第零节：小册简介

大家好，我是二哥呀。

来介绍一下《二哥的 JVM 进阶之路》小册吧。小册主要围绕着 Java 中的 JVM 展开，一共 19 个小节，10 万+字，手绘图 100+张，耗费了巨大的心血，以下是小册的个人信息。

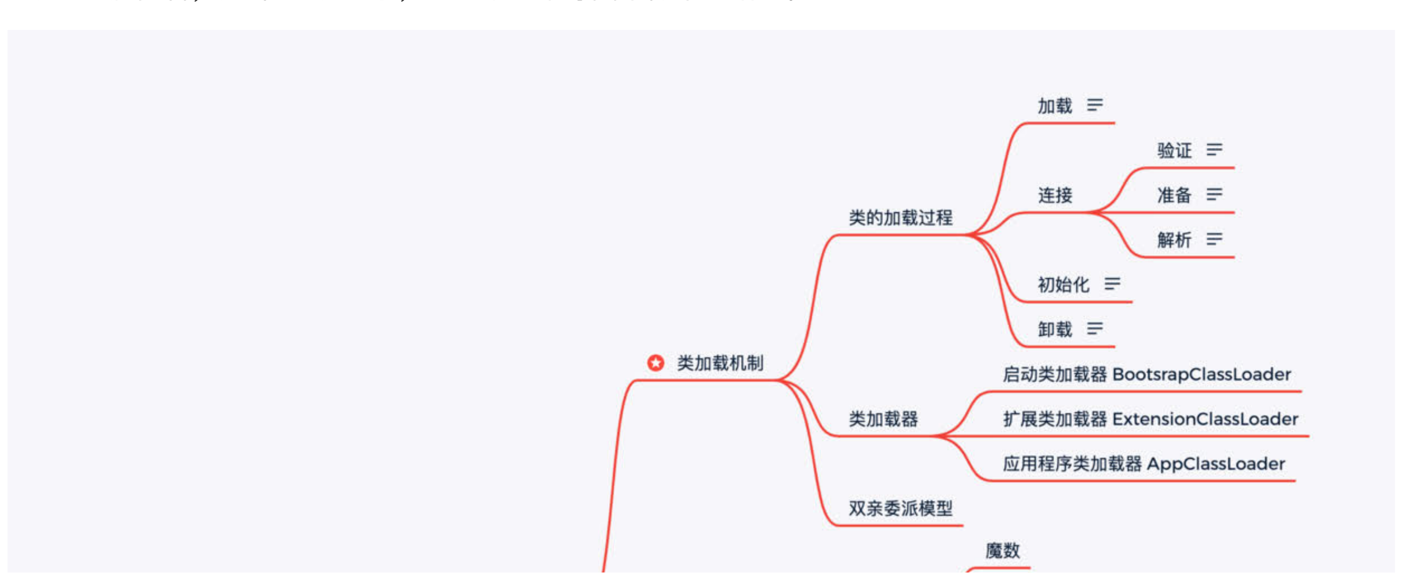
- 小册名字：二哥的JVM进阶之路
- 小册作者：沉默王二
- 小册品质：该小册的内容来源于二哥在 GitHub 上开源的知识库《Java 进阶之路》，能在 GitHub 取得 10000+ star 可以说品质是有目共睹，尤其是国内还有不少小伙伴在访问 GitHub 的时候很不顺利。
- 小册初衷：面试过小伙伴应该感受比较深，JVM 在 Java 岗的面试中占比挺大，尤其是去一些知名公司的话，像类加载机制、垃圾回收机制、Java运行时数据区、JIT 及时编译、字节码指令、性能优化等内容，都是面试中经常被考察的内容。另外，工作一两年后的初级程序员，如果想进阶为高级程序员，JVM的内容也是完全绕不开的，二哥之所以花大力气整理《JVM小册》的原因也在于此，就是希望能帮助大家轻松且深刻地掌握JVM方面的知识。
- 小册简介：主要针对JVM进行讲解，包括JVM是什么、运行时数据区、类加载机制、垃圾回收、性能优化的面试题等，内容涵盖了 Java 虚拟机的方方面面，是一份非常适合 JVM 的学习资料。
- 小册说明：小册算是《二哥的 Java 进阶之路基础篇》、《二哥的并发编程小册》的姐妹篇，可通过 GitHub 阅读或者二哥的网站在线阅读，同时提供 PDF 版本，10 万+字，手绘图 100+张，有亮白版、暗黑版和 epub 三个版本，前后耗费 2 个多月的时间，很不容易，希望你能好好珍惜。

小册包含哪些内容？

《二哥的JVM进阶之路》主要包含下面这些内容：

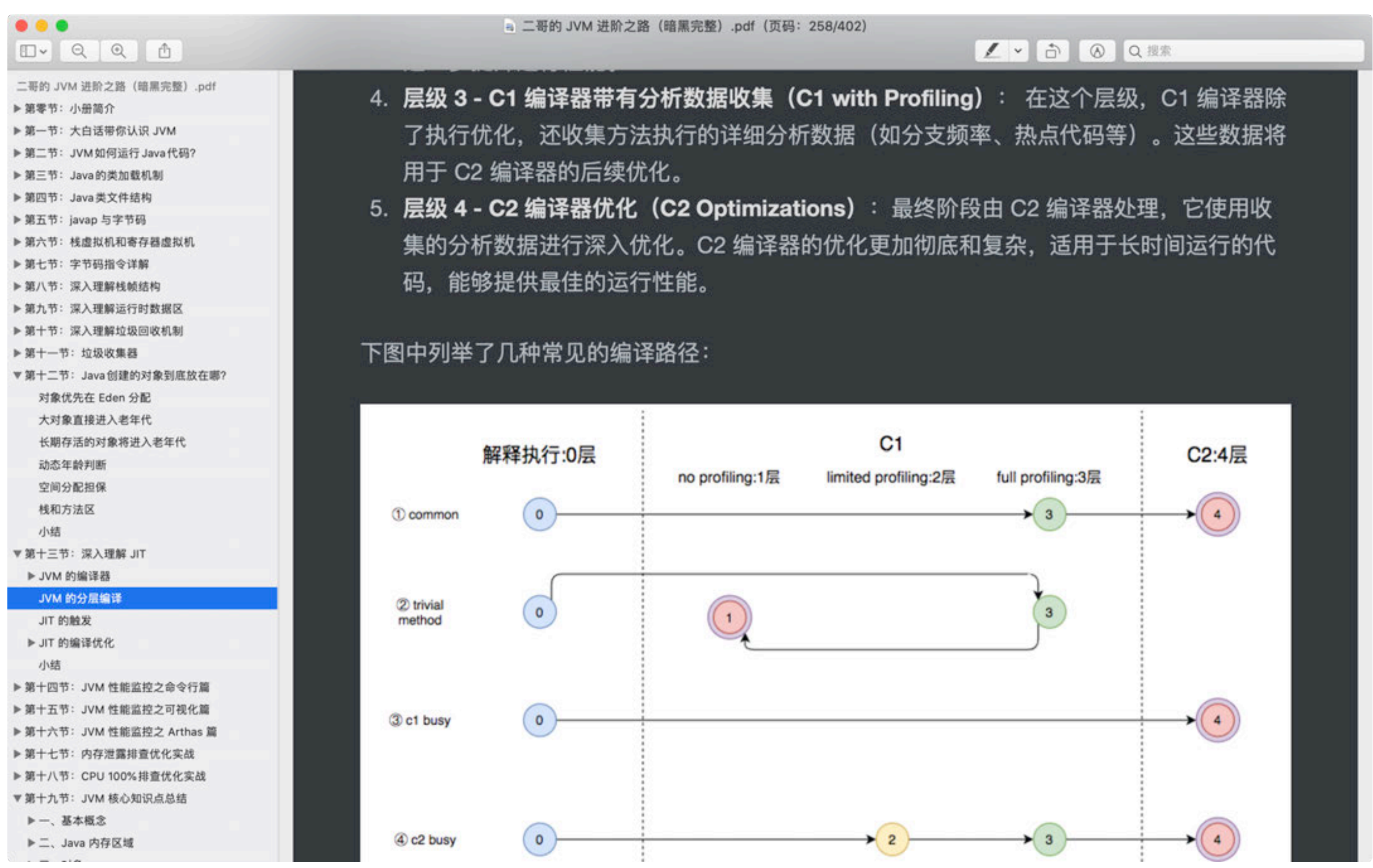
- JVM 是什么？
- Java 运行时数据区
- 类加载机制
- 垃圾回收机制
- JIT 及时编译
- 字节码指令
- 性能优化

一共 19 篇内容，共计 10 万+ 字，用一张思维导图来做个总结吧。

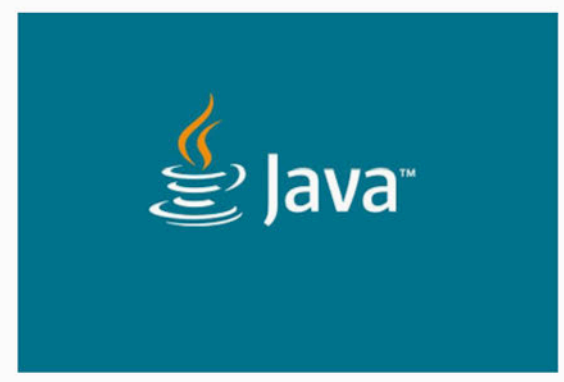


JVM
沉默王二
字节码与类的加载
字节码
类文件结构
class 文件版本
常量池
字面量
符号引用
访问标志
字段表集合
方法表集合
属性表集合
字节码指令
字节码执行引擎
运行时栈帧结构
实战字节码
Hotspot Debugger
ASM
自己编译JDK
内存与垃圾回收
JVM 内存结构
方法区
JDK8 以前永久代
JDK8 以后元空间
堆
新生代
Eden
Survivor
From
To
老年代
程序计数器
虚拟机栈
本地方法栈
垃圾回收算法
标记清除算法
复制算法
标记整理算法
分代手机算法
垃圾回收机制
Minor GC
Full GC
垃圾收集器
Serial
Serial Old
ParNew
Parallel Scavenge
Parallel Old
JDK8 默认的垃圾收集器
CMS
G1
JDK9 开始的默认垃圾收集器
性能监控和调优
调优指标
吞吐量
停顿时间
垃圾回收频率
调优对象
内存区域大小以及策略
垃圾回收器
调优工具
jps
jstat
jinfo
jmap
jstack

这里展示一下暗黑版的 PDF 视图，大家先感受一下，手绘图都画得非常用心。



这是 epub 版本的阅读效果，感觉左右翻动的效果好舒服，一次可以看两页，真的就像在读纸质版书籍一样，体验非常棒。



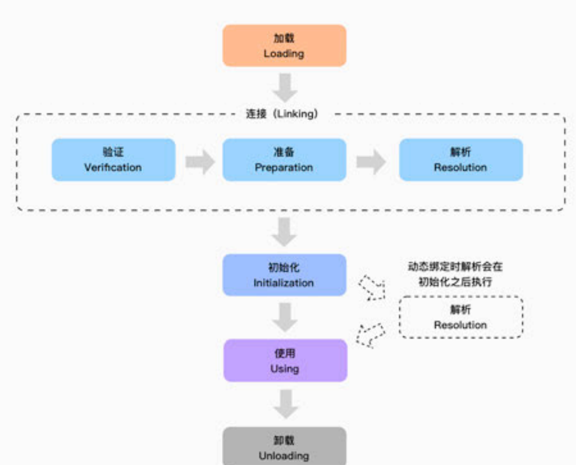
文件格式的定制者可以自由选择魔数值（只要没用过），比如说 .png 文件的魔数是 `8950 4e47`。

至于字节码文件中的其他内容，暂时先不用去管，我们后面会详细讲解。

类加载过程

知道什么是 Java 字节码后，我们来聊聊 Java 的类加载过程。

第 91 页



类从被加载到 JVM 开始，到卸载出内存，整个生命周期分为七个阶段，分别是加载、验证、准备、解析、初始化、使用和卸载。其中验证、准备和解析这三个阶段统称为连接。

除去使用和卸载，就是 Java 的类加载过程。这 5 个阶段一般是顺序发生的，但在动态绑定的情况下，解析阶段发生在初始化阶段之后（我们随后来解释）。

第 92 页　　本章最后一页

如果你喜欢在线阅读，请访问下面这两个网址：

> - 进阶之路：https://javabetter.cn/jvm/
> - 技术派：https://paicoding.com/column/8/1

paicoding.com
技术派 Java编程大3.0

技术派 首页 教程 派聪明 写文章 绑定星球编号

1 大白话带你认识JVM 星球
更新时间: 2024年01月16日
2 JVM是如何运行Java代码的? 星球
更新时间: 2024年01月16日
3 Java的类加载机制 星球
更新时间: 2024年01月16日
4 Java的类文件结构 星球
更新时间: 2024年01月16日
5 从javap的角度轻松看懂字节码 星球
更新时间: 2024年01月16日
6 栈虚拟机与寄存器虚拟机 星球
更新时间: 2024年01月16日
7 字节码指令详解 星球
更新时间: 2024年01月16日
8 深入理解JVM的栈帧结构 星球
更新时间: 2024年01月16日

二哥的 JVM 进阶之路
JDK 8
虚拟机栈
栈帧
本地方法栈
堆
字符串常量池
PC 寄存器
元空间
常量池

我们会在后面的章节单独来讲。

执行引擎

执行引擎（Execution Engine）就好像明朝时期的六部，主要用来干具体的事，“虚拟机”是一个相对于“物理机”的概念，这两种机器都有代码执行能力，其区别是物理机的执行引擎是直接建立在处理器、缓存、指令集和操作系统层面上的，而虚拟机的执行引擎则是由软件自行实现的，因此可以不受物理条件制约地定制指令集与执行引擎的结构关系，能够执行那些不被硬件直接支持的指令集格式。

执行引擎的任务就是将字节码指令解释/编译为对应平台上的本地机器指令才可以。简单来说，JVM 中的执行引擎充当了将高级语言翻译为机器语言的译者。

如果你在阅读过程中感觉这份小册写的还不错，甚至有亿点点收获，可以分享给你的同学或者同事，我的虚荣心也会得到恰当的满足😜

如何获取最新版?

小册分为 3 个版本，暗黑版（适合夜服）、亮白版（适合打印）、epub 版，绝对不虚市面上任何一本 Java 虚拟机的实体书！

小册会持续保持**更新**，如果想获得最新版，请扫下面的优惠券加入二哥的编程星球。

然后通过星球的第一个置顶帖「球友必看」在百度网盘和阿里云盘中下载。

返回上一级 | 全部文件 > 知识星球 > 01-PDF > 00-二哥的 Java 进...

文件名	修改时间	大小
二哥的 Java 进阶之路.epub	2023-04-12 00:53	75.7M
二哥的 Java 进阶之路暗黑版.pdf	2023-04-12 00:52	31.3M
二哥的 Java 进阶之路亮白版.pdf	2023-04-12 00:52	34.5M
二哥的JVM进阶之路.epub	2024-01-16 13:32	61M
二哥的并发编程进阶之路.pdf	2023-09-04 15:49	16.6M

或者你也可以通过星球的第一个置顶帖「球友必看」在语雀中查看，我已经将内容同步上去。

还可以通过知识星球的优质专栏进行查看，内容已同步更新。

PDF 一共 10 万+字，100+ 张手绘图。前后耗费 2 个多月的时间，很不容易，希望你能认真阅读，然后快速提升你在JVM方面的编码能力和面试功底。

面试指南（配套教程）

《Java 面试指南》是二哥编程星球的的一个付费专栏，和《Java 进阶之路》上的内容可以形成很好的互补，截止到目前，已经更新 48 万字，可以说是满满的干货和诚意。

08-27,2023（每日上午更新昨日数据，“--”表示暂无数据）

文档明细　　全部 ⌄　　搜索文档　　导出

文档名称	创建者	创建时间	最近更新	字数	阅读量	点赞量	评论量
如何准备面试？（...	沉默王二	2022-03-02	2022-11-14	8,200	4,973	14	4
学 Spring Boot ...	沉默王二	2022-08-25	2022-08-25	1,986	1,216	8	0
程序员如何做副业...	沉默王二	2022-06-22	2022-06-22	5,140	459	7	1
如何快速学习某项...	沉默王二	2022-03-04	2022-07-29	4,250	589	6	0
如何更高效地学习...	沉默王二	2022-03-04	2022-07-22	2,739	424	6	0
没有项目经验怎么...	沉默王二	2022-03-04	2022-07-22	3,851	1,390	6	0
如何写好简历？（...	沉默王二	2022-06-16	05-12 16:34	5,151	3,136	5	2

一共分为 6 大板块，对面试、职场、技术、学习都会帮助特别大。

- 面试准备篇（25+篇），手把手教你如何准备面试。
- 职场修炼篇（11+篇），手摸手教你如何在职场中如鱼得水。
- 学习路线篇（13+篇），手勾手教你如何快速学习一门技术栈。
- 技术提升篇（33+篇），手拉手教你如何成为团队不可或缺的技术攻坚小能手。
- 面经分享篇（23+篇），手牵手教你如何在面试中知彼知己，百战不殆。
- 场景设计篇（22+篇），手握手教你如何在面试中脱颖而出。

01、面试准备篇

所谓临阵磨枪，不快也光。更何况提前做好充足的准备呢？这 25+篇内容会系统地引导你该如何做好面试准备。涉及到的主题有：简历、源码、LeetCode、项目经验、开源项目、高并发、证书、和 HR 对线、国企名单、公司投递名单、银行、谈薪等等面试常见问题。

如何准备面试？（完结）

分享 编辑

如何准备面试？（完结）

全文 2万多字可以说是全面详细地剖析了面试前必须做的准备

所谓临阵磨枪，不快也光。更何况提前准备好面试呢！

如果提前准备好，那自然面试的成功率也会提高一大截，相应的，我们就会拿到更好的 offer、更高的起薪，这对以后的职业发展非常有利。

“面试造火箭，工作拧螺丝”已经成为常态，也就是说，尽管工作后很有可能之前准备的一些技能点、八股文完全用不到，但仍然要在面试前花心思去准备。

计算机专业经过这些年的发展，知识面已经变得又宽又深了，很多时候，一个应届生，在面试的时候会被问到有没有三高（**高并发**、**高性能**、**高可用**）的经验，离谱，确实离谱（😂）！

但筛选一个合格的候选人的成本确实在变得越来越昂贵，因为涌入这个行业的人实在是太多了。作为求职者，要做的第一步就是：接受它、拥抱它，毕竟打铁还需自身硬，先把自己搞强一点再说。

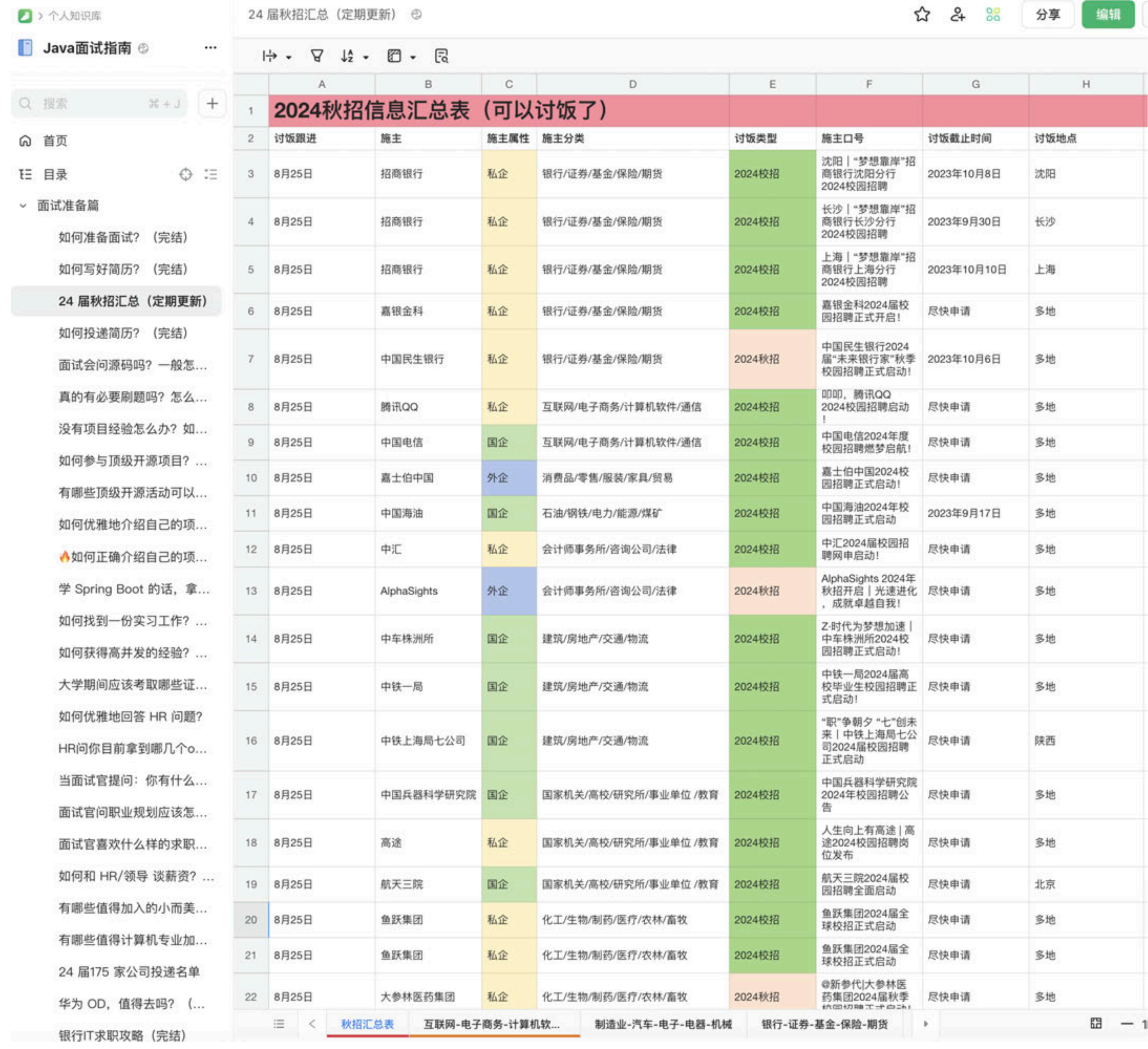
看右侧的小目录就能感受到

- 八股文要不要背？当然要背！
- 项目要不要搞？当然要搞！
- LeetCode 要不要刷？当然要刷！
- 操作系统、计算机组成原理、计算机网络这些枯燥的计算机基础知识到底要不要学？当然要学！

大纲

第一步，尽早确定自己的求职方向
社招
1）扎实的 Java 基本功
2）掌握主流的开发框架
3）熟悉微服务和分布式
4）加分项
校招
1）Java 基础
2）数据结构与算法
3）掌握基本的 Linux 命令
4）掌握操作系统和计算机网络基…
第二步，了解投递简历的黄金时间
1）秋招
2）寒假实习
3）春招
4）暑假实习
5）秋招和春招的对比
6）工作党
第三步、如何获取招聘信息
1、关注目标企业的官网和公众号
2、牛客网
3、超级简历

个人知识库
Java面试指南
搜索
首页
目录
面试准备篇
如何准备面试？（完结）
如何写好简历？（完结）
24 届秋招汇总（定期更新）
如何投递简历？（完结）
面试会问源码吗？一般怎...
真的有必要刷题吗？怎么...
没有项目经验怎么办？如...
如何参与顶级开源项目？...
有哪些顶级开源活动可以...
如何优雅地介绍自己的项...
如何正确介绍自己的项...
学 Spring Boot 的话，拿...
如何找到一份实习工作？...
如何获得高并发的经验？...
大学期间应该考取哪些证...
如何优雅地回答 HR 问题？
HR问你目前拿到哪几个o...
当面试官提问：你有什么...
面试官问职业规划应该怎...
面试官喜欢什么样的求职...
如何和 HR/领导 谈薪资？...
有哪些值得加入的小而美...
有哪些值得计算机专业加...
24 届175 家公司投递名单
华为 OD，值得去吗？（...
银行IT求职攻略（完结）
如何写好简历？（完结）
分享
编辑
社招2-3年模板案例
有球友在微信上问我“有没有社招 2 到 3 年好一点的简历模板啊？”这里就给大家分享一个，我之前的一个读者（工作 2 年多，最近刚面试）的面试情况和他在写简历时的注意事项，那希望能给球友们一些参考和启发。
在整个面试过程中，问的最多的几个问题：
1.Java 本地锁到分布式锁，各种锁的场景，为什么要用，以及不同锁实现方式的底层，优缺点，还有 volitale
2.hashmap，这个就不用多说了，put 过程啊，为什么线程不安全，1.7 和 1.8 的区别，为什么要用红黑树等等，可问的很多
3.多线程实现方式，线程池核心参数，运行过程，有什么问题需要注意的
4.jvm 方面，cms 问的比较多，和 g1 的区别，还有 rootsearching，类加载过程，jvm 内存模型以及各个模块运用
5.redis 哨兵同步，投票选举，集群模式，持久化方式，zset 实现方式
6.dubbo 调用链路，其 spi 和 java 的有什么区别
7.mysql 索引优化思路，事务 mvcc，日志系统，主从同步，buffer pool，分库分表等
8.zookeeper 脑裂问题，leader 选举过程
9.spring bean 生命周期，循环依赖，ioc 和 aop，事务实现方式等
10.kafka 高吞吐原因，丢失消息的场景，副本维护，leader 选举，消息幂等性保证等
对于 2-3 年工作经历的话，整个面试中问的很多的还是对于「基础、还有各个框架的理解」，这也是最基础的内容，还会有一些「设计选型」，如果你是项目负责人的话，需要知道为何要选某个框架，还有一些兜底，都是需要自己去做的
再来看看读者的这个「简历是如何写」的。
专业技能
给大家看下「专业技能」栏，读者在这里写的基本都是他记得很熟的，不熟悉的框架尽量不要写上去，比如 rabbitmq 虽然你可能看过，但不是很熟，就不要乱写，免得面试中被问到回答不上来。
专业技能
1.具备扎实的 Java 基础以及面向对象编程思想，熟悉 Git、Maven 等进行项目版本管理
大纲
核心注意事项
简历模板
简历内容
基本资料
教育背景
专业技能
项目经历
个人总结
再次强调
1）基本错误尽量减少，比如这样：
2）千万不要 Word，就用 PDF
3）HR/面试官关注的点
4）简历模板
社招2-3年模板案例
专业技能
项目经历
面试过程需要注意什么
1.跟着面试官的节奏回答问题
2.让面试官跟着自己的节奏来问
3.避重就轻，快速逃离
4.学会猜
5.别紧张，放轻松
6.先面小公司
问题回复
学历重要吗？
算法重要吗？
谈薪不敢要怎么办？



2024秋招信息汇总表（可以讨饭了）

讨饭跟进	施主	施主属性	施主分类	讨饭类型	施主口号	讨饭截止时间	讨饭地点
8月25日	招商银行	私企	银行/证券/基金/保险/期货	2024校招	沈阳丨"梦想靠岸"招商银行沈阳分行2024校园招聘	2023年10月8日	沈阳
8月25日	招商银行	私企	银行/证券/基金/保险/期货	2024校招	长沙丨"梦想靠岸"招商银行长沙分行2024校园招聘	2023年9月30日	长沙
8月25日	招商银行	私企	银行/证券/基金/保险/期货	2024校招	上海丨"梦想靠岸"招商银行上海分行2024校园招聘	2023年10月10日	上海
8月25日	嘉银金科	私企	银行/证券/基金/保险/期货	2024校招	嘉银金科2024届校园招聘正式开启!	尽快申请	多地
8月25日	中国民生银行	私企	银行/证券/基金/保险/期货	2024秋招	中国民生银行2024届"未来银行家"秋季校园招聘正式启动!	2023年10月6日	多地
8月25日	腾讯QQ	私企	互联网/电子商务/计算机软件/通信	2024校招	叩叩，腾讯QQ 2024校园招聘启动!	尽快申请	多地
8月25日	中国电信	国企	互联网/电子商务/计算机软件/通信	2024校招	中国电信2024年度校园招聘燃梦启航!	尽快申请	多地
8月25日	嘉士伯中国	外企	消费品/零售/服装/家具/贸易	2024校招	嘉士伯中国2024校园招聘正式启动!	尽快申请	多地
8月25日	中国海油	国企	石油/钢铁/电力/能源/煤矿	2024校招	中国海油2024年校园招聘正式启动	2023年9月17日	多地
8月25日	中汇	私企	会计师事务所/咨询公司/法律	2024校招	中汇2024届校园招聘网申启动!	尽快申请	多地
8月25日	AlphaSights	外企	会计师事务所/咨询公司/法律	2024秋招	AlphaSights 2024年秋招开启丨光速进化，成就卓越自我!	尽快申请	多地
8月25日	中车株洲所	国企	建筑/房地产/交通/物流	2024校招	Z·时代为梦想加速丨中车株洲所2024校园招聘正式启动!	尽快申请	多地
8月25日	中铁一局	国企	建筑/房地产/交通/物流	2024校招	中铁一局2024届高校毕业生校园招聘正式启动!	尽快申请	多地
8月25日	中铁上海局七公司	国企	建筑/房地产/交通/物流	2024校招	"职"争朝夕 "七"创未来丨中铁上海局七公司2024届校园招聘正式启动	尽快申请	陕西
8月25日	中国兵器科学研究院	国企	国家机关/高校/研究所/事业单位 /教育	2024校招	中国兵器科学研究院2024年校园招聘公告	尽快申请	多地
8月25日	高途	私企	国家机关/高校/研究所/事业单位 /教育	2024校招	人生向上有高途丨高途2024校园招聘岗位发布	尽快申请	多地
8月25日	航天三院	国企	国家机关/高校/研究所/事业单位 /教育	2024校招	航天三院2024届校园招聘全面启动	尽快申请	北京
8月25日	鱼跃集团	私企	化工/生物/制药/医疗/农林/畜牧	2024校招	鱼跃集团2024届全球校招正式启动	尽快申请	多地
8月25日	鱼跃集团	私企	化工/生物/制药/医疗/农林/畜牧	2024校招	鱼跃集团2024届全球校招正式启动	尽快申请	多地
8月25日	大参林医药集团	私企	化工/生物/制药/医疗/农林/畜牧	2024秋招	@新参代丨大参林医药集团2024届秋季	尽快申请	多地

02、职场修炼篇

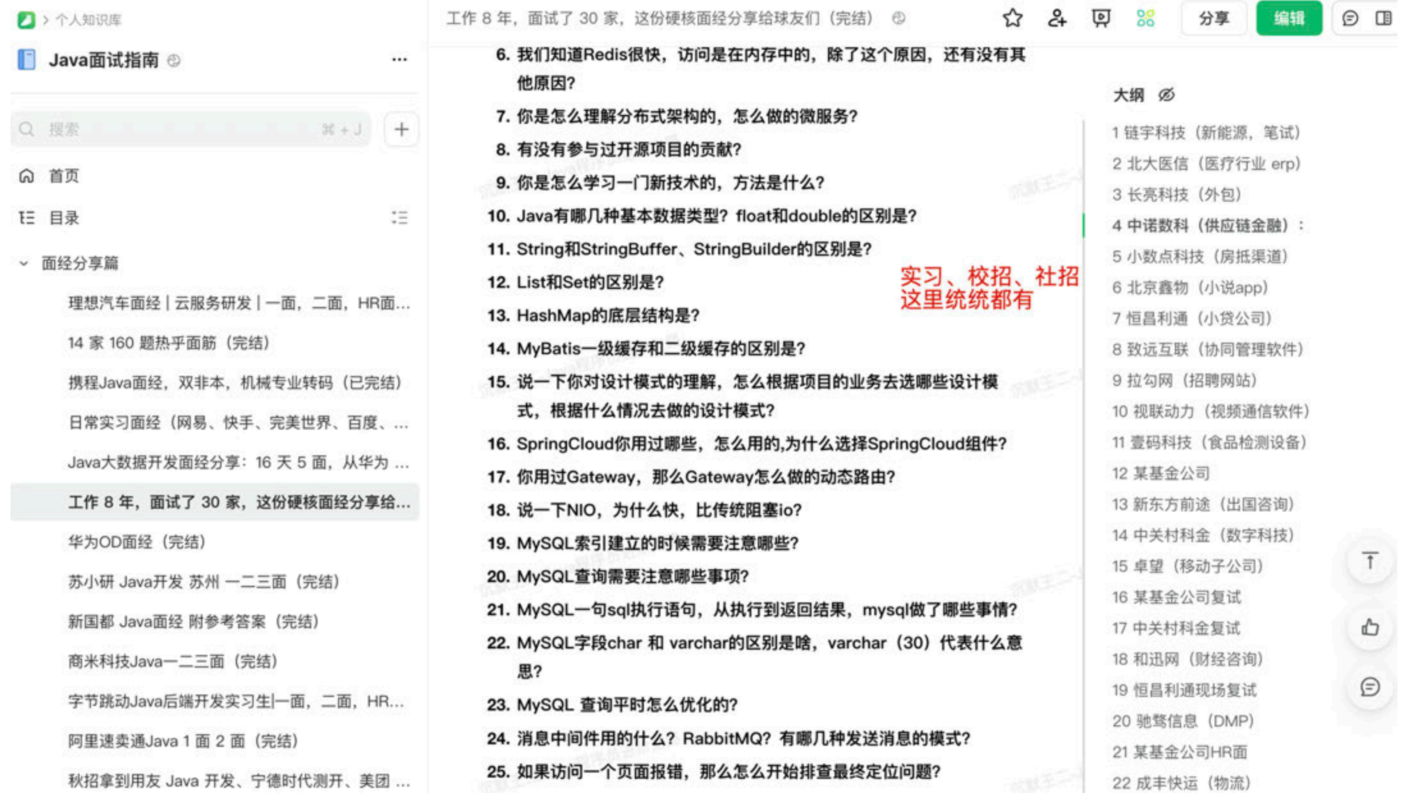
如何平滑度过试用期？如何平滑度过 35 岁程序员危机？如何在繁重的工作中持续成长？如何做副业？如何赚零花钱？如何达到 30 万+年薪等等，都是大家迫切关心的问题，这 11+篇内容会一一为你揭晓答案。

03、技术提升篇

编程能力、技术功底，是我们程序员安身立命之本，是我们求职/工作的最核心的武器。

04、面经分享篇

知彼知己，方能百战不殆，我们必须得站在前辈的肩膀上，才能走得更远更快。他们在面试中遇到过哪些经典的问题，我们能不能提前演练一下，对临场发挥有着至关重要的作用。



05、场景设计题篇

有些面试官不喜欢问八股文，反而更喜欢结合项目问一些非常经典的场景题，这种场景题没有标准的答案，但却很能考察一名求职者的逻辑思维能力。

个人知识库 > Java面试指南

如何保证数据库和缓存双写一致性？（完结）

如何保证数据库和缓存双写一致性？（完结）

数据库和缓存（比如：redis）双写数据一致性问题，是一个跟开发语言无关的公共问题。尤其在高并发的场景下，这个问题变得更加严重。

我很负责的告诉你，该问题无论在面试，还是工作中遇到的概率非常大，所以非常有必要跟大家一起探讨一下。

今天这篇文章我会从浅入深，跟大家一起聊聊，数据库和缓存双写数据一致性问题常见的解决方案，这些方案中可能存在的坑，以及最优方案是什么。

1. 常见方案

通常情况下，我们使用缓存的主要目的是为了提升查询的性能。
大多数情况下，我们是这样使用缓存的：

用户请求 → 查询缓存 → 是否存在？ —否→ 查询数据库 → 是否存在？ —是→ 放入缓存；是否存在？ —是→ …

大纲
1. 常见方案
2. 先写缓存，再写数据库
3. 先写数据库，再写缓存
3.1 写缓存失败了
3.1 高并发下的问题
3.2 浪费系统资源
4. 先删缓存，再写数据库
4.1 高并发下的问题
4.2 缓存双删
5. 先写数据库，再删缓存
6. 删缓存失败怎么办？
7. 定时任务
8. mq
9. binlog

更多优质专栏

除了《Java 面试指南》专栏，二哥编程星球还提供了：《技术派实战教程》、《编程喵实战笔记》、《二哥的 LeetCode 刷题笔记》、《算法突击 50 题》、《华为 OD 笔试 AB 卷题库》等五个额外的专栏。

01、技术派实战教程

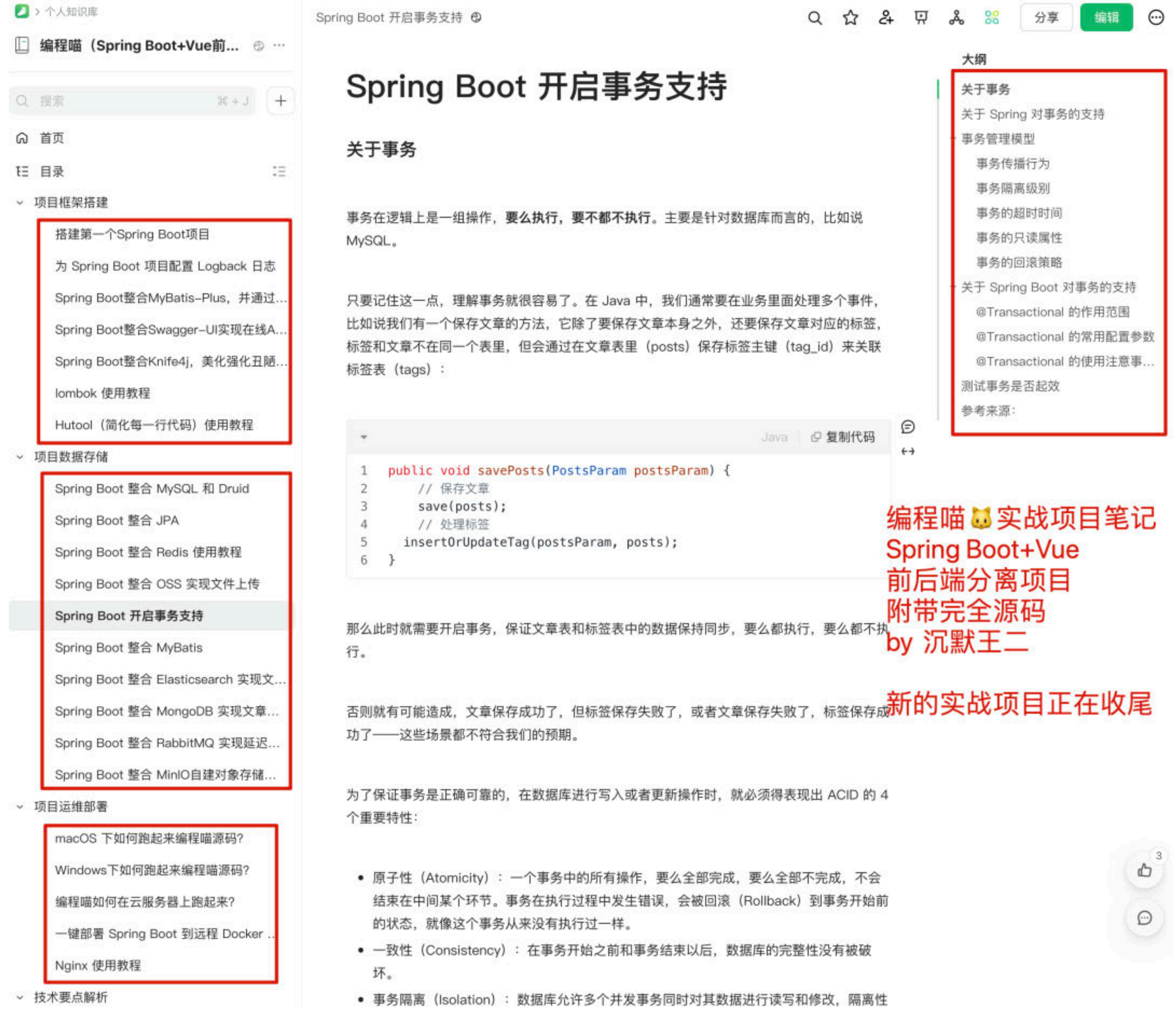
技术派是一个基于 Spring Boot、MyBatis-Plus、MySQL、Redis、ElasticSearch、MongoDB、Docker、RabbitMQ 等技术栈实现的社区系统，采用主流的互联网技术架构、全新的UI设计、支持一键源码部署，拥有完整的文章&教程发布/搜索/评论/统计流程等，代码完全开源，没有任何二次封装，是一个非常适合二次开发/实战的现代化社区项目👍 。

下面是《技术派教程》部分目录（包括大厂篇、基础篇、进阶篇、工程篇、扩展篇、前端篇，目前已完成 100+篇），很多球友都反馈说光这套教程就值 599 元。

- QA 自助排查
  - ✅技术答疑（持续更新，新人必看👉）
  - ✅问题反馈及解决方案（持续更新，新人必看）
- 开篇词
  - ✅技术派系统架构、功能模块一览
  - ✅如何将技术派写入简历
- 大厂篇
  - ✅技术派产品调研
  - ✅技术派产品设计
  - ✅技术派交互视觉设计
  - ✅技术派架构方案设计
  - ✅技术方案详细设计
  - ✅技术派项目管理流程
  - ✅技术派项目管理研发阶段
  - ✅代码约束规范
  - 技术调研和选型
- 基础篇
  - ✅技术派中的多配置文件说明
  - ✅技术派整合 Knife4j 实现在线 API 文档
  - ✅技术派整合MyBatis-Plus的基本使用
  - ✅技术派是如何应用MVC分层架构的?
  - ✅技术派中的实体对象DO,DTO,VO
  - ✅技术派整合Lombok，让代码更简洁
  - ✅技术派整合logback/lombok配置日志输出
  - ✅技术派中的全局异常处理
  - ✅技术派中的跨域问题解决方案
  - ✅WEB三大组件之Filter在技术派中的应用
  - ✅WEB三大组件之Servlet在技术派中的应用
  - ✅WEB三大组件之Listener在技术派中的应用
  - ✅技术派身份验证识别之session-cookie
  - ✅技术派是如何通过 AOP 实现切面日志的?
  - ✅技术派是如何解析请求参数的?
  - ✅技术派基于@Schedule注解实现定时任务
  - ✅技术派整合邮件服务实现邮件发送
  - ✅技术派中的事务使用实例
  - ✅技术派中使用事务的7条注意事项
  - ✅技术派中的 Spring 事件监听机制及原理
  - ✅技术派中是如何实现原生图片上传的
  - ✅技术派整合本地缓存之Guava
  - ✅技术派整合本地缓存之Caffeine
  - ✅技术派整合本地缓存Caffeine采坑实录
  - ✅技术派中基于Cacheable注解实现缓存示例
  - ✅技术派整合Redis（多 Redis 配置、Redis 集群）
  - ✅技术派中基于Redis的缓存示例
  - ✅技术派中基Redis实现作者白名单
  - ✅技术派实时在线人数统计-单机版
  - 技术派中基于redis实现计数统计
  - 技术派中基于redis实现用户活跃排行榜
  - 技术派如何解析试图返回
  - 技术派身份验证之JWT
- 进阶篇
  - ✅技术派中启动时端口号冲突的解决方案
  - ✅技术派之深入理解数据库连接池HikariCP
  - ✅技术派中的微信公众号自动登录方案
  - ✅技术派之扫码登录实现原理
  - ✅技术派中记录 SQL 执行日志的两种方式
  - ✅技术派中基于异常日志的报警通知
  - ✅技术派整合消息队列RabbitMQ
  - ✅技术派整合消息队列RabbitMQ连接池
  - ✅技术派中的缓存一致性解决方案
  - ✅技术派是如何利用Canal保证MySQL和Redis最终一致性的?
  - ✅技术派整合Redis分布式锁
  - ✅技术派整合xxl-job实现定时任务
  - ✅技术派整合Cancal实现MySQL和ElasticSearch同步
  - ✅技术派整合ES实现首页全局查询
  - ✅技术派整合Actuator、Prometheus、Grafana搭建应用监控
  - ✅技术派整合消息队列Kafka
  - 技术派整合zk分布式锁
  - 技术派中的网络请求WebClient
  - 技术派中的网络请求RestTemplate
  - 技术派中的数据埋点PV-UV示例
  - 技术派中的权限判定方案
  - 技术派中SpEL在AOP中的应用
  - 技术派整合SpringCloudConfig配置中心
  - 技术配整合Apollo配置中心
  - 技术派整合Nacos配置中心
  - 技术派整合ZK配置中心
  - 技术派整合elastic-job实现定时任务
  - 技术派整合Quartz Cluster实现定时任务
  - 技术派中自定义的分布式定时任务实现方案
  - 技术派整合SpringSecurity
  - 技术派数据库配置之druid
  - 技术派中的多数据源配置
  - 技术派整合分布式事务
  - 技术派整合MongoDB
  - 技术派的Bean拷贝之MapStruct

02、编程喵实战笔记

编程喵是一套成熟的学习教程网站，包括前台网站内容展示系统，以及后台网站内容管理系统，采用时下最流行的 Spring Boot + Vue 的前后端分离架构。配套的教程可以带你完成从小白到初级工程师的蜕变。

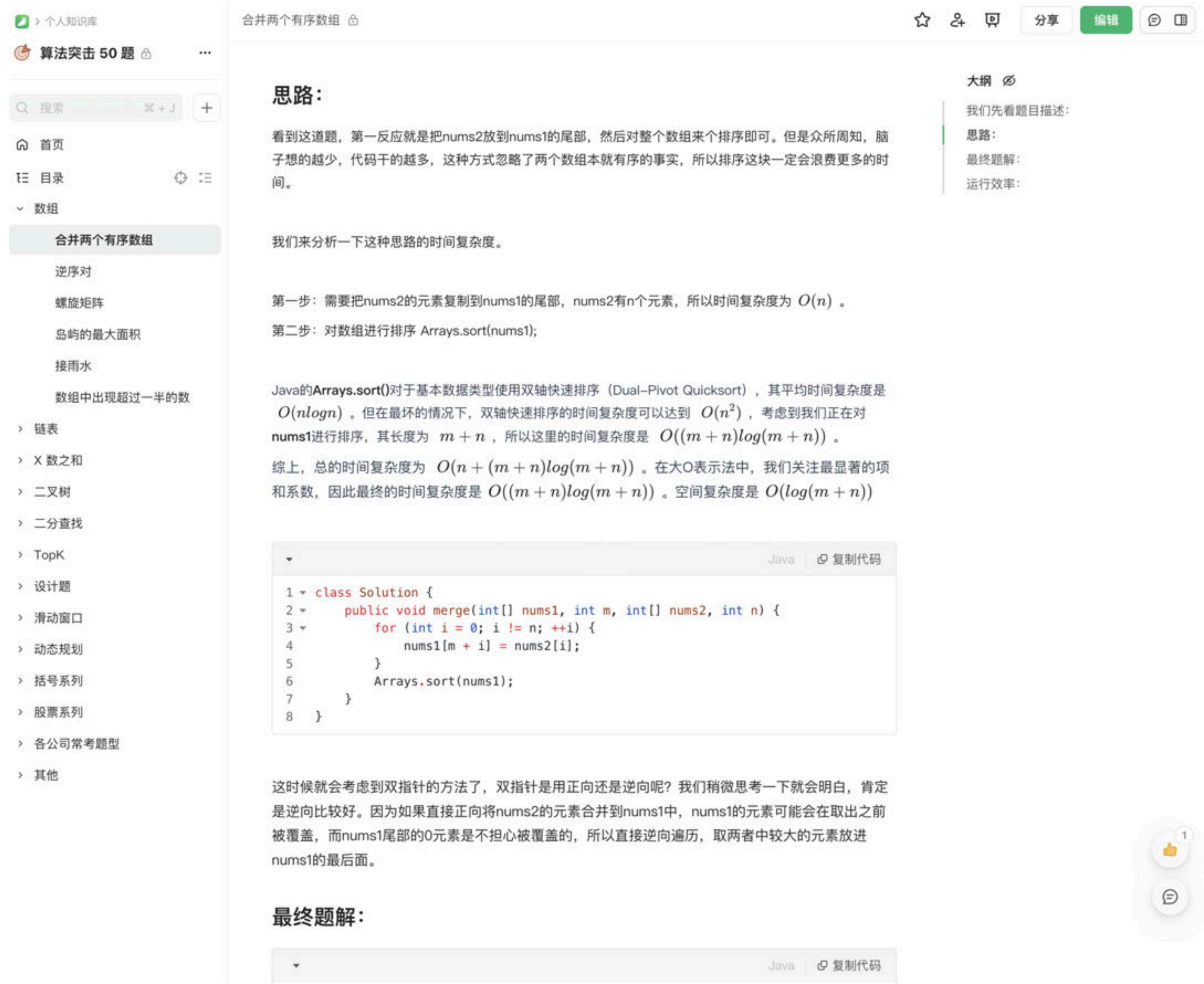


03、二哥的 LeetCode 刷题笔记

《二哥的 LeetCode 刷题笔记》，不仅有详细地解题思路，还有完整的代码示例，力求教会你举一反三的解题能力。

04、算法突击 50 题

准备秋招/春招/社招的小伙伴不少，但往往时间比较紧张，很多小伙伴精力有限，所以我这里精选了 50 道高频算法题，作为笔试的重点突击题型，可以在短时间内最大效率地提升你的笔试通过率。



05、华为 OD 笔试 AB 卷题库

LeetCode 的模式是你只需要输入核心代码就可以了，华为 OD 机考使用的是 ACM 模式，也就是需要手动编写输入输出的模式，更贴近机考的真实场景，所以我这里整理了一套完整的 AB 卷题库，帮助大家在最短的时间内快速提升机考的通过率。

星球限时优惠

一年前，星球的定价还是 99 元一年，第一批优惠券的额度是 30 元，等于说 69 元的低价就可以加入，再扣除掉星球手续费，几乎就是纯粹做公益。

随着时间的推移，星球积累的干货/资源越来越多，我花在星球上的时间也越来越多，星球的知识图谱里沉淀的问题，你可以戳这个链接去感受一下。有学习计划啊、有学生党秋招&春招&offer选择&考研&实习&专升本&培训班的问题啊、有工作党方向选择&转行&求职&职业规划的问题啊，还有大大小小的技术细节，我都竭尽全力去帮助球友，并且得到了球友的认可和尊重。

目前星球已经 4200+ 人了，所以星球也涨价到了 149 元，后续会讲星球的价格调整为 159 元/年，所以想加入的小伙伴一定要趁早。

沉默王二

邀请你加入星球，一起学习

Java程序员进阶之路

星主：沉默王二

4200+	1.8万	688
成员数量	内容数量	运营天数

这是一个编程学习指南+Java项目实战+leetcode刷题的高性价比圈子，欢迎想进步的小伙伴！加入后先阅读星球的两个置顶帖，你会发现物超所值✌️。...

知识星球

微信扫码加入星球 ▶

你可以微信扫码或者长按自动识别领取 30 元优惠券（仅有 100 张），**119/年** 加入，满 5000 人会涨价至 159 元，所以想要加入的话请趁早。

对了，**加入星球后记得花 10 分钟时间看一下星球的两个置顶贴，你会发现物超所值**！

成功没有一蹴而就，没有一飞冲天，但只要你能够一步一个脚印，就能取得你心满意足的好结果，请给自己一个机会！

最后，把二哥的座右铭送给你：**没有什么使我停留——除了目的，纵然岸旁有玫瑰、有绿荫、有宁静的港湾，我是不系之舟。**

共勉 ⛽。

如何贡献?

对了，如果你在阅读的过程中遇到一些错误，欢迎到我的开源仓库提交 issue、PR（审核通过后可成为 Contributor），我会第一时间修正，感谢你为后来者做出的贡献。

> - GitHub：https://github.com/itwanger/toBeBetterJavaer
> - 码云：https://gitee.com/itwanger/toBeBetterJavaer

更新记录

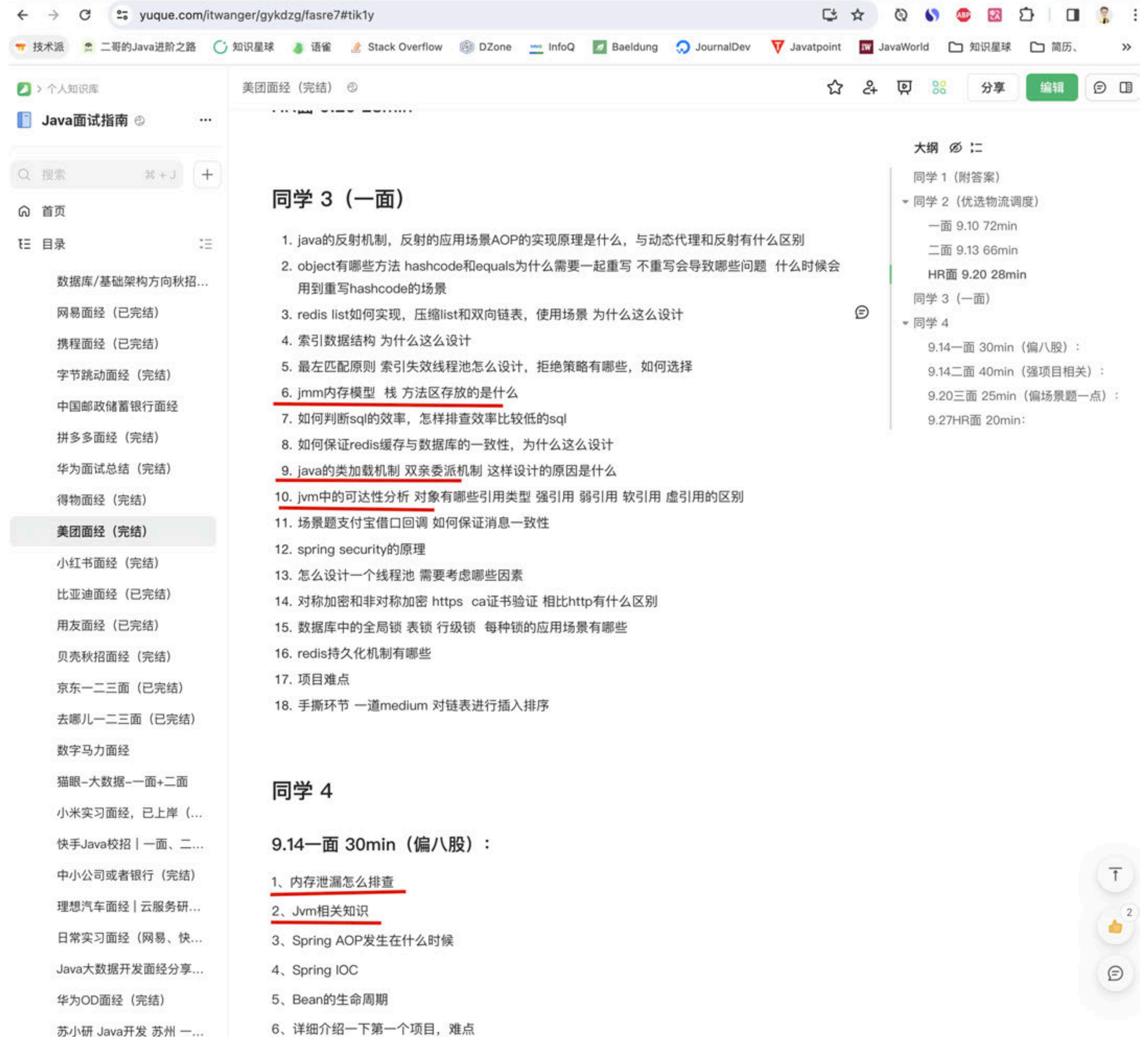
V1.0 版 2024年01月16日

第一版《二哥的JVM进阶之路》正式完结发布！

第一节：大白话带你认识 JVM

“二哥，之前你讲 Java 的第一行代码 hello world 的时候就提到了 JVM，那时候我就想知道 JVM 到底是什么，但你说这是一块非常大的内容，会放到后面专门来讲，那学完了 Java 基础知识，又学完了并发编程，今天我们就来学习 JVM 吧？”三妹咪了一口麦香可可奶茶后对我说。

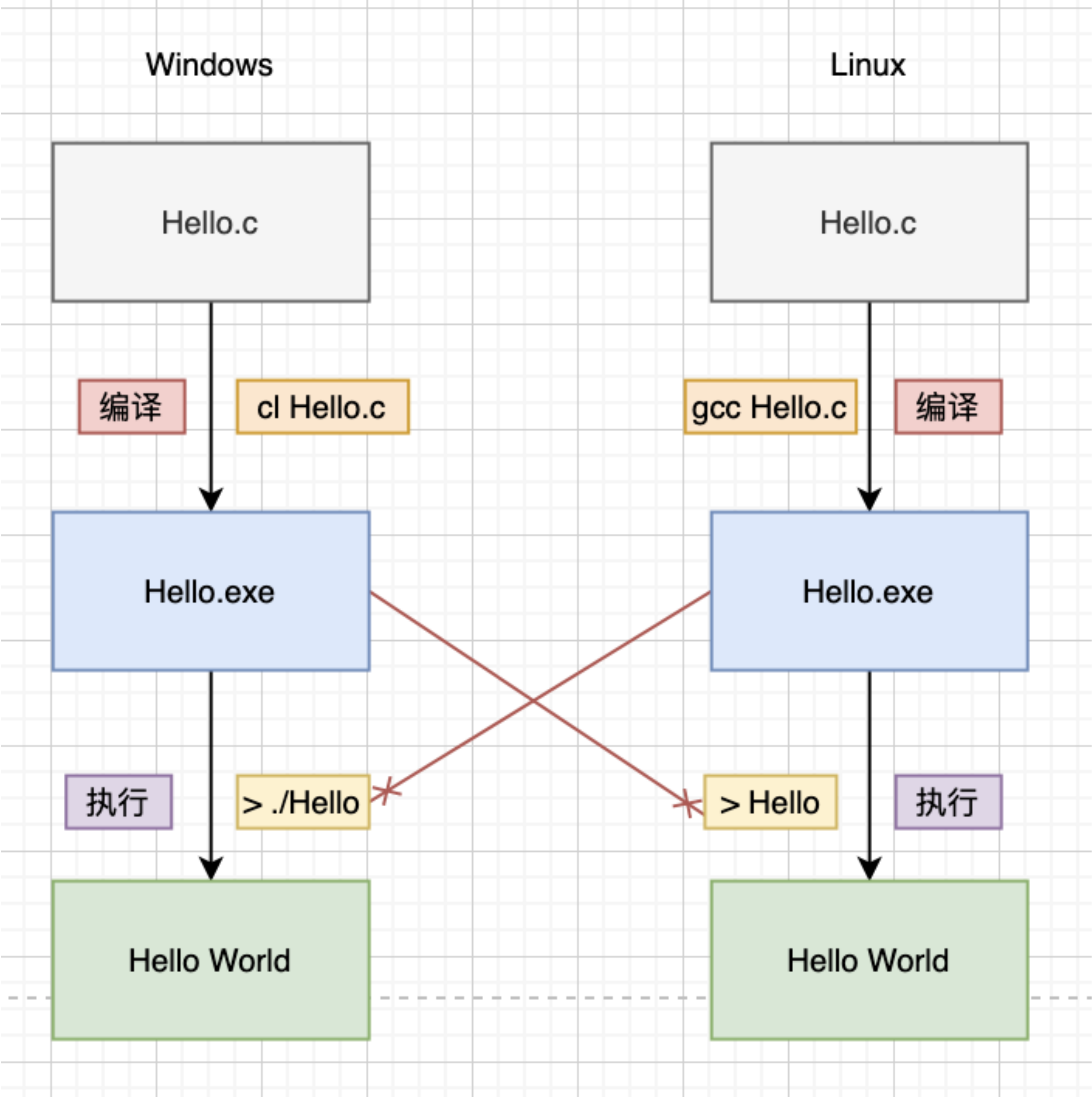
“好的，三妹，这篇内容就来带你认识一下什么是 JVM， JVM 是 Java 体系中非常重要，又有一些难度的知识，但每个想要更加优秀的程序员都应该掌握它。尤其是想去大厂或者中厂的球友，更应该掌握它，因为 JVM 在大中厂面试的时候，比重很大，我随便从《Java 面试指南》中截张图大家感受一下。”我回答。



“JVM 在校招面试中的比重还是非常大的；同时，对于工作党来说，如果项目遇到内存泄露、CPU飙升的问题，也需要通过 JVM 的性能监控进行定位和解决。好，那就让我们开始吧！”我继续补充道。

三妹，你看过《Java 发展简史》应该知道，Sun 在 1991 年成立了一个由詹姆斯·高斯林（James Gosling）领导的，名为“Green”的项目组，目的是开发一种能够在各种消费性电子产品上运行的程序架构。

一开始，项目组打算使用 C++，但 C++ 无法达到跨平台的要求，比如在 Windows 系统下编译的 Hello.exe 无法直接拿到 Linux 环境下执行。

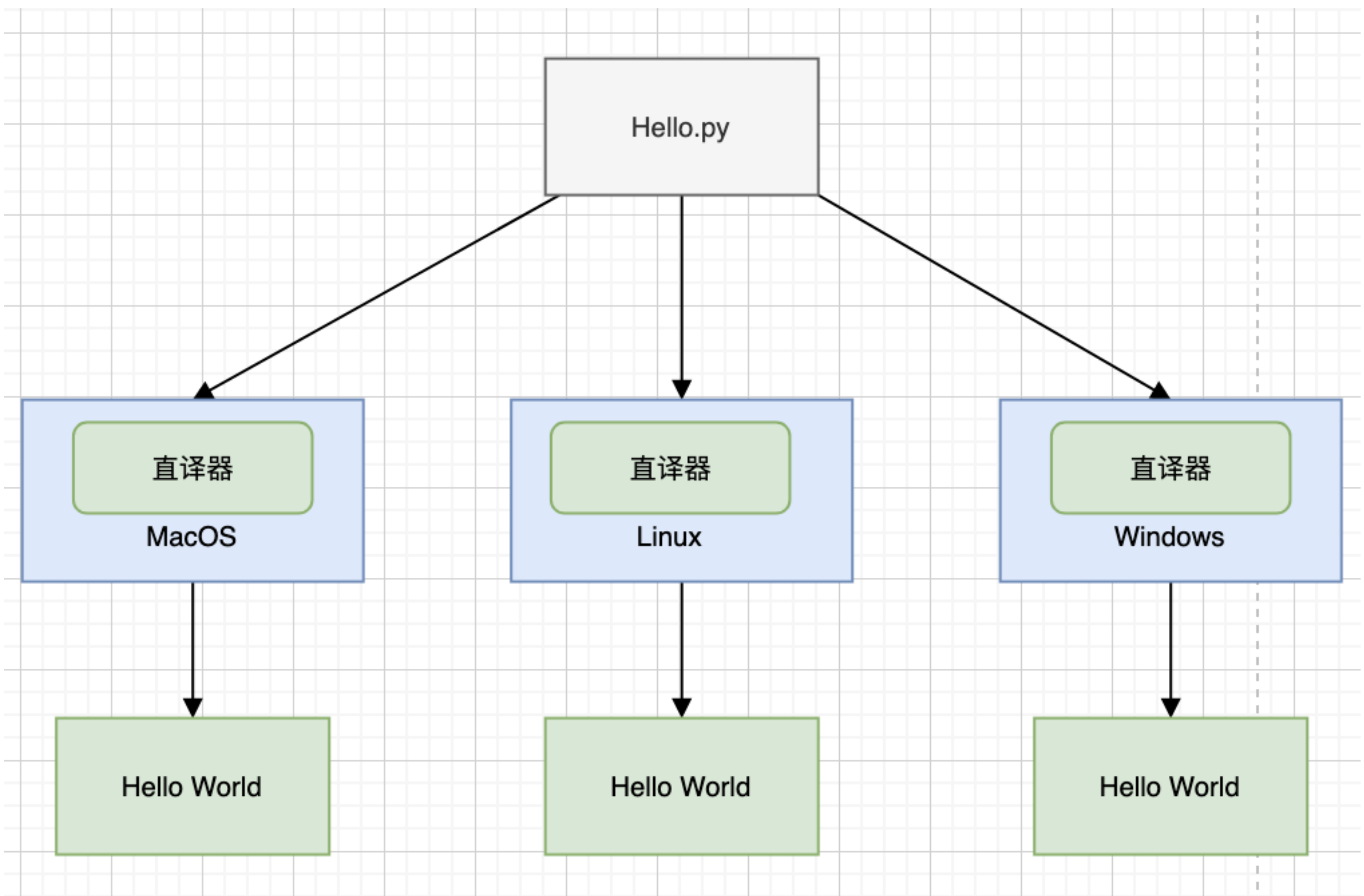


在当时，C++ 已经非常流行了，但无法跨平台，只能忍痛割爱了。

怎么办呢?

三妹不知道有没有听过直译器（解释器）这玩意？（估计你没听过）就是每跑一行代码就生成机器码，然后执行，比如说 Python 和 Ruby 用的就是直译器。

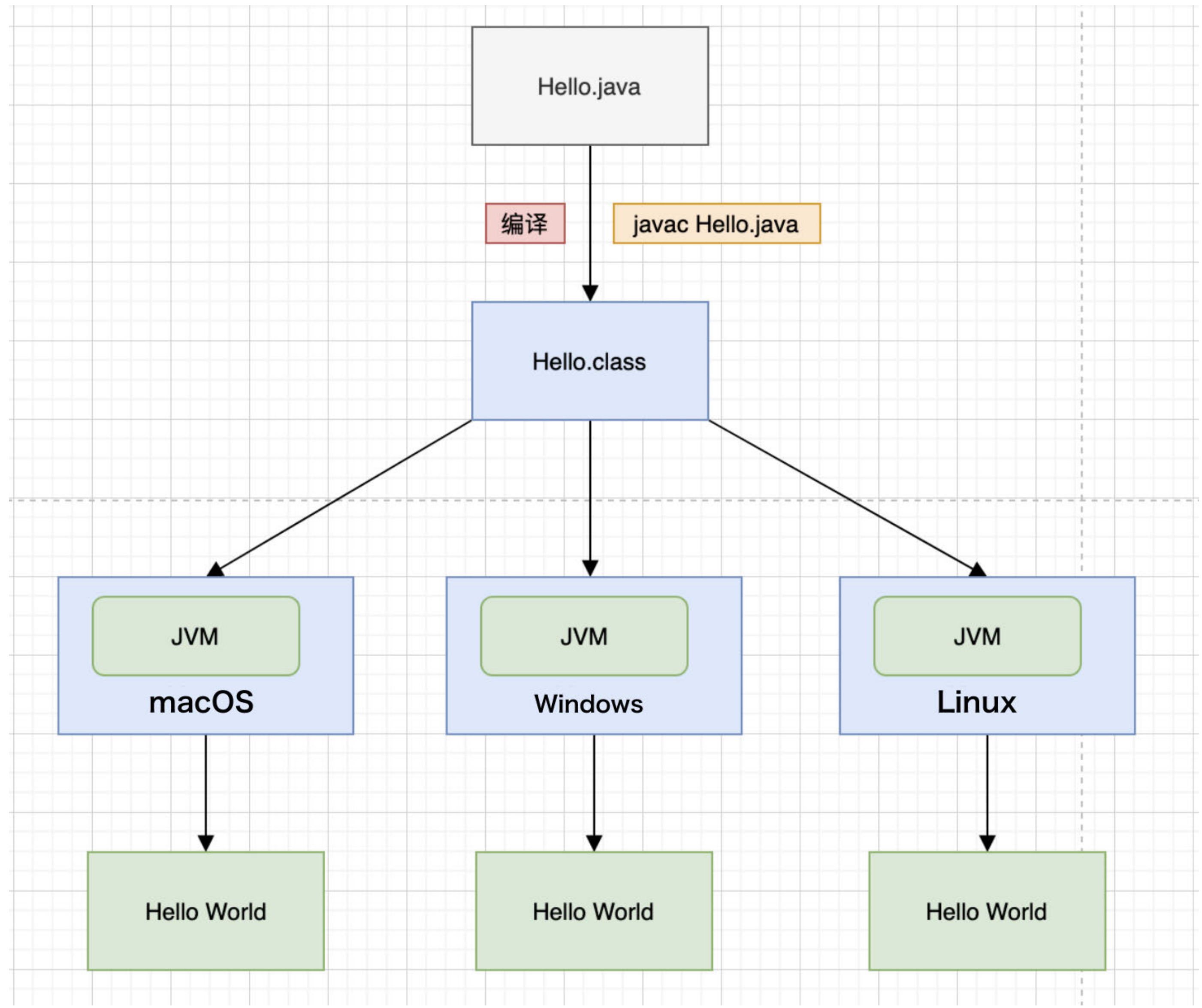
“那在每个操作系统上装一个直译器就好了，跨平台的目的就达到了啊。”三妹插话道。



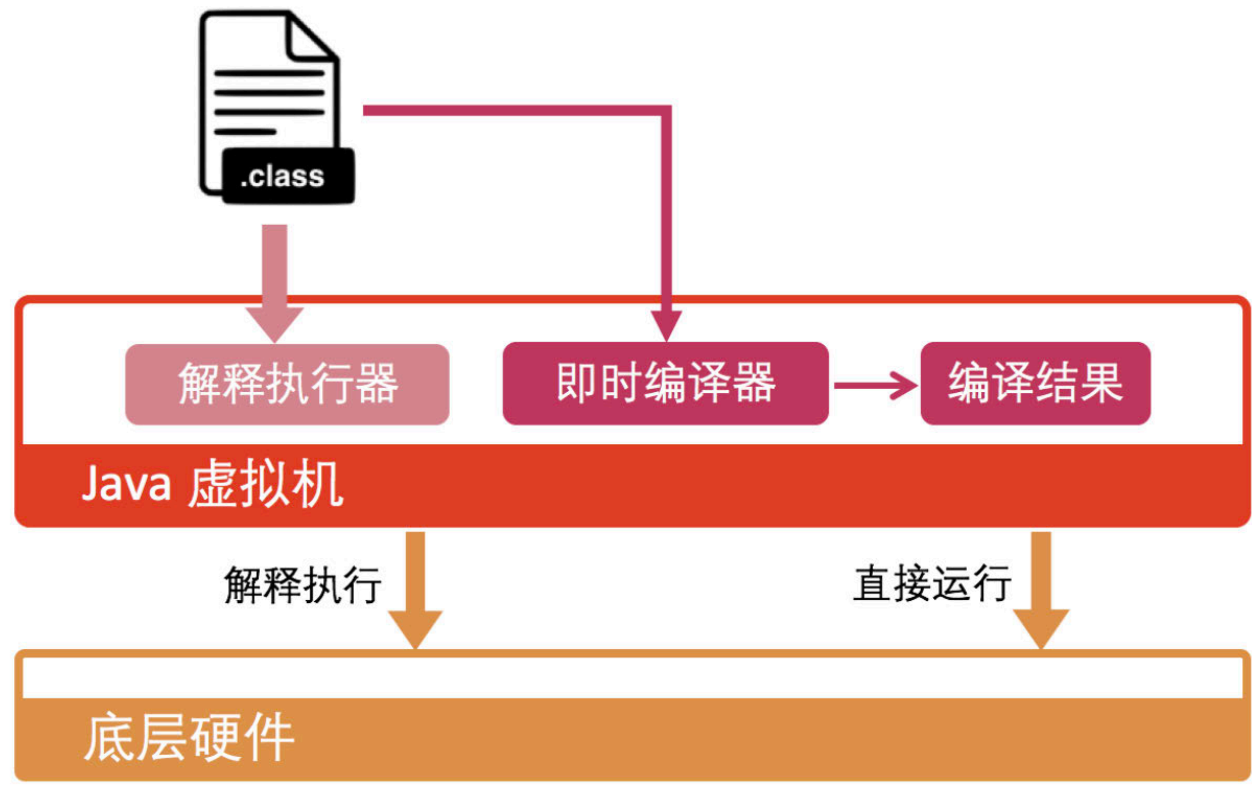
“但直译器有个缺点，就是没法像编译器那样对一些热点代码进行优化，从而让程序在机器上跑得更快一些。”我回答说。

怎么办呢？

来个结合体呗，编译器和直译器一块上！



编译器负责把 Java 源代码编译成字节码，Java 虚拟机负责把字节码转换成机器码。转换的时候，可以做一些压缩或者优化，通过 JIT 来完成，这样的程序跑起来就快多了。



> - Java 虚拟机：Java Virtual Machine，简称 JVM，也就是我们接下来要学习的重点。
> - 字节码：Bytecode，接下来会细讲。
> - JIT：Just-In-Time，即时编译器，后面会细讲。

这样的话，不仅跨平台的目的达到了，而且性能得到了优化，两全其美！

“为什么 Java 虚拟机会叫 Java 虚拟机呢？”三妹问了一个很古怪的问题。

虚拟机，顾名思义，就是虚拟的机器（多苍白的解释），反正就是看不见摸不着的机器，一个相对物理机的叫法，你把它想象成一个会执行字节码的怪兽吧。

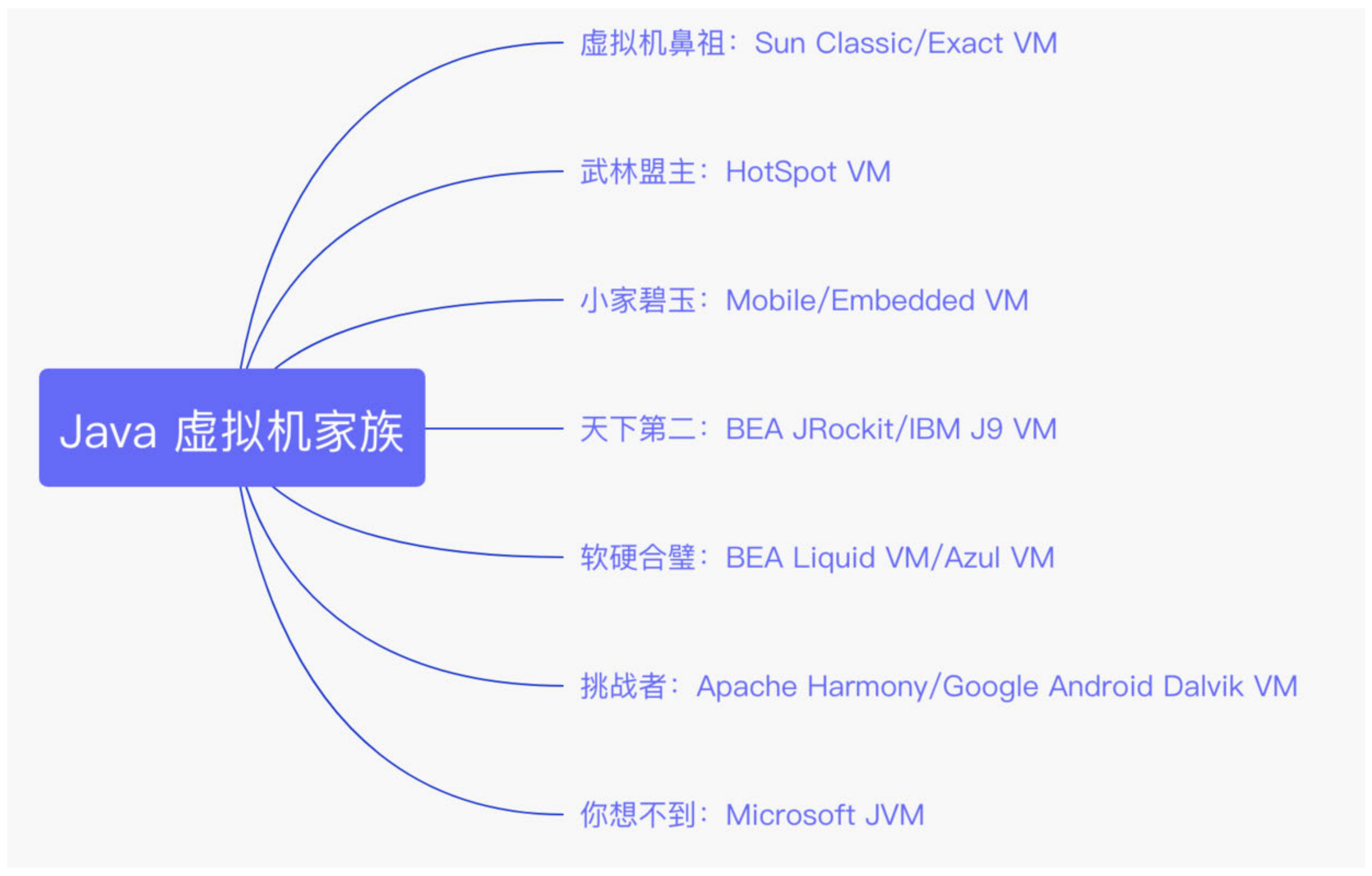
记得上大学那会，由于没有 Linux 环境，但又需要在上面玩一些命令，于是我就会在 Windows 上装一个 Linux 的虚拟机，这个 JVM 就类似这种东西。

说白了，就是我们编写 Java 代码，编译 Java 代码，目的不是让它在 Linux、Windows 或者 MacOS 上跑，而是在 JVM 上跑。

JVM 家族

我刚讲完这些，三妹就问：“都有哪些 Java 虚拟机呢？”

来看下面这张思维导图，一目了然。



除了我们经常看到、听到的 Hotspot VM，还有很多，下面我来简单介绍一下。

- Sun Classic：世界上第一款商用 Java 虚拟机，但执行效率低下，导致 Java 程序的性能和 C/C++ 存在很大差距，因此给后来者留下了“Java 语言很慢”的刻板印象。
- Exact VM：为了提升 Classic 的效率，Sun 的虚拟机团队曾在 Solaris（Sun 研发的一款类似 Unix 的操作系统）上发布过这款虚拟机，它的执行系统里包含有热点探测、即时编译等，但不是很成熟。Sun Classic 在 JDK 1.4 的时候被彻底抛弃，而 Exact VM 被抛弃得更早，取代它的正是 HotSpot VM——时也命也。
- Mobile VM：Java 在移动手机端（被 Android 和 IOS 二分天下）的发展并没有那么成功，因此 Mobile VM 的声望值比较低。
- Embedded VM：嵌入式设备上的虚拟机。
- BEA JRockit：曾经号称是“世界上最快的 Java 虚拟机”，后来被 Oracle 收购后就没有声音了。
- IBM J9 VM：提起 IBM，基本上所有程序员都知道了，也是个巨头，所以他家的虚拟机也很强，在职责分离和模块化上做得比 HotSpot 更好。目前已经开源给 Eclipse 基金会。
- BEA Liquid VM：是 BEA 公司开发的可以直接运行在自家系统上的虚拟机，可以越过操作系统直接和硬件打交道，因此可以更大程度上的发挥硬件的能力。不过核心用的还是 JRockit，所以伴随着 JRockit 的消失，Liquid VM 也退出历史舞台了。
- Azul VM：是 Azul 公司在 HotSpot 基础上进行大量改进后的，可以运行在 Azul 公司专有硬件上的虚拟机。2010 年起，Azul 公司的重心从硬件转移到软件上，并发布了 Zing 虚拟机，性能方面很强大。
- Apache Harmony 和 Google Android Dalvik VM 并不是 严格意义上的 Java 虚拟机，但对 Java 虚拟机的发展起到了很大的刺激作用。但它们终究没有熬过时间。
- Microsoft JVM：在早期的 Java Applets 年代，微软为了在 IE 中支持 Applets 开发了自己的 Java 虚拟机。你敢相信？Microsoft JVM 只有 Windows 版本，它与 JVM 实现的“一次编译，到处运行”的理念完全沾不上边。关键是，1997 年 10 月，Sun 公司因为这事把微软告了，最后微软赔给了 Sun 公司 2000 万美金，并且终止了在 Java 虚拟机方面的发展。如果，我是说如果，如果微软保持着对 Java 的热情，后面还有 .Net 什么事？

最后再来讲一下：HotSpot VM，OracleJDK（商用）和 OpenJDK（开源）的默认虚拟机，也是目前使用最广泛的 Java 虚拟机，也就是传说中的太子，Java 虚拟机中的嫡长子。

HotSpot 的技术优势就在于热点代码探测技术（名字就从这来的）和准确式内存管理技术，但其实这两个技术在 Exact VM 中都有体现，因此你看起个好的名字多重要（开玩笑了，这就是命）。

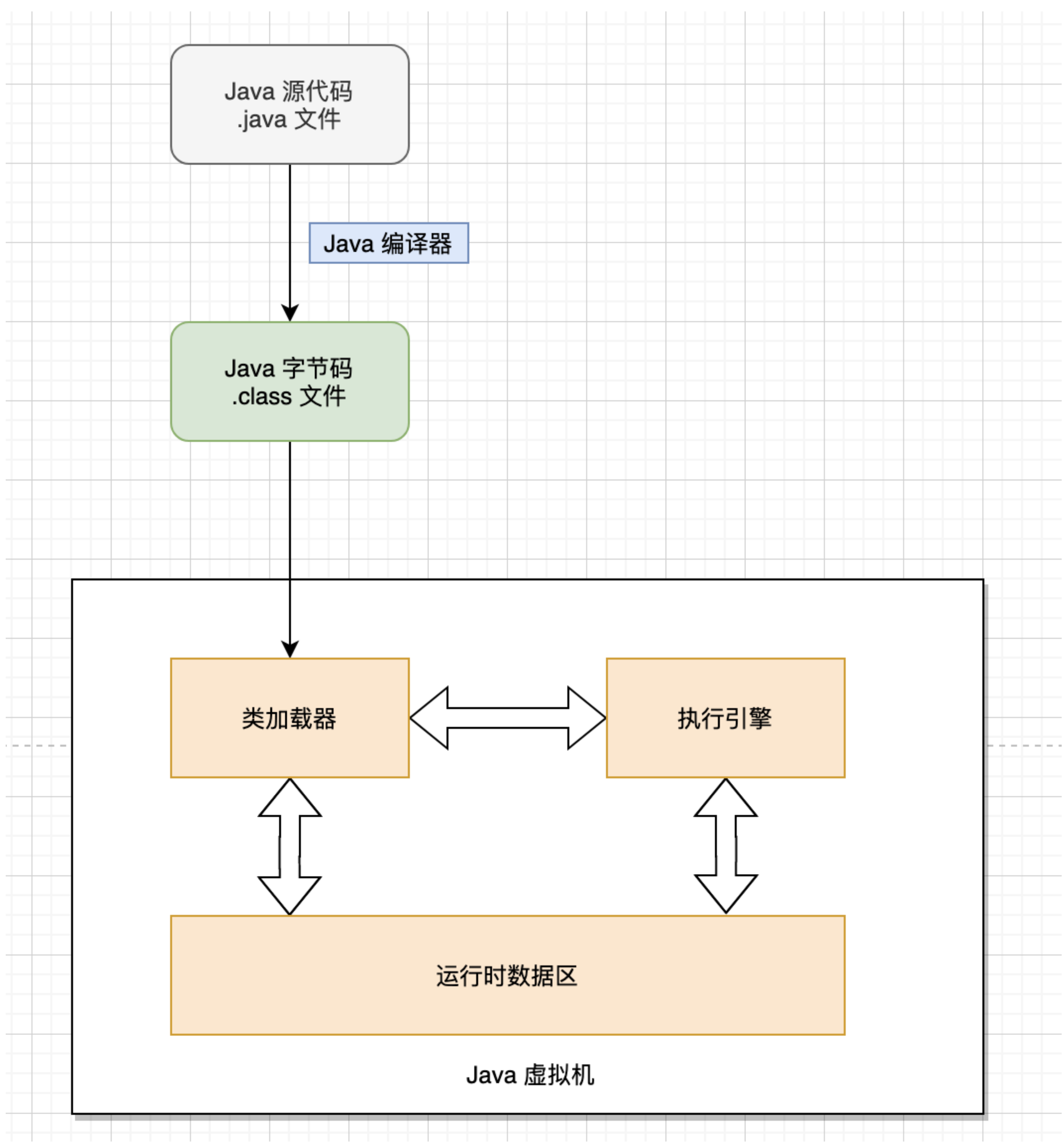
热点代码探测，指的是，通过执行计数器找出最具有编译价值的代码，然后通知即时编译器以方法为单位进行编译，解释器就可以不再逐行的将字节码翻译成机器码，而是将一整个方法的所有字节码翻译成机器码再执行。

这样的话，效率就提高了很多，对吧？也就是我们后面会讲到的 JIT 技术。

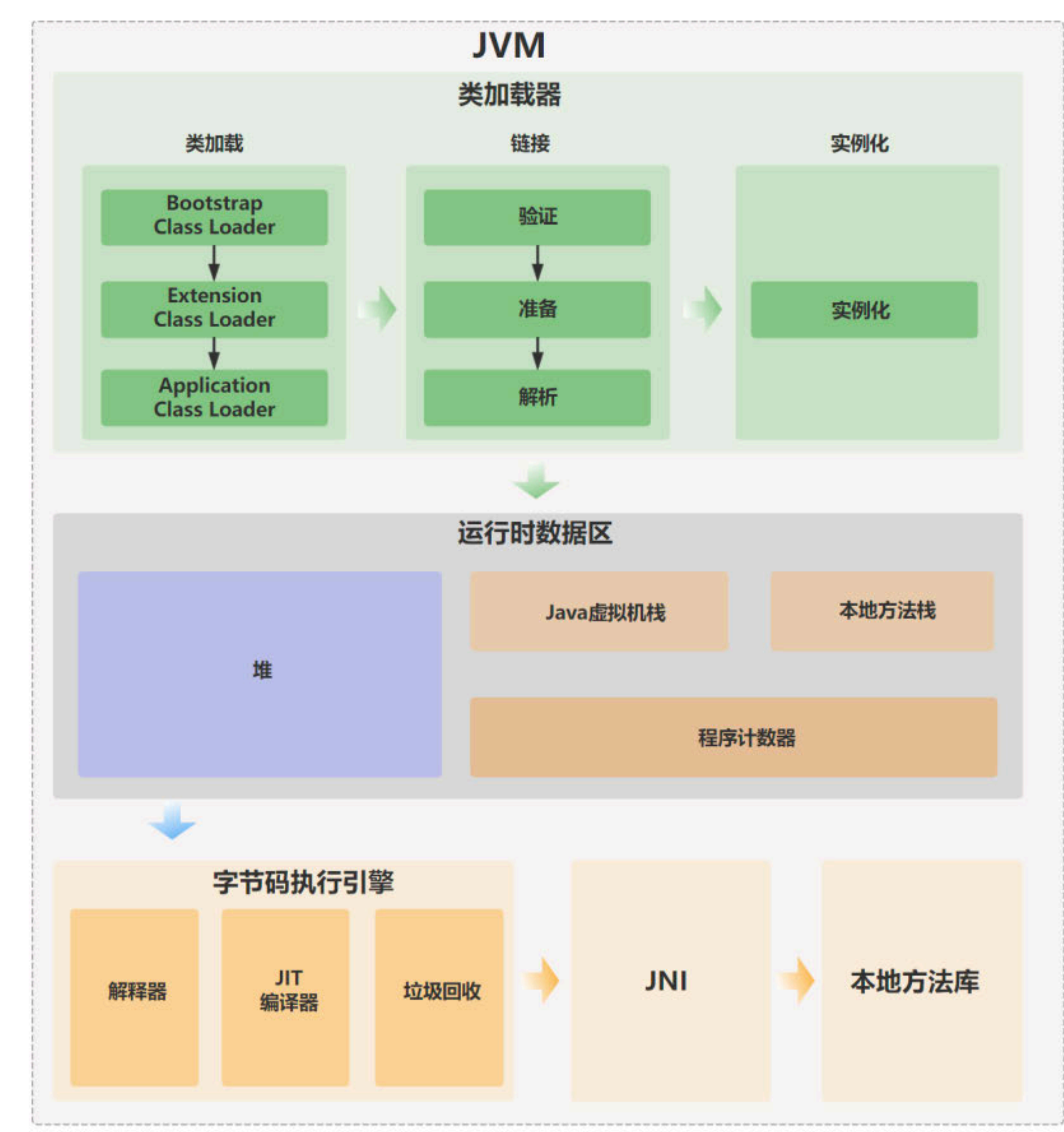
JVM 的组织架构

"JVM 的组织架构是什么样子的呢？它由哪些单位组成的呢？"三妹继续追问到。

JVM 大致可以划分为三个部门，分别是类加载器（Class Loader）、运行时数据区（Runtime Data Areas）和执行引擎（Excution Engine），见下图。



这三个部门具体又是干什么的，可以通过下面这幅图来了解。



类加载器

类加载器是 JVM 最有权威的一个部门，相当于明朝张居正时期的内阁，大全独揽，朝廷想干什么，都得经过我这关。

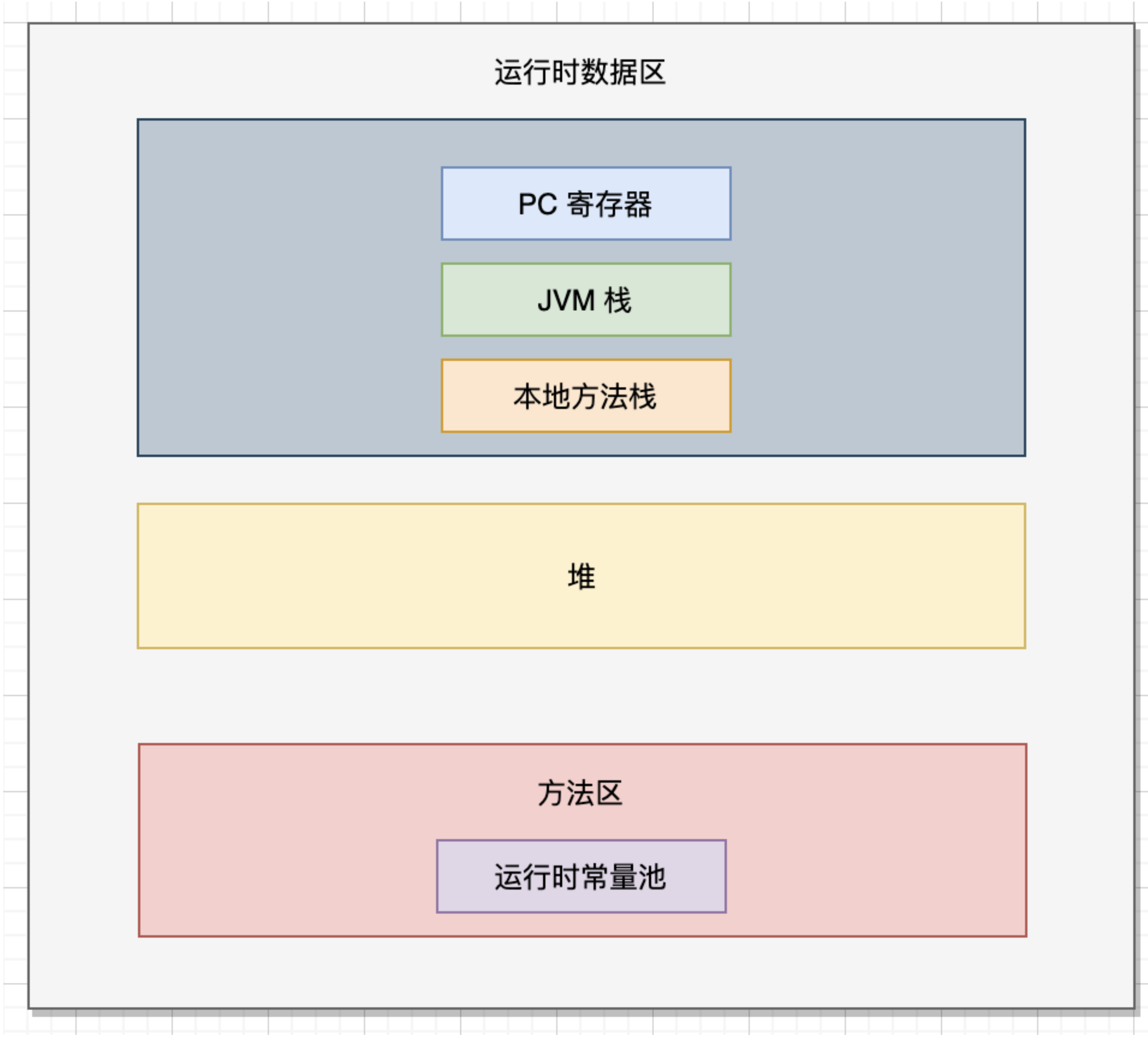
好，类加载器用来加载类文件，也就是 .class 文件。如果类文件加载失败，也就没有运行时数据区和执行引擎什么事了，它们什么也干不了。

类加载器负责将字节码文件加载到内存中，主要会经历加载->连接->实例化这三个阶段，我们会放在后面的章节单独来讲。

运行时数据区

运行时数据区就相当于明朝时期的国库，国库里有钱，那接下来的执行引擎就能够继续执行字节码，国库里没钱就会抛出 OutOfMemoryError 异常。

JVM 定义了 Java 程序运行期间需要使用到的内存区域，简单来说，这块内存区域存放了字节码信息以及程序执行过程的数据，垃圾收集器也会针对运行时数据区进行对象回收的工作。看下面这张图就能理解（JVM 规范）：



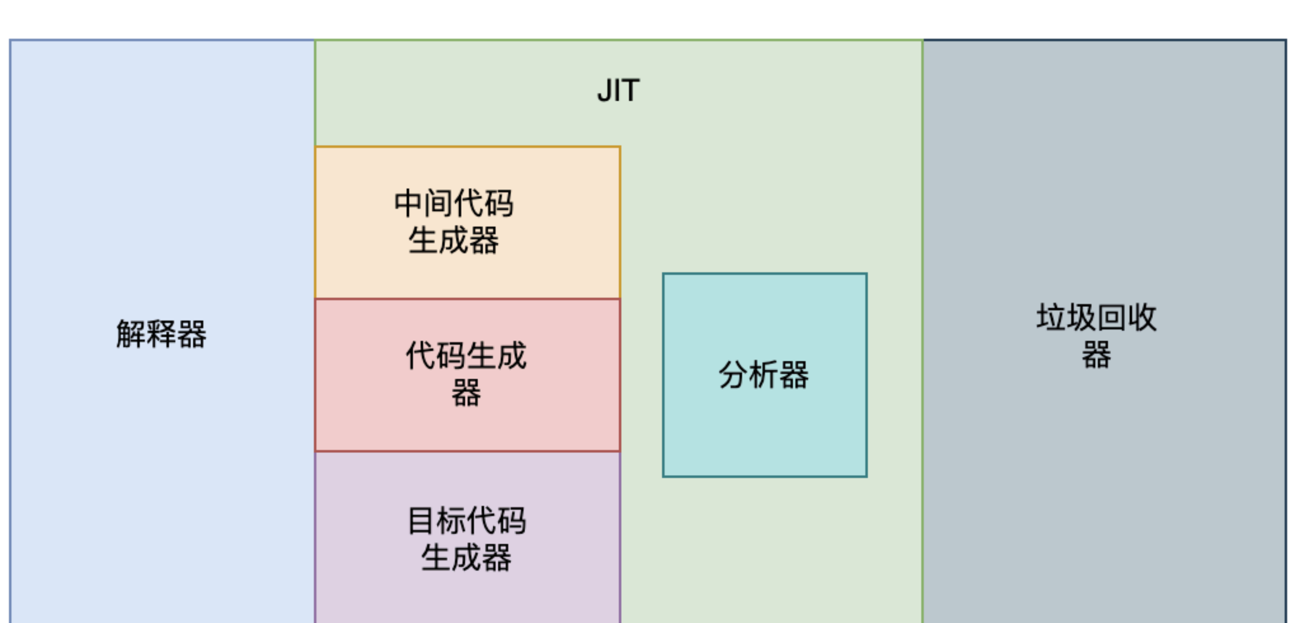
运行时数据区通常包括：方法区、堆、虚拟机栈、本地方法栈以及程序计数器五个部分。不过，运行时数据区的划分也随着JDK的发展不断变迁，JDK 1.6、JDK 1.7、JDK 1.8 的内存划分都会有所不同。

我们会在后面的章节单独来讲。

执行引擎

执行引擎（Execution Engine）就好像明朝时期的六部，主要用来干具体的事，“虚拟机”是一个相对于“物理机”的概念，这两种机器都有代码执行能力，其区别是物理机的执行引擎是直接建立在处理器、缓存、指令集和操作系统层面上的，而**虚拟机的执行引擎则是由软件自行实现的**，因此可以不受物理条件制约地定制指令集与执行引擎的结构关系，能够执行那些不被硬件直接支持的指令集格式。

执行引擎的任务就是将字节码指令解释/编译为对应平台上的本地机器指令才可以。简单来说，JVM 中的执行引擎充当了将高级语言翻译为机器语言的译者。



- 解释器：读取字节码，然后执行指令。因为它是一行一行地解释和执行指令，所以它可以很快地解释字节码，但是执行起来会比较慢（毕竟要一行执行完再执行下一行）。
- 即时编译器：执行引擎首先按照解释执行的方式来执行，随着时间推移，即时编译器会选择性的把一些热点代码编译成本地代码。执行本地代码比一条一条进行解释执行的速度快很多，因为本地代码是保存在缓存里的。
- 垃圾回收器，用来回收堆内存中的垃圾对象。

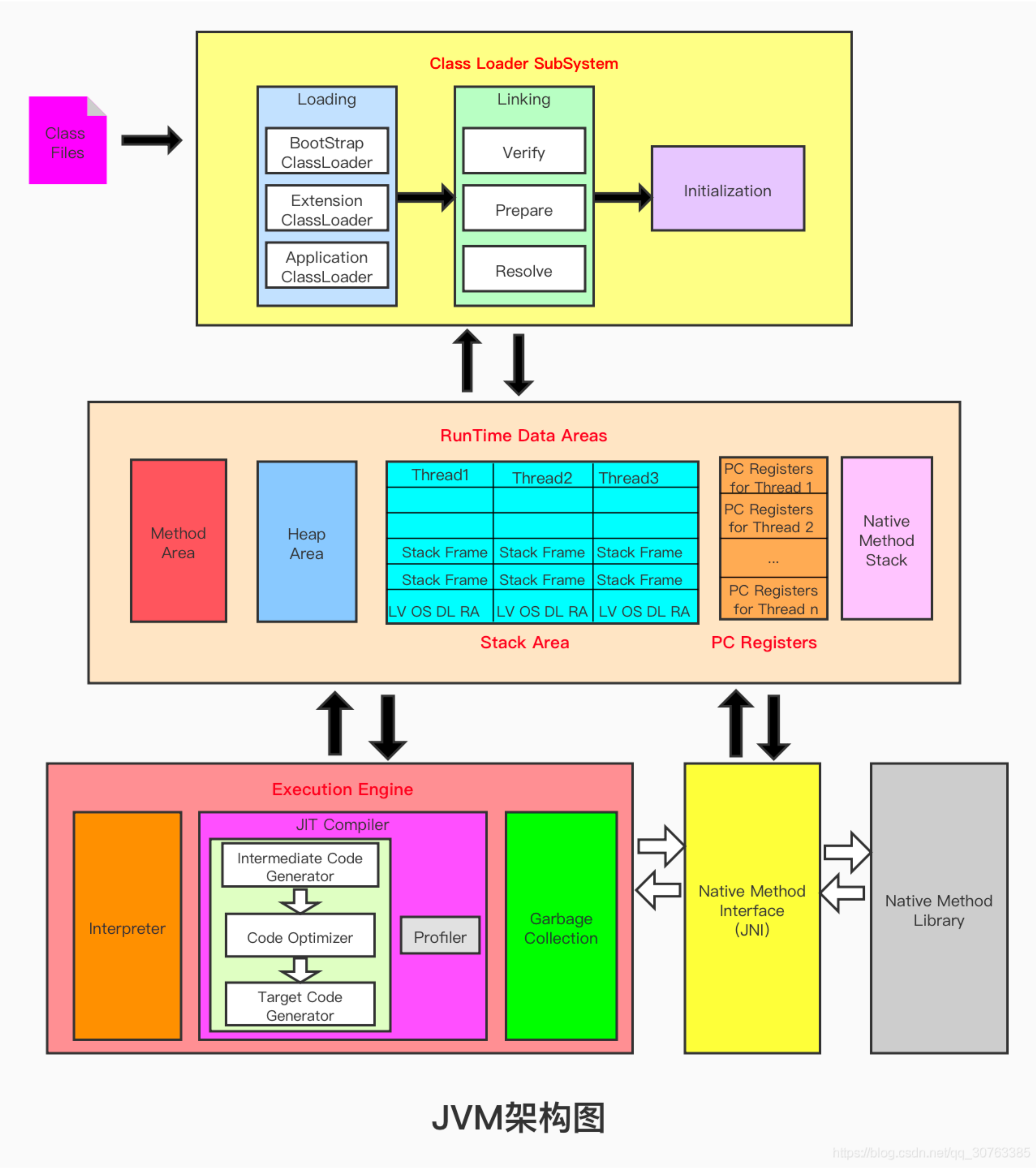
这些内容我们都会在后面的章节里单独来细讲。

小结

“三妹，关于 JVM 是什么的话题我们就先讲到这，后面再展开细讲其中的每一个细节，怎么样？”转动了一下僵硬的脖子后，我对三妹说，“JVM 是一块很大很深的内容，如果一上来学太多的话，我怕难倒你（😁）。”

“好的，二哥，我也觉得今天的知识量够了，我要好好消化几天。我会加油的！”三妹似乎对未来充满了希望，这正是我想看到的。

总的来说，JVM 是 Java 程序执行的环境，它隐藏了底层操作系统和硬件的复杂性，提供了一个统一、稳定和安全的运行平台。



JVM架构图

我们把 Java 源代码编译后的字节码文件扔给它，它就可以在 JVM 中执行，不管是在 Windows、Linux 还是 MacOS 环境下编译的，它都可以跑，屏蔽了底层操作系统的差异。

第二节：JVM如何运行Java代码?

