 };
 }

 // @Bean
 // public ErrorViewResolver MyErrorViewResolver() {
 // return new ErrorViewResolver() {
 // @Override
 // public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status, Map<String, Object>
 // // return null;
 // // }
 // };
 // }
}

```



以刚才优惠券详情为例

http://localhost:8080/coupon/detail.html	400
http://localhost:8080/coupon/detail.html?id=	400
http://localhost:8080/coupon/detail.html?id=abc	400
http://localhost:8080/coupon/detail222.html?id=123	404

无权限 403

int a = 1 / 0; 500

#### 6.4、@Valid与@Validated

[https://blog.csdn.net/qq\\_27680317/article/details/79970590](https://blog.csdn.net/qq_27680317/article/details/79970590)

#### 参考

<http://rensanning.iteye.com/blog/2357373>

<https://blog.csdn.net/kenight/article/details/77774465>

<https://www.cnblogs.com/mr-yang-localhost/p/7812038.html>

<https://www.phpsong.com/3567.html>

<https://www.cnblogs.com/mr-yang-localhost/p/7812038.html>

- END -

#### 推荐阅读

- [1. 实战:SpringBoot & Restful API 构建示例](#)
- [2. 关于 CPU 的一些基本知识总结](#)
- [3. 发布没有答案的面试题,都是耍流氓](#)
- [4. 什么是一致性 Hash 算法?](#)
- [5. 团队开发中 Git 最佳实践](#)



喜欢文章, 点个在看 

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

# 惊呆了，Spring Boot居然这么耗内存！

襄垣 Java后端 2019-09-16

点击上方**蓝色字体**, 选择“标星公众号”

优质文章, 第一时间送达

作者 | 襄垣

juejin.im/post/5c89f266f265da2d8763b5f9

Spring Boot总体来说，搭建还是比较容易的，特别是Spring Cloud全家桶，简称亲民微服务，但在发展趋势中，容器化技术已经成熟，面对巨耗内存的Spring Boot，小公司表示用不起。如今，很多刚诞生的JAVA微服务框架大多主打“轻量级”，主要还是因为Spring Boot太重。

## JAVA系微服务框架

### No1-Spring Cloud

#### 介绍

有Spring大靠山在，更新、稳定性、成熟度的问题根本不需要考虑。在JAVA系混的技术人员大约都听说过Spring的大名吧，所以不缺程序员……，而且这入手的难度十分低，完全可以省去一个架构师。

但是，你必然在服务器上付出：

- 至少一台“服务发现”的服务器；
- 可能有一个统一的网关Gateway；
- 可能需要一个用于“分布式配置管理”的配置中心；
- 可能进行“服务追踪”，知道我的请求从哪里来，到哪里去；
- 可能需要“集群监控”；
- 项目上线后发现，我们需要好多服务器，每次在集群中增加服务器时，都感觉心疼；

#### 压测30秒

##### 压测前的内存占用

```
Processes: 340 total, 2 running, 338 sleeping, 1715 threads 10:12:39
Load Avg: 2.87, 2.45, 1.99 CPU usage: 2.29% user, 1.81% sys, 95.88% idle
SharedLibs: 164M resident, 53M data, 34M linkedit.
MemRegions: 56640 total, 7188M resident, 179M private, 3249M shared.
PhysMem: 16G used (2653M wired), 228M unused.
VM: 1531G vsize, 1112M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 124631/37M in, 52794/14M out.
Disks: 164600/3838M read, 52539/937M written.

PID COMMAND %CPU TIME #TH #WQ #POR MEM PURG CMPR PGRP PPID
1526 java 0.1 00:10.58 43 1 124 304M 0B 0B 1526 1449
```

如图，内存占用304M。

##### 压测时的内存占用

```

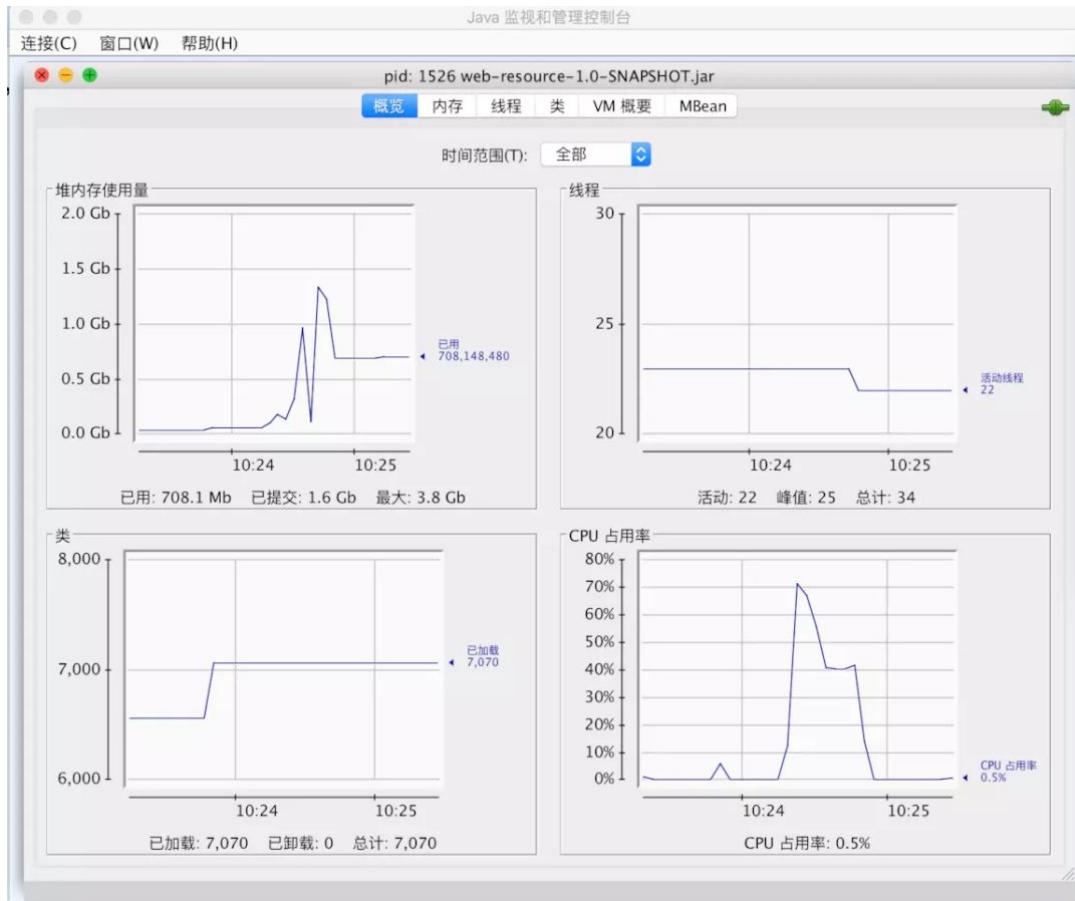
Processes: 347 total, 8 running, 339 sleeping, 1917 threads 10:29:42
Load Avg: 2.43, 2.61, 2.38 CPU usage: 45.93% user, 40.90% sys, 13.15% idle
SharedLibs: 165M resident, 53M data, 33M linkedit.
MemRegions: 60070 total, 8183M resident, 184M private, 2083M shared.
PhysMem: 16G used (2721M wired), 356M unused.
VM: 1568G vsize, 1112M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 16407645/1573M in, 16310036/1547M out.
Disks: 172958/3944M read, 61862/1207M written.

PID COMMAND %CPU TIME #TH #WQ #POR MEM PURG CMPR PGRP PPID
1526 java 321.0 02:50.33 41/4 1 120 1520M 0B 0B 1526 1449

```

如图，内存占用1520M（1.5G），CPU上升到321%

## 概览



## 总结

一个Spring Boot的简单应用，最少1G内存，一个业务点比较少的微服务编译后的JAR会大约50M；而Spring Cloud引入的组件会相对多一些，消耗的资源也会相对更多一些。

启动时间大约10秒左右: Started Application in 10.153 seconds (JVM running for 10.915)

## JAVA系响应式编程的工具包Vert.x

## 介绍

背靠Eclipse的Eclipse Vert.x是一个用于在JVM上构建响应式应用程序的工具包。定位上与Spring Boot不冲突，甚至可以将Vert.x结合Spring Boot使用。众多Vert.x模块提供了大量微服务的组件，在很多人眼里是一种微服务架构的选择。

## 压测30秒

### 压测前的内存占用

```
Processes: 319 total, 2 running, 317 sleeping, 1642 threads 12:11:55
Load Avg: 2.02, 1.87, 1.89 CPU usage: 3.15% user, 2.42% sys, 94.41% idle
SharedLibs: 199M resident, 58M data, 45M linkedit.
MemRegions: 69030 total, 7190M resident, 207M private, 3100M shared.
PhysMem: 15G used (2599M wired), 768M unused.
VM: 1447G vsize, 1111M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 282452/109M in, 84946/21M out.
Disks: 157962/4480M read, 126426/2246M written.

PID COMMAND %CPU TIME #TH #WQ #POR MEM PURG CMPR PGRP PPID
2656 java 0.1 00:01.23 25 1 88 65M 0B 0B 2656 2337
```

如图，内存占用65M。

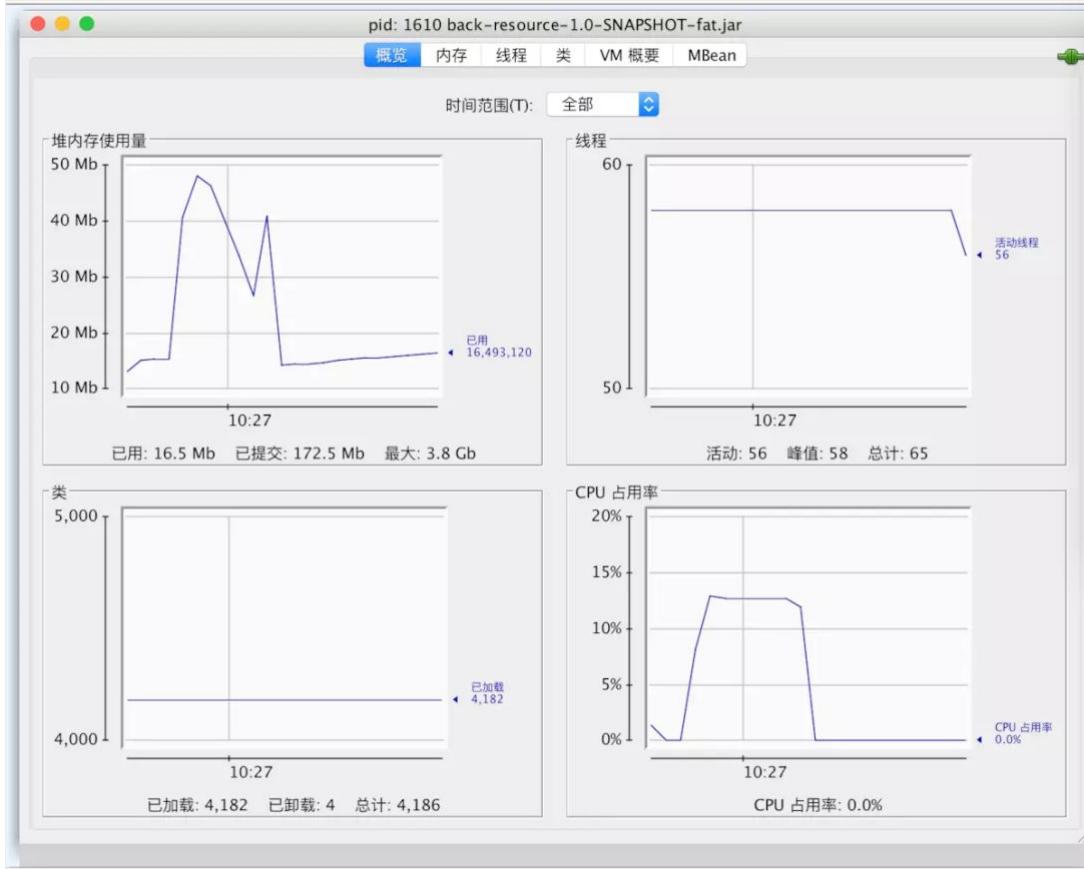
### 压测时的内存占用

```
Processes: 342 total, 3 running, 339 sleeping, 1682 threads 10:14:51
Load Avg: 2.05, 2.30, 1.99 CPU usage: 3.74% user, 3.14% sys, 93.10% idle
SharedLibs: 164M resident, 53M data, 33M linkedit.
MemRegions: 57424 total, 7319M resident, 180M private, 3223M shared.
PhysMem: 16G used (2683M wired), 29M unused.
VM: 1546G vsize, 1111M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 131426/38M in, 56136/14M out.
Disks: 165715/3854M read, 54070/1024M written.

PID COMMAND %CPU TIME #TH #WQ #POR MEM PURG CMPR PGRP PPID
1610 java 2.1 00:02.84 46 1 130+ 139M+ 0B 0B 1610 1396
```

如图，内存占139M，CPU占2.1%，给人的感觉似乎并没有进行压测。

## 概览



## 总结

Vert.x单个服务打包完成后大约7M左右的JAR，不依赖Tomcat、Jetty之类的容器，直接在JVM上跑。

Vert.x消耗的资源很低，感觉一个1核2G的服务器已经能够部署许多个Vert.x服务。除去编码方面的问题，真心符合小项目和小模块。git市场上已经出现了基于Vert.x实现的开源网关- VX-API-Gateway帮助文档

<https://duhua.gitee.io/vx-api-gateway-doc/>

对多语言支持，很适合小型项目快速上线。

启动时间不到1秒：Started Vert.x in 0.274 seconds (JVM running for 0.274)

## JAVA系其他微服务框架

### SparkJava

- jar比较小，大约10M
- 占内存小，大约30~60MB；
- 性能还可以，与Spring Boot相仿；

### Micronaut

- Grails团队新宠；
- 可以用 Java、Groovy 和 Kotlin 编写的基于微服务的应用程序；
- 相比Spring Boot已经比较全面；
- 性能较优，编码方式与Spring Boot比较类似；
- 启动时间和内存消耗方面比其他框架更高效；

- 多语言；
- 依赖注入；
- 内置多种云本地功能；
- 很新，刚发布1.0.0

## Javalin

- 上手极为容易；
- 灵活，可以兼容同步和异步两种编程思路；
- JAR小，4~5M；
- 多语言；
- 有KOA的影子；
- 只有大约2000行源代码，源代码足够简单，可以理解和修复；
- 符合当今趋势；
- 多语言；
- 嵌入式服务器Jetty；

## Quarkus

- 启动快；
- JAR小，大约10M；
- 文档很少；

如果喜欢本篇文章，欢迎[转发](#)、[点赞](#)。关注订阅号「Web项目聚集地」，回复「进群」即可进入无广告技术交流。

---

## 推荐阅读

1. Spring 中的 18 个注解，你会几个？
2. 寓教于乐，用玩游戏的方式学习 Git
3. 在浏览器输入 URL 回车之后发生了什么
4. 在阿里干了 5 年，面试个小公司挂了…



Web项目聚集地

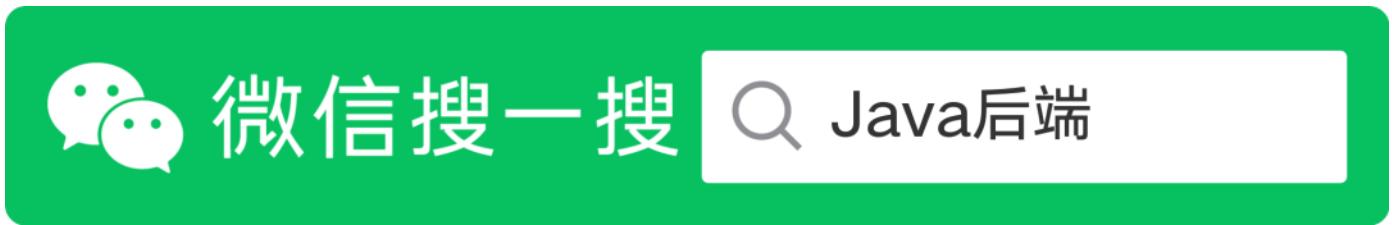
微信扫描二维码，关注我的公众号

喜欢文章，点个在看

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 我们老大半小时把我的 Spring Boot 项目并发提升几倍

lipengHeke Java后端 2月22日



作者: lipengHeke

出处: my.oschina.net/u/560547/blog/3162343

## 背景

生产环境偶尔会有一些慢请求导致系统性能下降，吞吐量下降，下面介绍几种优化建议。

## 方案

### 1、undertow替换tomcat

电子商务类型网站大多都是短请求，一般响应时间都在100ms，这时可以将web容器从tomcat替换为undertow，下面介绍下步骤：

#### 1、增加pom配置

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 <exclusions>
 <exclusion>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-tomcat</artifactId>
 </exclusion>
 </exclusions>
</dependency>
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

#### 2、增加相关配置

```
server:
 undertow:
 direct-buffers: true
 io-threads: 4
 worker-threads: 160
```

重新启动可以在控制台看到容器已经切换为undertow了

### 2、缓存

将部分热点数据或者静态数据放到本地缓存或者redis中，如果有需要可以定时更新缓存数据

### 3、异步

在代码过程中我们很多代码都不需要等返回结果，也就是部分代码是可以并行执行，这个时候可以使用异步，最简单的方案是使用springboot提供的@Async注解，当然也可以通过线程池来实现，下面简单介绍下异步步骤。1、pom依赖一般springboot引入web相关依赖就行

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

2、在启动类中增加@EnableAsync注解

```
@EnableAsync
@SpringBootApplication
public class AppApplication
{
 public static void main(String[] args)
 {
 SpringApplication.run(AppApplication.class, args);
 }
}
```

3、需要时在指定方法中增加@Async注解，如果是需要等待返回值，则demo如下

```
@Async
public Future<String> doReturn(int i){
 try {
 // 这个方法需要调用500毫秒
 Thread.sleep(500);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 / 消息汇总
 return new AsyncResult("<异步调用");
}
```

4、如果有线程变量或者logback中的mdc，可以增加传递

```
import java.util.Map;
import java.util.concurrent.Executor;

/**
 * @Description:
 */
@EnableAsync
@Configuration
public class AsyncConfig extends AsyncConfigurerSupport {
 @Override
 public Executor getAsyncExecutor() {
 ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
 executor.setTaskDecorator(new MdcTaskDecorator());
 executor.initialize();
 return executor;
 }
}

class MdcTaskDecorator implements TaskDecorator {

 @Override
 public Runnable decorate(Runnable runnable) {
 Map<String, String> contextMap = MDC.getCopyOfContextMap();
 return () -> {
 try {
 MDC.setContextMap(contextMap);
 runnable.run();
 } finally {
 MDC.clear();
 }
 };
 }
}
```

## 5、有时候异步需要增加阻塞

```

@Configuration
@Slf4j
public class TaskExecutorConfig {

 @Bean("localDbThreadPoolTaskExecutor")
 public Executor threadPoolTaskExecutor() {
 ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
 taskExecutor.setCorePoolSize(5);
 taskExecutor.setMaxPoolSize(200);
 taskExecutor.setQueueCapacity(200);
 taskExecutor.setKeepAliveSeconds(100);
 taskExecutor.setThreadNamePrefix("LocalDbTaskThreadPool");
 taskExecutor.setRejectedExecutionHandler((Runnable r, ThreadPoolExecutor executor) -> {
 if (!executor.isShutdown()) {
 try {
 Thread.sleep(300);
 executor.getQueue().put(r);
 } catch (InterruptedException e) {
 log.error(e.toString(), e);
 Thread.currentThread().interrupt();
 }
 }
 });
 taskExecutor.initialize();
 return taskExecutor;
 }

