

程序员
Java 面经手册

作者：小傅哥
公众号：bugstack虫洞栈



让懂了就是真的懂!

HashCode / HashMap / ArrayList / AQS / Thread / JVM
扰动函数 · 负载因子 · 斐波那契 · 开放寻址

作者

你好，我是小傅哥。一线互联网 [java](#) 工程师、架构师，开发过交易&营销、写过运营&活动、设计过中间件也倒腾过中继器、IO 板卡。不只是写 Java 语言，也搞过 C#、PHP，是一个技术活跃的折腾者。

从 19 年开始萌生编写技术文章的想法，以沉淀、分享、成长为核心，让己和他人都能有所收获的目的。截止到目前已编写的内容包括；《用 Java 实现 JVM》、《Netty4.x 专题》、《中间件开发》、《领域驱动设计》、《全链路监控》、《字节码编程》等 10 个专题共计 200 篇左右原创内容。

2020 年写了一本 PDF [《重学 Java 设计模式》](#)，全网下载量 21 万+，帮助很多同学成长。同年 github 的两个项目，[CodeGuide](#)、[itstack-demo-design](#)，持续霸榜 Trending，成为全球热门项目。

我的作品



[《重学 Java 设计模式》](#)，这是一本互联网真实案例的实践书籍，从实际业务中抽离出，交易、营销、秒杀、中间件、源码等众多场景进行学习代码设计。

[《字节码编程》](#)，全书共计 107 页，11 万 7 千字，20 个章节涵盖三个字节码框架(ASM、Javassist、Byte-buddy)和 JavaAgent 使用并附带整套案例源码！

[《面经手册》](#)，这是一本以面试为引子的核心技术讲解，全书分为 5 个章 29 节分别介绍了：面试、数据结构、算法、锁、多线程以及 JVM 的核心内容。

我的技术站

- 公众号： bugstack虫洞栈 - 日常原创技术推文
- 博客： <http://bugstack.cn/> - 原创技术文章汇总，适合电脑(PC)端阅读
- Github： <https://github.com/fuzhengwei/CodeGuide/wiki> - 所有文章涉及的源码
汇总以及各类资料

添加好友

如果你在学习和成长的过程中遇到什么问题，也可以添加我的微信(fustack)



小傅哥 | bugstack.cn

友情打赏

感谢对作者辛苦码文的支持，自愿赞赏 



“非常感谢您的赞赏支持！”

小傅哥 | bugstack.cn 的赞赏码

介绍

Hello, world of java ! 你好，java 的世界！

欢迎来到这里，很高兴你能拿到这本书。如果你能坚持看完书中每章节的内容，那么不仅可以在你的面试求职上有所帮助，也更能让你对 Java 核心技术有更加深入的学习。

[《Java 面经手册》](#) 是一本以面试题为入口讲解 Java 核心技术的 PDF 书籍，书中内容也极力的向你证实[代码是对数学逻辑的具体实现](#)。为什么这么说？当你仔细阅读书籍时，会发现这里有很多数学知识，包括：扰动函数、负载因子、拉链寻址、开放寻址、斐波那契（Fibonacci）散列法还有黄金分割点的使用等等。编码只是在确定了研发设计后的具体实现，而设计的部分包括：数据结构、算法逻辑以及设计模式等，而这部分数据结构和算法逻辑在 Java 的核心 API 中体现的淋漓尽致。那么，也就解释了为什么这些内容成为了热点面试题，虽然可能我们都会觉得这样的面试像是造火箭。

那么，汽车 75 马力就够奔跑了，那你怎么还想要 2.0 涡轮+9AT 呢？大桥两边的护栏你每次走的时候都会去摸吗？那怎么没有护栏的大桥你不敢上呢？很多时候，你额外的能力才是自身价值的体现，不要以为你的能力就只是做个业务开发每天 CRUD。其实有时候并不是产品让你写 CRUD，而是因为你的能力只能产品功能设计成 CRUD。

就像数据结构、算法逻辑、源码技能，它都是可以为你的业务开发赋能的，也是写出更好、更易扩展程序的根基，所以学好这份知识非常有必要。
所以，我非常建议你深度阅读此书，如果书中的知识点对你只是抛砖引玉，那么就更好了，你可以继续深入索取，吸纳更多的、更深的内容到自己的头脑中。

1. 适合人群

1. 具备一定编程基础，工作 1-3 年的研发人员
2. 想阅读 Java 核心源码，但总感觉看不懂的
3. 看了太多理论，但没有实践验证的
4. 求职面试，总被面试题搞的死去活来的

2. 我能学到什么

1. 怎么写简历、怎么面大厂、怎么补充不足
2. Java 核心 API 中用到的数据结构和算法逻辑
3. 必会的数学知识，扰动函数、负载因子、拉链寻址、开放寻址、斐波那契 (Fibonacci) 散列法等
4. 学到学习的能力，跟着作者的分析和学习方式，增强自己的学习能力

3. 阅读建议

本书虽然是源码分析、理论实践，但并不会让读者感觉枯燥。作者：小傅哥，在每一篇的知识里都写下了实践验证的结果，对于每一章节都有对应的源码实现。小伙伴在阅读的时候可以对照源码实践，并且在源码中还包括了一些必备的素材（10 万单词表验证扰动函数）、工具、图标等，来让大家切身的体会到知识乐趣。也让所有认真阅读的读者，看后都能[让懂了就是真的懂！](#)

目录

第 1 章 谈谈面试

- 第 1 节：面试官都问我啥
- 第 2 节：认知自己的技术栈盲区
- 第 3 节：简历该怎么写
- 第 4 节：大厂都爱聊啥

第 2 章 数据结构和算法

- 第 1 节：HashCode 为什么使用 31 作为乘数
- 第 2 节：HashMap 源码分析(上)
- 第 3 节：HashMap 源码分析(下)
- 第 4 节：2-3 树与红黑树学习(上)
- 第 5 节：2-3 树与红黑树学习(下)
- 第 6 节：ArrayList 详细分析
- 第 7 节：LinkedList、ArrayList，插入分析
- 第 8 节：双端队列、延迟队列、阻塞队列
- 第 9 节：java.util.Collections、排序、二分、洗牌、旋转算法
- 第 10 节：StringBuilder 与 String 对比
- 第 11 节：ThreadLocal 源码分析

第 3 章 码农会锁

- 第 1 节：volatile
- 第 2 节：synchronized
- 第 3 节：ReentrantLock 和 公平锁
- 第 4 节：AQS 原理分析和实践运用
- 第 5 节：AQS 共享锁，Semaphore、CountDownLatch

第 4 章 多线程

- 第 1 节：Thread.start() 启动原理
- 第 2 节：Thread，状态转换、方法使用、原理分析

- 第 3 节: ThreadPoolExecutor
- 第 4 节: 线程池讲解以及 JVMTI 监控

第 5 章 JVM 虚拟机

- 第 1 节: JDK、JRE、JVM
- 第 2 节: JVM 类加载实践
- 第 3 节: JVM 内存模型
- 第 4 节: JVM 故障处理工具
- 第 5 节: GC 垃圾回收

本书源码

《Java 面经手册》是一本以面试为引子的核心技术讲解，全书分为 5 个章 29 节分别介绍了：面试、数据结构、算法、锁、多线程以及 JVM 的核心内容。并且每一个章节都有对应的案例源码，读者在阅读的过程中可以参考源码进行验证实践的学习，只有这样才会让你有更多的收获。

获取源码

1. 添加作者小傅哥的微信：fustack，备注：面经手册
2. 微信公众号：bugstack 虫洞栈，回复：面经手册



源码截图

The screenshot shows a Java code editor with a file named `ApiTest.java` open. The code is a unit test for a hash function. It imports `java.util.List`, `java.util.ArrayList`, and `java.util.String`. The class contains a single method `test_128hash()` with the following content:

```
141 // Test
142 public void test_128hash() {
143
144     // 初始化一组字符串
145     List<String> list = new ArrayList<>();
146     list.add("jlk");
147     list.add("lopi");
148     list.add("小傅哥");
149     list.add("edwe");
150     list.add("alpo");
151     list.add("yhjk");
152     list.add("plop");
153
154     // 定义要存放的数组
155     String[] tab = new String[8];
156
157     // 循环存放
158     for (String key : list) {
159         int idx = key.hashCode() & (tab.length - 1); // 计算索引位置
160         System.out.println(String.format("key值=%s Idx=%d", key, idx));
161         if (null == tab[idx]) {
162             tab[idx] = key;
163             continue;
164         }
165         tab[idx] = tab[idx] + "→" + key;
166     }
167
168     // 输出测试结果
169     System.out.println("测试结果：" + JSON.toJSONString(tab));
170
171     System.out.println("小傅哥".hashCode() ^ ("小傅哥".hashCode() >>> 16) & 7);
172
173     }
174 }
```

下载：公众号：bugstack 虫洞栈，回复：面经手册

第 1 章 谈谈面试(4 节)

第 1 节：面试官都问我啥

一直以来都有小伙伴问我什么时候出一些面试系列的文章，脑袋一热一口答应下来，《重学 Java 设计模式》写完就安排。但是怎么写，要写成什么样才对读者有帮助成了难点。

再三思考，面试只是一时的，工作、学习，才是长久坚持的。很多人面试不理想多半也是来自于自我学习能力的不足和工作内容的单一以及业务体量小导致。所以我想从全局培养人才的角度出发，也算是技术成长的经历中提取学习框架，帮助小伙伴们提升技术能力的同时也可以应对面试。

好像面试越来越难？

招聘一个合格的研发有多难？近半年我差不多收了 400 份简历，筛选简历到初面通过的不足 10%。这里面很多人连简历都写不好，面试时也经常回答不到点上，技术栈广度不够深度不足，项目经验缺乏，没有解决复杂场景的经验等等。但也同样有很多优秀的，手里有多个 offer 最后流失。对于企业是损失，但对于个人来说，我佩服这样的人，他们技术好有更多的选择。

30 岁要有 30 岁的能力，35 岁要有 35 的经历

有时候不是面试难，而是年龄与能力不匹配，对企业招聘来说，同样能力下你价格还高，为什么不招聘个年轻有活力的呢？（[什么？找对象](#)）有时候你会说这是贩卖焦虑，这是洗脑，但这条路上终究有人前进，有人被动回退。

学习是你这个职业一辈子的事

手里有个 [1](#) [2](#) [3](#)，不要想着去怼别人的 [4](#) [5](#) [6](#)，因为还有你不知道的 [7](#) [8](#) [9](#)。保持空瓶心态从 [0](#) 开始才能学到 [10 全](#)。

这一篇我会从简历的视角出发，简要概况出研发人员应该具备的能力 有了这篇的基础上，后续再逐步扩展系列的面试场景，以及对应的面试题细节讲解和从哪学习这些知识的一个引导。

一、程序员的愿望

5年，时间不长不短，有人结婚生娃、有人回家开店、也有人继续在大城市打拼。头两天在研发群里做了一次愿望留言，期待下5年后的自己。如图；

The screenshot shows a GitHub issue thread with five comments:

- amosigeda commented yesterday**
5年后30岁，依旧保持初心，身体健康。做一个技术达人。
1 reply
- Shing20 commented yesterday • edited**
1.深圳有房了吗?
2.长辈养老，后辈教育，传承问题解决了吗?
3.职业规划实现了吗?
4.财富自由了吗，还要天天准时上班，担心工资问题吗?
5.身体还健康吗?
6.书读完了吗?
...
预见未来，利于当前决策。打怪升级进行时，逆水行舟，不进则退!
2 replies, 3 likes
- dejunyu commented yesterday**
5年后
和老婆可能有个双胞胎吧
介于两个职业之间
1 like
- dingjuhan521 commented yesterday**
有可能未来就不是程序员了，感觉自己不太适合，但是多年以后写程序成为兴趣还是可以的，比如写写小游戏，写写辅助软件，最近打算找个女朋友吧，感觉一个人很孤单，没啥意思，加油，今年22.30岁之前也会有自己想要的。
1 like
- tanghang55 commented yesterday**
从php转java 城市定居 在家附近上班。
1 like

留言心愿集中的点，在于：

1. 身体健康
2. 头发还在
3. 加薪升职

努力！也照顾好自己。奋斗！也爱惜好身体。

愿望地址：<https://github.com/fuzhengwei/CodeGuide/issues/111>

二、谢飞机简历

这是一份工作4年的谢飞机简历，如下；

- 先看看简历，与自己的对比下，有什么可取之处
- 如果你是求职者，还希望在这份简历里补充什么
- 如果你是面试官，你举得这份简历还差了什么

基本信息

- 姓名：谢飞机
- 性别：男
- 年龄：27岁
- 工作经验：4年
- 联系电话：152****0620
- 通信邮箱：xiefeiji@qq.com
- Github：<https://github.com/xiefeiji>

教育背景

- 学校：天津工业大学
- 专业：软件工程专业
- 学历：本科
- 在校时间：2012年9月 - 2016年6月

求职意向

- 目标职能：Java软件开发工程师
- 工作地点：北京
- 工作性质：全职，非外包
- 到岗时间：2周以内

技术栈

- 「技能」 Java、Mysql、Redis、Netty、Mycat、ASM、Spring、SpringMVC、MyBatis、SpringBoot
- 「框架」 Dubbo、RabbitMq、ZK、elasticsearch、Hbase
- 「工具」 IDEA、Maven、Git、Postman、Jenkins、JMeter
- 「环境」 Linux、Tomcat

深入学习和使用 Java 语言以及相应的服务框架，对 Spring、Mybatis、Dubbo 等有部分核心源码的了解学习。对于高并发场景有较丰富的设计和落地经验，擅长解决项目中的技术难点，喜欢并发编程以及 JVM 相关知识。常实践设计模式于项目中，具备面向对象的编程思维，可以有效合理的设计代码分层并快速实现功能。

工作经历

- 2016年6月 - 至今，途乐数字科技
 - 职位：java后端工程师
 - 职责：负责营销和订单系统开发，包括秒杀、抽奖、优惠券、决策引擎建设等。
- 2015年9月 - 2016年4月，佳乐科技
 - 职位：实习开发工程师
 - 职责：简单需求开发设计，编写接口文档，组织技术分享和组内活动

项目经验

项目一、营销活动系统

- 开发环境：Idea+Jdk1.8+Maven3.0+Mysql5.7+Redis+Dubbo+Tomcat8.0
- 系统架构：Spring、Mybatis、Dubbo，三层框架，分布式部署，网关API
- 项目描述：这是一个营销优惠券服务中心，用于给用户发放和领取优惠券，在下单时进行营销使用，对用促销拉动消费。系统的建设中包括优惠券系统、营销系统、秒杀服务、发奖中心等多个平台组合使用。营销类系统复杂度较高，涉及的技术创新和优化点较多，目前由一个小组负责开发维护。
- 工作职责：
 - 负责秒杀系统和营销活动平台的设计开发和维护，设计优化方案，提升系统性能。
 - 梳理相应文档，维护对接资料，以及抽取系统中可公用的模块进行组件化设计。
 - 日常与产品、运营开发，讨论需求，多了解业务的基础上建设系统。

项目二、日志采集服务

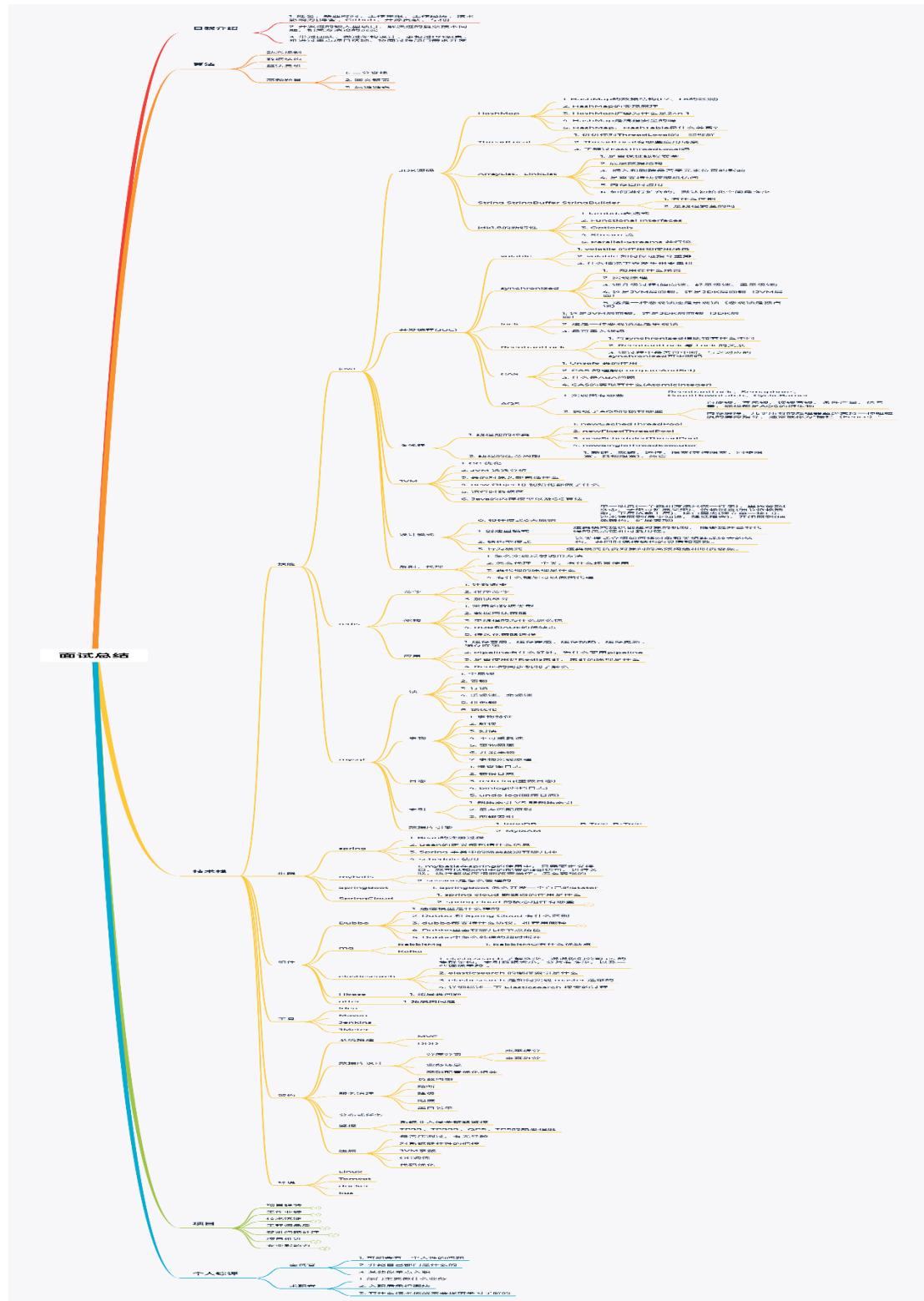
- 开发环境：Idea+Jdk1.8+Maven3.0+Mysql5.7+Redis+Dubbo+Tomcat8.0+ES+xxl-job
- 系统架构：Spring、Mybatis、SpringMVC
- 项目描述：负责采集系统运行过程的数据日志信息，用于分析异常时的运行信息。目前系统采用开发中间件的方式，集成到日志打印系统中，收集运行过程的全量或可配置的日志级别。
- 工作职责：
 - 这是组内的一个技术创新项目，我的职责主要开发日志存放和使用平台。
 - 对于大量的日志信息目前采用MQ接收，存放到ES+Hbase集群。
 - 同时记录日志相应接口的调用情况展示，以及日志会查操作。

个人评价

- 有较强的学习和解决问题的能力，善于通过一些系统问题总结分析根本原因，热爱于技术挑战。
- 在工作中可以独立承担相关职责，可以较强承受工作压力并愿意不断的提升自我。
- 性格开朗、积极乐观，可以精神饱满的投入到工作中。有着良好的团队协作能力和责任心，对工作有强烈的使命感。
- 想要长期从事软件开发行业，希望通过自己不断的努力来提升技术能力，并完成个人的职业发展计划。

- 这一份开篇的简历，可以阅读完下面的面试框架进行比对，看看你应该做些什么。
 - 后续的系列面经文章，将围绕简历开始，提问和讲解面试题。

三、面试框架



1. 自我介绍

- 1 分钟左右的自我介绍，简要的描述出；姓名、毕业时间、工作年限、工作经历、技术影响力[博客、Github、开源贡献、专利]等。
- 如果有较大型项目或者大家有一致性认知的项目和技术难点攻克，可以简单说出项目名称等。
- 方法论沉淀相关；架构设计能力、带过小组或者团队、跨部门协调、流程规范制定和执行等。

你要透露出的核心点就是个人的一个基本信息，以及项目和技术上的沉淀，给面试官留出和你聊下去的话题

例如：

面试官好，我叫谢飞机，16 年毕业于天津工业大学，软件工程专业，目前已工作 4 年。我从毕业后就职于途乐数字科技，负责营销和订单系统开发。在系统搭建、代码优化、问题处理上有较丰富的经验和处理能力。同时也喜欢写一些技术博客和看一些技术书籍，另外在 Spring、Dubbo 等源码学习上有过一些研究以及复用到业务开发中。感谢！

2. 面试类型

依赖于面试官的不同，与你一起进入面试的方式也不同，比如；

- 直接提问型；直接提问一些技术栈问题或者有些公司会考算法。
- 场景引导型；通过让你介绍的一些项目经历，用过什么技术栈，在场景下提问。
- 连环追问型；从一个点出发，你的每一个回答都在为下一个深入的问题做开始。
- 压力逼问型；压力面一般不多，但抗的住并能抓住重点，offer 基本稳了。

直接提问型 需要你有一定的技术栈广度和深度，问题往往也比较有跳跃性。但大部分题目会是热点问题，但可能不是日常开发中频率最高的技术点。

场景引导型 需要你有一定的开发架构经验和项目落地能力，这部分问题基本都会结合实际的业务场景进行提问，每一个场景就是一个复杂问题问题的解决能力。这里问到的场景会与你简历中的工作经历和项目相关，但复杂程度可能会超过你目前简历中的项目内容。比如；你写了一个订单类的，那么会问你秒杀的实际解决方案。这样的问题很难背题应付，需要真的经历过，研究过。

连环追问型 从一个小的技术点开始，一层层往下剥，每一次的回答也几乎都是下一个问题的深入点的来源。这种问题不仅考察面试者，也是对面试官的考验。往往在招聘高级别开发时会进行定向深入挖掘提问，找到匹配的行业专家级技术人员。

压力逼问型 这种面试方式一般不多，但可能有的高级面试官会让你感受到这种气场。一方面是确实人家有这样的技术气势，另外一个是来自自己的紧张。如果会不是问题，还会在面试官那留下很高的打分。如果不会，那么会感觉到你越不会什么，面试官越问你什么。

以上就是基本这四类面试官的风格，求职除了技术能力外还有一部分是眼缘，不一定一个问题不会就是你不优秀，而是在寻找这个职位最适合匹配度的人员。

3. 算法

算法一整块内容来考的互联网公司目前有一些，比如：头条、谷歌、百度，但不一定所有职位都需要去考算法。热频考点大部分可以分如下几块：

1. 动态规划
2. 数据结构
3. 算法思维

在题目上一般会有排序、二分查找、回文链表、反链链表，和数据结构设计方面。这些题目可以通过 leetcode-cn.com 刷题进行练习。

4. 技术栈

技术栈考查的是程序员在编程开发领域学习的广度和深度，日常的业务开发往往提升都是对 API 使用的熟练程度，如果再缺少一些系统设计和技术难点攻克，或者说没有大流量的冲击。那么确实很难回答一些技术深度问题。如下汇总了基本会再技术栈面试中涉及的考题，可以参考提升个人技术能力。

4.1 技能

4.1.1 Java

JDK 源码

1. HashMap
 1. HashMap 的数据结构(1.7、1.8 的区别)
 2. HashMap 的实现原理
 3. HashMap 扩容为什么是 $2^n - 1$
 4. HashMap 是线程安全的吗

5. HashMap、HashTable 是什么关系？
2. ThreadLocal
 1. 讲讲你对 ThreadLocal 的一些理解
 2. ThreadLocal 有哪些应用场景
 3. 了解过 FastThreadLocal 吗
3. ArrayList、LinkedList
 1. 是否保证线程安全
 2. 底层数据结构
 3. 插入和删除是否受元素位置的影响
 4. 是否支持快速随机访问
 5. 内存空间占用
 6. 如何进行扩容的，默认初始化空间是多少
4. String StringBuffer StringBuilder
 1. 有什么区别
 2. 是线程安全的吗
5. jdk1.8 的新特性
 1. lambda 表达式
 2. Functional Interfaces
 3. Optionals
 4. Stream 流
 5. Parallel-Streams 并行流

并发编程(j.u.c)

1. volatile
 1. volatile 的作用和使用场景
 2. volatile 如何保证指令重排
 3. 什么情况下会发生指令重排
2. synchronized
 1. 一般用在什么场景
 2. 实现原理
 3. 锁升级过程(偏向锁、轻量级锁、重量级锁)
 4. 这是 JVM 层面锁，还是 JDK 层面锁 {JVM 层面}
 5. 这是一种悲观锁还是乐观锁 {悲观锁是独占锁}
3. lock
 1. 这是 JVM 层面锁，还是 JDK 层面锁 {JDK 层面}
 2. 这是一种悲观锁还是乐观锁
 3. 是可重入锁吗
4. ReentrantLock
 1. 与 synchronized 相比较有什么不同

2. ReentrantLock 与 Lock 的关系
 3. 锁过程中是否可中断，与之对应的 synchronized 可中断吗
5. CAS
1. Unsafe 类的作用
 2. CAS 的理解 (compareAndSet)
 3. 什么是 ABA 问题
 4. CAS 的实现有什么 (AtomicInteger)
6. AQS
1. 实现类有哪些 ReentrantLock、Semaphore、CountDownLatch、CyclicBarrier
 2. 实现了 AQS 的锁有哪些 自旋锁、互斥锁、读锁写锁、条件产量、信号量、栅栏都是 AQS 的衍生物 内存屏障，几乎所有的处理器至少支持一种粗粒度的屏障指令，通常被称为“栅栏 (Fence) ”

多线程

1. 线程池的种类
 1. newCachedThreadPool
 2. newFixedThreadPool
 3. newScheduledThreadPool
 4. newSingleThreadExecutor
2. 线程的生命周期
 1. 新建、就绪、运行、阻塞(等待阻塞、同步阻塞、其他阻塞)、死亡

JVM

1. GC 优化
2. JVM 逃逸分析
3. 类的对象头都包括什么
4. new Object() 初始化都做了什么
5. 运行时数据区
6. Java 的内存模型以及 GC 算法

设计模式

1. 设计模式 6 大原则 单一职责(一个类和方法只做一件事)、里氏替换(多态，子类可扩展父类)、依赖倒置(细节依赖抽象，下层依赖上层)、接口隔离(建立单一接口)、迪米特原则(最少知道，降低耦合)、开闭原则(抽象架构，扩展实现)

2. 创建型模式 这类模式提供创建对象的机制，能够提升已有代码的灵活性和可复用性。
3. 结构型模式 这类模式介绍如何将对象和类组装成较大的结构，并同时保持结构的灵活和高效。
4. 行为模式 这类模式负责对象间的高效沟通和职责委派。

反射、代理

1. 怎么实现反射调用方法
2. 怎么代理一个类，有什么场景使用
3. 类代理的原理是什么
4. 有什么框架可以做类代理

4.1.2 Redis

命令

1. 计数命令
2. 排序命令
3. 加锁命令

架构

1. 常用的数据类型
2. 数据淘汰策略
3. 单线程的为什么那么快
4. RDB 和 AOF 的优缺点
5. 持久化策略选择

应用

1. 缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级
2. Pipeline 有什么好处，为什么要用 pipeline
3. 是否使用过 Redis 集群，集群的原理是什么
4. Redis 的同步机制了解么

4.1.3 Mysql

锁

1. 全局锁
2. 表锁

- 3. 行锁
- 4. 乐观锁、悲观锁
- 5. 排他锁
- 6. 锁优化

事务

- 1. 事物特征
- 2. 脏读
- 3. 幻读
- 4. 不可重复读
- 5. 事物隔离
- 6. 并发事物
- 7. 事物实现原理

日志

- 1. 慢查询日志
- 2. 错误日志
- 3. redo log(重做日志)
- 4. binlog(归档日志)
- 5. undo log(回滚日志)

索引

- 1. 聚集索引 VS 非聚集索引
- 2. 最左匹配原则
- 3. 前缀索引

引擎

- 1. InnoDB
- 2. MyISAM

4.2 框架

4.2.1 Spring

- 1. Bean 的注册过程
- 2. Bean 的定义都包括什么信息

3. Spring 事务中的隔离级别有哪几种
4. schedule 使用

4.2.2 Mybatis

1. mybatis 在 spring 的使用中，只需要定义接口，就可以和 xml 中的配置的 sql 语句，进行关联，执行数据库增删改查操作。怎么实现的
2. session 是怎么管理的

4.3.3 SpringBoot

1. SpringBoot 怎么开发一个自己的 Stater

4.2.4 SpringCloud

1. spring cloud 断路器的作用是什么
2. spring cloud 的核心组件有哪些 Eureka：服务注册与发现。 Feign：基于动态代理机制，根据注解和选择的机器，拼接请求 url 地址，发起请求。 Ribbon：实现负载均衡，从一个服务的多台机器中选择一台。 Hystrix：提供线程池，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题。 Zuul：网关管理，由 Zuul 网关转发请求给对应的服务。

4.3 组件

4.3.1 Dubbo

1. 通信模型是什么样的
2. Dubbo 和 Spring Cloud 有什么区别
3. dubbo 都支持什么协议，推荐用哪种 dubbo:// (推荐) rmi:// hessian:// http:// webservice:// thrift:// memcached:// redis:// rest://
4. Dubbo 里面有哪几种节点角色
5. Dubbo 中怎么处理的超时断开

4.3.2 Mq

1. RabbitMq
2. Kafka

4.3.3 elasticsearch

1. elasticsearch 了解多少，说说你们公司 es 的集群架构，索引数据大小，分片有多少，以及一些调优手段。
2. elasticsearch 的倒排索引是什么
3. elasticsearch 是如何实现 master 选举的
4. 详细描述一下 Elasticsearch 搜索的过程

4.3.4 Hbase

1. 拓展类问题

4.3.5 otter

1. 拓展类问题

4.4 工具

1. Idea
2. Maven
3. Jenkins
4. JMeter

4.5 架构

4.5.1 系统搭建

1. MVC
2. DDD 领域驱动设计

4.5.2 数据库设计

1. 分库分表(水平拆分、垂直拆分)
2. 业务场景
3. 基础配置优化相关

4.5.3 服务治理

1. 负载均衡
2. 熔断
3. 降级
4. 限流

5. 黑白名单

4.5.4 分布式任务

1. xxl-job

4.5.5 监控

1. 系统非入侵全链路监控
2. TP99、TP999、QPS、TPS 的熟悉程度

4.5.6 压测

1. 是否压测过，有无经验
2. 对系统健壮性的把控
3. JVM 参数
4. GC 调优
5. 代码优化

4.6 环境

1. Linux
2. Tomcat
3. docker
4. k8s

5. 项目

5.1 项目经验

1. 开发了哪些项目
2. 重点项目是什么
3. 你主要负责哪些
4. 有过什么优化

5.2 工作业绩

1. 中大型项目架构能力
2. 复杂项目落地能力

- 3. 重点项目执行落地
- 4. 交付能力&质量

5.3 技术沉淀

- 1. 方法论
- 2. 流程规范制定
- 3. 交付质量
- 4. 公用组件建设
- 5. 开源项目
- 6. 复杂架构设计经验
- 7. 团队技术分享

5.4 工程师品质

- 1. 认知范围，技术、业务、运营
- 2. 学习能力，接受能力
- 3. 创新技术，迁移能力

5.5 疑难问题处理

- 1. 复杂问题推进解决能力
- 2. 紧急事故解决能力

5.6 项目推进

- 1. 中大型项目推进落地
- 2. 资源协调安排
- 3. 流程规范实施

5.7 专业影响力

- 1. 项目推进过程中方案执行落地
- 2. 带动他人共同完成，并赋予能力提升
- 3. 技术价值创造
- 4. 开源项目和专利

6. 个人&面试官

6.1 个人

1. 可能会有一个人性的问题
2. 介绍自己部门是什么的
3. 其他你早点入职

6.2 面试官

1. 部门主要做什么业务
2. 入职后承担哪块
3. 有什么技术挑战需要提前学习了解的

四、总结

- 结合以上框架内容看自己是否是一个能抗住打的求职者，综合素质是否全面，技术栈广度、深度是否在瓶颈里徘徊，思考下怎么突破。
- 面试只是一份求职的开始，面试题也只是学习过程的知识点总结，只流于背题很容易被问倒。不同的面试官风格、水平、关注点也都不同，只有自己学扎实了才能随心所面。
- 以上的技术框架总结有一个 xmind 思维导图，以及面试简历，都可以通过在公众号：[bugstack 虫洞栈](#)，回复资源下载进行获取。[获取链接打开](#)，[找到 ID: 19](#)

第 2 节：认知自己的技术栈盲区

讲到技术盲区，先说说我自己。几年前我也是从传统行业跨到互联网，同时还是从 C# 开发转到 Java 开发，与其说转不如说是回归 Java。从上大学到毕业实习，我都是做 Java 的，但入职第一家公司，由于技术积累以及需要用 C# 与中继器、IO 板卡、摄像头等进行交互，以及开发的软件是部署到客户端的，所以整套服务都是 C# 与少部分的 C 语言进行编写。

但可能由于自身仍有一股热爱 Java 的兴趣，把公司里我接触到的 C# 软件，都用 Java 去实现了一遍。比如 Java 与 RS232 串口进行通信操作，读取摄像头扫描信息等等。最终承载着这份兴趣跳槽了，刚出来的时候面试也是晕的，毕竟很多技术栈的内容是我没用过的，好在研究过 Netty，算是当时一个技术亮点勉强通过，实现了我想写 Java 代码的心愿。

技术组里的 gai 溜子

可能每个人或多或少都有一些未接触的技术，虽然这些技术栈可能是整套架构中包括的，但可能由于经常忙于业务需求开发、技术组内缺少分享传承、再者是自己懒于探索究竟，而导致自己的技术盲区越来越多，久而久之就成了技术组里的 gai 溜子。

你的技术瓶颈也是一点点积累的

有一句常听到的话是：[你知道的越多，你不知道的越多](#)。这是积极向上学习的人总结出来的。与之相反的还有一句：[你知道的越少，你不知道的越少](#)。

认知半径决定了眼界目光，可扫描区域的大小决定发展轨迹。技术瓶颈往往都是因为自我学习能力被封锁，日积月累的流逝沉淀导致。就比如，大学生毕业前有 30 万行的代码编写量，找工作还会很难吗？[30 万行一天平均 200 行而已！](#)

一、技术栈采集问卷

1. 组织了一波采集活动

这是一波问卷收集，在两个 500 人的技术群中，邀请大家填写自己常用的技术栈。通过这样的方式互相了解目前在使用的技术栈都有哪些，你用的技术栈，肯定有我没用过的。很多时候有些技术的不知晓，就是因为各自圈子内的架构基本是固定的，所以限制了认知范围，而大家互相补充就可以知晓还有哪些技术栈是自己没用接触过的。

目前技术栈采集仍会持续进行中，欢迎随时补充并查看；

- 采集问卷：
https://docs.qq.com/form/page/DY0JsRHVNUnVyQXBZ?w_tencentdocx_form=1
- 汇总列表：<https://docs.qq.com/sheet/DY1ZTY1FQYUh6THNC>

2. 汇总出来的技术栈列表

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
小傅哥 bugstack.cn	2020/7/ 29 11:11:5 6	Drools
柠檬楠	2020/7/ 29 11:14:0 0	spring 全家桶、kafka、mysql、oracle、 netty、redis、docker、kubernetes、 python3、zookeeper、mycat、sharding-jdbc、
Shing	2020/7/ 29 11:14:0 8	Dubbo
不忘初心	2020/7/ 29 11:15:0 4	java
	2020/7/ 29 11:15:2 0	springboot、springcloud
BIUBIUBIU	2020/7/ 29 11:15:2 6	elasticsearch

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
油条	2020/7/ 29 11:16:0 1	SpringBoot+Vue+MySql
Spirit_wolf	2020/7/ 29 11:16:1 7	Java
调包侠	2020/7/ 29 11:16:1 7	docker kafka rabbitmq rocketmq redis cloud-alibaba dubbo nginx
缄默	2020/7/ 29 11:16:2 2	springboot mybatis redis mysql
飞鱼	2020/7/ 29 11:16:3 2	java
GROW	2020/7/ 29 11:16:4 5	Spark、Phoenix
Michael	2020/7/ 29 11:18:1 6	Java
子木	2020/7/ 29	java, oracle, mysql, es

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
	11:18:18	
寒小武	2020/7/29 11:18:35	请输入
建润	2020/7/29 11:19:00	ssm
	2020/7/29 11:19:03	spring, redis, hadoop, docker, elasticsearch
鎏鬱蘂鸞	2020/7/29 11:19:10	java
wangChen	2020/7/29 11:19:19	java、python、docker、
Jiao&Leon	2020/7/29 11:19:35	dojo
库里	2020/7/29 11:19:46	clickhouse

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
Chord	2020/7/ 29 11:19:5 2	mybatis-plus
查 尔斯	2020/7/ 29 11:19:5 3	SpringBoot+Vue
Jeao&Leon	2020/7/ 29 11:19:5 7	foxbase
向日葵不 流泪	2020/7/ 29 11:20:0 0	HBase
canonnk	2020/7/ 29 11:20:0 2	.net sqlserver node vue react
Chord	2020/7/ 29 11:20:0 5	springboot
王庆文	2020/7/ 29 11:20:1 4	Sentinel JVM-sandbox
噼里啪啦 稀碎	2020/7/ 29	Cat, apollo

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
	11:20:1 7	
JKM	2020/7/ 29 11:20:1 9	收集技术
Jeao&Leon	2020/7/ 29 11:20:3 1	ffmpeg
罐头	2020/7/ 29 11:20:3 3	spring、mybatis、dubbo、nacos、rabbitmq、 es
无名氏 ◎7173	2020/7/ 29 11:20:4 0	flink, clickhouse
日落黄昏 下	2020/7/ 29 11:20:4 1	java
起风了	2020/7/ 29 11:20:4 3	Spock
Gavin	2020/7/ 29 11:20:4 6	haddop, spark,

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
军	2020/7/ 29 11:20:4 9	SpringBoot
Shing	2020/7/ 29 11:21:0 8	activemq, redis, Spring-security, PGsql,
Believe 	2020/7/ 29 11:21:3 2	spring-boot-cloud-security dubbo netty mybatis-plus hadoop mysql redis disruptor
子木	2020/7/ 29 11:21:3 3	java, oracle, redis, mysql, es, mybatis, vue, ele mentui
康雁飞	2020/7/ 29 11:21:4 0	spring boot; redis
kirago	2020/7/ 29 11:22:5 8	ansible、django、flask、kubernetes、 springboot、springcloud、docker、 prometheus
张彬	2020/7/ 29 11:23:0 0	vert.x、vert.x、vert.x (重要的东西说三 遍)、netty
shmilylyp	2020/7/ 29	java

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
	11:23:1 1	
HQRNQF	2020/7/ 29 11:23:1 7	GraalVm
灯泡厂老 爷	2020/7/ 29 11:23:4 6	spring boot, mysql, reids, rabbitmq, elasticsearch
张彬	2020/7/ 29 11:25:1 0	vert.x、netty、RxJava、响应式编程、 Disruptor
调包侠	2020/7/ 29 11:25:2 6	t-io netty springboot supervisor solr elasticsearch
	2020/7/ 29 11:25:3 1	spark
Perry	2020/7/ 29 11:26:2 4	activemq-Artemis, couchbase
杰	2020/7/ 29 11:26:3 9	springboot、mybatis-plus、nacos、 sentinel、gateway、redis

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
Neck	2020/7/ 29 11:26:5 2	spring, mybatis
小丑人	2020/7/ 29 11:27:1 2	springcloud
刘志航	2020/7/ 29 11:28:1 9	Spring、SpringBoot、SpringCloud、Dubbo、zk、Redis、Nacos、apollo、xxJob、Mysql、ES、RocketMQ、Eureka
zedomi	2020/7/ 29 11:28:3 0	java, redis, mysql, jfinal, sparkjava
时光	2020/7/ 29 11:28:4 5	Springboot, SpringMVC, Dubbo, SpringCloud, Redis, Vue
唐宋元明清	2020/7/ 29 11:28:5 7	consul
唐宋元明清	2020/7/ 29 11:29:3 9	kong
小米	2020/7/ 29	机器学习

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
	11:29:5 1	
海洋之心	2020/7/ 29 11:29:5 3	springcloud、vue、ddd、工作流引擎、 elasticsearch
超人不会 飞	2020/7/ 29 11:30:2 7	dubbo、spring cloud 、netty、spring
Disappear `	2020/7/ 29 11:31:1 0	dubbo nacos xxljob mybatisplus
今宵多珍 重	2020/7/ 29 11:31:1 1	Springboot、Mybatis、Mysql、RabbitMQ、 Netty、Redis、VUE
L	2020/7/ 29 11:32:0 0	vue+elementui+node
木瓜 22	2020/7/ 29 11:32:1 2	spring、springmvc、springboot、mybatis、 mybatis-plus、redis、mongodb、mysql、 springcloud 全套组件、rabbitmq、shiro、 spring security、xxl-job、阿里云 OSS 或腾讯 云 OSS
Jack	2020/7/ 29	java spring spring boot spring cloud

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
	11:32:16	
感谢 郭嘉 PMP®	2020/7/29 11:32:37	dubbo, zookeeper, springboot, springcloud netflix, springcloud alibaba, apollo, cat, e(f)lk, mongodb, mysql, redis, mycat, openresty, prometheus, skywalking, rocketmq, rabbitmq, docker, jenkins, nexus, k8s, kubesphere
陈辉	2020/7/29 11:35:38	docker
@Violet	2020/7/29 11:36:20	zipkin, kibana, skywaking (没用过)
王凡 01	2020/7/29 11:36:48	spring boot
王泽东	2020/7/29 11:36:54	boost, asio, pthread, libuv, libevent
	2020/7/29 11:37:17	groovy
summer	2020/7/29	springcloud, springboot, dubbo, docker

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
	11:40:1 2	
弓長宏	2020/7/ 29 11:42:0 8	swoole
zedomi	2020/7/ 29 11:43:5 2	java, redis, mysql, nginx, sparkjava
红雷	2020/7/ 29 11:51:3 6	spring/dubbo
唐宋元明清	2020/7/ 29 11:52:5 8	php
渔人码头 គិតនឹង	2020/7/ 29 12:02:4 5	springboot, vue, layui
佛祖的 jio 不能抱	2020/7/ 29 12:02:4 5	oracle, spring
Joshua	2020/7/ 29 12:05:4 9	Java springboot vue react MySQL

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
小辰	2020/7/ 29 13:11:4 1	spring cloud alibaba dubbo
星宿海	2020/7/ 29 13:18:4 7	java python mysql redis hadoop docker spring vue
chenkx	2020/7/ 29 13:51:4 6	java
米高电气 马伟鸿	2020/7/ 29 13:57:3 0	netty
T. Jax	2020/7/ 29 14:03:0 8	springboot webflux mybatis jpa redis vue uni-app
那些年、 我们不曾 知道的事 情	2020/7/ 29 14:14:5 1	rabbitmq, kafka, hadoop, angular
旅人	2020/7/ 29 15:03:0 9	Antlr
郭小白	2020/7/ 29	github

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
	16:52:25	
往南更南	2020/7/29 17:07:41	springboot
😊	2020/7/29 17:14:46	java
田小麦	2020/7/29 18:13:11	guava-eventbus
一碗小米粥	2020/7/30 9:03:31	webSocket, mq, nacos
Angel's Trumpet	2020/7/30 9:05:40	debezium
自律等于自由	2020/7/30 9:05:42	ssh+springboot+vue+redis+activemq
Waiting	2020/7/30 9:06:35	springcloud alibaba 全家桶 springboot redis nginx vue mybatis-plus emqx mqtt netty rabbitmq
Shing	2020/7/30 9:08:02	spring cloud alibaba, oracle, vue,

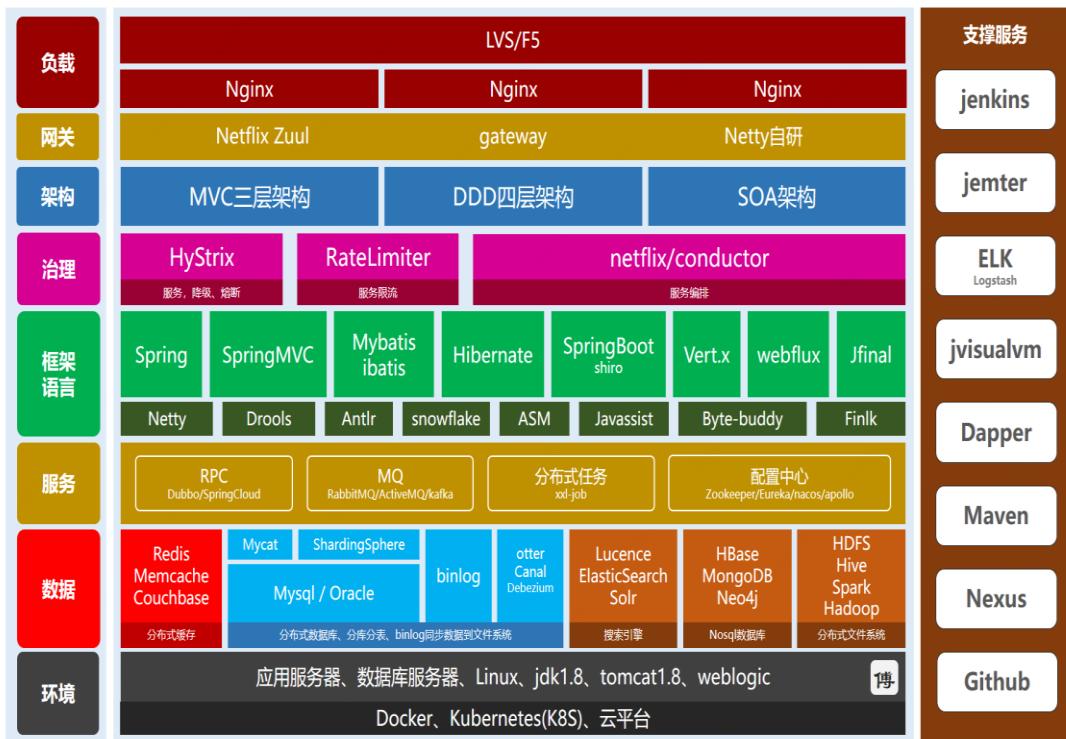
提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
路斌	2020/7/ 30 9:09:16	java, android, mybatis, spring boot
Sniper	2020/7/ 30 9:09:19	jvmti
向北	2020/7/ 30 9:10:47	ssm, springboot, layui, redis, docker.
.	2020/7/ 30 9:13:12	k8s
小傅哥 bugstack. cn	2020/7/ 30 9:14:47	大规模分布式系统的跟踪系统; dapper、 Zipkin、pinpoint、appdash、cat、hydra、鹰 眼、oneAPM
咖啡八宝 粥	2020/7/ 30 9:16:21	Java、Dubbo、MySQL、Spring、Redis、Maven、 Git、Nginx、C、Memcached
小傅哥 bugstack. cn	2020/7/ 30 9:17:24	基于 mysql binlog 的数据同步软件; otter
七号公路	2020/7/ 30 9:18:19	Vert.x
lhh	2020/7/ 30 9:21:06	nutz、

提交者 (自动)	提交时间 (自动)	你用过的技术栈(必填)
梦与孤独	2020/7/ 30 9:21:09	springboot + vue, Zuul 网关
可以的	2020/7/ 30 9:23:01	ssm、springboot、springcloud、redis、rabbitmq、mysql、mybatis-plus、springcloudalibaba、docker、layui
清风徐来	2020/7/ 30 9:25:24	spring boot+mybatis+node.js+vue+kotlin
韩俊臣	2020/7/ 30 9:40:08	ssm+oracle+redis+nginx
龙图腾	2020/7/ 30 9:50:44	spark
韶华如梦	2020/7/ 30 10:05:0 4	Rocketmq
小傅哥 bugstack. cn	2020/7/ 30 15:12:5 1	JVM 监控工具: jprofiler, perfino, Yourkit, Perf4j, JProbe, MAT

二、汇总技术架构图

在技术汇总中，可以看到有一些是自己常用的，也有一些是当前工作经历下没用接触过的。那么这些以上的技术栈，你有想过他们是怎么互相配合，组装出一张技术架构图吗？每一个地方用什么技术承载，这也可以考虑自己的日常开发中，都有哪些技术来支撑你们整个技术框架。

架构图



- 这是一张把服务端开发涉及的技术栈汇总出的一张技术架构图。
- 当然技术栈内容绝对不止局限于此，还有很多其他的框架，可以被替代，只不过这些是比较常用的。
- 这些构成你日常开发的整张技术图，可能有些是没用接触的，但这样就可以很好的让你去补充自己的盲区。
- 以上这张图的PPT已经汇总到面试手册中，可以在公众号：**bugstack 虫洞栈**，回复**下载**，把得到的**链接**打开，找到**ID:19**进行获取。

1. 负载

1. LVS 的英文全称是 Linux Virtual Server，即 Linux 虚拟服务器，主要用于多服务器的负载均衡。
2. F5 是负载均衡产品的一个品牌，除此以外还有，Radware、Array、A10、Cisco 等。
3. Nginx 就比较常见，它是由 C 语言编写的，是一个高性能的 HTTP 和反向代理 web 服务器，同时也提供了 IMAP/POP3/SMTP 服务。

2. 网关

1. 使用过 SpringCloud 的小伙伴都用过 zuul，或者公司内部自研，以及把 RPC 接口转换为 Http 接口的一种服务。

2. 为了方便客户端调用微服务，所以设计出了网关。在微服务实例地址发生改变的情况下，客户端调用服务要能够不受影响。
3. 网关可以完成的功能：路由、反向代理、日志记录、权限控制、限流、切量、黑白名单等

3. 架构

1. 大家最先接触的架构基本都是 MVC，后来进入互联网企业开始逐渐有了更加复杂的分层处理，以及接入了 RPC 和网关。
2. 随着 SpringBoot 的兴起，DDD 领域驱动设计下的微服务逐渐起来了，这种四层架构是一种设计理论，以领域为中心建模开发。
3. SOA 是面向服务的架构模型，它将应用程序的不同功能单元（称为服务）进行拆分，并通过这些服务之间定义良好的接口和协议联系起来。

4. 治理

1. 在服务的治理中一般包括：熔断、降级、限流和服务编排等。
2. 这里的思想是为了进行统一管理控制，各业务系统都是一个个自服务，最终提供给编排系统进行管理。

5. 框架语言

1. 这一层就是大家日常开发的技术语言层，用到各个技术栈来满足开发需求。
2. 比如框架类；Spring、SpringBoot、Mybatis 等。
3. 同时这里也会涉及到很多的盲区技术，因业务开发的特性不同，所需要的技术栈也会不同，为了不同的业务场景会引入不同的技术方案进行处理。比如；Drools、Snowflake、Finlk 等。

6. 服务

1. 这一层是我们在开发过程中用到的组件涉及的中心服务类配置。
2. 比如 Dubbo 以及它的注册中心、MQ 以及它的平台服务、分布式任务和配置中心等

7. 数据

1. 数据是整个系统的价值体现，因业务的体量发展从单库单表到分库分表，从数据查询到文件系统，再到各类搜索引擎的使用。
2. 同时在 DB 数据的同步过程中一般会使用基于 binlog 的 otter 进行同步 ES+Hbase 操作。
3. 以及 Neo4j 是一个高性能的，NOSQL 图形数据库，它将结构化数据存储在网络上而不是表中。它是一个嵌入式的、基于磁盘的、具备完全的事物特性的 Java 持久化引擎，但是它将结构化数据存储在网络(从数学角度叫做图)上而不是表中。

8. 环境

1. 在项目开发完成后一般会部署到自己的服务上或者云服务，以及选择不同的服务厂商。
2. 再比如现在比较火的 Docker 和 K8S，虽然不是日常开发的一部分，但也可以扩展学习了解。

9. 支撑服务

1. 这一部分包括的是：[部署](#)、[压测](#)、[日志](#)、[JVM 监控](#)、[业务全链路监控](#)、[Maven](#)、[Nexus](#)、[Github](#)以及其他需要的支撑功能。
2. 除此之外还有一些工具类的软件，比如：[IDEA](#)、[navicat](#)、[Xshell](#)、[XMind](#)、[Visio](#)等，熟练使用工具也是提升开发效率的最有效方式。

三、总结

- 在[《面经手册》](#)中，我们开篇介绍了面试官会问你啥的一个总结概述，以及到本篇介绍了技术栈盲区，通过这样两个章节的内容，可以让同好技术的小伙伴，有一个全局的认知，之后我们再去逐个攻破。
- 面试只是搂草打兔子，学习才是主干路线，千万不要以为学了什么神秘大招或者洗脑长文就能所向披靡，那都是凑巧的小概率事件。
- 我为技术用一年时间积累出来整套圈子；[推文的公众号](#)、[沉淀的博客](#)、[资源的 Github](#)、[交流的技术群](#)、[分享的技术圈子\[虫洞技术栈\]](#)、[推广的各大平台](#)，欢迎加入这个生态技术圈，每个人都有自己擅长的技术方向，互相学习，共同进步。

第 3 节：简历该怎么写

工作两年了目前的公司没有什么大项目，整天的维护别人的代码，有坑也不让重构，都烦死了。荒废我一身技能无处施展，投递的简历也没人看。我是不动物园里的猩猩，**狒狒了！**

我要加班，我要 996，我要疯狂编码，求给我个机会...

在程序这条职业发展的道路上，如果想专心走技术，并不断提升自己。那么，选择进入一个有挑战项目和一个可以跟随学技术的人，是非常重要的。而这样的资源和人脉基本还是来自一些较大型的公司，如果在前两年因为学历或者某些原因没有能进入，那么在 2 年后还是可以试一试的。

至少，你敢走出来，愿前程美好皆因你不断拼搏！

接下来我们看一份小伙伴的简历，并对内容和排版上进行优化。以及整理相应的面试点做准备，帮助这位小伙伴尽早找到心仪公司。

面试，也是一次相亲。不认识你之前我要看脸(简历)、见了面我要走心(我来问你来答)。那么简历写不好，相应的内容又没做好准备，你不失败谁失败。

一、我的简历都石沉大海

这是一份模拟真实的简历，如下：



由 Xnip 截图

Java 软件工程师-求职简历

个人基本信息

姓 名：谢飞机
性 别：男
出生年月：19960708
联系方式：1520120****
微信：和手机同步

工作年限：2 年
学 历：本科
毕业院校：长春理工大学
专 业：软件工程

求职意向

工作性质：全职
到岗时间：即时

目标职能：JAVA 软件工程师
目标地点：长春

技能描述

- 前端框架 Vue.js Jquery Bootstrap
- 后端框架 Netty, SpringBoot Spring SpringMVC Mybatis JFinal
- 数据库 MySQL
- 编写代码有封装，分层，职责单一的思想
- 使用 Netty 实现过分布式 Raft 算法 主要实现模块是选举，心跳。选举难点主要在于选举时消息的同步执行(先到先得)，每个节点一轮只能投一票，轮数低者不能向轮数高者投票的细节等。

工作经验

2018 年 10 月-2020 年 4 月 长春虫谷信息有限公司 java 后端工程师

工作内容：按照技术经理的功能要求，参与设计并实现

项目经验

项目一：抽奖系统

开发环境：Eclipse+Windows10+JDK1.8+Tomcat7.0+Mysql5.5 +SVN
软件架构：Spring + SpringMVC+ Hibernate
责任描述：前端页面+逻辑 后端提供接口

项目二：数据采集系统

开发环境：IDEA、JDK1.8.、Git 、MySQL8
软件架构：Netty+Spring+RocksDB
项目描述：分布式系统(具体不能说)

这份简历有哪些问题：

1. 简历格式不规整，由于是 `word` 格式在不同版本下展示可能有不兼容的问题。所以非常建议写成 `pdf` 格式的简历。
2. 个人信息联系方式中没有写邮箱，因为在面试前会发一些约面邀请函到个人邮箱中。
3. 技能描述缺少不完整并缺少核心内容，及时简历通过，面试官与你也没有太多的话题。
4. 项目经验描述不完整，并没有在项目中体现出个人的能力以及工作重点。这些都会影响与面试官的场景交流，如果你不留下可以聊的点，那么久等着被动的问。

以上就是对这份简历的一些基本修改项，虽然是模拟真实的，但是这可能也几乎是大部分人的简历样式了。

二、修改后电话约面不断

修改后的简历如下，如果需要模板可以关注公众号：[bugstack 虫洞栈](#)

Java 软件工程师-求职简历

个人信息

- 姓名：谢飞机
- 性别：男
- 工作年限：2年
- 学历：本科
- 专业：软件工程
- 院校：长春理工大学
- 联系电话：1520102****
- 通信邮箱：xiefeiji@qq.com

求职意向

- 目标职能：Java 软件开发工程师
- 目标地点：长春
- 工作性质：全职
- 到岗时间：2周以内

技术栈

1. 技能 Java、Mysql、Redis、Netty、Vue.js、Jquery
2. 框架 Spring、SpringMVC、Mybatis、SpringBoot、JFinal
3. 组件 Dubbo、ZK、RabbitMQ
4. 工具 IDEA、Maven、GIT、postman
5. 部署 Linux、Tomcat

深入学习和使用 Java 语言以及相应的服务技能，对部分核心组件以及源码有过了解，其中在 Netty 服务设计上有丰富的经验。并且具备面向对象编程思想，可以有效合理的设计代码分层并快速实现功能。

工作经验

- 2018年8月 - 至今，长春虫谷科技信息有限公司
 - 职位：java后端工程师
 - 职责：按照技术经理的功能文档，参与设计并实现功能交付
- 2017年6月 - 2018年7月，长春走码科技有限公司
 - 职位：实习开发工程师
 - 职责：维护项目系统文档、梳理组织会议、编写简单功能服务

项目经验

项目一、营销抽奖系统

- 开发环境：Idea+Mac+JDK1.8+Maven3.0+Mysql5.7+Redis+Dubbo+Tomcat7.0+Git
- 系统架构：SSM，三层架构，分布式部署
- 项目描述：这是一个需要整合到目前电商系统中的一个服务模块，主要用于营销抽奖，对用户促活拉动消费。在抽奖服务中使用到了核心模块功能有：防刷、黑白名单、秒杀、发货等。系统在秒杀场景中进行多次优化，目前压测指标为 QPS=1500，满足业务发展诉求。
- 工作职责：
 1. 负责开发系统服务中的对外提供的统一标准接口，并与外部联调。
 2. ERP运营后台开发，针对产品功能不断完善运营操作平台。以及相应的活动配置校验，保证每一个活动的准确性。
 3. 参与优化秒杀功能逻辑的设计和开发，对分布式场景下的自动分配库存锁有丰富经验。

项目二、数据采集服务

- 开发环境：Idea+JDK1.8+Dubbo+Netty+Mysql+Git+Tomcat7.0
- 系统架构：Spring+Netty+Mybatis+JavaFX，异步分层事务驱动框架
- 项目描述：采集机械设备客户端数据，定时分段回传消息。用于系统的对机房信息监测；温度、湿度、指标、以及系统各项信息。最终对数据进行加工展示到云平台。
- 工作职责：
 1. 设计基于 Netty 编写网络通信模块，对消息协议定制和封装编码解码器。以及对数据传输过程中的业务流程处理，例如：半包年包、弱网分段传输、数据流切块等
 2. 开发采集后的数据通过 RPC 框架回传到服务端，并做最终的逻辑处理。
 3. 编写工具类以及接口文档的更新，统一维护 3 个大区域的对接工作。

个人评价

1. 有较强的学习和解决问题的能力，善于通过一些系统问题总结分析根本原因，热爱于技术挑战。
2. 在工作中可以独立承担相关职责，可以较强承受工作压力并愿意不断的提升自我。
3. 性格开朗、积极乐观，可以精神饱满的投入到工作中。有着良好的团队协作能力和责任心，对工作有强烈的使命感。

好的简历是一次美好 **相亲** 的开始，接下里就是对 **相亲** 事项的准备。

三、简历好也要做好准备

虽然面试过程基本会问的点相差不太多，但是结合简历自身以及不同的面试官，都会聊出很多不一样的东西。所以不能完整回答出面试官的题目也没有关系，不用过于紧张，只要你能回答出 **80%** 以上基本都可以通过到下一轮面试。那么，结合上面的简历，大概会有如下一些问题点可以聊；

1. 技术栈

1.1 Java 基础

这种面试题基本不固定，主要考察你对 **java** 基础的学习程度和理解能力，比如；

1. byte 占几个字节
2. for 循环与 foreach
3. java8 的新特性
4. hashmap 的实现原理
5. 线程池和锁的使用和原理
6. 设计模式和面向对象

1.2 Spring

1. Spring 的好处
2. AOP 与 IOC
3. Spring 注解
4. SpringBean 加载过程
5. SpringBean 生命周期
6. Spring 中事务

1.3 Mybatis

1. Mybatis 的好处
2. Mybatis 的缓存
3. 如何进行分页
4. 插件是怎么运行的

5. #{} 和 \${} 的区别是什么
6. Mybatis 是否支持延迟加载？

1.4 Rpc 框架 – Dubbo

1. 通信模型是什么样
2. 怎么暴露服务的
3. 数据序列化方式
4. Rpc 框架还有哪些，有什么优缺点

1.5 Redis

1. redis 基本常用的方法
2. 分布式加锁
3. 主备同步
4. RDB 与 AOF
5. 什么是缓存击穿
6. 集群

1.6 RabbitMQ

1. MQ 的好处
2. MQ 发送失败怎么办
3. MQ 的应用场景
4. 广播

1.7 Netty

Netty 应用在各个框架中非常广泛，例如 Dubbo、MQ 等，属于一块核心技能。

1. Netty 有什么好处
2. 多路复用
3. 流量切块
4. 数据整形
5. 编码也解码器
6. 半包粘包

2. 源码理解

1. 对于一些 java 中的核心方法比如 hashmap、ArrayList 等的实现
2. Spring 一些源码
3. Mybatis 实现上的一些源码
4. 以及个人简历中写了的内容

3. 工具使用

1. linux 常用命令，查日志、线程
2. java 一些命令:jstack、jmap、jstat、javap 等
3. git 如何建分支（你说我就用过工具，显得就不专业了）
4. 监控工具、部署工具、压测工具等等，是一个知识面的拓展

4. 项目经验

项目方面完全依照个人在简历中提到的项目进行询问，但最好在项目中写出一些可以被拿来聊的点。比如这项目是做什么的、个人负责了那块、核心流程是什么、你解决过什么样的问题等等。

如果是一些金融或者订单等，会有专门的面试内容，比如：清分、结算、幂等、多支付、对账、秒杀、发货、分布式锁、一致性等等。这些东西是编不来的，需要亲身经历，否则问不了几个问题，你就漏了。

如果说你还没有一个什么像样的项目，那么可以去 [github](#) 寻找，并学习。

5. 个人发展

这个就是一个比较开放性的问题了，主要是面试的一个收尾，也收复心情。只要不聊的太放飞自我，这里不是问题。

四、系统的学习才能变强

面试成功后和相亲走到结婚一样，日后的日子还是需要靠实力去过。

研发人员的成长终究是需要系统的需求，无论从算法、源码、技术栈以及项目上，要不断的深挖、探索，我一直相信只要努力做一件事并坚持下来，一定会收获成绩。下面是[小傅哥](#)学习成长博客记录，可以 pc 访问：<https://bugstack.cn>

五、总结

- 可能会觉得我是在包装自己的吗？是的！只不过你是包的更加实，并补充自己的短板。通过面试事项的梳理，知道自己哪里不足并去完善。并不是背了几个题就能通过面试，而是为面试知道自己的短板尽快赶上。
- 可能有人都已经工作几年了，简历还是写的一塌糊涂，word 格式发来打不开的都有（怀疑你的个人电脑多久没打开了也不更新，平时也不写代码了）。
- 认真对待自己的将来，就是做好现在。从没有那个人一直努力还没有成绩，如果暂时没有就在努力试试。
- 以上的简历模板，可以通过关注公众号：[bugstack 虫洞栈](#)，回复 简历 进行获取。

第 4 节：大厂都爱聊啥

Java 学到什么程度可以找工作？

最近总看到类似这样的问题，也有一些工作 3 年左右的小伙伴问小傅哥，该怎么进大厂。其实你说 Java 学多少可以找到工作，主要看你想在哪个城市找、找什么样的公司、找什么样的待遇。因你的要求高低不同，你遇到的面试要求也会随之改变。

因此，为了让大家更清楚的看到学多少 Java 能找工作，我抽取了北京头部互联网公司的大量职位招聘要求，分析出一份可以让你明卷面试的考点。看过都说学习有方向、面试也不慌！

一、互联网公司都分布在哪里

知己知彼，百战不殆，先看看有哪些互联网公司，都分布在北京的哪些地方，也能方便你，面试不迷路、跳槽不辛苦。筛选了部分公司，还有很多不一一列举了！

1. 常见大厂



图 11-1 常见互联网大厂，筛选 20 家

- 没有排名，随机筛选，随机排序

- 公司包括：华为、联想、新浪、百度、小米、58 同城、搜狗、爱奇艺、腾讯、去哪儿网、美团、饿了吗、汽车之家、字节跳动、当当网、CSDN、亚马逊、京东、360、滴滴

2. 地理位置

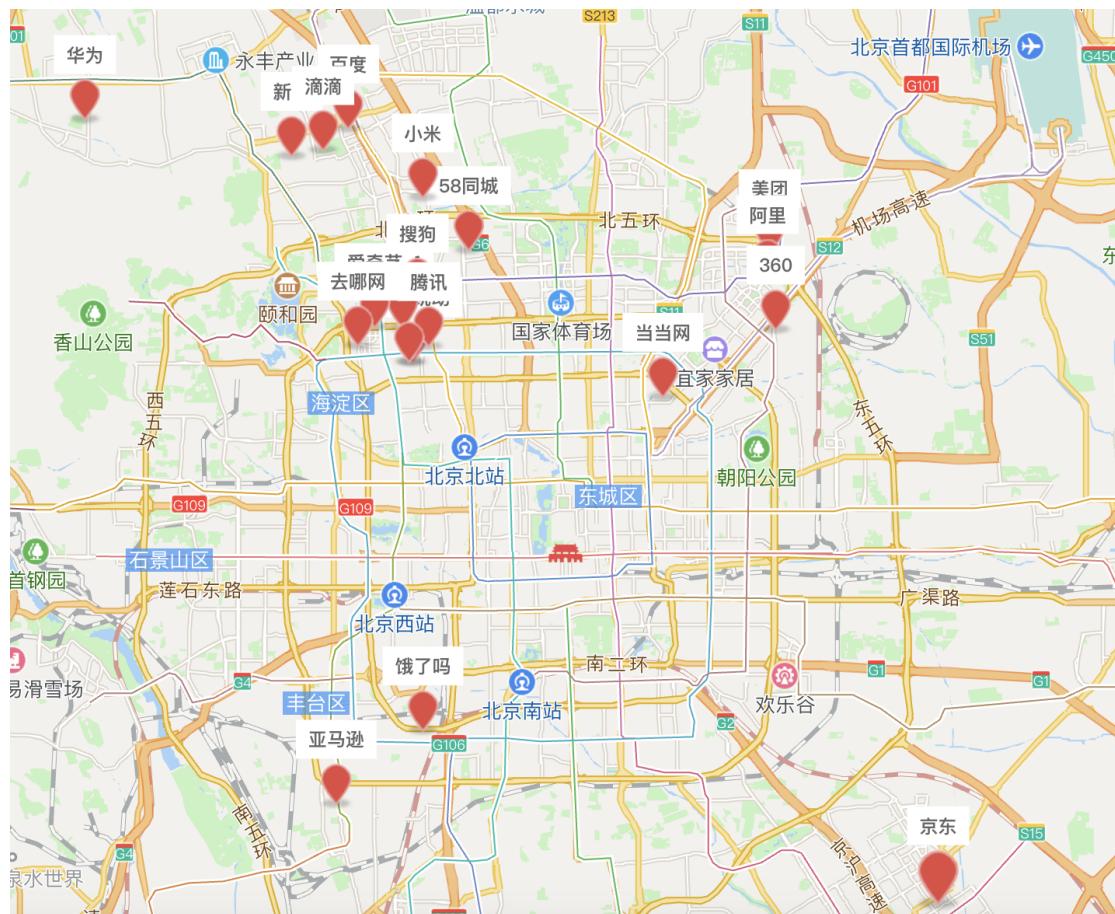


图 11-2 互联网大厂地理位置分布

- 从图上可以看到大部分互联网公司都分布在北边，让人怪不好意思的，集中在一块挺好，下楼吃个饭就跳槽了。
- 就我自己而言更喜欢靠边一点的公司，因为租房便宜、不用挤地铁、不用把时间浪费在路上、不用听马路的嘈杂。

二、什么样的技术能进入大厂

1. 你的简历

可能很大一部分 1~3 年找工作的小伙伴，只是按照模板填写好简历就完事了，很少考虑公司都需要什么、自己的职位是否匹配。

但你可能忽略了，你的这份简历才更多的决定了你会遇到一个什么样的公司、什么样的面试官、什么样的考题。最终决定你与这家公司的匹配的程度。

在与很多小伙伴沟通中发现，其实很大一部分程序员都不会写简历的，或者说写不好简历。好像是有话说不出来，或者是不知道该把这些话说在哪。一份简历主要得体现出你个人的信息、技术栈广度和深度、项目经验以及最后一块拓展内容。而这份简历想达到最终的效果，也就是拿 Offer。那么一定要给面试官挖坑，当然这个坑不是真坑。而是你要在简历中突出自己的优势项、技术亮点、优秀经历，也同时在这些点中留出技术话题，让面试官可以和你有的聊和撩。

但如果说你胡乱写简历，说自己懂 HashMap。那面试官来劲了，问你：`Hash` 为什么用 `31` 计算、扰动函数的作用是什么，以及它可以被应用在哪些地方、负载因子嘎哈的、`HashMap` 是开放寻址还是拉链寻址、链表什么时候树化以及迁移数据算法是什么、`2-3` 树和红黑树有什么关系等等，你不晕才怪，也不能给面试官留下好印象。

2. 大厂考题

以下这部分考题分析数据是通过抽样的方式，从 Boss 直聘中选取六个互联公司，每个公司找 3~5 个，工作 1~3 年岗位应聘要求，从中分析各面试考点综合汇总。
样例数据

- 阿里：
https://www.zhipin.com/job_detail/a651f649367bd40c1nR82NW_EIVW.html?ka=comp_joblist_6
- 百度：
https://www.zhipin.com/job_detail/ee5fe74c428cae881nR729q6GFFX.html?ka=search_list_jname_23_blank&lid=7BM4dKAQnha.search.23
- 腾讯：https://www.zhipin.com/job_detail/1e5e940eba4d86131Xdy2t-8FFQ~.html?ka=comp_joblist_2
- 字节：
https://www.zhipin.com/job_detail/035b480e47bbcf833x82Nm9EFU~.html?ka=search_list_jname_1_blank&lid=7TNhCRKNyz.search.1
- 美团：
https://www.zhipin.com/job_detail/860af0510dc7600b3nx62dq5FIY~.html?ka=search_list_jname_5_blank&lid=7BYHTvIbgTr.search.5
- 京东：
https://www.zhipin.com/job_detail/2f6609c137365cb51nR50tq7F1VR.html?ka=search_list_jname_1_blank&lid=7U1m8knPpTh.search.1

2.1 阿里、百度、腾讯

阿里			百度	腾讯
薪资待遇	25-35 K · 16薪		20-40 K · 15薪	20-40 K · 16薪
基本功底	Java基础扎实		Java基础扎实	算法
常用技术	Spring/SpringBoot Mybatis/batis Dubbo MQ(kafka) Git Redis Hive Elasticsearch Hbase Mysql, Oracle		Spring SpringMvc Hibernate/Mybatis SpringSecurity SpringData JPA Linux操作系统、Shell脚本 有基础的运维能力 Docker k8s	Spring SpringBoot Mybatis/batis Mysql Yii
	JVM 原理 多线程 理解IO 熟悉底层中间件、分布式技术 对于用过的开源框架，了解到它的原理和机制		至少精通一种RPC框架，Dubbo、ICE 精通Zookeeper	深入理解语言特性 对框架原理有一定的理解
技术深度				• 公众号：bugstack虫洞栈
技术经验	掌握设计模式、良好的代码设计能力 框架设计能力，能编写高质量简洁的代码 多线程及高性能的设计编码和性能调优 熟悉分布式系统的设计和应用 有互联网系统开发、高可用、高性能的设计经验 有一定的SQL性能优化经验		熟悉分布式缓存框架 有Redis集群部署经验 熟悉SOA设计与开发 对数据结构、算法设计、系统架构设计等，有较深刻的理解 嵌入理解分布式软件架构思想 熟悉面向对象思维和开发过程，熟悉相关建模工具 熟悉MySQL，以及调优技巧 对OOA、OOD、OOP等思想有较深入的理解和很强的应用能力 熟悉微服务架构并进行过微服务改造	熟悉常用数据结构和算法 熟练掌握SQL语言 对业界KV组件有一定了解 熟练掌握HTTP协议 具备Linux下的网络编程经验 对多线程技术、异步、并发有较深理解 熟悉MySQL应用开发、配置、维护、性能优化经验 有良好的数据库设计能力和SQL编写能力
学习能力	拥抱开源，喜欢阅读开源代码 热爱技术，具备较强的学习能力		活跃的Github用户，参与过开源项目	
工作能力	具备责任心、耐心、细心的品质 能解决复杂问题 具备非常好的推动和执行能力 业务理解和学习能力强 愿意深入了解业务知识，并能敏锐的发现业务痛点 善于沟通、团队协作 工作认真、严谨		工作积极主动、抗压 认真负责，善如团队合作	有较强的责任心和沟通协作以及抗压能力 有良好的分析问题和解决问题的能力

图 11-3 阿里、百度、腾讯，1~3 年招聘要求梳理

每个公司的每个职位要求会略有不同，所以不能一概而论，某一行没有写某项技术点也不能代表什么。以上更多的是参考以及自己在面试求职时可以按照这种方式进行梳理。

- 阿里，在技术上会更加希望你有深度和广度，也善于把技术能应用到项目中，并有一定的学习能力。同时在工作中，要有责任心、沟通能力和解决问题的落地的能力。
- 百度，同样希望可以精通一些框架的深层次内容，有一定的技术经验，更偏向于落地技能。同时也希望你是爱学习的面试者，最好有 Github 相关内容。工作中积极、主动、抗压，认真，善于沟通。
- 腾讯，除了基础语言学习外，要有一些扩展，同时要深入理解语言特性。这可能和腾讯本身是用 C、C++ 有关，要知其然，知其所以然。同时希望在数据结构和算法上有一定的了解和认知，也可以在工作中有责任心、抗压能力以及问题分析和解决能力。

2.2 字节、美团、京东

	字节跳动	美团	京东
薪资待遇	30~60 K · 17薪	20~40 K · 15薪	20~40 K · 14薪
基本功底	Java基础扎实 算法、数据结构	Java基础扎实 熟悉数据结构以及算法	Java基础扎实
	NoSQL 消息队列 WEB组件	Git Maven Spring体系 Servlet容器 Guava HttpComponents	Spring系列 Mybatis RPC Redis MQ Mysql ZooKeeper Linux常用命令、Shell脚本 Elastic Search
常用技术			
技术深度	深刻理解计算机原理 有良好的数据结构和算法基础 深入理解JIO原码 了解JVM调优 熟悉数据库原理 熟悉HTTP协议 熟悉MQ、AIO 字节码增强技ASM、Javassist、Byte-buddy 多线程、反射 了解数据库事务隔离级别、索引 对NoSQL有深入研究 了解常用中间件不限于消息队列、分库分表、任务调度等 对RPC、SOA、服务治理有相应了解 熟悉Hash、Paxos等理论和算法	对JVM原理有深入的了解 深入理解IO 多线程、集合 常用框架、RPC、MQ、缓存，了解其原理和实现机制 精通Spring、Mybatis源码 熟悉MySQL、Hbase等数据库技术 熟悉Java OOP和并发编程技术	对JVM原理有深入的了解 深入理解IO 多线程、集合 常用框架、RPC、MQ、缓存，了解其原理和实现机制 精通Spring、Mybatis源码 熟悉MySQL、Hbase等数据库技术 熟悉Java OOP和并发编程技术
技术经验	高并发服务设计和实现经验 优秀的设计和代码品味 扎实的编程能力 负责过大流量高并发系统设计	熟悉常见设计模式 MySQL数据库性能优化 分布式系统设计经验 大型互联网领域开发经验 用于与年限相关的技术广度和深度 参与过大兴分布式系统开发，熟悉CAP	有较大型分布式、高并发、高负载、高可用的系统设计开发经验 熟悉分布式部署以及相关技术 熟悉常见设计模式，灵活运用设计模式 熟悉DDD开发模式 有领域建模实践经验 具有良好的编码风格和习惯 具有良好的逻辑思维能力
学习能力	有较强的学习能力和求知欲 好奇心和进取心，能及时关注和学习业界最新技术	有技术钻研精神 有GitHub开源项目	阅读过开源框架代码 在GitHub、码云有开源项目或者贡献 个人技术博客
工作能力	有良好的产品意识 积极乐观、责任感强 工作认真细致，有良好的团队沟通协作能力 有独立解决问题的能力 有良好的沟通能力和业务理解能力 承担责任、难点的技术攻坚	有较强的逻辑思维，善于分析，归纳，解决问题 做事有责任感和主动性 面对复杂业务问题，可以从技术和业务多角度推进并落地 持续学习与创新，用技术解决业务问题，追求卓越，精益求精 建设高性能、高可用系统 难点攻克，技术输出，知道初级工程师，促进团队共同成长 具有敏捷开发经验，实践过持续集成自动化	思路清晰、善于思考 能独立分析和解决问题 熟悉软件工程、编码规范 有过过程控制意识、风险意识 具备良好的沟通能力、团队合作能力 能在巨大工作压力下保持良好的工作态度

图 11-4 字节、美团、京东，1~3 年招聘要求梳理

分析完上面三家公司，再看看这三家互联网对应聘者的要求。注意数据依旧是抽样，所以面试者在投递简历时，一定要自己拆解分析

- 字节，更注重 Java 基础、算法、数据结构，同时对于常用的技术要有一定的了解深度。对代码方面要有良好的设计和代码品味追求，同时希望你关注业界最新技术，有好奇心和进取心。工作中，有产品意识。对于研发来说，产品意识很重要
- 美团，美团的技术科目属于比较面的比较广和深的，尤其是字节码编程技术，在其他一些招聘中是没有看到的。除此之外更希望你有一定的学习能力，参与过 Github 开源项目，有技术钻研精神。在工作中，有较强的思维逻辑，难点攻克，复杂问题推进落地，责任感等。这可能也和美团的技术氛围有关，他们的技术博客做的也不错。[百度搜美团技术，<https://tech.meituan.com/>](https://tech.meituan.com/)
- 京东，各家公司都非常注重 Java 基本功，这些面试题可能不难但也很难。另外在常用框架和技术深度上要有一定了解，尤其是各个框架的原理和实现机制，如果你能自己动手写一个，那么会更好。另外在技术经验上，可以有分布式、高并发等经验，也可以非常用的运用设计模式，编写出不错的代码。同时希望你有一定的学习能力，博客、开源代码、Github、Gitee 等。工作中，有过过程控制意识、风险意识以及良好的沟通和解决问题的能力。

3. 考点总结

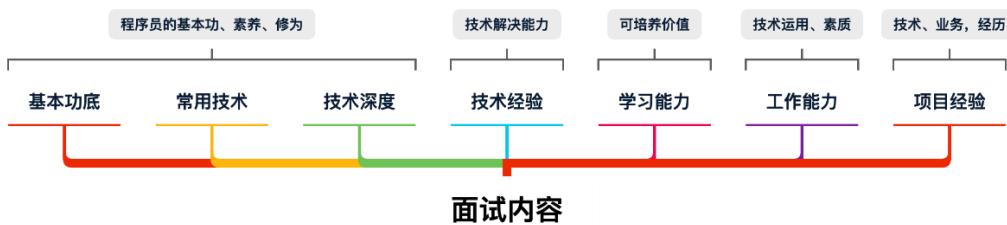


图 11-5 面试官考点总结

综上，各家公司的招聘要求，梳理出七个方向的考点，包括：基本功底、常用技术、技术深度、技术经验、学习能力、工作能力、项目经验。

- **基本功底**，是一个程序员的主科目语言的学习程度的一个基本考察，这部分内容需要平时大量积累和总结。否则一本简单的 Java 书很难全部给你讲透彻，因为 Java 中包括了太多的内容，远不止 API 使用。
- **常用技术**，这个聊的是你的技术广度，和岗位技术匹配度。比如需要用到过 RPC，那你用过 Dubbo。如果你的公司暂时用的技术不多，或者还是处于单体服务，那么需要自己补充。
- **技术深入**，除了技术广度接下来就是技术深入，在你常用的技术栈中，你有多了解他们，了解源码吗、了解运行机制吗、了解设计原理吗。这部分内容常被人说是造火箭，但这部分内容非常重要，可以承上启下的贯穿个人修为和薪资待遇。
- **技术经验**，什么是技术经验呢？这是落地能力，除了你可能认为上面一些是纸上谈兵，是造火箭。那么接下来这部分内容就是你是否真造过一个火箭，真完成过一个难题。所以这部分是从结果证明，不是你会什么，而是你做过什么。
- **学习能力**，作为程序员你是否保持热情，是否依旧在积极努力的关注技术，是否为自己的成长不断添砖加瓦、是否还有好奇心和较强的求知欲。一般会从这里看你是不是一个真正的 Coder!
- **工作能力**，以上的种种能力，最终要体现到工作上，要能看出你的交付能力。否则即使你再优秀，也不能把你当成一个吉祥物。工作能力的体现，才是真的为团队、为部门、为公司，贡献价值的。
- **项目经验**，这项内容会根据不同公司的不同业务线而不同，就像你懂交易、支付，那么面试花呗、借呗、白条等工作岗位就会很吃香。

三、总结

- 面试也是一场有准备的**战斗**，知己知彼才能游刃有余。面试怎么面主要是看简历怎么写，最终是你来主导面试，还是被主导，更多也是依赖于你的技术身家。

- 任何时候都需要主动学习、有技术眼光和魄力，既能吹得了造火箭的牛、也能落地出实际的产物、技能帮公司实现价值，也能让自己有一定的收入。才是你应该永久追求的目标，和突破瓶颈的价值。
- 少一些躁动、少一些不安，多一些沉稳、多一些沉淀，只要你愿意积累就一定会突破瓶颈，都是这条路上的打工人，不要总让自己的大脑被别人牵着走，也不要活在别人嘴里。[奥利给](#) 阅读原文，进入知识宝藏！

第 2 章 数据结构和算法(11 节)

第 1 节：HashCode 为什么使用 31 作为乘数？

在面经手册的前两篇介绍了《面试官都问我啥》和《认知自己的技术栈盲区》，这两篇内容主要为了说明面试过程的考查范围，包括个人的自我介绍、技术栈积累、项目经验等，以及在技术栈盲区篇章中介绍了一个整套技术栈在系统架构用的应用，以此全方面的扫描自己有哪些盲区还需要补充。而接下来的章节会以各个系列的技术栈中遇到的面试题作为切入点，讲解技术要点，了解技术原理，包括：数据结构、数据算法、技术栈、框架等进行逐步展开学习。

在进入数据结构章节讲解之前可以先了解下，数据结构都有哪些，基本可以包括：[数组\(Array\)](#)、[栈\(Stack\)](#)、[队列\(Queue\)](#)、[链表\(List\)](#)、[树\(Tree\)](#)、[散列表\(Hash\)](#)、[堆\(Heap\)](#)、[图\(Graph\)](#)。

而本文主要讲解的就是与散列表相关的 [HashCode](#)，本来想先讲 [HashMap](#)，但随着整理资料发现与 [HashMap](#) 的实现中，[HashCode](#) 的散列占了很重要的一设计思路，所以最好把这部分知识补全，再往下讲解。

一、面试题

说到 [HashCode](#) 的面试题，可能这是一个非常核心的了。其他考点：怎么实现散列、计算逻辑等，都可以通过这道题的学习了解相关知识。

Why does Java's `hashCode()` in `String` use 31 as a multiplier?

Why does Java's hashCode() in String use 31 as a multiplier?

Asked 11 years, 8 months ago Active 1 month ago Viewed 144k times

Per the Java documentation, the [hash code](#) for a `String` object is computed as:

486

$s[0]*31^{n-1} + s[1]*31^{n-2} + \dots + s[n-1]$

187

using `int` arithmetic, where $s[i]$ is the i th character of the string, n is the length of the string, and \wedge indicates exponentiation.



Why is 31 used as a multiplier?

I understand that the multiplier should be a relatively large prime number. So why not 29, or 37, or even 97?