“二哥，看了 Hello World 的代码后，我很好奇，它是怎么在 IDEA 的 Run 面板里打印出‘三妹，少看手机少打游戏，好好学，美美哒’呢？”三妹咪了一口麦香可可奶茶后对我说。

“三妹，我们通常把 Java 代码执行的过程分为编译期和运行时，弄清楚这两个阶段就知道原因了。”我微笑着对三妹说，“对于一个 Java 程序员来说，写了那么久的代码，总要搞清楚自己写的 Java 代码到底是怎么运行起来的。这个问题在面试的时候也经常会被问到。”

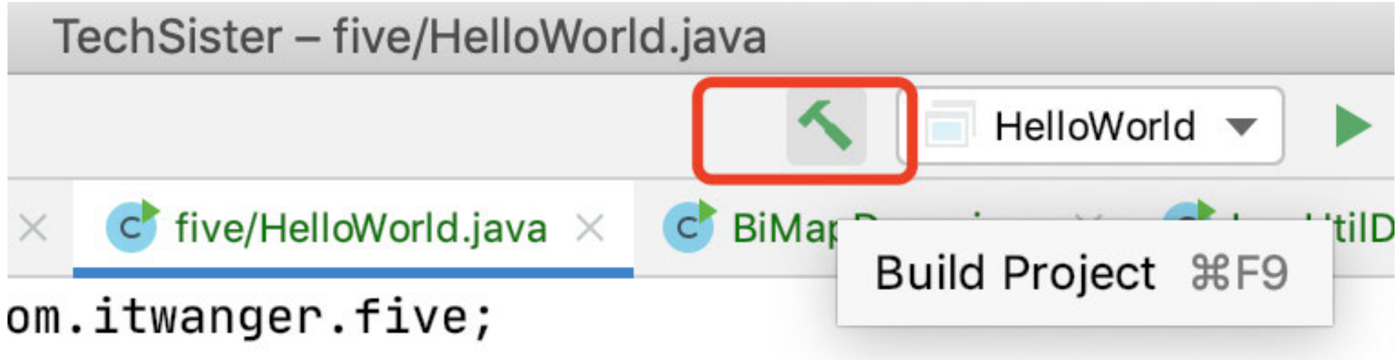
一起来看下吧。

编译期

贴一下 HelloWorld 这段代码：

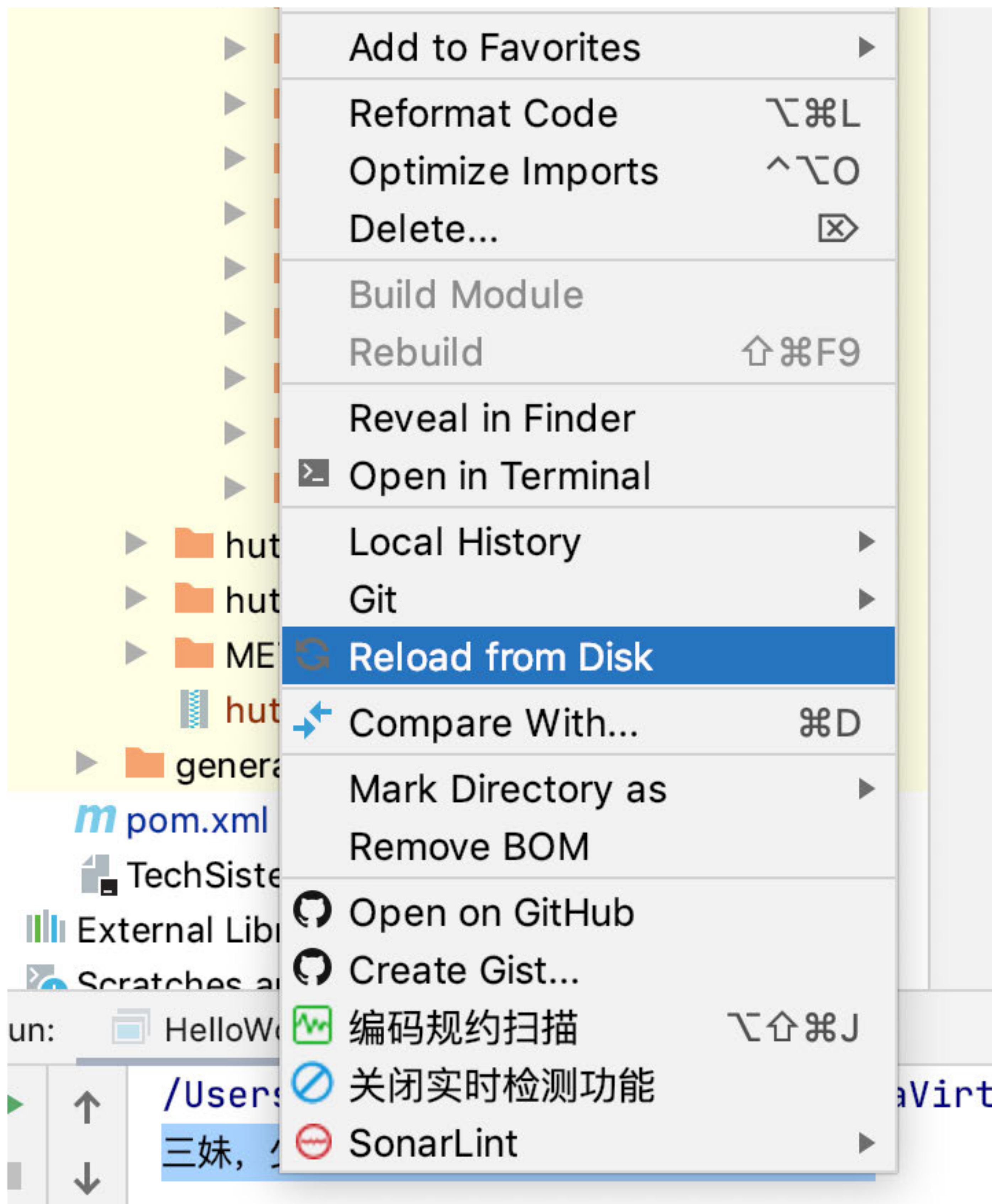
```
/**
 * @author 微信搜「沉默王二」，回复关键字 PDF
 */
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("三妹，少看手机少打游戏，好好学，美美哒。");
    }
}
```

点击 IDEA 工具栏中的锤子按钮（Build Project，编译整个项目，通常情况下，并不需要主动编译，IDEA 会自动帮我们编译），如下图所示。



这时候，就可以在 src 的同级目录 target 下找到一个名为 HelloWorld.class 的文件。

如果找不到的话，在目录上右键选择「Reload from Disk，从磁盘上重新加载」，如下图所示：



可以双击打开它，看到如下所示的内容。

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by Fernflower decompiler)
//
```

```
package com.itwanger.five;

public class HelloWorld {
    public HelloWorld() {
    }

    public static void main(String[] args) {
        System.out.println("三妹，少看手机少打游戏，好好学，美美哒。");
    }
}
```

IDEA 默认会用 Fernflower 这个反编译工具将字节码文件（后缀为 .class 的文件，也就是 Java 源代码编译后的文件）反编译为我们可以看得懂的 Java 源代码。

但实际上，字节码文件并不是这样的，它包含了 JVM 执行的指令，还有类的元数据信息，如类名、方法和属性等。如果用 「show bytecode」打开字节码文件的话，它是下面这样子的：

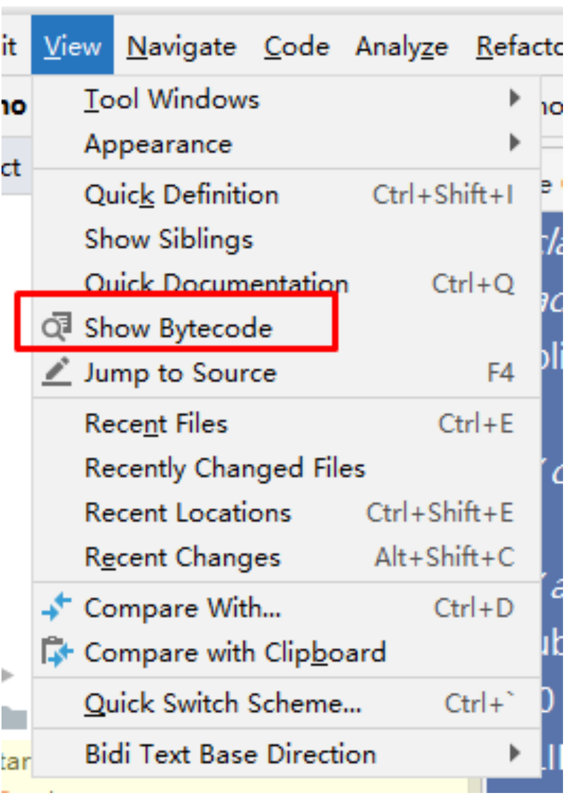
```
// class version 58.0 (58)
// access flags 0x21
public class com/itwanger/five/HelloWorld {

  // compiled from: HelloWorld.java

  // access flags 0x1
  public <init>()V
   L0
    LINENUMBER 6 L0
    ALOAD 0
    INVOKESPECIAL java/lang/Object.<init> ()V
    RETURN
   L1
    LOCALVARIABLE this Lcom/itwanger/five/HelloWorld; L0 L1 0
    MAXSTACK = 1
    MAXLOCALS = 1

  // access flags 0x9
  public static main([Ljava/lang/String;)V
   L0
    LINENUMBER 8 L0
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    LDC
"\u4e09\u59b9\uff0c\u5c11\u770b\u624b\u673a\u5c11\u6253\u6e38\u620f\uff0c\u597d\u597d\u5b66\uff0c\u7f8e\u7f8e\u54d2\u3002"
    INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V
   L1
    LINENUMBER 9 L1
    RETURN
   L2
```

```
    LOCALVARIABLE args [Ljava/lang/String; L0 L2 0
    MAXSTACK = 2
    MAXLOCALS = 1
}
```

是不是就有点看不懂了？我第一次看到这段内容的时候也很头大，不过不要担心，后面我们再一块深入研究，这里就提前感受一下 bytecode（也就是字节码）的魅力（😂）。

怎么查看 bytecode 呢?

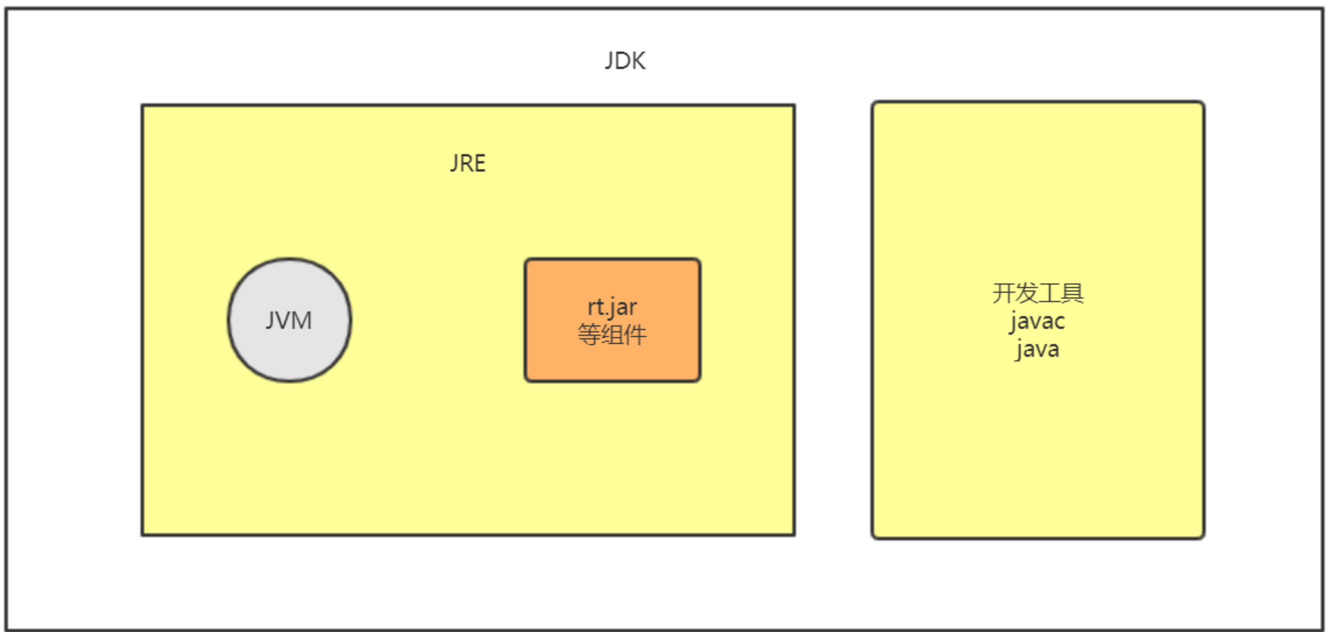
可以通过 IDEA 菜单栏中的「View」→「Show Bytecode」查看，如下图所示。



这个 bytecode 可以直译为字节码。

字节码并不是机器码，操作系统无法直接识别，需要在操作系统上安装不同版本的 JVM 来识别。

通常情况下，我们只需要安装不同版本的 JDK（Java Development Kit，Java 开发工具包）就行了，它里面包含了 JRE（Java Runtime Environment，Java 运行时环境），而 JRE 又包含了 JVM。



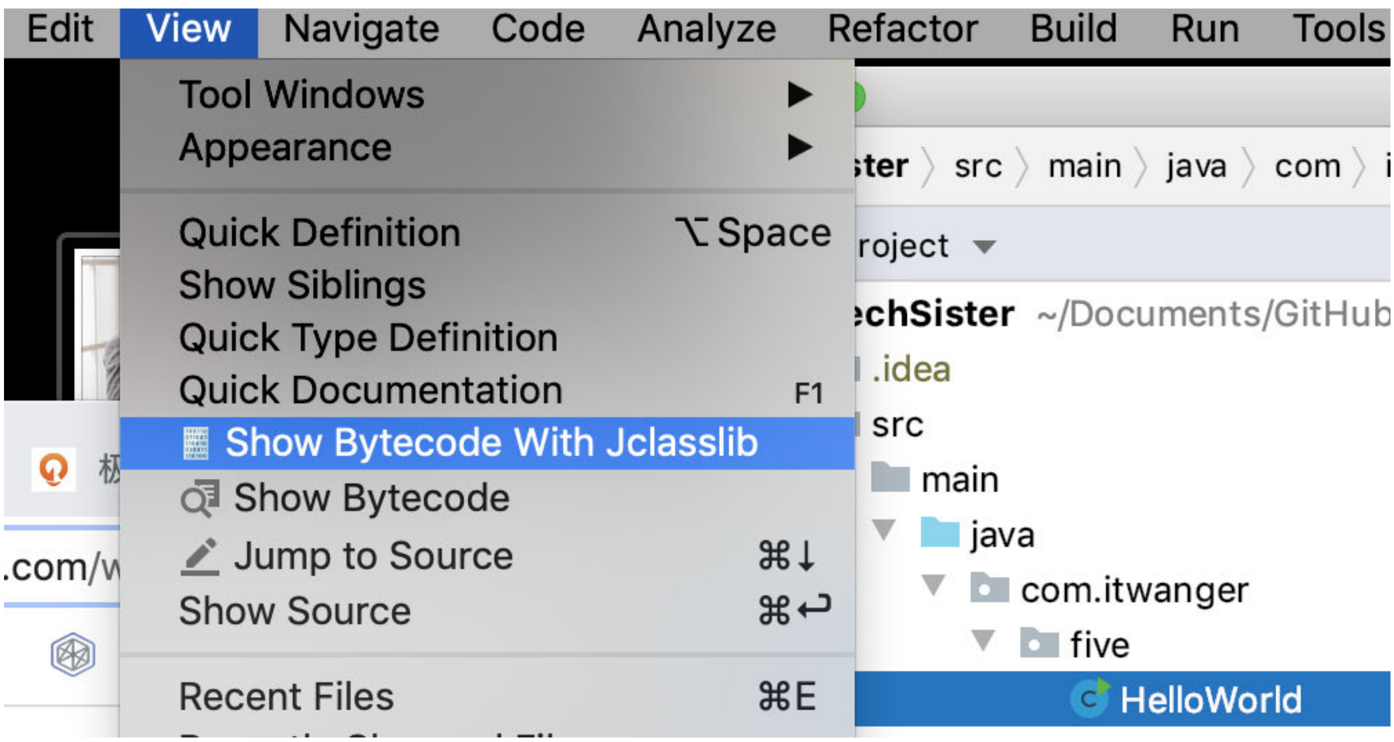
Windows、Linux、MacOS 等操作系统都有相应的 JDK，只要安装好了 JDK 就有了 Java 的运行时环境，就可以把 Java 源代码编译为字节码，然后字节码又可以在不同的操作系统上运行了。

> build once，run anywhere。

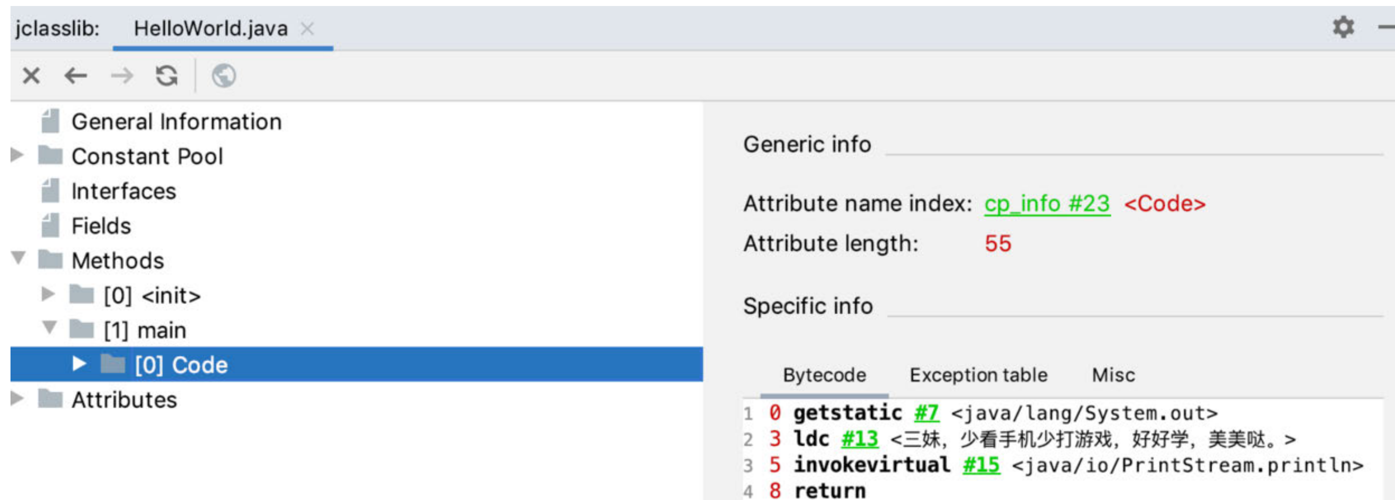
jclasslib

这里给推荐一款查看字节码文件的插件 `jclasslib`，可以在 IDEA 插件市场中安装。

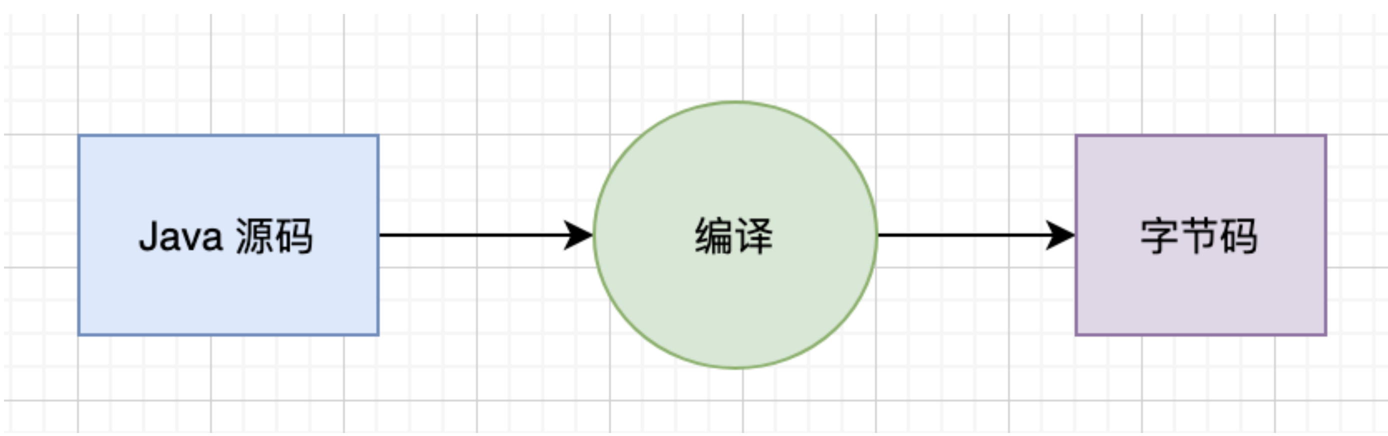
安装完成之后，点击 View -> Show Bytecode With jclasslib 即可查看字节码文件了（点击之前，光标要停留在对应的类文件上），如下图所示。



使用 jclasslib 不仅可以直观地查看类对应的字节码，还可以查看类的基本信息、常量池、接口、字段、方法等信息，如下图所示，后面也会细讲。



也就是说，**在编译阶段，Java 会将 Java 源代码文件编译为字节码文件。**



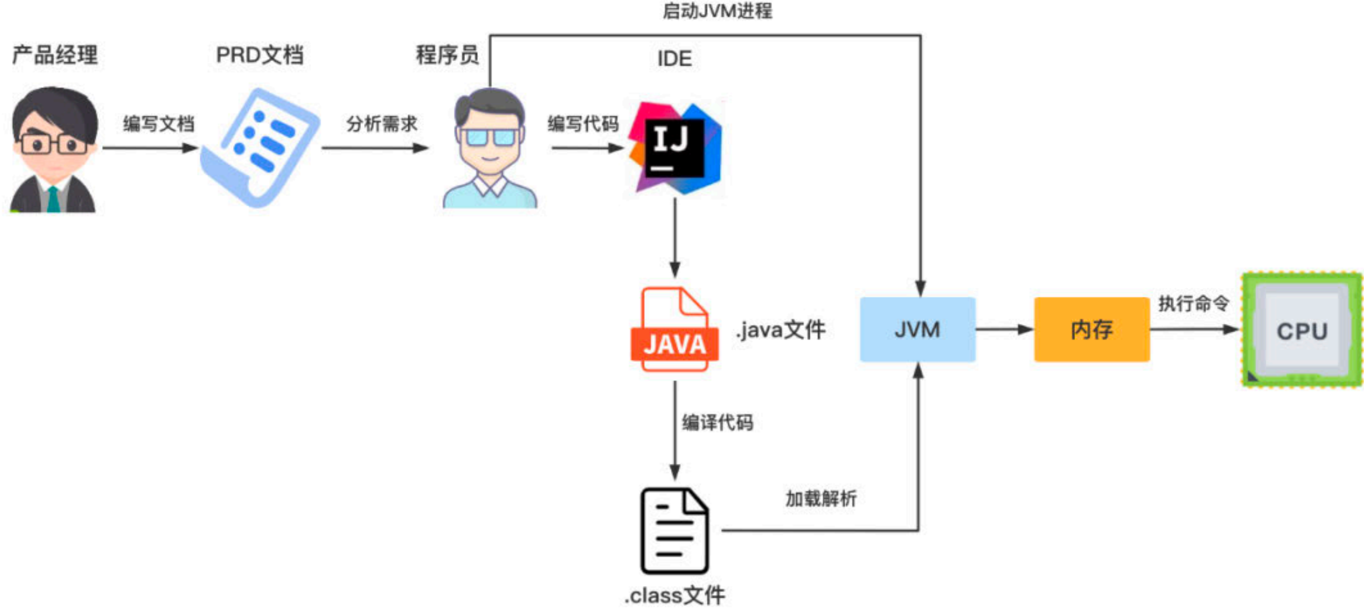
字节码文件如果用十六进制编辑器（下一节会讲到）打开的话，内容如下所示（本身是 01 串的二进制），十六进制更容易看懂（虽然肉眼也很难看得懂😂）。

```
cafe babe 0000 0034 0013 0a00 0400 0f09
0003 0010 0700 1107 0012 0100 016d 0100
0149 0100 063c 696e 6974 3e01 0003 2829
5601 0004 436f 6465 0100 0f4c 696e 654e
756d 6265 7254 6162 6c65 0100 0369 6e63
0100 0328 2949 0100 0a53 6f75 7263 6546
696c 6501 0009 4d61 696e 2e6a 6176 610c
0007 0008 0c00 0500 0601 0010 636f 6d2f
7268 7974 686d 372f 4d61 696e 0100 106a
6176 612f 6c61 6e67 2f4f 626a 6563 7400
2100 0300 0400 0000 0100 0200 0500 0600
0000 0200 0100 0700 0800 0100 0900 0000
1d00 0100 0100 0000 052a b700 01b1 0000
0001 000a 0000 0006 0001 0000 0003 0001
000b 000c 0001 0009 0000 001f 0002 0001
0000 0007 2ab4 0002 0460 ac00 0000 0100
0a00 0000 0600 0100 0000 0800 0100 0d00
0000 0200 0e
```

运行时

当有了 .class 文件也就是字节码文件之后，我们需要启动 JVM 来运行字节码文件，也就是 run 阶段，之前是 build 阶段。

JVM 会先通过类加载器加载字节码文件，然后将字节码加载到 JVM 的运行时数据区，再通过执行引擎转化为机器码最终交给操作系统执行。



我们使用 javap（后面会细讲）来看一下 HelloWorld 的字节码指令序列。

```
0 getstatic #2 <java/lang/System.out>
3 ldc #3 <Hello World>
5 invokevirtual #4 <java/io/PrintStream.println>
8 return
```

字节码指令序列通常由多条指令组成，每条指令由一个操作码和若干个操作数构成。

- 操作码：一个字节大小的指令，用于表示具体的操作。
- 操作数：跟随操作码，用于提供额外信息。

这段字节码序列的意思是调用 System.out.println 方法打印"Hello World"字符串。下面是详细的解释：

①、`0: getstatic #2 <java/lang/System.out>`：

- 操作码：getstatic
- 操作数：#2
- 描述：这条指令的作用是获取静态字段，这里获取的是 `java.lang.System` 类的 `out` 静态字段，它是一个 `PrintStream` 类型的输出流。#2 是一个指向常量池的索引，后面在讲类文件结构时会讲到。

②、`3: ldc #3 <Hello World>`：

- 操作码：ldc
- 操作数：#3
- 描述：这条指令的作用是从常量池中加载一个常量值（字符串"Hello World"）到操作数栈顶。#3 是一个指向常量池的索引，常量池里存储了字符串"Hello World"的引用。

③、`5: invokevirtual #4 <java/io/PrintStream.println>`：

- 操作码：invokevirtual
- 操作数：#4
- 描述：这条指令的作用是调用方法。这里调用的是`PrintStream`类的`println`方法，用来打印字符串。#4 是一个指向常量池的索引，常量池里存储了`java/io/PrintStream.println`方法的引用信息。

④、`8: return`：

- 操作码：return
- 描述：这条指令的作用是从当前方法返回。

上面的 getstatic、ldc、invokevirtual、return 等就是 字节码指令的操作码。

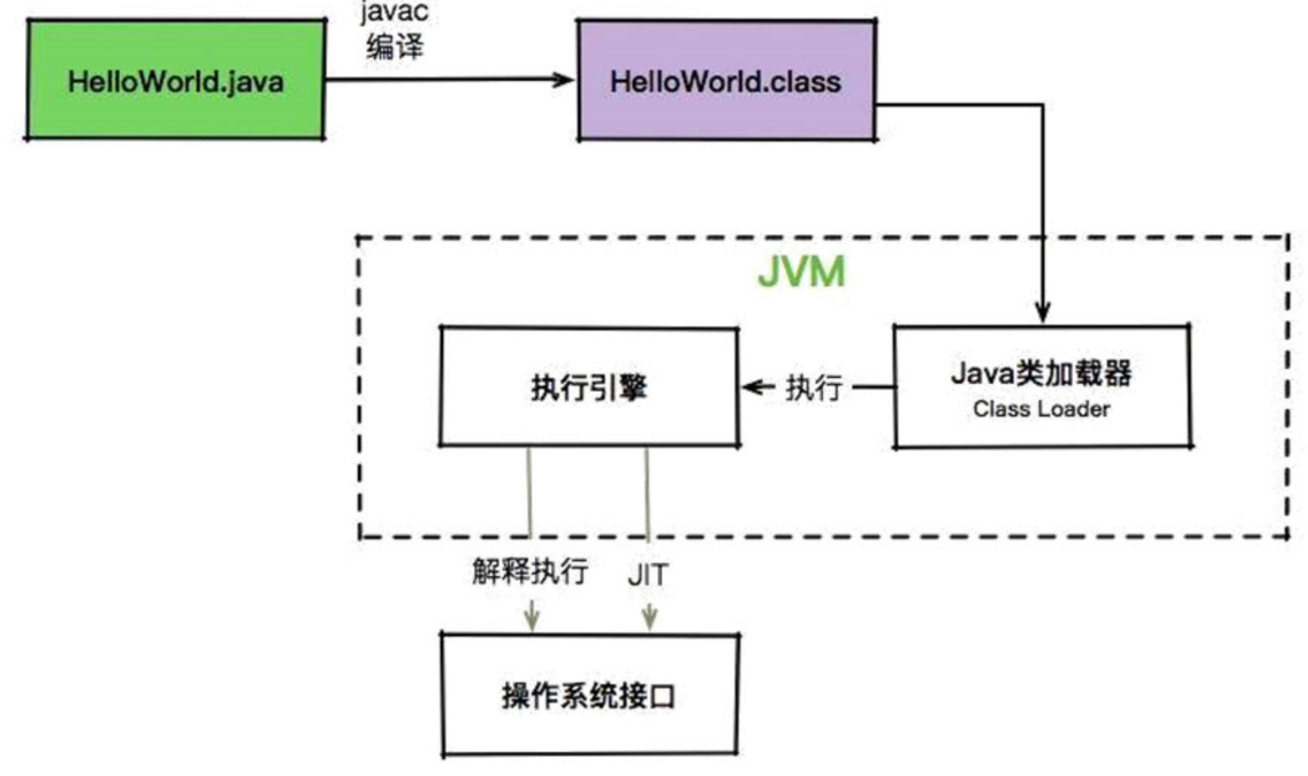
可以使用 hexdump，一个在 Unix 和 Linux 系统中常用的工具，用于以十六进制的形式显示文件的内容，看一下字节码的二进制内容。

```
b2 00 02 12 03 b6 00 04 b1
```

注意：这里是二进制文件的 16 进制表示，也就是 hex，一般分析二进制文件都是以 hex 进行分析。字节码指令和二进制之间的对应关系，以及对应的语义如下所示：

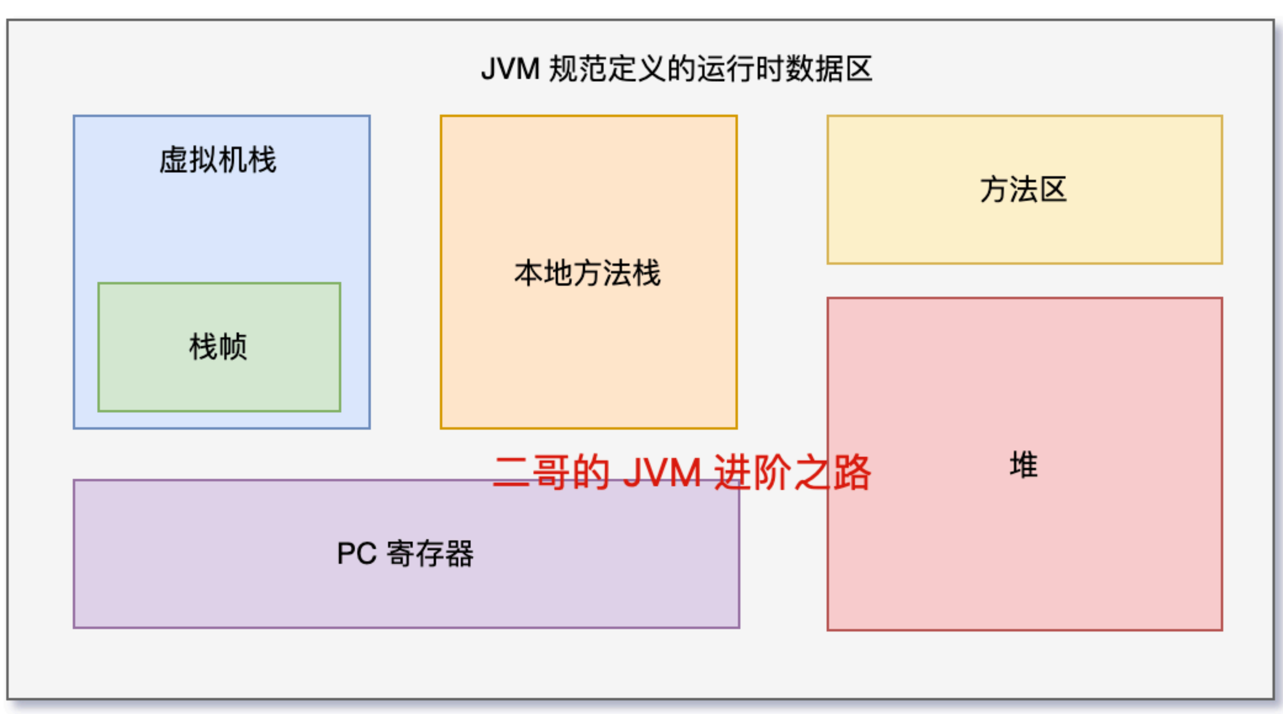
```
0xb2   getstatic       获取静态字段的值
0x12   ldc             常量池中的常量值入栈
0xb6   invokevirtual   运行时方法绑定调用方法
0xb1   return          void 方法返回
```

JVM 就是靠解析这些字节码指令来完成程序执行的。常见的执行方式有两种，一种是解释执行，对字节码逐条解释执行；一种是 JIT，也就是即时编译，它会在运行时将热点代码优化并缓存起来，下次再执行的时候直接使用缓存起来的机器码，而不需要再次解释执行。



这样就可以提高程序的执行效率。

注意，当类加载器完成字节码数据加载任务后，JVM 划分了专门的内存区域来装载这些字节码数据以及运行时中间数据。



其中 PC 寄存器、虚拟机栈以及本地方法栈属于线程私有的，堆以及元空间（方法区的实现）属于共享数据区，不同的线程共享这两部分内存数据。

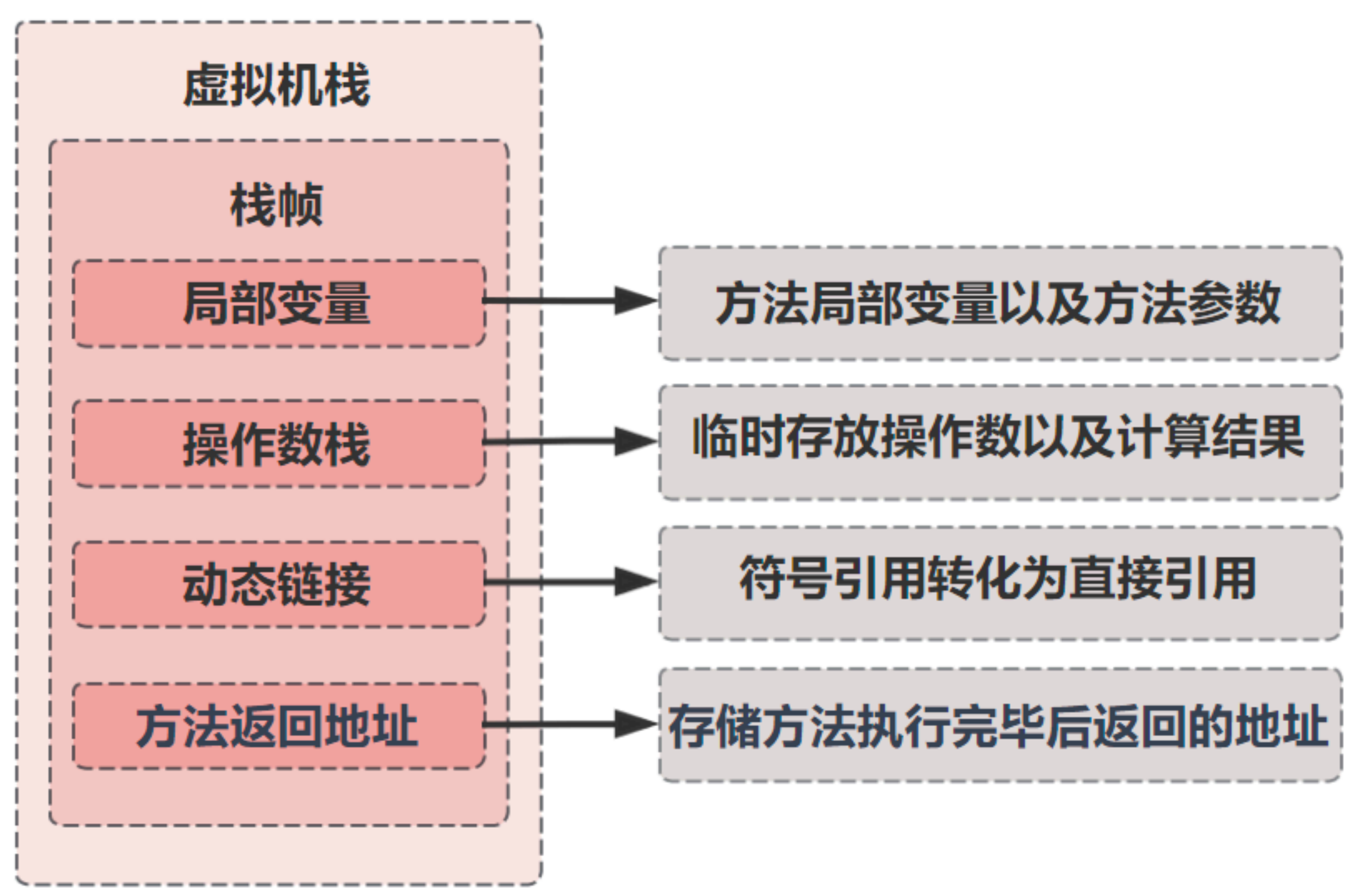
如果虚拟机中的当前线程执行的是 Java 的普通方法，那么 PC 寄存器中存储的是方法的第一条指令，当方法开始执行之后， PC 寄存器存储的是下一个字节码指令的地址。

如果虚拟机中的当前线程执行的是 native 方法，那么 PC 寄存器中的值为 undefined。

如果遇到判断分支、循环以及异常等不同的控制转移语句，PC 寄存器会被置为目标字节码指令的地址。

另外在多线程切换的时候，虚拟机会记录当前线程的 PC 寄存器，当线程切换回来的时候会根据此前记录的值恢复到 PC 寄存器中，来继续执行线程的后续的字节码指令。

除了 PC 寄存器外，字节码指令的执行流转还需要虚拟机栈的参与。我们先来看下虚拟机栈的大致结构，如下图所示。



虚拟机栈操作的基本元素就是栈帧，栈帧主要包含了局部变量表、操作数栈、动态连接以及方法返回地址。栈帧是一个先进后出的数据结构，每个方法从调用到执行完成都会对应一个栈帧在虚拟机栈中入栈和出栈。

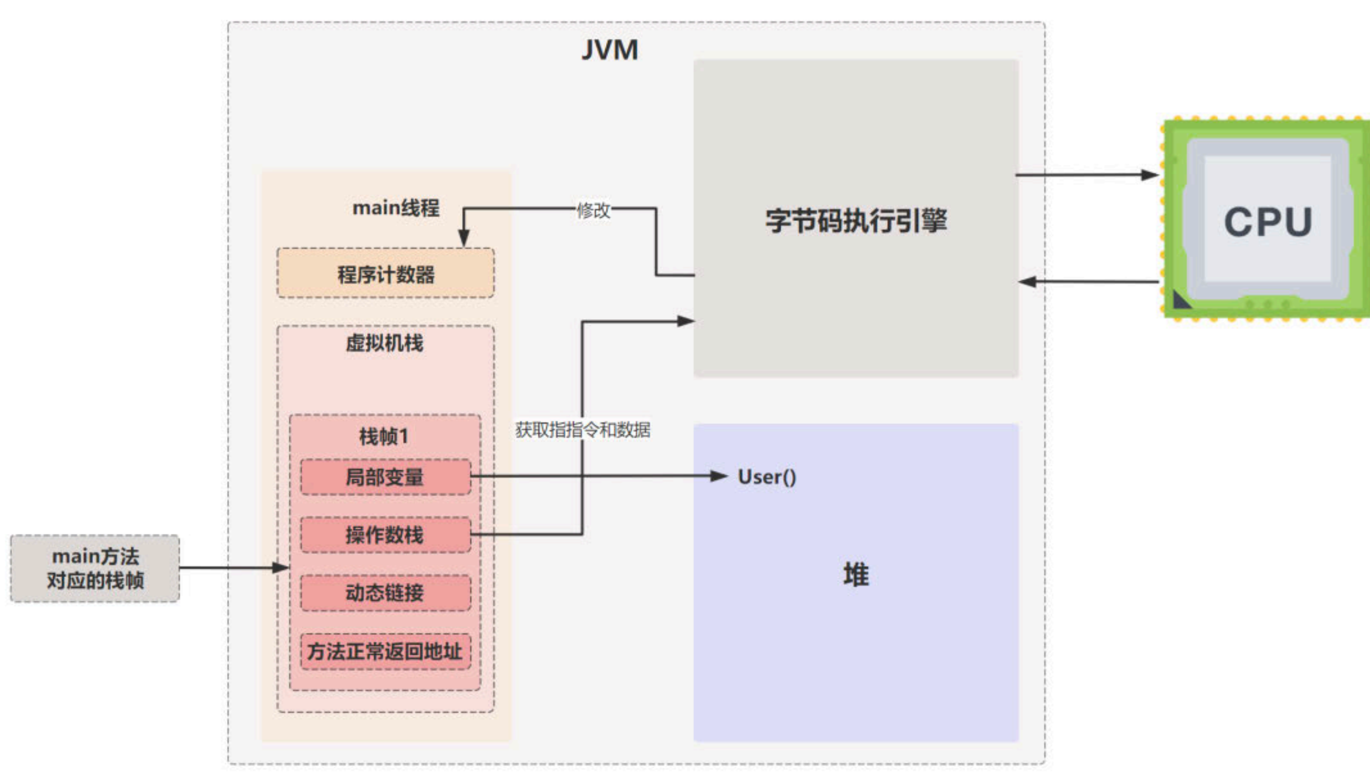
知道了虚拟机栈的结构之后，我们来看下方法执行的流转过程是怎样的。

以这段代码为例，一个 Test 类，main 方法里 new 了一个 Uesr 对象，会将 User 的 age 作为参数传递给静态方法 calculate 进行一个简单的加法操作并返回，最后打印到控制台。

```
public class Test {
    public static void main(String[] args) {
        User user  = new User();
        Integer result = calculate(user.getAge());
        System.out.println(result);
    }

    private static Integer calculate(Integer age) {
        Integer data = age + 3;
        return data;
    }

}
```

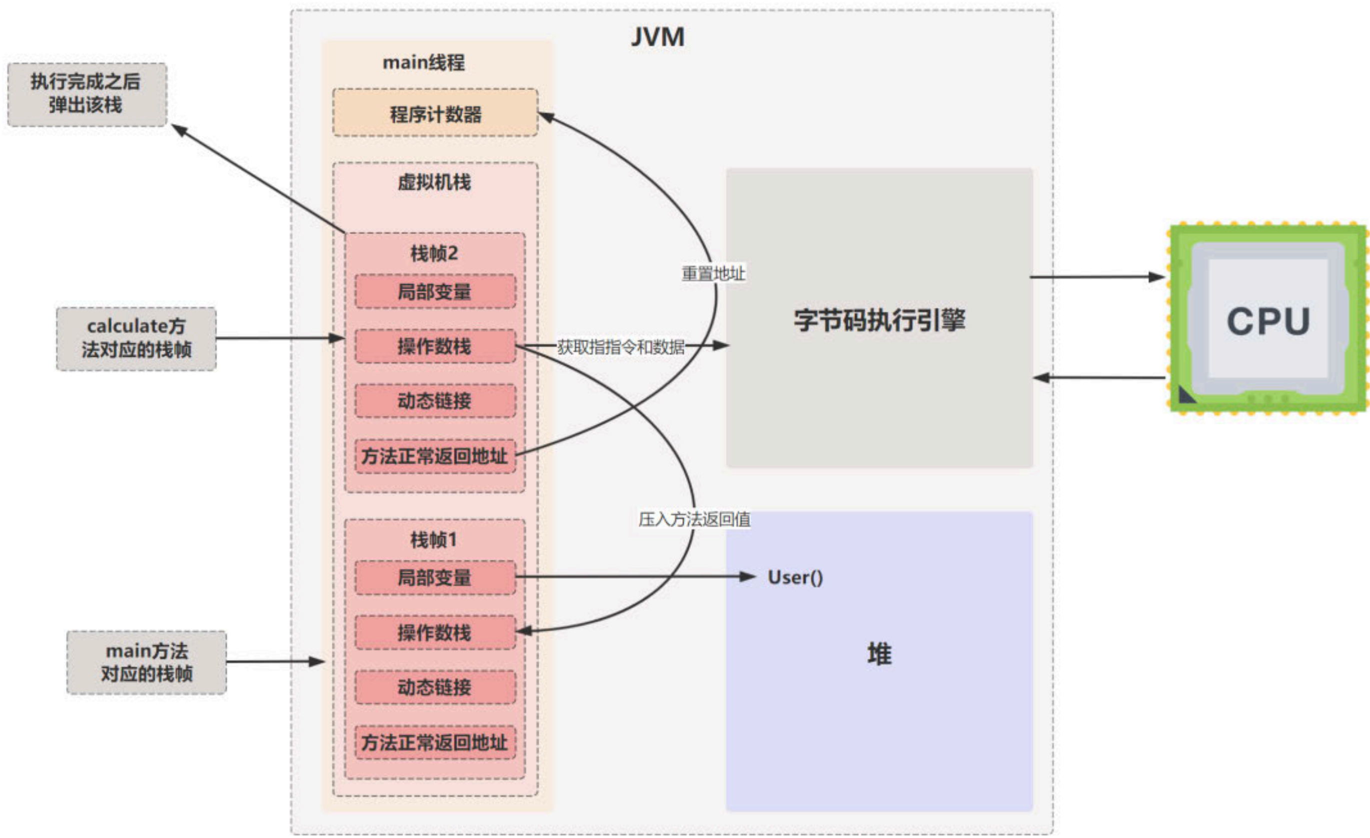
1、JVM 完成 .class 文件加载之后，会创建一个名为"main"的线程，该线程会自动调用名为"main"的静态方法，这是 Java 程序的入口点；

2、main 线程在执行 mian 方法时，JVM 会在虚拟机栈中压入 main 方法对应的栈帧；



3、栈帧的操作数栈中存储了操作的数据，JVM 执行字节码指令的时候会从操作数栈中获取数据，执行计算操作后会将结果再次压入操作数栈中；

4、当进行 calculate 方法调用的时候，虚拟机栈继续压入 calculate 方法对应的栈帧。



5、对于 age + 3 这条加法指令，在执行该指令前，JVM 会将操作数栈顶部的两个元素弹出，并将它们相加，然后将结果压入操作数栈中。

在这个例子中，指令的操作码是“add”，它表示执行加法操作；操作数 0，表示从操作数栈的顶部获取第一个操作数；操作数 1，表示从操作数栈的次顶部获取第二个操作数。

6、PC 寄存器中存储了下一条需要执行的字节码指令地址。

7、当 calculate 方法执行完成后，对应的栈帧将从虚拟机栈中弹出，方法执行的结果会被压入 main 栈帧中的操作数栈中，而方法返回地址被重置到 main 线程的 PC 寄存器中，以便于后续字节码执行引擎从 PC 寄存器中获取下一条命令的地址。

如果方法没有返回值，JVM 会将一个 null 值压入调用该方法的栈帧的操作数栈中，作为占位符，以便恢复调用方的操作数栈状态。

8、执行引擎中的解释器会从程序计数器中获取下一个字节码指令的地址，也就是元空间中对应的字节码指令，在获取到指令之后，通过解释器解释为对应的机器指令，最终由 CPU 进行执行。

小结

“好的，三妹，今天我们就先讲到这里，来简单总结下。”我长舒一口气，说到。

Java 代码首先被编译器转换为字节码，然后在 JVM 上运行。在运行时，JVM 通过解释执行或即时编译（JIT）将字节码转换为机器码。解释执行直接运行字节码，而 JIT 在运行时将热点代码编译优化为机器码以提升性能。

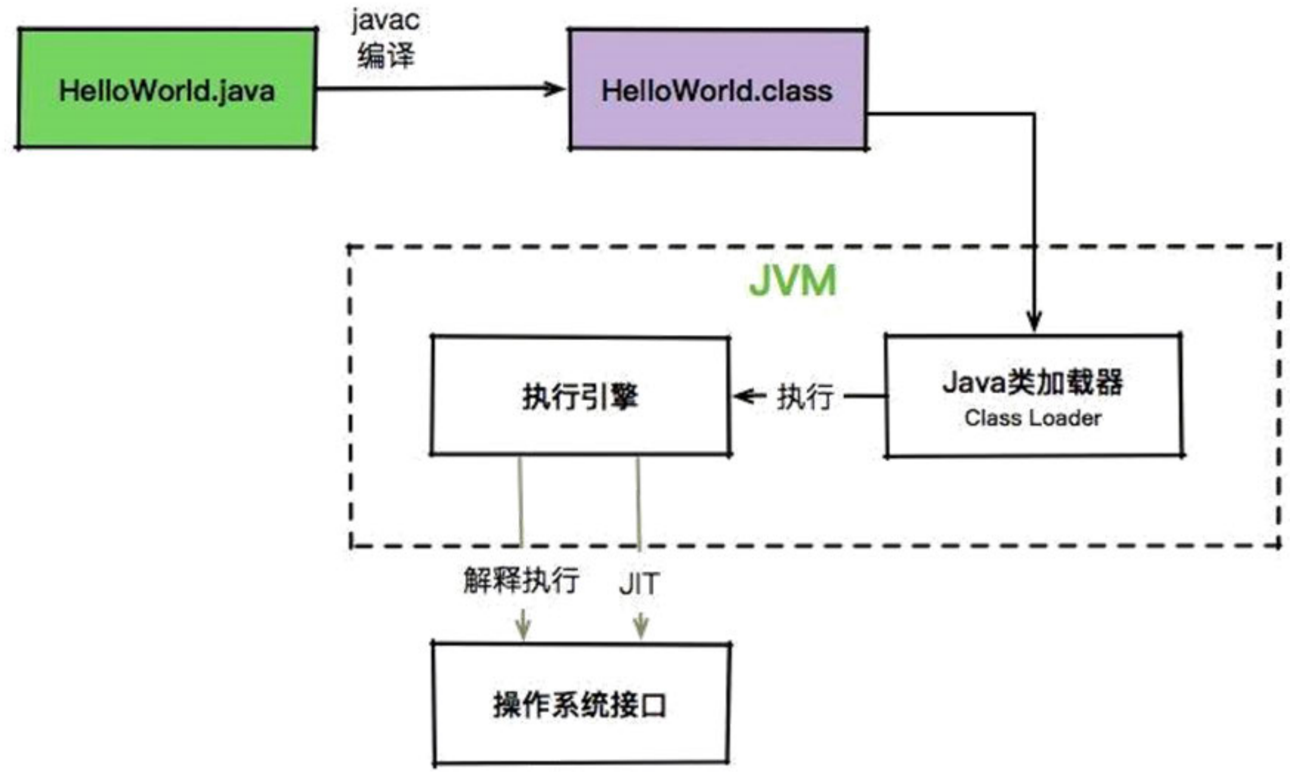
这中间需要运行时数据区来存储字节码数据以及运行时中间数据。

字节码是 JVM 中非常关键的内容，涉及到类的加载机制、字节码文件的结构、字节码指令的执行流程等等，后面我们会细讲。

> 参考链接：JVM 是如何运行 Java 程序的，作者梦尧技术，写的很不错。

第三节：Java的类加载机制

上一节在讲 JVM 运行 Java 代码的时候，我们提到，JVM 需要将编译后的字节码文件加载到其内部的运行时数据区域中进行执行。这个过程涉及到了 Java 的类加载机制（面试常问的知识点），所以我们来详细地讲一讲。



字节码我们上一节也讲过，它和类的加载机制息息相关，相信大家都还有印象。

这里再给大家普及一个小技巧，可以通过 xxd 命令来查看字节码文件，先看下面这段代码。

```
public class Test {
    public static void main(String[] args) {
        System.out.println("沉默王二");
    }
}
```

代码编译通过后，在命令行执行 `xxd Test.class`（macOS 用户可以直接执行，Windows 用户可以戳这个链接获取替代品）就可以快速查看字节码的十六进制内容。

> xxd 是一个用于在终端中创建十六进制转储（hex dump）或将十六进制转回二进制的工具。可通过维基百科了解更多信息。

```
00000000: cafe babe 0000 0034 0022 0700 0201 0019  .......4."......
00000010: 636f 6d2f 636d 6f77 6572 2f6a 6176 615f  com/cmower/java_
00000020: 6465 6d6f 2f54 6573 7407 0004 0100 106a  demo/Test......j
00000030: 6176 612f 6c61 6e67 2f4f 626a 6563 7401  ava/lang/Object.
00000040: 0006 3c69 6e69 743e 0100 0328 2956 0100  ..<init>...()V..
00000050: 0443 6f64 650a 0003 0009 0c00 0500 0601  .Code...........
00000060: 000f 4c69 6e65 4e75 6d62 6572 5461 626c  ..LineNumberTabl
```

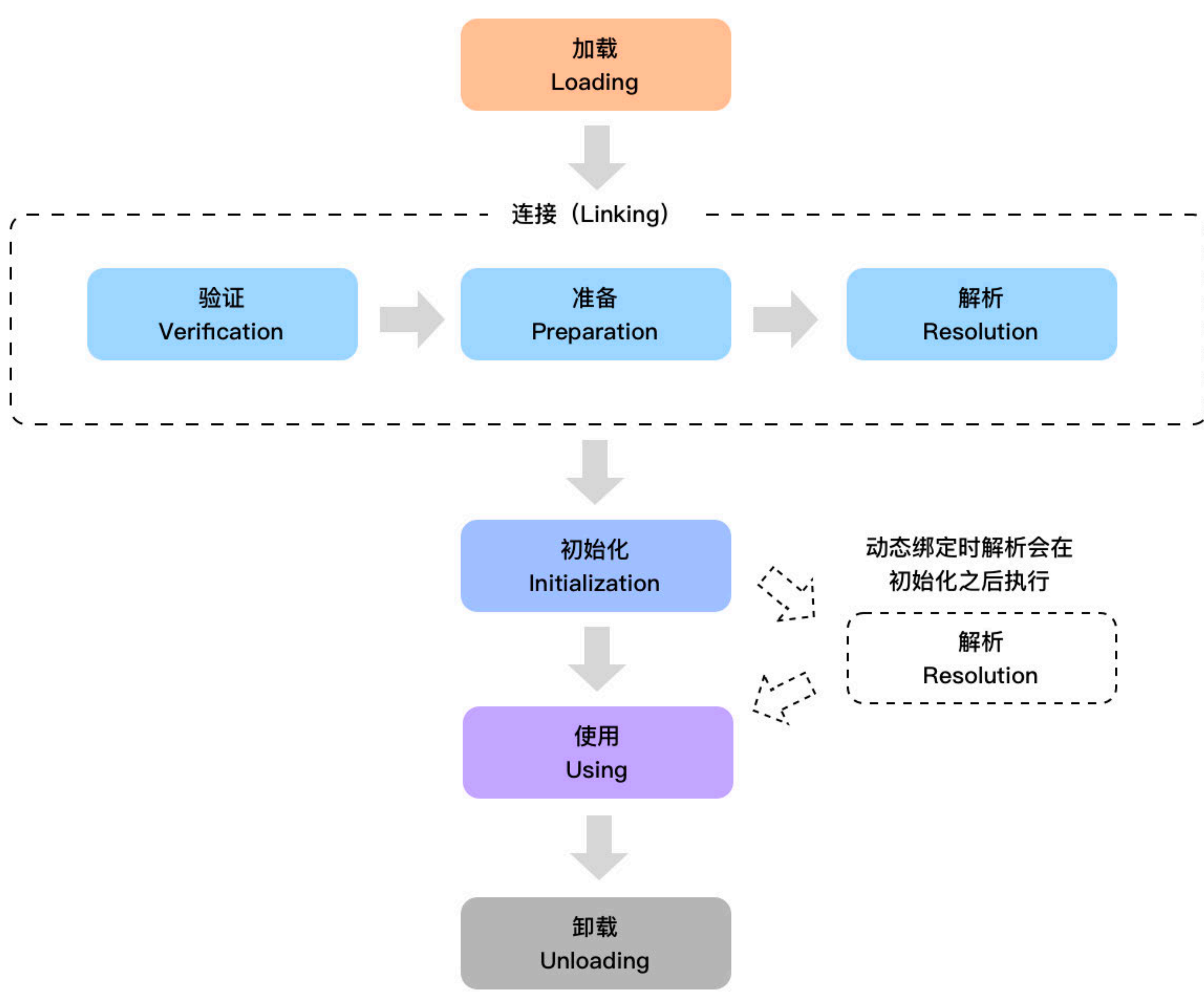
这里只说一点，这段字节码中的 `cafe babe` 被称为“魔数”，是 JVM 识别 .class 文件（字节码文件）的标志，相信大家都知道，Java 的 logo 是一杯冒着热气的咖啡，是不是又关联上了？

> 文件格式的定制者可以自由选择魔数值（只要没用过），比如说 .png 文件的魔数是 `8950 4e47`。

至于字节码文件中的其他内容，暂时先不用去管，我们后面会详细讲解。

类加载过程

知道什么是 Java 字节码后，我们来聊聊 Java 的类加载过程。



类从被加载到 JVM 开始，到卸载出内存，整个生命周期分为七个阶段，分别是加载、验证、准备、解析、初始化、使用和卸载。其中验证、准备和解析这三个阶段统称为连接。

除去使用和卸载，就是 Java 的类加载过程。这 5 个阶段一般是顺序发生的，但在动态绑定的情况下，解析阶段发生在初始化阶段之后（我们随后来解释）。

1）Loading（载入）

JVM 在该阶段的目的是将字节码从不同的数据源（可能是 class 文件、也可能是 jar 包，甚至网络）转化为二进制字节流加载到内存中，并生成一个代表该类的 `java.lang.Class` 对象（在学反射的时候有讲过）。

2）Verification（验证）

JVM 会在该阶段对二进制字节流进行校验，只有符合 JVM 字节码规范的才能被 JVM 正确执行。该阶段是保证 JVM 安全的重要屏障，下面是一些主要的检查。

- 确保二进制字节流格式符合预期（比如说是否以 `cafe bene` 开头，前面提到过）。
- 是否所有方法都遵守访问控制关键字的限定，protected、private 那些。
- 方法调用的参数个数和类型是否正确。
- 确保变量在使用之前被正确初始化了。
- 检查变量是否被赋予恰当类型的值。

- 还有更多。

3）Preparation（准备）

JVM 会在该阶段对类变量（也称为静态变量，`static` 关键字修饰的）分配内存并初始化，对应数据类型的默认初始值，如 0、0L、null、false 等。

也就是说，假如有这样一段代码：

```
public String chenmo = "沉默";
public static String wanger = "王二";
public static final String cmower = "沉默王二";
```

chenmo 不会被分配内存，而 wanger 会；但 wanger 的初始值不是“王二”而是 `null`。

需要注意的是，`static final` 修饰的变量被称作为常量，和类变量不同（这些在讲 static 关键字就讲过了）。常量一旦赋值就不会改变了，所以 cmower 在准备阶段的值为“沉默王二”而不是 `null`。

4）Resolution（解析）

该阶段将常量池中的符号引用转化为直接引用。

what？符号引用，直接引用？

符号引用以一组符号（任何形式的字面量，只要在使用时能够无歧义的定位到目标即可）来描述所引用的目标。

在编译时，Java 类并不知道所引用的类的实际地址，因此只能使用符号引用来代替。比如 `com.Wanger` 类引用了 `com.Chenmo` 类，编译时 Wanger 类并不知道 Chenmo 类的实际内存地址，因此只能使用符号 `com.Chenmo`。

直接引用通过对符号引用进行解析，找到引用的实际内存地址。我们再来对比说明一下。

符号引用

- **定义**：包含了类、字段、方法、接口等多种符号的全限定名。
- **特点**：在编译时生成，存储在编译后的字节码文件的常量池中。
- **独立性**：不依赖于具体的内存地址，提供了更好的灵活性。

直接引用

- **定义**：直接指向目标的指针、相对偏移量或者能间接定位到目标的句柄。
- **特点**：在运行时生成，依赖于具体的内存布局。
- **效率**：由于直接指向了内存地址或者偏移量，所以通过直接引用访问对象的效率较高。

下面通过一张简化的图来描述它们的区别：

```
+---------------+     +----------------------+     +---------------+
| Java Source   |     | .class File          |     | JVM Runtime   |
| Code          |     | (Constant Pool)      |     | Memory        |
+---------------+     +----------------------+     +---------------+
| class A {     |     | #1 = Class #2        |     | +-----------+ |
|   B b;        |     | #2 = Utf8    B       |     | |  class B    | |
|   ...         |     | ...                  |     | +-----------+ |
| }             |     +----------------------+     +---------------+
|               |            |      ^                       |
| class B {     |            |      |                       |
|   ...         |            |      |                       |
| }             |            |      |                       |
+---------------+            |      |                       |
                             |      |                       |
                             |      |                       |
                             |      |                       |
                             +-------+                      |
                             符号引用：在编译时确定，存储在常量池中    |
                                 |                          |
                                 | 解析                      |
                                 |                          |
                                 v                          |
                             +---------------+              |
                             |    Direct     |              |
                             |    Reference  |              |
                             +---------------+              |
                             指向运行时内存中的 class B 对象或其元数据
```

在上面的例子中：

- `class A` 引用了 `class B`。
- 在编译时，这个引用变成了符号引用，存储在 `.class` 文件的常量池中。
- 在运行时，当 `class A` 需要使用 `class B` 的时候，JVM 会将符号引用解析为直接引用，指向内存中的 `class B` 对象或其元数据。

通过这种方式，Java 程序能够在编译时和运行时具有更高的灵活性和解耦性，同时在运行时也能获得更好的性能。

Java 本身是一个静态语言，但后面又加入了动态加载特性，因此我们理解解析阶段需要从这两方面来考虑。

如果不涉及动态加载，那么一个符号的解析结果是可以缓存的，这样可以避免多次解析同一个符号，因为第一次解析成功后面多次解析也必然成功，第一次解析异常后面重新解析也会是同样的结果。

如果使用了动态加载，前面使用动态加载解析过的符号后面重新解析结果可能会不同。使用动态加载时解析过程发生在在程序执行到这条指令的时候，这就是为什么前面讲的动态加载时解析会在初始化后执行。

整个解析阶段主要做了下面几个工作：

- 类或接口的解析
- 类方法解析
- 接口方法解析
- 字段解析

5）Initialization（初始化）

该阶段是类加载过程的最后一步。在准备阶段，类变量已经被赋过默认初始值，而在初始化阶段，类变量将被赋值为代码期望赋的值。换句话说，初始化阶段是执行类构造器方法（javap 中看到的 `<clinit>()` 方法）的过程。

上面这段话可能说得很抽象，不好理解，我来举个例子。

```
String cmower = new String("沉默王二");
```

上面这段代码使用了 `new` 关键字来实例化一个字符串对象，那么这时候，就会调用 String 类的构造方法对 cmower 进行实例化。

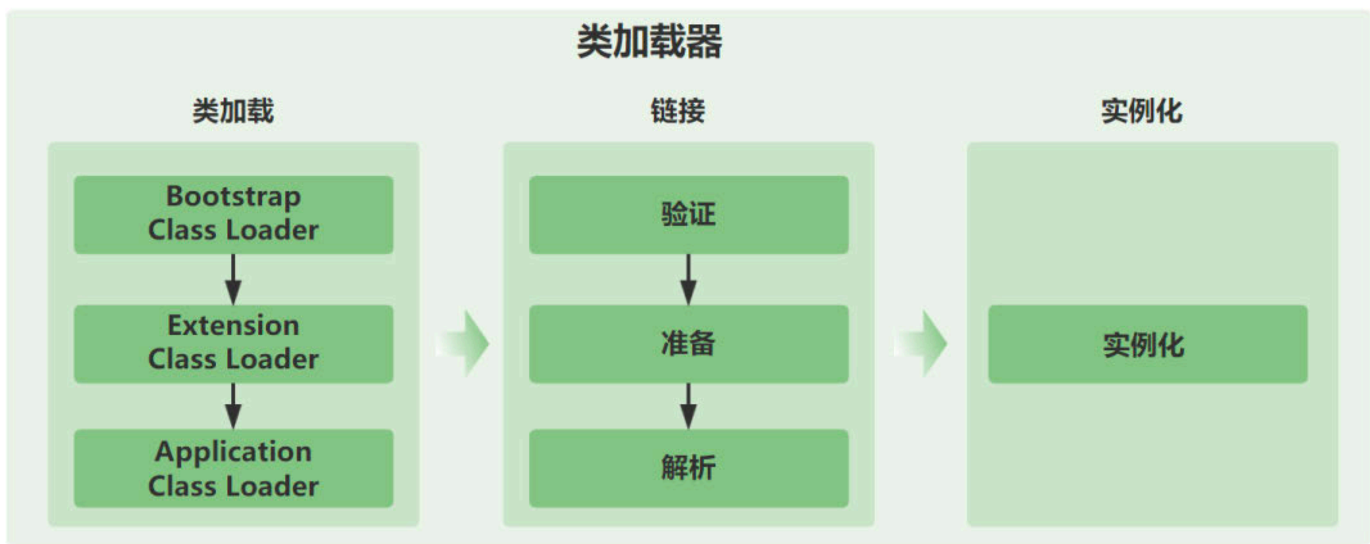
```
public String(String original) {
    this.value = original.value;
    this.hash = original.hash;
}
```

初始化时机包括以下这些：

- 创建类的实例时。
- 访问类的静态方法或静态字段时（除了 final 常量，它们在编译期就已经放入常量池）。
- 使用 java.lang.reflect 包的方法对类进行反射调用时。
- 初始化一个类的子类（首先会初始化父类）。
- JVM 启动时，用户指定的主类（包含 main 方法的类）将被初始化。

类加载器

聊完类加载过程，就不得不聊聊类加载器。



一般来说，Java 程序员并不需要直接同类加载器进行交互。JVM 默认的行为就已经足够满足大多数情况的需求了。不过，如果遇到了需要和类加载器进行交互的情况，而对类加载器的机制又不是很了解的话，就不得不花大量的时间去调试
`ClassNotFoundException` 和 `NoClassDefFoundError` 等异常（前面讲过）。

对于任意一个类，都需要由它的类加载器和这个类本身一同确定其在 JVM 中的唯一性。也就是说，如果两个类的加载器不同，即使两个类来源于同一个字节码文件，那这两个类就必定不相等（比如两个类的 Class 对象不 `equals`）。

来通过一段简单的代码了解下。

```
/**
 * @author 微信搜「沉默王二」，回复关键字 PDF
 */
public class Test {
    public static void main(String[] args) {
        ClassLoader loader = Test.class.getClassLoader();
        while (loader != null) {
            System.out.println(loader);
            loader = loader.getParent();
        }
    }
}
```

每个 Java 类都维护着一个指向定义它的类加载器的引用，通过 `类名.class.getClassLoader()` 可以获取到此引用；然后通过 `loader.getParent()` 可以获取类加载器的上层类加载器。

上面这段代码的输出结果如下：

```
jdk.internal.loader.ClassLoaders$AppClassLoader@512ddf17
jdk.internal.loader.ClassLoaders$PlatformClassLoader@2d209079
```

第一行输出为 Test 的类加载器，即应用类加载器，它是 `jdk.internal.loader.ClassLoaders$AppClassLoader` 类的实例；第二行输出为平台类加载器，是 `jdk.internal.loader.ClassLoaders$PlatformClassLoader` 类的实例。那启动类加载器呢？

按理说，扩展类加载器的上层类加载器是启动类加载器，但启动类加载器是虚拟机的内置类加载器，通常表示为 null。

也就是说，类加载器可以分为四种类型：

①、引导类加载器（Bootstrap ClassLoader）：负责加载 JVM 基础核心类库，如 rt.jar、sun.boot.class.path 路径下的类。

②、扩展类加载器（Extension ClassLoader）：负责加载 Java 扩展库中的类，例如 jre/lib/ext 目录下的类或由系统属性 java.ext.dirs 指定位置的类。

③、系统（应用）类加载器（System ClassLoader）：负责加载系统类路径 java.class.path 上指定的类库，通常是你的应用类和第三方库。

④、用户自定义类加载器：Java 允许用户创建自己的类加载器，通过继承 java.lang.ClassLoader 类的方式实现。这在需要动态加载资源、实现模块化框架或者特殊的类加载策略时非常有用。

```
import java.io.*;

public class CustomClassLoader extends ClassLoader {

    private String pathToBin;

    public CustomClassLoader(String pathToBin) {
        this.pathToBin = pathToBin;
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        try {
            byte[] classData = loadClassData(name);
            return defineClass(name, classData, 0, classData.length);
        } catch (IOException e) {
            throw new ClassNotFoundException("Class " + name + " not found", e);
        }
    }

    private byte[] loadClassData(String name) throws IOException {
        String file = pathToBin + name.replace('.', File.separatorChar) + ".class";
        InputStream is = new FileInputStream(file);
        ByteArrayOutputStream byteSt = new ByteArrayOutputStream();
        int len = 0;
        while ((len = is.read()) != -1) {
            byteSt.write(len);
        }
        return byteSt.toByteArray();
    }
```

```
}
```

这个自定义类加载器做了以下几件事情：

- 构造器：接受一个字符串参数，这个字符串指定了类文件的存放路径。
- 覆写 findClass 方法：当父类加载器无法加载类时，findClass 方法会被调用。在这个方法中，首先使用 loadClassData 方法读取类文件的字节码，然后调用 defineClass 方法来将这些字节码转换为 Class 对象。
- loadClassData 方法：读取指定路径下的类文件内容，并将内容作为字节数组返回。

双亲委派模型

双亲委派模型（Parent Delegation Model）是 Java 类加载器使用的一种机制，用于确保 Java 程序的稳定性和安全性。在这个模型中，类加载器在尝试加载一个类时，首先会委派给其父加载器去尝试加载这个类，只有在父加载器无法加载该类时，子加载器才会尝试自己去加载。

1. **委派给父加载器**：当一个类加载器接收到类加载的请求时，它首先不会尝试自己去加载这个类，而是将这个请求委派给它的父加载器。
2. **递归委派**：这个过程会递归向上进行，从启动类加载器（Bootstrap ClassLoader）开始，再到扩展类加载器（Extension ClassLoader），最后到系统类加载器（System ClassLoader）。
3. **加载类**：如果父加载器可以加载这个类，那么就使用父加载器的结果。如果父加载器无法加载这个类（它没有找到这个类），子加载器才会尝试自己去加载。
4. **安全性和避免重复加载**：这种机制可以确保不会重复加载类，并保护 Java 核心 API 的类不被恶意替换。

类加载器的层级结构如下图所示：

```
Bootstrap ClassLoader
         ↑
         |
Extension ClassLoader
         ↑
         |
System/Application ClassLoader
         ↑
         |
Custom ClassLoader
```

这种层次关系被称作为**双亲委派模型**：如果一个类加载器收到了加载类的请求，它会先把请求委托给上层加载器去完成，上层加载器又会委托上上层加载器，一直到最顶层的类加载器；如果上层加载器无法完成类的加载工作时，当前类加载器才会尝试自己去加载这个类。

PS：双亲委派模型突然让我联想到朱元璋同志，这个同志当上了皇帝之后连宰相都不要了，所有的事情都亲力亲为，只有自己没精力没时间做的事才交给大臣们去干。

使用双亲委派模型有一个很明显的好处，那就是 Java 类随着它的类加载器一起具备了一种带有优先级的层次关系，这对于保证 Java 程序的稳定运作很重要。

上文中曾提到，如果两个类的加载器不同，即使两个类来源于同一个字节码文件，那这两个类就必定不相等——双亲委派模型能够保证同一个类最终会被特定的类加载器加载。

小结

Java 的类加载机制通过类加载器和类加载过程的合作，确保了 Java 程序的动态加载、灵活性和安全性。双亲委派模型进一步增强了这种机制的安全性和类之间的协调性。

学习就是这样，只要你敢于挑战自己，就能收获知识——就像山就在那里，只要你肯攀登，就能到达山顶。

> 参考链接：详解 Java 类加载过程

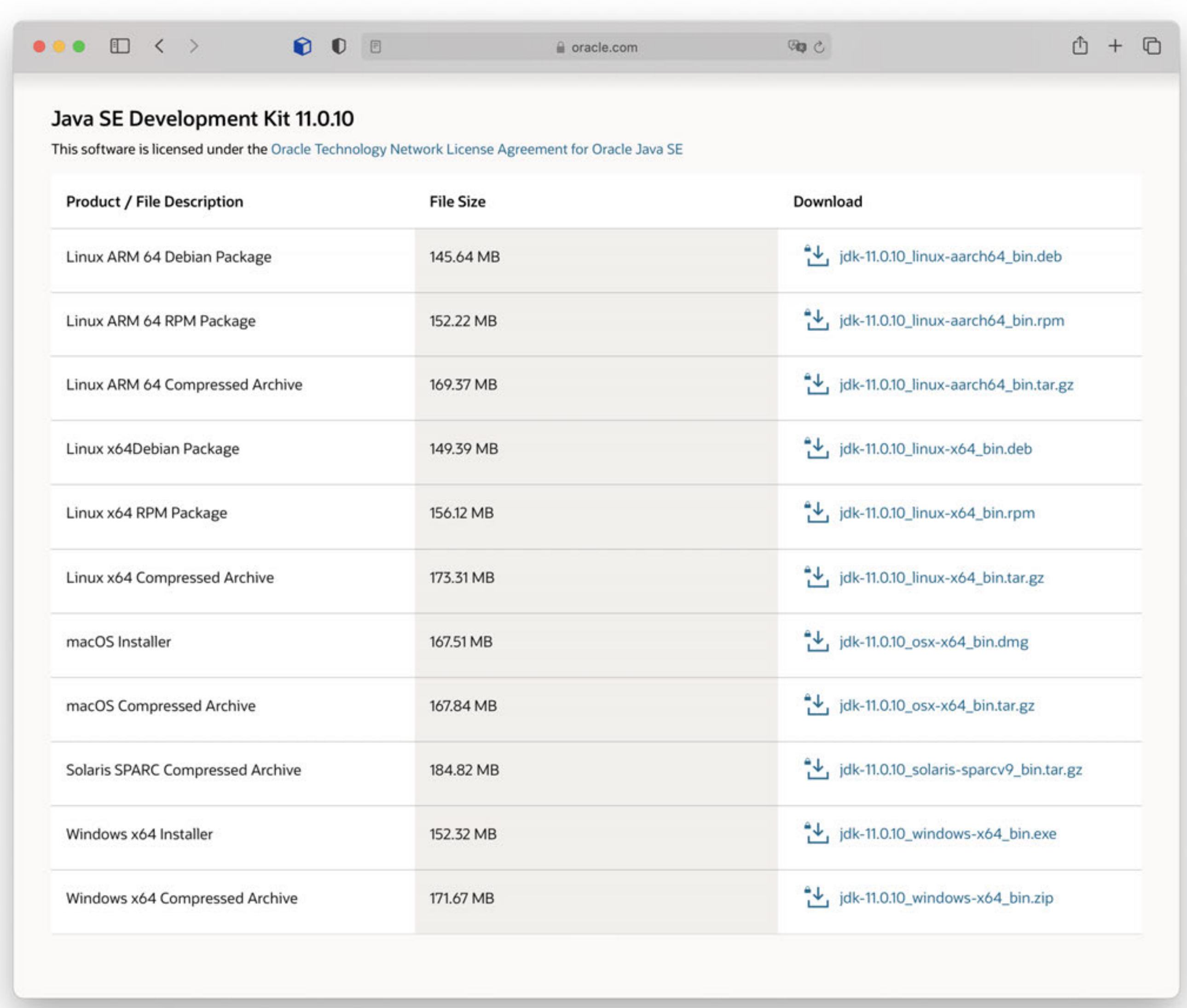
第四节：Java类文件结构

大家好，我是二哥呀，今天我拿了一把小刀，准备带大家解剖一下 Java 的类文件结构，也就是 .class 文件的内容结构，虽然它实际上是一串连续的二进制，由 0 和 1 组成，但我们仍然可以借助一些工具来看清楚它的真面目。

> 类文件结构=.class文件的结构=Class文件结构，这三个说法都是一个意思，.class是从文件后缀名的角度来说的，Class是从Java类的角度来说的，类文件结构就是 Class 的中文译名。

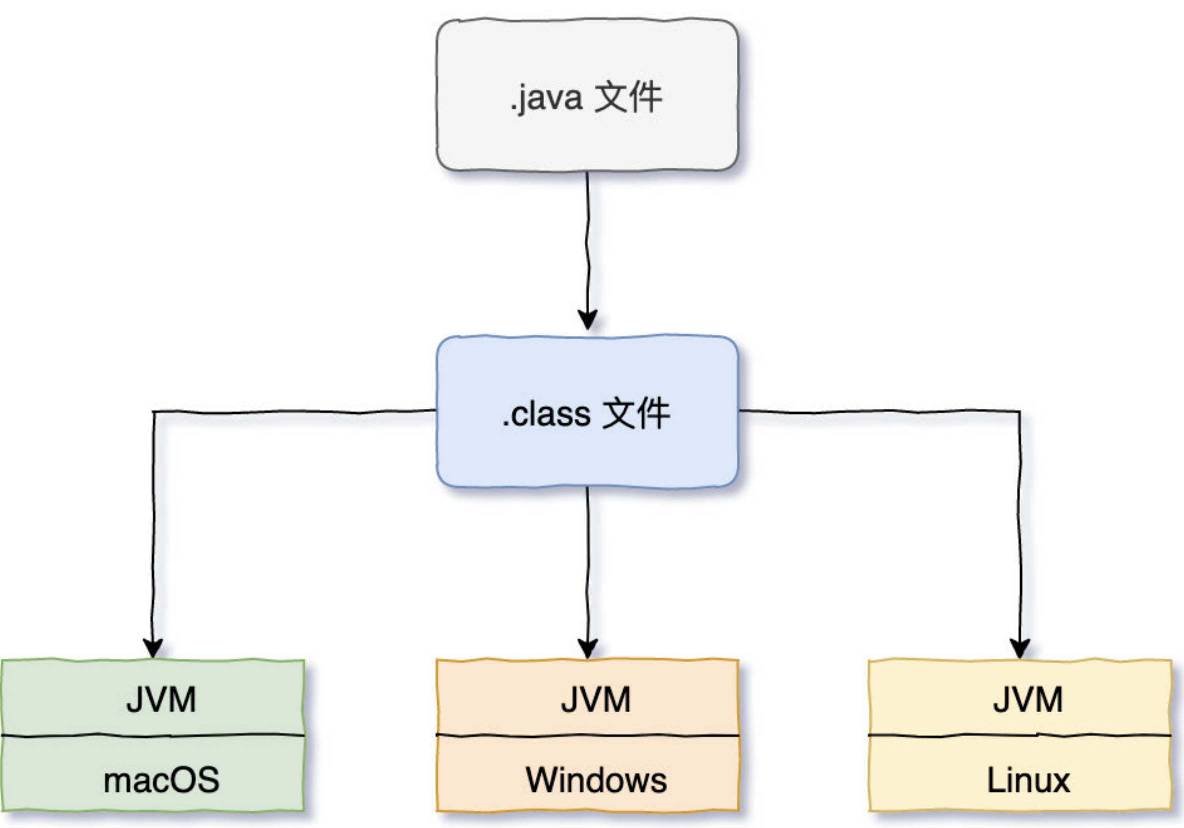
---这部分内容前面其实已经讲过，但为了保持这篇内容的完整性，就暂时保留了下来，已经掌握的同学可以略过start----

计算机的世界里流传着这么一句话，“**计算机科学领域的任何问题都可以通过增加一个中间层来解决**”。对于 Java 来说，JVM 就是这么一个产物，“Write once, Run anywhere”之所以能实现，靠得就是 JVM，它能在不同的操作系统下运行同一份源代码编译后的 class 文件。



Product / File Description	File Size	Download
Linux ARM 64 Debian Package	145.64 MB	jdk-11.0.10_linux-aarch64_bin.deb
Linux ARM 64 RPM Package	152.22 MB	jdk-11.0.10_linux-aarch64_bin.rpm
Linux ARM 64 Compressed Archive	169.37 MB	jdk-11.0.10_linux-aarch64_bin.tar.gz
Linux x64Debian Package	149.39 MB	jdk-11.0.10_linux-x64_bin.deb
Linux x64 RPM Package	156.12 MB	jdk-11.0.10_linux-x64_bin.rpm
Linux x64 Compressed Archive	173.31 MB	jdk-11.0.10_linux-x64_bin.tar.gz
macOS Installer	167.51 MB	jdk-11.0.10_osx-x64_bin.dmg
macOS Compressed Archive	167.84 MB	jdk-11.0.10_osx-x64_bin.tar.gz
Solaris SPARC Compressed Archive	184.82 MB	jdk-11.0.10_solaris-sparcv9_bin.tar.gz
Windows x64 Installer	152.32 MB	jdk-11.0.10_windows-x64_bin.exe
Windows x64 Compressed Archive	171.67 MB	jdk-11.0.10_windows-x64_bin.zip

Java 是跨平台的，JVM 作为中间层，自然要针对不同的操作系统提供不同的实现。拿 JDK 11 来说，它的实现就有上图中提到的这么多种（目前最新版本已经是 JDK 21 了）。



通过不同操作系统的 JVM，我们的源代码就可以不用根据不同的操作系统编译成不同的二进制可执行文件了，跨平台的目标也就实现了。

那这个 class 文件到底是什么玩意呢？它是怎么被 JVM 识别的呢？

我们用 IDEA 编写一段简单的 Java 代码，文件名为 Hello.java。

```
package com.itwanger.jvm;
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello!");
    }
}
```

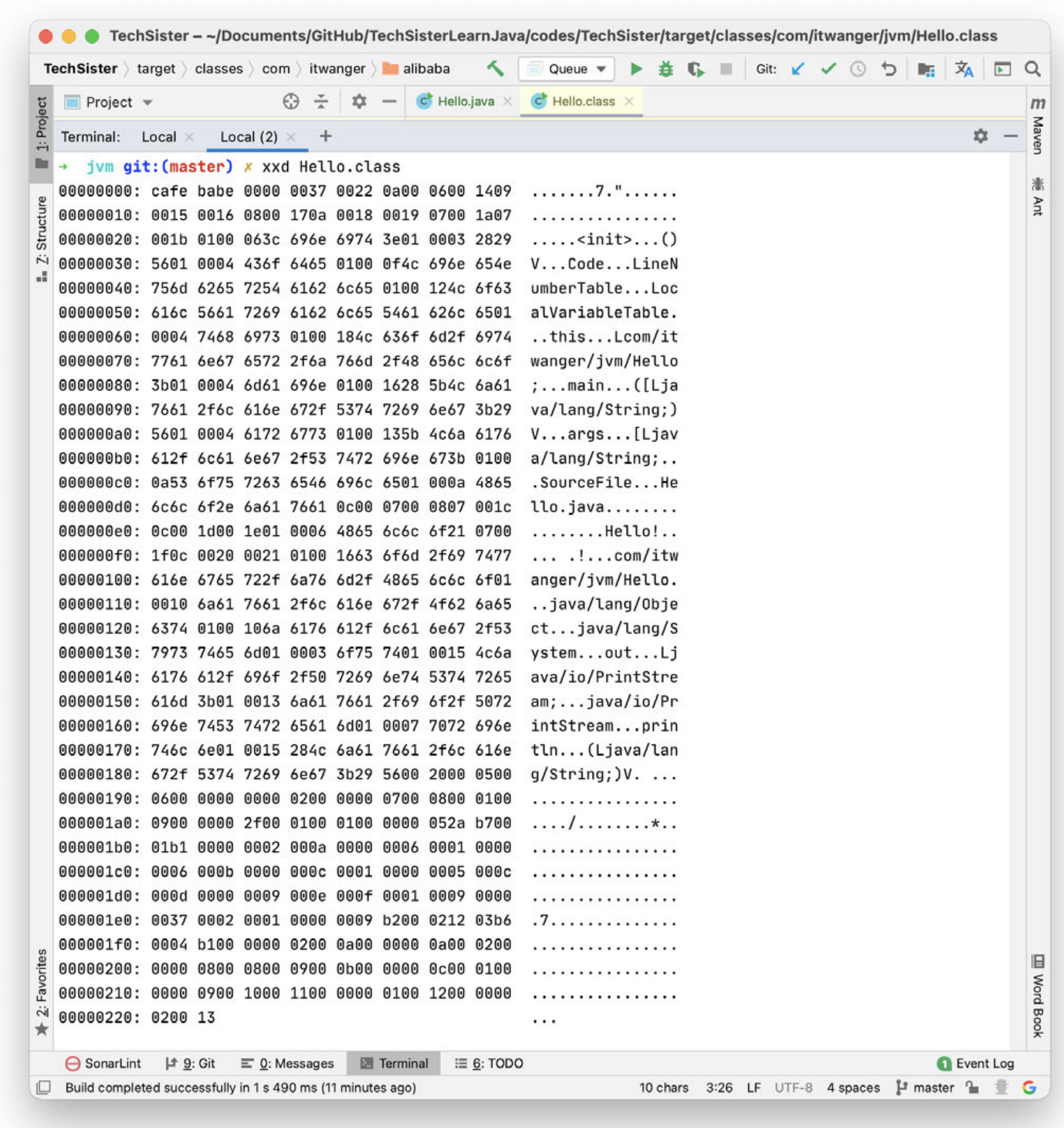
点击编译按钮后（也不用主动点，IDEA 会自动编译），IDEA 会帮我们生成一个名为 Hello.class 的文件，在 `target/classes` 的对应包目录下。直接双击打开后长下面这样子：

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by Fernflower decompiler)
//
```

```java
package com.itwanger.jvm;

class Hello {
    Hello() {
    }

    public static void main(String[] args) {
        System.out.println("Hello!");
    }
}
```

看起来和源代码很像，只是多了一个空的构造方法，对吧？它是 class 文件被 IDEA 自带的反编译工具 Fernflower 反编译后的样子。那真实的 class 文件长什么样子呢？

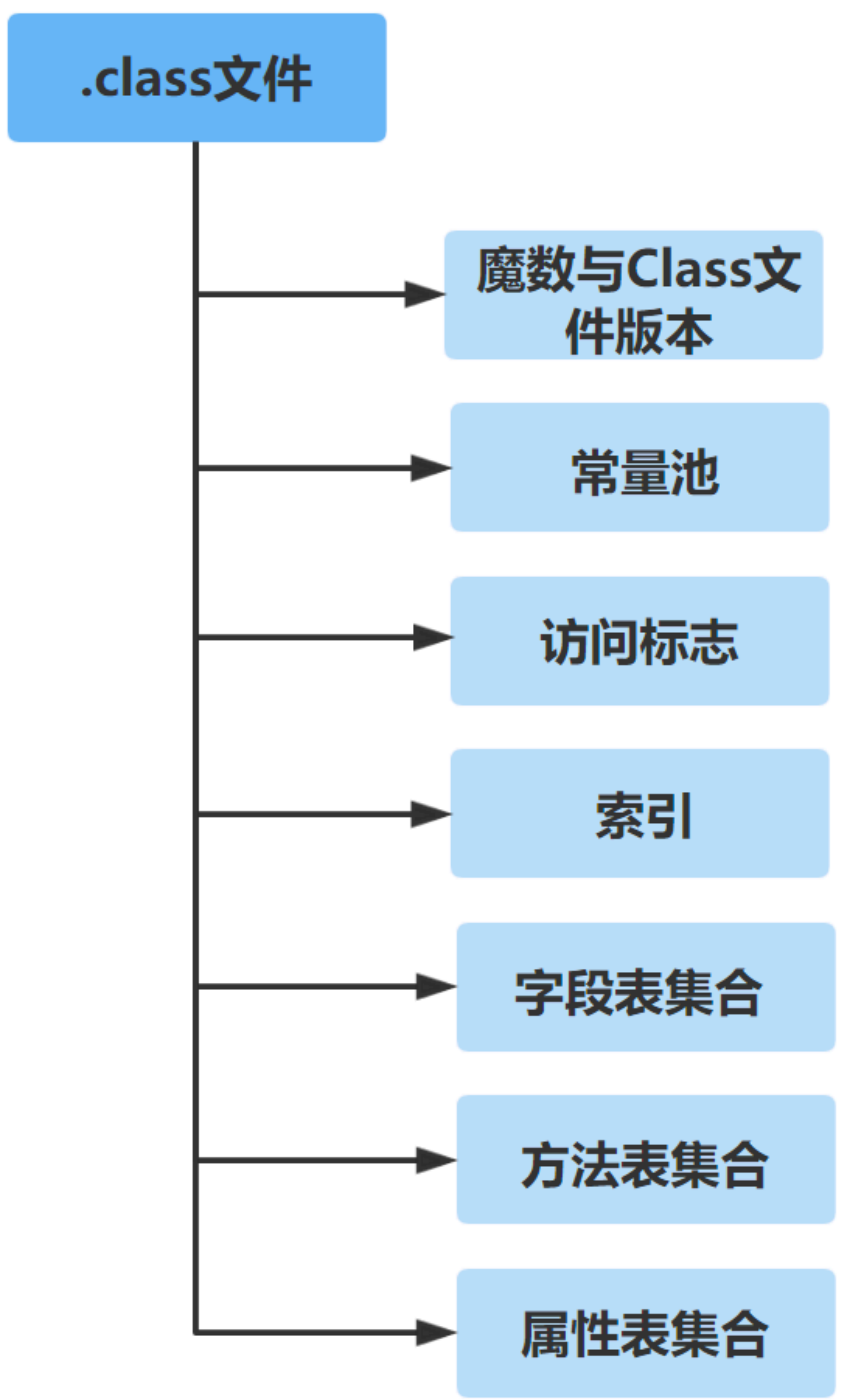
可以在终端中通过 `xxd Hello.class` 命令来查看（前面我们已经讲过了，大家可以戳这个链接回看）。



这就是 class 文件的十六进制形式。

---这部分内容前面其实已经讲过，但为了保持这篇内容的完整性，就暂时保留了下来，已经掌握的同学可以略过end----

类文件的内容通常可以分为下面这几部分，见下图。


.class文件
魔数与Class文件版本
常量池
访问标志
索引
字段表集合
方法表集合
属性表集合

01、魔数

回看 class 文件的十六进制形式截图。

第一行中有一串特殊的字符 `cafebabe`，它就是一个魔数，是 JVM 识别 class 文件的标志，JVM 会在验证阶段检查 class 文件是否以该魔数开头，如果不是则会抛出 `ClassFormatError`。

魔数 `cafebabe` 的中文意思显而易见，咖啡宝贝，再加上 Java 的图标本来就是一个热气腾腾的咖啡，可见 Java 与咖啡的渊源有多深。

02、版本号

紧跟着魔数后面的四个字节 `0000 0037` 分别表示副版本号和主版本号。也就是说，主版本号为 55（0x37 的十进制），也就是 Java 11 对应的版本号，副版本号为 0。

上一个 LTS 版本是 Java 8，对应的主版本号为 52，也就是说 Java 9 是 53，Java 10 是 54，只不过 Java 9 和 Java 10 都是过渡版本，下一个 LTS 版本是 Java 17，预计 2021 年 9 月份推出（从这里大家可以推断出这篇内容的初稿时间，哈哈哈）。

那现在是 2023年12月14日，Java 21 已经发布了。通过上面的方法，大家可以查看一下 Java 21 对应的版本号是多少，这个小作业就留给大家了，动动手，你会发现不一样的世界。

03、常量池

紧跟在版本号之后的是常量池，它包含了类、接口、字段和方法的符号引用，以及字符串字面量和数值常量。这些信息在编译时被创建，并在运行时被Java虚拟机（JVM）使用。

相当于一个资源仓库，主要存放量大类型常量：

- 字面量（Literals）：字面量是不变的数据，主要包括数值（如整数、浮点数）和字符串字面量。例如，一个整数100或一个字符串"Hello World"，在源代码中直接赋值，编译后存储在常量池中。
- 符号引用（Symbolic References）：符号引用是对类、接口、字段、方法等的引用，它们不是由字面量值给出的，而是通过符号名称（如类名、方法名）和其他额外信息（如类型、签名）来表示。这些引用在类文件中以一种抽象的方式存在，它们在类加载时被虚拟机解析为具体的内存地址。

（这部分内容我们前面讲过，戳链接回顾一下）

好，接下来，我们通过实际的代码示例来看一下常量池到底是什么。

Java 定义了 boolean、byte、short、char 和 int 等基本数据类型，它们在常量池中都会被当做 int 来处理。我们来通过一段简单的 Java 代码了解下。

```
public class ConstantTest {
    public final boolean bool = true;
    public final char aChar = 'a';
    public final byte b = 66;
    public final short s = 67;
    public final int i = 68;
}
```

布尔值 true 的十六进制是 0x01、字符 a 的十六进制是 0x61，字节 66 的十六进制是 0x42，短整型 67 的十六进制是 0x43，整型 68 的十六进制是 0x44。所以编译生成的整型常量在 class 文件中的位置如下图所示。

```
$ xxd -g 1 ConstantTest.class
00000000: ca fe ba be 00 00 00 37 00 2a 0a 00 08 00 22 09  .......7.*....".
00000010: 00 07 00 23 09 00 07 00 24 09 00 07 00 25 09 00  ...#....$....%..
00000020: 07 00 26 09 00 07 00 27 07 00 28 07 00 29 01 00  ..&....'..(..)..
00000030: 04 62 6f 6f 6c 01 00 01 5a 01 00 0d 43 6f 6e 73  .bool...Z...Cons
00000040: 74 61 6e 74 56 61 6c 75 65 03 00 00 00 01 01 00  tantValue.......
00000050: 05 61 43 68 61 72 01 00 01 43 03 00 00 00 61 01  .aChar...C....a.
00000060: 00 01 62 01 00 01 42 03 00 00 00 42 01 00 01 73  ..b...B....B...s
00000070: 01 00 01 53 03 00 00 00 43 01 00 01 69 01 00 01  ...S....C...i...
00000080: 49 03 00 00 00 44 01 00 06 3c 69 6e 69 74 3e 01  I....D...<init>.
00000090: 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c  ..()V...Code...L
000000a0: 69 6e 65 4e 75 6d 62 65 72 54 61 62 6c 65 01 00  ineNumberTable..
000000b0: 12 4c 6f 63 61 6c 56 61 72 69 61 62 6c 65 54 61  .LocalVariableTa
000000c0: 62 6c 65 01 00 04 74 68 69 73 01 00 1f 4c 63 6f  ble...this...Lco
000000d0: 6d 2f 69 74 77 61 6e 67 65 72 2f 6a 76 6d 2f 43  m/itwanger/jvm/C
000000e0: 6f 6e 73 74 61 6e 74 54 65 73 74 3b 01 00 0a 53  onstantTest;...S
```

第一个字节 0x03 表示常量的类型为 CONSTANT_Integer_info，是 JVM 定义的 14 种常量类型之一，对应的还有 CONSTANT_Float_info、CONSTANT_Long_info、CONSTANT_Double_info 等，它们对应的标识分别是 0x04、0x05、0x06。

我用表格来简单表示下：

常量类型	标识符	描述符
CONSTANT_Integer_info	0x03	int 类型字面量
CONSTANT_Float_info	0x04	float 类型字面量
CONSTANT_Long_info	0x05	long 类型字面量
CONSTANT_Double_info	0x06	double 类型字面量

对于 int 和 float 来说，它们占 4 个字节；对于 long 和 double 来说，它们占 8 个字节。来个 long 型的最大值观察下。

```
public class ConstantTest {
    public final long ong = Long.MAX_VALUE;
}
```

来看一下它在 class 文件中的位置。05 开头，7f ff ff ff ff ff ff ff 结尾，果然占 8 个字节，以前知道 long 型会占 8 个字节，但没有直观的感受，现在有了（😁）。

```
$ xxd -g 1 ConstantTest.class
00000000: ca fe ba be 00 00 00 37 00 19 0a 00 07 00 14 07  .......7........
00000010: 00 15 05 7f ff ff ff ff ff ff ff 09 00 06 00 16  ................
00000020: 07 00 17 07 00 18 01 00 03 6f 6e 67 01 00 01 4a  .........ong...J
00000030: 01 00 0d 43 6f 6e 73 74 61 6e 74 56 61 6c 75 65  ...ConstantValue
00000040: 01 00 06 3c 69 6e 69 74 3e 01 00 03 28 29 56 01  ...<init>...()V.
00000050: 00 04 43 6f 64 65 01 00 0f 4c 69 6e 65 4e 75 6d  ..Code...LineNum
00000060: 62 65 72 54 61 62 6c 65 01 00 12 4c 6f 63 61 6c  berTable...Local
```

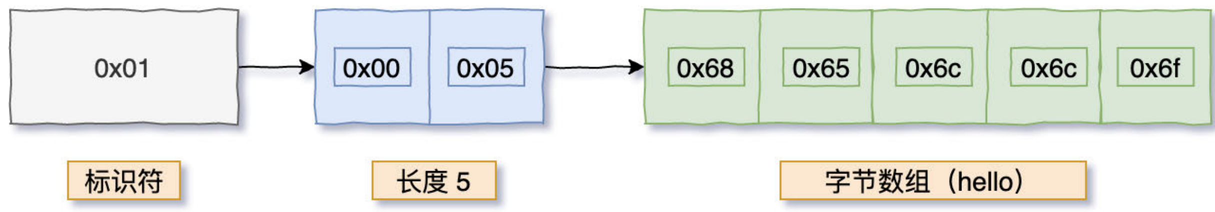
接下来，我们再来看一段代码。

```
class Hello {
    public final String s = "hello";
}
```

“hello”是一个字符串，它的十六进制为 `68 65 6c 6c 6f`，我们来看一下它在 class 文件中的位置。

```
$ xxd -g 1 Hello.class
00000000: ca fe ba be 00 00 00 37 00 17 0a 00 05 00 12 08  .......7........
00000010: 00 13 09 00 04 00 14 07 00 15 07 00 16 01 00 01  ................
00000020: 73 01 00 12 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53  s...Ljava/lang/S
00000030: 74 72 69 6e 67 3b 01 00 0d 43 6f 6e 73 74 61 6e  tring;...Constan
00000040: 74 56 61 6c 75 65 01 00 06 3c 69 6e 69 74 3e 01  tValue...<init>.
00000050: 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c  ..()V...Code...L
00000060: 69 6e 65 4e 75 6d 62 65 72 54 61 62 6c 65 01 00  ineNumberTable..
00000070: 12 4c 6f 63 61 6c 56 61 72 69 61 62 6c 65 54 61  .LocalVariableTa
00000080: 62 6c 65 01 00 04 74 68 69 73 01 00 18 4c 63 6f  ble...this...Lco
00000090: 6d 2f 69 74 77 61 6e 67 65 72 2f 6a 76 6d 2f 48  m/itwanger/jvm/H
000000a0: 65 6c 6c 6f 3b 01 00 0a 53 6f 75 72 63 65 46 69  ello;...SourceFi
000000b0: 6c 65 01 00 0a 48 65 6c 6c 6f 2e 6a 61 76 61 0c  le...Hello.java.
000000c0: 00 09 00 0a 01 00 05 68 65 6c 6c 6f 0c 00 06 00  .......hello....
000000d0: 07 01 00 16 63 6f 6d 2f 69 74 77 61 6e 67 65 72  ....com/itwanger
000000e0: 2f 6a 76 6d 2f 48 65 6c 6c 6f 01 00 10 6a 61 76  /jvm/Hello...jav
000000f0: 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 00 20 00  a/lang/Object. .
00000100: 04 00 05 00 00 00 01 00 11 00 06 00 07 00 01 00  ................
00000110: 08 00 00 00 02 00 02 00 01 00 00 00 09 00 0a 00  ................
00000120: 01 00 0b 00 00 00 39 00 02 00 01 00 00 00 0b 2a  ......9........*
00000130: b7 00 01 2a 12 02 b5 00 03 b1 00 00 00 02 00 0c  ...*............
00000140: 00 00 00 0a 00 02 00 00 00 06 00 04 00 07 00 0d  ................
00000150: 00 00 00 0c 00 01 00 00 00 0b 00 0e 00 0f 00 00  ................
00000160: 00 01 00 10 00 00 00 02 00 11                    ..........
```

前面还有 3 个字节，第一个字节 0x01 是标识，标识类型为 CONSTANT_Uft8_info，第二个和第三个 0x00 0x05 用来表示第三部分字节数组的长度，如下图所示。



与 CONSTANT_Uft8_info 类型对应的，还有一个 CONSTANT_String_info，用来表示字符串对象的引用（之前代码中的 s），标识是 0x08。前者存储了字符串真正的值，后者并不包含字符串的内容，仅仅包含了一个指向常量池中 CONSTANT_Uft8_info 的索引。

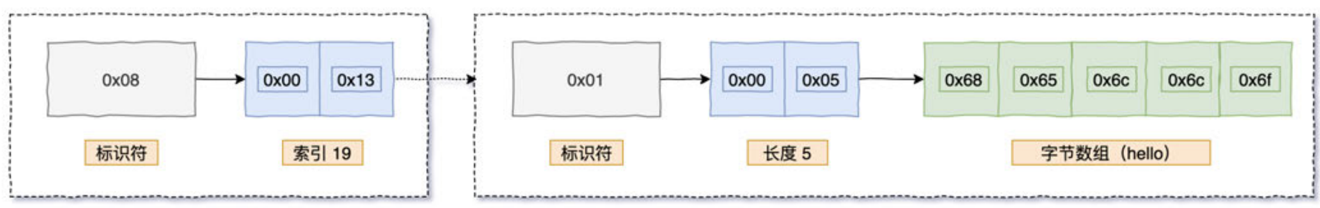
这和我们前面讲的对象和引用就关联起来了，有没有？（😁）

- Java到底是值传递还是引用传递？

来看一下它在 class 文件中的位置。

```
$ xxd -g 1 Hello.class
00000000: ca fe ba be 00 00 00 37 00 17 0a 00 05 00 12 08  .......7........
00000010: 00 13 09 00 04 00 14 07 00 15 07 00 16 01 00 01  ................
00000020: 73 01 00 12 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53  s...Ljava/lang/S
00000030: 74 72 69 6e 67 3b 01 00 0d 43 6f 6e 73 74 61 6e  tring;...Constan
00000040: 74 56 61 6c 75 65 01 00 06 3c 69 6e 69 74 3e 01  tValue...<init>.
00000050: 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c  ..()V...Code...L
00000060: 69 6e 65 4e 75 6d 62 65 72 54 61 62 6c 65 01 00  ineNumberTable..
00000070: 12 4c 6f 63 61 6c 56 61 72 69 61 62 6c 65 54 61  .LocalVariableTa
00000080: 62 6c 65 01 00 04 74 68 69 73 01 00 18 4c 63 6f  ble...this...Lco
00000090: 6d 2f 69 74 77 61 6e 67 65 72 2f 6a 76 6d 2f 48  m/itwanger/jvm/H
000000a0: 65 6c 6c 6f 3b 01 00 0a 53 6f 75 72 63 65 46 69  ello;...SourceFi
000000b0: 6c 65 01 00 0a 48 65 6c 6c 6f 2e 6a 61 76 61 0c  le...Hello.java.
000000c0: 00 09 00 0a 01 00 05 68 65 6c 6c 6f 0c 00 06 00  .......hello....
000000d0: 07 01 00 16 63 6f 6d 2f 69 74 77 61 6e 67 65 72  ....com/itwanger
000000e0: 2f 6a 76 6d 2f 48 65 6c 6c 6f 01 00 10 6a 61 76  /jvm/Hello...jav
000000f0: 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 00 20 00  a/lang/Object. .
00000100: 04 00 05 00 00 00 01 00 11 00 06 00 07 00 01 00  ................
00000110: 08 00 00 00 02 00 02 00 01 00 00 00 09 00 0a 00  ................
00000120: 01 00 0b 00 00 00 39 00 02 00 01 00 00 00 0b 2a  ......9........*
00000130: b7 00 01 2a 12 02 b5 00 03 b1 00 00 00 02 00 0c  ...*............
00000140: 00 00 00 0a 00 02 00 00 00 06 00 04 00 07 00 0d  ................
00000150: 00 00 00 0c 00 01 00 00 00 0b 00 0e 00 0f 00 00  ................
00000160: 00 01 00 10 00 00 00 02 00 11                    ..........
```

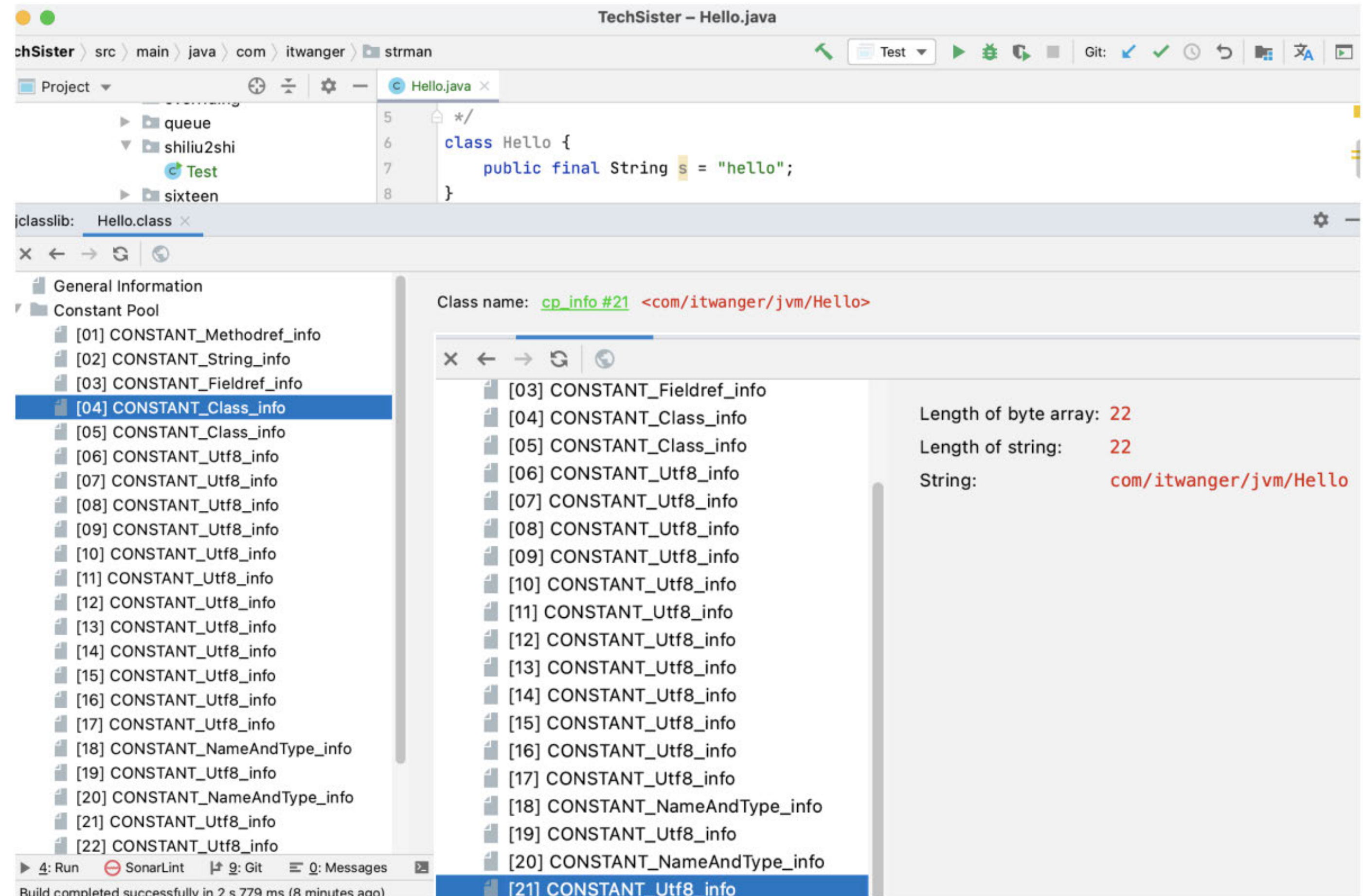
CONSTANT_String_info 通过索引 19 来找到 CONSTANT_Uft8_info，见下图。



除此之外，还有 CONSTANT_Class_info，用来表示类和接口，和 CONSTANT_String_info 类似，第一个字节是标识，值为 0x07，后面两个字节是常量池索引，指向 CONSTANT_Utf8_info——字符串存储的是类或者接口的全路径限定名。

拿 Hello.java 类来说，它的全路径限定名为 `com/itwanger/jvm/Hello`，对应的十六进制为 “636f6d2f697477616e6765722f6a766d2f48656c6c6f”，是一串 CONSTANT_Uft8_info，指向它的 CONSTANT_Class_info 在 class 文件中的什么位置呢？

先不着急，这里给大家介绍一款可视化字节码的工具 jclasslib bytecode viewer（前面也曾讲过），可以直接在 IDEA 的插件市场安装。安装完成后，选中 class 文件，然后在 View 菜单里找到 Show Bytecode With Jclasslib 子菜单，就可以查看 class 文件的关键信息了。



从上图中可以看到，常量池的总大小为 23，索引为 04 的 CONSTANT_Class_info 指向的是是索引为 21 的 CONSTANT_Uft8_info，值为 `com/itwanger/jvm/Hello`。21 的十六进制为 0x15，有了这个信息，我们就可以找到 CONSTANT_Class_info 在 class 文件中的位置了。

```
$ xxd -g 1 Hello.class
00000000: ca fe ba be 00 00 00 37 00 17 0a 00 05 00 12 08  .......7........
00000010: 00 13 09 00 04 00 14 07 00 15 07 00 16 01 00 01  ................
00000020: 73 01 00 12 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53  s...Ljava/lang/S
00000030: 74 72 69 6e 67 3b 01 00 0d 43 6f 6e 73 74 61 6e  tring;...Constan
00000040: 74 56 61 6c 75 65 01 00 06 3c 69 6e 69 74 3e 01  tValue...<init>.
00000050: 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c  ..()V...Code...L
00000060: 69 6e 65 4e 75 6d 62 65 72 54 61 62 6c 65 01 00  ineNumberTable..
00000070: 12 4c 6f 63 61 6c 56 61 72 69 61 62 6c 65 54 61  .LocalVariableTa
00000080: 62 6c 65 01 00 04 74 68 69 73 01 00 18 4c 63 6f  ble...this...Lco
00000090: 6d 2f 69 74 77 61 6e 67 65 72 2f 6a 76 6d 2f 48  m/itwanger/jvm/H
000000a0: 65 6c 6c 6f 3b 01 00 0a 53 6f 75 72 63 65 46 69  ello;...SourceFi
000000b0: 6c 65 01 00 0a 48 65 6c 6c 6f 2e 6a 61 76 61 0c  le...Hello.java.
000000c0: 00 09 00 0a 01 00 05 68 65 6c 6c 6f 0c 00 06 00  .......hello....
000000d0: 07 01 00 16 63 6f 6d 2f 69 74 77 61 6e 67 65 72  ....com/itwanger
000000e0: 2f 6a 76 6d 2f 48 65 6c 6c 6f 01 00 10 6a 61 76  /jvm/Hello...jav
000000f0: 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 00 20 00  a/lang/Object. .
00000100: 04 00 05 00 00 00 01 00 11 00 06 00 07 00 01 00  ................
00000110: 08 00 00 00 02 00 02 00 01 00 00 00 09 00 0a 00  ................
00000120: 01 00 0b 00 00 00 39 00 02 00 01 00 00 00 0b 2a  ......9........*
00000130: b7 00 01 2a 12 02 b5 00 03 b1 00 00 00 02 00 0c  ...*............
00000140: 00 00 00 0a 00 02 00 00 00 06 00 04 00 07 00 0d  ................
00000150: 00 00 00 0c 00 01 00 00 00 0b 00 0e 00 0f 00 00  ................
```

0x07 是第一个字节，CONSTANT_Class_info 的标识符，然后是两个字节，标识索引。

还有 CONSTANT_NameAndType_info，用来标识字段或方法，标识符为 12，对应的十六进制是 0x0c。后面还有 4 个字节，前两个是字段或者方法的索引，后两个是字段或方法的描述符，也就是字段或者方法的类型。

来看下面这段代码。

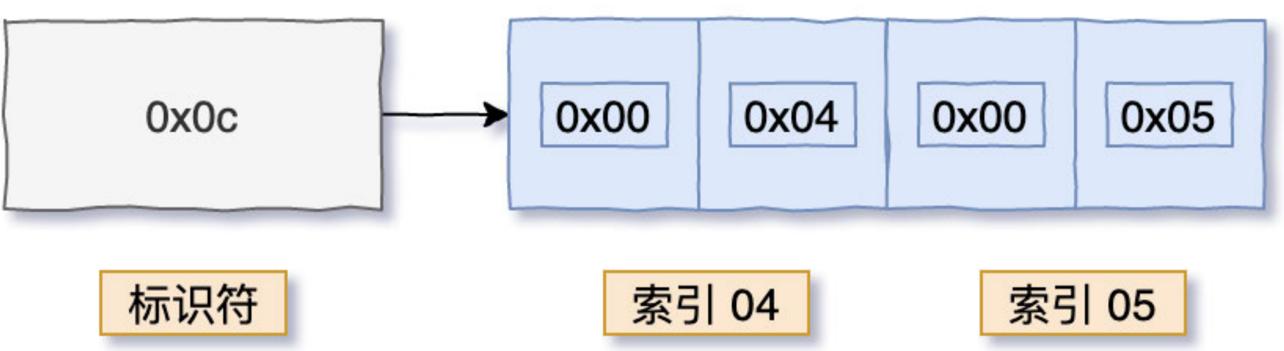
```
class Hello {
    public void testMethod(int id, String name) {
    }
}
```

用 jclasslib 可以看到 CONSTANT_NameAndType_info 包含的索引有两个。

Constant Pool
[01] CONSTANT_Methodref_info
[02] CONSTANT_Class_info
[03] CONSTANT_Class_info
[04] CONSTANT_Utf8_info
[05] CONSTANT_Utf8_info
[06] CONSTANT_Utf8_info
[07] CONSTANT_Utf8_info
[08] CONSTANT_Utf8_info
[09] CONSTANT_Utf8_info
[10] CONSTANT_Utf8_info
[11] CONSTANT_Utf8_info
[12] CONSTANT_Utf8_info
[13] CONSTANT_Utf8_info
[14] CONSTANT_Utf8_info
[15] CONSTANT_Utf8_info
[16] CONSTANT_Utf8_info
[17] CONSTANT_Utf8_info
[18] CONSTANT_Utf8_info
[19] CONSTANT_NameAndType_info
[20] CONSTANT_Utf8_info
[21] CONSTANT_Utf8_info

Name: cp_info #4 <<init>>
Descriptor: cp_info #5 <()V>

一个是 4，一个是 5，可以通过下图来表示 CONSTANT_NameAndType_info 的构成。



对应 class 文件中的位置如下图所示。

```
$ xxd -g 1 Hello.class
00000000: ca fe ba be 00 00 00 37 00 16 0a 00 03 00 13 07  .......7........
00000010: 00 14 07 00 15 01 00 06 3c 69 6e 69 74 3e 01 00  ........<init>..
00000020: 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c 69  .()V...Code...Li
00000030: 6e 65 4e 75 6d 62 65 72 54 61 62 6c 65 01 00 12  neNumberTable...
00000040: 4c 6f 63 61 6c 56 61 72 69 61 62 6c 65 54 61 62  LocalVariableTab
00000050: 6c 65 01 00 04 74 68 69 73 01 00 18 4c 63 6f 6d  le...this...Lcom
00000060: 2f 69 74 77 61 6e 67 65 72 2f 6a 76 6d 2f 48 65  /itwanger/jvm/He
00000070: 6c 6c 6f 3b 01 00 0a 74 65 73 74 4d 65 74 68 6f  llo;...testMetho
00000080: 64 01 00 16 28 49 4c 6a 61 76 61 2f 6c 61 6e 67  d...(ILjava/lang
00000090: 2f 53 74 72 69 6e 67 3b 29 56 01 00 02 69 64 01  /String;)V...id.
000000a0: 00 01 49 01 00 04 6e 61 6d 65 01 00 12 4c 6a 61  ..I...name...Lja
000000b0: 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 01  va/lang/String;.
000000c0: 00 0a 53 6f 75 72 63 65 46 69 6c 65 01 00 0a 48  ..SourceFile...H
000000d0: 65 6c 6c 6f 2e 6a 61 76 61 0c 00 04 00 05 01 00  ello.java.......
000000e0: 16 63 6f 6d 2f 69 74 77 61 6e 67 65 72 2f 6a 76  .com/itwanger/jv
000000f0: 6d 2f 48 65 6c 6c 6f 01 00 10 6a 61 76 61 2f 6c  m/Hello...java/l
00000100: 61 6e 67 2f 4f 62 6a 65 63 74 00 20 00 02 00 03  ang/Object. ....
00000110: 00 00 00 00 00 02 00 00 00 04 00 05 00 01 00 06  ................
00000120: 00 00 00 2f 00 01 00 01 00 00 00 05 2a b7 00 01  .../........*...
00000130: b1 00 00 00 02 00 07 00 00 00 06 00 01 00 00 00  ................
00000140: 06 00 08 00 00 00 0c 00 01 00 00 00 05 00 09 00  ................
00000150: 0a 00 00 00 01 00 0b 00 0c 00 01 00 06 00 00 00  ................
00000160: 3f 00 00 00 03 00 00 00 01 b1 00 00 00 02 00 07  ?...............
00000170: 00 00 00 06 00 01 00 00 00 07 00 08 00 00 00 20  ...............
00000180: 00 03 00 00 00 01 00 09 00 0a 00 00 00 00 00 01  ................
00000190: 00 0d 00 0e 00 01 00 00 00 01 00 0f 00 10 00 02  ................
000001a0: 00 01 00 11 00 00 00 02 00 12                    ..........
```

接下来是 CONSTANT_Fieldref_info 、CONSTANT_Methodref_info 和 CONSTANT_InterfaceMethodref_info，它们三个的结构比较类似，可以通过下面的伪代码来表示。

```
CONSTANT_*ref_info {
  u1 tag;
  u2 class_index;
  u2 name_and_type_index;
}
```

学过 C 语言的符号表（Symbol Table）的话，对这段伪代码并不会陌生。

- tag 为标识符，Fieldref 的为 9，也就是十六进制的 0x09；Methodref 的为 10，也就是十六进制的 0x0a；InterfaceMethodref 的为 11， 也就是十六进制的 0x0b。
- class_index 为 CONSTANT_Class_info 的常量池索引，表示字段 | 方法 | 接口方法所在的类信息。
- name_and_type_index 为 CONSTANT_NameAndType_info 的常量池索引，拿 Fieldref 来说，表示字段名和字段类型；拿 Methodref 来说，表示方法名、方法的参数和返回值类型；拿 InterfaceMethodref 来说，表示接口方法名、接口方法的参数和返回值类型。

还有 CONSTANT_MethodHandle_info 、CONSTANT_MethodType_info 和 CONSTANT_InvokeDynamic_info，我这里用一个表格来表示下：

常量类型	标识符	描述符
CONSTANT_MethodHandle_info	0x0f	方法句柄
CONSTANT_MethodType_info	0x10	方法类型
CONSTANT_InvokeDynamic_info	0x12	动态调用点
CONSTANT_Fieldref_info	0x09	字段
CONSTANT_Methodref_info	0x0a	普通方法
CONSTANT_InterfaceMethodref_info	0x0b	接口方法
CONSTANT_Class_info	0x07	类或接口的全限定名
CONSTANT_String_info	0x08	字符串字面量
CONSTANT_Uft8_info	0x01	字符串

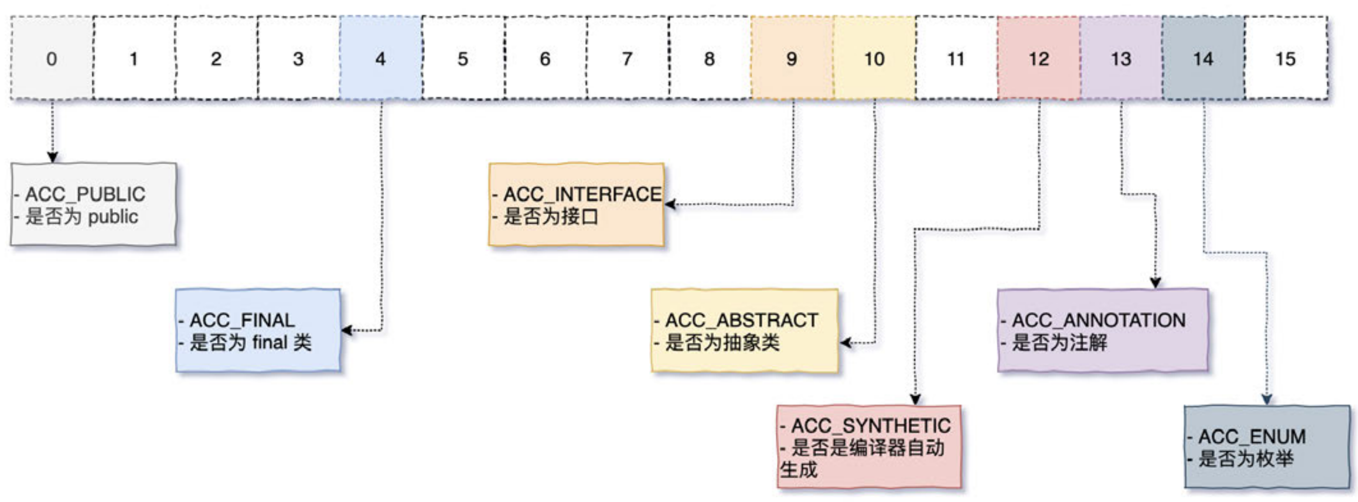
啊，class 文件中最复杂的常量池部分就算是解剖完了，真不容易！

04、访问标记

紧跟着常量池之后的区域就是访问标记（Access flags），这个标记用于识别类或接口的访问信息，比如说：

- 到底是 class 类 还是 interface 接口?
- 是 public 吗?
- 是 abstract 抽象类吗?
- 是 final 类吗?
- 等等。

总共有 16 个标记位可供使用，但常用的只有其中 7 个，见下图。



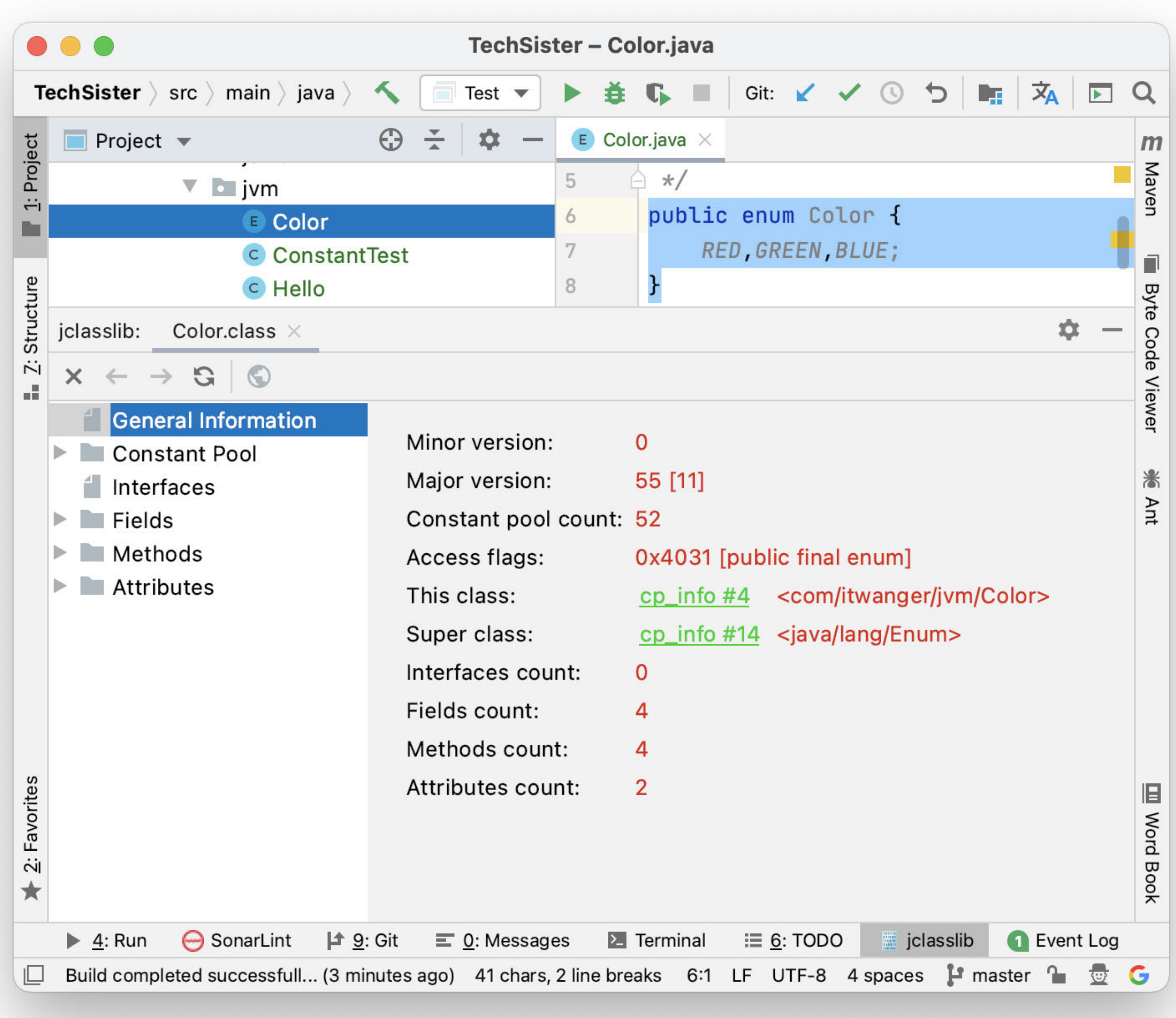
这里用一个表格来表示下。

标记位	标识符	描述
0x0001	ACC_PUBLIC	public 类型
0x0010	ACC_FINAL	final 类型
0x0020	ACC_SUPER	调用父类的方法时，使用 invokespecial 指令
0x0200	ACC_INTERFACE	接口类型
0x0400	ACC_ABSTRACT	抽象类类型
0x1000	ACC_SYNTHETIC	标记为编译器自动生成的类
0x2000	ACC_ANNOTATION	标记为注解类
0x4000	ACC_ENUM	标记为枚举类
0x8000	ACC_MODULE	标记为模块类

来看一个简单的枚举代码。

```
public enum Color {
    RED,GREEN,BLUE;
}
```

通过 jclasslib 可以看到访问标记的信息有 `0x4031 [public final enum]`。



对应 class 文件中的位置如下图所示。

```
00000190: 76 61 2f 6c 61 6e 67 2f 45 6e 75 6d 3c 4c 63 6f  va/lang/Enum<Lco
000001a0: 6d 2f 69 74 77 61 6e 67 65 72 2f 6a 76 6d 2f 43  m/itwanger/jvm/C
000001b0: 6f 6c 6f 72 3b 3e 3b 01 00 0a 53 6f 75 72 63 65  olor;>;...Source
000001c0: 46 69 6c 65 01 00 0a 43 6f 6c 6f 72 2e 6a 61 76  File...Color.jav
000001d0: 61 0c 00 13 00 14 07 00 14 0c 00 31 00 32 01 00  a.........1.2..
000001e0: 16 63 6f 6d 2f 69 74 77 61 6e 67 65 72 2f 6a 76  .com/itwanger/jv
000001f0: 6d 2f 43 6f 6c 6f 72 0c 00 19 00 33 0c 00 1e 00  m/Color....3....
00000200: 1f 0c 00 0f 00 10 0c 00 11 00 10 0c 00 12 00 10  ................
00000210: 01 00 0e 6a 61 76 61 2f 6c 61 6e 67 2f 45 6e 75  ...java/lang/Enu
00000220: 6d 01 00 05 63 6c 6f 6e 65 01 00 14 28 29 4c 6a  m...clone...()Lj
00000230: 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 3b  ava/lang/Object;
00000240: 01 00 35 28 4c 6a 61 76 61 2f 6c 61 6e 67 2f 43  ..5(Ljava/lang/C
00000250: 6c 61 73 73 3b 4c 6a 61 76 61 2f 6c 61 6e 67 2f  lass;Ljava/lang/
00000260: 53 74 72 69 6e 67 3b 29 4c 6a 61 76 61 2f 6c 61  String;)Ljava/la
00000270: 6e 67 2f 45 6e 75 6d 3b 40 31 00 04 00 0e 00 00  ng/Enum;@1......
00000280: 00 04 40 19 00 0f 00 10 00 00 40 19 00 11 00 10  ..@.......@.....
00000290: 00 00 40 19 00 12 00 10 00 00 10 1a 00 13 00 14  ..@.............
```