}

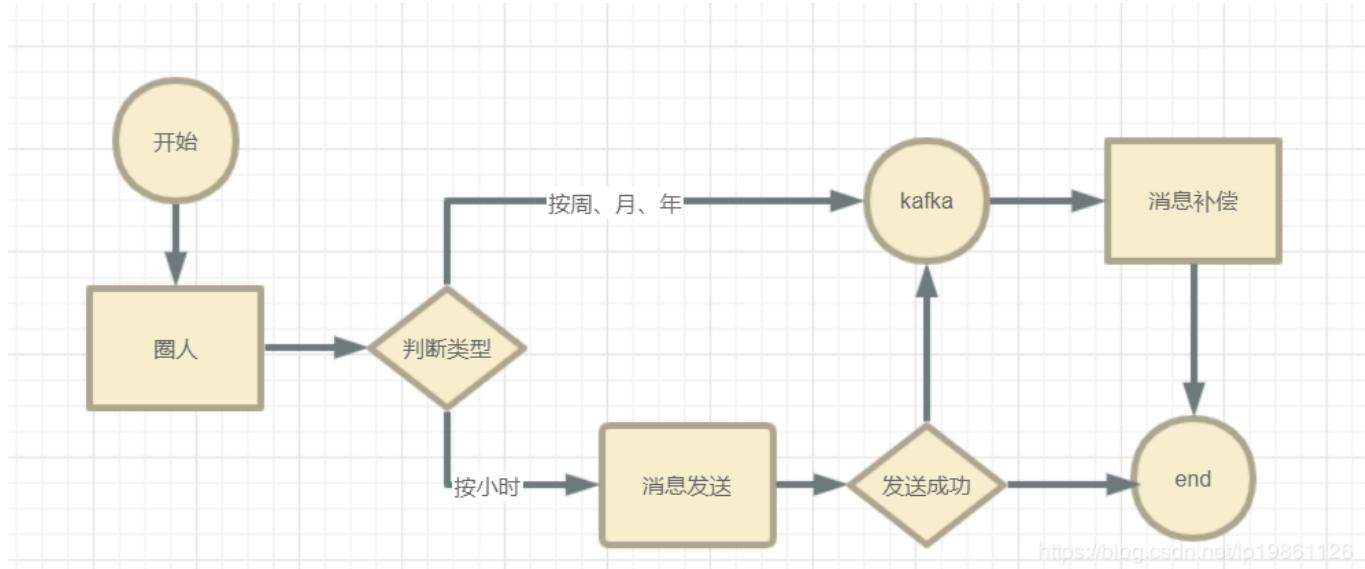
```

#### 4、业务拆分

可以将比较耗时或者不同的业务拆分出来提供单节点的吞吐量

#### 5、集成消息队列

有很多场景对数据实时性要求不那么强的，或者对业务进行业务容错处理时可以将消息发送到kafka，然后延时消费。举个例子，根据条件查询指定用户发送推送消息，这里可以时按时、按天、按月等等，这时就



如果看到这里，说明你喜欢这篇文章，请转发、点赞。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文(无广告)。

↓ 扫描二维码添加小编 ↓



#### 推荐阅读

1. [自费送新款 iPad，包邮！](#)
2. [18 个示例带你掌握 Java 8 日期时间处理！](#)
3. [安利一款 IDEA 中强大的代码生成利器](#)
4. [如何获取靠谱的新型冠状病毒疫情](#)



微信搜一搜  
Q Java后端

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 提升 10 倍生产力：IDEA 远程一键部署 Spring Boot 到 Docker

陶章好 Java后端 2019-09-29

点击上方 Java后端, 选择“[设为星标](#)”

优质文章, 及时送达

作 者 | 陶章好

链 接 | [juejin.im/post/5d026212f265da1b8608828b](https://juejin.im/post/5d026212f265da1b8608828b)

上一篇 | [推荐 7 个 Spring Boot 前后端分离项目](#)

IDEA是Java开发利器，Spring Boot是Java生态中最流行的微服务框架，docker是时下最火的容器技术，那么它们结合在一起会产生什么化学反应呢？

## 一、开发前准备

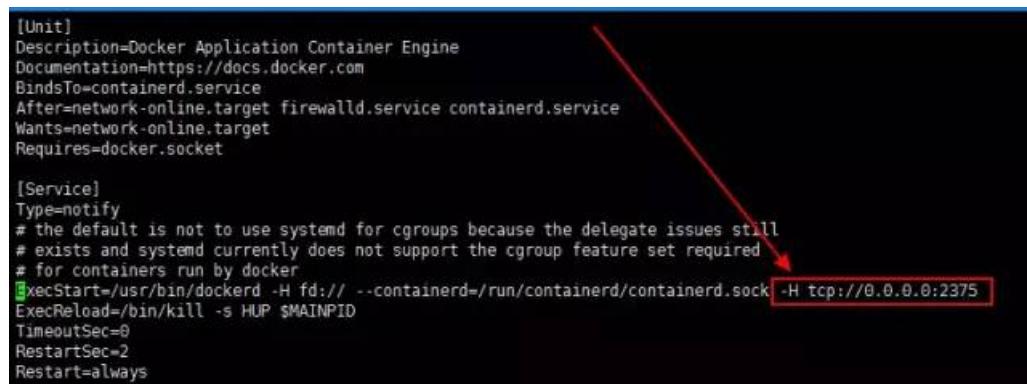
### 1. Docker安装

可以参考：<https://docs.docker.com/install/>

### 2. 配置docker远程连接端口

```
1 vi /usr/lib/systemd/system/docker.service
```

找到 ExecStart，在最后面添加 -H tcp://0.0.0.0:2375，如下图所示



```
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
BindsTo=containerd.service
After=network-online.target firewalld.service containerd.service
Wants=network-online.target
Requires=docker.socket

[Service]
Type=notify
the default is not to use systemd for cgroups because the delegate issues still
exists and systemd currently does not support the cgroup feature set required
for containers run by docker
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock -H tcp://0.0.0.0:2375
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always
```

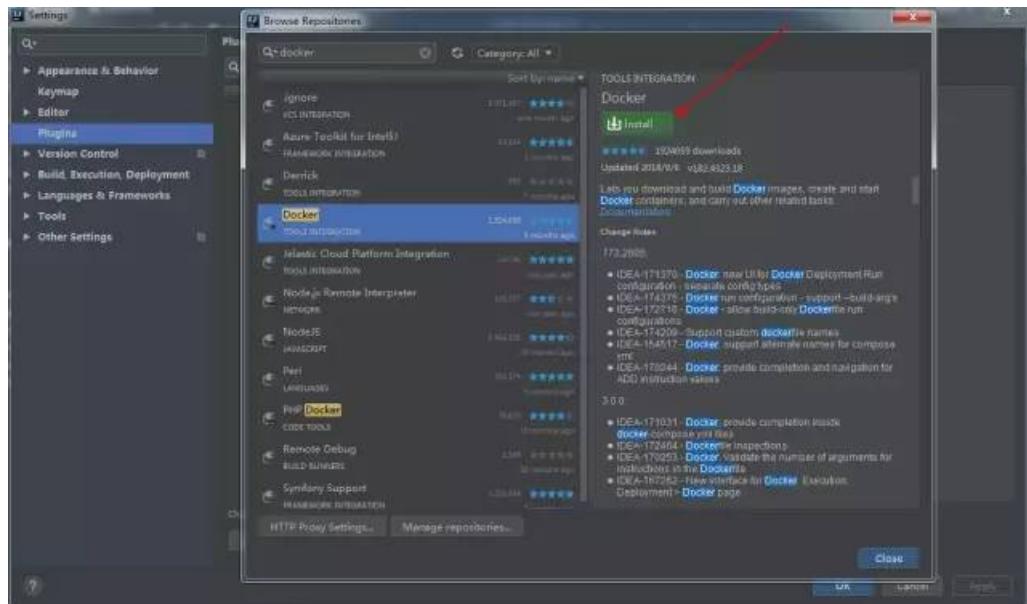
### 3. 重启docker

```
1 systemctl daemon-reload
2 systemctl start docker
```

### 4. 开放端口

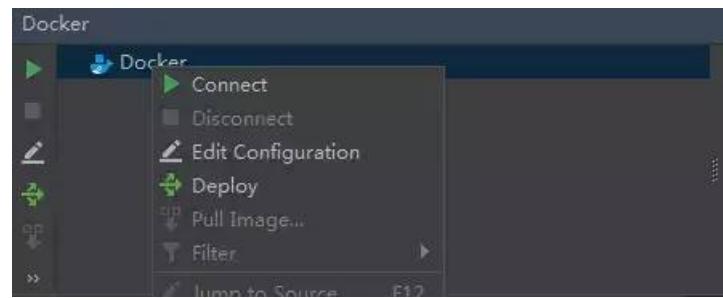
```
1 firewall-cmd --zone=public --add-port=2375/tcp --permanent
```

## 5. Idea安装插件，重启

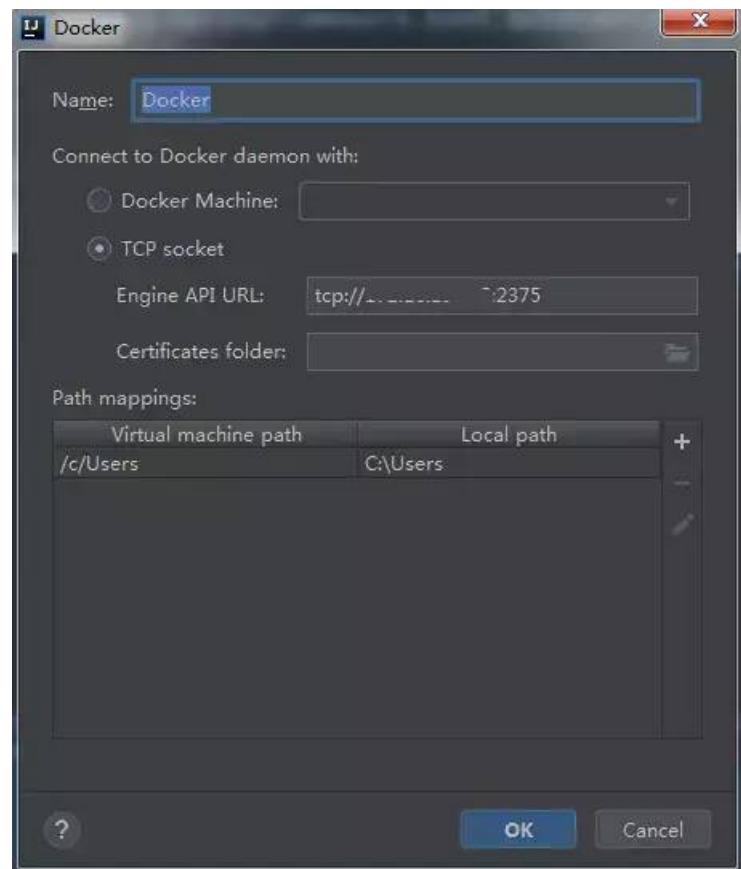


## 6. 连接远程docker

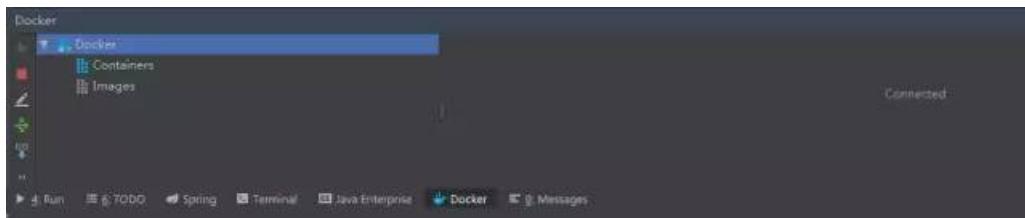
### 1、编辑配置



### 2、填远程docker地址



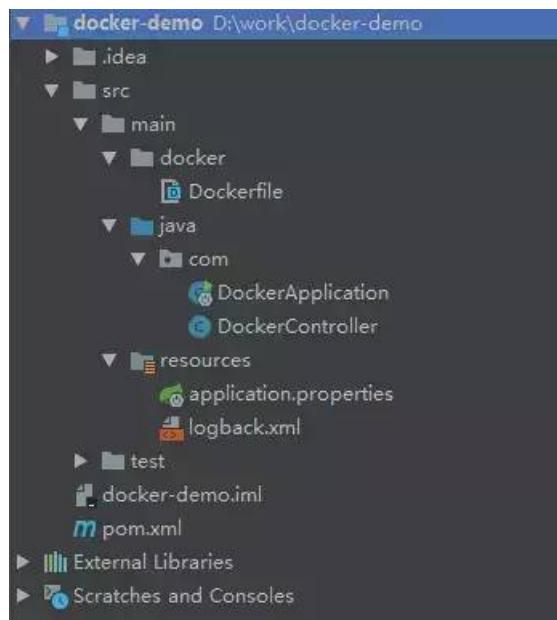
### 3、连接成功，会列出远程docker容器和镜像



## 二、新建项目

### 创建Spring Boot项目

#### 项目结构图



#### 1、配置pom文件

```
1 <properties>
2 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3 >
4 <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
5 >
6 <docker.image.prefix>com.demo</docker.image.prefix>
7 >
8 <java.version>1.8</java.version>
9 >
10 </properties>
11 >
12 <build>
13 >
14 <plugins>
15 >
16 <plugin>
17 >
18 <groupId>org.springframework.boot</groupId>
19 >
20 <artifactId>spring-boot-maven-plugin</artifactId>
21 >
```

```
21 </plugin>
22 >
23 <plugin>
24 >
25 <groupId>com.spotify</groupId>
26 >
27 <artifactId>docker-maven-plugin</artifactId>
28 >
29 <version>1.0.0</version>
30 >
31 <configuration>
32 >
33 <dockerDirectory>src/main/docker</dockerDirectory>
34 >
35 <resources>
36 >
37 <resource>
38 >
39 <targetPath>/</targetPath>
40 >
41 <directory>${project.build.directory}</directory>
42 >
43 <include>${project.build.finalName}.jar</include>
44 >
45 </resource>
46 >
47 </resources>
48 >
49 </configuration>
50 >
51 </plugin>
52 >
53 <plugin>
54 >
55 <artifactId>maven-antrun-plugin</artifactId>
56 >
57 <executions>
58 >
59 <execution>
60 >
61 <phase>package</phase>
62 >
63 <configuration>
64 >
65 <tasks>
66 >
67 <copy todir="src/main/docker" file="target/${project.artifactId}-${prc
68 >
69 </tasks>
70 >
71 </configuration>
72 >
73 <goals>
74 >
```

```
<goal>run</goal>
>
 </goals>
>
 </execution>
>
 </executions>
>
 </plugin>
>

 </plugins>
>
 </build>
>
<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>
>
 <dependency>
>
 <groupId>log4j</groupId>
>
 <artifactId>log4j</artifactId>
>
 <version>1.2.17</version>
>
 </dependency>
>
</dependencies>
```

## 2、在src/main目录下创建docker目录，并创建Dockerfile文件

```
1 FROM openjdk:8-jdk-alpine
2 ADD *.jar app.jar
3 ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/.urandom", "-jar", "/app.jar"]
```

### 3、在resource目录下创建application.properties文件

```
1 logging.config=classpath:logback.xml
2 logging.path=/home/developer/app/logs/
3 server.port=8990
```

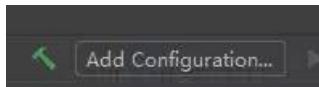
### 4、创建DockerApplication文件

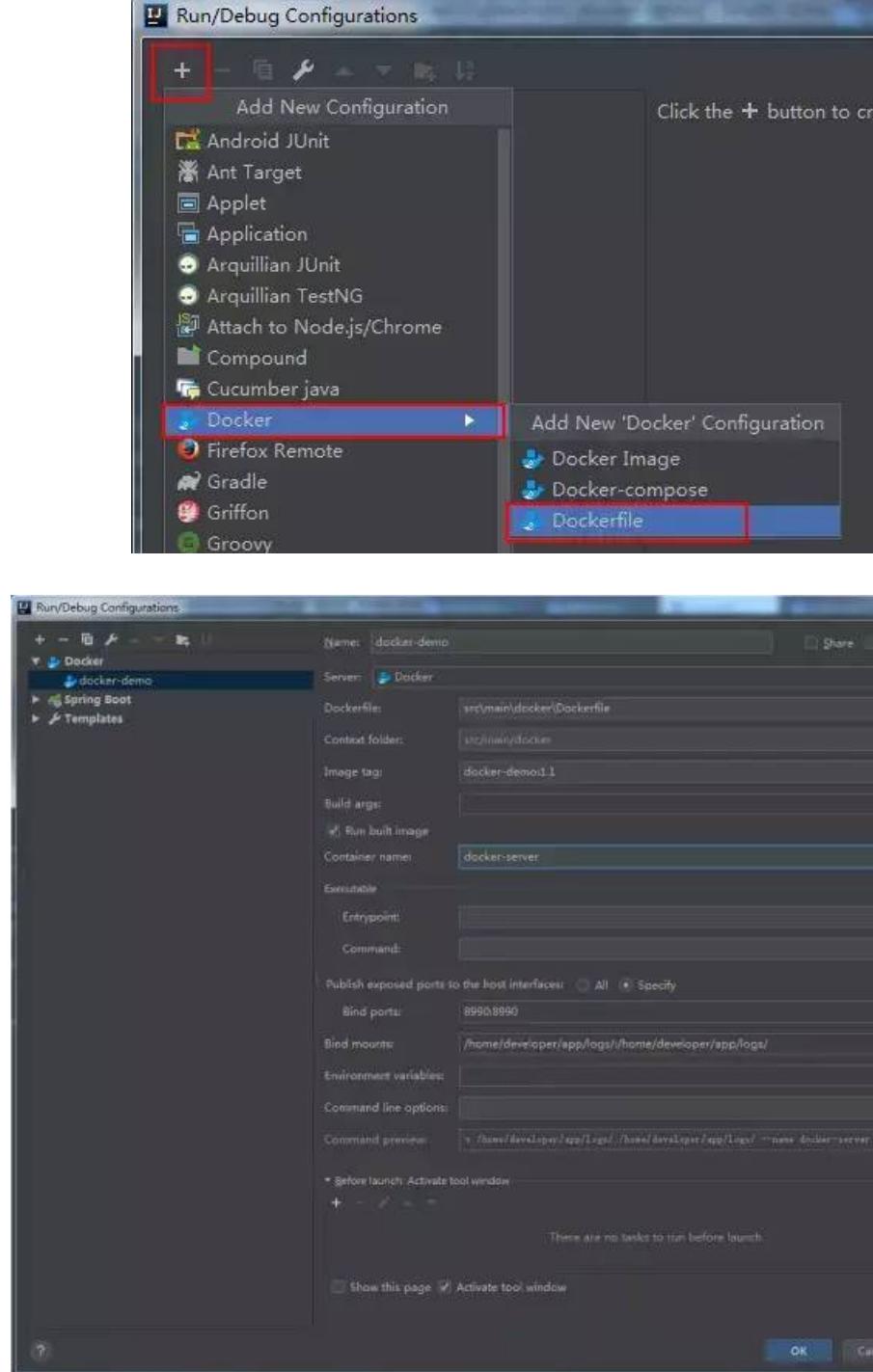
```
1 @SpringBootApplication
2 public class DockerApplication {
3 public static void main(String[] args)
4 {
5 SpringApplication.run(DockerApplication.class, args);
6 }
7 }
```

### 5、创建DockerController文件

