

10分钟实现Java发送邮件功能

yizhiwazi Java后端 2月23日



微信搜一搜

Java后端

作者 | yizhiwazi

链接 | jianshu.com/p/5eb000544dd7

Spring Boot集成邮件服务竟如此简单，快速掌握邮件业务类的核心逻辑和企业邮件的日常服务。

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。

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 {  
  
    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);  
  
    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) 为上下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。

首页

通讯录

收件箱

设置

常规设置

邮箱密码修改

签名/电子名片

来信分类

帐号与邮箱中心

邮箱安全设置

邮箱手机服务

反垃圾/黑白名单

POP3/SMTP/IMAP

客户端授权密码

文件夹和标签

多标签窗口

邮箱触点

信纸

换肤

□ 手机号码邮箱

POP3/SMTP/IMAP

设置POP3/SMTP/IMAP:

 POP3/SMTP服务 IMAP/SMTP服务

收取最近30天邮件 ▾

温馨提示: 请使用授权码登录第三方邮件客户端

设置POP3/SMTP/IMAP:

 开启客户端删除邮件提醒

当邮件客户端删除邮件时, 系统会通过邮件发送提醒信息

保存

取消

提示

服务器地址: POP3服务器: pop.163.com

SMTP服务器: smtp.163.com

IMAP服务器: imap.163.com

网易官方邮件客户端: 网易邮箱大师

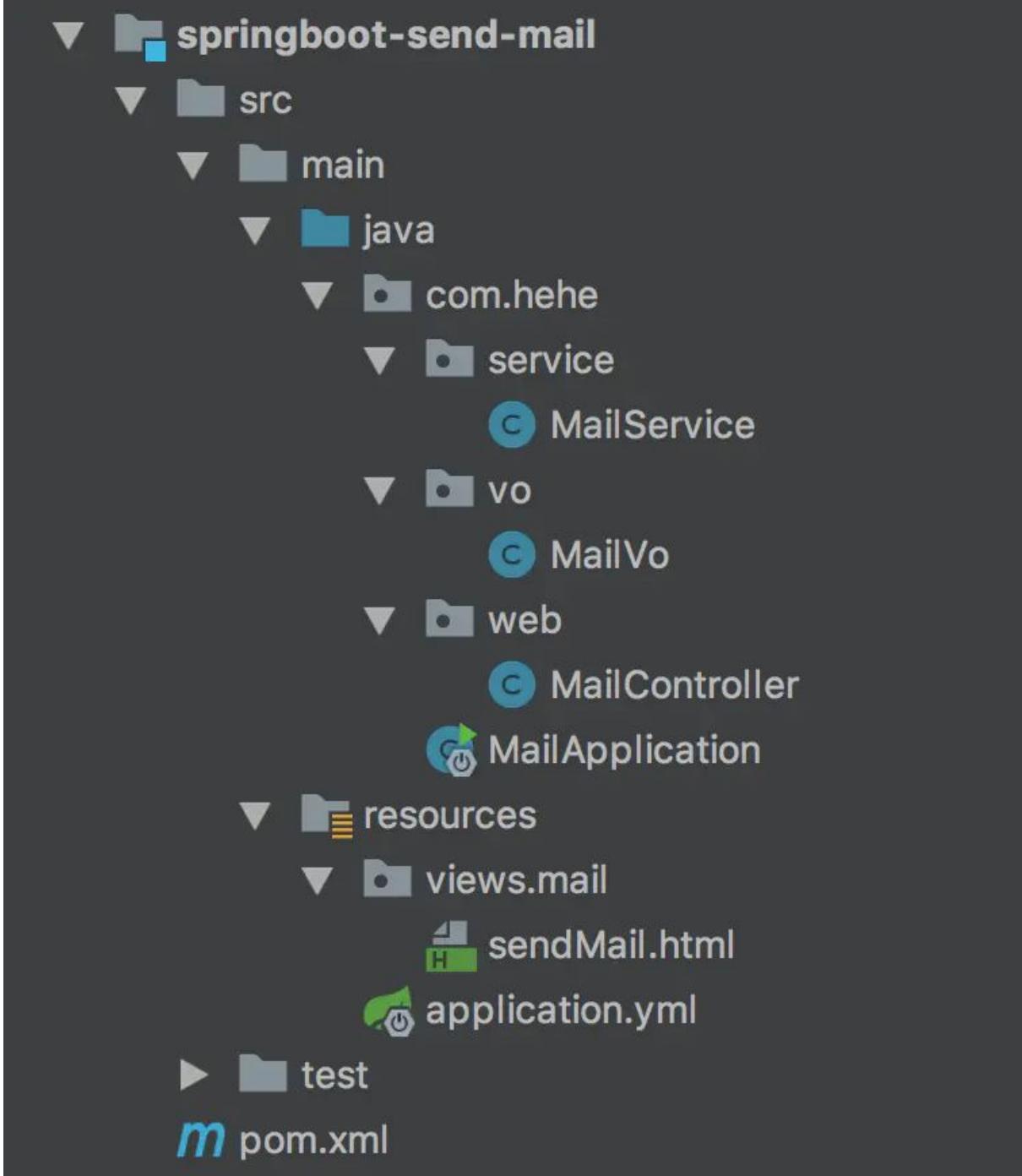
其他邮件客户端: PC端设置帮助

移动端设置帮助 (iOS、Android、Symbian)

安全支持: POP3/SMTP/IMAP服务全部支持SSL连接

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。

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;
    private String from;
    private String to;
    private String subject;
    private String text;
    private Date sentDate;
    private String cc;
    private String bcc;
    private String status;
    private String error;
    @JsonIgnore
    private MultipartFile[] multipartFiles;
}

}
```

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 saveMail(); 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;

@Service
public class MailService {

    private Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    private JavaMailSenderImpl mailSender;

    public MailVo sendMail(MailVo mailVo) {
        try {
            checkMail(mailVo);
            sendMimeMail(mailVo);
            return saveMail(mailVo);
        } catch (Exception e) {
            logger.error("Error sending email: " + e.getMessage());
        }
    }

    private void checkMail(MailVo mailVo) {
        if (mailVo.getFrom() == null || mailVo.getFrom().isEmpty()) {
            throw new IllegalArgumentException("From address cannot be empty");
        }
        if (mailVo.getSubject() == null || mailVo.getSubject().isEmpty()) {
            throw new IllegalArgumentException("Subject cannot be empty");
        }
        if (mailVo.getText() == null || mailVo.getText().isEmpty()) {
            throw new IllegalArgumentException("Text cannot be empty");
        }
    }

    private void sendMimeMail(MailVo mailVo) {
        MimeMessage message = mailSender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(message, true, "UTF-8");
        helper.setFrom(mailVo.getFrom());
        helper.setTo(mailVo.getTo());
        helper.setSubject(mailVo.getSubject());
        helper.setText(mailVo.getText());
        helper.setCc(mailVo.getCc());
        helper.setBcc(mailVo.getBcc());
        helper.setStatus(mailVo.getStatus());
        helper.setError(mailVo.getError());
        helper.setSentDate(mailVo.getSentDate());
        helper.setMultipartFiles(mailVo.getMultipartFiles());
        mailSender.send(message);
    }

    private MailVo saveMail(MailVo mailVo) {
        // Logic to save the mail to the database
        return mailVo;
    }
}
```

```

}, catch (Exception e) {
    logger.error("发送邮件失败:", e);
    mailVo.setStatus("fail");
    mailVo.setError(e.getMessage());
    return mailVo;
}

}

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");
    }
}

private void sendMimeMail(MailVo mailVo) {
    try {
        MimeMessageHelper messageHelper = new MimeMessageHelper(mailSender.createMimeMessage(), true);
        mailVo.setFrom(getMailSendFrom());
        messageHelper.setFrom(mailVo.getFrom());
        messageHelper.setTo(mailVo.getTo().split(","));
        messageHelper.setSubject(mailVo.getSubject());
        messageHelper.setText(mailVo.getText());
        if (!StringUtils.isEmpty(mailVo.getCc())) {
            messageHelper.setCc(mailVo.getCc().split(","));
        }
        if (!StringUtils.isEmpty(mailVo.getBcc())) {
            messageHelper.setCc(mailVo.getBcc().split(","));
        }
        if (mailVo.getMultipartFiles() != null) {
            for (MultipartFile multipartFile : mailVo.getMultipartFiles()) {
                messageHelper.addAttachment(multipartFile.getOriginalFilename(), multipartFile);
            }
        }
        if (StringUtils.isEmpty(mailVo.getSentDate())) {
            mailVo.setSentDate(new Date());
            messageHelper.setSentDate(mailVo.getSentDate());
        }
        mailSender.send(messageHelper.getMimeMessage());
        mailVo.setStatus("ok");
        logger.info("发送邮件成功: {}->{}", mailVo.getFrom(), mailVo.getTo());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

private MailVo saveMail(MailVo mailVo) {

    return mailVo;
}

public String getMailSendFrom() {
    return mailSender.getJavaMailProperties().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;

    @GetMapping("/")
    public ModelAndView index() {
        ModelAndView mv = new ModelAndView("mail/sendMail");
        mv.addObject("from", mailService.getMailSendFrom());
        return mv;
    }

    @PostMapping("/mail/send")
    public MailVo sendMail(MailVo mailVo, MultipartFile[] files) {
        mailVo.setMultipartFiles(files);
        return mailService.sendMail(mailVo);
    }
}
```

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:src="@{/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">
```

```

<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)')
    }
}, 100);

```

```
});  
</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。源码获取可以本公众号「Java后端」后台回复**邮件**获取。

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:

邮件收信人:

邮件主题:

邮件内容:

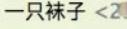
邮件附件: doge.gif

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:  <maveni_work@163.com> [举报](#)
时间: 2019年1月6日(星期天) 凌晨1:50
收件人: 一只袜子 <@qq.com>
附件: 1 个 ( doge.gif)

这不是腾讯公司的官方邮件②。 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)
[预览](#) [下载](#) [收藏](#) [转存](#)

邮件附件发送成功！！ 

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；

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 .bxh .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

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」，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

[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！

18个Java 8日期处理的工具类

胖先森 Java后端 1月20日

来源: juejin.im/post/5a795bad6fb9a0634f407ae

Java 8 推出了全新的日期时间API，在教程中我们将通过一些简单的实例来学习如何使用新API。

Java处理日期、日历和时间的方式一直为社区所诟病，将 `java.util.Date` 设定为可变类型，以及 `SimpleDateFormat` 的非线程安全使其应用非常受限。

新API基于ISO标准日历系统，`java.time`包下的所有类都是不可变类型而且线程安全。

编号	类的名称	描述
1	<code>Instant</code>	时间截
2	<code>Duration</code>	持续时间，时间差
3	<code>LocalDate</code>	只包含日期，比如：2018-02-05
4	<code>LocalTime</code>	只包含时间，比如：23:12:10
5	<code>LocalDateTime</code>	包含日期和时间，比如：2018-02-05 23:14:21
6	<code>Period</code>	时间段
7	<code>ZoneOffset</code>	时区偏移量，比如：+8:00
8	<code>ZonedDateTime</code>	带时区的时间
9	<code>Clock</code>	时钟，比如获取目前美国纽约的时间
10	<code>java.time.format.DateTimeFormatter</code>	时间格式化

示例1:Java 8中获取今天的日期

Java 8 中的 `LocalDate` 用于表示当天日期。和 `java.util.Date` 不同，它只有日期，不包含时间。当你仅需要表示日期时就用这个类。

```
package com.shxt.demo02;  
  
import java.time.LocalDate;  
  
public class Demo01 {  
    public static void main(String[] args) {  
        LocalDate today = LocalDate.now();  
        System.out.println("今天的日期:" + today);  
    }  
}
```

示例2:Java 8中获取年、月、日信息

```
package com.shxt.demo02;

import java.time.LocalDate;

public class Demo02{
    public static void main( String[] args){
        LocalDate today=LocalDate.now();
        int year=today.getYear();
        int month=today.getMonthValue();
        int day=today.getDayOfMonth();

        System.out.println("year:"+year);
        System.out.println("month:"+month);
        System.out.println("day:"+day);

    }
}
```

示例3:Java 8中处理特定日期

我们通过静态工厂方法now()非常容易地创建了当天日期，你还可以调用另一个有用的工厂方法LocalDate.of()创建任意日期，该方法需要传入年、月、日做参数，返回对应的LocalDate实例。这个方法的好处是没再犯老API的设计错误，比如年度起始于1900，月份是从0开始等等。

```
package com.shxt.demo02;

import java.time.LocalDate;

public class Demo03 {
    public static void main(String[] args) {
        LocalDate date=LocalDate.of( 2018,2,6);
        System.out.println("自定义日期:"+date);
    }
}
```

示例4:Java 8中判断两个日期是否相等

```
package com.shxt.demo02;

import java.time.LocalDate;

public class Demo04{
    public static void main(String[] args){
        LocalDate date1=LocalDate.now();

        LocalDate date2=LocalDate.of( 2018,2,5);

        if(date1.equals(date2)){
            System.out.println("时间相等");
        } else{
            System.out.println("时间不等");
        }

    }
}
```

示例5:Java 8中检查像生日这种周期性事件

```
package com.shxt.demo02;

import java.time.LocalDate;
import java.time.MonthDay;

public class Demo05{
    public static void main(String[] args){
        LocalDate date1=LocalDate.now();

        LocalDate date2=LocalDate.of( 2018,2,6);
        MonthDay birthday=MonthDay.of(date2.getMonth(),date2.getDayOfMonth());
        MonthDay currentMonthDay=MonthDay.from(date1);

        if(currentMonthDay.equals(birthday)){
            System.out.println("是你的生日");
        } else{
            System.out.println("你的生日还没有到");
        }

    }
}
```

只要当天的日期和生日匹配，无论是哪一年都会打印出祝贺信息。你可以把程序整合进系统时钟，看看生日时是否会受到提醒，或者写一个单元测试来检测代码是否运行正确。

示例6:Java 8中获取当前时间

```
package com.shxt.demo02;  
import java.time.LocalTime;  
  
public class Demo06 {  
    public static void main(String[] args) {  
        LocalTimetime=LocalTime.now();  
        System.out.println("获取当前的时间,不含有日期:"+time);  
    }  
}
```

可以看到当前时间就只包含时间信息，没有日期

示例7:Java 8中获取当前时间

通过增加小时、分、秒来计算将来的时间很常见。Java 8除了不变类型和线程安全的好处之外，还提供了更好的plusHours()方法替换add()，并且是兼容的。注意，这些方法返回一个全新的LocalTime实例，由于其不可变性，返回后一定要用变量赋值。

```
package com.shxt.demo02;  
import java.time.LocalTime;  
  
public class Demo07 {  
    public static void main(String[] args) {  
        LocalTimetime=LocalTime.now();  
        LocalTimenewTime=time.plusHours(3);  
        System.out.println("三个小时后的时间为:"+newTime);  
    }  
}
```

示例8:Java 8如何计算一周后的日期

和上个例子计算3小时以后的时间类似，这个例子会计算一周后的日期。LocalDate日期不包含时间信息，它的plus()方法用来增加天、周、月，ChronoUnit类声明了这些时间单位。由于LocalDate也是不变类型，返回后一定要用变量赋值。

```
package com.shxt.demo02;  
import java.time.LocalDate;  
import java.time.temporal.ChronoUnit;  
  
public class Demo08 {  
    public static void main(String[] args) {  
        LocalDatedate=LocalDate.now();  
        System.out.println("今天的日期为:"+date);  
        LocalDatenextWeek=date.plus(1,ChronoUnit.WEEKS);  
        System.out.println("一周后的日期为:"+nextWeek);  
    }  
}
```

可以看到新日期离当天日期是7天，也就是一周。你可以用同样的方法增加1个月、1年、1小时、1分钟甚至一个世纪，更多选项可以查看Java 8 API中的ChronoUnit类

示例9:Java 8计算一年前或一年后的日期

利用minus()方法计算一年前的日期

```

package com.shxt.demo02;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Demo09 {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();

        LocalDate previousYear = today.minus(1, ChronoUnit.YEARS);
        System.out.println("一年前的日期:" + previousYear);

        LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);
        System.out.println("一年后的日期:" + nextYear);
    }
}

```

示例10:Java 8的Clock时钟类

Java 8增加了一个Clock时钟类用于获取当时的时间戳，或当前时区下的日期时间信息。以前用到System.currentTimeMillis()和TimeZone.getDefault()的地方都可用Clock替换。

```

package com.shxt.demo02;

import java.time.Clock;

public class Demo10 {
    public static void main(String[] args) {
        //Returnsthe current time based on your system clock and set to UTC.
        Clock clock = Clock.systemUTC();
        System.out.println("Clock:" + clock.millis());

        //Returnsthe current time based on your system clock zone
        Clock defaultClock = Clock.systemDefaultZone();
        System.out.println("Clock:" + defaultClock.millis());
    }
}

```

示例11:如何用Java判断日期是早于还是晚于另一个日期

另一个工作中常见的操作就是如何判断给定的一个日期是大于某天还是小于某天？在Java 8中，`LocalDate`类有两类方法`isBefore()`和`isAfter()`用于比较日期。调用`isBefore()`方法时，如果给定日期小于当前日期则返回true。

```

package com.shxt.demo02;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Demo11 {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();

        LocalDate tomorrow = LocalDate.of(2018, 2, 6);
        if(tomorrow.isAfter(today)){
            System.out.println("之后的日期:" + tomorrow);
        }

        LocalDate yesterday = today.minus(1, ChronoUnit.DAYS);
        if(yesterday.isBefore(today)){
            System.out.println("之前的日期:" + yesterday);
        }
    }
}

```

示例12:Java 8中处理时区

Java 8不仅分离了日期和时间，也把时区分离出来了。现在有一系列单独的类如`ZoneId`来处理特定时区，`ZoneDateTime`类来表示某时区下的时间。这在Java 8以前都是`GregorianCalendar`类来做的。下面这个例子展示了如何把本时区的时间转换成另一个时区的时间。

```

package com.shxt.demo02;

import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class Demo12 {
    public static void main(String[] args) {
        // Date and time with timezone in Java 8
        ZoneId america = ZoneId.of("America/New_York");
        LocalDateTime localDateTime = LocalDateTime.now();
        ZonedDateTime dateAndTimeInNewYork = ZonedDateTime.of(localDateTime, america);
        System.out.println("Current date and time in a particular timezone:" + dateAndTimeInNewYork);
    }
}

```

示例13:如何表示信用卡到期这类固定日期，答案就在YearMonth

与 MonthDay检查重复事件的例子相似，YearMonth是另一个组合类，用于表示信用卡到期日、FD到期日、期货期权到期日等。还可以用这个类得到当月共有多少天，YearMonth实例的lengthOfMonth()方法可以返回当月的天数，在判断2月有28天还是29天时非常有用。

```

package com.shxt.demo02;

import java.time.*;

public class Demo13{
    public static void main(String[] args) {
        YearMonth currentYearMonth=YearMonth.now();
        System.out.printf("Days in month %s: %d%n", currentYearMonth, currentYearMonth.lengthOfMonth());
        YearMonth creditCardExpiry=YearMonth.of(2019, Month.FEBRUARY);
        System.out.printf("Your credit card expires on %s%n", creditCardExpiry);
    }
}

```

示例14:如何在Java 8中检查闰年

```

package com.shxt.demo02;

import java.time.LocalDate;

public class Demo14{
    public static void main(String[] args){
        LocalDate today=LocalDate.now();
        if(today.isLeapYear()){
            System.out.println("This year is a leap year");
        } else{
            System.out.println("2018 is not a leap year");
        }
    }
}

```

示例15:计算两个日期之间的天数和月数

有一个常见日期操作是计算两个日期之间的天数、周数或月数。在Java 8中可以用java.time.Period类来做计算。

下面这个例子中，我们计算了当天和将来某一天之间的月数。

```

package com.shxt.demo02;

import java.time.LocalDate;
import java.time.Period;

public class Demo15 {
    public static void main(String[] args) {
        LocalDate today=LocalDate.now();

        LocalDate java8Release=LocalDate.of(2018, 12, 14);

        Period periodToNextJavaRelease=Period.between(today, java8Release);
        System.out.println("Months left between today and Java 8 release:"
            + periodToNextJavaRelease.getMonths());
    }
}

```

示例16:在Java 8中获取当前的时间戳

Instant类有一个静态工厂方法now()会返回当前的时间戳，如下所示：

```
package com.shxt.demo02;  
import java.time.Instant;  
  
public class Demo16 {  
    public static void main(String[] args) {  
        Instant timestamp=Instant.now();  
        System.out.println("What is value of this instant"+timestamp.toEpochMilli());  
    }  
}
```

时间戳信息里同时包含了日期和时间，这和java.util.Date很像。实际上Instant类确实等同于 Java 8之前的Date类，你可以使用Date类和Instant类各自的转换方法互相转换，例如：Date.from(Instant) 将Instant转换成java.util.Date，Date.toInstant()则是将Date类转换成Instant类。

示例17:Java 8中如何使用预定义的格式化工具去解析或格式化日期

```
package com.shxt.demo02;  
  
import java.time.LocalDate;  
import java.time.format.DateTimeFormatter;  
  
public class Demo17 {  
    public static void main(String[] args) {  
        String dayAfterTommorrow="20180205";  
        LocalDate formatted=LocalDate.parse(dayAfterTommorrow,  
            DateTimeFormatter.BASIC_ISO_DATE);  
        System.out.println(dayAfterTommorrow+" 格式化后的日期为: "+formatted);  
    }  
}
```

示例18:字符串互转日期类型

```
package com.shxt.demo02;  
  
import java.time.LocalDate;  
import java.time.LocalDateTime;  
import java.time.format.DateTimeFormatter;  
  
public class Demo18 {  
    public static void main(String[] args) {  
        LocalDateTime date=LocalDateTime.now();  
  
        DateTimeFormatter format1=DateTimeFormatter.ofPattern("yyyy/MM/ddHH:mm:ss");  
        //日期转字符串  
        String str=date.format(format1);  
  
        System.out.println("日期转换为字符串:"+str);  
  
        DateTimeFormatter format2=DateTimeFormatter.ofPattern("yyyy/MM/ddHH:mm:ss");  
        //字符串转日期  
        LocalDate date2=LocalDate.parse(str,format2);  
        System.out.println("日期类型:"+date2);  
    }  
}
```



微信扫描二维码，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！

18个示例带你掌握Java 8日期时间处理！

胖先森 Java后端 2月21日



微信搜一搜

Java后端

作者 | 胖先森

链接 | juejin.im/post/5a795bad6fb9a0634f407ae5

Java 8 推出了全新的日期时间API，在教程中我们将通过一些简单的实例来学习如何使用新API。

Java处理日期、日历和时间的方式一直为社区所诟病，将 `java.util.Date` 设定为可变类型，以及 `SimpleDateFormat` 的非线程安全使其应用非常受限。

新API基于ISO标准日历系统，`java.time` 包下的所有类都是不可变类型而且线程安全。

类的名称	描述
<code>Instant</code>	时间戳
<code>Duration</code>	持续时间，时间差
<code>LocalDate</code>	只包含日期，比如：2018-02-05
<code>LocalTime</code>	只包含时间，比如：23:12:10
<code>LocalDateTime</code>	包含日期和时间，比如：2018-02-05 23:14:21
<code>Period</code>	时间段
<code>ZoneOffset</code>	时区偏移量，比如：+8:00
<code>ZonedDateTime</code>	带时区的时间
<code>Clock</code>	时钟，比如获取目前美国纽约的时间
<code>java.time.format.DateTimeFormatter</code>	时间格式化

示例1:Java 8中获取今天的日期

Java 8 中的 `LocalDate` 用于表示当天日期。和 `java.util.Date` 不同，它只有日期，不包含时间。当你仅需要表示日期时就用这个类。

```
package com.shxt.demo02;

import java.time.LocalDate;

public class Demo01 {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        System.out.println("今天的日期:" + today);
    }
}
```

示例2:Java 8中获取年、月、日信息

```
package com.shxt.demo02;

import java.time.LocalDate;

public class Demo02 {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        int year = today.getYear();
        int month = today.getMonthValue();
        int day = today.getDayOfMonth();

        System.out.println("year:" + year);
        System.out.println("month:" + month);
        System.out.println("day:" + day);

    }
}
```

示例3:Java 8中处理特定日期

我们通过静态工厂方法now()非常容易地创建了当天日期，你还可以调用另一个有用的工厂方法LocalDate.of()创建任意日期，该方法需要传入年、月、日做参数，返回对应的LocalDate实例。

这个方法的好处是没再犯老API的设计错误，比如年度起始于1900，月份是从0开始等等。

```
import java.time.LocalDate;

public class Demo03 {
    public static void main(String[] args) {
        LocalDate date = LocalDate.of(2018,2,6);
        System.out.println("自定义日期:" + date);
    }
}
```

示例4:Java 8中判断两个日期是否相等

```
import java.time.LocalDate;

public class Demo04 {
    public static void main(String[] args) {
        LocalDate date1 = LocalDate.now();

        LocalDate date2 = LocalDate.of(2018,2,5);

        if(date1.equals(date2)){
            System.out.println("时间相等");
        }else{
            System.out.println("时间不等");
        }

    }
}
```

示例5:Java 8中检查像生日这种周期性事件

```

import java.time.LocalDate;
import java.time.MonthDay;

public class Demo05 {
    public static void main(String[] args) {
        LocalDate date1 = LocalDate.now();

        LocalDate date2 = LocalDate.of(2018,2,6);
        MonthDay birthday = MonthDay.of(date2.getMonth(),date2.getDayOfMonth());
        MonthDay currentMonthDay = MonthDay.from(date1);

        if(currentMonthDay.equals(birthday)){
            System.out.println("是你的生日");
        }else{
            System.out.println("你的生日还没有到");
        }
    }
}

```

只要当天的日期和生日匹配，无论是哪一年都会打印出祝贺信息。你可以把程序整合进系统时钟，看看生日时是否会受到提醒，或者写一个单元测试来检测代码是否运行正确。

示例6:Java 8中获取当前时间

```

import java.time.LocalTime;

public class Demo06 {
    public static void main(String[] args) {
        LocalTime time = LocalTime.now();
        System.out.println("获取当前的时间,不含有日期:" +time);

    }
}

```

可以看到当前时间就只包含时间信息，没有日期

示例7:Java 8中获取当前时间

通过增加小时、分、秒来计算将来的时间很常见。Java 8除了不变类型和线程安全的好处之外，还提供了更好的plusHours()方法替换add()，并且是兼容的。

注意，这些方法返回一个全新的LocalTime实例，由于其不可变性，返回后一定要用变量赋值。

```

import java.time.LocalTime;

public class Demo07 {
    public static void main(String[] args) {
        LocalTime time = LocalTime.now();
        LocalTime newTime = time.plusHours(3);
        System.out.println("三个小时后的时间为:" +newTime);

    }
}

```

示例8:Java 8如何计算一周后的日期

和上个例子计算3小时以后的时间类似，这个例子会计算一周后的日期。LocalDate日期不包含时间信息，它的plus()方法用来增

加天、周、月，ChronoUnit类声明了这些时间单位。由于LocalDate也是不变类型，返回后一定要用变量赋值。

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Demo08 {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        System.out.println("今天的日期为:" + today);
        LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
        System.out.println("一周后的日期为:" + nextWeek);

    }
}
```

可以看到新日期离当天日期是7天，也就是一周。你可以用同样的方法增加1个月、1年、1小时、1分钟甚至一个世纪，更多选项可以查看Java 8 API中的ChronoUnit类。

示例9:Java 8计算一年前或一年后的日期

利用minus()方法计算一年前的日期

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Demo09 {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();

        LocalDate previousYear = today.minus(1, ChronoUnit.YEARS);
        System.out.println("一年前的日期：" + previousYear);

        LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);
        System.out.println("一年后的日期：" + nextYear);

    }
}
```

示例10:Java 8的Clock时钟类

Java 8增加了一个Clock时钟类用于获取当时的时间戳，或当前时区下的日期时间信息。以前用到System.currentTimeMillis()和TimeZone.getDefault()的地方都可用Clock替换。

```
import java.time.Clock;

public class Demo10 {
    public static void main(String[] args) {

        Clock clock = Clock.systemUTC();
        System.out.println("Clock：" + clock.millis());

        Clock defaultClock = Clock.systemDefaultZone();
        System.out.println("Clock：" + defaultClock.millis());

    }
}
```

示例11:如何用Java判断日期是早于还是晚于另一个日期

另一个工作中常见的操作就是如何判断给定的一个日期是大于某天还是小于某天？在Java 8中，`LocalDate`类有两类方法`isBefore()`和`isAfter()`用于比较日期。调用`isBefore()`方法时，如果给定日期小于当前日期则返回true。

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Demo11 {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();

        LocalDate tomorrow = LocalDate.of(2018,2,6);
        if(tomorrow.isAfter(today)){
            System.out.println("之后的日期:" +tomorrow);
        }

        LocalDate yesterday = today.minus(1, ChronoUnit.DAYS);
        if(yesterday.isBefore(today)){
            System.out.println("之前的日期:" +yesterday);
        }
    }
}
```

示例12:Java 8中处理时区

Java 8不仅分离了日期和时间，也把时区分离出来了。现在有一系列单独的类如`ZonedDateTime`来处理特定时区，`ZoneDateTime`类来表示某时区下的时间。

关注微信公众号：Java技术栈，在后台回复：新特性，可以获取我整理的N篇最新Java新特性教程，都是干货。

这在Java 8以前都是`GregorianCalendar`类来做的。下面这个例子展示了如何把本时区的时间转换成另一个时区的时间。

```
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class Demo12 {
    public static void main(String[] args) {

        ZoneId america = ZoneId.of("America/New_York");
        LocalDateTime localtDateAndTime = LocalDateTime.now();
        ZonedDateTime dateAndTimeInNewYork = ZonedDateTime.of(localetDateAndTime, america );
        System.out.println("Current date and time in a particular timezone :" + dateAndTimeInNewYork);
    }
}
```

示例13:如何表示信用卡到期这类固定日期，答案就在YearMonth

与`MonthDay`检查重复事件的例子相似，`YearMonth`是另一个组合类，用于表示信用卡到期日、FD到期日、期货期权到期日等。

还可以用这个类得到当月共有多少天，`YearMonth`实例的`lengthOfMonth()`方法可以返回当月的天数，在判断2月有28天还是29天时非常有用。

```
import java.time.*;  
  
public class Demo13 {  
    public static void main(String[] args) {  
        YearMonth currentYearMonth = YearMonth.now();  
        System.out.printf("Days in month year %s: %d%n", currentYearMonth, currentYearMonth.lengthOfMonth());  
        YearMonth creditCardExpiry = YearMonth.of(2019, Month.FEBRUARY);  
        System.out.printf("Your credit card expires on %s %n", creditCardExpiry);  
    }  
}
```

示例14:如何在Java 8中检查闰年

```
import java.time.LocalDate;  
  
public class Demo14 {  
    public static void main(String[] args) {  
        LocalDate today = LocalDate.now();  
        if(today.isLeapYear()){  
            System.out.println("This year is Leap year");  
        }else {  
            System.out.println("2018 is not a Leap year");  
        }  
    }  
}
```

示例15:计算两个日期之间的天数和月数

有一个常见日期操作是计算两个日期之间的天数、周数或月数。在Java 8中可以用java.time.Period类来做计算。下面这个例子中，我们计算了当天和将来某一天之间的月数。

```
import java.time.LocalDate;  
import java.time.Period;  
  
public class Demo15 {  
    public static void main(String[] args) {  
        LocalDate today = LocalDate.now();  
  
        LocalDate java8Release = LocalDate.of(2018, 12, 14);  
  
        Period periodToNextJavaRelease = Period.between(today, java8Release);  
        System.out.println("Months left between today and Java 8 release : "  
            + periodToNextJavaRelease.getMonths());  
  
    }  
}
```

示例16:在Java 8中获取当前的时间戳

Instant类有一个静态工厂方法now()会返回当前的时间戳，如下所示：

```
package com.shxt.demo02;

import java.time.Instant;

public class Demo16 {
    public static void main(String[] args) {
        Instant timestamp = Instant.now();
        System.out.println("What is value of this instant " + timestamp.toEpochMilli());
    }
}
```

时间戳信息里同时包含了日期和时间，这和java.util.Date很像。实际上Instant类确实等同于 Java 8之前的Date类，你可以使用Date类和Instant类各自的转换方法互相转换，例如：Date.from(Instant) 将Instant转换成java.util.Date, Date.toInstant()则是将Date类转换成Instant类。

示例17:Java 8中如何使用预定义的格式化工具去解析或格式化日期

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class Demo17 {
    public static void main(String[] args) {
        String dayAfterTommorrow = "20180205";
        LocalDate formatted = LocalDate.parse(dayAfterTommorrow,
            DateTimeFormatter.BASIC_ISO_DATE);
        System.out.println(dayAfterTommorrow+" 格式化后的日期为: "+formatted);
    }
}
```

示例18:字符串互转日期类型

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Demo18 {
    public static void main(String[] args) {
        LocalDateTime date = LocalDateTime.now();

        DateTimeFormatter format1 = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");

        String str = date.format(format1);

        System.out.println("日期转换为字符串:"+str);

        DateTimeFormatter format2 = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");

        LocalDate date2 = LocalDate.parse(str,format2);
        System.out.println("日期类型:"+date2);

    }
}
```

作者：胖先森

<https://juejin.im/post/5a795bad6fb9a0634f407ae5>

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. [2020 年 9 大顶级 Java 框架](#)
2. [聊聊 API 网关的作用](#)
3. [Class.forName 和 ClassLoader 有什么区别？](#)
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！

2019年 Github 上最热门的 Java 开源项目

Java后端 2019-12-24

点击上方 Java后端, 选择 **设为星标**

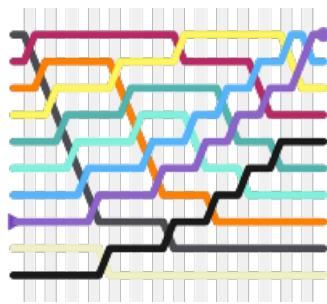
优质文章, 及时送达

来源:开源最前线(ID:OpenSourceTop)

10月份 GitHub 上最热门的 Java 开源项目排行已经出炉啦，在本月的名单中，实战项目类居多，当然也有像 JavaGuide 这样学习指南类项目，下面就是本月上榜的10个开源项目：

1 Java

<https://github.com/TheAlgorithms/Java> Star 18468



该项目用 Java 实现的所有算法，对算法感兴趣的小伙伴们不要错过了。

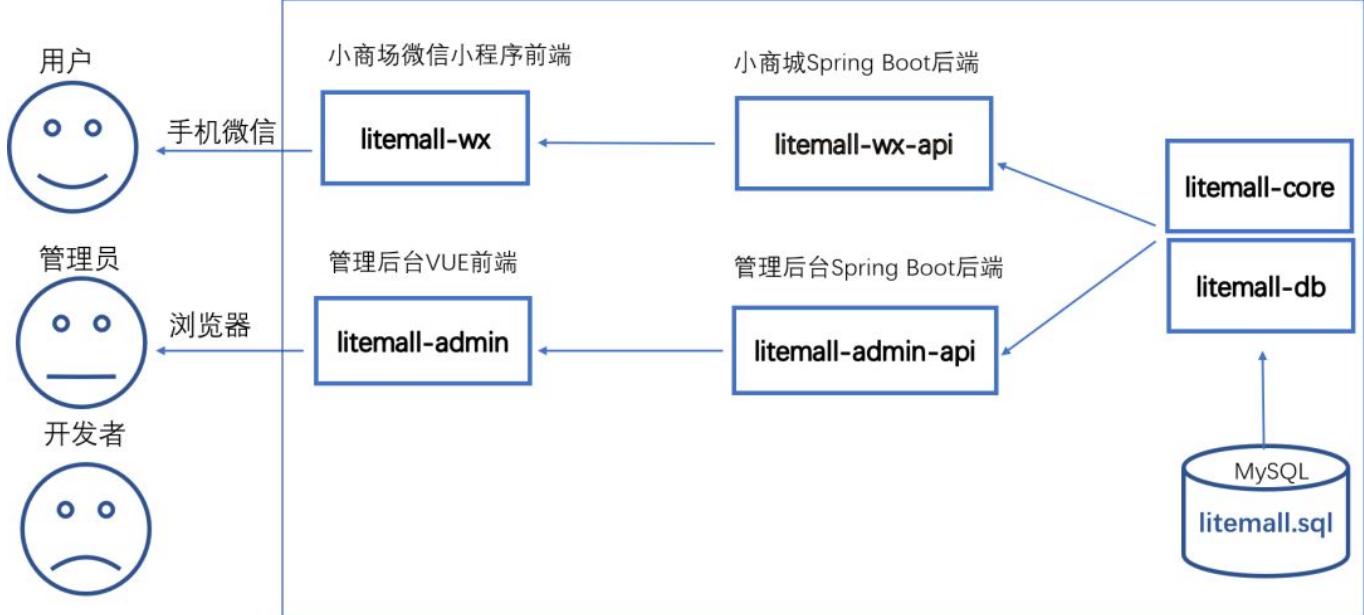
2 eladmin

<https://github.com/elunez/eladmin> Star 4639

该项目基于 Spring Boot 2.1.0、Jpa、Spring Security、redis、Vue 的前后端分离的后台管理系统，项目采用分模块开发方式，权限控制采用 RBAC，支持数据字典与数据权限管理，支持一键生成前后端代码，支持动态路由。

3 mall

<https://github.com/macrozhang/mall> Star 24084



一个小商城。litemall = Spring Boot后端 + Vue管理员前端 + 微信小程序用户前端，由于没有上线，只能在微信开发工具中测试运行。

4 java-design-patterns

<https://github.com/iluwatar/java-design-patterns> Star 52341

Design patterns是程序员在设计应用程序或系统时可用来解决常见问题的最佳实践手册。它可以帮助你加快开发进程，有效防止一些可能导致重大失误的细节问题，不过深入了解java-design-patterns之前，你应提前熟悉各种编程/软件设计原则。

5 JavaGuide

<https://github.com/Snailclimb/JavaGuide> Star 59540

这是一份Java学习指南，涵盖大部分Java程序员所需要掌握的核心知识。

6 paascloud-master

<https://github.com/paascloud/paascloud-master> Star 6302

spring cloud + vue + oAuth2.0全家桶实战，前后端分离模拟商城，完整的购物流程、后端运营平台，可以实现快速搭建企业级微服务项目，支持微信登录等三方登录。

7 netty

<https://github.com/netty/netty> Star 21346

Netty是一个广泛使用Java网络编程框架，Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠的网络服务器和客户端程序。

8 views-widgets-samples

<https://github.com/android/views-widgets-samples> Star 479

该仓库包含一组单独的Android Studio项目，可帮助您开始编写/理解Android视图和小部件功能。

9 ksql

<https://github.com/confluentinc/ksql> Star 2606



KSQL是一个用于Apache kafka的流式SQL引擎，它为Kafka的流处理提供了一个简单而完整的SQL界面；不需要再用编程语言（如Java或Python）编写代码。

10 Mindustry

<https://github.com/Anuken/Mindustry> Star 1644

Mindustry 是一款优秀的开源游戏，玩家可以自行下载源码进行修改，如果你想自己编译，请先确保自己已安装Java 8和JDK 8。

-END-

推荐阅读

1. [发布没有答案的面试题，都是耍流氓](#)
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？](#)



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！

2020 年 9 大顶级 Java 框架

Patricia
Neil

Java后端 2月20日



微信搜一搜

Q Java后端

来源 | Patricia Neil

责编 | Carol

出品 | CSDN云计算 (ID: CSDNcloud)

诞生于1995年的Java，目前已在134,861个网站上广泛使用，包括ESPN、SnapDeal等。在其24年的成长史中，Java已经证明了自己是用于自定义软件开发的顶级通用编程语言。

Java广泛应用于科学教育、金融、法律和政府等行业。在下面的饼图是Java语言在各个行业中的使用情况。



这种开源编程语言是面向对象的，其目的是给予应用程序开发人员编写一次代码就能够再任何地方运行(WORA)的自由。这能够让编译后的Java代码在每个支持Java的平台上都能运行。

最新版本的Java 13于2019年9月发布。根据TOIBE排行榜(基于排名最高的25个搜索引擎计算)，Java位列第一。

以下是2019年11月和2018年11月的编程语言排名榜单：

Nov 2019	Nov 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.246%	-0.50%
2	2		C	16.037%	+1.64%
3	4	▲	Python	9.842%	+2.16%
4	3	▼	C++	5.605%	-2.68%
5	6	▲	C#	4.316%	+0.36%
6	5	▼	Visual Basic .NET	4.229%	-2.26%
7	7		JavaScript	1.929%	-0.73%
8	8		PHP	1.720%	-0.66%
9	9		SQL	1.690%	-0.15%

Java始终排在第一位，这使它成为有史以来最享负盛誉的软件编程语言之一。及时的更新和新版本的发布使它成为一种充满活力的、有竞争力的编程语言。

但是，仅仅为你的下一个web应用程序开发项目选择这门顶级语言是不够的。在选择Java web框架时，你仍需要做出正确的选择。那么，你是否想知道如何为项目选择一个恰当的 Java框架呢？

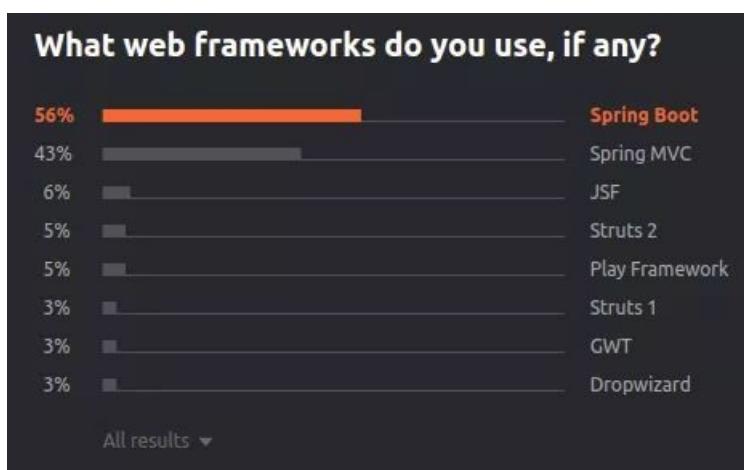
实际上，这并不容易，而且需要深入了解为特定行业业务选择特定java框架进行软件开发的优缺点。

现在，让我们一起来详细研究一下**2020年这9大顶级Java框架**。尝试在本公众号「Java后端」后台回复 `Java`，获取框架学习视频教程。



Spring排在第一位，是由于它能够开发以高性能著称的复杂web应用程序的出色能力。它能够使Java开发人员轻松地创建企业级应用程序。

Web应用程序开发人员可以担保Spring框架的能力。这也是Spring成为Java开发人员的最爱的原因。下面的数据进一步证明了这一点。以下是选择Java作为首选的三种编程语言之一的开发者们对Java框架的看法：



在开发人员的选择中，Spring MVC和Spring Boot远远领先于其他Java技术。对于开发人员来说，这里的一大优势是他们可以不受其他模块约束并专注于一个模块，因为spring利用了控制反转(IoC)。

这个框架的其他优点是：它提供了一个全面的配置模型，支持传统数据库和现代数据库，如NoSQL，并通过支持面向方面的编程实现了内聚开发。它提供了一些模块，如Spring MVC、Spring Core、Spring Boost、SpringTransaction等。



作为一个对象关系映射(ORM)数据库，Hibernate改变了我们以前查看数据库的方式。虽然它不是一个完整的全栈框架，但是它能够为多个数据库轻松转换数据。

它支持多个数据库的能力使得无论应用程序的大小或用户数量如何，都很容易进行扩展。此外，它速度快、功能强大、易于扩展、修改和配置。



该框架能够帮助自定义软件开发人员创建易于维护的企业级应用程序。这个框架的USP就是它的插件。它们是JAR包，这意味着它们是可移植的。

Hibernate 插件和spring 插件分别可以用于对象关系映射和依赖注入。使用此Java框架开发应用程序可以减少处理时间，因为它提供了组织良好的Java、JSP和Action类。



像领英、三星、卫报、威瑞森等顶尖公司都在应用这个框架，但这只能说明它的可信赖度。该框架提供了速度、可伸缩性和性能。

它的用户界面非常简单，能够使移动应用程序开发人员快速上手。它主要用于开发需要统一内容创建的应用程序。





这个框架用于客户端开发，类似Javascript。它是一个开源的Java框架，这意味着它是免费的。Google广泛使用这个框架，旗下的许多产品如AdSense、谷歌钱包、AdWords都是使用它编写的。

借助GWT代码，可以轻松地开发和调试Ajax应用程序。Java开发人员更喜欢这个框架来编写复杂的应用程序。它的一些特性包括书签、跨浏览器可移植性、历史记录和管理。



这个开源框架在Enterprise Java Beans (EJB) 中非常流行。它可用于为内容管理系统、Restful web服务和电子商务网站创建健壮的、可伸缩的应用程序。

它可以与Java Spring、Hibernate、quartz、EE容器和SiteMesh等其他Java技术相协调。它的一些优点包括：简单的GORM，灵活的配置文件，高级的插件系统，带有多个插件，简单的对象映射库，以及一个支持和响应社区。



任何自定义应用程序开发人员都可以在一天内快速理解这个框架。于2015年推出的Java Blade以简单和轻量级著称。这个框架最大的亮点是它能够快速创建web应用程序的能力。

它是一个全栈web开发框架，提供了一个简单而简洁的编码结构。Blade基于Java 8，它提供了RESTful风格的路由接口，同时支持webjar资源和插件扩展。



这个Java框架是由Oracle开发的，可用于创建企业应用程序、本机应用程序和web应用程序开发。它具有将表示层与应用程序代码轻松连接起来的优势。

JSF提供了一个用于表示和管理UI组件的API集。它具有清晰的体系结构，可以区分应用程序逻辑和表示形式。此外，JSF使用XML进行视图处理，而不是使用Java。



这是一个用于精简Java开发的优秀平台。你可以使用它来获得自定义的web开发服务。此框架的一大优点是能够保证服务器和浏览器之间的顺畅通信。

Vaadin提供了从Java虚拟机直接访问DOM的功能。在最新发布的版本中，它被分成了两部分。Vaadin Flow，一个允许服务器端通信和路由的轻量级框架。



因此，我总是建议大家咨询专业的Java开发人员，并与他们沟通所有的需求和目标。Java本身就是一种很有前途的编程语言。毫无疑问，选择正确的Java框架可以创造一个奇迹。

- E N D -

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)，同时欢迎关注微信公众号「Java后端」。欢迎添加小编微信「[focusioncode](#)」，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！

3分钟带你彻底搞懂 Java 泛型背后的秘密

的一幕 Java后端 2019-11-30

点击上方 Java后端, 选择 [设为星标](#)

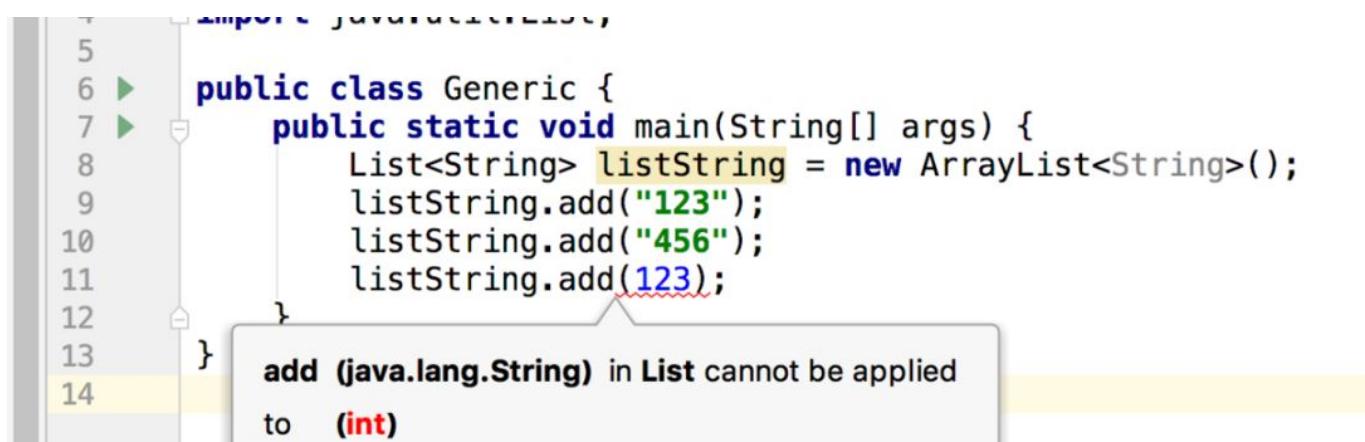
优质文章, 及时送达

作者 | 的一幕

来源 | www.jianshu.com/p/dd34211f2565

这一节主要讲的内容是java中泛型的应用，通过该篇让大家更好地理解泛型，以及面试中经常说的泛型类型擦除是什么概念，今天就带着这几个问题一起看下：

举一个简单的例子：



```
1 import java.util.ArrayList;
2
3 public class Generic {
4     public static void main(String[] args) {
5         List<String> listString = new ArrayList<String>();
6         listString.add("123");
7         listString.add("456");
8         listString.add(123);
9     }
10 }
11
12 }
```

add (java.lang.String) in List cannot be applied
to (int)

这里可以看出来在代码编写阶段就已经报错了，不能往string类型的集合中添加int类型的数据。

那可不可以往List集合中添加多个类型的数据呢，答案是可以的，其实我们可以把list集合当成普通的类也是没问题的，那么就有下面的代码：

```
List listString = new ArrayList();
listString.add("123");
listString.add("456");
listString.add(789);
for (int i = 0; i < listString.size(); i++) {
    System.out.println("listString====>" + listString.get(i));
}
```

从这里可以看出来，不定义泛型也是可以往集合中添加数据的，所以说 **泛型只是一种类型的规范，在代码编写阶段起一种限制**。

下面我们通过例子来介绍泛型背后数据是什么类型

```

public class BaseBean<T> {
    T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}

```

上面定义了一个泛型的类，然后我们通过反射获取属性和getValue方法返回的数据类型：

```

BaseBean<String> stringBaseBean = new BaseBean<>();
stringBaseBean.setValue("中国");
try {
    //获取属性上的泛型类型
    Field value = stringBaseBean.getClass().getDeclaredField("value");
    Class<?> type = value.getType();
    String name = type.getName();
    System.out.println("type:" + name);

    //获取方法上的泛型类型
    Method getValue = stringBaseBean.getClass().getDeclaredMethod("getValue");
    Object invoke = getValue.invoke(stringBaseBean);
    String methodName = invoke.getClass().getName();
    System.out.println("methodName:" + methodName);
} catch (NoSuchFieldException e) {
    e.printStackTrace();
} catch (NoSuchMethodException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
} catch (InvocationTargetException e) {
    e.printStackTrace();
}
}

```



从日志上看到通过反射获取到的属性是Object类型的，在方法中返回的是String类型，因此我们可以思考在getValue方法里面实际是做了个强转的动作，将object类型的value强转成String类型。是的，没错，因为泛型只是为了约束我们规范代码，而对于编译完之后的class交给虚拟机后，对于虚拟机它是没有泛型的说法的，所有的泛型在它看来都是object类型，因此泛型擦除是对于虚拟机而言的。

下面我们再来看一种泛型结构：

```
public class BaseBean<T extends String> {
    T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

这里我将泛型加了个关键字extends，对于泛型写得多的伙伴们来说，extends是约束了泛型是向下继承的，最后我们通过反射获取value的类型是String类型的，因此这里也不难看出，加extends关键字其实最终目的是约束泛型是属于哪一类的。所以我们在编写代码的时候如果没有向下兼容类型，会警告错误的：

```
Resource N 18
19
20 // BaseBean<Long> stringBaseBean = new BaseBean<>();
      stringBaseBean.setValue("中国");
Type parameter 'java.lang.Long' is not within its bound; should extend 'java.lang.String'
```

大家有没有想过为啥要用泛型呢，既然说了泛型其实对于jvm来说都是Object类型的，那咱们直接将类型定义成Object不就是的了，这种做法是可以，但是在拿到Object类型值之后，自己还得强转，因此泛型减少了代码的强转工作，而将这些工作交给了虚拟机。

比如下面是我们没有定义泛型的例子：

```
public class BaseBean {
    Object value;

    public Object getValue() {
        return value;
    }

    public void setValue(Object value) {
        this.value = value;
    }
}
```

势必在getValue的时候代码有个强转的过程，因此在能用泛型的时候，尽量用泛型来写，而且我认为一个好的架构师，业务的抽取是离不开泛型的定义。

作用在抽象类、接口、静态或非静态方法上。

类上面的泛型

比如实际项目中，我们经常会遇到服务端返回的接口中都有 `errMsg`、`status` 等公共返回信息，而变动的数据结构是`data`信息，因此我们可以抽取公共的 `BaseBean`：

```
public class BaseBean<T> {  
    public String errMsg;  
    public T data;  
    public int status;  
}
```

抽象类或接口上的泛型

```
//抽象类泛型  
public abstract class BaseAdapter<T> {  
    List<T> DATAS;  
}  
//接口泛型  
public interface Factory<T> {  
    T create();  
}
```

```
//方法泛型  
public static <T> T getData() {  
    return null;  
}
```

多元泛型

```
public interface Base<K, V> {  
    void setKey(K k);  
  
    V getValue();  
}
```

泛型二级抽象类或接口

```
public interface BaseCommon<K extends Common1, V> extends Base<K, V> {  
}  
//或抽象类  
public abstract class BaseCommon<K extends Common1, V> implements Base<K, V> {  
}
```

抽象里面包含抽象

```

public interface Base<K, V> {
    // void setKey(K k);
    // V getValue();
    void addNode(Map<K, V> map);

    Map<K, V> getNode(int index);
}

public abstract class BaseCommon<K, V> implements Base<K, V> {
    //多重泛型
    LinkedList<Map<K, V>> DATAS = new LinkedList<>();

    @Override
    public void addNode(Map<K, V> map) {
        DATAS.addLast(map);
    }

    @Override
    public Map<K, V> getNode(int index) {
        return DATAS.get(index);
    }
}

```

<?>通配符

<?>通配符 和 <T> 区别是<?>在你不知道泛型类型的时候，可以用<?>通配符来定义，下面通过一个例子来看看<?>的用处：

```

//定义了一个普通类
public class BaseBean<T> {
    T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}

//用来定义泛型的
public class Common1 extends Common {
}

```

```

BaseBean<Common> commonBaseBean = new BaseBean<>();
BaseBean<Common1> common1BaseBean = commonBaseBean;

```

```

BaseBean<?> comm1BaseBean = commonBaseBean;
```

Incompatible types.

Required: BaseBean <com.single.generic.Common1>

Found: BaseBean <com.single.generic.Common>

Class().getD
ects: "123");
ue();

在定义的时候将Common的泛型指向Common1的泛型，可以看到直接提示有问题，这里可以想，虽然Common1是继承自Common的，但是并不代表BaseBean之间是等量的，在开篇也讲过，如果泛型传入的是什么类型，那么在BaseBean中的getValue返回的类型就是什么，因此可以想两个不同的泛型类肯定是不等价的，但是如果我这里写呢：

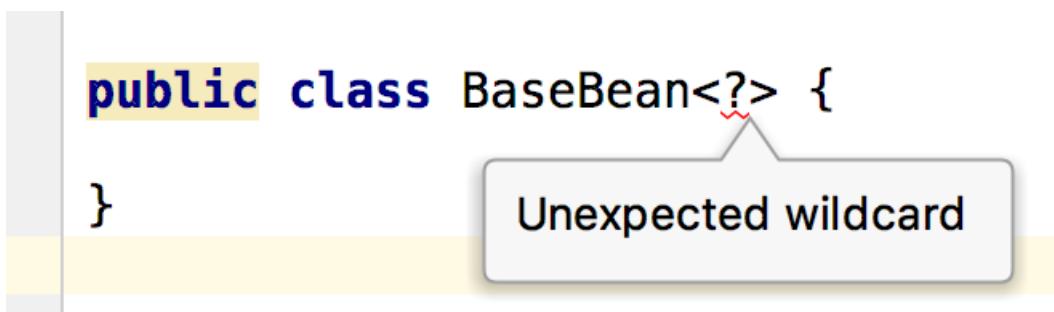
```

public static void main(String[] args) {
    BaseBean<Common> commonBaseBean = new BaseBean<>();
    //通配符定义就没有问题
    BaseBean<?> common1BaseBean = commonBaseBean;
    try {
        //通过反射猜测setValue的参数是Object类型的
        Method setValue = common1BaseBean.getClass().getDeclaredMethod("setValue", Object.class);
        setValue.invoke(common1BaseBean, "123");
        Object value = common1BaseBean.getValue();
        System.out.println("result:" + value);
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}

```

在上面如果定义的泛型是通配符是可以等价的，因为此时的setValue的参数是Object类型，所以能直接将上面定义的泛型赋给通配符的BaseBean。

通配符不能定义在类上面、接口或方法上，只能作用在方法的参数上



其他的几种情况自己去尝试，正确的使用通配符：

```

public void setClass(Class<?> class){
    //todo
}

```

<T extends >、<T super >、<? extends >、<? super >

<T extends **>表示上限泛型、<T super **>表示下限泛型

为了演示这两个通配符的作用，增加了一个类：

```

public class BaseBean<T extends Common> {
    T value;
}

```

```

//新增加的一个BaseCommon
public class Common extends BaseCommon{
}

```

```

21
22     BaseBean<Common1> common1BaseBean = new BaseBean<>();
23     BaseBean<BaseCommon> baseCommonBaseBean = new BaseBean<>();
24

```

Type parameter 'com.single.generic.BaseCommon' is not within its bound; should extend 'com.single.generic.Common'

第二个定义的泛型是不合法的，因为BaseCommon是Common的父类，超出了Common的类型范围。

<T super>不能作用在类、接口、方法上，只能通过方法传参来定义泛型

在BaseBean里面定义了个方法：

```
public void add(Class<? super Common> clazz) {  
}
```

```
BaseBean<Common1> common1BaseBean = new BaseBean<>();  
common1BaseBean.add(Common1.class);  
common1BaseBean.add(BaseCommon.class);
```

可以看到当传进去的是Common1.class的时候是不合法的，因为在add方法中需要传入Common父类的字节码对象，而Common1是继承自Common，所以直接不合法。

在实际开发中其实知道什么时候定义什么类型的泛型就ok，在mvp实际案例中泛型用得比较广泛，大家可以根据实际项目来找找泛型的感觉，只是面试的时候需要理解类型擦除是针对谁而言的。

类型擦除

其实在开篇的时候已经通过例子说明了，通过反射绕开泛型的定义，也说明了类中定义的泛型最终是以Object被jvm执行。所有的泛型在jvm中执行的时候，都是以Object对象存在的，加泛型只是为了一种代码的规范，避免了开发过程中再次强转。

泛型信息只存在于代码编译阶段，在进入 JVM 之前，与泛型相关的信息会被擦除掉，专业术语叫做类型擦除。

【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. 如果我是面试官,我会问你 Spring 那些问题?](#)

[2. Spring Boot 整合 Spring-cache](#)

[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 

[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！

40 道 Java 多线程面试题及答案

陈晨辰 Java后端 2月9日



微信搜一搜

Java后端

作者 | 陈晨辰

链接 | cnblogs.com/chen-chen-chen/p/12285283.html

这篇文章主要是对多线程的问题进行总结的，因此罗列了40个多线程的问题。

这些多线程的问题，有些来源于各大网站、有些来源于自己的思考。可能有些问题网上有、可能有些问题对应的答案也有、也可能有些各位网友也都看过，但是本文写作的重心就是所有的问题都会按照自己的理解回答一遍，不会去看网上的答案，因此可能有些问题讲的不对，能指正的希望大家不吝指教。

1、多线程有什么用？

一个可能在很多人看来很扯淡的一个问题：我会用多线程就好了，还管它有什么用？在我看来，这个回答更扯淡。所谓“知其然知其所以然”，“会用”只是“知其然”，“为什么用”才是“知其所以然”，只有达到“知其然知其所以然”的程度才可以说是把一个知识点运用自如。OK，下面说说我对这个问题的看法：

1) 发挥多核CPU的优势

随着工业的进步，现在的笔记本、台式机乃至商用的应用服务器至少也都是双核的，4核、8核甚至16核的也都不少见，如果是单线程的程序，那么在双核CPU上就浪费了50%，在4核CPU上就浪费了75%。**单核CPU上所谓的“多线程”那是假的多线程，同一时间处理器只会处理一段逻辑，只不过线程之间切换得比较快，看着像多个线程“同时”运行罢了。**多核CPU上的多线程才是真正意义上的多线程，它能让你的多段逻辑同时工作，多线程，可以真正发挥出多核CPU的优势来，达到充分利用CPU的目的。

2) 防止阻塞

从程序运行效率的角度来看，单核CPU不但不会发挥出多线程的优势，反而会因为在单核CPU上运行多线程导致线程上下文的切换，而降低程序整体的效率。但是单核CPU我们还是要应用多线程，就是为了防止阻塞。试想，如果单核CPU使用单线程，那么只要这个线程阻塞了，比方说远程读取某个数据吧，对端迟迟未返回又没有设置超时时间，那么你的整个程序在数据返回回来之前就停止运行了。多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞，也不会影响其它任务的执行。

3) 便于建模

这是另外一个没有这么明显的优点了。假设有一个大的任务A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务A分解成几个小任务，任务B、任务C、任务D，分别建立程序模型，并通过多线程分别运行这几个任务，那就简单很多了。

2、创建线程的方式

比较常见的一个问题了，一般就是两种：

1) 继承Thread类

2) 实现Runnable接口

至于哪个好，不用说肯定是后者好，因为实现接口的方式比继承类的方式更灵活，也能减少程序之间的耦合度，**面向接口编程**也是设计模式6大原则的核心。

3、start()方法和run()方法的区别

只有调用了start()方法，才会表现出多线程的特性，不同线程的run()方法里面的代码交替执行。如果只是调用run()方法，那么代码还是同步执行的，必须等待一个线程的run()方法里面的代码全部执行完毕之后，另外一个线程才可以执行其run()方法里面的代码。

4、Runnable接口和Callable接口的区别

有点深的问题了，也看出一个Java程序员学习知识的广度。

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已； Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

这其实是很有一个特性，因为**多线程相比单线程更难、更复杂的一个重要原因就是因为多线程充满着未知性**，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完毕而已。而Callable+Future/FutureTask却可以获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务，真的是非常有用。

5、CyclicBarrier和CountDownLatch的区别

两个看上去有点像的类，都在java.util.concurrent下，都可以用来表示代码运行到某个点上，二者的区别在于：

- 1) CyclicBarrier的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行； CountDownLatch则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行。
- 2) CyclicBarrier只能唤起一个任务， CountDownLatch可以唤起多个任务。
- 3) CyclicBarrier可重用， CountDownLatch不可重用，计数值为0该CountDownLatch就不可再用了。

6、volatile关键字的作用

一个非常重要的问题，是每个学习、应用多线程的Java程序员都必须掌握的。理解volatile关键字的作用的前提是要理解Java内存模型，这里就不讲Java内存模型了，可以参见第31点， volatile关键字的作用主要有两个：

- 1) 多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据。
- 2) 代码底层执行不像我们看到的高级语言----Java程序这么简单，它的执行是**Java代码-->字节码-->根据字节码执行对应的C/C++代码-->C/C++代码被编译成汇编语言-->和硬件电路交互**，现实中，为了获取更好的性能JVM可能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用volatile则会对禁止语义重排序，当然这也一定程度上降低了代码执行效率。

从实践角度而言，volatile的一个重要作用就是和CAS结合，保证了原子性，详细的可以参见java.util.concurrent.atomic包下的类，比如AtomicInteger.

7、什么是线程安全

又是一个理论的问题，各式各样的答案有很多，我给出一个个人认为解释最好的：**如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。**

这个问题有值得一提的地方，就是线程安全也是有几个级别的：

1) 不可变

像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用

2) 绝对线程安全

不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet

3) 相对线程安全

相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现 ConcurrentModificationException，也就是**fail-fast机制**。

4) 线程非安全

这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类

8、Java中如何获取到线程dump文件

死循环、死锁、阻塞、页面打开慢等问题，打线程dump是最好的解决问题的途径。所谓线程dump也就是线程堆栈，获取到线程堆栈有两步：

- 1) 获取到线程的pid，可以通过使用jps命令，在Linux环境下还可以使用ps -ef | grep java
- 2) 打印线程堆栈，可以通过使用jstack pid命令，在Linux环境下还可以使用kill -3 pid

另外提一点，Thread类提供了一个getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取获取到的是具体某个线程当前运行的堆栈。

9、一个线程如果出现了运行时异常会怎么样

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：**如果这个线程持有某个对象的监视器，那么这个对象监视器会被立即释放**

10、如何在两个线程之间共享数据

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的

11、sleep方法和wait方法有什么区别

这个问题常问，sleep方法和wait方法都可以用来放弃CPU一定的时间，不同点在于如果线程持有某个对象的监视器，sleep方法不会放弃这个对象的监视器，wait方法会放弃这个对象的监视器

12、生产者消费者模型的作用是什么

这个问题很理论，但是很重要：

- 1) 通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用
- 2) 解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约

13、ThreadLocal有什么用

简单说ThreadLocal就是一种以空间换时间的做法，在每个Thread里面维护了一个以开地址法实现的ThreadLocal.ThreadLocalMap，把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了

14、为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用

这是JDK强制的，wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁

15、wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于：**wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。**

16、为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。[点击这里学习线程池详解。](#)

17、怎么检测一个线程是否持有对象监视器

我也是在网上看到一道多线程面试题才知道有方法可以判断某个线程是否持有对象监视器：Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着“某条线程”指的是当前线程。

18、synchronized和ReentrantLock的区别

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：

- (1) ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁
- (2) ReentrantLock可以获取各种锁的信息
- (3) ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的。ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对对象头中mark word，这点我不能确定。

19、ConcurrentHashMap的并发度是什么

ConcurrentHashMap的并发度就是segment的大小，默认为16，这意味着最多同时可以有16条线程操作 ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势，任何情况下，Hashtable能同时有两条线程获取 Hashtable中的数据吗？

20、ReadWriteLock是什么

首先明确一下，不是说ReentrantLock不好，只是ReentrantLock某些时候有局限。如果使用ReentrantLock，可能本身是为了防止线程A在写数据、线程B在读数据造成的数据不一致，但这样，如果线程C在读数据、线程D也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。

因为这个，才诞生了读写锁ReadWriteLock。ReadWriteLock是一个读写锁接口，ReentrantReadWriteLock是ReadWriteLock接口的一个具体实现，实现了读写的分离，**读锁是共享的，写锁是独占的**，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

21、FutureTask是什么

这个其实前面有提到过，FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

22、Linux环境下如何查找哪个线程使用CPU最长

这是一个比较偏实践的问题，这种问题我觉得挺有意义的。可以这么做：

- (1) 获取项目的pid，jps或者ps -ef | grep java，这个前面有讲过
- (2) top -H -p pid，顺序不能改变

这样就可以打印出当前的项目，每条线程占用CPU时间的百分比。注意这里打出的是LWP，也就是操作系统原生线程的线程号，我笔记本没有部署Linux环境下的Java工程，因此没有办法截图演示，网友朋友们如果公司是使用Linux环境部署项目的话，可以尝试一下。

使用"top -H -p pid"+"jps pid"可以很容易地找到某条占用CPU高的线程的线程堆栈，从而定位占用CPU高的原因，一般是因为不当的代码操作导致了死循环。

最后提一点，"top -H -p pid"打出来的LWP是十进制的，"jps pid"打出来的本地线程号是十六进制的，转换一下，就能定位到占用CPU高的线程的当前线程堆栈了。

23、Java编程写一个会导致死锁的程序

第一次看到这个题目，觉得这是一个非常好的问题。很多人都知道死锁是怎么一回事儿：线程A和线程B相互等待对方持有的锁导致程序无限死循环下去。当然也仅限于此了，问一下怎么写一个死锁的程序就不知道了，这种情况说白了就是不懂什么是死锁，懂一个理论就完事儿了，实践中碰到死锁的问题基本上是看不出来的。

真正理解什么是死锁，这个问题其实不难，几个步骤：

- 1) 两个线程里面分别持有两个Object对象：lock1和lock2。这两个lock作为同步代码块的锁；

2) 线程1的run()方法中同步代码块先获取lock1的对象锁，Thread.sleep(xxx)，时间不需要太多，50毫秒差不多了，然后接着获取lock2的对象锁。这么做主要是为了防止线程1启动一下子就连续获得了lock1和lock2两个对象的对象锁

3) 线程2的run()方法中同步代码块先获取lock2的对象锁，接着获取lock1的对象锁，当然这时lock1的对象锁已经被线程1锁持有，线程2肯定是要等待线程1释放lock1的对象锁的

这样，线程1“睡觉”睡完，线程2已经获取了lock2的对象锁了，线程1此时尝试获取lock2的对象锁，便被阻塞，此时一个死锁就形成了。代码就不写了，占的篇幅有点多，Java多线程7：死锁这篇文章里面有，就是上面步骤的代码实现。

24、怎么唤醒一个阻塞的线程

如果线程是因为调用了wait()、sleep()或者join()方法而导致的阻塞，可以中断线程，并且通过抛出InterruptedException来唤醒它；如果线程遇到了IO阻塞，无能为力，因为IO是操作系统实现的，Java代码并没有办法直接接触到操作系统。

25、不可变对象对多线程有什么帮助

前面有提到过的一个问题，不可变对象保证了对象的内存可见性，对不可变对象的读取不需要进行额外的同步手段，提升了代码执行效率。

26、什么是多线程的上下文切换

多线程的上下文切换是指CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程。

27、如果你提交任务时，线程池队列已满，这时会发生什么

这里区分一下：

1) 如果使用的是无界队列LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列，可以无限存放任务

2) 如果使用的是有界队列比如ArrayBlockingQueue，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，会根据maximumPoolSize的值增加线程数量，如果增加了线程数量还是处理不过来，ArrayBlockingQueue继续满，那么则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy

28、Java中用到的线程调度算法是什么

抢占式。一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

29、Thread.sleep(0)的作用是什么

这个问题和上面那个问题是相关的，我就连在一起了。由于Java采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到CPU控制权的情况，为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

30、什么是自旋

很多synchronized里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然synchronized里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在synchronized的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

31、什么是Java内存模型

Java内存模型定义了一种多线程访问Java内存的规范。Java内存模型要完整讲不是这里几句话能说清楚的，我简单总结一下Java内存模型的几部分内容：

- 1) Java内存模型将内存分为了**主内存和工作内存**。类的状态，也就是类之间共享的变量，是存储在主内存中的，每次Java线程用到这些主内存中的变量的时候，会读一次主内存中的变量，并让这些内存在自己的工作内存中有一份拷贝，运行自己线程代码的时候，用到这些变量，操作的都是自己工作内存中的那一份。在线程代码执行完毕之后，会将最新的值更新到主内存中去
- 2) 定义了几个原子操作，用于操作主内存和工作内存中的变量
- 3) 定义了volatile变量的使用规则
- 4) happens-before，即先行发生原则，定义了操作A必然先行发生于操作B的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁unlock的动作一定先行发生于后面对于同一个锁进行锁定lock的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的happens-before规则，则这段代码一定是线程非安全的

32、什么是CAS

CAS，全称为Compare and Swap，即比较-替换。假设有三个操作数：**内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，才会将内存值修改为B并返回true，否则什么都不做并返回false**。当然CAS一定要volatile变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值A对某条线程来说，永远是一个不会变的值A，只要某次CAS操作失败，永远都不可能成功。

33、什么是乐观锁和悲观锁

- 1) 乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将**比较-替换**这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。
- 2) 悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像synchronized，不管三七二十一，直接上了锁就操作资源了。

34、什么是AQS

简单说一下AQS，AQS全称为AbstractQueuedSychronizer，翻译过来应该是抽象队列同步器。

如果说java.util.concurrent的基础是CAS的话，那么AQS就是整个Java并发包的核心了，ReentrantLock、CountDownLatch、Semaphore等等都用到了它。AQS实际上以双向队列的形式连接所有的Entry，比方说ReentrantLock，所有等待的线程都被放在一个Entry中并连成双向队列，前面一个线程使用ReentrantLock好了，则双向队列实际上的第一个Entry开始运行。

AQS定义了对双向队列所有的操作，而只开放了tryLock和tryRelease方法给开发者使用，开发者可以根据自己的实现重写

tryLock和tryRelease方法，以实现自己的并发功能。

35、单例模式的线程安全性

老生常谈的问题了，首先要说的是单例模式的线程安全意味着：**某个类的实例在多线程环境下只会被创建一次出来**。单例模式有很多种的写法，我总结一下：

- 1) 饿汉式单例模式的写法：线程安全
- 2) 懒汉式单例模式的写法：非线程安全
- 3) 双检锁单例模式的写法：线程安全

36、Semaphore有什么作用

Semaphore就是一个信号量，它的作用是**限制某段代码块的并发数**。Semaphore有一个构造函数，可以传入一个int型整数n，表示某段代码最多只有n个线程可以访问，如果超出了n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果Semaphore构造函数中传入的int型整数n=1，相当于变成了一个synchronized了。

37、Hashtable的size()方法中明明只有一条语句"return count"，为什么还要做同步？

这是我之前的一个困惑，不知道大家有没有想过这个问题。某个方法中如果有多条语句，并且都在操作同一个类变量，那么在多线程环境下不加锁，势必会引发线程安全问题，这很好理解，但是size()方法明明只有一条语句，为什么还要加锁？

关于这个问题，在慢慢地工作、学习中，有了理解，主要原因有两点：

- 1) **同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问**。所以，这样就有问题了，可能线程A在执行Hashtable的put方法添加数据，线程B则可以正常调用size()方法读取Hashtable中当前元素的个数，那读取到的值可能不是最新的，可能线程A添加了完了数据，但是没有对size++，线程B就已经读取size了，那么对于线程B来说读取到的size一定是不准确的。而给size()方法加了同步之后，意味着线程B调用size()方法只有在线程A调用put方法完毕之后才可以调用，这样就保证了线程安全性
- 2) **CPU执行代码，执行的不是Java代码，这点很关键，一定得记住**。Java代码最终是被翻译成机器码执行的，机器码才是真正可以和硬件电路交互的代码。**即使你看到Java代码只有一行，甚至你看到Java代码编译之后生成的字节码也只有一行，也不意味着对于底层来说这句语句的操作只有一个**。一句"return count"假设被翻译成了三句汇编语句执行，一句汇编语句和其机器码做对应，完全可能执行完第一句，线程就切换了。

38、线程类的构造方法、静态块是被哪个线程调用的

这是一个非常刁钻和狡猾的问题。请记住：线程类的构造方法、静态块是被new这个线程类所在的线程所调用的，而run方法里面的代码才是被线程自身所调用的。

如果说上面的说法让你感到困惑，那么我举个例子，假设Thread2中new了Thread1，main函数中new了Thread2，那么：

- 1) Thread2的构造方法、静态块是main线程调用的，Thread2的run()方法是Thread2自己调用的
- 2) Thread1的构造方法、静态块是Thread2调用的，Thread1的run()方法是Thread1自己调用的

39、同步方法和同步块，哪个是更好的选择

同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。请知道一条原则：**同步的范围越小越好。**

借着这一条，我额外提一点，虽说同步的范围越少越好，但是在Java虚拟机中还是存在着一种叫做**锁粗化**的优化方法，这种方法就是把同步范围变大。这是有用的，比方说StringBuffer，它是一个线程安全的类，自然最常用的append()方法是一个同步方法，我们写代码的时候会反复append字符串，这意味着要进行反复的加锁->解锁，这对性能不利，因为这意味着Java虚拟机在这些线程上要反复地在内核态和用户态之间进行切换，因此Java虚拟机会将多次append方法调用的代码进行一个锁粗化的操作，将多次的append的操作扩展到append方法的头尾，变成一个大的同步块，这样就减少了加锁-->解锁的次数，有效地提升了代码执行的效率。

40、高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

这是我在并发编程网上看到的一个问题，把这个问题放在最后一个，希望每个人都能看到并且思考一下，因为这个问题非常好、非常实际、非常专业。关于这个问题，个人看法是：

- 1) 高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换
- 2) 并发不高、任务执行时间长的业务要区分开看：
 - a) 假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务
 - b) 假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换
 - c) 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考其他有关线程池的文章。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

推荐阅读

1. Java：如何更优雅的处理空值？
2. 【免费】某平台 16980 元编程课程资料下载，仅此一次
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

Java后端

5分钟讲明白 JVM、Java、Java对象模型

Java后端 2019-10-13

以下文章来源于Hollis，作者Hollis



Hollis

Hollis，一个对Coding有着独特追求的人。现任阿里巴巴技术专家，《程序员的三门课》合作者。本公众号专注分享Ja…

Java作为一种面向对象的，跨平台语言，其对象、内存等一直是比较难的知识点。而且很多概念的名称看起来又那么相似，很多人会傻傻分不清楚。比如本文我们要讨论的**JVM内存结构**、**Java内存模型**和**Java对象模型**，这就是三个截然不同的概念，但是很多人容易弄混。

可以这样说，很多高级开发甚至都搞不清楚JVM内存结构、Java内存模型和Java对象模型这三者的概念及其间的区别。甚至我见过有些面试官自己也搞的不太清楚。不信的话，你去网上搜索Java内存模型，还会有很多文章的内容其实介绍的是JVM内存结构。

首先，这三个概念是完全不同的三个概念。**本文主要对这三个概念加以区分以及简单介绍。其中每一个知识点都可以单独写一篇文章，本文并不会深入介绍，感兴趣的朋友可以加入我的知识星球和球友们共同学习。**



JVM内存结构

我们都知道，Java代码是要运行在虚拟机上的，而虚拟机在执行Java程序的过程中会把所管理的内存划分为若干个不同的数据区域，这些区域都有各自的用途。

其中有些区域随着虚拟机进程的启动而存在，而有些区域则依赖用户线程的启动和结束而建立和销毁。在《Java虚拟机规范（Java SE 8）》中描述了JVM运行时内存区域结构如下：



各个区域的功能不是本文重点，就不在这里详细介绍了。这里简单提几个需要特别注意的点：

- 以上是Java虚拟机规范，不同的虚拟机实现会各有不同，但是一般会遵守规范。
- 规范中定义的方法区，只是一种概念上的区域，并说明了其应该具有什么功能。但是并没有规定这个区域到底应该处于何处。所以，对于不同的虚拟机实现来说，是有一定的自由度的。
- 不同版本的方法区所处位置不同，上图中划分的是逻辑区域，并不是绝对意义上的物理区域。因为某些版本的JDK中方法区其实是在堆中实现的。
- 运行时常量池用于存放编译期生成的各种字面量和符号应用。但是，Java语言并不要求常量只有在编译期才能产生。比如在运行期，String.intern也会把新的常量放入池中。

5、除了以上介绍的JVM运行时内存外，还有一块内存区域可供使用，那就是直接内存。Java虚拟机规范并没有定义这块内存区域，所以他并不由JVM管理，是利用本地方法库直接在堆外申请的内存区域。

6、堆和栈的数据划分也不是绝对的，如HotSpot的JIT会针对对象分配做相应的优化。

如上，做个总结，JVM内存结构，由Java虚拟机规范定义。描述的是Java程序执行过程中，由JVM管理的不同数据区域。各个区域有其特定的功能。

Java内存模型

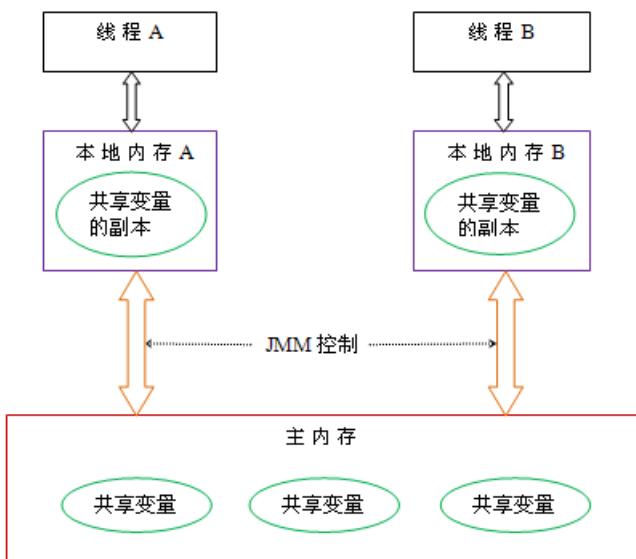
Java内存模型看上去和Java内存结构（JVM内存结构）差不多，很多人会误以为两者是一回事儿，这也就导致面试过程中经常答非所为。

在前面的关于JVM的内存结构的图中，我们可以看到，其中Java堆和方法区的区域是多个线程共享的数据区域。也就是说，多个线程可能可以操作保存在堆或者方法区中的同一个数据。这也就是我们常说的“Java的线程间通过共享内存进行通信”。

Java内存模型是根据英文Java Memory Model (JMM) 翻译过来的。其实JMM并不像JVM内存结构一样是真实存在的。他只是一个抽象的概念。JSR-133: Java Memory Model and Thread Specification 中描述了，JMM是和多线程相关的，他描述了一组规则或规范，这个规范定义了一个线程对共享变量的写入时对另一个线程是可见的。

那么，简单总结下，Java的多线程之间是通过共享内存进行通信的，而由于采用共享内存进行通信，在通信过程中会存在一系列如可见性、原子性、顺序性等问题，而JMM就是围绕着多线程通信以及与其相关的一系列特性而建立的模型。JMM定义了一些语法规集，这些语法规集映射到Java语言中就是volatile、synchronized等关键字。

在JMM中，我们把多个线程间通信的共享内存称之为内存，而在并发编程中多个线程都维护了一个自己的本地内存（这是个抽象概念），其中保存的数据是内存中的数据拷贝。而JMM主要是控制本地内存和内存之间的数据交互的。



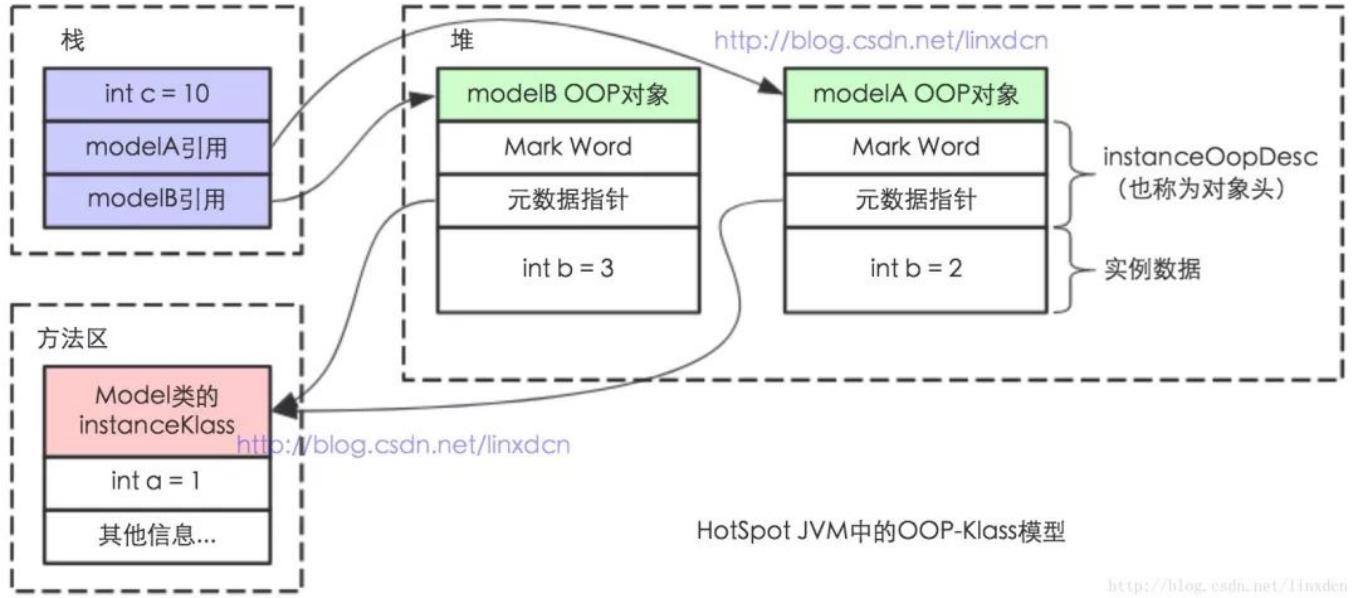
在Java中，JMM是一个非常重要的概念，正是由于有了JMM，Java的并发编程才能避免很多问题。这里就不对Java内存模型做更加详细的介绍了，想了解更多的朋友可以参考《Java并发编程的艺术》。

Java对象模型

Java是一种面向对象的语言，而Java对象在JVM中的存储也是有一定的结构的。而这个关于Java对象自身的存储模型称之为Java对象模型。

HotSpot虚拟机中，设计了一个OOP-Klass Model。OOP (Ordinary Object Pointer) 指的是普通对象指针，而Klass用来描述对象实例的具体类型。

每一个Java类，在被JVM加载的时候，JVM会给这个类创建一个 **instanceKlass**，保存在方法区，用来在JVM层表示该Java类。当我们在Java代码中，使用new创建一个对象的时候，JVM会创建一个 **instanceOopDesc** 对象，这个对象中包含了对象头以及实例数据。



这就是一个简单的Java对象的OOP-Klass模型，即Java对象模型。

总结

我们再来区分下JVM内存结构、 Java内存模型 以及 Java对象模型 三个概念。

JVM内存结构，和Java虚拟机的运行时区域有关。

Java内存模型，和Java的并发编程有关。

Java对象模型，和Java对象在虚拟机中的表现形式有关。

关于这三部分内容，本文并未分别展开，因为涉及到的知识点实在太多，如果读者感兴趣，可以自行学习。后面也会发文介绍这些内容，敬请期待。

最后，这三个概念非常重要，一定要严格区分开，千万不要在面试中出现答非所为的情况。

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！

6年 Java 老兵 BAT 面试心经

Java后端 2019-09-24

点击上方 Java后端, 选择“**设为星标**”

优质文章, 及时送达

推荐阅读: IntelliJ IDEA 2019.3 这回真的要飞起来了

我觉得有一个能够找一份大厂的 offer 的想法, 这是很正常的, 这并不是我们的饭后谈资而是每个技术人的追求。像阿里、腾讯、美团、字节跳动、京东等等的技术氛围与技术规范度还是要明显优于一些创业型公司 / 小公司, 如果说能够在这样的公司锻炼几年, 相信对自己能力的提升还是非常大的。不论是校招还是社招都避免不了各种面试、笔试, 如何去准备这些东西就显得格外重要。不论是笔试还是面试都是有章可循的。

因为大厂面试一般都有专业团队负责, 某个知识点你到底是掌握了还是单纯背下来, 面试官一问就可以看出来 (PS: 真正到面试特别是你觉得准备面试的时间不够的时候, 你可以多挑一些面试常问的问题来看, 注意理解, 一定不要死记硬背)。可以试着参考各种面经, 知道面试的大体思路, 然后去提高自己的综合能力。

“80% 的 offer 掌握在 20% 的人手” 中这句话也不是不无道理的。决定你面试能否成功的因素中实力固然占有很大一部分比例, 但是如果你的心态或者说运气不好的话, 依然无法拿到满意的 offer。运气暂且不谈, 就拿心态来说, 千万不要因为面试失败而气馁或者说怀疑自己的能力, 面试失败之后多总结一下失败的原因, 后面你就会发现自己会越来越强大。

从大厂实际招聘要求来看到底青睐什么样的人?

首先要明确的一点是: 985/211 的学历的确会为你加分很多。

另外, 再强调的一点是不要天天把自己的学校是双非学校这个借口当做你无法进入大厂的原因。只要你的能力足够, 大厂的大门就会为你打开。也有着很多双非学校甚至是三本的同学就拿到像阿里、腾讯这样的大公司的 offer。[微信搜索 web_resource 关注后获取更多优质文章](#)

从阿里、腾讯等大厂招聘官网对于 Java 后端方向/后端方向的要求, 我们大概可以总结看出大厂对招聘者的能力要求。

下面以阿里巴巴为例子, 看看大厂的实际要求

发布时间:	2019-02-15	工作地点:	杭州	工作年限:	二年以上
所属部门:	蚂蚁金服	学 历:	本科	招聘人数:	5

岗位描述:

在这里, 你就是支付宝最核心的系统的构建者。你将直面“双11”的巅峰挑战, 解决世界级的分布式处理难题。你将参与各类业务及技术改造的系统分析与设计工作, 承担核心功能代码编写, 维护公用核心模块。你将负责核心系统的性能和架构优化, 持续提升系统在海量并发请求下的处理能力, 保障系统稳定性, 提升容灾能力。你需要完成相关技术攻关, 识别和解决潜在的技术风险。而与你并肩的, 是蚂蚁金服经验丰富的工程师。这里学习, 成长, 上升潜力巨大。

岗位要求:

- 扎实的Java编程基础, 对常见开发框架, 如Spring, MyBatis等有深入了解, 有框架开发经验或相关开源项目贡献者优先。
- 有丰富的大型数据库系统应用经验, 熟悉分库分表, 数据库访问优化等技术, 了解数据库系统原理者优先。
- 精通微服务架构, 分布式系统原理, 有分布式系统研发经验者优先。
- 有良好的表达和沟通能力, 善于学习, 关注前沿, 能利用创新手段解决问题。
- 工作认真, 严谨, 敬业, 对系统质量有极致的追求。
- 有银行核心系统, 互联网金融系统研发背景者优先。

在面试 Java 工程师的时候，下面几点也提升你的个人竞争力：

1. 熟悉开源框架的底层，阅读源码；
2. 大型数据库系统经验；
3. 熟悉分布式，缓存，消息中间件；
4. 良好的表达和沟通能力，善于学习，关注前沿。

“一定要有一门自己的特长，不管是技术还好还是其他能力”。我觉得这句话真的非常有道理，大家可以仔细思考一下。在这里再强调一点：公司不需要你什么都会，但是在某一方面你一定要有过于常人的优点。换言之就是我们不需要去掌握每一门技术（你也没精力去掌握这么多技术），而是需要去深入研究某一门技术，对于其他技术我们可以简单了解一下。

我觉得比起你对每一门技术都是浅尝辄止，深入吃透某一门技术对你的个人竞争力提升才更有帮助。

如何获取大厂面试机会？

在讲如何获取大厂面试机会之前，先来对比一下两个非常常见的概念——春招和秋招。

招聘人数：秋招多于春招；

招聘时间：秋招一般 7 月左右开始，大概一直持续到 10 月底。但是大厂（如 BAT）都会早开始早结束，所以一定要把握好时间。春招最佳时间为 3 月，次佳时间为 4 月，进入 5 月基本就不会再有春招了（金三银四）。

应聘难度：秋招略大于春招；

招聘公司：秋招数量多，而春招数量较少，一般为秋招的补充。

综上，一般来说，秋招的含金量明显是高于春招的。

下面我就说一下我自己知道的一些方法，不过应该也涵盖了大部分获取面试机会的方法。

关注大厂官网，随时投递简历（走流程的网申）；

找到师兄师姐或者 **认识的前公司的技术牛人**，帮忙内推（能够让你避开网申简历筛选，笔试筛选，还是挺不错的，不过也还是需要你的简历够棒）；

求职类网站投递简历（不太推荐）。

除了这些方法，我也遇到过这样的经历：有些大公司的一些部门可能暂时没招够人，然后如果你的亲戚或者朋友刚好在这个公司，而你正好又在寻求 offer，那么面试机会基本上是有了，而且这种面试的难度好像一般还普遍比其他正规面试低很多。

想要取得一份自己满意的 offer，前提是自己要有过硬的实力作为资本，下面就如何提高个人硬实力给大家提几点建议！

微信搜索 *web_resource* 关注后获取更多优质文章

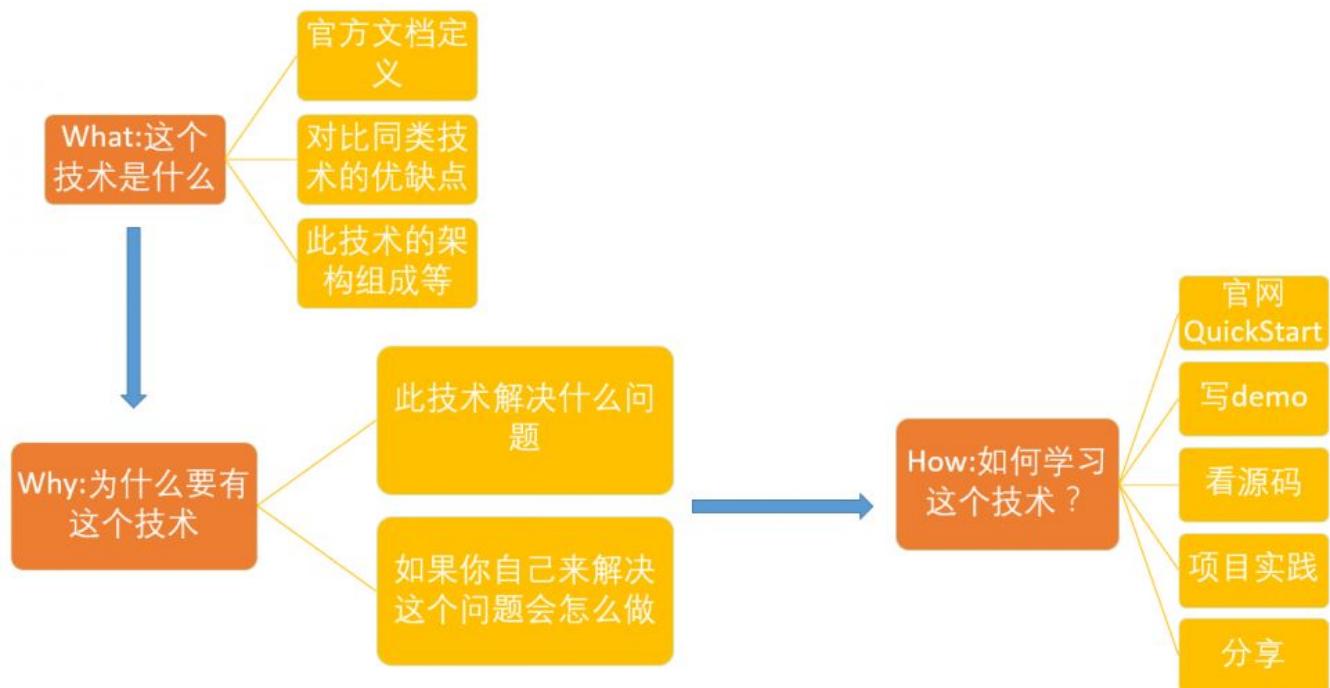
如何提高个人硬实力及大厂 Java 后端面试主要问些什么？

我在这里所说的个人硬实力更多的指的是个人的专业能力，比如构建高质量网站的能力或者是对专业知识的掌握程度。

我觉得不论是对于新手还是老手，想要提高个人硬实力最重要的就是不断深入学习并且将理论实践，最好可以将理论在具体项目中实践一下。

想要提高个人硬实力，那么学习一门新技术的方法一定是至关重要了。下面分享一下对于学习一门新技术的一些要点（在这以图片的方

式整理了出来，更加方便阅读)：



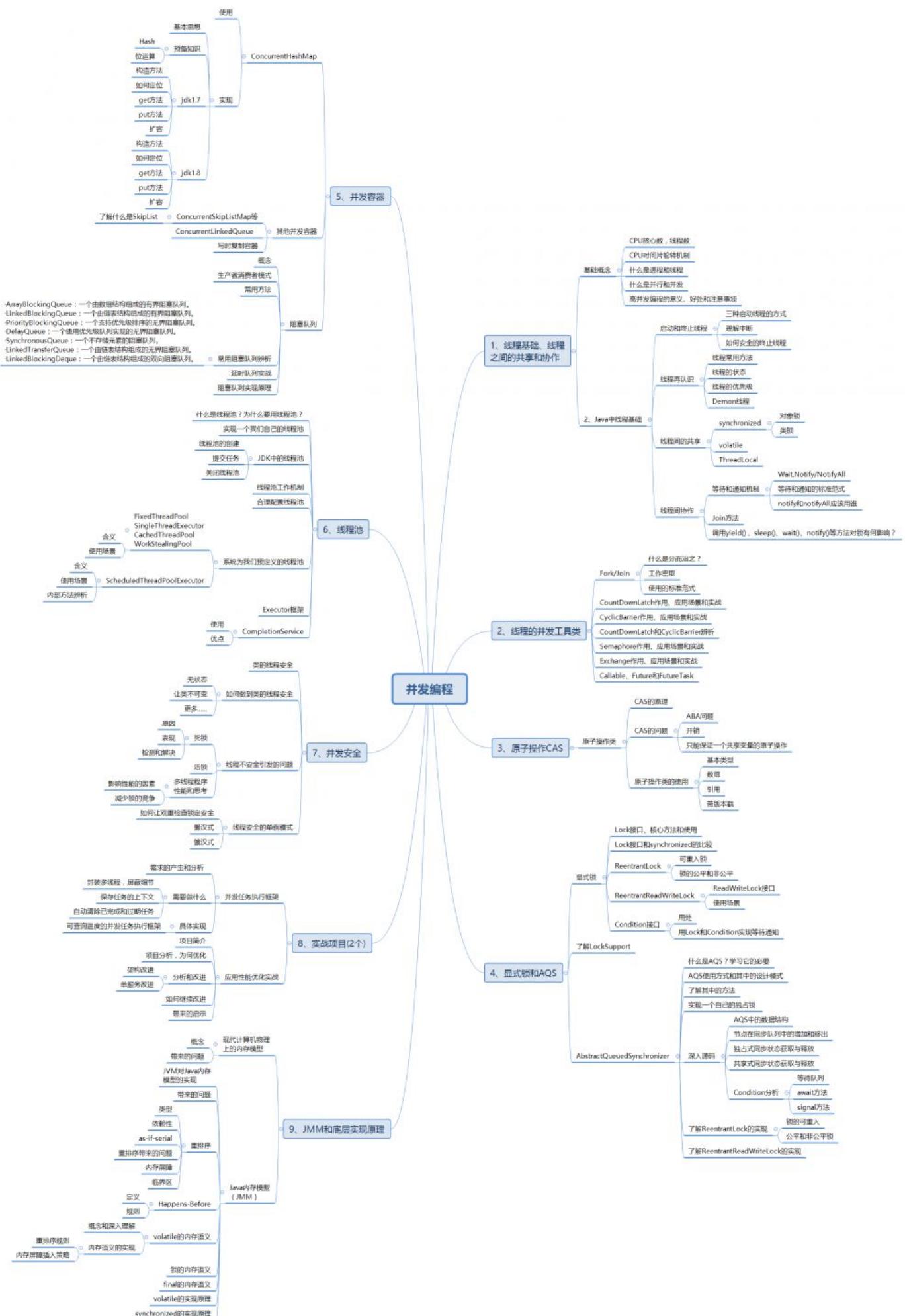
一定要有自己的技术优势，可能你懂得不是最多的，但是别人不会的你却会，那么你就是厉害的！然而如何准备大厂面试？我觉得最关键的一点之一就是搞清楚大厂面试主要在问些什么。下面我将分解每一个知识点，给大家简单说一下大厂面试主要会问些什么？

首先你要明确的是面试官所问的内容一定和你简历所写的东西是紧密联系的，一般你没有记录简历上的技能，面试官很少会去提问。

大厂面试大体上包括下面几方面知识类型：

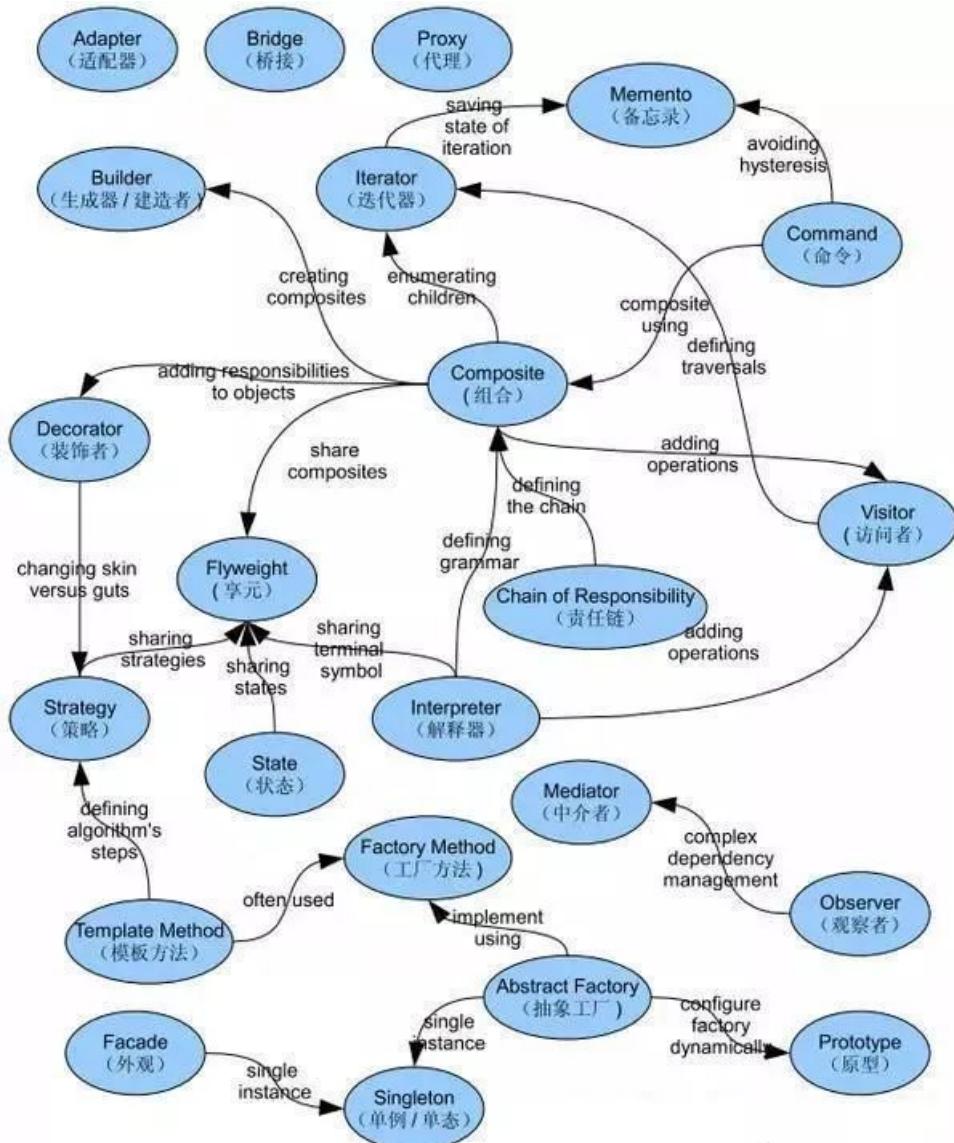
Java 基础、多线程、IO 与 NIO、虚拟机、设计模式

面试官在多线程这一部分很可能会问你有没有在项目中实际使用多线程的经历。所以，如果你在你的项目中有实际使用 Java 多线程的经历的话，会为你加分不少哦！



设计模式比较常见的就是让你手写一个单例模式(注意单例模式的几种不同的实现方法)或者让你说一下某个常见的设计模式在你的项目中是如何使用的,另外面试官还有可能问你“抽象工厂”和“工厂方法模式的区别”、“工厂模式”的思想这样的问题”。

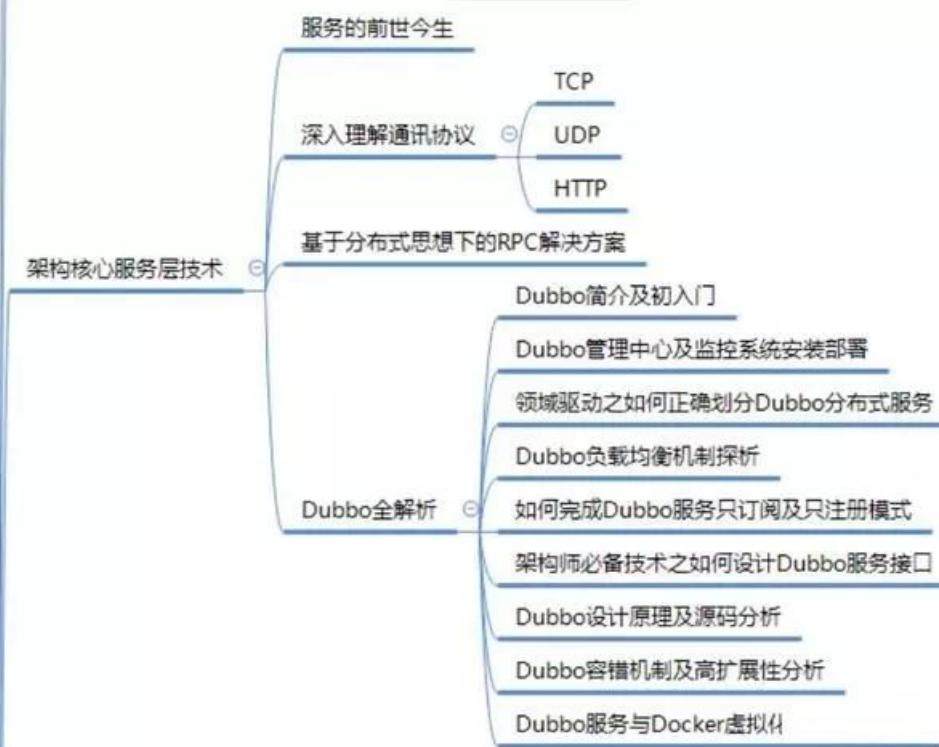
建议把 代理模式、观察者模式、(抽象)工厂模式 好好看一下，这三个设计模式很有用。



数据结构与算法(要有手写算法的能力)

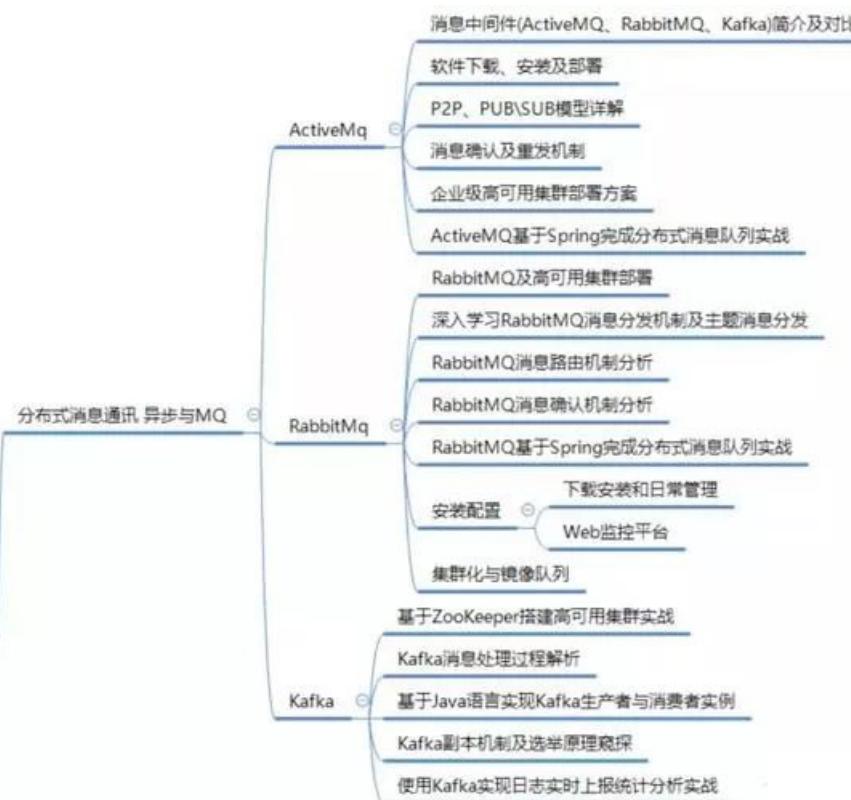
数据结构比较常问的就是：二叉树、红黑树（很可能让你手绘一个红黑树出来哦！）、二叉查找树（BST）、平衡二叉树（Self-balancing binary search tree）、B – 树，B + 树与 B * 树的优缺点比较、LSM 树这些知识点。数据结构很重要，而且学起来也相对要难一些。建议学习数据结构一定要循序渐进的来，一步一个脚印的走好。一定要搞懂原理，最好自己能用代码实现一遍。

计算机网络(TCP 三次握手和四次挥手)



数据通信(RESTful、RPC、消息队列)

如果你的简历上写了你会某个RPC框架（比如：阿里的开源的dubbo）或者消息队列（比如：RabbitMQ、Kafka）的使用的话，面试官一般会以你写在简历上的技术提问，回答的时候最好能结合在项目中的实际使用。



性能优化及操作系统(常见优化方式, Linux 的基本命令以及使用)

性能调优

深入内核，直击故障，拒绝蒙圈

性能优化如何理解

性能基准
什么是性能优化
衡量标准

JVM调优

jvm虚拟机内存剖析
垃圾收集器
实战调优案例与解决方法
JVM运行时区

Java程序性能优化

优雅的创建对象
注意对象的通用方法
类的设计陷阱
泛型需要注意的问题
java方法的那些坑
程序设计的通用规则

Tomcat

线程模型分析
生产环境配置及调优
运行机制及框架

Mysql

探析BTree机制
执行计划深入分析
Mysql索引优化详解
慢查询分析与SQL优化

主流框架(Spring 底层原理与源码问的很多)

Spring一般是不可避免的，如果你的简历上注明了你会 Spring Boot 或者 Spring Cloud 的话，那么面试官也可能会同时问你这两个技术，比如他可能会问你 springboot 和 spring 的区别。微信搜索 web_resource 关注后获取更多优质文章。所以，一定要谨慎对待写在简历上的东西，一定要对简历上的东西非常熟悉。

微信搜索 `web_resource` 关注后获取更多优质文章

另外，AOP 实现原理、动态代理和静态代理、Spring IOC 的初始化过程、IOC 原理、自己怎么实现一个 IOC 容器？这些东西都是经常会被问到的。

应用框架源码解读

站在巨人肩膀，收获不一样的视野

Spring IOC

- Spring Framework体系结构
- 源码分析
- BeanFactory源码分析
- BeanDefinition源码分析
- Bean生命同期
- 依赖实现

Spring Aop

- AOP源码分析
- transaction事务分析
- spring cache框架源码分析

Spring MVC

- MVC简介与设计思想
- SpringMVC组成
- 源码解读DispatchServlet

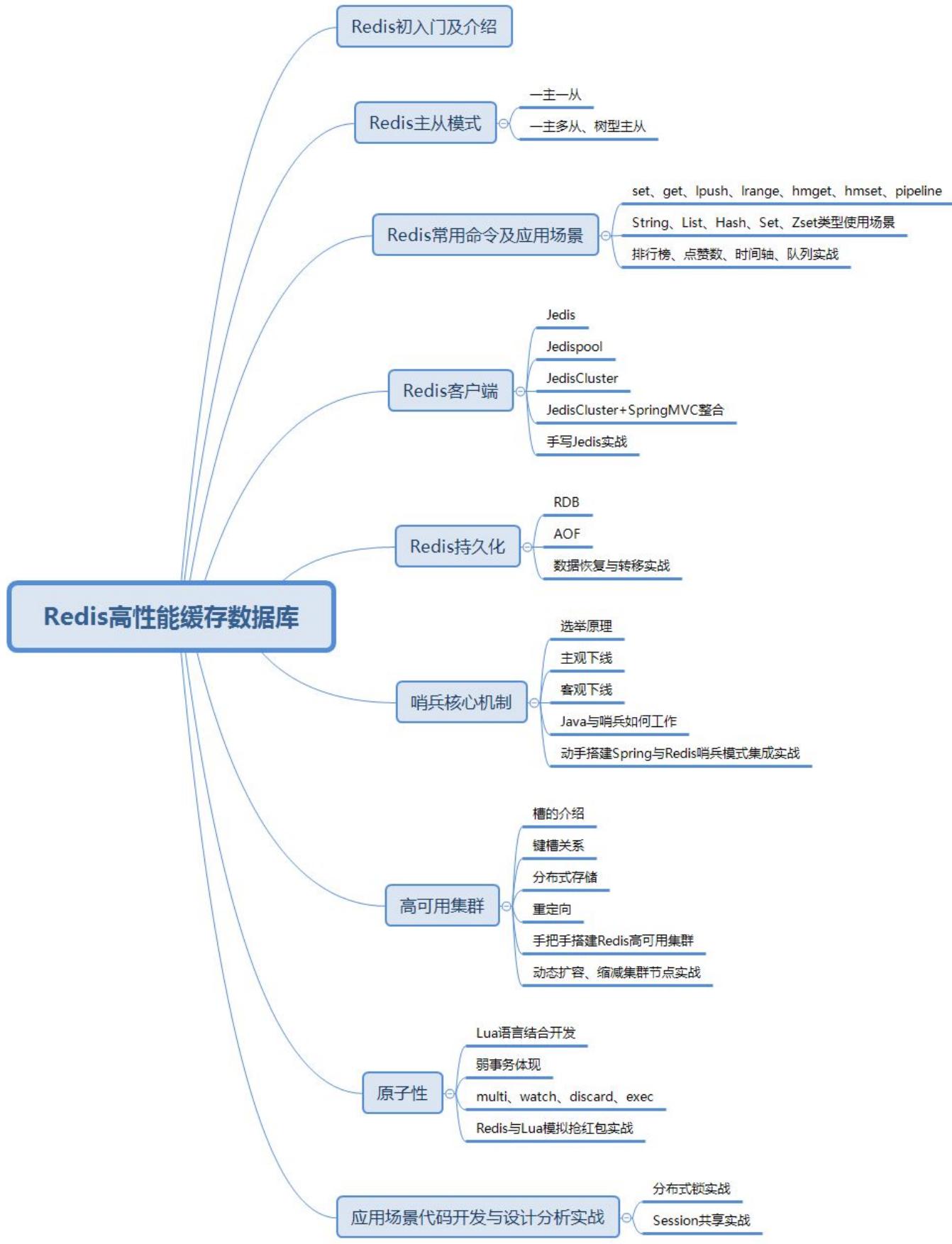
Spring 5新特性

- 容器增强
- 函数式编程
- webFlux模块介绍
- kotlin介绍
- Testing改进
- 兼容性问题

Mysql

- Mybatis组成
- 核心源码分析
- 手写mybatis框架

数据存储(最常见的是 MySQL、Redis)



分布式(分布式锁, 事务等)



除了这些东西还有什么其他问题：

实际场景题

实际场景题就是对你的知识运用能力以及思维能力的考察。建议在平时养成多思考问题的习惯，这样面试的时候碰到这样的问题就不至于慌了。另外，如果自己实在不会就给面试官委婉的说一下，面试官可能会给你提醒一下。切忌不懂装懂，乱答一气。

面试官可能会问你类似这样的问题：

1. 假设你要做一个银行 app，有可能碰到多个人同时向一个账户打钱的情况，有可能碰到什么问题，如何解决（锁）？
2. 你是怎么保证你的代码质量和正确性的？
3. 下单过程中是下订单减库存还是付款减库存，分析一下两者的优劣。
4. 同时给 10 万个人发工资，怎么样设计并发方案，能确保在 1 分钟内全部发完。
5. 如果让你设计 xxx 系统的话，你会如何设计。

生活

1. 一般到最后的 HR 面的时候，面试官基本就是和你聊聊天。他可能会问你类似如下的问题：
2. 父母是做什么的，具体一点

3. 自己平时是如何学习的

4. 平时的兴趣爱好是什么

性格/其他

1. 主要是看你个人的性格以及价值观是否适合他们公司，比如他会问你类似下面的问题：

2. 遇到压力大的情况自己是如何处理的

3. 遇到很难解决的困难怎么办

4. 遇到不是很喜欢同项目组的某个成员的情况怎么办

5. 如何看待加班

6. 你觉得自己有什么缺点/优点

总结强调

一定要谨慎对待写在简历上的东西，一定要对简历上的东西非常熟悉。因为一般情况下，面试官都是会根据你的简历来问的；能有一个上得了台面的项目也非常重要，这很可能是面试官会大量发问的地方，所以在面试之前好好回顾一下自己所做的项目；和面试官聊基础知识比如设计模式的使用、多线程的使用等等，可以结合具体的项目场景或者是自己在平时是如何使用的；建议提前了解一下自己想要面试的公司的价值观，判断一下自己究竟是否适合这个公司。

-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」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. IntelliJ IDEA 2019.3 这回真的要飞起来了

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？



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！

8 点建议：助你写出优雅的 Java 代码

Java后端 2019-11-04

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

原文地址:<https://dzone.com/articles/7-tips-to-write-better-java-code-you-should-know-1>^[1]

在每一位刚入行的程序员的心中，编写程序都是一门神圣的艺术创作。他们无不希望自己的代码作品既简洁清晰，又可读性强，而且还具有一定的容错能力。本文将为您带来八点建议和技巧，以帮助您编写出简洁、干练的 Java 代码。其中的有些可能会让你觉得有些不可思议，但是请相信我，如下的每一条我都亲身实践过的。

1. 使用 IntelliJ IDEA 作为您的集成开发环境 (IDE)

虽然我已经使用了六年的 Eclipse 和三年的 NetBeans，而且我仍然偶尔会使用到它们，但是如今的大部分时间，我只使用 IntelliJ IDEA。我并不是想在这里展开有关 IDE 的大讨论，而只是想告诉您，**IDEA 能够根据其自有的最佳整合实践标准，持续提示您写出更短、更好、更简洁的代码。**

您只需要按下 Alt + Enter 组合键，它就会自动开始为您工作了。在大多数时候，IntelliJ IDEA 能够为您提供各种智能且实用的代码建议。当然，您也可以向它学到不同的编程知识与技巧。

另外，**推荐使用 IDEA 的快捷键来加快编码速度**，IDEA 有很多非常方便的快捷键和功能，用的多了自然就熟悉了！

为了更好地获取 IDEA 的服务性能，您最好在自己的电脑上采用固态硬盘 (SSD)。就我自己的那台旧式笔记本电脑而言，由于使用的是传统硬盘 (HDD)，它已无法顺畅地运行 IDEA 了。因此我个人建议您至少使用一颗 256 GB 大小的固态硬盘。

2. 使用 JDK 8 或更高版本

JDK 8 及其更高版本引入了诸如：lambda 表达式 (lambda expression, 一种匿名函数)、功能接口、流式接口 (Stream APIs，提供更为可读的源代码实现方法) 等许多新的功能，这些都有助于您写出更简短、更高性能的 Java 代码。

当然，您并不需要去逐一地记住这些功能，因为前面提到的 IDEA 会帮助您在实际编程的过程中，实现这些功能与服务。这也就是为什么我首先向您建议使用 IDEA 的原因。

如果你对 Java8 新特性还不了解的话，可以查看 Java 入门教

程：<https://github.com/Snailclimb/JavaGuide/blob/master/docs/java/What's%20New%20in%20JDK8/Java8Tutorial.md>^[2]；

如果你想深入 Java8 新特性，我可以推荐你一些不错的学习资源：

<https://github.com/Snailclimb/JavaGuide/blob/master/docs/java/What's%20New%20in%20JDK8/Java8%E6%95%99%E7%A8%8B%E6%8E%A8%E8%8D%90.md>^[3]

3. 使用 Maven/Gradle

请使用 Maven(一个采用纯 Java 编写的开源项目管理工具，请参见) 或 Gradle(一个基于 Apache Ant 和 Maven 的项目自动化构建工具，请参见) 来管理代码中的依赖关系，以及构建和部署自己的项目。

大部分 Java 后端开发人员都用的是 Maven，或许是因为几乎所有 Java 开发者都熟悉 XML。下面我们来看看 Maven 能为我们做什么。

Maven 能为我们做什么？

- 我们可以使用 maven 轻松构建项目。
- 我们可以使用 maven 的帮助轻松添加项目的 jar 和其他依赖项。
- Maven 提供项目信息（日志文档，依赖列表，单元测试报告等）
- 在更新 JAR 和其他依赖项的中央存储库时，Maven 对项目非常有帮助。
- 在 Maven 的帮助下，我们可以将任意数量的项目构建为输出类型，如 JAR，WAR 等，而无需执行任何脚本编写。
- 使用 Maven，我们可以轻松地将我们的项目与源代码控制系统（例如 Subversion 或 Git）集成。

4. 使用 Lombok

是时候向 `setter/getter`、`ashcode>equals`、以及 `constructors/toString` 等样板式代码 (boilerplate code) 说再见了，您只需要一个注解：`@Data` 就能统统搞定了。

Lombok 是一款可以通过简单的注解形式，来帮助开发者简化并消除 Java 代码臃肿的工具（具体请参见）。它不但能够减少您的代码编写量，还能够帮助您打理那些生成的字节码。

5. 编写单元测试

可测试的代码通常意味着在组织结构上具有更合理、更简洁的代码质量。因为它会驱使您去事先管理好各个类之间的关系、各种方法的访问级别、以及其他方面。我甚至发现：即使是最小的单元测试也能够促进更快、更便捷的开发进程，进而能够让自己写出更加短、平、快的 Java 代码。

当然在现实开发工作中，您总会听到一些诸如“我根本没有时间来编写单元测试”或“项目时间节点将至，不要浪费时间些单元测试了”之类的反对意见。这些听起来貌似很合理，但是根据我的经验，在多数情况下，事实并非如此。

如果您没有时间去编写单元测试，那您是否有更多的时间，去修复代码中那些可见、或不可见的 bug 呢？如果跳过了单元测试，那些仓促完成的代码将无法保证稳定性。特别对于一些新的代码变更而言，您完全无法通过及时的反馈途径，知晓那些新产生的代码是否存在错误隐患，是否会在将来运行的某个特定场景中产生不可预知的异常问题。

一般而言，Junit 和 TestNG 是两款非常优秀的 Java 应用、及单元测试框架。而我个人则更喜欢使用 TestNG。

6. 重构：常见，但也很慢

简洁干练的 Java 程序代码从来不是一蹴而就的，它往往需要您进行反复地琢磨与改进。通过逐行进行代码重构、和运行各种测试用例，您可以确保自己的更改不会破坏既有代码的正确功能。

同样，IDEA 极大地提供了对于代码重构的支持，其中包括提取方法 (extract method，将某个大的函数拆分为多个小函数)、重命名、内联 (inline) 等功能。

当然，如果您对代码重构是什么，以及它的作用不太了解的话，Martin Fowler 的经典著作《重构：改善既有代码的设计 (第 2 版)，Refactoring: Improving the Design of Existing Code (2nd Edition)》绝对是一本您必备的参考书。

Tips：欢迎大家关注微信公众号：Java后端，来获取更多推送。

7.注意代码规范

从学习编程的第一天起就要养成不错的编码习惯，包、类、方法的命名这些是最基本的。

推荐阅读：

阿里巴巴 Java 开发手册（详尽版）[https://github.com/alibaba/p3c/blob/master/阿里巴巴Java开发手册\(详尽版\).pdf](https://github.com/alibaba/p3c/blob/master/阿里巴巴Java开发手册(详尽版).pdf)^[4]

Google Java 编程风格指南：<http://www.hawstein.com/posts/google-java-style.html>^[5]

Effective Java 第三版中文版：<https://legacy.gitbook.com/book/jiapengcai/effective-java>^[6]

8.定期联络客户，以获取他们的反馈

最后一点，可能也是最重要的：客户花钱让您通过编写代码，来解决他们的问题、满足他们的需求、并解决他们的痛点。然而，您可能在不知不觉得中花费了太多的时间，去实现自以为重要、却对客户无关紧要的特殊功能，进而忽略了代码整体的健壮性和可维护性。那么，我们怎么才能够尽早地发现该问题呢？请保持与客户经常联系，以尽早地获取他们的反馈。

话说回来，知易行难，即使是富有经验的产品经理也不一定能在较短的时间内领悟需求的真谛，何况是那些满脑子只注重功能实现的“码农”们呢？

因此，一个实用的建议是：**如果您不能直接联络到最终用户的话，请尽量与该系统的产品经理、或运维人员进行礼貌、且频繁的沟通。磨刀不误砍柴工，这些时间的投入对于后期时间的节省是绝对值得的。**

总结

在过去的多年编程实践和项目应用中，我一直受益于上述八点心得。在此，我希望它们也同样能给您的代码工作带来帮助。祝您编程愉快！

-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. 推荐一位大神, 手握 GitHub 16000 star
2. 附源码! Spring Boot 并发登录人数控制
3. 为什么 Redis 单线程却能支撑高并发?
4. 干货! MySQL 数据库开发规范
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!

Github标星10.8K！Java实战博客项目分享

Java后端 1月5日

点击上方 Java后端, 选择 **设为星标**

优质文章, 及时送达

来源:开源最前线(ID:OpenSourceTop)

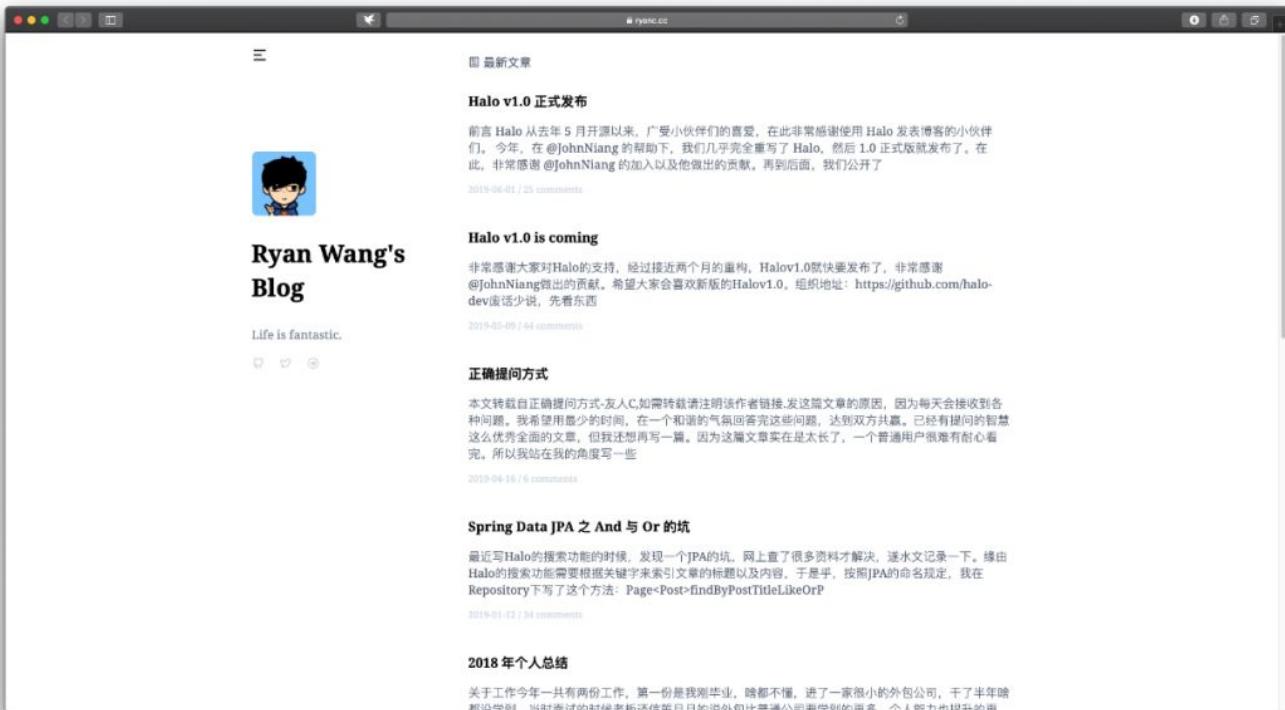
作为程序员每天就是不停的敲代码, 改Bug, 写起代码来那真是行云如流水, 但要你码出点文字, 写点技术总结, 好像比登天还难。

不过, 现在已经有越来越多的程序员喜欢上写博客, 分享一些自己的观点, 或者通过写博客加深自己对某项知识的认识, 使自己的知识体系更加健全。

国内很多程序员大牛都有自己的博客, 比如王垠、计算机科普博主阮一峰、vue.js项目作者尤雨溪、Python, Git系列教程作者廖雪峰、MegaEase创始人陈皓、PHP开发组成员鸟哥等。

对于程序员写博客这事, 可能开始会很困难, 想半天依然不知道写点什么, 建议你一开始, 就在博客上简单总结一些当下正在学习的笔记, 也可以记录一些自己在学习或者工作中遇到的一些问题, 然后再慢慢转向个人输出。

如果你也想有一个自己的博客, 那你今天算是来对地方了, 今天猿妹和大家分享一款现代化的个人独立博客系统。话不多说先让看看部分预览视图:



Ryan Wang
Life is fantastic.
Sichuan, Chengdu

40 4 23

关注数

日本 日志 其他 学习笔记 未分类

链接

SNAIL Blog	http://log.co...
大伟	http://taifaa.com
张亚东博客	https://www.zhyd.me
小米笔记	https://netem.cn
无趣	https://fhigherop...

2019-06-01 节日笔记
Halo v1.0 正式发布

前言 Halo 从去年 5 月开源以来，广受小伙伴们的喜爱，在此非常感谢使用 Halo 发表博客的小伙伴们。今年，在 @JohnNiang 的帮助下，我们几乎完全重写了 Halo，然后 1.0 正式版就发布了。在此，非常感谢 @JohnNiang 的加入以及他做出的贡献。再割后见，我们公开了

阅读更多...

最新文章

- 2019-06-01 Halo v1.0 正式发布
- 2019-05-09 Halo v1.0 is coming
- 2019-04-16 正确提问方式
- 2019-01-12 Spring Data JPA 之 And 与 Or 的坑
- 2018-01-01 2018 年个人总结

标签

- Flutter 1
- Vue 1
- jPress 1
- JavaScript 1
- Docker 4
- 敏捷 2
- 生活 2
- 评论系统 6
- HTML 1
- Holosh 1
- 文件 8
- 文本 3
- 思维 2
- 计划 1
- 表达 2
- JITZ 2
- JavaWeb 2
- maven 4
- SQL 5
- css 3
- SpringBoot 14
- Java 21
- Halo 16

Halo Dashboard

仪表盘 文章 页眉 附件 评论 外观 用户 系统

首页 / 仪表盘

文章 39 | 评论 648 | 总访问 98,146 | 建立天数 501

新动态

最近文章	最近评论
Halo v1.0 正式发布	2019-06-01 21:12
Halo v1.0 is coming	2019-05-09 00:24
正确提问方式	2019-04-16 10:17
小米 MIX3 体验	2019-03-01 17:31
Spring Data JPA 之 And 与 Or 的坑	2019-01-12 13:03

速记

写点什么吧...

操作记录

文章删除	1分钟前	写给你的信
用户登出	1分钟前	Ryan Wang
文章删除	2小时前	2019-09-11-16-41-48
文章修改	3小时前	2019-09-11-16-41-48
文章修改	3小时前	2019-09-11-16-41-48

Proudly power by [Halo](#)

The screenshot shows the Halo Dashboard interface. At the top, there are navigation links: '首页' (Home), '文章' (Articles), '日志' (Logs), '附件' (Attachments), '评论' (Comments), '外观' (Appearance), '用户' (Users), and '系统' (System). Below the navigation is a search bar with fields for '关键词' (Keywords), '文章状态' (Article Status), '分类目录' (Category Directory), and buttons for '搜索' (Search) and '重置' (Reset). A large table lists 10 blog posts. Each post includes a checkbox, a title, status ('已发布' - Published), category ('学习笔记' - Learning Notes), tags (e.g., 'Halo', 'Java', 'SpringBoot', 'Vue', 'Flutter'), comments count (e.g., 21, 44, 8), views count (e.g., 2846, 2109, 1162), publish time (e.g., 2019-06-01 21:12, 2019-05-09 00:24, 2019-04-16 10:17), and three operation buttons: '编辑' (Edit), '回收站' (Recycle Bin), and '设置' (Settings).

Halo是一款使用Java开发的开源博客系统，主要具备以下特性：

使用 Spring Boot框架，方便部署和更新，只需要一行命令便可完成安装。下载最新的 Halo安装包：

```
curl -L https://github.com/halo-dev/halo/releases/download/v1.1.1/halo-1.1.1.jar --output halo-latest.jar
```

或者

```
wget https://github.com/halo-dev/halo/releases/download/v1.1.1/halo-1.1.1.jar -O halo-latest.jar
```

启动 Halo

```
java -jar halo-latest.jar
```

完备的 Markdown 编辑器以及文章 /页面系统，包含分类 /标签 /预览图等；具备完善的评论系统（邮件提醒，盖楼，表情），另外还支持部分三方评论系统（如 Valine， Disqus 等），可以随意切换，支持任何主题；目前有6款主题模板，可以自由选择切换。

Tips：关注公众号：Java后端，每日技术博文推送。

The screenshot shows the GitHub repository page for 'halo-dev/halo'. At the top, it displays the repository name 'halo-dev / halo', a 'Sponsor' button, a 'Watch' button (278), a 'Star' button (10.3k), a 'Fork' button (3.1k), and a 'Code' tab. Below the header, there's a brief description: 'Halo 一款现代化的个人独立博客系统 <https://halo.run>' followed by a list of tags: 'java', 'halo', 'blog', 'blog-engine', 'spring-boot', 'java-blog', and 'cms'. Key statistics are shown: 1,473 commits, 2 branches, 0 packages, 40 releases, 16 contributors, and a license of 'GPL-3.0'. At the bottom, there are buttons for 'Create new file', 'Upload files', 'Find file', and a prominent green 'Clone or download' button. A note at the bottom right indicates the latest commit was made on 1 Oct.

目前，Halo已经在Github上标星**10.8K**，**3.6K**的Fork。（Github地址：<https://github.com/halo-dev/halo>），感兴趣的伙伴们可以体验一下。复制：<https://github.com/halo-dev/halo> 到浏览器打开即可下载项目源码。

- END -

推荐阅读

1. Tomcat 在 Spring Boot 中是如何启动的
2. 别乱提交代码了
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 

[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！

Google 出品 Java 编码规范，强烈推荐！

Java后端 2019-10-09

点击上方 Java后端, 选择设为星标

优质文章, 及时送达

编辑: | 可可

来源 | google.github.io/styleguide/javaguide.html

这是: [Google Python 编码规范](#), Google官方的Java编程风格规范。与其它的编程风格指南一样, 这里所讨论的不仅仅是编码格式美不美观的问题, 同时也讨论一些约定及编码标准。这份规范主要侧重于我们所普遍遵循的规则, 对于那些不是明确强制要求的, 我们尽量避免提供意见。

1.1 术语说明

1. 术语class可表示一个普通类, 枚举类, 接口或是annotation类型(`@interface`)
2. 术语comment只用来指代实现的注释(implementation comments), 我们不使用「documentation comments」一词, 而是用Javadoc。

1.2 指南说明

本文中的示例代码并不作为规范。也就是说, 虽然示例代码是遵循Google编程风格, 但并不意味着这是展现这些代码的唯一方式。示例中的格式选择不应该被强制定为规则。

源文件基础

2.1 文件名

源文件以其最顶层的类名来命名, 大小写敏感, 文件扩展名为`.java`。

2.2 文件编码: UTF-8

源文件编码格式为UTF-8。

2.3 特殊字符

2.3.1 空白字符

除了行结束符序列, ASCII水平空格字符(0x20, 即空格)是源文件中唯一允许出现的空白字符, 这意味着:

1. 所有其它字符串中的空白字符都要进行转义。
2. 制表符不用于缩进。

2.3.2 特殊转义序列

对于具有特殊转义序列的任何字符(\b, \t, \n, \f, \r, ", '及), 我们使用它的转义序列, 而不是相应的八进制(比如 \012)或Unicode(比如 \u000a)转义。

2.3.3 非ASCII字符

对于剩余的非ASCII字符，是使用实际的Unicode字符(比如 ∞)，还是使用等价的Unicode转义符(比如\u221e)，取决于哪个能让代码更易于阅读和理解。

Tip: 在使用Unicode转义符或是一些实际的Unicode字符时，建议做些注释给出解释，这有助于别人阅读和理解。

例如：

```
1  String unitAbbrev = "\u03bc";           // 费，即使没有注释也非常清晰
2  String unitAbbrev = "\u03bcs";          // "\u03bc" 允许，但没有理由要这样做
3  String unitAbbrev = "\u03bcs";          // Greek letter mu, "s" 允许，但这样做显得笨拙还容易出错
4  String unitAbbrev = "\u03bcs";          // 很糟，读者根本看不出这是什么
5  return '\ufffe' + content;             // byte order mark. Good, 对于非打印字符，使用转义，并在必要时写上注释
```

Tip: 永远不要由于害怕某些程序可能无法正确处理非ASCII字符而让你的代码可读性变差。当程序无法正确处理非ASCII字符时，它自然无法正确运行，你就会去fix这些问题了。(言下之意就是大胆去用非ASCII字符，如果真的有需要的话)

源文件结构

一个源文件包含(按顺序地)：

1. 许可证或版权信息(如有需要)
2. package语句
3. import语句
4. 一个顶级类(**只有一个**)

以上每个部分之间用一个空行隔开。

3.1 许可证或版权信息

如果一个文件包含许可证或版权信息，那么它应当被放在文件最前面。

3.2 package语句

package语句不换行，列限制(4.4节)并不适用于package语句。(即package语句写在一行里)

3.3 import语句

3.3.1 import不要使用通配符

即，不要出现类似这样的import语句：`import java.util.*;`

3.3.2 不要换行

import语句不换行，列限制(4.4节)并不适用于import语句。(每个import语句独立成行)

3.3.3 顺序和间距

import语句可分为以下几组，按照这个顺序，每组由一个空行分隔：

1. 所有的静态导入独立成组
2. **com.google imports**(仅当这个源文件是在 **com.google**包下)
3. 第三方的包。每个顶级包为一组，字典序。例如：android, com, junit, org, sun

4. `java imports`
5. `javax imports`

组内不空行，按字典序排列。

3.4 类声明

3.4.1 只有一个顶级类声明

每个顶级类都在一个与它同名的源文件中(当然，还包含 `.java`后缀)。

例外：`package-info.java`，该文件中可没有 `package-info`类。

3.4.2 类成员顺序

类的成员顺序对易读性有很大的影响，但这也不存在唯一的通用法则。不同的类对成员的排序可能是不同的。最重要的一点，每个类应该以某种逻辑去排序它的成员，维护者应该要能解释这种排序逻辑。

比如，新的方法不能总是习惯性地添加到类的结尾，因为这样就是按时间顺序而非某种逻辑来排序的。

3.4.2.1 重载：永不分离

当一个类有多个构造函数，或是多个同名方法，这些函数/方法应该按顺序出现在一起，中间不要放进其它函数/方法。

格式

术语说明：块状结构(block-like construct)指的是一个类，方法或构造函数的主体。需要注意的是，数组初始化中的初始值可被选择性地视为块状结构(4.8.3.1节)。

4.1 大括号

4.1.1 使用大括号(即使是可选的)

大括号与 `if,else,for,do,while`语句一起使用，即使只有一条语句(或是空)，也应该把大括号写上。

4.1.2 非空块：K & R 风格

对于非空块和块状结构，大括号遵循Kernighan和Ritchie风格 (Egyptian brackets)：

1. 左大括号前不换行
2. 左大括号后换行
3. 右大括号前换行
4. 如果右大括号是一个语句、函数体或类的终止，则右大括号后换行；否则不换行。例如，如果右大括号后面是`else`或逗号，则不换行。

示例：

```
1     return new MyClass() {
2         @Override public void method() {
3             if (condition()) {
4                 try {
5                     something();
6                 } catch (ProblemException e) {
7                     recover();
8                 }
9             }
10        }
11    };
```

4.8.1节给出了enum类的一些例外。

4.1.3 空块：可以用简洁版本

一个空的块状结构里什么也不包含，大括号可以简洁地写成 `{}`，不需要换行。例外：如果它是一个多块语句的一部分(if/else 或 try/catch/finally)，即使大括号内没内容，右大括号也要换行。

示例：

```
1     void doNothing() {}
```

4.2 块缩进：2个空格

每当开始一个新的块，缩进增加2个空格，当块结束时，缩进返回先前的缩进级别。缩进级别适用于代码和注释。(见4.1.2节中的代码示例)

4.3 一行一个语句

每个语句后要换行。

4.4 列限制：80或100

一个项目可以选择一行80个字符或100个字符的列限制，除了下述例外，任何一行如果超过这个字符数限制，必须自动换行。

例外：

1. 不可能满足列限制的行(例如，Javadoc中的一个长URL，或是一个长的JSNI方法参考)。
2. `package`和`import`语句(见3.2节和3.3节)。
3. 注释中那些可能被剪切并粘贴到shell中的命令行。

4.5 自动换行

术语说明：一般情况下，一行长代码为了避免超出列限制(80或100个字符)而被分为多行，我们称之为自动换行(line-wrapping)。

我们并没有全面、确定性的准则来决定在每一种情况下如何自动换行。很多时候，对于同一段代码会有好几种有效的自动换行方式。、

Tip: 提取方法或局部变量可以在不换行的情况下解决代码过长的问题(是合理缩短命名长度吧)

4.5.1 从哪里断开

自动换行的基本准则是：更倾向于在更高的语法级别处断开。

1. 如果在非赋值运算符处断开，那么在该符号前断开(比如+，它将位于下一行)。注意：这一点与Google其它语言的编程风格不同(如C++和JavaScript)。这条规则也适用于以下“类运算符”符号：点分隔符(.)，类型界限中的&(<TextendsFoo&Bar>)，catch块中的管道符号(catch(FooException|BarException))
2. 如果在赋值运算符处断开，通常的做法是在该符号后断开(比如=，它与前面的内容留在同一行)。这条规则也适用于foreach语句中的分号。
3. 方法名或构造函数名与左括号留在同一行。
4. 逗号(,)与其前面的内容留在同一行。

4.5.2 自动换行时缩进至少+4个空格

自动换行时，第一行后的每一行至少比第一行多缩进4个空格(注意：制表符不用于缩进。见2.3.1节)。

当存在连续自动换行时，缩进可能会多缩进不只4个空格(语法元素存在多级时)。一般而言，两个连续行使用相同的缩进当且仅当它们开始于同级语法元素。

第4.6.3水平对齐一节中指出，不鼓励使用可变数目的空格来对齐前面行的符号。

4.6 空白

4.6.1 垂直空白

以下情况需要使用一个空行：

1. 类内连续的成员之间：字段，构造函数，方法，嵌套类，静态初始化块，实例初始化块。**例如：**两个连续字段之间的空行是可选的，用于字段的空行主要用来对字段进行逻辑分组。
2. 在函数体内，语句的逻辑分组间使用空行。
3. 类内的第一个成员前或最后一个成员后的空行是可选的(既不鼓励也不反对这样做，视个人喜好而定)。
4. 要满足本文档中其他节的空行要求(比如3.3节：import语句)

多个连续的空行是允许的，但没有必要这样做(我们也不鼓励这样做)。

4.6.2 水平空白

除了语言需求和其它规则，并且除了文字，注释和Javadoc用到单个空格，单个ASCII空格也出现在以下几个地方：

1. 分隔任何保留字与紧随其后的左括号(())(如if,for,catch等)。
2. 分隔任何保留字与其前面的右大括号}()((如else,catch)。
3. 在任何左大括号前{}，两个例外：@SomeAnnotation({a,b}) (不使用空格)。String[] x=foo;(大括号间没有空格，见下面的Note)。
4. 在任何二元或三元运算符的两侧。这也适用于以下「类运算符」符号：类型界限中的&(<TextendsFoo&Bar>)。catch块中的管道符号(catch(FooException|BarException))。foreach语句中的分号。
5. 在,;;及右括号())后。
6. 如果在一条语句后做注释，则双斜杠//两边都要空格。这里可以允许多个空格，但没有必要。
7. 类型和变量之间：List<string>list。</string>
8. 数组初始化中，大括号内的空格是可选的，即new int[]{5,6}和new int[] {5,6}都是可以的。

Note: 这个规则并不要求或禁止一行的开关或结尾需要额外的空格，只对内部空格做要求。

4.6.3 水平对齐：不做要求

术语说明：水平对齐指的是通过增加可变数量的空格来使某一行的字符与上一行的相应字符对齐。

这是允许的(而且在不少地方可以看到这样的代码)，但Google编程风格对此不做要求。即使对于已经使用水平对齐的代码，我们也不需要去保持这种风格。

以下示例先展示未对齐的代码，然后是对齐的代码：

```
1  private int x;          // this is fine
2  private Color color;   // this too
3
4  private int    x;       // permitted, but future edits
5  private Color color;  // may leave it unaligned|
```

Tip: 对齐可增加代码可读性，但它为日后的维护带来问题。考虑未来某个时候，我们需要修改一堆对齐的代码中的一行。这可能导致原本很漂亮的对齐代码变得错位。很可能它会提示你调整周围代码的空白来使这一堆代码重新水平对齐(比如程序员想保持这种水平对齐的风格)，这就会让你做许多的无用功，增加了reviewer的工作并且可能导致更多的合并冲突。

4.7 用小括号来限定组：推荐

除非作者和reviewer都认为去掉小括号也不会使代码被误解，或是去掉小括号能让代码更易于阅读，否则我们不应该去掉小括号。我们没有理由假设读者能记住整个Java运算符优先级表。

4.8 具体结构

4.8.1 枚举类

枚举常量间用逗号隔开，换行可选。

没有方法和文档的枚举类可写成数组初始化的格式：

```
1  private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

由于枚举类也是一个类，因此所有适用于其它类的格式规则也适用于枚举类。

4.8.2 变量声明

4.8.2.1 每次只声明一个变量

不要使用组合声明，比如 `inta,b;`。

4.8.2.2 需要时才声明，并尽快进行初始化

不要在一个代码块的开头把局部变量一次性都声明了(这是C语言的做法)，而是在第一次需要使用它时才声明。局部变量在声明时最好就进行初始化，或者声明后尽快进行初始化。

4.8.3 数组

4.8.3.1 数组初始化：可写成块状结构

数组初始化可以写成块状结构，比如，下面的写法都是OK的：

```
1  new int[] {
2      0, 1, 2, 3
3  }
4
5  new int[] {
6      0,
7      1,
8      2,
9      3
10 }
11
12 new int[] {
13     0, 1,
14     2, 3
15 }
16
17 new int[]{0, 1, 2, 3}
```

4.8.3.2 非C风格的数组声明

中括号是类型的一部分：`String[] args`，而非`Stringargs[]`。

4.8.4 switch语句

术语说明：switch块的大括号内是一个或多个语句组。每个语句组包含一个或多个switch标签(`caseFOO:`或`default:`)，后面跟着一条或多条语句。

4.8.4.1 缩进

与其它块状结构一致，switch块中的内容缩进为2个空格。

每个switch标签后新起一行，再缩进2个空格，写下一条或多条语句。

4.8.4.2 Fall-through：注释

在一个switch块内，每个语句组要么通过`break,continue,return`或抛出异常来终止，要么通过一条注释来说明程序将继续执行到下一个语句组，任何能表达这个意思的注释都是OK的(典型的是用`// fall through`)。这个特殊的注释并不需要在最后一个语句组(一般是`default`)中出现。

示例：

```
1  switch (input) {
2      case 1:
3      case 2:
4          prepareOneOrTwo();
5          // fall through
6      case 3:
7          handleOneTwoOrThree();
8          break;
9      default:
10         handleLargeNumber(input);
11 }
```

4.8.4.3 default的情况要写出来

每个switch语句都包含一个default语句组，即使它什么代码也不包含。

4.8.5 注解(Annotations)

注解紧跟在文档块后面，应用于类、方法和构造函数，一个注解独占一行。这些换行不属于自动换行(第4.5节，自动换行)，因此缩进级别不变。例如：

```
1  @Override
2  @Nullable
3  public String getNameIfPresent() { ... }
```

例外：单个的注解可以和签名的第一行出现在同一行。例如：

```
1  @Override public int hashCode() { ... }
```

应用于字段的注解紧随文档块出现，应用于字段的多个注解允许与字段出现在同一行。例如：

```
1  @Partial @Mock DataLoader loader;
```

参数和局部变量注解没有特定规则。

4.8.6 注释

4.8.6.1 块注释风格

块注释与其周围的代码在同一缩进级别。它们可以是 /* ... */风格，也可以是 // ...风格。对于多行的 /* ... */注释，后续行必须从 *开始，并且与前一行的 *对齐。以下示例注释都是OK的。

```
1  /*
2   * This is           // And so           /* Or you can
3   * okay.            // is this.        * even do this. */
4   */
```

注释不要封闭在由星号或其它字符绘制的框架里。

Tip：在写多行注释时，如果你希望在必要时能重新换行(即注释像段落风格一样)，那么使用 /* ... */。

4.8.7 Modifiers

类和成员的modifiers如果存在，则按Java语言规范中推荐的顺序出现。

命名约定

5.1 对所有标识符都通用的规则

标识符只能使用ASCII字母和数字，因此每个有效的标识符名称都能匹配正则表达式 `\w+`。

在Google其它编程语言风格中使用的特殊前缀或后缀，如 `name_`, `mName`, `s_name` 和 `kName`，在Java编程风格中都不再使用。

5.2 标识符类型的规则

5.2.1 包名

包名全部小写，连续的单词只是简单地连接起来，不使用下划线。

5.2.2 类名

类名都以 `UpperCamelCase` 风格编写。

类名通常是名词或名词短语，接口名称有时可能是形容词或形容词短语。现在还没有特定的规则或行之有效的约定来命名注解类型。

测试类的命名以它要测试的类的名称开始，以 `Test` 结束。例如，`HashTest` 或 `HashIntegrationTest`。

5.2.3 方法名

方法名都以 `lowerCamelCase` 风格编写。

方法名通常是动词或动词短语。

下划线可能出现在JUnit测试方法名称中用以分隔名称的逻辑组件。一个典型的模式是：`test<MethodUnderTest>_<state>`，例如 `testPop_emptyStack`。并不存在唯一正确的方式来命名测试方法。

5.2.4 常量名

常量名命名模式为 `CONSTANT_CASE`，全部字母大写，用下划线分隔单词。

每个常量都是一个静态final字段，但不是所有静态final字段都是常量。在决定一个字段是否是一个常量时，考虑它是否真的感觉像是一个常量。例如，如果任何一个该实例的观测状态是可变的，则它几乎肯定不会是一个常量。只是永远不打算改变对象一般是不够的，它要真的一直不变才能将它示为常量。

```
1 // Constants
2 static final int NUMBER = 5;
3 static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
4 static final Joiner COMMA_JOINER = Joiner.on(','); // because Joiner is immutable
5 static final SomeMutableType[] EMPTY_ARRAY = {};
6 enum SomeEnum { ENUM_CONSTANT }
7
8 // Not constants
9 static String nonFinal = "non-final";
10 final String nonStatic = "non-static";
11 static final Set<String> mutableCollection = new HashSet<String>();
12 static final ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);
13 static final Logger logger = Logger.getLogger(MyClass.getName());
14 static final String[] nonEmptyArray = {"these", "can", "change"};
```

这些名字通常是名词或名词短语。

5.2.5 非常量字段名

非常量字段名以 `lowerCamelCase` 风格编写。

这些名字通常是名词或名词短语。

5.2.6 参数名

参数名以 `lowerCamelCase` 风格编写。参数应该避免用单个字符命名。

5.2.7 局部变量名

局部变量名以 `lowerCamelCase` 风格编写，比起其它类型的名称，局部变量名可以有更为宽松的缩写。

虽然缩写更宽松，但还是要避免用单字符进行命名，除了临时变量和循环变量。

即使局部变量是 `final` 和不可改变的，也不应该把它示为常量，自然也不能用常量的规则去命名它。

5.2.8 类型变量名

类型变量可用以下两种风格之一进行命名：

1. 单个的大写字母，后面可以跟一个数字(如： `E`, `T`, `X`, `T2`)。
2. 以类命名方式(5.2.2节)，后面加个大写的T(如： `RequestT`, `FooBarT`)。

5.3 驼峰式命名法(CamelCase)

驼峰式命名法分大驼峰式命名法(`UpperCamelCase`)和小驼峰式命名法(`lowerCamelCase`)。有时，我们有不只一种合理的方式将一个英语词组转换成驼峰形式，如缩略语或不寻常的结构(例如“IPv6”或“iOS”)。Google指定了以下的转换方案。

名字从 `散文形式(prose form)` 开始：

1. 把短语转换为纯ASCII码，并且移除任何单引号。例如：「Müller's algorithm」将变成「Muellers algorithm」。
2. 把这个结果切分成单词，在空格或其它标点符号(通常是连字符)处分割开。推荐：如果某个单词已经有了常用的驼峰表示形式，按它的组成将它分割开(如「AdWords」将分割成「ad words」)。需要注意的是“iOS”并不是一个真正的驼峰表示形式，因此该推荐对它并不适用。
3. 现在将所有字母都小写(包括缩写)，然后将单词的第一个字母大写：每个单词的第一个字母都大写，来得到大驼峰式命名。除了第一个单词，每个单词的第一个字母都大写，来得到小驼峰式命名。
4. 最后将所有的单词连接起来得到一个标识符。

示例：

1	Prose form	Correct	Incorrect
2			
3	"XML HTTP request"	XmlHttpRequest	XMLHTTPRequest
4	"new customer ID"	newCustomerId	newCustomerID
5	"inner stopwatch"	innerStopwatch	innerStopWatch
6	"supports IPv6 on iOS?"	supportsIpv6OnIos	supportsIPv6OnIOS
7	"YouTube importer"	YouTubeImporter	
8	YoutubeImporter*		

加星号处表示可以，但不推荐。

Note: 在英语中，某些带有连字符的单词形式不唯一。例如：「nonempty」和「non-empty」都是正确的，因此方法名 `checkNonempty` 和 `checkNonEmpty` 也都是正确的。

编程实践

6.1 @Override：能用则用

只要是合法的，就把 `@Override` 注解给用上。

6.2 捕获的异常：不能忽视

除了下面的例子，对捕获的异常不做响应是极少正确的。(典型的响应方式是打印日志，或者如果它被认为是不可能的，则把它当作一个 `AssertionError` 重新抛出。)

如果它确实是不需要在 `catch` 块中做任何响应，需要做注释加以说明(如下面的例子)。

```
1  try {
2      int i = Integer.parseInt(response);
3      return handleNumericResponse(i);
4  } catch (NumberFormatException ok) {
5      // it's not numeric; that's fine, just continue
6  }
7  return handleTextResponse(response);
```

例外：在测试中，如果一个捕获的异常被命名为 `expected`，则它可以被不加注释地忽略。下面是一种非常常见的情形，用以确保所测试的方法会抛出一个期望中的异常，因此在这里就没有必要加注释。

```
1  try {
2      emptyStack.pop();
3      fail();
4  } catch (NoSuchElementException expected) {
5  }
```

6.3 静态成员：使用类进行调用

使用类名调用静态的类成员，而不是具体某个对象或表达式。

```
1  Foo aFoo = ...;
2  Foo.aStaticMethod(); // good
3  aFoo.aStaticMethod(); // bad
4  somethingThatYieldsAFoo().aStaticMethod(); // very bad
```

6.4 Finalizers: 禁用

极少会去重写 `Object.finalize`。

Tip: 不要使用`finalize`。如果你非要使用它, 请先仔细阅读和理解Effective Java 第7条款: “Avoid Finalizers”, 然后不要使用它。

Javadoc

7.1 格式

7.1.1 一般形式

Javadoc块的基本格式如下所示:

```
1  /**
2   * Multiple lines of Javadoc text are written here,
3   * wrapped normally...
4   */
5  public int method(String p1) { ... }
```

或者是以下单行形式:

```
1  /** An especially short bit of Javadoc. */
```

基本格式总是OK的。当整个Javadoc块能容纳于一行时(且没有Javadoc标记@XXX), 可以使用单行形式。

7.1.2 段落

空行(即, 只包含最左侧星号的行)会出现在段落之间和Javadoc标记(@XXX)之前(如果有的话)。除了第一个段落, 每个段落第一个单词前都有标签`<p>`, 并且它和第一个单词间没有空格。

7.1.3 Javadoc标记

标准的Javadoc标记按以下顺序出现: `@param`, `@return`, `@throws`, `@deprecated`前面这4种标记如果出现, 描述都不能为空。当描述无法在一行中容纳, 连续行需要至少再缩进4个空格。

7.2 摘要片段

每个类或成员的Javadoc以一个简短的摘要片段开始。这个片段是非常重要的, 在某些情况下, 它是唯一出现的文本, 比如在类和方法索引中。

这只是一个小片段, 可以是一个名词短语或动词短语, 但不是一个完整的句子。它不会以`A{@code Foo}isa...`或`Thismethod returns...`开头, 它也不会是一个完整的祈使句, 如`Savethe record...`。然而, 由于开头大写及被加了标点, 它看起来就像是个完整的句子。

Tip: 一个常见的错误是把简单的Javadoc写成`/** @return the customer ID */`, 这是不正确的。它应该写成`/** Returns the customer ID */`。

7.3 哪里需要使用Javadoc

7.3.1 例外：不言自明的方法

对于简单明显的方法如 `getFoo`，Javadoc是可选的(即，是可以不写的)。这种情况下除了写「Returns the foo」，确实也没有什么值得写了。

单元测试类中的测试方法可能是不言自明的最常见例子了，我们通常可以从这些方法的描述性命名中知道它是干什么的，因此不需要额外的文档说明。

Tip：如果有一些相关信息是需要读者了解的，那么以上的例外不应作为忽视这些信息的理由。例如，对于方法名 `getCanonicalName`，就不应该忽视文档说明，因为读者很可能不知道词语canonical name指的是什么。

7.3.2 例外：重写

如果一个方法重写了超类中的方法，那么Javadoc并非必需的。

7.3.3 可选的Javadoc

对于包外不可见的类和方法，如有需要，也是要使用Javadoc的。如果一个注释是用来定义一个类，方法，字段的整体目的或行为，那么这个注释应该写成Javadoc，这样更统一更友好。

- 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 ↓



推荐阅读

1. [Java后端优质文章整理](#)
2. [JDK 13 新特性一览](#)
3. [14 个实用的数据库设计技巧](#)
4. [面试官:Redis 内存满了怎么办？](#)
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

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！

Java 14 有哪些新特性？

Nathan Esquenazi Java后端 1月15日

记录为 Java 提供了一种正确实现数据类的能力，不再需要为实现数据类而编写冗长的代码。下面就来看看 Java 14 中的记录有哪些新特性。



作者 | Nathan Esquenazi

译者 | 弯月，责编 | 郭芮

出品 | CSDN (ID:CSDNnews)

以下为译文：

Java 14 即将在 2020 年 3 月正式发布。Java 以 6 个月作为新版本的发布周期，和之前的版本发布一样，JDK 14 预计将在语言本身和 JVM 级别上带来一些新特性。

如果我们看一下特性列表，我们会注意到一些开发者非常期待的语言特性：记录 (records)、switch 表达式 (在 JDK 13 中就已经存在，不过仅仅是预览模式)，模式匹配。下面让我们看下其中比较有趣的记录这一特性。

前提条件

我们需要 OpenJDK 网站中的 JDK 14 先期预览版本 (<https://jdk.java.net/14/>)。

什么是一条记录？

记录表示“数据类”，是用于保存纯数据的一种特殊的类。其他语言中已经有类似记录的结构，比如 Kotlin 的数据类。通过将类型声明为记录，通过类型即可表达意图，即只表示数据。声明记录的语法比使用普通类要简单得多，普通类通常需要实现核心 Object 方法，如 equals () 和 hashCode ()（通常称为“样板”代码）。在对于模型类（可能通过 ORM 持久化）或数据传输对象（DTOs）等事物建模时，记录是一个不错的选择。

如果想知道记录如何在 Java 语言中实现的，可以参照枚举类型。枚举也是一个具有特殊语义和优雅语法的类。由于记录和枚举仍然是类，所以类中可用的许多特性都得到了保留，因此记录在设计的简单性和灵活性之间取得了平衡。

记录是一个预览语言特性，这意味着，尽管已经完全支持了这种特性，但是还没正式进入标准 JDK 中，目前只能通过激活标志来使用。预览语言功能可能在未来的版本中更新或删除。switch 达式也与之相似，它可能在未来的版本中永存。

一个记录的例子

下面给出一个记录的范例：