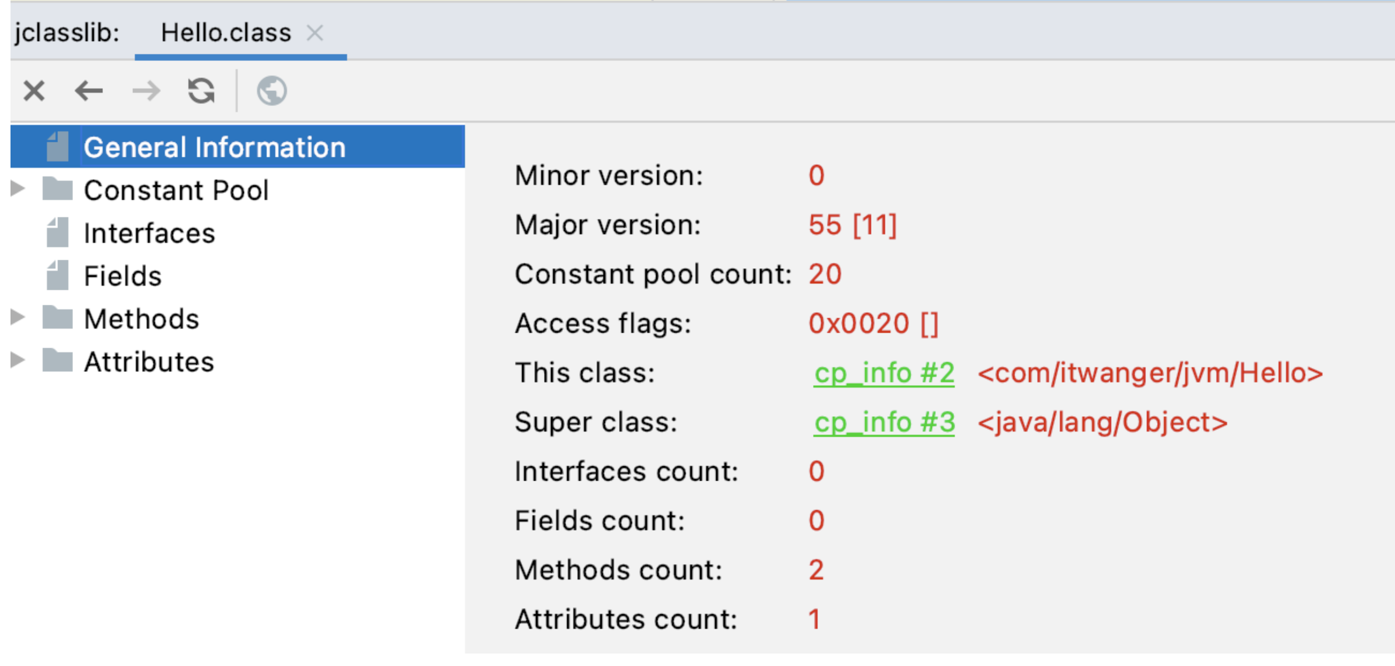
05、类索引、父类索引和接口索引

这三部分用来确定类的继承关系，this_class 为当前类的索引，super_class 为父类的索引，interfaces 为接口。

来看下面这段简单的代码，没有接口，默认继承 Object 类。

```java
class Hello {
    public static void main(String[] args) {

    }
}
```

通过 jclasslib 可以看到类的继承关系。



- this_class 指向常量池中索引为 2 的 CONSTANT_Class_info。
- super_class 指向常量池中索引为 3 的 CONSTANT_Class_info。
- 由于没有接口，所以 interfaces 的信息为空。

对应 class 文件中的位置如下图所示。

```
000000d0: 01 00 16 63 6f 6d 2f 69 74 77 61 6e 67 65 72 2f  ...com/itwanger/
000000e0: 6a 76 6d 2f 48 65 6c 6c 6f 01 00 10 6a 61 76 61  jvm/Hello...java
000000f0: 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 00 20 00 02  /lang/Object. ..
00000100: 00 03 00 00 00 00 00 02 00 00 00 04 00 05 00 01  ................
00000110: 00 06 00 00 00 2f 00 01 00 01 00 00 00 05 2a b7  ...../........*.
00000120: 00 01 b1 00 00 00 02 00 07 00 00 00 06 00 01 00  ................
00000130: 00 00 06 00 08 00 00 00 0c 00 01 00 00 00 05 00  ................
```

06、字段表

一个类中定义的字段会被存储在字段表（fields）中，包括静态的和非静态的。

来看这样一段代码。

```
public class FieldsTest {
    private String name;
}
```

字段只有一个，修饰符为 private，类型为 String，字段名为 name。可以用下面的伪代码来表示 field 的结构。

```
field_info {
  u2 access_flag;
  u2 name_index;
  u2 description_index;
}
```

- access_flag 为字段的访问标记，比如说是不是 public | private | protected，是不是 static，是不是 final 等。
- name_index 为字段名的索引，指向常量池中的 CONSTANT_Utf8_info， 比如说上例中的值就为 name。
- description_index 为字段的描述类型索引，也指向常量池中的 CONSTANT_Utf8_info，针对不同的数据类型，会有不同规则的描述信息。

1）对于基本数据类型来说，使用一个字符来表示，比如说 I 对应的是 int，B 对应的是 byte。

2）对于引用数据类型来说，使用 `L***;` 的方式来表示，`L` 开头，`;` 结束，比如字符串类型为 `Ljava/lang/String;`。

3）对于数组来说，会用一个前置的 `[` 来表示，比如说字符串数组为 `[Ljava/lang/String;`。

对应到 class 文件中的位置如下图所示。

```
$ xxd -g 1 FieldsTest.class
00000000: ca fe ba be 00 00 00 37 00 12 0a 00 03 00 0f 07  .......7........
00000010: 00 10 07 00 11 01 00 04 6e 61 6d 65 01 00 12 4c  ........name...L
00000020: 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67  java/lang/String
00000030: 3b 01 00 06 3c 69 6e 69 74 3e 01 00 03 28 29 56  ;...<init>...()V
00000040: 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e 65 4e 75  ...Code...LineNu
00000050: 6d 62 65 72 54 61 62 6c 65 01 00 12 4c 6f 63 61  mberTable...Loca
00000060: 6c 56 61 72 69 61 62 6c 65 54 61 62 6c 65 01 00  lVariableTable..
00000070: 04 74 68 69 73 01 00 1d 4c 63 6f 6d 2f 69 74 77  .this...Lcom/itw
00000080: 61 6e 67 65 72 2f 6a 76 6d 2f 46 69 65 6c 64 73  anger/jvm/Fields
00000090: 54 65 73 74 3b 01 00 0a 53 6f 75 72 63 65 46 69  Test;...SourceFi
000000a0: 6c 65 01 00 0f 46 69 65 6c 64 73 54 65 73 74 2e  le...FieldsTest.
000000b0: 6a 61 76 61 0c 00 06 00 07 01 00 1b 63 6f 6d 2f  java........com/
000000c0: 69 74 77 61 6e 67 65 72 2f 6a 76 6d 2f 46 69 65  itwanger/jvm/Fie
000000d0: 6c 64 73 54 65 73 74 01 00 10 6a 61 76 61 2f 6c  ldsTest...java/l
000000e0: 61 6e 67 2f 4f 62 6a 65 63 74 00 21 00 02 00 03  ang/Object.!....
000000f0: 00 00 00 01 00 02 00 04 00 05 00 00 00 01 00 01  ................
00000100: 00 06 00 07 00 01 00 08 00 00 00 2f 00 01 00 01  .........../....
00000110: 00 00 00 05 2a b7 00 01 b1 00 00 00 02 00 09 00  ....*...........
00000120: 00 00 06 00 01 00 00 00 06 00 0a 00 00 00 0c 00  ................
00000130: 01 00 00 00 05 00 0b 00 0c 00 00 00 01 00 0d 00  ................
00000140: 00 00 02 00 0e                                   .....
```

看到这里相信你就能明白经常在 javap 命令中看到的一些奇怪的字符的意思了。

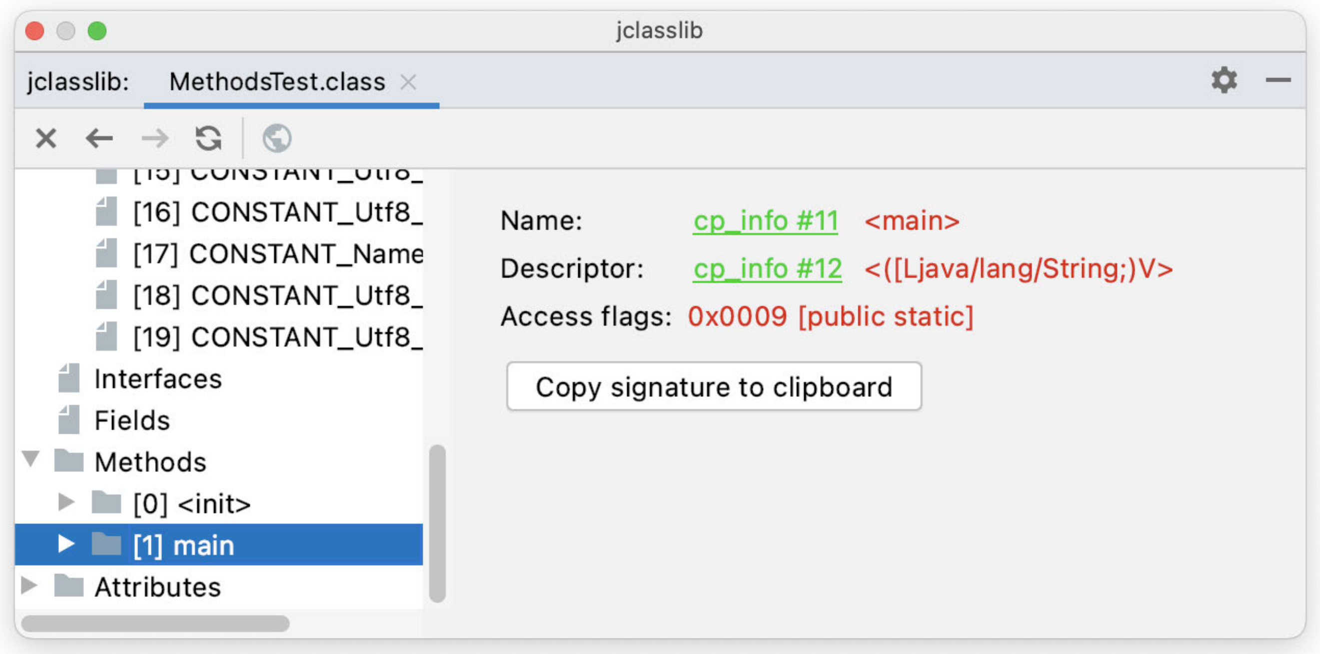
07、方法表

方法表和字段表类似，区别是用来存储方法的信息，包括方法名，方法的参数，方法的签名。

就拿 main 方法来说吧。

```java
public class MethodsTest {
    public static void main(String[] args) {

    }
}
```

先用 jclasslib 看一下大概的信息。



- 访问标记是 public static 的。
- 方法名为 main。
- 方法的参数为字符串数组；返回类型为 Void。

对应到 class 文件中的位置如下图所示。

```
00000100: 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 00 21  va/lang/Object.!
00000110: 00 02 00 03 00 00 00 00 00 02 00 01 00 04 00 05  ................
00000120: 00 01 00 06 00 00 00 2f 00 01 00 01 00 00 00 05  ......./........
00000130: 2a b7 00 01 b1 00 00 00 02 00 07 00 00 00 06 00  *...............
00000140: 01 00 00 00 06 00 08 00 00 00 0c 00 01 00 00 00  ................
00000150: 05 00 09 00 0a 00 00 00 09 00 0b 00 0c 00 01 00  ................
00000160: 06 00 00 00 2b 00 00 00 01 00 00 00 01 b1 00 00  ....+...........
00000170: 00 02 00 07 00 00 00 06 00 01 00 00 00 09 00 08  ................
```

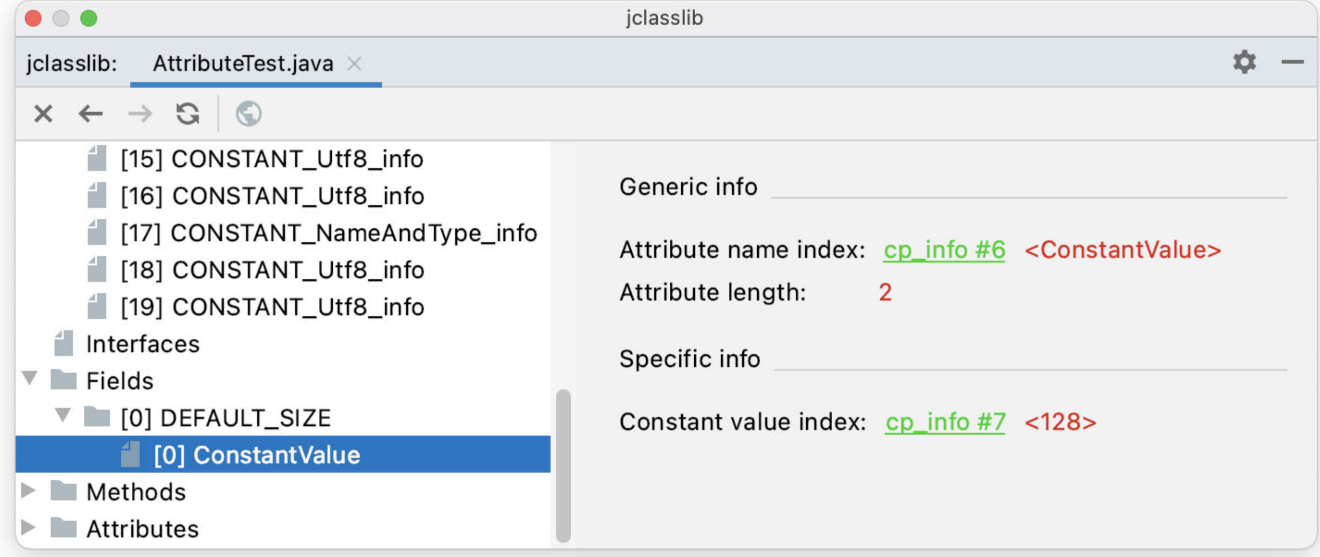
08、属性表

属性表是 class 文件中的最后一部分，通常出现在字段和方法中。

来看这样一段代码。

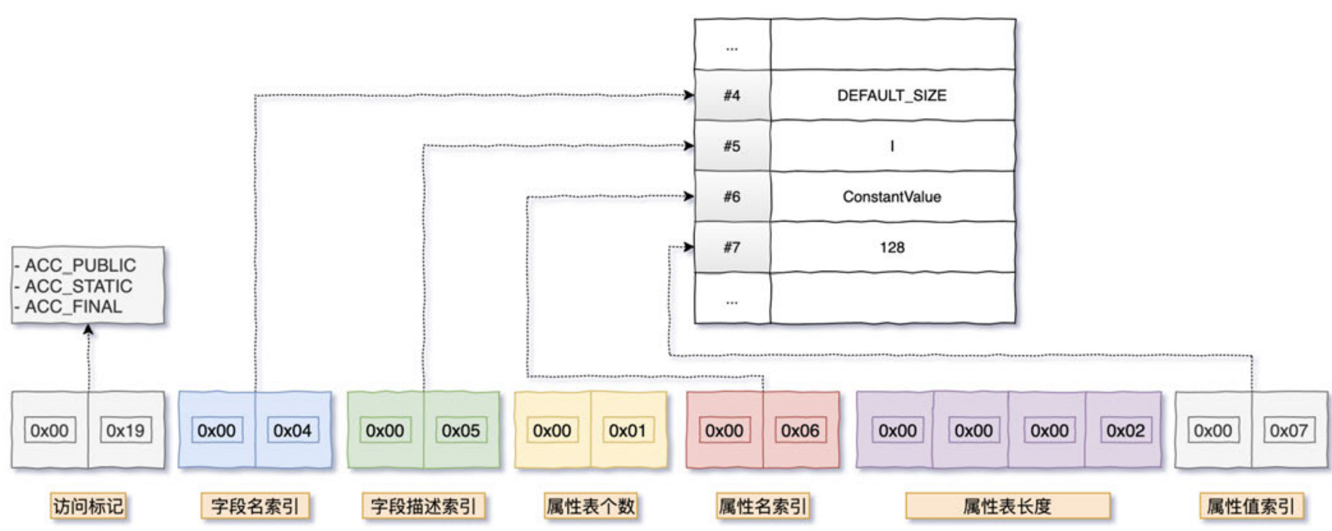
```
public class AttributeTest {
    public static final int DEFAULT_SIZE = 128;
}
```

只有一个常量 DEFAULT_SIZE，它属于字段中的一种，就是加了 final 的静态变量。先通过 jclasslib 看一下它当中一个很重要的属性——ConstantValue，用来表示静态变量的初始值。



- Attribute name index 指向常量池中值为“ConstantValue”的常量。
- Attribute length 的值为固定的 2，因为索引只占两个字节的大小。
- Constant value index 指向常量池中具体的常量，如果常量类型为 int，指向的就是 CONSTANT_Integer_info。

我画了一副图，可以完整的表示字段的结构，包含属性表在内。



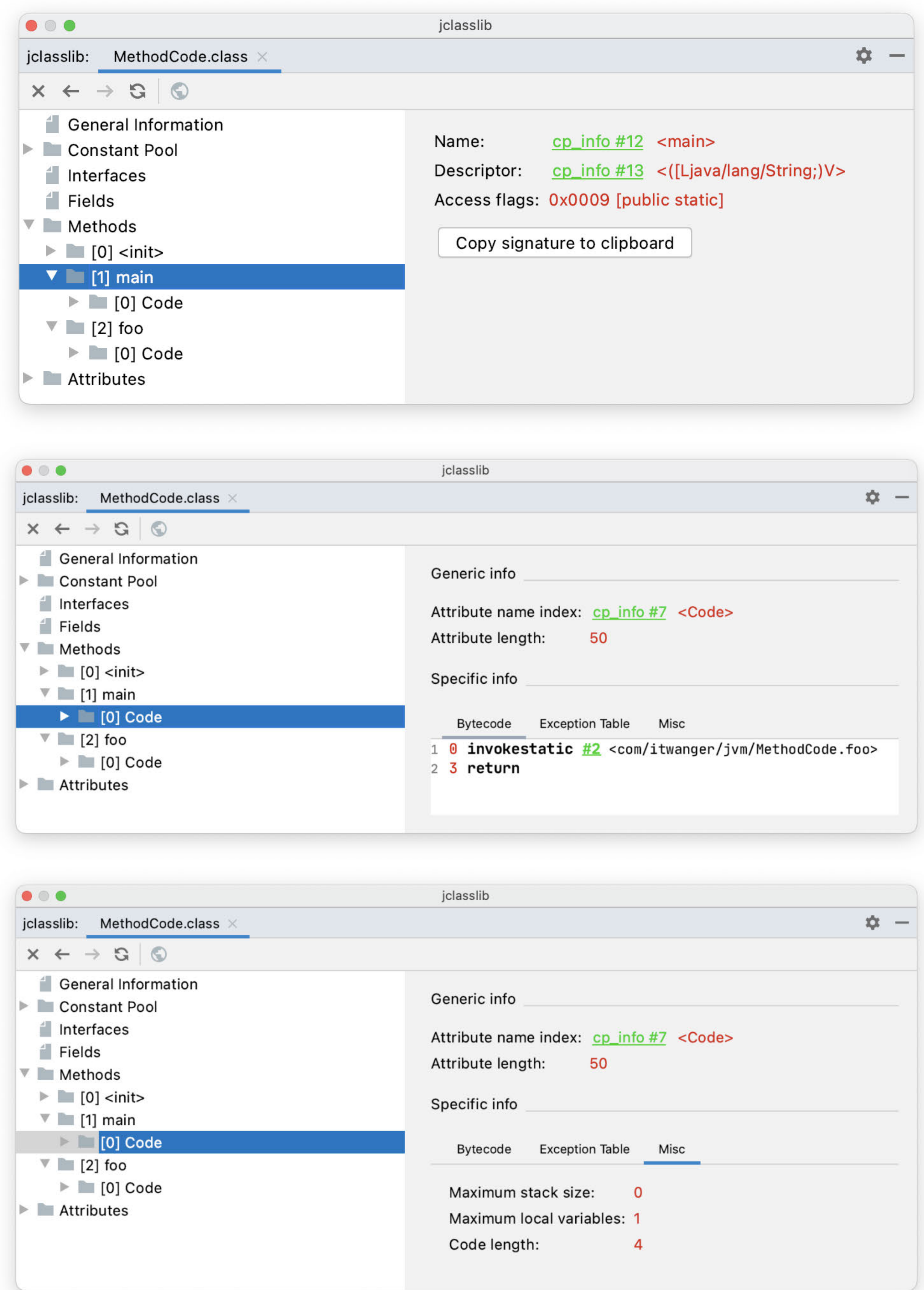
对应到 class 文件中的位置如下图所示。

```
$ xxd -g 1 AttributeTest.class
00000000: ca fe ba be 00 00 00 37 00 14 0a 00 03 00 11 07  .......7........
00000010: 00 12 07 00 13 01 00 0c 44 45 46 41 55 4c 54 5f  ........DEFAULT_
00000020: 53 49 5a 45 01 00 01 49 01 00 0d 43 6f 6e 73 74  SIZE...I...Const
00000030: 61 6e 74 56 61 6c 75 65 03 00 00 00 80 01 00 06  antValue........
00000040: 3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00 04 43  <init>...()V...C
00000050: 6f 64 65 01 00 0f 4c 69 6e 65 4e 75 6d 62 65 72  ode...LineNumber
00000060: 54 61 62 6c 65 01 00 12 4c 6f 63 61 6c 56 61 72  Table...LocalVar
00000070: 69 61 62 6c 65 54 61 62 6c 65 01 00 04 74 68 69  iableTable...thi
00000080: 73 01 00 20 4c 63 6f 6d 2f 69 74 77 61 6e 67 65  s.. Lcom/itwange
00000090: 72 2f 6a 76 6d 2f 41 74 74 72 69 62 75 74 65 54  r/jvm/AttributeT
000000a0: 65 73 74 3b 01 00 0a 53 6f 75 72 63 65 46 69 6c  est;...SourceFil
000000b0: 65 01 00 12 41 74 74 72 69 62 75 74 65 54 65 73  e...AttributeTes
000000c0: 74 2e 6a 61 76 61 0c 00 08 00 09 01 00 1e 63 6f  t.java........co
000000d0: 6d 2f 69 74 77 61 6e 67 65 72 2f 6a 76 6d 2f 41  m/itwanger/jvm/A
000000e0: 74 74 72 69 62 75 74 65 54 65 73 74 01 00 10 6a  ttributeTest...j
000000f0: 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 00  ava/lang/Object.
00000100: 21 00 02 00 03 00 00 00 01 00 19 00 04 00 05 00  !...............
00000110: 01 00 06 00 00 00 02 00 07 00 01 00 01 00 08 00  ................
00000120: 09 00 01 00 0a 00 00 00 2f 00 01 00 01 00 00 00  ......../.......
00000130: 05 2a b7 00 01 b1 00 00 00 02 00 0b 00 00 00 06  .*..............
00000140: 00 01 00 00 00 06 00 0c 00 00 00 0c 00 01 00 00  ................
00000150: 00 05 00 0d 00 0e 00 00 00 01 00 0f 00 00 00 02  ................
00000160: 00 10                                            ..
```

来看下面这段代码。

```java
public class MethodCode {
    public static void main(String[] args) {
        foo();
    }

    private static void foo() {
    }
}
```

main 方法中调用了 foo 方法。通过 jclasslib 看一下它当中一个很重要的属性——Code， 方法的关键信息都存储在里面。



- Attribute name index 指向常量池中值为“Code”的常量。

- Attribute length 为属性值的长度大小。
- bytecode 存储真正的字节码指令。
- exception table 表示方法内部的异常信息。
- maximum stack size 表示操作数栈的最大深度，方法执行的任意期间操作数栈深度都不会超过这个值。
- maximum local variable 表示临时变量表的大小，注意，并不等于方法中所有临时变量的数量之和，当一个作用域结束，内部的临时变量占用的位置就会被替换掉。
- code length 表示字节码指令的长度。

对应 class 文件中的位置如下图所示。

```
00000120: 04 00 00 00 00 00 03 00 01 00 05 00 06 00 01 00  ................
00000130: 07 00 00 00 2f 00 01 00 01 00 00 00 05 2a b7 00  ..../........*..
00000140: 01 b1 00 00 00 02 00 08 00 00 00 06 00 01 00 00  ................
00000150: 00 06 00 09 00 00 00 0c 00 01 00 00 00 05 00 0a  ................
00000160: 00 0b 00 00 00 09 00 0c 00 0d 00 01 00 07 00 00  ................
00000170: 00 32 00 00 00 01 00 00 00 04 b8 00 02 b1 00 00  .2..............
00000180: 00 02 00 08 00 00 00 0a 00 02 00 00 00 08 00 03  ................
00000190: 00 09 00 09 00 00 00 0c 00 01 00 00 00 04 00 0e  ................
000001a0: 00 0f 00 00 00 0a 00 10 00 06 00 01 00 07 00 00  ................
000001b0: 00 19 00 00 00 00 00 00 00 01 b1 00 00 00 01 00  ................
000001c0: 08 00 00 00 06 00 01 00 00 00 0c 00 01 00 11 00  ................
000001d0: 00 00 02 00 12                                   .....
```

09、QA

评论区有读者问到：“怎么通过索引值，定位到在class 文件中的位置，这个是咋算的？”

在Java类文件中，常量池是一个索引表，它从索引值1开始计数，每个条目都有一个唯一的索引。

- 常量池计数器：在常量池之前，类文件有一个16位的常量池计数器，表示常量池中有多少项。它的值比实际常量数大1（因为索引从1开始）。
- 常量池条目：每个常量池条目的开始是一个标签（1个字节），表明了常量的类型（如Class、Fieldref、Methodref等）。根据这个类型，后面跟着的数据结构也不同。

定位过程大致如下：

- 读取常量池计数器：首先，从类文件的开头读取常量池计数器的值，确定常量池中有多少条目。
- 遍历常量池：从常量池的第一项开始遍历。由于不同类型的常量长度不同，需要根据每个常量的类型来确定它的长度。
- 根据索引定位：继续遍历，直到到达所需的索引值。每次遍历时，根据条目类型读取相应长度的数据，直到达到目标索引。

可以抽象成一个数组和一个 for 循环，就能明白了。

```
int[] constantPool = new int[constantPoolCount];
for (int i = 1; i < constantPoolCount; i++) {
    int tag = constantPool[i];
    switch (tag) {
        case CONSTANT_Integer_info:
```

```
                i += 4;
                break;
            case CONSTANT_Float_info:
                i += 4;
                break;
            case CONSTANT_Long_info:
                i += 8;
                break;
            case CONSTANT_Double_info:
                i += 8;
                break;
            case CONSTANT_Utf8_info:
                int length = constantPool[i + 1];
                i += length + 1;
                break;
            case CONSTANT_String_info:
                i += 2;
                break;
            case CONSTANT_Class_info:
                i += 2;
                break;
            case CONSTANT_Fieldref_info:
                i += 4;
                break;
            case CONSTANT_Methodref_info:
                i += 4;
                break;
            case CONSTANT_InterfaceMethodref_info:
                i += 4;
                break;
            case CONSTANT_NameAndType_info:
                i += 4;
                break;
            case CONSTANT_MethodHandle_info:
                i += 3;
                break;
            case CONSTANT_MethodType_info:
                i += 2;
                break;
            case CONSTANT_InvokeDynamic_info:
                i += 4;
                break;
            default:
                throw new RuntimeException("Unknown tag: " + tag);
        }
    }
```

10、小结

到此为止，class 文件的内部算是剖析得差不多了，希望能对大家有所帮助。第一次拿刀，手有点颤，如果哪里有不足的地方，欢迎大家在评论区毫不留情地指出来！

- class 文件是一串连续的二进制，由 0 和 1 组成，但我们仍然可以借助一些工具来看清楚它的真面目。
- class 文件的内容通常可以分为下面这几部分，魔数、版本号、常量池、访问标记、类索引、父类索引、接口索引、字段表、方法表、属性表。
- 常量池包含了类、接口、字段和方法的符号引用，以及字符串字面量和数值常量。
- 访问标记用于识别类或接口的访问信息，比如说是不是 public | private | protected，是不是 static，是不是 final 等。
- 类索引、父类索引和接口索引用来确定类的继承关系。
- 字段表用来存储字段的信息，包括字段名，字段的参数，字段的签名。
- 方法表用来存储方法的信息，包括方法名，方法的参数，方法的签名。
- 属性表用来存储属性的信息，包括字段的初始值，方法的字节码指令等。

相信大家看完这篇内容应该能对 class 文件有一个比较清晰的认识了。

第五节：javap 与字节码

计算机比较“傻”，只认 0 和 1，这意味着我们编写的代码最终都要编译成机器码才能被计算机执行。Java 在诞生之初就提出了一个非常著名的宣传口号: "**一次编写，处处运行**"。

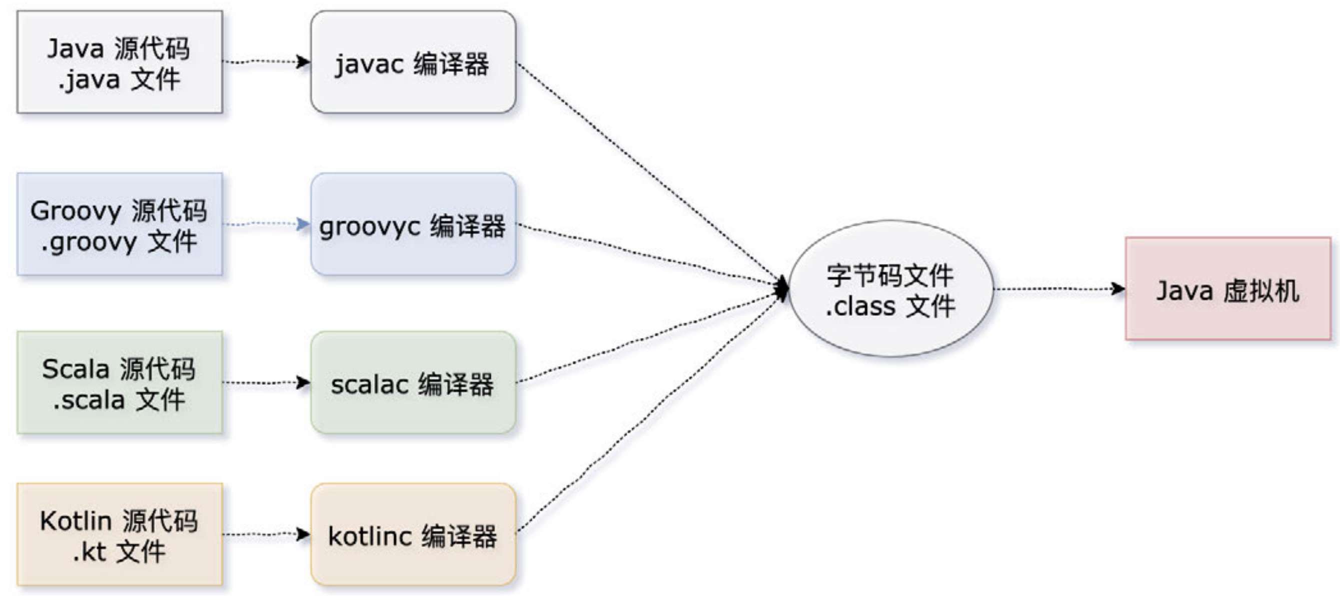
> **Write Once, Run Anywhere.**

为了这个口号，Java 的亲妈 Sun 公司以及其他虚拟机提供商发布了许多可以在不同平台上运行的 Java 虚拟机，而这些虚拟机都拥有一个共同的功能，那就是可以载入和执行同一种与平台无关的字节码（Byte Code）。

（前面其实我们也讲过，但为了这篇内容的完整性，我们简单过一下，这一节我们的重点是**通过 javap 这个命令来了解字节码**）

有了 Java 虚拟机的帮助，我们编写的 Java 源代码不必再根据不同平台编译成对应的机器码了，只需要生成一份字节码，然后再将字节码文件交由运行在不同平台上的 Java 虚拟机读取后执行就可以了。

如今的 Java 虚拟机非常强大，不仅支持 Java 语言，还支持很多其他的编程语言，比如说 Groovy、Scala、Koltin 等等。



来看一段代码吧。

```
public class Main {
    private int age = 18;
    public int getAge() {
        return age;
    }
}
```

编译生成 Main.class 文件后，可以在命令行使用 `xxd Main.class` 打开 class 文件（前面我们已经讲过了，还不会用的同学可以回头看一眼）。

```
➜  jvm git:(master) ✗ xxd Main.class
00000000: cafe babe 0000 0037 0016 0a00 0400 1209  .......7........
00000010: 0003 0013 0700 1407 0015 0100 0361 6765  .............age
00000020: 0100 0149 0100 063c 696e 6974 3e01 0003  ...I...<init>...
00000030: 2829 5601 0004 436f 6465 0100 0f4c 696e  ()V...Code...Lin
00000040: 654e 756d 6265 7254 6162 6c65 0100 124c  eNumberTable...L
00000050: 6f63 616c 5661 7269 6162 6c65 5461 626c  ocalVariableTabl
00000060: 6501 0004 7468 6973 0100 174c 636f 6d2f  e...this...Lcom/
00000070: 6974 7761 6e67 6572 2f6a 766d 2f4d 6169  itwanger/jvm/Mai
00000080: 6e3b 0100 0667 6574 4167 6501 0003 2829  n;...getAge...()
00000090: 4901 000a 536f 7572 6365 4669 6c65 0100  I...SourceFile..
000000a0: 094d 6169 6e2e 6a61 7661 0c00 0700 080c  .Main.java......
000000b0: 0005 0006 0100 1563 6f6d 2f69 7477 616e  .......com/itwan
000000c0: 6765 722f 6a76 6d2f 4d61 696e 0100 106a  ger/jvm/Main...j
000000d0: 6176 612f 6c61 6e67 2f4f 626a 6563 7400  ava/lang/Object.
000000e0: 2100 0300 0400 0000 0100 0200 0500 0600  !...............
000000f0: 0000 0200 0100 0700 0800 0100 0900 0000  ................
00000100: 3900 0200 0100 0000 0b2a b700 012a 1012  9........*...*..
00000110: b500 02b1 0000 0002 000a 0000 000a 0002  ................
00000120: 0000 0006 0004 0007 000b 0000 000c 0001  ................
00000130: 0000 000b 000c 000d 0000 0001 000e 000f  ................
00000140: 0001 0009 0000 002f 0001 0001 0000 0005  ......./........
00000150: 2ab4 0002 ac00 0000 0200 0a00 0000 0600  *...............
00000160: 0100 0000 0900 0b00 0000 0c00 0100 0000  ................
00000170: 0500 0c00 0d00 0000 0100 1000 0000 0200  ................
00000180: 11                                       .
```

对于这些 16 进制内容，除了开头的 cafe babe，剩下的内容大致可以翻译成：啥玩意啊这......

但经过上一节类文件结构的洗礼，相信大家对这份文件的内容已经很熟悉了。

javap

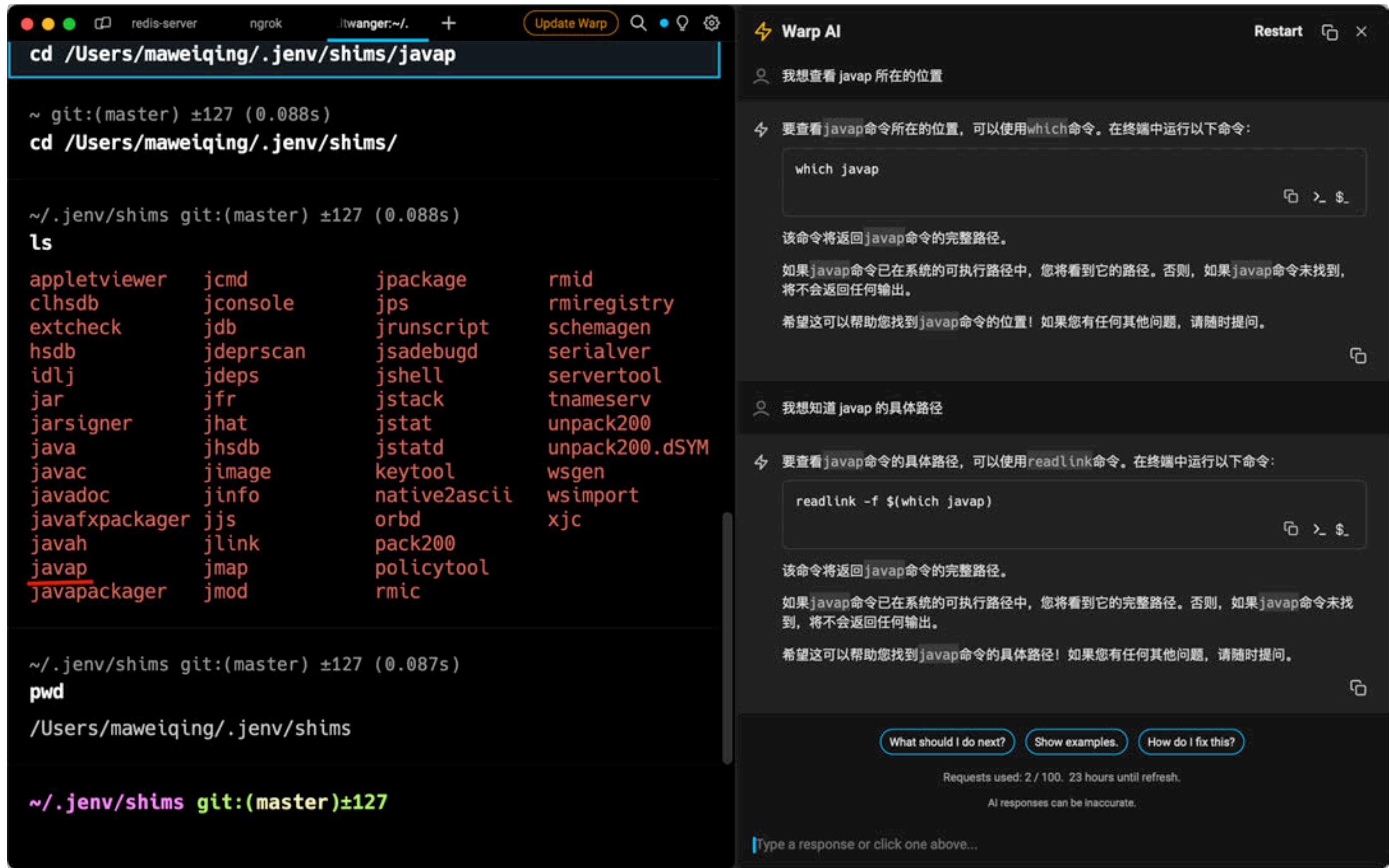
Java 内置了一个反编译命令 javap，可以通过 `javap -help` 了解 javap 的基本用法。

```
~ git:(master) ±127 (0.272s)
javap -help
用法: javap <options> <classes>
其中, 可能的选项包括:
  -help  --help  -?        输出此用法消息
  -version                 版本信息
  -v  -verbose             输出附加信息
  -l                       输出行号和本地变量表
  -public                  仅显示公共类和成员
  -protected               显示受保护的/公共类和成员
  -package                 显示程序包/受保护的/公共类
                           和成员 (默认)
  -p  -private             显示所有类和成员
  -c                       对代码进行反汇编
  -s                       输出内部类型签名
  -sysinfo                 显示正在处理的类的
                           系统信息 (路径, 大小, 日期, MD5 散列)
  -constants               显示最终常量
  -classpath <path>        指定查找用户类文件的位置
  -cp <path>               指定查找用户类文件的位置
  -bootclasspath <path>    覆盖引导类文件的位置
```

当然了，执行这个命令的前提条件是你需要配置好 Java 环境变量，如果没有配置好，可以参考这篇文章。

javap 是 JDK 自带的一个命令行工具，主要用于反编译类文件（.class 文件）。我本机是 macOS，使用了 jenv 来管理的 JDK 版本，所以看到的位置如下图所示。

> Windows 用户以及没有使用 jenv 的 macOS 用户可以根据这个帖子了解 jenv，真的好用。



javap 主要用于反编译 Java 类文件，即将编译后的 .class 文件转换回更易于理解的形式。虽然它不会生成原始的 Java 源代码，但它可以显示类的结构，包括构造方法、方法、字段等，帮助我们更好地理解 Java 字节码以及 Java 程序的运行机制。

前面我们已经写了一个简单的类，大家应该还记得：

```
public class Main {
    private int age = 18;
    public int getAge() {
        return age;
    }
}
```

当然了，我希望你是用 Intellij IDEA 来编写而不是记事本，这样就省去了我们主动编译的过程，可以直接在 target 目录下找到 class 文件，这些知识我们前面都已经讲过了。

OK，我们在 class 文件的同级目录下输入命令 `javap -v -p Main.class` 来查看一下输出的内容（-v 显示附加信息，如局部变量表、操作码等；-p 显示所有类和成员，包括私有的，不懂的同学可以回头看在看一眼 `javap -help` 的输出结果 😁）。

```
Classfile
/Users/maweiqing/Documents/GitHub/TechSisterLearnJava/codes/TechSister/target/classes/com/itwanger/jvm/Main.class
  Last modified 2021年4月15日; size 385 bytes
  SHA-256 checksum 6688843e4f70ae8d83040dc7c8e2dd3694bf10ba7c518a6ea9b88b318a8967c6
  Compiled from "Main.java"
public class com.itwanger.jvm.Main
```

```
  minor version: 0
  major version: 55
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #3                          // com/itwanger/jvm/Main
  super_class: #4                         // java/lang/Object
  interfaces: 0, fields: 1, methods: 2, attributes: 1
Constant pool:
   #1 = Methodref          #4.#18         // java/lang/Object."<init>":()V
   #2 = Fieldref           #3.#19         // com/itwanger/jvm/Main.age:I
   #3 = Class              #20            // com/itwanger/jvm/Main
   #4 = Class              #21            // java/lang/Object
   #5 = Utf8               age
   #6 = Utf8               I
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               LocalVariableTable
  #12 = Utf8               this
  #13 = Utf8               Lcom/itwanger/jvm/Main;
  #14 = Utf8               getAge
  #15 = Utf8               ()I
  #16 = Utf8               SourceFile
  #17 = Utf8               Main.java
  #18 = NameAndType        #7:#8          // "<init>":()V
  #19 = NameAndType        #5:#6          // age:I
  #20 = Utf8               com/itwanger/jvm/Main
  #21 = Utf8               java/lang/Object
{
  private int age;
    descriptor: I
    flags: (0x0002) ACC_PRIVATE

  public com.itwanger.jvm.Main();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: aload_0
         5: bipush        18
         7: putfield      #2                  // Field age:I
        10: return
      LineNumberTable:
        line 6: 0
        line 7: 4
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
```

```
            0      11     0  this   Lcom/itwanger/jvm/Main;

  public int getAge();
    descriptor: ()I
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: getfield      #2                  // Field age:I
         4: ireturn
      LineNumberTable:
        line 9: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/itwanger/jvm/Main;
}
SourceFile: "Main.java"
```

睁大眼睛瞧过去，内容挺多。同学们不要着急，我们来一行一行分析。

字节码的基本信息

第 1 行：

```
Classfile
/Users/maweiqing/Documents/GitHub/TechSisterLearnJava/codes/TechSister/target/classes/c
om/itwanger/jvm/Main.class
```

顾名思义，这行表示字节码文件的位置。

第 2 行：

```
Last modified 2021年4月15日; size 385 bytes
```

字节码文件的修改日期（我 2021 年在「沉默王二」公众号里分享过，不知道还有多少同学记得 😄）、文件大小是 385 bytes。

第 3 行：

```
SHA-256 checksum 6688843e4f70ae8d83040dc7c8e2dd3694bf10ba7c518a6ea9b88b318a8967c
```

字节码文件的 SHA-256 值，用于校验文件的完整性。

> SHA-256 是一种加密哈希算法，将任意长度的输入数据处理成固定长度（256 位，即 32 字节）的输出数据，且输出数据的哈希值在数学上很难被反向计算出原始数据，所以常用于校验数据的完整性。

第 4 行：

```
Compiled from "Main.java"
```

说明该字节码文件编译自 Main.java 源文件。

第 5 行：

```
public class com.itwanger.jvm.Main
```

类访问修饰符和类型，表明这是一个公开的类，名为 `com.itwanger.jvm.Main`。

第 6 行 `minor version: 0`，次版本号。

第 7 行 `major version: 55`，主版本号（由 Java 11 编译，上一节讲过）。

第 8 行：

```
flags: (0x0021) ACC_PUBLIC, ACC_SUPER
```

类访问标记，一共有 8 种，上一节我们曾提到。

标记名	标记值	释义
ACC_PUBLIC	0x0001	是否为 public
ACC_FINAL	0x0010	是否为 final
ACC_SUPER	0x0020	~~为了兼容旧版本，Java 1.1及更高版本始终设置此值~~
ACC_INTERFACE	0x0200	是否为一个接口
ACC_ABSTRACT	0x0400	是否为抽象的，接口和抽象类是的，其他不是
ACC_SYNTHETIC	0x1000	这个类的代码并非用户代码
ACC_ANNOTATION	0x2000	是否为注解
ACC_ENUM	0x4000	是否为枚举

表明当前类是 `ACC_PUBLIC | ACC_SUPER`（表明这个类是 public 的，并且使用了 super 关键字）。

位运算符 `|` 的意思是如果相对应位是 0，则结果为 0，否则为 1，所以 `0x0001 | 0x0020` 的结果是 `0x0021`（需要转成二进制进行运算）。

第 9 行：

```
this_class: #3 // com/itwanger/jvm/Main
```

当前类的索引，指向常量池中下标为 3 的常量（上一节刚讲过），可以看得出当前类是 Main 类。

第 10 行：

```
super_class: #4 // java/lang/Object
```

父类的索引，指向常量池中下标为 6 的常量，可以看得出当前类的父类是 Object 类（所有没有明确父类都默认继承超类，这也是**万物皆对象**的重要原因）。

第 11 行：

```
interfaces: 0, fields: 1, methods: 2, attributes: 1
```

当前类有 0 个接口，1 个字段（age），2 个方法（`write()`方法和缺省的默认构造方法，讲《面向对象编程》的时候都讲过），1 个属性（该类仅有的一个属性是 SourceFIle，包含了源码文件的信息，第一行讲过了）。

常量池

接下来是 Constant pool，也就是字节码文件最重要的常量池部分。可以把常量池理解为字节码文件中的资源仓库，主要存放两大类信息。

> 上一节我们就讲过字面量和符号引用，这里再讲一次，应该是第三次讲了，确实比较难懂，我们就多讲几次，直到大家都能理解为止（😁）。

1）字面量（Literal），有点类似 Java 中的常量概念，比如文本字符串，final 常量等。

2）符号引用（Symbolic References），属于编译原理方面的概念，包括 3 种：

- 类和接口的全限定名（Fully Qualified Name）
- 字段的名称和描述符（Descriptor）
- 方法的名称和描述符

Java 虚拟机是在加载字节码文件的时候才进行的动态链接，也就是说，字段和方法的符号引用只有经过运行期转换后才能获得真正的内存地址。

当 Java 虚拟机运行时，需要从常量池获取对应的符号引用，然后在类创建或者运行时解析并翻译到具体的内存地址上。

当前字节码文件中一共有 21 个常量，它们之间是有链接的，逐个分析会比较乱，我们采用顺藤摸瓜的方式，从上依次往下看，那些被链接的常量我们就点到为止。

注：

- `#` 号后面跟的是索引，索引没有从 0 开始而是从 1 开始，是因为设计者考虑到，“如果要表达不引用任何一个常量的含义时，可以将索引值设为 0 来表示”（周志明老师《深入理解 Java 虚拟机》一书描述的）。
- `=` 号后面跟的是常量的类型，没有包含前缀 `CONSTANT_` 和后缀 `_info`。
- **全文中提到的索引等同于下标**，为了灵活描述，没有做统一。

好，开始。

第 1 个常量：

```
#1 = Methodref #4.#18 // java/lang/Object."<init>":()V
```

类型为 Methodref，表明是用来定义方法的，指向常量池中下标为 4 和 18 的常量。

第 4 个常量：

```
#4 = Class #21 // java/lang/Object
```

类型为 Class，表明是用来定义类（或者接口）的，指向常量池中下标为 21 的常量。

第 21 个常量：

```
#21 = Utf8 java/lang/Object
```

类型为 Utf8，UTF-8 编码的字符串，值为 `java/lang/Object`。

第 18 个常量：

```
#18 = NameAndType #7:#8 // "<init>":()V
```

类型为 NameAndType，表明是字段或者方法的部分符号引用，指向常量池中下标为 7 和 8 的常量。

第 7 个常量：

```
#7 = Utf8 <init>
```

类型为 Utf8，UTF-8 编码的字符串，值为 `<init>`，表明为构造方法。

第 8 个常量：

```
#8 = Utf8 ()V
```

类型为 Utf8，UTF-8 编码的字符串，值为 `()V`，表明方法的返回值为 void。

到此为止，第 1 个常量算是摸完了。组合起来的意思就是，Main 类使用的是默认的构造方法，来源于 Object 类。`#4` 指向 `Class #21`（即 `java/lang/Object`），`#18` 指向 `NameAndType #7:#8`（即 `<init>:()V`）。

第 2 个常量：

```
#2 = Fieldref #3.#19 // com/itwanger/jvm/Main.age:I
```

类型为 Fieldref，表明是用来定义字段的，指向常量池中下标为 3 和 19 的常量。

第 3 个常量：

```
#3 = Class #20 // com/itwanger/jvm/Main
```

类型为 Class，表明是用来定义类（或者接口）的，指向常量池中下标为 20 的常量。

第 19 个常量：

```
#19 = NameAndType #5:#6 // age:I
```

类型为 NameAndType，表明是字段或者方法的部分符号引用，指向常量池中下标为 5 和 6 的常量。

第 5 个常量：

```
#5 = Utf8 age
```

类型为 Utf8，UTF-8 编码的字符串，值为 `age`，表明字段名为 age。

第 6 个常量：

```
#6 = Utf8               I
```

类型为 Utf8，UTF-8 编码的字符串，值为 `I`，表明字段的类型为 int。

关于字段类型的描述符映射表如下图所示，上一节其实也讲过，只不过是从 16 进制来看的，这一节是从 javap 的角度来看的。

标识字符	含义
B	基本数据类型 byte
C	基本数据类型 char
D	基本数据类型 double
F	基本数据类型 float
I	基本数据类型 int
J	基本数据类型 long
S	基本数据类型 short
Z	基本数据类型 boolean
V	特殊类型 void
L	引用数据类型，以分号“;”结尾
[	一维数组

到此为止，第 2 个常量算是摸完了。组合起来的意思就是，声明了一个类型为 int 的字段 age。`#3` 指向 `Class #20`（即 `com/itwanger/jvm/Main`），`#19` 指向 `NameAndType #5:#6`（即 `age:I`）。

字段表集合

字段表用来描述接口或者类中声明的变量，包括类变量和成员变量，但不包含声明在方法中局部变量。

> 带链接的都是我们之前讲过的，是不是发现所有的知识都串联起来了？这就是我们学习 javap 和字节码的原因，了解字节码的同时，也能够加深对 Java 知识的理解。

字段的修饰符一般有：

- 访问权限修饰符，比如 public private protected
- 静态变量修饰符，比如 static

- final 修饰符
- 并发可见性修饰符，比如 volatile
- 序列化修饰符，比如 transient

然后是字段的类型（可以是基本数据类型、数组和对象）和名称。

在 Main.class 字节码文件中，字段表的信息如下所示。

```
private int age;
    descriptor: I
    flags: (0x0002) ACC_PRIVATE
```

表明字段的访问权限修饰符为 private，类型为 int，名称为 age。字段的访问标志和类的访问标志非常类似。

标记名	标记值	释义
ACC_PUBLIC	0x0001	是否为 public
ACC_PRIVATE	0x0002	是否为 private
ACC_PROTECTED	0x0004	是否为 protected
ACC_STATIC	0x0008	是否为 static
ACC_FINAL	0x0010	是否为 final
ACC_VOLATILE	0x0040	是否为 volatile
ACC_TRANSIENT	0x0080	是否序列化
ACC_SYNTHETIC	0x1000	是否由编译器自动生成
ACC_ENUM	0x4000	是否为枚举

方法表集合

方法表用来描述接口或者类中声明的方法，包括类方法和成员方法，以及构造方法。方法的修饰符和字段略有不同，比如说 volatile 和 transient 不能用来修饰方法，再比如说方法的修饰符多了 synchronized、native、strictfp 和 abstract。

标记名	标记值	释义
ACC_PUBLIC	0x0001	是否为 public
ACC_PRIVATE	0x0002	是否为 private
ACC_PROTECTED	0x0004	是否为 protected
ACC_STATIC	0x0008	是否为 static
ACC_FINAL	0x0010	是否为 final
ACC_SYNCHRONIZED	0x0020	是否为同步的
ACC_NATIVE	0x0100	是否为本地方法
ACC_SYNTHETIC	0x1000	是否由编译器自动生成
ACC_ABSTRACT	0x0400	是否为抽象方法
ACC_STRICT	0x0800	限制浮点计算以确保可移植性
ACC_VARARGS	0x0080	是否接受可变参数

构造方法

下面这部分为构造方法，返回类型为 void，访问标志为 public。

```
public com.itwanger.jvm.Main();
  descriptor: ()V
  flags: (0x0001) ACC_PUBLIC
```

- 声明：`public com.itwanger.jvm.Main();` 这是 Main 类的构造方法，用于创建 Main 类的实例。它是公开的（public）。
- 描述符：`descriptor: ()V`
  这表示构造方法没有参数 (`()`) 并且没有返回值 （`V`，代表 `void`）。
- 访问标志：`flags: (0x0001) ACC_PUBLIC`，表示这个构造方法是公开的，可以从其他类中访问。

来详细看一下其中 Code 属性。

```
Code:
  stack=2, locals=1, args_size=1
      0: aload_0
      1: invokespecial #1                  // Method java/lang/Object."<init>":()V
      4: aload_0
```

```
      5: bipush        18
      7: putfield      #2                  // Field age:I
     10: return
   LineNumberTable:
     line 6: 0
     line 7: 4
   LocalVariableTable:
     Start  Length  Slot  Name   Signature
         0      11     0  this   Lcom/itwanger/jvm/Main;
```

①、stack 为最大操作数栈，Java 虚拟机在运行的时候会根据这个值来分配栈帧的操作数栈深度（关于操作数栈和栈帧，我们会在下一节详细讲解），这里的值为 2，意味着操作数栈的深度为 2。

操作栈是一个 LIFO（后进先出）栈，用于存放临时变量和中间结果。在构造方法中，bipush 和 aload_0 指令可能会同时需要栈空间，所以需要 2 个操作数栈深度。

②、locals 为局部变量所需要的存储空间，单位为槽（slot），方法的参数变量和方法内的局部变量都会存储在局部变量表中。

局部变量表的容量以变量槽为最小单位，一个变量槽可以存放一个 32 位以内的数据类型，比如 boolean、byte、char、short、int、float、reference 和 returnAddress 类型。

局部变量表所需的容量大小是在编译期间完成计算的，大小由编译器决定，因此不同的编译器编译出来的字节码可能会不一样。

locals=1，这表示局部变量表中有 1 个变量的空间。对于实例方法（如构造方法），局部变量表的第一个位置（索引 0）总是用于存储 this 引用。

③、args_size 为方法的参数个数。

为什么 stack 的值为 2，locals 的值为 1，args_size 的值为 1 呢？**默认的构造方法不是没有参数和局部变量吗**？

这是因为有一个隐藏的 this 变量，只要不是静态方法，都会有一个当前类的对象 this 悄悄的存在着。

这就解释了为什么 locals 和 args_size 的值为 1 的问题。

那为什么 stack 的值为 2 呢？因为字节码指令 invokespecial（调用父类的构造方法进行初始化）会消耗掉一个当前类的引用，所以 aload_0 执行了 2 次，也就意味着操作数栈的大小为 2。

关于字节码指令，我们后面会详细介绍，这里只是简单提一下。

④、LineNumberTable，该属性的作用是描述源码行号与字节码行号(字节码偏移量)之间的对应关系。这对于调试非常重要，因为它允许调试器将正在执行的字节码指令精确地关联到源代码的特定行。

```
LineNumberTable:
  line 6: 0
  line 7: 4
```

这里的意思是，第 6 行对应的字节码行号为 0，第 7 行对应的字节码行号为 4。

在调试过程中，当一个断点被触发或出现异常时，通过 LineNumberTable，我们可以知道这是源代码中的哪一行导致的。

④、LocalVariableTable，该属性的作用是描述帧栈中的局部变量与源码中定义的变量之间的关系。大家仔细看一下，就能看到 this 的影子了。

- Start 和 Length：定义变量在方法中的作用域。Start 是变量生效的字节码偏移量，Length 是它保持活动的长度。
- Slot：变量在局部变量数组中的索引。
- Name：变量的名称，如在源代码中定义的。
- Signature：变量的类型描述符。

这里，只有一个局部变量 this，它指代构造方法正在初始化的对象。它的作用域是从指令偏移量 0 开始，持续整个方法的长度（长度为 11），并且被分配到局部变量表的第一个槽位（索引 0）。`Lcom/itwanger/jvm/Main;` 表明这个变量的类型是 com.itwanger.jvm.Main。

成员方法

下面这部分为成员方法 `getAge()`，返回类型为 int，访问标志为 public。

```
public int getAge();
  descriptor: ()I
  flags: (0x0001) ACC_PUBLIC
```

理解了构造方法的 Code 属性后，再看 `getAge()` 方法的 Code 属性时，就很容易理解了。

```
Code:
  stack=1, locals=1, args_size=1
      0: aload_0
      1: getfield      #2                  // Field age:I
      4: ireturn
  LineNumberTable:
    line 9: 0
  LocalVariableTable:
    Start  Length  Slot  Name   Signature
        0       5     0  this   Lcom/itwanger/jvm/Main;
```

最大操作数栈为 1，局部变量所需要的存储空间为 1，方法的参数个数为 1，是因为局部变量只有一个隐藏的 this，并且字节码指令中只执行了一次 aload_0。

①、字节码指令

- aload_0: 加载 this 引用到栈顶，以便接下来访问实例字段 age。
- `getfield #2`: 获取字段值。这条指令读取 this 对象的 age 字段的值，并将其推送到栈顶。`#2` 是对常量池中的字段引用。
- ireturn: 返回栈顶整型值。这里返回的是 age 字段的值。

②、附加信息

LineNumberTable 和 LocalVariableTable 同样提供了源代码的行号对应和局部变量信息，有助于调试和理解代码的执行流程。

小结

其实学习是这样的，可以横向扩展，也可以纵向扩展。当我们初学编程的时候，特别想多学一点，属于横向扩展，当有了一定的编程经验后，想更上一层楼，就需要纵向扩展，不断深入地学，连根拔起，从而形成自己的知识体系。

无论是从十六进制的字节码角度，还是 jclasslib 图形化查看反编译后的字节码的角度，也或者是今天这样从 javap 反编译后的角度，都能窥探出一些新的内容来！

初学者一开始接触字节码的时候会感觉比较头大，没关系，我当初也是这样，随着时间的推移，经验的积累，慢慢就好了，越往深处钻，就越能体会到那种“技术我有，雄霸天下”的感觉~

第六节：栈虚拟机和寄存器虚拟机

本来这节内容是打算直接讲字节码指令的，但讲之前又必须得先讲指令集架构，而指令集架构又分为两种，一种是基于栈的，一种是基于寄存器的。

那不妨我们这节就单独来讲讲栈虚拟机和寄存器虚拟机，它们有什么不同，以及各自的优缺点。

栈和寄存器

栈（stack），有些地方喜欢称呼它为堆栈，我就很不喜欢，很容易和 heap（堆）搞混，尤其是对于新手来说，简直就是虐心。

栈是一种非常有用的数据结构，它就像一摞盘子，第一个放在最下面，第二个放在第一个上面，第三个放在第二个上面，最后一个放在最上面。

栈就好像一摞盘子

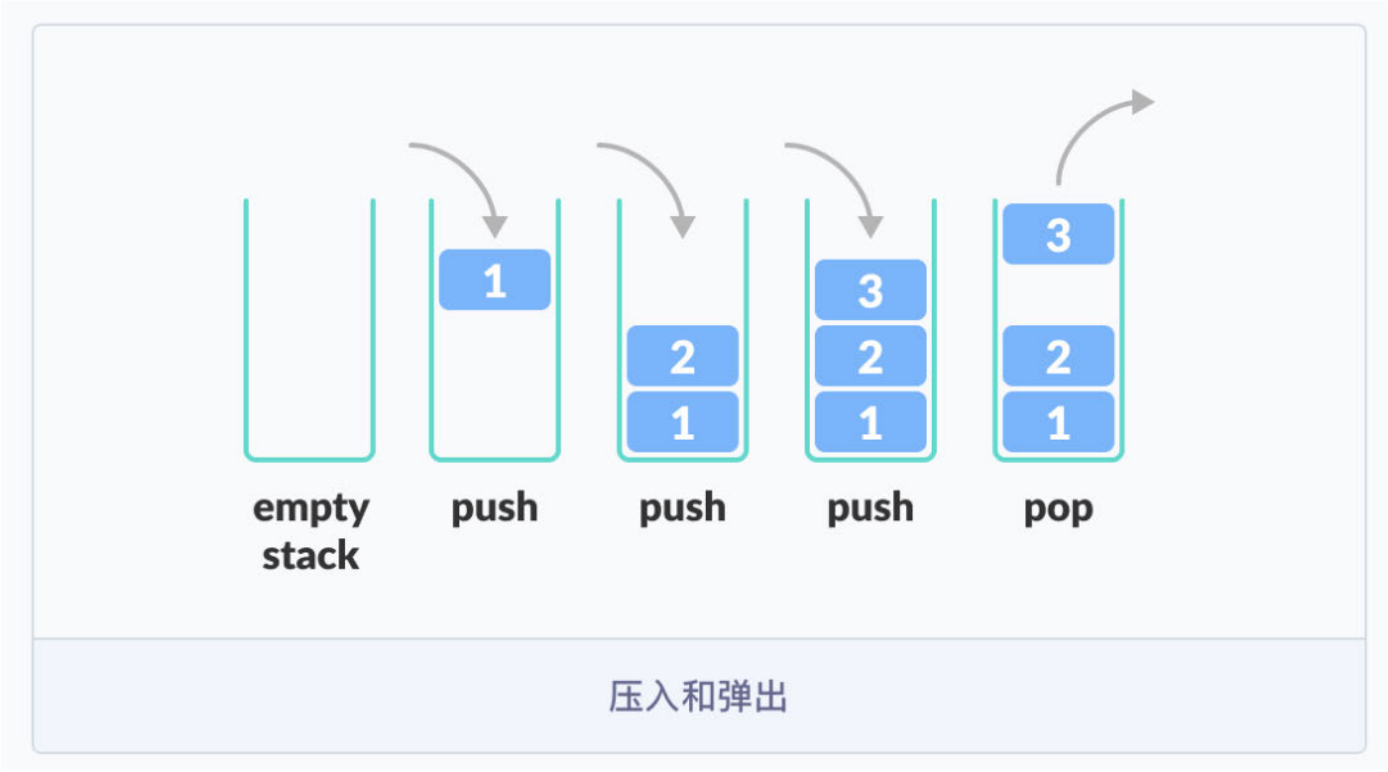
对于这一摞盘子，我们可以做两件事情：

- 在最上面放一个新盘子
- 把顶部的盘子拿走

这两件事情做起来很容易，但如果从中间或者底部抽出来一个盘子，就很难办到。如果我们想要拿到最下面的盘子，就必须把它上面的所有盘子都拿走，像这样的一个操作，我们称之为后进先出，也就是“Last In First Out”（简称 LIFO）——最后的一个进的，最先出去。

对于栈这样一个数据结构来说，它有两个常见的动作：

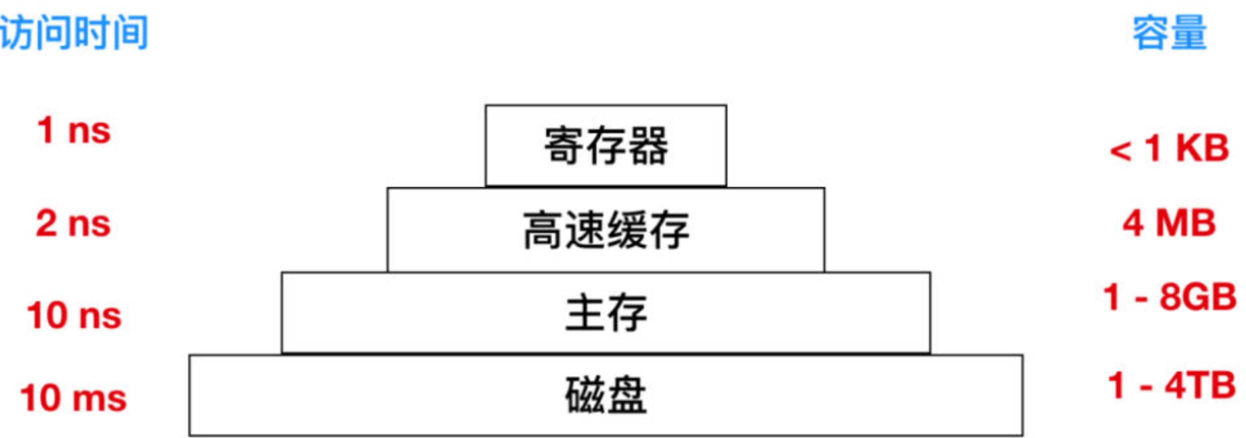
- push，中文释义有很多种，我个人更喜欢叫它“压入”，非常形象。当我们要把一个元素放入栈的顶部，这个动作就叫做 push。
- pop，同样的，我个人更喜欢叫它“弹出”，带有很强烈的动画效果，有没有？当我们要从栈中移除一个元素时，这个动作就叫做 pop。



压入和弹出

对于上面这幅图来说，3 这个元素最后放进去，却是最先被移除的——遵循 LIFO 的原则。

寄存器（Register）是中央处理器（CPU）内用来暂存指令、数据和地址的存储器，也是 CPU 中读写最快的存储器。

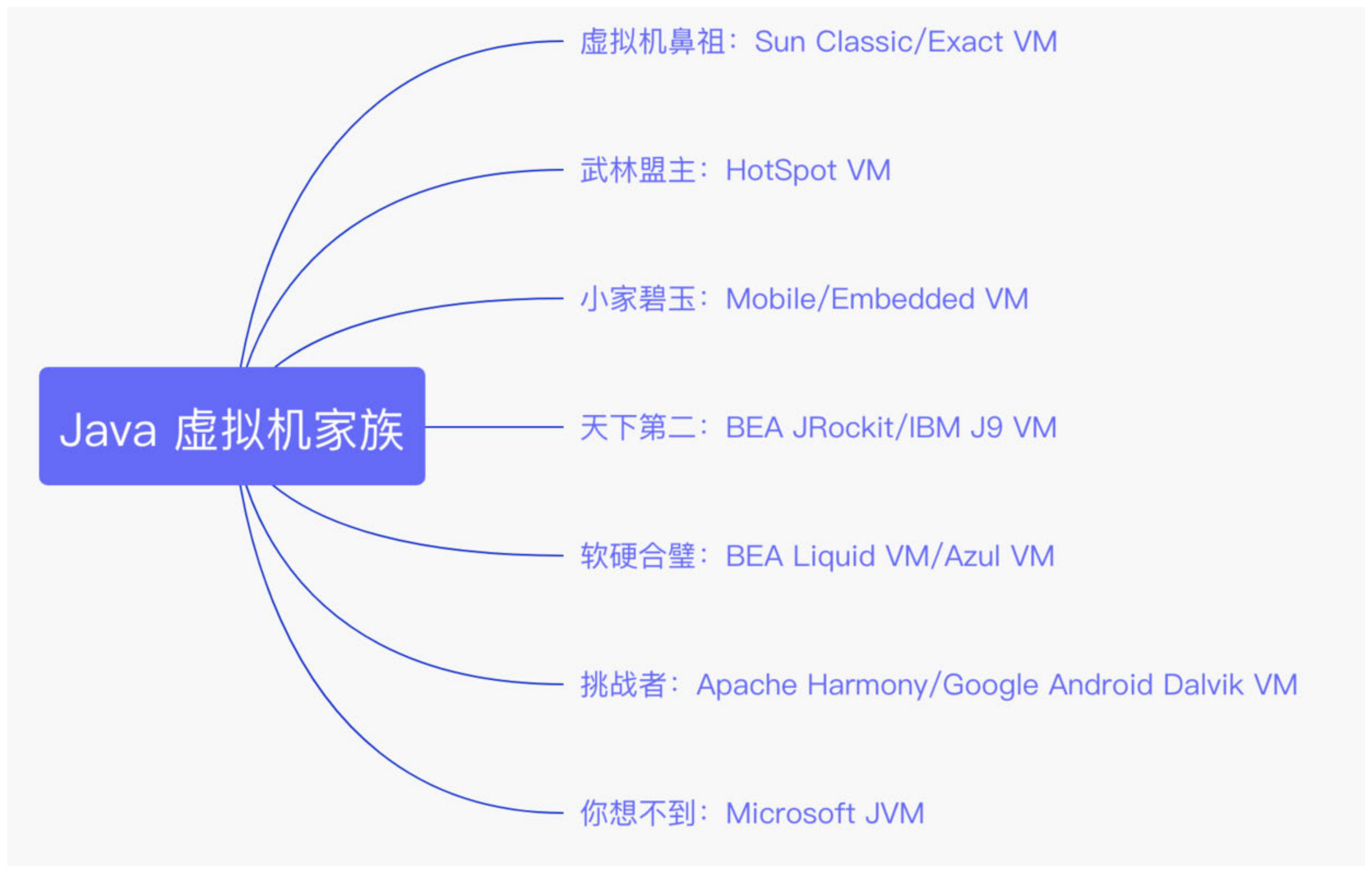


存储层次结构

从硬件层面来说，栈位于内存当中，而寄存器位于 CPU 当中，这也是为什么，我们通常会说，基于寄存器架构的虚拟机会比基于栈的虚拟机快的原因。

基于栈的虚拟机

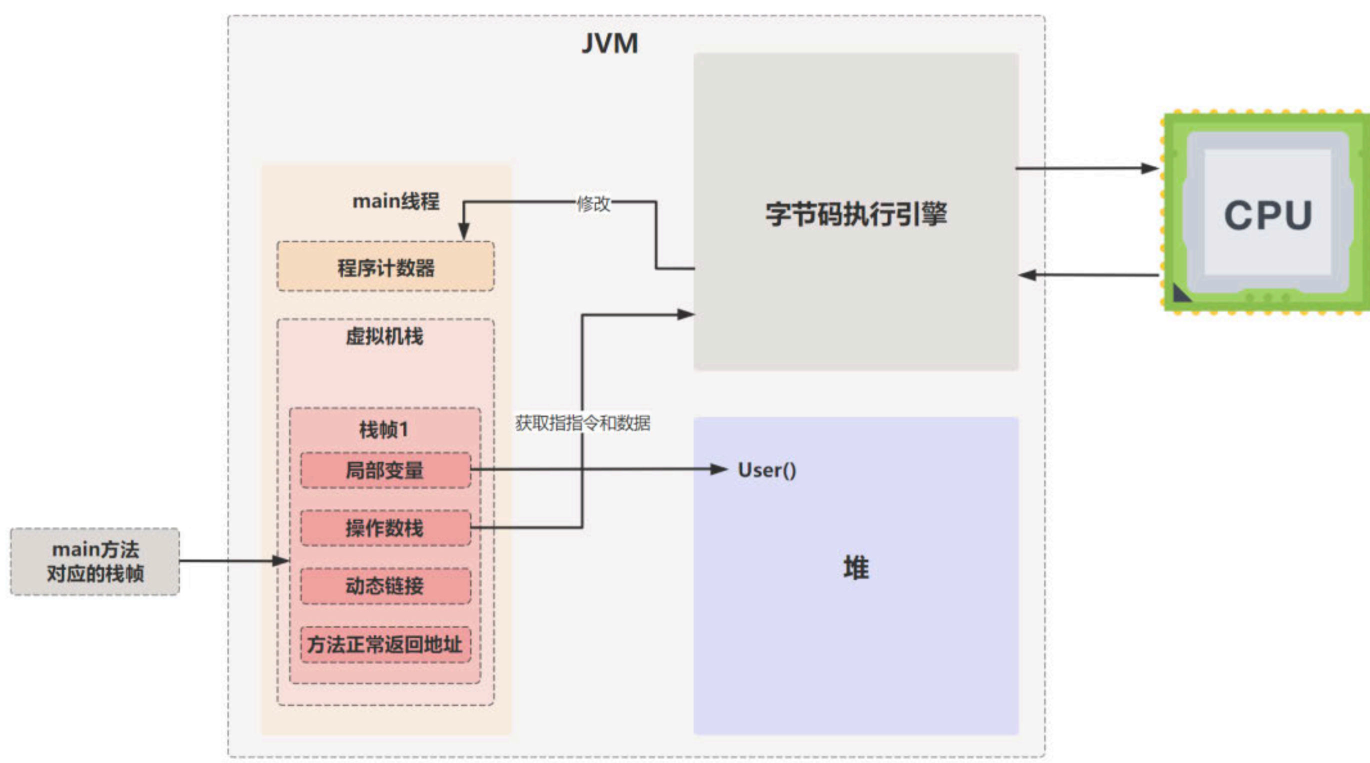
前面我们讲 JDK 的发展历程时，提到了 Hotspot VM，它是血缘最正统的 Java 虚拟机。



HotSpot VM 是基于栈的一种虚拟机，当 Java 程序运行时，HotSpot VM 加载编译后的字节码文件（也就是.class 文件），其解释器或JIT编译器会读取文件中的字节码指令，将它们解释（或编译）为机器码。

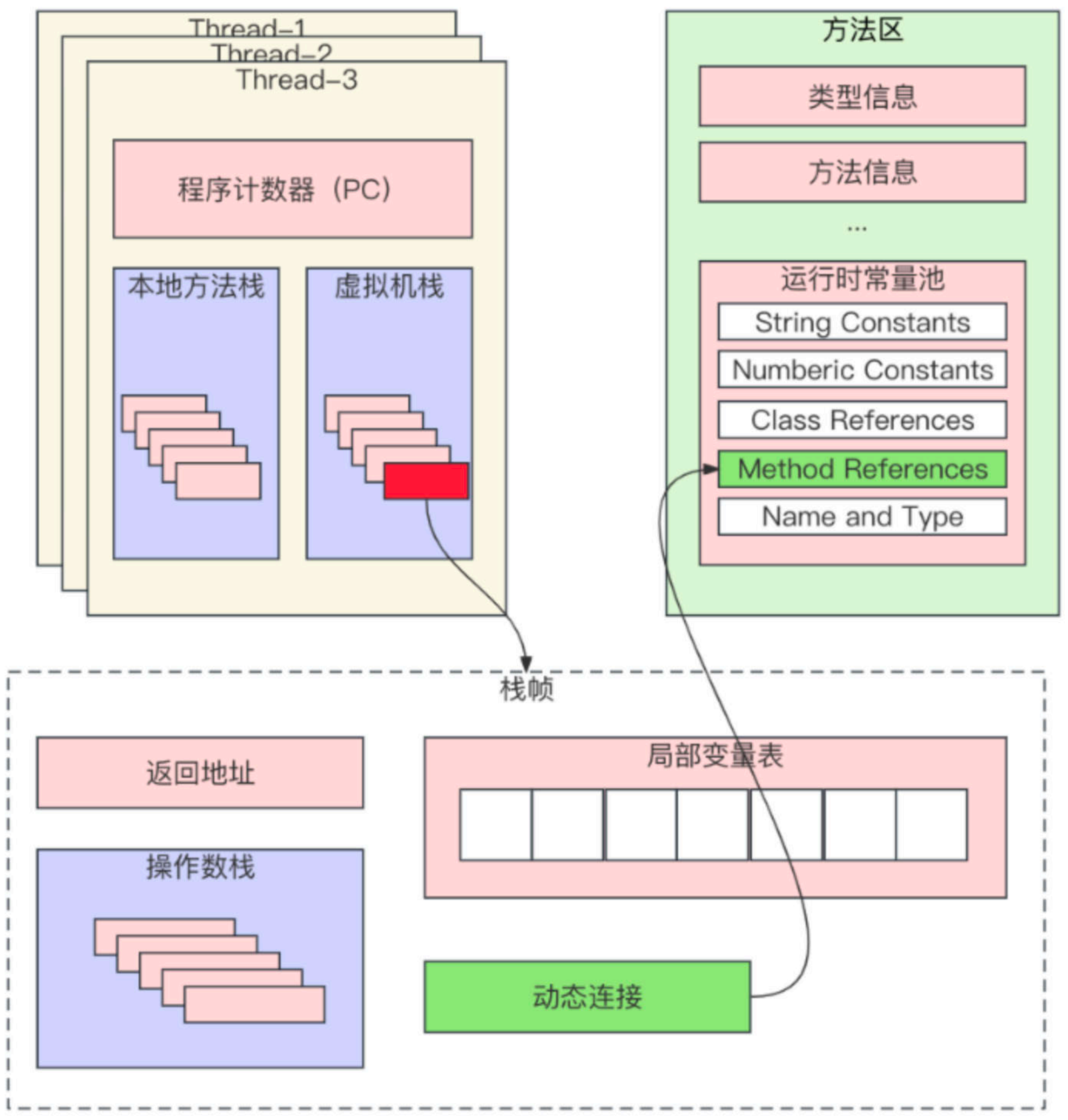
方法调用和执行过程中的数据（如局部变量和中间结果）会存储在栈（操作数栈，下面会讲）中，字节码指令操作这些数据，然后执行程序逻辑。

下面这幅图我们之前在讲JVM 是如何运行 Java 代码的时候讲过。



main 方法被执行的时候，JVM 会创建一个栈帧（Stack Frame），通过存储局部变量表、操作数栈、动态链接、方法出口等信息来支撑和完成方法的执行，栈帧就是虚拟机栈中的子单位。

栈帧本身也是一种栈结构，用于支持虚拟机进行方法调用和方法执行，遵循 LIFO 的原则，每个栈帧都包含了一个方法的运行信息，每个方法从调用到执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈和出栈的过程。



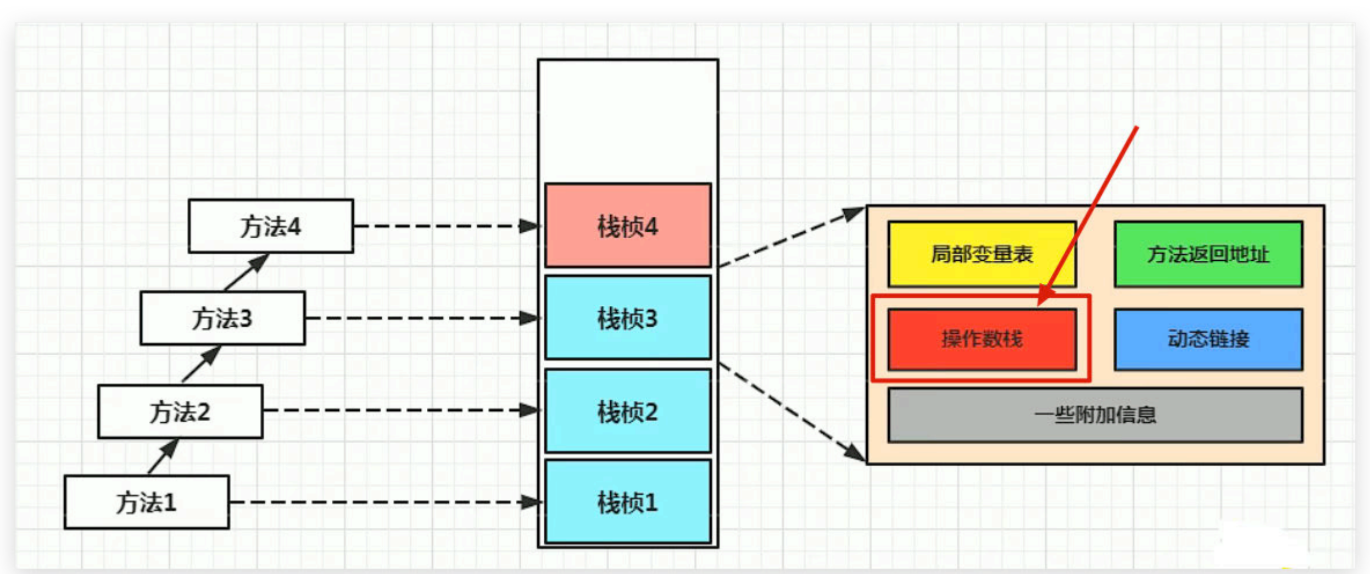
虚拟机栈是线程私有的，每个线程都有自己的 Java 虚拟机栈。方法调用时都会创建一个新的栈帧，该栈帧被推入虚拟机栈，成为当前活动栈帧。

- 入栈：方法调用时，虚拟机栈会为这个方法分配一个栈帧，这个栈帧被压入虚拟机栈，成为当前的活动栈帧。PC 寄存器指向当前栈帧的指令，执行方法的指令序列从该地址开始。
- 出栈：方法执行完成后，对应的栈帧会被移除，控制权回到前一个栈帧，前一个栈帧中的返回值成为当前活动栈帧的一个操作数，继续执行。

其中的操作数栈（Operand Stack）也是一种栈结构，用于保存方法执行时的中间结果、参数和返回值。当一个方法刚刚开始执行的时候，这个方法的操作数栈是空的。

在方法执行的过程中，操作数栈被用于执行各种字节码指令。例如，将两个数字相加的指令会从操作数栈中弹出两个数字，将它们相加，然后将结果压入操作数栈中。

另外，操作数栈的内容是临时的，它的生命周期和方法的生命周期是一样的，当方法执行结束后，操作数栈也会被销毁。



R 大曾在知乎的贴子里提到过：

> VM 当初设计的时候非常重视代码传输和存储的开销，因为假定的应用场景是诸如手持设备、机顶盒之类的嵌入式应用，所以要代码尽量小；外加基于栈的实现更简单（无论是在源码编译器的一侧还是在虚拟机的一侧），而且主要设计者 James Gosling 的个人经验上也对这种做法非常熟悉（例如他之前实现过 PostScript 的虚拟机，也是基于栈的指令集），所以就选择了基于栈。

我们简单来看一下基于栈的虚拟机方法执行的过程，以下面的代码为例：

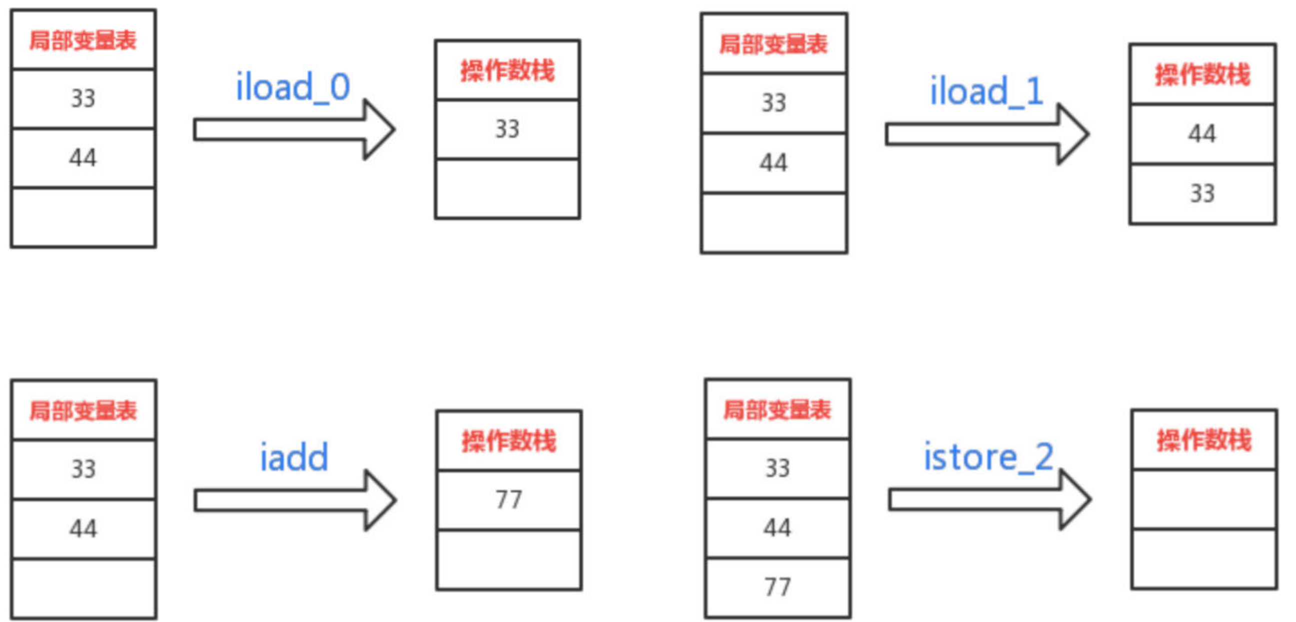
```
int a = 33;
int b = 44;
int c = a + b;
```

通过 `javap -c Main` 命令可以查看对应的字节码，如下所示：

```
Compiled from "Main.java"
public class com.github.paicoding.forum.test.javabetter.jvm.Main {
  public static void main(java.lang.String[]);
    Code:
       0: bipush        33
       2: istore_1
       3: bipush        44
       5: istore_2
       6: iload_1
       7: iload_2
       8: iadd
```

```
         9: istore_3
        10: return
}
```

我们用图来说明指令执行的过程，大致如下。



- iload_0 将 33 压入操作数栈中
- iload_1 将 44 压入操作数栈中
- iadd 将操作数栈中的 33 和 44 弹出，相加后将结果 77 压入操作数栈中
- istore_2 将栈顶的 77 弹出，存入局部变量表中下标为 2 的位置

关于字节码指令的具体释义，我们放到下一节去细讲，这里主要是带大家体会一下基于栈的虚拟机和基于寄存器的虚拟机之间的差别。

基于寄存器的虚拟机

那除了有基于栈的虚拟机实现，当然也有基于寄存器的虚拟机实现，比如 LuaVM，负责执行 Lua 语言，一门轻量级的脚本语言，可戳链接了解。

> 5.0 之前的 Lua 其实是用基于栈的指令集，到 5.0 才改为用基于寄存器的。出于两点考虑，一是减少数据移动次数，降低数据迁移带来的拷贝开销；二是减少虚拟指令条数，提高指令执行效率。

About

português · news · showcase · uses · quotes · press · authors · contact · license · versions · thanks · getting started

❖ What is Lua?

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode with a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

❖ Where does Lua come from?

Lua is designed, implemented, and maintained by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. Lua was born and raised in Tecgraf, formerly the Computer Graphics Technology Group of PUC-Rio. Lua is now housed at LabLua, a laboratory of the Department of Computer Science of PUC-Rio.

❖ What's in a name?

"Lua" (pronounced **LOO-ah**) means "Moon" in Portuguese. As

❖ Why choose Lua?

Lua is a proven, robust language

Lua has been used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems (e.g., the Ginga middleware for digital TV in Brazil) and games (e.g., World of Warcraft and Angry Birds). Lua is currently the leading scripting language in games. Lua has a solid reference manual and there are several books about it. Several versions of Lua have been released and used in real applications since its creation in 1993. Lua featured in HOPL III, the Third ACM SIGPLAN History of Programming Languages Conference, in 2007. Lua won the Front Line Award 2011 from the Game Developers Magazine.

Lua is fast

Lua has a deserved reputation for performance. To claim to be "as fast as Lua" is an aspiration of other scripting languages. Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages. Lua is fast not only in fine-tuned benchmark programs, but in real life too. Substantial fractions of large applications have been written in Lua.

If you need even more speed, try LuaJIT, an independent implementation of Lua using a just-in-time compiler.

好，我们就基于 lua 来看一下基于寄存器的虚拟机方法执行的过程。

第一步，安装 lua，这里我用的是 macOS，直接用 brew 安装就好了。

```
brew install lua
```

> Windows 用户可以查看这个文档：http://lua-users.org/wiki/BuildingLuaInWindowsForNewbies

也可以通过 Lua for Windows 来完成安装：

> https://github.com/rjpcomputing/luaforwindows/releases

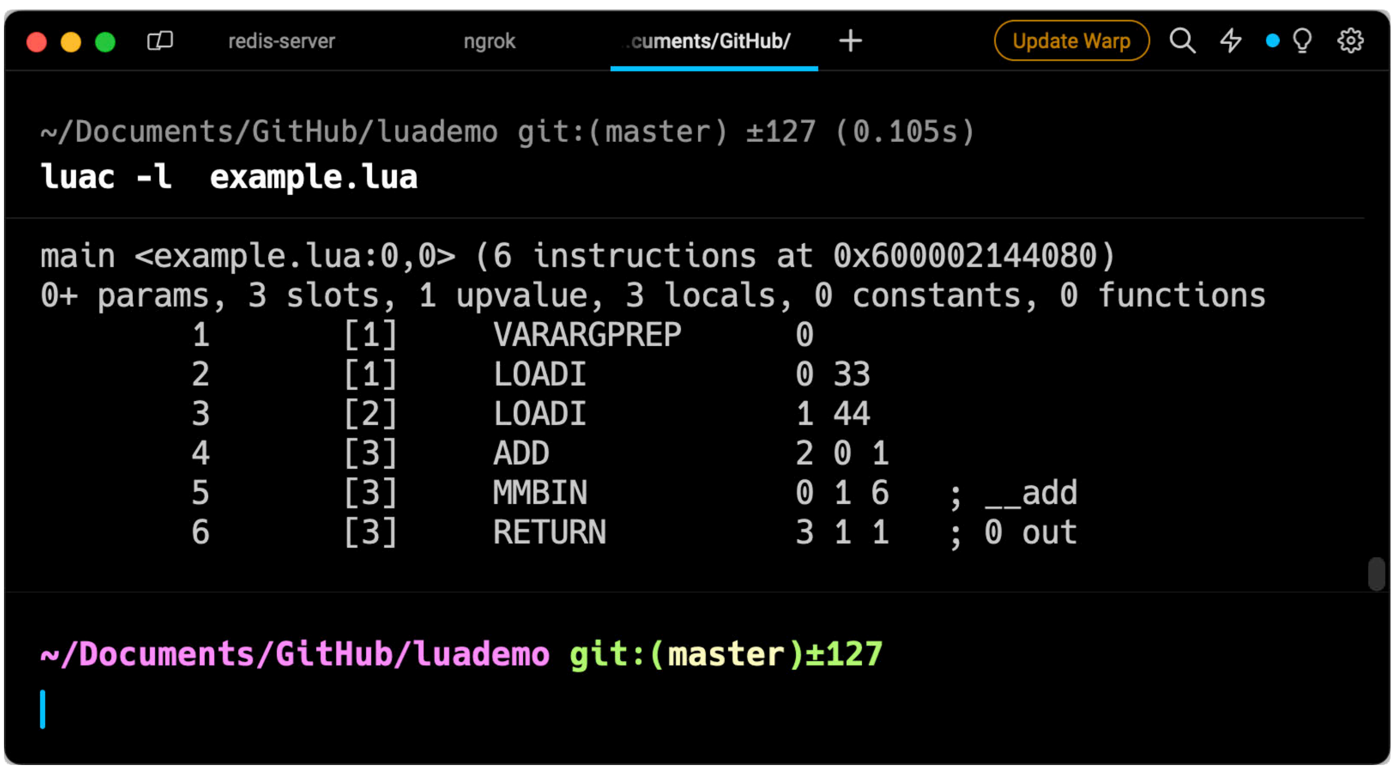
我们来编写一段简单的 lua 代码，保存为 example.lua。

```
local a = 33
local b = 44
local c = a + b
```

然后查看字节码指令。

```
luac -l example.lua
```

结果如下：




```
main <example.lua:0,0> (6 instructions at 0x600002144080)
0+ params, 3 slots, 1 upvalue, 3 locals, 0 constants, 0 functions
```

这是函数的描述，表示这是 example.lua 文件中的主函数。它包含 6 条指令。函数不接受参数（0+ params），有 3 个本地变量槽位（3 slots），1 个闭包变量（1 upvalue），3 个本地变量（3 locals），没有常量（0 constants）和内部函数（0 functions）。

接下来是具体的指令：

1. VARARGPREP 0：准备变长参数，用于处理传入的参数。
2. LOADI 0 33：将整数 33 加载到寄存器 0。
3. LOADI 1 44：将整数 44 加载到寄存器 1。
4. ADD 2 0 1：将寄存器 0 和寄存器 1 中的值相加，并将结果存放在寄存器 2。对应于脚本中两个数值的加法操作。
5. MMBIN 0 1 6; add：这是一个元方法（metamethod）调用，用于处理加法操作。这指示 Lua 虚拟机查找并执行 `add` 元方法。元方法是 Lua 中用于重载标准操作符的特殊方法。
6. RETURN 3 1 1; 0 out：返回操作，将寄存器 3 中的值作为返回值。`1 1` 表示从寄存器 3 返回一个值，`0 out` 指没有额外的返回值。

小结

基于栈的优点是可移植性更好、指令更短、实现起来简单，但不能随机访问栈中的元素，完成相同功能所需要的指令数也比寄存器的要多，需要频繁的入栈和出栈。

基于寄存器的优点是速度快，有利于程序运行速度的优化，但操作数需要显式指定，指令也比较长。

第七节：字节码指令详解

大家好，我是二哥呀。字节码指令是 JVM 体系中比较难啃的一块硬骨头，我估计有些球友会有这样的疑惑，“这么难啃，我还能学会啊？”

讲良心话，不是我谦虚，一开始学 Java 字节码和 Java 虚拟机方面的知识我也头大！但硬着头皮学了一阵子之后，突然就开窍了，觉得好有意思，尤其是明白了 Java 代码在底层竟然是这样执行的时候，感觉既膨胀又飘飘然，浑身上下散发着自信的光芒！

来吧，跟着二哥一起来学习吧，别畏难。前面我们已经讲过了，JVM 是基于栈结构的字节码指令集，那今天我们就来继续来学习，**什么是字节码指令**。

Java 的字节码指令由操作码和操作数组成：

- 操作码（Opcode）：一个字节长度（0-255，意味着指令集的操作码总数不可能超过 256 条），代表着某种特定的操作含义。
- 操作数（Operands）：零个或者多个，紧跟在操作码之后，代表此操作需要的参数。

由于 Java 虚拟机是基于栈而不是寄存器的结构，所以大多数字节码指令都只有一个操作码。比如 `aload_0` 就只有操作码没有操作数，而 `invokespecial #1` 则由操作码和操作数组成。

- aload_0：将局部变量表中下标为 0 的数据压入操作数栈中
- invokespecial #1：调用成员方法或者构造方法，并传递常量池中下标为 1 的常量

字节码指令主要有以下几种，分别是：

- 加载与存储指令
- 算术指令
- 类型转换指令
- 对象的创建与访问指令
- 方法调用和返回指令
- 操作数栈管理指令
- 控制转移指令

我们来一一说明下。

加载与存储指令

加载（load）和存储（store）指令是使用最频繁的指令，用于将数据从栈帧的局部变量表和操作数栈之间来回传递。

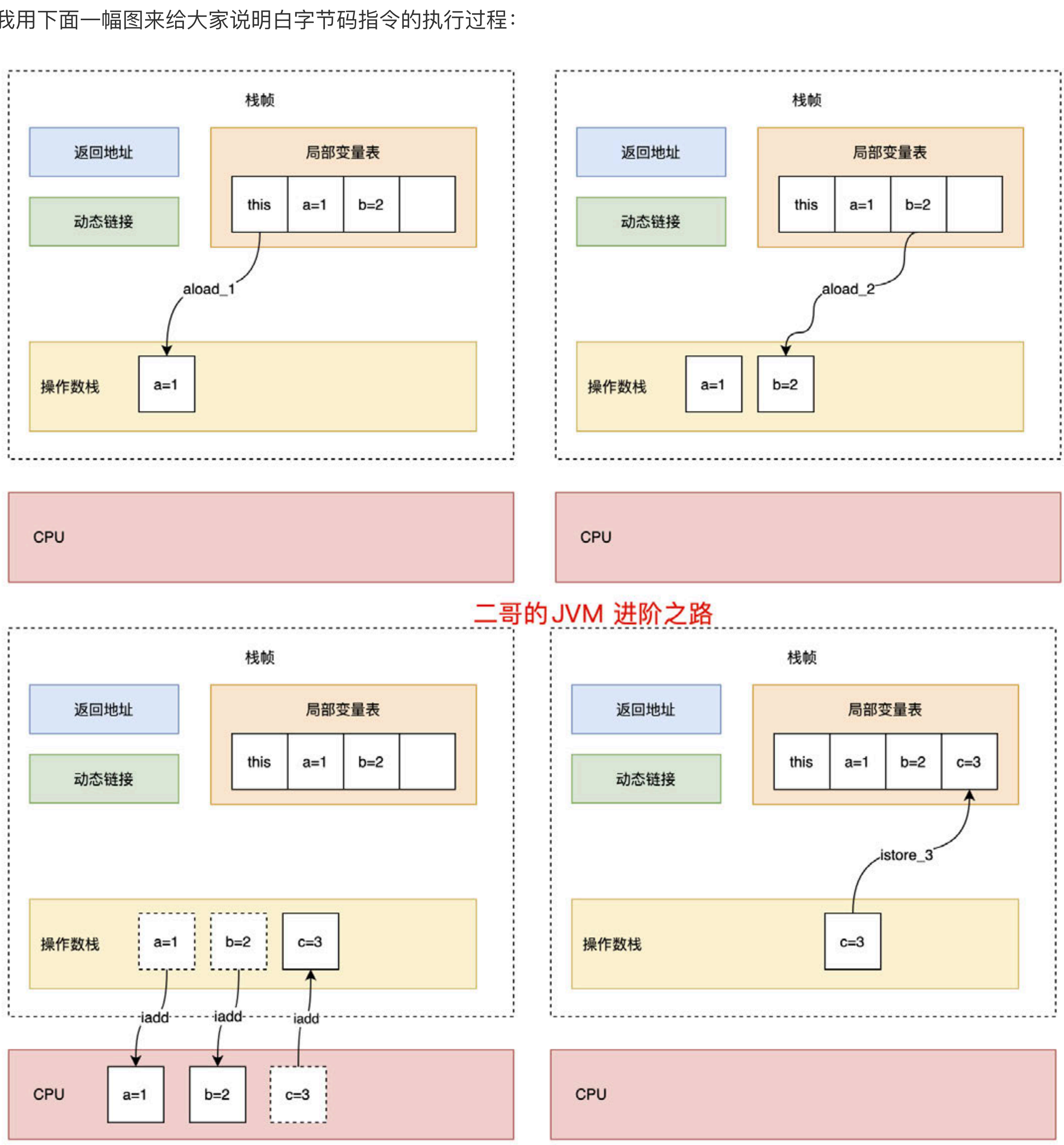
看下面这段代码。

```
public int add(int a, int b) {
    int result = a + b;
    return result;
}
```

使用 javap 查看字节码指令（大致）如下：

```
public int add(int, int);
    Code:
       0: iload_1
       1: iload_2
       2: iadd
       3: istore_3
       4: ireturn
```

我用下面一幅图来给大家说明白字节码指令的执行过程：



然后我们再来分析 load 和 store 指令的具体含义。

1）将局部变量表中的变量压入操作数栈中

- `xload_<n>`（x 为 i、l、f、d、a，n 默认为 0 到 3），表示将第 n 个局部变量压入操作数栈中。
- xload（x 为 i、l、f、d、a），通过指定参数的形式，将局部变量压入操作数栈中，当使用这个指令时，表示局部变量的数量可能超过了 4 个

解释一下。

x 为操作码助记符，表明是哪一种数据类型。见下表所示。

操作码助记符	数据类型
i	int
l	long
s	short
b	byte
c	char
f	float
d	double
a	引用数据类型

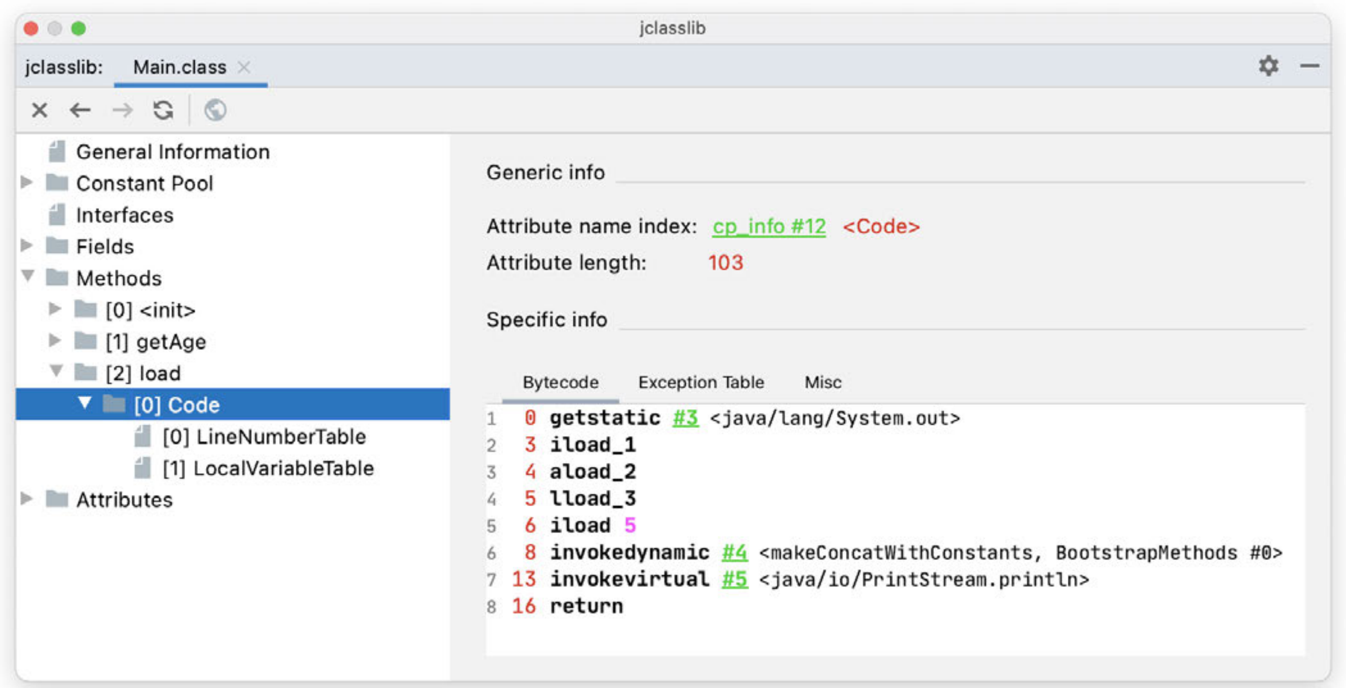
像 arraylength 指令，就没有操作码助记符，它没有代表数据类型的特殊字符，但操作数只能是一个数组类型的对象。

大部分的指令都不支持 byte、short 和 char，甚至没有任何指令支持 boolean 类型。编译器会将 byte 和 short 类型的数据带符号扩展（Sign-Extend）为 int 类型，将 boolean 和 char 零位扩展（Zero-Extend）为 int 类型。

举例来说。

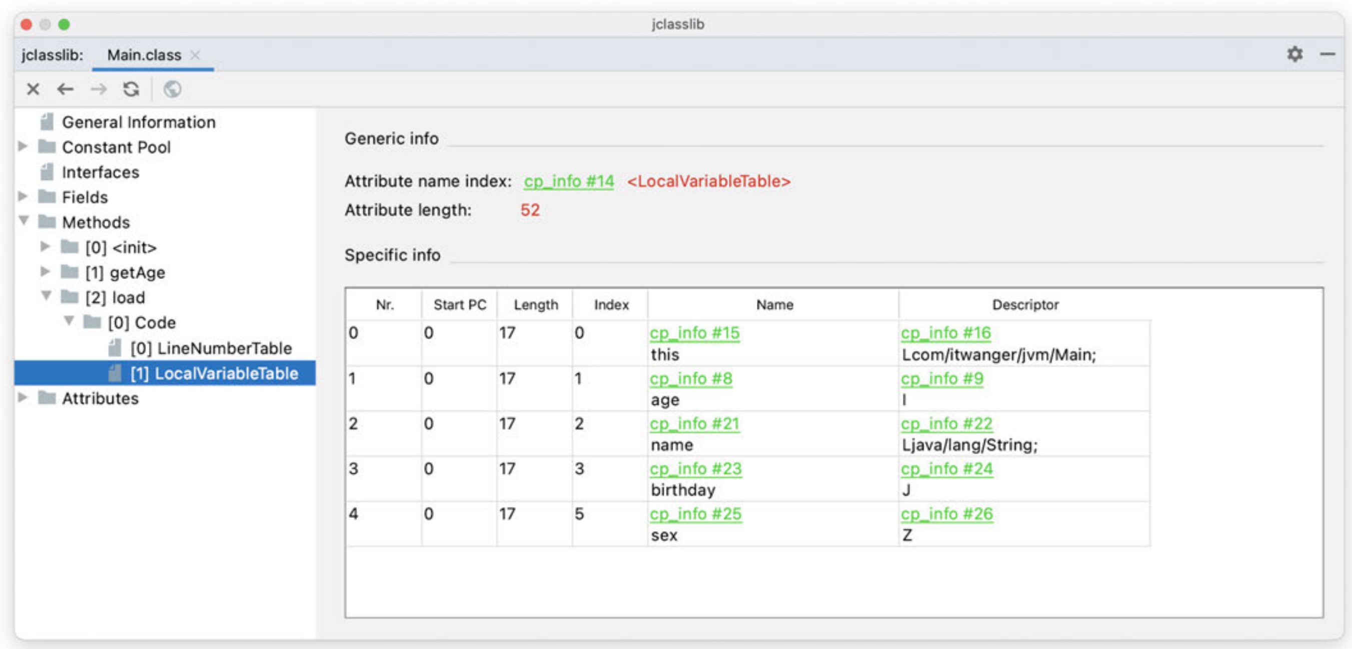
```
private void load(int age, String name, long birthday, boolean sex) {
    System.out.println(age + name + birthday + sex);
}
```

通过 jclasslib 看一下 `load()` 方法（4 个参数）的字节码指令。



- `iload_1`：将局部变量表中下标为 1 的 int 变量压入操作数栈中。
- `aload_2`：将局部变量表中下标为 2 的引用数据类型变量（此时为 String）压入操作数栈中。
- `lload_3`：将局部变量表中下标为 3 的 long 型变量压入操作数栈中。
- iload 5：将局部变量表中下标为 5 的 int 变量（实际为 boolean）压入操作数栈中。

通过查看局部变量表就能关联上了。



2）将常量池中的常量压入操作数栈中

根据数据类型和入栈内容的不同，又可以细分为 const 系列、push 系列和 ldc 指令。

const 系列，用于特殊的常量入栈，要入栈的常量隐含在指令本身。

指令	备注
iconst_<n>	n 从 -1 到 5
lconst_<n>	n 从 0 到 1
fconst_<n>	n 从 0 到 2
dconst_<n>	n 从 0 到 1
aconst_null	将 null 入栈

push 系列，主要包括 bipush 和 sipush，前者接收 8 位整数作为参数，后者接收 16 位整数。

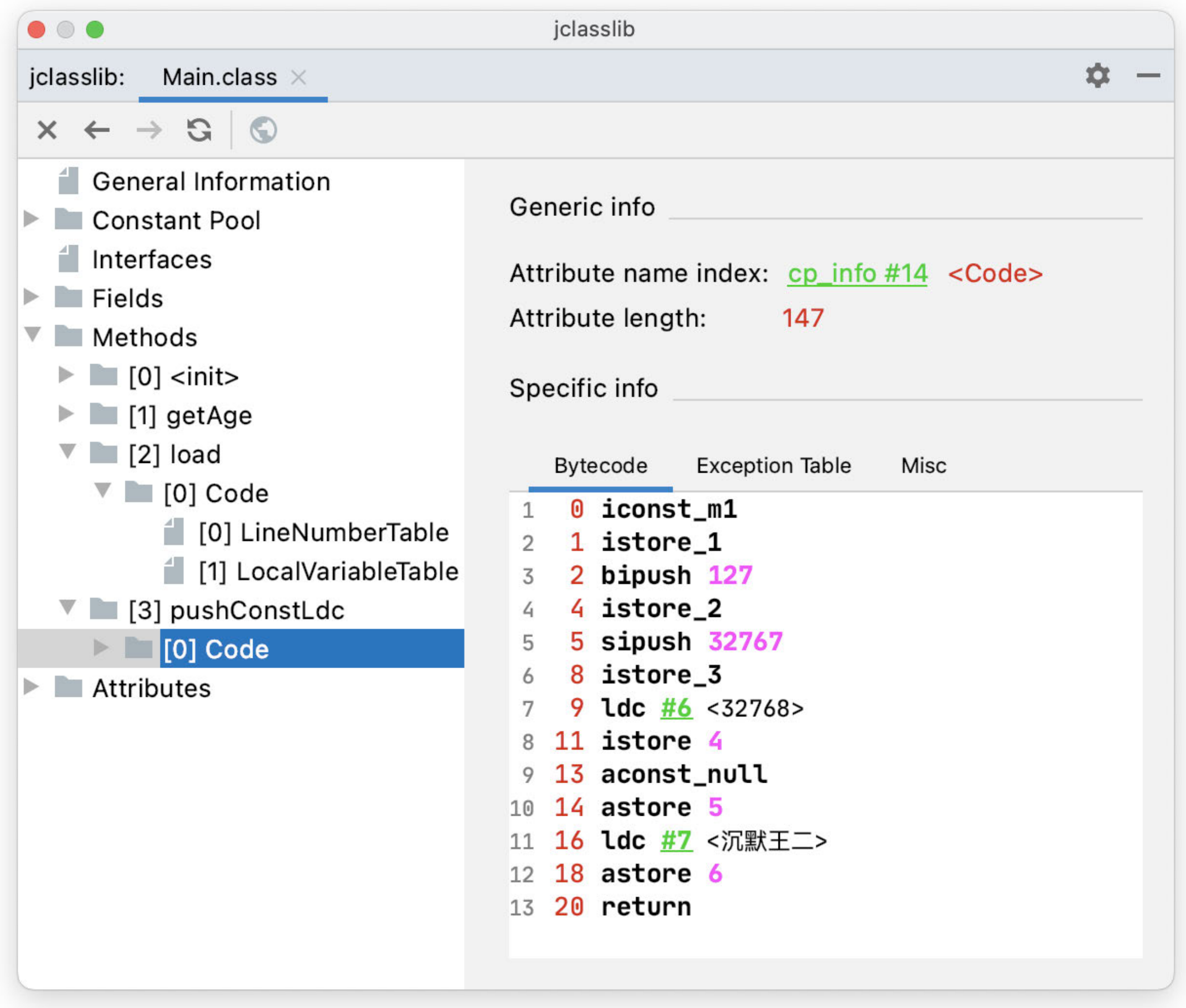
ldc 指令，当 const 和 push 不能满足的时候，万能的 ldc 指令就上场了，它接收一个 8 位的参数，指向常量池中的索引。

- `ldc_w`：接收两个 8 位数，索引范围更大。
- 如果参数是 long 或者 double，使用 `ldc2_w` 指令。

举例来说。

```
public void pushConstLdc() {
    // 范围 [-1,5]
    int iconst = -1;
    // 范围 [-128,127]
    int bipush = 127;
    // 范围 [-32768,32767]
    int sipush= 32767;
    // 其他 int
    int ldc = 32768;
    String aconst = null;
    String IdcString = "沉默王二";
}
```

通过 jclasslib 看一下 `pushConstLdc()` 方法的字节码指令。



- `iconst_m1`：将 -1 入栈。范围 [-1,5]。
- bipush 127：将 127 入栈。范围 [-128,127]。
- sipush 32767：将 32767 入栈。范围 [-32768,32767]。
- `ldc #6 <32768>`：将常量池中下标为 6 的常量 32768 入栈。
- aconst_null：将 null 入栈。
- `ldc #7 <沉默王二>`：将常量池中下标为 7 的常量“沉默王二”入栈。

3）将栈顶的数据出栈并装入局部变量表中

主要是用来给局部变量赋值，这类指令主要以 store 的形式存在。

- `xstore_<n>`（x 为 i、l、f、d、a，n 默认为 0 到 3）
- xstore（x 为 i、l、f、d、a）

明白了 `xload_<n>` 和 xload，再看 `xstore_<n>` 和 xstore 就会轻松得多，作用反了一下而已。

大家来想一个问题，为什么要有 `xstore_<n>` 和 `xload_<n>` 呢？它们的作用和 xstore n、xload n 不是一样的吗？

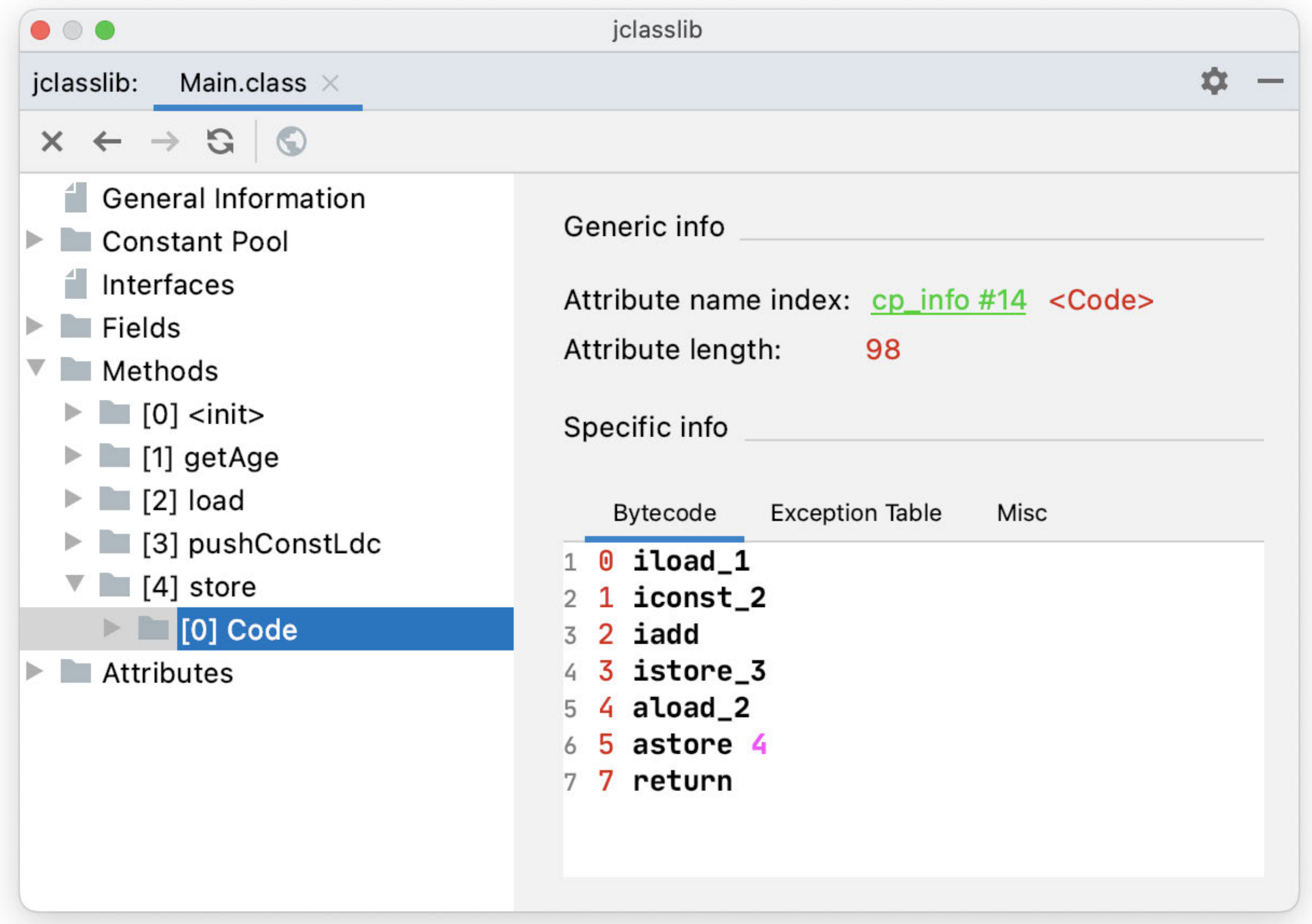
`xstore_<n>` 和 xstore n 的区别在于，前者相当于只有操作码，占用 1 个字节；后者相当于由操作码和操作数组成，操作码占 1 个字节，操作数占 2 个字节，一共占 3 个字节。

由于局部变量表中前几个位置总是非常常用，虽然 `xstore_<n>` 和 `xload_<n>` 增加了指令数量，但字节码的体积变小了！

举例来说。

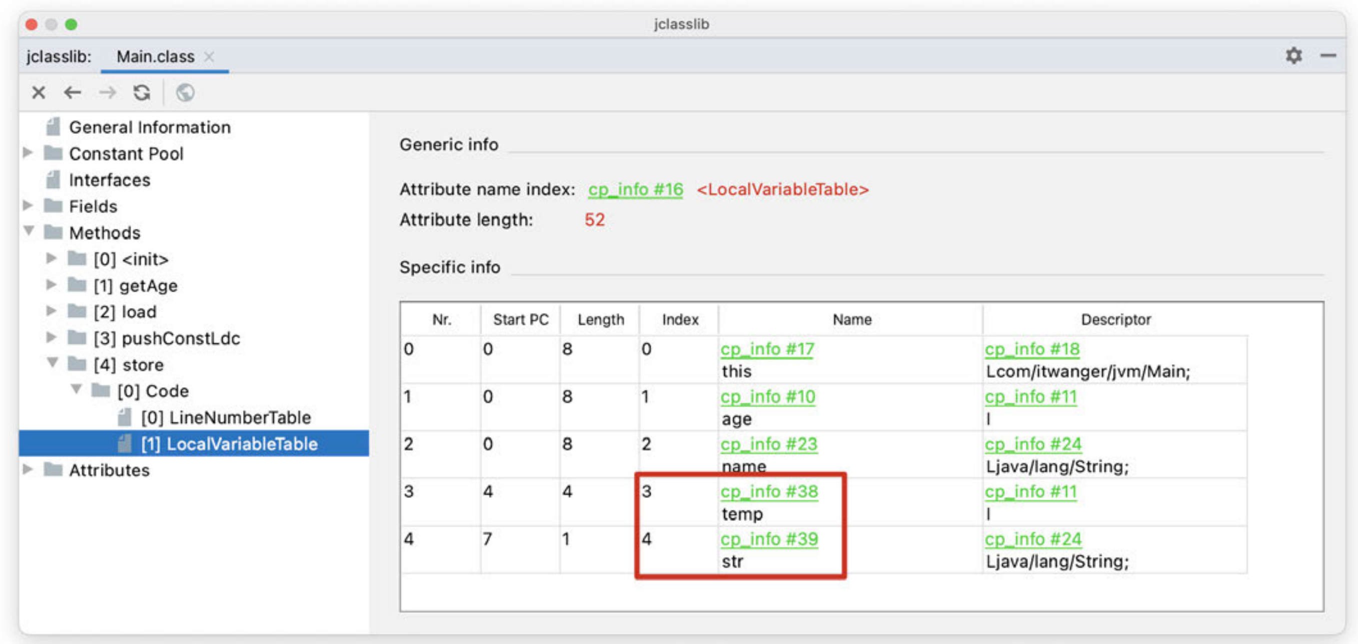
```
public void store(int age, String name) {
    int temp = age + 2;
    String str = name;
}
```

通过 jclasslib 看一下 `store()` 方法的字节码指令。



- `istore_3`：从操作数中弹出一个整数，并把它赋值给局部变量表中索引为 3 的变量。
- astore 4：从操作数中弹出一个引用数据类型，并把它赋值给局部变量表中索引为 4 的变量。

通过查看局部变量表就能关联上了。



算术指令

算术指令用于对两个操作数栈上的值进行某种特定运算，并把结果重新压入操作数栈。可以分为两类：整型数据的运算指令和浮点数据的运算指令。

这一节可以回顾一下 Java 运算符 ，就可以把一些非常简单的算术运算和 JVM 关联起来了。

需要注意的是，**数据运算可能会导致溢出**，比如两个很大的正整数相加，很可能会得到一个负数。但 Java 虚拟机规范中并没有对这种情况给出具体结果，因此程序是不会显式报错的。所以，**大家在开发过程中，如果涉及到较大的数据进行加法、乘法运算的时候，一定要注意**！

当发生溢出时，将会使用有符号的无穷大 Infinity 来表示；如果某个操作结果没有明确的数学定义的话，将会使用 NaN 值来表示。而且所有使用 NaN 作为操作数的算术操作，结果都会返回 NaN。

举例来说。

```
public void infinityNaN() {
    int i = 10;
    double j = i / 0.0;
    System.out.println(j); // Infinity

    double d1 = 0.0;
    double d2 = d1 / 0.0;
    System.out.println(d2); // NaN
}
```

- 任何一个非零的数除以浮点数 0（注意不是 int 类型），可以想象结果是无穷大 Infinity 的。
- 把这个非零的数换成 0 的时候，结果又不太好定义，就用 NaN 值来表示。

Java 虚拟机提供了两种**运算模式**：

- 向最接近数舍入：在进行浮点数运算时，所有的结果都必须舍入到一个适当的精度，不是特别精确的结果必须舍入为可被表示的最接近的精确值，如果有两种可表示的形式与该值接近，将优先选择最低有效位为零的（类

似四舍五入）。
- 向零舍入：将浮点数转换为整数时，采用该模式，该模式将在目标数值类型中选择一个最接近但是不大于原值的数字作为最精确的舍入结果（类似取整）。

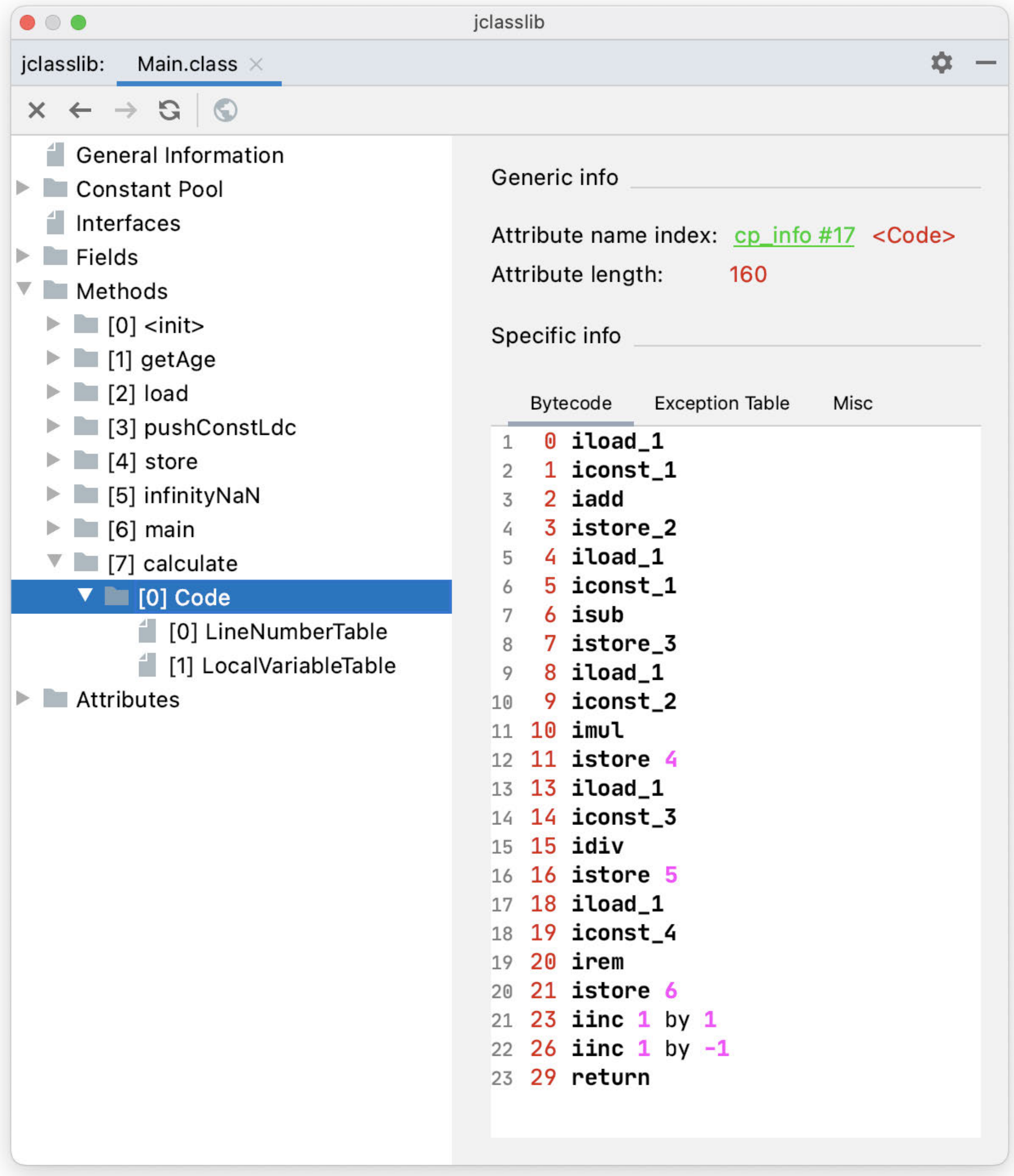
我把所有的算术指令列一下：

- 加法指令：iadd、ladd、fadd、dadd
- 减法指令：isub、lsub、fsub、dsub
- 乘法指令：imul、lmul、fmul、dmul
- 除法指令：idiv、ldiv、fdiv、ddiv
- 求余指令：irem、lrem、frem、drem
- 自增指令：iinc

举例来说。

```
public void calculate(int age) {
    int add = age + 1;
    int sub = age - 1;
    int mul = age * 2;
    int div = age / 3;
    int rem = age % 4;
    age++;
    age--;
}
```

通过 jclasslib 看一下 `calculate()` 方法的字节码指令。



- iadd，加法
- isub，减法
- imul，乘法
- idiv，除法
- irem，取余
- iinc，自增的时候 +1，自减的时候 -1

类型转换指令

类型转换指令可以分为两种：

1）宽化，小类型向大类型转换，比如 `int–>long–>float–>double`，对应的指令有：i2l、i2f、i2d、l2f、l2d、f2d。

- 从 int 到 long，或者从 int 到 double，是不会有精度丢失的；
- 从 int、long 到 float，或者 long 到 double 时，可能会发生精度丢失；
- 从 byte、char 和 short 到 int 的宽化类型转换实际上是隐式发生的，这样可以减少字节码指令，毕竟字节码指令只有 256 个，占一个字节。

2）窄化，大类型向小类型转换，比如从 int 类型到 byte、short 或者 char，对应的指令有：i2b、i2s、i2c；从 long 到 int，对应的指令有：l2i；从 float 到 int 或者 long，对应的指令有：f2i、f2l；从 double 到 int、long 或者 float，对应的指令有：d2i、d2l、d2f。

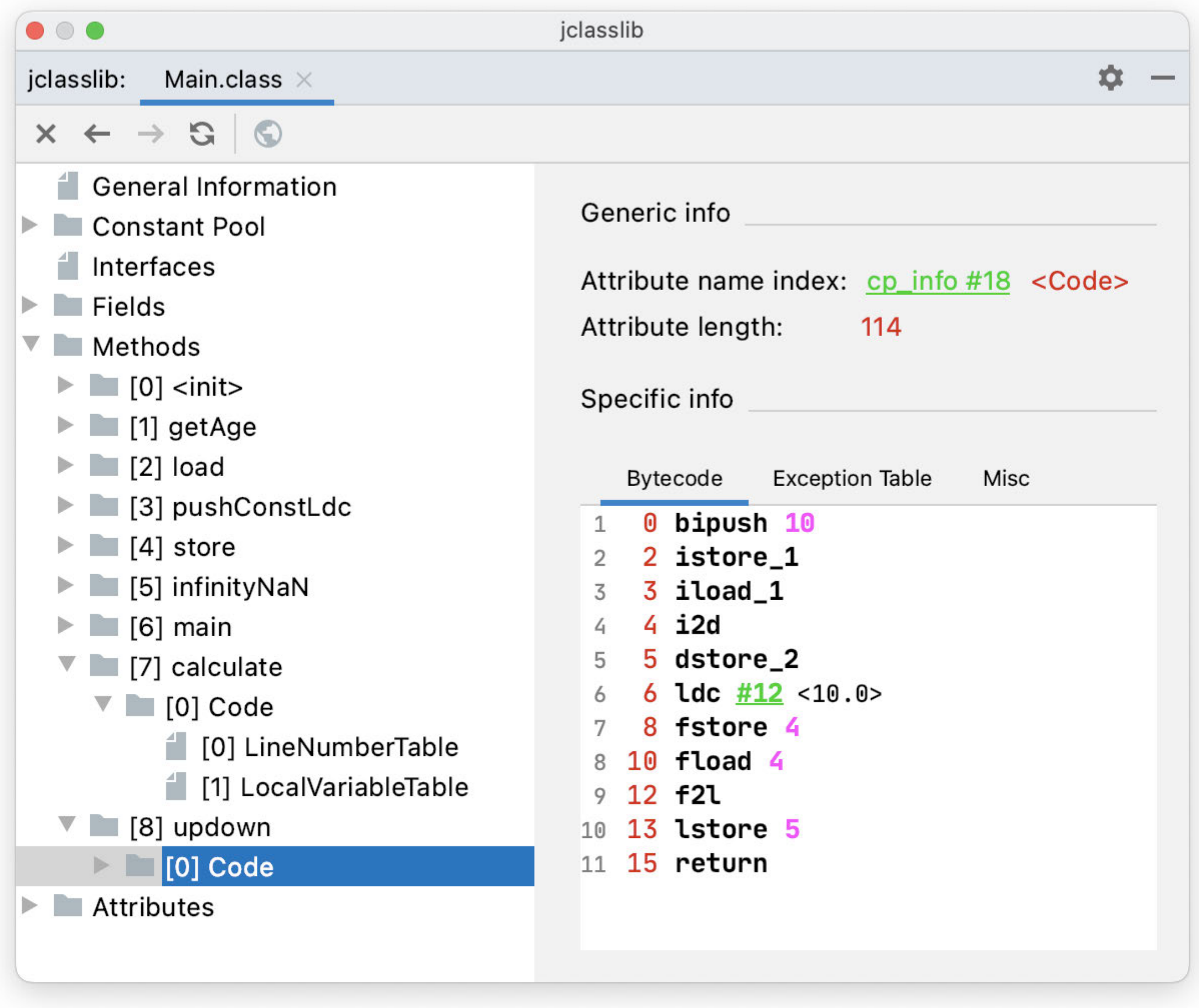
- 窄化很可能会发生精度丢失，毕竟是不同的数量级；
- 但 Java 虚拟机并不会因此抛出运行时异常。

> 可以回想一下前面讲过的：自动类型转换与强制类型转换

举例来说。

```
public void updown() {
    int i = 10;
    double d = i;

    float f = 10f;
    long ong = (long)f;
}
```

通过 jclasslib 看一下 `updown()` 方法的字节码指令。



- i2d，int 宽化为 double
- f2l， float 窄化为 long

对象的创建和访问指令

Java 是一门面向对象的编程语言，那么 Java 虚拟机是如何从字节码层面进行支持的呢?