[java](#) [string](#) [algorithm](#) [hash](#)

share improve this question follow

edited Jan 3 '19 at 13:14

Mark Amery

96.1k 47 330 373

asked Nov 18 '08 at 16:39



jacobko

7,770 6 28 36

这个问题其实~~是~~指的就是，`hashCode` 的计算逻辑中，为什么是 31 作为乘数。

```
public int hashCode() {  
    int h = hash;  
    if (h == 0 && value.length > 0) {  
        char val[] = value;  
  
        for (int i = 0; i < value.length; i++) {  
            h = 31 * h + val[i];  
        }  
        hash = h;  
    }  
    return h;  
}
```



写死的固定值 31

二、资源下载

本文讲解的过程中涉及部分源码等资源，可以通过关注公众号：[bugstack 虫洞栈](#)，
回复下载进行获取{回复下载后打开获得的链接，找到编号 ID: 19}，包括；

1. `HashCode` 源码测试验证工程，[interview-03](#)
2. 103976 个英语单词库. txt，验证 `HashCode` 值
3. `HashCode 散列分布.xlsx`，散列和碰撞图表

三、源码讲解

1. 固定乘积 31 在这用到了

```
// 获取 hashCode "abc".hashCode();
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

在获取 `hashCode` 的源码中可以看到，有一个固定值 `31`，在 `for` 循环每次执行时进行乘积计算，循环后的公式如下：`s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]`

那么这里为什么选择 `31` 作为乘积值呢？

2. 来自 stackoverflow 的回答

在 [stackoverflow](#) 关于为什么选择 `31` 作为固定乘积值，有一篇讨论文章，[Why does Java's hashCode\(\) in String use 31 as a multiplier?](#) 这是一个时间比较久的问题了，摘取两个回答点赞最多的：

413 个赞的回答

最多的这个回答是来自《Effective Java》的内容；

The value `31` was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, as multiplication by `2` is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of `31` is that the multiplication can be replaced by a shift and a subtraction `for` better performance: `31 * i == (i << 5) - i`. Modern VMs `do this sort of optimization automatically`.

这段内容主要阐述的观点包括：

1. `31` 是一个奇质数，如果选择偶数会导致乘积运算时数据溢出。
2. 另外在二进制中，`2` 个 `5` 次方是 `32`，那么也就是 `31 * i == (i << 5) - i`。这主要是说乘积运算可以使用位移提升性能，同时目前的 JVM 虚拟机也会自动支持此类的优化。

80 个赞的回答

As Goodrich and Tamassia point out, If you take over 50,000 English words (formed as the union of the word lists provided in two variants of Unix), using the constants 31, 33, 37, 39, and 41 will produce less than 7 collisions in each case. Knowing this, it should come as no surprise that many Java implementations choose one of these constants.

- 这个回答就很有实战意义了，告诉你用超过 5 千个单词计算 hashCode，这个 hashCode 的运算使用 31、33、37、39 和 41 作为乘积，得到的碰撞结果，31 被使用就很正常了。
- 他这句话就可以作为我们实践的指向了。

3. Hash 值碰撞概率统计

接下来要做的事情并不难，只是根据 [stackoverflow](#) 的回答，统计出不同的乘积数对 10 万个单词的 hash 计算结果。10 个单词表已提供，可以通过关注公众号：bugstack 虫洞栈进行下载

3.1 读取单词字典表

```
1 a "n.(A)As 或 A's 安(ampere(a) art.一;n.字母 A /[军] Analog.Digital,模拟/数字 / (=account of) 帐上"
2 aaal American Academy of Arts and Letters 美国艺术和文学学会
3 aachen 亚琛[德意志联邦共和国西部城市]
4 aacs Airways and Air Communications Service (美国)航路与航空通讯联络处
5 aah "[军]Armored Artillery Howitzer,装甲榴弹炮;[军]Advanced Attack Helicopter,先进攻击直升机"
6 aal "ATM Adaptation Layer,ATM 适应层"
7 aapamoor "n.[生]丘泽,高低位镶嵌沼泽"
```

- 单词表的文件格式如上，可以自行解析
- 读取文件的代码比较简单，这里不展示了，可以通过[资源下载](#)进行获取

3.2 Hash 计算函数

```
public static Integer hashCode(String str, Integer multiplier) {
    int hash = 0;
    for (int i = 0; i < str.length(); i++) {
```

```

        hash = multiplier * hash + str.charAt(i);
    }
    return hash;
}

```

- 这个过程比较简单，与原 hash 函数对比只是替换了可变参数，用于我们统计不同乘积数的计算结果。

3.3 Hash 碰撞概率计算

想计算碰撞很简单，也就是计算那些出现相同哈希值的数量，计算出碰撞总量即可。这里的实现方式有很多，可以使用 `set`、`map` 也可以使用 `java8` 的 `stream` 流统计 `distinct`。

```

private static RateInfo hashCollisionRate(Integer multiplier, List<Integer> hashCodeList) {
    int maxHash = hashCodeList.stream().max(Integer::compareTo).get();
    int minHash = hashCodeList.stream().min(Integer::compareTo).get();
    int collisionCount = (int) (hashCodeList.size() - hashCodeList.stream().distinct().count());
    double collisionRate = (collisionCount * 1.0) / hashCodeList.size();
    return new RateInfo(maxHash, minHash, multiplier, collisionCount, collisionRate);
}

```

- 这里记录了最大 hash 和最小 hash 值，以及最终返回碰撞数量的统计结果。

3.4 单元测试

```

@Before
public void before() {
    "abc".hashCode();
    // 读取文件, 103976 个英语单词库.txt
    words = FileUtil.readWordList("E:/itstack/git/github.com/interview/interview-
01/103976 个英语单词库.txt");
}


```

```

@Test
public void test_collisionRate() {

```

```

List<RateInfo> rateInfoList = HashCode.collisionRateList(words, 2, 3, 5, 7, 17,
31, 32, 33, 39, 41, 199);

for (RateInfo rate : rateInfoList) {
    System.out.println(String.format("乘数 = %4d, 最小 Hash = %11d, 最大
Hash = %10d, 碰撞数量 =%6d, 碰撞概
率 = %.4f%%", rate.getMultiplier(), rate.getMinHash(), rate.getMaxHash(), rate.getCo
llisionCount(), rate.getCollisionRate() * 100));
}
}

```

- 以上先设定读取英文单词表中的 10 个单词，之后做 hash 计算。
- 在 hash 计算中把单词表传递进去，同时还有乘积数：2, 3, 5, 7, 17, 31, 32, 33, 39, 41, 199，最终返回一个 list 结果并输出。
- 这里主要验证同一批单词，对于不同乘积数会有怎么样的 hash 碰撞结果。

测试结果

单词数量: 103976

乘数 = 2, 最小 Hash = 97, 最大 Hash = 1842581979, 碰撞数量 = 60382, 碰撞概
率 = 58.0730%

乘数 = 3, 最小 Hash = -2147308825, 最大 Hash = 2146995420, 碰撞数量 = 24300, 碰撞概
率 = 23.3708%

乘数 = 5, 最小 Hash = -2147091606, 最大 Hash = 2147227581, 碰撞数量 = 7994, 碰撞概
率 = 7.6883%

乘数 = 7, 最小 Hash = -2147431389, 最大 Hash = 2147226363, 碰撞数量 = 3826, 碰撞概
率 = 3.6797%

乘数 = 17, 最小 Hash = -2147238638, 最大 Hash = 2147101452, 碰撞数量 = 576, 碰撞概
率 = 0.5540%

乘数 = 31, 最小 Hash = -2147461248, 最大 Hash = 2147444544, 碰撞数量 = 2, 碰撞概
率 = 0.0019%

乘数 = 32, 最小 Hash = -2007883634, 最大 Hash = 2074238226, 碰撞数量 = 34947, 碰撞概
率 = 33.6106%

乘数 = 33, 最小 Hash = -2147469046, 最大 Hash = 2147378587, 碰撞数量 = 1, 碰撞概
率 = 0.0010%

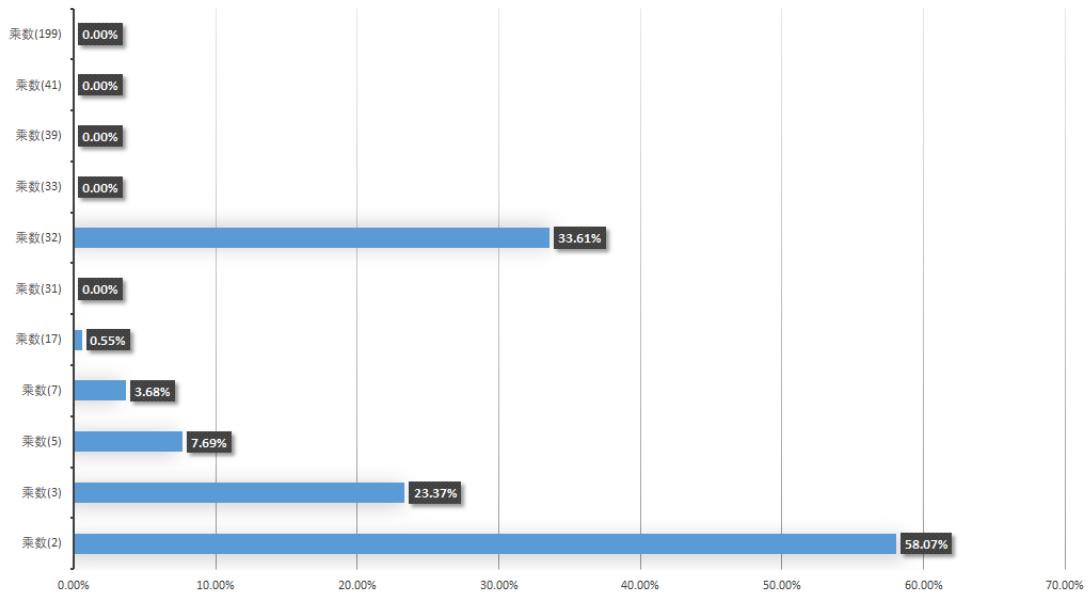
乘数 = 39, 最小 Hash = -2147463635, 最大 Hash = 2147443239, 碰撞数量 = 0, 碰撞概
率 = 0.0000%

乘数 = 41, 最小 Hash = -2147423916, 最大 Hash = 2147441721, 碰撞数量 = 1, 碰撞概
率 = 0.0010%

乘数 = 199, 最小 Hash = -2147459902, 最大 Hash = 2147480320, 碰撞数量 = 0, 碰撞概

率 = 0.0000%

Process finished with exit code 0



公众号: bugstack 虫洞栈, hash 碰撞图表

以上就是不同的乘数下的 hash 碰撞结果图标展示，从这里可以看出如下信息；

1. 乘数是 2 时，hash 的取值范围比较小，基本是堆积到一个范围内了，后面内容会看到这块的展示。
2. 乘数是 3、5、7、17 等，都有较大的碰撞概率
3. 乘数是 31 的时候，碰撞的概率已经很小了，基本稳定。
4. 顺着往下看，你会发现 199 的碰撞概率更小，这就相当于一排奇数的茅坑量多，自然会减少碰撞。但这个范围值已经远超过 int 的取值范围了，如果用此数作为乘数，又返回 int 值，就会丢失数据信息。

4. Hash 值散列分布

除了以上看到哈希值在不同乘数的一个碰撞概率后，关于散列表也就是 hash，还有一个非常重要的点，那就是要尽可能的让数据散列分布。只有这样才能减少 hash 碰撞次数，也就是后面章节要讲到的 hashMap 源码。

那么怎么看散列分布呢？如果我们能把 10 万个 hash 值铺到图表上，形成的一张图，就可以看出整个散列分布。但是这样的图会比较大，当我们缩小看后，就成为一个大黑点。所以这里我们采取分段统计，把 2^{32} 方分 64 个格子进行存放，每个格子都会有对应的数量的 hash 值，最终把这些数据展示在图表上。

4.1 哈希值分段存放

```
public static Map<Integer, Integer> hashArea(List<Integer> hashCodeList) {  
    Map<Integer, Integer> statistics = new LinkedHashMap<>();  
    int start = 0;  
    for (long i = 0x80000000; i <= 0x7fffffff; i += 67108864) {  
        long min = i;  
        long max = min + 67108864;  
        // 筛选出每个格子里的哈希值数量, java8 流统计: https://bugstack.cn/itstack-demo-any/2019/12/10/%E6%9C%89%E7%82%B9%E5%B9%B2%E8%B4%A7-Jdk1.8%E6%96%B0%E7%89%B9%E6%80%A7%E5%AE%9E%E6%88%98%E7%AF%87\(41%E4%B8%AA%E6%A1%88%E4%BE%8B\).html  
        int num = (int) hashCodeList.parallelStream().filter(x -> x >= min && x < max).count();  
        statistics.put(start++, num);  
    }  
    return statistics;
```

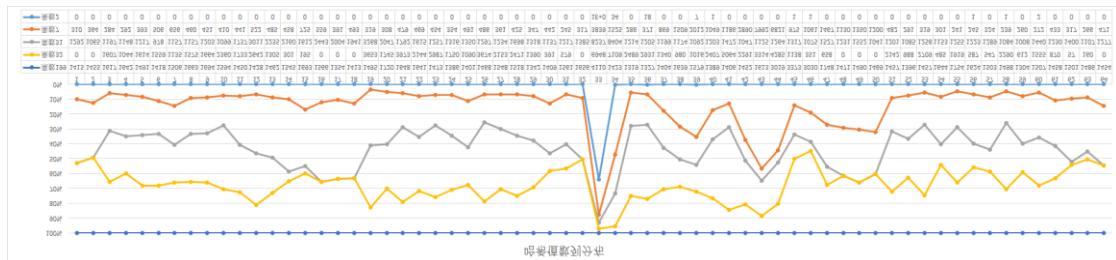
- 这个过程主要统计 `int` 取值范围内，每个哈希值存放到不同格子里的数量。
- 这里也是使用了 java8 的新特性语法，统计起来还是比较方便的。

4.2 单元测试

```
@Test  
public void test_hashArea() {  
    System.out.println(HashCode.hashArea(words, 2).values());  
    System.out.println(HashCode.hashArea(words, 7).values());  
    System.out.println(HashCode.hashArea(words, 31).values());  
    System.out.println(HashCode.hashArea(words, 32).values());  
    System.out.println(HashCode.hashArea(words, 199).values());  
}
```

- 这里列出我们要统计的乘数值，每一个乘数下都会有对应的哈希值数量汇总，也就是 64 个格子里的数量。
- 最终把这些统计值放入到 excel 中进行图表化展示。

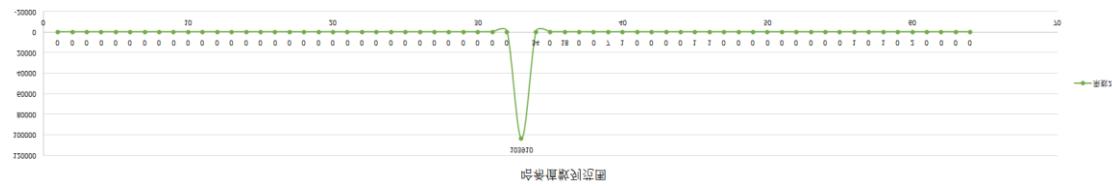
统计图表



公众号: bugstack 虫洞栈, hash 散列表

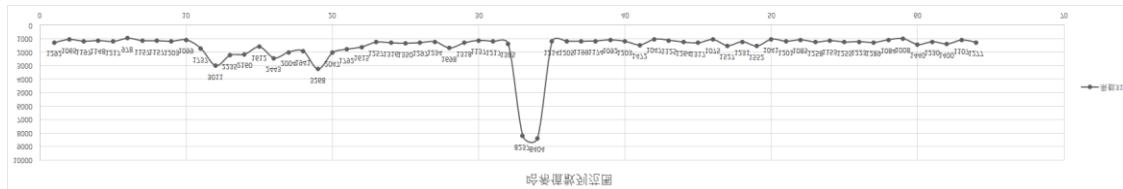
- 以上是一个堆积百分比统计图, 可以看到下方是不同乘数下的, 每个格子里的数据统计。
- 除了 199 不能用以外, 31 的散列结果相对来说比较均匀。

4.2.1 乘数 2 散列



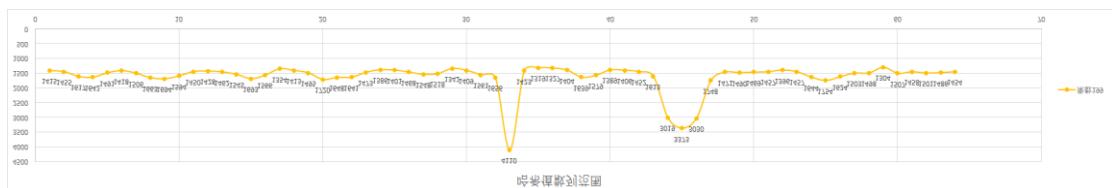
- 乘数是 2 的时候, 散列的结果基本都堆积在中间, 没有很好的散列。

4.2.2 乘数 31 散列



- 乘数是 31 的时候, 散列的效果就非常明显了, 基本在每个范围都有数据存放。

4.2.3 乘数 199 散列



- 乘数是 199 是不能用的散列结果, 但是它的数据是更加分散的, 从图上能看到有两个小山包。但因为数据区间问题会有数据丢失问题, 所以不能选择。

文中引用

- <http://www.tianxiaobo.com/2018/01/18/String-hashCode-%E6%96%B9%E6%B3%95%E4%B8%BA%E4%BB%80%E4%B9%88%E9%80%89%E6%8B%A9%E6%95%B0%E5%AD%9731%E4%BD%9C%E4%B8%BA%E4%B9%98%E5%AD%90/>
- <https://stackoverflow.com/questions/299304/why-does-javas-hashcode-in-string-use-31-as-a-multiplier>

四、总结

- 以上主要介绍了 hashCode 选择 31 作为乘数的主要原因和实验数据验证，算是一个散列的数据结构的案例讲解，在后续的类似技术中，就可以解释其他的源码设计思路了。
- 看过本文至少应该让你从根本上解释了 hashCode 的设计，关于他的所有问题也就不需要死记硬背了，学习编程内容除了最开始的模仿到深入以后就需要不断的研究数学逻辑和数据结构。
- 文中参考了优秀的 hashCode 资料和 stackoverflow，并亲自做实验验证结果，大家也可以下载本文中资源内容；英文字典、源码、excel 图表等内容。

第 2 节：HashMap 源码分析(上)

得益于 [Doug Lea](#) 老爷子的操刀，让 [HashMap](#) 成为使用和面试最频繁的 API，没办法设计的太优秀了！

HashMap 最早出现在 JDK 1.2 中，底层基于散列算法实现。HashMap 允许 null 键和 null 值，在计算哈键的哈希值时，null 键哈希值为 0。HashMap 并不保证键值对的顺序，这意味着在进行某些操作后，键值对的顺序可能会发生变化。另外，需要注意的是，HashMap 是非线程安全类，在多线程环境下可能会存在问题。

HashMap 最早在 JDK 1.2 中就出现了，底层是基于散列算法实现，随着几代的优化更新到目前为止它的源码部分已经比较复杂，涉及的知识点也非常多，在 JDK 1.8 中包括：[1、散列表实现](#)、[2、扰动函数](#)、[3、初始化容量](#)、[4、负载因子](#)、[5、扩容元素拆分](#)、[6、链表树化](#)、[7、红黑树](#)、[8、插入](#)、[9、查找](#)、[10、删除](#)、[11、遍历](#)、[12、分段锁](#)等等，因涉及的知识点较多所以需要分开讲解，本章节我们会先把目光放在前五项上，也就是关于数据结构的使用上。

数据结构相关往往与数学离不开，学习过程中建议下载相应源码进行实验验证，可能这个过程有点烧脑，但学会后不用死记硬背就可以理解这部分知识。

一、资源下载

本章节涉及的源码和资源在工程，[interview-04](#) 中，包括；

1. 10 万单词测试数据，在 doc 文件夹
2. 扰动函数 excel 展现，在 dock 文件夹
3. 测试源码部分在 [interview-04](#) 工程中

可以通过关注公众号：[bugstack 虫洞栈](#)，回复下载进行获取{[回复下载后打开获得的链接，找到编号 ID: 19](#)}

二、源码分析

1. 写一个最简单的 HashMap

学习 HashMap 前，最好的方式是先了解这是一种怎么样的数据结构来存放数据。而 HashMap 经过多个版本的迭代后，乍一看代码还是很复杂的。就像你原来只穿个裤衩，现在还有秋裤和风衣。所以我们先来看看最根本的 HashMap 是什么样，

也就是只穿裤衩是什么效果，之后再去分析它的源码。

问题：假设我们有一组 7 个字符串，需要存放到数组中，但要求在获取每个元素的时候时间复杂度是 $O(1)$ 。也就是说你不能通过循环遍历的方式进行获取，而是要定位到数组 ID 直接获取相应的元素。

方案：如果说我们需要通过 ID 从数组中获取元素，那么就需要把每个字符串都计算出一个在数组中的位置 ID。字符串获取 ID 你能想到什么方式？一个字符串最直接的获取跟数字相关的信息就是 `HashCode`，可 `HashCode` 的取值范围太大了 `[-2147483648, 2147483647]`，不可能直接使用。那么就需要使用 `HashCode` 与数组长度做与运算，得到一个可以在数组中出现的位置。如果说有两个元素得到同样的 ID，那么这个数组 ID 下就存放两个字符串。

以上呢其实这就是我们要把字符串散列到数组中的一个基本思路，接下来我们就把这个思路用代码实现出来。

1.1 代码实现

```
// 初始化一组字符串
List<String> list = new ArrayList<>();
list.add("j1kk");
list.add("lopi");
list.add("小傅哥");
list.add("e4we");
list.add("alpo");
list.add("yhjk");
list.add("plop");

// 定义要存放的数组
String[] tab = new String[8];

// 循环存放
for (String key : list) {
    int idx = key.hashCode() & (tab.length - 1); // 计算索引位置
    System.out.println(String.format("key 值=%s Idx=%d", key, idx));
    if (null == tab[idx]) {
        tab[idx] = key;
        continue;
}
```

```

    tab[idx] = tab[idx] + "->" + key;
}

// 输出测试结果
System.out.println(JSON.toJSONString(tab));

```

这段代码整体看起来也是非常简单，并没有什么复杂度，主要包括以下内容；

1. 初始化一组字符串集合，这里初始化了 7 个。
2. 定义一个数组用于存放字符串，注意这里的长度是 8，也就是 2 的倍数。这样的数组长度才会出现一个 **0111** 除高位以外都是 1 的特征，也是为了散列。
3. 接下来就是循环存放数据，计算出每个字符串在数组中的位置。
`key.hashCode() & (tab.length - 1)`
4. 在字符串存放到数组的过程，如果遇到相同的元素，进行连接操作[模拟链表的过程](#)。
5. 最后输出存放结果。

测试结果

```

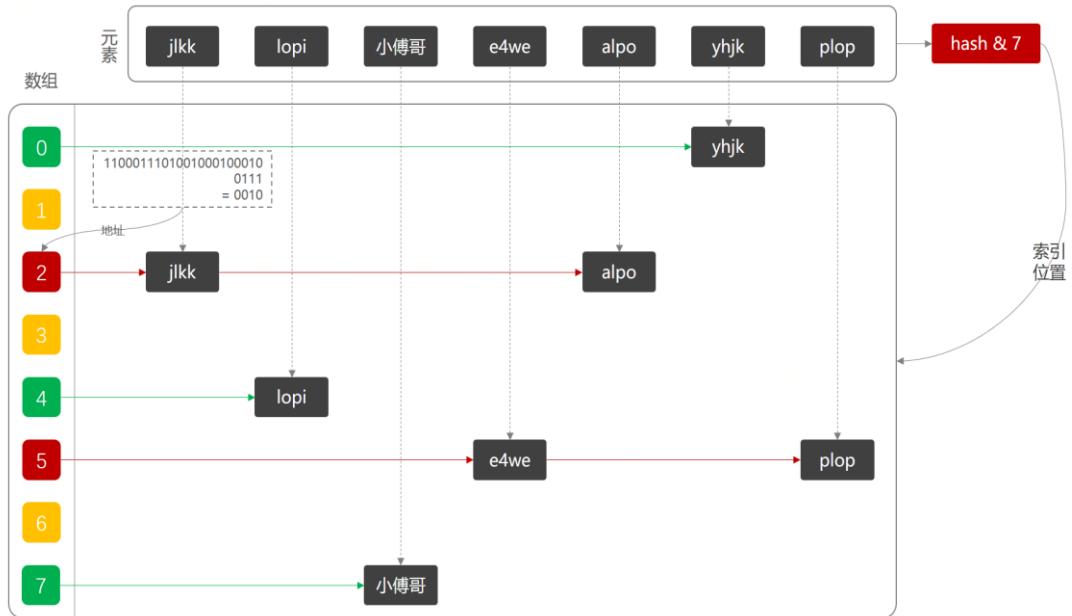
key 值=jlkk Idx=2
key 值=lopi Idx=4
key 值=小傅哥 Idx=7
key 值=e4we Idx=5
key 值=alpo Idx=2
key 值=yhjk Idx=0
key 值=plop Idx=5
测试结果: ["yhjk",null,"jlkk->alpo",null,"lopi","e4we->plop",null,"小傅哥"]

```

- 在测试结果首先是计算出每个元素在数组的 Idx，也有出现重复的位置。
- 最后是测试结果的输出，1、3、6，位置是空的，2、5，位置有两个元素被链接起来 **e4we->plop**。
- 这就达到了我们一个最基本的要求，将串元素散列存放到数组中，最后通过字符串元素的索引 ID 进行获取对应字符串。这样是 HashMap 的一个最基本原理，有了这个基础后面就会更容易理解 HashMap 的源码实现。

1.2 Hash 散列示意图

如果上面的测试结果不能在你的头脑中很好的建立出一个数据结构，那么可以看以下这张散列示意图，方便理解；



bugstack.cn Hash 散列示意图

- 这张图就是上面代码实现的全过程，将每一个字符串元素通过 Hash 计算索引位置，存放到数组中。
- 黄色的索引 ID 是没有元素存放、绿色的索引 ID 存放了一个元素、红色的索引 ID 存放了两个元素。

1.3 这个简单的 HashMap 有哪些问题

以上我们实现了一个简单的 HashMap，或者说还算不上 HashMap，只能算做一个散列数据存放的雏形。但这样的一个数据结构放在实际使用中，会有哪些问题呢？

1. 这里所有的元素存放都需要获取一个索引位置，而如果元素的位置不够散列碰撞严重，那么就失去了散列表存放的意义，没有达到预期的性能。
2. 在获取索引 ID 的计算公式中，需要数组长度是 2 的倍数，那么怎么进行初始化这个数组大小。
3. 数组越小碰撞的越大，数组越大碰撞的越小，时间与空间如何取舍。
4. 目前存放 7 个元素，已经有两个位置都存放了 2 个字符串，那么链表越来越长怎么优化。
5. 随着元素的不断添加，数组长度不足扩容时，怎么把原有的元素，拆分到新的位置上去。

以上这些问题可以归纳为：[扰动函数](#)、[初始化容量](#)、[负载因子](#)、[扩容方法](#)以及[链表](#)和[红黑树](#)转换的使用等。接下来我们会逐个问题进行分析。

2. 扰动函数

在 HashMap 存放元素时候有这样一段代码来处理哈希值，这是 `java 8` 的散列值扰动函数，用于优化散列效果：

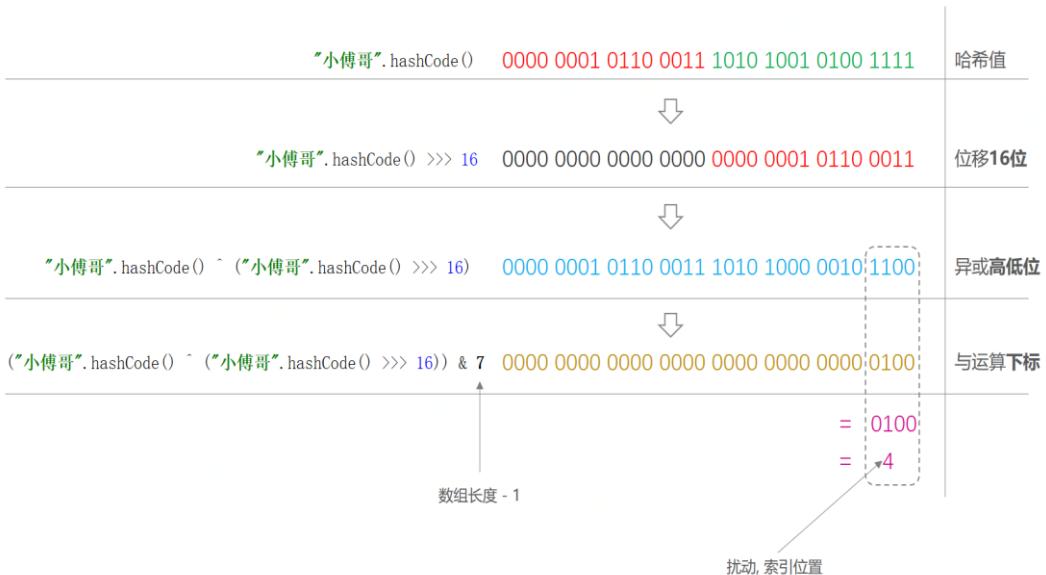
```
static final int hash(Object key) {  
    int h;  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}
```

2.1 为什么使用扰动函数

理论上来说字符串的 `hashCode` 是一个 `int` 类型值，那可以直接作为数组下标了，且不会出现碰撞。但是这个 `hashCode` 的取值范围是 $[-2147483648, 2147483647]$ ，有将近 40 亿的长度，谁也不能把数组初始化的这么大，内存也是放不下的。

我们默认初始化的 Map 大小是 16 个长度 `DEFAULT_INITIAL_CAPACITY = 1 << 4`，所以获取的 Hash 值并不能直接作为下标使用，需要与数组长度进行取模运算得到一个下标值，也就是我们上面做的散列例子。

那么，`HashMap` 源码这里不只是直接获取哈希值，还进行了一次扰动计算，`(h = key.hashCode()) ^ (h >>> 16)`。把哈希值右移 16 位，也就正好是自己长度的一半，之后与原哈希值做异或运算，这样就混合了原哈希值中的高位和低位，增大了随机性。计算方式如下图：



- 说白了，使用扰动函数就是为了增加随机性，让数据元素更加均衡的散列，减少碰撞。

2.2 实验验证扰动函数

从上面的分析可以看出，扰动函数使用了哈希值的高半区和低半区做异或，混合原始哈希码的高位和低位，以此来加大低位区的随机性。

但看不到实验数据的话，这终究是一段理论，具体这段哈希值真的被增加了随机性没有，并不知道。所以这里我们要做一个实验，这个实验是这样做；

- 选取 10 万个单词词库
- 定义 128 位长度的数组格子
- 分别计算在扰动和不扰动下，10 万单词的下标分配到 128 个格子的数量
- 统计各个格子数量，生成波动曲线。如果扰动函数下的波动曲线相对更平稳，那么证明扰动函数有效果。

2.2.1 扰动代码测试

扰动函数对比方法

```
public class Disturb {

    public static int disturbHashIdx(String key, int size) {
        return (size - 1) & (key.hashCode() ^ (key.hashCode() >>> 16));
    }

    public static int hashIdx(String key, int size) {
        return (size - 1) & key.hashCode();
    }
}
```

- `disturbHashIdx` 扰动函数下，下标值计算
- `hashIdx` 非扰动函数下，下标值计算

单元测试

```
// 10 万单词已经初始化到words 中
@Test
public void test_disturb() {
    Map<Integer, Integer> map = new HashMap<>(16);
```

```

for (String word : words) {
    // 使用扰动函数
    int idx = Disturb.disturbHashIdx(word, 128);
    // 不使用扰动函数
    // int idx = Disturb.hashIdx(word, 128);
    if (map.containsKey(idx)) {
        Integer integer = map.get(idx);
        map.put(idx, ++integer);
    } else {
        map.put(idx, 1);
    }
}
System.out.println(map.values());
}

```

以上分别统计两种函数下的下标值分配，最终将统计结果放到 excel 中生成图表。

2.2.2 扰动函数散列图表

以上的两张图，分别是没有使用扰动函数和使用扰动函数的，下标分配。实验数据：

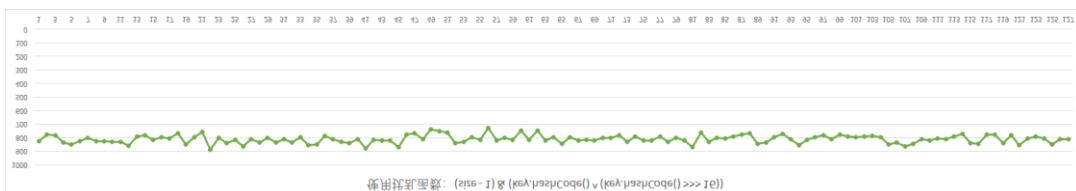
1. 10 万个不重复的单词
2. 128 个格子，相当于 128 长度的数组

未使用扰动函数



bugstack.cn 未使用扰动函数

使用扰动函数



bugstack.cn 使用扰动函数

- 从这两种的对比图可以看出来，在使用了扰动函数后，数据分配的更加均匀了。
- 数据分配均匀，也就是散列的效果更好，减少了 hash 的碰撞，让数据存放和获取的效率更佳。

3. 初始化容量和负载因子

接下来我们讨论下一个问题，从我们模仿 HashMap 的例子中以及 HashMap 默认的初始化大小里，都可以知道，散列数组需要一个 2 的倍数的长度，因为只有 2 的倍数在减 1 的时候，才会出现 `01111` 这样的值。

那么这里就有一个问题，我们在初始化 HashMap 的时候，如果传一个 17 个的值 `new HashMap<>(17);`，它会怎么处理呢？

3.1 寻找 2 的倍数最小值

在 HashMap 的初始化中，有这样一段方法：

```
public HashMap(int initialCapacity, float loadFactor) {
    ...
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}
```

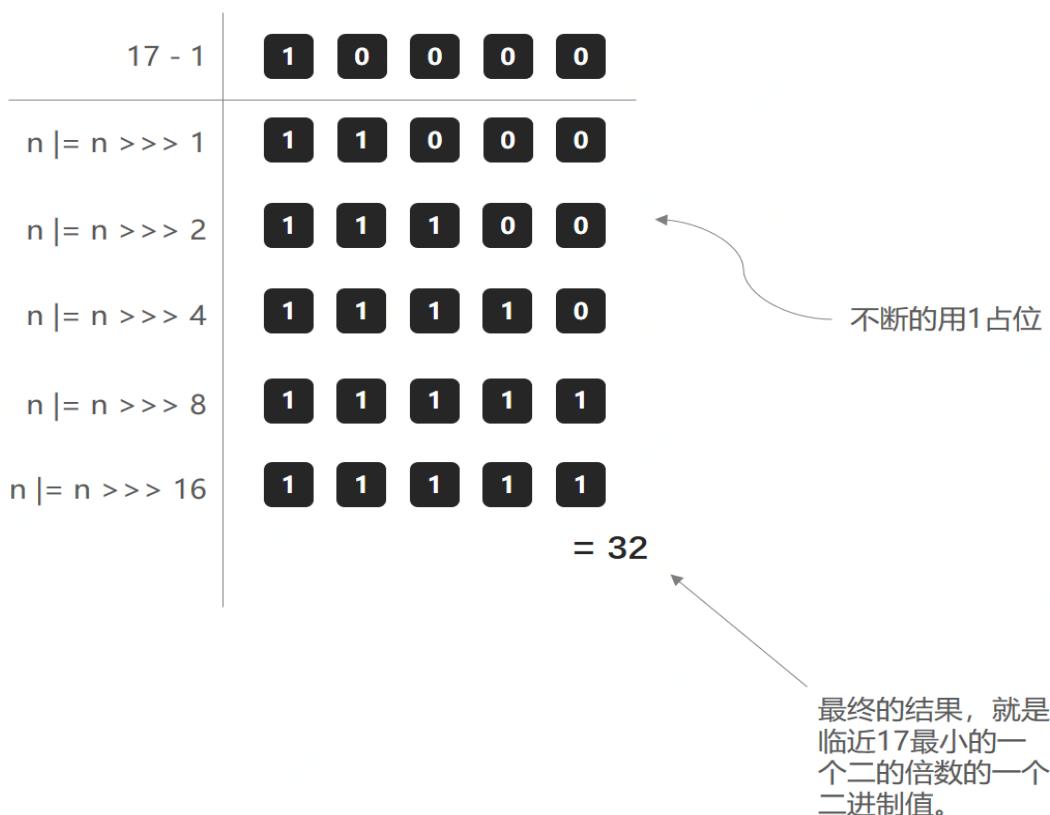
- 阀值 `threshold`，通过方法 `tableSizeFor` 进行计算，是根据初始化来计算的。
- 这个方法也就是要寻找比初始值大的，最小的那个 2 进制数值。比如传了 17，我应该找到的是 32。

计算阀值大小的方法：

```
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

- `MAXIMUM_CAPACITY = 1 << 30`, 这个是临界范围, 也就是最大的 Map 集合。
- 乍一看可能有点晕⑨怎么都在向右移位 1、2、4、8、16, 这主要是为了把二进制的各个位置都填上 1, 当二进制的各个位置都是 1 以后, 就是一个标准的 2 的倍数减 1 了, 最后把结果加 1 再返回即可。

那这里我们把 17 这样一个初始化计算阀值的过程, 用图展示出来, 方便理解;



bugstack.cn 计算阀值

3.2 负载因子

```
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

负载因子是做什么的?

负载因子, 可以理解成一辆车可承重重量超过某个阀值时, 把货放到新的车上。

那么在 HashMap 中, 负载因子决定了数据量多少了以后进行扩容。这里要提到上面做的 `HashMap` 例子, 我们准备了 7 个元素, 但是最后还有 3 个位置空余, 2 个位置存放了 2 个元素。所以可能即使你数据比数组容量大时也是不一定能正好好的把数组占满的, 而是在某些小标位置出现了大量的碰撞, 只能在同一个位

置用链表存放，那么这样就失去了 Map 数组的性能。

所以，要选择一个合理的大小下进行扩容，默认值 0.75 就是说当阀值容量占了 3/4 时赶紧扩容，减少 Hash 碰撞。

同时 0.75 是一个默认构造值，在创建 HashMap 也可以调整，比如你希望用更多的空间换取时间，可以把负载因子调的更小一些，减少碰撞。

4. 扩容元素拆分

为什么扩容，因为数组长度不足了。那扩容最直接的问题，就是需要把元素拆分到新的数组中。拆分元素的过程中，原 jdk1.7 中会需要重新计算哈希值，但是到 jdk1.8 中已经进行优化，不在需要重新计算，提升了拆分的性能，设计的还是非常巧妙的。

4.1 测试数据

```
@Test
public void test_hashMap() {
    List<String> list = new ArrayList<>();
    list.add("jIkk");
    list.add("lopi");
    list.add("jmdw");
    list.add("e4we");
    list.add("io98");
    list.add("nmhg");
    list.add("vfg6");
    list.add("gfrt");
    list.add("alpo");
    list.add("vfbh");
    list.add("bnhj");
    list.add("zuio");
    list.add("iu8e");
    list.add("yhjk");
    list.add("plop");
    list.add("dd0p");

    for (String key : list) {
        int hash = key.hashCode() ^ (key.hashCode() >>> 16);
        System.out.println("字符串: " + key + "\tIdx(16): "
            + hash);
    }
}
```

```

    " + ((16 - 1) & hash) + " \tBit 值:
    " + Integer.toBinaryString(hash) + " - " + Integer.toBinaryString(hash & 16) + " \t
\tIdx(32): " + (
    System.out.println(Integer.toBinaryString(key.hashCode()) +" "+ Integer.toBi
naryString(hash) + " " + Integer.toBinaryString((32 - 1) & hash));
}
}

```

测试结果

字符串: jlk **Idx(16):** 3 Bit 值: 1100011101001000010011 - 10000 **Idx(32):** 19
1100011101001000100010 1100011101001000010011 10011

字符串: lopi **Idx(16):** 14 Bit 值: 1100101100011010001110 - 0 **Idx(32):** 14
1100101100011010111100 1100101100011010001110 1110

字符串: jmdw **Idx(16):** 7 Bit 值: 1100011101010100100111 - 0 **Idx(32):** 7
1100011101010100010110 1100011101010100100111 111

字符串: e4we **Idx(16):** 3 Bit 值: 1011101011101101010011 - 10000 **Idx(32):** 19
1011101011101101111101 1011101011101101010011 10011

字符串: io98 **Idx(16):** 4 Bit 值: 1100010110001011110100 - 10000 **Idx(32):** 20
1100010110001011000101 1100010110001011110100 10100

字符串: nmhg **Idx(16):** 13 Bit 值: 1100111010011011001101 - 0 **Idx(32):** 13
110011101001101111110 1100111010011011001101 1101

字符串: vfg6 **Idx(16):** 8 Bit 值: 1101110010111101101000 - 0 **Idx(32):** 8
1101110010111101011111 1101110010111101101000 1000

字符串: gfrt **Idx(16):** 1 Bit 值: 1100000101111101010001 - 10000 **Idx(32):** 17
1100000101111101100001 1100000101111101010001 10001

字符串: alpo **Idx(16):** 7 Bit 值: 1011011011101101000111 - 0 **Idx(32):** 7
1011011011101101101010 1011011011101101000111 111

字符串: vfbh **Idx(16):** 1 Bit 值: 1101110010111101100001 - 0 **Idx(32):** 1
11011100101111011110110 1101110010111011000001 1

字符串: bnhj **Idx(16):** 0 Bit 值: 1011100011011001100000 - 0 **Idx(32):** 0
1011100011011001001110 1011100011011001100000 0

字符串: zuio **Idx(16):** 8 Bit 值: 1110010011100110011000 - 10000 **Idx(32):** 24
1110010011100110100001 1110010011100110011000 11000

字符串: iu8e **Idx(16):** 8 Bit 值: 1100010111100101101000 - 0 **Idx(32):** 8
1100010111100101011001 1100010111100101101000 1000

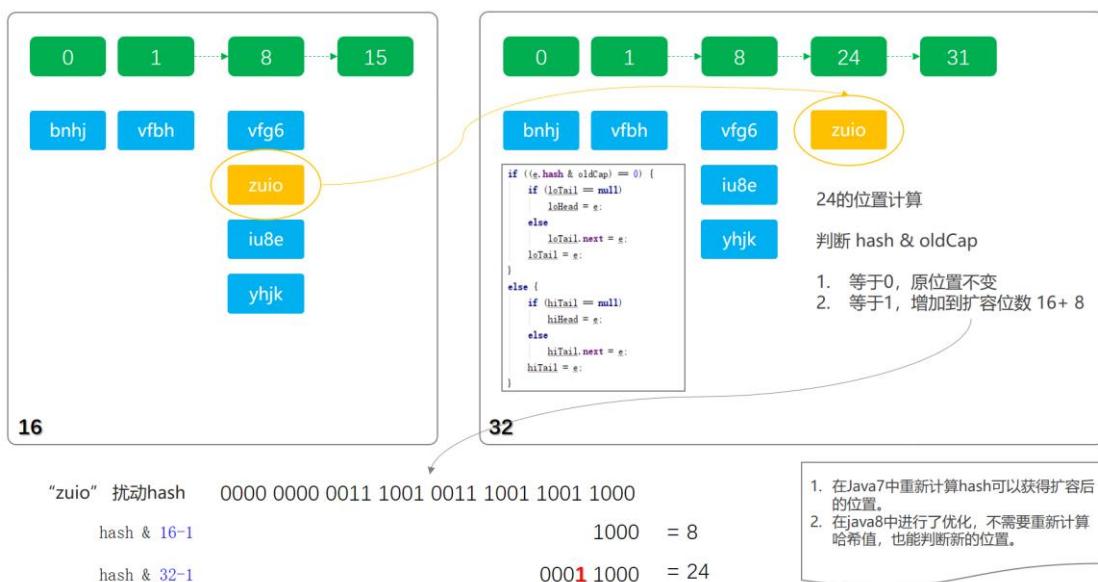
字符串: yhjk **Idx(16):** 8 Bit 值: 1110001001010010101000 - 0 **Idx(32):** 8
111000100101001000000 1110001001010010101000 1000

字符串: plop **Idx(16):** 9 Bit 值: 1101001000110011101001 - 0 **Idx(32):** 9
1101001000110011011101 1101001000110011101001 1001

字符串: dd0p **Idx(16)**: 14 Bit 值: 1011101111001011101110 - 0 **Idx(32)**: 14
1011101111001011100000 1011101111001011101110 1110

- 这里我们随机使用一些字符串计算他们分别在 16 位长度和 32 位长度数组下的索引分配情况，看哪些数据被重新路由到了新的地址。
- 同时，这里还可以观察出一个非常重要的信息，原哈希值与扩容新增出来的长度 16，进行&运算，如果值等于 0，则下标位置不变。如果不为 0，那么新的位置则是原来位置上加 16。{这个地方需要好好理解下，并看实验数据}
- 这样一来，就不需要在重新计算每一个数组中元素的哈希值了。

4.2 数据迁移



bugstack.cn 数据迁移

- 这张图就是原 16 位长度数组元素，向度中转移的过程。
- 其中黄色区域元素 `zuio` 因计算结果 `hash & oldCap` 不为 1，则被迁移到下标位置 24。
- 同时还是用重新计算哈希值的方式验证了，确实分配到 24 的位置，因为这是在二进制计算中补 1 的过程，所以可以通过上面简化的方式确定哈希值的位置。

三、总结

- 如果你能坚持看完这部分内容，并按照文中的例子进行相应的实验验证，那么一定可以学会本章节涉及这五项知识点：[1、散列表实现](#)、[2、扰动函数](#)、[3、初始化容量](#)、[4、负载因子](#)、[5、扩容元素拆分](#)。
- 对我个人来说以前也知道这部分知识，但是没有验证过，只知道概念如此，正好借着写面试手册专栏，加深学习，用数据验证理论，让知识点可以更加深入的理解。
- 这一章节完事，下一章节继续进行 HashMap 的其他知识点挖掘，让懂了就是真的懂了。好了，写到这里了，感谢大家的阅读。如果某处没有描述清楚，或者有不理解的点，欢迎与我讨论交流。

第 3 节：HashMap 源码分析(下)

在上一章节我们讲解并用数据验证了，HashMap 中的，散列表的实现、扰动函数、负载因子以及扩容拆分等核心知识点以及相应的作用。

除了以上这些知识点外，HashMap 还有基本的数据功能；存储、删除、获取、遍历，在这些功能中经常会听到链表、红黑树、之间转换等功能。而红黑树是在 jdk1.8 引入到 HashMap 中解决链表过长问题的，简单说当链表长度 $>=8$ 时，将链表转换为红黑树（当然这里还有一个扩容的知识点，不一定都会树化 $[MIN_TREEIFY_CAPACITY]$ ）。

那么本章节会进行讲解以下知识点：

1. 数据插入流程和源码分析
2. 链表树化以及树转链表
3. 遍历过程中的无序 Set 的核心知识

一、插入

1. 疑问点&考题

通过上一章节的学习，大家对于一个散列表数据结构的 HashMap 往里面插入数据时，基本已经有了一个印象。简单来说就是通过你的 Key 值取得哈希再计算下标，之后把相应的数据存放到里面。

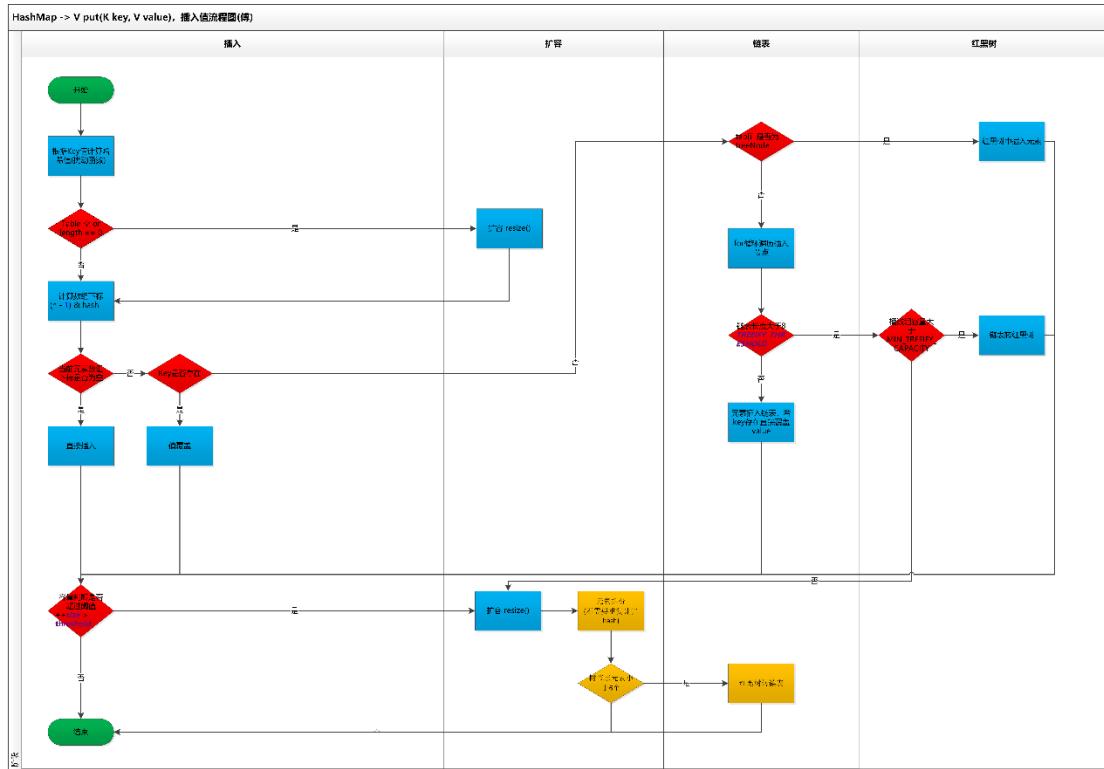
但再这个过程中会遇到一些问题，比如；

1. 如果出现哈希值计算的下标碰撞了怎么办？
2. 如果碰撞了是扩容数组还是把值存成链表结构，让一个节点有多个值存放呢？
3. 如果存放的数据的链表过长，就失去了散列表的性能了，怎么办呢？
4. 如果想解决链表过长，什么时候使用树结构呢，使用哪种树呢？

这些疑问点都会在后面的内容中逐步讲解，也可以自己思考一下，如果是你来设计，你会怎么做。

2. 插入流程和源码分析

HashMap 插入数据流程图



公众号: bugstack 虫洞栈, HashMap 插入数据流程图

visio 原版流程图, 可以通过关注公众号: bugstack 虫洞栈, 进行下载

以上就是 HashMap 中一个数据插入的整体流程, 包括了: 计算下标、何时扩容、何时链表转红黑树等, 具体如下:

- 首先进行哈希值的扰动, 获取一个新的哈希值。`(key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);`
- 判断 tab 是否位空或者长度为 0, 如果是则进行扩容操作。
`if ((tab = table) == null || (n = tab.length) == 0)`
`n = (tab = resize()).length;`
- 根据哈希值计算下标, 如果对应小标正好没有存放数据, 则直接插入即可
否则需要覆盖。`tab[i = (n - 1) & hash]`
- 判断 `tab[i]` 是否为树节点, 否则向链表中插入数据, 是则向树中插入节点。
- 如果链表中插入节点的时候, 链表长度大于等于 8, 则需要把链表转换为红黑树。`treeifyBin(tab, hash);`
- 最后所有元素处理完成后, 判断是否超过阈值; `threshold`, 超过则扩容。
- `treeifyBin`, 是一个链表转树的方法, 但不是所有的链表长度为 8 后都会转成树, 还需要判断存放 key 值的数组桶长度是否小于 `MIN_TREEIFY_CAPACITY`。如果小于则需要扩容, 扩容后链表上的数据会被拆分散列到相应的桶节点上, 也就把链表长度缩短了。

JDK1.8 HashMap 的 put 方法源码如下:

```

public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // 初始化桶数组 table, table 被延迟到插入新数据时再进行初始化
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 如果桶中不包含键值对节点引用, 则将新键值对节点的引用存入桶中即可
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        // 如果键的值以及节点 hash 等于链表中的第一个键值对节点时, 则将 e 指向该键值对
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        // 如果桶中的引用类型为 TreeNode, 则调用红黑树的插入方法
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            // 对链表进行遍历, 并统计链表长度
            for (int binCount = 0; ; ++binCount) {
                // 链表中不包含要插入的键值对节点时, 则将该节点接在链表的最后
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    // 如果链表长度大于或等于树化阈值, 则进行树化操作
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
            }
            // 条件为 true, 表示当前链表包含要插入的键值对, 终止遍历
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                break;
        }
    }
}

```

```

        p = e;
    }
}

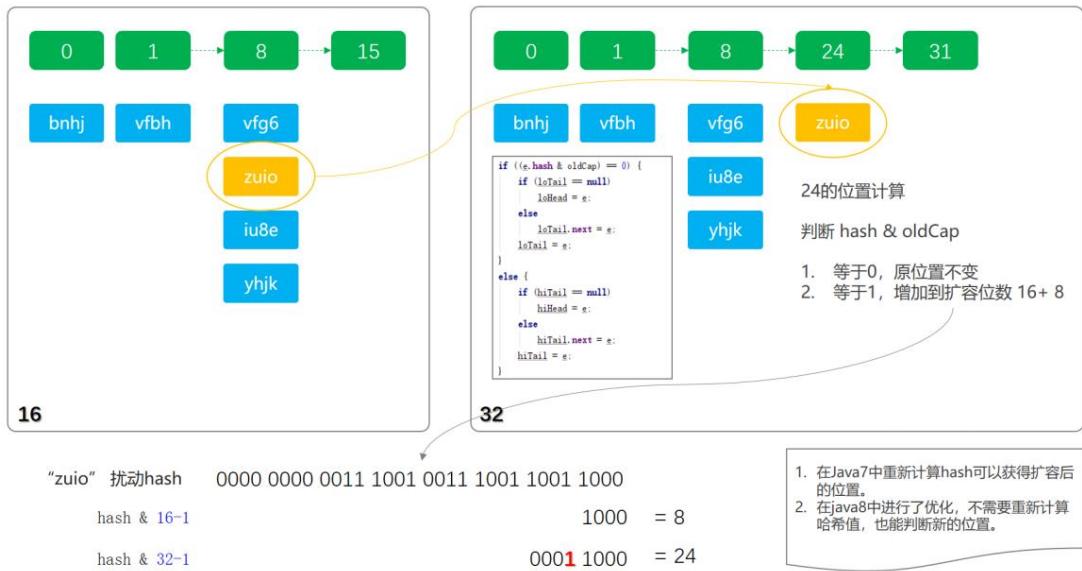
// 判断要插入的键值对是否存在 HashMap 中
if (e != null) { // existing mapping for key
    V oldValue = e.value;
    // onlyIfAbsent 表示是否仅在 oldValue 为 null 的情况下更新键值对的值
    if (!onlyIfAbsent || oldValue == null)
        e.value = value;
    afterNodeAccess(e);
    return oldValue;
}
++modCount;
// 键值对数量超过阈值时，则进行扩容
if (++size > threshold)
    resize();
afterNodeInsertion(evict);
return null;
}

```

3. 扩容机制

HashMap 是基于数组+链表和红黑树实现的，但用于存放 key 值得数组桶的长度是固定的，由初始化决定。

那么，随着数据的插入数量增加以及负载因子的作用下，就需要扩容来存放更多的数据。而扩容中有一个非常重要的点，就是 jdk1.8 中的优化操作，可以不需要再重新计算每一个元素的哈希值，这在上一章节中已经讲到，可以阅读系列专题文章，机制如下图：



这里我们主要看下扩容的代码(注释部分);

```

final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    // Cap 是 capacity 的缩写, 容量。如果容量不为空, 则说明已经初始化。
    if (oldCap > 0) {
        // 如果容量达到最大  $1 \ll 30$  则不再扩容
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        // 按旧容量和阀值的2倍计算新容量和阀值
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                  oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        // initial capacity was placed in threshold 翻译过来的意思, 如下:
        // 初始化时, 将 threshold 的值赋值给 newCap,
        // HashMap 使用 threshold 变量暂时保存 initialCapacity 参数的值

```

```

    newCap = oldThr;

else { // zero initial threshold signifies using defaults
    // 这一部分也是，源代码中也有相应的英文注释
    // 调用无参构造方法时，数组桶数组容量为默认容量  $1 \ll 4$ ; aka 16
    // 阈值：是默认容量与负载因子的乘积，0.75
    newCap = DEFAULT_INITIAL_CAPACITY;
    newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
}

// newThr 为 0，则使用阈值公式计算容量
if (newThr == 0) {
    float ft = (float)newCap * loadFactor;
    newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
        (int)ft : Integer.MAX_VALUE);
}
threshold = newThr;

@SuppressWarnings({"rawtypes", "unchecked"})
// 初始化数组桶，用于存放 key
Node<K,V>[] newTab = (Node<K,V>[][])new Node[newCap];
table = newTab;
if (oldTab != null) {
    // 如果旧数组桶，oldCap 有值，则遍历将键值映射到新数组桶中
    for (int j = 0; j < oldCap; ++j) {
        Node<K,V> e;
        if ((e = oldTab[j]) != null) {
            oldTab[j] = null;
            if (e.next == null)
                newTab[e.hash & (newCap - 1)] = e;
            else if (e instanceof TreeNode)
                // 这里 split，是红黑树拆分操作。在重新映射时操作的。
                ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
            else { // preserve order
                Node<K,V> loHead = null, loTail = null;
                Node<K,V> hiHead = null, hiTail = null;
                Node<K,V> next;
                // 这里是链表，如果当前是按照链表存放的，则将链表节点按原顺序进行分组
                {这里有专门的文章介绍，如何不需要重新计算哈希值进行拆分《HashMap 核心知识，扰动函数、负载因子、扩容链表拆分，深度学习》}
            }
        }
    }
}

```

```

do {
    next = e.next;
    if ((e.hash & oldCap) == 0) {
        if (loTail == null)
            loHead = e;
        else
            loTail.next = e;
        loTail = e;
    }
    else {
        if (hiTail == null)
            hiHead = e;
        else
            hiTail.next = e;
        hiTail = e;
    }
} while ((e = next) != null);

// 将分组后的链表映射到桶中
if (loTail != null) {
    loTail.next = null;
    newTab[j] = loHead;
}
if (hiTail != null) {
    hiTail.next = null;
    newTab[j + oldCap] = hiHead;
}
}

}

}

}

return newTab;
}

```

以上的代码稍微有些长，但是整体的逻辑还是蛮清晰的，主要包括：

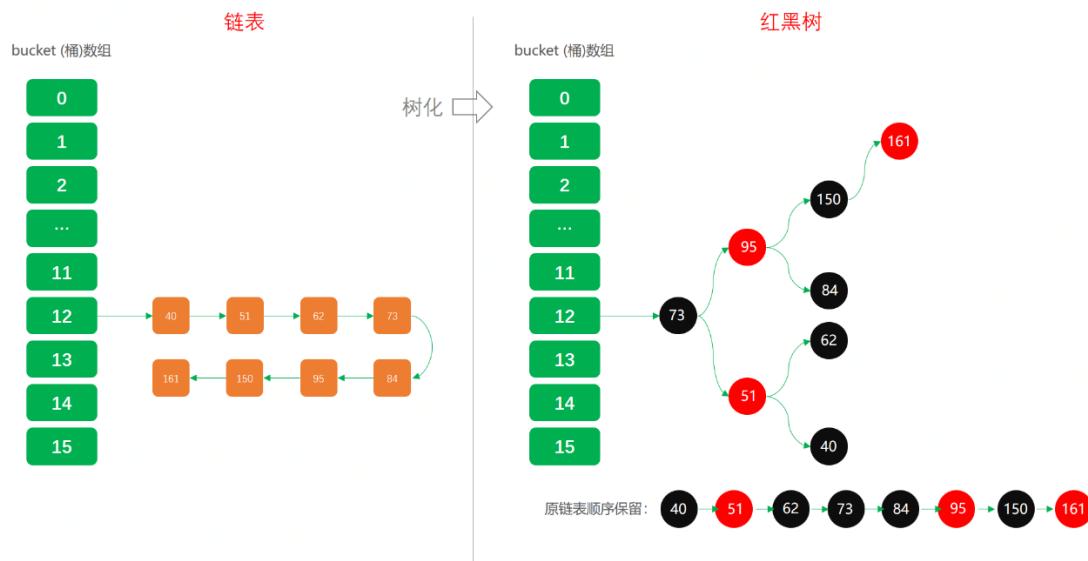
1. 扩容时计算出新的 newCap、newThr，这是两个单词的缩写，一个是 Capacity，另一个是阀 Threshold
2. newCap 用于创新的数组桶 `new Node[newCap];`

3. 随着扩容后，原来那些因为哈希碰撞，存放成链表和红黑树的元素，都需要进行拆分存放到新的位置中。

4. 链表树化

HashMap 这种散列表的数据结构，最大的性能在于可以 $O(1)$ 时间复杂度定位到元素，但因为哈希碰撞不得已在一个下标里存放多组数据，那么 jdk1.8 之前的设计只是采用链表的方式进行存放，如果需要从链表中定位到数据时间复杂度就是 $O(n)$ ，链表越长性能越差。因为在 jdk1.8 中把过长的链表也就是 8 个，优化为自平衡的红黑树结构，以此让定位元素的时间复杂度优化近似于 $O(\log n)$ ，这样来提升元素查找的效率。但也不是完全抛弃链表，因为在元素相对不多的情况下，链表的插入速度更快，所以综合考虑下设定阈值为 8 才进行红黑树转换操作。

链表转红黑树，如下图；



微信公众号：bugstack 虫洞栈，链表转红黑树

以上就是一组链表转换为红黑树的情况，元素包括：40、51、62、73、84、95、150、161 这些是经过实际验证可分配到 $Idx: 12$ 的节点

通过这张图，基本可以有一个链表换行到红黑树的印象，接下来阅读下对应的源码。

链表树化源码

```
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    // 这块就是我们上面提到的，不一定树化还可能只是扩容。主要桶数组容量是否小于
    64 MIN_TREEIFY_CAPACITY
```

```

if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
    resize();
else if ((e = tab[index = (n - 1) & hash]) != null) {
    // 又是单词缩写; hd = head (头部), tl = tile (结尾)
    TreeNode<K,V> hd = null, tl = null;
    do {
        // 将普通节点转换为树节点, 但此时还不是红黑树, 也就是说还不一定平衡
        TreeNode<K,V> p = replacementTreeNode(e, null);
        if (tl == null)
            hd = p;
        else {
            p.prev = tl;
            tl.next = p;
        }
        tl = p;
    } while ((e = e.next) != null);
    if ((tab[index] = hd) != null)
        // 转红黑树操作, 这里需要循环比较, 染色、旋转。关于红黑树, 在下一章节详细讲解
        hd.treeify(tab);
}
}

```

这一部分链表树化的操作并不复杂，复杂点在于下一层的红黑树转换上，这部分知识点会在后续章节中专门介绍；

以上源码主要包括的知识点如下；

1. 链表树化的条件有两点；链表长度大于等于 8、桶容量大于 64，否则只是扩容，不会树化。
2. 链表树化的过程中是先由链表转换为树节点，此时的树可能不是一颗平衡树。同时在树转换过程中会记录链表的顺序，`tl.next = p`，这主要方便后续树转链表和拆分更方便。
3. 链表转换成树完成后，在进行红黑树的转换。先简单介绍下，红黑树的转换需要染色和旋转，以及比对大小。在比较元素的大小中，有一个比较有意思的方法，`tieBreakOrder` 加时赛，这主要是因为 HashMap 没有像 TreeMap 那样本身就有 Comparator 的实现。

5. 红黑树转链

在链表转红黑树中我们重点介绍了一句，在转换树的过程中，记录了原有链表的顺序。

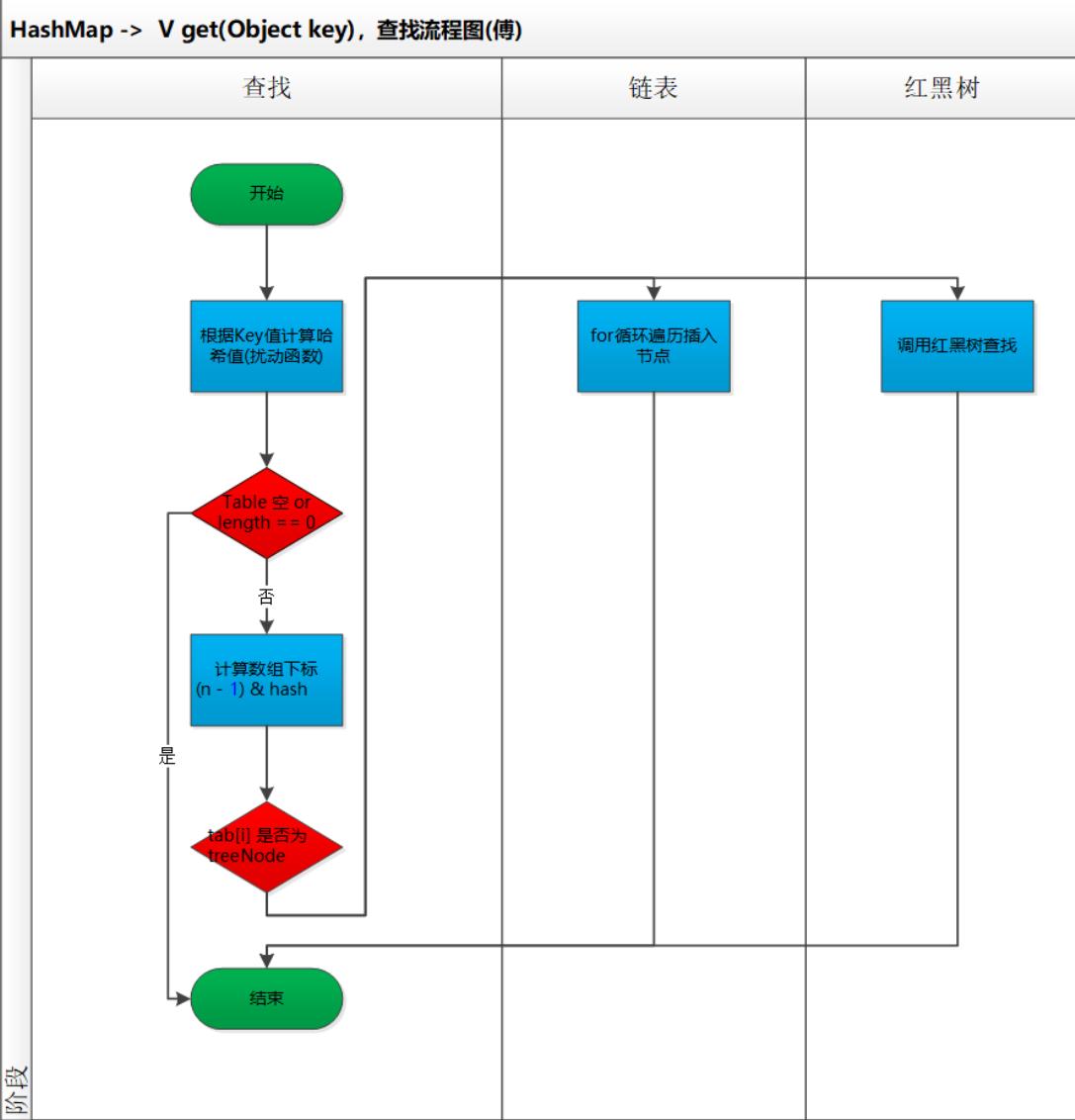
那么，这就简单了，红黑树转链表时候，直接把 TreeNode 转换为 Node 即可，源码如下；

```
final Node<K,V> untreeify(HashMap<K,V> map) {
    Node<K,V> hd = null, tl = null;
    // 遍历TreeNode
    for (Node<K,V> q = this; q != null; q = q.next) {
        // TreeNode 替换Node
        Node<K,V> p = map.replacementNode(q, null);
        if (tl == null)
            hd = p;
        else
            tl.next = p;
        tl = p;
    }
    return hd;
}

// 替换方法
Node<K,V> replacementNode(Node<K,V> p, Node<K,V> next) {
    return new Node<>(p.hash, p.key, p.value, next);
}
```

因为记录了链表关系，所以替换过程很容易。所以好的数据结构可以让操作变得更加容易。

二、查找



公众号：bugstack 虫洞栈，HashMap 查找流程图

上图就是 HashMap 查找的一个流程图，还是比较简单的，同时也是高效的。

接下来我们在结合代码，来分析这段流程，如下；

```

public V get(Object key) {
    Node<K,V> e;
    // 同样需要经过扰动函数计算哈希值
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    // 判断桶数组的是否为空和长度值
    if ((tab = table) != null && (n = tab.length) > 0 &&
        // 计算下标，哈希值与数组长度-1

```

```

        (first = tab[(n - 1) & hash]) != null) {
    if (first.hash == hash && // always check first node
        ((k = first.key) == key || (key != null && key.equals(k))))
        return first;
    if ((e = first.next) != null) {
        // TreeNode 节点直接调用红黑树的查找方法，时间复杂度 O(Logn)
        if (first instanceof TreeNode)
            return ((TreeNode<K,V>)first).getTreeNode(hash, key);
        // 如果是链表就依次遍历查找
        do {
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                return e;
        } while ((e = e.next) != null);
    }
    return null;
}

```

以上查找的代码还是比较简单的，主要包括以下知识点：

1. 扰动函数的使用，获取新的哈希值，这在上一章节已经讲过
2. 下标的计算，同样也介绍过 `tab[(n - 1) & hash]`
3. 确定了桶数组下标位置，接下来就是对红黑树和链表进行查找和遍历操作了

三、删除

```

public V remove(Object key) {
    Node<K,V> e;
    return (e = removeNode(hash(key), key, null, false, true)) == null ?
        null : e.value;
}

final Node<K,V> removeNode(int hash, Object key, Object value,
    boolean matchValue, boolean movable) {
    Node<K,V>[] tab; Node<K,V> p; int n, index;
    // 定位桶数组中的下标位置, index = (n - 1) & hash
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (p = tab[index = (n - 1) & hash]) != null) {

```

```

Node<K,V> node = null, e; K k; V v;
// 如果键的值与链表第一个节点相等，则将 node 指向该节点
if (p.hash == hash &&
    ((k = p.key) == key || (key != null && key.equals(k))))
    node = p;
else if ((e = p.next) != null) {
    // 树节点，调用红黑树的查找方法，定位节点。
    if (p instanceof TreeNode)
        node = ((TreeNode<K,V>)p).getTreeNode(hash, key);
    else {
        // 遍历链表，找到待删除节点
        do {
            if (e.hash == hash &&
                ((k = e.key) == key ||
                 (key != null && key.equals(k)))) {
                node = e;
                break;
            }
            p = e;
        } while ((e = e.next) != null);
    }
}

// 删除节点，以及红黑树需要修复，因为删除后会破坏平衡性。链表的删除更加简单。
if (node != null && (!matchValue || (v = node.value) == value ||
    (value != null && value.equals(v)))) {
    if (node instanceof TreeNode)
        ((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);
    else if (node == p)
        tab[index] = node.next;
    else
        p.next = node.next;
    ++modCount;
    --size;
    afterNodeRemoval(node);
    return node;
}

```

```
    return null;  
}
```

- 删除的操作也比较简单，这里面都没有太多的复杂的逻辑。
- 另外红黑树的操作因为被包装了，只看使用上也是很容易。

四、遍历

1. 问题点

HashMap 中的遍历也是非常常用的 API 方法，包括；

KeySet

```
for (String key : map.keySet()) {  
    System.out.print(key + " ");  
}
```

EntrySet

```
for (HashMap.Entry entry : map.entrySet()) {  
    System.out.print(entry + " ");  
}
```

从方法上以及日常使用都知道，KeySet 是遍历是无序的，但每次使用不同方式遍历包括 `keys.iterator()`，它们遍历的结果是固定的。

那么从实现的角度来看，这些种遍历都是从散列表中的链表和红黑树获取集合值，那么他们有一个什么固定的规律吗？

2. 用代码测试

测试的场景和前提：

1. 这里我们要设定一个既有红黑树又有链表结构的数据场景
2. 为了可以有这样的数据结构，我们最好把 HashMap 的初始长度设定为 64，避免在链表超过 8 位后扩容，而是直接让其转换为红黑树。
3. 找到 18 个元素，分别放在不同节点(这些数据通过程序计算得来)：
 1. 桶数组 02 节点：24、46、68
 2. 桶数组 07 节点：29
 3. 桶数组 12 节点：150、172、194、271、293、370、392、491、590

代码测试

```

@Test
public void test_Iterator() {
    Map<String, String> map = new HashMap<String, String>(64);
    map.put("24", "Idx: 2");
    map.put("46", "Idx: 2");
    map.put("68", "Idx: 2");
    map.put("29", "Idx: 7");
    map.put("150", "Idx: 12");
    map.put("172", "Idx: 12");
    map.put("194", "Idx: 12");
    map.put("271", "Idx: 12");
    System.out.println("排序 01: ");
    for (String key : map.keySet()) {
        System.out.print(key + " ");
    }

    map.put("293", "Idx: 12");
    map.put("370", "Idx: 12");
    map.put("392", "Idx: 12");
    map.put("491", "Idx: 12");
    map.put("590", "Idx: 12");
    System.out.println("\n\n 排序 02: ");
    for (String key : map.keySet()) {
        System.out.print(key + " ");
    }

    map.remove("293");
    map.remove("370");
    map.remove("392");
    map.remove("491");
    map.remove("590");
    System.out.println("\n\n 排序 03: ");
    for (String key : map.keySet()) {
        System.out.print(key + " ");
    }
}

```

这段代码分别测试了三种场景，如下：

1. 添加元素，在 HashMap 还是只链表结构时，输出测试结果 01
2. 添加元素，在 HashMap 转换为红黑树时候，输出测试结果 02
3. 删除元素，在 HashMap 转换为链表结构时，输出测试结果 03

3. 测试结果分析

排序 01:

```
24 46 68 29 150 172 194 271
```

排序 02:

```
24 46 68 29 271 150 172 194 293 370 392 491 590
```

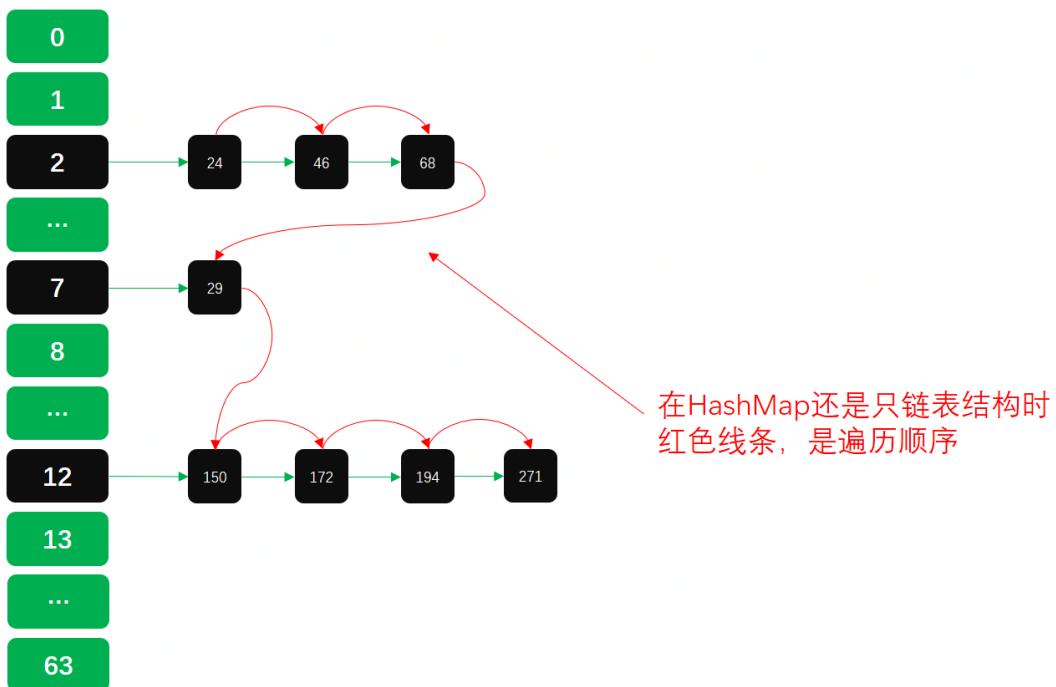
排序 03:

```
24 46 68 29 172 271 150 194
```

```
Process finished with exit code 0
```

从 map.keySet() 测试结果可以看到，如下信息；

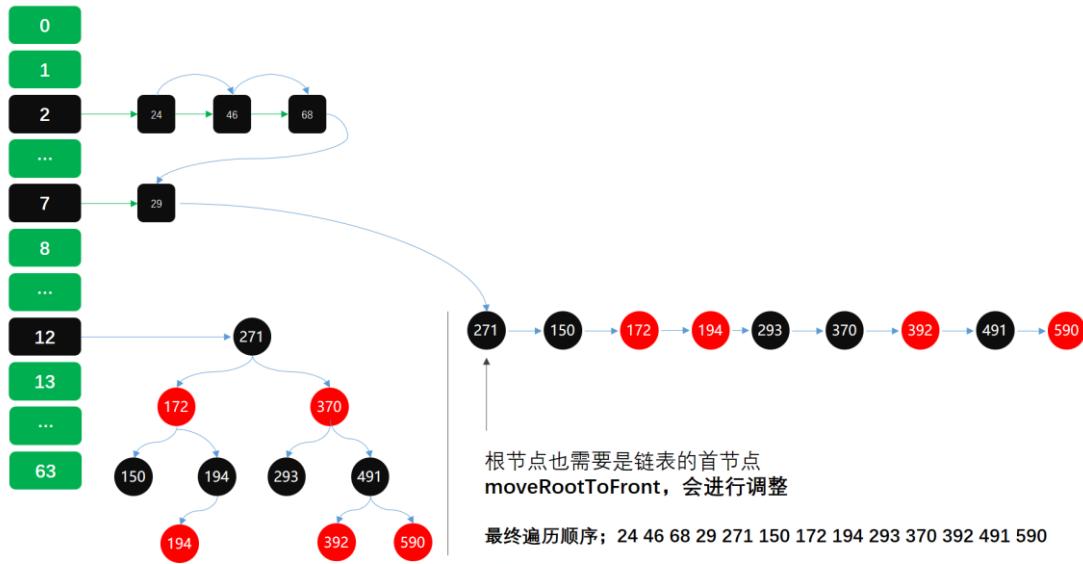
1. 01 情况下，排序定位哈希值下标和链表信息



公众号：bugstack 虫洞栈，链表结构

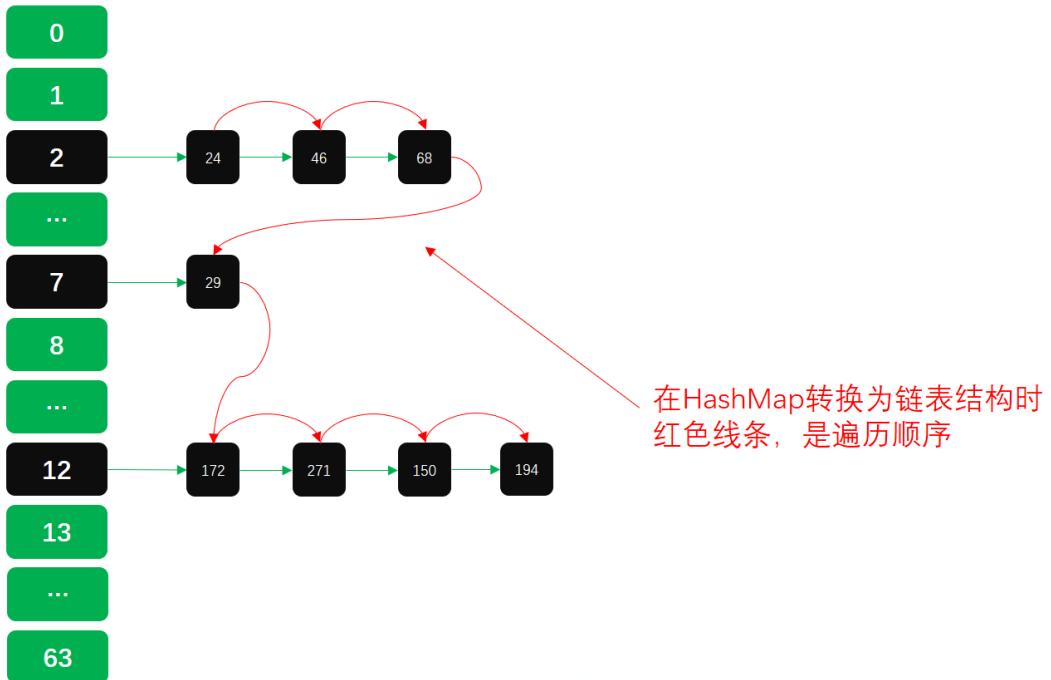
2. 02 情况下，因为链表转换为红黑树，树根会移动到数组头部。

[moveRootToFront\(\)方法](#)



公众号: bugstack 虫洞栈, 链表树化

3. 03 情况下, 因为删除了部分元素, 红黑树退化成链表。



公众号: bugstack 虫洞栈, 红黑树转链表

五、总结

- 这一篇 API 源码以及逻辑与上一篇数据结构中扰动函数、负载因子、散列表实现等，内容的结合，算是把 HashMap 基本常用技术点，梳理完成了。但知识绝不止于此，这里还有红黑树的相关技术内容，后续会进行详细。
- 除了 HashMap 以外还有 TreeMap、ConcurrentHashMap 等，每一个核心类都有一些相关的核心知识点，每一个都非常值得深入研究。这个烧脑的过程，是学习获得知识的最佳方式。
- 可能关于 HashMap 还有一些疏漏的点，也希望阅读的小伙伴可以提出更多的问题，互相学习，共同进步，本文就到这里，感谢您的阅读！

第 4 节：2-3 树与红黑树学习(上)

讲道理 5 年开发，没用过数据结构，你只是在做 CRUD！

很多时候大部分程序员  头疼于，查询慢、效率低、一堆的关联 SQL，主要原因是在程序设计上没有做出很好的数据结构。当然也还有一部分是由于老业务代码，或者没有用到一些大数据服务等。

数据结构、算法、设计模式，是每一个程序员成长过程中的内功心法修炼，而你的新技能用的再绚、多线程使的再 6、加锁玩的再牛 ，也只能说明你这个人身体好，但身体好是不能抗拒子弹的。**只有身体+心法都好，都能纵横捭阖。**

这一章节是结合 [HashMap](#) 的延展，在 Jdk1.8 中 HashMap 是使用 [桶数组+链表](#) 和 [红黑树](#) 实现，所以顺着上一章节的核心原理和 API 功能讲解后，本来这一章节想直接进入到红黑树，但如果想把红黑树学明白，就需要了解他的来龙去脉，也就是它的前身 [2-3 树](#)。

一、面试题

谢飞机，考你几个简单的知识点

1. 飞机，看你简历写了解数据结构，可以简单介绍下 2-3 树吗？
2. 这种树节点有什么特点，与你了解其他的树结构对比下？
3. 你看这个图，向里面插入和删除节点，要怎么操作？

飞机，回去等消息吧！

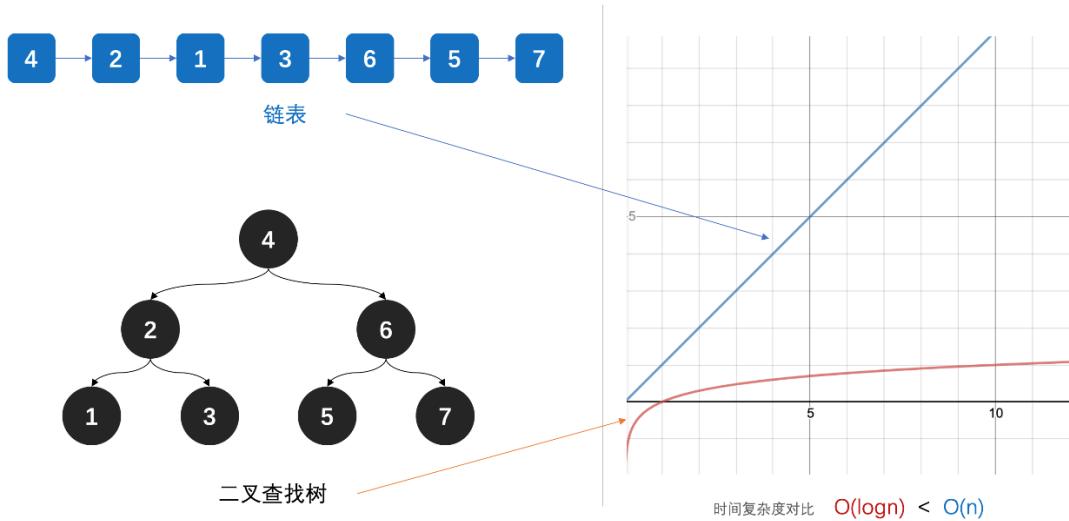
二、什么是 2-3 树

日常的学习和一部分伙伴的面试中，竟然会听到的是；从 [HashMap](#) 中文红黑树、从数据库索引为 B+Tree，但问 2-3 树的情况就不是很多了。

1. 为什么使用树结构

从最根本的原因来看，使用树结构就是为了提升整体的效率；插入、删除、查找（索引），尤其是索引操作。因为相比于链表，一个平衡树的索引时间复杂度是 $O(\log n)$ ，而数组的索引时间复杂度是 $O(n)$ 。

从以下的图上可以对比，两者的索引耗时情况；

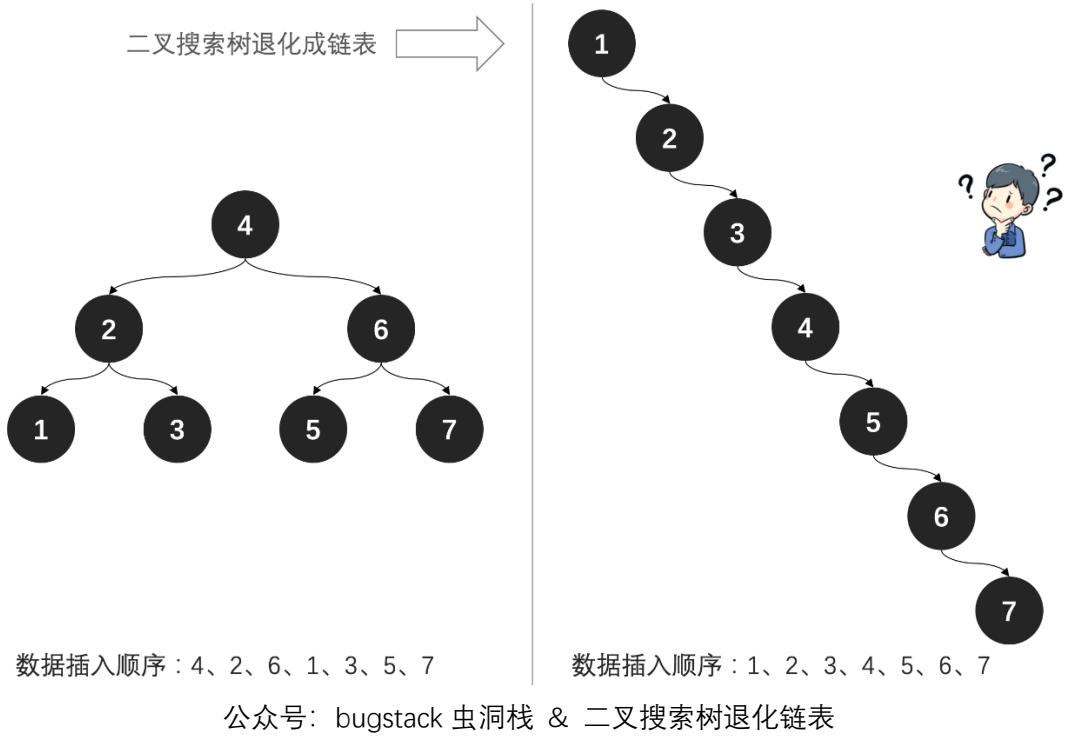


公众号：bugstack 虫洞栈 & 链表与二叉搜索树(Binary Search Tree)时间复杂度对比

- 从上图可以看到，使用树结构有效的降低时间复杂度，提升数据索引效率。
- 另外这个标准的树结构，是二叉搜索树(Binary Search Tree)。除此之外树形结构还有；AVL 树、红黑树、2-3 树等

2. 二叉搜索树退化链表

在树的数据结构中，最先有点是二叉查找树，也就是英文缩写 BST 树。在使用数据插入的过程中，理想情况下它是一个平衡的二叉树，但实际上可能会出现二叉树都一边倒，让二叉树像列表一样的数据结构。从而树形结构的时间复杂度也从 $O(\log n)$ 升级到 $O(n)$ ，如下图；



- 二叉搜索树的数据插入过程是，插入节点与当前树节点做比对，小于在左，大于在右。
- 随着数据的插入顺序不同，就会出现完全不同的数据结构。可能是一棵平衡二叉树，也极有可能退化成链表的树。
- 当树结构退化成链表以后，整个树索引的性能也跟着退化成链表。

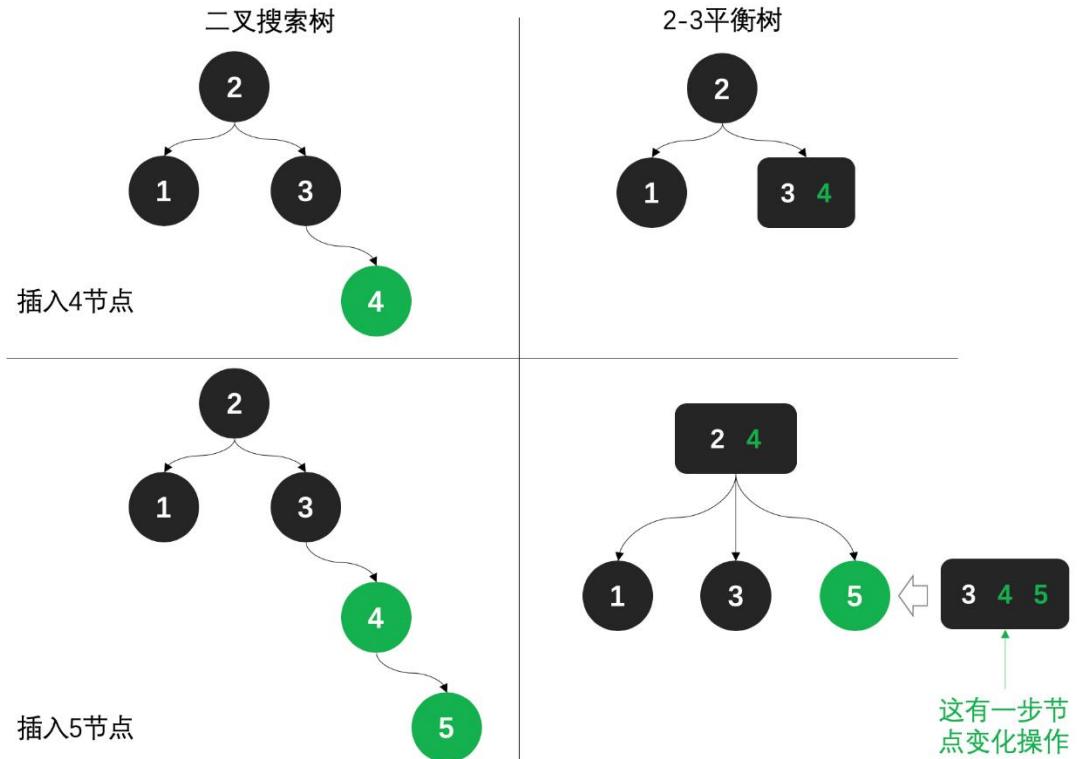
综上呢，如果我们希望在插入数据后又保持树的特点， $O(\log n)$ 的索引性能，那么就需要在插入时进行节点的调整

3. 2-3 树解决平衡问题

2-3 树是什么结构，它怎么解决平衡问题的。带着问题我们继续。

2-3 树是一种非常巧妙的结构，在保持树结构的基础上，它允许在一个节点中可以有两个元素，等元素数量等于 3 个时候再进行调整。通过这种方式呢，来保证整个二叉搜索树的平衡性。

这样说可能还没有感觉，来看下图：



公众号: bugstack 虫洞栈 & 2-3 树解决平稳问题

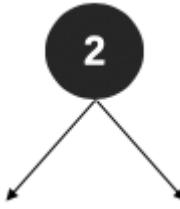
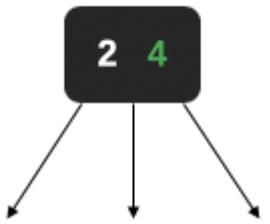
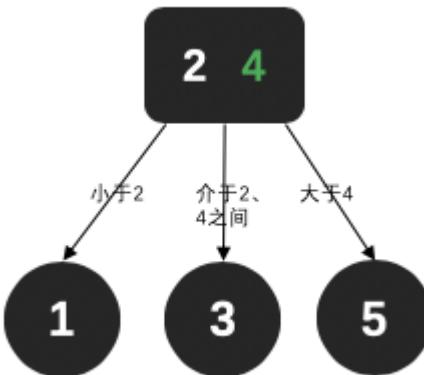
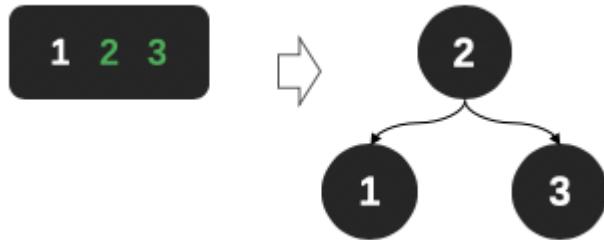
- 左侧是二叉搜索树，右侧是 2-3 平衡树，分别插入节点 4、5，观察树形结构变化。
- 二叉搜索树开始出现偏移，节点一遍倒。
- 2-3 树通过一个节点中存放 2 到 3 个元素，来调整树形结构，保持平衡。所谓的保持平衡就是从根节点，到每一个最底部的自己点，链路长度一致。

2-3 树已经可以解决平衡问题那么，数据是怎么存放和调整的呢，接下来我们开始分析。

三、2-3 树使用

1. 树结构定义和特点性质

2-3 树，读法；二三树，特性如下；

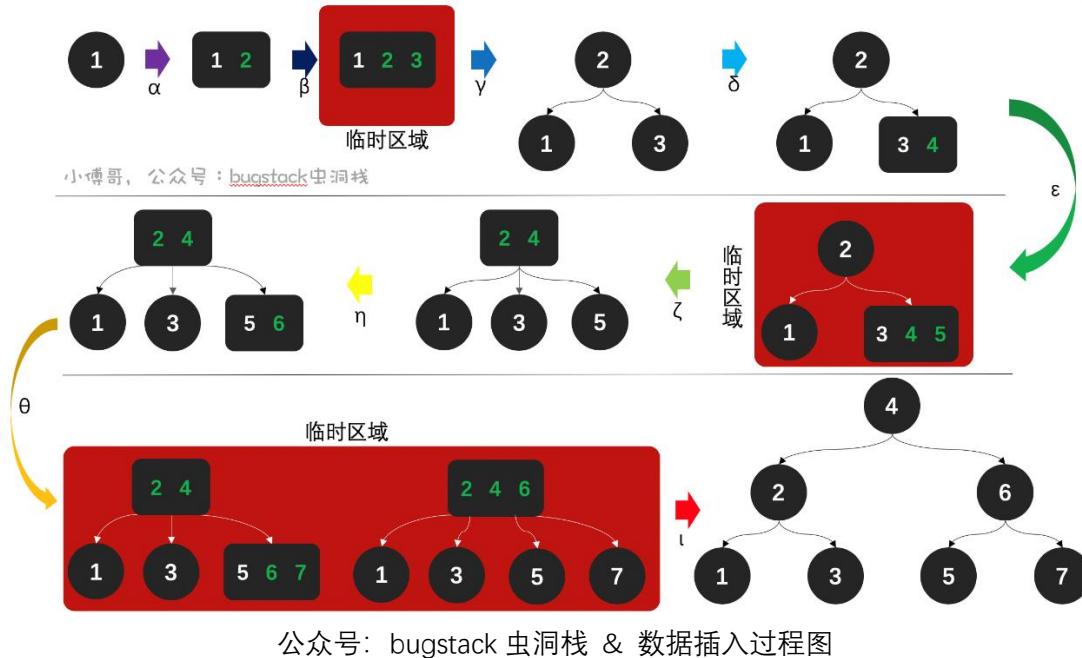
序号	描述	示意图
1	2-, 1 个数据节点 2 个树权	
2	3-, 2 个数据节点 3 个树权	
3	三叉与两叉的不同点在于，除了两边的节点，中间还有一个节点。这个节点是介于 2、4 之间的值。	
4	当随着插入数据，会出现临时的一个节点中，有三个元素。这时会被调整成一个二叉树。	

综上我们可以总结出，2-3 树的一些性质；

1. 2-3 树所有子叶节点都在同一层
2. 1 个节点可以有 1 到 2 个数据，如果有三个需要调整树结构
3. 1 个节点 1 个数据时，则有两个子节点
4. 1 个节点 2 个数据时，则有三个子节点，且中间子节点是介于两个节点间的值

2. 数据插入

接下来我们就模拟在二叉搜索树中退化成链表的数据，插入到 2-3 树的变化过程，数据包括：1、2、3、4、5、6、7，插入过程图如下：



以上，就是整个数据在插入过程中，2-3 树的演化过程，接下来我们具体讲解每一步的变化；

- α ，向节点 1 插入数据 2，此时为了保持平衡，不会新产生分支，只会在一个节点中存放两个节点。
- β ，继续插入数据 3，此时这个节点有三数据，1、2、3，是一个临时区域。
- γ ，把三个数据的节点，中间节点拉起来，调整成树形结构。
- δ ，继续插入数据 4，为了保持树平衡，会插在节点 3 的右侧。
- ε ，继续插入数据 5，插入后 3、4、5 共用 1 个节点，当一个节点上有三个数据时，则需要进行调整。
- ζ ，中间节点 4 向上 调整，调整后，1 节点在左、3 节点在中间、5 节点在右。
- η ，继续插入数据 6，在保持树平衡的情况下，与节点 5 公用。
- θ ，继续插入数据 7，插入后，节点 7 会与当前的节点 5、6 共用。此时是一个临时存放，需要调整。初步调整后，抽出 6 节点，向上存放，变为 2、4、6 共用一个节点，这是一个临时状态，还需要继续调整。
- ι ，因为根节点有三个数据 2、4、6，则继续需要把中间节点上移，1、3 和 5、7 则分别成二叉落到节点 2、节点 6 上。

GR希腊字母: α (阿尔法)、 β (贝塔)、 γ (伽马)、 δ (德尔塔)、 ε (伊普西隆)、 ζ (截塔)、 η (艾塔)、 θ (西塔)、 ι (约塔)

3. 数据删除

有了上面数据插入的学习，在看数据删除其实就是一个逆向的过程，在删除的主要包括这样两种情况；

1. 删除了 3-节点，也就是包含两个数据元素的节点，直接删除即可，不会破坏树平衡。
2. 删除了 2-节点，此时会破坏树平衡，需要将树高缩短或者元素合并，恢复树平衡。

承接上面的例子，我们把数据再从 7、6、5、4、3、2、1 顺序删除，观察 2-3 树的结构变化，如下；

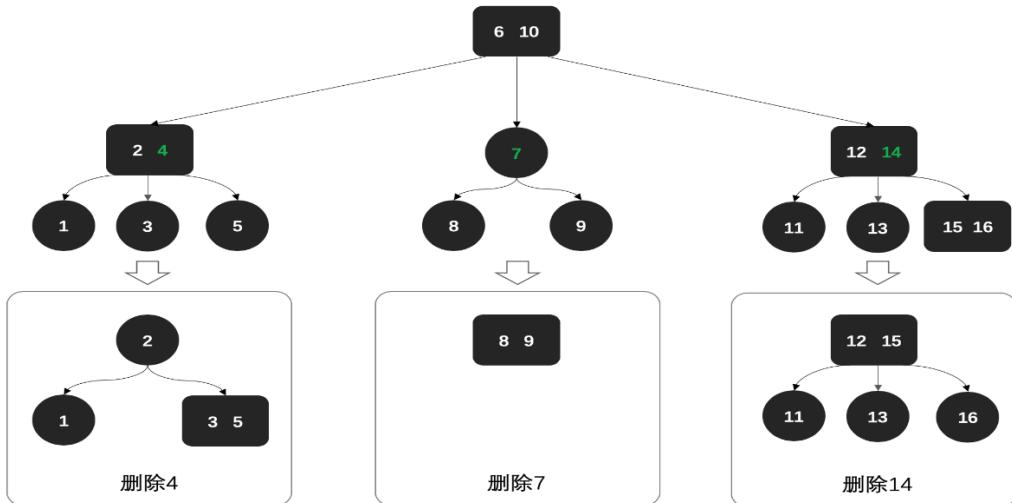


公众号: bugstack 虫洞栈 & 数据删除过程图

- α , 删除节点 7，因为节点 7 只有一个数据元素，删除节点 5、6 合并，但此时破坏了 2-3 树的平衡性，需要缩短树高进行调整。
- β , 因为删除节点后，整个树结构不平衡，所以需要缩短树高，调整元素。节点 2、4 合并，节点 1、3 分别插入左侧和中间。
- γ , 删除节点 6，这个节点是 3-节点(可以分出 3 个叉的意思)，删除后不会破坏树平衡，保持不变。
- δ , 删除节点 5，此时会破坏树平衡，需要把跟节点 4 下放，与 3 合并。

- ε , 删除节点 4, 这个节点依旧是 3-节点, 所以不需要改变树结构。
- ζ , 删除节点 3, 此时只有 1、2 节点, 需要合并。
- η , 删除节点 2, 此时节点依旧是 3-节点, 所以不需要改变树结构。

再看一个稍微复杂点 2-3 树删除:



公众号: bugstack 虫洞栈 & 复杂树删除过程

上面这张图, 就一个稍微复杂点的 2-3 平衡树, 树的删除过程主要包括:

1. 删除 4, 其实需要将节点 3、5 合并, 指向节点 2, 保持树平衡。
2. 删除 7, 节点 8、9 合并。
3. 删除 14, 节点 15 上移, 恢复成 3-叉树。

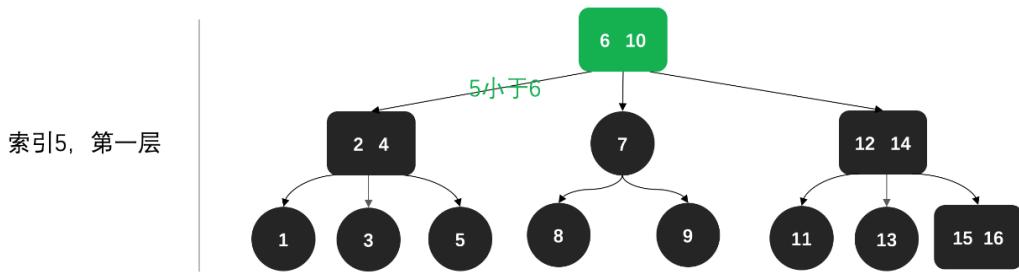
如果有时候不好理解删除, 可以试想下, 这个要删除的节点, 在插入的时候是一个什么效果。

4. 数据索引

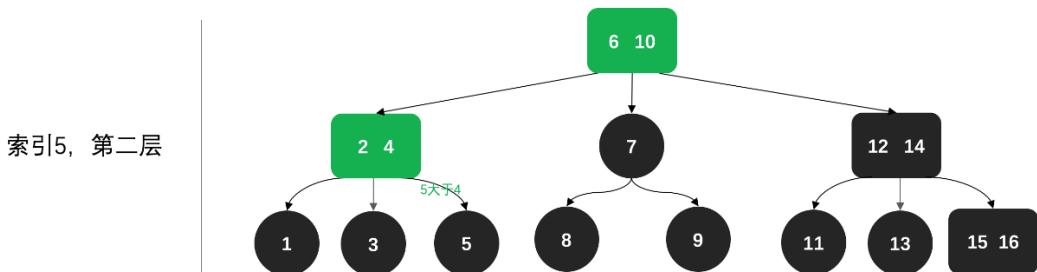
相比于插入和删除, 索引的过程还是比较简单的, 不需要调整数据结果。基本原则就是:

1. 小于当前节点值, 左侧寻找
2. 大于当前节点值, 右侧寻找
3. 一直到找到索引值, 停止。

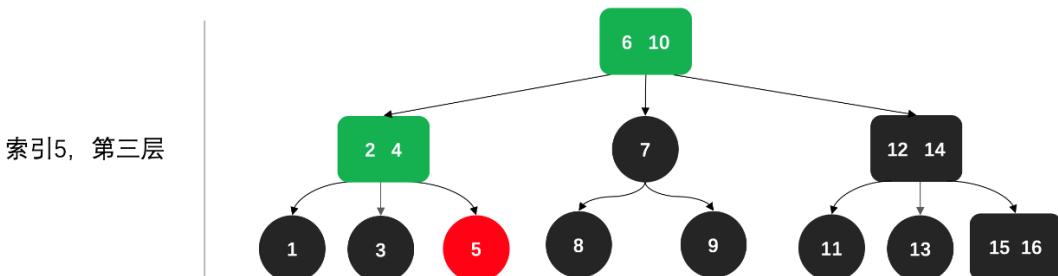
❑ 第一层寻找:



Q 第二层寻找:



Q 第三次寻找:



四、总结

- 综上讲解了 2-3 树的核心内容，通过本章节的学习，可以了解 2-3 树是一种怎样的数据结构、如何插入数据、删除数据以及数据的索引，同时要知道这是一种平衡树的结构，包括 2-叉和 3-叉节点以及数结构随着数据的添加删除调整。
- 2-3 树是红黑树的演变前身，通过这一章节的学习就很容易学习红黑树的相关知识，在红黑树中添加数据进行的渲染、旋转等来保持树平衡。**红黑树接近平衡**
- 数据结构方面的知识学习起来，可能会比较 烧脑，因为需要思考出那种模型结构和变化的过程，所以会感觉困难。但这个烧脑的过程也是对学习非常有帮助的，可以迅速建设知识凸起，当突破不理解到理解，可以有非常多的收获。

第 5 节：2-3 树与红黑树学习(下)

红黑树，是一种高效的自平衡二叉查找树

Rudolf Bayer 于 1972 年发明红黑树，在当时被称为对称二叉 B 树(symmetric binary B-trees)。后来，在 1978 年被 Leo J. Guibas 和 Robert Sedgewick 修改为如今的红黑树。

红黑树具有良好的效率，它可在近似 $O(\log N)$ 时间复杂度下完成插入、删除、查找等操作，因此红黑树在业界也被广泛应用，比如 Java 中的 TreeMap，JDK 1.8 中的 HashMap、C++ STL 中的 map 均是基于红黑树结构实现的。

死记硬背，很难学会

红黑树的结构和设计都非常优秀，也同样在实现上有着复杂的处理逻辑，包括插入或者删除节点时；颜色变化、旋转操作等操作。但如果只把这些知识点硬背下来，什么时候染色、什么时候旋转，是没有多大意义的，用不了多久也就忘记了。

所以这部分的学习，了解其根本更重要。

一、面试题

谢飞机，考你几个红黑树的知识点

1. 红黑树的数据结构都用在哪些场景，有什么好处？
2. 红黑树的时间复杂度是多少？
3. 红黑树中插入新的节点时怎么保持平衡？

飞机，2-3 树是不没看，回去等消息吧！

二、2-3 树与红黑树的等价性

在上一章节，使用了大量图例讲解了 2-3 树，并在标题处写出它是红黑树的前身。阅读后更容易理解红黑树相关知识。

红黑树规则

1. 根节点是黑色
2. 节点是红黑或者黑色
3. 所有子叶节点都是黑色(叶子是 NIL 节点，默认没有画出来)

4. 每个红色节点必须有两个黑色子节点(也同样说明一条链路上不能有链路的红色节点)

5. 黑高, 从任一节点到齐每个叶子节点, 经过的路径都包含相同数目的黑色节点

那么, 这些规则是怎么总结定义出来的呢? 接下里我们一步步分析讲解。

1. 为什么既有 2-3 树要有红黑树

首先 **2-3 树** (读法: 二三树) 就是一个节点有 1 个或者 2 个元素, 而实际上 2-3 树转红黑树是由概念模型 **2-3-4 树**转换而来的。-4 叉就是一个节点里有 3 个元素, 这在 2-3 树中会被调整, 但是在概念模型中是会被保留的。

虽然 **2-3-4 树**也是具备 **2-3 树**同样的平衡树的特性, 但是如果直接把这样的模型用代码实现就会很麻烦, 且效率不高, 这里的复杂点包括:

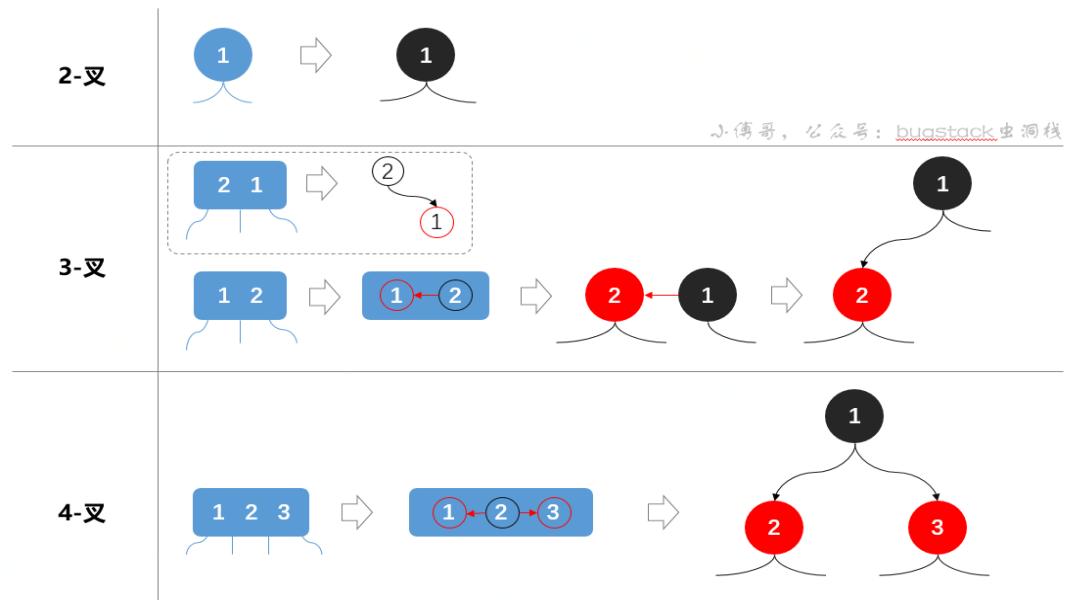
1. 2-叉、3-叉、4-叉, 三种结构的节点类型, 互相转换复杂度较高
2. 3-叉、4-叉, 节点在数据比较上需要进行多次, 不像 2-叉节点, 直接布尔类型比较即可非左即右
3. 代码实现上对每种差异, 都需要有额外的代码, 规则不够标准化

所以, 希望找到一种平衡关系, 既保持 2-3 树平衡和 $O(\log n)$ 的特性, 又能在代码实现上更加方便, 那么就诞生了红黑树。

2. 简单 2-3 树转红黑树

2-3 树转红黑树, 也可以说红黑树是 **2-3 树**和 **2-3-4 树**的另外一种表现形式, 也就是更利于编码实现的形式。

简单转换示例:



2-叉、3-叉、4-叉, 转换红黑树示意图

从上图可以看出，2-3-4 树与红黑树的转换关系，包括：

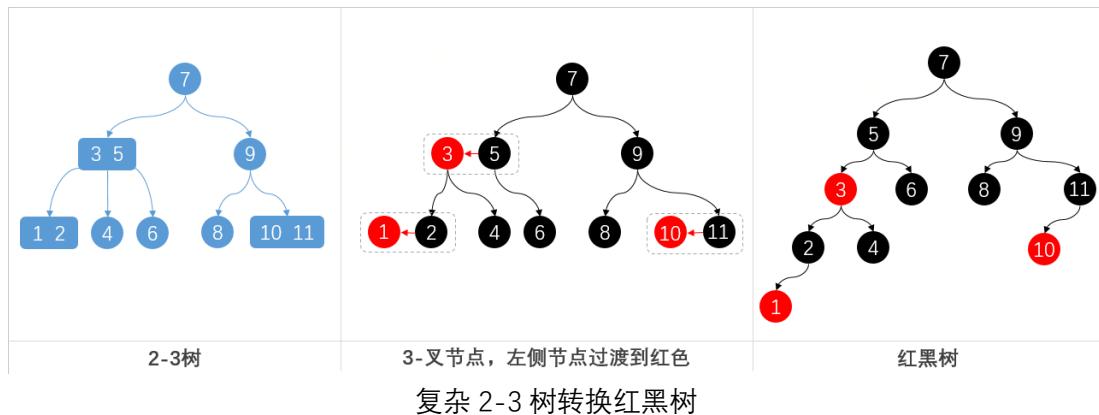
1. 2-叉节点，转换比较简单，只是把原有节点转换为黑色节点
2. 3-叉节点，包括了 2 个元素，先用红色线把两个节点相连，之后拆分出来，最后调整高度黑色节点在上
3. 4-叉节点，包括了 3 个元素，分别用红黑线连接，之后拆分出来拉升高度。这个拉升过程和 2-3 树调整一致，只是添加了颜色

综上，就是 2-3-4 树的节点转换，总结出来的规则，如下：

1. 将 2-3-4 树，用二叉树的形式表示
2. 3-叉、4-叉节点，使用红色、黑色连线进行连接
3. 另外，3-叉节点有两种情况，导致转换成二叉树，就有左倾和右倾

3. 复杂 2-3 树转红黑树

在[简单 2-3 树转换红黑树](#)的过程中，了解到一个基本的转换规则右旋定义，接下来我们在一个稍微复杂一点的[2-3 树](#)与红黑树的对应关系，如下图；



上图是一个稍微复杂点的 2-3 树，转换为红黑树的过程，是不这样一张图让你对红黑树更有感觉了，同时它也满足一下条件；

1. 从任意节点到叶子节点，所经过的黑色节点数目相同
2. 黑色节点保持着整体的平衡性，也就是让整个红黑树接近于 $O(\log n)$ 时间复杂度
3. 其他红黑树的特点也都满足，可以对照红黑树的特性进行比对

三、红黑树

1. 平衡操作

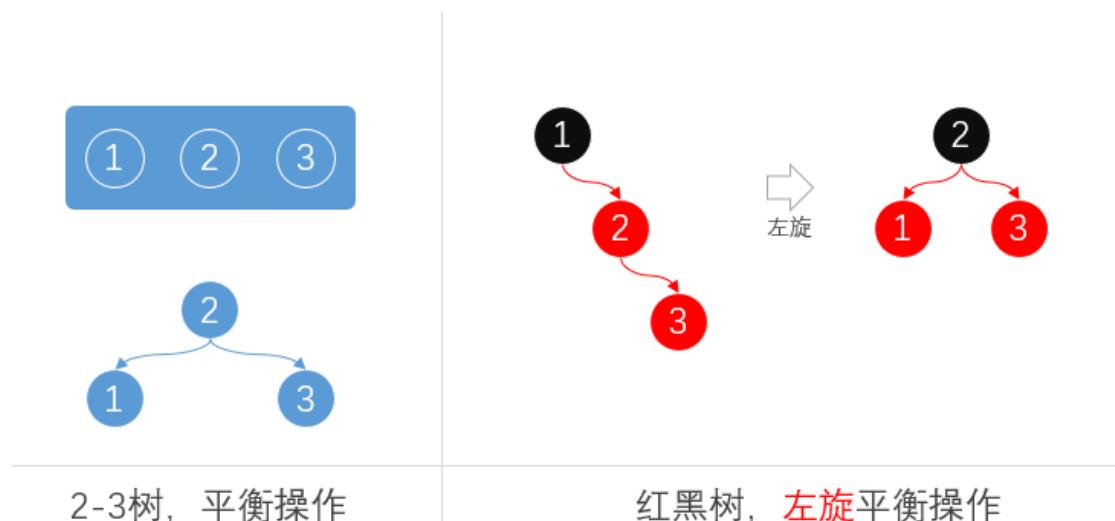
通过在上一章节 2-3 树的学习，在插入节点时并不会插到空位置，而是与现有节点融合以及调整，保持整个树的平衡。

而红黑树是 2-3-4 树的一种概念模型转换而来，在插入节点时通过红色链接相连，也就是插入红色节点。插入完成后进行调整，以保持树接近平衡。

那么，为了让红黑树达到平衡状态，主要包括染色、 \leftrightarrow 左右旋转、这些做法其实都是从 2-3 树演化过来的。接下来我们就分别讲解几种规则的演化过程，以此更好了解红黑树的平衡操作。

1.1 左旋转

左旋定义：把一个向右倾斜的红节点链接(2-3 树，3-叉双元素节点)，转化为左链接。



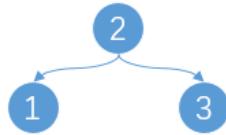
背景：顺序插入元素，1、2、3，2-3 树保持平衡，红黑树暂时处于右倾斜。
接下来我们分别对比两种树结构的平衡操作；

1. 2-3 树，所有插入的节点都会保持在一个节点上，之后通过调整节点位置，保持平衡。
2. 红黑树，则需要通过节点的左侧旋转，将元素 2 拉起来，元素 1 和元素 3，分别成为左右子节点。

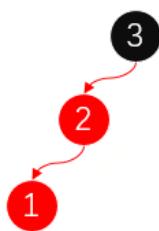
红黑树的左旋，只会处理与之对应的 2-3 树节点进行操作，不会整体改变。

1.2 右旋转

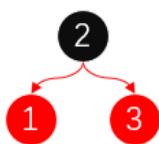
右旋定义：把一个向左倾斜的红节点链接(2-3 树，3-叉双元素节点)，转换为右链接。



2-3树，平衡操作



右旋



红黑树，右旋平衡操作

背景：顺序插入元素，3、1、1，2-3 树保持平衡，红黑树暂时处于左倾斜。

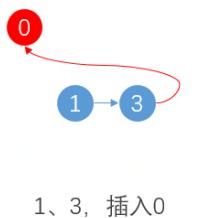
接下来我们分别对比两种树结构的平衡操作；

1. 2-3 树，所有插入的节点都会保持在一个节点上，之后通过调整节点位置，保持平衡。
2. 红黑树，则需要通过节点的右侧旋转，将元素 2 拉起来，元素 1 和元素 3，分别为左右子节点。

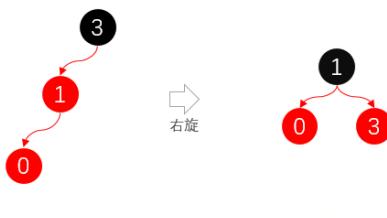
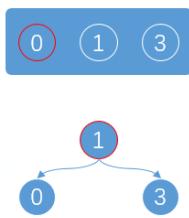
你会发现，左旋与右旋是相互对应的，但在 2-3 树中是保持不变的

1.3 左右旋综合运用

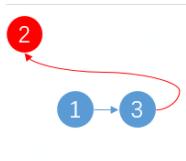
左旋、右旋，我们已经有了一个基本的概念，那么接下来我们再看一个可以综合左右旋以及对应 2-3 树的演化案例，如下：



1、3，插入0



右旋

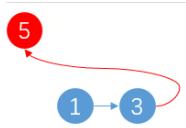


1、3，插入2

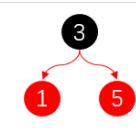
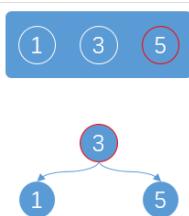


左旋

右旋



1、3，插入5



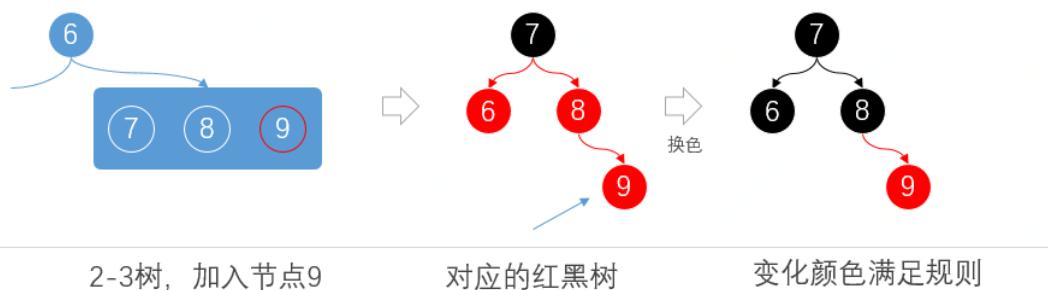
小傅哥，公众号：[bugstack 虫洞栈](#)

以上的例子分别演示了一个元素插入的三种情况，如下：

1. 1、3，插入 0，左侧底部插入，与 2-3 树相比，需要右旋保持平衡
2. 1、3，插入 2，中间位置插入，首先进行左旋调整元素位置，之后进行右旋进行树平衡
3. 1、3，插入 5，右侧位置插入，此时正好保持树平衡，不需要调整

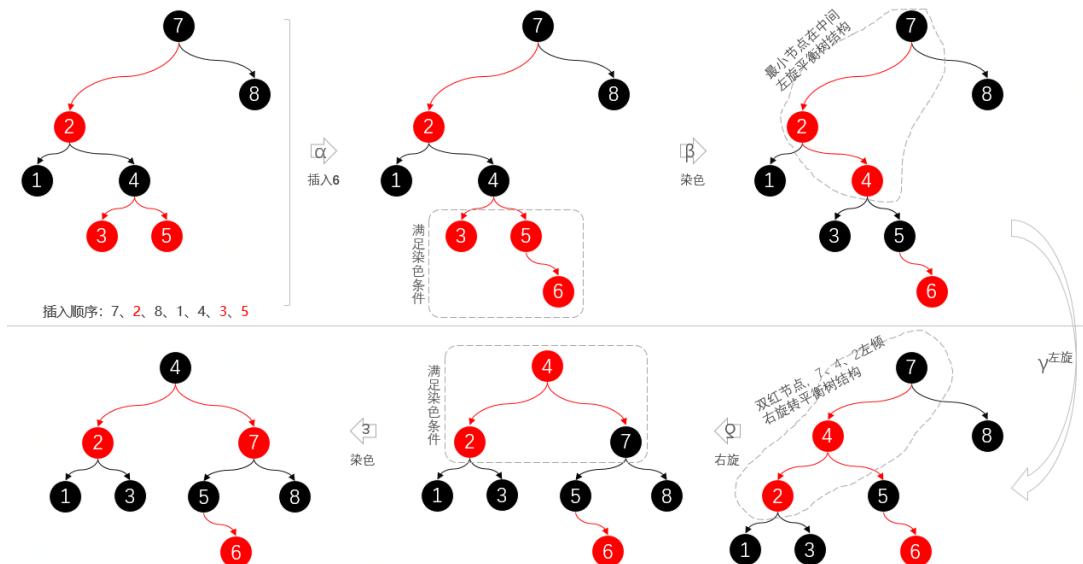
1.4 染色

在 2-3 树中，插入一个节点，为了保持树平衡是不插入到空位置上的，当插入节点后元素数量有 3 个后则需要调整中间元素向上，来保持树平衡。与之对应的红黑树则需要调整颜色，来保证红黑树的平衡规则，具体参考如下：



2. 旋转+染色运用案例

接下来我们把上面讲解到的 **旋转**、**染色**，运用到一个实际案例中，如下图：



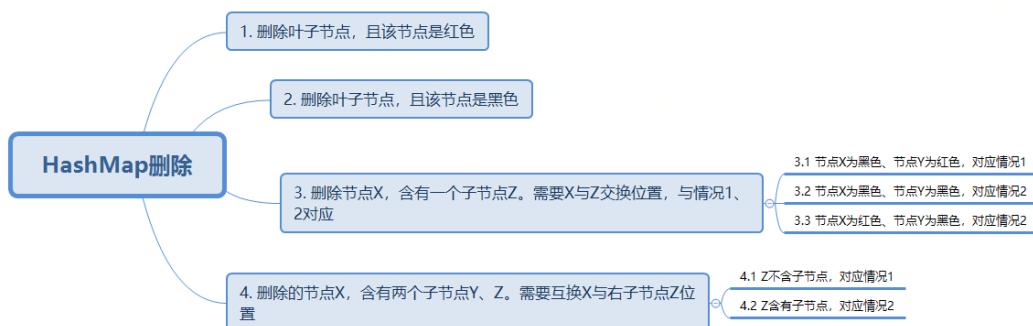
- 首先从左侧开始，是一个按照顺序插入生产出来的红黑树，插入顺序；7、2、8、1、4、3、5

- α , 向目前红黑树插入元素 6, 插入后右下角有三个红色节点; **3、5、6**。
- β , 因为右下角满足染色条件, 变换后; 黑色节点(3、5)、红色节点(4、6)。
- γ , 之后看被红色连线链接的节点 **7、4、2**, 最小节点在中间, 左旋平衡树结构。
- δ , 左旋完成后, 红色链接线的 **7、4、2** 为做倾顺序节点, 因此需要做右旋操作。
- ϵ , 左旋、右旋, 调整完成后, 又满足了染色操作。到此恢复红黑树平衡。

注意, 所有连接红色节点的, 都是红色线。以此与 2-3 树做对应。

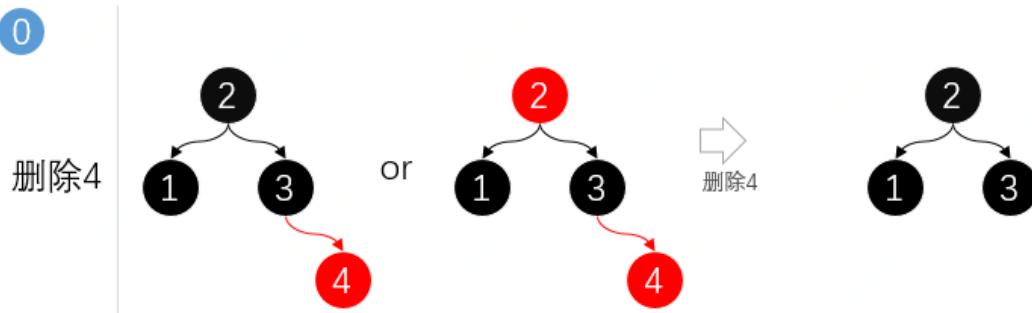
3. 删除操作

根据 2-3-4 树模型的红黑树, 在删除的时候基本是按照 2-3 方式进行删除, 只不过在这个过程中需要染色和旋转操作, 以保持树平衡。删除过程主要可以分为如图四种情况, 如下:



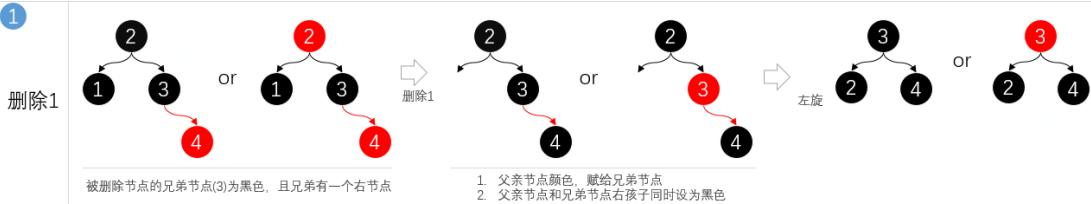
3.1 删除子叶红色节点

红色子叶节点的删除并不会破坏树平衡, 也不影响树高, 所以直接删除即可, 如下:

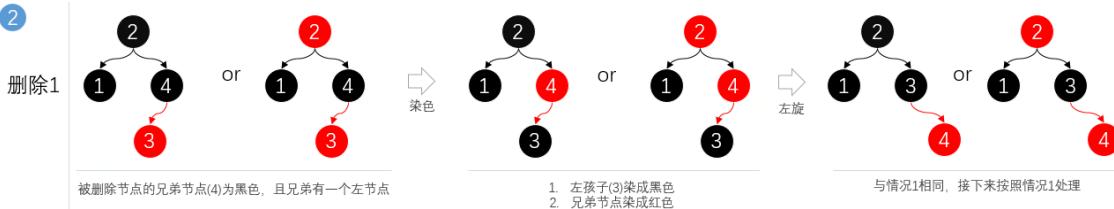


3.2 删除左侧节点

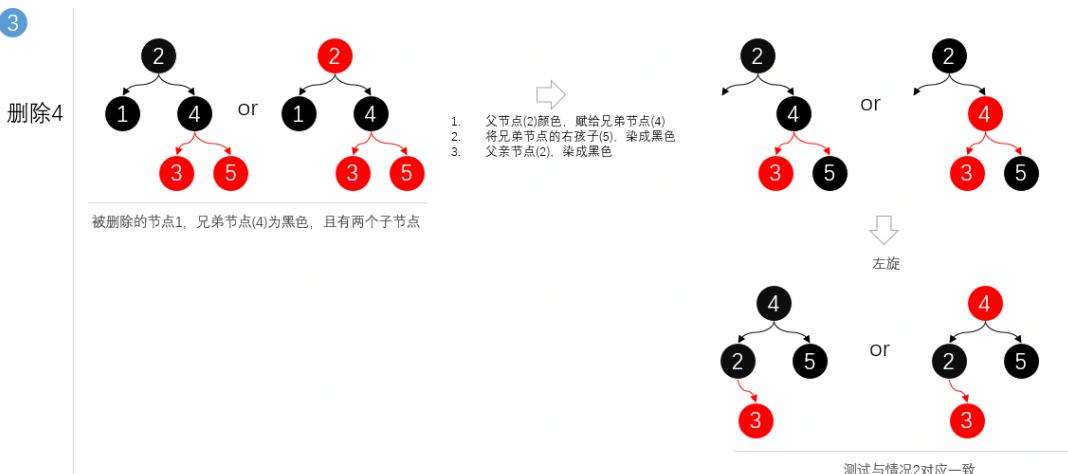
3.2.1 被删节点兄弟为黑色&含右子节点



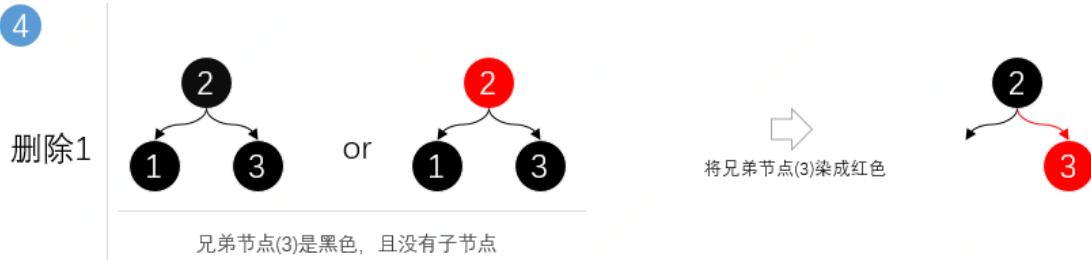
3.2.2 被删节点兄弟为黑色&含左子节点



3.2.3 被删节点兄弟为黑色&含双子节点(红)



3.2.4 被删节点兄弟为黑色&不含子节点

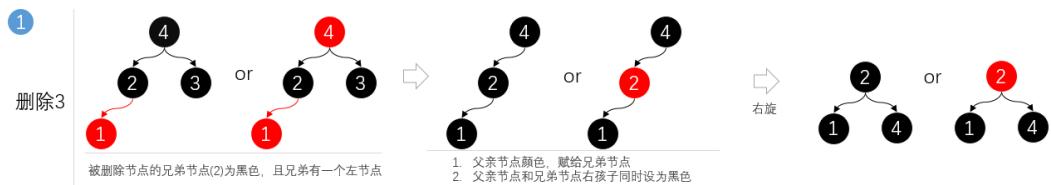


3.2.5 被删节点兄弟为黑色&含双黑节点(黑)

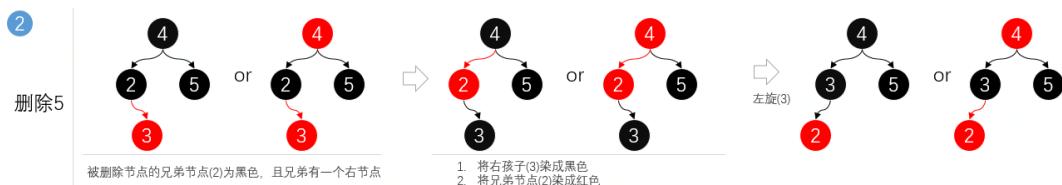


3.3. 删除右侧节点

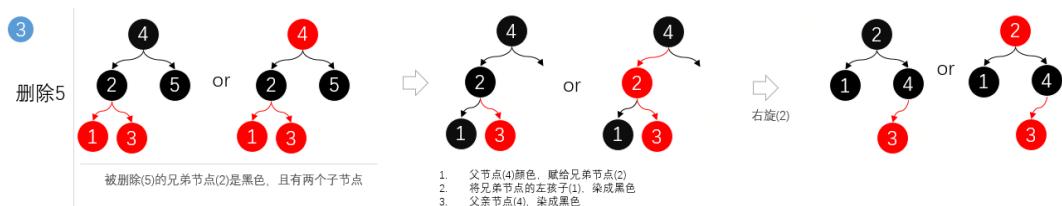
3.3.1 被删节点兄弟为黑色&含左子节点



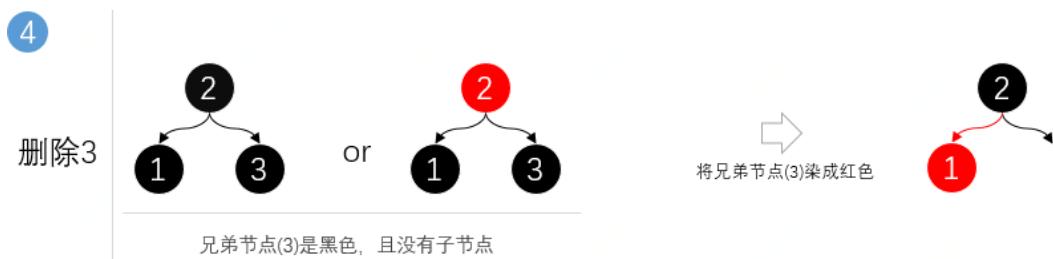
3.3.2 被删节点兄弟为黑色&含右子节点



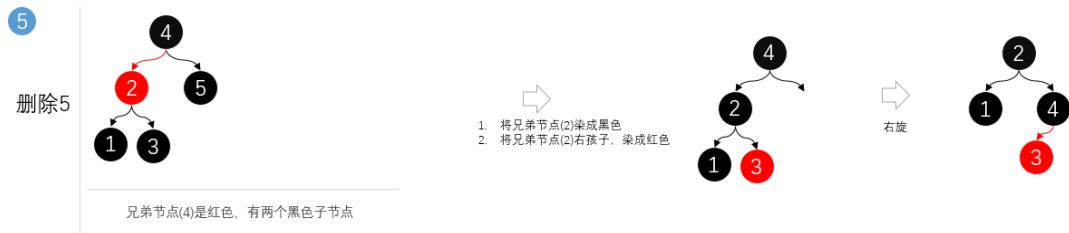
3.3.3 被删节点兄弟为黑色&含双子节点(红)



3.2.4 被删节点兄弟为黑色&不含子节点



3.2.5 被删节点兄弟为黑色&含双黑节点(黑)



四、总结

- 从 2-3 树到解释 2-3-4 树概念推导出红黑树，从元素的在 2-3 树中的插入删除对照到红黑树中保持平衡操作，从原理解析到各项情况实际操作等，以及把绝大部分红黑树内容全部介绍完成。
- 红黑树的原理理解要比背概念更重要，这是一种数据结构的学习，更重要的是技术迁移学习，而不是为了面试背几道题。可能这个学习过程非常烧脑，但适合学习根本。
- 在编写本篇文章时，参考了大量的资料进行校正，包括优秀文章：
 - 红黑树可视化：
<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>
 - 做倾红黑树论文：[Left-leaning Red-Black Trees](#)

第 6 节：ArrayList 详细分析

数据结构是写好代码的基础！

说到数据结构基本包括；数组、链表、队列、红黑树等，但当你看到这些数据结构以及想到自己平时的开发，似乎并没有用到过。那么为什么还要学习数据结构？其实这些知识点你并不是没有用到的，而是 Java 中的 API 已经将各个数据结构封装成对应的工具类，例如 ArrayList、LinkedList、HashMap 等，就像在前面的章节中，小傅哥写了 5 篇文章将近 2 万字来分析 HashMap，从而学习它的核心设计逻辑。

可能有人觉得这类知识就像八股文，学习只是为了应付面试。如果你真的把这些用于支撑其整个语言的根基当八股文学习，那么硬背下来不会有多少收获。理科学习更在乎逻辑，重在是理解基本原理，有了原理基础就复用这样的技术运用到实际的业务开发。

那么，你什么时候会用到这样的技术呢？就是，当你考虑体量、夯实服务、琢磨性能时，就会逐渐的深入到数据结构以及核心的基本原理当中，那里的每一分深入，都会让整个服务性能成指数的提升。

一、面试题

谢飞机，听说你最近在家很努力学习 HashMap？那考你个 ArrayList 知识点你看下面这段代码输出结果是什么？

```
public static void main(String[] args) {  
    List<String> list = new ArrayList<String>(10);  
    list.add(2, "1");  
    System.out.println(list.get(0));  
}
```

嗯？不知道！👀眼睛看题，看我脸干嘛？好好好，告诉你吧，这样会报错！至于为什么，回家看看书吧。

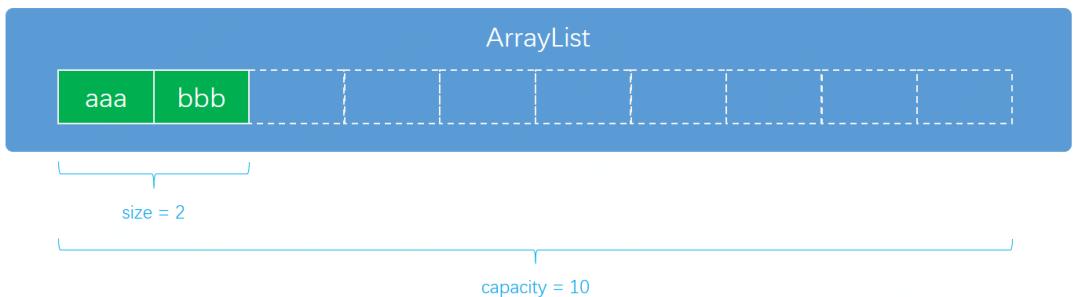
```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 2, Size: 0  
at java.util.ArrayList.rangeCheckForAdd(ArrayList.java:665)  
at java.util.ArrayList.add(ArrayList.java:477)  
at org.itstack.interview.test.ApiTest.main(ApiTest.java:13)
```

```
Process finished with exit code 1
```

谢飞机是懵了，咱们一点点分析 `ArrayList`

二、数据结构

`Array + List = 数组 + 列表 = ArrayList = 数组列表`



`ArrayList` 的数据结构是基于数组实现的，只不过这个数组不像我们普通定义的数组，它可以在 `ArrayList` 的管理下插入数据时按需动态扩容、数据拷贝等操作。接下来，我们就逐步分析 `ArrayList` 的源码，也同时解答谢飞机的疑问。

三、源码分析

1. 初始化

```
List<String> list = new ArrayList<String>(10);

public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}

/**
 * Constructs an empty list with the specified initial capacity.
 *
 * @param initialCapacity the initial capacity of the list
 * @throws IllegalArgumentException if the specified initial capacity
 *         is negative
 */
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    }
}
```

```
    } else {
        throw new IllegalArgumentException("Illegal Capacity: " +
            initialCapacity);
    }
}
```

- 通常情况空构造函数初始化 ArrayList 更常用，这种方式数组的长度会在第一次插入数据时候进行设置。
- 当我们已经知道要填充多少个元素到 ArrayList 中，比如 500 个、1000 个，那么为了提供性能，减少 ArrayList 中的拷贝操作，这个时候会直接初始化一个预先设定好的长度。
- 另外，`EMPTY_ELEMENTDATA` 是一个定义好的空对象；`private static final Object[] EMPTY_ELEMENTDATA = {}`

1.1 方式 01：普通方式

```
ArrayList<String> list = new ArrayList<String>();
list.add("aaa");
list.add("bbb");
list.add("ccc");
```

- 这个方式很简单也是我们最常用的方式。

1.2 方式 02：内部类方式

```
ArrayList<String> list = new ArrayList<String>() {{
    add("aaa");
    add("bbb");
    add("ccc");
}}
```

- 这种方式也是比较常用的，而且省去了多余的代码量。

1.3 方式 03：Arrays.asList

```
ArrayList<String> list = new ArrayList<String>(Arrays.asList("aaa", "bbb", "ccc"))
;
```

以上是通过 `Arrays.asList` 传递给 `ArrayList` 构造函数的方式进行初始化，这里有几个知识点：

1.3.1 ArrayList 构造函数

```
public ArrayList(Collection<? extends E> c) {  
    elementData = c.toArray();  
    if ((size = elementData.length) != 0) {  
        // c.toArray might (incorrectly) not return Object[] (see 6260652)  
        if (elementData.getClass() != Object[].class)  
            elementData = Arrays.copyOf(elementData, size, Object[].class);  
    } else {  
        // replace with empty array.  
        this.elementData = EMPTY_ELEMENTDATA;  
    }  
}
```

- 通过构造函数可以看到，只要实现 `Collection` 类的都可以作为入参。
- 在通过转为数组以及拷贝 `Arrays.copyOf` 到 `Object[]` 集合中在赋值给属性 `elementData`。

注意: `c.toArray might (incorrectly) not return Object[] (see 6260652)`
see 6260652 是 JDK bug 库的编号，有点像商品 sku，bug 地址:https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6260652
那这是个什么 bug 呢，我们来测试下面这段代码；

```
@Test  
public void t(){  
    List<Integer> list1 = Arrays.asList(1, 2, 3);  
    System.out.println("通过数组转换:  
" + (list1.toArray().getClass() == Object[].class));  
  
    ArrayList<Integer> list2 = new ArrayList<Integer>(Arrays.asList(1, 2, 3));  
    System.out.println("通过集合转换:  
" + (list2.toArray().getClass() == Object[].class));  
}
```

测试结果：

通过数组转换: `false`

通过集合转换: `true`

```
Process finished with exit code 0
```

- `public Object[] toArray()` 返回的类型不一定就是 `Object[]`，其类型取决于其返回的实际类型，毕竟 `Object` 是父类，它可以是其他任意类型。
- 子类实现和父类同名的方法，仅仅返回值不一致时，默认调用的是子类的实现方法。

造成这个结果的原因，如下：

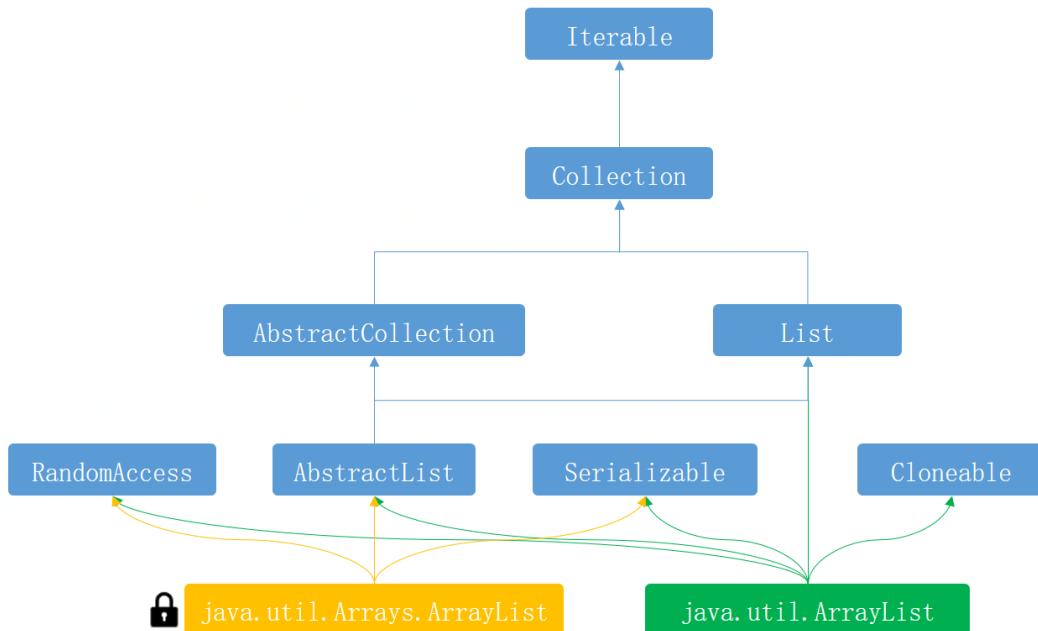
1. `Arrays.asList` 使用的是：`Arrays.copyOf(this.a, size, (Class<? extends T[]>) a.getClass());`
2. `ArrayList` 构造函数使用的是：`Arrays.copyOf(elementData, size, Object[].class);`

1.3.2 `Arrays.asList`

你知道吗？

- `Arrays.asList` 构建的集合，不能赋值给 `ArrayList`
- `Arrays.asList` 构建的集合，不能再添加元素
- `Arrays.asList` 构建的集合，不能再删除元素

那这到底为什么呢，因为 `Arrays.asList` 构建出来的 `List` 与 `new ArrayList` 得到的 `List`，压根就不是一个 `List`！类关系图如下；



小傅哥 bugstack.cn & List 类关系图

从以上的类图关系可以看到；

- 这两个 List 压根不是一个东西，而且 Arrasys 下的 List 是一个私有类，只能通过 `asList` 使用，不能单独创建。
- 另外还有这个 `ArrayList` 不能添加和删除，主要是因为它的实现方式，可以参考 `Arrays` 类中，这部分源码；`private static class ArrayList<E> extends AbstractList<E> implements RandomAccess, java.io.Serializable`

此外，`Arrays` 是一个工具包，里面还有一些非常好用的方法，例如：二分查找 `Arrays.binarySearch`、排序 `Arrays.sort` 等

1.4 方式 04：Collections.nCopies

`Collections.nCopies` 是集合方法中用于生成多少份某个指定元素的方法，接下来就用它来初始化 `ArrayList`，如下：

```
ArrayList<Integer> list = new ArrayList<Integer>(Collections.nCopies(10, 0));
```

- 这会初始化一个由 10 个 0 组成的集合。

2. 插入

`ArrayList` 对元素的插入，其实就是对数组的操作，只不过需要特定时候扩容。

2.1 普通插入

```
List<String> list = new ArrayList<String>();
list.add("aaa");
list.add("bbb");
list.add("ccc");
```

当我们依次插入添加元素时，`ArrayList.add` 方法只是把元素记录到数组的各个位置上了，源码如下：

```
/**
 * Appends the specified element to the end of this list.
 *
 * @param e the element to be appended to this list
 * @return <tt>true</tt> (as specified by {@link Collection#add})
 */
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
```

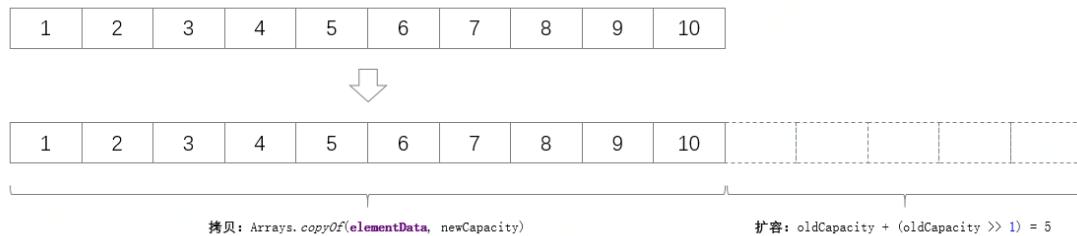
```
    return true;  
}
```

- 这是插入元素时候的源码，`size++`自增，把对应元素添加进去。

2.2 插入时扩容

在前面[初始化](#)部分讲到，ArrayList 默认初始化时会申请 10 个长度的空间，如果超过这个长度则需要进行扩容，那么它是怎么扩容的呢？

从根本上分析来说，数组是定长的，如果超过原来定长长度，扩容则需要申请新的数组长度，并把原数组元素拷贝到新数组中，如下图；



小傅哥 bugstack.cn & 数组扩容

图中介绍了当 List 结合可用空间长度不足时则需要扩容，这主要包括如下步骤：

1. 判断长度充足；`ensureCapacityInternal(size + 1);`
2. 当判断长度不足时，则通过扩大函数，进行扩容；`grow(int minCapacity)`
3. 扩容的长度计算；`int newCapacity = oldCapacity + (oldCapacity >> 1);`，旧容量 + 旧容量右移 1 位，这相当于扩容了原来容量的(`int`) $3/2$ 。4. 10，扩容时： $1010 + 1010 \gg 1 = 1010 + 0101 = 10 + 5 = 15$ 2. 7，扩容时： $0111 + 0111 \gg 1 = 0111 + 0011 = 7 + 3 = 10$
4. 当扩容完以后，就需要进行把数组中的数据拷贝到新数组中，这个过程会用到`Arrays.copyOf(elementData, newCapacity);`，但他的底层用到的是；`System.arraycopy`

`System.arraycopy;`

```
@Test  
public void test_arraycopy() {  
    int[] oldArr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
    int[] newArr = new int[oldArr.length + (oldArr.length >> 1)];  
    System.arraycopy(oldArr, 0, newArr, 0, oldArr.length);
```

```

newArr[11] = 11;
newArr[12] = 12;
newArr[13] = 13;
newArr[14] = 14;

System.out.println("数组元素: " + JSON.toJSONString(newArr));
System.out.println("数组长度: " + newArr.length);

/**
 * 测试结果
 *
 * 数组元素: [1,2,3,4,5,6,7,8,9,10,0,11,12,13,14]
 * 数组长度: 15
 */
}

}

```

- 拷贝数组的过程并不复杂，主要是对 `System.arraycopy` 的操作。
- 上面就是把数组 `oldArr` 拷贝到 `newArr`，同时新数组的长度，采用和 `ArrayList` 一样的计算逻辑；`oldArr.length + (oldArr.length >> 1)`

2.3 指定位置插入

```
list.add(2, "1");
```

到这，终于可以说说谢飞机的面试题，这段代码输出结果是什么，如下：

```

Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 2, Size: 0
at java.util.ArrayList.rangeCheckForAdd(ArrayList.java:665)
at java.util.ArrayList.add(ArrayList.java:477)
at org.itstack.interview.test.ApiTest.main(ApiTest.java:14)

```

其实，一段报错提示，为什么呢？我们翻开下源码学习下。

2.3.1 容量验证

```

public void add(int index, E element) {
    rangeCheckForAdd(index);

    ...
}

```

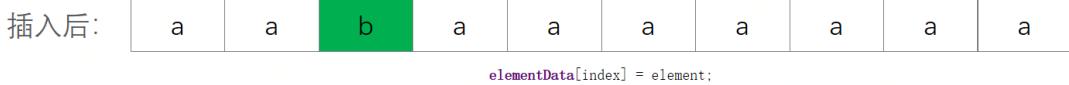
```

private void rangeCheckForAdd(int index) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}

```

- 指定位置插入首先要判断 `rangeCheckForAdd`, `size` 的长度。
- 通过上面的元素插入我们知道, 每插入一个元素, `size` 自增一次 `size++`。
- 所以即使我们申请了 10 个容量长度的 `ArrayList`, 但是指定位置插入会依赖于 `size` 进行判断, 所以会抛出 `IndexOutOfBoundsException` 异常。

2.3.2 元素迁移



小傅哥 bugstack.cn & 插入元素迁移

指定位置插入的核心步骤包括;

- 判断 `size`, 是否可以插入。
- 判断插入后是否需要扩容; `ensureCapacityInternal(size + 1);`。
- 数据元素迁移, 把从待插入位置后的元素, 顺序往后迁移。
- 给数组的指定位置赋值, 也就是把待插入元素插入进来。

部分源码:

```

public void add(int index, E element) {
    ...
    // 判断是否需要扩容以及扩容操作
    ensureCapacityInternal(size + 1);
    // 数据拷贝迁移, 把待插入位置空出来
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
}

```

```

    // 数据插入操作
    elementData[index] = element;
    size++;
}

```

- 这部分源码的主要核心是在，`System.arraycopy`，上面我们已经演示过相应的操作方式。
- 这里只是设定了指定位置的迁移，可以把上面的案例代码复制下来做测试验证。

实践：

```

List<String> list = new ArrayList<String>(Collections.nCopies(9, "a"));
System.out.println("初始化: " + list);

list.add(2, "b");
System.out.println("插入后: " + list);

```

测试结果：

```

初始化: [a, a, a, a, a, a, a, a, a]
插入后: [a, a, 1, a, a, a, a, a, a]

```

```
Process finished with exit code 0
```

- 指定位置已经插入元素 1，后面的数据向后迁移完成。

3. 删除

有了指定位置插入元素的经验，理解删除的过长就比较容易了，如下图；



`System.arraycopy(elementData, index+1, elementData, index, numMoved);`



`elementData[--size] = null`

小傅哥 bugstack.cn & 删除元素

这里我们结合着代码：

```
public E remove(int index) {  
    rangeCheck(index);  
    modCount++;  
    E oldValue = elementData(index);  
    int numMoved = size - index - 1;  
    if (numMoved > 0)  
        System.arraycopy(elementData, index+1, elementData, index,  
                         numMoved);  
    elementData[--size] = null; // clear to let GC do its work  
    return oldValue;  
}
```

删除的过程主要包括：

1. 校验是否越界；`rangeCheck(index);`
2. 计算删除元素的移动长度 `numMoved`，并通过 `System.arraycopy` 自己把元素复制给自己。
3. 把结尾元素清空，`null`。

这里我们做个例子：

```
@Test  
public void test_copy_remove() {  
    int[] oldArr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
    int index = 2;  
    int numMoved = 10 - index - 1;  
    System.arraycopy(oldArr, index + 1, oldArr, index, numMoved);  
    System.out.println("数组元素: " + JSON.toJSONString(oldArr));  
}
```

- 设定一个拥有 10 个元素的数组，同样按照 `ArrayList` 的规则进行移动元素。
- 注意，为了方便观察结果，这里没有把结尾元素设置为 `null`。

测试结果：

数组元素：[1, 2, 4, 5, 6, 7, 8, 9, 10]

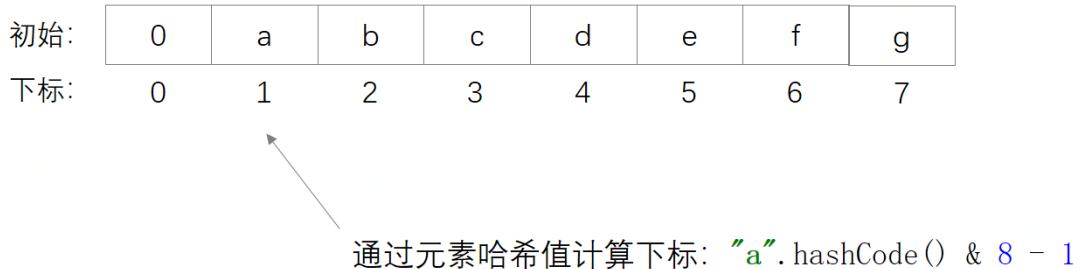
Process finished with exit code 0

- 可以看到指定位置 `index = 2`, 元素已经被删掉。
- 同时数组已经移动用元素 4 占据了原来元素 3 的位置, 同时结尾的 10 还等待删除。这就是为什么 `ArrayList` 中有这么一句代码: `elementData[--size] = null`

4. 扩展

如果给你一组元素: `a、b、c、d、e、f、g`, 需要你放到 `ArrayList` 中, 但是要求获取一个元素的时间复杂度都是 $O(1)$, 你怎么处理?

想解决这个问题, 就需要知道元素添加到集合中后知道它的位置, 而这个位置呢, 其实可以通过哈希值与集合长度与运算, 得出存放数据的下标, 如下图;



小傅哥 bugstack.cn & 下标计算

- 如图就是计算出每一个元素应该存放的位置, 这样就可以 $O(1)$ 复杂度获取元素。

4.1 代码操作(添加元素)

```
List<String> list = new ArrayList<String>(Collections.nCopies(8, "0"));

list.set("a".hashCode() & 8 - 1, "a");
list.set("b".hashCode() & 8 - 1, "b");
list.set("c".hashCode() & 8 - 1, "c");
list.set("d".hashCode() & 8 - 1, "d");
list.set("e".hashCode() & 8 - 1, "e");
list.set("f".hashCode() & 8 - 1, "f");
list.set("g".hashCode() & 8 - 1, "g");
```

- 以上是初始化 `ArrayList`, 并存放相应的元素。存放时候计算出每个元素的下标值。

4.2 代码操作(获取元素)

```
System.out.println("元素集合: " + list);
System.out.println("获取元素 f [\"f\".hashCode() & 8 - 1] Idx:
" + ("f".hashCode() & (8 - 1)) + " 元素: " + list.get("f".hashCode() & 8 - 1));
System.out.println("获取元素 e [\"e\".hashCode() & 8 - 1] Idx:
" + ("e".hashCode() & (8 - 1)) + " 元素: " + list.get("e".hashCode() & 8 - 1));
System.out.println("获取元素 d [\"d\".hashCode() & 8 - 1] Idx:
" + ("d".hashCode() & (8 - 1)) + " 元素: " + list.get("d".hashCode() & 8 - 1));
```

4.3 测试结果

元素集合:[0, a, b, c, d, e, f, g]

```
获取元素 f ["f".hashCode() & 8 - 1] Idx: 6 元素: f
获取元素 e ["e".hashCode() & 8 - 1] Idx: 5 元素: e
获取元素 d ["d".hashCode() & 8 - 1] Idx: 4 元素: d
```

Process finished with exit code 0

- 通过测试结果可以看到，下标位置 0 是初始元素，元素是按照指定的下标进行插入的。
- 那么现在获取元素的时间复杂度就是 $O(1)$ ，是不有点像 `HashMap` 中的桶结构。

四、总结

- 就像我们开头说的一样，数据结构是你写出代码的基础，更是写出高级代码的核心。只有了解好数据结构，才能更透彻的理解程序设计。并不是所有的逻辑都是 for 循环
- 面试题只是引导你学习的点，但不能为了面试题而忽略更重要的核心知识学习，背一两道题是不可能抗拒深度问的。因为任何一个考点，都不只是一种问法，往往可以从很多方面进行提问和考查。就像你看完整篇文章，是否理解了没有说到的知识，当你固定位置插入数据时会进行数据迁移，那么在拥有大量数据的 `ArrayList` 中是不适合这么做的，非常影响性能。
- 在本章的内容编写的时候也参考到一些优秀的资料，尤其发现这份外文文档：
<https://beginnersbook.com/> 大家可以参考学习。

第 7 节：LinkedList、ArrayList，插入分析

你以为考你个数据结构是要造火箭？

汽车 75 马力就够奔跑了，那你怎么还想要 2.0 涡轮+9AT 呢？大桥两边的护栏你每次走的时候都会去摸吗？那怎么没有护栏的大桥你不敢上呢？

很多时候，你额外的能力才是自身价值的体现，不要以为你的能力就只是做个业务开发每天 CRUD，并不是产品让你写 CRUD，而是因为你的能力只能产品功能设计成 CRUD。

就像数据结构、算法逻辑、源码技能，它都是可以为你的业务开发赋能的，也是写出更好、更易扩展程序的根基，所以学好这份知识非常有必要。

本文涉及了较多的代码和实践验证图稿，欢迎关注公众号：[bugstack 虫洞栈](#)，回复下载得到一个链接打开后，找到 ID: 19 获取！

一、面试题

谢飞机，ArrayList 资料看了吧？嗯，那行问问你哈

问：ArrayList 和 LinkedList，都用在什么场景呢？

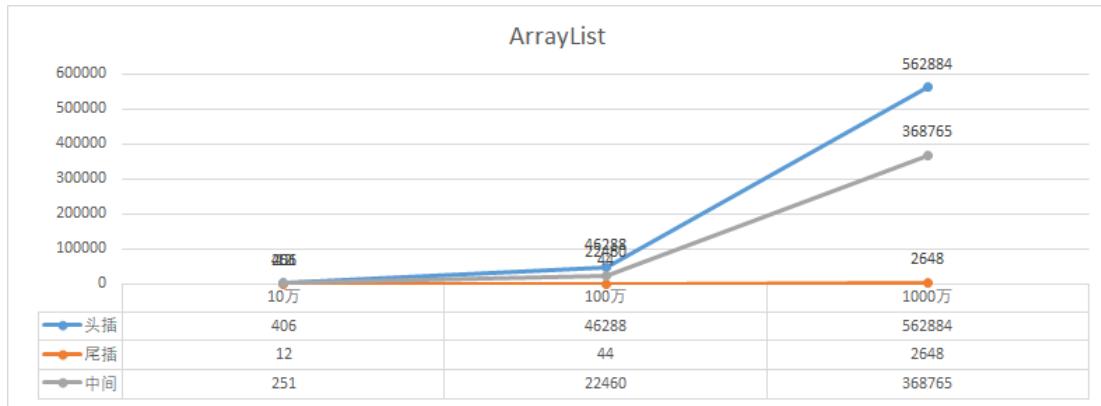
答：啊，这我知道了。ArrayList 是基于数组实现、LinkedList 是基于双向链表实现，所以基于数据结构的不同，遍历和查找多的情况下用 ArrayList、插入和删除频繁的情况下用 LinkedList。

问：嗯，那 LinkedList 的插入效率一定比 ArrayList 好吗？

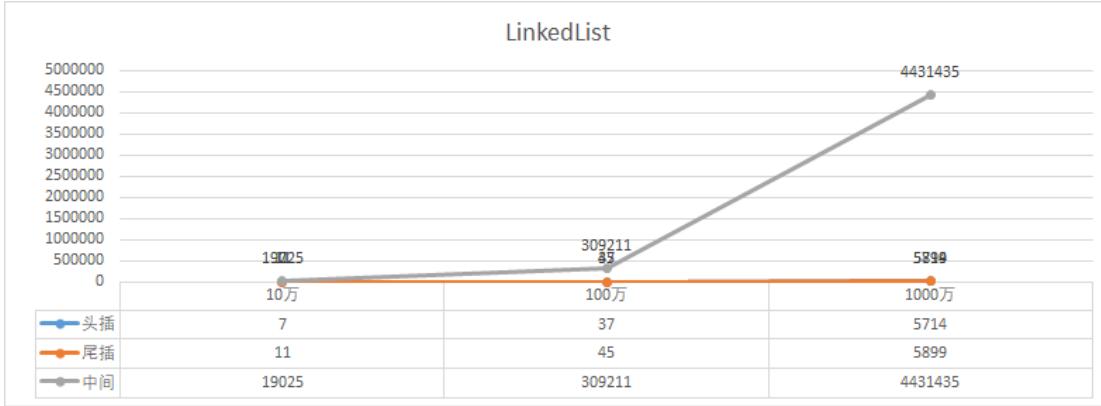
答：对，好！

送你个飞机✈，回去等消息吧！

其实，飞机回答的也不是不对，只是不全面。出门后不甘心买瓶[肥宅水](#)又回来，跟面试官聊了 2 个点，要到了两张图，如下：



小傅哥 bugstack.cn & ArrayList 头插、尾插、中间



小傅哥 bugstack.cn & LinkedList 头插、尾插、中间

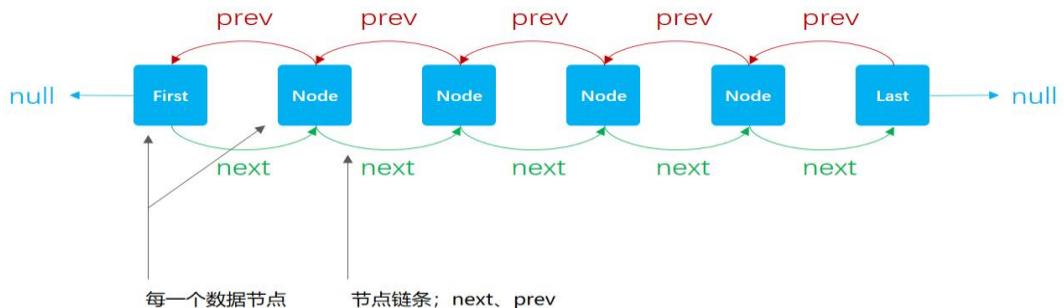
如图，分别是；10 万、100 万、1000 万，数据在两种集合下不同位置的插入效果，

所以：，不能说 LinkedList 插入就快，ArrayList 插入就慢，还需要看具体的操作情况。

接下来我们带着数据结构和源码，具体分析下。

二、数据结构

`Linked + List = 链表 + 列表 = LinkedList = 链表列表`



小傅哥 bugstack.cn & LinkedList 数据结构

LinkedList，是基于链表实现，由双向链条 next、prev，把数据节点穿插起来。所以，在插入数据时，是不需要像我们上一章节介绍的 ArrayList 那样，扩容数组。

但，又不能说所有的插入都是高效，比如中间区域插入，他还需要遍历元素找到插入位置。具体的细节，我们在下文的源码分析中进行讲解，也帮谢飞机扫除疑惑。

三、源码分析

1. 初始化

与 ArrayList 不同，LinkedList 初始化不需要创建数组，因为它是一个链表结构。而且也没有传给构造函数初始化多少个空间的入参，例如这样是不可以的，如下：

```
LinkedList<String> list = new LinkedList<String>(c: 10);
```

这憨批不能留了。。



不能这么干

但是，构造函数一样提供了和 ArrayList 一些相同的方式，来初始化入参，如下这四种方式：

```
@Test  
public void test_init() {  
    // 初始化方式：普通方式  
    LinkedList<String> list01 = new LinkedList<String>();  
    list01.add("a");  
    list01.add("b");  
    list01.add("c");  
    System.out.println(list01);  
  
    // 初始化方式：Arrays.asList  
    LinkedList<String> list02 = new LinkedList<String>(Arrays.asList("a", "b", "c"));
```

```
System.out.println(list02);

// 初始化方式: 内部类
LinkedList<String> list03 = new LinkedList<String>()\\{
    {add("a");add("b");add("c");}
}};

System.out.println(list03);

// 初始化方式: Collections.nCopies
LinkedList<Integer> list04 = new LinkedList<Integer>(Collections.nCopies(10, 0)
);
System.out.println(list04);
}

// 测试结果

[a, b, c]
[a, b, c]
[a, b, c]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Process finished with exit code 0

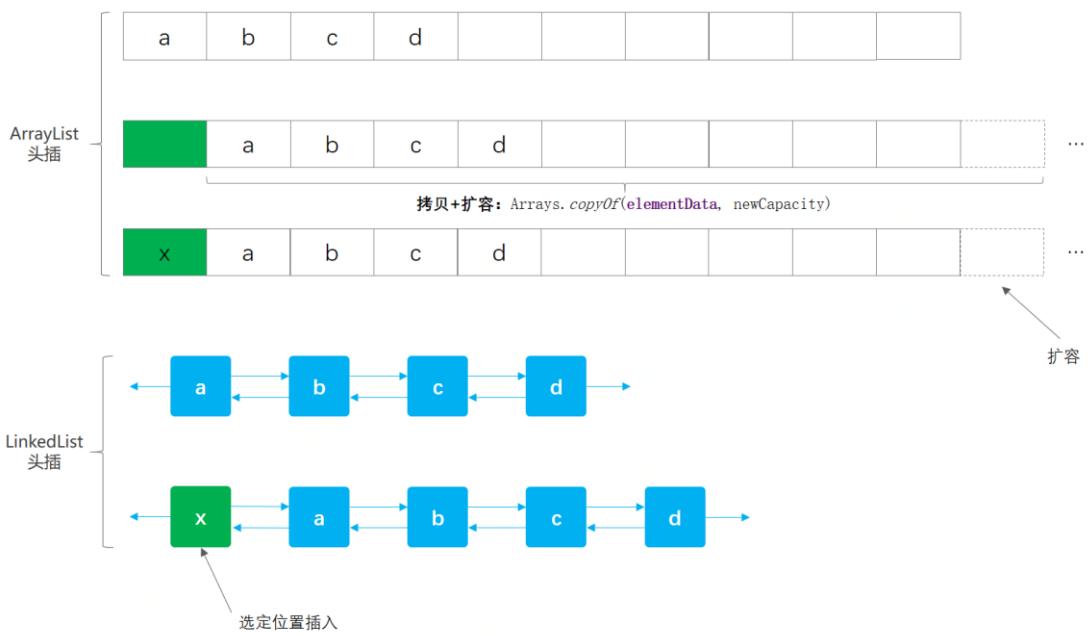
- 这些方式都可以初始化操作，按需选择即可。

2. 插入

LinkedList 的插入方法比较多，List 中接口中默认提供的是 add，也可以指定位置插入。但在 LinkedList 中还提供了头插 `addFirst` 和尾插 `addLast`。
关于插入这部分就会讲到为什么；有的时候 LinkedList 插入更耗时、有的时候 ArrayList 插入更好。

2.1 头插

先来看一张数据结构对比图，回顾下 ArrayList 的插入也和 LinkedList 插入做下对比，如下：



小傅哥 bugstack.cn & 插入对比

看上图我们可以分析出几点；

1. ArrayList 头插时，需要把数组元素通过 `Arrays.copyOf` 的方式把数组元素移位，如果容量不足还需要扩容。
2. LinkedList 头插时，则不需要考虑扩容以及移位问题，直接把元素定位到首位，接点链表链接上即可。

2.1.1 源码

这里我们再对照下 `LinkedList` 头插的源码，如下；

```
private void linkFirst(E e) {
    final Node<E> f = first;
    final Node<E> newNode = new Node<>(null, e, f);
    first = newNode;
    if (f == null)
        last = newNode;
    else
        f.prev = newNode;
    size++;
    modCount++;
}
```

- `first`, 首节点会一直被记录，这样就非常方便头插。

- 插入时候会创建新的节点元素, `new Node<>(null, e, f)`, 紧接着把新的头元素赋值给 `first`。
- 之后判断 `f` 节点是否存在, 不存在则把头插节点作为最后一个节点、存在则用 `f` 节点的上一个链条 `prev` 链接。
- 最后记录 `size` 大小、和元素数量 `modCount`。*modCount 用在遍历时做校验, modCount != expectedModCount*

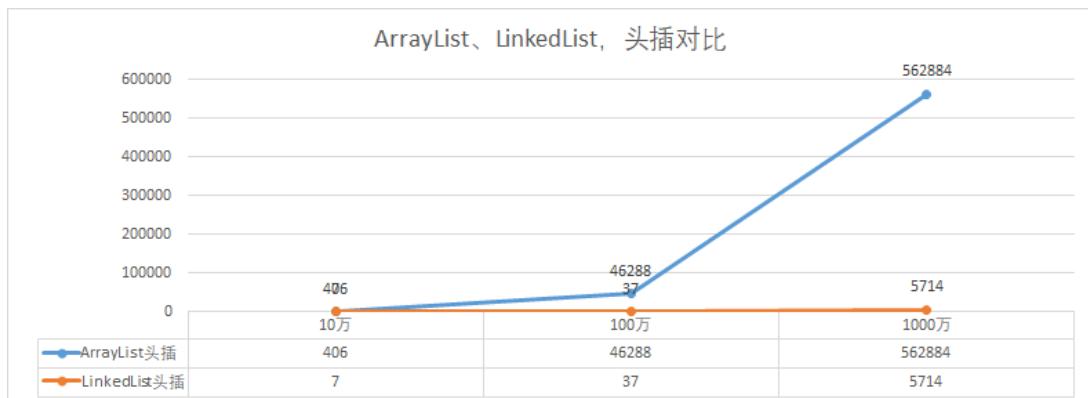
2.1.2 验证

ArrayList、LinkedList, 头插源码验证

```
@Test
public void test_ArrayList_addFirst() {
    ArrayList<Integer> list = new ArrayList<Integer>();
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < 10000000; i++) {
        list.add(0, i);
    }
    System.out.println("耗时: " + (System.currentTimeMillis() - startTime));
}
```

```
@Test
public void test_LinkedList_addFirst() {
    LinkedList<Integer> list = new LinkedList<Integer>();
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < 10000000; i++) {
        list.addFirst(i);
    }
    System.out.println("耗时: " + (System.currentTimeMillis() - startTime));
}
```

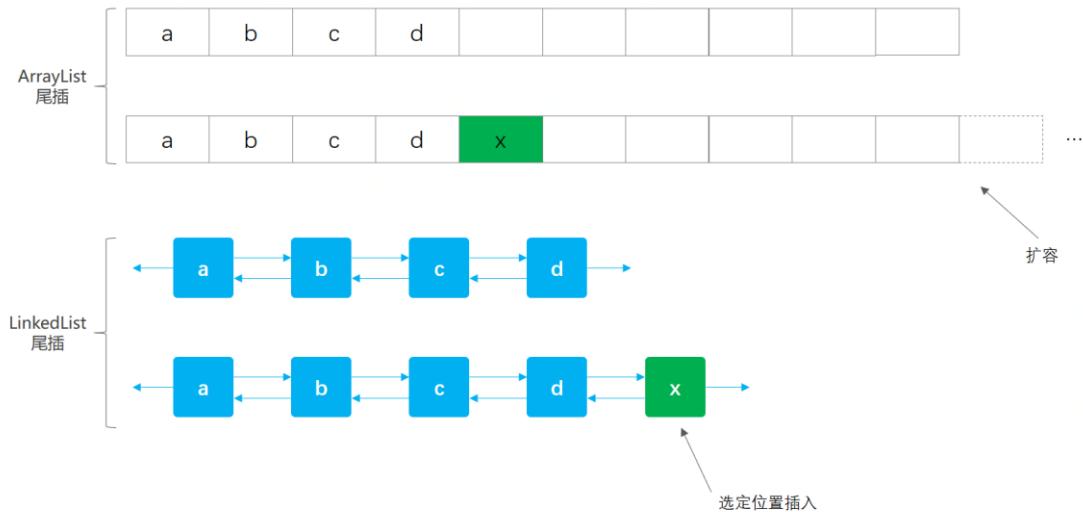
比对结果:



- 这里我们分别验证，10万、100万、1000万的数据量，在头插时的一个耗时情况。
- 如我们数据结构对比图中一样，ArrayList 需要做大量的位移和复制操作，而 LinkedList 的优势就体现出来了，耗时只是实例化一个对象。

2.2 尾插

先来看一张数据结构对比图，回顾下 ArrayList 的插入也和 LinkedList 插入做下对比，如下：



小傅哥 bugstack.cn & 插入对比

看上图我们可以分析出几点；

1. ArrayList 尾插时，是不需要数据位移的，比较耗时的是数据的扩容时，需要拷贝迁移。
2. LinkedList 尾插时，与头插相比耗时点会在对象的实例化上。

2.2.1 源码

这里我们再对照下 **LinkedList** 尾插的源码，如下；

```
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
```

```

        l.next = newNode;
        size++;
        modCount++;
    }
}

```

- 与头插代码相比几乎没有什么区别，只是 first 换成 last
- 耗时点只是在创建节点上，`Node<E>`

2.2.2 验证

ArrayList、LinkedList，尾插源码验证

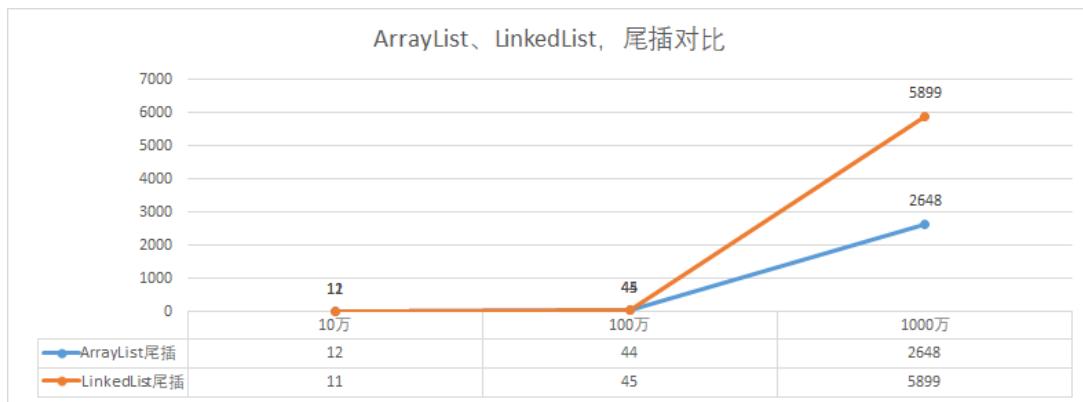
```

@Test
public void test_ArrayList_addLast() {
    ArrayList<Integer> list = new ArrayList<Integer>();
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < 10000000; i++) {
        list.add(i);
    }
    System.out.println("耗时: " + (System.currentTimeMillis() - startTime));
}

@Test
public void test_LinkedList_addLast() {
    LinkedList<Integer> list = new LinkedList<Integer>();
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < 1000000; i++) {
        list.addLast(i);
    }
    System.out.println("耗时: " + (System.currentTimeMillis() - startTime));
}

```

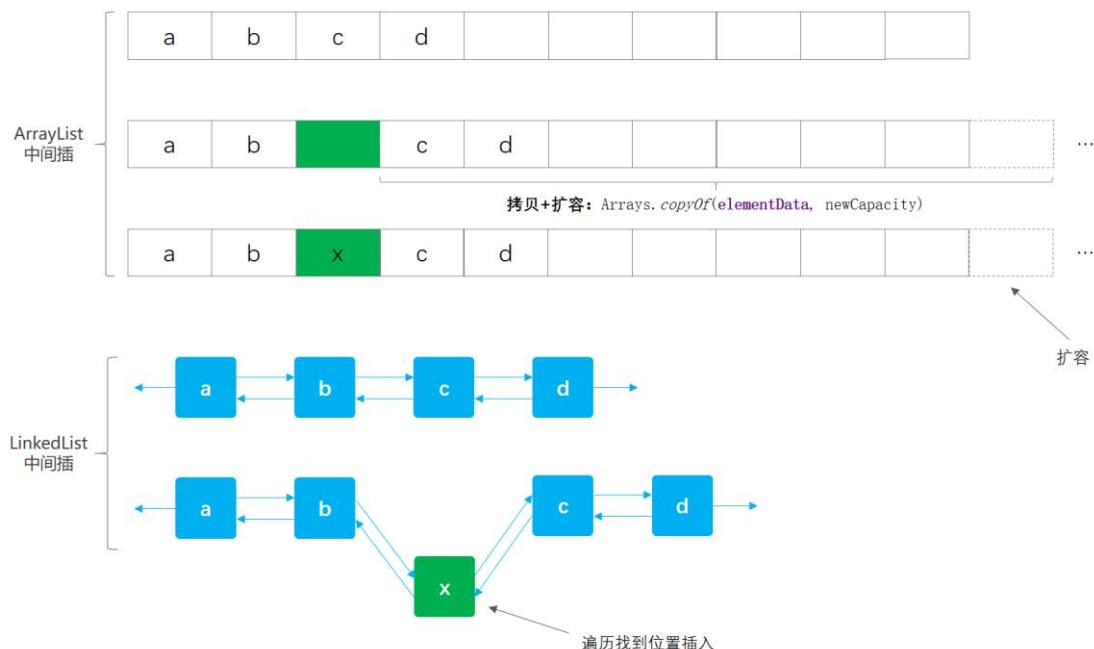
比对结果：



- 这里我们分别验证，10万、100万、1000万的数据量，在尾插时的一个耗时情况。
- 如我们数据结构对比图中一样，ArrayList 不需要做位移拷贝也就不那么耗时了，而 LinkedList 则需要创建大量的对象。所以这里 ArrayList 尾插的效果更好一些。

2.3 中间插

先来看一张数据结构对比图，回顾下 ArrayList 的插入也和 LinkedList 插入做下对比，如下：



看上图我们可以分析出几点；

- ArrayList 中间插入，首先我们知道他的定位时间复杂度是 $O(1)$ ，比较耗时的点在于数据迁移和容量不足的时候扩容。

2. LinkedList 中间插入，链表的数据实际插入时候并不会怎么耗时，但是它定位的元素的时间复杂度是 $O(n)$ ，所以这部分以及元素的实例化比较耗时。

2.3.1 源码

这里看下 LinkedList 指定位置插入的源码；

使用 add(位置、元素)方法插入：

```
public void add(int index, E element) {  
    checkPositionIndex(index);  
    if (index == size)  
        linkLast(element);  
    else  
        linkBefore(element, node(index));  
}
```

位置定位 node(index)：

```
Node<E> node(int index) {  
    // assert isElementIndex(index);  
    if (index < (size >> 1)) {  
        Node<E> x = first;  
        for (int i = 0; i < index; i++)  
            x = x.next;  
        return x;  
    } else {  
        Node<E> x = last;  
        for (int i = size - 1; i > index; i--)  
            x = x.prev;  
        return x;  
    }  
}
```

- `size >> 1`，这部分的代码判断元素位置在左半区间，还是右半区间，在进行循环查找。

执行插入：

```
void linkBefore(E e, Node<E> succ) {  
    // assert succ != null;
```

```

final Node<E> pred = succ.prev;
final Node<E> newNode = new Node<>(pred, e, succ);
succ.prev = newNode;
if (pred == null)
    first = newNode;
else
    pred.next = newNode;
size++;
modCount++;
}

```

- 找到指定位置插入的过程就比较简单了，与头插、尾插，相差不大。
- 整个过程可以看到，插入中比较耗时的点会在遍历寻找插入位置上。

2.3.2 验证

ArrayList、LinkedList，中间插入源码验证

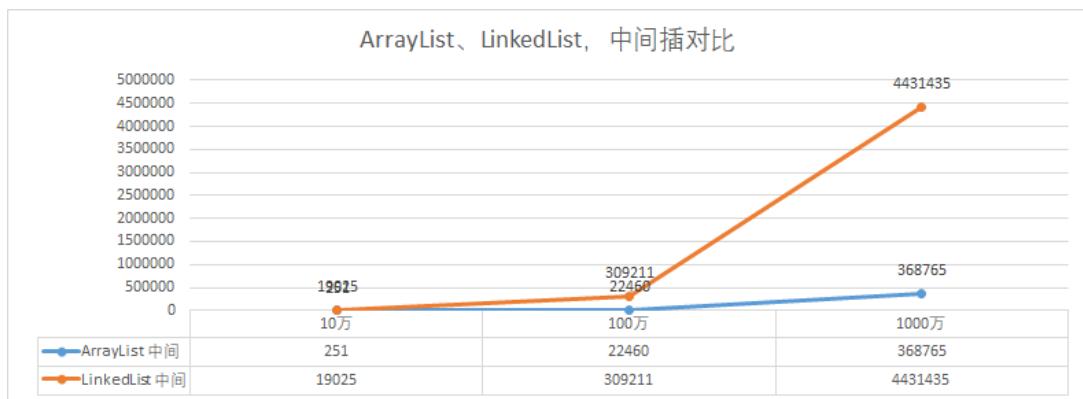
```

@Test
public void test_ArrayList_addCenter() {
    ArrayList<Integer> list = new ArrayList<Integer>();
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < 10000000; i++) {
        list.add(list.size() >> 1, i);
    }
    System.out.println("耗时: " + (System.currentTimeMillis() - startTime));
}

@Test
public void test_Linkedlist_addCenter() {
    LinkedList<Integer> list = new LinkedList<Integer>();
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < 10000000; i++) {
        list.add(list.size() >> 1, i);
    }
    System.out.println("耗时: " + (System.currentTimeMillis() - startTime));
}

```

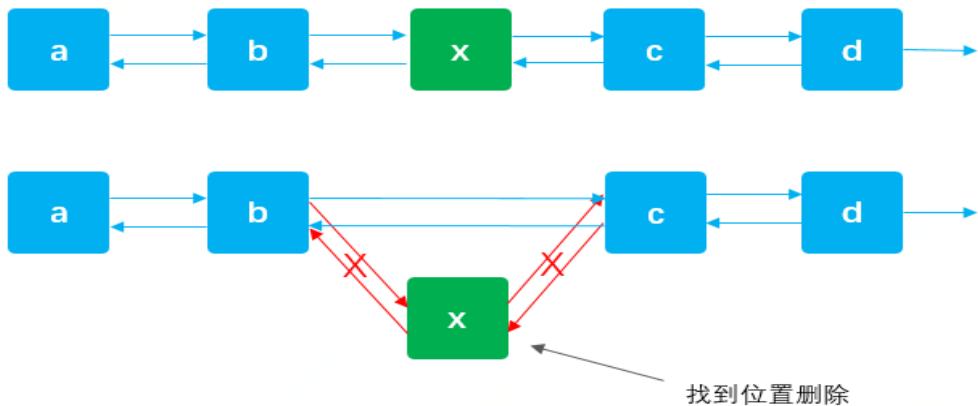
比对结果：



- 这里我们分别验证，10万、100万、1000万的数据量，在中间插时的一个耗时情况。
- 可以看到 LinkedList 在中间插入时，遍历寻找位置还是非常耗时了。所以不同的情况下，需要选择不同的 List 集合做业务。

3. 删 除

讲了这么多插入的操作后，删除的知识点就很好理解了。与 ArrayList 不同，删除不需要拷贝元素，LinkedList 是找到元素位置，把元素前后链连接上。基本如下图：



- 确定出要删除的元素 x，把前后的链接进行替换。
- 如果是删除首尾元素，操作起来会更加容易，这也就是为什么说插入和删除快。但中间位置删除，需要遍历找到对应位置。

3.1 删 除操 作方 法

序号	方法	描述
1	list.remove();	与 removeFirst()一致
2	list.remove(1);	删除 idx=1 的位置元素节点，需要遍历定位
3	list.remove("a");	删除元素="a"的节点，需要遍历定位
4	list.removeFirst();	删除首位节点
5	list.removeLast();	删除结尾节点
6	list.removeAll(Arrays.asList("a", "b"));	按照集合批量删除，底层是 Iterator 删除

源码：

```

@Test
public void test_remove() {
    LinkedList<String> list = new LinkedList<String>();
    list.add("a");
    list.add("b");
    list.add("c");

    list.remove();
    list.remove(1);
    list.remove("a");
    list.removeFirst();
    list.removeLast();
    list.removeAll(Arrays.asList("a", "b"));
}

```

3.2 源码

删除操作的源码都差不多，分为删除首尾节点与其他节点时候，对节点的解链操作。这里我们举例一个删除其他位置的源码进行学习，如下；

```

list.remove("a");

public boolean remove(Object o) {
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {

```

```

        if (x.item == null) {
            unlink(x);
            return true;
        }
    }
} else {
    for (Node<E> x = first; x != null; x = x.next) {
        if (o.equals(x.item)) {
            unlink(x);
            return true;
        }
    }
}
return false;
}

```

- 这一部分是元素定位，和 `unlink(x)` 解链。循环查找对应的元素，这部分没有什么难点。

`unlink(x)` 解链

```

E unlink(Node<E> x) {
    // assert x != null;
    final E element = x.item;
    final Node<E> next = x.next;
    final Node<E> prev = x.prev;

    if (prev == null) {
        first = next;
    } else {
        prev.next = next;
        x.prev = null;
    }

    if (next == null) {
        last = prev;
    } else {
        next.prev = prev;
        x.next = null;
    }
}

```

```
x.item = null;  
size--;  
modCount++;  
return element;  
}
```

这部分源码主要有以下几个知识点：

1. 获取待删除节点的信息；元素 item、元素下一个节点 next、元素上一个节点 prev。
2. 如果上个节点为空则把待删除元素的下一个节点赋值给首节点，否则把待删除节点的下一个节点，赋值给待删除节点的上一个节点的子节点。
3. 同样待删除节点的下一个节点 next，也执行 2 步骤同样操作。
4. 最后是把删除节点设置为 null，并扣减 size 和 modeCount 数量。

4. 遍历

接下来谈谈遍历，ArrayList 与 LinkedList 的遍历都是通用的，基本包括 5 种方式。

这里我们先初始化出待遍历的集合 1 千万数据；

```
int xx = 0;  
  
@Before  
  
public void init() {  
    for (int i = 0; i < 10000000; i++) {  
        list.add(i);  
    }  
}
```

4.1 普通 for 循环

```
@Test  
  
public void test_LinkedList_for0() {  
    long startTime = System.currentTimeMillis();  
    for (int i = 0; i < list.size(); i++) {  
        xx += list.get(i);  
    }  
    System.out.println("耗时：" + (System.currentTimeMillis() - startTime));  
}
```

4.2 增强 for 循环

```
@Test
public void test_LinkedList_for1() {
    long startTime = System.currentTimeMillis();
    for (Integer itr : list) {
        xx += itr;
    }
    System.out.println("耗时: " + (System.currentTimeMillis() - startTime));
}
```

4.3 Iterator 遍历

```
@Test
public void test_LinkedList_Iterator() {
    long startTime = System.currentTimeMillis();
    Iterator<Integer> iterator = list.iterator();
    while (iterator.hasNext()) {
        Integer next = iterator.next();
        xx += next;
    }
    System.out.println("耗时: " + (System.currentTimeMillis() - startTime))
}
```

4.4 forEach 循环

```
@Test
public void test_LinkedList_forEach() {
    long startTime = System.currentTimeMillis();
    list.forEach(integer -> {
        xx += integer;
    });
    System.out.println("耗时: " + (System.currentTimeMillis() - startTime));
}
```

4.5 stream(流)

```
@Test
public void test_LinkedList_stream() {
    long startTime = System.currentTimeMillis();
```

```
list.stream().forEach(integer -> {
    xx += integer;
});
System.out.println("耗时: " + (System.currentTimeMillis() - startTime));
}
```

那么，以上这 5 种遍历方式谁最慢呢？按照我们的源码学习分析下吧，欢迎留下你的答案在评论区！

四、总结

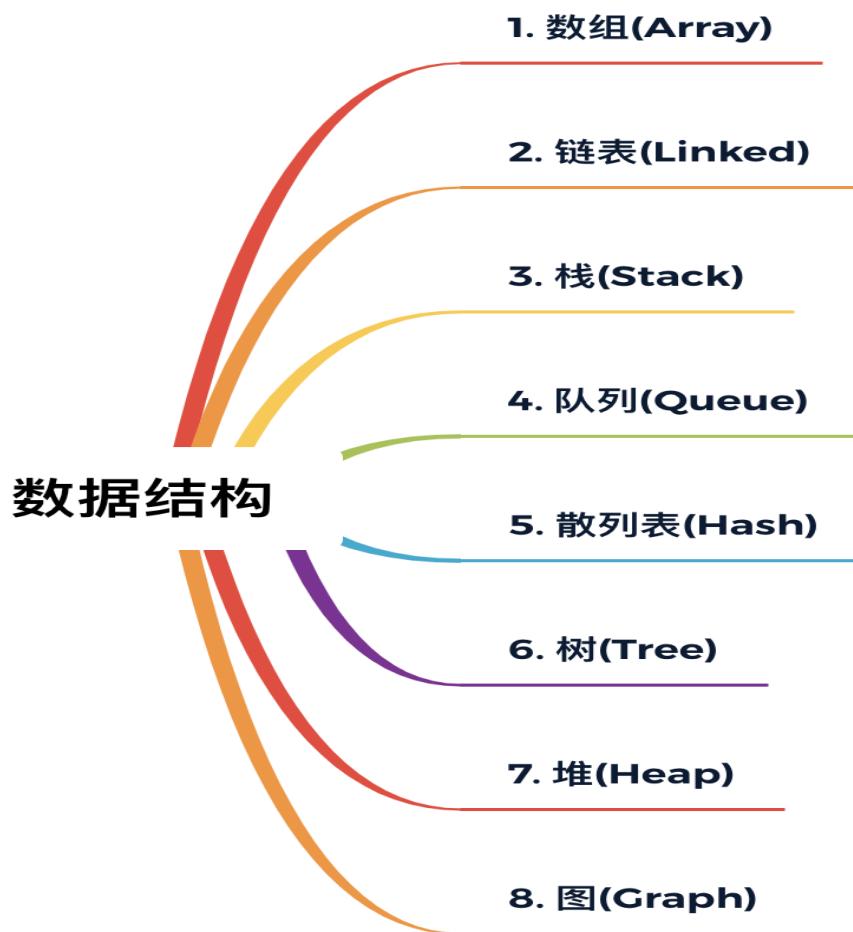
- `ArrayList` 与 `LinkedList` 都有自己的使用场景，如果你不能很好的确定，那么就使用 `ArrayList`。但如果你能确定你会在集合的首位有大量的插入、删除以及获取操作，那么可以使用 `LinkedList`，因为它都有相应的方法：`addFirst`、`addLast`、`removeFirst`、`removeLast`、`getFirst`、`getLast`，这些操作的时间复杂度都是 $O(1)$ ，非常高效。
- `LinkedList` 的链表结构不一定会比 `ArrayList` 节省空间，首先它所占用的内存不是连续的，其次他还需要大量的实例化对象创造节点。虽然不一定节省空间，但链表结构也是非常优秀的数据结构，它能在你的程序设计中起着非常优秀的作用，例如可视化的链路追踪图，就是需要链表结构，并需要每个节点自旋一次，用于串联业务。
- 程序的精髓往往就是数据结构的设计，这能为你的程序开发提供出非常高的效率改变。可能目前你还不能用到，但万一有一天你需要去造  火箭了呢？

第 8 节：双端队列、延迟队列、阻塞队列

买房子最重要的是房屋格局！

如果买房子能接受地理位置、平米价格外，最重要的就是房屋格局。什么？丈母娘！你 ，出去！房屋的格局其实对应的就是程序开发的根本，也就是数据结构。有的土豪可以用钱换空间，房间格局更大，那没钱的就只能选经济小空间节省钱。是不是很像不同的数据结构，直接影响着是空间换时间，还是时间换空间。那么，再细看房间，如：客厅沙发坐人像散列表、去厨房 叫进栈「LIFO」、上厕所  叫入队列「FIFO」、晚上各回各屋子像进数组。所以你能在这个屋子生活的舒服，很大一部分取决于整个房间的布局。也同样你能把程序写好，很大的原因是数据结构定义的合理。

那么决定这程序开发基础数据结构有哪些呢？



程序开发中数据结构可以分为这八类：数组、链表、栈、队列、散列表、树、堆、图。其中，数组、链表、散列表、树是程序开发直接或者间接用到的最多的。相关的对应实现类可以包括如下：

类型	实现	文章
数组	ArrayList	ArrayList 详细分析
链表	LinkedList	LinkedList、ArrayList 插入速度对比
树	2-3 树、红黑树	看图说话，讲解 2-3 平衡树「红黑树的前身」 红黑树操作原理，解析什么时候染色、怎么进行旋转、与 2-3 树有什么关联
散列表	HashMap	HashMap 核心知识，扰动函数、负载因子、扩容链表拆分，深度学习 HashMap 数据插入、查找、删除、遍历，源码分析
栈	Stack	
队列	Queue 、 Deque	本章

- 如上，除了栈和队列外，小傅哥已经编写了非常细致的文章来介绍了其他数据结构的核心知识和具体的实现应用。
- 接下来就把剩下的栈和队列在本章介绍完，其实这部分知识并不难了，有了以上对数组和链表的理解，其他的数据结构基本都从这两方面扩展出来的。

本文涉及了较多的代码和实践验证图稿，欢迎关注公众号：[bugstack 虫洞栈](#)，回复下载得到一个链接打开后，找到 ID: 19 获取！

一、面试题

问：谢飞机，飞机你旁边这是？

答：啊，谢坦克，我弟弟。还没毕业，想来看看大公司面试官的容颜。

问：飞机，上次把 [LinkedList](#) 都看了吧，那我问你哈。[LinkedList](#) 可以当队列用吗？

答：啊？可以，可以吧！

问：那，数组能当队列用吗？不能？对列有啥特点吗？

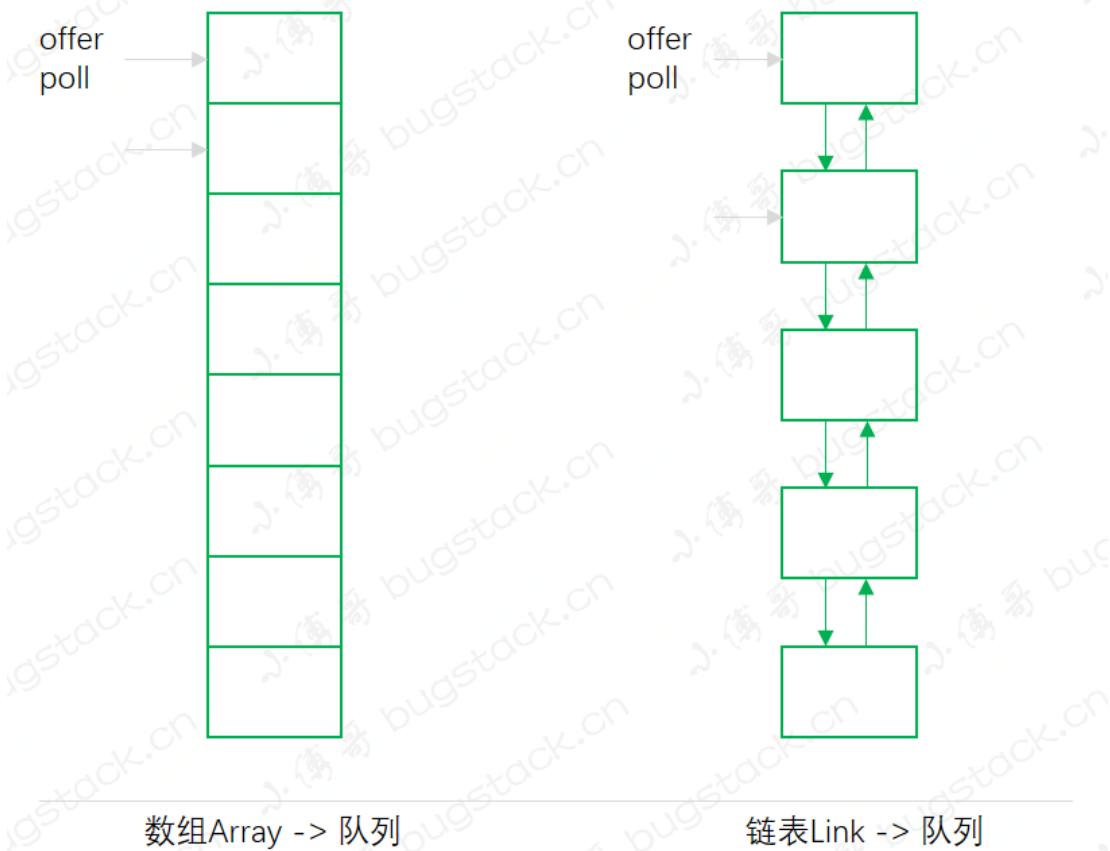
答：队列先进先出，嗯，嗯。

问：还有吗？了解延时队列吗？双端队列呢？

飞机拉着坦克的手出门了，还带走了面试官送的一本《面经手册》，坦克对飞机说，基础不牢，地动山摇，我要好好学习。

二、数据结构

把我们已经掌握了的数组和链表立起来，就是栈和队列了！



如图，这一章节的数据结构的知识点并不难，只要已经学习过数组和链表，那么对于掌握其他数据结构就已经有了基础，只不过对于数据的存放、读取加了一些限定规则。尤其像链表这样的数据结构，只操作头尾的效率是非常高的。

三、源码学习

1. 先说一个被抛弃 Stack

有时候不会反而不会犯错误！怕就怕在只知道一知半解。

抛弃的不是栈这种数据结构，而是 Stack 实现类，如果你还不了解就用到业务开发中，就很可能会影响系统性能。其实 Stack 这个栈已经是不建议使用了，但是为什么不建议使用，我们可以通过使用和源码分析了解下根本原因。

在学习之前先大概的了解下这样的数据结构，它很像羽毛球的摆放，是一种后进先出队列，如下：



1.1 功能使用

```
@Test
public void test_stack() {
    Stack<String> s = new Stack<String>();
    s.push("aaa");
    s.push("bbb");
    s.push("ccc");

    System.out.println("获取最后一个元素: " + s.peek());
    System.out.println("获取最后一个元素: " + s.lastElement());
    System.out.println("获取最先放置元素: " + s.firstElement());

    System.out.println("弹出一个元素[LIFO]: " + s.pop());
    System.out.println("弹出一个元素[LIFO]: " + s.pop());
    System.out.println("弹出一个元素[LIFO]: " + s.pop());
}
```

例子是对 `Stack` 栈的使用，如果不运行你能知道它的输出结果吗？

测试结果：

```
获取最后一个元素: ccc
获取最后一个元素: ccc
获取最先放置元素: aaa
弹出一个元素[LIFO]: ccc
弹出一个元素[LIFO]: bbb
弹出一个元素[LIFO]: aaa
```

```
Process finished with exit code 0
```

看到测试结果，与你想的答案是否一致？

- `peek`, 是偷看的意思，就是看一下，不会弹出元素。满足后进先出的规则，它看的是最后放进去的元素 `ccc`。
- `lastElement`、`firstElement`, 字面意思的方法，获取最后一个和获取第一个元素。
- `pop`, 是队列中弹出元素，弹出后也代表着要把属于这个位置都元素清空，删掉。

1.2 源码分析

我们说 `Stack` 栈，这个实现类已经不推荐使用了，需要从它的源码上看。

```
/*
 *
 * <p>A more complete and consistent set of LIFO stack operations is
 * provided by the {@link Deque} interface and its implementations, which
 * should be used in preference to this class. For example:
 *
 * <pre>  {@code
 *     Deque<Integer> stack = new ArrayDeque<Integer>();}</pre>
 *
 * @author  Jonathan Payne
 * @since   JDK1.0
 */
public class Stack<E> extends Vector<E>

    s.push("aaa");

    public synchronized void addElement(E obj) {
        modCount++;
        ensureCapacityHelper(elementCount + 1);
```

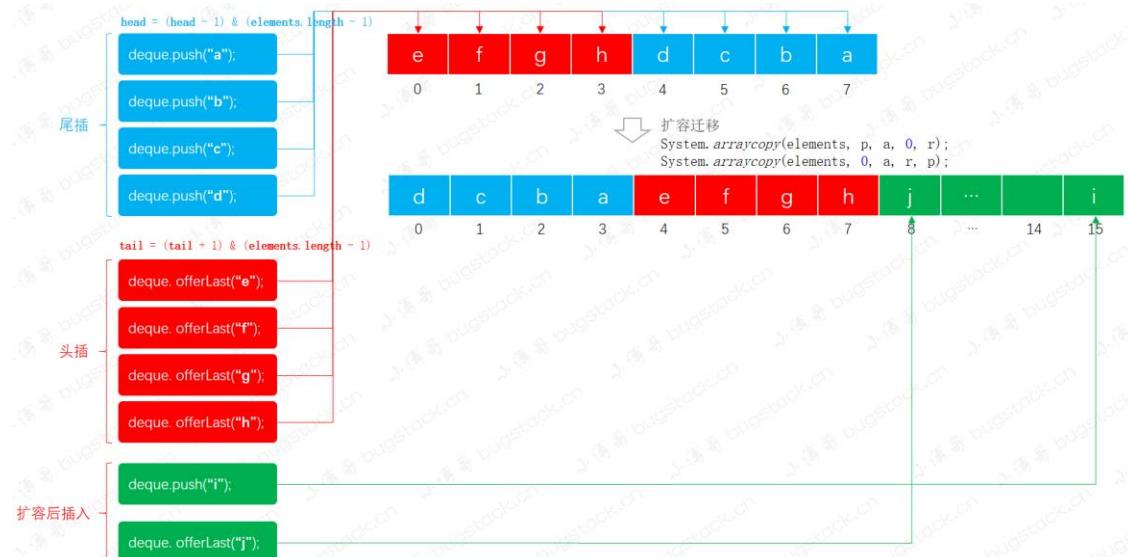
```
elementData[elementCount++] = obj;  
}
```

1. **Stack** 栈是在 JDK1.0 时代时，基于继承 **Vector**，实现的。本身 **Vector** 就是一个不推荐使用的类，主要在于它的一些操作方法锁 **synchronized** 的力度太粗，都是放到方法上。
2. **Stack** 栈底层是使用 **Vector** 数组实现，在学习 **ArrayList** 时候我们知道，数组结构在元素添加和擅长需要通过 **System.arraycopy**，进行扩容操作。而本身栈的特点是首尾元素的操作，也不需要遍历，使用数组结构其实并不太理想。
3. 同时在这个方法的注释上也明确标出来，推荐使用 **Deque<Integer> stack = new ArrayDeque<Integer>();**，虽然这也是数组结构，但是它没有粗粒度的锁，同时可以申请指定空间并且在扩容时操作时也要优于 **Stack**。并且它还是一个双端队列，使用起来更灵活。

2. 双端队列 **ArrayDeque**

ArrayDeque 是基于数组实现的可动态扩容的双端队列，也就是说你可以在队列的头和尾同时插入和弹出元素。当元素数量超过数组初始化长度时，则需要扩容和迁移数据。

数据结构和操作，如下：



小傅哥 bugstack.cn & 双端队列数据结构操作

从上图我们可以了解到如下几个知识点；

1. 双端队列是基于数组实现，所以扩容迁移数据操作。
2. **push**，像结尾插入、**offerLast**，向头部插入，这样两端都满足后进先出。
3. 整体来看，双端队列，就是一个环形。所以扩容后继续插入元素也满足后进先出。

2.1 功能使用

```
@Test  
public void test_ArrayDeque() {  
    Deque<String> deque = new ArrayDeque<String>(1);  
  
    deque.push("a");  
    deque.push("b");  
    deque.push("c");  
    deque.push("d");  
  
    deque.offerLast("e");  
    deque.offerLast("f");  
    deque.offerLast("g");  
    deque.offerLast("h"); // 这时候扩容了  
  
    deque.push("i");  
    deque.offerLast("j");  
  
    System.out.println("数据出栈: ");  
    while (!deque.isEmpty()) {  
        System.out.print(deque.pop() + " ");  
    }  
}
```

以上这部分代码就是与上图的展现是一致的，按照图中的分析我们看下输出结果，如下：

数据出栈：

```
i d c b a e f g h j  
Process finished with exit code 0
```

- **i d c b a e f g h j**，正好满足了我们的说的数据出栈顺序。可以参考上图再进行理解

2.2 源码分析

ArrayDeque 这种双端队列是基于数组实现的，所以源码上从初始化到数据入栈扩容，都会有数组操作的痕迹。接下来我们就依次分析下。

2.2.1 初始化

`new ArrayDeque<String>(1);`, 其实它的构造函数初始化默认也提供了几个方法, 比如你可以指定大小以及提供默认元素。

```
private static int calculateSize(int numElements) {  
    int initialCapacity = MIN_INITIAL_CAPACITY;  
    // Find the best power of two to hold elements.  
    // Tests "<=" because arrays aren't kept full.  
    if (numElements >= initialCapacity) {  
        initialCapacity = numElements;  
        initialCapacity |= (initialCapacity >>> 1);  
        initialCapacity |= (initialCapacity >>> 2);  
        initialCapacity |= (initialCapacity >>> 4);  
        initialCapacity |= (initialCapacity >>> 8);  
        initialCapacity |= (initialCapacity >>> 16);  
        initialCapacity++;  
        if (initialCapacity < 0) // Too many elements, must back off  
            initialCapacity >>>= 1; // Good Luck allocating 2 ^ 30 element  
    }  
    return initialCapacity;  
}
```

- 在初始化的过程中, 它需要找到你当前传输值最小的 2 的倍数的一个容量。这与 HashMap 的初始化过程相似。

2.2.2 数据入栈

`deque.push("a");`, ArrayDeque, 提供了一个 `push` 方法, 这个方法与 `deque.offerFirst("a")`, 一致, 因为它们的底层源码是一样的, 如下;

addFirst:

```
public void addFirst(E e) {  
    if (e == null)  
        throw new NullPointerException();  
    elements[head = (head - 1) & (elements.length - 1)] = e;  
    if (head == tail)  
        doubleCapacity();  
}
```

addLast:

```

public void addLast(E e) {
    if (e == null)
        throw new NullPointerException();
    elements[tail] = e;
    if ((tail = (tail + 1) & (elements.length - 1)) == head)
        doubleCapacity();
}

```

这部分入栈元素，其实就是给数组赋值，知识点如下；

1. 在 `addFirst()` 中，定位下标，`head = (head - 1) & (elements.length - 1)`，因为我们的数组长度是 2^n 的倍数，所以 $2^n - 1$ 就是一个全是 1 的二进制数，可以用于与运算得出数组下标。
2. 同样 `addLast()` 中，也使用了相同的方式定位下标，只不过它是从 0 开始，往上增加。
3. 最后，当头(head)与尾(tile)，数组则需要两倍扩容 `doubleCapacity`。

下标计算：`head = (head - 1) & (elements.length - 1)`:

- $(0 - 1) \& (8 - 1) = 7$
- $(7 - 1) \& (8 - 1) = 6$
- $(6 - 1) \& (8 - 1) = 5$
- ...