```
package examples;

record Person(String firstName, String lastName) {}
```

我们定义了一个 Person 对象，包含 firstName 和 lastName 两个组件，记录的 body 为空。

然后我们对其进行编译。注意 --enable-preview 选项。

```
javac --enable-preview --release 14 Person.java
```

```
Note: Person.java uses preview language features.
Note: Recompile with -Xlint:preview for details.
```

揭露其神秘面纱

正如前面提到的，记录只是一个用于保存和暴露数据的类。

接下来让我们来看看用 javap 工具生成的字节码：

```
javap -v -p Person.class
```

字节码：

```
Classfile examples/Person.class
Last modified Dec 22, 2019; size 1273 bytes
SHA-256 checksum 6f1b325121ca32a0b6127180eff29dcac4834f9c138c9613c526a4202fef972f
Compiled from "Person.java"
final class examples.Person extends java.lang.Record
  minor version: 65535
  major version: 58
  flags: (0x0030) ACC_FINAL, ACC_SUPER
  this_class: #8           // examples/Person
  super_class: #2          // java/lang/Record
  interfaces: 0, fields: 2, methods: 6, attributes: 4
Constant pool:
#1 = Methodref    #2.#3      // java/lang/Record."<":()V
#2 = Class        #4          // java/lang/Record
#3 = NameAndType   #5:#6      // "<":()V
```

```

#4 = Utf8          java/lang/Record
#5 = Utf8
#6 = Utf8          ()V
#7 = Fieldref      #8.#9      // examples/Person.firstName:Ljava/lang/String;
#8 = Class         #10        // examples/Person
#9 = NameAndType   #11:#12    // firstName:Ljava/lang/String;
#10 = Utf8         examples/Person
#11 = Utf8         firstName
#12 = Utf8         Ljava/lang/String;
#13 = Fieldref      #8.#14    // examples/Person.lastName:Ljava/lang/String;
#14 = NameAndType   #15:#12    // lastName:Ljava/lang/String;
#15 = Utf8         lastName
#16 = Fieldref      #8.#9      // examples/Person.firstName:Ljava/lang/String;
#17 = Fieldref      #8.#14    // examples/Person.lastName:Ljava/lang/String;
#18 = InvokeDynamic #0:#19    // #0:toString:(Lexamples/Person;)Ljava/lang/String;
#19 = NameAndType   #20:#21    // toString:(Lexamples/Person;)Ljava/lang/String;
#20 = Utf8         toString
#21 = Utf8         (Lexamples/Person;)Ljava/lang/String;
#22 = InvokeDynamic #0:#23    // #0:hashCode:(Lexamples/Person;)I
#23 = NameAndType   #24:#25    // hashCode:(Lexamples/Person;)I
#24 = Utf8         hashCode
#25 = Utf8         (Lexamples/Person;)I
#26 = InvokeDynamic #0:#27    // #0>equals:(Lexamples/Person;Ljava/lang/Object;)Z
#27 = NameAndType   #28:#29    // equals:(Lexamples/Person;Ljava/lang/Object;)Z
#28 = Utf8         equals
#29 = Utf8         (Lexamples/Person;Ljava/lang/Object;)Z
#30 = Utf8         (Ljava/lang/String;Ljava/lang/String;)V
#31 = Utf8         Code
#32 = Utf8         LineNumberTable
#33 = Utf8         MethodParameters
#34 = Utf8         ()Ljava/lang/String;
#35 = Utf8         ()I
#36 = Utf8         (Ljava/lang/Object;)Z
#37 = Utf8         SourceFile
#38 = Utf8         Person.java
#39 = Utf8         Record
#40 = Utf8         BootstrapMethods
#41 = MethodHandle  6:#42      // REF_invokeStatic java/lang/runtime/ObjectMethods.bootstrap:(L
#42 = Methodref     #43.#44    // java/lang/runtime/ObjectMethods.bootstrap:(Ljava/lang/invoke/M
#43 = Class         #45        // java/lang/runtime/ObjectMethods
#44 = NameAndType   #46:#47    // bootstrap:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lar
#45 = Utf8         java/lang/runtime/ObjectMethods
#46 = Utf8         bootstrap
#47 = Utf8         (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Ty
#48 = String        #49        // firstName;lastName
#49 = Utf8         firstName;lastName
#50 = MethodHandle  1:#7       // REF_getField examples/Person.firstName:Ljava/lang/String;
#51 = MethodHandle  1:#13      // REF_getField examples/Person.lastName:Ljava/lang/String;
#52 = Utf8         InnerClasses
#53 = Class         #54        // java/lang/invoke/MethodHandles$Lookup
#54 = Utf8         java/lang/invoke/MethodHandles$Lookup
#55 = Class         #56        // java/lang/invoke/MethodHandles
#57 = Utf8         Lookup
{
private final java.lang.String firstName;
descriptor: Ljava/lang/String;
flags: (0x0012) ACC_PRIVATE, ACC_FINAL

```

```
private final java.lang.String lastName;
descriptor: Ljava/lang/String;
flags: (0x0012) ACC_PRIVATE, ACC_FINAL

public examples.Person(java.lang.String, java.lang.String);
descriptor: (Ljava/lang/String;Ljava/lang/String;)V
flags: (0x0001) ACC_PUBLIC
Code:
stack=2, locals=3, args_size=3
0: aload_0
1: invokespecial #1           // Method java/lang/Record."":()V
4: aload_0
5: aload_1
6: putfield    #7           // Field firstName:Ljava/lang/String;
9: aload_0
10: aload_2
11: putfield   #13          // Field lastName:Ljava/lang/String;
14: return
LineNumberTable:
line 3: 0
MethodParameters:
Name          Flags
firstName
lastName

public java.lang.String toString();
descriptor: ()Ljava/lang/String;
flags: (0x0001) ACC_PUBLIC
Code:
stack=1, locals=1, args_size=1
0: aload_0
1: invokedynamic #18, 0      // InvokeDynamic #0:toString:(Lexamples/Person;)Ljava/lang/String;
6: areturn
LineNumberTable:
line 3: 0

public final int hashCode();
descriptor: ()I
flags: (0x0011) ACC_PUBLIC, ACC_FINAL
Code:
stack=1, locals=1, args_size=1
0: aload_0
1: invokedynamic #22, 0      // InvokeDynamic #0:hashCode:(Lexamples/Person;)I
6: ireturn
LineNumberTable:
line 3: 0

public final boolean equals(java.lang.Object);
descriptor: (Ljava/lang/Object;)Z
flags: (0x0011) ACC_PUBLIC, ACC_FINAL
Code:
stack=2, locals=2, args_size=2
0: aload_0
1: aload_1
2: invokedynamic #26, 0      // InvokeDynamic #0>equals:(Lexamples/Person;Ljava/lang/Object;)Z
7: ireturn
LineNumberTable:
line 3: 0
public java.lang.String firstName();
```

```

descriptor: ()Ljava/lang/String;
flags: (0x0001) ACC_PUBLIC
Code:
stack=1, locals=1, args_size=1
  0: aload_0
  1: getfield    #16           // Field firstName:Ljava/lang/String;
  4: areturn
LineNumberTable:
line 3: 0

public java.lang.String lastName();
descriptor: ()Ljava/lang/String;
flags: (0x0001) ACC_PUBLIC
Code:
stack=1, locals=1, args_size=1
  0: aload_0
  1: getfield    #17           // Field lastName:Ljava/lang/String;
  4: areturn
LineNumberTable:
line 3: 0
}

SourceFile: "Person.java"
Record:
java.lang.String firstName;
  descriptor: Ljava/lang/String;

java.lang.String lastName;
  descriptor: Ljava/lang/String;

BootstrapMethods:
0: #41 REF_invokeStatic java/lang/runtime/ObjectMethods.bootstrap:(Ljava/lang/invoke/MethodHandle)
Method arguments:
#8 examples/Person
#48 firstName;lastName
#50 REF_getField examples/Person.firstName:Ljava/lang/String;
#51 REF_getField examples/Person.lastName:Ljava/lang/String;
InnerClasses:
  public static final #57= #53 of #55;  // Lookup=class java/lang/invoke/MethodHandles$Lookup of class

```

我们要特别重视以下几点：

1. 这个类被标记为 `final` ,意味着不能创建子类。
2. 和所有的枚举都以 `java.lang.Enum` 为基类一样，所有的记录都以 `java.lang.Record` 为基类。
3. 两个组件： `firstName` 和 `lastName` 都是用 `private` 和 `final` 的。
4. 有一个提供构造对象的公有构造函数：`public examples.Person(java.lang.String, java.lang.String)` 。通过查看它的字节码，我们可以知道，这个构造函数只是将两个参数赋值给这两个组件。该构造函数等价于：

```

public Person(String firstName, String lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

```

5. 有两个获取对象值的方法，分别为 `firstName()` 和 `lastName()` .

6. 自动生成 `toString()` , `hashCode()` 和 `equals()` 三个函数。他们都依赖 `invokedynamic` 来实现动态调用包含隐式实现函数在内的方法。从字节码中可以看到，有一个启动函数 `ObjectMethods.bootstrap` 来根据记录组件的名称和它的 Getter 函数，生成对应的函数。他们的表现和我们设想的一致：

```
Person john = new Person("John", "Doe");
System.out.println(john.firstName());      // John
System.out.println(john.lastName());       // Doe
System.out.println(john);                 // Person[firstName=John, lastName=Doe]

Person jane = new Person("Jane", "Doe");
Person johnCopy = new Person("John", "Doe");
System.out.println(john.hashCode());        // 71819599
System.out.println(jane.hashCode());        // 71407578
System.out.println(johnCopy.hashCode());    // 71819599
System.out.println(john.equals(jane));     // false
System.out.println(john.equals(johnCopy)); // true
```

在记录的声明中添加成员

我们不能向记录中添加实例字段，这在意料之中，因为这种数据应该设置为组件。但是我们可以添加静态字段：

```
record Person(String firstName, String lastName){
    static int x;
}
```

我们可以定义静态函数和实例函数来操作对象的状态。

```
record Person (String firstName, String lastName) {
    static int x;
    public static void dox(){
        x++;
    }
    public String getFullName(){
        return firstName + " " + lastName ;
    }
}
```

我们也可以为记录添加构造函数，也可以编辑规范构造函数（带有两个字符串参数的构造函数）。如果你想重写规范构造函数，你可以编写一个不带参数的构造函数，不需要对属性进行赋值。

```
record Person (String firstName, String lastName) {
    public Person {
        if(firstName==null||lastName==null){
            throw new IllegalArgumentException("firstName and lastName must not be null");
        // 你可以忽略属性赋值，编译器会自动为你添加赋值代码
    }
    public Person(String fullName){
        this(fullName.split(" ")[0], fullName.split(" ")[1]);
    }
}
```

记录为 Java 提供了一种正确实现数据类的能力，不再需要为实现数据类而编写冗长的代码。这让编写纯数据类代码从几行缩减为一行代码。还有一些其他预览的语言特性可以和记录搭配使用，比如模式匹配。如果想深入了解记录和相关背景，请参阅 Brian Goetz 的 [OpenJDK 文档 \(<https://cr.openjdk.java.net/~briangoetz/amber/datum.html>\)](https://cr.openjdk.java.net/~briangoetz/amber/datum.html)。

原文：<https://dzone.com/articles/a-first-look-at-records-in-java-14>

作者：Mahmoud Anouti，高级软件工程师。译者：明明如月，知名互联网公司 Java 高级开发工程师，CSDN 博客专家。

本文为 CSDN 翻译，转载请注明来源出处。

- END -

推荐阅读

1. [彻底征服 Spring AOP](#)
2. [使用 ThreadLocal 一次解决老大难问题](#)
3. [一场近乎完美基于 Dubbo 的微服务改造实践](#)
4. [GitHub 项目搜索技巧](#)
5. [Spring Boot 常见错误及解决方法](#)



喜欢文章, 点个在看 

Java 8 中 Map 骚操作之 merge() 的用法

Java后端 2019-10-17



作者 | LQ木头

juejin.im/post/5d9b455ae51d45782b0c1bfb

Java 8 最大的特性无异于更多地面向函数，比如引入了 lambda 等，可以更好地进行函数式编程。前段时间无意间发现了 map.merge() 方法，感觉还是很好用的，此文简单做一些相关介绍。首先我们先看一个例子。

merge() 怎么用？

假设我们有这么一段业务逻辑，我有一个学生成绩对象的列表，对象包含学生姓名、科目、科目分数三个属性，要求求得每个学生的总成绩。加入列表如下：

```
1  private List<StudentScore> buildATestList()
2  {
3      List<StudentScore> studentScoreList = new ArrayList<>();
4      StudentScore studentScore1 = new StudentScore() {{
5          setStuName("张三")
6      ;
7          setSubject("语文")
8      ;
9          setScore(70)
10 ;
11     }};
12     StudentScore studentScore2 = new StudentScore() {{
13         setStuName("张三")
14     ;
15         setSubject("数学")
16     ;
17         setScore(80)
18     ;
19    }};
20     StudentScore studentScore3 = new StudentScore() {{
21         setStuName("张三")
22     ;
23         setSubject("英语")
24     ;
25         setScore(65)
26     }};
```

```
26    }};
27    StudentScore studentScore4 = new StudentScore() {{
28        setStuName("李四")
29    ;
30        setSubject("语文")
31    ;
32        setScore(68)
33    ;
34    }};
35    StudentScore studentScore5 = new StudentScore() {{
36        setStuName("李四")
37    ;
38        setSubject("数学")
39    ;
40        setScore(70)
41    ;
42    }};
43    StudentScore studentScore6 = new StudentScore() {{
44        setStuName("李四")
45    ;
46        setSubject("英语")
47    ;
48        setScore(90)
49    ;
50    }};
51    StudentScore studentScore7 = new StudentScore() {{
52        setStuName("王五")
53    ;
54        setSubject("语文")
55    ;
56        setScore(80)
57    ;
58    }};
59    StudentScore studentScore8 = new StudentScore() {{
60        setStuName("王五")
61    ;
62        setSubject("数学")
63    ;
64        setScore(85)
65    ;
66    }};
67    StudentScore studentScore9 = new StudentScore() {{
68        setStuName("王五")
69    ;
70        setSubject("英语")
71    ;
72        setScore(70)
73    ;
74    }};
75
76    studentScoreList.add(studentScore1);
77    studentScoreList.add(studentScore2);
78    studentScoreList.add(studentScore3);
79    studentScoreList.add(studentScore4);
```

```
studentScoreList.add(studentScore4);
studentScoreList.add(studentScore5);
studentScoreList.add(studentScore6);
studentScoreList.add(studentScore7);
studentScoreList.add(studentScore8);
studentScoreList.add(studentScore9);

return studentScoreList;
}
```

我们先看一下常规做法：

```
1 ObjectMapper objectMapper = new ObjectMapper();
2     List<StudentScore> studentScoreList = buildATestList();
3
4     Map<String, Integer> studentScoreMap = new HashMap<>();
5     studentScoreList.forEach(studentScore -> {
6         if (studentScoreMap.containsKey(studentScore.getStuName())) {
7             studentScoreMap.put(studentScore.getStuName(),
8                     studentScoreMap.get(studentScore.getStuName()) + studentScore.
9         } else
10    {
11        studentScoreMap.put(studentScore.getStuName(), studentScore.getScore());
12    }
13 });
14
15 System.out.println(objectMapper.writeValueAsString(studentScoreMap));
16
17 // 结果如下:
18 // {"李四":228, "张三":215, "王五":235}
```

然后再看一下 `merge()` 是怎么做的：

```
1 Map<String, Integer> studentScoreMap2 = new HashMap<>();
2     studentScoreList.forEach(studentScore -> studentScoreMap2.merge(
3         studentScore.getStuName(),
4         studentScore.getScore(),
5         Integer::sum))
6 ;
7
8 System.out.println(objectMapper.writeValueAsString(studentScoreMap2));
9
10 // 结果如下:
11 // {"李四":228, "张三":215, "王五":235}
```

merge() 简介

`merge()` 可以这么理解：它将新的值赋值到 `key`（如果不存在）或更新给定的`key` 值对应的 `value`，其源码如下：

```
1 default V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)
```

```
1 default V merge(K key, V value, BiFunction<? super K, ? super V, ? extends V> remappingFunction)
2 {
3     Objects.requireNonNull(remappingFunction);
4     Objects.requireNonNull(value);
5 }
6     V oldValue = this.get(key);
7 }
8     V newValue = oldValue == null ? value : remappingFunction.apply(oldValue, value);
9 }
10 if (newValue == null)
11 {
12     this.remove(key);
13 }
14 } else
15 {
16     this.put(key, newValue);
17 }
18
19 return newValue;
20 }
```



我们可以看到原理也是很简单的，该方法接收三个参数，一个 key 值，一个 value，一个 remappingFunction，如果给定的key不存在，它就变成了 put(key, value)。

但是，如果 key 已经存在一些值，我们 remappingFunction 可以选择合并的方式，然后将合并得到的 newValue 赋值给原先的 key。

使用场景

这个使用场景相对来说还是比较多的，比如分组求和这类的操作，虽然 stream 中有相关 groupingBy() 方法，但如果你想在循环中做一些其他操作的时候，merge() 还是一个挺不错的选择的。

其他

除了 merge() 方法之外，我还看到了一些Java 8 中 map 相关的其他方法，比如 putIfAbsent、compute()、computeIfAbsent()、computeIfPresent，这些方法我们看名字应该就知道是什么意思了，故此处就不做过多介绍了，感兴趣的可以简单阅读一下源码（都还是挺易懂的），这里我们贴一下 compute()(Map.class) 的源码，其返回值是计算后得到的新值：

```
1 default V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
2     Objects.requireNonNull(remappingFunction);
3     V oldValue = this.get(key);
4 }
5     V newValue = remappingFunction.apply(key, oldValue);
6     if (newValue == null)
7     {
8         if (oldValue == null && !this.containsKey(key)) {
9             return null;
10 }
11     } else
12 }
```

```
12     this.remove(key)
13 ;
14     return null
15 ;
16 }
} else
{
    this.put(key, newValue);
    return newValue
;
}
}
```

总结

本文简单介绍了一下 Map.merge() 的方法，除此之外，Java 8 中的 HashMap 实现方法使用了 TreeNode 和 红黑树，在源码阅读上可能有一点难度，不过原理上还是相似的，compute() 同理。所以，源码肯定是要看的，不懂的地方多读多练自然就理解了。

链接

参考：

<https://www.jianshu.com/p/68e6b30410b0>

测试代码地址：

<https://github.com/lq920320/algorithm-java-test/blob/master/src/test/java/other/MapMethodsTest.java>

-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. 感受 Lambda 之美，推荐收藏，需要时查阅
2. 如何优雅的导出 Excel
3. 文艺互联网公司 vs 二逼互联网公司
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



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！

Java 8 尽管很香，你想过升级到 Java11 吗？会踩那些坑？

Xn346 Java后端 2月24日



微信搜一搜

Java后端

作者 | Xn346

链接 | blog.csdn.net/Xn346/article/details/104317752

目前最新JDK 11， Oracle会一直维护到2026年。

Java11的新特性

1、更新支持到Unicode 10编码

Unicode 10(version 10.0 of the Unicode Standard) , Unicode是一个不断在演进的行业标准, Java一直在与它保持一致兼容。

Java8已经更新了Unicode8.0-9.0， Java10更新后将达到16018个characters、18种blocks和10种scripts。

2、将Http Client作为JDK标准发布、

原来作为jdk补充的http类放在jdk.incubator.http包中，现在统一改到java.net.http包下，核心类有下面4个。

- HttpClient
- HttpRequest
- HttpResponse
- WebSocket

```
HttpClient client = HttpClient.newBuilder()
    .version(HttpClient.Version.HTTP_1_1)
    .connectTimeout(Duration.ofSeconds(3))
    .build();

HttpRequest request = HttpRequest.newBuilder().uri(URI.create("http://www.baidu.com")).build();
HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
System.out.println(response.statusCode()); // 200
System.out.println(response.body()); // 百度页面的html/
```

3、新增优化很多方法

- java.util.Collection增加新方法toArray(IntFunction)，集合转数组的不二之选。
- String增加lines\stripLeading\stripTrailing等，一般项目都有StringUtils类。
- java.io.InputStream增加构造方法
- java.nio包下面很大类扩展了方法Channels\XXXBuffer等

4、支持动态分配 Compiler Threads

JVM启动参数新增-XX:+UseDynamicNumberOfCompilerThreads，动态的控制编程线程的数量，原来的编译线程默认会启动大量造成cpu和memory浪费。

5、GC能力大幅提升

低功耗可扩展GC (ZGC) 模块是一个试验性的并发GC，在线程执行是ZGC会做一些重型回收工作，如string表清理等。执行周期在10ms内，处理heaps大小从MB到TB范围，目前只能支持linux和x64系统，除此外还有个处理memory分配的Epsilon GZ，有兴趣的可以自己研究。

6、堆分析能力提升：JVMTI

提供了一个低负载的堆分配采集分析程序:JVMTI，默认启动方案可以持续工作且不造成服务器压力，面向接口编程，能够收集活着和死去的对象信息。

7、Transport Layer Security 1.3更新

简称TLS1.3是网络传输层协议，需要注意的它不兼容历史版本而且官方承认有风险，希望后续能不断优化。

8、嵌套访问控制

嵌套是一种访问控制上下文，它允许多个class同属一个逻辑代码块，但是被编译成多个分散的class文件，它们访问彼此的私有成员无需通过编译器添加访问扩展方法。

例子：

```
/**  
 * @author: Owen Jia  
 * @time: 2019/11/7  
 */  
  
public class NestBasedTest {  
  
    public static class Nest1 {  
        private int varNest1;  
        public void f() throws Exception {  
            final Nest2 nest2 = new Nest2();  
            //这里没问题  
            nest2.varNest2 = 2;  
            final Field f2 = Nest2.class.getDeclaredField("varNest2");  
            //下面代码在java 8环境下会报错，但在java 11中是没问题的  
            f2.setInt(nest2, 1);  
            System.out.println(nest2.varNest2);  
        }  
    }  
  
    public static class Nest2 {  
        private int varNest2;  
    }  
  
    public static void main(String[] args) throws Exception {  
        new Nest1().f();  
    }  
}
```

这里要提一下Class类新增的方法：

// 获取宿主类。非嵌套类的宿主类是它本身。

```
public Class<?> getNestHost()
```

// 判断该类是否是某个类的嵌套类

```
public boolean isNestmateOf(Class<?> c)
```

// 返回某个类的嵌套类数组。第1个固定是宿主类，之后的是该宿主类的嵌套成员，但不保证顺序，同时也会包含自身

```
public boolean isNestmateOf(Class<?> c)
```

9、新增和优化诸多加密算法

对PKCS#1 v2.2内提供更多算法，如RSASSA-PSS签名算法。同时新增ChaCha20和Poly1305密码算法，通过Cipher.getInstance使用。还有Curve25519和Curve448被添加。AES128和256也支持了Kerberos 5 encryption。

10、本地参数支持Lambda

简单理解就是lambda表达式的变量申明可以用var。

```
lst.forEach((var x) -> {
    System.out.print(x);
});
```

11、单java文件加载运行

单个的*.java文件可以直接用java命令来执行，格式：java HelloWorld.java。

12、飞行记录器分析工具

Jvm启动参数：-XX:StartFlightRecording

Java11中将这款原来商用的工具集成到jdk标准中了，它是一种低开销的事件信息收集框架，用来对应用程序和JVM 进行故障检查、分析，收集应用程序、JVM 和 OS的数据并保存在单独的事件记录文件中，故障发生后，能够从事件记录文件中提取出有用信息对故障进行分析。

更多其他能力

还有很多其他更新就不一一介绍了，这些都是JDK标准包支持的基础能力，得感谢Oracle持续对JDK发布的支持。完整的jdk11变化清单可以去官网查看；

从11开始移除的模块清单

- Removal of com.sun.awt.AWTUtilities Class
- Removal of Lucida Fonts from Oracle JDK
- Removal of appletviewer Launcher
- Oracle JDK's javax.imageio JPEG Plugin No Longer Supports Images with alpha

- Removal of sun.misc.Unsafe.defineClass
- Removal of Thread.destroy() and Thread.stop(Throwable) Methods
- Removal of sun.nio.ch.disableSystemWideOverlappingFileLockCheck Property
- Removal of sun.locale.formatasdefault Property
- Removal of JVM-MANAGEMENT-MIB.mib
- Removal of SNMP Agent
- Remove the Java EE and CORBA Modules
- Removal of JavaFX from the Oracle JDK
- Removal of JMC from the Oracle JDK
- Removal of Java Deployment Technologies

更多请查看官网

升级建议（重要）

从Java 11后Oracle不再单独发布JRE和Server JRE了，并统一JDK名称为：Oracle JDK。

另外Java 11及之后的版本，将不会再发布对32位操作系统支持的版本。

The screenshot shows the Java SE Development Kit 11.0.5 download page. At the top, it says "You must accept the Oracle Technology Network License Agreement for Oracle Java SE to download this software." There are two radio buttons: "Accept License Agreement" (selected) and "Decline License Agreement". Below this is a table with columns "Product / File Description", "File Size", and "Download". The table lists files for Linux, macOS, Solaris SPARC, and Windows. Each row has a download link labeled "jdk-11.0.5_".

Product / File Description	File Size	Download
Linux	147.82 MB	jdk-11.0.5_linux-x64_bin.deb
Linux	154.47 MB	jdk-11.0.5_linux-x64_bin.rpm
Linux	171.62 MB	jdk-11.0.5_linux-x64_bin.tar.gz
macOS	166.73 MB	jdk-11.0.5_osx-x64_bin.dmg
macOS	167.06 MB	jdk-11.0.5_osx-x64_bin.tar.gz
Solaris SPARC	188.32 MB	jdk-11.0.5_solaris-sparcv9_bin.tar.gz
Windows	151.39 MB	jdk-11.0.5_windows-x64_bin.exe
Windows	171.47 MB	jdk-11.0.5_windows-x64_bin.zip

新旧项目不同策略

新启的Java项目建议直接从Oracle JDK 11开始搭建，千万不要犹豫，因为技术都是越新越强的。Java8就像晚期的大众，而Java11却是新兴的特斯拉。

历史的项目如果只是维护的话，干脆就放着运行不要动好了，等哪天决定重构了再考虑升级到Java11。因为最大的问题不是自己公司开发的Code不能迁移到高版本，而是项目中引入的第三方Jar，这个东西搞起来十分头疼。

JDK升级分析工具

升级最担心的就是被删除的模块！

推荐IBM公司Liberty团队提供了一个十分好用的检测Toolkit程序，可以扫描应用程序二进制文件（.war），发现的任何潜在的Java 11问题并生成Html报告。**绝对的大利器**，详细内容直接查看IBM官方介绍：Scanner Kit。

直接运行java -jar binaryAppScannerInstaller.jar，按步骤安装有个lisence声明和目录指定，默认目录名wamt。



参考文档中会有使用详细介绍，也可以参考下面测试例子（扫描很慢，要些耐心等）：

```
java -jar binaryAppScanner.jar Root.war --analyzeJavaSE --sourceJava=oracle8 --targetJava=java11 --output=
```

查看帮助命令：

```
java -jar binaryAppScanner.jar Root.war --help --all
```

```
正在处理: C:\Windows\system32\cmd.exe - java -jar binaryAppScanner.jar Root.war --all --output=./RootReport.html
输入产品文件的目录或保留空白以接受缺省值。
缺省目标目录是: F:\temp\scant
产品文件的目录目录? <.
将压缩文件至: F:\temp\scant
成功解压缩所有产品文件。
F:\temp\scant>ls
驱动器 F 中的内容 新加坡卷
卷的序列号是: 7483-PCW?
F:\temp\scant 的目录
2019/11/07 15:48 <DIR> .
2019/11/07 15:48 <DIR> ..
2019/11/07 15:48 18,994,431 binaryAppScanner.jar
2019/11/07 15:48 <DIR> bin
2019/11/07 15:33 <DIR> license
2019/11/07 15:33 835,265 MigrationToolkit_Application_Binaries_de_DE_19.0.0.4.pdf
2019/11/07 15:33 821,158 MigrationToolkit_Application_Binaries_en_US_19.0.0.4.pdf
2019/11/07 15:33 859,872 MigrationToolkit_Application_Binaries_es_ES_19.0.0.4.pdf
2019/11/07 15:33 988,283 MigrationToolkit_Application_Binaries_fr_FR_19.0.0.4.pdf
2019/11/07 15:33 977,223 MigrationToolkit_Application_Binaries_ja_JP_19.0.0.4.pdf
2019/11/07 15:33 926,086 MigrationToolkit_Application_Binaries_ko_KR_19.0.0.4.pdf
2019/11/07 15:33 856,804 MigrationToolkit_Application_Binaries_pt_BR_19.0.0.4.pdf
2019/11/07 15:33 954,300 MigrationToolkit_Application_Binaries_zh_CN_19.0.0.1.pdf
2019/11/07 15:33 1,014,818 MigrationToolkit_Application_Binaries_zh_TW_19.0.0.4.pdf
2019/11/06 10:53 59,812,173 ROOT.war
    11 个文件   86,579,685 字节
    3 个目录 78,864,549,888 可用字节
F:\temp\scant>java -jar binaryAppScanner.jar Root.war --all --output=./RootReport.html
正在处理 Root.war 应用程序。
INFO: 载入配置了, 扫描排除以下包: com.ibm.com.informix.com.exchange.com.microsoft.com.sybase.com.sun.java.java.net.oracle.org.sqlj...libjpa。使用 --includePaths 或 --excludePackages 选项以覆盖缺省扫描包。
正在扫描文件...
SEVERE: 无法处理 Root.war/WEB-INF/lib/joda-time-2.9.9.jar/org/joda/time/format/messages_en.properties 文件, 因为它为空。
```

报告结果如下：

JDK不同于OpenJDK

推荐Oracle JDK，因为更加稳定可靠。

- 只有Oracle JDK支持Solaris系统；
 - 只有Oracle JDK才支持msi这样的安装程序；
 - Oracle JDK版本将每三年发布一次，而OpenJDK版本每三个月发布一次；
 - OpenJDK 是一个参考模型并且是完全开源的，而Oracle JDK是 OpenJDK的一个实现，并不是完全开源的；
 - Oracle JDK 比 OpenJDK 更稳定。OpenJDK和Oracle JDK的代码几乎相同，但建议您选择Oracle JDK，因为它经过了彻底的测试和稳定修复；
 - 在响应性和JVM性能方面，Oracle JDK与OpenJDK相比提供了更好的性能；
 - Oracle JDK不会为即将发布的版本提供长期支持，用户每次都必须通过更新到最新版本获得支持来获取最新版本；
 - Oracle JDK根据二进制代码许可协议获得许可，而OpenJDK根据GPL v2许可获得许可。**Oracle公司很善于打官司**，所以这点很重要。

Differences Between Oracle JDK and Oracle's OpenJDK

Although we have stated the goal to have OpenJDK and Oracle JDK binaries be as close to each other as possible there remains, at least for JDK 11, several differences between the two options.

The current differences are:

- Only Oracle JDK offers Solaris.
- Oracle JDK offers "installers" (mai, rpm, deb, etc.) which not only place the JDK binaries in your system but also contain update rules and in some cases handle some common configurations like set common environmental variables (such as, JAVA_HOME in Windows) and establish file associations (such as, use java to launch .jar files). OpenJDK is offered only as compressed archive (.tar.gz or .zip).
- javac --release for release values 9 and 10 behave differently. Oracle JDK binaries include APIs that were not added to OpenJDK binaries such as javaFX, resource management, and (pre JDK 11 changes) JFR APIs.
- Usage Logging is only available in Oracle JDK.
- OpenJDK continues to throw an error and halt if the -XX:+UnlockCommercialFeatures flag is used. Oracle JDK no longer requires the flag and prints a warning but continues execution if used.
- Oracle JDK requires that third-party cryptographic providers be signed with a Java Cryptography Extension (JCE) Code Signing Certificate. OpenJDK continues allowing the use of unsigned third-party crypto providers.
- The output of java --version is different. Oracle JDK returns java and includes LTS. OpenJDK returns OpenJDK and does not include the Oracle-specific LTS identifier.
- Oracle JDK is released under the OTN License. OpenJDK is released under GPLv2+CP. License files included with each will therefore be different.
- Oracle JDK distributes FreeType under the FreeType license and OpenJDK does so under GPLv2. The contents of \legal\java_desktop\Freetype.md is therefore different.
- Oracle JDK has Java cup and steam icons and OpenJDK has Duke icons.
- Oracle JDK source code includes "ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms." Source code distributed with OpenJDK refers to the GPL license terms instead.

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。

- 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「focuseoncode」，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！

Java8 新特性之日期-时间 API

Liant Java后端 2019-11-23

作者 | Liant

链接 | cnblogs.com/liantdev/p/10108504.html

在Java8之前的版本中，我们处理时间类型常常使用的是java.util包下的Date类。

但使用Date类却有诸多的弊端，如：java.util.Date是非线程安全的，所有的日期类都是可变的；日期/时间类的定义并不一致，在java.util和java.sql的包下都含有Date类，在开发过程中极易出错；日期类并不提供国际化，没有时区支持。

为了解决以上问题，Java8在java.time包下提供了很多新的API，常用的类包括LocalDate、LocalTime、LocalDateTime，用以处理日期，时间，日期/时间等

LocalDate类

LocalDate是一个不可变类，在不考虑时区的情况下可以对日期（不包括时间）进行各种操作，它的默认格式是yyyy-MM-dd

获取当前日期以及年、月、日

代码示例：

```
//获取当前日期以及年、月、日
LocalDate localDate = LocalDate.now();
int year = localDate.getYear();
int month = localDate.getMonthValue();
int day = localDate.getDayOfMonth();
System.out.println("当前日期：" + localDate);
System.out.println("年：" + year + " 月：" + month + " 日：" + day);
```

运行结果：

```
当前日期：2018-12-12
年：2018 月：12 日：12
```

获取指定的日期

代码示例：

```
//获取指定的日期
LocalDate specifiedDay = LocalDate.of(2008, 8, 18);
System.out.println("指定日期：" + specifiedDay);
```

运行结果：

```
指定日期：2008-08-18
```

比较两个时间的先后顺序以及是否相等

代码示例：

```
LocalDate localDate = LocalDate.now();
LocalDate otherDate = LocalDate.of(2018, 11, 11);

//equals方法用于比较两个日期是否相等
if(localDate.equals(otherDate)) {
    System.out.println("localDate与otherDate相等! ");
} else {
    //isAfter和isBefore方法用于比较两个日期前后顺序
    if(localDate.isAfter(otherDate)) {
        System.out.println("localDate晚于otherDate! ");
    }
    if(localDate.isBefore(otherDate)) {
        System.out.println("localDate早于otherDate! ");
    }
}
```

运行结果：

```
localDate晚于otherDate!
```

对日期做加减运算

代码示例：

```
LocalDate localDate = LocalDate.now();
System.out.println("2年后日期: " + localDate.plusYears(2));
System.out.println("6月后日期: " + localDate.plusMonths(6));
System.out.println("3周后日期: " + localDate.plusWeeks(3));
System.out.println("15天后日期: " + localDate.plusDays(15));

System.out.println("2年前日期: " + localDate.minusYears(2));
System.out.println("6月前日期: " + localDate.minusMonths(6));
System.out.println("3周前日期: " + localDate.minusWeeks(3));
System.out.println("15天前日期: " + localDate.minusDays(15));
```

运行结果：

```
2年后日期: 2020-12-12
6月后日期: 2019-06-12
3周后日期: 2019-01-02
15天后日期: 2018-12-27

2年前日期: 2016-12-12
6月前日期: 2018-06-12
3周前日期: 2018-11-21
15天前日期: 2018-11-27
```

获取日期间隔的天数

代码示例：

```
//获取某年份的第N天的日期
LocalDate specialDay = LocalDate.ofYearDay(2018, 100);
System.out.println("2018年的第100天: " + specialDay);
//获取两个日期的间隔天数
long intervalDay = localDate.toEpochDay() - specialDay.toEpochDay();
System.out.println("间隔天数: " + intervalDay);
```

运行结果：

2018年的第100天：2018-04-10

间隔天数：246

LocalTime类

LocalTime与LocalDate一样，也是一个不可变的类，默认格式是hh:mm:ss.zzz，它提供了对时间的各种操作

获取当前时间以及自定义时间

代码示例：

```
//获取当前时间、时、分、秒以及自定义时间
LocalTime localTime = LocalTime.now();
int hour = localTime.getHour();
int minute = localTime.getMinute();
int second = localTime.getSecond();
System.out.println("当前时间：" + localTime);
System.out.println("时：" + hour + " 分：" + minute + " 秒：" + second);

//获取自定义时间
LocalTime specifiedTime = LocalTime.of(15, 30, 45);
System.out.println("自定义时间：" + specifiedTime);
```

运行结果：

```
当前时间：13:45:59.039
时：13 分：45 秒：59
自定义时间：15:30:45
```

比较两个时间的先后顺序

代码示例：

```
//equals方法比较两个时间是否相等
if(localTime.equals(specifiedTime)) {
    System.out.println("localTime与specifiedTime相等！");
} else {
    //isAfter、isBefore方法比较两个时间的先后顺序
    if(localTime.isAfter(specifiedTime)) {
        System.out.println("localTime晚于specifiedTime！");
    }
    if(localTime.isBefore(specifiedTime)) {
        System.out.println("localTime早于specifiedTime！");
    }
}
```

运行结果：

```
localTime早于specifiedTime!
```

对时间做加减运算

代码示例：

```
LocalTime localTime = LocalTime.now();
System.out.println("当前时间: " + localTime);
System.out.println("2小时后时间: " + localTime.plusHours(2));
System.out.println("30分钟后时间: " + localTime.plusMinutes(30));
System.out.println("500秒后时间: " + localTime.plusSeconds(500));

System.out.println("2小时前时间: " + localTime.minusHours(2));
System.out.println("30分钟前时间: " + localTime.minusMinutes(30));
System.out.println("500秒前时间: " + localTime.minusSeconds(500));
```

运行结果：

```
当前时间: 14:10:15.666
2小时后时间: 16:10:15.666
30分钟后时间: 14:40:15.666
500秒后时间: 14:18:35.666

2小时前时间: 12:10:15.666
30分钟前时间: 13:40:15.666
500秒前时间: 14:01:55.666
```

LocalDateTime类

LocalDateTime是一个不可变的日期-时间对象，它既包含了日期同时又含有时间，默认格式是yyyy-MM-ddTHH-mm:ss.zzz

获取当前日期时间以及自定义日期时间

示例代码：

```
//获取当前的日期时间
LocalDateTime localDateTime = LocalDateTime.now();
System.out.println("当前的日期时间: " + localDateTime);

//获取自定义的日期时间
LocalDateTime specifiedDateTime = LocalDateTime.of(LocalDate.now(), LocalTime.now());
System.out.println("自定义的日期时间: " + specifiedDateTime);
```

运行结果：

```
当前的日期时间: 2018-12-12T14:31:00.163
自定义的日期时间: 2018-12-12T14:31:00.164
```

转化为日期和时间

代码示例：

```
//转化为LocalDate和LocalTime
LocalDate localDate = localDateTime.toLocalDate();
LocalTime localTime = localDateTime.toLocalTime();
System.out.println("当前日期: " + localDate);
System.out.println("当前时间: " + localTime);
```

运行结果：

```
当前日期: 2018-12-12
当前时间: 14:31:00.163
```

推荐阅读

1. 我采访了一位 Pornhub 工程师,聊了这些纯纯的话题
2. 常见排序算法总结 - Java 实现
3. Java:如何更优雅的处理空值?
4. MySQL:数据库优化,可以看看这篇文章
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！

Java 8 的这个特性，用起来真的很爽！

Java后端 2019-09-26

以下文章来源于占小狼的博客，作者占小狼



占小狼的博客

Java进阶技术干货、实践分享，跟着狼哥一起学习JVM、性能调优，欢迎关注。

上一篇：计算机专业的学生也太太太太太惨了吧？

一直在写中间件相关的代码，提供SDK给业务方使用，但很多业务方还一直停留在1.7版本，迟迟不升级，为了兼容性，不敢在代码中使用Java8的一些新特性，比如Stream之类的，虽然不能用，但还是要学一下。

Stream 是什么

Stream 是 Java 8 中添加的一个新特性，它与 `java.io` 包里的 `InputStream` 和 `OutputStream` 是完全不同的概念。它借助于 Lambda 表达式，可以让你以一种声明的方式处理数据，可以极大提高 Java 程序员的生产力，让程序员写出高效率、干净、简洁的代码。

Stream Demo

直接上Demo，感受一下

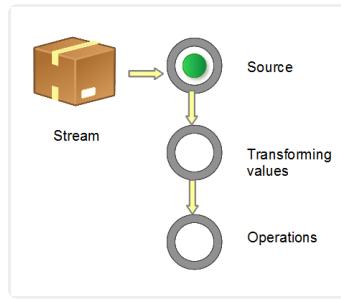
```
List<String> myList = Arrays.asList("a", "b", "c", "d", "e");
myList.stream()
    .filter(s -> s.startsWith("1"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Stream 如何工作

当使用一个流的时候，通常包括三个基本步骤：

- 获取一个数据源（source）
- 数据转换
- 执行操作获取想要的结果

每次转换原有 Stream 对象不改变，返回一个新的 Stream 对象（可以有多个转换），这就允许对其操作可以像链条一样排列，变成一个管道，如下图所示。



在Stream中，分为两种操作

- 中间操作
- 结束操作

中间操作返回 Stream，结束操作返回 void 或者非 Stream 结果，在 demo 中，`filter`、`map`、`sorted` 都算是中间操作，而 `forEach` 是一个

结束操作。

Stream 如何生成

创建Stream的方式很多，最常见的是从Collections，List 和 Set中生成

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");
Stream<String> stream = myList.stream();
```

在对象myList上调用方法 stream() 返回一个常规对象Stream。

也可以从一堆已知对象中生成。

```
Stream<String> stream = Stream.of("a1", "a2", "a3")
```

当然了，还有其它方式：

- Collection.stream()
- Collection.parallelStream()
- BufferedReader.lines()
- Files.walk()
- BitSet.stream()
- Random.ints()
- JarFile.stream()
-

常规操作

forEach

forEach 方法接收一个 Lambda 表达式，用来迭代流中的每个数据

```
Stream.of(1, 2, 3).forEach(System.out::println);
// 1
// 2
// 3
```

map

map 用于映射每个元素到对应的结果

```
Stream.of(1, 2, 3).map(i -> i * i).forEach(System.out::println);
// 1
// 4
// 9
```

filter

filter 用于通过设置的条件过滤出元素

```
Stream.of(1, 2, 3).filter(i -> i == 1).forEach(System.out::println);
// 1
```

limit

limit 用于获取指定数量的流

```
Stream.of(1, 2, 3, 4, 5).limit(2).forEach(System.out::println);
// 1
// 2
```

sorted

sorted 用于对流进行排序

```
Stream.of(4, 1, 5).sorted().forEach(System.out::println);
// 1
// 4
// 5
```

Match

有三个 match 方法，从语义上说：

- allMatch: Stream 中全部元素符合传入的 predicate，返回 true
- anyMatch: Stream 中只要有一个元素符合传入的 predicate，返回 true
- noneMatch: Stream 中没有一个元素符合传入的 predicate，返回 true

它们都不是要遍历全部元素才能返回结果。例如 allMatch 只要一个元素不满足条件，就 skip 剩下的所有元素，返回 false。

```
boolean result = Stream.of("a1", "a2", "a3").allMatch(i -> i.startsWith("a"));
System.out.println(result);
// true
```

reduce

reduce 方法根据指定的函数将元素序列累积到某个值。此方法有两个参数：

- 起始值
- 累加器函数。

如果有一个 List，希望得到所有这些元素和一些初始值的总和。

```
int result = Stream.of(1, 2, 3).reduce(20, (a,b) -> a + b);
System.out.println(result);
// 26
```

collect

Collectors类中提供了功能丰富的工具方法

- toList
- toSet
- toCollection
- toMap
- ...

而这些方法，都需要通过 collect 方法传入。

```
Set<Integer> result = Stream.of(1, 1, 2, 3).collect(Collectors.toSet());
System.out.println(result);
// [1, 2, 3]
```

collect 可以把Stream数据流转化为Collection对象，

for循环

除了常规的对象Stream，还有一些有特殊类型的Stream，用于处理基本数据类型int、long和double，它是IntStream、LongStream和DoubleStream。

比如可以使用IntStream.range()来代替常规的for循环。

```
IntStream.range(1, 4).forEach(System.out::println);
```

随机数

Random的ints方法可以返回一个随机数据流，比如返回1到100的10个随机数。

```
Random random = new Random();
random.ints(1, 100).limit(10).forEach(System.out::println);
```

大小写转化

```
List<String> output = wordList.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

Stream 特点

总之，Stream 的特性可以归纳为：

无存储

Stream并不是一种数据结构，它只是某种数据源的一个视图

安全性

对Stream的任何修改都不会修改背后的数据源，比如对stream执行过滤操作并不会删除被过滤的元素，而是会产生一个不包含被过滤元素的新Stream。

惰式执行

Stream上的操作并不会立即执行，只有等到用户真正需要结果的时候才会执行。

一次性

Stream只能被“消费”一次，一旦遍历过就会失效，就像容器的迭代器那样，想要再次遍历必须重新生成。

lambda

所有 Stream 的操作必须以 lambda 表达式为参数

- E N D -

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. Java后端优质文章整理
2. 计算机专业的学生也太太太太惨了吧？
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 

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！

Java 9 ← 2017, 2019 → Java 13 , 来看看Java两年来变化

Java后端 2019-11-27

以下文章来源于Hollis，作者Hollis



Hollis

Hollis, 一个对Coding有着独特追求的人。现任阿里巴巴技术专家，《程序员的三门课》合作者。本公众号专注分享Ja…

作者 | Hollis

来源 | Hollis (ID: hollischuang)

距离 2019 年结束，只剩下 35 天了。你做好准备迎接 2020 年了吗？

一到年底，人就特别容易陷入回忆和比较之中，比如说这几天，的对比挑战就火了！



这个话题登上了微博热搜榜，也刷爆了朋友圈，人们纷纷晒出自己2017和2019的照片对比。

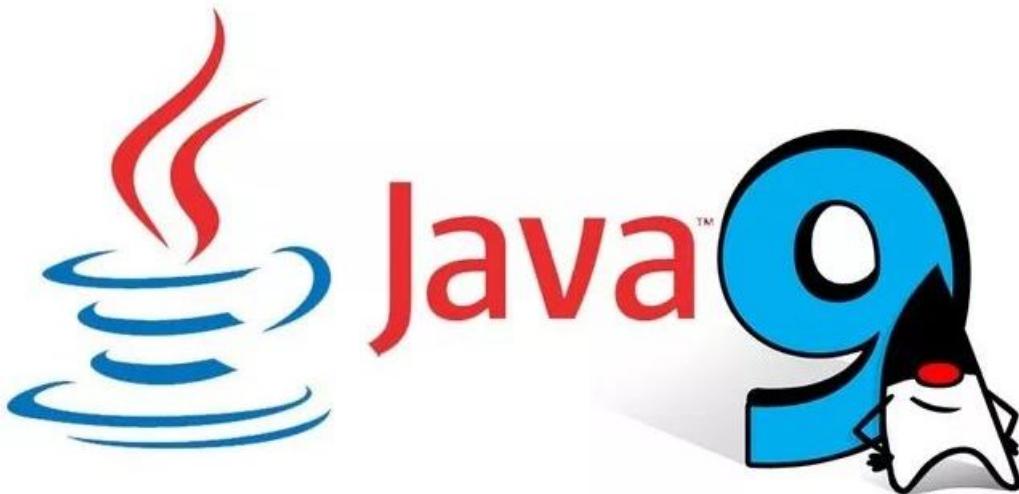
作为一个技术宅，我也做了一个对比：



2017年9月21日，Java 9 正式发布，并且在2017年8月，JCP执行委员 会提出将Java的发布频率改为每六个月一次，新的发布周期严格遵循时间点，将在每年的3月份和9月份发布。

1、Java 9

Java 9 于 2017 年 9 月 22 日正式发布，带来了很多新特性，其中最主要的变化是已经实现的模块化系统。



主要特性：

- 模块系统：模块是一个包的容器，Java 9 最大的变化之一是引入了模块系统（Jigsaw 项目）。
- HTTP 2 客户端：HTTP/2 标准是 HTTP 协议的最新版本，新的 HttpClient API 支持 WebSocket 和 HTTP2 流以及服务器推送特性。
- 改进的 Javadoc：Javadoc 现在支持在 API 文档中的进行搜索。另外，Javadoc 的输出现在符合兼容 HTML5 标准。
- 集合工厂方法：List, Set 和 Map 接口中，新的静态工厂方法可以创建这些集合的不可变实例。
- 私有接口方法：在接口中使用 private 私有方法。我们可以使用 private 访问修饰符在接口中编写私有方法。
- 改进的 Stream API：改进的 Stream API 添加了一些便利的方法，使流处理更容易，并使用收集器编写复杂的查询。
- 改进 try-with-resources：如果你已经有一个资源是 final 或等效于 final 变量，您可以在 try-with-resources 语句中使用该变量，而无需在 try-with-resources 语句中声明一个新变量。
- 改进的弃用注解 @Deprecated：注解 @Deprecated 可以标记 Java API 状态，可以表示被标记的 API 将会被移除，或者已经破坏。
- 改进 Optional 类：java.util.Optional 添加了很多新的有用方法，Optional 可以直接转为 stream。
- 响应式流（Reactive Streams）API：Java 9 中引入了新的响应式流 API 来支持 Java 9 中的响应式编程。

2、Java 10

Java 10 于 2018 年 3 月 21 日正式发布，这是作为当今使用最广泛的编程语言之一的 Java 语言的第十个大版本。



主要特性：

- 局部变量类型推断 (Local-Variable Type Inference)：使用var关键字进行变量声明，可以进行变量类型的推断。
- G1的并行Full GC (Parallel Full GC for G1)： G1 垃圾回收器的并行完整垃圾回收，实现并行性来改善最坏情况下的延迟。
- 应用程序类数据共享 (Application Class-Data Sharing)： 应用程序类数据 (AppCDS) 共享，通过跨进程共享通用类元数据来减少内存占用空间，和减少启动时间。
- ThreadLocal握手机制 (Thread-Local Handshakes)：在不进入到全局 JVM 安全点 (Safepoint) 的情况下，对线程执行回调。优化可以只停止单个线程，而不是停全部线程或一个都不停。

3、Java 11

Java 11 于 2018 年9 月25 日正式发布，可在生产环境中使用！ 这是自 Java 8 后的首个长期支持版本，将支持到2026 年。



主要特性：

- ZGC, 可扩展的低延迟垃圾收集器 (ZGC: A Scalable Low-Latency Garbage Collector)： ZGC是一款号称可以保证每次GC的停顿时间不超过10MS的垃圾回收器，并且和当前的默认垃圾回收起G1相比，吞吐量下降不超过15%。
- Epsilon: 什么事也不做的垃圾回收器 (Epsilon: A No-Op Garbage Collector)： 这是一款不做垃圾回收的垃圾回收器。这个垃圾回收器看起来并没什么用，主要可以用来进行性能测试、内存压力测试等，Epsilon GC可以作为度量其他垃圾回收器性能的对照组。
- 增强var用法 (Local-Variable Syntax for Lambda Parameters)： 在Java 11中，var可以用来作为Lambda表达式的局部变量声明。

4、Java 12

Java 12于 2019 年3 月 19 日正式发布，自Java 11 这一Long Term Support 版本发布之后半年的又一次版本更新



主要特性：

- 低暂停时间的 GC (Shenandoah: A Low-Pause-Time Garbage Collector) (Experimental): 新增 Shenandoah 算法，通过与正在运行的 Java 线程同时进行 evacuation 工作来减少 GC 暂停时间。
- Switch 表达式(Switch Expressions): 扩展了 switch 语句,使其不仅可以作为语句(statement),还可以作为表达式(expression)
- 可中止的 G1 Mixed GC (Abortable Mixed Collections for G1) G1 及时返回未使用的已分配内存(Promptly Return Unused Committed Memory from G1): 如果 G1 Mixed GC 存在超出暂停目标的可能性,则使其可中止。

5、Java 13

Java 13 于 2019 年 9 月 17 日正式发布，这一版本中引入了文本块等功能。



主要特性：

- 扩展应用程序类-数据共享(Dynamic CDS Archives): 以允许在 Java 应用程序执行结束时动态归档类。归档类将包括默认的基础层 CDS (class data-sharing) 存档中不存在的所有已加载的应用程序类和库类。
- 增强 ZGC 以将未使用的堆内存返回给操作系统(ZGC: Uncommit Unused Memory): ZGC 可以将未使用的堆内存返回给操作系统
- 可在生产环境中使用的 switch 表达式(Switch Expressions): 在switch块中引入了yield语句，用于返回值。
- 将文本块添加到 Java 语言(Text Blocks): 引入多行字符串文字，在其中可以放置多行的字符串，不需要进行任何转义。

哪些特性改变你写代码的方式？

1、本地变量类型推断

在以前的版本中，我们想定义局部变量时。我们需要在赋值的左侧提供显式类型，并在赋值的右边提供实现类型，如下面的片段所示:

```
MyObject value = new MyObject();
List list = new ArrayList();
```

在Java 10中，你可以这样定义对象：

```
var value = new MyObject();
var list = new ArrayList();
```

正如你所看到的，本地变量类型推断将引入“var”关键字，而不需要显式的规范变量的类型。

2、switch表达式

在JDK 12中引入了Switch表达式作为预览特性。在Java 13中又修改了这个特性，引入了yield语句，用于返回值。这意味着，switch表达式(返回值)应该使用yield，switch语句(不返回值)应该使用break。

在以前，我们想要在switch中返回内容，还是比较麻烦的，一般语法如下：

```
int i;
switch (x) {
    case "1":
        i=1;
        break;
    case "2":
        i=2;
        break;
    default:
        i=x.length();
        break;
}
```

在JDK13中使用以下语法：

```
int i = switch (x) {
    case "1" -> 1;
    case "2" -> 2;

    default -> {
        int len = args[1].length();
        yield len;
    }
};
```

或者

```
int i = switch (x) {
    case "1": yield 1;
    case "2": yield 2;

    default: {
        int len = args[1].length();
        yield len;
    }
};
```

3、文本块支持

text block，文本块，是一个多行字符串文字，它避免了对大多数转义序列的需要，以可预测的方式自动格式化字符串，并在需要时让开发人员控制格式。

我们以前从外部copy一段文本串到Java中，会被自动转义，如有一段以下字符串：

```
<html>
<body>
  <p>Hello, world</p>
</body>
</html>
```

将其复制到Java的字符串中，会展示成以下内容：

```
"<html>\n" +
"  <body>\n" +
"    <p>Hello, world</p>\n" +
"  </body>\n" +
"</html>\n";
```

即被自动进行了转义，这样的字符串看起来不是很直观，在JDK 13中，就可以使用以下语法了：

```
"""
<html>
<body>
  <p>Hello, world</p>
</body>
</html>
""",
```

使用“”“作为文本块的开始符合结束符，在其中就可以放置多行的字符串，不需要进行任何转义。看起来就十分清爽了。

后话

从2017到2019，不果短短两年时间，Java就发布了5个版本。每个版本都会有很多特性出来，这些特性中虽然没有像Java 8那样提供函数式编程这样的重大改变，但是也提供了switch表达式和block text等大家期待已久特性。

2年过去了，在Java学习的道路上，你进步了吗？欢迎谈谈你这两年的变化。

【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. 一文掌握 Redis 常用知识点 - 图文结合
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！

Java NIO：Buffer、Channel 和 Selector

Javadoop Java后端 2019-11-09

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | HongJie

链接 | javadoop.com/post/nio-and-aio

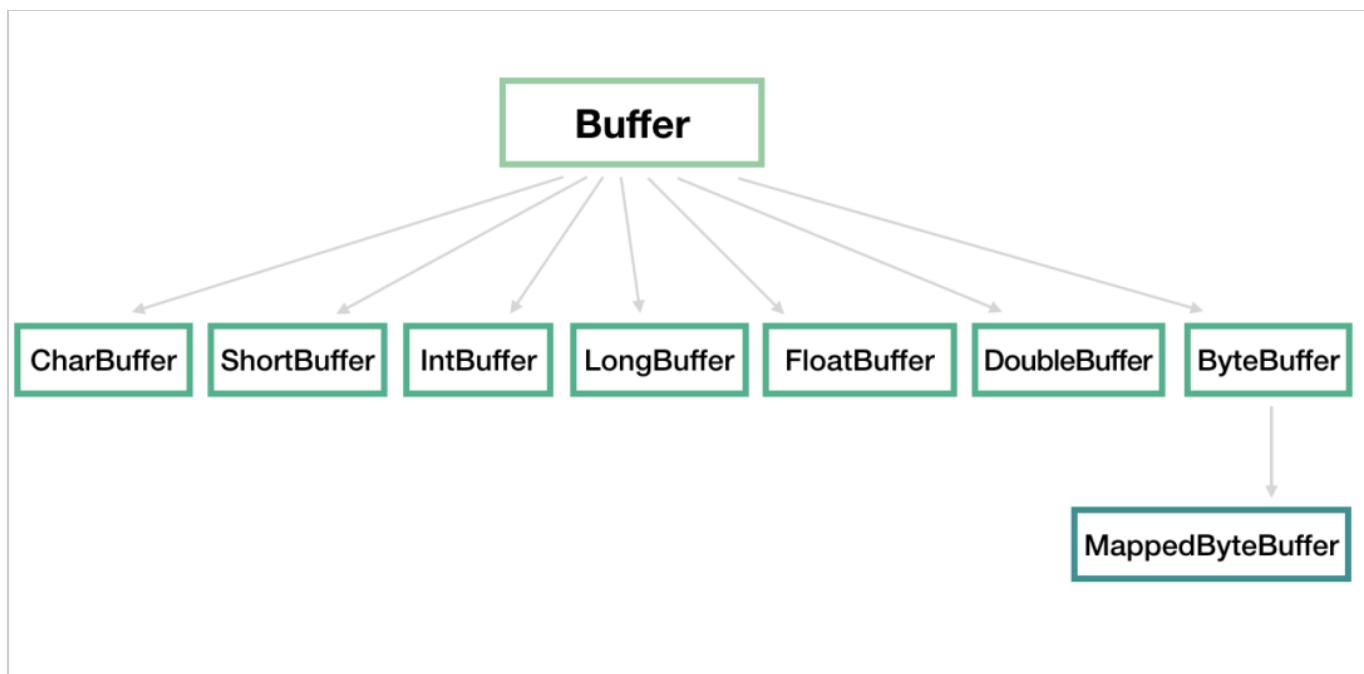
本文将介绍 Java NIO 中三大组件 **Buffer**、**Channel**、**Selector** 的使用。

本来要一起介绍**非阻塞 IO** 和 **JDK7 的异步 IO** 的，不过因为之前的文章真的太长了，有点影响读者阅读，所以这里将它们放到另一篇文章中进行介绍。

Buffer

一个 Buffer 本质上是内存中的一块，我们可以将数据写入这块内存，之后从这块内存获取数据。

java.nio 定义了以下几个 Buffer 的实现，这个图读者应该也在不少地方见过了吧。



其实核心是最后的 **ByteBuffer**，前面的一大串类只是包装了一下它而已，我们使用最多的通常也是 ByteBuffer。

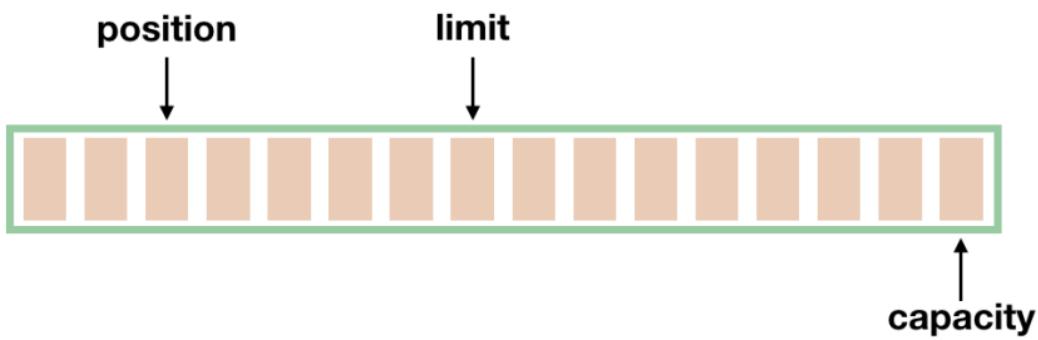
我们应该将 Buffer 理解为一个数组，IntBuffer、CharBuffer、DoubleBuffer 等分别对应 int[]、char[]、double[] 等。

MappedByteBuffer 用于实现内存映射文件，也不是本文关注的重点。

我觉得操作 Buffer 和操作数组、类集差不多，只不过大部分时候我们都把它放到了 NIO 的场景里面来使用而已。下面介绍 Buffer 中的几个重要属性和几个重要方法。

position、limit、capacity

就像数组有数组容量，每次访问元素要指定下标，Buffer 中也有几个重要属性：position、limit、capacity。



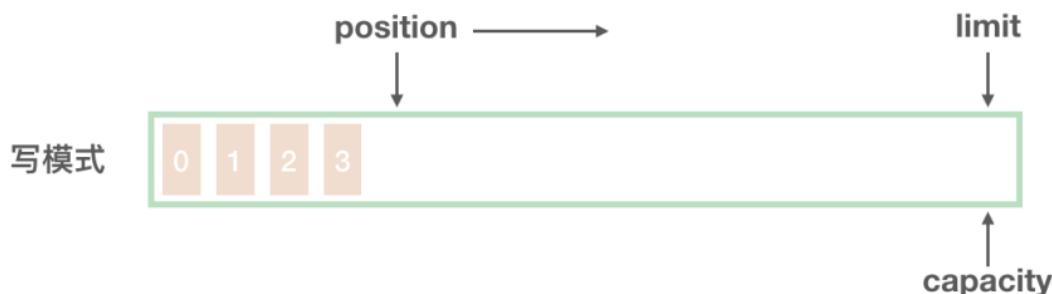
最好理解的当然是 capacity，它代表这个缓冲区的容量，一旦设定就不可以更改。比如 capacity 为 1024 的 IntBuffer，代表其一次可以存放 1024 个 int 类型的值。一旦 Buffer 的容量达到 capacity，需要清空 Buffer，才能重新写入值。

position 和 limit 是变化的，我们分别看下读和写操作下，它们是如何变化的。

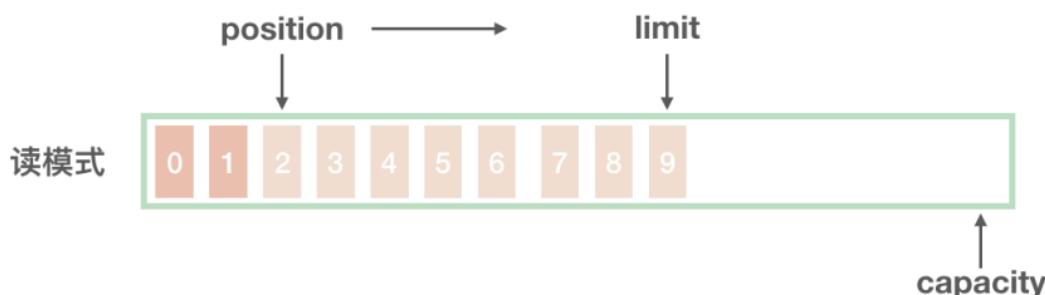
position 的初始值是 0，每往 Buffer 中写入一个值，position 就自动加 1，代表下一次的写入位置。读操作的时候也是类似的，每读一个值，position 就自动加 1。

从写操作模式到读操作模式切换的时候（**flip**），position 都会归零，这样就可以从头开始读写了。

Limit：写操作模式下，limit 代表的是最大能写入的数据，这个时候 limit 等于 capacity。写结束后，切换到读模式，此时的 limit 等于 Buffer 中实际的数据大小，因为 Buffer 不一定被写满了。



写模式下，limit 等于 capacity，此时 position 为 4



读模式下，limit 等于 Buffer 的实际数据大小，此时 limit 为 10

初始化 Buffer

每个 Buffer 实现类都提供了一个静态方法 `allocate(int capacity)` 帮助我们快速实例化一个 Buffer。如：

```
ByteBuffer byteBuf = ByteBuffer.allocate(1024);
IntBuffer intBuf = IntBuffer.allocate(1024);
LongBuffer longBuf = LongBuffer.allocate(1024);
// ...
```

另外，我们经常使用 wrap 方法来初始化一个 Buffer。

```
public static ByteBuffer wrap(byte[] array) {  
    ...  
}
```

填充 Buffer

各个 Buffer 类都提供了一些 put 方法用于将数据填充到 Buffer 中，如 ByteBuffer 中的几个 put 方法：

```
// 填充一个 byte 值  
public abstract ByteBuffer put(byte b);  
  
// 在指定位置填充一个 int 值  
public abstract ByteBuffer put(int index, byte b);  
  
// 将一个数组中的值填充进去  
public final ByteBuffer put(byte[] src) {...}  
public ByteBuffer put(byte[] src, int offset, int length) {...}
```

上述这些方法需要自己控制 Buffer 大小，不能超过 capacity，超过会抛 java.nio.BufferOverflowException 异常。

对于 Buffer 来说，另一个常见的操作中就是，我们要将来自 Channel 的数据填充到 Buffer 中，在系统层面上，这个操作我们称为**读操作**，因为数据是从外部（文件或网络等）读到内存中。

```
int num = channel.read(buf);
```

上述方法会返回从 Channel 中读入到 Buffer 的数据大小。

提取 Buffer 中的值

前面介绍了写操作，每写入一个值，position 的值都需要加 1，所以 position 最后会指向最后一次写入的位置的后面一个，如果 Buffer 写满了，那么 position 等于 capacity (position 从 0 开始)。

如果要读 Buffer 中的值，需要切换模式，从写入模式切换到读出模式。注意，通常在说 NIO 的读操作的时候，我们说的是从 Channel 中读数据到 Buffer 中，对应的是对 Buffer 的写入操作，初学者需要理清楚这个。

调用 Buffer 的**flip()** 方法，可以从写入模式切换到读取模式。其实这个方法也就是设置了一下 position 和 limit 值罢了。

```
public final Buffer flip() {  
    limit = position; // 将 limit 设置为实际写入的数据数量  
    position = 0; // 重置 position 为 0  
    mark = -1; // mark 之后再说  
    return this;  
}
```

对应写入操作的一系列 put 方法，读操作提供了一系列的 get 方法：

```
// 根据 position 来获取数据
public abstract byte get();
// 获取指定位置的数据
public abstract byte get(int index);
// 将 Buffer 中的数据写入到数组中
public ByteBuffer get(byte[] dst)
```

附一个经常使用的方法：

```
new String(buffer.array()).trim();
```

当然了，除了将数据从 Buffer 取出来使用，更常见的操作是将我们写入的数据传输到 Channel 中，如通过 FileChannel 将数据写入到文件中，通过 SocketChannel 将数据写入网络发送到远程机器等。对应的，这种操作，我们称之为**写操作**。

```
int num = channel.write(buf);
```

mark() & reset()

除了 position、limit、capacity 这三个基本的属性外，还有一个常用的属性就是 mark。

mark 用于临时保存 position 的值，每次调用 mark() 方法都会将 mark 设值为当前的 position，便于后续需要的时候使用。

```
public final Buffer mark() {
    mark = position;
    return this;
}
```

那到底什么时候用呢？考虑以下场景，我们在 position 为 5 的时候，先 mark() 一下，然后继续往下读，读到第 10 的时候，我想重新回到 position 为 5 的地方重新来一遍，那只要调一下 reset() 方法，position 就回到 5 了。

```
public final Buffer reset() {
    int m = mark;
    if (m < 0)
        throw new InvalidMarkException();
    position = m;
    return this;
}
```

rewind() & clear() & compact()

rewind()：会重置 position 为 0，通常用于重新从头读写 Buffer。

```
public final Buffer rewind() {
    position = 0;
    mark = -1;
    return this;
}
```

clear()：有点重置 Buffer 的意思，相当于重新实例化了一样。

通常，我们会先填充 Buffer，然后从 Buffer 读取数据，之后我们再重新往里填充新的数据，我们一般在重新填充之前先调用 clear()。

```
public final Buffer clear() {  
    position = 0;  
    limit = capacity;  
    mark = -1;  
    return this;  
}
```

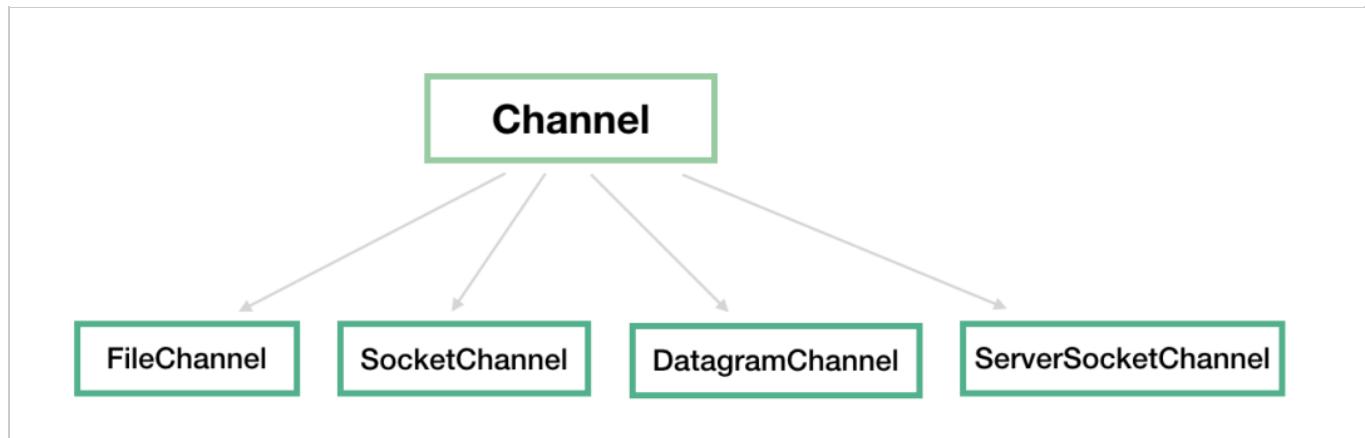
compact(): 和 clear() 一样的是，它们都是在准备往 Buffer 填充新的数据之前调用。

前面说的 clear() 方法会重置几个属性，但是我们要看到，clear() 方法并不会将 Buffer 中的数据清空，只不过后续的写入会覆盖掉原来的数据，也就相当于清空了数据了。

而 compact() 方法有点不一样，调用这个方法以后，会先处理还没有读取的数据，也就是 position 到 limit 之间的数据（还没有读过的数据），先将这些数据移到左边，然后在这个基础上再开始写入。很明显，此时 limit 还是等于 capacity，position 指向原来数据的右边。

Channel

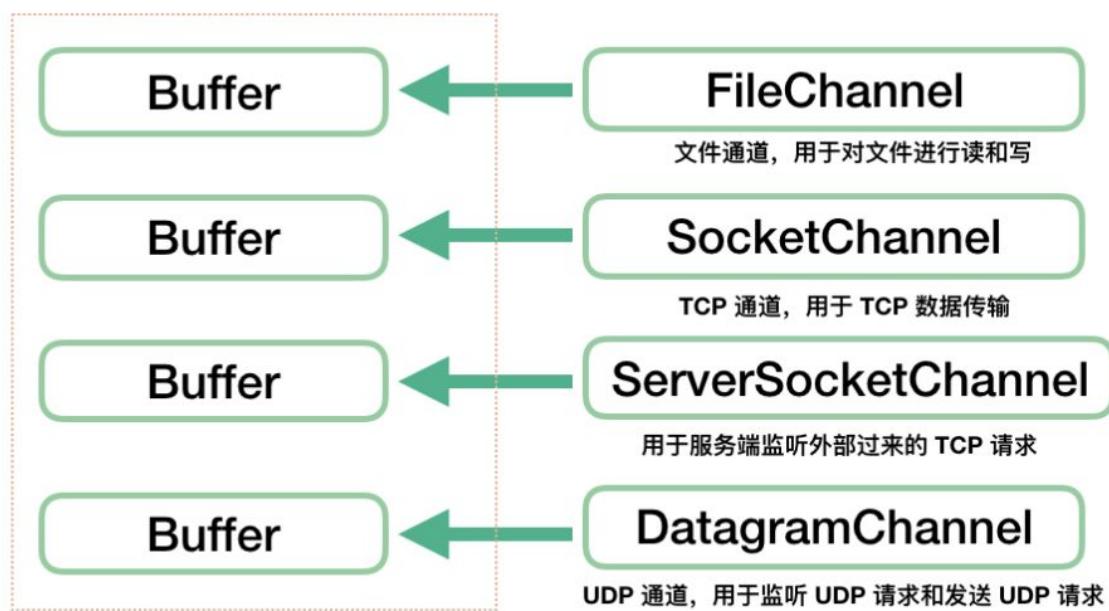
所有的 NIO 操作始于通道，通道是数据来源或数据写入的目的地，主要地，我们将关心 java.nio 包中实现的以下几个 Channel:



- FileChannel: 文件通道，用于文件的读和写
- DatagramChannel: 用于 UDP 连接的接收和发送
- SocketChannel: 把它理解为 TCP 连接通道，简单理解就是 TCP 客户端
- ServerSocketChannel: TCP 对应的服务端，用于监听某个端口进来的请求

这里不是很理解这些也没关系，后面介绍了代码之后就清晰了。还有，我们最应该关注，也是后面将会重点介绍的是 **SocketChannel** 和 **ServerSocketChannel**。

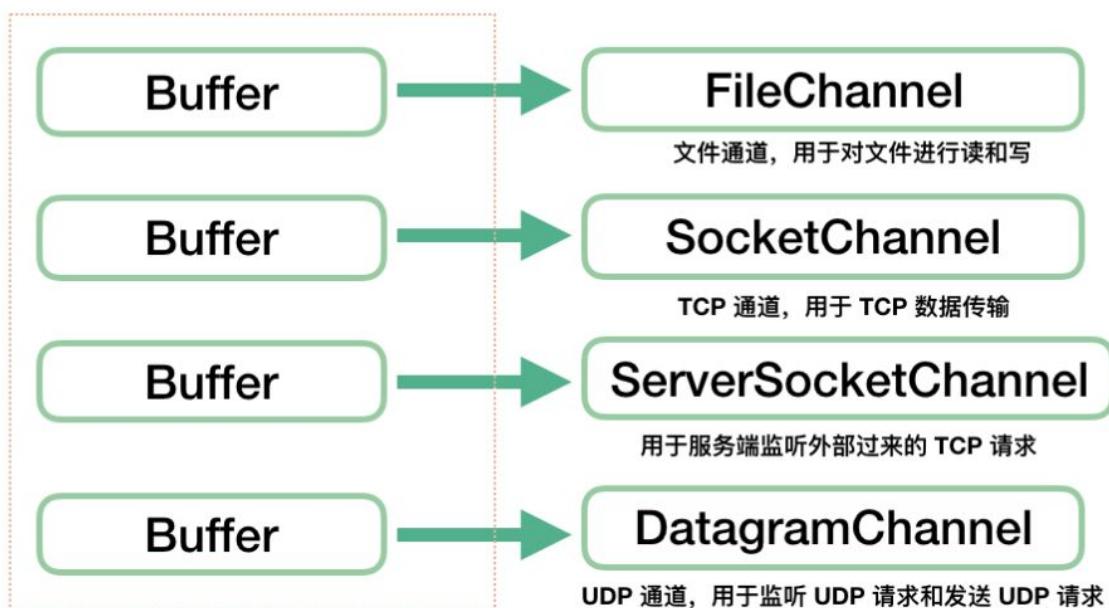
Channel 经常翻译为通道，类似 IO 中的流，用于读取和写入。它与前面介绍的 Buffer 打交道，读操作的时候将 Channel 中的数据填充到 Buffer 中，而写操作时将 Buffer 中的数据写入到 Channel 中。



读操作: 就是将数据从 Channel 读到 Buffer 中，进行后续处理。

方法: `channel.read(buffer);`

<https://www.javadoop.com>



写操作: 就是将数据从 Buffer 写入到 Channel 中

方法: `channel.write(buffer);`

<https://www.javadoop.com>

至少读者应该记住一点，这两个方法都是 channel 实例的方法。

FileChannel

我想文件操作对于大家来说应该是最熟悉的，不过我们在说 NIO 的时候，其实 **FileChannel** 并不是关注的重点。而且后面我们说非阻塞的时候会看到，**FileChannel** 是不支持非阻塞的。

这里算是简单介绍下常用的操作吧，感兴趣的读者瞄一眼就是了。

初始化：

```
FileInputStream inputStream = new FileInputStream(new File("/data.txt"));
FileChannel fileChannel = inputStream.getChannel();
```

当然了，我们也可以从 RandomAccessFile#getChannel 来得到 FileChannel。

读取文件内容：

```
ByteBuffer buffer = ByteBuffer.allocate(1024);

int num = fileChannel.read(buffer);
```

前面我们也说了，所有的 Channel 都是和 Buffer 打交道的。

写入文件内容：

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
buffer.put("随机写入一些内容到 Buffer 中".getBytes());
// Buffer 切换为读模式
buffer.flip();
while(buffer.hasRemaining()) {
    // 将 Buffer 中的内容写入文件
    fileChannel.write(buffer);
}
```

SocketChannel

我们前面说了，我们可以将 SocketChannel 理解成一个 TCP 客户端。虽然这么理解有点狭隘，因为我们在介绍 ServerSocketChannel 的时候会看到另一种使用方式。

打开一个 TCP 连接：

```
SocketChannel socketChannel = SocketChannel.open(new InetSocketAddress("https://www.javadoop.com", 80));
```

当然了，上面的这行代码等价于下面的两行：

```
// 打开一个通道
SocketChannel socketChannel = SocketChannel.open();
// 发起连接
socketChannel.connect(new InetSocketAddress("https://www.javadoop.com", 80));
```

SocketChannel 的读写和 FileChannel 没什么区别，就是操作缓冲区。

```
// 读取数据  
socketChannel.read(buffer);  
  
// 写入数据到网络连接中  
while(buffer.hasRemaining()) {  
    socketChannel.write(buffer);  
}  
}
```

不要在这里停留太久，先继续往下走。

ServerSocketChannel

之前说 SocketChannel 是 TCP 客户端，这里说的 ServerSocketChannel 就是对应的服务端。

ServerSocketChannel 用于监听机器端口，管理从这个端口进来的 TCP 连接。

```
// 实例化  
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();  
// 监听 8080 端口  
serverSocketChannel.socket().bind(new InetSocketAddress(8080));  
  
while (true) {  
    // 一旦有一个 TCP 连接进来，就对应创建一个 SocketChannel 进行处理  
    SocketChannel socketChannel = serverSocketChannel.accept();  
}
```

这里我们可以看到 SocketChannel 的第二个实例化方式

到这里，我们应该能理解 SocketChannel 了，它不仅仅是 TCP 客户端，它代表的是一个网络通道，可读可写。

ServerSocketChannel 不和 Buffer 打交道了，因为它并不实际处理数据，它一旦接收到请求后，实例化 SocketChannel，之后在这个连接通道上的数据传递它就不管了，因为它需要继续监听端口，等待下一个连接。

DatagramChannel

UDP 和 TCP 不一样，DatagramChannel 一个类处理了服务端和客户端。

科普一下，UDP 是面向无连接的，不需要和对方握手，不需要通知对方，就可以直接将数据包投出去，至于能不能送达，它是不知道的

监听端口：

```
DatagramChannel channel = DatagramChannel.open();  
channel.socket().bind(new InetSocketAddress(9090));
```

```
ByteBuffer buf = ByteBuffer.allocate(48);
```

```
channel.receive(buf);
```

发送数据：

```
String newData = "New String to write to file..."  
    + System.currentTimeMillis();  
  
ByteBuffer buf = ByteBuffer.allocate(48);  
buf.put(newData.getBytes());  
buf.flip();  
  
int bytesSent = channel.send(buf, new InetSocketAddress("jenkov.com", 80));
```

Selector

NIO 三大组件就剩 Selector 了，Selector 建立在非阻塞的基础之上，大家经常听到的**多路复用**在 Java 世界中指的就是它，用于实现一个线程管理多个 Channel。

读者在这一节不能消化 Selector 也没关系，因为后续在介绍非阻塞 IO 的时候还得说到这个，这里先介绍一些基本的接口操作。

1. 首先，我们开启一个 Selector。你们爱翻译成**选择器**也好，**多路复用器**也好。

```
Selector selector = Selector.open();
```

2. 将 Channel 注册到 Selector 上。前面我们说了，Selector 建立在非阻塞模式之上，所以注册到 Selector 的 Channel 必须要支持非阻塞模式，**FileChannel 不支持非阻塞**，我们这里讨论最常见的 SocketChannel 和 ServerSocketChannel。

```
// 将通道设置为非阻塞模式，因为默认都是阻塞模式的  
channel.configureBlocking(false);  
// 注册  
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

register 方法的第二个 int 型参数（使用二进制的标记位）用于表明需要监听哪些感兴趣的事件，共以下四种事件：

我们可以同时监听一个 Channel 中的发生的多个事件，比如我们要监听 ACCEPT 和 READ 事件，那么指定参数为二进制的 00010001 即十进制数值 17 即可。

注册方法返回值是 **SelectionKey** 实例，它包含了 Channel 和 Selector 信息，也包括了一个叫做 Interest Set 的信息，即我们设置的我们感兴趣的正在监听的事件集合。

- SelectionKey.OP_READ

 对应 00000001，通道中有数据可以进行读取

- SelectionKey.OP_WRITE

 对应 00000100，可以往通道中写入数据

- SelectionKey.OP_CONNECT

 对应 00001000，成功建立 TCP 连接

- SelectionKey.OP_ACCEPT

 对应 00010000，接受 TCP 连接

3. 调用 select() 方法获取通道信息。用于判断是否有我们感兴趣的事件已经发生了。

Selector 的操作就是以上 3 步，这里来一个简单的示例，大家看一下就好了。之后在介绍非阻塞 IO 的时候，会演示一份可执行的示例代码。

```

Selector selector = Selector.open();

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, SelectionKey.OP_READ);

while(true) {
    // 判断是否有事件准备好
    int readyChannels = selector.select();
    if(readyChannels == 0) continue;

    // 遍历
    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
    while(keyIterator.hasNext()) {
        SelectionKey key = keyIterator.next();

        if(key.isAcceptable()) {
            // a connection was accepted by a ServerSocketChannel.

        } else if (key.isConnectable()) {
            // a connection was established with a remote server.

        } else if (key.isReadable()) {
            // a channel is ready for reading

        } else if (key.isWritable()) {
            // a channel is ready for writing
        }

        keyIterator.remove();
    }
}

```

对于 Selector，我们还需要非常熟悉以下几个方法：

1. **select()**

调用此方法，会将上次 **select** 之后的准备好的 channel 对应的 SelectionKey 复制到 selected set 中。如果没有任何通道准备好，这个方法会阻塞，直到至少有一个通道准备好。

2. **selectNow()**

功能和 select 一样，区别在于如果没有准备好的通道，那么此方法会立即返回 0。

3. **select(long timeout)**

看了前面两个，这个应该很好理解了，如果没有通道准备好，此方法会等待一会

4. **wakeup()**

这个方法是用来唤醒等待在 select() 和 select(timeout) 上的线程的。如果 wakeup() 先被调用，此时没有线程在 select 上阻塞，那么之后的一个 select() 或 select(timeout) 会立即返回，而不会阻塞，当然，它只会作用一次。

小结

到此为止，介绍了 Buffer、Channel 和 Selector 的常见接口。

Buffer 和数组差不多，它有 position、limit、capacity 几个重要属性。put() 一下数据、flip() 切换到读模式、然后用 get() 获取数据、clear() 一下清空数据、重新回到 put() 写入数据。

Channel 基本上只和 Buffer 打交道，最重要的接口就是 channel.read(buffer) 和 channel.write(buffer)。

Selector 用于实现非阻塞 IO，这里仅仅介绍接口使用，后续请关注非阻塞 IO 的介绍。

- END -



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！

Java Web 前端到后台常用框架介绍

Java后端 2019-09-11

1 SpringMVC

参考博文：

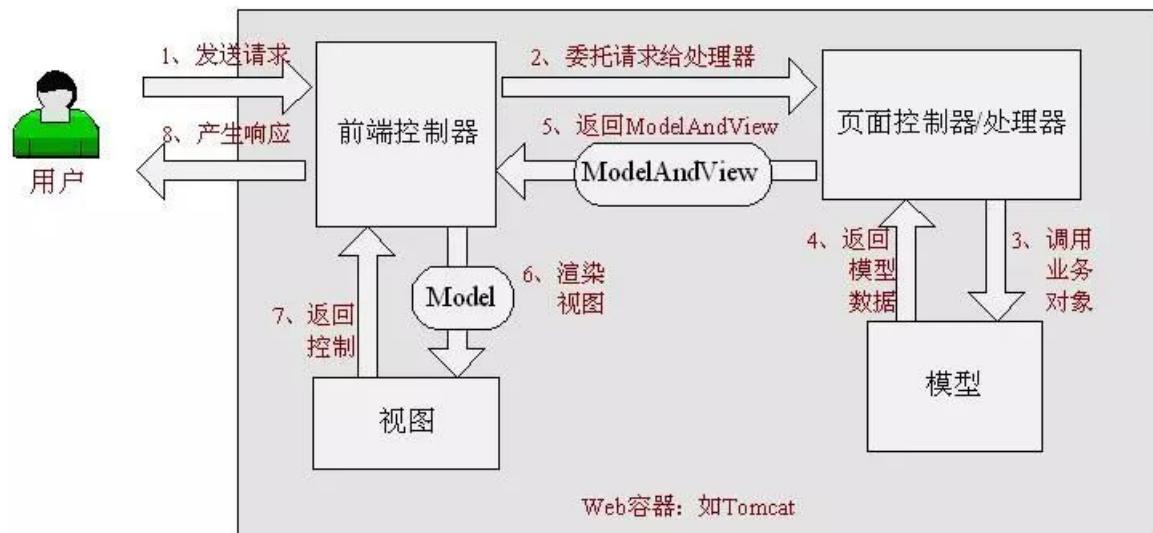
<http://blog.csdn.net/evankaka/article/details/45501811>

基本原理流程，3个线程以及之间的关联；Spring Web MVC是一种基于Java的实现了Web MVC设计模式的请求驱动类型的轻量级Web框架，即使用了MVC架构模式的思想，将web层进行职责解耦，基于请求驱动指的就是使用请求-响应模型，框架的目的就是帮助我们简化开发，Spring Web MVC也是要简化我们日常Web开发的。

- 1 模型（Model）封装了应用程序的数据和一般他们会组成的POJO。
- 2 视图（View）是负责呈现模型数据和一般它生成的HTML输出，客户端的浏览器能够解释。
- 3 控制器（Controller）负责处理用户的请求，并建立适当的模型，并把它传递给视图渲染。

Spring的web模型 - 视图 - 控制器（MVC）框架是围绕着处理所有的HTTP请求和响应的DispatcherServlet的设计。

Spring Web MVC处理请求的流程



具体执行步骤如下：

- 1.首先用户发送请求——>前端控制器，前端控制器根据请求信息(如URL)来决定选择哪一个页面控制器进行处理并把请求委托给它，即以前的控制器的控制逻辑部分；图2-1中的1、2步骤；
- 2.页面控制器接收到请求后，进行功能处理，首先需要收集和绑定请求参数到一个对象，这个对象在Spring Web MVC中叫命令对象，并进行验证，然后将命令对象委托给业务对象进行处理；处理完毕后返回一个ModelAndView(模型数据和逻辑视图名)；图2-1中的3、4、5步骤；
- 3.前端控制器收回控制权，然后根据返回的逻辑视图名，选择相应的视图进行渲染，并把模型数据传入以便视图渲染；图2-1中的步骤6、7；
- 4.前端控制器再次收回控制权，将响应返回给用户，图2-1中的步骤8；至此整个结束。

2 Spring

参考博文：

<http://blog.csdn.net/cainiaowys/article/details/7107925>

2.1 IOC容器

参考博文：

<http://www.cnblogs.com/linjiqin/archive/2013/11/04/3407126.html>

IOC容器就是具有依赖注入功能的容器，IOC容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。应用程序无需直接在代码中new相关的对象，应用程序由IOC容器进行组装。在Spring中BeanFactory是IOC容器的实际代表者。

2.2 AOP

参考博文：

<http://blog.csdn.net/moreevan/article/details/11977115>

简单地说，就是将那些与业务无关，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，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP代表的是一个横向的关系

AOP用来封装横切关注点，具体可以在下面的场景中使用：

- 1 Authentication 权限
- 2 Caching 缓存
- 3 Context passing 内容传递
- 4 Error handling 错误处理
- 5 Lazy loading 懒加载
- 6 Debugging 调试
- 7 logging, tracing, profiling and monitoring 记录跟踪 优化 校准
- 8 Performance optimization 性能优化
- 9 Persistence 持久化
- 10 Resource pooling 资源池
- 11 Synchronization 同步
- 12 Transactions 事务

3 MyBatis

参考博文：

<http://blog.csdn.net/u013142781/article/details/50388204>

MyBatis 是支持普通 SQL查询，存储过程和高级映射的优秀持久层框架。MyBatis 消除了几乎所有的JDBC代码和参数的手工设置以及结果集的检索。MyBatis 使用简单的 XML或注解用于配置和原始映射，将接口和 Java 的 POJOs(Plain Old Java Objects,普通的 Java对象)映射成数据库中的记录。

总体流程：

(1) 加载配置并初始化

触发条件：加载配置文件 将SQL的配置信息加载成为一个个MappedStatement对象（包括了传入参数映射配置、执

行的SQL语句、结果映射配置），存储在内存中。

(2)接收调用请求

触发条件：调用Mybatis提供的API传入参数：为SQL的ID和传入参数对象
处理过程：将请求传递给下层的请求处理器进行处理。

(3)处理操作请求

触发条件：API接口层传递请求过来 传入参数：为SQL的ID和传入参数对象

处理过程：

1. 根据SQL的ID查找对应的MappedStatement对象。
2. 根据传入参数对象解析MappedStatement对象，得到最终要执行的SQL和执行传入参数。
3. 获取数据库连接，根据得到的最终SQL语句和执行传入参数到数据库执行，并得到执行结果。
4. 根据MappedStatement对象中的结果映射配置对得到的执行结果进行转换处理，并得到最终的处理结果。
5. 释放连接资源。

(4)返回处理结果将最终的处理结果返回。

MyBatis 最强大的特性之一就是它的动态语句功能。如果您以前有使用JDBC或者类似框架的经历，您就会明白把SQL语句条件连接在一起是多么的痛苦，要确保不能忘记空格或者不要在columns列后面省略一个逗号等。动态语句能够完全解决掉这些痛苦。

4 Dubbo

参考博文：

<http://blog.csdn.net/u013142781/article/details/50387583>

Dubbo是一个分布式服务框架，致力于提供高性能和透明化的RPC（远程过程调用协议）远程服务调用方案，以及SOA服务治理方案。简单的说，dubbo就是个服务框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有dubbo这样的分布式服务框架的需求，并且本质上是个服务调用的东东，说白了就是个远程服务调用的分布式框架。

- 1、透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何API侵入。
- 2、软负载均衡及容错机制，可在内网替代F5等硬件负载均衡器，降低成本，减少单点。
- 3、服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者。
4. 微信搜索 Web项目聚集地 获取更多实战教程。

节点角色说明：

- 1 Provider：暴露服务的服务提供方。
- 2 Consumer：调用远程服务的服务消费方。
- Registry：服务注册与发现的注册中心。

- 3 Monitor: 统计服务的调用次调和调用时间的监控中心。
- 4 Container: 服务运行容器。
- 5

5 Maven

参考博文:

<http://blog.csdn.net/u013142781/article/details/50316383>

Maven这个个项目管理和构建自动化工具,越来越多的开发人员使用它来管理项目中的jar包。但是对于我们程序员来说,我们最关心的是它的项目构建功能。

6 RabbitMQ

参考博文:

<http://blog.csdn.net/u013142781/article/category/6061896>

消息队列一般是在项目中,将一些无需即时返回且耗时的操作提取出来,进行了异步处理,而这种异步处理的方式大大的节省了服务器的请求响应时间,从而提高了系统的吞吐量。

RabbitMQ是用Erlang实现的一个高并发高可靠AMQP消息队列服务器。

Erlang是一门动态类型的函数式编程语言。对应到Erlang里,每个Actor对应着一个Erlang进程,进程之间通过消息传递进行通信。相比共享内存,进程间通过消息传递来通信带来的直接好处就是消除了直接的锁开销(不考虑Erlang虚拟机底层实现中的锁应用)。

AMQP(Advanced Message Queue Protocol)定义了一种消息系统规范。这个规范描述了在一个分布式的系统中各个子系统如何通过消息交互。Dubbo是一个分布式服务框架,致力于提供高性能和透明

7 Log4j

参考博文:

<http://blog.csdn.net/u013142781/article/category/6045728>

日志记录的优先级,分为OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL或者您定义的级别。微信搜索 Web项目聚集地 获取更多实战教程。

8 Ehcache

参考博文:

<http://blog.csdn.net/u013142781/article/category/6066337>

EhCache 是一个纯Java的进程内缓存框架,具有快速、精干等特点,是Hibernate中默认的CacheProvider。Ehcache是一种广泛使用的开源Java分布式缓存。主要面向通用缓存,Java EE和轻量级容器。它具有内存和磁盘存储,缓存加载器,缓存扩展,缓存异常处理程序,一个gzip缓存servlet过滤器,支持REST和SOAP api等特点。

优点：

- 1 快速
- 2 简单
- 3 多种缓存策略
- 4 缓存数据有两级：内存和磁盘，因此无需担心容量问题
- 5 缓存数据会在虚拟机重启的过程中写入磁盘
- 6 可以通过RMI、可插入API等方式进行分布式缓存
- 7 具有缓存和缓存管理器的侦听接口
- 8 支持多缓存管理器实例，以及一个实例的多个缓存区域
- 9 提供Hibernate的缓存实现
- 10 微信搜索 Web项目聚集地 获取更多实战教程。

缺点：

1. 使用磁盘Cache的时候非常占用磁盘空间：这是因为DiskCache的算法简单，该算法简单也导致Cache的效率非常高。它只是对元素直接追加存储。因此搜索元素的时候非常的快。如果使用DiskCache的，在很频繁的应用中，很快磁盘会满。
2. 不能保证数据的安全：当突然kill掉java的时候，可能会产生冲突，EhCache的解决方法是如果文件冲突了，则重建cache。这对于Cache数据需要保存的时候可能不利。当然，Cache只是简单的加速，而不能保证数据的安全。如果想保证数据的存储安全，可以使用Bekeley DB Java Edition版本。这是个嵌入式数据库。可以确保存储安全和空间的利用率。

9 Redis

参考博文：

<http://blog.csdn.net/u013142781/article/category/6067864>

redis是一个key-value存储系统。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set -有序集合)和hash(哈希类型)。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。微信搜索 Web项目聚集地 获取更多实战教程。

在此基础上，redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

Redis数据库完全在内存中，使用磁盘仅用于持久性。相比许多键值数据存储，Redis拥有一套较为丰富数据类型。Redis可以将数据复制到任意数量的从服务器。

Redis优点：

异常快速：Redis的速度非常快，每秒能执行约11万集合，每秒约81000+条记录。支持丰富的数据类型：Redis支持大多数开发人员已经知道像列表，集合，有序集合，散列数据类型。这使得它非常容易解决各种各样的问题，因为我们知道哪些问题是可以通过它的数据类型更好。操作都是原子性：所有Redis操作是原子的，这保证了如果两个客户端同时访问的Redis服务器将获得更新后的值。多功能实用工具：Redis是一个多实用的工具，可以在多个用例如缓存，消息，队列使用(Redis原生支持发布/订阅)，任何短暂的数据，应用程序，如Web应用程序会话，网页命中计数等。

Redis缺点：

单线程 耗内存

10 Shiro

参考博文：

<http://blog.csdn.net/u013142781/article/details/50629708>

Apache Shiro是Java的一个安全框架，旨在简化身份验证和授权。Shiro在JavaSE和JavaEE项目中都可以使用。它主要用来处理身份认证，授权，企业会话管理和加密等。Shiro的具体功能点如下：

身份认证/登录，验证用户是不是拥有相应的身份；

授权，即权限验证，验证某个已认证的用户是否拥有某个权限；

即判断用户是否能做事情，常见的如：验证某个用户是否拥有某个角色。或者细粒度的验证某个用户对某个资源是否具有某个权限；

会话管理，即用户登录后就是一次会话，在没有退出之前，它的所有信息都在会话中；

会话可以是普通JavaSE环境的，也可以是如Web环境的；加密，保护数据的安全性，如密码加密存储到数据库，而不是明文存储；

Web支持，可以非常容易的集成到Web环境；

Caching：缓存，比如用户登录后，其用户信息、拥有的角色/权限不必每次去查，这样可以提高效率；

shiro支持多线程应用的并发验证，即如在一个线程中开启另一个线程，能把权限自动传播过去；提供测试支持；

允许一个用户假装为另一个用户（如果他们允许）的身份进行访问；

记住我，这个是非常常见的功能，即一次登录后，下次再来的话不用登录了。

文字描述可能并不能让猿友们完全理解具体功能的意思。下面我们以登录验证为例，向猿友们介绍Shiro的使用。至于其他功能点，猿友们用到的时候再去深究其用法也不迟。SSSS日志记录的优先级，分为OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL或者您定义的级别。

11 设计模式

这个算不上框架，可自行忽略，不过博主认为设计模式的思想很有必要了解一下。

参考博文：

<http://blog.csdn.net/u013142781/article/details/50816245>

<http://blog.csdn.net/u013142781/article/details/50821155>

<http://blog.csdn.net/u013142781/article/details/50825301>

思想：

开闭原则: 开闭原则就是说对扩展开放, 对修改关闭。在程序需要进行拓展的时候, 不能去修改原有的代码。

针对接口编程, 真对接口编程, 依赖于抽象而不依赖于具体。

尽量使用合成/聚合的方式, 而不是使用继承。

一个实体应当尽量少的与其他实体之间发生相互作用, 使得系统功能模块相对独立。

使用多个隔离的接口, 比使用单个接口要好。

里氏代换原则: 子类的能力必须大于等于父类, 即父类可以使用的方法, 子类都可以使用。

返回值也是同样的道理。假设一个父类方法返回一个List, 子类返回一个ArrayList, 这当然可以。如果父类方法返回一个ArrayList, 子类返回一个List, 就说不通了。这里子类返回值的能力是比父类小的。

还有抛出异常的情况。任何子类方法可以声明抛出父类方法声明异常的子类。

而不能声明抛出父类没有声明的异常。

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. 史上最烂的项目：苦撑12年，600 多万行代码 ...
2. 请给 Spring Boot 多一些内存
3. 如何从零搭建百亿流量系统？
4. 惊了！原来 Web 发展历史是这样的



Web项目聚集地

微信扫描二维码, 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！

Java equals 和 hashCode 的这几个问题可以说明白吗？

Java后端 2月22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



前言

昨日留言有一个有关 equals 和 hashCode 问题。基础面试经常会碰到与之相关的问题，这不是一个复杂的问题，但很多朋友都苦于说明他们二者的关系和约束，于是写本文做单独说明，本篇文章将循序渐进（通过举例，让记忆与理解更轻松）说明这些让你有些苦恼的问题，Let's go

面试问题

1. Java 里面有了 == 运算符，为什么还需要 equals ?

== 比较的是对象地址，equals 比较的是对象值

先来看一看 Object 类中 equals 方法：

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

我们看到 equals 方法同样是通过 == 比较对象地址，并没有帮我们比较值。Java 世界中 Object 绝对是“老祖宗”的存在，== 号我们没办法改变或重写。但 equals 是方法，这就给了我们重写 equals 方法的可能，让我们实现其对值的比较：

```
@Override  
public boolean equals(Object obj) {  
  
}
```

新买的电脑，每个电脑都有唯一的序列号，通常情况下，两个一模一样的电脑放在面前，你会说由于序列号不一样，这两个电脑不一样吗？

如果我们要说两个电脑一样，通常是比较其「品牌/尺寸/配置」(值)，比如这样：

```
@Override  
public boolean equals(Object obj) {  
    return 品牌相等 && 尺寸相等 && 配置相等  
}
```

当遇到如上场景时，我们就需要重写 equals 方法。这就解释了 Java 世界为什么有了 == 还有equals 这个问题了。

2. equals相等 和 hashCode 相等问题

关于二者，你经常会碰到下面的两个问题：

- 两个对象 equals 相等，那他们 hashCode 相等吗？

- 两个对象 hashCode 相等，那他们 equals 相等吗？

为了说明上面两个问题的结论，这里举一个不太恰当的例子，只为方便记忆，我们将 equals 比作一个单词的拼写； hashCode 比作一个单词的发音，在相同语境下：

sea / sea 「大海」，两个单词拼写一样，所以 equals 相等，他们读音 /si:/ 也一样，所以 hashCode 就相等，这就回答了第一个问题：

两个对象 equals 相等，那他们 hashCode 一定也相等

sea / see 「大海/看」，两个单词的读音 /si:/ 一样，显然单词是不一样的，这就回答了第二个问题：

两个对象 hashCode 相等，那他们 equals 不一定相等

查看 Object 类的 hashCode 方法：

```
public native int hashCode();
```

继续查看该方法的注释，明确写明关于该方法的约束

The screenshot shows the Java Object class's hashCode() method documentation. Several parts of the text are highlighted with red boxes and arrows pointing to specific annotations:

- A red box highlights the first bullet point under the general contract: "Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode() method must consistently return the same integer, provided no information used in equals() comparisons on the object is modified." An arrow from this box points to the text: "应用的两次启动执行，不用保证 hashCode 值时一样的，这也反面验证了为什么该方法是 native 的" (In two consecutive executions of a Java application, if the hashCode() method is called on the same object more than once, it must return the same integer value, provided no information used in equals() comparisons on the object is modified. This also proves why this method is native).
- A red box highlights the second bullet point: "If two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same integer result." An arrow from this box points to the text: "两个对象 'equals' 相等，那他们 'hashCode' 一定也相等" (If two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same integer result).
- A red box highlights the third bullet point: "It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables." An arrow from this box points to the text: "两个对象 'hashCode' 相等，那他们 'equals' 不一定相等" (It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables).

其实在这个结果的背后，还有的是关于重写 equals 方法的约束

3. 重写 equals 有哪些约束？

关于重写 equals 方法的约束，同样在该方法的注释中写的很清楚了，我在这里再说明一下：

```

/** 
 * Indicates whether some other object is "equal to" this one.
 * <p>
 * The {@code equals} method implements an equivalence relation
 * on non-null object references:
 * <ul>
 * <li>It is <b><code>reflexive</code></b>; for any non-null reference value
 *      {@code x}, {@code x.equals(x)} should return
 *      {@code true}.
 * <li>It is <b><code>symmetric</code></b>; for any non-null reference values
 *      {@code x} and {@code y}, {@code x.equals(y)}
 *      should return {@code true} if and only if
 *      {@code y.equals(x)} returns {@code true}.
 * <li>It is <b><code>transitive</code></b>; for any non-null reference values
 *      {@code x}, {@code y}, and {@code z}, if
 *      {@code x.equals(y)} returns {@code true} and
 *      {@code y.equals(z)} returns {@code true}, then
 *      {@code x.equals(z)} should return {@code true}.
 * <li>It is <b><code>consistent</code></b>; for any non-null reference values
 *      {@code x} and {@code y}, multiple invocations of
 *      {@code x.equals(y)} consistently return {@code true}
 *      or consistently return {@code false}, provided no
 *      information used in {@code equals} comparisons on the
 *      objects is modified.
 * <li>For any non-null reference value {@code x},
 *      {@code x.equals(null)} should return {@code false}.
 * </ul>

```

自反性：对于任何非空引用值 x, x.equals(x) 都应返回 true

对称性：对于任何非空引用值 x 和 y, 当且仅当 y.equals(x) 返回 true 时, x.equals(y) 才应返回 true

传递性：对于任何非空引用值 x、y 和 z, 如果 x.equals(y) 返回 true, 并且 y.equals(z) 返回 true, 那么 x.equals(z) 应返回 true

一致性：对于任何非空引用值 x 和 y, 多次调用 x.equals(y) 始终返回 true 或始终返回 false, 前提是对象上 equals 比较中所用的信息没有被修改

非空性：对于任何非空引用值 x, x.equals(null) 都应返回 false

赤橙红绿青蓝紫, 七彩以色列; 哆来咪发唆拉西, 一曲安哥拉 , 这些规则不是用来背诵的, 只是在你需要重写 equals 方法时, 打开 JDK 查看该方法, 按照准则重写就好

4. 什么时候需要我们重写 hashCode?

为了比较值, 我们重写 equals 方法, 那什么时候又需要重写 hashCode 方法呢?

通常只要我们重写 equals 方法就要重写 hashCode 方法

为什么会有这样的约束呢? 按照上面讲的原则, 两个对象 equals 相等, 那他们的 hashCode 一定也相等。如果我们只重写 equals 方法而不重写 hashCode 方法, 看看会发生什么, 举个例子来看:

定义学生类, 并通过 IDE 只帮我们生成 equals 方法:

```

public class Student {

    private String name;

    private int age;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return age == student.age &&
               Objects.equals(name, student.name);
    }
}

```

编写测试代码:

```
Student student1 = new Student();
student1.setName("日拱一兵");
student1.setAge(18);

Student student2 = new Student();
student2.setName("日拱一兵");
student2.setAge(18);

System.out.println("student1.equals(student2)的结果是: " + student1.equals(student2));

Set<Student> students = new HashSet<Student>();
students.add(student1);
students.add(student2);
System.out.println("Student Set 集合长度是: " + students.size());

Map<Student, java.lang.String> map = new HashMap<Student, java.lang.String>();
map.put(student1, "student1");
map.put(student2, "student2");
System.out.println("Student Map 集合长度是: " + map.keySet().size());
```

查看运行结果:

```
student1.equals(student2)的结果是: true
Student Set 集合长度是: 2
Student Map 集合长度是: 2
```

很显然,按照集合 Set 和 Map 加入元素的标准来看,student1 和 student2 是两个对象,因为在调用他们的 put (Set add 方法的背后也是 HashMap 的 put)方法时, 会先判断 hash 值是否相等, 这个小伙伴们打开 JDK 自行查看吧

所以我们继续重写 Student 类的 hashCode 方法:

```
@Override
public int hashCode() {
    return Objects.hash(name, age);
}
```

重新运行上面的测试, 查看结果:

```
student1.equals(student2)的结果是: true
Student Set 集合长度是: 1
Student Map 集合长度是: 1
```

得到我们预期的结果, 这也就是为什么通常我们重写 equals 方法为什么最好也重写 hashCode 方法的原因

- 如果你在使用 Lombok, 不知道你是否注意到 Lombok 只有一个 @EqualsAndHashCode 注解, 而没有拆分成 @Equals 和 @HashCode 两个注解, 想了解更多 Lombok 的内容, 也可以查看我之前写的文章 Lombok 使用详解
- 另外通过 IDE 快捷键生成重写方法时, 你也会看到这两个方法放在一起, 而不是像 getter 和 setter 那样分开

Generate

Constructor

Getter

Setter

Getter and Setter

equals() and hashCode()

toString()

Override Methods... ^O

Delegate Methods...

Test...

Copyright

GsonFormat

VS

以上两点都是隐形的规范约束，希望大家也严格遵守这个规范，以防带来不必要的麻烦，记忆的方式有多样，如果记不住这个文字约束，脑海中记住上面的图你也就懂了

5. 重写 hashCode 为什么总有 31 这个数字？

细心的朋友可能注意到，我上面重写 hashCode 的方法很简答，就是用了 Objects.hash 方法，进去查看里面的方法：

```
public static int hashCode(Object a[]) {  
    if (a == null)  
        return 0;  
  
    int result = 1;  
  
    for (Object element : a)  
        result = 31 * result + (element == null ? 0 : element.hashCode());  
  
    return result;  
}
```

这里通过 31 来计算对象 hash 值

在 如何妙用 Spring 数据绑定？文章末尾提到的在 HandlerMethodArgumentResolverComposite 类中有这样一个成员变量：

```
private final Map<MethodParameter, HandlerMethodArgumentResolver> argumentResolverCache =  
    new ConcurrentHashMap<MethodParameter, HandlerMethodArgumentResolver>(256)
```

Map 的 key 是 MethodParameter，根据我们上面的分析，这个类一定也会重写 equals 和 hashCode 方法，进去查看发现，hashCode 的计算也用到了 31 这个数字

```
@Override  
public boolean equals(Object other) {  
    if (this == other) {  
        return true;  
    }  
    if (!(other instanceof MethodParameter)) {  
        return false;  
    }  
    MethodParameter otherParam = (MethodParameter) other;  
    return (this.parameterIndex == otherParam.parameterIndex && getMember().equals(otherParam.getMember()));  
}  
  
@Override  
public int hashCode() {  
    return (getMember().hashCode() * 31 + this.parameterIndex);  
}
```

为什么计算 hash 值要用到 31 这个数字呢？我在网上看到一篇不错的文章，分享给大家，作为科普，可以简单查看一下：

String hashCode 方法为什么选择数字31作为乘子

总结

如果还对equals 和 hashCode 关系及约束含混，我们只需要按照上述步骤逐步回忆即可，更好的是直接查看 JDK 源码；另外拿出实际的例子来反推验证是非常好的办法。如果你还有相关疑问，也可以留言探讨。

- 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

[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！

Java 中一个令人惊讶的 BUG

海纳 Java后端 1周前

Java后端

作者：海纳

链接：zhuanlan.zhihu.com/p/88555159

今天分享一个JDK中令人惊讶的BUG，这个BUG的神奇之处在于，复现它的用例太简单了，人肉眼就能回答的问题，JDK中却存在了十几年。经过测试，我们发现从JDK8到14都存在这个问题。

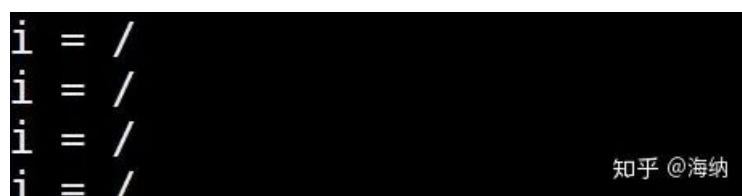
大家可以在自己的开发平台上试试这段代码：

```
public class Hello {  
    public void test() {  
        int i = 8;  
        while ((i -= 3) > 0);  
        System.out.println("i = " + i);  
    }  
  
    public static void main(String[] args) {  
        Hello hello = new Hello();  
        for (int i = 0; i < 50_000; i++) {  
            hello.test();  
        }  
    }  
}
```

再使用以下命令执行：

```
java Hello
```

然后，就会看到这样的输出：



i = /
i = /
i = /
i = /

知乎 @海纳

当然，在程序的开始阶段，还是能打印出正确的“i = -1”。

这个问题最终Huawei JDK的两名同事解决掉了，并且回合到社区。我这里大概讲一下分析的思路。

首先，使用解释执行可以发现，结果都是正确的，这就说明，这基本上是JIT编译器的问题，然后通过-XX:-TieredCompilation关闭C1编译，问题同样复现，但是使用-XX:TieredStopAtLevel=3将JIT编译停留在C阶段，问题就不复现，这可以确定是C2的问题了。

接下来，一名同事立即猜想到这个“/”其实是('0'-1)，刚好是字符零的ascii码减掉1。嗯，熟记ascii码表的重要性就体现出来了。

接下来，就是找到c2中 int 转字符的地方。关键点，就在于这个字符'0'，当然这里要对C2有足够的了解，马上就找到c2中字符转化的方法（具体的代码，请参考OpenJDK社区）：

```

void PhaseStringOpts::int_getChars(GraphKit& kit, Node* arg, Node* char_array, Node* start, Node* end) {
//.....
//char sign = 0;

Node* i = arg;
Node* sign = __ intcon(0);

//if(i < 0) {
//sign = '-';
//i = -i;
//}
{
IfNode* iff = kit.create_and_map_if(kit.control(),
    __ Bool(__ Cmpl(arg, __ intcon(0)), BoolTest::Lt),
    PROB_FAIR, COUNT_UNKNOWN);

RegionNode *merge = new (C) RegionNode(3);
kit.gvn().set_type(merge, Type::CONTROL);
i = new (C) PhiNode(merge, TypeInt::INT);
kit.gvn().set_type(i, TypeInt::INT);
sign = new (C) PhiNode(merge, TypeInt::INT);
kit.gvn().set_type(sign, TypeInt::INT);

merge->init_req(1, __ IfTrue(iff));
i->init_req(1, __ SubI(__ intcon(0), arg));
sign->init_req(1, __ intcon(-1));
merge->init_req(2, __ IfFalse(iff));
i->init_req(2, arg);
sign->init_req(2, __ intcon(0));

kit.set_control(merge);

C->record_for_igvn(merge);
C->record_for_igvn(i);
C->record_for_igvn(sign);
}

//for (;;) {
//q = i / 10;
//r = i - ((q << 3) + (q << 1)); //r = i - (q * 10) ...
//buf[-charPos] = digits[r];
//i = q;
//if (i == 0) break;
//}

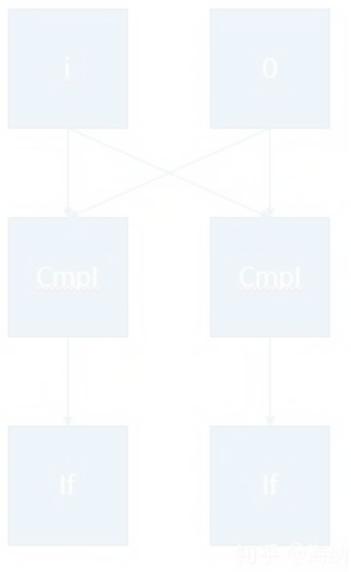
{
//略去和这个循环相对应的代码
}

//略去很多代码
}

```

可以看到，这里在中间表示阶段引入了一个“ $i < 0$ ”的判断。主要就是那个`Cmpl`结点，看起来这里的逻辑走错了，导致 i 明明小于0，结果却走到了大于0的分支，这样，直接拿字符'0'与 i 求和的结果，就是错的了。

那这个`Cmpl`为什么会错呢？使用c2visualizer工具可以看到，在GVN阶段，上面循环中的`Cmpl`和这里引入的`Cmpl`被合并了。GVN的全称是Global Value Numbering，名字很高大上，其实也就是表达式去重。例如：



上面的例子中，两个 Cmpl 的输入参数是完全相同的。都是变量 i 和整数 0，那么，这两个 Cmpl 结点其实也是完全相同的。这样的话，编译器在做中间优化的时候就会把这两个 Cmpl 结点合并成一个。

到这里为止，其实还是没问题的。但接下来，编译器会对空的循环体做一些特别的变换，编译器能直接计算出空循环体结束以后， i 的值是 -1，又发现空循环体什么都不做，所以，它干脆把 Cmpl 的两个参数都换成了 -1，以便于让循环走不进来——而且，编译器再做一次常量传播就可以把这个 Cmpl 彻底干掉了。但是，这里 Cmpl 就有问题了，这里强行搞成 False 让循环不执行，并且把 i 的值也直接变成循环结束的那个值。但刚才合并的那个 Cmpl 也被吃掉了。

这就导致，直接拿着 $i = -1$ 这个值进到了 $i >= 0$ 的分支里去了。所以修改也很简单，那就是在对 Cmpl 变换的时候，看看它还有没有其他的 out，如果有，就复制一份出来。

这个BUG的相关 issue 和 patch 在这里：

<https://bugs.openjdk.java.net/projects/JDK/issues/JDK-8231988?filter=allissues> bugs.openjdk.java.net

JBS 系统上没有详细的分析过程，只有最后的 patch，所以我把这个问题写了个总结发在这里。可以看到，即使是很简单的测试用例，在编译器内部也会经历各种复杂的变换和优化。然后一些阶段的优化可能会影响后一个阶段的，所以编译器的 BUG 也往往晦涩。但反过来说，也很有意思。

打个广告，Huawei JDK 团队诚聘各路英才，如果你对这篇文章的内容感兴趣，并且能看懂个大概，可以直接私信（知乎 id：海纳）。当然，只要你对 JVM 感兴趣，在垃圾回收，Java 类库开发等方面有相当的经验，我们也是欢迎的。

- END -

推荐阅读

1. Spring 中运用的 9 种设计模式吗？

2. Java 中的继承和多态（深入版）

3. 如何降低程序员的工资？

4. 编写 Spring MVC 的 14 个小技巧



微信搜一搜

Q 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！

Java 中关于 try、catch、finally 中的细节分析

Java后端 2月21日

作者：God Is Coder

<https://www.cnblogs.com/aigongsi>

看了一位博友的一篇文章，讲解的是关于java中关于try、catch、finally中一些问题

下面看一个例子（例1），来讲解java里面中try、catch、finally的处理流程

```
public class TryCatchFinally {  
  
    @SuppressWarnings("finally")  
    public static final String test() {  
        String t = "";  
  
        try {  
            t = "try";  
            return t;  
        } catch (Exception e) {  
            //result = "catch";  
            t = "catch";  
            return t;  
        } finally {  
            t = "finally";  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.print(TryCatchFinally.test());  
    }  
}
```

首先程序执行try语句块，把变量t赋值为try，由于没有发现异常，接下来执行finally语句块，把变量t赋值为finally，然后return t，则t的值是finally，最后t的值就是finally，程序结果应该显示finally，但是实际结果为try。为什么会这样，我们不妨先看看这段代码编译出来的class对应的字节码，看虚拟机内部是如何执行的。

我们用javap -verbose TryCatchFinally 来显示目标文件(.class文件)字节码信息

- 系统运行环境：mac os lion系统 64bit
- jdk信息：Java(TM) SE Runtime Environment (build 1.6.0_29-b11-402-11M3527) Java HotSpot(TM) 64-Bit Server VM (build 20.4-b02-402, mixed mode)

编译出来的字节码部分信息，我们只看test方法，其他的先忽略掉

```
public static final java.lang.String test();
```

Code:

```
Stack=1, Locals=4, Args_size=0
```

```
0: ldc #16; //String
```

```
2: astore_0
```

```
3: ldc #18; //String try
```

```
5: astore_0
```

```
6: aload_0
```

```
7: astore_3
```

```
8: ldc #20; //String finally
```

```
10: astore_0
```

```
11: aload_3
```

```
12: areturn
```

```
13: astore_1
```

```
14: ldc #22; //String catch
```

```
16: astore_0
```

```
17: aload_0
```

```
18: astore_3
```

```
19: ldc #20; //String finally
```

```
21: astore_0
```

```
22: aload_3
```

```
23: areturn
```

```
24: astore_2
```

```
25: ldc #20; //String finally
```

```
27: astore_0
```

```
28: aload_2
```

```
29: athrow
```

Exception table:

from to target type

```
8 13 Class java/lang/Exception
```

```
8 24 any
```

```
19 24 any
```

LineNumberTable:

```
line 5: 0
```

```
line 8: 3
```

```
line 9: 6
```

```
line 15: 8
```

```
line 9: 11
```

```
line 10: 13
```

```
line 12: 14
```

```
line 13: 17
```

```
line 15: 19
```

```
line 13: 22
```

```
line 14: 24
```

```
line 15: 25
```

```
line 16: 28
```

LocalVariableTable:

Start Length Slot Name Signature

```
27 0 t Ljava/lang/String;
```

```
10 1 e Ljava/lang/Exception;
```

StackMapTable: number_of_entries = 2

```
frame_type = 255 /* full_frame */
```

```
offset_delta = 13
```

```
locals = [ class java/lang/String ]
```

```
stack = [ class java/lang/Exception ]
```

```
frame_type = 74 /* same_locals_1_stack_item */
```

```
stack = [ class java/lang/Throwable ]
```

首先看LocalVariableTable信息，这里面定义了两个变量 一个是t String类型,一个是e Exception 类型

接下来看Code部分

第[0-2]行，给第0个变量赋值 “” ，也就是String t="";

第[3-6]行，也就是执行try语句块 赋值语句，也就是 t = "try";

第7行，重点是第7行，把第s对应的值"try"付给第三个变量，但是这里面第三个变量并没有定义,这个比较奇怪

第[8-10]行，对第0个变量进行赋值操作，也就是t="finally"

第[11-12]行，把第三个变量对应的值返回

通过字节码，我们发现，在try语句的return块中，return 返回的引用变量（t 是引用类型）并不是try语句外定义的引用变量t，而是系统重新定义了一个局部引用t'，这个引用指向了引用t对应的值，也就是try，即使在finally语句中把引用t指向了值finally，因为return的返回引用已经不是t，所以引用t的对应的值和try语句中的返回值无关了。

下面在看一个例子：（例2）

```
public class TryCatchFinally {  
  
    @SuppressWarnings("finally")  
    public static final String test() {  
        String t = "";  
  
        try {  
            t = "try";  
            return t;  
        } catch (Exception e) {  
            //result = "catch";  
            t = "catch";  
            return t;  
        } finally {  
            t = "finally";  
            return t;  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.print(TryCatchFinally.test());  
    }  
}
```

这里稍微修改了第一段代码，只是在finally语句块里面加入了一个return t 的表达式。

按照第一段代码的解释，先进行try{}语句，然后在return之前把当前的t的值try保存到一个变量t'，然后执行finally语句块，修改了变量t的值，在返回变量t。

这里面有两个return语句，但是程序到底返回的是try 还是 finally。接下来我们还是看字节码信息

```
public static final java.lang.String test();
```

Code:

```
Stack=1, Locals=2, Args_size=0
```

```
0: ldc #16; //String
```

```
2: astore_0
```

```
3: ldc #18; //String try
```

```
5: astore_0
```

```
6: goto 17
```

```
9: astore_1
```

```
10: ldc #20; //String catch
```

```
12: astore_0
```

```
13: goto 17
```

```
16: pop
```

```
17: ldc #22; //String finally
```

```
19: astore_0
```

```
20: aload_0
```

```
21: areturn
```

Exception table:

from to target type

```
9  9  Class java/lang/Exception
```

```
16  16  any
```

LineNumberTable:

```
line 5: 0
```

```
line 8: 3
```

```
line 9: 6
```

```
line 10: 9
```

```
line 12: 10
```

```
line 13: 13
```

```
line 14: 16
```

```
line 15: 17
```

```
line 16: 20
```

LocalVariableTable:

Start Length Slot Name Signature

```
19  0  t Ljava/lang/String;
```

```
6  1  e Ljava/lang/Exception;
```

StackMapTable: number_of_entries = 3

```
frame_type = 255 /* full_frame */
```

```
offset_delta = 9
```

```
locals = [ class java/lang/String ]
```

```
stack = [ class java/lang/Exception ]
```

```
frame_type = 70 /* same_locals_1_stack_item */
```

```
stack = [ class java/lang/Throwable ]
```

```
frame_type = 0 /* same */
```

这段代码翻译出来的字节码和第一段代码完全不同，还是继续看code属性

第[0-2]行、[3-5]行第一段代码逻辑类似，就是初始化t，把try中的t进行赋值try

第6行，这里面跳转到第17行，[17-19]就是执行finally里面的赋值语句，把变量t赋值为finally，然后返回t对应的值

我们发现try语句中的return语句给忽略。可能jvm认为一个方法里面有两个return语句并没有太大的意义，所以try中的return语句给忽略了，直接起作用的是finally中的return语句，所以这次返回的是finally。

接下来在看看复杂一点的例子：（例3）

```

public class TryCatchFinally {

    @SuppressWarnings("finally")
    public static final String test() {
        String t = "";

        try {
            t = "try";
            Integer.parseInt(null);
            return t;
        } catch (Exception e) {
            t = "catch";
            return t;
        } finally {
            t = "finally";
            //System.out.println(t);
            //return t;
        }
    }

    public static void main(String[] args) {
        System.out.print(TryCatchFinally.test());
    }
}

```

这里面try语句里面会抛出 java.lang.NumberFormatException，所以程序会先执行catch语句中的逻辑，t赋值为catch，在执行return之前，会把返回值保存到一个临时变量里面t'，执行finally的逻辑，t赋值为finally，但是返回值和t'，所以变量t的值和返回值已经没有关系了，返回的是catch

例4：

```

public class TryCatchFinally {

    @SuppressWarnings("finally")
    public static final String test() {
        String t = "";

        try {
            t = "try";
            Integer.parseInt(null);
            return t;
        } catch (Exception e) {
            t = "catch";
            return t;
        } finally {
            t = "finally";
            return t;
        }
    }

    public static void main(String[] args) {
        System.out.print(TryCatchFinally.test());
    }
}

```

这个和例2有点类似，由于try语句里面抛出异常，程序转入catch语句块，catch语句在执行return语句之前执行finally，而finally语句有return，则直接执行finally的语句值，返回finally

例5：

```

public class TryCatchFinally {

    @SuppressWarnings("finally")
    public static final String test() {
        String t = "";

        try {
            t = "try";
            Integer.parseInt(null);
            return t;
        } catch (Exception e) {
            t = "catch";
            Integer.parseInt(null);
            return t;
        } finally {
            t = "finally";
            //return t;
        }
    }

    public static void main(String[] args) {
        System.out.print(TryCatchFinally.test());
    }
}

```

这个例子在catch语句块添加了Integer.parseInt(null)语句，强制抛出了一个异常。然后finally语句块里面没有return语句。继续分析一下，由于try语句抛出异常，程序进入catch语句块，catch语句块又抛出一个异常，说明catch语句要退出，则执行finally语句块，对t进行赋值。然后catch语句块里面抛出异常。结果是抛出java.lang.NumberFormatException异常

例子6：

```

public class TryCatchFinally {

    @SuppressWarnings("finally")
    public static final String test() {
        String t = "";

        try {
            t = "try";
            Integer.parseInt(null);
            return t;
        } catch (Exception e) {
            t = "catch";
            Integer.parseInt(null);
            return t;
        } finally {
            t = "finally";
            return t;
        }
    }

    public static void main(String[] args) {
        System.out.print(TryCatchFinally.test());
    }
}

```

这个例子和上面例子中唯一不同的是，这个例子里面finally语句里面有return语句块。try catch中运行的逻辑和上面例子一样，当catch语句块里面抛出异常之后，进入finally语句块，然后返回t。则程序忽略catch语句块里面抛出的异常信息，直接返回t对应的值也就是finally。方法不会抛出异常

例子7：

```
public class TryCatchFinally {  
  
    @SuppressWarnings("finally")  
    public static final String test() {  
        String t = "";  
  
        try {  
            t = "try";  
            Integer.parseInt(null);  
            return t;  
        } catch (NullPointerException e) {  
            t = "catch";  
            return t;  
        } finally {  
            t = "finally";  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.print(TryCatchFinally.test());  
    }  
}
```

这个例子里面catch语句里面catch的是NPE异常，而不是java.lang.NumberFormatException异常，所以不会进入catch语句块，直接进入finally语句块，finally对s赋值之后，由try语句抛出java.lang.NumberFormatException异常。

例子8

```
public class TryCatchFinally {  
  
    @SuppressWarnings("finally")  
    public static final String test() {  
        String t = "";  
  
        try {  
            t = "try";  
            Integer.parseInt(null);  
            return t;  
        } catch (NullPointerException e) {  
            t = "catch";  
            return t;  
        } finally {  
            t = "finally";  
            return t;  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.print(TryCatchFinally.test());  
    }  
}
```

和上面的例子中try catch的逻辑相同，try语句执行完成执行finally语句，finally赋值s 并且返回s，最后程序结果返回finally

例子9：

```
public class TryCatchFinally {  
  
    @SuppressWarnings("finally")  
    public static final String test() {  
        String t = "";  
  
        try {  
            t = "try"; return t;  
        } catch (Exception e) {  
            t = "catch";  
            return t;  
        } finally {  
            t = "finally";  
            String.valueOf(null);  
            return t;  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.print(TryCatchFinally.test());  
    }  
}
```

这个例子中，对finally语句中添加了String.valueOf(null)，强制抛出NPE异常。首先程序执行try语句，在返回执行，执行finally语句块，finally语句抛出NPE异常，整个结果返回NPE异常。

对以上所有的例子进行总结

1. try、catch、finally语句中，在如果try语句有return语句，则返回的之后当前try中变量此时对应的值，此后对变量做任何的修改，都不影响try中return的返回值
2. 如果finally块中有return语句，则返回try或catch中的返回语句忽略。
3. 如果finally块中抛出异常，则整个try、catch、finally块中抛出异常

所以使用try、catch、finally语句块中需要注意的是

1. 尽量在try或者catch中使用return语句。通过finally块中达到对try或者catch返回值修改是不可行的。
2. finally块中避免使用return语句，因为finally块中如果使用return语句，会显示的消化掉try、catch块中的异常信息，屏蔽了错误的发生
3. finally块中避免再次抛出异常，否则整个包含try语句块的方法会抛出异常，并且会消化掉try、catch块中的异常

-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. [2020 年 9 大顶级 Java 框架](#)
2. [聊聊 API 网关的作用](#)
3. [Class.forName 和 ClassLoader 有什么区别？](#)
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！

Java后端

面向对象的三大特性：封装、继承、多态。在这三个特性中，如果没有封装和继承，也不会有多态。

那么多态实现的途径和必要条件是什么呢？以及**多态中的重写和重载在JVM中的表现**是怎么样？在Java中是如何展现继承的特性呢？对于子类继承于父类时，又有什么限制呢？本文系基础，深入浅出过一遍 Java 中的多态和继承。

多态

多态是同一个行为具有多个不同表现形式或形态的能力。

举个栗子，一只鸡可以做成白切鸡、豉油鸡、吊烧鸡、茶油鸡、盐焗鸡、葱油鸡、手撕鸡、清蒸鸡、叫花鸡、啤酒鸡、口水鸡、香菇滑鸡、盐水鸡、啫啫滑鸡、鸡公煲等等。

多态实现的必要条件

用上面的“鸡的十八种吃法”来举个栗子。

首先，我们先给出一只鸡：

```
class Chicken{
    public void live(){
        System.out.println("这是一只鸡");
    }
}
```

1. 子类必须继承父类

对于子类必须继承父类，小编个人认为，是因为按照面向对象的五大基本原则所说的中的依赖倒置原则：**抽象不依赖于具体，具体依赖于抽象**。既然要实现多态，那么必定有一个作为“抽象”类来定义“行为”，以及若干个作为“具体”类来呈现不同的行为形式或形态。

所以我们给出的一个具体类——白切鸡类：

```
class BaiqieChicken extends Chicken{ }
```

但仅是定义一个白切鸡类是不够的，因为在此我们只能做到复用父类的属性和行为，而没有呈现出行为上的不同的形式或形态。

2. 必须有重写

重写，简单地理解就是重新定义的父类方法，使得父类和子类对同一行为的表现形式各不相同。我们用白切鸡类来举个栗子。

```
class BaiqieChicken extends Chicken{  
    public void live(){  
        System.out.println("这是一只会被做成白切鸡的鸡");  
    }  
}
```

这样就实现了重写，鸡类跟白切鸡类在live()方法中定义的行为不同，鸡类是一只命运有着无限可能的鸡，而白切鸡类的命运就是做成一只白切鸡。

但是为什么还要有“父类引用指向子类对象”这个条件呢？

3. 父类引用指向子类对象

其实这个条件是面向对象的五大基本原则里面的里氏替换原则，简单说就是父类可以引用子类，但不能反过来。

当一只鸡被选择做白切鸡的时候，它的命运就不是它能掌控的。

```
Chicken c = new BaiqieChicken();  
c.live();
```

运行结果：

这是一只会被做成白切鸡的鸡

为什么要有这个原则？因为父类对于子类来说，是属于“抽象”的层面，子类是“具体”的层面。“抽象”可以提供接口给“具体”实现，但是“具体”凭什么来引用“抽象”呢？而且“子类引用指向父类对象”是不符合“依赖倒置原则”的。

当一只白切鸡想回头重新选择自己的命运，抱歉，它已经在锅里，逃不出来了。

```
BaiqieChicken bc = new Chicken();  
bc.live();
```

多态的实现途径

多态的实现途径有三种：重写、重载、接口实现，虽然它们的实现方式不一样，但是核心都是：同一行为的不同表现形式。

1. 重写

重写，指的是子类对父类方法的重新定义，但是子类方法的参数列表和返回值类型，必须与父类方法一致！所以可以简单的理解，重写就是子类对父类方法的核心进行重新定义。

举个栗子：

```
class Chicken{
    public void live(String lastword){
        System.out.println(lastword);
    }
}

class BaiqieChicken extends Chicken{
    public void live(String lastword){
        System.out.println("这只白切鸡说：");
        System.out.println(lastword);
    }
}
```

这里白切鸡类重写了鸡类的live()方法，为什么说是重写呢？因为白切鸡类中live()方法的参数列表和返回值与父类一样，但方法体不一样了。

2. 重载

重载，指的是在一个类中有若干个方法名相同，但参数列表不同的情况，返回值可以相同也可以不同的方法定义场景。也可以简单理解成，同一行为（方法）的不同表现形式。

举个栗子：

```
class BaiqieChicken extends Chicken{
    public void live(){
        System.out.println("这是一只会被做成白切鸡的鸡");
    }

    public void live(String lastword){
        System.out.println("这只白切鸡说：");
        System.out.println(lastword);
    }
}
```

这里的白切鸡类中的两个live()方法，一个无参一个有参，它们对于白切鸡类的live()方法的描述各不相同，但它们的方法名都是live。通俗讲，它们对于白切鸡的表现形式不同。

3. 接口实现

接口，是一种无法被实例化，但可以被实现的抽象类型，是抽象方法的集合，多用作定义方法集合，而方法的具体实现则交给继承接口的具体类来定义。所以，**接口定义方法，方法的实现在继承接口的具体类中定义，也是对同一行为的不同表现形式。**

```
interface Chicken{
    public void live();
}

class BaiqieChicken implements Chicken{
    public void live(){
        System.out.println("这是一只会被做成白切鸡的鸡");
    }
}

class ShousiChicken implements Chicken{
    public void live(){
        System.out.println("这是一只会被做成手撕鸡的鸡");
    }
}
```

从上面我们可以看到，对于鸡接口中的live()方法，白切鸡类和手撕鸡类都有自己对这个方法的独特的定义。

在虚拟机中多态如何表现

前文我们知道，java文件在经过javac编译后，生成class文件之后在JVM中再进行编译后生成对应平台的机器码。而JVM的编译过程中体现多态的过程，在于选择出正确的方法执行，这一过程称为**方法调用**。

方法调用的唯一任务是确定被调用方法的版本，暂时还不涉及方法内部的具体运行过程。（注：方法调用不等于方法执行）

在介绍多态的重载和重写在JVM的实现之前，我们先简单了解JVM提供的5条方法调用字节码指令：

invokestatic：调用静态方法。

invokespecial：调用实例构造器方法、私有方法和父类方法。

invokevirtual：调用所有的虚方法（这里的虚方法泛指除了invokestatic、invokespecial指令调用的方法，以及final方法）。

invokeinterface：调用接口方法，会在运行时再确定一个实现此接口的对象。

invokedynamic：先在运行时动态解析出调用点限定符所应用的方法（说人话就是用于动态指定运行的方法）。

而方法调用过程中，在编译期就能确定下来调用方法版本的**静态方法、实例构造器方法、私有方法、父类方法和final方法**（虽是由invokevirtual指令调用）在编译期就已经完成了运行方法版本的确定，这是一个静态的过程，也称为**解析调用**。

而**分派调用**则有可能是静态的也可能是动态的，可能会在编译期发生或者运行期才确定运行方法的版本。

而分派调用的过程与多态的实现有着紧密联系，所以我们先了解一下两个概念：

静态分派：所有依赖静态类型来定位方法执行版本的分派动作。

动态分派：根据运行期实际类型来定位方法执行版本的分派动作。

1. 重载

我们先看看这个例子：

```
public class StaticDispatch {  
    static abstract class Human{}  
    static class Man extends Human{}  
    static class Woman extends Human{}  
  
    public void sayHello(Human guy){  
        System.out.println("hello, guy!");  
    }  
    public void sayHello(Man guy){  
        System.out.println("hello, gentleman!");  
    }  
    public void sayHello(Woman guy){  
        System.out.println("hello, lady!");  
    }  
  
    public static void main(String[] args){  
        Human man = new Man();  
        Human woman = new Woman();  
        StaticDispatch sr = new StaticDispatch();  
        sr.sayHello(man);  
        sr.sayHello(woman);  
    }  
}
```

想想以上代码的运行结果是什么？3，2，1，运行结果如下：

```
hello, guy!  
hello, guy!
```

为什么会出现这样的结果？让我们来看这行代码：

```
Human man = new Man();
```

根据里氏替换原则，子类必须能够替换其基类，也就是说子类相对于父类是“具体类”，而父类是处于“奠定”子类的基本功能的地位。

所以，我们把上面代码中的“Human”称为变量man的**静态类型**(Static Type)，而后面的"Man"称为变量的**实际类型**(Actual Type)，二者的区别在于，**静态类型是在编译期可知的；而实际类型的结果在运行期才能确定**，编译期在编译程序时并不知道一个对象的实际类型是什么。

在了解了这两个概念之后，我们来看看字节码文件是怎么说的：

```
javac -verbose StaticDispatch.class
```

```
0: new           #7                  // class StaticDispatch$Man
3: dup
4: invokespecial #8                // Method StaticDispatch$Man."<init>":()V
7: astore_1
8: new           #9                  // class StaticDispatch$Woman
11: dup
12: invokespecial #10              // Method StaticDispatch$Woman."<init>":()V
15: astore_2
16: new           #11                 // class StaticDispatch
19: dup
20: invokespecial #12              // Method "<init>":()V
23: astore_3
24: aload_3
25: aload_1
26: invokevirtual #13              // Method sayHello:(LStaticDispatch$Human;)V
29: aload_3
30: aload_2
31: invokevirtual #13              // Method sayHello:(LStaticDispatch$Human;)V
```

我们看到，图中的黄色框的invokespecial指令以及标签，我们可以知道这三个是指令是在调用实例构造器方法。同理，下面两个红色框的invokevirtual指令告诉我们，这里是采用分派调用的调用虚方法，而且入参都是“Human”。

因为在分派调用的时候，使用哪个重载版本完全取决于传入参数的数量和数据类型。而且，**虚拟机（准确说是编译期）在重载时是通过参数的静态类型而不是实际类型作为判断依据**，并且静态类型是编译期可知的。

所以，在编译阶段，Java编译期就会根据参数的静态类型决定使用哪个重载版本。重载是静态分派的经典应用。

2. 重写

我们还是用上面的例子：

```
public class StaticDispatch {  
    static abstract class Human{  
        protected abstract void sayHello();  
    }  
    static class Man extends Human{  
        @Override  
        protected void sayHello() {  
            System.out.println("man say hello");  
        }  
    }  
    static class Woman extends Human{  
        @Override  
        protected void sayHello() {  
            System.out.println("woman say hello");  
        }  
    }  
    public static void main(String[] args){  
        Human man = new Man();  
        Human woman = new Woman();  
        man.sayHello();  
        woman.sayHello();  
    }  
}
```

其运行结果为：

```
man say hello  
woman say hello
```

相信你看到这里也会会心一笑，这一看就很明显嘛，重写是按照实际类型来选择方法调用的版本嘛。先别急，我们来看看它的字节码：

Code:

```
stack=2, locals=3, args_size=1  
 0: new           #2                  // class DynamicDispatch$Man  
 3: dup  
 4: invokespecial #3                // Method DynamicDispatch$Man."<init>":()V  
 7: astore_1  
 8: new           #4                  // class DynamicDispatch$Woman  
11: dup  
12: invokespecial #5                // Method DynamicDispatch$Woman."<init>":()V  
15: astore_2  
16: aload_1  
17: invokevirtual #6                // Method DynamicDispatch$Human.sayHello:()V  
20: aload_2  
21: invokevirtual #6                // Method DynamicDispatch$Human.sayHello:()V  
24: return
```

<https://blog.csdn.net/NYfor2017>

嘶…这好像跟静态分派的字节码一样啊，但是从运行结果看，这两句指令最终执行的目方法并不相同啊，那原因就得从 invokevirtual指令的多态查找过程开始找起。

我们来看看invokevirtual指令的运行时解析过程的步骤：

1. 找到操作数栈顶的第一个元素所指向的对象的**实际类型**, 记作C。
2. 如果在在类型C中找到与常量中的描述符和简单名称都相符的方法, 则进行访问权限校验, 如果通过则返回这个方法的**直接引用**, 查找过程结束; 如果不通过, 则返回java.lang.IllegalAccessError异常。
3. 否则, 按照继承关系从下往上依次对**C的各个父类**进行第2步的搜索和验证过程。
4. 如果始终没有找到合适的方法, 则抛出java.lang.AbstractMethodError异常。

我们可以看到, 由于invokevirtual指令在执行的第一步就是**在运行期确定接收者的实际类型**, 所以字节码中会出现 invokevirtual指令把常量池中的类方法符号引用解析到了不同的直接引用上, 这个就是Java重写的本质。

总结一下, 重载的本质是在编译期就会根据参数的静态类型来决定重载方法的版本, 而重写的基本**在运行期确定接收者的实际类型**。

继承

假如我们有两个类: 生物类、猫类。

生物类:

```
class Animal{  
    private String name;  
    public void setName(String name){  
        this.name = name;  
    }  
    public String getName(){  
        return this.name;  
    }  
}
```

猫类:

```
class Cat{  
    private String name;  
    private String sound;  
    public void setName(String name){  
        this.name = name;  
    }  
    public void setSound(String sound){  
        this.sound = sound;  
    }  
    public String getName(){  
        return this.name;  
    }  
    public String getSound(){  
        return this.sound;  
    }  
}
```

我们知道, 猫也是属于生物中的一种, 生物有的属性和行为, 猫按理来说也是有的。但此时**没有继承的概念, 那么代码就得不到复用**, 长期发展, 代码冗余、维护困难且开发者的工作量也非常大。

继承的概念

继承就是子类继承父类的特征和行为, 使得子类对象(实例)具有父类的实例域和方法, 或子类从父类继承方法, 使得子类具有父类相同的行为。

简单来说，子类能吸收父类已有的属性和行为。除此之外，子类还可以扩展自身功能。子类又被称为派生类，父类被称为超类。

在 Java 中，如果要实现继承的关系，可以使用如下语法：

```
class 子类 extends 父类{}
```

继承的基本实现

继承的基本实现如下：

```
class Animal{
    private String name;
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return this.name;
    }
}
class Cat extends Animal{

public class Test{
    public static void main(String[] args){
        Cat cat = new Cat();
        cat.setName("猫");
        System.out.println(cat.getName());
    }
}
```

运行结果为：

猫

我们可以看出，子类可以在不扩展操作的情况下，使用父类的属性和功能。

子类扩充父类

继承的基本实现如下：

```

class Animal{
    private String name;
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return this.name;
    }
}
class Cat extends Animal{
    private String sound;
    public void setSound(String sound){
        this.sound = sound;
    }
    public String getSound(){
        return this.sound;
    }
}

public class Test{
    public static void main(String[] args){
        Cat cat = new Cat();
        cat.setName("NYfor2020")
        cat.setSound("我不是你最爱的小甜甜了吗？");
        System.out.println(cat.getName()+":"+cat.getSound());
    }
}

```

运行结果为：

NYfor2020:我不是你最爱的小甜甜了吗？

我们可以看出，子类在父类的基础上进行了扩展，而且对于父类来说，子类定义的范围更为具体。也就是说，**子类是将父类具体化的一种手段。**

总结一下，Java中的继承利用子类和父类的关系，**可以实现代码复用**，子类还可以根据需求扩展功能。

继承的限制

1. 子类只能继承一个父类

为什么子类不能多继承？举个栗子。

```

class ACat{
    public void mewo(){...}
}
class BCat{
    public void mewo(){...}
}
class CCat extends ACat, BCat{
    @Override
    public void mewo(){...?} //提问：这里的mewo()是继承自哪个类？
}

```

虽说Java只支持单继承，但是**不反对多层继承呀！**

```
class ACat{}  
class BCat extends ACat{}  
class CCat extends BCat{}
```

这样，BCat就继承了ACat所有的方法，而CCat继承了ACat、BCat所有的方法，实际上CCat是ACat的子（孙）类，是BCat的子类。

总结一下，子类虽然不支持多重继承，只能单继承，但是可以多层继承。

2. private修饰不可直接访问，final修饰不可修改

private修饰

对于子类来说，父类中用**private**修饰的属性对其隐藏的，但如果提供了这个变量的**setter/getter**接口，还是能够访问和修改这个变量的。

```
class ACat {  
    private String sound = "meow";  
    public String getSound(){  
        return sound;  
    }  
    public void setSound(String sound){  
        this.sound = sound;  
    }  
}  
  
class BCat extends ACat {  
}  
  
public class Test {  
    public static void main(String[] args) {  
        BCat b = new BCat();  
        b.setSound("我不是你最爱的小甜甜了吗？");  
        System.out.println(b.getSound());  
    }  
}
```

final修饰

父类已经定义好的**final**修饰变量（方法也一样），子类可以访问这个属性（或方法），但是不能对其进行更改。

```
class ACat {  
    final String sound = "你是我最爱的小甜甜";  
    public String getSound(){  
        return sound;  
    }  
    public void setSound(String sound){  
        this.sound = sound; //这句执行不了，会报错的  
    }  
}  
  
class BCat extends ACat {  
}
```

总结一下，用**private**修饰的变量可以通过**getter/setter**接口来操作，**final**修饰的变量就只能访问，不能更改。

3. 实例化子类时默认先调用父类的构造方法

在实例化子类对象时，会调用父类的构造方法对属性进行初始化，之后再调用子类的构造方法。

```
class A {
    public A(){
        System.out.println("我不是你最爱的小甜甜了吗？");
    }
    public A(String q){
        System.out.println(q);
    }
}

class B extends A {
    public B(){
        System.out.println("你是个好姑娘");
    }
}

public class Test {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

运行结果为：

我不是你最爱的小甜甜了吗？

你是个好姑娘

从结果我们可以知道，在实例化子类时，会默认先调用父类中无参构造方法，然后再调动子类的构造方法。

那么怎么调用父类带参的构造方法呢？只要在子类构造方法的第一行调用super()方法就好。

```
class A {
    public A(String q){
        System.out.println(q);
    }
}

class B extends A {
    public B(){
        super("我是你的小甜甜？");
        System.out.println("你是个好姑娘");
    }
}

public class Test {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

运行结果为：

我是你的小甜甜？

你是个好姑娘

在子类实例化时，默认调用的是父类的无参构造方法，而如果没有父类无参构造方法，则子类必须通过super()来调用父类的有参构造方法，且super()方法必须在子类构造方法的首行。

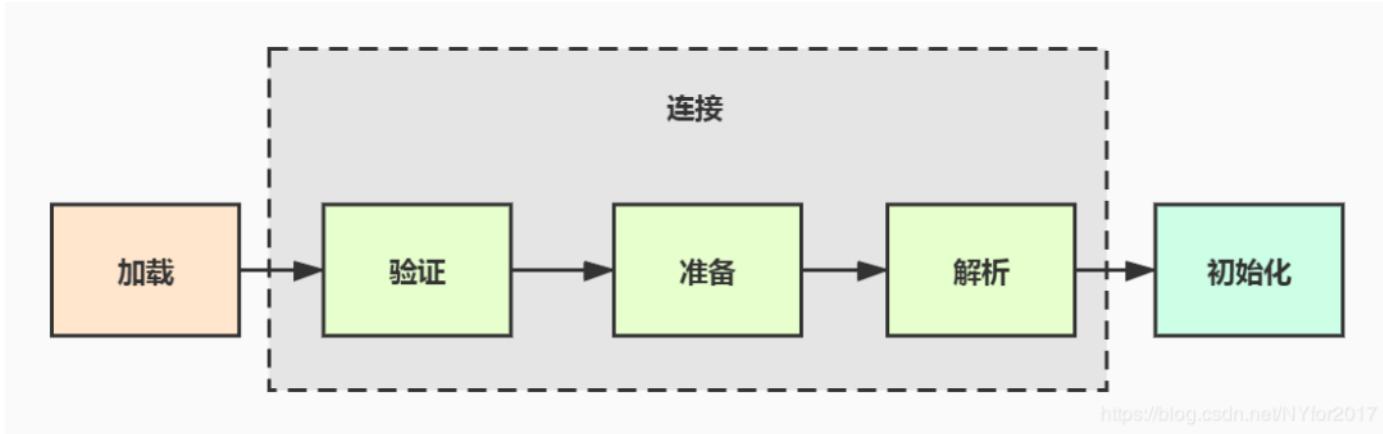
总结一下，Java继承中有三种继承限制，分别是子类只能单继承、父类中private修饰的变量不能显式访问和final修饰的变量不能改变，以及实例化子类必定会先调用父类的构造方法，之后才调用子类的构造方法。

类是怎么加载的？

(此处只是粗略介绍类加载的过程，想了解更多可参考《深入理解Java虚拟机》)

类加载过程包括三个大步骤：**加载、连接、初始化**。

这三个步骤的开始时间仍然保持着固定的先后顺序，但是进行和完成的进度就不一定是这样的顺序了。



1. 加载：虚拟机通过这个类的全限定名来获取这个类的二进制字节流，然后在字节流中提取出这个类的结构数据，并转换成这个类在方法区（存储类结构）的运行时数据结构；
2. 验证：先验证这字节流是否符合Class文件格式的规范，然后检查这个类的其父类中数据是否存在冲突（如这个类的父类是否继承被final修饰的类），接着对这个类内的方法体进行检查，如果都没问题了，那就把之前的符号引用换成直接引用；
3. 准备：为类变量（static修饰的变量）分配内存（方法区）并设置类变量初始值，而这里的初始值是指这个数据类型的零值，如int的初始值是0；
4. 解析：在Class文件加载过程中，会将Class文件中的标识方法、接口的常量放进常量池中，而这些常量对于虚拟机来说，就是 符号引用。此阶段就是针对类、接口、字段等7类符号引用，转换成直接指向目标的句柄——直接引用。
5. 初始化：这阶段是执行static代码块和类构造器的过程，有小伙伴可能会疑惑类构造器不是默认static的吗？详情请看这个博客：
<https://www.cnblogs.com/dolphin0520/p/10651845.html>

总结一下，类加载的过程中，首先会对Class文件中的类提取并转换成运行时数据结构，然后对类的父类和这个类的数据信息进行检验之后，为类中的类变量分配内存并且设置初始值，接着将Class文件中与这个类有关的符号引用转换成直接引用，最后再执行类构造器。

而且我们可以从第二步看出，在加载类的时候，会先去检查这个类的父类的信息，然后再检查这个类的方法体，也就是说，在加载类的时候，会先去加载它的父类。

结语

初学Java的时候知道这些概念，但只是浅尝而止。现在跟着Hollis大佬的《Java 工程师成神之路！》，重新回顾这些知识的时候，发现如果自己只是像以前一样片面了解，那岂不是没有成长？所以在写文章的过程中，尝试写得更加深入且尽量易懂。当然，本人水平有限，如有不正之处，欢迎指正。

坚持写技术文章的确是一件不容易的事情。现在技术更新越来越快，但是依然想把基础再打牢一点。

参考资料：

Java多态性理解

<https://www.cnblogs.com/jack204/archive/2012/10/29/2745150.html>

从虚拟机指令执行的角度分析JAVA中多态的实现原理

<https://www.cnblogs.com/hapjin/p/9248525.html>

《深入理解Java虚拟机》

<https://blog.csdn.net/weizhi/article/details/52780026>

- E N D -

推荐阅读

1. 如何降低程序员的工资？
2. 编写 Spring MVC 的 14 个小技巧
3. 技术大佬的呕心力作！
4. 详述 Spring Data JPA 的那些事儿



[阅读全文](#)

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！

Java中的责任链设计模式

Mazin Java后端 2019-12-03

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | Mazin

来源 | my.oschina.net/u/3441184/blog/889552

责任链设计模式的思想很简单，就是按照链的顺序执行一个个处理方法，链上的每一个任务都持有它后面那个任务的对象引用，以方便自己这段执行完成之后，调用其后面的处理逻辑。

下面是一个责任链设计模式的简单的实现：

```
public interface Task{  
    public void run();  
}  
  
public class Task1 implements Task{  
  
    private Task task;  
  
    public Task1() {}  
  
    public Task1(Task task){  
        this.task = task;  
    }  
  
    @Override  
    public void run() {  
        System.out.println("task1 is run");  
        if(task != null){  
            task.run();  
        }  
    }  
}  
  
public class Task2 implements Task{  
  
    private Task task;  
  
    public Task2() {}  
  
    public Task2(Task task){  
        this.task = task;  
    }  
  
    @Override  
    public void run() {
```

```
System.out.println("task2 is run");

if(task != null){
    task.run();
}

}

}

public class Task3 implements Task{

private Task task;

public Task3() {}

public Task3(Task task){
    this.task = task;
}

@Override
public void run() {
    System.out.println("task3 is run");
    if(task != null){
        task.run();
    }
}
}
```

以上代码是模拟了三个任务类，它们都实现了统一一个接口，并且它们都有一个构造函数，可以在它们初始化的时候就持有另一个任务类的对象引用，以方便该任务调用。

Tips：欢迎关注微信公众号：Java后端，每日推送 Java 项目技术博文。

这个和服务器的过滤器有点类似，过滤器的实现也都是实现了同一个接口Filter。

```
public class LiabilityChain {

public void runChain(){
    Task task3 = new Task1();
    Task task2 = new Task2(task3);
    Task task1 = new Task3(task2);
    task1.run();
}

}
```

上面这段代码就是一个任务链对象，它要做的事情很简单，就是将所有要执行的任务都按照指定的顺序串联起来。

```
public class ChainTest{  
  
    public static void main(String[] args) {  
        LiabilityChain chain = new LiabilityChain();  
        chain.runChain();  
    }  
  
}
```

当我们获取到责任链对象之后，调用其方法，得到以下运行结果：

```
task1 is run  
task2 is run  
task3 is run  
|
```

以上是一个责任链的简单的实现，如果想要深入理解其思想，建议去观察一个过滤器链的执行源码。

作者：Mazin

编辑：id：javastack

<https://my.oschina.net/u/3441184/blog/889552>

【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 多模块项目实践
2. MyBatis大揭秘：Plugin 插件设计原理
3. 你知道 Spring Batch 吗？

[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！



前言：

早就有整理历史文章的打算，一直没有时间。趁着今天晚上有空，便把我认为优质的文章整理了出来。如果能有一篇文章帮到你，请把公众号推荐给自己的同学或者同事吧。同时置顶标星公众号，可以第一时间接受到推送。

公众号简介：

Java后端：没错，这个公众号的名字就叫做 **Java后端**。专注于 Java 技术，包括 JavaWeb, SSM, Linux, Spring Boot, MyBatis, MySQL, Nginx, Git, GitHub, Servlet, IDEA, 多线程, 集合, JVM, DeBug, Dubbo, Redis, 算法, 面试题等相关内容。

不管你是在校大学生，还是软件工程师，在这个公众号都能得到你想要的文章和资源。公众号的特色是发布的文章**图文结合，生动形象，并且提供源码**。

历史发布过包括实战、分库分表、微服务实战、单点登陆、**支付宝支付、微信登陆、微信支付、QQ登陆对接**、前后端分离、权限控制、短信发送等实战，图文并茂且提供源码。

扫描下方二维码，关注 **Java后端**



热门文章：

[这种 Github 不要写在简历上 \(阅读量5.8K\)](#)

[推荐 8 个常用 Spring Boot 项目 \(阅读量5.7K\)](#)

你用什么软件做笔记？（阅读量5.7k）

Git使用教程（5.4k）！

一千行 MySQL 学习笔记（5.4k）

这简历一看就是包装过的（5.3k）

堪称神器的 Chrome 插件（5.3k）

华为外包程序员跳楼，难道这是我的35（5.3k）

扫码登陆实现原理（4.9k）

微信扫码登陆实战（附代码）4.7k

7分钟实现 Java 发送短信功能（4.6k）

你还在从零搭建项目（4.6k）

最近面试 Java 后端开发的感受（4.5k）

MyBatis 动态SQL4.4k

一键脱衣的 AI 软件（4.3k）

推荐一款小巧、提高效率的软件（4.3k）

IntelliJ IDEA 2019 从入门到上瘾 图文教程（4.3k）

IEEE下令清理华为系审稿人（4.3k）

为什么要放弃 JSP（4.2k）

Java 工程师成神之路（4.1k）

69道 Spring 面试题及答案（4.2k）

Spring Boot+MyBatis+MySQL读写分离（4.1k）

只要学会它，再多 Bug 也不怕（4.1k）

如何设计user 表？加入第三方登录呢（4.1k）

Spring Boot 实战 - 打造私人云盘（4.0k）

GitHub官宣：私有仓库免费（4.0k）

什么是红黑树？面试必问（3.9k）

一张图看懂 SQL 的各种 join 用法（3.8k）

Nginx+Tomcat实现动静分离（3.7k）

Web登录很简单？开玩笑（3.7k）

为什么很多公司不喜欢招培训机构出来的（3.7k）

这些 Spring 中的设计模式，你都知道吗（3.7k）

两个月拿到阿里的 Offer，经验分享（3.7k）

使用了 Eclipse 10 年之后，我终于投向了 IDEA（3.7k）

有 Bug 不会调试？这篇文章很详细（3.7k）

盘点阿里巴巴 15 款开发者工具 (3.6k)

如何实现登录、URL和页面按钮的访问控制 (3.6k)

Win10 引入真 Linux 内核 (3.6k)

Apache-Commons家族的八兄弟 (上) 3.6k

Java SSM框架基础面试题 (3.6k)

8 张图理解Java (3.6k)

两个值得Star的GitHub仓库 (3.5k)

各类面试题汇总，这些搞懂了，何愁无Offer (3.5k)

Hi，一起学Vue.js吗 (3.5k)

为什么不推荐使用 select * ? (3.5k)

恕我直言，在座的各位根本写不好Java (3.4k)

Cookie、Session、Token那点事儿 (3.4k)

20 分钟梳理 Spring 全家桶 (3.3k)

没有项目经验 怎么办 (3.3k)