1）创建指令

数组是一种特殊的对象，它创建的字节码指令和普通对象的创建指令不同。创建数组的指令有三种：

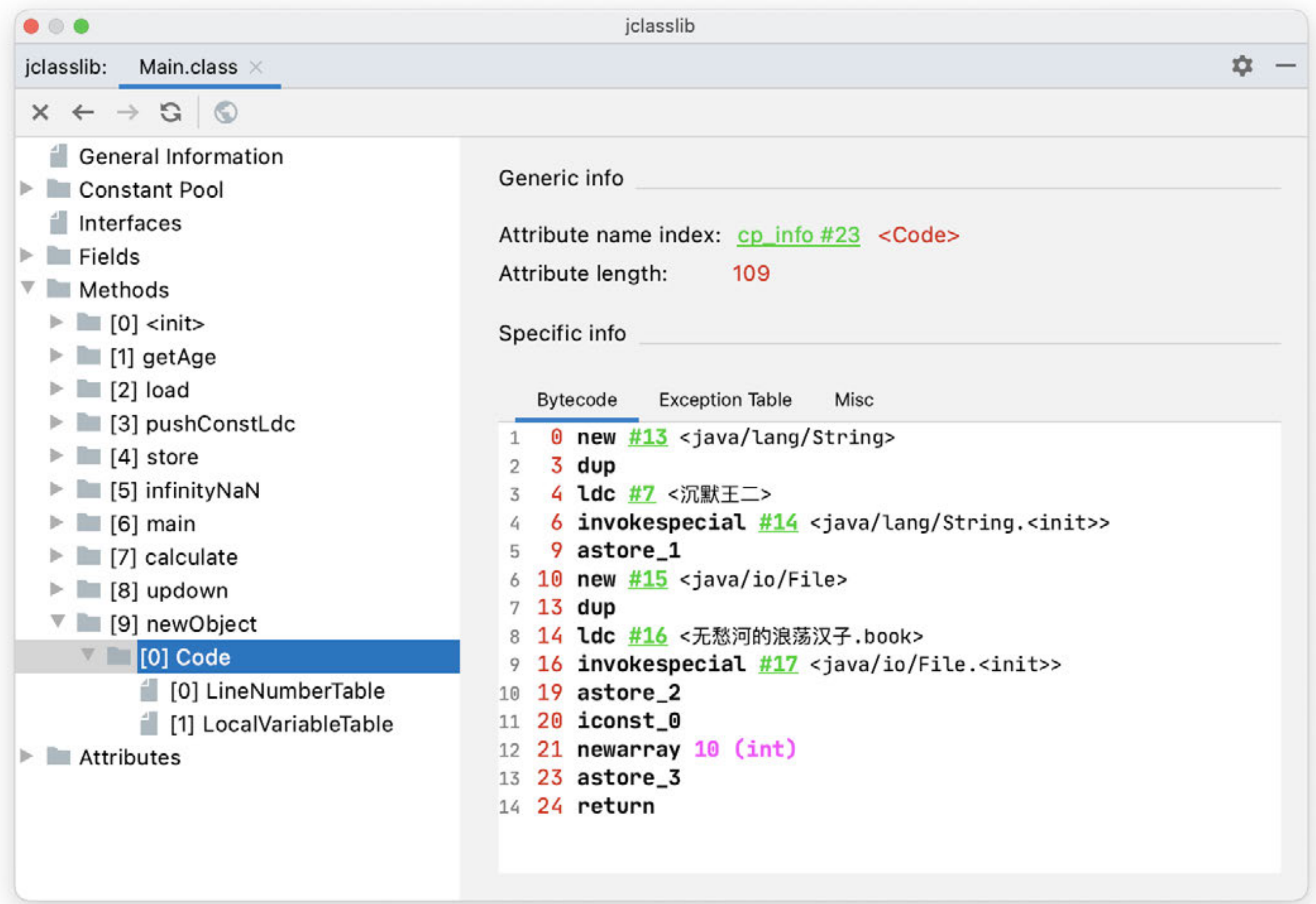
- newarray：创建基本数据类型的数组
- anewarray：创建引用类型的数组
- multianewarray：创建多维数组

而对象的创建指令只有一个，就是 `new`，它会接收一个操作数，指向常量池中的一个索引，表示要创建的类型。

举例来说。

```java
public void newObject() {
    String name = new String("沉默王二");
    File file = new File("无愁河的浪荡汉子.book");
    int [] ages = {};
}
```

通过 jclasslib 看一下 `newObject()` 方法的字节码指令。



- `new #13 <java/lang/String>`，创建一个 String 对象。
- `new #15 <java/io/File>`，创建一个 File 对象。
- `newarray 10 (int)`，创建一个 int 类型的数组。

2）字段访问指令

字段可以分为两类，一类是成员变量，一类是静态变量（也就是类变量），所以字段访问指令可以分为两类：

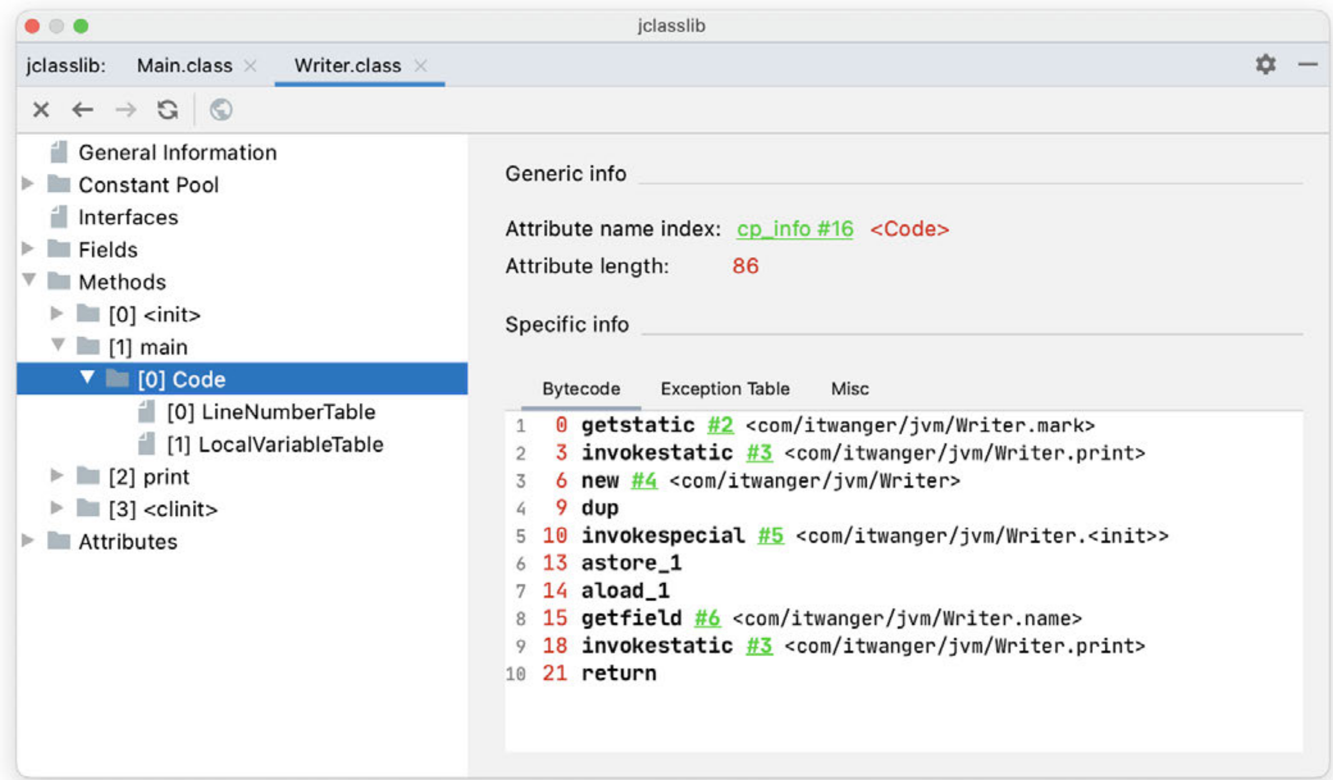
- 访问静态变量：getstatic、putstatic。
- 访问成员变量：getfield、putfield，需要创建对象后才能访问。

举例来说。

```java
public class Writer {
    private String name;
    static String mark = "作者";
```

```
    public static void main(String[] args) {
        print(mark);
        Writer w = new Writer();
        print(w.name);
    }

    public static void print(String arg) {
        System.out.println(arg);
    }
}
```

通过 jclasslib 看一下 `main()` 方法的字节码指令。



- `getstatic #2 <com/itwanger/jvm/Writer.mark>`，访问静态变量 mark
- `getfield #6 <com/itwanger/jvm/Writer.name>`，访问成员变量 name

方法调用和返回指令

方法调用指令有 5 个，分别用于不同的场景：

- invokevirtual：用于调用对象的成员方法，根据对象的实际类型进行分派，支持多态。
- invokeinterface：用于调用接口方法，会在运行时搜索由特定对象实现的接口方法进行调用。
- invokespecial：用于调用一些需要特殊处理的方法，包括构造方法、私有方法和父类方法。
- invokestatic：用于调用静态方法。
- invokedynamic：用于在运行时动态解析出调用点限定符所引用的方法，并执行。

举例来说。

```
public class InvokeExamples {
    private void run() {
        List ls = new ArrayList();
        ls.add("难顶");

        ArrayList als = new ArrayList();
        als.add("学不动了");
    }

    public static void print() {
        System.out.println("invokestatic");
    }

    public static void main(String[] args) {
        print();
        InvokeExamples invoke = new InvokeExamples();
        invoke.run();
    }
}
```

我们用 `javap -c InvokeExamples.class` 来反编译一下。

```
Compiled from "InvokeExamples.java"
public class com.itwanger.jvm.InvokeExamples {
  public com.itwanger.jvm.InvokeExamples();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  private void run();
    Code:
       0: new           #2                  // class java/util/ArrayList
       3: dup
       4: invokespecial #3                  // Method java/util/ArrayList."<init>":()V
       7: astore_1
       8: aload_1
       9: ldc           #4                  // String 难顶
      11: invokeinterface #5,  2            // InterfaceMethod java/util/List.add:
(Ljava/lang/Object;)Z
      16: pop
      17: new           #2                  // class java/util/ArrayList
      20: dup
      21: invokespecial #3                  // Method java/util/ArrayList."<init>":()V
      24: astore_2
      25: aload_2
      26: ldc           #6                  // String 学不动了
```

```
      28: invokevirtual #7                  // Method java/util/ArrayList.add:
(Ljava/lang/Object;)Z
      31: pop
      32: return

  public static void print();
    Code:
       0: getstatic     #8                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #9                  // String invokestatic
       5: invokevirtual #10                 // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
       8: return

  public static void main(java.lang.String[]);
    Code:
       0: invokestatic  #11                 // Method print:()V
       3: new           #12                 // class com/itwanger/jvm/InvokeExamples
       6: dup
       7: invokespecial #13                 // Method "<init>":()V
      10: astore_1
      11: aload_1
      12: invokevirtual #14                 // Method run:()V
      15: return
}
```

InvokeExamples 类有 4 个方法，包括缺省的构造方法在内。

1）invokespecial

缺省的构造方法内部会调用超类 Object 的初始化构造方法：

```
`invokespecial #1 // Method java/lang/Object."<init>":()V`
```

2）invokeinterface和invokevirtual

```
invokeinterface #5,  2  // InterfaceMethod java/util/List.add:(Ljava/lang/Object;)Z
```

由于 ls 变量的引用类型为接口 List，所以 `ls.add()` 调用的是 `invokeinterface` 指令，等运行时再确定是不是接口 List 的实现对象 ArrayList 的 `add()` 方法。

```
invokevirtual #7 // Method java/util/ArrayList.add:(Ljava/lang/Object;)Z
```

由于 als 变量的引用类型已经确定为 ArrayList，所以 `als.add()` 方法调用的是 `invokevirtual` 指令。

3）invokestatic

```
invokestatic  #11 // Method print:()V
```

`print()` 方法是静态的，所以调用的是 `invokestatic` 指令。

invokedynamic

`invokedynamic` 指令是 Java 7 引入的，主要是为了支持动态语言，比如 Groovy、Scala、JRuby 等。这些语言都是在运行时动态解析出调用点限定符所引用的方法，并执行。

看下面这段代码，用到了 [Lambda 表达式](#)，Lambda 表达式的实现就依赖于 invokedynamic 指令：

```java
import java.util.function.Function;

public class LambdaExample {
    public static void main(String[] args) {
        // 使用 Lambda 表达式定义一个函数
        Function<Integer, Integer> square = x -> x * x;

        // 调用这个函数
        int result = square.apply(5);

        System.out.println(result); // 输出 25
    }
}
```

在这个例子中，Lambda 表达式 `x -> x * x` 定义了一个接受一个整数并返回其平方的函数。在编译这段代码时，编译器会使用 invokedynamic 指令来动态地绑定这个 Lambda 表达式。

```
→  jvm git:(main) ✗ javap -c -p LambdaExample
警告：二进制文件LambdaExample包含com.github.paicoding.forum.test.javabetter.jvm.LambdaExample
Compiled from "LambdaExample.java"
public class com.github.paicoding.forum.test.javabetter.jvm.LambdaExample {
  public com.github.paicoding.forum.test.javabetter.jvm.LambdaExample();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: invokedynamic #2,  0              // InvokeDynamic #0:apply:()Ljava/util/function/Function;
       5: astore_1
       6: aload_1
       7: iconst_5
       8: invokestatic  #3                  // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
      11: invokeinterface #4,  2            // InterfaceMethod java/util/function/Function.apply:(Ljava/lang/Object;
va/lang/Object;
      16: checkcast     #5                  // class java/lang/Integer
      19: invokevirtual #6                  // Method java/lang/Integer.intValue:()I
      22: istore_2
      23: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
      26: iload_2
      27: invokevirtual #8                  // Method java/io/PrintStream.println:(I)V
      30: return

  private static java.lang.Integer lambda$main$0(java.lang.Integer);
    Code:
       0: aload_0
       1: invokevirtual #6                  // Method java/lang/Integer.intValue:()I
       4: aload_0
       5: invokevirtual #6                  // Method java/lang/Integer.intValue:()I
       8: imul
       9: invokestatic  #3                  // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
      12: areturn
}
→  jvm git:(main) ✗
```

①、`invokedynamic #2,  0`：使用 `invokedynamic` 调用一个引导方法（Bootstrap Method），这个方法负责实现并返回一个 `Function` 接口的实例。这里的 Lambda 表达式 `x -> x * x` 被转换成了一个 `Function` 对象。引导方法在首次执行时会被调用，它负责生成一个 `CallSite`，该 `CallSite` 包含了指向具体实现 Lambda 表达式的方法句柄（Method Handle）。在这个例子中，这个方法句柄指向了 `lambda$main$0` 方法。

②、`astore_1`：将 `invokedynamic` 指令的结果（Lambda 表达式的 `Function` 对象）存储到局部变量表的位置 1。

③、Lambda 表达式的实现是：`lambda$main$0`，这是 Lambda 表达式 `x -> x * x` 的实际实现。它接收一个 `Integer` 对象作为参数，计算其平方，然后返回结果。

```
public class LambdaExample {
    public LambdaExample() {
    }

    public static void main(String[] args) {
        Function<Integer, Integer> square = (x) -> {
            return x * x;
        };
        int result = (Integer)square.apply(5);
        System.out.println(result);
    }
}
```

其他指令这里就不再分析下去了，大家可以尝试一下，检验自己的学习成果。

方法返回指令

方法返回指令根据方法的返回值类型进行区分，常见的返回指令见下图，就是各种 return。

指令	类型
return	void
ireturn	int（boolean、byte、char、short）
lreturn	long
freturn	float
dreturn	double
areturn	引用类型

操作数栈管理指令

常见的操作数栈管理指令有 pop、dup 和 swap。

- 将一个或两个元素从栈顶弹出，并且直接废弃，比如 pop，pop2；

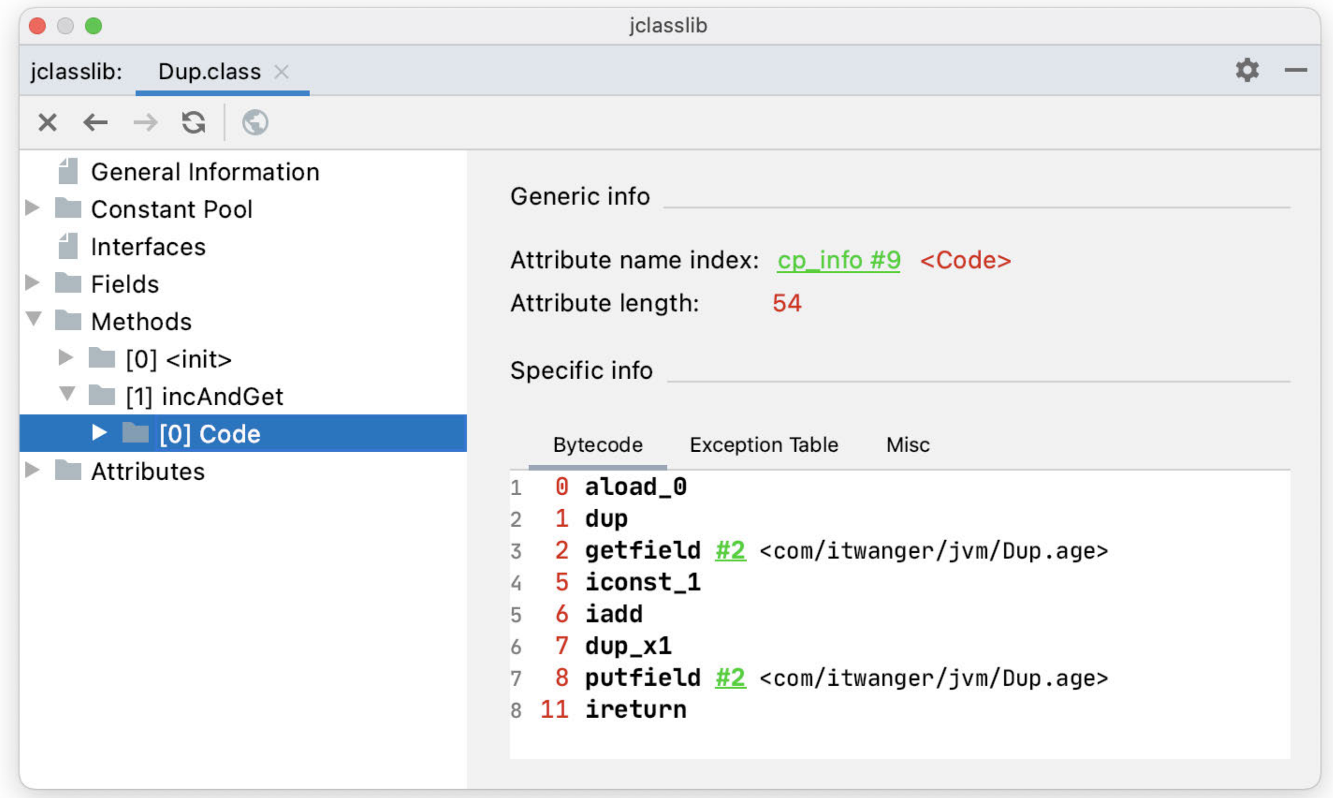
- 复制栈顶的一个或两个数值并将其重新压入栈顶，比如 dup，dup2，`dup*×1`，`dup2**×1`，`dup*×2`，`dup2**×2`；
- 将栈最顶端的两个槽中的数值交换位置，比如 swap。

这些指令不需要指明数据类型，因为是按照位置压入和弹出的。

举例来说。

```
public class Dup {
    int age;
    public int incAndGet() {
        return ++age;
    }
}
```

通过 jclasslib 看一下 `incAndGet()` 方法的字节码指令。



- `aload_0`：将 this 入栈。
- dup：复制栈顶的 this。
- `getfield #2`：将常量池中下标为 2 的常量加载到栈上，同时将一个 this 出栈。
- `iconst_1`：将常量 1 入栈。
- iadd：将栈顶的两个值相加后出栈，并将结果放回栈上。
- `dup_x1`：复制栈顶的元素，并将其插入 this 下面。
- `putfield #2`： 将栈顶的两个元素出栈，并将其赋值给字段 age。
- ireturn：将栈顶的元素出栈返回。

控制转移指令

控制转移指令包括：

- 比较指令，比较栈顶的两个元素的大小，并将比较结果入栈。
- 条件跳转指令，通常和比较指令一块使用，在条件跳转指令执行前，一般先用比较指令进行栈顶元素的比较，然后进行条件跳转。
- 比较条件转指令，类似于比较指令和条件跳转指令的结合体，它将比较和跳转两个步骤合二为一。
- 多条件分支跳转指令，专为 switch-case 语句设计的。
- 无条件跳转指令，目前主要是 goto 指令。

和之前学过《流程控制语句》就关联了起来。

1）比较指令

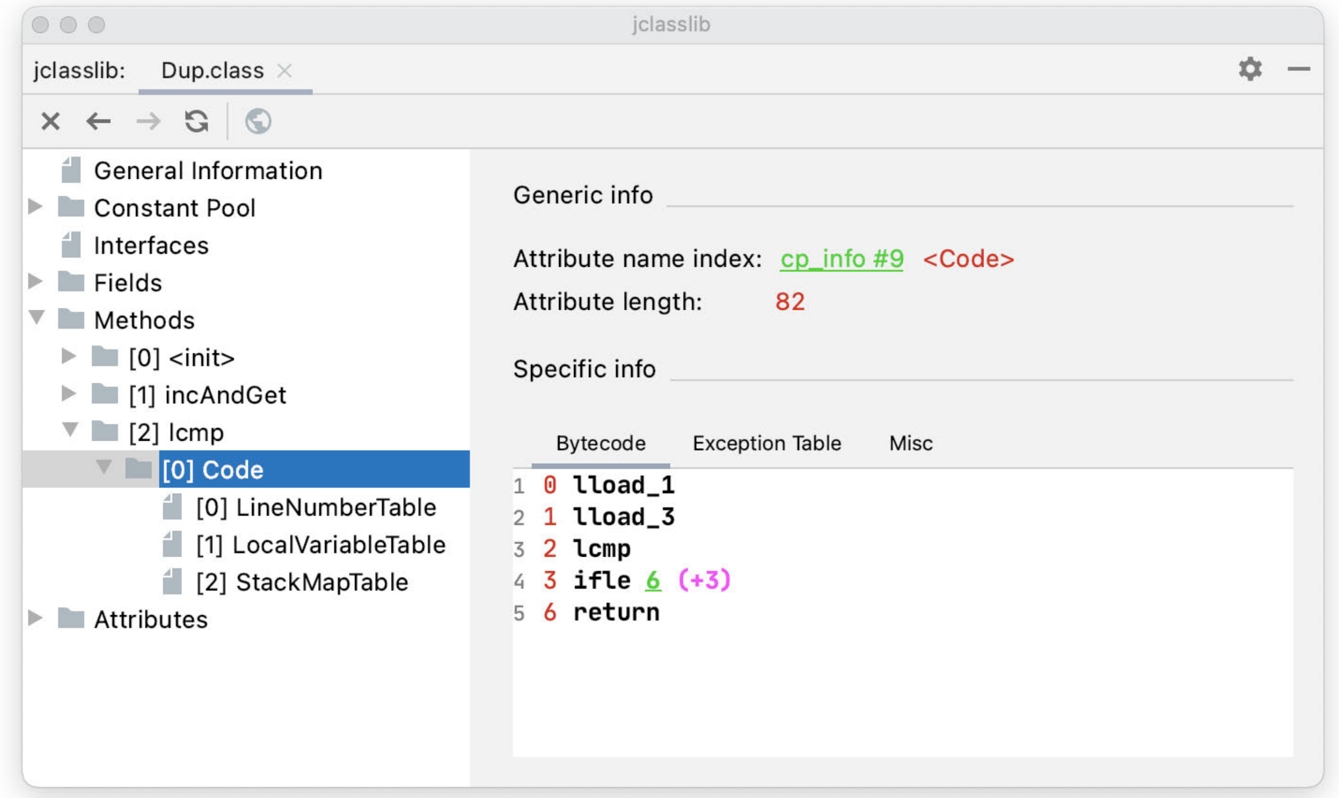
比较指令有：dcmpg，dcmpl、fcmpg、fcmpl、lcmp，指令的第一个字母代表的含义分别是 double、float、long。注意，没有 int 类型。

对于 double 和 float 来说，由于 NaN 的存在，有两个版本的比较指令。拿 float 来说，有 fcmpg 和 fcmpl，区别在于，如果遇到 NaN，fcmpg 会将 1 压入栈，fcmpl 会将 -1 压入栈。

举例来说。

```
public void lcmp(long a, long b) {
    if(a > b){}
}
```

通过 jclasslib 看一下 `lcmp()` 方法的字节码指令。



lcmp 用于两个 long 型的数据进行比较。

2）条件跳转指令

指令	说明
ifeq	等于 0 时跳转
ifne	不等于 0 时跳转
iflt	小于 0 时跳转
ifle	小于等于 0 时跳转
ifgt	大于 0 时跳转
ifge	大于等于 0 时跳转
ifnull	为 null 时跳转
ifnonnull	不为 null 时跳转

这些指令都会接收两个字节的操作数，它们的统一含义是，弹出栈顶元素，测试它是否满足某一条件，满足的话，跳转到对应位置。

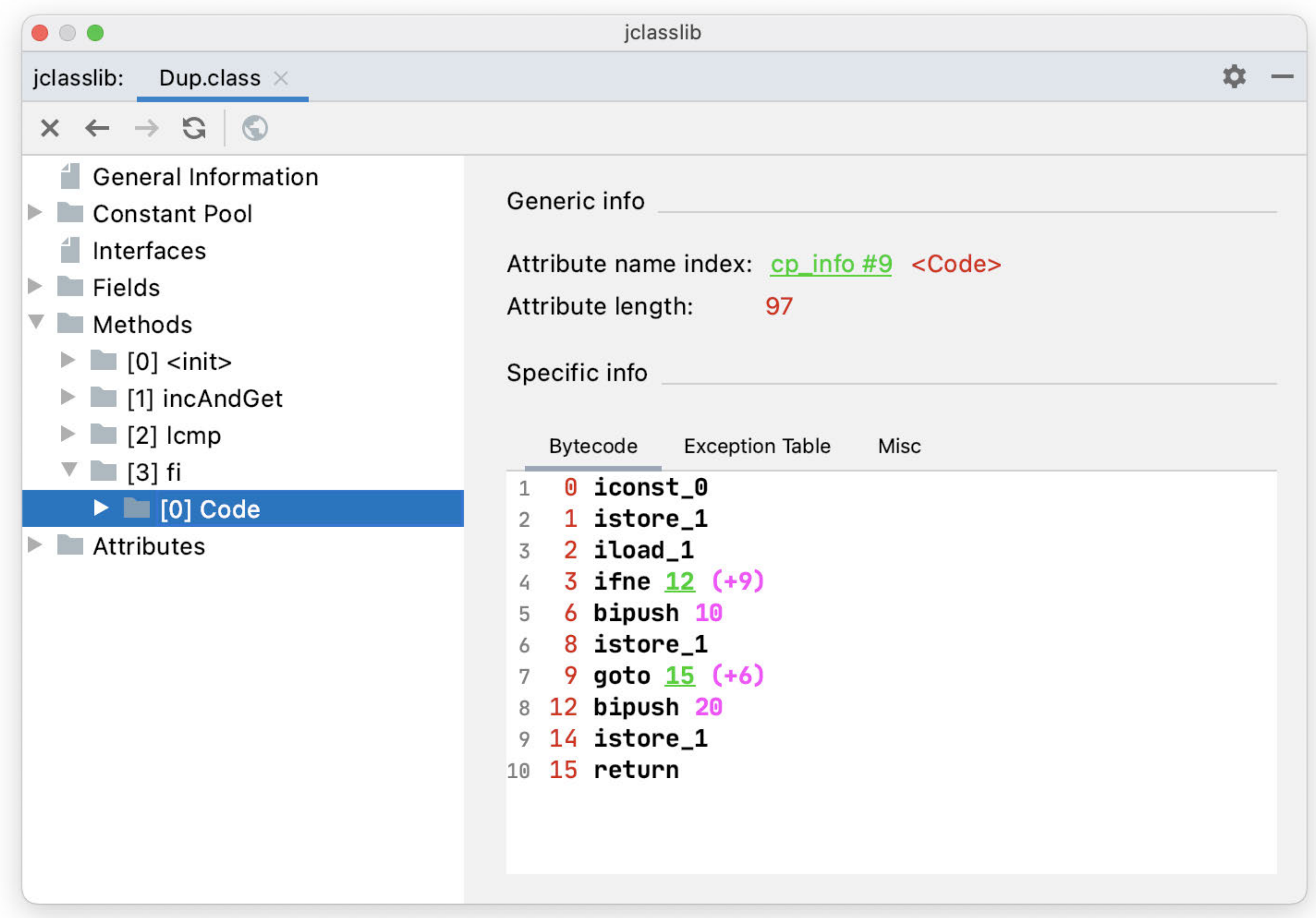
对于 long、float 和 double 类型的条件分支比较，会先执行比较指令返回一个整型值到操作数栈中后再执行 int 类型的条件跳转指令。

对于 boolean、byte、char、short，以及 int，则直接使用条件跳转指令来完成。

举例来说。

```
public void fi() {
    int a = 0;
    if (a == 0) {
        a = 10;
    } else {
        a = 20;
    }
}
```

通过 jclasslib 看一下 `fi()` 方法的字节码指令。



`3 ifne 12 (+9)` 的意思是，如果栈顶的元素不等于 0，跳转到第 12（3+9）行 `12 bipush 20`。

3）比较条件转指令

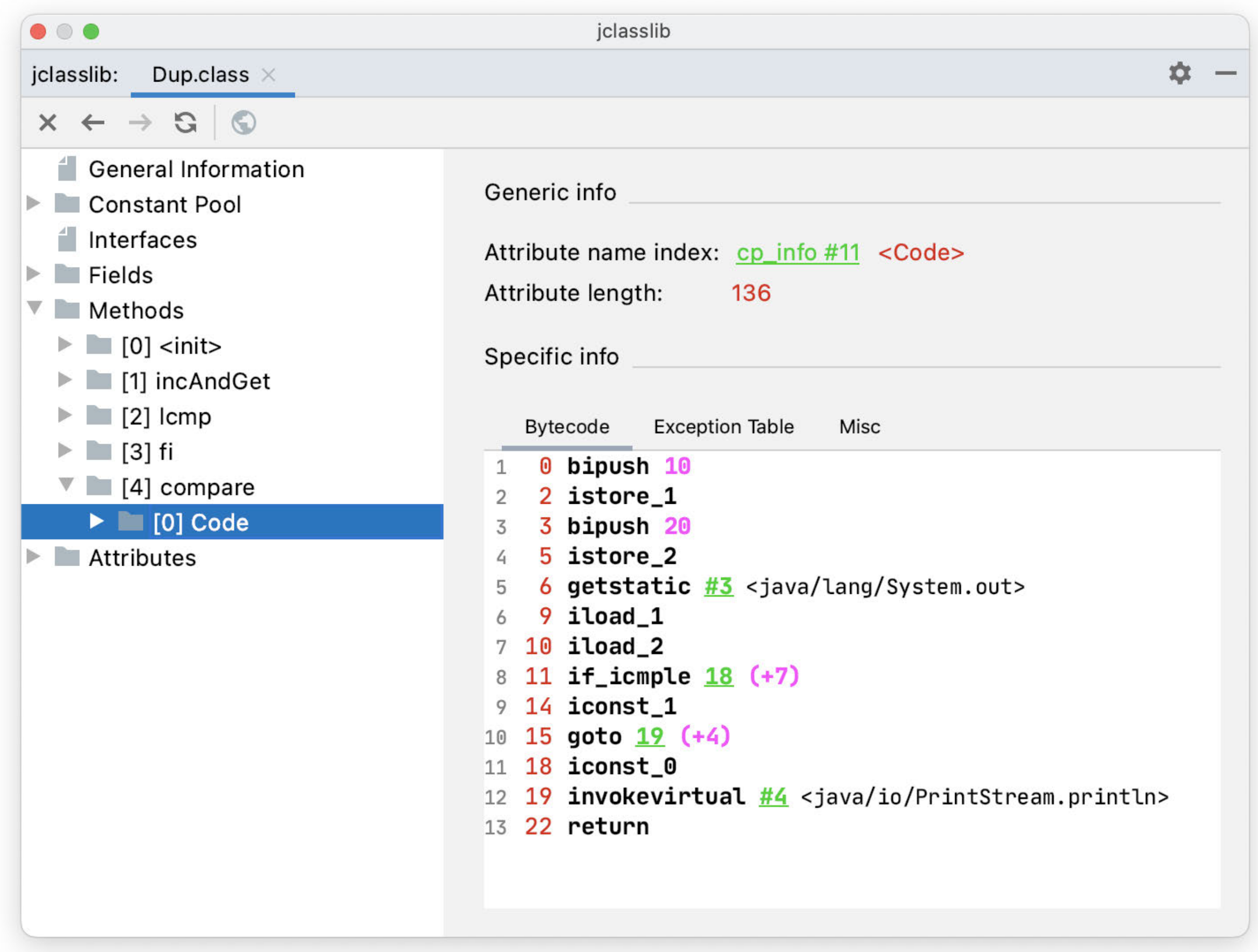
指令	说明
if_icmpeq	比较栈顶两个 int 类型的数值大小，相等时跳转
if_icmpne	比较栈顶两个 int 类型的数值大小，不等时跳转
if_icmplt	比较栈顶两个 int 类型的数值大小，前者小于后者时跳转
if_icmple	比较栈顶两个 int 类型的数值大小，前者小于等于后者时跳转
if_icmpgt	比较栈顶两个 int 类型的数值大小，前者大于后者时跳转
if_icmpge	比较栈顶两个 int 类型的数值大小，前者大于等于后者时跳转
if_acmpeq	比较栈顶两个引用类型的大小，相等时跳转
if_acmpne	比较栈顶两个引用类型的大小，相等时跳转

前缀“`if_`”后，以字符“i”开头的指令针对 int 型整数进行操作，以字符“a”开头的指令表示对象的比较。

举例来说。

```
public void compare() {
    int i = 10;
    int j = 20;
    System.out.println(i > j);
}
```

通过 jclasslib 看一下 `compare()` 方法的字节码指令。



`11 if_icmple 18 (+7)` 的意思是，如果栈顶的两个 int 类型的数值比较的话，如果前者小于后者时跳转到第 18 行（11+7）。

4）多条件分支跳转指令

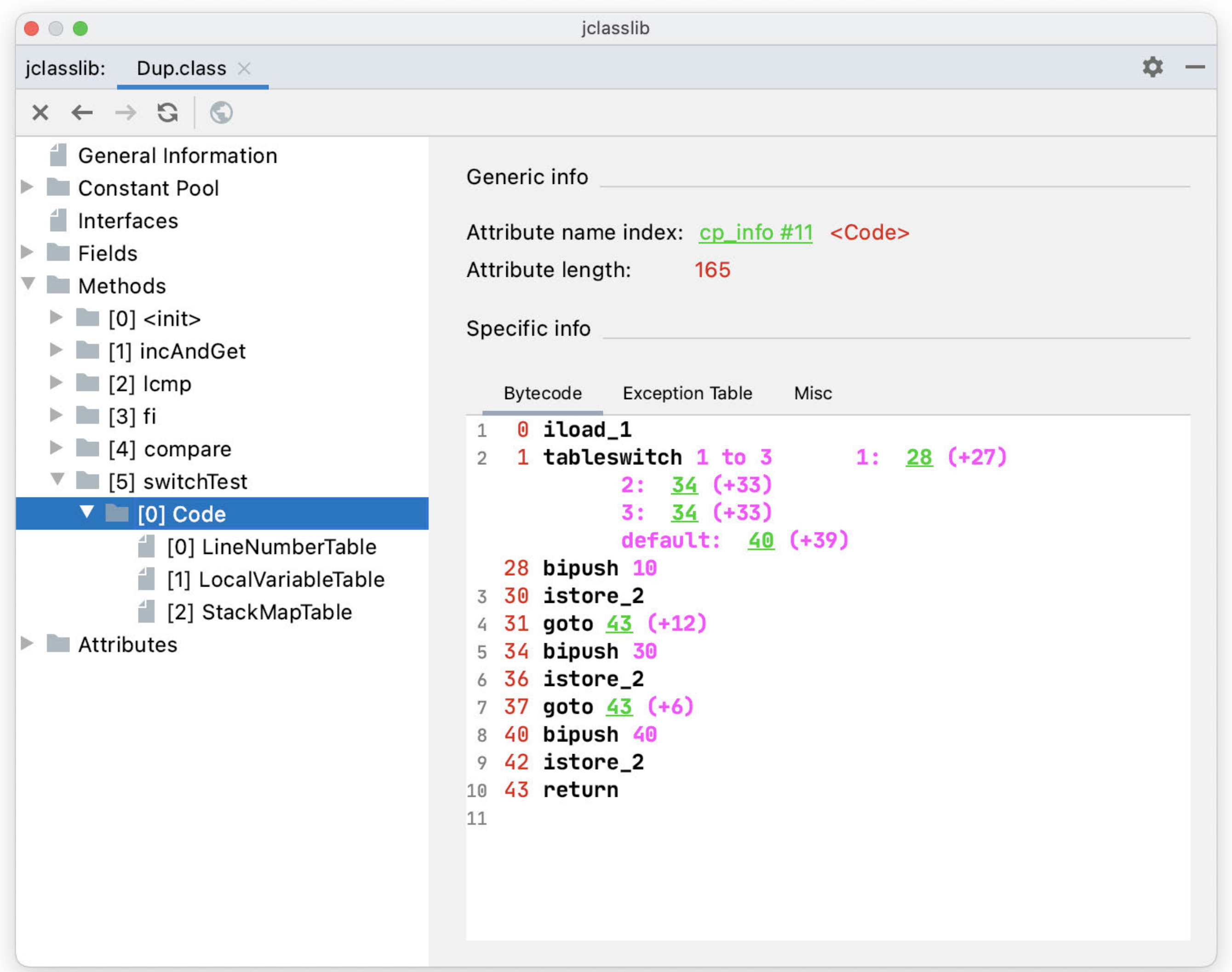
主要有 tableswitch 和 lookupswitch，前者要求多个条件分支值是连续的，它内部只存放起始值和终止值，以及若干个跳转偏移量，通过给定的操作数 index，可以立即定位到跳转偏移量位置，因此效率比较高；后者内部存放着各个离散的 case-offset 对，每次执行都要搜索全部的 case-offset 对，找到匹配的 case 值，并根据对应的 offset 计算跳转地址，因此效率较低。

举例来说。

```
public void switchTest(int select) {
    int num;
    switch (select) {
        case 1:
            num = 10;
            break;
        case 2:
        case 3:
            num = 30;
            break;
```

```
        default:
            num = 40;
    }
}
```

通过 jclasslib 看一下 `switchTest()` 方法的字节码指令。



case 2 的时候没有 break，所以 case 2 和 case 3 是连续的，用的是 tableswitch。如果等于 1，跳转到 28 行；如果等于 2 和 3，跳转到 34 行，如果是 default，跳转到 40 行。

5）无条件跳转指令

goto 指令接收两个字节的操作数，共同组成一个带符号的整数，用于指定指令的偏移量，指令执行的目的就是跳转到偏移量给定的位置处。

前面的例子里都出现了 goto 的身影，也很好理解。如果指令的偏移量特别大，超出了两个字节的范围，可以使用指令 goto_w，接收 4 个字节的操作数。

异常处理时的字节码指令

让我们通过一个简单的 Java 代码示例来说明异常处理时的字节码指令。

```
public class ExceptionExample {
    public void testException() {
        try {
            int a = 1 / 0; // 这将导致除以零的异常
        } catch (ArithmeticException e) {
            System.out.println("发生算术异常");
        }
    }
}
```

编译上述代码后，使用 `javap -c ExceptionExample` 可以查看其字节码，大致如下：

```
public void testException();
  Code:
    0: iconst_1
    1: iconst_0
    2: idiv
    3: istore_1
    4: goto 12
    7: astore_1
    8: getstatic     #2  // Field java/lang/System.out:Ljava/io/PrintStream;
    11: ldc           #3  // String 发生算术异常
    13: invokevirtual #4  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    16: return
  Exception table:
    from    to  target type
       0     4     7   Class java/lang/ArithmeticException
```

①、除法

`0: iconst_1`、`1: iconst_0`、`2: idiv` 这三个指令是执行除法运算 `1 / 0`。前两个指令将常数 1 和 0 分别推送到操作数栈，然后 `idiv` 指令执行除法操作。

`3: istore_1` 将除法的结果存储到局部变量表中（这里会发生异常，指令实际上不会执行）。

②、异常处理

`4: goto 12`：在 `try` 块的末尾，有一个 `goto` 指令跳过 `catch` 块的代码。

`7: astore_1` 这是 `catch` 块的开始。如果捕获到异常，将异常对象存储到局部变量表。

`8 - 13: getstatic, ldc, invokevirtual` 这些指令执行 `System.out.println("发生算术异常")`。

`16: return` 方法返回。

Exception table 这部分定义了异常处理器。在这个例子中，当在字节码偏移量 0 到 4 之间发生 `ArithmeticException` 时，控制跳转到偏移量 7，即 `catch` 块的开始。

详细大家经过这里例子可以和前面学过的《异常处理》关联起来。

synchronized 的字节码指令

好，我们再来看一个关于 synchronized 关键字的示例，就一个简单的同步代码块：

```
public class SynchronizedExample {
    public void syncBlockMethod() {
        synchronized(this) {
            // 同步块体
        }
    }
}
```

对应的字节码大致如下：

```
public void syncBlockMethod();
  Code:
    0: aload_0
    1: dup
    2: astore_1
    3: monitorenter
    4: aload_1
    5: monitorexit
    6: goto          14
    9: astore        2
    11: aload_1
    12: monitorexit
    13: aload         2
    15: athrow
    16: return
  Exception table:
    from    to  target type
        4     6     9   any
        9    13     9   any
```

`monitorenter / monitorexit` 这两个指令用于同步块的开始和结束。`monitorenter` 指令用于获取对象的监视器锁，`monitorexit` 指令用于释放锁。

希望大家通过这个简单的示例，把前面学过的《synchronized》关键字关联起来。

> 推荐阅读：https://segmentfault.com/a/1190000037628881

小结

路漫漫其修远兮，吾将上下而求索。

这节我们详细地介绍了 Java 字节码指令，包括算术指令、类型转换指令、对象的创建和访问指令、方法调用和返回指令、操作数栈管理指令、控制转移指令、异常处理时的字节码指令、synchronized 的字节码指令等。

想要走得更远，Java 字节码指令这块就必须得硬碰硬地吃透，希望二哥的这些分享可以帮助到大家~

第八节：深入理解栈帧结构

前面我们讲栈虚拟机和寄存器虚拟机的时候，提到过栈帧结构；在讲字节码指令的时候，又提到了栈帧中的操作数栈，那今天我们就来详细地讲一讲 JVM 的栈帧结构，好让大家对栈帧有一个更加清晰的认知。

我们从下面这幅图开始讲起。

Java 的源码文件经过编译器编译后会生成字节码文件，然后由 JVM 的类加载器进行加载，再交给执行引擎执行。在执行过程中，JVM 会划出一块内存空间来存储程序执行期间所需要用到的数据，这块空间一般被称为运行时数据区。

栈帧（Stack Frame）是运行时数据区中用于支持虚拟机进行方法调用和方法执行的数据结构。每一个方法从调用开始到执行完成，都对应着一个栈帧在虚拟机栈/本地方法栈里从入栈到出栈的过程。

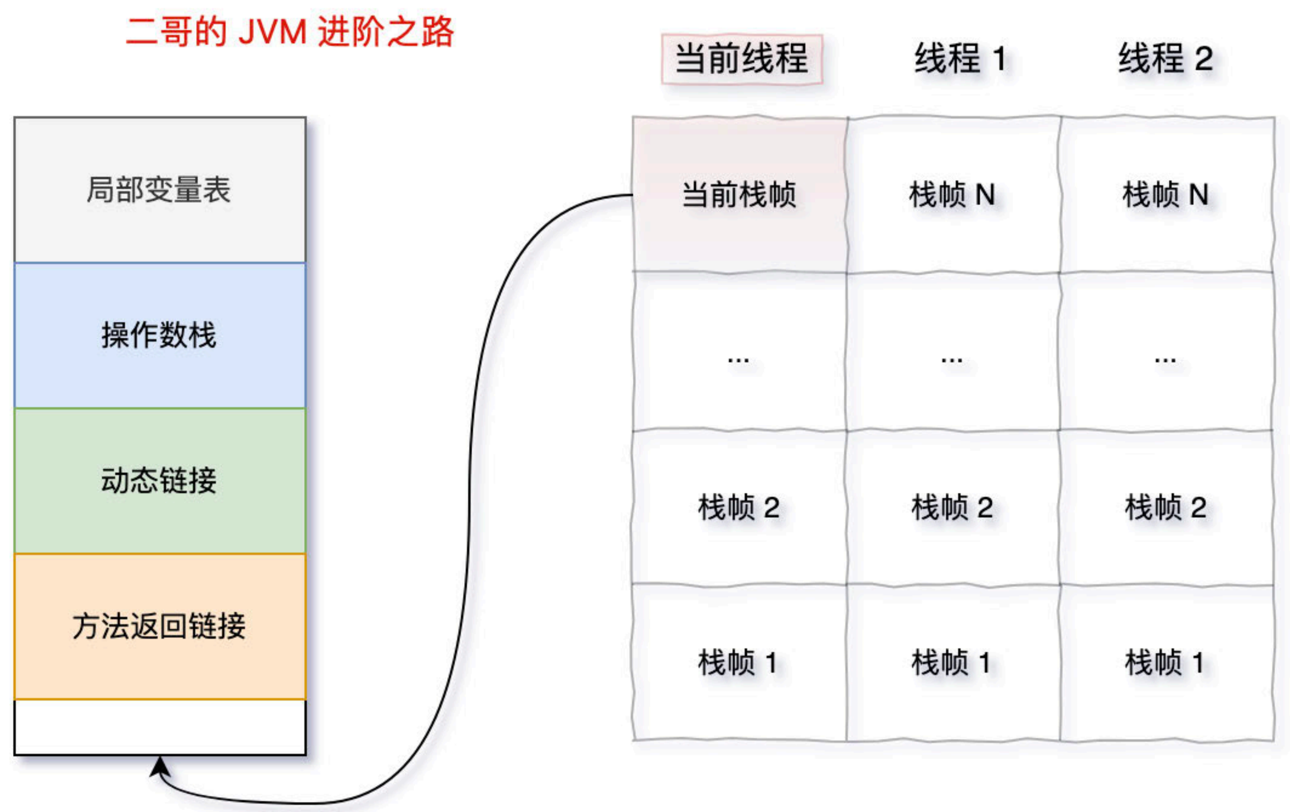
本地方法，也就是 native 方法，我们前面有详细地讲过，由 C/C++ 实现。

每一个栈帧都包括了局部变量表、操作数栈、动态链接、方法返回地址和一些额外的附加信息。

在编译程序代码时，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写入到方法表的 Code 属性之中。

> 方法表、局部变量表我们在讲字节码的时候有讲过，可以戳链接再回头看一下，这篇内容也会继续盘一盘。

一个线程中的方法调用链可能会很长，很多方法都处于执行状态。在当前线程中，位于栈顶的栈帧被称为当前栈帧（Current Stack Frame），与这个栈帧相关联的方法成为当前方法。执行引擎运行的所有字节码指令都是对当前栈帧进行操作，在概念模型上，栈帧的结构如下图所示：



局部变量表

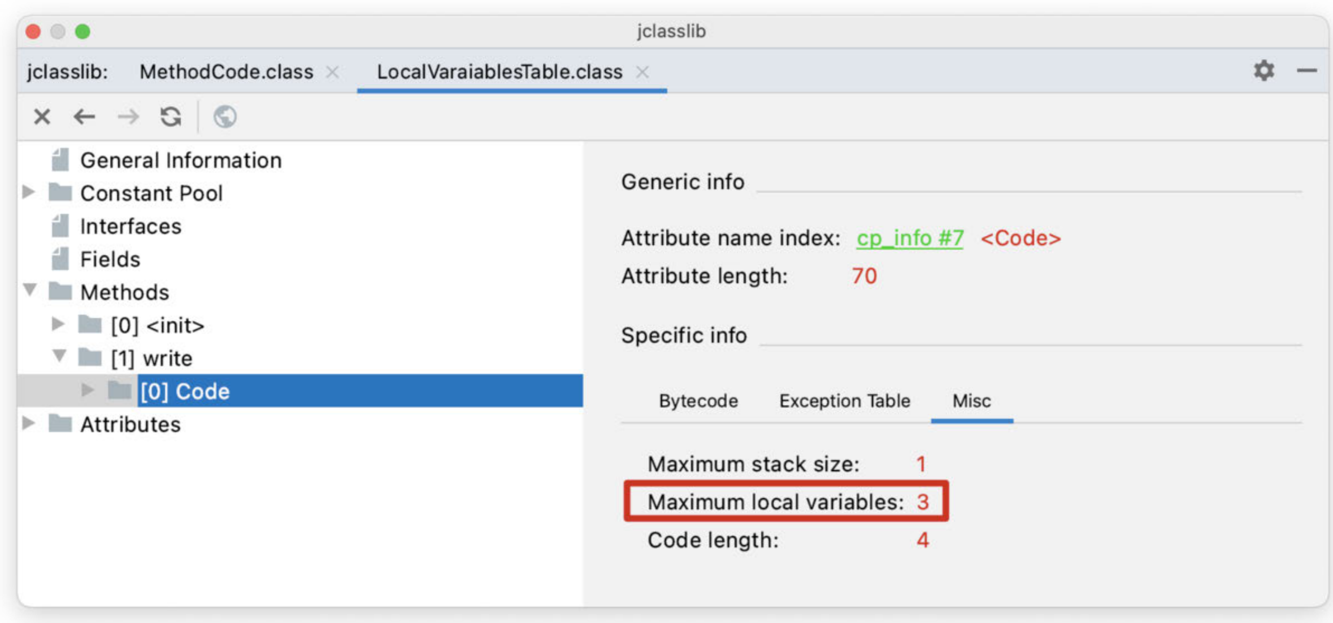
局部变量表（Local Variables Table）用来保存方法中的局部变量，以及方法参数。当 Java 源代码文件被编译成 class 文件的时候，局部变量表的最大容量就已经确定了。

我们来看这样一段代码。

```
public class LocalVaraiablesTable {
    private void write(int age) {
        String name = "沉默王二";
    }
}
```

`write()` 方法有一个参数 age，一个局部变量 name。

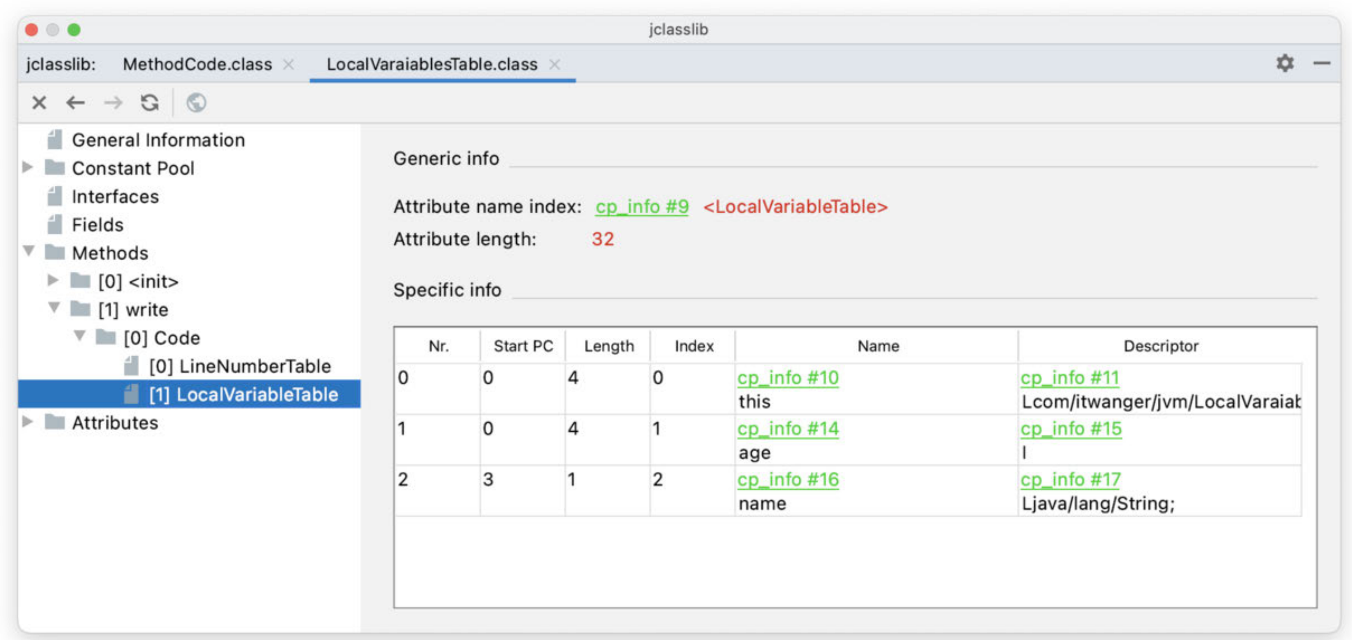
然后用 Intellij IDEA 的 jclasslib 查看一下编译后的字节码文件 LocalVaraiablesTable.class。可以看到 `write()` 方法的 Code 属性中，Maximum local variables（局部变量表的最大容量）的值为 3。



按理说，局部变量表的最大容量应该为 2 才对，一个 age，一个 name，为什么是 3 呢?

当一个成员方法（非静态方法）被调用时，第 0 个变量其实是调用这个成员方法的对象引用，也就是那个大名鼎鼎的 this。调用方法 `write(18)`，实际上是调用 `write(this, 18)`。

点开 Code 属性，查看 LocalVaraiableTable 就可以看到详细的信息了。



第 0 个是 this，类型为 LocalVaraiablesTable 对象；第 1 个是方法参数 age，类型为整型 int；第 2 个是方法内部的局部变量 name，类型为字符串 String。

当然了，局部变量表的大小并不是方法中所有局部变量的数量之和，它与变量的类型和变量的作用域有关。当一个局部变量的作用域结束了，它占用的局部变量表中的位置就被接下来的局部变量取代了。

来看下面这段代码。

```
public static void method() {
    // ①
    if (true) {
        // ②
        String name = "沉默王二";
    }
    // ③
    if(true) {
        // ④
        int age = 18;
    }
    // ⑤
}
```

- `method()` 方法的局部变量表大小为 1，因为是静态方法，所以不需要添加 this 作为局部变量表的第一个元素；
- ②的时候局部变量有一个 name，局部变量表的大小变为 1；
- ③的时候 name 变量的作用域结束；
- ④的时候局部变量有一个 age，局部变量表的大小为 1；
- ⑤的时候局 age 变量的作用域结束；

关于局部变量的作用域，《Effective Java》 中的第 57 条建议：

> 将局部变量的作用域最小化，可以增强代码的可读性和可维护性，并降低出错的可能性。

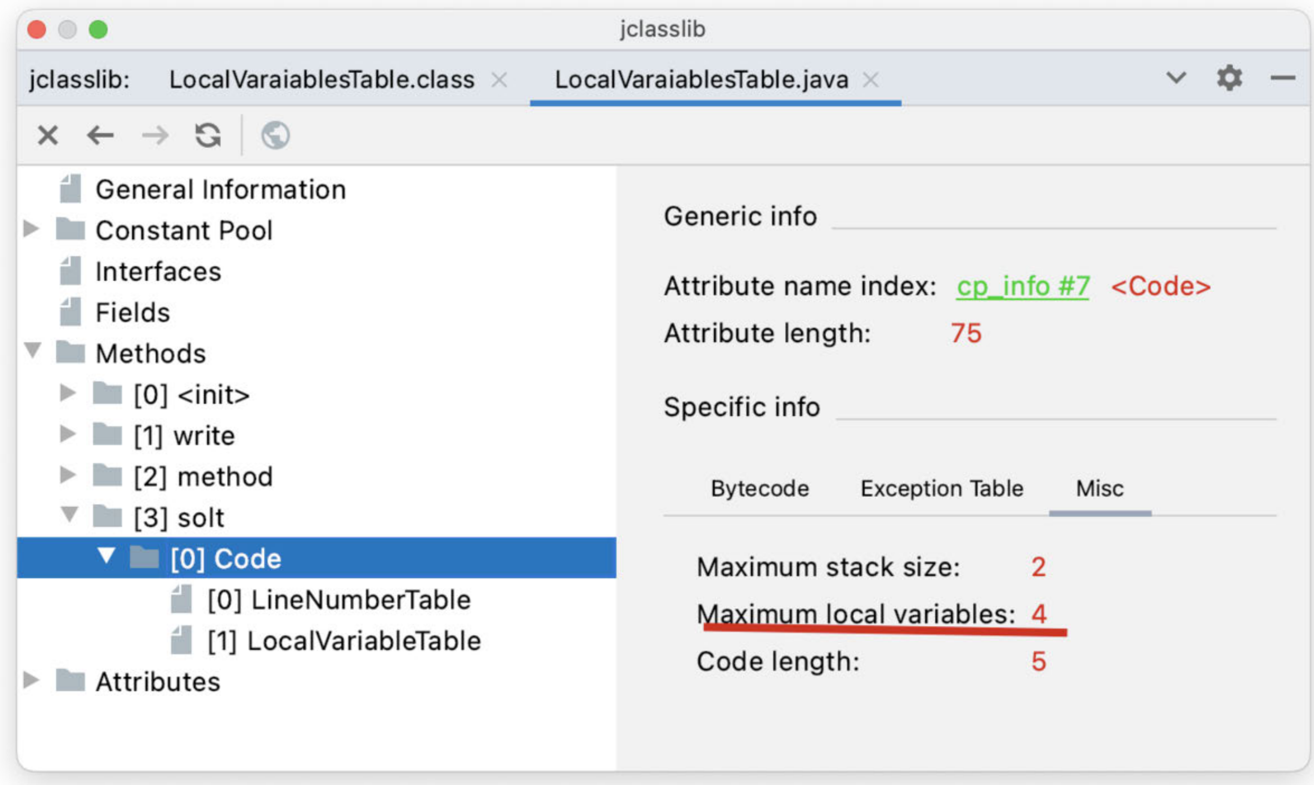
在此，我还有一点要提醒大家。为了尽可能节省栈帧耗用的内存空间，局部变量表中的槽是可以重用的，就像 `method()` 方法演示的那样，这就意味着，合理的作用域有助于提高程序的性能。是不是很有意思？

> 局部变量表的容量以槽（slot）为最小单位，一个槽可以容纳一个 32 位的数据类型（比如说 int，当然了，《Java 虚拟机规范》中没有明确指出一个槽应该占用的内存空间大小，但我认为这样更容易理解），像 float 和 double 这种明确占用 64 位的数据类型会占用两个紧挨着的槽。

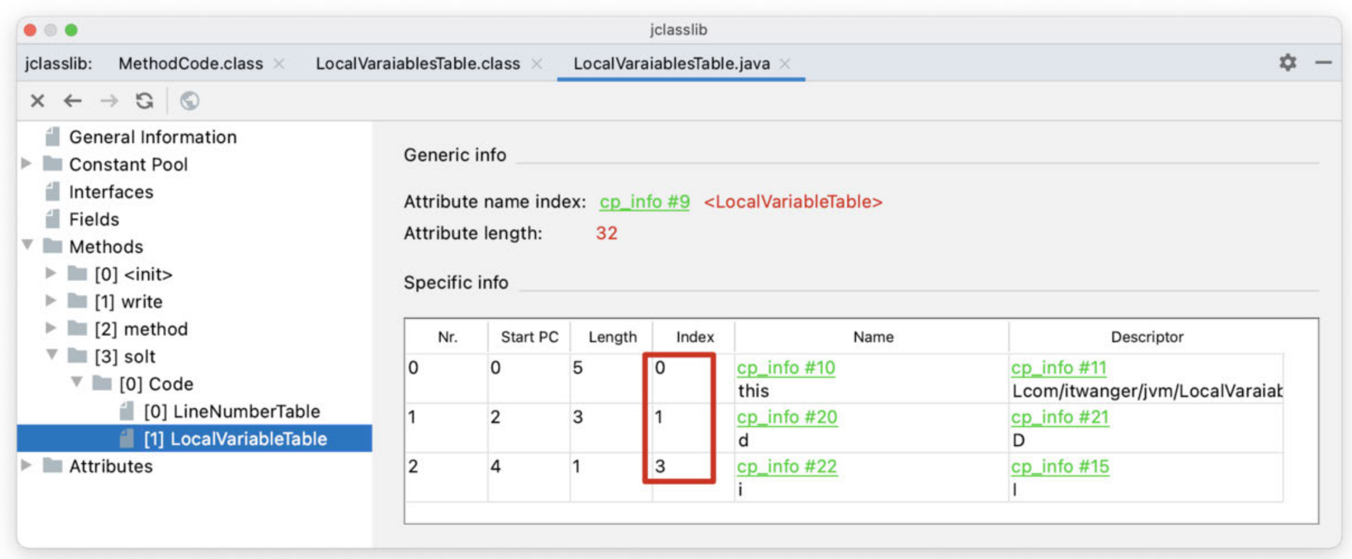
来看下面的代码。

```
public void solt() {
    double d = 1.0;
    int i = 1;
}
```

用 jclasslib 可以查看到，`solt()` 方法的 Maximum local variables 的值为 4。

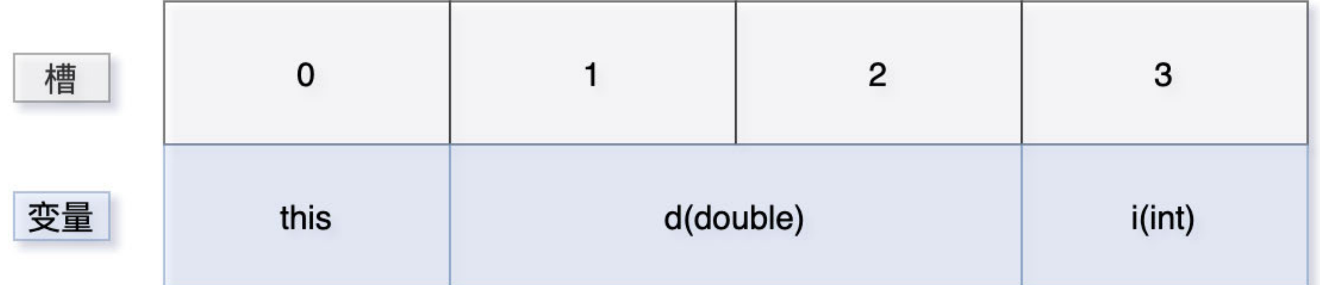


为什么等于 4 呢？带上 this 也就 3 个呀？



Nr.	Start PC	Length	Index	Name	Descriptor
0	0	5	0	cp_info #10 this	cp_info #11 Lcom/itwanger/jvm/LocalVaraiat
1	2	3	1	cp_info #20 d	cp_info #21 D
2	4	1	3	cp_info #22 i	cp_info #15 I

查看 LocalVaraiableTable 就明白了，变量 i 的下标为 3，也就意味着变量 d 占了两个槽。



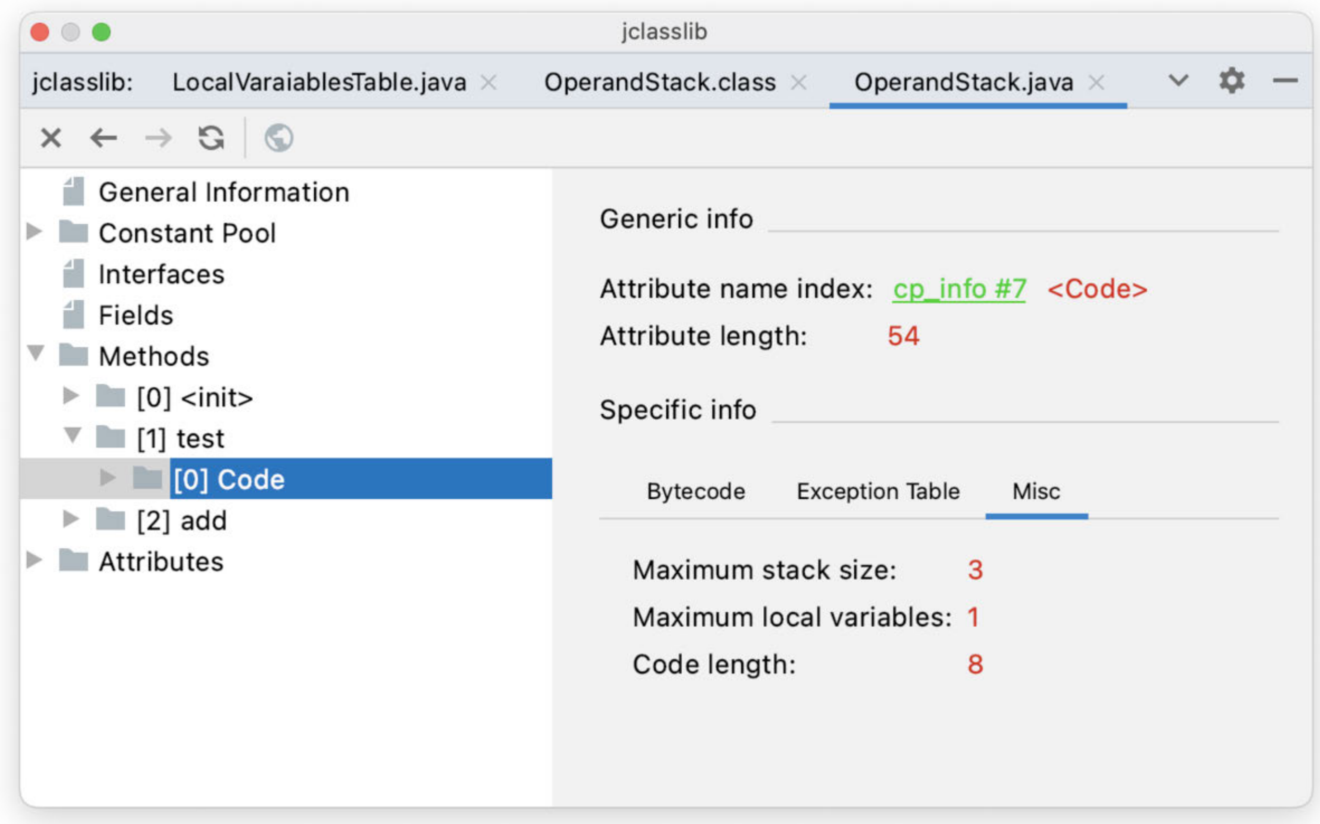
操作数栈

同局部变量表一样，操作数栈（Operand Stack）的最大深度也在编译的时候就确定了，被写入到了 Code 属性的 maximum stack size 中。当一个方法刚开始执行的时候，操作数栈是空的，在方法执行过程中，会有各种字节码指令往操作数栈中写入和取出数据，也就是入栈和出栈操作。

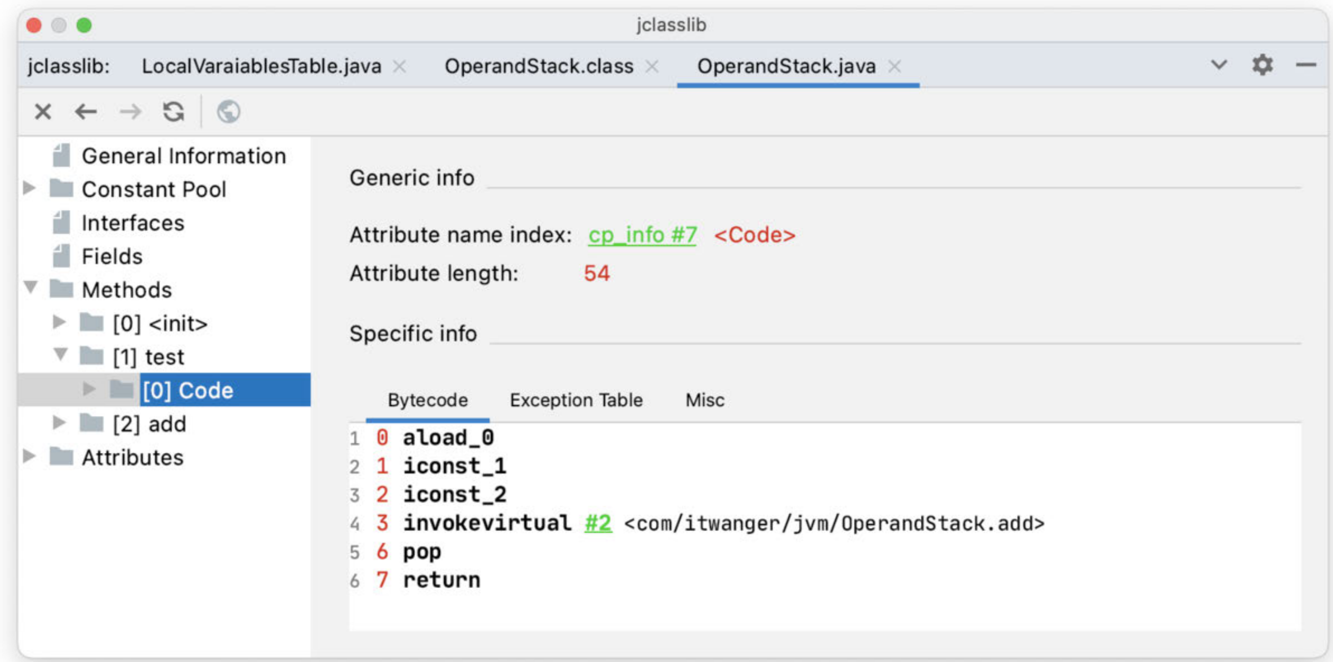
来看下面这段代码。

```
public class OperandStack {
    public void test() {
        add(1,2);
    }

    private int add(int a, int b) {
        return a + b;
    }
}
```

OperandStack 类共有 2 个方法，`test()` 方法中调用了 `add()` 方法，传递了 2 个参数。用 jclasslib 可以看到，`test()` 方法的 maximum stack size 的值为 3。



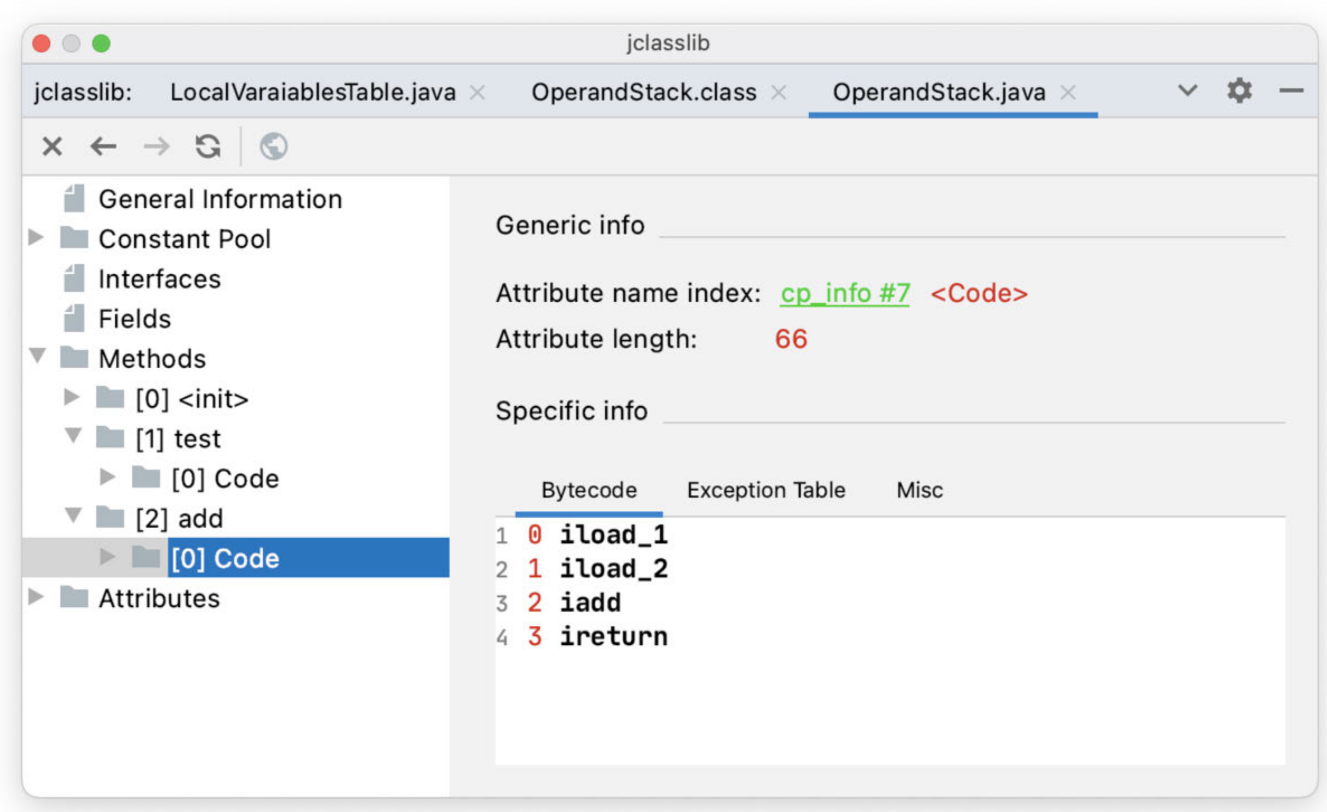
这是因为调用成员方法的时候会将 this 和所有参数压入栈中，调用完毕后 this 和参数都会一一出栈。通过「Bytecode」 面板可以查看到对应的字节码指令。



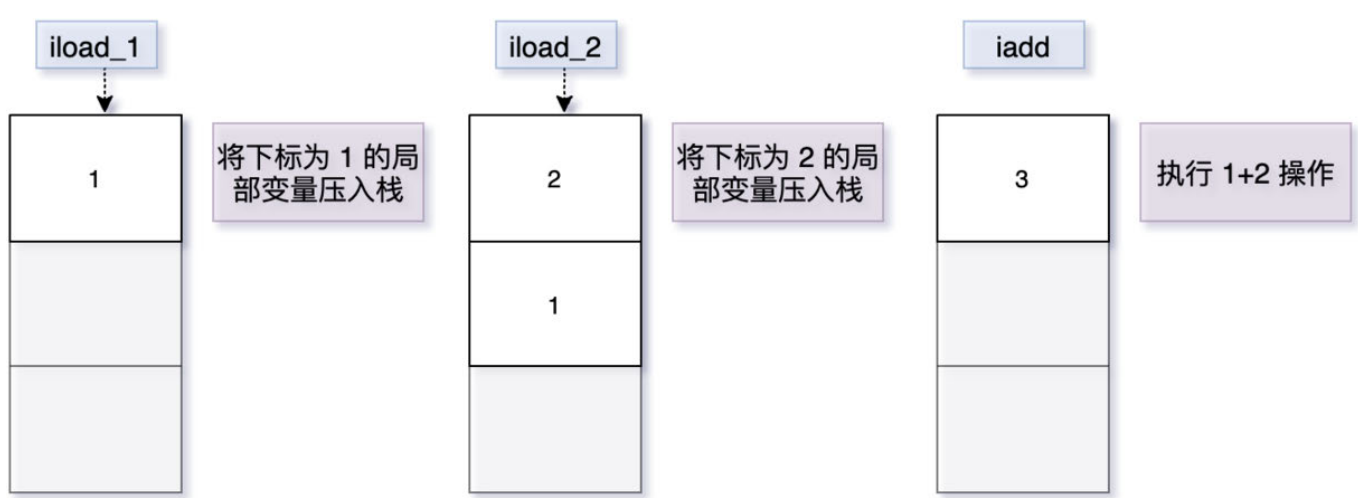
- aload_0 用于将局部变量表中下标为 0 的引用类型的变量，也就是 this 加载到操作数栈中；
- iconst_1 用于将整数 1 加载到操作数栈中；
- iconst_2 用于将整数 2 加载到操作数栈中；
- invokevirtual 用于调用对象的成员方法；
- pop 用于将栈顶的值出栈；

- return 为 void 方法的返回指令。

字节码指令前面我们已经讲过了，忘记的球友可以再回顾一下。再来看一下 `add()` 方法的字节码指令。



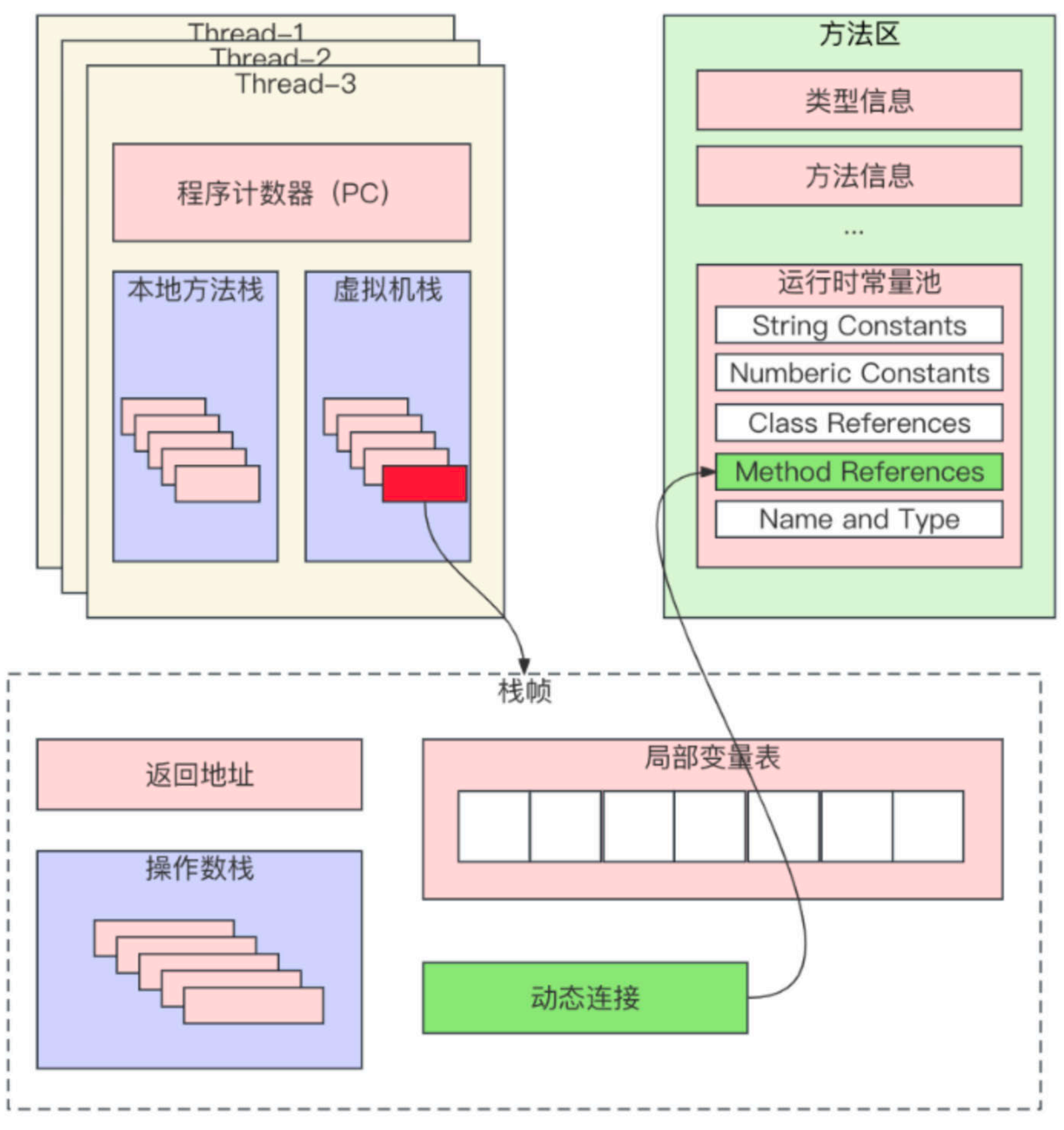
- iload_1 用于将局部变量表中下标为 1 的 int 类型变量加载到操作数栈上（下标为 0 的是 this）；
- iload_2 用于将局部变量表中下标为 2 的 int 类型变量加载到操作数栈上；
- iadd 用于 int 类型的加法运算；
- ireturn 为返回值为 int 的方法返回指令。



操作数中的数据类型必须与字节码指令匹配，以上面的 iadd 指令为例，该指令只能用于整型数据的加法运算，它在执行的时候，栈顶的两个数据必须是 int 类型的，不能出现一个 long 型和一个 double 型的数据进行 iadd 命令相加的情况。

动态链接

每个栈帧都包含了一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态链接（Dynamic Linking）。



①、前面我们就讲过，方法区是 JVM 的一个运行时内存区域，属于逻辑定义，不同版本的 JDK 都有不同的实现，但主要的作用就是用于存储已被虚拟机加载的类信息、常量、静态变量，以及即时编译器编译后的代码等。

②、运行时常量池（Runtime Constant Pool）是方法区的一部分，用于存放编译期生成的各种字面量和符号引用——在类加载后进入运行时常量池。关于方法区我们也会在后面进行详细地讲解。

来看下面这段代码。

```java
public class DynamicLinking {
    static abstract class Human {
       protected abstract void sayHello();
    }

    static class Man extends Human {
        @Override
        protected void sayHello() {
            System.out.println("男人哭吧哭吧不是罪");
        }
    }

    static class Woman extends Human {
        @Override
        protected void sayHello() {
            System.out.println("山下的女人是老虎");
        }
    }

    public static void main(String[] args) {
        Human man = new Man();
        Human woman = new Woman();
        man.sayHello();
        woman.sayHello();
        man = new Woman();
        man.sayHello();
    }
}
```

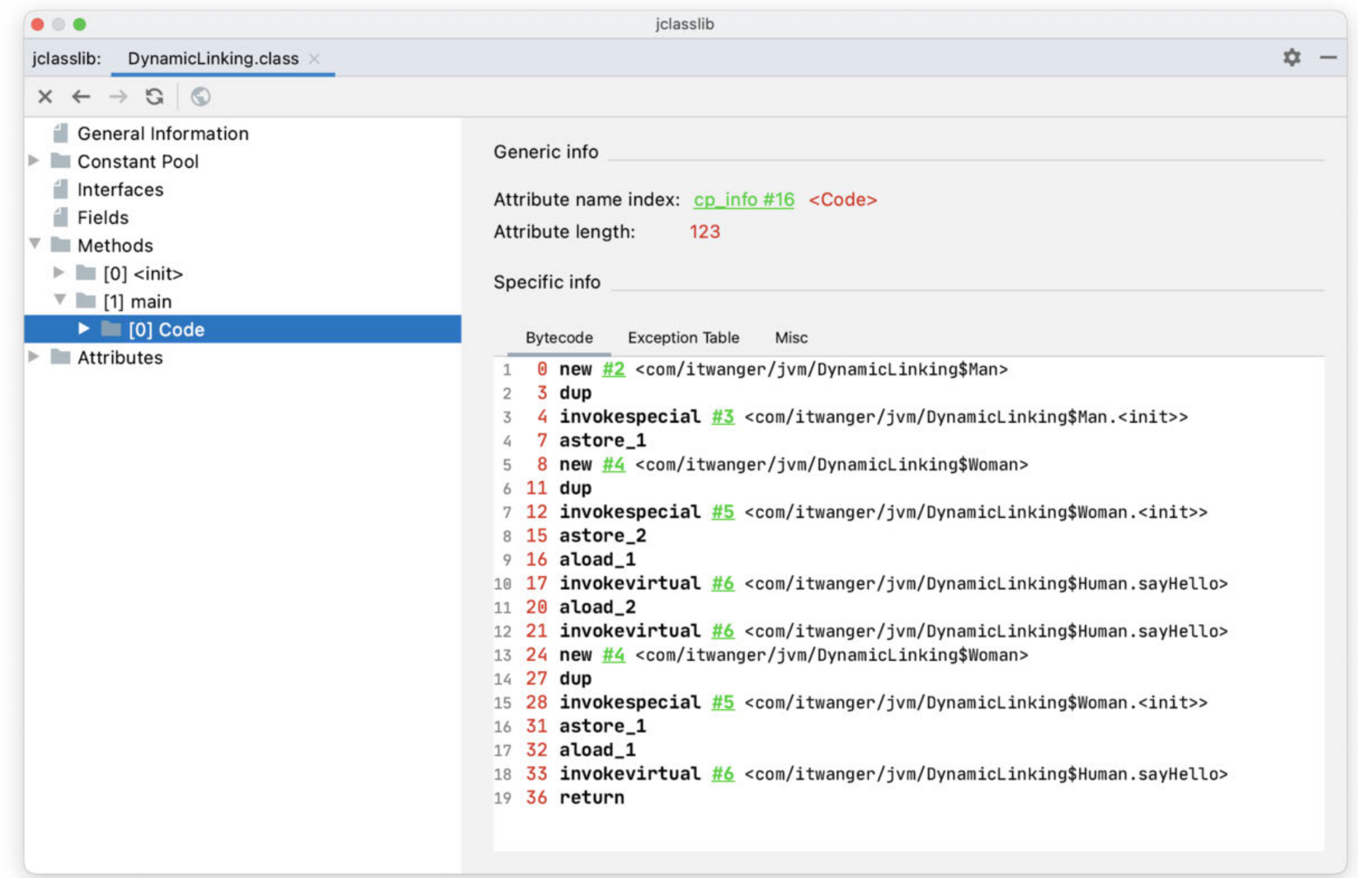
大家对 Java 重写有了解的话，应该能看懂这段代码的意思。Man 类和 Woman 类继承了 Human 类，并且重写了 `sayHello()` 方法。来看一下运行结果：

```
男人哭吧哭吧不是罪
山下的女人是老虎
山下的女人是老虎
```

这个运行结果很好理解，man 的引用类型为 Human，但指向的是 Man 对象，woman 的引用类型也为 Human，但指向的是 Woman 对象；之后，man 又指向了新的 Woman 对象。

从面向对象编程的角度，从多态的角度，我们对运行结果是很好理解的，但站在 Java 虚拟机的角度，它是如何判断 man 和 woman 该调用哪个方法的呢？

用 jclasslib 看一下 main 方法的字节码指令。



- 第 1 行：new 指令创建了一个 Man 对象，并将对象的内存地址压入栈中。
- 第 2 行：dup 指令将栈顶的值复制一份并压入栈顶。因为接下来的指令 invokespecial 会消耗掉一个当前类的引用，所以需要复制一份。
- 第 3 行：invokespecial 指令用于调用构造方法进行初始化。
- 第 4 行：astore_1，Java 虚拟机从栈顶弹出 Man 对象的引用，然后将其存入下标为 1 局部变量 man 中。
- 第 5、6、7、8 行的指令和第 1、2、3、4 行类似，不同的是 Woman 对象。
- 第 9 行：aload_1 指令将第局部变量 man 压入操作数栈中。
- 第 10 行：invokevirtual 指令调用对象的成员方法 `sayHello()`，注意此时的对象类型为 `com/itwanger/jvm/DynamicLinking$Human`。
- 第 11 行：aload_2 指令将第局部变量 woman 压入操作数栈中。
- 第 12 行同第 10 行。

注意，从字节码的角度来看，`man.sayHello()`（第 10 行）和 `woman.sayHello()`（第 12 行）的字节码是完全相同的，但我们都知道，这两句指令最终执行的目标方法并不相同。

究竟发生了什么呢?