```
1 @RestController
2 public class DockerController {
3 static Log log = LogFactory.getLog(DockerController.class)
4 ;
5
6 @RequestMapping("/")
7 public String index() {
8 log.info("Hello Docker!")
9 ;
10 return "Hello Docker!"
11 }
12 }
```

### 6、增加配置



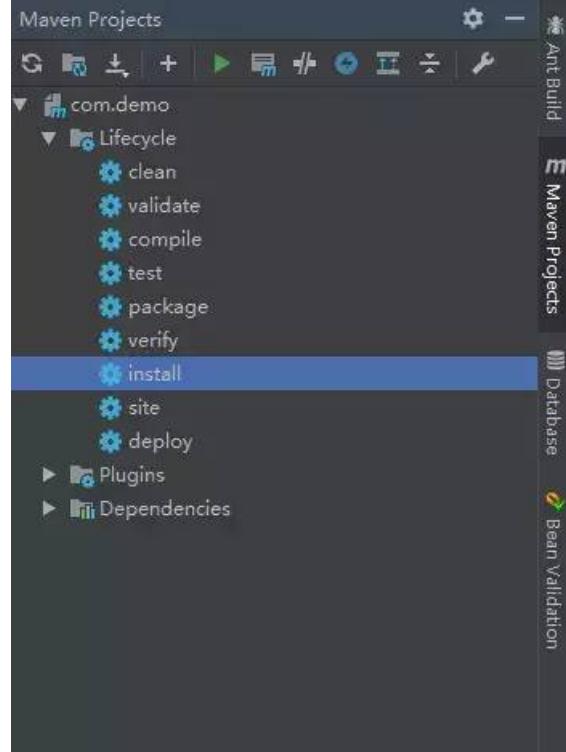


### 命令解释：

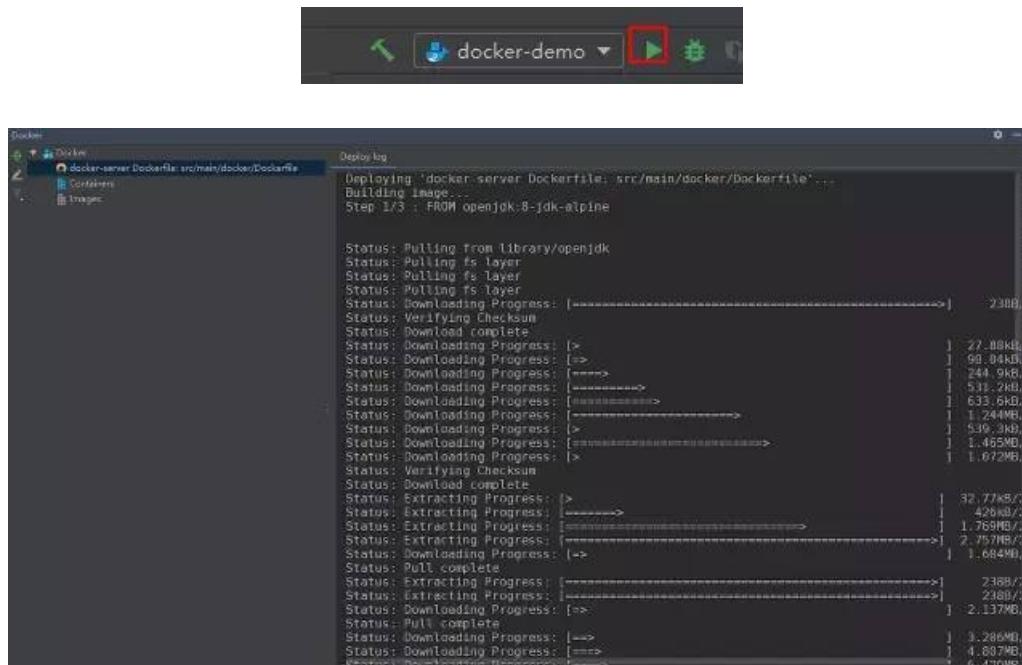
- Image tag : 指定镜像名称和tag，镜像名称为 docker-demo，tag为1.1
- Bind ports : 绑定宿主机端口到容器内部端口。格式为[宿主机端口]:[容器内部端口]
- Bind mounts : 将宿主机目录挂载到容器内部目录中。

格式为[宿主机目录]:[容器内部目录]。这个springboot项目会将日志打印在容器 /home/developer/app/logs/ 目录下，将宿主机目录挂载到容器内部目录后，那么日志就会持久化容器外部的宿主机目录中。

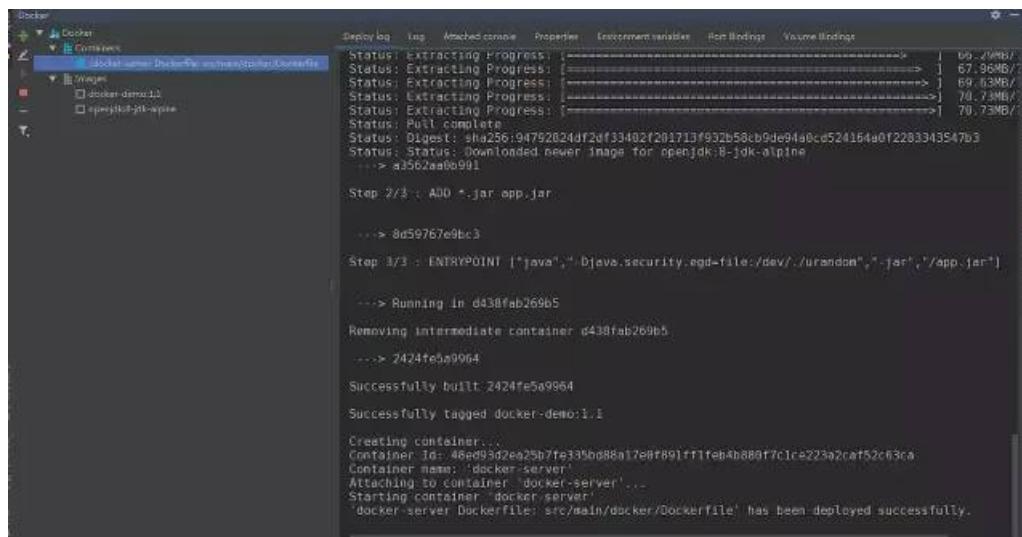
## 7、Maven打包



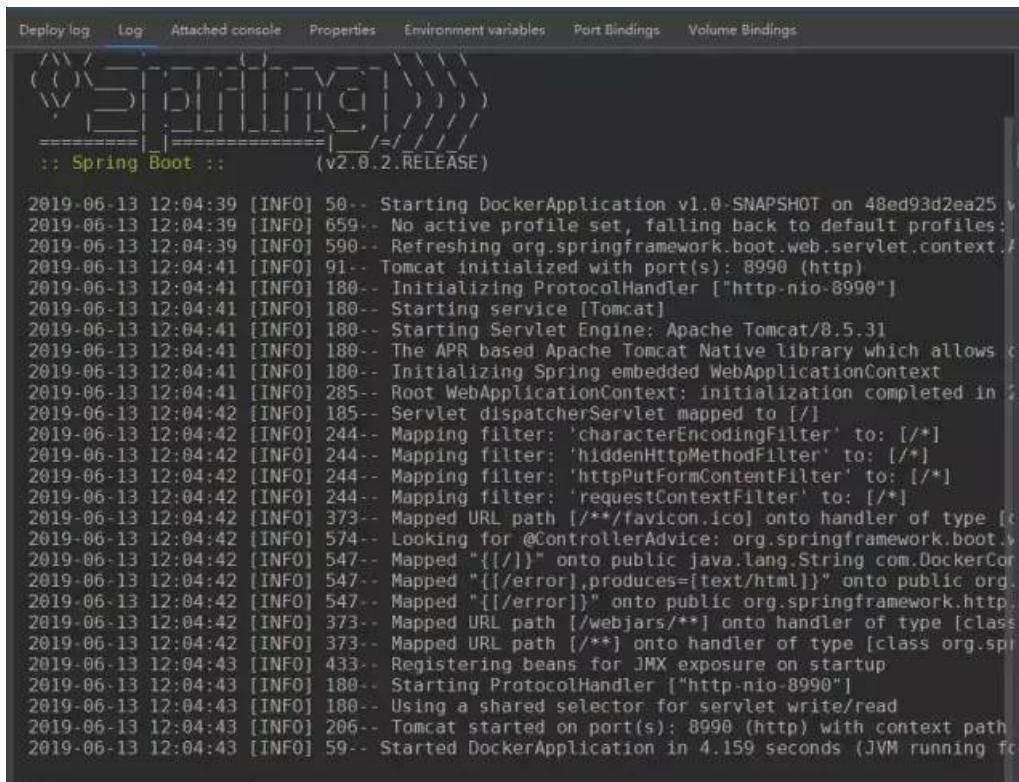
## 8、运行



先pull基础镜像，然后再打包镜像，并将镜像部署到远程docker运行



## 9、运行成功



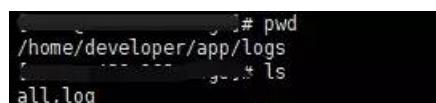
The screenshot shows the Docker tool window in IDEA. The 'Log' tab is selected, displaying the application's startup logs. The logs indicate the application is starting on port 8990, initializing Tomcat, and mapping various URL paths to handlers. The log output ends with the message 'Started DockerApplication in 4.159 seconds (JVM running for 4.159 seconds)'.

```
2019-06-13 12:04:39 [INFO] 50-- Starting DockerApplication v1.0-SNAPSHOT on 48ed93d2ea25 with PID 1 in directory /app
2019-06-13 12:04:39 [INFO] 659-- No active profile set, falling back to default profiles: []
2019-06-13 12:04:39 [INFO] 590-- Refreshing org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@590535e: startup date [2019-06-13T12:04:39.144+0000]; root of context hierarchy
2019-06-13 12:04:41 [INFO] 91-- Tomcat initialized with port(s): 8990 (http)
2019-06-13 12:04:41 [INFO] 180-- Initializing ProtocolHandler ["http-nio-8990"]
2019-06-13 12:04:41 [INFO] 180-- Starting service [Tomcat]
2019-06-13 12:04:41 [INFO] 180-- Starting Servlet Engine: Apache Tomcat/8.5.31
2019-06-13 12:04:41 [INFO] 180-- The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: [/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.232-b09-0ubuntu1~16.04.1-amd64/lib/amd64]
2019-06-13 12:04:41 [INFO] 180-- Initializing Spring embedded WebApplicationContext
2019-06-13 12:04:41 [INFO] 285-- Root WebApplicationContext: initialization completed in 2 ms
2019-06-13 12:04:42 [INFO] 185-- Servlet dispatcherServlet mapped to [/]
2019-06-13 12:04:42 [INFO] 244-- Mapping filter: 'characterEncodingFilter' to: [//*]
2019-06-13 12:04:42 [INFO] 244-- Mapping filter: 'hiddenHttpMethodFilter' to: [//*]
2019-06-13 12:04:42 [INFO] 244-- Mapping filter: 'httpPutFormContentFilter' to: [//*]
2019-06-13 12:04:42 [INFO] 244-- Mapping filter: 'requestContextFilter' to: [//*]
2019-06-13 12:04:42 [INFO] 373-- Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.boot.web.servlet.error.ErrorMvcHandler]
2019-06-13 12:04:42 [INFO] 574-- Looking for @ControllerAdvice: org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration$EnableWebMvcConfiguration
2019-06-13 12:04:42 [INFO] 547-- Mapped "={"/}" onto public java.lang.String com.DockerController.error()
2019-06-13 12:04:42 [INFO] 547-- Mapped "[{/error}], produces=[text/html]}" onto public org.springframework.http.ResponseEntity<org.springframework.web.util.ErrorPage> com.DockerController.error(javax.servlet.http.HttpServletRequest)
2019-06-13 12:04:42 [INFO] 547-- Mapped "[{/error}]" onto public org.springframework.http.ResponseEntity<org.springframework.web.util.ErrorPage> com.DockerController.error()
2019-06-13 12:04:42 [INFO] 373-- Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.handler.BeanNameUrlHandler]
2019-06-13 12:04:42 [INFO] 373-- Mapped URL path [/**] onto handler of type [class org.springframework.boot.web.servlet.error.ErrorMvcHandler]
2019-06-13 12:04:43 [INFO] 433-- Registering beans for JMX exposure on startup
2019-06-13 12:04:43 [INFO] 180-- Starting ProtocolHandler ["http-nio-8990"]
2019-06-13 12:04:43 [INFO] 180-- Using a shared selector for servlet write/read
2019-06-13 12:04:43 [INFO] 206-- Tomcat started on port(s): 8990 (http) with context path /
2019-06-13 12:04:43 [INFO] 59-- Started DockerApplication in 4.159 seconds (JVM running for 4.159 seconds)
```

## 10、浏览器访问



## 11、日志查看