Spring Boot 注解：全家桶快速通 (3.3k)

为什么阿里巴巴禁止使用 isSuccess 作为变量名 (3.3k)

外行人都能看懂的 Spring Cloud，错过了血亏 (3.3k)

两小时入门 Docker (3.2k)

Java 必须掌握的 20+ 种 Spring 常用注解 (3.2k)

Nginx是什么？能干嘛？ (3.2k)

自增 ID 用完了怎么办 (3.2k)

想和你聊聊前端分离 (3.2k)

我建议你尽早进入大厂的 6 个理由 (3.2k)

字节跳动、腾讯后台开发面经分享 (3.2k)

Maven 的这 7 个问题你思考过没有 (3.1k)

MyBatis 使用的 9 种设计模式，真是太有用了 (3.1k)

减少该死的 if else 嵌套 (3.1k)

算法成神之路，请看这一篇 (3.1k)

什么是 SQL 注入？怎么进行？如何防范 (3.1k)

张小龙：PC时代，流量最大的页面是哪个 (3.0k)

IntelliJ IDEA 常用配置详细图解 (3.0k)

闭关修炼 5 个月的源码，终于拿到蚂蚁 Offer (3.0k)

通俗易懂讲解 Java 线程安全 (3.0k)

Java安全框架「shiro」3.0k

一文教会你如何搭建个人博客（3.0k）

这 10 款插件让你的 GitHub 更好用、更有趣（3.0k）

使用自定义注解校验用户是否登录（2.9k）

推荐几个IDEA插件，Java开发者撸码利器（2.9k）

面试必备：30 个 Java 集合面试问题及答案（2.9k）

你是一直认为 count(1) 比 count(*) 效率高么（2.9k）

IntelliJ IDEA 吐血推荐这几个插件（2.9k）

MyBatis框架教程「模糊查询」2.9k

为什么要 前后端分离（2.8k）

TCP为什么要三次握手（2.8k）

Spring Boot 快速整合 MyBatis（2.8k）

Linux养成计划（六）2.8k

JavaWeb项目部署到云服务器教程（2.8k）

几张趣图带你了解程序员眼中的世界（2.8k）

什么是微服务（2.8k）

什么是消息队列（2.8k）

如何设计第三方账号登陆（2.8k）

浅谈JavaWeb项目代码如何分层（2.8k）

到底是 Java 好还是 Python 好（2.8k）

你知道 Spring Boot 他爹有多大背景吗（2.8k）

Spring Boot 为什么这么火火火火火（2.8k）

Dubbo入门-搭建一个最简单的Demo框架（2.8k）

再有人问你Java内存模型是什么，就把这篇文章发给他！

Nginx 反向代理、负载均衡图文教程！

如何在面试中介绍自己的项目经验？

理解 IntelliJ IDEA 的项目配置和 Web 部署

我们能从 IntelliJ IDEA 中学到什么？

想了解Java后端学习路线？你只需要这一张图！

45 个值得收藏的 CSS 形状

从头搭建 IntelliJ IDEA 环境

你知道 MyBatis 分页插件的原理吗？

小白也能看懂，30分钟搭建个人博客！

面试前你必须知道的三个排序算法

你的项目应该如何正确分层？

IDEA中用好Lombok，撸码效率至少提升5倍

Java程序员必备的IntelliJ插件

购书福利：

除此之外，此公众号还有正版书籍全场6折读者专属福利，关注公众号「Java后端」然后回复「购书」即可进入书店，或者关注后点击底部菜单栏「低价购书」。



见面礼

作为见面礼，这里有很多套源码送给新关注的读者，添加小编的微信即可免费获取：



↓ 扫码添加小编微信



久一



扫一扫上面的二维码图案，加我微信

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！

Java 实现 QQ 登陆

我是小茗同学 Java后端 2019-09-11

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

1. 前言

个人网站最近增加了评论功能,为了方便用户不用注册就可以评论,对接了 QQ 和微博这 2 大常用软件的一键登录,总的来说其实都挺简单的,可能会有一点小坑,但不算多,完整记录下来方便后来人快速对接。

2. 后台设计

在真正开始对接之前,我们先来聊一聊后台的方案设计。既然是对接第三方登录,那就免不了如何将用户信息保存。首先需要明确一点的是,用户在第三方登录成功之后,我们能拿到的仅仅是一个代表用户唯一身份的ID(微博是真实uid,QQ是加密的openId)以及用来识别身份的accessToken,当然还有昵称、头像、性别等有限资料,对接第三方登录的关键就是如何确定用户是合法登录,如果确定这次登录的和上次登录的是同一个人并且不是假冒的。

其实这个并不用我们特别操心,就以微博登录为例,用户登录成功之后会回调一个code 给我们,然后我们再拿code去微博那换取accessToken,如果这个code是用户乱填的,那这一关肯定过不了,所以,前面的担心有点多余。

另外一个问题是如果和现有用户系统打通,有的网站在用户已经登录成功之后还要用户输入手机号和验证码,或者要用户重新注册账号和密码来绑定第三方账户,感觉这种实现用户体验非常差,碰到这种网站我一般都是直接关掉,都已经登录了还让用户注册,什么鬼!由于我做的是评论功能,我并不希望评论用户和现有用户表打通,所以就不存在这件事了,如果想打通的话,我觉得无非就是登录成功之后默认往老用户表插入一条数据,然后和OpenUser表关联起来,判断用户是否登录时把OpenUser的鉴权也加进去就OK了。

本文的后台以Java为例。

2.1. 数据库设计

再来说说**数据库**设计,为了系统的扩展性,我有一个专门的OpenUser表用来存放第三方登录用户,主要字段如下:

字段名	字段含义
id	主键
openType	第三方类型, 比如 qq 、 weibo
openId	代表用户唯一身份的ID
accessToken	调用接口需要用到的token, 比如利用accessToken发表微博等, 如果只是对接登录的话, 这个其实没啥用
expiredTime	授权过期时间, 第三方登录授权都是有过期时间的, 比如3个月
nickname	昵称
avatar	头像

这样设计理论上就可以无限扩展了。

2.2. 鉴权流程

这里我只是说说我的方案，把accessToken写入cookie肯定是不安全的，因为accessToken相當于第三方网站的临时密码，被别人窃取了就可以随意拿来干坏事了。可以在用户登录成功之后我们自己生成一个token，这样的token即使泄露了顶多就是被人拿来随意评论，损失不大，但是如果accessToken被泄露了，以微博为例，人家可以利用这个accessToken随意发微博、删微博、加关注等等，很危险。当然，如果不想token泄露的话也可以通过绑定IP等方式来限制。

鉴权的话就是首先判断cookie中是否有我们自己的token，然后判断是否合法，合法再判断第三方授权是否已过期等等。

QQ登陆

3.1. 实名认证

QQ登录我们对接的是QQ互联，地址：<https://connect.qq.com>，首先需要注册成为开发者并实名认证，需要手持身份证照片，具体就不讲了。

3.2. 创建应用

进入应用管理页面(<https://connect.qq.com/manage.html#/>)创建应用，根据实际需要是创建网站应用还是移动应用，我这里是网站应用：



第一步：

其他

其他

网站名称

必须和备案登记的网站名称一模一样，否则审核一定被拒

网站简介，200字以内

这里随便填就可以

我已阅读并已同意《开发者协议》

创建应用

第二步：

http://[REDACTED]

网站域名



http://[REDACTED]

登录成功回调地址，必须是
主域名下的某一个地址



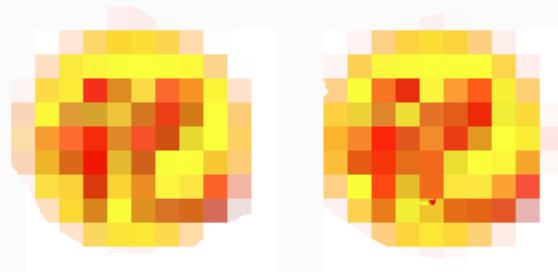
[REDACTED]

网站所有者，必须和备案登记的名称
一模一样，否则审核不会通过



粤ICP备[REDACTED]

备案号，网站没有备案貌似不能申请



一张64*64，一张100*100
的icon图标

创建应用

提交完之后会自动提交审核，基本上就是审核你的资料和备案的资料是否一致，所有资料必须和备案资料一模一样，否则审核不会通过：

恭喜您， 创建应用成功！

我们会在七个工作日内完成资料的审核：)

前往[应用管理](#)

当然，这些资料后面还是可以修改的。申请成功之后你会得到appId和appKey。

3.3. 引导用户登录

这里可以下载一些视觉素材，在页面合适位置放一个QQ登录按钮，点击时引导用户进入授权页面：



游客

各位程序猿大大和程序媛妹妹，来都来了，那就说点什么吧~

使用社交账号登录



7条评论



低调程序员

啦啦啦，我是个卖报的小行家 😊

11月14日 ← 回复 ❤ 顶(0)

代码：

```
1 function openWindow(url, width, height)
2 {
3     width = width || 600
4 ;
5     height = height || 400
6 ;
7     var left = (window.screen.width - width) / 2
8 ;
9     var top = (window.screen.height - height) / 2
10 ;
11    window.open(url, "_blank", "toolbar=yes, location=yes, directories=no, status=no, menubar=yes, scrollbars=yes"
12 ;
13 }
14
15 function qqLogin()
16 {
17     var qqAppId = '424323422'; // 上面申请得到的appi
18     d
19     var qqAuthPath = 'http://www.test.com/auth'; // 前面设置的回调地
20    址
21     var state = 'fjdslfjsdlkfd'; // 防止CSRF攻击的随机参数，必传，登录成功之后会回传，最好后台自己生成然后校验合法
22    性
23     openWindow(`https://graph.qq.com/oauth2.0/authorize?response_type=token&client_id=${qqAppId}&redirect_uri=${e
24 }
```

然后会打开一个授权页面，这个页面大家应该都熟悉：



然后到了这里我就碰到一个问题了，官方文档(<https://wiki.connect.qq.com>)写的是登录成功之后首先会回传一个code，然后拿code调接口换取accessToken，然后我试了很多次也换过2个账号发现每次都是直接返回了accessToken，帮我省了一步了，不知道是什么情况，郁闷。[微信搜索 Web 项目聚集地 获取更多实战教程。](#)

3.4. 拿到accessToken

现在假设我们都是直接拿到accessToken(因为我暂时还没搞明白QQ为啥会直接返回，跟文档说的不一样)，但是授权回调时 accessToken会被放在 # 后面，URL地址中的hash值好像不会被传到后台(貌似是这样，如有不正确欢迎评论指正)，所以只能写一个下面这样的临时页面：

```
1  @RequestMapping("/authqq")
2  public void authQQ(HttpServletRequest request, HttpServletResponse response) throws Exception
3  {
4      // QQ登录有点特殊，参数放在#后面，后台无法获取#后面的参数，只能用JS做中间转换
5      String html =    "<!DOCTYPE html>"
6      +
7      "    <html lang=\"zh-cn\">"
8      +
9      "        <head>"
10     +
11     "            <title>QQ登录重定向页</title>"
12     +
13     "            <meta charset=\"utf-8\"/>"
14     +
15     "        </head>"
16     +
17     "        <body>"
18     +
19     "            <script type=\"text/javascript\">"
20     +
21     "                location.href = location.href.replace('#', '&').replace('auth_qq', 'auth_qq_redirect');"
22     +
23
24     </script>
25     +
26 }
```

```
"</body>"
```

```
+  
    "</html>"  
;  
    response.getWriter().print(html);  
}
```

3.5. 获取openId

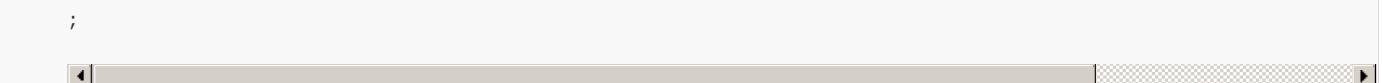
根据accessToken调接口获取用户的openId, 特别注意这个openId是相对于QQ号+appId唯一的, 换句话说同一个QQ号登录2个不同appId时获取到的openId是不同的。顺便说一句, QQ登录的相关接口做的还真够“随便”的, 全部都是最简单的get请求, 所以对接起来非常顺利。微信搜索 Web项目聚集地 获取更多实战教程。

直接看代码:

```
1 // 根据accessToken换取openId  
2 // 错误示例: callback( {"error":100016,"error_description":"access token check failed"} );  
3 // 正确示例: callback( {"client_id":"10XXXXX49","openid":"CF2XXXXXXXX9F4C"} );  
4 String result = HttpsUtil.get("https://graph.qq.com/oauth2.0/me?access_token=" + accessToken);  
5 Map<String, Object> resp = parseQAuthResponse(result); // 这个方法就是把结果转Map  
6 // 欢迎关注 Web项目聚集地 获取更多实战教程  
7 Integer errorCode = (Integer)resp.get("error")  
8 ;  
9 String errorMsg = (String)resp.get("error_description");  
10 String openId = (String)resp.get("openid")  
;  
if(errorCode != null) return new ErrorResult(errorCode, "获取QQ用户openId失败: "+errorMsg)  
;
```

3.6. 获取用户头像昵称等信息

```
1 // 获取用户昵称、头像等信息, {ret: 0, msg: '', nickname: ''}, ...} ret不为0表示失败  
2 result = HttpsUtil.get("https://graph.qq.com/user/get_user_info?access_token="+accessToken+"&oauth_consumer_key="+  
3 resp = JsonUtil.parseJsonToMap(result);  
4 // 欢迎关注 Web项目聚集地 获取更多实战教程  
5 Integer ret = (Integer)resp.get("ret")  
6 ;  
7 String msg = (String)resp.get("msg")  
8 ;  
9 if(ret != 0) return new ErrorResult("获取用户QQ信息失败: "+msg)  
10 ;  
11 // 用户昵称可能存在4个字节的utf-8字符, MySQL默认不支持, 直接插入会报错, 所以过滤掉  
12 String nickname = StringUtil.filterUtf8Mb4((String)resp.get("nickname")).trim(); // 这个方法可以自行百度  
13 // figureurl_qq_2=QQ的100*100头像, figureurl_2=QQ 100&100空间头像, QQ头像不一定有, 空间头像一定有  
14 String avatar = (String)resp.get("figureurl_qq_2")  
;  
if(StringUtil.isBlank(avatar)) avatar = (String)resp.get("figureurl_2")  
;  
String gender = (String)resp.get("gender")
```



3.7. 注意事项

到了这一步基本上涉及第三方的就结束了，是不是很简单？后面无非就是如何插入数据库、如何保存token、写入session等。

有几点注意事项：

- 需要注意数据库中是否已经有该用户，没有的添加，有的修改，不要重复添加了；
- QQ昵称有各种奇奇怪怪的字符，包括emoji，MySQL默认没有开启utf8mb4，直接插入会报错，所以需要过滤掉；
- 需要做好对各种错误的兼容；
- 接口会同时返回QQ头像和空间头像，QQ头像不一定有，空间头像一定有；
- 回调地址必须和申请的域名一致，否则会报错。
- QQ互联有个特大的bug，有时候显示已登录但是点击授权管理一直报错，此时只需要退出重新登录即可；
- 授权之后用户可能会在过期之前提前取消授权；
- 微信搜索 Web项目聚集地 获取更多实战教程。

相关文档官网已经写得比较细了，但是比较乱：<http://wiki.connect.qq.com/>

对接微博登陆

4.1. 实名认证

这个我就不具体讲了，登录 <http://open.weibo.com/> 很容易找到相关入口，注册成为开发者，实名认证，一模一样的。

4.2. 创建应用

点击链接 <http://open.weibo.com/apps/new?sort=web> 创建web应用：



创建成果后完善相关信息，主要是下面这些：

应用地址 :	<input type="text"/>	含有微博组件的网站主页地址链接。
(必填) 应用简介 :	<input type="text"/>	用于行为动态模块中应用介绍，应用授权页等，不超过15个汉字
(必填) 应用介绍 :	<input type="text"/>	
安全域名 :	<input checked="" type="radio"/> 是 <input type="radio"/> 否	
(必填) 标签 :	<input type="text"/>	标签最多是三个
官方运营帐号 :	<input checked="" type="radio"/> 使用当前微博帐号 <input type="radio"/> 使用其他微博帐号	应用将以此关联的官方运营帐号名义向用户发送通知
(必填) 应用图标16*16 :	<input type="button" value="选择文件"/>	16*16 , 2M以内 , 支持PNG、JPG
(必填) 应用图标80*80 :	<input type="button" value="选择文件"/>	80*80 , 2M以内 , 支持PNG、JPG
(必填) 应用图标120*120 :	<input type="button" value="选择文件"/>	120*120 , 2M以内 , 支持PNG、JPG
(必填) 应用介绍图片 :	<input type="button" value="添加图片1"/> <input type="button" value="添加图片2"/> <input type="button" value="添加图片3"/>	用于应用频道推广展示 至少上传三张图片 , 2M以内 , 支持PNG、JPG 高 : 300px 宽 : 450px ; 示意图 

我就不一一介绍了，都看得懂。

微博登录不需要网站一定要备案，但对网站本身有一定要求，不能弄一个空壳网站让人家去审核，肯定审核不通过的。

有关微博的对接可以参考我好几年前写的一篇文章：

<http://www.cnblogs.com/liuxianan/archive/2012/11/11/2765123.html>

4.3. 引导用户登录

微博视觉素材(<https://open.weibo.com/wiki/微博标识下载>)下载在这里，页面合适位置放一个登录按钮：

```

1 function weiboLogin()
2 {
3     let weiboAppId = '432432'
4 ;
5     let weiboAuthPath = 'http://www.test.com/authweibo'
6 ;
7     openWindow(`https://api.weibo.com/oauth2/authorize?client_id=${weiboAppId}&response_type=code&redirect_uri=${e
8 }

```

微博登录有一个好处，第一次登录需要授权，后面第二次登录时只会一闪而过自动就登录成功了，都不需要点一下，用户体验非常好，看下图：



游客

各位程序猿大大和程序媛妹妹，来都来了，那就说点什么吧~

使用社交账号登录

7条评论



低调程序员

啦啦啦，我是个卖报的小行家 😊

11月14日 ← 回复 ❤️ 顶(0)



小茗同学

板凳 😊

11月14日 ← 回复 ❤️ 顶(0)



小茗同学

测试楼中楼回复 😊

11月14日 ← 回复 ❤️ 顶(0)

4.4. 获取accessToken

登录成功会返回一个code, 根据code换取accessToken:

```
1 String params = "client_id=" + appId
2         + "&client_secret=" + appSecret
3         + "&grant_type=authorization_code"
4         + "&redirect_uri=" + URLUtil.encode(authPath)
5         + "&code=" + code
6 ;
7 // 用code换取accessToken
8 String result = HttpsUtil.post("https://api.weibo.com/oauth2/access_token", params);
9 Map<String, Object> resp = JsonUtil.toObject(result, new TypeReference<Map<String, Object>>(){});
10
11 Integer errorCode = (Integer)resp.get("error_code")
12 ;
13 String error = (String)resp.get("error")
14 ;
15 String errorMsg = (String)resp.get("error_description");
16 if(errorCode != null && errorCode != 0) return new ErrorResult(errorCode, error + (errorMsg==null?"":errorMsg));
17 ;
18 String accessToken = (String)resp.get("access_token")
19 ;
20 String uid = (String)resp.get("uid"); // 这个uid就是微博用户的唯一用户ID, 可以通过这个id直接访问到用户微博主页
21
22 int expires = (Integer)resp.get("expires_in"); // 有效期, 单位
23 秒
```

4.5. 获取用户头像等信息

```
1 // 用uid和accessToken换取用户信息
2 String result = HttpsUtil.get("https://api.weibo.com/2/users/show.json?access_token="+accessToken+"&uid="+uid);
3 Map<String, Object> resp = JsonUtil.toObject(result, new TypeReference<Map<String, Object>>(){});
4
5 errorCode = (Integer)resp.get("error_code")
```

```
5 ;
6 error = (String)resp.get("error")
7 ;
8 errorMsg = (String)resp.get("error_description");
9 if(errorCode != null && errorCode != 0) return new ErrorResult(errorCode, error + (errorMsg==null?"":errorMsg))
10 ;
11
12 String nickname = (String)resp.get("screen_name");
13 // 微博180*180高清头像
14 String avatar = (String)resp.get("avatar_large")
;
String gender = (String)resp.get("gender")
;
gender = "m".equals(gender) ? "男" : ("f".equals(gender) ? "女" : "")
```

至此涉及第三方的东西都完了，剩下的就是用户自己保存到数据库、写入token

保存 session 以及鉴权接口开发了。

4.6. 注意事项

- 微博接口都有频率限制，不过一般不会超过；
- 需做好错误兼容；
- 微博直接返回的uid，可以根据这个uid直达用户微博主页 <https://weibo.com/u/xxxxx>，所以可以把用户头像链接到这里；
- 其实也有现成的js-sdk，可以根据自己实际需要选择是否使用；
- 微博的接口是https，并且是post，需要注意；

相关链接

- 微博开放平台:open.weibo.com/
- 微博登录授权机制:open.weibo.com/wiki/授权机制
- QQ互联:connect.qq.com/
- QQ授权管理页面:connect.qq.com/manage.html#/appauth/user

作者：我是小茗同学

排版：Web项目聚集地（web_resource）

链接：www.cnblogs.com/liuxianan.html

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. 史上最烂的项目：苦撑12年，600 多万行代码 ...
2. 请给 Spring Boot 多一些内存
3. 如何从零搭建百亿流量系统？
4. 惊了！原来 Web 发展历史是这样的



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！

Java常用代码汇总

Java后端 2019-12-23

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

1. 字符串有整型的相互转换

```
String a = String.valueOf(2); //integer to numeric string  
int i = Integer.parseInt(a); //numeric string to an int
```

2. 向文件末尾添加内容

```
BufferedWriter out = null;  
try {  
    out = new BufferedWriter(new FileWriter("filename", true));  
    out.write(" aString ");  
} catch (IOException e) {  
    //error processing code  
} finally {  
    if (out != null) {  
        out.close();  
    }  
}
```

3. 得到当前方法的名字

```
String methodName = Thread.currentThread().getStackTrace()[1].getMethodName();
```

4. 转字符串到日期

```
java.util.Date = java.text.DateFormat.getDateInstance().parse(date String);  
或者是：  
SimpleDateFormat format = new SimpleDateFormat( "yyyy-MM-dd" );  
Date date = format.parse( myString );
```

5. 使用JDBC链接Oracle

```

public class OracleJdbcTest
{
    String driverClass = "oracle.jdbc.driver.OracleDriver";
    Connection con;
    public void init(FileInputStream fs) throws ClassNotFoundException, SQLException, FileNotFoundException, IOException
    {
        Properties props = new Properties();
        props.load(fs);
        String url = props.getProperty("db.url");
        String userName = props.getProperty("db.user");
        String password = props.getProperty("db.password");
        Class.forName(driverClass);
        con=DriverManager.getConnection(url, userName, password);
    }
    public void fetch() throws SQLException, IOException
    {
        PreparedStatement ps = con.prepareStatement("select SYSDATE from dual");
        ResultSet rs = ps.executeQuery();
        while (rs.next())
        {
            // do the thing you do
        }
        rs.close();
        ps.close();
    }
    public static void main(String[] args)
    {
        OracleJdbcTest test = new OracleJdbcTest();
        test.init();
        test.fetch();
    }
}

```

6.列出文件和目录

```

File dir = new File("directoryName");
String[] children = dir.list();
if (children == null) {
    // Either dir does not exist or is not a directory
} else {
    for (int i=0; i < children.length; i++) {
        // Get filename of file or directory
        String filename = children[i];
    }
}
// It is also possible to filter the list of returned files.
// This example does not return any files that start with '..'
FilenameFilter filter = new FilenameFilter() {
    public boolean accept(File dir, String name) {
        return !name.startsWith(".");
    }
};
children = dir.list(filter);
// The list of files can also be retrieved as File objects
File[] files = dir.listFiles();
// This filter only returns directories
FileFilter fileFilter = new FileFilter() {
    public boolean accept(File file) {
        return file.isDirectory();
    }
};
files = dir.listFiles(fileFilter);

```

7.解析/读取XML文件

```
<?xml version="1.0"?>
<students>
<student>
<name>John</name>
<grade>B</grade>
<age>12</age>
</student>
<student>
<name>Mary</name>
<grade>A</grade>
<age>11</age>
</student>
<student>
<name>Simon</name>
<grade>A</grade>
<age>18</age>
</student>
</students>
```

8.java分页代码实现

```
1 public class PageBean {  
2     private int curPage; //当前页  
3     private int pageCount; //总页数  
4     private int rowsCount; //总行数  
5     private int pageSize=10; //每页多少行  
6  
7  
8  
9     public PageBean(int rows){  
10         this.setRowsCount(rows);  
11         if(this.rowsCount % this.pageSize == 0){  
12             this.pageCount=this.rowsCount / this.pageSize;  
13         }  
14         else if(rows<this.pageSize){  
15             this.pageCount=1;  
16         }  
17         else{  
18             this.pageCount=this.rowsCount / this.pageSize +1;  
19         }  
20     }  
21 }  
22  
23  
24     public int getCurPage(){  
25         return curPage;  
26     }  
27     public void setCurPage(int curPage) {  
28         this.curPage = curPage;  
29     }  
30     public int getPageCount() {  
31         return pageCount;  
32     }  
33     public void setPageCount(int pageCount) {  
34         this.pageCount = pageCount;  
35     }  
36     public int getPageSize() {  
37         return pageSize;  
38     }  
39     public void setPageSize(int pageSize) {  
40         this.pageSize = pageSize;  
41     }  
42     public int getRowsCount(){  
43         return rowsCount;  
44     }  
45     public void setRowsCount(int rowsCount){  
46         this.rowsCount = rowsCount;  
47     }  
48 }
```

分页展示如下

```
1 List clist=adminbiz.queryNotFullCourse();//将查询结果存放在List集合里
2 PageBean pagebean=new PageBean(clist.size());//初始化PageBean对象
3 //设置当前页
4 pagebean.setCurPage(page); //这里page是从页面上获取的一个参数，代表页数
5 //获得分页大小
6 int pagesize=pagebean.getPageSize();
7 //获得分页数据在list集合中的索引
8 int firstIndex=(page-1)*pagesize;
9 int toIndex=page*pagesize;
10 if(toIndex>clist.size()){
11     toIndex=clist.size();
12 }
13 if(firstIndex>toIndex){
14     firstIndex=0;
15     pagebean.setCurPage(1);
16 }
17 //截取数据集合，获得分页数据
18 List courseList=clist.subList(firstIndex, toIndex);
```

-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 

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！

Java 并发编程 73 道面试题及答案

Java后端 2019-12-09

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

原文地址：

https://blog.csdn.net/qq_34039315/article/details/7854931

1、在java中守护线程和本地线程区别？

java中的线程分为两种：守护线程（Daemon）和用户线程（User）。

任何线程都可以设置为守护线程和用户线程，通过方法Thread.setDaemon(bool on); true则把该线程设置为守护线程，反之则为用户线程。Thread.setDaemon()必须在Thread.start()之前调用，否则运行时会抛出异常。

两者区别：

唯一的区别是判断虚拟机(JVM)何时离开，Daemon是为其他线程提供服务，如果全部的User Thread已经撤离，Daemon 没有可服务的线程，JVM撤离。也可以理解为守护线程是JVM自动创建的线程（但不一定），用户线程是程序创建的线程；比如JVM的垃圾回收线程是一个守护线程，当所有线程已经撤离，不再产生垃圾，守护线程自然就没事可干了，当垃圾回收线程是Java虚拟机上仅剩的线程时，Java虚拟机会自动离开。

扩展：Thread Dump打印出来的线程信息，含有daemon字样的线程即为守护进程，可能会有：服务守护进程、编译守护进程、windows下的监听Ctrl+break的守护进程、Finalizer守护进程、引用处理守护进程、GC守护进程。

2、线程与进程的区别？

进程是操作系统分配资源的最小单元，线程是操作系统调度的最小单元。

一个程序至少有一个进程，一个进程至少有一个线程。

3、什么是多线程中的上下文切换？

多线程会共同使用一组计算机上的CPU，而线程数大于给程序分配的CPU数量时，为了让各个线程都有执行的机会，就需要轮转使用CPU。不同的线程切换使用CPU发生的切换数据等就是上下文切换。

4、死锁与活锁的区别，死锁与饥饿的区别？

死锁：是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

产生死锁的必要条件：

- 互斥条件：所谓互斥就是进程在某一时间内独占资源。
- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

- 不剥夺条件:进程已获得资源，在未使用完之前，不能强行剥夺。
- 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

活锁: 任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。

活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，所谓的“活”，而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。

饥饿: 一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。

Java中导致饥饿的原因：

- 高优先级线程吞噬所有的低优先级线程的CPU时间。
- 线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。
- 线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的wait方法)，因为其他线程总是被持续地获得唤醒。

5、Java中用到的线程调度算法是什么？

采用时间片轮转的方式。可以设置线程的优先级，会映射到下层的系统上面的优先级上，如非特别需要，尽量不要用，防止线程饥饿。

6、什么是线程组，为什么在Java中不推荐使用？

ThreadGroup类，可以把线程归属到某一个线程组中，线程组中可以有线程对象，也可以有线程组，组中还可以有线程，这样的组织结构有点类似于树的形式。

为什么不推荐使用？因为使用有很多的安全隐患吧，没有具体追究，如果需要使用，推荐使用线程池。

7、为什么使用Executor框架？

- I. 每次执行任务创建线程 new Thread() 比较消耗性能，创建一个线程是比较耗时、耗资源的。
- II. 调用 new Thread() 创建的线程缺乏管理，被称为野线程，而且可以无限制的创建，线程之间的相互竞争会导致过多占用系统资源而导致系统瘫痪，还有线程之间的频繁交替也会消耗很多系统资源。
- III. 接使用new Thread() 启动的线程不利于扩展，比如定时执行、定期执行、定时定期执行、线程中断等都不便实现。

8、在Java中Executor和Executors的区别？

Executors 工具类的不同方法按照我们的需求创建了不同的线程池，来满足业务的需求。

Executor 接口对象能执行我们的线程任务。

ExecutorService接口继承了Executor接口并进行了扩展，提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。

使用ThreadPoolExecutor 可以创建自定义线程池。

Future 表示异步计算的结果，他提供了检查计算是否完成的方法，以等待计算的完成，并可以使用get()方法获取计算的结果。

9、什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？

原子操作 (atomic operation) 意为”不可被中断的一个或一系列操作”。

处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。

在Java中可以通过锁和循环CAS的方式来实现原子操作。CAS操作——Compare & Set, 或是 Compare & Swap, 现在几乎所有的CPU指令都支持CAS的原子操作。

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加1时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在JDK1.5之前我们可以使用同步技术来做到这一点。到JDK1.5，java.util.concurrent.atomic包提供了int和long类型的原子包装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。

java.util.concurrent这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由JVM从等待队列中选择一个另一个线程进入，这只是一种逻辑上的理解。

原子类：AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference

原子数组：AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray

原子属性更新器：AtomicLongFieldUpdater, AtomicIntegerFieldUpdater, AtomicReferenceFieldUpdater

解决ABA问题的原子类：AtomicMarkableReference（通过引入一个boolean来反映中间有没有变过），AtomicStampedReference（通过引入一个int来累加来反映中间有没有变过）

10、Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？

Lock接口比同步方法和同步块提供了更具扩展性的锁操作。

他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

- 可以使锁更公平
- 可以使线程在等待锁的时候响应中断
- 可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间
- 可以在不同的范围，以不同的顺序获取和释放锁

整体上来说Lock是synchronized的扩展版，Lock提供了无条件的、可轮询的(tryLock方法)、定时的(tryLock带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition方法)锁操作。另外Lock的实现类基本都支持非公平锁(默认)和公平锁，synchronized只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

11、什么是Executors框架？

Executor框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

无限制的创建线程会引起应用程序内存溢出。所以创建一个线程池是个更好的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用Executors框架可以非常方便的创建一个线程池。

12、什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。

这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

JDK7提供了7个阻塞队列。分别是：

ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。

LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。

PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。

DelayQueue：一个使用优先级队列实现的无界阻塞队列。

SynchronousQueue：一个不存储元素的阻塞队列。

LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

Java 5之前实现同步存取时，可以使用普通的一个集合，然后在使用线程的协作和线程同步可以实现生产者，消费者模式，主要的技术就是用好，`wait` ,`notify`,`notifyAll`,`synchronized`这些关键字。而在java 5之后，可以使用阻塞队列来实现，此方式大大减少了代码量，使得多线程编程更加容易，安全方面也有保障。

BlockingQueue接口是Queue的子接口，它的主要用途并不是作为容器，而是作为线程同步的工具，因此他具有一个很明显的特性，当生产者线程试图向BlockingQueue放入元素时，如果队列已满，则线程被阻塞，当消费者线程试图从中取出一个元素时，如果队列为空，则该线程会被阻塞，正是因为它所具有这个特性，所以在程序中多个线程交替向BlockingQueue中放入元素，取出元素，它可以很好的控制线程之间的通信。

阻塞队列使用最经典的场景就是socket客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。

13、什么是Callable和Future？

Callable接口类似于Runnable，从名字就可以看出来了，但是Runnable不会返回结果，并且无法抛出返回结果的异常，而Callable功能更强大一些，被线程执行后，可以返回值，这个返回值可以被Future拿到，也就是说，Future可以拿到异步执行任务的返回值。

可以认为是带有回调的Runnable。

Future接口表示异步任务，是还没有完成的任务给出的未来结果。所以说Callable用于产生结果，Future用于获取结果。

14、什么是FutureTask?使用ExecutorService启动任务。

在Java并发程序中FutureTask表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成get方法将会阻塞。一个FutureTask对象可以对调用了Callable和Runnable的对象进行包装，由于FutureTask也是调用了Runnable接口所以它可以提交给Executor来执行。

15、什么是并发容器的实现？

何为同步容器：可以简单地理解为通过synchronized来实现同步的容器，如果有多个线程调用同步容器的方法，它们将会串行执行。比如Vector，Hashtable，以及Collections.synchronizedSet，synchronizedList等方法返回的容器。

可以通过查看Vector，Hashtable等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字synchronized。

并发容器使用了与同步容器完全不同的加锁策略来提供更高的并发性和伸缩性，例如在ConcurrentHashMap中采用了一种粒度更细的加锁机制，可以称为分段锁，在这种锁机制下，允许任意数量的读线程并发地访问map，并且执行读操作的线程和写操作的线程也可以并发的访问map，同时允许一定数量的写操作线程并发地修改map，所以它可以在并发环境下实现更高的吞吐量。

16、多线程同步和互斥有几种实现方法，都是什么？

线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。

线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

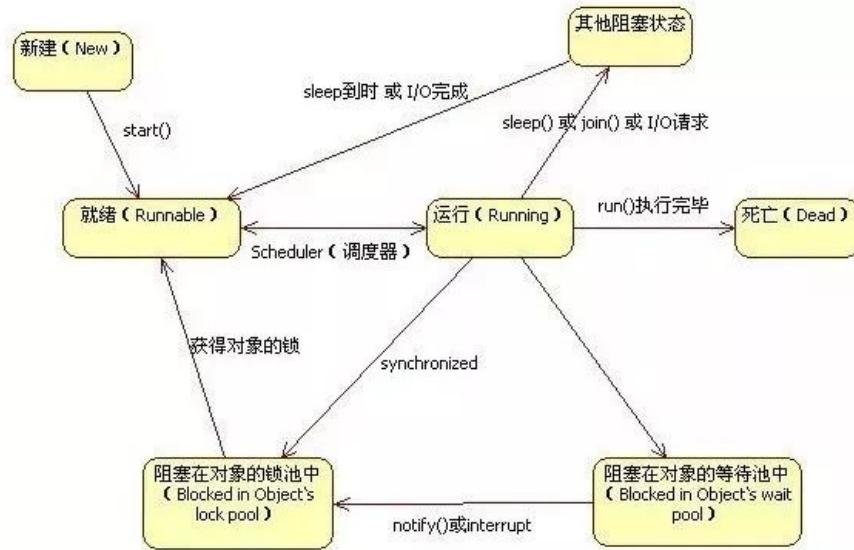
线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。

用户模式下的方法有：原子操作（例如一个单一的全局变量），临界区。内核模式下的方法有：事件，信号量，互斥量。

17、什么是竞争条件？你怎样发现和解决竞争？

当多个进程都企图对共享数据进行某种处理，而最后的结果又取决于进程运行的顺序时，则我们认为这发生了竞争条件（race condition）。

18、你将如何使用thread dump？你将如何分析Thread dump？



● 新建状态 (New)

用new语句创建的线程处于新建状态，此时它和其他Java对象一样，仅仅在堆区中被分配了内存。

● 就绪状态 (Runnable)

当一个线程对象创建后，其他线程调用它的start()方法，该线程就进入就绪状态，Java虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得CPU的使用权。

● 运行状态 (Running)

处于这个状态的线程占用CPU，执行程序代码。只有处于就绪状态的线程才有机会转到运行状态。

● 阻塞状态 (Blocked)

阻塞状态是指线程因为某些原因放弃CPU，暂时停止运行。当线程处于阻塞状态时，Java虚拟机不会给线程分配CPU。直到线程重新进入就绪状态，它才有机会转到运行状态。

阻塞状态可分为以下3种：

- ① 位于对象等待池中的阻塞状态 (Blocked in object's wait pool) : 当线程处于运行状态时，如果执行了某个对象的 **wait()** 方法，Java虚拟机就会把线程放到这个对象的等待池中，这涉及到“线程通信”的内容。
- ② 位于对象锁池中的阻塞状态 (Blocked in object's lock pool) : 当线程处于运行状态时，试图获得某个对象的同步锁时，如果该对象的同步锁已经被其他线程占用，Java虚拟机就会把这个线程放到这个对象的锁池中，这涉及到“线程同步”的内容。
- ③ 其他阻塞状态 (Otherwise Blocked) : 当前线程执行了 **sleep()** 方法，或者调用了其他线程的 **join()** 方法，或者发出了 I/O 请求时，就会进入这个状态。

● 死亡状态 (Dead)

当线程退出run()方法时，就进入死亡状态，该线程结束生命周期。

我们运行之前的那个死锁代码SimpleDeadLock.java，然后尝试输出信息(*这是注释，作者自己加的*)：

```

/* 时间, jvm信息 */
2017-11-01 17:36:28
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.144-b01 mixed mode):

```

/* 线程名称: DestroyJavaVM
编号: #13
优先级: 5
系统优先级: 0
jvm内部线程id: 0x0000000001c88800
对应系统线程id (NativeThread ID) : 0x1c18
线程状态: waiting on condition [0x0000000000000000] (等待某个条件)
线程详细状态: java.lang.Thread.State: RUNNABLE 及之后所有*/
"DestroyJavaVM" #13 prio=5 os_prio=0 tid=0x0000000001c88800 nid=0x1c18 waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"Thread-1" #12 prio=5 os_prio=0 tid=0x0000000018d49000 nid=0x17b8 waiting for monitor entry [0x0000000019d7f000]
/* 线程状态: 阻塞 (在对象同步上)
代码位置: at com.leo.interview.SimpleDeadLock\$B.run(SimpleDeadLock.java:56)
等待锁: 0x00000000d629b4d8
已经获得锁: 0x00000000d629b4e8*/
java.lang.Thread.State: BLOCKED (on object monitor)
at com.leo.interview.SimpleDeadLock\$B.run(SimpleDeadLock.java:56)
- waiting to lock <0x00000000d629b4d8> (a java.lang.Object)
- locked <0x00000000d629b4e8> (a java.lang.Object)

"Thread-0" #11 prio=5 os_prio=0 tid=0x0000000018d44000 nid=0x1ebc waiting for monitor entry [0x000000001907f000]
java.lang.Thread.State: BLOCKED (on object monitor)
at com.leo.interview.SimpleDeadLock\$A.run(SimpleDeadLock.java:34)
- waiting to lock <0x00000000d629b4e8> (a java.lang.Object)
- locked <0x00000000d629b4d8> (a java.lang.Object)

"Service Thread" #10 daemon prio=9 os_prio=0 tid=0x0000000018ca5000 nid=0x1264 runnable [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"C1 CompilerThread2" #9 daemon prio=9 os_prio=2 tid=0x0000000018c46000 nid=0xb8c waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #8 daemon prio=9 os_prio=2 tid=0x0000000018be4800 nid=0xdb4 waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #7 daemon prio=9 os_prio=2 tid=0x0000000018be3800 nid=0x810 waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"Monitor Ctrl-Break" #6 daemon prio=5 os_prio=0 tid=0x0000000018bcc800 nid=0x1c24 runnable [0x00000000193ce000]
java.lang.Thread.State: RUNNABLE
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
at java.net.SocketInputStream.read(SocketInputStream.java:171)
at java.net.SocketInputStream.read(SocketInputStream.java:141)
at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
- locked <0x00000000d632b928> (a java.io.InputStreamReader)
at java.io.InputStreamReader.read(InputStreamReader.java:184)
at java.io.BufferedReader.fill(BufferedReader.java:161)
at java.io.BufferedReader.readLine(BufferedReader.java:324)
- locked <0x00000000d632b928> (a java.io.InputStreamReader)
at java.io.BufferedReader.readLine(BufferedReader.java:389)
at com.intellij.rt.execution.application.AppMainV2\$1.run(AppMainV2.java:64)

"Attach Listener" #5 daemon prio=5 os_prio=2 tid=0x0000000017781800 nid=0x524 runnable [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os_prio=2 tid=0x000000001778f800 nid=0x1b08 waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"Finalizer" #3 daemon prio=8 os_prio=1 tid=0x000000001776a800 nid=0xdac in Object.wait() [0x0000000018b6f000]
java.lang.Thread.State: WAITING (on object monitor)

```
at java.lang.Object.wait(Native Method)
- waiting on <0x00000000d6108ec8> (a java.lang.ref.ReferenceQueue$Lock)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:143)
- locked <0x00000000d6108ec8> (a java.lang.ref.ReferenceQueue$Lock)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:164)
at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)
```

```
"Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x0000000017723800 nid=0x1670 in Object.wait() [0x00000000189ef000]
java.lang.Thread.State: WAITING (on object monitor)
at java.lang.Object.wait(Native Method)
- waiting on <0x00000000d6106b68> (a java.lang.ref.Reference$Lock)
at java.lang.Object.wait(Object.java:502)
at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
- locked <0x00000000d6106b68> (a java.lang.ref.Reference$Lock)
at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)
```

```
"VM Thread" os_prio=2 tid=0x000000001771b800 nid=0x604 runnable
```

```
"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x000000001c9d800 nid=0x9f0 runnable
```

```
"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x000000001c9f000 nid=0x154c runnable
```

```
"GC task thread#2 (ParallelGC)" os_prio=0 tid=0x0000000001ca0800 nid=0xcd0 runnable
```

```
"GC task thread#3 (ParallelGC)" os_prio=0 tid=0x0000000001ca2000 nid=0x1e58 runnable
```

```
"VM Periodic Task Thread" os_prio=2 tid=0x0000000018c5a000 nid=0x1b58 waiting on condition
```

```
JNI global references: 33
```

```
/* 此处可以看待死锁的相关信息！ */
```

```
Found one Java-level deadlock:
```

```
=====
"Thread-1":
```

```
waiting to lock monitor 0x0000000017729fc8 (object 0x00000000d629b4d8, a java.lang.Object),
which is held by "Thread-0"
```

```
"Thread-0":
```

```
waiting to lock monitor 0x0000000017727738 (object 0x00000000d629b4e8, a java.lang.Object),
which is held by "Thread-1"
```

```
Java stack information for the threads listed above:
```

```
=====
"Thread-1":
```

```
at com.leo.interview.SimpleDeadLock$B.run(SimpleDeadLock.java:56)
- waiting to lock <0x00000000d629b4d8> (a java.lang.Object)
- locked <0x00000000d629b4e8> (a java.lang.Object)
```

```
"Thread-0":
```

```
at com.leo.interview.SimpleDeadLock$A.run(SimpleDeadLock.java:34)
- waiting to lock <0x00000000d629b4e8> (a java.lang.Object)
- locked <0x00000000d629b4d8> (a java.lang.Object)
```

```
Found 1 deadlock.
```

```
/* 内存使用状况，详情得看JVM方面的书 */
```

```
Heap
```

```
PSYoungGen total 37888K, used 4590K [0x00000000d6100000, 0x00000000d8b00000, 0x00000000100000000)
```

```
eden space 32768K, 14% used [0x00000000d6100000, 0x00000000d657b968, 0x00000000d8100000)
```

```
from space 5120K, 0% used [0x00000000d8600000, 0x00000000d8600000, 0x00000000d8b00000)
```

```
to space 5120K, 0% used [0x00000000d8100000, 0x00000000d8100000, 0x00000000d8600000)
```

```
ParOldGen total 86016K, used 0K [0x0000000082200000, 0x0000000087600000, 0x00000000d6100000)
```

```
object space 86016K, 0% used [0x0000000082200000, 0x0000000082200000, 0x0000000087600000)
```

```
Metaspace used 3474K, capacity 4500K, committed 4864K, reserved 1056768K
```

```
class space used 382K, capacity 388K, committed 512K, reserved 1048576K
```

19、为什么我们调用start()方法时会执行run()方法，为什么我们不能直接调用run()方法？

当你调用start()方法时你将创建新的线程，并且执行在run()方法里的代码。

但是如果你直接调用run()方法，它不会创建新的线程也不会执行调用线程的代码，只会把run方法当作普通方法去执行。

20、Java中你怎样唤醒一个阻塞的线程？

在Java发展史上曾经使用suspend()、resume()方法对于线程进行阻塞唤醒，但随之出现很多问题，比较典型的还是死锁问题。

解决方案可以使用以对象为目标的阻塞，即利用Object类的wait()和notify()方法实现线程阻塞。

首先，wait、notify方法是针对对象的，调用任意对象的wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的notify()方法则将随机解除该对象阻塞的线程，但它需要重新获取改对象的锁，直到获取成功才能往下执行；其次，wait、notify方法必须在synchronized块或方法中被调用，并且要保证同步块或方法的锁对象与调用wait、notify方法的对象是同一个，如此一来在调用wait之前当前线程就已经成功获取某对象的锁，执行wait阻塞后当前线程就将之前获取的对象锁释放。

21、在Java中CyclicBarrier和CountdownLatch有什么区别？

CyclicBarrier可以重复使用，而CountdownLatch不能重复使用。

Java的concurrent包里面的CountDownLatch其实可以把它看作一个计数器，只不过这个计数器的操作是原子操作，同时只能有一个线程去操作这个计数器，也就是同时只能有一个线程去减这个计数器里面的值。

你可以向CountDownLatch对象设置一个初始的数字作为计数值，任何调用这个对象上的await()方法都会阻塞，直到这个计数器的计数值被其他的线程减为0为止。

所以在当前计数到达零之前，await方法会一直受阻塞。之后，会释放所有等待的线程，await的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。如果需要重置计数，请考虑使用 CyclicBarrier。

CountDownLatch的一个非常典型的应用场景是：有一个任务想要往下执行，但必须要等到其他的任务执行完毕后才可以继续往下执行。假如我们这个想要继续往下执行的任务调用一个CountDownLatch对象的await()方法，其他的任务执行完自己的任务后调用同一个CountDownLatch对象上的countDown()方法，这个调用await()方法的任务将一直阻塞等待，直到这个CountDownLatch对象的计数值减到0为止

CyclicBarrier一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环的 barrier。

22、什么是不可变对象，它对写并发应用有什么帮助？

不可变对象(Immutable Objects)即对象一旦被创建它的状态(对象的数据，也即对象属性值)就不能改变，反之即为可变对象(Mutable Objects)。

不可变对象的类即为不可变类(Immutable Class)。Java平台类库中包含许多不可变类,如String、基本类型的包装类、BigInteger和BigDecimal等。

不可变对象天生是线程安全的。它们的常量（域）是在构造函数中创建的。既然它们的状态无法修改，这些常量永远不会变。

不可变对象永远是线程安全的。

只有满足如下状态，一个对象才是不可变的；

- 它的状态不能在创建后再被修改；
- 所有域都是final类型；并且，
- 它被正确创建（创建期间没有发生this引用的逸出）。

23、什么是多线程中的上下文切换？

在上下文切换过程中，CPU会停止处理当前运行的程序，并保存当前程序运行的具体位置以便之后继续运行。从这个角度来看，上下文切换有点像我们同时阅读几本书，在来回切换书本的同时我们需要记住每本书当前读到的页码。在程序中，上下文切换过程中的“页码”信息是保存在进程控制块（PCB）中的。PCB还经常被称作“切换桢”（switchframe）。 “页码”信息会一直保存到CPU的内存中，直到他们被再次使用。

上下文切换是存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

24、Java中用到的线程调度算法是什么？

计算机通常只有一个CPU,在任意时刻只能执行一条机器指令,每个线程只有获得CPU的使用权才能执行指令.所谓多线程的并发运行,其实是指从宏观上看,各个线程轮流获得CPU的使用权,分别执行各自的任务.在运行池中,会有多个处于就绪状态的线程在等待CPU,JAVA虚拟机的一项任务就是负责线程的调度,线程调度是指按照特定机制为多个线程分配CPU的使用权.

有两种调度模型：分时调度模型和抢占式调度模型。

分时调度模型是指让所有的线程轮流获得cpu的使用权,并且平均分配每个线程占用的CPU的时间片这个也比较好理解。

java虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用CPU。处于运行状态的线程会一直运行，直至它不得不放弃CPU。

25、什么是线程组，为什么在Java中不推荐使用？

线程组和线程池是两个不同的概念，他们的作用完全不同，前者是为了方便线程的管理，后者是为了管理线程的生命周期，复用线程，减少创建销毁线程的开销。

26、为什么使用Executor框架比使用应用创建和管理线程好？

为什么要使用Executor线程池框架

- I. 每次执行任务创建线程 new Thread()比较消耗性能，创建一个线程是比较耗时、耗资源的。
- II. 调用 new Thread()创建的线程缺乏管理,被称为野线程,而且可以无限制的创建,线程之间的相互竞争会导致过多占用系

统资源而导致系统瘫痪，还有线程之间的频繁交替也会消耗很多系统资源。

III. 直接使用new Thread() 启动的线程不利于扩展，比如定时执行、定期执行、定时定期执行、线程中断等都不便实现。

使用Executor线程池框架的优点

- I. 能复用已存在并空闲的线程从而减少线程对象的创建从而减少了消亡线程的开销。
- II. 可有效控制最大并发线程数，提高系统资源使用率，同时避免过多资源竞争。
- III. 框架中已经有定时、定期、单线程、并发数控制等功能。

综上所述使用线程池框架Executor能更好的管理线程、提供系统资源使用率。

27、java中有几种方法可以实现一个线程？

- 继承 Thread 类
- 实现 Runnable 接口
- 实现 Callable 接口，需要实现的是 call() 方法

28、如何停止一个正在运行的线程？

- 使用共享变量的方式

在这种方式中，之所以引入共享变量，是因为该变量可以被多个执行相同任务的线程用来作为是否中断的信号，通知中断线程的执行。

- 使用interrupt方法终止线程

如果一个线程由于等待某些事件的发生而被阻塞，又该怎样停止该线程呢？这种情况经常会发生，比如当一个线程由于需要等候键盘输入而被阻塞，或者调用Thread.join()方法，或者Thread.sleep()方法，在网络中调用ServerSocket.accept()方法，或者调用了DatagramSocket.receive()方法时，都有可能导致线程阻塞，使线程处于不可运行状态时，即使主程序中将该线程的共享变量设置为true，但该线程此时根本无法检查循环标志，当然也就无法立即中断。这里我们给出的建议是，不要使用stop()方法，而是使用Thread提供的interrupt()方法，因为该方法虽然不会中断一个正在运行的线程，但是它可以使一个被阻塞的线程抛出一个中断异常，从而使线程提前结束阻塞状态，退出堵塞代码。

29、notify()和notifyAll()有什么区别？

当一个线程进入wait之后，就必须等其他线程notify/notifyall,使用notifyall,可以唤醒所有处于wait状态的线程，使其重新进入锁的争夺队列中，而notify只能唤醒一个。

如果没把握，建议notifyAll，防止notigy因为信号丢失而造成程序异常。

30、什么是Daemon线程？它有什么意义？

所谓后台(daemon)线程，是指在程序运行的时候在后台提供一种通用服务的线程，并且这个线程并不属于程序中不可或缺的部分。因此，当所有的非后台线程结束时，程序也就终止了，同时会杀死进程中的所有后台线程。

反过来说，只要有任何非后台线程还在运行，程序就不会终止。必须在线程启动之前调用setDaemon()方法，才能把它设置为后

台线程。注意：后台进程在不执行finally子句的情况下就会终止其run()方法。

比如：JVM的垃圾回收线程就是Daemon线程，Finalizer也是守护线程。

31、java如何实现多线程之间的通讯和协作？

中断和共享变量

32、什么是可重入锁（ReentrantLock）？

举例来说明锁的可重入性

```
public class UnReentrant{
    Lock lock = new Lock();
    public void outer(){
        lock.lock();
        inner();
        lock.unlock();
    }
    public void inner(){
        lock.lock();
        //do something
        lock.unlock();
    }
}
```

outer中调用了inner，outer先锁住了lock，这样inner就不能再获取lock。其实调用outer的线程已经获取了lock锁，但是不能在inner中重复利用已经获取的锁资源，这种锁即称之为 不可重入可重入就意味着：线程可以进入任何一个它已经拥有的锁所同步着的代码块。

synchronized、ReentrantLock都是可重入的锁，可重入锁相对来说简化了并发编程的开发。

33、当一个线程进入某个对象的一个synchronized的实例方法后，其它线程是否可进入此对象的其它方法？

如果其他方法没有synchronized的话，其他线程是可以进入的。

所以要开放一个线程安全的对象时，得保证每个方法都是线程安全的。

34、乐观锁和悲观锁的理解及如何实现，有哪些实现方式？

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如Java里面的同步原语synchronized关键字的实现也是悲观锁。

乐观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

乐观锁的实现方式：

- 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。
- java中的Compare and Swap即CAS，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。 CAS操作中包含三个操作数——需要读写的内存位置(V)、进行比较的预期原值(A)和拟写入的新值(B)。如果内存位置V的值与预期原值A相匹配，那么处理器会自动将该位置值更新为新值B。否则处理器不做任何操作。

CAS缺点：

- **ABA问题：**

比如说一个线程one从内存位置V中取出A，这时候另一个线程two也从内存中取出A，并且two进行了一些操作变成了B，然后two又将V位置的数据变成A，这时候线程one进行CAS操作发现内存中仍然是A，然后one操作成功。尽管线程one的CAS操作成功，但可能存在潜藏的问题。从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。

- **循环时间长开销大：**

对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

- **只能保证一个共享变量的原子操作：**

当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁。

35、SynchronizedMap和ConcurrentHashMap有什么区别？

SynchronizedMap一次锁住整张表来保证线程安全，所以每次只能有一个线程来访为map。

ConcurrentHashMap使用分段锁来保证在多线程下的性能。ConcurrentHashMap中则是一次锁住一个桶。

ConcurrentHashMap默认将hash表分为16个桶，诸如get,put,remove等常用操作只锁当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有16个写线程执行，并发性能的提升是显而易见的。

另外ConcurrentHashMap使用了一种不同的迭代方式。在这种迭代方式中，当iterator被创建后集合再发生改变就不再是抛出ConcurrentModificationException，取而代之的是在改变时new新的数据从而不影响原有的数据，iterator完成后再将头指针替换为新的数据，这样iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变。

36、CopyOnWriteArrayList可以用于什么应用场景？

CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出ConcurrentModificationException。在CopyOnWriteArrayList中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行。

- I. 由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致young gc或者full gc；
- II. 不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个set操作后，读取到数据可能还是旧的，虽然CopyOnWriteArrayList 能做到最终一致性，但是还是没法满足实时性要求；

- 读写分离，读和写分开
- 最终一致性
- 使用另外开辟空间的思路，来解决并发冲突

37、什么叫线程安全？servlet是线程安全吗？

线程安全是编程中的术语，指某个函数、函数库在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。

Servlet不是线程安全的，servlet是单实例多线程的，当多个线程同时访问同一个方法，是不能保证共享变量的线程安全性的。

Struts2的action是多实例多线程的，是线程安全的，每个请求过来都会new一个新的action分配给这个请求，请求完成后销毁。

SpringMVC的Controller是线程安全的吗？不是的，和Servlet类似的处理流程

Struts2好处是不用考虑线程安全问题；Servlet和SpringMVC需要考虑线程安全问题，但是性能可以提升不用处理太多的gc，可以使用ThreadLocal来处理多线程的问题。

38、volatile有什么用？能否用一句话说明下volatile的应用场景？

volatile保证内存可见性和禁止指令重排。

volatile用于多线程环境下的单次操作(单次读或者单次写)。

39、为什么代码会重排序？

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

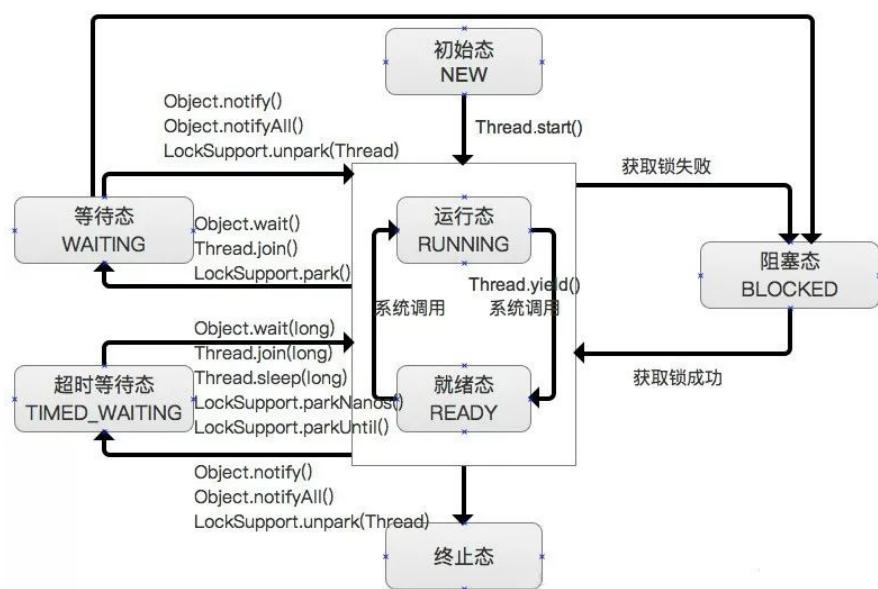
- 在单线程环境下不能改变程序运行的结果；
- 存在数据依赖关系的不允许重排序

需要注意的是：重排序不会影响单线程环境的执行结果，但是会破坏多线程的执行语义。

40、在java中wait和sleep方法的不同？

最大的不同是在等待时wait会释放锁，而sleep一直持有锁。Wait通常被用于线程间交互，sleep通常被用于暂停执行。

直接了解的深入一点吧：



在Java中线程的状态一共被分成6种：

初始态：NEW

创建一个Thread对象，但还未调用start()启动线程时，线程处于初始态。

运行态：RUNNABLE

在Java中，运行态包括就绪态和运行态。

就绪态该状态下的线程已经获得执行所需的所有资源，只要CPU分配执行权就能运行。所有就绪态的线程存放在就绪队列中。

运行态获得CPU执行权，正在执行的线程。由于一个CPU同一时刻只能执行一条线程，因此每个CPU每个时刻只有一条运行态的线程。

阻塞态

当一条正在执行的线程请求某一资源失败时，就会进入阻塞态。而在Java中，阻塞态专指请求锁失败时进入的状态。由一个阻塞队列存放所有阻塞态的线程。处于阻塞态的线程会不断请求资源，一旦请求成功，就会进入就绪队列，等待执行。PS：锁、IO、Socket等都资源。

等待态

当前线程中调用wait、join、park函数时，当前线程就会进入等待态。也有一个等待队列存放所有等待态的线程。线程处于等待态表示它需要等待其他线程的指示才能继续运行。进入等待态的线程会释放CPU执行权，并释放资源（如：锁）

超时等待态

当运行中的线程调用sleep(time)、wait、join、parkNanos、parkUntil时，就会进入该状态；它和等待态一样，并不是因为请求不到资源，而是主动进入，并且进入后需要其他线程唤醒；进入该状态后释放CPU执行权和占有的资源。与等待态的区别：到了超时时间后自动进入阻塞队列，开始竞争锁。

终止态

线程执行结束后的状态。

注意：

- `wait()`方法会释放CPU执行权和占有的锁。
- `sleep(long)`方法仅释放CPU使用权，锁仍然占用；线程被放入超时等待队列，与`yield`相比，它会使线程较长时间得不到运行。
- `yield()`方法仅释放CPU执行权，锁仍然占用，线程会被放入就绪队列，会在短时间内再次执行。
- `wait`和`notify`必须配套使用，即必须使用同一把锁调用；
- `wait`和`notify`必须放在一个同步块中调用`wait`和`notify`的对象必须是他们所处同步块的锁对象。

41、一个线程运行时发生异常会怎样？

如果异常没有被捕获该线程将会停止执行。`Thread.UncaughtExceptionHandler`是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候JVM会使用`Thread.getUncaughtExceptionHandler()`来查询线程的`UncaughtExceptionHandler`并将线程和异常作为参数传递给handler的`uncaughtException()`方法进行处理。

42、如何在两个线程间共享数据？

在两个线程间共享变量即可实现共享。

一般来说，共享变量要求变量本身是线程安全的，然后在线程内使用的时候，如果有对共享变量的复合操作，那么也得保证复合操作的线程安全性。

43、Java中`notify` 和 `notifyAll`有什么区别？

`notify()`方法不能唤醒某个具体的线程，所以只有一个线程在等待的时候它才有用武之地。而`notifyAll()`唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行。

44、为什么`wait`, `notify` 和 `notifyAll`这些方法不在`thread`类里面？

一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。由于`wait`, `notify`和`notifyAll`都是锁级别的操作，所以把他们定义在`Object`类中因为锁属于对象。

45、什么是`ThreadLocal`变量？

`ThreadLocal`是Java里一种特殊的变量。每个线程都有一个`ThreadLocal`就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。它是为创建代价高昂的对象获取线程安全的好方法，比如你可以用`ThreadLocal`让`SimpleDateFormat`变成线程安全的，因为那个类创建代价高昂且每次调用都需要创建不同的实例所以不值得在局部范围使用它，如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。首先，通过复用减少了代价高昂的对象的创建个数。其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全。

46、Java中`interrupted` 和 `isInterrupted`方法的区别？

`interrupt`

`interrupt`方法用于中断线程。调用该方法的线程的状态为将被置为“中断”状态。

注意：线程中断仅仅是置线程的中断状态位，不会停止线程。需要用户自己去监视线程的状态为并做处理。支持线程中断的方法（也就是线程中断后会抛出interruptedException的方法）就是在监视线程的中断状态，一旦线程的中断状态被置为“中断状态”，就会抛出中断异常。

interrupted

查询当前线程的中断状态，并且清除原状态。如果一个线程被中断了，第一次调用interrupted则返回true，第二次和后面的就返回false了。

isInterrupted

仅仅是查询当前线程的中断状态

47、为什么wait和notify方法要在同步块中调用？

Java API强制要求这样做，如果你不这么做，你的代码会抛出IllegalMonitorStateException异常。还有一个原因是为了避免wait和notify之间产生竞态条件。

48、为什么你应该在循环中检查等待条件？

处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件下退出。

49、Java中的同步集合与并发集合有什么区别？

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。在Java1.5之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。Java5介绍了并发集合像ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

50、什么是线程池？为什么要使用它？

创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。从JDK1.5开始，Java API提供了Executor框架让你可以创建不同的线程池。

51、怎么检测一个线程是否拥有锁？

在java.lang.Thread中有一个方法叫holdsLock()，它返回true如果当且仅当当前线程拥有某个具体对象的锁。

52、你如何在Java中获取线程堆栈？

- kill -3 [java pid]

不会在当前终端输出，它会输出到代码执行的或指定的地方去。比如，kill -3 tomcat pid，输出堆栈到log目录下。

- Jstack [java pid]

这个比较简单，在当前终端显示，也可以重定向到指定文件中。

不做说明，打开JvisualVM后，都是界面操作，过程还是很简单的。

53、JVM中哪个参数是用来控制线程的栈堆栈小的？

-Xss 每个线程的栈大小

54、Thread类中的yield方法有什么作用？

使当前线程从执行状态（运行状态）变为可执行态（就绪状态）。

当前线程到了就绪状态，那么接下来哪个线程会从就绪状态变成执行状态呢？可能是当前线程，也可能是其他线程，看系统的分配了。

55、Java中ConcurrentHashMap的并发度是什么？

ConcurrentHashMap把实际map划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是ConcurrentHashMap类构造函数的一个可选参数，默认值为16，这样在多线程情况下就能避免争用。

在JDK8后，它摒弃了Segment（锁段）的概念，而是启用了一种全新的方式实现，利用CAS算法。同时加入了更多的辅助变量来提高并发度，具体内容还是查看源码吧。

56、Java中Semaphore是什么？

Java中的Semaphore是一种新的同步类，它是一个计数信号。从概念上讲，从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore只对可用许可的号码进行计数，并采取相应的行动。信号量常常用在多线程的代码中，比如数据库连接池。

57、Java线程池中submit() 和 execute()方法有什么区别？

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中。

而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

58、什么是阻塞式方法？

阻塞式方法是指程序会一直等待该方法完成期间不做其他事情，ServerSocket的accept()方法就是一直等待客户端连接。这里的阻塞是指调用结果返回之前，当前线程会被挂起，直到得到结果之后才会返回。此外，还有异步和非阻塞式方法在任务完成前就返回。

59、Java中的ReadWriteLock是什么？

读写锁是用来提升并发程序性能的锁分离技术的成果。

60、volatile 变量和 atomic 变量有什么不同？

Volatile变量可以确保先行关系，即写操作会发生在后续的读操作之前，但它并不能保证原子性。例如用volatile修饰count变量那么 count++ 操作就不是原子性的。

而AtomicInteger类提供的atomic方法可以让这种操作具有原子性如getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

61、可以直接调用Thread类的run ()方法么？

当然可以。但是如果我们调用了Thread的run()方法，它的行为就会和普通的方法一样，会在当前线程中执行。为了在新的线程中执行我们的代码，必须使用Thread.start()方法。

62、如何让正在运行的线程暂停一段时间？

我们可以使用Thread类的Sleep()方法让线程暂停一段时间。需要注意的是，这并不会让线程终止，一旦从休眠中唤醒线程，线程的状态将会被改变为Runnable，并且根据线程调度，它将得到执行。

63、你对线程优先级的理解是什么？

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个int变量(从1-10)，1代表最低优先级，10代表最高优先级。

java的线程优先级调度会委托给操作系统去处理，所以与具体的操作系统优先级有关，如非特别需要，一般无需设置线程优先级。

64、什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)？

线程调度器是一个操作系统服务，它负责为Runnable状态的线程分配CPU时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。

同上一个问题，线程调度并不受到Java虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

时间分片是指将可用的CPU时间分配给可用的Runnable线程的过程。分配CPU时间可以基于线程优先级或者线程等待的时间。

65、你如何确保main()方法所在的线程是Java 程序最后结束的线程？

我们可以使用Thread类的join()方法来确保所有程序创建的线程在main()方法退出前结束。

66、线程之间是如何通信的？

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object类中wait() otify() otifyAll()方法可以用于线程间通信关于资源的锁的状态。

67、为什么线程通信的方法wait(), notify()和notifyAll()被定义在Object类里?

Java的每个对象中都有一个锁(monitor, 也可以成为监视器) 并且wait(), notify()等方法用于等待对象的锁或者通知其他线程对象的监视器可用。在Java的线程中并没有可供任何对象使用的锁和同步器。这就是为什么这些方法是Object类的一部分，这样Java的每一个类都有用于线程间通信的基本方法。

68、为什么wait(), notify()和notifyAll ()必须在同步方法或者同步块中被调用?

当一个线程需要调用对象的wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的notify()方法。同样的，当一个线程需要调用对象的notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

69、为什么Thread类的sleep()和yield ()方法是静态的?

Thread类的sleep()和yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

70、如何确保线程安全?

在Java中可以有很多方法来保证线程安全——同步，使用原子类(atomic concurrent classes)，实现并发锁，使用volatile关键字，使用不变类和线程安全类。

71、同步方法和同步块，哪个是更好的选择?

同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

同步块更要符合开放调用的原则，只在需要锁住的代码块锁住相应的对象，这样从侧面来说也可以避免死锁。

72、如何创建守护线程?

使用Thread类的setDaemon(true)方法可以将线程设置为守护线程，需要注意的是，需要在调用start()方法前调用这个方法，否则会抛出IllegalThreadStateException异常。

73、什么是Java Timer 类? 如何创建一个有特定时间间隔的任务?

java.util.Timer是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer类可以用安排一次性任务或者周期任务。

java.util.TimerTask是一个实现了Runnable接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用Timer去安排它的执行。

目前有开源的Qurtz可以用来创建定时任务。

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. 为什么你学不会递归？
2. MySQL:Left Join 避坑指南
3. AJAX 请求真的不安全么？
4. 一个女生不主动联系你还有机会吗？
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！

Java 开发中如何正确的踩坑

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: mafly | 责编: 乐乐

链接: cnblogs.com/mafly/p/trap.html

为什么说一个好的员工能顶 100 个普通员工?

我们的做法是, 要用最好的人。我一直都認為研发本身是很有创造性的, 如果人不放松, 或不够聪明, 都很难做得好。你要找到最好的人, 一个好的工程师不是顶10个, 是顶100个。

所以, 在核心工程师上面, 大家一定要不惜血本去找, 千万不要想偷懒只用培养大学生的方法去做。最好的人本身有很强的驱动力, 你只要把他放到他喜欢的事情上, 让他自己有玩的心态, 他才能真正做出一些事情, 打动他自己, 才能打动别人。所以你今天看到我们很多的工程师, 他自己在边玩边创新。

所以, 找最好的人, 要给他做他喜欢和擅长的事情。研发人员千万不要去管太严, 一管就“死”了。工程师很讨厌跟规章制度打交道, 作汇报他都很烦, 大家不要管他, 让用户去管他。他做好了一个产品, 用户表扬他, 这个大神多牛逼。他做不好了, 用户骂他, 他自己赶紧去改。

再谈阿里巴巴 Java 开发手册

之前在这个手册刚发布的时候看过一遍, 当时感觉真是每个开发者都应该必读的一本手册, 期间还写过一篇关于日志规约的文章:

<http://www.cnblogs.com/mafly/p/slf4j.html>

最近由于在总结一些我们日常开发中容易忽略的问题, 可能是最低级的编码常见问题, 往往这也是最最容易忽略的, 所以, 又重新看了一遍这个手册, 好像最近它也更新到了 1.2 版本。

这个手册目的就是让我们尽可能少踩坑, 杜绝踩重复的坑。我接下来就打算试着写一些“坑”出来, 来看看我们如何一不留神踩坑的, 以及如何用正确的姿势跳出坑。

随随便便写出 NPE

首先声明一个 User 对象, 接下来所有代码可能都会用到这个对象做演示, 在下面将不在赘述。很简单, 不上代码, 上图片:

```

2 /**
3  * Created by mafly
4  * Date: 2017/5/17
5 */
6
7 public class User {
8
9     private Integer id;
10
11    private String name;
12
13    public Integer getId() { return id; }
14
15    public void setId(Integer id) { this.id = id; }
16
17    public String getName() { return name; }
18
19    public void setName(String name) { this.name = name; }
20
21    @Override
22    public String toString() { ... }
23
24
25
26
27

```

1. 自动解箱抛 NPE

代码只有一行，再简单不过了：int method() { return new User().getId(); }

```

12
13     * Created by mafly
14     * Date: 2017/5/19
15     */
16 public class NPE {
17
18     public static void main(String[] args) {
19         method();
20     }
21
22     static int method() {
23         return new User().getId();
24     }
25
26
27

```

Run NPE

```

D:\Program Files\Java\jdk1.7.0_80\bin\java" ...
Exception in thread "main" java.lang.NullPointerException
at Exception.NPE.method(NPE.java:27)
at Exception.NPE.main(NPE.java:18)

Process finished with exit code 1

```

踩坑姿势：包装类型为 null 时，进行自动转换为基本数据类型报错。

解决方案：返回之前进行判断与处理或者改为相同类型。

2. 级联调用易产生 NPE

这段代码有点容易迷惑人，因为它进行了集合元素的 isEmpty 判断，按说不会出问题了吧。看代码：

```

1 static void method1() {
2     List<User> list = new ArrayList<User>();
3     list.add(new User());
4
5     if (!CollectionUtils.isEmpty(list)) {
6         for (User user : list) {
7             System.out.println("userid:" + user.getId().toString());
8         }
9     }
10 }

```

不废话，看运行结果：

```
18 public static void main(String[] args) {
19     method1();
20 }
21
22 /**
23  * ...
24 */
25 static void method1() {
26     List<User> list = new ArrayList<User>();
27     list.add(new User());
28
29     if (!CollectionUtils.isEmpty(list)) {
30         for (User user : list) {
31             System.out.println("userid:" + user.getId().toString());
32         }
33     }
34 }
35
36
37
```

Run NPE

D:\Program Files\Java\jdk1.7.0_80\bin\java ...
Exception in thread "main" java.lang.NullPointerException
at Exception.NPE.method1(NPE.java:32)
at Exception.NPE.main(NPE.java:19)

没错，还是报错了。

踩坑姿势：其实就是在之前做了对象不为空的判断，但你并不能保证对象中的值不为空，而且这时候去级联调用就会抛 NPE。

手册中关于 NPE 的描述：

防止 NPE 是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败、序列化失败、运行时异常等场景返回 null 的情况。
集合里的元素即使 isEmpty，取出的数据元素也可能为 null。

级联调用 obj.getA().getB().getC(); 一连串调用，易产生 NPE

3. 关于 Equals

这是日常开发中用于相等比较使用最多的方法了吧，因为当年谁没被 == 坑过阿。现在一般我们都会这么写：

user.getName().equals("mafly");

```
9  * Created by mafly
10 * Date: 2017/5/17
11 */
12 public class Equals {
13     public static void main(String[] args) {
14
15         User user = new User();
16
17         user.getName().equals("mafly");
18     }
19 }
20
21
```

Run Equals

D:\Program Files\Java\jdk1.7.0_80\bin\java ...
Exception in thread "main" java.lang.NullPointerException
at Exception.Equals.main(Equals.java:17)

Process finished with exit code 1

踩坑姿势：一不小心使用了 null 值调用了 Equals 方法。

解决方案：很简单咯，这么写："mafly".equals(user.getName());

equals 方法容易抛空指针异常，应使用常量或确定有值的对象来调用 equals。

4. Map 下的 NPE

Map 应该是我们开发中使用最频繁的了，最常用的可能有 `HashMap`、`ConcurrentHashMap` 这俩了，可能会一不留神写出这样的代码：

```
18 public static void main(String[] args) {
19     User user = new User();
20
21     Map<Integer, Object> hashMap = new HashMap<>();
22     hashMap.put(user.getId(), user.getName());
23
24     Map<Integer, Object> concurrentHashMap = new ConcurrentHashMap<>();
25     concurrentHashMap.put(user.getId(), user.getName());
26
27 }
28
```

Run MapNPE
D:\Program Files\Java\jdk1.7.0_80\bin\java ...
+Exception in thread "main" java.lang.NullPointerException <1 internal calls>
at Exception.MapNPE.main(MapNPE.java:25)
Process finished with exit code 1

踩坑姿势：可能我们知道 `ConcurrentHashMap` 的 K/V 都不能为空，但我们有时候并不知道传进来的值是否为空。

解决方案：设置时做下检验，对它的特性正确理解及使用。

由于 `HashMap` 的干扰，很多人认为 `ConcurrentHashMap` 是可以置入 `null` 值，而事实上，存储 `null` 值时会抛出 NPE 异常。

Map 类集合 K/V 能不能存储 `null` 值的情况，如下表格：

集合类	Key	Value	Super	说明
Hashtable	不允许为 null	不允许为 null	Dictionary	线程安全
ConcurrentHashMap	不允许为 null	不允许为 null	AbstractMap	分段锁技术
TreeMap	不允许为 null	允许为 null	AbstractMap	线程不安全
HashMap	允许为 null	允许为 null	AbstractMap	线程不安全

简单聊聊常用的集合

5.foreach 遍历集合删除元素

大家应该都知道，在遍历集合时对元素进行 `add/remove` 操作要使用 `Iterator`，使用 `for` 循环时会报错，一定会报错吗？看代码：

```
1 public static void main(String[] args) {
2     List<String> a = new ArrayList<>();
3     a.add("1");
4     a.add("2");
5     a.add("3");
6
7     for (String temp : a) {
8         if ("2".equals(temp)) {
9             a.remove(temp);
10        }
11    }
12
13     Iterator<String> it = a.iterator();
14     while (it.hasNext()) {
15         String temp = it.next();
```

```
15     String temp = it.next();
16     if ("2".equals(temp)) {
17         it.remove();
18     }
19 }
20 }
```

应该会报错的吧？因为在 for 循环中移出了元素，如果你运行了就会惊讶的，输出如下：



```
17 public static void main(String[] args) {
18     List<String> a = new ArrayList<>();
19     a.add("1");
20     a.add("2");
21     a.add("3");
22
23     for (String temp : a) {
24         if ("2".equals(temp)) {
25             a.remove(temp);
26         }
27     }
28 }
```

Run FforeachList
D:\Program Files\Java\jdk1.7.0_80\bin\java" ...
Process finished with exit code 0

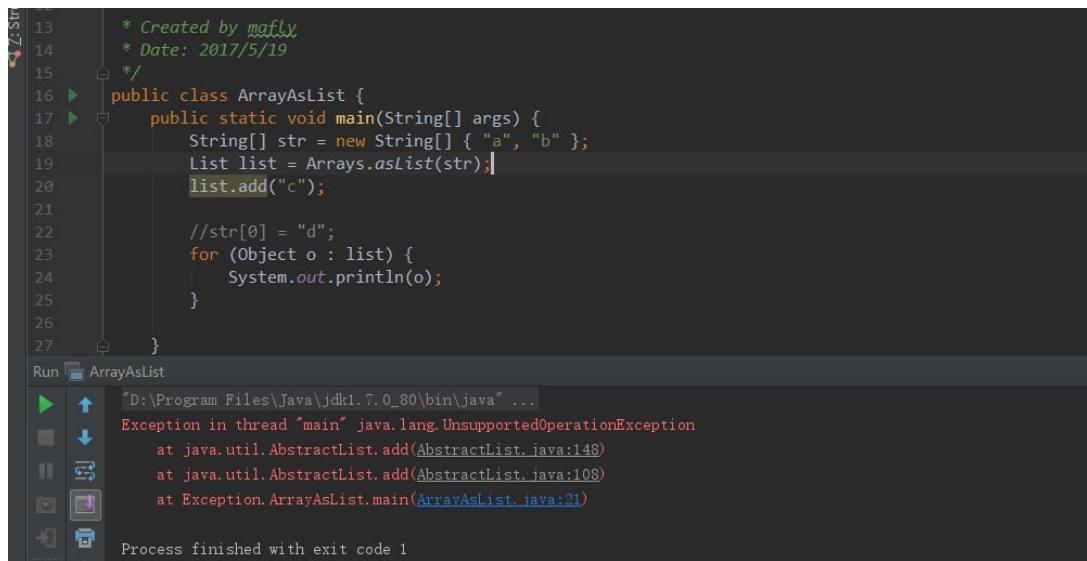
不解释其中原因了，感兴趣的可以看这篇文章：

<http://blog.csdn.net/bimuyulaila/article/details/52088124>

不管是不是倒数第二个元素才没问题，我们依然要注意不要在 foreach 循环里进行元素的 remove/add 操作。remove 元素请使用 Iterator 方式（代码第二种），如果并发操作，需要对 Iterator 对象加锁。

6.Arrays.asList() 数组转换集合

这个工具类应该都用过，可以很方便的把数组转换为集合，直接看结果吧：



```
13 * Created by mafly
14 * Date: 2017/5/19
15 */
16 public class ArrayAsList {
17     public static void main(String[] args) {
18         String[] str = new String[] { "a", "b" };
19         List list = Arrays.asList(str);
20         list.add("c");
21
22         //str[0] = "d";
23         for (Object o : list) {
24             System.out.println(o);
25         }
26     }
27 }
```

Run FarrayAsList
D:\Program Files\Java\jdk1.7.0_80\bin\java" ...
Exception in thread "main" java.lang.UnsupportedOperationException
at java.util.AbstractList.add(AbstractList.java:148)
at java.util.AbstractList.add(AbstractList.java:108)
at Exception.ArrayAsList.main(ArrayAsList.java:21)
Process finished with exit code 1

踩坑姿势：Arrays.asList() 把数组转换成集合时，不能使用其修改集合相关的方法，它的 add/remove/clear 方法会抛出 UnsupportedOperationException 异常。asList() 的返回对象是一个 Arrays 内部类，并没有实现集合的修改方法。

解决方案：在转换之前操作咯。还需要注意一点，在你转换后，再对数组的值进行修改时，集合也会跟着变哦（注释掉的代码）。

7. toArray() 集合转换数组

当我们需要把一个集合转换为数组时，往往会调用 `toArray()` 方法，如果你用的是无参的这个可以吗？

```
12
13     * Created by mafly
14     * Date: 2017/5/19
15     */
16 public class ListToArray {
17     public static void main(String[] args) {
18         List<String> list = new ArrayList<>();
19         list.add("hello");
20         list.add("mafly");
21
22         String[] array = (String[]) list.toArray();
23
24
25 }
```

Run ListToArray
D:\Program Files\Java\jdk1.7.0_80\bin\java ...
Exception in thread "main" java.lang.ClassCastException: [Ljava.lang.Object; cannot be cast to [Ljava.lang.String;
at Exception.ListToArray.main(ListToArray.java:22)
Process finished with exit code 1

当然不可以啦！会报 `ClassCastException` 异常。

踩坑姿势：直接使用 `toArray()` 无参方法返回值只能是 `Object[]` 类，若强转其它类型数组将会抛异常。

解决方案：使用 `T[] toArray(T[] a);` 有参数这个方法

代码如下：

```
1 String[] array = new String[list.size()];
2 array = list.toArray(array);
```

8. subList 的使用

集合中的 `subList` 是用于来返回某一部分的视图内容的，可能我们不是很常用，但是其中有好多坑的，直接看代码：

```
12
13     public static void main(String[] args) {
14         List<String> list = new java.util.ArrayList<>();
15         list.add("hello");
16
17         //java.lang.IndexOutOfBoundsException
18         //list.subList(0,4);
19
20         List<String> newList = list.subList(0, 1);
21         newList.add("mafly");
22
23         System.out.println("list size:" + list.size());
24         for (String str : list) {
25             System.out.println(str);
26         }
27
28         newList.remove(0: "mafly");
29         for (String str : newList) {
30             System.out.println(str);
31         }
32
33         System.out.println("-----");
34         list.add("mafly");
35         for (String str : list) {
36             System.out.println(str);
37         }
38
39         //java.util.ConcurrentModificationException
40         for (String str : newList) {
41             System.out.println(str);
42         }
43
44     }
```

The screenshot shows a Java IDE interface with a terminal window. The terminal output is as follows:

```
"D:\Program Files\Java\jdk1.7.0_80\bin\java" ...
list size:2
hello
mafly
hello
-----
hello
mafly
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$SubList.checkForComodification(ArrayList.java:1169)
    at java.util.ArrayList$SubList.listIterator(ArrayList.java:1049)
    at java.util.AbstractList.listIterator(AbstractList.java:299)
    at java.util.ArrayList$SubList.iterator(ArrayList.java:1045)
    at Exception.SubList.main(SubList.java:40)

Process finished with exit code 1
```

这次我们从输出来看上面的所有关于 subList 的代码。

- 18行：当你原始集合大小没有那么大时，毫无疑问抛异常。
- 20-21行：得到一个新的集合，我们往新集合中增加一条数据。
- 23-26行：遍历原始集合，竟然 size=2 了，而且往新集合中增加的数据存在与原始集合。
- 28-31行：移除新集合中一条数据，遍历新集合。
- 33-37行：原始集合增加一条数据并遍历。
- 40-42行：遍历新集合，抛出 ConcurrentModificationException 异常。

从上述代码中，我们应该可以得出如下结论：返回的新集合是靠原来的集合支持的，修改都会影响到彼此对方。在 subList 场景中，高度注意对原集合元素个数的修改，会导致子列表的遍历、增加、删除均产生异常。

先总结一下

写到这只是其中关于异常部分的一些坑吧，还有另外一些令人异常惊讶的“我的天呐”的问题，由于篇幅太长了点，感觉不能再写下去了，过两天再接着写吧。

异常真的是一个有意思的问题。

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！

Java 开发中常用的 4 种加密方法

爱编程的浪子 Java后端 2019-10-14

点击上方 Java后端, 选择设为星标

技术博文, 及时送达

来源 | my.oschina.net/u/4139951/blog/3077236

作者 | 爱编程的浪子

一、工具类

1. md5加密工具类
2. base64加密工具类
3. Bcrypt工具类

二、加密测试

1. MD5加密测试
2. base64加密测试
3. SHA加密测试
4. BCrypt加密测试

一、工具类

1. md5加密工具类

```
1 public class MD5Utils {  
2  
3     private static final String hexDigits[] = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "a", "b", "c",  
4 ;  
5  
6     /*  
7     *  
8     * MD5加密  
9     * @param origin 字符串  
10    * @param charsetname 编码  
11    * @return  
12    */  
13  
14     public static String MD5Encode(String origin, String charsetname)  
15     {  
16         String resultString = null  
17 ;  
18         try  
19     {  
20             resultString = new String(origin);  
21             MessageDigest md = MessageDigest.getInstance("MD5")  
22 ;  
23             if(null == charsetname || "".equals(charsetname)){  
24                 resultString = byteArrayToHexString(md.digest(resultString.getBytes()));  
25             }else  
26             {  
27                 resultString = encodeStr(resultString, charsetname);  
28             }  
29         } catch (Exception e) {  
30             e.printStackTrace();  
31         }  
32     }  
33  
34     private static String byteArrayToHexString(byte b[]) {  
35         int n = b.length;  
36         String stmp = "  
37         String result = "";  
38         int i = 0;  
39         while(i < n){  
40             result += hexDigits[b[i] & 0xf];  
41             i++;  
42         }  
43         return result;  
44     }  
45  
46     private static String encodeStr(String str, String charsetname) {  
47         byte[] bytes = null;  
48         try {  
49             bytes = str.getBytes(charsetname);  
50         } catch (Exception e) {  
51             e.printStackTrace();  
52         }  
53         return byteArrayToHexString(bytes);  
54     }  
55 }
```

```
26         resultString = byteArrayToHexString(md.digest(resultString.getBytes(charsetname)));
27     }
28 }catch (Exception e){
29 }
30 return resultString;
31 }
32
33
34 public static String byteArrayToHexString(byte b[])
35 {
36     StringBuffer resultSb = new StringBuffer();
37     for(int i = 0; i < b.length; i++){
38         resultSb.append(byteToHexString(b[i]));
39     }
40     return resultSb.toString();
41 }
42
43 public static String byteToHexString(byte b)
44 {
45     int n = b
46 ;
47     if(n < 0)
48     {
49         n += 256
50 ;
51     }
52     int d1 = n / 16
53 ;
54     int d2 = n % 16
55 ;
56     return hexDigits[d1] + hexDigits[d2];
57 }
```

2. base64加密工具类

```
1 public class Base64Util {  
2  
3     // 字符串编  
4     码  
5     private static final String UTF_8 = "UTF-8"  
6;  
7  
8     /*  
9     *  
10    * 加密字符串  
11    * @param inputData  
12    * @return  
13    */  
14    public static String decodeData(String inputData) {  
15        try {  
16            {  
17                // 加密逻辑  
18            }  
19        } catch (Exception e) {  
20            // 处理异常  
21        }  
22    }  
23}
```

```

16     if (null == inputData)
17     {
18         return null
19     }
20     }
21     return new String(Base64.decodeBase64(inputData.getBytes(UTF_8)), UTF_8);
22 } catch (UnsupportedEncodingException e) {
23 }
24 return null
25 ;
26 }
27
28 /*
29 *
30 * 解密加密后的字符串
31 * @param inputData
32 * @return
33 */
34 public static String encodeData(String inputData) {
35     try
36     {
37         if (null == inputData)
38     {
39         return null
40     }
41     }
42     return new String(Base64.encodeBase64(inputData.getBytes(UTF_8)), UTF_8);
43 } catch (UnsupportedEncodingException e) {
44 }
45 return null
46 ;
47 }
48
49 public static void main(String[] args) {
50     System.out.println(Base64Util.encodeData("我是中文"))
51 ;
52     String enStr = Base64Util.encodeData("我是中文")
53 ;
54     System.out.println(Base64Util.decodeData(enStr));
55 }
56 }
```

3. Bcrypt工具类

```

1 public class BcryptCipher {
2     // generate salt seed
3     private static final int SALT_SEED = 12
4 ;
5     // the head fo salt
6     private static final String SALT_STARTSWITH = "$2a$12"
7 ;
8     public static final String SALT_KEY = "salt"
```

```

9 ;
10
11     public static final String CIPHER_KEY = "cipher"
12 ;
13
14 /**
15 * Bcrypt encryption algorithm method
16 * @param encryptSource
17 * need to encrypt the string
18 * @return Map , two values in Map , salt and cipher
19 */
20     public static Map<String, String> Bcrypt(final String encryptSource) {
21         String salt = BCrypt.gensalt(SALT_SEED);
22         Map<String, String> bcryptResult = Bcrypt(salt, encryptSource);
23         return bcryptResult;
24     }
25 /**
26 *
27 * @param salt encrypt salt, Must conform to the rules
28 * @param encryptSource
29 * @return
30 */
31     public static Map<String, String> Bcrypt(final String salt, final String encryptSource) {
32         if (StringUtils.isBlank(encryptSource)) {
33             throw new RuntimeException("Bcrypt encrypt input params can not be empty")
34         }
35     }
36
37     if (StringUtils.isBlank(salt) || salt.length() != 29)
38 {
39         throw new RuntimeException("Salt can't be empty and length must be to 29")
40 ;
41     }
42     if (!salt.startsWith(SALT_STARTSWITH)) {
43         throw new RuntimeException("Invalid salt version, salt version is $2a$12")
44 ;
45     }
46
47     String cipher = BCrypt.hashpw(encryptSource, salt);
48     Map<String, String> bcryptResult = new HashMap<String, String>()
49 ;
50     bcryptResult.put(SALT_KEY, salt);
51     bcryptResult.put(CIPHER_KEY, cipher);
52     return bcryptResult;
53 }
54
55 }
```

二、加密测试

1. MD5加密测试

```
1 /**
```

```
2 * MD5加密
3 */
4 public class MD5Test {
5     public static void main(String[] args)
6     {
7         String string = "我是一句话"
8 ;
9         String byteArrayToHexString = MD5Utils.byteArrayToHexString(string.getBytes());
10        System.out.println(byteArrayToHexString); //e68891e698afe4b880e58fa5e8af9d
11
12    }
13 }
```

2. base64加密测试

```
1 /**
2  * base64加密
3 */
4 public class Base64Tester {
5
6     public static void main(String[] args)
7     {
8         String string = "我是一个字符串"
9 ;
10        String encodeData = Base64Util.encodeData(string); //加
11        密
12        String decodeData = Base64Util.decodeData(encodeData); //解
13        密
14        System.out.println(encodeData); //5oiR5piv5LiA5Liq5a2X56ym5Liy
15        System.out.println(decodeData); //我是一个字符串
16
17    }
18 }
```

3. SHA加密测试

```
1 /**
2  * SHA加密
3 */
4 public class ShaTest {
5
6     public static void main(String[] args)
7     {
8         String string = "我是一句话"
9 ;
10
11        String sha256Crypt = Sha2Crypt.sha256Crypt(string.getBytes());
12        System.out.println(sha256Crypt); //5$AFoQTeyt$TiqmobvcQXjXaAQMYosAA04KI8LfigZMGHzq.Dlp4NC
13
14    }
15 }
```

4. BCrypt加密测试

```
1  /**
2  * BCrypt加密
3  */
4  public class BCryptTest {
5
6      public static void main(String[] args)
7  {
8
9      String string = "我是一句话"
10 ;
11     Map<String, String> bcrypt = BcryptCipher.Bcrypt(string)
12 ;
13     System.out.println(bcrypt.keySet()); // [cipher, salt]
14
15     System.out.println(bcrypt.get("cipher")); // $2a$12$y1b92Z84gqlrsFzIzt1CV.dK0xNbwp0v3UwXXA7611
16     System.out.println(bcrypt.get("salt")); // $2a$12$y1b92Z84gqlrsFzIzt1CV.
17
18     Map<String, String> bcrypt2 = BcryptCipher.Bcrypt(bcrypt.get("salt"), string)
19 ;
20     System.out.println(bcrypt2.get("SALT_KEY")); // null
21     System.out.println(bcrypt2.get("CIPHER_KEY")); // null
22 }
23 }
```

-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. 从零搭建创业公司后台技术栈
2. 如何阅读 Java 源码？
3. 某小公司RESTful、前后端分离的实践
4. 该如何弥补 GitHub 功能缺陷？
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！

Java 开发人员 2019 生态系统信息图

Java后端 1月19日

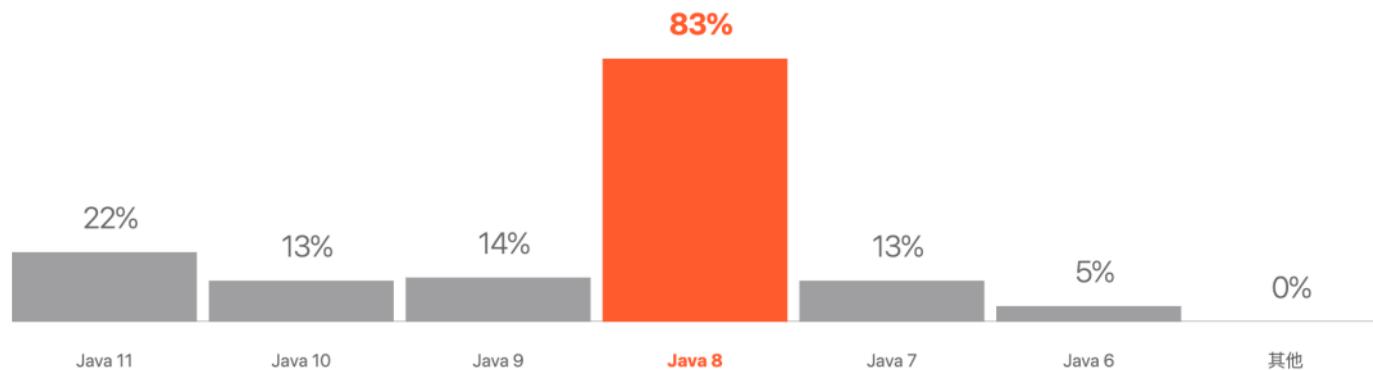
点击上方 Java后端, 选择 **设为星标**

优质文章, 及时送达

编辑 | 程序员DD

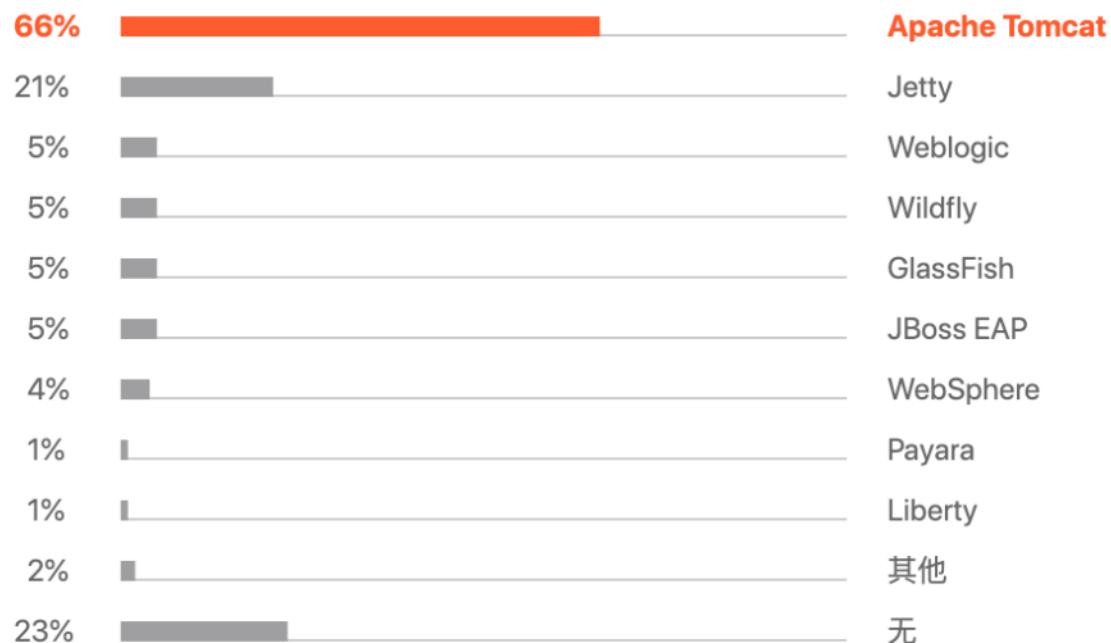
来源 | www.jetbrains.com/zh-cn/lp/devecosystem-2019/java/

您通常使用哪种（哪些）版本的Java？

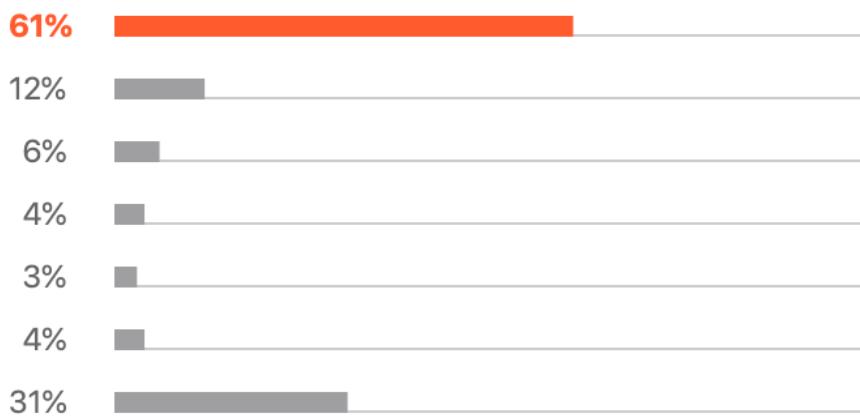


虽然 Java 10 和 11 越发流行，但 Java 8 仍是使用最多的版本。

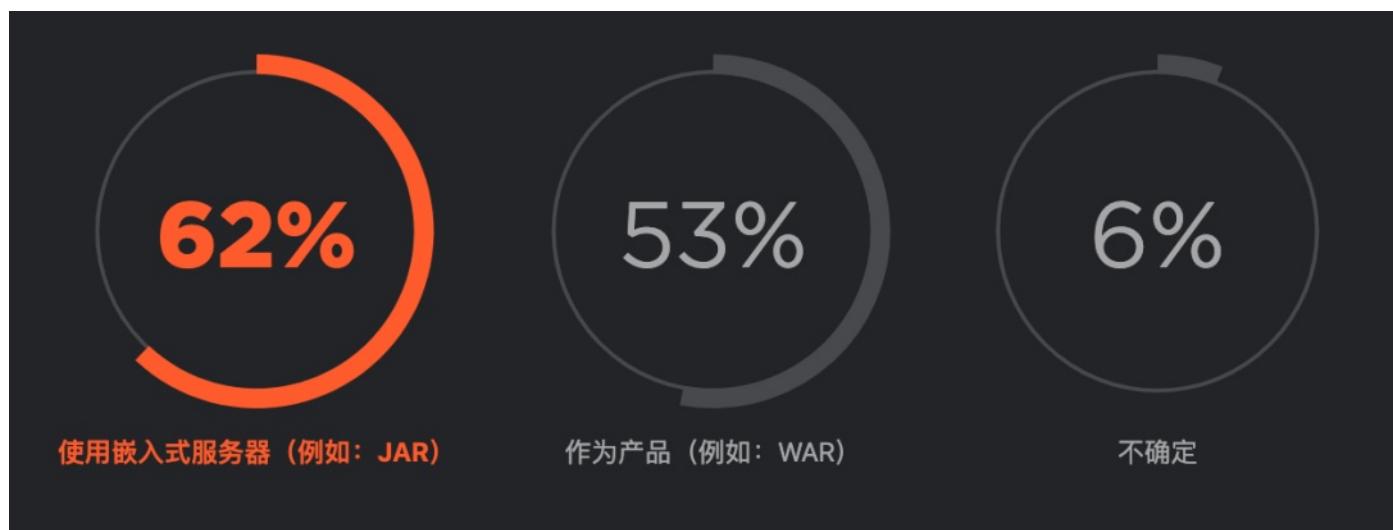
您通常使用哪种（哪些）应用程序服务器（如果使用）？



您使用哪种（哪些）框架代替应用服务器（如果使用）？



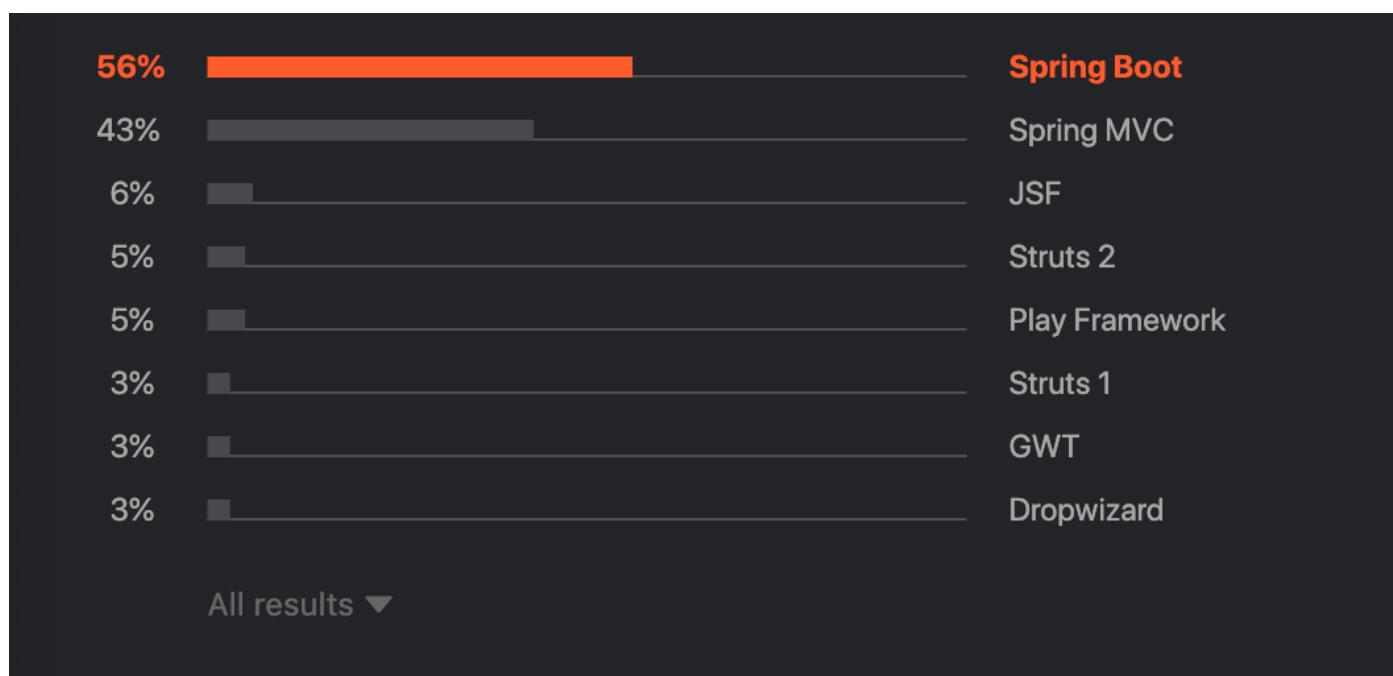
如何封装您的Web应用？



93% 的 Java 开发人员使用 JUnit 进行单元测试，而 51% 的人使用 Mockito。

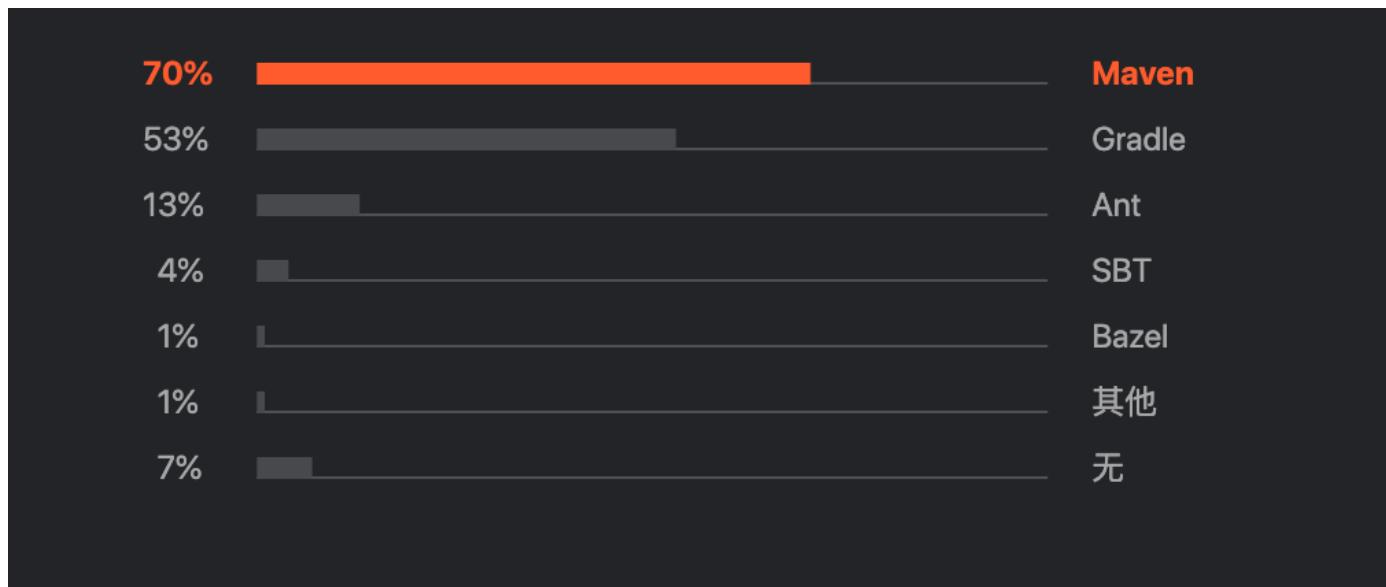
Tips：关注公众号：Java后端，每日推送技术博文。

您使用哪种（哪些）web框架（如果使用）？



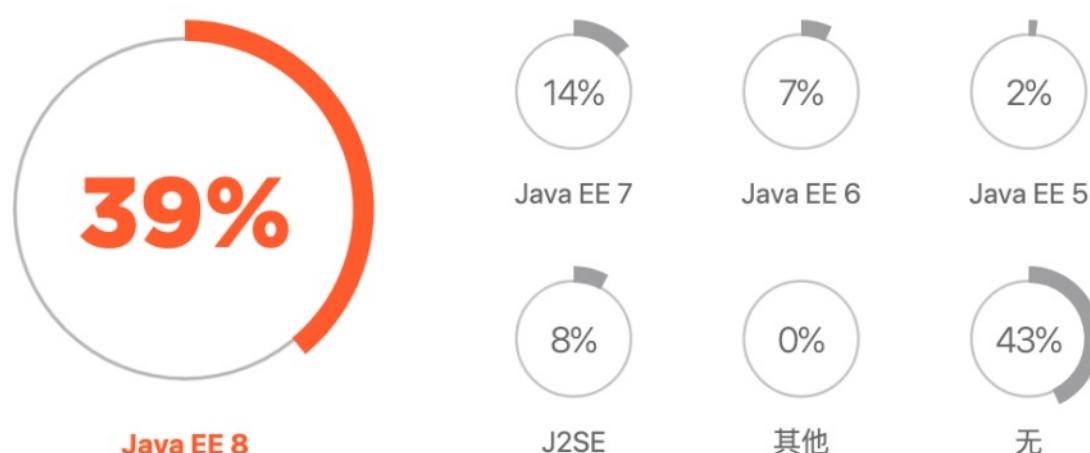
Spring Boot 已成为最流行的 Java web 框架，自去年以来增加 14%。

您通常使用哪种（哪些）构建系统（如果使用）？

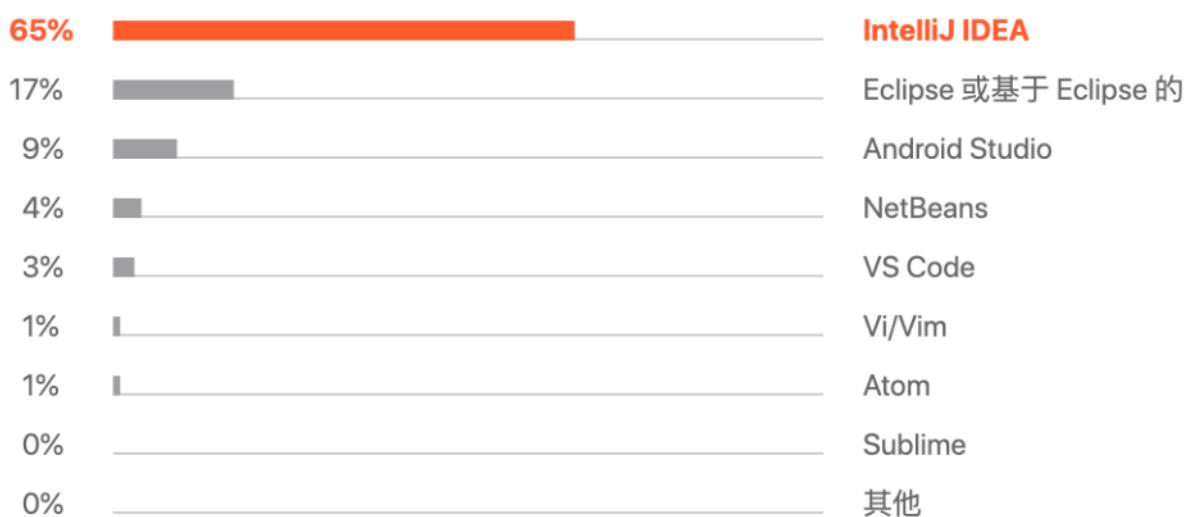


Maven 和 Gradle 继续成为使用最多的构建系统。

您通常使用哪种（哪些）版本的Java EE (EE4J)（如果使用）？



您最常使用哪种IDE/编辑器进行Java开发？



赠送一台苹果 iPad 过年如何呢？



微信扫描二维码，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！

Java性能优化的50个细节（珍藏版）

技术小能手 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 | 技术小能手

来源 | <https://yq.aliyun.com/articles/662001>

在Java程序中，性能问题的大部分原因并不在于Java语言，而是程序本身。养成良好的编码习惯非常重要，能够显著地提升程序性能。

1、尽量在合适的场合使用单例

使用单例可以减轻加载的负担，缩短加载的时间，提高加载的效率，但并不是所有地方都适用于单例，简单来说，单例主要适用于以下三个方面：

第一，控制资源的使用，通过线程同步来控制资源的并发访问；

第二，控制实例的产生，以达到节约资源的目的；

第三，控制数据共享，在不建立直接关联的条件下，让多个不相关的进程或线程之间实现通信。

2、尽量避免随意使用静态变量

当某个对象被定义为static变量所引用，那么GC通常是不会回收这个对象所占有的内存，如

```
public class A{  
    private static B b = new B();  
}
```

此时静态变量b的生命周期与A类同步，如果A类不会卸载，那么b对象会常驻内存，直到程序终止。

3、尽量避免过多过常地创建Java对象

尽量避免在经常调用的方法，循环中new对象，由于系统不仅要花费时间来创建对象，而且还要花时间对这些对象进行垃圾回收和处理，在我们可以控制的范围内，最大限度地重用对象，最好能用基本的数据类型或数组来替代对象。

4、尽量使用final修饰符

带有final修饰符的类是不可派生的。在JAVA核心API中，有许多应用final的例子，例如java、lang、String，为String类指定final防止了使用者覆盖length()方法。另外，如果一个类是final的，则该类所有方法都是final的。java编译器会寻找机会内联(inline)所有的final方法（这和具体的编译器实现有关），此举能够使性能平均提高50%。

如：让访问实例内变量的getter/setter方法变成”final”

简单的getter/setter方法应该被置成final，这会告诉编译器，这个方法不会被重载，所以，可以变成”inlined”，例子：

```
class MAF {
    public void setSize (int size) {
        _size = size;
    }
    private int _size;
}
更正
class DAF_fixed {
    final public void setSize (int size) {
        _size = size;
    }
    private int _size;
}
```

5、尽量使用局部变量

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈（Stack）中，速度较快；其他变量，如静态变量、实例变量等，都在堆（Heap）中创建，速度较慢。

6、尽量处理好包装类型和基本类型两者的使用场所

虽然包装类型和基本类型在使用过程中是可以相互转换，但它们两者所产生的内存区域是完全不同的，基本类型数据产生和处理都在栈中处理，包装类型是对象，是在堆中产生实例。在集合类对象，有对象方面需要的处理适用包装类型，其他的处理提倡使用基本类型。

7、慎用synchronized，尽量减小synchronize的方法

都知道，实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。synchronize方法被调用时，直接会把当前对象锁了，在方法执行完之前其他线程无法调用当前对象的其他方法。所以，synchronize的方法尽量减小，并且应尽量使用方法同步代替代码块同步。

9、尽量不要使用finalize方法

实际上，将资源清理放在finalize方法中完成是非常不好的选择，由于GC的工作量很大，尤其是回收Young代内存时，大都会引起应用程序暂停，所以再选择使用finalize方法进行资源清理，会导致GC负担更大，程序运行效率更差。

10、尽量使用基本数据类型代替对象

```
String str = "hello";
```

上面这种方式会创建一个“hello”字符串，而且JVM的字符缓存池还会缓存这个字符串；

```
String str = new String("hello");
```

此时程序除创建字符串外，str所引用的String对象底层还包含一个char[]数组，这个char[]数组依次存放了h,e,l,l,o

11、多线程在未发生线程安全前提下应尽量使用HashMap、ArrayList

HashTable、Vector等使用了同步机制，降低了性能。

12、尽量合理的创建HashMap

当你要创建一个比较大的hashMap时，充分利用这个构造函数

```
public HashMap(int initialCapacity, float loadFactor);
```

避免HashMap多次进行了hash重构,扩容是一件很耗费性能的事，在默认中initialCapacity只有16，而loadFactor是0.75，需要多大的容量，你最好能准确的估计你所需要的最佳大小，同样的Hashtable，Vectors也是一样的道理。

13、尽量减少对变量的重复计算

如：

```
for(int i=0;i<list.size();i++)
```

应该改为：

```
for(int i=0,len=list.size();i<len;i++)
```

并且在循环中应该避免使用复杂的表达式，在循环中，循环条件会被反复计算，如果不使用复杂表达式，而使循环条件值不变的话，程序将会运行的更快。

14、尽量避免不必要的创建

如：

```
A a = new A();
if(i==1){
    list.add(a);
}
```

应该改为：

```
if(i==1){
    A a = new A();
    list.add(a);
}
```

15、尽量在finally块中释放资源

程序中使用到的资源应当被释放，以避免资源泄漏，这最好在finally块中去做。不管程序执行的结果如何，finally块总是会执行的，以确保资源的正确关闭。

16、尽量使用移位来代替'a/b'的操作

"/"是一个代价很高的操作，使用移位的操作将会更快和更有效

如：

```
int num = a / 4;
int num = a / 8;
```

应该改为：

```
int num = a >> 2;
int num = a >> 3;
```

但注意的是使用移位应添加注释，因为移位操作不直观，比较难理解。

17、尽量使用移位来代替'a*b'的操作

同样的，对于'*'操作，使用移位的操作将会更快和更有效

如：

```
int num = a * 4;
int num = a * 8;
```

应该改为：

```
int num = a << 2;
int num = a << 3;
```

18、尽量确定StringBuffer的容量

StringBuffer 的构造器会创建一个默认大小(通常是16)的字符数组。在使用中，如果超出这个大小，就会重新分配内存，创建一个更大的数组，并将原先的数组复制过来，再丢弃旧的数组。在大多数情况下，你可以在创建 StringBuffer的时候指定大小，这样就避免了在容量不够的时候自动增长，以提高性能。

如：

```
StringBuffer buffer = new StringBuffer(1000);
```

19、尽量早释放无用对象的引用

大部分时，方法局部引用变量所引用的对象会随着方法结束而变成垃圾，因此，大部分时候程序无需将局部，引用变量显式设为null。

Tips：关注微信公众号：Java后端，每日提送技术博文。

例如：

Java代码

```
Public void test(){
Object obj = new Object();
.....
obj=null;
}
```

上面这个就没必要了，随着方法test()的执行完成，程序中obj引用变量的作用域就结束了。但是如果是改成下面：

Java代码

```
Public void test(){
Object obj = new Object();
.....
obj=null;
//执行耗时，耗内存操作；或调用耗时，耗内存的方法
.....
}
```

这时候就有必要将obj赋值为null，可以尽早的释放对Object对象的引用。

20、尽量避免使用二维数组

二维数据占用的内存空间比一维数组多得多，大概10倍以上。

21、尽量避免使用split

除非是必须的，否则应该避免使用split，split由于支持正则表达式，所以效率比较低，如果是频繁的几十，几百万的调用将会耗费大量资源，如果确实需要频繁的调用split，可以考虑使用apache的String_Utils.split(string,char)，频繁split的可以缓存结果。

22、ArrayList & LinkedList

一个是线性表，一个是链表，一句话，随机查询尽量使用ArrayList，ArrayList优于LinkedList，LinkedList还要移动指针，添加删除的操作LinkedList优于ArrayList，ArrayList还要移动数据，不过这是理论性分析，事实未必如此，重要的是理解好2者得数据结构，对症下药。

23、尽量使用System.arraycopy ()代替通过来循环复制数组

System.arraycopy() 要比通过循环来复制数组快的多。

24、尽量缓存经常使用的对象

尽可能将经常使用的对象进行缓存，可以使用数组，或HashMap的容器来进行缓存，但这种方式可能导致系统占用过多的缓存，性能下降，推荐可以使用一些第三方的开源工具，如EhCache，Oscache进行缓存，他们基本都实现了FIFO/FLU等缓存算法。

25、尽量避免非常大的内存分配

有时候问题不是由当时的堆状态造成的，而是因为分配失败造成的。分配的内存块都必须是连续的，而随着堆越来越满，找到较大的连续块越来越困难。

26、慎用异常

当创建一个异常时，需要收集一个栈跟踪(stack track)，这个栈跟踪用于描述异常是在何处创建的。构建这些栈跟踪时需要为运行时栈做一份快照，正是这一部分开销很大。当需要创建一个 Exception 时，JVM 不得不说：先别动，我想就您现在的样子存一份快照，所以暂时停止入栈和出栈操作。栈跟踪不只包含运行时栈中的一两个元素，而是包含这个栈中的每一个元素。

如果您创建一个 Exception ，就得付出代价，好在捕获异常开销不大，因此可以使用 try-catch 将核心内容包起来。从技术上讲，你甚至可以随意地抛出异常，而不用花费很大的代价。招致性能损失的并不是 throw 操作——尽管在没有预先创建异常的情况下就抛出异常是有点不寻常。真正要花代价的是创建异常，幸运的是，好的编程习惯已教会我们，不应该不管三七二十一就抛出异常。异常是为异常的情况而设计的，使用时也应该牢记这一原则。

27、尽量重用对象

特别是String对象的使用中，出现字符串连接情况时应使用StringBuffer代替，由于系统不仅要花时间生成对象，以后可能还需要花时间对这些对象进行垃圾回收和处理。因此生成过多的对象将会给程序的性能带来很大的影响。

28、不要重复初始化变量

默认情况下，调用类的构造函数时，java会把变量初始化成确定的值，所有的对象被设置成null，整数变量设置成0，float和double变量设置成0.0，逻辑值设置成false。当一个类从另一个类派生时，这一点尤其应该注意，因为用new关键字创建一个对

象时，构造函数链中的所有构造函数都会被自动调用。

这里有个注意，给成员变量设置初始值但需要调用其他方法的时候，最好放在一个方法。比如initXXX()中，因为直接调用某方法赋值可能会因为类尚未初始化而抛空指针异常，如：public int state = this.getState();

29、在java+Oracle的应用系统开发中，java中内嵌的SQL语言应尽量使用大写形式，以减少Oracle解析器的解析负担。

30、在java编程过程中，进行数据库连接，I/O流操作，在使用完毕后，及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销。

31、保证过期的对象的及时回收具有重要意义

过分的创建对象会消耗系统的大量内存，严重时，会导致内存泄漏，因此，保证过期的对象的及时回收具有重要意义。JVM的GC并非十分智能，因此建议在对象使用完毕后，手动设置成null。

32、在使用同步机制时，应尽量使用方法同步代替代码块同步。

33、不要在循环中使用Try/Catch语句，应把Try/Catch放在循环最外层

Error是获取系统错误的类，或者说是虚拟机错误的类。不是所有的错误Exception都能获取到的，虚拟机报错Exception就获取不到，必须用Error获取。

34、通过StringBuffer的构造函数来设定它的初始化容量，可以明显提升性能

StringBuffer的默认容量为16，当StringBuffer的容量达到最大容量时，它会将自身容量增加到当前的2倍+2，也就是 $2^n + 2$ 。无论何时，只要StringBuffer到达它的最大容量，它就不得不创建一个新的对象数组，然后复制旧的对象数组，这会浪费很多时间。所以给StringBuffer设置一个合理的初始化容量值，是很有必要的！

35、合理使用java.util.Vector

Vector与StringBuffer类似，每次扩展容量时，所有现有元素都要赋值到新的存储空间中。Vector的默认存储能力为10个元素，扩容加倍。

vector.add(index,obj) 这个方法可以将元素obj插入到index位置，但index以及之后的元素依次都要向下移动一个位置（将其索引加1）。除非必要，否则对性能不利。同样规则适用于remove(int index)方法，移除此向量中指定位置的元素。将所有后续元素左移（将其索引减1）。返回此向量中移除的元素。所以删除vector最后一个元素要比删除第1个元素开销低很多。删除所有元素最好用removeAllElements()方法。

如果要删除vector里的一个元素可以使用 vector.remove(obj)；而不必自己检索元素位置，再删除，如：

```
int index = indexOf (obj) ;  
  
vector.remove(index);
```

38、不用new关键字创建对象的实例

用new关键词创建类的实例时，构造函数链中的所有构造函数都会被自动调用。但如果一个对象实现了Cloneable接口，我们可以调用它的clone()方法。clone()方法不会调用任何类构造函数。

下面是Factory模式的一个典型实现：

```
public static Credit getNewCredit()
{
    return new Credit();
}
```

改进后的代码使用clone()方法：

```
private static Credit BaseCredit = new Credit();
public static Credit getNewCredit()
{
    return (Credit)BaseCredit.clone();
}
```

39、不要将数组声明为： public static final

40、HashMap的遍历

```
Map<String, String[]> paraMap = new HashMap<String, String[]>();
for( Entry<String, String[]> entry : paraMap.entrySet() )
{
    String appFieldDefId = entry.getKey();
    String[] values = entry.getValue();
}
```

利用散列值取出相应的Entry做比较得到结果，取得entry的值之后直接取key和value。

41、array(数组)和ArrayList的使用

array 数组效率最高，但容量固定，无法动态改变，ArrayList容量可以动态增长，但牺牲了效率。

42、单线程应尽量使用 HashMap, ArrayList

除非必要，否则不推荐使用HashTable,Vector，它们使用了同步机制，而降低了性能。

43、StringBuffer,StringBuilder的区别在于

java.lang.StringBuffer 线程安全的可变字符序列。一个类似于String的字符串缓冲区，但不能修改。StringBuilder与该类相比，通常应该优先使用StringBuilder类，因为它支持所有相同的操作，但由于它不执行同步，所以速度更快。

为了获得更好的性能，在构造StringBuffer或StringBuilder时应尽量指定她的容量。当然如果不超过16个字符时就不用了。 相同情况下，使用StringBuilder比使用StringBuffer仅能获得10%~15%的性能提升，但却要冒多线程不安全的风险。综合考虑还是建议使用StringBuffer。

44、尽量使用基本数据类型代替对象。

45、使用具体类比使用接口效率高，但结构弹性降低了，但现代IDE都可以解决这个问题。

46、考虑使用静态方法

如果你没有必要去访问对象的外部，那么就使你的方法成为静态方法。它会被更快地调用，因为它不需要一个虚拟函数导向表。这同时也是一个很好的实践，因为它告诉你如何区分方法的性质，调用这个方法不会改变对象的状态。

47、应尽可能避免使用内在的GET,SET方法。

48、避免枚举，浮点数的使用。

以下举几个实用优化的例子：

49、避免在循环条件中使用复杂表达式

在不做编译优化的情况下，在循环中，循环条件会被反复计算，如果不使用复杂表达式，而使循环条件值不变的话，程序将会运行的更快。例子：

```
import java.util.Vector;
class CEL {
    void method (Vector vector) {
        for (int i = 0; i < vector.size (); i++) // Violation
        ; // ...
    }
}
```

更正：

```
class CEL_fixed {
    void method (Vector vector) {
        int size = vector.size ()
        for (int i = 0; i < size; i++)
        ; // ...
    }
}
```

50、为'Vectors' 和 'Hashtables'定义初始大小

JVM为Vector扩充大小的时候需要重新创建一个更大的数组，将原原先数组中的内容复制过来，最后，原先的数组再被回收。可见Vector容量的扩大是一个颇费时间的事。

通常，默认的10个元素大小是不够的。你最好能准确的估计你所需要的最佳大小。例子：

```
import java.util.Vector;
public class DIC {
    public void addObjects (Object[] o) {
        // if length > 10, Vector needs to expand
        for (int i = 0; i < o.length; i++) {
            v.add(o); // capacity before it can add more elements.
        }
    }
    public Vector v = new Vector(); // no initialCapacity.
}
```

更正：

自己设定初始大小。

```
public Vector v = new Vector(20);
public Hashtable hash = new Hashtable(10);
```

51、在finally块中关闭Stream

程序中使用到的资源应当被释放，以避免资源泄漏。这最好在finally块中去做。不管程序执行的结果如何，finally块总是会执行的，以确保资源的正确关闭。

52、使用'System.arraycopy ()'代替通过来循环复制数组

例子：

```
public class IRB
{
void method () {
int[] array1 = new int [100];
for (int i = 0; i < array1.length; i++) {
array1 [i] = i;
}
int[] array2 = new int [100];
for (int i = 0; i < array2.length; i++) {
array2 [i] = array1 [i]; // Violation
}
}
}
```

更正：

```
public class IRB
{
void method () {
int[] array1 = new int [100];
for (int i = 0; i < array1.length; i++) {
array1 [i] = i;
}
int[] array2 = new int [100];
System.arraycopy(array1, 0, array2, 0, 100);
}
}
```

53、让访问实例内变量的getter/setter方法变成” final”

简单的getter/setter方法应该被置成final，这会告诉编译器，这个方法不会被重载，所以，可以变成” inlined” ,例子：

```
class MAF {
public void setSize (int size) {
_size = size;
}
private int _size;
}
```

更正：

```
class DAF_fixed {
final public void setSize (int size) {
_size = size;
}
private int _size;
}
```

54、对于常量字符串，用'String' 代替 'StringBuffer'

常量字符串并不需要动态改变长度。

例子：

```
public class USC {
String method () {
StringBuffer s = new StringBuffer ("Hello");
String t = s + "World!";
return t;
}
}
```

更正：把StringBuffer换成String，如果确定这个String不会再变的话，这将会减少运行开销提高性能。

55、在字符串相加的时候，使用 '' 替代 "", 如果该字符串只有一个字符的话

例子：

```
public class STR {  
    public void method(String s) {  
        String string = s + "d" // violation.  
        string = "abc" + "d" // violation.  
    }  
}
```

更正：

将一个字符的字符串替换成''

```
public class STR {  
    public void method(String s) {  
        String string = s + 'd'  
        string = "abc" + 'd'  
    }  
}
```

以上仅是Java方面编程时的性能优化，性能优化大部分都是在时间、效率、代码结构层次等方面权衡，各有利弊，不要把上面内容当成教条，或许有些对我们实际工作适用，有些不适用，还望根据实际工作场景进行取舍，活学活用，变通为宜。

【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！

Java 性能优化：35 个小细节，提升你的 Java 代码运行效率

萌小Q Java后端 2019-11-02

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 萌小Q

来源 | cnblogs.com/Qian123/p/6046096.html

[上篇](#) | [为什么我不建议你去外包公司？](#)

前言

代码优化，一个很重要的课题。可能有些人觉得没用，一些细小的地方有什么好修改的，改与不改对于代码的运行效率有什么影响呢？这个问题我是这么考虑的，就像大海里面的鲸鱼一样，它吃一条小虾米有用吗？没用，但是，吃的小虾米一多之后，鲸鱼就被喂饱了。

代码优化也是一样，如果项目着眼于尽快无BUG上线，那么此时可以抓大放小，代码的细节可以不精打细磨；但是如果足够的时间开发、维护代码，这时候就必须考虑每个可以优化的细节了，一个一个细小的优化点累积起来，对于代码的运行效率绝对是有提升的。

代码优化的目标是

1. 减小代码的体积

2. 提高代码运行的效率

代码优化细节

1. 尽量指定类、方法的final修饰符

带有final修饰符的类是不可派生的。在Java核心API中，有许多应用final的例子，例如java.lang.String，整个类都是final的。为类指定final修饰符可以让类不可以被继承，为方法指定final修饰符可以让方法不可以被重写。如果指定了一个类为final，则该类所有的方法都是final的。Java编译器会寻找机会内联所有的final方法，内联对于提升Java运行效率作用重大，具体参见Java运行期优化。此举能够使性能平均提高50%。

2. 尽量重用对象

特别是String对象的使用，出现字符串连接时应该使用StringBuilder/StringBuffer代替。由于Java虚拟机不仅要花时间生成对象，以后可能还需要花时间对这些对象进行垃圾回收和处理，因此，生成过多的对象将会给程序的性能带来很大的影响。

3. 尽可能使用局部变量

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈中速度较快，其他变量，如静态变量、实例变量等，都在堆中创建，速度较慢。另外，栈中创建的变量，随着方法的运行结束，这些内容就没了，不需要额外的垃圾回收。

4. 及时关闭流

Java编程过程中，进行数据库连接、I/O流操作时务必小心，在使用完毕后，及时关闭以释放资源。因为对这些对象的操作会造成系统大的开销，稍有不慎，将会导致严重的后果。

5. 尽量减少对变量的重复计算

明确一个概念，对方法的调用，即使方法中只有一句语句，也是有消耗的，包括创建栈帧、调用方法时保护现场、调用方法完毕时恢复现场等。所以例如下面的操作：

```
for (int i = 0; i < list.size(); i++)  
{...}
```

建议替换为：

```
for (int i = 0, int length = list.size(); i < length; i++)  
{...}
```

这样，在list.size()很大的时候，就减少了很多的消耗

6. 尽量采用懒加载的策略，即在需要的时候才创建

例如：

```
String str = "aaa"; if (i == 1)  
{  
    list.add(str);  
}
```

建议替换为：

```
if (i == 1)  
{  
    String str = "aaa";  
    list.add(str);  
}
```

7. 慎用异常

异常对性能不利。抛出异常首先要创建一个新的对象，Throwable接口的构造函数调用名为fillInStackTrace()的本地同步方法，fillInStackTrace()方法检查堆栈，收集调用跟踪信息。只要有异常被抛出，Java虚拟机就必须调整调用堆栈，因为在处理过程中创建了一个新的对象。异常只能用于错误处理，不应该用来控制程序流程。

8. 不要在循环中使用try…catch…，应该把其放在最外层

除非不得已。如果毫无理由地这么写了，只要你的领导资深一点、有强迫症一点，八成就要骂你为什么写出这种垃圾代码来了。

9. 如果能估计到待添加的内容长度，为底层以数组方式实现的集合、工具类指定初始长度

比如ArrayList、LinkedList、StringBuilder、StringBuffer、HashMap、HashSet等等，以StringBuilder为例：

- (1) StringBuilder() // 默认分配16个字符的空间
- (2) StringBuilder(int size) // 默认分配size个字符的空间
- (3) StringBuilder(String str) // 默认分配16个字符+str.length()个字符空间

可以通过类（这里指的不仅仅是上面的StringBuilder）的来设定它的初始化容量，这样可以明显地提升性能。比如StringBuilder吧，length表示当前的StringBuilder能保持的字符数量。因为当StringBuilder达到最大容量的时候，它会将自身容量增加到当前的2倍再加2，无论何时只要StringBuilder达到它的最大容量，它就不得不创建一个新的字符数组然后将旧的字符数组内容拷贝到新字符数组中——这是十分耗费性能的一个操作。试想，如果能预估到字符数组中大概要存放5000个字符而不指定长度，最接近5000的2次幂是4096，每次扩容加的2不管，那么：

- (1) 在4096 的基础上，再申请8194个大小的字符数组，加起来相当于一次申请了12290个大小的字符数组，如果一开始能指定5000个大小的字符数组，就节省了一倍以上的空间；
- (2) 把原来的4096个字符拷贝到新的的字符数组中去。

这样，既浪费内存空间又降低代码运行效率。所以，给底层以数组实现的集合、工具类设置一个合理的初始化容量是错不了的，这会带来立竿见影的效果。但是，注意，像HashMap这种是以数组+链表实现的集合，别把初始大小和你估计的大小设置得一样，因为一个table上只连接一个对象的可能性几乎为0。初始大小建议设置为2的N次幂，如果能估计到有2000个元素，设置成new HashMap(128)、new HashMap(256)都可以。

10. 当复制大量数据时，使用System.arraycopy()命令

11. 乘法和除法使用移位操作

例如：

```
for (val = 0; val < 100000; val += 5)
{
    a = val * 8;
    b = val / 2;
}
```

用移位操作可以极大地提高性能，因为在计算机底层，对位的操作是最方便、最快的，因此建议修改为：

```
for (val = 0; val < 100000; val += 5)
{
    a = val << 3;
    b = val >> 1;
}
```

移位操作虽然快，但是可能会使代码不太好理解，因此最好加上相应的注释。

12. 循环内不要不断创建对象引用

例如：

```
for (int i = 1; i <= count; i++)
{Object obj = new Object();
}
```

这种做法会导致内存中有count份Object对象引用存在，count很大的话，就耗费内存了，建议为改为：

```
Object obj = null;
for (int i = 0; i <= count; i++) { obj = new Object();}
```

这样的话，内存中只有一份Object对象引用，每次new Object()的时候，Object对象引用指向不同的Object罢了，但是内存中只有一份，这样就大大节省了内存空间了。

13. 基于效率和类型检查的考虑，应该尽可能使用array，无法确定数组大小时才使用ArrayList

14. 尽量使用HashMap、ArrayList、StringBuilder，除非线程安全需要，否则不推荐使用Hashtable、Vector、StringBuffer，后三者由于使用同步机制而导致了性能开销

15. 不要将数组声明为public static final

因为这毫无意义，这样只是定义了引用为static final，数组的内容还是可以随意改变的，将数组声明为public更是一个安全漏洞，这意味着这个数组可以被外部类所改变。

16. 尽量在合适的场合使用单例

使用单例可以减轻加载的负担、缩短加载的时间、提高加载的效率，但并不是所有地方都适用于单例，简单来说，单例主要适用于以下三个方面：

- (1) 控制资源的使用，通过线程同步来控制资源的并发访问
- (2) 控制实例的产生，以达到节约资源的目的
- (3) 控制数据的共享，在不建立直接关联的条件下，让多个不相关的进程或线程之间实现通信

17. 尽量避免随意使用静态变量

要知道，当某个对象被定义为static的变量所引用，那么gc通常是不会回收这个对象所占有的堆内存的，如：

```
public class A
{
private static B b = new B();
}
```

此时静态变量b的生命周期与A类相同，如果A类不被卸载，那么引用B指向的B对象会常驻内存，直到程序终止

18. 及时清除不再需要的会话

为了清除不再活动的会话，许多应用服务器都有默认的会话超时时间，一般为30分钟。当应用服务器需要保存更多的会话时，如果内存不足，那么操作系统会把部分数据转移到磁盘，应用服务器也可能根据MRU（最近最频繁使用）算法把部分不活跃的会话转储到磁盘，甚至可能抛出内存不足的异常。如果会话要被转储到磁盘，那么必须要先被序列化，在大规模集群中，对对象进行序列化的代价是很昂贵的。因此，当会话不再需要时，应当及时调用 HttpSession 的 invalidate() 方法清除会话。

19. 实现 RandomAccess 接口的集合比如 ArrayList，应当使用最普通的 for 循环而不是 foreach 循环来遍历

这是JDK推荐给用户的。JDK API对于 RandomAccess 接口的解释是：实现 RandomAccess 接口用来表明其支持快速随机访问，此接口的主要目的是允许一般的算法更改其行为，从而将其应用到随机或连续访问列表时能提供良好的性能。实际经验表明，实现 RandomAccess 接口的类实例，假如是随机访问的，使用普通 for 循环效率将高于使用 foreach 循环；反过来，如果是顺序访问的，则使用 Iterator 会效率更高。可以使用类似如下的代码作判断

```
if (list instanceof RandomAccess)
{
    for (int i = 0; i < list.size(); i++)
}
else
    Iterator<?> iterator = list.iterator();
    while (iterator.hasNext()) {iterator.next()}
}
```

foreach 循环的底层实现原理就是迭代器 Iterator，参见 Java 语法糖 1：可变长度参数以及 foreach 循环原理。所“以后半句”反过来，如果是顺序访问的，则使用 Iterator 会效率更高”的意思就是顺序访问的那些类实例，使用 foreach 循环去遍历。

20. 使用同步代码块替代同步方法

这点在多线程模块中的 synchronized 锁方法块一文中已经讲得很清楚了，除非能确定一整个方法都是需要进行同步的，否则尽量使用同步代码块，避免对那些不需要进行同步的代码也进行了同步，影响了代码执行效率。

21. 将常量声明为 static final，并以大写命名

这样在编译期间就可以把这些内容放入常量池中，避免运行期间计算生成常量的值。另外，将常量的名字以大写命名也可以方便区分出常量与变量

22. 不要创建一些不使用的对象，不要导入一些不使用的类

这毫无意义，如果代码中出现“`The value of the local variable i is not used`”、“`The import java.util is never used`”，那么请删除这些无用的内容

23. 程序运行过程中避免使用反射

关于，请参见反射。反射是 Java 提供给用户一个很强大的功能，功能强大往往意味着效率不高。不建议在程序运行过程中使用尤其是频繁使用反射机制，特别是 Method 的 invoke 方法，如果确实有必要，一种建议性的做法是将那些需要通过反射加载的类在项目启动的时候通过反射实例化出一个对象并放入内存——用户只关心和对端交互的时候获取最快的响应速度，并不关心对端的项目启动花多久时间。

24. 使用数据库连接池和线程池

这两个池都是用于重用对象的，前者可以避免频繁地打开和关闭连接，后者可以避免频繁地创建和销毁线程

25. 使用带缓冲的输入输出流进行 IO 操作

带缓冲的输入输出流，即 BufferedReader、BufferedWriter、BufferedInputStream、BufferedOutputStream，这可以极大地提升 IO 效率

26. 顺序插入和随机访问比较多的场景使用 ArrayList，元素删除和中间插入比较多的场景使用 LinkedList 这个，理解 ArrayList 和 LinkedList 的原理就知道了

27. 不要让public方法中有太多的形参

public方法即对外提供的方法，如果给这些方法太多形参的话主要有两点坏处：

1、违反了面向对象的编程思想，Java讲求一切都是对象，太多的形参，和面向对象的编程思想并不契合

2、参数太多势必导致方法调用的出错概率增加

至于这个“太多”指的是多少个，3、4个吧。比如我们用JDBC写一个insertStudentInfo方法，有10个学生信息字段要插入Student表中，可以把这10个参数封装在一个实体类中，作为insert方法的形参。

28. 字符串变量和字符串常量equals的时候将字符串常量写在前面

这是一个比较常见的小技巧了，如果有以下代码：

```
String str = "123";
if(str.equals("123")) {...}
```

建议修改为：

```
String str = "123";
if ("123".equals(str))
{
...
}
```

这么做主要是可以避免空指针异常

29. 请知道，在java中if (i == 1)和if (1 == i)是没有区别的，但从阅读习惯上讲，建议使用前者

平时有人问，“if (i == 1)”和“if (1 == i)”有没有区别，这就要从C/C++讲起。

在C/C++中，“if (i == 1)”判断条件成立，是以0与非0为基准的，0表示false，非0表示true，如果有这么一段代码：

```
int i = 2;
if(i == 1)
{
...
}else{
...
}
```

C/C++判断“i==1”不成立，所以以0表示，即false。但是如果：

```
int i = 2;
if (i = 1) { ... }else{ ... }
```

万一程序员一个不小心，把“if (i == 1)”写成“if (i = 1)”，这样就有问题了。在if之内将i赋值为1，if判断里面的内容非0，返回的就是true了，但是明明i为2，比较的值是1，应该返回的false。这种情况在C/C++的开发中是很可能发生的并且会导致一些难以理解的错误产生，所以，为了避免开发者在if语句中不正确的赋值操作，建议将if语句写为：

```
int i = 2;
if (1 == i) { ... }else{ ... }
```

这样，即使开发者不小心写成了“1=i”，C/C++编译器也可以第一时间检查出来，因为我们可以对一个变量赋值i为1，但是不能对一个常量赋值1为i。

但是，在Java中，C/C++这种“if (i = 1)”的语法是不可能出现的，因为一旦写了这种语法，Java就会编译报错“Type mismatch: cannot convert from int to boolean”。但是，尽管Java的“if (i == 1)”和“if (1 == i)”在语义上没有任何区别，但是从阅读习惯上讲，建议使用前者会更好些。

30. 不要对数组使用toString()方法

看一下对数组使用toString()打印出来的是什么：

```
public static void main(String[] args)
{
int[] is = new int[]{1, 2, 3};
System.out.println(is.toString());
}
```

结果是：

```
[I@18a992f
```

本意是想打印出数组内容，却有可能因为数组引用is为空而导致空指针异常。不过虽然对数组toString()没有意义，但是对集合toString()是可以打印出集合里面的内容的，因为集合的父类AbstractCollections重写了Object的toString()方法。

31、不要对超出范围的基本数据类型做向下强制转型

这绝不会得到想要的结果：

```
public static void main(String[] args)
{
long l = 12345678901234L;
int i = (int)l;
System.out.println(i);
}
```

我们可能期望得到其中的某几位，但是结果却是：

```
1942892530
```

解释一下。Java中long是8个字节64位的，所以12345678901234在计算机中的表示应该是：

```
0000 0000 0000 0000 0000 1011 0011 1010 0111 0011 1100 1110 0010 1111 1111 0010
```

一个int型数据是4个字节32位的，从低位取出上面这串二进制数据的前32位是：

```
0111 0011 1100 1110 0010 1111 1111 0010
```

这串二进制表示为十进制1942892530，所以就是我们上面的控制台上输出的内容。从这个例子上还能顺便得到两个结论

- 1、整型默认的数据类型是int，long l = 12345678901234L，这个数字已经超出了int的范围了，所以最后有一个L，表示这是一个long型数。顺便，浮点型的默认类型是double，所以定义float的时候要写成”float f = 3.5f”
- 2、接下来再写一句”int ii = l + i;”会报错，因为long + int是一个long，不能赋值给int

32、公用的集合类中不使用的数据一定要及时remove掉

如果一个集合类是公用的（也就是说不是方法里面的属性），那么这个集合里面的元素是不会自动释放的，因为始终有引用指向它们。所以，如果公用集合里面的某些数据不使用而不去remove掉它们，那么将会造成这个公用集合不断增大，使得系统有内存泄露的隐患。

33、把一个基本数据类型转为字符串，基本数据类型.toString()是最快的方式、String.valueOf(数据)次之、数据+””最慢

把一个基本数据类型转为一般有三种方式，我有一个Integer型数据i，可以使用i.toString()、String.valueOf(i)、i+””三种方式，三种方式的效率如何，看一个测试：

```

public static void main(String[] args)
{
int loopTime = 50000;
Integer i = 0;

long startTime = System.currentTimeMillis();

for (int j = 0; j < loopTime; j++) {

String str = String.valueOf(i);
}

System.out.println("String.valueOf(): " + (System.currentTimeMillis() - startTime) + "ms");
startTime = System.currentTimeMillis();for (int j = 0; j < loopTime; j++)

{

String str = i.toString();
}

System.out.println("Integer.toString(): " + (System.currentTimeMillis() - startTime) + "ms");
startTime = System.currentTimeMillis();

for (int j = 0; j < loopTime; j++)
{

String str = i + "";
}

System.out.println("i + \"\": " + (System.currentTimeMillis() - startTime) + "ms");
}

```

运行结果为：

```
String.valueOf(): 11ms Integer.toString(): 5ms i + "": 25ms
```

所以以后遇到把一个基本数据类型转为String的时候，优先考虑使用toString()方法。至于为什么，很简单：

1、String.valueOf()方法底层调用了Integer.toString()方法，但是会在调用前做空判断

2、Integer.toString()方法就不说了，直接调用了

3、i + “”底层使用了StringBuilder实现，先用append方法拼接，再用toString()方法获取字符串

三者对比下来，明显是2最快、1次之、3最慢

34. 使用最有效率的方式去遍历Map

遍历Map的方式有很多，通常场景下我们需要的是遍历Map中的Key和Value，那么推荐使用的、效率最高的方式是：

```

public static void main(String[] args)
{
HashMap<String, String> hm = new HashMap<String, String>();
hm.put("111", "222");

Set<Map.Entry<String, String>> entrySet = hm.entrySet();

Iterator<Map.Entry<String, String>> iter = entrySet.iterator();

while (iter.hasNext())
{
    Map.Entry<String, String> entry = iter.next();
    System.out.println(entry.getKey() + "\t" + entry.getValue());
}
}

```

如果你只是想遍历一下这个Map的key值，那用“Set keySet = hm.keySet();”会比较合适一些

35. 对资源的close()建议分开操作

意思是，比如我有这么一段代码：

```
try{  
    XXX.close();  
    YYY.close();  
}catch (Exception e)  
{...}
```

建议修改为：

```
try{ XXX.close(); }catch (Exception e) { ... }try{ YYY.close(); }catch (Exception e) { ... }
```

虽然有些麻烦，却能避免资源泄露。我想，如果没有修改过的代码，万一XXX.close()抛异常了，那么就进入了catch块中了，YYY.close()不会执行，YYY这块资源就不会回收了，一直占用着，这样的代码一多，是可能引起资源句柄泄露的。而改为上面的写法之后，就保证了无论如何XXX和YYY都会被close掉。

-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!