还得从 `invokevirtual` 这个指令着手，看它是如何实现多态的。根据《Java 虚拟机规范》，invokevirtual 指令在运行时的解析过程可以分为以下几步：

- ①、找到操作数栈顶的元素所指向的对象的实际类型，记作 C。
- ②、如果在类型 C 中找到与常量池中的描述符匹配的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找结束；否则返回 `java.lang.IllegalAccessError` 异常。
- ③、否则，按照继承关系从下往上一次对 C 的各个父类进行第二步的搜索和验证。
- ④、如果始终没有找到合适的方法，则抛出 `java.lang.AbstractMethodError` 异常。

也就是说，invokevirtual 指令在第一步的时候就确定了运行时的实际类型，所以两次调用中的 invokevirtual 指令并不是把常量池中方法的符号引用解析到直接引用上就结束了，还会根据方法接受者的实际类型来选择方法版本，这个过程就是 Java 重写的本质。我们把这种在运行期根据实际类型确定方法执行版本的过程称为**动态链接**。

方法返回地址

当一个方法开始执行后，只有两种方式可以退出这个方法：

- 正常退出，可能会有返回值传递给上层的方法调用者，方法是否有返回值以及返回值的类型根据方法返回的指令来决定，像之前提到的 ireturn 用于返回 int 类型，return 用于 void 方法；还有其他的一些，lreturn 用于 long 型，freturn 用于 float，dreturn 用于 double，areturn 用于引用类型。
- 异常退出，方法在执行的过程中遇到了异常，并且没有得到妥善的处理，这种情况下，是不会给它的上层调用者返回任何值的。

无论是哪种方式退出，在方法退出后，都必须返回到方法最初被调用时的位置，程序才能继续执行。一般来说，方法正常退出的时候，PC 计数器的值会作为返回地址，栈帧中很可能会保存这个计数器的值，异常退出时则不会。

> PC 计数器：JVM 运行时数据区的一部分，跟踪当前线程执行字节码的位置。

方法退出的过程实际上等同于把当前栈帧出栈，因此接下来可能执行的操作有：恢复上层方法的局部变量表和操作数栈，把返回值（如果有的话）压入调用者栈帧的操作数栈中，调整 PC 计数器的值，找到下一条要执行的指令等。

附加信息

虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧中，例如与调试相关的信息，这部分信息完全取决于具体的虚拟机实现。实际开发中，一般会把动态连接、方法返回地址与其他附加信息全部归为一类，成为栈帧信息。

StackOverflowError

下面这段代码在运行的时候会抛出 StackOverflowError 异常。

```java
public class StackOverflowErrorTest {
    public static void main(String[] args) {
        StackOverflowErrorTest test = new StackOverflowErrorTest();
        test.testStackOverflowError();
    }

    public void testStackOverflowError() {
        testStackOverflowError();
    }
}
```

我们来看一下异常的堆栈信息。

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java ...
objc[13058]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (0x10be34
Exception in thread "main" java.lang.StackOverflowError
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError(StackOverflowErrorTest.java:16)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest.testStackOverflowError
```

之所以抛出 StackOverflowError 异常，是因为在执行 `testStackOverflowError()` 方法的时候，会创建一个栈帧，然后调用 `testStackOverflowError()` 方法，又会创建一个栈帧，然后调用 `testStackOverflowError()` 方法，又会创建一个栈帧......这样一直循环下去，直到栈内存溢出。

我们来简单改造了一下代码，看一下异常的堆栈信息。

```
public class StackOverflowErrorTest1 {
    private static AtomicInteger count = new AtomicInteger(0);
    public static void main(String[] args) {
        while (true) {
            testStackOverflowError();
        }
    }

    public static void testStackOverflowError() {
        System.out.println(count.incrementAndGet());
        testStackOverflowError();
    }
}
```

在 10924 次的时候，抛出了 StackOverflowError 异常。大家可以试试自己的本地环境，看多少次的时候会抛出异常。

```
Run: StackOverflowErrorTest1
10916
10917
10918
10919
10920
10921
10922
10923
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:21)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:21)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:21)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:21)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:21)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:21)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:21)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:21)
```

小结

栈帧是 JVM 中用于方法执行的数据结构，每当一个方法被调用时，JVM 会为该方法创建一个栈帧，并在方法执行完毕后销毁。

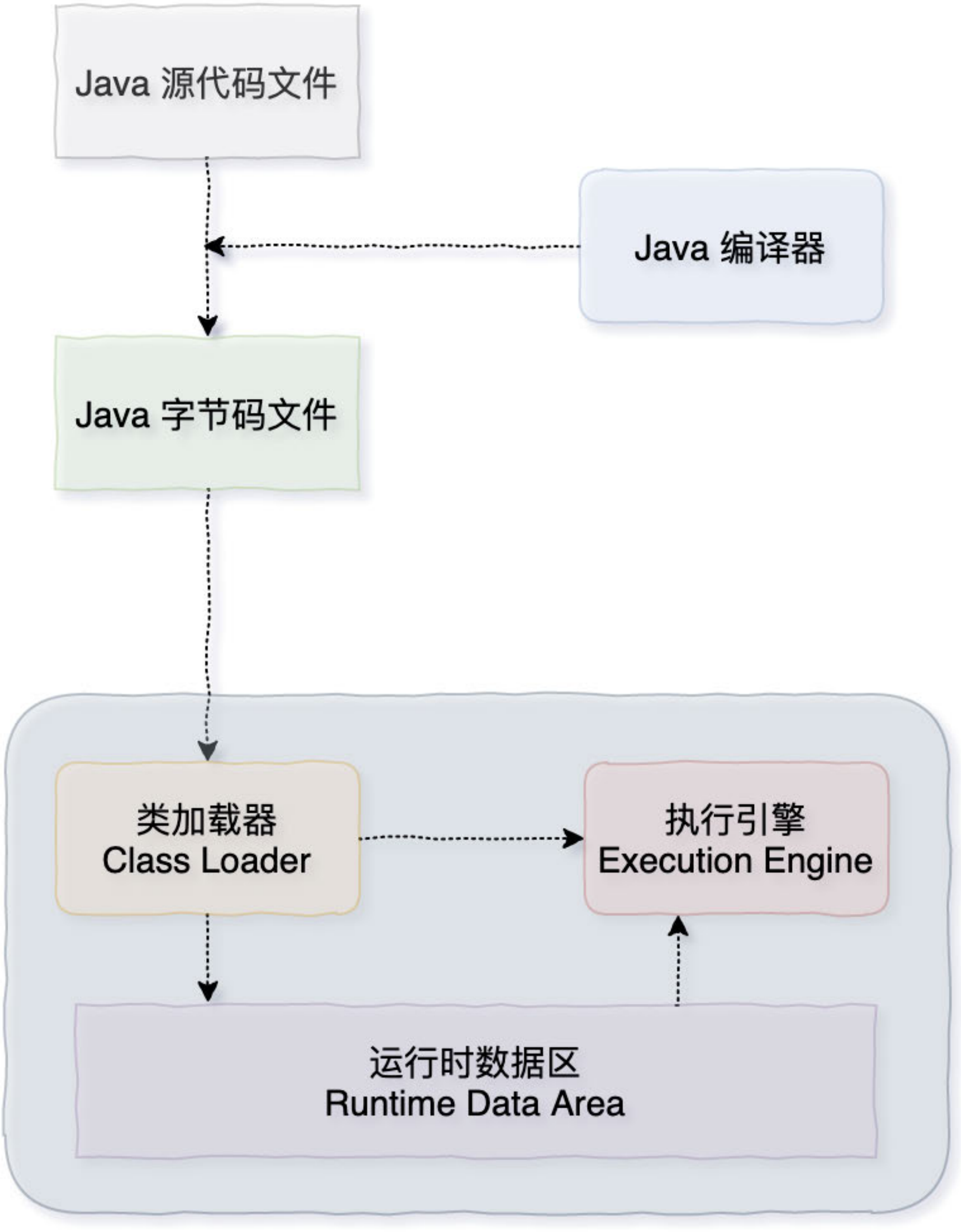
- **局部变量表**：存储方法的参数和局部变量，由基本数据类型或对象引用组成。
- **操作数栈**：后进先出（LIFO）的栈结构，用于存储操作数和中间计算结果。
- **动态链接**：关联到方法所属类的常量池，支持动态方法调用。
- **方法返回地址**：记录方法结束后控制流应返回的位置。

栈帧是线程私有的，每个线程有自己的 JVM 栈。方法调用时，新栈帧被推入栈顶；方法完成后，栈帧出栈。

栈帧的局部变量表的大小和操作数栈的最大深度在编译时就已确定。栈空间不足时可能引发 `StackOverflowError`。理解栈帧对于深入理解 Java 程序的运行机制至关重要。

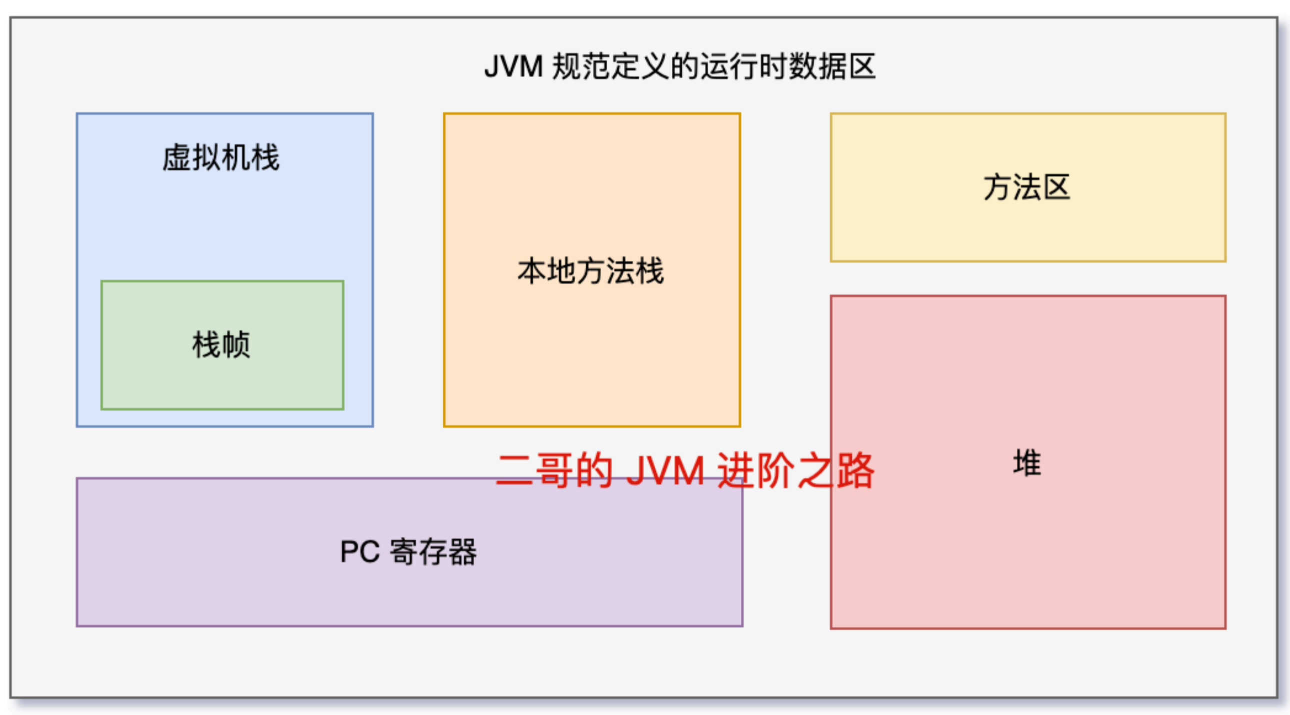
第九节：深入理解运行时数据区

前面我们就讲过，Java 源代码文件经过编译器编译后会生成字节码文件，经过加载器加载完毕后会交给执行引擎执行。在执行的过程中，JVM 会划出来一块空间来存储程序执行期间需要用到的数据，这块空间一般被称为运行时数据区，见下图。



根据 Java 虚拟机规范的规定，运行时数据区可以分为以下几个部分：

- 程序计数器（Program Counter Register）
- Java 虚拟机栈（Java Virtual Machine Stacks）
- 本地方法栈（Native Method Stack）
- 堆（Heap）
- 方法区（Method Area）



> JDK 8 开始，永久代被彻底移除，取而代之的是元空间。元空间不再是 JVM 内存的一部分，而是通过本地内存（Native Memory）来实现的。也就是说，JDK 8 开始，方法区的实现就是元空间。

程序计数器

程序计数器（Program Counter Register）所占的内存空间不大，很小很小一块，可以看作是当前线程所执行的字节码指令的行号指示器。字节码解释器会在工作的时候改变这个计数器的值来选取下一条需要执行的字节码指令，像分支、循环、跳转、异常处理、线程恢复等功能都需要依赖这个计数器来完成。

在 JVM 中，多线程是通过线程轮流切换来获得 CPU 执行时间的，因此，在任一具体时刻，一个 CPU 的内核只会执行一条线程中的指令，因此，为了线程切换后能恢复到正确的执行位置，每个线程都需要有一个独立的程序计数器，并且不能互相干扰，否则就会影响到程序的正常执行次序。

也就是说，我们要求**程序计数器是线程私有的**。

《Java 虚拟机规范》中规定，如果线程执行的是非本地方法，则程序计数器中保存的是当前需要执行的指令地址；如果线程执行的是本地方法，则程序计数器中的值是 undefined。

为什么本地方法在程序计数器中的值是 undefined 的？因为本地方法大多是通过 C/C++ 实现的，并未编译成需要执行的字节码指令。

我们来通过代码以及字节码指令来看看程序计数器的作用。

```
public static int add(int a, int b) {
    return a + b;
}
```

字节码指令大致如下：

```
0: iload_0       // 从局部变量表中加载变量 a 到操作数栈
1: iload_1       // 从局部变量表中加载变量 b 到操作数栈
2: iadd          // 两数相加
3: ireturn       // 返回
```

现在，让我们逐步分析程序计数器是如何在执行这些指令时更新的：

1. **初始状态**：当方法开始执行时，PC 计数器设置为 0，指向第一条指令 `0: iload_0`。
2. **执行第一条指令**：
    - 执行 `iload_0` 指令，将局部变量表中索引为 0 的整数（即方法的第一个参数 `a`）加载到操作数栈顶。
    - 执行完成后，PC 计数器更新为 1，指向下一条指令 `1: iload_1`。
3. **执行第二条指令**：
    - 执行 `iload_1` 指令，将局部变量表中索引为 1 的整数（即方法的第二个参数 `b`）加载到操作数栈顶。
    - 执行完成后，PC 计数器更新为 2，指向下一条指令 `2: iadd`。
4. **执行第三条指令**：
    - 执行 `iadd` 指令，弹出操作数栈顶的两个整数（即 `a` 和 `b`），将它们相加，然后将结果压入操作数栈顶。
    - 执行完成后，PC 计数器更新为 3，指向下一条指令 `3: ireturn`。
5. **执行最后一条指令：**
    - 执行 `ireturn` 指令，弹出操作数栈顶的整数（即 `a + b` 的结果），并将这个值作为方法的返回值。
    - 方法执行完成，控制权返回到方法调用者。

Java 虚拟机栈

Java 虚拟机栈（JVM 栈）中是一个个栈帧，每个栈帧对应一个被调用的方法。当线程执行一个方法时，会创建一个对应的栈帧，并将栈帧压入栈中。当方法执行完毕后，将栈帧从栈中移除。

栈帧包含以下 5 个部分，见下图。我们前面已经详细地讲过栈帧了，忘记的球友可以回头去看一下。

假设我们有一个简单的 add 方法，如下所示：

```
public int add(int a, int b) {
    int result = a + b;
    return result;
}
```

当 `add` 方法被调用时，JVM 为这次方法调用创建一个新的栈帧。然后执行方法内的字节码指令，这部分我们前面已经讲过了，大家可以自己通过 javap 查看字节码并模拟一下字节码指令执行的过程。

当 `add` 方法执行完毕后，对应的栈帧会从 JVM 栈中弹出。

Java 虚拟机栈的特点如下：

- **线程私有：** 每个线程都有自己的 JVM 栈，线程之间的栈是不共享的。
- **栈溢出：** 如果栈的深度超过了 JVM 栈所允许的深度，将会抛出 `StackOverflowError`，这个我们讲栈帧的时候讲过了。

大家可以猜一下 JVM 栈的默认大小是多少？

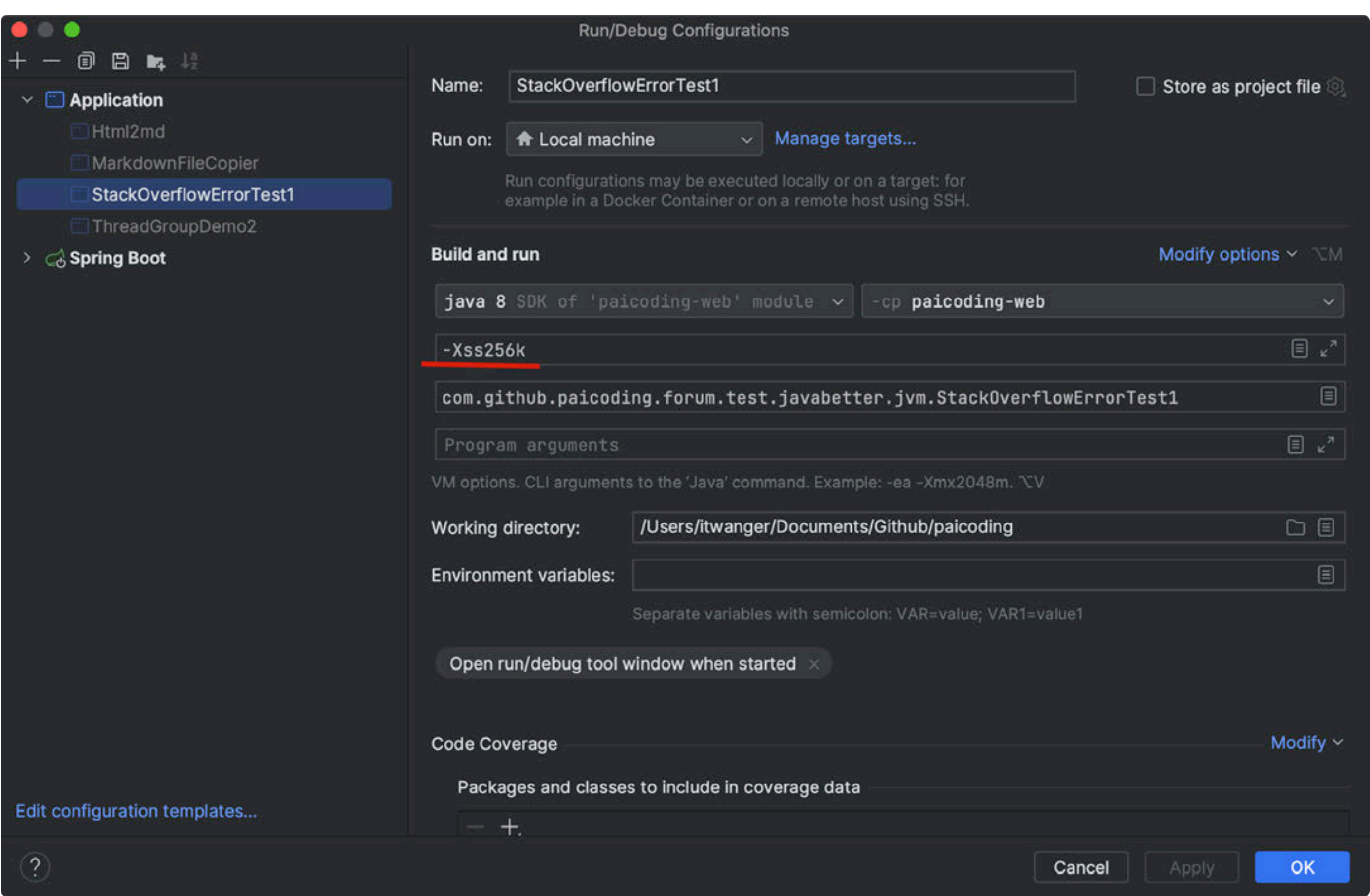
还用我们之前的讲栈帧时候的例子：

```
public class StackOverflowErrorTest1 {
    private static AtomicInteger count = new AtomicInteger(0);
    public static void main(String[] args) {
        while (true) {
            testStackOverflowError();
        }
    }

    public static void testStackOverflowError() {
        System.out.println(count.incrementAndGet());
        testStackOverflowError();
    }
}
```

默认配置下，堆栈异常出现在 10886 次：

```
Run    StackOverflowErrorTest1 ×
10881
10882
10883
10884
10885
*** java.lang.instrument ASSERTION FAILED ***: "!errorOutstanding" with message transform method call failed at JPLISAgent.c line: 844
*** java.lang.instrument ASSERTION FAILED ***: "!errorOutstanding" with message transform method call failed at JPLISAgent.c line: 844
*** java.lang.instrument ASSERTION FAILED ***: "!errorOutstanding" with message transform method call failed at JPLISAgent.c line: 844
Exception in thread "main" java.lang.StackOverflowError Create breakpoint
    at sun.nio.cs.UTF_8$Encoder.encodeLoop(UTF_8.java:691)
    at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
    at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:271)
    at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:125)
    at java.io.OutputStreamWriter.write(OutputStreamWriter.java:207)
    at java.io.BufferedWriter.flushBuffer(BufferedWriter.java:129)
    at java.io.PrintStream.newLine(PrintStream.java:545)
    at java.io.PrintStream.println(PrintStream.java:737)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:20)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:21)
    at com.github.paicoding.forum.test.javabetter.jvm.StackOverflowErrorTest1.testStackOverflowError(StackOverflowErrorTest1.java:21)
```

增加 `-Xss256k` 后，来试试。



1991 次出现了堆栈异常。

```
1984
1985
1986
1987
1988
1989
1990
*** java.lang.instrument ASSERTION FAILED ***: "!errorOutstanding" with message transform method call failed at JPLISAgent.c line: 844
*** java.lang.instrument ASSERTION FAILED ***: "!errorOutstanding" with message transform method call failed at JPLISAgent.c line: 844
*** java.lang.instrument ASSERTION FAILED ***: "!errorOutstanding" with message transform method call failed at JPLISAgent.c line: 844
Exception in thread "main" java.lang.StackOverflowError Create breakpoint
    at sun.nio.cs.UTF_8.updatePositions(UTF_8.java:77)
    at sun.nio.cs.UTF_8.access$200(UTF_8.java:57)
    at sun.nio.cs.UTF_8$Encoder.encodeArrayLoop(UTF_8.java:636)
    at sun.nio.cs.UTF_8$Encoder.encodeLoop(UTF_8.java:691)
    at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
    at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:271)
    at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:125)
    at java.io.OutputStreamWriter.write(OutputStreamWriter.java:207)
    at java.io.BufferedWriter.flushBuffer(BufferedWriter.java:129)
    at java.io.PrintStream.write(PrintStream.java:526)
    at java.io.PrintStream.print(PrintStream.java:597)
```

这之间存在什么关系呢?

通过 `java -XX:+PrintFlagsFinal -version | grep ThreadStackSize` 这个命令可以查看 JVM 栈的默认大小。

```
maweiqing@itwanger-2 paicoding-web % java -XX:+PrintFlagsFinal -version | grep ThreadStackSize

     intx CompilerThreadStackSize                   = 0                                   {pd product}
     intx ThreadStackSize                           = 1024                                {pd product}
     intx VMThreadStackSize                         = 1024                                {pd product}
java version "1.8.0_301"
Java(TM) SE Runtime Environment (build 1.8.0_301-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.301-b09, mixed mode)
```

其中 `ThreadStackSize` 的单位是字节，也就是说默认的 JVM 栈大小是 1024 KB，也就是 1M。

也就是说，默认 1024 KB 的 JVM 栈可以执行 10885 次 `testStackOverflowError` 方法，而 256 KB 的 JVM 栈只能执行 1990 次 `testStackOverflowError` 方法，四五倍的样子。

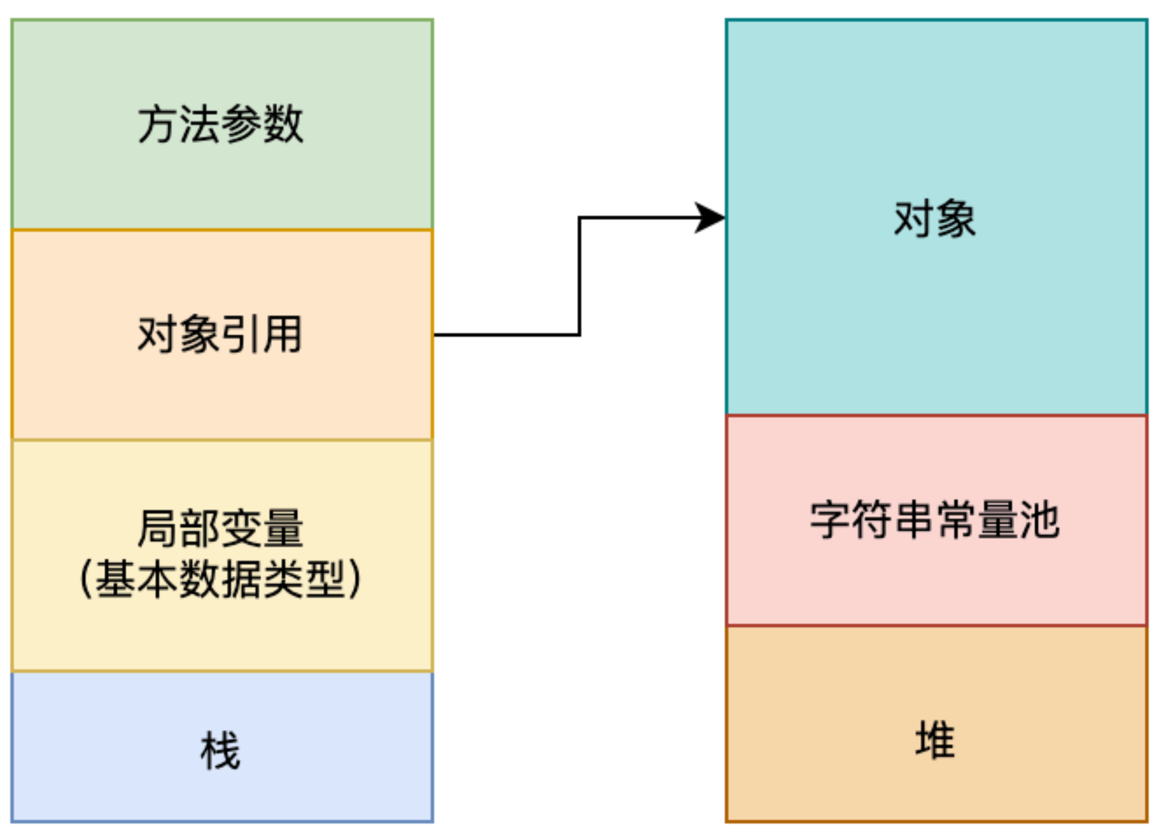
本地方法栈

本地方法栈（Native Method Stack）与 Java 虚拟机栈类似，只不过 Java 虚拟机栈为虚拟机执行 Java 方法服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。

堆

堆是所有线程共享的一块内存区域，在 JVM 启动的时候创建，用来存储对象（数组也是一种对象）。

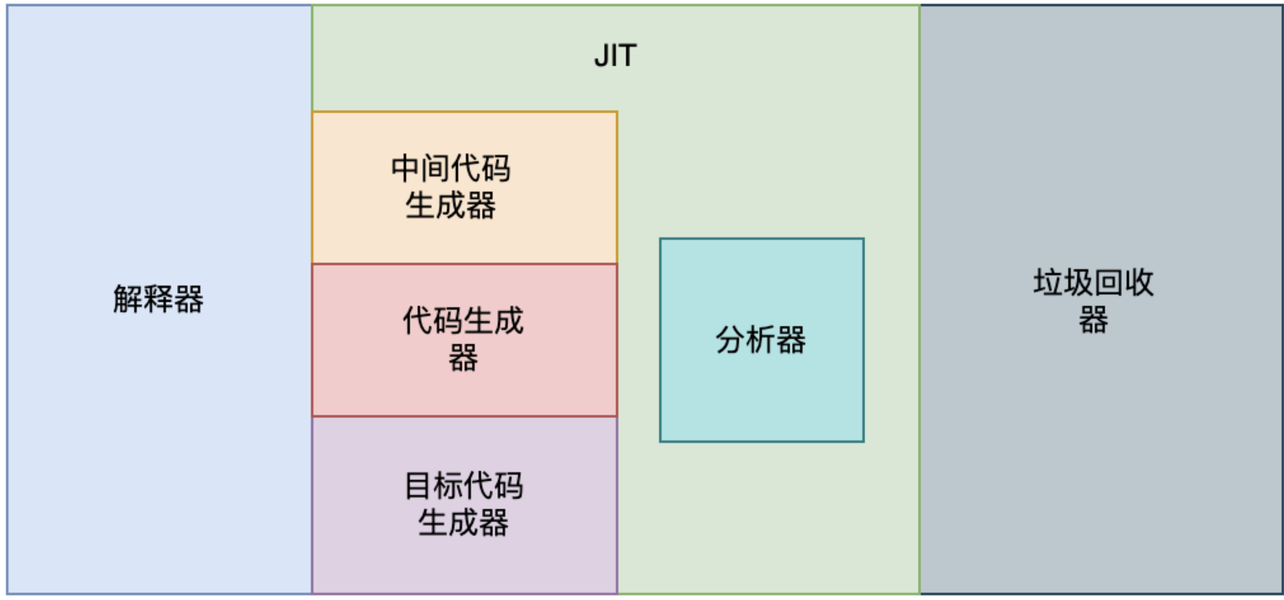
以前，Java 中“几乎”所有的对象都会在堆中分配，但随着 JIT 编译器的发展和逃逸技术的逐渐成熟，所有的对象都分配到堆上渐渐变得不那么“绝对”了。从 JDK 7 开始，Java 虚拟机已经默认开启逃逸分析了，意味着如果某些方法中的对象引用没有被返回或者未被外面使用（也就是未逃逸出去），那么对象可以直接在栈上分配内存。



栈就是前面提到的 JVM 栈（主要存储局部变量、方法参数、对象引用等），属于线程私有，通常随着方法调用的结束而消失，也就无需进行垃圾收集；堆前面也讲了，属于线程共享的内存区域，几乎所有的对象都在对上分配，生命周期不由单个方法调用所决定，可以在方法调用结束后继续存在，直到不在被任何变量引用，然后被垃圾收集器回收。

简单解释一下 JIT 和逃逸分析（后面讲 JIT 会细讲）。

常见的编译型语言如 C++，通常会把代码直接编译成 CPU 所能理解的机器码来运行。而 Java 为了实现“一次编译，处处运行”的特性，把编译的过程分成两部分，首先它会先由 javac 编译成通用的中间形式——字节码，然后再由解释器逐条将字节码解释为机器码来执行。所以在性能上，Java 可能会干不过 C++ 这类编译型语言。



为了优化 Java 的性能 ，JVM 在解释器之外引入了 JIT 编译器：当程序运行时，解释器首先发挥作用，代码可以直接执行。随着时间推移，即时编译器逐渐发挥作用，把越来越多的代码编译优化成本地代码，来获取更高的执行效率。解释器这时可以作为编译运行的降级手段，在一些不可靠的编译优化出现问题时，再切换回解释执行，保证程序可以正常运行。

逃逸分析（Escape Analysis）是一种编译器优化技术，用于判断对象的作用域和生命周期。如果编译器确定一个对象不会逃逸出方法或线程的范围，它可以选择在栈上分配这个对象，而不是在堆上。这样做可以减少垃圾回收的压力，并提高性能。

我们来写一段可能触发栈分配的代码。

```
public class EscapeAnalysisExample {

    private static class Point {
        private int x;
        private int y;

        Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        int calculate() {
            return x + y;
        }
    }

    public static void main(String[] args) {
        int total = 0;
        for (int i = 0; i < 1000000; i++) {
```

```
            total += createAndCalculate();
        }
        System.out.println(total);
    }

    private static int createAndCalculate() {
        Point p = new Point(1, 2);
        return p.calculate();
    }
}
```

- createAndCalculate 方法创建了一个 Point 对象，并调用它的 calculate 方法。
- Point 对象在 createAndCalculate 方法中创建，并且不会逃逸到该方法之外。
- 如果 JVM 的逃逸分析确定 Point 对象不会逃逸出 createAndCalculate 方法，它可能会在栈上分配 Point 对象，而不是在堆上。

堆我们前面已经讲过了，它除了是对象的聚集地，也是 Java 垃圾收集器管理的主要区域，因此也被称作 GC 堆（Garbage Collected Heap）。从垃圾回收的角度来看，由于垃圾收集器基本都采用了分代垃圾收集的算法，所以堆还可以细分为：新生代和老年代。新生代还可以细分为：Eden 空间、From Survivor、To Survivor 空间等。进一步划分的目的是更好地回收内存，或者更快地分配内存。

> 不要担心，这些我们会放到后面垃圾回收的章节来细讲。

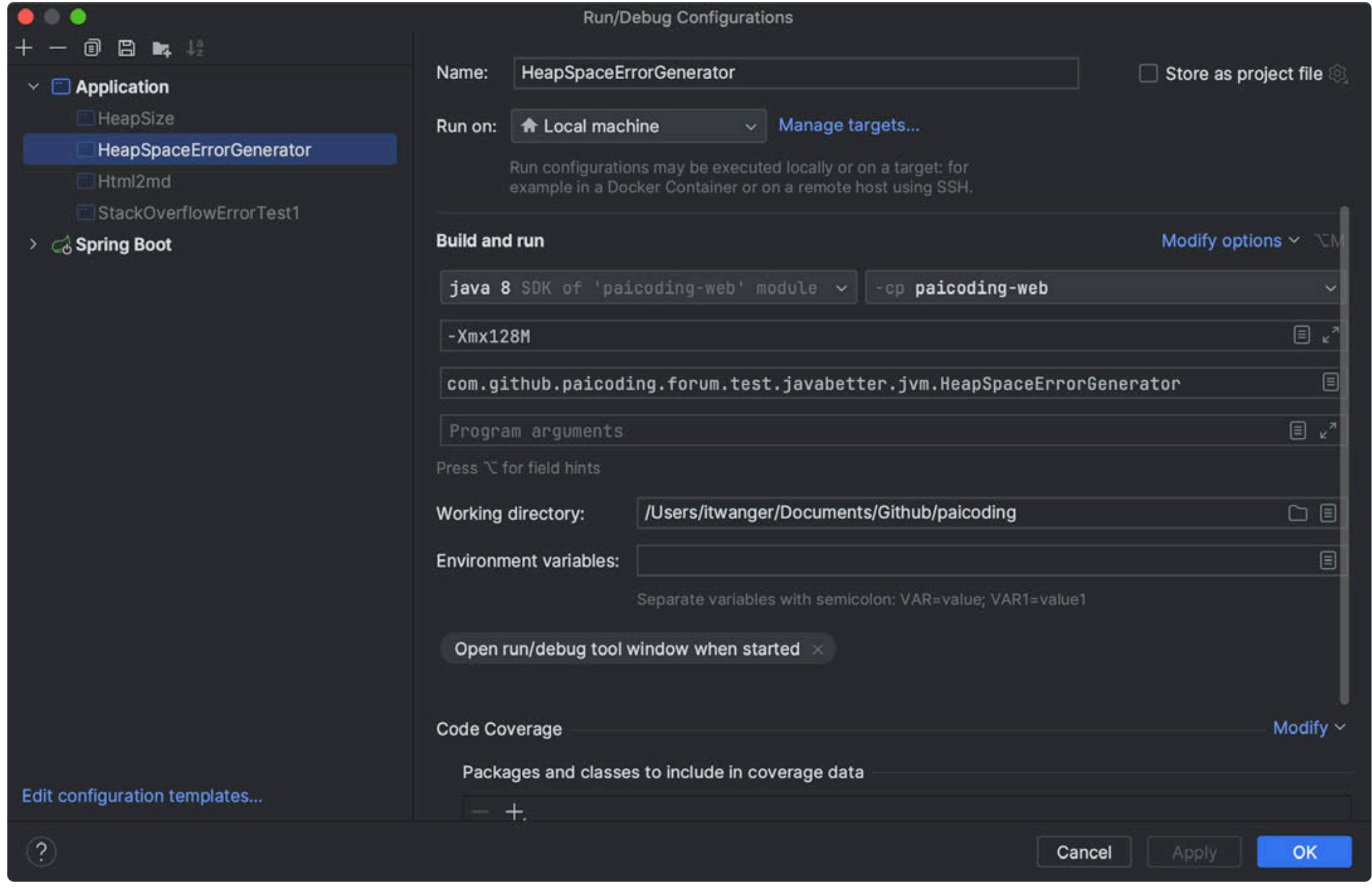
堆这最容易出现的就是 OutOfMemoryError 错误，分为以下几种表现形式：

- `OutOfMemoryError: GC Overhead Limit Exceeded`：当 JVM 花太多时间执行垃圾回收并且只能回收很少的堆空间时，就会发生该错误。
- `java.lang.OutOfMemoryError: Java heap space`：假如在创建新的对象时, 堆内存中的空间不足以存放新创建的对象, 就会引发该错误。和本机的物理内存无关，和我们配置的虚拟机内存大小有关！

我们先来通过代码模拟一下堆内存溢出的情况。

```
public class HeapSpaceErrorGenerator {
    public static void main(String[] args) {
        List<byte[]> bigObjects = new ArrayList<>();
        try {
            while (true) {
                // 创建一个大约 10MB 的数组
                byte[] bigObject = new byte[10 * 1024 * 1024];
                bigObjects.add(bigObject);
            }
        } catch (OutOfMemoryError e) {
            System.out.println("OutOfMemoryError 发生在 " + bigObjects.size() + " 对象后");
            throw e;
        }
    }
}
```

通过 VM 参数设置堆内存大小为 `-Xmx128M`，然后运行程序。



可以看到，堆内存溢出发生在 11 个对象后。

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_301.jdk/Contents/Home/bin/java ...
OutOfMemoryError 发生在 11 对象后
Exception in thread "main" java.lang.OutOfMemoryError Create breakpoint : Java heap space
    at com.github.paicoding.forum.test.javabetter.jvm.HeapSpaceErrorGenerator.main(HeapSpaceErrorGenerator.java:12)

Process finished with exit code 1
```

默认的堆内存大小是多少呢?

通过 `java -XX:+PrintFlagsFinal -version | grep HeapSize` 这个命令可以查看 JVM 堆的默认大小。

```
maweiqing@itwanger-2 paicoding-web % java -XX:+PrintFlagsFinal -version | grep HeapSize
    uintx ErgoHeapSizeLimit                         = 0                                   {product}
    uintx HeapSizePerGCThread                       = 87241520                            {product}
    uintx InitialHeapSize                          := 268435456                           {product}
    uintx LargePageHeapSizeThreshold                = 134217728                           {product}
    uintx MaxHeapSize                              := 4294967296                          {product}
java version "1.8.0_301"
Java(TM) SE Runtime Environment (build 1.8.0_301-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.301-b09, mixed mode)
```

也可以通过下面这行代码获取：

```
System.out.println(Runtime.getRuntime().maxMemory() / 1024.0 / 1024 + "MB");
```

大家可以通过上面的方法查看一下自己本机电脑的堆内存大小。

元空间和方法区

方法区是 Java 虚拟机规范上的一个逻辑区域，在不同的 JDK 版本上有着不同的实现。在 JDK 7 的时候，方法区被称为永久代（PermGen），而在 JDK 8 的时候，永久代被彻底移除，取而代之的是元空间。

如果你在有些资料上依然看到了永久代，要么就是二哥这样在给你解释，要么就是内容过时了。

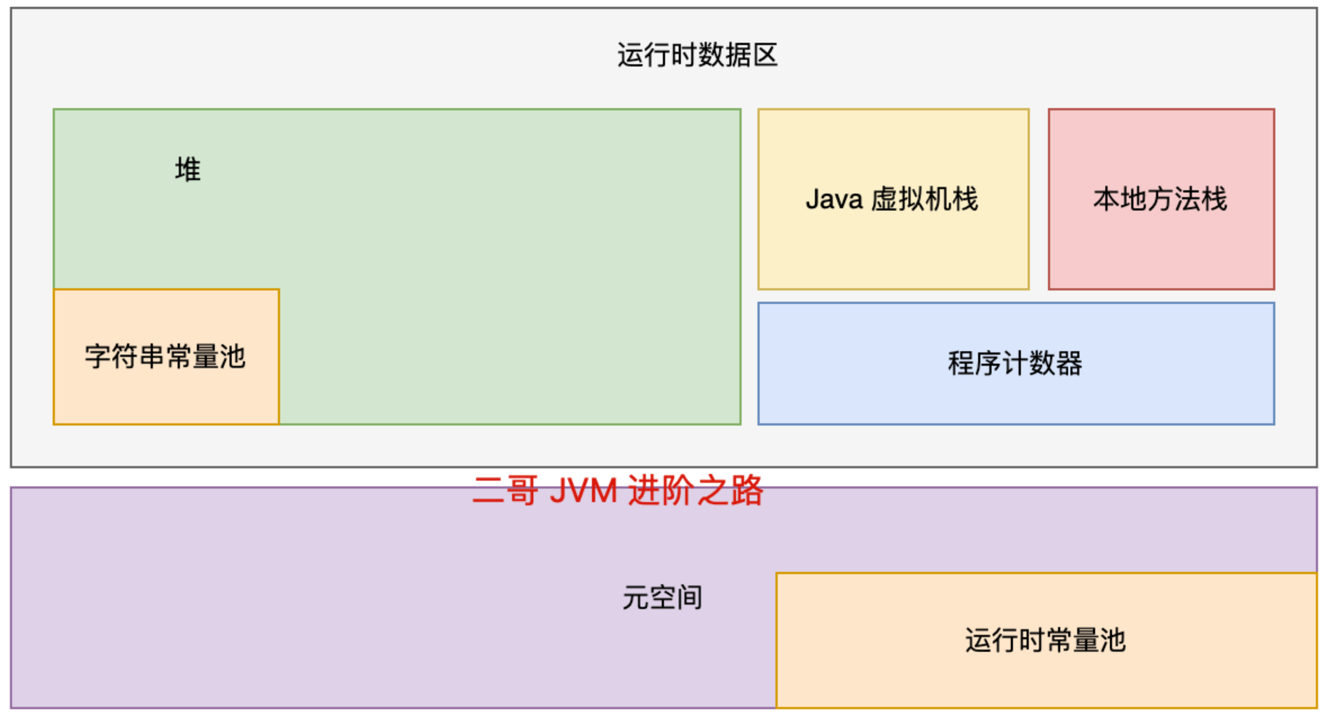
> 《Java 虚拟机规范》中只规定了有方法区这么一个概念和它的作用，并没有规定如何去实现它。不同的 Java 虚拟机可能就会有不同的实现。永久代是 HotSpot 对方法区的一种实现形式。也就是说，永久代是 HotSpot 旧版本中的一个实现，而方法区则是 Java 虚拟机规范中的一个定义，一种规范。

换句话说，方法区和永久代的关系就像是 Java 中接口和类的关系，类实现了接口，接口还是那个接口，但实现已经完全升级了。

JDK 7 之前，只有常量池的概念，都在方法区中。

JDK 7 的时候，字符串常量池从方法区中拿出来放到了堆中，运行时常量池还在方法区中（也就是永久代中）。

JDK 8 的时候，HotSpot 移除了永久代，取而代之的是元空间。字符串常量池还在堆中，而运行时常量池跑到了元空间。



为什么要废弃永久代，而使用元空间来进行替换呢?

旧版的 Hotspot 虚拟机是没有 JIT 的，而 Oracle 旗下的另外一款虚拟机 JRocket 是有的，那为了将 Java 帝国更好的传下去，Oracle 就想把庶长子 JRocket 的 JIT 技术融合到嫡长子 Hotspot 中。

但 JRockit 虚拟机中并没有永久代的概念，因此新的 HotSpot 索性就不要永久代了，直接占用操作系统的一部分内存好了，并且把这块内存取名叫做元空间。

元空间的大小不再受限于 JVM 启动时设置的最大堆大小，而是直接利用本地内存，也就是操作系统的内存。有效地解决了 OutOfMemoryError 错误。

> 可以通过 `java -XX:+PrintFlagsFinal -version | grep HeapSize` 查看 JVM 默认的堆内存大小。

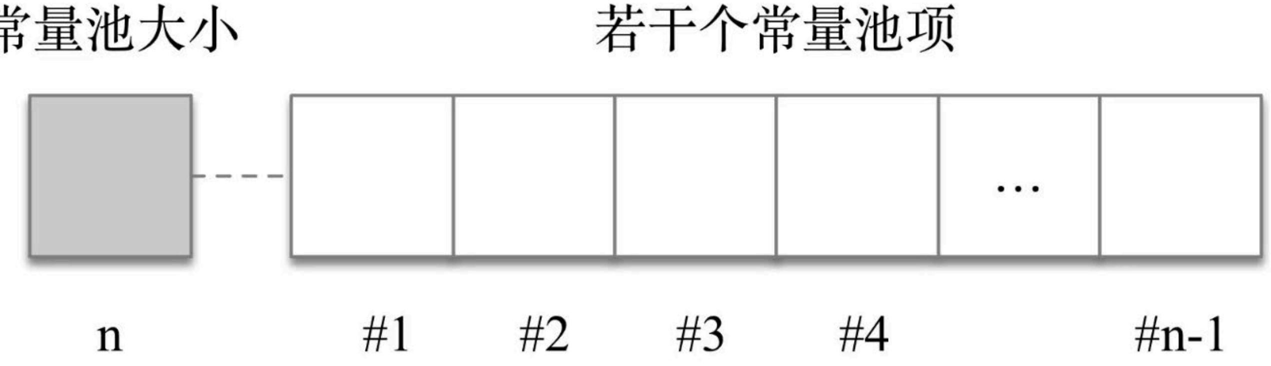
```
→ jvm git:(main) ✗ java -XX:+PrintFlagsFinal -version | grep HeapSize
    uintx ErgoHeapSizeLimit                         = 0                    {product}
    uintx HeapSizePerGCThread                       = 87241520             {product}
    uintx InitialHeapSize                          := 536870912            {product}
    uintx LargePageHeapSizeThreshold                = 134217728            {product}
    uintx MaxHeapSize                              := 8589934592           {product}
java version "1.8.0_301"
Java(TM) SE Runtime Environment (build 1.8.0_301-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.301-b09, mixed mode)
```

当元空间的数据增长时，JVM 会请求操作系统分配更多的内存。如果内存空间足够，操作系统就会满足 JVM 的请求。那会不会出现元空间溢出的情况呢?

答案是肯定的，这个我们留到内存溢出的章节里来细讲。

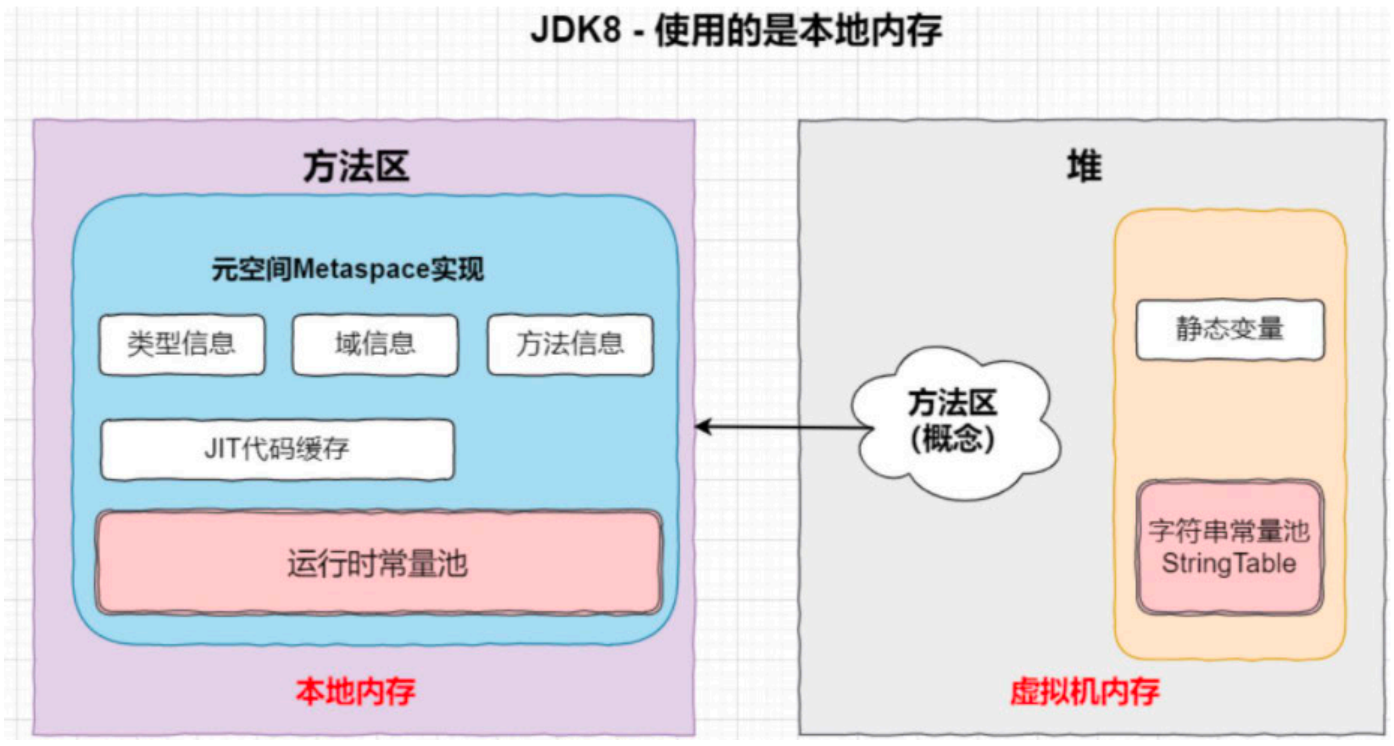
运行时常量池

在讲字节码的时候，我们详细的讲过常量池，它是字节码文件的资源仓库，先是一个常量池大小，从 1 到 n-1，0 为保留索引，然后是常量池项的集合，包括类信息、字段信息、方法信息、接口信息、字符串常量等。



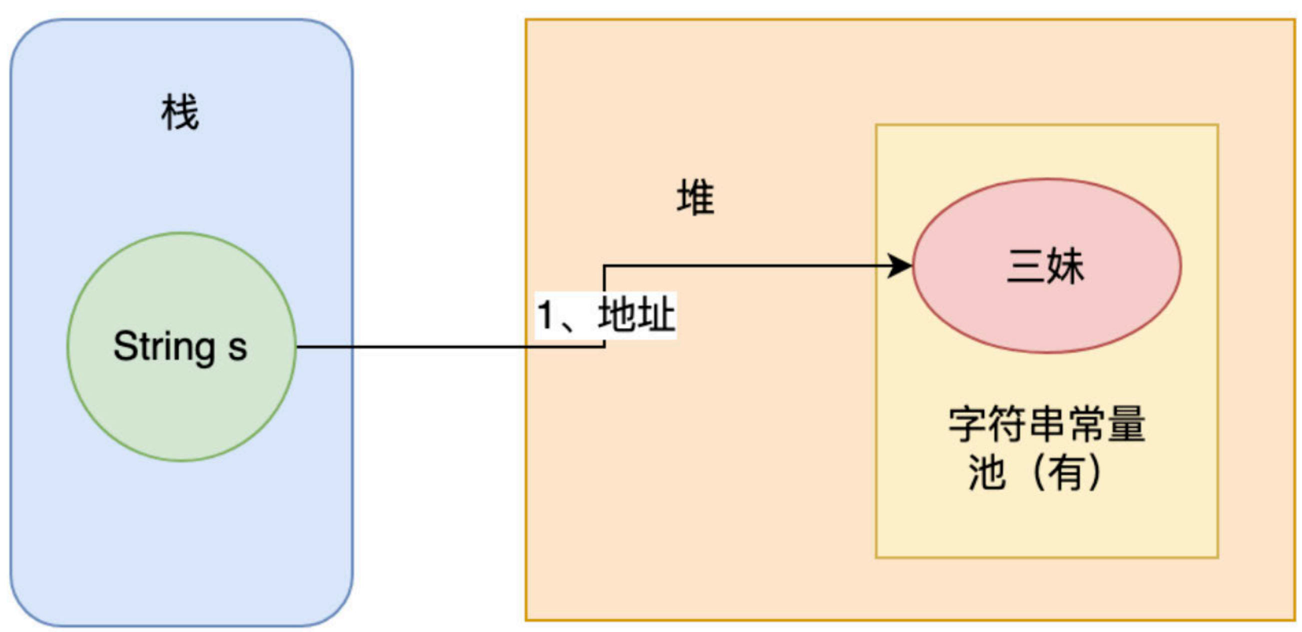
运行时常量池，顾名思义，就是在运行时期间，JVM 会将字节码文件中的常量池加载到内存中，存放在运行时常量池中。

也就是说，常量池是在字节码文件中，而运行时常量池在元空间当中（JDK 8 及以后），讲的是一个东西，但形态不一样，就好像一个是固态，一个是液态；或者一个是模子，一个是模子里的锅碗瓢盆。



字符串常量池

字符串常量池我们在讲字符串的时候已经详细讲过了，它的作用是存放字符串常量，也就是我们在代码中写的字符串。依然在堆中。



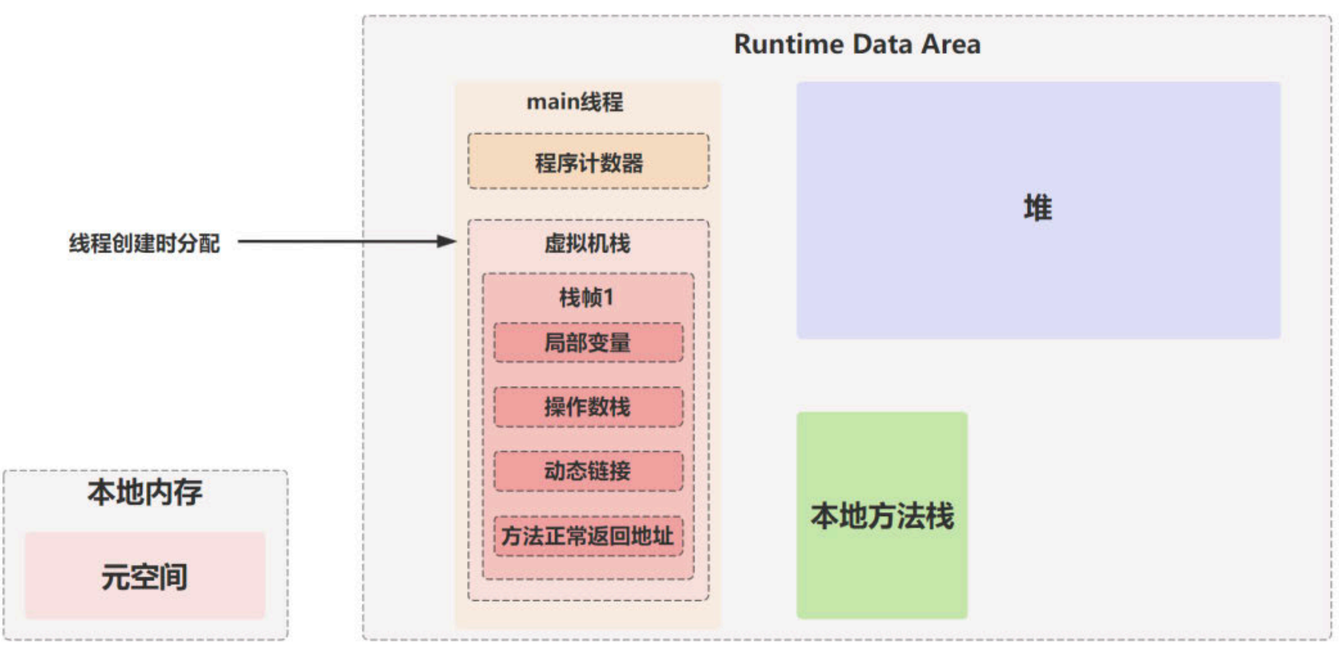
OK，方法区（不管是永久代还是元空间的实现）和堆一样，**是线程共享的区域。**

小结

来总结一下运行时数据区的主要组成：

- PC 寄存器（PC Register），也叫程序计数器（Program Counter Register），是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的信号指示器。
- JVM 栈（Java Virtual Machine Stack），与 PC 寄存器一样，JVM 栈也是线程私有的。每一个 JVM 线程都有自己的 JVM 栈（也叫方法栈），这个栈与线程同时创建，它的生命周期与线程相同。
- 本地方法栈（Native Method Stack），JVM 可能会使用到传统的栈来支持 Native 方法的执行，这个栈就是本地方法栈。
- 堆（Heap），在 JVM 中，堆是可供各条线程共享的运行时内存区域，也是供所有类实例和数据对象分配内存的区域。
- 方法区（Method area），JDK 8 开始，使用元空间取代了永久代。**方法区是 JVM 中的一个逻辑区域**，用于存储类的结构信息，包括类的定义、方法的定义、字段的定义以及字节码指令。不同的是，元空间不再是 JVM 内存的一部分，而是通过本地内存（Native Memory）来实现的。
- 运行时常量池，运行时常量池是每一个类或接口的常量在运行时的表现形式，它包括了编译器可知的数值字面量，以及运行期解析后才能获得的方法或字段的引用。简而言之，当一个方法或者变量被引用时，JVM 通过运行时常量区来查找方法或者变量在内存里的实际地址。

在 JVM 启动时，元空间的大小由 MaxMetaspaceSize 参数指定，JVM 在运行时会自动调整元空间的大小，以适应不同的程序需求。



第十节：深入理解垃圾回收机制

记得以前有这样一副动图，用来嘲笑 JVM 的垃圾回收机制，大致的意思就是，JVM 的垃圾回收机制很工业化，但是好像是在做无用功，垃圾回收不彻底（😂）。

C/C++ 虽然需要手动释放内存，但开发者信誓旦旦，认为自己一定能清理得很彻底。那这次，我们就从头到尾来详细地聊一聊 JVM 的垃圾回收机制，看看到底如何。

垃圾回收的概念

垃圾回收（Garbage Collection，GC），顾名思义就是释放垃圾占用的空间，防止内存爆掉。有效的使用可以使用的内存，对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收。

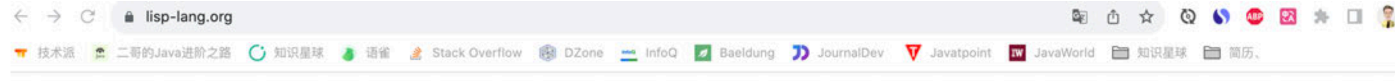
Java 语言出来之前，大家都在拼命的写 C 或者 C++ 的程序，而此时存在一个很大的矛盾，C++ 等语言创建对象要不断的去开辟空间，不用的时候又需要不断的去释放空间，既要写构造函数，又要写析构函数。

> 构造函数和 Java 中的构造方法类似，用来创建对象，析构函数和 Java 中的 finalize 方法有一点类似，可以在对象被垃圾回收器回收之前执行清理操作，但不推荐，因为 finalize 的执行时机并不确定。

于是，有人就提出，能不能写一段程序实现这块功能，每次创建对象、释放内存空间的时候复用这段代码?

牛人还是多啊，1960 年，基于 MIT 的 Lisp 首先提出了垃圾回收的概念，用于处理 C 语言等不停的析构操作，Java 的垃圾回收机制算是发扬光大了。

> Lisp 是一种函数式编程语言，我从官网上截幅图大家感受下。



Functional

Functions are first class objects: you can pass them around, store them, call them dynamically. Build your application by composing small, functional building blocks.

```
(reduce #'-
        (reverse (list 1 2 3)))
⇒ 0

(mapcar #'string-downcase
        (list "Hello" "world!"))
=> ("hello" "world!")
```

Object-Oriented

Build reusable and extensible class hierarchies using the Common Lisp Object System. Design patterns disappear as you adapt the language to your problem domain.

```
(defclass book ()
  ((title :reader book-title
          :initarg :title)
   (author :reader book-author
           :initarg :author))
  (:documentation "Describes a book."))

(make-instance 'book
               :title "ANSI Common Lisp"
               :author "Paul Graham")
```

垃圾判断算法

既然 JVM 要做垃圾回收，就要搞清楚什么是垃圾，什么不是垃圾。通常会有这么几种算法来确定一个对象是否是垃圾，这块也是面试当中常考的一个知识点，大家一定要掌握。

- 引用计数算法
- 可达性分析算法
- 分代收集算法
- 复制算法

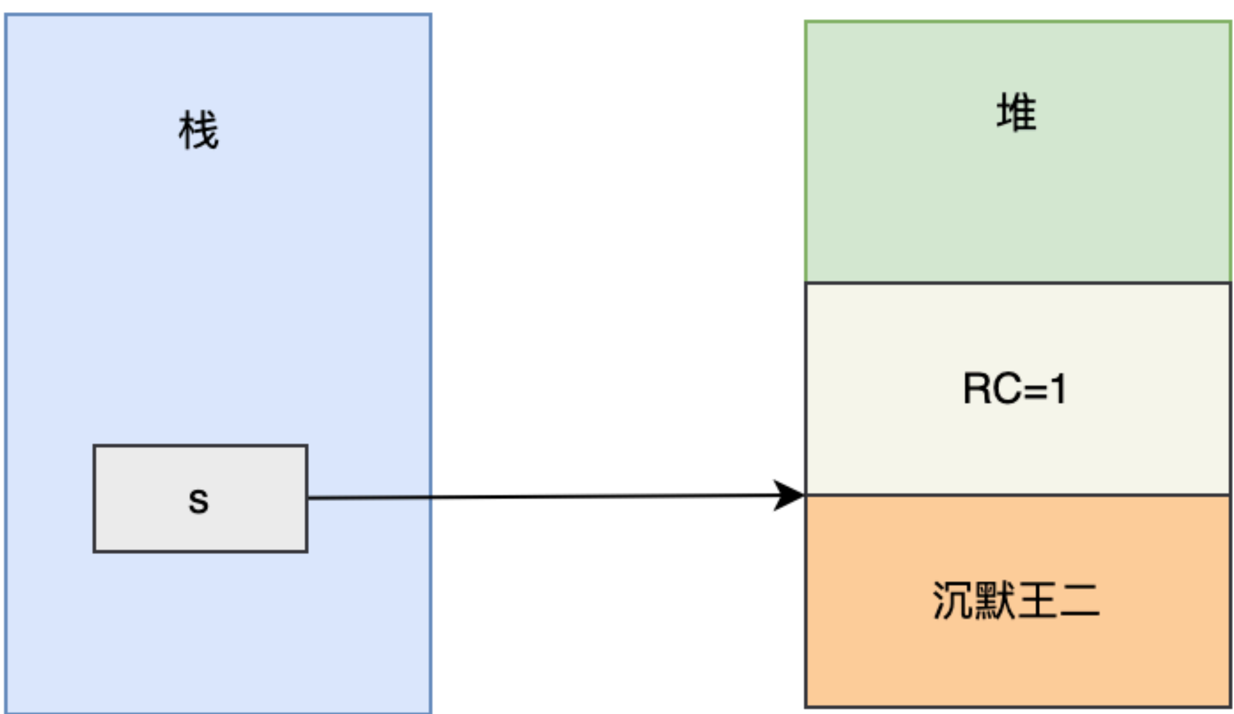
引用计数算法

引用计数算法（Reachability Counting）是通过在对象头中分配一个空间来保存该对象被引用的次数（Reference Count）。

如果该对象被其它对象引用，则它的引用计数加 1，如果删除对该对象的引用，那么它的引用计数就减 1，当该对象的引用计数为 0 时，那么该对象就会被回收。

```
String s = new String("沉默王二");
```

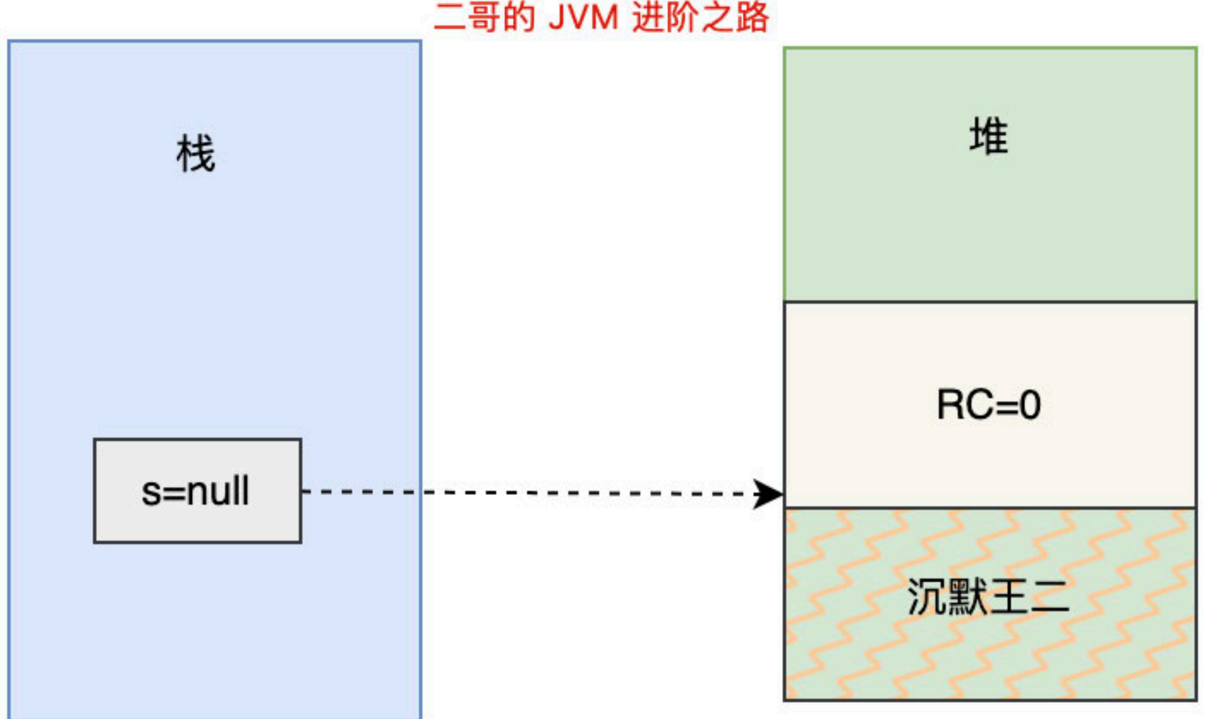
我们来创建一个字符串，这时候"沉默王二"有一个引用，就是 s。此时 Reference Count 为 1。



然后将 s 设置为 null。

```
s = null;
```

这时候"沉默王二"的引用次数就等于 0 了，在引用计数算法中，意味着这块内容就需要被回收了。



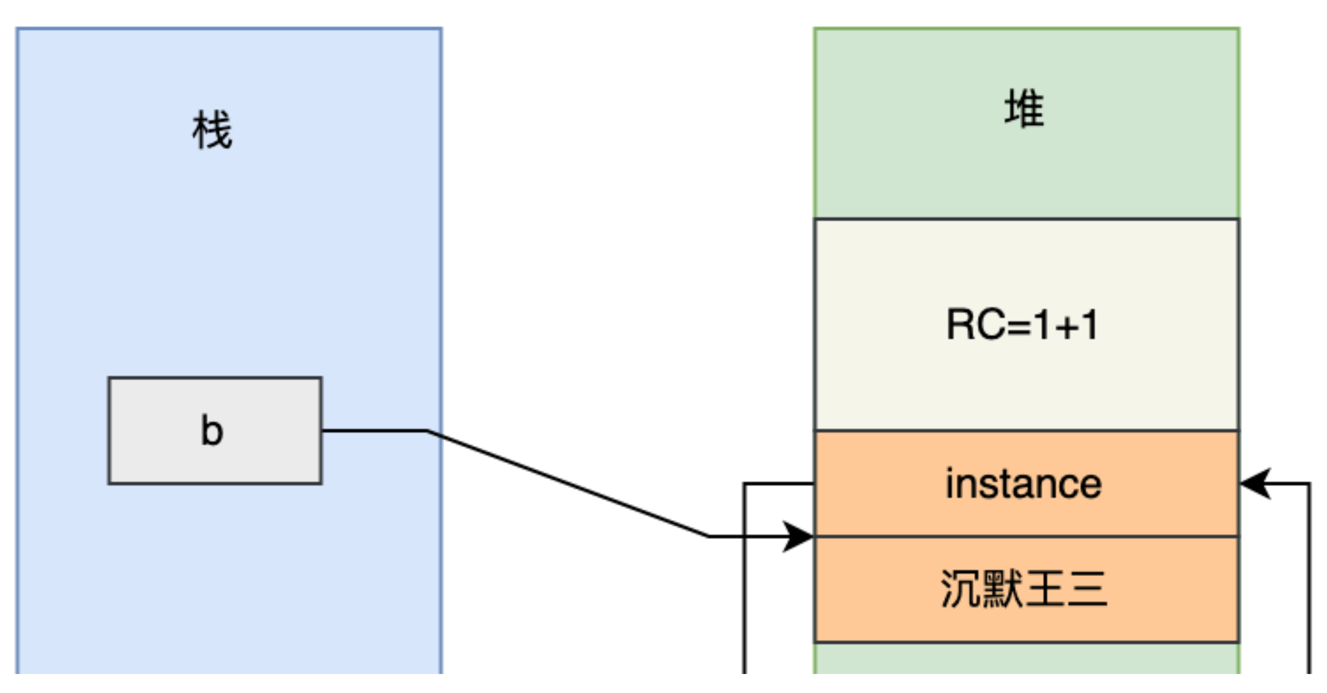
引用计数算法将垃圾回收分摊到整个应用程序的运行当中，而不是集中在垃圾收集时。因此，采用引用计数的垃圾收集不属于严格意义上的"Stop-The-World"的垃圾收集机制（随后我们会细讲）。

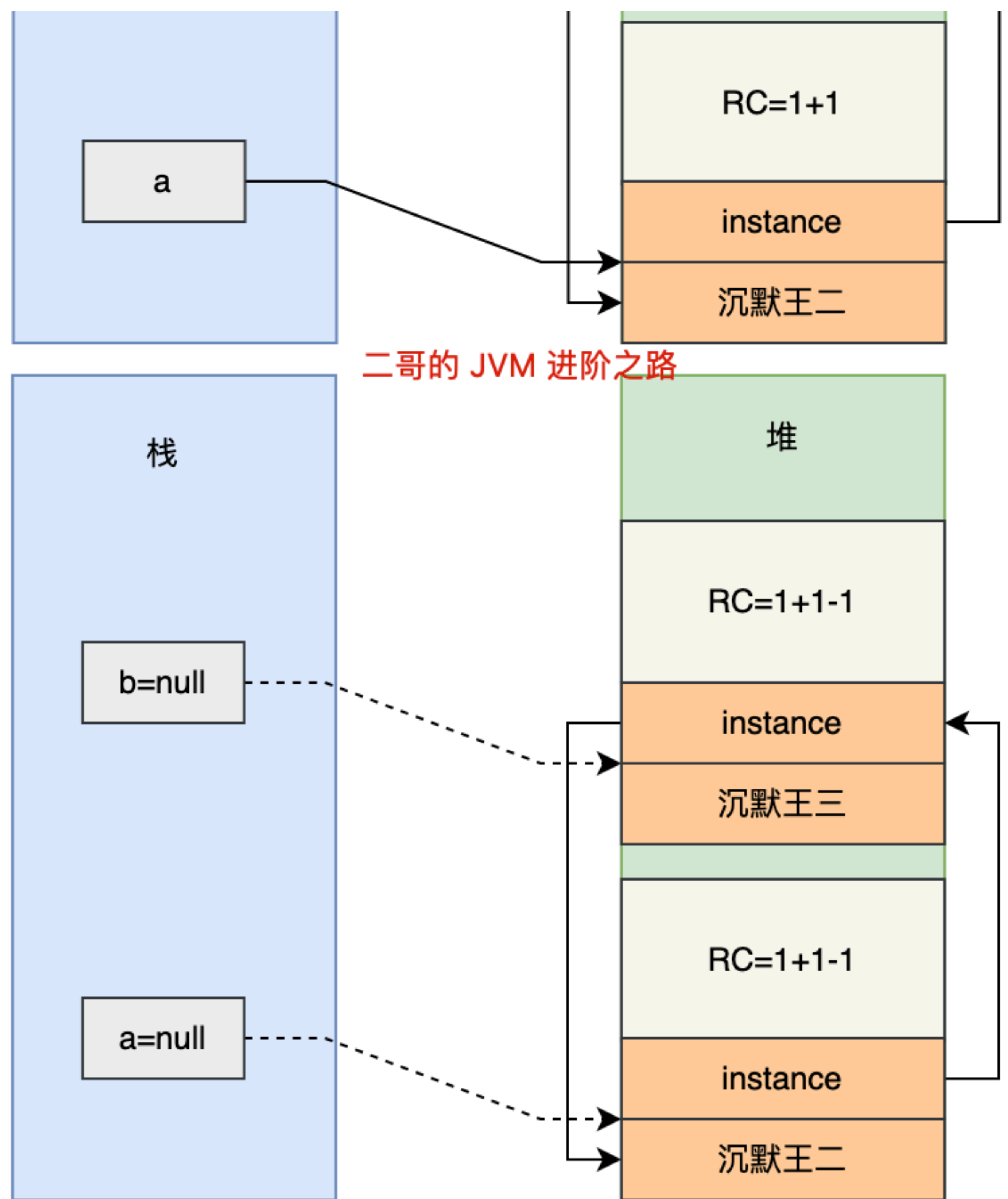
引用计数算法看似很美好，但实际上它存在一个很大的问题，那就是无法解决循环依赖的问题。来看下面的代码。

```
public class ReferenceCountingGC {

    public Object instance;  // 对象属性，用于存储对另一个 ReferenceCountingGC 对象的引用

    public ReferenceCountingGC(String name) {
        // 构造方法
    }

    public static void testGC() {
        // 创建两个 ReferenceCountingGC 对象
        ReferenceCountingGC a = new ReferenceCountingGC("沉默王二");
        ReferenceCountingGC b = new ReferenceCountingGC("沉默王三");

        // 使 a 和 b 相互引用
        a.instance = b;
        b.instance = a;

        // 将 a 和 b 设置为 null
        a = null;
        b = null;

        // 这个位置是垃圾回收的触发点
    }
}
```

代码中创建了两个 ReferenceCountingGC 对象 a 和 b。

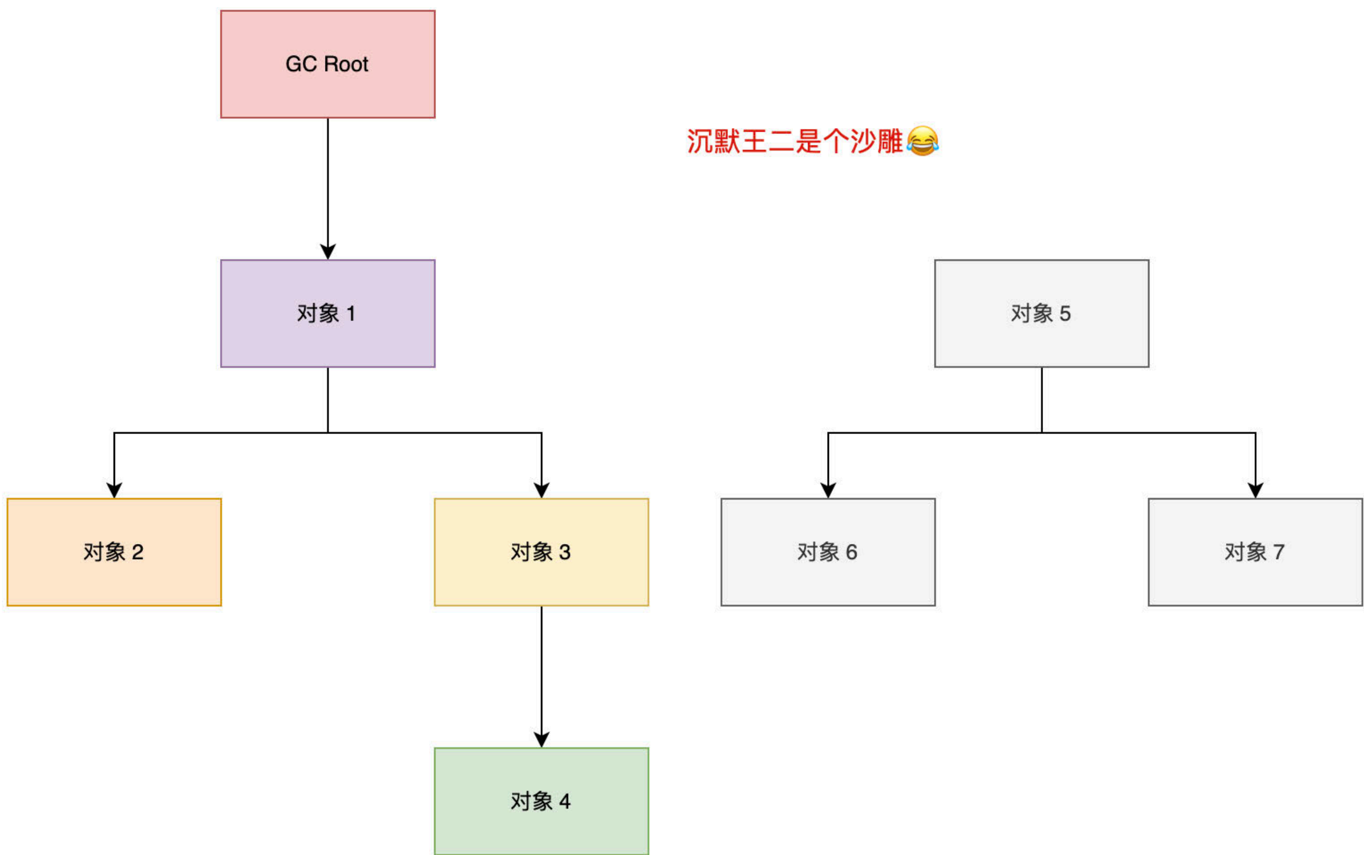
然后使它们相互引用。接着，将这两个对象的引用设置为 null，理论上它们会在接下来被垃圾回收器回收。但由于它们相互引用着对方，导致它们的引用计数永远都不会为 0，通过引用计数算法，也就永远无法通知 GC 收集器回收它们。





可达性分析算法

可达性分析算法（Reachability Analysis）的基本思路是，通过一些被称为引用链（GC Roots）的对象作为起点，然后向下搜索，搜索走过的路径被称为（Reference Chain），当一个对象到 GC Roots 之间没有任何引用相连时，即从 GC Roots 到该对象节点不可达，则证明该对象是需要垃圾收集的。

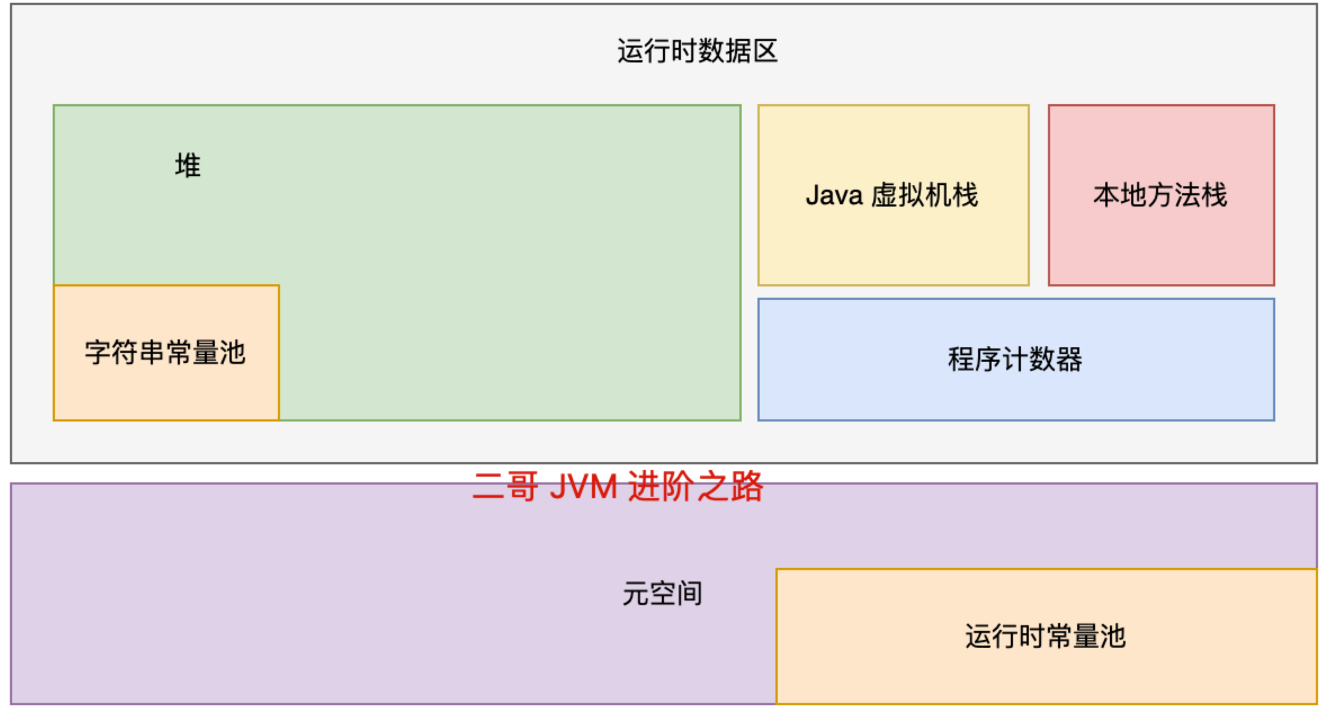


通过可达性算法，成功解决了引用计数无法解决的问题-“循环依赖”，只要你无法与 GC Root 建立直接或间接的连接，系统就会判定你为可回收对象。

在 Java 语言中，可作为 GC Root 的对象包括以下 4 种：

- 虚拟机栈中引用的对象
- 本地方法栈引用的对象
- 类静态变量引用的对象
- 常量引用的对象

大家可以回想一下我们前面讲过的JVM 运行时数据区。



1、虚拟机栈中引用的对象

来看下面这段代码：

```
public class StackLocalParameter {
    // 构造方法
    public StackLocalParameter(String name) {}

    public static void testGC() {
        // 创建一个 StackLocalParameter 对象，引用存储在栈上，对象在堆上
        StackLocalParameter s = new StackLocalParameter("localParameter");

        // 将引用 s 设置为 null，使 StackLocalParameter 对象成为垃圾回收的候选对象
        s = null;

        // 此时，垃圾回收器可以回收之前创建的 StackLocalParameter 对象，
        // 因为它不再有任何强引用指向它
    }
}
```

这个代码片段主要展示了 Java 中局部变量的生命周期和垃圾回收机制。当局部变量（如这里的 s）不再指向任何对象，或者变量本身离开了作用域，它指向的对象就可以被视为垃圾回收的候选对象。

> 局部变量的引用存储在虚拟机栈中，而对象存储在堆中。

2、本地方法栈中引用的对象

来看下面这段代码：

```
public class NativeExample {
    private native void nativeMethod(Object obj);

    public void exampleMethod() {
        Object myObject = new Object();

        // 调用本地方法，传递对象
        nativeMethod(myObject);

        // 即使在 Java 代码中不再使用 myObject，
        // 只要 nativeMethod 还持有它的引用，它就不会被垃圾回收。
    }
}
```

在这个示例中，nativeMethod 是一个本地方法，它从 Java 代码中接收一个对象引用。即使 Java 方法 exampleMethod 完成了对 myObject 的使用，只要本地方法 nativeMethod 还在执行并且持有对 myObject 的引用，myObject 就不会被回收。

把本地方法栈中引用的对象作为 GC Root，是 JVM 保证 Java 与本地代码交互时内存安全的一个关键机制。

3、类静态变量引用的对象

来看下面这段代码：

```
public class MethodAreaStaicProperties {
    // 静态变量的引用 m 存储在方法区中（JDK8 以后的元空间）
    public static MethodAreaStaicProperties m;

    // 构造方法
    public MethodAreaStaicProperties(String name) {}

    public static void testGC() {
        // 创建一个 MethodAreaStaicProperties 实例
        MethodAreaStaicProperties s = new MethodAreaStaicProperties("properties");

        // 通过实例 s 设置静态变量 m 的值
        // 此时，m 指向堆上的一个新 MethodAreaStaicProperties 实例
        s.m = new MethodAreaStaicProperties("parameter");

        // 将 s 置为 null
        // 此时，s 指向的对象可以被垃圾回收，但静态变量 m 指向的对象不会被回收
        s = null;

        // 虽然 s 被置为 null，但 m 作为静态变量，仍然引用着一个 MethodAreaStaicProperties 实例
        // 因此，这个由 m 引用的对象不会被垃圾回收
```

```
    }
}
```

在这个示例中，静态变量 m 指向堆上的一个 MethodAreaStaicProperties 实例。即使 s 被置为 null，但 m 仍然引用着一个对象实例，因此 m 引用的对象不会被垃圾回收。

> 静态变量的引用通常存储在元空间，而对象仍然存储在堆中。

3、常量引用的对象

来看这段代码：

```java
public class MethodAreaStaicProperties {
    // 常量 m 的引用存储在方法区（JDK8 以后元空间）
    public static final MethodAreaStaicProperties m = new
MethodAreaStaicProperties("final");

    // 构造方法
    public MethodAreaStaicProperties(String name) {}

    public static void testGC() {
        // 创建 MethodAreaStaicProperties 类的实例
        MethodAreaStaicProperties s = new
MethodAreaStaicProperties("staticProperties");

        // 将引用 s 设置为 null，这使得 s 指向的对象成为垃圾回收的候选对象
        s = null;

        // 常量 m 的对象不会被回收
    }
}
```

常量 m 应用的对象和 testGC 方法中的 s 其实不存在关系，所以局部变量 s 引用的对象回收和其没有任何关系。

Stop The World

"Stop The World"是 Java 垃圾收集中的一个重要概念。在垃圾收集过程中，JVM 会暂停所有的用户线程，这种暂停被称为"Stop The World"事件。

这么做的主要原因是为了防止在垃圾收集过程中，用户线程修改了堆中的对象，导致垃圾收集器无法准确地收集垃圾。

值得注意的是，"Stop The World"事件会对 Java 应用的性能产生影响。如果停顿时间过长，就会导致应用的响应时间变长，对于对实时性要求较高的应用，如交易系统、游戏服务器等，这种情况是不能接受的。

因此，在选择和调优垃圾收集器时，需要考虑其停顿时间。Java 中的一些垃圾收集器，如 G1 和 ZGC，都会尽可能地减少了"Stop The World"的时间，通过并发的垃圾收集，提高应用的响应性能。

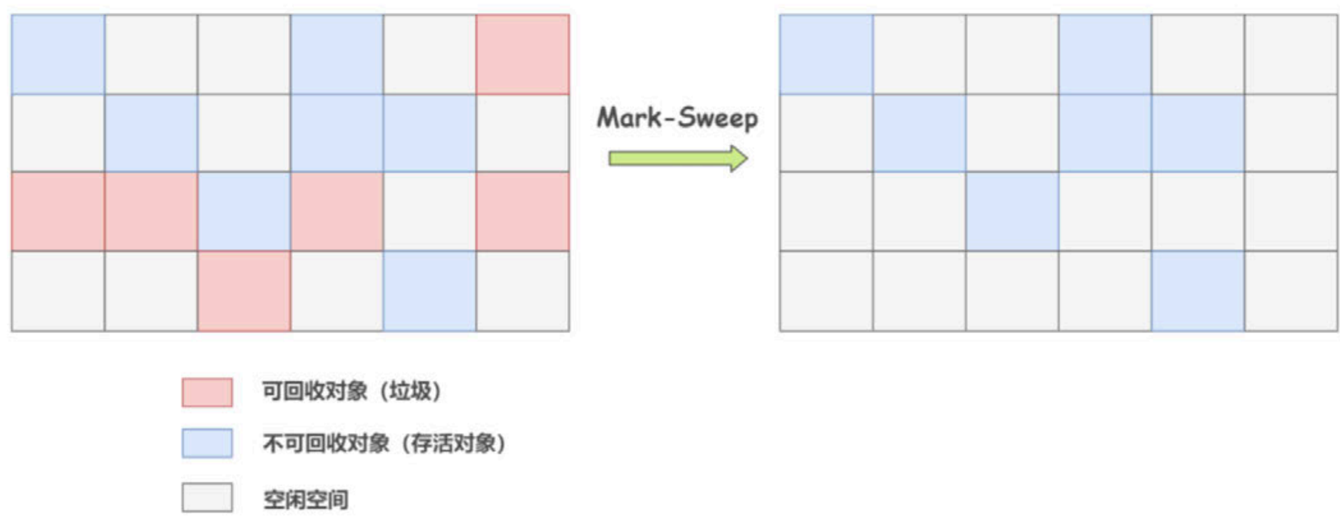
总的来说，"Stop The World"是 Java 垃圾收集中必须面对的一个挑战，其目标是在保证内存的有效利用和应用的响应性能之间找到一个平衡。

垃圾收集算法

在确定了哪些垃圾可以被回收后，垃圾收集器要做的事情就是进行垃圾回收，但是这里面涉及到一个问题是：**如何高效地进行垃圾回收**。由于 JVM 规范并没有对如何实现垃圾收集器做出明确的规定，因此各个厂商的虚拟机可以采用不同的方式来实现垃圾收集器，这里我们讨论几种常见的垃圾收集算法。

标记清除算法

标记清除算法（Mark-Sweep）是最基础的一种垃圾回收算法，它分为 2 部分，先把内存区域中的这些对象进行标记，哪些属于可回收的标记出来（用前面提到的可达性分析法），然后把这些垃圾拎出来清理掉。

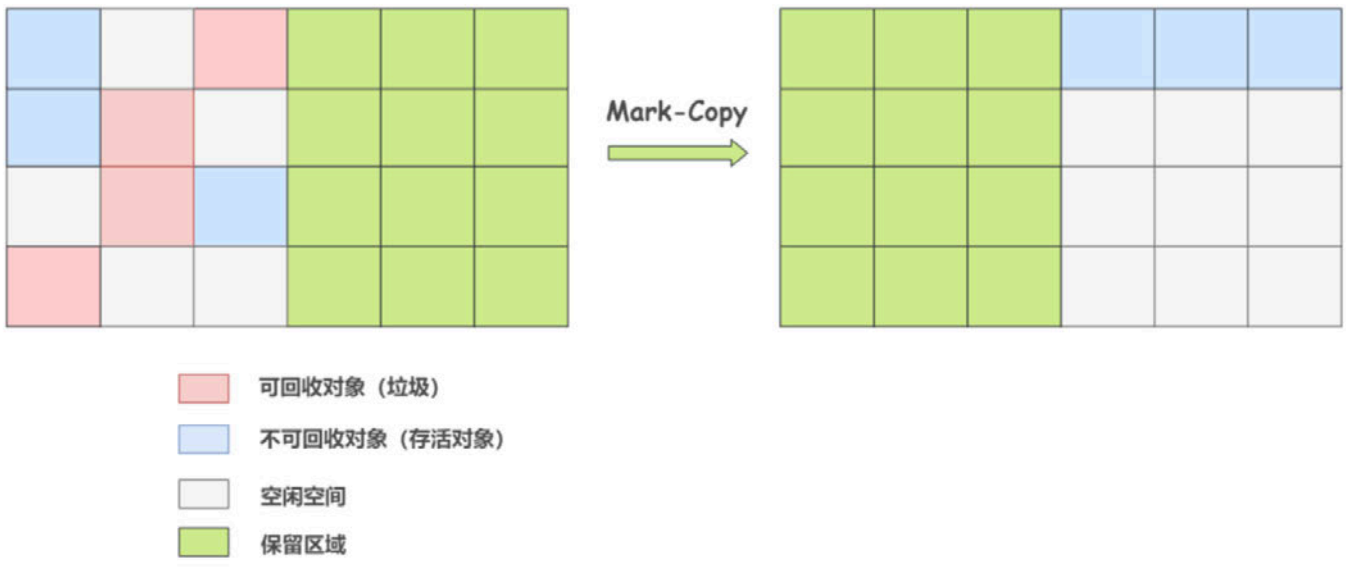


就像上图一样，清理掉的垃圾就变成可使用的空闲空间，等待被再次使用。逻辑清晰，并且也很好操作，但它存在一个很大的问题，那就是内存碎片。碎片太多可能会导致当程序运行过程中需要分配较大对象时，因无法找到足够的连续内存而不得不提前触发新一轮的垃圾收集。

复制算法

复制算法（Copying）是在标记清除算法上演化而来的，用于解决标记清除算法的内存碎片问题。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。

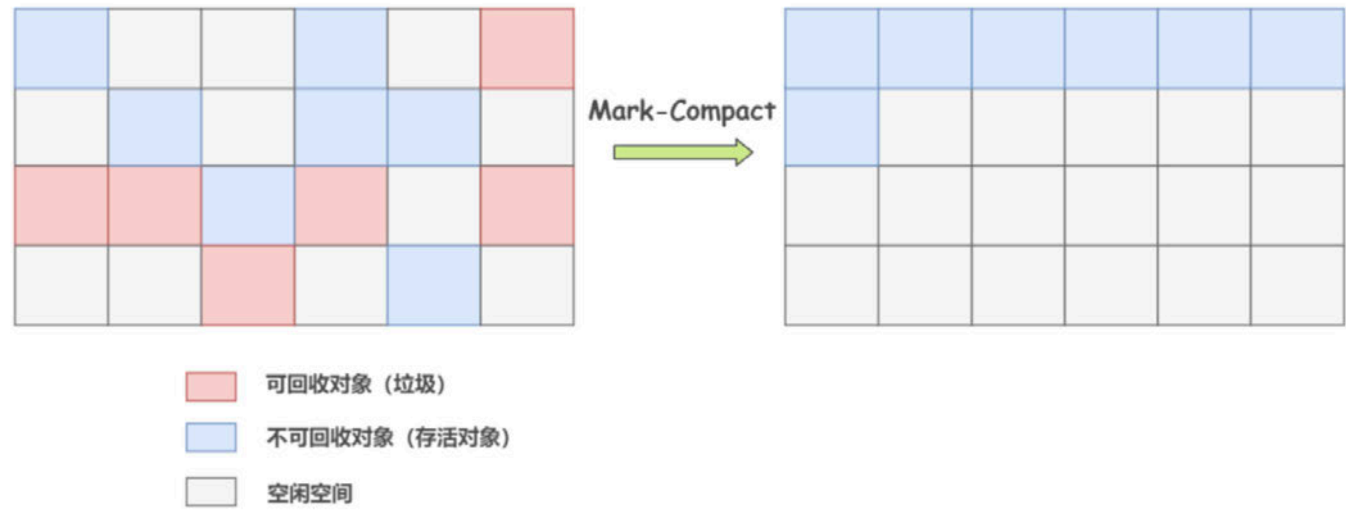
当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样就保证了内存的连续性，逻辑清晰，运行高效。



但复制算法也存在一个很明显的问题，合着我这 190 平的大四室，只能当 90 平米的小两室来居住？代价实在太高。

标记整理算法

标记整理算法（Mark-Compact），标记过程仍然与标记清除算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，再清理掉端边界以外的内存区域。

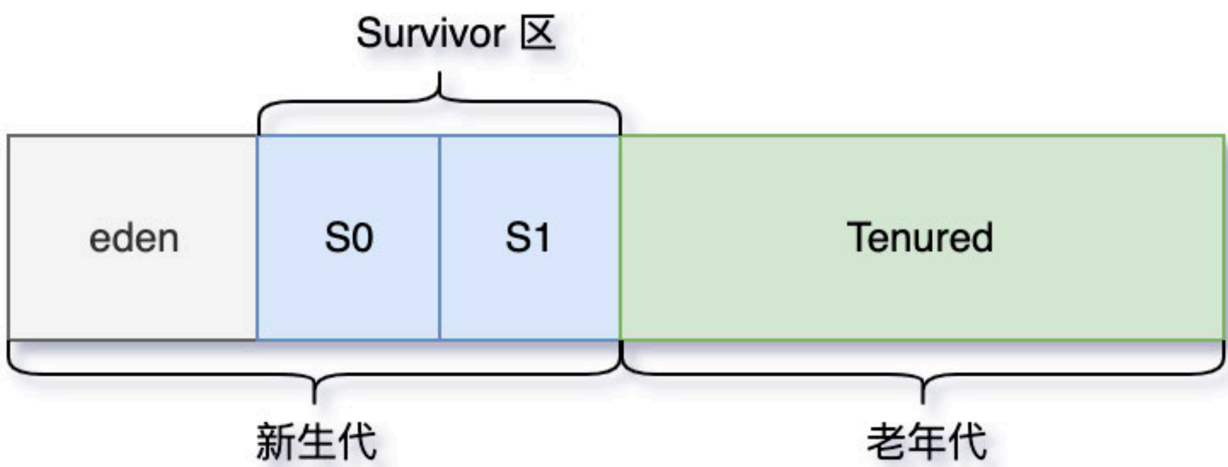


标记整理算法一方面在标记-清除算法上做了升级，解决了内存碎片的问题，也规避了复制算法只能利用一半内存区域的弊端。看起来很美好，但内存变动更频繁，需要整理所有存活对象的引用地址，在效率上比复制算法差很多。

分代收集算法

分代收集算法（Generational Collection）严格来说并不是一种思想或理论，而是融合上述 3 种基础的算法思想，而产生的针对不同情况所采用不同算法的一套组合拳。

对象存活周期的不同将内存划分为几块，一般是把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。



在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。

老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记清理或者标记整理算法来进行回收。

新生代和老年代

堆（Heap）是 JVM 中最大的一块内存区域，也是垃圾收集器管理的主要区域。

新生代
老年代
Eden
From Survivor
To Survicor
二哥的 JVM 进阶之路
Old

堆主要分为 2 个区域，年轻代与老年代，其中年轻代又分 Eden 区和 Survivor 区，其中 Survivor 区又分 From 和 To 两个区。

Eden 区

据 IBM 公司之前的研究表明，有将近 98% 的对象是朝生夕死，所以针对这一现状，大多数情况下，对象会在新生代 Eden 区中进行分配，当 Eden 区没有足够空间进行分配时，JVM 会发起一次 Minor GC，Minor GC 相比 Major GC 更频繁，回收速度也更快。

通过 Minor GC 之后，Eden 区中绝大部分对象会被回收，而那些无需回收的存活对象，将会进到 Survivor 的 From 区，如果 From 区不够，则直接进入 To 区。

Survivor 区

Survivor 区相当于是 Eden 区和 Old 区的一个缓冲，类似于我们交通灯中的黄灯。

1、为啥需要 Survivor 区?

不就是新生代到老年代吗，直接 Eden 到 Old 不好了吗，为啥要这么复杂。

如果没有 Survivor 区，Eden 区每进行一次 Minor GC，存活的对象就会被送到老年代，老年代很快就会被填满。而有很多对象虽然一次 Minor GC 没有消灭，但其实也并不会蹦跶多久，或许第二次，第三次就需要被清除。

这时候移入老年区，很明显不是一个明智的决定。

所以，Survivor 的存在意义就是减少被送到老年代的对象，进而减少 Major GC 的发生。Survivor 的预筛选保证，只有经历 16 次 Minor GC 还能在新生代中存活的对象，才会被送到老年代。

2、Survivor 区为啥划分为两块?

设置两个 Survivor 区最大的好处就是解决内存碎片化，我们先假设一下，Survivor 只有一个区域会怎样。

Minor GC 执行后，Eden 区被清空，存活的对象放到了 Survivor 区，而之前 Survivor 区中的对象，可能也有一些是需要被清除的。那么问题来了，这时候我们怎么清除它们?

在这种场景下，我们只能标记清除，而我们知道标记清除最大的问题就是内存碎片，在新生代这种经常会消亡的区域，采用标记清除必然会让内存产生严重的碎片化。

但因为 Survivor 有 2 个区域，所以每次 Minor GC，会将之前 Eden 区和 From 区中的存活对象复制到 To 区域。第二次 Minor GC 时，From 与 To 职责兑换，这时候会将 Eden 区和 To 区中的存活对象再复制到 From 区域，以此反复。

这种机制最大的好处就是，整个过程中，永远有一个 Survivor space 是空的，另一个非空的 Survivor space 是无碎片的。

那么，Survivor 为什么不分更多块呢? 比方说分成三个、四个、五个?

显然，如果 Survivor 区再细分下去，每一块的空间就会比较小，容易导致 Survivor 区满，两块 Survivor 区可能是经过权衡之后的最佳方案。

Old 区

老年代占据着 2/3 的堆内存空间，只有在 Major GC 的时候才会进行清理，每次 GC 都会触发“Stop-The-World”。内存越大，STW 的时间也越长，所以内存也不仅仅是越大就越好。

由于复制算法在对象存活率较高的老年代会进行很多次的复制操作，效率很低，所以老年代这里采用的是标记整理算法。

除了上述所说，在内存担保机制下，无法安置的对象会直接进到老年代，以下几种情况也会进入老年代。

1、大对象

大对象指需要大量连续内存空间的对象，这部分对象不管是不是“朝生夕死”，都会直接进到老年代。这样做主要是为了避免在 Eden 区及 2 个 Survivor 区之间发生大量的内存复制。当你的系统有非常多“朝生夕死”的大对象时，得注意了。

2、长期存活对象

虚拟机给每个对象定义了一个对象年龄（Age）计数器。正常情况下对象会不断的在 Survivor 的 From 区与 To 区之间移动，对象在 Survivor 区中每经历一次 Minor GC，年龄就增加 1 岁。当年龄增加到 15 岁时，这时候就会被转移到老年代。当然，这里的 15，JVM 也支持进行特殊设置 `-XX:MaxTenuringThreshold=10`。

可通过 `java -XX:+PrintFlagsFinal -version | grep MaxTenuringThreshold` 查看默认的阈值。

```
maweiqing@itwanger-2 paicoding-web % java -XX:+PrintFlagsFinal -version | grep MaxTenuringThreshold

    uintx MaxTenuringThreshold                      = 15                                  {product}
java version "1.8.0_301"
Java(TM) SE Runtime Environment (build 1.8.0_301-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.301-b09, mixed mode)
```

3、动态对象年龄

JVM 并不强制要求对象年龄必须到 15 岁才会放入老年区，如果 Survivor 空间中某个年龄段的对象总大小超过了 Survivor 空间的一半，那么该年龄段及以上年龄段的所有对象都会在下一次垃圾回收时被晋升到老年代，无需等你“成年”。

有点类似于负载均衡，轮询是负载均衡的一种，保证每台机器都分得同样的请求。看似很均衡，但每台机器的硬件不同，健康状况不同，所以我们可以基于每台机器接收的请求数、响应时间等，来调整负载均衡算法。

这种动态调整机制有助于优化内存使用和减少垃圾收集的频率，特别是在处理大量短生命周期对象的应用程序时。

小结

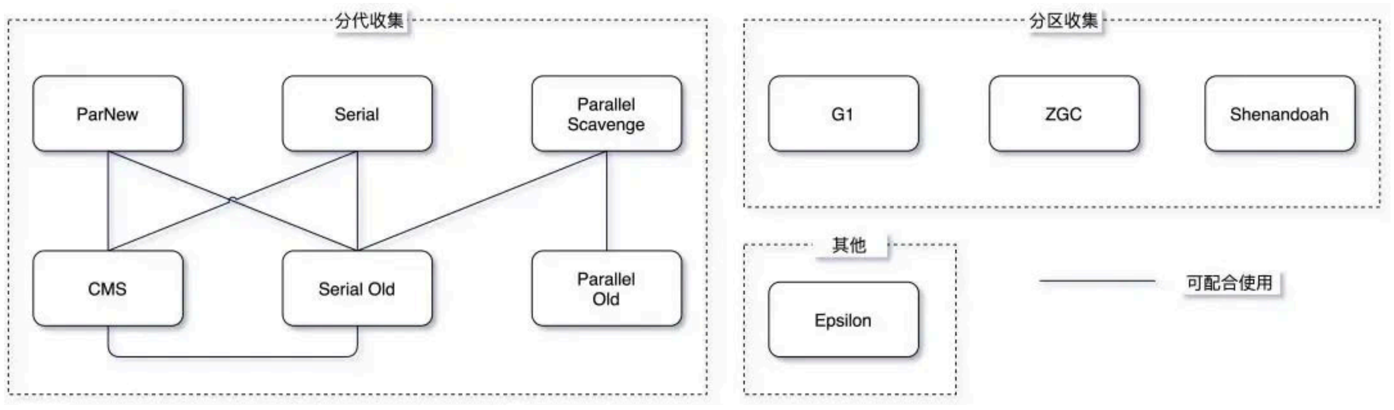
本篇内容我们从头到尾讲了一遍 JVM 的垃圾回收机制，包括垃圾回收的概念、垃圾判断算法、垃圾收集算法、Stop The World、新生代和老年代等等。

> - 参考链接 1：从头到尾再讲一次 Java 的垃圾回收
> - 参考链接 2：详解Java的垃圾回收机制
> - 参考链接 3：三大垃圾收集算法

第十一节：垃圾收集器

垃圾回收对于 Java 党来说，是一个绕不开的话题，工作中涉及到的调优工作也经常围绕着垃圾回收器展开。面对不同的业务场景，往往需要不同的垃圾收集器才能保证 GC 性能，因此，对于面大厂或者有远大志向的球友可以卷一下垃圾收集器。

就目前来说，JVM 的垃圾收集器主要分为两大类：**分代收集器**和**分区收集器**，分代收集器的代表是 CMS，分区收集器的代表是 G1 和 ZGC，下面我们来看看这两大类的垃圾收集器。



分代收集器

CMS

以获取最短回收停顿时间为目标，采用“标记-清除”算法，分 4 大步进行垃圾收集，其中初始标记和重新标记会 STW，JDK 1.5 时引入，JDK9 被标记弃用，JDK14 被移除，详情可见 JEP 363。

CMS（Concurrent Mark Sweep）垃圾收集器是第一个关注 GC 停顿时间（STW 的时间）的垃圾收集器。之前的垃圾收集器，要么是串行的垃圾回收方式，要么只关注系统吞吐量。

CMS 垃圾收集器之所以能够实现对 GC 停顿时间的控制，其本质来源于对「可达性分析算法」的改进，即三色标记算法。在 CMS 出现之前，无论是 Serious 垃圾收集器，还是 ParNew 垃圾收集器，以及 Parallel Scavenge 垃圾收集器，它们在进行垃圾回收的时候都需要 Stop the World，无法实现垃圾回收线程与用户线程的并发执行。

> 标记-清除算法、Stop the World、可达性分析算法等知识我们上一节也讲过了，忘记的球友可以回顾一下。

CMS 垃圾收集器通过三色标记算法，实现了垃圾回收线程与用户线程的并发执行，从而极大地降低了系统响应时间，提高了强交互应用程序的体验。它的运行过程分为 4 个步骤，包括：

- 初始标记
- 并发标记
- 重新标记
- 并发清除

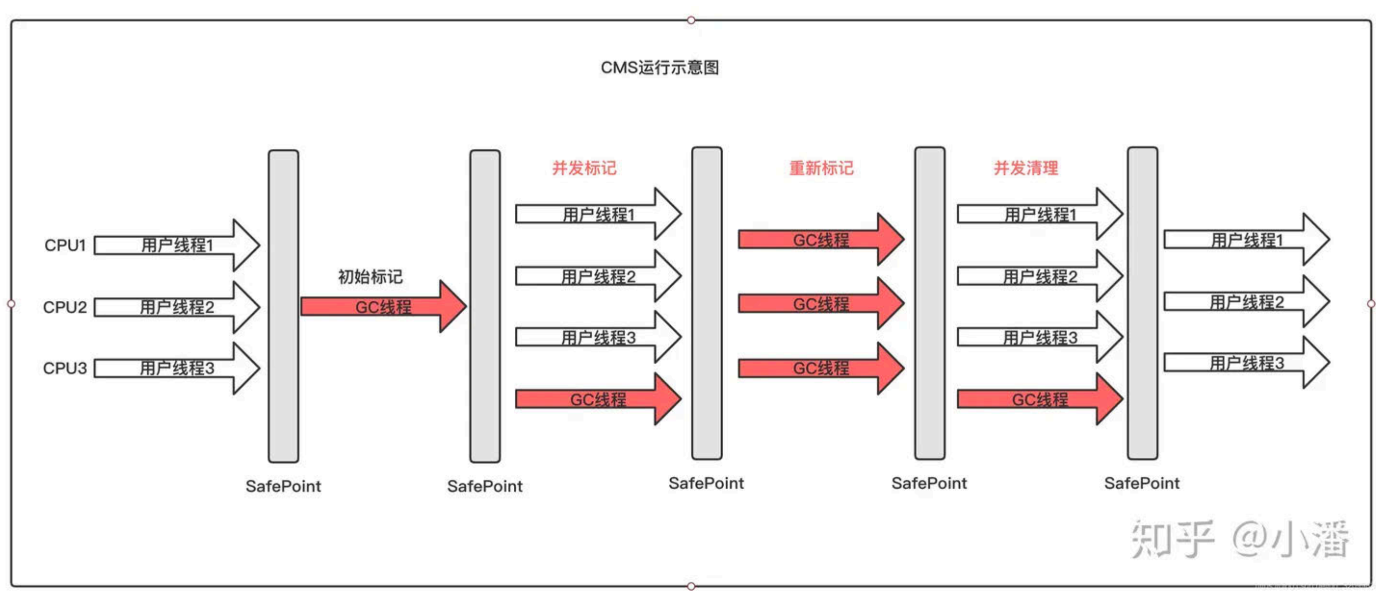
初始标记，指的是寻找所有被 GCRoots 引用的对象，该阶段需要「Stop the World」。这个步骤仅仅只是标记一下 GC Roots 能直接关联到的对象，并不需要做整个引用的扫描，因此速度很快。

并发标记，指的是对「初始标记阶段」标记的对象进行整个引用链的扫描，该阶段不需要「Stop the World」。 对整个引用链做扫描需要花费非常多的时间，因此通过垃圾回收线程与用户线程并发执行，可以降低垃圾回收的时间。

这也是 CMS 能极大降低 GC 停顿时间的核心原因，但这也带来了一些问题，即：并发标记的时候，引用可能发生变化，因此可能发生漏标（本应该回收的垃圾没有被回收）和多标（本不应该回收的垃圾被回收）了。

重新标记，指的是对「并发标记」阶段出现的问题进行校正，该阶段需要「Stop the World」。正如并发标记阶段说到的，由于垃圾回收算法和用户线程并发执行，虽然能降低响应时间，但是会发生漏标和多标的问题。所以对于 CMS 来说，它需要在这个阶段做一些校验，解决并发标记阶段发生的问题。

并发清除，指的是将标记为垃圾的对象进行清除，该阶段不需要「Stop the World」。 在这个阶段，垃圾回收线程与用户线程可以并发执行，因此并不影响用户的响应时间。



CMS 的优点是：并发收集、低停顿。但缺点也很明显：

①、对 CPU 资源非常敏感，因此在 CPU 资源紧张的情况下，CMS 的性能会大打折扣。

默认情况下，CMS 启用的垃圾回收线程数是`（CPU数量 + 3）/4`，当 CPU 数量很大时，启用的垃圾回收线程数占比就越小。但如果 CPU 数量很小，例如只有 2 个 CPU，垃圾回收线程占用就达到了 50%，这极大地降低系统的吞吐量，无法接受。

②、CMS 采用的是「标记-清除」算法，会产生大量的内存碎片，导致空间不连续，当出现大对象无法找到连续的内存空间时，就会触发一次 Full GC，这会导致系统的停顿时间变长。

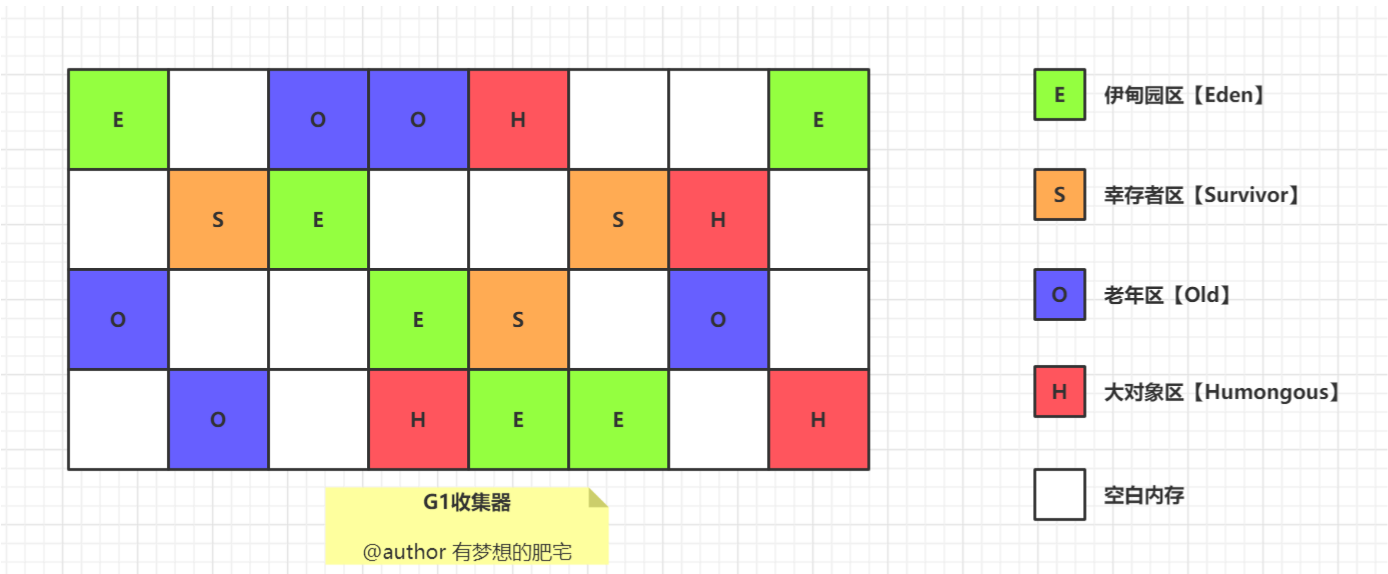
③、CMS 无法处理浮动垃圾，当 CMS 在进行垃圾回收的时候，应用程序还在不断地产生垃圾，这些垃圾会在 CMS 垃圾回收结束之后产生，这些垃圾就是浮动垃圾，CMS 无法处理这些浮动垃圾，只能在下一次 GC 时清理掉。

分区收集器

G1

G1（Garbage-First Garbage Collector）在 JDK 1.7 时引入，在 JDK 9 时取代 CMS 成为了默认的垃圾收集器。G1 有五个属性：分代、增量、并行、标记整理、STW。

①、分代：相信大家还记得我们上一讲中的年轻代和老年代，G1 也是基于这个思想进行设计的。它将堆内存分为多个大小相等的区域（Region），每个区域都可以是 Eden 区、Survivor 区或者 Old 区。



可以通过 `-XX:G1HeapRegionSize=n` 来设置 Region 的大小，可以设定为 1M、2M、4M、8M、16M、32M（不能超过）。

G1 有专门分配大对象的 Region 叫 Humongous 区，而不是让大对象直接进入老年代的 Region 中。在 G1 中，大对象的判定规则就是一个大对象超过了一个 Region 大小的 50%，比如每个 Region 是 2M，只要一个对象超过了 1M，就会被放入 Humongous 中，而且一个大对象如果太大，可能会横跨多个 Region 来存放。

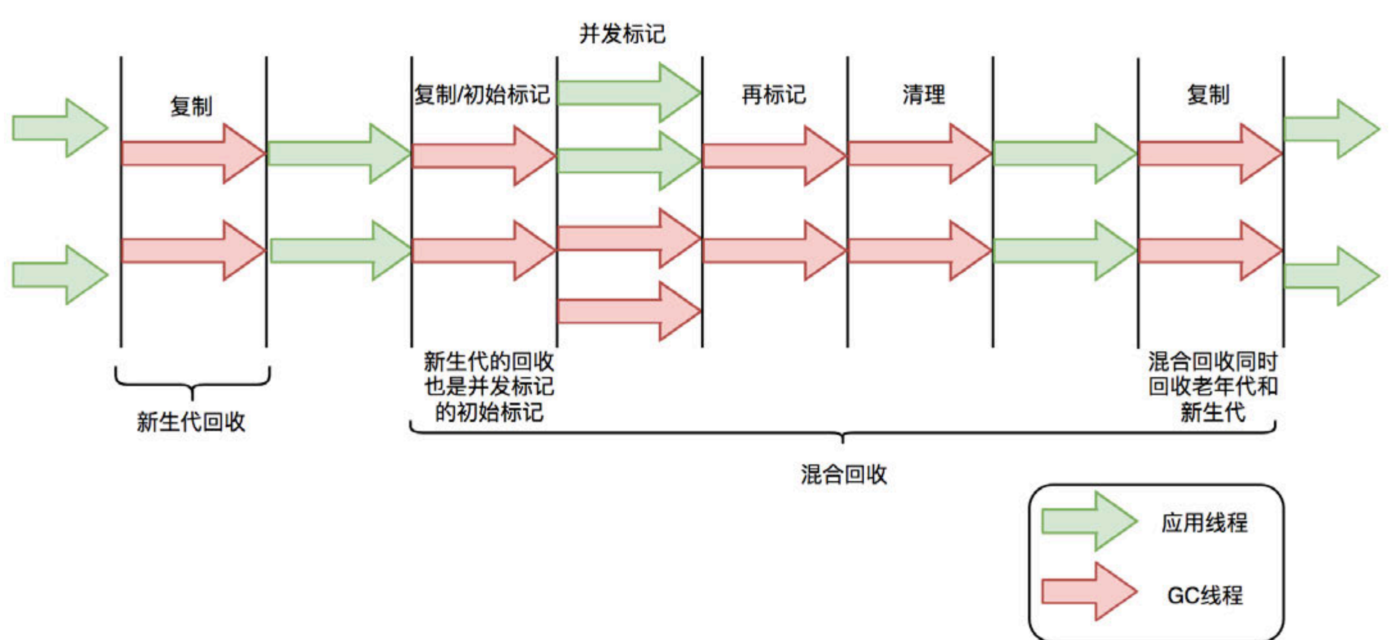
G1 会根据各个区域的垃圾回收情况来决定下一次垃圾回收的区域，这样就避免了对整个堆内存进行垃圾回收，从而降低了垃圾回收的时间。

②、增量：G1 可以以增量方式执行垃圾回收，这意味着它不需要一次性回收整个堆空间，而是可以逐步、增量地清理。有助于控制停顿时间，尤其是在处理大型堆时。

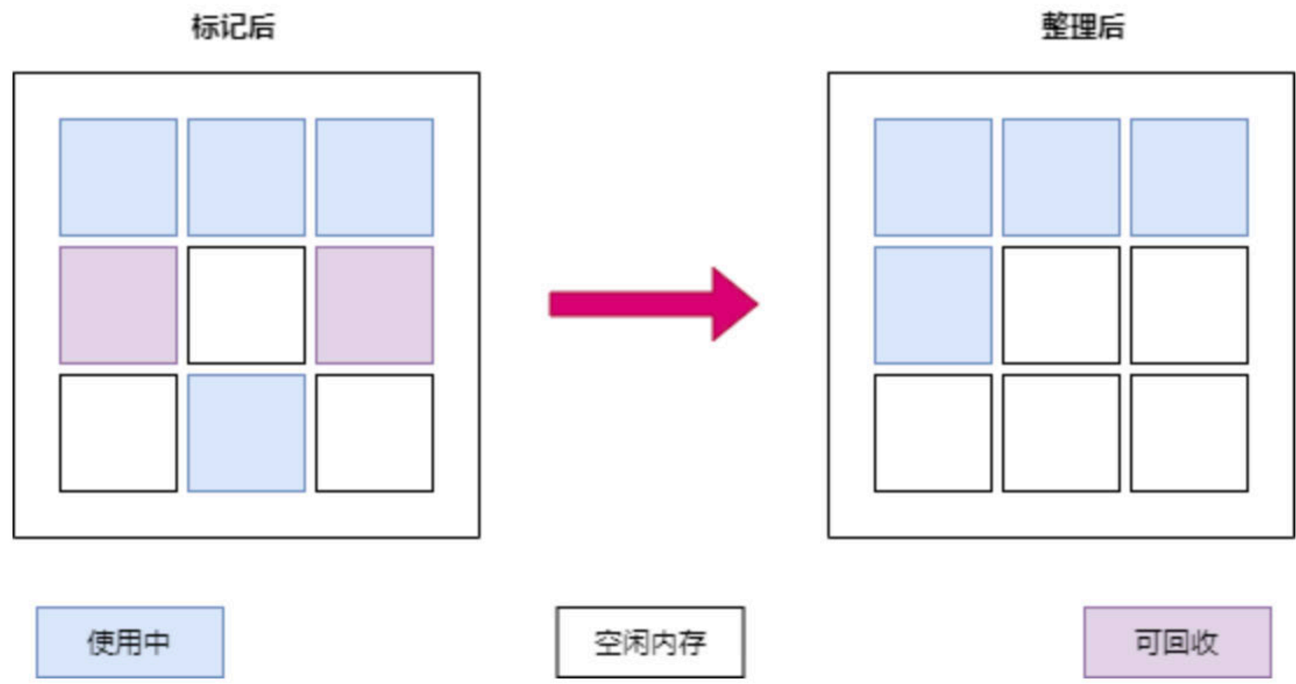
③、并行：G1 垃圾回收器可以并行回收垃圾，这意味着它可以利用多个 CPU 来加速垃圾回收的速度，这一特性在年轻代的垃圾回收（Minor GC）中特别明显，因为年轻代的回收通常涉及较多的对象和较高的回收速率。

④、标记整理：在进行老年代的垃圾回收时，G1 使用标记-整理算法。这个过程分为两个阶段：标记存活的对象和整理（压缩）堆空间。通过整理，G1 能够避免内存碎片化，提高内存利用率。

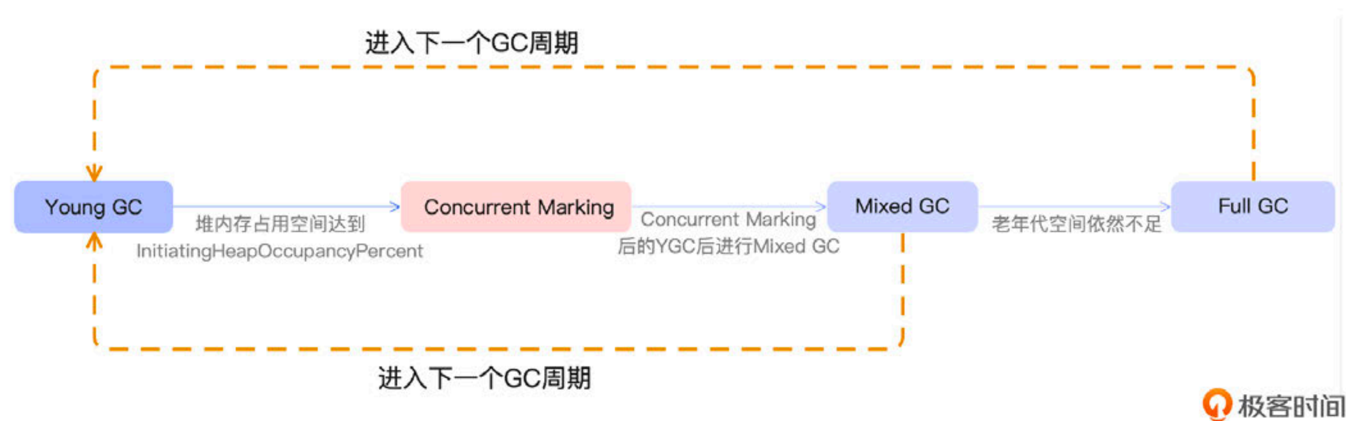
年轻代的垃圾回收（Minor GC）使用复制算法，因为年轻代的对象通常是朝生夕死的。



⑤、STW：G1 也是基于「标记-清除」算法，因此在进行垃圾回收的时候，仍然需要「Stop the World」。不过，G1 在停顿时间上添加了预测机制，用户可以指定期望停顿时间。



G1 中存在三种 GC 模式，分别是 Young GC、Mixed GC 和 Full GC。



当 Eden 区的内存空间无法支持新对象的内存分配时，G1 会触发 Young GC。

当需要分配对象到 Humongous 区域或者堆内存的空间占比超过 `-XX:G1HeapWastePercent` 设置的 InitiatingHeapOccupancyPercent 值时，G1 会触发一次 concurrent marking，它的作用就是计算老年代中有多少空间需要被回收，当发现垃圾的占比达到 `-XX:G1HeapWastePercent` 中所设置的 G1HeapWastePercent 比例时，在下次 Young GC 后会触发一次 Mixed GC。

Mixed GC 是指回收年轻代的 Region 以及一部分老年代中的 Region。Mixed GC 和 Young GC 一样，采用的也是复制算法。

在 Mixed GC 过程中，如果发现老年代空间还是不足，此时如果 G1HeapWastePercent 设定过低，可能引发 Full GC。`-XX:G1HeapWastePercent` 默认是 5，意味着只有 5% 的堆是“浪费”的。如果浪费的堆的百分比大于 G1HeapWastePercent，则运行 Full GC。

在以 Region 为最小管理单元以及所采用的 GC 模式的基础上，G1 建立了停顿预测模型，即 Pause Prediction Model 。这也是 G1 非常被人所称道的特性。

我们可以借助 `-XX:MaxGCPauseMillis` 来设置期望的停顿时间（默认 200ms），G1 会根据这个值来计算出一个合理的 Young GC 的回收时间，然后根据这个时间来制定 Young GC 的回收计划。

ZGC

ZGC（The Z Garbage Collector）是 JDK11 推出的一款低延迟垃圾收集器，适用于大内存低延迟服务的内存管理和回收，SPEC jbb 2015 基准测试，在 128G 的大堆下，最大停顿时间才 1.68 ms，停顿时间远胜于 G1 和 CMS。

ZGC 的设计目标是：在不超过 10ms 的停顿时间下，支持 TB 级的内存容量和几乎所有的 GC 功能，这也是 ZGC 名字的由来，Z 代表着 Zettabyte，也就是 1024EB，也就是 1TB 的 1024 倍。

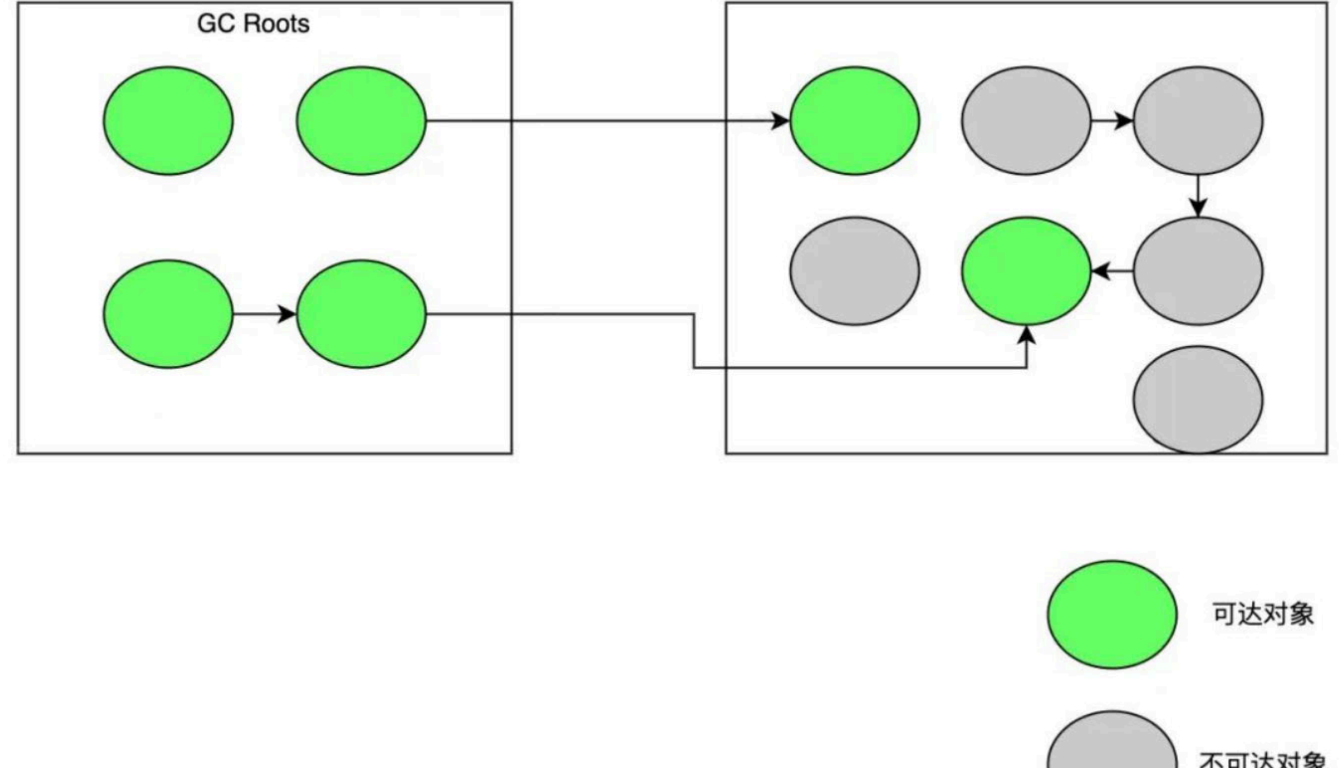
不过，我需要告诉大家的是，上面这段是我胡编的（😂），JDK 官方并没有明确给出 Z 的定义，就像小米汽车 su7，7 也是个魔数，没有明确的定义。

总之就是，ZGC 很牛逼，它的目标是：

- 停顿时间不超过 10ms；
- 停顿时间不会随着堆的大小，或者活跃对象的大小而增加；
- 支持 8MB~4TB 级别的堆，未来支持 16TB。

前面讲 G1 垃圾收集器的时候提到过，Young GC 和 Mixed GC 均采用的是复制算法，复制算法主要包括以下 3 个阶段：

①、标记阶段，从 GC Roots 开始，分析对象可达性，标记出活跃对象。

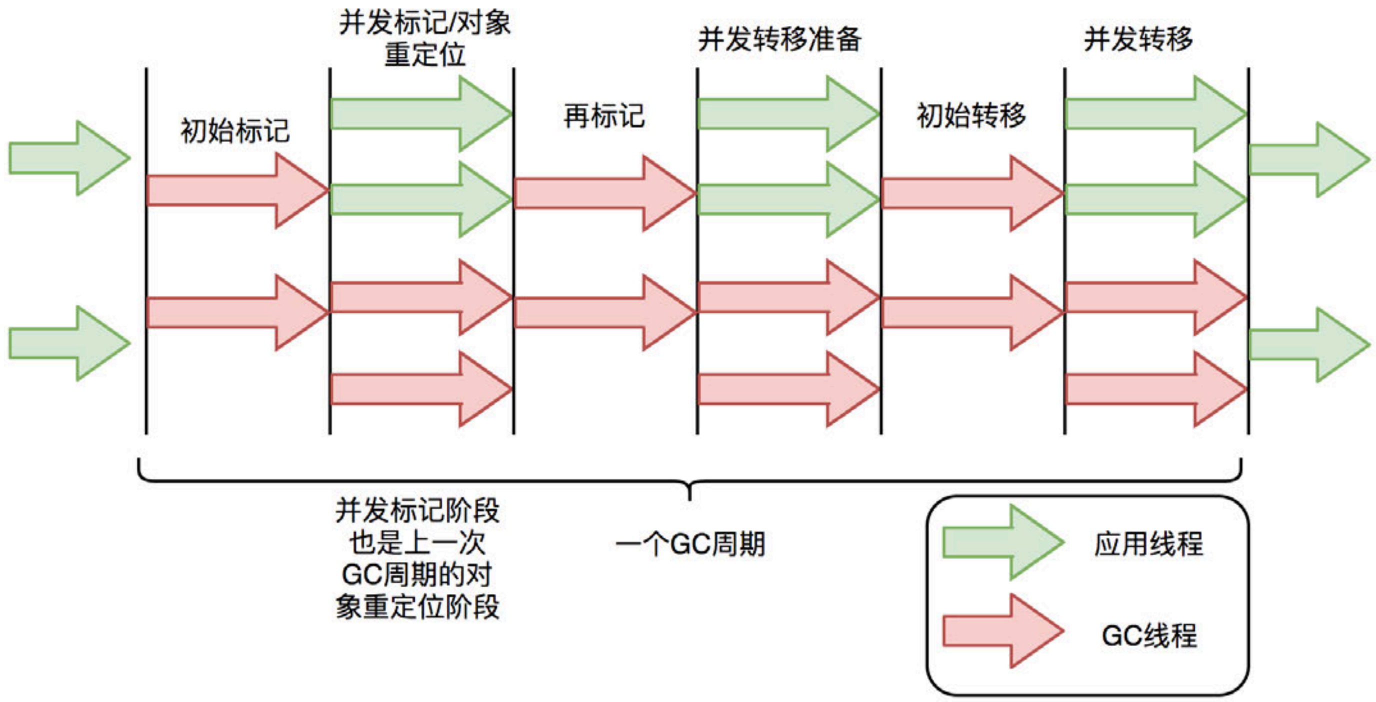


②、对象转移阶段，把活跃对象复制到新的内存地址上。

③、重定位阶段，因为转移导致对象地址发生了变化，在重定位阶段，所有指向对象旧地址的引用都要调整到对象新的地址上。

标记阶段因为只标记 GC Roots，耗时较短。但转移阶段和重定位阶段需要处理所有存活的对象，耗时较长，并且转移阶段是 STW 的，因此，G1 的性能瓶颈就主要卡在转移阶段。

与 G1 和 CMS 类似，ZGC 也采用了复制算法，只不过做了重大优化，ZGC 在标记、转移和重定位阶段几乎都是并发的，这是 ZGC 实现停顿时间小于 10ms 的关键所在。



ZGC 是怎么做到的呢?