```
[# pwd
/home/developer/app/logs
[# ls
all.log
```

自此，通过IDEA部署Spring Boot项目到Docker成功！难以想象，部署一个Java web项目竟然如此简单方便！

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



扫一扫上面的二维码图案，加我微信

## 推荐阅读

1. Java后端优质文章整理
2. 7 个开源的 Spring Boot 前后端分离项目
3. 如何设计 API 接口, 实现统一格式返回?
4. 花 20 分钟, 再来梳理一下 Git 基础知识
5. 在 Spring Boot 中, 如何干掉 if else



Java后端

长按识别二维码，关注我的公众号

喜欢文章，点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 收藏了！7个开源的 Spring Boot 前后端分离优质项目

松哥 Java后端 2019-09-28

点击上方 Java后端, 选择“[设为星标](#)”

优质文章, 及时送达

作者 | 松哥

来源 | 江南一点雨 (id: a\_javaboy)

上篇 | 在 Spring Boot 中, 如何干掉 if else

前后端分离已经在慢慢走进各公司的技术栈，不少公司都已经切换到这个技术栈上面了。即使贵司目前没有切换到这个技术栈上面，松哥也非常建议大家学习一下前后端分离开发，以免在公司干了两三年，SSH 框架用的滚瓜烂熟，出来却发现依然没有任何优势！

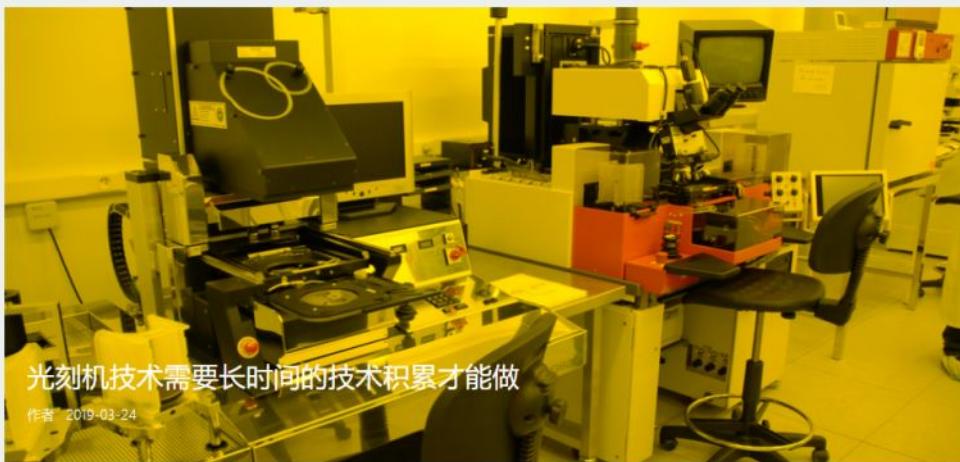
其实前后端分离本身并不难，后段提供接口，前端做数据展示，关键是这种思想。很多人做惯了前后端不分的开发，在做前后端分离的时候，很容易带进来一些前后端不分时候的开发思路，结果做出来的产品不伦不类，因此松哥这里给大家整理了几个开源的前后端分离项目，帮助大家快速掌握前后端分离开发技术栈。

## 美人鱼

- star 数 3499
- 项目地址：<https://gitee.com/mumu-osc/NiceFish>

听名字就知道这是个不错的项目，事实上确实不赖。NiceFish（美人鱼）是一个系列项目，目标是示范前后端分离的开发模式：前端浏览器、移动端、Electron 环境中的各种开发模式；后端有两个版本：SpringBoot 版本和 SpringCloud 版本，前端有 Angular、React 以及 Electron 等版本。

项目效果图：



QQ群-请勿加多个  
 ● Angular-1区:286047042  
 ● Angular-2区:139357161  
 ● Angular-3区:473129930  
 ● Angular-4区:483016484  
 ● Angular-5区:604253120  
 ● Angular-6区:124641447  
 ● 脚本娃娃-桃花岛:83163037



### 从头了解光刻机

● 大漠穷秋 ○ 2018-05-17 10:44

光刻是集成电路最重要的加工工艺，他的作用，如同金工车间中车床的作用。在整个芯片制造工艺中，几乎每个工艺的实施，都离不开光刻的技术。光刻也是制造芯片的最关键技术，他占芯片制造成本的35%以上。在如今的科技与社会发展中，光刻技术的增长，直接关系到大型计算机的运作等高科技领域。<p>测...



### 从头了解光刻机

● 大漠穷秋 ○ 2018-05-17 10:44

光刻是集成电路最重要的加工工艺，他的作用，如同金工车间中车床的作用。在整个芯片制造工艺中，几乎每个工艺的实施，都离不开光刻的技术。光刻也是制造芯片的最关键技术，他占芯片制造成本的35%以上。在如今的科技与社会发展中，光刻技术的增长，直接关系到大型计算机的运作等高科技领域。...



### 从头了解光刻机

● 大漠穷秋 ○ 2018-05-17 10:44

光刻是集成电路最重要的加工工艺，他的作用，如同金工车间中车床的作用。在整个芯片制造工艺中，几乎每个工艺的实施，都离不开光刻的技术。光刻也是制造芯片的最关键技术，他占芯片制造成本的35%以上。在如今的科技与社会发展中，光刻技术的增长，直接关系到大型计算机的运作等高科技领域。...

## 微人事

- star 数 9313
- 项目地址: <https://github.com/lenvve/vhr>

微人事是一个前后端分离的人力资源管理系统，项目采用 SpringBoot + Vue 开发。项目打通了前后端，并且提供了非常详尽的文档，从 Spring Boot 接口设计到前端 Vue 的开发思路，作者全部都记录在项目的 wiki 中，是不可多得的 Java 全栈学习资料。

项目效果图:

项目部分文档截图：

GitHub, Inc. [US] | <https://github.com/lenve/vhr/wiki>

# Home

江南一点雨 edited this page on 5 Feb 2018 · 25 revisions

文档是对项目开发过程中遇到的一些问题的详细记录，主要是为了帮助没有基础的小伙伴快速理解这个项目。

- 1.权限数据库设计
- 2.服务端环境搭建
- 3.动态处理角色和资源的关系
- 4.密码加密并加盐
- 5.服务端异常的统一处理
- 6.axios请求封装,请求异常统一处理
- 7.将请求方法挂到Vue上
- 8.登录状态的保存
- 9.登录成功后动态加载组件
- 10.角色资源关系管理
- 11.用户角色关系管理
- 12.部门数据库设计与存储过程编写
- 13.递归查询与存储过程调用
- 14.Tree树形控件使用要点
- 15.职位管理和职称管理功能介绍
- 16.组件复用
- 17.[题外话]利用git标签回退至任意版本
- 18.员工基本信息管理功能介绍
- 19.SpringBoot中自定义参数绑定
- 20.高级搜索功能介绍
- 21.Excel导入导出效果图

## bootshiro

- star 数 1370
- 项目地址：<https://gitee.com/tomsun28/bootshiro>

bootshiro 是基于 Spring Boot + Shiro + JWT 的真正 RESTful URL 资源无状态认证权限管理系统的后端,前端 usthe 。区别于一般项目,该项目提供页面可配置式的、动态的 RESTful api 安全管理支持,并且实现数据传输动态秘钥加密,jwt 过期刷新,用户操作监控等,加固应用安全。

项目效果图：

The screenshot shows a list of REST APIs:

ID	名称	方法	URI	访问方式	状态
101	用户登录	ACCOUNT_LOGIN	/account/login	POST	正常
112	用户注册	ACCOUNT_REGISTER	/account/register	POST	正常
119	获取对应用户角色	USER_ROLE_APPID	/user/role	GET	正常
120	获取用户列表	USER_LIST	/user/list	GET	正常
121	给用户授权添加角色	USER_AUTHORITY_ROLE	/user/authority/role	POST	正常
122	删除已经授权的用户角色	USER_AUTHORITY_ROLE	/user/authority/role	DELETE	正常
123	获取用户被授权菜单	RESOURCE_AUTORITYMENU	/resource/authorityMenu	GET	正常
124	获取全部菜单列	RESOURCE_MENUS	/resource/menus	GET	正常
125	增加菜单	RESOURCE_MENU	/resource/menu	POST	正常
126	修改菜单	RESOURCE_MENU	/resource/menu	PUT	正常

选中资源：删除已经授权的用户角色

## open-capacity-platform

- star 数 2643
- 项目地址：<https://gitee.com/owenwangwen/open-capacity-platform>

open-capacity-platform 微服务能力开放平台，简称 ocp，是基于 layui + springcloud 的企业级微服务框架(用户权限管理，配置中心管理，应用管理，....)，其核心的设计目标是分离前后端，快速开发部署，学习简单，功能强大，提供快速接入核心接口能力，其目标是帮助企业搭建一套类似百度能力开放平台的框架。

## 项目效果图：

The screenshot shows a list of permissions:

权限名称	权限标识	创建时间	操作
获取文件列表	file:query	2019-05-17 21:34:05	[修改] [删除]
根据roleId获取权限	permission:get/permissions/{roleId}/permissions	2019-05-10 00:22:23	[修改] [删除]
获取菜单以及顶层菜单	menu:get/menus/findOnes	2019-05-09 23:48:13	[修改] [删除]
服务树	service:get/service/findOnes	2018-01-18 17:06:39	[修改] [删除]
查询所有服务	service:get/service/findAlls	2018-01-18 17:06:39	[修改] [删除]
删除应用	clientdelete/clients	2018-01-18 17:06:39	[修改] [删除]
根据id获取应用	client:get/clients/{id}	2018-01-18 17:06:39	[修改] [删除]
应用列表	client:get/clients	2018-01-18 17:06:39	[修改] [删除]
保存应用	client:post/clients	2018-01-18 17:06:39	[修改] [删除]
所有菜单	menu:get/menus/findAlls	2018-01-18 17:06:39	[修改] [删除]

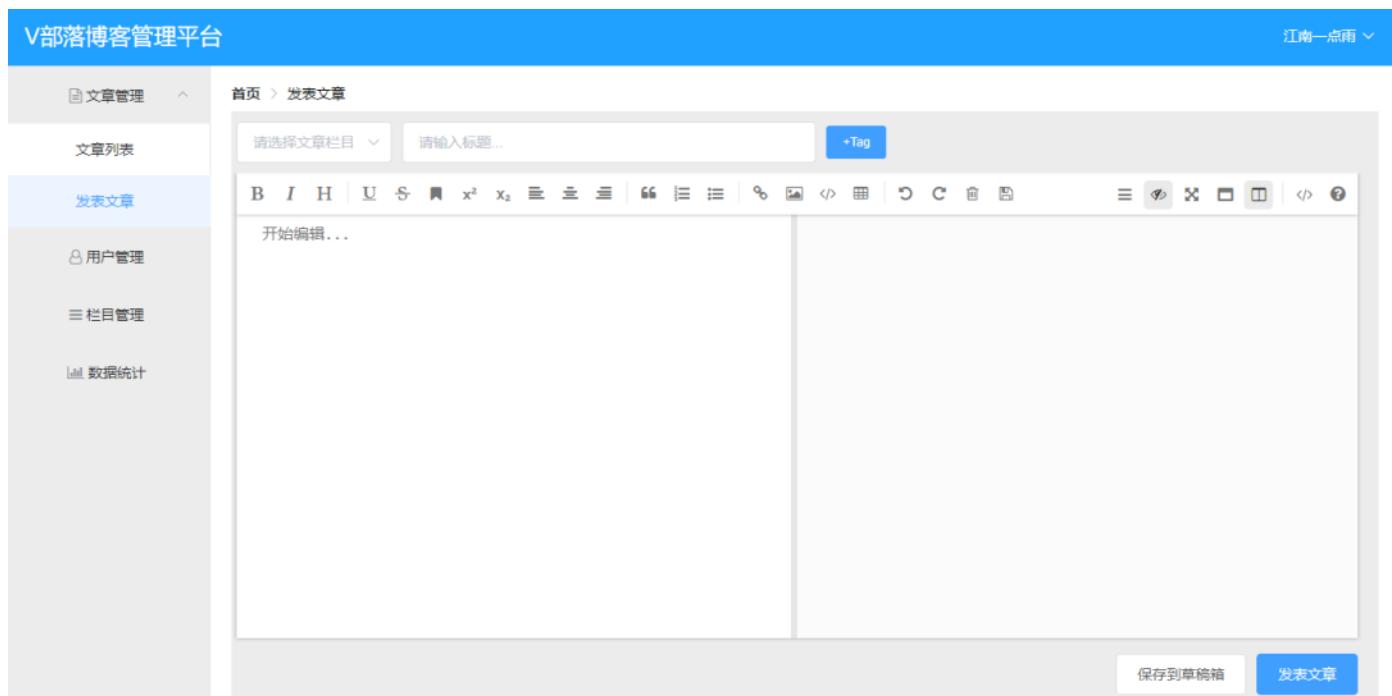
Copyright © 2018 JeeCp All rights reserved.

## V部落

- star 数 2902
- 项目地址：<https://github.com/lenve/VBlog>

V部落是一个多用户博客管理平台，采用 Vue + SpringBoot + ElementUI 开发。这个项目最大的优势是简单，属于功能完整但是又非常简单的那种，非常适合初学者。

项目效果图：

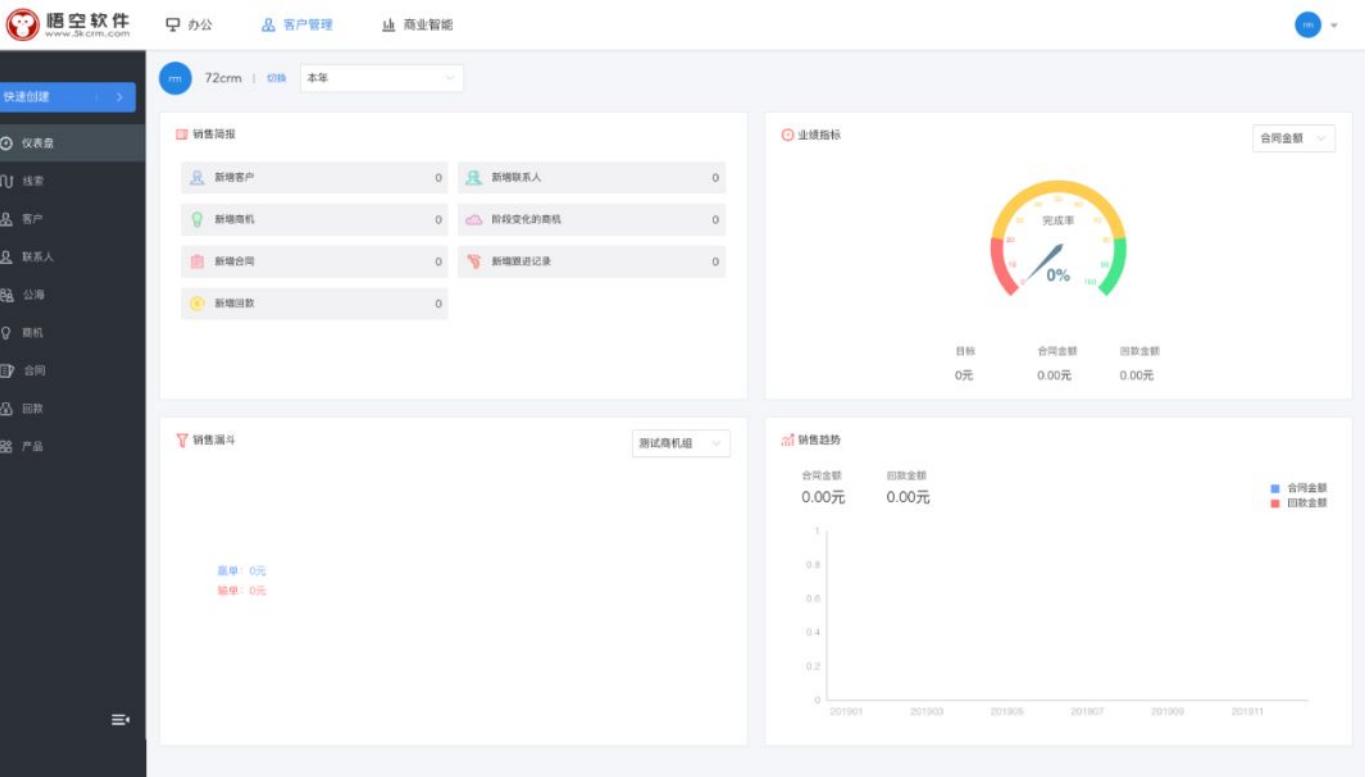


## 悟空 CRM

- star 数 650
- 项目地址：<https://gitee.com/wukongcrm/72crm-java>

悟空 CRM 是基于 jfinal + vue + ElementUI 的前后端分离 CRM 系统。

老实说，jfinal 了解下就行了，没必要认真研究，Vue + ElementUI 的组合可以认真学习下、前后端交互的方式可以认真学习下。



## paascloud-master

- star 数 5168
- [github.com/paascloud/paascloud-master](https://github.com/paascloud/paascloud-master)

paascloud-master 核心技术为 SpringCloud + Vue 两个全家桶实现,采取了取自开源用于开源的目标,所以能用开源绝不用收费框架,整体技术栈只有阿里云短信服务是收费的,都是目前 java 前瞻性的框架,可以为中小企业解决微服务架构难题,可以帮助企业快速建站。

由于服务器成本较高,尽量降低开发成本的原则,本项目由 10 个后端项目和 3 个前端项目共同组成。真正实现了基于 RBAC、jwt 和 oauth2 的无状态统一权限认证的解决方案,实现了异常和日志的统一管理,实现了 MQ 落地保证 100% 到达的解决方案。关注微信公众号 Java后端 获取更多推送。

项目效果图:

用户列表											
筛选条件		排序条件		操作		数据统计					
模块	操作	姓名	状态	联系电话	工号	岗位	部门名称	请假时间	备注时间	上次登录时间	操作
1. 操作监控	1	5628154019824c0942119f9997026eae	启用	123456789012345	5628154019824c0942119f9997026eae	江海一局(www.5628154019824c0942119f9997026eae.org)	5628154019824c0942119f9997026eae	2019-06-10 10:00:00	2019-06-10 20:00:00	2019-06-10 20:00:00	禁用
2. 操作日志监控	2	5628154019824c0942119f9997026eae	启用	123456789012345	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	2019-06-10 10:00:00	2019-06-10 20:00:00	2019-06-10 20:00:00	禁用
3. 系统管理	3	5628154019824c0942119f9997026eae	启用	123456789012345	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	2019-06-10 10:00:00	2019-06-10 20:00:00	2019-06-10 20:00:00	禁用
4. 用户管理	4	5628154019824c0942119f9997026eae	启用	123456789012345	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	2019-06-10 10:00:00	2019-06-10 20:00:00	2019-06-10 20:00:00	禁用
5. 角色管理	5	5628154019824c0942119f9997026eae	启用	123456789012345	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	2019-06-10 10:00:00	2019-06-10 20:00:00	2019-06-10 20:00:00	禁用
6. 权限管理	6	5628154019824c0942119f9997026eae	启用	123456789012345	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	2019-06-10 10:00:00	2019-06-10 20:00:00	2019-06-10 20:00:00	禁用
7. 菜单管理	7	5628154019824c0942119f9997026eae	启用	123456789012345	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	2019-06-10 10:00:00	2019-06-10 20:00:00	2019-06-10 20:00:00	禁用
8. 其他管理	8	5628154019824c0942119f9997026eae	启用	123456789012345	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	2019-06-10 10:00:00	2019-06-10 20:00:00	2019-06-10 20:00:00	禁用
9. 审核管理	9	5628154019824c0942119f9997026eae	启用	123456789012345	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	2019-06-10 10:00:00	2019-06-10 20:00:00	2019-06-10 20:00:00	禁用
10. 日志管理	10	5628154019824c0942119f9997026eae	启用	123456789012345	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	5628154019824c0942119f9997026eae	2019-06-10 10:00:00	2019-06-10 20:00:00	2019-06-10 20:00:00	禁用

共 4322 条 100 / 100

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 89 90 91 92 93 94 95 96 97 98 99 100

## 总结

他山之石，可以攻玉。当我们学会了很多知识点之后，需要一个项目来将这些知识点融会贯通，这些开源项目就是很好的资料。现在前后端分离开发方式日渐火热，松哥也强烈建议大家有空学习下这种开发方式。虽然我们身为 Java 工程师，可是也不能固步自封，看看前端单页面应用怎么构建，看看前端工程化是怎么回事，这些都有助于我们开发出更加合理好用的后端接口。好了，七个开源项目，助力大家在全栈的路上更进一步！

- END -

如果看到这里，说明你喜欢这篇文章，请[转发](#)、[点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



1. Java后端优质文章整理
2. 在 Spring Boot 中,如何干掉 if else
3. Java 性能优化:教你提高代码运行的效率
4. 在浏览器输入 URL 回车之后发生了什么?
5. 接私活必备的 10 个开源项目



Java后端

长按识别二维码，关注我的公众号

喜欢文章，点个在看

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 盘点一下企业最常用的几个 Spring Boot Starter

SimpleWu Java后端 2019-08-23

点击上方**蓝色字体**, 选择“标星公众号”

优质文章, 第一时间送达

来自 | SimpleWu

链接 | [www.cnblogs.com/SimpleWu/p/9798146.html](http://www.cnblogs.com/SimpleWu/p/9798146.html)

## Spring Boot 简介

Spring Boot是由Pivotal团队提供的全新框架, 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, 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。

Spring Boot让我们的Spring应用变的更轻量化。比如:你可以仅仅依靠一个Java类来运行一个Spring引用。你也可以打包你的应用为jar并通过使用java -jar来运行你的Spring Web应用。

Spring Boot的主要优点:

- 为所有Spring开发者更快的入门
- 开箱即用, 提供各种默认配置来简化项目配置
- 内嵌式容器简化Web项目
- 没有冗余代码生成和XML配置的要求

在下面的代码中只要有一定基础会发现这写代码实例非常简单对于开发者来说几乎是“零配置”。

## SpringBoot运行

开发工具: jdk8, IDEA, STS, eclipse(需要安装STS插件) 这些都支持快速启动SpringBoot工程。我这里就不快速启动了, 使用maven工程。学习任何一项技术首先就要精通HelloWord, 那我们来跑个初体验。

首先只用maven我们创建的maven工程直接以jar包的形式创建就行了, 首先我们来引入SpringBoot的依赖

首先我们需要依赖SpringBoot父工程, 这是每个项目中必须要有的。

```
1 <!-- 引入SpringBoot父依赖 欢迎关注公众号 Web项目聚集地-->
2 <parent>
3 <groupId>org.springframework.boot</groupId>
4 <artifactId>spring-boot-starter-parent</artifactId>
5 <version>2.0.5.RELEASE</version>
6 <relativePath/>
7 </parent>
8 <!-- 编码与JAVA版本-->
9 <encoding>UTF-8</encoding>
10 <build>
11 <plugins>
```

```
12 <properties>
13 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
14 >
15 <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
16 >
17 <java.version>1.8</java.version>
18 >
19 </properties>
```

我们启动WEB模块当然必须要引入WEB模块的依赖

```
1 <dependencies>
2 <!-- 引入SpringBoot-WEB模块 欢迎关注公众号 Web项目聚集地-->
3 <dependency>
4 <groupId>org.springframework.boot</groupId>
5 >
6 <artifactId>spring-boot-starter-web</artifactId>
7 >
8 </dependency>
9 >
10 </dependencies>
11 >
```

我们需要编写一个SpringBoot启动类, SpringbootFirstExperienceApplication.java

```
1 @SpringBootApplication
2 public class SpringbootFirstExperienceApplication {
3
4 public static void main(String[] args)
5 {
6 SpringApplication.run(SpringbootFirstExperienceApplication.class, args);
7 }
8 }
```

到了这里我们直接把他当成SpringMVC来使用就行了, 不过这里默认是不支持JSP官方推荐使用模板引擎, 后面会写到整合JSP。这里我就不写Controller了。

@SpringBootApplication:之前用户使用的是3个注解注解他们的main类。分别是

@Configuration,@EnableAutoConfiguration,@ComponentScan。由于这些注解一般都是一起使用,spring boot提供了一个统一的注解@SpringBootApplication。

注意事项:我们使用这个注解在不指定扫描路径的情况下, SpringBoot只能扫描到和 SpringbootFirstExperienceApplication同包或子包的Bean;

## SpringBoot目录结构

在src/main/resources中我们可以有几个文件夹:

- **templates**: 用来存储模板引擎的, Thymeleaf, FreeMarker, Velocity等都是不错的选择。
- **static**: 存储一些静态资源, css, js等
- **public**: 在默认SpringBoot工程中是不生成这个文件夹的,但是在自动配置中我们可以有这个文件夹用来存放公共的资源(html等)
- **application.properties**: 这个文件名字是固定的, SpringBoot启动会默认加载这些配置在这里面可以配置端口号, 访问路径, 数据库连接信息等等。这个文件非常重要,当然官方中推出了一个yml格式这是非常强大的数据格式。

## 整合JdbcTemplate

引入依赖:

```
1 <parent>
2 <groupId>org.springframework.boot</groupId>
3 >
4 <artifactId>spring-boot-starter-parent</artifactId>
5 >
6 <version>1.5.2.RELEASE</version>
7 >
8 </parent>
9 >
10 <dependencies>
11 >
12 <!-- 引入WEB模块-->
13 <dependency>
14 >
15 <groupId>org.springframework.boot</groupId>
16 >
17 <artifactId>spring-boot-starter-web</artifactId>
18 >
19 </dependency>
20 >
21 <!-- 引入JDBC模块-->
22 <dependency>
23 >
24 <groupId>org.springframework.boot</groupId>
25 >
26 <artifactId>spring-boot-starter-jdbc</artifactId>
27 >
28 </dependency>
29 >
30 <!-- 引入数据库驱动-->
31 >
32 <dependency>
33 >
34 <groupId>mysql</groupId>
35 >
36 <artifactId>mysql-connector-java</artifactId>
37 >
38 </dependency>
39 >
```

```

> <dependency>
> <groupId>org.springframework.boot</groupId>
> <artifactId>spring-boot-starter-test</artifactId>
> <scope>test</scope>
> </dependency>
> </dependencies>
>

```

配置application.properties，虽然说是“零配置”但是这些必要的肯定要指定，否则它怎么知道连那个数据库？

```

1 spring.datasource.url=jdbc:mysql://localhost:3306/mybatis
2 spring.datasource.username=root
3 spring.datasource.password=root
4 spring.datasource.driver-class-name=com.mysql.jdbc.Driver

```

使用方式：

```

1 @Service
2 public class EmployeeService {
3 @Autowired
4 private JdbcTemplate jdbcTemplate;
5
6 public boolean saveEmp(String name, String email, String gender){
7 String sql = "insert into tal_employee values(null,?, ?, ?)"
8 ;
9 int result = jdbcTemplate.update(sql, name, email, gender);
10 System.out.println("result : " + result);
11 return result > 0 ? true:false
12 }
1 }

```

```

1 @RestController
2 public class EmployeeController {
3
4 @Autowired
5 private EmployeeService employeeService;
6
7 @RequestMapping("/save"
8)
9 public String insert(String name, String email, String gender)
10 {
11 boolean result = employeeService.saveEmp(name, email, gender);
12 if(result){

```

```
13 return "success"
14 }
15 return "error"
16 ;
17 }
18 }
```

这里我们直接返回一个文本格式。

## @RestController

在上面的代码中我们使用到这个注解修改我们的Controller类而是不使用@Controller这个注解，其实中包含了@Controller，同时包含@ResponseBody既然修饰在类上面那么就是表示这个类中所有的方法都是@ResponseBody所以在里我们返回字符串在前台我们会以文本格式展示，如果是对象那么它会自动转换成json格式返回。

## 整合JPA

同样的整合JPA我们只需要启动我们SpringBoot已经集成好的模块即可。

添加依赖：

```
1 <parent>
2 <groupId>org.springframework.boot</groupId>
3 >
4 <artifactId>spring-boot-starter-parent</artifactId>
5 >
6 <version>1.5.2.RELEASE</version>
7 >
8 </parent>
9 >
10 <dependencies>
11 >
12 <dependency>
13 >
14 <groupId>org.springframework.boot</groupId>
15 >
16 <artifactId>spring-boot-starter-web</artifactId>
17 >
18 </dependency>
19 >
20 <!-- 启动JPA组件-->
21 <dependency>
22 >
23 <groupId>org.springframework.boot</groupId>
24 >
25 <artifactId>spring-boot-starter-data-jpa</artifactId>
26 >
27 </dependency>
28 >
29 <dependency>
```

```
>
 <groupId>org.springframework.boot</groupId>
>
 <artifactId>spring-boot-starter-test</artifactId>
>
 <scope>test</scope>
>
</dependency>
>
<dependency>
>
 <groupId>mysql</groupId>
>
 <artifactId>mysql-connector-java</artifactId>
>
</dependency>
>
</dependencies>
>
```

启动JPA组件后直接配置数据库连接信息就可以使用JPA功能。

### Application.properties

```
1 spring.datasource.url=jdbc:mysql://localhost:3306/mybatis
2 spring.datasource.username=root
3 spring.datasource.password=root
4 spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

### 实体类:Employee.java

```
1 @Table(name="tal_employee")
2 @Entity
3 public class Employee implements Serializable{
4 @Id
5 @GeneratedValue(strategy = GenerationType.AUTO)
6 private Integer id;
7 @Column(name="last_Name")
8 }
9 private String lastName;
10 private String email;
11 private String gender;
12 //get set 省略
}
```

### EmployeeDao接口:

```
1 public interface EmployeeDao extends JpaRepository<Employee, Integer>{
2 }
```

## EmployeeController.java:

```
1 @Controller
2 public class EmployeeController {
3 @Autowired
4 private EmployeeDao employeeDao;
5
6 @ResponseBody
7 @RequestMapping("/emps"
8)
9 public List<Employee> getEmployees(){
10 List<Employee> employees = employeeDao.findAll();
11 System.out.println(employees);
12 return employees;
13 }
14 }
```

## 整合MyBatis

引入依赖：

```
1 <parent>
2 <groupId>org.springframework.boot</groupId
3 >
4 <artifactId>spring-boot-starter-parent</artifactId
5 >
6 <version>1.5.2.RELEASE</version
7 >
8 </parent>
9 >
10 <dependencies
11 >
12 <dependency>
13 >
14 <groupId>org.springframework.boot</groupId
15 >
16 <artifactId>spring-boot-starter-web</artifactId
17 >
18 </dependency>
19 >
20 <!-- 引入对JDBC的支持-->
21 <dependency>
22 >
23 <groupId>org.springframework.boot</groupId
24 >
25 <artifactId>spring-boot-starter-jdbc</artifactId
26 >
27 </dependency>
28 >
<!-- 引入对logging的支持-->
```

```

29 <dependency>
30 >
31 <groupId>org.springframework.boot</groupId>
32 >
33 <artifactId>spring-boot-starter-logging</artifactId>
34 >
35 </dependency>
36 >
37 <!-- SpringBoot MyBatis启动器 -->
38 <dependency>
39 >
40 <groupId>org.mybatis.spring.boot</groupId>
41 >
42 <artifactId>mybatis-spring-boot-starter</artifactId>
43 >
44 <version>1.2.2</version>
45 >
46 </dependency>
47 >
48 <dependency>
49 >
50 <groupId>org.springframework.boot</groupId>
51 >
52 <artifactId>spring-boot-starter-test</artifactId>
53 >
54 <scope>test</scope>
55 >
56 </dependency>
57 >
58 <dependency>
59 >
60 <groupId>mysql</groupId>
61 >
62 <artifactId>mysql-connector-java</artifactId>
63 >
64 </dependency>
65 >
66 </dependencies>
67 >