Java 性能优化：教你提高代码运行的效率

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

作者 | 五月的仓颉

链接 | cnblogs.com/xrq730/p/4865416.html

推荐 | Java 8 的这个特性, 用起来真的很爽 ([点击查看](#))

我认为, 代码优化的最重要的作用应该是: **避免未知的错误**。在代码上线运行的过程中, 往往会出现很多我们意想不到的错误, 因为线上环境和开发环境是非常不同的, 错误定位到最后往往是一个非常小的原因。

然而为了解决这个错误, 我们需要先自验证、再打包出待替换的class文件、暂停业务并重启, 对于一个成熟的项目而言, 最后一条其实影响是非常大的, 这意味着这段时间用户无法访问应用。因此, 在写代码的时候, 从源头开始注意各种细节, 权衡并使用最优的选择, 将会很大程度上避免出现未知的错误, 从长远看也极大的降低了工作量。

代码优化的目标是:

- 减小代码的体积
- 提高代码运行的效率

本文的内容有些来自网络, 有些来自平时工作和学习, 当然这不重要, 重要的是这些代码优化的细节是否真真正正地有用。那本文会保持长期更新, 只要有遇到值得分享的代码优化细节, 就会不定时地更新此文。

代码优化细节

1、尽量指定类、方法的final修饰符

带有final修饰符的类是不可派生的。在Java核心API中, 有许多应用final的例子, 例如java.lang.String, 整个类都是final的。为类指定final修饰符可以让类不可以被继承, 为方法指定final修饰符可以让方法不可以被重写。如果指定了一个类为final, 则该类所有的方法都是final的。Java编译器会寻找机会内联所有的final方法, 内联对于提升Java运行效率作用重大, 具体参见Java运行期优化。此举能够使性能平均提高50%。

2、尽量重用对象

特别是String对象的使用, 出现字符串连接时应该使用StringBuilder/StringBuffer代替。由于Java虚拟机不仅要花时间生成对象, 以后可能还需要花时间对这些对象进行垃圾回收和处理, 因此, 生成过多的对象将会给程序的性能带来很大的影响。

3、尽可能使用局部变量

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈中, 速度较快, 其他变量, 如静态变量、实例变量等, 都在堆中创建, 速度较慢。另外, 栈中创建的变量, 随着方法的运行结束, 这些内容就没了, 不需要额外的垃圾回收。

4、及时关闭流

Java编程过程中, 进行数据库连接、I/O流操作时务必小心, 在使用完毕后, 及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销, 稍有不慎, 将会导致严重的后果。

5、尽量减少对变量的重复计算

明确一个概念，对方法的调用，即使方法中只有一句语句，也是有消耗的，包括创建栈帧、调用方法时保护现场、调用方法完毕时恢复现场等。所以例如下面的操作：

```
1 for (int i = 0; i < list.size(); i++)
2 {...}
```

建议替换为：

```
1 for (int i = 0, length = list.size(); i < length; i++
2 )
3 {...}
```

这样，在list.size()很大的时候，就减少了很多的消耗

6、尽量采用懒加载的策略，即在需要的时候才创建

例如：

```
1 String str = "aaa"
2 ;
3 if (i == 1
4 )
5 {
6     list.add(str);
7 }
```

建议替换为：

```
1 if (i == 1
2 )
3 {
4     String str = "aaa"
5 ;
6     list.add(str);
7 }
```

7、慎用异常

异常对性能不利。抛出异常首先要创建一个新的对象，`Throwable`接口的构造函数调用名为 `fillInStackTrace()` 的本地同步方法，`fillInStackTrace()` 方法检查堆栈，收集调用跟踪信息。只要有异常被抛出，Java虚拟机就必须调整调用堆栈，因为在处理过程中创建了一个新的对象。异常只能用于错误处理，不应该用来控制程序流程。

8、不要在循环中使用try…catch…，应该把其放在最外层

根据网友们提出的意见，这一点我认为值得商榷，欢迎大家提出看法！

9、如果能估计到待添加的内容长度，为底层以数组方式实现的集合、工具类指定初始长度

比如ArrayList、LinkedList、StringBuilder、StringBuffer、HashMap、HashSet等等，以StringBuilder为例：

```
1 StringBuilder()          // 默认分配16个字符的空间  
2 StringBuilder(int size) // 默认分配size个字符的空间  
3 StringBuilder(String str) // 默认分配16个字符+str.length()个字符空间
```

可以通过类（这里指的不仅仅是上面的StringBuilder）的构造函数来设定它的初始化容量，这样可以明显地提升性能。比如StringBuilder吧，length表示当前的StringBuilder能保持的字符数量。

因为当StringBuilder达到最大容量的时候，它会将自身容量增加到当前的2倍再加2，无论何时只要StringBuilder达到它的最大容量，它就不得不创建一个新的字符数组然后将旧的字符数组内容拷贝到新字符数组中----这是十分耗费性能的一个操作。试想，如果能预估到字符数组中大概要存放5000个字符而不指定长度，最接近5000的2次幂是4096，每次扩容加的2不管，那么：

- 在4096 的基础上，再申请8194个大小的字符数组，加起来相当于一次申请了12290个大小的字符数组，如果一开始能指定5000个大小的字符数组，就节省了一倍以上的空间
- 把原来的4096个字符拷贝到新的的字符数组中去

这样，既浪费内存空间又降低代码运行效率。所以，给底层以数组实现的集合、工具类设置一个合理的初始化容量是错不了的，这会带来立竿见影的效果。

但是，注意，像HashMap这种是以数组+链表实现的集合，别把初始大小和你估计的大小设置得一样，因为一个table上只连接一个对象的可能性几乎为0。初始大小建议设置为2的N次幂，如果能估计到有2000个元素，设置成 new HashMap(128)、new HashMap(256) 都可以。

10、当复制大量数据时，使用 `System.arraycopy()` 命令

11、乘法和除法使用移位操作

例如：

```
1 for (val = 0; val < 100000; val += 5  
2 )  
3 {  
4     a = val * 8  
5 ;  
6     b = val / 2  
7 ;  
8 }
```

用移位操作可以极大地提高性能，因为在计算机底层，对位的操作是最方便、最快的，因此建议修改为：

```
1 for (val = 0; val < 100000; val += 5  
2 )  
3 {  
4     a = val << 3  
5 ;  
6     b = val >> 1  
7 ;
```

```
}
```

移位操作虽然快，但是可能会使代码不太好理解，因此最好加上相应的注释。

12、循环内不要不断创建对象引用

例如：

```
1 for (int i = 1; i <= count; i++)  
2 {  
3     Object obj = new Object();  
4 }  
5  
6 }
```

这种做法会导致内存中有count份Object对象引用存在，count很大的话，就耗费内存了，建议为改为：

```
1 Object obj = null  
2 ;  
3 for (int i = 0; i <= count; i++)  
4 {  
5     obj = new Object()  
6 ;  
7 }
```

这样的话，内存中只有一份Object对象引用，每次 `new Object()` 的时候，Object对象引用指向不同的Object罢了，但是内存中只有一份，这样就大大节省了内存空间了。

13、基于效率和类型检查的考虑，应该尽可能使用array，无法确定数组大小时才使用ArrayList

14、尽量使用HashMap、ArrayList、StringBuilder，除非线程安全需要，否则不推荐使用Hashtable、Vector、StringBuffer，后三者由于使用同步机制而导致了性能开销

15、不要将数组声明为public static final

因为这毫无意义，这样只是定义了引用为 `static final`，数组的内容还是可以随意改变的，将数组声明为public更是一个安全漏洞，这意味着这个数组可以被外部类所改变

16、尽量在合适的场合使用单例

使用单例可以减轻加载的负担、缩短加载的时间、提高加载的效率，但并不是所有地方都适用于单例，简单来说，单例主要适用于以下三个方面：

- 控制资源的使用，通过线程同步来控制资源的并发访问
- 控制实例的产生，以达到节约资源的目的
- 控制数据的共享，在不建立直接关联的条件下，让多个不相关的进程或线程之间实现通信

17、尽量避免随意使用静态变量

要知道，当某个对象被定义为static的变量所引用，那么gc通常是不会回收这个对象所占有的堆内存的，如：

```
1 public class A
2 {
3     private static B b = new B();
4 }
```

此时静态变量b的生命周期与A类相同，如果A类不被卸载，那么引用B指向的B对象会常驻内存，直到程序终止

18、及时清除不再需要的会话

为了清除不再活动的会话，许多应用服务器都有默认的会话超时时间，一般为30分钟。当应用服务器需要保存更多的会话时，如果内存不足，那么操作系统会把部分数据转移到磁盘，应用服务器也可能根据MRU（最近最频繁使用）算法把部分不活跃的会话转储到磁盘，甚至可能抛出内存不足的异常。如果会话要被转储到磁盘，那么必须要先被序列化，在[大规模集群](#)中，对对象进行序列化的代价是很昂贵的。因此，当会话不再需要时，应当及时调用HttpSession的 `invalidate()` 方法清除会话。

19、实现RandomAccess接口的集合比如ArrayList，应当使用最普通的for循环而不是foreach循环来遍历

这是JDK推荐给用户的。JDK API对于 `RandomAccess` 接口的解释是：实现 `RandomAccess` 接口用来表明其支持快速随机访问，此接口的主要目的是允许一般的算法更改其行为，从而将其应用到随机或连续访问列表时能提供良好的性能。

实际经验表明，实现RandomAccess接口的类实例，假如是随机访问的，使用普通for循环效率将高于使用foreach循环；反过来，如果是顺序访问的，则使用Iterator会效率更高。可以使用类似如下的代码作判断：

```
1 if (list instanceof RandomAccess)
2 {
3     for (int i = 0; i < list.size(); i++){}
4 }
5 else
6 {
7     Iterator<?> iterator = list.iterator();
8     while (iterator.hasNext())iterator.next()
9 }
```

foreach循环的底层实现原理就是迭代器Iterator，参见[Java语法糖1：可变长度参数以及foreach循环原理](#)。所以后半句“反过来，如果是顺序访问的，则使用Iterator会效率更高”的意思就是顺序访问的那些类实例，使用foreach循环去遍历。

<http://www.cnblogs.com/xrq730/p/4868465.html>

20、使用同步代码块替代同步方法

这点在多线程模块中的[synchronized锁方法块](#)一文中已经讲得很清楚了，除非能确定一整个方法都是需要进行同步的，否则尽量使用同步代码块，避免对那些不需要进行同步的代码也进行了同步，影响了代码执行效率。

<http://www.cnblogs.com/xrq730/p/4851530.html>

21、将常量声明为static final，并以大写命名

这样在编译期间就可以把这些内容放入常量池中，避免运行期间计算生成常量的值。另外，将常量的名字以大写命名也可以方便区分出常量与变量

22、不要创建一些不使用的对象，不要导入一些不使用的类

这毫无意义，如果代码中出现 **The value of the local variable i is not used**、**"The import java.util is never used**，那么请删除这些无用的内容

23、程序运行过程中避免使用反射

关于，请参见[反射](#)。反射是Java提供给用户一个很强大的功能，功能强大往往意味着效率不高。不建议在程序运行过程中使用尤其是频繁使用反射机制，特别是Method的invoke方法，如果确实有必要，一种建议性的做法是将那些需要通过反射加载的类在项目启动的时候通过反射实例化出一个对象并放入内存----用户只关心和对端交互的时候获取最快的响应速度，并不关心对端的项目启动花多久时间。

<http://www.cnblogs.com/xrq730/p/4862111.html>

24、使用数据库连接池和线程池

这两个池都是用于重用对象的，前者可以避免频繁地打开和关闭连接，后者可以避免频繁地创建和销毁线程

25、使用带缓冲的输入输出流进行IO操作

带缓冲的输入输出流，即BufferedReader、BufferedWriter、BufferedInputStream、BufferedOutputStream，这可以极大地提升IO效率

26、顺序插入和随机访问比较多的场景使用ArrayList，元素删除和中间插入比较多的场景使用LinkedList

这个，理解ArrayList和LinkedList的原理就知道了

27、不要让public方法中有太多的形参

public方法即对外提供的方法，如果给这些方法太多形参的话主要有两点坏处：

- 违反了面向对象的编程思想，Java讲求一切都是对象，太多的形参，和面向对象的编程思想并不契合
- 参数太多势必导致方法调用的出错概率增加

至于这个"太多"指的是多少个，3、4个吧。比如我们用JDBC写一个insertStudentInfo方法，有10个学生信息字段要插入Student表中，可以把这10个参数封装在一个实体类中，作为insert方法的形参

28、字符串变量和字符串常量equals的时候将字符串常量写在前面

这是一个比较常见的小技巧了，如果有以下代码：

```
1 String str = "123"
2 ;
3 if (str.equals("123"))
4 {
5     ...
6 }
```

建议修改为：

```
1 String str = "123"
2 ;
3 if ("123".equals(str))
```

```
4 {  
5     ...  
6 }
```

这么做主要是可以避免空指针异常

29、请知道，在java中 if (*i == 1*) 和 if (*1 == i*) 是没有区别的，但从阅读习惯上讲，建议使用前者

平时有人问， if (*i == 1*) 和 if (*1 == i*) 有没有区别，这就要从C/C++讲起。

在C/C++中， if (*i == 1*) 判断条件成立，是以0与非0为基准的，0表示false，非0表示true，如果有这么一段代码：

```
1 int i = 2  
2 ;  
3 if (i == 1  
4 )  
5 {  
6     ...  
7 }  
8 else  
9 {  
    ...  
}
```

C/C++判断 *i==1* 不成立，所以以0表示，即false。但是如果：

```
1 int i = 2  
2 ;  
3 if (i = 1  
4 )  
5 {  
6     ...  
7 }  
8 else  
9 {  
    ...  
}
```

万一程序员一个不小心，把 if (*i == 1*) 写成 if (*i = 1*) ，这样就有问题了。在if之内将i赋值为1，if判断里面的内容非0，返回的就是true了，但是明明i为2，比较的值是1，应该返回的false。这种情况在C/C++的开发中是很可能发生的并且会导致一些难以理解的错误产生，所以，为了避免开发者在if语句中不正确的赋值操作，建议将if语句写为：

```
1 int i = 2  
2 ;  
3 if (1 == i)  
4 {  
    ...  
6 }  
7 else  
{
```

```
8 ...
9 }
```

这样，即使开发者不小心写成了 `i = 1`，C/C++编译器也可以第一时间检查出来，因为我们可以对一个变量赋值为1，但是不能对一个常量赋值为1。

但是，在Java中，C/C++这种 `if (i = 1)` 的语法是不可能出现的，因为一旦写了这种语法，Java就会编译报错 `Type mismatch: cannot convert from int to boolean`。但是，尽管Java的 `if (i == 1)` 和 `if (1 == i)` 在语义上没有任何区别，从阅读习惯上讲，建议使用前者会更好些。

30、不要对数组使用 `toString()` 方法

看一下对数组使用 `toString()` 打印出来的是什么：

```
1 public static void main(String[] args)
2 {
3     int[] is = new int[]{1, 2, 3}
4 ;
5     System.out.println(is.toString());
6 }
```

结果是：

```
[I@18a992f
```

本意是想打印出数组内容，却有可能因为数组引用is为空而导致空指针异常。不过虽然对数组 `toString()` 没有意义，但是对集合 `toString()` 是可以打印出集合里面的内容的，因为集合的父类 `AbstractCollections<E>` 重写了Object的 `toString()` 方法。

31、不要对超出范围的基本数据类型做向下强制转型

这绝不会得到想要的结果：

```
1 public static void main(String[] args)
2 {
3     long l = 12345678901234L
4 ;
5     int i = (int)l
6 ;
7     System.out.println(i);
8 }
```

我们可能期望得到其中的某几位，但是结果却是：

```
1942892530
```

解释一下。Java中long是8个字节64位的，所以12345678901234在计算机中的表示应该是：

```
0000 0000 0000 0000 1011 0011 1010 0111 0011 1100 1110 0010 1111 1111 0010
```

一个int型数据是4个字节32位的，从低位取出上面这串二进制数据的前32位是：

```
0111 0011 1100 1110 0010 1111 1111 0010
```

这串二进制表示为十进制1942892530，所以就是我们上面的控制台上输出的内容。从这个例子上还能顺便得到两个结论：

- 整型默认的数据类型是 `int, long l = 12345678901234L`，这个数字已经超出了int的范围了，所以最后有一个L，表示这是一个long型数。顺便，浮点型的默认类型是double，所以定义float的时候要写成 `float f = 3.5f`
- 接下来再写一句 `int ii = l + i;` 会报错，因为long + int是一个long，不能赋值给int

32、公用的集合类中不使用的数据一定要及时remove掉

如果一个集合类是公用的（也就是说不是方法里面的属性），那么这个集合里面的元素是不会自动释放的，因为始终有引用指向它们。所以，如果公用集合里面的某些数据不使用而去remove掉它们，那么将会造成这个公用集合不断增大，使得系统有内存泄露的隐患。

33、把一个基本数据类型转为字符串，`基本数据类型.toString()`是最快的方式、`String.valueOf(数据)`次之、`数据+""`最慢

把一个基本数据类型转为一般有三种方式，我有一个Integer型数据i，可以使用 `i.toString()`、`String.valueOf(i)`、`i + ""` 三种方式，三种方式的效率如何，看一个测试：

```
1 public static void main(String[] args)
2 {
3     int loopTime = 50000
4 ;
5     Integer i = 0
6 ;
7     long startTime = System.currentTimeMillis();
8     for (int j = 0; j < loopTime; j++)
9     {
10         String str = String.valueOf(i);
11     }
12     System.out.println("String.valueOf(): " + (System.currentTimeMillis() - startTime) + "ms")
13 ;
14     startTime = System.currentTimeMillis();
15     for (int j = 0; j < loopTime; j++)
16     {
17         String str = i.toString();
18     }
19     System.out.println("Integer.toString(): " + (System.currentTimeMillis() - startTime) + "ms")
20 ;
21     startTime = System.currentTimeMillis();
22     for (int j = 0; j < loopTime; j++)
23     {
24         String str = i + ""
25     }
26     System.out.println("i + \"\": " + (System.currentTimeMillis() - startTime) + "ms")
27 ;
28 }
```

运行结果为：

```
String.valueOf(): 11ms  
Integer.toString(): 5ms  
i + "": 25ms
```

所以以后遇到把一个基本数据类型转为String的时候，优先考虑使用 `toString()` 方法。至于为什么，很简单：

- `String.valueOf()` 方法底层调用了 `Integer.toString()` 方法，但是会在调用前做空判断
- `Integer.toString()` 方法就不说了，直接调用了
- `i + ""` 底层使用了StringBuilder实现，先用append方法拼接，再用 `toString()` 方法获取字符串

三者对比下来，明显是2最快、1次之、3最慢

34、使用最有效率的方式去遍历Map

遍历Map的方式有很多，通常场景下我们需要的是遍历Map中的Key和Value，那么推荐使用的、效率最高的方式是：

```
1 public static void main(String[] args)  
2 {  
3     HashMap<String, String> hm = new HashMap<String, String>()  
4 ;  
5     hm.put("111", "222")  
6 ;  
7  
8     Set<Map.Entry<String, String>> entrySet = hm.entrySet();  
9     Iterator<Map.Entry<String, String>> iter = entrySet.iterator();  
10    while (iter.hasNext())  
11    {  
12        Map.Entry<String, String> entry = iter.next();  
13        System.out.println(entry.getKey() + "\t" + entry.getValue());  
14    }  
15 }
```

如果你只是想遍历一下这个Map的key值，那用 `Set<String> keySet = hm.keySet();` 会比较合适一些

35、对资源的 `close()` 建议分开操作

意思是，比如我有这么一段代码：

```
1 try  
2 {  
3     XXX.close();  
4     YYY.close();  
5 }  
6 catch (Exception e)  
7 {  
8     ...  
9 }
```

建议修改为：

```
1 try
2 {
3     XXX.close();
4 }
5 catch (Exception e)
6 {
7     ...
8 }
9 try
10 {
11     YYY.close();
12 }
13 catch (Exception e)
14 {
15     ...
16 }
```

虽然有些麻烦，却能避免资源泄露。我们想，如果没有修改过的代码，万一 `XXX.close()` 抛异常了，那么就进入了catch块中了，`YYY.close()` 不会执行，YYY这块资源就不会回收了，一直占用着，这样的代码一多，是可能引起资源句柄泄露的。而改为下面的写法之后，就保证了无论如何XXX和YYY都会被close掉

36、对于ThreadLocal使用前或者使用后一定要先remove

当前基本所有的项目都使用了线程池技术，这非常好，可以动态配置线程数、可以重用线程。

然而，如果你在项目中使用到了ThreadLocal，一定要记得使用前或者使用后remove一下。这是因为上面提到了线程池技术做的是一个线程重用，这意味着代码运行过程中，一条线程使用完毕，并不会被销毁而是等待下一次的使用。我们看一下Thread类中，持有 `ThreadLocal.ThreadLocalMap` 的引用：

```
1 /* ThreadLocal values pertaining to this thread. This map is maintained
2  * by the ThreadLocal class. */
3 ThreadLocal.ThreadLocalMap threadLocals = null
;
```

线程不销毁意味着上条线程set的 `ThreadLocal.ThreadLocalMap` 中的数据依然存在，那么在下一条线程重用这个Thread的时候，很可能get到的是上条线程set的数据而不是自己想要的内容。

这个问题非常隐晦，一旦出现这个原因导致的错误，没有相关经验或者没有扎实的基础非常难发现这个问题，因此在写代码的时候就要注意这一点，这将给你后续减少很多的工作量。

37、切记以常量定义的方式替代魔鬼数字，魔鬼数字的存在将极大地降低代码可读性，字符串常量是否使用常量定义可以视情况而定

38、long或者Long初始赋值时，使用大写的L而不是小写的l，因为字母l极易与数字1混淆，这个点非常细节，值得注意

39、所有重写的方法必须保留 @Override 注解

这么做有三个原因：

- 清楚地可以知道这个方法由父类继承而来
- `getObject()` 和 `getobject()` 方法，前者第四个字母是"O"，后者第四个字母是"0"，加了 `@Override` 注解可以马上判断是否重写

成功

- 在抽象类中对方法签名进行修改，实现类会马上报出编译错误

40、推荐使用JDK7中新引入的Objects工具类来进行对象的equals比较，直接 a.equals(b)，有空指针异常的风险

41、循环体内不要使用" +"进行字符串拼接，而直接使用StringBuilder不断append

说一下不使用" +"进行字符串拼接的原因，假如我有一个方法：

```
1 public String appendStr(String oriStr, String... appendStrs) {  
2     if (appendStrs == null || appendStrs.length == 0)  
3     {  
4         return oriStr;  
5     }  
6  
7     for (String appendStr : appendStrs) {  
8         oriStr += appendStr;  
9     }  
10  
11    return oriStr;  
}
```

将这段代码编译之后的.class文件，使用javap -c进行反编译一下，截取关键的一部分：

意思就是每次虚拟机碰到" +"这个操作符对字符串进行拼接的时候，会new出一个StringBuilder，然后调用append方法，最后调用 `toString()` 方法转换字符串赋值给oriStr对象，即循环多少次，就会new出多少个 `StringBuilder()` 来，这对于内存是一种浪费。

42、不捕获Java类库中定义的继承自RuntimeException的运行时异常类

异常处理效率低，`RuntimeException`的运行时异常类，其中绝大多数完全可以由程序员来规避，比如：

- `ArithmaticException`可以通过判断除数是否为空来规避
- `NullPointerException`可以通过判断对象是否为空来规避
- `IndexOutOfBoundsException`可以通过判断数组/字符串长度来规避
- `ClassCastException`可以通过`instanceof`关键字来规避
- `ConcurrentModificationException`可以使用迭代器来规避

43、避免Random实例被多线程使用，虽然共享该实例是线程安全的，但会因竞争同一seed导致的性能下降，JDK7之后，可以使用ThreadLocalRandom来获取随机数

解释一下竞争同一个seed导致性能下降的原因，比如，看一下`Random`类的 `nextInt()` 方法实现：

```
1 public int nextInt() {  
2     return next(32)  
3 }
```

调用了`next(int bits)`方法，这是一个受保护的方法：

```
1 protected int next(int bits) {
```

```
1 protected int next(int bits) {
2     long oldseed, nextseed;
3     AtomicLong seed = this.seed
4 ;
5     do
6     {
7         oldseed = seed.get()
8 ;
9         nextseed = (oldseed * multiplier + addend) & mask;
10    } while (!seed.compareAndSet(oldseed, nextseed));
11    return (int)(nextseed >> (48 - bits))
12 ;
13 }
```

而这边的seed是一个全局变量：

```
1 /**
2  * The internal state associated with this pseudorandom number generator.
3  * (The specs for the methods in this class describe the ongoing
4  * computation of this value.)
5 */
6 private final AtomicLong seed;
```

多个线程同时获取随机数的时候，会竞争同一个seed，导致了效率的降低。

44、静态类、单例类、工厂类将它们的构造函数置为private

这是因为静态类、单例类、工厂类这种类本来我们就不需要外部将它们new出来，将构造函数置为private之后，保证了这些类不会产生实例对象。

-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↓



1. Java后端优质文章整理
2. Java 8 的这个特性,用起来真的很爽!
3. 面试挂在 Dubbo RPC ?
4. 在浏览器输入 URL 回车之后发生了什么?
5. 计算机专业的学生也太太太太太惨了吧?



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！

Java 效率工具之 Lombok

LiWenD正在掘金 Java后端 2019-12-27

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | LiWenD正在掘金

链接 | juejin.im/post/5b00517cf265da0ba0636d4b

还在编写无聊枯燥又难以维护的POJO吗? 洁癖者的春天在哪里? 请看Lombok!

在过往的Java项目中, 充斥着太多不友好的代码: POJO的getter/setter/toString; 异常处理; I/O流的关闭操作等等, 这些样板代码既没有技术含量, 又影响着代码的美观, Lombok应运而生。

首先说明一下: 任何技术的出现都是为了解决某一类问题的, 如果在此基础上再建立奇技淫巧, 不如回归Java本身。应该保持合理使用而不滥用。

Lombok的使用非常简单, 下面我们一起来看下:

1) 引入相应的maven包:

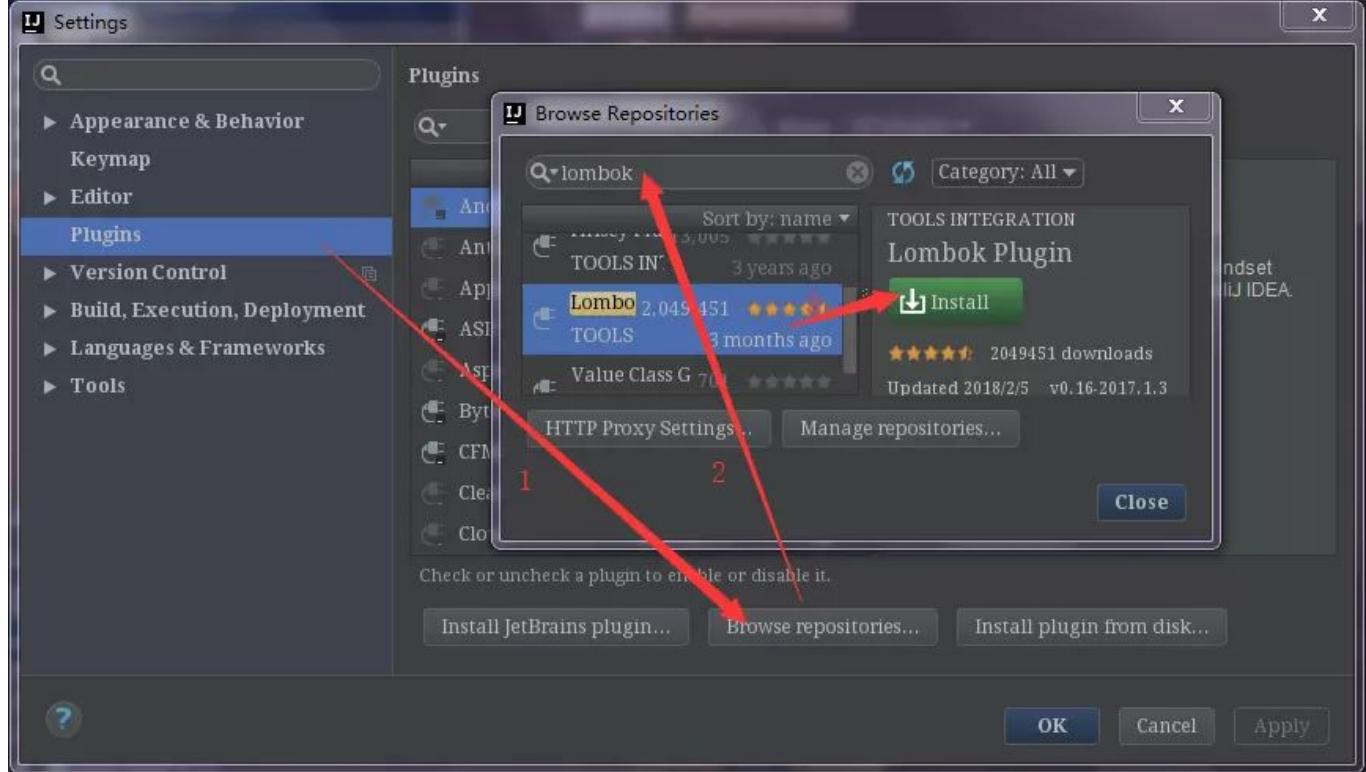
```
<dependency>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<version>1.16.18</version>
<scope>provided</scope>
</dependency>
```

Lombok的scope=provided, 说明它只在编译阶段生效, 不需要打入包中。事实正是如此, Lombok在编译期将带Lombok注解的Java文件正确编译为完整的Class文件。

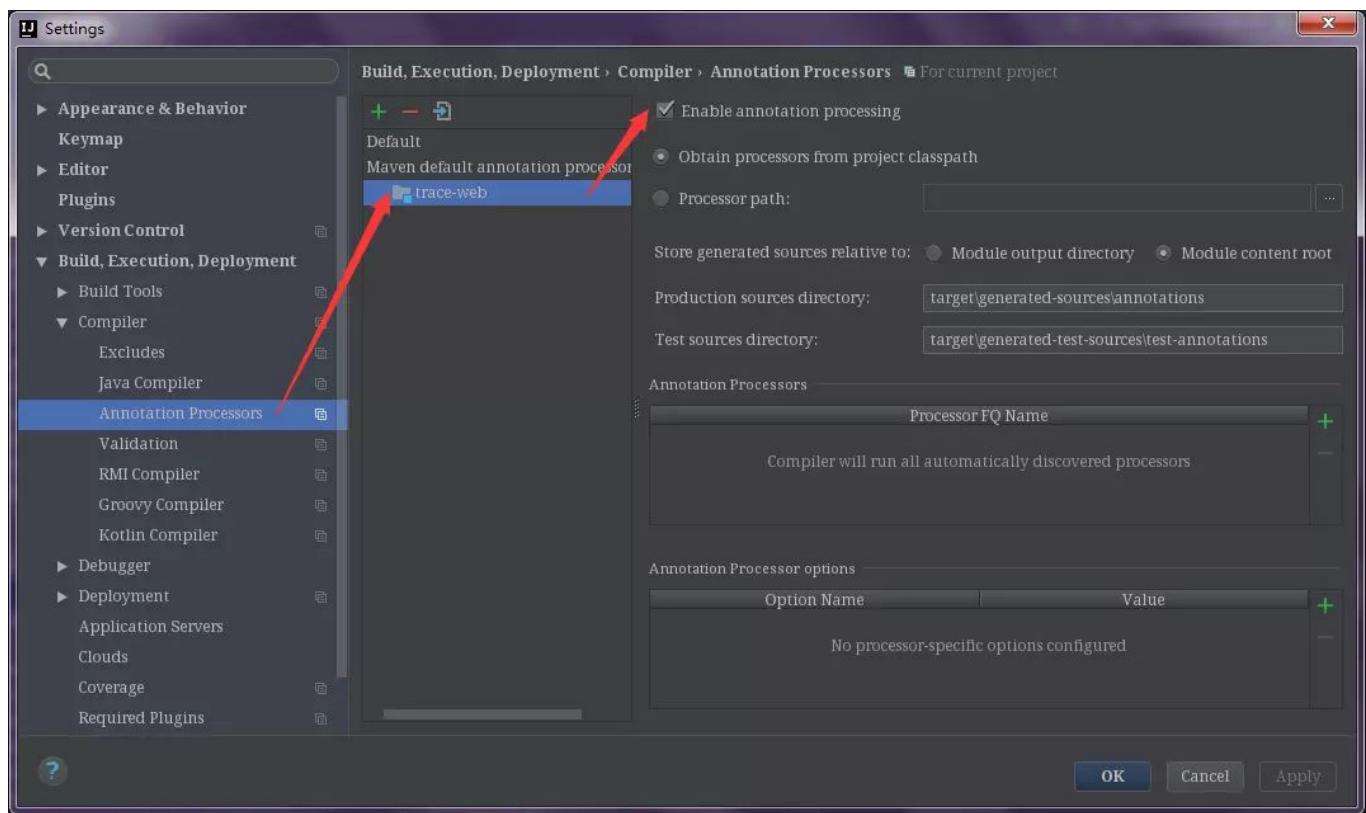
2) 添加IDE工具对Lombok的支持:

IDEA中引入Lombok支持如下:

- 点击File-- Settings设置界面, 安装Lombok插件:



- 点击File-- Settings设置界面，开启Annotation Processors：



开启该项是为了让Lombok注解在编译阶段起到作用。

Eclipse的Lombok插件安装可以自行百度，也比较简单，值得一提的是，由于Eclipse内置的编译器不是Oracle javac，而是eclipse自己实现的Eclipse Compiler for Java (ECJ)。要让ECJ支持Lombok，需要在eclipse.ini配置文件中添加如下两项内容：

-Xbootclasspath/a:[lombok.jar所在路径]

-javaagent:[lombok.jar所在路径]

3) Lombok实现原理：

自从Java 6起，javac就支持“JSR 269 Pluggable Annotation Processing API”规范，只要程序实现了该API，就能在javac运行的时候得到调用。

欢迎关注微信公众号：Java后端

Lombok就是一个实现了"JSR 269 API"的程序。在使用javac的过程中，它产生作用的具体流程如下：

- javac对源代码进行分析，生成一棵抽象语法树(AST)
- javac编译过程中调用实现了JSR 269的Lombok程序
- 此时Lombok就对第一步骤得到的AST进行处理，找到Lombok注解所在类对应的语法树(AST)，然后修改该语法树(AST)，增加Lombok注解定义的相应树节点
- javac使用修改后的抽象语法树(AST)生成字节码文件

4) Lombok注解的使用：

POJO类常用注解：

@Getter/@Setter: 作用类上，生成所有成员变量的getter/setter方法；作用于成员变量上，生成该成员变量的getter/setter方法。可以设定访问权限及是否懒加载等

```
package com.trace;

import lombok.AccessLevel;
import lombok.Getter;
import lombok.Setter;

/**
 * Created by Trace on 2018/5/19.<br/>
 * DESC: 测试类
 */
@SuppressWarnings("unused")
public class TestClass {

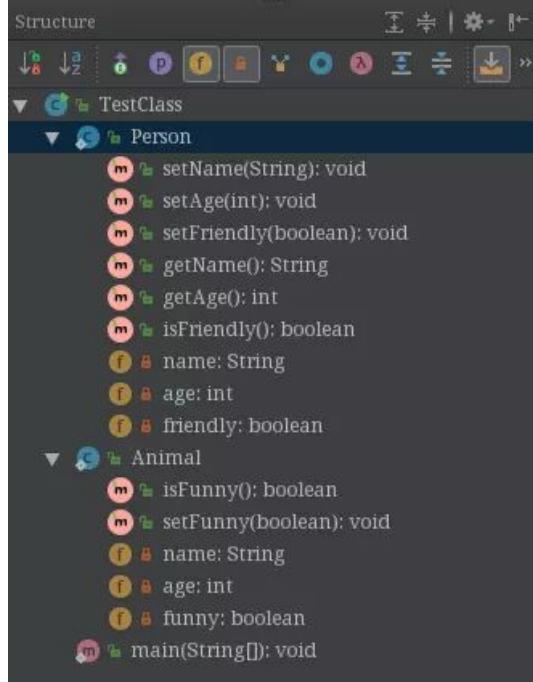
    public static void main(String[] args) {

    }

    @Getter(value = AccessLevel.PUBLIC)
    @Setter(value = AccessLevel.PUBLIC)
    public static class Person {
        private String name;
        private int age;
        private boolean friendly;
    }

    public static class Animal {
        private String name;
        private int age;
        @Getter @Setter private boolean funny;
    }
}
```

在Structure视图中，可以看到已经生成了getter/setter等方法：



编译后的代码如下：[这也是传统Java编程需要编写的样板代码]

```
//  
// Source code recreated from a .class file by IntelliJ IDEA  
//(powered by Fernflower decompiler)  
  
  
package com.trace;  
  
public class TestClass {  
    public TestClass() {  
    }  
  
    public static void main(String[] args) {  
    }  
  
    public static class Animal {  
        private String name;  
        private int age;  
        private boolean funny;  
  
        public Animal() {  
        }  
  
        public boolean isFunny() {  
            return this.funny;  
        }  
  
        public void setFunny(boolean funny) {  
            this.funny = funny;  
        }  
    }  
  
    public static class Person {  
        private String name;  
        private int age;  
        private boolean friendly;  
  
        public Person() {  
        }  
    }  
}
```

```

public String getName() {
    return this.name;
}

public int getAge() {
    return this.age;
}

public boolean isFriendly() {
    return this.friendly;
}

public void setName(String name) {
    this.name = name;
}

public void setAge(int age) {
    this.age = age;
}

public void setFriendly(boolean friendly) {
    this.friendly = friendly;
}
}
}
}

```

@ToString: 作用于类，覆盖默认的toString()方法，可以通过of属性限定显示某些字段，通过exclude属性排除某些字段。

```

@ToString(of = {"name", "age"}, exclude = {"age"})
@Getter(value = AccessLevel.PUBLIC)
@Setter(value = AccessLevel.PUBLIC)
public static class Person {
    private String name;
    private int age;
    private boolean friendly;
}

```

@EqualsAndHashCode: 作用于类，覆盖默认的equals和hashCode

@NonNull: 主要作用于成员变量和参数中，标识不能为空，否则抛出空指针异常。

```

@ToString(of = {"name", "age"}, exclude = {"age"})
@Getter(value = AccessLevel.PUBLIC)
@Setter(value = AccessLevel.PUBLIC)
public static class Person {
    @NonNull private String name;
    private int age;
    private boolean friendly;
}

```

@NoArgsConstructor, @RequiredArgsConstructor, @AllArgsConstructor: 作用于类上，用于生成构造函数。有staticName、access等属性。

staticName属性一旦设定，将采用静态方法的方式生成实例，access属性可以限定访问权限。

@NoArgsConstructor: 生成无参构造器；

@RequiredArgsConstructor: 生成包含final和@NonNull注解的成员变量的构造器；

@AllArgsConstructor：生成全参构造器。

```
@ToString(of = {"name", "age"}, exclude = {"age"})
@Getter(value = AccessLevel.PUBLIC)
@Setter(value = AccessLevel.PUBLIC)
@NoArgsConstructor(staticName = "of", access = AccessLevel.PRIVATE)
@RequiredArgsConstructor(access = AccessLevel.PACKAGE)
@AllArgsConstructor(access = AccessLevel.PUBLIC)
public static class Person {
    @NonNull private String name;
    private int age;
    private boolean friendly;
}
```

编译后结果：

```
public static class Person {
    @NonNull
    private String name;
    private int age;
    private boolean friendly;

    public String toString() {
        return "TestClass.Person(name=" + this.getName() + ", age=" + this.getAge() + ")";
    }

    @NonNull
    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }

    public boolean isFriendly() {
        return this.friendly;
    }

    public void setName(@NonNull String name) {
        if(name == null) {
            throw new NullPointerException("name");
        } else {
            this.name = name;
        }
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setFriendly(boolean friendly) {
        this.friendly = friendly;
    }

    private Person() {
    }

    private static TestClass.Person of() {
        return new TestClass.Person();
    }

    @ConstructorProperties({"name"})
}
```

```

Person(@NonNull String name) {
    if(name == null) {
        throw new NullPointerException("name");
    } else {
        this.name = name;
    }
}

@ConstructorProperties({"name", "age", "friendly"})
public Person(@NonNull String name, int age, boolean friendly) {
    if(name == null) {
        throw new NullPointerException("name");
    } else {
        this.name = name;
        this.age = age;
        this.friendly = friendly;
    }
}

```

@Data: 作用于类上, 是以下注解的集合:@ToString @EqualsAndHashCode @Getter @Setter
@RequiredArgsConstructor:

@Builder: 作用于类上, 将类转变为建造者模式

@Log: 作用于类上, 生成日志变量。针对不同的日志实现产品, 有不同的注解:

```

@CommonsLog
Creates
private static final org.apache.commons.logging.Log log = org.apache.commons.logging.LogFactory.getLog(LogExample.class);
@JBossLog
Creates
private static final org.jboss.logging.Logger log = org.jboss.logging.Logger.getLogger(LogExample.class);
@Log
Creates
private static final java.util.logging.Logger log = java.util.logging.Logger.getLogger(LogExample.class.getName());
@Log4j
Creates
private static final org.apache.log4j.Logger log = org.apache.log4j.Logger.getLogger(LogExample.class);
@Log4j2
Creates
private static final org.apache.logging.log4j.Logger log = org.apache.logging.log4j LogManager.getLogger(LogExample.class);
@Slf4j
Creates private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExample.class);
@XSlf4j
Creates
private static final org.slf4j.ext.XLogger log = org.slf4j.ext.XLoggerFactory.getXLogger(LogExample.class);

```

其他重要注解:

@Cleanup: 自动关闭资源, 针对实现了java.io.Closeable接口的对象有效, 如: 典型的IO流对象

```

public static void main(String[] args) throws Exception {
    File file = new File( s: "d:\\test.txt");
    @Cleanup InputStream inputStream = new FileInputStream(file);
    int len; byte[] bs = new byte[1024];
    while ((len = inputStream.read(bs)) != -1)
        System.out.println("content: " + new String(bs, 0, len));
}

```

编译后结果如下:

```
public static void main(String[] args) throws Exception {
    File file = new File("d:\\test.txt");
    FileInputStream inputStream = new FileInputStream(file);

    try {
        byte[] bs = new byte[1024];

        int len;
        while ((len = inputStream.read(bs)) != -1) {
            System.out.println("content: " + new String(bs, 0, len));
        }
    } finally {
        if (Collections.singletonList(inputStream).get(0) != null) {
            inputStream.close();
        }
    }
}
```

是不是简洁了太多。

@SneakyThrows: 可以对受检异常进行捕捉并抛出，可以改写上述的main方法如下：

```
@SneakyThrows
public static void main(String[] args) {
    File file = new File("d:\\test.txt");
    @Cleanup InputStream inputStream = new FileInputStream(file);
    int len; byte[] bs = new byte[1024];
    while ((len = inputStream.read(bs)) != -1)
        System.out.println("content: " + new String(bs, 0, len));
}
```

@Synchronized: 作用于方法级别，可以替换synchronize关键字或lock锁，用处不大。

- END -

推荐阅读

1. [60 个 Chrome 神器插件大收集](#)
2. [Spring 的 Bean 生命周期](#)
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！

Java 最常见的 208 道面试题

王磊的博客 Java后端 2019-09-18

来源 | 王磊的博客

博客 | www.cnblogs.com/vipstone

这份面试清单是我从 2015 年做 TeamLeader 之后开始收集的，一方面是给公司招聘用，另一方面是想用它来挖掘我在 Java 技术栈中的技术盲点，然后修复和完善它，以此来提高自己的技术水平。虽然我从 2009 年就开始参加编程工作了，但依旧觉得还有很多东西要学，当然学习的过程也给我带来了很多成就感，这些成就感也推动我学习更多的技术知识。

聊回面试题这件事，这份面试清单原本是我们公司内部使用的，可到后来有很多朋友在微信上联系到我，让我帮他们找一些面试方面的资料，而且这些关系也不太好拒绝，一呢，是因为这些找我，要面试题的人，不是我的好朋友的弟弟妹妹，就是我的弟弟妹妹们；二呢，我也不能马马虎虎的对付，受人之事忠人之命，我也不能辜负这份信任。慢慢的我产生了一个想法，要不要把我整理的这 200 多道面试题分享出来，来帮助更多需要的人。[微信搜索 web_resource 关注后获取每日一道面试题推送](#)。

说实话刚开始的时候还是比较犹豫的，首先我会觉得这么做会不会有点帮人“作弊”的嫌疑，最后我想通了，这是一件值得去做的事情。

- 第一：让更多的人因此而学到了更多的知识，这是一件大好事。
- 第二：这只是经验的高度提炼，让那些原本就掌握了技术却不知道怎么表达的人，学会如何在面试中展示自己。
- 第三：如果只是死记硬背这些面试题，只要面试官再深入问纠一下，也可对这个人有一个准确的认识，之前说的“帮人作弊”的事就存在了。
- 第四：学习有很多种方式，但只有好学者才会临池学书。如果是不想学的人，提供再多再好的资料放在他们的面前，他们也会视而不见。

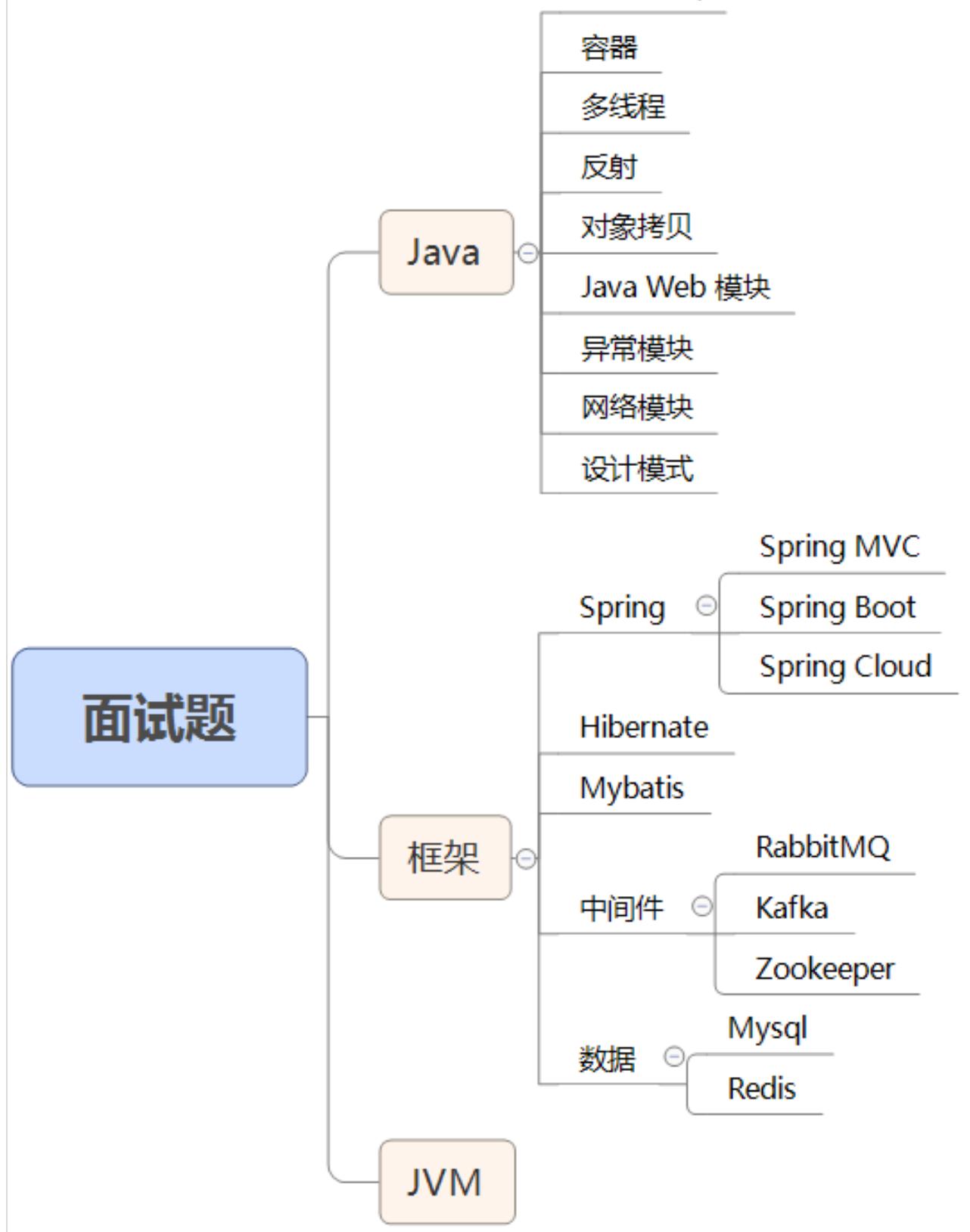
就像之前听过的一个故事，为什么在美国有些企业只要看你是哈佛的学历就直接录取？并不是哈佛有多么厉害，当然教学质量也是其中原因之一，但更多的是在美国上大学还是挺贵的，首先你能上的起哈佛，说明你的家庭条件还不错，从小应该就有很多参加更好教育的机会；第二，你能进入哈佛，也说明你脑子不笨，能考的上哈佛；最后才是哈佛确实能给你提供一个，相对不错的教育环境。综合以上特质，所以这些企业才敢直接聘请那些有哈佛学历的人。[微信搜索 web_resource 关注后获取每日一道面试题推送](#)。

对应到我们这份面试题其实也一样，首先你如果能记住其中大部分的答案说明你，第一，你很聪明并且记性还很好；第二，说明你有上进心，也愿意学习；第三，有了这份面试题做理论支撑之后，即使你的实践经验没有那么多，但懂得原理的你，做出来的程序也一定不会太差。

所以如果您是面试官，恰好又看到这里，如果条件允许的话，请多给这样愿意学又很聪明的年轻人多一些机会。

面试题模块介绍

说了这么多，下面进入我们本文的主题，我们这份面试题，包含的内容了十九个模块：Java 基础、容器、多线程、反射、对象拷贝、Java Web 模块、异常、网络、设计模式、Spring/Spring MVC、Spring Boot/Spring Cloud、Hibernate、Mybatis、RabbitMQ、Kafka、Zookeeper、MySQL、Redis、JVM。如下图所示：



可能对于初学者不需要看后面的框架和 JVM 模块的知识，读者朋友们可根据自己的情况，选择对应的模块进行阅读。微信搜索 `web_resource` 关注后获取每日一道面试题推送。

适宜阅读人群

- 需要面试的初/中/高级 java 程序员
- 想要查漏补缺的人
- 想要不断完善和扩充自己 java 技术栈的人

具体面试题

下面一起来看 208 道面试题，具体的内容。

一、Java 基础

- 1.JDK 和 JRE 有什么区别？
 - 2.== 和 equals 的区别是什么？
 - 3.两个对象的 hashCode() 相同，则 equals() 也一定为 true，对吗？
 - 4.final 在 java 中有什么作用？
 - 5.java 中的 Math.round(-1.5) 等于多少？
 - 6.String 属于基础的数据类型吗？
 - 7.java 中操作字符串都有哪些类？它们之间有什么区别？
 - 8.String str="i" 与 String str=new String("i") 一样吗？
 - 9.如何将字符串反转？
 - 10.String 类的常用方法都有那些？
 - 11.抽象类必须要有抽象方法吗？
 - 12.普通类和抽象类有哪些区别？
 - 13.抽象类能使用 final 修饰吗？
 - 14.接口和抽象类有什么区别？
 - 15.java 中 IO 流分为几种？
 - 16.BIO、NIO、AIO 有什么区别？
 - 17.Files 的常用方法都有哪些？
- ### 二、容器
- 18.java 容器都有哪些？
 - 19.Collection 和 Collections 有什么区别？
 - 20.List、Set、Map 之间的区别是什么？
 - 21.HashMap 和 Hashtable 有什么区别？
 - 22.如何决定使用 HashMap 还是 TreeMap？
 - 23.说一下 HashMap 的实现原理？
 - 24.说一下 HashSet 的实现原理？

25.ArrayList 和 LinkedList 的区别是什么?

26.如何实现数组和 List 之间的转换?

27.ArrayList 和 Vector 的区别是什么?

28.Array 和 ArrayList 有何区别?

29.在 Queue 中 poll() 和 remove() 有什么区别?

30.哪些集合类是线程安全的?

31.迭代器 Iterator 是什么?

32.Iterator 怎么使用?有什么特点?

33.Iterator 和 ListIterator 有什么区别?

34.怎么确保一个集合不能被修改?

三、多线程

35.并行和并发有什么区别?

36.线程和进程的区别?

37.守护线程是什么?

38.创建线程有哪几种方式?

39.说一下 runnable 和 callable 有什么区别?

40.线程有哪些状态?

41.sleep() 和 wait() 有什么区别?

42.notify() 和 notifyAll() 有什么区别?

43.线程的 run() 和 start() 有什么区别?

44.创建线程池有哪几种方式?

45.线程池都有哪些状态?

46.线程池中 submit() 和 execute() 方法有什么区别?

47.在 java 程序中怎么保证多线程的运行安全?

48.多线程锁的升级原理是什么?

49.什么是死锁?

50.怎么防止死锁?

51.ThreadLocal 是什么?有哪些使用场景?

52.说一下 synchronized 底层实现原理?

53.synchronized 和 volatile 的区别是什么?

54.synchronized 和 Lock 有什么区别?

55.synchronized 和 ReentrantLock 区别是什么?

56.说一下 atomic 的原理?

四、反射

57.什么是反射?

58.什么是 java 序列化?什么情况下需要序列化?

59.动态代理是什么?有哪些应用?

60.怎么实现动态代理?

五、对象拷贝

61.为什么要使用克隆?

62.如何实现对象克隆?

63.深拷贝和浅拷贝区别是什么?

六、Java Web

64.jsp 和 servlet 有什么区别?

65.jsp 有哪些内置对象?作用分别是什么?

66.说一下 jsp 的 4 种作用域?

67.session 和 cookie 有什么区别?

68.说一下 session 的工作原理?

69.如果客户端禁止 cookie 能实现 session 还能用吗?

70.spring mvc 和 struts 的区别是什么?

71.如何避免 sql 注入?

72.什么是 XSS 攻击,如何避免?

73.什么是 CSRF 攻击,如何避免?

七、异常

74.throw 和 throws 的区别?

75.final、finally、finalize 有什么区别?

76.try-catch-finally 中哪个部分可以省略?

77.try-catch-finally 中,如果 catch 中 return 了,finally 还会执行吗?

78.常见的异常类有哪些?

八、网络

79.http 响应码 301 和 302 代表的是什么?有什么区别?

80.forward 和 redirect 的区别?

81.简述 tcp 和 udp的区别?

82.tcp 为什么要三次握手,两次不行吗?为什么?

83.说一下 tcp 粘包是怎么产生的?

84.OSI 的七层模型都有哪些?

85.get 和 post 请求有哪些区别?

86.如何实现跨域?

87.说一下 JSONP 实现原理?

九、设计模式

88.说一下你熟悉的设计模式?

89.简单工厂和抽象工厂有什么区别?

十、Spring/Spring MVC

90.为什么要使用 spring?

91.解释一下什么是 aop?

92.解释一下什么是 ioc?

93.spring 有哪些主要模块?

94.spring 常用的注入方式有哪些?

95.spring 中的 bean 是线程安全的吗?

96.spring 支持几种 bean 的作用域?

97.spring 自动装配 bean 有哪些方式?

98.spring 事务实现方式有哪些?

99.说一下 spring 的事务隔离?

100.说一下 spring mvc 运行流程?

101.spring mvc 有哪些组件?

102.@RequestMapping 的作用是什么?

103.@Autowired 的作用是什么?

十一、Spring Boot/Spring Cloud

104.什么是 spring boot?

105.为什么要用 spring boot?

106.spring boot 核心配置文件是什么?

107.spring boot 配置文件有哪几种类型?它们有什么区别?

108.spring boot 有哪些方式可以实现热部署?

109.jpa 和 hibernate 有什么区别?

110.什么是 spring cloud?

111.spring cloud 断路器的作用是什么?

112.spring cloud 的核心组件有哪些?

十二、Hibernate

113.为什么要使用 hibernate?

114.什么是 ORM 框架?

115.hibernate 中如何在控制台查看打印的 sql 语句?

116.hibernate 有几种查询方式?

117.hibernate 实体类可以被定义为 final 吗?

118.在 hibernate 中使用 Integer 和 int 做映射有什么区别?

119.hibernate 是如何工作的?

120.get()和 load()的区别?

121.说一下 hibernate 的缓存机制?

122.hibernate 对象有哪些状态?

123.在 hibernate 中 getCurrentSession 和 openSession 的区别是什么?

124.hibernate 实体类必须要有无参构造函数吗?为什么?

十三、Mybatis

125.mybatis 中 #{}和 \${}的区别是什么?

126.mybatis 有几种分页方式?

127.RowBounds 是一次性查询全部结果吗?为什么?

128.mybatis 逻辑分页和物理分页的区别是什么?

129.mybatis 是否支持延迟加载?延迟加载的原理是什么?

130.说一下 mybatis 的一级缓存和二级缓存?

131.mybatis 和 hibernate 的区别有哪些?

132.mybatis 有哪些执行器(Executor)?

133.mybatis 分页插件的实现原理是什么?

134.mybatis 如何编写一个自定义插件?

十四、RabbitMQ

135.rabbitmq 的使用场景有哪些?

136.rabbitmq 有哪些重要的角色?

137.rabbitmq 有哪些重要的组件?

138.rabbitmq 中 vhost 的作用是什么?

139.rabbitmq 的消息是怎么发送的?

140.rabbitmq 怎么保证消息的稳定性?

141.rabbitmq 怎么避免消息丢失?

142.要保证消息持久化成功的条件有哪些?

143.rabbitmq 持久化有什么缺点?

144.rabbitmq 有几种广播类型?

145.rabbitmq 怎么实现延迟消息队列?

146.rabbitmq 集群有什么用?

147.rabbitmq 节点的类型有哪些?

148.rabbitmq 集群搭建需要注意哪些问题?

149.rabbitmq 每个节点是其他节点的完整拷贝吗?为什么?

150.rabbitmq 集群中唯一一个磁盘节点崩溃了会发生什么情况?

151.rabbitmq 对集群节点停止顺序有要求吗?

十五、Kafka

152.kafka 可以脱离 zookeeper 单独使用吗?为什么?

153.kafka 有几种数据保留的策略?

154.kafka 同时设置了 7 天和 10G 清除数据,到第五天的时候消息达到了 10G,这个时候 kafka 将如何处理?

155.什么情况会导致 kafka 运行变慢?

156.使用 kafka 集群需要注意什么?

十六、Zookeeper

157.zookeeper 是什么?

158.zookeeper 都有哪些功能?

159.zookeeper 有几种部署模式?

160.zookeeper 怎么保证主从节点的状态同步?

161.集群中为什么要有主节点?

162.集群中有 3 台服务器,其中一个节点宕机,这个时候 zookeeper 还可以使用吗?

163.说一下 zookeeper 的通知机制?

十七、 MySql

164.数据库的三范式是什么?

165.一张自增表里面总共有 7 条数据,删除了最后 2 条数据,重启 mysql 数据库,又插入了一条数据,此时 id 是几?

166.如何获取当前数据库版本?

167.说一下 ACID 是什么?

168.char 和 varchar 的区别是什么?

169.float 和 double 的区别是什么?

170.mysql 的内连接、左连接、右连接有什么区别?

171.mysql 索引是怎么实现的?

172.怎么验证 mysql 的索引是否满足需求?

173.说一下数据库的事务隔离?

174.说一下 mysql 常用的引擎?

175.说一下 mysql 的行锁和表锁?

176.说一下乐观锁和悲观锁?

177.mysql 问题排查都有哪些手段?

178.如何做 mysql 的性能优化?

179. 微信搜索 *web_resource* 关注后获取每日一道面试题推送。

十八、Redis

179.redis 是什么?都有哪些使用场景?

180.redis 有哪些功能?

181.redis 和 memecache 有什么区别?

182.redis 为什么是单线程的?

183.什么是缓存穿透?怎么解决?

184.redis 支持的数据类型有哪些?

185.redis 支持的 java 客户端都有哪些?

186.jedis 和 redisson 有哪些区别?

187.怎么保证缓存和数据库数据的一致性?

188.redis 持久化有几种方式?

189.redis 怎么实现分布式锁?

190.redis 分布式锁有什么缺陷?

191.redis 如何做内存优化?

192.redis 淘汰策略有哪些?

193.redis 常见的性能问题有哪些?该如何解决?

194.说一下 jvm 的主要组成部分?及其作用?

195.说一下 jvm 运行时数据区?

196.说一下堆栈的区别?

197.队列和栈是什么?有什么区别?

198.什么是双亲委派模型?

199.说一下类加载的执行过程?

200.怎么判断对象是否可以被回收?

201.java 中都有哪些引用类型?

202.说一下 jvm 有哪些垃圾回收算法?

203.说一下 jvm 有哪些垃圾回收器?

204.详细介绍一下 CMS 垃圾回收器?

205.新生代垃圾回收器和老生代垃圾回收器都有哪些?有什么区别?

206.简述分代垃圾回收器是怎么工作的?

207.说一下 jvm 调优的工具?

208.常用的 jvm 调优的参数都有哪些?

面试答案

由于文章篇幅问题,后续逐步更新答案。可以微信搜索「web_resource」关注后关注每日推送即可。此订阅号有「每日一道面试题」栏目。会对面试题进行详细讲解,关注搜索关注。

- 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」即可进入无广告技术交流群。

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. 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！

Java 泛型背后是什么

Java后端 2019-12-28

点击上方 Java后端, 选择 **设为星标**

优质文章, 及时送达

链接 | www.jianshu.com/p/dd34211f2565

这一节主要讲的内容是java中泛型的应用，通过该篇让大家更好地理解泛型，以及面试中经常说的泛型类型擦除是什么概念，今天就带着这几个问题一起看下：

举一个简单的例子：

```
5
6 public class Generic {
7     public static void main(String[] args) {
8         List<String> listString = new ArrayList<String>();
9         listString.add("123");
10        listString.add("456");
11        listString.add(123);
12    }
13 }
14 }
```

add (java.lang.String) in List cannot be applied
to (int)

这里可以看出来在代码编写阶段就已经报错了，不能往string类型的集合中添加int类型的数据。

那可不可以往List集合中添加多个类型的数据呢，答案是可以的，其实我们可以把list集合当成普通的类也是没问题的，那么就有下面的代码：

```
List listString = new ArrayList();
listString.add("123");
listString.add("456");
listString.add(789);
for (int i = 0; i < listString.size(); i++) {
    System.out.println("listString====>" + listString.get(i));
}
```

从这里可以看出来，不定义泛型也是可以往集合中添加数据的，所以说泛型只是一种类型的规范，在代码编写阶段起一种限制。

下面我们通过例子来介绍泛型背后数据是什么类型

```

public class BaseBean<T> {
    T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}

```

上面定义了一个泛型的类，然后我们通过反射获取属性和getValue方法返回的数据类型：

```

BaseBean<String> stringBaseBean = new BaseBean<>();
stringBaseBean.setValue("中国");
try {
    //获取属性上的泛型类型
    Field value = stringBaseBean.getClass().getDeclaredField(s: "value");
    Class<?> type = value.getType();
    String name = type.getName();
    System.out.println("type:" + name);

    //获取方法上的泛型类型
    Method getValue = stringBaseBean.getClass().getDeclaredMethod(s: "getValue");
    Object invoke = getValue.invoke(stringBaseBean);
    String methodName = invoke.getClass().getName();
    System.out.println("methodName:" + methodName);
} catch (NoSuchFieldException e) {
    e.printStackTrace();
} catch (NoSuchMethodException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
} catch (InvocationTargetException e) {
    e.printStackTrace();
}

```



从日志上看到通过反射获取到的属性是Object类型的，在方法中返回的是String类型，因此我们可以思考在getValue方法里面实际是做了个强转的动作，将Object类型的value强转成String类型。是的，没错，因为泛型只是为了约束我们规范代码，而对于编译完之后的class交给虚拟机后，对于虚拟机它是没有泛型的说法的，所有的泛型在它看来都是Object类型，因此泛型擦除是对于虚拟机而言的。

下面我们再来看一种泛型结构：

```
public class BaseBean<T extends String> {
    T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

这里我将泛型加了个关键字extends，对于泛型写得多的伙伴们来说，extends是约束了泛型是向下继承的，最后我们通过反射获取value的类型是String类型的，因此这里也不难看出，加extends关键字其实最终目的是约束泛型是属于哪一类的。所以我们在编写代码的时候如果没有向下兼容类型，会警告错误的：

```
Resource N 18
19         // BaseBean<Long> stringBaseBean = new BaseBean<>();
20         stringBaseBean.setValue("中国");
Type parameter 'java.lang.Long' is not within its bound; should extend 'java.lang.String'
```

大家有没有想过为啥要用泛型呢，既然说了泛型其实对于jvm来说都是Object类型的，那咱们直接将类型定义成Object不就是得了，这种做法是可以，但是在拿到Object类型值之后，自己还得强转，因此泛型减少了代码的强转工作，而将这些工作交给了虚拟机。

比如下面是我们没有定义泛型的例子：

```
public class BaseBean {
    Object value;

    public Object getValue() {
        return value;
    }

    public void setValue(Object value) {
        this.value = value;
    }
}
```

势必要在getValue的时候代码有个强转的过程，因此在能用泛型的时候，尽量用泛型来写，而且我认为一个好的架构师，业务的抽取是离不开泛型的定义。

类上面的泛型

比如实际项目中，我们经常会遇到服务端返回的接口中都有errMsg、status等公共返回信息，而变动的数据结构是data信息，因此我们可以抽取公共的BaseBean：

```
public class BaseBean<T> {  
    public String errMsg;  
    public T data;  
    public int status;  
}
```

抽象类或接口上的泛型

```
public abstract class BaseAdapter<T> {  
    List<T> DATAS;  
}  
  
public interface Factory<T> {  
    T create();  
}  
  
public static <T> T getData() {  
    return null;  
}
```

多元泛型

```
public interface Base<K, V> {  
    void setKey(K k);  
  
    V getValue();  
}
```

泛型二级抽象类或接口

```
public interface BaseCommon<K extends Common1, V> extends Base<K, V> {  
}  
  
public abstract class BaseCommon<K extends Common1, V> implements Base<K, V> {  
}
```

抽象里面包含抽象

```

public interface Base<K, V> {

    void addNode(Map<K, V> map);

    Map<K, V> getNode(int index);
}

public abstract class BaseCommon<K, V> implements Base<K, V> {

    LinkedList<Map<K, V>> DATAS = new LinkedList<>();

    @Override
    public void addNode(Map<K, V> map) {
        DATAS.addLast(map);
    }

    @Override
    public Map<K, V> getNode(int index) {
        return DATAS.get(index);
    }
}

```

<?>通配符

<?>通配符和<T>区别是<?>在你不知道泛型类型的时候，可以用<?>通配符来定义，下面通过一个例子来看看<?>的用处：

```

public class BaseBean<T> {
    T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}

```

```

public class Common1 extends Common {
}

```

```

BaseBean<Common> commonBaseBean = new BaseBean<>();
BaseBean<Common1> common1BaseBean = commonBaseBean;
// BaseBean<?> comm^1BaseBean = commonBaseBean;

Incompatible types.
Required: BaseBean <com.single.generic.Common1>
Found:    BaseBean <com.single.generic.Common>

```

在定义的时候将Common的泛型指向Common1的泛型，可以看到直接提示有问题，这里可以想，虽然Common1是继承自Common的，但是并不代表BaseBean之间是等量的，在开篇也讲过，如果泛型传入的是什么类型，那么在BaseBean中的getValue返回的类型就是什么，因此可以想两个不同的泛型类肯定是不等价的，但是如果我这里写呢：

```

public static void main(String[] args) {
    BaseBean<Common> commonBaseBean = new BaseBean<>();

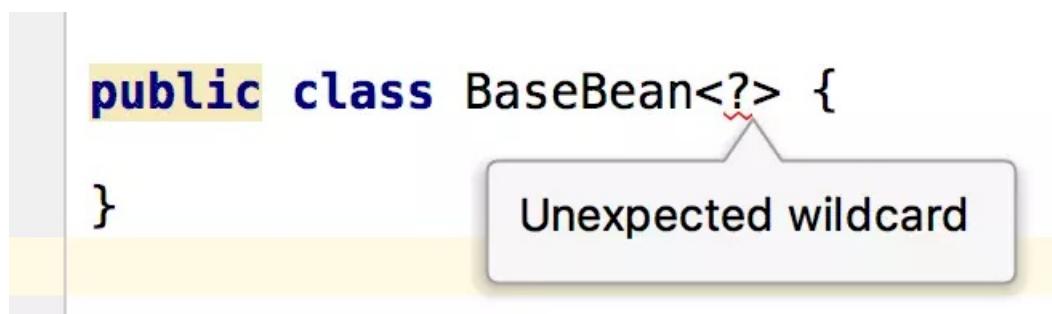
    BaseBean<?> common1BaseBean = commonBaseBean;
    try {

        Method setValue = common1BaseBean.getClass().getDeclaredMethod("setValue", Object.class);
        setValue.invoke(common1BaseBean, "123");
        Object value = common1BaseBean.getValue();
        System.out.println("result:" + value);
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}

```

在上面如果定义的泛型是通配符是可以等价的，因为此时的setValue的参数是Object类型，所以能直接将上面定义的泛型赋给通配符的BaseBean。

通配符不能定义在类上面、接口或方法上，只能作用在方法的参数上



其他的几种情况自己去尝试，正确的使用通配符：

```

public void setClass(Class<?> class){
}

```

<T extends>、<T super>、<? extends>、<? super>

<T extends **>表示上限泛型、<T super **>表示下限泛型

为了演示这两个通配符的作用，增加了一个类：

```

public class BaseBean<T extends Common> {
    T value;
}

```

```

public class Common extends BaseCommon{
}

```

```
21 BaseBean<Common1> common1BaseBean = new BaseBean<>();  
22 BaseBean<BaseCommon> baseCommonBaseBean = new BaseBean<>();  
23  
24
```

Type parameter 'com.single.generic.BaseCommon' is not within its bound; should extend 'com.single.generic.Common'

第二个定义的泛型是不合法的，因为BaseCommon是Common的父类，超出了Common的类型范围。

<T super>不能作用在类、接口、方法上，只能通过方法传参来定义泛型

在BaseBean里面定义了个方法：

```
public void add(Class<? super Common> clazz){  
}
```

```
BaseBean<Common1> common1BaseBean = new BaseBean<>();  
common1BaseBean.add(Common1.class);  
common1BaseBean.add(BaseCommon.class);
```

可以看到当传进去的是Common1.class的时候是不合法的，因为在add方法中需要传入Common父类的字节码对象，而Common1是继承自Common，所以直接不合法。

在实际开发中其实知道什么时候定义什么类型的泛型就ok，在mvp实际案例中泛型用得比较广泛，大家可以根据实际项目来找找泛型的感觉，只是面试的时候需要理解类型擦除是针对谁而言的。

类型擦除

其实在开篇的时候已经通过例子说明了，通过反射绕开泛型的定义，也说明了类中定义的泛型最终是以Object被jvm执行。所有的泛型在jvm中执行的时候，都是以Object对象存在的，加泛型只是为了一种代码的规范，避免了开发过程中再次强转。

泛型信息只存在于代码编译阶段，在进入 JVM 之前，与泛型相关的信息会被擦除掉，专业术语叫做类型擦除。

[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！

Java 线程有哪些不太为人所知的技巧与用法?

花名有孚 Java后端 2019-12-28

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

链接 | www.it.deepinmind.com

萝卜白菜各有所爱。像我就喜欢Java。学无止境, 这也是我喜欢它的一个原因。日常工作生活中你所用到的工具, 通常都有些你从来没有了解过的东西, 比方说某个方法或者是一些有趣的用法。比如说线程。没错, 就是线程。或者确切说是Thread这个类。当我们在构建高可扩展性系统的时候, 通常会面临各种各样的并发编程的问题, 不过我们现在所要讲的可能会略有不同。

从本文中你将会看到线程提供的一些不太常用的方法及技术。不管你是初学者还是高级用户或者是Java专家, 希望都能看一下哪些是你已经知道的, 而哪些是刚了解的。如果你认为关于线程还有什么值得分享给大家的, 希望能在下面积极回复。那我们就先开始吧。

初学者

1.线程名

程序中的每个线程都有一个名字, 创建线程的时候会给它分配一个简单的Java字符串来作为线程名。默认的名字是” Thread-0”, “Thread-1”, “Thread-2”等等。现在有趣的事情来了——Thread提供了两种方式来设置线程名:

线程构造函数, 下面是最简单的一个实现:

```
class SuchThread extends Thread {  
    Public void run() {  
        System.out.println ("Hi Mom! " + getName());  
    }  
}  
SuchThread wow =new SuchThread("much-name");
```

线程名setter方法:

```
wow.setName( "Just another thread name" );
```

没错, 线程名是可变的。因此我们可以在运行时修改它的名字, 而不用在初始化的时候就指定好。name字段其实就是一个简单的字符串对象。也就是说它能达到 $2^{31}-1$ 个字符那么长 (Integer.MAX_VALUE) 。这足够用了。注意这个名字并不是一个唯一性的标识, 因此不同的线程也可以拥有同样的线程名。还有一点就是, 不要把null用作线程名, 否则会抛出异常 (当然了, ” null” 还是可以的) 。

使用线程名来调试问题

既然可以设置线程名, 那么如果遵循一定的命名规则的话, 出了问题的时候排查起来就能更容易一些。 “Thread-6”这样的名字看起来就太没心没肺了, 肯定有比它更好的名字。在处理用户请求的时候, 可以将事务ID追加到线程名后面, 这样能显著减少你

排查问题的时间。

```
"pool-1-thread-1" #17 prio=5 os_prio=31 tid=0x00007f9d620c9800
```

```
nid=0x6d03 in Object.wait() [0x000000013ebcc000]
```

“pool-1-thread-1”，这也太严肃了吧。我们来看下这是什么情况，给它起一个好点的名字：

```
Thread.currentThread().setName(Context + TID + Params + current Time, ...);
```

现在我们再来运行下jstack，情况便豁然开朗了：

```
" Queue Processing Thread, MessageID: AB5CAD, type:
```

```
AnalyzeGraph, queue: ACTIVE_PROD, Transaction_ID: 5678956,
```

```
Start Time: 30/12/2014 17:37" #17 prio=5 os_prio=31 tid=0x00007f9d620c9800
```

```
nid=0x6d03 in Object.wait() [0x000000013ebcc000]
```

如果我们能知道线程在做什么，这样当它出问题的时候，至少可以拿到事务ID来开始排查。你可以回溯这个问题，复现它，然后定位问题并搞定它。如果你想知道jstack有什么给力的用法，可以看下这篇文章。

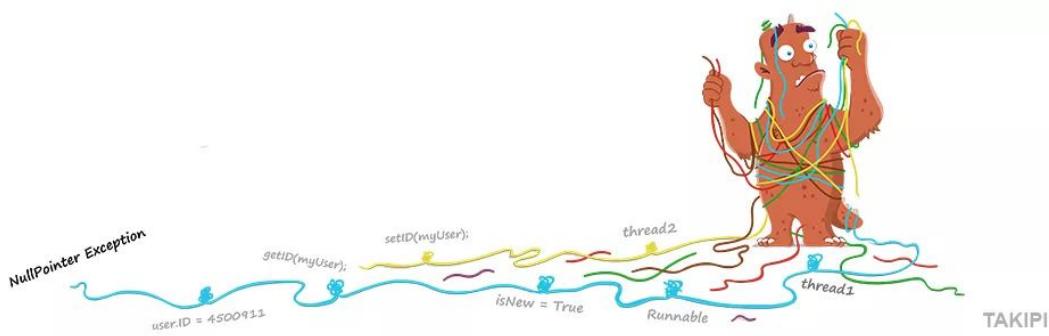
2. 线程优先级

线程还有一个有意思的属性就是它的优先级。线程的优先级介于1 (MINPRIORITY)到10 (MAXPRIORITY)之间，主线程默认是5 (NORM_PRIORITY)。每个新线程都默认继承父线程的优先级，因此如果你没有设置过的话，所有线程的优先级都是5。这个是通常被忽视的属性，我们可以通过getPriority()与setPriority()方法来获取及修改它的值。线程的构造函数里是没有这个功能的。

什么地方会用到优先级？

当然并不是所有的线程都是平等的，有的线程需要立即引起CPU的重视，而有些线程则只是后台任务而已。优先级就是用来把这些告诉给操作系统的线程调度器的。在Takipi中，这是我们开发的一错误跟踪及排查的工具，负责处理用户异常的线程的优先级是MAX_PRIORITY，而那些只是在上报新的部署情况的线程，它们的优先级就要低一些。你可能会觉得优先级高的线程从JVM的线程调度器那得到的时间会多一些。但其实并都是这样的。

在操作系统层面，每一个新线程都会对应一个本地线程，你所设置的Java线程的优先级会被转化成本地线程的优先级，这个在各个平台上是不一样的。在Linux上，你可以打开 “-XX:+UseThreadPriorities” 选项来启用这项功能。正如前面所说的，线程优先级只是你所提供的一个建议。和Linux本地的优先级相比，Java线程的优先级并不能覆盖全所有的级别（Linux共有1到99个优先级，线程的优先级在是-20到20之间）。最大的好处就是你所设定的优先级能在每个线程获得的CPU时间上有所体现，不过完全依赖于线程优先级的做法是不推荐的。



进阶篇

3.线程本地存储

这个和前面提到的两个略有不同。ThreadLocal是在Thread类之外实现的一个功能（java.lang.ThreadLocal），但它会为每个线程分别存储一份唯一的数据。正如它的名字所说的，它为线程提供了本地存储，也就是说你所创建出来变量对每个线程实例来说都是唯一的。和线程名，线程优先级类似，你可以自定义出一些属性，就好像它们是存储在Thread线程内部一样，是不是觉得酷？不过先别高兴得太早了，有几句丑话得先说在前头。

创建ThreadLocal有两种推荐方式：要么是静态变量，要么是单例实例中的属性，这样可以是非静态的。注意，它的作用域是全局的，只不过对访问它的线程而言好像是本地的而已。在下面这个例子中，ThreadLocal里面存储了一个数据结构，这样我们可以很容易地访问到它：

```
public static class CriticalData
{
    public int transactionId;
    public int username;
}

public static final ThreadLocal<CriticalData> globalData =new ThreadLocal<CriticalData>();
```

一旦获取到了ThreadLocal对象，就可以通过globalData.set()和globalData.get()方法来对它进行操作了。

全局变量？这不是什么好事

也尽然。ThreadLocal可以用来存储事务ID。如果代码中出现未捕获异常的时候它就相当有用了。最佳实践是设置一个UncaughtExceptionHandler，这个是Thread类本身就支持的，但是你得自己去实现一下这个接口。一旦执行到了UncaughtExceptionHandler里，就几乎没有任何线索能够知道到底发生了什么事情了。这会儿你能获取到的就只有Thread对象，之前导致异常发生的所有变量都无法再访问了，因为那些栈帧都已经被弹出了。一旦到了UncaughtExceptionHandler里，这个线程就只剩下最后一口气了，唯一能抓住的最后一根稻草就是ThreadLocal。

我们来试下这么做：

```
System.err.println("Transaction ID " + globalData.get().transactionId);
```

我们可以将一些与错误相关的有价值的上下文信息给存储到里面添。ThreadLocal还有一个更有创意的用法，就是用它来分配一

块特定的内存，这样工作线程可以把它当作缓存来不停地使用。当然了，这有没有用得看你在CPU和内存之间是怎么权衡的了。没错，`ThreadLocal`需要注意的就是会造成内存空间的浪费。只要线程还活着，那么它就会一直存在，除非你主动释放否则它是不会被回收的。因此如果使用它的话你最好注意一下，尽量保持简单。

4. 用户线程及守护线程

我们再回到`Thread`类。程序中的每个线程都会有一个状态，要么是用户状态，要么是守护状态。换句话说，要么是前台线程要么是后台线程。主线程默认是用户线程，每个新线程都会从创建它的线程中继承线程状态。因此如果你把一个线程设置成守护线程，那么它所创建的所有线程都会被标记成守护线程。如果程序中的所有线程都是守护线程的话，那么这个进程便会终止。我们可以通过`Boolean .setDaemon(true)`和`.isDaemon()`方法来查看及设置线程状态。

什么时候会用到守护线程？

如果进程不必等到某个线程结束才能终止，那么这个线程就可以设置成守护线程。这省掉了正常关闭线程的那些麻烦事，可以立即将线程结束掉。换个角度来说，如果一个正在执行某个操作的线程必须要正确地关闭掉否则就会出现不好的后果的话，那么这个线程就应该是用户线程。通常都是些关键的事务，比方说，数据库录入或者更新，这些操作都是不能中断的。

专家级

5. 处理器亲和性（Processor Affinity）

这里要讲的会更靠近硬件，也就是说，当软件遇上了硬件。处理器亲和性使得你能够将线程或者进程绑定到特定的CPU核上。这意味着只要是某个特定的线程，它就肯定只会在某个特定的CPU核上执行。通常来讲如何绑定是由操作系统的线程调度器根据它自己的逻辑来决定的，它很可能会将我们前面提到的线程优先级也一并考虑进来。

这么做的好处在于CPU缓存。如果某个线程只会在某个核上运行，那么它的数据恰好在缓存里的概率就大大提高了。如果数据正好就在CPU缓存里，那么就没有必要重新再从内存里加载了。你所节省的这几毫秒时间就能用在刀刃上，在这段时间里代码可以马上开始执行，也就能更好地利用所分配给它的CPU时间。当然了，操作系统层面可能会存在某种优化，硬件架构当然也是个很重要的因素，但利用了处理器的亲和性至少能够减小线程切换CPU的机率。

由于这里掺杂着多种因素，处理器亲和性到底对吞吐量有多大的影响，最好还是通过测试的方式来进行证明。也许这个方法并不是总能显著地提升性能，但至少有一个好处就是吞吐量会相对稳定。亲和策略可以细化到非常细的粒度上，这取决于你具体想要什么。高频交易行业便是这一策略最能大显身手的场景之一。

处理器亲和性的测试

Java对处理器的亲和性并没有原生的支持，当然了，故事也还没有就此结束。在Linux上，我们可以通过`taskset`命令来设置进程的亲和性。假设我们现在有一个Java进程在运行，而我们希望将它绑定到某个特定的CPU上：

```
taskset -c 1 "java AboutToBePinned"
```

如果是一个已经在运行了的进程：

```
taskset -c 1 <PID>
```

要想深入到线程级别还得再加些代码才行。所幸的是，有一个开源库能完成这样的功能：Java-Thread-Affinity。这个库是由OpenHFT的Peter Lawrey开发的，实现这一功能最简单直接的方式应该就是使用这个库了。我们通过一个例子来快速看下如何绑定某个线程，关于该库的更多细节请参考它在Github上的文档：

```
AffinityLock al = AffinityLock.acquireLock();
```

这样就可以了。关于获取锁的一些更高级的选项——比如说根据不同的策略来选择CPU——在Github上都有详细的说明。

结论

本文我们介绍了关于线程的5点知识：线程名，线程本地存储，优先级，守护线程以及处理器亲和性。希望这能为你日常工作中所用到的内容打开一扇新的窗户，期待你们的反馈！还有什么有关线程处理的方法可以分享给大家的吗，请不吝赐教。

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！

Java 线程池 8 大拒绝策略，面试必问！

Java后端 2019-10-12

点击上方 Java后端, 选择设为星标

技术博文, 及时送达

前言

谈到java的线程池最熟悉的莫过于ExecutorService接口了，jdk1.5新增的java.util.concurrent包下的这个api，大大的简化了多线程代码的开发。而不论你用FixedThreadPool还是CachedThreadPool其背后实现都是ThreadPoolExecutor。

ThreadPoolExecutor是一个典型的缓存池化设计的产物，因为池子有大小，当池子体积不够承载时，就涉及到拒绝策略。JDK中已经预设了4种线程池拒绝策略，下面结合场景详细聊聊这些策略的使用场景，以及我们还能扩展哪些拒绝策略。

池化设计思想

池化设计应该不是一个新名词。我们常见的如java线程池、jdbc连接池、redis连接池等就是这类设计的代表实现。

这种设计会初始预设资源，解决的问题就是抵消每次获取资源的消耗，如创建线程的开销，获取远程连接的开销等。就好比你去食堂打饭，打饭的大妈会先把饭盛好几份放那里，你来了就直接拿着饭盒加菜即可，不用再临时又盛饭又打菜，效率就高了。

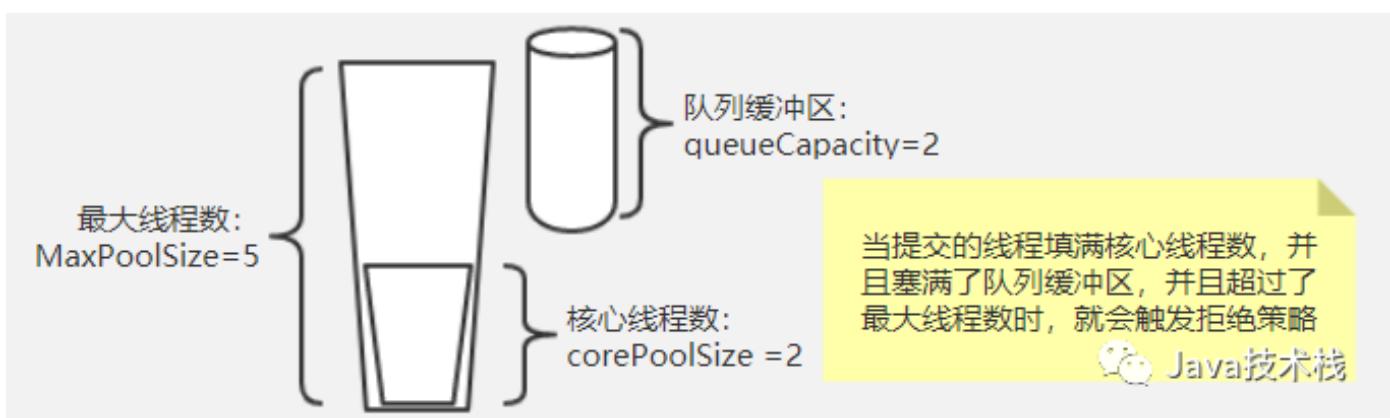
除了初始化资源，池化设计还包括如下这些特征：池子的初始值、池子的活跃值、池子的最大值等，这些特征可以直接映射到java线程池和数据库连接池的成员属性中。

线程池触发拒绝策略的时机

和数据源连接池不一样，线程池除了初始大小和池子最大值，还多了一个阻塞队列来缓冲。

数据源连接池一般请求的连接数超过连接池的最大值的时候就会触发拒绝策略，策略一般是阻塞等待设置的时间或者直接抛异常。

而线程池的触发时机如下图：



如图，想要了解线程池什么时候触发拒绝策略，需要明确上面三个参数的具体含义，是这三个参数总体协调的结果，而不是简单的

超过最大线程数就会触发线程拒绝策略，当提交的任务数大于corePoolSize时，会优先放到队列缓冲区，只有填满了缓冲区后，才会判断当前运行的任务是否大于maxPoolSize，小于时会新建线程处理。大于时就触发了拒绝策略，总结就是：当前提交任务数大于 (maxPoolSize + queueCapacity) 时就会触发线程池的拒绝策略了。

JDK内置4种线程池拒绝策略

拒绝策略接口定义

在分析JDK自带的线程池拒绝策略前，先看下JDK定义的 拒绝策略接口，如下：

```
1 public interface RejectedExecutionHandler {  
2     void rejectedExecution(Runnable r, ThreadPoolExecutor executor)  
3 }  
4 }
```

接口定义很明确，当触发拒绝策略时，线程池会调用你设置的具体的策略，将当前提交的任务以及线程池实例本身传递给你处理，具体作何处理，不同场景会有不同的考虑，下面看JDK为我们内置了哪些实现：

CallerRunsPolicy（调用者运行策略）

```
1 public static class CallerRunsPolicy implements RejectedExecutionHandler {  
2  
3     public CallerRunsPolicy() {  
4 }  
5  
6     public void rejectedExecution(Runnable r, ThreadPoolExecutor e)  
7 {  
8         if (!e.isShutdown()) {  
9             r.run();  
10        }  
11    }  
12 }
```

功能：当触发拒绝策略时，只要线程池没有关闭，就由提交任务的当前线程处理。

使用场景：一般在不允许失败的、对性能要求不高、并发量较小的场景下使用，因为线程池一般情况下不会关闭，也就是提交的任务一定会被运行，但是由于是调用者线程自己执行的，当多次提交任务时，就会阻塞后续任务执行，性能和效率自然就慢了。

AbortPolicy（中止策略）

```
1 public static class AbortPolicy implements RejectedExecutionHandler {  
2  
3     public AbortPolicy() {  
4 }  
5  
6     public void rejectedExecution(Runnable r, ThreadPoolExecutor e)  
7 {  
8         throw new RejectedExecutionException("Task " + r.toString() +  
9             ", on thread " + Thread.currentThread().getName());  
10    }  
11 }
```

```
8             " rejected from "
9
10            +
11            e.toString());
12        }
13    }
```

功能：当触发拒绝策略时，直接抛出拒绝执行的异常，中止策略的意思也就是打断当前执行流程

使用场景：这个就没有特殊的场景了，但是一点要正确处理抛出的异常。

ThreadPoolExecutor中默认的策略就是AbortPolicy，ExecutorService接口的系列ThreadPoolExecutor因为都没有显示的设置拒绝策略，所以默认的都是这个。但是请注意，ExecutorService中的线程池实例队列都是无界的，也就是说把内存撑爆了都不会触发拒绝策略。当自己自定义线程池实例时，使用这个策略一定要处理好触发策略时抛的异常，因为他会打断当前的执行流程。

DiscardPolicy（丢弃策略）

```
1 public static class DiscardPolicy implements RejectedExecutionHandler {
2
3     public DiscardPolicy() {
4 }
5
6     public void rejectedExecution(Runnable r, ThreadPoolExecutor e)
7 {
8     }
9 }
```

功能：直接静悄悄的丢弃这个任务，不触发任何动作

使用场景：如果你提交的任务无关紧要，你就可以使用它。因为它就是个空实现，会悄无声息的吞噬你的的任务。所以这个策略基本上不用了

DiscardOldestPolicy（弃老策略）

```
1 public static class DiscardOldestPolicy implements RejectedExecutionHandler {
2
3     public DiscardOldestPolicy() {
4 }
5
6     public void rejectedExecution(Runnable r, ThreadPoolExecutor e)
7 {
8         if (!e.isShutdown()) {
9             e.getQueue().poll();
10            e.execute(r);
11        }
12    }
13 }
```

功能：如果线程池未关闭，就弹出队列头部的元素，然后尝试执行

使用场景：这个策略还是会丢弃任务，丢弃时也是毫无声息，但是特点是丢弃的是老的未执行的任务，而且是待执行优先级较高的任务。基于这个特性，我能想到的场景就是，发布消息，和修改消息，当消息发布出去后，还未执行，此时更新的消息又来了，这个时候未执行的消息的版本比现在提交的消息版本要低就可以被丢弃了。因为队列中还有可能存在消息版本更低的消息会排队执行，所以在真正处理消息的时候一定要做好消息的版本比较。

第三方实现的拒绝策略

dubbo中的线程拒绝策略

```
1 public class AbortPolicyWithReport extends ThreadPoolExecutor.AbortPolicy {
2
3     protected static final Logger logger = LoggerFactory.getLogger(AbortPolicyWithReport.class);
4
5     private final String threadName;
6
7     private final URL url;
8
9     private static volatile long lastPrintTime = 0
10 ;
11
12     private static Semaphore guard = new Semaphore(1)
13 ;
14
15     public AbortPolicyWithReport(String threadName, URL url)
16 {
17         this.threadName = threadName;
18         this.url = url;
19     }
20
21     @Override
22     public void rejectedExecution(Runnable r, ThreadPoolExecutor e)
23 {
24         String msg = String.format("Thread pool is EXHAUSTED!"
25 +
26             " Thread Name: %s, Pool Size: %d (active: %d, core: %d, max: %d, largest:
27 +
28             " Executor status:(isShutdown:%s, isTerminated:%s, isTerminating:%s), in %
29 ,
30             threadName, e.getPoolSize(), e.getActiveCount(), e.getCorePoolSize(), e.getMaximum
31             e.getTaskCount(), e.getCompletedTaskCount(), e.isShutdown(), e.isTerminated(), e.i
32             url.getProtocol(), url.getIp(), url.getPort());
33         logger.warn(msg);
34         dumpJStack();
35         throw new RejectedExecutionException(msg);
36     }
37
38     private void dumpJStack()
39 {
40     //省略实
41     现
42 }
```

```
}
```

可以看到，当dubbo的工作线程触发了线程拒绝后，主要做了三个事情，原则就是尽量让使用者清楚触发线程拒绝策略的真实原因。

- 1) 输出了一条警告级别的日志，日志内容为线程池的详细设置参数，以及线程池当前的状态，还有当前拒绝任务的一些详细信息。可以说，这条日志，使用dubbo的有过生产运维经验的或多或少是见过的，这个日志简直就是日志打印的典范，其他的日志打印的典范还有spring。得益于这么详细的日志，可以很容易定位到问题所在
- 2) 输出当前线程堆栈详情，这个太有用了，当你通过上面的日志信息还不能定位问题时，案发现场的dump线程上下文信息就是你发现问题的救命稻草。
- 3) 继续抛出拒绝执行异常，使本次任务失败，这个继承了JDK默认拒绝策略的特性

Netty中的线程池拒绝策略

```
1 private static final class NewThreadRunsPolicy implements RejectedExecutionHandler {  
2     NewThreadRunsPolicy() {  
3         super()  
4     }  
5     }  
6  
7     public void rejectedExecution(Runnable r, ThreadPoolExecutor executor)  
8     {  
9         try  
10     {  
11         final Thread t = new Thread(r, "Temporary task executor")  
12     ;  
13         t.start();  
14     } catch (Throwable e) {  
15         throw new RejectedExecutionException(  
16             "Failed to start a new thread", e  
17         );  
18     }  
19     }  
20 }
```

Netty中的实现很像JDK中的CallerRunsPolicy，舍不得丢弃任务。不同的是，CallerRunsPolicy是直接在调用者线程执行的任务。而 Netty是新建了一个线程来处理的。

所以，Netty的实现相较于调用者执行策略的使用面就可以扩展到支持高效率高性能的场景了。但是也要注意一点，Netty的实现里，在创建线程时未做任何的判断约束，也就是说只要系统还有资源就会创建新的线程来处理，直到new不出新的线程了，才会抛创建线程失败的异常

activeMq中的线程池拒绝策略

```

1 new RejectedExecutionHandler() {
2     @Override
3     public void rejectedExecution(final Runnable r, final ThreadPoolExecutor executor)
4     {
5         try
6         {
7             executor.getQueue().offer(r, 60, TimeUnit.SECONDS);
8         } catch (InterruptedException e) {
9             throw new RejectedExecutionException("Interrupted waiting for BrokerService.worker")
10        ;
11    }
12    throw new RejectedExecutionException("Timed Out while attempting to enqueue Task.")
13;
14}
15);

```

activeMq中的策略属于最大努力执行任务型，当触发拒绝策略时，在尝试一分钟的时间重新将任务塞进任务队列，当一分钟超时还没成功时，就抛出异常

pinpoint中的线程池拒绝策略

```

1 public class RejectedExecutionHandlerChain implements RejectedExecutionHandler {
2     private final RejectedExecutionHandler[] handlerChain;
3
4     public static RejectedExecutionHandler build(List<RejectedExecutionHandler> chain)
5     {
6         Objects.requireNonNull(chain, "handlerChain must not be null")
7     ;
8         RejectedExecutionHandler[] handlerChain = chain.toArray(new RejectedExecutionHandler[0])
9     ;
10    return new RejectedExecutionHandlerChain(handlerChain);
11 }
12
13 private RejectedExecutionHandlerChain(RejectedExecutionHandler[] handlerChain)
14 {
15     this.handlerChain = Objects.requireNonNull(handlerChain, "handlerChain must not be null")
16 ;
17 }
18
19 @Override
20 public void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
{
21     for (RejectedExecutionHandler rejectedExecutionHandler : handlerChain) {
22         rejectedExecutionHandler.rejectedExecution(r, executor);
23     }
24 }

```

pinpoint的拒绝策略实现很有特点，和其他的实现都不同。他定义了一个拒绝策略链，包装了一个拒绝策略列表，当触发拒绝策略时，会将策略链中的rejectedExecution依次执行一遍。

结语

前文从线程池设计思想，以及线程池触发拒绝策略的时机引出java线程池拒绝策略接口的定义。并辅以JDK内置4种以及四个第三方开源软件的拒绝策略定义描述了线程池拒绝策略实现的各种思路和使用场景。

希望阅读此文后能让你对java线程池拒绝策略有更加深刻的认识，能够根据不同的使用场景更加灵活的应用。

转自：KL博客

地址：www.kailing.pub/article/index/arcid/255.html

编辑：Java技术栈 (id: javastack)

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！

Java 线程池的四种用法与使用场景

Java后端 2019-11-08

以下文章来源于一个程序员的成长，作者小涛

一个程序员的成长

“业精于勤荒于嬉，行成于思毁于随。”，一起来学习几斤技术啊！

一、如下方式存在的问题

```
new Thread() {
    @Override
    public void run() {
        // 业务逻辑
    }
}.start();
```

- 1、首先频繁的创建、销毁对象是一个很消耗性能的事情；
- 2、如果用户量比较大，导致占用过多的资源，可能会导致我们的服务由于资源不足而宕机；
- 3、综上所述，在实际的开发中，这种操作其实是不可取的一种方式。

二、使用线程池有什么优点

- 1、线程池中线程的使用率提升，减少对象的创建、销毁；
- 2、线程池可以控制线程数，有效的提升服务器的使用资源，避免由于资源不足而发生宕机等问题；

三、线程池的四种使用方式

1、newCachedThreadPool

创建一个线程池，如果线程池中的线程数量过大，它可以有效的回收多余的线程，如果线程数不足，那么它可以创建新的线程。

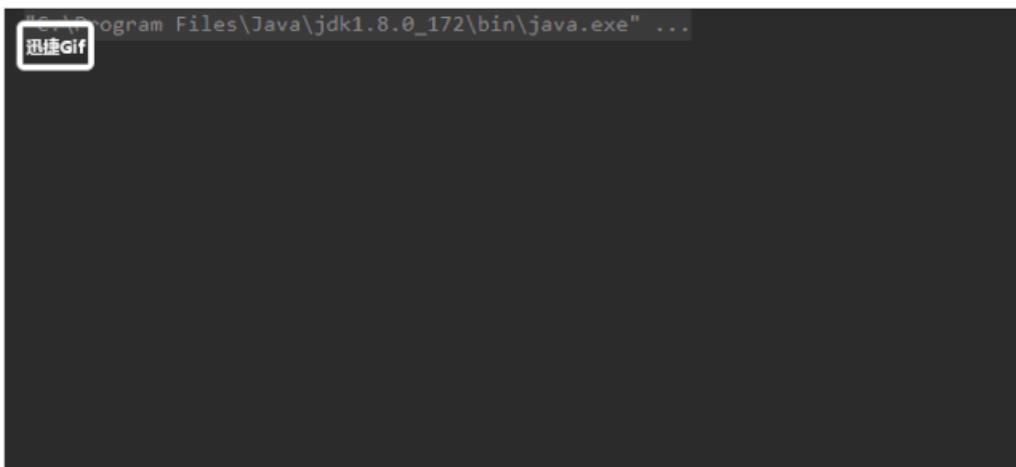
```
public static void method() throws Exception {
    ExecutorService executor = Executors.newCachedThreadPool();
    for (int i = 0; i < 5; i++) {
        final int index = i;
        Thread.sleep(1000);
        executor.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + " " + index);
            }
        });
    }
}
```

执行结果

```
"C:\Program Files\Java\jdk1.8.0_172\bin\java.exe" ...
pool-1-thread-1 0
pool-1-thread-1 1
pool-1-thread-1 2
pool-1-thread-1 3
pool-1-thread-1 4
```

通过分析我看可以看到，至始至终都由一个线程执行，实现了线程的复用，并没有创建多余的线程。

如果当我们的业务需要一定的时间进行处理，那么将会出现什么结果。我们来模拟一下。



可以明显的看出，现在就需要几条线程来交替执行。

不足：这种方式虽然可以根据业务场景自动的扩展线程数来处理我们的业务，但是最多需要多少个线程同时处理缺是我们无法控制的；

优点：如果当第二个任务开始，第一个任务已经执行结束，那么第二个任务会复用第一个任务创建的线程，并不会重新创建新的线程，提高了线程的复用率；

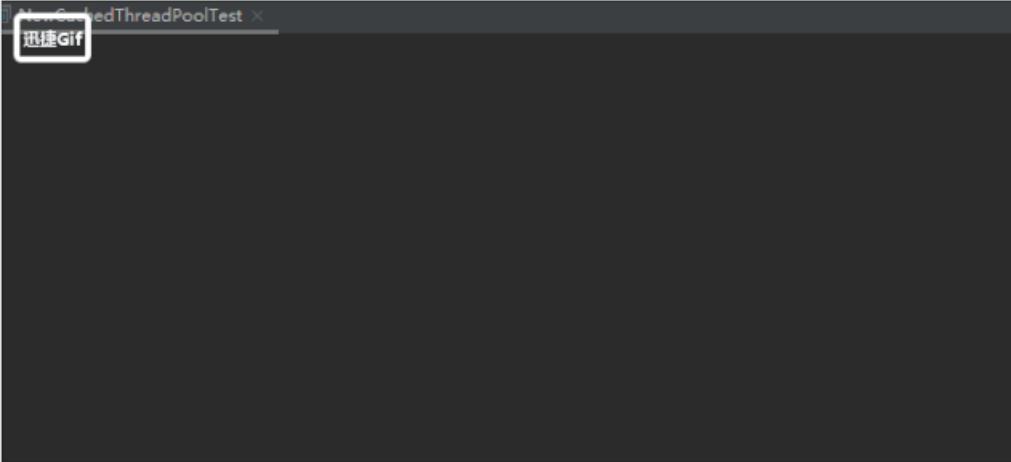
2、newFixedThreadPool

这种方式可以指定线程池中的线程数。举个栗子，如果一间澡堂子最大只能容纳20个人同时洗澡，那么后面来的人只能在外面排队等待。如果硬往里冲，那么只会出现一种情景，摩擦摩擦...

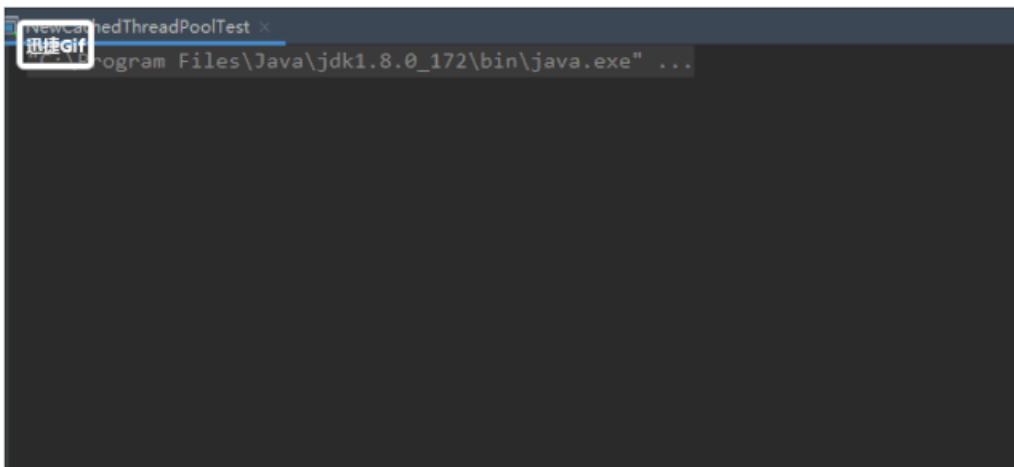
首先测试一下最大容量为一个线程，那么会不会是我们预测的结果。

```
public static void method_01() throws InterruptedException {
    ExecutorService executor = Executors.newFixedThreadPool(1);
    for (int i = 0; i < 10; i++) {
        Thread.sleep(1000);
        final int index = i;
        executor.execute(() -> {
            try {
                Thread.sleep(2 * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + " " + index);
        });
    }
    executor.shutdown();
}
```

执行结果



我们改为3条线程再来看下结果



优点：两个结果综合说明，`newFixedThreadPool`的线程数是可以进行控制的，因此我们可以通过控制最大线程来使我们的服务器打到最大的使用率，同事又可以保证及时流量突然增大也不会占用服务器过多的资源。

3、`newScheduledThreadPool`

该线程池支持定时，以及周期性的任务执行，我们可以延迟任务的执行时间，也可以设置一个周期性的时间让任务重复执行。该线程池中有以下两种延迟的方法。

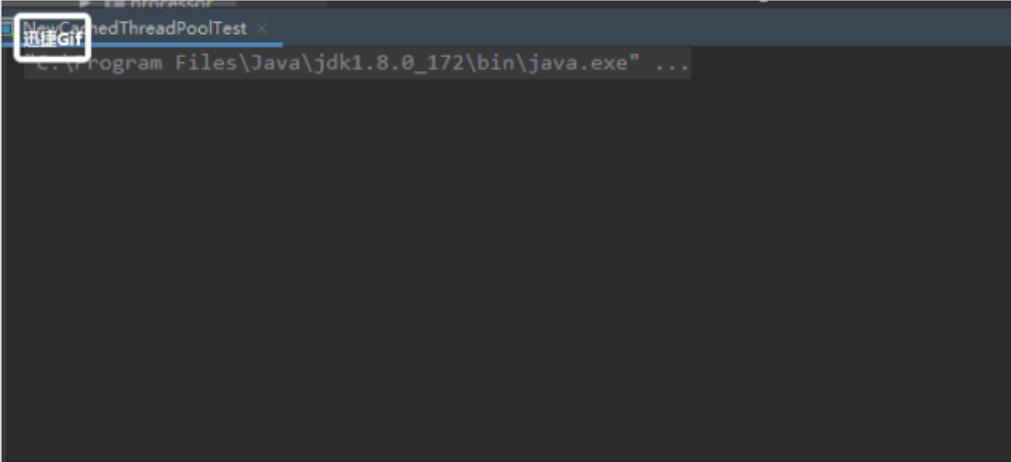
- `scheduleAtFixedRate`

测试一

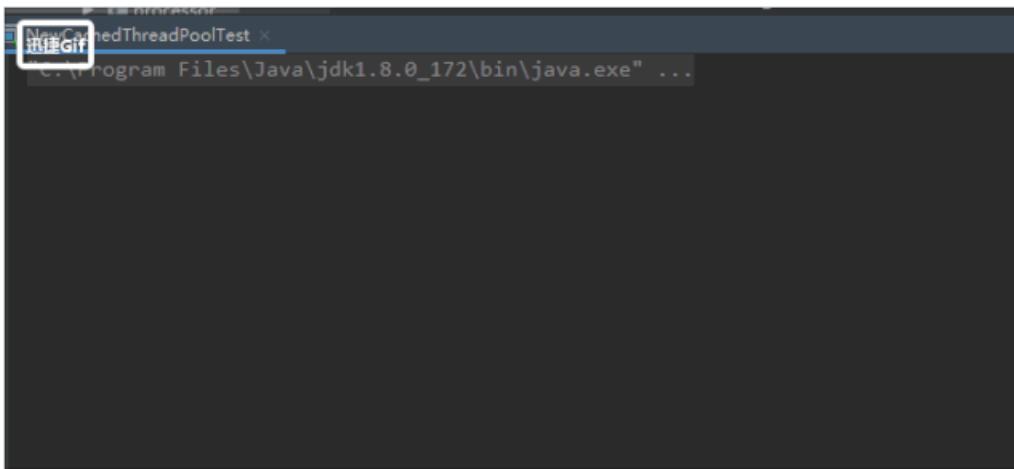
```
public static void method_02() {
    ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);

    executor.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            long start = new Date().getTime();
            System.out.println("scheduleAtFixedRate 开始执行时间：" +
                DateFormat.getTimeInstance().format(new Date()));
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            long end = new Date().getTime();
            System.out.println("scheduleAtFixedRate 执行花费时间=" + (end - start) / 1000 + "m");
            System.out.println("scheduleAtFixedRate 执行完成时间：" + DateFormat.getTimeInstance().format(new Date()));
        }
        System.out.println("=====");
    }, 1, 5, TimeUnit.SECONDS);
}
```

执行结果



测试二



总结：以上两种方式不同的地方是任务的执行时间，如果间隔时间大于任务的执行时间，任务不受执行时间的影响。如果间隔时间小于任务的执行时间，那么任务执行结束之后，会立马执行，至此间隔时间就会被打乱。

● scheduleWithFixedDelay

测试一

```
public static void method_03() {
    ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);

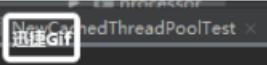
    executor.scheduleWithFixedDelay(new Runnable() {
        @Override
        public void run() {
            long start = new Date().getTime();
            System.out.println("scheduleWithFixedDelay 开始执行时间：" +
                DateFormat.getTimeInstance().format(new Date()));

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            long end = new Date().getTime();
            System.out.println("scheduleWithFixedDelay 执行花费时间=" + (end - start) / 1000 + "m");
            System.out.println("scheduleWithFixedDelay 执行完成时间：" +
                + DateFormat.getTimeInstance().format(new Date()));
            System.out.println("=====");

        }
    }, 1, 2, TimeUnit.SECONDS);
}
```

执行结果



测试二

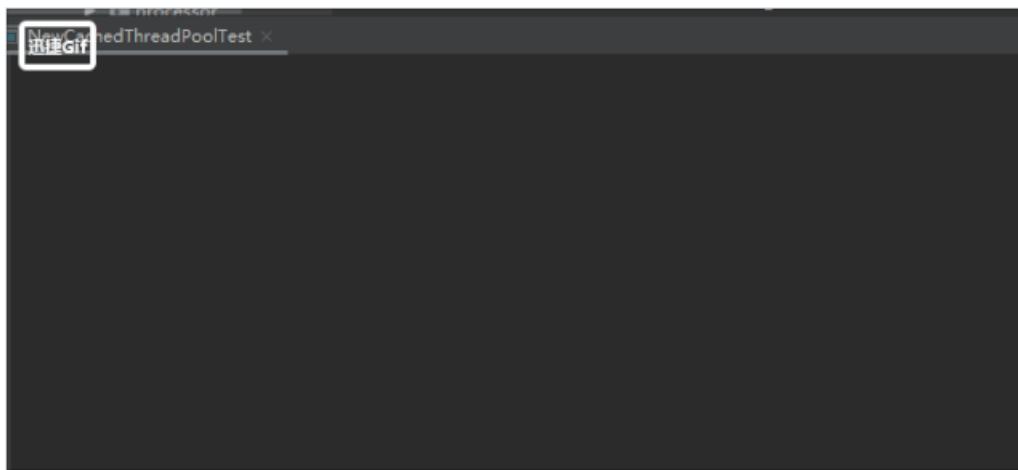
```
public static void method_03() {
    ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);

    executor.scheduleWithFixedDelay(new Runnable() {
        @Override
        public void run() {
            long start = new Date().getTime();
            System.out.println("scheduleWithFixedDelay 开始执行时间：" +
                DateFormat.getTimeInstance().format(new Date()));

            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            long end = new Date().getTime();
            System.out.println("scheduleWithFixedDelay 执行花费时间：" + (end - start) / 1000 + "m");
            System.out.println("scheduleWithFixedDelay 执行完成时间：" +
                DateFormat.getTimeInstance().format(new Date()));
            System.out.println("=====");
        }
    }, 1, 2, TimeUnit.SECONDS);
}
```

执行结果



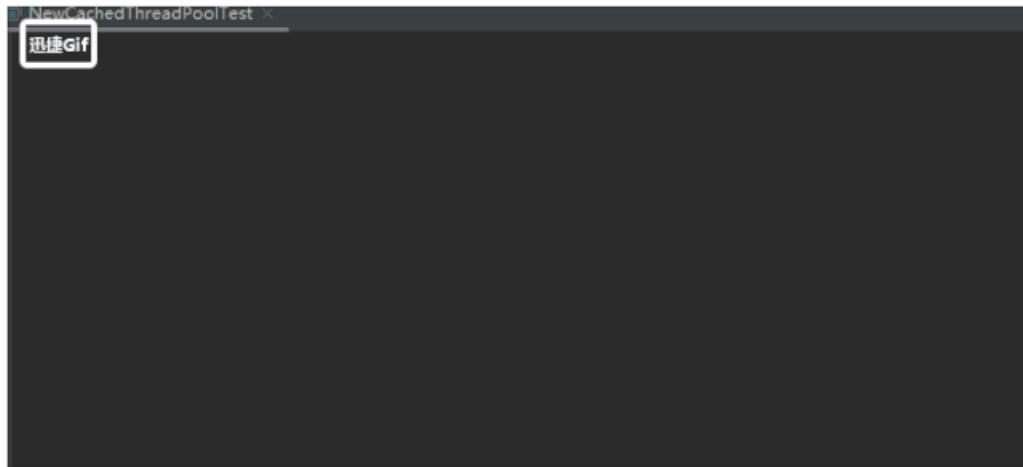
总结：同样的，跟scheduleWithFixedDelay测试方法一样，可以测出scheduleWithFixedDelay的间隔时间不会受任务执行时间长短的影响。

4、newSingleThreadExecutor

这是一个单线程池，至始至终都由一个线程来执行。

```
public static void method_04() {  
  
    ExecutorService executor = Executors.newSingleThreadExecutor();  
  
    for (int i = 0; i < 5; i++) {  
        final int index = i;  
        executor.execute(() -> {  
            try {  
                Thread.sleep(2 * 1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            System.out.println(Thread.currentThread().getName() + " " + index);  
        });  
    }  
    executor.shutdown();  
}
```

执行结果



四、线程池的作用

线程池的作用主要是为了提升系统的性能以及使用率。文章刚开始就提到，如果我们使用最简单的方式创建线程，如果用户量比较大，那么就会产生很多创建和销毁线程的动作，这会导致服务器在创建和销毁线程上消耗的性能可能要比处理实际业务花费的时间和性能更多。线程池就是为了解决这种问题而出现的。

同样思想的设计还有很多，比如**数据库连接池**，由于频繁的连接数据库，然而创建连接是一个很消耗性能的事情，所有数据库连接池就出现了。

-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 ↓



扫一扫上面的二维码图案，加我微信

推荐阅读

1. Java 代码是如何一步步输出结果的?
2. IntelliJ IDEA 详细图解最常用的配置
3. Maven 实战问题和最佳实践
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

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！

Java 编程中，有哪些好的习惯从一开始就值得坚持？

Java后端 2月18日



微信搜一搜

Java后端

来源 | zhihu.com/question/32255673/answer/532272606

1. 规范化自己的代码，少点个人风格，多点通用规矩，并学会使用CheckStyle工具。

其实任何东西我们都希望它能够“自动化”，随着编程经验的提升，大部分编程规范你已经了然于心，但是实际操作的时候，又总是忘这忘那，我们希望一个工具来帮我们自动检测我们的程序是否是符合规范，结构良好的。

事实上，任何语言都是有自己的编程规范的，编程规范的制定，十分有利于代码的阅读和潜在Bug风险的降低，比如在Java中，有严格的命名规范：

对于类 (Class)的命名，有这样的规范：

Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

类的名字必须是名词，每个单词的第一个字母需大写。尽可能让你的类名称简洁又能传递清楚含义。尽量使用单词全拼，避免同义词或缩写（除非缩写使用更广泛，比如URL, HTML等）。

比如在Java中，有严格的文档规范：

```
/**  
 * Returns an Image object that can then be painted on the screen.  
 * The url argument must specify an absolute {@link URL}. The name  
 * argument is a specifier that is relative to the url argument.  
 * <p>  
 * This method always returns immediately, whether or not the  
 * image exists. When this applet attempts to draw the image on  
 * the screen, the data will be loaded. The graphics primitives  
 * that draw the image will incrementally paint on the screen.  
 *  
 * @param url an absolute URL giving the base location of the image  
 * @param name the location of the image, relative to the url argument  
 * @return the image at the specified URL  
 * @see Image  
 */  
  
public Image getImage(URL url, String name) {  
    try {  
        return getImage(new URL(url, name));  
    } catch (MalformedURLException e) {  
        return null;  
    }  
}
```

文档是用HTML语言写成的，前半部分是关于当前方法或类的描述，后面需要有参数标签@param和返回标签@return，还可以加一些别的标签，比如@see，只有这样，当别人试图引用你的程序时，才能马上明白你的某段程序是用来干嘛的，参数传递，返回等一目了然，要知道，实际工作中，大量的协作就意味着代码需要高度的重用性，你必须把你的程序封装完美，并且让别人仅仅看你的文档，就知道你的这个API怎么用。

上面说的仅仅是编程规范的冰山一角了，问题是，你有时会忘掉或用错一些规范，即便你知道它。

所以我们需要使用checkstyle插件去自动检测我们的程序是否符合规范。

对于Java而言，详情请见：<http://checkstyle.sourceforge.net/>

Github地址：<https://github.com/checkstyle/checkstyle>

在各大Java IDE中，可以直接在Eclipse Marketplace中下载：

Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.

Search Recent Popular Favorites Installed Eclipse Newsletter: Boot & Build Eclipse...

Find: checkstyle All Markets All Categories Go

checkstyle

★ 15 Installs: 18.0K (394 last month) Install

Stereotype Checker

The Stereotype Checker is an extension to checkstyle checking if your code matches the defined architecture. More information can be found on the GitHub pages [more info](#)

by [NovaTec Consulting GmbH](#), Apache 2.0
[architecture](#) [checkstyle](#) [validation](#) [analyzer](#)

★ 5 Installs: 356 (0 last month) Install

Checkstyle Plug-in 8.12.0

The Checkstyle Plugin (eclipse-CS) integrates the well-known source code analyzer Checkstyle into the Eclipse IDE. Checkstyle is a development tool to help you...
[more info](#)

by [Lars Ködderitzsch](#), LGPL
[static analysis](#) [favorite](#) [validator](#) [coding style](#)

★ 713 Installs: 393K (5,934 last month) Installed

Marketplaces

?

< Back Install Now > Finish Cancel 知乎 @牛街

其他的语言应该也有自己的插件，可以自行谷歌了解。

- 宁可变量名长，也不要让变量名短得让人无法推测其含义。
- 在电脑里安装两套输入法，编程的时候，将中文输入法彻底关掉，确保任何快捷键都不会将其转换成中文输入法，防止中文类似符号引起混淆，比如：

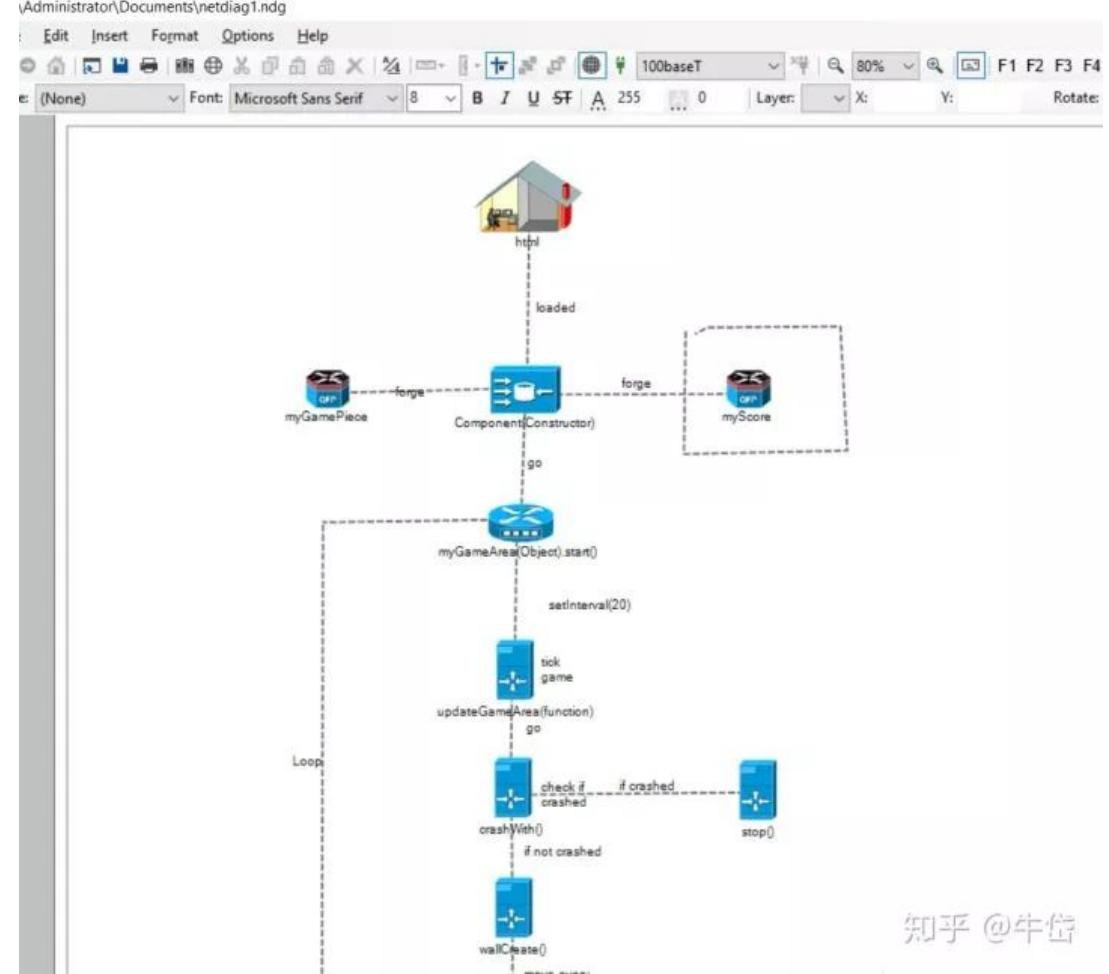
中文：()； English:();

一点点小错误，就有可能让你多花一两个小时在没有意义的事上，人生苦短，尽量避免低级错误。

- 尽可能杜绝重复代码，凡是需要用至少两次的代码，给它单独做一个类或函数。
- 避免类与类之间的内部调用(Cycle Reference)，其实也就是降低函数模块的耦合程度。类与类之间的调用只允许通过接口，保

证更改某个类的时候，其他的仍然能工作。

6. 多读别人的优秀代码，拿别人的优秀代码和自己的代码进行对比，学习别人的长处，吸收经验。
7. 尝试着做内容的生产者，尝试着写一些教程或笔记，分享给社区，不要只做社区内容的吸收者，还要不断地生产内容，回馈社区给你的帮助，比如在StackOverFlow上回答别人的问题等。
8. 既要脚踏实地，也要多看看社区发生了什么新闻，有什么新的技术和软件的发布，这些技术和软件将怎样影响你的开发工作，现在使用的IDE或Editor是否有更好的替代产品等等。
9. 没有任务的时候，也不要闲下来，去开发点你喜欢的东西，从中挑战自己，增长经验。
10. 不要过分依赖教程，要学会看官方文档。 凡是能被做成教程的东西，往往已经过时了，最新的技术，最新发布的标准，往往没有现成的教程，你需要去认真阅读官方文档，那里的东西才是最权威的。
11. 不要参与语言好坏的争论，人们往往偏向于喜欢自己用得熟练，用得多的那个语言，语言好坏之争，就和争谁的女朋友漂亮一样，我当然觉得自己的女朋友（虽然是null）最漂亮，但是别人并不这么觉得。
12. 当你有什么需求的时候，往往别人也有这个需求，而且往往也有了相应的工具去解决你这个需求，比如，你想将函数的调用关系可视化，弄成树状图那样，这样的工具已经有了，比如SourceInsight（付费），Source Navigator（免费）等。
13. 少在国内的XX软件园里下载各种破解软件，盗版软件等，这些软件园为了盈利，会在你安装的过程中，悄无声息地给你安装上一堆其他的流氓软件，360首当其冲，这些垃圾软件，删的越干净越好。
14. 你的开发电脑，CPU可以差些，但内存最好大些，推荐至少要8G，甚至推荐10G往上走，你常常需要同时打开一堆浏览器页面和一个IDE甚至还有别的一堆工具，如果你做过安卓开发，AndroidStudio动辄就调用你电脑2-3G的内存，一般的4G电脑肯定是吃不消的，严重降低开发体验，但也并不是让你换电脑，内存条了解一下。
15. 保持一个健康，干净的电脑状态，硬盘里的文件存储要有调理，容易寻找指定文件，降低文件丢失概率，加快文件寻找速度。
16. C盘快满了的话，可以通过Disk Manager将别的磁盘的空间送给C盘。
17. 用NetWork NotePad画网络图表示函数调用关系（当然你可以用别的来画），像这样：



知乎 @牛岱

这是前两天编一个FlappyBird时草草画的图，虽然简陋，但有用。

18. 可以考虑用一个电脑架子，防止乌龟颈，保护颈椎。



19. 下载一个护眼宝，保护视力。



-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. Dubbo 爆出严重漏洞!](#)
- [2. Spring Boot+Redis 分布式锁：模拟抢单](#)
- [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！

Java 非阻塞 IO 和异步 IO

HongJie Java后端 2019-11-09

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | HongJie

链接 | javadoop.com/post/nio-and-aio

本文将介绍**非阻塞 IO 和异步 IO**, 也就是大家耳熟能详的 NIO 和 AIO。很多初学者可能分不清楚异步和非阻塞的区别, 只是在各种场合能听到**异步非阻塞**这个词。

本文会先介绍并演示阻塞模式, 然后引入非阻塞模式来对阻塞模式进行优化, 最后再介绍 JDK7 引入的异步 IO, 由于网上关于异步 IO 的介绍相对较少, 所以这部分内容我会介绍得具体一些。

希望看完本文, 读者可以对非阻塞 IO 和异步 IO 的迷雾看得更清晰些, 或者为初学者解开一丝丝疑惑也是好的。

阻塞模式 IO

我们已经介绍过使用 Java NIO 包组成一个简单的**客户端-服务端**网络通讯所需要的 ServerSocketChannel、SocketChannel 和 Buffer, 我们这里整合一下它们, 给出一个完整的可运行的例子:

```
public class Server{  
  
    public static void main(String[] args) throws IOException {  
  
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();  
  
        // 监听 8080 端口进来的 TCP 链接  
        serverSocketChannel.socket().bind(new InetSocketAddress(8080));  
  
        while (true) {  
  
            // 这里会阻塞, 直到有一个请求的连接进来  
            SocketChannel socketChannel = serverSocketChannel.accept();  
  
            // 开启一个新的线程来处理这个请求, 然后在 while 循环中继续监听 8080 端口  
            SocketHandler handler = new SocketHandler(socketChannel);  
            new Thread(handler).start();  
        }  
    }  
}
```

这里看一下新的线程需要做什么, SocketHandler:

```
public class SocketHandler implements Runnable{  
  
    private SocketChannel socketChannel;  
  
    public SocketHandler(SocketChannel socketChannel) {  
        this.socketChannel = socketChannel;  
    }  
  
    @Override  
    public void run() {  
  
        ByteBuffer buffer = ByteBuffer.allocate(1024);  
        try {  
            // 将请求数据读入 Buffer 中  
            int num;  
            while ((num = socketChannel.read(buffer)) > 0) {  
                // 读取 Buffer 内容之前先 flip 一下  
                buffer.flip();  
  
                // 提取 Buffer 中的数据  
                byte[] bytes = new byte[num];  
                buffer.get(bytes);  
  
                String re = new String(bytes, "UTF-8");  
                System.out.println("收到请求: " + re);  
  
                // 回应客户端  
                ByteBuffer writeBuffer = ByteBuffer.wrap(("我已经收到你的请求，你的请求内容是: " + re).getBytes());  
                socketChannel.write(writeBuffer);  
  
                buffer.clear();  
            }  
        } catch (IOException e) {  
            IOUtils.closeQuietly(socketChannel);  
        }  
    }  
}
```

最后，贴一下客户端 SocketChannel 的使用，客户端比较简单：

```

public class SocketChannelTest{
    public static void main(String[] args) throws IOException {
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.connect(new InetSocketAddress("localhost", 8080));

        // 发送请求
        ByteBuffer buffer = ByteBuffer.wrap("1234567890".getBytes());
        socketChannel.write(buffer);

        // 读取响应
        ByteBuffer readBuffer = ByteBuffer.allocate(1024);
        int num;
        if ((num = socketChannel.read(readBuffer)) > 0) {
            readBuffer.flip();

            byte[] re = new byte[num];
            readBuffer.get(re);

            String result = new String(re, "UTF-8");
            System.out.println("返回值: " + result);
        }
    }
}

```

上面介绍的阻塞模式的代码应该很好理解：来一个新的连接，我们就新开一个线程来处理这个连接，之后的操作全部由那个线程来完成。

那么，这个模式下的性能瓶颈在哪里呢？

- 首先，每次来一个连接都开一个新的线程这肯定是不合适的。当活跃连接数在几十几百的时候当然是可以这样做的，但如果活跃连接数是几万几十万的时候，这么多线程明显就不行了。每个线程都需要一部分内存，内存会被迅速消耗，同时，线程切换的开销非常大。
- 其次，阻塞操作在这里也是一个问题。首先，accept() 是一个阻塞操作，当 accept() 返回的时候，代表有一个连接可以使用了，我们这里是马上就新建线程来处理这个 SocketChannel 了，但是，但是这里不代表对方就将数据传输过来了。所以，SocketChannel#read 方法将阻塞，等待数据，明显这个等待是不值得的。同理，write 方法也需要等待通道可写才能执行写入操作，这边的阻塞等待也是不值得的。

非阻塞 IO

说完了阻塞模式的使用及其缺点以后，我们这里就可以介绍非阻塞 IO 了。

非阻塞 IO 的核心在于使用一个 Selector 来管理多个通道，可以是 SocketChannel，也可以是 ServerSocketChannel，将各个通道注册到 Selector 上，指定监听的事件。

之后可以只用一个线程来轮询这个 Selector，看看上面是否有通道是准备好的，当通道准备好可读或可写，然后才去开始真正的读写，这样速度就很快了。我们就完全没有必要给每个通道都起一个线程。

NIO 中 Selector 是对底层操作系统实现的一个抽象，管理通道状态其实都是底层系统实现的，这里简单介绍下在不同系统下的实现。

select: 上世纪 80 年代就实现了，它支持注册 FD_SETSIZE(1024) 个 socket，在那个年代肯定是够用的，不过现在嘛，肯定不行了。

poll: 1997 年，出现了 poll 作为 select 的替代者，最大的区别就是，poll 不再限制 socket 数量。

select 和 poll 都有一个共同的问题，那就是**它们都只会告诉你有几个通道准备好了，但是不会告诉你具体是哪几个通道**。所以，一旦知道有通道准备好以后，自己还是需要进行一次扫描，显然这个不太好，通道少的时候还行，一旦通道的数量是几十万个以上的时候，扫描一次的时间都很可观了，时间复杂度 $O(n)$ 。所以，后来才催生了以下实现。

epoll: 2002 年随 Linux 内核 2.5.44 发布，epoll 能直接返回具体的准备好的通道，时间复杂度 $O(1)$ 。

除了 Linux 中的 epoll，2000 年 FreeBSD 出现了**Kqueue**，还有就是，Solaris 中有 **/dev/poll**。

前面说了那么多实现，但是没有出现 Windows，Windows 平台的非阻塞 IO 使用 select，我们也不必觉得 Windows 很落后，在 Windows 中 IOCP 提供的异步 IO 是比较强大的。

我们回到 Selector，毕竟 JVM 就是这么一个屏蔽底层实现的平台，**我们面向 Selector 编程就可以了**。

之前在介绍 Selector 的时候已经了解过了它的基本用法，这边来一个可运行的实例代码，大家不妨看看：

```
public class SelectorServer{  
  
    public static void main(String[] args) throws IOException {  
        Selector selector = Selector.open();  
  
        ServerSocketChannel server = ServerSocketChannel.open();  
        server.socket().bind(new InetSocketAddress(8080));  
  
        // 将其注册到 Selector 中，监听 OP_ACCEPT 事件  
        server.configureBlocking(false);  
        server.register(selector, SelectionKey.OP_ACCEPT);  
  
        while (true) {  
            int readyChannels = selector.select();  
            if (readyChannels == 0) {  
                continue;  
            }  
            Set<SelectionKey> readyKeys = selector.selectedKeys();  
            // 遍历  
            Iterator<SelectionKey> iterator = readyKeys.iterator();  
            while (iterator.hasNext()) {  
                SelectionKey key = iterator.next();  
                iterator.remove();  
  
                if (key.isAcceptable()) {  
                    // 有已经接受的新的到服务端的连接  
                    SocketChannel socketChannel = server.accept();  
                }  
            }  
        }  
    }  
}
```

```
// 有新的连接并不代表这个通道就有数据，  
// 这里将这个新的 SocketChannel 注册到 Selector，监听 OP_READ 事件，等待数据  
socketChannel.configureBlocking(false);  
socketChannel.register(selector, SelectionKey.OP_READ);  
} else if (key.isReadable()) {  
    // 有数据可读  
    // 上面一个 if 分支中注册了监听 OP_READ 事件的 SocketChannel  
    SocketChannel socketChannel = (SocketChannel) key.channel();  
    ByteBuffer readBuffer = ByteBuffer.allocate(1024);  
    int num = socketChannel.read(readBuffer);  
    if (num > 0) {  
        // 处理进来的数据...  
        System.out.println("收到数据：" + new String(readBuffer.array()).trim());  
        ByteBuffer buffer = ByteBuffer.wrap("返回给客户端的数据...".getBytes());  
        socketChannel.write(buffer);  
    } else if (num == -1) {  
        // -1 代表连接已经关闭  
        socketChannel.close();  
    }  
}  
}  
}  
}
```

至于客户端，大家可以继续使用上一节介绍阻塞模式时的客户端进行测试。

NIO.2 异步 IO

More New IO，或称 NIO.2，随 JDK 1.7 发布，包括了引入异步 IO 接口和 Paths 等文件访问接口。

异步这个词，我想对于绝大多数开发者来说都很熟悉，很多场景下我们都会使用异步。

通常，我们会有一个线程池用于执行异步任务，提交任务的线程将任务提交到线程池就可以立马返回，不必等到任务真正完成。如果想要知道任务的执行结果，通常是通过传递一个回调函数的方式，任务结束后去调用这个函数。

同样的原理，Java 中的异步 IO 也是一样的，都是由一个线程池来负责执行任务，然后使用回调或自己去查询结果。

大部分开发者都知道为什么要这么设计了，这里再啰嗦一下。异步 IO 主要是为了控制线程数量，减少过多的线程带来的内存消耗和 CPU 在线程调度上的开销。

在 Unix/Linux 等系统中，JDK 使用了并发包中的线程池来管理任务，具体可以查看 AsynchronousChannelGroup 的源码。

在 Windows 操作系统中，提供了一个叫做 I/O Completion Ports 的方案，通常简称为 **IOCP**，操作系统负责管理线程池，其性能非常优异，所以在 Windows 中 JDK 直接采用了 IOCP 的支持，使用系统支持，把更多的操作信息暴露给操作系统，也使得操作系统能够对我们的 IO 进行一定程度的优化。

在 Linux 中其实也是有异步 IO 系统实现的，但是限制比较多，性能也一般，所以 JDK 采用了自建线程池的方式。

本文还是以实用为主，想要了解更多信息请自行查找其他资料，下面对 Java 异步 IO 进行实践性的介绍。

总共有三个类需要我们关注，分别

是 **AsynchronousSocketChannel**, **AsynchronousServerSocketChannel** 和 **AsynchronousFileChannel**, 只不过是在之前介绍的 **FileChannel**、**SocketChannel** 和 **ServerSocketChannel** 的类名上加了个前缀 **Asynchronous**。

Java 异步 IO 提供了两种使用方式，分别是返回 Future 实例和使用回调函数。

1、返回 Future 实例

返回 `java.util.concurrent.Future` 实例的方式我们应该很熟悉，JDK 线程池就是这么使用的。Future 接口的几个方法语义在这里也是通用的，这里先做简单介绍。

- `future.isDone();`

判断操作是否已经完成，包括了**正常完成、异常抛出、取消**

- `future.cancel(true);`

取消操作，方式是中断。参数 true 说的是，即使这个任务正在执行，也会进行中断。

- `future.isCancelled();`

是否被取消，只有在任务正常结束之前被取消，这个方法才会返回 true

- `future.get();`

这是我们的老朋友，获取执行结果，阻塞。

- `future.get(10, TimeUnit.SECONDS);`

如果上面的 `get()` 方法的阻塞你不满意，那就设置个超时时间。

2、提供 CompletionHandler 回调函数

`java.nio.channels.CompletionHandler` 接口定义：

```
public interfaceCompletionHandler<V,A> {  
  
    voidcompleted(V result, A attachment);  
  
    voidfailed(Throwable exc, A attachment);  
}
```

注意，参数上有个 `attachment`，虽然不常用，我们可以在各个支持的方法中传递这个参数值

```
AsynchronousServerSocketChannel listener = AsynchronousServerSocketChannel.open().bind(null);

// accept 方法的第一个参数可以传递 attachment
listener.accept(attachment, new CompletionHandler<AsynchronousSocketChannel, Object>() {
    public void completed(
        AsynchronousSocketChannel client, Object attachment) {
        // ...
    }

    public void failed(Throwable exc, Object attachment) {
        // ...
    }
});
```

AsynchronousFileChannel

网上关于 Non-Blocking IO 的介绍文章很多，但是 Asynchronous IO 的文章相对就少得多了，所以我这边会多介绍一些相关内容。

首先，我们就来关注异步的文件 IO，前面我们说了，文件 IO 在所有的操作系统中都不支持非阻塞模式，但是我们可以对文件 IO 采用异步的方式来提高性能。

下面，我会介绍 AsynchronousFileChannel 里面的一些重要的接口，都很简单，读者要是觉得无趣，直接滑到下一个标题就可以了。

实例化：

```
AsynchronousFileChannel channel = AsynchronousFileChannel.open(Paths.get("/Users/hongjie/test.txt"));
```

一旦实例化完成，我们就可以着手准备将数据读入到 Buffer 中：

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
Future<Integer> result = channel.read(buffer, 0);
```

异步文件通道的读操作和写操作都需要提供一个文件的开始位置，文件开始位置为 0

除了使用返回 Future 实例的方式，也可以采用回调函数进行操作，接口如下：

```
public abstract <A> void read(ByteBuffer dst,
    long position,
    A attachment,
    CompletionHandler<Integer, ? super A> handler);
```

顺便也贴一下写操作的两个版本的接口：

```
public abstract Future<Integer> write(ByteBuffer src, long position);
```

```
public abstract <A> void write(ByteBuffer src,  
                                long position,  
                                A attachment,  
                                CompletionHandler<Integer,? super A> handler);
```

我们可以看到，AIO 的读写主要也还是与 Buffer 打交道，这个与 NIO 是一脉相承的。

另外，还提供了用于将内存中的数据刷入到磁盘的方法：

```
public abstract void force(boolean metaData) throws IOException;
```

因为人们对文件的写操作，操作系统并不会直接针对文件操作，系统会缓存，然后周期性地刷入到磁盘。如果希望将数据及时写入到磁盘中，以免断电引发部分数据丢失，可以调用此方法。参数如果设置为 true，意味着同时也将文件属性信息更新到磁盘。

还有，还提供了对文件的锁定功能，我们可以锁定文件的部分数据，这样可以进行排他性的操作。

```
public abstract Future<FileLock> lock(long position, long size, boolean shared);
```

position 是要锁定内容的开始位置，size 指示了要锁定的区域大小，shared 指示需要的是共享锁还是排他锁

当然，也可以使用回调函数的版本：

```
public abstract <A> void lock(long position,  
                               long size,  
                               boolean shared,  
                               A attachment,  
                               CompletionHandler<FileLock,? super A> handler);
```

文件锁定功能上还提供了 tryLock 方法，此方法会快速返回结果：

```
public abstract FileLock tryLock(long position, long size, boolean shared)  
throws IOException;
```

这个方法很简单，就是尝试去获取锁，如果该区域已被其他线程或其他应用锁住，那么立刻返回 null，否则返回 FileLock 对象。

AsynchronousFileChannel 操作大体上也就以上介绍的这些接口，还是比较简单的，这里就少一些废话早点结束好了。

AsynchronousServerSocketChannel

这个类对应的是非阻塞 IO 的 ServerSocketChannel，大家可以类比下使用方式。

我们就废话少说，用代码说事吧：

```
package com.javadoop.nio;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;

public class Server{

    public static void main(String[] args) throws IOException {

        // 实例化，并监听端口
        AsynchronousServerSocketChannel server =
            AsynchronousServerSocketChannel.open().bind(new InetSocketAddress(8080));

        // 自己定义一个 Attachment 类，用于传递一些信息
        Attachment att = new Attachment();
        att.setServer(server);

        server.accept(att, new CompletionHandler<AsynchronousSocketChannel, Attachment>() {
            @Override
            public void completed(AsynchronousSocketChannel client, Attachment att) {
                try {
                    SocketAddress clientAddr = client.getRemoteAddress();
                    System.out.println("收到新的连接：" + clientAddr);

                    // 收到新的连接后，server 应该重新调用 accept 方法等待新的连接进来
                    att.getServer().accept(att, this);
                }
            }

            Attachment newAtt = new Attachment();
            newAtt.setServer(server);
            newAtt.setClient(client);
            newAtt.setReadMode(true);
            newAtt.setBuffer(ByteBuffer.allocate(2048));

            // 这里也可以继续使用匿名实现类，不过代码不好看，所以这里专门定义一个类
            client.read(newAtt.getBuffer(), newAtt, new ChannelHandler());
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

```
@Override  
public void failed(Throwable t, Attachment att) {  
    System.out.println("accept failed");  
}  
});  
// 为了防止 main 线程退出  
try {  
    Thread.currentThread().join();  
} catch (InterruptedException e) {  
}  
}  
}  
}
```

看一下 ChannelHandler 类：

```
package com.javadoop.nio;  
  
import java.io.IOException;  
import java.nio.ByteBuffer;  
import java.nio.channels.CompletionHandler;  
import java.nio.charset.Charset;  
  
public class ChannelHandler implements CompletionHandler<Integer, Attachment> {  
  
    @Override  
    public void completed(Integer result, Attachment att) {  
        if (att.isReadMode()) {  
            // 读取来自客户端的数据  
            ByteBuffer buffer = att.getBuffer();  
            buffer.flip();  
            byte bytes[] = new byte[buffer.limit()];  
            buffer.get(bytes);  
            String msg = new String(buffer.array()).toString().trim();  
            System.out.println("收到来自客户端的数据: " + msg);  
  
            // 响应客户端请求，返回数据  
            buffer.clear();  
            buffer.put("Response from server!".getBytes(Charset.forName("UTF-8")));  
            att.setReadMode(false);  
            buffer.flip();  
            // 写数据到客户端也是异步  
            att.getClient().write(buffer, att, this);  
        } else {  
            // 到这里，说明往客户端写数据也结束了，有以下两种选择：  
            // 1. 继续等待客户端发送新的数据过来  
            // att.setReadMode(true);  
        }  
    }  
}
```

```

// att.setReadMode(true);
// att.getBuffer().clear();
// att.getClient().read(att.getBuffer(), att, this);

// 2. 既然服务端已经返回数据给客户端，断开这次的连接

try {
    att.getClient().close();
} catch (IOException e) {
}
}

@Override
public void failed(Throwable t, Attachment att) {
    System.out.println("连接断开");
}

```

顺便再贴一下自定义的 Attachment 类：

```

public class Attachment{
    private AsynchronousServerSocketChannel server;
    private AsynchronousSocketChannel client;
    private boolean isReadMode;
    private ByteBuffer buffer;
    // getter & setter
}

```

这样，一个简单的服务端就写好了，接下来可以接收客户端请求了。上面我们用的都是回调函数的方式，读者要是感兴趣，可以试试写个使用 Future 的。

AsynchronousSocketChannel

其实，说完上面的 AsynchronousServerSocketChannel，基本上读者也就知道怎么使用 AsynchronousSocketChannel 了，和非阻塞 IO 基本类似。

这边做个简单演示，这样读者就可以配合之前介绍的 Server 进行测试使用了。

```

package com.javadoop.nio;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.charset.Charset;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class Client {

    public static void main(String[] args) throws Exception {
        AsynchronousSocketChannel client = AsynchronousSocketChannel.open();
        // 来个 Future 形式的
        Future<?> future = client.connect(new InetSocketAddress(8080));
        // 阻塞一下，等待连接成功
        future.get();

        Attachment att = new Attachment();
        att.setClient(client);
        att.setReadMode(false);
        att.setBuffer(ByteBuffer.allocate(2048));
        byte[] data = "I am obot!".getBytes();
        att.getBuffer().put(data);
        att.getBuffer().flip();

        // 异步发送数据到服务端
        client.write(att.getBuffer(), att, new ClientChannelHandler());
    }

    // 这里休息一下再退出，给出足够的时间处理数据
    Thread.sleep(2000);
}

}

```

往里面看下 ClientChannelHandler 类：

```

package com.javadoop.nio;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.CompletionHandler;
import java.nio.charset.Charset;

public class ClientChannelHandler implements CompletionHandler<Integer, Attachment> {

```

```

public class ClientChannel implements CompletionHandler<Integer, Attachment> {
    @Override
    public void completed(Integer result, Attachment att) {
        ByteBuffer buffer = att.getBuffer();
        if (att.isReadMode()) {
            // 读取来自服务端的数据
            buffer.flip();
            byte[] bytes = new byte[buffer.limit()];
            buffer.get(bytes);
            String msg = new String(bytes, Charset.forName("UTF-8"));
            System.out.println("收到来自服务端的响应数据: " + msg);

            // 接下来，有以下两种选择:
            // 1. 向服务端发送新的数据
            // att.setReadMode(false);
            // buffer.clear();
            // String newMsg = "new message from client";
            // byte[] data = newMsg.getBytes(Charset.forName("UTF-8"));
            // buffer.put(data);
            // buffer.flip();
            // att.getClient().write(buffer, att, this);
            // 2. 关闭连接
            try {
                att.getClient().close();
            } catch (IOException e) {
            }
        } else {
            // 写操作完成后，会进到这里
            att.setReadMode(true);
            buffer.clear();
            att.getClient().read(buffer, att, this);
        }
    }

    @Override
    public void failed(Throwable t, Attachment att) {
        System.out.println("服务器无响应");
    }
}

```

以上代码都是可以运行调试的，如果读者碰到问题，请在评论区留言。

Asynchronous Channel Groups

为了知识的完整性，有必要对 group 进行介绍，其实也就是介绍 AsynchronousChannelGroup 这个类。之前我们说过，异步

IO 一定存在一个线程池，这个线程池负责接收任务、处理 IO 事件、回调等。这个线程池就在 group 内部，group 一旦关闭，那么相应的线程池就会关闭。

AsynchronousServerSocketChannels 和 AsynchronousSocketChannels 是属于 group 的，当我们调用 AsynchronousServerSocketChannel 或 AsynchronousSocketChannel 的 open() 方法的时候，相应的 channel 就属于默认的 group，这个 group 由 JVM 自动构造并管理。

如果我们想要配置这个默认的 group，可以在 JVM 启动参数中指定以下系统变量：

- `java.nio.channels.DefaultThreadPool.threadFactory`

此系统变量用于设置 ThreadFactory，它应该是 `java.util.concurrent.ThreadFactory` 实现类的全限定类名。一旦我们指定了这个 ThreadFactory 以后，group 中的线程就会使用该类产生。

- `java.nio.channels.DefaultThreadPool.initialSize`

此系统变量也很好理解，用于设置线程池的初始大小。

可能你会想要使用自己定义的 group，这样可以对其中的线程进行更多的控制，使用以下几个方法即可：

- `AsynchronousChannelGroup.withCachedThreadPool(ExecutorService executor, int initialSize)`
- `AsynchronousChannelGroup.withFixedThreadPool(int nThreads, ThreadFactory threadFactory)`
- `AsynchronousChannelGroup.withThreadPool(ExecutorService executor)`

熟悉线程池的读者对这些方法应该很好理解，它们都是 `AsynchronousChannelGroup` 中的静态方法。

至于 group 的使用就很简单了，代码一看就懂：

```
AsynchronousChannelGroup group = AsynchronousChannelGroup
    .withFixedThreadPool(10, Executors.defaultThreadFactory());
AsynchronousServerSocketChannel server = AsynchronousServerSocketChannel.open(group);
AsynchronousSocketChannel client = AsynchronousSocketChannel.open(group);
```

AsynchronousFileChannels 不属于 group。但是它们也是关联到一个线程池的，如果不指定，会使用系统默认的线程池，如果想要使用指定的线程池，可以在实例化的时候使用以下方法：

```
public static AsynchronousFileChannel open(Path file,
    Set<? extends OpenOption> options,
    ExecutorService executor,
    FileAttribute<?>... attrs) {
    ...
}
```

到这里，异步 IO 就算介绍完成了。

小结

我想，本文应该是说清楚了非阻塞 IO 和异步 IO 了，对于异步 IO，由于网上的资料比较少，所以不免篇幅多了些。

我们也要知道，看懂了这些，确实可以学到一些东西，多了解一些知识，但是我们还是很少在工作中将这些知识变成工程代码。

一般而言，我们需要在网络应用中使用 NIO 或 AIO 来提升性能，但是，在工程上，绝不是了解了一些概念，知道了一些接口就可

以的，需要处理的细节还非常多。

这也是为什么 Netty/Mina 如此盛行的原因，因为它们帮助封装好了很多细节，提供给我们用户友好的接口，后面有时间我也会对 Netty 进行介绍。

- E N D -



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！

Java 面试题：百度前 200 页都在这里

唐尤华 Java后端 2019-10-14

点击上方 Java后端, 选择设为星标

技术博文, 及时送达

作者 | 唐尤华

来源 | github.com/tangyouhua

基本概念

- 操作系统中 heap 和 stack 的区别
- 什么是基于注解的切面实现
- 什么是 对象/关系 映射集成模块
- 什么是 Java 的反射机制
- 什么是 ACID
- BS与CS的联系与区别
- Cookie 和 Session的区别
- fail-fast 与 fail-safe 机制有什么区别
- get 和 post请求的区别
- Interface 与 abstract 类的区别
- IOC的优点是什么
- IO 和 NIO的区别， NIO优点
- Java 8 / Java 7 为我们提供了什么新功能
- 什么是竞态条件？举个例子说明。
- JRE、JDK、JVM 及 JIT 之间有什么不同
- MVC的各个部分都有那些技术来实现?如何实现?
- RPC 通信和 RMI 区别
- 什么是 Web Service (Web服务)
- JSWDL开发包的介绍。JAXP、JAXM的解释。SOAP、UDDI,WSDL解释。
- WEB容器主要有哪些功能? 并请列出一些常见的WEB容器名字。
- 一个” .java” 源文件中是否可以包含多个类（不是内部类）？有什么限制
- 简单说说你了解的类加载器。是否实现过类加载器
- 解释一下什么叫AOP (面向切面编程)
- 请简述 Servlet 的生命周期及其相关的方法
- 请简述一下 Ajax 的原理及实现步骤
- 简单描述Struts的主要功能
- 什么是 N 层架构
- 什么是CORBA? 用途是什么
- 什么是Java虚拟机？为什么Java被称作是“平台无关的编程语言”
- 什么是正则表达式？用途是什么？哪个包使用正则表达式来实现模式匹配
- 什么是懒加载 (Lazy Loading)
- 什么是尾递归，为什么需要尾递归
- 什么是控制反转 (Inversion of Control) 与依赖注入 (Dependency Injection)

关键字

finalize

- 什么是finalize()方法

- finalize()方法什么时候被调用
- 析构函数(finalization)的目的是什么
- final 和 finalize 的区别

final

- final关键字有哪些用法
- final 与 static 关键字可以用于哪里？它们的作用是什么
- final, finally, finalize的区别
- final、finalize 和 finally 的不同之处？

能否在运行时向 static final 类型的赋值

- 使用final关键字修饰一个变量时，是引用不能变，还是引用的对象不能变
- 一个类被声明为final类型，表示了什么意思
- throws, throw, try, catch, finally分别代表什么意义

Java 有几种修饰符？分别用来修饰什么

volatile

- volatile 修饰符的有过什么实践
- volatile 变量是什么？ volatile 变量和 atomic 变量有什么不同
- volatile 类型变量提供什么保证？能使得一个非原子操作变成原子操作吗
- 能创建 volatile 数组吗？
- transient变量有什么特点
- super什么时候使用
- public static void 写成 static public void会怎样
- 说明一下public static void main(String args[])这段声明里每个关键字的作用
- 请说出作用域public, private, protected, 以及不写时的区别
- sizeof 是Java 的关键字吗

static

- static class 与 non static class的区别
- static 关键字是什么意思？ Java中是否可以覆盖(override)一个private或者是static的方法
- 静态类型有什么特点
- main() 方法为什么必须是静态的？能不能声明 main() 方法为非静态
- 是否可以从一个静态 (static) 方法内部发出对非静态 (non-static) 方法的调用
- 静态变量在什么时候加载？编译期还是运行期？静态代码块加载的时机呢
- 成员方法是否可以访问静态变量？为什么静态方法不能访问成员变量

switch

- switch 语句中的表达式可以是什么类型数据
- switch 是否能作用在byte 上，是否能作用在long 上，是否能作用在String上
- while 循环和 do 循环有什么不同

操作符

- &操作符和&&操作符有什么区别？

- `a = a + b` 与 `a += b` 的区别?
- 逻辑操作符 (`&`, `|`, `^`) 与条件操作符 (`&&`, `||`) 的区别
- `3 * 0.1 == 0.3` 将会返回什么? `true` 还是 `false`?
- `float f=3.4;` 是否正确?
- `short s1 = 1; s1 = s1 + 1;` 有什么错?

数据结构

基础类型(Primitives)

- 基础类型(Primitives)与封装类型(Wrappers)的区别在哪里
- 简述九种基本数据类型的大小, 以及他们的封装类
- `int` 和 `Integer` 哪个会占用更多的内存? `int` 和 `Integer` 有什么区别? `parseInt()` 函数在什么时候使用到
- `float` 和 `double` 的默认值是多少
- 如何去小数四舍五入保留小数点后两位
- `char` 型变量中能不能存贮一个中文汉字, 为什么

类型转换

- 怎样将 `bytes` 转换为 `long` 类型
- 怎么将 `byte` 转换为 `String`
- 如何将数值型字符转换为数字
- 我们能将 `int` 强制转换为 `byte` 类型的变量吗? 如果该值大于 `byte` 类型的范围, 将会出现什么现象
- 能在不进行强制转换的情况下将一个 `double` 值赋值给 `long` 类型的变量吗
- 类型向下转换是什么

数组

- 如何权衡是使用无序的数组还是有序的数组
- 怎么判断数组是 `null` 还是为空
- 怎么打印数组? 怎样打印数组中的重复元素
- `Array` 和 `ArrayList` 有什么区别? 什么时候应该使用 `Array` 而不是 `ArrayList`
- 数组和链表数据结构描述, 各自的时间复杂度
- 数组有没有 `length()` 这个方法? `String` 有没有 `length()` 这个方法

队列

- 队列和栈是什么, 列出它们的区别
- `BlockingQueue` 是什么
- 简述 `ConcurrentLinkedQueue` `LinkedBlockingQueue` 的用处和不同之处。

ArrayList、Vector、LinkedList的存储性能和特性

String

StringBuffer

- `ByteBuffer` 与 `StringBuffer` 有什么区别

HashMap

- `HashMap` 的工作原理是什么
- 内部的数据结构是什么

- HashMap 的 table 的容量如何确定? loadFactor 是什么? 该容量如何变化? 这种变化会带来什么问题?
- HashMap 实现的数据结构是什么? 如何实现
- HashMap 和 HashTable、ConcurrentHashMap 的区别
- HashMap 的遍历方式及效率
- HashMap、LinkedMap、TreeMap 的区别
- 如何决定选用HashMap还是TreeMap
- 如果HashMap的大小超过了负载因子(load factor)定义的容量, 怎么办
- HashMap 是线程安全的吗? 并发下使用的 Map 是什么, 它们内部原理分别是什么, 比如存储方式、hashcode、扩容、默认容量等

HashSet

- HashSet 和 TreeSet 有什么区别
- HashSet 内部是如何工作的
- WeakHashMap 是怎么工作的?

Set

- Set 里的元素是不能重复的, 那么用什么方法来区分重复与否呢? 是用 == 还是 equals()? 它们有何区别?
- TreeMap: TreeMap 是采用什么树实现的? TreeMap、HashMap、LinkedHashMap 的区别。TreeMap 和 TreeSet 在排序时如何比较元素? Collections 工具类中的 sort() 方法如何比较元素?
- TreeSet: 一个已经构建好的 TreeSet, 怎么完成倒排序。
- EnumSet 是什么

Hash 算法

- Hashcode 的作用
- 简述一致性 Hash 算法
- 有没有可能 两个不相等的对象有相同的 hashcode? 当两个对象 hashcode 相同怎么办? 如何获取值对象
- 为什么在重写 equals 方法的时候需要重写 hashCode 方法? equals 与 hashCode 的异同点在哪里
- a.hashCode() 有什么用? 与 a.equals(b) 有什么关系
- hashCode() 和 equals() 方法的重要性体现在什么地方
- Object: Object 有哪些公用方法? Object 类 hashCode, equals 设计原则? sun 为什么这么设计? Object 类的概述
- 如何在父类中为子类自动完成所有的 hashCode 和 equals 实现? 这么做有何优劣。
- 可以在 hashCode() 中使用随机数字吗?

LinkedHashMap

- LinkedHashMap 和 PriorityQueue 的区别是什么

List

- List, Set, Map 三个接口, 存取元素时各有什么特点
 - List, Set, Map 是否继承自 Collection 接口
 - 遍历一个 List 有哪些不同的方式
 - LinkedList
1. LinkedList 是单向链表还是双向链表
 2. LinkedList 与 ArrayList 有什么区别
 3. 描述下 Java 中集合 (Collections), 接口 (Interfaces), 实现 (Implementations) 的概念。LinkedList 与 ArrayList 的区别是什么?

4. 插入数据时，ArrayList, LinkedList, Vector谁速度较快？

- ArrayList

1. ArrayList 和 HashMap 的默认大小是多少
2. ArrayList 和 LinkedList 的区别，什么时候用 ArrayList?
3. ArrayList 和 Set 的区别?
4. ArrayList, LinkedList, Vector的区别
5. ArrayList是如何实现的，ArrayList 和 LinkedList 的区别
6. ArrayList如何实现扩容
7. Array 和 ArrayList 有何区别？什么时候更适合用Array
8. 说出ArrayList,Vector, LinkedList的存储性能和特性

Map

- Map, Set, List, Queue, Stack
- Map 接口提供了哪些不同的集合视图
- 为什么 Map 接口不继承 Collection 接口

Collections

- 介绍Java中的Collection FrameWork。集合类框架的基本接口有哪些
- Collections类是什么？Collection 和 Collections的区别？Collection、Map的实现
- 集合类框架的最佳实践有哪些
- 为什么 Collection 不从 Cloneable 和 Serializable 接口继承
- 说出几点 Java 中使用 Collections 的最佳实践？
- Collections 中遗留类 (HashTable、Vector) 和现有类的区别

什么是 B+树，B-树，列出实际的使用场景。

接口

- Comparator 与 Comparable 接口是干什么的？列出它们的区别

对象

拷贝(clone)

- 如何实现对象克隆
- 深拷贝和浅拷贝区别
- 深拷贝和浅拷贝如何实现激活机制
- 写clone()方法时，通常都有一行代码，是什么

比较

- 在比较对象时，“==”运算符和 equals 运算有何区别
- 如果要重写一个对象的equals方法，还要考虑什么
- 两个对象值相同(x.equals(y) == true)，但却可有不同的hash code，这句话对不对

构造器

- 构造器链是什么
- 创建对象时构造器的调用顺序

不可变对象

- 什么是不可变象 (immutable object)
- 为什么 Java 中的 String 是不可变的 (Immutable)
- 如何构建不可变的类结构？关键点在哪里
- 能创建一个包含可变对象的不可变对象吗
- 如何对一组对象进行排序

方法

- 构造器 (constructor) 是否可被重写 (override)
- 方法可以同时即是 static 又是 synchronized 的吗
- abstract 的 method 是否可同时是 static，是否可同时是 native，是否可同时是 synchronized
- Java 支持哪种参数传递类型
- 一个对象被当作参数传递到一个方法，是值传递还是引用传递
- 当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递
- 我们能否重载 main() 方法
- 如果 main 方法被声明为 private 会怎样

GC

概念

- GC 是什么？为什么要有关 GC
- 什么时候会导致垃圾回收
- GC 是怎么样运行的
- 新老以及永久区是什么
- GC 有几种方式？怎么配置
- 什么时候一个对象会被 GC？如何判断一个对象是否存活
- System.gc() Runtime.gc() 会做什么事情？能保证 GC 执行吗
- 垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？
- Minor GC、Major GC、Young GC 与 Full GC 分别在什么时候发生
- 垃圾回收算法的实现原理
- 如果对象的引用被置为 null，垃圾收集器是否会立即释放对象占用的内存？
- 垃圾回收的最佳做法是什么

GC 收集器有哪些

- 垃圾回收器的基本原理是什么？
- 串行(serial)收集器和吞吐量(throughput)收集器的区别是什么
- Serial 与 Parallel GC 之间的不同之处
- CMS 收集器与 G1 收集器的特点与区别
- CMS 垃圾回收器的工作过程
- JVM 中一次完整的 GC 流程是怎样的？对象如何晋升到老年期
- 吞吐量优先和响应优先的垃圾收集器选择

GC 策略

- 举个实际的场景，选择一个GC策略
- JVM的永久代中会发生垃圾回收吗

收集方法

标记清除、标记整理、复制算法的原理与特点？分别用在什么地方
如果让你优化收集方法，有什么思路

JVM

参数

- 说说你知道的几种主要的jvm参数
- -XX:+UseCompressedOops有什么作用

类加载器(ClassLoader)

- Java 类加载器都有哪些
- JVM如何加载字节码文件

内存管理

- JVM内存分哪几个区，每个区的作用是什么
- 一个对象从创建到销毁都是怎么在这些部分里存活和转移的
- 解释内存中的栈(stack)、堆(heap)和方法区(method area)的用法
- JVM中哪个参数是用来控制线程的栈堆栈小
- 简述内存分配与回收策略
- 简述重排序，内存屏障，happen-before，主内存，工作内存
- Java中存在内存泄漏问题吗？请举例说明
- 简述 Java 中软引用 (SoftReference)、弱引用 (WeakReference) 和虚引用
- 内存映射缓存区是什么

1. jstack, jstat, jmap, jconsole怎么用
2. 32位JVM和64位JVM的最大堆内存分别是多大？32位和64位的JVM，int类型变量的长度是多少？
3. 怎样通过Java程序来判断JVM是32位还是64位
4. JVM自身会维护缓存吗？是不是在堆中进行对象分配，操作系统的堆还是JVM自己管理堆
5. 什么情况下会发生栈内存溢出
6. 双亲委派模型是什么

多线程

基本概念

- 什么是线程
- 多线程的优点
- 多线程的几种实现方式

1. 用Runnable还是Thread

- 什么是线程安全

1. Vector, SimpleDateFormat 是线程安全类吗
2. 什么 Java 原型不是线程安全的
3. 哪些集合类是线程安全的

- 多线程中的忙循环是什么
- 如何创建一个线程
- 编写多线程程序有几种实现方式
- 什么是线程局部变量
- 线程和进程有什么区别？进程间如何通讯，线程间如何通讯
- 什么是多线程环境下的伪共享（false sharing）
- 同步和异步有何异同，在什么情况下分别使用他们？举例说明

Current

- ConcurrentHashMap 和 Hashtable 的区别
- ArrayBlockingQueue, CountDownLatch 的用法
- ConcurrentHashMap 的并发度是什么

CyclicBarrier 和 CountDownLatch 有什么不同？各自的内部原理和用法是什么

Semaphore 的用法

Thread

- 启动一个线程是调用 run() 还是 start() 方法？start() 和 run() 方法有什么区别
- 调用 start() 方法时会执行 run() 方法，为什么不能直接调用 run() 方法
- sleep() 方法和对象的 wait() 方法都可以让线程暂停执行，它们有什么区别
- yield 方法有什么作用？sleep() 方法和 yield() 方法有什么区别
- Java 中如何停止一个线程
- stop() 和 suspend() 方法为何不推荐使用
- 如何在两个线程间共享数据
- 如何强制启动一个线程
- 如何让正在运行的线程暂停一段时间
- 什么是线程组，为什么在 Java 中不推荐使用
- 你是如何调用 wait（方法的）？使用 if 块还是循环？为什么

生命周期

- 有哪些不同的线程生命周期
- 线程状态，BLOCKED 和 WAITING 有什么区别
- 画一个线程的生命周期状态图

ThreadLocal 用途是什么，原理是什么，用的时候要注意什么

ThreadPool

- 线程池是什么？为什么要使用它
- 如何创建一个 Java 线程池
- ThreadPool 用法与优势
- 提交任务时，线程池队列已满时会发什么
- newCache 和 newFixed 有什么区别？简述原理。构造函数的各个参数的含义是什么，比如 coreSize, maxsize 等
- 线程池的实现策略
- 线程池的关闭方式有几种，各自的区别是什么
- 线程池中 submit() 和 execute() 方法有什么区别？

线程调度

- Java中用到的线程调度算法是什么
- 什么是多线程中的上下文切换
- 你对线程优先级的理解是什么
- 什么是线程调度器 (Thread Scheduler) 和时间分片 (Time Slicing)

线程同步

- 请说出你所知的线程同步的方法
- synchronized 的原理是什么
- synchronized 和 ReentrantLock 有什么不同
- 什么场景下可以使用 volatile 替换 synchronized
- 有T1, T2, T3三个线程, 怎么确保它们按顺序执行? 怎样保证T2在T1执行完后执行, T3在T2执行完后执行
- 同步块内的线程抛出异常会发生什么
- 当一个线程进入一个对象的 synchronized 方法A之后, 其它线程是否可进入此对象的 synchronized 方法B
- 使用 synchronized 修饰静态方法和非静态方法有什么区别
- 如何从给定集合那里创建一个 synchronized 的集合

锁

- Java Concurrency API 中的 Lock 接口是什么? 对比同步它有什么优势
- Lock 与 Synchronized 的区别? Lock 接口比 synchronized 块的优势是什么
- ReadWriteLock是什么?
- 锁机制有什么用
- 什么是乐观锁 (Optimistic Locking) ? 如何实现乐观锁? 如何避免ABA问题
- 解释以下名词: 重排序, 自旋锁, 偏向锁, 轻量级锁, 可重入锁, 公平锁, 非公平锁, 乐观锁, 悲观锁
- 什么时候应该使用可重入锁
- 简述锁的等级方法锁、对象锁、类锁
- Java中活锁和死锁有什么区别?
- 什么是死锁(Deadlock)? 导致线程死锁的原因? 如何确保 N 个线程可以访问 N 个资源同时又不导致死锁
- 死锁与活锁的区别, 死锁与饥饿的区别
- 怎么检测一个线程是否拥有锁
- 如何实现分布式锁
- 有哪些无锁数据结构, 他们实现的原理是什么
- 读写锁可以用于什么应用场景
- Executors类是什么? Executor和Executors的区别
- 什么是Java线程转储(Thread Dump), 如何得到它
- 如何在Java中获取线程堆栈
- 说出 3 条在 Java 中使用线程的最佳实践
- 在线程中你怎么处理不可捕捉异常
- 实际项目中使用多线程举例。你在多线程环境中遇到的常见的问题是什么? 你是怎么解决它的
- 请说出与线程同步以及线程调度相关的方法
- 程序中有3个 socket, 需要多少个线程来处理
- 假如有一个第三方接口, 有很多个线程去调用获取数据, 现在规定每秒钟最多有 10 个线程同时调用它, 如何做到
- 如何在 Windows 和 Linux 上查找哪个线程使用的 CPU 时间最长
- 如何确保 main() 方法所在的线程是 Java 程序最后结束的线程
- 非常多个线程 (可能是不同机器), 相互之间需要等待协调才能完成某种工作, 问怎么设计这种协调方案
- 你需要实现一个高效的缓存, 它允许多个用户读, 但只允许一个用户写, 以此来保持它的完整性, 你会怎样去实现它

异常

基本概念

- Error 和 Exception有什么区别
- 1. UnsupportedOperationException是什么
- 2. NullPointerException 和 ArrayIndexOutOfBoundsException 之间有什么相同之处
- 什么是受检查的异常，什么是运行时异常
- 运行时异常与一般异常有何异同
- 简述一个你最常见到的runtime exception(运行时异常)

finally

- finally关键词在异常处理中如何使用
- 1. 如果执行finally代码块之前方法返回了结果，或者JVM退出了，finally块中的代码还会执行吗
- 2. try里有return, finally还执行么？那么紧跟在这个try后的finally {}里的code会不会被执行，什么时候被执行，在return前还是后
- 3. 在什么情况下，finally语句不会执行
- throw 和 throws 有什么区别？
- OOM你遇到过哪些情况？你是怎么搞定的？
- SOF你遇到过哪些情况？
- 既然我们可以用RuntimeException来处理错误，那么你认为为什么Java中还存在检查型异常
- 当自己创建异常类的时候应该注意什么
- 导致空指针异常的原因
- 异常处理 handle or declare 原则应该如何理解
- 怎么利用 JUnit 来测试一个方法的异常
- catch块里别不写代码有什么问题
- 你曾经自定义实现过异常吗？怎么写的
- 什么是 异常链
- 在try块中可以抛出异常吗

JDBC

- 通过 JDBC 连接数据库有哪几种方式
- 阐述 JDBC 操作数据库的基本步骤
- JDBC 中如何进行事务处理
- 什么是 JdbcTemplate
- 什么是 DAO 模块
- 使用 JDBC 操作数据库时，如何提升读取数据的性能？如何提升更新数据的性能
- 列出 5 个应该遵循的 JDBC 最佳实践

IO

File

- File类型中定义了什么方法来创建一级目录
- File类型中定义了什么方法来判断一个文件是否存在

流

- 为了提高读写性能，可以采用什么流

- Java中有几种类型的流
- JDK 为每种类型的流提供了一些抽象类以供继承，分别是哪些类
- 对文本文件操作用什么I/O流
- 对各种基本数据类型和String类型的读写，采用什么流
- 能指定字符编码的 I/O 流类型是什么

序列化

- 什么是序列化？如何实现 Java 序列化及注意事项
- Serializable 与 Externalizable 的区别

Socket

- socket 选项 TCP NO DELAY 是指什么
- Socket 工作在 TCP/IP 协议栈是哪一层
- TCP、UDP 区别及 Java 实现方式
- 说几点 IO 的最佳实践
- 直接缓冲区与非直接缓冲器有什么区别？
- 怎么读写 ByteBuffer？ByteBuffer 中的字节序是什么
- 当用System.in.read(buffer)从键盘输入一行n个字符后，存储在缓冲区buffer中的字节数是多少
- 如何使用扫描器类（Scanner Class）令牌化

面向对象编程 (OOP)

- 解释下多态性 (polymorphism)，封装性 (encapsulation)，内聚 (cohesion) 以及耦合 (coupling)
- 多态的实现原理
- 封装、继承和多态是什么
- 对象封装的原则是什么？
- 类

1. 获得一个类的类对象有哪些方式
2. 重载 (Overload) 和重写 (Override) 的区别。重载的方法能否根据返回类型进行区分？
3. 说出几条 Java 中方法重载的最佳实践

- 抽象类
 1. 抽象类和接口的区别
 2. 抽象类中是否可以有静态的main方法
 3. 抽象类是否可实现(implements)接口
 4. 抽象类是否可继承具体类(concrete class)

- 匿名类 (Anonymous Inner Class)
 1. 匿名内部类是否可以继承其它类？是否可以实现接口
- 内部类
 1. 内部类分为几种
 2. 内部类可以引用它的包含类（外部类）的成员吗
 3. 请说一下 Java 中为什么要引入内部类？还有匿名内部类

- 继承

1. 继承 (Inheritance) 与聚合 (Aggregation) 的区别在哪里
2. 继承和组合之间有什么不同
3. 为什么类只能单继承，接口可以多继承
4. 存在两个类，B 继承 A, C 继承 B, 能将 B 转换为 C 吗？如 $C = (C) B$
5. 如果类 a 继承类 b，实现接口c，而类 b 和接口 c 中定义了同名变量，请问会出现什么问题

- 接口

1. 接口是什么
2. 接口是否可继承接口
3. 为什么要使用接口而不是直接使用具体类？接口有什么优点

泛型

- 泛型的存在是用来解决什么问题
- 泛型的常用特点
- List能否转为List

工具类

日历

- Calendar Class的用途
- 如何在Java中获取日历类的实例
- 解释一些日历类中的重要方法
- GregorianCalendar 类是什么
- SimpleTimeZone 类是什么
- Locale类是什么
- 如何格式化日期对象
- 如何添加小时(hour)到一个日期对象(Date Objects)
- 如何将字符串 YYYYMMDD 转换为日期

Math

- Math.round()什么作用？Math.round(11.5) 等于多少？Math.round(-11.5)等于多少？

XML

- XML文档定义有几种形式？它们之间有何本质区别？解析XML文档有哪几种方式？DOM 和 SAX 解析器有什么不同？
- Java解析XML的方式
- 用 jdom 解析 xml 文件时如何解决中文问题？如何解析
- 你在项目中用到了 XML 技术的哪些方面？如何实现

动态代理

- 描述动态代理的几种实现方式，分别说出相应的优缺点

设计模式

- 什么是设计模式（Design Patterns）？你用过哪种设计模式？用在什么场合
- 你知道哪些商业级设计模式？
- 哪些设计模式可以增加系统的可扩展性
- 单例模式

1. 除了单例模式，你在生产环境中还用过什么设计模式？
2. 写 Singleton 单例模式
3. 单例模式的双检锁是什么
4. 如何创建线程安全的 Singleton
5. 什么是类的单例模式
6. 写出三种单例模式实现

- 适配器模式

1. 适配器模式是什么？什么时候使用
2. 适配器模式和代理模式之前有什么不同
3. 适配器模式和装饰器模式有什么区别

- 什么时候使用享元模式
- 什么时候使用组合模式
- 什么时候使用访问者模式
- 什么是模板方法模式
- 请给出1个符合开闭原则的设计模式的例子

开放问题

- 用一句话概括 Web 编程的特点
- Google是如何在一秒内把搜索结果返回给用户
- 哪种依赖注入方式你建议使用，构造器注入，还是 Setter方法注入
- 树（二叉或其他）形成许多普通数据结构的基础。请描述一些这样的数据结构以及何时可以使用它们
- 某一项功能如何设计
- 线上系统突然变得异常缓慢，你如何查找问题
- 什么样的项目不适合用框架
- 新浪微博是如何实现把微博推给订阅者
- 简要介绍下从浏览器输入 URL 开始到获取到请求界面之后 Java Web 应用中发生了什么
- 请你谈谈SSH整合
- 高并发下，如何做到安全的修改同一行数据
- 12306网站的订票系统如何实现，如何保证不会票不被超卖
- 网站性能优化如何优化的
- 聊了下曾经参与设计的服务器架构
- 请思考一个方案，实现分布式环境下的 countDownLatch
- 请思考一个方案，设计一个可以控制缓存总体大小的自动适应的本地缓存
- 在你的职业生涯中，算得上最困难的技术挑战是什么
- 如何写一篇设计文档，目录是什么
- 大写的O是什么？举几个例子
- 编程中自己都怎么考虑一些设计原则的，比如开闭原则，以及在工作中的应用
- 解释一下网络应用的模式及其特点
- 设计一个在线文档系统，文档可以被编辑，如何防止多人同时对同一份文档进行编辑更新
- 说出数据连接池的工作机制是什么
- 怎么获取一个文件中单词出现的最高频率
- 描述一下你最常用的编程风格

- 如果有机会重新设计你们的产品，你会怎么做
- 如何搭建一个高可用系统
- 如何启动时不需输入用户名与密码
- 如何在基于Java的Web项目中实现文件上传和下载
- 如何实现一个秒杀系统，保证只有几位用户能买到某件商品。
- 如何实现负载均衡，有哪些算法可以实现
- 如何设计一个购物车？想想淘宝的购物车如何实现的
- 如何设计一套高并发支付方案，架构如何设计
- 如何设计建立和保持 100w 的长连接
- 如何避免浏览器缓存。
- 如何防止缓存雪崩
- 如果AB两个系统互相依赖，如何解除依
- 如果有人恶意创建非法连接，怎么解决
- 如果有几十亿的白名单，每天白天需要高并发查询，晚上需要更新一次，如何设计这个功能
- 如果系统要使用超大整数（超过long长度范围），请你设计一个数据结构来存储这种超大型数字以及设计一种算法来实现超大整数加法运算）
- 如果要设计一个图形系统，请你设计基本的图形元件(Point,Line,Rectangle,Triangle)的简单实现
- 如果让你实现一个并发安全的链表，你会怎么做
- 应用服务器与WEB 服务器的区别？应用服务器怎么监控性能，各种方式的区别？你使用过的应用服务器优化技术有哪些
- 大型网站在架构上应当考虑哪些问题
- 有没有处理过线上问题？出现内存泄露，CPU利用率标高，应用无响应时如何处理的
- 最近看什么书，印象最深刻的是什么
- 描述下常用的重构技巧
- 你使用什么版本管理工具？分支（Branch）与标签（Tag）之间的区别在哪里
- 你有了解过存在哪些反模式（Anti-Patterns）吗
- 你用过的网站前端优化的技术有哪些
- 如何分析Thread dump
- 你如何理解AOP中的连接点（Joinpoint）、切点（Pointcut）、增强（Advice）、引介（Introduction）、织入（Weaving）、切面（Aspect）这些概念
- 你是如何处理内存泄露或者栈溢出问题的
- 你们线上应用的 JVM 参数有哪些
- 怎么提升系统的QPS和吞吐量

知识面

- 解释什么是 MESI 协议(缓存一致性)
- 谈谈 reactor 模型
- Java 9 带来了怎样的新功能
- Java 与 C++ 对比，C++ 或 Java 中的异常处理机制的简单原理和应用
- 简单讲讲 Tomcat 结构，以及其类加载器流程
- 虚拟内存是什么
- 阐述下 SOLID 原则
- 请简要讲一下你对测试驱动开发（TDD）的认识
- CDN实现原理
- Maven 和 ANT 有什么区别
- UML中有哪些常用的图
- Linux

1. Linux 下 IO 模型有几种，各自的含义是什么。
2. Linux 系统下你关注过哪些内核参数，说说你知道的
3. Linux 下用一行命令查看文件的最后五行

4. 平时用到哪些 Linux 命令
5. 用一行命令输出正在运行的 Java 进程
6. 使用什么命令来确定是否有 Tomcat 实例运行在机器上

- 什么是 N+1 难题
- 什么是 paxos 算法
- 什么是 restful，讲讲你理解的 restful
- 什么是 zab 协议
- 什么是领域模型(domain model)? 贫血模型(anaemic domain model) 和充血模型(rich domain model)有什么区别
- 什么是领域驱动开发 (Domain Driven Development)
- 介绍一下了解的 Java 领域的 Web Service 框架
- Web Server、Web Container 与 Application Server 的区别是什么
- 微服务 (MicroServices) 与巨石型应用 (Monolithic Applications) 之间的区别在哪里
- 描述 Cookie 和 Session 的作用，区别和各自的应用范围，Session 工作原理
- 你常用的持续集成 (Continuous Integration) 、静态代码分析 (Static Code Analysis) 工具有哪些
- 简述下数据库正则化 (Normalizations)
- KISS,DRY,YAGNI 等原则是什么含义
- 分布式事务的原理，优缺点，如何使用分布式事务？
- 布式集群下如何做到唯一序列号
- 网络

1. HTTPS 的加密方式是什么，讲讲整个加密解密流程
2. HTTPS 和 HTTP 的区别
3. HTTP 连接池实现原理
4. HTTP 集群方案
5. Nginx、lighttpd、Apache 三大主流 Web 服务器的区别

- 是否看过框架的一些代码
- 持久层设计要考虑的问题有哪些？你用过的持久层框架有哪些
- 数值提升是什么
- 你能解释一下里氏替换原则吗
- 你是如何测试一个应用的？知道哪些测试框架
- 传输层常见编程协议有哪些？并说出各自的特点

编程题

计算加班费

加班10小时以下加班费是时薪的1.5倍。加班10小时或以上，按4元/时算。提示：（一个月工作26天，一天正常工作8小时）

- 计算1000月

-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. [从零搭建创业公司后台技术栈](#)
2. [如何阅读 Java 源码？](#)
3. [某小公司RESTful、前后端分离的实践](#)
4. [该如何弥补 GitHub 功能缺陷？](#)
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！

Java：如何更优雅的处理空值？

lrwinx Java后端 2019-11-22

点击上方 Java后端, 选择 [设为星标](#)

优质文章，及时送达

来源 | lrwinx

作者 | <https://lrwinx.github.io/>

导语

在笔者几年的开发经验中，经常看到项目中存在到处空值判断的情况，这些判断，会让人觉得摸不着头绪，它的出现很有可能和当前的业务逻辑并没有关系。但它会让你很头疼。

有时候，更可怕的是系统因为这些空值的情况，会抛出空指针异常，导致业务系统发生问题。

此篇文章，我总结了几种关于空值的处理手法，希望对读者有帮助。

业务中的空值

场景

存在一个UserSearchService用来提供用户查询的功能：