- 指针染色（Colored Pointer）：一种用于标记对象状态的技术。
- 读屏障（Load Barrier）：一种在程序运行时插入到对象访问操作中的特殊检查，用于确保对象访问的正确性。

这两种技术可以让所有线程在并发的条件下就指针的颜色 (状态) 达成一致，而不是对象地址。因此，ZGC 可以并发的复制对象，这大大的降低了 GC 的停顿时间。

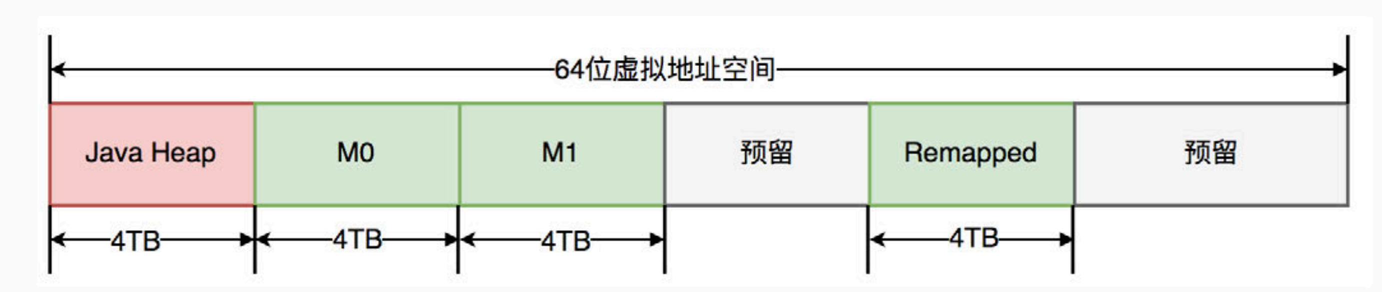
指针染色

在一个指针中，除了存储对象的实际地址外，还有额外的位被用来存储关于该对象的元数据信息。这些信息可能包括：

- 对象是否被移动了（即它是否在回收过程中被移动到了新的位置）。
- 对象的存活状态。
- 对象是否被锁定或有其他特殊状态。

通过在指针中嵌入这些信息，ZGC 在标记和转移阶段会更快，因为通过指针上的颜色就能区分出对象状态，不用额外做内存访问。

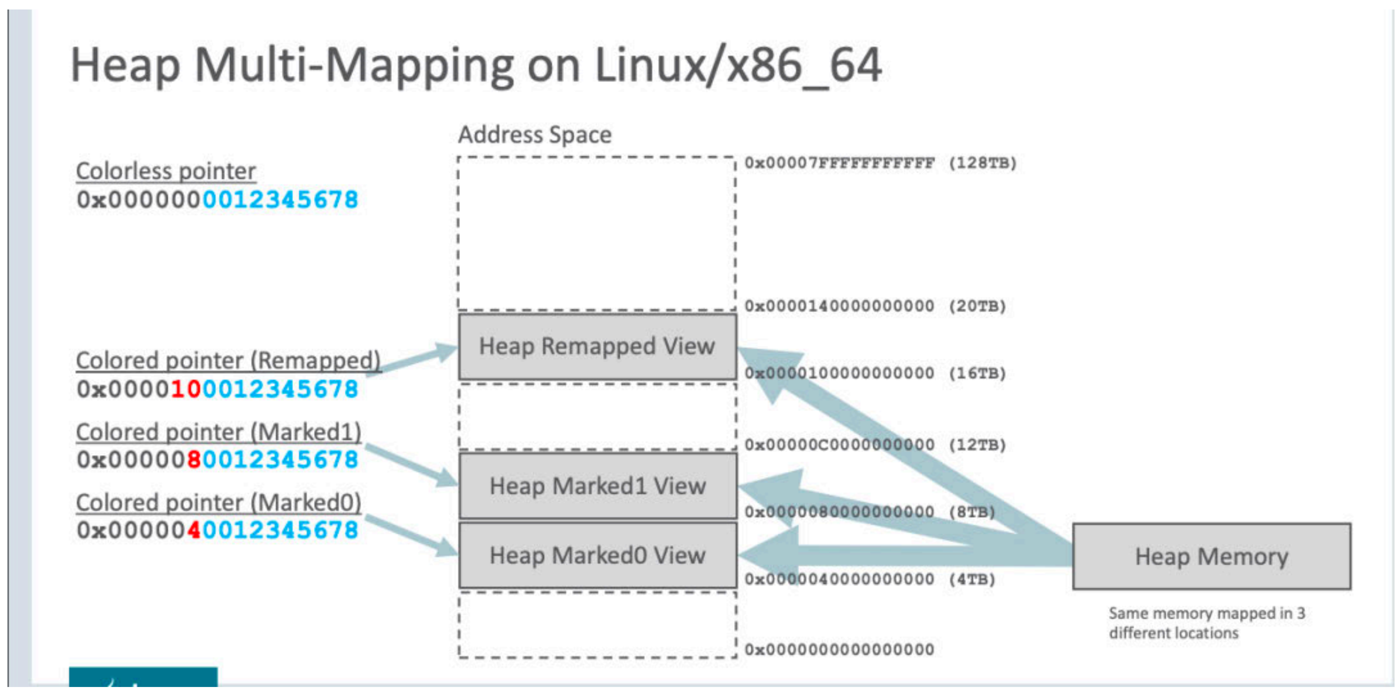
ZGC仅支持64位系统，它把64位虚拟地址空间划分为多个子空间，如下图所示：



其中，0-4TB 对应 Java 堆，4TB-8TB 被称为 M0 地址空间，8TB-12TB 被称为 M1 地址空间，12TB-16TB 预留未使用，16TB-20TB 被称为 Remapped 空间。

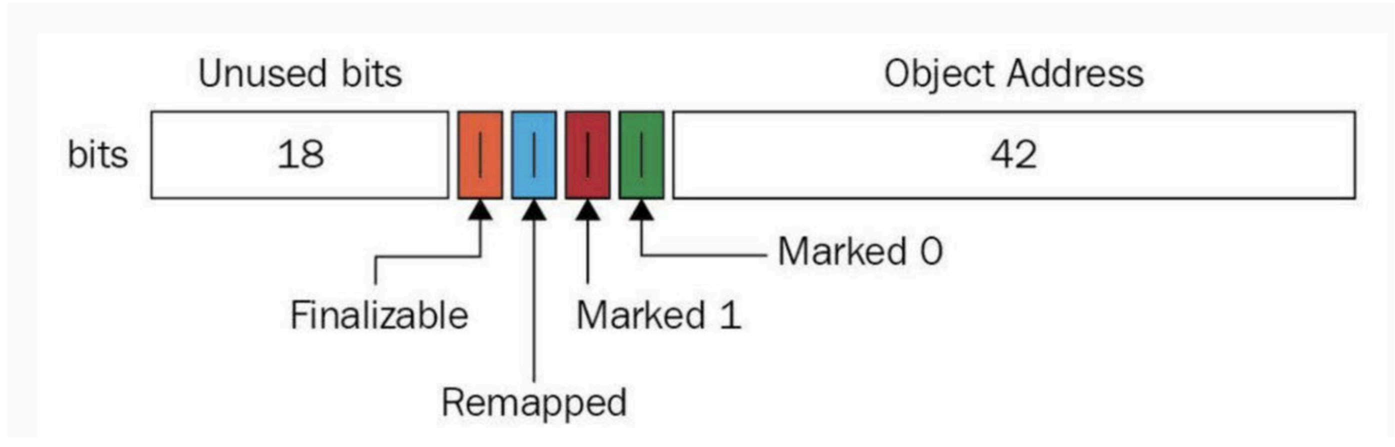
当创建对象时，首先在堆空间申请一个虚拟地址，该虚拟地址并不会映射到真正的物理地址。同时，ZGC 会在 M0、M1、Remapped 空间中为该对象分别申请一个虚拟地址，且三个虚拟地址都映射到同一个物理地址。

下图是虚拟地址的空间划分：



不过，三个空间在同一时间只有一个空间有效。ZGC 之所以设置这三个虚拟地址，是因为 ZGC 采用的是“空间换时间”的思想，去降低 GC 的停顿时间。

与上述地址空间划分相对应，ZGC实际仅使用64位地址空间的第0-41位，而第42-45位存储元数据，第47-63位固定为0。



由于仅用了第 0~43 位存储对象地址，2^{44} = 16TB，所以 ZGC 最大支持 16TB 的堆。

至于对象的存活信息，则存储在42-45位中，这与传统的垃圾回收并将对象存活信息放在对象头中完全不同。

读屏障

当程序尝试读取一个对象时，读屏障会触发以下操作：

- 检查指针染色：读屏障首先检查指向对象的指针的颜色信息。
- 处理移动的对象：如果指针表示对象已经被移动（例如，在垃圾回收过程中），读屏障将确保返回对象的新位置。
- 确保一致性：通过这种方式，ZGC 能够在并发移动对象时保持内存访问的一致性，从而减少对应用程序停顿的需要。

ZGC读屏障如何实现呢?

来看下面这段伪代码，涉及 JVM 的底层 C++ 代码：

```
// 伪代码示例，展示读屏障的概念性实现
Object* read_barrier(Object* ref) {
    if (is_forwarded(ref)) {
        return get_forwarded_address(ref); // 获取对象的新地址
    }
    return ref; // 对象未移动，返回原始引用
}
```

- read_barrier 代表读屏障。
- 如果对象已被移动（is_forwarded(ref)），方法返回对象的新地址（get_forwarded_address(ref)）。
- 如果对象未被移动，方法返回原始的对象引用。

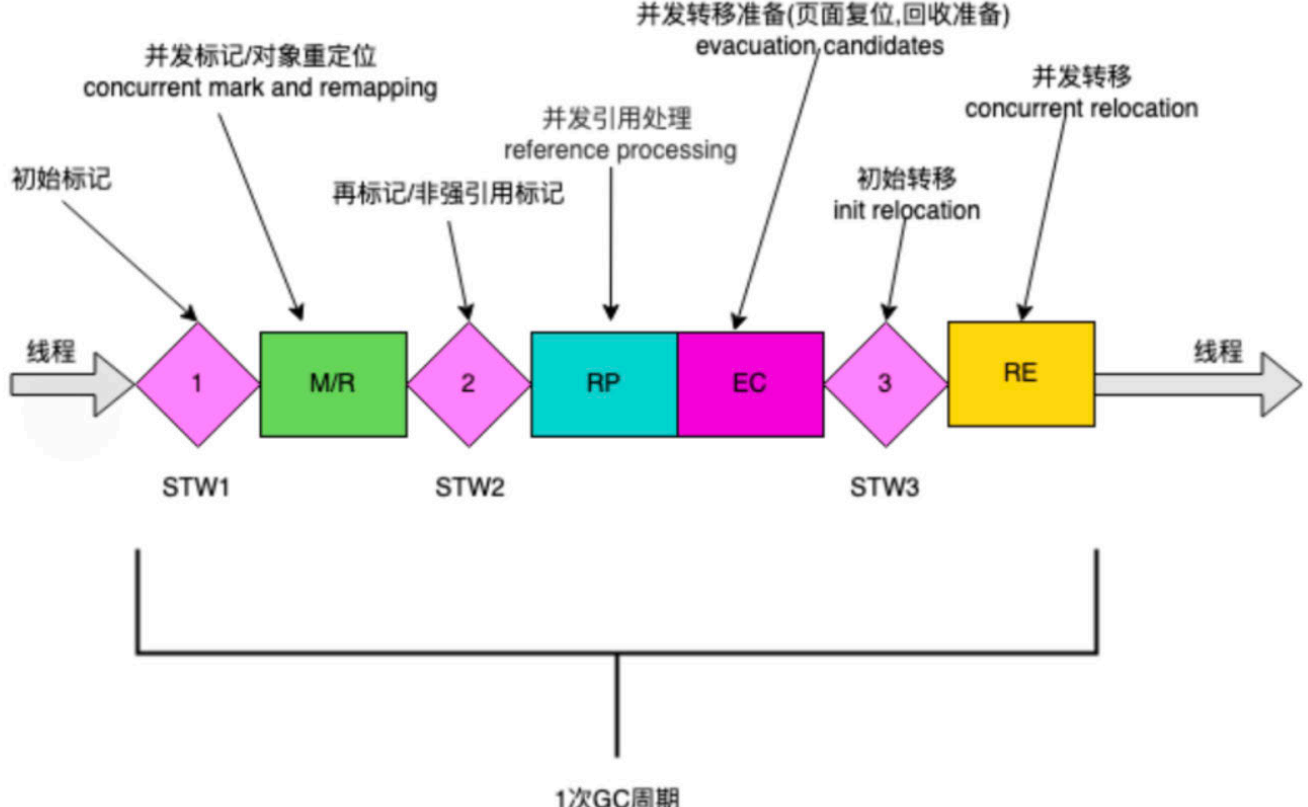
读屏障可能被GC线程和业务线程触发，并且只会在访问堆内对象时触发，访问的对象位于GC Roots时不会触发，这也是扫描GC Roots时需要STW的原因。

下面是一个简化的示例代码，展示了读屏障的触发时机。

```
Object o = obj.FieldA   // 从堆中读取引用，需要加入屏障
<Load barrier>
Object p = o  // 无需加入屏障，因为不是从堆中读取引用
o.dosomething() // 无需加入屏障，因为不是从堆中读取引用
int i =  obj.FieldB  //无需加入屏障，因为不是对象引用
```

ZGC 的工作过程

ZGC 周期由三个 STW 暂停和四个并发阶段组成：标记/重新映射(M/R)、并发引用处理(RP)、并发转移准备(EC) 和并发转移(RE)。

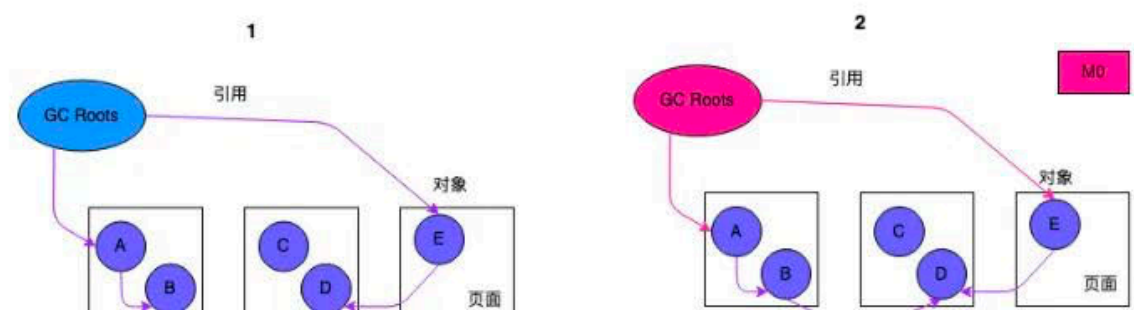


Stop-The-World 暂停阶段

1. **标记开始（Mark Start）STW 暂停**：这是 ZGC 的开始，进行 GC Roots 的初始标记。在这个短暂的停顿期间，ZGC 标记所有从 GC Root 直接可达的对象。
2. **重新映射开始（Relocation Start）STW 暂停**：在并发阶段之后，这个 STW 暂停是为了准备对象的重定位。在这个阶段，ZGC 选择将要清理的内存区域，并建立必要的数据结构以进行对象移动。
3. **暂停结束（Pause End）STW 暂停**：ZGC 结束。在这个短暂的停顿中，完成所有与该 GC 周期相关的最终清理工作。

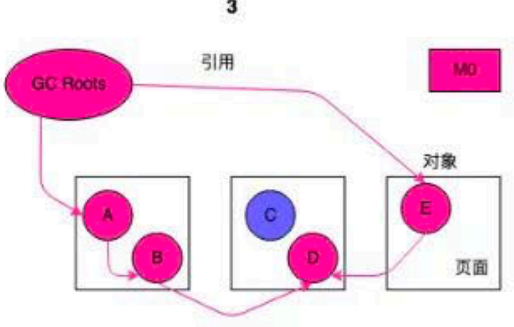
并发阶段

1. **并发标记/重新映射 (M/R)**：这个阶段包括并发标记和并发重新映射。在并发标记中，ZGC 遍历对象图，标记所有可达的对象。然后，在并发重新映射中，ZGC 更新指向移动对象的所有引用。
2. **并发引用处理 (RP)**：在这个阶段，ZGC 处理各种引用类型（如软引用、弱引用、虚引用和幽灵引用）。这些引用的处理通常需要特殊的考虑，因为它们与对象的可达性和生命周期密切相关。
3. **并发转移准备 (EC)**：这是为对象转移做准备的阶段。ZGC 确定哪些内存区域将被清理，并准备相关的数据结构。
4. **并发转移 (RE)**：在这个阶段，ZGC 将存活的对象从旧位置移动到新位置。由于这一过程是并发执行的，因此应用程序可以在大多数垃圾回收工作进行时继续运行。

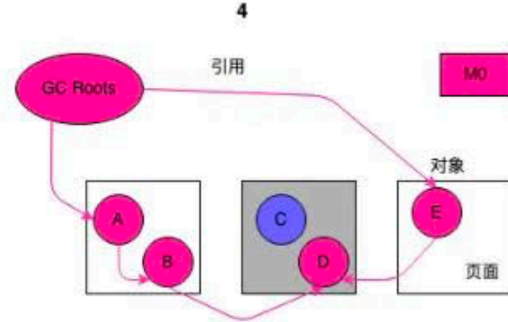


GC初始化后内存状态,4个存活对象

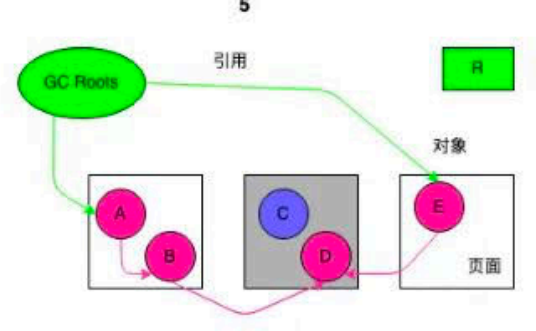
初始标记，完成GC Roots标记



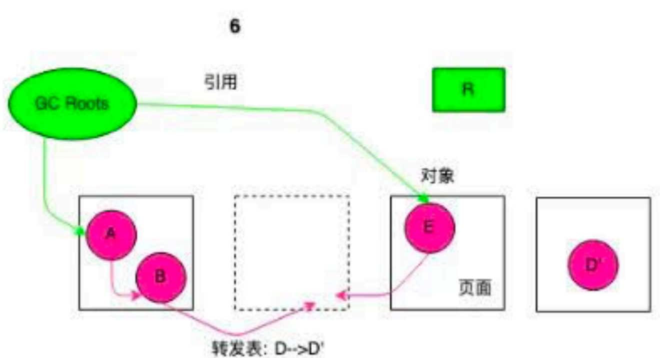
并发标记，所有可达对象被标记标记



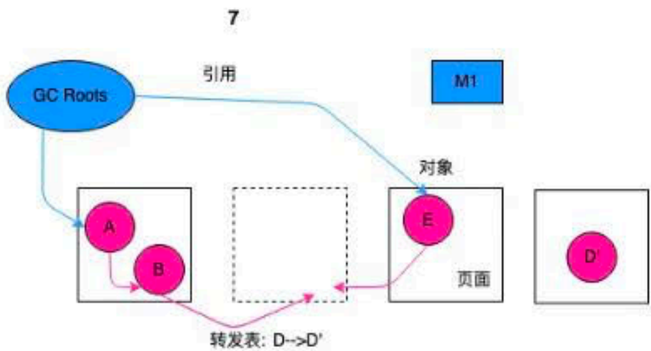
并发转移准备，中间页面被选中为转移集



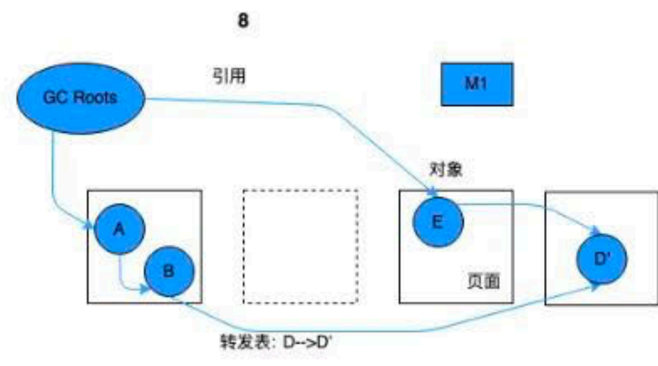
初始转移，状态被设置为Remmped



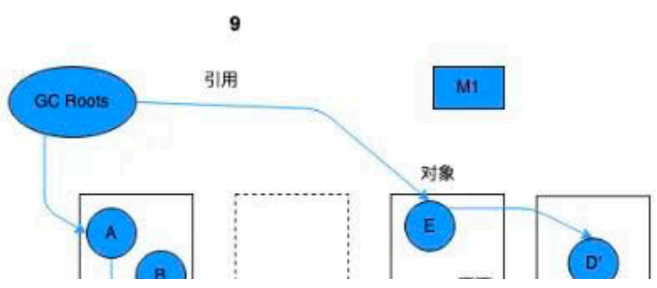
并发转移，对象D被转移到新页面



初始标记(下一周期)，完成GC Roots标记

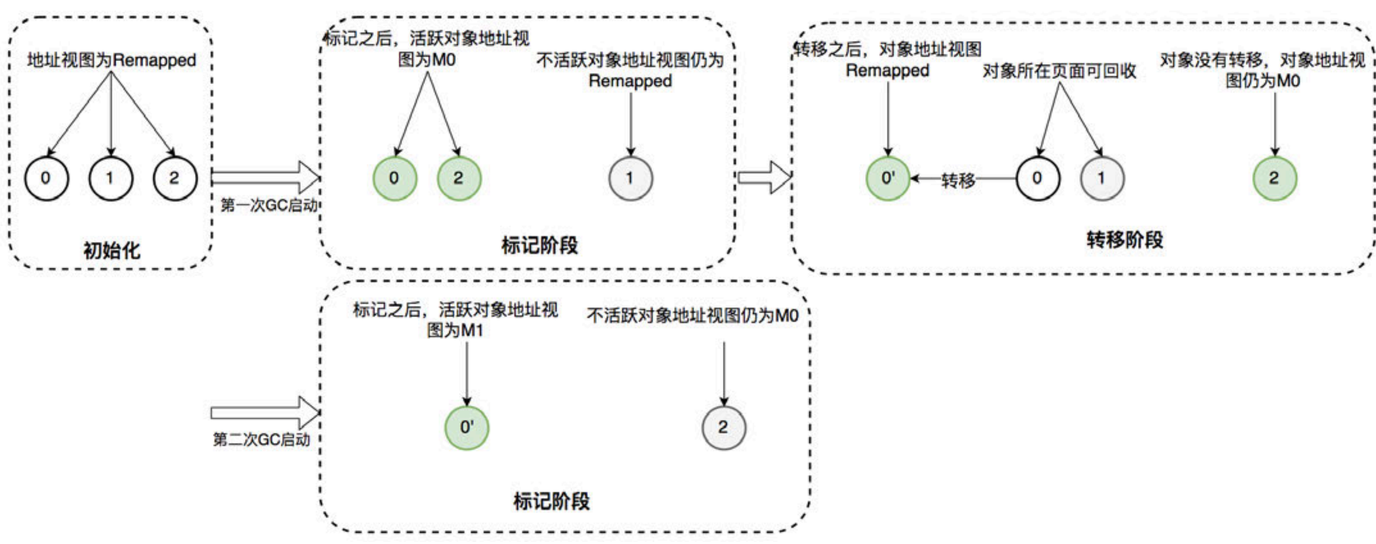


并发标记(下一周期)，失效指针被重定位



并发转移准备(下一周期)，上一周期转发表被删除

ZGC 的两个关键技术：指针染色和读屏障，不仅应用在并发转移阶段，还应用在并发标记阶段：将对象设置为已标记，传统的垃圾回收器需要进行一次内存访问，并将对象存活信息放在对象头中；而在ZGC中，只需要设置指针地址的第42-45位即可，并且因为是寄存器访问，所以速度比访问内存更快。



小结

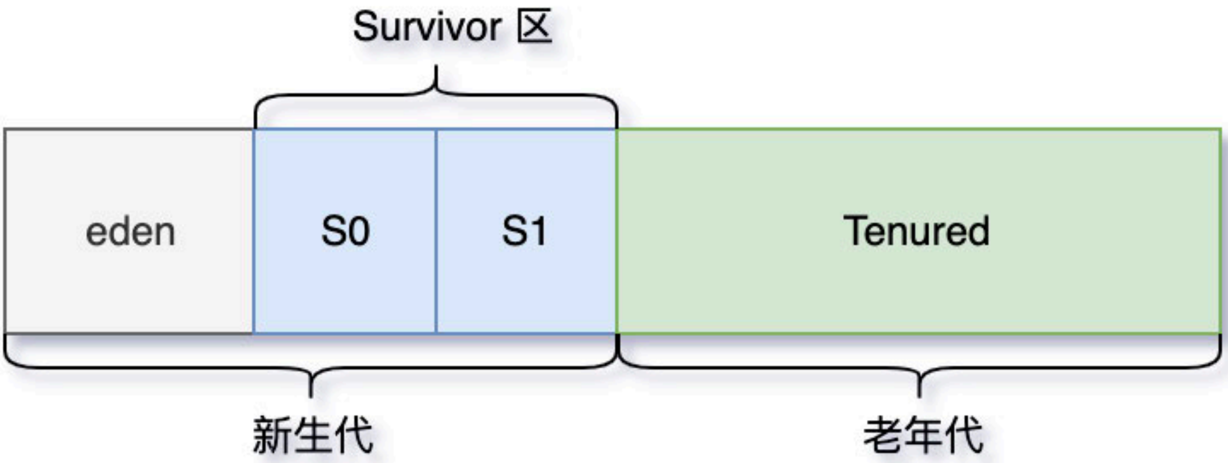
本篇内容我们主要介绍了 CMS、G1 和 ZGC 三种垃圾收集器，它们都是分区收集器，都是为了降低 GC 停顿时间而生的，但是它们各有优缺点，我们可以根据业务场景选择合适的垃圾收集器。

参考资料：

> 1、树哥聊编程：CMS 垃圾收集器
> 2、军哥聊技术：G1 垃圾收集器
> 3、美团技术专家：G1 GC 的一些关键技术
> 4、极客时间：为什么 G1 被叫做 GC 中的王
> 5、得物技术：ZGC 关键技术分析
> 6、美团技术：ZGC 的探索与实践
> 7、CoderW：ZGC 垃圾收集器

第十二节：Java创建的对象到底放在哪？

经过前面章节的学习，详细大家都知道了，Java 的对象是在堆中创建的，但堆又分为新生代和老年代，新生代又细分为 Eden、From Survivor、To Survivor。**那我们创建的对象到底在哪里**？

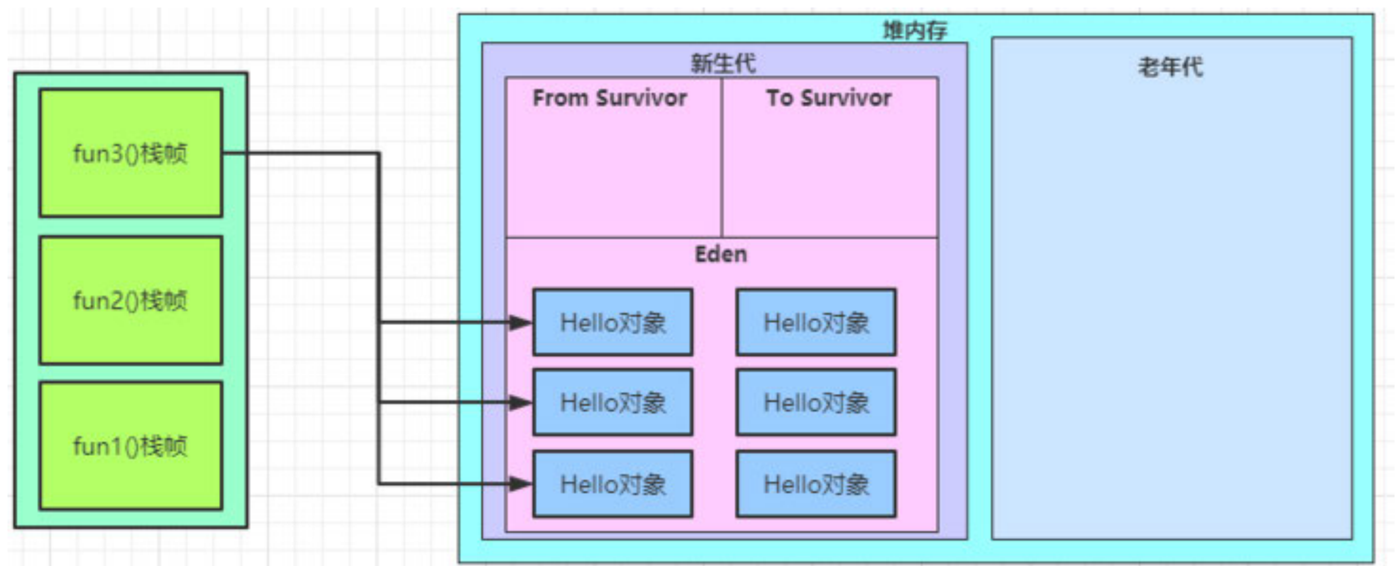


其实这部分内容我们在讲垃圾回收机制的时候提到过了，但没有细讲，这次我们就来详细讲讲。

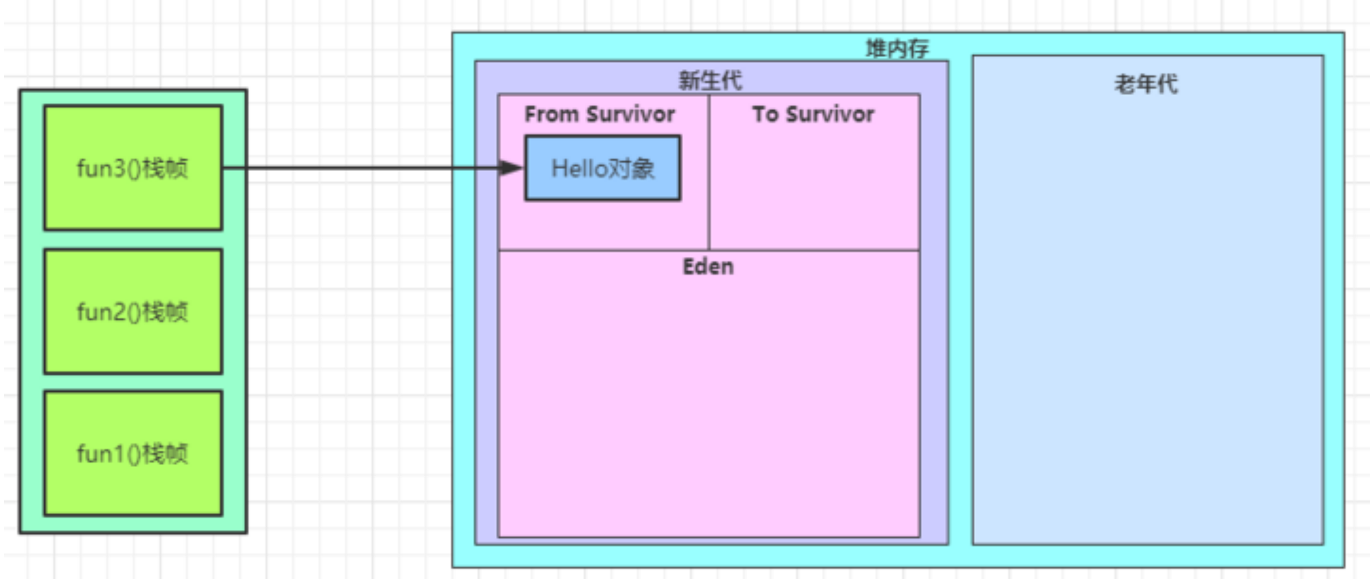
对象优先在 Eden 分配

堆分为新生代和老年代，新生代用于存放使用后就要被回收的对象（朝生夕死），老年代用于存放生命周期比较长的对象。

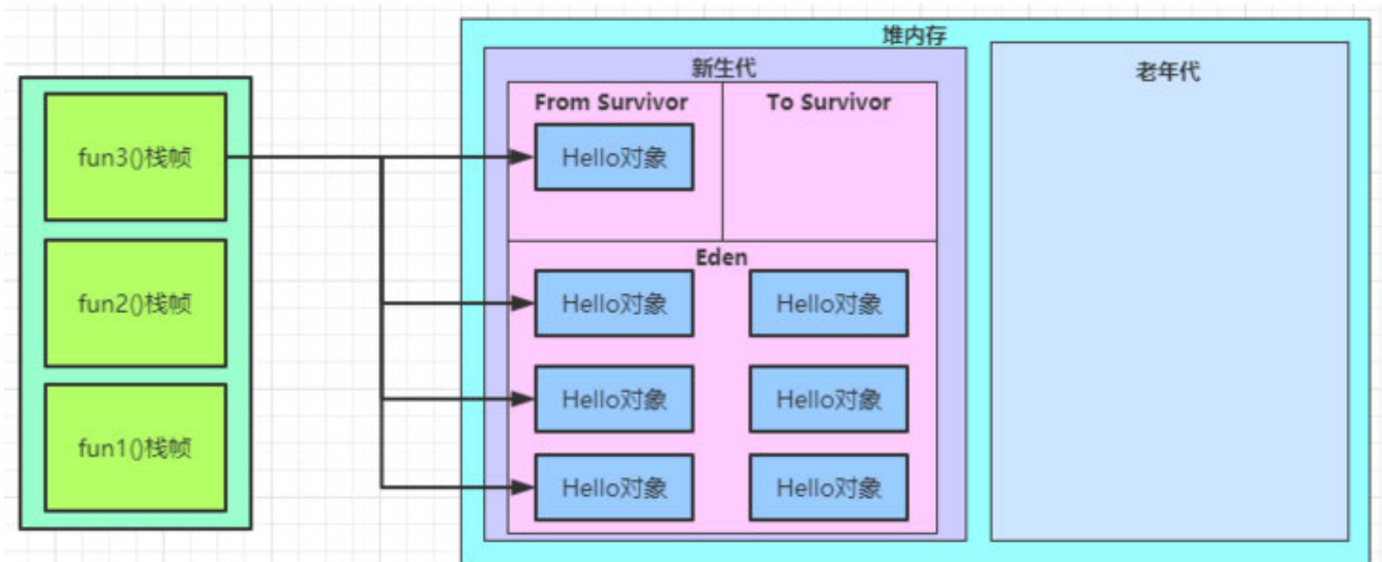
我们创建的大部分对象，都属于生命周期较短的对象，所以会存放在新生代。新生代又细分 Eden、From Survivor、To Survivor，那我们创建的对象会优先在 Eden 区分配，见下图。



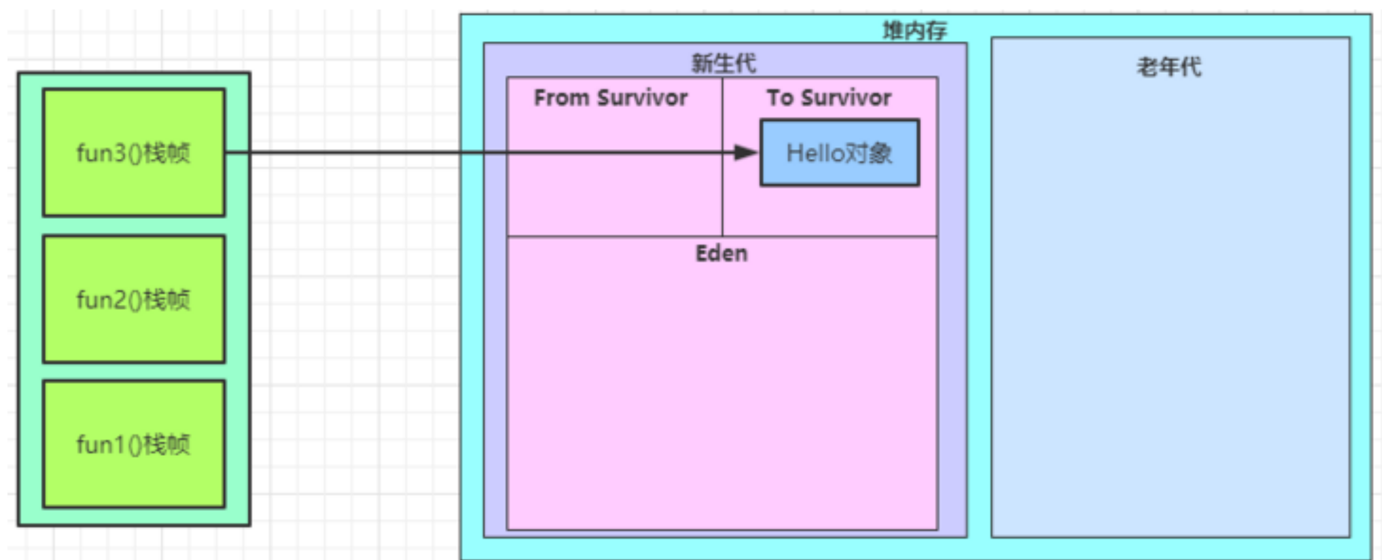
随着对象的不断创建，Eden 剩余地内存空间就会越来越少，随后就会触发 Minor GC，于是 JVM 会把 Eden 区存活的对象转入 From Survivor 空间。



Minor GC 后，又创建的新对象会继续往 Eden 区分配。



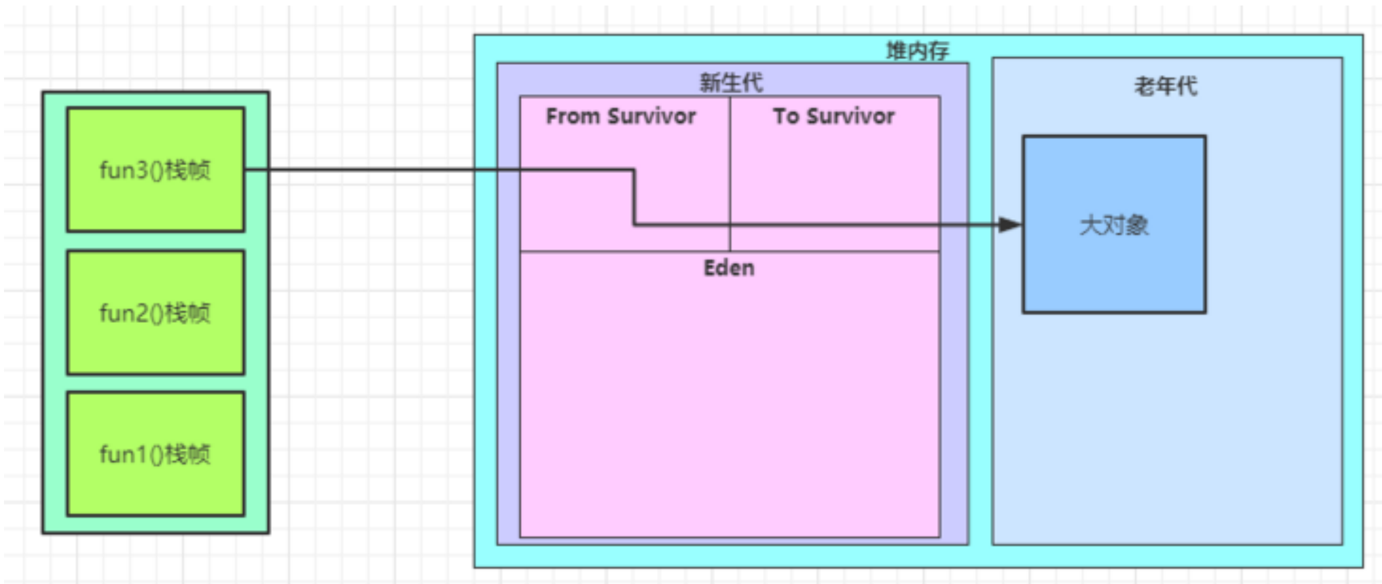
于是，随着新对象的创建，Eden 的剩余内存空间就会越来越少，又会触发 Minor GC，此时，JVM 会对 Eden 区和 From Survivor 区中的对象进行存活判断，对于存活的对象，会转移到 To Survivor 区。



下一次 Minor GC，存活的对象又会从 To 到 From，这样就总有一个 Survivor 区是空的，而另外一个是无碎片的。

大对象直接进入老年代

对于上面的流程，也有例外的存在，如果一个对象很大，一直在 Survivor 空间复制来复制去，就会很浪费性能，所以这些大对象会直接进入老年代。



这种策略的目的是减少垃圾回收时的复制开销，因为大对象的复制比小对象更耗时。

可以通过 `-XX:PretenureSizeThreshold` 参数设置直接分配大对象到老年代的阈值。如果对象的大小超过这个阈值，它将直接在老年代中分配。例如，如果想将阈值设置为 1MB（1024KB），可以这样设置：

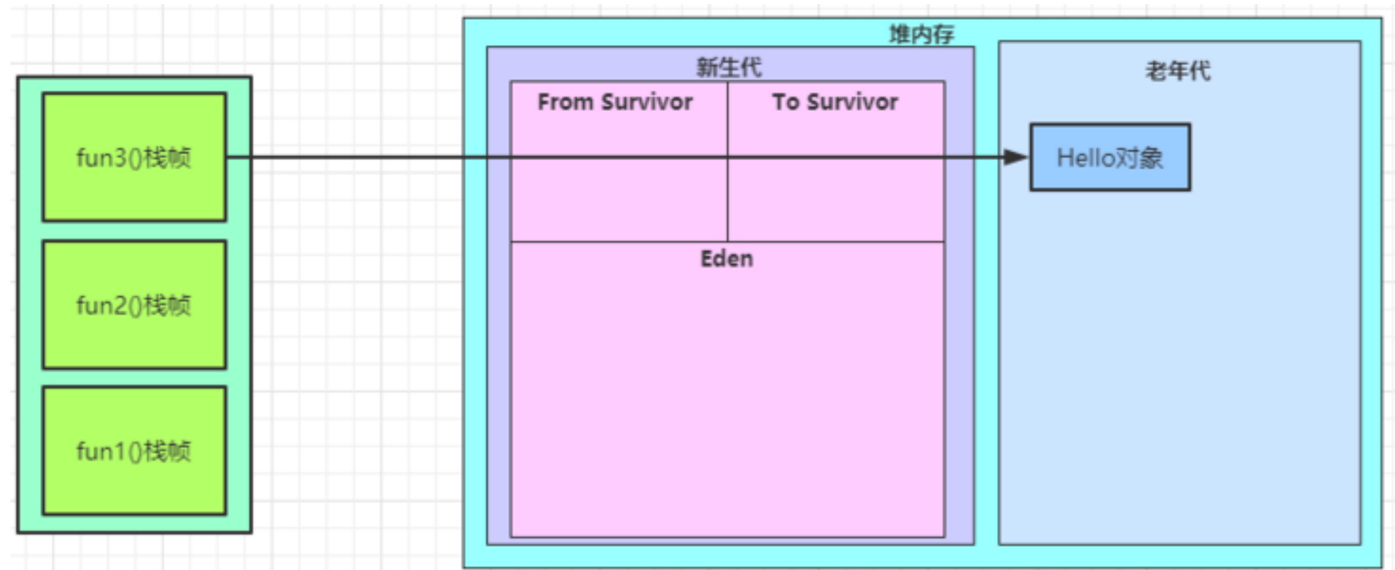
```
-XX:PretenureSizeThreshold=1048576
```

长期存活的对象将进入老年代

对象在每次从一个 Survivor 区转移到另外一个 Survivor 区时，它的年龄就会增加。当对象的年龄达到一定阈值（默认为 15），则它会被转移到老年代。

可以用 `-XX:PretenureSizeThreshold=10` 来设置年龄。

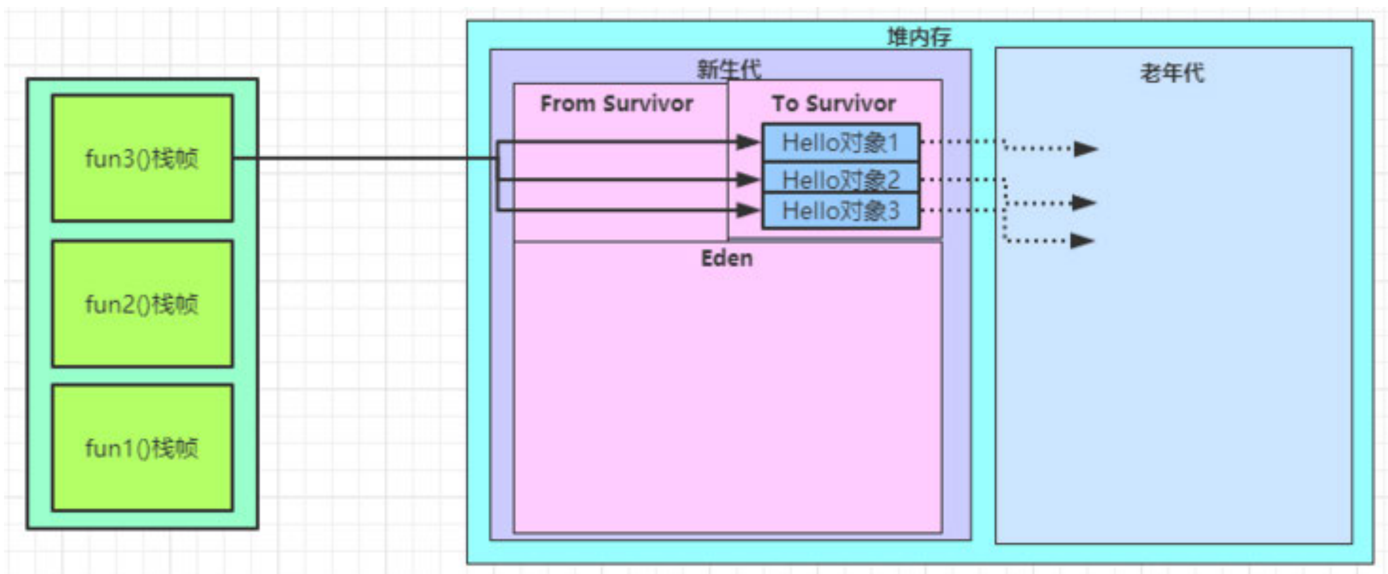
虚拟机为了给对象计算他到底经历了几次 Minor GC，会给每个对象定义了一个对象年龄计数器。如果对象在 Eden 中经过第一次 Minor GC 后仍然存活，移动到 Survivor 空间年龄加 1，在 Survivor 区中每经历过 Minor GC 后仍然存活年龄再加 1。年龄到了 15，就到了老年代。



动态年龄判断

除了年龄达到 MaxTenuringThreshold，还有另外一个方式进入老年代，那就是动态年龄判断：JVM 会检查每个年龄段的对象大小，并估算它们在 Survivor 空间中所占的总体积。JVM 会选择一个最小的年龄，使得该年龄及以上的对象可以填满 Survivor 空间的一部分（通常小于总空间的一半），然后将这些对象晋升到老年代。

比如 Survivor 是 100M，Hello1 和 Hello2 都是 3 岁，且总和超过了 50M，Hello3 是 4 岁，这个时候，这三个对象都将到老年代。



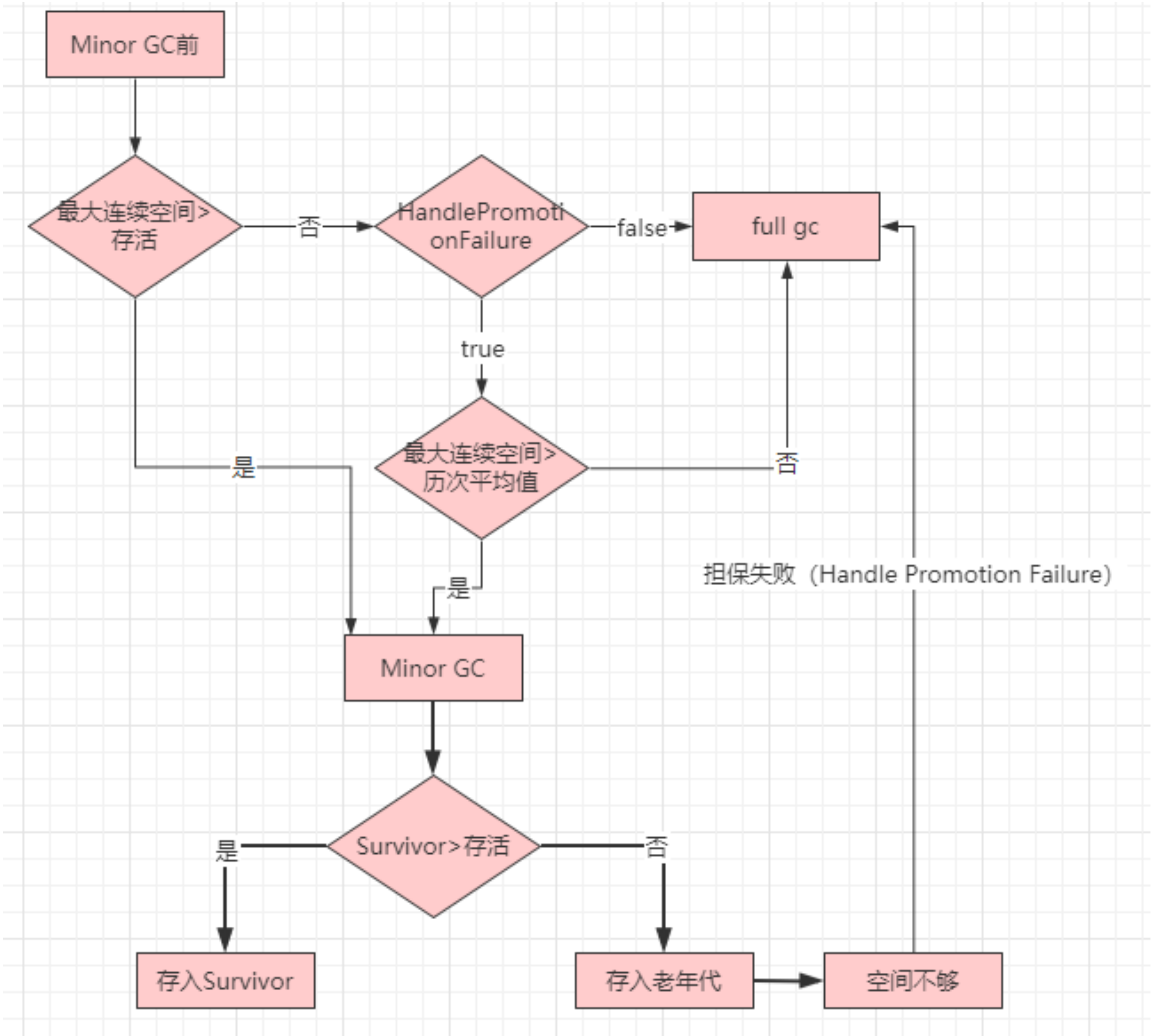
空间分配担保

上面提到过，存活的对象会放入另外一个 Survivor 空间，如果这些存活的对象比 Survivor 空间还大呢?

整个流程如下：

- Minor GC 之前，JVM 会先检查老年代最大可用的连续空间是否大于新生代所有对象的总空间，如果大于，则发起 Minor GC。
- 如果小于，则看 HandlePromotionFailure 有没有设置，如果没有设置，就发起 Full GC。
- 如果设置了 HandlePromotionFailure，则看老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果小于，就发起 Full GC。

- 如果大于，发起 Minor GC。Minor GC 后，看 Survivor 空间是否足够存放存活对象，如果不够，就放入老年代，如果够放，就直接存放 Survivor 空间。如果老年代都不够放存活对象，担保失败（Handle Promotion Failure），发起 Full GC。



> HandlePromotionFailure 的作用，当设置为 true 时（默认值），JVM 会尝试继续 Minor GC，即使老年代空间不足以容纳所有需要晋升的对象。JVM 会尝试清理更多的老年代空间或者采用其他措施来应对空间不足的情况。避免因为老年代空间不足而过早触发 Full GC（全堆回收）。Full GC 通常比 Minor GC 更耗时，会导致更长时间的停顿。

栈和方法区

Java 创建的对象几乎都在堆中，这包括通过 new 关键字创建的对象和数组。

对象的引用，通常存放在栈中，比如说当你在方法中声明一个变量 `MyClass obj = new MyClass();` 时，变量 obj（一个指向堆中对象的引用）存储在栈上。

方法区用于存储已被 JVM 加载的类信息、常量、静态变量以及即时编译器编译后的代码。

Java 8 中，永久代被元空间（Metaspace）所取代。元空间使用本地内存（操作系统的内存），而非 JVM 内存。

小结

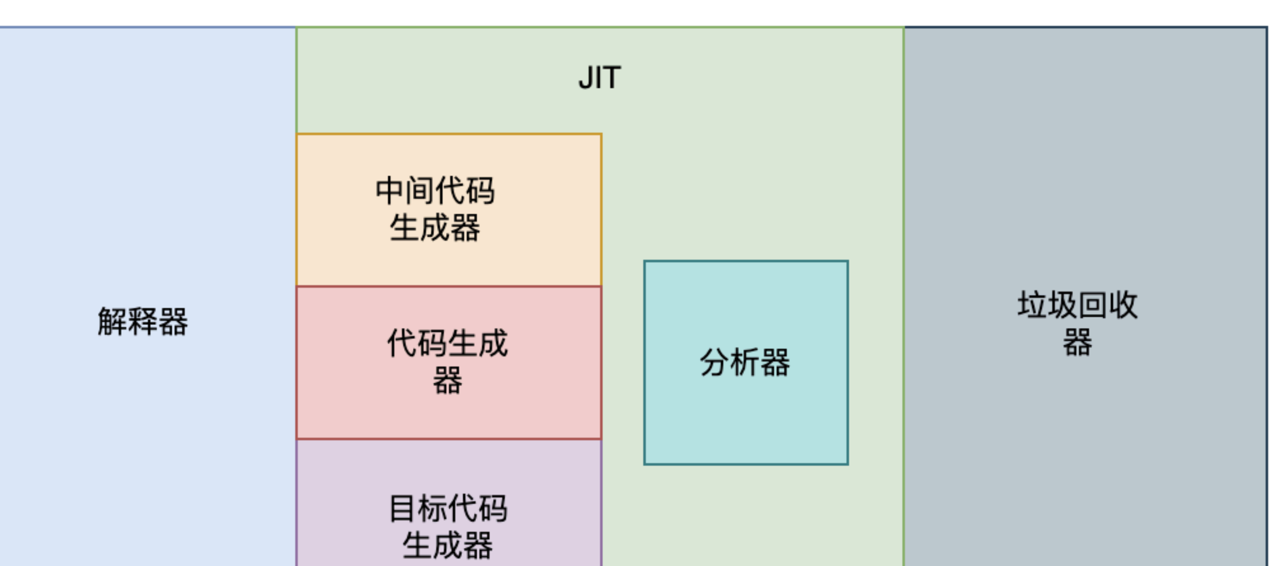
这篇内容我们主要讲了：

- Java 创建的对象优先在 Eden 区分配。
- 大对象直接进入老年代。
- 长期存活的对象将进入老年代。
- 动态年龄判断。
- 空间分配担保。

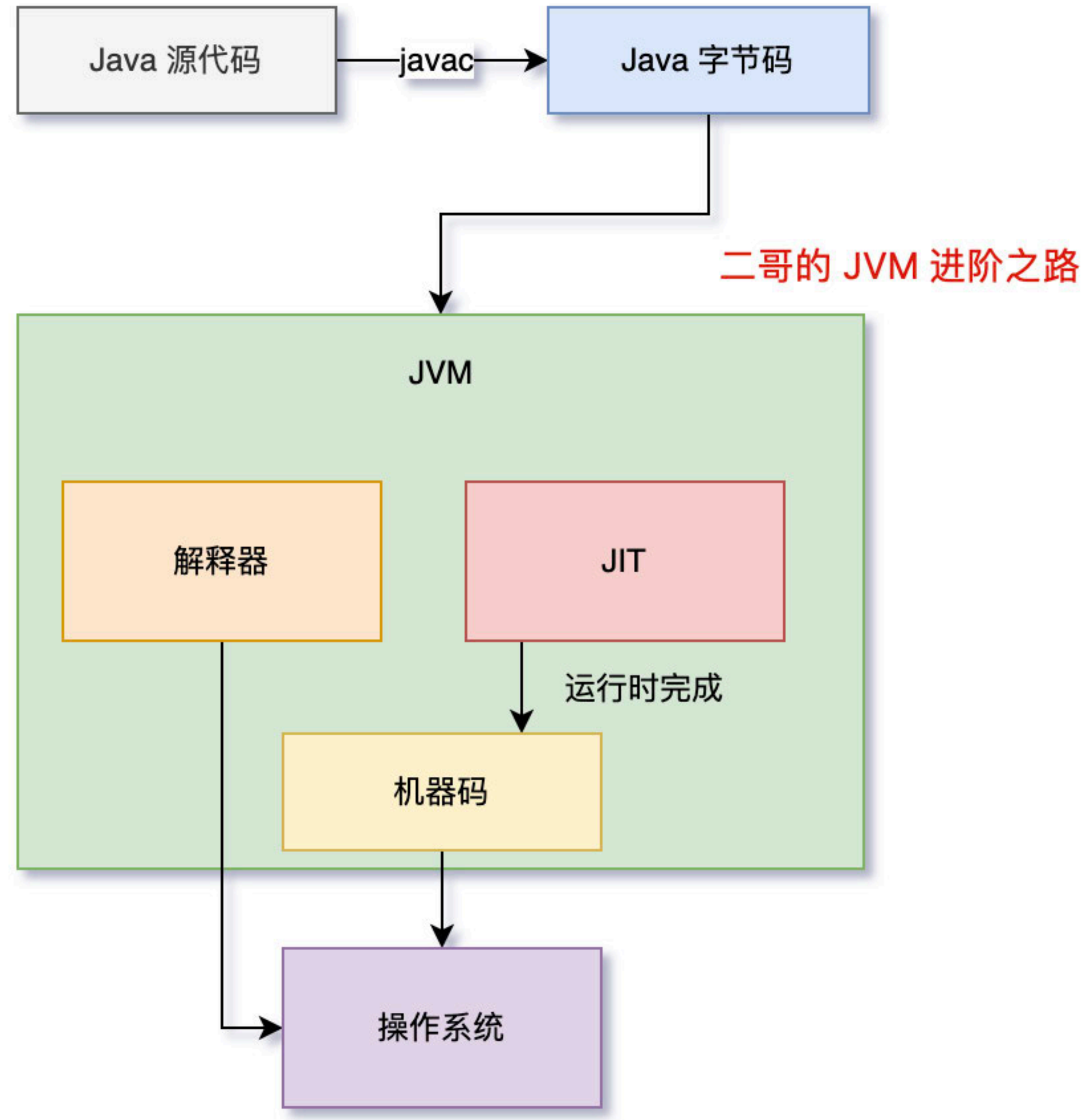
算是对我们之前讲过的垃圾回收机制中对象的转移做了补充。了解这些内存区域对于理解 Java 的内存管理和优化程序性能非常重要。

第十三节：深入理解 JIT

前面我们讲了，为了提升 Java 运行时的性能，JVM 引入了 JIT，也就是即时编译（Just In Time）技术。



Java 代码首先被编译为字节码，JVM 在运行时通过解释器执行字节码。当某部分的代码被频繁执行时，JIT 会将这些热点代码编译为机器码，以此来提高程序的执行效率。



那为什么 JIT 就能提高程序的执行效率呢，解释器不也是将字节码翻译为机器码交给操作系统执行吗?

解释器在执行程序时，对于每一条字节码指令，都需要进行一次解释过程，然后执行相应的机器指令。这个过程在每次执行时都会重复进行，因为解释器不会记住之前的解释结果。

与此相对，JIT 会将频繁执行的字节码编译成机器码。这个过程只发生一次。一旦字节码被编译成机器码，之后每次执行这部分代码时，直接执行对应的机器码，无需再次解释。

除此之外，JIT 生成的机器码更接近底层，能够更有效地利用 CPU 和内存等资源，同时，JIT 能够在运行时根据实际情况对代码进行优化（如内联、循环展开、分支预测优化等），这些优化是在机器码级别上进行的，可以显著提升执行效率。

换句话说，解释器是一个循规蹈矩的人，每次都要按照规则来执行，而 JIT 是一个“偷奸耍滑”的人，他会根据实际情况来做出最优的选择。

好，我们再来梳理一下。

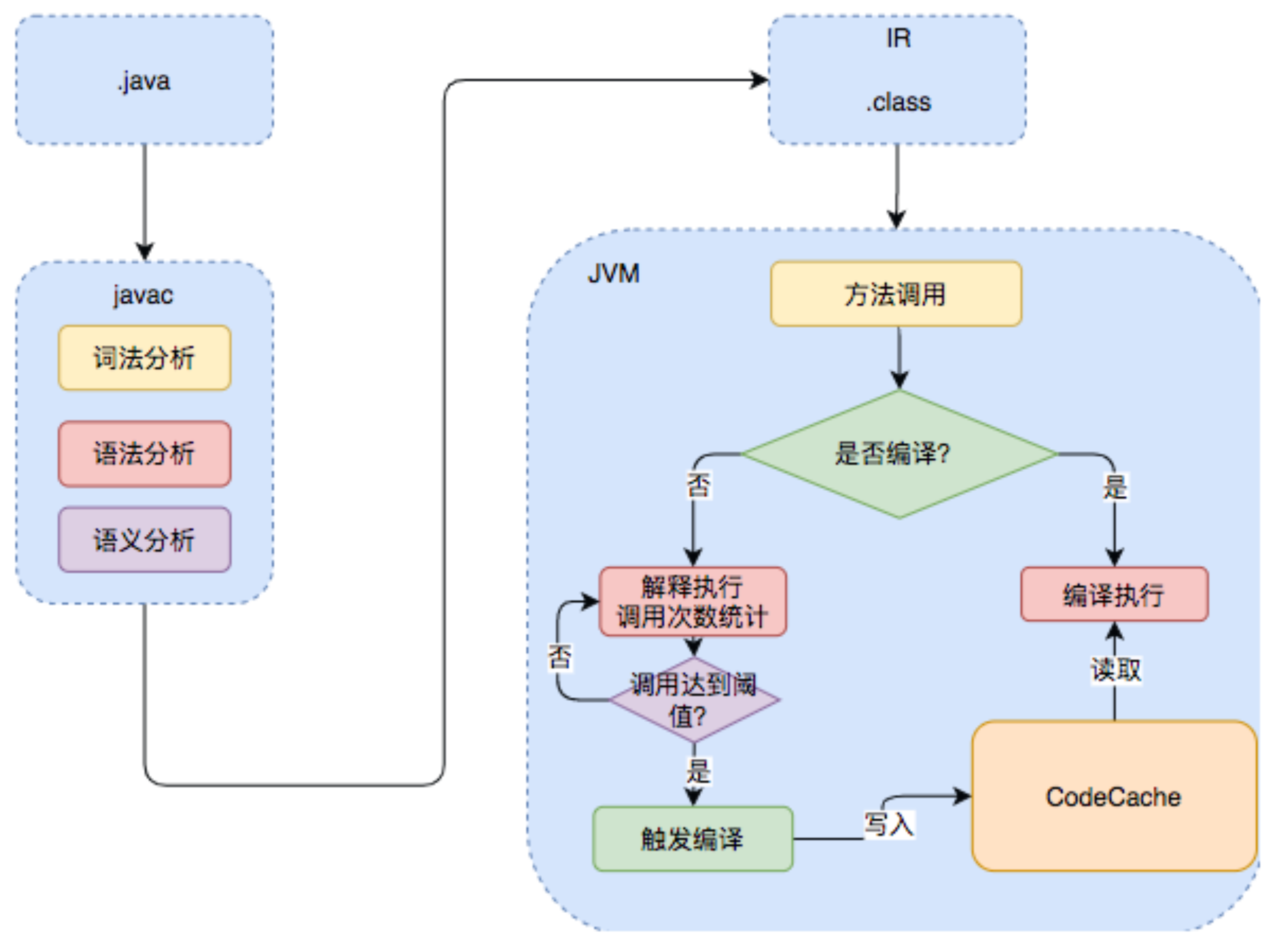
Java 的执行过程分为两步，第一步由 javac 将源码编译成字节码，在这个过程中会进行词法分析、语法分析、语义分析。

第二步，解释器会逐行解释字节码并执行，在解释执行的过程中，JVM 会对程序运行时的信息进行收集，在这些信息的基础上，JIT 会逐渐发挥作用，它会把字节码编译成机器码，但不是所有的代码都会被编译，只有被 JVM 认定为热点代码，才会被编译。

怎么样才会被认为是热点代码呢?

JVM 中有一个阈值，当方法或者代码块的在一定时间内的调用次数超过这个阈值时就会被认定为热点代码，然后编译存入 codeCache 中。当下次执行时，再遇到这段代码，就会从 codeCache 中直接读取机器码，然后执行，以此来提升程序运行的性能。

整体的执行过程大致如下图所示：



这里的 codeCache 让我想起了 Redis，Redis 也是将热点数据存储在内存中，以此来提升访问速度。

OK，解释清楚了 JIT 的原理，我们来看看 JIT 的实现。

JVM 的编译器

JVM 中集成了两种编译器，一种是 Client Compiler，另外一种是 Server Compiler。

Client Compiler 注重启动速度和局部的优化，Server Compiler 则更加关注全局的优化，性能会更好，但由于会进行更多的全局分析，所以启动速度会慢一些。

两种编译器相辅相成，互为臂膀，共同把 JVM 的性能带到了一个新的高度。

Client Compiler

就那虚拟机中的太子 HotSpot 来说吧，它就带有一个 Client Compiler，被称为 C1 编译器，启动速度极快。

C1 通常会做这三件事：

①、**局部简单可靠的优化**，比如在字节码上进行一些基础优化，方法内联、常量传播等。

我们来举例看一下什么是方法内联，假设我们有两个简单的方法：

```
public class Example {
    public int add(int a, int b) {
        return a + b;
    }

    public void run() {
        int result = add(5, 3);
        System.out.println(result);
    }
}
```

在执行 run 方法时，会调用 add 方法，方法内联优化后，会将 add 方法的字节码直接插入到 run 方法中，这样就不用再去调用 add 方法了，直接执行 run 方法就可以了。

```
public class Example {
    public void run() {
        int a = 5;
        int b = 3;
        int result = a + b;  // 这里是内联后的 add 方法体
        System.out.println(result);
    }
}
```

②、**将字节码编译成 HIR**（High-level Intermediate Representation），别计较它中文名叫什么，我觉得与其死板的翻译，不如就记住它叫 HIR，一种比较接近源代码的形式。

通过借助 HIR 我们可以实现冗余代码消除、死代码删除等编译优化工作，我们同样通过代码来看一下。

```
public class OptimizationExample {
    public int calculate(int x, int y) {
        int a = x + y;
        int b = x + y;  // 冗余计算
        return a * b;
    }
}
```

很明显，上面的代码中，b 的值是可以直接通过 a 的值计算出来的，所以 b 的计算就是冗余的，我们可以通过 HIR 来消除这种冗余计算。

```java
public class OptimizationExample {
    public int calculate(int x, int y) {
        int a = x + y;
        return a * a;  // 使用一个计算结果
    }
}
```

在 HIR 优化阶段，编译器识别到 x + y 的计算是冗余的，因此它将第二次计算的结果用第一次的结果替换。

③、**最后将 HIR 转换成 LIR**（Low-level Intermediate Representation），比较接近机器码了。这期间会做一些寄存器分配、窥孔优化等。

寄存器分配是指在编译时将程序中的变量分配到 CPU 的寄存器上。由于寄存器的访问速度远快于内存，因此合理的寄存器分配可以显著提高程序的执行效率。

来看这段代码：

```java
int a = 5;
int b = 10;
int c = a + b;
System.out.println(c);
```

在没有寄存器优化的情况下，编译器会将变量 a、b、c 分配到内存中，然后在执行时，再从内存中读取变量的值。有了寄存器分配优化呢？

```
R1 = 5      // 将 5 赋值给寄存器 R1
R2 = 10     // 将 10 赋值给寄存器 R2
R3 = R1 + R2 // 将 R1 和 R2 的和赋值给寄存器 R3
```

这样，变量 a、b、c 就被分配到了寄存器 R1、R2、R3 上，而不是内存中，寄存器的访问速度远快于内存，所以这样的优化可以提高程序的执行效率。

窥孔优化（Peephole Optimization）是一种在生成机器码阶段进行的局部优化技术。编译器“窥视”一小段生成的机器码，并尝试找出并替换更高效的指令序列。

假设有这样一段简单的机器码：

```
MOV R1, 0
ADD R1, 5
```

这段代码首先将寄存器 R1 置零，然后再将 5 加到 R1 上，窥孔优化会将这两条指令合并成一条：

```
MOV R1, 5
```

这样，仅用一条指令就完成了同样的操作，显然会提高代码执行的效率。

Server Compiler

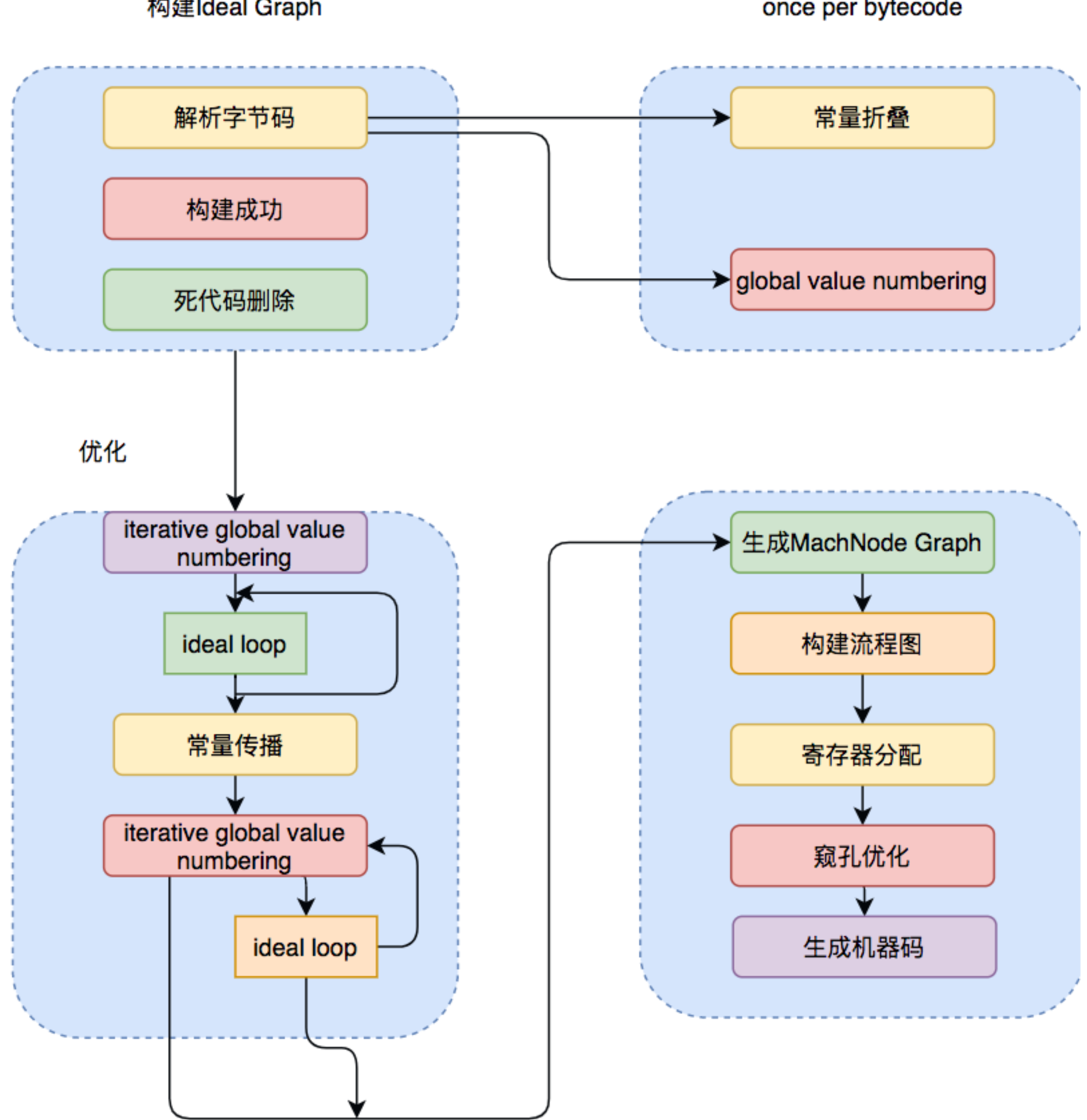
Server Compiler 主要关注一些编译耗时较长的全局优化，甚至会还会根据程序运行的信息进行一些不可靠的激进优化。这种编译器的启动时间长，适用于长时间运行的后台程序，它的性能通常比 Client Compiler 高 30%以上。目前，Hotspot 虚拟机中使用的 Server Compiler 有两种：C2 和 Graal。

C2 Compiler

Hotspot 中，默认的 Server Compiler 是 C2 编译器。

C2 编译器在进行编译优化时，会使用一种控制流与数据流结合的图数据结构，称为 Ideal Graph，我愿称之为“理想图”。

Ideal Graph 表示当前程序的数据流向和指令间的依赖关系，依靠这种图结构，某些优化步骤（尤其是涉及浮动代码块的优化步骤）会变得不那么复杂。



解析字节码的时候，C2 会向一个空的 Graph 中添加节点，Graph 中的节点通常对应一个指令块，每个指令块包含多条相关联的指令，JVM 会利用一些优化技术对这些指令进行优化，比如 Global Value Numbering、常量折叠等，解析结束后，还会进行一些死代码剔除的操作。

生成 Ideal Graph 后，会在这个基础上结合收集到的程序运行信息来进行一些全局的优化。

无论是否进行全局优化，Ideal Graph 都会被转化为一种更接近机器层面的 MachNode Graph，最后编译的机器码就是从 MachNode Graph 中得到的。

Graal Compiler

从 JDK 9 开始，Hotspot 中集成了一种新的 Server Compiler，也就是 Graal 编译器。相比 C2，Graal 有这样几种关键特性：

①、JVM 会在解释执行的时候收集程序运行的各种信息，然后根据这些信息进行一些基于预测的激进优化，比如分支预测，根据程序不同分支的运行概率，选择性地编译一些概率较大的分支。Graal 比 C2 更加青睐这种优化，所以 Graal 的峰值性能通常要比 C2 更好。

②、与 C2（主要用 C++ 编写）不同，Graal 使用 Java 语言编写。这样做的好处是，Graal 可以直接使用 JVM 的内存管理机制，不需要像 C2 那样自己实现内存管理，这样就可以避免一些内存管理上的问题。

③、Graal 引入了许多现代化的编译优化技术，例如更复杂的内联策略、循环优化等，这些在某些情况下可以比 C2 产生更优化的代码。

④、改进的逃逸分析有助于更好地进行栈上分配和锁消除，从而提升性能。

⑤、Graal 不仅能作为 JIT 编译器使用，还支持 Ahead-of-Time（AOT）编译，这有助于减少 Java 应用的启动时间和内存占用。

Graal 编译器可以通过 JVM 参数`-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler`启用。当启用时，它将替换掉 HotSpot 中的 C2，并响应原本由 C2 负责的编译请求。

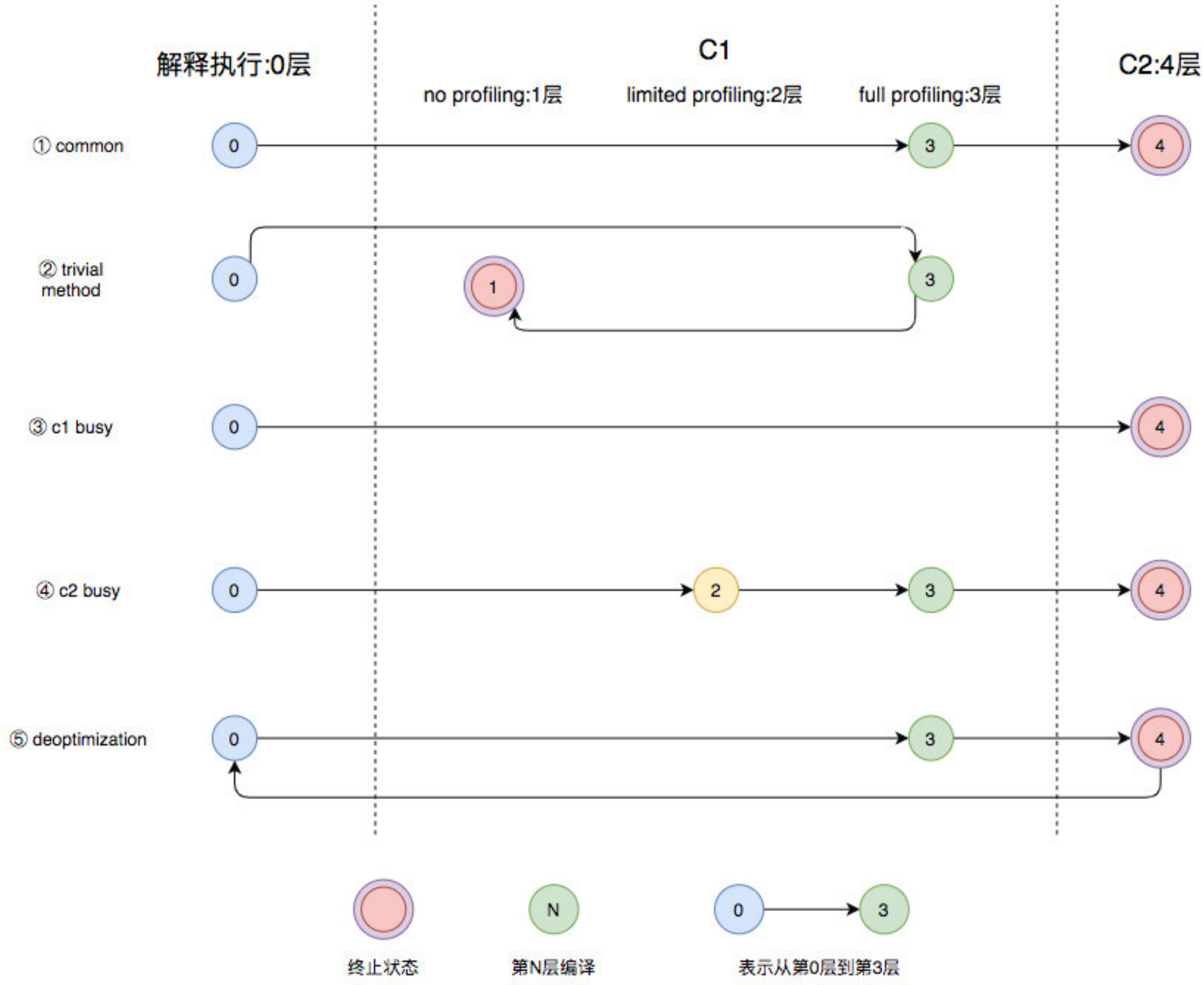
JVM 的分层编译

Java 7 引入了分层编译的概念，它结合了 C1 和 C2 的优势，追求启动速度和峰值性能的一个平衡。分层编译将 JVM 的执行状态分为了五个层次。五个层级分别是：

Java 7 中引入的分层编译（Tiered Compilation）确实是一种结合了 C1 编译器（Client Compiler）和 C2 编译器（Server Compiler）优势的技术。分层编译旨在优化 Java 程序的启动速度和长期运行时的性能。这一机制通过在不同的层级应用不同的编译策略，以达到快速启动和最高性能的平衡。在 HotSpot JVM 中，分层编译将程序执行状态分为五个层次：

1. **层级 0 - 解释器（Interpreter）**：这是程序最初执行的阶段，代码通过解释器逐行解释执行。这一阶段的目的是尽快开始执行而不等待编译完成。
2. **层级 1 - C1 编译器带有轻量级优化（C1 with Simple Optimizations）**：在这一层级，代码首次由 C1 编译器编译，应用了一些基本的优化，如方法内联。这一阶段的编译速度较快，能迅速提供优于解释执行的性能。
3. **层级 2 - C1 编译器带有完整优化（C1 with Full Optimizations）**：此层级仍由 C1 编译器处理，但应用了更多优化技术，如逃逸分析。虽然这些优化需要更长的编译时间，但能进一步提升运行性能。
4. **层级 3 - C1 编译器带有分析数据收集（C1 with Profiling）**： 在这个层级，C1 编译器除了执行优化，还收集方法执行的详细分析数据（如分支频率、热点代码等）。这些数据将用于 C2 编译器的后续优化。
5. **层级 4 - C2 编译器优化（C2 Optimizations）**：最终阶段由 C2 编译器处理，它使用收集的分析数据进行深入优化。C2 编译器的优化更加彻底和复杂，适用于长时间运行的代码，能够提供最佳的运行性能。

下图中列举了几种常见的编译路径：



1）图中第 ① 条路径，代表编译的一般情况，热点方法从解释执行到被 3 层的 C1 编译，最后被 4 层的 C2 编译。

2）如果方法比较小（比如 getter/setter），3 层的 profiling 没有收集到有价值的数据，JVM 就会断定该方法对于 C1 代码和 C2 代码的执行效率相同，就会执行图中第 ② 条路径。

在这种情况下，JVM 会在 3 层编译之后，放弃进入 C2 编译，直接选择用 1 层的 C1 编译运行。

3）在 C1 忙碌的情况下，执行图中第 ③ 条路径，在解释执行过程中对程序进行 profiling，根据信息直接由第 4 层的 C2 编译。

4）C1 中的执行效率是 1 层>2 层>3 层，第 3 层一般要比第 2 层慢 35%以上，所以在 C2 忙碌的情况下，执行图中第 ④ 条路径。这时方法会被 2 层的 C1 编译，然后再被 3 层的 C1 编译，以减少方法在 3 层的执行时间。

5）如果编译器做了一些比较激进的优化，比如分支预测，在实际运行时发现预测出错，这时就会进行反优化，重新进入解释执行，图中第 ⑤ 条执行路径代表的就是反优化。

分层编译通过在不同的阶段应用不同程度的优化，既提供了较快的应用启动时间，又确保了长时间运行的应用能达到峰值性能。这种动态适应的编译策略是 Java 平台持续优化性能的关键手段之一。

从 JDK 8 开始，JVM 默认开启分层编译。

JIT 的触发

JVM 根据方法的调用次数以及循环回边的执行次数来触发 JIT。

循环回边是一个控制流图中的概念，程序中可以简单理解为往回跳转的指令，比如下面这段代码：

```
public void nlp(Object obj) {
  int sum = 0;
  for (int i = 0; i < 200; i++) {
    sum += i;
  }
}
```

上面这段代码经过编译生成下面的字节码。

```
public void nlp(java.lang.Object);
    Code:
       0: iconst_0
       1: istore_1
       2: iconst_0
       3: istore_2
       4: iload_2
       5: sipush        200
       8: if_icmpge     21
      11: iload_1
      12: iload_2
      13: iadd
      14: istore_1
      15: iinc          2, 1
      18: goto          4
      21: return
```

其中，偏移量为 18 的字节码将往回跳至偏移量为 4 的字节码中。在解释执行时，每当运行一次该指令，JVM 便会将该方法的循环回边计数器加 1。

在即时编译过程中，编译器会识别循环的头部和尾部。上面这段字节码中，循环体的头部和尾部分别为偏移量为 11 的字节码和偏移量为 15 的字节码。编译器将在循环体结尾增加循环回边计数器的代码，来对循环进行计数。

当方法的调用次数和循环回边的次数的和，超过由参数 `-XX:CompileThreshold` 指定的阈值时，就会触发即时编译。

> C1 默认值为 1500；C2 默认值为 10000。

开启分层编译的情况下，`-XX:CompileThreshold` 参数设置的阈值将会失效，触发及时编译会由以下的条件来判断：

- 方法调用次数大于由参数`-XX:TierXInvocationThreshold` 指定的阈值乘以系数。
- 方法调用次数大于由参数`-XX:TierXMINInvocationThreshold` 指定的阈值乘以系数，并且方法调用次数和循环回边次数之和大于由参数`-XX:TierXCompileThreshold` 指定的阈值乘以系数时。

分层编译触发条件公式(i 为调用次数，b 是循环回边次数，s 是系数)：

```
i > TierXInvocationThreshold * s || (i > TierXMinInvocationThreshold * s  && i + b >
TierXCompileThreshold * s)
```

满足其中一个条件就会触发即时编译，并且 JVM 会根据当前的编译方法数以及编译线程数动态调整系数 s。

JIT 的编译优化

即时编译器会对正在运行的程序进行一系列优化，包括：

- 字节码解析过程中的分析
- 根据编译过程中代码的一些中间形式来做局部优化
- 根据程序依赖图进行全局优化

最后才会生成机器码。

中间表达形式

在编译原理中，通常会把编译器分为前端和后端，前端编译经过词法分析、语法分析、语义分析生成中间表达形式 IR（Intermediate Representation），后端会对 IR 进行优化，生成目标代码。

Java 字节码就是一种 IR，但是字节码的结构复杂，也不适合做全局的分析优化。

现代编译器一般采用图结构的 IR，也就是所谓的静态单赋值——Static Single Assignment，SSA 是目前比较常用的一种 IR。这种 IR 的特点是每个变量只能被赋值一次，而且只有当变量被赋值之后才能使用。

举个例子（前面也讲过，这里再强调一遍）：

```
{
  a = 1;
  a = 2;
  b = a;
}
```

我们可以轻易地发现 a = 1 的赋值是冗余的。传统的编译器需要借助数据流分析，从后至前依次确认哪些变量的值被覆盖掉了。不过，如果借助了 SSA IR，编译器则可以很容易识别冗余赋值。

上面代码的 SSA IR 形式的伪代码可以表示为：

```
{
  a_1 = 1;
  a_2 = 2;
  b_1 = a_2;
}
```

由于 SSA IR 中每个变量只能赋值一次，所以代码中的 a 在 SSA IR 中会分成 a_1、a_2 两个变量来赋值，这样编译器就可以很容易通过扫描这些变量来发现 a_1 的赋值后并没有使用，由此认定该赋值是冗余的。

除此之外，SSA IR 对其他优化方式也有很大的帮助，例如下面这个死代码删除（Dead Code Elimination）的例子：

```
public void DeadCodeElimination{
  int a = 2;
  int b = 0
  if(2 > 1){
    a = 1;
  } else{
    b = 2;
  }
  add(a,b)
}
```

可以得到 SSA IR 伪代码：

```
a_1 = 2;
b_1 = 0
if true:
  a_2 = 1;
else
  b_2 = 2;
add(a,b)
```

编译器通过执行字节码可以发现 else 分支不会被执行。经过死代码删除后就可以得到代码：

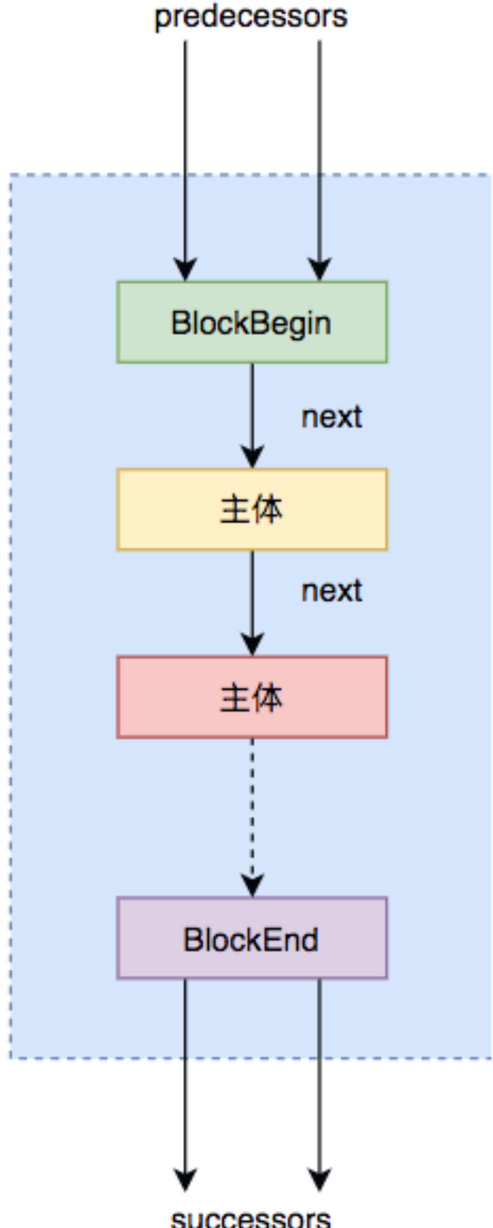
```
public void DeadCodeElimination{
  int a = 1;
  int b = 0;
  add(a,b)
}
```

我们可以将编译器的每一种优化看成一个图优化算法，它接收一个 IR 图，并输出经过转换后的 IR 图。编译器优化的过程就是一个个图节点的优化串联起来的。

C1 的 HIR

前文提到了 C1 编译器内部使用了高级中间表达形式 HIR，低级中间表达形式 LIR 来进行各种优化，这两种 IR 都是 SSA 形式的。

HIR 是由很多基本块（Basic Block）组成的控制流图结构，每个块包含很多 SSA 形式的指令。基本块的结构如下图所示：



其中，predecessors 表示前驱基本块，由于前驱可能是多个，所以是 BlockList 结构，由多个 BlockBegin 组成的可扩容数组。

同样，successors 表示多个后继基本块 BlockEnd。

除了这两部分就是主体块，里面包含程序执行的指令和一个 next 指针，指向下一个执行的主体块。

从字节码到 HIR 的构造最终调用的是 GraphBuilder，GraphBuilder 会遍历字节码，将所有代码基本块存储为一个链表结构，但是这个时候的基本块只有 BlockBegin，不包括具体的指令。

第二步 GraphBuilder 会用一个 ValueStack 作为操作数栈和局部变量表，模拟执行字节码，构造出对应的 HIR，填充之前空的基本块，这里给出简单字节码块构造 HIR 的过程示例，如下所示：

```
字节码                        Local Value                operand stack              HIR
5: iload_1                      [i1,i2]                    [i1]
6: iload_2                      [i1,i2]                    [i1,i2]
                             ................................................     i3: i1 *
i2
7: imul
8: istore_3                     [i1,i2, i3]                 [i3]
```

可以看出，当执行 iload_1 时，操作数栈压入变量 i1，执行 iload_2 时，操作数栈压入变量 i2，执行相乘指令 imul 时弹出栈顶两个值，构造出 HIR i3 : i1 * i2，生成的 i3 入栈。

C1 编译器的大部分优化工作都是在 HIR 之上完成的。当优化完成之后它会将 HIR 转化为 LIR，LIR 和 HIR 类似，也是一种编译器内部用到的 IR，HIR 通过优化消除一些中间节点就可以生成 LIR，形式上更加简化。

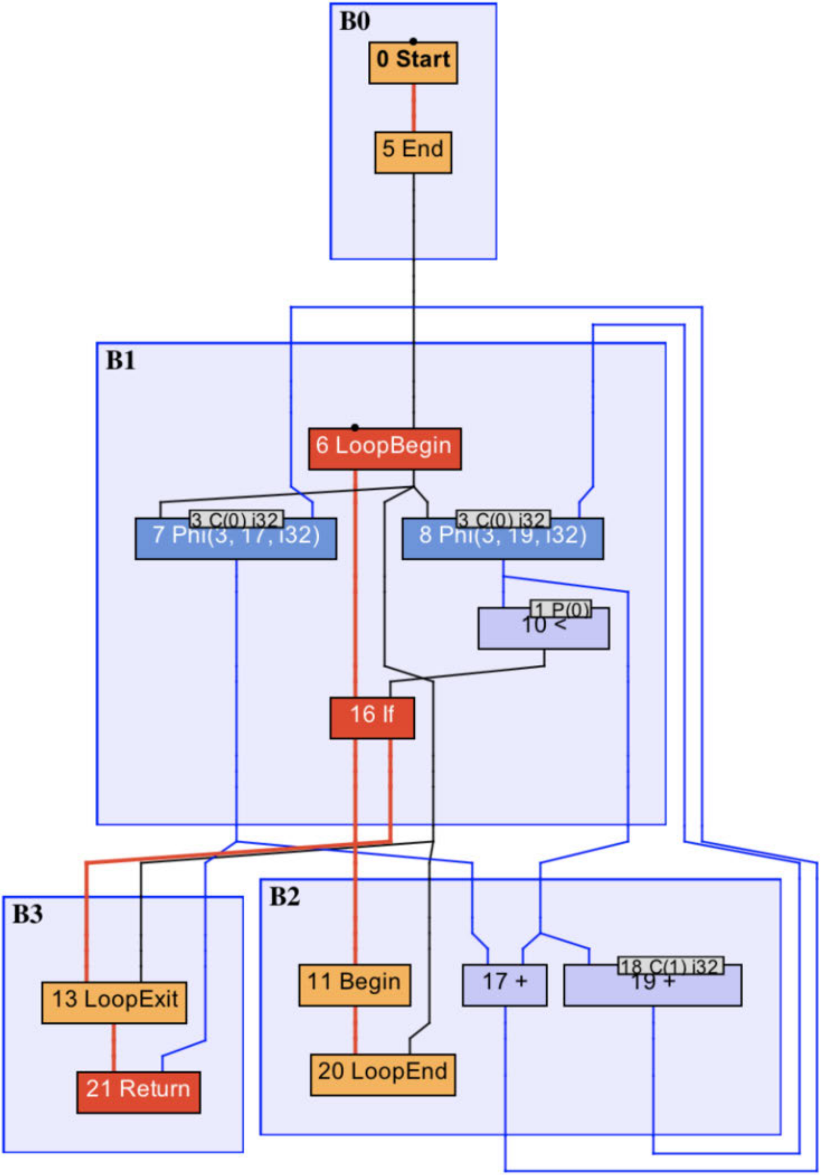
C2 的 Sea-of-Nodes IR

C2 编译器中的 Ideal Graph 采用的是一种名为 Sea-of-Nodes 中间表达形式，同样也是 SSA 形式。

它最大的特点是去除了变量的概念，直接采用值来进行运算。为了方便理解，可以利用 IR 可视化工具 Ideal Graph Visualizer（IGV），来展示具体的 IR 图。比如下面这段代码：

```java
public static int foo(int count) {
  int sum = 0;
  for (int i = 0; i < count; i++) {
    sum += i;
  }
  return sum;
}
```

对应的 IR 图如下所示：



B0 基本块中 0 号 Start 节点是方法入口，B3 中 21 号 Return 节点是方法出口。

红色加粗线条为控制流，蓝色线条为数据流，其他颜色的线条则是特殊的控制流或数据流。

被控制流所连接的是固定节点，其他的则是浮动节点。

依赖于这种图结构，通过收集程序运行的信息，JVM 可以通过 Schedule 那些浮动节点，从而获得最好的编译效果。

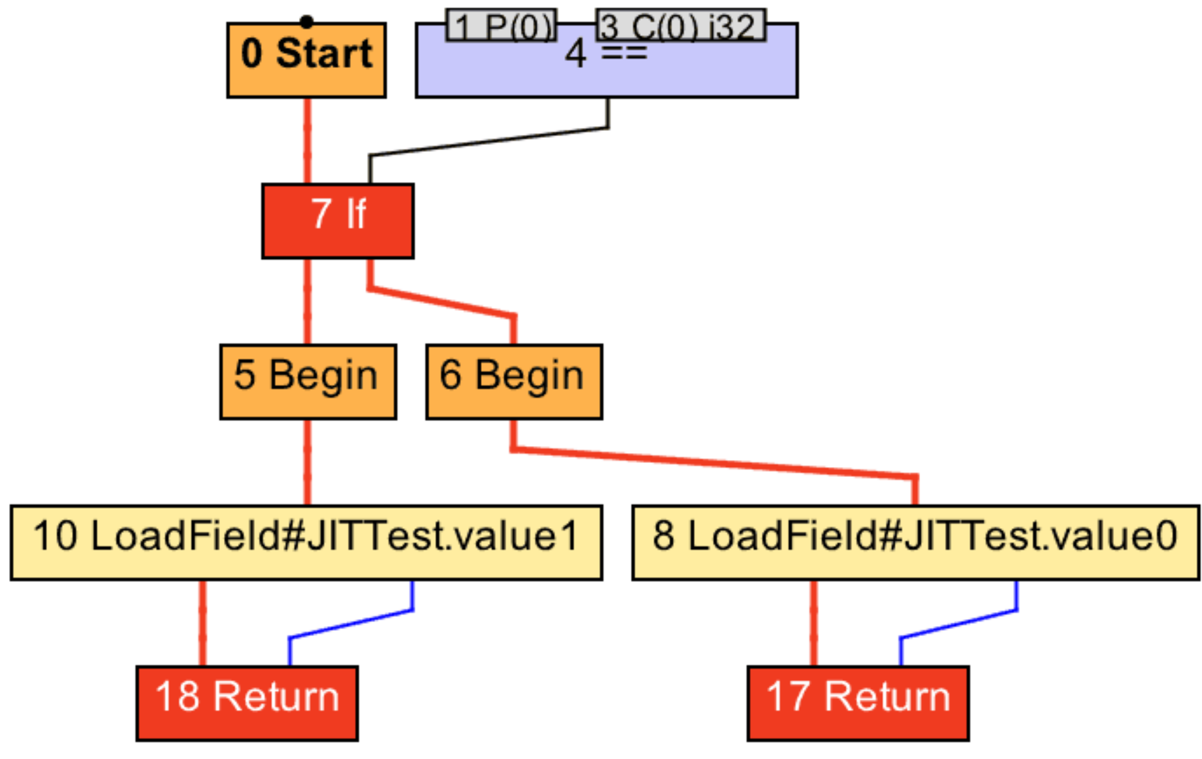
方法内联

来看下面这段代码：

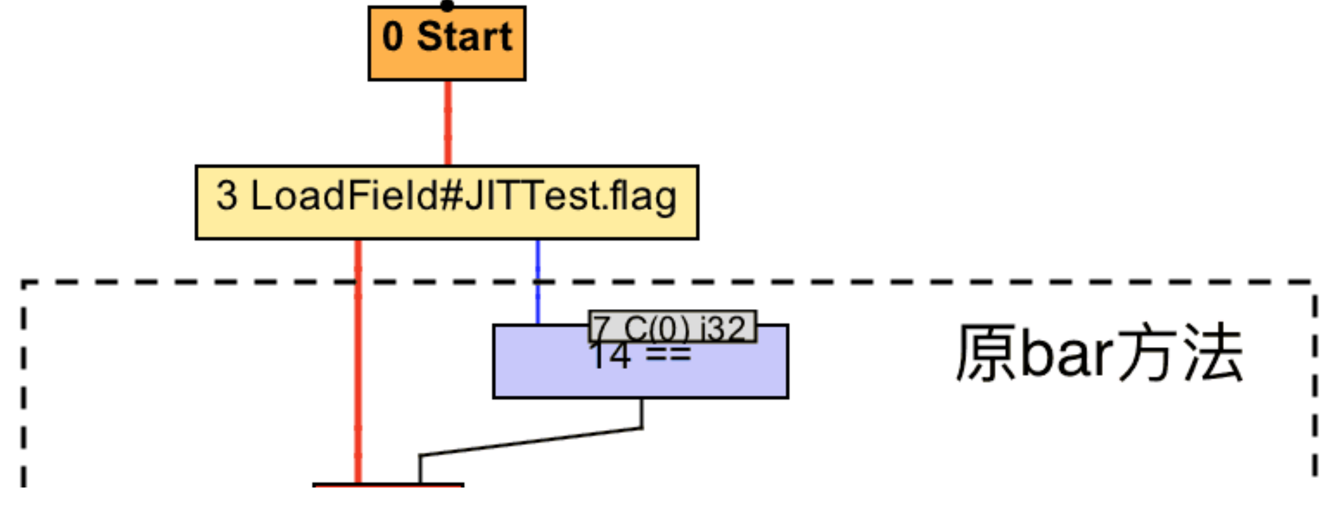
```
public static boolean flag = true;
public static int value0 = 0;
public static int value1 = 1;

public static int foo(int value) {
```

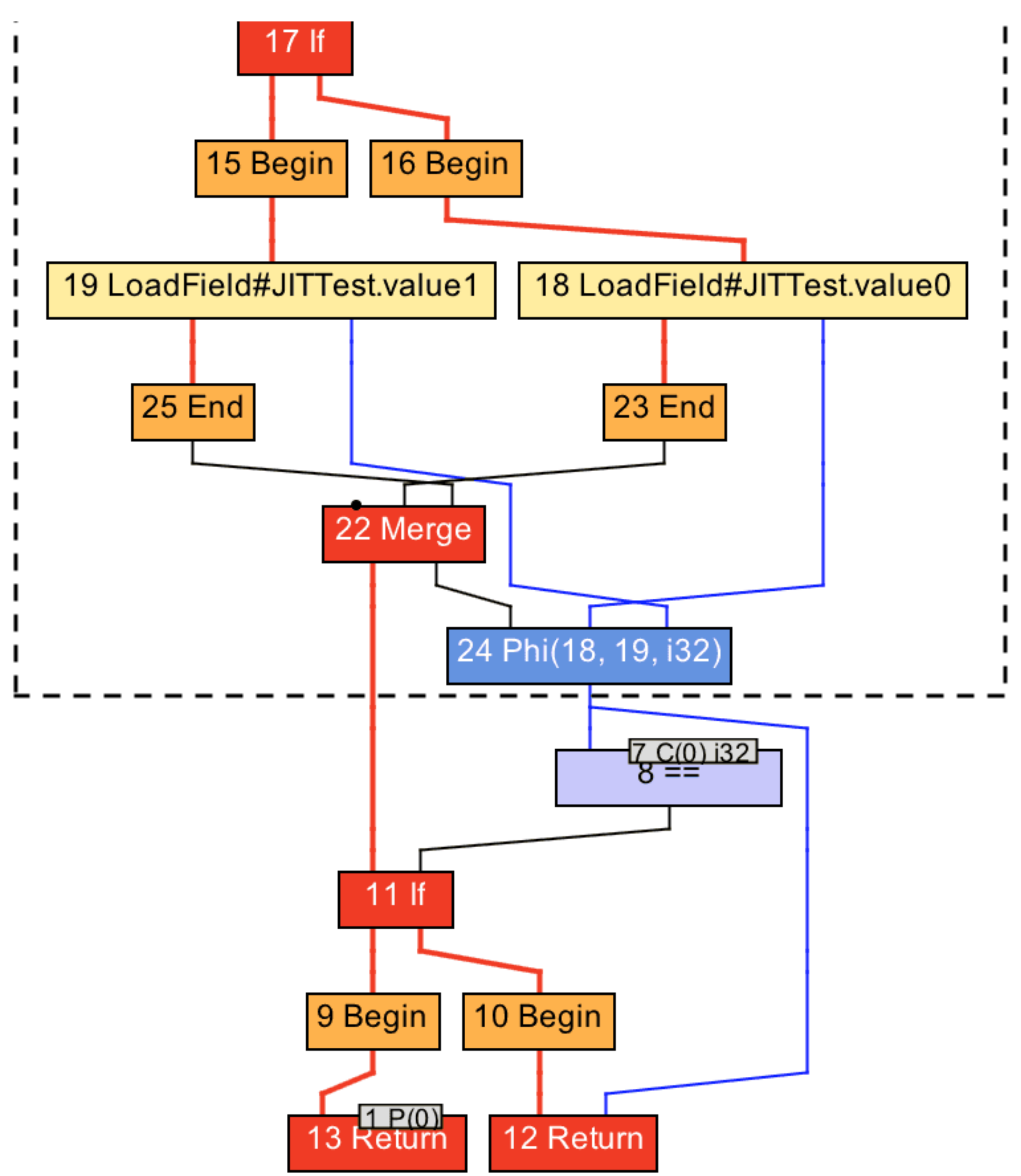
```
    int result = bar(flag);
    if (result != 0) {
        return result;
    } else {
        return value;
    }
}

public static int bar(boolean flag) {
    return flag ? value0 : value1;
}
```

来看一下 bar 方法的 IR 图：



内联后的 IR 图：





内联将被调用方法的 IR 图节点复制到调用者方法的 IR 图中。在这个例子中，bar 方法的 IR 图中的 0 号 Start 节点被复制到了 foo 方法的 IR 图中，从而避免了方法调用的开销。

逃逸分析

逃逸分析是 JIT 用于优化内存管理和同步操作的重要技术。通过分析对象是否逃逸到方法或线程的外部，编译器可以做出更智能的存储和同步决策。

逃逸分析通常是在方法内联的基础上进行的，JIT 可以根据逃逸分析的结果进行诸如锁消除、栈上分配以及标量替换的优化。

下面这段代码的就是对象未逃逸的例子：

```
public class Example {
    public static void main(String[] args) {
        example();
    }

    public static void example() {
        Foo foo = new Foo();
        Bar bar = new Bar();
        bar.setFoo(foo);
    }
}

class Foo {}

class Bar {
    private Foo foo;
    public void setFoo(Foo foo) {
        this.foo = foo;
    }
}
```

在这个例子中，`example` 方法创建了两个对象：`Foo` 和 `Bar`。然后，`Bar` 对象通过 `setFoo` 方法引用了 `Foo` 对象。

1. **Foo 对象的逃逸情况**：
    - `Foo` 对象被创建并传递给 `Bar` 对象的 `setFoo` 方法。
    - 一旦 `setFoo` 方法被调用，`Foo` 对象的引用存储在 `Bar` 对象的实例变量 `foo` 中。
    - 但是，`Bar` 对象本身在 `example` 方法结束后就不再使用。
    - 这意味着即使 `Foo` 对象的引用被存储在另一个对象中，但由于 `Bar` 对象本身也不会逃逸出 `example` 方法，因此 `Foo` 对象实际上也没有逃逸。
2. **Bar 对象的逃逸情况**：
    - `Bar` 对象在 `example` 方法中被创建并使用，但之后没有被传递到其他方法或返回。
    - 因此，`Bar` 对象也没有逃逸出 `example` 方法。

根据逃逸分析的结果，JIT 可能做出以下优化决策：

①、**锁消除**：如果 `Foo` 或 `Bar` 类中有同步块（使用 `synchronized`），由于对象没有逃逸，编译器可以安全地消除这些锁操作。

②、**栈上分配**：由于 `Foo` 和 `Bar` 对象都没有逃逸到方法之外，编译器可以选择在栈上分配这两个对象，而非在堆上分配。这样可以提高内存分配的效率，并减少垃圾收集器的压力。

> 堆和栈的区别可以查看这篇内容：JVM 的内存数据区

我们都知道 Java 的对象是在堆上分配的，而堆是对所有对象可见的。同时，JVM 需要对所分配的堆内存进行管理，并且在对象不再被引用时回收其所占据的内存。

如果逃逸分析能够证明某些新建的对象不逃逸，那么 JVM 完全可以将其分配至栈上，并且在 new 语句所在的方法退出时，通过弹出当前方法的栈桢来自动回收所分配的内存空间。

这样一来，我们便无须借助垃圾收集器来处理不再被引用的对象。

不过 Hotspot 并没有进行实际的栈上分配，而是使用了标量替换的技术。

③、**标量替换**（Scalar Replacement）

标量替换是一种优化技术，其中编译器将一个聚合对象分解为其各个字段。如果这个对象没有逃逸出方法，那么它的各个字段可以视为独立的局部变量。

这种技术允许编译器进行更细粒度的优化，如更好的寄存器分配和减少不必要的内存分配。

考虑这段代码：

```
public class Example {

    @AllArgsConstructor
    static class Cat {
        int age;
        int weight;
    }

    public static void example() {
        Cat cat = new Cat(1, 10);
        addAgeAndWeight(cat.age, cat.weight);
    }

    public static void addAgeAndWeight(int age, int weight) {
        // 对年龄和体重进行一些操作
    }

    public static void main(String[] args) {
        example();
    }
}
```

1. **对象的使用范围**：
    - 在 `example` 方法中创建了一个 `Cat` 对象，并将其字段 `age` 和 `weight` 传递给了 `addAgeAndWeight` 方法。
    - `Cat` 对象在 `example` 方法中创建且只在该方法中使用，没有被传递到方法外部或赋值给外部引用。
2. **逃逸分析**：
    - 由于 `Cat` 对象在方法外部没有引用，它没有逃逸出 `example` 方法的作用域。
    - 这意味着 `Cat` 对象是一个局部对象，适合进行标量替换。
3. **标量替换的应用**：
    - JVM 的 JIT 编译器会分析 `Cat` 对象的使用情况。基于逃逸分析，编译器可以决定不在堆上分配 `Cat` 对象，而是将其分解为两个独立的局部变量 `age` 和 `weight`。
    - 这样，原本由 `Cat` 对象占用的堆空间就被节省下来，而且减少了垃圾回收的压力。