2.2.3 两倍扩容，数据迁移

```

private void doubleCapacity() {
    assert head == tail;
    int p = head;
    int n = elements.length;
    int r = n - p; // number of elements to the right of p
    int newCapacity = n << 1;
    if (newCapacity < 0)
        throw new IllegalStateException("Sorry, deque too big");
    Object[] a = new Object[newCapacity];
    System.arraycopy(elements, p, a, 0, r);
    System.arraycopy(elements, 0, a, r, p);
    elements = a;
    head = 0;
    tail = n;
}

```

其实以上这部分源码，就是进行两倍 $n \ll 1$ 扩容，同时把两端数据迁移进新的数组，整个操作过程也与我们上图对应。为了更好的理解，我们单独把这部分代码做一些测试。

测试代码：

```
@Test
public void test_arraycopy() {
    int head = 0, tail = 0;
    Object[] elements = new Object[8];
    elements[head = (head - 1) & (elements.length - 1)] = "a";
    elements[head = (head - 1) & (elements.length - 1)] = "b";
    elements[head = (head - 1) & (elements.length - 1)] = "c";
    elements[head = (head - 1) & (elements.length - 1)] = "d";

    elements[tail] = "e";
    tail = (tail + 1) & (elements.length - 1);
    elements[tail] = "f";
    tail = (tail + 1) & (elements.length - 1);
    elements[tail] = "g";
    tail = (tail + 1) & (elements.length - 1);
    elements[tail] = "h";
    tail = (tail + 1) & (elements.length - 1);

    System.out.println("head: " + head);
    System.out.println("tail: " + tail);

    int p = head;
    int n = elements.length;
    int r = n - p; // number of elements to the right of p

    // 输出当前的元素
    System.out.println(JSON.toJSONString(elements));

    // head == tail 扩容
    Object[] a = new Object[8 << 1];
    System.arraycopy(elements, p, a, 0, r);
    System.out.println(JSON.toJSONString(a));
    System.arraycopy(elements, 0, a, r, p);
```

```

        System.out.println(JSON.toJSONString(a));

        elements = a;
        head = 0;
        tail = n;
        a[head = (head - 1) & (a.length - 1)] = "i";
        System.out.println(JSON.toJSONString(a));
    }
}

```

以上的测试过程主要模拟了 8 个长度的空间的数组，在进行双端队列操作时数组扩容，数据迁移操作，可以单独运行，测试结果如下：

```

head: 4
tail: 4
["e", "f", "g", "h", "d", "c", "b", "a"]
["d", "c", "b", "a", null, null]
["d", "c", "b", "a", "e", "f", "g", "h", null, null, null, null, null, null, null]
["d", "c", "b", "a", "e", "f", "g", "h", "j", null, null, null, null, null, "i"]

```

Process finished with exit code 0

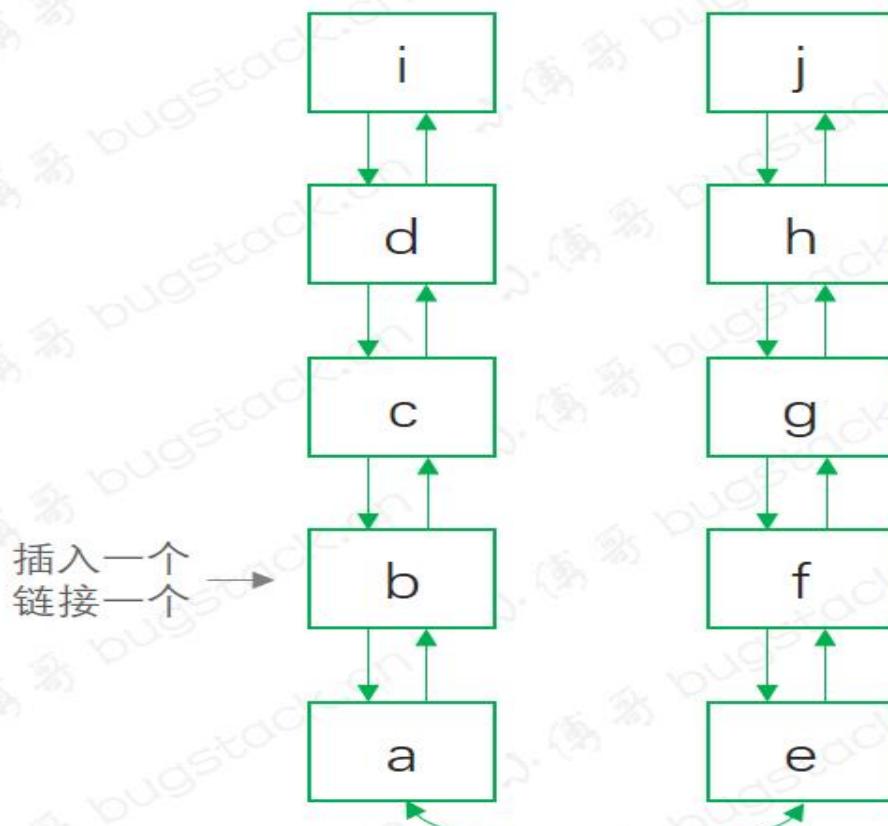
从测试结果可以看到：

1. 当 head 与 tail 相等时，进行扩容操作。
2. 第一次数据迁移，`System.arraycopy(elements, p, a, 0, r);`, d、c、b、a，落入新数组。
3. 第二次数据迁移，`System.arraycopy(elements, 0, a, r, p);`, e、f、g、h，落入新数组。
4. 最后再尝试添加新的元素，i 和 j。每一次的输出结果都可以看到整个双端链路的变化。

3. 双端队列 LinkedList

`LinkedList`天生就可以支持双端队列，而且从头尾取数据也是它时间复杂度 $O(1)$ 的。同时数据的插入和删除也不需要像数组队列那样拷贝数据，虽然 `LinkedList` 有这些优点，但不能说 `ArrayDeque` 因为有数组复制性能比它低。

`LinkedList`, 数据结构:



插入顺序:

```
push("a");
push("b");
push("c");
push("d");
```

```
offerLast("e");
offerLast("f");
offerLast("g");
offerLast("h");
```

3.1 功能使用

```
@Test
public void test_Deque_LinkedList(){
    Deque<String> deque = new LinkedList<>();
    deque.push("a");
    deque.push("b");
    deque.push("c");
}
```

```

deque.push("d");
deque.offerLast("e");
deque.offerLast("f");
deque.offerLast("g");
deque.offerLast("h");
deque.push("i");
deque.offerLast("j");

System.out.println("数据出栈: ");
while (!deque.isEmpty()) {
    System.out.print(deque.pop() + " ");
}
}

```

测试结果:

数据出栈:

i d c b a e f g h j

Process finished with exit code 0

- 测试结果上看与使用 `ArrayDeque` 是一样的，功能上没有差异。

3.2 源码分析

压栈: `deque.push("a");`、`deque.offerFirst("a");`

```

private void linkFirst(E e) {
    final Node<E> f = first;
    final Node<E> newNode = new Node<>(null, e, f);
    first = newNode;
    if (f == null)
        last = newNode;
    else
        f.prev = newNode;
    size++;
    modCount++;
}

```

压栈: `deque.offerLast("e");`

```

void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}

```

- `linkFirst`、`linkLast`，两个方法分别是给链表的首尾节点插入元素，因为这是链表结构，所以也不存在扩容，只需要把双向链路链接上即可。

4. 延时队列 DelayQueue

你是否有时候需要把一些数据存起来，倒计时到某个时刻在使用？

在 Java 的队列数据结构中，还有一种队列是延时队列，可以通过设定存放时间，依次轮训获取。

4.1 功能使用

先写一个 `Delayed` 的实现类：

```

public class TestDelayed implements Delayed {

    private String str;
    private long time;

    public TestDelayed(String str, long time, TimeUnit unit) {
        this.str = str;
        this.time = System.currentTimeMillis() + (time > 0 ? unit.toMillis(time) :
        0);
    }

    @Override
    public long getDelay(TimeUnit unit) {
        return time - System.currentTimeMillis();
    }
}

```

```

@Override
public int compareTo(Delayed o) {
    TestDelayed work = (TestDelayed) o;
    long diff = this.time - work.time;
    if (diff <= 0) {
        return -1;
    } else {
        return 1;
    }
}

public String getStr() {
    return str;
}

```

- 这个相当于延时队列的一个固定模版方法，通过这种方式来控制延时。

案例测试：

```

@Test
public void test_DelayQueue() throws InterruptedException {
    DelayQueue<TestDelayed> delayQueue = new DelayQueue<TestDelayed>();
    delayQueue.offer(new TestDelayed("aaa", 5, TimeUnit.SECONDS));
    delayQueue.offer(new TestDelayed("ccc", 1, TimeUnit.SECONDS));
    delayQueue.offer(new TestDelayed("bbb", 3, TimeUnit.SECONDS));

    logger.info(((TestDelayed) delayQueue.take()).getStr());
    logger.info(((TestDelayed) delayQueue.take()).getStr());
    logger.info(((TestDelayed) delayQueue.take()).getStr());
}

```

测试结果：

```

01:44:21.000 [main] INFO org.itstack.interview.test.ApiTest - ccc
01:44:22.997 [main] INFO org.itstack.interview.test.ApiTest - bbb
01:44:24.997 [main] INFO org.itstack.interview.test.ApiTest - aaa

```

Process finished with exit code 0

- 在案例测试中我们分别设定不同的休眠时间，1、3、5，`TimeUnit.SECONDS`。
- 测试结果分别在21、22、24，输出了我们要的队列结果。
- 队列中的元素不会因为存放的先后顺序而导致输出顺序，它们是依赖于休眠时长决定。

4.2 源码分析

4.2.1 元素入栈

入栈：：`delayQueue.offer(new TestDelayed("aaa", 5, TimeUnit.SECONDS));`

```
public boolean offer(E e) {
    if (e == null)
        throw new NullPointerException();
    modCount++;
    int i = size;
    if (i >= queue.length)
        grow(i + 1);
    size = i + 1;
    if (i == 0)
        queue[0] = e;
    else
        siftUp(i, e);
    return true;
}

private void siftUpUsingComparator(int k, E x) {
    while (k > 0) {
        int parent = (k - 1) >>> 1;
        Object e = queue[parent];
        if (comparator.compare(x, (E) e) >= 0)
            break;
        queue[k] = e;
        k = parent;
    }
    queue[k] = x;
}
```

- 关于数据存放还有 `ReentrantLock` 可重入锁，但暂时不是我们本章节数据结构的重点，后面章节会介绍到。

- **DelayQueue** 是基于数组实现的，所以可以动态扩容，另外它插入元素的顺序并不影响最终的输出顺序。
- 而元素的排序依赖于 `compareTo` 方法进行排序，也就是休眠的时间长短决定的。
- 同时只有实现了 **Delayed** 接口，才能存放元素。

4.2.2 元素出栈

出栈: `delayQueue.take()`

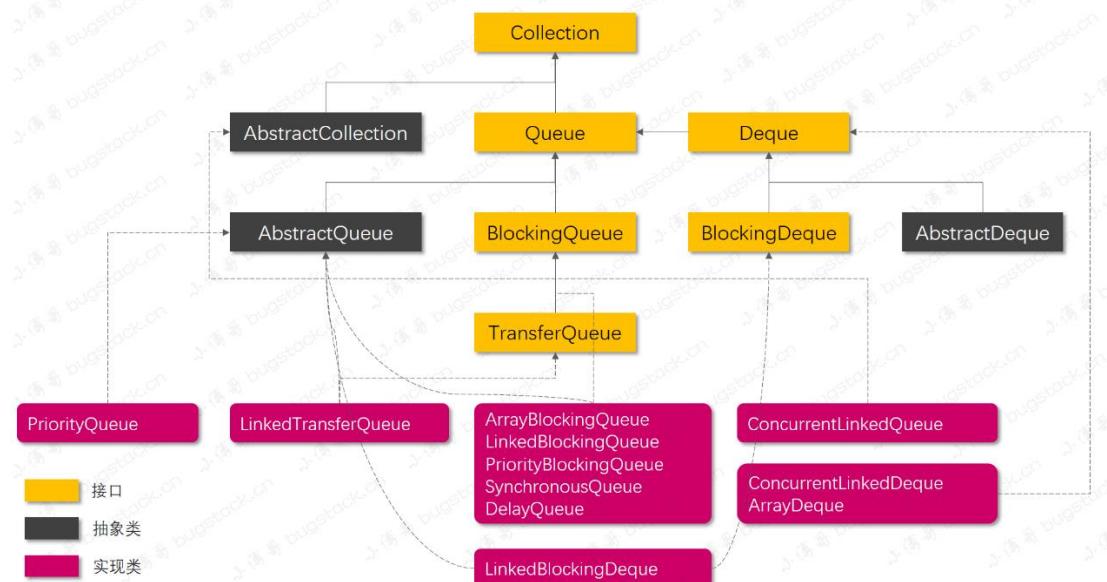
```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        for (;;) {
            E first = q.peek();
            if (first == null)
                available.await();
            else {
                long delay = first.getDelay(NANOSECONDS);
                if (delay <= 0)
                    return q.poll();
                first = null; // don't retain ref while
                if (leader != null)
                    available.await();
                else {
                    Thread thisThread = Thread.currentThread();
                    leader = thisThread;
                    try {
                        available.awaitNanos(delay);
                    } finally {
                        if (leader == thisThread)
                            leader = null;
                    }
                }
            }
        }
    } finally {
        if (leader == null && q.peek() != null)
            available.signal();
        lock.unlock();
    }
}
```

```
    }  
}
```

- 这部分的代码有点长，主要是元素的获取。`DelayQueue` 是 `Leader-Follower` 模式的变种，消费者线程处于等待 `await` 时，总是等待最先休眠完成的元素。
- 这里会最小化的空等时间，提高线程利用率。数据结构讲完后，后面会有专门章节介绍

5. 还有哪些队列？

5.1 队列类结构



类型	实现	描述
Queue	<code>LinkedBlockingQueue</code>	由链表结构组成的有界阻塞队列
Queue	<code>ArrayBlockingQueue</code>	由数组结构组成的有界阻塞队列
Queue	<code>PriorityBlockingQueue</code>	支持优先级排序的无界阻塞队列
Queue	<code>SynchronousQueue</code>	不存储元素的阻塞队列
Queue	<code>LinkedTransferQueue</code>	由链表结构组成的无界阻塞队列
Deque	<code>LinkedBlockingDeque</code>	由链表结构组成的双向阻塞队列
Deque	<code>ConcurrentLinkedDeque</code>	由链表结构组成的线程安全的双向阻塞队列

- 除了我们已经讲过的队列以外，剩余的基本都是阻塞队列，也就是上面这些。
- 在数据结构方面基本没有差异，只不过添加了相应的阻塞功能和锁的机制。

5.2 使用案例

```

public class DataQueueStack {

    private BlockingQueue<DataBean> dataQueue = null;

    public DataQueueStack(){
        //实例化队列
        dataQueue = new LinkedBlockingQueue<DataBean>(100);
    }

    /**
     * 添加数据到队列
     * @param dataBean
     * @return
     */
    public boolean doOfferData(DataBean dataBean){
        try {
            return dataQueue.offer(dataBean, 2, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 弹出队列数据
     * @return
     */
    public DataBean doPollData(){
        try {
            return dataQueue.poll(2, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
            return null;
        }
    }
}

```

```
}

/**
 * 获得队列数据个数
 * @return
 */
public int doGetQueueCount(){
    return dataQueue.size();
}

}
```

- 这是一个 `LinkedBlockingQueue` 队列使用案例，一方面存储数据，一方面从队列中获取进行消费。
- 因为这是一个阻塞队列，所以在获取元素的时候，如果队列为空，会进行阻塞。
- `LinkedBlockingQueue` 是一个阻塞队列，内部由两个 `ReentrantLock` 来实现出入队列的线程安全，由各自的 `Condition` 对象的 `await` 和 `signal` 来实现等待和唤醒功能。

四、总结

- 关于栈和队列的数据结构方面到这里就介绍完了，另外这里还有一些关于阻塞队列锁🔒的应用过程，到我们后面讲锁相关知识点，再重点介绍。
- 队列结构的设计非常适合某些需要 LIFO 或者 FIFO 的应用场景，同时在队列的数据结构中也有双端、延时和组合的功能类，使用起来也非常方便。
- 数据结构方面的知识到本章节算是告一段落，如果有优秀的内容，后面还会继续补充。再下一章节小傅哥(bugstack.cn)准备给大家介绍，关于数据结构中涉及的算法部分，这些主要来自于 Collections 类的实现部分。

第 9 节：java.util.Collections、排序、二分、洗牌、旋转算法

算法是数据结构的灵魂！

好的算法搭配上合适的数据结构，可以让代码功能大大的提升效率。当然，算法学习不只是刷题，还需要落地与应用，否则到了写代码的时候，还是会 `for` 循环 + `if` `else`。

当开发一个稍微复杂点的业务流程时，往往要用到与之契合的数据结构和算法逻辑，在与设计模式结合，这样既能让你的写出具有高性能的代码，也能让这些代码具备良好的扩展性。

那么，有了这些数据结构的基础，接下来我们再学习一下 Java 中提供的算法工具类，`Collections`。

一、面试题

谢飞机，今天怎么无精打采的呢，还有黑眼圈？

答：好多知识盲区呀，最近一直在努力补短板，还有[面经手册](#)里的数据结构。

问：那数据结构看的差不多了吧，你有考虑过，数据结构里涉及的排序、二分查找吗？

答：二分查找会一些，巴拉巴拉。

问：还不错，那你知道这个方法在 Java 中有提供对应的工具类吗？是哪个！

答：这！？好像没注意过，没用过！

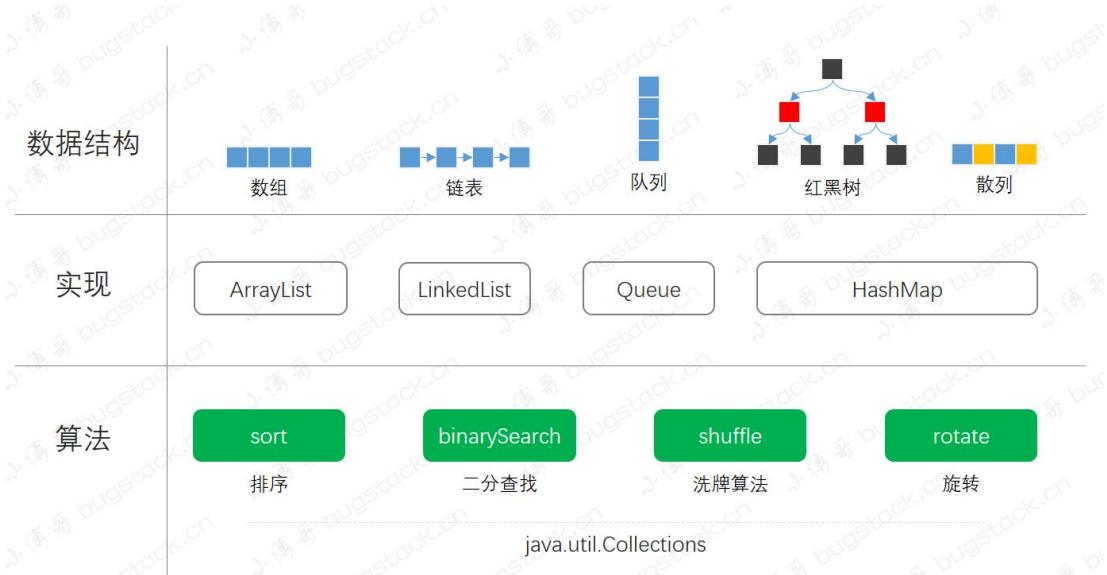
问：去吧，回家在看看书，这两天也休息下。

飞机悄然的出门了，但这次面试题整体回答的还是不错的，面试官决定下次再给他一个机会。

二、Collections 工具类

`java.util.Collections`，是 java 集合框架的一个工具类，主要用于 `Collection` 提供的通用算法；排序、二分查找、洗牌等算法操作。

从数据结构到具体实现，再到算法，整体的结构如下图：



1. Collections.sort 排序

1.1 初始化集合

```
List<String> list = new ArrayList<String>();  
list.add("7");  
list.add("4");  
list.add("8");  
list.add("3");  
list.add("9");
```

1.2 默认排列[正序]

```
Collections.sort(list);
```

```
// 测试结果: [3, 4, 7, 8, 9]
```

1.3 Comparator 排序

```
Collections.sort(list, new Comparator<String>() {  
    @Override  
    public int compare(String o1, String o2) {  
        return o2.compareTo(o1);  
    }  
});
```

- 我们使用 `o2` 与 `o1` 做对比，这样会出来一个倒叙排序。
- 同时，`Comparator` 还可以对对象类按照某个字段进行排序。
- 测试结果如下：

```
[9, 8, 7, 4, 3]
```

1.4 reverseOrder 倒排

```
Collections.sort(list, Collections.<String>reverseOrder());
```

```
// 测试结果: [9, 8, 7, 4, 3]
```

- `Collections.<String>reverseOrder()` 的源码部分就和我们上面把两个对比的类调换过来一样。`c2.compareTo(c1);`

1.5 源码简述

关于排序方面的知识点并不少，而且有点复杂。本文主要介绍 `Collections` 集合工具类，后续再深入每一个排序算法进行讲解。

`Collections.sort`

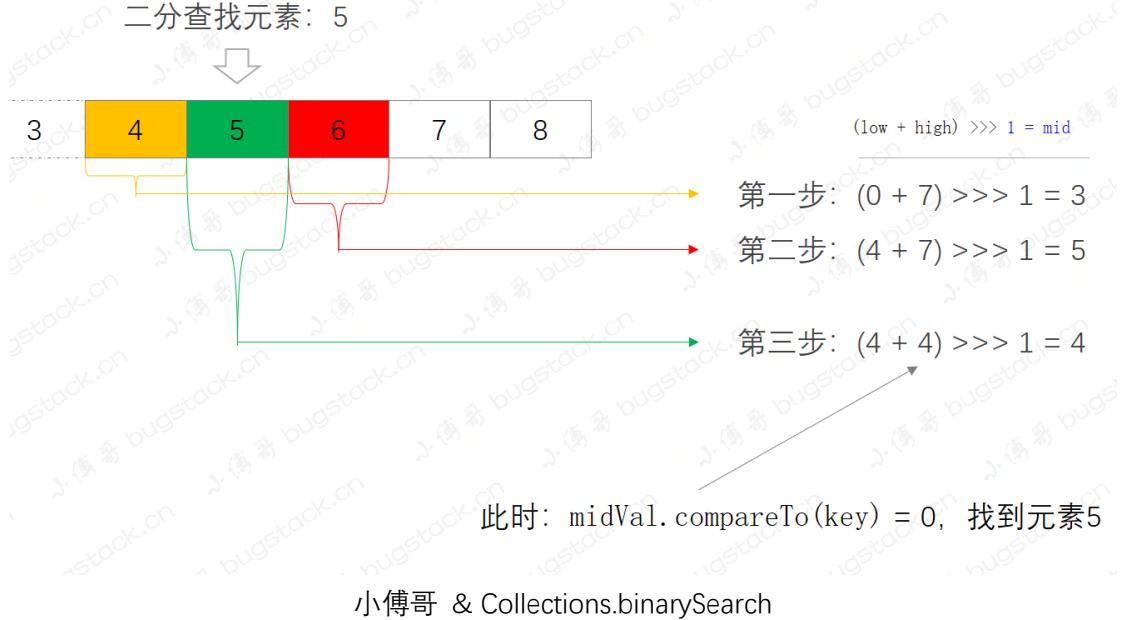
集合排序，最终使用的方法：`Arrays.sort((E[]) elementData, 0, size, c);`

```
public static <T> void sort(T[] a, int fromIndex, int toIndex,
                           Comparator<? super T> c) {
    if (c == null) {
        sort(a, fromIndex, toIndex);
    } else {
        rangeCheck(a.length, fromIndex, toIndex);
        if (LegacyMergeSort.userRequested)
            legacyMergeSort(a, fromIndex, toIndex, c);
        else
            TimSort.sort(a, fromIndex, toIndex, c, null, 0, 0);
    }
}
```

- 这一部分排序逻辑包括了，旧版的归并排序 `legacyMergeSort` 和 `TimSort` 排序。
- 但因为开关的作用，`LegacyMergeSort.userRequested = false`，基本都是走到 `TimSort` 排序。

- 同时在排序的过程中还会因为元素的个数是否大于 32，而选择分段排序和二分插入排序。这一部分内容我们后续专门在排序内容讲解

2. Collections.binarySearch 二分查找



- 看到这张图熟悉吗，这就是集合元素中通过二分查找定位指定元素 5。
- 二分查找的前提是集合有序，否则不能满足二分算法的查找过程。
- 查找集合元素 5，在这个集合中分了三部；
 - 第一步， $(0 + 7) >>> 1 = 3$ ，定位 `list.get(3) = 4`，则继续向右侧二分查找。
 - 第二步， $(4 + 7) >>> 1 = 5$ ，定位 `list.get(5) = 6`，则继续向左侧二分查找。
 - 第三步， $(4 + 4) >>> 1 = 4$ ，定位 `list.get(4) = 5`，找到元素，返回结果。

2.1 功能使用

```
@Test
public void test_binarySearch() {
    List<String> list = new ArrayList<String>();
    list.add("1");
    list.add("2");
```

```

        list.add("3");
        list.add("4");
        list.add("5");
        list.add("6");
        list.add("7");
        list.add("8");

        int idx = Collections.binarySearch(list, "5");
        System.out.println("二分查找: " + idx);
    }
}

```

- 此功能就是上图中的具体实现效果，通过 `Collections.binarySearch` 定位元素。
- 测试结果：二分查找：4

2.2 源码分析

`Collections.binarySearch`

```

public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key) {
    if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
        return Collections.indexedBinarySearch(list, key);
    else
        return Collections.iteratorBinarySearch(list, key);
}

```

这段二分查找的代码还是蛮有意思的，`list instanceof RandomAccess` 这是为什么呢？因为 `ArrayList` 有实现 `RandomAccess`，但是 `LinkedList` 并没有实现这个接口。同时还有元素数量阈值的校验 `BINARYSEARCH_THRESHOLD = 5000`，小于这个范围的都采用 `indexedBinarySearch` 进行查找。那么这里其实使用 `LinkedList` 存储数据，在元素定位的时候，需要 `get` 循环获取元素，就会比 `ArrayList` 更耗时。

`Collections.indexedBinarySearch(list, key)`:

```

private static <T> int indexedBinarySearch(List<? extends Comparable<? super T>> list, T key) {
    int low = 0;

```

```

int high = list.size()-1;
while (low <= high) {
    int mid = (low + high) >>> 1;
    Comparable<? super T> midVal = list.get(mid);
    int cmp = midVal.compareTo(key);
    if (cmp < 0)
        low = mid + 1;
    else if (cmp > 0)
        high = mid - 1;
    else
        return mid; // key found
}
return -(low + 1); // key not found
}

```

以上这段代码就是通过每次折半二分定位元素，而上面所说的耗时点就是 `list.get(mid)`，这在我们分析数据结构时已经讲过。

`Collections.iteratorBinarySearch(list, key):`

```

private static <T> int iteratorBinarySearch(List<? extends Comparable<? super T>> list, T key)
{
    int low = 0;
    int high = list.size()-1;
    ListIterator<? extends Comparable<? super T>> i = list.listIterator();
    while (low <= high) {
        int mid = (low + high) >>> 1;
        Comparable<? super T> midVal = get(i, mid);
        int cmp = midVal.compareTo(key);
        if (cmp < 0)
            low = mid + 1;
        else if (cmp > 0)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1); // key not found
}

```

上面这段代码是元素数量大于 5000 个，同时是 `LinkedList` 时会使用迭代器 `list.listIterator` 的方式进行二分查找操作。也算是一个优化，但是对于链表的数据结构，仍然没有数组数据结构，二分处理的速度快，主要在获取元素的时间复杂度上 $O(1)$ 和 $O(n)$ 。

3. Collections.shuffle 洗牌算法

洗牌算法，其实就是将 `List` 集合中的元素进行打乱，一般可以用在抽奖、摇号、洗牌等各个场景中。

3.1 功能使用

```
Collections.shuffle(list);  
  
Collections.shuffle(list, new Random(100));
```

它的使用方式非常简单，主要有这么两种方式，一种直接传入集合、另外一种还可以传入固定的随机种子这种方式可以控制洗牌范围范围

3.2 源码分析

按照洗牌的逻辑，我们来实现下具体的核心逻辑代码，如下：

```
@Test  
public void test_shuffle() {  
    List<String> list = new ArrayList<String>();  
    list.add("1");  
    list.add("2");  
    list.add("3");  
    list.add("4");  
    list.add("5");  
    list.add("6");  
    list.add("7");  
    list.add("8");  
  
    Random random = new Random();  
    for (int i = list.size(); i > 1; i--) {  
        int ri = random.nextInt(i); // 随机位置  
        int ji = i - 1; // 顺延  
        System.out.println("ri: " + ri + " ji: " + ji);  
        list.set(ji, list.set(ri, list.get(ji))); // 元素置换
```

```

    }
    System.out.println(list);
}

```

运行结果：

```

ri: 6 ji: 7
ri: 4 ji: 6
ri: 1 ji: 5
ri: 2 ji: 4
ri: 0 ji: 3
ri: 0 ji: 2
ri: 1 ji: 1
[8, 6, 4, 1, 3, 2, 5, 7]

```

这部分代码逻辑主要是通过随机数从逐步缩小范围的集合中找到对应的元素，与递减的下标位置进行元素替换，整体的执行过程如下；



4. Collections.rotate 旋转算法

旋转算法，可以把 ArrayList 或者 Linkedlist，从指定的某个位置开始，进行正旋或者逆旋操作。有点像把集合理解成圆盘，把要的元素转到自己这，其他的元素顺序跟随。

4.1 功能应用

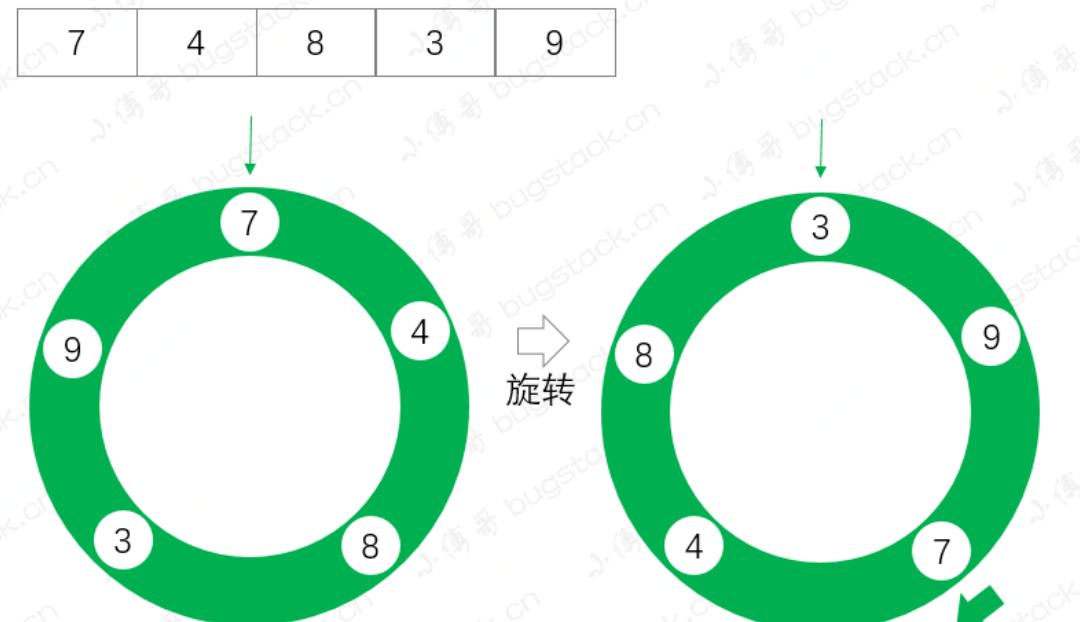
```
List<String> list = new ArrayList<String>();  
list.add("7");  
list.add("4");  
list.add("8");  
list.add("3");  
list.add("9");  
Collections.rotate(list, 2);  
System.out.println(list);
```

这里我们将集合顺序；7、4、8、3、9，顺时针旋转2位，测试结果如下；逆时针旋转为负数

测试结果

[3, 9, 7, 4, 8]

通过测试结果我们可以看到，从元素7开始，正向旋转了两位，执行效果如下图：



4.2 源码分析

```
public static void rotate(List<?> list, int distance) {  
    if (list instanceof RandomAccess || list.size() < ROTATE_THRESHOLD)  
        rotate1(list, distance);
```

```
    else
        rotate2(list, distance);
}
```

关于旋转算法的实现类分为两部分：

1. ArrayList 或者元素数量不多时，ROTATE_THRESHOLD = 100，则通过算法rotate1实现。
2. 如果是 LinkedList 元素数量又超过了 ROTATE_THRESHOLD，则需要使用算法rotate2实现。

那么，这两个算法有什么不同呢？

4.2.1 旋转算法，rotate1

```
private static <T> void rotate1(List<T> list, int distance) {
    int size = list.size();
    if (size == 0)
        return;
    distance = distance % size;
    if (distance < 0)
        distance += size;
    if (distance == 0)
        return;
    for (int cycleStart = 0, nMoved = 0; nMoved != size; cycleStart++) {
        T displaced = list.get(cycleStart);
        int i = cycleStart;
        do {
            i += distance;
            if (i >= size)
                i -= size;
            displaced = list.set(i, displaced);
            nMoved++;
        } while (i != cycleStart);
    }
}
```

这部分代码看着稍微多一点，但是数组结构的操作起来并不复杂，基本如上面圆圈图操作，主要包括以下步骤：

1. `distance = distance % size;`，获取旋转的位置。

2. for 循环和 dowhile，配合每次的旋转操作，比如这里第一次会把 0 位置元素替换到 2 位置，接着在 dowhile 中会判断 `i != cycleStart`，满足条件继续把替换了 2 位置的元素继续往下替换。直到一轮循环把所有元素都放置到正确位置。
3. 最终 list 元素被循环替换完成，也就相当从某个位置开始，正序旋转 2 个位置的操作。

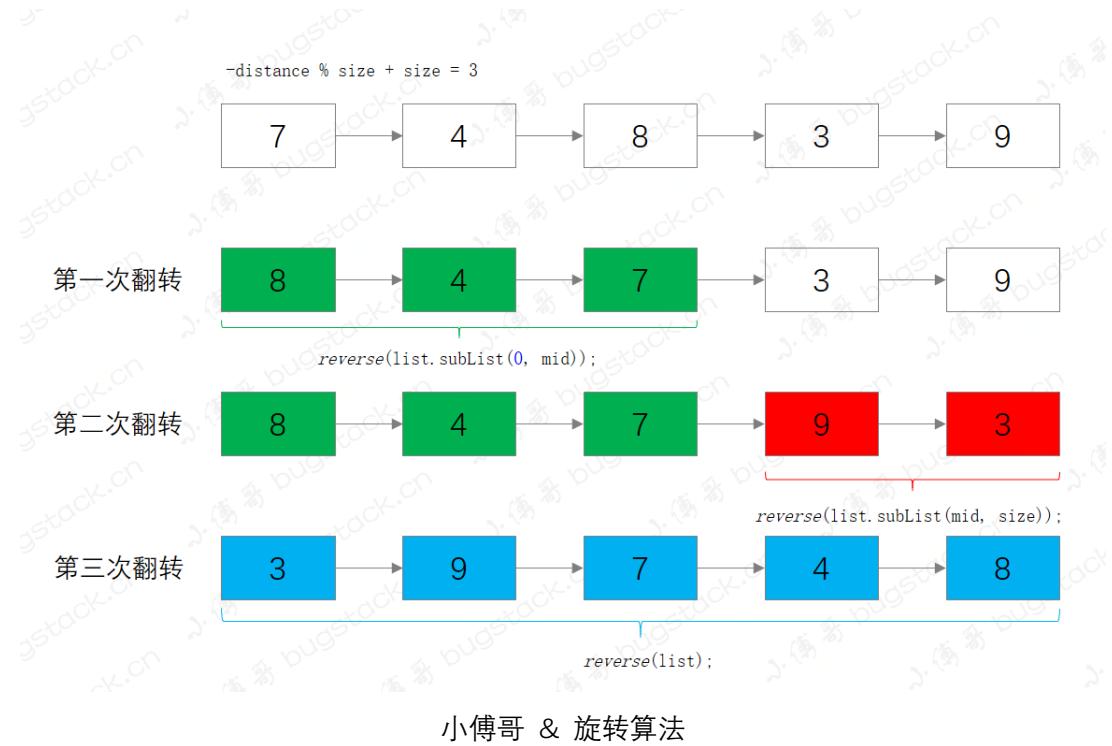
4.2.2 旋转算法，rotate2

```
private static void rotate2(List<?> list, int distance) {
    int size = list.size();
    if (size == 0)
        return;
    int mid = -distance % size;
    if (mid < 0)
        mid += size;
    if (mid == 0)
        return;
    reverse(list.subList(0, mid));
    reverse(list.subList(mid, size));
    reverse(list);
}
```

接下来这部分源码主要是针对大于 `100` 个元素的 LinkedList 进行操作，所以整个算法也更加有意思，它的主要操作包括：

1. 定位拆链位置，`-distance % size + size`，也就是我们要旋转后找到的元素位置
2. 第一次翻转，把从位置 0 到拆链位置
3. 第二次翻转，把拆链位置到结尾
4. 第三次翻转，翻转整个链表

整体执行过程，可以参考下图，更方便理解：



5. 其他 API 介绍

这部分 API 内容，使用和设计上相对比较简单，平时可能用的时候不多，但有的小伙伴还没用过，就当为你扫盲了。

5.1 最大最小值

```
String min = Collections.min(Arrays.asList("1", "2", "3"));
String max = Collections.max(Arrays.asList("1", "2", "3"));
```

- `Collections` 工具包可以进行最值的获取。

5.2 元素替换

```
List<String> list = new ArrayList<String>();
list.add("7");
list.add("4");
list.add("8");
list.add("8");
Collections.replaceAll(list, "8", "9");

// 测试结果: [7, 4, 9, 9]
```

- 可以把集合中某个元素全部替换成新的元素。

5.3 连续集合位置判断

```
@Test
public void test_indexOfSubList() {
    List<String> list = new ArrayList<String>();
    list.add("7");
    list.add("4");
    list.add("8");
    list.add("3");
    list.add("9");
    int idx = Collections.indexOfSubList(list, Arrays.asList("8", "3"));
    System.out.println(idx);

    // 测试结果: 2
}
```

在使用 String 操作中，我们知道 `"74839".indexOf("8");`，可以获取某个元素在什么位置。而在 `ArrayList` 集合操作中，可以获取连续的元素，在集合中的位置。

5.4 synchronized

```
List<String> list = Collections.synchronizedList(new ArrayList<String>());
Map<String, String> map = Collections.synchronizedMap(new HashMap<String, String>());
```

- 这个很熟悉吧，甚至经常使用，但可能会忽略这些线程安全的方法来自于 `Collections` 集合工具包。

三、总结

- 本章节基本将 `java.util.Collections` 工具包中的常用方法介绍完了，以及一些算法的讲解。这样在后续需要使用到这些算法逻辑时，就可以直接使用并不需要重复造轮子。
- 学习数据结构、算法、设计模式，这三方面的知识，重点还是能落地到日常的业务开发中，否则空、假、虚，只能适合吹吹牛，并不会给项目研发带来实际的价值。
- 懂了就是真的懂，别让自己太难受。死记硬背谁也受不了，耗费了大量的时间，知识也没有吸收，学习一个知识点最好就从根本学习，不要心浮气躁。

第 10 节：StringBuilder 与 String 对比

聊的是八股的文，干的是搬砖的活！

面我的题开发都用不到，你为什么要问？可能这是大部分程序员求职时的经历，甚至也是大家讨厌和烦躁的点。明明给的是拧螺丝的钱、明明做的是写 CRUD 的事、明明担的是成工具的人！

明明... 有很多，可明明公司不会招 5 年开发做 3 年经验的事、明明公司也更喜欢具有附加价值的研发。有些小公司不好说，但在一些互联网大厂中，我们都希望招聘到具有培养价值的，也更喜欢能快速打怪升级的，也更愿意让这样的人承担更大的职责。

但，你酸了！别人看源码你打游戏、别人学算法你刷某音、别人写博客你浪 98。

所以，没有把时间用到个人成长上，就一直会被别人榨取。

一、面试题

谢飞机，总感觉自己有技术瓶颈、有知识盲区，但是又不知道在哪。所以约面试官聊天，虽然也面不过去！

面试官：飞机，你又抱着大脸，来白嫖我了啦？

谢飞机：嘿嘿，我需要知识，我渴。

面试官：好，那今天聊聊最常用的 `String` 吧，你怎么初始化一个字符串类型。

谢飞机：`String str = "abc";`

面试官：还有吗？

谢飞机：还有？啊，这样 `String str = new String("abc");` ☺

面试官：还有吗？

谢飞机：啊！？还有！不知道了！

面试官：你不懂 `String`，你没看过源码。还可以这样；`new String(new char[]{'c', 'd'})`；回家再学学吧，下次记得给我买百事，我不喝可口。

二、StringBuilder 比 String 快吗？

1. StringBuilder 比 String 快，证据呢？

老子代码一把梭，总有人絮叨这么搞不好，那 `StringBuilder` 到底那快了！

1.1 String

```
long startTime = System.currentTimeMillis();
String str = "";
for (int i = 0; i < 1000000; i++) {
    str += i;
}
System.out.println("String 耗时: " + (System.currentTimeMillis() - startTime) + "毫秒");
```

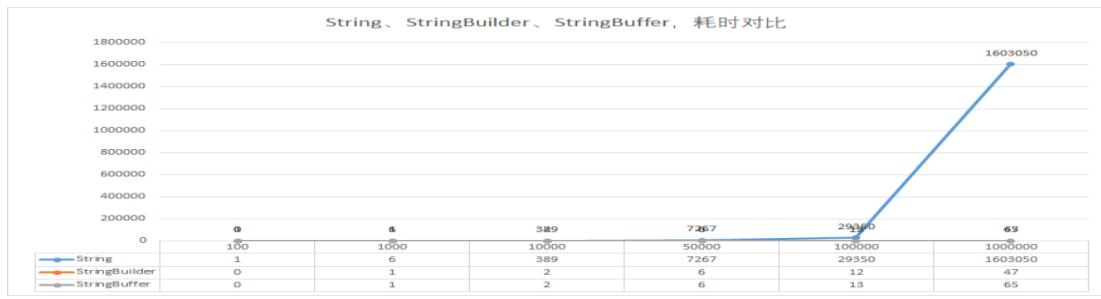
1.2 StringBuilder

```
long startTime = System.currentTimeMillis();
StringBuilder str = new StringBuilder();
for (int i = 0; i < 1000000; i++) {
    str.append(i);
}
System.out.println("StringBuilder 耗时
" + (System.currentTimeMillis() - startTime) + "毫秒");
```

1.3 StringBuffer

```
long startTime = System.currentTimeMillis();
StringBuffer str = new StringBuffer();
for (int i = 0; i < 1000000; i++) {
    str.append(i);
}
System.out.println("StringBuffer 耗时
" + (System.currentTimeMillis() - startTime) + "毫秒");
```

综上，分别使用了 `String`、`StringBuilder`、`StringBuffer`，做字符串链接操作（100个、1000个、1万个、10万个、100万个），记录每种方式的耗时。最终汇总图表如下：



小傅哥 & 耗时对比

从上图可以得出以下结论；

1. **String** 字符串链接是耗时的，尤其数据量大的时候，简直没法使用了。这是做实验，基本也不会有人这么干！
2. **StringBuilder**、**StringBuffer**，因为没有发生多线程竞争也就没有**锁升级**，所以两个类耗时几乎相同，当然在单线程下更推荐使用 **StringBuilder**。

2. StringBuilder 比 String 快，为什么？

```
String str = "";
for (int i = 0; i < 10000; i++) {
    str += i;
}
```

这段代码就是三种字符串拼接方式，最慢的一种。不是说这种`+`加的符号，会被优化成 **StringBuilder** 吗，那怎么还慢？

确实会被 JVM 编译期优化，但优化成什么样子了呢，先看下字节码指令；`javap -c ApiTest.class`

```
public void test_03();
  Code:
    0: invokestatic #16           // Method java/lang/System.currentTimeMillis:()J
    3: istore_1
    4: ldc      #17             // String
    6: astore_3
    7: iconst_0
    8: istore   4
   10: iload    4
   12: ldc      #18             // int 1000000
   14: if_icmpge #43
   17: new     #11              // class java/lang/StringBuilder
   20: dup
   21: invokespecial #12         // Method java/lang/StringBuilder."<init>":()V
   24: aload_3
   25: invokevirtual #13         // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   28: iload    4
   30: invokevirtual #19         // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
   33: invokevirtual #15         // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   36: astore_3
   37: iinc    4, 1
   40: goto    10
   43: getstatic #3             // Field java/lang/System.out:Ljava/io/PrintStream;
   46: new     #11              // class java/lang/StringBuilder
   49: dup
   50: invokespecial #12         // Method java/lang/StringBuilder."<init>":()V
   53: ldc      #20             // String String 耗时:
   55: invokevirtual #13         // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   58: invokevirtual #16         // Method java/lang/System.currentTimeMillis:()J
   61: iload_1
   62: isub
   63: invokevirtual #21         // Method java/lang/StringBuilder.append:(J)Ljava/lang/StringBuilder;
   66: ldc      #22             // String毫秒
   68: invokevirtual #13         // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   71: invokevirtual #15         // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   74: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   77: return
```

小傅哥 & 反编译

一看指令码，这不是在循环里(*if_icmpgt*)给我 `new` 了 `StringBuilder` 了吗，怎么还这么慢呢？再仔细看，其实你会发现，这 `new` 是在循环里吗呀，我们把这段代码写出来再看看：

```
String str = "";
for (int i = 0; i < 10000; i++) {
    str = new StringBuilder().append(str).append(i).toString();
}
```

现在再看这段代码就很清晰了，所有的字符串链接操作，都需要实例化一次 `StringBuilder`，所以非常耗时。并且你可以验证，这样写代码耗时与字符串直接链接是一样的。所以把 `StringBuilder` 提到上一层 `for` 循环外更快。

三、String 源码分析

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0

    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = -6849794470754667710L;

    ...
}
```

1. 初始化

在与 [谢飞机](#) 的面试题中，我们聊到了 `String` 初始化的问题，按照一般我们应用的频次上，能想到的只有直接赋值，`String str = "abc";`，但因为 `String` 的底层数据结构是数组 `char value[]`，所以它的初始化方式也会有很多跟数组相关的，如下：

```
String str_01 = "abc";
System.out.println("默认方式: " + str_01);

String str_02 = new String(new char[]{'a', 'b', 'c'});
System.out.println("char 方式: " + str_02);
```

```

String str_03 = new String(new int[]{0x61, 0x62, 0x63}, 0, 3);
System.out.println("int 方式: " + str_03);

String str_04 = new String(new byte[]{0x61, 0x62, 0x63});
System.out.println("byte 方式: " + str_04);

```

以上这些方式都可以初始化，并且最终的结果是一致的，abc。如果说初始化的方式没用让你感受到它是数据结构，那么 `str_01.charAt(0);` 呢，只要你往源码里一点，就会发现它是 `O(1)` 的时间复杂度从数组中获取元素，所以效率也是非常高，源码如下；

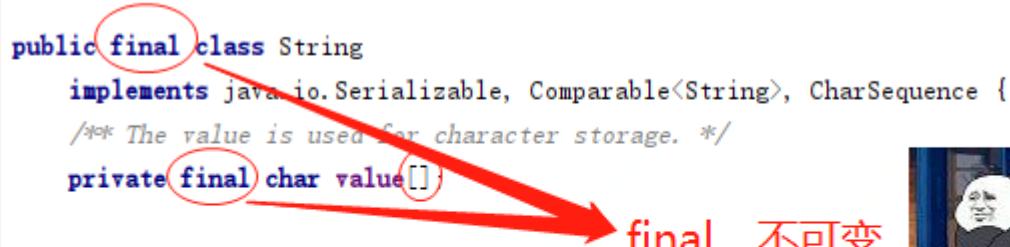
```

public char charAt(int index) {
    if ((index < 0) || (index >= value.length)) {
        throw new StringIndexOutOfBoundsException(index);
    }
    return value[index];
}

```

2. 不可变(final)

字符串创建后是不可变的，你看到的`+加号`连接操作，都是创建了新的对象把数据存放过去，通过源码就可以看到；



```

public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /* The value is used for character storage. */
    private final char value[];
    /* Cache the hash code for the string */
    private int hash; // Default to 0

    /* use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = -6849794470754667710L;
}

```



小傅哥 & String 不可变

从源码中可以看到，`String` 的类和用于存放字符串的方法都用了 `final` 修饰，也就是创建了以后，这些都是不可变的。

举个例子

```
String str_01 = "abc";
String str_02 = "abc" + "def";
String str_03 = str_01 + "def";
```

不考虑其他情况，对于程序初始化。以上这些代码 `str_01`、`str_02`、`str_03`，都会初始化几个对象呢？其实这个初始化几个对象从侧面就是反应用对象是否可变性。

接下来我们把上面代码反编译，通过指令码看到底创建了几个对象。

反编译下

```
public void test_00();
Code:
  0: ldc           #2                  // String abc
  2: astore_1
  3: ldc           #3                  // String abcdef
  5: astore_2
  6: new           #4                  // class java/lang/StringBuilder
  9: dup
 10: invokespecial #5                // Method java/lang/StringBuilder."<init>:()V"
 13: aload_1
 14: invokevirtual #6                // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
 17: ldc           #7                  // String def
 19: invokevirtual #6                // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
 22: invokevirtual #8                // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
 25: astore_3
 26: return
```

- `str_01 = "abc"`，指令码: `0: ldc`，创建了一个对象。
- `str_02 = "abc" + "def"`，指令码: `3: ldc // String abcdef`，得益于 JVM 编译期的优化，两个字符串会进行相连，创建一个对象存储。
- `str_03 = str_01 + "def"`，指令码: `invokevirtual`，这个就不一样了，它需要把两个字符串相连，会创建 `StringBuilder` 对象，直至最后 `toString:()` 操作，共创建了三个对象。

所以，我们看到，字符串的创建是不能被修改的，相连操作会创建出新对象。

3. intern()

3.1 经典题目

```
String str_1 = new String("ab");
String str_2 = new String("ab");
String str_3 = "ab";

System.out.println(str_1 == str_2);
System.out.println(str_1 == str_2.intern());
System.out.println(str_1.intern() == str_2.intern());
System.out.println(str_1 == str_3);
System.out.println(str_1.intern() == str_3);
```

这是一道经典的 `String` 字符串面试题，乍一看可能还会有点晕。答案如下：

```
false
false
true
false
true
```

3.2 源码分析

看了答案有点感觉了吗，其实可能你了解方法 `intern()`，这里先看下它的源码；

```
/*
 * Returns a canonical representation for the string object.
 * <p>
 * A pool of strings, initially empty, is maintained privately by the
 * class {@code String}.
 * <p>
 * When the intern method is invoked, if the pool already contains a
 * string equal to this {@code String} object as determined by
 * the {@link #equals(Object)} method, then the string from the pool is
 * returned. Otherwise, this {@code String} object is added to the
 * pool and a reference to this {@code String} object is returned.
 * <p>
 * It follows that for any two strings {@code s} and {@code t},
 * s.intern() == t
```

```

* {@code s.intern() == t.intern()} is {@code true}
* if and only if {@code s.equals(t)} is {@code true}.
* <p>
* All literal strings and string-valued constant expressions are
* interned. String literals are defined in section 3.10.5 of the
* <cite>The Java&trade; Language Specification</cite>.
*
* @return a string that has the same contents as this string, but is
*         guaranteed to be from a pool of unique strings.
*/
public native String intern();

```

这段代码和注释什么意思呢？

`native`, 说明 `intern()` 是一个本地方法, 底层通过 JNI 调用 C++语言编写的功能。

```

\openjdk8\jdk\src\share\native\java\lang\String.c

Java_java_lang_String_intern(JNIEnv *env, jobject this)
{
    return JVM_InternString(env, this);
}

oop result = StringTable::intern(string, CHECK_NULL);

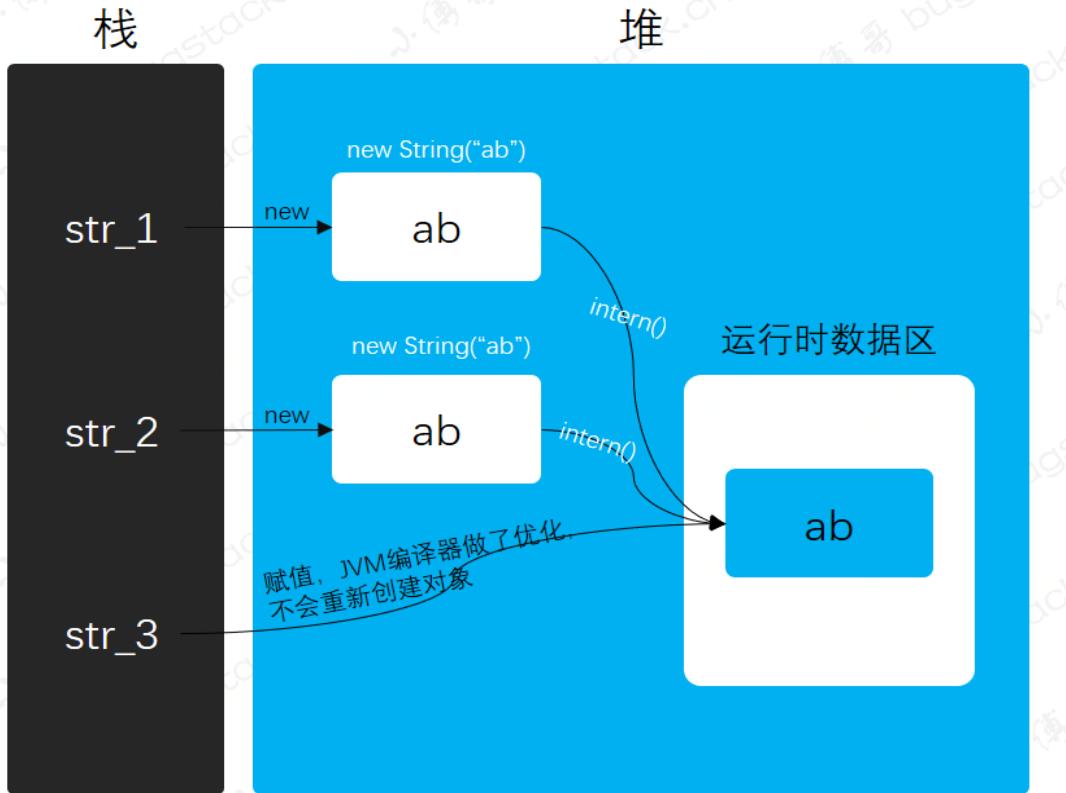
oop StringTable::intern(Handle string_or_null, jchar* name,
                       int len, TRAPS) {
    unsigned int hashValue = java_lang_String::hash_string(name, len);
    int index = the_table()->hash_to_index(hashValue);
    oop string = the_table()->lookup(index, name, len, hashValue);
    if (string != NULL) return string;
    return the_table()->basic_add(index, string_or_null, name, len,
                                   hashValue, CHECK_NULL);
}

```

- 代码块有点长这里只截取了部分内容, 源码可以学习开源 jdk 代码, 连接:
<https://codeload.github.com/abhijangda/OpenJDK8/zip/master>
- C++这段代码有点像 HashMap 的哈希桶+链表的数据结构, 用来存放字符串, 所以如果哈希值冲突严重, 就会导致链表过长。这在我们讲解 hashMap 中已经介绍, 可以回看 [HashMap 源码](#)

- StringTable 是一个固定长度的数组 1009 个大小, jdk1.6 不可调、jdk1.7 可以设置 `-XX:StringTableSize`, 按需调整。

3.3 问题图解



小傅哥 & 图解 true/false

看图说话, 如下;

- 先说 `==`, 基础类型比对的是值, 引用类型比对的是地址。另外, `equal` 比对的是哈希值。
- 两个 `new` 出来的对象, 地址肯定不同, 所以是 `false`。
- `intern()`, 直接把值推进了常量池, 所以两个对象都做了 `intern()` 操作后, 比对是常量池里的值。
- `str_3 = "ab"`, 赋值, JVM 编译器做了优化, 不会重新创建对象, 直接引用常量池里的值。所以 `str_1.intern() == str_3`, 比对结果是 `true`。

理解了这个结构, 根本不需要死记硬背应对面试, 让懂了就是真的懂, 大脑也会跟着愉悦。

四、StringBuilder 源码分析

1. 初始化

```
new StringBuilder();
new StringBuilder(16);
new StringBuilder("abc");
```

这几种方式都可以初始化，你可以传一个初始化容量，也可以初始化一个默认的字符串。它的源码如下；

```
public StringBuilder() {
    super(16);
}

AbstractStringBuilder(int capacity) {
    value = new char[capacity];
}
```

定睛一看，这就是在初始化数组呀！那是不操作起来跟使用 `ArrayList` 似的呀！

2. 添加元素

```
stringBuilder.append("a");
stringBuilder.append("b");
stringBuilder.append("c");
```

添加元素的操作很简单，使用 `append` 即可，那么它是怎么往数组中存放的呢，需要扩容吗？

2.1 入口方法

```
public AbstractStringBuilder append(String str) {
    if (str == null)
        return appendNull();
    int len = str.length();
    ensureCapacityInternal(count + len);
    str.getChars(0, len, value, count);
    count += len;
    return this;
}
```

- 这个是 `public final class StringBuilder extends AbstractStringBuilder`，的父类与 `StringBuffer` 共用这个方法。

- 这里包括了容量检测、元素拷贝、记录 `count` 数量。

2.2 扩容操作

```

ensureCapacityInternal(count + len);

/*
 * This method has the same contract as ensureCapacity, but is
 * never synchronized.
 */

private void ensureCapacityInternal(int minimumCapacity) {
    // overflow-conscious code
    if (minimumCapacity - value.length > 0)
        expandCapacity(minimumCapacity);
}

/*
 * This implements the expansion semantics of ensureCapacity with no
 * size check or synchronization.
 */

void expandCapacity(int minimumCapacity) {
    int newCapacity = value.length * 2 + 2;
    if (newCapacity - minimumCapacity < 0)
        newCapacity = minimumCapacity;
    if (newCapacity < 0) {
        if (minimumCapacity < 0) // overflow
            throw new OutOfMemoryError();
        newCapacity = Integer.MAX_VALUE;
    }
    value = Arrays.copyOf(value, newCapacity);
}

```

如上，`StringBuilder`，就跟操作数组的原理一样，都需要检测容量大小，按需扩容。扩容的容量是 $n * 2 + 2$ ，另外把原有元素拷贝到新新数组中。

2.3 填充元素

```

str.getChars(0, len, value, count);

public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin) {
    // ...

```

```
        System.arraycopy(value, srcBegin, dst, dstBegin, srcEnd - srcBegin);  
    }  
}
```

添加元素的方式是基于 `System.arraycopy` 拷贝操作进行的，这是一个本地方法。

2.4 `toString()`

既然 `StringBuilder` 是数组，那么它是怎么转换成字符串的呢？

```
StringBuilder.toString();
```

```
@Override  
public String toString() {  
    // Create a copy, don't share the array  
    return new String(value, 0, count);  
}
```

其实需要用到它是 `String` 字符串的时候，就是使用 `String` 的构造函数传递数组进行转换的，这个方法在我们上面讲解 `String` 的时候已经介绍过。

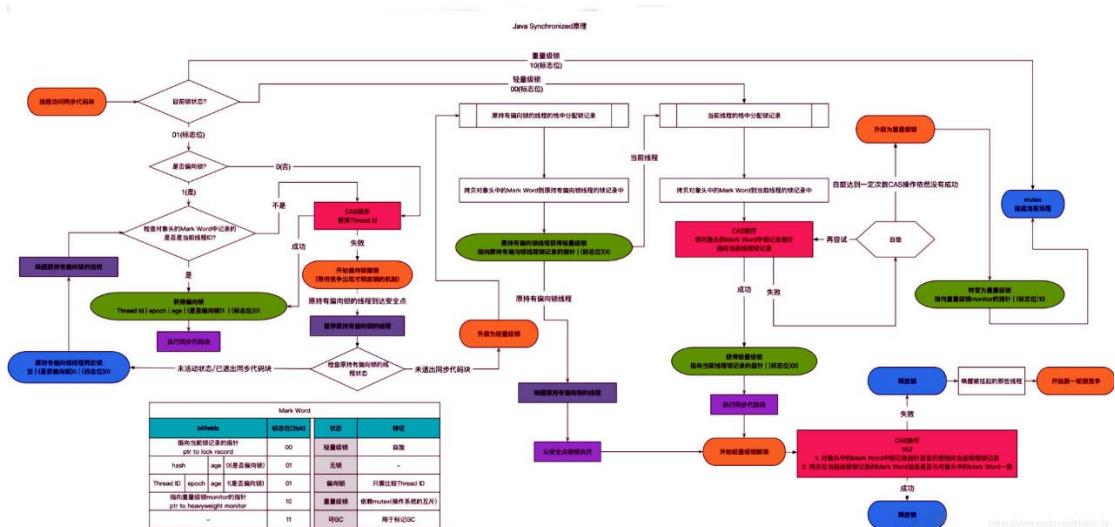
五、`StringBuffer` 源码分析

`StringBuffer` 与 `StringBuilder`，API 的使用和底层实现上基本一致，维度不同的是 `StringBuffer` 加了 `synchronized` 锁，所以它是线程安全的。源码如下；

```
@Override  
public synchronized StringBuffer append(String str) {  
    toStringCache = null;  
    super.append(str);  
    return this;  
}
```

那么，`synchronized` 不是重量级锁吗，JVM 对它有什么优化呢？

其实为了减少获得锁与释放锁带来的性能损耗，从而引入了偏向锁、轻量级锁、重量级锁来进行优化，它的进行一个锁升级，如下图(此图引自互联网用户：韭菜韭菜，画的非常优秀)；



小傅哥 & 此图引自互联网，画的非常漂亮

- 从无锁状态开始，当线程进入 `synchronized` 同步代码块，会检查对象头和栈帧内是否有当前线下 ID 编号，无则使用 `CAS` 替换。
- 解锁时，会使用 `CAS` 将 `Displaced Mark Word` 替换回到对象头，如果成功，则表示竞争没有发生，反之则表示当前锁存在竞争锁就会升级成重量级锁。
- 另外，大多数情况下锁是不发生竞争的，基本由一个线程持有。所以，为了避免获得锁与释放锁带来的性能损耗，所以引入锁升级，升级后不能降级。

六、常用 API

序号	方法	描述
1	<code>str.concat("cde")</code>	字符串连接，替换+号
2	<code>str.length()</code>	获取长度
3	<code>isEmpty()</code>	判空
4	<code>str.charAt(0)</code>	获取指定位置元素
5	<code>str.codePointAt(0)</code>	获取指定位置元素，并返回 ascii 码值
6	<code>str.getBytes()</code>	获取 byte[]
7	<code>str.equals("abc")</code>	比较
8	<code>str.equalsIgnoreCase("")</code>	忽略大小写，比对

序号	方法	描述
	AbC")	
9	str.startsWith("a")	开始位置值判断
10	str.endsWith("c")	结尾位置值判断
11	str.indexOf("b")	判断元素位置，开始位置
12	str.lastIndexOf("b")	判断元素位置，结尾位置
13	str.substring(0, 1)	截取
14	str.split(",")	拆分，可以支持正则
15	str.replace("a", "d") replaceAll	替换
16	str.toUpperCase()	转大写
17	str.toLowerCase()	转小写
18	str.toCharArray()	转数组
19	String.format(str, "")	格式化，%s、%c、%b、%d、%x、%o、%f、%a、%e、%g、%h、%%、%n、%tx
20	str.valueOf("123")	转字符串
21	trim()	格式化，首尾去空格
2	str.hashCode()	获取哈希值

序号	方法	描述
2		

七、总结

- 业精于勤，荒于嬉，你学到的知识不一定只是为了面试准备，还更应该是拓展自己的技术深度和广度。这个过程可能很痛苦，但总得需要某一个烧脑的过程，才让其他更多的知识学起来更加容易。
- 本文介绍了 `String`、`StringBuilder`、`StringBuffer`，的数据结构和源码分析，更加透彻的理解后，也能更加准确的使用，不会被因为不懂而犯错误。
- 想把代码写好，至少要有这四面内容，包括；数据结构、算法、源码、设计模式，这四方面在加上业务经验与个人视野，才能真的把一个需求、一个大项目写的具备良好的扩展性和易维护性。

第 11 节：ThreadLocal 源码分析

说到底，你真的会造火箭吗？

常说面试造火箭，入职拧螺丝。但你真的有造火箭的本事吗，大部分都是不敢承认自己的知识盲区和技术瓶颈以及经验不足的自嘲。

面试时：

- 我希望你懂数据结构，因为这样的你在使用 HashMap、ArrayList、LinkedList，更加得心应手。
- 我希望你懂散列算法，因为这样的你在设计路由时，会有很多选择；[除法散列法](#)、[平方散列法](#)、[斐波那契（Fibonacci）散列法](#)等。
- 我希望你懂开源代码，因为这样的你在遇到问题时，可以快速定位，还可能创造出一些系统服务的中间件，来更好的解耦系统。
- 我希望你懂设计模式，因为这样的你可以写出可扩展、易维护的程序，让整个团队都能向更好的方向发展。

所以，从不是 CRUD 选择了你，也不是造螺丝让你成为工具人。而是你的技术能力决定你的眼界，眼界又决定了你写出的代码！

一、面试题

谢飞机，小记 还没有拿到 offer 的飞机，早早起了床，吃完两根油条，又跑到公司找面试官取经！



灵魂画手 & 老纪

面试官：飞机，听坦克说，你最近贪黑起早的学习呀。

谢飞机：嗯嗯，是的，最近头发都快掉没了！

面试官：那今天我们聊聊 `ThreadLocal`，一般可以用在什么场景中？

谢飞机：嗯，`ThreadLocal` 要解决的是线程内资源共享 (*This class provides thread-local variables.*)，所以一般会用在全链路监控中，或者是像日志框架 `MDC` 这样的组件里。

面试官：飞机不错哈，最近确实学习了。那你知道 `ThreadLocal` 是怎样的数据结构吗，采用的是什么散列方式？

谢飞机：数组？嗯，怎么散列的不清楚...

面试官：那 `ThreadLocal` 有内存泄漏的风险，是怎么发生的呢？另外你了解在这个过程的，探测式清理和启发式清理吗？

谢飞机：这...，盲区了，盲区了，可乐我放桌上了，我回家再看看书！

二、`ThreadLocal` 分析

`ThreadLocal`，作者：[Josh Bloch](#) 和 [Doug Lea](#)，两位大神👉

如果仅是日常业务开发来看，这是一个比较冷门的类，使用频率并不高。并且它提供的方法也非常简单，一个功能只是潦潦数行代码。但，如果深挖实现部分的源码，就会发现事情并不那么简单。这里涉及了太多的知识点，包括：[数据结构](#)、[拉链存储](#)、[斐波那契散列](#)、[神奇的 0x61c88647](#)、[弱引用 Reference](#)、[过期 key 探测清理](#)和[启发式清理](#)等等。

接下来，我们就逐步学习这些盲区知识。本文涉及了较多的代码和实践验证图稿，欢迎关注公众号：[bugstack 虫洞栈](#)，回复下载得到一个链接打开后，找到 ID: 19 获取！*

1. 应用场景

1.1 `SimpleDateFormat`

```
private SimpleDateFormat f = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

public void seckillSku(){
    String dateStr = f.format(new Date());
    // 业务流程
}
```

你写过这样的代码吗？如果还在这么写，那就已经犯了一个线程安全的错误。`SimpleDateFormat`，并不是一个线程安全的类。

1.1.1 线程不安全验证

```
private static SimpleDateFormat f = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

public static void main(String[] args) {
    while (true) {
        new Thread(() -> {
            String dateStr = f.format(new Date());
            try {
                Date parseDate = f.parse(dateStr);
                String dateStrCheck = f.format(parseDate);
                boolean equals = dateStr.equals(dateStrCheck);
                if (!equals) {
                    System.out.println(equals + " " + dateStr + " " + dateStrCheck)
                }
            } else {
                System.out.println(equals);
            }
        }) catch (ParseException e) {
            System.out.println(e.getMessage());
        }
    }).start();
}
```

这是一个多线程下 `SimpleDateFormat` 的验证代码。当 `equals` 为 `false` 时，证明线程不安全。运行结果如下：

```
true
true
false 2020-09-23 11:40:42 2230-09-23 11:40:42
true
true
false 2020-09-23 11:40:42 2020-09-23 11:40:00
false 2020-09-23 11:40:42 2020-09-23 11:40:00
false 2020-09-23 11:40:00 2020-09-23 11:40:42
true
```

```
false 2020-09-23 11:40:42 2020-08-31 11:40:42  
true
```

1.1.2 使用 ThreadLocal 优化

为了线程安全最直接的方式，就是每次调用都直接 `new SimpleDateFormat`。但这样的方式终究不是最好的，所以我们使用 `ThreadLocal`，来优化这段代码。

```
private static ThreadLocal<SimpleDateFormat> threadLocal = ThreadLocal.withInitial(  
() -> new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));  
  
public static void main(String[] args) {  
  
    while (true) {  
  
        new Thread(() -> {  
  
            String dateStr = threadLocal.get().format(new Date());  
  
            try {  
  
                Date parseDate = threadLocal.get().parse(dateStr);  
  
                String dateStrCheck = threadLocal.get().format(parseDate);  
  
                boolean equals = dateStr.equals(dateStrCheck);  
  
                if (!equals) {  
  
                    System.out.println(equals + " " + dateStr + " " + dateStrCheck)  
;  
                } else {  
  
                    System.out.println(equals);  
                }  
            } catch (ParseException e) {  
  
                System.out.println(e.getMessage());  
            }  
        }).start();  
    }  
}
```

如上我们把 `SimpleDateFormat`，放到 `ThreadLocal` 中进行使用，即不需要重复 `new` 对象，也避免了线程不安全问题。测试结果如下；

```
true  
true  
true  
true  
true  
true
```

```
true
```

```
...
```

1.2 链路追踪

近几年基于[谷歌 Dapper](#) 论文实现非入侵全链路追踪，使用的越来越广了。简单说这就是一套监控系统，但不需要你硬编码的方式进行监控方法，而是基于它的设计方案采用 [javaagent + 字节码](#) 插桩的方式，动态采集方法执行信息。如果你想了解字节码插桩技术，可以阅读我的字节码编程专栏：
<https://bugstack.cn/itstack-demo-agent/itstack-demo-agent.html>

重点，动态采集方法执行信息。这块是主要部分，跟 [ThreadLocal](#) 相关。[字节码插桩](#)解决的是非入侵式编程，那么在一次服务调用时，在各个系统间以及系统内多个方法的调用，都需要进行采集。这个时候就需要使用 [ThreadLocal](#) 记录方法执行 ID，当然这里还有跨线程调用使用的也是增强版本的 [ThreadLocal](#)，但无论如何基本原理不变。

1.2.1 追踪代码

这里举例全链路方法调用链追踪，部分代码

```
public class TrackContext {  
  
    private static final ThreadLocal<String> trackLocal = new ThreadLocal<>();  
  
    public static void clear(){  
        trackLocal.remove();  
    }  
  
    public static String getLinkId(){  
        return trackLocal.get();  
    }  
  
    public static void setLinkId(String linkId){  
        trackLocal.set(linkId);  
    }  
}
```

```

@Advice.OnMethodEnter()
public static void enter(@Advice.Origin("#t") String className, @Advice.Origin("#m")
) String methodName) {
    Span currentSpan = TrackManager.getCurrentSpan();
    if (null == currentSpan) {
        String linkId = UUID.randomUUID().toString();
        TrackContext.setLinkId(linkId);
    }
    TrackManager.createEntrySpan();
}

@Advice.OnMethodExit()
public static void exit(@Advice.Origin("#t") String className, @Advice.Origin("#m")
String methodName) {
    Span exitSpan = TrackManager.getExitSpan();
    if (null == exitSpan) return;
    System.out.println("链路追踪(MQ):
" + exitSpan.getLinkId() + " " + className + "." + methodName + " 耗时:
" + (System.currentTimeMillis() - exitSpan.getEnterTime().getTime()) + "ms");
}

```

- 以上这部分就是非入侵监控中，链路追踪的过程。具体的案例和代码可以参考阅读，系列专题文章[《基于 JavaAgent 的全链路监控》](#)
- 这也只是其中一个实现方式，字节码插桩使用的是 `byte-buddy`，其实还是使用，`ASM` 或者 `Javassist`。

1.2.2 测试结果

测试方法

配置参数： `-javaagent:E:\itstack\GIT\itstack.org\itstack-demo-agent\itstack-demo-agent-06\target\itstack-demo-agent-06-1.0.0-SNAPSHOT.jar=testargs`

```

public void http_lt1(String name) {
    try {
        Thread.sleep((long) (Math.random() * 500));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("测试结果: hi1 " + name);
}

```

```

    http_lt2(name);
}

public void http_lt2(String name) {
    try {
        Thread.sleep((long) (Math.random() * 500));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("测试结果: hi2 " + name);
    http_lt3(name);
}

```

运行结果

```

onTransformation: class org.itstack.demo.test.ApiTest
测试结果: hi2 悟空
测试结果: hi3 悟空
链路追踪(MQ): 90c7d543-c7b8-4ec3-af4d-
b4d4f5cff760 org.itstack.demo.test.ApiTest.http_lt3 耗时: 104ms

init: 256MB max: 3614MB used: 44MB committed: 245MB use rate: 18%
init: 2MB max: 0MB used: 13MB committed: 14MB use rate: 95%

name: PS Scavenge count:0 took:0 pool name:[PS Eden Space, PS Survivor Space]
name: PS MarkSweep count:0 took:0 pool name:[PS Eden Space, PS Survivor Space, P
S Old Gen]

-----
-----
链路追踪(MQ): 90c7d543-c7b8-4ec3-af4d-
b4d4f5cff760 org.itstack.demo.test.ApiTest.http_lt2 耗时: 233ms

init: 256MB max: 3614MB used: 44MB committed: 245MB use rate: 18%
init: 2MB max: 0MB used: 13MB committed: 14MB use rate: 96%

name: PS Scavenge count:0 took:0 pool name:[PS Eden Space, PS Survivor Space]
name: PS MarkSweep count:0 took:0 pool name:[PS Eden Space, PS Survivor Space, P
S Old Gen]

```

- 以上是链路追踪的测试结果，可以看到两个方法都会打出相应的编码 ID：
`90c7d543-c7b8-4ec3-af4d-b4d4f5cff760`。
- 这部分也就是全链路追踪的核心应用，而且还可以看到这里打印了一些系统简单的 JVM 监控指标，这也是监控的一部分。

咳咳，除此之外所有需要活动方法调用链的，都需要使用到 `ThreadLocal`，例如 `MDC` 日志框架等。接下来我们开始详细分析 `ThreadLocal` 的实现。

2. 数据结构

了解一个功能前，先了解它的数据结构。这就相当于先看看它的地基，有了这个根本也就好往后理解了。以下是 `ThreadLocal` 的简单使用以及部分源码。

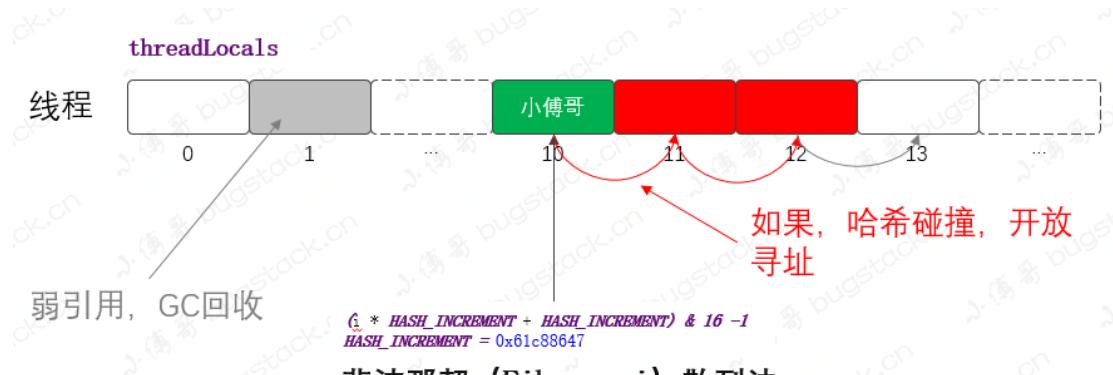
```
new ThreadLocal<String>().set("小傅哥");

private void set(ThreadLocal<?> key, Object value) {

    Entry[] tab = table;
    int len = tab.length;
    int i = key.threadLocalHashCode & (len-1);

    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
        ...
    }
}
```

从这部分源码中可以看到，`ThreadLocal` 底层采用的是数组结构存储数据，同时还有哈希值计算下标，这说明它是一个散列表的数组结构，演示如下图；



小傅哥 & `threadLocal` 数据结构

如上图是 `ThreadLocal` 存放数据的底层数据结构，包括知识点如下；

1. 它是一个数组结构。
2. `Entry`, 这里没用再打开, 其实它是一个弱引用实现, `static class Entry extends WeakReference<ThreadLocal<?>>`。这说明只要没用强引用存在, 发生 GC 时就会被垃圾回收。
3. 数据元素采用哈希散列方式进行存储, 不过这里的散列使用的是 [斐波那契 \(Fibonacci\) 散列法](#), 后面会具体分析。
4. 另外由于这里不同于 `HashMap` 的数据结构, 发生哈希碰撞不会存成链表或红黑树, 而是使用拉链法进行存储。也就是同一个下标位置发生冲突时, 则[+1 向后寻找地址](#), 直到找到空位置或垃圾回收位置进行存储。

3. 散列算法

既然 `ThreadLocal` 是基于数组结构的拉链法存储, 那就一定会有哈希的计算。但我们翻阅源码后, 发现这个哈希计算与 `HashMap` 中的散列求数组下标计算的哈希方式不一样。

3.1 神秘的数字 0x61c88647

当我们查看 `ThreadLocal` 执行设置元素时, 有这么一段计算哈希值的代码;

```
private static final int HASH_INCREMENT = 0x61c88647;

private static int nextHashCode() {
    return nextHashCode.getAndAdd(HASH_INCREMENT);
}
```

看到这里你一定会有这样的疑问, 这是什么方式计算哈希? 这个数字怎么来的? 讲到这里, 其实计算哈希的方式, 绝不止是我们平常看到 `String` 获取哈希值的一种方式, 还包括: [除法散列法](#)、[平方散列法](#)、[斐波那契 \(Fibonacci\) 散列法](#)、[随机数法](#)等。

而 `ThreadLocal` 使用的就是 [斐波那契 \(Fibonacci\) 散列法](#) + 拉链法存储数据到数组结构中。之所以使用斐波那契数列, 是为了让数据更加散列, 减少哈希碰撞。具体来自数学公式的计算求值, 公式: $f(k) = ((k * 2654435769) \gg X) \ll Y$ 对于常见的 32 位整数而言, 也就是 $f(k) = (k * 2654435769) \gg 28$

第二个问题，数字 `0x61c88647`，是怎么来的？

其实这是一个哈希值的黄金分割点，也就是 `0.618`，你还记得你学过的数学吗？计算方式如下：

```
// 黄金分割点: ( $\sqrt{5} - 1$ ) / 2 = 0.6180339887      1.618:1 == 1:0.618
System.out.println(BigDecimal.valueOf(Math.pow(2, 32) * 0.6180339887).intValue());
// -1640531527
```

- 学过数学都应该知道，黄金分割点是， $(\sqrt{5} - 1) / 2$ ，取 10 位近似 `0.6180339887`。
- 之后用 $2^{32} \times 0.6180339887$ ，得到的结果是：`-1640531527`，也就是 16 进制的，`0x61c88647`。这个数呢也就是这么来的

3.2 验证散列

既然，`Josh Bloch` 和 `Doug Lea`，两位老爷子选择使用斐波那契数列，计算哈希值。那一定有它的过人之处，也就是能更好的散列，减少哈希碰撞。

接下来我们按照源码中获取哈希值和计算下标的方式，把这部分代码提出出来做验证。

3.2.1 部分源码

```
private static AtomicInteger nextHashCode = new AtomicInteger();

private static final int HASH_INCREMENT = 0x61c88647;

// 计算哈希
private static int nextHashCode() {
    return nextHashCode.getAndAdd(HASH_INCREMENT);
}

// 获取下标
int i = key.threadLocalHashCode & (len-1);
```

如上，源码部分采用的是 `AtomicInteger`，原子方法计算下标。我们不需要保证线程安全，只需要简单实现即可。另外 `ThreadLocal` 初始化数组长度是 16，我们也初始化这个长度。

3.2.2 单元测试

```
@Test  
public void test_idx() {  
    int hashCode = 0;  
    for (int i = 0; i < 16; i++) {  
        hashCode = i * HASH_INCREMENT + HASH_INCREMENT;  
        int idx = hashCode & 15;  
        System.out.println("斐波那契散列: " + idx + " 普通散列:  
" + (String.valueOf(i).hashCode() & 15));  
    }  
}
```

测试代码部分，采用的就是斐波那契数列，同时我们加入普通哈希算法进行比对散列效果。当然 `String` 这个哈希并没有像 `HashMap` 中进行扰动

测试结果：

```
斐波那契散列: 7 普通散列: 0  
斐波那契散列: 14 普通散列: 1  
斐波那契散列: 5 普通散列: 2  
斐波那契散列: 12 普通散列: 3  
斐波那契散列: 3 普通散列: 4  
斐波那契散列: 10 普通散列: 5  
斐波那契散列: 1 普通散列: 6  
斐波那契散列: 8 普通散列: 7  
斐波那契散列: 15 普通散列: 8  
斐波那契散列: 6 普通散列: 9  
斐波那契散列: 13 普通散列: 15  
斐波那契散列: 4 普通散列: 0  
斐波那契散列: 11 普通散列: 1  
斐波那契散列: 2 普通散列: 2  
斐波那契散列: 9 普通散列: 3  
斐波那契散列: 0 普通散列: 4
```

```
Process finished with exit code 0
```

发现没？，斐波那契散列的非常均匀，普通散列到 15 个以后已经发生碰撞。这也就是斐波那契散列的魅力，减少碰撞也就可以让数据存储的更加分散，获取数据的时间复杂度基本保持在 $O(1)$ 。

4. 源码解读

4.1 初始化

```
new ThreadLocal<>()
```

初始化的过程也很简单，可以按照自己需要的泛型进行设置。但在 `ThreadLocal` 的源码中有一点非常重要，就是获取 `threadLocal` 的哈希值的获取，`threadLocalHashCode`。

```
private final int threadLocalHashCode = nextHashCode();

/**
 * Returns the next hash code.
 */
private static int nextHashCode() {
    return nextHashCode.getAndAdd(HASH_INCREMENT);
}
```

如源码中，只要实例化一个 `ThreadLocal`，就会获取一个相应的哈希值，则例我们做一个例子。

```
@Test
public void test_threadLocalHashCode() throws Exception {
    for (int i = 0; i < 5; i++) {
        ThreadLocal<Object> objectThreadLocal = new ThreadLocal<>();
        Field threadLocalHashCode = objectThreadLocal.getClass().getDeclaredField("threadLocalHashCode");
        threadLocalHashCode.setAccessible(true);
        System.out.println("objectThreadLocal:
" + threadLocalHashCode.get(objectThreadLocal));
    }
}
```

因为 `threadLocalHashCode`，是一个私有属性，所以我们实例化后通过上面的方式进行获取哈希值。

```
objectThreadLocal: -1401181199
objectThreadLocal: 239350328
objectThreadLocal: 1879881855
objectThreadLocal: -774553914
objectThreadLocal: 865977613
```

```
Process finished with exit code 0
```

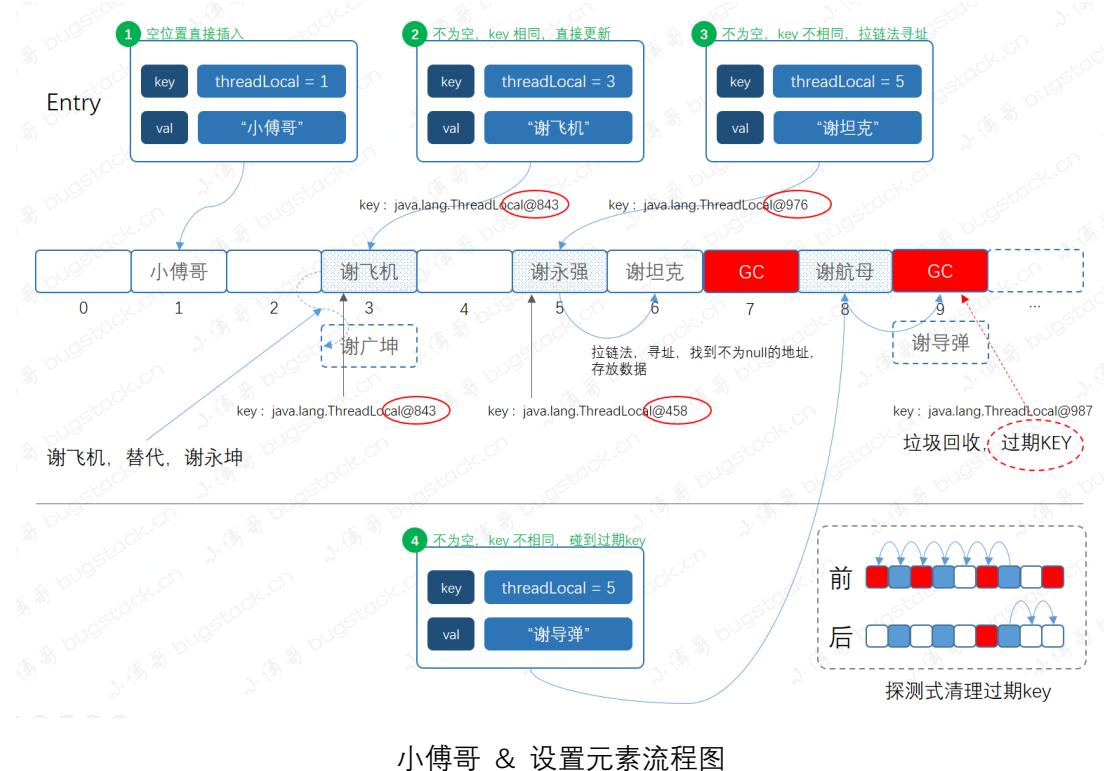
这个值的获取，也就是计算 `ThreadLocalMap`，存储数据时，`ThreadLocal` 的数组下标。只要是这同一个对象，在 `set`、`get` 时，就可以设置和获取对应的值。

4.2 设置元素

4.2.1 流程图解