```
public interface UserSearchService{
    List<User> listUser();
    User get(Integer id);
}
```

问题现场

对于面向对象语言来讲，抽象层级特别的重要。尤其是对接口的抽象，它在设计和开发中占很大的比重，我们在开发时希望尽量面向接口编程。

对于以上描述的接口方法来看，大概可以推断出可能它包含了以下两个含义：

- listUser(): 查询用户列表
- get(Integer id): 查询单个用户

在所有的开发中，XP推崇的TDD模式可以很好的引导我们对接口的定义，所以我们将TDD作为开发代码的“推动者”。

对于以上的接口，当我们使用TDD进行测试用例先行时，发现了潜在的问题：

- listUser() 如果没有数据，那它是返回空集合还是null呢？
- get(Integer id) 如果没有这个对象，是抛异常还是返回null呢？

深入listUser研究

listUser()

这个接口，我经常看到如下实现：

```
public List<User> listUser(){
    List<User> userList = userListRepostity.selectByExample(new UserExample());
    if(CollectionUtils.isEmpty(userList)){//spring util工具类
        return null;
    }
    return userList;
}
```

这段代码返回是null,从我多年的开发经验来讲，对于集合这样返回值，最好不要返回null，因为如果返回了null，会给调用者带来很多麻烦。你将会把这种调用风险交给调用者来控制。

如果调用者是一个谨慎的人，他会进行是否为null的条件判断。如果他并非谨慎，或者他是一个面向接口编程的狂热分子(当然，面向接口编程是正确的方向)，他会按照自己的理解去调用接口，而不进行是否为null的条件判断，如果这样的话，是非常危险的，它很有可能出现空指针异常！

根据墨菲定律来判断：“**很有可能出现的问题，在将来一定会出现！”**

基于此，我们将它进行优化：

```
public List<User> listUser(){
    List<User> userList = userListRepostity.selectByExample(new UserExample());
    if(CollectionUtils.isEmpty(userList)){
        return Lists.newArrayList();//guava类库提供的方法
    }
    return userList;
}
```

对于接口 **(List listUser())**，它一定会返回List，即使没有数据，它仍然会返回List（集合中没有任何元素）；

通过以上的修改，我们成功的避免了有可能发生的空指针异常，这样的写法更安全！

深入研究get方法

对于接口

```
User get(Integer id)
```

你能看到的现象是，我给出id，它一定会给我返回User.但事实真的很有可能不是这样的。

我看到过的实现：

```
public User get(Integer id){
    return userRepository.selectByPrimaryKey(id); //从数据库中通过id直接获取实体对象
}
```

相信很多人也都会这样写。

通过代码的时候得知它的返回值很有可能是null! 但我们通过的接口是分辨不出来的!

这个是个非常危险的事情。尤其对于调用者来说！

我给出的建议是，需要在接口明确定义时补充文档,比如对于异常的说明,使用注解@exception:

```
public interface UserSearchService{  
  
    /**  
     * 根据用户id获取用户信息  
     * @param id 用户id  
     * @return 用户实体  
     * @exception UserNotFoundException  
     */  
    User get(Integer id);  
  
}
```

我们把接口定义加上了说明之后，调用者会看到，如果调用此接口，很有可能抛出“UserNotFoundException(找不到用户)”这样的异常。

这种方式可以在调用者调用接口的时候看到接口的定义，但是，这种方式是”弱提示”的！

如果调用者忽略了注释，有可能就对业务系统产生了风险，这个风险有可能导致一个亿！

除了以上这种”弱提示”的方式，还有一种方式是，返回值是有可能为空的。那要怎么办呢？

我认为我们需要增加一个接口，用来描述这种场景.

引入jdk8的Optional,或者使用guava的Optional.看如下定义:

```
public interface UserSearchService{  
  
    /**  
     * 根据用户id获取用户信息  
     * @param id 用户id  
     * @return 用户实体,此实体有可能是缺省值  
     */  
    Optional<User> getOptional(Integer id);  
}
```

Optional有两个含义: 存在 or 缺省。

那么通过阅读接口getOptional()，我们可以很快的了解返回值的意图，这个其实是我们想看到的，它去除了二义性。

它的实现可以写成:

```
public Optional<User> getOptional(Integer id){  
    return Optional.ofNullable(userRepository.selectByPrimaryKey(id));  
}
```

深入入参

通过上述的所有接口的描述，你能确定入参id一定是必传的吗？我觉得答案应该是：不能确定。除非接口的文档注释上加以说明。

那如何约束入参呢？

我给大家推荐两种方式：

- 强制约束

■ 文档性约束（弱提示）

1. 强制约束，我们可以通过jsr 303进行严格的约束声明：

```
public interface UserSearchService{  
    /**  
     * 根据用户id获取用户信息  
     * @param id 用户id  
     * @return 用户实体  
     * @exception UserNotFoundException  
     */  
    User get(@NotNull Integer id);  
  
    /**  
     * 根据用户id获取用户信息  
     * @param id 用户id  
     * @return 用户实体,此实体有可能是缺省值  
     */  
    Optional<User> getOptional(@NotNull Integer id);  
}
```

当然，这样写，要配合AOP的操作进行验证，但让spring已经提供了很好的集成方案，在此我就不在赘述了。

2. 文档性约束

在很多时候，我们会遇到遗留代码，对于遗留代码，整体性改造的可能性很小。

我们更希望通过阅读接口的实现，来进行接口的说明。

jsr 305规范，给了我们一个描述接口入参的一个方式(需要引入库 com.google.code.findbugs:jsr305)：

可以使用注解: `@Nullable` `@Nonnull` `@CheckForNull` 进行接口说明。比如：

```
public interface UserSearchService{  
    /**  
     * 根据用户id获取用户信息  
     * @param id 用户id  
     * @return 用户实体  
     * @exception UserNotFoundException  
     */  
    @CheckForNull  
    User get(@NotNull Integer id);  
  
    /**  
     * 根据用户id获取用户信息  
     * @param id 用户id  
     * @return 用户实体,此实体有可能是缺省值  
     */  
    Optional<User> getOptional(@NotNull Integer id);  
}
```

小结

通过 空集合返回值,Optional,jsr 303, jsr 305这几种方式，可以让我们的代码可读性更强，出错率更低！

- 空集合返回值：如果有集合这样返回值时，除非真的有说服自己的理由，否则，一定要返回空集合，而不是null
- Optional：如果你的代码是jdk8，就引入它！如果不是，则使用Guava的Optional，或者升级jdk版本！它很大程度的能增加了接口的可读性！

- jsr 303: 如果新的项目正在开发, 不妨加上这个试试! 一定有一种特别爽的感觉!
- jsr 305: 如果老的项目在你的手上, 你可以尝试的加上这种文档型注解, 有助于你后期的重构, 或者新功能增加了, 对于老接口的理解!

空对象模式

场景

我们来看一个DTO转化的场景, 对象:

```
@Data  
static class PersonDTO{  
    private String dtoName;  
    private String dtoAge;  
}  
  
@Data  
static class Person{  
    private String name;  
    private String age;  
}
```

需求是将Person对象转化成PersonDTO, 然后进行返回。

当然对于实际操作来讲, 返回如果Person为空, 将返回null, 但是PersonDTO是不能返回null的 (尤其Rest接口返回的这种DTO)。

在这里, 我们只关注转化操作, 看如下代码:

```
@Test  
public void shouldConvertDTO(){  
  
    PersonDTO personDTO = new PersonDTO();  
  
    Person person = new Person();  
    if(!Objects.isNull(person)){  
        personDTO.setDtoAge(person.getAge());  
        personDTO.setDtoName(person.getName());  
    }else{  
        personDTO.setDtoAge("");  
        personDTO.setDtoName("");  
    }  
}
```

优化修改

这样的数据转化, 我们认识可读性非常差, 每个字段的判断, 如果是空就设置为空字符串(“”)

换一种思维方式进行思考, 我们是拿到Person这个类的数据, 然后进行赋值操作(setXXX), 其实是不关系Person的具体实现是谁的。

那我们可以创建一个Person子类:

```

static class NullPerson extends Person{
    @Override
    public String getAge() {
        return "";
    }

    @Override
    public String getName() {
        return "";
    }
}

```

它作为Person的一种特例而存在，如果当Person为空的时候，则返回一些get*的默认行为。

所以代码可以修改为：

```

@Test
public void shouldConvertDTO(){

    PersonDTO personDTO = new PersonDTO();

    Person person = getPerson();
    personDTO.setDtoAge(person.getAge());
    personDTO.setDtoName(person.getName());
}

private Person getPerson(){
    return new NullPerson(); //如果Person是null，则返回空对象
}

```

其中getPerson()方法，可以用来根据业务逻辑获取Person有可能的对象（对当前例子来讲，如果Person不存在，返回Person的特例NullPerson），如果修改成这样，代码的可读性就会变的很强了。

使用Optional可以进行优化

空对象模式，它的弊端在于需要创建一个特例对象，但是如果特例的情况比较多，我们是不是需要创建多个特例对象呢，虽然我们也使用了面向对象的多态特性，但是，业务的复杂性如果真的让我们创建多个特例对象，我们还是要再三考虑一下这种模式，它可能会带来代码的复杂性。

对于上述代码，还可以使用Optional进行优化。

```

@Test
public void shouldConvertDTO(){

    PersonDTO personDTO = new PersonDTO();

    Optional.ofNullable(getPerson()).ifPresent(person -> {
        personDTO.setDtoAge(person.getAge());
        personDTO.setDtoName(person.getName());
    });
}

private Person getPerson(){
    return null;
}

```

Optional对空值的使用，我觉得更为贴切，它只适用于“是否存在”的场景。

如果只对控制的存在判断，我建议使用Optional。

Optioanl的正确使用

Optional如此强大，它表达了计算机最原始的特性(0 or 1),那它如何正确的被使用呢!

Optional不要作为参数

如果你写了一个public方法，这个方法规定了一些输入参数，这些参数中有一些是可以传入null的，那这时候是否可以使用Optional呢？

我给的建议是：一定不要这样使用！

举个例子：

```
public interface UserService{  
    List<User> listUser(Optional<String> username);  
}
```

这个例子的方法 listUser,可能在告诉我们需要根据username查询所有数据集合，如果username是空，也要返回所有的用户集合.

当我们看到这个方法的时候，会觉得有一些歧义：

“如果username是absent,是返回空集合吗？还是返回全部的用户数据集合？”

Optioanl是一种分支的判断，那我们究竟是关注 Optional还是Optional.get()呢？

我给大家的建议是，如果不想要这样的歧义，就不要使用它！

如果你真的想表达两个含义，就給它拆分出两个接口：

```
public interface UserService{  
    List<User> listUser(String username);  
    List<User> listUser();  
}
```

我觉得这样的语义更强，并且更能满足 软件设计原则中的 “单一职责”。

如果你觉得你的入参真的有必要可能传null,那请使用jsr 303或者jsr 305进行说明和验证！

请记住！Optional不能作为入参的参数！

Optional作为返回值

当个实体的返回

那Optioanl可以做为返回值吗？

其实它是非常满足是否存在这个语义的。

你如说，你要根据id获取用户信息，这个用户有可能存在或者不存在。

你可以这样使用：

```
public interface UserService{  
    Optional<User> get(Integer id);  
}
```

当调用这个方法的时候，调用者很清楚get方法返回的数据，有可能不存在，这样可以做一些更合理的判断，更好的防止空指针的错误！

当然，如果业务方真的需要根据id必须查询出User的话，就不要这样使用了，请说明，你要抛出的异常。

只有当考虑它返回null是合理的情况下，才进行Optional的返回

集合实体的返回

不是所有的返回值都可以这样用的！如果你返回的是集合：

```
public interface UserService{  
    Optional<List<User>> listUser();  
}
```

这样的返回结果，会让调用者不知所措，是否我判断Optional之后，还用进行isEmpty的判断呢？

这样带来的返回值歧义！我认为是没有必要的。

我们要约定，对于List这种集合返回值，如果集合真的是null的，请返回空集合(Lists.newArrayList());

使用Optional变量

```
Optional<User> userOpt = ...
```

如果有这样的变量userOpt,请记住：

- 一定不能直接使用get，如果这样用，就丧失了Optional本身的含义（比如userOp.get()）
- 不要直接使用getOrThrow,如果你有这样的需求：获取不到就抛异常。那就要考虑，是否是调用的接口设计的是不合理

getter中的使用

对于一个java bean,所有的属性都有可能返回null,那是否需要改写所有的getter成为Optional类型呢？

我给大家的建议是，不要这样滥用Optional.

即便我java bean中的getter是符合Optional的，但是因为java bean 太多了，这样会导致你的代码有50%以上进行Optional的判断，这样便污染了代码。（我想说，其实你的实体中的字段应该都是由业务含义的，会认真的思考过它存在的价值的，不能因为Optional的存在而滥用）

我们应该更关注于业务，而不只是空值的判断。

请不要在getter中滥用Optional.

小结

可以这样总结Optional的使用：

- 当使用值为空的情况，并非源于错误时，可以使用Optional!

- Optional不要用于集合操作!
- 不要滥用Optional,比如在java bean的getter中!

【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. 一千行 MySQL 学习笔记
2. 一份工作坚持多久跳槽最合适?
3. 项目中常用到的 19 条 MySQL 优化
4. 零基础认识 Spring Boot
5. 团队开发中 Git 最佳实践



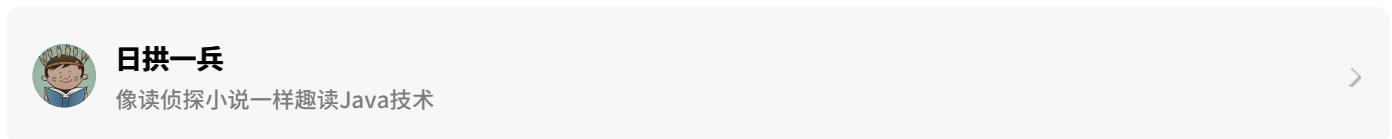
喜欢文章，点个在看

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！

Lombok 使用详解，简化 Java 编程

Java后端 2月18日

以下文章来源于日拱一兵，作者tan日拱一兵



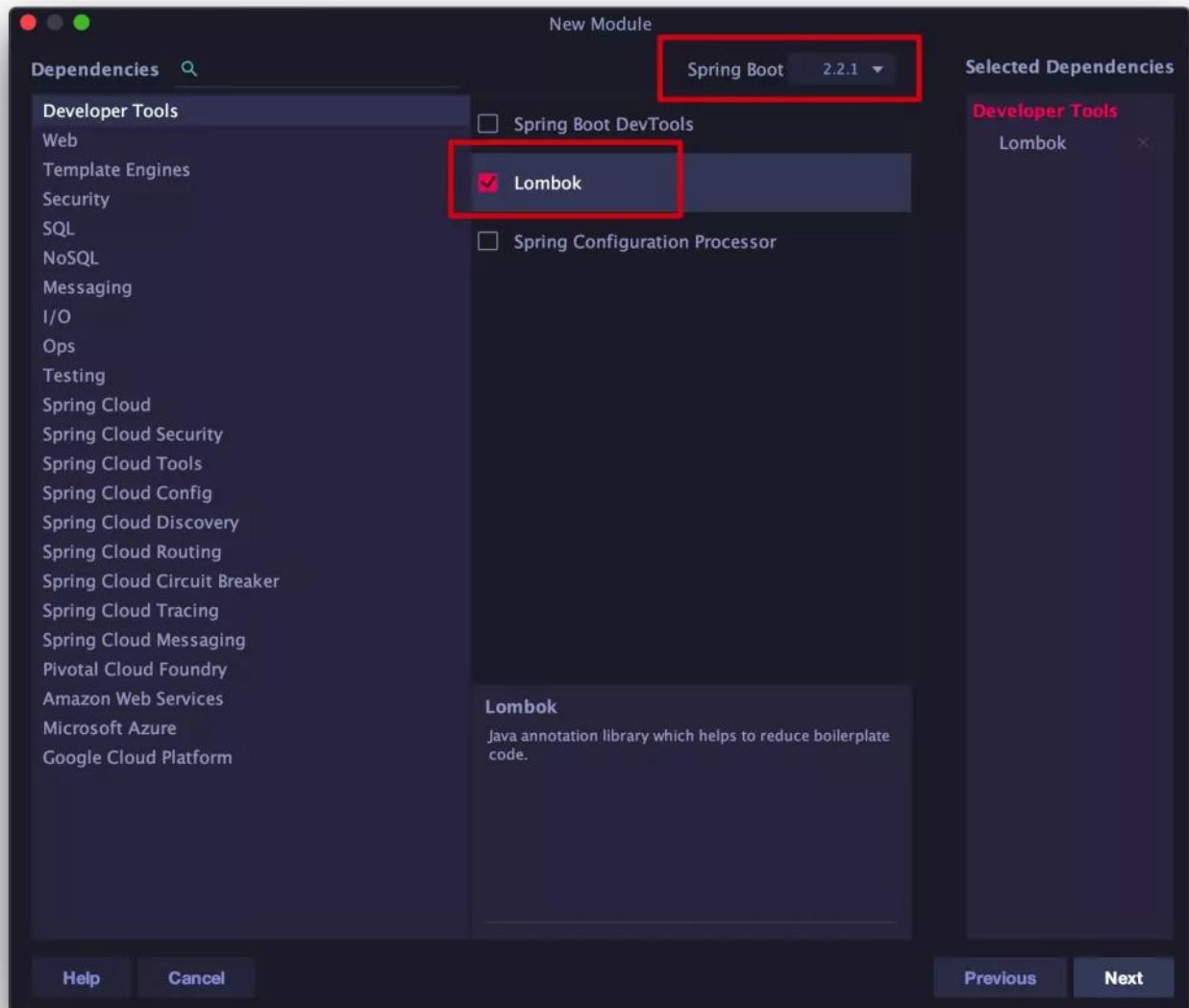
作者 | 日拱一兵

来源 | 公众号（日拱一兵）

在 Java 应用程序中存在许多重复相似的、生成之后几乎不对其做更改的代码，但是我们还不得不花费很多精力编写它们来满足 Java 的编译需求

比如，在 Java 应用程序开发中，我们几乎要为所有 Bean 的成员变量添加 get() ,set() 等方法，这些相对固定但又不得不编写的代码浪费程序员很多精力，同时让类内容看着更杂乱，我们希望将有限的精力关注在更重要的地方。

Lombok 已经诞生很久了，甚至在 Spring Boot Initializr 中都已加入了 Lombok 选项，



这里我们将 Lombok 做一下详细说明：

Lombok

官网的介绍: Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java. Never write another getter or equals method again. Early access to future java features such as val, and much more.

直白的说: Lombok 是一种 Java™ 实用工具, 可用来帮助开发人员消除 Java 的冗长, 尤其是对于简单的 Java 对象 (POJO)。它通过注解实现这一目的, 且看:

Bean 的对比

传统的 POJO 类是这样的

The screenshot shows a Java code editor with two sections side-by-side. On the left, the IDE's navigation bar and a tree view showing a package named 'com.learning.domain' and a class named 'Employee'. Below the tree view, the class definition is shown with various methods and fields. A red box highlights the list of methods: getId(), setId(Integer), getName(), setName(String), getAge(), setAge(Integer), getPassword(), setPassword(String), id, name, age, and password. On the right, the generated Lombok code is shown. A red box highlights the generated getters and setters for id, name, age, and password. The code uses the Lombok annotations @Data and @AllArgsConstructor to generate these methods.

```
1 package com.learning.domain;
2
3 public class Employee {
4     private Integer id;
5     private String name;
6     private Integer age;
7     private String password;
8
9     public Integer getId() {
10         return id;
11     }
12
13     public void setId(Integer id) {
14         this.id = id;
15     }
16
17     public String getName() {
18         return name;
19     }
20
21     public void setName(String name) {
22         this.name = name;
23     }
24
25     public Integer getAge() {
26         return age;
27     }
28
29     public void setAge(Integer age) {
30         this.age = age;
31     }
32
33     public String getPassword() {
34         return password;
35     }
36
37     public void setPassword(String password) {
38         this.password = password;
39     }
40
41 }
42 }
```

通过Lombok改造后的 POJO 类是这样的

The screenshot shows the Employee class in an IDE. The class has the following annotations and methods:

```
1 package com.learning.domain;
2
3 import lombok.Data;
4
5 @Data
6 public class Employee {
7     private Integer id;
8     private String name;
9     private Integer age;
10    private String password;
11 }
12
13 |
```

The methods listed in the code editor are:

- m Employee()
- m getId(): Integer
- m getName(): String
- m getAge(): Integer
- m getPassword(): String
- m setId(Integer): void
- m setName(String): void
- m setAge(Integer): void
- m setPassword(String): void
- m equals(Object): boolean ↑Object
- m hashCode(): int ↑Object
- m canEqual(Object): boolean
- m toString(): String ↑Object

Below the methods, the generated fields are shown:

- f id: Integer
- f name: String
- f age: Integer
- f password: String

一眼可以观察出来我们在编写 Employee 这个类的时候通过@Data 注解就已经实现了所有成员变量的 get() 与 set() 方法等，同时 Employee 类看起来更加清晰简洁。Lombok 的神奇之处不止这些，丰富的注解满足了我们开发的多数需求。

Lombok的安装

查看下图，@Data的实现，我们发现这个注解是应用在编译阶段的

```
package lombok;
import ...
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE) ←
public @interface Data {
    String staticConstructor() default "";
}
```

这和我们大多数使用的注解，如 Spring 的注解（在运行时，通过反射来实现业务逻辑）是有很大差别的，如Spring 的 @RestController 注解

```

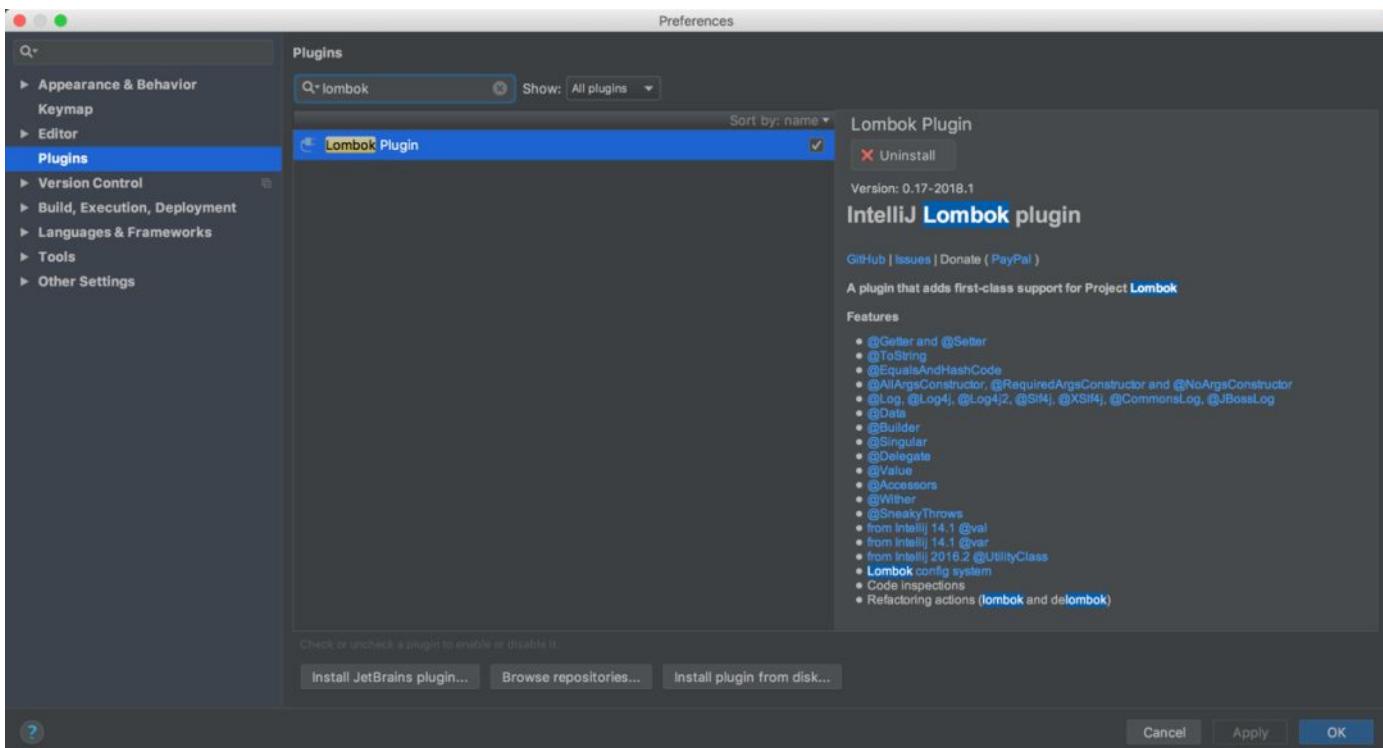
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME) ←——
@Documented
@Controller
@ResponseBody
public @interface RestController {

    /**
     * The value may indicate a suggestion for a logical component name,
     * to be turned into a Spring bean in case of an autodetected component.
     * @return the suggested component name, if any
     * @since 4.0.1
     */
    String value() default "";
}

```

一个更直接的体现就是，普通的包在引用之后一般的 IDE 都能够自动识别语法，但是 Lombok 的这些注解，一般的 IDE 都无法自动识别，因此如果要使用 Lombok 的话还需要配合安装相应的插件来支持 IDE 的编译，防止 IDE 的自动检查报错，下面以 IntelliJ IDEA 举例安装插件。

在Repositories中搜索Lombok，安装后重启IDE即可



在Maven或Gradle工程中添加依赖

```

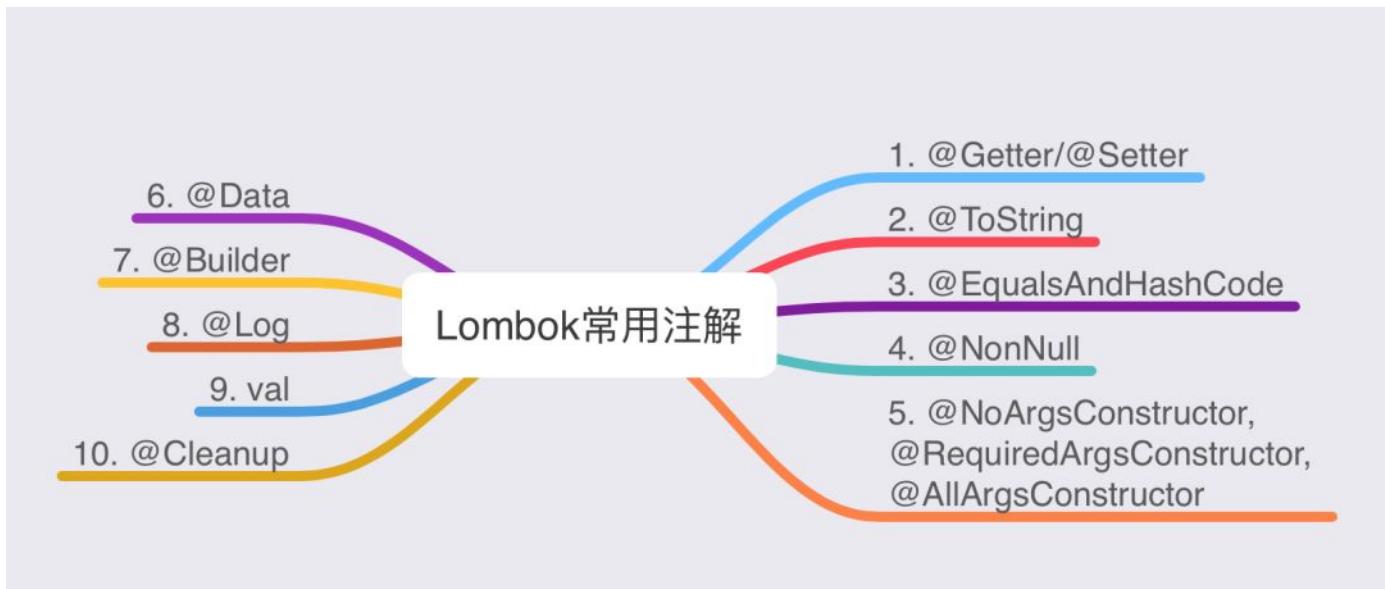
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.10</version>
    <scope>provided</scope>
</dependency>

```

至此我们就可以应用 Lombok 提供的注解干些事情了。

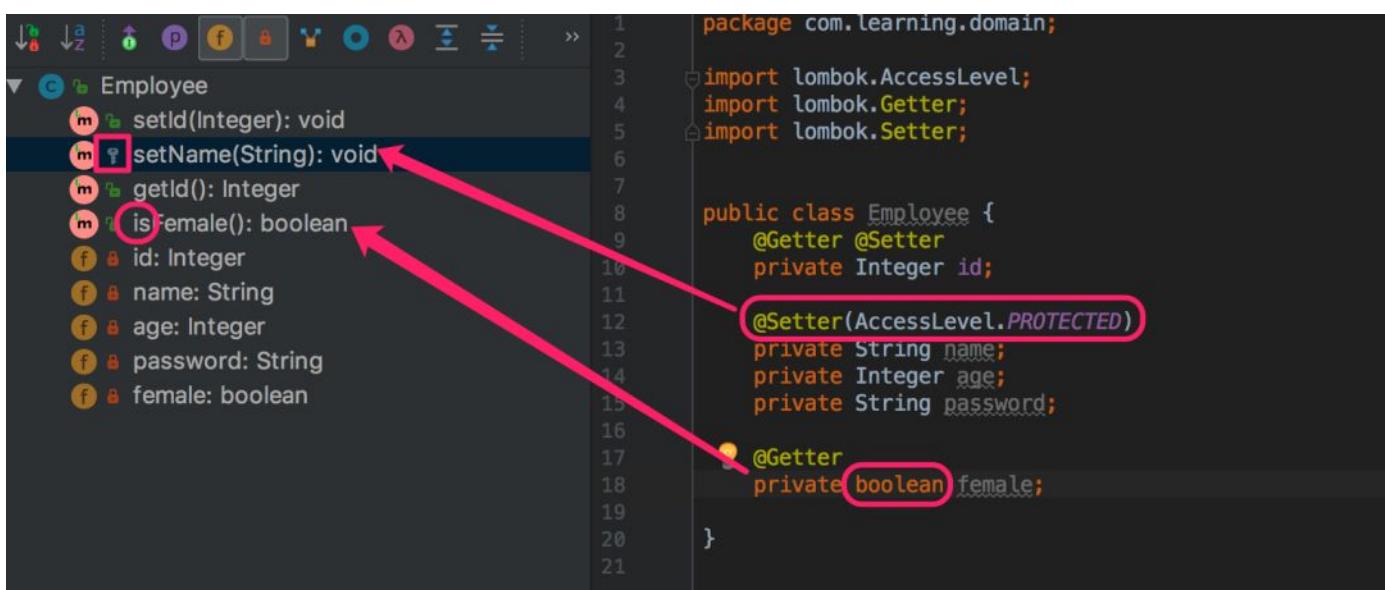
Lombok注解详解

Lombok官网提供了许多注解，但是“劲酒虽好，可不要贪杯哦”，接下来逐一讲解官网推荐使用的注解(有些注解和原有Java编写方式没太大差别的也没有在此处列举，如@ Synchronized等)



@Getter和@Setter

该注解可应用在类或成员变量之上，和我们预想的一样，@Getter 和 @Setter 就是为成员变量自动生成 get 和 set 方法，默认生成访问权限为 public 方法，当然我们也可以指定访问权限 protected 等，如下图：



成员变量name指定生成set方法，并且访问权限为protected；boolean类型的成员变量 female 只生成get方法，并修改方法名为 isFemale()。当把该注解应用在类上，默认为所有非静态成员变量生成 get 和 set 方法，也可以通过 AccessLevel.NONE 手动禁止生成get或set方法，如下图：

```

1 package com.learning.domain;
2
3 import lombok.AccessLevel;
4 import lombok.Getter;
5 import lombok.Setter;
6
7
8 @Getter @Setter
9 public class Employee {
10
11     private Integer id;
12
13     @Setter(AccessLevel.NONE)
14     private String name;
15
16     private Integer age;
17     private String password;
18     private boolean female;
19
20 }
21

```

@ToString

该注解需应用在类上,为我们生成 Object 的 `toString` 方法,而该注解里面的几个属性能更加丰富我们想要的内容, `exclude` 属性禁止在 `toString` 方法中使用某字段,而`of`属性可以指定需要使用的字段,如下图:

```

1 package com.learning.domain;
2
3 import lombok.ToString;
4
5
6 @ToString(exclude = {"id", "name"}, of = {"age", "password"})
7 public class Employee {
8
9     private Integer id;
10    private String name;
11    private Integer age;
12    private String password;
13    private boolean female;
14
15 }
16

```

查看编译后的Employee.class得到我们预期的结果,如下图

```

1 package com.learning.domain;
2
3 public class Employee {
4     private Integer id;
5     private String name;
6     private Integer age;
7     private String password;
8     private boolean female;
9
10    public Employee() {
11    }
12
13    public String toString() {
14        return "Employee{age=" + this.age + ", password=" + this.password + ", female=" + this.female + "}";
15    }
16
17 }
18

```

@EqualsAndHashCode

该注解需应用在类上,使用该注解,lombok会为我们生成 `equals(Object other)` 和 `hashcode()` 方法,包括所有非静态属性和非transient的属性,同样该注解也可以通过 `exclude` 属性排除某些字段, `of` 属性指定某些字段,也可以通过 `callSuper` 属性在重写的方法中使用父类的字段,这样我们可以更灵活的定义bean的比对,如下图:

The screenshot shows the Employee class definition in an IDE. The class has two methods annotated with `@EqualsAndHashCode`: `equals(Object)` and `hashCode()`. The code also includes private fields `id` and `name`.

```
1 package com.learning.domain;
2
3 import lombok.EqualsAndHashCode;
4
5 @EqualsAndHashCode
6 public class Employee {
7     private Integer id;
8     private String name;
9 }
10
11 }
```

查看编译后的Employee.class文件，如下图：

The screenshot shows the decompiled bytecode for the Employee class. It contains the implementation for the `equals`, `canEqual`, and `hashCode` methods.

```
13
14
15     public boolean equals(Object o) {
16         if (o == this) {
17             return true;
18         } else if (!(o instanceof Employee)) {
19             return false;
20         } else {
21             Employee other = (Employee)o;
22             if (!other.canEqual( other: this)) {
23                 return false;
24             } else {
25                 Object this$id = this.id;
26                 Object other$id = other.id;
27                 if (this$id == null) {
28                     if (other$id != null) {
29                         return false;
30                     }
31                 } else if (!this$id.equals(other$id)) {
32                     return false;
33                 }
34
35                 Object this$name = this.name;
36                 Object other$name = other.name;
37                 if (this$name == null) {
38                     if (other$name != null) {
39                         return false;
40                     }
41                 } else if (!this$name.equals(other$name)) {
42                     return false;
43                 }
44
45             }
46         }
47     }
48
49     protected boolean canEqual(Object other) {
50         return other instanceof Employee;
51     }
52
53
54     public int hashCode() {
55         int PRIME = true;
56         int result = 1;
57         Object $id = this.id;
58         int result = result * 59 + ($id == null ? 43 : $id.hashCode());
59         Object $name = this.name;
60         result = result * 59 + ($name == null ? 43 : $name.hashCode());
61         return result;
62     }
63
64 }
```

@NonNull

该注解需应用在方法或构造器的参数上或属性上，用来判断参数的合法性，默认抛出 `NullPointerException` 异常

```
import com.learning.domain.Employee;
import lombok.NonNull;

public class NonNullExample {
    private String name;
    public NonNullExample @NonNull Employee employee){
        this.name = employee.getName();
    }
}
```

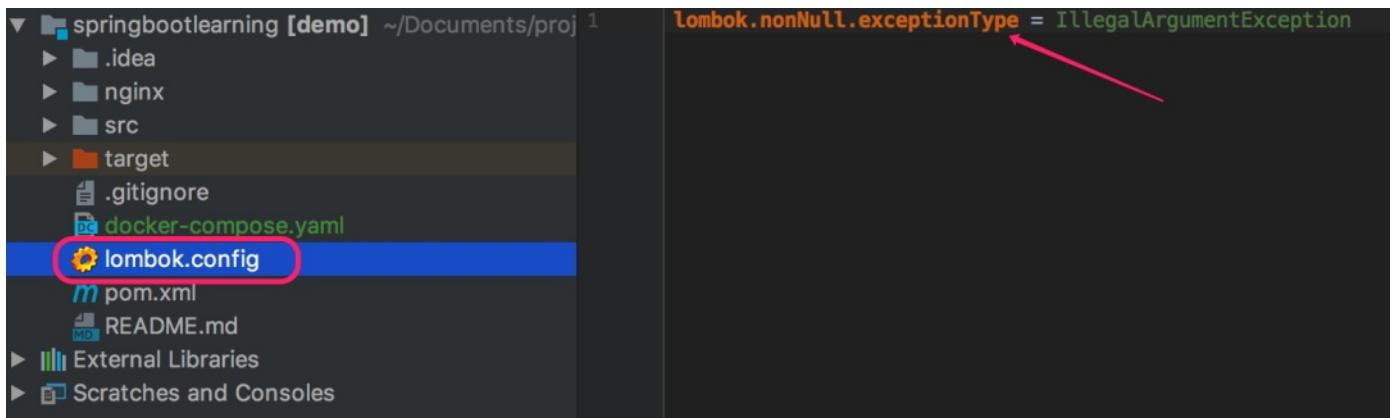
查看NonNullExample.class文件，会为我们抛出空指针异常，如下图：

```
import ...

public class NonNullExample {
    private String name;

    public NonNullExample(@NonNull Employee employee) {
        if (employee == null) {
            throw new NullPointerException("employee");
        } else {
            this.name = employee.getName();
        }
    }
}
```

当然我们可以通过指定异常类型抛出其他异常，lombok.nonNull.exceptionType = [NullPointerException |
IllegalArgumentException]，为实现此功能我们需要在项目的根目录新建lombok.config文件：



重新编译NonNullExample类，已经为我们抛出非法参数异常：

```
package com.learning.service;

import ...

public class NonNullExample {
    private String name;

    public NonNullExample(@NonNull Employee employee) {
        if (employee == null) {
            throw new IllegalArgumentException("employee is null");
        } else {
            this.name = employee.getName();
        }
    }
}
```

@NoArgsConstructor, @RequiredArgsConstructor, @AllArgsConstructor

以上三个注解分别为我们生成无参构造器，指定参数构造器和包含所有参数的构造器，默认情况下，@RequiredArgsConstructor, @AllArgsConstructor 生成的构造器会对所有标记 @NonNull 的属性做非空校验。

无参构造器很好理解，我们主要看看后两种，先看 @RequiredArgsConstructor

The screenshot shows the Employee.java file in an IDE. The code uses Lombok annotations to generate constructor logic. The annotations highlighted are:

- `@NoArgsConstructor`: Located at the top of the class definition.
- `@RequiredArgsConstructor`: Located on the first constructor definition.
- `@NonNull`: Used as a type annotation on the `name` field.
- `staticName = "generateEmployee"`: A static variable assignment within the constructor definition.

The code also includes standard Java annotations like `/**`, `* ... *`, `@author`, and `@date`.

```
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

/**
 * 员工, 用于说明lombok
 *
 * @author fraser
 * @date 2019/11/24 8:56 PM
 */

@RequiredArgsConstructor staticName = "generateEmployee"
public class Employee {

    final private Long id;

    @NonNull private String name;

    private static Integer age;

    private String password;
}
```

从上图中我们可以看出，`@RequiredArgsConstructor` 注解生成有参数构造器时只会包含有 `final` 和 `@NonNull` 标识的 field，同时我们可以指定 `staticName` 通过生成静态方法来构造对象

查看Employee.class文件

```
public class Employee {  
    private final Long id;  
    @NotNull  
    private String name;  
    private static Integer age;  
    private String password;  
  
    private Employee(Long id, @NotNull String name) {  
        if (name == null) {  
            throw new NullPointerException("name is marked non-null but is null");  
        } else {  
            this.id = id;  
            this.name = name;  
        }  
    }  
  
    public static Employee generateEmployee(Long id, @NotNull String name) { return new Employee(id, name); }  
}
```

当我们把 staticName 属性去掉我们来看遍以后的文件:

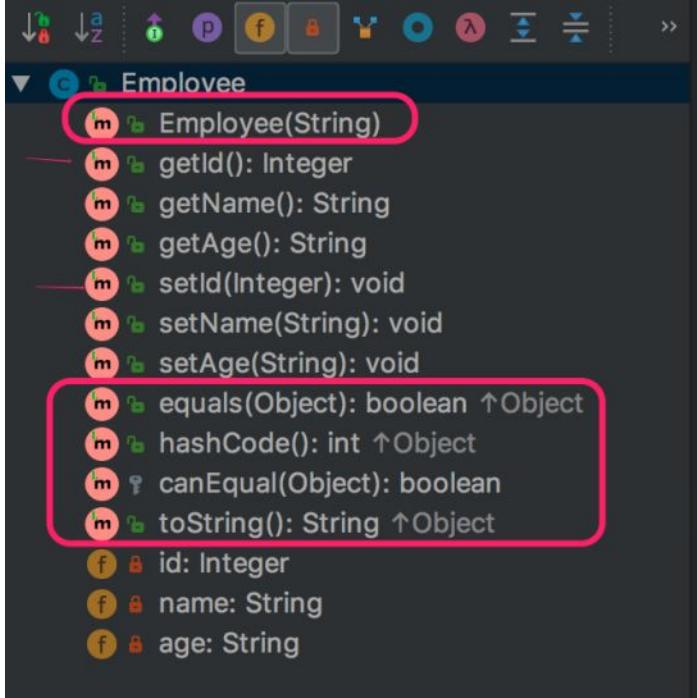
```
public class Employee {  
    private final Long id;  
    @NotNull  
    private String name;  
    private static Integer age;  
    private String password;  
  
    public Employee(Long id, @NotNull String name) {  
        if (name == null) {  
            throw new NullPointerException("name is marked non-null but is null");  
        } else {  
            this.id = id;  
            this.name = name;  
        } }  
}
```

相信你已经注意到细节

@AllArgsConstructor 就更简单了, 请大家自行查看吧

@Data

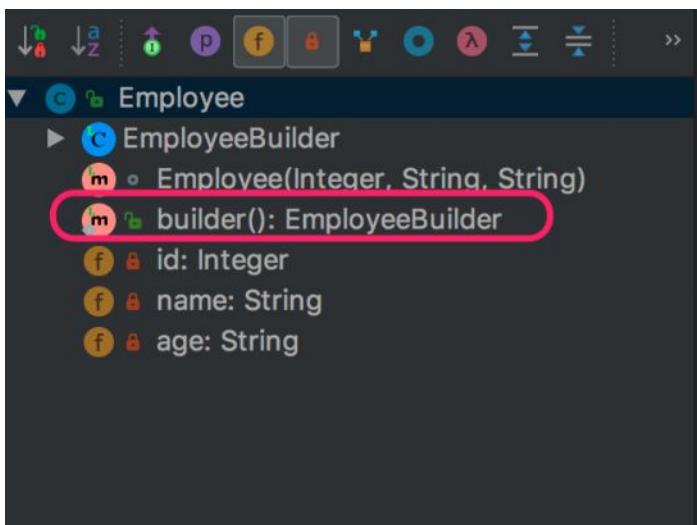
介绍了以上的注解, 再来介绍 @Data 就非常容易懂了, @Data 注解应用在类上, 是 @ToString, @EqualsAndHashCode, @Getter / @Setter 和 @RequiredArgsConstructor 合力的体现, 如下图:



```
1 package com.learning.domain;
2
3 import lombok.*;
4
5 @Data
6 public class Employee {
7
8     private Integer id;
9     @NotNull
10    private String name;
11
12    private String age;
13
14 }
15
16 }
```

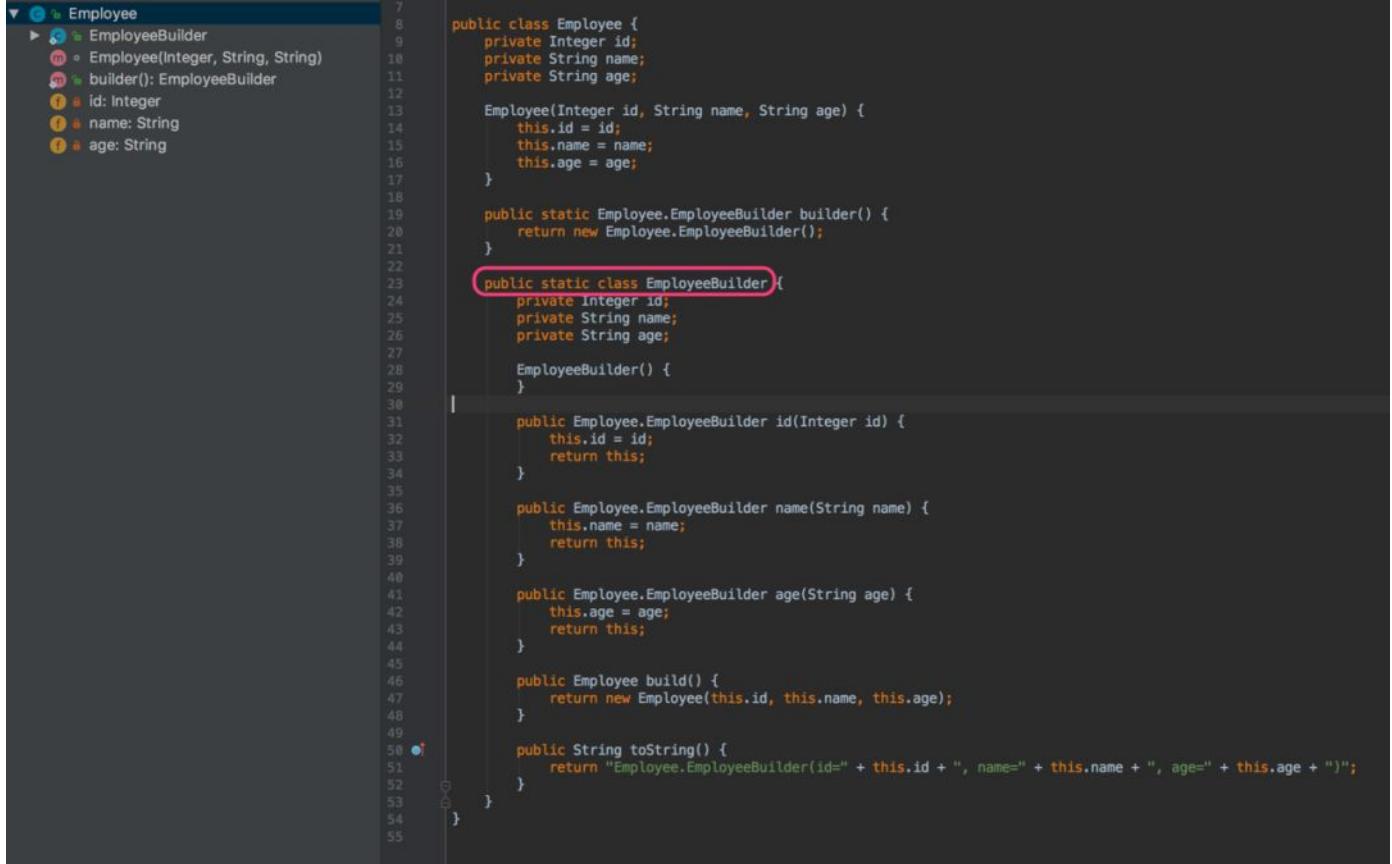
@Builder

函数式编程或者说流式的操作越来越流行，应用在大多数语言中，让程序更具更简介，可读性更高，编写更连贯，`@Builder`就带来了这个功能，生成一系列的builder API，该注解也需要应用在类上，看下面的例子就会更加清晰明了。



```
1 package com.learning.domain;
2
3 import lombok.*;
4
5 @Builder
6 public class Employee {
7
8     private Integer id;
9     private String name;
10    private String age;
11
12 }
13
14 }
```

编译后的Employee.class文件如下：



```
Employee
EmployeeBuilder
Employee(Integer, String, String)
builder(): EmployeeBuilder
id: Integer
name: String
age: String

public class Employee {
    private Integer id;
    private String name;
    private String age;

    Employee(Integer id, String name, String age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }

    public static Employee.EmployeeBuilder builder() {
        return new Employee.EmployeeBuilder();
    }

    public static class EmployeeBuilder {
        private Integer id;
        private String name;
        private String age;

        EmployeeBuilder() {
        }

        public Employee.EmployeeBuilder id(Integer id) {
            this.id = id;
            return this;
        }

        public Employee.EmployeeBuilder name(String name) {
            this.name = name;
            return this;
        }

        public Employee.EmployeeBuilder age(String age) {
            this.age = age;
            return this;
        }

        public Employee build() {
            return new Employee(this.id, this.name, this.age);
        }

        public String toString() {
            return "Employee.EmployeeBuilder(id=" + this.id + ", name=" + this.name + ", age=" + this.age + ")";
        }
    }
}
```

妈妈再也不用担心我 set 值那么麻烦了，流式操作搞定：

```
public Employee test(){
    return Employee.builder().name("Fraser").age("26").build();
}
```

@Log

该注解需要应用到类上，在编写服务层，需要添加一些日志，以便定位问题，我们通常会定义一个静态常量Logger，然后应用到我们想日志的地方，现在一个注解就可以实现：



```
@Builder
@Log
public class Employee {

    private Integer id;
    private String name;

    private String age;

    public Employee test(){
        log.info( msg: "this is test method");
        return Employee.builder().name("Fraser").age("26").build();
    }
}
```

查看class文件，和我们预想的一样：

```
import java.util.logging.Logger;

public class Employee {
    private static final Logger log = Logger.getLogger(Employee.class.getName());
    private Integer id;
    private String name;
    private String age;

    public Employee test() {
        log.info( msg: "this is test method");
        return builder().name("Fraser").age("26").build();
    }
}
```

Log有很多变种，CommonLog，Log4j，Log4j2，Slf4j等，lombok依旧良好的通过变种注解做良好的支持：

```
@CommonsLog
Creates
private static final org.apache.commons.logging.Log log = org.apache.commons.logging.LogFactory.getLog(LogExample.class);
@JBossLog
Creates
private static final org.jboss.logging.Logger log = org.jboss.logging.Logger.getLogger(LogExample.class);
@Log
Creates
private static final java.util.logging.Logger log = java.util.logging.Logger.getLogger(LogExample.class.getName());
@Log4j
Creates
private static final org.apache.log4j.Logger log = org.apache.log4j.Logger.getLogger(LogExample.class);
@Log4j2
Creates
private static final org.apache.logging.log4j.Logger log = org.apache.logging.log4j LogManager.getLogger(LogExample.class);
@Slf4j
Creates
private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExample.class);
@XSlf4j
Creates
private static final org.slf4j.ext.XLogger log = org.slf4j.ext.XLoggerFactory.getXLogger(LogExample.class);
```

我实际使用的是 @Slf4j 注解

val

熟悉 Javascript 的同学都知道，var 可以定义任何类型的变量，而在 java 的实现中我们需要指定具体变量的类型，而 val 让我们摆脱指定，编译之后就精准匹配上类型，默认是 final 类型，就像 java8 的函数式表达式，() -> System.out.println("hello lombok"); 就可以解析到 Runnable 函数式接口。

```
import lombok.*;  
  
import java.util.HashMap;  
  
public class Employee {  
  
    private Integer id;  
    private String name;  
  
    private String age;  
  
    public void test(){  
        val map = new HashMap<String, String>();  
        map.put("name", "Fraser");  
        map.put("age", "26");  
    }  
  
}
```

查看解析后的class文件：

```
import java.util.HashMap;  
  
public class Employee {  
    private Integer id;  
    private String name;  
    private String age;  
  
    public Employee() {  
    }  
  
    public void test() {  
        HashMap<String, String> map = new HashMap();  
        map.put("name", "Fraser");  
        map.put("age", "26");  
    }  
}
```

@Cleanup

当我们对流进行操作，我们通常需要调用 close 方法来关闭或结束某资源，而 @Cleanup 注解可以帮助我们调用 close 方法，并且放到 try/finally 处理块中，如下图：

```
import lombok.Cleanup;

import java.io.*;

public class NonNullExample {

    public static void main(String[] args) throws IOException {
        @Cleanup InputStream in = new FileInputStream(args[0]);
        @Cleanup OutputStream out = new FileOutputStream(args[1]);
        byte[] b = new byte[10000];
        while (true) {
            int r = in.read(b);
            if (r == -1) break;
            out.write(b, off: 0, r);
        }
    }
}
```

编译后的class文件如下，我们发现被try/finally包围处理，并调用了流的close方法

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Collections;

public class NonNullExample {
    public NonNullExample() {
    }

    public static void main(String[] args) throws IOException {
        FileInputStream in = new FileInputStream(args[0]);
        try {
            FileOutputStream out = new FileOutputStream(args[1]);
            try {
                byte[] b = new byte[10000];

                while(true) {
                    int r = in.read(b);
                    if (r == -1) {
                        return;
                    }

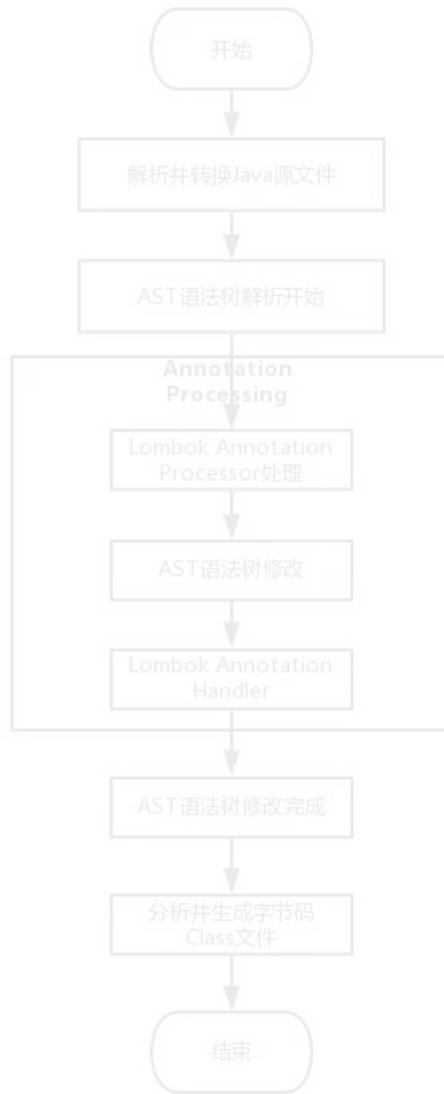
                    out.write(b, off: 0, r);
                }
            } finally {
                if (Collections.singletonList(out).get(0) != null) {
                    out.close();
                }
            }
        } finally {
            if (Collections.singletonList(in).get(0) != null) {
                in.close();
            }
        }
    }
}
```

其实在 JDK1.7 之后就有了 try-with-resource，不用我们显式的关闭流，这个请大家自行看吧

总结

Lombok的基本操作流程是这样的：

1. 定义编译期的注解
2. 利用JSR269 api(Pluggable Annotation Processing API)创建编译期的注解处理器
3. 利用tools.jar的javac api处理AST(抽象语法树)
4. 将功能注册进jar包



Lombok 当然还有很多注解，我推荐使用以上就足够了，这个工具是带来便利的，而不能被其捆绑，“弱水三千只取一瓢饮，代码千万需抓重点看”，Lombok 能让我更加专注有效代码排除意义微小的障眼代码 (get, set等)，另外Lombok生成的代码还能像使用工具类一样方便 (@Builder)。

更多内容请查看官网：<https://www.projectlombok.org/>

灵魂追问

1. 为什么只有一个整体 @EqualsAndHashCode 注解？

而不是 @Equals 和 @HashCode？

这涉及到一个规范哦

2. 如果把三种构造器方式同时应用又加上了 @Builder 注解，会发生什么？

欢迎评论！

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. [Dubbo 爆出严重漏洞！](#)
2. [Spring Boot+Redis 分布式锁：模拟抢单](#)
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

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！

【Java 8】一个非常实用的特性，很多人没用过

Java后端 2月27日



Java8中有两个非常有名的改进，一个是Lambda表达式，一个是Stream。如果我们了解过函数式编程的话，都知道Stream真正把函数式编程的风格引入到了java中。这篇文章由简入繁逐步介绍Stream。

一、Stream是什么

从名字来看，Stream就是一个流，他的主要作用就是对集合数据进行查找过滤等操作。有点类似于SQL的数据库操作。一句话来解释就是一种高效且易用的数据处理方式。大数据领域也有一个Steam实时流计算框架，不过和这个可不一样。别搞混了。

举个例子吧，比如说有一个集合Student数据，我们要删选出学生名字为“张三”的学生，或者是找出所有年龄大于18岁的所有学生。此时我们就可以直接使用Stream来筛选。当然了这只是给出了其中一个例子。Stream还有很多其他的功能。

Stream和Collection的区别就是：Collection只是负责存储数据，不对数据做其他处理，主要是和内存打交道。但是Stream主要是负责计算数据的，主要是和CPU打交道。现在明白了吧。

往期关于Java小技巧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、Stream语法讲解

Stream执行流程很简单，主要有三个，首先创建一个Stream，然后使用Stream操作数据，最后终止Stream。有点类似于Stream的生命周期。下面我们根据其流程来一个一个讲解。

1、前提准备

首先我们创建一个Student类，以后我们每次都是操作这个类

```
1 public class Student {  
2     private Integer id;  
3     private String name;  
4     private Integer age;  
5     private Double score;  
6     public Student() {  
7     }  
8     public Student(Integer id, String name, Integer age, Double score) {  
9         this.id = id;  
10        this.name = name;  
11        this.age = age;  
12        this.score = score;  
13    }  
14    //getter和setter方法  
15    //toString方法  
16 }
```

然后下面我们再创建一个StudentData类，用于获取其数据

```
1  public class StudentData {
2      public static List<Student> getStudents(){
3          List<Student> list = new ArrayList<>();
4          list.add(new Student(1,"刘备",18,90.4));
5          list.add(new Student(2,"张飞",19,87.4));
6          list.add(new Student(3,"关羽",21,67.4));
7          list.add(new Student(4,"赵云",15,89.4));
8          list.add(new Student(5,"马超",16,91.4));
9          list.add(new Student(6,"曹操",19,83.4));
10         list.add(new Student(7,"荀彧",24,78.4));
11         list.add(new Student(8,"孙权",26,79.4));
12         list.add(new Student(9,"鲁肃",21,93.4));
13         return list;
14     }
15 }
```

我们只需要把方法变成static类型的就可以了。

2、创建一个Stream

方式一：通过一个集合创建Stream

```
1  @Test
2  public void test1(){
3      List<Student> studentList = StudentData.getStudents();
4      //第一种：返回一个顺序流
5      Stream<Student> stream = studentList.stream();
6      //第二种：返回一个并行流
7      Stream<Student> stream2 = studentList.parallelStream();
8  }
```

方式二：通过一个数组创建Stream

```
1  @Test
2  public void test2(){
3      //获取一个整形Stream
4      int[] arr = new int[]{1,2,34,4,65,7,87,};
5      IntStream intStream = Arrays.stream(arr);
6      //获取一个Student对象Stream
7      Student[] students = StudentData.getArrStudents();
8      Stream<Student> stream = Arrays.stream(students);
9
10 }
```

方式三：通过Stream.of

```
1  @Test
2  public void test3(){
3      Stream<Integer> integerStream = Stream.of(1, 2, 3, 5, 6, 7, 8);
4      Stream<String> stringStream = Stream.of("1", "2", "3", "4", "5");
5      Stream<Student> studentStream = Stream.of(
6          new Student(1, "刘备", 18, 90.4),
7          new Student(2, "张飞", 19, 87.4),
8          new Student(3, "关羽", 21, 67.4));
9
10 }
```

方式四：创建一个无限流

```
1  @Test
2  public void test4(){
3      //每隔5个数取一个，从0开始，此时就会无限循环
4      Stream.iterate(0,t->t+5).forEach(System.out::println);
5      //每隔5个数取一个，从0开始，只取前5个数
6      Stream.iterate(0,t->t+5).limit(5).forEach(System.out::println);
7      //取出一个随机数
8      Stream.generate(Math::random).limit(5).forEach(System.out::println);
9  }
```

3、使用Stream操作数据

操作1：筛选和切片

```
1  @Test
2  public void test1(){
3      List<Student> list = StudentData.getStudents();
4      // (1) 过滤：过滤出所有年龄大于20岁的同学
5      list.stream().filter(item->item.getAge()>20).forEach(System.out::println);
6      // (2) 截断流：筛选出前3条数据
7      list.stream().limit(3).forEach(System.out::println);
8      // (3) 跳过元素：跳过前5个元素
9      list.stream().skip(5).forEach(System.out::println);
10     // (4) 过滤重复数据：
11     list.stream().distinct().forEach(System.out::println);
12 }
```

操作2：映射

```
1  @Test
2  public void test2(){
3      // (1) map操作
4      List<String> list = Arrays.asList("java","python","go");
5      Stream<String> stream = list.stream();
6      //此时每一个小写字母都有一个大写的映射
7      stream.map(str -> str.toUpperCase()).forEach(System.out::println);
8      //筛选出所有的年龄，再过滤出所有大于20的年龄有哪些
9      List<Student> studentList = StudentData.getStudents();
10     Stream<Student> stream1 = studentList.stream();
11     Stream<Integer> ageStream = stream1.map(Student::getAge);
12     ageStream.filter(age->age>20).forEach(System.out::println);
13     // (2) floatMap：将流中的每一个值换成另外一个流
14 }
```

操作3：排序

```
1  public void test3(){
2      // (1) 自然排序
3      List<Integer> list = Arrays.asList(4,3,7,9,12,8,10,23,2);
4      Stream<Integer> stream = list.stream();
5      stream.sorted().forEach(System.out::println);
6      // (2) 对象排序：对对象类可以先实现comparable接口，或者是直接指定
7      // 第一种：先实现comparable接口
8      List<Student> studentList = StudentData.getStudents();
9      studentList.stream().sorted().forEach(System.out::println);
10     // 第二种：直接指定comparable
11     List<Student> studentList1 = StudentData.getStudents();
12     studentList1.stream()
13         .sorted((e1,e2)-> Integer.compare(e1.getAge(),e2.getAge()))
14         .forEach(System.out::println);
15 }
```

4、终止Stream

操作1：匹配和查找

```
1  public void test1(){
2      List<Student> list = StudentData.getStudents();
3      // (1) 判断所有的学生年龄是否都大于20岁
4      boolean allMatch = list.stream().allMatch(item -> item.getAge() > 20);
5      // (2) 判断是否存在学生的年龄大于20岁
6      boolean anyMatch = list.stream().anyMatch(item -> item.getAge() > 20);
7      // (3) 判断是否存在学生叫曹操
8      boolean noneMatch = list.stream().noneMatch(item -> item.getName().equals("曹操"));
9      // (4) 查找第一个学生
10     Optional<Student> first = list.stream().findFirst();
11     // (5) 查找所有的学生数量
12     long count = list.stream().count();
13     long count1 = list.stream().filter(item -> item.getScore() > 90.0).count();
14     // (6) 查找当前流中的元素
15     Optional<Student> any = list.stream().findAny();
16     // (7) 查找学生成绩最高的分(如果Student实现了comparable接口的话, 可直接比较)
17     Stream<Double> doubleStream = list.stream().map(item -> item.getScore());
18     doubleStream.max(Double::compare);
19     // (8) 查找学生成绩最低的分
20 }
```

操作2：归约

```
1  public void test2(){
2      // (1) 计算数的总和
3      List<Integer> list = Arrays.asList(1,2,3,4,5);
4      list.stream().reduce(0, Integer::sum);
5      // (3) 计算学生总分
6      List<Student> studentList = StudentData.getStudents();
7      Stream<Double> doubleStream = studentList.stream().map(Student::getScore);
8      doubleStream.reduce(Double::sum);
9 }
```

操作3：收集

```
1  public void test3(){
2      List<Student> studentList = StudentData.getStudents();
3      // 返回一个list
4      List<Student> listStream = studentList.stream()
5          .filter(e -> e.getAge() > 18)
6          .collect(Collectors.toList());
7      // 返回一个Set
8      Set<Student> setStream = studentList.stream()
9          .filter(e -> e.getAge() > 18)
10         .collect(Collectors.toSet());
11     // 返回其他的类型
12 }
```

stream基本的语法就是这样，你会发现Stream就像是一个工具一样，可以帮我们分析处理数据，极其的好用，但是目前还不知道其效率如何。根据网上一位大佬的内存时间分析，其实在数据量比较庞大的时候，Stream可以为我们节省大量的时间，数据量小的时候并不明显。

转自 | Java的架构师技术栈

原文 | toutiao.com/a6770603570233344524

- 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！

【Java】这35个Java代码优化细节，你用了吗？

Java后端 2月28日



链接：<https://www.jianshu.com/p/6e472304b5ac>

前言

代码优化，一个很重要的课题。可能有些人觉得没用，一些细小的地方有什么好修改的，改与不改对于代码的运行效率有什么影响呢？这个问题我是这么考虑的，就像大海里面的鲸鱼一样，它吃一条小虾米有用吗？没用，但是，吃的小虾米一多之后，鲸鱼就被喂饱了。

代码优化也是一样，如果项目着眼于尽快无BUG上线，那么此时可以抓大放小，代码的细节可以不精打细磨；但是如果有足够的时间开发、维护代码，这时候就必须考虑每个可以优化的细节了，一个一个细小的优化点累积起来，对于代码的运行效率绝对是有提升的。

代码优化的目标是：

减小代码的体积

提高代码运行的效率

代码优化细节

1、尽量指定类、方法的final修饰符

带有final修饰符的类是不可派生的。在Java核心API中，有许多应用final的例子，例如java.lang.String，整个类都是final的。为类指定final修饰符可以让类不可以被继承，为方法指定final修饰符可以让方法不可以被重写。如果指定了一个类为final，则该类所有的方法都是final的。Java编译器会寻找机会内联所有的final方法，内联对于提升Java运行效率作用重大，具体参见Java运行期优化。此举能够使性能平均提高50%。

2、尽量重用对象

特别是String对象的使用，出现字符串连接时应该使用StringBuilder/StringBuffer代替。由于Java虚拟机不仅要花时间生成对象，以后可能还需要花时间对这些对象进行垃圾回收和处理，因此，生成过多的对象将会给程序的性能带来很大的影响。

3、尽可能使用局部变量

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈中速度较快，其他变量，如静态变量、实例变量等，都在堆中创建，速度较慢。另外，栈中创建的变量，随着方法的运行结束，这些内容就没了，不需要额外的垃圾回收。

4、及时关闭流

Java编程过程中，进行数据库连接、I/O流操作时务必小心，在使用完毕后，及时关闭以释放资源。因为对这些对象的操作会造成系统大的开销，稍有不慎，将会导致严重的后果。

5、尽量减少对变量的重复计算

明确一个概念，对方法的调用，即使方法中只有一句语句，也是有消耗的，包括创建栈帧、调用方法时保护现场、调用方法完毕时恢复现场等。所以例如下面的操作：

```
for (int i = 0; i < list.size(); i++)
{...}
```

建议替换为：

```
for (int i = 0, int length = list.size(); i < length; i++)  
{...}
```

这样，在list.size很大的时候，就减少了很多的消耗

6、尽量采用懒加载的策略，即在需要的时候才创建*

例如：

```
String str = "aaa"; if (i == 1)  
{  
    list.add(str);  
}
```

建议替换为：

```
if (i == 1)  
{  
    String str = "aaa";  
    list.add(str);  
}
```

7、慎用异常

异常对性能不利。抛出异常首先要创建一个新的对象，`Throwable`接口的构造函数调用名为`fillInStackTrace`的本地同步方法，`fillInStackTrace`方法检查堆栈，收集调用跟踪信息。只要有异常被抛出，Java虚拟机就必须调整调用堆栈，因为在处理过程中创建了一个新的对象。异常只能用于错误处理，不应该用来控制程序流程。

[图片上传失败...(image-b49a00-1567085107360)]

8、不要在循环中使用try…catch…，应该把其放在最外层

除非不得已。如果毫无理由地这么写了，只要你的领导资深一点、有强迫症一点，八成就要骂你为什么写出这种垃圾代码来了。

9、如果能估计到待添加的内容长度，为底层以数组方式实现的集合、工具类指定初始长度

比如`ArrayList`、`LinkedList`、`StringBuilder`、`StringBuffer`、`HashMap`、`HashSet`等等，以`StringBuilder`为例：

(1) `StringBuilder` // 默认分配16个字符的空间

(2) `StringBuilder(int size)` // 默认分配size个字符的空间

(3) `StringBuilder(String str)` // 默认分配16个字符+`str.length`个字符空间

可以通过类（这里指的不仅仅是上面的`StringBuilder`）的来设定它的初始化容量，这样可以明显地提升性能。比如`StringBuilder`吧，`length`表示当前的`StringBuilder`能保持的字符数量。因为当`StringBuilder`达到最大容量的时候，它会将自身容量增加到当前的2倍再加2，无论何时只要`StringBuilder`达到它的最大容量，它就不得不创建一个新的字符数组然后将旧的字符数组内容拷贝到新字符数组中——这是十分耗费性能的一个操作。试想，如果能

预估到字符数组中大概要存放5000个字符而不指定长度，最接近5000的2次幂是4096，每次扩容加的2不管，那么：

- (1) 在4096 的基础上，再申请8194个大小的字符数组，加起来相当于一次申请了12290个大小的字符数组，如果一开始能指定5000个大小的字符数组，就节省了一倍以上的空间；
- (2) 把原来的4096个字符拷贝到新的的字符数组中去。

这样，既浪费内存空间又降低代码运行效率。所以，给底层以数组实现的集合、工具类设置一个合理的初始化容量是错不了的，这会带来立竿见影的效果。但是，注意，像HashMap这种是以数组+链表实现的集合，别把初始大小和你估计的大小设置得一样，因为一个table上只连接一个对象的可能性几乎为0。初始大小建议设置为2的N次幂，如果能估计到有2000个元素，设置成new HashMap(128)、new HashMap(256)都可以。

10、当复制大量数据时，使用System.arraycopy命令

11、乘法和除法使用移位操作

例如：

```
for (val = 0; val < 100000; val += 5) {  
    a = val * 8;  
    b = val / 2;  
}
```

用移位操作可以极大地提高性能，因为在计算机底层，对位的操作是最方便、最快的，因此建议修改为：

```
for (val = 0; val < 100000; val += 5) {  
    a = val << 3;  
    b = val >> 1;  
}
```

移位操作虽然快，但是可能会使代码不太好理解，因此最好加上相应的注释。

12、循环内不要不断创建对象引用

例如：

```
for (int i = 1; i <= count; i++) {  
    Object obj = new Object();  
}
```

这种做法会导致内存中有count份Object对象引用存在，count很大的话，就耗费内存了，建议为改为：

```
Object obj = null;  
for (int i = 0; i <= count; i++) {  
    obj = new Object();  
}
```

这样的话，内存中只有一份Object对象引用，每次new Object的时候，Object对象引用指向不同的Object罢了，但是内存中只有一份，这样就大大节省了内存空间了。

13、基于效率和类型检查的考虑，应该尽可能使用array，无法确定数组大小时才使用ArrayList

14、尽量使用HashMap、ArrayList、StringBuilder，除非线程安全需要，否则不推荐使用Hashtable、Vector、StringBuffer，后三者由于使用同步机制而导致了性能开销

15、不要将数组声明为public static final

因为这毫无意义，这样只是定义了引用为static final，数组的内容还是可以随意改变的，将数组声明为public更是

一个安全漏洞，这意味着这个数组可以被外部类所改变。

16、尽量在合适的场合使用单例

使用单例可以减轻加载的负担、缩短加载的时间、提高加载的效率，但并不是所有地方都适用于单例，简单来说，单例主要适用于以下三个方面：

- (1) 控制资源的使用，通过线程同步来控制资源的并发访问
- (2) 控制实例的产生，以达到节约资源的目的
- (3) 控制数据的共享，在不建立直接关联的条件下，让多个不相关的进程或线程之间实现通信

17、尽量避免随意使用静态变量

要知道，当某个对象被定义为static的变量所引用，那么gc通常是不会回收这个对象所占有的堆内存的，如：

```
public class A {  
    private static B b = new B();  
}
```

此时静态变量b的生命周期与A类相同，如果A类不被卸载，那么引用B指向的B对象会常驻内存，直到程序终止

18、及时清除不再需要的会话

为了清除不再活动的会话，许多应用服务器都有默认的会话超时时间，一般为30分钟。当应用服务器需要保存更多的会话时，如果内存不足，那么操作系统会把部分数据转移到磁盘，应用服务器也可能根据MRU（最近最频繁使用）算法把部分不活跃的会话转储到磁盘，甚至可能抛出内存不足的异常。如果会话要被转储到磁盘，那么必须要先被序列化，在大规模集群中，对对象进行序列化的代价是很昂贵的。因此，当会话不再需要时，应当及时调用HttpSession的invalidate方法清除会话。

19、实现RandomAccess接口的集合比如ArrayList，应当使用最普通的for循环而不是foreach循环来遍历

这是JDK推荐给用户的。JDK API对于RandomAccess接口的解释是：实现RandomAccess接口用来表明其支持快速随机访问，此接口的主要目的是允许一般的算法更改其行为，从而将其应用到随机或连续访问列表时能提供良好的性能。实际经验表明，实现RandomAccess接口的类实例，假如是随机访问的，使用普通for循环效率将高于使用foreach循环；反过来，如果是顺序访问的，则使用Iterator会效率更高。可以使用类似如下的代码作判断：

```
if (list instanceof RandomAccess) {  
    for (int i = 0; i < list.size(); i++) {}  
} else {  
    Iterator<?> iterator = list.iterator();  
    while (iterator.hasNext()) {  
        iterator.next()  
    }  
} else {  
    Iterator<?> iterator = list.iterator();  
    while (iterator.hasNext()) {  
        iterator.next()  
    }  
}
```

foreach循环的底层实现原理就是迭代器Iterator，参见Java语法糖1：可变长度参数以及foreach循环原理。所以“以后半句”反过来，如果是顺序访问的，则使用Iterator会效率更高”的意思就是顺序访问的那些类实例，使用foreach循环去遍历。

20、使用同步代码块替代同步方法

这点在多线程模块中的synchronized锁方法块一文中已经讲得很清楚了，除非能确定一整个方法都是需要进行同步的，否则尽量使用同步代码块，避免对那些不需要进行同步的代码也进行了同步，影响了代码执行效率。

21、将常量声明为static final，并以大写命名

这样在编译期间就可以把这些内容放入常量池中，避免运行期间计算生成常量的值。另外，将常量的名字以大写命名也可以方便区分出常量与变量。

22、不要创建一些不使用的对象，不要导入一些不使用的类

这毫无意义，如果代码中出现”The value of the local variable i is not used”、“The import java.util is never used”，那么请删除这些无用的内容

23、程序运行过程中避免使用反射

关于，请参见反射。反射是Java提供给用户一个很强大的功能，功能强大往往意味着效率不高。不建议在程序运行过程中使用尤其是频繁使用反射机制，特别是Method的invoke方法，如果确实有必要，一种建议性的做法是将那些需要通过反射加载的类在项目启动的时候通过反射实例化出一个对象并放入内存—用户只关心和对端交互的时候获取最快的响应速度，并不关心对端的项目启动花多久时间。

24、使用数据库连接池和线程池

这两个池都是用于重用对象的，前者可以避免频繁地打开和关闭连接，后者可以避免频繁地创建和销毁线程

25、使用带缓冲的输入输出流进行IO操作

带缓冲的输入输出流，即BufferedReader、BufferedWriter、BufferedInputStream、BufferedOutputStream，这可以极大地提升IO效率

26、顺序插入和随机访问比较多的场景使用ArrayList，元素删除和中间插入比较多的场景使用LinkedList这个，理解ArrayList和LinkedList的原理就知道了

27、不要让public方法中有太多的形参

public方法即对外提供的方法，如果给这些方法太多形参的话主要有两点坏处：

- 1、违反了面向对象的编程思想，Java讲求一切都是对象，太多的形参，和面向对象的编程思想并不契合
- 2、参数太多势必导致方法调用的出错概率增加

至于这个“太多”指的是多少个，3、4个吧。比如我们用JDBC写一个insertStudentInfo方法，有10个学生信息字段要插入Student表中，可以把这10个参数封装在一个实体类中，作为insert方法的形参。

28、字符串变量和字符串常量equals的时候将字符串常量写在前面

这是一个比较常见的小技巧了，如果有以下代码：

```
String str = "123";
if (str.equals("123")) {...}
```

建议修改为：

```
String str = "123";
if ("123".equals(str))
{
...
}
```

这么做主要是可以避免空指针异常

29、请知道，在java中if (i == 1)和if (1 == i)是没有区别的，但从阅读习惯上讲，建议使用前者

平时有人问，“if (i == 1)”和“if (1 == i)”有没有区别，这就要从C/C++讲起。

在C/C++中，”if (i == 1)”判断条件成立，是以0与非0为基准的，0表示false，非0表示true，如果有这么一段代码：

```
int i = 2;
if (i == 1)
{
...
} else {
...
}
```

C/C++判断”i==1”不成立，所以以0表示，即false。但是如果：

```
int i = 2;
if (i = 1) { ... }
else { ... }
```

万一程序员一个不小心，把”if (i == 1)”写成”if (i = 1)”，这样就有问题了。在if之内将i赋值为1，if判断里面的内容非0，返回的就是true了，但是明明i为2，比较的值是1，应该返回的false。这种情况在C/C++的开发中是很可能发生的并且会导致一些难以理解的错误产生，所以，为了避免开发者在if语句中不正确的赋值操作，建议将if语句写为：

```
int i = 2;
if (1 == i) { ... }
else { ... }
```

这样，即使开发者不小心写成了”1 = i”，C/C++编译器也可以第一时间检查出来，因为我们可以对一个变量赋值i为1，但是不能对一个常量赋值1为i。

但是，在Java中，C/C++这种”if (i = 1)”的语法是不可能出现的，因为一旦写了这种语法，Java就会编译报错”Type mismatch: cannot convert from int to boolean”。但是，尽管Java的”if (i == 1)”和”if (1 == i)”在语义上没有任何区别，但是从阅读习惯上讲，建议使用前者会更好些。

30、不要对数组使用toString方法

看一下对数组使用toString打印出来的是什么：

```
public static void main(String[] args)
{
    int[] is = new int[] {1, 2, 3};
    System.out.println(is.toString());
}
```

结果是：

```
[I@18a992f
```

本意是想打印出数组内容，却有可能因为数组引用is为空而导致空指针异常。不过虽然对数组toString没有意义，但是对集合toString是可以打印出集合里面的内容的，因为集合的父类AbstractCollections重写了Object的

`toString`方法。

31、不要对超出范围的基本数据类型做向下强制转型

这绝不会得到想要的结果：

```
public static void main(String[] args)
{
    long l = 12345678901234L;
    int i = (int)l;
    System.out.println(i);
}
```

我们可能期望得到其中的某几位，但是结果却是：

1942892530

解释一下。Java中`long`是8个字节64位的，所以12345678901234在计算机中的表示应该是：

0000 0000 0000 0000 0000 1011 0011 1010 0111 0011 1100 1110 0010 1111 1111 0010

一个`int`型数据是4个字节32位的，从低位取出上面这串二进制数据的前32位是：

0111 0011 1100 1110 0010 1111 1111 0010

这串二进制表示为十进制1942892530，所以就是我们上面的控制台上输出的内容。从这个例子上还能顺便得到两个结论：

1、整型默认的数据类型是`int`, `long l = 12345678901234L`, 这个数字已经超出了`int`的范围了，所以最后有一个

`L`, 表示这是一个`long`型数。顺便，浮点型的默认类型是`double`, 所以定义`float`的时候要写成”`float f = 3.5f`”

2、接下来再写一句”`int ii = l + i;`”会报错，因为`long + int`是一个`long`, 不能赋值给`int`

32、公用的集合类中不使用的数据一定要及时remove掉

如果一个集合类是公用的（也就是说不是方法里面的属性），那么这个集合里面的元素是不会自动释放的，因为始终有引用指向它们。所以，如果公用集合里面的某些数据不使用而不去`remove`掉它们，那么将会造成这个公用集合不断增大，使得系统有内存泄露的隐患。

33、把一个基本数据类型转为字符串，`基本数据类型.toString`是最快的方式、`String.valueOf`次之、`数据+””`最慢

把一个基本数据类型转为一般有三种方式，我有一个`Integer`型数据`i`, 可以使用`i.toString`、`String.valueOf(i)`、`i+””`三种方式，三种方式的效率如何，看一个测试：

```
public static void main(String[] args)
{
int loopTime = 50000;
Integer i = 0; long startTime = System.currentTimeMillis(); for (int j = 0; j < loopTime; j++)
{
String str = String.valueOf(i);
}
System.out.println("String.valueOf(): " + (System.currentTimeMillis() - startTime) + "ms");
startTime = System.currentTimeMillis(); for (int j = 0; j < loopTime; j++)
{
String str = i.toString();
}
System.out.println("Integer.toString(): " + (System.currentTimeMillis() - startTime) + "ms");
startTime = System.currentTimeMillis(); for (int j = 0; j < loopTime; j++)
{
String str = i + "";
}
System.out.println("i + \"\": " + (System.currentTimeMillis() - startTime) + "ms");
}
```

运行结果为：

```
String.valueOf(): 11ms Integer.toString(): 5ms i + "": 25ms
```

所以以后遇到把一个基本数据类型转为String的时候，优先考虑使用toString方法。至于为什么，很简单：

1、String.valueOf方法底层调用了Integer.toString方法，但是会在调用前做空判断

2、Integer.toString方法就不说了，直接调用了

3、i + “”底层使用了StringBuilder实现，先用append方法拼接，再用toString方法获取字符串

三者对比下来，明显是2最快、1次之、3最慢

34、使用最有效率的方式去遍历Map

遍历Map的方式有很多，通常场景下我们需要的是遍历Map中的Key和Value，那么推荐使用的、效率最高的方式是：

```
public static void main(String[] args)
{
HashMap<String, String> hm = new HashMap<String, String>();
hm.put("111", "222");Set<Map.Entry<String, String>> entrySet = hm.entrySet();
Iterator<Map.Entry<String, String>> iter = entrySet.iterator(); while (iter.hasNext())
{
Map.Entry<String, String> entry = iter.next();
System.out.println(entry.getKey() + "\t" + entry.getValue());
}
}
```

如果你只是想遍历一下这个Map的key值，那用“Set keySet = hm.keySet;”会比较合适一些

35、对资源的close建议分开操作

意思是，比如我有这么一段代码：

```
try{
    XXX.close();
    YYY.close();
} catch (Exception e)
{...}
```

建议修改为：

```
try{
    XXX.close();
} catch (Exception e) {
    ...
}

try{
    YYY.close();
} catch (Exception e) {
    ...
}
```

虽然有些麻烦，却能避免资源泄露。我想，如果没有修改过的代码，万一`XXX.close`抛异常了，那么就进入了`catch`块中了，`YYY.close`不会执行，`YYY`这块资源就不会回收了，一直占用着，这样的代码一多，是可能引起资源句柄泄露的。而改为上面的写法之后，就保证了无论如何`XXX`和`YYY`都会被`close`掉。

最后

欢迎大家一起交流，喜欢文章记得关注我点个赞哟，感谢支持！

作者：程序员追风，链接：<https://www.jianshu.com/p/6e472304b5ac>。

- 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



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！

【枚举】用好 Java 中的枚举，真的没有那么简单！

Java后端 2月28日

“

最近重看 Java 枚举，看到这篇觉得还不错的文章，于是简单翻译和完善了一些内容，分享给大家，希望你们也能有所收获。另外，不要忘了文末还有补充哦！ps：这里发一篇枚举的文章，也是因为后面要发一篇非常实用的关于 SpringBoot 全局异常处理的比较好的实践里面就用到了枚举。这篇文章由 JavaGuide 翻译，公众号：JavaGuide，原文地址：<https://www.baeldung.com/a-guide-to-java-enums>。转载请注明上面这段文字。

”

1.概览

在本文中，我们将看到什么是 Java 枚举，它们解决了哪些问题以及如何在实践中使用 Java 枚举实现一些设计模式。

enum关键字在 java5 中引入，表示一种特殊类型的类，其总是继承java.lang.Enum类，更多内容可以自行查看其官方文档。

枚举很多时候会和常量拿来对比，可能因为本身我们大量实际使用枚举的地方就是为了替代常量。那么这种方式由什么优势呢？

以这种方式定义的常量使代码更具可读性，允许进行编译时检查，预先记录可接受值的列表，并避免由于传入无效值而引起的意外行为。

下面示例定义一个简单的枚举类型 pizza 订单的状态，共有三种 ORDERED, READY, DELIVERED 状态：

```
package shuang.kou.enumdemo.enumtest;

public enum PizzaStatus{
    ORDERED,
    READY,
    DELIVERED;
}
```

简单来说，我们通过上面的代码避免了定义常量，我们将所有和 pizza 订单的状态的常量都统一放到了一个枚举类型里面。

```
System.out.println(PizzaStatus.ORDERED.name());//ORDERED
System.out.println(PizzaStatus.ORDERED);//ORDERED
System.out.println(PizzaStatus.ORDERED.name().getClass());//class java.lang.String
System.out.println(PizzaStatus.ORDERED.getClass());//class shuang.kou.enumdemo.enumtest.PizzaStatus
```

2.自定义枚举方法

现在我们对枚举是什么以及如何使用它们有了基本的了解，让我们通过在枚举上定义一些额外的API方法，将上一个示例提升到一个新的水平：

```
public class Pizza {  
    private PizzaStatus status;  
    public enum PizzaStatus {  
        ORDERED,  
        READY,  
        DELIVERED;  
    }  
  
    public boolean isDeliverable() {  
        if (getStatus() == PizzaStatus.READY) {  
            return true;  
        }  
        return false;  
    }  
  
    // Methods that set and get the status variable.  
}
```

3. 使用 == 比较枚举类型

由于枚举类型确保JVM中仅存在一个常量实例，因此我们可以安全地使用“==”运算符比较两个变量，如上例所示；此外，“==”运算符可提供编译时和运行时的安全性。

首先，让我们看一下以下代码段中的运行时安全性，其中“==”运算符用于比较状态，并且如果两个值均为 null 都不会引发 NullPointerException。相反，如果使用 equals 方法，将抛出 NullPointerException：

```
if(testPz.getStatus().equals(Pizza.PizzaStatus.DELIVERED));  
if(testPz.getStatus() == Pizza.PizzaStatus.DELIVERED);
```

对于编译时安全性，我们看另一个示例，两个不同枚举类型进行比较，使用 equal 方法比较结果确定为 true，因为 getStatus 方法的枚举值与另一个类型枚举值一致，但逻辑上应该为 false。这个问题可以使用 == 操作符避免。因为编译器会表示类型不兼容错误：

```
if(testPz.getStatus().equals(TestColor.GREEN));  
if(testPz.getStatus() == TestColor.GREEN);
```

4. 在 switch 语句中使用枚举类型

```
public int getDeliveryTimeInDays() {  
    switch (status) {  
        case ORDERED: return 5;  
        case READY: return 2;  
        case DELIVERED: return 0;  
    }  
    return 0;  
}
```

5. 枚举类型的属性,方法和构造函数

“

文末有我 (JavaGuide) 的补充。

”

你可以通过在枚举类型中定义属性,方法和构造函数让它变得更加强大。

下面,让我们扩展上面的示例,实现从比萨的一个阶段到另一个阶段的过渡,并了解如何摆脱之前使用的if语句和switch语句:

```
public class Pizza {  
  
    private PizzaStatus status;  
    public enum PizzaStatus {  
        ORDERED (5){  
            @Override  
            public boolean isOrdered() {  
                return true;  
            }  
        },  
        READY (2){  
            @Override  
            public boolean isReady() {  
                return true;  
            }  
        },  
        DELIVERED (0){  
            @Override  
            public boolean isDelivered() {  
                return true;  
            }  
        };  
  
        private int timeToDelivery;  
  
        public boolean isOrdered() {return false;}  
  
        public boolean isReady() {return false;}  
  
        public boolean isDelivered(){return false;}  
  
        public int getTimeToDelivery() {  
            return timeToDelivery;  
        }  
  
        PizzaStatus (int timeToDelivery) {  
            this.timeToDelivery = timeToDelivery;  
        }  
    }  
  
    public boolean isDeliverable() {  
        return this.status.isReady();  
    }  
  
    public void printTimeToDeliver() {  
        System.out.println("Time to delivery is " +  
            this.getStatus().getTimeToDelivery());  
    }  
  
    // Methods that set and get the status variable.  
}
```

下面这段代码展示它是如何 work 的:

```
@Test  
public void givenPizzaOrder_whenReady_thenDeliverable() {  
    Pizza testPz = new Pizza();  
    testPz.setStatus(Pizza.PizzaStatus.READY);  
    assertTrue(testPz.isDeliverable());  
}
```

6.EnumSet and EnumMap

6.1. EnumSet

EnumSet 是一种专门为枚举类型所设计的 Set 类型。

与 HashSet 相比，由于使用了内部位向量表示，因此它是特定 Enum 常量集的非常有效且紧凑的表示形式。

它提供了类型安全的替代方法，以替代传统的基于 int 的“位标志”，使我们能够编写更易读和易于维护的简洁代码。

EnumSet 是抽象类，其有两个实现：RegularEnumSet、JumboEnumSet，选择哪一个取决于实例化时枚举中常量的数量。

在很多场景中的枚举常量集合操作（如：取子集、增加、删除、containsAll 和 removeAll 批操作）使用 EnumSet 非常合适；如果需要迭代所有可能的常量则使用 Enum.values()。

```
public class Pizza {

    private static EnumSet<PizzaStatus> undeliveredPizzaStatuses =
        EnumSet.of(PizzaStatus.ORDERED, PizzaStatus.READY);

    private PizzaStatus status;

    public enum PizzaStatus {
        ...
    }

    public boolean isDeliverable() {
        return this.status.isReady();
    }

    public void printTimeToDeliver() {
        System.out.println("Time to delivery is " +
            this.getStatus().getTimeToDelivery() + " days");
    }

    public static List<Pizza> getAllUndeliveredPizzas(List<Pizza> input) {
        return input.stream().filter(
            (s) -> undeliveredPizzaStatuses.contains(s.getStatus()))
            .collect(Collectors.toList());
    }

    public void deliver() {
        if (isDeliverable()) {
            PizzaDeliverySystemConfiguration.getInstance().getDeliveryStrategy()
                .deliver(this);
            this.setStatus(PizzaStatus.DELIVERED);
        }
    }

    // Methods that set and get the status variable.
}
```

下面的测试演示了展示了 `EnumSet` 在某些场景下的强大功能：

```
@Test
```

```
public void givenPizzaOrders_whenRetrievingUndeliveredPzs_thenCorrectlyRetrieved() {  
    List<Pizza> pzList = new ArrayList<>();  
    Pizza pz1 = new Pizza();  
    pz1.setStatus(Pizza.PizzaStatus.DELIVERED);  
  
    Pizza pz2 = new Pizza();  
    pz2.setStatus(Pizza.PizzaStatus.ORDERED);  
  
    Pizza pz3 = new Pizza();  
    pz3.setStatus(Pizza.PizzaStatus.ORDERED);  
  
    Pizza pz4 = new Pizza();  
    pz4.setStatus(Pizza.PizzaStatus.READY);  
  
    pzList.add(pz1);  
    pzList.add(pz2);  
    pzList.add(pz3);  
    pzList.add(pz4);  
  
    List<Pizza> undeliveredPzs = Pizza.getAllUndeliveredPizzas(pzList);  
    assertTrue(undeliveredPzs.size() == 3);  
}
```

6.2. EnumMap

EnumMap是一个专门化的映射实现，用于将枚举常量用作键。与对应的HashMap相比，它是一个高效紧凑的实现，并且在内部表示为一个数组：

```
EnumMap<Pizza.PizzaStatus, Pizza> map;
```

让我们快速看一个真实的示例，该示例演示如何在实践中使用它：

```
public static EnumMap<PizzaStatus, List<Pizza>>  
groupPizzaByStatus(List<Pizza> pizzaList) {  
    EnumMap<PizzaStatus, List<Pizza>> pzByStatus =  
        new EnumMap<PizzaStatus, List<Pizza>>(PizzaStatus.class);  
  
    for (Pizza pz : pizzaList) {  
        PizzaStatus status = pz.getStatus();  
        if (pzByStatus.containsKey(status)) {  
            pzByStatus.get(status).add(pz);  
        } else {  
            List<Pizza> newPzList = new ArrayList<Pizza>();  
            newPzList.add(pz);  
            pzByStatus.put(status, newPzList);  
        }  
    }  
    return pzByStatus;  
}
```

下面的测试演示了展示了EnumMap在某些场景下的强大功能：

```
@Test  
public void givenPizzaOrders_whenGroupByStatusCalled_thenCorrectlyGrouped() {  
    List<Pizza> pzList = new ArrayList<>();  
    Pizza pz1 = new Pizza();  
    pz1.setStatus(Pizza.PizzaStatus.DELIVERED);  
  
    Pizza pz2 = new Pizza();  
    pz2.setStatus(Pizza.PizzaStatus.ORDERED);  
  
    Pizza pz3 = new Pizza();  
    pz3.setStatus(Pizza.PizzaStatus.ORDERED);  
  
    Pizza pz4 = new Pizza();  
    pz4.setStatus(Pizza.PizzaStatus.READY);  
  
    pzList.add(pz1);  
    pzList.add(pz2);  
    pzList.add(pz3);  
    pzList.add(pz4);  
  
    EnumMap<Pizza.PizzaStatus, List<Pizza>> map = Pizza.groupPizzaByStatus(pzList);  
    assertTrue(map.get(Pizza.PizzaStatus.DELIVERED).size() == 1);  
    assertTrue(map.get(Pizza.PizzaStatus.ORDERED).size() == 2);  
    assertTrue(map.get(Pizza.PizzaStatus.READY).size() == 1);  
}
```

7. 通过枚举实现一些设计模式

7.1 单例模式

通常，使用类实现 `Singleton` 模式并非易事，枚举提供了一种实现单例的简便方法。

《Effective Java》和《Java与模式》都非常推荐这种方式，使用这种方式实现枚举可以有什么好处呢？

《Effective Java》

“

这种方法在功能上与公有域方法相近，但是它更加简洁，无偿提供了序列化机制，绝对防止多次实例化，即使是在面对复杂序列化或者反射攻击的时候。虽然这种方法还没有广泛采用，但是单元素的枚举类型已经成为实现 `Singleton` 的最佳方法。——《Effective Java 中文版 第二版》

”

《Java与模式》

“

《Java与模式》中，作者这样写道，使用枚举来实现单实例控制会更加简洁，而且无偿地提供了序列化机制，并由JVM从根本上提供保障，绝对防止多次实例化，是更简洁、高效、安全的实现单例的方式。

”

下面的代码段显示了如何使用枚举实现单例模式：

```

public enum PizzaDeliverySystemConfiguration {
    INSTANCE;
    PizzaDeliverySystemConfiguration() {
        // Initialization configuration which involves
        // overriding defaults like delivery strategy
    }

    private PizzaDeliveryStrategy deliveryStrategy = PizzaDeliveryStrategy.NORMAL;

    public static PizzaDeliverySystemConfiguration getInstance() {
        return INSTANCE;
    }

    public PizzaDeliveryStrategy getDeliveryStrategy() {
        return deliveryStrategy;
    }
}

```

如何使用呢？请看下面的代码：

```
PizzaDeliveryStrategy deliveryStrategy = PizzaDeliverySystemConfiguration.getInstance().getDeliveryStrategy();
```

通过 `PizzaDeliverySystemConfiguration.getInstance()` 获取的就是单例的 `PizzaDeliverySystemConfiguration`

7.2 策略模式

通常，策略模式由不同类实现同一个接口来实现的。

这也就意味着添加新策略意味着添加新的实现类。使用枚举，可以轻松完成此任务，添加新的实现意味着只定义具有某个实现的另一个实例。

下面的代码段显示了如何使用枚举实现策略模式：

```

public enum PizzaDeliveryStrategy {
    EXPRESS {
        @Override
        public void deliver(Pizza pz) {
            System.out.println("Pizza will be delivered in express mode");
        }
    },
    NORMAL {
        @Override
        public void deliver(Pizza pz) {
            System.out.println("Pizza will be delivered in normal mode");
        }
    };

    public abstract void deliver(Pizza pz);
}

```

给 `Pizza` 增加下面的方法：

```
public void deliver() {  
    if (isDeliverable()) {  
        PizzaDeliverySystemConfiguration.getInstance().getDeliveryStrategy()  
            .deliver(this);  
        this.setStatus(PizzaStatus.DELIVERED);  
    }  
}
```

如何使用呢？请看下面的代码：

```
@Test  
public void givenPizzaOrder_whenDelivered_thenPizzaGetsDeliveredAndStatusChanges() {  
    Pizza pz = new Pizza();  
    pz.setStatus(Pizza.PizzaStatus.READY);  
    pz.deliver();  
    assertTrue(pz.getStatus() == Pizza.PizzaStatus.DELIVERED);  
}
```

8. Java 8 与枚举

Pizza 类可以用 Java 8 重写，您可以看到方法 lambda 和 Stream API 如何使 getAllUndeliveredPizzas() 和 groupPizzaByStatus() 方法变得如此简洁：

getAllUndeliveredPizzas() :

```
public static List<Pizza> getAllUndeliveredPizzas(List<Pizza> input) {  
    return input.stream().filter(  
        (s) -> !deliveredPizzaStatuses.contains(s.getStatus()))  
        .collect(Collectors.toList());  
}
```

groupPizzaByStatus() :

```
public static EnumMap<PizzaStatus, List<Pizza>>  
groupPizzaByStatus(List<Pizza> pzList) {  
    EnumMap<PizzaStatus, List<Pizza>> map = pzList.stream().collect(  
        Collectors.groupingBy(Pizza::getStatus,  
        () -> new EnumMap<>(PizzaStatus.class), Collectors.toList()));  
    return map;  
}
```

9. Enum 类型的 JSON 表现形式

使用 Jackson 库，可以将枚举类型的 JSON 表示为 POJO。下面的代码段显示了可以用于同一目的的 Jackson 批注：

```

@JsonFormat(shape = JsonFormat.Shape.OBJECT)
public enum PizzaStatus {
    ORDERED (5){
        @Override
        public boolean isOrdered() {
            return true;
        }
    },
    READY (2){
        @Override
        public boolean isReady() {
            return true;
        }
    },
    DELIVERED (0){
        @Override
        public boolean isDelivered() {
            return true;
        }
    };

    private int timeToDelivery;

    public boolean isOrdered() {return false;}

    public boolean isReady() {return false;}

    public boolean isDelivered(){return false;}

    @JsonProperty("timeToDelivery")
    public int getTimeToDelivery() {
        return timeToDelivery;
    }

    private PizzaStatus (int timeToDelivery) {
        this.timeToDelivery = timeToDelivery;
    }
}

```

我们可以按如下方式使用 Pizza 和 PizzaStatus：

```

Pizza pz = new Pizza();
pz.setStatus(Pizza.PizzaStatus.READY);
System.out.println(Pizza.getString(pz));

```

生成 Pizza 状态以以下 JSON 展示：

```
{
    "status": {
        "timeToDelivery": 2,
        "ready": true,
        "ordered": false,
        "delivered": false
    },
    "deliverable": true
}
```

有关枚举类型的 JSON 序列化/反序列化(包括自定义)的更多信息,请参阅 Jackson - 将枚举序列化为 JSON 对象。

10. 总结

本文我们讨论了 Java 枚举类型,从基础知识到高级应用以及实际应用场景,让我们感受到枚举的强大功能。

11. 补充

我们在上面讲到了,我们可以通过在枚举类型中定义属性,方法和构造函数让它变得更加强大。

下面我通过一个实际的例子展示一下,当我们调用短信验证码的时候可能有几种不同的用途,我们在下面这样定义:

```
public enum PinType {  
  
    REGISTER(100000, "注册使用"),  
    FORGET_PASSWORD(100001, "忘记密码使用"),  
    UPDATE_PHONE_NUMBER(100002, "更新手机号码使用");  
  
    private final int code;  
    private final String message;  
  
    PinType(int code, String message) {  
        this.code = code;  
        this.message = message;  
    }  
  
    public int getCode() {  
        return code;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
  
    @Override  
    public String toString() {  
        return "PinType{" +  
            "code=" + code +  
            ", message='" + message + '\'' +  
            '}';  
    }  
}
```

实际使用:

```
System.out.println(PinType.FORGET_PASSWORD.getCode());  
System.out.println(PinType.FORGET_PASSWORD.getMessage());  
System.out.println(PinType.FORGET_PASSWORD.toString());
```

Output:

100001

忘记密码使用

PinType{code=100001, message='忘记密码使用'}

这样的话，在实际使用起来就会非常灵活方便！

-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」，欢迎添加小编微信「focuseoncode」，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



[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！

一份详细的 Java 项目实践清单

Java后端 2019-10-18



来自：等你归去来 | 责编：乐乐

链接：cnblogs.com/yougewe

正文

虽说工作就是简单的事情重复做，但不是所有简单的事你都能有机会做的。

我们平日工作里，大部分时候都是在做修修补补的工作，而这也是非常重要的。做好修补工作，做好优化工作，足够让你升职加薪！

但是如果有机会，去尝试些自己平日里少做的事，我觉得是一件值得庆幸的事。

前段时间，接了个新项目。只有一些idea在业务需求方脑海里，然后就开始需求讨论，然后就开始做事了。项目不复杂，但是由于是用JAVA语言实现（这相对来说是我的薄弱点），对我个人显得比较有意义。

总结下来，其实也就是一个项目清单。个人觉得还是有点意义吧，给没有一定全面实践的同学参考吧

1. 项目规划

1.1 首先，你得彻底明白到底要做什么？

这个过程，可能是你要读需求一遍、两遍、三遍……然后假设，你已经在使用这个产品了。

1.2 其次，明白需求后，就要进行整体框架的构思！

比如用什么呈现给用户，用什么来存储数据，需要些什么样的系统等。

在这个层次上，一般都会遵循公司的规定，然后再根据项目本身需求，做些相应的调整。

我们在这个项目里的整体框架为：前端使用 APP(ios&android)、H5进行用户界面呈现 ==> 接入网关进行数据加解密，流控转发等 ==> 第一层API服务，接受客户端请求，做简单业务检验组装 ==> 第二层核心业务SERVICE服务，进行核心业务处理，如写库、调用第三方接口等 ==> 最下层基础服务，提供单一的功能服务，如消息服务，订单服务。

前期只提供APP，因此不存在单独H5调用API服务的情况，但是H5的应用场景仍然存在，此时的H5地址，由服务接口提供地址返回到APP进行webview加载。

1.3 人员规划

项目整体框架出来后，得要有人去实施才行。

这里一般需要遵循一个最小原则，即划分出的人员，尽量做到能够独立完成自有的模块，而不是一定要依赖于另一方的实现才能进一步。比如 android,ios各一人，API与SERVICE可以多个人，但是都要让其有全部权限，因为API与SERVICE有强依赖，脱离一方，将无法独立完成。基础服务各自安排相关人员实现。最后进行联调即可。

1.4 时间规划

有了人员之后，也不能无限时间的去做事。肯定是要规划的，否则没有压力也没有动力。项目不知何时才能结束。订时间计划一定要去询问当事人，要多少时间，尽量站在专业的角度给出合理的建议和评估。促进项目的完成。

2. 框架规划及搭建

2.1 有了整体框架的构思后，就要细节到每个层次的实践了

因为都是应用的分层，所以，不可能有统一的描述，只能是针对每个应用层。做自己该做的事。如android/ios 有自己的开发框架；

h5有自己的开发框架(因为很多应用场景可能涉及到h5与app原生的交互，所以即使功能简单，也尽量利用一些已有的框架进行开发)。

而服务端，虽分为多层应用，但是应尽量使用同一门语言，利用同一套开发框架，自己公司有研发框架自然最好，没有也尽量利用统一的开源框架。这样做的好处是，当有人员变动时，可以立即熟悉其代码及应用场景，从而增加适应性和管理性。

针对服务端的框架，我觉得有必要多说点。因为整个应用运行的流畅性，可靠性，准确性，都是由服务端来决定的。虽然用户看到的是APP或者H5，但是可以说，服务端才是应用的核心。所以，服务端要做的事情自然很多了。

2.2 怎样搭建好一些服务端的框架呢？

首先，框架类的东西，自然是要提前学习的。但是，就目前市场行情来说，要想利用框架应该都是比较简单的，尤其是公司内部提供的框架，一定要有demo。这样，照着demo，一步步调试，直到整个应用接通；

删除不需要的模块，添加特别需要的模块，保证在具体开发过程中，能够想利用啥就有啥可利用；

充分了解框架需要的一些配置参数，知道事务从哪里来，到哪里去？这里，应有一个配置中心与之对应，但是自己得清楚。

使用一个顺手的IDE工具，不是说你技术不够牛逼，而是一个好的工具，能够让你事半功倍。（其实能够多背点套路，也不一定

写出第一个可供使用的接口服务，可以说，第一个永远是比较重要的。因为，第一个的思路，就是你后续所有功能的方向，因此，写好第一个"hello, world."；

3. 开发环境的搭建(服务端)

其实这项工作是及其重要的，之所以把它放在第三点，是因为，没有代码作铺垫，开发环境搭了也没用。

开发环境的搭建，主要也是服从于整体框架的构思。

主要包括，需要多少个服务，需要多少台服务器，需要多少个基础应用，需要多少个基础配置等等。

当然，开发环境本身就是一个很大的难题，一般还是交给专业运维几十年的老司机来完成了。自己就当作了解得了。

目前的项目开发，除一些小规模公司还在利用一套服务端代码，干完所有的事外，大部分应该都是多个应用的配合完成。而测试环境，不太可能利用多个服务器提供服务。因此，使用docker进行测试环境搭建尤佳。建立多个docker进行多个服务器模拟，也算是和线上环境保持一致了。

目前的主流技术得用上（当然关键还得看你的框架规划），zookeeper, dubbo, redis, mongo, mq, …

只有开发环境搭建好了，才能让后面的流程无忧。搭建的过程一定是，又搭建，又改代码，又排错…

4. 进度的同步

4.1 及时向领导同步项目进度

对于一个新项目，有些地方行动缓慢是很正常的。而部分开发同学（比如我自己），就喜欢沉浸在自己的小世界里纠结，走不出来，从而忘却向领导汇报工作。而作为一个有点同理心的领导来说，他又不愿意实时都来盯着你做事，因为也怕你遇到困难，想多给你点时间解决。

但是，这种情况，开发同学自己其实是要吃亏的，因为，给外人的感觉就是，你啥都没做。所以，解决问题的同时，也不忘向领导汇报。

4.2 有处理不了的问题，及时向大牛们或者领导请教

独立解决问题是好事，但是千万别过了头，实在解决不了，就要及时请教。否则，浪费的是时间。进步最快的方式，莫过于向比自己牛逼的人请教。知之为知之，不知为不知！

4.3 尽量将问题分摊下去

问题肯定是有，而且会很多。千万不要把所有的事情都压在自己这儿，那样自己会累死的，而且项目进度也会因此变得缓慢。要多利用小组成员的各自优点，适当多让其搞点事情。

工作永远都不是单一的一件事，肯定还会有其他的事情插入进来，观察事情的重要性解决。如果能够让其他同学解决的，尽量让其他同学处理，这点也得与领导同步。否则分心过于利害，受阻的只有项目进度，延期可不是自己一人的事情了。

需求也不可能一下就是完善的，在做的过程中，才可能发现一些潜在的问题，这时及时与需求方沟通，保持高效的状态。当然，后期的跟进，也是尽量做到不要一人大包大揽，而是相应的人就去负责相应事情的跟进。其他人只要知道结果就行。

5. 功能模块的完成

说到具体的业务实现，个人觉得，已经不那么难了。不过就是，先尽力提出的一个初稿，然后发现问题解决问题，发现问题，解决问题的过程。

各自系统能做的事情完成后，就是联调各系统间的调用关系，保持高效的沟通，让问题在短时间内解决，尤为重要。在这种时候，我觉得，一个小黑屋也许也是个不错的选择。

联调的过程，其实就是一个自测的过程，应把尽可能多情况给考虑到位。

代码检查，自己开发的代码，基本上很难发现其中的问题，即时找到相应人帮忙检查代码，是比较好的解决代码问题的方案。其实，在给别人检查的时候，也是自己检查的时候，相当于自己再一次的开发，也能及时发现问题。

6. 多轮的测试验证

测试同学，其实在开发快结束的时候，已经把测试用例给到大家。这也是另一个角度的开发，因此，参考测试用例进行相应开发修改也是很有必要的。

1. 第一轮测试，可能主要是大功能的验证，小功能的检查，挡板环境即可，无需真实环境。
2. 第二轮测试，则是要把之前的测试及各种配置，全部清空，以一个全新的项目来对待，重新进行相应环境搭建，代码部署，然后再进行测试，确保问题解决后，做好了相应的处理方案备份。这时，就需要用到真实的应用环境了。对之前一些暂未解决的问题进行重新测试。确保无问题。
3. 第三轮测试，应该是一个灰度发布的环境，也可以认为是预上线。将所有环境当作是线上来处理，如果运行ok,即可准备发布上线了。

在测试过程中，因测试人员只是人工的处理，有时不一定能捕获所有的问题，开发在这时，也应站在测试的角度，发现问题，即时监控，即时处理。

4. 自动化测试，这个其实应该是靠后的处理，但是如果能做到这些的话，也能够快速的重现问题。
5. 压力测试，应对线上环境，需有一定的能力评估，不然，只瞎猜，恐怕也不是好事。随时准备横向扩展，也只是出现问题后的解决方案。做好压测，发现代码中存在的问题，即时处理掉。

7. 外围处理(上线前)

7.1 上线前，肯定是有许多事务要处理的。

1. 测试环境中的各种基础数据，随时导出备份，到线上时，直接插入使用；
2. 服务器，在架构评审过程中进行数量评估；
3. 域名，对外网提供服务一定是要域名的；
4. 权限，比如上线后，出现了问题，谁有权限来处理问题，一定提前给到；
5. 验收，这是关键的一点，功能完成后，及时验收，如果上线有些小问题，尽量协商，不要在线上频繁改动。

如此，整个项目就完工了。

其实发现，一个项目真正的功能实现，并没有占多大的比例，而是一些前期的准备及后续的处理，反而占了更多的时间。

第一个版本上线后，可能接着就是迅速迭代了。（如果运营还可以的话！）

以上，就是一整个项目的流程清单，以一步一个脚印的经历总结，不涉及具体语言代码，但是思路都是相通的，希望对你有帮助！

- 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 ↓



推荐阅读

1. 感受 Lambda 之美，推荐收藏，需要时查阅
2. 如何优雅的导出 Excel
3. 文艺互联网公司 vs 二逼互联网公司
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



喜欢文章，点个在看 

一台 Java 服务器可以跑多少个线程？

Java后端 2019-10-15

点击上方 Java后端, 选择设为星标

技术博文, 及时送达

作者 | 新栋BOOK

链接 | www.jianshu.com/p/f1930596947d

一台Java服务器能跑多少个线程？这个问题来自一次线上报警如下图，超过了我们的配置阈值。



京东自研UMP监控分析

打出jstack文件，通过IBM Thread and Monitor Dump Analyzer for Java工具查看如下：

● Thread Status Analysis

Status	Number of Threads : 1661	Percentage
Deadlock	0	0 (%)
Runnable	375	23 (%)
Waiting on condition	1228	74 (%)
Waiting on monitor	0	0 (%)
Suspended	0	0 (%)
Object.wait()	51	3 (%)
Blocked	7	0 (%)
Parked	0	0 (%)

● Thread Method Analysis

Method Name	Number of Threads : 1661	Percentage
sun.misc.Unsafe.park(Native Method)	1201	72 (%)
sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)	307	18 (%)
NO JAVA STACK	60	4 (%)
java.lang.Object.wait(Native Method)	51	3 (%)
java.lang.Thread.sleep(Native Method)	32	2 (%)
java.net.SocketInputStream.socketRead0(Native Method)	5	0 (%)
java.net.PlainSocketImpl.socketAccept(Native Method)	5	0 (%)

● Thread Aggregation Analysis

Thread Type	Number of Threads : 1661	Percentage
Thread	31	2 (%)

IBM Thread and Monitor Dump Analyzer for Java

共计1661个线程，和监控数据得出的吻合。但这个数量应该是大了，我们都知道线程多了，就会有线程切换，带来性能开销。

当时就想到一台java服务器到底可以跑多少个线程呢？跟什么有关系？现整理如下。

每个线程都有一个线程栈空间通过-Xss设置，查了一下我们服务器的关于jvm内存的配置

```

1 -Xms4096m
2 -Xmx4096m
3 -XX:MaxPermSize=1024m

```

只有这三个，并没有-Xss 和-XX:ThreadStackSize的配置，因此是走的默认值。几种JVM的默认栈大小

操作系统	32位	64位
Linux	320 KB	1 MB
Mac OS	N/A	1 MB
Solaris Sparc	512 KB	1 MB
Solaris X86	320 KB	1 MB
Windows	320 KB	1 MB

可以通过如下命令打印输出默认值的大小，命令：**jinfo -flag ThreadStackSize**；例如

```
1 [root@host-192-168-202-229 ~]#jinfo -flag ThreadStackSize 1807  
2  
3 -XX:ThreadStackSize=1024
```

不考虑系统限制，可以通过如下公式计算，得出最大线程数量

线程数量 = (机器本身可用内存-JVM分配的堆内存) /Xss的值，比如我们的容器本身大小是8G,堆大小是4096M,走-Xss默认值，可以得出 最大线程数量：4096个。

根据计算公式，得出如下结论：

结论1：jvm堆越大，系统创建的线程数量越小。

结论2：当-Xss的值越小，可生成线程数量越多。

我们知道操作系统分配给每个进程的内存大小是有限制的，比如32位的Windows是2G。因此操作系统对一个进程下的线程数量是有限制的，不能无限的增多。经验值：3000-5000左右（我没有验证）。

刚才说的是不考虑系统限制的情况，那如果考虑系统限制呢，主要跟以下几个参数有关系

/proc/sys/kernel/pid_max 增大，线程数量增大，*pid_max*有最高值，超过之后不再改变，而且32, 64位也不一样

/proc/sys/kernel/thread-max 系统可以生成最大线程数量

max_user_process (ulimit -u) centos系统上才有，没有具体研究

/proc/sys/vm/max_map_count 增大，数量增多

线程是非常宝贵的资源，我们要严格控制线程的数量，象上面我们的截图情况，显然线程数量过多。这个是跟我们自己配置了fixed大小的线程池有关系。京东有自己的rpc框架jsf，里面可以针对每个服务端口设置线程大小。

-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 IntelliJ IDEA 那些不为人知的小技巧
2. Java 开发中常用的 4 种加密方法
3. 某小公司RESTful、前后端分离的实践
4. 该如何弥补 GitHub 功能缺陷？
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！

一起来学 Java 注解 (Annotation)

Java后端 2019-10-11

点击上方 Java后端, 选择设为星标

技术博文, 及时送达

作者 | 工匠初心

链接 | blog.csdn.net/fengdongsuixin

[上一篇 从零搭建创业公司后台技术栈](#)

一. 什么是Annotation

我们在平时的开发过程中看到很多如@Override, @SuppressWarnings, @Test等样式的代码就是注解, 注解是放到类、构造器、方法、属性、参数前的标记。

二. Annotation的作用

给某个类、方法…添加了一个注解, 这个环节仅仅是做了一个标记, 对代码本身并不会造成任何影响, 需要后续环节的配合, 需要其他方法对该注解赋予业务逻辑处理。就如同我们在微信上发了一个共享定位, 此时并没有什么用, 只有当后面其他人都进入了这个共享定位, 大家之间的距离才能明确, 才知道该怎么聚在一起。

注解分为三类:

2.1 编译器使用到的注解

如@Override, @SuppressWarnings都是编译器使用到的注解, 作用是告诉编译器一些事情, 而不会进入编译后的.class文件。

@Override: 告诉编译器检查一下是否重写了父类的方法;

@SuppressWarnings: 告诉编译器忽略该段代码产生的警告;

对于开发人员来说, 都是直接使用, 无需进行其他操作

2.2 .class文件使用到的注解

需要通过工具对.class字节码文件进行修改的一些注解, 某些工具会在类加载的时候, 动态修改用某注解标注的.class文件, 从而实现一些特殊的功能, 一次性处理完成后, 并不会存在于内存中, 都是非常底层的工具库、框架会使用, 对于开发人员来说, 一般不会涉及到。

2.3 运行期读取的注解

一直存在于JVM中, 在运行期间可以读取的注解, 也是最常用的注解, 如Spring的@Controller, @Service, @Repository, @AutoWired, Mybatis的@Mapper, Junit的@Test等, 这类注解很多都是工具框架自定义在运行期间发挥特殊作用的注解, 一般开发人员也可以自定义这类注解。微信搜索 web_resource 关注获取更多推送

三. 定义Annotation

我们使用@interface来定义一个注解

```
1 public @interface Table
2 {
3     String value() default ""
4 ;
5 }
```

```
1 public @interface Column {
2     String value() default ""
3 ;
4     String name() default ""
5 ;
6     String dictType() default ""
7 }
```

这样就简单地将一个注解定义好了

我们上面定义的注解主要用到了String类型,但实际上还可以是基本数据类型(不能为包装类)、枚举类型。

注解也有一个约定俗成的东西,最常用的参数应该命名为value,同时一般情况下我们都会通过default参数设置一个默认值。

但这样是不是就满足于我们的使用了呢,我想把@Table注解仅用于类上,@Column注解仅用于属性上,怎么办?而且开始提到的三类注解,一般开发人员用的都是运行期的注解,那我们定义的是吗?

要回答这些问题,就需要引入一个概念“元注解”。

3.1 元注解

可以修饰注解的注解即为元注解,Java已经定义了一些元注解,我们可以直接使用。微信搜索 web_resource 关注获取更多推送

3.1.1 @Target

顾名思义指定注解使用的目标对象,参数为ElementType[]

```
1 public @interface Target
2 {
3     /**
4      * Returns an array of the kinds of elements an annotation type
5      * can be applied to.
6      * @return an array of the kinds of elements an annotation type
7      * can be applied to
8      */
9     ElementType[] value()
10 }
```

```
}
```

而下面是ElementType枚举中定义的属性，不设置Target的时候，除了TYPE_PARAMETER, TYPE_USE, 其他地方都相当于配置上了。

```
1 public enum ElementType {
2     /** 通过ElementType.TYPE可以修饰类、接口、枚举 */
3     TYPE,
4
5     /** 通过ElementType.FIELD可以修饰类属性 */
6     FIELD,
7
8     /** 通过ElementType.METHOD可以修饰方法 */
9     METHOD,
10
11    /** 通过ElementType.PARAMETER可以修饰参数（如构造器或者方法中的） */
12    PARAMETER,
13
14    /** 通过ElementType.CONSTRUCTOR可以修改构造器 */
15    CONSTRUCTOR,
16
17    /** 通过ElementType.LOCAL_VARIABLE可以修饰方法内部的局部变量 */
18    LOCAL_VARIABLE,
19
20    /** 通过ElementType.ANNOTATION_TYPE可以修饰注解 */
21    ANNOTATION_TYPE,
22
23    /** 通过ElementType.PACKAGE可以修饰包 */
24    PACKAGE,
25
26    /*
27     *
28     * 可以用在Type的声明式前
29     *
30     * @since 1.
31 8
32     */
33    TYPE_PARAMETER,
34
35    /*
36     *
37     * 可以用在所有使用Type的地方（如泛型、类型转换等）
38     *
39     * @since 1.
40 8
41     */
42    TYPE_USE
43 }
```

ElementType.PACKAGE

```

1 @Target(ElementType.PACKAGE)
2 public @interface Table {
3     String value() default "";
4 }

```

含义是用来修饰包，但我们用来修饰包的时候却提示错误

2 @Table package annotation;

Package annotations should be in file package-info.java

```

5
6     public class AnnoTest {
7 }
```

我们按照提示创建package-info.java文件，这里需要注意一下，通过IDE 进行new --> Java Class是创建不了的，需要通过new File文件创建

```

1 @Table
2 package annotation;
3 class PackageInfo {
4     public void hello()
5     {
6         System.out.println("hello")
7     }
}
```

```

javase-learning E:\idea-work\javase-learn
> .idea
> out
src
  & annotation
    C AnnoTest
    & @ Colum
    & package-info.java
      & C PackageInfo
        m hello():void
    & Table

```

```

1 @Table
2 package annotation;
3 class PackageInfo {
4     public void hello() {
5         System.out.println("hello")
6     }
7 }
```

package-info.java看似一个类，但实际上仅仅是用
来存放PackageInfo类的一个容器，或许可以理解
为“类包”，在这里面定义的类仅能被当前包
(annotation) 调用，当前包下的子包都无法调用

ElementType.TYPE_PARAMETER和ElementType.TYPE_USE

这两个一起说，因为它们有相似之处。都是Java1.8后添加的

```
1 @Target(ElementType.TYPE_USE)
2 public @interface NoneEmpty {
3     String value() default ""
4 ;
5 }
```

```
1 @Target(ElementType.TYPE_PARAMETER)
2 public @interface NoneBlank {
3     String value() default ""
4 ;
5 }
```

```
public class Holder<@NoneBlank T> {
    public @NoneBlank T test(@NoneBlank T a){
        new ArrayList<@NoneBlank String>();
    }
}

'@NoneBlank' not applicable to type use
⋮
Remove Alt+Shift+Enter More actions... Alt+Enter
```

```
public @NoneEmpty T test1(@NoneEmpty T a){
    new ArrayList<@NoneEmpty String>();
    return a;
}
```

很明显使用ElementType.TYPE_PARAMETER修饰的注解@NoneBlank无法在泛型使用的时候编译通过，仅能用于类的泛型声明，而通过ElementType.TYPE_USE修饰的注解@NoneEmpty可以。微信搜索 web_resource 关注获取更多推送

3.1.2 @Retention

可以用于定义注解的生命周期，参数为枚举RetentionPolicy，包括了SOURCE,CLASS,RUNTIME

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.ANNOTATION_TYPE)
3 public @interface Retention {
4     /*
5     *
```

```
5     * Returns the retention policy.
6     * @return the retention policy
7     */
8     RetentionPolicy value();
9 }
```

```
1 public enum RetentionPolicy {
2     /*
3     *
4     * 仅存在于源代码中，编译阶段会被丢弃，不会包含于class字节码文件中。
5     */
6     SOURCE
7 ,
8
9     /*
10    *
11    * 【默认策略】，在class字节码文件中存在，在类加载的时被丢弃，运行时无法获取到
12    */
13    CLASS
14 ,
15
16    /*
17    *
18    * 始终不会丢弃，可以使用反射获得该注解的信息。自定义的注解最常用的使用方式。
19    */
20    RUNTIME
21 }
22 }
```

3.1.3 @Documented

表示是否将此注解的相关信息添加到javadoc文档中

3.1.4 @Inherited

定义该注解和子类的关系，使用此注解声明出来的自定义注解，在使用在类上面时，子类会自动继承此注解，否则，子类不会继承此注解。注意，使用@Inherited声明出来的注解，只有在类上使用时才会有效，对方法，属性等其他无效。

```
1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Inherited
4 public @interface Person {
5     String value() default "man"
6 ;
7 }
```

```
1 @Person
2 public class Parent {
3 }
4 //子类也拥有@Person注解
```

```
5 class Son extends Parent {  
6  
7 }
```

3.2 定义注解小结

用@interface定义注解

可以添加多个参数，核心参数按约定用value，为每个参数可以设置默认值，参数类型包括基本类型、String和枚举

可以使用元注解来修饰注解，元注解包括多个，必须设置@Target和@Retention，@Retention一般设置为RUNTIME。

四. Annotation处理

我们前面已经提到光配置了注解，其实没有作用，需要通过相应的代码来实现该注解想要表达的逻辑。

注解定义后也是一种class，所有的注解都继承自java.lang.annotation.Annotation，因此，读取注解，需要使用反射API。

```
1 //定义的注解  
2 @Target(ElementType.FIELD)  
3 @Retention(RetentionPolicy.RUNTIME)  
4 public @interface Colum {  
5     String value() default ""  
6 ;  
7     //用于表示某个属性代表的中文含  
8    义  
9     String name() default ""  
10    ;  
11 }
```

用注解@Colum来修饰某个类的属性

```
1 public class Person {  
2  
3     @Colum(name = "姓名"  
4 )  
5     private String name;  
6  
7     @Colum(name = "性别"  
8 )  
9     private String gender;  
10  
11    @Colum(name = "年龄"  
12 )  
13    private int age;  
14  
15    @Colum(name = "住址"  
16 )  
17    private String address;
```

```
18 public String getName() {return name;}
19 }
20 public void setName(String name) {this.name = name;}
21 public String getGender() {return gender;}
22 public void setGender(String gender) {this.gender = gender;}
23 public int getAge() {return age;}
24 public void setAge(int age) {this.age = age;}
public String getAddress() {return address;}
public void setAddress(String address) {this.address = address;}
}
```

通过反射读取这个类的所有字段的中文含义，并保存到list中，然后打印出来

```
1 public static void main(String[] args) throws ClassNotFoundException
2 {
3     List<String> columnNames = new ArrayList<>();
4     Class clazz = Class.forName("annotation.Person")
5 ;
6     //获取Person类所有属性
7     Field[] fields = clazz.getDeclaredFields();
8     for (Field field : fields){
9         //获取该属性的Colum注解
10        Colum colum = field.getAnnotation(Colum.class);
11        //或者可以先判断有无该注
12        解
13        field.isAnnotationPresent(Colum.class);
14        //将该属性通过注解配置好的中文含义取出来放到集合
15        中
16        columnNames.add(colum.name());
17    }
18
19     //打印集
20     columnNames.forEach((columnName) -> System.out.println(columnName));
21 }
```

结果如下：

```
1 姓名
2 性别
3 年龄
4 住址
```

比如我们有一些常见的应用场景，需要把网站上的列表导出成excel表格，我们通过注解的方式把列名配置好，再通过反射读取实体需要导出（是否需要导出，也可通过注解配置）的每个字段的值，从而实现excel导出的组件。

五. 总结

本文只是抛砖引玉地讲解了注解的基本概念,注解的作用,几种元注解的功用以及使用方法,并通过一个简单的例子讲解了一下注解的处理,并不全面,文中通过Field讲解了注解的基本Api,但注解还可以修饰类、构造器、方法等,也有相对应的注解处理方法,大家可自行查一下API手册相关内容,大同小异,有不对之处,请批评指正,望共同进步,谢谢!

-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. [从零搭建创业公司后台技术栈](#)
2. [如何阅读 Java 源码?](#)
3. [某小公司RESTful、前后端分离的实践](#)
4. [该如何弥补 GitHub 功能缺陷?](#)
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!

世界 10 大编程语言，Java 不是第一，PHP 才第五

Java后端 1月9日



来源:toutiao.com/a6764554659349676557/

如果你是软件开发领域的新手，那么你会想到的第一个问题是“如何开始？”编程语言有数百种可供选择，但是你怎么发现哪个最适合你，你的兴趣和职业目标又在哪里呢？选择最佳编程语言以学习的最简单方法之一，是通过市场反响、技术趋势的发展…

阅读下文，你会发现一些用于Web开发，移动开发，游戏开发等的优秀、专业的编程语言。最后，你将清楚地了解哪种编程语言可以在未来几年甚至更长时间内帮助你的职业发展。让我们来看一看……

1、JavaScript

如今，如果连JavaScript都不会用，那么你不可能称之为一名合格的软件开发人员。榜单中的第一个是JavaScript，根本无法想象没有JavaScript的软件开发会是怎样的世界。从Stack Overflow的2019年开发人员调查中可以看出，JavaScript已经连续7年成为开发人员中最受欢迎的语言。过去一年中，大约有75%的人使用了这种语言。

首先，JavaScript是轻量级的，可解释的，并且在前端开发中起着重要作用的一门语言。甚至一些主要的社交媒体平台都认为JavaScript提供了一种轻松创建交互式网页的简便方法，并且是由职业驱动的。最受青睐的是JavaScript，因为它与所有主要浏览器兼容，并且其语法确实很灵活。作为一种前端语言，JavaScript还通过Node.js在服务器端使用。

JavaScript是初学者中最可爱的编程语言。

2、Python

这可能会让你感到惊讶；python出现在第二位。在许多调查中，它可能都放在第5上。但是，我一定会让你相信，这是为什么

呢？在我的list中，Python是通用的，用户友好的编程语言之一。为什么这么说？像Java一样，Python语法清晰，直观并且几乎类似于英语。Python的“基于对象”子集类似于JavaScript。根据Stack Overflow的说法，有一个部分说“被采用或被迁移，或者迁移得太早”，广泛来说，迁移到python的人接近42%，这表明它排名第二。

如果你有兴趣从事后端开发工作，例如Django –开放源代码框架，则是使用python编写的，这使得它易于学习且功能丰富，但却很受欢迎。另外，python具有多种应用程序，使其功能强大。在科学计算，机器学习和工程学等领域中，Python支持一种编程样式，该样式使用简单的函数和变量，而无需过多地查询类定义。

人生苦短，我用Python!

再者，因为人工智能这几年大热，而python尤其在大数据和人工智能领域有广泛的使用。

python本身面向对象语言，具有丰富和强大的库，轻松地使用C语言、C++、Cython来编写扩充模块，所以很多称它为“胶水语言”。当然仅仅知道这些还是不够的。

3、Java

如果有人问为什么Java，最常出现的句子是“写一遍，哪都可以运行” – Java在过去20年来一直是统治性的编程语言。Java是99%面向对象的，并且很强大，因为Java对象不包含对自身外部数据的引用。它比C ++更简单，因为Java使用自动内存分配和垃圾回收。

Java具有高度的跨平台兼容性或平台无关性。由于你可以在任何地方（我指的是所有设备）进行编码，因此可以编译为低级机器代码，最后，可以使用JVM – Java虚拟机（取决于平台）在任何平台上执行。

Java构成了Android操作系统的基础，并选择了约90%的财富500强公司来制作各种后端应用程序。我会毫不犹豫地采用由Amazon Web Services和Windows Azure运行的最大的Apache Hadoop数据处理。有许多充分的理由和广泛的业务应用程序，拥有巨大的灵活性，而Java一直是初学者的最爱。

4、C /

C++

“越老越吃香” – C用不同的方式证明了这句话。C语言于1970年代后期被引入，为编程世界做出了巨大贡献。C是少数几种语言的母语。有些是从C派生的，或者是从其语法，构造和范例（包括Java，Objective-C和C#）启发而来的。

即使在当今，可以看出，每当需要构建高性能应用程序时，C仍然是最受欢迎的选择。Linux OS是基于C的。CPP是C的混合版本。C ++是一种基于C的面向对象的编程语言。因此，在设计更高级别的应用程序时，它比其他方法更可取。

C ++比动态类型的语言具有更好的性能，因为在真正执行代码之前先对代码进行类型检查。开发的核心领域是虚拟现实，游戏，计算机图形等。

5、PHP

这个事实会让你感到非常惊奇，这种语言是为维护Rasmus的个人主页（PHP）而创建的，实际上到今天已占据了全球83%的网站。PHP代表超文本预处理器，是一种通用编程语言。显然，PHP是一种脚本语言，可在服务器上运行，并且用于创建以HTML编写的网页。它之所以受欢迎，是因为它免费，而且易于设置并且易于新程序员使用。

对于全球的Web开发人员来说，PHP是一个非常强大的选择。它被广泛用于创建动态网页内容以及网站上使用的图像。由于使用范围广泛，因此排名第五。另外，PHP可以很好地用于WordPress CMS（内容管理系统）。

它位于第五的原因之一，是英文PHP降低了网站性能并影响了加载时间。（无奈）

6、Swift

接下来是Swift。Swift就像它的名字一样流畅，是Apple Inc.开发的一种通用、开放源代码的、已编译的编程语言。如果你正在寻找针对本机iOS或Mac OS应用程序的开发，则Swift就是首选。Swift受Python和Ruby的影响很深，并且被设计为对初学者友好且易于使用。与它的前一个Objective-C相比，Swift被认为是一种更快，更安全，更易于阅读和调试的工具。与Objective-C不同，Swift需要更少的代码，类似于自然的英语。因此，来自JavaScript, Java, Python, C#和C ++的现有技术人员可以更轻松地切换到Swift。

除此之外，人才储备有限是它面临的一个挑战。与其他开源语言相比，你周围可能找不到很多Swift开发人员。最近的调查表明，在78,000名受访者中，只有8.1%的人使用Swift，这比其他人要少。并且由于频繁的更新，Swift被认为在每个新版本中都不太稳定。

7、C# (C-shap)

C-sharp是Microsoft 2000年开发的功能强大的面向对象的编程语言。C-sharp用于开发桌面应用程序和最近的Windows 8/10应用程序，并且需要.NET框架来运行。微软开发了C#作为Java的竞争对手。实际上，Sun不想让微软的干扰来改变Java，于是C #诞生了。

C#具有多种功能，使初学者更容易学习。与C ++相比，代码是一致且合乎逻辑的。由于C#是静态类型的语言，因此在C#中发现错误很容易，因为在将代码转到应用程序之前会先检查代码。

简而言之，它是开发Web应用程序、桌面应用程序的完美选择，并且在VR, 2D和3D游戏中也得到了证明。像Xamarin这样的跨平台工具已经用C#编写，使其与所有设备兼容。

8、Ruby

一种开源的动态编程语言，着重简单性和生产率，于1990年中在日本开发。它的设计主题是简化编程环境并增加乐趣。Ruby在全栈Web框架Ruby on Rails框架中流行。Ruby具有动态类型化的语言，它没有硬性规定，并且是一种高级语言，在很大程度上类似于英语。

简而言之，你可以使用更少的代码来构建应用程序。但是Ruby面临的挑战是动态类型化的语言，它不容易维护，并且灵活性使其运行缓慢。

9、Objective-C

Objective-C (ObjC) 是一种面向对象的编程语言。Apple将其用于OS X和iOS操作系统及其应用程序编程接口 (API)。它开发于1980年代，并在某些最早的操作系统中得到使用。Objective-C是面向对象的通用对象。你可以将其称为混合C，因为它为C编程语言添加了功能。

SQL (es-que-el) 代表结构化查询语言，是一种用于操作数据库的编程语言。它包括存储，处理和检索存储在关系数据库中的数据。SQL保持数据的准确性和安全性，并且无论其大小如何，都有助于维护数据库的完整性。

今天，SQL已在Web框架和数据库应用程序中使用。如果你精通SQL，则可以更好地掌握数据探索和有效的决策制定。

如果你打算选择数据库管理作为你的职业，请首先使用C或C ++。SQL开发人员的需求量很大，而且薪水也不低。

- END -

推荐阅读

1. 警告！你的隐私正在被上亿网友围观偷看！
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 

[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！

什么是 Java 对象深拷贝？面试必问！

吴大山 Java后端 2019-11-08

点击上方 Java后端, 选择 [设为星标](#)

技术博文, 及时送达

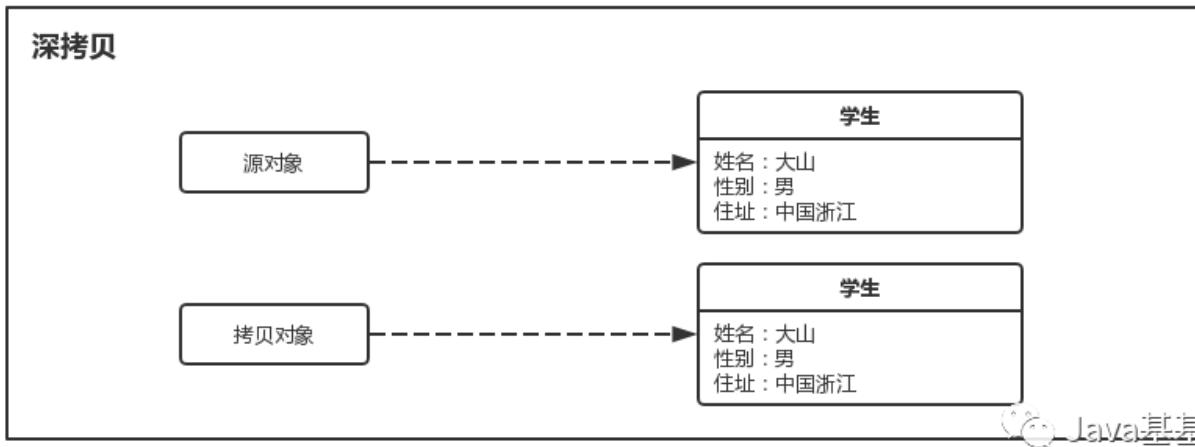
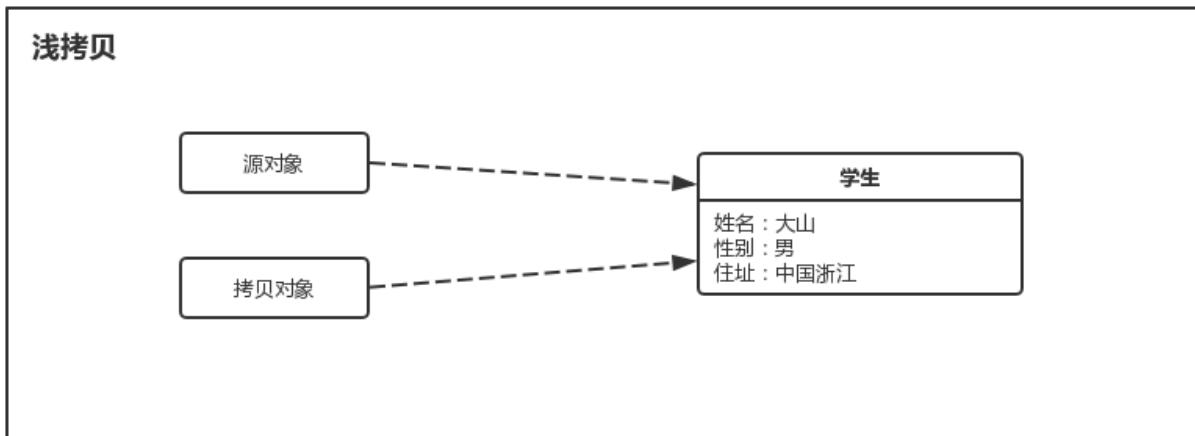
作者 | 吴大山

编辑 | Java基基

链接 | wudashan.com/2018/10/14/Java-Deep-Copy

介绍

在Java语言里，当我们需要拷贝一个对象时，有两种类型的拷贝：浅拷贝与深拷贝。浅拷贝只是拷贝了源对象的地址，所以源对象的值发生变化时，拷贝对象的值也会发生变化。而深拷贝则是拷贝了源对象的所有值，所以即使源对象的值发生变化时，拷贝对象的值也不会改变。如下图描述：



了解了浅拷贝和深拷贝的区别之后，本篇博客将教大家几种深拷贝的方法。

拷贝对象

首先，我们定义一下需要拷贝的简单对象。

```

/**
 * 用户
 */
public class User {

    private String name;
    private Address address;

    // constructors, getters and setters

}

/**
 * 地址
 */
public class Address {

    private String city;
    private String country;

    // constructors, getters and setters

}

```

如上述代码，我们定义了一个User用户类，包含name姓名，和address地址，其中address并不是字符串，而是另一个Address类，包含country国家和city城市。构造方法和成员变量的get()、set()方法此处我们省略不写。接下来我们将详细描述如何深拷贝User对象。

方法一 构造函数

我们可以通过在调用构造函数进行深拷贝，形参如果是基本类型和字符串则直接赋值，如果是对象则重新new一个。

测试用例

```

@Test
public void constructorCopy() {

    Address address = new Address("杭州", "中国");
    User user = new User("大山", address);

    // 调用构造函数时进行深拷贝
    User copyUser = new User(user.getName(), new Address(address.getCity(), address.getCountry()));

    // 修改源对象的值
    user.getAddress().setCity("深圳");

    // 检查两个对象的值不同
    assertNotSame(user.getAddress().getCity(), copyUser.getAddress().getCity());
}

```

方法二 重载clone()方法

Object父类有个clone()的拷贝方法，不过它是protected类型的，我们需要重写它并修改为public类型。除此之外，子类还需要实现Cloneable接口来告诉JVM这个类是可以拷贝的。

Tips：关注微信公众号：Java后端，获取每日技术博文推送。

重写代码

让我们修改一下User类，Address类，实现Cloneable接口，使其支持深拷贝。

```
/*
 * 地址
 */
public class Address implements Cloneable {

    private String city;
    private String country;

    // constructors, getters and setters

    @Override
    public Address clone() throws CloneNotSupportedException {
        return (Address) super.clone();
    }

}

/*
 * 用户
 */
public class User implements Cloneable {

    private String name;
    private Address address;

    // constructors, getters and setters

    @Override
    public User clone() throws CloneNotSupportedException {
        User user = (User) super.clone();
        user.setAddress(this.address.clone());
        return user;
    }

}
```

需要注意的是，super.clone()其实是浅拷贝，所以在重写User类的clone()方法时，address对象需要调用address.clone()重新赋值。

测试用例

```
@Test
public void cloneCopy() throws CloneNotSupportedException {

    Address address = new Address("杭州", "中国");
    User user = new User("大山", address);

    // 调用clone()方法进行深拷贝
    User copyUser = user.clone();

    // 修改源对象的值
    user.getAddress().setCity("深圳");

    // 检查两个对象的值不同
    assertNotSame(user.getAddress().getCity(), copyUser.getAddress().getCity());

}
```

方法三 Apache Commons Lang序列化

Java提供了序列化的能力，我们可以先将源对象进行序列化，再反序列化生成拷贝对象。但是，使用序列化的前提是拷贝的类（包括其成员变量）需要实现Serializable接口。Apache Commons Lang包对Java序列化进行了封装，我们可以直接使用它。

重写代码

让我们修改一下User类，Address类，实现Serializable接口，使其支持序列化。

```
/*
 * 地址
 */
public class Address implements Serializable {

    private String city;
    private String country;

    // constructors, getters and setters

}

/*
 * 用户
 */
public class User implements Serializable {

    private String name;
    private Address address;

    // constructors, getters and setters

}
```

测试用例

```
@Test
public void serializableCopy() {

    Address address = new Address("杭州", "中国");
    User user = new User("大山", address);

    // 使用Apache Commons Lang序列化进行深拷贝
    User copyUser = (User) SerializationUtils.clone(user);

    // 修改源对象的值
    user.getAddress().setCity("深圳");

    // 检查两个对象的值不同
    assertNotSame(user.getAddress().getCity(), copyUser.getAddress().getCity());

}
```

方法四 Gson序列化

Gson可以将对象序列化成JSON，也可以将JSON反序列化成对象，所以我们可以用它进行深拷贝。

测试用例

```
@Test  
public void gsonCopy() {
```

```
    Address address = new Address("杭州", "中国");  
    User user = new User("大山", address);  
  
    // 使用Gson序列化进行深拷贝  
    Gson gson = new Gson();  
    User copyUser = gson.fromJson(gson.toJson(user), User.class);  
  
    // 修改源对象的值  
    user.getAddress().setCity("深圳");  
  
    // 检查两个对象的值不同  
    assertNotSame(user.getAddress().getCity(), copyUser.getAddress().getCity());  
}
```

方法五 Jackson序列化

Jackson与Gson相似，可以将对象序列化成JSON，明显不同的地方是拷贝的类（包括其成员变量）需要有默认的无参构造函数。

重写代码**

让我们修改一下User类，Address类，实现默认的无参构造函数，使其支持Jackson。

```
/**  
 * 用户  
 */  
public class User {  
  
    private String name;  
    private Address address;  
  
    // constructors, getters and setters  
  
    public User() {  
    }  
  
}  
  
/**  
 * 地址  
 */  
public class Address {  
  
    private String city;  
    private String country;  
  
    // constructors, getters and setters  
  
    public Address() {  
    }  
  
}
```

测试用例

```
@Test  
public void jacksonCopy() throws IOException {
```

```
    Address address = new Address("杭州", "中国");  
    User user = new User("大山", address);  
  
    // 使用Jackson序列化进行深拷贝  
    ObjectMapper objectMapper = new ObjectMapper();  
    User copyUser = objectMapper.readValue(objectMapper.writeValueAsString(user), User.class);  
  
    // 修改源对象的值  
    user.getAddress().setCity("深圳");  
  
    // 检查两个对象的值不同  
    assertNotSame(user.getAddress().getCity(), copyUser.getAddress().getCity());  
}
```

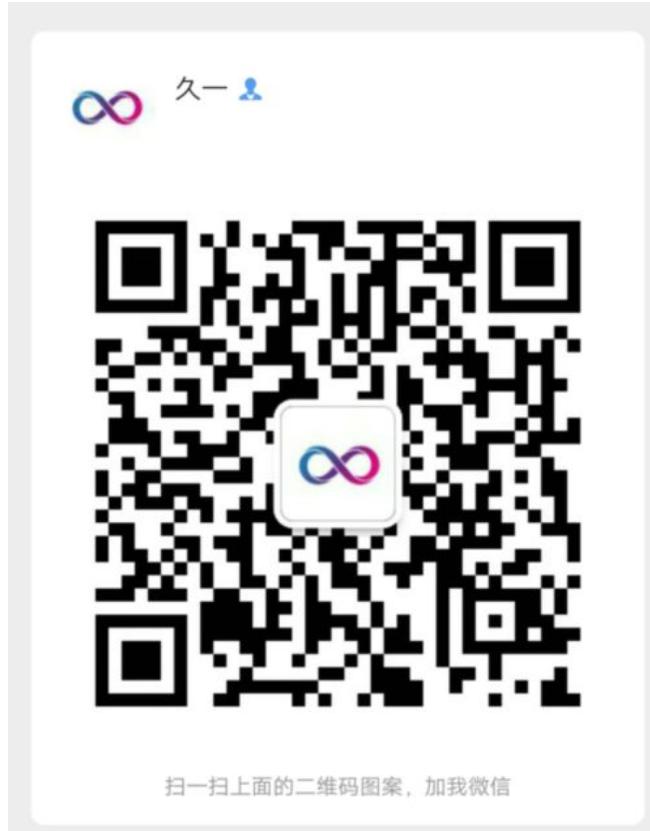
总结

说了这么多深拷贝的实现方法，哪一种方法才是最好的呢？最简单的判断就是根据拷贝的类（包括其成员变量）是否提供了深拷贝的构造函数、是否实现了Cloneable接口、是否实现了Serializable接口、是否实现了默认的无参构造函数来进行选择。如果需要详细的考虑，则可以参考下面的表格：

深拷贝方法	优点	缺点
构造函数	1. 底层实现简单 2. 不需要引入第三方包 3. 系统开销小 4. 对拷贝类没有要求，不需要实现额外接口和方法	1. 可用性差，每次新增成员变量都需要新增新的拷贝构造函数
重载clone()方法	1. 底层实现较简单 2. 不需要引入第三方包 3. 系统开销小	1. 可用性较差，每次新增成员变量可能需要修改clone()方法 2. 拷贝类（包括其成员变量）需要实现Cloneable接口
Apache Commons Lang序列化	1. 可用性强，新增成员变量不需要修改拷贝方法	1. 底层实现较复杂 2. 需要引入Apache Commons Lang第三方JAR包 3. 拷贝类（包括其成员变量）需要实现Serializable接口 4. 序列化与反序列化存在一定的系统开销
Gson序列化	1. 可用性强，新增成员变量不需要修改拷贝方法 2. 对拷贝类没有要求，不需要实现额外接口和方法	1. 底层实现复杂 2. 需要引入Gson第三方JAR包 3. 序列化与反序列化存在一定的系统开销
Jackson序列化	1. 可用性强，新增成员变量不需要修改拷贝方法	1. 底层实现复杂 2. 需要引入Jackson第三方JAR包 3. 拷贝类（包括其成员变量）需要实现默认的无参构造函数 4. 序列化与反序列化存在一定的系统开销

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. [你写的 Java 代码是如何一步步输出结果的？](#)
2. [IntelliJ IDEA 详细图解最常用的配置，新人收藏](#)
3. [Maven 实战问题和最佳实践](#)
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 

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!

今天聊聊 Java 序列化

原创 NYfor2020 Java后端 4天前

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

在开发过程中经常会对实体进行序列化，但其实我们只是在“只知其然，不知其所以然”的状态，很多时候会有这些问题：

- 什么是序列化和反序列化？为什么要序列化？
- 怎么实现序列化？
- 序列化的原理是什么呢？
- transient关键字
- 序列化时应注意什么？

如果你也有这些疑问，不妨看看本文？

1. 什么是序列化和反序列化？

Java序列化是指把Java对象转换为字节序列的过程；

Java反序列化是指把字节序列恢复为Java对象的过程；

2. 为什么要序列化？

其实我们的对象不只是存储在内存中，它还需要在传输网络中进行传输，并且保存起来之后下次再加载出来，这时候就需要序列化技术。

- 一般Java对象的生命周期比Java虚拟机端，而实际开发中如果需要JVM停止后能够继续持有对象，则需要用到序列化技术将对象持久化到磁盘或数据库。
- 在多个项目进行RPC调用时，需要在网络上传输JavaBean对象，而网络上只允许二进制形式的数据进行传输，这时则需要用到序列化技术。

Java的序列化技术就是把**对象转换成一串由二进制字节组成的数组**，然后将这二进制数据保存在磁盘或传输网络。而后需要用到对对象时，磁盘或者网络接收者可以通过反序列化得到此对象，达到**对象持久化**的目的。

3. 怎么实现序列化？

序列化的过程一般会是这样的：

- 将对象实例相关的**类元数据**输出
- **递归地**输出类的超类描述，直到没有超类
- 类元数据输出之后，开始从最顶层的超类输出对象实例的**实际数据值**
- 从上至下**递归**输出实例的数据

所以，如果父类已经序列化了，子类继承之后也可以进行序列化。

实现第一步，则需要先将对象实例相关的类标记为需要序列化。

实现序列化的要求：目标对象实现Serializable接口

我们先创建一个NY类，实现Serializable接口，并生成一个版本号：