```

## 配置application.properties

```

1 spring.datasource.url=jdbc:mysql://localhost:3306/mybatis
2 spring.datasource.username=root
3 spring.datasource.password=root
4 spring.datasource.driver-class-name=com.mysql.jdbc.Driver
5 ##### datasource classpath 数据连接池地址#####
6 #spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
7
8 #指定我们的mapper.xml位置 欢迎关注公众号 Web项目聚集地
9 mybatis.mapper-locations=classpath:com/simple/springboot/mybatis/dao/mapper/*.xml
10 #entity.class 指定我们实体类所在包位置

```

当然这里还有很多属性如果想要使用可以参考官方文档。到了这里其他就不写了，把他当作SSM使用就ok。

注意事项：在我们的Dao层接口中一定要在类上加上注解@Mapper否则无法扫描到。

## AOP功能使用

在我们SpringBoot中使用AOP非常简单。

```

1 @Aspect
2 @Component
3 public class SpringBootAspect {
4
5 /**
6 * 定义一个切入点
7 * @author:SimpleWu
8 * @Date:2018年10月12日
9 */
10 @Pointcut(value="execution(* com.simple.springboot.util.*.*(..))")
11)
12 public void aop(){
13 }
14
15 /**
16 * 定义一个前置通知
17 * @author:SimpleWu
18 * @Date:2018年10月12日
19 */
20 @Before("aop()")
21)
22 public void aopBefore()
23 {
24 System.out.println("前置通知 SpringBootAspect....aopBefore")
25 ;
26 }
27
28 /**
29 * 定义一个后置通知 欢迎关注公众号 Web项目聚集地
30 * @author:SimpleWu
31 * @Date:2018年10月12日
32 */
33 @After("aop()")
34)
35 public void aopAfter()
36 {
37 System.out.println("后置通知 SpringBootAspect....aopAfter")
38 ;
39 }
40
41 /**
42 * 处理未处理的JAVA异常

```

```

42 * @author:SimpleWu
43 * @Date:2018年10月12日
44 */
45 @AfterThrowing(pointcut="aop()", throwing="e"
46)
47 public void exception(Exception e)
48 {
49 System.out.println("异常通知 SpringBootAspect...exception .." + e);
50 }
51
52 /**
53 * 环绕通知
54 * @author:SimpleWu
55 * @throws Throwable
56 * @Date:2018年10月12日
57 */
58 @Around("aop()")
59
60 public void around(ProceedingJoinPoint invocation) throws Throwable
61 {
62 System.out.println("SpringBootAspect..环绕通知 Before")
63 ;
64 invocation.proceed();
65 System.out.println("SpringBootAspect..环绕通知 After")
66 ;
67 }
68
69 }

```

## 任务调度

SpringBoot已经集成好一个调度功能。

```

1 @Component
2 public class ScheduledTasks {
3 private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss")
4 ;
5
6 /**
7 * 任务调度，每隔5秒执行一次，欢迎关注公众号 Web项目聚集地
8 * @author:SimpleWu
9 * @Date:2018年10月12日
10 */
11
12 @Scheduled(fixedRate = 1000
13)
14 public void reportCurrentTime()
15 {
16 System.out.println("现在时间：" + dateFormat.format(new Date()));
17 }
18
19 }

```

```
1 /**
2 * SpringBoot 使用任务调度
3 * @EnableScheduling 标注程序开启任务调度
4
5 * @author :SimpleWu
6 * @Date :2018年10月12日
7 */
8
9 @SpringBootApplication
10 @EnableScheduling
11 public class App {
12 public static void main(String[] args)
13 {
14 SpringApplication.run(App.class, args);
15 }
16 }
```

## 仓库地址

本文所有测试代码实例:a

<https://gitlab.com/450255266/code>

喜欢文章, 点个在看

[阅读原文](#)

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

# 记一次 Spring Boot 项目启动卡住问题排查记录

陈凯玲 Java后端 2019-12-02

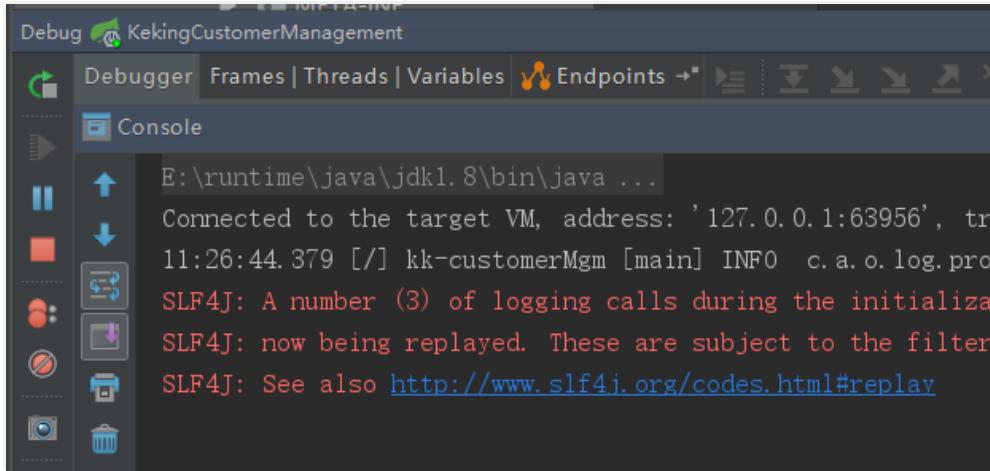
点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 陈凯玲

来源 | [my.oschina.net/keking/blog/3058921](http://my.oschina.net/keking/blog/3058921)

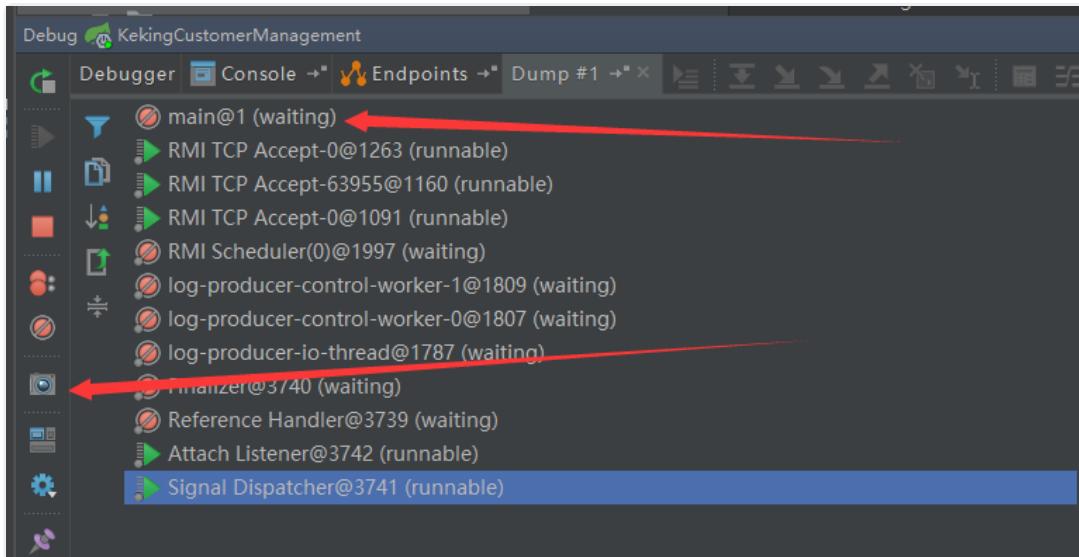
一个 Spring Boot 开发的项目, Spring Boot 版本是 1.5.7, 携带的 spring 版本是 4.1.3。开发反馈, 突然在本地启动不起来了, 表象特征就是在本地 IDEA 上运行时, 进程卡住也不退出, 应用启动时加载相关组件的日志也不输出。症状如下图:



## 问题分析

因为没有有用的日志信息, 所以不能从日志这个层面上排查问题。但是像这种没有输出日志的话, 一般情况下, 肯定是程序内部启动流程卡在什么地方了, 只能通过打印下当前线程堆栈信息了解下。一般情况下, 在服务器环境, 我们会使用 java 工具包中的 jstack 工具来查看: 如 jstack pid (应用 java 进程)。

但是, 在 IDEA 本地开发的话, IDEA 内置了一个工具, 可以直接查看当前应用的线程上下文信息, 如:



注意下面那个箭头指向的像照相机一样的图标, 故图思意, 就是打印当前线程快照的意思。

点击后, 就出现了右边那些线程上下文信息了, 可以看到有很多的线程, 我们主要关注下 main 线程, 线程状态确实是 waiting 的, 接着点

点击箭头所指向的main线程，可以看到如下内容：

```
"main@1" prio=5 tid=0x1 nid=NA waiting
java.lang.Thread.State: WAITING
at sun.misc.Unsafe.park(Unsafe.java:-1)
at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:836)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireSharedInterruptibly(AbstractQueuedSynchronizer.java:997)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(AbstractQueuedSynchronizer.java:1304)
at java.util.concurrent.CountDownLatch.await(CountDownLatch.java:231)
at org.springframework.boot.autoconfigure.BackgroundPreinitializer.onApplicationEvent(BackgroundPreinitializer.java:63)
at org.springframework.boot.autoconfigure.BackgroundPreinitializer.onApplicationEvent(BackgroundPreinitializer.java:45)
at org.springframework.context.event.SimpleApplicationEventMulticaster.doinvokeListener(SimpleApplicationEventMulticaster.java:172)
at org.springframework.context.event.SimpleApplicationEventMulticaster.invokeListener(SimpleApplicationEventMulticaster.java:158)
at org.springframework.context.event.SimpleApplicationEventMulticaster.multicastEvent(SimpleApplicationEventMulticaster.java:139)
at org.springframework.context.event.SimpleApplicationEventMulticaster.multicastEvent(SimpleApplicationEventMulticaster.java:127)
at org.springframework.boot.context.event.EventPublishingRunListener.finished(EventPublishingRunListener.java:115)
at org.springframework.boot.SpringApplicationRunListeners.callFinishedListener(SpringApplicationRunListeners.java:79)
at org.springframework.boot.SpringApplicationRunListeners.finished(SpringApplicationRunListeners.java:72)
at org.springframework.boot.SpringApplication.handleRunFailure(SpringApplication.java:745)
at org.springframework.boot.SpringApplication.run(SpringApplication.java:314)
at org.springframework.boot.SpringApplicationBuilder.run(SpringApplicationBuilder.java:134)
- locked <0xe6> (a java.util.concurrent.atomic.AtomicBoolean)
at org.springframework.cloud.bootstrap.BootstrapApplicationListener.bootstrapServiceContext(BootstrapApplicationListener.java:175)
at org.springframework.cloud.bootstrap.BootstrapApplicationListener.onApplicationEvent(BootstrapApplicationListener.java:98)
at org.springframework.cloud.bootstrap.BootstrapApplicationListener.onApplicationEvent(BootstrapApplicationListener.java:64)
at org.springframework.context.event.SimpleApplicationEventMulticaster.doinvokeListener(SimpleApplicationEventMulticaster.java:172)
at org.springframework.context.event.SimpleApplicationEventMulticaster.invokeListener(SimpleApplicationEventMulticaster.java:165)
at org.springframework.context.event.SimpleApplicationEventMulticaster.multicastEvent(SimpleApplicationEventMulticaster.java:139)
at org.springframework.context.event.SimpleApplicationEventMulticaster.multicastEvent(SimpleApplicationEventMulticaster.java:127)
at org.springframework.boot.context.event.EventPublishingRunListener.environmentPrepared(EventPublishingRunListener.java:74)
at org.springframework.boot.SpringApplicationRunListeners.environmentPrepared(SpringApplicationRunListeners.java:54)
at org.springframework.boot.SpringApplication.prepareEnvironment(SpringApplication.java:325)
at org.springframework.boot.SpringApplication.run(SpringApplication.java:296)
at cn.keking.project.customerManagement.KekingCustomerManagement.main(KekingCustomerManagement.java:36)
```

可以看到是通过CountDownLatch.await()阻塞了线程，接着看下面那行，所在代码块如下：

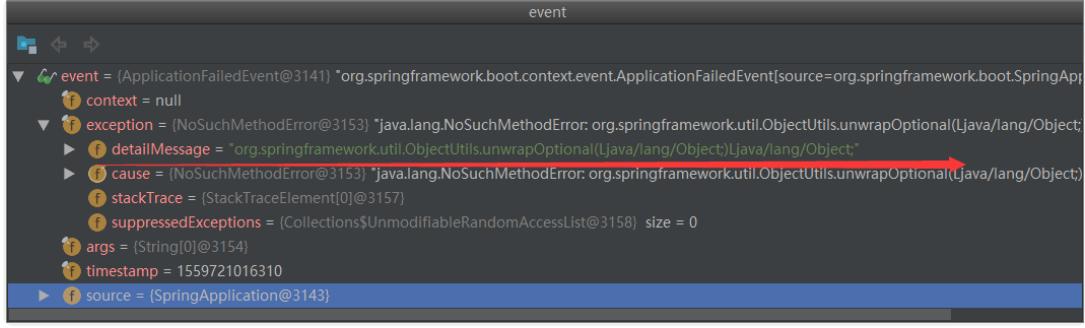
```
private static final CountDownLatch preinitializationComplete = new CountDownLatch(1);

@Override
public void onApplicationEvent(SpringApplicationEvent event) {
 if (event instanceof ApplicationEnvironmentPreparedEvent) {
 if (preinitializationStarted.compareAndSet(false, true)) {
 performPreinitialization();
 }
 }
 if (event instanceof ApplicationReadyEvent || event instanceof ApplicationFailedEvent) {
 try {
 preinitializationComplete.await();
 }
 catch (InterruptedException ex) {
 Thread.currentThread().interrupt();
 }
 }
}
```

这是spring boot中的一个安全初始化资源的一个类，代码所示为监听SpringApplicationEvent事件，可以肯定的是，它的逻辑执行到了preinitializationComplete.await();这里，导致了线程阻塞了。

正常情况下，spring会触发ApplicationEnvironmentPreparedEvent事件完成资源的初始化，这里先不深究Spring为什么这么做，主要

通过程序逻辑看下为什么卡在这里了，在preinitializationComplete.await();所在行打个断点，看下event对象里的信息，如下：



原来event是一个Spring上下文初始化失败的异常事件对象，对象里包含了具体的异常信息，如箭头所指，关键异常信息如：

*NoSuchMethodError: "org.springframework.util.ObjectUtils.unwrapOptional(Ljava/lang/Object;)Ljava/lang/Object;"*

## 假设问题

通过上面的分析，基本定位到Spring boot应用启动卡住这个表象背后的真实原因了，而且也定位到了异常信息。

出现NoSuchMethodError异常，是因为调用方法的时候，找不到方法了。一般出现在两个有关联的jar包，但是版本对不上了，也就是说的jar版本依赖冲突。查看了下，ObjectUtils是spring-core包里的一个类，当前的4.1.3版本确实没有这个unwrapOptional方法，spring-core-5.x的版本才新增了这个方法。因为之前的依赖是没有问题，为什么现在spring上下文会调用5.x的版本的方法呢？

所以先假设近期有开发在pom.xml里添加了新的的依赖，导致了这个问题。

## 小心求证

有了找问题的方向就好办了，因为代码都是git管理维护的，所以查看下pom.xml文件近期的提交记录即可，查看后，确实发现了近期对pom.xml有改动，添加了一个依赖

```
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-context</artifactId>
 <version>5.1.6.RELEASE</version>
</dependency>
```

这里还涉及到一点Maven依赖优先级的问题，在pom.xml里直接这样添加的依赖优先于其他jar中pom.xml依赖的，也就是说，即使spring boot1.5.7自带了Spring-context.4.1.3，但是这样指定后，应用最后依赖的还是5.1.6的版本。

具体的Maven依赖关系，可以参考我的博文《关于Maven的使用，这些你都了解了么？》。结合之前的分析，八九不离十了就是因为加了这个依赖导致的问题，spring-context.5.1.6配合spring-core.4.1.3肯定得出问题啊。直接移除这个依赖，然后启动系统一切正常，日志打印了Spring加载上线文的信息。

## 问题总结

定位这个问题的关键在于要了解java中线程堆栈的知识，在没有足够异常日志情况下通过线程快照排查问题。

在定位到问题后，如NoSuchMethodError这样的异常，需要平时的经验积累来假设问题的真实原因，然后在追本溯源证明问题所在根本原因。找问题本质一定要这种循序渐进的思路。

举例，出现这种问题，如果你直接去搜索引擎搜：“Spring boot应用启动卡住了”，是搜不出来什么东西的，但是当你发现了是由于jar冲突。去搜索引擎搜索：

“*NoSuchMethodError: "org.springframework.util.ObjectUtils.unwrapOptional(Ljava/lang/Object;)Ljava/lang/Object;"*”。就会有很多的内容，很容易解决问题。

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



### 推荐阅读

1. 你在公司项目里面看过哪些操蛋的代码？
2. 你知道 Spring Batch 吗？
3. 面试题：Lucene、Solr、ElasticSearch
4. 3 分钟带你彻底搞懂 Java 泛型背后的秘密
5. 团队开发中 Git 最佳实践



喜欢文章，点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 请给 Spring Boot 多一些内存

襄垣 Java后端 2019-09-09

Spring Boot总体来说，搭建还是比较容易的，特别是Spring Cloud全家桶，简称亲民微服务，但在发展趋势中，容器化技术已经成熟，面对巨耗内存的Spring Boot，小公司表示用不起。如今，很多刚诞生的JAVA微服务框架大多主打“轻量级”，主要还是因为Spring Boot太重。

## JAVA系微服务框架

### No1-Spring Cloud

#### 介绍

有Spring大靠山在，更新、稳定性、成熟度的问题根本不需要考虑。在JAVA系混的技术人员大约都听说过Spring的大名吧，所以不缺程序员……，而且这入手的难度十分低，完全可以省去一个架构师。

但是，你必然在服务器上付出：

- 至少一台“服务发现”的服务器；
- 可能有一个统一的网关Gateway；
- 可能需要一个用于“分布式配置管理”的配置中心；
- 可能进行“服务追踪”，知道我的请求从哪里来，到哪里去；
- 可能需要“集群监控”；
- 项目上线后发现，我们需要好多服务器，每次在集群中增加服务器时，都感觉心疼；

#### 压测30秒

##### 压测前的内存占用

```
Processes: 340 total, 2 running, 338 sleeping, 1715 threads 10:12:39
Load Avg: 2.87, 2.45, 1.99 CPU usage: 2.29% user, 1.81% sys, 95.88% idle
SharedLibs: 164M resident, 53M data, 34M linkedit.
MemRegions: 56640 total, 7188M resident, 179M private, 3249M shared.
PhysMem: 16G used (2653M wired), 228M unused.
VM: 1531G vsize, 1112M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 124631/37M in, 52794/14M out.
Disks: 164600/3838M read, 52539/937M written.

PID COMMAND %CPU TIME #TH #WQ #POR MEM PURG CMPR PGRP PPID
1526 java 0.1 00:10.58 43 1 124 304M 0B 0B 1526 1449
```

如图，内存占用304M。

##### 压测时的内存占用

```

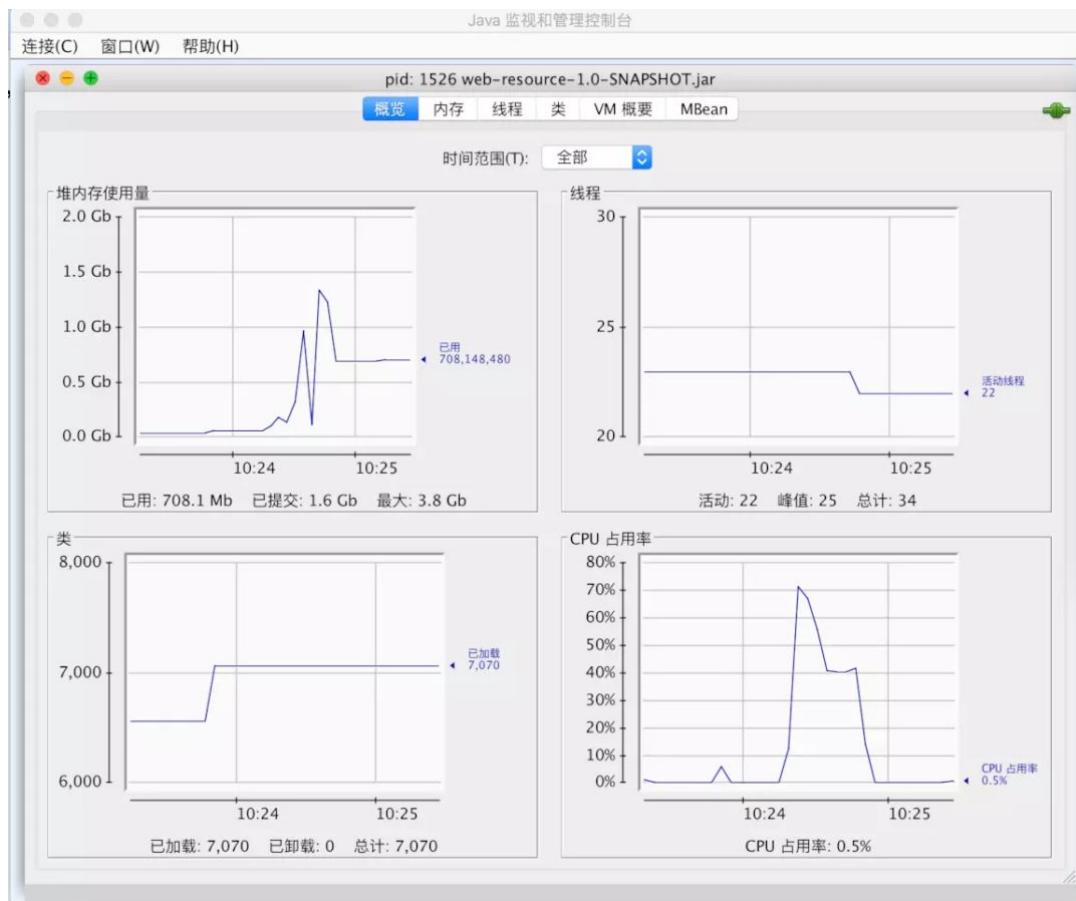
Processes: 347 total, 8 running, 339 sleeping, 1917 threads 10:29:42
Load Avg: 2.43, 2.61, 2.38 CPU usage: 45.93% user, 40.90% sys, 13.15% idle
SharedLibs: 165M resident, 53M data, 33M linkedit.
MemRegions: 60070 total, 8183M resident, 184M private, 2083M shared.
PhysMem: 16G used (2721M wired), 356M unused.
VM: 1568G vsize, 1112M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 16407645/1573M in, 16310036/1547M out.
Disks: 172958/3944M read, 61862/1207M written.

PID COMMAND %CPU TIME #TH #WQ #POR MEM PURG CMPR PGRP PPID
1526 java 321.0 02:50.33 41/4 1 120 1520M 0B 0B 1526 1449