```
new ThreadLocal<>().set("小傅哥");
```

设置元素的方法，也就这么一句代码。但设置元素的流程却涉及的比较多，在详细分析代码前，我们先来看一张设置元素的流程图，从图中先了解不同情况的流程之后再对比着学习源码。流程图如下：



小傅哥 & 设置元素流程图

乍一看可能感觉有点晕，我们从左往右看，分别有如下知识点； 0. 中间是 `ThreadLocal` 的数组结构，之后在设置元素时分为四种不同的情况，另外元素的插入是通过斐波那契散列计算下标值，进行存放的。

1. 情况 1，待插入的下标，是空位置直接插入。
2. 情况 2，待插入的下标，不为空，key 相同，直接更新
3. 情况 3，待插入的下标，不为空，key 不相同，拉链法寻址

4. 情况 4, 不为空, key 不相同, 碰到过期 key。其实情况 4, 遇到的是弱引用发生 GC 时, 产生的情况。碰到这种情况, `ThreadLocal` 会进行探测清理过期 key, 这部分清理内容后续讲解。

4.2.2 源码分析

```
private void set(ThreadLocal<?> key, Object value) {
    Entry[] tab = table;
    int len = tab.length;
    int i = key.threadLocalHashCode & (len-1);
    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
        ThreadLocal<?> k = e.get();
        if (k == key) {
            e.value = value;
            return;
        }
        if (k == null) {
            replaceStaleEntry(key, value, i);
            return;
        }
    }
    tab[i] = new Entry(key, value);
    int sz = ++size;
    if (!cleanSomeSlots(i, sz) && sz >= threshold)
        rehash();
}
```

在有了上面的图解流程, 再看代码部分就比较容易理解了, 与之对应的内容包括, 如下:

1. `key.threadLocalHashCode & (len-1);`, 斐波那契散列, 计算数组下标。
2. `Entry`, 是一个弱引用对象的实现类, `static class Entry extends WeakReference<ThreadLocal<?>>`, 所以在没有外部强引用下, 会发生 GC, 删除 key。
3. for 循环判断元素是否存在, 当前下标不存在元素时, 直接设置元素 `tab[i] = new Entry(key, value);`。
4. 如果元素存在, 则会判断是否 key 值相等 `if (k == key)`, 相等则更新值。

- 如果不相等，就到了我们的 `replaceStaleEntry`，也就是上图说到的探测式清理过期元素。

综上，就是元素存放的全部过程，整体结构的设计方式非常赞，极大的利用了散列效果，也把弱引用使用的非常 6！

4.3 扩容机制

4.3.1 扩容条件

只要使用到数组结构，就一定会有扩容

```
if (!cleanSomeSlots(i, sz) && sz >= threshold)
    rehash();
```

在我们阅读设置元素时，有以上这么一块代码，判断是否扩容。

- 首先，进行启发式清理`*cleanSomeSlots*`，把过期元素清理掉，看空间是否
- 之后，判断 `sz >= threshold`，其中 `threshold = len * 2 / 3`，也就是说数组中未填充的元素，大于 `len * 2 / 3`，就需要扩容了。
- 最后，就是我们要分析的重点，`rehash()`，扩容重新计算元素位置。

4.3.2 源码分析

探测式清理和校验

```
private void rehash() {
    expungeStaleEntries();

    // Use lower threshold for doubling to avoid hysteresis
    if (size >= threshold - threshold / 4)
        resize();
}

private void expungeStaleEntries() {
    Entry[] tab = table;
    int len = tab.length;
    for (int j = 0; j < len; j++) {
        Entry e = tab[j];
        if (e != null && e.get() == null)
            expungeStaleEntry(j);
    }
}
```

```
    }
}
```

- 这部分主要是探测式清理过期元素，以及判断清理后是否满足扩容条件， $\text{size} \geq \text{threshold} * 3/4$
- 满足后执行扩容操作，其实扩容完的核心操作就是重新计算哈希值，把元素填充到新的数组中。

rehash() 扩容

```
private void resize() {
    Entry[] oldTab = table;
    int oldLen = oldTab.length;
    int newLen = oldLen * 2;
    Entry[] newTab = new Entry[newLen];
    int count = 0;
    for (int j = 0; j < oldLen; ++j) {
        Entry e = oldTab[j];
        if (e != null) {
            ThreadLocal<?> k = e.get();
            if (k == null) {
                e.value = null; // Help the GC
            } else {
                int h = k.threadLocalHashCode & (newLen - 1);
                while (newTab[h] != null)
                    h = nextIndex(h, newLen);
                newTab[h] = e;
                count++;
            }
        }
    }
    setThreshold(newLen);
    size = count;
    table = newTab;
}
```

以上，代码就是扩容的整体操作，具体包括如下步骤：

- 首先把数组长度扩容到原来的 2 倍， $\text{oldLen} * 2$ ，实例化新数组。
- 遍历 for，所有的旧数组中的元素，重新放到新数组中。

3. 在放置数组的过程中，如果发生哈希碰撞，则链式法顺延。
4. 同时这还有检测 key 值的操作 `if (k == null)`，方便 GC。

4.4 获取元素

4.4.1 流程图解

```
new ThreadLocal<>().get();
```

同样获取元素也就这么一句代码，如果没有分析源码之前，你能考虑到它在不同的数据结构下，获取元素时候都做了什么操作吗。我们先来看下图，分为如下种情况：



小傅哥 & 获取元素图解

按照不同的数据元素存储情况，基本包括如下情况；

1. 直接定位到，没有哈希冲突，直接返回元素即可。
2. 没有直接定位到了，key 不同，需要拉链式寻找。
3. 没有直接定位到了，key 不同，拉链式寻找，遇到 GC 清理元素，需要探测式清理，再寻找元素。

4.4.2 源码分析

```
private Entry getEntry(ThreadLocal<?> key) {
    int i = key.threadLocalHashCode & (table.length - 1);
    Entry e = table[i];
    if (e != null && e.get() == key)
        return e;
    else
        return getEntryAfterMiss(key, i, e);
```

```

    }

private Entry getEntryAfterMiss(ThreadLocal<?> key, int i, Entry e) {
    Entry[] tab = table;
    int len = tab.length;
    while (e != null) {
        ThreadLocal<?> k = e.get();
        if (k == key)
            return e;
        if (k == null)
            expungeStaleEntry(i);
        else
            i = nextIndex(i, len);
        e = tab[i];
    }
    return null;
}

```

好了，这部分就是获取元素的源码部分，和我们图中列举的情况是一致的。
`expungeStaleEntry`，是发现有 `key == null` 时，进行清理过期元素，并把后续位置的元素，前移。

4.5 元素清理

4.5.1 探测式清理[`expungeStaleEntry`]

探测式清理，是以当前遇到的 GC 元素开始，向后不断的清理。直到遇到 `null` 为止，才停止 rehash 计算 `Rehash until we encounter null`。
`expungeStaleEntry`

```

private int expungeStaleEntry(int staleSlot) {
    Entry[] tab = table;
    int len = tab.length;
    // expunge entry at staleSlot
    tab[staleSlot].value = null;
    tab[staleSlot] = null;
    size--;
    // Rehash until we encounter null
    Entry e;
    int i;
    for (i = nextIndex(staleSlot, len);

```

```

        (e = tab[i]) != null;
        i = nextIndex(i, len)) {
    ThreadLocal<?> k = e.get();
    if (k == null) {
        e.value = null;
        tab[i] = null;
        size--;
    } else {
        int h = k.threadLocalHashCode & (len - 1);
        if (h != i) {
            tab[i] = null;
            // Unlike Knuth 6.4 Algorithm R, we must scan until
            // null because multiple entries could have been stale.
            while (tab[h] != null)
                h = nextIndex(h, len);
            tab[h] = e;
        }
    }
}
return i;
}

```

以上，探测式清理在获取元素中使用到； new ThreadLocal<()>.get() -> map.getEntry(this) -> getEntryAfterMiss(key, i, e) -> expungeStaleEntry(i)

4.5.2 启发式清理[cleanSomeSlots]

Heuristically scan some cells looking for stale entries.
This is invoked when either a new element is added, or
another stale one has been expunged. It performs a
logarithmic number of scans, as a balance between no
scanning (fast but retains garbage) and a number of scans
proportional to number of elements, that would find all
garbage but would cause some insertions to take O(n) time.

启发式清理，有这么一段注释，大概意思是：试探的扫描一些单元格，寻找过期元素，也就是被垃圾回收的元素。当添加新元素或删除另一个过时元素时，将调用此函数。它执行对数扫描次数，作为不扫描（快速但保留垃圾）和与元素数量成比例的扫描次数之间的平衡，这将找到所有垃圾，但会导致一些插入花费 $O(n)$ 时间。

```

private boolean cleanSomeSlots(int i, int n) {
    boolean removed = false;
    Entry[] tab = table;
    int len = tab.length;
    do {
        i = nextIndex(i, len);
        Entry e = tab[i];
        if (e != null && e.get() == null) {
            n = len;
            removed = true;
            i = expungeStaleEntry(i);
        }
    } while ((n >>>= 1) != 0);
    return removed;
}

```

while 循环中不断的右移进行寻找需要被清理的过期元素，最终都会使用 `expungeStaleEntry` 进行处理，这里还包括元素的移位。

三、总结

- 写到这算是把 `ThreadLocal` 知识点的一角分析完了，在 `ThreadLocal` 的家族里还有 `Netty` 中用到的，`FastThreadLocal`。在全链路跨服务线程间获取调用链路，还有 `TransmittableThreadLocal`，另外还有 JDK 本身自带的一种线程传递解决方案 `InheritableThreadLocal`。但站在本文的基础上，了解了最基础的原理，在理解其他的拓展设计，就更容易接受了。
- 此外在我们文中分析时经常会看到探测式清理，其实这也是非常耗时。为此我们在使用 `ThreadLocal` 一定要记得 `new ThreadLocal<>().remove();` 操作。避免弱引用发生 GC 后，导致内存泄漏的问题。
- 最后，你发现了吗！** 我们学习这样的底层原理性知识，都离不开数据结构和良好的设计方案，或者说是算法的身影。这些代码才是支撑整个系统良好运行的地基，如果我们可以把一些思路抽取到我们开发的核心业务流程中，也是可以大大提升性能的。

第 3 章 码农会锁(5 节)

第 1 节: volatile



你是个能吃苦的人吗？

从前的能吃苦大多指的体力劳动的苦，但现在的能吃苦已经包括太多维度，包括：
读书学习&寂寞的苦、深度思考&脑力的苦、自律习惯&修行的苦、自控能力&放弃的苦、低头做人&尊严的苦。

虽然这些苦摆在眼前，但大多数人还是喜欢吃简单的苦。熬夜加班、日复一日、重复昨天、CRUD，最后身体发胖、体质下降、能力不足、自抱自泣！所以有些苦能不吃就不吃，要吃就吃那些有成长价值的苦。

今天你写博客了吗？

如果一件小事能坚持 5 年以上，那你一定是很了不起的人。是的，很了不起。人最难的就是想清楚了但做不到，或者偶尔做到长期做不到。

其实大多数走在研发路上的伙伴们，都知道自己该努力，但明明下好了的决心就是坚持不了多久。就像你是否也想过要写技术博客，做技术积累。直到有一天被瓶颈限制在困局中才会着急，但这时候在想破局就真的很难了！

一、面试题

谢飞机，小记，飞机趁着周末，吃完火锅。又去约面试官喝茶了！

谢飞机：嗨，我在这，这边，这边。

面试官：你怎么又来了，最近学的不错了？

谢飞机：还是想来大厂，别害羞，面我吧！

面试官：我好像是你补课老师... 既然来了，就问问你吧！`volatile` 是干啥的？

谢飞机：啊，`volatile` 是保证变量对所有线程的可见性的。

面试官：那 `volatile` 可以解决原子性问题吗？

谢飞机：不可以！

面试官：那 `volatile` 的底层原理是如何实现的呢？

谢飞机：...，这！面试官，刚问两个题就甩雷，你是不家里有事要忙？

面试官：你管我！

二、`volatile` 讲解

1. 可见性案例

```
public class ApiTest {

    public static void main(String[] args) {
        final VT vt = new VT();

        Thread Thread01 = new Thread(vt);
        Thread Thread02 = new Thread(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException ignore) {
                }
                vt.sign = true;
                System.out.println("vt.sign = true 通知 while (!sign) 结束！");
            }
        });
        Thread01.start();
        Thread02.start();
    }

    class VT implements Runnable {
```

```
public boolean sign = false;

public void run() {
    while (!sign) {
    }
    System.out.println("你坏");
}
}
```

这段代码，是两个线程操作一个变量，程序期望当 `sign` 在线程 Thread01 被操作 `vt.sign = true` 时，Thread02 输出 `你坏`。

但实际上这段代码永远不会输出 `你坏`，而是一直处于死循环。这是为什么呢？接下来我们就一步步讲解和验证。

2. 加上 volatile 关键字

我们把 `sign` 关键字加上 `volatile` 描述，如下：

```
class VT implements Runnable {

    public volatile boolean sign = false;

    public void run() {
        while (!sign) {
        }
        System.out.println("你坏");
    }
}
```

测试结果

```
vt.sign = true 通知 while (!sign) 结束!
你坏
```

```
Process finished with exit code 0
```

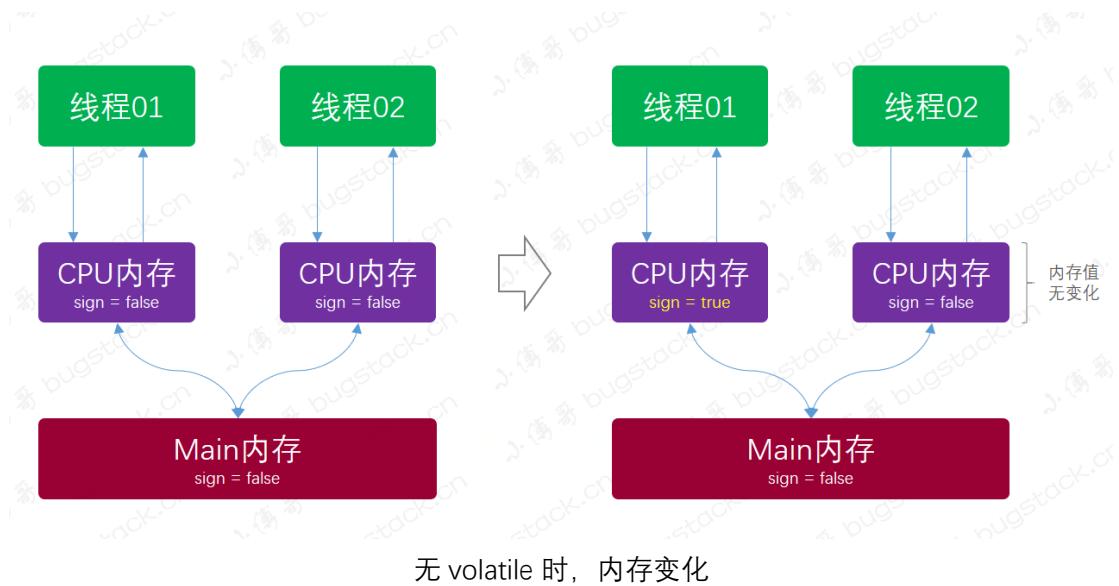
`volatile` 关键字是 Java 虚拟机提供的的最轻量级的同步机制，它作为一个修饰符出现，用来修饰变量，但是这里不包括局部变量哦

在添加 `volatile` 关键字后，程序就符合预期的输出了 `你坏`。从我们对 `volatile` 的学习认知可以知道。`volatile` 关键字是 JVM 提供的最轻量级的同步机制，用来修饰变量，用来保证变量对所有线程可见性。

正在修饰后可以让字段在线程见可见，那么这个属性被修改值后，可以及时的在另外的线程中做出相应的反应。

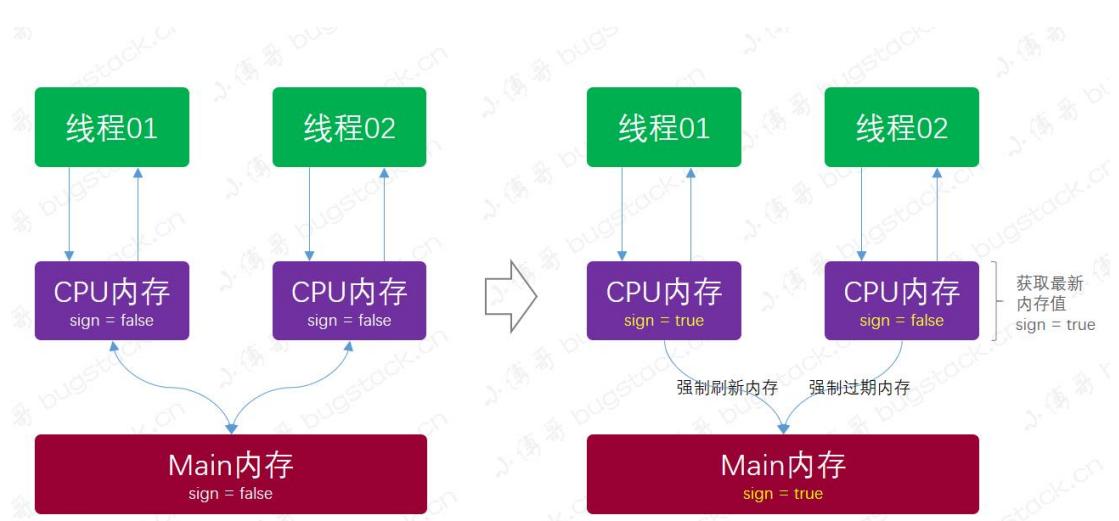
3. volatile 怎么保证的可见性

3.1 无 volatile 时，内存变化



首先是当 `sign` 没有 `volatile` 修饰时 `public boolean sign = false;`，线程 01 对变量进行操作，线程 02 并不会拿到变化的值。所以程序也就不会输出结果“你坏”

3.2 有 volatile 时，内存变化



当我们把变量使用 `volatile` 修饰时 `public volatile boolean sign = false;`，线程 01 对变量进行操作时，会把变量变化的值强制刷新的到主内存。当线程 02

获取值时，会把自己的内存里的 sign 值过期掉，之后从主内存中读取。所以添加关键字后程序如预期输出结果。

4. 反编译解毒可见性

类似这样有深度的技术知识，最佳的方式就是深入理解原理，看看它到底做了什么才保证的内存可见性操作。

4.1 查看 JVM 指令

指令: `javap -v -p VT`

```
public volatile boolean sign;
descriptor: Z
flags: ACC_PUBLIC, ACC_VOLATILE

org.itstack.interview.test.VT();
descriptor: ()V
flags:
Code:
  stack=2, locals=1, args_size=1
    0: aload_0
    1: invokespecial #1                  // Method java/lang/Object.<init>():
)V
    4: aload_0
    5: iconst_0
    6: putfield      #2                  // Field sign:Z
    9: return
LineNumberTable:
  line 35: 0
  line 37: 4
LocalVariableTable:
  Start  Length  Slot  Name   Signature
    0       10      0  this   Lorg/itstack/interview/test/VT;

public void run();
descriptor: ()V
flags: ACC_PUBLIC
Code:
  stack=2, locals=1, args_size=1
```

```

    0: aload_0
    1: getfield      #2                  // Field sign:Z
    4: ifne          10
    7: goto          0
   10: getstatic     #3                  // Field java/lang/System.out:Ljava/i
o/PrintStream;
   13: ldc           #4                  // String 你坏
   15: invokevirtual #5                  // Method java/io/PrintStream.println
:(Ljava/lang/String;)V
   18: return

LineNumberTable:
  line 40: 0
  line 42: 10
  line 43: 18

LocalVariableTable:
  Start  Length  Slot  Name   Signature
    0       19      0  this   Long/itstack/interview/test/VT;

StackMapTable: number_of_entries = 2
  frame_type = 0 /* same */
  frame_type = 9 /* same */

}

```

从 JVM 指令码中只会发现多了，`ACC_VOLATILE`，并没有什么其他的点。所以，也不能看出是怎么实现的可见性。

4.2 查看汇编指令

通过 Class 文件查看汇编，需要下载 `hsdis-amd64.dll` 文件，复制到 `JAVA_HOME\jre\bin\server` 目录下。下载资源如下：

- <http://vorboss.dl.sourceforge.net/project/fcml/fcml-1.1.1/hsdis-1.1.1-win32-amd64.zip>
- <http://vorboss.dl.sourceforge.net/project/fcml/fcml-1.1.1/hsdis-1.1.1-win32-i386.zip>

另外是执行命令，包括：

1. 基础指令：`java -Xcomp -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly`
2. 指定打印：`-XX:CompileCommand=dontinline,类名.方法名`

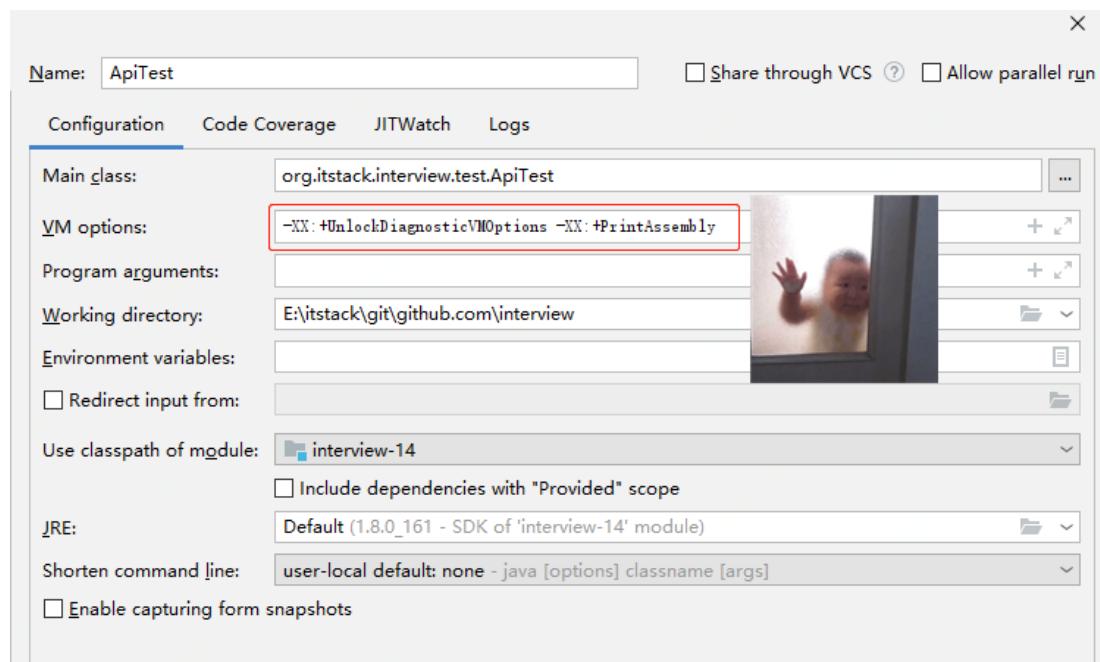
3. 指定打印: `-XX:CompileCommand=compileonly,类名.方法名`

4. 输出位置: > `xxx`

最终使用: `java -Xcomp -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -XX:CompileCommand=dontinline,ApiTest.main -XX:CompileCommand=compileonly,ApiTest.mian`

指令可以在 IDEA 中的 Terminal 里使用, 也可以到 DOS 黑窗口中使用

另外, 为了更简单的使用, 我们把指令可以配置到 idea 的 VM options 里, 如下图:



Idea VM options 配置编译指令

配置完成后, 不出意外的运行结果如下:

```
Loaded disassembler from C:\Program Files\Java\jdk1.8.0_161\jre\bin\server\hsdis-amd64.dll
Decoding compiled method 0x0000000003744990:
Code:
Argument 0 is unknown.RIP: 0x3744ae0 Code size: 0x00000110
[Disassembling for mach='amd64']
[Entry Point]
[Constants]
# {method} {0x000000001c853d18} 'getSnapshotTransformerList' '()' [Lsun/instrument/TransformerManager$TransformerInfo; in 'sun/instrument/TransformerManager'
#           [sp+0x40] (sp of caller)
0x0000000003744ae0: mov     r10d,dword ptr [rdx+8h]
0x0000000003744ae4: shl     r10,3h
```

```

0x0000000003744ae8: cmp      r10,rax
0x0000000003744aeb: jne      3685f60h          ; {runtime_call}
0x0000000003744af1: nop      word ptr [rax+rax+0h]
0x0000000003744afc: nop

[Verified Entry Point]

0x0000000003744b00: mov      dword ptr [rsp+0xfffffffffffffa000h],eax
0x0000000003744b07: push     rbp
0x0000000003744b08: sub      rsp,30h          ;*aload_0
                                                ; - sun.instrument.TransformerManag
er::getSnapshotTransformerList@0 (line 166)

0x0000000003744b0c: mov      eax,dword ptr [rdx+10h]
0x0000000003744b0f: shl      rax,3h          ;*getfield mTransformerList
                                                ; - sun.instrument.TransformerManag
er::getSnapshotTransformerList@1 (line 166)

0x0000000003744b13: add      rsp,30h
...

```

运行结果就是汇编指令，比较多这里就不都放了。我们只观察▲重点部分：

```

0x0000000003324cda: mov      0x74(%r8),%edx      ;*getstatic state
                                                ; - VT::run@28 (line 27)

0x0000000003324cde: inc      %edx
0x0000000003324ce0: mov      %edx,0x74(%r8)
0x0000000003324ce4: lock    addl $0x0,(%rsp)      ;*putstatic state
                                                ; - VT::run@33 (line 27)

```

编译后的汇编指令中，有 volatile 关键字和没有 volatile 关键字，主要差别在于多了一个 `lock addl $0x0,(%rsp)`，也就是 lock 的前缀指令。

lock 指令相当于一个内存屏障，它保证如下三点：

1. 将本处理器的缓存写入内存。
2. 重排序时不能把后面的指令重排序到内存屏障之前的位置。
3. 如果是写入动作会导致其他处理器中对应的内存无效。

那么，这里的 1、3 就是用来保证被修饰的变量，保证内存可见性。

5. 不加 volatile 也可见吗

有质疑就要有验证

我们现在再把例子修改下，在 `while (!sign)` 循环体中添加一段执行代码，如下：

```
class VT implements Runnable {  
  
    public boolean sign = false;  
  
    public void run() {  
        while (!sign) {  
            System.out.println("你好");  
        }  
        System.out.println("你坏");  
    }  
  
}
```

修改后去掉了 `volatile` 关键字，并在 `while` 循环中添加一段代码。现在的运行结果是：

```
...  
你好  
你好  
你好  
vt.sign = true 通知 while (!sign) 结束!  
你坏
```

```
Process finished with exit code 0
```

咋样，又可见了吧！

这是因为在没 `volatile` 修饰时，jvm 也会尽量保证可见性。有 `volatile` 修饰的时候，一定保证可见性。但可能并非如此，下章节继续深挖！

三、总结

- 最后我们再总结下 `volatile`，它呢，会控制被修饰的变量在内存操作上主动把值刷新到主内存，JMM 会把该线程对应的 CPU 内存设置过期，从主内存中读取最新值。
- 那么，`volatile` 如何防止指令重排也是内存屏障，`volatile` 的内存屏障是在读写操作的前后各添加一个 `StoreStore` 屏障，也就是四个位置，来保证重排序时不能把内存屏障后面的指令重排序到内存屏障之前的位置。

- 另外 volatile 并不能解决原子性，如果需要解决原子性问题，需要使用 synchronized 或者 lock，这部分内容在我们后续章节中介绍。

第 2 节：synchronized

感觉什么都不会，从哪开始呀！

这是最近我总能被问到的问题，也确实是。一个初入编程职场的新人，或是一个想重新努力学习的老司机，这也不会，那也不会，总会犯愁从哪开始。

讲道理，毕竟 Java 涉及的知识太多了，要学应该是学会学习的能力，而不是去背题、背答案，拾人牙慧是不会有太多收益的。

学习的过程要找对方法，遇到问题时最好能自己想想，你有哪些方式学会这些知识。是不感觉即使让你去百度搜，你都不知道应该拿哪个关键字搜！只能拿着问题直接找人问，这样缺少思考，缺少大脑撞南墙的过程，其实最后也很难学会。所以，你要学会的是自我学习的能力，之后是从哪开始都可以，重要的是开始和坚持！

一、面试题

谢飞机，小记，周末逛完奥特莱斯，回来就跑面试官家去了！

谢飞机：duang、duang、duang，我来了！

面试官：来的还挺准时，洗洗手吃饭吧！

谢飞机：嘿嘿...

面试官：你看我这块鱼豆腐，像不像 synchronized 锁！

谢飞机：啊！？

面试官：飞机，正好问你。synchronized、volatile，有什么区别呀？

谢飞机：嗯，volatile 保证可见性，synchronized 保证原子性！

面试官：那不用 volatile，只用 synchronized 修饰方式，能保证可见性吗？

谢飞机：这...，我没验证过！

面试官：吃吧，吃吧！一会给你个 synchronized 学习大纲，照着整理知识点！

二、synchronized 解毒

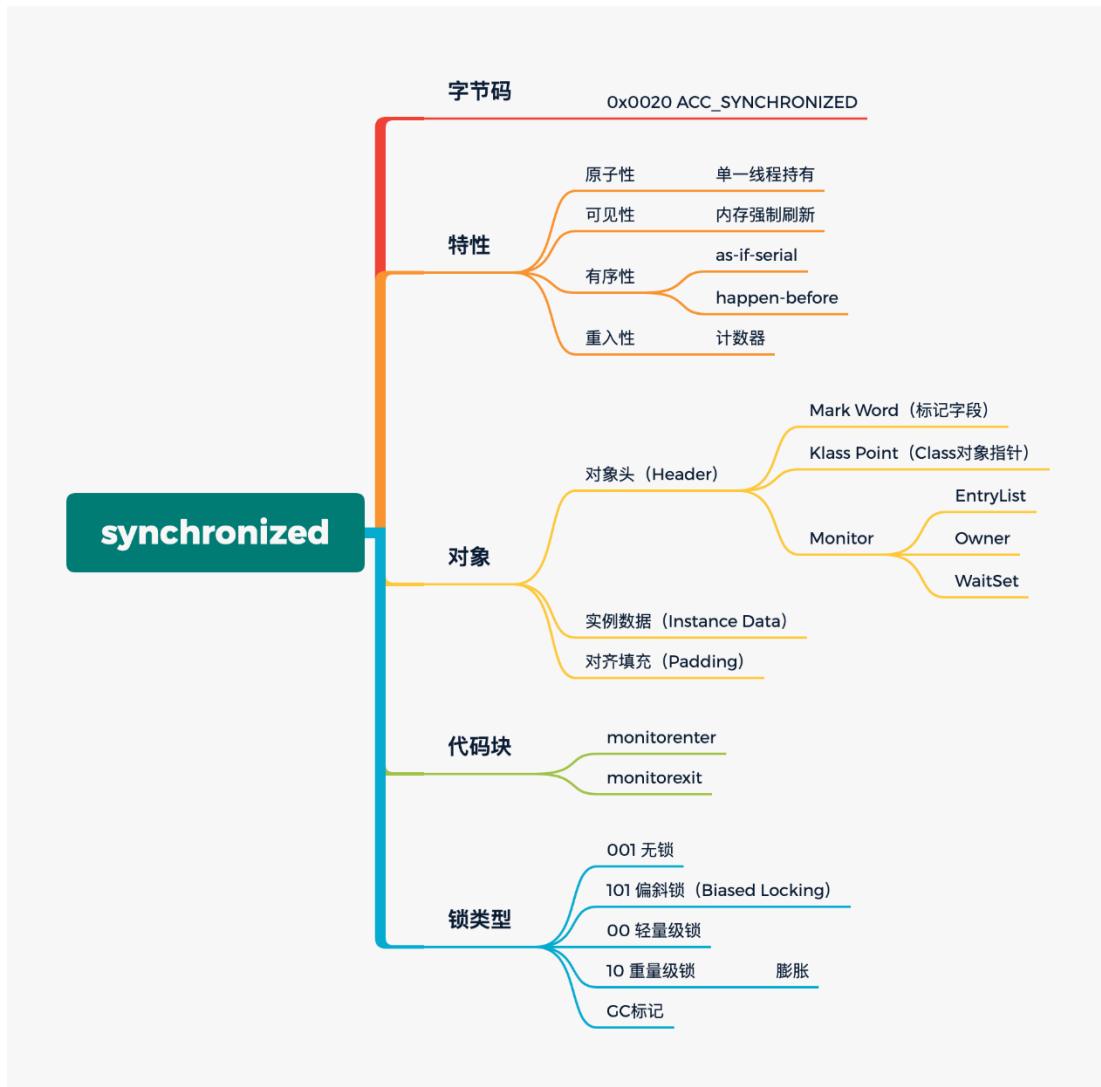


图 15-0 面试官给谢飞机的, synchronized 学习大纲

1. 对象结构

1.1 对象结构介绍

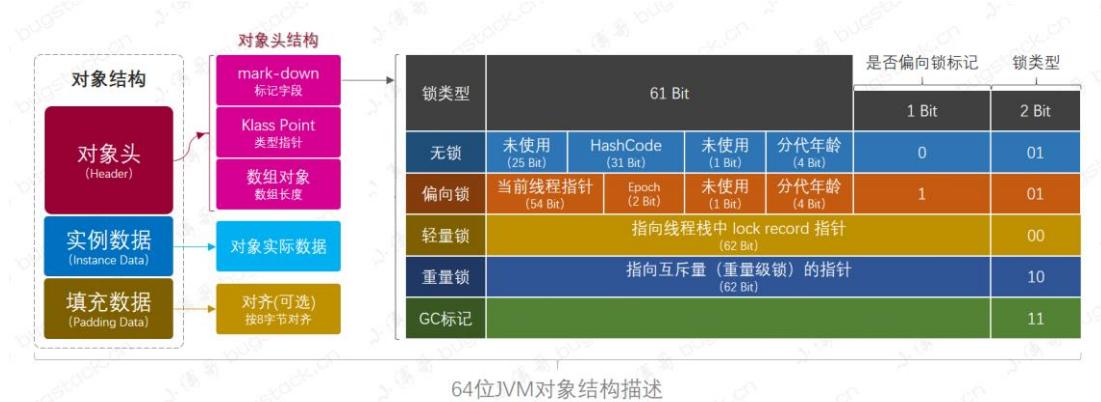


图 15-1 64 位 JVM 对象结构描述

HotSpot 虚拟机 markOop.cpp 中的 C++ 代码注释片段，描述了 64bits 下 mark-word 的存储状态，也就是图 15-1 的结构示意。

这部分的源码注释如下：

```
64 bits:  
-----  
unused:25 hash:31 -->| unused:1    age:4     biased_lock:1 lock:2 (normal object)  
JavaThread*:54 epoch:2 unused:1    age:4     biased_lock:1 lock:2 (biased object)  
PromotedObject*:61 ----->| promo_bits:3 ----  
->| (CMS promoted object)  
size:64 ----->| (CMS free block)  
  
unused:25 hash:31 -  
->| cms_free:1 age:4     biased_lock:1 lock:2 (COOPs && normal object)  
JavaThread*:54 epoch:2 cms_free:1 age:4     biased_lock:1 lock:2 (COOPs && biased ob  
ject)  
narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----  
->| (COOPs && CMS promoted object)  
unused:21 size:35 -->| cms_free:1 unused:7 -----  
->| (COOPs && CMS free block)
```

源码地址：[jdk8/hotspot/file/vm/oops/markOop.hpp](#)

HotSpot 虚拟机中，对象在内存中存储的布局可以分为三块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

- mark-word：对象标记字段占 4 个字节，用于存储一些列的标记位，比如：哈希值、轻量级锁的标记位、偏向锁标记位、分代年龄等。
- Klass Pointer：Class 对象的类型指针，Jdk1.8 默认开启指针压缩后为 4 字节，关闭指针压缩（`-XX:-UseCompressedOops`）后，长度为 8 字节。其指向的位置是对象对应的 Class 对象（其对应的元数据对象）的内存地址。
- 对象实际数据：包括对象的所有成员变量，大小由各个成员变量决定，比如：byte 占 1 个字节 8 比特位、int 占 4 个字节 32 比特位。
- 对齐：最后这段空间补全并非必须，仅仅为了起到占位符的作用。由于 HotSpot 虚拟机的内存管理系统要求对象起始地址必须是 8 字节的整数倍，所以对象头正好是 8 字节的倍数。因此当对象实例数据部分没有对齐的话，就需要通过对齐填充来补全。

另外，在 mark-word 锁类型标记中，无锁，偏向锁，轻量锁，重量锁，以及 GC 标记，5 种类中没法用 2 比特标记（2 比特最终有 4 种组合 `00`、`01`、`10`、`11`），所以无锁、偏向锁，前又占了一位偏向锁标记。最终：101 为无锁、001 为偏向锁。

1.2 验证对象结构

为了可以更加直观的看到对象结构，我们可以借助 `openjdk` 提供的 `jol-core` 进行打印分析。

引入 POM

```
<!-- https://mvnrepository.com/artifact/org.openjdk.jol/jol-cli -->
<dependency>
    <groupId>org.openjdk.jol</groupId>
    <artifactId>jol-cli</artifactId>
    <version>0.14</version>
</dependency>
```

测试代码

```
public static void main(String[] args) {
    System.out.println(VM.current().details());
    Object obj = new Object();
    System.out.println(obj + " 十六进制哈希:
" + Integer.toHexString(obj.hashCode()));
    System.out.println(ClassLayout.parseInstance(obj).toPrintable());
}
```

1.2.1 指针压缩开启(默认)

运行结果

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

java.lang.Object object internals:
OFFSET  SIZE   TYPE DESCRIPTION                                VALUE
          4           (object header)                            01 00 00 00 (00000000
1 00000000 00000000 00000000) (1)
```

```

        4      4      (object header)          00 00 00 00 (00000000
0 00000000 00000000 00000000) (0)

        8      4      (object header)          e5 01 00 f8 (1110010
1 00000001 00000000 11111000) (-134217243)

        12     4      (loss due to the next object alignment)

Instance size: 16 bytes

Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

```

java.lang.Object object internals: mark-down			
OFFSET	SIZE	TYPE	DESCRIPTION
0	4	(object header)	01 00 00 00 (00000001 00000000 00000000 00000000) (1)
4	4	(object header)	00 00 00 00 (00000000 00000000 00000000 00000000) (0)
8	4	(object header)	e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
12	4	(loss due to the next object alignment)	

Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

图 15-2 指针压缩开启，对象头布局

- Object 对象，总共占 16 字节
- 对象头占 12 个字节，其中：mark-down 占 8 字节、Klass Point 占 4 字节
- 最后 4 字节，用于数据填充找齐

1.2.2 指针压缩关闭

在 Run-->Edit Configurations->VM Options 配置参数 -XX:-UseCompressedOops 关闭指针压缩。

运行结果

```

java.lang.Object object internals:
OFFSET  SIZE  TYPE DESCRIPTION          VALUE
0      4      (object header)          01 12 0c 53 (00000000
1 00010010 00001100 01010011) (1393299969)

        4      4      (object header)          02 00 00 00 (00000001
0 00000000 00000000 00000000) (2)

        8      4      (object header)          00 1c b9 1b (00000000
0 00011100 10111001 00011011) (465116160)

        12     4      (object header)          00 00 00 00 (00000000
0 00000000 00000000 00000000) (0)

Instance size: 16 bytes

Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

```

java.lang.Object object internals:			
OFFSET	SIZE	TYPE	DESCRIPTION
0	4	(object header)	01 12 0c 53 (00000001 00010010 00001100 01010011) (1393299969)
4	4	(object header)	02 00 00 00 (00000010 00000000 00000000 00000000) (2)
8	4	(object header)	00 1c b9 1b (00000000 00011100 10111001 00011011) (465116160)
12	4	(object header)	00 00 00 00 (00000000 00000000 00000000 00000000) (0)

Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

Klass Point 8字节

图 15-3 指针压缩关闭，对象头布局

- 关闭指针压缩后，mark-word 还是占 8 字节不变。
- 重点在类型指针 Klass Point 的变化，由原来的 4 字节，现在扩增到 8 字节。

1.2.3 对象头哈希值存储验证

接下来，我们调整下测试代码，看下哈希值在对象头中具体是怎么存放的。

测试代码

```
public static void main(String[] args) {
    System.out.println(VM.current().details());
    Object obj = new Object();
    System.out.println(obj + " 十六进制哈希:
" + Integer.toHexString(obj.hashCode()));
    System.out.println(ClassLayout.parseInstance(obj).toPrintable());
}
```

- 改动不多，只是把哈希值和对象打印出来，方便我们验证对象头关于哈希值的存放结果。

运行结果

java.lang.Object @2530c12 十六进制哈希: 2530c12			
java.lang.Object object internals:			
OFFSET	SIZE	TYPE	DESCRIPTION
0	4	(object header)	01 12 0c 53 (00000001 00010010 00001100 01010011) (1393299969)
4	4	(object header)	02 00 00 00 (00000010 00000000 00000000 00000000) (2)
8	4	(object header)	00 1c b9 1b (00000000 00011100 10111001 00011011) (465116160)
12	4	(object header)	00 00 00 00 (00000000 00000000 00000000 00000000) (0)

Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

图 15-3 对象头哈希值存放

- 如图 15-3，对象的哈希值是 16 进制的，**0x2530c12**
- 在对象头哈希值存放的结果上看，也有对应的数值。只不过这个结果是倒过来的。

关于这个倒过来的问题是因为，大小端存储导致；

- Big-Endian：高位字节存放于内存的低地址端，低位字节存放于内存的高地址端
- Little-Endian：低位字节存放于内存的低地址端，高位字节存放于内存的高地址端

mark-down 结构



图 15-5 无锁状态，64 位虚拟机 mark-down 结构

如图 15-5 最右侧的 3 Bit (1 Bit 标识偏向锁，2 Bit 描述锁的类型) 是跟锁类型和 GC 标记相关的，而 synchronized 的锁优化升级膨胀就是修改的这三位上的标识，来区分不同的锁类型。从而采取不同的策略来提升性能。

1.3 Monitor 对象

在 HotSpot 虚拟机中，monitor 是由 C++ 中 ObjectMonitor 实现。

synchronized 的运行机制，就是当 JVM 监测到对象在不同的竞争状况时，会自动切换到适合的锁实现，这种切换就是锁的升级、降级。

那么三种不同的 Monitor 实现，也就是常说的三种不同的锁：偏斜锁（Biased Locking）、轻量级锁和重量级锁。当一个 Monitor 被某个线程持有后，它便处于锁定状态。

Monitor 主要数据结构如下：

```
// initialize the monitor, exception the semaphore, all other fields
// are simple integers or pointers

ObjectMonitor() {
    _header      = NULL;
    _count       = 0;           // 记录个数
    _waiters     = 0,
    _recursions  = 0;           // 线程重入次数
    _object      = NULL;        // 存储 Monitor 对象
    _owner       = NULL;        // 持有当前线程的 owner
    _WaitSet     = NULL;        // 处于wait 状态的线程，会被加入到 _WaitSet
    _WaitSetLock = 0 ;
    _Responsible = NULL ;
    _succ        = NULL ;
    _cxq         = NULL ;      // 单向列表
    FreeNext     = NULL ;
}
```

```

    _EntryList      = NULL ; // 处于等待锁bLock 状态的线程, 会被加入到该列表
    _SpinFreq       = 0 ;
    _SpinClock      = 0 ;
    OwnerIsThread  = 0 ;
    _previous_owner_tid = 0;
}

```

源码地址: [jdk8/hotspot/file/vm/runtime/objectMonitor.hpp](https://github.com/AdrianLi1993/jdk8/blob/master/hotspot/file/vm/runtime/objectMonitor.hpp)

- ObjectMonitor, 有两个队列: `_WaitSet`、`_EntryList`, 用来保存 ObjectWaiter 对象列表。
- `_owner`, 获取 Monitor 对象的线程进入 `_owner` 区时, `_count + 1`。如果线程调用了 `wait()` 方法, 此时会释放 Monitor 对象, `_owner` 恢复为空, `_count - 1`。同时该等待线程进入 `_WaitSet` 中, 等待被唤醒。

锁  执行效果如下:

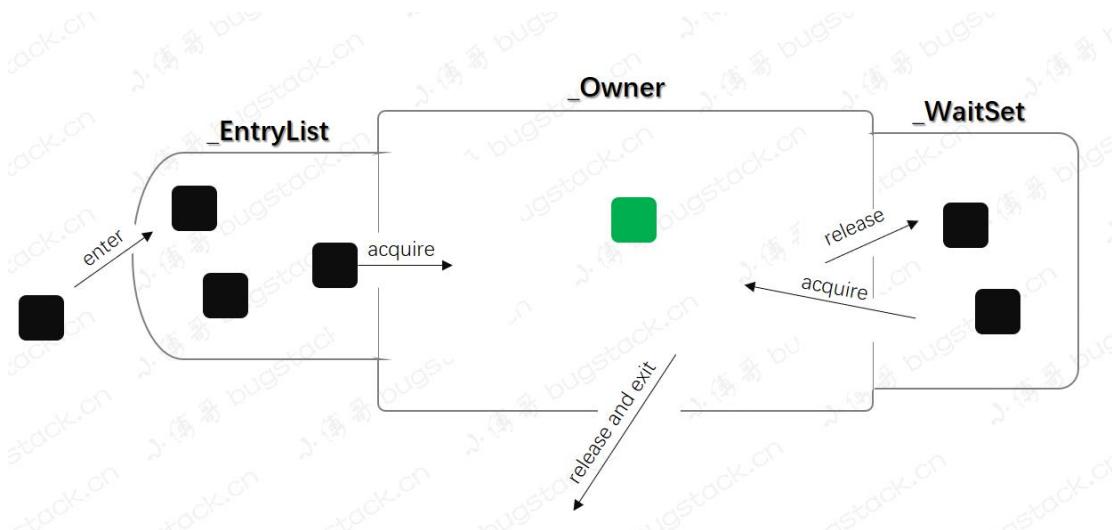


图 15-06, 锁  执行效果

如图 15-06, 每个 Java 对象头中都包括 Monitor 对象(存储的指针的指向), `synchronized` 也就是通过这一种方式获取锁, 也就解释了为什么 `synchronized()` 括号里放任何对象都能获得锁  !

2. synchronized 特性

2.1 原子性

原子性是指一个操作是不可中断的, 要么全部执行成功要么全部执行失败。

案例代码

```

private static volatile int counter = 0;

public static void main(String[] args) throws InterruptedException {
    for (int i = 0; i < 10; i++) {
        Thread thread = new Thread(() -> {
            for (int i1 = 0; i1 < 10000; i1++) {
                add();
            }
        });
        thread.start();
    }
    // 等10个线程运行完毕
    Thread.sleep(1000);
    System.out.println(counter);
}
public static void add() {
    counter++;
}

```

这段代码开启了 10 个线程来累加 counter，按照预期结果应该是 100000。但实际运行会发现，counter 值每次运行都小于 10000，这是因为 volatile 并不能保证原子性，所以最后的结果不会是 10000。

修改方法 add()，添加 synchronized：

```

public static void add() {
    synchronized (AtomicityTest.class) {
        counter++;
    }
}

```

这次测试结果就是：100000 了！

因为 synchronized 可以保证统一时间只有一个线程能拿到锁，进入到代码块执行。

反编译查看指令码

```
javap -v -p AtomicityTest
```

```

public static void add();
descriptor: ()V
flags: ACC_PUBLIC, ACC_STATIC
Code:

```

```

stack=2, locals=2, args_size=0
0: ldc           #12           // class org/itstack/interview/Atomic
ityTest
2: dup
3: astore_0
4: monitorenter
5: getstatic     #10           // Field counter:I
8: iconst_1
9: iadd
10: putstatic    #10           // Field counter:I
13: aload_0
14: monitorexit
15: goto         23
18: astore_1
19: aload_0
20: monitorexit
21: aload_1
22: athrow
23: return
Exception table:

```

同步方法

ACC_SYNCHRONIZED 这是一个同步标识，对应的 16 进制值是 0x0020
这 10 个线程进入这个方法时，都会判断是否有此标识，然后开始竞争 Monitor 对象。

同步代码

- **monitorenter**，在判断拥有同步标识 **ACC_SYNCHRONIZED** 抢先进入此方法的线程会优先拥有 Monitor 的 owner，此时计数器 +1。
- **monitorexit**，当执行完退出后，计数器 -1，归 0 后被其他进入的线程获得。

2.2 可见性

在上一章节 volatile 篇中，我们知道它保证变量对所有线程的可见性。最终的效果就是在添加 volatile 的属性变量时，线程 A 修改值后，线程 B 使用此变量可以做出相应的反应，比如 **while(!变量)** 退出。

那么，**synchronized** 具备可见性吗，我们做给例子。

```

public static boolean sign = false;
public static void main(String[] args) {
    Thread Thread01 = new Thread(() -> {
        int i = 0;
        while (!sign) {
            i++;
            add(i);
        }
    });
    Thread Thread02 = new Thread(() -> {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException ignore) {
        }
        sign = true;
        logger.info("vt.sign = true  while (!sign)");
    });
    Thread01.start();
    Thread02.start();
}

public static int add(int i) {
    return i + 1;
}

```

这是两个线程操作一个变量的例子，因为线程间对变量 `sign` 的不可见性，线程 `Thread01` 中的 `while (!sign)` 会一直执行，不会随着线程 `Thread02` 修改 `sign = true` 而退出循环。

现在我们给方法 `add` 添加 `synchronized` 关键字修饰，如下：

```

public static synchronized int add(int i) {
    return i + 1;
}

```

添加后运行结果：

```

23:55:33.849 [Thread-
1] INFO org.itstack.interview.VisibilityTest - vt.sign = true  while (!sign)

```

```
Process finished with exit code 0
```

可以看到当线程 Thread02 改变变量 sign = true 后，线程 Thread01 立即退出了循环。

注意：不要在方法中添加 `System.out.println()`，因为这个方法中含有 `synchronized` 会影响测试结果！

那么为什么添加 `synchronized` 也能保证变量的可见性呢？

因为：

1. 线程解锁前，必须把共享变量的最新值刷新到主内存中。
2. 线程加锁前，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值。
3. `volatile` 的可见性都是通过内存屏障（Memory Barrier）来实现的。
4. `synchronized` 靠操作系统内核互斥锁实现，相当于 JMM 中的 lock、unlock。退出代码块时刷新变量到主内存。

2.3 有序性

`as-if-serial`，保证不管编译器和处理器为了性能优化会如何进行指令重排序，都需要保证单线程下的运行结果的正确性。也就是说：如果在本线程内观察，所有的操作都是有序的；如果在一个线程观察另一个线程，所有的操作都是无序的。

这里有一段双重检验锁（Double-checked Locking）的经典案例：

```
public class Singleton {  
    private Singleton() {  
    }  
  
    private volatile static Singleton instance;  
  
    public Singleton getInstance() {  
        if (instance == null) {  
            synchronized (Singleton.class) {  
                if (instance == null) {  
                    instance = new Singleton();  
                }  
            }  
        }  
    }  
}
```

```
        }
    }
    return instance;
}

}
```

为什么，synchronized 也有可见性的特点，还需要 volatile 关键字？
因为，synchronized 的有序性，不是 volatile 的防止指令重排序。

那如果不加 volatile 关键字可能导致的结果，就是第一个线程在初始化初始化对象，设置 instance 指向内存地址时。第二个线程进入时，有指令重排。在判断 if (instance == null) 时就会有出错的可能，因为这会可能 instance 可能还没有初始化成功。

2.4 可重入性

synchronized 是可重入锁，也就是说，允许一个线程二次请求自己持有对象锁的临界资源，这种情况称为可重入锁 。

那么我们就写一个例子，来证明这样的情况。

```
public class ReentryTest extends A{

    public static void main(String[] args) {
        ReentryTest reentry = new ReentryTest();
        reentry.doA();
    }

    public synchronized void doA() {
        System.out.println("子类方法: ReentryTest.doA() ThreadId:
" + Thread.currentThread().getId());
        doB();
    }

    private synchronized void doB() {
        super.doA();
        System.out.println("子类方法: ReentryTest.doB() ThreadId:
" + Thread.currentThread().getId());
    }
}
```

```
    }

}

class A {
    public synchronized void doA() {
        System.out.println("父类方法: A.doA() ThreadId:
" + Thread.currentThread().getId());
    }
}
```

测试结果

```
子类方法: ReentryTest.doA() ThreadId: 1
父类方法: A.doA() ThreadId: 1
子类方法: ReentryTest.doB() ThreadId: 1

Process finished with exit code 0
```

这段单例代码是递归调用含有 `synchronized` 锁的方法，从运行正常的测试结果看，并没有发生死锁。所有可以证明 `synchronized` 是可重入锁。

`synchronized` 锁对象的时候有个计数器，他会记录下线程获取锁的次数，在执行完对应的代码块之后，计数器就会-1，直到计数器清零，就释放锁了。

之所以，是可以重入。是因为 `synchronized` 锁对象有个计数器，会随着线程获取锁后 +1 计数，当线程执行完毕后 -1，直到清零释放锁。

3. 锁升级过程

关于 `synchronized` 锁升级有一张非常完整的图，可以参考：

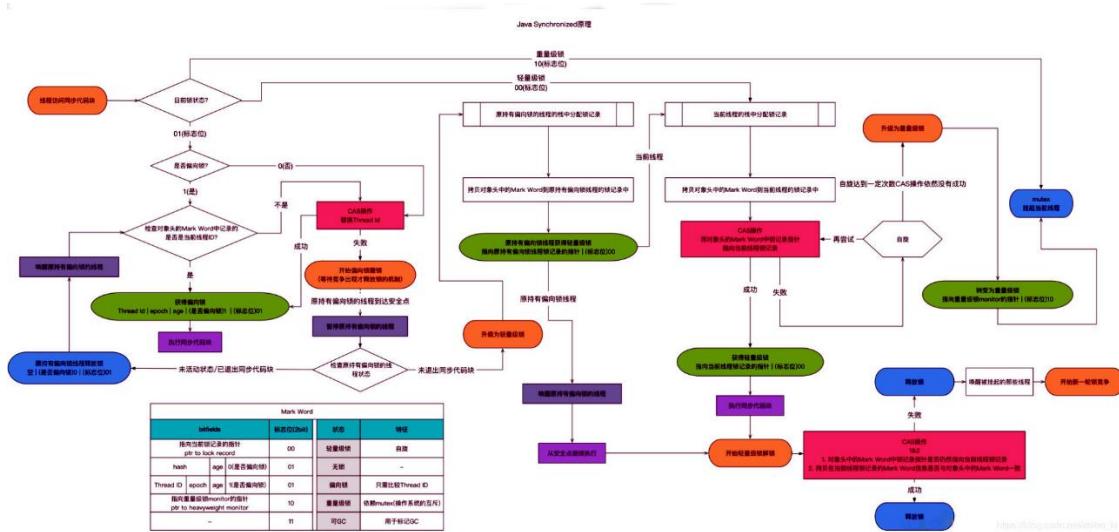


图 15-7 synchronized 锁升级过程

synchronized 锁有四种交替升级的状态：无锁、偏向锁、轻量级锁和重量级，这几个状态随着竞争情况逐渐升级。

3.1 偏向锁

synchronizer 源码：[/src/share/vm/runtime/synchronizer.cpp](#)

```
// NOTE: must use heavy weight monitor to handle jni monitor exit
void ObjectSynchronizer::jni_exit(oop obj, Thread* THREAD) {
    TEVENT (jni_exit) ;
    if (UseBiasedLocking) {
        Handle h_obj(THREAD, obj);
        BiasedLocking::revoke_and_rebias(h_obj, false, THREAD);
        obj = h_obj();
    }
    assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");

    ObjectMonitor* monitor = ObjectSynchronizer::inflate(THREAD, obj);
    // If this thread has locked the object, exit the monitor. Note: can't use
    // monitor->check(CHECK); must exit even if an exception is pending.
    if (monitor->check(THREAD)) {
        monitor->exit(true, THREAD);
    }
}
```

- UseBiasedLocking 是一个偏向锁检查，1.6 之后是默认开启的，1.5 中是关闭的，需要手动开启参数是 `XX:-UseBiasedLocking=false`

偏斜锁会延缓 JIT 预热进程，所以很多性能测试中会显式地关闭偏斜锁，偏斜锁并不适合所有应用场景，撤销操作（revoke）是比较重的行为，只有当存在较多不会真正竞争的 synchronized 块儿时，才能体现出明显改善。

3.2 轻量级锁

当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。

在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），JVM 虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的 Mark Word 的拷贝，官方称之为 Displaced Mark Word。

3.3 自旋锁

自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗 CPU。

自旋锁的默认大小是 10 次，可以调整：[-XX: PreBlockSpin](#)

如果自旋 n 次失败了，就会升级为重量级的锁。重量级的锁，在 [1.3 Monitor 对象中已经介绍](#)。

3.4 锁会降级吗？

之前一直了解到 Java 不会进行锁降级，但最近整理了大量的资料发现锁降级确实是会发生。

When safepoints are used?

Below are few reasons for HotSpot JVM to initiate a safepoint:

Garbage collection pauses

Code deoptimization

Flushing code cache

Class **redefinition** (e.g. hot swap or instrumentation)

Biased lock revocation

Various debug **operation** (e.g. deadlock check or stacktrace dump)

Biased lock revocation，当 JVM 进入安全点 [SafePoint](#) 的时候，会检查是否有闲置的 Monitor，然后试图进行降级。

三、总结

- 本章关于 `synchronized` 锁涉及到了较多的 C++ 源码分析学习，源码地址：
https://github.com/JetBrains/jdk8u_hotspot
- 关于锁的细节挖掘除了本文提到的还有很多知识点可以继续学习，可以结合 ifeve、并发编程、深入理解 JVM 虚拟机，等系列知识整理。
- 学习过程中结合 C++ 源代码中关于锁的实现，更容易理解可能原本晦涩难懂的概念。在结合实际的案例验证，会容易接受这部分知识。
- 好了，这篇就写到这里了，如果有观点和文章不准确的表达欢迎留言，互相学习，互相扫盲，互相进步。

四、傅诗一手

- 会所，里的码农会锁。
- 拥挤，就需加价升级。
- 项目，按摩对象头皮。
- 效果，可见原子有序。

第 3 节：ReentrantLock 和 公平锁

Java 学多少才能找到工作？

最近经常有小伙伴问我，以为我的经验来看，学多少够，好像更多的是看你的野心有多大。如果你只是想找个 10k 以内的二线城市的工作，那还是比较容易的。也不需要学数据结构、也不需要会算法、也需要懂源码、更不要有多少项目经验。

但反之我遇到一个国内大学 TOP2 毕业的娃，这货就是 Offer 收割机：腾讯、阿里、字节还有国外新加坡的工作机会等等，薪资待遇也是贼高，可能超过你对白菜价的认知。**上学无用、学习无用，纯属扯淡！**

你能在这条路上能付出的越多，能努力的越早，收获就会越大！

一、面试题

谢飞机，小记，刚去冬巴拉泡完脚放松的飞机，因为耐克袜子丢了，骂骂咧咧的赴约面试官。

面试官：咋了，飞机，怎么看上去不高兴。

谢飞机：没事，没事，我心思我学的 synchronized 呢！

面试官：那正好，飞机你会锁吗？

谢飞机：啊。。。我没去会所呀！！！你咋知道

面试官：我说 Java 锁，你想啥呢！你了解公平锁吗，知道怎么实现的吗，给我再说说！

谢飞机：公平锁！？嗯，是不 ReentrantLock 中用到了，我怎么感觉似乎有印象，但是不记得了。

面试官：哎，回家搜搜 CLH 吧！

二、ReentrantLock 和 公平锁

1. ReentrantLock 介绍

鉴于上一篇小傅哥已经基于，HotSpot 虚拟机源码分析 [synchronized](#) 实现和相应核心知识点，本来想在本章直接介绍下 ReentrantLock。但因为 ReentrantLock 知识点较多，因此会分几篇分别讲解，突出每一篇重点，避免猪八戒吞枣。

介绍：ReentrantLock 是一个可重入且独占式锁，具有与 synchronized 监视器 (monitor enter、monitor exit) 锁基本相同的行为和语意。但与 synchronized

相比，它更加灵活、强大、增加了轮询、超时、中断等高级功能以及可以创建公平和非公平锁。

2. ReentrantLock 知识链条

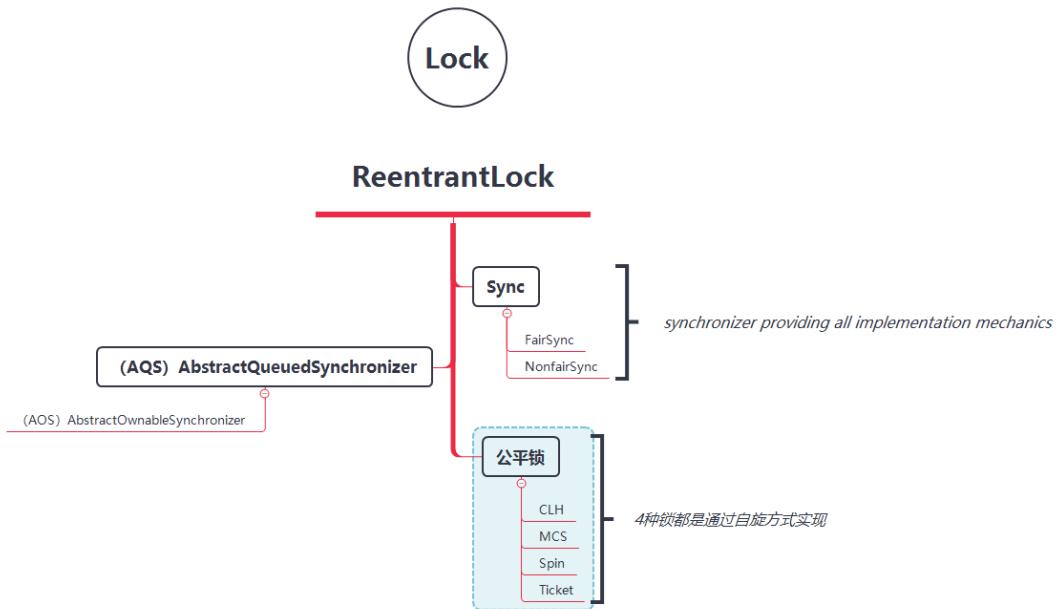


图 16-1 ReentrantLock 锁知识链条

ReentrantLock 是基于 Lock 实现的可重入锁，所有的 Lock 都是基于 AQS 实现的，AQS 和 Condition 各自维护不同的对象，在使用 Lock 和 Condition 时，其实就是两个队列的互相移动。它所提供的共享锁、互斥锁都是基于对 state 的操作。而它的可重入是因为实现了同步器 Sync，在 Sync 的两个实现类中，包括了公平锁和非公平锁。

这个公平锁的具体实现，就是我们本章节要介绍的重点，了解什么是公平锁、公平锁的具体实现。学习完基础的知识可以更好的理解 ReentrantLock

3. ReentrantLock 公平锁代码

3.1 初始化

```
ReentrantLock lock = new ReentrantLock(true); // true: 公平锁
lock.lock();
try {
    // todo
} finally {
```

```
    lock.unlock();
}
```

- 初始化构造函数入参，选择是否为初始化公平锁。
- 其实一般情况下并不需要公平锁，除非你的场景中需要保证顺序性。
- 使用 ReentrantLock 切记需要在 finally 中关闭，`lock.unlock()`。

3.2 公平锁、非公平锁，选择

```
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

- 构造函数中选择公平锁（FairSync）、非公平锁（NonfairSync）。

3.3 hasQueuedPredecessors

```
static final class FairSync extends Sync {

    protected final boolean tryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            if (!hasQueuedPredecessors() &&
                compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        ...
    }
}
```

- 公平锁和非公平锁，主要是在方法 `tryAcquire` 中，是否有 `!hasQueuedPredecessors()` 判断。

3.4 队列首位判断

```
public final boolean hasQueuedPredecessors() {  
    Node t = tail; // Read fields in reverse initialization order  
    Node h = head;  
    Node s;  
    return h != t &&  
        ((s = h.next) == null || s.thread != Thread.currentThread());  
}
```

- 在这个判断中主要就是看当前线程是不是同步队列的首位，是：true、否：false
- 这部分就涉及到了公平锁的实现，CLH（Craig, Landin and Hagersten）。三个作者的首字母组合

三、什么是公平锁

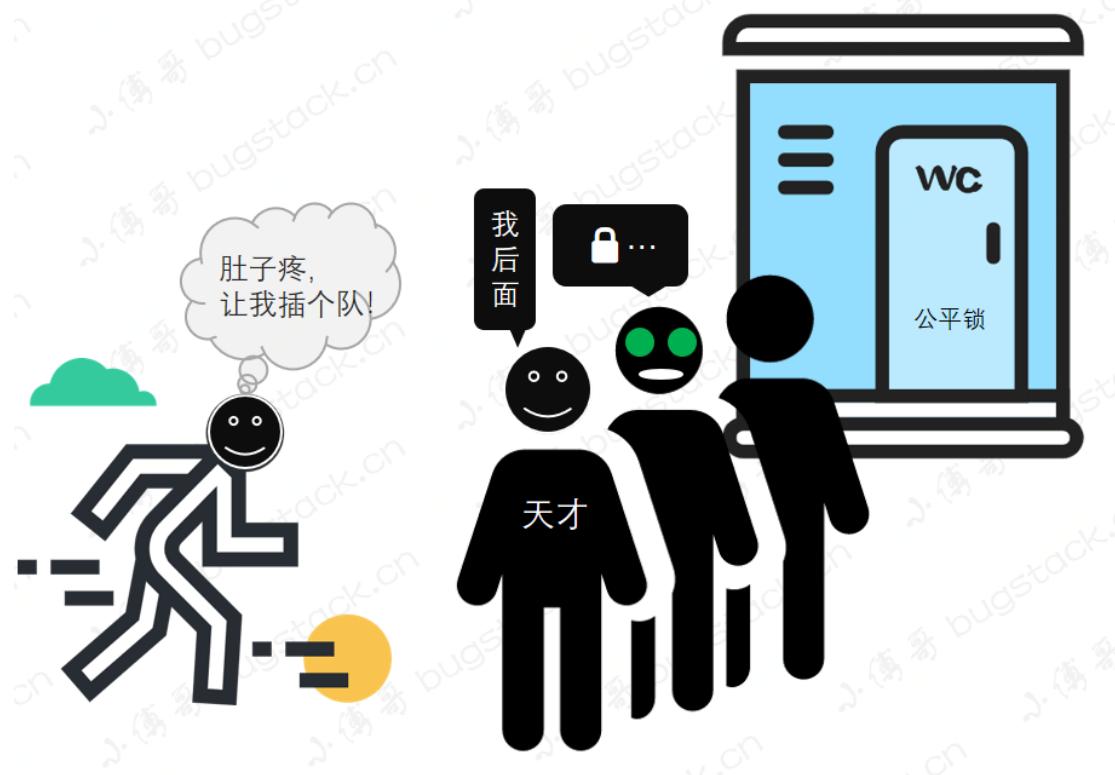


图 16-2 公共厕所排队入坑

公平锁就像是马路边上的卫生间，上厕所需要排队。当然如果有人不排队，那么就是非公平锁了，比如领导要先上。

CLH 是一种基于单向链表的高性能、公平的自旋锁。AQS 中的队列是 CLH 变体的虚拟双向队列（FIFO），AQS 是通过将每条请求共享资源的线程封装成一个节点来实现锁的分配。

为了更好的学习理解 CLH 的原理，就需要有实践的代码。接下来以 CLH 为核心分别介绍 4 种公平锁的实现，从而掌握最基本的技术栈知识。

四、公平锁实现

1. CLH

1.1 看图说话

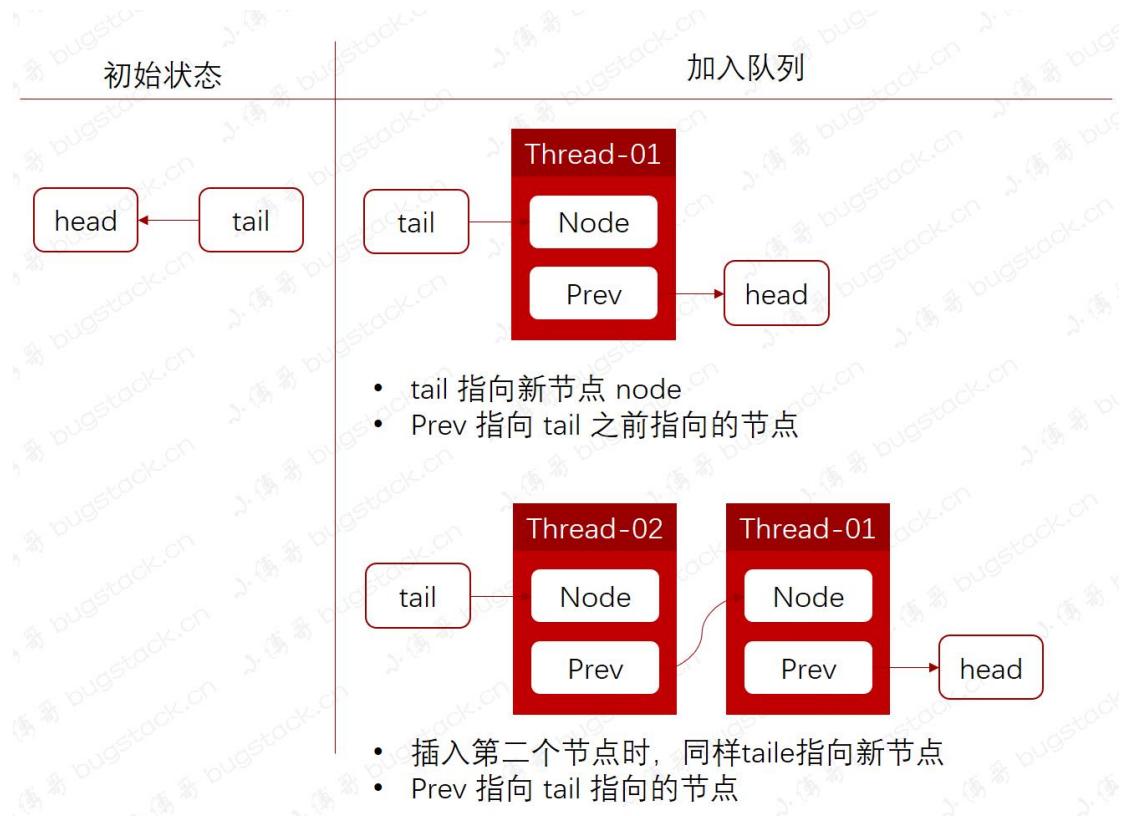


图 16-3 CLH 实现过程原理图

1.2 代码实现

```
public class CLHLock implements Lock {  
  
    private final ThreadLocal<CLHLock.Node> prev;  
    private final ThreadLocal<CLHLock.Node> node;  
    private final AtomicReference<CLHLock.Node> tail = new AtomicReference<>(new CL
```

```

HLock.Node());

private static class Node {
    private volatile boolean locked;
}

public CLHLock() {
    this.prev = ThreadLocal.withInitial(() -> null);
    this.node = ThreadLocal.withInitial(CLHLock.Node::new);
}

@Override
public void lock() {
    final Node node = this.node.get();
    node.locked = true;
    Node pred_node = this.tail.getAndSet(node);
    this.prev.set(pred_node);
    // 自旋
    while (pred_node.locked);
}

@Override
public void unlock() {
    final Node node = this.node.get();
    node.locked = false;
    this.node.set(this.prev.get());
}

}

```

1.3 代码讲解

CLH (Craig, Landin and Hagersten) , 是一种基于链表的可扩展、高性能、公平的自旋锁。

在这段代码的实现过程中，相当于是虚拟出来一个链表结构，由 `AtomicReference` 的方法 `getAndSet` 进行衔接。`getAndSet` 获取当前元素，设置新的元素

`lock()`

- 通过 `this.node.get()` 获取当前节点，并设置 `locked` 为 `true`。
- 接着调用 `this.tail.getAndSet(node)`，获取当前尾部节点 `pred_node`，同时把新加入的节点设置成尾部节点。
- 之后就是把 `this.prev` 设置为之前的尾部节点，也就相当于链路的指向。
- 最后就是自旋 `while (pred_node.locked)`，直至程序释放。

`unlock()`

- 释放锁的过程就是拆链，把释放锁的节点设置为 `false node.locked = false`。
- 之后最重要的是把当前节点设置为上一个节点，这样就相当于把自己的节点拆下来了，等着垃圾回收。

`CLH` 队列锁的优点是空间复杂度低，在 SMP (Symmetric Multi-Processor) 对称多处理器结构(一台计算机由多个 CPU 组成，并共享内存和其他资源，所有的 CPU 都可以平等地访问内存、I/O 和外部中断) 效果还是不错的。但在 NUMA (Non-Uniform Memory Access) 下效果就不太好，这部分知识可以自行扩展。

2. MCSLock

2.1 代码实现

```
public class MCSLock implements Lock {  
  
    private AtomicReference<MCSLock.Node> tail = new AtomicReference<>(null);  
    ;  
    private ThreadLocal<MCSLock.Node> node;  
  
    private static class Node {  
        private volatile boolean locked = false;  
        private volatile Node next = null;  
    }  
  
    public MCSLock() {  
        node = ThreadLocal.withInitial(Node::new);  
    }  
  
    @Override
```

```

public void lock() {
    Node node = this.node.get();
    Node preNode = tail.getAndSet(node);
    if (null == preNode) {
        node.locked = true;
        return;
    }
    node.locked = false;
    preNode.next = node;
    while (!node.locked) ;
}

@Override
public void unlock() {
    Node node = this.node.get();
    if (null != node.next) {
        node.next.locked = true;
        node.next = null;
        return;
    }
    if (tail.compareAndSet(node, null)) {
        return;
    }
    while (node.next == null) ;
}

}

```

2.1 代码讲解

MCS 来自于发明人名字的首字母： John Mellor-Crummey 和 Michael Scott。它也是一种基于链表的可扩展、高性能、公平的自旋锁，但与 CLH 不同。它是真的有下一个节点 next，添加这个真实节点后，它就可以只在本地变量上自旋，而 CLH 是前驱节点的属性上自旋。

因为自旋节点的不同，导致 CLH 更适合于 SMP 架构、MCS 可以适合 NUMA 非一致存储访问架构。你可以想象成 CLH 更需要线程数据在同一块内存上效果才更好，MCS 因为是在本地变量自选，所以无论数据是否分散在不同的 CPU 模块都没有影响。

代码实现上与 CLH 没有太多差异，这里就不在叙述了，可以看代码学习。

3. TicketLock

3.1 看图说话

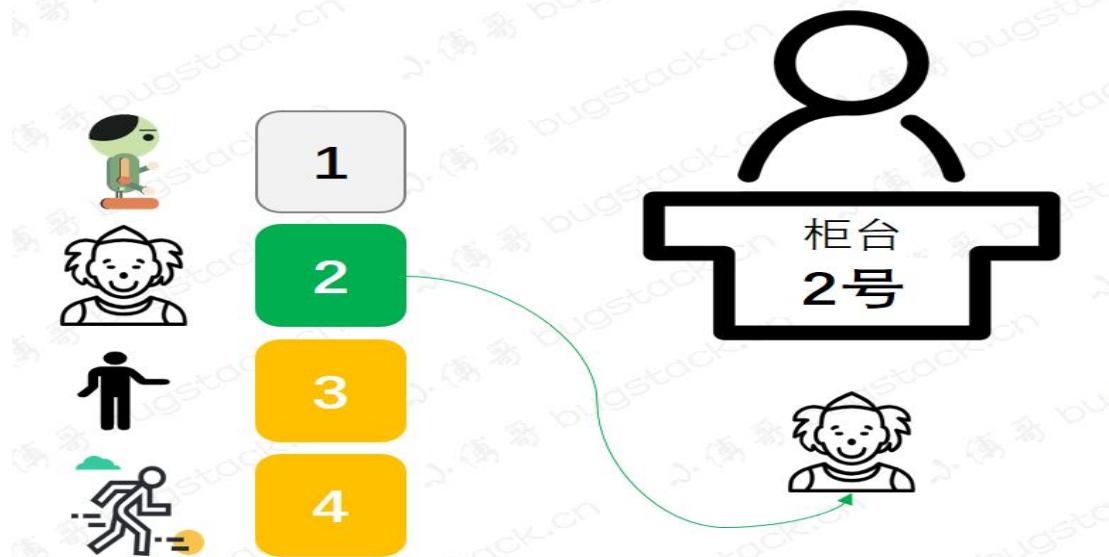


图 16-4 银行排队叫号图

3.2 代码实现

```
public class TicketLock implements Lock {

    private AtomicInteger serviceCount = new AtomicInteger(0);
    private AtomicInteger ticketCount = new AtomicInteger(0);
    private final ThreadLocal<Integer> owner = new ThreadLocal<>();

    @Override
    public void lock() {
        owner.set(ticketCount.getAndIncrement());
        while (serviceCount.get() != owner.get());
    }

    @Override
    public void unlock() {
        serviceCount.compareAndSet(owner.get(), owner.get() + 1);
    }
}
```

```
        owner.remove();
    }
}
```

3.3 代码讲解

TicketLock 就像你去银行、呷哺给你一个排号卡一样，叫到你号你才能进去。属于严格的公平性实现，但是多处理器系统上，每个进程/线程占用的处理器都在读写同一个变量，每次读写操作都需要进行多处理间的缓存同步，非常消耗系统性能。

代码实现上也比较简单，lock() 中设置拥有者的号牌，并进入自旋比对。unlock() 中使用 CAS 进行解锁操作，并处理移除。

五、总结

- ReentrantLock 是基于 Lock 实现的可重入锁，对于公平锁 CLH 的实现，只是这部分知识的冰山一角，但有这一脚，就可以很好热身便于后续的学习。
- ReentrantLock 使用起来更加灵活，可操作性也更大，但一定要在 finally 中释放锁，目的是保证在获取锁之后，最终能够被释放。同时不要将获取锁的过程写在 try 里面。
- 公平锁的实现依据不同场景和 SMP、NUMA 的使用，会有不同的优劣效果。在实际的使用中一般默认会选择非公平锁，即使是自旋也是耗费性能的，一般会用在较少等待的线程中，避免自旋时过长。

第 4 节：AQS 原理分析和实践运用

如果你相信你做什么都能成，你会自信的多！

千万不要总自我否定，尤其是职场的打工人。如果你经常感觉，这个做不好，那个学不会，别的也不懂，那么久而久之会越来越缺乏自信。

一般说能成事的人都具有赌徒精神，在他们眼里只要做这事那就一定能成，当然也有可能最后就没成，但在整个过程中人的心态是良好的，每天都有一个饱满的精神状态，孜孜不倦的奋斗着。最后也就是这样的斗志让走在一个起点的小伙伴，有了差距。

一、面试题

谢飞机，小记，今天打工人呀，明天早上困呀，嘟嘟嘟，喂？谁呀，打农药呢！？

谢飞机：哎呦，面试官大哥，咋了！

面试官：偷偷告诉你哈，你一面过了。

谢飞机：嘿嘿，真的呀！太好了！哈哈哈，那我还准备点什么呢！？

面试官：二面会比较难喽，嗯，我顺便问你一个哈。AQS 你了解吗，ReentrantLock 获取锁的过程是什么样的？什么是 CAS？…

谢飞机：我我我，脑子还在后羿射箭里，我一会就看看！！

面试官：好好准备下吧，打工人，打工魂！

二、ReentrantLock 和 AQS

1. ReentrantLock 知识链

ReentrantLock 可重入独占锁涉及的知识点较多，为了更好的学习这些知识，在上一章节先分析源码和学习实现了公平锁的几种方案。包括：CLH、MCS、Ticket，通过这部分内容的学习，再来理解 ReentrantLock 中关于 CLH 的变体实现和相应的应用就比较容易了。

接下来沿着 ReentrantLock 的知识链，继续分析 AQS 独占锁的相关知识点，如图 17-1

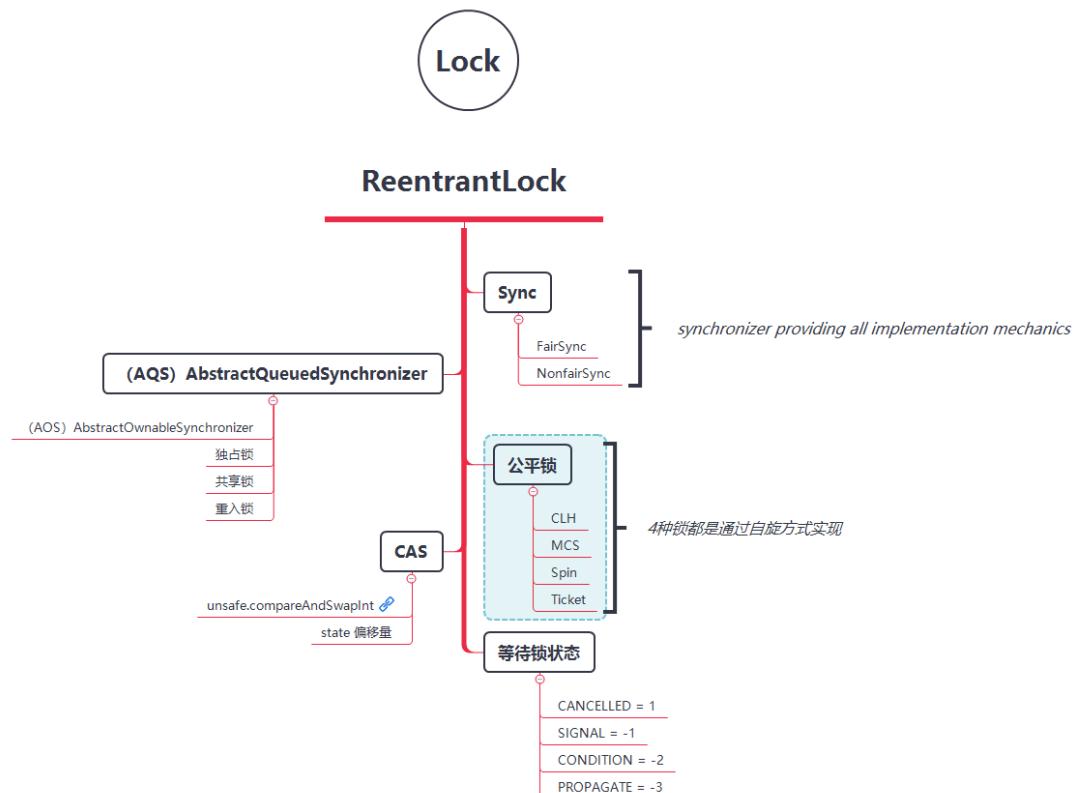


图 17-1 ReentrantLock 的知识链

在这部分知识学习中，会主要围绕 ReentrantLock 中关于 AQS 的使用进行展开，逐步分析源码了解原理。

AQS 是 AbstractQueuedSynchronizer 的缩写，几乎所有 Lock 都是基于 AQS 来实现了，其底层大量使用 CAS 提供乐观锁服务，在冲突时采用自旋方式进行重试，以此实现轻量级和高效的获取锁。

另外 AbstractQueuedSynchronizer 是一个抽象类，但并没有定义相应的抽象方法，而是提供了可以被子类继承时覆盖的 protected 的方法，这样就可以非常方便的支持继承类的使用。

2. 写一个简单的 AQS 同步类

在学习 ReentrantLock 中应用的 AQS 之前，先实现一个简单的同步类，来体会下 AQS 的作用。

2.1 代码实现

```
public class SyncLock {

    private final Sync sync;

    public SyncLock() {
        sync = new Sync();
    }

    public void lock() {
        sync.acquire(1);
    }

    public void unlock() {
        sync.release(1);
    }

    private static class Sync extends AbstractQueuedSynchronizer {
        @Override
        protected boolean tryAcquire(int arg) {
            return compareAndSetState(0, 1);
        }
    }
}
```

```

    }

    @Override
    protected boolean tryRelease(int arg) {
        setState(0);
        return true;
    }

    // 该线程是否正在独占资源，只有用到 Condition 才需要去实现
    @Override
    protected boolean isHeldExclusively() {
        return getState() == 1;
    }
}

}

```

这个实现的过程属于 ReentrantLock 简版，主要包括如下内容：

1. Sync 类继承 AbstractQueuedSynchronizer，并重写方法：tryAcquire、tryRelease、isHeldExclusively。
2. 这三个方法基本是必须重写的，如果不重写在使用的时候就会抛异常 `UnsupportedOperationException`。
3. 重写的过程也比较简单，主要是使用 AQS 提供的 CAS 方法。以预期值为 0，写入更新值 1，写入成功则获取锁成功。其实这个过程就是对 state 使用 `unsafe` 本地方法，传递偏移量 `stateOffset` 等参数，进行值交换操作。
`unsafe.compareAndSwapInt(this, stateOffset, expect, update)`
4. 最后提供 lock、unlock 两个方法，实际的类中会实现 Lock 接口中的相应方法，这里为了简化直接自定义这样两个方法。

2.2 单元测试

```

@Test
public void test_SyncLock() throws InterruptedException {
    final SyncLock lock = new SyncLock();
    for (int i = 0; i < 10; i++) {
        Thread.sleep(200);
        new Thread(new TestLock(lock), String.valueOf(i)).start();
    }
}

```

```

    }

    Thread.sleep(100000);
}

static class TestLock implements Runnable {

    private SyncLock lock;

    public TestLock(SyncLock lock) throws InterruptedException {
        this.lock = lock;
    }

    @Override
    public void run() {
        try {
            lock.lock();
            Thread.sleep(1000);
            System.out.println(String.format("Thread %s Completed", Thread.currentThread().getName()));
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}

```

- 以上这个单元测试和我们在上一章节介绍公平锁时是一样的，验证顺序输出。当然你也可以选择多线程操作一个方法进行加和运算。
- 在测试的过程中可以尝试把加锁代码注释掉，进行比对。如果可以顺序输出，那么就是预期结果。

测试结果

```

Thread 0 Completed
Thread 1 Completed
Thread 2 Completed
Thread 3 Completed
Thread 4 Completed
Thread 5 Completed
Thread 6 Completed
Thread 7 Completed

```

Thread 8 Completed

Thread 9 Completed

- 从测试结果看，以上 AQS 实现的同步类，满足预期效果。
- 有了这段代码的概念结构，接下来在分析 ReentrantLock 中的 AQS 使用就有一定的感觉了！

3. CAS 介绍

CAS 是 compareAndSet 的缩写，它的应用场景就是对一个变量进行值变更，在变更时会传入两个参数：一个是预期值、另外一个是更新值。如果被更新的变量预期值与传入值一致，则可以变更。

CAS 的具体操作使用到了 `unsafe` 类，底层用到了本地方法 `unsafe.compareAndSwapInt` 比较交换方法。

CAS 是一种无锁算法，这种操作是 CPU 指令集操作，只有一步原子操作，速度非常快。而且 CAS 避免了请求操作系统来裁定锁问题，直接由 CPU 搞定，但也不是没有开销，比如 Cache Miss，感兴趣的小伙伴可以自行了解 CPU 硬件相关知识。

4. AQS 核心源码分析

4.1 获解锁流程图

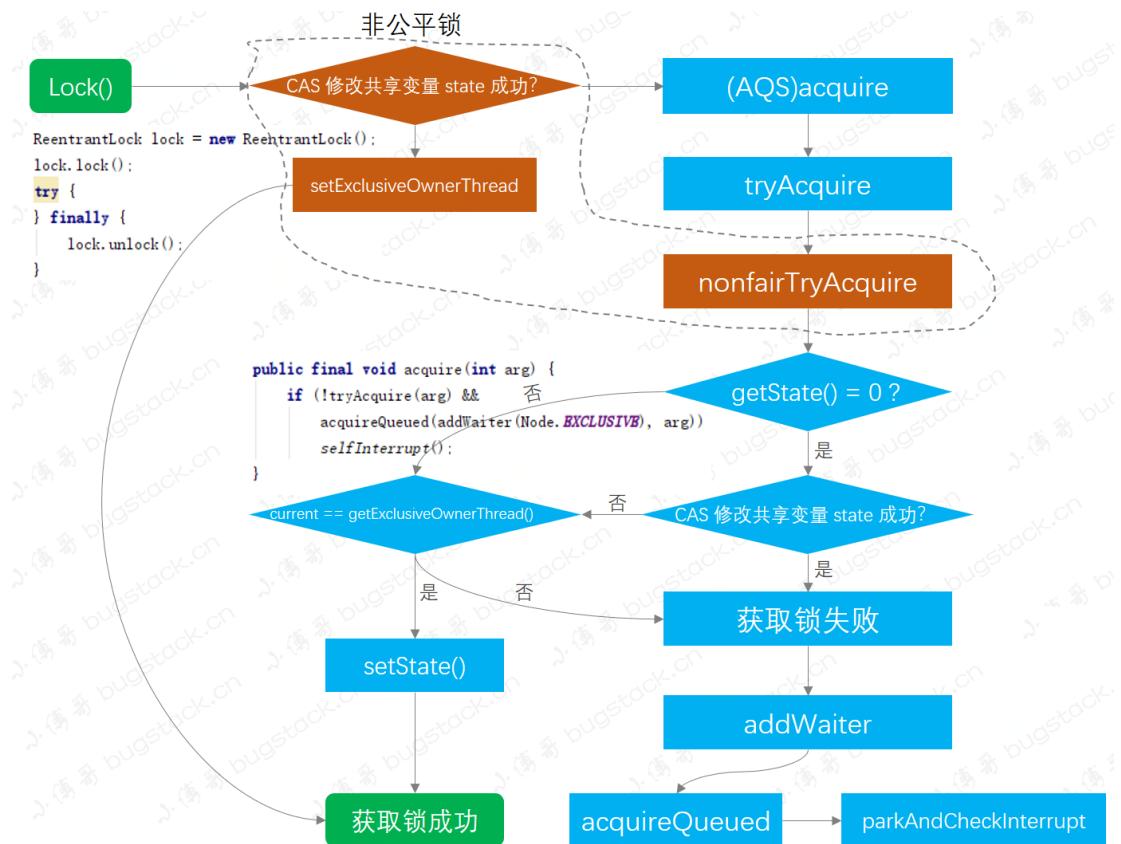


图 17-2 获得锁流程图

图 17-2 就是整个 ReentrantLock 中获得锁的核心流程，包括非公平锁和公平锁的一些交叉流程。接下来我们就以此按照此流程来讲解相应的源码部分。

4.2 lock

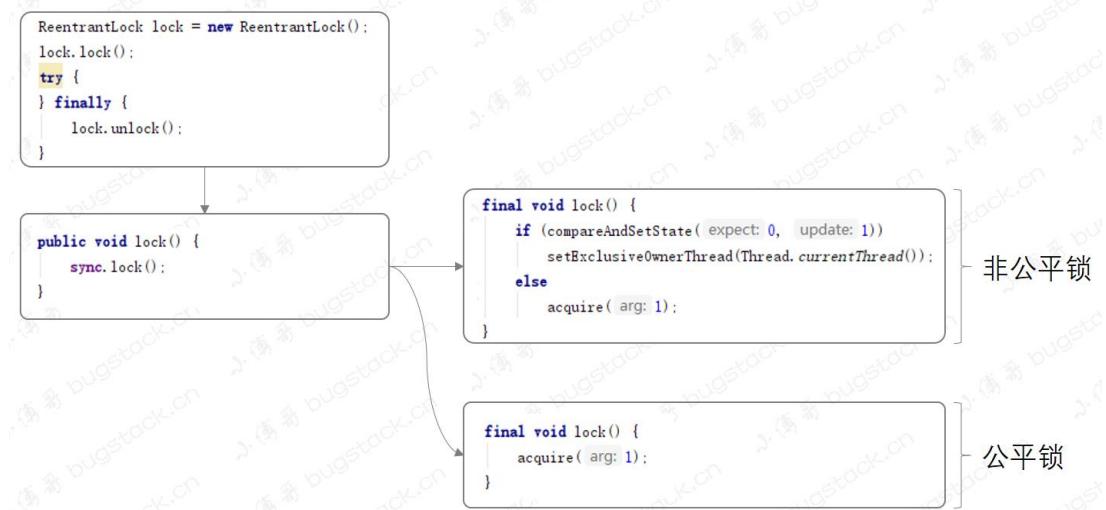


图 17-3 lock -> CAS

ReentrantLock 实现了非公平锁和公平锁，所以在调用 `lock.lock();` 时，会有不同的实现类：

1. 非公平锁，会直接使用 CAS 进行抢占，修改变量 state 值。如果成功则直接把自己的线程设置到 exclusiveOwnerThread，也就是获得锁成功。[不成功后续分析](#)
2. 公平锁，则不会进行抢占，而是规规矩矩的进行排队。老实人

4.3 compareAndSetState

```
final void lock() {  
    if (compareAndSetState(0, 1))  
        setExclusiveOwnerThread(Thread.currentThread());  
    else  
        acquire(1);  
}
```

在非公平锁的实现类里，获取锁的过程，有这样一段 CAS 操作的代码。`compareAndSetState` 赋值成功则获取锁。那么 CAS 这里面做了什么操作？

```
protected final boolean compareAndSetState(int expect, int update) {  
    // See below for intrinsics setup to support this  
    return unsafe.compareAndSwapInt(this, stateOffset, expect, update);  
}
```

往下翻我们看到这样一段代码，这里是 unsafe 功能类的使用，两个参数到这里变成四个参数。多了 `this`、`stateOffset`。`this` 是对象本身，那么 `stateOffset` 是什么？

```
stateOffset = unsafe.objectFieldOffset  
(AbstractQueuedSynchronizer.class.getDeclaredField("state"));
```

再往下看我们找到，`stateOffset` 是偏移量值，偏移量是一个固定的值。接下来我们就看看这个值到底是多少！

引用 POM jol-cli

```
<!-- https://mvnrepository.com/artifact/org.openjdk.jol/jol-cli -->  
<dependency>  
    <groupId>org.openjdk.jol</groupId>  
    <artifactId>jol-cli</artifactId>  
    <version>0.14</version>  
</dependency>
```

单元测试

```
@Test  
public void test_stateOffset() throws Exception {  
    Unsafe unsafe = getUnsafeInstance();  
    long state = unsafe.objectFieldOffset  
        (AbstractQueuedSynchronizer.class.getDeclaredField("state"));  
    System.out.println(state);  
}  
  
// 16
```

- 通过 `getUnsafeInstance` 方法获取 `Unsafe`, 这是一个固定的方法。
- 在获取 AQS 类中的属性字段 `state` 的偏移量, 16。
- 除了这个属性外你还可以拿到: `headOffset`、`tailOffset`、`waitStatusOffset`、`nextOffset`, 的值, 最终自旋来变更这些变量的值。

4.4 (AQS) acquire

```
public final void acquire(int arg) {  
    if (!tryAcquire(arg) &&  
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))  
        selfInterrupt();  
}
```

整个这块代码里面包含了四个方法的调用, 如下:

1. `tryAcquire`, 分别由继承 AQS 的公平锁 (`FairSync`)、非公平锁 (`NonfairSync`) 实现。
2. `addWaiter`, 该方法是 AQS 的私有方法, 主要用途是方法 `tryAcquire` 返回 `false` 以后, 也就是获取锁失败以后, 把当前请求锁的线程添加到队列中, 并返回 `Node` 节点。
3. `acquireQueued`, 负责把 `addWaiter` 返回的 `Node` 节点添加到队列结尾, 并会执行获取锁操作以及判断是否把当前线程挂起。
4. `selfInterrupt`, 是 AQS 中的 `Thread.currentThread().interrupt()` 方法调用, 它的主要作用是在执行完 `acquire` 之前自己执行中断操作。

4.5 tryAcquire

```

final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}

```

这部分获取锁的逻辑比较简单，主要包括两部分：

1. 如果 `c == 0`，锁没有被占用，尝试使用 CAS 方式获取锁，并返回 `true`。
2. 如果 `current == getExclusiveOwnerThread()`，也就是当前线程持有锁，则需要调用 `setState` 进行锁重入操作。`setState` 不需要加锁，因为是在自己的当前线程下。
3. 最后如果两种都不满足 \odot ，则返回 `false`。

4.6 addWaiter

```

private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    Node pred = tail;
    // 如果队列不为空，使用 CAS 方式将当前节点设为尾节点
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
}

```

```

    }

    // 队列为空、CAS 失败，将节点插入队列
    enq(node);

    return node;
}

```

- 当执行方法 `addWaiter`, 那么就是 `!tryAcquire = true`, 也就是 `tryAcquire` 获取锁失败了。
- 接下来就是把当前线程封装到 `Node` 节点中, 加入到 FIFO 队列中。因为先进先出, 所以后来的队列加入到队尾
- `compareAndSetTail` 不一定一定成功, 因为在并发场景下, 可能会出现操作失败。那么失败后, 则需要调用 `enq` 方法, 该方法会自旋操作, 把节点入队列。

enq

```

private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}

```

- 自旋转 `for 循环 + CAS 入队列。`
- 当队列为空时, 则会新创建一个节点, 把尾节点指向头节点, 然后继续循环。
- 第二次循环时, 则会把当前线程的节点添加到队尾。`head` 节是一个无用节点, 这和我们做 `CLH` 实现时类似

注意，从尾节点逆向遍历

- 首先这里的节点连接操作并不是原子，也就是说在多线程并发的情况下，可能会出现个别节点并没有设置 next 值，就失败了。
- 但这些节点的 prev 是有值的，所以需要逆向遍历，让 prev 属性重新指向新的尾节点，直至全部自旋入队列。

4.7 acquireQueued

```
final boolean acquireQueued(final Node node, int arg) {  
    boolean failed = true;  
    try {  
        boolean interrupted = false;  
        for (;;) {  
            final Node p = node.predecessor();  
            // 当前节点的前驱就是 head 节点时，再次尝试获取锁  
            if (p == head && tryAcquire(arg)) {  
                setHead(node);  
                p.next = null; // help GC  
                failed = false;  
                return interrupted;  
            }  
            // 获取锁失败后，判断是否把当前线程挂起  
            if (shouldParkAfterFailedAcquire(p, node) &&  
                parkAndCheckInterrupt())  
                interrupted = true;  
        }  
    } finally {  
        if (failed)  
            cancelAcquire(node);  
    }  
}
```

当获取锁流程走到这，说明节点已经加入队列完成。看源码中接下来就是让该方法再次尝试获取锁，如果获取锁失败会判断是否把线程挂起。

setHead

```
private void setHead(Node node) {  
    head = node;
```

```
    node.thread = null;
    node.prev = null;
}
```

在学习 CLH 公平锁数据结构中讲到 Head 节点是一个虚节点，如果当前节点的前驱节点是 Head 节点，那么说明此时 Node 节点排在队列最前面，可以尝试获取锁。

获取锁后设置 Head 节点，这个过程就是一个出队列过程，原来节点设置 Null 方便 GC。

shouldParkAfterFailedAcquire

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL)
        // SIGNAL 设置了前一个节点完结唤醒，安心干别的去了，这里是睡。
        return true;
    if (ws > 0) {
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

你是否还 CANCELLED、SIGNAL、CONDITION 、PROPAGATE ，这四种状态，在这个方法中用到了两种如下：

1. CANCELLED，取消排队，放弃获取锁。
2. SIGNAL，标识当前节点的下一个节点状态已经被挂起，意思就是大家一起排队上厕所，队伍太长了，后面的谢飞机说，我去买个油条哈，一会到我了，你微信我哈。其实就是当前线程执行完毕后，需要额外执行唤醒后继节点操作。

那么，以上这段代码主要的执行内容包括：

1. 如果前一个节点状态是 SIGNAL，则返回 true。*安心睡觉②等着被叫醒*

2. 如果前一个节点状态是 **CANCELLED**, 就是它放弃了, 则继续向前寻找其他节点。
3. 最后如果什么都没找到, 就给前一个节点设置个闹钟 **SIGNAL**, 等着被通知。

4.8 parkAndCheckInterrupt

```
if (shouldParkAfterFailedAcquire(p, node) &&
    parkAndCheckInterrupt())
    interrupted = true;

// 线程挂起等待被唤醒
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

- 当方法 **shouldParkAfterFailedAcquire** 返回 `false` 时, 则执行 `parkAndCheckInterrupt()` 方法。
- 那么, 这一段代码就是对线程的挂起操作, `LockSupport.park(this);`。
- `Thread.interrupted()` 检查当前线程的中断标识。

三、总结

- ReentrantLock 的知识比较多, 涉及的代码逻辑也比较复杂, 在学习的过程中需要对照源码和相关并发书籍和资料一起学习, 以及最好的是自身实践。
- AQS 的实现部分涉及的内容较多, 例如: state 属性使用 unsafe 提供的本地方法进行 CAS 操作, 把初始值 0 改为 1, 则获得了锁。addWaiter、acquireQueued、shouldParkAfterFailedAcquire、parkAndCheckInterrupt 等, 可以细致总结。
- 所有的 Lock 都是基于 AQS 来实现了。AQS 和 Condition 各自维护了不同的队列, 在使用 Lock 和 Condition 的时候, 就是两个队列的互相移动。这句话可以细细体会。可能文中会有一些不准确或者错字, 欢迎留言, 我会不断的更新博客。

第 5 节：AQS 共享锁，Semaphore、CountDownLatch

学 Java 怎么能，突飞猛进的成长？

是不是你看见过的突飞猛进都是别人，但自己却很难！

其实并没有一天的突飞猛进，也没有一口吃出来的胖子。有得更多的时候日积月累、不断沉淀，最后才能爆发、破局！

举个简单的例子，如果你大学毕业时候已经写了 40 万行代码，还找不到工作吗？但 40 万行平均到每天并不会很多，重要的是持之以恒的坚持。

一、面试题

谢飞机，小记！东风吹、战鼓擂，不加班、谁怕谁！哈哈哈，找我大哥去。

谢飞机：喂，大哥。我女友面试卡住了，强人锁难，锁我也不会！

面试官：你不应该不会呀，问你一个，基于 AQS 实现的锁都有哪些？

谢飞机：嗯，有 ReentrantLock...