```
public class NY implements Serializable {  
    private static final long serialVersionUID = 8891488565683643643L;  
    private String name;  
    private String blogName;  
  
    @Override  
    public String toString() {  
        return "NY{" +  
            "name='" + name + '\'' +  
            ", blogName='" + blogName + '\'' +  
            '}';  
    }  
}
```

在这里，Serializable接口的作用只是标识这个类是需要进行序列化的，而且Serializable接口中并没有提供任何方法。而且serialVersionUID序列化版本号的作用是用来区分我们所编写的类的版本，用于反序列化时确定版本。

JDK类库中序列化和反序列化API

java.io.ObjectInputStream: 对象输入流

该类中的readObject()方法从输入流中读取字节序列，然后将字节序列反序列化为一个对象并返回。

java.io.ObjectOutputStream: 对象输出流

该类的writeObject()方法将传入的obj对象进行序列化，把得到的字节序列写入到目标输出流中进行输出。

结合上面的NY类，我们来看看使用JDK类库中的API怎么实现序列化和反序列化：

```
public class SerializeNY {  
    public static void main(String[] args) throws IOException, ClassNotFoundException {  
        serializeNY();  
        NY ny = deserializeNY();  
        System.out.println(ny.toString());  
    }  
  
    private static void serializeNY() throws IOException {  
        NY ny = new NY();  
        ny.setName("NY");  
        ny.setBlogName("NYfor2020");  
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(new File("D:\\\\serializable.txt")));  
        oos.writeObject(ny);  
        System.out.println("NY 对象序列化成功!");  
        oos.close();  
    }  
  
    private static NY deserializeNY() throws IOException, ClassNotFoundException {  
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(new File("D:\\\\serializable.txt")));  
        NY ny = (NY) ois.readObject();  
        System.out.println("NY 对象反序列化成功");  
        return ny;  
    }  
}
```

运行结果为：

```
NY 对象序列化成功!
NY 对象反序列化成功
NY{name='NY', blogName='NYfor2020'}
```

可以看到，这整个过程简单来说就是把对象存在磁盘，然后再从磁盘读出来。

但是我们平时看到序列化的实体中的serialVersionUID，为什么有的是1L，有的是一长串数字？

上面我们提到serialVersionUID作用是用来区分类的版本，所以无论是1L还是一长串数字，都是用来确认版本的。如果序列化的类版本改变，则在反序列化的时候就会报错。

举个栗子，刚刚我们已经在磁盘中生成了NY对象的序列化文件，如果我们对NY类的serialVersionUID稍作改动，改成：

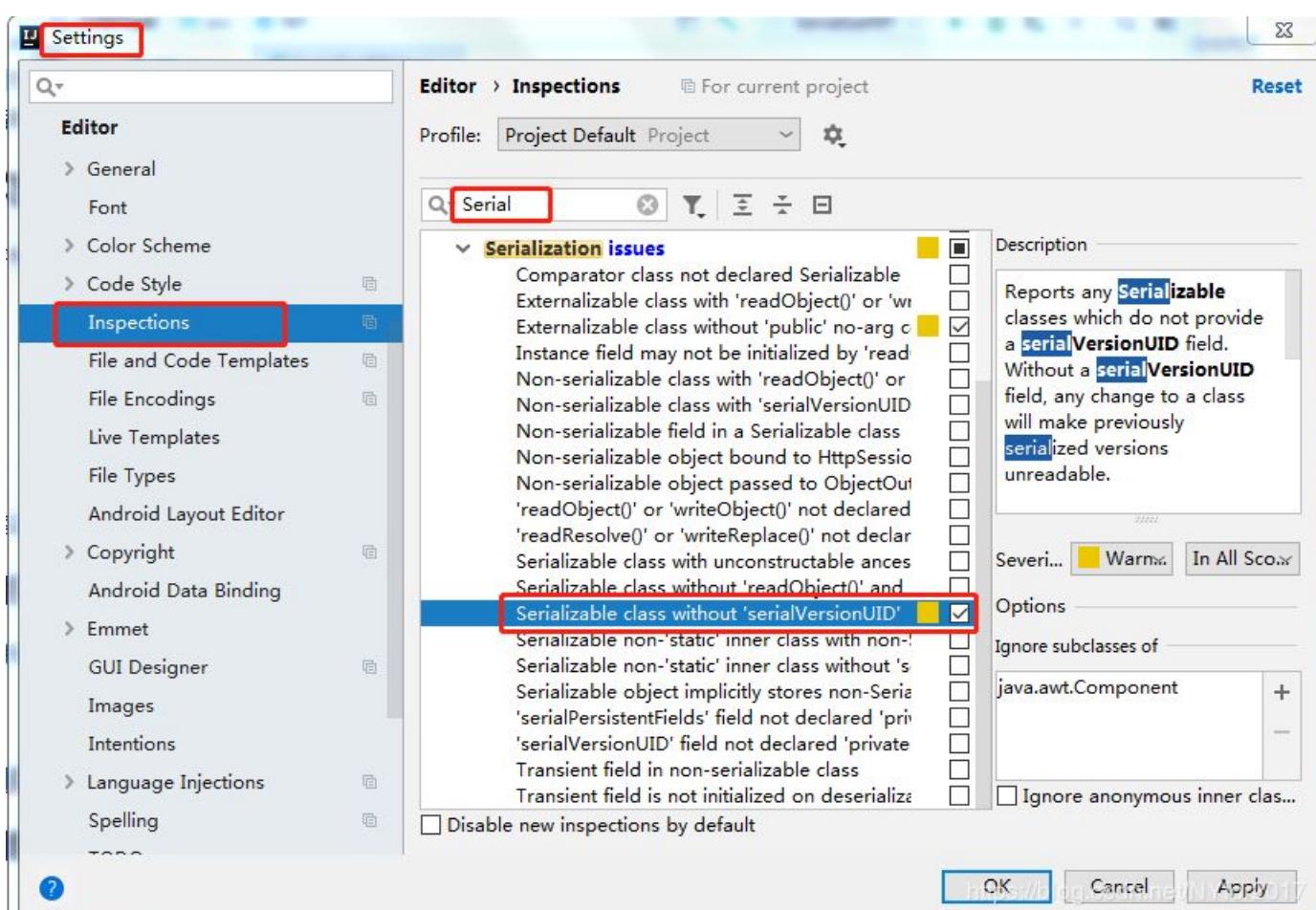
```
private static final long serialVersionUID = 8891488565683643643L; //将末尾的2改成3
```

再执行一次反序列化方法，运行结果如下：

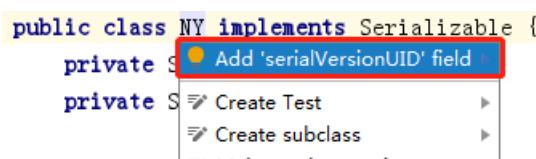
```
Exception in thread "main" java.io.InvalidClassException: NY; local class incompatible: stream classdesc serialVersionUID = 88914885656
```

.....

至于怎么让idea生成serialVersionUID，则需要在idea设置中改个配置即可：



之后再使用"Alt+Enter"键即可调出下图选项：



序列化的原理是什么呢？

既然知道了序列化是怎么使用的，那么序列化的原理是怎样的呢？

我们用上面的例子来作为探寻序列化原理的入口：

```
private static void serializeNY() throws IOException {
    NY ny = new NY();
    ny.setName("NY");
    ny.setBlogName("NYfor2020");
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(new File("D:\\Serializable.txt")));
    oos.writeObject(ny);
    System.out.println("NY 对象序列化成功! ");
    oos.close();
}
```

1. 进入ObjectOutputStream的构造函数：

```
public ObjectOutputStream(OutputStream out) throws IOException {
    verifySubclass();

    bout = new BlockDataOutputStream(out);
    handles = new HandleTable(10, (float) 3.00);
    subs = new ReplaceTable(10, (float) 3.00);
    enableOverride = false;

    writeStreamHeader();

    bout.setBlockDataMode(true);
    if (extendedDebugInfo) {
        debugInfoStack = new DebugTraceInfoStack();
    } else {
        debugInfoStack = null;
    }
}
```

我们进入writeStreamHeader()方法：

```
protected void writeStreamHeader() throws IOException {
    bout.writeShort(STREAM_MAGIC);
    bout.writeShort(STREAM_VERSION);
}
```

这个方法是将序列化文件的魔数和版本写入序列化文件头：

```
final static short STREAM_MAGIC = (short)0xaced;
final static short STREAM_VERSION = 5;
```

2. 在 writeObject() 方法进行具体的序列化写入操作：

```

public final void writeObject(Object obj) throws IOException {
    if (enableOverride) {
        writeObjectOverride(obj);
        return;
    }
    try {
        writeObject0(obj, false);
    } catch (IOException ex) {
        if (depth == 0) {
            writeFatalException(ex);
        }
        throw ex;
    }
}

```

进入writeObject0()方法：

```

private void writeObject0(Object obj, boolean unshared)
    throws IOException
{
    boolean oldMode = bout.setBlockDataMode(false);
    depth++;
    try {

        int h;
        if ((obj = subs.lookup(obj)) == null) {
            writeNull();
            return;
        } else if (!unshared && (h = handles.lookup(obj)) != -1) {
            writeHandle(h);
            return;
        } else if (obj instanceof Class) {
            writeClass((Class) obj, unshared);
            return;
        } else if (obj instanceof ObjectStreamClass) {
            writeClassDesc((ObjectStreamClass) obj, unshared);
            return;
        }
    }

    Object orig = obj;

    Class<?> cl = obj.getClass();
    ObjectStreamClass desc;
    for (;;) {

        Class<?> repCl;

        desc = ObjectStreamClass.lookup(cl, true);

        if (!desc.hasWriteReplaceMethod() ||
            (obj = desc.invokeWriteReplace(obj)) == null ||
            (repCl = obj.getClass()) == cl)
        {
            break;
        }
        cl = repCl;
    }
}

```

```

if (enableReplace) {
    Object rep = replaceObject(obj);
    if (rep != obj && rep != null) {
        cl = rep.getClass();
        desc = ObjectStreamClass.lookup(cl, true);
    }
    obj = rep;
}

if (obj != orig) {
    subs.assign(orig, obj);
    if (obj == null) {
        writeNull();
        return;
    } else if (!unshared && (h = handles.lookup(obj)) != -1) {
        writeHandle(h);
        return;
    } else if (obj instanceof Class) {
        writeClass((Class) obj, unshared);
        return;
    } else if (obj instanceof ObjectStreamClass) {
        writeClassDesc((ObjectStreamClass) obj, unshared);
        return;
    }
}
}

if (obj instanceof String) {
    writeString((String) obj, unshared);
} else if (cl.isArray()) {
    writeArray(obj, desc, unshared);
} else if (obj instanceof Enum) {
    writeEnum((Enum<?>) obj, desc, unshared);
} else if (obj instanceof Serializable) {

    writeOrdinaryObject(obj, desc, unshared);
} else {
    if (extendedDebugInfo) {
        throw new NotSerializableException(
            cl.getName() + "\n" + debugInfoStack.toString());
    } else {
        throw new NotSerializableException(cl.getName());
    }
}
} finally {

    depth--;
    bout.setBlockDataMode(oldMode);
}
}

```

这一段代码中创建了**ObjectStreamClass**对象，并根据不同的对象类型来执行不同的写入操作。而在此例子中，对象对应的类实现了**Serializable**接口，所以下一步会执行**writeOrdinaryObject()**方法。

writeOrdinaryObject()是当对象对应的类实现了**Serializable**接口的时才会被调用：

```

private void writeOrdinaryObject(Object obj,
                                ObjectStreamClass desc,
                                boolean unshared)
throws IOException
{
    if (extendedDebugInfo) {
        debugInfoStack.push(
            (depth == 1 ? "root " : "") + "object (class '" +
            obj.getClass().getName() + "', " + obj.toString() + ")");
    }
    try {
        desc.checkSerialize();

        bout.writeByte(TC_OBJECT);

        writeClassDesc(desc, false);
        handles.assign(unshared ? null : obj);
        if (desc.isExternalizable() && !desc.isProxy()) {
            writeExternalData((Externalizable) obj);
        } else {
            writeSerialData(obj, desc);
        }
    } finally {
        if (extendedDebugInfo) {
            debugInfoStack.pop();
        }
    }
}

```

接下来是将类的描述写入类元数据中的**writeClassDesc()**:

```

private void writeClassDesc(ObjectStreamClass desc, boolean unshared)
throws IOException
{
    int handle;
    if (desc == null) {

        writeNull();
    } else if (!unshared && (handle = handles.lookup(desc)) != -1) {
        writeHandle(handle);
    } else if (desc.isProxy()) {
        writeProxyDesc(desc, unshared);
    } else {
        writeNonProxyDesc(desc, unshared);
    }
}

```

在desc为null时，会执行**writeNull()**方法：

```

private void writeNull() throws IOException {
    bout.writeByte(TC_NULL);
}

final static byte TC_NULL = (byte)0x70;

```

可以看到，在**writeNull()**中，会将表示NULL的标识写入序列中。

那么如果desc不为null时，一般执行writeNonProxyDesc()方法：

```
private void writeNonProxyDesc(ObjectStreamClass desc, boolean unshared)
    throws IOException
{
    bout.writeByte(TC_CLASSDESC);
    handles.assign(unshared ? null : desc);

    if (protocol == PROTOCOL_VERSION_1) {

        desc.writeNonProxy(this);
    } else {

        writeClassDescriptor(desc);
    }

    Class<?> cl = desc.forClass();
    bout.setBlockDataMode(true);
    if (cl != null && isCustomSubclass()) {
        ReflectUtil.checkPackageAccess(cl);
    }

    annotateClass(cl);
    bout.setBlockDataMode(false);

    bout.writeByte(TC_ENDBLOCKDATA);
}

    writeClassDesc(desc.getSuperDesc(), false);
}
```

在上一个方法执行过程中，会执行writeClassDescriptor()方法将类的描述写入类元数据中：

```
protected void writeClassDescriptor(ObjectStreamClass desc)
    throws IOException{
    desc.writeNonProxy(this);
}
```

在这里我们可以看到，写入类元信息的方法调用了writeNonProxy()方法：

```

void writeNonProxy(ObjectOutputStream out) throws IOException {
    out.writeUTF(name);
    out.writeLong(getSerialVersionUID());

    byte flags = 0;
    if (externalizable) {
        flags |= ObjectStreamConstants.SC_EXTERNALIZABLE;
        int protocol = out.getProtocolVersion();
        if (protocol != ObjectStreamConstants.PROTOCOL_VERSION_1) {
            flags |= ObjectStreamConstants.SC_BLOCK_DATA;
        }
    } else if (serializable) {

        flags |= ObjectStreamConstants.SC_SERIALIZABLE;
    }
    if (hasWriteObjectData) {

        flags |= ObjectStreamConstants.SC_WRITE_METHOD;
    }
    if (isEnum) {

        flags |= ObjectStreamConstants.SC_ENUM;
    }

    out.writeByte(flags);

    out.writeShort(fields.length);
    for (int i = 0; i < fields.length; i++) {
        ObjectStreamField f = fields[i];

        out.writeByte(f.getTypeCode());

        out.writeUTF(f.getName());
        if (!f.isPrimitive()) {

            out.writeTypeString(f.getTypeString());
        }
    }
}

```

这次方法中我们可以看到：

1. 调用writeUTF()方法将对象所属类的名字写入。
2. 调用writeLong()方法将类的序列号serialVersionUID写入。
3. 判断被序列化对象所属类的流类型flag，写入底层字节容器中（占两个字节）。
4. 写入对象中的所有字段，以及对应的属性

所以直到这个方法的执行，一个对象及其对应类的所有属性和属性值才被序列化。当上述流程完成之后，回到writeOrdinaryObject()方法中，继续往下运行：

```
private void writeOrdinaryObject(Object obj, ObjectStreamClass desc, boolean unshared)
throws IOException{
...
    writeClassDesc(desc, false);

handles.assign(unshared ? null : obj);
if (desc.isExternalizable() && !desc.isProxy()) {
    writeExternalData((Externalizable) obj);
} else {
    writeSerialData(obj, desc);
}
} finally {
    if (extendedDebugInfo) {
        debugInfoStack.pop();
    }
}
}
```

调用writeSerialData()方法将实例化数据写入：

```
private void writeSerialData(Object obj, ObjectStreamClass desc)
    throws IOException
{
    ObjectStreamClass.ClassDataSlot[] slots = desc.getClassDataLayout();
    for (int i = 0; i < slots.length; i++) {
        ObjectStreamClass slotDesc = slots[i].desc;

        if (slotDesc.hasWriteObjectMethod()) {
            PutFieldImpl oldPut = curPut;
            curPut = null;
            SerialCallbackContext oldContext = curContext;

            if (extendedDebugInfo) {
                debugInfoStack.push(
                    "custom writeObject data (class '" +
                    slotDesc.getName() + "')";
            }
            try {
                curContext = new SerialCallbackContext(obj, slotDesc);
                bout.setBlockDataMode(true);
                slotDesc.invokeWriteObject(obj, this);
                bout.setBlockDataMode(false);
                bout.writeByte(TC_ENDBLOCKDATA);
            } finally {
                curContext.setUsed();
                curContext = oldContext;
                if (extendedDebugInfo) {
                    debugInfoStack.pop();
                }
            }
        }
        curPut = oldPut;
    } else {
        defaultWriteFields(obj, slotDesc);
    }
}
}
```

当执行到defaultWriteFields()方法时，会将实例数据写入：

```

private void defaultWriteFields(Object obj, ObjectStreamClass desc)
    throws IOException
{
    Class<?> cl = desc.forClass();
    if (cl != null && obj != null && !cl.isInstance(obj)) {
        throw new ClassCastException();
    }

    desc.checkDefaultSerialize();

    int primDataSize = desc.getPrimDataSize();
    if (primVals == null || primVals.length < primDataSize) {
        primVals = new byte[primDataSize];
    }

    desc.getPrimFieldValues(obj, primVals);

    bout.write(primVals, 0, primDataSize, false);

    ObjectStreamField[] fields = desc.getFields(false);
    Object[] objVals = new Object[desc.getNumObjFields()];
    int numPrimFields = fields.length - objVals.length;

    desc.getObjFieldValues(obj, objVals);

    for (int i = 0; i < objVals.length; i++) {
        if (extendedDebugInfo) {
            debugInfoStack.push(
                "field (class '" + desc.getName() + "', name: '" +
                fields[numPrimFields + i].getName() + "', type: '" +
                fields[numPrimFields + i].getType() + "');");
        }
        try {
            writeObject0(objVals[i],
                        fields[numPrimFields + i].isUnshared());
        } finally {
            if (extendedDebugInfo) {
                debugInfoStack.pop();
            }
        }
    }
}

```

在执行完上述方法之后，程序将会回到**writeNonProxyDesc()**方法中，并且在**writeClassDesc()**中会将对象对应的类的父类信息进行写入：

```

private void writeNonProxyDesc(ObjectStreamClass desc, boolean unshared)
    throws IOException
{
    ...

    writeClassDescriptor(desc);

    }

    Class<?> cl = desc.forClass();
    bout.setBlockDataMode(true);
    if (cl != null && isCustomSubclass()) {
        ReflectUtil.checkPackageAccess(cl);
    }

    annotateClass(cl);
    bout.setBlockDataMode(false);

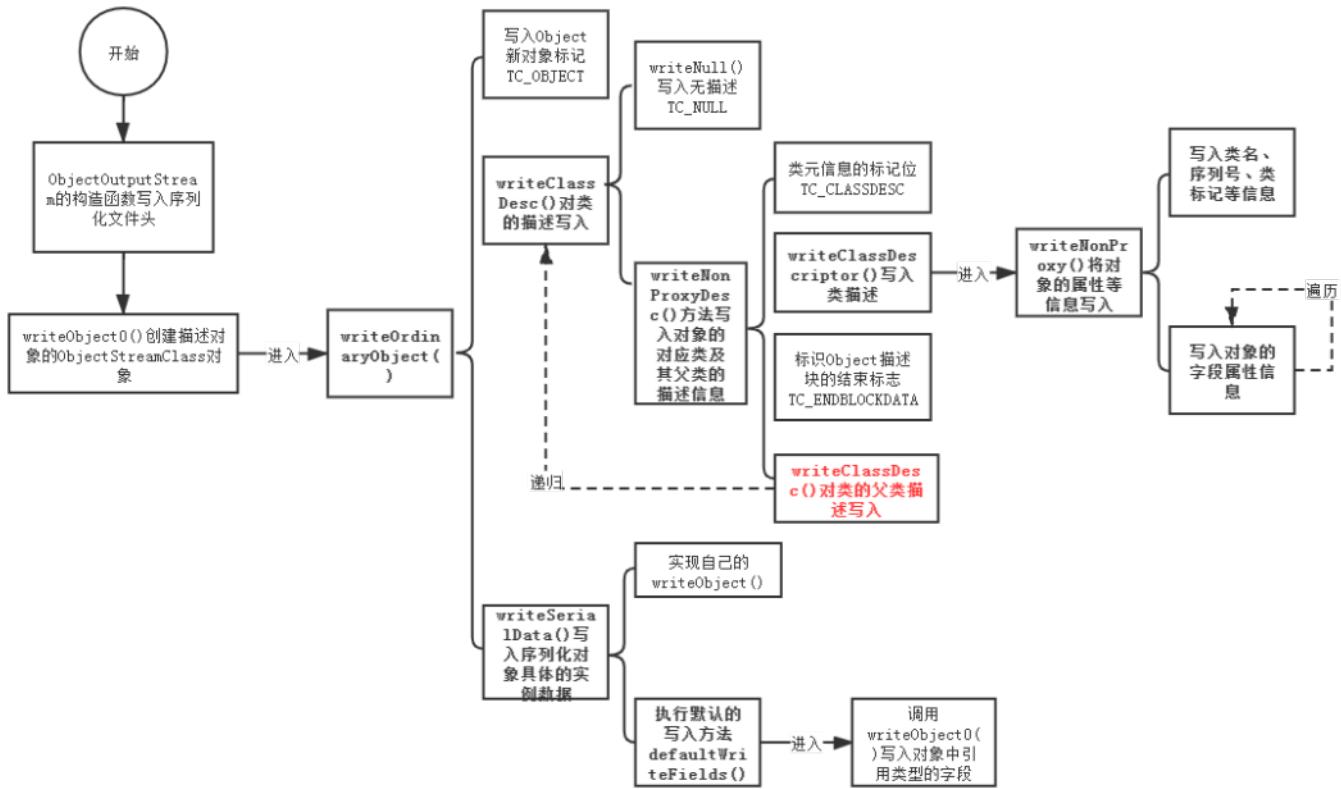
    bout.writeByte(TC_ENDBLOCKDATA);

    writeClassDesc(desc.getSuperDesc(), false);
}

```

至此，我们可以知道，整个序列化的过程其实就是一个递归写入的过程。

将上面的过程进行简化，可以总结为这幅图：



transient关键字

在有些时候，我们并不想将一些敏感信息序列化，如密码等，这个时候就需要**transient关键字**来标注属性为非序列化属性。

transient关键字的使用

将上面的NY类中的name属性稍作修改：

```
private transient String name;
```

当我们再次运行SerializeNY类中的main()方法时，运行结果如下：

```
NY 对象序列化成功!  
NY 对象反序列化成功  
NY{name='null', blogName='NYfor2020'}
```

我们可以看到，name属性为null，说明反序列化时根本没有从文件中获取到信息。

transient关键字的特点

变量一旦被transient修饰，则不再是对象持久化的一部分了，而且变量内容在反序列化时也不能获得。

transient关键字只能修饰变量，而不能修饰方法和类，而且本地变量是不能被transient修饰的，如果变量是类变量，则需要该类也实现Serializable接口。

一个静态变量不管是否被transient修饰，都不会被序列化。

关于这一点，可能会有读者感到疑惑。举个栗子，如果用static修饰NY类中的name：

```
private static String name;
```

运行SerializeNY类中的main程序，可以看到运行结果：

```
NY 对象序列化成功!  
NY 对象反序列化成功  
NY{name='NY', blogName='NYfor2020'}
```

嘶…这是翻车了吗？并没有，因为这里出现的name值是当前JVM中对应的static变量值，这个值是JVM中的而不是反序列化得出的。

不信？我们来改变一下SerializeNY类中的serializeNY()函数

```
private static void serializeNY() throws IOException {  
    NY ny = new NY();  
    ny.setName("NY");  
    ny.setBlogName("NYfor2020");  
    ny.setTest("12");  
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(new File("D:\\\\serializable.txt")));  
    oos.writeObject(ny);  
    System.out.println("NY 对象序列化成功！ ");  
    System.out.println(ny.toString());  
    oos.close();  
    ny.setName("hey, NY");  
}
```

笔者在NY对象被序列化之后，改变了NY对象的name值。运行结果为：

```
NY 对象序列化成功!  
NY{name='NY', blogName='NYfor2020'}  
NY 对象反序列化成功  
NY{name='hey, NY', blogName='NYfor2020'}
```

transient修饰的变量真的就不能被序列化了吗？

举个栗子：

```
public class ExternalizableTest implements Externalizable {  
  
    private transient String content = "即使被transient修饰，我也会序列化";  
  
    @Override  
    public void writeExternal(ObjectOutput out) throws IOException {  
        out.writeObject(content);  
    }  
  
    @Override  
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  
        content = (String)in.readObject();  
    }  
  
    public static void main(String[] args) throws IOException, ClassNotFoundException {  
        ExternalizableTest et = new ExternalizableTest();  
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(new File("D:\\\\externalizable.txt")));  
        oos.writeObject(et);  
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(new File("D:\\\\externalizable.txt")));  
        et = (ExternalizableTest) ois.readObject();  
        System.out.println(et.content);  
        oos.close();  
        ois.close();  
    }  
}
```

运行结果为：

即使被transient修饰，我也会序列化

我们可以看到，content变量在被transient修饰的情况下，还是被序列化了。因为在Java中，对象序列化可以通过实现两种接口来实现：

- 如果实现的是**Serializable**接口，则所有信息（不包括被static、transient修饰的变量信息）的序列化将**自动**进行。
- 如果实现的是**Externalizable**接口，则不会进行自动序列化，需要开发者在**writeExternal()**方法中**手工**指定需要序列化的变量，与是否被transient修饰无关。

序列化注意事项

- 序列化对象必须实现序列化接口**Serializable**。
- 序列化对象中的属性如果也有对象的话，其对象需要实现序列化接口。
- 类的对象序列化后，类的序列号不能轻易更改，否则反序列化会失败。
- 类的对象序列化后，类的属性增加或删除不会影响序列化，只是值会丢失。
- 如果父类序列化，子类会继承父类的序列化；如果父类没序列化，子类序列化了，子类中的属性能正常序列化，但父类的属性会丢失，不能序列化。
- 用Java序列化的二进制字节数据只能由Java反序列化，如果要转换成其他语言反序列化，则需要先转换成Json/XML通用格式的数据。
- 如果某个字段不想序列化，在该字段前加上**transient**关键字即可。（咳咳，下一篇就是写这个了，敬请关注~）

第一次写关于JDK实现原理的文章，还是觉得有点难度的，但是对于源码分析能力还是有点提升的。在这个过程中最好多打断点，多调试。

参考资料：

序列化和反序列化

<https://www.cnblogs.com/sinceret/p/10285807.html>

序列化和反序列化的详解

https://blog.csdn.net/tree_ifconfig/article/details/82766587

Java 之 Serializable 序列化和反序列化的概念,作用的通俗易懂的解释

<https://blog.csdn.net/u013870094/article/details/82765907>

Java中序列化实现原理研究

https://blog.csdn.net/weixin_39723544/article/details/80527550

关于Java序列化你应该知道的一切

- 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. 41 道 Spring Boot 面试题，帮你整理好了！](#)

[2. Zookeeper 入门文章](#)

[3. 女程序员做了个梦。。。](#)

[4. Spring 和 Spring Boot 之间到底有啥区别？](#)



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 Java 程序员的角度理解加密

Java后端 2019-09-03

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 | 张丰哲

链接 | www.jianshu.com/p/7c665d5f734e

在我们日常的程序开发中，或多或少会遇到一些加密/解密的场景，比如在一些接口调用的过程中，我们（Client）不仅仅需要传递给接口服务（Server）必要的业务参数，还得提供Signature（数字签名）以供Server端进行校验（是否是非法请求？是否有篡改？）；Server端进行处理后返回给Client的响应结果中还会包含Signature，以供校验。本篇博客将从Java程序员的角度出发，通俗理解加密、解密的那些事！

理解一些术语：单向、对称、非对称

假设场景：client需要发送一段消息"hello world"给server

单向加密

所谓单向加密是指client将消息"hello world"加密的过程不需要server参与，即加密不依赖server；同时，server将受到的消息解密成"hello world"的过程也不依赖client。

例如，咱们知道的MD5就是一种单向加密算法，是一种不可逆的算法。

对称加密

client加密消息需要依赖server，双方可以相互解密。

非对称加密

client加密消息需要依赖server，但是双方不能相互解密。

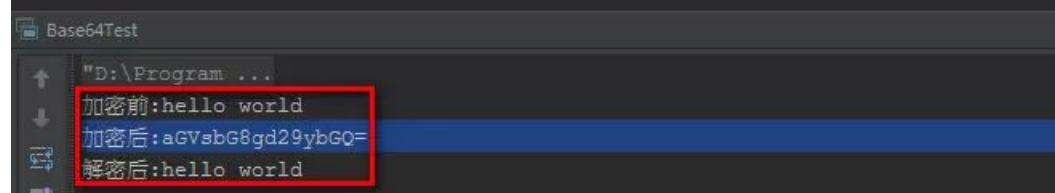
不可不知的Base64编码

先看一段代码：

```
import sun.misc.BASE64Decoder;
import sun.misc.BASE64Encoder;
import java.io.IOException;
/**
 * Created by zhangfz on 2018/2/3,
 */
public class Base64Test {

    public static void main(String[] args) throws IOException {

        String msg = "hello world";
        System.out.println("加密前:" + msg);
        String encode = new BASE64Encoder().encode(msg.getBytes("utf-8"));
        System.out.println("加密后:" + encode);
        String s = new String(new BASE64Decoder().decodeBuffer(encode), "utf-8");
        System.out.println("解密后:" + s);
    }
}
```



需要注意的是，`BASE64Encoder`和`BASE64Decoder`并不是官方JDK实现类，如果需要使用，需要引入`sun.misc`包。

严格来说，`BASE64`并不是一种加密算法，而是一种编码格式。说白了，`BASE64`的作用是，将人肉眼可以识别的信息，转换为不可以识别的数据，并不是对数据进行加密，只是给数据换了一身衣服而已。(骗的了你的眼睛，骗不了程序)

原数据越大，那么`BASE64`生成的结果就越大，这是需要额外注意的点。

`BASE64`的生成结果始终由64个字符来组成。

由于`BASE64`的编码特性，在一些场景中有应用，比如有些网站会把图片的二进制流编码成`BASE64`传递给客户端；比如有些邮件服务器会将邮件的附件直接编码成`BASE64`连同邮件内容一起发送；比如在URL中有中文需要传递，可以先将中文进行`BASE64`编码，来避免传输过程中的乱码。

使用广泛的MD5

MD5，即Message Digest，信息摘要算法第5版。比如在和微信支付、支付宝支付接口交互的过程中，你就可以选择MD5算法来加密。

先来看一段代码：

```
public static void main(String[] args) throws UnsupportedEncodingException, NoSuchAlgorithmException {
    String msg = "hello world";
    MessageDigest md5 = MessageDigest.getInstance("MD5");
    md5.update(msg.getBytes());
    byte[] digest = md5.digest();
    BigInteger bigInteger = new BigInteger(digest);
    System.out.println("加密前：" + msg);
    System.out.println("加密后：" + bigInteger.toString(16).toLowerCase());
}
```

MD5Test

```
"D:\Program ...  
加密前:hello world  
加密后:5eb63bbbe01e3e093cb22bb8f5acdc3
```

MD5破解？

如前文所说，MD5是一种不可逆的算法，但是为什么存在破解呢？其实，所谓的破解，并不是真正的破解，只不过是大数据查询的一个碰撞而已。比如，有一台服务器存储了大量key以及key的MD5编码的信息，那么就可以拿着数据去进行比对。

那么实际场景中，一般我们如何防止这种暴力破解呢？

答案：进行二次加密。

比如client在调用server接口的时候，server分配给client一个Token，每次client调用server接口的时候，需要对Token以及业务参数一起进行MD5加密。其实这就是所谓的一个“加盐”的过程。

MD5的一些特性分析

第一，我们知道BASE64随着原数据的增大而导致编码后的结果长度变大，而MD5结果的长度值是固定的，就是32位。也就是MD5的压缩性很好。

第二，从原数据计算出MD5是一个快速且容易的过程，不可逆。

第三，要找到2个不同的数据，它们计算后的MD5一致，这是非常困难的。这是MD5的弱碰撞性，也即是说想要伪造数据太困难了。

第四，对原数据的任何修改，哪怕只改动一个字节数据，也会导致MD5值发生很大变化，说明MD5的抗修改性非常好，非常适合密码、业务数据校验、文件比对等。

了解SHA

SHA，即Security Hash Algorithm，安全散列算法，比如，我们的程序开发完毕，我们发布的时候，想指定的人才可以使用，该怎么办呢？这个时候就可以考虑使用SHA算法。SHA是公认的比MD5更加安全的加密算法，在数字签名领域应用广泛。

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 

[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！

你写的 Java 代码是如何一步步输出结果的？

Alan Java后端 2019-11-07

点击上方 Java后端, 选择 [设为星标](#)

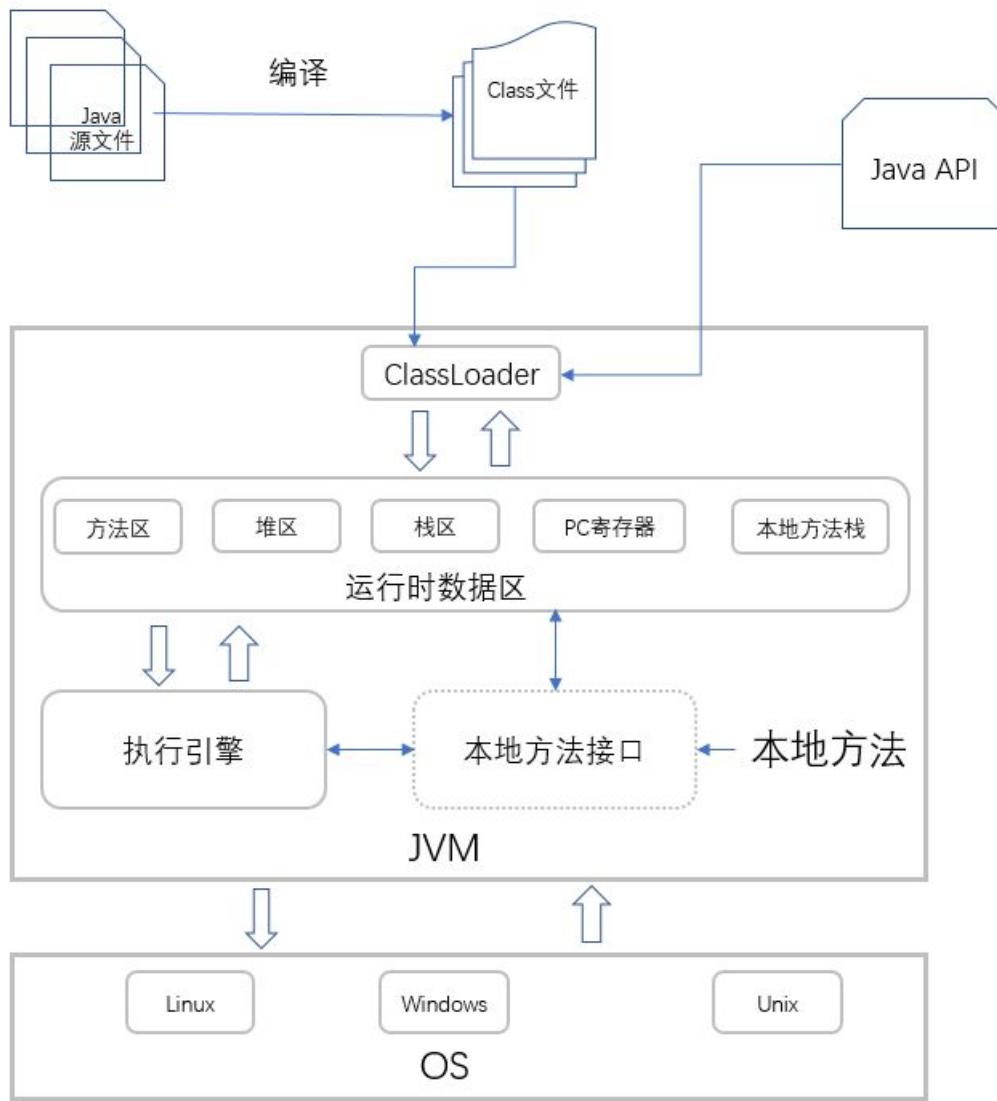
优质文章, 及时送达

作者 | Alan

来源 | cnblogs.com/wangjiming/p/10455993.html

对于任何一门语言，要想达到精通的水平，研究它的执行原理(或者叫底层机制)不失为一种良好的方式。

在本篇文章中，将重点研究java源代码的执行原理，即从程序员编写JAVA源代码，到最终形成产品，在整个过程中，都经历了什么？每一步又是怎么执行的？执行原理又是什么？



一、编写java源程序

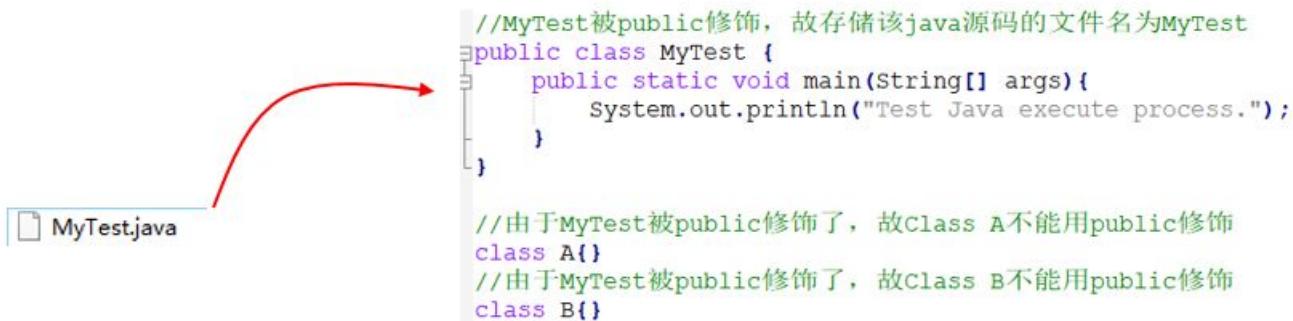
java源文件:指存储java源码的文件。

先来看看如下代码：

```
//MyTest被public修饰，故存储该java源码的文件名为MyTest
public class MyTest {
    public static void main(String[] args){
        System.out.println("Test Java execute process.");
    }
}
//由于MyTest被public修饰了，故Class A不能用public修饰
class A{}
```

```
//由于MyTest被public修饰了，故Class B不能用public修饰
class B{}
```

1、java源文件名就是该源文件中public类的名称



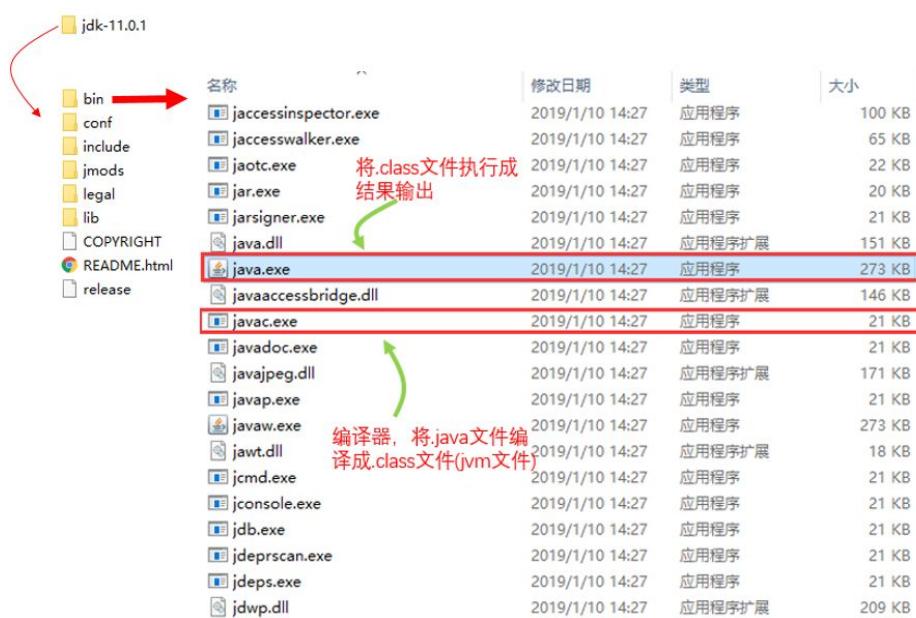
2、一个java源文件可以包含多个类，但只允许一个类为public

二、编译java源代码

当java源程序编码结束后，就需要编译器编译。

安装好jdk后，我们打开jdk目录，有两个.exe文件，即javac.exe(编译源代码, xxx.java文件) 和 java.exe(执行字节码，xxx.class文件)。

如下图所示：



1、切换到MyTest.java文件夹

```
C:\Users\Alan_beijing>e:
E:\>cd e:\Blogs
```

2、javac.exe编译MyTest.java

```
e:\Blogs>javac.exe MyTest.java
```

编译后，发现e:\Blogs 目录多了以class为后缀的文件：A.class,B.class和MyTest.class

```
e:\Blogs>dir e:\Blogs
驱动器 E 中的卷是 软件
卷的序列号是 92F3-024B

e:\Blogs 的目录

2019/03/01 10:56    <DIR>          .
2019/03/01 10:56    <DIR>          ..
2019/03/01 10:56      181 A.class
2019/03/01 10:56      181 B.class
2019/03/01 10:56      432 MyTest.class
2019/03/01 10:43      366 Mytest.java
                      4 个文件          1,160 字节
                      2 个目录 105,573,572,608 可用字节
```

Tip:当javac.exe编译java源代码时，java源代码有几个类，就会编译成一个对应的字节码文件(.class文件)

其中，字节码文件的文件名就是每个类的类名。需要注意的是，类即使不在源文件中定义，但被源文件引用，编译后，也会编程相应的字节码文件。

如类A引用类C，但类C不定义在类A的源文件中，编译后，类C也被编译成对应的字节码文件C.class

Tips: 关注微信公众号：Java后端，每日获取技术博文推送。

三、执行java源文件

执行java源文件，用java.exe执行即可

```
e:\Blogs>java.exe MyTest
Test Java execute process.
```

到现在，java源程序基本执行结果，并正确打印我们期望的结果，那么，如上的步骤，我们可以总结如下：



如上总结，已经抽象化了在JVM中的执行。接下来，我们将分析字节码文件（.class文件）如何在虚拟机中一步一执行的。

四、JVM如何执行字节码文件

1、装载字节码文件

当.java 源码被javac.exe 编译器编译成.class 字节码文件后，接下来的工作就交给JVM处理。

JVM首先通过类加载器(ClassLoader)，将class文件和相关Java API加载装入JVM，以供JVM后续处理。

在该阶段中，涉及到如下一些基本概念和知识。

1) JDK,JRE和JVM关系

- JDK (Java Development Kit) , Java开发工具包，主要用于开发，在JDK7前，JDK包括JRE

- JRE (Java Runtime Environment) , Java程序运行的核心环境，包括JVM和一些核心库
- JVM (Java Virtual Machine) , VM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的，是JRE核心模块。

2) JVM

JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

Java虚拟机的主要任务是装载class文件，并执行其中的字节码，不同的Java虚拟机中，执行引擎可能有不同的实现。

大致有如下几种引擎：

- 一次性解释字节码引擎
- 即时编译引擎
- 自适应优化器

关于虚拟机的实现方式，采用软件方式、硬件方式和软件硬件结合方式，这个要根据具体厂商而定。

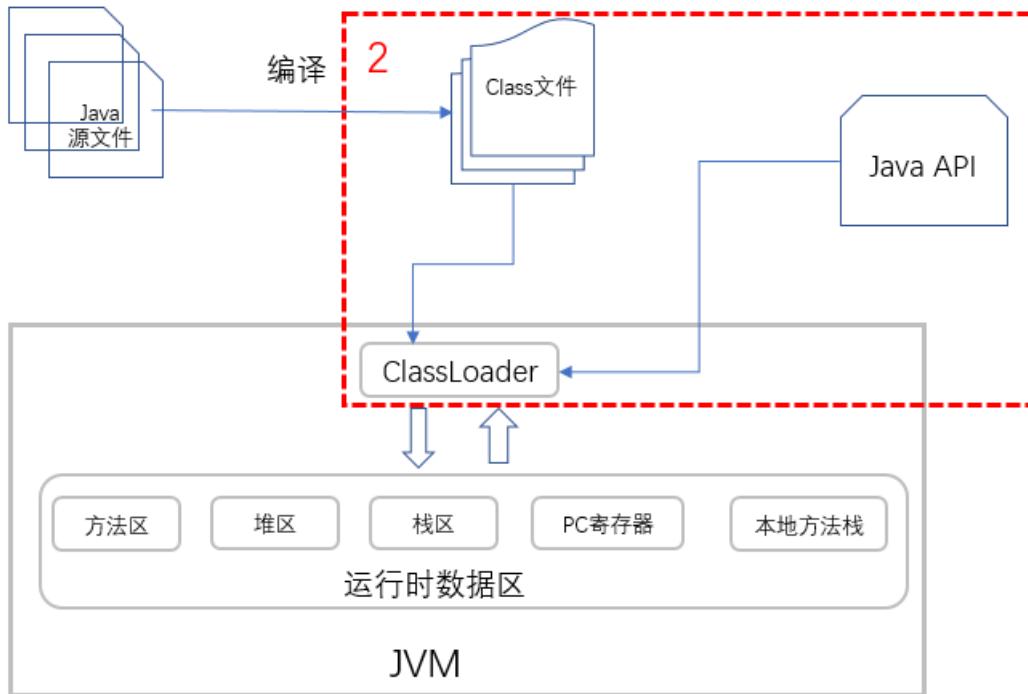
3) 什么是ClassLoader

虚拟机的主要任务是装载class文件并执行其中的字节码，而class文件是由虚拟机的类加载器(ClassLoader)完成的，在一个Java虚拟机中有可能存在多个类加载器。

任何java应用程序，可能会使用两种类加载器，即启动类加载器(bootstrap)和用户自定义类加载器。

启动类加载器是Java虚拟机唯一实现的一部分，它又可分为原始类装载器，系统类装载器或默认类装载器。它的主要作用是从操作系统的磁盘装载相应的类，如Java API类等。

用户自定义装载类，即按照用户自定义的方式来装载类。



2、将字节码文件存储在JVM内存区

当JAVA虚拟机运行一个程序时，它需要内存来存储许多东西。

比如如字节码，程序创建的对象，传递给方法的参数，返回值，局部变量以及运算的中间结果等，这些相关信息被组织到“运行时数据区”。

根据厂商的不同，在Java虚拟机中，运行时数据区也有所不同。有些运行时数据区由线程共享，有些只能由某个特定线程共享。

运行时数据区大致可分几个区：方法区，堆区，栈区，PC寄存器区和本地方法栈区。

在该阶段中，涉及到如下基本概念和知识。

1) 方法区

方法区用来存储解析被加载的class文件的相关信息。

当虚拟装载一个class文件后，它会从这个class文件包含的二进制数据中解析类型信息，然后将该相关信息存储到方法区中。

2) 堆

堆是用来存储相关引用类型的，如new对象。当程序运行时，虚拟机会把所有该程序在运行时创建的对象都放到堆中。

3) PC寄存器

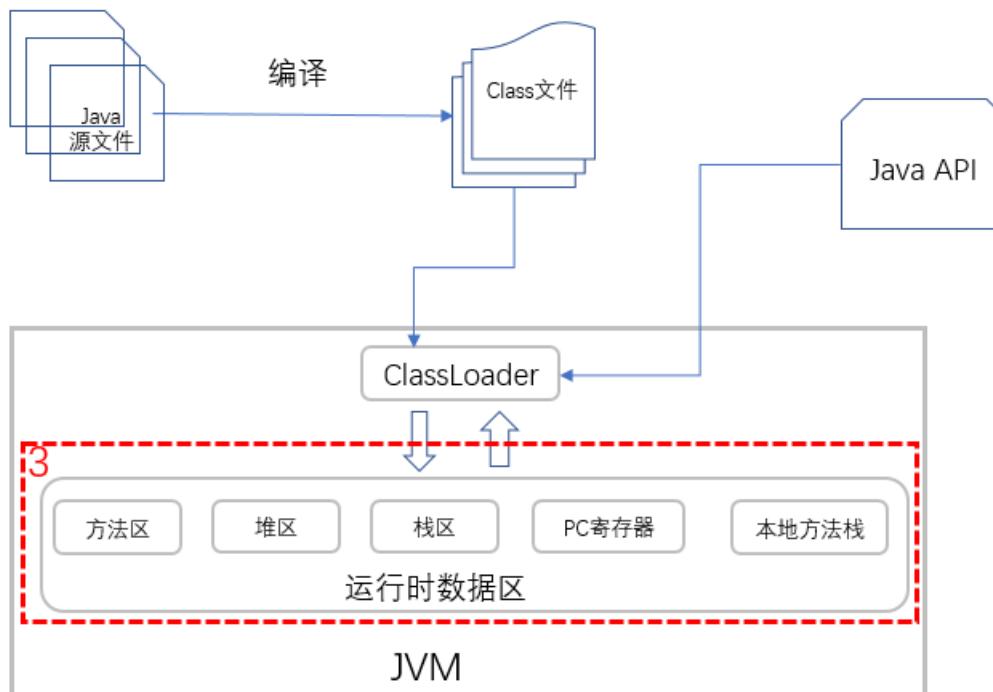
PC寄存器主要用来存储线程。当新创建一个线程时，该线程都将得到一个自己的PC寄存器(程序计数器)以及一个java栈。

Java虚拟机没有寄存器，其指令集使用Java栈来存储中间数据。

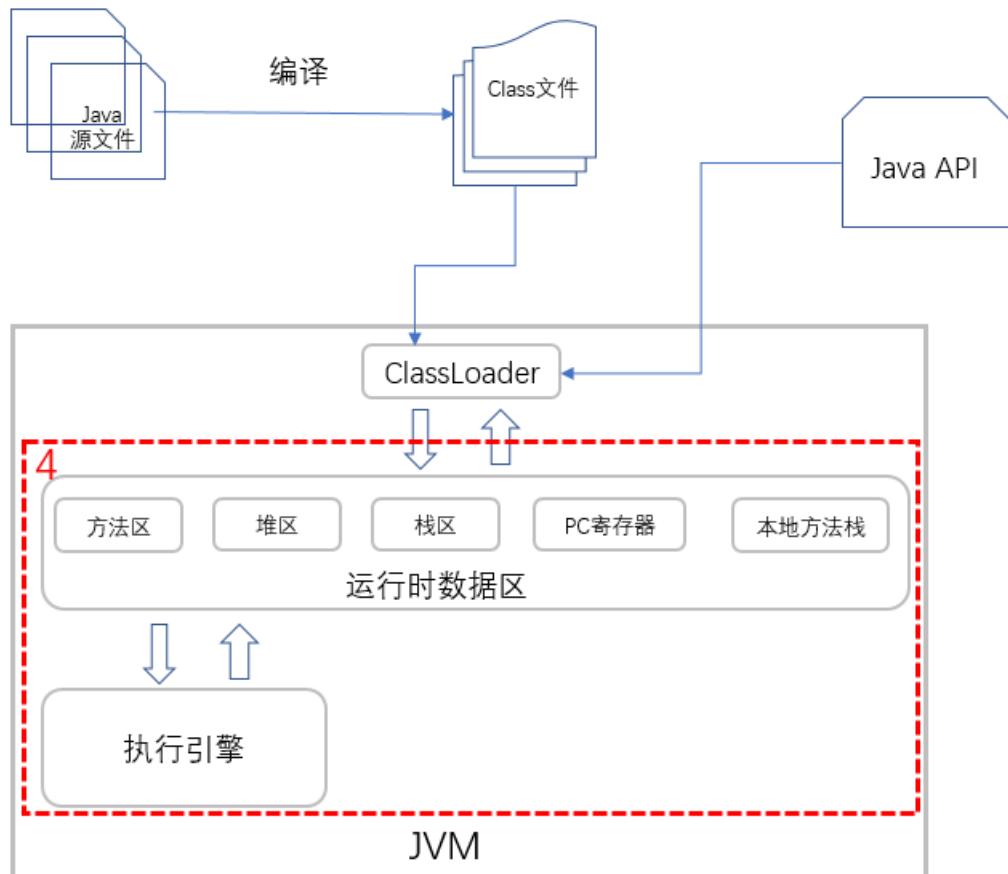
4) 栈区

栈区主要用来存储值类型的，如基本数据类型。需要注意的是，String为引用类型，是存在堆中的。

Java栈是由许多栈帧组成的，一个栈帧包含一个Java方法调用的状态，当线程调用一个方法时，虚拟机压入一个新的栈帧到该线程的Java栈中，当该方法返回时，这个栈帧从Java栈中弹出。



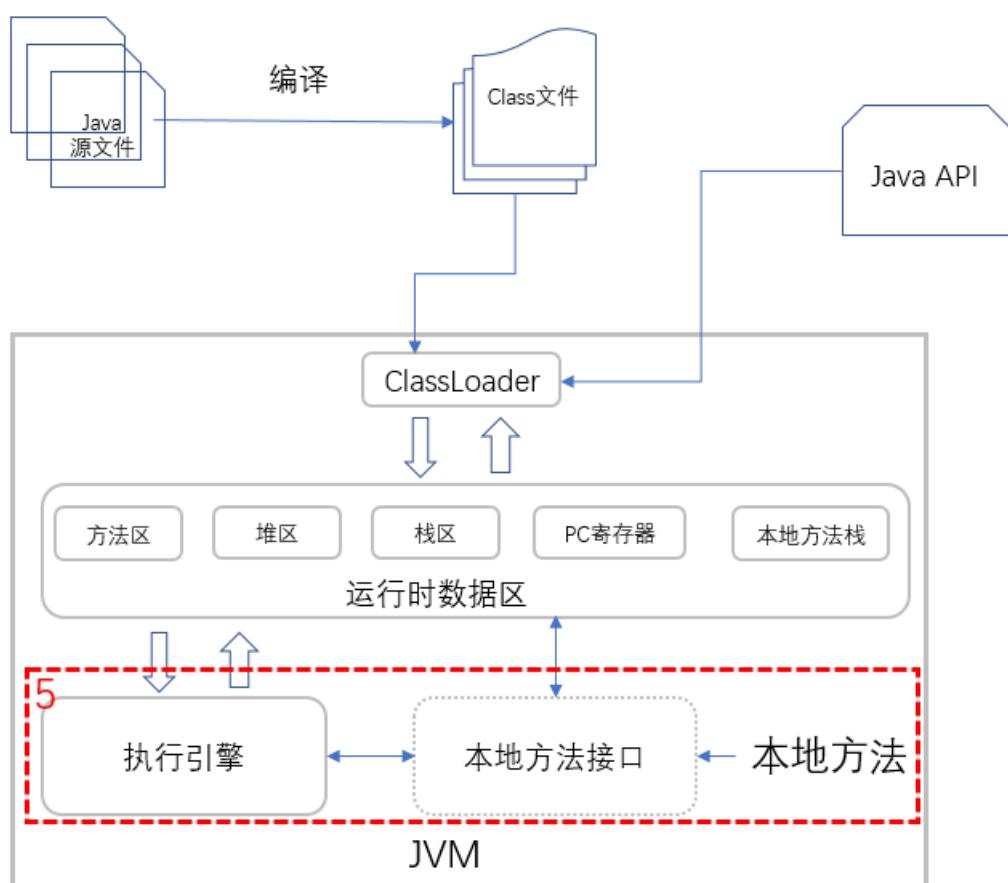
运行时数据区为执行引擎提供了执行环境和相关数据，执行引擎通过与运行时数据区交互，从而获取执行时需要的相关信息，存储执行的中间结果等



4、执行引擎与本地方法接口

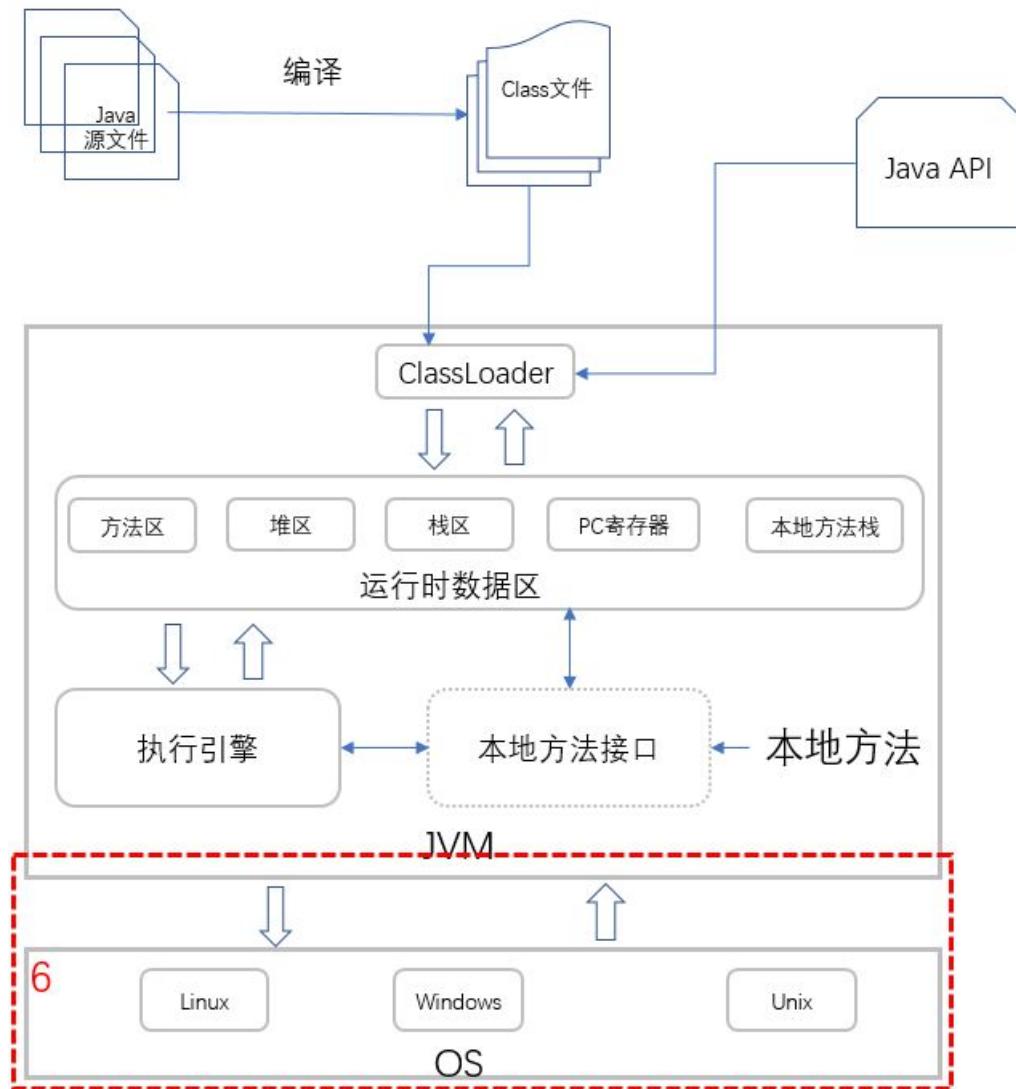
当要执行本地方法时，执行引擎将调用本地方法接口来获取相关OS本地方法。

需要注意的是，本地方法与操作系统强耦合的。



5、JVM在具体操作系统上执行

JVM通过调用本地接口来获取本地方法，从而实现在具体的平台上执行。比如在Linux系统上执行，在Window系统上执行和在Unix系统上执行。



五、参考文献

- 1、深入Java虚拟机（原书第2版）(美)Bill Venners著
- 2、Core Java Volume I - Fundamentals(10th Edition)
- 3、Core Java Volume I - Advanced Features(10th Edition)

-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!

别在 Java 代码里乱打日志了，这才是正确姿势

Irwin 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 | Irwin

来源 | <http://lrwinx.github.io>

使用slf4j

1. 使用门面模式的日志框架,有利于维护和各个类的日志处理方式统一。
2. 实现方式统一使用: Logback框架

打日志的正确方式

什么时候应该打日志

1. 当你遇到问题的时候,只能通过debug功能来确定问题,你应该考虑打日志,良好的系统,是可以通过日志进行问题定位的。
2. 当你碰到if…else 或者 switch这样的分支时,要在分支的首行打印日志,用来确定进入了哪个分支
3. 经常以功能为核心进行开发,你应该在提交代码前,可以确定通过日志可以看到整个流程

基本格式

必须使用参数化信息的方式:

```
1 logger.debug("Processing trade with id:[{}] and symbol : [{}]", id, symbol)
;
```

对于debug日志,必须判断是否为debug级别后,才进行使用:

```
1 if (logger.isDebugEnabled()) {
2     logger.debug("Processing trade with id: " + id + " symbol: " + symbol)
3 }
;
```

不要进行字符串拼接,那样会产生很多String对象,占用空间,影响性能。

反例(**不要这么做**):

```
1 logger.debug("Processing trade with id: " + id + " symbol: " + symbol)
;
```

使用[]进行参数变量隔离

如有参数变量，应该写成如下写法：

```
1 logger.debug("Processing trade with id:[{}], symbol : [{}]", id, symbol)
;
```

这样的格式写法，可读性更好，对于排查问题更有帮助。

不同级别的使用

ERROR:

基本概念

影响到程序正常运行、当前请求正常运行的异常情况：

1. 打开配置文件失败
2. 所有第三方对接的异常(包括第三方返回错误码)
3. 所有影响功能使用的异常，包括:SQLException和除了业务异常之外的所有异常(RuntimeException和Exception)

不应该出现的情况：

1. 比如要使用Azure传图片，但是Azure未响应

如果有Throwable信息，需要记录完成的堆栈信息：

```
1 log.error("获取用户[{}]的用户信息时出错",userName,e);
```

说明

如果进行了抛出异常操作，请不要记录error日志，由最终处理方进行处理：

反例(不要这么做)：

```
1 try{
2     ...
3 }catch(Exception ex){
4     String errorMessage=String.format("Error while reading information of user [%s]",userName)
5 ;
6     logger.error(errorMessage,ex);
7     throw new UserServiceException(errorMessage,ex);
8 }
```

WARN

基本概念

不应该出现但是不影响程序、当前请求正常运行的异常情况:

1. 有容错机制的时候出现的错误情况
2. 找不到配置文件,但是系统能自动创建配置文件

即将接近临界值的时候,例如:

1. 缓存池占用达到警告线

业务异常的记录,比如:

2. 当接口抛出业务异常时,应该记录此异常

INFO:

基本概念

系统运行信息

1. Service方法中对于系统/业务状态的变更

2. 主要逻辑中的分步骤

外部接口部分

1. 客户端请求参数(REST/WS)

2. 调用第三方时的调用参数和调用结果

说明

1. 并不是所有的service都进行出入口打点记录,单一、简单service是没有意义的(job除外,job需要记录开始和结束,)。

反例(**不要这么做**):

```
1 public List listByBaseType(Integer baseTypeId) {  
2     log.info("开始查询基地")  
3 ;  
4     BaseExample ex=new BaseExample();  
5     BaseExample.Criteria ctr = ex.createCriteria();  
6     ctr.andIsDeleteEqualTo(IsDelete.USE.getValue());  
7     Optionals.doIfPresent(baseTypeId, ctr::andBaseTypeIdEqualTo);  
8     log.info("查询基地结束")  
9 ;  
10    return baseRepository.selectByExample(ex);  
11 }
```

2.对于复杂的业务逻辑,需要进行日志打点,以及埋点记录,比如电商系统中的下订单逻辑,以及OrderAction操作(业务状态变更)。

3.对于整个系统的提供出的接口(REST/WS),使用info记录入参

4.如果所有的service为SOA架构,那么可以看成是一个外部接口提供方,那么必须记录入参。

5.调用其他第三方服务时,所有的出参和入参是必须要记录的(因为你很难追溯第三方模块发生的问题)

DEBUG

基本概念

1. 可以填写所有的想知道的相关信息(但不代表可以随便写,debug信息要有义,最好有相关参数)

2. 生产环境需要关闭DEBUG信息

3. 如果在生产情况下需要开启DEBUG,需要使用开关进行管理,不能一直开启。

说明

如果代码中出现以下代码,可以进行优化:

```
1 //1. 获取用户基本薪资  
2 //2. 获取用户休假情况  
3 //3. 计算用户应得薪资
```

优化后的代码:

```
1 logger.debug("开始获取员工[{}] [{}]年基本薪资", employee, year);  
2  
3 logger.debug("获取员工[{}] [{}]年的基本薪资为[{}]", employee, year, basicSalary);  
4 logger.debug("开始获取员工[{}] [{}]年[{}]月休假情况", employee, year, month);  
5  
6 logger.debug("员工[{}][{}]年[{}]月年假/病假/事假为[{}]/[{}]/[{}]", employee, year, month, annualLeaveDays, sickLeaveDays, no  
7 logger.debug("开始计算员工[{}][{}]年[{}]月应得薪资", employee, year, month);  
8  
9 logger.debug("员工[{}] [{}]年[{}]月应得薪资为[{}]", employee, year, month, actualSalary);
```

TRACE

基本概念

特别详细的系统运行完成信息,业务代码中,不要使用.(除非有特殊用意,否则请使用DEBUG级别替代)

规范示例说明

```
1 @Override
```

```
2     @Transactional
3
4     public void createUserAndBindMobile(@NotBlank String mobile, @NotNull User user) throws CreateConflictException{
5
6         boolean debug = log.isDebugEnabled();
7
8         if(debug){
9             log.debug("开始创建用户并绑定手机号. args[mobile=[{}],user=[{}]]", mobile, LogObjects.toString(user));
10        }
11
12        try{
13
14            user.setCreateTime(new Date());
15            user.setUpdateTime(new Date());
16            userRepository.insertSelective(user);
17
18            if(debug)
19            {
14                log.debug("创建用户信息成功. insertedUser=[{}]",LogObjects.toString(user));
15            }
16
17            UserMobileRelationship relationship = new UserMobileRelationship();
18            relationship.setMobile(mobile);
19            relationship.setOpenId(user.getOpenId());
20            relationship.setCreateTime(new Date())
21;
22            relationship.setUpdateTime(new Date())
23;
24            userMobileRelationshipRepository.insertOnDuplicateKey(relationship);
25
26            if(debug)
27            {
26                log.debug("绑定手机成功. relationship=[{}]",LogObjects.toString(relationship));
27            }
28
29            log.info("创建用户并绑定手机号. userId=[{}],openId=[{}],mobile=[{}]",user.getId(),user.getOpenId(),mobile);
30        }catch(DuplicateKeyException e){
31            log.info("创建用户并绑定手机号失败, 已存在相同的用户. openId=[{}],mobile=[{}]",user.getOpenId(),mobile);
32            throw new CreateConflictException("创建用户发生冲突, openid=[%s]",user.getOpenId());
33        }
34    }
35}
```

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

[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！

动图演示 Java 中常用数据结构

Java后端 2019-08-28

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

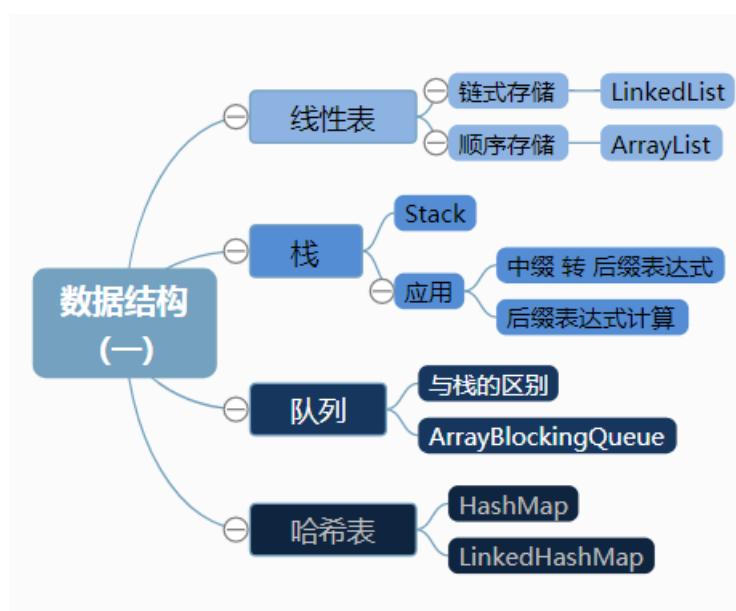
来自 | 大道方圆

链接 | cnblogs.com/xdecode/p/9321848.html

最近在整理数据结构方面的知识, 系统化看了下Java中常用数据结构, 突发奇想用动画来绘制数据流转过程.

主要基于jdk8, 可能会有些特性与jdk7之前不相同, 例如LinkedList LinkedHashMap中的双向列表不再是回环的.

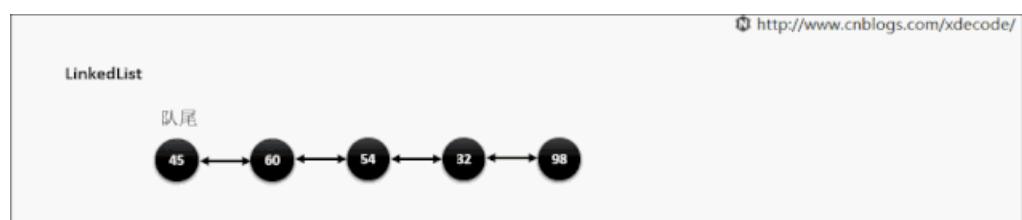
HashMap中的单链表是尾插, 而不是头插入等等, 后文不再赘叙这些差异, 本文目录结构如下:



LinkedList

经典的双链表结构, 适用于乱序插入, 删除. 指定序列操作则性能不如ArrayList, 这也是其数据结构决定的.

add(E) / addLast(E)



add(index, E)

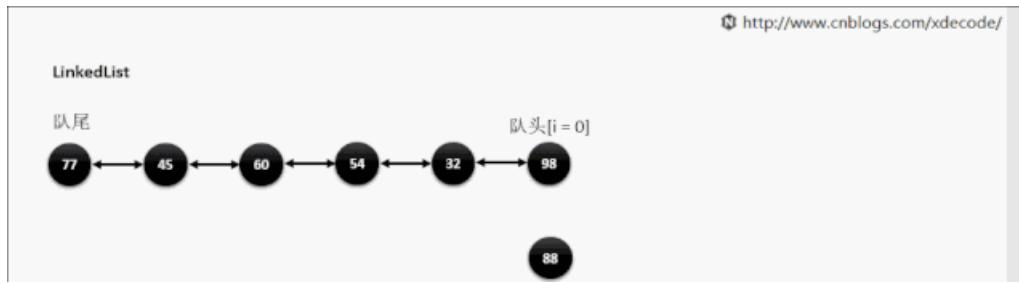
这边有个小的优化, 他会先判断index是靠近队头还是队尾, 来确定从哪个方向遍历链入.

```
1 if (index < (size >> 1))
2 {
3     Node<E> x = first
4 }
```

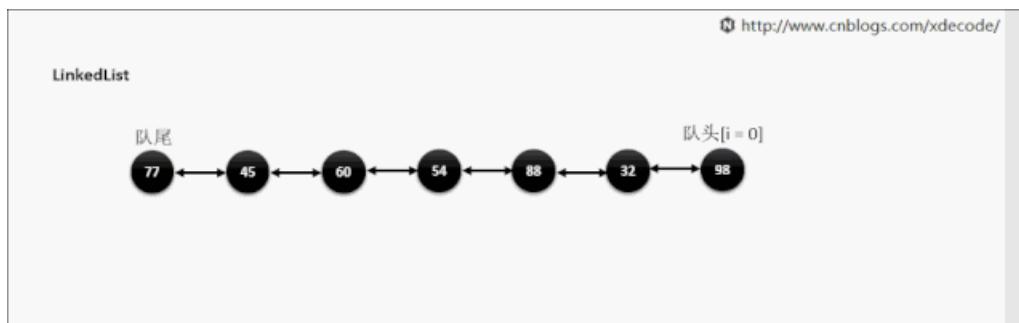
```

5     for (int i = 0; i < index; i+
6     +
7     ; x = x.next
8
9 }
10 } else
11 {
12     Node<E> x = last
13
14     for (int i = size - 1; i > index; i-
15     -)
16         x = x.prev
17
18     return x
19 }

```

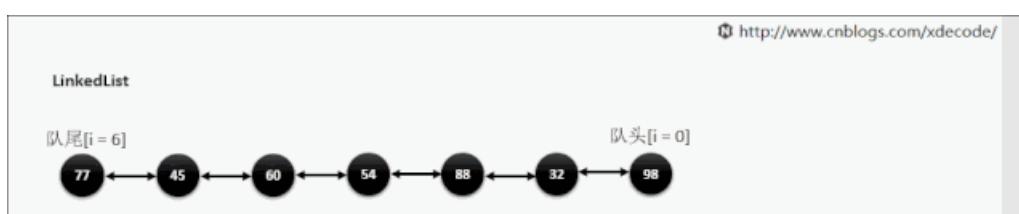
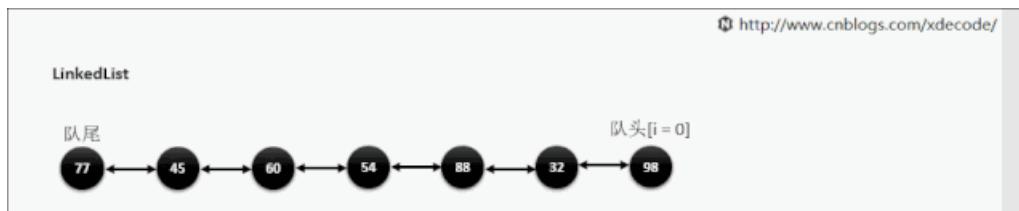


靠队尾

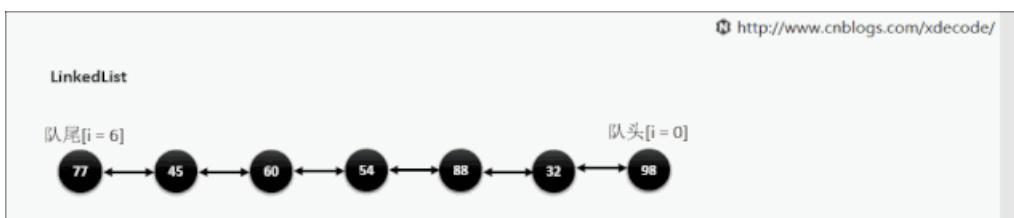


get(index)

也是会先判断index, 不过性能依然不好, 这也是为什么不推荐用for(int i = 0; i < length; i++)的方式遍历linkedlist, 而是使用iterator的方式遍历.



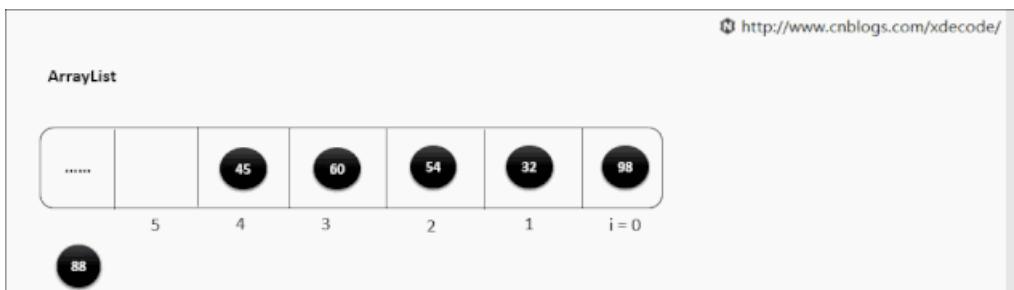
remove(E)



ArrayList

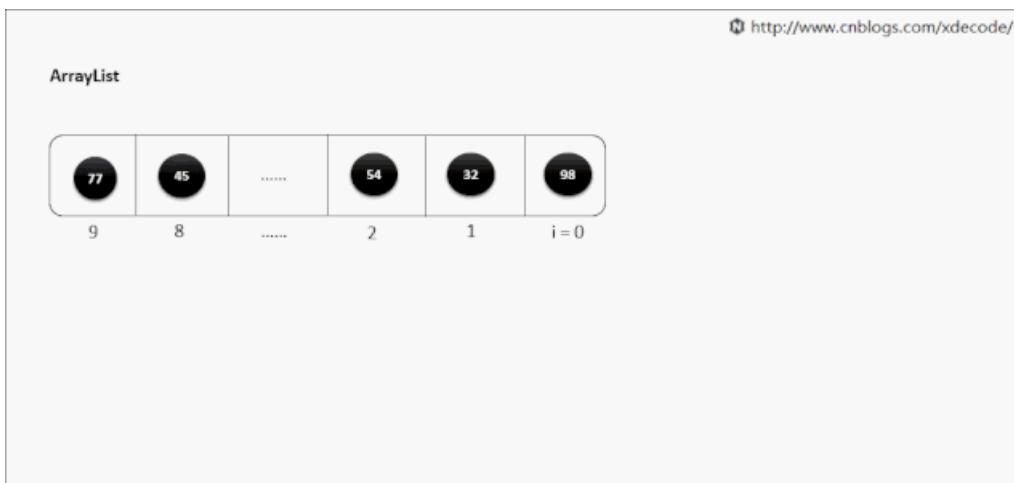
底层就是一个数组，因此按序查找快，乱序插入，删除因为涉及到后面元素移位所以性能慢。

add(index, E)



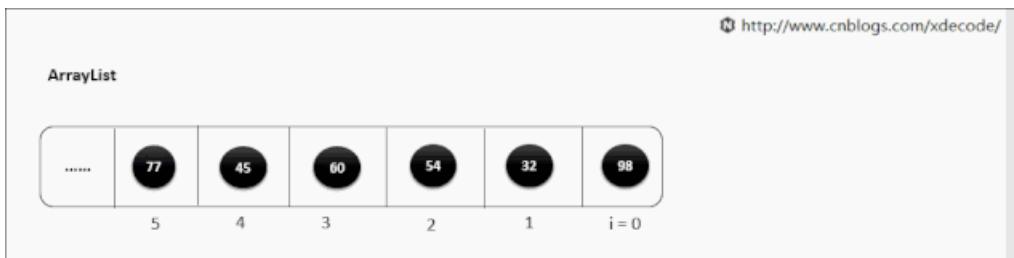
扩容

一般默认容量是10，扩容后，会 $\text{length} * 1.5$.



remove(E)

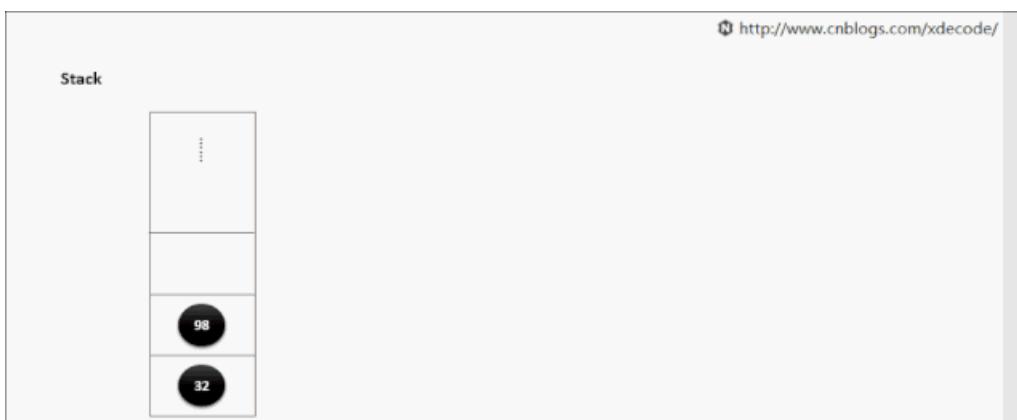
循环遍历数组，判断E是否equals当前元素，删除性能不如LinkedList.



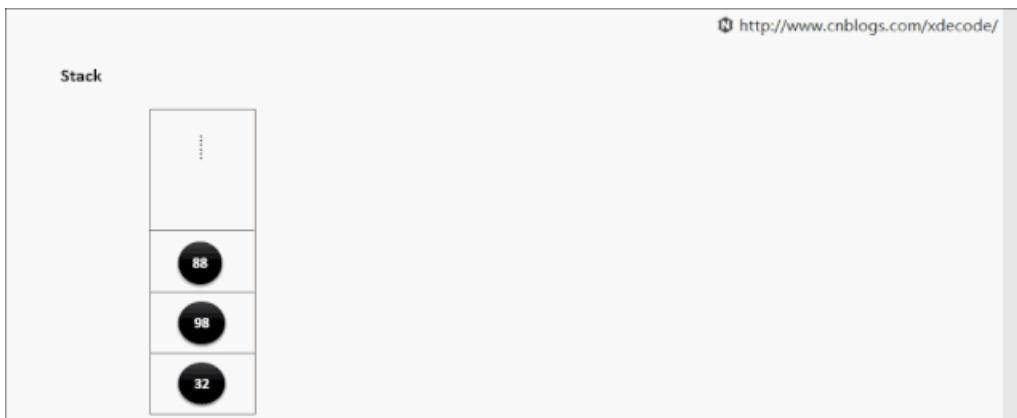
Stack

经典的数据结构，底层也是数组，继承自Vector，先进后出FILO，默认new Stack()容量为10，超出自动扩容。

push(E)



pop()



后缀表达式

Stack的一个典型应用就是计算表达式如 $9 + (3 - 1) * 3 + 10 / 2$, 计算机将中缀表达式转为后缀表达式, 再对后缀表达式进行计算。

中缀转后缀

- 数字直接输出
- 栈为空时, 遇到运算符, 直接入栈
- 遇到左括号, 将其入栈
- 遇到右括号, 执行出栈操作, 并将出栈的元素输出, 直到弹出栈的是左括号, 左括号不输出。
- 遇到运算符(加减乘除): 弹出所有优先级大于或者等于该运算符的栈顶元素, 然后将该运算符入栈
- 最终将栈中的元素依次出栈, 输出。

后缀表达式

$9 + (3 - 1) * 3 + 10 / 2$

数字输出, 运算符进栈, 括号匹配出栈, 栈顶优先级低出栈



计算后缀表达式

- 遇到数字时, 将数字压入堆栈
- 遇到运算符时, 弹出栈顶的两个数, 用运算符对它们做相应的计算, 并将结果入栈
- 重复上述过程直到表达式最右端
- 运算得出的值即为表达式的结果

后缀表达式

$9 + (3 - 1) * 3 + 10 / 2$



队列

与 Stack 的区别在于, Stack 的删除与添加都在队尾进行, 而 Queue 删除在队头, 添加在队尾.

ArrayBlockingQueue

生产消费者中常用的阻塞有界队列, FIFO.

put(E)

ArrayBlockingQueue



put(E) 队列满了

```

1 final ReentrantLock lock = this.lock
2 ;
3     lock.lockInterruptibly();
4     try
5 {
6         while (count == items.length)
7             notFull.await()
8 ;
9         enqueue(e);
10 } finally
11 {
12     lock.unlock()
13 ;
14 }
```

ArrayBlockingQueue



take()

当元素被取出后，并没有对数组后面的元素位移，而是更新takeIndex来指向下一个元素.

takeIndex是一个环形的增长，当移动到队列尾部时，会指向0，再次循环.

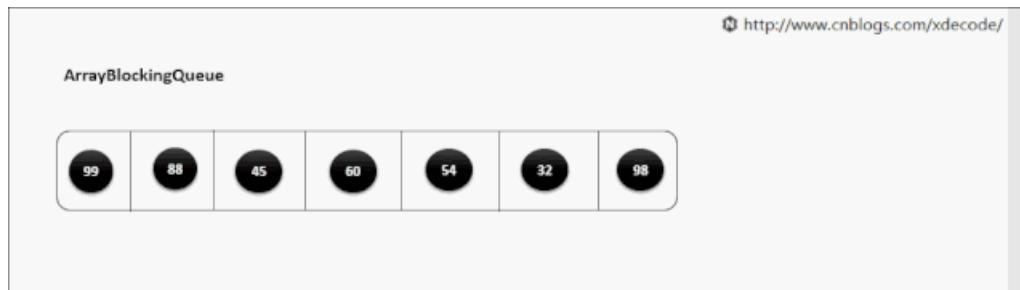
```

1 private E dequeue() {
2     // assert lock.getHoldCount() == 1;
3     // assert items[takeIndex] != null;
4     final Object[] items = this.items
5 ;
6     @SuppressWarnings("unchecked"
7 )
8     E x = (E) items[takeIndex];
9     items[takeIndex] = null
10 ;
11     if (++takeIndex == items.length)
```

```

12         takeIndex = 0
13     ;
14     count--;
15     if (itrs != null
16     )
17         itrs.elementDequeued();
18     notFull.signal();
19     return x
20     ;
21 }

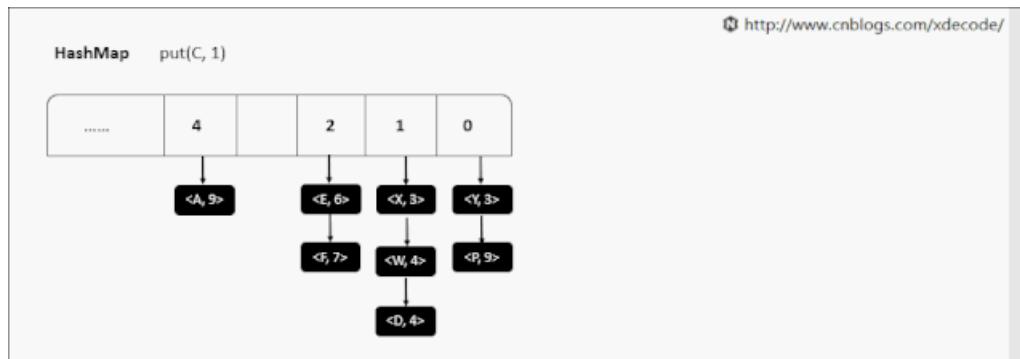
```



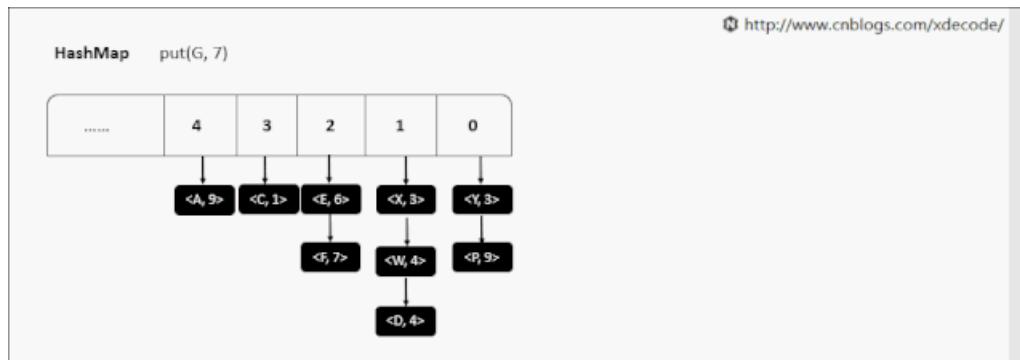
HashMap

最常用的哈希表，面试的童鞋必备知识了，内部通过数组 + 单链表的方式实现。jdk8中引入了红黑树对长度 > 8 的链表进行优化，我们另外篇幅再讲。

put(K, V)



put(K, V) 相同hash值



resize 动态扩容

当map中元素超出设定的阈值后，会进行resize ($\text{length} * 2$)操作，扩容过程中对元素一通操作，并放置到新的位置。

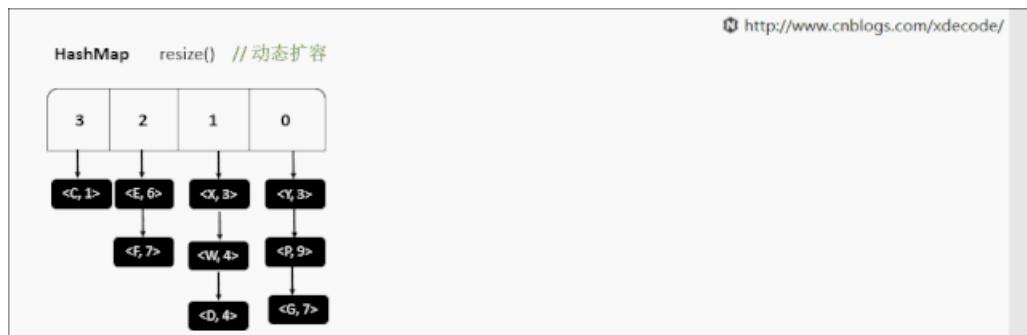
具体操作如下：

■ 在jdk7中对所有元素直接rehash, 并放到新的位置.

■ 在jdk8中判断元素原hash值新增的bit位是0还是1, 0则索引不变, 1则索引变成"原索引 + oldTable.length".

```
1 //定义两条链
2     //原来的hash值新增的bit为0的链, 头部和尾部
3     Node<K,V> loHead = null, loTail = null
4 ;
5     //原来的hash值新增的bit为1的链, 头部和尾部
6     Node<K,V> hiHead = null, hiTail = null
7 ;
8     Node<K,V> next;
9     //循环遍历出链条
10    链
11    do
12    {
13        next = e.next;
14        if ((e.hash & oldCap) == 0)
15        {
16            if (loTail == null
17        )
18                loHead = e;
19            else
20                e
21                    loTail.next = e;
22                loTail = e;
23            }
24        else
25        {
26            if (hiTail == null
27        )
28                hiHead = e;
29            else
30                e
31                    hiTail.next = e;
32                hiTail = e;
33            }
34        } while ((e = next) != null)
35 ;
36     //扩容前后位置不变的
37     链
38     if (loTail != null)
39     {
40         loTail.next = null
41 ;
42         newTab[j] = loHead;
43     }
44     //扩容后位置加上原数组长度的
45     链
46     if (hiTail != null)
47     {
48         hiTail.next = null
49 ;
```

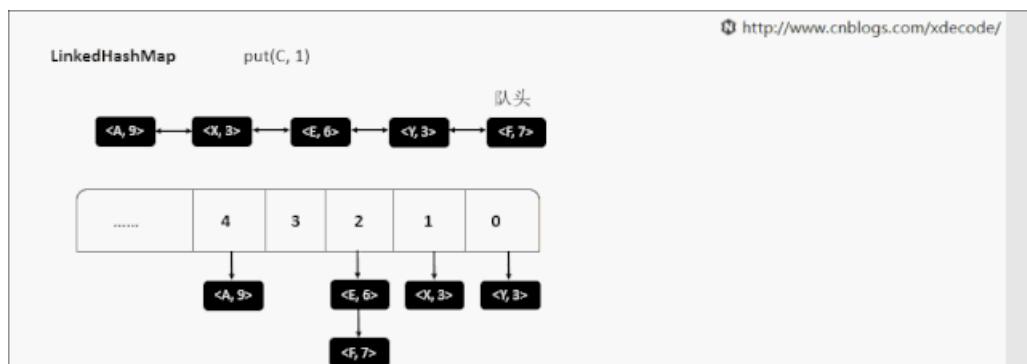
```
newTab[j + oldCap] = hiHead;  
}
```



LinkedHashMap

继承自HashMap, 底层额外维护了一个双向链表来维持数据有序. 可以通过设置accessOrder来实现FIFO(插入有序)或者LRU(访问有序)缓存.

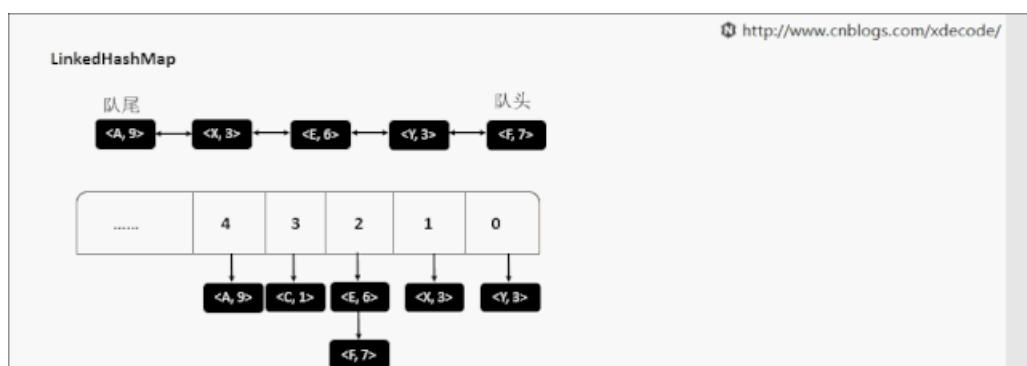
put(K, V)



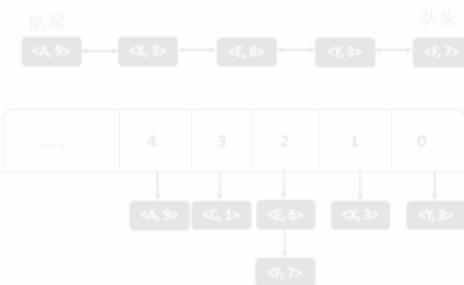
get(K)

accessOrder为false的时候, 直接返回元素就行了, 不需要调整位置.

accessOrder为true的时候, 需要将最近访问的元素, 放置到队尾.



removeEldestEntry 删除最老的元素



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. 经常用 HashMap ?这 6 个问题回答下 !](#)
- [2. 我是如何通过开源项目月入 10 万的?](#)
- [3. 小白也能看懂,30 分钟搭建个人博客!](#)
- [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！

去阿里面试 Java 都是问什么？

huashiou Java后端 2019-10-31

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 不穿格子衫的Java程序猿

来源 | jianshu.com/p/1c8271f03aa5

上篇 | [终于有人把 Docker 讲清楚了](#)

每一个互联网人心中都有一个大厂梦，百度、阿里巴巴、腾讯是很多互联网人梦寐以求的地方，而我也不例外。但是，BAT等一线互联网大厂并不是想进就能够进的，它对人才的技术能力和学历都是有一定要求的，所以除了学历以外，我们的技术和能力都要过硬才行。

今年前前后后我参加了阿里巴巴两次面试，一次是社招，一次是内推，第一次社招3面过后就被挂了，内推历经5面拿到的offer，进入的是阿里口碑部门，分享一下这次的面经，希望能帮助到大家。

社招阿里巴巴（新零售部门），三面被挂

阿里巴巴一面（55分钟）

- 先介绍一下自己吧
- 说一下自己的优缺点
- 具体讲一下之前做过的项目
- 你觉得项目里给里最大的挑战是什么？
- Hashmap为什么不用平衡树？
- AQS知道吗？知道哪一些呢？讲一讲。
- CLH同步队列是怎么实现非公平和公平的？
- ReentrantLock和synchronized的区别
- 讲一下JVM的内存结构
- JVM 里 new 对象时，堆会发生抢占吗？你是怎么去设计JVM的堆的线程安全的？
- 讲一下redis的数据结构
- redis缓存同步问题
- 讲一讲MySQL的索引结构
- 你有什么问题要问我吗？
- 直接口头通知我：答得不错，准备二面吧

阿里巴巴二面（45分钟）

- 根据项目问了一些细节问题
- 说一下HashMap的数据结构
- 红黑树和AVL树有什么区别？
- 如何才能得到一个线程安全的HashMap？

- 讲一下JVM常用垃圾回收期
- redis分布式锁
- 再描述一下你之前的项目吧
- 你觉得这个项目的亮点在哪里呢？
- 你设计的数据库遵循的范式？
- 你有没有问题？

阿里巴巴三面（50分钟）

- 又聊项目
- 在项目中，并发量大的情况下，如何才能够保证数据的一致性？
- elasticsearch为什么检索快，它的底层数据结构是怎么样的？
- JVM内存模型
- netty应用在哪些中间件和框架中呢？
- 线程池的参数
- 讲一下B树和B+树的区别
- 为什么要用redis做缓存？
- 了解Springboot吗？那讲一下Springboot的启动流程吧
- 如何解决bean的循环依赖问题？
- Java有哪些队列？
- 讲一讲Spring和Springboot的区别
- 最近看了什么书？为什么？
- 你平时是怎么学习Java的呢？

内推阿里巴巴（阿里口碑）

5面拿offer（3轮技术面+总监面+HR面）

阿里巴巴一面（38分钟） - 自我介绍

- 介绍项目，具体一点
- 讲一下Redis分布式锁的实现
- HashMap了解么吗？说一下put方法过程
- HashMap是不是线程安全？
- ConcurrentHashMap如何保证线程安全？
- 数据库索引了解吗？讲一下
- 常见排序算法
- TCP三次握手，四次挥手。
- 深入问了乐观锁，悲观锁及其实现。

阿里巴巴二面（45分钟）

- 自我介绍+项目介绍

- 你在项目中担任什么样的角色？
- 那你觉得你比别人的优势在哪里？你用了哪些别人没有的东西吗？
- Java怎么加载类？
- linux常用命令有哪些？
- Spring的IOC, AOP。
- 讲一下ORM框架Hibernate
- 设计模式了解吗？讲一下
- 自己实现一个二阶段提交，如何设计？
- 你还有什么想问的？

阿里巴巴三面（30分钟）

- 说一下自己做的项目
- 问了一些项目相关的问题
- wait()和sleep()的区别
- 原子变量的实现原理
- CAS的问题，讲一下解决方案。
- 有没有更好的计数器解决策略
- 讲一讲NIO和BIO的区别
- Nginx负载均衡时是如何判断某个节点挂掉了？
- 讲一下redis的数据类型和使用场景
- k8s的储存方式是怎样的？
- Spring AOP原理是什么？怎么使用？什么是切点，什么是切面？最好是举个例子
- 算法题：给一堆硬币的array，返回所有的组合

阿里巴巴总监面（34分钟）

- 算法：给一个set打印出所有子集；多线程从多个文件中读入数据，写到同一个文件中；判断ip是否在给定范围内；打乱一副扑克牌，不能用额外空间，证明为什么是随机的。
- Tcp和udp区别
- 线程池的原理以及各种线程池的应用场景
- 线程池中使用有限的阻塞队列和无限的阻塞队列的区别
- 如果你发现你的sql语句始终走另一个索引，但是你希望它走你想要的索引，怎么办？
- mysql执行计划
- 数据库索引为什么用b+树？
- 你在做sql优化主要从哪几个方面做，用到哪些方法工具？
- 有没有想问的？

阿里巴巴HR面（23分钟）

- 自我介绍
- 平时怎么学习的？
- 有什么兴趣爱好吗？

- 怎么看待996?
- 怎么平衡工作和学习?
- 有没有什么想问的

总结

社招时面试新零售部门，主要因为准备不充分，面试又比较紧张，所以发挥不是很好，三面之后没有了后续。之后意识到学习的重要性，平时多拿出时间来学习，后来幸运地拿到内推资格，为了把握住这次机会，做了很多准备，好在已经拿到offer。

-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. 淘宝为什么能抗住双 11 ?
2. 为什么 ?阿里规定超过 3 张表禁止 join
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



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！

如何使用 JavaScript 录制屏幕？

Java后端 2019-09-08

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

屏幕录制? 截屏? 网页生成图片? 帧图? 说到录屏, 我一开始想到的是前面这些词。大致的想法是持续的生成当前页面的截图, 然后把这些帧图再合并成一个视频文件。前端页面生成图片我们应该比较熟悉的是html2canvas。另外也有一些现成的库可以使用来进行屏幕的录制, RecordRTC上就有很多屏幕录制的实现。有声音 (Audio) 、视频 (Video) 、屏幕 (Screen) 的录制; 有针对canvas的录制等等, 一共有三十多个示例。这里主要想简单的讲一讲原生的 Screen Capture API。参见: Using the Screen Capture API

一、屏幕内容的捕获

```
1 navigator.mediaDevices.getDisplayMedia()
```

该方法会返回一个promise, 该promise会resolve当前屏幕内容的实时数据流。

使用 **async / await** 实现如下:

```
1 async function startCapture(displayMediaOptions) {
2   let captureStream = null
3 ;
4
5   try {
6     captureStream = await navigator.mediaDevices.getDisplayMedia(displayMediaOptions);
7   } catch(err) {
8     console.error("Error: " + err)
9   }
10 }
11 return captureStream;
12 }
```

使用 **promise** 的方式实现如下:

```
1 function startCapture(displayMediaOptions) {
2   let captureStream = null
3 ;
4
5   return navigator.mediaDevices.getDisplayMedia(displayMediaOptions)
6     .catch(err => { console.error("Error:" + err); return null; })
7 ;
8 }
```

我们在获取屏幕数据的时候有可能会获取到一些敏感信息, 所有在使用**getDisplayMedia**的时候, 为了安全考虑, 会弹出一个选择框, 然后用户自己选择需要共享那一部分的内容。可以共享当前屏幕, 也可以共享其他的应用窗口和浏览器的其他标签页。

共享屏幕

http://127.0.0.1:5500想要共享您屏幕上的内容。请选择您希望共享哪些内容。



二、参数配置：

我们在上面的实现中可以看到，传递给startCapture函数的参数为displayMediaOptions。这个参数是用于配置返回数据流的。数据形式如下：

```
1 const displayMediaOptions = {
2   video: {
3     cursor: "never"
4   },
5   audio: false,
6   logicalSurface: false,
7 };
```

可以针对音视频做详细的配置：

```
1 const gdmOptions = {
2   video: {
3     cursor: "always" // 始终显示鼠标信息
4   },
5   // audio 配置信息是可选的
6   audio: {
7     echoCancellation: true,
8     noiseSuppression: true,
9     sampleRate: 44100
10 }
11 }
```

三、示例

HTML:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8"
5  >
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <meta http-equiv="X-UA-Compatible" content="ie=edge">
8      <title>Screen Record</title>
9  >
10     <link rel="stylesheet" href="./css/index.css"
11  >
12  </head>
13  <body>
14      <p>This example shows you the contents of the selected part of your display.
15      Click the Start Capture button to begin.</p>
16  >
17      <p><button id="start">Start Capture</button>&ampnbsp<button id="stop">Stop Capture</button></p>
18  >
19      <video id="video" autoplay></video>
20  >
21      <br>
22  >
23      <strong>Log:</strong>
24  >
25      <br>
26  >
27      <pre id="log"></pre>
28  >
29      <script src="./js/index.js"></script>
30  >
31  </body>
32  </html>
```

CSS:

```
1  #video {
2      border: 1px solid #999
3  ;
4      width: 98%
5  ;
6      max-width: 860px
7  ;
8  }
9  .error {
10     color: red;
11 }
```

```
12   .warn {
13     color: orange;
14   }
15   .info {
16     color: darkgreen;
17 }
```

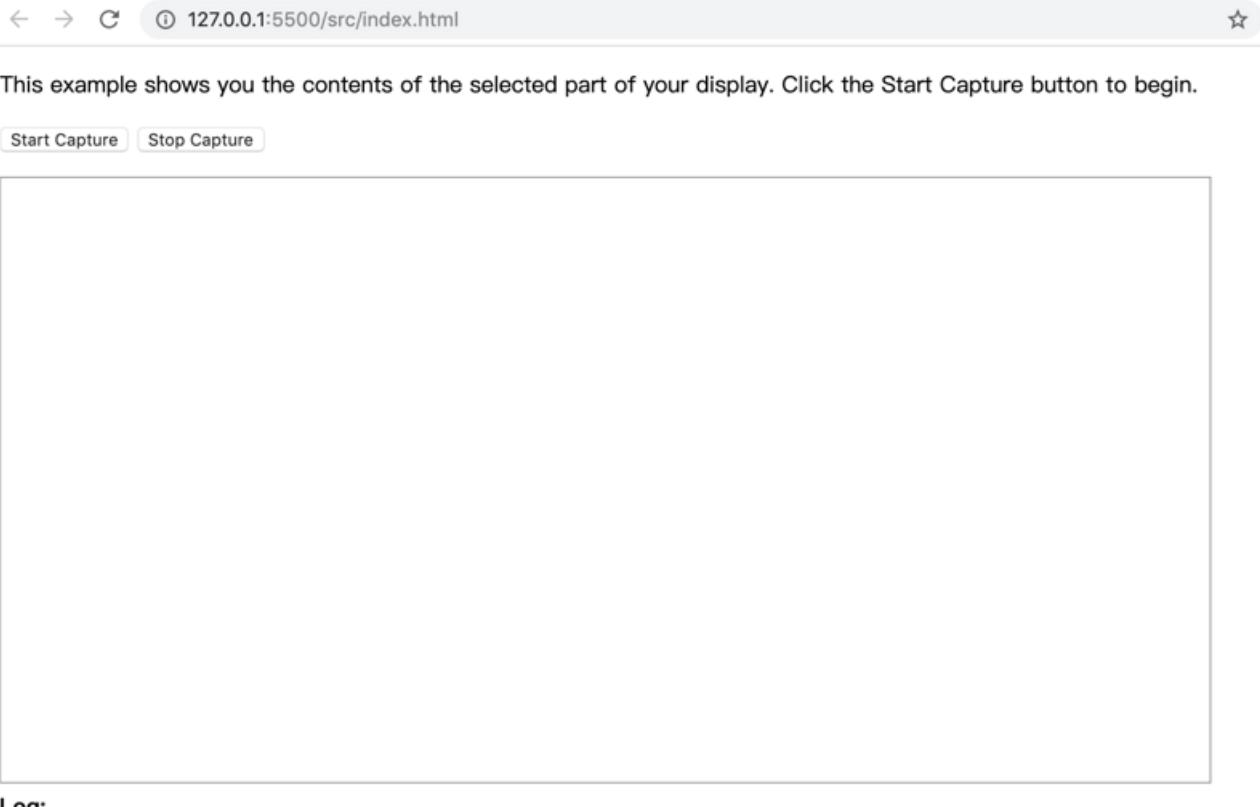
JS:

```
1 const videoElem = document.getElementById("video");
2 const logElem = document.getElementById("log")
3 ;
4 const startElem = document.getElementById("start");
5 const stopElem = document.getElementById("stop");
6
7 const displayMediaOptions = {
8   video: {
9     cursor: "never"
10 },
11   audio: false
12 };
13
14 startElem.addEventListener("click", function(evt) {
15   startCapture();
16 }, false)
17 ;
18 stopElem.addEventListener("click", function(evt) {
19   stopCapture();
20 }, false)
21 ;
22 console.log = msg => logElem.innerHTML += `${msg}<br>`
23 ;
24 console.error = msg => logElem.innerHTML += `<span class="error">${msg}</span><br>`
25 ;
26 console.warn = msg => logElem.innerHTML += `<span class="warn">${msg}</span><br>`
27 ;
28 console.info = msg => logElem.innerHTML += `<span class="info">${msg}</span><br>`
29 ;
30
31 async function startCapture() {
32   logElem.innerHTML = ""
33 ;
34
35   try {
36     videoElem.srcObject = await navigator.mediaDevices.getDisplayMedia(displayMediaOptions);
37     dumpOptionsInfo();
38   } catch(err) {
39     console.error("Error: " + err)
40   }
41 }
42 }
```

```
44 function stopCapture(evt) {
45   let tracks = videoElem.srcObject.getTracks();
46
47   tracks.forEach(track => track.stop());
48   videoElem.srcObject = null
49 ;
50 }
```

```
function dumpOptionsInfo() {
  const videoTrack = videoElem.srcObject.getVideoTracks()[0]
;
  console.info("Track settings:");
  console.info(JSON.stringify(videoTrack.getSettings(), null, 2));
  console.info("Track constraints:")
;
  console.info(JSON.stringify(videoTrack.getConstraints(), null, 2));
}
```

效果如下：



点击Start Capture之后选择需要共享的部分就可以共享如下的内容：

This example shows you the contents of the selected part of your display. Click the Start Capture button to begin.

[Start Capture](#) [Stop Capture](#)

```

1 const videoElem = document.getElementById("video");
2 const logElem = document.getElementById("log");
3 const startElem = document.getElementById("start");
4 const stopElem = document.getElementById("stop");
5 // Options for getDisplayMedia()
6 const displayMediaOptions = {
7   video: {
8     cursor: "never"
9   },
10   audio: false
11 };
12 // Set event listeners for the start and stop buttons
13 startElem.addEventListener("click", function(evt) {
14   startCapture();
15 }, false);
16 stopElem.addEventListener("click", function(evt) {
17   stopCapture();
18 }, false);
19 console.log = msg => logElem.innerHTML += `${msg}<br>`;
20 console.error = msg => logElem.innerHTML += `${msg}</span><br>`;
21 console.warn = msg => logElem.innerHTML += `${msg}</span><br>`;
22 console.info = msg => logElem.innerHTML += `${msg}</span><br>`;
23
24 async function startCapture() {
25   logElem.innerHTML = "";
26
27   try {
28     videoElem.srcObject = await navigator.mediaDevices.getDisplayMedia(displayMediaOptions);
29     dumpOptionsInfo();
30   } catch(err) {
31     console.error("Error: " + err);
32   }
}

```

点击Stop Capture即可 · 停止录制共享。

这个例子只是调取接口获取到当前分享屏幕的数据流，并通过video的形式显示出来。我们在拿到数据流信息这个，可以把这些信息上传到服务器，生成相应的视频文件。也可以结合websocket之类的处理方式，实现实时的屏幕共享功能。

作者：饭等米

链接：segmentfault.com/a/1190000020267689

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！

如何使用 Java 灵活读取 Excel 内容？

Java后端 3月9日

以下文章来源于日拱一兵，作者tan日拱一兵



日拱一兵

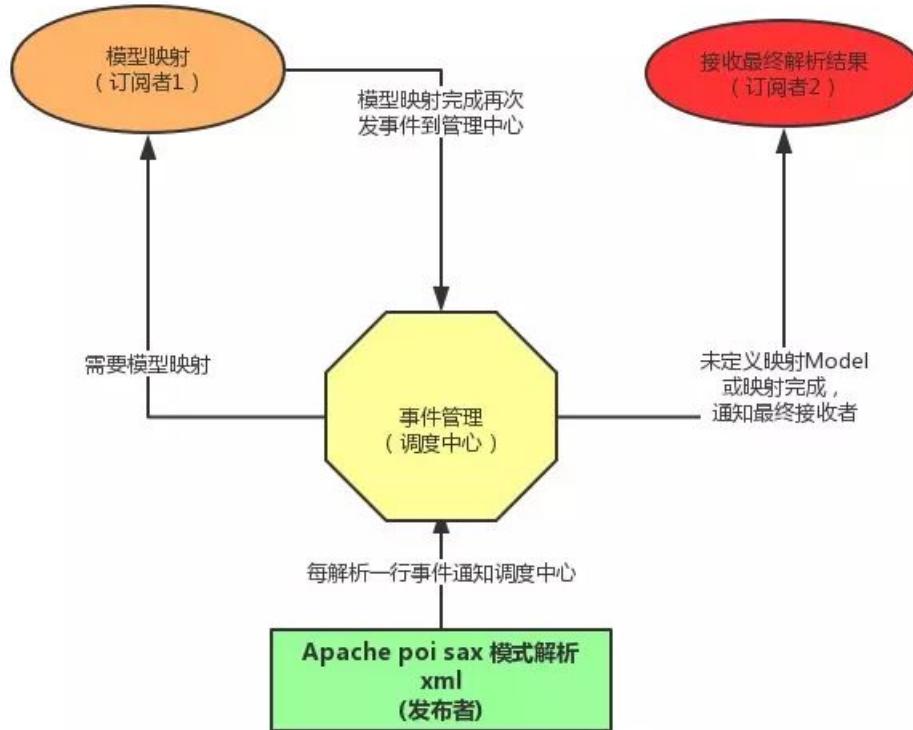
像读侦探小说一样趣读Java技术

写在前面

Java 后端程序员应该会遇到读取 Excel 信息到 DB 等相关需求，脑海中可能突然间想起 Apache POI 这个技术解决方案，但是当 Excel 的数据量非常大的时候，你也许发现，POI 是将整个 Excel 的内容全部读出来放入到内存中，所以内存消耗非常严重，如果同时进行包含大数据量的 Excel 读操作，很容易造成内存溢出问题。

本博文源码在公众号：Java后端，后台回复 excel 获取。

但 EasyExcel 的出现很好的解决了 POI 相关问题，原本一个 3M 的 Excel 用 POI 需要 100M 左右内存，而 EasyExcel 可以将其降低到几 M，同时再大的 Excel 都不会出现内存溢出的情况，因为是逐行读取 Excel 的内容（老规矩，这里不用过分关心下图，脑海中有个印象即可，看完下面的用例再回看这个图，就很简单了）



另外 EasyExcel 在上层做了模型转换的封装，不需要 cell 等相关操作，让使用者更加简单和方便，且看

简单读

假设我们 excel 中有以下内容：

	A	B
1	姓名	年龄
2	张三	28
3	李四	27
4	韩梅梅	27
5	李雷	28
6	王二毛	26
7		

我们需要新建 User 实体，同时为其添加成员变量

```
@Data
public class User {

    @ExcelProperty(index = 0)
    private String name;

    @ExcelProperty(index = 1)
    private Integer age;
}
```

你也许关注到了 @ExcelProperty 注解，同时使用了 index 属性 (0 代表第一列, 以此类推)，该注解同时支持以「列名」name 的方式匹配，比如：

```
@ExcelProperty("姓名")
private String name;
```

按照 github 文档的说明：

不建议 index 和 name 同时用，要么一个对象只用index，要么一个对象只用name去匹配

1. 如果读取的 Excel 模板信息列固定，这里建议以 index 的形式使用，因为如果用名字去匹配，名字重复，会导致只有一个字段读取到数据，所以 index 是更稳妥的方式
2. 如果 Excel 模板的列 index 经常有变化，那还是选择 name 方式比较好，不用经常性修改实体的注解 index 数值
所以大家可以根据自己的情况自行选择

编写测试用例

```
@Test
public void readExcel(){
    String fileName = TestUtils.getPath() + "excel" + File.separator + "users1.xlsx";
    EasyExcel.read(fileName, User.class, new UserExcelListener()).sheet().doRead();
}
```

EasyExcel 类中重载了很多个 read 方法，这里不一一列举说明，请大家自行查看；同时 sheet 方法也可以指定 sheetNo，默认是第一个 sheet 的信息

上面代码的 new UserExcelListener() 异常醒目，这也是 EasyExcel 逐行读取 Excel 内容的关键所在，自定义 UserExcelListener 继承 AnalysisEventListener

```

@Slf4j
public class UserExcelListener extends AnalysisEventListener<User> {

    private static final int BATCH_COUNT = 2;
    List<User> list = new ArrayList<User>(BATCH_COUNT);

    @Override
    public void invoke(User user, AnalysisContext analysisContext) {
        log.info("解析到一条数据:{}" , JSON.toJSONString(user));
        list.add(user);
        if (list.size() >= BATCH_COUNT) {
            saveData();
            list.clear();
        }
    }

    @Override
    public void doAfterAllAnalysed(AnalysisContext analysisContext) {
        saveData();
        log.info("所有数据解析完成!");
    }

    private void saveData(){
        log.info("{}条数据, 开始存储数据库!", list.size());
        log.info("存储数据库成功!");
    }
}

```

到这里请回看文章开头的 EasyExcel 原理图, invoke 方法逐行读取数据, 对应的就是订阅者 1; doAfterAllAnalysed 方法对应的就是订阅者 2, 这样你理解了吗?

打印结果:

```

解析到一条数据:{"age":28,"name":"张三"}
解析到一条数据:{"age":27,"name":"李四"}
2条数据, 开始存储数据库!
存储数据库成功!
解析到一条数据:{"age":27,"name":"韩梅梅"}
解析到一条数据:{"age":28,"name":"李雷"}
2条数据, 开始存储数据库!
存储数据库成功!
解析到一条数据:{"age":26,"name":"王二毛"}
1条数据, 开始存储数据库!
存储数据库成功!
所有数据解析完成!

```

从这里可以看出, 虽然是逐行解析数据, 但我们可以自定义阈值, 完成数据的批处理操作, 可见 EasyExcel 操作的灵活性

自定义转换器

这是最基本的数据读写, 我们的业务数据通常不可能这么简单, 有时甚至需要将其转换为程序可读的数据

性别信息转换

比如 Excel 中新增「性别」列, 其性别为男/女, 我们需要将 Excel 中的性别信息转换成程序信息: 「1: 男; 2:女」

	A	B	C
1	姓名	年龄	性别
2	张三	28	男
3	李四	27	男
4	韩梅梅	27	女
5	李雷	28	男
6	王二毛	26	女
7			

首先在 User 实体中添加成员变量 gender:

```
@ExcelProperty(index = 2)
private Integer gender;
```

EasyExcel 支持我们自定义 converter, 将 excel 的内容转换为我们程序需要的信息, 这里新建 GenderConverter, 用来转换性别信息

```
public class GenderConverter implements Converter<Integer> {

    public static final String MALE = "男";
    public static final String FEMALE = "女";

    @Override
    public Class supportJavaTypeKey() {
        return Integer.class;
    }

    @Override
    public CellDataTypeEnum supportExcelTypeKey() {
        return CellDataTypeEnum.STRING;
    }

    @Override
    public Integer convertToJavaData(CellData cellData, ExcelContentProperty excelContentProperty, GlobalConfiguration globalConfigura-
        String stringValue = cellData.getStringValue();
        if (MALE.equals(stringValue)){
            return 1;
        }else {
            return 2;
        }
    }

    @Override
    public CellData convertToExcelData(Integer integer, ExcelContentProperty excelContentProperty, GlobalConfiguration globalConfigurat
        return null;
    }
}
```

上面程序的 Converter 接口的泛型是指要转换的 Java 数据类型, 与 supportJavaTypeKey 方法中的返回值类型一致

打开注解 @ExcelProperty 查看, 该注解是支持自定义 Converter 的, 所以我们为 User 实体添加 gender 成员变量, 并指定 converter

```
@ExcelProperty(index = 2, converter = GenderConverter.class)
private Integer gender;
```

来看运行结果:

```
解析到一条数据:{"age":28 "gender":1, "name":"张三"}  
解析到一条数据: {"age":27 "gender":1, "name":"李四"}  
2条数据, 开始存储数据库!  
存储数据库成功!  
解析到一条数据: {"age":27 "gender":2, "name":"韩梅梅"}  
解析到一条数据: {"age":28 "gender":1, "name":"李雷"}  
2条数据, 开始存储数据库!  
存储数据库成功!  
解析到一条数据: {"age":26 "gender":2, "name":"王二毛"}  
1条数据, 开始存储数据库!  
存储数据库成功!  
所有数据解析完成!
```

数据按照我们预期做出了转换，从这里也可以看出，Converter 可以一次定义到处是用的便利性

日期信息转换

日期信息也是我们常见的转换数据，比如 Excel 中新增「出生年月」列，我们要解析成 `yyyy-MM-dd` 格式，我们需要将其进行格式化，EasyExcel 通过 `@DateTimeFormat` 注解进行格式化

	A	B	C	D
1	姓名	年龄	性别	出生日期
2	张三		28 男	1991年1月1日
3	李四		27 男	1992年2月2日
4	韩梅梅		27 女	1992年3月3日
5	李雷		28 男	1991年4月4日
6	王二毛		26 女	1993年5月5日
7				

在 User 实体中添加成员变量 `birth`，同时应用 `@DateTimeFormat` 注解，按照要求做格式化

```
@ExcelProperty(index = 3)  
@DateTimeFormat("yyyy-MM-dd HH:mm:ss")  
private String birth;
```

来看运行结果：

```
解析到一条数据: {"age":28, "birth":"1991-01-01 00:00:00", "gender":1, "name":"张三"}  
解析到一条数据: {"age":27, "birth":"1992-02-02 00:00:00", "gender":1, "name":"李四"}  
2条数据, 开始存储数据库!  
存储数据库成功!  
解析到一条数据: {"age":27, "birth":"1992-03-03 00:00:00", "gender":2, "name":"韩梅梅"}  
解析到一条数据: {"age":28, "birth":"1991-04-04 00:00:00", "gender":1, "name":"李雷"}  
2条数据, 开始存储数据库!  
存储数据库成功!  
解析到一条数据: {"age":26, "birth":"1993-05-05 00:00:00", "gender":2, "name":"王二毛"}  
1条数据, 开始存储数据库!  
存储数据库成功!  
所有数据解析完成!
```

如果这里你指定 `birth` 的类型为 `Date`，试试看，你得到的结果是什么？

到这里都是以测试的方式来编写程序代码，作为 Java Web 开发人员，尤其在目前主流 Spring Boot 的架构下，所以如何实现 Web 方式读取 Excel 的信息呢？

简单 Web

很简单，只是将测试用例的关键代码移动到 Controller 中即可，我们新建一个 UserController，在其添加 upload 方法

```
@RestController
@RequestMapping("/users")
@Slf4j
public class UserController {
    @PostMapping("/upload")
    public String upload(MultipartFile file) throws IOException {
        EasyExcel.read(file.getInputStream(), User.class, new UserExcelListener()).sheet().doRead();
        return "success";
    }
}
```

其实在写测试用例的时候你也许已经发现，listener 是以 new 的形式作为参数传入到 EasyExcel.read 方法中的，这是不符合 Spring IoC 的规则的，我们通常读取 Excel 数据之后都要针对读取的数据编写一些业务逻辑的，而业务逻辑通常又会写在 Service 层中，我们如何在 listener 中调用到我们的 service 代码呢？

先不要向下看，你脑海中有哪些方案呢？

匿名内部类方式

匿名内部类是最简单的方式，我们需要先新建 Service 层的信息：新建 IUser 接口：

```
public interface IUser {
    public boolean saveData(List<User> users);
}
```

新建 IUser 接口实现类 UserServiceImpl：

```
@Service
@Slf4j
public class UserServiceImpl implements IUser {
    @Override
    public boolean saveData(List<User> users) {
        log.info("UserService {}条数据，开始存储数据库！", users.size());
        log.info(JSON.toJSONString(users));
        log.info("UserService 存储数据库成功！");
        return true;
    }
}
```

接下来，在 Controller 中注入 IUser：

```
@Autowired
private IUser iUser;
```

修改 upload 方法，以匿名内部类重写 listener 方法的形式来实现：

```

@PostMapping("/uploadWithAnonyInnerClass")
public String uploadWithAnonyInnerClass(MultipartFile file) throws IOException {
    EasyExcel.read(file.getInputStream(), User.class, new AnalysisEventListener<User>(){
        private static final int BATCH_COUNT = 2;
        List<User> list = new ArrayList<User>();

        @Override
        public void invoke(User user, AnalysisContext analysisContext) {
            log.info("解析到一条数据:{}", JSON.toJSONString(user));
            list.add(user);
            if (list.size() >= BATCH_COUNT) {
                saveData();
                list.clear();
            }
        }

        @Override
        public void doAfterAllAnalysed(AnalysisContext analysisContext) {
            saveData();
            log.info("所有数据解析完成! ");
        }
    });

    private void saveData(){
        iUser.saveData(list);
    }
}).sheet().doRead();
return "success";
}

```

查看结果:

```

解析到一条数据:{"age":28,"birth":"1991-01-01 00:00:00","gender":1,"name":"张三"}
解析到一条数据:{"age":27,"birth":"1992-02-02 00:00:00","gender":1,"name":"李四"}
UserService 2条数据, 开始存储数据库
[{"age":28,"birth":"1991-01-01 00:00:00","gender":1,"name":"张三"}, {"age":27,"birth":"1992-02-02 00:00:00","gender":1,"name":"李四"}]
UserService 存储数据库成功!
解析到一条数据:{"age":27,"birth":"1992-03-03 00:00:00","gender":2,"name":"韩梅梅"}
解析到一条数据:{"age":28,"birth":"1991-04-04 00:00:00","gender":1,"name":"李雷"}
UserService 2条数据, 开始存储数据库
[{"age":27,"birth":"1992-03-03 00:00:00","gender":2,"name":"韩梅梅"}, {"age":28,"birth":"1991-04-04 00:00:00","gender":1,"name":"李雷"}]
UserService 存储数据库成功!
解析到一条数据:{"age":26,"birth":"1993-05-05 00:00:00","gender":2,"name":"王二毛"}
UserService 1条数据, 开始存储数据库
[{"age":26,"birth":"1993-05-05 00:00:00","gender":2,"name":"王二毛"}]
UserService 存储数据库成功!
所有数据解析完成!

```

这种实现方式, 其实这只是将 listener 中的内容全部重写, 并在 controller 中展现出来, 当你看着这么臃肿的 controller 是不是非常难受? 很显然这种方式不是我们的最佳编码实现

构造器传参

在之前分析 SpringBoot 统一返回源码时, 不知道你是否发现, Spring 底层源码多数以构造器的形式传参, 所以我们可以将为 listener 添加有参构造器, 将 Controller 中依赖注入的 IUser 以构造器的形式传入到 listener:

```
@Slf4j
public class UserExcelListener extends AnalysisEventListener<User> {

    private IUser iUser;

    public UserExcelListener(IUser iUser) {
        this.iUser = iUser;
    }

    private void saveData() {
        iUser.saveData(list);
    }
}
```

更改 Controller 方法:

```
@PostMapping("/uploadWithConstructor")
public String uploadWithConstructor(MultipartFile file) throws IOException {
    EasyExcel.read(file.getInputStream(), User.class, new UserExcelListener(iUser)).sheet().doRead();
    return "success";
}
```

运行结果: 同上

这样更改后, controller 代码看着很清晰,但如果后续业务还有别的 Service 需要注入,我们难道要一直添加有参构造器吗?很明显,这种方式同样不是很灵活。

其实在使用匿名内部类的时候,你也许会想到,我们可以通过 Java8 lambda 的方式来解决这个问题

Lambda 传参

为了解决构造器传参的痛点,同时我们又希望 listener 更具有通用性,没必要为每个 Excel 业务都新建一个 listener,因为 listener 都是逐行读取 Excel 数据,只需要将我们的业务逻辑代码传入给 listener 即可,所以我们需用到 Consumer<T>,将其作为构造 listener 的参数。

新建一个工具类 ExcelDemoUtils, 用来构造 listener:

```
public class ExcelDemoUtils {

    /**
     * 指定阈值
     * @param consumer
     * @param threshold
     * @param <T>
     * @return
     */
    public static <T> AnalysisEventListener<T> getListener(Consumer<List<T>> consumer, int threshold) {
        return new AnalysisEventListener<T>() {
            private LinkedList<T> linkedList = new LinkedList<T>();

            @Override
            public void invoke(T t, AnalysisContext analysisContext) {
                linkedList.add(t);
                if (linkedList.size() == threshold){
                    consumer.accept(linkedList);
                    linkedList.clear();
                }
            }

            @Override
            public void doAfterAllAnalysed(AnalysisContext analysisContext) {
                if (linkedList.size() > 0){
                    consumer.accept(linkedList);
                }
            }
        };
    }

    /**
     * 不指定阈值，阈值默认为10
     * @param consumer
     * @param <T>
     * @return
     */
    public static <T> AnalysisEventListener<T> getListener(Consumer<List<T>> consumer){
        return getListener(consumer, 10);
    }
}
```

我们看到，`getListener`方法接收一个`Consumer<List<T>>`的参数，这样下面代码被调用时，我们的业务逻辑也就会被相应的执行了：

```
consumer.accept(linkedList);
```

继续改造 Controller 方法：

```
@PostMapping("/uploadWithLambda")
public String uploadWithLambda(MultipartFile file) throws IOException {
    AnalysisEventListener<User> userAnalysisEventListener = ExcelDemoUtils.getListener(this.batchInsert(), 2);
    EasyExcel.read(file.getInputStream(), User.class, userAnalysisEventListener).sheet().doRead();
    return "success";
}

private Consumer<List<User>> batchInsert(){
    //其他业务逻辑只需要添加到该方法中即可
    return users -> iUser.saveData(users);
}
```

运行结果: 同上

到这里, 我们只需要将业务逻辑定制在 batchInsert 方法中:

1. 满足 Controller RESTful API 的简洁性
2. listener 更加通用和灵活, 它更多是扮演了抽象类的角色, 具体的逻辑交给抽象方法的实现来完成
3. 业务逻辑可扩展性也更好, 逻辑更加清晰

总结

到这里, 关于如何使用 EasyExcel 读取 Excel 信息的基本使用方式已经介绍完了, 还有很多细节内容没有讲, 大家可以自行查阅 EasyExcel Github 文档去发现更多内容。灵活使用 Java 8 的函数式接口, 更容易让你提高代码的复用性, 同时看起来更简洁规范

除了读取 Excel 的读取, 还有 Excel 的写入, 如果需要将其写入到指定位置, 配合 HuTool 的工具类 FileWriter 的使用是非常方便的, 针对 EasyExcel 的使用, 如果大家有什么问题, 也欢迎到博客下方探讨。

- E N D -

推荐阅读

1. IntelliJ IDEA 快捷键 Windows 版本
2. IntelliJ IDEA 常用快捷键 - Mac版本
3. 盘点那些改变过世界的代码
4. 基于token的多平台身份认证架构设计



微信搜一搜

Q Java后端

文章已于修改

[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！

如何使用 Java 生成二维码？

dunwu 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 | dunwu

来源 | github.com/dunwu/java-tutorial

简介

ZXing 是一个开源 Java 类库用于解析多种格式的 1D/2D 条形码。目标是能够对QR编码、Data Matrix、UPC的1D条形码进行解码。其提供了多种平台下的客户端包括：J2ME、J2SE和Android。

官网：[ZXing github仓库](https://github.com/zxing/zxing)

实战

本例演示如何在一个非 android 的 Java 项目中使用 ZXing 来生成、解析二维码图片。

安装

maven项目只需引入依赖：

```
<dependency>
<groupId>com.google.zxing</groupId>
<artifactId>core</artifactId>
<version>3.3.0</version>
</dependency>
<dependency>
<groupId>com.google.zxing</groupId>
<artifactId>javase</artifactId>
<version>3.3.0</version>
</dependency>
```

如果非maven项目，就去官网下载发布版本：[下载地址](https://github.com/zxing/zxing/releases)

生成二维码图片

ZXing 生成二维码图片有以下步骤：

1. com.google.zxing.MultiFormatWriter 根据内容以及图像编码参数生成图像2D矩阵。
2. com.google.zxing.client.j2se.MatrixToImageWriter 根据图像矩阵生成图片文件或图片缓存 BufferedImage。

```
public void encode(String content, String filepath) throws WriterException, IOException {
    int width = 100;
    int height = 100;
    Map<EncodeHintType, Object> encodeHints = new HashMap<EncodeHintType, Object>();
    encodeHints.put(EncodeHintType.CHARACTER_SET, "UTF-8");
    BitMatrix bitMatrix = new MultiFormatWriter().encode(content, BarcodeFormat.QR_CODE, width, height, encodeHints);
    Path path = FileSystems.getDefault().getPath(filepath);
    MatrixToImageWriter.writeToPath(bitMatrix, "png", path);
}
```

解析二维码图片

ZXing 解析二维码图片有以下步骤：

1. 使用 javax.imageio.ImageIO 读取图片文件，并存为一个 java.awt.image.BufferedImage 对象。
2. 将 java.awt.image.BufferedImage 转换为 ZXing 能识别的 com.google.zxing.BinaryBitmap 对象。
3. com.google.zxing.MultiFormatReader 根据图像解码参数来解析 com.google.zxing.BinaryBitmap。

```
public String decode(String filepath) throws IOException, NotFoundException {
    BufferedImage bufferedImage = ImageIO.read(new FileInputStream(filepath));
    LuminanceSource source = new BufferedImageLuminanceSource(bufferedImage);
    Binarizer binarizer = new HybridBinarizer(source);
    BinaryBitmap bitmap = new BinaryBitmap(binarizer);
    HashMap<DecodeHintType, Object> decodeHints = new HashMap<DecodeHintType, Object>();
    decodeHints.put(DecodeHintType.CHARACTER_SET, "UTF-8");
    Result result = new MultiFormatReader().decode(bitmap, decodeHints);
    return result.getText();
}
```

完整参考示例：测试例代码

以下是一个生成的二维码图片示例：



[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！

如何更新线上的 Java 服务器代码

未分配微服务 Java后端 1周前

Java后端

来源: cnblogs.com/orange911/p/10583245.html

一、前言

1、热更新代码的场景

(1) 当线上服务器出现问题时，有些时候现有的手段不足以发现问题所在，可能需要追加打印日志或者增加一些调试代码，如果我们去改代码重新部署，会破坏问题现场，可以通过热部署的手段来增加调试代码

(2) 线上出现紧急bug，通过Review代码找到问题，修改好后打包部署的流程可能比较久，可以通过热部署代码及时解决问题

二、Arthas的使用

使用阿里巴巴开源的Java诊断工具---Arthas，他可以附着在我们的Java服务器进程上面，查看服务器状态，jvm状态等各种参数指标，还可以进行热更新

1、下载启动Arthas

```
wget https://alibaba.github.io/arthas/arthas-boot.jar  
java -jar arthas-boot.jar
```

2、启动后会显示当前机器上面所有的java进程，选择我们需要监控/修改的进程，输入序号回车

3、一些常用命令，如果线上出现问题，可以通过以下命令查看各项指标是否有异常

```
dashboard——当前系统的实时数据面板  
thread——查看当前 JVM 的线程堆栈信息  
jvm——查看当前 JVM 的信息  
sysprop——查看和修改JVM的系统属性  
sysenv——查看JVM的环境变量  
getstatic——查看类的静态属性
```

(1) 打印前五名最消耗CPU的线程，可以及时找到CPU过高的代码位置

```
thread -n  
5
```

(2) 查看某个函数的调用堆栈

```
stack <类全包名> <函数名>
```

(3) 查看某个函数的哪个子调用最慢，耗时最长的调用会标红显示，可以方便找出某个功能中最耗时的操作

```
trace <类全包名> <函数名>
```

(4) 监控某个函数的调用统计数据，包括总调用次数，平均运行时间，成功率等信息

```
monitor <类全包名> <函数名>
```

4、输入exit可以退出当前的连接，但是附着在服务器进程上的Arthas依然在运行，完全退出可以输入shutdown

三、热更新

1、首先找到我们需要更新代码的全包名，通过jad命令将线上正在运行的代码反编译出来

```
jad --source-only <全包名> <导出目录+文件名>
```

2、拿到java代码后，我们根据需求来修改代码，需要注意的是这里热更新代码的实际原理是调用Java基础类java.lang.instrument.Instrumentation的redefineClasses方法，他可以通过修改字节码来替换已有的class文件，其中有诸多的限制：

(1) 比如不能增加或删除field/method

(2) 没有退出的函数不能生效，比如一个函数体内是一个where(true)循环，永远不会结束，那么我们修改的代码也永远不会生效

我们可以在函数中增加一些代码，比如增加日志打印等

3、修改好代码后，我们要找到这个类对应的类加载器，再去加载这个class，执行如下命令会返回类加载器的对象地址

```
sc -d <全包名> | grep classLoaderHash
```

4、通过内存编译将Java文件编译成Class文件

```
mc -c <类加载器的对象地址> <Java文件所在目录+文件名>
```

5、最后，我们通过命令将class文件进行热更新

```
redefine <Class文件所在目录+文件名>
```

6、更新完毕不出意外会立即生效，这时候就可以去验证代码是否生效了

- E N D -

推荐阅读

[1. Spring 中运用的 9 种设计模式吗？](#)

[2. Java 中的继承和多态（深入版）](#)

[3. 如何降低程序员的工资？](#)

[4. 编写 Spring MVC 的 14 个小技巧](#)



微信搜一搜

Q 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！

如何热更新线上的 Java 服务器代码

未分配微服务 Java后端 2月23日



微信搜一搜

Java后端

来源 | 未分配微服务

链接 | cnblogs.com/orange911/p/10583245.html

一、前言

1、热更新代码的场景

- (1) 当线上服务器出现问题时，有些时候现有的手段不足以发现问题所在，可能需要追加打印日志或者增加一些调试代码，如果我们去改代码重新部署，会破坏问题现场，可以通过热部署的手段来增加调试代码
- (2) 线上出现紧急bug，通过Review代码找到问题，修改好后打包部署的流程可能比较久，可以通过热部署代码及时解决问题

二、Arthas的使用

使用阿里巴巴开源的Java诊断工具---Arthas，他可以附着在我们的Java服务器进程上面，查看服务器状态，jvm状态等各种参数指标，还可以进行热更新

1、下载启动Arthas

```
wget https://alibaba.github.io/arthas/arthas-boot.jar  
java -jar arthas-boot.jar
```

2、启动后会显示当前机器上面所有的java进程，选择我们需要监控/修改的进程，输入序号回车

3、一些常用命令，如果线上出现问题，可以通过以下命令查看各项指标是否有异常

```
dashboard——当前系统的实时数据面板  
thread——查看当前 JVM 的线程堆栈信息  
jvm——查看当前 JVM 的信息  
sysprop——查看和修改JVM的系统属性  
sysenv——查看JVM的环境变量  
getstatic——查看类的静态属性
```

(1) 打印前五名最消耗CPU的线程，可以及时找到CPU过高的代码位置

```
thread -n 5
```

(2) 查看某个函数的调用堆栈

```
stack <类全包名> <函数名>
```

(3) 查看某个函数的哪个子调用最慢，耗时最长的调用会标红显示，可以方便找出某个功能中最耗时的操作

```
trace <类全包名> <函数名>
```

- (4) 监控某个函数的调用统计数据，包括总调用次数，平均运行时间，成功率等信息

```
monitor <类全包名> <函数名>
```

4、输入exit可以退出当前的连接，但是附着在服务器进程上的Arthas依然在运行，完全退出可以输入shutdown

三、热更新

1、首先找到我们需要更新代码的全包名，通过jad命令将线上正在运行的代码反编译出来

```
jad --source-only <全包名> > <导出目录+文件名>
```

2、拿到java代码后，我们根据需求来修改代码，需要注意的是这里热更新代码的实际原理是调用Java基础类java.lang.instrument.Instrumentation的redefineClasses方法，他可以通过修改字节码来替换已有的class文件，其中有诸多的限制：

(1) 比如不能增加或删除field/method

(2) 没有退出的函数不能生效，比如一个函数体内是一个where(true)循环，永远不会结束，那么我们修改的代码也永远不会生效

我们可以在函数中增加一些代码，比如增加日志打印等

3、修改好代码后，我们要找到这个类对应的类加载器，再去加载这个class，执行如下命令会返回类加载器的对象地址

```
sc -d <全包名> | grep classLoaderHash
```

4、通过内存编译将Java文件编译成Class文件

```
mc -c <类加载器的对象地址> <Java文件所在目录+文件名>
```

5、最后，我们通过命令将class文件进行热更新

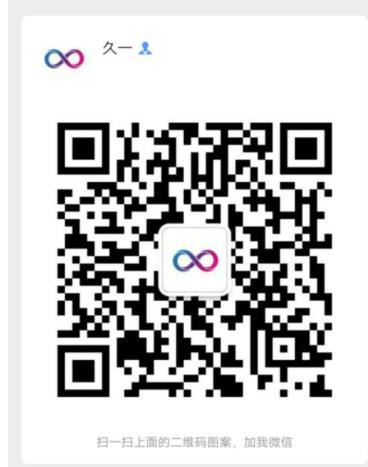
```
redefine <Class文件所在目录+文件名>
```

6、更新完毕不出意外会立即生效，这时候就可以去验证代码是否生效了

- 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. [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

[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！

如何衡量一个人的 JavaScript 水平？

Java后端 2019-10-18



zhihu.com/question/22855484/answer/657320514

1、立即执行函数

立即执行函数，即Immediately Invoked Function Expression (IIFE)，正如它的名字，就是创建函数的同时立即执行。它没有绑定任何事件，也无需等待任何异步操作：

```
1 (function()
2 {
3 // 代码
4 })();
```

function(){…}是一个匿名函数，包围它的一对括号将其转换为一个表达式，紧跟其后的一对括号调用了这个函数。

立即执行函数也可以理解为立即调用一个匿名函数。**立即执行函数**最常见的应用场景就是：将var变量的作用域限制于你们函数内，这样可以避免命名冲突。

2、闭包

对于闭包(closure)，当外部函数返回之后，内部函数依然可以访问外部函数的变量。

```
1 function f1()
2 {
3 var N = 0; // N是f1函数的局部变
4 量
5 function f2()
6 {
7 // f2是f1函数的内部函数，是闭包
8     N += 1; // 内部函数f2中使用了外部函数f1中的变量
9
10    N
11    console.log(N);
12 }
13 return f2;
14 }
15 var result = f1();
16 result(); // 输出1
17 result(); // 输出2
```

```
result(); // 输出3
```

代码中，外部函数**f1**只执行了一次，变量**N**设为**0**，并将内部函数**f2**赋值给了变量**result**。

由于外部函数**f1**已经执行完毕，其内部变量**N**应该在内存中被清除，然而事实并不是这样：我们每次调用**result**的时候，发现变量**N**一直在内存中，并且在累加。为什么呢？这就是闭包的神奇之处了！

3、使用闭包定义私有变量

通常，JavaScript开发者使用下划线作为私有变量的前缀。但是实际上，这些变量依然可以被访问和修改，并非真正的私有变量。这时，使用闭包可以定义真正的私有变量：

```
1 function Product()
2 {
3     var name;
4
5     this.setName = function(value)
6     {
7         name = value;
8     };
9
10    this.getName = function()
11    {
12        return name;
13    };
14 }
15
16 var p = new Product();
17 p.setName("Fundebug");
18
19 console.log(p.name); // 输出undefined
20 console.log(p.getName()); // 输出Fundebug
```

代码中，对象**p**的的**name**属性为私有属性，使用**p.name**不能直接访问。

4、prototype

每个JavaScript构造函数都有一个**prototype**属性，用于设置所有实例对象需要共享的属性和方法。

prototype属性不能列举。JavaScript仅支持通过**prototype**属性进行继承属性和方法。

```
1 function Rectangle(x, y)
2 {
3     this._length = x;
4     this._breadth = y;
5 }
6 Rectangle.prototype.getDimensions = function()
```

```

7  {
8    return {
9      length: this._length,
10     breadth: this._breadth
11   };
12 };
13 var x = new Rectangle(3, 4)
14 ;
15 var y = new Rectangle(4, 3)
16 ;
17
18 console.log(x.getDimensions()); // { length: 3, breadth: 4 }
19 console.log(y.getDimensions()); // { length: 4, breadth: 3 }

```

代码中，**x**和**y**都是构造函数**Rectangle**创建的对象实例，它们通过prototype继承了**getDimensions**方法。

5、模块化

JavaScript并非模块化编程语言，至少ES6落地之前都不是。然而对于一个复杂的Web应用，模块化编程是一个最基本的要求。

这时，可以使用**立即执行函数**来实现模块化，正如很多JS库比如jQuery以及我们Fundebug都是这样实现的。

```

1 var module = (function()
2 {
3   var N = 5
4 ;
5   function print(x)
6   {
7     console.log("The result is: " + x)
8   ;
9   }
10  function add(a)
11  {
12    var x = a + N;
13    print(x);
14  }
15  return {
16    description: "This is description"
17  ,
18    add: add
19  };
20 })();
21 console.log(module.description); // 输出"this is description"
22 module.add(5); // 输出"The result is: 10"

```

所谓模块化，就是根据需要控制模块内属性与方法的可访问性，即私有或者公开。在代码中，**module**为一个独立的模块，**N**为其私有属性，**print**为其私有方法，**description**为其公有属性，**add**为其共有方法。

6、变量提升

JavaScript会将所有变量和函数声明移动到它的作用域的最前面，这就是所谓的**变量提升(Hoisting)**。也就是说，无论你在什么地方声明变量和函数，解释器都会将它们移动到作用域的最前面。因此我们可以先使用变量和函数，而后声明它们。

但是，仅仅是变量声明被提升了，而变量赋值不会被提升。如果你不明白这一点，有时则会出错：

```
1 console.log(y); // 输出undefined
2 y = 2; // 初始化
```

上面的代码等价于下面的代码：

```
1 var y; // 声明
2 console.log(y); // 输出undefined
3 y = 2; // 初始化
y
```

为了避免BUG，开发者应该在每个作用域开始时声明变量和函数。

7、柯里化

柯里化，即**Currying**，可以是函数变得更加灵活。我们可以一次性传入多个参数调用它；也可以只传入一部分参数来调用它，让它返回一个函数去处理剩下的参数。

```
1 var add = function(x)
2 {
3   return function(y)
4   {
5     return x + y;
6   };
7 };
8
9 console.log(add(1)(1)); // 输出
10 2
11
12 var add1 = add(1);
13 console.log(add1(1)); // 输出2

var add10 = add(10);
console.log(add10(1)); // 输出11
```

代码中，我们可以一次性传入2个1作为参数**add(1)(1)**，也可以传入1个参数之后获取**add1**与**add10**函数，这样使用起来非常灵活。

8、apply, call与bind方法

JavaScript开发者有必要理解**apply**、**call**与**bind**方法的不同点。它们的共同点是第一个参数都是**this**，即函数运行时依赖的上下文。

三者之中，**call**方法是最简单的，它等价于指定**this**值调用函数：

```
1 var user = {  
2   name: "Rahul Mhatre"  
3 ,  
4   whatIsYourName: function()  
5 {  
6     console.log(this.name);  
7   }  
8 };  
9 user.whatIsYourName(); // 输出"Rahul Mhatre",  
10 var user2 = {  
11   name: "Neha Sampat"  
12 };  
13 user.whatIsYourName.call(user2); // 输出"Neha Sampat"
```

apply方法与**call**方法类似。两者唯一的不同点在于，**apply**方法使用数组指定参数，而**call**方法每个参数单独需要指定：

- **apply(thisArg, [argsArray])**
- **call(thisArg, arg1, arg2, …)**

```
1 var user = {  
2   greet: "Hello!"  
3 ,  
4   greetUser: function(userName)  
5 {  
6     console.log(this.greet + " " + userName);  
7   }  
8 };  
9 var greet1 = {  
10   greet: "Hola"  
11 };  
12 user.greetUser.call(greet1, "Rahul"); // 输出"Hello Rahul"  
13 user.greetUser.apply(greet1, ["Rahul"]); // 输出"Hello Rahul"
```

使用**bind**方法，可以为函数绑定**this**值，然后作为一个新的函数返回：

```
1 var user = {  
2   greet: "Hello!"  
3 ,  
4   greetUser: function(userName)  
5 {  
6     console.log(this.greet + " " + userName);  
7   }  
8 };
```

```
9 var greetHola = user.greetUser.bind({greet: "Hola"})
10 ;
11 var greetBonjour = user.greetUser.bind({greet: "Bonjour"})
12 ;
13 greetHola("Rahul") // 输出"Hello Rahul"
14 greetBonjour("Rahul") // 输出"Bonjour Rahul"
```

9、Memoization

Memoization用于优化比较耗时的计算，通过将计算结果缓存到内存中，这样对于同样的输入值，下次只需要从内存中读取结果。

```
1 function memoizeFunction(func)
2 {
3     var cache = {};
4     return function()
5     {
6         var key = arguments[0]
7 ;
8         if (cache[key]) {
9             return cache[key];
10        } else
11        {
12            var val = func.apply(this, arguments)
13 ;
14            cache[key] = val;
15        }
16    }
17 };
18 }
var fibonacci = memoizeFunction(function(n)
{
    return n === 0 || n === 1 ? n : fibonacci(n - 1) + fibonacci(n - 2
);
});
console.log(fibonacci(100)); // 输出354224848179262000000
console.log(fibonacci(100)); // 输出354224848179262000000
```

代码中，第2次计算**fibonacci(100)**则只需要在内存中直接读取结果。

10、函数重载

所谓**函数重载(method overloading)**，就是函数名称一样，但是输入输出不一样。或者说，允许某个函数有各种不同输入，根据不同的输入，返回不同的结果。

凭直觉，**函数重载**可以通过**if...else**或者**switch**实现，这就不去管它了。jQuery之父John Resig提出了一个非常巧(bian)妙(tai)的方法，利用了闭包。

从效果上来说，**people**对象的**find**方法允许3种不同的输入：0个参数时，返回所有人名；1个参数时，根据**firstName**查找人名并返回；2个参数时，根据完整的名称查找人名并返回。

难点在于，**people.find**只能绑定一个函数，那它为何可以处理3种不同的输入呢？它不可能同时绑定3个函数**find0, find1与find2**啊！这里的关键在于**old**属性。

由**addMethod**函数的调用顺序可知，**people.find**最终绑定的是**find2**函数。然而，在绑定**find2**时，**old**为**find1**；同理，绑定**find1**时，**old**为**find0**。3个函数**find0, find1与find2**就这样通过闭包链接起来了。

根据**addMethod**的逻辑，当**f.length**与**arguments.length**不匹配时，就会去调用**old**，直到匹配为止。

```
1 function addMethod(object, name, f)
2 {
3     var old = object[name];
4     object[name] = function()
5     {
6         // f.length为函数定义时的参数个数
7         // arguments.length为函数调用时的参数个数
8         if (f.length === arguments.length) {
9             return f.apply(this, arguments);
10        }
11        } else if (typeof old === "function")
12        {
13            return old.apply(this, arguments);
14        }
15    };
16 }
17
18 // 不传参数时，返回所有name
19 function find0()
20 {
21     return this.names;
22 }
23
24 // 传一个参数时，返回firstName匹配的name
25 function find1(firstName)
26 {
27     var result = [];
28     for (var i = 0; i < this.names.length; i++) {
29         if (this.names[i].indexOf(firstName) === 0)
30         {
31             result.push(this.names[i]);
32         }
33     }
34     return result;
35 }
36
37 // 传两个参数时，返回firstName和lastName都匹配的name
38 function find2(firstName, lastName)
39 {
40     var result = [];
41     for (var i = 0; i < this.names.length; i++) {
42         if (this.names[i] === firstName + " " + lastName) {
```

```
42         result.push(this.names[i]);
43     }
44 }
45 return result;
46 }
47
48 var people = {
49     names: ["Dean Edwards", "Alex Russell", "Dean Tom"
50 ]
51 };
52
53 addMethod(people, "find", find0);
54 addMethod(people, "find", find1);
55 addMethod(people, "find", find2);
56
57 console.log(people.find()); // 输出["Dean Edwards", "Alex Russell", "Dean Tom"]
58 console.log(people.find("Dean")); // 输出["Dean Edwards", "Dean Tom"]
59 console.log(people.find("Dean", "Edwards")); // 输出["Dean Edwards"]
```

-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. 感受 Lambda 之美，推荐收藏，需要时查阅
2. 如何优雅的导出 Excel
3. 文艺互联网公司 vs 二逼互联网公司
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



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！

如何阅读 Java 源码？

Java后端 2019-10-10

点击上方 Java后端, 选择 [设为星标](#)

技术博文, 及时送达

作者 | IamDing

链接 | blog.csdn.net/dj673344908/article/details/81701595

阅读 Java 源码的前提条件：

1、技术基础

在阅读源码之前，我们要有一定程度的技术基础的支持。

假如你从来都没有学过Java, 也没有其它编程语言的基础, 上来就啃《Core Java》, 那样是很难有收获的, 尤其是《深入Java虚拟机》这类书, 或许别人觉得好, 但是未必适合现在的你。

比如设计模式, 许多Java源码当中都会涉及到。再比如阅读Spring源码的时候, 势必要先对IOC, AOP, Java动态代理等知识点有所了解。

2、强烈的求知欲

强烈的求知欲是阅读源码的核心动力！

大多数程序员的学习态度分为如下几个层次：

- 完成自己的项目就可以了, 遇到不懂的地方就百度一下。
- 不仅做好项目, 还会去阅读一些和项目有关的书籍。
- 除了阅读和项目相关的书籍之外, 还会阅读一些IT行业相关的书籍。
- 平时会经常逛逛GitHub, 找一些开源项目看看。
- 阅读基础框架、J2EE规范、源码。

大多数程序员的层次都是在第一层, 到第五层的人就需要有强烈的求知欲了。

3、足够的耐心

通过阅读源码我们可以学习大佬的设计思路, 技巧。还可以把我们一些零碎的知识点整合起来, 从而融会贯通。总之阅读源码的好处多多, 想必大家也清楚。

但是真的把那么庞大复杂的代码放到你的眼前时, 肯定会在阅读的过程中卡住, 就如同陷入了一个巨大的迷宫, 如果想要在这个巨大的迷宫中找到一条出路, 那就需要把整个迷宫的整体结构弄清楚, 比如: API结构、框架的设计图。而且还有理解它的核心思想, 确实很不容易。

刚开始阅读源码的时候肯定会很痛苦, 所以, 没有足够的耐心是万万不行的。

如何读Java源码：

团长也是经历过阅读源码种种痛苦的人, 算是有一些成功的经验吧, 今天来给大家分享一下。

如果你已经有了一年左右的Java开发经验的话，那么你就有阅读Java源码的技术基础了。

1、建议从JDK源码开始读起，这个直接和eclipse集成，不需要任何配置。

可以从JDK的工具包开始，也就是我们学的《数据结构和算法》Java版，如List接口和ArrayList、LinkedList实现，HashMap和TreeMap等。这些数据结构里也涉及到排序等算法，一举两得。

面试时，考官总喜欢问ArrayList和Vector的区别，你花10分钟读读源码，估计一辈子都忘不了。

然后是core包，也就是String、StringBuffer等。如果你有一定的Java IO基础，那么不妨读读FileReader等类。

建议大家看看《Java In A Nutshell》，里面有整个Java IO的架构图。Java IO类库，如果不理解其各接口和继承关系，则阅读始终是一头雾水。

Java IO 包，我认为是对继承和接口运用得最优雅的案例。如果你将来做架构师，你一定会经常和它打交道，如项目中部署和配置相关的核心类开发。

读这些源码时，只需要读懂一些核心类即可，如和ArrayList类似的二三十个类，对于每一个类，也不一定要每个方法都读懂。像String有些方法已经到虚拟机层了(native方法)，如hashCode方法。

当然，如果有兴趣，可以对照看看JRockit的源码，同一套API，两种实现，很有意思的。

如果你再想钻的话，不妨看看针对虚拟机的那套代码，如System ClassLoader的原理，它不在JDK包里，JDK是基于它的。JDK的源码Zip包只有10来M，它像是有50来M，Sun公司有下载的，不过很隐秘。我曾经为自己找到、读过它很兴奋了一阵。

2、Java Web项目源码阅读

步骤：表结构 → web.xml → mvc → db → spring ioc → log → 代码

① 先了解项目数据库的表结构，这个方面是最容易忘记的，有时候我们只顾着看每一个方法是怎么进行的，却没有去了解数据库之间的主外键关联。其实如果先了解数据库表结构，再去看一个方法的实现会更加容易。

② 然后需要过一遍web.xml，知道项目中用到了什么拦截器，监听器，过滤器，拥有哪些配置文件。如果是拦截器，一般负责过滤请求，进行AOP等；如果是监听器，可能是定时任务，初始化任务；配置文件有如 使用了spring后的读取mvc相关，db相关，service相关，aop相关的文件。

③ 查看拦截器，监听器代码，知道拦截了什么请求，这个类完成了怎样的工作。有的人就是因为缺少了这一步，自己写了一个action，配置文件也没有写错，但是却怎么调试也无法进入这个action，直到别人告诉他，请求被拦截了。

④ 接下来，看配置文件，首先一定是mvc相关的，如springmvc中，要请求哪些请求是静态资源，使用了哪些view策略，controller注解放在哪个包下等。然后是db相关配置文件，看使用了什么数据库，使用了什么orm框架，是否开启了二级缓存，使用哪种产品作为二级缓存，事务管理的处理，需要扫描的实体类放在什么位置。最后是spring核心的ioc功能相关的配置文件，知道接口与具体类的注入大致是怎样的。当然还有一些如aspectj等的配置文件，也是在这个步骤中完成。

⑤ log相关文件，日志的各个级别是如何处理的，在哪些地方使用了log记录日志。

⑥ 从上面几点后知道了整个开源项目的整体框架，阅读每个方法就不再那么难了。

⑦ 当然如果有项目配套的开发文档也是要阅读的。

3、Java框架源码阅读

当然了，就是Spring、MyBatis这类框架。

在读Spring源码前，一定要先看看《J2EE Design and Development》这本书，它是Spring的设计思路。注意，不是中文版，中文版完全被糟蹋了。

想要阅读MyBatis的源码就要先了解它的一些概念，否则云里来雾里去的什么也不懂。有很多人会选择去买一些书籍来帮助阅读，当然这是可取的。那么如果不想的话，就可以去官网查看它的介绍（MyBatis网站：<http://www.mybatis.org/mybatis-3/zh/getting-started.html>），团长也是按照官网上面的介绍来进行源码阅读的。团长认为MyBatis的亮点就是管理SQL语句。

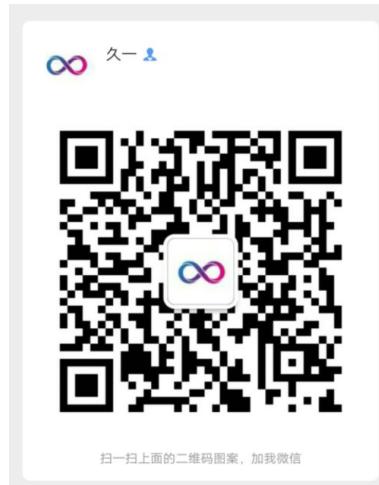
总结

没有人一开始就可以看得懂那些源码，我们都是从0开始的，而且没有什么捷径可寻，无非就是看我们谁愿意花时间去研究，谁的求知欲更强烈，谁更有耐心。阅读源码的过程中我们的能力肯定会提升，可以从中学到很多东西。在我们做项目的时候就会体现出来了，的确会比以前顺手很多。

-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. 团队开发中 Git 最佳实践
2. Google 出品 Java 编码规范，强烈推荐！
3. 支付系统高可用架构设计实战
4. 重构你乱糟糟的代码
5. 如何设计 API 接口，实现统一格式返回？



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！

实用：9个可以快速掌握的Java性能调优技巧

DZone 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

来源 | DZone

链接 | <http://sina.lt/gnWz>

1. 在明确必要之前别急着优化
2. 使用分析器找到真正的瓶颈
3. 为整个应用程序创建一个性能测试套件
4. 优先关注最大瓶颈
5. 使用 `StringBuilder` 以编程方式连接字符串
6. 尽可能使用基本类型
7. 尽量避免大整数和小数
8. 使用 Apache Commons StringUtils.Replace 而不是 `String.replace`
9. 昂贵的缓存资源, 如数据库连接

大多数开发者认为性能优化是一个复杂的话题, 它需要大量的工作经验和相关知识理论。好吧, 这也不完全错。

优化一个应用做到性能最优化可能不是件容易的任务, 但是这并不意味着你没有相关的知识就什么也做不了。这里有一些易于遵循的建议和最佳实践可以帮助你创建一个性能良好的应用程序。

这些建议的大部分都是针对 Java 语言的。但是也有一些是跟语言无关的, 你可以运用到任意的应用和程序中。在我们学习特定的 Java 编程性能调优之前, 先来探讨一些通用的技巧。

1. 在明确必要之前别急着优化

这可能是最重要的性能优化技巧之一。你应该遵循常见的最佳实践做法并在案例中高效地应用它。但是这并不意味在证明必要之前, 你应该更换任何标准库或构建复杂的优化。

多数情况下, 过早地优化会占用大量的时间, 而且会使代码变得难以理解和阅读。更糟糕的是, 这些优化通常并没带来任何好处, 因为你花了大量的时间在优化应用中的非关键部分。

那么, 要怎么证明东西需要优化呢?