```

如图，内存占用1520M（1.5G），CPU上升到321%

## 概览



## 总结

一个Spring Boot的简单应用，最少1G内存，一个业务点比较少的微服务编译后的JAR会大约50M；而Spring Cloud引入的组件会相对多一些，消耗的资源也会相对更多一些。

启动时间大约10秒左右: Started Application in 10.153 seconds (JVM running for 10.915)

## JAVA系响应式编程的工具包Vert.x

## 介绍

背靠Eclipse的Eclipse Vert.x是一个用于在JVM上构建响应式应用程序的工具包。定位上与Spring Boot不冲突，甚至可以将Vert.x结合Spring Boot使用。众多Vert.x模块提供了大量微服务的组件，在很多人眼里是一种微服务架构的选择。

华为微服务框架Apache ServiceComb就是以Vert.x为底层框架实现的，在"基准测试网站TechEmpower"中，Vert.x的表现也十分亮眼。

## 压测前的内存占用

```

Processes: 319 total, 2 running, 317 sleeping, 1642 threads 12:11:55
Load Avg: 2.02, 1.87, 1.89 CPU usage: 3.15% user, 2.42% sys, 94.41% idle
SharedLibs: 199M resident, 58M data, 45M linkedit.
MemRegions: 69030 total, 7190M resident, 207M private, 3100M shared.
PhysMem: 15G used (2599M wired), 768M unused.
VM: 1447G vsize, 1111M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 282452/109M in, 84946/21M out.
Disks: 157962/4480M read, 126426/2246M written.

PID COMMAND %CPU TIME #TH #WQ #POR MEM PURG CMPR PGRP PPID
2656 java 0.1 00:01.23 25 1 88 65M 0B 0B 2656 2337

```

如图，内存占用65M。

## 压测时的内存占用

```

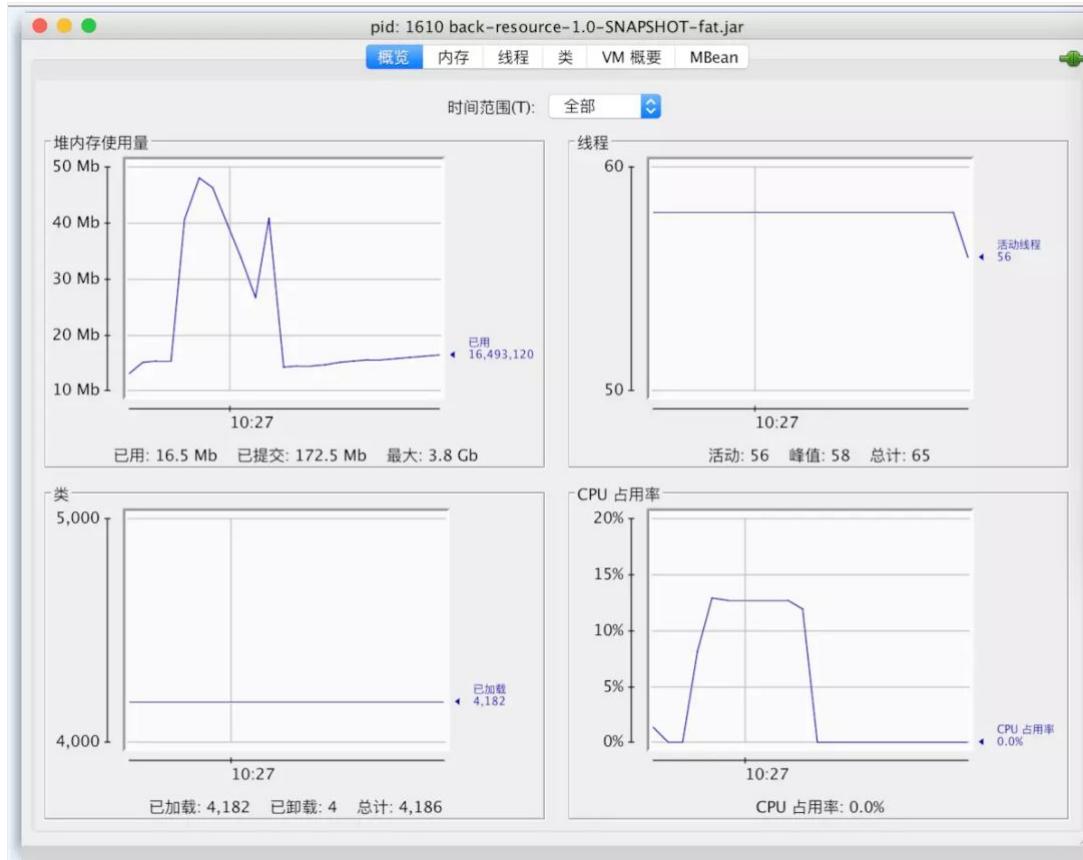
Processes: 342 total, 3 running, 339 sleeping, 1682 threads 10:14:51
Load Avg: 2.05, 2.30, 1.99 CPU usage: 3.74% user, 3.14% sys, 93.10% idle
SharedLibs: 164M resident, 53M data, 33M linkedit.
MemRegions: 57424 total, 7319M resident, 180M private, 3223M shared.
PhysMem: 16G used (2683M wired), 29M unused.
VM: 1546G vsize, 1111M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 131426/38M in, 56136/14M out.
Disks: 165715/3854M read, 54070/1024M written.

PID COMMAND %CPU TIME #TH #WQ #POR MEM PURG CMPR PGRP PPID
1610 java 2.1 00:02.84 46 1 130+ 139M+ 0B 0B 1610 1396

```

如图，内存占139M，CPU占2.1%，给人的感觉似乎并没有进行压测。

## 概览



## 总结

Vert.x单个服务打包完成后大约7M左右的JAR，不依赖Tomcat、Jetty之类的容器，直接在JVM上跑。

Vert.x消耗的资源很低，感觉一个1核2G的服务器已经能够部署许多个Vert.x服务。除去编码方面的问题，真心符合小项目和小模块。git市场上已经出现了基于Vert.x实现的开源网关- VX-API-Gateway帮助文档

<https://duhua.gitee.io/vx-api-gateway-doc>

对多语言支持，很适合小型项目快速上线。

启动时间不到1秒：Started Vert.x in 0.274 seconds (JVM running for 0.274)

## JAVA系其他微服务框架

### SparkJava

- jar比较小，大约10M；
- 占内存小，大约30~60MB；
- 性能还可以，与Spring Boot相仿；

### Micronaut

- Grails团队新宠；
- 可以用 Java、Groovy 和 Kotlin 编写的基于微服务的应用程序；
- 相比Spring Boot已经比较全面；
- 性能较优，编码方式与Spring Boot比较类似；
- 启动时间和内存消耗方面比其他框架更高效；
- 多语言；
- 依赖注入；
- 内置多种云本地功能；
- 很新，刚发布1.0.0

### Javalin

- 上手极为容易；
- 灵活，可以兼容同步和异步两种编程思路；
- JAR小，4~5M；
- 多语言；
- 有KOA的影子；
- 只有大约2000行源代码，源代码足够简单，可以理解和修复；
- 符合当今趋势；
- 多语言；
- 嵌入式服务器Jetty；

### Quarkus

- 启动快；
- JAR小，大约10M；
- 文档很少；

来源：[juejin.im/post/5c89f266f265da2d8763b5f9](http://juejin.im/post/5c89f266f265da2d8763b5f9)

作者：襄垣

如果喜欢本篇文章，欢迎[转发](#)、[点赞](#)。关注订阅号「Web项目聚集地」，回复「进群」即可进入无广告技术交流。

## 推荐阅读

1. 基于 Spring Boot 的 Restful 风格实现增删改查
2. 如何使用牛逼的插件帮你规范代码
3. IntelliJ IDEA 构建maven多模块工程项目
4. 别在 Java 代码里乱打日志了，这才是正确姿势
5. 挑战 10 道超难 Java 面试题
6. 什么时候进行分库分表？



Web项目聚集地

微信扫描二维码，关注我的公众号

喜欢文章，点个在看

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 附源码！Spring Boot 并发登录人數控制

Java后端 2019-11-03

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 殷天文

链接 | [www.jianshu.com/p/b6f5ec98d790](http://www.jianshu.com/p/b6f5ec98d790)

上篇 | 35 个小细节, 提升 Java 代码运行效率

通常系统都会限制同一个账号的登录人数，多人登录要么限制后者登录，要么踢出前者，Spring Security 提供了这样的功能，本文讲解一下在没有使用Security的时候如何手动实现这个功能

## demo 技术选型

- SpringBoot
- JWT
- Filter
- Redis + Redisson

JWT (token) 存储在Redis中，类似 JSessionId-Session的关系，用户登录后每次请求在Header中携带jwt

如果你是使用session的话，也完全可以借鉴本文的思路，只是代码上需要加些改动

## 两种实现思路

### 比较时间戳

维护一个 `username: jwtToken` 这样的一个 `key-value` 在Redis中, Filter逻辑如下



```
public class CompareKickOutFilter extends KickOutFilter {

 @Autowired
 private UserService userService;

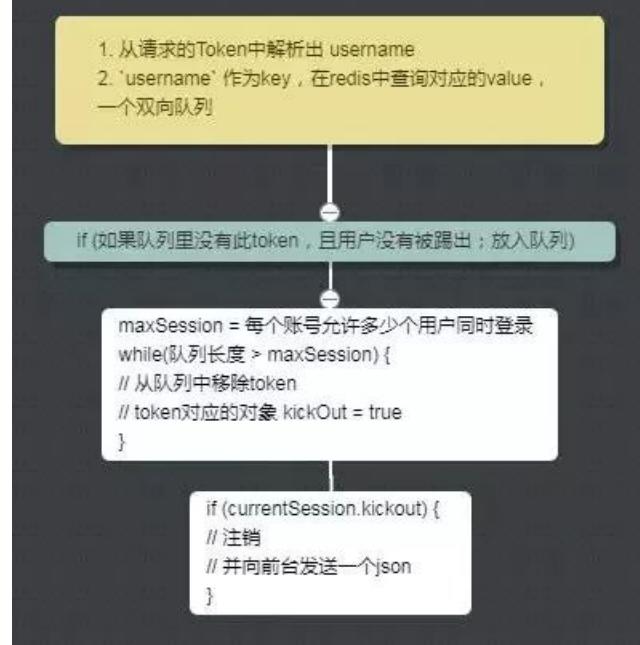
 @Override
 public boolean isAccessAllowed(HttpServletRequest request, HttpServletResponse response){
 String token = request.getHeader("Authorization");
 String username = JWTUtil.getUsername(token);
 String userKey = PREFIX + username;

 RBucket<String> bucket = redissonClient.getBucket(userKey);
 String redisToken = bucket.get();

 if(token.equals(redisToken)) {
 return true;
 } else if (StringUtils.isBlank(redisToken)) {
 bucket.set(token);
 } else {
 Long redisTokenUnixTime = JWTUtil.getClaim(redisToken, "createTime").asLong();
 Long tokenUnixTime = JWTUtil.getClaim(token, "createTime").asLong();

 //token > redisToken 则覆盖
 if(tokenUnixTime.compareTo(redisTokenUnixTime) > 0) {
 bucket.set(token);
 } else {
 //注销当前token
 userService.logout(token);
 sendJsonResponse(response, 4001, "您的账号已在其他设备登录");
 return false;
 }
 }
 return true;
 }
}
```

## 队列踢出



```

public class QueueKickOutFilter extends KickOutFilter {
 /**
 * 踢出之前登录的/之后登录的用户 默认踢出之前登录的用户
 */
 private boolean kickoutAfter = false;
 /**
 * 同一个帐号最大会话数 默认1
 */
 private int maxSession = 1;

 public void setKickoutAfter(boolean kickoutAfter) {
 this.kickoutAfter = kickoutAfter;
 }

 public void setMaxSession(int maxSession) {
 this.maxSession = maxSession;
 }

 @Override
 public boolean isAccessAllowed(HttpServletRequest request, HttpServletResponse response) throws Exception {
 String token = request.getHeader("Authorization");
 UserBO currentSession = CurrentUser.get();
 Assert.notNull(currentSession, "currentSession cannot null");
 String username = currentSession.getUsername();
 String userKey = PREFIX + "deque_" + username;
 String lockKey = PREFIX_LOCK + username;

 RLock lock = redissonClient.getLock(lockKey);

 lock.lock(2, TimeUnit.SECONDS);

 try {
 RDeque<String> deque = redissonClient.getDeque(userKey);

 // 如果队列里没有此token，且用户没有被踢出；放入队列
 if (!deque.contains(token) && currentSession.isKickout() == false) {
 deque.push(token);
 }

 // 如果队列里的sessionId数超出最大会话数，开始踢人
 while (deque.size() > maxSession) {
 String kickoutSessionId;
 if (kickoutAfter) { // 如果踢出后者
 kickoutSessionId = deque.removeFirst();
 }
 }
 } finally {
 lock.unlock();
 }
 }
}

```

```

} else { //否则踢出前者
 kickoutSessionId = deque.removeLast();
}

try {
 RBucket<UserBO> bucket = redissonClient.getBucket(kickoutSessionId);
 UserBO kickoutSession = bucket.get();

 if (kickoutSession != null) {
 // 设置会话的kickout属性表示踢出了
 kickoutSession.setKickout(true);
 bucket.set(kickoutSession);
 }
}

} catch (Exception e) {
}

}

//如果被踢出了，直接退出，重定向到踢出后的地址
if (currentSession.isKickout()) {
 //会话被踢出了
 try {
 //注销
 userService.logout(token);
 sendJsonResponse(response, 4001, "您的账号已在其他设备登录");
 }

 } catch (Exception e) {
 }

 return false;
}

}

} finally {
 if (lock.isHeldByCurrentThread()) {
 lock.unlock();
 LOGGER.info(Thread.currentThread().getName() + " unlock");
 }
}

return true;
}
}

```

## 比较两种方法

- 第一种方法逻辑简单粗暴，只维护一个key-value 不需要使用锁，非要说缺点的话没有第二种方法灵活。
- 第二种方法我很喜欢，代码很优雅灵活，但是逻辑相对麻烦一些，而且为了保证线程安全地操作队列，要使用分布式锁。目前我们项目中使用的是第一种方法

## 演示

下载地址:

<https://gitee.com/yintianwen7/taven-springboot-learning/tree/master/login-control>

1. 运行项目，访问localhost:8887 demo中没有存储用户信息，随意输入用户名密码，用户名相同则被踢出
2. 访问 localhost:8887/index.html 弹出用户信息，代表当前用户有效
3. 另一个浏览器登录相同用户名，回到第一个浏览器刷新页面，提示被踢出
4. application.properties中选择开启哪种过滤器模式，默认是比较时间戳踢出，开启队列踢出 queue-filter.enabled=true

本文借鉴了<https://jinnianshilongnian.iteye.com/blog/2039760>, 如果你是使用 Shiro + Session 的模式，推荐阅读此文。

-END-

如果看到这里，说明你喜欢这篇文章，请[转发](#)、[点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



## 推荐阅读

1. 盘点阿里巴巴 33 个牛逼的开源项目
2. 为什么我不建议你去外包公司？
3. 理解 IntelliJ IDEA 的项目配置和 Web 部署
4. Java 开发中常用的 4 种加密方法
5. 团队开发中 Git 最佳实践



学Java, 请关注公众号: Java后端

喜欢文章, 点个在看

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

# 零基础认识 Spring Boot

Lee宇斌 Java后端 2019-11-15

点击上方 Java后端, 选择 [设为星标](#)

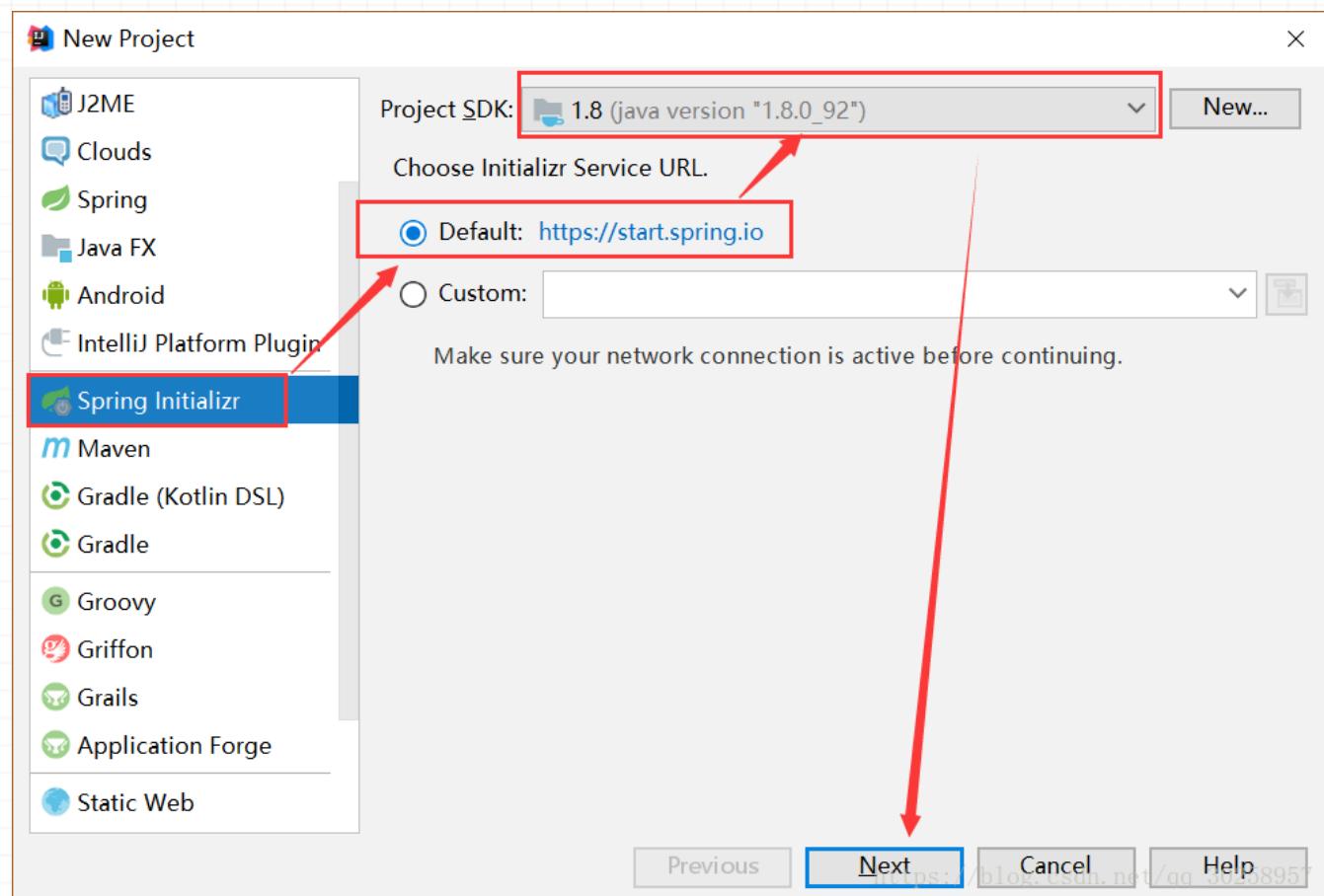
优质文章, 及时送达

作者 | Lee宇斌

来源 | [blog.csdn.net/qq\\_30258957](http://blog.csdn.net/qq_30258957)