面试官：还有呢？

谢飞机：好像想不起来了，sync 也不是！

面试官：哎，学点漏点，不思考、不总结、不记录。你这样人家面试你就没法聊了，最起码你要有点深度。

谢飞机：嘿嘿，记住了。来我家吃火锅吧，细聊。

二、共享锁 和 AQS

1. 基于 AQS 实现的锁有哪些？

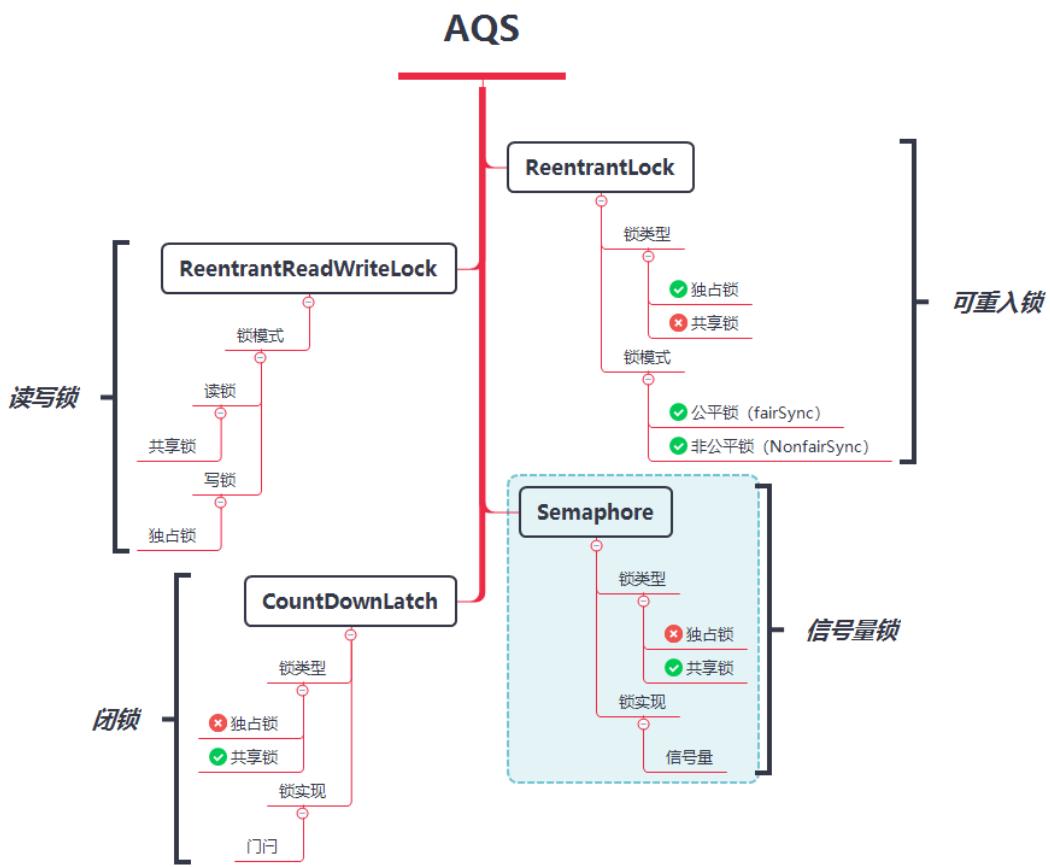


图 18-1 基于 AQS 实现的锁

AQS (AbstractQueuedSynchronizer)，是 Java 并发包中非常重要的一个类，大部分锁的实现也是基于 AQS 实现的，包括：

- **ReentrantLock**，可重入锁。这个是我们最开始介绍的锁，也是最常用的锁。通常会与 synchronized 做比较使用。
- **ReentrantReadWriteLock**，读写锁。读锁是共享锁、写锁是独占锁。
- **Semaphore**，信号量锁。主要用于控制流量，比如：数据库连接池给你分配 10 个链接，那么让你来一个连一个，连到 10 个还没有人释放，那你就等等。
- **CountDownLatch**，闭锁。Latch 门闩的意思，比如：说四个人一个漂流艇，坐满了就推下水。

这一章节我们主要来介绍 Semaphore，信号量锁的实现，其实也就是介绍一个关于共享锁的使用和源码分析。

2. Semaphore 共享锁使用

```

Semaphore semaphore = new Semaphore(2, false); // 构造函数入参, permits: 信号量、
fair: 公平锁/非公平锁

for (int i = 0; i < 8; i++) {
    new Thread(() -> {
        try {
            semaphore.acquire();
            System.out.println(Thread.currentThread().getName() + "蹲坑");
            Thread.sleep(1000L);
        } catch (InterruptedException ignore) {
        } finally {
            semaphore.release();
        }
    }, "蹲坑编号: " + i).start();
}

```

这里我们模拟了一个在高速服务区，厕所排队蹲坑的场景。由于坑位有限，为了避免造成拥挤和踩踏，保安人员在门口拦着，感觉差不多，一次释放两个进去，一直到都释放。你也可以想成早上坐地铁上班，或者旺季去公园，都是一批一批的放行

测试结果

蹲坑编号: 0 蹲坑

蹲坑编号: 1 蹲坑

蹲坑编号: 2 蹲坑

蹲坑编号: 3 蹲坑

蹲坑编号: 4 蹲坑

蹲坑编号: 5 蹲坑

蹲坑编号: 6 蹲坑

蹲坑编号: 7 蹲坑

Process finished with exit code 0

- Semaphore 的构造函数可以传递是公平锁还是非公平锁，最终的测试结果也不同，可以自行尝试。

- 测试运行时，会先输出 **0 坑、1 坑**，之后 **2 坑、3 坑**...，每次都是这样两个，两个的释放。这就是 Semaphore 信号量锁的作用。

3. Semaphore 源码分析

3.1 构造函数

```
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

public Semaphore(int permits, boolean fair) {
    sync = fair ? new FairSync(permits) : new NonfairSync(permits);
}
```

permits: n. 许可证, 特许证(尤指限期的)

默认情况下只需要传入 permits 许可证数量即可，也就是一次允许放行几个线程。构造函数会创建非公平锁。如果你需要使用 Semaphore 共享锁中的公平锁，那么可以传入第二个构造函数的参数 fair = false/true。true: FairSync，公平锁。在我们前面的章节已经介绍了公平锁相关内容和实现，以及 CLH、MCS [《公平锁介绍》](#)

初始许可证数量

```
FairSync/NonfairSync(int permits) {
    super(permits);
}

Sync(int permits) {
    setState(permits);
}

protected final void setState(int newState) {
    state = newState;
}
```

在构造函数初始化的时候，无论是公平锁还是非公平锁，都会设置 AQS 中 state 数量值。这个值也就是为了下文中可以获取的信号量扣减和增加的值。

3.2 acquire 获取信号量

270 / 417

小傅哥，公众号：bugstack 虫洞栈 | 沉淀、分享、成长，让自己和他人都能有所收获

方法	描述
semaphore.acquire()	一次获取一个信号量，响应中断
semaphore.acquire(2)	一次获取 n 个信号量，响应中断（一次占 2 个坑）
semaphore.acquireUninterruptibly()	一次获取一个信号量，不响应中断
semaphore.acquireUninterruptibly(2)	一次获取 n 个信号量，不响应中断

- 其实获取信号量的这四个方法，主要就是，一次获取几个和是否响应中断的组合。
- `semaphore.acquire()`，源码中实际调用的方法是，
`sync.acquireSharedInterruptibly(1)`。也就是相应中断，一次只占一个坑。
- `semaphore.acquire(2)`，同理这个就是要占两个名额，也就是许可证。
生活中的场景就是我给我朋友排的对，她来了，进来吧。

3.3 acquire 释放信号量

方法	描述
semaphore.release()	一次释放一个信号量
semaphore.release(2)	一次释放 n 个信号量

有获取就得有释放，获取了几个信号量就要释放几个信号量。当然你可以尝试一下，获取信号量 `semaphore.acquire(2)` 两个，释放信号量 `semaphore.release(1)`，看看运行效果

3.4 公平锁实现

信号量获取过程，一直到公平锁实现。`semaphore.acquire` \rightarrow `sync.acquireSharedInterruptibly(permits)` \rightarrow `tryAcquireShared(arg)`

```
semaphore.acquire(1);

public void acquire(int permits) throws InterruptedException {
    if (permits < 0) throw new IllegalArgumentException();
    sync.acquireSharedInterruptibly(permits);
}
```

```
public final void acquireSharedInterruptibly(int arg)
    throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (tryAcquireShared(arg) < 0)
        doAcquireSharedInterruptibly(arg);
}
```

FairSync. tryAcquireShared

```
protected int tryAcquireShared(int acquires) {
    for (;;) {
        if (hasQueuedPredecessors())
            return -1;
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

- `hasQueuedPredecessors`, 公平锁的主要实现逻辑都在于这个方法的使用。它的目的就是判断有线程排在自己前面没，以及把线程添加到队列中的逻辑实现。在前面我们介绍过 CLH 等实现，可以往前一章节阅读。
- `for (;;)`, 是一个自旋的过程，通过 CAS 来设置 state 偏移量对应值。这样就可以避免多线程下竞争获取信号量冲突。
- `getState()`, 在构造函数中已经初始化 state 值，在这里获取信号量时就是使用 CAS 不断的扣减。
- 另外需要注意，共享锁和独占锁在这里是有区别的，独占锁直接返回 true/false，共享锁返回的是 int 值。
 - 如果该值小于 0，则当前线程获取共享锁失败。
 - 如果该值大于 0，则当前线程获取共享锁成功，并且接下来其他线程尝试获取共享锁的行为很可能成功。
 - 如果该值等于 0，则当前线程获取共享锁成功，但是接下来其他线程尝试获取共享锁的行为会失败。

3.5 非公平锁实现

NonfairSync. nonfairTryAcquireShared

```

protected int tryAcquireShared(int acquires) {
    return nonfairTryAcquireShared(acquires);
}

final int nonfairTryAcquireShared(int acquires) {
    for (;;) {
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}

```

- 有了公平锁的实现，非公平锁的理解就比较简单了，只是拿去了 `if (hasQueuedPredecessors())` 的判断操作。
- 其他的逻辑实现都和公平锁一致。

3.6 获取信号量失败，加入同步等待队列

在公平锁和非公平锁的实现中，我们已经看到正常获取信号量的逻辑。那么如果此时不能正常获取信号量呢？其实这部分线程就需要加入到同步队列。

doAcquireSharedInterruptibly

```

public final void acquireSharedInterruptibly(int arg)
    throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (tryAcquireShared(arg) < 0)
        doAcquireSharedInterruptibly(arg);
}

private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        for (;;) {

```

```

        final Node p = node.predecessor();
        if (p == head) {
            int r = tryAcquireShared(arg);
            if (r >= 0) {
                setHeadAndPropagate(node, r);
                p.next = null; // help GC
                failed = false;
                return;
            }
        }
        if (shouldParkAfterFailedAcquire(p, node) &&
            parkAndCheckInterrupt())
            throw new InterruptedException();
    }
} finally {
    if (failed)
        cancelAcquire(node);
}
}

```

- 首先 `doAcquireSharedInterruptibly` 方法来自 AQS 的内部方法，与我们在学习竞争锁时有部分知识点相同，但也有一些差异。比如：`addWaiter(Node.SHARED)`, `tryAcquireShared`, 我们主要介绍下这内容。
- `Node.SHARED`, 其实没有特殊含义, 它只是一个标记作用, 用于判断是否共享。
`final boolean isShared() { return nextWaiter == SHARED; }`
- `tryAcquireShared`, 主要是来自 `Semaphore` 共享锁中公平锁和非公平锁的实现。用来获取同步状态。
- `setHeadAndPropagate(node, r)`, 如果 $r > 0$, 同步成功后则将当前线程结点设置为头结点, 同时 `helpGC`, `p.next = null`, 断链操作。
- `shouldParkAfterFailedAcquire(p, node)`, 调整同步队列中 `node` 结点的状态, 并判断是否应该被挂起。这在我们之前关于锁的文章中已经介绍。
- `parkAndCheckInterrupt()`, 判断是否需要被中断, 如果中断直接抛出异常, 当前结点请求也就结束。
- `cancelAcquire(node)`, 取消该节点的线程请求。

4. CountDownLatch 共享锁使用

CountDownLatch 也是共享锁的一种类型，之所以在这里体现下，是因为它和 Semaphore 共享锁，既相似有不同。

CountDownLatch 更多体现的组团一波的思想，同样是控制人数，但是需要够一窝。比如：我们说过的 4 个人一起上皮划艇、两个人一起上跷跷板、2 个人一起蹲坑我没见过，这样的方式就是门诊 CountDownLatch 锁的思想。

```
public static void main(String[] args) throws InterruptedException {
    CountDownLatch latch = new CountDownLatch(10);
    ExecutorService exec = Executors.newFixedThreadPool(10);
    for (int i = 0; i < 10; i++) {
        exec.execute(() -> {
            try {
                int millis = new Random().nextInt(10000);
                System.out.println("等待游客上船，耗时：" + millis + "(millis)");
                Thread.sleep(millis);
            } catch (Exception ignore) {
            } finally {
                latch.countDown(); // 完事一个扣减一个名额
            }
        });
    }
    // 等待游客
    latch.await();
    System.out.println("船长急躁了，开船!");
    // 关闭线程池
    exec.shutdown();
}
```

- 这一个公园游船的场景案例，等待 10 个乘客上传，他们比较墨迹。
- 上一个扣减一个 `latch.countDown()`
- 等待游客都上船 `latch.await()`
- 最后船长开船!! 急躁了

测试结果

等待游客上船，耗时：`6689(millis)`

等待游客上船，耗时：`2303(millis)`

```
等待游客上船，耗时: 8208(millis)
等待游客上船，耗时: 435(millis)
等待游客上船，耗时: 9489(millis)
等待游客上船，耗时: 4937(millis)
等待游客上船，耗时: 2771(millis)
等待游客上船，耗时: 4823(millis)
等待游客上船，耗时: 1989(millis)
等待游客上船，耗时: 8506(millis)
船长急躁了，开船！
```

```
Process finished with exit code 0
```

- 在你实际的测试中会发现，**船长急躁了，开船！**，会需要等待一段时间。
- 这里体现的就是门闩的思想，组队、一波带走。
- CountDownLatch 的实现与 Semaphore 基本相同、细节略有差异，就不再做源码分析了。

三、总结

- 在有了 AQS、CLH、MCS，等相关锁的知识了解后，在学习其他知识点也相对容易。基本以上和前几章节关于锁的介绍，也是面试中容易问到的点。**可能由于目前分布式开发较多，单机的多线程性能压榨一般较少，但是对这部分知识的了解非常重要**
- 得益于 Lee 老爷子的操刀，并发包锁的设计真的非常优秀。每一处的实现都可以说是精益求精，所以在学习的时候可以把小傅哥的文章当作抛砖，之后继续深挖设计精髓，不断深入。
- 共享锁的使用可能平时并不多，但如果需要设计一款类似数据库线程池的设计，那么这样的信号量锁的思想就非常重要了。所以在学习的时候也需要有技术迁移的能力，不断把这些知识复用到实际的业务开发中。

第 4 章 多线程(4 节)

第 1 节：Thread.start() 启动原理

有句话：正因为你优秀，所以难以卓越！

刚开始听这句话还在上学，既不卓越、也不优秀，甚至可能还有点笨！但突然从某次爬到班级的前几名后，开始喜欢上了这种感觉，原来前面的风景是如此灿烂

😊！

优秀和卓越差的不是一个等级，当你感觉自己优秀后，还能保持空瓶的心态开始，才能逐步的像卓越迈进，并漫漫长！

是小时候更容易学会更多的知识，但越大越笨了！人可能很容易被自己的年纪大了，当成长者。却很少能保持一个低姿态谦卑的心态，不断的学习。[所以最后](#)，放不下自己，也拾不起能力。

喜欢一句话，[蓝是天的颜色、红是火的象征，我不学大海抄袭天的蓝、也不学晚霞模拟火的红。我就是我，生命是我的、命运是我的。健身也是你的、学习也是你的，只要你有一个好心态，自然会走到前面卓越那里！](#)

一、面试题

谢飞机，小记！ 码德，年轻人写代码好猖狂，不遵守规范还喷我，你要耗子尾汁！
谢飞机骂骂咧咧的下班后，找面试官聊心得。

谢飞机：我感觉天天就像活在粪堆，代码都是乱糟糟，我有心无力！

面试官：怎么，想跳槽了？

谢飞机：想去写代码有规范的公司，想提升！

面试官：嗯！确实，有些大公司的代码质量要好一些。但是你也要自身能力强的。

谢飞机：是的，我一直在努力学习！准备跑路！

面试官：那我顺便考你个题，看看你进大厂的几率大不。[嗯... Java 线程如何启动的？](#)

谢飞机：如何启动的？[start](#) 启动的！

面试官：还有吗？

谢飞机：嗯...，没了！

面试官：嗯，可能会与不会这个题并不会让你代码有多牛、有多好，但是你的技术栈深度和广度，决定你的编程职业生涯是否有一条康庄大道。还是要多努力！

二、线程启动分析

```
new Thread(() -> {  
    // todo  
}).start();
```

咳咳，Java 的线程创建和启动非常简单，但如果问一个线程是怎么启动起来的往往并不清楚，甚至不知道为什么启动时是调用 `start()`，而不是调用 `run()` 方法呢？

那么，为了让大家有一个更直观的认知，我们先站在上帝视角。把这段 Java 的线程代码，到 JDK 方法使用，以及 JVM 的相应处理过程，展示给大家，以方便我们后续逐步分析。

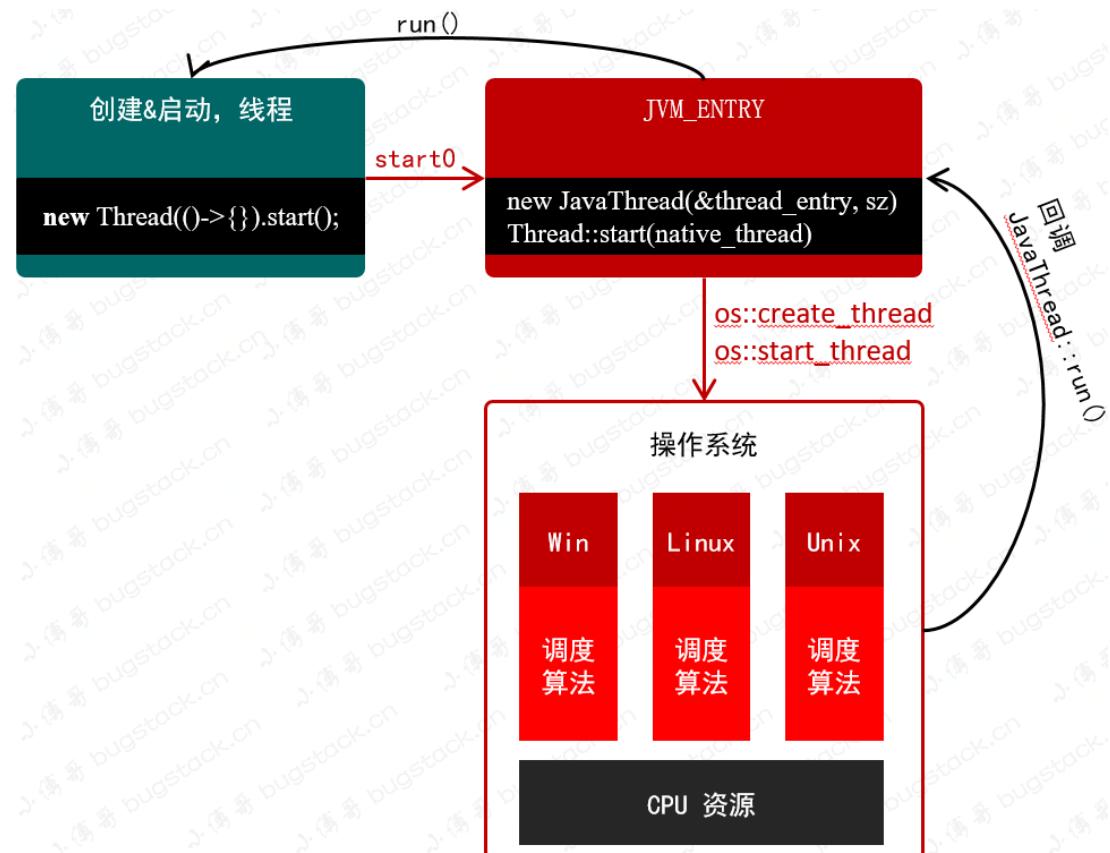


图 19-1 线程启动分析

以上，就是一个线程启动的整体过程分析，会涉及到如下知识点：

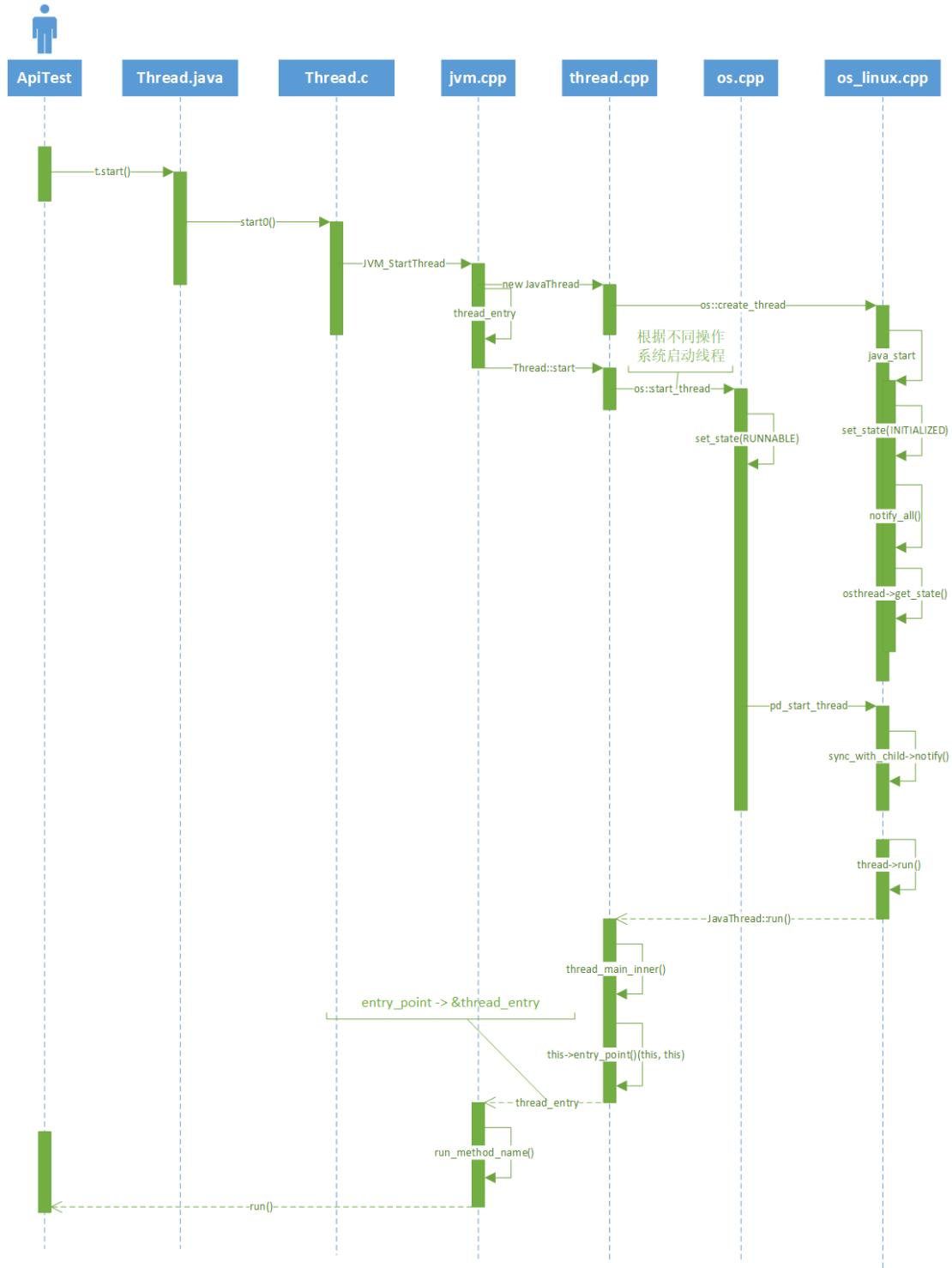
- 线程的启动会涉及到本地方法 (JNI) 的调用，也就是那部分 C++ 编写的代码。
- JVM 的实现中会有不同操作系统对线程的统一处理，比如：Win、Linux、Unix。

- 线程的启动会涉及到线程的生命周期状态 (RUNNABLE)，以及唤醒操作，所以最终会有回调操作。也就是调用我们的 `run()` 方法

接下来，我们就开始逐步分析每一步源码的执行内容，从而了解线程启动过程。

三、线程启动过程

1. Thread start UML 图



Thread start UML 图

如图 19-2 是线程的启动过程时序图，整体的链路较长，会涉及到 JVM 的操作。
核心源码如下：

1. **Thread.c:** <https://github.com/unofficial-openjdk/openjdk/blob/jdk/jdk/src/java.base/share/native/libjava/Thread.c>

2. `jvm.cpp`:
https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/prims/jvm.cpp
3. `thread.cpp`:
https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/runtime/thread.cpp
4. `os.cpp`:
https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/runtime/os.hpp
5. `os_linux.cpp`:
https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/os/linux/vm/os_linux.cpp
6. `os_windows.cpp`:
https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/os/windows/vm/os_windows.cpp
7. `vmSymbols.hpp`:
https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/classfile/vmSymbols.hpp

2. Java 层面 Thread 启动

2.1 start() 方法

```
new Thread(() -> {
    // todo
}).start();

// JDK 源码
public synchronized void start() {

    if (threadStatus != 0)
        throw new IllegalThreadStateException();

    group.add(this);
    boolean started = false;
    try {
        start0();
        started = true;
    }
}
```

```

    } finally {
        try {
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {}
    }
}

```

- 线程启动方法 `start()`, 在它的方法英文注释中已经把核心内容描述出来。
`Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.` 这段话的意思是：由 JVM 调用此线程的 `run` 方法，使线程开始执行。其实这就是一个 JVM 的回调过程，下文源码分析中会讲到
- 另外 `start()` 是一个 `synchronized` 方法，但为了避免多次调用，在方法中会由线程状态判断。`threadStatus != 0`。
- `group.add(this)`, 是把当前线程加入到线程组, `ThreadGroup`。
- `start0()`, 是一个本地方法，通过 JNI 方式调用执行。这一步的操作才是启动线程的核心步骤。

2.2 start0() 本地方法

```

// 本地方法 start0
private native void start0();

// 注册本地方法
public class Thread implements Runnable {
    /* Make sure registerNatives is the first thing <clinit> does. */
    private static native void registerNatives();
    static {
        registerNatives();
    }
    // ...
}

```

- `start0()`, 是一个本地方法，用于启动线程。
- `registerNatives()`, 这个方法是用于注册线程执行过程中需要的一些本地方法，比如：`start0`、`isAlive`、`yield`、`sleep`、`interrupt0` 等。

`registerNatives`, 本地方法定义在 `Thread.c` 中, 以下是定义的核心源码:

```
static JNINativeMethod methods[] = {
    {"start0",           "()V",          (void *)&JVM_StartThread},
    {"stop0",            "(" OBJ ")" V", (void *)&JVM_StopThread},
    {"isAlive",          "()Z",          (void *)&JVM_IsThreadAlive},
    {"suspend0",         "()V",          (void *)&JVM_SuspendThread},
    {"resume0",          "()V",          (void *)&JVM_ResumeThread},
    {"setPriority0",     "(I)V",         (void *)&JVM_SetThreadPriority},
    {"yield",             "()V",          (void *)&JVM_Yield},
    {"sleep",             "()V",          (void *)&JVM_Sleep},
    {"currentThread",    "()" THD,        (void *)&JVM_CurrentThread},
    {"interrupt0",       "()V",          (void *)&JVM_Interrupt},
    {"holdsLock",        "(" OBJ ")" Z", (void *)&JVM_HoldsLock},
    {"getThreads",       "()[" THD,        (void *)&JVM_GetAllThreads},
    {"dumpThreads",      "([" THD ")[[" STE, (void *)&JVM_DumpThreads},
    {"setNativeName",    "(" STR ")" V", (void *)&JVM_SetNativeThreadName},
};

};
```

- 源码: <https://github.com/unofficial-openjdk/openjdk/blob/jdk/jdk/src/java.base/share/native/libjava/Thread.c>
- 从定义中可以看到, `start0` 方法会执行 `&JVM_StartThread` 方法, 最终由 JVM 层面启动线程。

3. JVM 创建线程

3.1 JVM_StartThread

源 码 :

https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/prims/jvm.cpp

```
JVM_ENTRY(void, JVM_StartThread(JNIEnv* env, jobject jthread))
JVMWrapper("JVM_StartThread");
JavaThread *native_thread = NULL;

// 创建线程
native_thread = new JavaThread(&thread_entry, sz);
// 启动线程
Thread::start(native_thread);
```

JVM_END

- 这部分代码比较多，但核心内容主要是[创建线程](#)和[启动线程](#)，另外
`&thread_entry` 也是一个方法，如下：

`thread_entry`, 线程入口

```
static void thread_entry(JavaThread* thread, TRAPS) {
    HandleMark hm(THREAD);
    Handle obj(THREAD, thread->threadObj());
    JavaValue result(T_VOID);
    JavaCalls::call_virtual(&result,
                           obj,
                           KlassHandle(THREAD, SystemDictionary::Thread_klass()),
                           vmSymbols::run_method_name(),
                           vmSymbols::void_method_signature(),
                           THREAD);
}
```

重点，在创建线程引入这个线程入口的方法时，`thread_entry` 中包括了 Java 的回调函数 `JavaCalls::call_virtual`。这个回调函数会由 JVM 调用。

`vmSymbols::run_method_name()`，就是那个被回调的方法，源码如下：

源码：

https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/classfile/vmSymbols.hpp

```
#define VM_SYMBOLS_DO(template, do_alias)
template(run_method_name, "run")
```

- 这个 `run` 就是我们的 Java 程序中会被调用的 `run` 方法。接下来我们继续按照代码执行链路，寻找到这个被回调的方法在什么时候调用的。

3.2 JavaThread

```
native_thread = new JavaThread(&thread_entry, sz);
```

接下来，我们继续看 `JavaThread` 的源码执行内容。

源码:

https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/runtime/thread.cpp

```
JavaThread::JavaThread(ThreadFunction entry_point, size_t stack_sz) :  
    Thread()  
#if INCLUDE_ALL_GCS  
    , _satb_mark_queue(&_satb_mark_queue_set),  
    _dirty_card_queue(&_dirty_card_queue_set)  
#endif // INCLUDE_ALL_GCS  
{  
    if (TraceThreadEvents) {  
        tty->print_cr("creating thread %p", this);  
    }  
    initialize();  
    _jni_attach_state = _not_attaching_via_jni;  
    set_entry_point(entry_point);  
    // Create the native thread itself.  
    // %note runtime_23  
    os::ThreadType thr_type = os::java_thread;  
    thr_type = entry_point == &compiler_thread_entry ? os::compiler_thread :os::java_thread;  
    os::create_thread(this, thr_type, stack_sz);  
}
```

- `ThreadFunction entry_point`, 就是我们上面的 `thread_entry` 方法。
- `size_t stack_sz`, 表示进程中已有的线程个数。
- 这两个参数, 都会传递给 `os::create_thread` 方法, 用于创建线程使用。

3.3 os::create_thread

源码:

- `os_linux.cpp`:
https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/os/linux/vm/os_linux.cpp
- `os_windows.cpp`:
https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/os/windows/vm/os_windows.cpp

众所周知，JVM 是个啥！，所以它的 OS 服务实现，Liunx 还有 Windows 等，都会实现线程的创建逻辑。这有点像适配器模式

os_linux -> os::create_thread

```
bool os::create_thread(Thread* thread, ThreadType thr_type, size_t stack_size) {
    assert(thread->osthread() == NULL, "caller responsible");

    // Allocate the OSThread object
    OSThread* osthread = new OSThread(NULL, NULL);
    // Initial state is ALLOCATED but not INITIALIZED
    osthread->set_state(ALLOCATED);

    pthread_t tid;
    int ret = pthread_create(&tid, &attr, (void* (*)(void*)) java_start, thread);

    return true;
}
```

- `osthread->set_state(ALLOCATED)`, 初始化已分配的状态，但此时并没有初始化。
- `pthread_create`, 是类 Unix 操作系统 (Unix、Linux、Mac OS X 等) 的创建线程的函数。
- `java_start`, 重点关注类，是实际创建线程的方法。

3.4 java_start

源码：

https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/os/linux/vm/os_linux.cpp

```
static void *java_start(Thread *thread) {

    // 线程 ID
    int pid = os::current_process_id();

    // 设置线程
    ThreadLocalStorage::set_thread(thread);

    // 设置线程状态: INITIALIZED 初始化完成
    osthread->set_state(INITIALIZED);
```

```

// 唤醒所有线程
sync->notify_all();

// 循环，初始化状态，则一致等待 wait
while (osthread->get_state() == INITIALIZED) {
    sync->wait(Mutex::no_safepoint_check_flag);
}

// 等待唤醒后，执行 run 方法
thread->run();

return 0;
}

```

- JVM 设置线程状态，INITIALIZED 初始化完成。
- `sync->notify_all()`，唤醒所有线程。
- `osthread->get_state() == INITIALIZED`, while 循环等待
- `thread->run()`，是等待线程唤醒后，也就是状态变更后，才能执行到。这在我们的线程执行 UML 图中，也有所体现

4. JVM 启动线程

```

JVM_ENTRY(void, JVM_StartThread(JNIEnv* env, jobject jthread))
JVMWrapper("JVM_StartThread");
JavaThread *native_thread = NULL;

// 创建线程
native_thread = new JavaThread(&thread_entry, sz);
// 启动线程
Thread::start(native_thread);

JVM_END

```

- `JVM_StartThread` 中有两步，创建 (`new JavaThread`)、启动 (`Thread::start`)。创建的过程聊完了，接下来我们聊启动。

4.1 Thread::start

源码:

https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/runtime/thread.cpp

```
void Thread::start(Thread* thread) {
    trace("start", thread);

    if (!DisableStartThread) {
        if (thread->is_Java_thread()) {
            java_lang_Thread::set_thread_status(((JavaThread*)thread)->threadObj(),
                java_lang_Thread::RUNNABLE);
        }
        // 不同的 OS 会有不同的启动代码逻辑
        os::start_thread(thread);
    }
}
```

- 如果没有禁用线程 `DisableStartThread` 并且是 Java 线程 `thread->is_Java_thread()`, 那么设置线程状态为 `RUNNABLE`。
- `os::start_thread(thread)`, 调用线程启动方法。不同的 OS 会有不同的启动代码逻辑

4.2 os::start_thread(thread)

源码:

https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/runtime/os.hpp

```
void os::start_thread(Thread* thread) {
    // guard suspend/resume
    MutexLockerEx ml(thread->SR_lock(), Mutex::_no_safepoint_check_flag);
    OSThread* osthread = thread->osthread();
    osthread->set_state(RUNNABLE);
    pd_start_thread(thread);
}
```

- `osthread->set_state(RUNNABLE)`, 设置线程状态 `RUNNABLE`
- `pd_start_thread(thread)`, 启动线程, 这个就由各个 OS 实现类, 实现各自系统的启动方法了。比如, `windows` 系统和 `Linux` 系统的代码是完全不同的。

4.3 pd_start_thread(thread)

源码:

https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/os/linux/vm/os_linux.cpp

```
void os::pd_start_thread(Thread* thread) {
    OSThread * osthread = thread->osthread();
    assert(osthread->get_state() != INITIALIZED, "just checking");
    Monitor* sync_with_child = osthread->startThread_lock();
    MutexLockerEx ml(sync_with_child, Mutex::_no_safepoint_check_flag);
    sync_with_child->notify();
}
```

- 这部分代码 `notify()` 最关键，它可以唤醒线程。
- 线程唤醒后，3.4 中的 `thread->run()` 就可以继续执行了。

5. JVM 线程回调

5.1 thread->run() [JavaThread::run()]

源码:

https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/runtime/thread.cpp

```
// The first routine called by a new Java thread
void JavaThread::run() {
    // ... 初始化线程操作

    thread_main_inner();
}
```

- `os_linux.cpp` 类中的 `java_start` 里的 `thread->run()`，最终调用的就是 `thread.cpp` 的 `JavaThread::run()` 方法。
- 这部分还需要继续往下看，`thread_main_inner()` 方法。

5.2 thread_main_inner

源码：

https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/runtime/thread.cpp

```
void JavaThread::thread_main_inner() {  
  
    if (!this->has_pending_exception() &&  
        !java_lang_Thread::is_stillborn(this->threadObj())) {  
  
        {  
            ResourceMark rm(this);  
            this->set_native_thread_name(this->get_thread_name());  
        }  
        HandleMark hm(this);  
        this->entry_point()(this, this);  
    }  
  
    DTRACE_THREAD_PROBE(stop, this);  
  
    this->exit(false);  
    delete this;  
}
```

- 这里有你熟悉的设置的线程名称，
`this->set_native_thread_name(this->get_thread_name())。`
- `this->entry_point()`, 实际调用的就是 3.1 中的 `thread_entry` 方法。
- `thread_entry`, 方法最终会调用到 `JavaCalls::call_virtual` 里的
`vmSymbols::run_method_name()`。也就是 `run()` 方法, 至此线程启动完成。终于串回来了!

四、总结

- 线程的启动过程涉及到了 JVM 的参与, 所以如果没有认真了解过, 确实很难从一个本地方法了解的如此透彻。
- 整个源码分析可以结合着代码调用 UML 时序图进行学习, 基本核心过程包括:

`Java` 创建线程和启动、调用本地方法 `start0()`、`JVM` 中
`JVM_StartThread` 的创建和启动、设置线程状态等待被唤醒、根据不同的 OS 启动线程并唤醒、最后回调 `run()` 方法启动 `Java` 线程。

- 有时候可能只是一步很简单的方法，也会有它的深入之处，当真的懂了以后，就不用死记硬背。如果需要获得以上高清大图，可以添加小傅哥微信([fustack](#))，备注：*Thread 大图*

第 2 节：Thread，状态转换、方法使用、原理分析

考不常用的、考你不会的、考你忽略的，才是考试！

大部分考试考的，基本都是不怎么用的。例外的咱们不说^② 就像你做程序开发，尤其在 RPC+MQ+分库分表，其实很难出现让你用一个机器实例编写多线程压榨 CPU 性能。很多时候是扔出一个 MQ，异步消费了。如果没有资源竞争，例如库表秒杀，那么其实你确实很难接触多并发编程以及锁的使用。

但！凡有例外，比如你需要开发一个数据库路由中间件，那么就肯定会出现一台应用实例上分配数据库资源池的情况，如果出现竞争就要合理分配资源。如此，类似这样的中间件开发，就会涉及到一些更核心底层的技术的应用。

所以，有时候不是没用，而是你没有用。

一、面试题

谢飞机，小记！ 线程我玩定了，面试也拦不住我，我说的！

谢飞机：嘿，你好哇，我是谢飞机！

面试官：好，今天电话面试，你准备好了？

谢飞机：准备好了，嘿嘿！

面试官：嗯，我看你简历里写了不少线程的东西，看来了解的不错。问你一个线程吧那就，线程之间状态是怎么转换的？

谢飞机：扒拉扒拉，扒拉扒拉！

面试官：嗯，还不错。那 yield 方法是怎么使用的。

谢飞机：嗯！好像是让出 CPU。具体的没怎么用过！

面试官：做做测试，验证下，下次问你。

二、Thread 状态关系

Java 的线程状态描述在枚举类 `java.lang.Thread.State` 中，共包括如下五种状态：

```
public enum State {  
    NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED;  
}
```

这五种状态描述了一个线程的生命周期，其实这种状态码的定义在我们日常的业务开发中，也经常出现。比如：一个活动的提交、审核、拒绝、修改、通过、运行、关闭等，是类似的。那么线程的状态是通过下图的方式进行流转的，如图 20-1

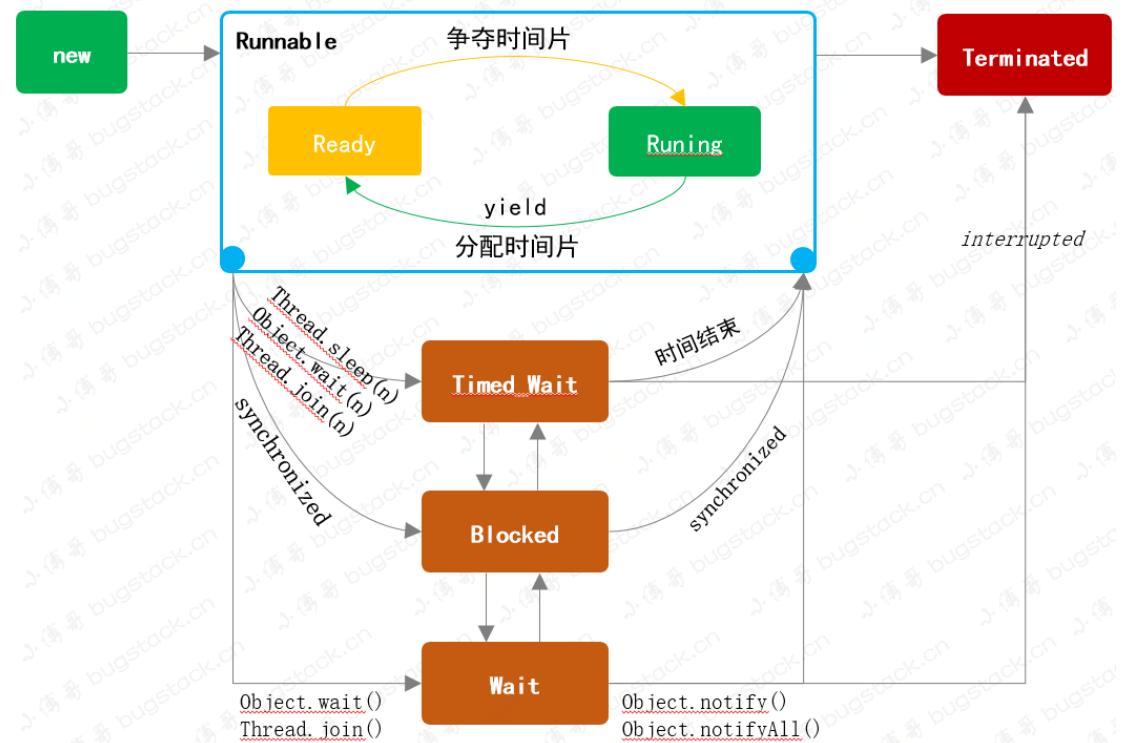


图 20-1 线程状态流转

- **New**: 新创建的一个线程，处于等待状态。
- **Runnable**: 可运行状态，并不是已经运行，具体的线程调度各操作系统决定。在 **Runnable** 中包含了 **Ready**、**Running** 两个状态，当线程调用了 **start()** 方法后，线程则处于就绪 **Ready** 状态，等待操作系统分配 CPU 时间片，分配后则进入 **Running** 运行状态。此外当调用 **yield()** 方法后，只是谦让的允许当前线程让出 CPU，但具体让不让不一定，由操作系统决定。如果让了，那么当前线程则会处于 **Ready** 状态继续竞争 CPU，直至执行。
- **Timed_waiting**: 指定时间内让出 CPU 资源，此时线程不会被执行，也不会被系统调度，直到等待时间到期后才会被执行。下列方法都可以触发：
Thread.sleep、**Object.wait**、**Thread.join**、
LockSupport.parkNanos、**LockSupport.parkUntil**。
- **Waiting**: 可被唤醒的等待状态，此时线程不会被执行也不会被系统调度。此状态可以通过 **synchronized** 获得锁，调用 **wait** 方法进入等待状态。最后通过 **notify**、**notifyall** 唤醒。下列方法都可以触发：**Object.wait**、
Thread.join、**LockSupport.park**。

- **Blocked**: 当发生锁竞争状态下，没有获得锁的线程会处于挂起状态。例如 `synchronized` 锁，先获得的先执行，没有获得的进入阻塞状态。
- **Terminated**: 这个是终止状态，从 `New` 到 `Terminated` 是不可逆的。一般是程序流程正常结束或者发生了异常。

这里参考枚举 `State` 类的英文注释了解了每一个状态码的含义，接下来我们去尝试操作线程方法，把这些状态体现出来。

三、Thread 状态测试

1. NEW

```
Thread thread = new Thread(() -> {  
});  
System.out.println(thread.getState());  
  
// NEW
```

- 这个状态很简单，就是线程创建还没有启动时就是这个状态。

2. RUNNABLE

```
Thread thread = new Thread(() -> {  
});  
// 启动  
thread.start();  
System.out.println(thread.getState());  
  
// RUNNABLE
```

- 创建的线程启动后 `start()`，就会进入 `RUNNABLE` 状态。但此时并不一定在执行，而是说这个线程已经就绪，可以竞争 CPU 资源。

3. BLOCKED

```
Object obj = new Object();  
new Thread(() -> {  
    synchronized (obj) {  
        try {  
            Thread.sleep(10000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
});  
System.out.println(thread.getState());  
// BLOCKED
```

```

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}).start();

Thread thread = new Thread(() -> {
    synchronized (obj) {
        try {
            obj.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

thread.start();
while (true) {
    Thread.sleep(1000);
    System.out.println(thread.getState());
}
}

// BLOCKED
// BLOCKED
// BLOCKED

```

- 这段代码稍微有点长，主要是为了让两个线程发生锁竞争。
- 第一个线程，synchronized 获取锁后休眠，不释放锁。
- 第二个线程，synchronized 获取不到锁，会被挂起。
- 那么最后的输出结果就会是，BLOCKED

4. WAITING

```

Object obj = new Object();
Thread thread = new Thread(() -> {
    synchronized (obj) {
        try {
            obj.wait();
        } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    }
}

});

thread.start();

while (true) {
    Thread.sleep(1000);
    System.out.println(thread.getState());
}

```

// WAITING
// WAITING
// WAITING

- 只要在 synchronized 代码块或者修饰的方法中，调用 `wait` 方法，又没有被 `notify` 就会进入 **WAITING** 状态。
- 另外 `Thread.join` 源码中也是调用的 `wait` 方法，所以也会让线程进入等待状态。

5. TIMED_WAITING

```

Object obj = new Object();
Thread thread = new Thread(() -> {
    synchronized (obj) {
        try {
            Thread.sleep(100000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

thread.start();

while (true) {
    Thread.sleep(1000);
    System.out.println(thread.getState());
}

```

```
// TIMED_WAITING  
// TIMED_WAITING  
// TIMED_WAITING
```

- 有了上面状态获取的对比，这个状态的获取就没什么难度了。只要改成 `Thread.sleep(100000);` 就可以了。

6. TERMINATED

```
Thread thread = new Thread(() -> {  
});  
thread.start();  
  
System.out.println(thread.getState());  
System.out.println(thread.getState());  
System.out.println(thread.getState());  
  
// RUNNABLE  
// TERMINATED  
// TERMINATED
```

- 这个就比较简单了，只要一个线程运行完，它的生命周期结束了，就进入了 `TERMINATED` 状态。

四、Thread 方法使用

一般情况下 `Thread` 中最常用的方法就是 `start` 启动，除此之外一些其他方法可能在平常的开发中用的不多，但这些方法在一些框架中却经常出现。因此只了解它们的概念，但是却缺少一些实例来参考！接下来我们就来做一些案例来验证这些方法，包括：`yield`、`wait`、`notify`、`join`。

1. yield

`yield` 方法让出 CPU，但不一定，一定让出！。这种可能会用在一些同时启动的线程中，按照优先级保证重要线程的执行，也可以是其他一些特殊的业务场景（例如这个线程内容很耗时，又不那么重要，可以放在后面）。

为了验证这个方法，我们做一个例子：启动 50 个线程进行，每个线程都进行 1000 次的加和计算。其中 10 个线程会执行让出 CPU 操作。那么，如果让出 CPU 那 10 个线程的计算加和时间都比较长，说明确实在进行让出操作。

案例代码

```
private static volatile Map<String, AtomicInteger> count = new ConcurrentHashMap<>();

static class Y implements Runnable {

    private String name;

    private boolean isYield;

    public Y(String name, boolean isYield) {
        this.name = name;
        this.isYield = isYield;
    }

    @Override
    public void run() {
        long l = System.currentTimeMillis();
        for (int i = 0; i < 1000; i++) {
            if (isYield) Thread.yield();
            AtomicInteger atomicInteger = count.get(name);
            if (null == atomicInteger) {
                count.put(name, new AtomicInteger(1));
                continue;
            }
            atomicInteger.addAndGet(1);
            count.put(name, atomicInteger);
        }
        System.out.println("线程编号：" + name + " 执行完成耗时：" +
" + (System.currentTimeMillis() - l) + " (毫秒)" + (isYield ? "让出 CPU-----" :
-----" : "不让 CPU"));
    }
}

public static void main(String[] args) {
    for (int i = 0; i < 50; i++) {
        if (i < 10) {
            new Thread(new Y(String.valueOf(i), true)).start();
            continue;
        }
    }
}
```

```
        }

        new Thread(new Y(String.valueOf(i), false)).start();
    }

}
```

测试结果

线程编号: 10 执行完成耗时: 2 (毫秒)不让 CPU
线程编号: 11 执行完成耗时: 2 (毫秒)不让 CPU
线程编号: 15 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 14 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 19 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 18 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 22 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 26 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 27 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 30 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 31 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 34 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 12 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 16 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 13 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 17 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 20 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 23 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 21 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 25 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 24 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 28 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 38 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 39 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 37 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 40 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 44 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 36 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 42 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 45 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 43 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 46 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 47 执行完成耗时: 0 (毫秒)不让 CPU

线程编号: 35 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 33 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 32 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 41 执行完成耗时: 0 (毫秒)不让 CPU
线程编号: 48 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 6 执行完成耗时: 15 (毫秒)让出 CPU-----
线程编号: 7 执行完成耗时: 15 (毫秒)让出 CPU-----
线程编号: 49 执行完成耗时: 2 (毫秒)不让 CPU
线程编号: 29 执行完成耗时: 1 (毫秒)不让 CPU
线程编号: 2 执行完成耗时: 17 (毫秒)让出 CPU-----
线程编号: 1 执行完成耗时: 11 (毫秒)让出 CPU-----
线程编号: 4 执行完成耗时: 15 (毫秒)让出 CPU-----
线程编号: 8 执行完成耗时: 12 (毫秒)让出 CPU-----
线程编号: 5 执行完成耗时: 12 (毫秒)让出 CPU-----
线程编号: 9 执行完成耗时: 12 (毫秒)让出 CPU-----
线程编号: 0 执行完成耗时: 21 (毫秒)让出 CPU-----
线程编号: 3 执行完成耗时: 21 (毫秒)让出 CPU-----

- 从测试结果可以看到，那些让出 CPU 的，执行完计算已经在 10 毫秒以上，说明我们的测试是效果的。

2. wait & notify

wait 和 notify/notifyall，是一对方法，有一个等待，就会有一个叫醒，否则程序就停在那不动了。关于这部分会使用到的 `synchronized` 在之前小傅哥有深入的源码分析，讲到它是怎么加锁在对象头的，如果你忘记了可以往前翻翻。

接下来我们模拟鹿鼎记·丽春院，清倌喝茶吟诗聊风月日常。当有达官贵人来时，需要分配清倌给大老爷。中间会有一些等待、叫醒操作。只为让你更好的记住这样的案例，不要想歪喽。清倌人即是只卖艺欢场人，喊麦的。

案例代码

```
public class 丽春院 {  
  
    public static void main(String[] args) {  
        老鸨 鸽子 = new 老鸨();  
  
        清倌 miss = new 清倌(鸽子);  
    }  
}
```

```

    客官 guest = new 客官(鴇子);

    Thread t_miss = new Thread(miss);
    Thread t_guest = new Thread(guest);

    t_miss.start();
    t_guest.start();
}

}

class 清倌 implements Runnable {

    老鴇 鴇子;

    public 清倌(老鴇 鴇子) {
        this.鴇子 = 鴇子;
    }

    @Override
    public void run() {
        int i = 1;
        while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }
            if (i == 1) {
                try {
                    鴇子.在岗清倌("苍田野子", "500 日元");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } else {
                try {
                    鴇子.在岗清倌("花田岗子", "800 日元");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

```
        }
    }
    i = (i + 1) % 2;
}
}

}

class 客官 implements Runnable {
    老鸨 鸽子;

    public 客官(老鸨 鸽子) {
        this.鸽子 = 鸽子;
    }

    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }
            try {
                鸽子.喝茶吟诗聊风月();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class 老鸨 {
    private String 清倌 = null;
    private String price = null;
    private boolean 工作状态 = true;
}
```

302 / 417

小傅哥，公众号：bugstack 虫洞栈 | 沉淀、分享、成长，让自己和他人都能有所收获

```

public synchronized void 在岗清信(String 清
信, String price) throws InterruptedException {
    if (!工作状态)
        wait(); // 等待
    this.清信 = 清信;
    this.price = price;
    工作状态 = false;
    notify(); // 叫醒
}

public synchronized void 喝茶吟诗聊风月() throws InterruptedException {
    if (工作状态)
        wait(); // 等待
    System.out.println("聊风月: " + 清信);
    System.out.println("茶水费: " + price);
    System.out.println("    " + "    " + "    " + "    " + "    " + "    " + "    " + "    " +
"    " + "    " + 清信 + "完事" + "准备 ... ...");
    System.out.println("*****");
    工作状态 = true;
    notify(); // 叫醒
}

}

```

测试结果

聊风月: 苍田野子
茶水费: 500 日元
苍田野子完事准备

聊风月: 花田岗子
茶水费: 800 日元
花田岗子完事准备

聊风月: 苍田野子
茶水费: 500 日元
苍田野子完事准备

...

- 效果主要体现 `wait`、`notify`，这两个方法的使用。我相信你一定能记住这个例子！

3. join

`join` 是两个线程的合并吗？不是的！

`join` 是让线程进入 `wait`，当线程执行完毕后，会在 JVM 源码中找到，它执行完毕后，其实执行 `notify`，也就是 等待 和 叫醒 操作。

源码：[jdk8u hotspot/blob/master/src/share/vm/runtime/thread.cpp](https://github.com/AdoptOpenJDK/jdk8u/blob/master/src/share/vm/runtime/thread.cpp)

```
void JavaThread::exit(bool destroy_vm, ExitType exit_type) {
    // Notify waiters on thread object. This has to be done after exit() is called
    // on the thread (if the thread is the last thread in a daemon ThreadGroup the
    // group should have the destroyed bit set before waiters are notified).
    ensure_join(this);
}

static void ensure_join(JavaThread* thread) {
    // 叫醒
    java_lang_Thread::set_thread(threadObj(), NULL);
    lock.notify_all(thread);
}
```

好的，就是这里！`lock.notify_all(thread)`，执行到这，就对上了。

案例代码

```
Thread thread = new Thread(() -> {
    System.out.println("thread before");
    try {
        Thread.sleep(3000);
    } catch (Exception e) {
        e.printStackTrace();
    }
    System.out.println("thread after");
});
thread.start();
System.out.println("main begin! ");
```

```
thread.join();
System.out.println("main end!");
```

测试结果

```
main begin!
thread before
thread after
main end!

Process finished with exit code 0
```

首先 `join()` 是一个 `synchronized` 方法，里面调用了 `wait()`，这个过程的目的是让持有这个同步锁的线程进入等待，那么谁持有了这个同步锁呢？答案是主线程，因为主线程调用了 `threadA.join()` 方法，相当于在 `threadA.join()` 代码这块写了一个同步代码块，谁去执行了这段代码呢，是主线程，所以主线程被 `wait()` 了。然后在子线程 `threadA` 执行完毕之后，JVM 会调用 `lock.notify_all(thread)`；唤醒持有 `threadA` 这个对象锁的线程，也就是主线程，会继续执行。

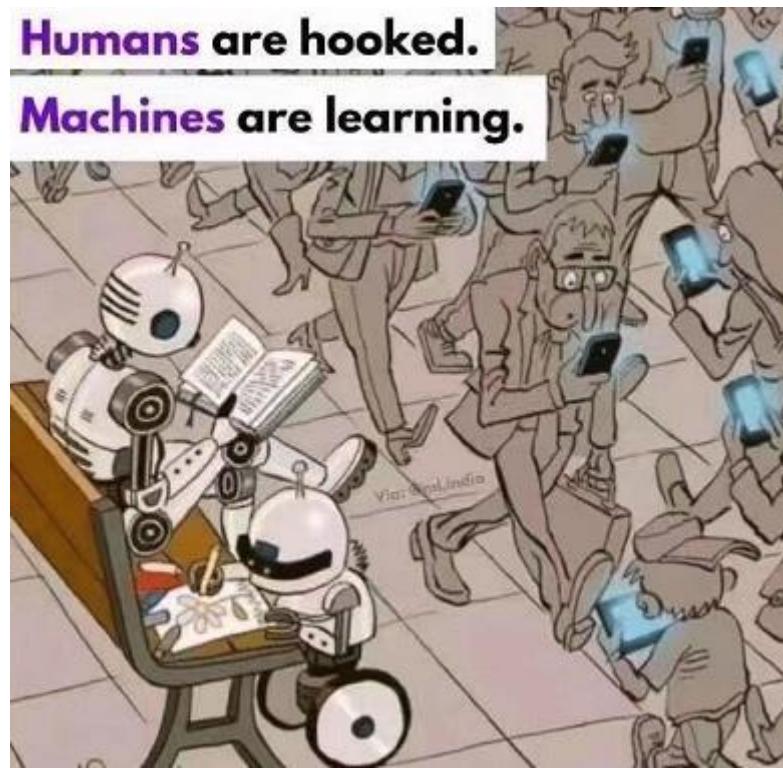
- 这部分验证的主要体现就是加了 `thread.join()` 后，会影响到输出结果。如果不去加，`main end!` 会优先 `thread after` 提前打印出来。
- `join()` 是一个 `synchronized` 方法，里面调用了 `wait()` 方法，让持有当前同步锁的线程进入等待状态，也就是主线程。当子线程执行完毕后，我们从源码中可以看到 JVM 调用了 `lock.notify_all(thread)` 所以唤醒了主线程继续执行。

五、总结

- 线程状态和状态的转换也是面试中必问的问题，但除了面试是我们自己在开发中，如果真的使用线程，是非常有必要了解线程状态是如何转换的。*模模糊糊的使用，总会觉得担心，那么你是个好程序员！*
- 线程的一些深入学习都是在调用本地方法，也就是需要了解到 JVM 层面，才能更加深刻的见到 c++ 代码是如何实现这部分逻辑的。
- 在使用线程的时候一定要让自己有一个类似多核的脑子，*线程一起、生死由你！*本章节就扯到这了，很多的知识都是为了整套内容体系的全面，为后续介绍其他知识打下根基。感谢！

第 3 节：ThreadPoolExecutor

人看手机，机器学习！



正好是 2020 年，看到这张图还是蛮有意思的。以前小时候总会看到一些科技电影，讲到机器人会怎样怎样，但没想到人似乎被娱乐化的东西，搞成了低头族、大肚子！

当意识到这一点时，其实非常怀念小时候。放假的早上跑出去，喊上三五个伙伴，要不下河摸摸鱼、弹弹玻璃球、打打 pia、跳跳房子！一天下来真的不会感觉累，但现在如果是放假的一天，你的娱乐安排，很多时候会让头很累！

就像，你有试过学习一天英语头疼，还是刷一天抖音头疼吗？或者玩一天游戏与打一天球！[如果你意识到了，那么争取放下一会儿手机，适当娱乐，锻炼保持个好身体！](#)

一、面试题

[谢飞机，小记！](#)，上次吃亏在线程上，这次可能一次坑掉两次了！

谢飞机：你问吧，我准备好了！！！

面试官：嗯，线程池状态是如何设计存储的？

谢飞机：这！下一个，下一个！

面试官：Worker 的实现类，为什么不使用 ReentrantLock 来实现呢，而是自己继承 AQS？

谢飞机：我...！

面试官：那你简述下，execute 的执行过程吧！

谢飞机：再见！

二、线程池讲解

1. 先看个例子

```
ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(10, 10, 0L, TimeUnit
.MILLISCONDS, new ArrayBlockingQueue<>(10));
threadPoolExecutor.execute(() -> {
    System.out.println("Hi 线程池！");
});
threadPoolExecutor.shutdown();

// Executors.newFixedThreadPool(10);
// Executors.newCachedThreadPool();
// Executors.newScheduledThreadPool(10);
// Executors.newSingleThreadExecutor();
```

这是一段用于创建线程池的例子，相信你已经用了很多次了。

线程池的核心目的就是资源的利用，避免重复创建线程带来的资源消耗。因此引入一个池化技术的思想，避免重复创建、销毁带来的性能开销。

那么，接下来我们就通过实践的方式分析下这个池子的构造，看看它是如何处理线程的。

2. 手写一个线程池

2.1 实现流程

为了更好的理解和分析关于线程池的源码，我们先来按照线程池的思想，手写一个非常简单的线程池。

其实很多时候一段功能代码的核心主逻辑可能并没有多复杂，但为了让核心流程顺利运行，就需要额外添加很多分支的辅助流程。就像我常说的，为了保护手才把擦屁屁纸弄那么大！

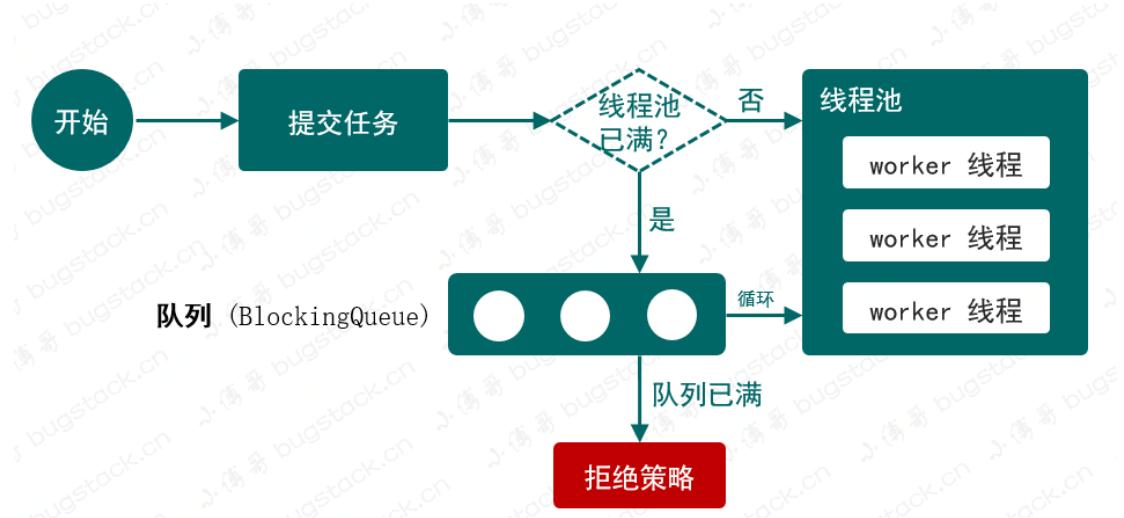


图 21-1 线程池简化流程

关于图 21-1，这个手写线程池的实现也非常简单，只会体现出核心流程，包括：

1. 有 n 个一直在运行的线程，相当于我们创建线程池时允许的线程池大小。
2. 把线程提交给线程池运行。
3. 如果运行线程池已满，则把线程放入队列中。
4. 最后当有空闲时，则获取队列中线程进行运行。

2.2 实现代码

```

public class ThreadPoolTrader implements Executor {

    private final AtomicInteger ctl = new AtomicInteger(0);

    private volatile int corePoolSize;
    private volatile int maximumPoolSize;

    private final BlockingQueue<Runnable> workQueue;

    public ThreadPoolTrader(int corePoolSize, int maximumPoolSize, BlockingQueue<Ru
nnable> workQueue) {
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
    }

    @Override
    public void execute(Runnable command) {

```

```

        int c = ctl.get();
        if (c < corePoolSize) {
            if (!addWorker(command)) {
                reject();
            }
            return;
        }
        if (!workQueue.offer(command)) {
            if (!addWorker(command)) {
                reject();
            }
        }
    }

private boolean addWorker(Runnable firstTask) {
    if (ctl.get() >= maximumPoolSize) return false;

    Worker worker = new Worker(firstTask);
    worker.thread.start();
    ctl.incrementAndGet();
    return true;
}

private final class Worker implements Runnable {

    final Thread thread;
    Runnable firstTask;

    public Worker(Runnable firstTask) {
        this.thread = new Thread(this);
        this.firstTask = firstTask;
    }

    @Override
    public void run() {
        Runnable task = firstTask;
        try {
            while (task != null || (task = getTask()) != null) {
                task.run();
            }
        } catch (Exception e) {
            handleException(e);
        }
    }

    private void handleException(Throwable e) {
        if (e instanceof RejectedExecutionException) {
            // ...
        } else {
            logger.error("Unexpected exception in worker thread", e);
        }
    }
}

```

```

        if (ctl.get() > maximumPoolSize) {
            break;
        }
        task = null;
    }
} finally {
    ctl.decrementAndGet();
}
}

private Runnable getTask() {
    for ( ; ; ) {
        try {
            System.out.println("workQueue.size: " + workQueue.size());
            return workQueue.take();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

private void reject() {
    throw new RuntimeException("Error! ctl.count:
" + ctl.get() + " workQueue.size: " + workQueue.size());
}

public static void main(String[] args) {
    ThreadPoolTrader threadPoolTrader = new ThreadPoolTrader(2, 2, new ArrayBlockingQueue<Runnable>(10));

    for (int i = 0; i < 10; i++) {
        int finalI = i;
        threadPoolTrader.execute(() -> {
            try {
                Thread.sleep(1500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
    }
}

```

```
        System.out.println("任务编号: " + finalI);
    });
}
}

}

// 测试结果
```

```
任务编号: 1
任务编号: 0
workQueue.size: 8
workQueue.size: 8
任务编号: 3
workQueue.size: 6
任务编号: 2
workQueue.size: 5
任务编号: 5
workQueue.size: 4
任务编号: 4
workQueue.size: 3
任务编号: 7
workQueue.size: 2
任务编号: 6
workQueue.size: 1
任务编号: 8
任务编号: 9
workQueue.size: 0
workQueue.size: 0
```

以上，关于线程池的实现还是非常简单的，从测试结果上已经可以把最核心的池化思想体现出来了。主要功能逻辑包括：

- `ctl`，用于记录线程池中线程数量。
- `corePoolSize`、`maximumPoolSize`，用于限制线程池容量。
- `workQueue`，线程池队列，也就是那些还不能被及时运行的线程，会被装入到这个队列中。
- `execute`，用于提交线程，这个是通用的接口方法。在这个方法里主要实现的就是，当前提交的线程是加入到 worker、队列还是放弃。

- `addWorker`, 主要是类 `Worker` 的具体操作, 创建并执行线程。这里还包括了 `getTask()` 方法, 也就是从队列中不断的获取未被执行的线程。

好, 那么以上呢, 就是这个简单线程池实现的具体体现。但如果深思熟虑就会发现这里需要很多完善, 比如: [线程池状态呢, 不可能一直奔跑呀! ?](#)、[线程池的锁呢, 不会有并发问题吗?](#)、[线程池拒绝后的策略呢?](#), 这些问题都没有在主流程解决, 也正因为没有这些流程, 所以上面的代码才更容易理解。

接下来, 我们就开始分析线程池的源码, 与我们实现的简单线程池参考对比, 会更加容易理解⑩!

3. 线程池源码分析

3.1 线程池类关系图

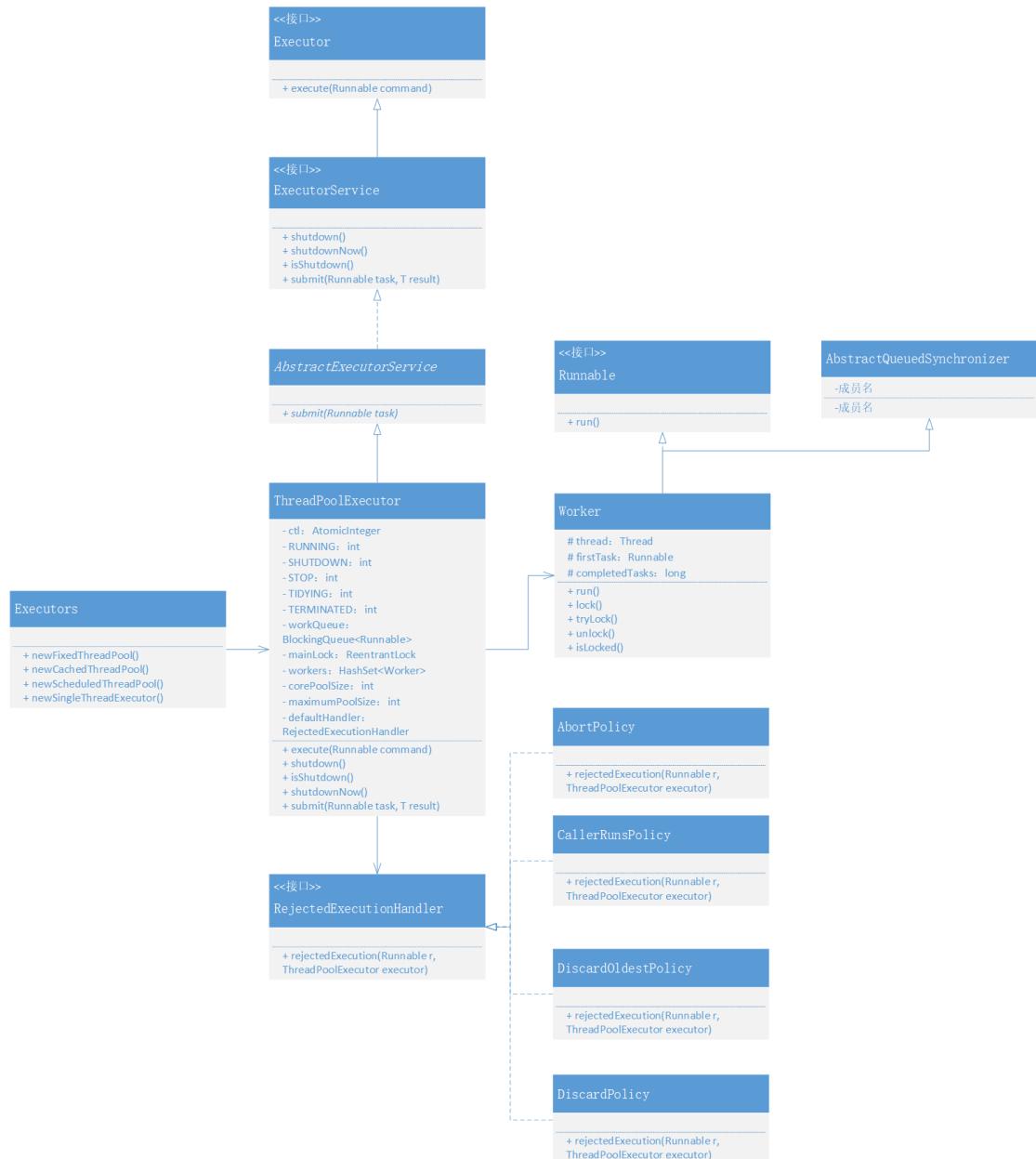


图 21-2 线程池类关系图

以围绕核心类 `ThreadPoolExecutor` 的实现展开的类之间实现和继承关系，如图 21-2 线程池类关系图。

- 接口 `Executor`、`ExecutorService`，定义线程池的基本方法。尤其是 `execute(Runnable command)` 提交线程池方法。
- 抽象类 `AbstractExecutorService`，实现了基本通用的接口方法。
- `ThreadPoolExecutor`，是整个线程池最核心的工具类方法，所有的其他类和接口，为围绕这个类来提供各自的功能。
- `Worker`，是任务类，也就是最终执行的线程的方法。

- `RejectedExecutionHandler`, 是拒绝策略接口, 有四个实现类;
`AbortPolicy`(抛异常方式拒绝)、`DiscardPolicy`(直接丢弃)、
`DiscardOldestPolicy`(丢弃存活时间最长的任务)、
`CallerRunsPolicy`(谁提交谁执行)。
- `Executors`, 是用于创建我们常用的不同策略的线程池,
`newFixedThreadPool`、`newCachedThreadPool`、
`newScheduledThreadPool`、`newSingleThreadExecutor`。

3.2 高 3 位与低 29 位

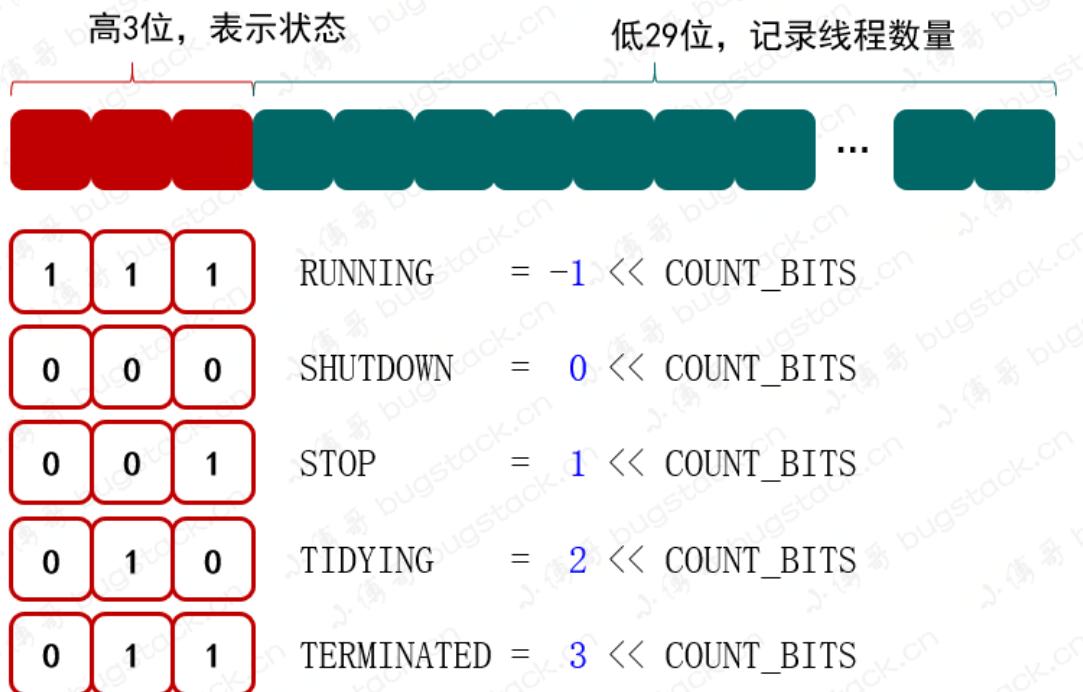


图 22-3 线程状态, 高 3 位与低 29 位

```

private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
private static final int COUNT_BITS = Integer.SIZE - 3;
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;

private static final int RUNNING   = -1 << COUNT_BITS;
private static final int SHUTDOWN  = 0 << COUNT_BITS;
private static final int STOP      = 1 << COUNT_BITS;
private static final int TIDYING   = 2 << COUNT_BITS;
private static final int TERMINATED= 3 << COUNT_BITS;

```

在 `ThreadPoolExecutor` 线程池实现类中，使用 `AtomicInteger` 类型的 `ctl` 记录线程池状态和线程池数量。在一个类型上记录多个值，它采用的分割数据区域，高 3 位记录状态，低 29 位存储线程数量，默认 `RUNNING` 状态，线程数为 0 个。

3.2 线程池状态

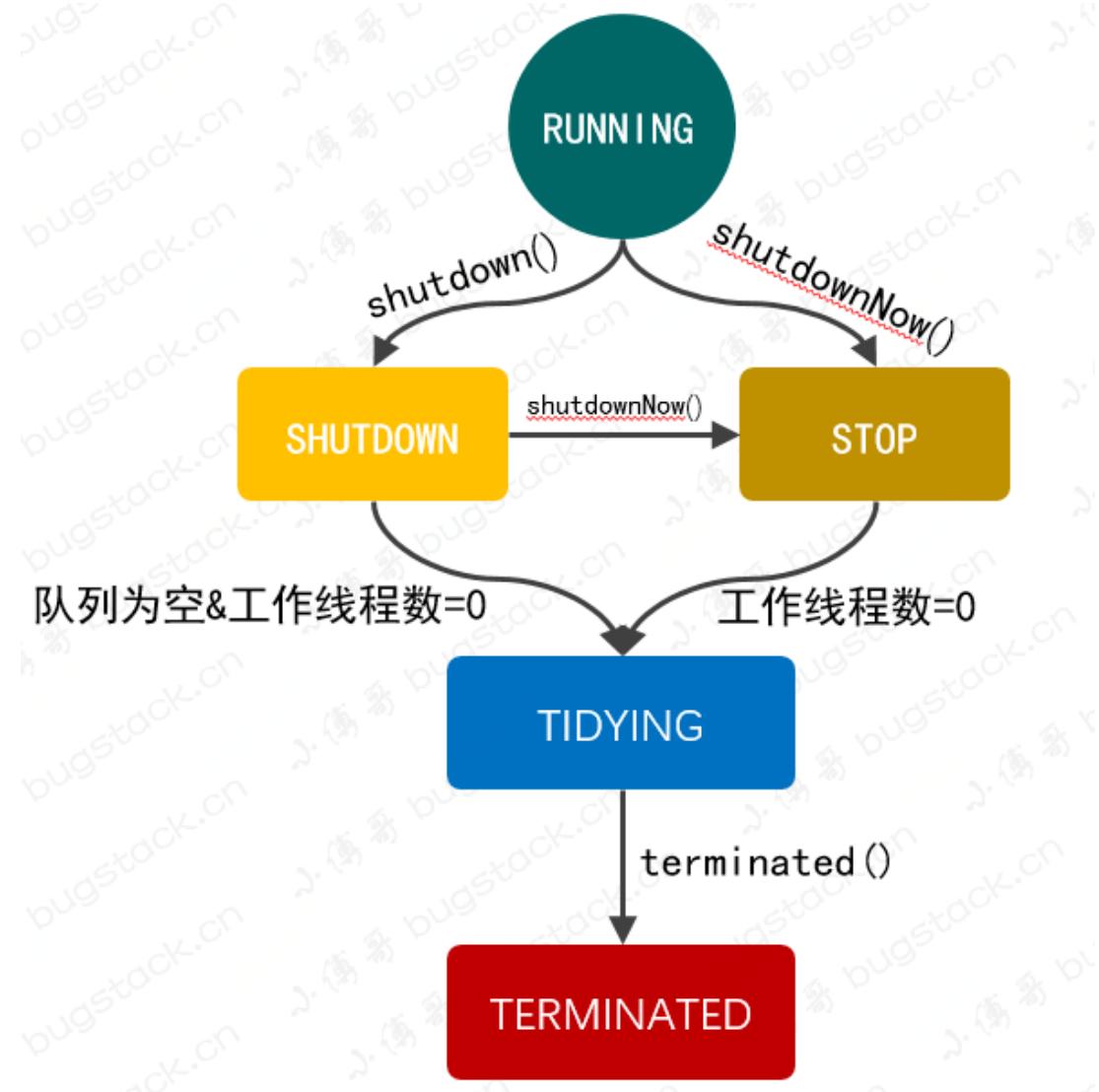


图 22-4 线程池状态流转

图 22-4 是线程池中的状态流转关系，包括如下状态：

- **RUNNING**: 运行状态，接受新的任务并且处理队列中的任务。
- **SHUTDOWN**: 关闭状态(调用了 `shutdown` 方法)。不接受新任务，但是要处理队列中的任务。
- **STOP**: 停止状态(调用了 `shutdownNow` 方法)。不接受新任务，也不处理队列中的任务，并且要中断正在处理的任务。

- **TIDYING**: 所有的任务都已终止了, workerCount 为 0, 线程池进入该状态后会调用 terminated() 方法进入 TERMINATED 状态。
- **TERMINATED**: 终止状态, terminated() 方法调用结束后的状态。

3.3 提交线程(execute)

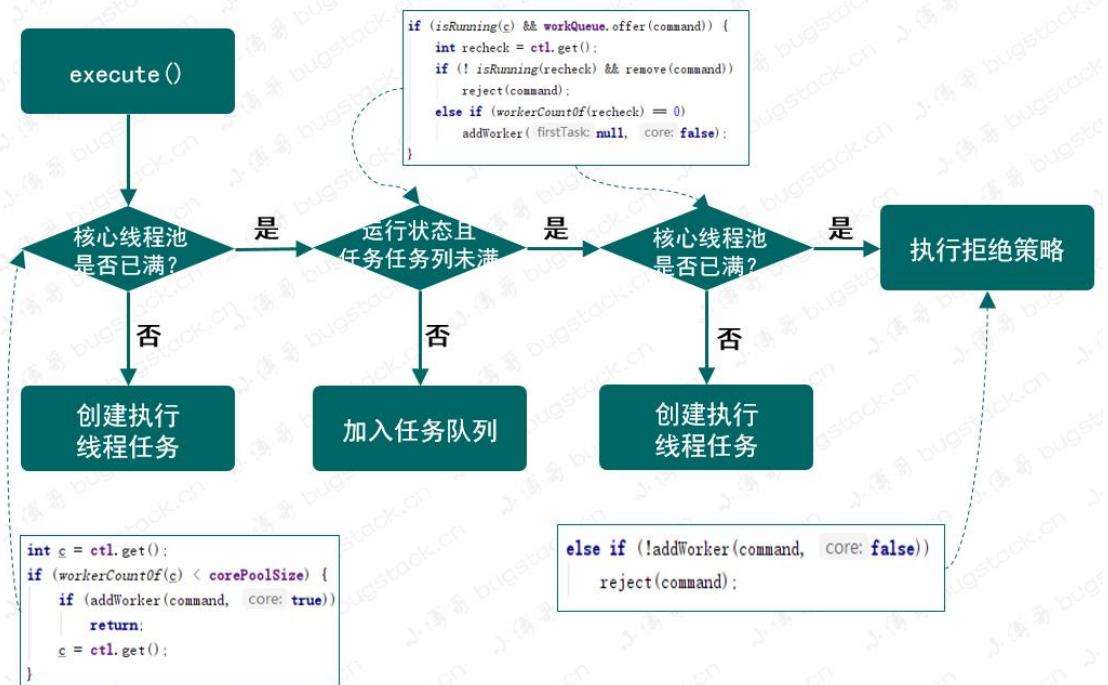


图 22-5 提交线程流程图

```

public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        if (!isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
}
  
```

```

    else if (!addWorker(command, false))
        reject(command);
}

```

在阅读这部分源码的时候，可以参考我们自己实现的线程池。其实最终的目的都是一样的，就是这段被提交的线程，[启动执行](#)、[加入队列](#)、[决策策略](#)，这三种方式。

- [ctl.get\(\)](#)，取的是记录线程状态和线程个数的值，最终需要使用方法 [workerCountOf\(\)](#)，来获取当前线程数量。[workerCountOf](#) 执行的是 $c \& CAPACITY$ 运算
- 根据当前线程池中线程数量，与核心线程数 [corePoolSize](#) 做对比，小于则进行添加线程到任务执行队列。
- 如果说此时线程数已满，那么则需要判断线程池是否为运行状态 [isRunning\(c\)](#)。如果是运行状态则把不能被执行的线程放入线程队列中。
- 放入线程队列以后，还需要重新判断线程是否运行以及移除操作，如果非运行且移除，则进行拒绝策略。否则判断线程数量为 0 后添加新线程。
- 最后就是再次尝试添加任务执行，此时方法 [addWorker](#) 的第二个入参是 `false`，最终会影响添加执行任务数量判断。如果添加失败则进行拒绝策略。

3.5 添加执行任务 (addWorker)

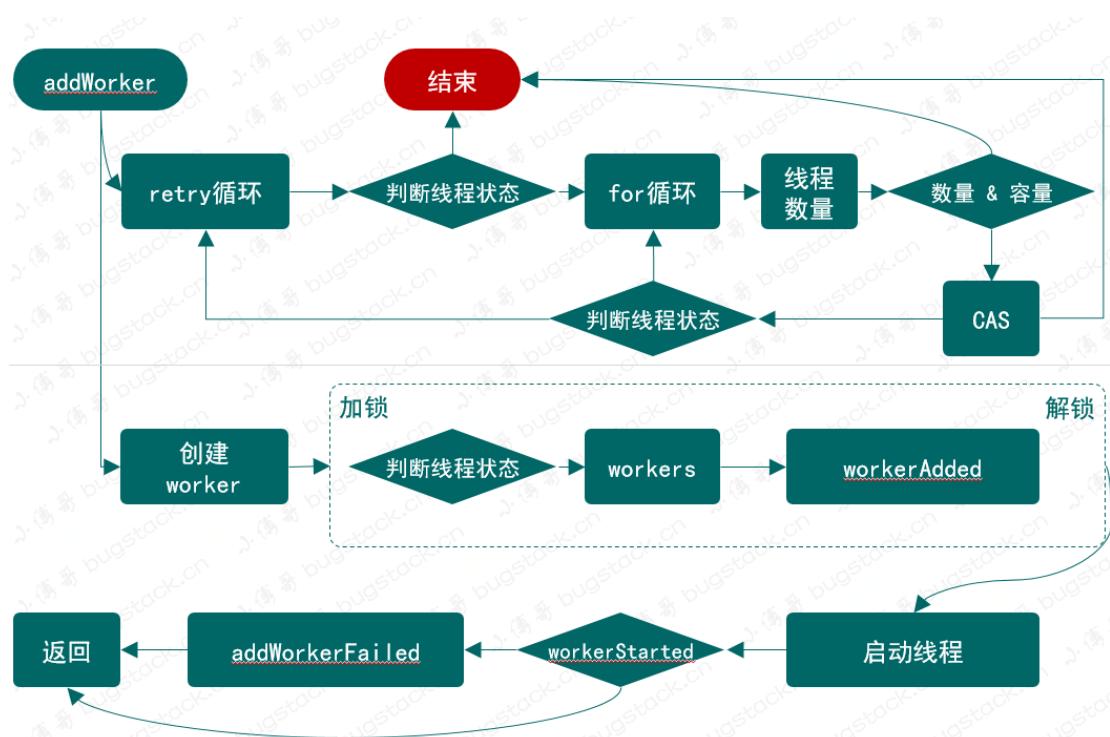


图 22-6 添加执行任务逻辑流程

`private boolean addWorker(Runnable firstTask, boolean core)`

第一部分、增加线程数量

```

retry:
for (;;) {
    int c = ctl.get();
    int rs = runStateOf(c);
    // Check if queue empty only if necessary.
    if (rs >= SHUTDOWN &&
        ! (rs == SHUTDOWN &&
            firstTask == null &&
            ! workQueue.isEmpty()))
        return false;
    for (;;) {
        int wc = workerCountOf(c);
        if (wc >= CAPACITY ||
            wc >= (core ? corePoolSize : maximumPoolSize))
            return false;
        if (compareAndIncrementWorkerCount(c))
            break retry;
        c = ctl.get(); // Re-read ctl
        if (runStateOf(c) != rs)
            continue retry;
        // else CAS failed due to workerCount change; retry inner Loop
    }
}

```

第一部分、创建启动线程

```

boolean workerStarted = false;
boolean workerAdded = false;
Worker w = null;
try {
    w = new Worker(firstTask);
    final Thread t = w.thread;
    if (t != null) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            int rs = runStateOf(ctl.get());
            if (rs < SHUTDOWN ||
                (rs == SHUTDOWN && firstTask == null)) {
                if (t.isAlive()) // precheck that t is startable

```

```

        throw new IllegalThreadStateException();

        workers.add(w);

        int s = workers.size();

        if (s > largestPoolSize)
            largestPoolSize = s;
        workerAdded = true;

    }

} finally {
    mainLock.unlock();
}

if (workerAdded) {
    t.start();
    workerStarted = true;
}

}

} finally {
    if (! workerStarted)
        addWorkerFailed(w);
}

return workerStarted;
}