4. **优化后的执行**：
    - 在执行 `example` 方法时，`Cat` 对象的字段 `age` 和 `weight` 直接作为栈上的局部变量处理，避免了堆分配。

标量替换后的伪代码如下所示：

```
public class Example {
    public static void example() {
        int catAge = 1;
        int catWeight = 10;
        addAgeAndWeight(catAge, catWeight);
    }

    public static void addAgeAndWeight(int age, int weight) {
        // 方法实现
    }
}
```

可以看到，`Cat` 对象被分解为两个局部变量 `catAge` 和 `catWeight`，并且直接作为参数传递给了 `addAgeAndWeight` 方法。

窥孔优化与寄存器分配

前面我们也简单分析了一下窥孔优化与寄存器分配，相信大家对这两个概念都有了一定的了解，这里简单总结下。

窥孔优化就是将编译器所生成的中间代码中的某些组合替换为效率更高的指令组，比如强度削减、常数合并等，看下面这个例子就是一个强度削减的例子：

```
y1=x1*3
经过强度削减后得到
y1=(x1<<1)+x1
```

编译器使用移位和加法削减乘法的强度，使用更高效率的指令组。

寄存器分配也是一种编译的优化手段，在 C2 编译器中普遍的使用。它是通过把频繁使用的变量保存在寄存器中，CPU 访问寄存器的速度比内存快得多，可以提升程序的运行速度。

经过寄存器分配和窥孔优化之后，程序就会被转换成机器码保存在 codeCache 中。

小结

本文主要介绍了 JIT 即时编译的原理以及编译优化的过程，包括：

- JIT 的触发条件
- JIT 的编译优化
- JIT 的编译器

JIT 是 JVM 的重要组成部分，它可以根据程序运行的情况，对热点代码进行编译优化，从而提升程序的运行效率。

参考链接：美团技术

第十四节：JVM 性能监控之命令行篇

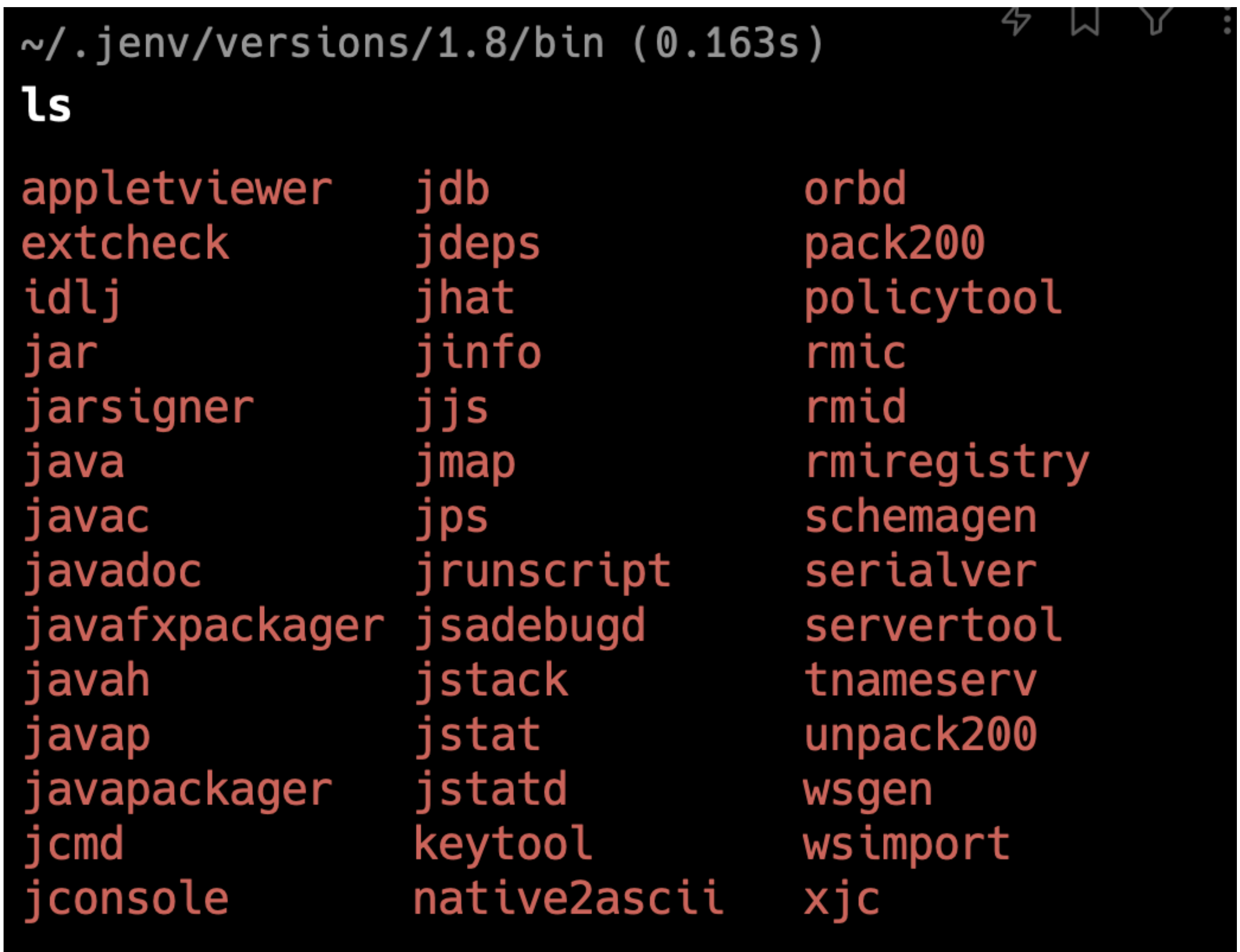
记得 2014 年我在写大宗期货交易平台的时候，遇到了一些棘手的问题，可能是因为我的并发编程知识掌握的不够扎实，导致出现了内存泄漏的问题。

当时排查了好久，用的工具就是 JDK 自带的 jconsole，之前也没有过类似的性能监控经验，就导致在查找问题的时候非常痛苦，至今印象深刻。

那今天我们就从工具篇出发，来看看这些命令行工具的具体使用方法，以及如何排查问题。

JDK 性能监控工具

除了我们的老朋友 java 和 javac 命令，在 Java 的 bin 目录下，还有很多其他的命令行工具，比如说用于性能监控的 jps、jstat、jinfo、jmap、jstack、jcmd 等等。



我用的 macOS，Windows 用户看到的可能是带有 .exe 结尾的，但是功能都是一样的，我就不再刻意去截图了。

接下来，我来给大家一一介绍一下这些工具的用途，认个脸熟。

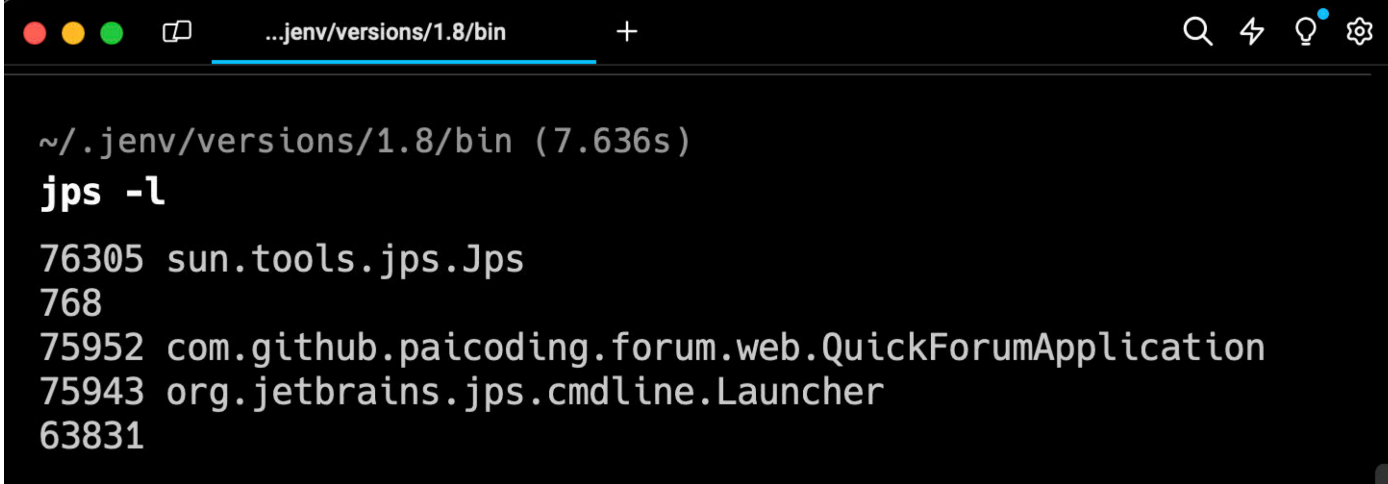
jps：查看虚拟机进程

jps（Java Virtual Machine Process Status Tool）类似 Linux 下的 ps，用于快速查看哪些 Java 应用正在运行，以及它们的进程 ID，这对于进一步使用其他 JVM 工具进行诊断是必要的。

jps 命令格式：

```
jps [ options ] [ hostid ]
```

jps 命令示例：



①、注意看第三个进程正是我本地运行着的技术派实战项目，一个前后端分离的 Spring Boot+React 的社区项目，帮助不少球友拿到了心仪的校招 Offer。

②、pid 是什么？pid 是进程 ID，是操作系统分配给进程的唯一标识符，可以用来查看进程的详细信息。

通常情况下，我们关闭一个进程可以通过右上角的 X 号来完成，但有了 pid，我们可以直接在命令行通过 kill 命令来关闭进程，比如：

```
kill -9 pid
```

意思是强制关闭 pid 对应的进程，新手可千万别在生产环境下乱 kill 哈（😂）。

再来看一下 jps 的常用选项：

选项列表	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() 方法的参数
-v	输出虚拟机进程启动时的 JVM 参数

jstat：查看 JVM 运行时信息

jstat（Java Virtual Machine Statistics Monitoring Tool）用于监控 JVM 的各种运行时状态信息，提供有关垃圾回收、类加载、JIT 编译等运行数据。

jstat 命令格式为：

```
jstat [ option vmid [interval[s|ms] [count]] ]
```

选项 option 主要分为三类：类加载、垃圾收集、运行期编译状况。

①、`-class`：监视类装载、卸载数量、总空间以及类装载所耗费的时间。

如下命令 `jstat -class -t 75952 1000 2` 会输出进程 75952 的类装载信息，每秒统计一次，一共输出两次。

```
~/.jenv/versions/1.8/bin (2.202s)
jstat -class -t 75952 1000 2
Timestamp        Loaded  Bytes  Unloaded  Bytes     Time
          730.4   14328 26949.9        31    48.4    45.55
          731.5   14328 26949.9        31    48.4    45.55
```

- Loaded：加载的类的数量。
- Bytes：所有加载类占用的空间大小。
- Unloaded：卸载的类的数量。
- Time：类加载器所花费的时间。

②、`-gc`：监视 Java 堆状况，包括 Eden 区、2 个 Survivor 区、老年代等容量、已用空间、GC 时间合计等信息。

如下命令 `jstat -gc 75952 1000 2` 会输出进程 75952 的 GC 信息，每秒统计一次，一共输出两次。结果比较多，我就截断折叠了一下，方便大家查看。

```
~/.jenv/versions/1.8/bin (1.271s)
jstat -gc 75952 1000 2
 S0C    S1C    S0U    S1U      EC       EU        OC
512.0  40960.0 352.0   0.0   946688.0 945424.2  288256.0
512.0  40960.0 352.0   0.0   946688.0 945424.2  288256.0

   OU        MC       MU      CCSC     CCSU     YGC
122610.5  78588.0  73752.4  10496.0  9549.8
122610.5  78588.0  73752.4  10496.0  9549.8

 YGCT     FGC     FGCT     GCT
 20     0.202    3       0.266     0.468
 20     0.202    3       0.266     0.468
```

- S0C, S1C, S0U, S1U：Survivor 区的大小和使用情况，一个 From 一个 To，C 为当前大小（Current），U 为已使用大小（Used）。
- EC, EU：Eden 区的大小和使用情况。
- OC, OU：老年代（Old）的大小和使用情况。
- MC, MU：元空间（Metaspace）的大小和使用情况。
- GC，GCC：GC 表示垃圾回收器进行 Minor GC（年轻代垃圾回收）的累计次数和总时间；GCC 表示垃圾回收器进行 Major GC（老年代垃圾回收，也称为 Full GC）的累计次数和总时间。

③、`-compiler`：监视 JIT 编译器编译过的方法、耗时等信息。

- Compiled：编译的方法数量。
- Failed：编译失败的方法数量。
- Invalid：失效的编译方法数量。
- Time：编译所花费的时间。

如下命令 `jstat -compiler 75952 1000 2` 会输出进程 75952 的编译信息，每秒统计一次，一共输出两次。

```
~/.jenv/versions/1.8/bin (1.991s)
jstat -compiler 75952 1000 2

Compiled Failed Invalid   Time   FailedType FailedMethod
    8445      0       0     1.80          0
    8445      0       0     1.80          0
```

好，我们再来总结一下 jstat 的主要选项，见下表：

选项列表	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、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

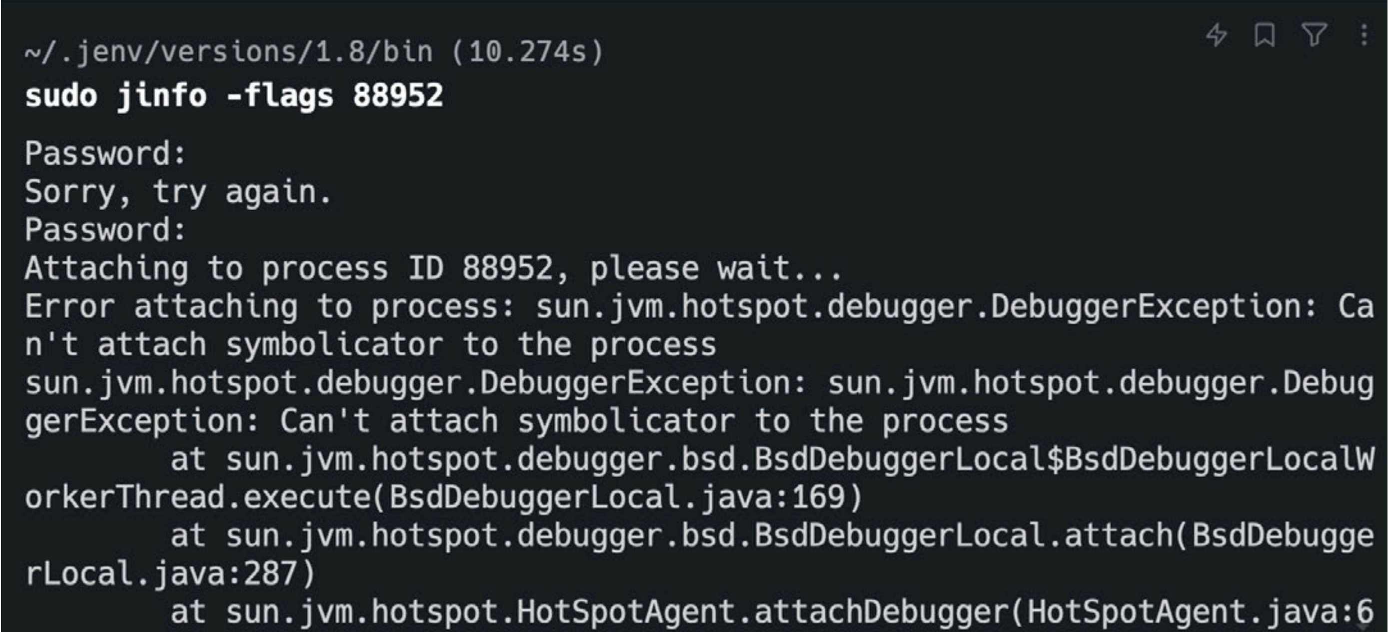
jinfo：查看虚拟机配置

jinfo（Configuration Info for Java）用于在补重启应用的情况下，调整虚拟机的各项参数，或者输出 Java 进程的详细信息。

jinfo 命令格式：

```
jinfo [ option ] pid
```

如下命令 `jinfo -flags 88952` 会输出进程 88952 的 JVM 参数信息。


```
~/.jenv/versions/1.8/bin (10.274s)
sudo jinfo -flags 88952

Password:
Sorry, try again.
Password:
Attaching to process ID 88952, please wait...
Error attaching to process: sun.jvm.hotspot.debugger.DebuggerException: Can't attach symbolicator to the process
sun.jvm.hotspot.debugger.DebuggerException: sun.jvm.hotspot.debugger.DebuggerException: Can't attach symbolicator to the process
        at sun.jvm.hotspot.debugger.bsd.BsdDebuggerLocal$BsdDebuggerLocalWorkerThread.execute(BsdDebuggerLocal.java:169)
        at sun.jvm.hotspot.debugger.bsd.BsdDebuggerLocal.attach(BsdDebuggerLocal.java:287)
        at sun.jvm.hotspot.HotSpotAgent.attachDebugger(HotSpotAgent.java:6
```

不过很遗憾的是，我的 macOS 系统上，jinfo 命令无法执行成功，后来经过各种实验找到了解决办法。

可能的原因是，我的 macOS 上装了太多的 JDK 版本，导致 Intellij IDEA 中编译的 JDK 和 jinfo 的版本不一致。

那怎么解决呢?

尝试方案 1：用相同的 JDK 版本编译运行 Java 程序，并使用相同的 JDK 的 jinfo 来查看。

```
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin (0.139s)
jenv which java
/Users/maweiqing/.jenv/versions/1.8/bin/java

/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin (0.086s)
cd /Users/maweiqing/.jenv/versions/1.8/bin/

~/.jenv/versions/1.8/bin (0.085s)
ls
appletviewer   javadoc         jdb      jrunscript    orbd          serialver
extcheck       javafxpackager  jdeps    jsadebugd     pack200       servertool
idlj           javah           jhat     jstack        policytool    tnameserv
jar            javap           jinfo    jstat         rmic          unpack200
jarsigner      javapackager    jjs      jstatd        rmid          wsgen
java           jcmd            jmap     keytool       rmiregistry   wsimport
javac          jconsole        jps      native2ascii  schemagen     xjc

~/.jenv/versions/1.8/bin (0.089s)
pwd
/Users/maweiqing/.jenv/versions/1.8/bin
```

结果依然报错，可能的原因是 JDK 版本过旧。

```
~/Documents/GitHub/HelloWorld git:(master) ±127 (0.226s)
/Users/maweiqing/.jenv/versions/1.8/bin/jps
768
3154 Example
5380 JDK8
63831
97195 JDK11Main
7422 Jps

~/Documents/GitHub/HelloWorld git:(master) ±127 (0.182s)
/Users/maweiqing/.jenv/versions/1.8/bin/jinfo 5380
Attaching to process ID 5380, please wait...
ERROR: attach: task_for_pid(5380) failed: '(os/kern) failure' (5)
Error attaching to process: sun.jvm.hotspot.debugger.DebuggerException: Can't attach to the process.
 Could be caused by an incorrect pid or lack of privileges.
sun.jvm.hotspot.debugger.DebuggerException: sun.jvm.hotspot.debugger.DebuggerException: Can't attach
 to the process. Could be caused by an incorrect pid or lack of privileges.
        at sun.jvm.hotspot.debugger.bsd.BsdDebuggerLocal$BsdDebuggerLocalWorkerThread.execute(BsdDeb
uggerLocal.java:169)
        at sun.jvm.hotspot.debugger.bsd.BsdDebuggerLocal.attach(BsdDebuggerLocal.java:287)
        at sun.jvm.hotspot.HotSpotAgent.attachDebugger(HotSpotAgent.java:671)
        at sun.jvm.hotspot.HotSpotAgent.setupDebuggerDarwin(HotSpotAgent.java:659)
        at sun.jvm.hotspot.HotSpotAgent.setupDebugger(HotSpotAgent.java:341)
        at sun.jvm.hotspot.HotSpotAgent.go(HotSpotAgent.java:304)
```

尝试方案 2： 用 JDK 11 来测试，代码用 JDK 11 编译和运行。

```
~/.jenv/versions/1.8/bin (0.121s)
export JAVA_HOME=$(/usr/libexec/java_home -v 11)
cd $JAVA_HOME

/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home (0.088s)
ls
README.html bin          conf         include      jmods        legal        lib          release

/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home (0.083s)
cd bin

/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin (0.084s)
pwd
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin

/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin (0.084s)
ls
jaotc        javadoc      jdeprscan    jinfo        jps          jstatd       rmiregistry
jar          javap        jdeps        jjs          jrunscript   keytool      serialver
jarsigner    jcmd         jfr          jlink        jshell       pack200      unpack200
java         jconsole     jhsdb        jmap         jstack       rmic
javac        jdb          jimage       jmod         jstat        rmid

/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin
```

然后用 JDK 11 的 jinfo 来查看，成功了。

```
~/Documents/GitHub/HelloWorld git:(master) ±127 (0.406s)
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/jinfo 10025

Java System Properties:
#Sat Jan 06 18:35:54 CST 2024
gopherProxySet=false
socksProxyHost=127.0.0.1
awt.toolkit=sun.lwawt.macosx.LWCToolkit
http.proxyHost=127.0.0.1
java.specification.version=11
sun.cpu.isalist=
sun.jnu.encoding=UTF-8
java.class.path=.
https.proxyPort=7890
java.vm.vendor=Oracle Corporation
sun.arch.data.model=64
java.vendor.url=https\://openjdk.java.net/
user.timezone=
java.vm.specification.version=11
os.name=Mac OS X
sun.java.launcher=SUN_STANDARD
user.country=US
sun.boot.library.path=/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/lib
sun.java.command=JDK11
http.nonProxyHosts=192.168.0.0/16|*.192.168.0.0/16|10.0.0.0/8|*.10.0.0.0/8|172.16.0.0/12|*.172.16
0/12|127.0.0.1|localhost|*.localhost|local|*.local|timestamp.apple.com|*.timestamp.apple.com|sequ
.apple.com|*.sequoia.apple.com|seed-sequoia.siri.apple.com|*.seed-sequoia.siri.apple.com
jdk.debug=release
sun.cpu.endian=little
user.home=/Users/maweiqing
user.language=zh
java.specification.vendor=Oracle Corporation
java.version.date=2020-07-14
java.home=/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home
file.separator=/
https.proxyHost=127.0.0.1
java.vm.compressedOopsMode=Zero based
```

再试一下 `jinfo -flags 10025` 命令，也 OK。

```
~/Documents/GitHub/HelloWorld git:(master) ±127 (0.263s)
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/jinfo -flags 10025

VM Flags:
-XX:CICompilerCount=4 -XX:ConcGCThreads=2 -XX:G1ConcRefinementThreads=8 -XX:G1HeapRegionSize=2097152
 -XX:GCDrainStackTargetSize=64 -XX:InitialHeapSize=536870912 -XX:MarkStackSize=4194304 -XX:MaxHeapSi
ze=8589934592 -XX:MaxNewSize=5152702464 -XX:MinHeapDeltaBytes=2097152 -XX:NonNMethodCodeHeapSize=583
6300 -XX:NonProfiledCodeHeapSize=122910970 -XX:ProfiledCodeHeapSize=122910970 -XX:ReservedCodeCacheS
ize=251658240 -XX:+SegmentedCodeCache -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+Us
eFastUnorderedTimeStamps -XX:+UseG1GC
```

之所以把这个问题的解决思路同步上来，也是希望能给球友们提供一些日常遇到开发问题时的解决思路。

jmap：导出堆快照

jmap 命令用于生成堆转储快照（一般称为 heap dump 或 dump 文件）。堆转储包含了 JVM 堆中所有对象的信息，包括类、属性、引用等。这对于分析内存泄漏和优化内存使用非常有帮助。

当然了，jmap 的作用不局限于此，它还可以查看堆的空间使用率、当前用的是哪种垃圾收集器等。

jmap 命令格式：

```
jmap [ option ] vmid
```

如下命令 `jmap -histo 10025` 会输出进程 10025 的堆内存中所有对象的数量和占用内存大小的汇总信息，按照内存使用量排序。

```
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/jmap -histo 10025
 num     #instances         #bytes  class name (module)
-------------------------------------------------------
   1:           871        2392376  [I (java.base@11.0.8)
   2:          9519         548304  [B (java.base@11.0.8)
   3:          8038         192912  java.lang.String (java.base@11.0.8)
   4:          4713         150816  java.util.HashMap$Node (java.base@11.0.8)
   5:          2111         121328  [Ljava.lang.Object; (java.base@11.0.8)
   6:           860         105264  java.lang.Class (java.base@11.0.8)
   7:           408          77576  [Ljava.util.HashMap$Node; (java.base@11.0.8)
   8:            45          56200  [C (java.base@11.0.8)
   9:          1294          41408  java.util.concurrent.ConcurrentHashMap$Node (java.base@11.0.8)
  10:           661          34616  [Ljava.lang.String; (java.base@11.0.8)
  11:           513          28728  jdk.internal.org.objectweb.asm.Item (java.base@11.0.8)
  12:           614          24560  java.util.LinkedHashMap$Entry (java.base@11.0.8)
  13:            57          18960  [Ljava.util.concurrent.ConcurrentHashMap$Node; (java.base@11.0.8)
  14:           358          17184  java.util.HashMap (java.base@11.0.8)
  15:            18          16800  [Ljdk.internal.org.objectweb.asm.Item; (java.base@11.0.8)
  16:           379          12128  java.lang.invoke.MethodType$ConcurrentWeakInternSet$WeakEntry (ja
va.base@11.0.8)
  17:           221          10608  java.lang.invoke.MemberName (java.base@11.0.8)
  18:           261          10440  java.lang.invoke.MethodType (java.base@11.0.8)
  19:           432          10368  java.lang.StringBuilder (java.base@11.0.8)
  20:            41           9184  jdk.internal.org.objectweb.asm.MethodWriter (java.base@11.0.8)
```

如下命令 `jmap -dump:format=b,file=heap.hprof 10025` 会输出进程 10025 的堆快照信息，保存到文件 heap.hprof 中。

```
~/Documents/GitHub/HelloWorld git:(master) ±127 (0.318s)
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/
jmap -dump:format=b,file=heap.hprof 10025
Heap dump file created

~/Documents/GitHub/HelloWorld git:(master) ±127 (0.083s)
ls
Example.class   HelloWorld.iml JDK11.java     JDK8.java      out
Example.java    JDK11.class    JDK8.class     heap.hprof     src
```

简单解释一下这条命令：

- format：文件格式，这里是 b，表示二进制格式。
- file：文件名。

那么，我们可以用什么工具来打开这个文件呢？后面会讲。我们先来看一下 jmap 的主要选项:

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
-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

jstack：跟踪Java堆栈

jstack 用于打印出 JVM 中某个进程或远程调试服务的线程堆栈信息（一般称为 threaddump 或者 javacore 文件）。它常用于诊断应用程序中的线程问题，比如线程死锁、死循环或长时间等待。

jstack 命令格式：

```
jstack [ option ] vmid
```

如下 `jstack -l 10025` 会输出进程 10025 的线程堆栈信息，包括锁信息。

```
~/Documents/GitHub/HelloWorld git:(master) ±127 (0.314s)
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/jstack -l 10025
2024-01-06 19:13:00
Full thread dump Java HotSpot(TM) 64-Bit Server VM (11.0.8+10-LTS mixed mode):

Threads class SMR info:
_java_thread_list=0x00006000003727e0, length=10, elements={
0x00007f8e05813800, 0x00007f8e08008800, 0x00007f8e07826800, 0x00007f8e0715e800,
0x00007f8e0715f800, 0x00007f8de580c800, 0x00007f8e0585b800, 0x00007f8e058cf000,
0x00007f8e08011000, 0x00007f8dd6828800
}

"main" #1 prio=5 os_prio=31 cpu=101.90ms elapsed=2263.80s tid=0x00007f8e05813800 nid=0x1703 runnable
 [0x000070000658b000]
   java.lang.Thread.State: RUNNABLE
        at java.io.FileInputStream.readBytes(java.base@11.0.8/Native Method)
        at java.io.FileInputStream.read(java.base@11.0.8/FileInputStream.java:279)
        at java.io.BufferedInputStream.read1(java.base@11.0.8/BufferedInputStream.java:290)
        at java.io.BufferedInputStream.read(java.base@11.0.8/BufferedInputStream.java:351)
        - locked <0x000000061fe0eb88> (a java.io.BufferedInputStream)
        at sun.nio.cs.StreamDecoder.readBytes(java.base@11.0.8/StreamDecoder.java:284)
        at sun.nio.cs.StreamDecoder.implRead(java.base@11.0.8/StreamDecoder.java:326)
        at sun.nio.cs.StreamDecoder.read(java.base@11.0.8/StreamDecoder.java:178)
        - locked <0x000000061ffc09f0> (a java.io.InputStreamReader)
        at java.io.InputStreamReader.read(java.base@11.0.8/InputStreamReader.java:185)
        at java.io.Reader.read(java.base@11.0.8/Reader.java:189)
        at java.util.Scanner.readInput(java.base@11.0.8/Scanner.java:882)
        at java.util.Scanner.next(java.base@11.0.8/Scanner.java:1476)
        at JDK11.main(JDK11.java:9)

   Locked ownable synchronizers:
        - None

"Reference Handler" #2 daemon prio=10 os_prio=31 cpu=0.11ms elapsed=2263.78s tid=0x00007f8e08008800 n
id=0x3503 waiting on condition  [0x0000700006ca0000]
   java.lang.Thread.State: RUNNABLE
        at java.lang.ref.Reference.waitForReferencePendingList(java.base@11.0.8/Native Method)
```

jstack 工具主要选项：

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

我们来通过一个线程死锁的问题，来看一下 jstack 的使用方法。

首先，我们编写一个死锁的程序：

```
class DeadLockDemo {
    private static final Object lock1 = new Object();
    private static final Object lock2 = new Object();

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (lock1) {
                System.out.println("线程1获取到了锁1");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (lock2) {
                    System.out.println("线程1获取到了锁2");
                }
            }
        }).start();

        new Thread(() -> {
            synchronized (lock2) {
                System.out.println("线程2获取到了锁2");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (lock1) {
                    System.out.println("线程2获取到了锁1");
                }
            }
        }).start();
    }
}
```

我们创建了两个线程，每个线程都试图按照不同的顺序获取两个锁（lock1 和 lock2）。这种锁的获取顺序不一致很容易导致死锁。

运行这段代码，果然卡住了。

```
~/Documents/GitHub/HelloWorld git:(master) ±127 (20.775s)
vim DeadLockDemo.java

~/Documents/GitHub/HelloWorld git:(master) ±127 (0.593s)
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/javac DeadLockDemo.java

~/Documents/GitHub/HelloWorld git:(master) ±127
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/java DeadLockDemo
线程1获取到了锁1
线程2获取到了锁2
```

运行 `jstack pid` 命令，可以看到死锁的线程信息。诚不欺我！

```
~/Documents/GitHub/HelloWorld git:(master) ±127 (0.369s)
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/jstack 21437

Found one Java-level deadlock:
=============================
"Thread-0":
  waiting to lock monitor 0x00007feb8000ff00 (object 0x000000061ff8cff8, a java.lang.Object),
  which is held by "Thread-1"
"Thread-1":
  waiting to lock monitor 0x00007feb8000fd00 (object 0x000000061ff8cfe8, a java.lang.Object),
  which is held by "Thread-0"

Java stack information for the threads listed above:
===================================================
"Thread-0":
        at DeadLockDemo.lambda$main$0(DeadLockDemo.java:15)
        - waiting to lock <0x000000061ff8cff8> (a java.lang.Object)
        - locked <0x000000061ff8cfe8> (a java.lang.Object)
        at DeadLockDemo$$Lambda$1/0x0000000800060840.run(Unknown Source)
        at java.lang.Thread.run(java.base@11.0.8/Thread.java:834)
"Thread-1":
        at DeadLockDemo.lambda$main$1(DeadLockDemo.java:29)
        - waiting to lock <0x000000061ff8cfe8> (a java.lang.Object)
        - locked <0x000000061ff8cff8> (a java.lang.Object)
        at DeadLockDemo$$Lambda$2/0x0000000800061040.run(Unknown Source)
        at java.lang.Thread.run(java.base@11.0.8/Thread.java:834)

Found 1 deadlock.
```

jcmd：多功能命令

jcmd 是一个多功能命令，可以用于收集堆转储、生成 JVM 和 Java 应用程序的性能数据，以及动态更改某些 Java 运行时参数。jcmd 提供的功能比其他单一命令，如 jstack, jmap, jstat 都要强大。

例如，使用 `jcmd -l` 列出当前的所有 Java 应用，和 jps 类似：

```
jcmd -l

10025 JDK11
38377 jdk.jcmd/sun.tools.jcmd.JCmd -l
97195 JDK11Main
```

例如，使用 `jcmd 10025 help` 查看进程 10025 支持的命令：

```
jcmd 10025 help

10025:
The following commands are available:
Compiler.CodeHeap_Analytics
Compiler.codecache
Compiler.codelist
Compiler.directives_add
Compiler.directives_clear
Compiler.directives_print
Compiler.directives_remove
Compiler.queue
GC.class_histogram
GC.class_stats
GC.finalizer_info
GC.heap_dump
GC.heap_info
GC.run
GC.run_finalization
JFR.check
JFR.configure
JFR.dump
JFR.start
JFR.stop
```

例如，使用 `jcmd 10025 VM.flags` 查看进程 10025 的 JVM 参数，相当于 `jinfo -flags 10025`：

```
~/Documents/GitHub/HelloWorld git:(master) ±127 (0.329s)
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/
jcmd 10025 VM.flags
10025:
-XX:CICompilerCount=4 -XX:ConcGCThreads=2 -XX:G1ConcRefinementThrea
ds=8 -XX:G1HeapRegionSize=2097152 -XX:GCDrainStackTargetSize=64 -XX
:InitialHeapSize=536870912 -XX:MarkStackSize=4194304 -XX:MaxHeapSiz
e=8589934592 -XX:MaxNewSize=5152702464 -XX:MinHeapDeltaBytes=209715
2 -XX:NonNMethodCodeHeapSize=5836300 -XX:NonProfiledCodeHeapSize=12
2910970 -XX:ProfiledCodeHeapSize=122910970 -XX:ReservedCodeCacheSiz
e=251658240 -XX:+SegmentedCodeCache -XX:+UseCompressedClassPointers
 -XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps -XX:+UseG1G
C
```

例如，使用 `jcmd 10025 Thread.print` 查看进程 10025 的线程信息，相当于 `jstack 10025`：

```
~/Documents/GitHub/HelloWorld git:(master) ±127 (0.288s)
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/
jcmd 10025 Thread.print
10025:
2024-01-06 21:03:31
Full thread dump Java HotSpot(TM) 64-Bit Server VM (11.0.8+10-LTS m
ixed mode):

Threads class SMR info:
_java_thread_list=0x00006000003727e0, length=10, elements={
0x00007f8e05813800, 0x00007f8e08008800, 0x00007f8e07826800, 0x00007
f8e0715e800,
0x00007f8e0715f800, 0x00007f8de580c800, 0x00007f8e0585b800, 0x00007
f8e058cf000,
0x00007f8e08011000, 0x00007f8dd6828800
}

"main" #1 prio=5 os_prio=31 cpu=101.90ms elapsed=8895.01s tid=0x000
07f8e05813800 nid=0x1703 runnable  [0x000070000658b000]
   java.lang.Thread.State: RUNNABLE
        at java.io.FileInputStream.readBytes(java.base@11.0.8/Nativ
e Method)
        at java.io.FileInputStream.read(java.base@11.0.8/FileInputS
tream.java:279)
        at java.io.BufferedInputStream.read1(java.base@11.0.8/Buffe
```

jmcd 命令格式：

```
jcmd <pid | main class> <command ... | PerfCounter.print | -f  file>
```

jmcd 的主要选项：

选项	描述	补充
help	打印帮助信息，示例：jcmd help []	无
ManagementAgent.stop	停止 JMX Agent	无
ManagementAgent.start_local	开启本地 JMX Agent	无
ManagementAgent.start	开启 JMX Agent	无
Thread.print	参数-l 打印 java.util.concurrent 锁信息，相当于：jstack	无
PerfCounter.print	相当于：jstat -J-Djstat.showUnsupported=true -snap	无
GC.class_histogram	相当于：jmap -histo	无
GC.heap_dump	相当于：jmap -dump:format=b,file=xxx.bin	无
GC.run_finalization	相当于：System.runFinalization()	无
GC.run	相当于：System.gc()	无
VM.uptime	参数-date 打印当前时间，VM 启动到现在的时候，以秒为单位显示	无
VM.flags	参数-all 输出全部，相当于：jinfo -flags , jinfo -flag	无
VM.system_properties	相当于：jinfo -sysprops	无
VM.command_line	相当于：jinfo -sysprops	grep command
VM.version	相当于：jinfo -sysprops	grep version

操作系统工具

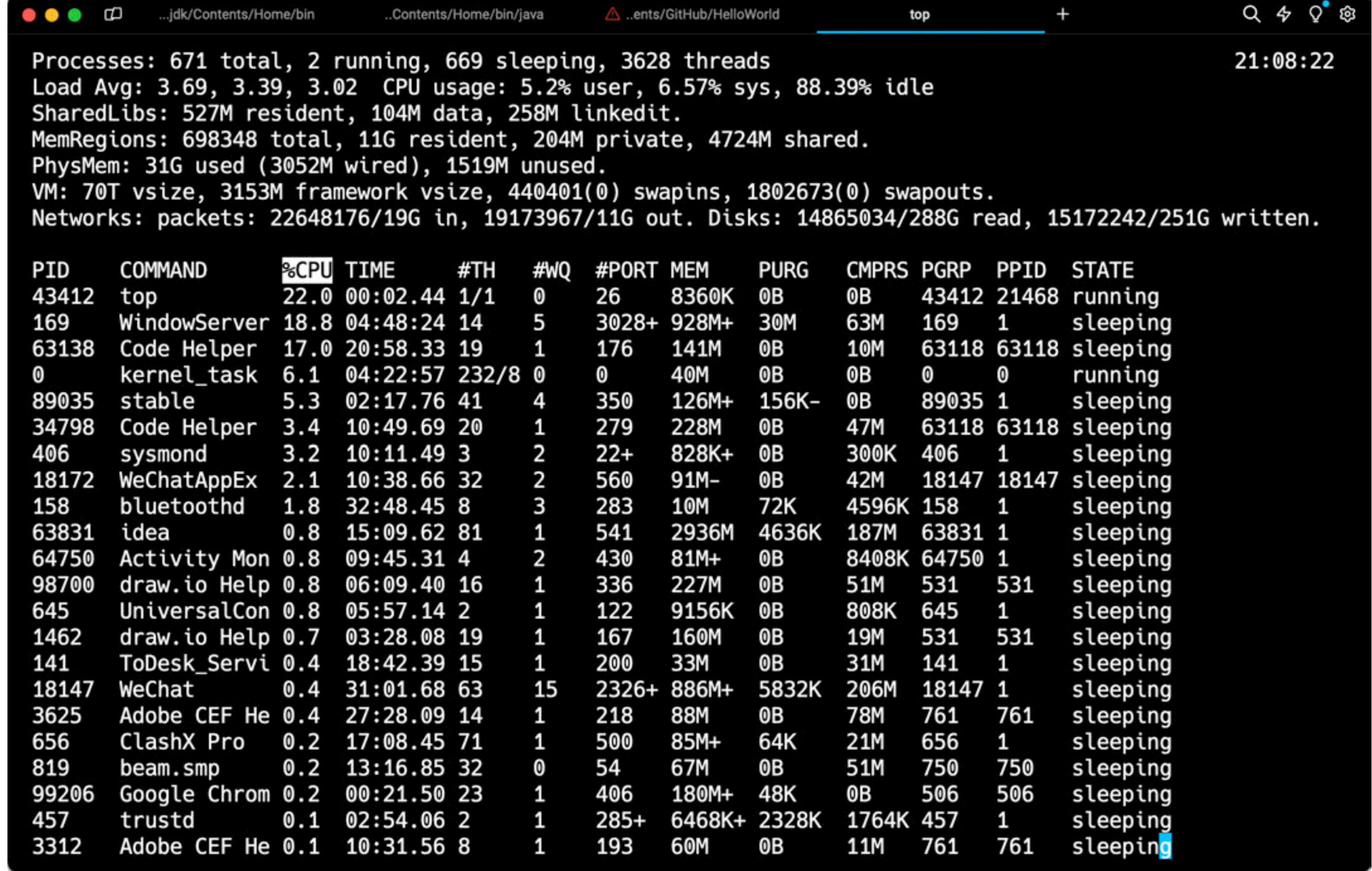
除了 JDK 自带的命令行，我们很多时候还要使用操作系统为我们提供的命令行工具，来完成性能监控的监测。

比如说 top、vmstat、iostat、netstat 等等。

top：显示系统整体资源使用情况

top 命令用于实时显示系统中各个进程的资源占用情况，如 CPU 和内存使用率。常用于快速查看哪些进程占用了较高的资源。

该命令的输出结果是实时变化的，可以使用 `ctrl + c` 来退出。下图是我的 macOS 上输出的结果：



top 命令的输出可以分为两个部分：前半部分是系统统计信息，后半部分是进程信息。

统计信息

统计信息是针对整个系统的，主要包括系统负载、CPU 使用率、内存使用情况、虚拟内存使用情况、网络和硬盘使用情况等。

- 第 1 行是进程和线程信息，分别表示总进程数、正在运行的进程数、睡眠的进程数、线程数。
- 第 2 行是负载均衡和 CPU 使用率信息，`Load Avg: 4.02, 3.89, 3.29`：这表示过去 1 分钟、5 分钟和 15 分钟的平均系统负载。负载大于 3 意味着系统相对繁忙；`CPU usage: 6.97% user, 3.54% sys, 89.47% idle`：用户占用了 6.97% 的 CPU，系统占用了 3.54%，还有 89.47% 的 CPU 处于空闲。
- 第 3 行是共享库（Shared Libraries）内存使用的信息。这一行的数据主要涉及到操作系统加载的共享库（如动态链接库或共享对象文件）。
- 第 4 行是内存区域（Memory Regions）的使用信息。内存区域是指操作系统为应用程序和进程分配的内存块。每个内存区域都有特定的用途和属性，比如代码、数据、堆、栈等。这一行的数据提供了系统内存使用的更详细的视图。
- 第 5 行是内存使用情况，`PhysMem: 30G used (3018M wired), 1547M unused`：内存总共使用了 30GB，还有大约 1547MB 的内存未使用；
- 第 6 行是虚拟内存的信息，虚拟内存是计算机内存管理的一种技术，它为每个程序提供一种“虚拟”的地址空间，这些地址空间对于每个程序来说都是连续的，但实际上可能分散在物理内存和磁盘的交换空间（swap space）上。
- 第 7 行是网络和硬盘信息，`Networks: packets: 22655692/19G in, 19180791/11G out`：网络接收了 19GB 的数据包；发送了约 11GB 的数据包；`Disks: 14866544/288G read, 15176739/251G written`：硬盘读取 14866544 次；写入了约 15176739 次。

进程信息

在进程信息区中，显示了系统各个进程的资源使用情况。主要字段的含义：

- PID：进程 id
- COMMAND：命令名/命令行
- %CPU：进程占用的 CPU 使用率
- TIME：进程使用的 CPU 时间总计，单位 1/100 秒
- MEM：进程使用的物理内存和虚拟内存大小，单位 KB

Windows 用户可以使用 tasklist 命令来查看进程信息。

vmstat：监控内存和 CPU

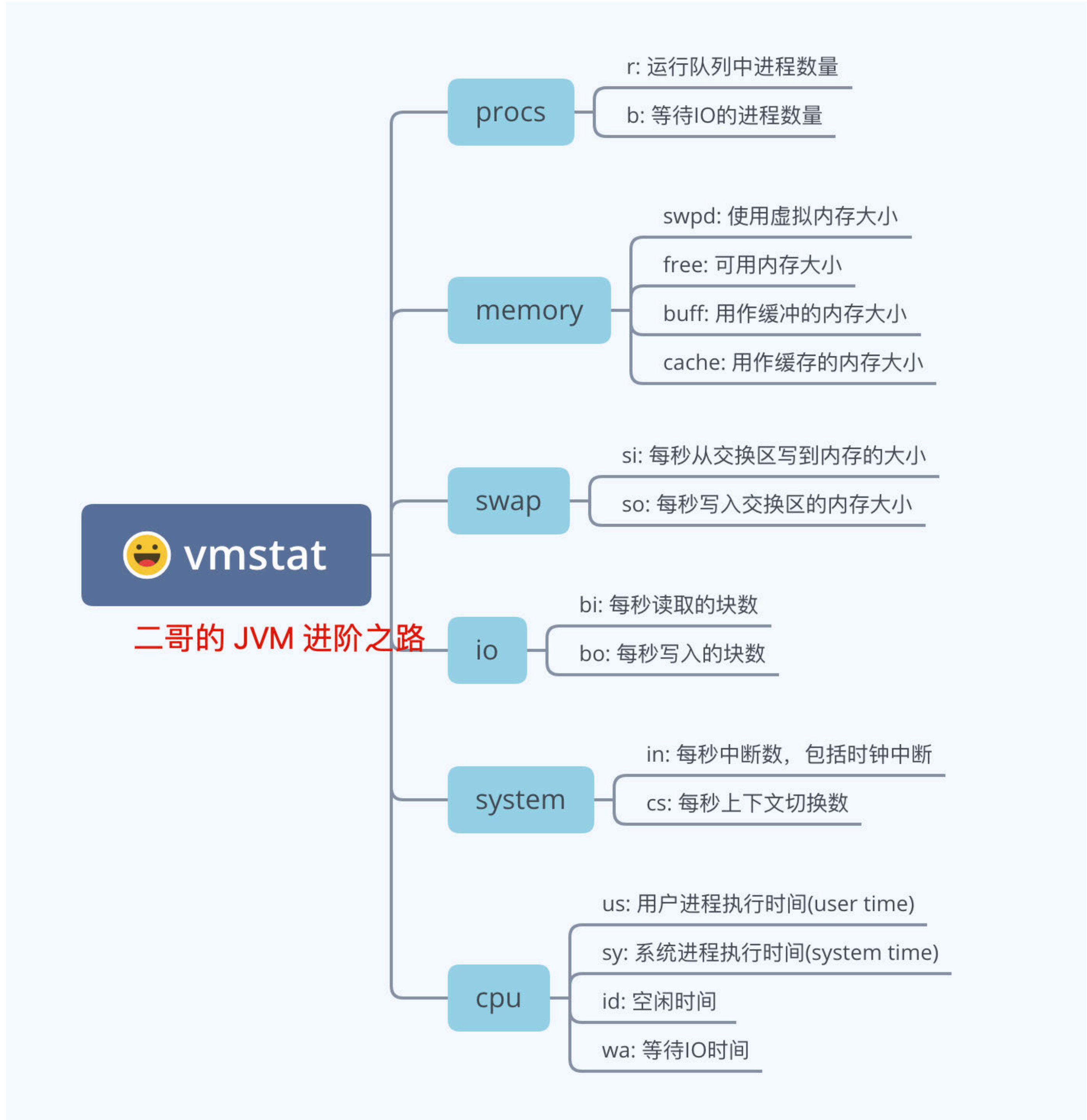
vmstat 是 Linux 上的一款功能比较齐全的性能监测工具。它可以统计 CPU、内存、swap 的使用情况。

一般 vmstat 工具的使用是通过两个数字参数来完成的，第一个参数是采样的时间间隔数，单位是秒，第二个参数是采样的次数，如:

```
[illegible]:~# vmstat 1 3
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 204664 164148 916204    0    0     0     9    0    0  1  0 99  0  0
 0  0      0 204664 164148 916240    0    0     0     0  267  506  0  0 100  0  0
 0  0      0 204664 164148 916240    0    0     0     0  256  499  0  0 100  0  0
```

`vmstat 1 3` 命令表示每秒采样一次，共三次。

输出的各个列的含义：



vmstat 的用法如下：

```
vmstat [options] [delay [count]]
```

vmstat 的主要选项：

- `[options]`：提供不同的输出选项，例如 -a 显示活跃和非活跃内存，-d 显示磁盘统计，-s 显示内存统计等。
- `[delay]`：在连续模式下，两次报告之间的延迟时间（秒）。
- `[count]`：要显示的报告数量。

iostat：监控 IO 使用

iostat 用于统计 CPU 使用信息和磁盘的 IO 信息。

```
[illegible]:~# iostat
Linux 4.15.0-128-generic (iZ2zebrh6ffesjwecx7eq9Z)      01/06/2024      _x86_64_        (2 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.52    0.00    0.16    0.01    0.00   99.31

Device             tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
vda               1.46         0.42        16.97   18047431  729014448
```

基本用法如下：

```
iostat [options] [interval [count]]
```

- `[options]`：提供不同的输出选项。例如，-c 显示 CPU 使用情况，-d 显示磁盘使用情况，-x 显示扩展统计信息等。
- `[interval]`：报告之间的延迟时间（秒）。
- `[count]`：显示报告的次数。

`iostat` 的输出包括两个主要部分：

1. **CPU 使用情况**：
    - `user`：用户程序使用的 CPU 时间百分比。
    - `system`：系统（内核）级程序使用的 CPU 时间百分比。
    - `idle`：CPU 空闲时间百分比。
2. **磁盘 I/O 统计**：
    - `tps`（Transfers Per Second）：每秒传输次数。
    - `kB_read/s`：每秒读取的千字节数。
    - `kB_wrtn/s`：每秒写入的千字节数。
    - `kB_read` 和 `kB_wrtn`：分别是读取和写入的总千字节数。

    如果使用 `-x` 选项，会显示更详细的统计信息，例如：

    - `%util`：表示磁盘的繁忙程度。
    - `await`：I/O 请求的平均等待时间（毫秒）。
    - `svctm`：服务时间，即完成一个 I/O 请求所需的平均时间。

使用示例如下：

①、查看 CPU 和所有磁盘设备的基本 I/O 统计信息：

```
iostat
```

②、查看磁盘 I/O 统计信息，每 2 秒更新一次：

```
iostat -dx 2
```

③、只查看 CPU 使用情况：

```
iostat -c
```

netstat：监控网络使用

`netstat`（network statistics）用于监控和显示网络相关信息。基本用法如下：

```
netstat [options]
```

- `[options]`：提供不同的输出选项。常见的选项包括 `-a`（显示所有连接和侦听端口），`-t`（显示 TCP 连接），`-u`（显示 UDP 连接），`-n`（以数字形式显示地址和端口号），`-r`（显示路由表）等。

```
netstat -a
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  192.168.1.3.62537      ecs-139-9-45-78..talar ESTABLISHED
tcp4       0      0  localhost.7890         localhost.62535        ESTABLISHED
tcp4       0      0  localhost.62535        localhost.7890         ESTABLISHED
tcp4       0      0  192.168.1.3.62512      ecs-139-9-45-78..talar ESTABLISHED
tcp4       0      0  localhost.7890         localhost.62511        ESTABLISHED
tcp4       0      0  localhost.62511        localhost.7890         ESTABLISHED
tcp4       0      0  localhost.7890         localhost.62433        ESTABLISHED
```

`netstat` 的输出通常包括以下几个方面的信息：

①、**网络连接**：显示活动的或监听的套接字连接，包括服务名、本地地址和端口、远程地址和端口、连接状态等。

②、**路由表**：显示网络路由表，包括目的地址、网关、子网掩码、使用的接口等。

小结

今天我们介绍了 JDK 自带的性能监控工具，以及操作系统提供的一些命令行工具。

这些工具在排查问题时非常有用，希望大家一定要掌握，以备不时之需。

参考

- 星球嘉宾三分恶：[JVM性能监控命令行篇]

第十五节：JVM 性能监控之可视化篇

前面我们已经讲了 [JVM 性能监控工具之命令行篇]，本篇我们来介绍一些可视化的性能监控工具，包括 JConsole、VisualVM、Java Mission Control 等，阿里的 Arthas 我们留到后面单独去讲。

可视化工具比命令行工具强大的地方就在于这些工具提供了更直观、更易于理解的性能数据视图，肉眼看上去，脑子就能快速 get 到问题所在，那这篇就来带大家看看这些工具的强大之处。

JConsole

JConsole（Java Monitoring and Management Console），是一款基于 JMX（Java Manage-ment Extensions）的可视化监视管理工具。

> JMX 的全称是 Java Management Extensions，翻译过来就是 Java 管理扩展，既是 Java 管理系统中的一个标准，一个规范，也是一个接口，一个框架。JConsole 就相当于是 JMX 的一个实现类。

JConsole 可以用来监视 Java 应用程序的运行状态，包括内存使用、线程状态、类加载、GC 等，还可以进行一些基本的性能分析。

JConsole 连接 Java 程序

JConsole 程序位于%JAVA_HOME%bin 目录下，不过我当前的操作系统是 macOS，和 Windows 有一些不一样，我就不再刻意截 Windows 的图了，希望大家可以理解。

```
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin (0.092s)
ls

jaotc        javadoc      jdeprscan    jinfo        jps          jstatd       rmiregistry
jar          javap        jdeps        jjs          jrunscript   keytool      serialver
jarsigner    jcmd         jfr          jlink        jshell       pack200      unpack200
java         jconsole     jhsdb        jmap         jstack       rmic
javac        jdb          jimage       jmod         jstat        rmid
```

直接启动 JConsole，会弹出一个窗口，显示本机正在运行的 Java 程序，选择一个程序（比如说技术派的 28966），点击 `连接` 即可。

JConsole: 新建连接

还可以进行远程链接，比如说对服务器上的 Java 程序进行监控，需要远程服务器上的 Java 程序在启动的时候加上以下这些参数：

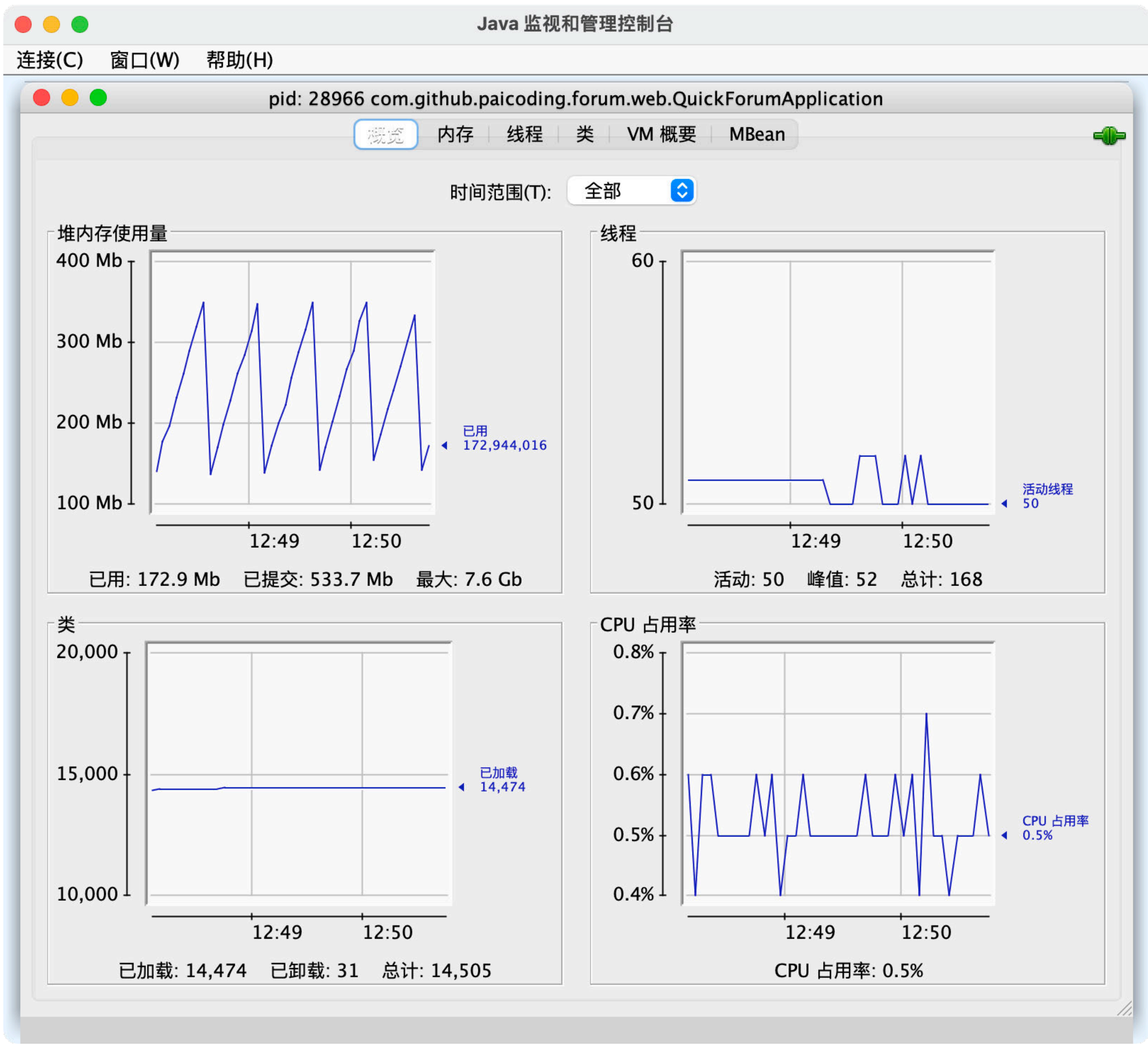
```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=<PORT>
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

- `<PORT>` 是您想要 JMX 代理监听的端口号，例如 9999。

- authenticate=false 表示不需要身份验证来连接到 JMX（注意：这在生产环境中可能不安全）。
- ssl=false 表示不需要使用 SSL 加密连接（同样，这在生产环境中可能不安全）。

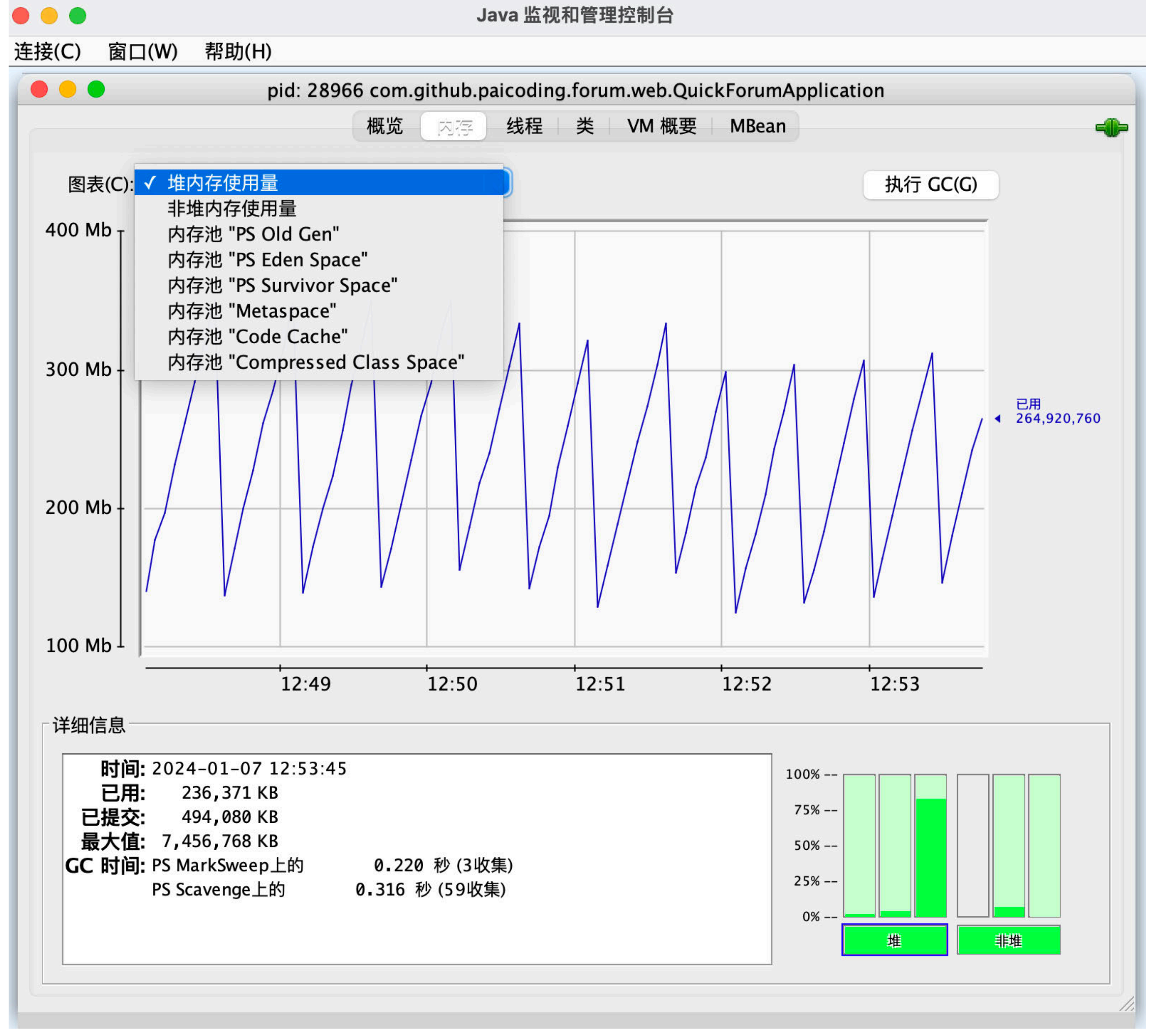
Java 程序概况

使用 JConsole 连接了一个本地程序，在`概述`可以看到 Java 程序的运行时概况，包括`堆内存使用量`、`线程`、`类`、`CPU占用率`四项信息的曲线图。



内存监控

`内存`的作用相当于可视化的 jstat 命令（上一节讲过了），用于监视 Java 堆的使用情况，可以更细化到 eden 区、suvivior 区、老年代的使用情况。



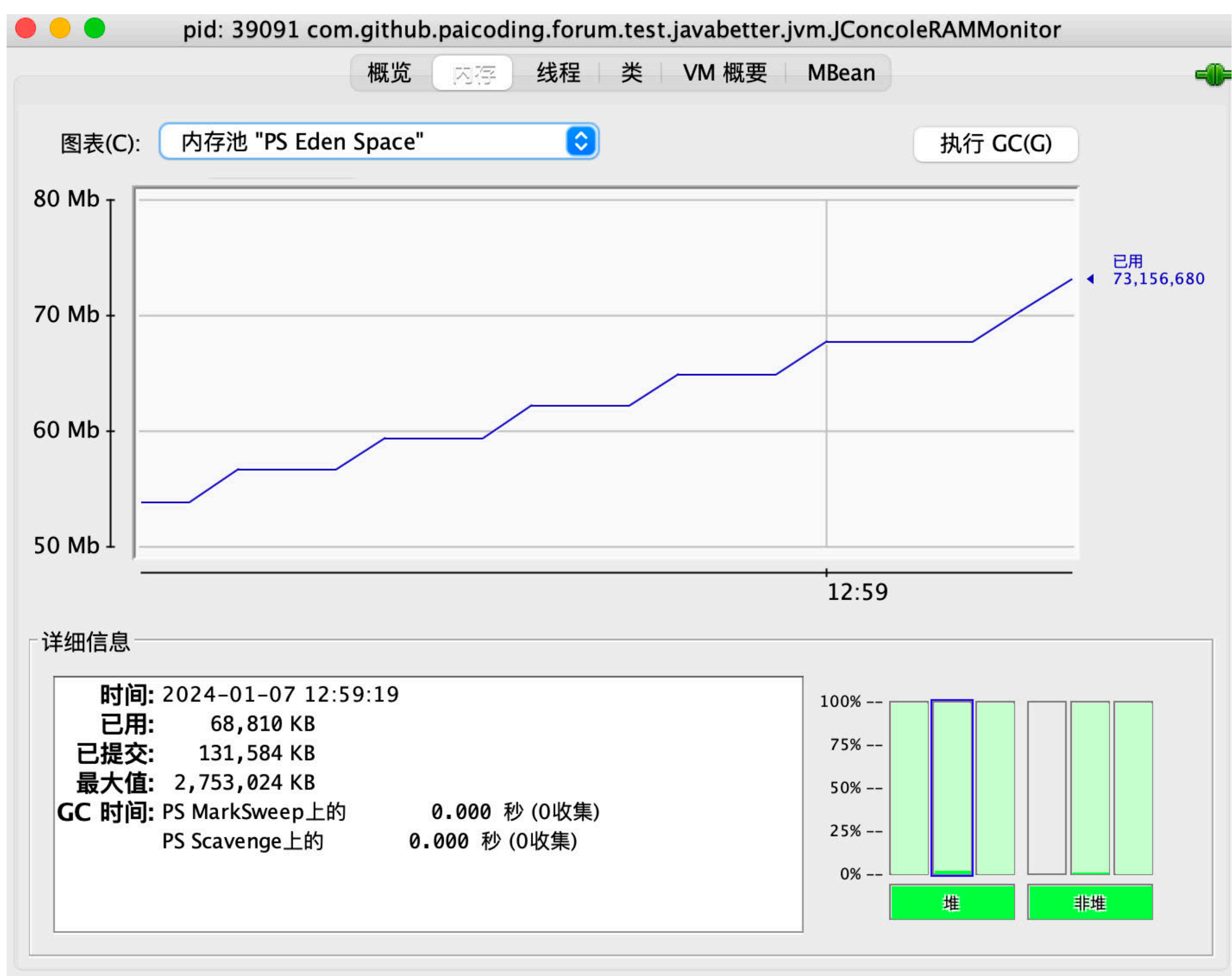
为了更加清晰地查看内存变化，可以运行下面这段，然后连接：

```
/**
 * VM参数： -Xms100m -Xmx100m -XX:+UseSerialGC
 */
class JConcoleRAMMonitor {

    /***
     * 内存占位符对象，一个OOMObject大约占64KB
     */
    static class OOMObject {
        public byte[] placeholder = new byte[64 * 1024];
    }

    public static void fillHeap(int num) throws InterruptedException {
        List<OOMObject> list = new ArrayList<OOMObject>();
```

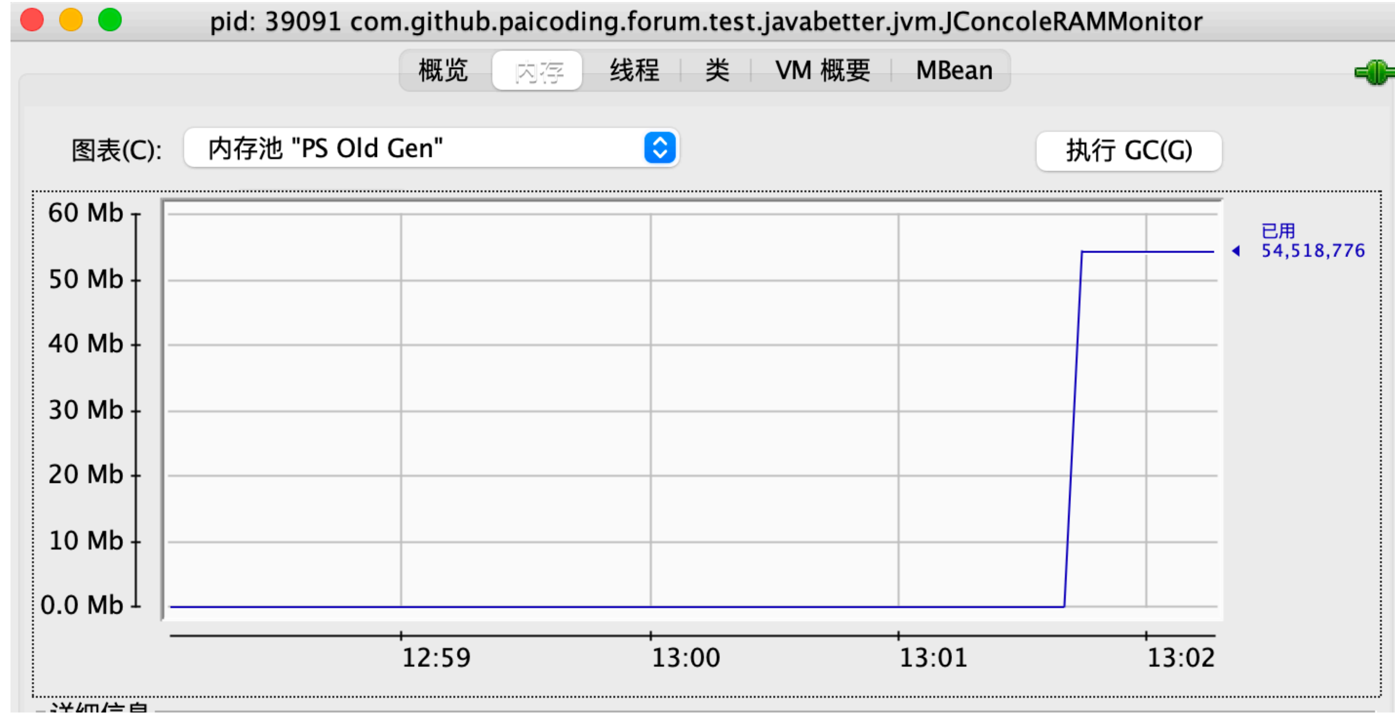
```
        for (int i = 0; i < num; i++) {
            // 稍作延时，令监视曲线的变化更加明显
            Thread.sleep(300);
            list.add(new OOMObject());
        }
        System.gc();
    }

    public static void main(String[] args) throws Exception {
        fillHeap(2000);
    }
}
```

这段代码的作用是以 64KB/50ms 的速度向 Java 堆中填充数据，一共填充 1000 次。

观察 Eden 区的运行趋势，发现呈折线趋势增长。



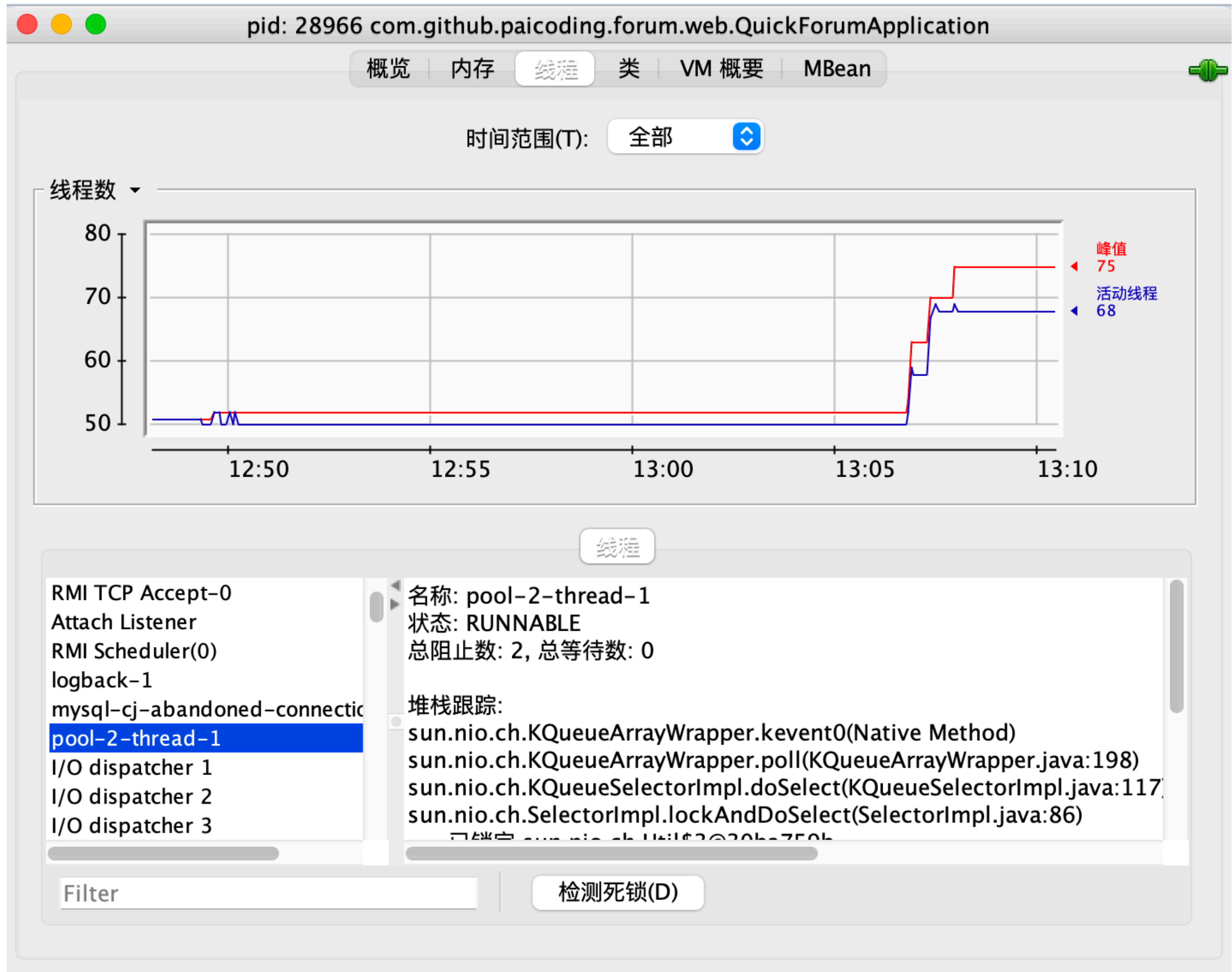
点击「执行 GC」之后，会发现老年代的柱状图会达到峰值状态，是因为执行 GC 之后，Eden 区的对象被回收，存活的对象被移动到老年代。



线程监控

JConcole 还可以监控线程，相当于可视化的 jstack 命令（上一节讲过了）。

如下图，JConcole 显示了应用系统内的线程数量，左下方显示了程序中所有的线程。点击线程名称，就可以查看线程的栈信息。

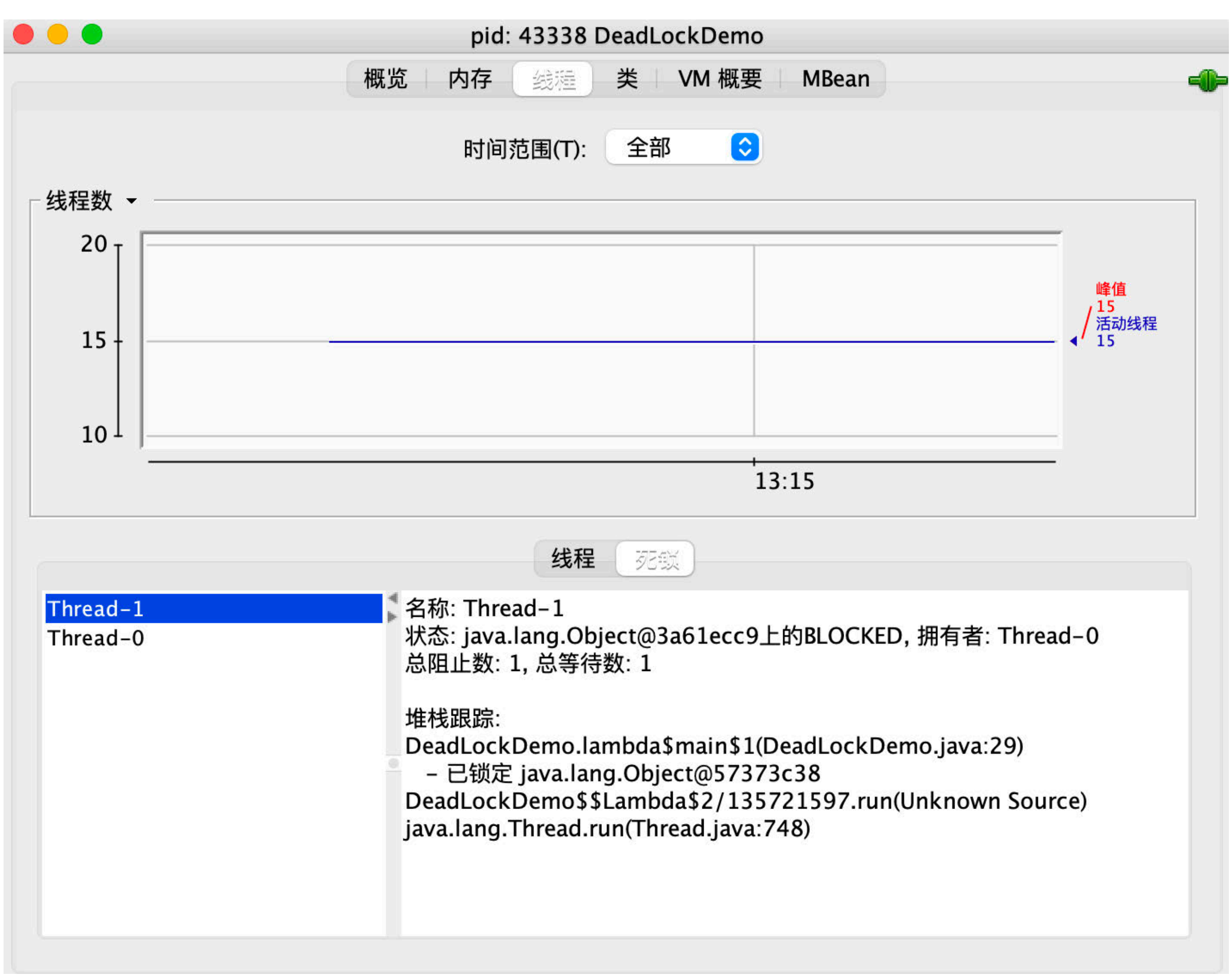


使用 JConsole 还可以快速定位死锁问题。上一篇我们曾写过一个死锁的例子，这里我们再来看一下。

```java
class DeadLockDemo {
    private static final Object lock1 = new Object();
    private static final Object lock2 = new Object();

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (lock1) {
                System.out.println("线程1获取到了锁1");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (lock2) {
                    System.out.println("线程1获取到了锁2");
                }
            }
        }).start();
```

```
        new Thread(() -> {
            synchronized (lock2) {
                System.out.println("线程2获取到了锁2");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (lock1) {
                    System.out.println("线程2获取到了锁1");
                }
            }
        }).start();
    }
}
```

运行以上代码，点击 JConsole 线程面板下的 「检测到死锁」按钮，将会看到线程的死锁信息。



类加载情况

如下图，「类」面板显示了已经装载的类数量。在详细信息栏中，还显示了已经卸载的类的数量。

VM 概要

在`VM 概要`面板，JConsole 显示了当前应用程序的运行时环境，包括虚拟机类型、版本、堆信息以及虚拟机参数等。

pid: 28966 com.github.paicoding.forum.web.QuickForumApplication

概览 | 内存 | 线程 | 类 | VM 概要 | MBean

VM 概要
2024年1月7日 星期日 下午01时17分36秒 CST

连接名称: pid: 28966 com.github.paicoding.forum.web.QuickForumApplication
虚拟机: Java HotSpot(TM) 64-Bit Server VM版本 25.131-b11
供应商: Oracle Corporation
名称: 28966@itwanger.local
运行时间: 57 分钟
进程 CPU 时间: 2 分钟
JIT 编译器: HotSpot
总编译时间: 2.358 秒

活动线程: 68
峰值: 75
守护程序线程: 47
启动的线程总数: 215
已加装当前类: 16,839
已加载类总数: 16,870
已卸载类总数: 31

当前堆大小: 192,880 KB
最大堆大小: 7,456,768 KB
提交的内存: 361,472 K
暂挂最终处理: 0对象
垃圾收集器: 名称 = 'PS MarkSweep', 收集 = 3, 总花费时间 = 0.220 秒
垃圾收集器: 名称 = 'PS Scavenge', 收集 = 178, 总花费时间 = 0.779 秒

操作系统: Mac OS X 10.16
体系结构: x86_64
处理程序数: 8
提交的虚拟内存: 45,489,744 KB
总物理内存: 33,554,43
空闲物理内存: 847,89
总交换空间: 1,572,86
空闲交换空间: 1,187,84

VM 参数: -agentlib:jdwp=transport=dt_socket,address=127.0.0.1:59448,suspend=y,server=n -Dvisualgc.id=

VisualVM

VisualVM（All-in-One Java Troubleshooting Tool）一款功能强大的运行监视和故障处理工具之一，在很长一段时间内，VisualVM 都是 Oracle 官方主推的故障处理工具。

集成了多个 JDK 命令行工具的功能，提供了一个友好的图形界面，非常适用于开发和生产环境。

VisualVM 安装插件

VisualVM 的安装非常简单，下载地址：https://visualvm.github.io

VisualVM Home Features Download Plugins Documentation | Issues Feedback Sources

Download

VisualVM is currently only distributed as a standalone tool at GitHub. Be sure to download it from this page to get the latest features and bugfixes.

VisualVM was previously distributed also in GraalVM 19~23.0 and Oracle JDK 6~8. See the Upgrading VisualVM from GraalVM and Upgrading Java VisualVM documents to learn how to upgrade to the latest VisualVM.

VisualVM 2.1.7 (.zip, 19.5 MB)
macOS Application Bundle (.dmg, 21.3 MB)

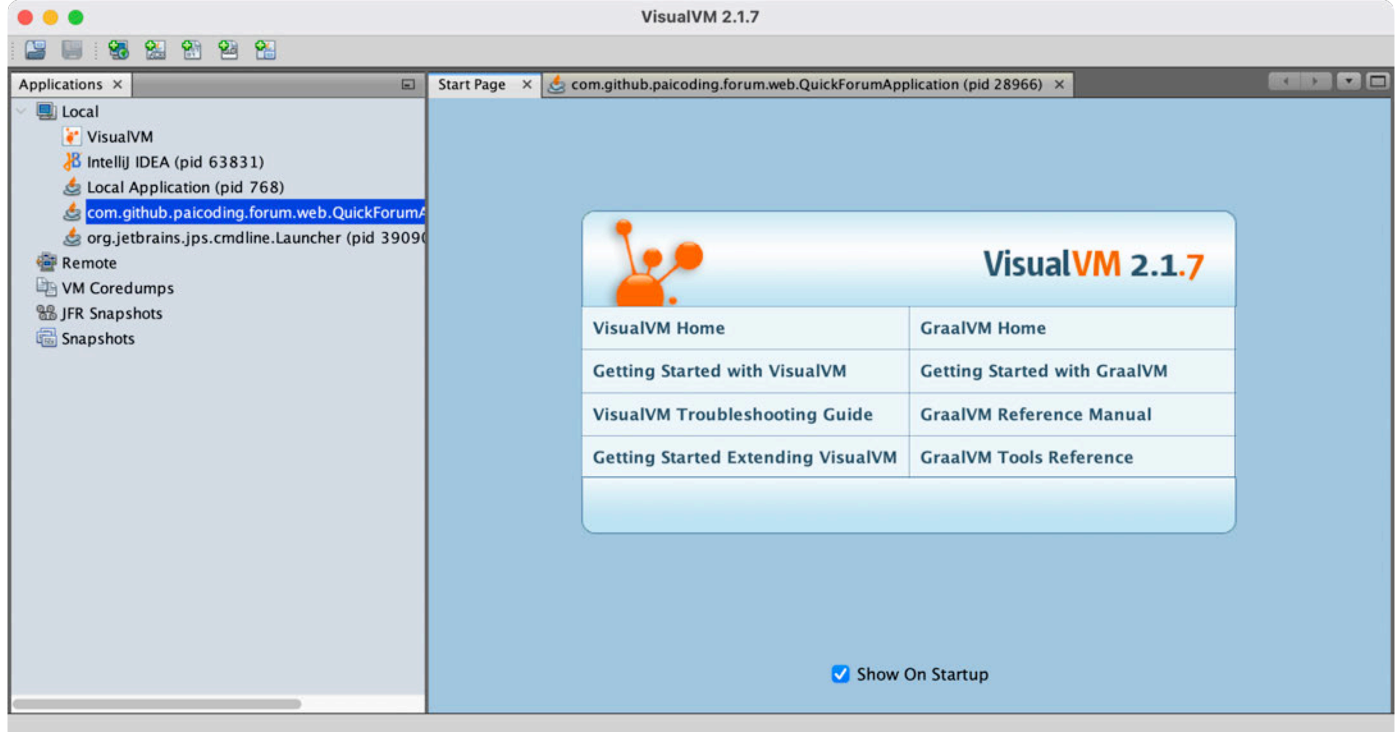
IDE Integrations | Plugins Offline | Previous Releases

Systems:	Java:	What's New:
Microsoft Windows	Oracle JDK 8~21	Support for JDK 21
Linux	OpenJDK 8~21	Heap Viewer improvements
macOS	GraalVM 19~23	

See the Release Notes for details on system requirements, new features, API changes and fixed bugs.

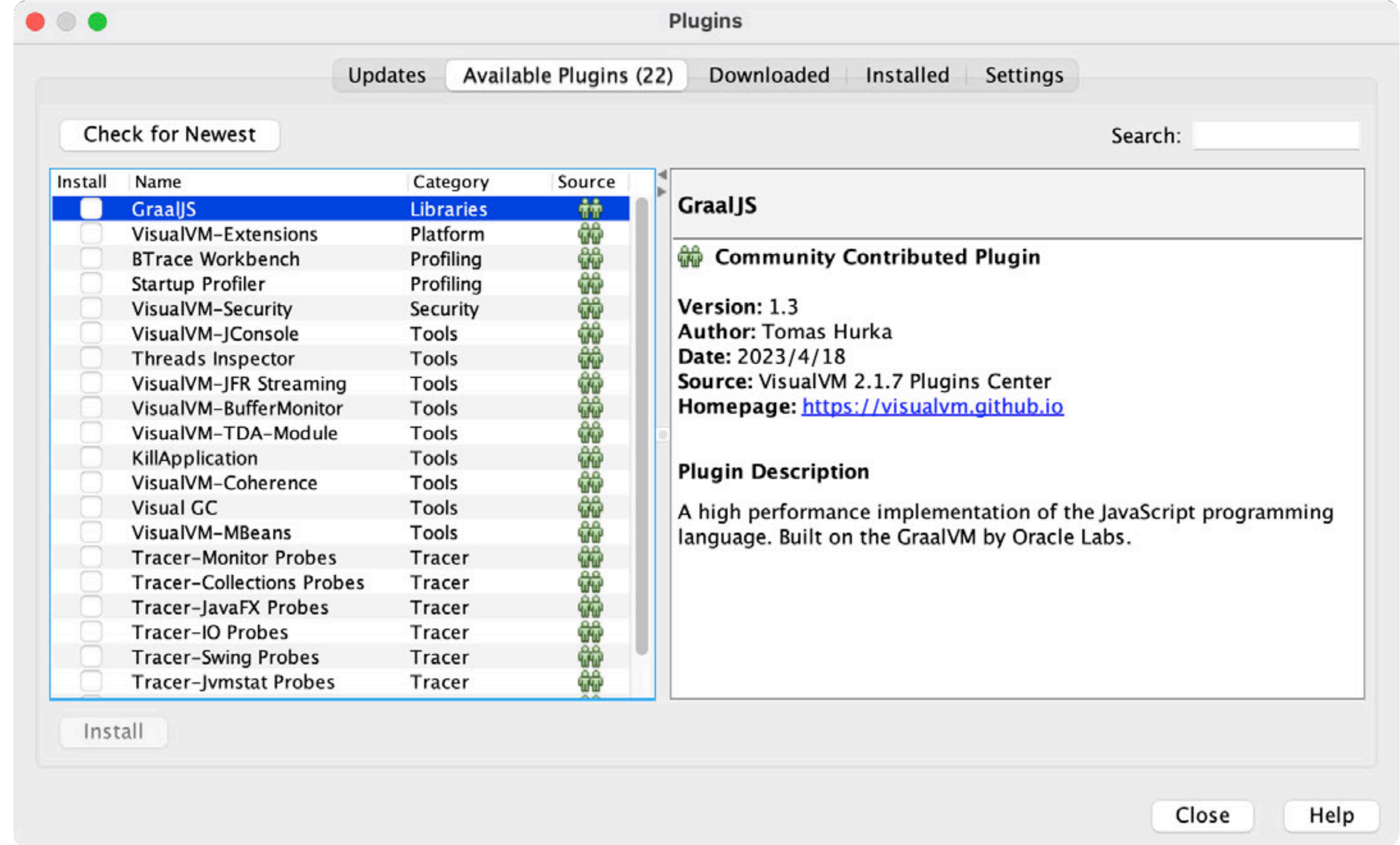
Follow the Getting Started guide to learn how to use VisualVM. To get more details, see the Features and Plugins sections and read the VisualVM Documentation. See the Troubleshooting Guide if experiencing any problems with starting or using the tool.

安装完成后打开的界面如下所示：



VisualVM 比 JConsole 强的不是一星半点，它不仅拥有更漂亮的身段，还支持插件功能。

点击 `tools` -> `plugins`，在 `可用插件` 里可以看到大量的插件，按需安装即可。



VisualVM 中`概述`、`监视`、`线程`与 JConsole 差别不大，这里就不在赘述。

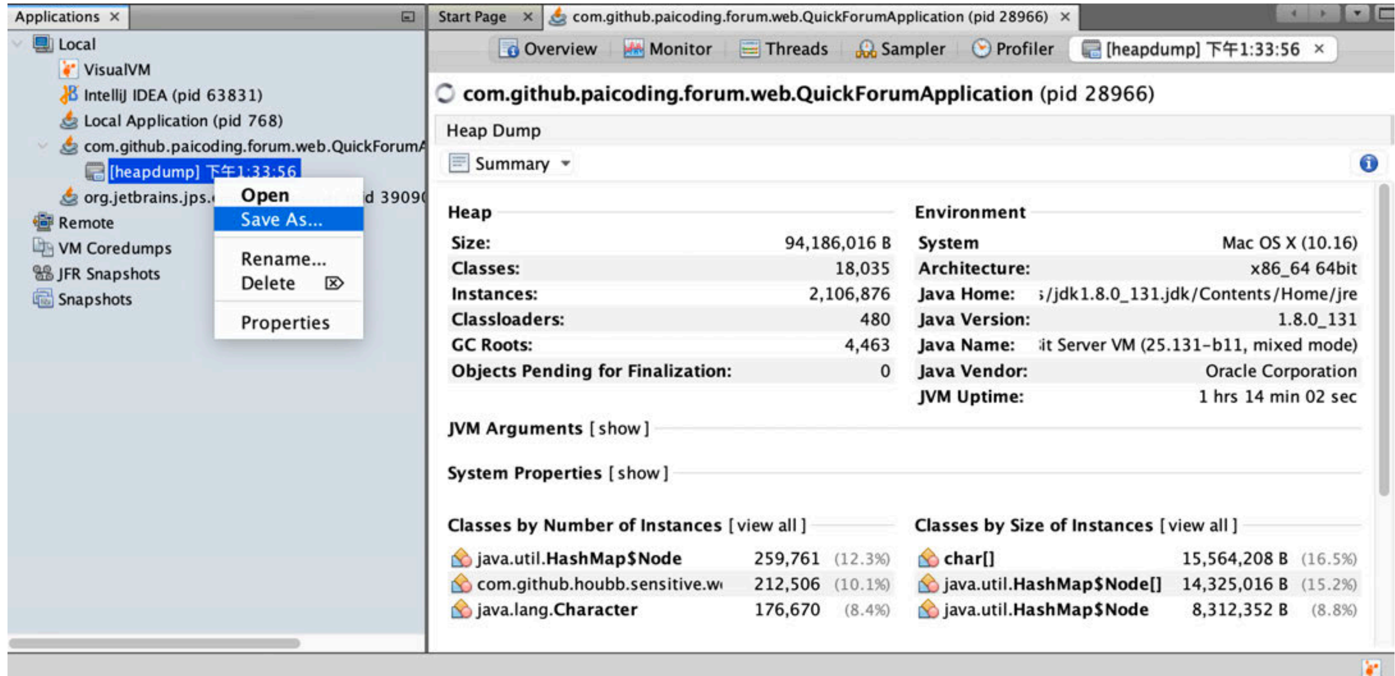
生成、浏览堆转储快照

在 VisualVM 中生成堆转储快照文件有两种方式，可以执行下列任一操作：

①、在`应用程序`面板中右键选择`堆Dump`（也就是 Heap Dump）。

②、在`应用程序`面板中选择应用程序，在“监视”面板中单击`堆Dump`。

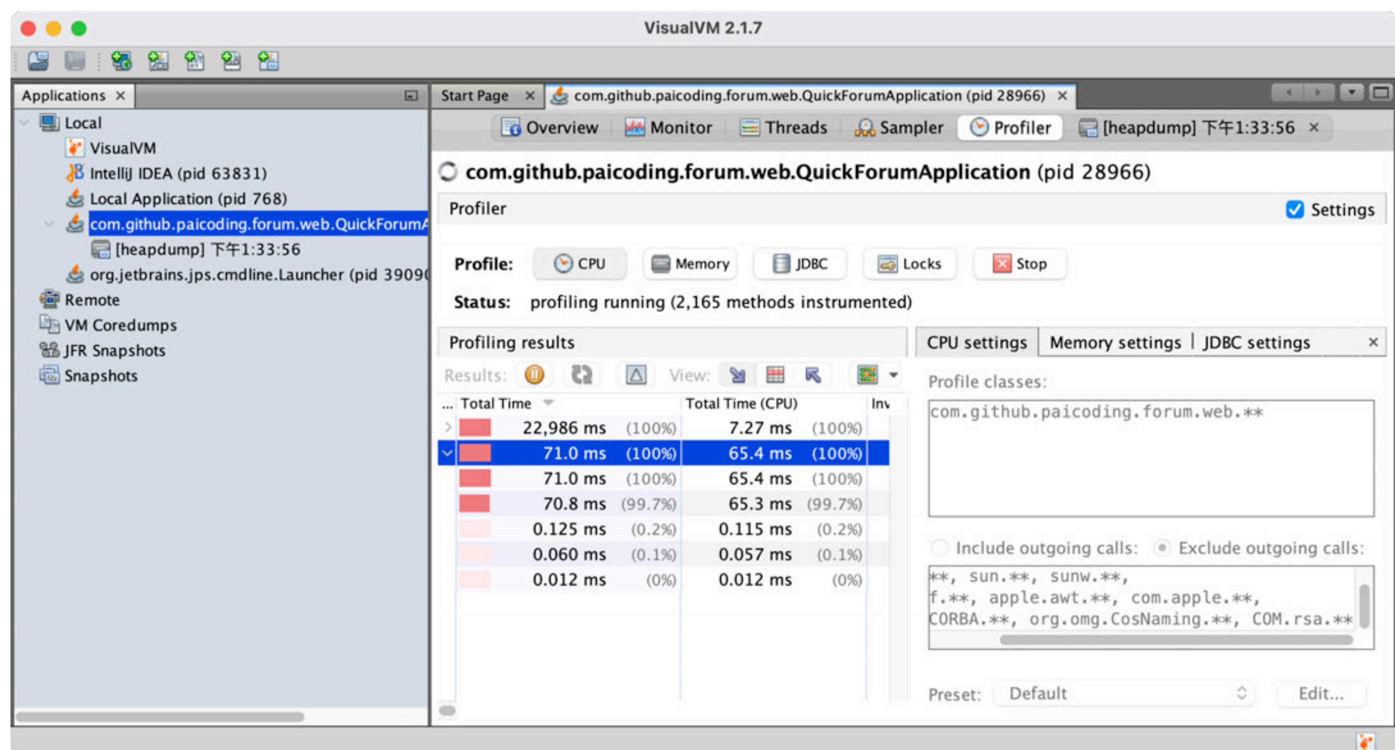
如果需要把堆转储快照保存文件后分享出去，可以在 heapdump 节点上右键选择“另存为”菜单，否则当 VisualVM 关闭时，生成的堆转储快照文件会被当作临时文件自动清理掉。



分析程序性能

如果想对应用程序的 CPU 和内存情况进行分析，可以在「分析 profiler」面板中点击「CPU」或者「Memory」，然后 VisualVM 会记录这段时间中应用程序执行过的所有方法。

比如说 CPU 将会统计每个方法的执行次数、执行耗时。比如说内存将会统计每个方法的内存分配情况。

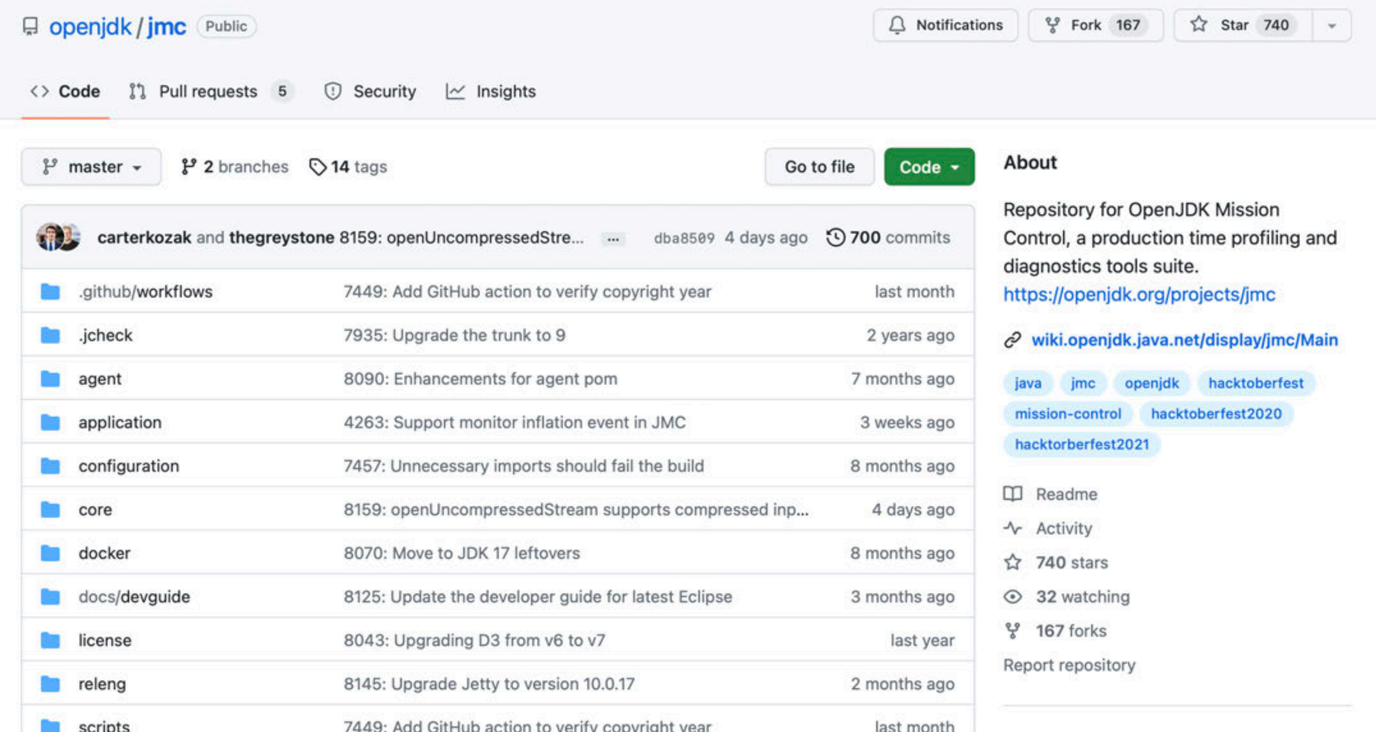


注意点击开始后，回到应用程序进行操作。等要分析的操作执行结束后，点击“停止”按钮结束监控过程。

Java Mission Control

JMV 最初是 JRockit VM 中的诊断工具，但在 Oracle JDK7 Update 40 以后，就绑定到了 HotSpot VM 中。不过后来又被 Oracle 开源出来作为一个单独的产品。

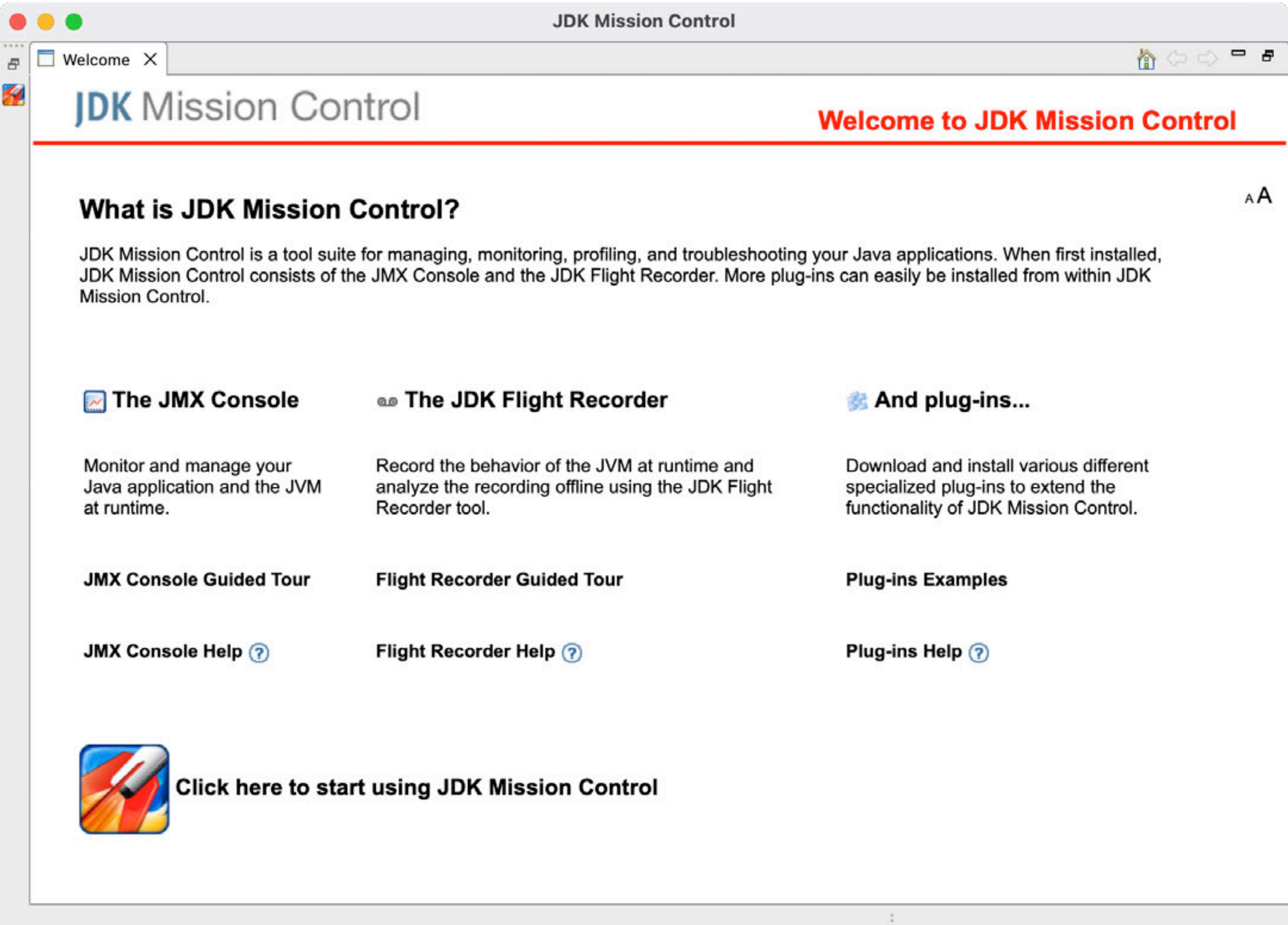
> GitHub 地址：https://github.com/openjdk/jmc



Oracle 官方下载比较慢，可以通过 jdk.java.net 下载。

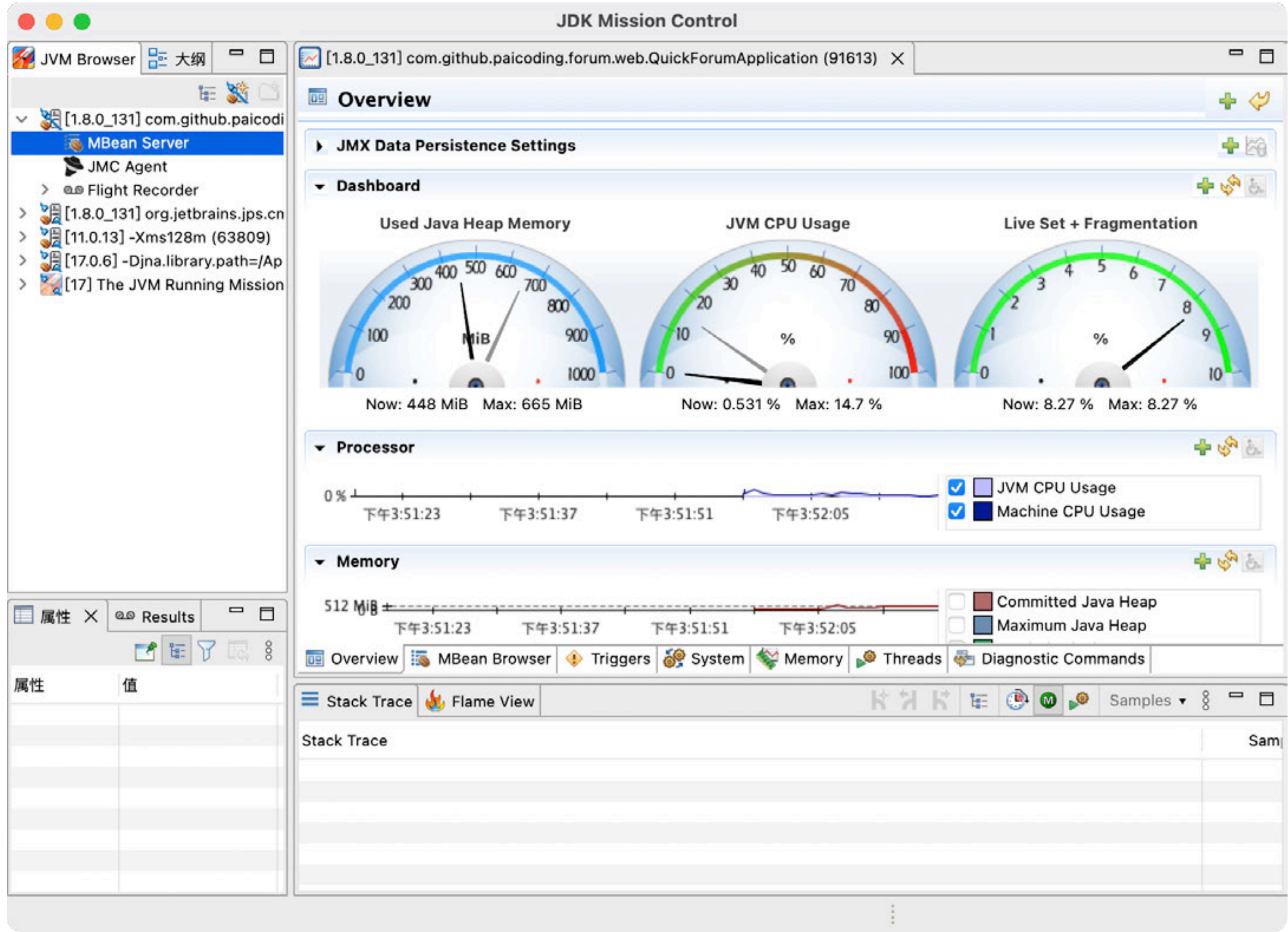
> https://jdk.java.net/jmc/8/

解压后启动的界面如下所示：



MBean Server

点击本地进程的`MBean服务器`：



> MBean 是管理 Java 应用程序的一种标准方式，它是 Java 管理扩展（JMX）的一部分。MBean 代表可管理的 Java 对象，它们的属性和操作可以通过 JMX 进行访问。

仪表盘显示了 Java 堆的使用率，CPU 使用率和 Live Set+Fragmentation（Live Set 是指存活对象的大小，Fragmentation 是指碎片的大小）。

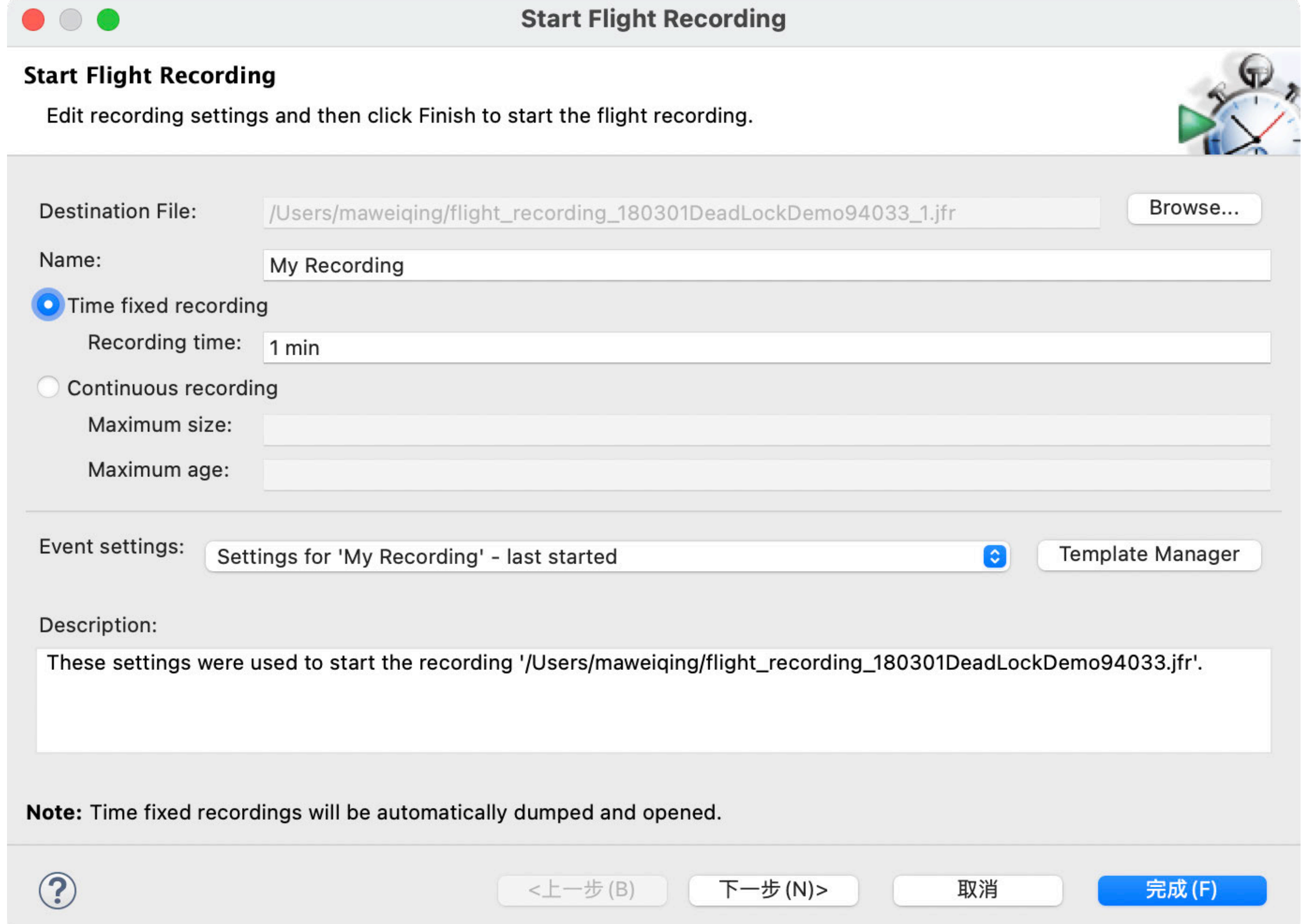
飞行记录器（Flight Recorder）

飞行记录器（JFR）是 JMC 提供的另一功能，通过记录应用程序在一段时间内的运行情况，再进行分析和展示，可以更进一步对应用程序的性能进行分析和诊断。

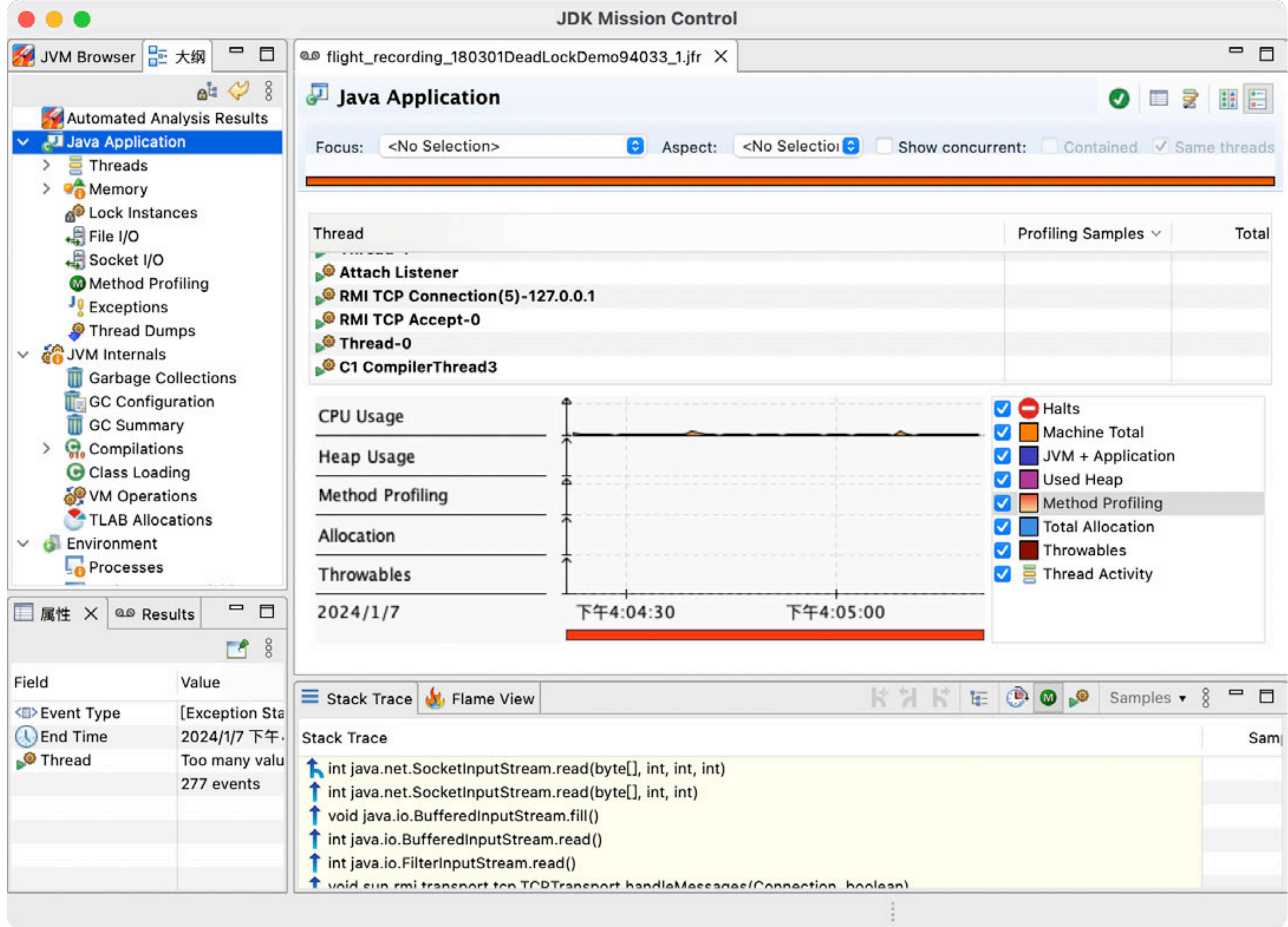
要使用 JFR，程序启动需要带以下参数：