首先, 你需要定义你的代码速度得多快。例如, 为所有 API 调用指定最大响应时间, 或者指定在特定时间范围内要导入的记录数量。在做完这些后, 你需要确定你应用中哪些部分太慢需要改进。当完成这些后, 你就可以来看看第二个技巧提示。

2. 使用分析器找到真正的瓶颈

在完成第一部分的优化建议以鉴别出你应用中需要提升的部分后, 要从哪里入手呢?

你可以有两种途径来解决这个问题:

- 查看你的代码, 从看起来可疑的或者你觉得可能会导致出现问题的地方入手。
- 或者使用分析器获取代码每个部分的行为(执行过程)和性能的详细信息。

希望我不需要解释为什么应该始终遵循第二种途径/方法的原因。

很显然，基于分析器的方式可以让你更好地理解代码的性能影响，并允许你去专注于更关键的部分(代码)。即使你曾经使用过分析器，你一定记得你曾经多么惊讶于一下就找到了代码的哪些部分产生了性能问题。我第一次的猜测不止一次地导致我走错了方向。

3. 为整个应用程序创建一个性能测试套件

这是另一个通用的可以帮助你避免在将性能改进部署到产品中之后经常会发生许多意外问题的技巧。你应该总是定义一个性能测试套件来测试整个应用程序，并在性能改进之前和之后运行它。

这些额外的测试运行将帮助你识别你的改动所引起的功能和性能上的副作用，并确保不会导致弊大于利的更新。如果你处理的是被应用程序的多个不同部分使用的组件，如数据库或缓存，那这一点尤为重要。

4. 优先关注最大瓶颈

在创建了测试套件并使用分析器分析你的应用程序之后，你可以列出一系列需要解决以提高性能的问题列表。这很好，但这并没有回答你需要从哪里开始的问题。你可以专注于速成方案，或从最重要的问题开始。

速成方案一开始可能会很有吸引力，因为你可以很快显示第一个成果。但有时，可能有必要说服其他团队成员或管理层认为性能分析是值得的。

一般来说，我建议从顶层开始，首先开始处理最重要的性能问题。这将为你提供最大的性能改进，而且你可能仅需要解决这些问题中的一小部分就能满足你的性能要求。

常见的通用调优技巧到此结束。接下来让我们仔细看看一些特定于 Java 的技巧。

5. 使用 StringBuilder 以编程方式连接字符串

在 Java 中有很多不同的选项来连接字符串。例如，你可以使用简单的 + 或 +=，以及老的 StringBuffer 或 StringBuilder。

那么，你应该选择哪种方法呢？

答案取决于连接字符串的代码。如果你是以编程方式将新内容添加到字符串中，例如在for循环中，则应使用 StringBuilder。它很容易使用，并提供比 StringBuffer 更好的性能。但请记住，与 StringBuffer 相比， StringBuilder 不是线程安全的，可能并不适用于所有情况。

你只需要实例化一个新的 StringBuilder 并调用append方法来向String中添加一个新的部分。在你添加完了所有的部分后，你可以调用toString()方法来检索已连接的字符串。下面的代码片段展示了一个简单的例子。

在每次迭代期间，该循环将 i 转换为一个 String，并将其与空格一起添加到 StringBuilder sb 中。所以，最后，这段代码在日志文件中写入 “This is a test0 1 2 3 4 5 6 7 8 9”。

```
StringBuilder sb = new StringBuilder("This is a test");for (int i=0; i<10; i++) {    sb.append(i);    sb.append(" ");}log.info(sb.toString());
```

正如你在代码片段中看到的。我们可以为字符串的第一个元素提供到构造函数中。这会创建一个 StringBuilder，其中包含了你所提供的字符串以及 16 个额外字符的容量。当你向 StringBuilder 中添加更多字符时，你的 JVM 将动态的增加 StringBuilder 的

大小。

如果你已经知道字符串将包含多少个字符，则可以将该数字提供给不同的构造方法以实例化具有指定容量的 `StringBuilder`。这进一步提高了效率，因为它不需要动态扩展其容量。

6. 尽可能使用基本类型

避免任何开销并提高应用程序性能的另一种简便快速的方法是使用基本类型而不是其包装类。所以，最好使用 `int` 而不是 `Integer`，是 `double` 而不是 `Double`。这将使得你的 JVM 将值存储在堆栈而不是堆中，以减少内存消耗，并更有效地处理它。

7. 尽量避免大整数和小数

由于我们已经在讨论数据类型，所以我们也应该快速浏览大整数和小数。尤其是后者因其精确性而受欢迎。但这是有代价的。大整数和小数比一个简单的 `long` 型或 `double` 型需要更多的内存，并会显著减慢所有的运算。所以，如果你需要额外的精度，或者如果你的数字超出一个较长的范围，最好要三思。这可能是你需要更改并解决性能问题的唯一方法，尤其是在实现数学算法时。

8. 使用 Apache Commons StringUtils.Replace 而不是 String.replace

一般来说，`String.replace` 方法可以正常工作，并且效率很高，尤其是在你使用 Java 9 的情况下。但是，如果你的应用程序需要大量的替换操作，并且没有更新到最新的 Java 版本，那么检查更快和更有效的替代品依然是有必要的。

有一种候选方案是 Apache Commons Lang 的 `StringUtils.replace` 方法。正如 Lukas Eder 在他最近的一篇博客文章中所描述的，它远远胜过了 Java 8 的 `String.replace` 方法。

而且它只需要很小的改动。你只需要将 Apache Commons Lang 项目的 Maven 依赖项添加到你的应用程序的 `pom.xml` 中，并将 `String.replace` 的所有调用替换为 `StringUtils.replace` 方法。

```
// replace this
test.replace( "test" , "simple test" );
// with this
StringUtils.replace(test, "test" , "simple test" );
```

9. 昂贵的缓存资源，如数据库连接

缓存是避免重复执行昂贵或常用代码片段的流行解决方案。总的思路很简单：重复使用这些资源比创建一个新的资源更划算。

一个典型的例子是缓存池中的数据库连接。新连接的创建需要时间，如果你重用现有连接，则可以避免这种情况。

你也可以在 Java 语言源码中找到其他的例子。例如，在 `Integer` 类中的 `valueOf` 方法缓存了介于 -128 到 127 之间的值。你可能会说创建一个新的 `Integer` 并不是太昂贵，但是由于它经常被使用，因此缓存最常用的值也可以提供性能优势。

但是，当你考虑使用缓存时，请记住缓存实现也会产生开销。你需要花费额外的内存来储存可重复使用的资源，因此你可能需要管理你的缓存以使资源可访问，并删除过期的资源。

所以，在开始缓存任何资源之前，请确保它们是经常使用的，以超过缓存实现的开销(代价)。

总结

正如你所看到的，有时不需要太多的工作就可以提高你的应用程序的性能。本文中的大部分建议只需要稍作努力就可以将它们应用于你的代码中。

但还是那句话，最重要的还是那些与是什么编程语言无关的技巧：

- 在你知道其必要性之前不要进行优化
- 使用分析器（profiler）来查找真正的瓶
- 优先处理最大的瓶颈

【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！

常见排序算法总结 - Java 实现

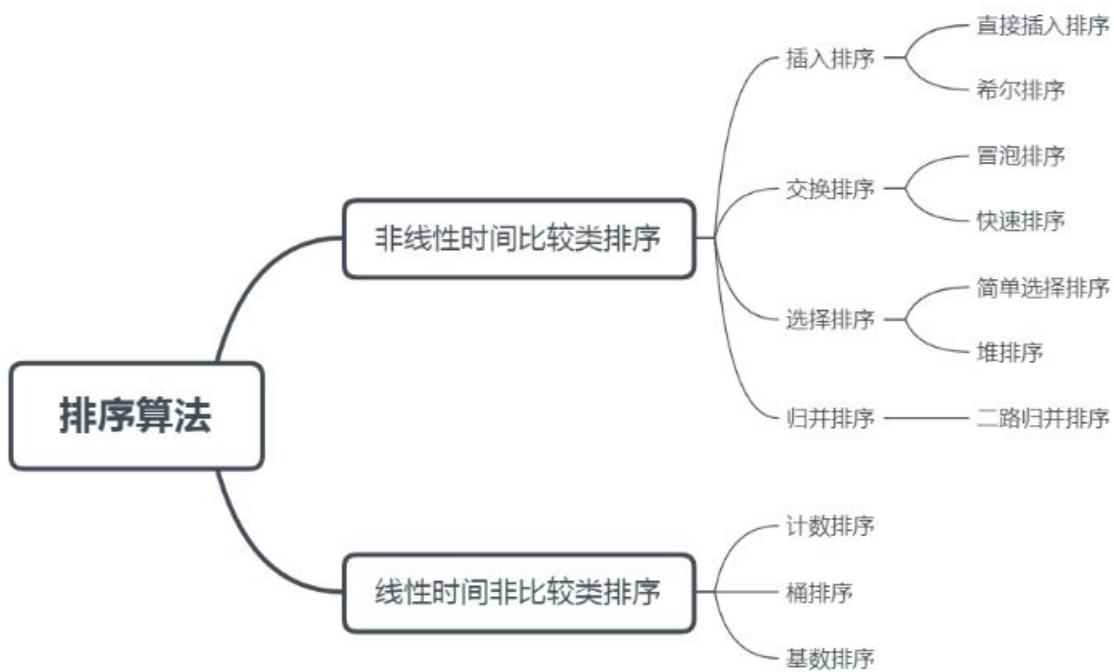
蓝桥 Java后端 2019-11-22

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 小蓝

来源 | 蓝桥 (id:lanqiaocenter)



排序算法可以分为两大类：

1、非线性时间比较类排序：通过比较来决定元素间的相对次序, 由于其时间复杂度不能突破 $O(n \log n)$, 因此称为非线性时间比较类排序。

2、线性时间非比较类排序：不通过比较来决定元素间的相对次序, 它可以突破基于比较排序的时间下界, 以线性时间运行, 因此称为线性时间非比较类排序。

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	$O(n \log_2 n)$	不稳定
归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

相关概念

- 稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面。
- 不稳定：如果a原本在b的前面，而a=b，排序之后 a 可能会出现在 b 的后面。
- 时间复杂度：对排序数据的总的操作次数。反映当n变化时，操作次数呈现什么规律。
- 空间复杂度：是指算法在计算机内执行时所需存储空间的度量，它也是数据规模n的函数。

一、插入排序

1.1 直接插入排序

插入排序的基本操作就是将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据，算法适用于少量数据的排序，时间复杂度为 $O(n^2)$ 。是稳定的排序方法。

直接插入排序的算法思路：

- 1、设置监视哨temp，将待插入记录的值赋值给temp；
- 2、设置开始查找的位置j；
- 3、在数组arr中进行搜索，搜索中将第j个记录后移，直至 $temp \geq arr[j]$ 为止；
- 4、将temp插入arr[j+1]的位置上。

```

/***
 * 直接插入排序
 */
public void insertSort(int[] arr) {
    //外层循环确定待比较数值
    //必须i=1，因为开始从第二个数与第一个数进行比较
    for (int i = 1; i < arr.length; i++) {
        //待比较数值
        int temp = arr[i];
        int j = i - 1;
        //内层循环为待比较数值确定其最终位置
        //待比较数值比前一位置小，应插往前插一位
        for (; j >= 0 && arr[j] > temp; j--) {
            //将大于temp的值整体后移一个单位
            arr[j + 1] = arr[j];
        }
        //待比较数值比前一位置大，最终位置无误
        arr[j + 1] = temp;
    }
}

```

1.2 希尔排序

希尔排序(Shell's Sort)是插入排序的一种，又称“缩小增量排序”(Diminishing Increment Sort)，是直接插入排序算法的一种更高效的改进版本。希尔排序是非稳定排序算法。该方法因D.L.Shell于1959年提出而得名。

希尔排序的算法思路：

- 1、把数组按下标的一定增量分组；
- 2、对每组使用直接插入排序算法排序；
- 3、随着增量逐渐减少，每组包含的值越来越多，当增量减至1时，整个文件被分成一组，算法便终止。

```

/***
 * 希尔排序
 */
public void shellSort(int[] arr) {
    int d = arr.length;
    while (d >= 1) {
        d = d / 2;
        for (int x = 0; x < d; x++) {
            //按下标的一一定增量分组然后进行插入排序
            for (int i = x + d; i < arr.length; i = i + d) {
                int temp = arr[i];
                int j;
                for (j = i - d; j >= 0 && arr[j] > temp; j = j - d) {
                    //移动下标
                    arr[j + d] = arr[j];
                }
                arr[j + d] = temp;
            }
        }
    }
}

```

二、交换排序

2.1 冒泡排序

在一组数据中，相邻元素依次比较大小，最大的放后面，最小的冒上来。

冒泡排序算法的算法思路：

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

```
/**  
 * 冒泡排序  
 */  
public void bubbleSort(int[] arr) {  
    for (int i = 0; i < arr.length - 1; i++) {  
        for (int j = 0; j < arr.length - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                // temp 临时存储 arr[j] 的值  
                int temp = arr[j];  
                // 交换 arr[j] 和 arr[j+1] 的值  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    }  
}
```

2.2 快速排序

快速排序(Quicksort)是对冒泡排序的一种改进。

通过一次排序将数组分成两个子数组，其中一个数字的值都比另外一个数字的值小，然后再对这两子数组分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

快速排序的算法思路：(分治法)

1. 先从数列中取出一个数作为中间值middle；
2. 将比这个数小的数全部放在它的左边，大于或等于它的数全部放在它的右边；
3. 对左右两个小数列重复第二步，直至各区间只有1个数。

```
/**  
 * 冒泡排序  
 */  
public void bubbleSort(int[] arr) {  
    for (int i = 0; i < arr.length - 1; i++) {  
        for (int j = 0; j < arr.length - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                // temp 临时存储 arr[j] 的值  
                int temp = arr[j];  
                // 交换 arr[j] 和 arr[j+1] 的值  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    }  
}
```

```

/**
 * 查找出中轴 ( 默认是最低位low ) 的在arr数组排序后所在位置
 *
 * @param arr 待排序数组
 * @param low 开始位置
 * @param high 结束位置
 * @return 中轴所在位置
 */
private int getMiddle(int[] arr, int low, int high) {
    int temp = arr[low]; //数组的第一个作为中轴
    while (low < high) {
        while (low < high && arr[high] >= temp) {
            high--;
        }
        arr[low] = arr[high]; //比中轴小的记录移到低端
        while (low < high && arr[low] < temp) {
            low++;
        }
        arr[high] = arr[low]; //比中轴大的记录移到高端
    }
    arr[low] = temp; //中轴记录到尾
    return low; // 返回中轴的位置
}

```

三、选择排序

3.1 简单选择排序

选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小(大)元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小(大)元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

简单选择排序的算法思路：

- 1.首先在未排序序列中找到最小(大)元素，存放到排序序列的起始位置
- 2.再从剩余未排序元素中继续寻找最小(大)元素，然后放到已排序序列的末尾
- 3.以此类推，直到所有元素均排序完毕。

```

/**
 * 简单选择排序
 */
public void selectSort(int[] arr) {
    int minIndex = 0;
    int temp;
    for (int i = 0; i < arr.length - 1; i++) {
        minIndex = i;
        for (int j = i + 1; j < arr.length; j++) {
            // 找到当前循环最小值索引
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        temp = arr[i];
        // 交换当前循环起点值和最小值索引位置的值
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}

```

3.2 堆排序

堆排序(英语:Heap Sort)是指利用堆这种数据结构所设计的一种排序算法。堆是一个近似完全二叉树的结构,并同时满足堆积的性质:即子结点的键值或索引总是小于(或者大于)它的父节点。在堆的数据结构中,堆中的最大值总是位于根节点(在优先队列中使用堆的话堆中的最小值位于根节点)。

堆排序的算法思路:

1.最大堆调整(Max Heapify):将堆的末端子节点作调整,某个节点的值最多和其父节点的值一样大;

2.创建最大堆(Build Max Heap):将堆中的所有数据重新排序,堆中的最大元素存放在根节点中;

3.堆排序(HeapSort):移除位在第一个数据的根节点,并做最大堆调整的递归运算。

```
/*
 * 堆排序
 */
public void heapSort(int[] arr) {
    buildMaxHeap(arr);

    //进行n-1次循环,完成排序
    for (int i = arr.length - 1; i > 0; i--) {
        //最后一个元素和第一个元素进行交换
        int temp = arr[i];
        arr[i] = arr[0];
        arr[0] = temp;
        //筛选 R[0] 结点,得到i-1个结点的堆 将arr中前i-1个记录重新调整为大顶堆
        heapAdjust(arr, 0, i);
    }
}

/**
 * 构建大顶堆
 * <p>
 * 将数组中最大的值放在根节点
 * </p>
private void buildMaxHeap(int[] arr) {
    for (int i = arr.length / 2; i >= 0; i--) {
        heapAdjust(arr, i, arr.length - 1);
    }
}

/**
 * 堆调整
 * <p>
 * 将数组中最大的值放在根节点
 *
 * @param arr    待排序数组
 * @param parent 父节点索引
 * @param length 数组长度
 * </p>
private void heapAdjust(int[] arr, int parent, int length) {
    int temp = arr[parent]; //temp保存当前父节点
    int child = 2 * parent + 1; //获取左子节点
```

```
/***
 * 希尔排序
 */
public void shellSort(int[] arr) {
    int d = arr.length;
    while (d >= 1) {
        d = d / 2;
        for (int x = 0; x < d; x++) {
            //按下标的一定增量分组然后进行插入排序
            for (int i = x + d; i < arr.length; i = i + d) {
                int temp = arr[i];
                int j;
                for (j = i - d; j >= 0 && arr[j] > temp; j = j - d) {
                    //移动下标
                    arr[j + d] = arr[j];
                }
                arr[j + d] = temp;
            }
        }
    }
}
```

四、归并排序

4.1 二路归并排序

归并排序(mergeSort)是建立在归并操作上的一种有效的排序算法,该算法是采用分治法(Divide and Conquer)的一个非常典型的应用。将已有序的子序列合并,得到完全有序的序列;即先使每个子序列有序,再使子序列段间有序。

若将两个有序表合并成一个有序表,称为二路归并。例如:将2个有序数组合并。比较2个数组的第一个数,谁小就先取谁,取了后就在对应数组中删除这个数。然后再进行比较,如果有数组为空,那直接将另一个数组的数依次取出即可。

二路归并排序的算法思路:

- 1、将数组分成A, B 两个数组,如果这2个数组都是有序的,那么就可以很方便的将这2个数组进行排序。
- 2、让这2个数组有序,可以将A, B组各自再分成2个数组。依次类推,当分出来的数组只有1个数据时,可以认为数组已经达到了有序。
- 3、然后再合并相邻的2个数组。这样通过先递归的分解数组,再合并数组就完成了归并排序。

```

/**
 * 二路归并排序
 */
public void mergeSort(int[] arr) {
    int[] temp = new int[arr.length]; //临时数组
    sort(arr, temp, 0, arr.length - 1);
}

/**
 * @param arr 待排序数组
 * @param left 开始位置
 * @param right 结束位置
 */
private void sort(int[] arr, int[] temp, int left, int right) {
    if (left >= right) {
        return;
    }
    int mid = left + (right - left) / 2;
    sort(arr, temp, left, mid);
    sort(arr, temp, mid + 1, right);
    merge(arr, temp, left, mid, right);
}

/**
 * 将两个有序表归并成一个有序表
 *
 * @param arr 待排序数组
 * @param temp 临时数组
 * @param leftStart 左边开始下标
 * @param leftEnd 左边结束下标(mid)
 * @param rightEnd 右边结束下标
 */
private static void merge(int[] arr, int[] temp, int leftStart, int leftEnd, int rightEnd) {
    int rightStart = leftEnd + 1;
    int tempIndex = leftStart; // 从左边开始算
    int len = rightEnd - leftStart + 1; // 元素个数
    while (leftStart <= leftEnd && rightStart <= rightEnd) {
        if (arr[leftStart] <= arr[rightStart]) {
            temp[tempIndex++] = arr[leftStart++];
        } else {
            temp[tempIndex++] = arr[rightStart++];
        }
    }
    // 左边如果有剩余 将左边剩余的归并
    while (leftStart <= leftEnd) {
        temp[tempIndex++] = arr[leftStart++];
    }
    // 右边如果有剩余 将右边剩余的归并
    while (rightStart <= rightEnd) {
        temp[tempIndex++] = arr[rightStart++];
    }
    // 从临时数组拷贝到原数组
    for (int i = 0; i < len; i++) {
        arr[rightEnd] = temp[rightEnd];
        rightEnd--;
    }
}

```

五、计数排序

计数排序(Counting sort)不是基于比较的排序算法，其核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

计数排序的算法思路：

1、求出待排序数组的最大值 max 和最小值 min。

2、实例化辅助计数数组temp，temp数组中每个下标对应arr中的一个元素，temp用来记录每个元素出现的次数。

3、计算 arr 中每个元素在temp中的位置 position = arr[i] - min。

4、根据 temp 数组求得排序后的数组。

```
/*
 * 计数排序
 */
public void countSort(int[] arr) {
    if (arr == null || arr.length == 0) {
        return;
    }

    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;

    //找出数组中的最大最小值
    for (int i = 0; i < arr.length; i++) {
        max = Math.max(max, arr[i]);
        min = Math.min(min, arr[i]);
    }

    int[] temp = new int[max];

    //找出每个数字出现的次数
    for (int i = 0; i < arr.length; i++) {
        //每个元素在temp中的位置 position = arr[i] - min
        int position = arr[i] - min;
        temp[position]++;
    }

    int index = 0;
    for (int i = 0; i < temp.length; i++) {
        //temp[i] 大于0 表示有重复元素
        while (temp[i]-- > 0) {
            arr[index++] = i + min;
        }
    }
}
```

六、桶排序

桶排序 (Bucket sort) 的工作原理是将数组分到有限数量的桶里。每个桶再分别排序 (有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排序)。当要被排序的数组内的数值是均匀分配的时候，桶排序使用线性时间 ($\Theta(n)$)。但桶排序并不是比较排序，他不受到 $O(n \log n)$ 下限的影响，桶排序可用于最大最小值相差较大的数据情况。

桶排序的算法思路：

1、找出待排序数组中的最大值max和最小值min；

2、我们使用动态数组ArrayList 作为桶，桶里放的元素也用 ArrayList 存储。桶的数量为 $(\text{max}-\text{min}) / \text{arr.length} + 1$ ；

3、遍历数组 arr，计算每个元素 arr[i] 放的桶；

4、每个桶各自排序；

5、遍历桶数组，把排序好的元素放进输出数组。

```
/**
 * 桶排序
 *
 * @param arr 待排序数组
 */
public static void bucketSort(int[] arr) {
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < arr.length; i++) {
        max = Math.max(max, arr[i]);
        min = Math.min(min, arr[i]);
    }

    //桶数
    int bucketNum = (max - min) / arr.length + 1;
    ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketNum);
    for (int i = 0; i < bucketNum; i++) {
        bucketArr.add(new ArrayList<>());
    }

    //将每个元素放入桶
    for (int i = 0; i < arr.length; i++) {
        int num = (arr[i] - min) / arr.length;
        bucketArr.get(num).add(arr[i]);
    }

    //对每个桶进行排序
    for (int i = 0; i < bucketNum; i++) {
        Collections.sort(bucketArr.get(i));
    }

    int position = 0;
    //合并桶
    for (int i = 0; i < bucketNum; i++) {
        for (int j = 0; j < bucketArr.get(i).size(); j++) {
            arr[position++] = bucketArr.get(i).get(j);
        }
    }
}
```

七、基数排序

基数排序 (radix sort) 是桶排序的扩展，基本思想是将整数按位数切割成不同的数字，然后按每个位数分别比较。

基数排序法是属于稳定性的排序，其时间复杂度为 $O(n \log(r)m)$ ，其中 r 为所采取的基数，而 m 为堆数，在某些时候，基数排序法的效率高于其它的稳定性排序法。基数排序的算法思路：

- 1、取得数组中的最大数，并取得位数；
- 2、 arr 为原始数组，从最低位开始取每个位组成 $radix$ 数组；
- 3、对 $radix$ 进行计数排序（利用计数排序适用于小范围数的特点）。

```

/**
 * 基数排序
 *
 * @param arr 待排序数组
 */
public void radixSort(int[] arr) {
    int max = getMax(arr); // 数组arr中的最大值

    for (int exp = 1; max / exp > 0; exp *= 10) {
        //从个位开始，对数组arr按"exp指数"进行排序
        countSort(arr, exp);
        //bucketSort(arr, exp);
    }
}

/**
 * 获取数组中最大值
 */
private int getMax(int[] arr) {
    int max = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

/**
 * 对数组按照"某个位数"进行排序(计数排序)
 * <p>
 * 例如：
 * 1、当exp=1 表示按照"个位"对数组进行排序
 * 2、当exp=10 表示按照"十位"对数组进行排序
 *
 * @param arr 待排序数组
 * @param exp 指数 对数组arr按照该指数进行排序
 */
private void countSort(int[] arr, int exp) {
    int[] temp = new int[arr.length]; // 存储"被排序数据"的临时数组
    int[] buckets = new int[10];

    // 将数据出现的次数存储在buckets[]中
    for (int i = 0; i < arr.length; i++) {
        buckets[(arr[i] / exp) % 10]++;
    }

    // 计算数据在temp[]中的位置 0 1 2 2 3 --> 0 1 3 5 8
    for (int i = 1; i < 10; i++) {
        buckets[i] += buckets[i - 1];
    }

    // 将数据存储到临时数组temp[]中
    for (int i = arr.length - 1; i >= 0; i--) {
        temp[buckets[(arr[i] / exp) % 10] - 1] = arr[i];
        buckets[(arr[i] / exp) % 10]--;
    }

    // 将排序好的数据赋值给arr[]
    for (int i = 0; i < arr.length; i++) {
        arr[i] = temp[i];
    }
}

```

```
/*
 * 桶排序
 */
private void bucketSort(int[] arr, int exp) {
    int[][] buckets = new int[10][arr.length]; //这是二维数组组成的桶
    int[] counter = new int[10]; //此数组用来记录0-9每个桶中的数字个数，计数器
    for (int i = 0; i < arr.length; i++) {
        int index = (arr[i] / exp) % 10; //得出相应位置(如个位、十位)上的数字
        buckets[index][counter[index]] = arr[i]; //取出来放到桶里
        counter[index]++;
    }

    int position = 0;
    //合并桶
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < counter[i]; j++) {
            arr[position++] = buckets[i][j];
        }
    }
}
```

。

【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. 一千行 MySQL 学习笔记
2. 一份工作坚持多久跳槽最合适？
3. 项目中常用到的 19 条 MySQL 优化
4. 零基础认识 Spring Boot
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！

我们再来聊一聊 Java 的单例吧

张新强 Java后端 2019-11-23

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 张新强

链接 | www.barryzhang.com/archives/521

1. 前言

单例(Singleton) 应该是开发者们最熟悉的设计模式了, 并且好像也是最容易实现的——基本上每个开发者都能够随手写出——但是, 真的是这样吗?

作为一个Java开发者, 也许你觉得自己对单例模式的了解已经足够多了。我并不想危言耸听说一定还有你不知道的——毕竟我自己的了解也的确有限, 但究竟你自己了解的程度到底怎样呢? 往下看, 我们一起来聊聊看~

2. 什么是单例?

单例对象的类必须保证只有一个实例存在 ——这是维基百科上对单例的定义, 这也可以作为对意图实现单例模式的代码进行检验的标准。

对单例的实现可以分为两大类—— 懒汉式 和 饿汉式 , 他们的区别在于:

- 懒汉式 : 指全局的单例实例在第一次被使用时构建。
- 饿汉式 : 指全局的单例实例在类装载时构建。

从它们的区别也能看出来, 日常我们使用的较多的应该是 懒汉式 的单例, 毕竟按需加载才能做到资源的最大化利用嘛。

3. 懒汉式单例

先来看一下懒汉式单例的实现方式。

3.1 简单版本

看最简单的写法Version 1:

```
// Version 1
public class Single1 {
    private static Single1 instance;
    public static Single1 getInstance() {
        if (instance == null) {
            instance = new Single1();
        }
        return instance;
    }
}
```

或者再进一步, 把构造器改为私有的, 这样能够防止被外部的类调用。

```
// Version 1.1
public class Single1 {
    private static Single1 instance;
    private Single1() {}
    public static Single1 getInstance() {
        if (instance == null) {
            instance = new Single1();
        }
        return instance;
    }
}
```

我仿佛记得当初学校的教科书就是这么教的？—— 每次获取instance之前先进行判断，如果instance为空就new一个出来，否则就直接返回已存在的instance。

这种写法在大多数的时候也是没问题的。问题在于，当多线程工作的时候，如果有多个线程同时运行到 `if (instance == null)`，都判断为null，那么两个线程就各自会创建一个实例——这样一来，就不是单例了。

3.2 synchronized版本

那既然可能会因为多线程导致问题，那么加上一个同步锁吧！

修改后的代码如下，相对于Version1.1，只是在方法签名上多加了一个 `synchronized`：

```
// Version 2
public class Single2 {
    private static Single2 instance;
    private Single2() {}
    public static synchronized Single2 getInstance() {
        if (instance == null) {
            instance = new Single2();
        }
        return instance;
    }
}
```

OK，加上 `synchronized` 关键字之后，`getInstance`方法就会锁上了。如果有两个线程（T1、T2）同时执行到这个方法时，会有其中一个线程T1获得同步锁，得以继续执行，而另一个线程T2则需要等待，当第T1执行完毕`getInstance`之后（完成了null判断、对象创建、获得返回值之后），T2线程才会执行执行。——所以这端代码也就避免了Version1中，可能出现因为多线程导致多个实例的情况。

但是，这种写法也一个问题：给`getInstance`方法加锁，虽然会避免了可能会出现的多个实例问题，但是会强制除T1之外的所有线程等待，实际上会对程序的执行效率造成负面影响。

3.3 双重检查（Double-Check）版本

Version2代码相对于Version1d代码的效率问题，其实是为了解决1%几率的问题，而使用了一个100%出现的防护盾。那有一个优化的思路，就是把100%出现的防护盾，也改为1%的几率出现，使之只出现在可能会导致多个实例出现的地方。

——有没有这样的方法呢？当然是有的，改进后的代码Vsersion3如下：

```
// Version 3
public class Single3 {
    private static Single3 instance;
    private Single3() {}
    public static Single3 getInstance() {
        if (instance == null) {
            synchronized (Single3.class) {
                if (instance == null) {
                    instance = new Single3();
                }
            }
        }
        return instance;
    }
}
```

这个版本的代码看起来有点复杂，注意其中有两次 `if (instance == null)` 的判断，这个叫做『双重检查 Double-Check』。

- 第一个 `if (instance == null)`，其实是为了解决Version2中的效率问题，只有`instance`为`null`的时候，才进入 `synchronized` 的代码段——大大减少了几率。
- 第二个 `if (instance == null)`，则是跟Version2一样，是为了防止可能出现多个实例的情况。

这段代码看起来已经完美无瑕了。当然，只是『看起来』，还是有小概率出现问题的。

这弄清楚为什么这里可能出现问题，首先，我们需要弄清楚几个概念：`原子操作`、`指令重排`。

知识点：什么是原子操作？

简单来说，`原子操作 (atomic)` 就是不可分割的操作，在计算机中，就是指不会因为线程调度被打断的操作。

比如，简单的赋值是一个原子操作：

```
m = 6; // 这是个原子操作
```

假如`m`原先的值为0，那么对于这个操作，要么执行成功`m`变成了6，要么是没执行`m`还是0，而不会出现诸如`m=3`这种中间态——即使是在并发的线程中。

而，声明并赋值就不是一个原子操作：

```
int n = 6; // 这不是一个原子操作
```

对于这个语句，至少有两个操作：

- ①声明一个变量`n`
- ②给`n`赋值为6

——这样就会有一个中间状态：变量`n`已经被声明了但是还没有被赋值的状态。

——这样，在多线程中，由于线程执行顺序的不确定性，如果两个线程都使用`m`，就可能会导致不稳定的结果出现。

知识点：什么是指令重排？

简单来说，就是计算机为了提高执行效率，会做的一些优化，在不影响最终结果的情况下，可能会对一些语句的执行顺序进行调整。

比如，这一段代码：

```
int a; // 语句1  
a = 8; // 语句2  
int b = 9; // 语句3  
int c = a + b; // 语句4
```

正常来说，对于顺序结构，执行的顺序是自上到下，也即1234。

但是，由于 **指令重排** 的原因，因为不影响最终的结果，所以，实际执行的顺序可能会变成3124或者1324。

由于语句3和4没有原子性的问题，语句3和语句4也可能会拆分成原子操作，再重排。

——也就是说，对于非原子性的操作，在不影响最终结果的情况下，其拆分成的原子操作可能会被重新排列执行顺序。

OK，了解了 **原子操作** 和 **指令重排** 的概念之后，我们再继续看Version3代码的问题。

下面这段话直接从陈皓的文章(深入浅出单实例SINGLETON设计模式)中复制而来：

主要在于`singleton = new Singleton()`这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情。

1. 给 singleton 分配内存
2. 调用 Singleton 的构造函数来初始化成员变量，形成实例
3. 将singleton对象指向分配的内存空间（执行完这步 singleton才是非 null 了）

但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 `instance` 已经是非 null 了（但却没有初始化），所以线程二会直接返回 `instance`，然后使用，然后顺理成章地报错。

再稍微解释一下，就是说，由于有一个『`instance`已经不为null但是仍没有完成初始化』的中间状态，而这个时候，如果有其他线程刚好运行到第一层 `if (instance == null)` 这里，这里读取到的`instance`已经不为null了，所以就直接把这个中间状态的`instance`拿去用了，就会产生问题。

这里的关键在于——线程T1对`instance`的写操作没有完成，线程T2就执行了读操作。

3.4 终极版本：volatile

对于Version3中可能出现的问题（当然这种概率已经非常小了，但毕竟还是有的嘛~），解决方案是：只需要给`instance`的声明加上 **volatile** 关键字即可，Version4版本：

```
// Version 4  
public class Single4 {  
    private static volatile Single4 instance;  
    private Single4() {}  
    public static Single4 getInstance() {  
        if (instance == null) {  
            synchronized (Single4.class) {  
                if (instance == null) {  
                    instance = new Single4();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

`volatile` 关键字的一个作用是禁止 **指令重排**，把`instance`声明为 `volatile` 之后，对它的写操作就会有一个 **内存屏障**（什么是内存屏障？），这样，在它的赋值完成之前，就不用会调用读操作。

注意：`volatile`阻止的不`singleton = new Singleton()`这句话内部[1-2-3]的指令重排，而是保证了在一个写操作 ([1-2-3]) 完成之前，不会调用读操作 (`if (instance == null)`)。

也就彻底防止了Version3中的问题发生。

好了，现在彻底没什么问题了吧？

好了，别紧张，的确没问题了。大名鼎鼎的EventBus中，其入口方法 `EventBus.getDefault()` 就是用这种方法来实现的。

不过，非要挑点刺的话还是能挑出来的，就是这个写法有些复杂了，不够优雅、简洁。

(傲娇脸) (—_—)

4. 饿汉式单例

下面再聊了解一下饿汉式的单例。

如上所说，**饿汉式** 单例是指：指全局的单例实例在类装载时构建的实现方式。

由于类装载的过程是由类加载器（ClassLoader）来执行的，这个过程也是由JVM来保证同步的，所以这种方式先天就有一个优势——能够免疫许多由多线程引起的问题。

4.1 饿汉式单例的实现方式

饿汉式 单例的实现如下：

```
//饿汉式实现
public class SingleB {
    private static final SingleB INSTANCE = new SingleB();
    private SingleB() {}
    public static SingleB getInstance() {
        return INSTANCE;
    }
}
```

对于一个饿汉式单例的写法来说，它基本上是完美的了。

所以它的缺点也就只是饿汉式单例本身的缺点所在了——由于`INSTANCE`的初始化是在类加载时进行的，而类的加载是由`ClassLoader`来做的，所以开发者本来对于它初始化的时机就很难去准确把握：

1. 可能由于初始化的太早，造成资源的浪费
2. 如果初始化本身依赖于一些其他数据，那么也就很难保证其他数据会在它初始化之前准备好。

当然，如果所需的单例占用的资源很少，并且也不依赖于其他数据，那么这种实现方式也是很好的。

知识点：什么时候是类装载时？

前面提到了单例在**类装载时**被实例化，那究竟什么时候才是『类装载时』呢？

不严格的说，大致有这么几个条件会触发一个类被加载：

1. `new`一个对象时
2. 使用反射创建它的实例时
3. 子类被加载时，如果父类还没被加载，就先加载父类

4. jvm启动时执行的主类会首先被加载

(类在什么时候加载和初始化?)

5. 一些其他的实现方式

5.1 Effective Java 1 —— 静态内部类

《Effective Java》一书的第一版中推荐了一个中写法：

```
// Effective Java 第一版推荐写法
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton(){}
    public static final Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

这种写法非常巧妙：

- 对于内部类SingletonHolder，它是一个饿汉式的单例实现，在SingletonHolder初始化的时候会由ClassLoader来保证同步，使INSTANCE是一个真·单例。
- 同时，由于SingletonHolder是一个内部类，只在外部类的Singleton的getInstance()中被使用，所以它被加载的时机也就是在getInstance()方法第一次被调用的时候。
——它利用了ClassLoader来保证了同步，同时又能让开发者控制类加载的时机。从内部看是一个饿汉式的单例，但是从外部看来，又的确是懒汉式的实现。

简直是神乎其技。

5.2 Effective Java 2 —— 枚举

你以为到这就算完了？不，并没有，因为厉害的大神又发现了其他的方法。

《Effective Java》的作者在这本书的第二版又推荐了另外一种方法，来直接看代码：

```
// Effective Java 第二版推荐写法
public enum SingleInstance {
    INSTANCE;
    public void fun1() {
        // do something
    }
}

// 使用
SingleInstance.INSTANCE.fun1();
```

看到了么？这是一个枚举类型……连class都不用了，极简。

由于创建枚举实例的过程是线程安全的，所以这种写法也没有同步的问题。

作者对这个方法的评价：

这种写法在功能上与共有域方法相近，但是它更简洁，无偿地提供了序列化机制，绝对防止对此实例化，即使是在面对复杂的序列化或者反射攻击的时候。虽然这中方法还没有广泛采用，但是单元素的枚举类型已经成为实现Singleton的最佳方法。

枚举单例这种方法问世一些，许多分析文章都称它是实现单例的最完美方法——写法超级简单，而且又能解决大部分的问题。

不过我个人认为这种方法虽然很优秀，但是它仍然不是完美的一一比如，在需要继承的场景，它就不适用了。

6. 总结

OK，看到这里，你还会觉得单例模式是最简单的设计模式了么？再回头看一下你之前代码中的单例实现，觉得是无懈可击的么？

可能我们在实际的开发中，对单例的实现并没有那么严格的要求。比如，我如果能保证所有的getInstance都是在一个线程的话，那其实第一种最简单的教科书方式就够用了。再比如，有时候，我的单例变成了多例也可能对程序没什么太大影响……

但是，如果我们能了解更多其中的细节，那么如果哪天程序出了些问题，我们起码能多一个排查问题的点。早点解决问题，就能早点回家吃饭……:-D

—— 还有，完美的方案是不存在，任何方式都会有一个『度』的问题。 比如，你觉得代码已经无懈可击了，但是因为你用的是JAVA语言，可能ClassLoader有些BUG啊……你的代码谁运行在JVM上的，可能JVM本身有BUG啊……你的代码运行在手机上，可能手机系统有问题啊……你生活在这个宇宙里，可能宇宙本身有些BUG啊……o(╯□╰)o

所以，尽力做到能做到的最好就行了。

—— 感谢你花费了不少时间看到这里，但愿你没有觉得虚度。

【END】

推荐阅读

1. [我采访了一位 Pornhub 工程师，聊了这些纯纯的话题](#)
2. [常见排序算法总结 - Java 实现](#)
3. [Java：如何更优雅的处理空值？](#)
4. [MySQL：数据库优化，可以看看这篇文章](#)
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！

拿去吧！Java后端学习路线

三太子敖丙 Java后端 3月3日



作 者 | 三太子敖丙

公众号 | 三太子敖丙

前言

出自公众号：三太子敖丙

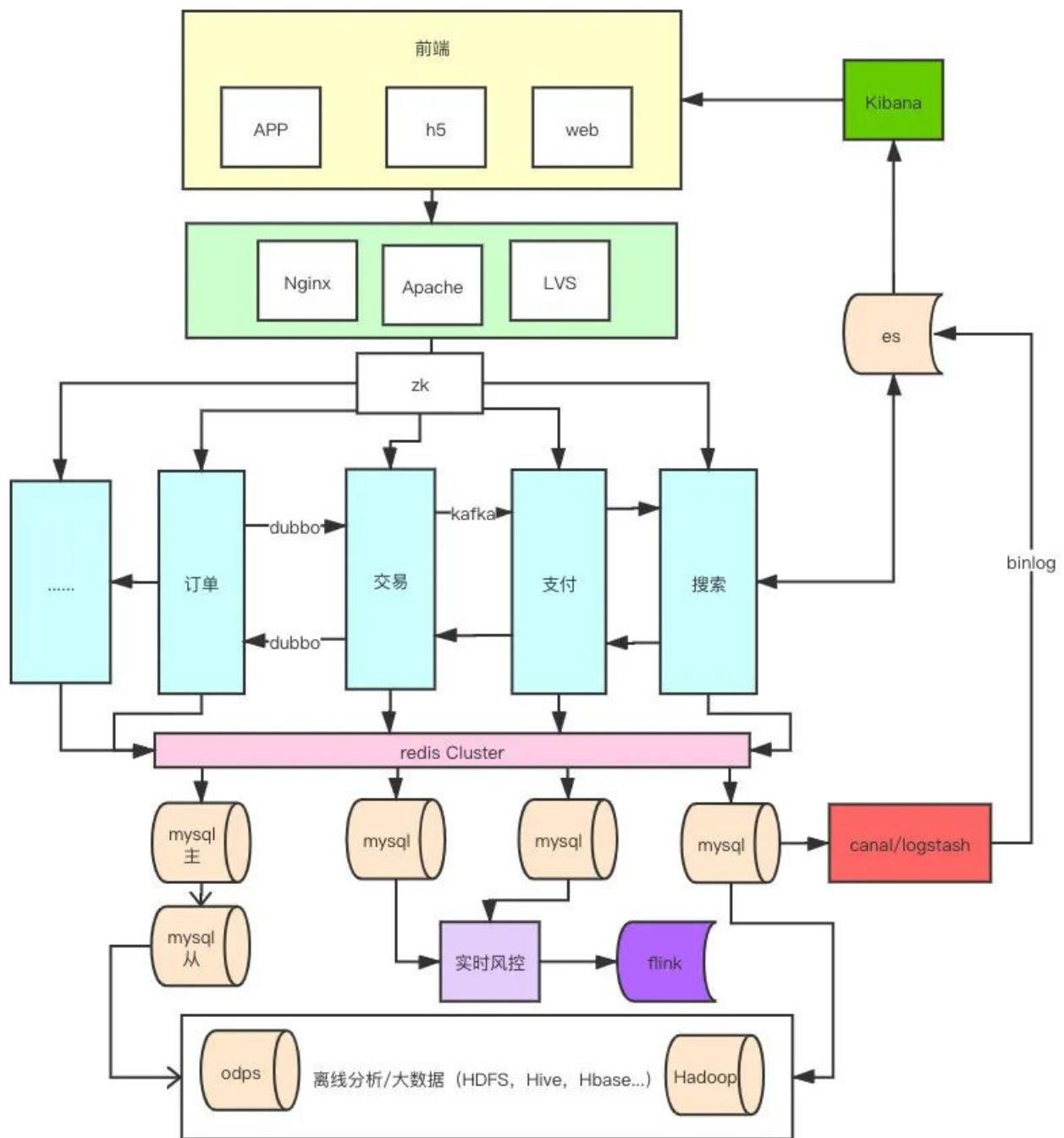
这期我想写很久了，但是因为时间的原因一直拖到了现在，我以为一两天就写完了，结果从构思到整理资料，再到写出来用了差不多一周的时间吧。

你们也知道丙丙一直都是创作鬼才来的，所以我肯定不会一本正经的写，我想了好几个切入点，最后决定用一个**完整的电商系统**作为切入点，带着大家看看，我们需要学些啥，我甚至还收集配套视频和资料，**暖男**石锤啊，这期是呕心沥血之作，**不要白嫖**了。

正文

在写这个文章之前，我花了点时间，自己臆想了一个电商系统，基本上算是麻雀虽小五脏俱全，我今天就用它开刀，一步步剖析，我会讲一下我们可能会接触的技术栈可能不全，但是够用，最后给个学习路线。

Tip：请多欣赏一会，每个点看一下，看看什么地方是你接触过的，什么技术栈是你不太熟悉的，我觉得还是比较全的，有什么建议也可以留言给我。



不知道大家都看了一下没，现在我们就要庖丁解牛了，我从上到下依次分析。

前端

你可能会会好奇，你不是讲后端学习路线嘛，为啥还有前端的部分，我只能告诉你，**傻瓜**，肤浅。

我们可**不能闭门造车**，谁告诉你后端就不学点前端了？

前端现在很多也了解后端的技术栈的，你想我们去一个网站，最先接触的，最先看到的是啥？

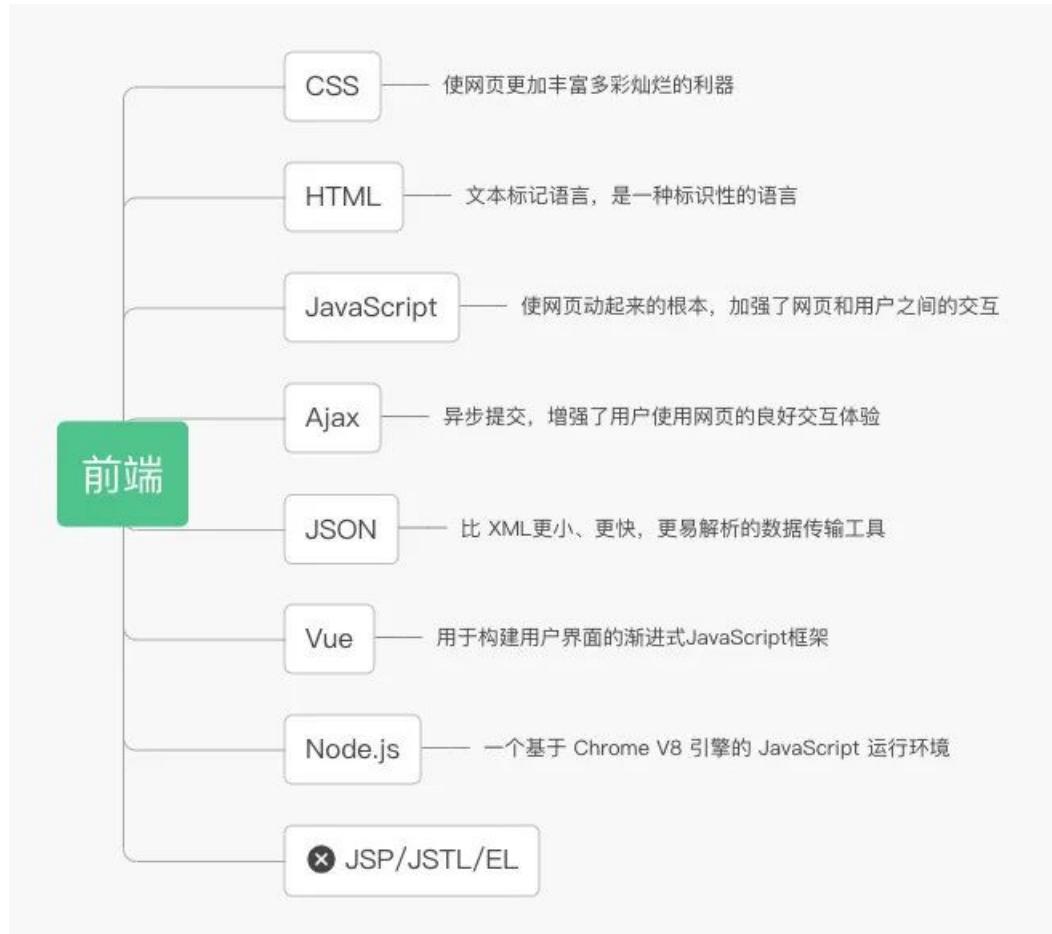
没错就是前端，在大学你是找不到专门的前端同学，去做系统肯定也要自己顶一下前端的，那我觉得最基本的技术栈得熟悉和了解吧，丙丙现在也是偶尔会开发一下我们的管理系统主要是**VUE**和**React**。

在这里我列举了我目前觉得比较简单和我们后端可以了解的技术栈，都是比较基础的。

作为一名后端了解部分前端知识还是很有必要的，在以后开发的时候，公司有前端那能帮助你前后端联调更顺畅，如果没前端你自己也能顶一下简单的页面。

HTML、CSS、JS、Ajax我觉得是必须掌握的点，看着简单其实深究或者去操作的话还是有很多东西的，其他作为扩展有兴趣可以了解，反正入门简单，只是精通很难很难。

在这一层不光有这些还有**Http协议**和**Servlet**, **request**、**response**、**cookie**、**session**这些也会伴随你整个技术生涯，理解他们对后面的你肯定有不少好处。



Tip：我这里最后删除了**JSP**相关的技术，我个人觉得没必要学了，很多公司除了老项目之外，新项目都不会使用那些技术了。

前端在我看来比后端难，技术迭代比较快，知识好像也没特定的体系，所以面试大厂的前端很多朋友都说难，不是技术多难，而是知识多且复杂，找不到一个完整的体系，相比之下后端明朗很多，我后面就开始讲后端了。

网关层：

互联网发展到现在，涌现了很多互联网公司，技术更新迭代了很多个版本，从早期的单机时代，到现在超大规模的互联网时代，几亿人参与的春运，几千亿成交规模的双十一，无数互联网前辈的造就了现在互联网的辉煌。

微服务，分布式，负载均衡等我们经常提到的这些名词都是这些技术在场景背后支撑。

单机顶不住，我们就多找点服务器，但是怎么将流量均匀的打到这些服务器上呢？

负载均衡，LVS

我们机器都是IP访问的，那怎么通过我们申请的域名去请求到服务器呢？

DNS

大家刷的抖音，B站，快手等等视频服务商，是怎么保证同时为全国的用户提供快速的体验？

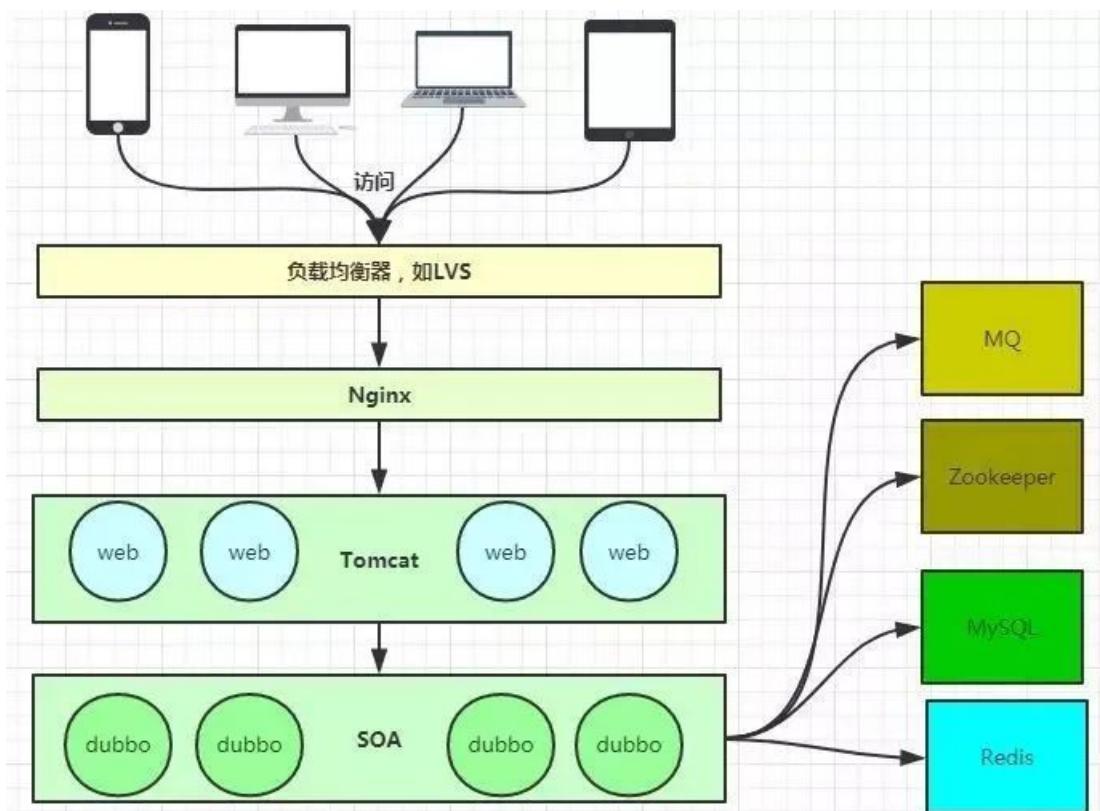
CDN

我们这么多系统和服务，还有这么多中间件的调度怎么去管理调度等等？

zk

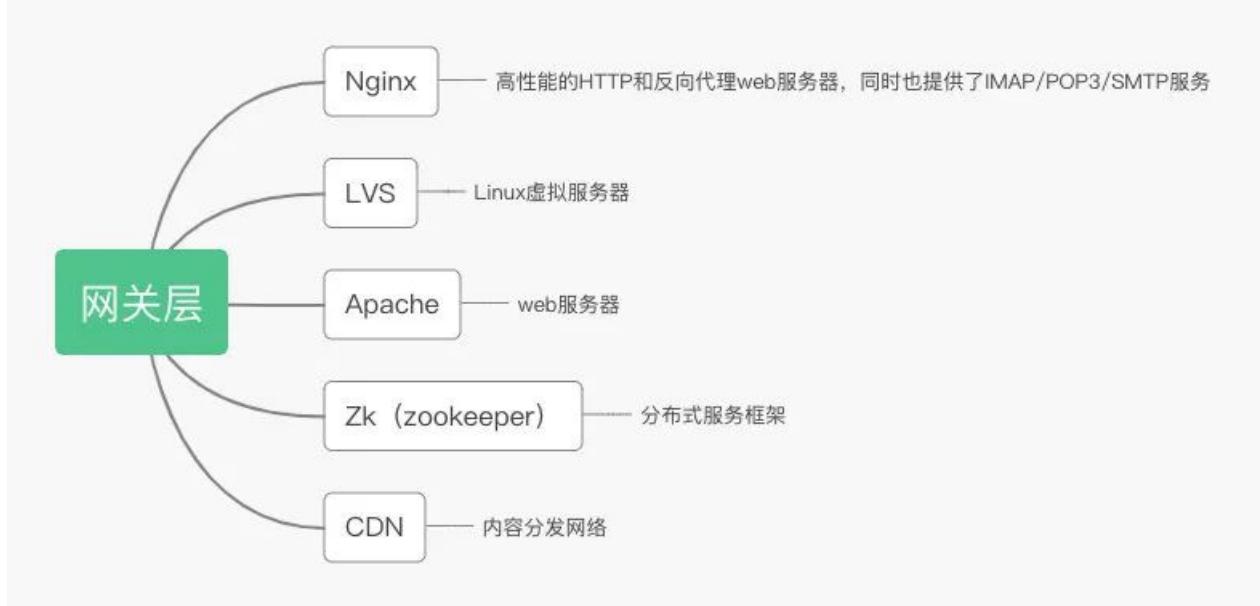
这么多的服务器，怎么对外统一访问呢，就可能需要知道反向代理的服务器。

Nginx



这一层做了**反向负载、服务路由、服务治理、流量管理、安全隔离、服务容错**等等都做了，大家公司的**内外网隔离**也是这一层做的。

我之前还接触过一些比较有意思的项目，所有对外的接口都是加密的，几十个服务会经过网关解密，找到真的路由再去请求。



这一层的知识点其实也不少，你往后面学会发现**分布式事务**, **分布式锁**, 还有很多中间件都离不开**zk**这一层，我们继续往下看。

服务层：

这一层有点东西了，算是整个框架的核心，如果你跟我帅丙一样以后都是从事后端开发的话，我们基本上整个技术生涯，大部分时间都在跟这一层的技术栈打交道了，各种琳琅满目的中间件，计算机基础知识，Linux操作，算法数据结构，架构框架，研发工具等等。

我想在看这个文章的各位，计算机基础肯定都是学过的吧，如果大学的时候没好好学，我觉得还是有必要再看看的。

为什么我们网页能保证安全可靠的传输，你可能会了解到**HTTP**, **TCP协议**，什么三次握手，四次挥手。

还有**进程**、**线程**、**协程**, **什么内存屏障**, **指令乱序**, **分支预测**, **CPU亲和性**等等，在之后的编程生涯，如果你能掌握这些东西，会让你在遇到很多问题的时候瞬间get到点，而不是像个无头苍蝇一样乱撞（然而丙丙还做得不够）。

了解这些计算机知识后，你就需要接触编程语言了，大学的**C语言**基础会让你学什么语言入门都会快点，我选择了面向对象的**JAVA**，但是也不知道为啥现在还没对象。

JAVA的基础也一样重要，**面向对象**（包括类、对象、方法、继承、封装、抽象、多态、消息解析等），常见API，数据结构，**集合框架**，**设计模式**（包括创建型、结构型、行为型），**多线程和并发**, **I/O流**, **Stream**, **网络编程**你都需要了解。

代码会写了，你就要开始学习一些能帮助你把系统变得更加规范的框架，SSM可以会让你的开发更加便捷，结构层次更加分明。

写代码的时候你会发现你大学用的**Eclipse**在公司看不到了，你跟大家一样去用了**IDEA**，第一天这是什么玩意，一周后，真香，但是这玩意收费有点贵，那免费的**VSCode**真的就是不错的选择了。

代码写的时候你会接触代码的仓库管理工具**maven**、**Gradle**，提交代码的时候会去写项目版本管理工具**Git**。

代码提交之后，发布之后你会发现很多东西需要自己去服务器亲自排查，那**Linux**的知识点就可以在里面灵活运用了，查看进程，查看文件，各种**Vim**操作等等。

系统的优化很多地方没优化的空间了，你可能会尝试从**算法**，或者优化**数据结构**去优化，你看到了HashMap的源码，想去了解红黑树，然后在算法网上看到了二叉树搜索树和各种常见的算法问题，刷多了，你也能总结出精华所在，什么**贪心，分治，动态规划**等。

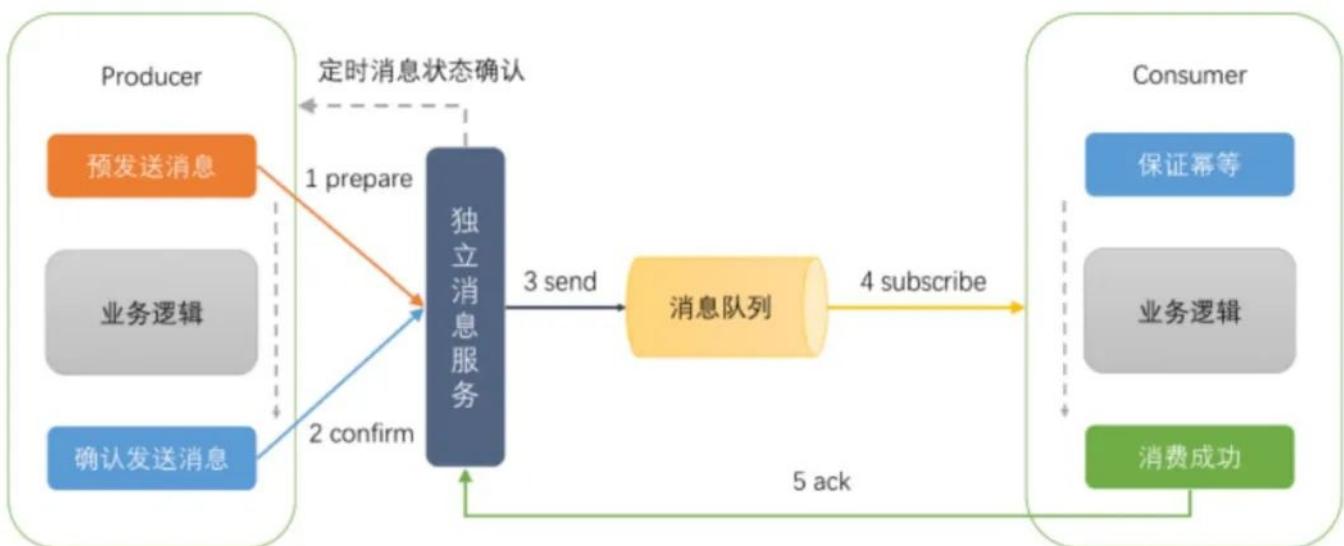
这么多个服务，你发现**HTTP**请求已经开始有点不满足你的需求了，你想开发更便捷，像访问本地服务一样访问远程服务，所以我们去了解了**Dubbo, Spring cloud**。

了解Dubbo的过程中，你发现了RPC的精华所在，所以你去接触到了高性能的**NIO**框架，**Netty**。

代码写好了，服务也能通信了，但是你发现你的代码链路好长，都耦合在一起了，所以你接触了**消息队列**，这种异步的处理方式，真香。

他还可以帮你在突发流量的时候用队列做缓冲，但是你发现分布式的情况，事务就不好管理了，你就了解到分布式事务，什么**两段式, 三段式, TCC, XA, 阿里云的全局事务服务GTS**等等。

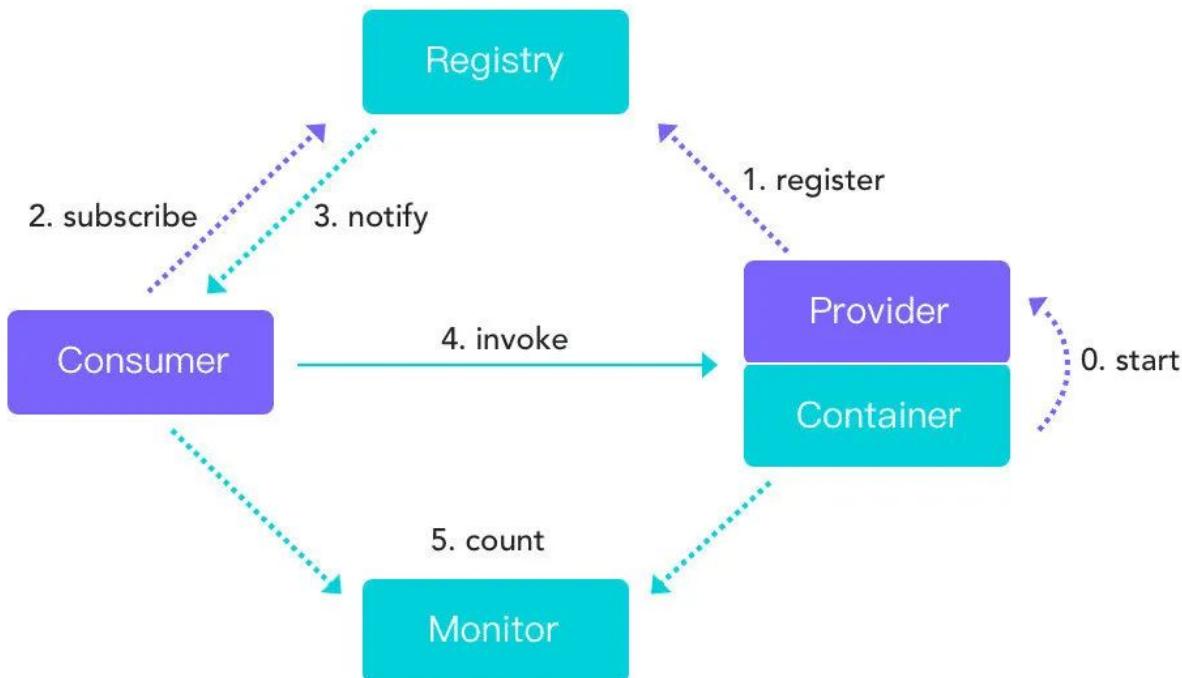
分布式事务的时候你会想去了解**RocketMQ**，因为他自带了分布式事务的解决方案，大数据的场景你又看到了**Kafka**。



我上面提到过**zk**，像**Dubbo, Kafka**等中间件都是用它做注册中心的，所以很多技术栈最后都组成了一个知识体系，你先了解了体系中的每一员，你才能把它们联系起来。

Dubbo Architecture

..... init async → sync



服务的交互都从进程内通信变成了远程通信，所以性能必然会受到一些影响。

此外由于很多不确定性的因素，例如网络拥塞、Server 端服务器宕机、挖掘机铲断机房光纤等等，需要许多额外的功能和措施才能保证微服务流畅稳定的工作。

Spring Cloud 中就有**Hystrix** 熔断器、**Ribbon**客户端负载均衡器、**Eureka**注册中心等等都是用来解决这些问题的微服务组件。

你感觉学习得差不多了，你发现各大论坛博客出现了一些前沿技术，比如容器化，你可能就会去了解容器化的知识，像**Docker, Kubernetes (K8s)**等。

微服务之所以能够快速发展，很重要的一个原因就是：容器化技术的发展和容器管理系统的成熟。



这一层的东西呢其实远远不止这些的，我不过多赘述，写多了像个劝退师一样，但是大家也不用慌，大部分的技术都是慢慢接触了，工作中慢慢去了解，去深入的。

好啦我们继续沿着图往下看，那再往下是啥呢？

数据层：

数据库可能是整个系统中最值钱的部分了，在我码文字的前一天，刚好发生了微盟程序员删库跑路的操作，删库跑路其实是我们在网上最常用的笑话，没想到还是照进了现实。



数据库删了肯定要跑路啊

这里也提一点点吧，36小时的故障，其实在互联网公司应该是个笑话了吧，权限控制没做好类似`rm -rf`、`fdisk`、`drop`等等这样的高危命令是可以实时拦截掉的，**备份，全量备份，增量备份，延迟备份，异地容灾**全部都考虑一下应该也不至于这样，一家上市公司还是有点点不应该。

数据库事务特性

原子性
(Atomicity)

一致性
(Consistency)

隔离性
(Isolation)

持久性
(Durability)

数据库基本的**事务隔离级别**, **索引**, **SQL**, **主被同步**, **读写分离**等都可能是你学的时候要了解到的。

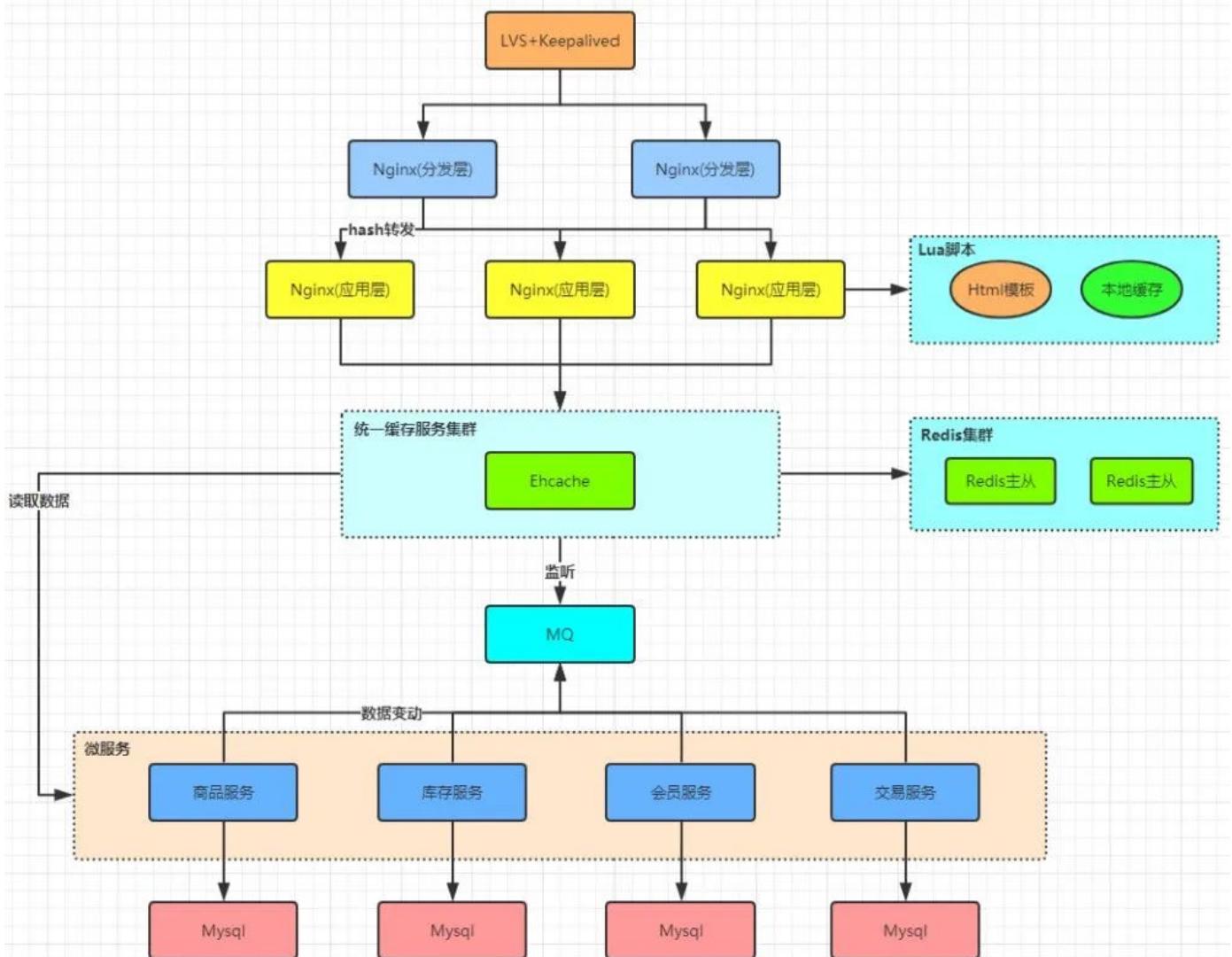
上面我们提到了安全, 不要把鸡蛋放一个篮子的道理大家应该都知道, 那**分库**的意义就很明显了, 然后你会发现时间久了表的数据大了, 就会想到去接触分表, 什么**TDDL**、**Sharding-JDBC**、**DRDS**这些插件都会接触到。

你发现流量大的时候, 或者热点数据打到数据库还是有点顶不住, 压力太大了, 那非关系型数据库就进场了, **Redis**当然是首选, 但是**MongoDB**、**memcache**也有各自的应用场景。

Redis使用后, 真香, 真快, 但是你会开始担心最开始提到的安全问题, 这玩意快是因为在内存中操作, 那断点了数据丢了怎么办? 你就开始阅读官方文档, 了解**RDB**, **AOF**这些持久化机制, 线上用的时候还会遇到**缓存雪崩击穿**、**穿透**等等问题。

单机不满足你就用了, 他的集群模式, 用了集群可能也担心集群的健康状态, 所以就得去了解**哨兵**, 他的**主从同步**, 时间久了Key多了, 就得了解**内存淘汰机制**……

他的大容量存储有问题, 你可能需要去了解**Pika**……



其实远远没完，每个的点我都点到为止，但是其实要深究每个点都要学很久，我们接着往下看。

实时/离线/大数据

等你把几种关系型非关系型数据库的知识点，整理清楚后，你会发现数据还是大啊，而且数据的场景越来越多样化了，那大数据的各种中间件你就得了解了。

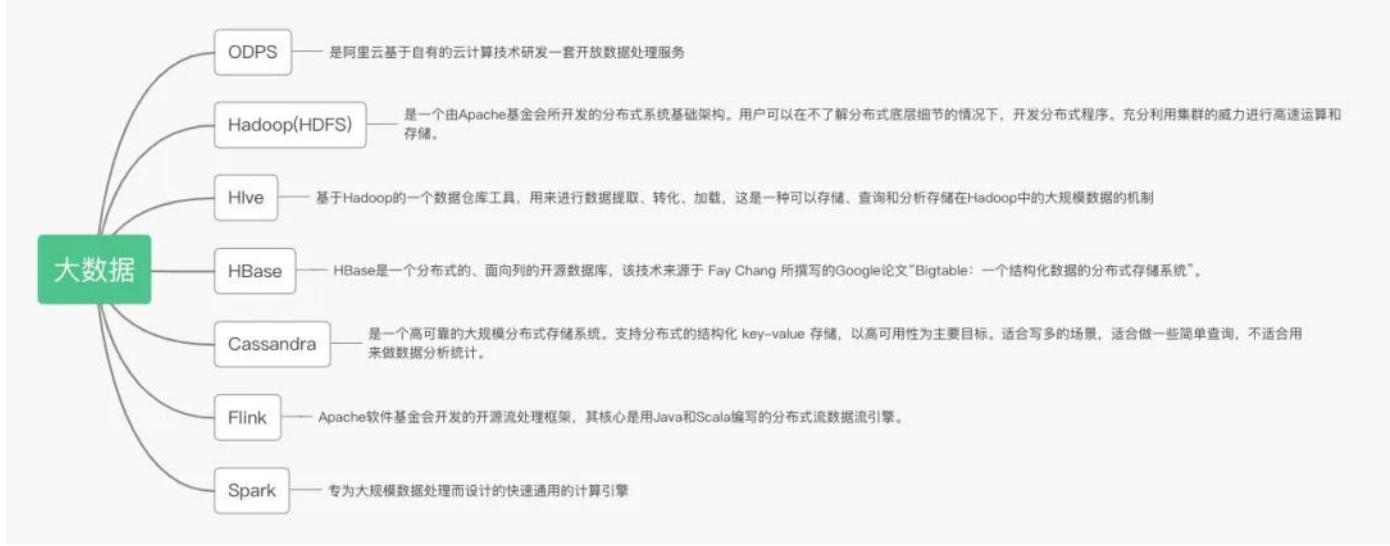
你会发现很多场景，不需要实时的数据，比如你查你的支付宝去年的，上个月的账单，这些都是不会变化的数据，没必要实时，那你可能会接触像 **ODPS** 这样的中间件去做数据的离线分析。

然后你可能会接触 Hadoop 系列相关的东西，比如于 **Hadoop (HDFS)** 的一个数据仓库工具 **Hive**，是建立在 **Hadoop** 文件系统之上的分布式面向列的数据库 **HBase**。

写多的场景，适合做一些简单查询，用他们又有点大材小用，那**Cassandra**就再合适不过了。

离线的数据分析没办法满足一些实时的常见，类似风控，那**Flink**你也得略知一二，他的窗口思想还是很有意思。

数据接触完了，计算引擎**Spark**你是不是也不能放过……



搜索引擎：

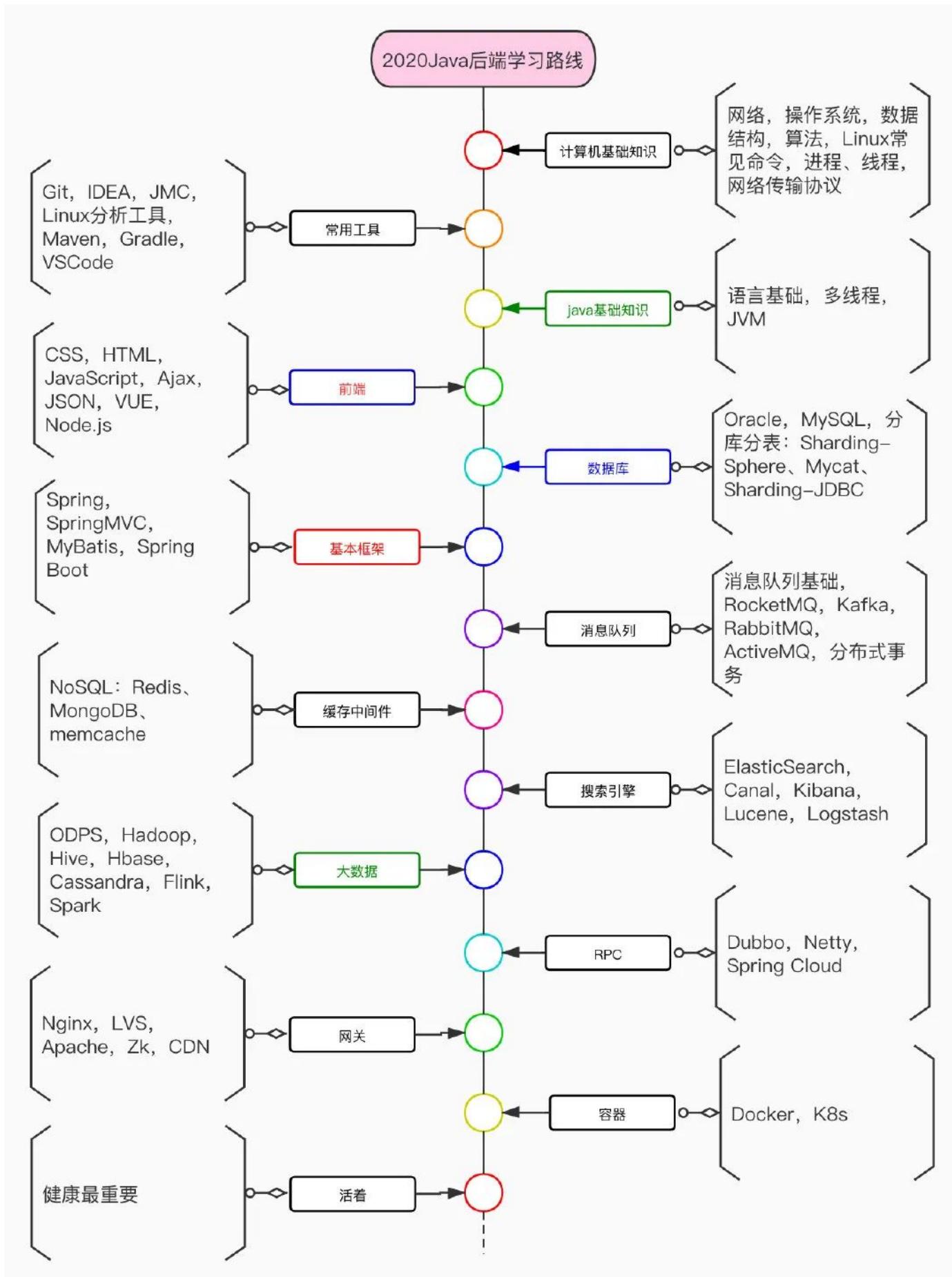
传统关系型数据库和NoSQL非关系型数据都没办法解决一些问题，比如我们在百度，淘宝搜索东西的时候，往往都是几个关键字在一起一起搜索东西的，在数据库除非把几次的结果做交集，不然很难去实现。

那全文检索引擎就诞生了，解决了搜索的问题，你得思考怎么把数据库的东西实时同步到**ES**中去，那你可能会思考到**logstash**去定时跑脚本同步，又或者去接触伪装成一台**MySQL**从服务的**Canal**，他会去订阅MySQL主服务的**binlog**，然后自己解析了去操作Es中的数据。

这些都搞定了，那可视化的后台查询又怎么解决呢？**Kibana**，他他是一个可视化的平台，甚至对Es集群的健康管理都做了可视化，很多公司的日志查询系统都是用它做的。



看了这么久你是不是发现，帅丙只是一直在介绍每个层级的技术栈，并没说到具体的一个路线，那是因为我想让大家先有个认知或者说是扫盲吧，我一样用脑图的方式汇总一下吧，如果图片被平台二压了，可以去公众号回复【[路线](#)】。



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. 那些神一样的程序员
2. 16000 字 Redis 面试知识点总结，建议收藏！
3. 不会 IntelliJ IDEA 项目配置？
4. 高频使用的 Git 命令



[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！

挑战 10 道超难 Java 面试题

Java后端 2019-09-04

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

译者:Yujiaao

来源:segmentfault.com/a/1190000019962661

原文:<http://t.cn/AiH7NCW1>

这是我收集的10个最棘手的Java面试问题列表。这些问题主要来自 Java 核心部分 ,不涉及 Java EE 相关问题。你可能知道这些棘手的 Java 问题的答案,或者觉得这些不足以挑战你的 Java 知识,但这些问题都是容易在各种 Java 面试中被问到的,而且包括我的朋友和同事在内的许多程序员都觉得很难回答。

1.为什么等待和通知是在 Object 类而不是 Thread 中声明的?

一个棘手的 Java 问题,如果 Java 编程语言不是你设计的,你怎么能回答这个问题呢。Java 编程的常识和深入了解有助于回答这种棘手的 Java 核心方面的面试问题。

为什么 wait, notify 和 notifyAll 是在 Object 类中定义的而不是在 Thread 类中定义

这是有名的 Java 面试问题,招2~4年经验的到高级 Java 开发人员面试都可能碰到。

这个问题的好在它能反映了面试者对等待通知机制的了解,以及他对此主题的理解是否明确。就像为什么 Java 中不支持多继承或者为什么 String 在 Java 中是 final 的问题一样,这个问题也可能有多个答案。

为什么在 Object 类中定义 wait 和 notify 方法,每个人都能说出一些理由。从我的面试经验来看, wait 和 notify 仍然是大多数Java 程序员最困惑的,特别是2到3年的开发人员,如果他们要求使用 wait 和 notify,他们会很困惑。因此,如果你去参加 Java 面试,请确保对 wait 和 notify 机制有充分的了解,并且可以轻松地使用 wait 来编写代码,并通过生产者-消费者问题或实现阻塞队列等了解通知的机制。

为什么等待和通知需要从同步块或方法中调用,以及 Java 中的 wait, sleep 和 yield 方法之间的差异,如果你还没有读过,你会觉得有趣。为何 wait, notify 和 notifyAll 属于 Object 类?为什么它们不应该在 Thread 类中?以下是我认为有意义的一些想法:

1) wait 和 notify 不仅仅是普通方法或同步工具,更重要的是它们是 Java 中两个线程之间的通信机制对语言设计者而言,如果不能通过 Java 关键字(例如 synchronized)实现通信此机制,同时又要确保这个机制对每个对象可用,那么 Object 类则是的正确声明位置。记住同步和等待通知是两个不同的领域,不要把它们看成是相同的或相关的。同步是提供互斥并确保 Java 类的线程安全,而 wait 和 notify 是两个线程之间的通信机制。

2) 每个对象都可上锁,这是在 Object 类而不是 Thread 类中声明 wait 和 notify 的另一个原因。

3) 在 Java 中为了进入代码的临界区,线程需要锁定并等待锁定他们不知道哪些线程持有锁,而只是知道锁被某个线程持有,并且他们应该等待取得锁,而不是去了解哪个线程在同步块内,并请求它们释放锁定。

4) Java 是基于 Hoare 的监视器的思想在 Java 中,所有对象都有一个监视器。

线程在监视器上等待,为执行等待,我们需要2个参数:

- 一个线程
- 一个监视器(任何对象)

在 Java 设计中,线程不能被指定,它总是运行当前代码的线程。但是,我们可以指定监视器(这是我们称之为等待的对象)。这是一个很好的设计,因为如果我们可以让任何其他线程在所需的监视器上等待,这将导致“入侵”,导致在设计并发程序时会遇到困难。请记住,在 Java 中,所有在另一个线程的执行中侵入的操作都被弃用了(例如 stop 方法)。

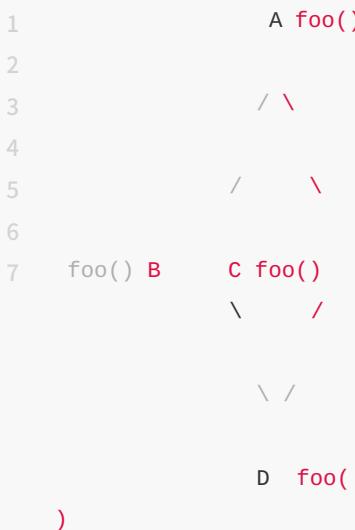
2.为什么Java中不支持多重继承?

我发现这个 Java 核心问题很难回答,因为你的答案可能不会让面试官满意,在大多数情况下,面试官正在寻找答案中的关键点,如果你提到这些关键点,面试官会很高兴。在 Java 中回答这种棘手问题的关键是准备好相关主题,以应对后续的各种可能的问题。

这是非常经典的问题,与为什么 String 在 Java 中是不可变的很类似;这两个问题之间的相似之处在于它们主要是由 Java 创作者的设计决策使然。

为什么Java不支持多重继承,可以考虑以下两点:

1)第一个原因是围绕钻石形继承问题产生的歧义,考虑一个类 A 有 foo() 方法,然后 B 和 C 派生自 A,并且有自己的 foo() 实现,现在 D 类使用多个继承派生自 B 和 C,如果我们只引用 foo(),编译器将无法决定它应该调用哪个 foo()。这也称为 Diamond 问题,因为这个继承方案的结构类似于菱形,见下图:



与 C++ 不同, Java 不支持运算符重载。Java 不能为程序员提供自由的标准算术运算符重载, 例如 +, -, * 和 / 等。如果你以前用过 C++, 那么 Java 与 C++ 相比少了很多功能, 例如 Java 不支持多重继承, Java 中没有指针, Java 中没有引用传递。另一个类似的问题是关于 Java 通过引用传递, 这主要表现为 Java 是通过值还是引用传参。虽然我不知道背后的真正原因, 但我认为以下说法有些道理, 为什么 Java 不支持运算符重载。

1)简单性和清晰性。清晰性是 Java 设计者的目标之一。设计者不是只想复制语言, 而是希望拥有一种清晰, 真正面向对象的语言。添加运算符重载比没有它肯定会使设计更复杂, 并且它可能导致更复杂的编译器, 或减慢 JVM, 因为它需要做额外的工作来识别运算符的实际含义, 并减少优化的机会, 以保证 Java 中运算符的行为。

2)避免编程错误。Java 不允许用户定义的运算符重载, 因为如果允许程序员进行运算符重载, 将为同一运算符赋予多种含义, 这将使任何开发人员的学习曲线变得陡峭, 事情变得更加混乱。据观察, 当语言支持运算符重载时, 编程错误会增加, 从而增加了开发和交付时间。由于 Java 和 JVM 已经承担了大多数开发人员的责任, 如在通过提供垃圾收集器进行内存管理时, 因为这个功能增加污染代码的机会, 成为编程错误之源, 因此没有多大意义。

3)JVM复杂性。从 JVM 的角度来看, 支持运算符重载使问题变得更加困难。通过更直观, 更干净的方式使用方法重载也能实现同样的事情, 因此不支持 Java 中的运算符重载是有意义的。与相对简单的 JVM 相比, 复杂的 JVM 可能导致 JVM 更慢, 并为保证在 Java 中运算符行为的确定性从而减少了优化代码的机会。

4)让开发工具处理更容易。这是在 Java 中不支持运算符重载的另一个好处。省略运算符重载使语言更容易处理, 这反过来又更容易开发处理语言的工具, 例如 IDE 或重构工具。Java 中的重构工具远胜于 C++。

4.为什么 String 在 Java 中是不可变的?

我最喜欢的 Java 面试问题, 很棘手, 但同时也非常有用。一些面试者也常问这个问题, 为什么 String 在 Java 中是 final 的。

字符串在 Java 中是不可变的, 因为 String 对象缓存在 String 池中。由于缓存的字符串在多个客户之间共享, 因此始终存在风险, 其中一个客户的操作会影响所有其他客户。例如, 如果一段代码将 String “Test”的值更改为 “TEST”, 则所有其他客户也将看到该值。由于 String 对象的缓存性能是很重要的一方面, 因此通过使 String 类不可变来避免这种风险。

同时, String 是 final 的, 因此没有人可以通过扩展和覆盖行为来破坏 String 类的不变性、缓存、散列值的计算等。String 类不可变的另一个原因可能是由于 HashMap。

由于把字符串作为 HashMap 键很受欢迎。对于键值来说, 重要的是它们是不可变的, 以便用它们检索存储在 HashMap 中的值对象。由于 HashMap 的工作原理是散列, 因此需要具有相同的值才能正常运行。如果在插入后修改了 String 的内容, 可变的 String 将在插入和检索时生成两个不同的哈希码, 可能会丢失 Map 中的值对象。

如果你是印度板球迷, 你可能能够与我的下一句话联系起来。字符串是 Java 的 VVS Laxman, 即非常特殊的类。我还没有看到一个没有使用 String 编写的 Java 程序。这就是为什么对 String 的充分理解对于 Java 开发人员来说非常重要。

String 作为数据类型, 传输对象和中间人角色的重要性和流行性也使这个问题在 Java 面试中很常见。

为什么 String 在 Java 中是不可变的是 Java 中最常被问到的字符串访问问题之一, 它首先讨论了什么是 String, Java 中的 String 如何与 C 和 C++ 中的 String 不同, 然后转向在 Java 中什么是不可变对象, 不可变对象有什么好处, 为什么要使用它们以及应该使用哪些场景。

这个问题有时也会问: “**为什么 String 在 Java 中是 final 的?**”在类似的说明中, 如果你正在准备 Java 面试, 我建议你看《Java 程序员面试宝典(第4版)》, 这是高级和中级 Java 程序员的优秀资源。它包含来自所有重要 Java 主题的问题, 包括多线程, 集合, GC, JVM 内部以及 Spring 和 Hibernate 框架等。

正如我所说, 这个问题可能有很多可能的答案, 而 String 类的唯一设计者可以放心地回答它。我在 Joshua Bloch 的

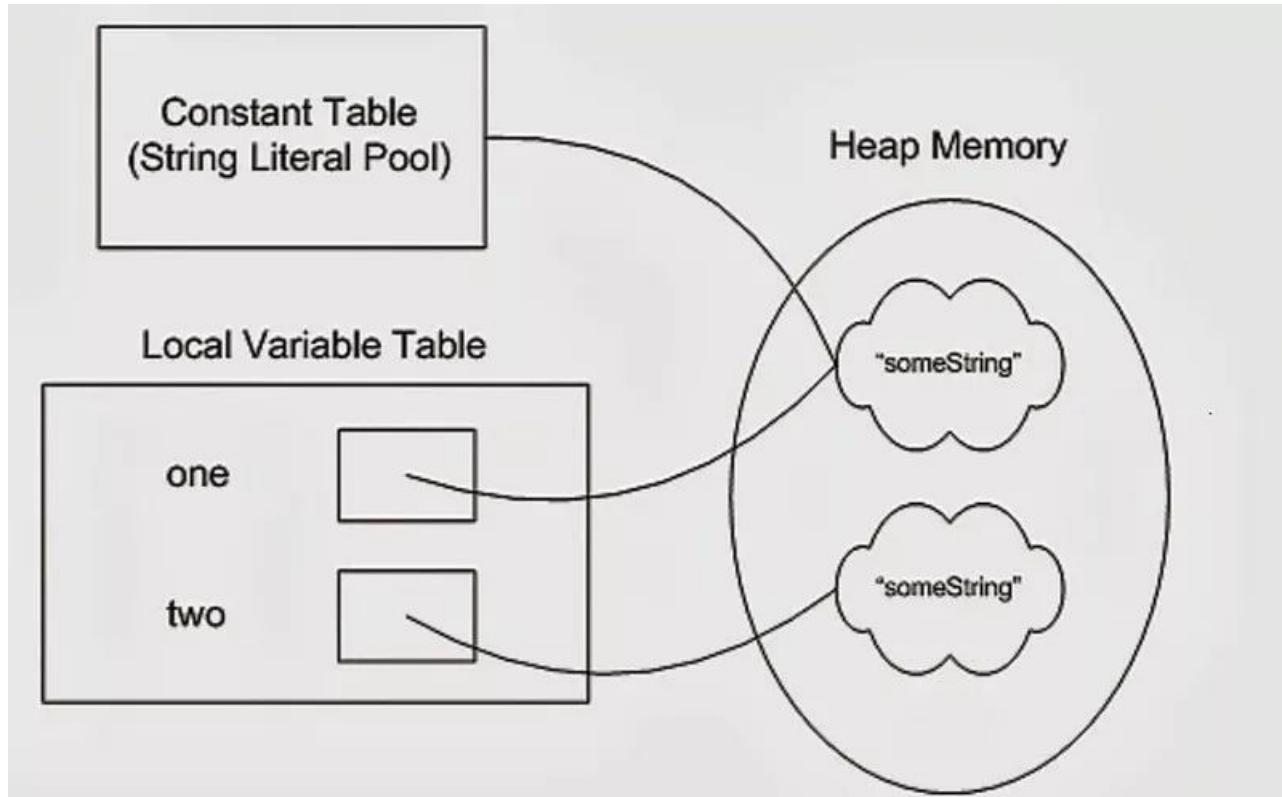
Effective Java 书中期待一些线索,但他也没有提到它。我认为以下几点解释了为什么 String 类在 Java 中是不可变的或 final 的:

1) 想象字符串池没有使字符串不可变, 它根本不可能, 因为在字符串池的情况下, 一个字符串对象/文字, 例如 “Test” 已被许多参考变量引用, 因此如果其中任何一个更改了值, 其他参数将自动受到影响, 即假设

```
1 String A="Test";
2 String B="Test";
```

现在字符串 B 调用 "Test".toUpperCase(), 将同一个对象改为“TEST”, 所以 A 也是“TEST”, 这不是期望的结果。

下图显示了如何在堆内存和字符串池中创建字符串。



2) 字符串已被广泛用作许多 Java 类的参数, 例如, 为了打开网络连接, 你可以将主机名和端口号作为字符串传递, 你可以将数据库 URL 作为字符串传递, 以打开数据库连接, 你可以通过将文件名作为参数传递给 File I/O 类来打开 Java 中的任何文件。如果 String 不是不可变的, 这将导致严重的安全威胁, 我的意思是有人可以访问他有权限的任何文件, 然后可以故意或意外地更改文件名并获得对该文件的访问权限。由于不变性, 你无需担心这种威胁。这个原因也说明了, 为什么 String 在 Java 中是最终的, 通过使 java.lang.String final, Java 设计者确保没有人覆盖 String 类的任何行为。

3) 由于 String 是不可变的, 它可以安全地共享许多线程, 这对于多线程编程非常重要. 并且避免了 Java 中的同步问题, 不变性也使得 String 实例在 Java 中是线程安全的, 这意味着你不需要从外部同步 String 操作。关于 String 的另一个要点是由截取字符串 SubString 引起的内存泄漏, 这不是与线程相关的问题, 但也是需要注意的。

4) 为什么 String 在 Java 中是不可变的另一个原因是允许 String 缓存其哈希码, Java 中的不可变 String 缓存其哈希码, 并且不会在每次调用 String 的 hashCode 方法时重新计算, 这使得它在 Java 中的 HashMap 中使用的 HashMap 键非常快。简而言之, 因为 String 是不可变的, 所以没有人可以在创建后更改其内容, 这保证了 String 的 hashCode 在多次调用时是相同的。

5) String 不可变的绝对最重要的原因是它被类加载机制使用, 因此具有深刻和基本的安全考虑。如果 String 是可变的, 加载“java.io.Writer”的请求可能已被更改为加载“mil.vogoon.DiskErasingWriter”。安全性和字符串池是使字符串不可变的主要原因。顺便说一句, 上面的理由很好回答另一个Java面试问题: “为什么String在Java中是最终的”。要想是不

可变的，你必须是最终的，这样你的子类不会破坏不变性。你怎么看？

5.为什么 char 数组比 Java 中的 String 更适合存储密码？

另一个基于 String 的棘手 Java 问题，相信我只有很少的 Java 程序员可以正确回答这个问题。这是一个真正艰难的核心 Java 面试问题，并且需要对 String 的扎实知识才能回答这个问题。

这是最近在 Java 面试中向我的一位朋友询问的问题。他正在接受技术主管职位的面试，并且有超过6年的经验。如果你还没有遇到过这种情况，那么字符数组和字符串可以用来存储文本数据，但是选择一个而不是另一个很难。但正如我的朋友所说，任何与 String 相关的问题都必须对字符串的特殊属性有一些线索，比如不变性，他用它来说服访问提问的人。在这里，我们将探讨为什么你应该使用char[]存储密码而不是String的一些原因。

字符串：

1)由于字符串在 Java 中是不可变的，如果你将密码存储为纯文本，它将在内存中可用，直到垃圾收集器清除它。并且为了可重用性，会存在 String 在字符串池中，它很可能会保留在内存中持续很长时间，从而构成安全威胁。

由于任何有权访问内存转储的人都可以以明文形式找到密码，这是另一个原因，你应该始终使用加密密码而不是纯文本。由于字符串是不可变的，所以不能更改字符串的内容，因为任何更改都会产生新的字符串，而如果你使用char[]，你就可以将所有元素设置为空白或零。因此，在字符数组中存储密码可以明显降低窃取密码的安全风险。

2)Java 本身建议使用 JPasswordField 的 getPassword() 方法，该方法返回一个 char[] 和不推荐使用的getTex() 方法，该方法以明文形式返回密码，由于安全原因。应遵循 Java 团队的建议，坚持标准而不是反对它。

3)使用 String 时，总是存在在日志文件或控制台中打印纯文本的风险，但如果使用 Array，则不会打印数组的内容而是打印其内存位置。虽然不是一个真正的原因，但仍然有道理。

```
1  String strPassword = "Unknown";
2  char [] charPassword = new char [] {'U', 'n', 'k', 'w', 'o', 'n'};
3
4  System.out.println("字符密码：" + strPassword);
5  System.out.println("字符密码：" + charPassword);
```

输出

```
1 字符串密码：Unknown
2 字符密码：[C @110b053
```

我还建议使用散列或加密的密码而不是纯文本，并在验证完成后立即从内存中清除它。因此，在Java中，用字符数组用存储密码比字符串是更好的选择。虽然仅使用char[]还不够，还你需要擦除内容才能更安全。

6.如何使用双重检查锁定在 Java 中创建线程安全的单例？

这个 Java 问题也常被问：**什么是线程安全的单例，你怎么创建它。**好吧，在Java 5之前的版本，使用双重检查锁定创建单例 Singleton 时，如果多个线程试图同时创建 Singleton 实例，则可能有多个 Singleton 实例被创建。从 Java 5 开始，使用 Enum 创建线程安全的Singleton很容易。但如果面试官坚持双重检查锁定，那么你必须为他们编写代码。记得使用 volatile 变量。

[为什么枚举单例在 Java 中更好](#)

枚举单例是使用一个实例在 Java 中实现单例模式的新方法。虽然 Java 中的单例模式存在很长时间,但枚举单例是相对较新的概念,在引入 Enum 作为关键字和功能之后,从 Java 5 开始在实践中。本文与之前关于 Singleton 的内容有些相关,其中讨论了有关 Singleton 模式的面试中的常见问题,以及 10 个 Java 枚举示例,其中我们看到了如何通用枚举可以。这篇文章是关于为什么我们应该使用 Enum 作为 Java 中的单例,它比传统的单例方法相比有什么好处等等。

Java 枚举和单例模式

Java 中的枚举单例模式是使用枚举在 Java 中实现单例模式。单例模式在 Java 中早有应用,但使用枚举类型创建单例模式时间却不长。如果感兴趣,你可以了解下构建者设计模式和装饰器设计模式。

1) 枚举单例易于书写

这是迄今为止最大的优势,如果你在 Java 5 之前一直在编写单例,你知道,即使双检查锁定,你仍可以有多个实例。虽然这个问题通过 Java 内存模型的改进已经解决了,从 Java 5 开始的 volatile 类型变量提供了保证,但是对于许多初学者来说,编写起来仍然很棘手。与同步双检查锁定相比,枚举单例实在是太简单了。如果你不相信,那就比较一下下面的传统双检查锁定单例和枚举单例的代码:

在 Java 中使用枚举的单例

这是我们通常声明枚举的单例的方式,它可能包含实例变量和实例方法,但为了简单起见,我没有使用任何实例方法,只是要注意,如果你使用的实例方法且该方法能改变对象的状态的话,则需要确保该方法的线程安全。默认情况下,创建枚举实例是线程安全的,但 Enum 上的任何其他方法是否线程安全都是程序员的责任。

```
1 /**
2 * 使用 Java 枚举的单例模式示例
3 */
4 public enum EasySingleton{
5     INSTANCE
6 ;
7 }
```

你可以通过 EasySingleton.INSTANCE 来处理它,这比在单例上调用 getInstance() 方法容易得多。

具有双检查锁定的单例示例

下面的代码是单例模式中双重检查锁定的示例,此处的 getInstance() 方法检查两次,以查看 INSTANCE 是否为空,这就是为什么它被称为双检查锁定模式,请记住,双检查锁定是代理之前 Java 5,但 Java 5 内存模型中易失变量的干扰,它应该工作完美。

```
1 /**
2 * 单例模式示例,双重锁定检查
3 */
4 public class DoubleCheckedLockingSingleton{
5     private volatile DoubleCheckedLockingSingleton INSTANCE;
6
7     private DoubleCheckedLockingSingleton(){
8 }
9
10    public DoubleCheckedLockingSingleton getInstance()
11 {
12     if(INSTANCE == null)
```

```
11     INSTANCE == null)
12     {
13         synchronized(DoubleCheckedLockingSingleton.class){
14             //double checking Singleton instance
15             if(INSTANCE == null)
16             {
17                 INSTANCE = new DoubleCheckedLockingSingleton();
18             }
19         }
20     }
21     return INSTANCE;
22 }
23 }
```

你可以调用DoubleCheckedLockingSingleton.getInstance() 来获取此单例类的访问权限。

现在,只需查看创建延迟加载的线程安全的 Singleton 所需的代码量。使用枚举单例模式,你可以在一行中具有该模式,因为创建枚举实例是线程安全的,并且由 JVM 进行。

人们可能会争辩说,有更好的方法来编写 Singleton 而不是双检查锁定方法,但每种方法都有自己的优点和缺点,就像我最喜欢在类加载时创建的静态字段 Singleton,如下面所示,但请记住,这不是一个延迟加载单例:

单例模式用静态工厂方法

这是我最喜欢的在 Java 中影响 Singleton 模式的方法之一,因为 Singleton 实例是静态的,并且最后一个变量在类首次加载到内存时初始化,因此实例的创建本质上是线程安全的。

```
1 /**
2 * 单例模式示例与静态工厂方法
3 */
4 public class Singleton{
5     //initialized during class loading
6     private static final Singleton INSTANCE = new Singleton();
7
8     //to prevent creating another instance of Singleton
9     private Singleton(){
10 }
11
12     public static Singleton getSingleton()
13 {
14     return INSTANCE;
15 }
16 }
```

你可以调用 Singleton.getSingleton() 来获取此类的访问权限。

2) 枚举单例自行处理序列化

传统单例的另一个问题是,一旦实现可序列化接口,它们就不再是 Singleton,因为 readObject() 方法总是返回一个新实例,就像 Java 中的构造函数一样。通过使用 readResolve() 方法,通过在以下示例中替换 Singeton 来避免这种情况:

```
1 //readResolve to prevent another instance of Singleton
```

```
2 private Object readResolve()
3 {
4     return INSTANCE;
}
```

如果 Singleton 类保持内部状态, 这将变得更加复杂, 因为你需要标记为 transient(不被序列化), 但使用枚举单例, 序列化由 JVM 进行。

3) 创建枚举实例是线程安全的

如第 1 点所述, 因为 Enum 实例的创建在默认情况下是线程安全的, 你无需担心是否要做双重检查锁定。

总之, 在保证序列化和线程安全的情况下, 使用两行代码枚举单例模式是在 Java 5 以后的世界中创建 Singleton 的最佳方式。你仍然可以使用其他流行的方法, 如你觉得更好, 欢迎讨论。

7. 编写 Java 程序时, 如何在 Java 中创建死锁并修复它?

经典但核心 Java 面试问题之一。

如果你没有参与过多线程并发 Java 应用程序的编码, 你可能会失败。

如何避免 Java 线程死锁?

如何避免 Java 中的死锁? 是 Java 面试的热门问题之一, 也是多线程的编程中的重口味之一, 主要在招高级程序员时容易被问到, 且有很多后续问题。尽管问题看起来非常基本, 但大多数 Java 开发人员一旦你开始深入, 就会陷入困境。

面试问题总是以“什么是死锁?”开始

当两个或多个线程在等待彼此释放所需的资源(锁定)并陷入无限等待即是死锁。它仅在多任务或多线程的情况下发生。

如何检测 Java 中的死锁?

虽然这可以有很多答案, 但我的版本是首先我会看看代码, 如果我看到一个嵌套的同步块, 或从一个同步的方法调用其他同步方法, 或试图在不同的对象上获取锁, 如果开发人员不是非常小心, 就很容易造成死锁。

另一种方法是在运行应用程序时实际锁定时找到它, 尝试采取线程转储, 在 Linux 中, 你可以通过 kill -3 命令执行此操作, 这将打印应用程序日志文件中所有线程的状态, 并且你可以看到哪个线程被锁定在哪个线程对象上。

你可以使用 fastthread.io 网站等工具分析该线程转储, 这些工具允许你上载线程转储并对其进行分析。

另一种方法是使用 jConsole 或 VisualVM, 它将显示哪些线程被锁定以及哪些对象被锁定。

如果你有兴趣了解故障排除工具和分析线程转储的过程, 我建议你看看 Uriah Levy 在多元视觉(Pluralsight)上《分析 Java 线程转储》课程。旨在详细了解 Java 线程转储, 并熟悉其他流行的高级故障排除工具。

编写一个将导致死锁的 Java 程序?

一旦你回答了前面的问题, 他们可能会要求你编写代码, 这将导致 Java 死锁。

这是我的版本之一

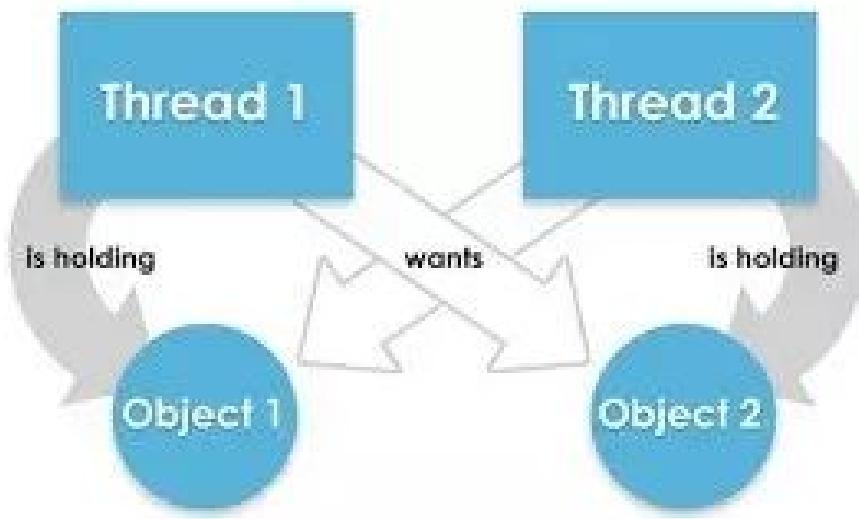
```
1 /**
2 * Java 程序通过强制循环等待来创建死锁。
```

```

3 *
4 *
5 */
6 public class DeadLockDemo {
7
8     /*
9      * 此方法请求两个锁,第一个字符串,然后整数
10     */
11
12     public void method1()
13 {
14         synchronized (String.class) {
15             System.out.println("Aquired lock on String.class object")
16         ;
17
18         synchronized (Integer.class) {
19             System.out.println("Aquired lock on Integer.class object")
20         ;
21     }
22 }
23
24
25
26     /*
27      * 此方法也请求相同的两个锁,但完全
28      * 相反的顺序,即首先整数,然后字符串。
29      * 如果一个线程持有字符串锁,则这会产生潜在的死锁
30      * 和其他持有整数锁,他们等待对方,永远。
31      */
32
33     public void method2()
34 {
35         synchronized (Integer.class) {
36             System.out.println("Aquired lock on Integer.class object")
37         ;
38
39         synchronized (String.class) {
40             System.out.println("Aquired lock on String.class object")
41         ;
42     }
43 }
44 }

```

如果 method1() 和 method2() 都由两个或多个线程调用,则存在死锁的可能性,因为如果线程 1 在执行 method1() 时在 Sting 对象上获取锁,线程 2 在执行 method2() 时在 Integer 对象上获取锁,等待彼此释放 Integer 和 String 上的锁以继续进行一步,但这永远不会发生。



此图精确演示了我们的程序，其中一个线程在一个对象上持有锁，并等待其他线程持有的其他对象锁。

你可以看到，Thread1 需要 Thread2 持有的 Object2 上的锁，而 Thread2 希望获得 Thread1 持有的 Object1 上的锁。由于没有线程愿意放弃，因此存在死锁，Java 程序被卡住。

其理念是，你应该知道使用常见并发模式的正确方法，如果你不熟悉这些模式，那么 Jose Paumard《应用于并发和多线程的常见 Java 模式》是学习的好起点。

如何避免Java中的死锁？

现在面试官来到最后一部分，在我看来，最重要的部分之一；如何修复代码中的死锁？或如何避免Java中的死锁？

如果你仔细查看了上面的代码，那么你可能已经发现死锁的真正原因不是多个线程，而是它们请求锁的方式，如果你提供有序访问，则问题将得到解决。

下面是我的修复版本，它通过避免循环等待，而避免死锁，而不需要抢占，这是需要死锁的四个条件之一。

```

1  public class DeadLockFixed {
2
3      /*
4      *
5      * 两种方法现在都以相同的顺序请求锁，首先采用整数，然后是 String。
6      * 你也可以做反向，例如，第一个字符串，然后整数，
7      * 只要两种方法都请求锁定，两者都能解决问题
8      * 顺序一致。
9      */
10     public void method1()
11     {
12         synchronized (Integer.class) {
13             System.out.println("Aquired lock on Integer.class object")
14         ;
15
16         synchronized (String.class) {
17             System.out.println("Aquired lock on String.class object")
18         ;
19         }
20     }
21 }

```

```

23     public void method2()
24     {
25         synchronized (Integer.class) {
26             System.out.println("Aquired lock on Integer.class object")
27         ;
28         synchronized (String.class) {
29             System.out.println("Aquired lock on String.class object")
30         ;
31     }
32 }

```

现在没有任何死锁,因为两种方法都按相同的顺序访问 Integer 和 String 类文本上的锁。因此,如果线程 A 在 Integer 对象上获取锁,则线程 B 不会继续,直到线程 A 释放 Integer 锁,即使线程 B 持有 String 锁,线程 A 也不会被阻止,因为现在线程 B 不会期望线程 A 释放 Integer 锁以继续。

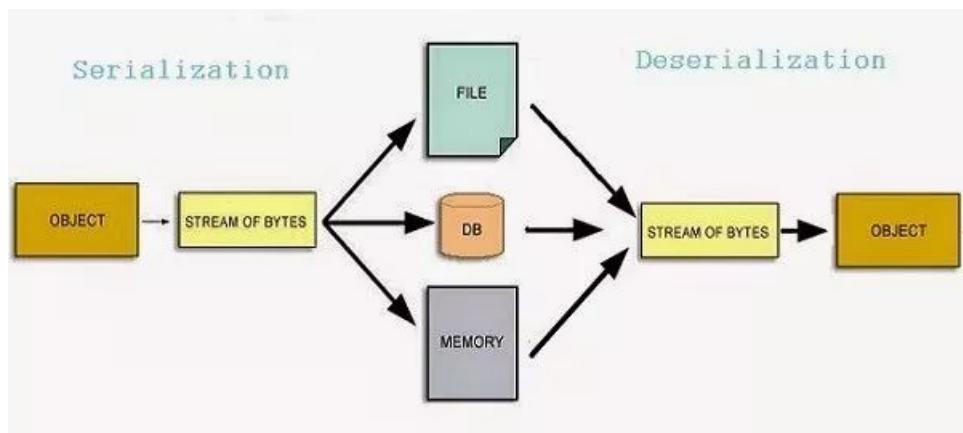
8. 如果你的Serializable类包含一个不可序列化的成员,会发生什么?你是如何解决的?

任何序列化该类的尝试都会因NotSerializableException而失败,但这可以通过在 Java 中为 static 设置瞬态 (transient) 变量来轻松解决。

Java 序列化相关的常见问题

Java 序列化是一个重要概念,但它很少用作持久性解决方案,开发人员大多忽略了 Java 序列化 API。根据我的经验,Java 序列化在任何 Java 核心内容面试中都是一个相当重要的话题,在几乎所有的网面试中,我都遇到过一两个 Java 序列化问题,我看过去一次面试,在问几个关于序列化的问题之后候选人开始感到不自在,因为缺乏这方面的经验。

他们不知道如何在 Java 中序列化对象,或者他们不熟悉任何 Java 示例来解释序列化,忘记了诸如序列化在 Java 中如何工作,什么是标记接口,标记接口的目的是什么,瞬态变量和可变变量之间的差异,可序列化接口具有多少种方法,在 Java 中,Serializable 和 Externalizable 有什么区别,或者在引入注解之后,为什么不用 @Serializable 注解或替换 Serializable 接口。



在本文中,我们将从初学者和高级别进行提问,这对新手和具有多年 Java 开发经验的高级开发人员同样有益。

关于Java序列化的10个面试问题

大多数商业项目使用数据库或内存映射文件或只是普通文件,来满足持久性要求,只有很少的项目依赖于 Java 中的序列化过程。无论如何,这篇文章不是 Java 序列化教程或如何序列化在 Java 的对象,但有关序列化机制和序列化 API 的面试问

题，这是值得去任何 Java 面试前先看看以免让一些未知的内容惊到自己。

对于那些不熟悉 Java 序列化的人，Java 序列化是用来通过将对象的状态存储到带有.ser扩展名的文件来序列化 Java 中的对象的过程，并且可以通过这个文件恢复重建 Java 对象状态，这个逆过程称为 **deserialization**。

什么是 Java 序列化

序列化是把对象改成可以存到磁盘或通过网络发送到其他运行中的 Java 虚拟机的二进制格式的过程，并可以通过反序列化恢复对象状态。Java 序列化 API 给开发人员提供了一个标准机制，通过 `java.io.Serializable` 和 `java.io.Externalizable` 接口，`ObjectInputStream` 及 `ObjectOutputStream` 处理对象序列化。Java 程序员可自由选择基于类结构的标准序列化或是他们自定义的二进制格式，通常认为后者才是最佳实践，因为序列化的二进制文件格式成为类输出 API 的一部分，可能破坏 Java 中私有和包可见的属性的封装。

如何序列化

让 Java 中的类可以序列化很简单。你的 Java 类只需要实现 `java.io.Serializable` 接口，JVM 就会把 `Object` 对象按默认格式序列化。让一个类是可序列化的需要有意为之。类可序列化可能为是一个长期代价，可能会因此而限制你修改或改变其实现。当你通过实现添加接口来更改类的结构时，添加或删除任何字段可能会破坏默认序列化，这可以通过自定义二进制格式使不兼容的可能性最小化，但仍需要大量的努力来确保向后兼容性。序列化如何限制你更改类的能力的一个示例是 `SerialVersionUID`。

如果不显式声明 `SerialVersionUID`，则 JVM 会根据类结构生成其结构，该结构依赖于类实现接口和可能更改的其他几个因素。假设你新版本的类文件实现的另一个接口，JVM 将生成一个不同的 `SerialVersionUID` 的，当你尝试加载旧版本的程序序列化的旧对象时，你将获得无效类异常 `InvalidClassException`。

问题 1) Java 中的可序列化接口和可外部接口之间的区别是什么？

这是 Java 序列化访谈中最常问的问题。下面是我的版本 `Externalizable` 给我们提供 `writeExternal()` 和 `readExternal()` 方法，这让我们灵活地控制 Java 序列化机制，而不是依赖于 Java 的默认序列化。正确实现 `Externalizable` 接口可以显著提高应用程序的性能。

问题 2) 可序列化方法有多少？如果没有方法，那么可序列化接口的用途是什么？

可序列化 `Serializable` 接口存在于 `java.io` 包中，构成了 Java 序列化机制的核心。它没有任何方法，在 Java 中也称为标记接口。当类实现 `java.io.Serializable` 接口时，它将在 Java 中变得可序列化，并指示编译器使用 Java 序列化机制序列化此对象。

问题 3) 什么是 `serialVersionUID` 如果你不定义这个，会发生什么？

我最喜欢的关于 Java 序列化的问题面试问题之一。`serialVersionUID` 是一个 `private static final long` 型 ID，当它被印在对象上时，它通常是对象的哈希码，你可以使用 `serialver` 这个 JDK 工具来查看序列化对象的 `serialVersionUID`。`SerialVerionUID` 用于对象的版本控制。也可以在类文件中指定 `serialVersionUID`。不指定 `serialVersionUID` 的后果是，当你添加或修改类中的任何字段时，则已序列化类将无法恢复，因为新类和旧序列化对象生成的 `serialVersionUID` 将有所不同。Java 序列化过程依赖于正确的序列化对象恢复状态的，并在序列化对象序列版本不匹配的情况下引发 `java.io.InvalidClassException` 无效类异常。

问题 4) 序列化时，你希望某些成员不要序列化？你如何实现它？

另一个经常被问到的序列化面试问题。这也是一些时候也问，如什么是瞬态 `trasient` 变量，瞬态和静态变量会不会得到序列化等，所以，如果你不希望任何字段是对象的状态的一部分，然后声明它静态或瞬态根据你的需要，这样就不会是在 Java 序列化过程中被包含在内。

问题 5) 如果类中的一个成员未实现可序列化接口, 会发生什么情况?

关于Java序列化过程的一个简单问题。如果尝试序列化实现可序列化的类的对象,但该对象包含对不可序列化类的引用,则在运行时将引发不可序列化异常 `NotSerializableException`, 这就是为什么我始终将一个可序列化警报(在我的代码注释部分中), 代码注释最佳实践之一, 指示开发人员记住这一事实, 在可序列化类中添加新字段时要注意。

问题 6) 如果类是可序列化的, 但其超类不是, 则反序列化后从超级类继承的实例变量的状态如何?

Java 序列化过程仅在对象层次都是可序列化结构中继续, 即实现 Java 中的可序列化接口, 并且从超级类继承的实例变量的值将通过调用构造函数初始化, 在反序列化过程中不可序列化的超级类。一旦构造函数链接将启动, 就不可能停止, 因此, 即使层次结构中较高的类实现可序列化接口, 也将执行构造函数。正如你从陈述中看到的, 这个序列化面试问题看起来非常棘手和有难度, 但如果你熟悉关键概念, 则并不难。

问题 7) 是否可以自定义序列化过程, 或者是否可以覆盖 Java 中的默认序列化过程?

答案是肯定的, 你可以。我们都知道, 对于序列化一个对象需调用

`ObjectOutputStream.writeObject(saveThisObject)`, 并用 `ObjectInputStream.readObject()` 读取对象, 但 Java 虚拟机为你提供的还有一件事, 是定义这两个方法。如果在类中定义这两种方法, 则 JVM 将调用这两种方法, 而不是应用默认序列化机制。你可以在此处通过执行任何类型的预处理或后处理任务来自定义对象序列化和反序列化的行为。

需要注意的重要一点是要声明这些方法为私有方法, 以避免被继承、重写或重载。由于只有 Java 虚拟机可以调用类的私有方法, 你的类的完整性会得到保留, 并且 Java 序列化将正常工作。在我看来, 这是在任何 Java 序列化面试中可以问的最好问题之一, 一个很好的后续问题是, 为什么要为你的对象提供自定义序列化表单?

问题 8) 假设新类的超级类实现可序列化接口, 如何避免新类被序列化?

在 Java 序列化中一个棘手的面试问题。如果类的 Super 类已经在 Java 中实现了可序列化接口, 那么它在 Java 中已经可以序列化, 因为你不能取消接口, 它不可能真正使它无法序列化类, 但是有一种方法可以避免新类序列化。为了避免 Java 序列化, 你需要在类中实现 `writeObject()` 和 `readObject()` 方法, 并且需要从该方法引发不序列化异常 `NotSerializableException`。这是自定义 Java 序列化过程的另一个好处, 如上述序列化面试问题中所述, 并且通常随着面试进度, 它作为后续问题提出。

问题 9) 在 Java 中的序列化和反序列化过程中使用哪些方法?

这是很常见的面试问题, 在序列化基本上面试官试图知道: 你是否熟悉 `readObject()` 的用法、`writeObject()`、`readExternal()` 和 `writeExternal()`。Java 序列化由 `java.io.ObjectOutputStream` 类完成。该类是一个筛选器流, 它封装在较低级别的字节流中, 以处理序列化机制。要通过序列化机制存储任何对象, 我们调用 `ObjectOutputStream.writeObject(savethisobject)`, 并反序列化该对象, 我们称之为 `ObjectInputStream.readObject()` 方法。调用以 `writeObject()` 方法在 java 中触发序列化过程。关于 `readObject()` 方法, 需要注意的一点很重要一点是, 它用于从持久性读取字节, 并从这些字节创建对象, 并返回一个对象, 该对象需要类型强制转换为正确的类型。

问题 10) 假设你有一个类, 它序列化并存储在持久性中, 然后修改了该类以添加新字段。如果对已序列化的对象进行反序列化, 会发生什么情况?

这取决于类是否具有其自己的 `serialVersionUID`。正如我们从上面的问题知道, 如果我们不提供 `serialVersionUID`, 则 Java 编译器将生成它, 通常它等于对象的哈希代码。通过添加任何新字段, 有可能为该类新版本生成的新 `serialVersionUID` 与已序列化的对象不同, 在这种情况下, Java 序列化 API 将引发 `java.io.InvalidClassException`, 因此建议在代码中拥有自己的 `serialVersionUID`, 并确保在单个类中始终保持不变。

11) Java序列化机制中的兼容更改和不兼容更改是什么?

真正的挑战在于通过添加任何字段、方法或删除任何字段或方法来更改类结构，方法是使用已序列化的对象。根据 Java 序列化规范，添加任何字段或方法都面临兼容的更改和更改类层次结构或取消实现的可序列化接口，有些接口在非兼容更改下。对于兼容和非兼容更改的完整列表，我建议阅读 Java 序列化规范。

12) 我们可以通过网络传输一个序列化的对象吗？

是的，你可以通过网络传输序列化对象，因为 Java 序列化对象仍以字节的形式保留，字节可以通过网络发送。你还可以将序列化对象存储在磁盘或数据库中作为 Blob。

13) 在 Java 序列化期间，哪些变量未序列化？

这个问题问得不同，但目的还是一样的，Java 开发人员是否知道静态和瞬态变量的细节。由于静态变量属于类，而不是对象，因此它们不是对象状态的一部分，因此在 Java 序列化过程中不会保存它们。由于 Java 序列化仅保留对象的状态，而不是对象本身。瞬态变量也不包含在 Java 序列化过程中，并且不是对象的序列化状态的一部分。在提出这个问题之后，面试官会询问后续内容，如果你不存储这些变量的值，那么一旦对这些对象进行反序列化并重新创建这些变量，这些变量的价值是多少？这是你们要考虑的。

9. 为什么 Java 中 wait 方法需要在 synchronized 的方法中调用？

另一个棘手的核心 Java 问题，wait 和 notify。它们是在有 synchronized 标记的方法或 synchronized 块中调用的，因为 wait 和 modify 需要监视对其上调用 wait 或 notify-get 的 Object。

大多数 Java 开发人员都知道对象类的 wait()，notify() 和 notifyAll() 方法必须在 Java 中的 synchronized 方法或 synchronized 块中调用，但是我们想过多少次，为什么在 Java 中 wait，notify 和 notifyAll 来自 synchronized 块或方法？

最近这个问题在 Java 面试中被问到我的一位朋友，他思索了一下，并回答说：如果我们不从同步上下文中调用 wait() 或 notify() 方法，我们将在 Java 中收到 IllegalMonitorStateException。

他的回答从实际效果上是正确的，但面试官对这样的答案不会完全满意，并希望向他解释这个问题。面试结束后他和我讨论了同样的问题，我认为他应该告诉面试官关于 Java 中 wait() 和 notify() 之间的竞态条件，如果我们不在同步方法或块中调用它们就可能存在。

让我们看看竞态条件如何在 Java 程序中发生。它也是流行的线程面试问题之一，并经常在电话和面对面的 Java 开发人员面试中出现。因此，如果你正在准备 Java 面试，那么你应该准备这样的问题，并且可以真正帮助你的一本书是《Java 程序员面试公式书》的。这是一本罕见的书，涵盖了 Java 访谈的几乎所有重要主题，例如核心 Java，多线程，IO 和 NIO 以及 Spring 和 Hibernate 等框架。你可以在这里查看。

wait/notify

```
// Thread 1
synchronized(lock) { You must synchronize.
    while(! someCondition()) { Always wait in a loop.
        lock.wait();
    }
}

// Thread 2
synchronized(lock) { Synchronize here too!
    satisfyCondition();
    lock.notifyAll();
}
```



为什么要等待来自 Java 中的 synchronized 方法的 wait 方法为什么必须从 Java 中的 synchronized 块或方法调用？我们主要使用 wait()，notify() 或 notifyAll() 方法用于 Java 中的线程间通信。一个线程在检查条件后正在等待，例如，在经典的生产者 - 消费者问题中，如果缓冲区已满，则生产者线程等待，并且消费者线程通过使用元素在缓冲区中创建空间后通知生产者线程。调用 notify() 或 notifyAll() 方法向单个或多个线程发出一个条件已更改的通知，并且一旦通知线程离开 synchronized 块，正在等待的所有线程开始获取正在等待的对象锁定，幸运的线程在重新获取锁之后从 wait() 方法返回并继续进行。

让我们将整个操作分成几步，以查看 Java 中 wait() 和 notify() 方法之间的竞争条件的可能性，我们将使用 Produce Consumer 线程示例更好地理解方案：

- Producer 线程测试条件(缓冲区是否完整)并确认必须等待(找到缓冲区已满)。
- Consumer 线程在使用缓冲区中的元素后设置条件。
- Consumer 线程调用 notify() 方法；这是不会被听到的，因为 Producer 线程还没有等待。
- Producer 线程调用 wait() 方法并进入等待状态。

因此，由于竞态条件，我们可能会丢失通知，如果我们使用缓冲区或只使用一个元素，生产线程将永远等待，你的程序将挂起。“在 java 同步中等待 notify 和 notifyall 现在让我们考虑如何解决这个潜在的竞态条件？

这个竞态条件通过使用 Java 提供的 synchronized 关键字和锁定来解决。为了调用 wait()，notify() 或 notifyAll()，在 Java 中，我们必须获得对我们调用方法的对象的锁定。由于 Java 中的 wait() 方法在等待之前释放锁定并在从 wait() 返回之前重新获取锁定方法，我们必须使用这个锁来确保检查条件(缓冲区是否已满)和设置条件(从缓冲区获取元素)是原子的，这可以通过在 Java 中使用 synchronized 方法或块来实现。

我不确定这是否是面试官实际期待的，但这个我认为至少有意义，请纠正我如果我错了，请告诉我们是否还有其他令人信服的理由调用 wait()，notify() 或 Java 中的 notifyAll() 方法。

总结一下，我们用 Java 中的 synchronized 方法或 synchronized 块调用 Java 中的 wait()，notify() 或 notifyAll() 方法来避免：

- 1) Java 会抛出 IllegalMonitorStateException，如果我们不调用来自同步上下文的 wait()，notify() 或者 notifyAll() 方法。

2) Javac 中 wait 和 notify 方法之间的任何潜在竞争条件。

10. 你能用 Java 覆盖静态方法吗？如果我在子类中创建相同的方法是编译时错误？

不，你不能在Java中覆盖静态方法，但在子类中声明一个完全相同的方法不是编译时错误，这称为隐藏在Java中的方法。

你不能覆盖Java中的静态方法，因为方法覆盖基于运行时的动态绑定，静态方法在编译时使用静态绑定进行绑定。虽然可以在子类中声明一个具有相同名称和方法签名的方法，看起来可以在Java中覆盖静态方法，但实际上这是方法隐藏。Java不会在运行时解析方法调用，并且根据用于调用静态方法的 Object 类型，将调用相应的方法。这意味着如果你使用父类的类型来调用静态方法，那么原始静态将从父类中调用，另一方面如果你使用子类的类型来调用静态方法，则会调用来自子类的方法。简而言之，你无法在Java中覆盖静态方法。如果你使用像Eclipse或Netbeans这样的Java IDE，它们将显示警告静态方法应该使用类名而不是使用对象来调用，因为静态方法不能在Java中重写。

```
1  /**
2   *
3   * Java program which demonstrate that we can not override static method in Java.
4   * Had Static method can be overridden, with Super class type and sub class object
5   * static method from sub class would be called in our example, which is not the case.
6   */
7  public class CanWeOverrideStaticMethod {
8
9      public static void main(String args[])
10     {
11
12         Screen scrn = new ColorScreen();
13
14         //if we can override static , this should call method from Child class
15         scrn.show(); //IDE will show warning, static method should be called from classname
16
17     }
18
19 }
20
21 class Screen{
22     /
23
24     * public static method which can not be overridden in Java
25     */
26     public static void show()
27     {
28         System.out.printf("Static method from parent class")
29     }
30 }
31
32
33 class ColorScreen extends Screen{
34     /
35
36     * static method of same name and method signature as existed in super
37     * class, this is not method overriding instead this is called
38     * method hiding in Java
39     */
40
41     public static void show()
```

```
public static void show()  
{  
    System.out.println("Overridden static method in Child Class in Java")  
};  
}  
}
```

输出:

Static method from parent class

此输出确认你无法覆盖Java中的静态方法，并且静态方法基于类型信息而不是基于Object进行绑定。如果要覆盖静态method，则会调用子类或ColorScreen中的方法。这一切都在讨论中我们可以覆盖Java中的静态方法。我们已经确认没有，我们不能覆盖静态方法，我们只能在Java中隐藏静态方法。创建具有相同名称和method签名的静态方法称为Java隐藏方法。IDE将显示警告：“静态方法应该使用类名而不是使用对象来调用”，因为静态方法不能在Java中重写。

这些都是我的核心Java面试问题和答案的清单。对于有经验的程序员来说，一些Java问题看起来并不那么难，但对于Java中的中级和初学者来说，它们真的很难回答。顺便说一句，如果你在面试中遇到任何棘手的Java问题，请与我们分享。

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」即进入咱自己的技术交流群。

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

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！

推荐 16 个超级实用的 Java 工具类

alterem Java后端 2019-08-30

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

来源: juejin.im/post/5d4a25b351882505c105cc6e

在Java中，工具类定义了一组公共方法，这篇文章将介绍Java中使用最频繁及最通用的Java工具类。以下工具类、方法按使用流行度排名，参考数据来源于Github上随机选取的5万个开源项目源码。

一. org.apache.commons.io.IOUtils

- closeQuietly: 关闭一个IO流、socket、或者selector且不抛出异常，通常放在finally块
- toString: 转换IO流、 Uri、 byte[]为String
- copy: IO流数据复制，从输入流写到输出流中，最大支持2GB
- toByteArray: 从输入流、URI获取byte[]
- write: 把字节.字符等写入输出流
- toInputStream: 把字符转换为输入流
- readLines: 从输入流中读取多行数据，返回List<String>
- copyLarge: 同copy，支持2GB以上数据的复制
- lineIterator: 从输入流返回一个迭代器，根据参数要求读取的数据量，全部读取，如果数据不够，则失败

二. org.apache.commons.io.FileUtils

- deleteDirectory: 删除文件夹
- readFileToString: 以字符形式读取文件内容
- deleteQuietly: 删除文件或文件夹且不会抛出异常
- copyFile: 复制文件
- writeStringToFile: 把字符写到目标文件，如果文件不存在，则创建
- forceMkdir: 强制创建文件夹，如果该文件夹父级目录不存在，则创建父级
- write: 把字符写到指定文件中
- listFiles: 列举某个目录下的文件(根据过滤器)
- copyDirectory: 复制文件夹
- forceDelete: 强制删除文件

三. org.apache.commons.lang.StringUtils

- isBlank: 字符串是否为空(trim后判断)
- isEmpty: 字符串是否为空(不trim并判断)
- equals: 字符串是否相等
- join: 合并数组为单一字符串，可传分隔符

- `split`: 分割字符串
- `EMPTY`: 返回空字符串
- `trimToNull`: trim后为空字符串则转换为null
- `replace`: 替换字符串

四. org.apache.http.util.EntityUtils

- `toString`: 把Entity转换为字符串
- `consume`: 确保Entity中的内容全部被消费。可以看到源码里又一次消费了Entity的内容，假如用户没有消费，那调用Entity时候将会把它消费掉
- `toByteArray`: 把Entity转换为字节流
- `consumeQuietly`: 和consume一样，但不抛异常
- `getContentCharset`: 获取内容的编码

五. org.apache.commons.lang3.StringUtils

- `isBlank`: 字符串是否为空(trim后判断)
- `isEmpty`: 字符串是否为空(不trim并判断)
- `equals`: 字符串是否相等
- `join`: 合并数组为单一字符串，可传分隔符
- `split`: 分割字符串
- `EMPTY`: 返回空字符串
- `replace`: 替换字符串
- `capitalize`: 首字符大写

六. org.apache.commons.io.FilenameUtils

- `getExtension`: 返回文件后缀名
- `getBaseName`: 返回文件名，不包含后缀名
- `getName`: 返回文件全名
- `concat`: 按命令行风格组合文件路径(详见方法注释)
- `removeExtension`: 删除后缀名
- `normalize`: 使路径正常化
- `wildcardMatch`: 匹配通配符
- `seperatorToUnix`: 路径分隔符改成unix系统格式的，即/
- `getFullPath`: 获取文件路径，不包括文件名
- `isExtension`: 检查文件后缀名是不是传入参数(List<String>)中的一个

七. org.springframework.util.StringUtils

- `hasText`: 检查字符串中是否包含文本

- hasLength: 检测字符串是否长度大于0
- isEmpty: 检测字符串是否为空 (若传入为对象，则判断对象是否为null)
- commaDelimitedStringToArray: 逗号分隔的String转换为数组
- collectionToDelimitedString: 把集合转为CSV格式字符串
- replace 替换字符串
- delimitedListToStringArray: 相当于split
- uncapitalize: 首字母小写
- collectionToDelimitedCommaString: 把集合转为CSV格式字符串
- tokenizeToStringArray: 和split基本一样，但能自动去掉空白的单词

八. org.apache.commons.lang.ArrayUtils

- contains: 是否包含某字符串
- addAll: 添加整个数组
- clone: 克隆一个数组
- isEmpty: 是否空数组
- add: 向数组添加元素
- subarray: 截取数组
- indexOf: 查找某个元素的下标
- equals: 比较数组是否相等
- toObject: 基础类型数据数组转换为对应的Object数组

九. org.apache.commons.lang.StringEscapeUtils

- 参考十五：
org.apache.commons.lang3.StringEscapeUtils

十. org.apache.http.client.utils.URLEncodedUtils

- format: 格式化参数，返回一个HTTP POST或者HTTP PUT可用application/x-www-form-urlencoded字符串
- parse: 把String或者URI等转换为List<NameValuePair>

十一. org.apache.commons.codec.digest.DigestUtils

- md5Hex: MD5加密，返回32位字符串
- sha1Hex: SHA-1加密
- sha256Hex: SHA-256加密
- sha512Hex: SHA-512加密
- md5: MD5加密，返回16位字符串

十二. org.apache.commons.collections.CollectionUtils

- isEmpty: 是否为空
- select: 根据条件筛选集合元素
- transform: 根据指定方法处理集合元素, 类似List的map()
- filter: 过滤元素, 雷瑟List的filter()
- find: 基本和select一样
- collect: 和transform 差不多一样, 但是返回新数组
- forAllDo: 调用每个元素的指定方法
- isEqualCollection: 判断两个集合是否一致

十三. org.apache.commons.lang3.ArrayUtils

- contains: 是否包含某个字符串
- addAll: 添加整个数组
- clone: 克隆一个数组
- isEmpty: 是否空数组
- add: 向数组添加元素
- subarray: 截取数组
- indexOf: 查找某个元素的下标
- equals: 比较数组是否相等
- toObject: 基础类型数据数组转换为对应的Object数组

十四. org.apache.commons.beanutils.PropertyUtils

- getProperty: 获取对象属性值
- setProperty: 设置对象属性值
- getPropertyDescriptor: 获取属性描述器
- isReadable: 检查属性是否可访问
- copyProperties: 复制属性值, 从一个对象到另一个对象
- getPropertyDescriptors: 获取所有属性描述器
- isWriteable: 检查属性是否可写
- getPropertyType: 获取对象属性类型

十五. org.apache.commons.lang3.StringEscapeUtils

- unescapeHtml4: 转义html
- escapeHtml4: 反转义html
- escapeXml: 转义xml
- unescapeXml: 反转义xml
- escapeJava: 转义unicode编码
- escapeEcmaScript: 转义EcmaScript字符
- unescapeJava: 反转义unicode编码

- `escapeJson`: 转义json字符
- `escapeXml10`: 转义Xml10

这个现在已经废弃了，建议使用`commons-text`包里面的方法。

十六. org.apache.commons.beanutils.BeanUtils

- `copyProperties`: 复制属性值，从一个对象到另一个对象
- `getProperty`: 获取对象属性值
- `setProperty`: 设置对象属性值
- `populate`: 根据Map给属性复制
- `copyProperty`: 复制单个值，从一个对象到另一个对象
- `cloneBean`: 克隆bean实例

现在你只要了解了以上16种最流行的工具类方法，你就不必要再自己写工具类了，不必重复造轮子。大部分工具类方法通过其名字就能明白其用途，如果不清楚的，可以看下别人是怎么用的，或者去网上查询其用法。

另外，工具类，根据阿里开发手册，包名如果要使用util不能带s，工具类命名为 XxxUtils

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. 经常用 `HashMap` ?这 6 个问题回答下 !
2. 数据库这么多锁，能锁住小姐姐吗？
3. 小白也能看懂，30 分钟搭建个人博客！
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！

教你用纯 Java 实现一个网页版的 Xshell（附源码）

Java后端 1周前

以下文章来源于Java知音，作者Object



Java知音

专注于java。分享java基础、原理性知识、JavaWeb实战、spring全家桶、设计模式及面试资料、开源项目，助力开发者…

前言

最近由于项目需求，项目中需要实现一个WebSSH连接终端的功能，由于自己第一次做这类型功能，所以首先上了GitHub找了找有没有现成的轮子可以拿来直接用，当时看到了很多这方面的项目，例如：GateOne、webssh、shellinabox等，这些项目都可以很好地实现webssh的功能。

但是最终并没有采用，原因是在于这些底层大都是python写的，需要依赖很多文件，自己用的时候可以使用这种方案，快捷省事，但是做到项目中供用户使用时，总不能要求用户做到服务器中必须包含这些底层依赖，这显然不太合理，所以我决定自己动手写一个WebSSH的功能，并且作为一个独立的项目开源出来。

github项目开源地址：<https://github.com/NoCortY/WebSSH>

技术选型

由于webssh需要实时数据交互，所以会选用长连接的WebSocket，为了开发的方便，框架选用SpringBoot，另外还自己了解了Java用户连接ssh的jsch和实现前端shell页面的xterm.js.

所以，**最终的技术选型就是 SpringBoot+Websocket+jsch+xterm.js。**

导入依赖

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.7.RELEASE</version>
  <relativePath /> <!-- lookup parent from repository -->
</parent>
<dependencies>
  <!-- Web相关 -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <!-- jsch支持 -->
  <dependency>
    <groupId>com.jcraft</groupId>
    <artifactId>jsch</artifactId>
    <version>0.1.54</version>
  </dependency>
  <!-- WebSocket 支持 -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
  </dependency>
  <!-- 文件上传解析器 -->
  <dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.4</version>
  </dependency>
  <dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.1</version>
  </dependency>
</dependencies>
```

一个简单的 xterm 案例

由于xterm是一个冷门技术，所以很多同学并没有这方面的知识支撑，我也是为了实现这个功能所以临时学的，所以在这给大家介绍一下。

xterm.js是一个基于WebSocket的容器，它可以帮助我们在前端实现命令行的样式。就像是我们平常再用SecureCRT或者XShell连接服务器时一样。

下面是官网上的入门案例：

```
<!doctype html>
<html>
<head>
<link rel="stylesheet" href="node_modules/xterm/css/xterm.css" />
<script src="node_modules/xterm/lib/xterm.js"></script>
</head>
<body>
<div id="terminal"></div>
<script>
var term = new Terminal();
term.open(document.getElementById('terminal'));
term.write('Hello from \x1B[1;31mxterm.js\x1B[0m $ ')
</script>
</body>
</html>
```

最终测试，页面就是下面这个样子：



xterm入门

可以看到页面已经出现了类似与shell的样式，那就根据这个继续深入，实现一个webssh。

后端实现

由于xterm只要只是实现了前端的样式，并不能真正地实现与服务器交互，与服务器交互主要还是靠我们Java后端来进行控制的，所以我们从后端开始，使用jsch+websocket实现这部分内容。

WebSocket配置

由于消息实时推送到前端需要用到WebSocket，不了解WebSocket的同学可以先去自行了解一下，这里就不过多介绍了，我们直接开始进行WebSocket的配置。

```

/**
 * @Description: websocket配置
 * @Author: NoCortY
 * @Date: 2020/3/8
 */
@Configuration
@EnableWebSocket
public class WebSSHWebSocketConfig implements WebSocketConfigurer{
    @Autowired
    WebSSHWebSocketHandler webSSHWebSocketHandler;
    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry webSocketHandlerRegistry) {
        //socket通道
        //指定处理器和路径，并设置跨域
        webSocketHandlerRegistry.addHandler(webSSHWebSocketHandler, "/webssh")
            .addInterceptors(new WebSocketInterceptor())
            .setAllowedOrigins("/*");
    }
}

```

处理器(Handler)和拦截器(Interceptor)的实现

刚才我们完成了WebSocket的配置，并指定了一个处理器和拦截器。所以接下来就是处理器和拦截器的实现。

拦截器：

```

public class WebSocketInterceptor implements HandshakeInterceptor {
    /**
     * @Description: Handler处理前调用
     * @Param: [serverHttpRequest, serverHttpResponse, webSocketHandler, map]
     * @return: boolean
     * @Author: NoCortY
     * @Date: 2020/3/1
     */
    @Override
    public boolean beforeHandshake(ServerHttpRequest serverHttpRequest, ServerHttpResponse serverHttpResponse, WebSocketHandler webSocketHandler, Map map) {
        if (serverHttpRequest instanceof ServletServerHttpRequest) {
            ServletServerHttpRequest request = (ServletServerHttpRequest) serverHttpRequest;
            //生成一个UUID，这里由于是独立的项目，没有用户模块，所以可以用随机的UUID
            //但是如果要集成到自己的项目中，需要将其改为自己识别用户的标识
            String uuid = UUID.randomUUID().toString().replace("-", "");
            //将uuid放到websocketSession中
            map.put(ConstantPool.USER_UUID_KEY, uuid);
            return true;
        } else {
            return false;
        }
    }

    @Override
    public void afterHandshake(ServerHttpRequest serverHttpRequest, ServerHttpResponse serverHttpResponse, WebSocketHandler webSocketHandler, Map map) {
    }
}

```

处理器：

```

/**
 * @Description: WebSSH的WebSocket处理器
 * @Author: NoCortY
 * @Date: 2020/3/9
 */

```

```
/* @Date: 2020/3/8
*/
@Component
public class WebSSHWebSocketHandler implements WebSocketHandler{
    @Autowired
    private WebSSHService webSSHService;
    private Logger logger = LoggerFactory.getLogger(WebSSHWebSocketHandler.class);

    /**
     * @Description: 用户连接上WebSocket的回调
     * @Param: [webSocketSession]
     * @return: void
     * @Author: Object
     * @Date: 2020/3/8
     */
    @Override
    public void afterConnectionEstablished(WebSocketSession webSocketSession) throws Exception {
        logger.info("用户:{}连接WebSSH", webSocketSession.getAttributes().get(ConstantPool.USER_UUID_KEY));
        //调用初始化连接
        webSSHService.initConnection(webSocketSession);
    }

    /**
     * @Description: 收到消息的回调
     * @Param: [webSocketSession, webSocketMessage]
     * @return: void
     * @Author: NoCortY
     * @Date: 2020/3/8
     */
    @Override
    public void handleMessage(WebSocketSession webSocketSession, WebSocketMessage<?> webSocketMessage) throws Exception {
        if (webSocketMessage instanceof TextMessage) {
            logger.info("用户:{}发送命令:{}",
                    webSocketSession.getAttributes().get(ConstantPool.USER_UUID_KEY), webSocketMessage);
            //调用service接收消息
            webSSHService.recvHandle(((TextMessage) webSocketMessage).getPayload(), webSocketSession);
        } else if (webSocketMessage instanceof BinaryMessage) {

        } else if (webSocketMessage instanceof PongMessage) {

        } else {
            System.out.println("Unexpected WebSocket message type: " + webSocketMessage);
        }
    }

    /**
     * @Description: 出现错误的回调
     * @Param: [webSocketSession, throwable]
     * @return: void
     * @Author: Object
     * @Date: 2020/3/8
     */
    @Override
    public void handleTransportError(WebSocketSession webSocketSession, Throwable throwable) throws Exception {
        logger.error("数据传输错误");
    }

    /**
     * @Description: 连接关闭的回调
     * @Param: [webSocketSession, closeStatus]
     * @return: void
     * @Author: NoCortY
     * @Date: 2020/3/8
     */
    @Override
```

```
public void afterConnectionClosed(WebSocketSession webSocketSession, CloseStatus closeStatus) throws Exception {
    logger.info("用户:{}断开webssh连接", String.valueOf(webSocketSession.getAttributes().get(ConstantPool.USER_UUID_KEY)))
    //调用service关闭连接
    webSSHSERVICE.close(webSocketSession);
}

@Override
public boolean supportsPartialMessages() {
    return false;
}
}
```

需要注意的是，我在拦截器中加入的用户标识是使用了**随机的UUID**，这是因为作为一个独立的websocket项目，没有用户模块，如果需要将这个项目集成到自己的项目中，需要修改这部分代码，将其改为自己项目中识别一个用户所用的用户标识。

WebSSH的业务逻辑实现（核心）

刚才我们实现了websocket的配置，都是一些死代码，实现了接口再根据自身需求即可实现，现在我们将进行后端主要业务逻辑的实现，在实现这个逻辑之前，我们先来想想，WebSSH，我们主要想要呈现一个什么效果。

我这里做了一个总结：

- 1.首先我们得先连接上终端（初始化连接）
- 2.其次我们的服务端需要处理来自前端的消息（接收并处理前端消息）
- 3.我们需要将终端返回的消息回写到前端（数据回写前端）
- 4.关闭连接

根据这四个需求，我们先定义一个接口，这样可以让需求明了起来。

```

/**
 * @Description: WebSSH的业务逻辑
 * @Author: NoCortY
 * @Date: 2020/3/7
 */
public interface WebSSHSERVICE {
    /**
     * @Description: 初始化ssh连接
     * @Param:
     * @return:
     * @Author: NoCortY
     * @Date: 2020/3/7
     */
    public void initConnection(WebSocketSession session);

    /**
     * @Description: 处理客户段发的数据
     * @Param:
     * @return:
     * @Author: NoCortY
     * @Date: 2020/3/7
     */
    public void recvHandle(String buffer, WebSocketSession session);

    /**
     * @Description: 数据写回前端 for websocket
     * @Param:
     * @return:
     * @Author: NoCortY
     * @Date: 2020/3/7
     */
    public void sendMessage(WebSocketSession session, byte[] buffer) throws IOException;

    /**
     * @Description: 关闭连接
     * @Param:
     * @return:
     * @Author: NoCortY
     * @Date: 2020/3/7
     */
    public void close(WebSocketSession session);
}

```

现在我们可以根据这个接口去实现我们定义的功能了。

1. 初始化连接

由于我们的底层是依赖jsch实现的，所以这里是需要使用jsch去建立连接的。而所谓初始化连接，实际上就是将我们所需要的连接信息，保存在一个Map中，这里并不进行任何的真实连接操作。为什么这里不直接进行连接？因为这里前端只是连接上了WebSocket，但是我们还需要前端给我们发来linux终端的用户名和密码，没有这些信息，我们是无法进行连接的。

```

public void initConnection(WebSocketSession session) {
    JSch jSch = new JSch();
    SSHConnectInfo sshConnectInfo = new SSHConnectInfo();
    sshConnectInfo.setjSch(jSch);
    sshConnectInfo.setWebSocketSession(session);
    String uuid = String.valueOf(session.getAttributes().get(ConstantPool.USER_UUID_KEY));
    //将这个ssh连接信息放入map中
    sshMap.put(uuid, sshConnectInfo);
}

```

2. 处理客户端发送的数据

在这一步骤中，我们会分为两个分支。

- 第一个分支：如果客户端发来的是终端的用户名和密码等信息，那么我们进行终端的连接。
- 第二个分支：如果客户端发来的是操作终端的命令，那么我们就直接转发到终端并且获取终端的执行结果。

具体代码实现：

```
public void recvHandle(String buffer, WebSocketSession session) {  
    ObjectMapper objectMapper = new ObjectMapper();  
    WebSSHData webSSHData = null;  
    try {  
        //转换前端发送的JSON  
        webSSHData = objectMapper.readValue(buffer, WebSSHData.class);  
    } catch (IOException e) {  
        logger.error("Json转换异常");  
        logger.error("异常信息:{}", e.getMessage());  
        return;  
    }  
    //获取刚才设置的随机的uuid  
    String userId = String.valueOf(session.getAttributes().get(ConstantPool.USER_UUID_KEY));  
    if (ConstantPool.WEBSSH_OPERATE_CONNECT.equals(webSSHData.getOperate())) {  
        //如果是连接请求  
        //找到刚才存储的ssh连接对象  
        SSHConnectInfo sshConnectInfo = (SSHConnectInfo) sshMap.get(userId);  
        //启动线程异步处理  
        WebSSHData finalWebSSHData = webSSHData;  
        executorService.execute(new Runnable() {  
            @Override  
            public void run() {  
                try {  
                    //连接到终端  
                    connectToSSH(sshConnectInfo, finalWebSSHData, session);  
                } catch (JSchException | IOException e) {  
                    logger.error("webssh连接异常");  
                    logger.error("异常信息:{}", e.getMessage());  
                    close(session);  
                }  
            }  
        });  
    } else if (ConstantPool.WEBSSH_OPERATE_COMMAND.equals(webSSHData.getOperate())) {  
        //如果是发送命令的请求  
        String command = webSSHData.getCommand();  
        SSHConnectInfo sshConnectInfo = (SSHConnectInfo) sshMap.get(userId);  
        if (sshConnectInfo != null) {  
            try {  
                //发送命令到终端  
                transToSSH(sshConnectInfo.getChannel(), command);  
            } catch (IOException e) {  
                logger.error("webssh连接异常");  
                logger.error("异常信息:{}", e.getMessage());  
                close(session);  
            }  
        }  
    } else {  
        logger.error("不支持的操作");  
        close(session);  
    }  
}
```

3. 数据通过websocket发送到前端

```
public void sendMessage(WebSocketSession session, byte[] buffer) throws IOException {
    session.sendMessage(new TextMessage(buffer));
}
```

4. 关闭连接

```
public void close(WebSocketSession session) {
    //获取随机生成的uuid
    String userId = String.valueOf(session.getAttributes().get(ConstantPool.USER_UUID_KEY));
    SSHConnectInfo sshConnectInfo = (SSHConnectInfo) sshMap.get(userId);
    if (sshConnectInfo != null) {
        //断开连接
        if (sshConnectInfo.getChannel() != null) sshConnectInfo.getChannel().disconnect();
        //map中移除该ssh连接信息
        sshMap.remove(userId);
    }
}
```

至此，我们的整个后端实现就结束了，由于篇幅有限，这里将一些操作封装成了方法，就不做过多展示了，重点讲逻辑实现的思路吧。接下来我们将进行前端的实现。

前端实现

前端工作主要分为这么几个步骤：

1. 页面的实现
2. 连接WebSocket并完成数据的接收并回写
3. 数据的发送

所以我们一步一步来实现它。

页面实现

页面的实现很简单，我们只不过需要在一整个屏幕上都显示终端那种大黑屏幕，所以我们并不用写什么样式，只需要创建一个div，之后将terminal实例通过xterm放到这个div中，就可以实现了。

```
<!doctype html>
<html>
<head>
    <title>WebSSH</title>
    <link rel="stylesheet" href="../css/xterm.css" />
</head>
<body>
    <div id="terminal" style="width: 100%;height: 100%"></div>

    <script src="../lib/jquery-3.4.1/jquery-3.4.1.min.js"></script>
    <script src="../js/xterm.js" charset="utf-8"></script>
    <script src="../js/webssh.js" charset="utf-8"></script>
    <script src="../js/base64.js" charset="utf-8"></script>
</body>
</html>
```

```

openTerminal(){

    //这里的內容可以写死，但是要整合到项目中时，需要通过参数的方式传入，可以动态连接某个终端。
    operate:'connect',
    host: 'ip地址',
    port: '端口号',
    username: '用户名',
    password: '密码'
};

function openTerminal(options){
    var client = new WSSHClient();
    var term = new Terminal({
        cols: 97,
        rows: 37,
        cursorBlink: true, // 光标闪烁
        cursorStyle: "block", // 光标样式 null / 'block' / 'underline' / 'bar'
        scrollback: 800, //回滚
        tabStopWidth: 8, //制表宽度
        screenKeys: true
    });

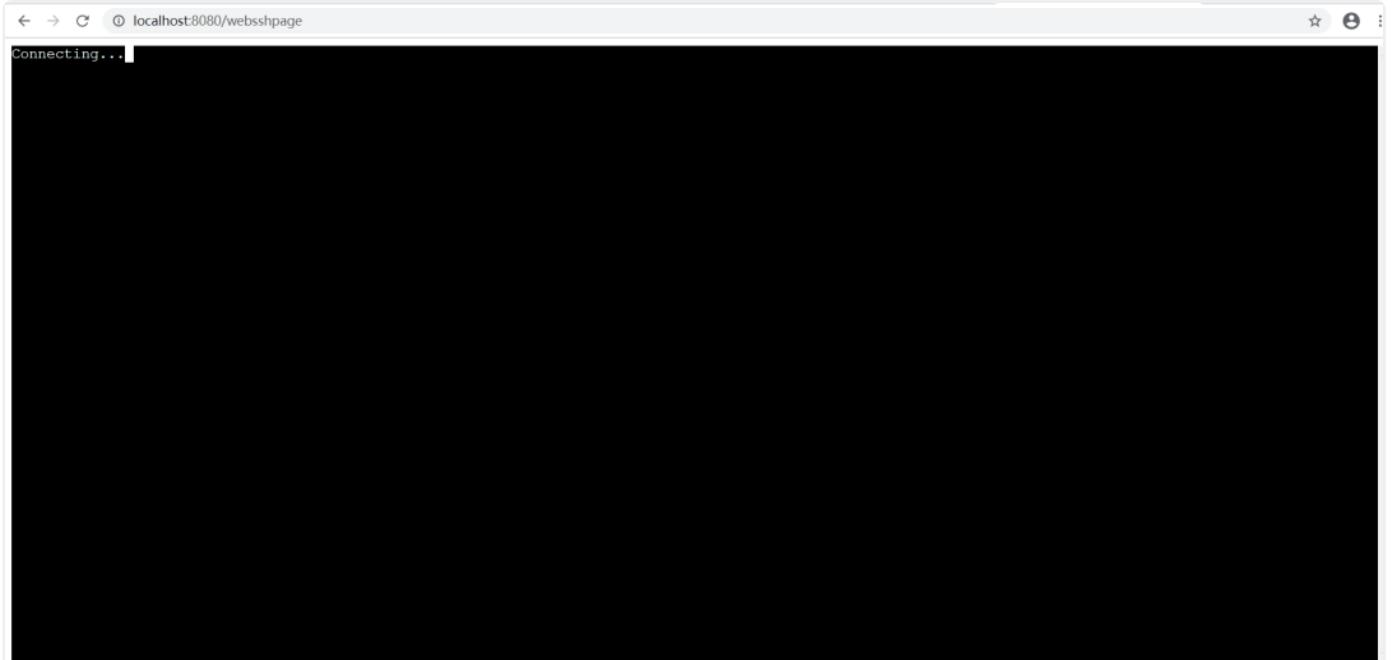
    term.on('data', function (data) {
        //键盘输入时的回调函数
        client.sendClientData(data);
    });
    term.open(document.getElementById('terminal'));
    //在页面上显示连接中...
    term.write('Connecting...');

    //执行连接操作
    client.connect({
        onError: function (error) {
            //连接失败回调
            term.write('Error: ' + error + '\r\n');
        },
        onConnect: function () {
            //连接成功回调
            client.sendInitData(options);
        },
        onClose: function () {
            //连接关闭回调
            term.write("\rconnection closed");
        },
        onData: function (data) {
            //收到数据时回调
            term.write(data);
        }
    });
}
}

```

效果展示

连接



连接

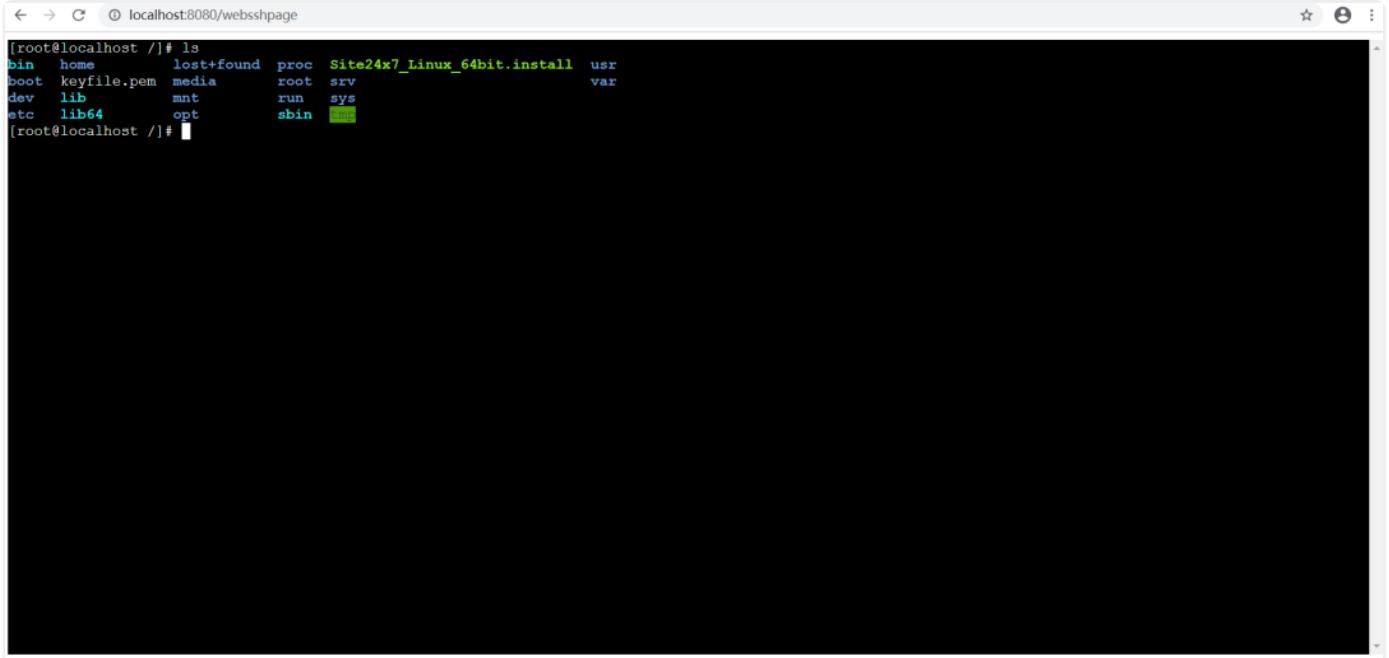
连接成功



连接成功

命令操作

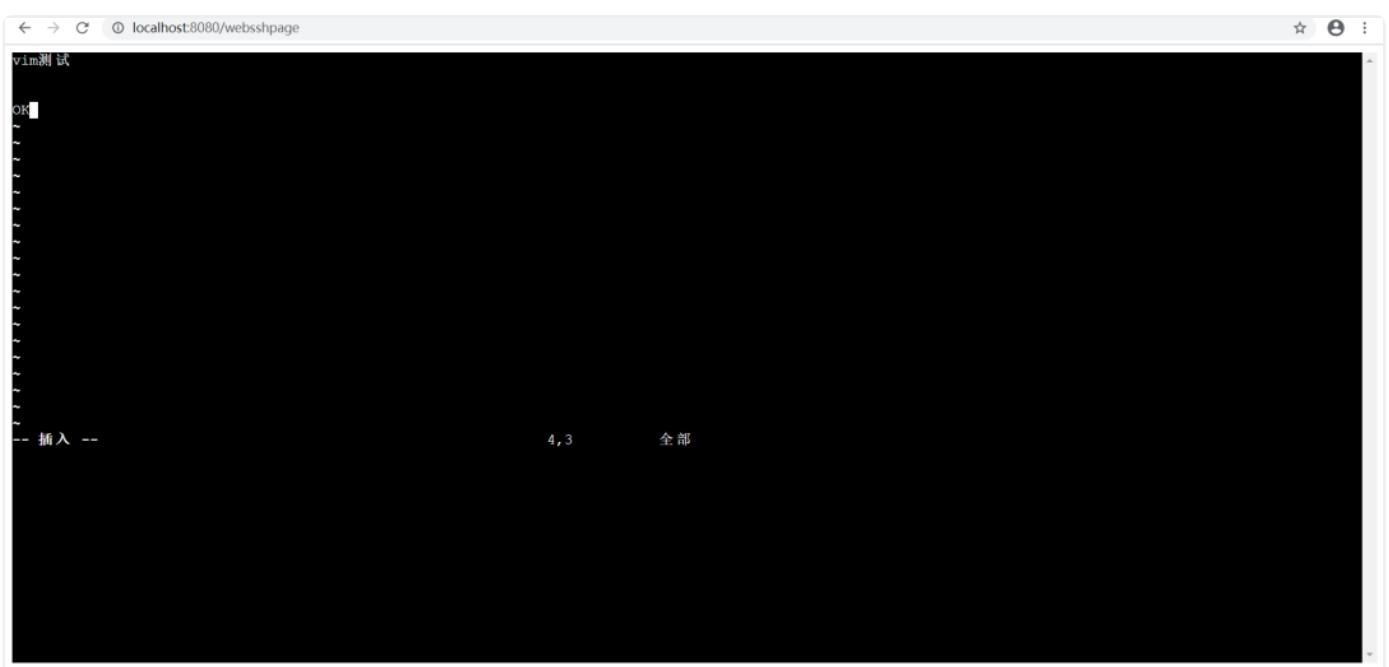
ls命令：



```
[root@localhost /]# ls
bin   home      lost+found  proc  Site24x7_Linux_64bit.install  usr
boot  keyfile.pem  media    root   srv
dev   lib        mnt       run    sys
etc   lib64     opt       sbin  [..]
```

ls命令

vim编辑器:



```
vim测试
OK
```

vim编辑器

top命令:

```
top - 19:49:10 up 1 day, 5:38, 3 users, load average: 0.00, 0.01, 0.05
Tasks: 92 total, 1 running, 87 sleeping, 0 stopped, 4 zombie
%Cpu(s): 0.0 us, 0.3 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 498744 total, 53012 free, 205388 used, 240344 buff/cache
KiB Swap: 1048572 total, 1043700 free, 4872 used. 278728 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
9	root	20	0	0	0	0	S	0.3	0.0	0:30.14	rcu_sched
307	root	0	-20	0	0	0	S	0.3	0.0	0:02.14	kworker/0:1H
1	root	20	0	127936	4764	2780	S	0.0	1.0	0:09.74	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.01	kthreadd
4	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H
6	root	20	0	0	0	0	S	0.0	0.0	0:02.94	ksoftirqd/0
7	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	migration/0
8	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_bh
10	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	lru-add-dra+
11	root	rt	0	0	0	0	S	0.0	0.0	0:02.06	watchdog/0
13	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kdevtmpfs
14	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	netns
15	root	20	0	0	0	0	S	0.0	0.0	0:00.02	khungtaskd
16	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	writeback
17	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kintegrityd
18	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	bioset
19	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	bioset

top命令

结语

这样我们就完成了一个webssh项目的实现，没有依赖其它任何的组件，后端完全使用Java实现，由于用了SpringBoot，非常容易部署。

但是，我们还可以对这个项目进行扩展，比如新增上传或下载文件，就像Xftp一样，可以很方便地拖拽式上传下载文件。

这个项目之后我会持续更新，上述功能也会慢慢实现，Github: <https://github.com/NoCortY/WebSSH>

喜欢可以给个Star哦

- END -

推荐阅读

1. Java 中一个令人惊讶的 BUG
2. 如何更新线上的 Java 服务器代码
3. 你知道 Spring 中运用的 9 种设计模式吗？
4. Java 中的继承和多态（深入版）



微信搜一搜

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！

最强 Java Redis 客户端

唐尤华 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

来源 | dzone.com/articles/java-distributed-caching-in-redis

为什么要在 Java 分布式应用程序中使用缓存?