```

添加执行任务的流程可以分为两块看，上面代码部分是用于记录线程数量、下面代码部分是在独占锁里创建执行线程并启动。这部分代码在不看锁、CAS 等操作，那么就和我们最开始手写的线程池基本一样了

- `if (rs >= SHUTDOWN &&! (rs == SHUTDOWN &&firstTask == null &&! workQueue.isEmpty()))`，判断当前线程池状态，是否为 `SHUTDOWN`、`STOP`、`TIDYING`、`TERMINATED` 中的一个。并且当前状态为 `SHUTDOWN`、且传入的任务为 `null`，同时队列不为空。那么就返回 `false`。
- `compareAndIncrementWorkerCount`，CAS 操作，增加线程数量，成功就会跳出标记的循环体。
- `runStateOf(c) != rs`，最后是线程池状态判断，决定是否循环。
- 在线程池数量记录成功后，则需要进入加锁环节，创建执行线程，并记录状态。在最后如果判断没有启动成功，则需要执行 `addWorkerFailed` 方法，剔除到线程方法等操作。

3.6 执行线程(runWorker)

```

final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // 允许中断
    boolean completedAbruptly = true;
    try {
        while (task != null || (task = getTask()) != null)
            w.lock();
        if ((runStateAtLeast(ctl.get(), STOP) ||
             (Thread.interrupted() &&
              runStateAtLeast(ctl.get(), STOP))) &&
            !wt.isInterrupted())
            wt.interrupt();
        try {
            beforeExecute(wt, task);
            Throwable thrown = null;
            try {
                task.run();
            } finally {
                afterExecute(task, thrown);
            }
        } finally {
            task = null;
            w.completedTasks++;
            w.unlock();
        }
    }
    completedAbruptly = false;
} finally {
    processWorkerExit(w, completedAbruptly);
}
}

```

其实，有了手写线程池的基础，到这也就基本了解了，线程池在干嘛。到这最核心的点就是 `task.run()` 让线程跑起来。额外再附带一些其他流程如下；

- `beforeExecute`、`afterExecute`，线程执行的前后做一些统计信息。
- 另外这里的锁操作是 Worker 继承 AQS 自己实现的不可重入的独占锁。

- `processWorkerExit`, 如果你感兴趣, 类似这样的方法也可以深入了解下。在线程退出时候 workers 做到一些移除处理以及完成任务数等, 也非常有意思

3.7 队列获取任务(getTask)

如果你已经开始阅读源码, 可以在 `runWorker` 方法中, 看到这样一句循环代码 `while (task != null || (task = getTask()) != null)`。这与我们手写线程池中操作的方式是一样的, 核心目的就是从队列中获取线程方法。

```
private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);
        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount();
            return null;
        }
        int wc = workerCountOf(c);
        // Are workers subject to culling?
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
        if ((wc > maximumPoolSize || (timed && timedOut)
            && (wc > 1 || workQueue.isEmpty())))
            if (compareAndDecrementWorkerCount(c))
                return null;
        continue;
    }
    try {
        Runnable r = timed ?
            workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
            workQueue.take();
        if (r != null)
            return r;
        timedOut = true;
    } catch (InterruptedException retry) {
        timedOut = false;
    }
}
```

```
    }  
}
```

- `getTask` 方法从阻塞队列中获取等待被执行的任务，也就是一条条往出拿线程方法。
- `if (rs >= SHUTDOWN ...)`，判断线程是否关闭。
- `wc = workerCountOf(c), wc > corePoolSize`，如果工作线程数超过核心线程数量 `corePoolSize` 并且 `workQueue` 不为空，则增加工作线程。但如果超时未获取到线程，则会把大于 `corePoolSize` 的线程销毁掉。
- `timed`，是 `allowCoreThreadTimeOut` 得来的。最终 `timed` 为 `true` 时，则通过阻塞队列的 `poll` 方法进行超时控制。
- 如果在 `keepAliveTime` 时间内没有获取到任务，则返回 `null`。如果为 `false`，则阻塞。

三、总结

- 这一章节并没有完全把线程池的所有知识点都介绍完，否则一篇内容会有些臃肿。在这一章节我们从手写线程池开始，逐步的分析这些代码在 Java 的线程池中是如何实现的，涉及到的知识点也几乎是我们以前介绍过的内容，包括：队列、CAS、AQS、重入锁、独占锁等内容。所以这些知识也基本是环环相扣的，最好有一些根基否则会有些不好理解。
- 除了本章介绍的，我们还没有讲到线程的销毁过程、四种线程池方法的选择和使用、以及在 **CPU 密集型任务**、**IO 密集型任务** 时该怎么配置。另外在 Spring 中也有自己实现的线程池方法。这些知识点都非常贴近实际操作。
- 好了，今天的内容先扯到这，后续的内容陆续完善。如果以上内容有错字、流程缺失、或者不好理解以及描述错误，欢迎留言。互相学习、互相进步。

第 4 节：线程池讲解以及 JVMTI 监控

五常大米好吃！

哈哈哈，是不你总买五常大米，其实五常和榆树是挨着的，榆树大米也好吃，榆树还是天下第一粮仓呢！但是五常出名，所以只认识五常。

为什么提这个呢，因为阿里不允许使用 Executors 创建线程池！其他很多大厂也不允许，这么创建的话，控制不好会出现 OOM。

好，本篇就带你学习四种线程池的不同使用方式、业务场景应用以及如何监控线程。

一、面试题

谢飞机，小记！，上次从面试官那逃跑后，恶补了多线程，自己好像也内卷了，所以出门逛逛！

面试官：嗨，飞机，飞机，这边！

谢飞机：嗯？！哎呀，面试官你咋来南海子公园了？

面试官：我家就附近，跑步来了。最近你咋样，上次问你的多线程学了吗？

谢飞机：哎，看了是看了，记不住鸭！

面试官：嗯，不常用确实记不住。不过你可以选择跳槽，来大厂，大厂的业务体量较大！

谢飞机：我就纠结呢，想回家考教师资格证了，我们村小学要教 java 了！

面试官：哈哈哈哈，一起！

二、四种线程池使用介绍

Executors 是创建线程池的工具类，比较典型的常见的四种线程池包括：

`newFixedThreadPool`、`newSingleThreadExecutor`、`newCachedThreadPool`、`newScheduledThreadPool`。每一种都有自己特定的典型例子，可以按照每种的特性用在不同的业务场景，也可以做为参照精细化创建线程池。

1. `newFixedThreadPool`

```
public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(3);
    for (int i = 1; i < 5; i++) {
        int groupId = i;
        executorService.execute(() -> {
            for (int j = 1; j < 5; j++) {
```

323 / 417

小傅哥，公众号：bugstack 虫洞栈 | 沉淀、分享、成长，让自己和他人都能有所收获

```
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ignored) {
        }
        logger.info("第 {} 组任务, 第 {} 次执行完成", groupId, j);
    }
});

executorService.shutdown();
}

// 测试结果
23:48:24.628 [pool-2-thread-1] INFO o.i.i.test.Test_newFixedThreadPool - 第 1 组任务, 第 1 次执行完成
23:48:24.628 [pool-2-thread-2] INFO o.i.i.test.Test_newFixedThreadPool - 第 2 组任务, 第 1 次执行完成
23:48:24.628 [pool-2-thread-3] INFO o.i.i.test.Test_newFixedThreadPool - 第 3 组任务, 第 1 次执行完成
23:48:25.633 [pool-2-thread-3] INFO o.i.i.test.Test_newFixedThreadPool - 第 3 组任务, 第 2 次执行完成
23:48:25.633 [pool-2-thread-1] INFO o.i.i.test.Test_newFixedThreadPool - 第 1 组任务, 第 2 次执行完成
23:48:25.633 [pool-2-thread-2] INFO o.i.i.test.Test_newFixedThreadPool - 第 2 组任务, 第 2 次执行完成
23:48:26.633 [pool-2-thread-3] INFO o.i.i.test.Test_newFixedThreadPool - 第 3 组任务, 第 3 次执行完成
23:48:26.633 [pool-2-thread-2] INFO o.i.i.test.Test_newFixedThreadPool - 第 2 组任务, 第 3 次执行完成
23:48:26.633 [pool-2-thread-1] INFO o.i.i.test.Test_newFixedThreadPool - 第 1 组任务, 第 3 次执行完成
23:48:27.634 [pool-2-thread-3] INFO o.i.i.test.Test_newFixedThreadPool - 第 3 组任务, 第 4 次执行完成
23:48:27.634 [pool-2-thread-2] INFO o.i.i.test.Test_newFixedThreadPool - 第 2 组任务, 第 4 次执行完成
23:48:27.634 [pool-2-thread-1] INFO o.i.i.test.Test_newFixedThreadPool - 第 1 组任务, 第 4 次执行完成
23:48:28.635 [pool-2-thread-3] INFO o.i.i.test.Test_newFixedThreadPool - 第 4 组任务, 第 1 次执行完成
23:48:29.635 [pool-2-thread-3] INFO o.i.i.test.Test_newFixedThreadPool - 第 4 组任务, 第 2 次执行完成
```

务, 第 2 次执行完成

23:48:30.635 [pool-2-thread-3] INFO o.i.i.test.Test_newFixedThreadPool - 第 4 组任务, 第 3 次执行完成

23:48:31.636 [pool-2-thread-3] INFO o.i.i.test.Test_newFixedThreadPool - 第 4 组任务, 第 4 次执行完成

Process finished with exit code 0

图解

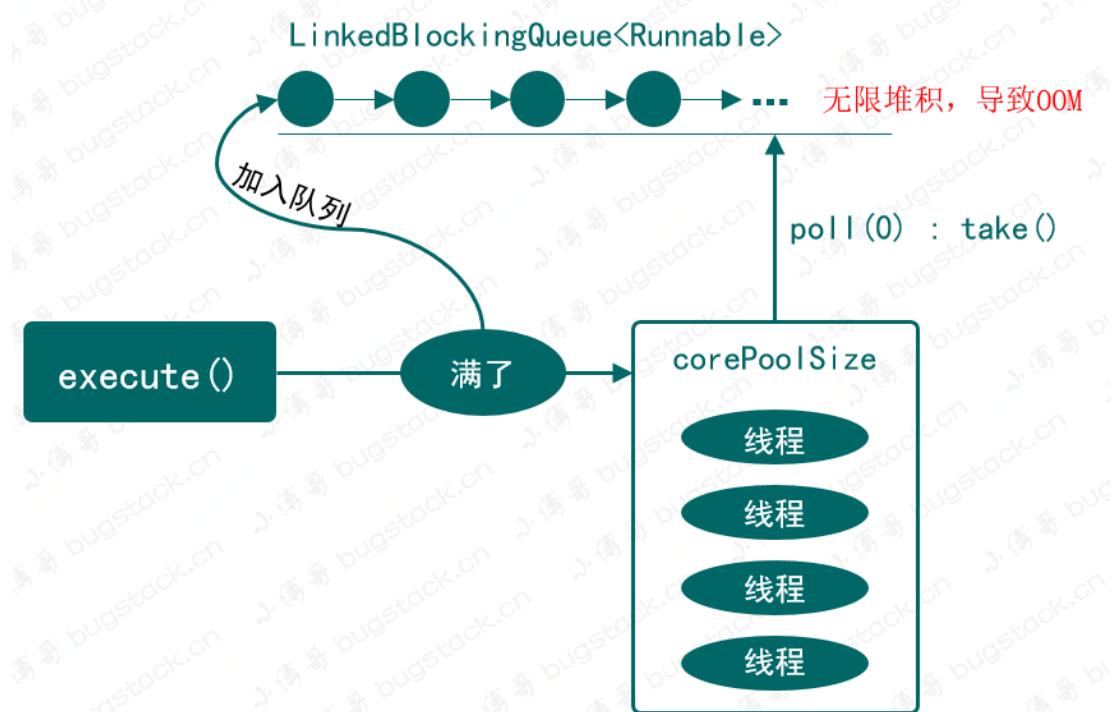


图 22-1 newFixedThreadPool 执行过程

- 代码: `new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>())`
- 介绍: 创建一个固定大小可重复使用的线程池, 以 `LinkedBlockingQueue` 无界阻塞队列存放等待线程。
- 风险: 随着线程任务不能被执行的的无限堆积, 可能会导致 OOM。

2. newSingleThreadExecutor

```
public static void main(String[] args) {
    ExecutorService executorService = Executors.newSingleThreadExecutor();
    for (int i = 1; i < 5; i++) {
        int groupId = i;
```

```
        executorService.execute(() -> {
            for (int j = 1; j < 5; j++) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException ignored) {
                }
                logger.info("第 {} 组任务, 第 {} 次执行完成", groupId, j);
            }
        });
    }

    executorService.shutdown();
}

// 测试结果
23:20:15.066 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 1 组
任务, 第 1 次执行完成
23:20:16.069 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 1 组
任务, 第 2 次执行完成
23:20:17.070 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 1 组
任务, 第 3 次执行完成
23:20:18.070 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 1 组
任务, 第 4 次执行完成
23:20:19.071 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 2 组
任务, 第 1 次执行完成
23:23:280.071 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 2 组
任务, 第 2 次执行完成
23:23:281.072 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 2 组
任务, 第 3 次执行完成
23:23:282.072 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 2 组
任务, 第 4 次执行完成
23:23:283.073 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 3 组
任务, 第 1 次执行完成
23:23:284.074 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 3 组
任务, 第 2 次执行完成
23:23:285.074 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 3 组
任务, 第 3 次执行完成
23:23:286.075 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 3 组
任务, 第 4 次执行完成
23:23:287.075 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 4 组
```

任务，第 1 次执行完成

23:23:288.075 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 4 组任务，第 2 次执行完成

23:23:289.076 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 4 组任务，第 3 次执行完成

23:20:30.076 [pool-2-thread-1] INFO o.i.i.t.Test_newSingleThreadExecutor - 第 4 组任务，第 4 次执行完成

图解

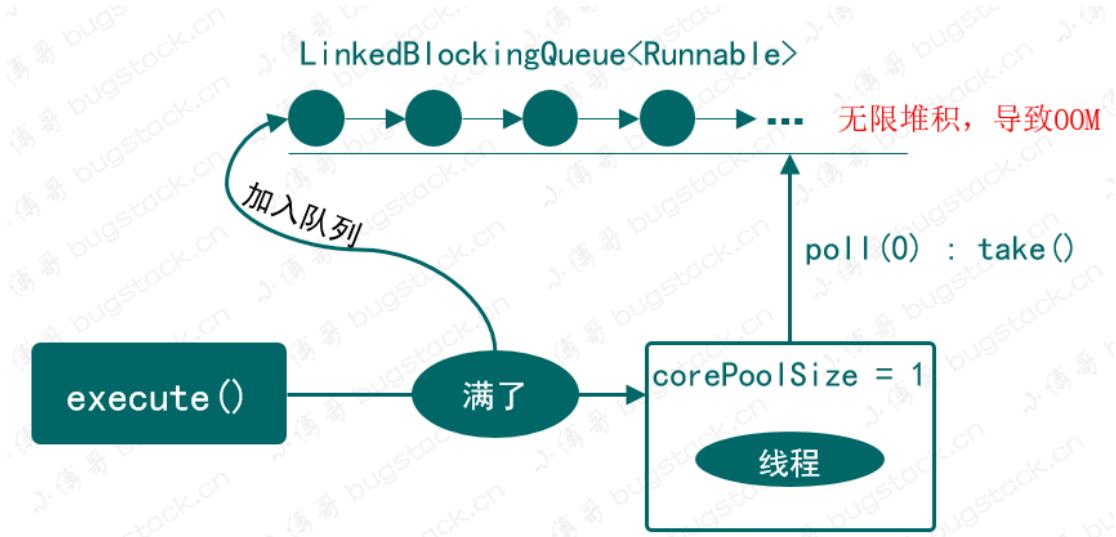


图 22-2 newSingleThreadExecutor 执行过程

- 代码: `new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>())`
- 介绍: 只创建一个执行线程任务的线程池, 如果出现意外终止则再创建一个。
- 风险: 同样这也是一个无界队列存放待执行线程, 无限堆积下会出现 OOM。

3. newCachedThreadPool

```
public static void main(String[] args) throws InterruptedException {
    ExecutorService executorService = Executors.newCachedThreadPool();
    for (int i = 1; i < 5; i++) {
        int groupId = i;
        executorService.execute(() -> {
            for (int j = 1; j < 5; j++) {
                try {
                    Thread.sleep(1000);
                }
```

```
        } catch (InterruptedException ignored) {
        }
        logger.info("第 {} 组任务, 第 {} 次执行完成", groupId, j);
    }
});  
}  
executorService.shutdown();  
  
// 测试结果  
23:25:59.818 [pool-2-thread-  
2] INFO o.i.i.test.Test_newCachedThreadPool - 第 2 组任务, 第 1 次执行完成  
23:25:59.818 [pool-2-thread-  
3] INFO o.i.i.test.Test_newCachedThreadPool - 第 3 组任务, 第 1 次执行完成  
23:25:59.818 [pool-2-thread-  
1] INFO o.i.i.test.Test_newCachedThreadPool - 第 1 组任务, 第 1 次执行完成  
23:25:59.818 [pool-2-thread-  
4] INFO o.i.i.test.Test_newCachedThreadPool - 第 4 组任务, 第 1 次执行完成  
23:25:00.823 [pool-2-thread-  
4] INFO o.i.i.test.Test_newCachedThreadPool - 第 4 组任务, 第 2 次执行完成  
23:25:00.823 [pool-2-thread-  
1] INFO o.i.i.test.Test_newCachedThreadPool - 第 1 组任务, 第 2 次执行完成  
23:25:00.823 [pool-2-thread-  
2] INFO o.i.i.test.Test_newCachedThreadPool - 第 2 组任务, 第 2 次执行完成  
23:25:00.823 [pool-2-thread-  
3] INFO o.i.i.test.Test_newCachedThreadPool - 第 3 组任务, 第 2 次执行完成  
23:25:01.823 [pool-2-thread-  
4] INFO o.i.i.test.Test_newCachedThreadPool - 第 4 组任务, 第 3 次执行完成  
23:25:01.823 [pool-2-thread-  
1] INFO o.i.i.test.Test_newCachedThreadPool - 第 1 组任务, 第 3 次执行完成  
23:25:01.824 [pool-2-thread-  
2] INFO o.i.i.test.Test_newCachedThreadPool - 第 2 组任务, 第 3 次执行完成  
23:25:01.824 [pool-2-thread-  
3] INFO o.i.i.test.Test_newCachedThreadPool - 第 3 组任务, 第 3 次执行完成  
23:25:02.824 [pool-2-thread-  
1] INFO o.i.i.test.Test_newCachedThreadPool - 第 1 组任务, 第 4 次执行完成  
23:25:02.824 [pool-2-thread-  
4] INFO o.i.i.test.Test_newCachedThreadPool - 第 4 组任务, 第 4 次执行完成  
23:25:02.825 [pool-2-thread-  
3] INFO o.i.i.test.Test_newCachedThreadPool - 第 3 组任务, 第 4 次执行完成
```

```
23:25:02.825 [pool-2-thread-  
2] INFO o.i.i.test.Test_newCachedThreadPool - 第 2 组任务, 第 4 次执行完成  
}
```

图解

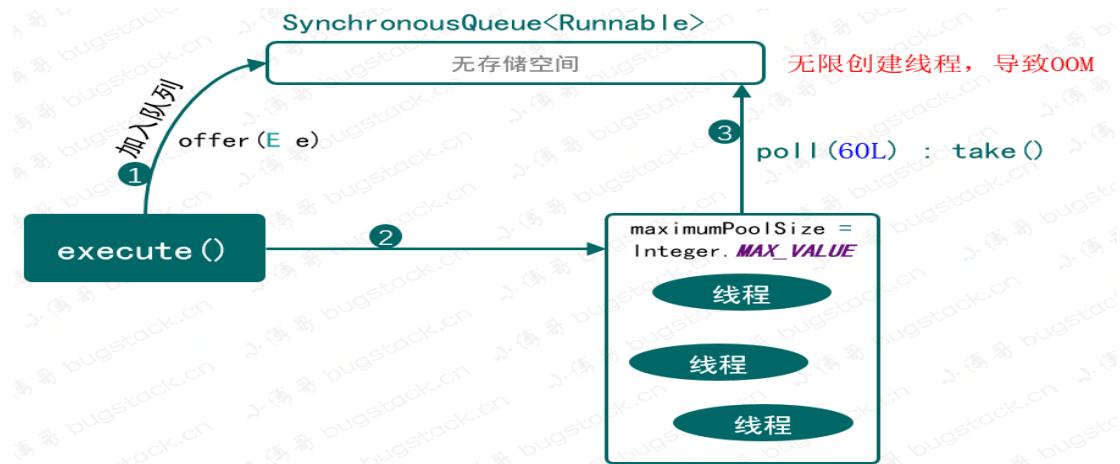


图 22-3 newCachedThreadPool 执行过程

- 代码: `new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS, new SynchronousQueue<Runnable>())`
- 介绍: 首先 `SynchronousQueue` 是一个生产消费模式的阻塞任务队列, 只要有任务就需要有线程执行, 线程池中的线程可以重复使用。
- 风险: 如果线程任务比较耗时, 又大量创建, 会导致 OOM

4. newScheduledThreadPool

```
public static void main(String[] args) {  
    ScheduledExecutorService executorService = Executors.newScheduledThreadPool(1);  
    executorService.schedule(() -> {  
        logger.info("3 秒后开始执行");  
    }, 3, TimeUnit.SECONDS);  
    executorService.scheduleAtFixedRate(() -> {  
        logger.info("3 秒后开始执行, 以后每 2 秒执行一次");  
    }, 3, 2, TimeUnit.SECONDS);  
    executorService.scheduleWithFixedDelay(() -> {  
        logger.info("3 秒后开始执行, 后续延迟 2 秒");  
    }, 3, 2, TimeUnit.SECONDS);  
}
```

```
// 测试结果
```

```
23:28:32.442 [pool-2-thread-1] INFO o.i.i.t.Test_newScheduledThreadPool - 3 秒后开始执行
23:28:32.444 [pool-2-thread-1] INFO o.i.i.t.Test_newScheduledThreadPool - 3 秒后开始执行, 以后每 2 秒执行一次
23:28:32.444 [pool-2-thread-1] INFO o.i.i.t.Test_newScheduledThreadPool - 3 秒后开始执行, 后续延迟 2 秒
23:28:34.441 [pool-2-thread-1] INFO o.i.i.t.Test_newScheduledThreadPool - 3 秒后开始执行, 以后每 2 秒执行一次
23:28:34.445 [pool-2-thread-1] INFO o.i.i.t.Test_newScheduledThreadPool - 3 秒后开始执行, 后续延迟 2 秒
23:28:36.440 [pool-2-thread-1] INFO o.i.i.t.Test_newScheduledThreadPool - 3 秒后开始执行, 以后每 2 秒执行一次
23:28:36.445 [pool-2-thread-1] INFO o.i.i.t.Test_newScheduledThreadPool - 3 秒后开始执行, 后续延迟 2 秒
```

图解

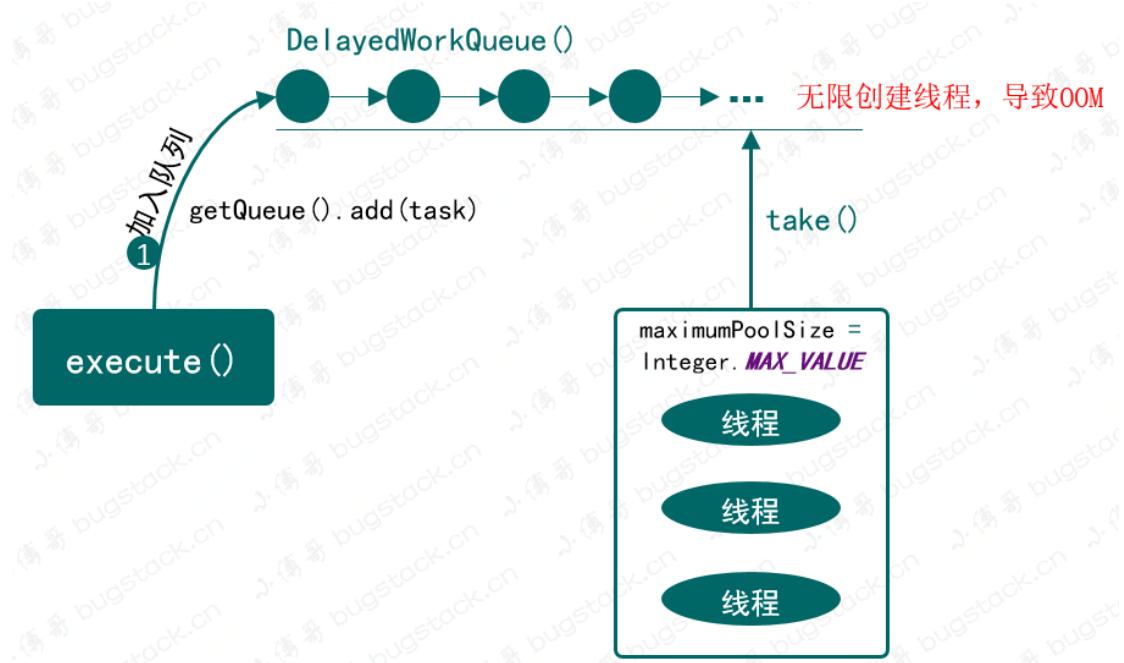


图 22-4 newScheduledThreadPool 执行过程

- 代码: `public ScheduledThreadPoolExecutor(int corePoolSize){ super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new ScheduledThreadPoolExecutor.DelayedWorkQueue()); }`
- 介绍: 这就是一个比较有意思的工作线程池了, 它可以延迟定时执行, 有点像我们的定时任务。同样它也是一个无限大小的线程池 `Integer.MAX_VALUE`。它提供的

调用方法比较多，包括：`scheduleAtFixedRate`、`scheduleWithFixedDelay`，可以按需选择延迟执行方式。

- **风险：**同样由于这是一组无限容量的线程池，所以依旧有 OOM 风险。

三、线程池使用场景说明

什么时候使用线程池？

说简单是当为了给老板省钱的时候，因为使用线程池可以降低服务器资源的投入，让每台机器尽可能更大限度的使用 CPU。

②那你这么说肯定没办法升职加薪了！

所以如果说的高大上一点，那么是在符合[科特尔法则](#)和[阿姆达尔定律](#)的情况下，引入线程池的使用最为合理。啥意思呢，还得简单说！

假如：我们有一套电商服务，用户浏览商品的并发访问速率是：1000 客户/每分钟，平均每个客户在服务器上的耗时 0.5 分钟。根据利特尔法则，在任何时刻，服务端都承担着 $1000 \times 0.5 = 500$ 个客户的业务处理量。过段时间大促了，并发访问的用户扩了一倍 2000 客户了，那怎么保障服务性能呢？

1. 提高服务器并发处理的业务量，即提高到 $2000 \times 0.5 = 1000$
2. 减少服务器平均处理客户请求的时间，即减少到： $2000 \times 0.25 = 500$

所以：在有些场景下会把串行的请求接口，压缩成并行执行，如图 22-5

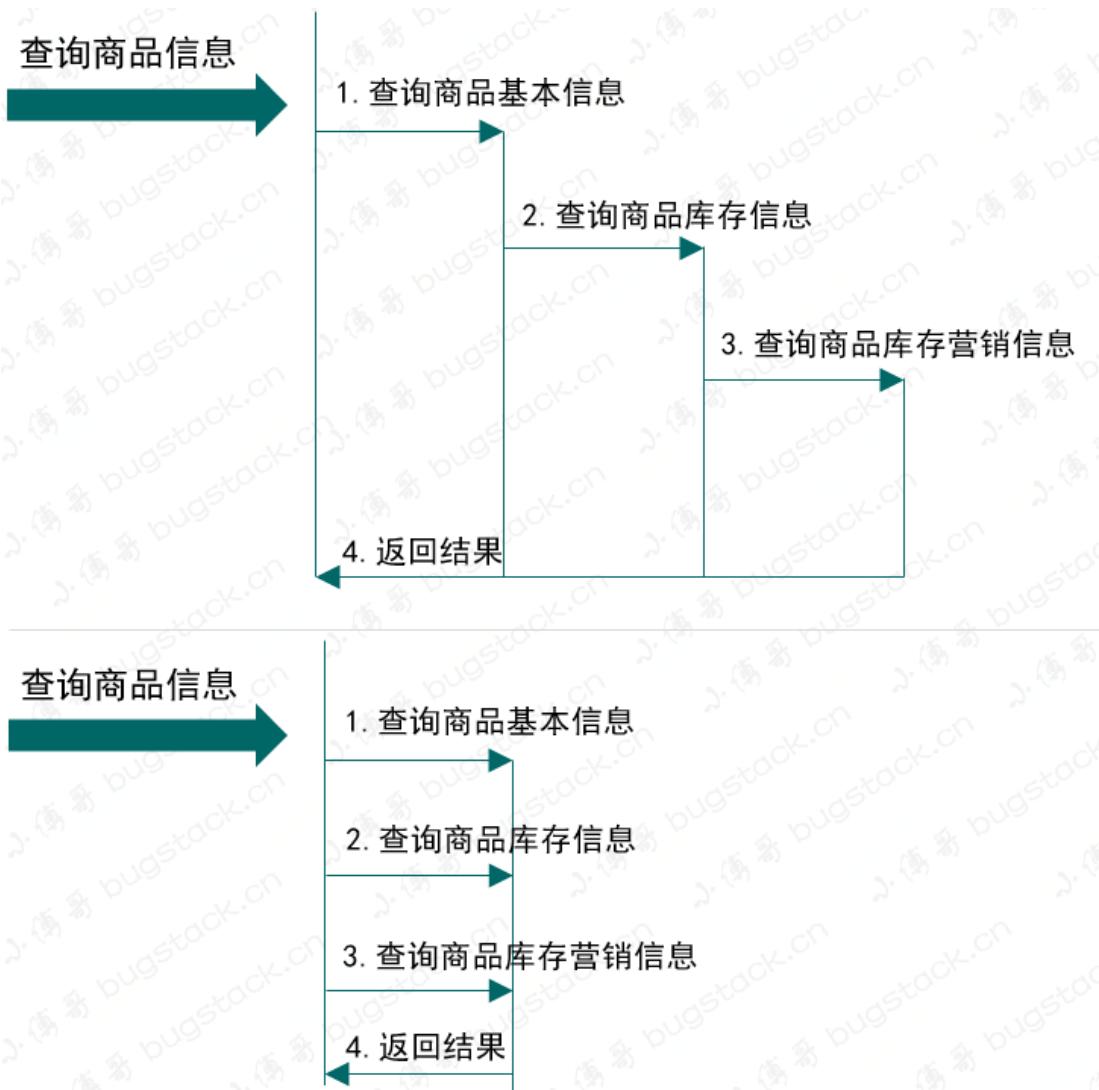


图 22-5 多线程接口查询使用

但是，线程池的使用会随着业务场景变化而不同，如果你的业务需要大量的使用线程池，并非常依赖线程池，那么就不可能用 `Executors` 工具类中提供的方法。因为这些线程池的创建都不够精细化，也非常容易造成 OOM 风险，而且随着业务场景逻辑不同，会有 I/O 密集型和 CPU 密集型。

最终，大家使用的线程池都是使用 `new ThreadPoolExecutor()` 创建的，当然也有基于 Spring 的线程池配置

`org.springframework.scheduling.concurrent.ThreadPoolExecutor`。

可你想过吗，同样一个接口在有活动时候怎么办、有大促时候怎么办，可能你当时设置的线程池是合理的，但是一到流量非常大的时候就很不适合了，所以如果能动态调整线程池就非常有必要了。而且使用 `new ThreadPoolExecutor()` 方式创建的线程池是可以通过提供的 `set` 方法进行动态调整的。有了这个动态调整

的方法后，就可以把线程池包装起来，在配合动态调整的页面，动态更新线程池参数，就可以非常方便的调整线程池了。

四、获取线程池监控信息

你收过报警短信吗？

收过，半夜还有报警机器人打电话呢！歲，你的系统有个机器睡着了，快起来看看！！！

所以，如果你高频、高依赖线程池，那么有一个完整的监控系统，就非重要了。总不能线上挂了，你还不知道！

可监控内容

方法	含义
getActiveCount()	线程池中正在执行任务的线程数量
getCompletedTaskCount()	线程池已完成的任务数量，该值小于等于 taskCount
getCorePoolSize()	线程池的核心线程数量
getLargestPoolSize()	线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过，也就是达到了 maximumPoolSize
getMaximumPoolSize()	线程池的最大线程数量
getPoolSize()	线程池当前的线程数量
getTaskCount()	线程池已经执行的和未执行的任务总数

1. 重写线程池方式监控

如果我们想监控一个线程池的方法执行动作，最简单的方式就是继承这个类，重写方法，在方法中添加动作收集信息。

伪代码

```
public class ThreadPoolMonitor extends ThreadPoolExecutor {  
  
    @Override  
    public void shutdown() {  
        // 监控逻辑  
    }  
}
```

```

    // 统计已执行任务、正在执行任务、未执行任务数量
    super.shutdown();
}

@Override
public List<Runnable> shutdownNow() {
    // 统计已执行任务、正在执行任务、未执行任务数量
    return super.shutdownNow();
}

@Override
protected void beforeExecute(Thread t, Runnable r) {
    // 记录开始时间
}

@Override
protected void afterExecute(Runnable r, Throwable t) {
    // 记录完成耗时
}

...
}

```

2. 基于 JVMTI 方式监控

这块是监控的重点，因为我们不太可能让每一个需要监控的线程池都来重写的方式记录，这样的改造成本太高了。

那么除了这个笨方法外，可以选择使用基于 JVMTI 的方式，进行开发监控组件。
JVMTI: JVMTI (JVM Tool Interface) 位于 jpda 最底层，是 Java 虚拟机所提供的 native 编程接口。JVMTI 可以提供性能分析、debug、内存管理、线程分析等功能。

基于 jvmti 提供的接口服务，运用 C++ 代码 (win32-add_library) 在 Agent_OnLoad 里开发监控服务，并生成 d11 文件。开发完成后在 java 代码中加入 agentpath，这样就可以监控到我们需要的信息内容。

环境准备：

1. Dev-C++

2. JetBrains CLion 2018.2.3
3. IntelliJ IDEA Community Edition 2018.3.1 x64
4. jdk1.8.0_45 64 位
5. jvmti (在 jdk 安装目录下 jdk1.8.0_45\include 里, 把 include 整个文件夹复制到和工程案例同层级目录下, 便于 include 引用)

配置信息: (路径相关修改为自己的)

1. C++开发工具 Clion 配置 1.配置位置;
Settings->Build,Execution,Deployment->Toolchains
2. MinGM 配置: D:\Program Files (x86)\Dev-Cpp\MinGW64
2. java 调试时配置
2. 配置位置: Run/Debug Configurations ->VM options
3. 配置内容: -agentpath:E:\itstack\git\github.com\itstack-jvmti\cmake-build-debug\libitstack_jvmti.dll

2.1 先做一个监控例子

Java 工程

```
public class TestLocationException {  
  
    public static void main(String[] args) {  
        Logger logger = Logger.getLogger("TestLocationException");  
        try {  
            PartnerEggResourceImpl resource = new PartnerEggResourceImpl();  
            Object obj = resource.queryUserInfoById(null);  
            logger.info("测试结果: " + obj);  
        } catch (Exception e) {  
            //屏蔽异常  
        }  
    }  
}  
  
class PartnerEggResourceImpl {  
    Logger logger = Logger.getLogger("PartnerEggResourceImpl");  
    public Object queryUserInfoById(String userId) {  
        logger.info("根据用户 Id 获取用户信息" + userId);  
        if (null == userId) {  
            throw new NullPointerException("根据用户 Id 获取用户信息, 空指针异常");  
        }  
    }  
}
```

```

        }

        return userId;
    }

}

c++监控

#include <iostream>
#include <cstring>
#include "jvmti.h"

using namespace std;

// 异常回调函数

static void JNICALL
callbackException(jvmtiEnv *jvmti_env, JNIEnv *env, jthread thr, jmethodID methodID
, jlocation location,
jobject exception, jmethodID catch_method, jlocation catch_location) {
    // 获得方法对应的类
    jclass clazz;
    jvmti_env->GetMethodDeclaringClass(methodId, &clazz);

    // 获得类的签名
    char *class_signature;
    jvmti_env->GetClassSignature(clazz, &class_signature, nullptr);

    // 过滤非本工程类信息
    string::size_type idx;
    string class_signature_str = class_signature;
    idx = class_signature_str.find("org/itstack");
    if (idx != 1) {
        return;
    }

    // 异常类名称
    char *exception_class_name;
    jclass exception_class = env->GetObjectClass(exception);
    jvmti_env->GetClassSignature(exception_class, &exception_class_name, nullptr);

    // 获得方法名称
}

```

```

char *method_name_ptr, *method_signature_ptr;
jvmti_env->GetMethodName(methodId, &method_name_ptr, &method_signature_ptr, nullptr
);

// 获取目标方法的起止地址和结束地址
jlocation start_location_ptr; //方法的起始位置
jlocation end_location_ptr; //用于方法的结束位置
jvmti_env->GetMethodLocation(methodId, &start_location_ptr, &end_location_ptr);

//输出测试结果
cout << "测试结果 - 定位类的签名: " << class_signature << endl;
cout << "测试结果 - 定位方法信息:
" << method_name_ptr << " -> " << method_signature_ptr << endl;
cout << "测试结果 - 定位方法位置:
" << start_location_ptr << " -> " << end_location_ptr + 1 << endl;
cout << "测试结果 - 异常类的名称: " << exception_class_name << endl;

cout << "测试结果-输出异常信息(可以分析行号): " << endl;
jclass throwable_class = (*env).FindClass("java/lang/Throwable");
jmethodID print_method = (*env).GetMethodID(throwable_class, "printStackTrace", "()
V");
(*env).CallVoidMethod(exception, print_method);

}

```

```

JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void *reserved) {
    jvmtiEnv *gb_jvmti = nullptr;
    //初始化
    vm->GetEnv(reinterpret_cast<void **>(&gb_jvmti), JVMTI_VERSION_1_0);
    // 创建一个新的环境
    jvmtiCapabilities caps;
    memset(&caps, 0, sizeof(caps));
    caps.can_signal_thread = 1;
    caps.can_get_owned_monitor_info = 1;
    caps.can_generate_method_entry_events = 1;
    caps.can_generate_exception_events = 1;
    caps.can_generate_vm_object_alloc_events = 1;
    caps.can_tag_objects = 1;

```

```

// 设置当前环境
gb_jvmti->AddCapabilities(&caps);

// 创建一个新的回调函数
jvmtiEventCallbacks callbacks;
memset(&callbacks, 0, sizeof(callbacks));

// 异常回调
callbacks.Exception = &callbackException;

// 设置回调函数
gb_jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));

// 开启事件监听(JVMTI_EVENT_EXCEPTION)
gb_jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_EXCEPTION, nullptr
);

return JNI_OK;
}

JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm) {
}

```

测试结果

在 VM options 中配置: `-agentpath:E:\itstack\git\github.com\itstack-jvmti\cmake-build-debug\libitstack_jvmti.dll`

```

十二月 16, 2020 23:53:27 下
午 org.itstack.demo.PartnerEggResourceImpl queryUserInfoById
信息: 根据用户 Id 获取用户信息 null
java.lang.NullPointerException: 根据用户 Id 获取用户信息, 空指针异常
at org.itstack.demo.PartnerEggResourceImpl.queryUserInfoById(TestLocationException
.java:26)
at org.itstack.demo.TestLocationException.main(TestLocationException.java:13)
测试结果-定位类的签名: Lorg/itstack/demo/PartnerEggResourceImpl;
测试结果-定位方法信息: queryUserInfoById -> (Ljava/lang/String;)Ljava/lang/Object;
测试结果-定位方法位置: 0 -> 43
测试结果-异常类的名称: Ljava/lang/NullPointerException;
测试结果-输出异常信息(可以分析行号):

```

- 这就是基于 JVMTI 的方式进行监控, 这样的方式可以做到非入侵代码。不需要硬编码, 也就节省了人力, 否则所有人都会进行开发监控内容, 而这部分内容与业务逻辑并无关系。

2.2 扩展线程监控

其实方法差不多，都是基于 C++ 开发 DLL 文件，引入使用。不过这部分代码会监控方法信息，并采集线程的执行内容。

```
static void JNICALL callbackMethodEntry(jvmtiEnv *jvmti_env, JNIEnv *env, jthread t
hr, jmethodID method) {
    // 获得方法对应的类
    jclass clazz;
    jvmti_env->GetMethodDeclaringClass(method, &clazz);

    // 获得类的签名
    char *class_signature;
    jvmti_env->GetClassSignature(clazz, &class_signature, nullptr);

    // 过滤非本工程类信息
    string::size_type idx;
    string class_signature_str = class_signature;
    idx = class_signature_str.find("org/itstack");

    gb_jvmti->RawMonitorEnter(gb_lock);

    {
        // must be deallocate
        char *name = NULL, *sig = NULL, *gsig = NULL;
        jint thr_hash_code = 0;

        error = gb_jvmti->GetMethodName(method, &name, &sig, &gsig);
        error = gb_jvmti->GetObjectHashCode(thr, &thr_hash_code);

        if (strcmp(name, "start") == 0 || strcmp(name, "interrupt") == 0 ||
            strcmp(name, "join") == 0 || strcmp(name, "stop") == 0 ||
            strcmp(name, "suspend") == 0 || strcmp(name, "resume") == 0) {

            // must be deallocate
            jobject thd_ptr = NULL;
            jint hash_code = 0;
            gb_jvmti->GetLocalObject(thr, 0, 0, &thd_ptr);
            gb_jvmti->GetObjectHashCode(thd_ptr, &hash_code);

            printf("[线程监
```

```

控]: thread (%10d) %10s (%10d)\n", thr_hash_code, name, hash_code);

}

}

gb_jvmti->RawMonitorExit(gb_lock);
}

JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm, char *options, void *reserved) {

// 初始化
jvm->GetEnv((void **) &gb_jvmti, JVMTI_VERSION_1_0);
// 创建一个新的环境
memset(&gb_capa, 0, sizeof(jvmtiCapabilities));
gb_capa.can_signal_thread = 1;
gb_capa.can_get_owned_monitor_info = 1;
gb_capa.can_generate_method_exit_events = 1;
gb_capa.can_generate_method_entry_events = 1;
gb_capa.can_generate_exception_events = 1;
gb_capa.can_generate_vm_object_alloc_events = 1;
gb_capa.can_tag_objects = 1;
gb_capa.can_generate_all_class_hook_events = 1;
gb_capa.can_generate_native_method_bind_events = 1;
gb_capa.can_access_local_variables = 1;
gb_capa.can_get_monitor_info = 1;
// 设置当前环境
gb_jvmti->AddCapabilities(&gb_capa);
// 创建一个新的回调函数
jvmtiEventCallbacks callbacks;
memset(&callbacks, 0, sizeof(jvmtiEventCallbacks));
// 方法回调
callbacks.MethodEntry = &callbackMethodEntry;
// 设置回调函数
gb_jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));

gb_jvmti->CreateRawMonitor("XFG", &gb_lock);

// 注册事件监听(JVMTI_EVENT_VM_INIT、JVMTI_EVENT_EXCEPTION、
JVMTI_EVENT_NATIVE_METHOD_BIND、JVMTI_EVENT_CLASS_FILE_LOAD_HOOK、
JVMTI_EVENT_METHOD_ENTRY、JVMTI_EVENT_METHOD_EXIT)

```

```

    error = gb_jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_VM_INIT,
(jthread) NULL);

    error = gb_jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_EXCEPTION,
(jthread) NULL);

    error = gb_jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_NATIVE_METHOD_BIND,
(jthread) NULL);

    error = gb_jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_CLASS_FILE_LOAD_HOOK,
(jthread) NULL);

    error = gb_jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_METHOD_ENTRY,
(jthread) NULL);

    error = gb_jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_METHOD_EXIT,
(jthread) NULL);

    return JNI_OK;
}

```

- 从监控的代码可以看到，这里有线程的 start、stop、join、interrupt 等，并可以记录执行信息。
- 另外这里监控的方法执行回调，`SetEventCallbacks(&callbacks, sizeof(callbacks));` 以及相应事件的添加。

五、总结

- 如果说你所经历的业务体量很小，那么几乎并不需要如此复杂的技术栈深度学习，甚至几乎不需要扩展各类功能，也不需要监控。但终究有一些需要造飞机的大厂，他们的业务体量庞大，并发数高，让原本可能就是一个简单的查询接口，也要做熔断、降级、限流、缓存、线程、异步、预热等等操作。
- 知其然才敢用，如果对一个技术点不太熟悉，就不要胡乱使用，否则遇到的 OOM 并不是那么好复现，尤其是在并发场景下。当然如果你们技术体系中有各种服务，比如流量复现、链路追踪等等，那么还好。
- 又扯到了这，一个坚持学习、分享、沉淀的男人！好了，如果有错字、内容不准确，欢迎直接怼给我，我喜欢接受。但不要欺负我哦哈哈哈哈！

第 5 章 JVM 虚拟机(5 节)

第 1 节：JDK、JRE、JVM

截至到这已经写了 22 篇面经手册，你看了多少？

⑤其实小傅哥就是借着面经的幌子在讲 Java 核心技术，探索这些核心知识点面试的背后到底在问什么。

想问一些面试官，是因为大家都在问所以你问，还是你想从这里问出什么？ 其实可能很多面试官如果不了解这些技术，往往会被求职者的答案击碎内心，哈哈哈哈哈哈。比如：[梅森旋转算法](#)、[开放寻址](#)、[斐波那契散列](#)、[启发式清理](#)、[Javassist 代理方式](#)、[扰动函数](#)、[哈希一致](#)等等。

记住，让懂了就是真的懂，比看水文、背答案要爽的多！嗯，就是有时候烧脑！

一、面试题

谢飞机，小记！，也不知道咋了，总感觉有些面试攻击性不大，但侮辱性极强！

面试官：谢飞机写过 Java 吗？

谢飞机：那当然写过，写了 3 年多了！

面试官：那，JDK、JRE、JVM 之间是什么关系？

谢飞机：嗯 J J J，JDK 里面有 JRE，JVM 好像在 JRE 里！？

面试官：那，Client 模式、Server 模式是啥？

谢飞机：嗯！？啥？

面试官：好吧，问个简单的。JVM 是如何工作的？背答案了吗？

谢飞机：再见，面试官！

二、Java 平台标准 (JDK 8)

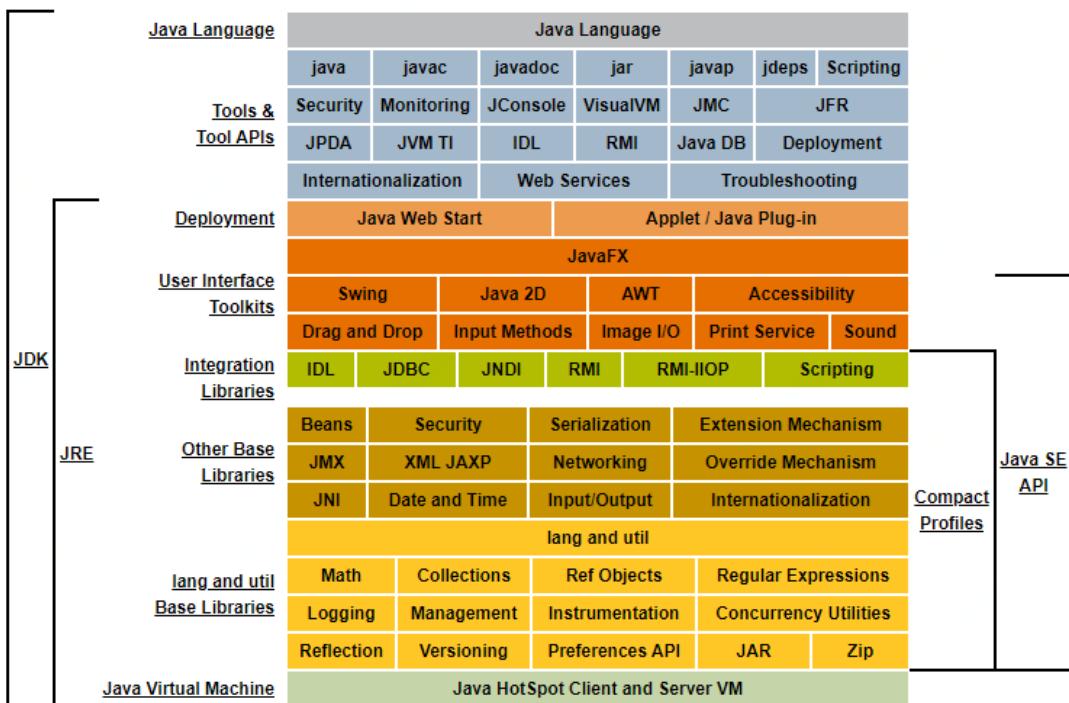
Oracle has two products that implement Java Platform Standard Edition (Java SE) 8: Java SE Development Kit (JDK) 8 and Java SE Runtime Environment (JRE) 8.

JDK 8 is a superset of JRE 8, and contains everything that is in JRE 8, plus tools such as the compilers and debuggers necessary for developing applets and applications. JRE 8 provides the libraries, the Java Virtual Machine (JVM), and other components to run applets and applications written in the Java programming language. Note that the

JRE includes components not required by the Java SE specification, including both standard and non-standard Java components.

The following conceptual diagram illustrates the components of Oracle's Java SE products:

Description of Java Conceptual Diagram



Java Platform Standard Edition 8 Documentation

关于 JDK、JRE、JVM 之间是什么关系，在 Java 平台标准中已经明确定义了。也就是上面的英文介绍部分。

- Oracle 有两个 Java 平台标准的产品，Java SE 开发工具包(JDK) 和 Java SE 运行时环境(JRE)。
- JDK(Java Development Kit Java 开发工具包)，JDK 是提供给 Java 开发人员使用的，其中包含了 java 的开发工具，也包括了 JRE。所以安装了 JDK，就不用在单独安装 JRE 了。其中的开发工具包括编译工具(javac.exe) 打包工具(jar.exe)等。
- JRE(Java Runtime Environment Java 运行环境) 是 JDK 的子集，也就是包括 JRE 所有内容，以及开发应用程序所需的编译器和调试器等工具。JRE 提供了库、Java 虚拟机 (JVM) 和其他组件，用于运行 Java 编程语言、小程序、应用程序。
- JVM(Java Virtual Machine Java 虚拟机)，JVM 可以理解为是一个虚拟出来的计算机，具备着计算机的基本运算方式，它主要负责把 Java 程序生成的字节码文件，解释成具体系统平台上的机器指令，让其在各个平台运行。

综上，从这段官网的平台标准介绍和概念图可以看出，我们运行程序的 JVM 是已经安装到 JDK 中，只不过可能你开发了很久的代码，也没有注意过。没有注意过的最大原因是，没有开发过一些和 JVM 相关的组件代码

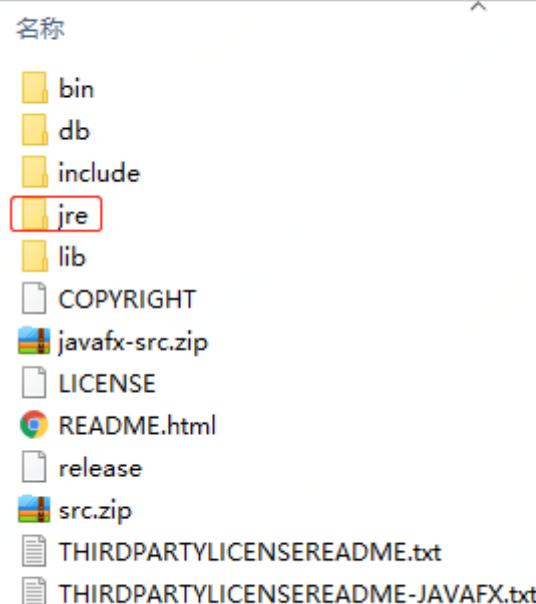
关于，各 JDK 版本的平台标准，可以自行比对学习，如下：

- Java SE 6 Documentation: <https://docs.oracle.com/javase/6/docs/>
- Java Platform Standard Edition 7 Documentation:
<https://docs.oracle.com/javase/7/docs/>
- Java Platform Standard Edition 8 Documentation:
<https://docs.oracle.com/javase/8/docs/>

三、JDK 目录结构和作用

我们默认安装完 JDK 会有 `jdk1.8.0_45`、`jre1.8.0_45`，两个文件夹。其实在 JDK 的文件中还会有 JRE 的文件夹，他们两个 JRE 文件夹的结构是一样的。

电脑 → Windows (C:) → Program Files → Java → jdk1.8.0_45 →



JDK 目录结构

- bin：一堆 EXE 可执行文件，`java.exe`、`javac.exe`、`javadoc.exe`，以及密钥管理工具等。
- db：内置了 Derby 数据库，体积小，免安装。
- include：Java 和 JVM 交互的头文件，例如我们 JVMTI 写的 C++ 工程时，就需要把这个 include 包引入进去 `jvmti.h`。[例如：基于 jvmti 设计非入侵监控](#)

- jre: Java 运行环境, 包含了运行时需要的可执行文件, 以及运行时需要依赖的 Java 类库和动态链接库。**.so .dll .dylib**
- lib: Java 类库, 例如 dt.jar、tools.jar

那么 jvm 在哪个文件夹呢?

Windows (C:) > Program Files > Java > jdk1.8.0_45 > jre > bin > server



jvm.dll

可能你之前并没有注意过 jvm 原来在这里 : C:\Program Files\Java\jdk1.8.0_45\jre\bin\server

- 这部分是整个 Java 实现跨平台的最核心内容, 由 Java 程序编译成的 .class 文件会在虚拟机上执行。
- 另外在 JVM 解释 class 文件时需要调用类库 lib。在 JRE 目录下有两个文件夹 lib、bin, 而 lib 就是 JVM 执行所需要的类库。
- jvm.dll 并不能独立工作, 当 jvm.dll 启动后, 会使用 explicit 方法来载入辅助动态链接库一起执行。

四、JDK 是什么?

综上通过 Java 平台标准和 JDK 的目录结构, JDK 是 JRE 的超集, JDK 包含了 JRE 所有的开发、调试以及监视应用程序的工具。以及如下重要的组件:

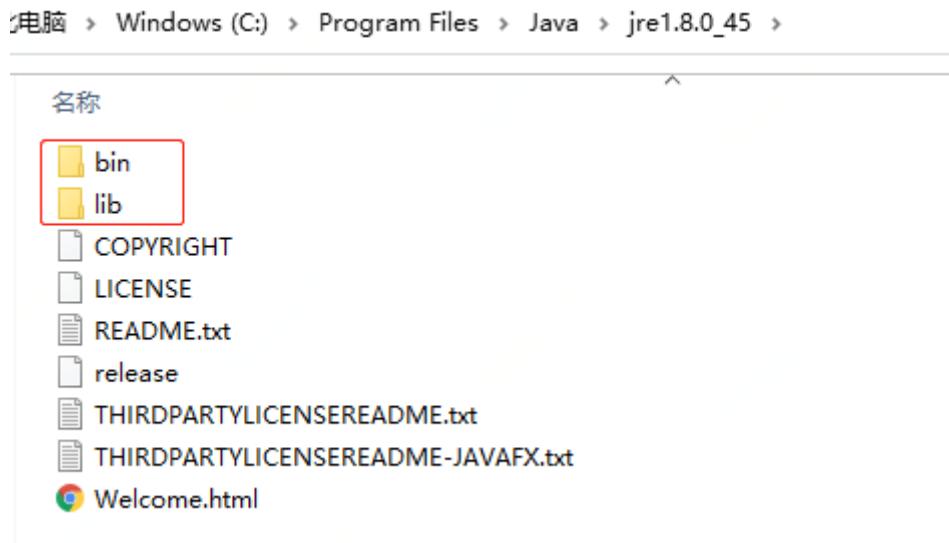
- java – 运行工具, 运行 .class 的字节码
- javac – 编译器, 将后缀名为.java 的源代码编译成后缀名为.class 的字节码
- javap – 反编译程序
- javadoc – 文档生成器, 从源码注释中提取文档, 注释需符合规范
- jar – 打包工具, 将相关的类文件打包成一个文件
- jdb – debugger, 调试工具
- jps – 显示当前 java 程序运行的进程状态
- appletviewer – 运行和调试 applet 程序的工具, 不需要使用浏览器
- javah – 从 Java 类生成 C 头文件和 C 源文件。这些文件提供了连接胶合, 使 Java 和 C 代码可进行交互。
- javaws – 运行 JNLP 程序

- extcheck – 一个检测 jar 包冲突的工具
- apt – 注释处理工具
- jhat – java 堆分析工具
- jstack – 栈跟踪程序
- jstat – JVM 检测统计工具
- jstard – jstat 守护进程
- jinfo – 获取正在运行或崩溃的 java 程序配置信息
- jmap – 获取 java 进程内存映射信息
- idlj – IDL-to-Java 编译器. 将 IDL 语言转化为 java 文件
- policytool – 一个 GUI 的策略文件创建和管理工具
- jrunscript – 命令行脚本运行
- appletviewer: 小程序浏览器，一种执行 HTML 文件上的 Java 小程序的 Java 浏览器

五、JRE 是什么？

JRE 本身也是一个运行在 CPU 上的程序，用于解释执行 Java 代码。

一般像是实施的工作，会在客户现场安装 JRE，因为这是运行 Java 程序的最低要求。



JRE 目录结构 lib、bin

- bin: 有 java.exe 但没有 javac.exe。也就是无法编译 Java 程序，但可以运行 Java 程序，可以把这个 bin 目录理解成 JVM。
- lib: Java 基础&核心类库，包含 JVM 运行时需要的类库和 rt.jar。也包含用于安全管理的文件，这些文件包括安全策略(security policy)和安全属性(security properties)等。

六、JVM 是什么？

其实简单说 JVM 就是运行 Java 字节码的虚拟机，JVM 是一种规范，各个供应商都可以实现自己 JVM 虚拟机。就像小傅哥自己也按照虚拟机规范和手写 JVM 的相关书籍实现了，基于 Java 实现的 JVM 虚拟机。



用 Java 实现 JVM 源码

源码地址: <https://github.com/fuzhengwei/itstack-demo-jvm>

内容简介: 本代码主要介绍如何通过 java 代码来实现 JVM 的基础功能（搜索解析 class 文件、字节码命令、运行时数据区等），从而让 java 程序员通过最熟知的 java 程序，学习 JVM 是如何将 java 程序一步步跑起来的。

当然，我们下载 Oracle 公司的 JVM 与自己实现的相比，要高级的多。他们的设计有不断优化的内存模型、GC 回收策略、自适应优化器等。

另外，JVM 之所以称为虚拟机，主要就是因为它为了实现 “write-once-run-anywhere” 提供了一个不依赖于底层操作系统和机器硬件结构的运行环境。

1. Client 模式、Server 模式

在 JVM 中有两种不同风格的启动模式，Client 模式、Server 模式。

- Client 模式：加载速度较快。可以用于运行 GUI 交互程序。
- Server 模式：加载速度较慢但运行起来较快。可以用于运行服务器后台程序。

修改配置模式文件: C:\Program Files\Java\jre1.8.0_45\lib\amd64\jvm.cfg

```
# List of JVMs that can be used as an option to java, javac, etc.  
# Order is important -- first in this list is the default JVM.
```

```
# NOTE that this both this file and its format are UNSUPPORTED and
# WILL GO AWAY in a future release.

#
# You may also select a JVM in an arbitrary location with the
# "-XX:altjvm=<jvm_dir>" option, but that too is unsupported
# and may not be available in a future release.

#
-server KNOWN
-client IGNORE
```

- 如果需要调整，可以把 client 设置为 KNOWN，并调整到 server 前面。
- JVM 默认在 Server 模式下，-Xms128M、-Xmx1024M
- JVM 默认在 Client 模式下，-Xms1M、-Xmx64M

2. JVM 结构和执行器

这部分属于 JVM 的核心知识，但不是本篇重点，会在后续的章节中陆续讲到。本章只做一些介绍。

- Class Loader：类装载器是用于加载类文件的一个子系统，其主要功能有三个：loading(加载)，linking（链接），initialization（初始化）。
- JVM Memory Areas：方法区、堆区、栈区、程序计数器。
- Interpreter(解释器)：通过查找预定义的 JVM 指令到机器指令映射，JVM 解释器可以将每个字节码指令转换为相应的本地指令。它直接执行字节码，不执行任何优化。
- JIT Compiler(即时编译器)：为了提高效率，JIT Compiler 在运行时与 JVM 交互，并适当将字节码序列编译为本地机器代码。典型地，JIT Compiler 执行一段代码，不是每次一条语句。优化这块代码，并将其翻译为优化的机器代码。*JIT Compiler 是默认开启*

七、总结

- 这篇的知识并不复杂，涉及的面试内容也较少，更多的是对接下来要讲到 JVM 相关面试内容的一个开篇介绍，为后续的要讲的内容做一个铺垫。
- 如果你在此之前没有关注过 JDK、JRE、JVM 的结构和相应的组件配置以及执行模式，那么可以在此基础上继续学习加深印象。另外想深入学习 JVM 并不太容易，既要学习 JVM 规范也要上手应用实践，所以很建议先手写 JVM，再实践验证 JVM。

- 好了，本章节就扯到这了。这些知识点即使分享给大家，也是我自己学习、收录、整理、验证的过程。互相学习、互相成长，如果有错误之处，直接留言给我，我会不断的改正。大家一起进步！

第 2 节：JVM 类加载实践

学习，不知道从哪下手？

当学习一个新知识不知道从哪下手的时候，最有效的办法是梳理这个知识结构的脉络信息，汇总出一整张的思维导图。接下来就是按照思维导图的知识结构，一个个学习相应的知识点，并汇总记录。

就像 JVM 的学习，可以说它包括了非常多的内容，也是一个庞大的知识体系。例如：**类加载、加载器、生命周期、性能优化、调优参数、调优工具、优化方案、内存区域、虚拟机栈、直接内存、内存溢出、元空间、垃圾回收、可达性分析、标记清除、回收过程**等等。如果没有梳理的一头扎进去，东一榔头西一棒子，很容易造成学习恐惧感。

如图 24-1 是 JVM 知识框架梳理，后续我们会按照这个结构陆续讲解每一块内容。

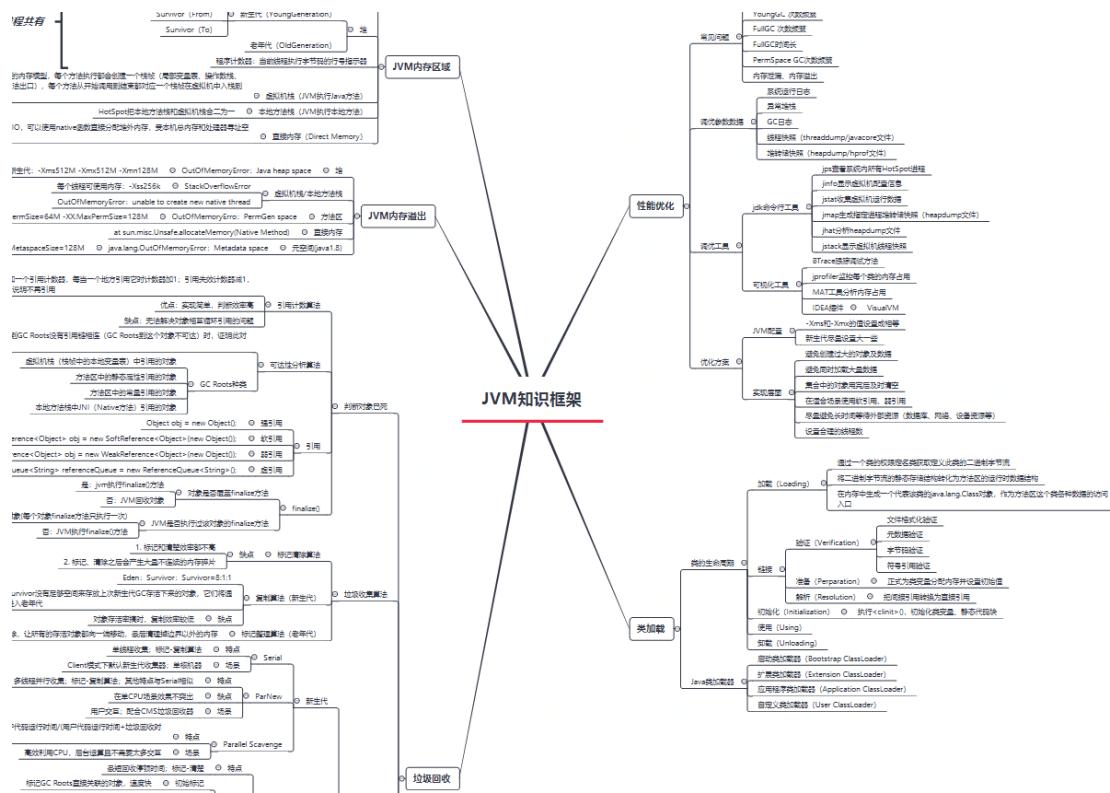


图 24-1 JVM 知识框架

一、面试题

谢飞机，小记！，很多知识根本就是背背背，也没法操作，难学！

谢飞机：大哥，你问我两个 JVM 问题，我看看我自己还行不！

面试官：啊？嗯！往死了问还是？

谢飞机：就就就，都行！你看着来！

面试官：啊，那 JVM 加载过程都是什么步骤？

谢飞机：巴拉巴拉，加载、验证、准备、解析、初始化、使用、卸载！

面试官：嗯，背的挺好！我怀疑你没操作过！那加载的时候，JVM 规范规定从第几位开始是解析常量池，以及数据类型是如何定义的，u1、u2、u4，是怎么个玩意？

谢飞机：握草！算了，告诉我看啥吧！

二、类加载过程描述

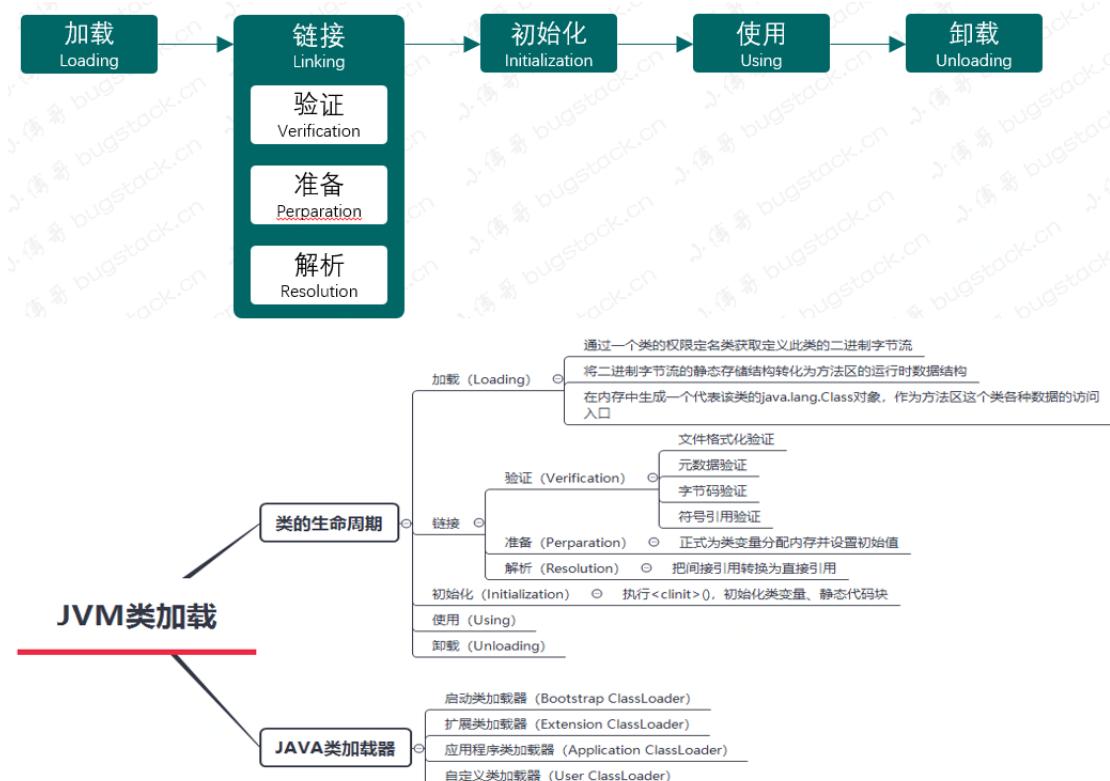


图 24-2 JVM 类加载过程

JVM 类加载过程分为，加载、链接、初始化、使用和卸载这四个阶段，在链接中又包括：验证、准备、解析。

- **加载**：Java 虚拟机规范对 class 文件格式进行了严格的规则，但对于从哪里加载 class 文件，却非常自由。Java 虚拟机实现可以从文件系统读取、从 JAR(或 ZIP)压缩包中提取 class 文件。除此之外也可以通过网络下载、数据库加载，甚至是运行时直接生成的 class 文件。
- **链接**：包括了三个阶段：
 - 验证，确保被加载类的正确性，验证字节流是否符合 class 文件规范，例魔数 0xCAFEBAE，以及版本号等。

- 准备，为类的静态变量分配内存并设置变量初始值等
- 解析，解析包括解析出常量池数据和属性表信息，这里会包括 ConstantPool 结构体以及 AttributeInfo 接口等。
- **初始化**：类加载完成的最后一歩就是初始化，目的就是为标记常量值的字段赋值，以及执行 `<clinit>` 方法的过程。JVM 虚拟机通过锁的方式确保 `clinit` 仅被执行一次
- **使用**：程序代码执行使用阶段。
- **卸载**：程序代码退出、异常、结束等。

三、写个代码加载下

JVM 之所以不好掌握，主要是因为不好实操。虚拟机是 C++ 写的，很多 Java 程序员根本就不会去读，或者读不懂。那么，也就没办法实实在在的体会到，到底是怎么加载的，加载的时候都干了啥。只有看到代码，我才觉得自己学会了！

所以，我们这里要手动写一下，JVM 虚拟机的部分代码，也就是类加载的过程。通过 Java 代码来实现 Java 虚拟机的部分功能，让开发 Java 代码的程序员更容易理解虚拟机的执行过程。

1. 案例工程

```
interview-24
├── pom.xml
└── src
    └── main
        └── java
            └── org.itstack.interview.jvm
                ├── classpath
                │   └── impl
                │       ├── CompositeEntry.java
                │       ├── DirEntry.java
                │       ├── WildcardEntry.java
                │       └── ZipEntry.java
                └── Classpath.java
            └── Entry.java
        └── Cmd.java
    └── Main.java
└── test
    └── java
```

```
└── org.itstack.interview.jvm.test  
    └── HelloWorld.java
```

以上，工程结构就是按照 JVM 虚拟机规范，使用 Java 代码实现 JVM 中加载 class 文件部分内容。当然这部分还不包括解析，因为解析部分的代码非常庞大，我们先从把 .class 文件加载读取开始了解。

2. 代码讲解

2.1 定义类路径接口 (Entry)

```
public interface Entry {  
  
    byte[] readClass(String className) throws IOException;  
  
    static Entry create(String path) {  
        //File.separator; 路径分隔符(win\Linux)  
        if (path.contains(File.separator)) {  
            return new CompositeEntry(path);  
        }  
        if (path.endsWith("*")) {  
            return new WildcardEntry(path);  
        }  
        if (path.endsWith(".jar") || path.endsWith(".JAR") ||  
            path.endsWith(".zip") || path.endsWith(".ZIP")) {  
            return new ZipEntry(path);  
        }  
        return new DirEntry(path);  
    }  
}
```

- 接口中提供了接口方法 `readClass` 和静态方法 `create(String path)`。
- jdk1.8 是可以在接口中编写静态方法的，在设计上属于补全了抽象类的类似功能。这个静态方法主要是按照不同的路径地址类型，提供不同的解析方法。包括：CompositeEntry、WildcardEntry、ZipEntry、DirEntry，这四种。接下来分别看每一种的具体实现

2.2 目录形式路径 (DirEntry)

```

public class DirEntry implements Entry {

    private Path absolutePath;

    public DirEntry(String path){
        //获取绝对路径
        this.absolutePath = Paths.get(path).toAbsolutePath();
    }

    @Override
    public byte[] readClass(String className) throws IOException {
        return Files.readAllBytes(absolutePath.resolve(className));
    }

    @Override
    public String toString() {
        return this.absolutePath.toString();
    }
}

```

- 目录形式的通过读取绝对路径下的文件，通过 `Files.readAllBytes` 方式获取字节码。

2.3 压缩包形式路径(ZipEntry)

```

public class ZipEntry implements Entry {

    private Path absolutePath;

    public ZipEntry(String path) {
        //获取绝对路径
        this.absolutePath = Paths.get(path).toAbsolutePath();
    }

    @Override
    public byte[] readClass(String className) throws IOException {
        try (FileSystem zipFs = FileSystems.newFileSystem(absolutePath, null)) {
            return Files.readAllBytes(zipFs.getPath(className));
        }
    }
}

```

```
}

@Override
public String toString() {
    return this.getAbsolutePath().toString();
}

}
```

- 其实压缩包形式与目录形式，只有在文件读取上有包装差别而已。

```
FileSystems.newFileSystem
```

2.4 混合形式路径(CompositeEntry)

```
public class CompositeEntry implements Entry {

    private final List<Entry> entryList = new ArrayList<>();

    public CompositeEntry(String pathList) {
        String[] paths = pathList.split(File.separator);
        for (String path : paths) {
            entryList.add(Entry.create(path));
        }
    }

    @Override
    public byte[] readClass(String className) throws IOException {
        for (Entry entry : entryList) {
            try {
                return entry.readClass(className);
            } catch (Exception ignored) {
                //ignored
            }
        }
        throw new IOException("class not found " + className);
    }

    @Override
```

```

public String toString() {
    String[] strs = new String[entryList.size()];
    for (int i = 0; i < entryList.size(); i++) {
        strs[i] = entryList.get(i).toString();
    }
    return String.join(File.separator, strs);
}

}

```

- `File.separator`, 是一个分隔符属性, win/linux 有不同的类型, 所以使用这个方法进行分割路径。
- 分割后的路径装到 List 集合中, 这个过程属于拆分路径。

2.5 通配符类型路径 (WildcardEntry)

```

public class WildcardEntry extends CompositeEntry {

    public WildcardEntry(String path) {
        super(toPathList(path));
    }

    private static String toPathList(String wildcardPath) {
        String baseDir = wildcardPath.replace("*", ""); // remove *
        try {
            return Files.walk(Paths.get(baseDir))
                .filter(Files::isRegularFile)
                .map(Path::toString)
                .filter(p -> p.endsWith(".jar") || p.endsWith(".JAR"))
                .collect(Collectors.joining(File.separator));
        } catch (IOException e) {
            return "";
        }
    }
}

```

- 这个类属于混合形式路径处理类的子类, 唯一提供的方法就是把类路径解析出来。

2.6 类路径解析(Classpath)

启动类路径、扩展类路径、用户类路径，熟悉吗？是不经常看到这几句话，那么时候怎么实现的呢？

有了上面我们做的一些基础类的工作，接下来就是类解析的实际调用过程。代码如下：

```
public class Classpath {

    private Entry bootstrapClasspath; //启动类路径
    private Entry extensionClasspath; //扩展类路径
    private Entry userClasspath; //用户类路径

    public Classpath(String jreOption, String cpOption) {
        //启动类&扩展类 "C:\Program Files\Java\jdk1.8.0_161\jre"
        bootstrapAndExtensionClasspath(jreOption);
        //用户类 F:\..\org\itstack\demo\test\HelloWorld
        parseUserClasspath(cpOption);
    }

    private void bootstrapAndExtensionClasspath(String jreOption) {

        String jreDir = getJreDir(jreOption);

        //..jre/*lib/*
        String jreLibPath = Paths.get(jreDir, "lib") + File.separator + "*";
        bootstrapClasspath = new WildcardEntry(jreLibPath);

        //..jre/*lib/ext/*
        String jreExtPath = Paths.get(jreDir, "lib", "ext") + File.separator + "*";
        extensionClasspath = new WildcardEntry(jreExtPath);

    }

    private static String getJreDir(String jreOption) {
        if (jreOption != null && Files.exists(Paths.get(jreOption))) {
            return jreOption;
        }
        if (Files.exists(Paths.get("./jre"))) {

```

```

        return "./jre";
    }

    String jh = System.getenv("JAVA_HOME");
    if (jh != null) {
        return Paths.get(jh, "jre").toString();
    }
    throw new RuntimeException("Can not find JRE folder!");
}

private void parseUserClasspath(String cpOption) {
    if (cpOption == null) {
        cpOption = ".";
    }
    userClasspath = Entry.create(cpOption);
}

public byte[] readClass(String className) throws Exception {
    className = className + ".class";

    // [readClass] 启动类路径
    try {
        return bootstrapClasspath.readClass(className);
    } catch (Exception ignored) {
        // ignored
    }

    // [readClass] 扩展类路径
    try {
        return extensionClasspath.readClass(className);
    } catch (Exception ignored) {
        // ignored
    }

    // [readClass] 用户类路径
    return userClasspath.readClass(className);
}
}

```

- 启动类路径, bootstrapClasspath.readClass(className);
- 扩展类路径, extensionClasspath.readClass(className);
- 用户类路径, userClasspath.readClass(className);
- 这回就看到它们具体在哪使用了吧! 有了具体的代码也就方便理解了

2.7 加载类测试验证

```
private static void startJVM(Cmd cmd) {
    Classpath cp = new Classpath(cmd.jre, cmd.classpath);
    System.out.printf("classpath: %s class: %s args: %s\n", cp, cmd.getMainClass(),
        cmd.getAppArgs());
    //获取 className
    String className = cmd.getMainClass().replace(".", "/");
    try {
        byte[] classData = cp.readClass(className);
        System.out.println(Arrays.toString(classData));
    } catch (Exception e) {
        System.out.println("Could not find or load main class " + cmd.getMainClass());
    });
    e.printStackTrace();
}
}
```

这段就是使用 Classpath 类进行类路径加载, 这里我们测试加载 java.lang.String 类。你可以加载其他的类, 或者自己写的类

- 配置 IDEA, program arguments 参数: -Xjre "C:\Program Files\Java\jdk1.8.0_161\jre" java.lang.String
- 另外这里读取出的 class 文件信息, 打印的是 byte 类型信息。

测试结果

```
[-54, -2, -70, -66, 0, 0, 0, 52, 2, 28, 3, 0, 0, -40, 0, 3, 0, 0, -37, -
1, 3, 0, 0, -33, -
1, 3, 0, 1, 0, 0, 8, 0, 15, 8, 0, 61, 8, 0, 85, 8, 0, 88, 8, 0, 89, 8, 0, 112, 8, 0
, -81, 8, 0, -75, 8, 0, -47, 8, 0, -
45, 1, 0, 0, 1, 0, 3, 40, 41, 73, 1, 0, 20, 40, 41, 76, 106, 97, 118, 97, 47, 108,
97, 110, 103, 47, 79, 98, 106, 101, 99, 116, 59, 1, 0, 20, 40, 41, 76, 106, 97, 118
```

```
, 97, 47, 108, 97, 110, 103, 47, 83, 116, 114, 105, 110, 103, 59, 1, 0, 3, 40, 41, 86, 1, 0, 3, 40, 41, 90, 1, 0, 4, 40, 41, 91, ...]
```

这块部分截取的程序运行打印结果，就是读取的 class 文件信息，只不过暂时还不能看出什么。接下来我们再把它翻译过来！

四、解析字节码文件

JVM 在把 class 文件加载完成后，接下来就进入[链接](#)的过程，这个过程包括了内容的校验、准备和解析，其实就是把 byte 类型 class 翻译过来，做相应的操作。

整个这个过程内容相对较多，这里只做部分逻辑的实现和讲解。如果读者感兴趣可以阅读小傅哥的[《用 Java 实现 JVM》](#)专栏。

1. 提取部分字节码

```
//取部分字节码: java.lang.String
private static byte[] classData = {
    -54, -2, -70, -66, 0, 0, 0, 52, 2, 26, 3, 0, 0, -40, 0, 3, 0, 0, -37, -
1, 3, 0, 0, -33, -1, 3, 0, 1, 0, 0, 8, 0,
    59, 8, 0, 83, 8, 0, 86, 8, 0, 87, 8, 0, 110, 8, 0, -83, 8, 0, -77, 8, 0, -
49, 8, 0, -47, 1, 0, 3, 40, 41, 73, 1,
    0, 20, 40, 41, 76, 106, 97, 118, 97, 47, 108, 97, 110, 103, 47, 79, 98, 106
, 101, 99, 116, 59, 1, 0, 20, 40, 41,
    76, 106, 97, 118, 97, 47, 108, 97, 110, 103, 47, 83, 116, 114, 105, 110, 10
3, 59, 1, 0, 3, 40, 41, 86, 1, 0, 3,
    40, 41, 90, 1, 0, 4, 40, 41, 91, 66, 1, 0, 4, 40, 41, 91, 67, 1, 0, 4, 40,
67, 41, 67, 1, 0, 21, 40, 68, 41, 76,
    106, 97, 118, 97, 47, 108, 97, 110, 103, 47, 83, 116, 114, 105, 110, 103, 5
9, 1, 0, 4, 40, 73, 41, 67, 1, 0, 4};
```

- java.lang.String 解析出来的字节码内容较多，当然包括的内容也多，比如魔数、版本、类、常量、方法等等。所以我们这里只截取部分进行解析。

2. 解析魔数并校验

很多文件格式都会规定满足该格式的文件必须以某几个固定字节开头，这几个字节主要起到标识作用，叫作魔数 (magic number)。

例如：

- PDF 文件以 4 字节“%PDF”(0x25、0x50、0x44、0x46)开头,
- ZIP 文件以 2 字节“PK”(0x50、0x4B)开头
- class 文件以 4 字节“0xCAFEBAE”开头

```

private static void readAndCheckMagic() {
    System.out.println("\r\n----- 校验魔数 -----");
    //从class字节码中读取前四位
    byte[] magic_byte = new byte[4];
    System.arraycopy(classData, 0, magic_byte, 0, 4);

    //将4位byte字节转成16进制字符串
    String magic_hex_str = new BigInteger(1, magic_byte).toString(16);
    System.out.println("magic_hex_str: " + magic_hex_str);

    //magic_hex_str 是16进制的字符串, cafebabe, 因为java中没有无符号整型, 所以如果想要无符号只能放到更高位中
    long magic_unsigned_int32 = Long.parseLong(magic_hex_str, 16);
    System.out.println("magic_unsigned_int32: " + magic_unsigned_int32);

    //魔数比对, 一种通过字符串比对, 另外一种使用假设的无符号16进制比较。如果使用无符号比较需要将0xCAFEBAE & 0xFFFFFFFFL与运算
    System.out.println("0xCAFEBAE & 0xFFFFFFFFL: " + (0xCAFEBAE & 0xFFFFFFFFL));

    if (magic_unsigned_int32 == (0xCAFEBAE & 0xFFFFFFFFL)) {
        System.out.println("class字节码魔数无符号16进制数值一致校验通过");
    } else {
        System.out.println("class字节码魔数无符号16进制数值一致校验拒绝");
    }
}

```

- 读取字节码中的前四位, **-54, -2, -70, -66**, 将这四位转换为 16 进制。
- 因为 java 中是没有无符号整型的, 所以只能用更高位存放。
- 解析后就是魔数的对比, 看是否与 CAFEBAE 一致。

测试结果

```
----- 校验魔数 -----
magic_hex_str: cafebabe
magic_unsigned_int32: 3405691582
0xCAFEBAE & 0xFFFFFFFFL: 3405691582
class 字节码魔数无符号 16 进制数值一致校验通过
```

3. 解析版本号信息

刚才我们已经读取了 4 位魔数信息，接下来再读取 2 位，是版本信息。

魔数之后是 class 文件的次版本号和主版本号，都是 u2 类型。假设某 class 文件的主版本号是 M，次版本号是 m，那么完整的版本号可以表示成 “M.m” 的形式。次版本号只在 J2SE 1.2 之前用过，从 1.2 开始基本上就没有什么用了（都是 0）。主版本号在 J2SE 1.2 之前是 45，从 1.2 开始，每次有大版本的 Java 版本发布，都会加 1 {45、46、47、48、49、50、51、52}

```
private static void readAndCheckVersion() {
    System.out.println("\r\n----- 校验版本号 -----");
    //从 class 字节码第4位开始读取，读取2位
    byte[] minor_byte = new byte[2];
    System.arraycopy(classData, 4, minor_byte, 0, 2);

    //将2位byte字节转成16进制字符串
    String minor_hex_str = new BigInteger(1, minor_byte).toString(16);
    System.out.println("minor_hex_str: " + minor_hex_str);

    //minor_unsigned_int32 转成无符号16进制
    int minor_unsigned_int32 = Integer.parseInt(minor_hex_str, 16);
    System.out.println("minor_unsigned_int32: " + minor_unsigned_int32);

    //从 class 字节码第6位开始读取，读取2位
    byte[] major_byte = new byte[2];
    System.arraycopy(classData, 6, major_byte, 0, 2);

    //将2位byte字节转成16进制字符串
    String major_hex_str = new BigInteger(1, major_byte).toString(16);
    System.out.println("major_hex_str: " + major_hex_str);

    //major_unsigned_int32 转成无符号16进制
    int major_unsigned_int32 = Integer.parseInt(major_hex_str, 16);
```

```
        System.out.println("major_unsigned_int32: " + major_unsigned_int32);
        System.out.println("版本号:
" + major_unsigned_int32 + "." + minor_unsigned_int32);
    }
}
```

- 这里有一个小技巧，class 文件解析出来是一整片的内容，JVM 需要按照虚拟机规范，一段一段的解析出所有的信息。
- 同样这里我们需要把 2 位 byte 转换为 16 进制信息，并继续从第 6 位继续读取 2 位信息。组合出来的才是版本信息。

测试结果

```
----- 校验版本号 -----
minor_hex_str: 0
minor_unsigned_int32: 0
major_hex_str: 34
major_unsigned_int32: 52
版本号: 52.0
```

4. 解析全部内容对照

按照 JVM 的加载过程，其实远不止魔数和版本号信息，还有很多其他内容，这里我们可以把测试结果展示出来，方便大家有一个学习结果的比对印象。

```
classpath: org.itstack.demo.jvm.classpath.Classpath@4bf558aa class:
java.lang.String args: null
version: 52.0
constants count: 540
access flags: 0x31
this class: java/lang/String
super class: java/lang/Object
interfaces: [java/io/Serializable, java/lang/Comparable, java/lang/CharSequence]
fields count: 5
value   [C
hash   I
serialVersionUID   J
serialPersistentFields   [Ljava/io/ObjectStreamField;
CASE_INSENSITIVE_ORDER   Ljava/util/Comparator;
methods count: 94
<init>   ()V
<init>   (Ljava/lang/String;)V
```

```
<init>      ([C)V
<init>      ([CII)V
<init>      ([III)V
<init>      ([BIII)V
<init>      ([BI)V
checkBounds      ([BII)V
<init>      ([BILjava/lang/String;)V
<init>      ([BILjava/nio/charset/Charset;)V
<init>      ([BLjava/lang/String;)V
<init>      ([BLjava/nio/charset/Charset;)V
<init>      ([BII)V
<init>      ([B)V
<init>      (Ljava/lang/StringBuffer;)V
<init>      (Ljava/lang/StringBuilder;)V
<init>      ([CZ)V
length      ()I
isEmpty      ()Z
charAt      (I)C
codePointAt      (I)I
codePointBefore      (I)I
codePointCount      (II)I
offsetByCodePoints      (II)I
getChars      ([CI)V
getChars      (II[CI)V
getBytes      (II[B)V
getBytes      (Ljava/lang/String;)[[B
getBytes      (Ljava/nio/charset/Charset;)[[B
getBytes      ()[[B
equals      (Ljava/lang/Object;)Z
contentEquals      (Ljava/lang/StringBuffer;)Z
nonSyncContentEquals      (Ljava/lang/AbstractStringBuilder;)Z
contentEquals      (Ljava/lang/CharSequence;)Z
equalsIgnoreCase      (Ljava/lang/String;)Z
compareTo      (Ljava/lang/String;)I
compareToIgnoreCase      (Ljava/lang/String;)I
regionMatches      (ILjava/lang/String;II)Z
regionMatches      (ZILjava/lang/String;II)Z
startsWith      (Ljava/lang/String;I)Z
startsWith      (Ljava/lang/String;)Z
```

```
endsWith      (Ljava/lang/String;)Z
hashCode      ()I
indexOf       (I)I
indexOf       (II)I
indexOfSupplementary (II)I
lastIndexOf   (I)I
lastIndexOf   (II)I
lastIndexOfSupplementary (II)I
indexOf       (Ljava/lang/String;)I
indexOf       (Ljava/lang/String;I)I
indexOf       ([CII)Ljava/lang/String;I)
indexOf       ([CII[CIII])I
lastIndexOf   (Ljava/lang/String;)I
lastIndexOf   (Ljava/lang/String;I)I
lastIndexOf   ([CII)Ljava/lang/String;I)
lastIndexOf   ([CII[CIII])I
substring     (I)Ljava/lang/String;
substring     (II)Ljava/lang/String;
subSequence   (II)Ljava/lang/CharSequence;
concat       (Ljava/lang/String;)Ljava/lang/String;
replace       (CC)Ljava/lang/String;
matches       (Ljava/lang/String;)Z
contains      (Ljava/lang/CharSequence;)Z
replaceFirst  (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
replaceAll   (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
replace       (Ljava/lang/CharSequence;Ljava/lang/CharSequence;)Ljava/lang/String;
split         (Ljava/lang/String;I)[Ljava/lang/String;
split         (Ljava/lang/String;)[Ljava/lang/String;
join          (Ljava/lang/CharSequence;[Ljava/lang/CharSequence;)Ljava/lang/String;
join          (Ljava/lang/CharSequence;Ljava/lang/Iterable;)Ljava/lang/String;
toLowerCase   (Ljava/util/Locale;)Ljava/lang/String;
toLowerCase   ()Ljava/lang/String;
toUpperCase   (Ljava/util/Locale;)Ljava/lang/String;
toUpperCase   ()Ljava/lang/String;
trim          ()Ljava/lang/String;
toString      ()Ljava/lang/String;
toCharArray   ()[C
format        (Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;
format        (Ljava/util/Locale;Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String
```

```
g;
valueOf    (Ljava/lang/Object;)Ljava/lang/String;
valueOf    ([C)Ljava/lang/String;
valueOf    ([CII)Ljava/lang/String;
copyValueOf   ([CII)Ljava/lang/String;
copyValueOf   ([C)Ljava/lang/String;
valueOf    (Z)Ljava/lang/String;
valueOf    (C)Ljava/lang/String;
valueOf    (I)Ljava/lang/String;
valueOf    (J)Ljava/lang/String;
valueOf    (F)Ljava/lang/String;
valueOf    (D)Ljava/lang/String;
intern     ()Ljava/lang/String;
compareTo   (Ljava/lang/Object;)I
<clinit>   ()V
```

```
Process finished with exit code 0
```

- 如果大家对这部分验证、准备、解析，的实现过程感兴趣，可以参照这部分用 Java 实现的 JVM 源码：<https://github.com/fuzhengwei/itstack-demo-jvm>

五、总结

- 学习 JVM 最大的问题是不好实践，所以本文以案例实操的方式，学习 JVM 的加载解析过程。也让更多的对 JVM 感兴趣的研发，能更好的接触到 JVM 并深入的学习。
- 有了以上这段代码，大家可以参照 JVM 虚拟机规范，在调试 Java 版本的 JVM，这样就可以非常容易理解整个 JVM 的加载过程，都做了什么。
- 如果大家需要文章中一些原图 xmind 或者源码，可以添加作者小傅哥(fustack)，或者关注公众号：bugstack 虫洞栈进行获取。好了，本章节就扯到这，后续还有很多努力，持续原创，感谢大家的支持！

第 3 节：JVM 内存模型

看了一篇文章 [30 岁有多难！](#)

每篇文章的开篇总喜欢写一些，从个人视角看这个世界的感悟。

最近看到一篇文章，[30 岁有多难](#)。文中的一些主人公好像在学业、工作、生活、爱情等方面都过的都不如意。要不是错过这，要不是走错那。总结来看，就像是很倒霉的一群倒霉蛋儿在跟生活对干！

但其实每个人可能都遇到过生活中最难的时候，或早或晚。就像我刚毕业不久时一连串遇到；[冬天里丢过第一部手机](#)、[修一个进了水的电脑](#)、[租的房子第一次被骗](#)，一连串下来头一次要赶在工资没发的时候，选择少吃早饭还是午饭，看看能扛过去那顿。

哈哈哈哈，现在想想还挺有意思的，不过这些乱遭的事很多是自己的意识和能力不足时做出的错误选择而导致的。

人那，想开车就要考驾照，想走远就要有能力。多提升认知，多拓宽眼界！[生活的意义就是不断的更新自己！](#)

一、面试题

谢飞机，小记！，冬风吹、战鼓擂。被窝里，谁怕谁。

谢飞机：歪？大哥，你在吗？

面试官：咋了，大周末的，这么早打电话！？

谢飞机：我梦见，我去谷歌写 JVM 了，给你们公司用，之后蹦了，让我起来改 bug！

面试官：啊！？啊，那我问你，JDK 1.8 与 JDK 1.7 在运行时数据区的设计上，你都怎么做的优化策略的？

谢飞机：我没写这，我不知道！

面试官：擦。。。

二、JDK1.6、JDK1.7、JDK1.8 内存模型演变

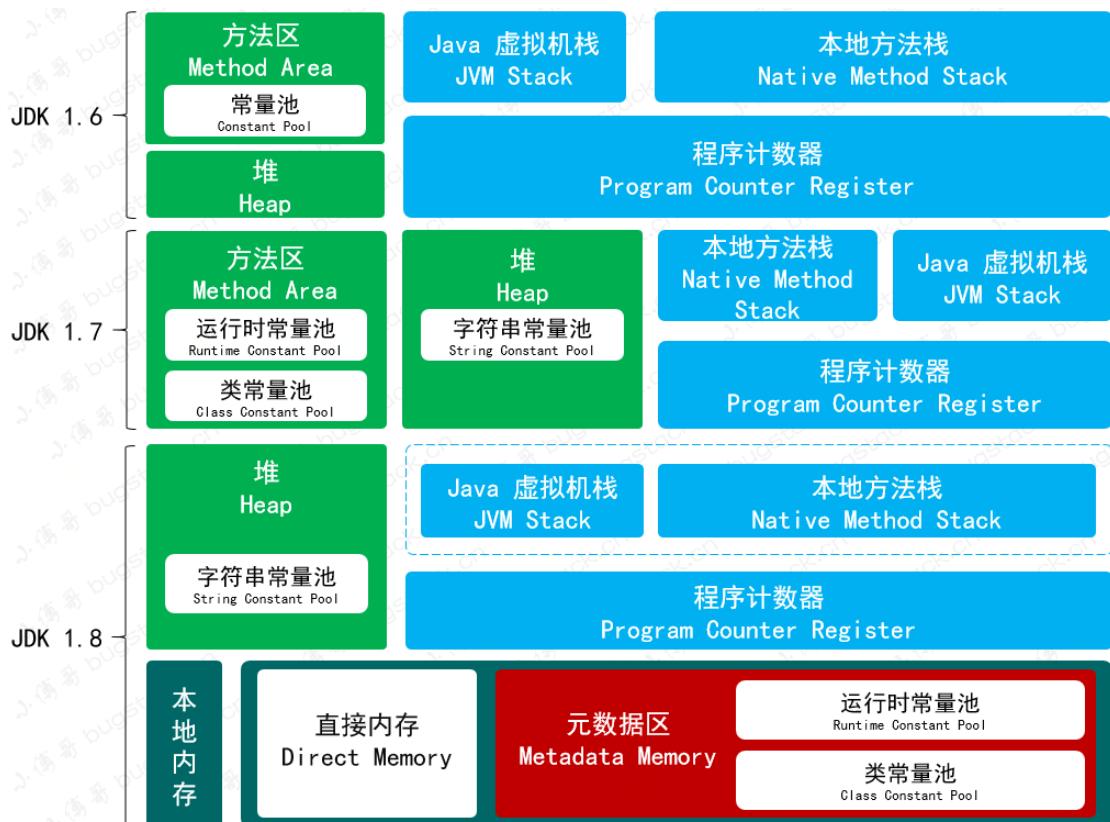


图 25-1 JDK1.6、JDK1.7、JDK1.8，内存模型演变

如图 25-1 是 JDK 1.6、1.7、1.8 的内存模型演变过程，其实这个内存模型就是 JVM 运行时数据区依照 JVM 虚拟机规范的具体实现过程。

在图 25-1 中各个版本的迭代都是为了更好的适应 CPU 性能提升，最大限度提升的 JVM 运行效率。这些版本的 JVM 内存模型主要有以下差异：

- JDK 1.6：有永久代，静态变量存放在永久代上。
- JDK 1.7：有永久代，但已经把字符串常量池、静态变量，存放在堆上。逐渐的减少永久代的使用。
- JDK 1.8：无永久代，运行时常量池、类常量池，都保存在元数据区，也就是常说的元空间。但字符串常量池仍然存放在堆上。