## 新建项目

New Project – Spring Initializr – 选择web



确定文件路径

New Project

### Project Metadata

Group: com.leekoko

Artifact: demo

Type: Maven Project (Generate a Maven based project archive)

Language: Java

Packaging: Jar

Java Version: 8

Version: 0.0.1-SNAPSHOT

Name: girl

Description: Demo project for Spring Boot

Package: com.leekoko

Previous Next Cancel Help

选择版本，组件

springboot版本

Dependencies

Core

Web

Template Engines

SQL

NoSQL

Integration

Cloud Core

Cloud Support

Cloud Config

Cloud Discovery

Cloud Routing

Cloud Circuit Breaker

Cloud Tracing

Cloud Messaging

Cloud AWS

Cloud Contract

Pivotal Cloud Foundry

Azure

Spring Cloud GCP

I/O

Spring Boot 2.0.4

Web

Reactive Web

Rest Repositories

Rest Repositories HAL Browser

HATEOAS

Web Services

Jersey (JAX-RS)

Websocket

REST Docs

Vaadin

Apache CXF (JAX-RS)

Ratpack

Mobile

Keycloak

**Web**

Full-stack web development with Tomcat and Spring

[Building a RESTful Web Service](#)

[Serving Web Content with Spring MVC](#)

[Building REST services with Spring](#)

[Reference doc](#)

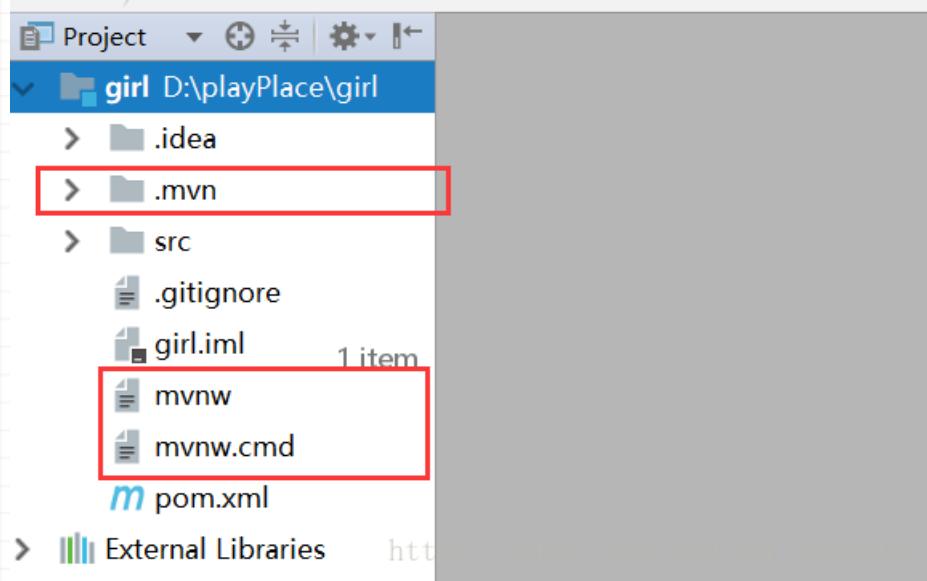
Selected Dep...

Web

Next

Previous Cancel Help

选择路径进行保存，删除没用的文件



### 启动SpringBoot项目

运行自动生成的XXApplication类，其必须带有@SpringBootApplication注解，右键Run XX即可启动项目。

```
1 @SpringBootApplication
2 public class HellospringbootApplication {
3 public static void main(String[] args) {
4 SpringApplication.run(HellospringbootApplication.class, args);
5 }
6 }
```

Idea在初次启动的时候需要加载许多东西，建议maven使用阿里云的仓库，加载完之后才会出现Run XX按钮。当出现此页面的时候，说明springBoot启动成功

## Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Aug 08 00:09:22 SGT 2018  
There was an unexpected error (type=Not Found, status=404).  
No message available

[https://blog.csdn.net/qq\\_30258957](https://blog.csdn.net/qq_30258957)

怎么编写一个Controller文件呢？添加类似Spring的注解，启动即可访问。(也可以先编译，通过命令启动)

```
@RestController
public class HelloController {
 @RequestMapping(value = "/hello", method = RequestMethod.GET)
 public String say(){
 return "Hello spring Boot!";
 }
}
```

由于代码补全快捷键冲突了，所以需要进行修改。

### 配置文件使用

新建的项目中，application.properties就是新建项目默认的配置文件。这里可以对访问端口和访问路径进行配置。

```
server.port=8081
server.context-path=/girl
```

相似的，application.yml也是默认配置文件，其使用分组的格式，:之后必须加**空格**，子内容前面为**tab键**

```
server:
 port: 8081
 context-path: /girl
```

yml可以配置java代码中注入的值，直接写键:值，用@Value("\${键}")的方式即可注入。直接用\${}就可以在xml中进行引用。

```
server:
 port: 8081
age: 18
size: B
content: "size: ${size}, age: ${age}"
```

当配置文件需要频繁变换，怎么灵活切换呢？可以将其写成两个配置文件，而主配置文件只要选好要哪一个配置文件即可。新建两个配置文件 application-dev.yml & application-prod.yml，在application.yml中指定调用哪一个配置文件：

```
1 spring:
2 profiles:
3 active:dev
```

调用dev后缀的配置文件。

## 注解的使用

### @Component & @ConfigurationProperties

一个个属性注入太麻烦了，有没有注入对象的方法呢？修改配置文件为组的形式，编写pojo对象映射，再将pojo对象注入

```
1 server:
2 port: 8081
3 girl:
4 age: 18
5 size: B
```

pojo对象，需要@Component定义Spring管理Bean，@ConfigurationProperties指定前缀内容。

@Component注解相当于:@Service,@Controller,@Repository，并下面类纳入进spring容器中管理。这样才能被下一层@Autowired注入该对象。

```

@Component
@ConfigurationProperties(prefix = "girl")
public class GirlProperties {

 private String size;

 private Integer age;

 public String getSize() {
 return size;
 }

 public void setSize(String size) {
 this.size = size;
 }

 ...
}

```

运行@SpringBootApplication,即可访问Controller的内容。

```

@RestController
public class HelloController {

 @Autowired
 private GirlProperties girlProperties;

 @RequestMapping(value = "/hello",method = RequestMethod.GET)
 public String say(){
 return girlProperties.getSize();
 }
}

```

**@RestController**

**@RestController** = **@ResponseBody** + **@Controller**

**@RequestMapping**

**@RequestMapping**可以指定多个value:@RequestMapping(value={"/say","/hi"})。

**@RequestMapping**的Get请求获取参数的方式：

方式一：**PathVariable**:访问地址中间参数传输：

```

@RequestMapping(value="/{id}/say",method = RequestMethod.GET)
public String say(@PathVariable("id") Integer id){
 return "Hello Spring Boot:"+id;
}

```

url访问地址可以将id中间位置：<http://localhost:8080/hello/233333/say>

方式二：**RequestParam**:访问地址后面传值：

```

@RequestMapping(value="/say",method = RequestMethod.GET)
public String say(@RequestParam("id") Integer id){
 return "Hello Spring Boot:"+id;
}

```

url访问方式: <http://localhost:8080/hello/say?id=110>

添加默认值: (@RequestParam(value = "id", required = false, defaultValue = "0") Integer id ,如何不传id,它就会默认为0。)

@RequestMapping(value="/say",method = RequestMethod.GET) 也可以写成 @GetMapping(value="/say") 的方式。

### @Transactional

当我在一个Service的方法里有两条sql插入操作,怎么保证其同时执行成功或者同时执行失败?在方法上面添加@Transactional注解,即说明其为同个事务。

```
@Transactional
public void insertTwo(){
 Girl girlA = new Girl();
 girlA.setSize("A");
 girlA.setAge(10);
 girlRepository.save(girlA);

 Girl girlB = new Girl();
 girlB.setSize("BBBBB");
 girlB.setAge(20);
 girlRepository.save(girlB);
}
```

只有在innodb引擎下事务才能工作。所以需要在数据库中执行ALTER TABLE girl ENGINE=innodb命令。

## 数据库操作

### 创建表

要操作数据库,首先添加组件

pom.xml添加组件

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

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

application.yml配置数据库连接:

```
spring:
 datasource:
 driver-class-name: com.mysql.jdbc.Driver
 url: jdbc:mysql://127.0.0.1:3306/test
 username: root
 password: 123456
 jpa:
 hibernate:
 ddl-auto: create
 show-sql: true
```

**ddl-auto: create**:每次都重新创建数据库，数据不保存，要保存得用 **update**。

**show-sql: true**:打印sql语句。

并且需要在mysql中创建对应的数据库。配置完上方的jpa之后，编写pojo对象，添加@Entity注解，标注id@Id，自增长@GeneratedValue，运行之后数据库就会自动生成对应表。**ddl-auto: create**配置将决定表是创建create还是更新update

```
@Entity
public class Girl {

 @Id
 @GeneratedValue
 private Integer id;

 private String size;

 private Integer age;

 public Integer getId() {
 return id;
 }
 ...
}
```

## JPA实现增删改查

新建接口，继承JpaRepository<Girl, Integer>，注入接口，直接调用 JpaRepository中的CRUD方法即可实现查询所有。

### 新建接口

```
1 public interface GirlRepository extends JpaRepository<Girl, Integer> {
2
3 }
```

### 调用CRUD方法

```
/**
 * 查询所有
 * @return
 */

@GetMapping(value = "/girls")

public List<Girl> girlList()
```

```
public List<Girl> girlList() {
 return girlRepository.findAll();
}

}

/***
 * 根据id查询
 * @param id
 * @return
 */
@GetMapping(value = "/girlById/{id}")
public Girl girlFindOne(@PathVariable("id") Integer id) {
 Optional<Girl> temp = girlRepository.findById(id);
 //从返回值中获取值
 return temp.get();
}

}

/***
 * 添加内容
 * @param age
 */
@PostMapping(value = "/girlAdd")
public Girl girlAdd(@RequestParam("size") String size, @RequestParam("age") Integer age) {
 Girl girl = new Girl();
 girl.setAge(age);
 girl.setSize(size);
 return girlRepository.save(girl);
}

}

/***
 * 更新
 */
@PutMapping(value = "/moGirlById/{id}")
public Girl girlUpdate(@PathVariable("id") Integer id, @RequestParam("age") Integer age, @RequestParam("size")
```

```
public Girl girlUpdate(@PathVariable("id") Integer id, @RequestParam("age") Integer age, @RequestParam("size") Integer size) {
 Girl girl = new Girl();

 girl.setId(id);

 girl.setAge(age);

 girl.setSize(size);

 return girlRepository.save(girl);
}

/**
 * 删除
 */

@DeleteMapping(value = "/delGirls/{id}")

public void girlDelete(@PathVariable("id") Integer id) {

 Girl girl = new Girl();

 girl.setId(id);

 girlRepository.delete(girl);
}
```

如果某些方法在JpaRepository中不存在，可以自己使用扩展方法，写在接口中，调用即可。但是要求方法名要规范。

```
1 public interface GirlRepository extends JpaRepository<Girl, Integer> {
2 //通过年龄查询
3 public List<Girl> findByAge(Integer age);
4 }

/**
 * 通过年龄查询
 */
@GetMapping(value = "/girlByAge/{age}")
public List<Girl> getListByAge(@PathVariable("age") Integer age) {
 return girlRepository.findByAge(age);
}
```

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

面试官：说一说 Spring Boot 自动配置原理吧，我懵逼了...

你在我家门口 Java后端 2019-09-22



作者 | 你在我家门口

juejin.im/post/5ce5effb6fb9a07f0b039a14

**推荐：**公众号高质量博文整理（请戳我）

小伙伴们是否想起曾经被 SSM 整合支配的恐惧？相信很多小伙伴都是有过这样的经历的，一大堆配置问题，各种排除扫描，导入一个新的依赖又得添加新的配置。自从有了 Spring Boot 之后，咱们就起飞了！各种零配置开箱即用，而我们之所以开发起来能够这么爽，自动配置的功劳少不了，今天我们就一起来讨论一下 Spring Boot 自动配置原理，看完心里有个大概，不至于被面试官问的面红耳赤。

本文主要分为三大部分：

- SpringBoot 源码常用注解拾遗
- SpringBoot 启动过程
- SpringBoot 自动配置原理

## 1. SpringBoot 源码常用注解拾遗

这部分主要讲一下 SpringBoot 源码中经常使用到的注解，以扫清后面阅读源码时候的障碍。

### 组合注解

当可能大量同时使用到几个注解到同一个类上，就可以考虑将这几个注解到别的注解上。被注解的注解我们就称之为组合注解。

- 元注解：可以注解到别的注解上的注解。
- 组合注解：被注解的注解我们就称之为组合注解。

### @Value

@Value 就相当于传统 xml 配置文件中的 value 字段。

假设存在代码：

```
1 @Component
2 public class Person {
```

```
3
4 @Value("i am name")
5
6 private String name;
7
}
```

上面代码等价于的配置文件：

```
1 <bean class="Person">
2 <property name ="name" value="i am name"></property
3 >
4 </bean>
```

我们知道配置文件中的 value 的取值可以是：

- 字面量
- 通过 \${key} 方式从环境变量中获取值
- 通过 \${key} 方式全局配置文件中获取值
- #\${SpEL}

所以，我们就可以通过 `@Value(${key})` 的方式获取全局配置文件中的指定配置项。微信搜索 `web_resource` 回复 爆文 送你高质量技术博文。

## @ConfigurationProperties

如果我们需要取 N 个配置项，通过 `@Value` 的方式去配置项需要一个一个去取，这就显得有点 low 了。我们可以使用 `@ConfigurationProperties`。

标有 `@ConfigurationProperties` 的类的所有属性和配置文件中相关的配置项进行绑定。（默认从全局配置文件中获取配置值），绑定之后我们就可以通过这个类去访问全局配置文件中的属性值了。

下面看一个实例：

### 1.在主配置文件中添加如下配置

```
1 person.name=kundy
2 person.age=13
3 person.sex=maile
```

2.创建配置类，由于篇幅问题这里省略了 setter、getter 方法，但是实际开发中这个是必须的，否则无法成功注入。另外，`@Component` 这个注解也还是需要添加的。

```
1 @Component
2 @ConfigurationProperties(prefix = "person")
3
4 public class Person {
5
```

```
6 private String name;
7 private Integer age;
8 private String sex;
9 }
```

这里 `@ConfigurationProperties` 有一个 `prefix` 参数，主要是用来指定该配置项在配置文件中的前缀。

3. 测试，在 `SpringBoot` 环境中，编写个测试方法，注入 `Person` 类，即可通过 `Person` 对象取到配置文件的值。

## @Import

`@Import` 注解支持导入普通 `java` 类，并将其声明成一个 `bean`。主要用于将多个分散的 `java config` 配置类融合成一个更大的 `config` 类。

- `@Import` 注解在 4.2 之前只支持导入配置类。
- 在 4.2 之后 `@Import` 注解支持导入普通的 `java` 类，并将其声明成一个 `bean`。

## @Import 三种使用方式

- 直接导入普通的 `Java` 类。
- 配合自定义的 `ImportSelector` 使用。
- 配合 `ImportBeanDefinitionRegistrar` 使用。

### 1. 直接导入普通的 Java 类

1. 创建一个普通的 `Java` 类。

```
1 public class Circle {
2
3 public void sayHi() {
4 System.out.println("Circle sayHi()");
5 }
6
7 }
```

2. 创建一个配置类，里面没有显式声明任何的 `Bean`，然后将刚才创建的 `Circle` 导入。[微信搜索 web\\_resource 回复 爆文 送你高质量技术博文。](#)

```
1 @Import({Circle.class})
2 @Configuration
3 public class MainConfig {
4
5 }
```

3. 创建测试类。

```
1 public static void main(String[] args) {
```

```
2
3 ApplicationContext context = new AnnotationConfigApplicationContext(MainConfig.class);
4 Circle circle = context.getBean(Circle.class);
5 circle.sayHi();
6
7 }
```

#### 4. 运行结果：

```
Circle sayHi()
```

可以看到我们顺利的从 IOC 容器中获取到了 Circle 对象，证明我们在配置类中导入的 Circle 类，确实被声明为了一个 Bean。

## 2. 配合自定义的 ImportSelector 使用

ImportSelector 是一个接口，该接口中只有一个 selectImports 方法，用于返回全类名数组。所以利用该特性我们可以给容器动态导入 N 个 Bean。

### 1. 创建普通 Java 类 Triangle。

```
1 public class Triangle {
2
3 public void sayHi(){
4 System.out.println("Triangle sayHi()");
5 }
6
7 }
```

### 2. 创建 ImportSelector 实现类，selectImports 返回 Triangle 的全类名。

```
1 public class MyImportSelector implements ImportSelector {
2
3 @Override
4 public String[] selectImports(AnnotationMetadata annotationMetadata) {
5 return new String[]{"annotation.importannotation.waytwo.Triangle"};
6 }
7 }
8 }
```

### 3. 创建配置类，在原来的基础上还导入了 MyImportSelector。

```
1 @Import({Circle.class,MyImportSelector.class})
2 @Configuration
3 public class MainConfigTwo {
4
5 }
```

#### 4. 创建测试类

```
1 public static void main(String[] args) {
2
3 ApplicationContext context = new AnnotationConfigApplicationContext(MainConfigTwo.class);
4 Circle circle = context.getBean(Circle.class);
5 Triangle triangle = context.getBean(Triangle.class);
6 circle.sayHi();
7 triangle.sayHi();
8
9 }
```

#### 5. 运行结果：

```
Circle sayHi()
Triangle sayHi()
```

可以看到 Triangle 对象也被 IOC 容器成功的实例化出来了。

### 3. 配合 ImportBeanDefinitionRegistrar 使用

ImportBeanDefinitionRegistrar 也是一个接口，它可以手动注册bean到容器中，从而我们可以对类进行个性化的定制。(需要搭配 @Import 与 @Configuration 一起使用。) [微信搜索 web\\_resource 回复 爆文 送你高质量技术博文。](#)

#### 1. 创建普通 Java 类 Rectangle。

```
1 public class Rectangle {
2
3 public void sayHi() {
4 System.out.println("Rectangle sayHi()");
5 }
6
7 }
```

#### 2. 创建 ImportBeanDefinitionRegistrar 实现类，实现方法直接手动注册一个名叫 rectangle 的 Bean 到 IOC 容器中。

```
1 public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
2
3 @Override
4 public void registerBeanDefinitions(AnnotationMetadata annotationMetadata, BeanDefinitionRegistry beanDefinitionRe
5
6 RootBeanDefinition rootBeanDefinition = new RootBeanDefinition(Rectangle.class);
7 // 注册一个名字叫做 rectangle 的 bean
8 beanDefinitionRegistry.registerBeanDefinition("rectangle", rootBeanDefinition);
9
10 }
11 }
```

### 3. 创建配置类，导入 MyImportBeanDefinitionRegistrar 类。

```
1 @Import({Circle.class, MyImportSelector.class, MyImportBeanDefinitionRegistrar.class})
2 @Configuration
3 public class MainConfigThree {
4
5 }
```

### 4. 创建测试类。

```
1 public static void main(String[] args) {
2
3 ApplicationContext context = new AnnotationConfigApplicationContext(MainConfigThree.class);
4
5 Circle circle = context.getBean(Circle.class);
6
7 Triangle triangle = context.getBean(Triangle.class);
8 Rectangle rectangle = context.getBean(Rectangle.class);
9 circle.sayHi();
10 triangle.sayHi();
11 rectangle.sayHi();
}
}
```

### 5. 运行结果

```
Circle sayHi()
Triangle sayHi()
Rectangle sayHi()
```

嗯对， Rectangle 对象也被注册进来了。

### @Conditional

@Conditional 注释可以实现只有在特定条件满足时才启用一些配置。

下面看一个简单的例子：

#### 1. 创建普通 Java 类 ConditionBean，该类主要用来验证 Bean 是否成功加载。

```
1 public class ConditionBean {
2
3 public void sayHi() {
4 System.out.println("ConditionBean sayHi()");
5 }
6
7 }
```

2. 创建 Condition 实现类，@Conditional 注解只有一个 Condition 类型的参数，Condition 是一个接口，该接口只有一个返回布尔值的 matches() 方法，该方法返回 true 则条件成立，配置类生效。反之，则不生效。在该例子中我们直接返回 true。

```
1 public class MyCondition implements Condition {
2
3 @Override
4 public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata annotatedTypeMetadata) {
5 return true;
6 }
7
8 }
```

3. 创建配置类，可以看到该配置的 @Conditional 传了我们刚才创建的 Condition 实现类进去，用作条件判断。[微信搜索 web\\_resource 回复 爆文 送你高质量技术博文。](#)

```
1 @Configuration
2 @Conditional(MyCondition.class)
3 public class ConditionConfig {
4
5 @Bean
6 public ConditionBean conditionBean(){
7
8 return new ConditionBean();
9 }
10 }
```