在提高应用程序速度和性能上, 每一毫秒都很重要。根据谷歌的一项研究, 假如一个网站在3秒钟或更短时间内没有加载成功, 会有 53% 的手机用户会离开。

缓存是让分布式应用程序加速的重要技术之一。存储的信息越接近 CPU, 访问速度就越快。从 CPU 缓存中加载数据比从 RAM 中加载要快得多, 比从硬盘或网络上加载要快得多得多。

要存储经常访问的数据, 分布式应用程序需要在多台机器中维护缓存。分布式缓存是降低分布式应用程序延迟、提高并发性和可伸缩性的一种重要策略。

Redis 是一种流行的开源内存数据存储, 可用作数据库、缓存或消息代理。由于是从内存而非磁盘加载数据, Redis 比许多传统的数据库解决方案更快。

然而, 对开发者来说让 Redis 分布式缓存正确工作是一个巨大挑战。例如, 必须谨慎处理本地缓存失效, 即替换或删除缓存条目。每次更新或删除存储计算机本地缓存中的信息时, 必须更新分布式缓存系统所有计算机内存中的缓存。

好消息是, 有一些类似 Redisson 这样的 Redis 框架, 可以帮助构建应用程序所需的分布式缓存。下一节将讨论 Redisson 中分布式缓存的三个重要实现: Maps、Spring Cache 和 JCache。

1. Redisson 分布式缓存

Redisson 是一个基于 Redis 的框架, 用 Java 实现了一个 Redis 包装器 (wrapper) 和接口。Redisson 包含许多常见的 Java 类, 例如分布式对象、分布式服务、分布式锁和同步器, 以及分布式集合。正如下面即将介绍的, 其中一些接口同时支持分布式缓存和本地缓存。

2. Map

Map 是 Java 最有用的集合之一。Redisson 提供了一个名为 RMap 的 Java Map 实现, 支持本地缓存。

如果希望执行多个读操作或网络环回 (roundtrip), 应使用支持本地缓存的 RMap。通过本地存储 Map 数据, RMap 比不启用本地缓存时快45倍。通用分布式缓存使用 RMapCache, 本地缓存使用 RLocalCachedMap。

Redis 引擎自身能够执行缓存, 不需要在客户端执行代码。然而, 虽然本地缓存能显著提高读取速度, 但需要由开发人员维护, 并且可能需要一些开发工作。Redisson 为开发人员提供了 RLocalCachedMap 对象, 让本地缓存实现起来更容易。

下面的代码展示了如何初始化 RMapCache 对象:

```
RMapCache<String, SomeObject> map = redisson.getMapCache("anyMap");
map.put("key1", new SomeObject(), 10, TimeUnit.MINUTES, 10, TimeUnit.SECONDS);
```

上面的代码将字符串 "key1" 放到 RMapCache 中, 并与 SomeObject() 关联。然后它指定了两个参数, TTL设为10分钟、最大空闲时间10秒。

当不再需要时, 应销毁 RMapCache 对象:

```
map.destroy();
```

3. Spring Cache

Spring 是一个用于构建企业级 Web 应用程序的 Java 框架，也提供了缓存支持。

Redisson 包含了 Spring 缓存功能，提供两个对象：RedissonSpringCacheManager 和 RedissonSpringLocalCachedCacheManager。RedissonSpringLocalCachedCacheManager 支持本地缓存。

下面是一个 RedissonSpringLocalCachedCacheManager 对象的示例配置：

```
@Configuration
@ComponentScan
@EnableCaching
public static class Application {
    @Bean(destroyMethod="shutdown")
    RedissonClient redisson() throws IOException {
        Config config = new Config();
        config.useClusterServers()
            .addNodeAddress("127.0.0.1:7004", "127.0.0.1:7001");
        return Redisson.create(config);
    }
    @Bean
    CacheManager cacheManager(RedissonClient redissonClient) {
        Map<String, CacheConfig> config = new HashMap<String, CacheConfig>();
        // 新建 "testMap" 缓存: ttl=24分钟, maxIdleTime=12分钟
        config.put("testMap", new CacheConfig(24*60*1000, 12*60*1000));
        return new RedissonSpringCacheManager(redissonClient, config);
    }
}
```

此外，还可以读取 JSON 或 YAML 文件配置 RedissonSpringCacheManager。

与 RMaps 一样，每个 RedissonSpringCacheManager 实例都有两个重要参数: ttl（生存时间）和 maxIdleTime。如果这些参数设为0或者没有定义，那么数据将无限期地保留在缓存中。

4. JCache

JCache 是一个 Java 缓存 API，允许开发人员从缓存临时存储、检索、更新和删除对象。

Redisson 提供了 Redis 的 JCache API 实现。下面是在 Redisson 中使用默认配置调用 JCache API 的示例：

```
MutableConfiguration<String, String> config = new MutableConfiguration<>();
CacheManager manager = Caching.getCacheProvider().getCacheManager();
Cache<String, String> cache = manager.createCache("namedCache", config);
```

此外，还可以使用自定义配置文件、Redisson Config 对象或 Redisson RedissonClient 对象配置 JCache。例如，下面的代码使用自定义 Redisson 配置来调用 JCache：

```
MutableConfiguration<String, String> jcacheConfig = new MutableConfiguration<>();
Config redissonCfg = ...
Configuration<String, String> config = RedissonConfiguration.fromConfig(redissonCfg, jcacheConfig);
CacheManager manager = Caching.getCacheProvider().getCacheManager();
Cache<String, String> cache = manager.createCache("namedCache", config);
```

Redisson 的 JCache 实现已经通过 JCache TCK 的所有测试。你也可以自行验证。

让我们愉快地使用缓存吧！

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 

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!

最近面试 Java 后端开发的感受

hsm_computer Java后端 2019-11-28

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | hsm_computer

链接 | cnblogs.com/JavaArchitect/p/10011253.html

在上周，我密集面试了若干位Java后端的候选人，工作经验在3到5年间。我的标准其实不复杂：第一能干活，第二Java基础要好，第三最好熟悉些分布式框架，我相信其它公司招初级开发时，应该也照着这个标准来面的。

我也知道，不少候选人能力其实不差，但面试时没准备或不会说，这样的人可能在进团队干活后确实能达到期望，但可能就无法通过面试，但面试官总是只根据面试情况来判断。

但现实情况是，大多数人可能面试前没准备，或准备方法不得当。要知道，我们平时干活更偏重于业务，不可能大量接触到算法，数据结构，底层代码这类面试必问的问题点，换句话说，面试准备点和平时工作要点匹配度很小。

作为面试官，我只能根据候选人的回答来决定面试结果。不过，与人方便自己方便，所以我在本文里，将通过一些常用的问题来介绍面试的准备技巧。大家在看后一定会感叹：只要方法得当，准备面试第一不难，第二用的时间也不会太多。

1、框架是重点，但别让人感觉你只会山寨别人的代码

在面试前，我会阅读简历以查看候选人在框架方面的项目经验，在候选人的项目介绍的环节，我也会着重关注候选人最近的框架经验，目前比较热门的是SSM。

不过，一般工作在5年内的候选人，大多仅仅是能“山寨”别人的代码，也就是说能在现有框架的基础上，照着别人写的流程，扩展出新的功能模块。比如要写个股票挂单的功能模块，是会模仿现有的下单流程，然后从前端到后端再到数据库，依样画葫芦写一遍，最多把功能相关的代码点改掉。

其实我们每个人都这样过来的，但在面试时，如果你仅仅表现出这样的能力，就和大多数人的水平差不多了，在这点就没法体现出你的优势了。

我们知道，如果单纯使用SSM框架，大多数项目都会有痛点。比如数据库性能差，或者业务模块比较复杂，并发量比较高，用Spring MVC里的Controller无法满足跳转的需求。所以我一般还会主动问：你除了依照现有框架写业务代码时，还做了哪些改动？

我听到的回答有：增加了Redis缓存，以避免频繁调用一些不变的数据。或者，在MyBatis的xml里，select语句where条件有isnull，即这个值有就增加一个where条件，对此，会对任何一个where增加一个不带isnull的查询条件，以免该语句当传入参数都是null时，做全表扫描。或者，干脆说，后端异步返回的数据量很大，时间很长，我在项目里就调大了异步返回的最大时间，或者对返回信息做了压缩处理，以增加网络传输性能。

对于这个问题，我不在乎听到什么回答，我只关心回答符不符合逻辑。一般只要答对，我就会给出“在框架层面有自己的体会，有一定的了解”，否则，我就只会给出“只能在项目经理带领下编写框架代码，对框架本身了解不多”。

其实，在准备面试时，归纳框架里的要点并不难，我就不信所有人在做项目时一点积累也没，只要你说出来，可以说，这方面你就碾压了将近7成的竞争者。

2、别单纯看单机版的框架，适当了解些分布式

此外，在描述项目里框架技术时，最好你再带些分布式的技术。下面我列些大家可以准备的分布式技术。

1、反向代理方面，nginx的基本配置，比如如何通过lua语言设置规则，如何设置session粘滞。如果可以，再看些nginx的底层，比如协议，集群设置，失效转移等。

2、远程调用dubbo方面，可以看下dubbo和zookeeper整合的知识点，再深一步，了解下dubbo底层的传输协议和序列化方式。

3、消息队列方面，可以看下kafka或任意一种组件的使用方式，简单点可以看下配置，工作组的设置，再深入点，可以看下Kafka集群，持久化的方式，以及发送消息是用长连接还是短拦截。

以上仅仅是用3个组件举例，大家还可以看下Redis缓存，日志框架，MyCAT分库分表等。准备的方式有两大类，第一是要会说怎么用，这比较简单，能通过配置文件搭建成一个功能模块即可，第二是可以适当读些底层代码，以此了解下协议，集群和失效转移之类的高级知识点。

如果能在面试中侃侃而谈分布式组件的底层，那么得到的评价就会比较好，比如“深入了解框架底层”，或“框架经验丰富”，这样就算去面试架构师也行了，更何况是高级开发。

3、数据库方面，别就知道增删改查，得了解性能优化

在实际项目里，大多数程序员用到的可能仅仅是增删改查，当我们用Mybatis时，这个情况更普遍。不过如果你面试时也这样表现，估计你的能力就和其它竞争者差不多了。

这方面，你可以准备如下的技能。

1、SQL高级方面，比如group by, having，左连接，子查询（带in），行转列等高级用法。

2、建表方面，你可以考虑下，你项目是用三范式还是反范式，理由是什么？

3、尤其是优化，你可以准备下如何通过执行计划查看SQL语句改进点的方式，或者其它能改善SQL性能的方式（比如建索引等）。

4、如果你感觉有能力，还可以准备些MySQL集群，MyCAT分库分表的技能。比如通过LVS+Keepalived实现MySQL负载均衡，MyCAT的配置方式。同样，如果可以，也看些相关的底层代码。

哪怕你在前三点表现一般，那么至少也能超越将近一般的候选人，尤其当你在SQL优化方面表现非常好，那么你在面试高级开发时，数据库层面一定是达标的，如果你连第四点也回答非常好，那么恭喜你，你在数据库方面的能力甚至达到了初级架构的级别。

4、Java核心方面，围绕数据结构和性能优化准备面试题

Java核心这块，网上的面试题很多，不过在此之外，大家还应当着重关注集合（即数据结构）和多线程并发这两块，在此基础上，大家可以准备些设计模式和虚拟机的说辞。

下面列些我一般会问的部分问题：

- 1、String a = "123"; String b = "123"; a==b的结果是什么？这包含了内存，String存储方式等诸多知识点。
- 2、HashMap里的hashCode方法和equal方法什么时候需要重写？如果不重写会有什么后果？对此大家可以进一步了解HashMap（甚至ConcurrentHashMap）的底层实现。
- 3、ArrayList和LinkedList底层实现有什么差别？它们各自适用于哪些场合？对此大家也可以了解下相关底层代码。
- 4、volatile关键字有什么作用？由此展开，大家可以了解下线程内存和堆内存的差别。
- 5、CompletableFuture，这个是JDK1.8里的新特性，通过它怎么实现多线程并发控制？
- 6、JVM里，new出来的对象是在哪个区？再深入一下，问下如何查看和优化JVM虚拟机内存。
- 7、Java的静态代理和动态代理有什么差别？最好结合底层代码来说。

通过上述的问题点，我其实不仅仅停留在“会用”级别，比如我不会问如何在ArrayList里放元素。大家可以看到，上述问题包含了“多线程并发”，“JVM优化”，“数据结构对象底层代码”等细节，大家也可以举一反三，通过看一些高级知识，多准备些其它类似面试题。

我们知道，目前Java开发是以Web框架为主，那么为什么还要问Java核心知识点呢？我这个是有切身体会的。

之前在我团队里，我见过两个人，一个是就会干活，具体表现是会用Java核心基本的API，而且也没有深入了解的意愿（估计不知道该怎么深入了解），另一位平时专门会看些Java并发，虚拟机等的高级知识。过了半年以后，后者的能力快速升级到高级开发，由于对JAVA核心知识点了解很透彻，所以看一些分布式组件的底层实现没什么大问题。而前者，一直在重复劳动，能力也只一直停留在“会干活”的层面。

而在现实的面试中，如果不熟悉Java核心知识点，估计升高级开发都难，更别说是面试架构师级别的岗位了。

5、Linux方面，至少了解如何看日志排查问题

如果候选人能证明自己有“排查问题”和“解决问题”的能力，这绝对是个加分项，但怎么证明？

目前大多数的互联网项目，都是部署在Linux上，也就是说，日志都是在Linux，下面归纳些实际的Linux操作。

- 1、能通过less命令打开文件，通过Shift+G到达文件底部，再通过?+关键字的方式来根据关键来搜索信息。
- 2、能通过grep的方式查关键字，具体用法是, grep 关键字 文件名，如果要两次在结果里查找的话，就用grep 关键字1 文件名 | 关键字2 --color。最后--color是高亮关键字。
- 3、能通过vi来编辑文件。
- 4、能通过chmod来设置文件的权限。

当然，还有更多更实用的Linux命令，但在实际面试过程中，不少候选人连一条linux命令也不知道。还是这句话，你哪怕知道些很基本的，也比一般人强了。

6、通读一段底层代码，作为加分项

如何证明自己对一个知识点非常了解？莫过于能通过底层代码来说明。我在和不少工作经验在5年之内的程序员沟通时，不少人认为这很难？确实，如果要通过阅读底层代码了解分布式组件，那难度不小，但如果如下部分的底层代码，并不难懂。

- 1、ArrayList,LinkedList的底层代码里，包含着基于数组和链表的实现方式，如果大家能以此讲清楚扩容，“通过枚举器遍历”等方式，绝对能证明自己。
- 2、HashMap直接对应着Hash表这个数据结构，在HashMap的底层代码里，包含着hashcode的put，get等的操作，甚至在 ConcurrentHashMap里，还包含着Lock的逻辑。我相信，如果大家在面试中，看看而言ConcurrentHashMap，再结合在纸上演边画，那一定能征服面试官。
- 3、可以看下静态代理和动态代理的实现方式，再深入一下，可以看下Spring AOP里的实现代码。
- 4、或许Spring IOC和MVC的底层实现代码比较难看懂，但大家可以说些关键的类，根据关键流程说下它们的实现方式。

其实准备的底层代码未必要多，而且也不限于在哪个方面，比如集合里基于红黑树的TreeSet，基于NIO的开源框架，甚至分布式组件的Dubbo，都可以准备。而且准备时未必要背出所有的底层（事实上很难做到），你只要能结合一些重要的类和方法，讲清楚思路即可（比如讲清楚HashMap如何通过hashCode快速定位）。

那么在面试时，如何找到个好机会说出你准备好的上述底层代码？在面试时，总会被问到集合，Spring MVC框架等相关知识点，你在回答时，顺便说一句，“我还了解这块的底层实现”，那么面试官一定会追问，那么你就可以说出来了。

不要小看这个对候选人的帮助，一旦你讲了，只要意思到位，那么最少能得到个“肯积极专业”的评价，如果描述很清楚，那么评价就会升级到“熟悉Java核心技能（或Spring MVC），且基本功扎实”。要知道，面试中，很少有人能讲清楚底层代码，所以你抛出了这个话题，哪怕最后没达到预期效果，面试官也不会由此对你降低评价。所以说，准备这块绝对是“有百利而无一害”的挣钱买卖。

7、一切的一切，把上述技能嵌入到你做过的项目里

在面试过程中，我经常会听到一些比较遗憾的回答，比如候选人对SQL优化技能讲得头头是道，但最后得知，这是他平时自学时掌握的，并没用在实际项目里。

当然这总比不说要好，所以我会写下“在平时自学过SQL优化技能”，但如果在项目里实践过，那么我就会写下“有实际数据库SQL优化的技能”。大家可以对比下两者的差别，一个是偏重理论，一个是直接能干活了。其实，很多场景里，我就不信在实际项目里一定没有实践过SQL优化技能。

从这个案例中，我想告诉大家的是，你之前费了千辛万苦（其实方法方向得到，也不用费太大精力）准备的很多技能和说辞，最后应该落实到你的实际项目里。

比如你有过在Linux日志里查询关键字排查问题的经验，在描述时你可以带一句，在之前的项目里我就这样干的。又如，你通过看底层代码，了解了TreeSet和HashSet的差别以及它们的适用范围，那么你就可以回想下你之前做的项目，是否有个场景仅仅适用于TreeSet？如果有，那么你就可以适当描述下项目的需求，然后说，通过读底层代码，我了解了两者的差别，而且在这个实际需求里，我就用了TreeSet，而且我还专门做了对比性试验，发现用TreeSet比HashSet要高xx个百分点。

请记得，“实践经验”一定比“理论经验”值钱，而且大多数你知道的理论上的经验，一定在你的项目里用过。所以，如果你仅仅让面试官感觉你只有“理论经验”，那就太亏了。

8、小结：本文更多讲述的准备面试的方法

本文给出的面试题并不多，但本文并没有打算给出太多的面试题。从本文里，大家更多看到的是面试官发现的诸多候选人的痛点。

本文的用意是让大家别再重蹈别人的覆辙，这还不算，本文还给出了不少准备面试的方法。你的能力或许比别人出众，但如果你准备面试的方式和别人差不多，或者就拿你在项目里干的活来说事，而没有归纳出你在项目中的亮点，那么面试官还真的会看扁你。

本文欢迎转载，不过请注明文章来源，如果可以，请同时给出本人写的两本书的链接Java Web轻量级开发面试教程
(<https://item.jd.com/12136095.html>) 和Java核心技术及面试指南 (<https://item.jd.com/12421187.html>)。

再次感谢大家读完本文。

【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 ↓



扫一扫上面的二维码图案，加我微信

推荐阅读

1. Java 9 ← 2017, 2019 → Java 13
2. Spring Boot 整合 Spring-cache
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

[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！

深入浅出 Java 中的 clone 克隆方法

Java后端 2019-10-08

点击上方 Java后端, 选择设为星标

优质文章, 及时送达

作者 | 张纪刚

链接 | blog.csdn.net/zhangjg_blog

Java中对象的创建

clone 顾名思义就是复制，在Java语言中，clone方法被对象调用，所以会复制对象。所谓的复制对象，首先要分配一个和源对象同样大小的空间，在这个空间中创建一个新的对象。

我们回顾一下：在java语言中，有几种方式可以创建对象呢？

1. 使用new操作符创建一个对象
2. 使用clone方法复制一个对象

那么这两种方式有什么相同和不同呢？

new操作符的本意是分配内存。程序执行到new操作符时，首先去看new操作符后面的类型，因为知道了类型，才能知道要分配多大的内存空间。分配完内存之后，再调用构造函数，填充对象的各个域，这一步叫做对象的初始化，构造方法返回后，一个对象创建完毕，可以把自己的引用（地址）发布到外部，在外部就可以使用这个引用操纵这个对象。

而clone在第一步是和new相似的，都是分配内存，调用clone方法时，分配的内存和源对象（即调用clone方法的对象）相同，然后再使用原对象中对应的各个域，填充新对象的域，填充完成之后，clone方法返回，一个新的相同的对象被创建，同样可以把这个新对象的引用发布到外部。微信搜索 web_resource 获取更多推送。

复制对象 or 复制引用

在Java中，以下类似的代码非常常见：

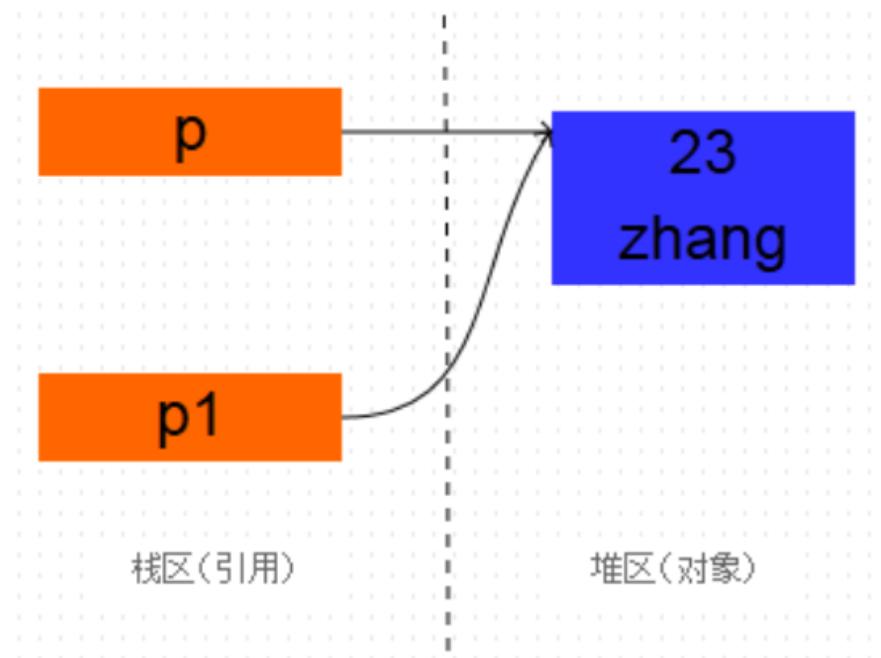
```
1 Person p = new Person(23, "zhang");
2 Person p1 = p;
3
4 System.out.println(p);
5 System.out.println(p1);
```

打印结果：

```
1 com.pansoft.zhangjg.testclone.Person@2f9ee1ac
2 com.pansoft.zhangjg.testclone.Person@2f9ee1ac
```

可以看出，打印的地址值是相同的，既然地址都是相同的，那么肯定是同一个对象。p和p1只是引用而已，他们都指向了一个相同的对象Person(23, "zhang")。可以把这种现象叫做引用的复制。

上面代码执行完成之后，内存中的情景如下图所示：



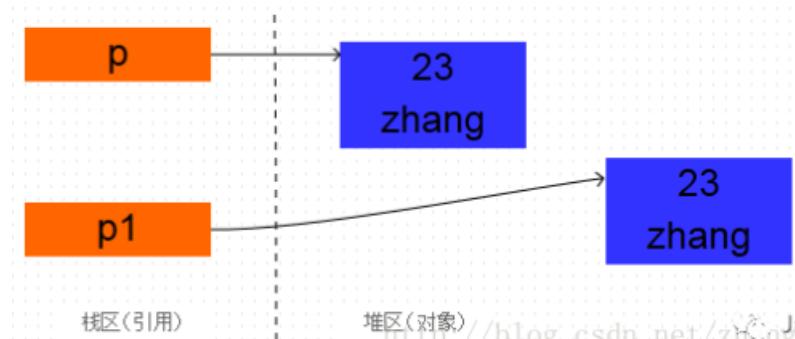
而下面的代码是真真正正的克隆了一个对象：

```
1 Person p = new Person(23, "zhang")
2 ;
3 Person p1 = (Person) p.clone();
4
5 System.out.println(p);
System.out.println(p1);
```

打印结果：

```
1 com.pansoft.zhangjg.testclone.Person@2f9ee1ac
2 com.pansoft.zhangjg.testclone.Person@67f1fba0
```

以上代码执行完成后，内存中的情景如下图所示：



深拷贝 or 浅拷贝

上面的示例代码中，Person中有两个成员变量，分别是name和age，name是String类型，age是int类型。代码非常简单，如下所示：

```
1 public class Person implements Cloneable{
2 }
```

```
3  private int age ;
4  private String name;
5
6  public Person(int age, String name)
7  {
8      this.age = age;
9      this.name = name;
10 }
11
12 public Person() {}
13
14 public int getAge()
15 {
16     return age;
17 }
18
19 public String getName()
20 {
21     return name;
22 }
23
24 @Override
25 protected Object clone() throws CloneNotSupportedException
{
    return (Person)super.clone();
}
```

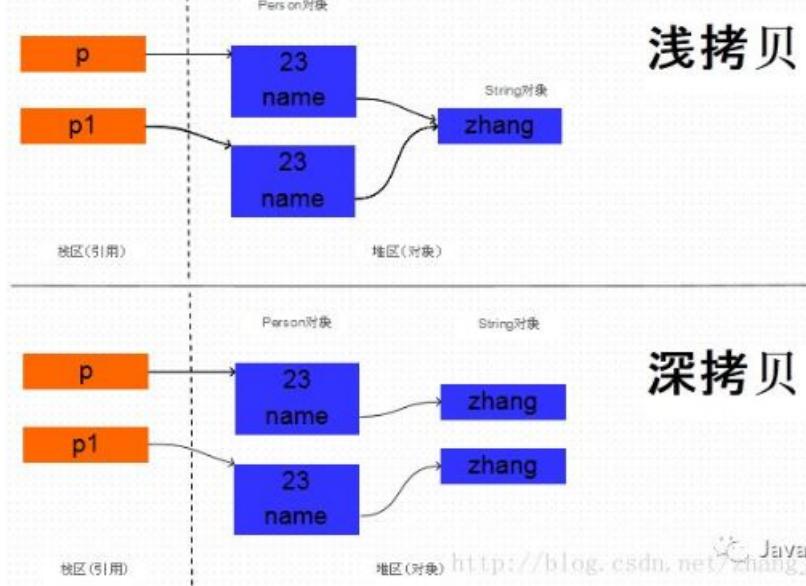
由于age是基本数据类型，那么对它的拷贝没有什么疑议，直接将一个4字节的整数值拷贝过来就行。但是name是String类型的，它只是一个引用，指向一个真正的String对象，那么对它的拷贝有两种方式：

1. 直接将源对象中的name的引用值拷贝给新对象的name字段；
2. 根据原Person对象中的name指向的字符串对象创建一个新的相同的字符串对象，将这个新字符串对象的引用赋给新拷贝的Person对象的name字段。

这两种拷贝方式分别叫做 **浅拷贝** 和 **深拷贝**。

深拷贝和浅拷贝的原理如下图所示：

浅拷贝



深拷贝

Java:

<http://blog.csdn.net/zhangj>

下面通过代码进行验证。

如果两个Person对象的name的地址值相同，说明两个对象的name都指向同一个String对象，也就是浅拷贝，而如果两个对象的name的地址值不同，那么就说明指向不同的String对象，也就是在拷贝Person对象的时候，同时拷贝了name引用的String对象，也就是深拷贝。验证代码如下：

```
1 Person p = new Person(23, "zhang")
2 ;
3 Person p1 = (Person) p.clone();
4
5 String result = p.getName() == p1.getName()
6     ? "clone是浅拷贝的" : "clone是深拷贝的"
7 ;

System.out.println(result);
```

打印结果：

```
1 clone是浅拷贝的
```

所以，clone方法执行的是浅拷贝，在编写程序时要注意这个细节。

如果想要实现深拷贝，可以通过覆盖Object中的clone方法的方式。微信搜索 web_resource 获取更多推送。

现在为了要在clone对象时进行深拷贝，那么就要Clonable接口，覆盖并实现clone方法，除了调用父类中的clone方法得到新的对象，还要将该类中的引用变量也clone出来。如果只是用Object中默认的clone方法，是浅拷贝的，再次以下面的代码验证：

```
1 static class Body implements Cloneable{
2     public Head head;
3
4     public Body() {}
5
6     public Body(Head head) {this.head = head;}
7
8     @Override
9     protected Object clone() throws CloneNotSupportedException
```

```

10 {
11     return super.clone();
12 }
13 }
14 }
15 static class Head /*implements Cloneable*/{
16     public Face face;
17 }
18 public Head() {}
19 public Head(Face face){this.face = face;}
20 }
21 }
22 public static void main(String[] args) throws CloneNotSupportedException
23 {
24
25     Body body = new Body(new Head());
26
27     Body body1 = (Body) body.clone();
28
29     System.out.println("body == body1 : " + (body == body1));
30
31     System.out.println("body.head == body1.head : " + (body.head == body1.head));
32
33 }

```

在以上代码中，有两个主要的类，分别为Body和Face，在Body类中，组合了一个Face对象。当对Body对象进行clone时，它组合的Face对象只进行浅拷贝。打印结果可以验证该结论：

```

1 body == body1 : false
2 body.head == body1.head : true

```

如果要使Body对象在clone时进行深拷贝，那么就要在Body的clone方法中，将源对象引用的Head对象也clone一份。微信搜索web_resource 获取更多推送。

```

1 static class Body implements Cloneable
2 {
3     public Head head;
4     public Body() {
5     }
6     public Body(Head head) {this.head = head;}
7
8     @Override
9     protected Object clone() throws CloneNotSupportedException
10    {
11        Body newBody = (Body) super.clone();
12        newBody.head = (Head) head.clone();
13        return newBody;
14    }
15
16 }

```

```
17 static class Head implements Cloneable
18 {
19     public Face face;
20
21     public Head() {
22 }
23     public Head(Face face){this.face = face;}
24     @Override
25     protected Object clone() throws CloneNotSupportedException
26     {
27         return super.clone();
28     }
29 }
30 public static void main(String[] args) throws CloneNotSupportedException
31 {
32     Body body = new Body(new Head());
33
34     Body body1 = (Body) body.clone();
35
36     System.out.println("body == body1 : " + (body == body1));
37
38     System.out.println("body.head == body1.head : " + (body.head == body1.head));
39 }
```

打印结果:

```
1 body == body1 : false  
2 body.head == body1.head : false
```

由此可见，`body`和`body1`内的`head`引用指向了不同的`Head`对象，也就是说在`clone Body`对象的同时，也拷贝了它所引用的`Head`对象，进行了深拷贝。

真的是深拷贝吗

通过上面的讲解我们已经知道：如果想要深拷贝一个对象，这个对象必须要实现Cloneable接口，实现clone方法，并且在clone方法内部，把该对象引用的其他对象也要clone一份，这就要求这个被引用的对象必须也要实现Cloneable接口并且实现clone方法。微信搜索 web_resource 获取更多推送。

那么，按照上面的结论，Body类组合了Head类，而Head类组合了Face类，要想深拷贝Body类，必须在Body类的clone方法中将Head类也要拷贝一份，但是在拷贝Head类时，默认执行的是浅拷贝，也就是说Head中组合的Face对象并不会被拷贝。

验证代码如下：（这里本来只给出Face类的代码就可以了，但是为了阅读起来具有连贯性，避免丢失上下文信息，还是给出整个程序，整个程序也非常简短）

```
1     static class Body implements Cloneable
2     {
3         public Head head;
4         public Body() {
```

```

4 }
5     public Body(Head head) {this.head = head;}
6
7     @Override
8     protected Object clone() throws CloneNotSupportedException
9 {
10     Body newBody = (Body) super.clone();
11     newBody.head = (Head) head.clone();
12     return newBody;
13 }
14
15 }
16
17 static class Head implements Cloneable
18 {
19     public Face face;
20
21     public Head() {
22 }
23     public Head(Face face){this.face = face;}
24     @Override
25     protected Object clone() throws CloneNotSupportedException
26 {
27     return super.clone();
28 }
29 }
30
31 static class Face{}
32
33 public static void main(String[] args) throws CloneNotSupportedException
34 {
35
    Body body = new Body(new Head(new Face()));
    Body body1 = (Body) body.clone();
    System.out.println("body == body1 : " + (body == body1));
    System.out.println("body.head == body1.head : " + (body.head == body1.head));
    System.out.println("body.head.face == body1.head.face : " + (body.head.face == body1.head.face));
}

```

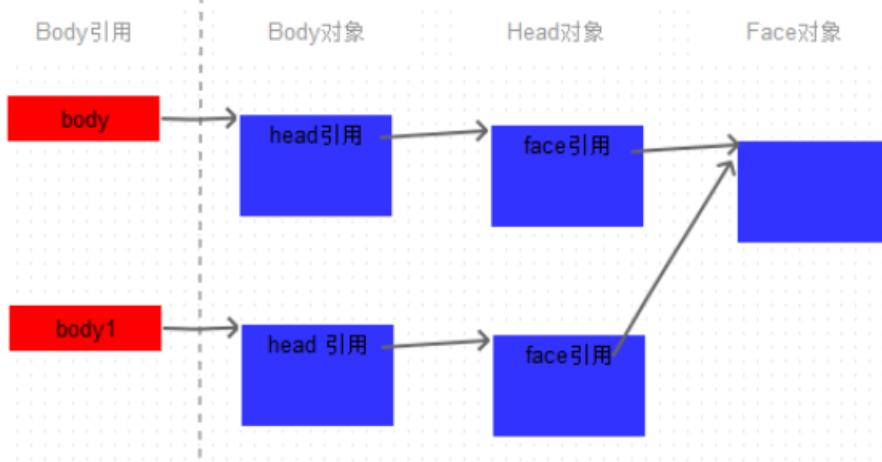
打印结果:

```

1 body == body1 : false
2 body.head == body1.head : false
3 body.head.face == body1.head.face : true

```

内存结构图如下图所示:



那么，对Body对象来说，算是这算是深拷贝吗？其实应该算是深拷贝，因为对Body对象内所引用的其他对象（目前只有Head）都进行了拷贝，也就是说两个独立的Body对象内的head引用已经指向了独立的两个Head对象。微信搜索 web_resource 获取更多推送。

但是，这对于两个Head对象来说，他们指向了同一个Face对象，这就说明，两个Body对象还是有一定的联系，并没有完全的独立。这应该说是一种 **不彻底的深拷贝**

如何进行彻底的深拷贝

对于上面的例子来说，怎样才能保证两个Body对象完全独立呢？只要在拷贝Head对象的时候，也将Face对象拷贝一份就可以了。这需要让Face类也实现Cloneable接口，实现clone方法，并且在在Head对象的clone方法中，拷贝它所引用的Face对象。修改的部分代码如下：

```

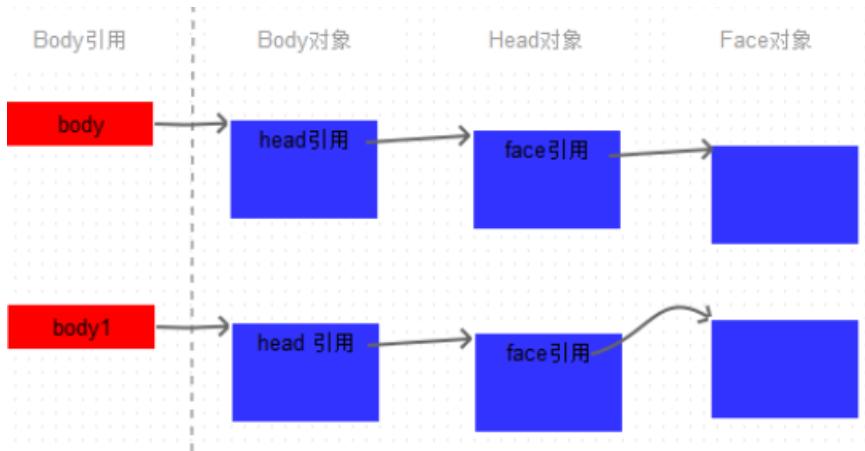
1  static class Head implements Cloneable
2  {
3      public Face face;
4
5      public Head() {
6      }
7
8      public Head(Face face){this.face = face;}
9
10     @Override
11     protected Object clone() throws CloneNotSupportedException
12     {
13         //return super.clone();
14         Head newHead = (Head) super.clone();
15         newHead.face = (Face) this.face.clone();
16         return newHead;
17     }
18
19     static class Face implements Cloneable
20     {
21         @Override
22         protected Object clone() throws CloneNotSupportedException
23         {
24             return super.clone();
25         }
26     }

```

再次运行上面的示例，得到的运行结果如下：

```
1 body == body1 : false
2 body.head == body1.head : false
3 body.head.face == body1.head.face : false
```

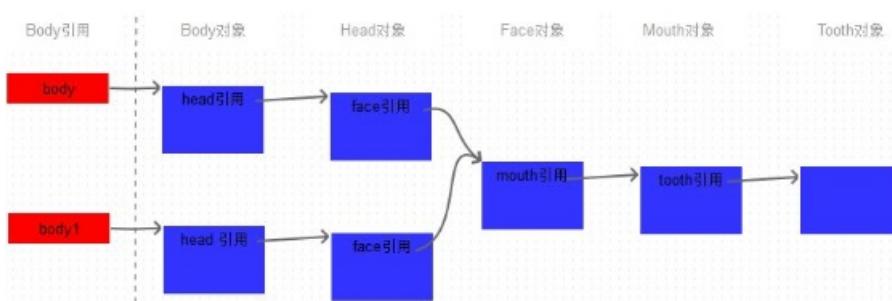
这说明两个Body已经完全独立了，他们间接引用的face对象已经被拷贝，也就是引用了独立的Face对象。内存结构图如下：



依此类推，如果Face对象还引用了其他的对象，比如说Mouth，如果不经过处理，Body对象拷贝之后还是会通过一级一级的引用，引用到同一个Mouth对象。同理，如果要让Body在引用链上完全独立，只能显式的让Mouth对象也被拷贝。

到此，可以得到如下结论：如果在拷贝一个对象时，要想让这个拷贝的对象和源对象完全彼此独立，那么在引用链上的每一级对象都要被显式的拷贝。所以创建彻底的深拷贝是非常麻烦的，尤其是在引用关系非常复杂的情况下，或者在引用链的某一级上引用了一个第三方的对象，而这个对象没有实现clone方法，那么在它之后的所有引用的对象都是被共享的。

举例来说，如果被Head引用的Face类是第三方库中的类，并且没有实现Cloneable接口，那么在Face之后的所有对象都会被拷贝前后的两个Body对象共同引用。假设Face对象内部组合了Mouth对象，并且Mouth对象内部组合了Tooth对象，内存结构如下图：



写在最后

clone在平时项目的开发中可能用的不是很频繁，但是区分深拷贝和浅拷贝会让我们对java内存结构和运行方式有更深的了解。至于彻底深拷贝，几乎是不可能实现的，原因已经在上一节中进行了说明。

深拷贝和彻底深拷贝，在创建不可变对象时，可能对程序有着微妙的影响，可能会决定我们创建的不可变对象是不是真的不可变。clone的一个重要的应用也是用于不可变对象的创建。关于创建不可变对象，我会在后续的文章中进行阐述，敬请期待。

- 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. JDK 13 新特性一览
3. 14 个实用的数据库设计技巧
4. 面试官：Redis 内存满了怎么办？
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 

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！

牛逼哄哄的 Java 8 Stream，性能如何？

CarpenterLee Java后端 2019-11-17

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

Java8的Stream API可以极大提高Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

那么，Stream API的性能到底如何呢，代码整洁的背后是否意味着性能的损耗呢？本文对Stream API的性能一探究竟。

为保证测试结果真实可信，我们将JVM运行在 `-server` 模式下，测试数据在GB量级，测试机器采用常见的商用服务器，配置如下：

OS	CentOS 6.7 x86_64
CPU	Intel Xeon X5675, 12M Cache 3.06 GHz, 6 Cores 12 Threads
内存	96GB
JDK	java version 1.8.0_91, Java HotSpot(TM) 64-Bit Server VM

测试所用代码在这里

<https://github.com/CarpenterLee/JavaLambdaInternals/blob/master/perf/StreamBenchmark/src/lee>

测试结果汇总

https://github.com/CarpenterLee/JavaLambdaInternals/blob/master/perf/Stream_performance.xlsx

测试方法和测试数据

性能测试并不是容易的事，Java性能测试更费劲，因为虚拟机对性能的影响很大，JVM对性能的影响有两方面：

1、GC的影响。GC的行为是Java中很不好控制的一块，为增加确定性，我们手动指定使用CMS收集器，并使用10GB固定大小的堆内存。集体到JVM参数就是 `-XX:+UseConcMarkSweepGC-Xms10G-Xmx10G`

2、JIT(Just-In-Time)即时编译技术。即时编译技术会将热点代码在JVM运行的过程中编译成本地代码，测试时我们会先对程序预热，触发对测试函数的即时编译。相关的JVM参数是 `-XX:CompileThreshold=10000`。

Stream并行执行时用到 `ForkJoinPool.commonPool()` 得到的线程池，为控制并行度我们使用Linux的 `taskset` 命令指定JVM可用的核数。

测试数据由程序随机生成。为防止一次测试带来的抖动，测试4次求出平均时间作为运行时间。

实验一 基本类型迭代

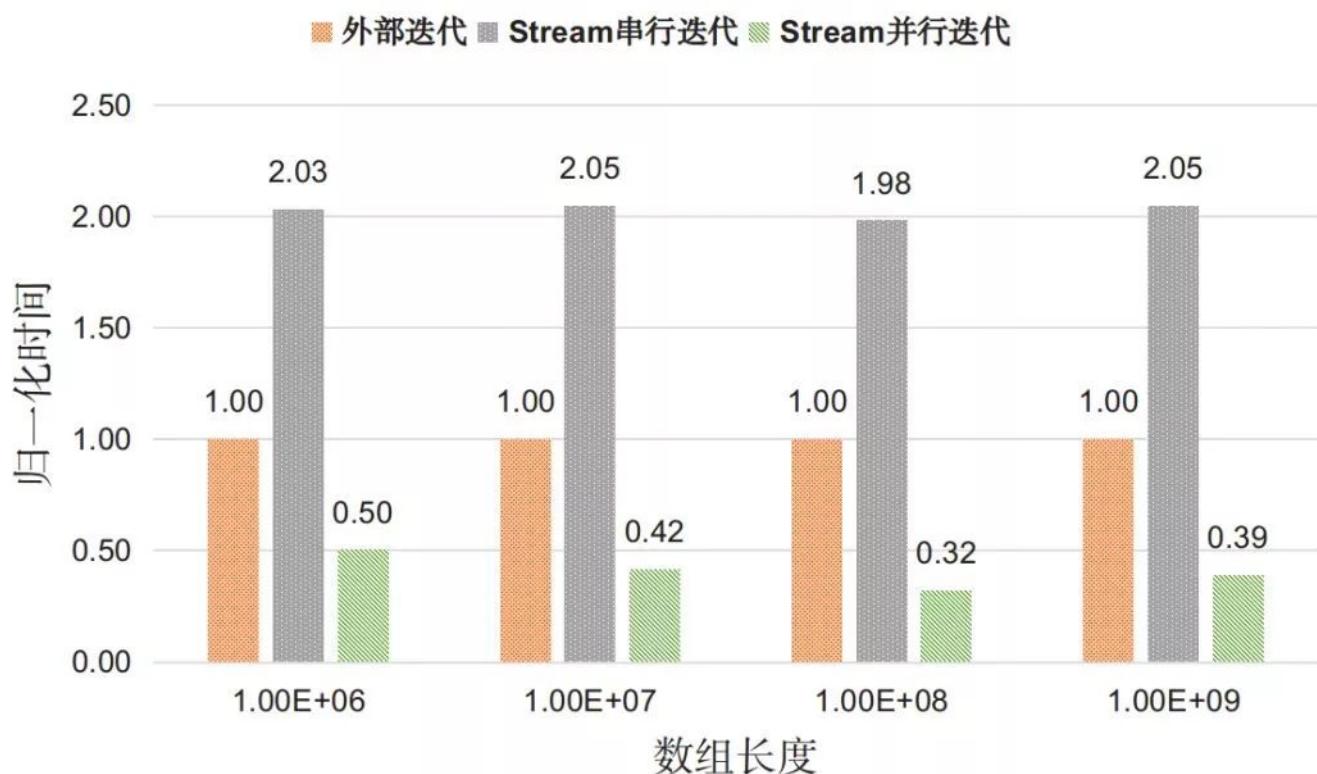
测试内容：找出整型数组中的最小值。对比for循环外部迭代和Stream API内部迭代性能。

测试程序 `IntTest`

<https://github.com/CarpenterLee/JavaLambdaInternals/blob/master/perf/StreamBenchmark/src/lee/IntTest.java>

测试结果如下图：

Stream求int数组最小值性能

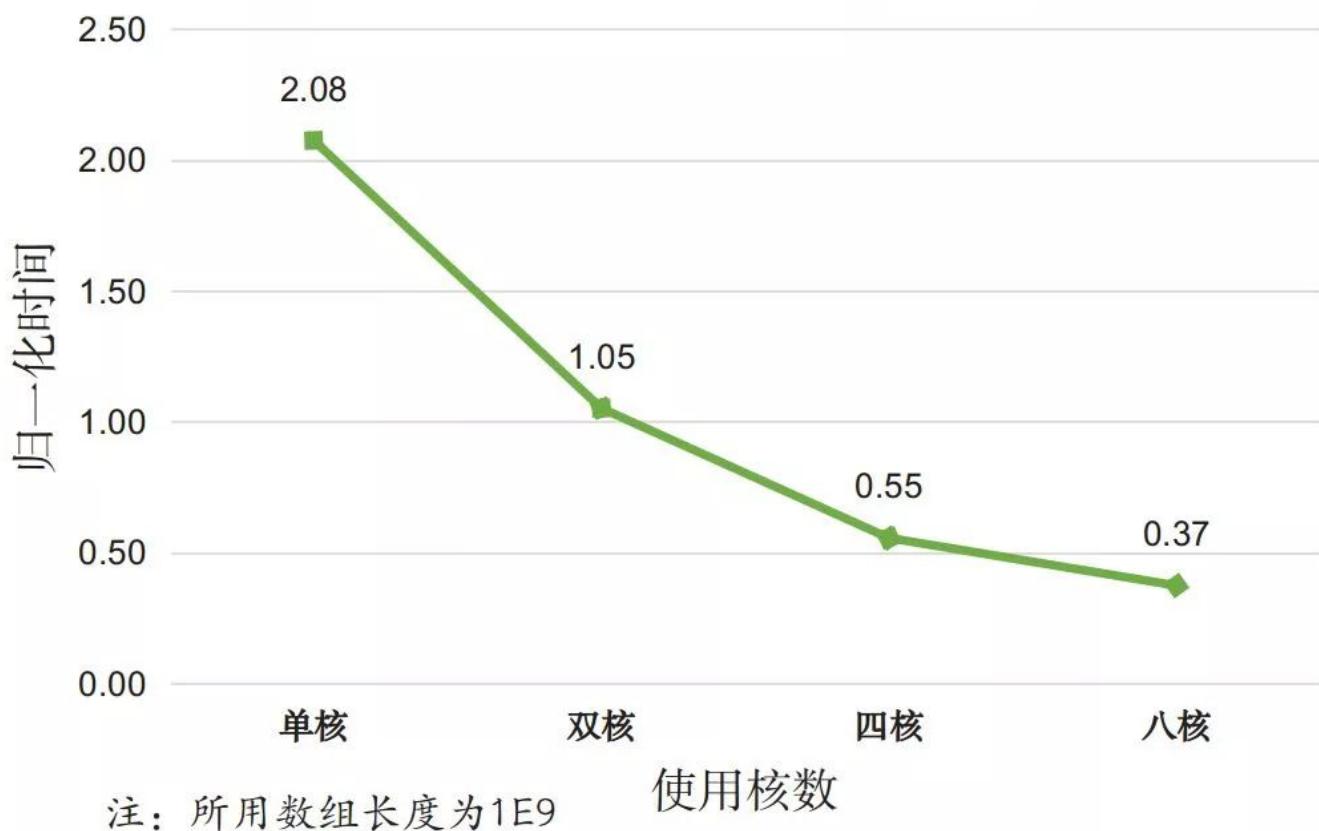


图中展示的是for循环外部迭代耗时为基准的时间比值。分析如下：

- 1、对于基本类型Stream串行迭代的性能开销明显高于外部迭代开销（两倍）；
- 2、Stream并行迭代的性能比串行迭代和外部迭代都好。

并行迭代性能跟可利用的核数有关，上图中的并行迭代使用了全部12个核，为考察使用核数对性能的影响，我们专门测试了不同核数下的Stream并行迭代效果：

Stream并行求int数组最小值性能



分析，对于基本类型：

- 1、使用Stream并行API在单核情况下性能很差，比Stream串行API的性能还差；
- 2、随着使用核数的增加，Stream并行效果逐渐变好，比使用for循环外部迭代的性能还好。

以上两个测试说明，对于基本类型的简单迭代，Stream串行迭代性能更差，但多核情况下Stream迭代时性能较好。

实验二 对象迭代

再来看对象的迭代效果。

测试内容：找出字符串列表中最小的元素（自然顺序），对比for循环外部迭代和Stream API内部迭代性能。

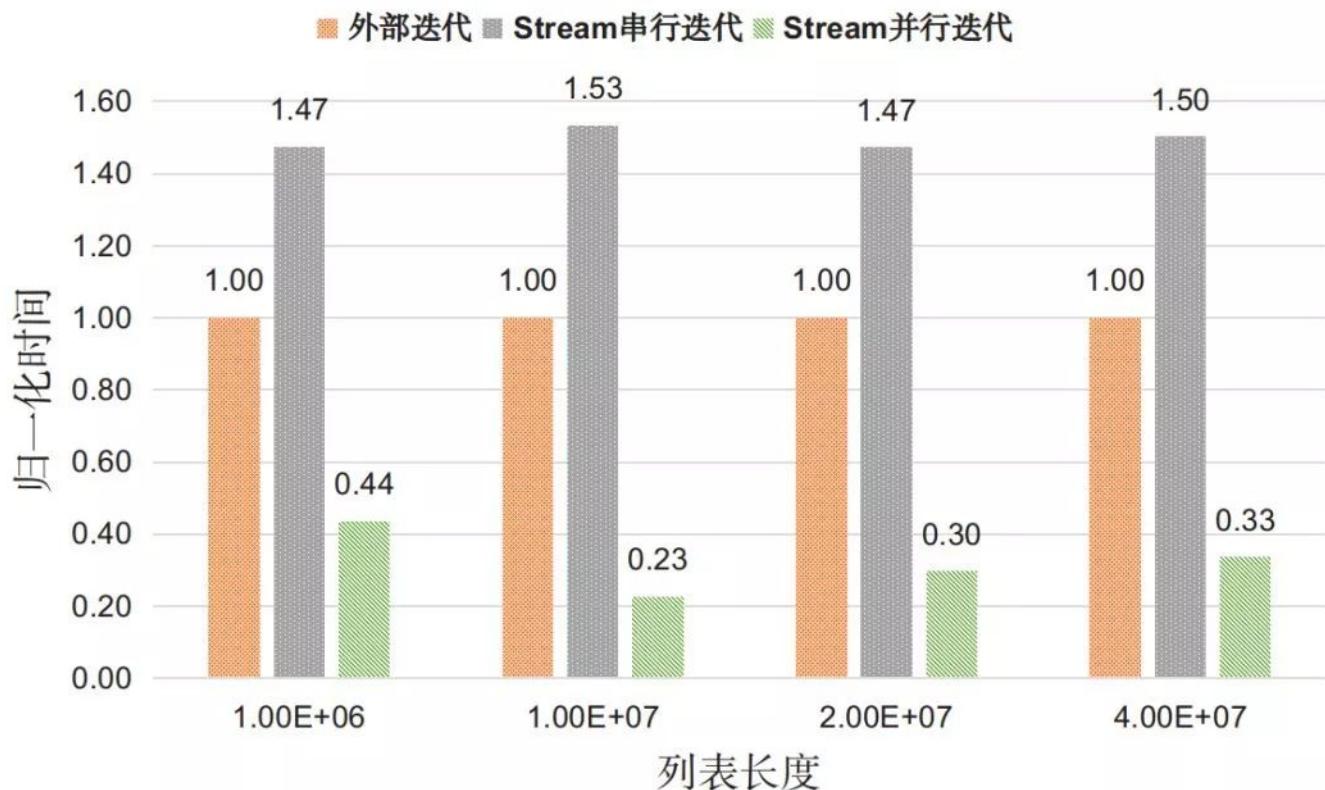
测试程序StringTest

<https://github.com/CarpenterLee/JavaLambdaInternals/blob/master/perf/StreamBenchmark/src/lee/StringTest.java>

对 Stream 不熟悉的，可以关注微信公众号：Java技术栈，在后台回复：Java。

测试结果如下图：

Stream求String列表最小值性能

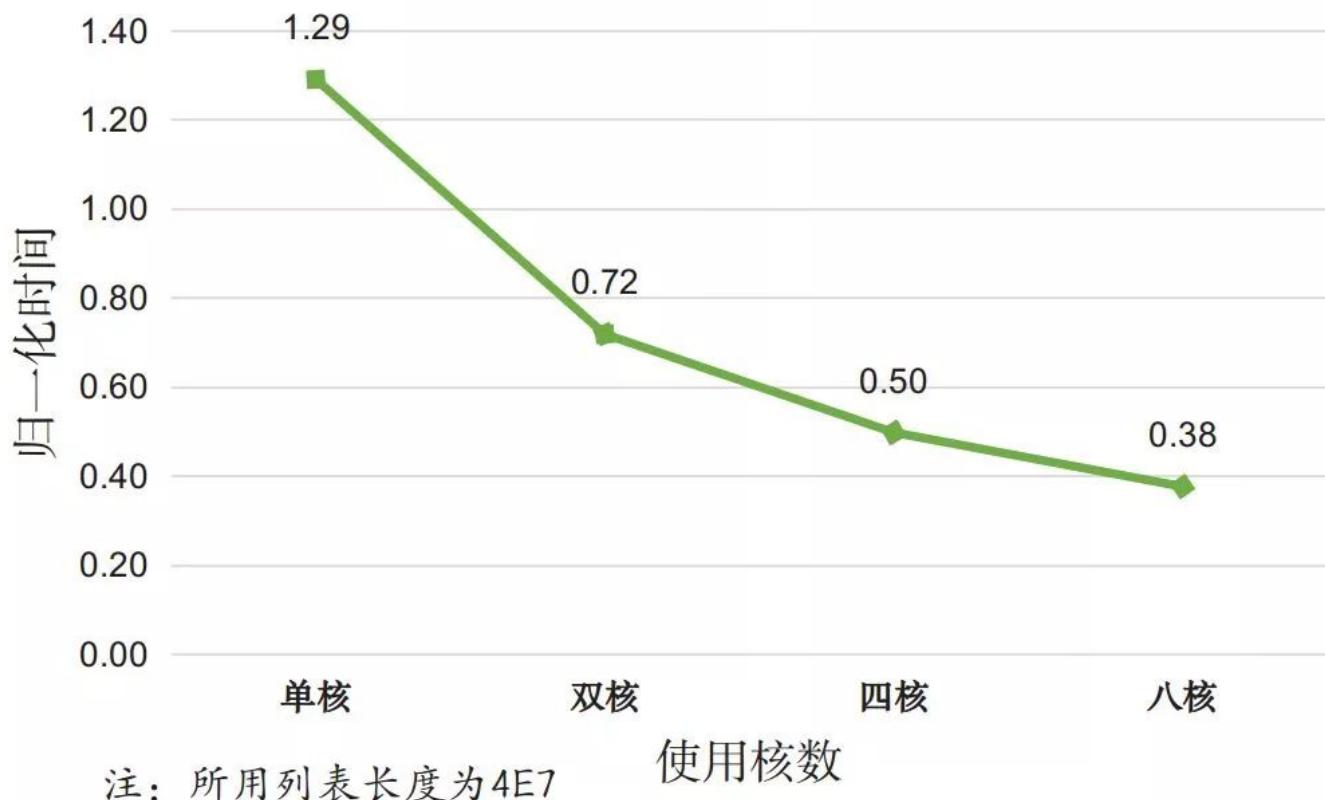


结果分析如下：

- 1、对于对象类型Stream串行迭代的性能开销仍然高于外部迭代开销（1.5倍），但差距没有基本类型那么大。
- 2、Stream并行迭代的性能比串行迭代和外部迭代都好。

再来单独考察Stream并行迭代效果：

Stream并行求String列表最小值性能



分析，对于对象类型：

- 1、使用Stream并行API在单核情况下性能比for循环外部迭代差；
- 2、随着使用核数的增加，Stream并行效果逐渐变好，多核带来的效果明显。

以上两个测试说明，对于对象类型的简单迭代，Stream串行迭代性能更差，但多核情况下Stream迭代时性能较好。

实验三 复杂对象归约

从实验一、二的结果来看，Stream串行执行的效果都比外部迭代差（很多），是不是说明Stream真的不行了？先别下结论，我们再来考察一下更复杂的操作。

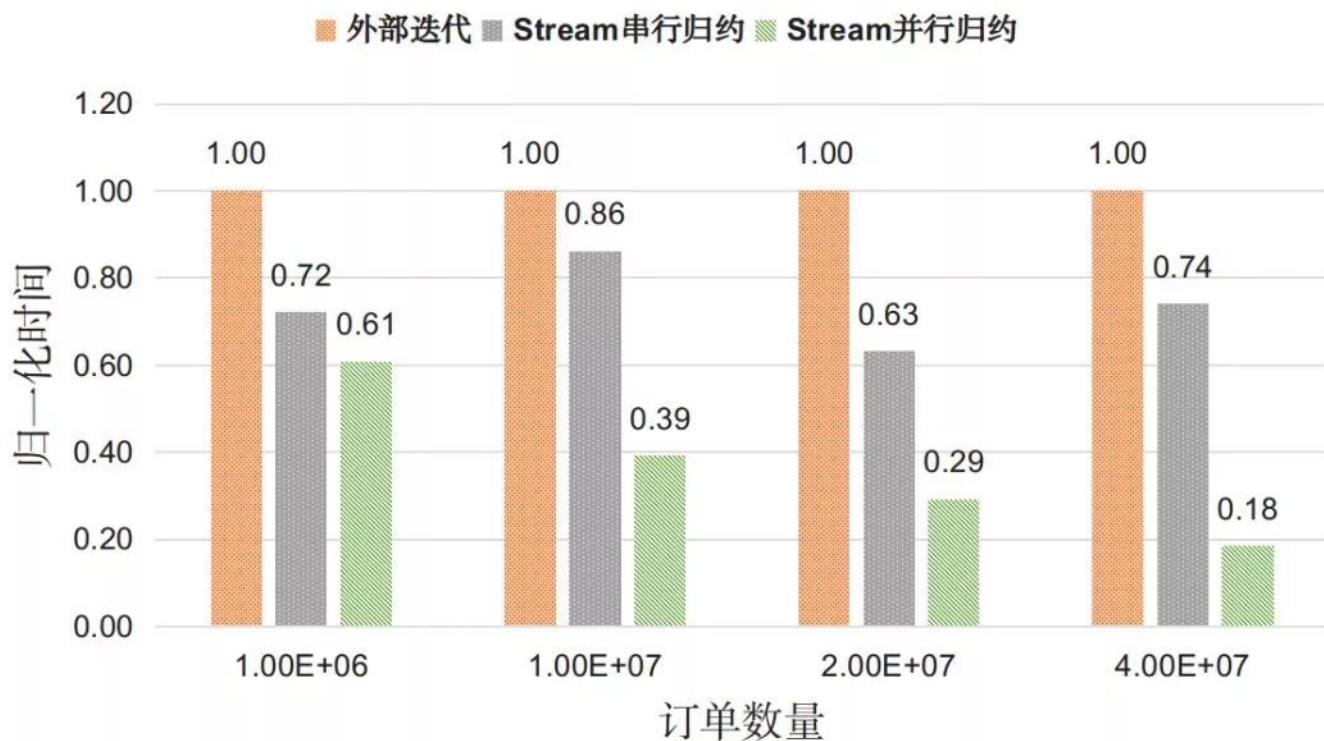
测试内容：给定订单列表，统计每个用户的总交易额。对比使用外部迭代手动实现和Stream API之间的性能。

我们将订单简化为 `<userName,price,timeStamp>` 构成的元组，并用 `Order` 对象来表示。测试程序 `ReductionTest`

<https://github.com/CarpenterLee/JavaLambdaInternals/blob/master/perf/StreamBenchmark/src/lee/ReductionTest.java>

测试结果如下图：

Stream归约操作性能



分析，对于复杂的归约操作：

- 1、Stream API的性能普遍好于外部手动迭代，并行Stream效果更佳；

再来考察并行度对并行效果的影响，测试结果如下：



分析，对于复杂的归约操作：

- 1、使用Stream并行归约在单核情况下性能比串行归约以及手动归约都要差，简单说就是最差的；

2、随着使用核数的增加，Stream并行效果逐渐变好，多核带来的效果明显。

以上两个实验说明，对于复杂的归约操作，Stream串行归约效果好于手动归约，在多核情况下，并行归约效果更佳。我们有理由相信，对于其他复杂的操作，Stream API也能表现出相似的性能表现。

结论

上述三个实验的结果可以总结如下：

- 1、对于简单操作，比如最简单的遍历，Stream串行API性能明显差于显示迭代，但并行的Stream API能够发挥多核特性。
- 2、对于复杂操作，Stream串行API性能可以和手动实现的效果匹敌，在并行执行时Stream API效果远超手动实现。

所以，如果出于性能考虑

- 1、对于简单操作推荐使用外部迭代手动实现
- 2、对于复杂操作，推荐使用Stream API
- 3、在多核情况下，推荐使用并行Stream API来发挥多核优势，
- 4、单核情况下不建议使用并行Stream API。

如果出于代码简洁性考虑，使用Stream API能够写出更短的代码。即使是从性能方面说，尽可能的使用Stream API也另外一个优势，那就是只要Java Stream类库做了升级优化，代码不用做任何修改就能享受到升级带来的好处。

作者：CarpenterLee

来源：<https://dwz.cn/pSW0u0Qr>

-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. 你们心心念念的 GitHub 客户端终于来了！

2. Redis 实现「附近的人」这个功能

3. 一个秒杀系统的设计思考

4. 零基础认识 Spring Boot

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！

用大白话告诉你：Java 后端到底是在做什么？

Java后端 2019-09-13

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

来源: 公众号「黄小斜」

新手程序员通常会走入一个误区，就是认为学习了一门语言，就可以称为是某某语言工程师了。但事实上真的是这样吗？其实并非如此。

今天我们就来聊一聊，Java 开发工程师到底开发的是什么东西。准确点来说，Java后端到底在做什么？

大家都知道 Java 是一门后端语言，后端指的就是服务端，服务端代码一般运行在服务器上，通常我们运行Java 程序的服务器都是 Linux 服务器。

这些服务器在互联网公司中一般放在一个叫做机房的地方里，于是像我们这类 Java 程序员的代码一般也运行在这些机房里的服务器中。



Java 里有一个概念叫做虚拟机，你可以把它理解为一个安卓的模拟器，比如你在电脑上装了一个安卓模拟器，就可以通过它来运行安卓应用程序，比如装个 APP，手机游戏什么的。

所以当你在电脑上安装了一个叫做 JDK 的东西时，电脑里就有了 JRE 也就是 Java 运行环境，有了这个运行环境，你就可以运行 Java 应用程序了。

知道 Java 程序如何运行在计算机上之后，我们再来讲一讲平时学的一些 Java 基础知识，它们到底有什么用？

其实平时这一些 Java 基础语法都仅仅是你写代码的一些基础知识，就相当于英语中的 26 个字母，常见的有基本类型变量、for 循环、

if else 等等基本语法，掌握了这些基础知识之后，你就可以上手写一些很简单的代码了。

除此之外，Java 还有一些比较特别的概念，比如面向对象的特性，其中有类、接口等概念。为什么 Java 要引入这些东西呢，其实就想让使用者更好地进行设计、抽象和编程。

对于新手来说，你不需要理解得特别的深刻，因为这些东西只有你在真正写代码之后才能逐步去理解。



说完基本知识之后，我想你也会好奇，Java里经常提到的一些集合类是干嘛的呢，因为在现实生活中有很多场景，需要用到集合类，比如说一个用户名列表，你要怎么存呢？

你会用一个 List 来做对不对，所以集合类的作用就是让你在编程中更好的存储数据。

事实上，集合类的概念最早是来源于数据结构的，因为计算机里有很多特殊的数据存储结构，比如文件树，比如链表和数组等结构，因此计算机理论把这些存储数据的模型抽象成一些常见的结构，统称为数据结构。

那么，Java 中的并发编程又是做什么的呢，Java 中的多线程是为了更好地利用电脑中的CPU核心，通过并发编程，就可以提高程序并发的效率。

但是并发编程的背后需要操作系统的支持，以及计算机硬件的支持，所以，如果你要完全地理解多线程，绝不仅仅是理解 Java 里的 Thread 或者是线程池就足够了，你还需要去理解操作系统，以及计算机组原理。



和并发编程类似，Java 里也有网络编程的概念，Java 里的网络编程和其他语言大同小异，其实也是基于 TCP/IP 协议实现的一套 API，通过网络编程，你就可以在程序中把你想要传输的数据传输到网络的另一端，有了网络编程和并发编程之后，Java 程序员的能量已经很大了

讲完这几点之后接下来再谈谈，我们通常说的 Java 后端技术到底是什么，就拿支付宝来举例吧，曾经的支付宝用户数并不多，一台服务器，一个数据库就可以支持所有的业务了。

当支付宝的用户越来越多的时候，一台服务器无法同时满足海量用户的需求，于是开始出现了多台服务器，多台服务器组成了一个集群，用户可以通过负载均衡的方式访问这些服务器，每个用户可能会访问到不同的机器上，这样子就达到了分流的效果，服务器的压力就会减小。

由于数据库需要保证数据的可靠性，万一某一台数据库挂了，并且没有备份的话，那么这个数据就无法访问了，这在大型系统中是不允许出现的，于是乎，就有了数据库的主从部署。

但事实上，随着业务发展，数据库的压力也越来越大，主备部署并不能解决数据库访问性能的问题，于是乎我们需要进行分库分表，在数据库主备的基础上，我们会把一个数据量很大的表拆成多个表，并且把数据库请求分流到不同的数据上，比如说100个分库，100个分表，就相当于把一个数据表划分成10000个数据表。

此时又出现一个问题，如果一个数据库有多个备库，并且当主库挂掉的时候需要进行主从切换时，主备数据库之间的数据就可能发生不一致，而这也是分布式理论研究的问题之一，因为比较复杂，我们这里就略过不讲。



刚才说到了分布式技术，其实负载均衡、分库分表都是分布式技术的一种实现，如果你不想做分库分表，那还有什么办法能够减轻数据库访问的压力呢？于是缓存就出现了，缓存可以让服务器先把请求打到缓存上，由于缓存的数据一般在内存中，所以访问速度会非常快，这些请求无需经过数据库。

随着业务发展，缓存的单点压力也会比较大，于是平分布式缓存就出现了，通常来说，缓存难以保证数据的可靠性，因为它们的数据可能会丢失，同时缓存只能存储一部分的数据，并不能解决所有问题。

所以当某些业务的请求量非常大的时候，光靠缓存也解决不了问题，此时我们还可以通过消息队列来帮我们解决大流量并发请求的问题。

我们可以通过消息队列来存储一部分的请求消息，然后根据我们服务器处理请求的能力，把消息再逐步取出来，接着去把这些消息逐渐地进行处理，这样就可以很好的解决高并发的问题。当然，前提是消息队列要保证消息存储的可靠性，这也是大部分消息队列都会保证的能力。



一口气讲了这么多，算是把 Java 后端的大概面貌介绍清楚了，除此之外还有很多东西没讲到，真要讲完的话一晚上也说不完。

总体来说，Java 后端技术，说难不难说简单也不简单，我尽量把这些内容都讲的比较通俗易懂，事实上每项技术的背后都有特别多复杂的实现原理，当然，在你理解了 Java 后端技术的整体概念以后，相信对于你之后的学习会更有帮助。

如果有哪里说错了，偷偷留言告诉我 😊

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. [Java 实现 QQ 登陆](#)
2. [在阿里干了 5 年，面试个小公司挂了…](#)
3. [如何写出让同事无法维护的代码？](#)
4. [史上最烂的项目：苦撑12年，600 多万行代码 …](#)



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！

科普：教你如何看懂 JavaGC 日志

张振伟的博客 [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 | 张振伟的博客

链接 | zhangzw.com/posts/20190417.html

JVM GC 相关的参数

```
-XX:+PrintGC 输出 GC 日志  
-XX:+PrintGCDetails 输出 GC 的详细日志  
-XX:+PrintGCTimeStamps 输出 GC 的时间戳（以基准时间的形式）  
-XX:+PrintGCDates 输出 GC 的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）  
-XX:+PrintHeapAtGC 在进行 GC 的前后打印出堆的信息  
-Xloggc:D:/gc.log 日志文件的输出路径
```

比如在某个应用中，配置：

```
-XX:+PrintGCDetails -XX:+PrintGCDates -Xloggc:D:/gc.log
```

启动后打印如下 GC 日志：

YongGC

```
2019-04-18T14:52:06.790+0800: 2.653: [GC (Allocation Failure) [PSYoungGen: 33280K->5113K(38400K)] 33280K->5848K(125952K), 0.0095764 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
```

含义：

```
2019-04-18T14:52:06.790+0800 (当前时间戳): 2.653 (应用启动基准时间): [GC (Allocation Failure) [PSYoungGen (表示 Young GC): 33280K (年轻代回收前大小) ->5113K (年轻代回收后大小) (38400K (年轻代总大小))] 33280K (整个堆回收前大小) ->5848K (整个堆回收后大小) (125952K (堆总大小)), 0.0095764 (耗时) secs] [Times: user=0.00 (用户耗时) sys=0.00 (系统耗时), real=0.01 (实际耗时) secs]
```

Full GC

```
2019-04-18T14:52:15.359+0800: 11.222: [Full GC (Metadata GC Threshold) [PSYoungGen: 6129K->0K(143360K)] [ParOldGen: 13088K->13236K(55808K)] 19218K->13236K(199168K), [Metaspace: 20856K->20856K(1069056K)], 0.1216713 secs] [Times: user=0.44 sys=0.02, real=0.12 secs]
```

含义：

```
2019-04-18T14:52:15.359+0800 (当前时间戳): 11.222 (应用启动基准时间): [Full GC (Metadata GC Threshold) [PSYoungGen: 6129K (年轻代回收前大小) ->0K (年轻代回收后大小) (143360K (年轻代总大小))] [ParOldGen: 13088K (老年代回收前大小) ->13236K (老年代回收后大小) (55808K (老年代总大小))] 19218K (整个堆回收前大小) ->13236K (整个堆回收后大小) (199168K (堆总大小)), [Metaspace: 20856K (持久代回收前大小) ->20856K (持久代回收后大小) (1069056K (持久代总大小))], 0.1216713 (耗时) secs] [Times: user=0.44 (用户耗时) sys=0.02 (系统耗时), real=0.12 (实际耗时) secs]
```

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 

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!

细说 Java 主流日志工具库

静默虚空 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 | 静默虚空

链接 | juejin.im/post/5c8f35bfe51d4545cc650567

在项目开发中, 为了跟踪代码的运行情况, 常常要使用日志来记录信息。在 Java 世界, 有很多的日志工具库来实现日志功能, 避免了我们重复造轮子。

我们先来逐一了解一下主流日志工具。

日志框架

java.util.logging (JUL)

JDK1.4 开始, 通过 `java.util.logging` 提供日志功能。

它能满足基本的日志需要, 但是功能没有 Log4j 强大, 而且使用范围也没有 Log4j 广泛。

Log4j

Log4j 是 apache 的一个开源项目, 创始人 Ceki Gulcu。

Log4j 应该说是 Java 领域资格最老, 应用最广的日志工具。从诞生之日到现在一直广受业界欢迎。

Log4j 是高度可配置的, 并可通过在运行时的外部文件配置。它根据记录的优先级别, 并提供机制, 以指示记录信息到许多的目的地, 诸如: 数据库, 文件, 控制台, UNIX 系统日志等。

Log4j 中有三个主要组成部分:

- loggers - 负责捕获记录信息。
- appenders - 负责发布日志信息, 以不同的首选目的地。
- layouts - 负责格式化不同风格的日志信息。

官网地址:

<http://logging.apache.org/log4j/2.x/>

Logback

Logback 是由 log4j 创始人 Ceki Gulcu 设计的又一个开源日记组件, 目标是替代 log4j。

logback 当前分成三个模块: `logback-core`、`logback-classic` 和 `logback-access`。

- `logback-core` - 是其它两个模块的基础模块。

- logback-classic - 是 log4j 的一个改良版本。此外 logback-classic 完整实现 SLF4J API 使你可以很方便地更换成其它日记系统如 log4j 或 JDK14 Logging。
- logback-access - 访问模块与 Servlet 容器集成提供通过 Http 来访问日记的功能。

官网地址:

<http://logback.qos.ch/>

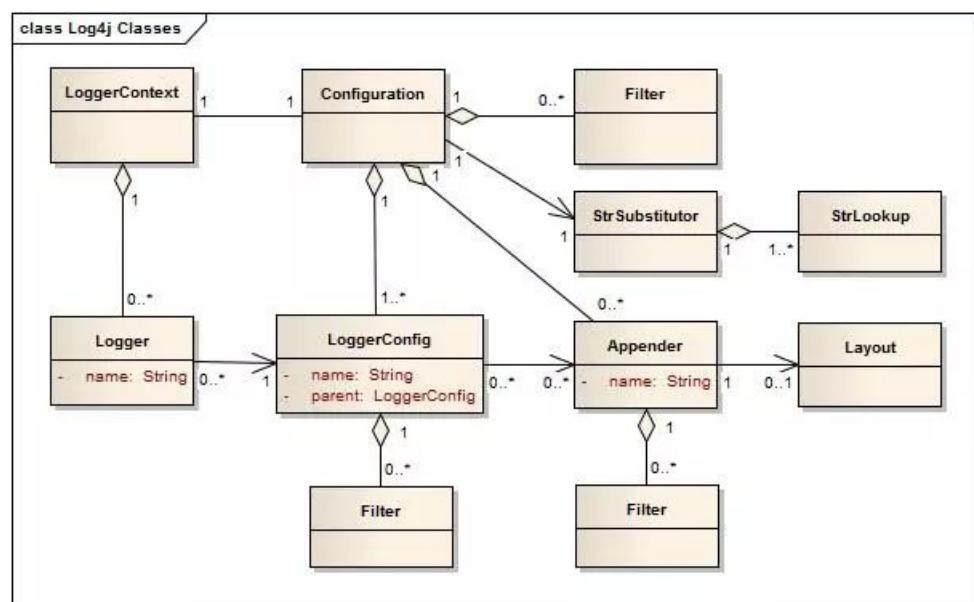
Log4j2

官网地址:

<http://logging.apache.org/log4j/2.x/>

按照官方的说法，Log4j2 是 Log4j 和 Logback 的替代。

Log4j2 架构:



Log4j vs Logback vs Log4j2

按照官方的说法，Log4j2 大大优于 Log4j 和 Logback。

那么，Log4j2 相比于先问世的 Log4j 和 Logback，它具有哪些优势呢？

- Log4j2 旨在用作审计日志记录框架。Log4j 1.x 和 Logback 都会在重新配置时丢失事件。Log4j 2 不会。在 Logback 中，Appender 中的异常永远不会对应用程序可见。在 Log4j 中，可以将 Appender 配置为允许异常渗透到应用程序。
- Log4j2 在多线程场景中，异步 Loggers 的吞吐量比 Log4j 1.x 和 Logback 高 10 倍，延迟低几个数量级。
- Log4j2 对于独立应用程序是无垃圾的，对于稳定状态日志记录期间的 Web 应用程序来说是低垃圾。这减少了垃圾收集器的压力，并且可以提供更好的响应时间性能。
- Log4j2 使用插件系统，通过添加新的 Appender、Filter、Layout、Lookup 和 Pattern Converter，可以非常轻松地扩展框架，而无需对 Log4j 进行任何更改。
- 由于插件系统配置更简单。配置中的条目不需要指定类名。

- 支持自定义日志等级。
- 支持 lambda 表达式。
- 支持消息对象。
- Log4j 和 Logback 的 Layout 返回的是字符串，而 Log4j2 返回的是二进制数组，这使得它能被各种 Appender 使用。
- Syslog Appender 支持 TCP 和 UDP 并且支持 BSD 系统日志。
- Log4j2 利用 Java5 并发特性，尽量小粒度的使用锁，减少锁的开销。

日志门面

何谓日志门面？

日志门面是对不同日志框架提供的一个 API 封装，可以在部署的时候不修改任何配置即可接入一种日志实现方案。

common-logging

common-logging 是 apache 的一个开源项目。也称 Jakarta Commons Logging，缩写 JCL。

common-logging 的功能是提供日志功能的 API 接口，本身并不提供日志的具体实现（当然，common-logging 内部有一个 Simple logger 的简单实现，但是功能很弱，直接忽略），而是在运行时动态的绑定日志实现组件来工作（如 log4j、java.util.logging）。

官网地址：

<http://commons.apache.org/proper/commons-logging/>

slf4j

全称为 Simple Logging Facade for Java，即 java 简单日志门面。

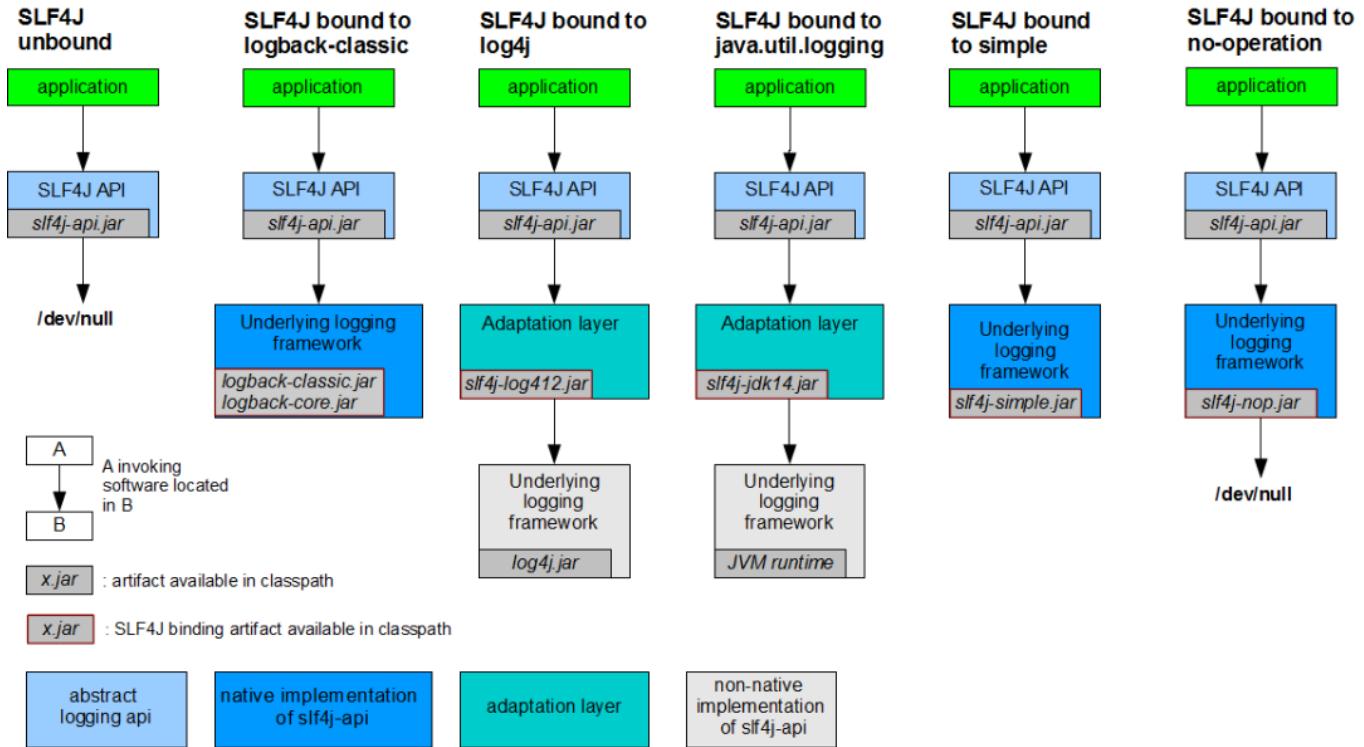
什么，作者又是 Ceki Gulcu！这位大神写了 Log4j、Logback 和 slf4j，专注日志组件开发五百年，一直只能超越自己。

类似于 Common-Logging，slf4j 是对不同日志框架提供的一个 API 封装，可以在部署的时候不修改任何配置即可接入一种日志实现方案。但是，slf4j 在编译时静态绑定真正的 Log 库。使用 SLF4J 时，如果你需要使用某一种日志实现，那么你必须选择正确的 SLF4J 的 jar 包的集合（各种桥接包）。

Tips：大家可以关注微信公众号：Java后端，获取更多推送。

官网地址：

<http://www.slf4j.org/>



common-logging vs slf4j

slf4j 库类似于 Apache Common-Logging。但是，他在编译时静态绑定真正的日志库。这点似乎很麻烦，其实也不过是导入桥接 jar 包而已。

slf4j 一大亮点是提供了更方便的日志记录方式：

不需要使用`logger.isDebugEnabled()`来解决日志因为字符拼接产生的性能问题。slf4j 的方式是使用`{}`作为字符串替换符，形式如下：

```
logger.debug("id: {}, name: {}", id, name);
```

总结

综上所述，使用 slf4j + Logback 可谓是目前最理想的日志解决方案了。

接下来，就是如何在项目中实施了。

实施日志解决方案

使用日志解决方案基本可分为三步：

- 引入 jar 包
- 配置
- 使用 API

常见的各种日志解决方案的第 2 步和第 3 步基本一样，实施上的差别主要在第 1 步，也就是使用不同的库。

引入 jar 包

这里首选推荐使用 slf4j + logback 的组合。

如果你习惯了 common-logging，可以选择 common-logging+log4j。

强烈建议不要直接使用日志实现组件(logback、log4j、java.util.logging)，理由前面也说过，就是无法灵活替换日志库。

还有一种情况：你的老项目使用了 common-logging，或是直接使用日志实现组件。如果修改老的代码，工作量太大，需要兼容处理。在下文，都将看到各种应对方法。

注：据我所知，当前仍没有方法可以将 slf4j 桥接到 common-logging。如果我孤陋寡闻了，请不吝赐教。

slf4j 直接绑定日志组件

slf4j + logback

添加依赖到 pom.xml 中即可。

logback-classic-1.0.13.jar 会自动将 slf4j-api-1.7.21.jar 和 logback-core-1.0.13.jar 也添加到你的项目中。

slf4j + log4j

添加依赖到 pom.xml 中即可。

slf4j-log4j12-1.7.21.jar 会自动将 slf4j-api-1.7.21.jar 和 log4j-1.2.17.jar 也添加到你的项目中。

```
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-log4j12</artifactId>
<version>1.7.21</version>
</dependency>
```

slf4j + java.util.logging

添加依赖到 pom.xml 中即可。

slf4j-jdk14-1.7.21.jar 会自动将 slf4j-api-1.7.21.jar 也添加到你的项目中。

```
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-jdk14</artifactId>
<version>1.7.21</version>
</dependency>
```

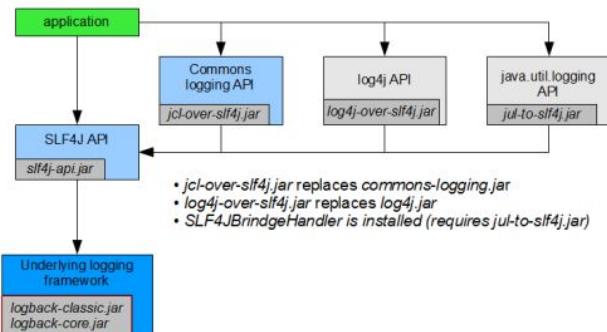
slf4j 兼容非 slf4j 日志组件

在介绍解决方案前，先提一个概念——桥接

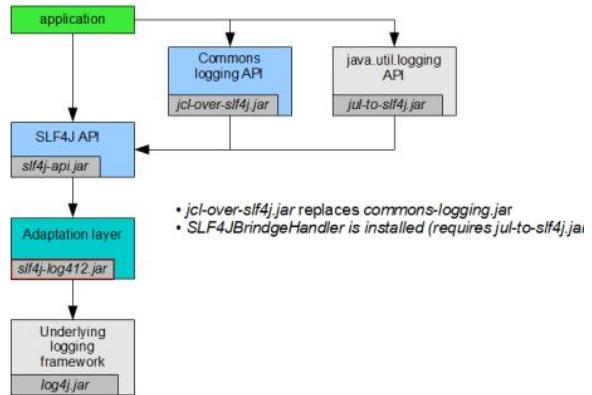
什么是桥接呢

假如你正在开发应用程序所调用的组件当中已经使用了 common-logging，这时你需要 jcl-over-slf4j.jar 把日志信息输出重定向到 slf4j-api，slf4j-api 再去调用 slf4j 实际依赖的日志组件。这个过程称为桥接。下图是官方的 slf4j 桥接策略图：

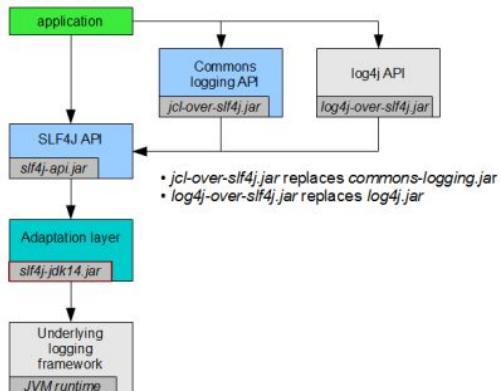
SLF4J bound to logback-classic with redirection of commons-logging, log4j and java.util.logging to SLF4J



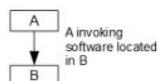
SLF4J bound to log4j with redirection of commons-logging and java.util.logging to SLF4J



SLF4J bound to java.util.logging with redirection of commons-logging and log4j to SLF4J



These diagrams illustrate all possible redirections for various bindings for reasons of convenience and expediency. Redirections should be performed only when necessary. For instance, it makes no sense to redirect java.util.logging to SLF4J if java.util.logging is not being used in your application.



X.jar : artifact available in classpath

X.jar SLF4J binding artifact available in classpath

abstract
logging api

native implementation
of slf4j-api

adaptation layer

non-native implementation
of slf4j-api

从图中应该可以看出，无论你的老项目中使用的是 common-logging 或是直接使用 log4j、java.util.logging，都可以使用对应的桥接 jar 包来解决兼容问题。

slf4j 兼容 common-logging

```
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>jcl-over-slf4j</artifactId>
<version>1.7.12</version>
</dependency>
```

slf4j 兼容 log4j

```
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>log4j-over-slf4j</artifactId>
<version>1.7.12</version>
</dependency>
```

slf4j 兼容 java.util.logging

```
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>jul-to-slf4j</artifactId>
<version>1.7.12</version>
</dependency>
```

spring 集成 slf4j

做 java web 开发，基本离不开 spring 框架。很遗憾，spring 使用的日志解决方案是 common-logging + log4j。(系统的学习

所以，你需要一个桥接 jar 包：logback-ext-spring。

```
<dependency>
<groupId>ch.qos.logback</groupId>
<artifactId>logback-classic</artifactId>
<version>1.1.3</version>
</dependency>
<dependency>
<groupId>org.logback-extensions</groupId>
<artifactId>logback-ext-spring</artifactId>
<version>0.1.2</version>
</dependency>
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>jcl-over-slf4j</artifactId>
<version>1.7.12</version>
</dependency>
```

common-logging 绑定日志组件

common-logging + log4j

添加依赖到 pom.xml 中即可。

```
<dependency>
<groupId>commons-logging</groupId>
<artifactId>commons-logging</artifactId>
<version>1.2</version>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.17</version>
</dependency>
```

使用 API

slf4j 用法

使用 slf4j 的 API 很简单。使用 LoggerFactory 初始化一个Logger实例，然后调用 Logger 对应的打印等级函数就行了。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class App {
    private static final Logger log = LoggerFactory.getLogger(App.class);
    public static void main(String[] args) {
        String msg = "print log, current level: {}";
        log.trace(msg, "trace");
        log.debug(msg, "debug");
        log.info(msg, "info");
        log.warn(msg, "warn");
        log.error(msg, "error");
    }
}
```

common-logging 用法

common-logging 用法和 slf4j 几乎一样，但是支持的打印等级多了一个更高级别的：fatal。

此外，common-logging 不支持{}替换参数，你只能选择拼接字符串这种方式了。

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class JclTest {
    private static final Log log = LogFactory.getLog(JclTest.class);

    public static void main(String[] args) {
        String msg = "print log, current level: ";
        log.trace(msg + "trace");
        log.debug(msg + "debug");
        log.info(msg + "info");
        log.warn(msg + "warn");
        log.error(msg + "error");
        log.fatal(msg + "fatal");
    }
}
```

log4j2 配置

log4j2 基本配置形式如下：

```
<?xml version="1.0" encoding="UTF-8"?>;
<Configuration>
    <Properties>
        <Property name="name1">value</property>
        <Property name="name2" value="value2"/>
    </Properties>
    <Filter type="type" ... />
    <Appenders>
        <Appender type="type" name="name">
            <Filter type="type" ... />
        </Appender>
        ...
    </Appenders>
    <Loggers>
        <Logger name="name1">
            <Filter type="type" ... />
        </Logger>
        ...
        <Root level="level">
            <AppenderRef ref="name"/>
        </Root>
    </Loggers>
</Configuration>
```

配置示例：

```

<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="debug" strict="true" name="XMLConfigTest"
    packages="org.apache.logging.log4j.test">

<Properties>
    <Property name="filename">target/test.log</Property>
</Properties>
<Filter type="ThresholdFilter" level="trace"/>

<Appenders>
    <Appender type="Console" name="STDOUT">
        <Layout type="PatternLayout" pattern="%m MDC%X%n"/>
        <Filters>
            <Filter type="MarkerFilter" marker="FLOW" onMatch="DENY" onMismatch="NEUTRAL"/>
            <Filter type="MarkerFilter" marker="EXCEPTION" onMatch="DENY" onMismatch="ACCEPT"/>
        </Filters>
    </Appender>
    <Appender type="Console" name="FLOW">
        <Layout type="PatternLayout" pattern="%C{1}.%M %m %ex%n"/><!-- class and line number -->
        <Filters>
            <Filter type="MarkerFilter" marker="FLOW" onMatch="ACCEPT" onMismatch="NEUTRAL"/>
            <Filter type="MarkerFilter" marker="EXCEPTION" onMatch="ACCEPT" onMismatch="DENY"/>
        </Filters>
    </Appender>
    <Appender type="File" name="File" fileName="${filename}">
        <Layout type="PatternLayout">
            <Pattern>%d %p %C{1} [%t] %m%n</Pattern>
        </Layout>
    </Appender>
</Appenders>

<Loggers>
    <Logger name="org.apache.logging.log4j.test1" level="debug" additivity="false">
        <Filter type="ThreadContextMapFilter">
            <KeyValuePair key="test" value="123"/>
        </Filter>
        <AppenderRef ref="STDOUT"/>
    </Logger>

    <Logger name="org.apache.logging.log4j.test2" level="debug" additivity="false">
        <AppenderRef ref="File"/>
    </Logger>

    <Root level="trace">
        <AppenderRef ref="STDOUT"/>
    </Root>
</Loggers>

</Configuration>

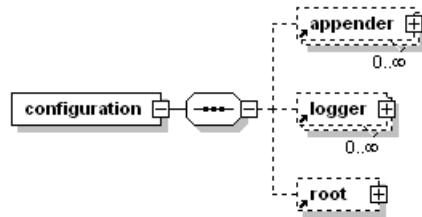
```

logback 配置

<configuration>

作用： <configuration> 是 logback 配置文件的根元素。

要点： 它有 <appender>、<logger>、<root> 三个子元素。



<appender>

作用：将记录日志的任务委托给名为 appender 的组件。

要点：可以配置零个或多个；它有 **<file>**、**<filter>**、**<layout>**、**<encoder>** 四个子元素。

属性：

- **name**: 设置 appender 名称。
- **class**: 设置具体的实例化类。

<file>

作用：设置日志文件路径。

<filter>

作用：设置过滤器。

要点：可以配置零个或多个。

<layout>

作用：设置 appender。

要点：可以配置零个或一个。

属性：

- **class**: 设置具体的实例化类。

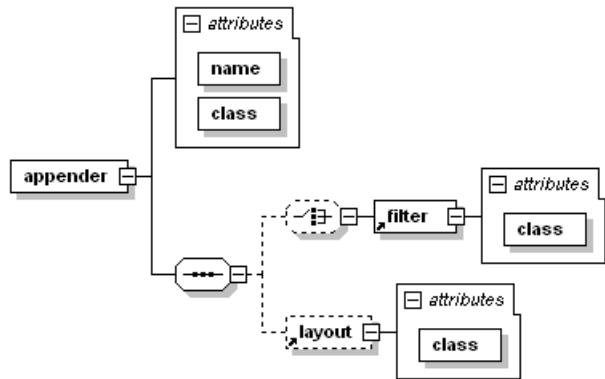
<encoder>

作用：设置编码。

要点：可以配置零个或多个。

属性：

- **class**: 设置具体的实例化类。



<logger>

作用：设置 logger。

要点：可以配置零个或多个。

属性：

- name
- level：设置日志级别。不区分大小写。可选值：TRACE、DEBUG、INFO、WARN、ERROR、ALL、OFF。
- additivity：可选值：true 或 false。

<appender-ref>

作用：appender 引用。

要点：可以配置零个或多个。

<root>

作用：设置根 logger。

要点：只能配置一个；除了 level，不支持任何属性。level 属性和 <logger> 中的相同；有一个子元素 <appender-ref>，与 <logger> 中的相同。

完整的 logback.xml 参考示例

在下面的配置文件中，我为自己的项目代码（根目录：org.zp.notes.spring）设置了五种等级：

TRACE、DEBUG、INFO、WARN、ERROR，优先级依次从低到高。

因为关注 spring 框架本身的一些信息，我增加了专门打印 spring WARN 及以上等级的日志。

```

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

<!-- logback中一共有5种有效级别，分别是TRACE、DEBUG、INFO、WARN、ERROR，优先级依次从低到高 -->
<configuration scan="true" scanPeriod="60 seconds" debug="false">

<property name="DIR_NAME" value="spring-helloworld"/>

<!-- 将记录日志打印到控制台 -->
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
  
```

```
<pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%n</pattern>
</encoder>
</appender>

<!-- RollingFileAppender begin -->
<appender name="ALL" class="ch.qos.logback.core.rolling.RollingFileAppender">
<!-- 根据时间来制定滚动策略 -->
<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
<fileNamePattern>${user.dir}/logs/${DIR_NAME}/all.%d{yyyy-MM-dd}.log</fileNamePattern>
<maxHistory>30</maxHistory>
</rollingPolicy>

<!-- 根据文件大小来制定滚动策略 -->
<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
<maxFileSize>30MB</maxFileSize>
</triggeringPolicy>

<encoder>
<pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%n</pattern>
</encoder>
</appender>

<appender name="ERROR" class="ch.qos.logback.core.rolling.RollingFileAppender">
<!-- 根据时间来制定滚动策略 -->
<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
<fileNamePattern>${user.dir}/logs/${DIR_NAME}/error.%d{yyyy-MM-dd}.log</fileNamePattern>
<maxHistory>30</maxHistory>
</rollingPolicy>

<!-- 根据文件大小来制定滚动策略 -->
<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
<maxFileSize>10MB</maxFileSize>
</triggeringPolicy>

<filter class="ch.qos.logback.classic.filter.LevelFilter">
<level>ERROR</level>
<onMatch>ACCEPT</onMatch>
<onMismatch>DENY</onMismatch>
</filter>

<encoder>
<pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%n</pattern>
</encoder>
</appender>

<appender name="WARN" class="ch.qos.logback.core.rolling.RollingFileAppender">
<!-- 根据时间来制定滚动策略 -->
<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
<fileNamePattern>${user.dir}/logs/${DIR_NAME}/warn.%d{yyyy-MM-dd}.log</fileNamePattern>
<maxHistory>30</maxHistory>
</rollingPolicy>

<!-- 根据文件大小来制定滚动策略 -->
<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
<maxFileSize>10MB</maxFileSize>
</triggeringPolicy>

<filter class="ch.qos.logback.classic.filter.LevelFilter">
<level>WARN</level>
<onMatch>ACCEPT</onMatch>
<onMismatch>DENY</onMismatch>
</filter>

<encoder>
```

```
<encoder>
<pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%n</pattern>
</encoder>
</appender>

<appender name="INFO" class="ch.qos.logback.core.rolling.RollingFileAppender">
<!-- 根据时间来制定滚动策略 -->
<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
<fileNamePattern>${user.dir}/logs/${DIR_NAME}/info.%d{yyyy-MM-dd}.log</fileNamePattern>
<maxHistory>30</maxHistory>
</rollingPolicy>

<!-- 根据文件大小来制定滚动策略 -->
<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
<maxFileSize>10MB</maxFileSize>
</triggeringPolicy>

<filter class="ch.qos.logback.classic.filter.LevelFilter">
<level>INFO</level>
<onMatch>ACCEPT</onMatch>
<onMismatch>DENY</onMismatch>
</filter>

<encoder>
<pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%n</pattern>
</encoder>
</appender>

<appender name="DEBUG" class="ch.qos.logback.core.rolling.RollingFileAppender">
<!-- 根据时间来制定滚动策略 -->
<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
<fileNamePattern>${user.dir}/logs/${DIR_NAME}/debug.%d{yyyy-MM-dd}.log</fileNamePattern>
<maxHistory>30</maxHistory>
</rollingPolicy>

<!-- 根据文件大小来制定滚动策略 -->
<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
<maxFileSize>10MB</maxFileSize>
</triggeringPolicy>

<filter class="ch.qos.logback.classic.filter.LevelFilter">
<level>DEBUG</level>
<onMatch>ACCEPT</onMatch>
<onMismatch>DENY</onMismatch>
</filter>

<encoder>
<pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%n</pattern>
</encoder>
</appender>

<appender name="TRACE" class="ch.qos.logback.core.rolling.RollingFileAppender">
<!-- 根据时间来制定滚动策略 -->
<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
<fileNamePattern>${user.dir}/logs/${DIR_NAME}/trace.%d{yyyy-MM-dd}.log</fileNamePattern>
<maxHistory>30</maxHistory>
</rollingPolicy>

<!-- 根据文件大小来制定滚动策略 -->
<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
<maxFileSize>10MB</maxFileSize>
</triggeringPolicy>

<filter class="ch.qos.logback.classic.filter.LevelFilter">
<level>TRACE</level>
</filter>
```

```

<level>TRACE</level>
<onMatch>ACCEPT</onMatch>
<onMismatch>DENY</onMismatch>
</filter>

<encoder>
<pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%n</pattern>
</encoder>
</appender>

<appender name="SPRING" class="ch.qos.logback.core.rolling.RollingFileAppender">
<!-- 根据时间来制定滚动策略 -->
<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
<fileNamePattern>${user.dir}/logs/${DIR_NAME}/springframework.%d{yyyy-MM-dd}.log
</fileNamePattern>
<maxHistory>30</maxHistory>
</rollingPolicy>

<!-- 根据文件大小来制定滚动策略 -->
<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
<maxFileSize>10MB</maxFileSize>
</triggeringPolicy>

<encoder>
<pattern>%d{HH:mm:ss.SSS} [%thread] [%-5p] %c{36}.%M - %m%n</pattern>
</encoder>
</appender>
<!-- RollingFileAppender end -->

<!-- logger begin -->
<!-- 本项目的日志记录，分级打印 -->
<logger name="org.zp.notes.spring" level="TRACE" additivity="false">
<appender-ref ref="STDOUT"/>
<appender-ref ref="ERROR"/>
<appender-ref ref="WARN"/>
<appender-ref ref="INFO"/>
<appender-ref ref="DEBUG"/>
<appender-ref ref="TRACE"/>
</logger>

<!-- SPRING框架日志 -->
<logger name="org.springframework" level="WARN" additivity="false">
<appender-ref ref="SPRING"/>
</logger>

<root level="TRACE">
<appender-ref ref="ALL"/>
</root>
<!-- logger end -->

</configuration>

```

log4j 配置

完整的 log4j.xml 参考示例

log4j 的配置文件一般有 xml 格式或 properties 格式。这里为了和 logback.xml 做个对比，就不介绍 properties 了，其实也没太大差别。

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>

    <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                  value="%d{yyyy-MM-dd HH:mm:ss,SSS} [%-5p] [%t] %c{36}.%M - %m%n"/>
        </layout>

        <!--过滤器设置输出的级别-->
        <filter class="org.apache.log4j.varia.LevelRangeFilter">
            <param name="levelMin" value="debug"/>
            <param name="levelMax" value="fatal"/>
            <param name="AcceptOnMatch" value="true"/>
        </filter>
    </appender>

    <appender name="ALL" class="org.apache.log4j.DailyRollingFileAppender">
        <param name="File" value="${user.dir}/logs/spring-common/jcl/all"/>
        <param name="Append" value="true"/>
        <!-- 每天重新生成日志文件 -->
        <param name="DatePattern" value="'-'yyyy-MM-dd'.log'"/>
        <!-- 每小时重新生成日志文件 -->
        <!--<param name="DatePattern" value="-'yyyy-MM-dd-HH'.log'"/>-->
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                  value="%d{yyyy-MM-dd HH:mm:ss,SSS} [%-5p] [%t] %c{36}.%M - %m%n"/>
        </layout>
    </appender>

    <!-- 指定logger的设置，additivity指示是否遵循缺省的继承机制-->
    <logger name="org.zp.notes.spring" additivity="false">
        <level value="error"/>
        <appender-ref ref="STDOUT"/>
        <appender-ref ref="ALL"/>
    </logger>

    <!-- 根logger的设置-->
    <root>
        <level value="warn"/>
        <appender-ref ref="STDOUT"/>
    </root>
</log4j:configuration>

```

参考

- [1] <http://www.slf4j.org/manual.html>
- [2] <http://logback.qos.ch/>
- [3] <http://logging.apache.org/log4j/1.2/>
- [4] <http://commons.apache.org/proper/commons-logging/>
- [5] <http://blog.csdn.net/yycdaizi/article/details/8276265>

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. [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

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!

聊一聊 Java 泛型中的通配符

Java后端 2019-08-26

点击上方**蓝色字体**, 选择“置顶公众号”

优质文章, 第一时间送达

前言

Java 泛型(generics)是 JDK 5 中引入的一个新特性, 泛型提供了编译时类型安全检测机制, 该机制允许开发者在编译时检测到非法的类型。

泛型的本质是参数化类型, 也就是说所操作的数据类型被指定为一个参数。

泛型带来的好处

在没有泛型的情况的下, 通过对类型 Object 的引用实现参数的“任意化”, “任意化”带来的缺点是要做显式的强制类型转换, 而这种转换是要求开发者对实际参数类型可以预知的情况下进行的。

对于强制类型转换错误的情况, 编译器可能不提示错误, 在运行的时候才出现异常, 这是本身就是一个安全隐患。

那么泛型的好处就是在编译的时候能够检查类型安全, 并且所有的强制转换都是自动和隐式的。

```
1 public class GlmapperGeneric<T> {
2     private T t
3 ;
4     public void set(T t) { this.t = t;
5 }
6     public T get() { return t;
7 }
8
9     public static void main(String[] args)
10 {
11     // do nothing
12 g
13 }
14
15 /**
16 * 不指定类型
17 */
18 public void noSpecifyType(){
19     GlmapperGeneric glmapperGeneric = new GlmapperGeneric();
20     glmapperGeneric.set("test")
21 ;
22     // 需要强制类型转
23 换
24     String test = (String) glmapperGeneric.get();
25     System.out.println(test);
26 }
27 }
```

```
28  /**
29   * 指定类型
30   * 欢迎关注订阅哈 Web 项目聚集地, 回复 技术博文 获取更多文章
31   */
32 public void specifyType(){
    GlmapperGeneric<String> glmapperGeneric = new GlmapperGeneric();
    glmapperGeneric.set("test")
;
    // 不需要强制类型转换
换
    String test = glmapperGeneric.get();
    System.out.println(test);
}
}
```

上面这段代码中的 `specifyType` 方法中省去了强制转换，可以在编译时候检查类型安全，可以用在类，方法，接口上。

泛型中通配符

我们在定义泛型类，泛型方法，泛型接口的时候经常会碰见很多不同的通配符，比如 `T, E, K, V` 等等，这些通配符又都是什么意思呢？

常用的 `T, E, K, V, ?`

本质上这些个都是通配符，没啥区别，只不过是编码时的一种约定俗成的东西。比如上述代码中的 `T`，我们可以换成 `A-Z` 之间的任何一个字母都可以，并不会影响程序的正常运行，但是如果换成其他的字母代替 `T`，在可读性上可能会弱一些。通常情况下，`T, E, K, V, ?` 是这样约定的：

- `?` 表示不确定的 java 类型
- `T (type)` 表示具体的一个java类型
- `K V (key value)` 分别代表java键值中的Key Value
- `E (element)` 代表Element

?无界通配符

先从一个小例子看起，原文在：

<https://codeday.me/bug/20180113/116421.html>

我有一个父类 `Animal` 和几个子类，如狗、猫等，现在我需要一个动物的列表，我的第一个想法是像这样的：

```
1 List<Animal> listAnimals
```

但是老板的想法确实这样的：

```
1 List<? extends Animal> listAnimals
```

为什么要使用通配符而不是简单的泛型呢？通配符其实在声明局部变量时是没有什么意义的，但是当你为一个方法声明一个参数时，它是非常重要的。

```
1 static int countLegs (List<? extends Animal > animals )
2 {
3     int retVal = 0
4 ;
5     for ( Animal animal : animals )
6     {
7         retVal += animal.countLegs();
8     }
9     return retVal;
10 }
11
12 static int countLegs1 (List< Animal > animals )
13 {
14     int retVal = 0
15 ;
16     for ( Animal animal : animals )
17     {
18         retVal += animal.countLegs();
19     }
20     return retVal;
21 }
22
23 public static void main(String[] args) {
24     List<Dog> dogs = new ArrayList<>();
25     // 不会报
错
        countLegs( dogs );
        // 报
错
        countLegs1(dogs);
    }
```

当调用 `countLegs1` 时，就会飘红，提示的错误信息如下：

```
countLegs1 (java.util.List<com.gimMapper.bridge.boot.generic.Animal>) in Test3 cannot be applied
to          (java.util.List<com.gimMapper.bridge.boot.generic.Dog>)
```

所以，对于不确定或者不关心实际要操作的类型，可以使用无限制通配符（尖括号里一个问号，即 `<?>`），表示可以持有任何类型。像 `countLegs` 方法中，限定了上届，但是不关心具体类型是什么，所以对于传入的 `Animal` 的所有子类都可以支持，并且不会报错。而 `countLegs1` 就不行。

[上界通配符 `< ? extends E >`](#)

上界:用 `extends` 关键字声明,表示参数化的类型可能是所指定的类型,或者是此类型的子类。

在类型参数中使用 `extends` 表示这个泛型中的参数必须是 E 或者 E 的子类,这样有两个好处:

- 如果传入的类型不是 E 或者 E 的子类,编译不成功
- 泛型中可以使用 E 的方法,要不然还得强转成 E 才能使用

```
1 private <K extends A, E extends B> E test(K arg1, E arg2){  
2     E result = arg2;  
3     arg2.compareTo(arg1);  
4     //....  
5     return result;  
6 }
```

类型参数列表中如果有多个类型参数上限,用逗号分开

下界通配符 `< ? super E >`

下界:用 `super` 进行声明,表示参数化的类型可能是所指定的类型,或者是此类型的父类型,直至 `Object`

在类型参数中使用 `super` 表示这个泛型中的参数必须是 E 或者 E 的父类。

```
1 private <T> void test(List<? super T> dst, List<T> src)  
2 {  
3     for (T t : src) {  
4         dst.add(t);  
5     }  
6 }  
7  
8 public static void main(String[] args) {  
9     List<Dog> dogs = new ArrayList<>();  
10    List<Animal> animals = new ArrayList<>();  
11    new Test3().test(animals,dogs);  
12 }  
13 // Dog 是 Animal 的子类  
14 class Dog extends Animal {  
15 }
```

`dst` 类型“大于等于”`src` 的类型,这里的“大于等于”是指 `dst` 表示的范围比 `src` 要大,因此装得下 `dst` 的容器也就能装 `src`。

?和 T 的区别



```
// 指定集合元素只能是 T 类型
List<T> list=new ArrayList<T>();
// 集合元素可以是任意类型，这种没有意义，一般是方法中，只是为了说明用法
List<?> list=new ArrayList<?>();
```

?和 T 都表示不确定的类型，区别在于我们可以对 T 进行操作，但是对 ?不行，比如如下这种：

```
1 // 可以
2 T t = operate();
3
4 // 不可以
5 ? car = operate();
```

简单总结下：

T 是一个 确定 的 类型，通常用于泛型类和泛型方法的定义，?是一个 不确定 的类型，通常用于泛型方法的调用代码和形参，不能用于定义类和泛型方法。

区别1：通过 T 来 确保 泛型参数的一致性

```
1 // 通过 T 来 确保 泛型参数的一致性
2 public <T extends Number> void
3 test(List<T> dest, List<T> src)
4
5 //通配符是 不确定的，所以这个方法不能保证两个 List 具有相同的元素类型
6 // 欢迎关注订阅哈 Web项目聚集地， 回复 技术博文 获取更多文章
7 public void
8 test(List<? extends Number> dest, List<? extends Number> src)
```

像下面的代码中，约定的 T 是 Number 的子类才可以，但是申明时是用的 String，所以就会飘红报错。

不能保证两个 List 具有相同的元素类型的情况

```
1 GlmapperGeneric<String> glmapperGeneric = new GlmapperGeneric<>();
2 List<String> dest = new ArrayList<>();
3 List<Number> src = new ArrayList<>();
4 glmapperGeneric.testNon(dest,src);
```

上面的代码在编译器并不会报错，但是当进入到 testNon 方法内部操作时（比如赋值），对于 dest 和 src 而言，就还是需要进行类型转换。

区别2：类型参数可以多重限定而通配符不行

```
public class MultiLimit implements MultiLimitInterfaceA , MultiLimitInterfaceB {  
    /**  
     * 使用 "&" 符号设定多重边界 (Multi Bounds)  
     * @param t  
     * @param <T>  
     */  
    public static<T extends MultiLimitInterfaceA & MultiLimitInterfaceB> void test(T t){  
        }  
    }  
  
    /**  
     * 接口 A  
     */  
    interface MultiLimitInterfaceA { }  
    /**  
     * 接口 B  
     */  
    interface MultiLimitInterfaceB { }
```

使用 & 符号设定多重边界 (Multi Bounds)，指定泛型类型 T 必须是 MultiLimitInterfaceA 和 MultiLimitInterfaceB 的共有子类型，此时变量 t 就具有了所有限定的方法和属性。对于通配符来说，因为它不是一个确定的类型，所以不能进行多重限定。

区别3：通配符可以使用超类限定而类型参数不行

类型参数 T 只具有一种 类型限定方式：

```
1 T extends A
```

但是通配符 ? 可以进行 两种限定：

```
1 ? extends A  
2 ? super A
```

Class< T > 和 Class< ? > 区别

前面介绍了 ? 和 T 的区别，那么对于， Class<T> 和 <Class<?> 又有什么区别呢？

最常见的是在反射场景下的使用，这里以用一段发射的代码来说明下。

```
1 // 通过反射的方式生成 multiLimit  
2 // 对象，这里比较明显的是，我们需要使用强制类型转换  
3 // 欢迎关注订阅哈 Web 项目聚集地， 回复 技术博文 获取更多文章  
4 MultiLimit multiLimit = (MultiLimit)  
5 Class.forName("com.glmapper.bridge.boot.generic.MultiLimit").newInstance();
```

对于上述代码，在运行期，如果反射的类型不是 MultiLimit 类，那么一定会报 java.lang.ClassCastException 错误。

对于这种情况，则可以使用下面的代码来代替，使得在编译期就能直接检查到类型的问题：

```
public class Test3 {  
    public static <T> T createInstance(Class<T> clazz) throws IllegalAccessException, InstantiationException {  
        return clazz.newInstance();  
    }  
  
    public static void main(String[] args) throws IllegalAccessException, InstantiationException {  
        A a = createInstance(A.class);  
        B b = createInstance(B.class);  
    }  
}  
  
class A {}  
class B {}
```

Class<T> 在实例化的时候，T 要替换成具体类。**Class<?>** 它是个通配泛型，? 可以代表任何类型，所以主要用于声明时的限制情况。比如，我们可以这样做申明：

```
1 // 可以  
2 public Class<?> clazz;  
3 // 不可以，因为 T 需要指定类型  
4 public Class<T> clazzT;
```

所以当不知道定声明什么类型的 Class 的时候可以定义一个**Class<?>**。

```
public class Test3 {  
    public Class<?> clazz;  
    public Class<T> clazzT;
```

那如果也想 **public Class<T> clazzT;** 这样的话，就必须让当前的类也指定 T

```
1 public class Test3<T> {  
2     public Class<?> clazz;  
3     // 不会报错  
4     public Class<T> clazzT;
```

小结

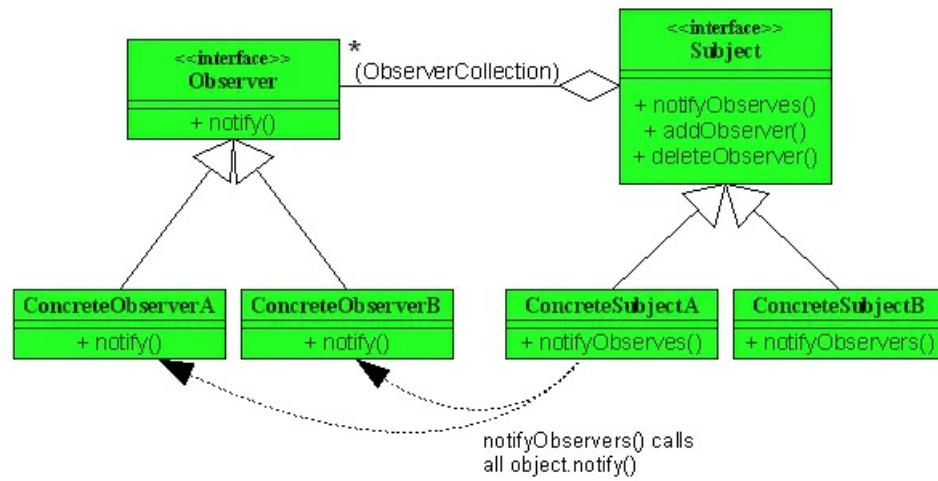
本文零碎整理了下 JAVA 泛型中的一些点，不是很全，仅供参考。如果有不当的地方，欢迎指正。文章参考 www.toutiao.com/a6694132392728199683

观察者模式的 Java 实现及应用

Java后端 2月21日

观察者模式定义

观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。



结构

关键字

- **Observable**

即被观察者，也可以被叫做主题（Subject）是被观察的对象。通常有注册方法（register），取消注册方法(remove)和通知方法(notify)。

- **Observer**

即观察者，可以接收到主题的更新。当对某个主题感兴趣的时候需要注册自己，在不需要接收更新时进行注销操作。

例子与应用

举一个生活中的例子：比如用户从报社订阅报纸，报社和用户之间是一对多依赖，用户可以在报社订阅（register）报纸，报社可以把最新的报纸发给用户（notify），用户自动收到更新。在用户不需要的时候还可以取消注册（remove）。

再比如Android中的EventBus，Rxjava的实现都是基于观察者模式的思想。再比如回调函数：Android中对Button的点击监听等等。

观察者模式可以用来解耦

自己用代码实现一个观察者模式

现在我们用代码来实现上面订阅报纸的例子：

NewProvider作为对于报社的抽象，每隔两秒钟向用户发送报纸；User作为用户的抽象，可以收到报纸。NewsModel作为对报纸本身的抽象。

```
public interface MyObservable {  
  
    void register(MyObserver myObserver);  
  
    void remove(MyObserver myObserver);  
}
```

```
void send(NewsModel model);

}

public interface MyObserver {

    void receive(NewsModel model);

}

public class NewsProvider implements MyObservable {
    private static final long DELAY = 2 * 1000;
    private List<MyObserver> mObservers;

    public NewsProvider() {
        mObservers = new ArrayList<>();
        generateNews();
    }

    private void generateNews() {
        Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            int titleCount = 1;
            int contentCount = 1;

            @Override
            public void run() {
                send(new NewsModel("title:" + titleCount++, "content:" + contentCount++));
            }
        }, DELAY, 1000);
    }

    @Override
    public void register(MyObserver myObserver) {
        if (myObserver == null)
            return;
        synchronized (this) {
            if (!mObservers.contains(myObserver))
                mObservers.add(myObserver);
        }
    }

    @Override
    public synchronized void remove(MyObserver myObserver) {
        mObservers.remove(myObserver);
    }

    @Override
    public void send(NewsModel model) {
        for (MyObserver observer : mObservers) {
            observer.receiveNews(model);
        }
    }
}

public class User implements MyObserver {
    private String mName;

    public User(String name) {
        mName = name;
    }

    void receive(NewsModel model) {
        System.out.println(mName + " received news: " + model.getTitle() + " " + model.getContent());
    }
}
```

```
}

@Override
public void receive(NewsModel model) {
    System.out.println(mName + " receive news:" + model.getTitle() + " " + model.getContent());
}
}

public class Test {
    public static void main(String[] args) {

        NewsProvider provider = new NewsProvider();
        User user;
        for (int i = 0; i < 10; i++) {
            user = new User("user:" + i);
            provider.register(user);
        }
    }
}
```

运行结果如下：

```
user:0 receive news:title:1 content:1
user:1 receive news:title:1 content:1
user:2 receive news:title:1 content:1
user:3 receive news:title:1 content:1
user:4 receive news:title:1 content:1
user:5 receive news:title:1 content:1
user:6 receive news:title:1 content:1
user:7 receive news:title:1 content:1
user:8 receive news:title:1 content:1
user:9 receive news:title:1 content:1
user:0 receive news:title:2 content:2
user:1 receive news:title:2 content:2
user:2 receive news:title:2 content:2
user:3 receive news:title:2 content:2
user:4 receive news:title:2 content:2
user:5 receive news:title:2 content:2
```

这样我们就自己动手完成了一个简单的观察者模式。

其实在JDK的util包内Java为我们提供了一套观察者模式的实现，在使用的时候我们只需要继承Observable和Observer类即可，其实观察者模式十分简单，推荐阅读JDK的实现代码真心没有几行。此外在JDK的实现中还增加了一个布尔类型的changed域，通过设置这个变量来确定是否通知观察者。

下面我们用JDK的类来实现一遍我们的观察者模式：

```

public class NewsProvider extends Observable {
    private static final long DELAY = 2 * 1000;

    public NewsProvider() {
        Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            private int titleCount = 1;
            private int contentCount = 1;
            @Override
            public void run() {
                setChanged();
                notifyObservers(new NewsModel("title:" + titleCount++, "content:" + contentCount++));
            }
        }, DELAY, 1000);
    }
}

public class User implements Observer {
    private String mName;

    public User(String name) {
        mName = name;
    }

    @Override
    public void update(Observable observable, Object data) {
        NewsModel model = (NewsModel) data;
        System.out.println(mName + " receive news:" + model.getTitle() + " " + model.getContent());
    }
}

```

非常简单有木有

回调函数与观察者模式

关于回调函数的定义在知乎上看到过一个[很赞的解释][2]:

你到一个商店买东西，刚好你要的东西没有货，于是在店员那里留下了你的电话，过了几天店里有货了，店员就打了你的电话，然后你接到电话后就到店里去取了货。在这个例子里，你的电话号码就叫回调函数，你把电话留给店员就叫登记回调函数，店里后来有货了叫做触发了回调关联的事件，店员给你打电话叫做调用回调函数，你到店里去取货叫做响应回调事件。回答完毕。

在Android中我们有一个常用的回调：对与View点击事件的监听。现在我们就来分析一下对于View的监听。

通常在我们使用的时候是这样的：

```

xxxView.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
    });
}

```

这样我们就注册好了一个回调函数。

我们可以在View的源码里发现这个接口：

```
public interface OnClickListener {
```

```
    void onClick(View v);  
}
```

当你setClickListener的时候在View的源码中可以看到对本地OnClickListener的初始化

```
public void setOnClickListener(@Nullable OnClickListener l) {  
    if (!isClickable()) {  
        setClickable(true);  
    }  
    mListenerInfo().mOnClickListener = l;  
}
```

当你的点击到一个View后Android系统经过一系列的调用最后到了View的performClick方法中：

```
public boolean performClick() {  
    final boolean result;  
    final ListenerInfo li = mListenerInfo;  
    if (li != null && li.mOnClickListener != null) {  
        playSoundEffect(SoundEffectConstants.CLICK);  
        li.mOnClickListener.onClick(this);  
        result = true;  
    } else {  
        result = false;  
    }  
  
    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);  
    return result;  
}
```

就在这里，触发了你的onClick方法，然后执行方法体。

这里我们的被观察者就是View，他的注册方法（register）就是setOnClickListener(),通知方法就是performClick；而OnClickListener就是观察者。只不过这里的只能注册一个观察对象而已。

[1]: /img/bVuUQI

[2]: www.zhihu.com/question/19801131/answer/13005983#showWechatShareTip

- 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. [2020 年 9 大顶级 Java 框架](#)
2. [聊聊 API 网关的作用](#)
3. [Class.forName 和 ClassLoader 有什么区别？](#)
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！

贼好用的 Java 工具类库

Ryan Wang Java后端 3月9日



作者 | Ryan Wang

链接 | ryanc.cc/archives/hutool-java-tools-lib

简介

Hutool是Hu + tool的自造词，前者致敬我的“前任公司”，后者为工具之意，谐音“糊涂”，寓意追求“万事都作糊涂观，无所谓失，无所谓得”的境界。

Hutool是一个Java工具包，只是一个工具包，它帮助我们简化每一行代码，减少每一个方法，让Java语言也可以“甜甜的”。Hutool最初是我项目中“util”包的一个整理，后来慢慢积累并加入更多非业务相关功能，并广泛学习其它开源项目精髓，经过自己整理修改，最终形成丰富的开源工具集。

功能

一个Java基础工具类，对文件、流、加密解密、转码、正则、线程、XML等JDK方法进行封装，组成各种Util工具类，同时提供以下组件：

- hutool-aop JDK动态代理封装，提供非IOC下的切面支持
- hutool-bloomFilter 布隆过滤，提供一些Hash算法的布隆过滤
- hutool-cache 缓存
- hutool-core 核心，包括Bean操作、日期、各种Util等
- hutool-cron 定时任务模块，提供类Crontab表达式的定时任务
- hutool-crypto 加密解密模块
- hutool-db JDBC封装后的数据操作，基于ActiveRecord思想
- hutool-dfa 基于DFA模型的多关键字查找
- hutool-extra 扩展模块，对第三方封装（模板引擎、邮件等）
- hutool-http 基于HttpURLConnection的Http客户端封装
- hutool-log 自动识别日志实现的日志门面
- hutool-script 脚本执行封装，例如Javascript
- hutool-setting 功能更强大的Setting配置文件和Properties封装
- hutool-system 系统参数调用封装（JVM信息等）
- hutool-json JSON实现
- hutool-captcha 图片验证码实现

简单测试

这两天使用Hutool把Halo里面的一些代码给替换掉了，不得不说，用起来十分顺心，下面简单介绍一下我用到的一些Hutool的工具类。

SecureUtil（加密解密工具）

主要是在登录的时候还有修改密码的时候用到的，因为数据库里面的密码是md5加密处理的，所以登录的时候需要先加密之后再到数据库进行查询，使用Hutool的话，只需要调用SecureUtil中的md5方法就可以了。

```
user = userService.userLoginByName(loginName, SecureUtil.md5(loginPwd));
```

HtmlUtil（HTML工具类）

这个工具类就比较厉害了，不过我在Halo当中用得最多的还是 `HtmlUtil.encode`，可以将一些字符转化为安全字符，防止xss注入和SQL注入，比如下面的评论提交。

```
comment.setCommentAuthor(HtmlUtil.encode(comment.getCommentAuthor()));
```

这就是防止有小坏蛋故意写一些可执行的js代码，然后提交评论，在后台面板就会执行这一段代码，比较危险，使用encode方法就可以将 `<script>` 标签给转化成 `<script>`，这样转化之后，js代码就不会执行了。

另外，HtmlUtil还提供了以下方法，有兴趣的可以去试一下。

- `HtmlUtil.restoreEscaped` 还原被转义的HTML特殊字符
- `HtmlUtil.encode` 转义文本中的HTML字符为安全的字符
- `HtmlUtil.cleanHtmlTag` 清除所有HTML标签
- `HtmlUtil.removeHtmlTag` 清除指定HTML标签和被标签包围的内容
- `HtmlUtil.unwrapHtmlTag` 清除指定HTML标签，不包括内容
- `HtmlUtil.removeHtmlAttr` 去除HTML标签中的属性
- `HtmlUtil.removeAllHtmlAttr` 去除指定标签的所有属性
- `HtmlUtil.filter` 过滤HTML文本，防止XSS攻击
- `CronUtil`（定时任务）

这个工具就更厉害了，完全不需要类似quartz这样的框架来做定时任务，而且CronUtil也不需要任何其他依赖，只需要在resources下建一个配置文件，然后在程序启动的时候将定时任务开启就行了，如Halo的定时备份功能（每天凌晨1点备份一次）。

cron.setting:

```
cc.ryanc.halo.web.controller.admin.BackupController.backupResources = 0 0 1 * * ?  
cc.ryanc.halo.web.controller.admin.BackupController.backupDatabase = 0 0 1 * * ?  
cc.ryanc.halo.web.controller.admin.BackupController.backupPosts = 0 0 1 * * ?  
  
@Override  
public void onApplicationEvent(ContextRefreshedEvent event){  
    this.loadActiveTheme();  
    this.loadOptions();  
    this.loadFiles();  
    this.loadThemes();  
    //启动定时任务  
    CronUtil.start();  
    log.info("定时任务启动成功! ");  
}
```

好了，就介绍这三个工具类，有兴趣的可以去试试其他的工具，挺全的，这应该是我用过最好用的一个工具类库了，值得一试。

- E N D -

推荐阅读

- [1. IntelliJ IDEA 快捷键 Windows 版本](#)
- [2. IntelliJ IDEA 常用快捷键 - Mac版本](#)
- [3. 盘点那些改变过世界的代码](#)
- [4. 基于token的多平台身份认证架构设计](#)



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！

还搞不懂 Java NIO？快来读读这篇文章！

Java后端 2019-12-15

以下文章来源于会点代码的大叔，作者会点代码的大叔



会点代码的大叔

程序员大叔，擅编码，懂调优，会架构，能讲段子，喜欢用大白话讲解复杂的技术。

首先，我们需要弄清楚几个概念：同步和异步，阻塞和非阻塞。



同步和异步

1. 同步

进程触发 IO 操作的时候，必须亲自处理；

比如你必须亲自去银行取钱。

2. 异步

进程触发 IO 操作的时候，可以不亲自处理，它把操作委托给 OS 处理，委托的时候需要告知数据的地址和大小，然后自己去做别的事情，当 IO 操作结束后会得到通知；

比如你把银行卡给我，让我帮你去银行取钱，你需要告诉我银行卡密码和取多少钱，我取完了之后把钱给你。

3. 总结

自己干就是同步，别人干就是异步。



阻塞和非阻塞

1. 阻塞

进程触发 IO 操作的时候，如果此时此时没办法读或者写，那么进程就一直等待，直到读写结束；

比如你去银行 ATM 取钱，前面有人在排队，那么就要一直等待，直到你取完钱；

2. 非阻塞

进程触发 IO 操作的时候，如果此时此时没办法读或者写，那么就先去做别的，等到有通知后，再继续读写；

比如你去银行柜台取钱，人比较多，那就先领一个号，等着叫到号再去对应的窗口办理业务；这里稍微有些不太恰当的是，我们等待的过程中，还得听着叫号。

3. 总结

我要等着不能做其他事就是阻塞，我不用等可以做其他事就是异步。



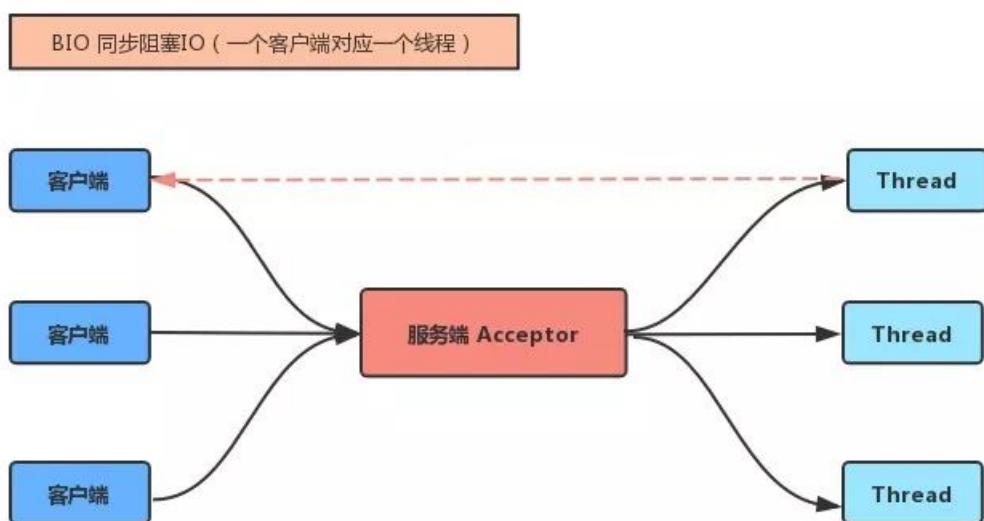
BIO

同步阻塞：一个请求过来，应用程序开了一个线程，等 IO 准备好，IO 操作也是自己干；

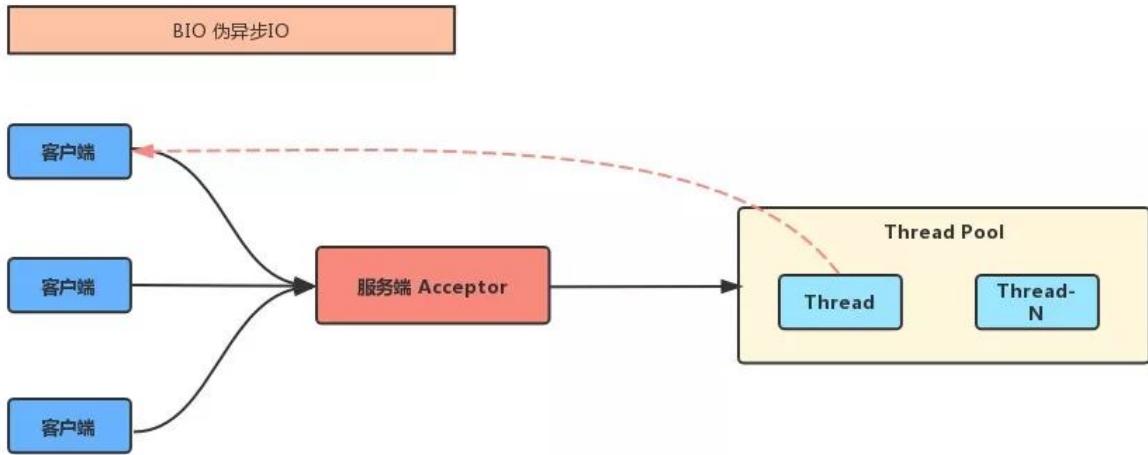
采用 BIO 模型的服务端，由一个独立的 Acceptor 线程负责进行监听；在 `while(true)` 循环中调用 `accept()` 方法，等待客户端的请求；

一旦接收到请求，就可以建立套接字开始进行读写操作，这时候不再接收其他的请求，直到读写完成；

为了让 BIO 能够同时处理多个请求，那么就需要使用多线程处理；当服务端接收到请求，就为客户端创建一个线程进行处理，处理完成后再做线程销毁；



不过因为一个请求就要启动一个线程，所以开销是比较大的，启动和销毁线程开销很大，而且每个线程都要占用内存，所以可以引入线程池，可以在一定程度上减少线程创建和销毁的开销；这也被叫做 **伪异步 IO**。



线程池维护着 N 个线程和一个消息队列；当有请求接入时，服务端将 Socket 作为参数传递到一个线程任务中进行处理；通过对线程池最大线程数和消息队列大小进行控制，所以就算访问量高于服务端的承载能力，也不会因为服务端的资源耗尽而导致宕机；

这个模型在 **请求量不高的时候，效率还是不错的**，而且也不需要考虑限流的问题（控制线程池的最大线程数量）。

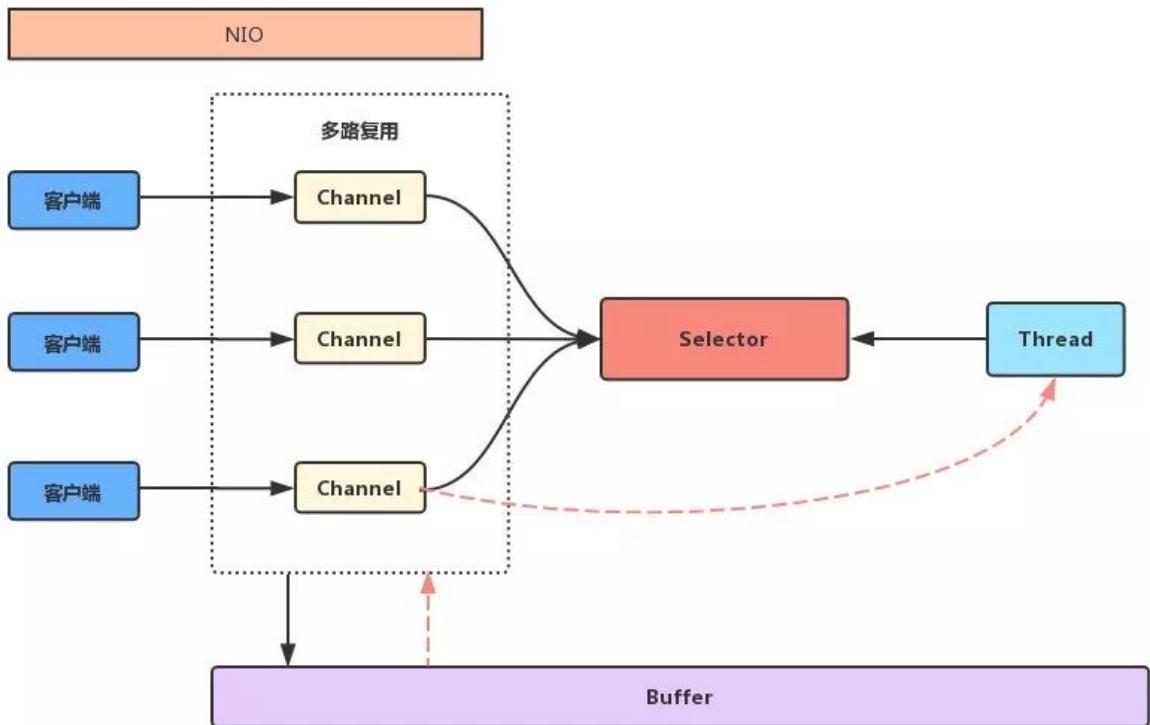


NIO

同步非阻塞：不用等待 IO 准备，准备好了会通知，不过 IO 操作还是要自己干；NIO 是一种多路复用机制，利用单线程轮询事件，Channel 来决定做什么，避免连接数多的时候，频繁进行线程切换导致性能问题（Select 阶段阻塞）。

听到这里，很多人可能已经懵了...什么是多路复用?Channel又是啥?Select 阶段到底是什么阶段?这里我用白话解释一下。

NIO是面向缓冲区的，可以将数据读取到一个缓冲区，稍后进行处理。NIO 有几个核心概念：



1. Channel 和 Buffer

Channel 可以理解成一个双向流，或者理解成一个通道，Buffer 就是缓存区，或者你就把它看做是一块内存空间，数据可以从 Channel 流进 Buffer，也可以从 Buffer 流进 Channel。

Channel 有很多种实现，比如：FileChannel 是从文件中读写数据，SocketChannel 通过 TCP 读写网络中的数据等等。

Buffer 也有多重类型，比如：ByteBuffer、CharBuffer、IntBuffer 等等，光看他们的名字就知道他们代表了不同的数据类型。

2. Selector

我们可以把 Selector 看做是一个管理员，可以管理多个 Channel，Selector 能够知道到哪个 Channel 已经做好了读写的准备。这样一个线程只要操作这个管理员就可以了，相当于一个线程可以管理多个 Channel；一旦监听到有准备好的 Channel，就可以进行相应的处理。

不过 Java 原生的 NIO 不好用，直到 Netty 的出现。



AIO
— - - -

异步非阻塞；因为事情不是自己做，其实也没有阻塞一说（都是非阻塞）；

AIO 是在 NIO 的基础上，引入异步通道的概念；NIO 是采用轮询的方式，不停地询问数据是否准备好了，准备好了就处

理;AIO 是向操作系统注册 IO 监听,操作系统完成 IO 操作了之后,主动通知,触发响应的函数(自己不做,让操作系统来做)。

目前看, AIO 应该的还不是很广泛。

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!

面试了 N 个候选人后，我总结出这份 Java 面试准备技巧！

Java后端 2019-11-04

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | hsm_computer

来源 | cnblogs.com/JavaArchitect/p/10011253.html

目录：

1. 框架是重点，但别让人感觉你只会山寨别人的代码
2. 别只看单机版的框架，分布式也需要了解
3. 对于数据库，别只知道增删改查，得了解性能优化
4. Java核心，围绕数据结构和性能优化准备面试题
5. Linux方面，至少了解如何看日志排查问题
6. 通读一段底层代码，作为加分项
7. 切记切记，把上述技能嵌入到你的项目里
8. 小结：本文更多讲述的准备面试的方法

在上周，我密集面试了若干位 Java后端的候选人，工作经验在3到5年间。

我的标准其实不复杂：

- 第一能干活
- 第二 Java基础要好
- 第三最好熟悉些分布式框架

相信其它公司招初级开发时，应该也照着这个标准来面的。

我也知道，不少候选人能力其实不差，但面试时没准备或不会说，这样的人可能进团队干活后确实能达到期望，但可能无法通过面试，面试官只根据面试情况来判断。

要知道，我们平时干活更偏重于业务，不可能大量接触到算法，数据结构，底层代码这类面试必问的问题点。

换句话说，面试准备点和平时工作要点匹配度很小。

作为面试官，我只能根据候选人的回答来决定面试结果。不过，与人方便自己方便。

所以我在本文里，将通过一些常用的问题来介绍面试的准备技巧。

大家在看后一定会感叹：只要方法得当，准备面试第一不难，第二用的时间也不会太多。

1、框架是重点，但别让人感觉你只会山寨别人的代码

在面试前，我会阅读简历以查看候选人在框架方面的项目经验，在候选人的项目介绍的环节，我也会着重关注候选人最近的框架经验，目前比较热门的是SSM。

不过，一般工作在5年内的候选人，大多仅仅是能“山寨”别人的代码，也就是说能在现有框架的基础上，照着别人写的流程，扩展出新的功能模块。

比如要写个股票挂单的功能模块，是会模仿现有的下单流程，然后从前端到后端再到数据库，依样画葫芦写一遍，最

多把功能相关的代码点改掉。

其实我们每个人都这样过来的，但在面试时，如果你仅仅表现出这样的能力，就和大多数人的水平差不多了，在这点就没法体现出你的优势了。

我们知道，如果单纯使用SSM框架，大多数项目都会有痛点。比如数据库性能差，或者业务模块比较复杂，并发量比较高，用Spring MVC里的Controller无法满足跳转的需求。

所以我一般还会主动问：你除了依照现有框架写业务代码时，还做了哪些改动？

我听到的回答有：

增加了Redis缓存，以避免频繁调用一些不变的数据。

或者，在MyBatis的xml里，select语句where条件有isnull，即这个值有就增加一个where条件，对此，会对任何一个where增加一个不带isnull的查询条件，以免该语句当传入参数都是null时，做全表扫描。

或者干脆说，后端异步返回的数据量很大，时间很长，我在项目里就调大了异步返回的最大时间，或者对返回信息做了压缩处理，以增加网络传输性能。

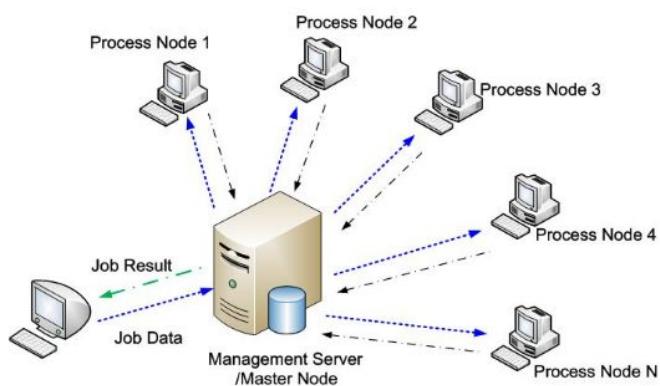
对于这个问题，我不在乎听到什么回答，我只关心回答符不符逻辑。一般只要答对，我就会给出“在框架层面有自己的体会，有一定的了解”的面试评价。

否则，我就只会给出“只能在项目经理带领下编写框架代码，对框架本身了解不多”。

其实，在准备面试时，归纳框架里的要点并不难，我就不信所有人在做项目时一点积累也没，只要你说出来，可以说，这方面你就碾压了将近7成的竞争者。

2、别只看单机版的框架，分布式也要了解

此外，在描述项目里框架技术时，最好你再带些分布式的技术。下面我列些大家可以准备的分布式技术。



1. 反向代理方面，nginx的基本配置，比如如何通过lua语言设置规则，如何设置session粘滞。如果可以，再看些nginx的底层，比如协议，集群设置，失效转移等。

2. 远程调用dubbo方面，可以看下dubbo和zookeeper整合的知识点，再深一步，了解下dubbo底层的传输协议和序列化方式。

3. 消息队列方面，可以看下kafka或任意一种组件的使用方式，简单点可以看下配置，工作组的设置，再深入点，可以看下Kafka集群，持久化的方式，以及发送消息是用长连接还是短拦截。

以上仅仅是用3个组件举例，大家还可以看下Redis缓存，日志框架，MyCAT分库分表等。

准备的方式有两大类：

- 第一是要学会说怎么用，这比较简单，能通过配置文件搭建成一个功能模块即可

- 第二是可以适当读些底层代码，以此了解下协议，集群和失效转移之类的高级知识点。

如果能在面试中侃侃而谈分布式组件的底层，那么得到的评价就会比较好，比如“深入了解框架底层”，或“框架经验丰富”，这样就算去面试架构师也行了，更何况是高级开发。

Tips:欢迎大家关注微信公众号：Java后端，来获取更多推送。

3、对于数据库，别只知道增删改查，得了解性能优化

在实际项目里，大多数程序员用到的可能仅仅是增删改查，当我们用Mybatis时，这个情况更普遍。

不过如果你面试时也这样表现，估计你的能力就和其它竞争者差不多了。

这方面，你可以准备如下的技能。

1. **SQL高级方面**，比如group by, having, 左连接，子查询（带in），行转列等高级用法。

2. **建表方面**，你可以考虑下，你项目是用三范式还是反范式，理由是什么？

3. 尤其是**优化**，你可以准备下如何通过执行计划查看SQL语句改进点的方式，或者其它能改善SQL性能的方式（比如建索引等）。

4. 如果你感觉有能力，还可以准备些**MySQL集群, MyCAT分库分表**的技能。比如通过LVS+Keepalived实现MySQL负载均衡，MyCAT的配置方式。同样，如果可以，也看些相关的底层代码。

哪怕你在前三点表现一般，那么至少也能超越将近一半的候选人，尤其当你在SQL优化方面表现非常好，那么你在面试高级开发时，数据库层面一定是达标的。

如果你连第四点也回答非常好，那么恭喜你，你在数据库方面的能力甚至达到了初级架构的级别。

4、Java核心，围绕数据结构和性能优化准备面试题

Java核心这块，网上的面试题很多，不过在此之外，大家还应当着重关注集合（即数据结构）和多线程并发这两块。在此基础上，大家可以准备些设计模式和虚拟机的说辞。

下面列些我一般会问的部分问题：

1. String a = "123"; String b = "123"; a==b的结果是什么？这包含了内存，String存储方式等诸多知识点

2. HashMap里的hashCode方法和equal方法什么时候需要重写？如果不重写会有什么后果？对此大家可以进一步了解HashMap（甚至ConcurrentHashMap）的底层实现

3. ArrayList和LinkedList底层实现有什么差别？它们各自适用于哪些场合？对此大家也可以了解下相关底层代码。

4. volatile关键字有什么作用？由此展开，大家可以了解下线程内存和堆内存的差别。

5. CompletableFuture，这是JDK1.8里的新特性，通过它怎么实现多线程并发控制？

6. JVM里, new出来的对象是在哪个区?再深入一下,问下如何查看和优化JVM虚拟机内存。

7. Java的静态代理和动态代理有什么差别?最好结合底层代码来说。

通过上述的问题点,我其实不仅仅停留在“会用”级别,比如我不会问如何在ArrayList里放元素。

大家可以看到,上述问题包含了“多线程并发”,“JVM优化”,“数据结构对象底层代码”等细节,大家也可以举一反三,通过看一些高级知识,多准备些其它类似面试题。

我们知道,目前Java开发是以Web框架为主,那么为什么还要问Java核心知识点呢?我这个是有切身体会的。

之前在我团队里,我见过两个人,一个是就会干活,具体表现是会用Java核心基本的API,而且也没有深入了解的意愿(估计不知道该怎么深入了解),另一位平时专门会看些Java并发,虚拟机等的高级知识。

过了半年以后,后者的能力快速升级到高级开发,由于对JAVA核心知识点了解很透彻,所以看一些分布式组件的底层实现没什么大问题。而前者,一直在重复劳动,能力也只一直停留在“会干活”的层面。

而在现实的面试中,如果不熟悉Java核心知识点,估计升高级开发都难,更别说是面试架构师级别的岗位了。

5、Linux方面,至少了解如何看日志排查问题

如果候选人能证明自己有“**排查问题**”和“**解决问题**”的能力,这绝对是个加分项,但怎么证明?

目前大多数的互联网项目,都是部署在Linux上,也就是说,日志都是在Linux,下面归纳些实际的Linux操作。

1. 能通过less命令打开文件,通过Shift+G到达文件底部,再通过?+关键字的方式来根据关键来搜索信息
2. 能通过grep的方式查关键字,具体用法是, grep 关键字 文件名, 如果要两次在结果里查找的话,就用grep 关键字1 文件名 | 关键字2 --color。最后--color是高亮关键字
3. 能通过vi来编辑文件
4. 能通过chmod来设置文件的权限

当然,还有更多更实用的Linux命令,但在实际面试过程中,不少候选人连一条linux命令也不知道。还是这句话,你哪怕知道些很基本的,也比一般人强了。

6、通读一段底层代码,作为加分项

如何证明自己对一个知识点非常了解?莫过于能通过底层代码来说明。

我在和不少工作经验在5年之内的程序员沟通时,不少人认为这很难?确实,如果要通过阅读底层代码了解分布式组件,那难度不小,但如果如下部分的底层代码,并不难懂。

1. ArrayList,LinkedList的底层代码里,包含着基于数组和链表的实现方式,如果大家能以此讲清楚扩容,“通过枚举器遍历”等方式,绝对能证明自己。
2. HashMap直接对应着Hash表这个数据结构,在HashMap的底层代码里,包含着hashcode的put, get等的操作,甚至在ConcurrentHashMap里,还包含着Lock的逻辑。如果大家在面试中,看看而言

3. 可以看下静态代理和动态代理的实现方式, 再深入一下, 可以看下Spring AOP里的实现代码。
4. 或许Spring IOC和MVC的底层实现代码比较难看懂, 但大家可以说些关键的类, 根据关键流程说下它们的实现方式。

其实准备的底层代码未必要多, 而且也不限于在哪个方面, 比如集合里基于红黑树的TreeSet, 基于NIO的开源框架, 甚至分布式组件的Dubbo, 都可以准备。

而且准备时未必要背出所有的底层(事实上很难做到), 你只要能结合一些重要的类和方法, 讲清楚思路即可(比如讲清楚HashMap如何通过hashCode快速定位)。

那么在面试时, 如何找到个好机会说出你准备好的上述底层代码?

在面试时, 总会被问到集合, Spring MVC框架等相关知识点, 你在回答时, 顺便说一句, “**我还了解这块的底层实现**”, 那么面试官一定会追问, 那么你就可以说出来了。

不要小看这个对候选人的帮助, 一旦你讲了, 只要意思到位, 那么最少能得到个“积极专业”的评价, 如果描述很清楚, 那么评价就会升级到“熟悉Java核心技能(或Spring MVC), 且基本功扎实”。

要知道, 面试中, 很少有人能讲清楚底层代码, 所以你抛出了这个话题, 哪怕最后没达到预期效果, 面试官也不会因此对你降低评价。

所以说, 准备这块绝对是“有百利而无一害”的挣钱买卖。

7、切记切记, 把上述技能嵌入到你的项目里

在面试过程中, 我经常会听到一些比较遗憾的回答, 比如候选人对SQL优化技能讲得头头是道, 但最后得知, 这是他平时自学时掌握的, 并没用在实际项目里。

当然这总比不说要好, 所以我会写下“在平时自学过SQL优化技能”, 但如果在项目里实践过, 那么我就会写下“有实际数据库SQL优化的技能”。

大家可以对比下两者的差别, 一个是偏重理论, 一个是直接能干活了。

其实, 很多场景里, 我就不信在实际项目里一定没有实践过SQL优化技能。

从这个案例中, 我想告诉大家的是, 你之前费了千辛万苦(其实方法方向得到, 也不用费太大精力)准备的很多技能和说辞, 最后应该落实到你的实际项目里。

比如你有过在Linux日志里查询关键字排查问题的经验, 在描述时你可以带一句, 在之前的项目里我就这样干的。

又如, 你通过看底层代码, 了解了TreeSet和HashSet的差别以及它们的适用范围, 那么你可以回想你之前做的项

目，是否有个场景仅仅适用于TreeSet？

如果有，那么你就可以适当描述下项目的需求，然后说，通过读底层代码，我了解了两者的差别，而且在这个实际需求里，我就用了TreeSet，而且我还专门做了对比性试验，发现用TreeSet比HashSet要高xx个百分点。

请记得，“**实践经验**”一定比“**理论经验**”值钱，而且大多数你知道的理论上的经验，一定在你的项目里用过。

所以，如果你仅仅让面试官感觉你只有“理论经验”，那就太亏了。

8、小结：本文更多讲述的准备面试的方法

本文给出的面试题并不多，但本文并没有打算给出太多的面试题。从本文里，大家更多看到的是面试官发现的诸多候选人的痛点。

本文的用意是让大家别再重蹈别人的覆辙，因此给出了不少准备面试的方法。

你的能力或许比别人出众，但如果你准备面试的方式和别人差不多，或者就拿你在项目里干的活来说事，而没有归纳出你在项目中的亮点，那么面试官还真的会看扁你。

作者 | hsm_computer

来源 | cnblogs.com/JavaArchitect/p/10011253.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」，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. 推荐一位大神，手握 GitHub 16000 star

2. 附源码！Spring Boot 并发登录人数控制

3. 为什么 Redis 单线程却能支撑高并发？

4. 干货！MySQL 数据库开发规范



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!