三、内存模型各区域介绍

1. 程序计数器

- 较小的内存空间、线程私有，记录当前线程所执行的字节码行号。

- 如果执行 Java 方法，计数器记录虚拟机字节码当前指令的地址，本方法则为空。
- 这一块区域没有任何 OutOfMemoryError 定义。

以上，就是关于程序计数器的定义，如果这样看没有感觉，我们举一个例子。定义一段 Java 方法的代码，这段代码是计算圆形的周长。

```
public static float circumference(float r){
    float pi = 3.14f;
    float area = 2 * pi * r;
    return area;
}
```

接下来，如图 25-2 是这段代码的在虚拟机中的执行过程，左侧是它的程序计数器对应的行号。

Instructions			Local Vars			Operand Stack
pc	指令	描述	#0	#1	#2	
程序计数器			1.6			
	00 ldc	第一条指定是 ldc，把3.14f推出栈顶	1.6			3.14
	02 fstore_1	把栈顶的3.14f弹出，放到#1号局部变量表中	1.6	3.14		
	03 fconst_2	把2.0f推到栈顶	1.6	3.14		2.0
	04 fload_1	把#1号局部变量推入栈顶	1.6	3.14		3.14 2.0
	05 fmul	将栈顶的两个浮点数弹出，相乘，然后把结果推入栈顶	1.6	3.14		6.28
	06 fload_0	把#0号局部变量推入栈顶	1.6	3.14	1.6	6.28
	07 fmul	执行浮点数乘法	1.6	3.14		10.0 48
	08 fstore_2	把操作数栈顶的float值弹出，放入#2号局部变量表	1.6	3.14	10.04 8	
	09 fload_2	把#2号局部变量推入操作数栈顶	1.6	3.14	10.04 8	10.0 48
	10 freturn	最后把操作数栈顶的float变量弹出，返回给方法调用者	1.6	3.14	10.04 8	

图 25-2 程序计数器

- 这些行号每一个都会对应一条需要执行的字节码指令，是压栈还是弹出或是执行计算。
- 之所以说是线程私有的，因为如果不是私有的，那么整个计算过程最终的结果也将错误。

2. Java 虚拟机栈

- 每一个方法在执行的同时，都会创建出一个栈帧，用于存放局部变量表、操作数栈、动态链接、方法出口、线程等信息。
- 方法从调用到执行完成，都对应着栈帧从虚拟机中入栈和出栈的过程。
- 最终，栈帧会随着方法的创建到结束而销毁。

可能这么只从定义看上去仍然没有什么感觉，我们再找一个例子。

这是一个关于斐波那契数列（Fibonacci sequence）求值的例子，我们通过斐波那契数列在虚拟机中的执行过程，来体会 Java 虚拟机栈的用途。

斐波那契数列（Fibonacci sequence），又称黄金分割数列、因数学家列昂纳多·斐波那契（Leonardoda Fibonacci）以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：1、1、2、3、5、8、13、21、34、……在数学上，斐波那契数列以如下被以递推的方法定义： $F(1)=1$ ， $F(2)=1$ ， $F(n)=F(n-1)+F(n-2)$ ($n \geq 3$, $n \in \mathbb{N}^*$) 在现代物理、准晶体结构、化学等领域，斐波那契数列都有直接的应用，为此，美国数学会从 1963 年起出版了以《斐波那契数列季刊》为名的一份数学杂志，用于专门刊载这方面的研究成果。

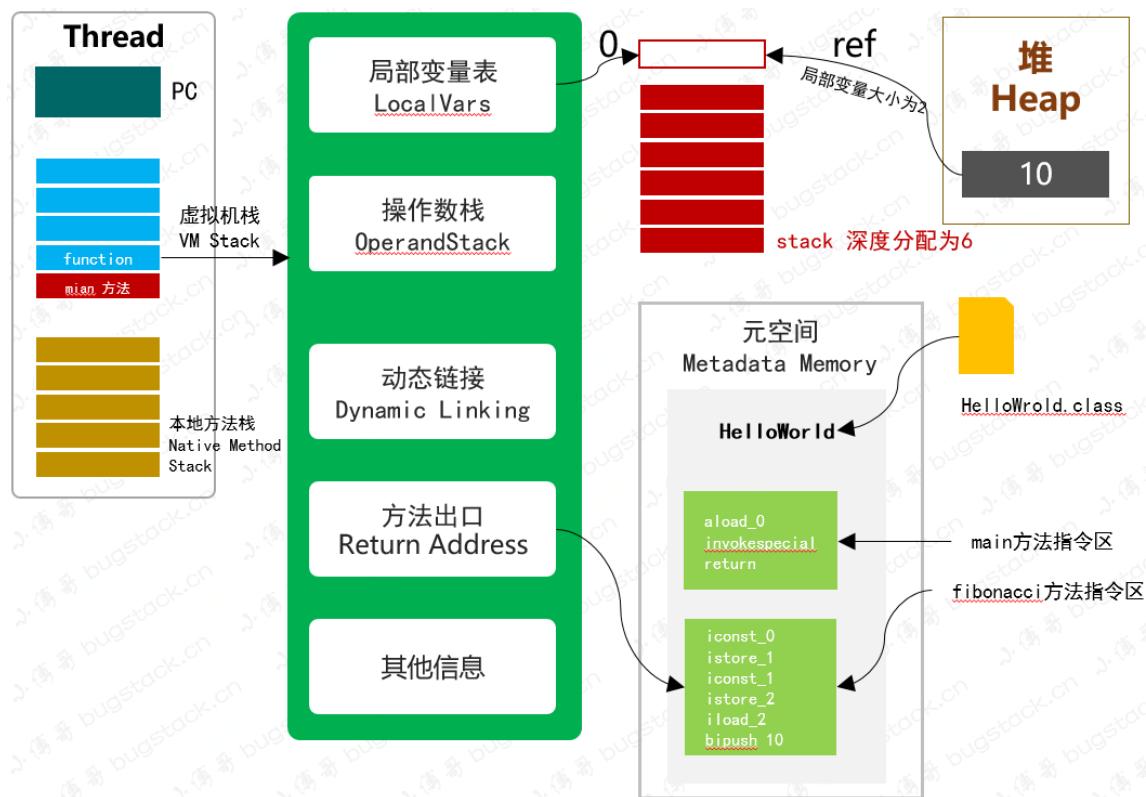


图 25-3 斐波那契数列在虚拟机栈中的执行过程

- 整个这段流程，就是方法的调用和返回。在调用过程申请了操作数栈的深度和局部变量的大小。
- 以及相应的信息从各个区域获取并操作，其实也就是入栈和出栈的过程。

3. 本地方法栈

- 本地方法栈与 Java 虚拟机栈作用类似，唯一不同的就是本地方法栈执行的是 Native 方法，而虚拟机栈是为 JVM 执行 Java 方法服务的。
- 另外，与 Java 虚拟机栈一样，本地方法栈也会抛出 StackOverflowError 和 OutOfMemoryError 异常。
- JDK1.8 HotSpot 虚拟机直接就把本地方法栈和虚拟机栈合二为一。

关于本地方法栈在以上的例子已经涉及了这部分内容，这里就不在赘述了。

4. 堆和元空间

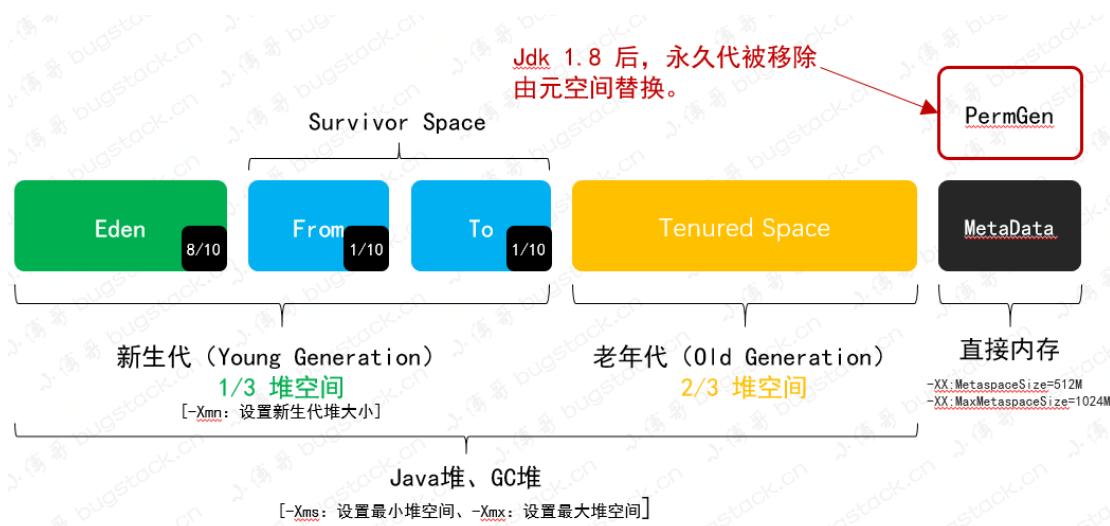


图 25-4 Java 堆区域划分

- JDK 1.8 JVM 的内存结构主要由三大块组成：堆内存、元空间和栈，Java 堆是内存空间占据最大的一块区域。
- Java 堆，由年轻代和年老代组成，分别占据 1/3 和 2/3。
- 而年轻代又分为三部分，Eden、From Survivor、To Survivor，占据比例为 8:1:1，可调。
- 另外这里我们特意画出了元空间，也就是直接内存区域。在 JDK 1.8 之后就不在堆上分配方法区了。
- 元空间从虚拟机 Java 堆中转移到本地内存，默认情况下，元空间的大小仅受本地内存的限制，说白了也就是以后不会因为永久代空间不够而抛出 OOM 异常出现了。*jdk1.8 以前版本的 class 和 JAR 包数据存储在 PermGen 下面，PermGen 大小是固定的，而且项目之间无法共用，公有的 class，所以比较容易出现 OOM 异常。*

- 升级 JDK 1.8 后，元空间配置参数，`-XX:MetaspaceSize=512M`
`XX:MaxMetaspaceSize=1024M`。教你个小技巧通过 jps、jinfo 查看元空间，如下：

```
E:\itstack\git\github.com\interview>jps  
66016 Jps  
59000  
32172
```

```
E:\itstack\git\github.com\interview>jinfo -flag MetaspaceSize 32172  
-XX:MetaspaceSize=21807104
```

```
E:\itstack\git\github.com\interview>jinfo -flag MaxMetaspaceSize 32172  
-XX:MaxMetaspaceSize=18446744073709486080
```

- 通过命令查看元空间
- 通过 jinfo 查看默认 MetaspaceSize 大小（约 20M）,MaxMetaspaceSize 比较大。

其他：关于 JDK1.8 元空间的介绍：Move part of the contents of the permanent generation in Hotspot to the Java heap and the remainder to native memory. <http://openjdk.java.net/jeps/122>

5. 常量池

- 从 JDK 1.7 开始把常量池从永久代中剥离，直到 JDK1.8 去掉了永久代。而字符串常量池一直放在堆空间，用于存储字符串对象，或是字符串对象的引用。

四、手撸虚拟机(内存模型)

其实以上的内容，已经完整的介绍了 JVM 虚拟机的内存模型，也就是运行时数据区的结构。但是这东西看完可能就忘记了，因为缺少一个可亲手操作的代码。

所以，这里我给大家用 Java 代码写一段关于数据槽、栈帧、局部变量、虚拟机栈以及堆的代码结构，让大家更好的加深对虚拟机内存模型的印象。

1. 工程结构

运行时数据区

|— heap

```
|   └── constantpool
|   └── methodarea
|   |   ├── Class.java
|   |   ├── ClassMember.java
|   |   ├── Field.java
|   |   ├── Method.java
|   |   ├── MethodDescriptor.java
|   |   ├── MethodDescriptorParser.java
|   |   ├── MethodLookup.java
|   |   ├── Object.java
|   |   ├── Slots.java
|   |   └── StringPool.java
|   └── ClassLoader.java
└── Frame.java
└── JvmStack.java
└── LocalVars.java
└── OperandStack.java
└── Slot.java
└── Thread.java
```

以上这部分就是使用 Java 实现的部分 JVM 虚拟机功能，这部分主要包括如下内容：

- Frame, 栈帧
- JvmStack, 虚拟机栈
- LocalVars, 局部变量
- OperandStack, 操作数栈
- Slot, 数据槽
- Thread, 线程
- heap, 堆, 里面包括常量池和方法区

2. 重点代码

操作数栈 OperandStack

```
public class OperandStack {
```



```
    private int size = 0;
    private Slot[] slots;
```

```
public OperandStack(int maxStack) {
    if (maxStack > 0) {
        slots = new Slot[maxStack];
        for (int i = 0; i < maxStack; i++) {
            slots[i] = new Slot();
        }
    }
    //...
}
```

虚拟机栈 OperandStack

```
public class JvmStack {
```

```
    private int maxSize;
    private int size;
    private Frame _top;

    //...
}
```

栈帧 Frame

```
public class Frame {
```

```
//stack is implemented as Linked List
Frame lower;
```

```
//局部变量表
private LocalVars localVars;
```

```
//操作数栈
private OperandStack operandStack;
```

```
private Thread thread;
```

```
private Method method;
```

```
private int nextPC;
```

```
//...
}
```

- 关于代码结构看到这有点感觉了吗？
- Slot 数据槽，就是一个数组结构，用于存放数据的。
- 操作数栈、局部变量表，都是使用数据槽进行入栈入栈操作。
- 在栈帧里，可以看到连接、局部变量表、操作数栈、方法、线程等，那么文中说到的当有一个新的每一个方法在执行的同时，都会创建出一个栈帧，是不就对了上，可以真的理解了。
- 如果你对 JVM 的实现感兴趣，可以阅读用 Java 实现 JVM 源码：

<https://github.com/fuzhengwei/itstack-demo-jvm>

五、jconsole 监测元空间溢出

不是说 JDK 1.8 的内存模型把永久代下掉，换上元空间了吗？但不测试下，就感受不到呀，没有证据！

所有关于代码逻辑的学习，都需要有数据基础和证明过程，这样才能有深刻的印象。走着，带你把元空间干满，让它 OOM！

1. 找段持续创建大对象的代码

```
public static void main(String[] args) throws InterruptedException {

    Thread.sleep(5000);

    ClassLoadingMXBean loadingBean = ManagementFactory.getClassLoadingMXBean();
    while (true) {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(MetaSpaceOomMock.class);
        enhancer.setCallbackTypes(new Class[]{Dispatcher.class, MethodInterceptor.class});
        enhancer.setCallbackFilter(new CallbackFilter() {
            @Override
            public int accept(Method method) {
                return 1;
            }
            @Override
        });
    }
}
```

```
public boolean equals(Object obj) {
    return super.equals(obj);
}
});

System.out.println(enhancer.createClass().getName() + loadingBean.getTotalLoadedClassCount() + loadingBean.getLoadedClassCount() + loadingBean.getUnloadedClassCount());
}

}
```

- 网上找了一段基于 CGLIB 的，你可以写一些其他的。
- `Thread.sleep(5000);`，睡一会，方便我们点检测，要不程序太快就异常了。

2. 调整元空间大小

默认情况下元空间太大了，不方便测试出结果，所以我们把它调的小一点。

```
-XX:MetaspaceSize=8m
-XX:MaxMetaspaceSize=80m
```

3. 设置监控参数

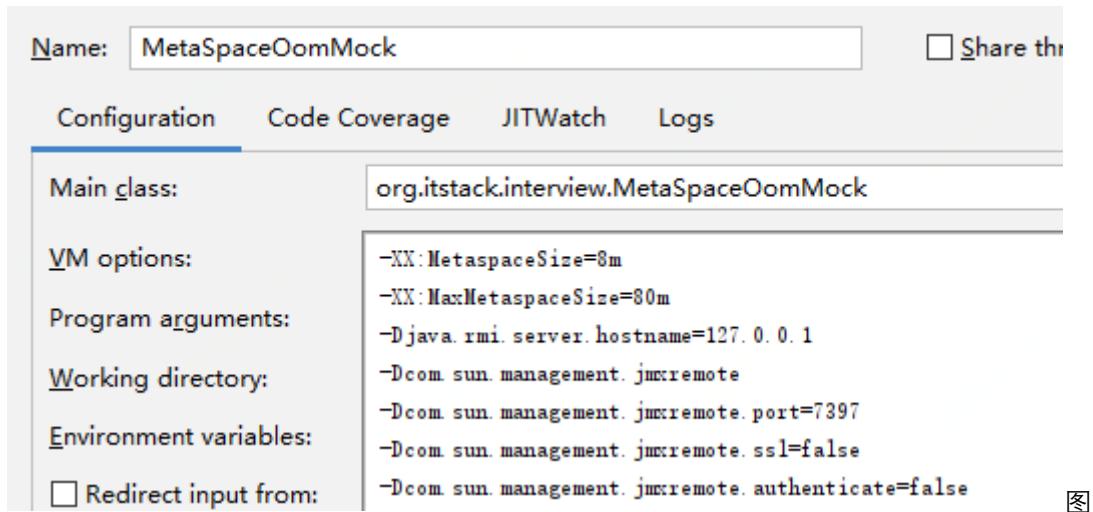
基于 jconsole 监控，我们需要设置下参数。

```
-Djava.rmi.server.hostname=127.0.0.1
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=7397
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
```

4. 测试运行

4.1 配置参数

以上的测试参数，配置到 IDEA 中运行程序里就可以，如下：



图

25-5 设置程序运行参数，监控 OOM

另外，jconsole 可以通过 IDEA 提供的 Terminal 启动，直接输入 `jconsole`，回车即可。

4.2 测试结果

```
org.itstack.interview.MetaSpaceOomMock$$EnhancerByCGLIB$$bd2bb16e999099900
org.itstack.interview.MetaSpaceOomMock$$EnhancerByCGLIB$$9c774e64999199910
org.itstack.interview.MetaSpaceOomMock$$EnhancerByCGLIB$$cac97732999299920
org.itstack.interview.MetaSpaceOomMock$$EnhancerByCGLIB$$91c6a15a999399930
Exception in thread "main" java.lang.IllegalStateException: Unable to load cache item
	at net.sf.cglib.core.internal>LoadingCache.createEntry>LoadingCache.java:79)
	at net.sf.cglib.core.internal>LoadingCache.get>LoadingCache.java:34)
	at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData.get>AbstractClassGenerator.java:119)
	at net.sf.cglib.core.AbstractClassGenerator.create>AbstractClassGenerator.java:294)
	at net.sf.cglib.proxy.Enhancer.createHelper>Enhancer.java:480)
	at net.sf.cglib.proxy.Enhancer.createClass>Enhancer.java:337)
	at org.itstack.interview.MetaSpaceOomMock.main>MetaSpaceOomMock.java:34)
Caused by: java.lang.OutOfMemoryError: Metaspace
	at java.lang.Class.forName0>Native Method)
	at java.lang.Class.forName>Class.java:348)
	at net.sf.cglib.core.ReflectUtils.defineClass>ReflectUtils.java:467)
	at net.sf.cglib.core.AbstractClassGenerator.generate>AbstractClassGenerator.java:39)
	at net.sf.cglib.proxy.Enhancer.generate>Enhancer.java:492)
```

```

at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData$3.apply(AbstractClassG
enerator.java:96)
at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData$3.apply(AbstractClassG
enerator.java:94)
at net.sf.cglib.core.internal.LoadingCache$2.call(LoadingCache.java:54)
at java.util.concurrent.FutureTask.run$$capture(FutureTask.java:266)
at java.util.concurrent.FutureTask.run(FutureTask.java)
at net.sf.cglib.core.internal.LoadingCache.createEntry(LoadingCache.java:61)
...

```

- 要的就是这句，`java.lang.OutOfMemoryError: Metaspace`，元空间 OOM，证明 JDK1.8 已经去掉永久代，换位元空间。

4.3 监控截图

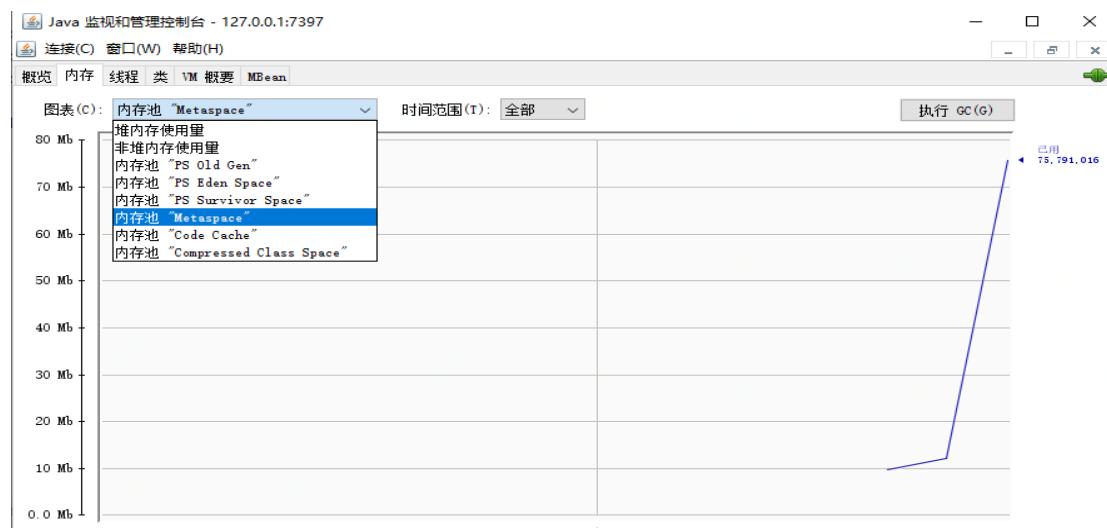


图 25-6 jconsole 监测元空间溢出

- 图 25-6，就是监测程序 OOM 时的元空间表现。这回对这个元空间就有感觉了吧！

六、总结

- 本文从 JDK 各个版本关于内存模型结构的演变，来了解各个区域，包括：程序计数器、Java 虚拟机栈、本地方法栈、堆和元空间。并了解从 JDK 1.8 开始去掉方法区引入元空间的核心目的和作用。
- 在通过手撸 JVM 代码的方式让大家对运行时数据区有一个整体的认知，也通过这样的方式让大家对学习这部分知识有一个抓手。

- 最后我们通过 jconsole 检测元空间溢出的整个过程，来学以致用，看看元空间到底在解决什么问题以及怎么测试。

第 4 节：JVM 故障处理工具

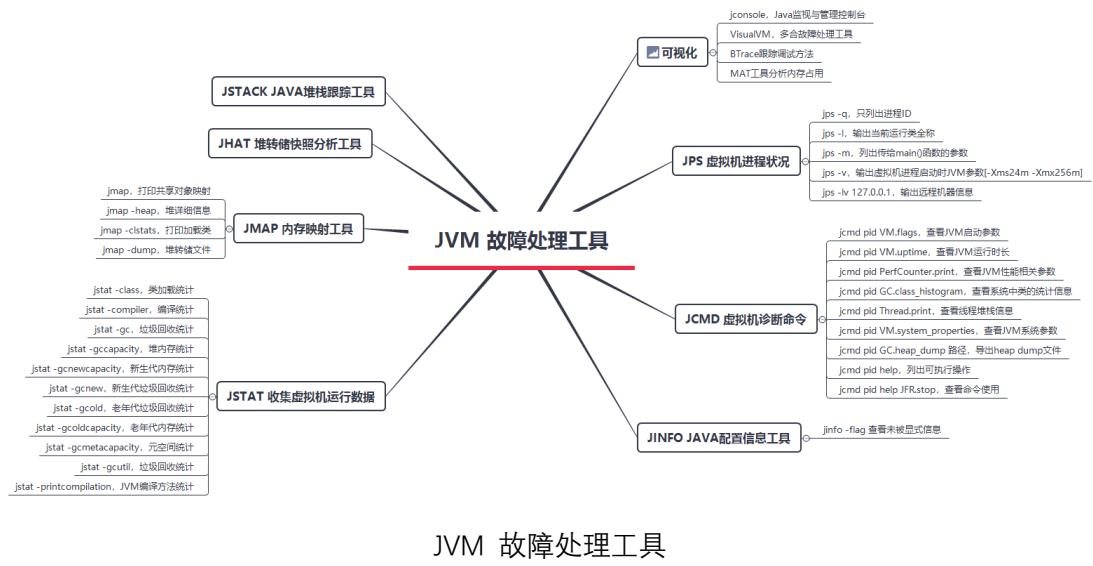
用都用不到怎么学？

没有场景、没有诉求，怎么学习这些似乎用不上知识点。

其实最好的方式就是归纳、整理、实践、输出，一套组合拳下来，你就掌握了这个系列的知识了。

但在当前阶段可能真的用不上，JVM 是一个稳定服务，哪能天天出问题，哪需要你老排查。又不是像你写的代码那样！，可是知识的学习就是把你垫基到更高层次后，才有机会接触更有意思的工作和技术创新。如果只是单纯的学几个指令，其实并没有多有意思。但让你完成一套全链路监控，里面需要含有一次方法调用的整体耗时、执行路径、参数信息、异常结果、GC 次数、堆栈数据、分代内容等等的时候，那么你的知识储备够开发一个这样的系统吗？

好，先上图看看本文要讲啥，再跟着小傅哥的步伐往下走。



一、面试题

谢飞机，小记！，周末休息在家无聊，把已经上灰了的 JVM 虚拟机学习翻出来。

谢飞机：呱...呱...，喂大哥，这个，这个 JVM 虚拟机看啥呀。

面试官：看啥？不知道从哪开始？嗯，那你从问题点下手！

谢飞机：啥问题点呢，我就是不知道自己不会啥，也不知道问你啥。

面试官：啊！那我问你个，怎么通过 JVM 故障处理工具，查看 JVM 启动时参数都配置了什么呢？

谢飞机：这个！？不道呀！

面试官：那你熟悉的监控指令都有啥，如果问你堆内存统计如何统计，你可知晓！？

谢飞机：也不知道，哈哈哈，好像知道要去看啥了！

面试官：去吧，带着问题看，看完整理出来！

二、基础故障处理工具

1. jps 虚拟机进程状况

jps (JVM Process Status Tool)，它的功能与 ps 命令类似，可以列出正在运行的虚拟机进程，并显示虚拟机执行主类 (Main Class, main()函数所在的类) 名称以及这些进程的本地虚拟机唯一 ID (Local Virtual Machine Identifier, LVMID)，类似于 ps -ef | grep java 的功能。

这小家伙虽然不大，功能又单一。但可以说基本你用其他命令都得先用它，来查询到 LVMID 来确定要监控的是哪个虚拟机进程。

命令格式

`jps [options] [hostid]`

- options: 选项、参数，不同的参数可以输出需要的信息
- hostid: 远程查看

选项列表

选项	描述
-q	只输出进程 ID，忽略主类信息
-l	输出主类全名，或者执行 JAR 包则输出路径
-m	输出虚拟机进程启动时传递给主类 main()函数的参数
-v	输出虚拟机进程启动时的 JVM 参数

1.1 jps -q，只列出进程 ID

```
E:\itstack\git\github.com\interview>jps -q
104928
111552
26852
96276
59000
```

8460

76188

1.2 jps -l，输出当前运行类全称

```
E:\itstack\git\github.com\interview>jps -l
111552 org/netbeans/Main
26852
96276 org.jetbrains.jps.cmdline.Launcher
59000
62184 sun.tools.jps.Jps
8460 org/netbeans/Main
76188 sun.tools.jstatd.Jstatd
```

- 用这个命令输出的内容就清晰多了，`-l` 也是非常常用的一个参数选项。

1.3 jps -m，列出传给 main() 函数的参数

```
E:\itstack\git\github.com\interview>jps -m
111552 Main --branding visualvm --
cachedir C:/Users/xiaofuge/AppData/Local/VisualVM/Cache/8u131 --
openid 3041391569375200
26852
96276 Launcher C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/plugins/java/lib/j
avac2.jar;C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/plugins/java/lib/aether
-api-1.1.0.jar;C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/lib/jna-
platform.jar;C:/Program Fi
les/JetBrains/IntelliJ IDEA 2019.3.1/lib/guava-27.1-
jre.jar;C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/lib/httpclient-
4.5.10.jar;C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/lib/forms-1.1-
preview.jar;C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/plu
gins/java/lib/aether-connector-basic-
1.1.0.jar;C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/plugins/java/lib/maven-
model-builder-3.3.9.jar;C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/lib/jps-
model.jar;C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/plu
gins/java/lib/maven-model-
3.3.9.jar;C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/plugins/java/lib/aether
-impl-1.1.0.jar;C:/Program Files/JetBrains/IntelliJ IDEA 2019.3.1/lib/gson-
2.8.5.jar;C:/Program File
```

```
59000
16844 Jps -m
8460 Main --branding visualvm --
cachedir C:\Users\xiaofuge\AppData\Local\VisualVM\Cache\8u131 --
openid 3041414336579200
76188 Jstatd
```

1. 4 jps -v, 输出虚拟机进程启动时 JVM 参数 [-Xms24m -Xmx256m]

```
E:\itstack\git\github.com\interview>jps -v
111552 Main -Xms24m -Xmx256m -Dsun.jvmstat.perdata.syncWaitMs=10000 -
Dsun.java2d.noddraw=true -Dsun.java2d.d3d=false -Dnetbeans.keyring.no.master=true -
Dplugin.manager.install.global=false --add-exports=java.desktop/sun.awt=ALL-
UNNAMED --add-exports=jdk.jvmstat/sun
.jvmstat.monitor.event=ALL-UNNAMED --add-
exports=jdk.jvmstat/sun.jvmstat.monitor=ALL-UNNAMED --add-
exports=java.desktop/sun.swing=ALL-UNNAMED --add-
exports=jdk.attach/sun.tools.attach=ALL-UNNAMED --add-modules=java.activation -
XX:+IgnoreUnrecognizedVMOptions -Djdk.
home=C:/Program Files/Java/jdk1.8.0_161 -
Dnetbeans.home=C:/Program Files/Java/jdk1.8.0_161/lib/visualvm/platform -
Dnetbeans.user=C:/Users\xiaofuge1\AppData\Roaming\VisualVM\8u131 -
Dnetbeans.default_userdir_root=C:/Users\xiaofuge1\AppData\Roaming\VisualVM -XX:+H
eapDumpOnOutOfMemoryError -
XX:HeapDumpPath=C:/Users\xiaofuge1\AppData\Roaming\VisualVM\8u131\var\log\heapdump.
hprof -Dsun.awt.keepWorkingSetOnMinimize=true -
Dnetbeans.dirs=C:/Program Files/Java/jdk1.8.0_161/lib/visualvm;C:/Program
59000 -Dfile.encoding=UTF-8 -Xms128m -Xmx1024m -XX:MaxPermSize=256m
76188 Jstatd -
Denv.class.path=.;C:/Program Files/Java/jre1.8.0_161/lib;C:/Program Files/Java/jre1
.8.0_161/lib/tool.jar; -Dapplication.home=C:/Program Files/Java/jdk1.8.0_161 -
Xms8m -Djava.security.policy=jstatd.all.policy
```

1. 5 jps -lv 127.0.0.1, 输出远程机器信息

jps 链接远程输出 JVM 信息，需要注册 RMI，否则会报错 `RMI Registry not available at 127.0.0.1`。

注册 RMI 开启 jstatd 在你的 `C:/Program Files/Java/jdk1.8.0_161/bin` 目录下添加名称为 `jstatd.all.policy` 的文件。无其他后缀

`jstatd.all.policy` 文件内容如下:

```
grant codebase "file:${java.home}/../lib/tools.jar" {  
    permission java.security.AllPermission;  
};
```

添加好配置文件后，在 bin 目录下注册添加的 `jstatd.all.policy` 文件：

```
C:\Program Files\Java\jdk1.8.0_161\bin>jstatd -J-Djava.security.policy=jstatd.all.policy
```

顺利的话现在就可以查看原创机器 JVM 信息了，如下：

```
E:\itstack\git\github.com\interview>jps -l 127.0.0.1  
111552 org/netbeans/Main  
26852  
96276 org.jetbrains.jps.cmdline.Launcher  
36056 sun.tools.jps.Jps  
59000  
8460 org/netbeans/Main  
76188 sun.tools.jstatd.Jstatd
```

- 也可以组合使用 jps 的选项参数，比如：`jps -lm 127.0.0.1`

2. jcmand 虚拟机诊断命令

jcmand，是从 jdk1.7 开始新发布的 JVM 相关信息诊断工具，可以用它来导出堆和线程信息、查看 Java 进程、执行 GC、还可以进行采样分析（jmc 工具的飞行记录器）。注意其使用条件是只能在被诊断的 JVM 同台 sever 上，并且具有相同的用户和组（user and group）。

命令格式

```
jcmand <pid | main class> <command ...|PerfCounter.print|-f file>
```

- pid，接收诊断命令请求的进程 ID
 - main class，接收诊断命令请求的进程 main 类。
- command，接收诊断命令请求的进程 main 类。
- PerfCounter.print，打印目标 Java 进程上可用的性能计数器。
- -f file，从文件 file 中读取命令，然后在目标 Java 进程上调用这些命令。
- -l，查看所有进程列表信息。
- -h、-help，查看帮助信息。

2.1 jcmd pid VM.flags，查看 JVM 启动参数

```
E:\itstack\git\github.com\interview>jcmd 111552 VM.flags
111552:
-XX:CICompilerCount=4 -XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath=C:\Users\xiaofuge1\AppData\Roaming\VisualVM\8u131\var\log\heapdump.
hprof -XX:+IgnoreUnrecognizedVMOptions -XX:InitialHeapSize=25165824 -
XX:MaxHeapSize=268435456 -XX:MaxNewSize=89128960 -XX:Min
HeapDeltaBytes=524288 -XX:NewSize=8388608 -XX:OldSize=16777216 -
XX:+UseCompressedClassPointers -XX:+UseCompressedOops -
XX:+UseFastUnorderedTimeStamps -XX:-UseLargePagesIndividualAllocation -
XX:+UseParallelGC
```

2.2 jcmd pid VM.uptime，查看 JVM 运行时长

```
E:\itstack\git\github.com\interview>jcmd 111552 VM.uptime
111552:
583248.912 s
```

2.3 jcmd pid PerfCounter.print，查看 JVM 性能相关参数

```
E:\itstack\git\github.com\interview>jcmd 111552 PerfCounter.print
111552:
java.ci.totalTime=56082522
java.cls.loadedClasses=5835
java.cls.sharedLoadedClasses=0
java.cls.sharedUnloadedClasses=0
java.cls.unloadedClasses=37
...
```

2.4 jcmd pid GC.class_histogram，查看系统中类的统计信息

```
E:\itstack\git\github.com\interview>jcmd 111552 GC.class_histogram
111552:
```

num	#instances	#bytes	class name
1:	50543	3775720	[C
2:	3443	2428248	[I
3:	50138	1203312	java.lang.String

```
4:      25351      811232  java.util.HashMap$Node
5:      6263       712208  java.lang.Class
6:      3134       674896  [B
7:      6687       401056  [Ljava.lang.Object;
8:      2468       335832  [Ljava.util.HashMap$Node;
```

2.5 jcmd pid Thread.print，查看线程堆栈信息

```
E:\itstack\git\github.com\interview>jcmd 111552 Thread.print
111552:
2021-01-10 23:31:13
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.161-b12 mixed mode):

"Computes values in handlers" #52 daemon prio=5 os_prio=0 tid=0x0000000019839000 ni
d=0x16014 waiting for monitor entry [0x0000000026bce000]
    java.lang.Thread.State: BLOCKED (on object monitor)
        at com.sun.tools.visualvm.core.model.ModelFactory.getModel(ModelFactory.jav
a:76)
            - waiting to lock <0x00000000f095bcf8> (a com.sun.tools.visualvm.jvmstat.ap
plication.JvmstatApplication)
        at com.sun.tools.visualvm.application.jvm.JvmFactory.getJVMFor(JvmFactory.j
ava:45)
        at com.sun.tools.visualvm.application.options.Open.openApplication(Open.jav
a:108)
        at com.sun.tools.visualvm.application.options.Open.process(Open.java:93)
        at org.netbeans.spi.sendopts.Option$1.process(Option.java:348)
        at org.netbeans.api.sendopts.CommandLine.process(CommandLine.java:278)
        at org.netbeans.modules.sendopts.HandlerImpl.execute(HandlerImpl.java:23)
        at org.netbeans.modules.sendopts.Handler.cli(Handler.java:30)
        at org.netbeans.CLIHandler.notifyHandlers(CLIHandler.java:195)
        at org.netbeans.core.startup.CLICoreBridge.cli(CLICoreBridge.java:43)
        at org.netbeans.CLIHandler.notifyHandlers(CLIHandler.java:195)
        at org.netbeans.CLIHandler$Server$1ComputingAndNotifying.run(CLIHandler.jav
a:1176)
```

2.6 jcmd pid VM.system_properties，查看 JVM 系统参数

```
E:\itstack\git\github.com\interview>jcmd 111552 VM.system_properties
111552:
```

```
#Sun Jan 13 23:33:19 CST 2021
java.vendor=Oracle Corporation
netbeans.user=C:\\\\Users\\\\xiaofuge1\\\\AppData\\\\Roaming\\\\VisualVM\\\\8u131
sun.java.launcher=SUN_STANDARD
sun.management.compiler=HotSpot 64-Bit Tiered Compilers
netbeans.autoupdate.version=1.23
os.name=Windows 10
```

2.7 jcmd pid GC.heap_dump 路径，导出 heap dump 文件

```
E:\\itstack\\git\\github.com\\interview>jcmd 111552 GC.heap_dump C:\\Users\\xiaofuge1\\Des
ktop\\_dump_0110
111552:
Heap dump file created
```

- 导出的文件需要配合 jvisualvm 查看

2.8 jcmd pid help，列出可执行操作

```
E:\\itstack\\git\\github.com\\interview>jcmd 111552 help
111552:
The following commands are available:
JFR.stop
JFR.start
JFR.dump
JFR.check
```

2.9 jcmd pid help JFR.stop，查看命令使用

```
E:\\itstack\\git\\github.com\\interview>jcmd 111552 help JFR.stop
111552:
JFR.stop
Stops a JFR recording
```

Impact: Low

Permission: java.lang.management.ManagementPermission(monitor)

Syntax : JFR.stop [options]

```
Options: (options must be specified using the <key> or <key>=<value> syntax)
    name : [optional] Recording name, e.g. "My Recording" (STRING, no default
value)
    recording : [optional] Recording number, see JFR.check for a list of available recordings (JLONG, -1)
    discard : [optional] Skip writing data to previously specified file (if any) (BOOLEAN, false)
    filename : [optional] Copy recording data to file, e.g. "C:\Users\user\My Recording.jfr" (STRING, no default value)
    compress : [optional] GZip-compress "filename" destination (BOOLEAN, false)
```

3. jinfo Java 配置信息工具

jinfo (Configuration Info for Java)，实时查看和调整 JVM 的各项参数。在上面讲到 `jps -v` 指令时，可以看到它把虚拟机启动时显式的参数列表都打印出来了，但如果想更加清晰的看具体的一个参数或者想知道未被显式指定的参数时，就可以通过 `jinfo -flag` 来查询了。

命令格式

```
jinfo [ option ] pid
```

使用方式

```
E:\itstack\git\github.com\interview>jinfo -flag MetaspaceSize 111552
-XX:MetaspaceSize=21807104
```

```
E:\itstack\git\github.com\interview>jinfo -flag MaxMetaspaceSize 111552
-XX:MaxMetaspaceSize=18446744073709486080
```

```
E:\itstack\git\github.com\interview>jinfo -flag HeapDumpPath 111552
-
XX:HeapDumpPath=C:\Users\xiaofuge\AppData\Roaming\VisualVM\8u131\var\log\heapdump.h
prof
```

- 各种 JVM 参数你都可以去查询，这样更加方便的只把你想要的显示出来。

4. jstat 收集虚拟机运行数据

jstat (JVM Statistics Monitoring Tool)，用于监视虚拟机各种运行状态信息。它可以查看本地或者远程虚拟机进程中，类加载、内存、垃圾收集、即时编

译等运行时数据。

命令格式

```
jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]
```

- vmid: 如果是查看远程机器, 需要按照此格式:
`[protocol://]lvmid[@hostname[:port]/servername]`
- interval 和 count, 表示查询间隔和次数, 比如每隔 1000 毫秒查询一次进程 ID 的 gc 收集情况, 每次查询 5 次。`jstat -gc 111552 1000 5`

选项列表

选项	描述
-class	监视类加载、卸载数量、总空间以及类装载所耗费时长
-gc	监视 Java 堆情况, 包括 Eden 区、2 个 Survivor 区、老年代、永久代或者 jdk1.8 元空间等, 容量、已用空间、垃圾收集时间合计等信息
-gccapacity	监视内容与-gc 基本一致, 但输出主要关注 Java 堆各个区域使用到的最大、最小空间
-gcutil	监视内容与-gc 基本相同, 但输出主要关注已使用空间占总空间的百分比
-gccause	与 -gcutil 功能一样, 但是会额外输出导致上一次垃圾收集产生的原因
-gcnew	监视新生代垃圾收集情况
-gcnewcapacity	监视内容与 -gcnew 基本相同, 输出主要关注使用到的最大、最小空间
-gcold	监视老年代垃圾收集情况
-gcoldcapacity	监视内容与 -gcold 基本相同, 输出主要关注使用到的最大、最小空间
-compiler	输出即时编译器编译过的方法、耗时等信息
-printcompilation	输出已经被即时编译的方法

选项	描述
-gcpermcapacity	jdk1.7 及以下, 永久代空间统计
-gcmetacapacity	jdk1.8, 元空间统计

- jstat 的监视选项还是非常多的, 但最常用的主要有上面这些。

4.01 jstat -class, 类加载统计

```
E:\itstack\git\github.com\interview>jstat -class 111552
Loaded  Bytes  Unloaded  Bytes      Time
5835 12059.6       37     53.5      3.88
```

- Loaded, 加载 class 的数量
- Bytes: 所占用空间大小
- Unloaded: 未加载数量
- Bytes: 未加载占用空间
- Time: 时间

4.02 jstat -compiler, 编译统计

```
E:\itstack\git\github.com\interview>jstat -compiler 111552
Compiled Failed Invalid  Time  FailedType FailedMethod
3642      0      0    5.61      0
```

- Compiled: 编译数量
- Failed: 失败数量
- Invalid: 不可用数量
- Time: 时间
- FailedType: 失败类型
- FailedMethod: 失败方法

4.03 jstat -gc, 垃圾回收统计

```
E:\itstack\git\github.com\interview>jstat -gc 111552
S0C   S1C   S0U   S1U      EC      EU      OC          OU        MC        MU
CCSC   CCSU   YGC    YGCT     FGC     FGCT      GCT
1024.0 512.0   0.0    0.0  77312.0    35.1  39424.0  13622.9  37120.0 34423.3
5376.0 4579.4    60    0.649   52    3.130    3.779
```

- SOC、S1C, 第一个和第二个幸存区大小
- SOU、S1U, 第一个和第二个幸存区使用大小
- EC、EU, 伊甸园的大小和使用
- OC、OU, 老年代的大小和使用
- MC、MU, 方法区的大小和使用
- CCSC、CCSU, 压缩类空间大小和使用
- YGC、YGCT, 年轻代垃圾回收次数和耗时
- FGC、FGCT, 老年代垃圾回收次数和耗时
- GCT, 垃圾回收总耗时

4.04 jstat -gccapacity, 堆内存统计

```
E:\itstack\git\github.com\interview>jstat -gccapacity 111552
NGCMN      NGCMX      NGC      SOC      S1C      EC      OGCMN      OGCMX      OGC
          OC          MCMN      MCMX      MC      CCSMN      CCSMX      CCSC      YGC      FGC
8192.0    87040.0   80384.0  1024.0   512.0    77312.0   16384.0   175104.0   39424.0
39424.0      0.0  1081344.0  37120.0      0.0  1048576.0   5376.0      60      52
```

- NGCMN、NGCMX, 新生代最小和最大容量
- NGC, 当前新生代容量
- SOC、S1C, 第一和第二幸存区大小
- EC, 伊甸园区的大小
- OGCMN、OGCMX, 老年代最小和最大容量
- OGC、OC, 当前老年代大小
- MCMN、MCMX, 元数据空间最小和最大容量
- MC, 当前元空间大小
- CCSMN、CCSMX, 压缩类最小和最大空间
- YGC, 年轻代 GC 次数
- FGC, 老年代 GC 次数

4.05 jstat -gcnewcapacity, 新生代内存统计

```
E:\itstack\git\github.com\interview>jstat -gcnewcapacity 111552
NGCMN      NGCMX      NGC      SOC MX      S1C      EC MX
          EC          YGC      FGC
8192.0    87040.0   80384.0  28672.0  1024.0  28672.0   512.0  86016.0
77312.0      60      52
```

- NGCMN、NGCMX, 新生代最小和最大容量

- NGC, 当前新生代容量
- SOCMX, 最大幸存 0 区大小
- SOC, 当前幸存 0 区大小
- S1CMX, 最大幸存 1 区大小
- S1C, 当前幸存 1 区大小
- ECMX, 最大伊甸园区大小
- EC, 当前伊甸园区大小
- YGC, 年轻代垃圾回收次数
- FGC, 老年代回收次数

4.06 jstat -gcnew, 新生代垃圾回收统计

```
E:\itstack\git\github.com\interview>jstat -gcnew 111552
SOC      S1C      SOU      S1U      TT MTT    DSS       EC        EU        YGC        YGCT
1024.0   512.0    0.0     0.0     3 15  512.0  77312.0    70.2      60      0.649
```

- SOC、S1C, 第一和第二幸存区大小
- SOU、S1U, 第一和第二幸存区使用
- TT, 对象在新生代存活的次数
- MTT, 对象在新生代存活的最大次数
- DSS: 期望的幸存区大小
- EC, 伊甸园区的大小
- EU, 伊甸园区的使用
- YGC, 年轻代垃圾回收次数
- YGCT, 年轻代垃圾回收消耗时间

4.07 jstat -gcold, 老年代垃圾回收统计

```
E:\itstack\git\github.com\interview>jstat -gcold 111552
MC        MU        CCSC      CCSU      OC          OU        YGC        FGC        FGCT
GCT
37120.0   34423.3   5376.0   4579.4    39424.0    13622.9    60      52      3.130
3.779
```

- MC、MU, 方法区的大小和使用
- CCSC、CCSU, 压缩类空间大小和使用
- OC、OU, 老年代大小和使用
- YGC, 年轻代垃圾回收次数
- FGC, 老年代垃圾回收次数

- FGCT, 老年代垃圾回收耗时
- GCT, 垃圾回收总耗时

4.08 jstat -gcoldcapacity, 老年代内存统计

```
E:\itstack\git\github.com\interview>jstat -gcoldcapacity 111552
    OGCMN      OGCMX      OGC      OC      YGC      FGC      FGCT      GCT
    16384.0    175104.0   39424.0   39424.0   60       52      3.130     3.779
```

- OGCMN、OGCMX, 老年代最小和最大容量
- OGC, 当前老年代大小
- OC, 老年代大小
- YGC, 年轻代垃圾回收次数
- FGC, 老年代垃圾回收次数
- FGCT, 老年代垃圾回收耗时
- GCT, 垃圾回收消耗总耗时

4.09 jstat -gcmetacapacity, 元空间统计

```
E:\itstack\git\github.com\interview>jstat -gcmetacapacity 111552
    MCMN      MCMX      MC      CCSMN      CCSMX      CCSC      YGC      FGC      FGC
T      GCT
    0.0  1081344.0   37120.0      0.0  1048576.0   5376.0     60       52      3.
130     3.779
```

- MCMN、MCMX, 元空间最小和最大容量
- MC, 当前元数据空间大小
- CCSMN、CCSMX, 压缩类最小和最大空间
- CCSC, 压缩类空间大小
- YGC, 年轻代垃圾回收次数
- FGC, 老年代垃圾回收次数
- FGCT, 老年代垃圾回收耗时
- GCT, 垃圾回收消耗总耗时

4.10 jstat -gcutil, 垃圾回收统计

```
E:\itstack\git\github.com\interview>jstat -gcutil 111552
    S0      S1      E      O      M      CCS      YGC      YGCT      FGC      FGCT      GCT
    0.00    0.00   0.09  34.55  92.74  85.18     60      0.649     52      3.130     3.779
```

- S0、S1、幸存 1 区和 2 区，当前使用占比
- E，伊甸园区使用占比
- O，老年代区使用占比
- M，元数据区使用占比
- CCS，压缩类使用占比
- YGC，年轻代垃圾回收次数
- FGC，老年代垃圾回收次数
- FGCT，老年代垃圾回收耗时
- GCT，垃圾回收消耗总耗时

4.11 jstat -printcompilation，JVM 编译方法统计

```
E:\itstack\git\github.com\interview>jstat -printcompilation 111552
Compiled  Size  Type Method
 3642      9      1 java/io/BufferedWriter min
```

- Compiled：最近编译方法的数量
- Size：最近编译方法的字节码数量
- Type：最近编译方法的编译类型
- Method：方法名标识

5. jmap 内存映射工具

jmap (Memory Map for Java)，用于生成堆转储快照 (heapdump 文件)。
jmap 的作用除了获取堆转储快照，还可以查询 finalize 执行队列、Java 堆和方法区的详细信息。

命令格式

```
jmap [ option ] pid
```

- option：选项参数
- pid：需要打印配置信息的进程 ID
- executable：产生核心 dump 的 Java 可执行文件
- core：需要打印配置信息的核心文件
- server-id：可选的唯一 id，如果相同的远程主机上运行了多台调试服务器，用此选项参数标识服务器
- remote server IP or hostname：远程调试服务器的 IP 地址或主机名

选项列表

选项	描述
-dump	生成 Java 堆转储快照。
-finalizerinfo	显示在 F-Queue 中等待 Finalizer 线程执行 finalize 方法的对象。Linux 平台
-heap	显示 Java 堆详细信息，比如：用了哪种回收器、参数配置、分代情况。Linux 平台
-histo	显示堆中对象统计信息，包括类、实例数量、合计容量
-permstat	显示永久代内存状态，jdk1.7，永久代
-F	当虚拟机进程对 -dump 选项没有响应式，可以强制生成快照。Linux 平台

5.1 jmap，打印共享对象映射

```
E:\itstack\git\github.com\interview>jmap 111552
Attaching to process ID 111552, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.161-b12
0x000000005b4a0000      1632K    C:\Program Files\Java\jdk1.8.0_161\jre\bin\awt.dll
0x000000005b8c0000      264K     C:\Program Files\Java\jdk1.8.0_161\jre\bin\t2k.dll
0x000000005b910000      284K     C:\Program Files\Java\jdk1.8.0_161\jre\bin\fontmanager.dll
0x000000005b960000      224K     C:\Program Files\Java\jdk1.8.0_161\jre\bin\splashscreen.dll
0x000000005b9a0000      68K      C:\Program Files\Java\jdk1.8.0_161\jre\bin\nio.dll
```

5.2 jmap -heap，堆详细信息

```
E:\itstack\git\github.com\interview>jmap -heap 111552
Attaching to process ID 111552, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.161-b12
```

```
using thread-local object allocation.
```

```
Parallel GC with 8 thread(s)
```

Heap Configuration:

```
MinHeapFreeRatio      = 0
MaxHeapFreeRatio     = 100
MaxHeapSize          = 268435456 (256.0MB)
NewSize              = 8388608 (8.0MB)
MaxNewSize           = 89128960 (85.0MB)
OldSize              = 16777216 (16.0MB)
NewRatio             = 2
SurvivorRatio        = 8
MetaspaceSize        = 21807104 (20.796875MB)
CompressedClassSpaceSize = 1073741824 (1024.0MB)
MaxMetaspaceSize     = 17592186044415 MB
G1HeapRegionSize     = 0 (0.0MB)
```

5.3 jmap -clstats, 打印加载类

```
E:\itstack\git\github.com\interview> jmap -clstats 111552
Attaching to process ID 111552, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.161-b12
finding class loader instances ..done.
computing per loader stat ..done.
please wait.. computing liveness.....
.....liveness analysis may be inaccurate ...
class_loader    classes   bytes   parent_loader   alive?   type
<bootstrap>    3779    6880779   null            live   <internal>
0x00000000f03853b8      57     132574  0x00000000f031aac8      live   org/netbean
s/StandardModule$OneModuleClassLoader@0x00000001001684f0
0x00000000f01b9b98      0      0      0x00000000f031aac8      live   org/netbean
s/StandardModule$OneModuleClassLoader@0x00000001001684f0
0x00000000f005b280      0      0      0x00000000f031aac8      live   java/util/R
esourceBundle$RBClassLoader@0x00000001000c6ae0
0x00000000f01dfa98      0      0      0x00000000f031aac8      live   org/netbean
```

```
s/StandardModule$OneModuleClassLoader@0x00000001001684f0
0x00000000f01ec518      79      252894  0x00000000f031aac8      live      org/netbean
s/StandardModule$OneModuleClassLoader@0x00000001001684f0
```

5.4 jmap -dump, 堆转储文件

```
E:\itstack\git\github.com\interview>jmap -
dump:live,format=b,file=C:/Users/xiaofuge/Desktop/heap.bin 111552
Dumping heap to C:\Users\xiaofuge\Desktop\heap.bin ...
Heap dump file created
```

6. jhat 堆转储快照分析工具

jhat (JVM Heap Analysis Tool)，与 jmap 配合使用，用于分析 jmap 生成的堆转储快照。

jhat 内置了一个小型的 http/web 服务器，可以把堆转储快照分析的结果，展示在浏览器中查看。不过用途不大，基本大家都会使用其他第三方工具。

命令格式

```
jhat [-stack <bool>] [-refs <bool>] [-port <port>] [-baseline <file>] [-debug <int>] [-version] [-h|-help] <file>
```

命令使用

```
E:\itstack\git\github.com\interview>jhat -
port 8090 C:/Users/xiaofuge1/Desktop/heap.bin
Reading from C:/Users/xiaofuge1/Desktop/heap.bin...
Dump file created Wed Jan 13 16:53:47 CST 2021
Snapshot read, resolving...
Resolving 246455 objects...
Chasing references, expect 49 dots.....
Eliminating duplicate references.....
Snapshot resolved.
Started HTTP server on port 8090
Server is ready.
```

<http://localhost:8090/>

Package <Arrays>

```
class [Lcom.sun.tools.visualvm.core.ui.DataSourceWindow; [0xf0ac1150]
class [Lcom.sun.tools.visualvm.tools.jmx.JmxModel$ConnectionState; [0xf09f6518]
class [Lorg.netbeans.JarClassLoader$Source; [0xf003d1f8]
class [Lorg.netbeans.Module$PackageExport; [0xf0190038]
class [Lorg.netbeans.api.autoupdate.UpdateManager$TYPE; [0xf0ac0b20]
class [Lorg.netbeans.api.autoupdate.UpdateUnitProvider$CATEGORY; [0xf0acd298]
class [Lorg.netbeans.api.progress.ProgressHandle; [0xf09b4e88]
class [Lorg.netbeans.core.windows.ModeStructureSnapshot$ElementSnapshot; [0xf05852d8]
class [Lorg.netbeans.core.windows.ModeStructureSnapshot$ModeSnapshot; [0xf0585270]
class [Lorg.netbeans.core.windows.ModeStructureSnapshot$SlidingModeSnapshot; [0xf0581fb0]
class [Lorg.netbeans.core.windows.SplitConstraint; [0xf044cb70]
class [Lorg.netbeans.core.windows.ViewRequest; [0xf09f67a0]
class [Lorg.netbeans.core.windows.WindowSystemEventType; [0xf05855b0]
class [Lorg.netbeans.core.windows.persistence.GroupConfig; [0xf05854e0]
class [Lorg.netbeans.core.windows.persistence.ModeConfig; [0xf0585548]
class [Lorg.netbeans.core.windows.persistence.TCGroupConfig; [0xf05853a8]
class [Lorg.netbeans.core.windows.persistence.TCRefConfig; [0xf0585478]
class [Lorg.netbeans.core.windows.persistence.TCRefParser; [0xf0585410]
class [Lorg.netbeans.core.windows.view.ElementAccessor; [0xf0581f48]
class [Lorg.netbeans.core.windows.view.ModeAccessor; [0xf0581ee0]
class [Lorg.netbeans.core.windows.view.SlidingAccessor; [0xf0581e78]
class [Lorg.netbeans.core.windows.view.ViewEvent; [0xf0585340]
class [Lorg.netbeans.core.windows.view.ui.toolbars.ToolbarConstraints$Align; [0xf0585618]
class [Lorg.netbeans.modules.autoupdate.ui.wizards.OperationWizardModel$OperationType; [0xf0ac12a8]
class [Lorg.netbeans.modules.autoupdate.updateprovider.AutoupdateCatalogParser$ELEMENTS; [0xf0ac0ab8]
class [Lorg.netbeans.modules.openide.filesystems.declmime.FileElement; [0xf048f0f0]
class [Lorg.netbeans.modules.sendopts.OptionImpl$Appearance; [0xf0580208]
class [Lorg.netbeans.modules.sendopts.OptionImpl; [0xf0580270]
class [Lorg.netbeans.spi.sendopts.Option; [0xf05802d8]
class [Lorg.netbeans.swing.tabcontrol.TabData; [0xf09f6210]
class [Lorg.netbeans.swing.tabcontrol.customtabs.TabbedType; [0xf09f61a8]
class [Lorg.openide.awt.Toolbar; [0xf0587340]
class [Lorg.openide.awt.ToolbarWithOverflow; [0xf0585a40]
class [Lorg.openide.cookies.InstanceCookie$Of; [0xf048cc40]
```

```
jhat -port 8090
```

7. jstack Java 堆栈跟踪工具

jstack (Stack Trace for Java)，用于生成虚拟机当前时刻的线程快照 (threaddump、javacore)。

线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的目的通常是定位线程出现长时间停顿的原因，如：线程死锁、死循环、请求外部资源耗时较长导致挂起等。

线程出现听顿时通过 jstack 来查看各个线程的调用堆栈，就可以获得没有响应的线程在搞什么鬼。

命令格式

```
jstack [ option ] vmid
```

选项参数

选项	描述
-F	当正常输出的请求不被响应时，强制输出线程堆栈
-l	除了堆栈外，显示关于锁的附加信息
-m	如果调用的是本地方法的话，可以显示 c/c++ 的堆栈

命令使用

```
E:\itstack\git\github.com\interview>jstack 111552  
2021-01-10 23:15:03  
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.161-b12 mixed mode):  
  
"Inactive RequestProcessor thread [Was:StdErr Flush/org.netbeans.core.startup.logging.PrintStreamLogger]" #59 daemon prio=1 os_prio=-  
2 tid=0x000000001983a800 nid=0x688 in Object.wait() [0x0000000017fbf000]  
    java.lang.Thread.State: TIMED_WAITING (on object monitor)  
        at java.lang.Object.wait(Native Method)  
        at org.openide.util.RequestProcessor$Processor.run(RequestProcessor.java:19  
39)  
        - locked <0x00000000fab31d88> (a java.lang.Object)
```

- 在验证使用的过程中，可以尝试写一个死循环的线程，之后通过 jstack 查看线程信息。

三、可视化故障处理工具

1. jconsole，Java 监视与管理控制台

JConsole(Java Monitoring and Management Console)，是一款基于 JMX(Java Management Extensions) 的可视化监视管理工具。

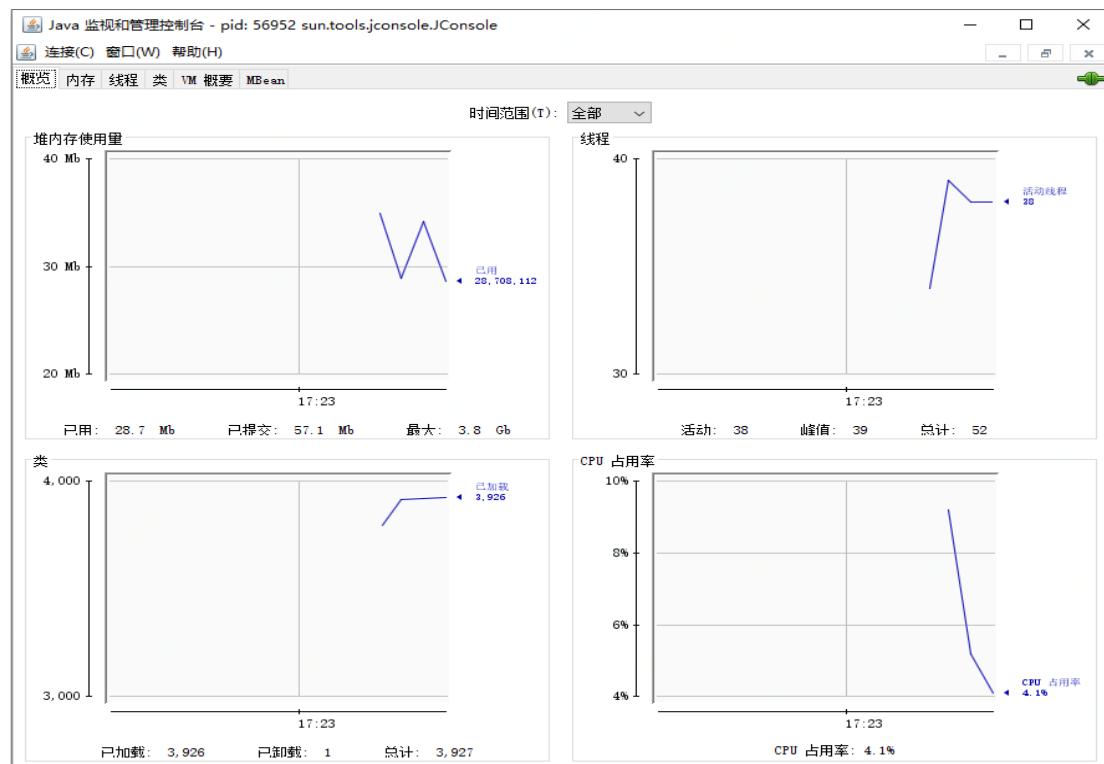
它的功能主要是对系统进行收集和参数调整，不仅可以在虚拟机本身管理还可以开发在软件上，是开放的服务，有相应的代码 API 调用。

JConsole 启动



JConsole 启动

JConsole 使用



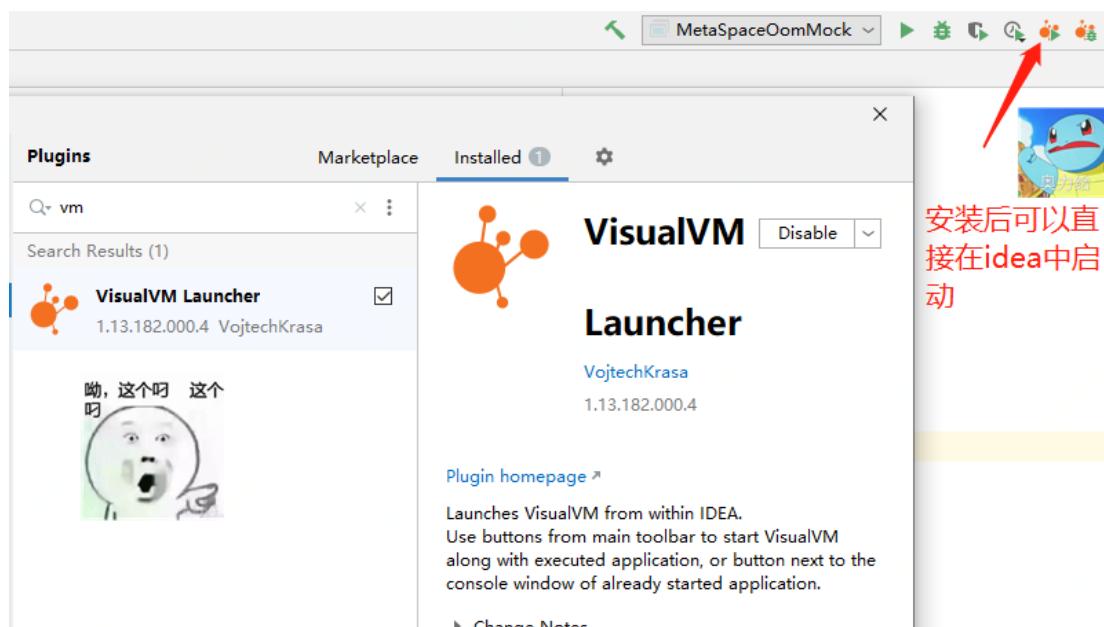
JConsole 使用

2. VisualVM，多合故障处理工具

VisualVM (All-in-One Java Troubleshooting Tool)，是功能最强大的运行监视和故障处理工具之一。

它除了常规的运行监视、故障处理外，还可以做性能分析等工作。因为它的通用性很强，对应用程序影响较小，所以可以直接接入到生产环境中。

VisualVM IDEA 安装



VisualVM IDEA 安装

VisualVM 使用

```
public static void main(String[] args) throws InterruptedException {  
  
    Thread.sleep(5000);  
    ClassLoadingMXBean loadingBean = ManagementFactory.getClassLoadingMXBean();  
    while (true) {  
        Enhancer enhancer = new Enhancer();  
        enhancer.setSuperclass(MetaSpaceOomMock.class);  
        enhancer.setCallbackTypes(new Class[]{Dispatcher.class, MethodInterceptor.class});  
        enhancer.setCallbackFilter(new CallbackFilter() {  
            @Override  
            public int accept(Method method) {  
                return 1;  
            }  
            @Override  
        
```

```

        public boolean equals(Object obj) {
            return super.equals(obj);
        }
    });

    System.out.println(enhancer.createClass().getName() + loadingBean.getTotalLoadedClassCount() + loadingBean.getLoadedClassCount() + loadingBean.getUnloadedClassCount());
}

}

```

记得调整元空间大小

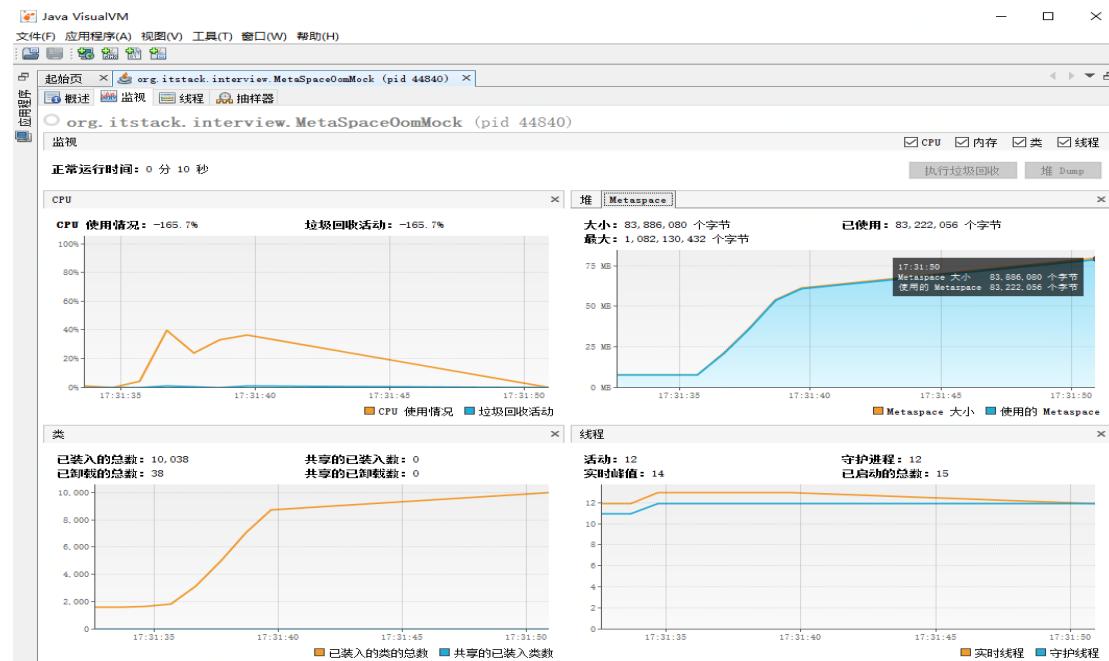
```

-XX:MetaspaceSize=8m
-XX:MaxMetaspaceSize=80m
-Djava.rmi.server.hostname=127.0.0.1
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=7397
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false

```

- 我们就监测这段让元空间溢出的代码, `java.lang.OutOfMemoryError: Metaspace`

监控结果



VisualVM, 监控结果

四、总结

- 本文也是小傅哥在学习《深入理解 Java 虚拟机》过程中的一个总结，这里包括了很多常用的指令，通过这些指令的学习我们也大概会知道 JVM 都给我们提供了什么样的监控信息。
- 其其实实际的业务使用中很少通过指令去监控 JVM 而是有一整套的非入侵全链路监控，在监控服务里与之方法调用时的 JVM 一并监控，可以让研发人员更快速的排查问题。但这些工具的实现依然是需要这些基础，在有了基础的知识掌握后，可以更多地使用工具。
- 编程技术类知识的学习一定要实践验证，否则很容易忘记，也很难掌握。当你经过自己手多敲几遍以后，就会有完全不一样的认识。好了，加油！希望本篇文章能为你的薪资鼓鼓劲！

第 5 节：GC 垃圾回收

提升自身价值有多重要？

经过了风风雨雨，看过了男男女女。时间经过的岁月就没有永恒不变的！
在这趟车上有人下、有人上，外在别人给你点评的标签、留下的烙印，都只是这趟车上的故事。只有个人成长了、积累了、沉淀了，才有机会当自己的司机。

可能某个年龄段的你还看不懂，但如果某天你不那么忙了，要思考思考自己的路、自己的脚步。看看这些是不是你想要的，如果都是你想要的，为什么你看起来不开心？

好！加油，走向你想成为的自己！

一、面试题

谢飞机，小记！，中午吃饱了开始发呆，怎么就学不来这些知识呢，它也不进脑子！

谢飞机：喂，面试官大哥，我想问个问题。

面试官：什么？

谢飞机：就是这知识它不进脑子呀！

面试官：这....

谢飞机：就是看了忘，忘了看的！

面试官：是不是没有实践？只是看了就觉得会了，收藏了就表示懂了？哪哪都不深入！？

谢飞机：好像是！那有什么办法？

面试官：也没有太好的办法，学习本身就是一件枯燥的事情。减少碎片化的时间浪费，多用在系统化的学习上会更好一些。哪怕你写写博客记录下，验证下也是好的。

二、动手验证垃圾回收

说是垃圾回收，我不引用了它就回收了？什么时候回收的？咋回收的？

没有看到实际的例子，往往就很难让理科生接受这类知识。我自己也一样，最好是让我看得见。代码是对数学逻辑的具体实现，没有实现过程只看答案是没有意义的。

测试代码

```

public class ReferenceCountingGC {

    public Object instance = null;
    private static final int _1MB = 1024 * 1024;
    /**
     * 这个成员属性的唯一意义就是占点内存，以便能在 GC 日志中看清楚是否有回收过
     */
    private byte[] bigSize = new byte[2 * _1MB];

    public static void main(String[] args) {
        testGC();
    }

    public static void testGC() {
        ReferenceCountingGC objA = new ReferenceCountingGC();
        ReferenceCountingGC objB = new ReferenceCountingGC();
        objA.instance = objB;
        objB.instance = objA;
        objA = null;
        objB = null;
        // 假设在这行发生 GC，objA 和 objB 是否能被回收？
        System.gc();
    }
}

```

例子来自于《深入理解 Java 虚拟机》中引用计数算法章节。

例子要说明的结果是，相互引用下却已经置为 null 的两个对象，是否会被 GC 回收。如果只是按照引用计数器算法来看，那么这两个对象的计数标识不会为 0，也就不能被回收。但到底有没有被回收呢？

这里我们先采用 jvm 工具指令，jstat 来监控。因为监控的过程需要我手敲代码，比较耗时，所以我们在调用 testGC() 前，睡眠会 Thread.sleep(55000);。启动代码后执行如下指令。

```

E:\itstack\git\github.com\interview>jps -l
10656
88464
38372 org.itstack.interview.ReferenceCountingGC
26552 sun.tools.jps.Jps

```

405 / 417

小傅哥，公众号：bugstack 虫洞栈 | 沉淀、分享、成长，让自己和他人都能有所收获

110056 org.jetbrains.jps.cmdline.Launcher

```
E:\itstack\git\github.com\interview>jstat -gc 38372 2000
```

SOC	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU
CCSC	CCSU	YGC	YGCT	FGC	FGCT	GCT			
10752.0	10752.0	0.0	0.0	65536.0	6561.4	175104.0	0.0	4480.0	770.9
384.0	75.9	0	0.000	0	0.000	0.000			
10752.0	10752.0	0.0	0.0	65536.0	6561.4	175104.0	0.0	4480.0	770.9
384.0	75.9	0	0.000	0	0.000	0.000			
10752.0	10752.0	0.0	0.0	65536.0	6561.4	175104.0	0.0	4480.0	770.9
384.0	75.9	0	0.000	0	0.000	0.000			
10752.0	10752.0	0.0	0.0	65536.0	6561.4	175104.0	0.0	4480.0	770.9
384.0	75.9	0	0.000	0	0.000	0.000			
10752.0	10752.0	0.0	0.0	65536.0	6561.4	175104.0	0.0	4480.0	770.9
384.0	75.9	0	0.000	0	0.000	0.000			
10752.0	10752.0	0.0	0.0	65536.0	6561.4	175104.0	0.0	4480.0	770.9
384.0	75.9	0	0.000	0	0.000	0.000			
10752.0	10752.0	0.0	0.0	65536.0	6561.4	175104.0	0.0	4480.0	770.9
384.0	75.9	0	0.000	0	0.000	0.000			
10752.0	10752.0	0.0	0.0	65536.0	6561.4	175104.0	0.0	4480.0	770.9
384.0	75.9	0	0.000	0	0.000	0.000			
10752.0	10752.0	0.0	0.0	1288.0	65536.0	0.0	175104.0	8.0	4864.0 3982.6
512.0	440.5	1	0.003	1	0.000	0.003			
10752.0	10752.0	0.0	0.0	65536.0	437.3	175104.0	1125.5	4864.0 3982.6	
512.0	440.5	1	0.003	1	0.012	0.015			
10752.0	10752.0	0.0	0.0	65536.0	437.3	175104.0	1125.5	4864.0 3982.6	
512.0	440.5	1	0.003	1	0.012	0.015			

- SOC、S1C, 第一个和第二个幸存区大小
- SOU、S1U, 第一个和第二个幸存区使用大小
- EC、EU, 伊甸园的大小和使用
- OC、OU, 老年代的大小和使用
- MC、MU, 方法区的大小和使用
- CCSC、CCSU, 压缩类空间大小和使用
- YGC、YGCT, 年轻代垃圾回收次数和耗时
- FGC、FGCT, 老年代垃圾回收次数和耗时
- GCT, 垃圾回收总耗时

注意: 观察后面三行, `S1U = 1288.0`、`GCT = 0.003`, 说明已经在执行垃圾回收。接下来, 我们再换种方式测试。在启动的程序中, 加入 GC 打印参数, 观察 GC 变化结果。

-XX:+PrintGCDetails 打印每次 gc 的回收情况 程序运行结束后打印堆空间内存信息(包含内存溢出的情况)
-XX:+PrintHeapAtGC 打印每次 gc 前后的内存情况
-XX:+PrintGCTimeStamps 打印每次 gc 的间隔的时间戳 full gc 为每次对新生代老年代以及整个空间做统一的回收 系统中应该尽量避免
-XX:+TraceClassLoading 打印类加载情况
-XX:+PrintClassHistogram 打印每个类的实例的内存占用情况
-Xloggc:/Users/xiaofuge/Desktop/logs/log.log 配合上面的使用将上面的日志打印到指定文件
-XX: HeapDumpOnOutOfMemoryError 发生内存溢出将堆信息转存起来 以便分析

这回就可以把睡眠去掉了，并添加参数 `-XX:+PrintGCDetails`，如下：

The screenshot shows the 'Configuration' tab of the JVisualVM dialog. The 'Main class:' field contains `org.itstack.interview.ReferenceCountingGC`. The 'VM options:' field contains `-XX:+PrintGCDetails`, which is highlighted with a red box and has a red arrow pointing to it from below. Other fields include 'Program arguments:', 'Working directory:', 'Environment variables:', 'Redirect input from:', 'Use classpath of module:' (set to interview-27), 'JRE:' (Default 1.8.0_161), 'Shorten command line:' (user-local default: none - java [options] classname [args]), and 'Enable capturing from snapshots'. A small cartoon character is visible on the right side of the dialog.

测试结果

```
[GC (System.gc()) [PSYoungGen: 9346K->936K(76288K)] 9346K->944K(251392K), 0.0008518 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (System.gc()) [PSYoungGen: 936K->0K(76288K)] [ParOldGen: 8K->764K(175104K)] 944K->764K(251392K), [Metaspace: 3405K->3405K(1056768K)], 0.0040034 secs] [Times: user=0.08 sys=0.00, real=0.00 secs]
Heap
PSYoungGen      total 76288K, used 1966K [0x0000000076b500000, 0x00000000770a00000,
0x000000007c000000)
eden space 65536K, 3% used [0x0000000076b500000, 0x0000000076b6eb9e0, 0x0000000076f500]
```

```
000)
from space 10752K, 0% used [0x000000076f500000,0x000000076f500000,0x000000076ff80
000)
to   space 10752K, 0% used [0x000000076ff80000,0x000000076ff80000,0x0000000770a00
000)
ParOldGen      total 175104K, used 764K [0x00000006c1e00000, 0x00000006cc900000,
0x000000076b500000)
object space 175104K, 0% used [0x00000006c1e00000,0x00000006c1ebf100,0x00000006cc
900000)
Metaspace       used 3449K, capacity 4496K, committed 4864K, reserved 1056768K
class space    used 376K, capacity 388K, committed 512K, reserved 1048576K
```

- 从运行结果可以看出内存回收日志，Full GC 进行了回收。
- 也可以看出 JVM 并不是依赖引用计数器的方式，判断对象是否存活。否则他们就
不会被回收啦

有了这个例子，我们再接着看看 JVM 垃圾回收的知识框架！

三、JVM 垃圾回收知识框架

垃圾收集（Garbage Collection，简称 GC），最早于 1960 年诞生于麻省理工学院的 Lisp 是第一门开始使用内存动态分配和垃圾收集技术的语言。

垃圾收集器主要做的三件事：哪些内存需要回收、什么时候回收、怎么回收。
而从垃圾收集器的诞生到现在有半个世纪的发展，现在的内存动态分配和内存回收技术已经非常成熟，一切看起来都进入了“自动化”。但在某些时候还是需要我们去监测在高并发的场景下，是否有内存溢出、泄漏、GC 时间过程等问题。所以在了解和知晓垃圾收集的相关知识对于高级程序员的成长就非常重要。

垃圾收集器的核心知识项主要包括：判断对象是否存活、垃圾收集算法、各类垃圾收集器以及垃圾回收过程。如下图；

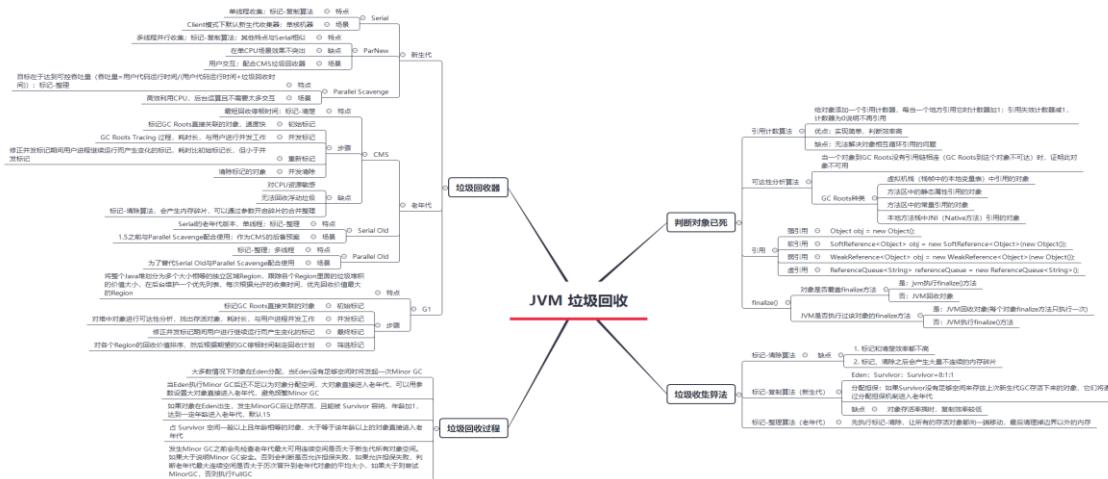


图 27-1 垃圾收集器知识框架

原图下载链接: <http://book.bugstack.cn/#s/6jJp2icA>

1. 判断对象已死

1.1 引用计数器

- 为每一个对象添加一个引用计数器，统计指向该对象的引用次数。
- 当一个对象有相应的引用更新操作时，则对目标对象的引用计数器进行增减。
- 一旦当某个对象的引用计数器为 0 时，则表示此对象已经死亡，可以被垃圾回收。

从实现来看，引用计数器法（Reference Counting）虽然占用了一些额外的内存空间来进行计数，但是它的实现方案简单，判断效率高，是一个不错的算法。也有一些比较出名的引用案例，比如：微软 COM（Component Object Model）技术、使用 ActionScript 3 的 FlashPlayer、Python 语言等。

但是，在主流的 Java 虚拟机中并没有选用引用技术算法来管理内存，主要是因为这个简单的计数方式在处理一些相互依赖、循环引用等就会非常复杂。可能会存在不再使用但又不能回收的内存，造成内存泄漏

1.2 可达性分析法

Java、C# 等主流语言的内存管理子系统，都是通过可达性分析（Reachability Analysis）算法来判定对象是否存活的。

它的算法思路是通过定义一系列称为 GC Roots 根对象作为起始节点集，从这些节点出发，穷举该集合引用到的全部对象填充到该集合中（live set）。这个过程教过标记，只标记那些存活的对象 好，那么现在未被标记的对象就是可以

被回收的对象了。

GC Roots 包括：

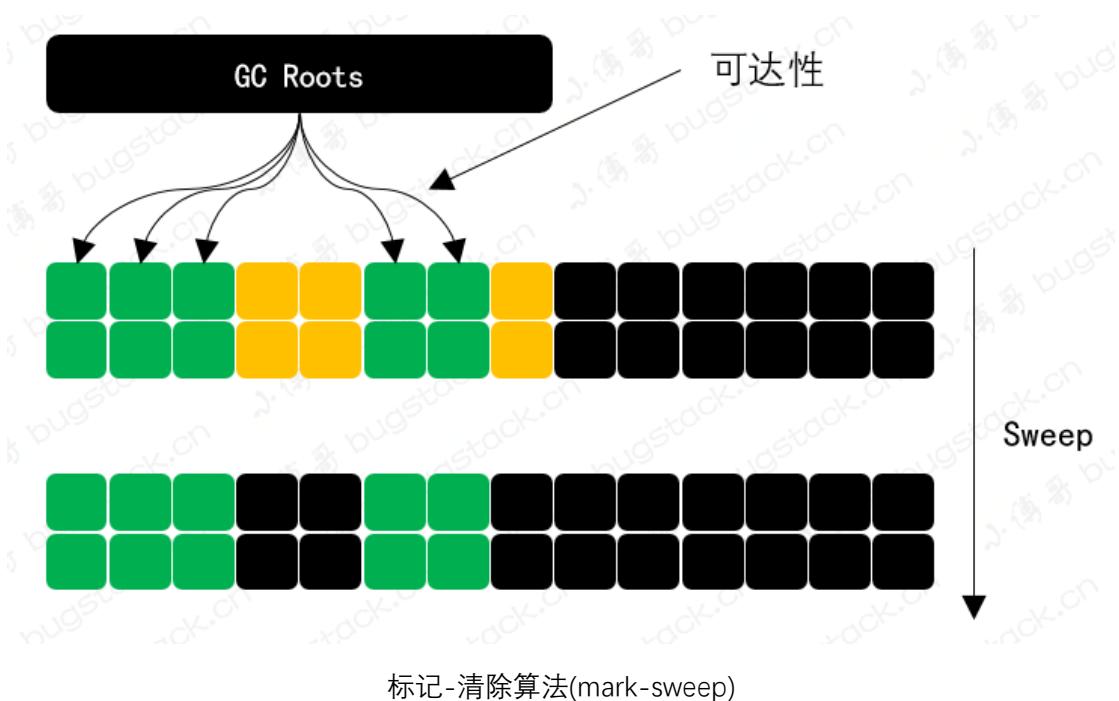
- 全局性引用，对方法区的静态对象、常量对象的引用
- 执行上下文，对 Java 方法栈帧中的局部对象引用、对 JNI handles 对象引用
- 已启动且未停止的 Java 线程

两大问题

- 误报：已死亡对象被标记为存活，垃圾收集不到。多占用一会内存，影响较小。
- 漏报：引用的对象（正在使用的）没有被标记为存活，被垃圾回收了。那么直接导致的就是 JVM 奔溃。（STW 可以确保可达性分析法的准确性，避免漏报）

2. 垃圾回收算法

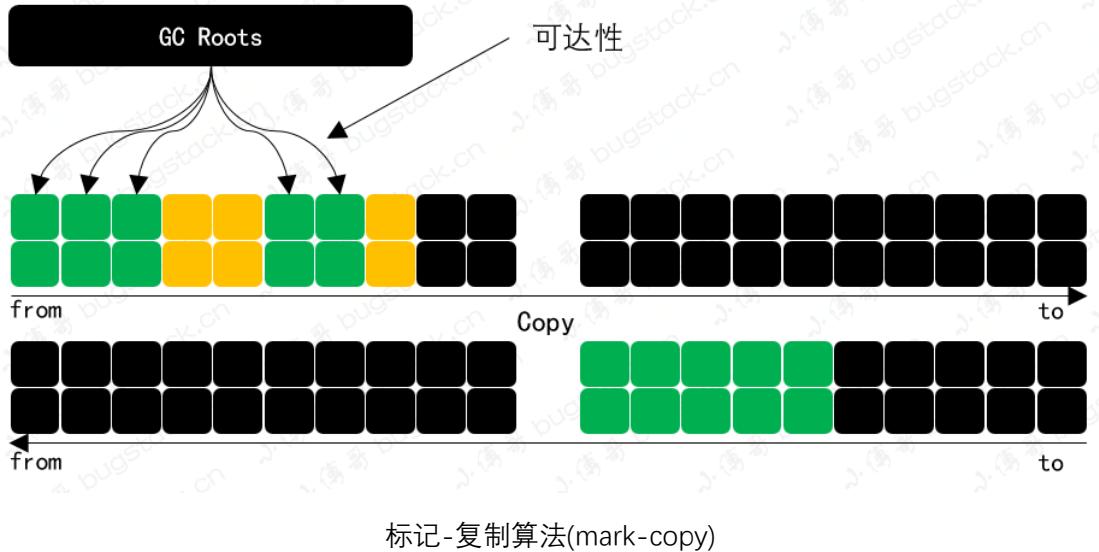
2.1 标记-清除算法(mark-sweep)



- 标记无引用的死亡对象所占据的空闲内存，并记录到空闲列表中 (free list)。
- 当需要创建新对象时，内存管理模块会从 free list 中寻找空闲内存，分配给新建的对象。
- 这种清理方式其实非常简单高效，但是也有一个问题内存碎片化太严重了。

- Java 虚拟机的堆中对象，必须是连续分布的，所以极端的情况下可能即使总剩余内存充足，但寻找连续内存分配效率低，或者严重到无法分配内存。重启汤姆猫！
- 在 CMS 中有此类算法的使用，GC 暂停时间短，但存在算法缺陷。

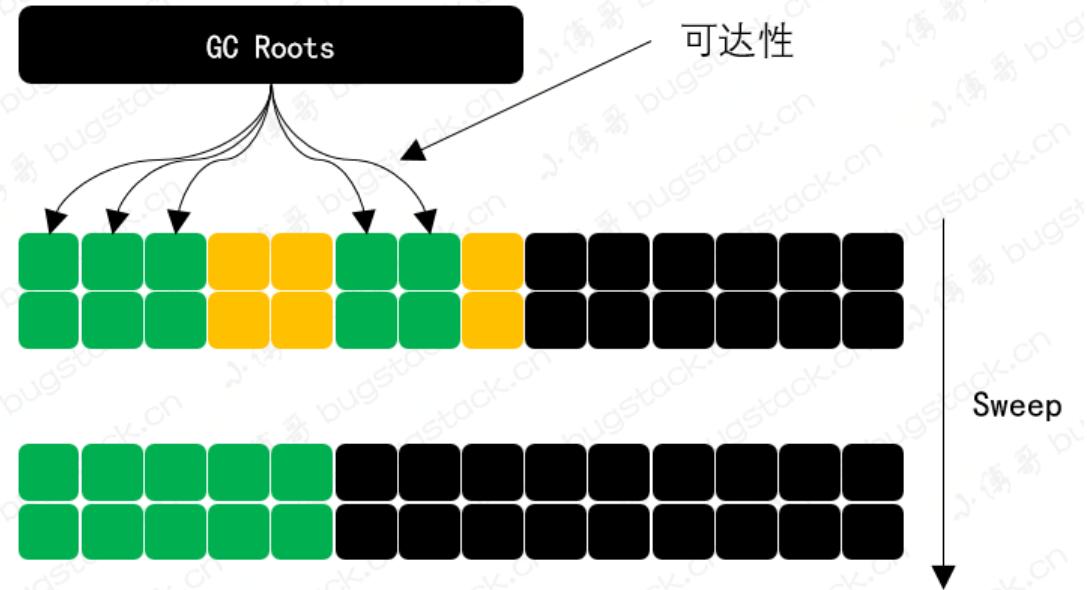
2.2 标记-复制算法(mark-copy)



标记-复制算法(mark-copy)

- 从图上看这回做完垃圾清理后连续的内存空间就大了。
- 这种方式是把内存区域分成两份，分别用两个指针 `from` 和 `to` 维护，并且只使用 `from` 指针指向的内存区域分配内存。
- 当发生垃圾回收时，则把存活对象复制到 `to` 指针指向的内存区域，并交换 `from` 与 `to` 指针。
- 它的好处很明显，就是解决内存碎片化问题。但也带来了其他问题，堆空间浪费了一半。

2.3 标记-压缩算法(mark-compact)



标记-压缩算法(mark-compact)

- 1974 年, Edward Lueders 提出了标记-压缩算法, 标记的过程和标记清除算法一样, 但在后续对象清理步骤中, 先把存活对象都向内存空间一端移动, 然后在清理掉其他内存空间。
- 这种算法能够解决内存碎片化问题, 但压缩算法的性能开销也不小。

3. 垃圾回收器

3.1 新生代

1. Serial
 1. 算法: 标记-复制算法
 2. 说明: 简单高效的单核机器, Client 模式下默认新生代收集器;
2. Parallel ParNew
 1. 算法: 标记-复制算法
 2. 说明: GC 线程并行版本, 在单 CPU 场景效果不突出。常用于 Client 模式下的 JVM
3. Parallel Scavenge
 1. 算法: 标记-复制算法
 2. 说明: 目标在于达到可控吞吐量 (吞吐量=用户代码运行时间/(用户代码运行时间+垃圾回收时间));

3.2 老年代

1. Serial Old
 1. 算法：标记-压缩算法
 2. 说明：性能一般，单线程版本。1.5 之前与 Parallel Scavenge 配合使用；作为 CMS 的后备预案。
2. Parallel Old
 1. 算法：标记-压缩算法
 2. 说明：GC 多线程并行，为了替代 Serial Old 与 Parallel Scavenge 配合使用。
3. CMS
 1. 算法：标记-清除算法
 2. 说明：对 CPU 资源敏感、停顿时间长。标记-清除算法，会产生内存碎片，可以通过参数开启碎片的合并整理。**基本已被 G1 取代**

3.3 G1

1. 算法：标记-压缩算法
2. 说明：适用于多核大内存机器、GC 多线程并行执行，低停顿、高回收效率。

四、总结

- JVM 的关于自动内存管理的知识众多，包括本文还没提到的 HotSpot 实现算法细节的相关知识，包括：安全节点、安全区域、卡表、写屏障等。每一项内容都值得深入学习。
- 如果不仅仅是为了面试背题，最好的方式是实践验证学习。否则这类知识就像 3 分以下的过电影一样，很难记住它的内容。
- 整个的内容也是小傅哥学习整理的一个过程，后续还会不断的继续深挖和分享。感兴趣的小伙伴可以一起讨论学习。

结尾

感谢，你对本书的支持，可能书中会因作者水平有限，有一些描述不准确或者错字内容。欢迎提交给我，也欢迎和我讨论相关的技术内容，作者小傅哥(fustack)，非常愿意与同好进行行流，一起提升技术。

本书到这里还不是结束，接下来还会继续编写，Spring、SpringBoot、Rpc、Mysql以及中间件相关的[面经](#)。同样，面经不只是面经，更是核心技术的学习和深入的了解。

所有的内容的输出都是一个目的，让更多的人对知识能做到，让懂了就是真的懂！本书编写了417页11.5万字，请尊重作者辛苦创作和版权，凡是转载、传播、引流，都需要与原创作者：小傅哥，微信：fustack沟通确认后再使用。