4. 编写测试方法。

```
1 public static void main(String[] args) {
2
3 ApplicationContext context = new AnnotationConfigApplicationContext(ConditionConfig.class);
4 ConditionBean conditionBean = context.getBean(ConditionBean.class);
5 conditionBean.sayHi();
6
7 }
```

5. 结果分析

因为 Condition 的 matches 方法直接返回了 true，配置类会生效，我们可以把 matches 改成返回 false，则配置类就不会生效了。

除了自定义 Condition，Spring 还为我们扩展了一些常用的 Condition。

扩展注解	作用
ConditionalOnBean	容器中存在指定 Bean，则生效。
ConditionalOnMissingBean	容器中不存在指定 Bean，则生效。
ConditionalOnClass	系统中有指定的类，则生效。
ConditionalOnMissingClass	系统中没有指定的类，则生效。
ConditionalOnProperty	系统中指定的属性是否有指定的值。
ConditionalOnWebApplication	当前是web环境，则生效。

## 2. SpringBoot 启动过程

在看源码的过程中，我们会看到以下四个类的方法经常会被调用，我们需要对一下几个类有点印象：

- ApplicationContextInitializer
- ApplicationRunner
- CommandLineRunner
- SpringApplicationRunListener

下面开始源码分析，先从 SpringBoot 的启动类的 run() 方法开始看，以下是调用链：`SpringApplication.run() -> run(new Class[]{primarySource}, args) -> new SpringApplication(primarySources).run(args)`。

一直在run，终于到重点了，我们直接看 `new SpringApplication(primarySources).run(args)` 这个方法。

上面的方法主要包括两大步骤：

- 创建 SpringApplication 对象。
- 运行 run() 方法。

### 创建 SpringApplication 对象

```

1 public SpringApplication(ResourceLoader resourceLoader, Class... primarySources) {
2
3 this.sources = new LinkedHashSet();
4 this.bannerMode = Mode.CONSOLE;
5 this.logStartupInfo = true;
6 this.addCommandLineProperties = true;
7 this.addConversionService = true;
8
9 this.headless = true;
10 this.registerShutdownHook = true;
11
12 this.additionalProfiles = new HashSet();
13 this.isCustomEnvironment = false;
14 this.resourceLoader = resourceLoader;

```

```

14 Assert.notNull(primarySources, "PrimarySources must not be null");
15 // 保存主配置类（这里是一个数组，说明可以有多个主配置类）
16 this.primarySources = new LinkedHashSet(Arrays.asList(primarySources));
17 // 判断当前是否是一个 Web 应用
18 this.webApplicationType = WebApplicationType.deduceFromClasspath();
19 // 从类路径下找到 META/INF/Spring.factories 配置的所有 ApplicationContextInitializer，然后保存起来
20 this.setInitializers(this.getSpringFactoriesInstances(ApplicationContextInitializer.class));
21 // 从类路径下找到 META/INF/Spring.factories 配置的所有 ApplicationListener，然后保存起来
22 this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.class));
23 // 从多个配置类中找到有 main 方法的主配置类（只有一个）
24 this.mainApplicationClass = this.deduceMainApplicationClass();
25
}

```

## 运行 run() 方法

```

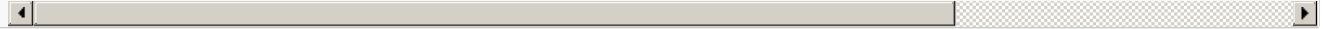
1 public ConfigurableApplicationContext run(String... args) {
2
3 // 创建计时器
4 Stopwatch stopwatch = new Stopwatch();
5 stopwatch.start();
6 // 声明 IOC 容器
7 ConfigurableApplicationContext context = null;
8
9 Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList();
10 this.configureHeadlessProperty();
11 // 从类路径下找到 META/INF/Spring.factories 获得 SpringApplicationRunListeners
12 SpringApplicationRunListeners listeners = this.getRunListeners(args);
13 // 回调所有 SpringApplicationRunListeners 的 starting() 方法
14 listeners.starting();
15 Collection exceptionReporters;
16 try {
17 // 封装命令行参数
18 ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
19 // 准备环境，包括创建环境，创建环境完成后回调 SpringApplicationRunListeners#environmentPrepared()方法，表示环境准备完成
20 ConfigurableEnvironment environment = this.prepareEnvironment(listeners, applicationArguments);
21 this.configureIgnoreBeanInfo(environment);
22 // 打印 Banner
23 Banner printedBanner = this.printBanner(environment);
24 // 创建 IOC 容器（决定创建 web 的 IOC 容器还是普通的 IOC 容器）
25 context = this.createApplicationContext();
26 exceptionReporters = this.getSpringFactoriesInstances(SpringBootExceptionReporter.class, new Class[]{ConfigurableA
27 /*
28 * 准备上下文环境，将 environment 保存到 IOC 容器中，并且调用 applyInitializers() 方法
29 * applyInitializers() 方法回调之前保存的所有的 ApplicationContextInitializer 的 initialize() 方法
30 * 然后回调所有的 SpringApplicationRunListener#contextPrepared() 方法
31 * 最后回调所有的 SpringApplicationRunListener#contextLoaded() 方法
32 */
33 this.prepareContext(context, environment, listeners, applicationArguments, printedBanner);
34 // 刷新容器，IOC 容器初始化（如果是 Web 应用还会创建嵌入式的 Tomcat），扫描、创建、加载所有组件的地方
35 this.refreshContext(context);
36 // 从 IOC 容器中获取所有的 ApplicationRunner 和 CommandLineRunner 进行回调
37 }
38
39 return context;
40 }

```

```

36 // 从 IOC 容器中获取所有的 ApplicationRunner 和 CommandLineRunner 进行调用
37 this.afterRefresh(context, applicationArguments);
38 stopWatch.stop();
39 if (this.logStartupInfo) {
40 (new StartupInfoLogger(this.mainApplicationClass)).logStarted(this.getApplicationLog(), stopWatch);
41 }
42 // 调用 所有 SpringApplicationRunListeners#started()方法
43 listeners.started(context);
44 this.callRunners(context, applicationArguments);
45 } catch (Throwable var10) {
46 this.handleRunFailure(context, var10, exceptionReporters, listeners);
47 throw new IllegalStateException(var10);
48 }
49 try {
50 listeners.running(context);
51 return context;
52 } catch (Throwable var9) {
53 this.handleRunFailure(context, var9, exceptionReporters, (SpringApplicationRunListeners)null);
54 throw new IllegalStateException(var9);
55 }

```



## 小结：

run() 阶段主要就是回调本节开头提到过的4个监听器中的方法与加载项目中组件到 IOC 容器中，而所有需要回调的监听器都是从类路径下的 META-INF/Spring.factories 中获取，从而达到启动前后的各种定制操作。微信搜索 *web\_resource* 回复 爆文 送你高质量技术博文。

## 3. SpringBoot 自动配置原理

### @SpringBootApplication 注解

SpringBoot 项目的一切都要从 @SpringBootApplication 这个注解开始说起。

@SpringBootApplication 标注在某个类上说明：

- 这个类是 SpringBoot 的主配置类。
- SpringBoot 就应该运行这个类的 main 方法来启动 SpringBoot 应用。

该注解的定义如下：

```

1 @SpringBootConfiguration
2 @EnableAutoConfiguration
3 @ComponentScan(
4 excludeFilters = {@Filter(
5 type = FilterType.CUSTOM,
6 classes = {TypeExcludeFilter.class}
7), @Filter(
8
9)
10)

```

```

11 type = FilterType.CUSTOM,
12 classes = {AutoConfigurationExcludeFilter.class}
13
14 })
)
public @interface SpringApplication {
}

}

```

可以看到 `SpringBootApplication` 注解是一个组合注解（关于组合注解文章的开头有讲到），其主要组合了一下三个注解：

- `@SpringBootConfiguration`: 该注解表示这是一个 SpringBoot 的配置类，其实它就是一个 `@Configuration` 注解而已。
- `@ComponentScan`: 开启组件扫描。
- `@EnableAutoConfiguration`: 从名字就可以看出来，就是这个类开启自动配置的。嗯，自动配置的奥秘全都在这个注解里面。

## `@EnableAutoConfiguration` 注解

先看该注解是怎么定义的：

```

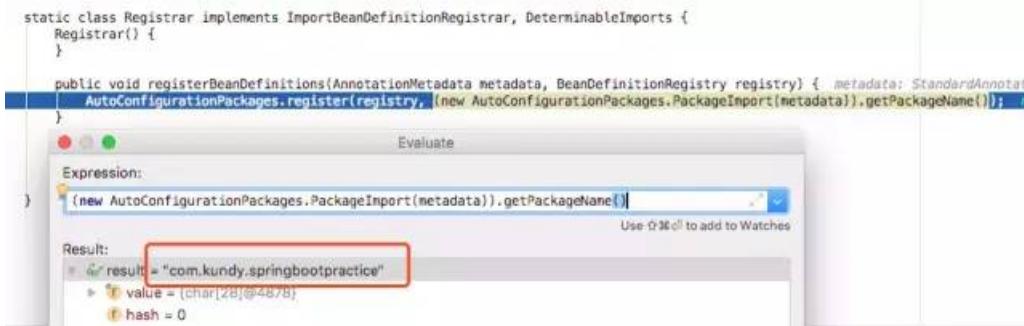
1 @AutoConfigurationPackage
2 @Import({AutoConfigurationImportSelector.class})
3 public @interface EnableAutoConfiguration {
}

```

### `@AutoConfigurationPackage`

从字面意思理解就是自动配置包。点进去可以看到就是一个 `@Import` 注解： `@Import({Registrar.class})`，导入了一个 `Registrar` 的组件。关于 `@Import` 的用法文章上面也有介绍哦。

我们在 `Registrar` 类中的 `registerBeanDefinitions` 方法上打上断点，可以看到返回了一个包名，该包名其实就是主配置类所在的包。[微信搜索 web\\_resource 回复 爆文 送你高质量技术博文](#)。



一句话：`@AutoConfigurationPackage` 注解就是将主配置类（`@SpringBootConfiguration` 标注的类）的所在包及下面所有子包里面的所有组件扫描到 Spring 容器中。所以说，默认情况下主配置类包及子包以外的组件，Spring 容器是扫描不到的。

### `@Import({AutoConfigurationImportSelector.class})`

该注解给当前配置类导入另外的 N 个自动配置类。（该注解详细用法上文有提及）。

## 配置类导入规则

那具体的导入规则是什么呢？我们来看一下源码。在开始看源码之前，先啰嗦两句。就像小马哥说的，我们看源码不用全部都看，不用每一行代码都弄明白是什么意思，我们只要抓住关键的地方就可以了。

我们知道 AutoConfigurationImportSelector 的 selectImports 就是用来返回需要导入的组件的全类名数组的，那么如何得到这些数组呢？

在 selectImports 方法中调用了一个 getAutoConfigurationEntry() 方法。

```
public class AutoConfigurationImportSelector implements RequiredImportSelector, BeanClassLoaderAware, ResourceLoaderAware, BeanFactoryAware, EnvironmentAware, Ordered {
 private static final AutoConfigurationImportSelector autoConfigurationImportSelector = new AutoConfigurationImportSelector();
 private static final String[] NO_IMPORTS = new String[0];
 private static final String PROPERTY_NAME_AUTOCONFIGURE_EXCLUDE = "spring.autoconfigure.exclude";
 private ConfigurableListableBeanFactory beanFactory;
 private Environment environment;
 private ClassLoader beanClassLoader;
 private ResourceLoader resourceLoader;
 public AutoConfigurationImportSelector() {
 }

 public String[] selectImports(AnnotationMetadata annotationMetadata) {
 if (!this.isEnabled(annotationMetadata)) {
 return NO_IMPORTS;
 } else {
 AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
 AutoConfigurationImportSelector autoConfigurationImportSelector = this.getAutoConfigurationEntry(autoConfigurationMetadata, annotationMetadata);
 return StringUtils.toStringArray(autoConfigurationImportSelector.getConfigurations());
 }
 }
}
```

由于篇幅问题我就不一一截图了，我直接告诉你们调用链：在 `getAutoConfigurationEntry() -> getCandidateConfigurations() -> loadFactoryNames()` 。

在这里 `loadFactoryNames()` 方法传入了 `EnableAutoConfiguration.class` 这个参数。先记住这个参数，等下会用到。

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
 List<String> configurations = SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClass(), this.getBeanClassLoader());
 Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.factories. If you are using a custom packaging, make sure that file is correct.");
 return configurations;
}

protected Class<?> getSpringFactoriesLoaderFactoryClass() {
 return EnableAutoConfiguration.class;
}
```

`loadFactoryNames()` 中关键的三步：

- 从当前项目的类路径中获取所有 `META-INF/spring.factories` 这个文件下的信息。
- 将上面获取到的信息封装成一个 Map 返回。
- 从返回的 Map 中通过刚才传入的 `EnableAutoConfiguration.class` 参数，获取该 key 下的所有值。

```
public static List<String> loadFactoryNames(Class<?> factoryClass, MutableClassLoader classLoader) {
 String factoryClassName = factoryClass.getName();
 return (List<String>) SpringFactoriesLoader.loadDefault(factoryClassName, Collections.emptyList());
}

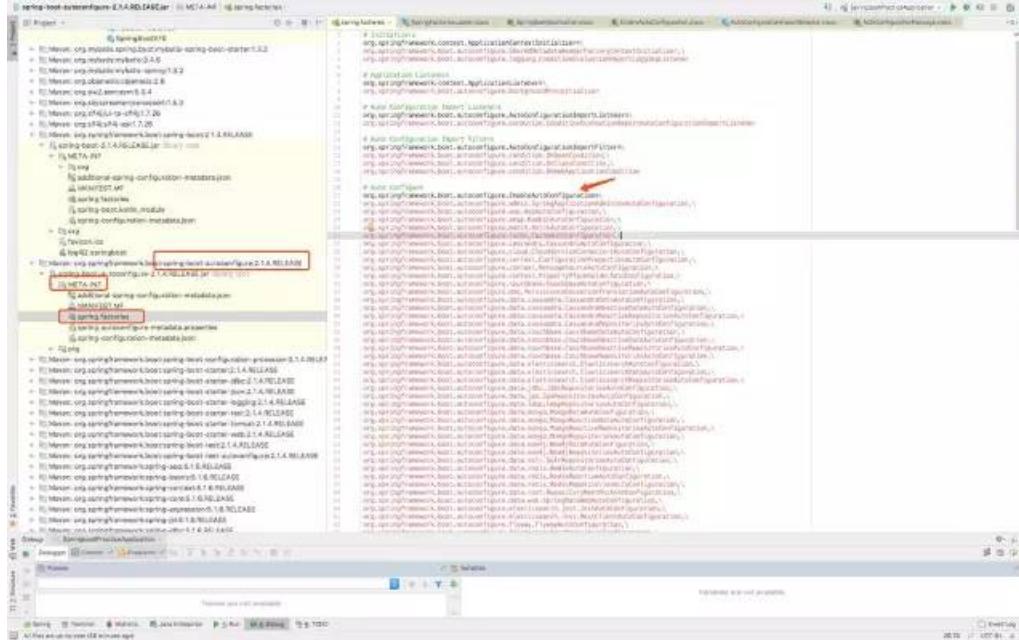
private static Map<String, List<String>> loadSpringFactories(MutableClassLoader classLoader) {
 Multimap<String, String> result = MultiValueMapCache.get(classLoader);
 if (result != null) {
 return result;
 } else {
 try {
 Enumeration<URL> urls = classLoader != null ? classLoader.getResources("META-INF/spring.factories") : ClassLoader.getSystemResources("META-INF/spring.factories");
 while(urls.hasMoreElements()) {
 URL url = urls.nextElement();
 URLResource resource = new URLResource(url);
 Properties properties = PropertiesLoaderUtils.loadProperties(resource);
 Iterator var6 = properties.entrySet().iterator();

 while(var6.hasNext()) {
 Entry<String, String> entry = (Entry<String, String>) var6.next();
 String factoryClassName = ((StringEntry)entry.getKey()).trim();
 String[] var9 = StringUtils.commaDelimitedListToTrimArray((StringEntry)entry.getValue());
 int var10 = var9.length;

 for(int var11 = 0; var11 < var10; ++var11) {
 String factoryName = var9[var11];
 result.add(factoryClassName, factoryName.trim());
 }
 }
 }
 } catch (IOException var13) {
 throw new IllegalArgumentException("Unable to load factories from location " + "META-INF/spring.factories", var13);
 }
 }
}
```

## META-INF/spring.factories 探究

听我这样说完可能会有点懵，我们来看一下 `META-INF/spring.factories` 这类文件是什么就不懵了。当然在很多第三方依赖中都会有这个文件，一般每导入一个第三方的依赖，除了本身的jar包以外，还会有一个 `xxx-spring-boot-autoConfigure`，这个就是第三方依赖自己编写的自动配置类。我们现在就以 `spring-boot-autoconfigure` 这个依赖来说。微信搜索 `web_resource 回复爆文送你高质量技术博文`。



可以看到 `EnableAutoConfiguration` 下面有很多类，这些就是我们项目进行自动配置的类。

一句话：将类路径下 `META-INF/spring.factories` 里面配置的所有 `EnableAutoConfiguration` 的值加入到 Spring 容器中。

## HttpEncodingAutoConfiguration

通过上面方式，所有的自动配置类就被导进主配置类中了。但是这么多的配置类，明显有很多自动配置我们平常是没有使用到的，没理由全部都生效吧。

接下来我们以 `HttpEncodingAutoConfiguration` 为例来看一个自动配置类是怎么工作的。为啥选这个类呢？主要是这个类比较的简单典型。

先看一下该类标有的注解：

```

1 @Configuration
2 @EnableConfigurationProperties({HttpProperties.class})
3 @ConditionalOnWebApplication(
4 type = Type.SERVLET
5)
6 @ConditionalOnClass({CharacterEncodingFilter.class})
7 @ConditionalOnProperty(
8 prefix = "spring.http.encoding",
9
10 value = {"enabled"},
11
12 matchIfMissing = true
13)
14 public class HttpEncodingAutoConfiguration {
15
}
```

- `@Configuration`: 标记为配置类。
- `@ConditionalOnWebApplication`: web 应用下才生效。
- `@ConditionalOnClass`: 指定的类（依赖）存在才生效。
- `@ConditionalOnProperty`: 主配置文件中存在指定的属性才生效。

- `@EnableConfigurationProperties({HttpProperties.class})`: 启动指定类的ConfigurationProperties功能；将配置文件中对应的值和HttpProperties绑定起来；并把HttpProperties加入到IOC容器中。

因为 `@EnableConfigurationProperties({HttpProperties.class})` 把配置文件中的配置项与当前HttpProperties类绑定上了。

然后在HttpEncodingAutoConfiguration中又引用了HttpProperties，所以最后就能在HttpEncodingAutoConfiguration中使用配置文件中的值了。

最终通过@Bean和一些条件判断往容器中添加组件，实现自动配置。（当然该Bean中属性值是从HttpProperties中获取）

## HttpProperties

HttpProperties通过@ConfigurationProperties注解将配置文件与自身属性绑定。

所有在配置文件中能配置的属性都是在xxxProperties类中封装着；配置文件能配置什么就可以参照某个功能对应的这个属性类。微信搜索web\_resource回复爆文送你高质量技术博文。

```
1 @ConfigurationProperties(
2 prefix = "spring.http"
3)
4 // 从配置文件中获取指定的值和bean的属性进行绑定
5 public class HttpProperties {
}
```

## 小结：

- SpringBoot启动会加载大量的自动配置类。
- 我们看需要的功能有没有SpringBoot默认写好的自动配置类。
- 我们再来看这个自动配置类中到底配置了那些组件（只要我们要用的组件有，我们就不需要再来配置了）。
- 给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们就可以在配置文件中指定这些属性的值。  
`xxxAutoConfiguration`: 自动配置类给容器中添加组件。  
`xxxProperties`: 封装配置文件中相关属性。

不知道小伙伴们有没有发现，很多需要待加载的类都放在类路径下的META-INF/Spring.factories文件下，而不是直接写死这代码中，这样做就可以很方便我们自己或者是第三方去扩展，我们也可以实现自己starter，让SpringBoot去加载。现在明白为什么SpringBoot可以实现零配置，开箱即用了吧！

-END-

如果看到这里，说明你喜欢这篇文章，帮忙[转发](#)一下吧，感谢。微信搜索「web\_resource」，关注后回复「进群」即可进入无广告交流群。

↓ 扫描二维码进群 ↓



扫一扫上面的二维码图案，加我微信

### 推荐阅读

1. 经济疲软，公司盈利困难，程序员这样也能涨工资？
2. 标星1.3W！GitHub热榜第一，全网最牛掰的12306抢票神器
3. Java 最常见的 208 道面试题
4. 在浏览器输入 URL 回车之后发生了什么？
5. 接私活必备的 10 个开源项目



Java后端

长按识别二维码，关注我的公众号

喜欢文章，点个在看