```
-XX:+UnlockCommercialFeatures  -XX:+FlightRecorder
```

连接加了相关参数启动的程序，启动飞行记录，进行一分钟的性能记录：



记录结束后，JMC 会自动打开刚才的记录：



JFR 提供的数据质量通常也要比其他工具通过代理形式采样获得的更高。

以垃圾搜集为例，HotSpot 的 MBean 中一般有各个分代大小、收集次数、时间、占用率等数据，这些都属于“结果”类的信息，而 JFR 中还可以看到内存中这段时间分配了哪些对象、哪些对象被回收了，这些都属于“过程”类的信息。

我这里提供一些可供测试的代码，大家可以在本地跑一下，看看 JFR 的效果。

第一个：CPU 使用过高：

```
/**
 * 消耗CPU的线程
 * 不断循环进行浮点运算
 */
private static void cpuHigh() {
    Thread thread = new Thread(() -> {
        Thread.currentThread().setName("cpu_high_thread");
        while (true){
            double pi = 0;
            for (int i = 0; i < Integer.MAX_VALUE; i++) {
                pi += Math.pow(-1, i) / (2 * i + 1);
            }
            System.out.println("Pi: " + pi * 4);
        }
    });
```

```
    thread.start();
}
```

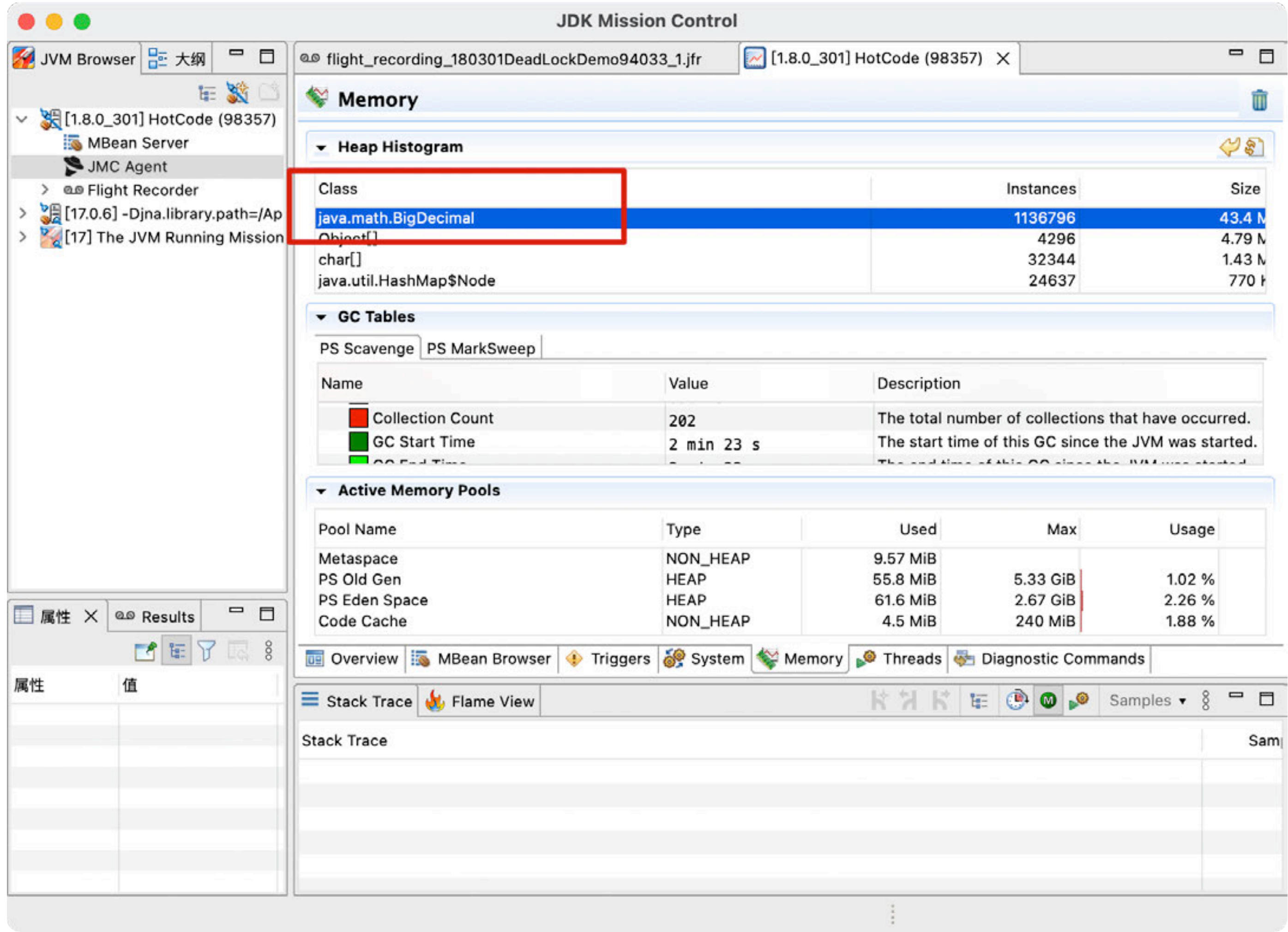
第二个：内存使用过高：

```
/**
 * 不断新增 BigDecimal 信息到 list
 */
private static void allocate() {
    new Thread(()->{
        Thread.currentThread().setName("memory_allocate_thread");
        List<BigDecimal> list = new ArrayList<>();
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            list.add(new BigDecimal(i));
        }
    }).start();
}
```

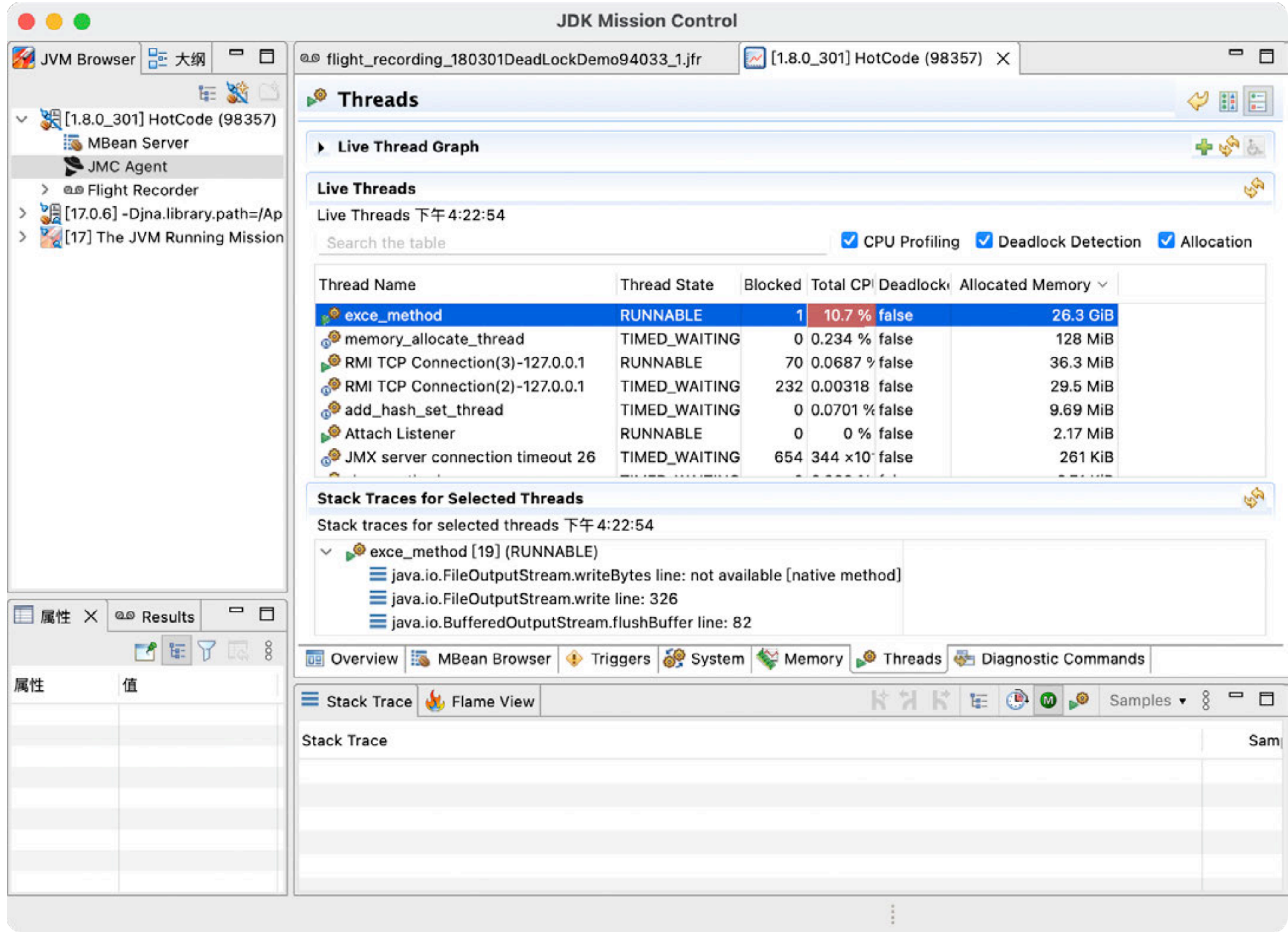
完整示例可以参考这个 GitHub 仓库：

> https://github.com/itwanger/paicoding/blob/main/paicoding-web/src/test/java/com/github/paicoding/forum/test/javabetter/jvm/HotCode.java

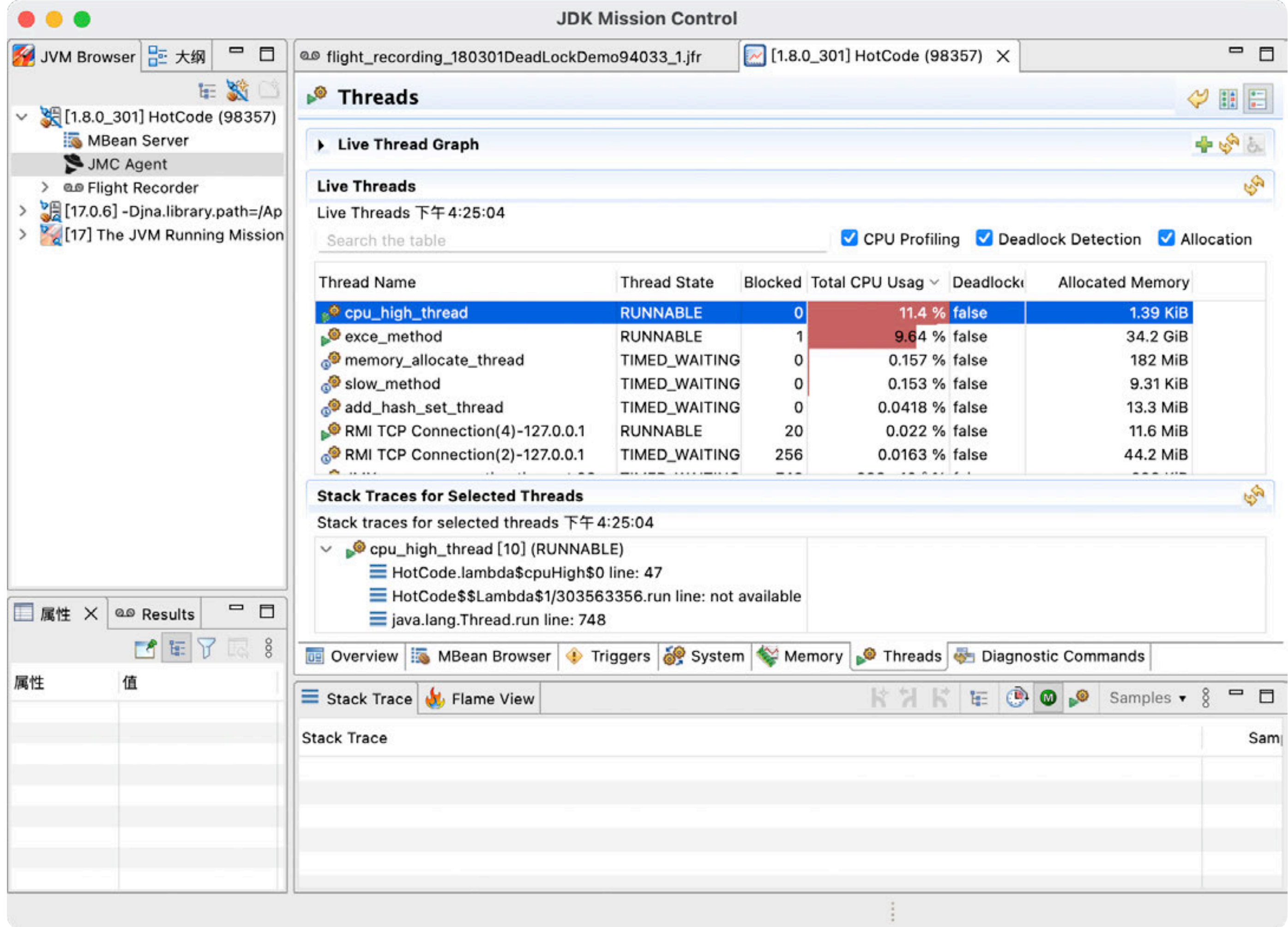
比如说通过「内存面板」可以看出 BigDecimal 对象占用了最多的内存。



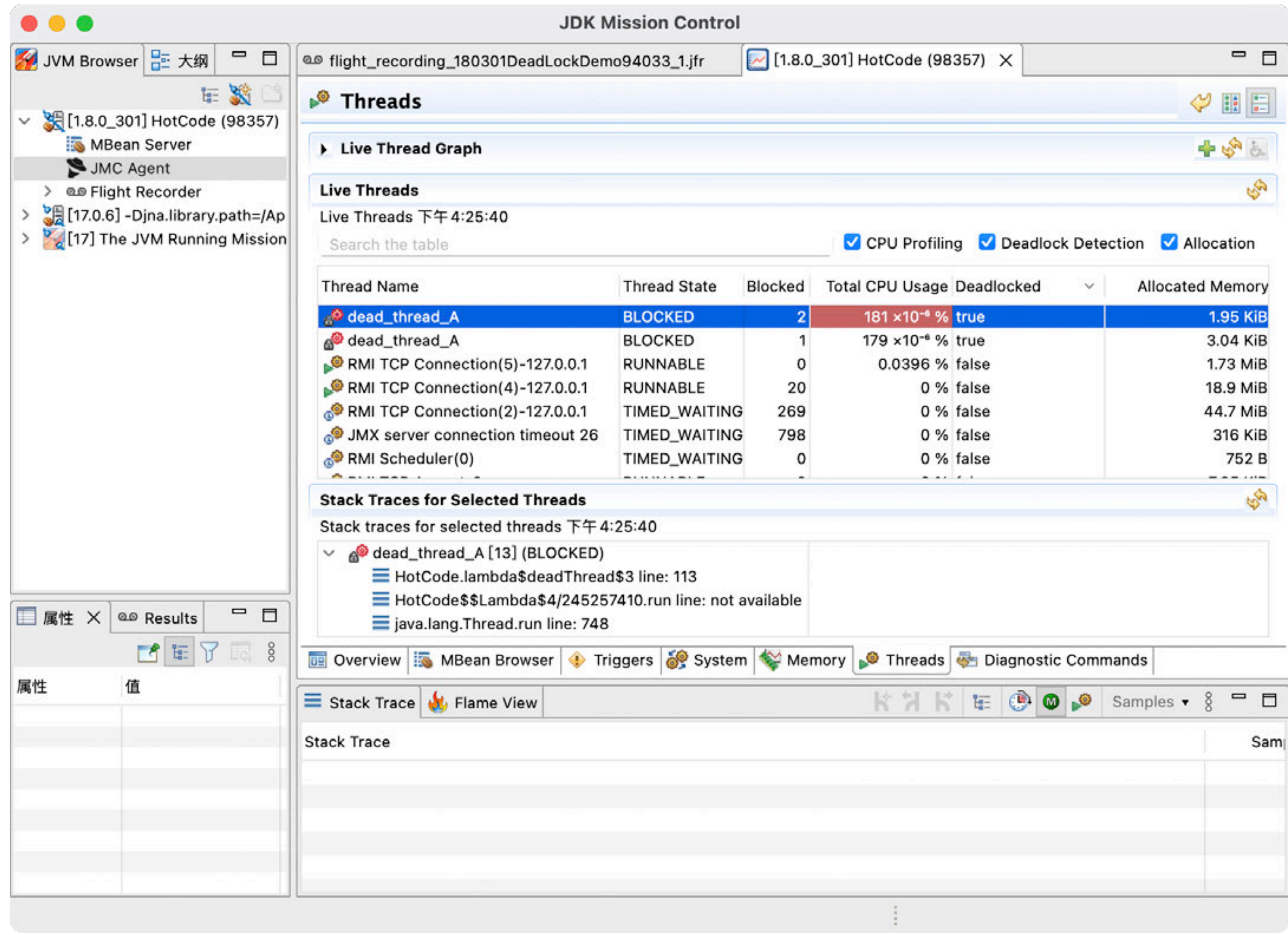
如果想进一步分析内存占用来源，可以切到线程页面，勾选三个复选框，可以在 memory 这里看到内存情况。



还可以看到这里的 `cpu_high_thread` 在不断地计算浮点数，所以占用了较多的 CPU。



死锁的情况也可以在这里看得到。



第三方工具

以上三个都属于 Oracle 官方提供的性能监控工具，除此之外还有一些第三方的性能监控工具。

- **「MAT」**

Java 堆内存分析工具。

- **「GChisto」**

GC 日志分析工具。

- **「GCViewer」**

`GC` 日志分析工具。

- **「JProfiler」**

商用的性能分析利器。

- **「arthas」**

阿里开源诊断工具。

- **「async-profiler」**

Java 应用性能分析工具，开源、火焰图、跨平台。

小结

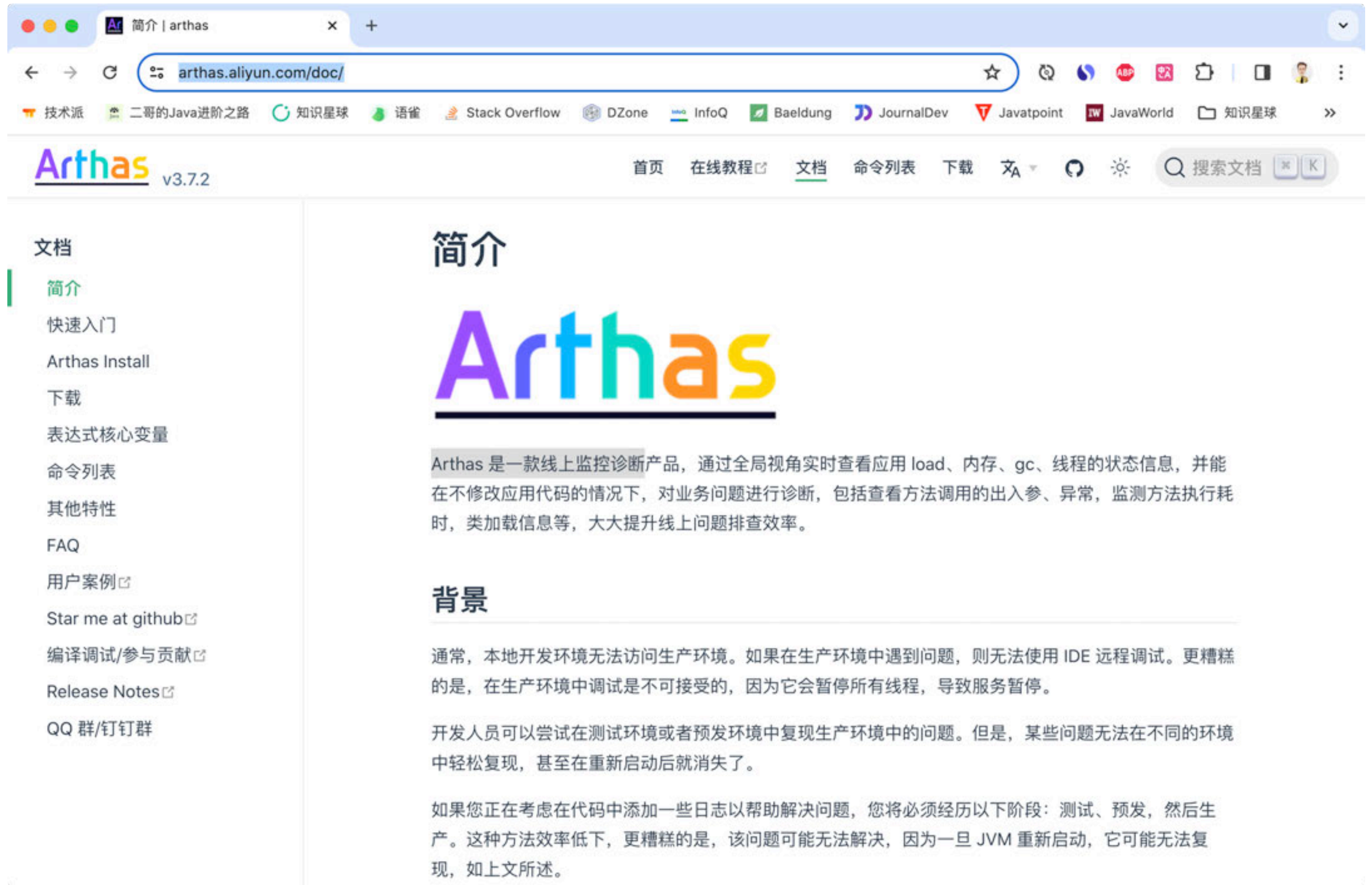
本篇我们介绍了一些可视化的性能监控工具，包括 JConsole、VisualVM、Java Mission Control 等，阿里的 Arthas 我们留到后面单独去讲。

> 参考链接：星球嘉宾三分恶 性能监控工具-可视化工具篇

第十六节：JVM 性能监控之 Arthas 篇

Arthas 是阿里开源的一款线上 Java 诊断神器，通过全局的视角可以查看应用程序的内存、GC、线程等状态信息，并且能够在不修改代码的情况下，对业务问题进行诊断，包括查看方法的参数调用、执行时间、异常堆栈等信息，大大提升了生产环境中问题排查的效率。

Arthas 的官方网站是 https://arthas.aliyun.com/doc/，目前最新的版本是 3.7.2。



比我们前面介绍的命令行工具和可视化工具，都要强大得多，如果你再遇到下面这些问题，就可以迎刃而解了。

- 客户线上问题，应该如何复现，让客户再点一下吗？
- 异常被吃掉，手足无措，看是哪个家伙写的，竟然是自己！
- 排查别人线上的 bug，不仅代码还没看懂，还没一行日志，捏了一把汗！
- 预发 debug，稍微时间长点，群里就怨声载道！
- 加日志重新部署，半个小时就没了，问题还没有找到，头顶的灯却早已照亮了整层楼......
- 线上机器不能 debug，也不能开 debug 端口，重新部署会不会破坏现场呢？
- 怀疑入参有问题，怀疑合并代码有问题，怀疑没有部署成功，全是问号......
- 一个问题排查一天，被 Diss 排查问题慢......

星球里也有球友一直在呼唤 Arthas 的教程，那这篇内容我们就来详细地盘一盘。

安装 Arthas

macOS 安装

我们先在本地试一下哈，由于我本机是 macOS，所以我这里就以 macOS 为例，Windows 用户可以参考官方文档，非常简单。

我本机已经启动了技术派项目，我们就以技术派为例，来看看 Arthas 的使用。

```
maweiqing@itwanger-2 string1 % jps
88497 Jps
87744 Launcher
63218 Main
79209 QuickForumApplication
maweiqing@itwanger-2 string1 %
```

官方推荐的方式是通过 arthas-boot 来安装，那我们就按照这种来：

```
curl -O https://arthas.aliyun.com/arthas-boot.jar
java -jar arthas-boot.jar
```

执行完上述命令后，Arthas 会列出可以进行监控的 Java 进程，比如说下图中的第 2 个 `[2]: 79209 com.github.paicoding.forum.web.QuickForumApplication` 就是技术派的进程，直接输入 `2`，然后回车。Arthas 会连接到技术派的进程上，并输出带有 Arthas 的日志，进入 Arthas 的命令交互界面。

```
~/Documents/Github/arthas
curl -O https://arthas.aliyun.com/arthas-boot.jar
java -jar arthas-boot.jar
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  138k  100  138k    0     0   597k      0 --:--:-- --:--:-- --:--:--  597k
[INFO] JAVA_HOME: /Library/Java/JavaVirtualMachines/jdk1.8.0_301.jdk/Contents/Home/jre
[INFO] arthas-boot version: 3.7.2
[INFO] Found existing java process, please choose one and input the serial number of the process, eg : 1.
 Then hit ENTER.
* [1]: 63218 com.intellij.idea.Main
  [2]: 79209 com.github.paicoding.forum.web.QuickForumApplication
  [3]: 88589 org.jetbrains.jps.cmdline.Launcher
2
[INFO] Start download arthas from remote server: https://arthas.aliyun.com/download/3.7.2?mirror=aliyun
[INFO] File size: 17.84 MB, downloaded size: 5.34 MB, downloading ...
[INFO] File size: 17.84 MB, downloaded size: 9.65 MB, downloading ...
[INFO] File size: 17.84 MB, downloaded size: 16.15 MB, downloading ...
[INFO] Download arthas success.
[INFO] arthas home: /Users/itwanger/.arthas/lib/3.7.2/arthas
[INFO] Try to attach process 79209
Picked up JAVA_TOOL_OPTIONS:
[INFO] Attach process 79209 success.
[INFO] arthas-client connect 127.0.0.1 3658

wiki       https://arthas.aliyun.com/doc
tutorials  https://arthas.aliyun.com/doc/arthas-tutorials.html
version    3.7.2
main_class com.github.paicoding.forum.web.QuickForumApplication
pid        79209
time       2024-01-09 10:46:47

[arthas@79209]$
```

Linux 安装

本地 OK 后，我们来试一下服务器上的项目，技术派是部署在腾讯云的香港服务器上，我们先登录到服务器上，然后执行下面的命令获取 arthas-boot.jar：

```
curl -O https://arthas.aliyun.com/arthas-boot.jar
```

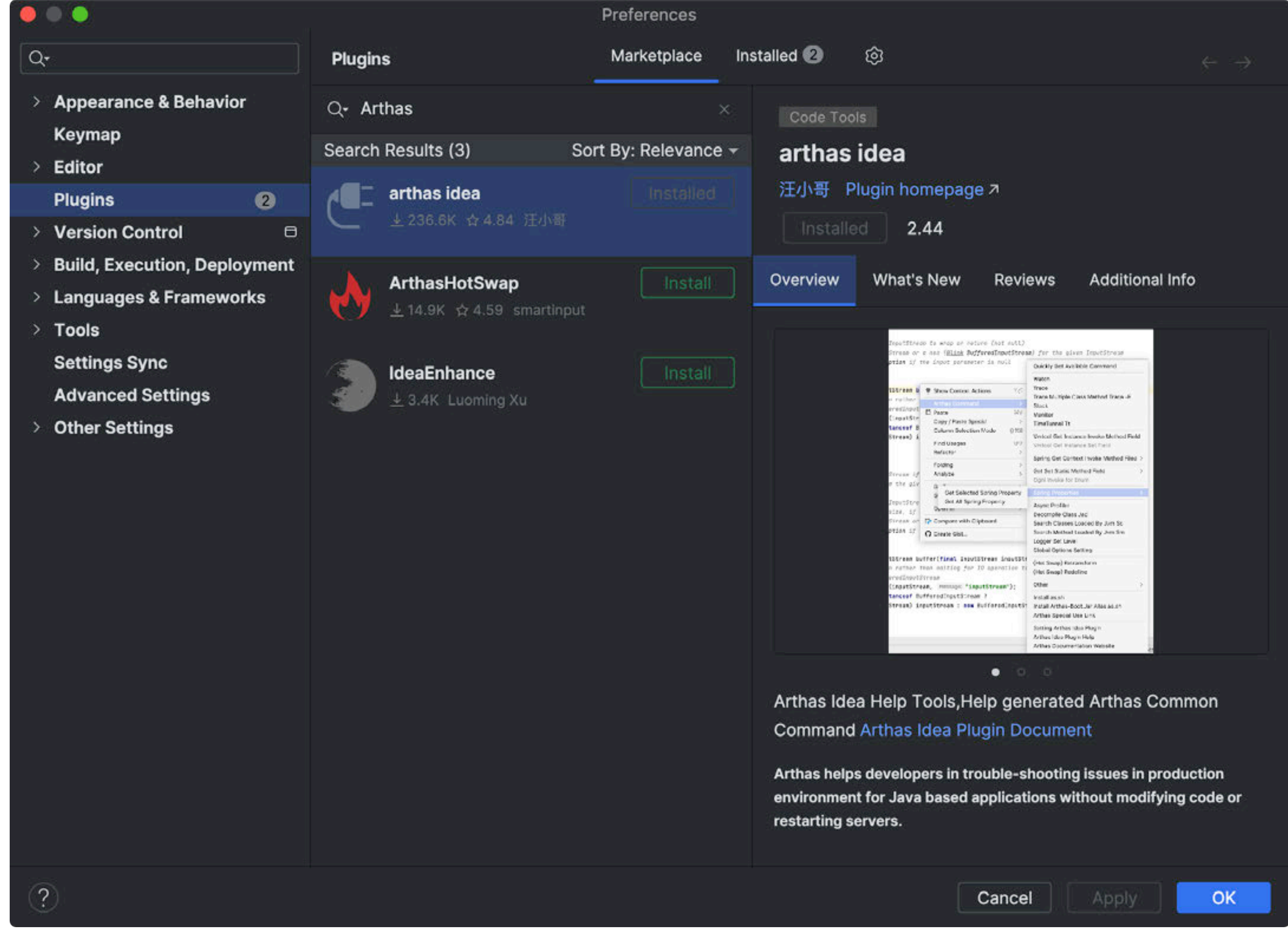
然后执行 `java -jar arthas-boot.jar`，Arthas 会列出可以进行监控的 Java 进程，我们输入 `1`，然后回车，Arthas 就会连接到技术派的进程上，并输出带有 Arthas 的日志，进入 Arthas 的命令交互界面。

```
[admin@VM-0-5-tencentos paicoding]$ curl -O https://arthas.aliyun.com/arthas-boot.jar
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  138k  100  138k    0     0  34898      0  0:00:04  0:00:04 --:--:-- 34898
[admin@VM-0-5-tencentos paicoding]$ java -jar arthas-boot.jar
[INFO] JAVA_HOME: /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.382.b05-2.tl3.x86_64/jre
[INFO] arthas-boot version: 3.7.2
[INFO] Found existing java process, please choose one and input the serial number of the process, eg : 1. Then h
* [1]: 3211052 paicoding-web-0.0.1-SNAPSHOT.jar
1
[INFO] Start download arthas from remote server: https://arthas.aliyun.com/download/3.7.2?mirror=aliyun
[INFO] File size: 17.84 MB, downloaded size: 639.54 KB, downloading ...
[INFO] File size: 17.84 MB, downloaded size: 12.27 MB, downloading ...
[INFO] Download arthas success.
[INFO] arthas home: /home/admin/.arthas/lib/3.7.2/arthas
[INFO] Try to attach process 3211052
Picked up JAVA_TOOL_OPTIONS:
[INFO] Attach process 3211052 success.
[INFO] arthas-client connect 127.0.0.1 3658
  ,---.  ,------. ,--------.,--.  ,--.  ,---.   ,---.
 /  O  \ |  .--. ''--.  .--'|  '--'  | /  O  \ '   .-'
|  .-.  ||  '--'.'   |  |   |  .--.  ||  .-.  |`.  `-.
|  | |  ||  |\  \    |  |   |  |  |  ||  | |  |.-'    |
`--' `--'`--' '--'   `--'   `--'  `--'`--' `--'`-----'

wiki       https://arthas.aliyun.com/doc
tutorials  https://arthas.aliyun.com/doc/arthas-tutorials.html
version    3.7.2
main_class
pid        3211052
time       2024-01-09 10:56:22
```

OK，非常简单，相信大家都能搞定。

IDEA Arthas 插件

Arthas 也提供了 IDEA 插件，可以直接在 IDEA 中使用 Arthas，非常方便，我们来看看怎么安装。



官方文档：

> https://www.yuque.com/arthas-idea-plugin/help

Arthas 常用命令

Arthas 提供了非常多的命令供我们使用，比如说和 JVM 相关的：

- `dashboard`：查看 JVM 的实时数据，包括 CPU、内存、GC、线程等信息。
- `jvm`：查看 JVM 的信息，包括 JVM 参数、类加载器、类信息、线程信息等。
- `sysprop`：查看和修改 JVM 的系统属性。
- `vmoption`：查看和修改 JVM 的启动参数。
- `heapdump`：生成堆内存快照，类似于 jmap 命令。

比如说和类加载相关的：

- `sc`：查看类的信息，包括类的结构、方法、字段等。
- `sm`：查看方法的信息，包括方法的参数、返回值、异常等。

比如说和方法调用相关的：

- `tt`：统计方法的调用次数和耗时。
- `trace`：跟踪方法的调用过程，包括方法的参数、返回值、异常等。
- `monitor`：监控方法的调用过程。

我来带大家体验一些比较常用的命令，其他的命令大家可以参考官方文档。

dashboard 命令

dashboard 命令可以查看 JVM 的实时数据，包括线程、内存、线程、运行时参数等。

```
redis-server    curl    +    Update Warp

~/Documents/Github/arthas
curl -O https://arthas.aliyun.com/arthas-boot.jar
java -jar arthas-boot.jar

[arthas@79209]$ dashboard
ID  NAME                        GROUP       PRIORITY STATE    %CPU    DELTA_TI TIME      INTERRU DAEMON
68  File Watcher                main        5        TIMED_W  0.0     0.000    35:4.952  false   true
-1  VM Periodic Task Thread     -           -1       -        0.0     0.000    0:17.793  false   true
-1  VM Thread                   -           -1       -        0.0     0.000    0:12.611  false   true
62  lettuce-timer-3-1           main        5        TIMED_W  0.0     0.000    0:11.546  false   true
-1  GC task thread#2 (Parallel  -           -1       -        0.0     0.000    0:9.023   false   true
-1  GC task thread#0 (Parallel  -           -1       -        0.0     0.000    0:9.015   false   true
-1  GC task thread#3 (Parallel  -           -1       -        0.0     0.000    0:8.978   false   true
-1  GC task thread#4 (Parallel  -           -1       -        0.0     0.000    0:8.975   false   true
-1  GC task thread#5 (Parallel  -           -1       -        0.0     0.000    0:8.967   false   true
-1  GC task thread#7 (Parallel  -           -1       -        0.0     0.000    0:8.938   false   true
-1  GC task thread#6 (Parallel  -           -1       -        0.0     0.000    0:8.924   false   true
-1  GC task thread#1 (Parallel  -           -1       -        0.0     0.000    0:8.900   false   true
-1  C1 CompilerThread3          -           -1       -        0.0     0.000    0:4.201   false   true
50  Catalina-utility-2          main        1        TIMED_W  0.0     0.000    0:2.118   false   false
49  Catalina-utility-1          main        1        WAITING  0.0     0.000    0:2.085   false   false
87  DestroyJavaVM               main        5        RUNNABL  0.0     0.000    0:2.038   false   false
69  http-nio-8080-exec-1        main        5        WAITING  0.0     0.000    0:1.266   false   true
79  http-nio-8080-Poller        main        5        RUNNABL  0.0     0.000    0:1.146   false   true
34  mysql-cj-abandoned-connect  main        5        TIMED_W  0.0     0.000    0:1.137   false   true
44  I/O dispatcher 6            main        5        RUNNABL  0.0     0.000    0:0.827   false   false
40  I/O dispatcher 2            main        5        RUNNABL  0.0     0.000    0:0.808   false   false
Memory                   used    total   max     usage   GC
heap                     175M    364M    3641M   4.81%   gc.ps_scavenge.count          4246
ps_eden_space            71M     127M    1363M   5.24%   gc.ps_scavenge.time(ms)       22209
ps_survivor_space        480K    512K    512K    93.75%  gc.ps_marksweep.count         4
ps_old_gen               103M    237M    2731M   3.78%   gc.ps_marksweep.time(ms)      903
nonheap                  128M    136M    -1      93.79%
code_cache               20M     20M     240M    8.67%
metaspace                95M     102M    -1      93.03%
compressed_class_space   12M     13M     1024M   1.18%
direct                   80K     80K     -       100.00%
mapped                   0K      0K      -       0.00%
Runtime
os.name                                                  Mac OS X
os.version                                               10.15.7
java.version                                             1.8.0_301
```

thread 命令

thread 命令可以查看线程的信息，包括线程的状态、线程的堆栈等。

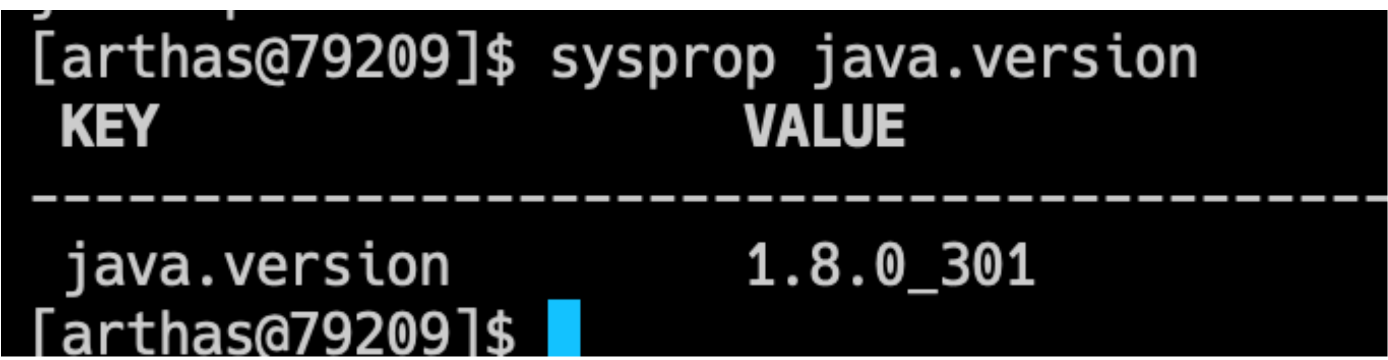
```
[arthas@79209]$ thread
Threads Total: 88, NEW: 0, RUNNABLE: 29, BLOCKED: 0, WAITING: 31, TIMED_WAITING: 13, TERMINATED: 0, Inter
nal threads: 15
ID   NAME                        GROUP     PRIORITY  STATE    %CPU    DELTA_TI  TIME      INTERRU  DAEMON
369  arthas-command-execute      system    5         RUNNABL  0.27    0.000     0:0.004   false    true
-1   VM Periodic Task Thread     -         -1        -        0.1     0.000     0:17.900  false    true
62   lettuce-timer-3-1           main      5         TIMED_W  0.03    0.000     0:11.611  false    true
2    Reference Handler           system    10        WAITING  0.0     0.000     0:0.026   false    true
3    Finalizer                   system    8         WAITING  0.0     0.000     0:0.051   false    true
4    Signal Dispatcher           system    9         RUNNABL  0.0     0.000     0:0.000   false    true
14   RMI TCP Accept-0            system    5         RUNNABL  0.0     0.000     0:0.002   false    true
15   Attach Listener             system    9         RUNNABL  0.0     0.000     0:0.107   false    true
17   RMI Scheduler(0)            system    5         WAITING  0.0     0.000     0:0.002   false    true
344  arthas-timer                system    9         WAITING  0.0     0.000     0:0.000   false    true
347  arthas-NettyHttpTelnetBoot  system    5         RUNNABL  0.0     0.000     0:0.020   false    true
348  arthas-NettyWebsocketTtyBo  system    5         RUNNABL  0.0     0.000     0:0.000   false    true
```

thread 命令的参数如下：

```
# 打印当前最忙的3个线程的堆栈信息
thread -n 3
# 查看ID为1的线程堆栈信息
thread 1
# 找出当前阻塞其他线程的线程
thread -b
# 查看指定状态的线程
thread -state WAITING
```

sysprop 命令

sysprop 命令可以查看和修改 JVM 的系统属性。



支持 TAB 键补全命令哈~

logger 命令

logger 命令可以查看和修改日志的级别，这个命令非常有用。

比如说生产环境上一般是不会打印 DEBUG 级别的日志的，但是有时候我们需要打印 DEBUG 级别的日志来排查问题，这个时候就可以使用 logger 命令来修改日志的级别。

第一步，先用 logger 命令查看默认使用的日志级别：

```
[arthas@79209]$ logger
 name                ROOT
 class               ch.qos.logback.classic.Logger
 classLoader         sun.misc.Launcher$AppClassLoader@18b4aac2
 classLoaderHash     18b4aac2
 level               INFO
 effectiveLevel      INFO
 additivity          true
 codeSource          file:/Users/itwanger/.m2/repository/ch/qos/logback/logback-classic/1.2.11/logbac
                     ssic-1.2.11.jar
 appenders           name            STDOUT
                     class           ch.qos.logback.core.ConsoleAppender
                     classLoader     sun.misc.Launcher$AppClassLoader@18b4aac2
                     classLoaderHash 18b4aac2
                     target          System.out
                     name            service
                     class           ch.qos.logback.core.rolling.RollingFileAppender
                     classLoader     sun.misc.Launcher$AppClassLoader@18b4aac2
                     classLoaderHash 18b4aac2
                     file            logs/pai-dev.log
```

第二步，使用这个命令`logger --name ROOT --level DEBUG`，将日志级别修改为 DEBUG，再次查看日志级别，发现已经修改成功了：

```
[arthas@79209]$ logger --name ROOT --level DEBUG
Update logger level success.
[arthas@79209]$ logger
 name               ROOT
 class              ch.qos.logback.classic.Logger
 classLoader        sun.misc.Launcher$AppClassLoader@18b4aac2
 classLoaderHash    18b4aac2
 level              DEBUG
 effectiveLevel     DEBUG
 additivity         true
 codeSource         file:/Users/itwanger/.m2/repository/ch/qos/logback/logback-classi
                    ssic-1.2.11.jar
 appenders          name            STDOUT
                    class           ch.qos.logback.core.ConsoleAppender
                    classLoader     sun.misc.Launcher$AppClassLoader@18b4aac2
                    classLoaderHash 18b4aac2
                    target          System.out
                    name            service
                    class           ch.qos.logback.core.rolling.RollingFileAppender
                    classLoader     sun.misc.Launcher$AppClassLoader@18b4aac2
                    classLoaderHash 18b4aac2
                    file            logs/pai-dev.log
```

sc 命令

sc 命令可以查看类的信息，包括类的结构、方法、字段等。

示例 1：通过 `sc com.github.paicoding.forum.web.front.*` 查看包下的所有类：

```
[arthas@79209]$ sc com.github.paicoding.forum.web.front.*
com.github.paicoding.forum.web.front.article.rest.ArticleListRestController
com.github.paicoding.forum.web.front.article.rest.ArticleRestController
com.github.paicoding.forum.web.front.article.rest.ArticleRestController$$EnhancerByS
com.github.paicoding.forum.web.front.article.rest.ColumnRestController
com.github.paicoding.forum.web.front.article.view.ArticleListViewController
com.github.paicoding.forum.web.front.article.view.ArticleViewController
com.github.paicoding.forum.web.front.article.view.ColumnViewController
com.github.paicoding.forum.web.front.article.vo.ArticleDetailVo
com.github.paicoding.forum.web.front.chat.helper.WsAnswerHelper
com.github.paicoding.forum.web.front.chat.rest.ChatRestController
com.github.paicoding.forum.web.front.chat.stomp.AuthHandshakeHandler
com.github.paicoding.forum.web.front.chat.stomp.AuthHandshakeInterceptor
com.github.paicoding.forum.web.front.chat.stomp.AuthInChannelInterceptor
com.github.paicoding.forum.web.front.chat.stomp.AuthOutChannelInterceptor
com.github.paicoding.forum.web.front.chat.stomp.WsChatConfig
com.github.paicoding.forum.web.front.chat.stomp.WsChatConfig$$EnhancerBySpringCGLIB$
com.github.paicoding.forum.web.front.chat.stomp.WsChatConfig$$EnhancerBySpringCGLIB$
ySpringCGLIB$$91076ccd
```

示例 2：通过 `sc -d com.github.paicoding.forum.web.front.home.IndexController` 查看类的详细信息：

```
[arthas@79209]$ sc -d com.github.paicoding.forum.web.front.home.IndexController
 class-info        com.github.paicoding.forum.web.front.home.IndexController
 code-source       /Users/itwanger/Documents/Github/paicoding/paicoding-web/target/classes/
 name              com.github.paicoding.forum.web.front.home.IndexController
 isInterface       false
 isAnnotation      false
 isEnum            false
 isAnonymousClass  false
 isArray           false
 isLocalClass      false
 isMemberClass     false
 isPrimitive       false
 isSynthetic       false
 simple-name       IndexController
 modifier          public
 annotation        org.springframework.stereotype.Controller
 interfaces
 super-class       +-com.github.paicoding.forum.web.global.BaseViewController
                     +-java.lang.Object
 class-loader      +-org.springframework.boot.devtools.restart.classloader.RestartClassLoade
                     +-sun.misc.Launcher$AppClassLoader@18b4aac2
                       +-sun.misc.Launcher$ExtClassLoader@548ad73b
 classLoaderHash   3dd47df3

Affect(row-cnt:1) cost in 50 ms.
```

示例 3：通过 `sc -d -f com.github.paicoding.forum.web.front.home.vo.IndexVo` 查看类的字段信息：

```
classLoaderHash   3dd47df3
fields            name     categories
                  type     java.util.List
                  modifier private

                  name     currentCategory
                  type     java.lang.String
                  modifier private

                  name     categoryId
                  type     java.lang.Long
                  modifier private

                  name     topArticles
                  type     java.util.List
                  modifier private

                  name     articles
                  type     com.github.paicoding.forum.api.model.vo.PageListVo
                  modifier private
```

jad 命令

jad 命令可以反编译类的字节码，如果觉得线上代码和预期的不一致，可以反编译看看。

示例 1：通过 `jad com.github.paicoding.forum.web.front.home.IndexController` 反编译类的字节码：

```
[arthas@79209]$ jad com.github.paicoding.forum.web.front.home.IndexController

ClassLoader:
+-org.springframework.boot.devtools.restart.classloader.RestartClassLoader@3dd47df3
  +-sun.misc.Launcher$AppClassLoader@18b4aac2
    +-sun.misc.Launcher$ExtClassLoader@548ad73b

Location:
/Users/itwanger/Documents/Github/paicoding/paicoding-web/target/classes/

       /*
        * Decompiled with CFR.
        *
        * Could not load the following classes:
        *  com.github.paicoding.forum.web.front.home.helper.IndexRecommendHelper
        *  com.github.paicoding.forum.web.front.home.vo.IndexVo
        *  com.github.paicoding.forum.web.global.BaseViewController
        */
       package com.github.paicoding.forum.web.front.home;

       import com.github.paicoding.forum.web.front.home.helper.IndexRecommendHelper;
       import com.github.paicoding.forum.web.front.home.vo.IndexVo;
       import com.github.paicoding.forum.web.global.BaseViewController;
       import javax.servlet.http.HttpServletRequest;
       import org.springframework.beans.factory.annotation.Autowired;
       import org.springframework.stereotype.Controller;
       import org.springframework.ui.Model;
       import org.springframework.web.bind.annotation.GetMapping;

       @Controller
       public class IndexController
       extends BaseViewController {
           @Autowired
           private IndexRecommendHelper indexRecommendHelper;

           @GetMapping(path={"/", "", "/index", "/login"})
           public String index(Model model, HttpServletRequest request) {
```

monitor 命令

monitor 命令可以监控方法的执行信息，包括执行耗时等信息。

示例 1：通过 `monitor -c 3 com.github.paicoding.forum.web.front.home.IndexController index` 监控方法的执行信息，`-c` 参数表示监控的次数：

```
[arthas@79209]$ monitor -c 3 com.github.paicoding.forum.web.front.home.IndexController index
Press Q or Ctrl+C to abort.
Affect(class count: 1 , method count: 1) cost in 113 ms, listenerId: 1
 timestamp       class                   method                   total   succes  fail    avg-rt  fail-r
                                                                          s               (ms)    ate
-----------------------------------------------------------------------------------------------------------
 2024-01-09 11:  com.github.paicoding.f  index                    1       1       0       227.35  0.00%
 57:58           orum.web.front.home.In
                 dexController

 timestamp       class                   method                   total   succes  fail    avg-rt  fail-r
                                                                          s               (ms)    ate
-----------------------------------------------------------------------------------------------------------
 2024-01-09 11:  com.github.paicoding.f  index                    0       0       0       0.00    0.00%
 58:03           orum.web.front.home.In
                 dexController

 timestamp       class                   method                   total   succes  fail    avg-rt  fail-r
                                                                          s               (ms)    ate
```

watch 命令

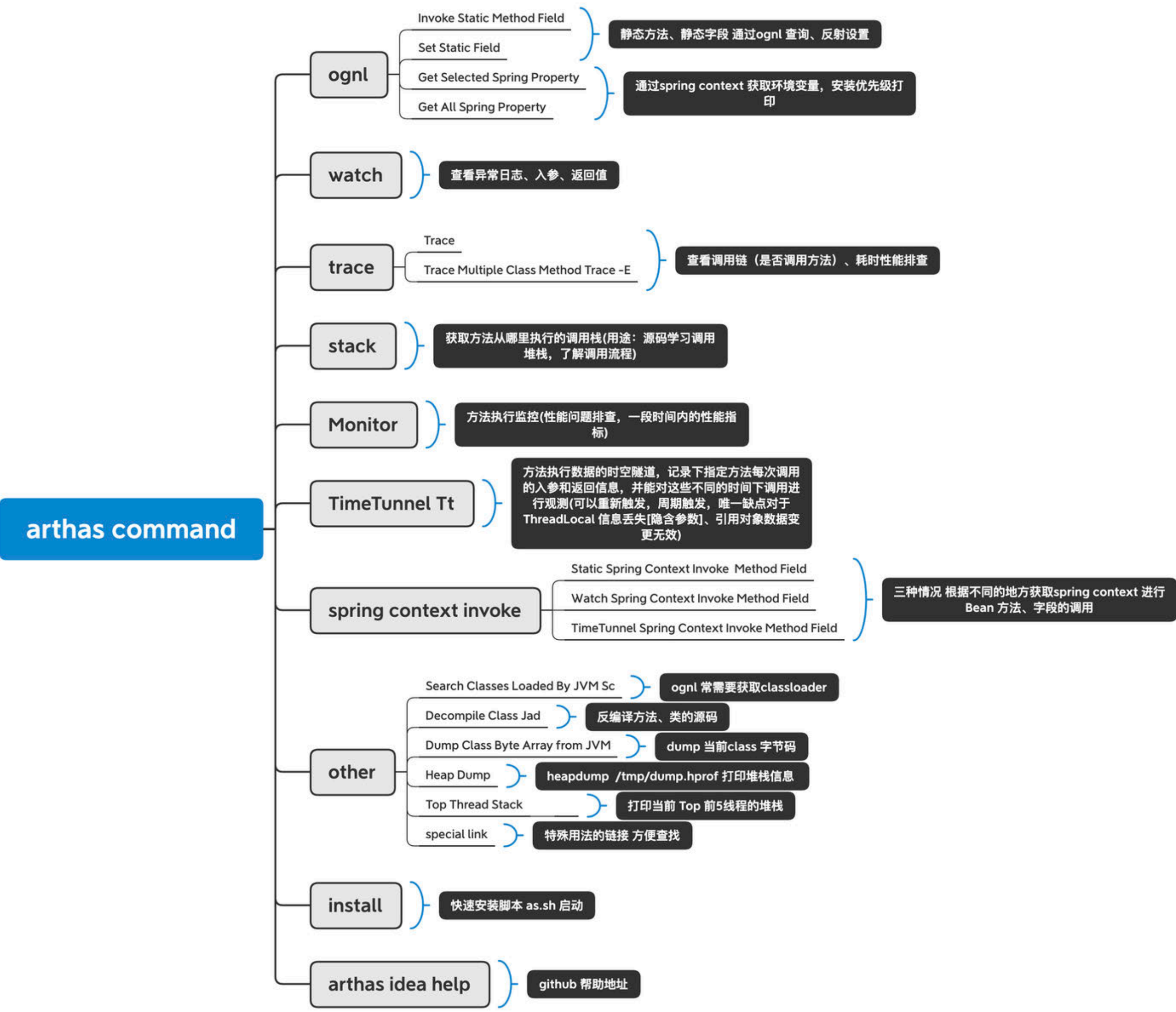
watch 命令可以观察方法执行过程中的参数和返回值。

示例 1：通过 `watch com.github.paicoding.forum.web.front.home.IndexController index` 观察方法的执行过程中的参数和返回值：

```
[arthas@79209]$ watch com.github.paicoding.forum.web.front.home.IndexController index
Press Q or Ctrl+C to abort.
Affect(class count: 1 , method count: 1) cost in 77 ms, listenerId: 2
method=com.github.paicoding.forum.web.front.home.IndexController.index location=AtExit
ts=2024-01-09 12:01:20; [cost=192.87478ms] result=@ArrayList[
    @Object[][isEmpty=false;size=2],
    @IndexController[com.github.paicoding.forum.web.front.home.IndexController@69dfbc33],
    @String[views/home/index],
]
[arthas@79209]$ session (604b500d-81ac-4428-bfcb-a7db0aef8a16) is closed because session i
30 min(s).
```

小结

Arthas 非常强大，还有很多插件可以配合使用，比如我们前面提到的 Arthas IDEA 插件，支持的命令还有以下这些：



文档写得也非常完整，我就不再赘述了，这篇内容也权当一个抛砖引玉。

等后面遇到线上问题了，再用 Arthas 来实战一把，给大家讲一讲。

推荐阅读：

- Arthas 的强大
- Arthas 的热部署
- IDEA Arthas 插件

第十七节：内存泄露排查优化实战

`OutOfMemoryError`，也就是臭名昭著的 OOM（内存溢出），相信很多球友都遇到过，相对于常见的业务异常，如数组越界、空指针等，OOM 问题更难难定位和解决。

这篇内容就以之前碰到的一次线上内存溢出的定位、解决问题的方式展开；希望能对碰到类似问题的球友带来思路和帮助。

主要从`表现-->排查-->定位-->解决` 四个步骤来分析和解决问题。

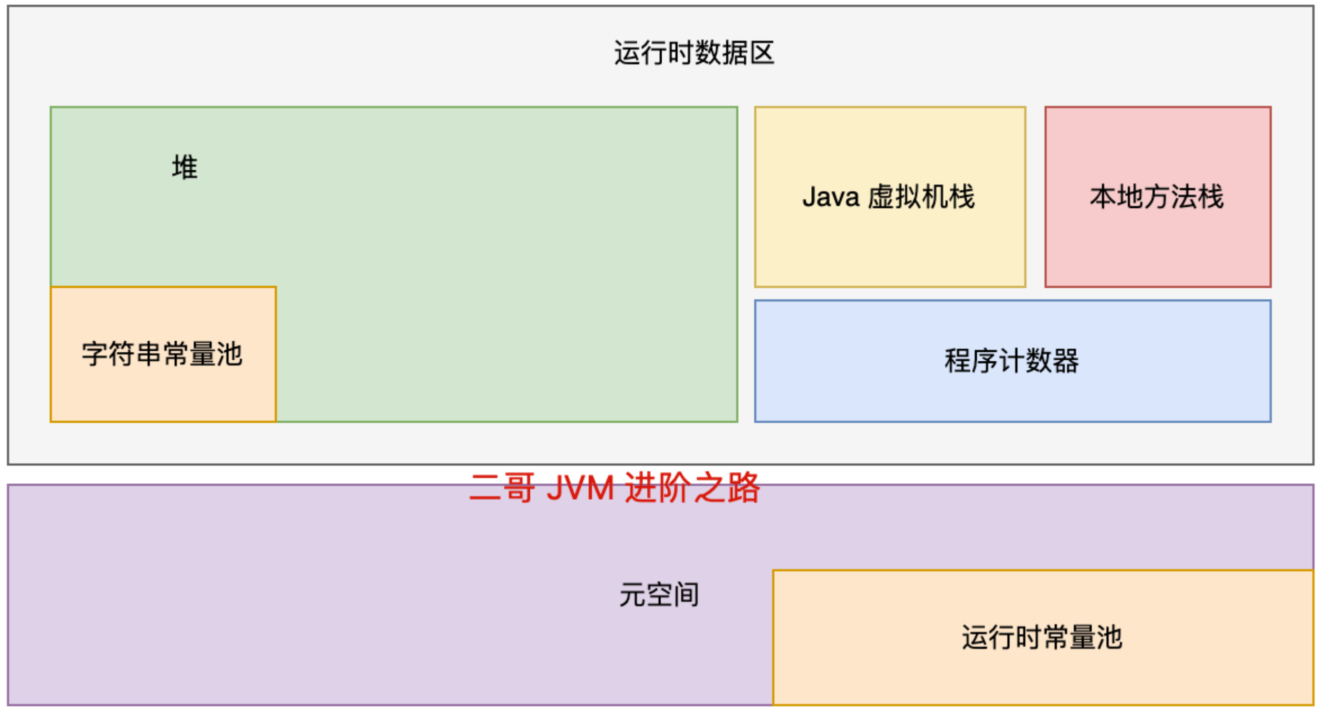
内存溢出和内存泄露

在 Java 中，和内存相关的问题主要有两种，内存溢出和内存泄漏。

- 内存溢出（Out Of Memory）：就是申请内存时，JVM 没有足够的内存空间。通俗说法就是去蹲坑发现坑位满了。
- 内存泄露 （Memory Leak）：就是申请了内存，但是没有释放，导致内存空间浪费。通俗说法就是有人占着茅坑不拉屎。

内存溢出

在 JVM 的内存区域中，除了程序计数器，其他的内存区域都有可能发生内存溢出。



大家都知道，Java 堆中存储的都是对象，或者叫对象实例，那只要我们不断地创建对象，并且保证 GC Roots 到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么就一定会产生内存溢出。

比如说运行下面这段代码：

```
public class OOM {
    public static void main(String[] args) {
        List<Object> list = new ArrayList<>();
        while (true) {
            list.add(new Object());
        }
    }
}
```

运行程序的时候记得设置一下 VM 参数：`-Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError`，限制堆内存大小为 20M，并且不允许扩展，并且当发生 OOM 时 dump 出当前内存的快照。

运行结果如下：

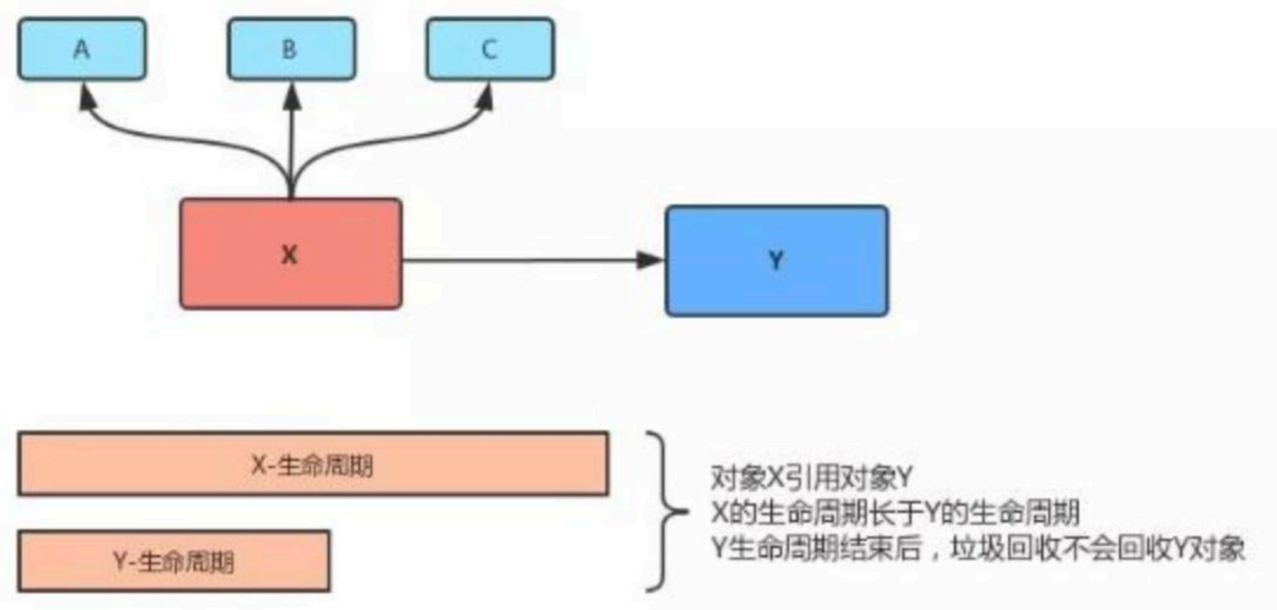
```
Run:   OOM ×
/Library/Java/JavaVirtualMachines/jdk1.8.0_301.jdk/Contents/Home/bin/java ...
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid57657.hprof ...
Heap dump file created [28880052 bytes in 0.127 secs]
Exception in thread "main" java.lang.OutOfMemoryError Create breakpoint : Java heap space
    at java.util.Arrays.copyOf(Arrays.java:3210)
    at java.util.Arrays.copyOf(Arrays.java:3181)
    at java.util.ArrayList.grow(ArrayList.java:267)
    at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:241)
    at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:233)
    at java.util.ArrayList.add(ArrayList.java:464)
    at com.github.paicoding.forum.test.javabetter.jvm.OOM.main(OOM.java:16)
```

我们在讲运行时数据区的时候也曾讲过。

内存泄露

内存泄露是指程序中己动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

简单来说，就是应该被垃圾回收的对象没有回收掉，导致占用的内存越来越多，最终导致内存溢出。



在上图中：对象 X 引用对象 Y，X 的生命周期比 Y 的生命周期长，Y 生命周期结束的时候，垃圾回收器不会回收对象 Y。

来看下面的例子：

```
public class MemoryLeak {
    public static void main(String[] args) {
      try{
          Connection conn =null;
          Class.forName("com.mysql.jdbc.Driver");
          conn =DriverManager.getConnection("url","","");
          Statement stmt =conn.createStatement();
```

```
            ResultSet rs =stmt.executeQuery("....");
        } catch (Exception e) {//异常日志
        } finally {
          // 1. 关闭结果集 Statement
          // 2. 关闭声明的对象 ResultSet
          // 3. 关闭连接 Connection
      }
    }
}
```

创建的连接不再使用时，需要调用 close 方法关闭连接，只有连接被关闭后，GC 才会回收对应的对象（Connection，Statement，ResultSet，Session）。忘记关闭这些资源会导致持续占有内存，无法被 GC 回收。

这样就会导致内存泄露，最终导致内存溢出。

换句话说，内存泄露不是内存溢出，但会加快内存溢出的发生。

内存溢出后的表象

之前生产环境爆出的内存溢出问题会随着业务量的增长，出现的频次也越来越高。

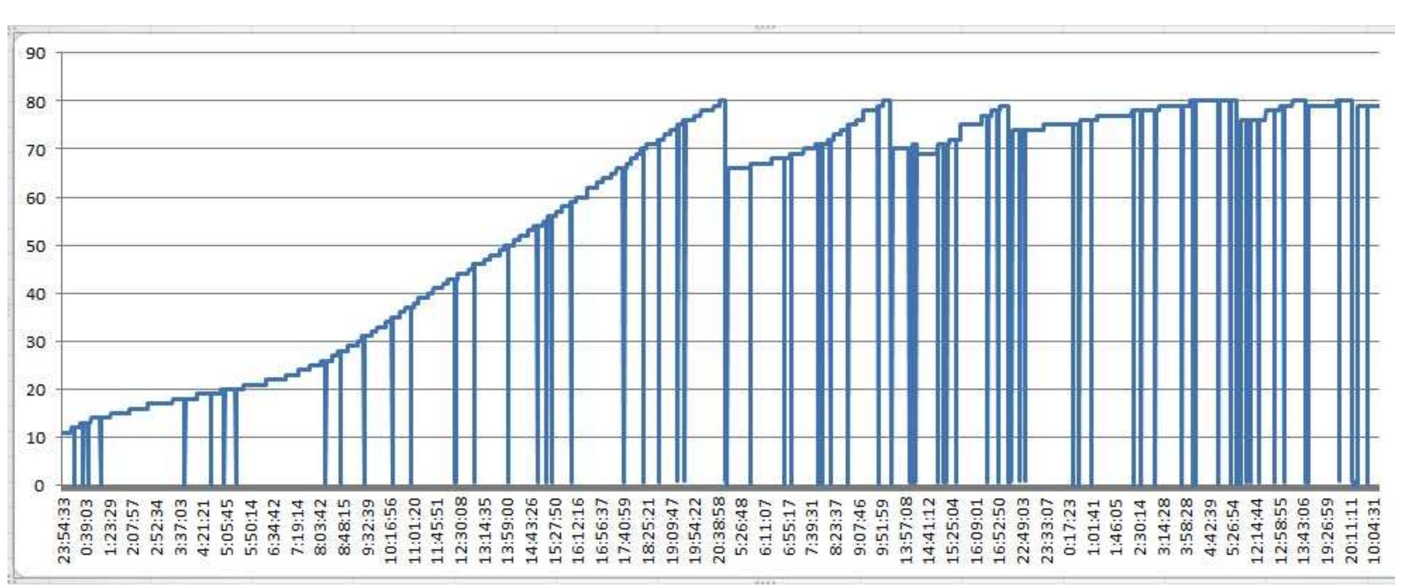
应用程序的业务逻辑非常简单，就是从 Kafka 中将数据消费下来，然后批量的做持久化操作。

OOM 现象则是随着 Kafka 的消息越多，出现异常的频次就越快。由于当时还有其他工作所以只能让运维做重启，并且监控好堆内存以及 GC 情况。

> 不得不说，重启大法真的好，能解决大量的问题，但不是长久之计。

内存泄露的排查

于是我们想根据运维之前收集到的内存数据、GC 日志尝试判断哪里出现了问题。



结果发现老年代的内存使用就算是发生 GC 也一直居高不下，而且随着时间推移也越来越高。

结合 jstat 的日志发现就算是发生了 FGC，老年代也回收不了，内存已经到顶。

0.00	0.00	54.22	99.95	34.04	770	102.637	19	77.008	179.645
0.00	0.00	65.42	99.95	34.04	770	102.637	19	77.008	179.645
0.00	0.00	78.18	99.95	34.04	770	102.637	19	77.008	179.645
0.00	0.00	89.34	99.95	34.04	770	102.637	19	77.008	179.645
0.00	0.00	99.40	99.95	34.04	770	102.637	19	77.008	179.645
0.00	0.00	14.73	100.00	34.09	770	102.637	20	90.137	192.774
0.00	0.00	26.05	100.00	34.09	770	102.637	20	90.137	192.774
0.00	0.00	37.22	100.00	34.09	770	102.637	20	90.137	192.774
0.00	0.00	48.88	100.00	34.09	770	102.637	20	90.137	192.774
0.00	0.00	57.56	100.00	34.09	770	102.637	20	90.137	192.774
0.00	0.00	69.07	100.00	34.09	770	102.637	20	90.137	192.774
0.00	0.00	78.45	100.00	34.09	770	102.637	20	90.137	192.774

甚至有几台应用 FGC 达到了上百次，时间也高的可怕。

这说明应用的内存使用肯定是有问题的，有许多赖皮对象始终回收不掉。

内存泄露的定位

由于生产上的内存 dump 文件非常大，达到了几十 G。也和我们生产环境配置的内存太大有关。

所以导致想使用 MAT 分析需要花费大量时间。

> MAT 是 Eclipse 的一个插件，也可以单独使用，可以用来分析 Java 的堆内存，找出内存泄露的原因。

因此我们就想是否可以在本地复现，这样就好定位的多。

为了尽快的复现问题，我将本地应用最大堆内存设置为 150M。然后在消费 Kafka 那里 Mock 了一个 while 循环一直不断的生成数据。

同时当应用启动之后利用 VisualVM 连上应用实时监控内存、GC 的使用情况。

结果跑了 10 几分钟内存使用并没有什么问题。根据图中可以看出，每一次 GC 内存都能有效的回收，所以并没有复现问题。

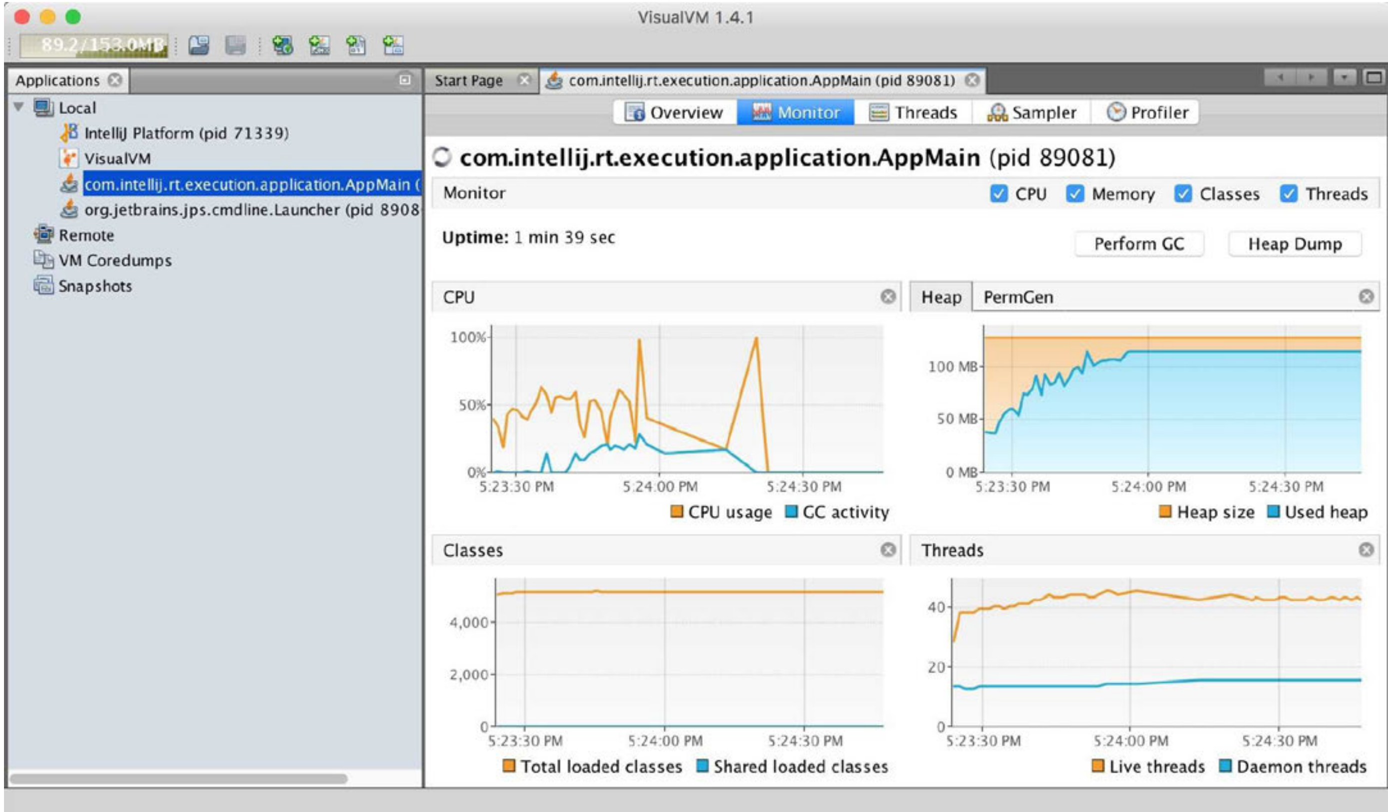
没法复现问题就很难定位。于是我们就采用了一种古老的方法——review 代码，发现生产的逻辑和我们用 while 循环 Mock 的数据还不太一样。

> 果然 review 代码是保障程序性能的第一道防线，诚不欺我。大家在写完代码的时候，尽量也要团队 review 一次。

后来查看生产日志发现每次从 Kafka 中取出的都是几百条数据，而我们 Mock 时每次只能产生**一条**。

为了尽可能的模拟生产情况便在服务器上跑了一个生产者程序，一直源源不断的向 Kafka 中发送数据。

果然不出意外只跑了一分多钟内存就顶不住了，观察下图发现 GC 的频次非常高，但是内存的回收却是相形见拙。

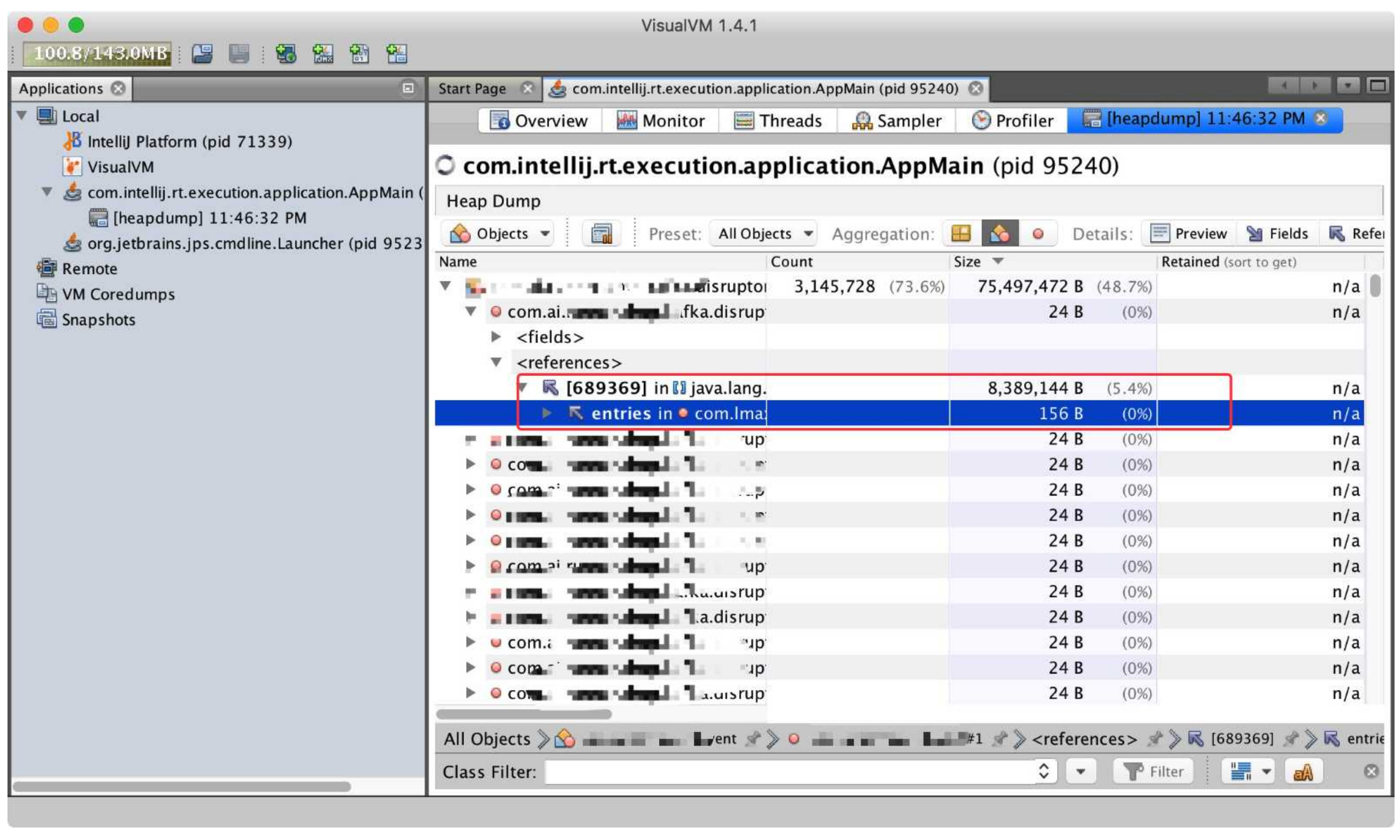


同时后台也开始打印内存溢出了，这样便复现出了问题。

内存泄露的解决

从目前的表现来看，就是内存中有许多对象一直存在强引用关系导致得不到回收。

于是便想看看到底是什么对象占用了这么多的内存，利用 VisualVM 的 HeapDump 功能，就可以立即 dump 出当前应用的内存情况。



结果发现 `com.lmax.disruptor.RingBuffer` 类型的对象占用了将近 50% 的内存。

看到这个包自然就想到了 `Disruptor` 环形队列了。

> Disruptor 是一个高性能的异步处理框架，它的核心思想是：通过无锁的方式来实现高性能的并发处理，其性能是高于 JDK 的 BlockingQueue 的。

再次 review 代码发现：从 Kafka 里取出的 700 条数据是直接往 Disruptor 里丢的。

这里也就能说明为什么第一次模拟数据没复现问题了。

模拟的时候是一个对象放进队列里，而生产的情况是 700 条数据放进队列里。这个数据量就是 700 倍的差距啊。

而 Disruptor 作为一个环形队列，在对象没有被覆盖之前是一直存在的。

我也做了一个实验，证明确实如此。

我设置队列大小为 8 ，从 0~9 往里面写 10 条数据，当写到 8 的时候就会把之前 0 的位置覆盖掉，后面的以此类推（类似于 HashMap 的取模定位）。

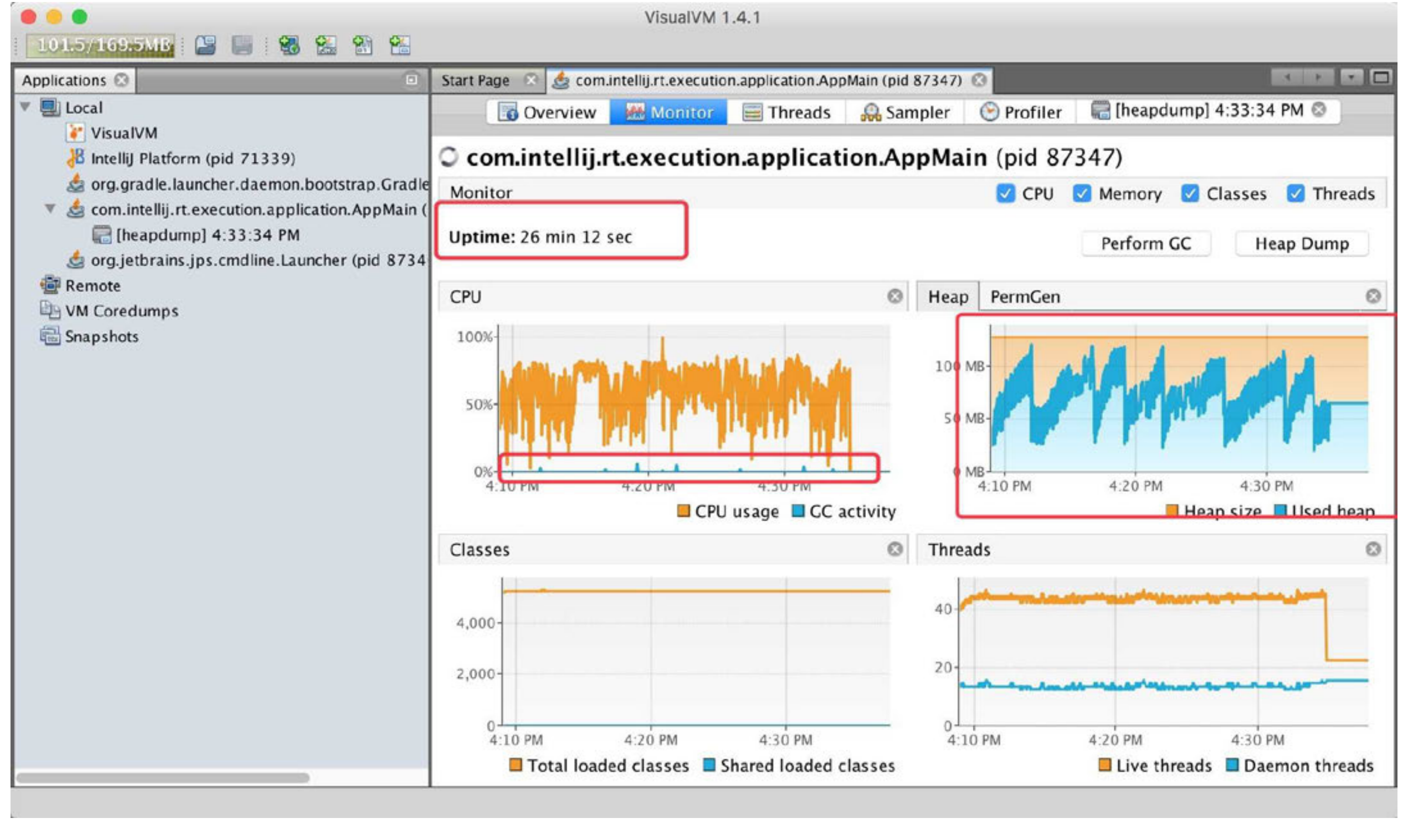
所以在生产环境上，假设我们的队列大小是 1024，那么随着系统的运行最终会导致 1024 个位置上装满了对象，而且每个位置都是 700 个！

于是查看了生产环境上 Disruptor 的 RingBuffer 配置，结果是：`1024*1024`。

这个数量级就非常吓人了。

为了验证是否是这个问题，我在本地将该值设为 2 ，一个最小值试试。

同样的 128M 内存，也是通过 Kafka 一直源源不断的取出数据。通过监控如下：



跑了 20 几分钟系统一切正常，每当一次 GC 都能回收大部分内存，最终呈现锯齿状。

这样问题就找到了，不过生产上这个值具体设置多少还得根据业务情况测试才能知道，但原有的 1024*1024 是绝对不能再使用了。

小结

虽然到了最后也就改了一行代码(还没改，直接修改配置)，但这个排查过程我觉得是很有意义的。

也会让大部分觉得 JVM 这样的黑盒难以下手的球友有一个直观感受。

```
同时也得感叹 Disruptor 东西虽好，也不能乱用哦！
```

相关演示代码查看：

> https://github.com/crossoverJie/JCSprout/tree/master/src/main/java/com/crossoverjie/disruptor

- 参考链接 1：内存泄露的排查
- 参考链接 2：内存溢出和内存泄露

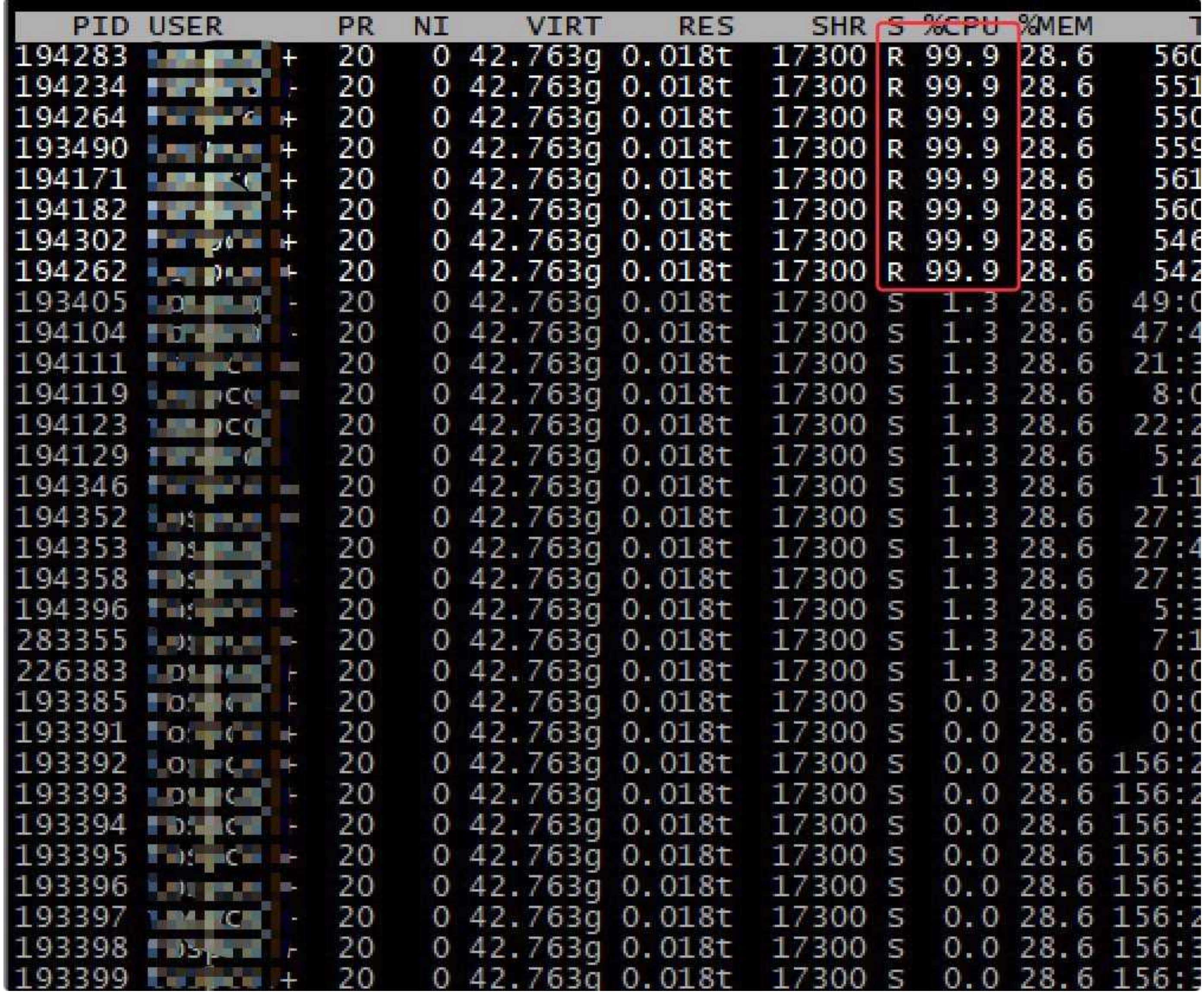
第十八节：CPU 100%排查优化实战

前面给大家讲过一次 OOM 的优化排查实战，今天再给大家讲一个 CPU 100% 优化排查实战。

收到运维同学的报警，说某些服务器负载非常高，让我们开发定位问题。拿到问题后先去服务器上看了看，发现运行的只有我们的 Java 应用程序。于是先用 `ps` 命令拿到了应用的 `PID`。

> ps：查看进程的命令；PID：进程 ID。`ps -ef | grep java` 可以查看所有的 Java 进程。前面也曾讲过。

接着使用 `top -Hp pid` 将这个进程的线程显示出来。输入大写 P 可以将线程按照 CPU 使用比例排序，于是得到以下结果。



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	T
194283		20	0	42.763g	0.018t	17300	R	99.9	28.6	560
194234		20	0	42.763g	0.018t	17300	R	99.9	28.6	551
194264		20	0	42.763g	0.018t	17300	R	99.9	28.6	550
193490		20	0	42.763g	0.018t	17300	R	99.9	28.6	559
194171		20	0	42.763g	0.018t	17300	R	99.9	28.6	561
194182		20	0	42.763g	0.018t	17300	R	99.9	28.6	560
194302		20	0	42.763g	0.018t	17300	R	99.9	28.6	546
194262		20	0	42.763g	0.018t	17300	R	99.9	28.6	542
193405		20	0	42.763g	0.018t	17300	S	1.3	28.6	49:0
194104		20	0	42.763g	0.018t	17300	S	1.3	28.6	47:4
194111		20	0	42.763g	0.018t	17300	S	1.3	28.6	21:3
194119		20	0	42.763g	0.018t	17300	S	1.3	28.6	8:0
194123		20	0	42.763g	0.018t	17300	S	1.3	28.6	22:2
194129		20	0	42.763g	0.018t	17300	S	1.3	28.6	5:2
194346		20	0	42.763g	0.018t	17300	S	1.3	28.6	1:1
194352		20	0	42.763g	0.018t	17300	S	1.3	28.6	27:3
194353		20	0	42.763g	0.018t	17300	S	1.3	28.6	27:4
194358		20	0	42.763g	0.018t	17300	S	1.3	28.6	27:3
194396		20	0	42.763g	0.018t	17300	S	1.3	28.6	5:3
283355		20	0	42.763g	0.018t	17300	S	1.3	28.6	7:1
226383		20	0	42.763g	0.018t	17300	S	1.3	28.6	0:0
193385		20	0	42.763g	0.018t	17300	S	0.0	28.6	0:0
193391		20	0	42.763g	0.018t	17300	S	0.0	28.6	0:0
193392		20	0	42.763g	0.018t	17300	S	0.0	28.6	156:2
193393		20	0	42.763g	0.018t	17300	S	0.0	28.6	156:2
193394		20	0	42.763g	0.018t	17300	S	0.0	28.6	156:3
193395		20	0	42.763g	0.018t	17300	S	0.0	28.6	156:3
193396		20	0	42.763g	0.018t	17300	S	0.0	28.6	156:3
193397		20	0	42.763g	0.018t	17300	S	0.0	28.6	156:2
193398		20	0	42.763g	0.018t	17300	S	0.0	28.6	156:3
193399		20	0	42.763g	0.018t	17300	S	0.0	28.6	156:3

果然，某些线程的 CPU 使用率非常高，99.9% 可不是非常高嘛（😂）。

为了方便问题定位，我立马使用 `jstack pid > pid.log` 将线程栈 `dump` 到日志文件中。关于 [jstack](#) 命令，我们前面刚刚讲过。

我在上面 99.9% 的线程中随机选了一个 `pid=194283` 的，转换为 16 进制（2f6eb）后在线程快照中查询：

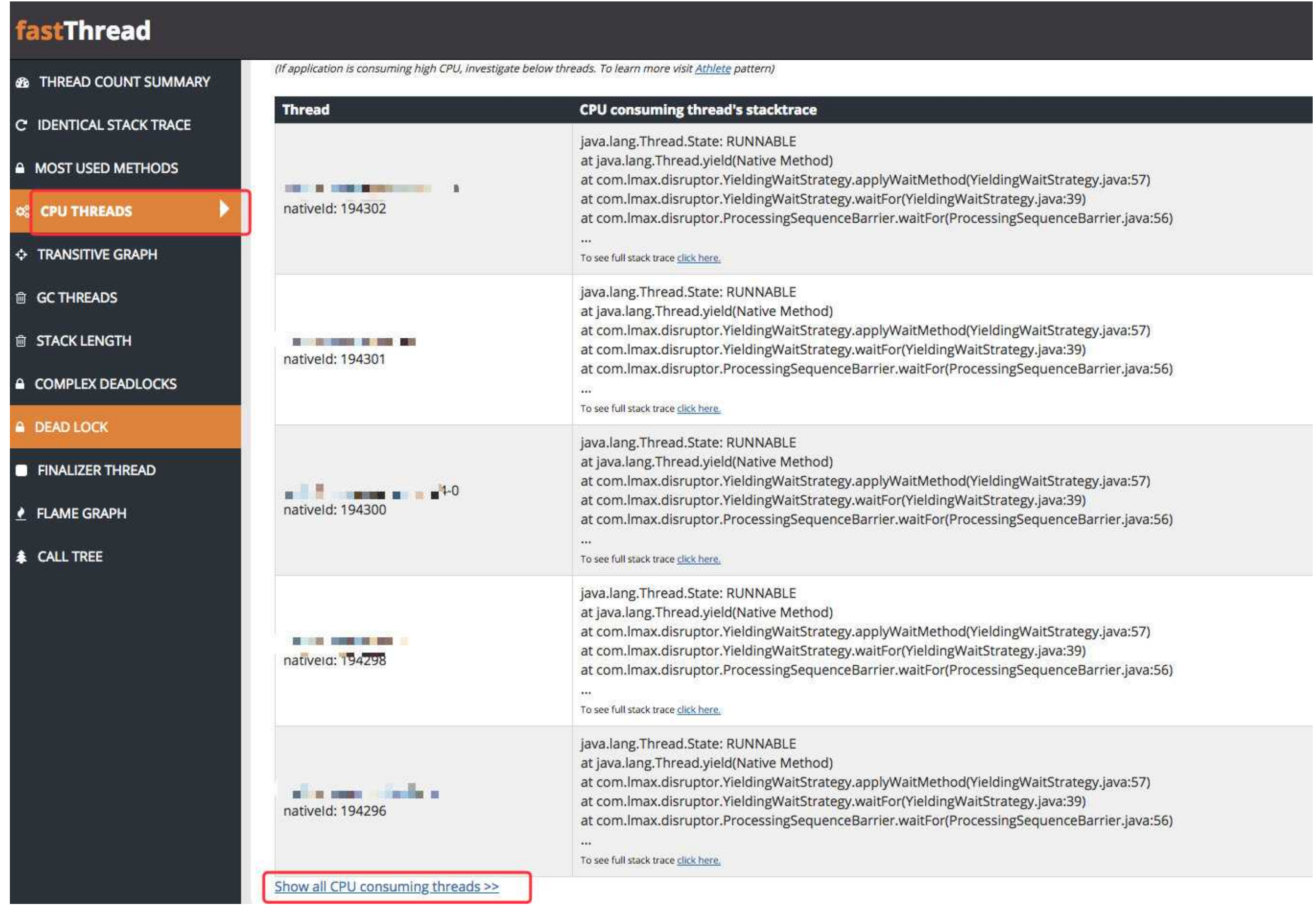
```
"(          23-0" prio=10 tid=0x00007f0fd0026000 nid=0x2f6eb runnable [0x00007f0ec3a7a000]
   java.lang.Thread.State: RUNNABLE
    at java.lang.Thread.yield(Native Method)
    at com.lmax.disruptor.YieldingWaitStrategy.applyWaitMethod(YieldingWaitStrategy.java:57)
    at com.lmax.disruptor.YieldingWaitStrategy.waitFor(YieldingWaitStrategy.java:39)
    at com.lmax.disruptor.ProcessingSequenceBarrier.waitFor(ProcessingSequenceBarrier.java:56)
    at com.lmax.disruptor.BatchEventProcessor.run(BatchEventProcessor.java:128)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)
```

> 线程快照中线程 ID 都是16进制的。

发现这是 `Disruptor` 的一个堆栈，好家伙，这不前面刚遇到过嘛，老熟人啊，[强如 Disruptor 也发生内存溢出？](#)

真没想到，再来一次！

为了更加直观的查看线程的状态，我将快照信息上传到了专门的分析平台上：http://fastthread.io/，估计有球友用过。



其中有一项展示了所有消耗 CPU 的线程，我仔细看了下，发现几乎都和上面的堆栈一样。

也就是说，都是 `Disruptor` 队列的堆栈，都在执行 `java.lang.Thread.yield`。

众所周知，`yield` 方法会暗示当前线程让出 `CPU` 资源，让其他线程来竞争（多线程的时候我们讲过 yield，相信大家还有印象）。

根据刚才的线程快照发现，处于 `RUNNABLE` 状态并且都在执行 `yield` 的线程大概有 30几个。

初步判断，大量线程执行 `yield` 之后，在互相竞争导致 CPU 使用率增高，通过对堆栈的分析可以发现，确实和 `Disruptor` 有关。

好家伙，又是它。

既然如此，我们来大致看一下 `Disruptor` 的使用方式吧。看有多少球友使用过。

第一步，在 pom.xml 文件中引入 `Disruptor` 的依赖：

```
<dependency>
    <groupId>com.lmax</groupId>
    <artifactId>disruptor</artifactId>
    <version>3.4.2</version>
</dependency>
```

第二步，定义事件 LongEvent：

```
public static class LongEvent {
    private long value;

    public void set(long value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return "LongEvent{value=" + value + '}';
    }
}
```

第三步，定义事件工厂：

```
// 定义事件工厂
public static class LongEventFactory implements EventFactory<LongEvent> {
    @Override
    public LongEvent newInstance() {
        return new LongEvent();
    }
}
```

第四步，定义事件处理器：

```
// 定义事件处理器
public static class LongEventHandler implements EventHandler<LongEvent> {
    @Override
    public void onEvent(LongEvent event, long sequence, boolean endOfBatch) {
        System.out.println("Event: " + event);
    }
}
```

第五步，定义事件发布者：

```
public static void main(String[] args) throws InterruptedException {
    // 指定 Ring Buffer 的大小
    int bufferSize = 1024;

    // 构建 Disruptor
    Disruptor<LongEvent> disruptor = new Disruptor<>(
            new LongEventFactory(),
            bufferSize,
            Executors.defaultThreadFactory());

    // 连接事件处理器
```

```
    disruptor.handleEventsWith(new LongEventHandler());

    // 启动 Disruptor
    disruptor.start();

    // 获取 Ring Buffer
    RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();

    // 生产事件
    ByteBuffer bb = ByteBuffer.allocate(8);
    for (long l = 0; l < 100; l++) {
        bb.putLong(0, l);
        ringBuffer.publishEvent((event, sequence, buffer) ->
event.set(buffer.getLong(0)), bb);
        Thread.sleep(1000);
    }

    // 关闭 Disruptor
    disruptor.shutdown();
}
```

简单解释下：

- LongEvent：这是要通过 Disruptor 传递的数据或事件。
- LongEventFactory：用于创建事件对象的工厂类。
- LongEventHandler：事件处理器，定义了如何处理事件。
- Disruptor 构建：创建了一个 Disruptor 实例，指定了事件工厂、缓冲区大小和线程工厂。
- 事件发布：示例中演示了如何发布事件到 Ring Buffer。

大家可以运行看一下输出结果。

解决问题

我查了下代码，发现每一个业务场景在内部都会使用 2 个 `Disruptor` 队列来解耦。

假设现在有 7 个业务，那就等于创建了 `2*7=14` 个 `Disruptor` 队列，同时每个队列有一个消费者，也就是总共有 14 个消费者（生产环境更多）。

同时发现配置的消费等待策略为 `YieldingWaitStrategy`，这种等待策略会执行 yield 来让出 CPU。代码如下：

```java
public final class YieldingWaitStrategy implements WaitStrategy
{
    private static final int SPIN_TRIES = 100;

    @Override
    public long waitFor(
        final long sequence, Sequence cursor, final Sequence dependentSequence, final SequenceBarrier barrier)
        throws AlertException, InterruptedException
    {
        long availableSequence;
        int counter = SPIN_TRIES;

        while ((availableSequence = dependentSequence.get()) < sequence)
        {
            counter = applyWaitMethod(barrier, counter);
        }

        return availableSequence;
    }

    @Override
    public void signalAllWhenBlocking()
    {
    }

    private int applyWaitMethod(final SequenceBarrier barrier, int counter)
        throws AlertException
    {
        barrier.checkAlert();

        if (0 == counter)
        {
            Thread.yield();
        }
        else
        {
            --counter;
        }

        return counter;
    }
}
```

初步来看，和等待策略有很大的关系。

本地模拟

为了验证，我在本地创建了 15 个 `Disruptor` 队列，同时结合监控观察 CPU 的使用情况。

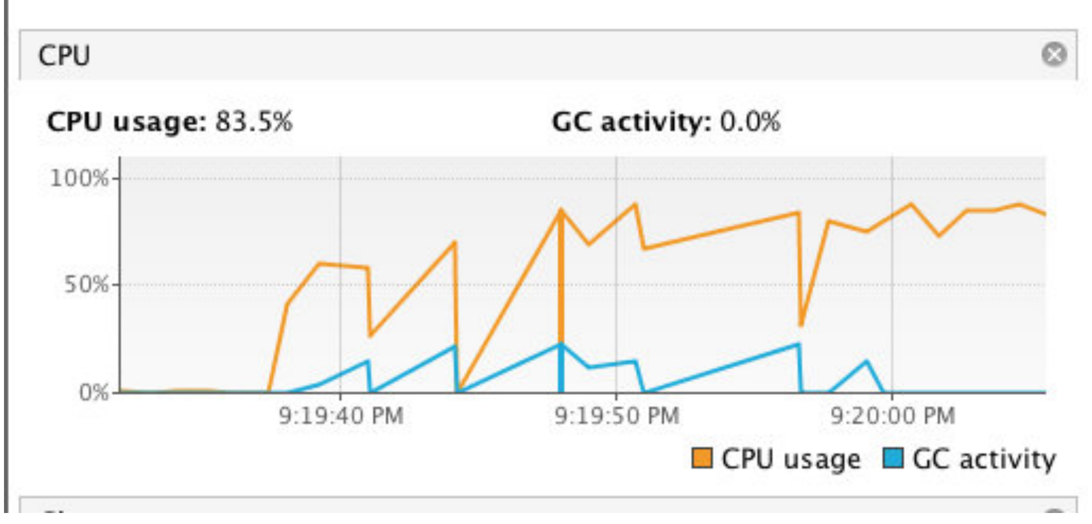
注意看代码 YieldingWaitStrategy：

```java
        for (int i = 0; i < 15; i++) {
            // Construct the Disruptor
            //Disruptor<LongEvent> disruptor = new Disruptor<>(factory, bufferSize, executor);
            Disruptor<LongEvent> disruptor = new Disruptor<>(factory, bufferSize, executor, ProducerType.SINGLE,
                    new YieldingWaitStrategy());

            // Connect the handler
            disruptor.handleEventsWith(new LongEventHandler());

            // Start the Disruptor, starts all threads running
            disruptor.start();

            // Get the ring buffer from the Disruptor to be used for publishing.
            RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();

            LongEventProducer producer = new LongEventProducer(ringBuffer);

            for (long l = 0; l < 1000000; l++) {
                //producer.onData(l);
                //Thread.sleep(1000);
                productExecutor.execute(new Work(producer,l));
            }

        }
```

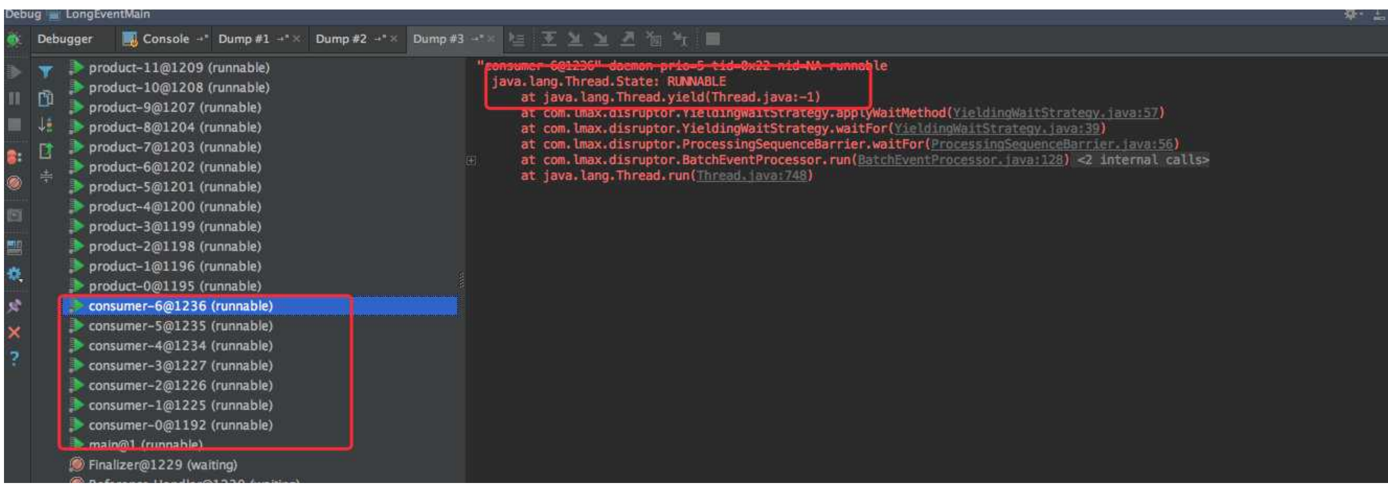
以及事件处理器：

```java
public class LongEventHandler implements EventHandler<LongEvent> {
    private final static Logger LOGGER = LoggerFactory.getLogger(LongEventHandler.class);
    @Override
    public void onEvent(LongEvent event, long sequence, boolean endOfBatch) throws InterruptedException {
        LOGGER.info("消费 Event=[{}]",event.getValue()) ;
        //Thread.sleep(1000);
    }
}
```

创建了 15 个 `Disruptor` 队列，同时每个队列都用线程池来往 `Disruptor队列` 里面发送 100W 条数据。消费程序仅仅只是打印一下。



跑了一段时间，发现 CPU 使用率确实很高。



同时 `dump` 线程发现和生产环境中的现象也是一致的：消费线程都处于 `RUNNABLE` 状态，同时都在执行 `yield`。

通过查询 `Disruptor` 官方文档发现：

YieldingWaitStrategy

The YieldingWaitStrategy is one of 2 Wait Strategies that can be use in low latency systems, where there is the option to burn CPU cycles with the goal of improving latency. The YieldingWaitStrategy will busy spin waiting for the sequence to increment to the appropriate value. Inside the body of the loop `Thread.yield()` will be called allowing other queued threads to run. This is the recommended wait strategy when need very high performance and the number of Event Handler threads is less than the total number of logical cores, e.g. you have hyper-threading enabled.

YieldingWaitStrategy 是一种充分压榨 CPU 的策略，使用`自旋 + yield`的方式来提高性能。当消费线程（Event Handler threads）的数量小于 CPU 核心数时推荐使用该策略。

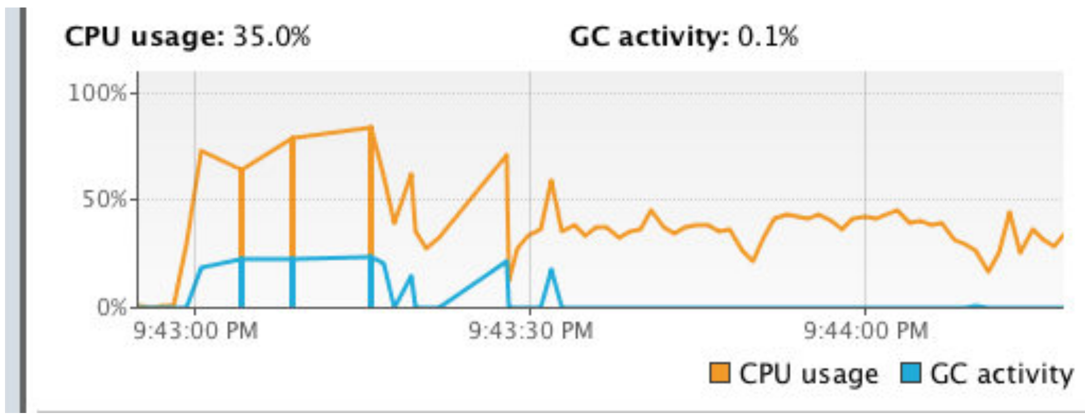
The default wait strategy used by the Disruptor is the BlockingWaitStrategy. Internally the BlockingWaitStrategy uses a typical lock and condition variable to handle thread wake-up. The BlockingWaitStrategy is the slowest of the available wait strategies, but is the most conservative with the respect to CPU usage and will give the most consistent behaviour across the widest variety of deployment options. However, again knowledge of the deployed system can allow for additional performance.

同时查到其他的等待策略，比如说 `BlockingWaitStrategy` （也是默认的策略），使用的是锁的机制，对 CPU 的使用率不高。

于是我将等待策略调整为 `BlockingWaitStrategy`。

```java
    for (int i = 0; i < 15; i++) {
        // Construct the Disruptor
        //Disruptor<LongEvent> disruptor = new Disruptor<>(factory, bufferSize, executor);
        Disruptor<LongEvent> disruptor = new Disruptor<>(factory, bufferSize, executor, ProducerType.SINGLE,
                new BlockingWaitStrategy());

        // Connect the handler
        disruptor.handleEventsWith(new LongEventHandler());

        // Start the Disruptor, starts all threads running
        disruptor.start();

        // Get the ring buffer from the Disruptor to be used for publishing.
        RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();

        LongEventProducer producer = new LongEventProducer(ringBuffer);

        for (long l = 0; l < 1000000; l++) {
            //producer.onData(l);
            //Thread.sleep(1000);
            productExecutor.execute(new Work(producer,l));
        }

    }
```

运行后的结果如下：



和刚才的结果对比，发现 CPU 的使用率有明显的降低；同时 dump 线程后，发现大部分线程都处于 waiting 状态。

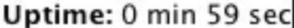
优化解决

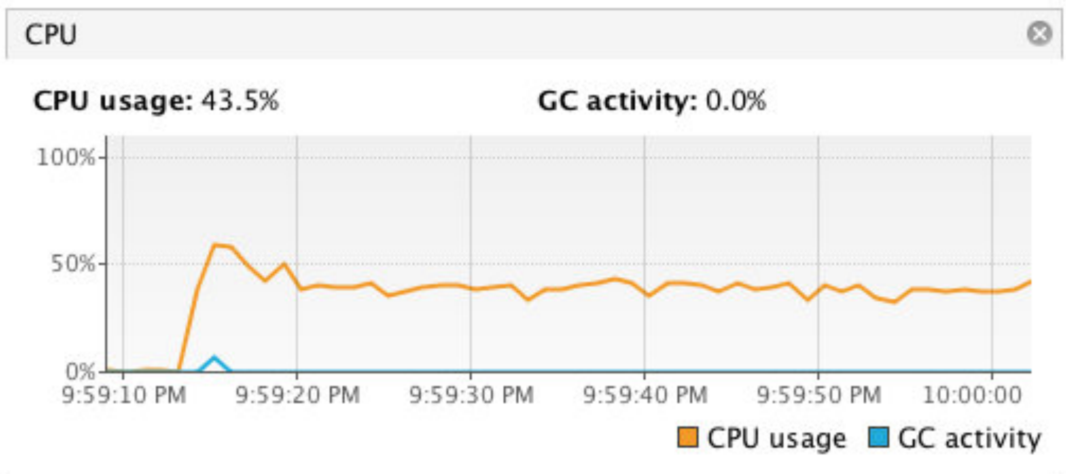
看样子，将等待策略换为 `BlockingWaitStrategy` 可以减缓 CPU 的使用，不过我留意到官方对 `YieldingWaitStrategy` 的描述是这样的：
当消费线程（Event Handler threads）的数量小于 CPU 核心数时推荐使用该策略。

而现在的使用场景是，消费线程数已经大大的超过了核心 CPU 数，因为我的使用方式是一个 `Disruptor` 队列一个消费者，所以我将队列调整为 1 个又试了试(策略依然是 `YieldingWaitStrategy`)。

```java
for (int i = 0; i < 1; i++) {
    // Construct the Disruptor
    //Disruptor<LongEvent> disruptor = new Disruptor<>(factory, bufferSize, executor);
    Disruptor<LongEvent> disruptor = new Disruptor<>(factory, bufferSize, executor, ProducerType.SINGLE,
            new YieldingWaitStrategy());

    // Connect the handler
    disruptor.handleEventsWith(new LongEventHandler());

    // Start the Disruptor, starts all threads running
    disruptor.start();

    // Get the ring buffer from the Disruptor to be used for publishing.
    RingBuffer<LongEvent> ringBuffer = disruptor.getRingBuffer();

    LongEventProducer producer = new LongEventProducer(ringBuffer);

    for (long l = 0; l < 1000000; l++) {
        //producer.onData(l);
        //Thread.sleep(1000);
        productExecutor.execute(new Work(producer,l));
    }
```

查看运行效果：





跑了一分钟，发现 CPU 的使用率一直都比较平稳。

小结

排查到此，可以得出结论了，想要根本解决这个问题需要将我们现有的业务拆分；现在是一个应用里同时处理了 N 个业务，每个业务都会使用好几个 `Disruptor` 队列。

由于在一台服务器上运行，所以就会导致 CPU 的使用率居高不下。

由于是老系统，所以我们的调整方式如下：

- 先将等待策略调整为 `BlockingWaitStrategy`，可以有效降低 CPU 的使用率（业务上也还能接受）。
- 第二步就需要将应用拆分，一个应用处理一种业务类型；然后分别部署，这样可以互相隔离互不影响。

当然还有一些其他的优化，比如说这次 dump 发现应用程序创建了 800+ 个线程。创建线程池的方式也是核心线程数和最大线程数一样，就导致一些空闲的线程得不到回收。应该将创建线程池的方式调整一下，将线程数降下来，尽量物尽其用。

好，生产环境中，一般也就是会遇到 OOM 和 CPU 这两个问题，那也希望这种排查思路能够给大家一些启发~

> - 演示代码已上传至 GitHub：https://github.com/crossoverJie/JCSprout

> - 参考链接：crossoverJie 的CPU 100% 排查

第十九节：JVM 核心知识点总结

一、基本概念

1.1 OpenJDK

自 1996 年 `JDK 1.0` 发布以来，Sun 公司在大版本上发行了 `JDK 1.1`、`JDK 1.2`、`JDK 1.3`、`JDK 1.4`、`JDK 5`，`JDK 6` ，这些版本的 JDK 都可以统称为 SunJDK 。

之后在 2006 年的 JavaOne 大会上，Sun 公司宣布将 Java 开源，在随后的一年多里，它陆续将 JDK 的各个部分在 GPL v2（GNU General Public License，version 2）协议下开源，并建立了 OpenJDK 组织来对这些代码进行独立的管理，这就是 OpenJDK 的来源，此时的 OpenJDK 拥有当时 sunJDK 7 的几乎全部代码。

1.2 OracleJDK

在 JDK 7 的开发期间，由于各种原因的影响，Sun 公司市值一路下跌，已无力推进 JDK 7 的开发，于是 JDK 7 的发布一直被推迟。

之后在 2009 年 Sun 公司被 Oracle 公司收购，为解决 JDK 7 长期跳票的问题，Oracle 将 JDK 7 中大部分未能完成的项目推迟到 JDK 8 ，并于 2011 年发布了JDK 7，在这之后由 Oracle 公司正常发行的 JDK 版本就由 SunJDK 改称为 Oracle JDK。

在 2017 年 JDK 9 发布后，Oracle 公司宣布：以后 JDK 将会在每年的 3 月和 9 月各发布一个大版本，即半年发行一个大版本，目的是为了避免众多功能被捆绑到一个 JDK 版本上而引发的无法交付的风险。

在 JDK 11 发布后，Oracle 同步调整了 JDK 的商业授权，宣布从 JDK 11 起，将以前的商业特性全部开源给 OpenJDK ，这样 OpenJDK 11 和 OracleJDK 11 的代码和功能，在本质上就完全相同了。

同时还宣布以后会发行两个版本的 JDK ：

- 一个是在 GPLv2 + CE 协议下由 Oracle 开源的 OpenJDK；
- 一个是在 OTN 协议下正常发行的 OracleJDK。

两者共享大部分源码，在功能上几乎一致。唯一的区别是 Oracle OpenJDK 可以在开发、测试或者生产环境中使用，但只有半年的更新支持；而 OracleJDK 对个人免费，但在生产环境中商用收费，可以有三年时间的更新支持。

目前最新的长期支持的 JDK 是 JDK 21（LTS），详情可以参考朋友 why 技术的帖子。

1.3 HotSpot VM

它是 Sun/Oracle JDK 和 OpenJDK 中默认的虚拟机，也是目前使用最为广泛的虚拟机。

最初由 Longview Technologies 公司设计发明，该公司在 1997 年被 Sun 公司收购，随后 Sun 公司在 2006 年开源 SunJDK 时也将 HotSpot 虚拟机一并进行了开源。

Oracle 收购 Sun 以后，建立了 HotRockit 项目，并将其收购的另外一家公司（BEA）的 JRockit 虚拟机中的优秀特性集成到 HotSpot 中。

HotSpot 在这个过程里移除掉永久代，并吸收了 JRockit 的 Java Mission Control 监控工具等功能。

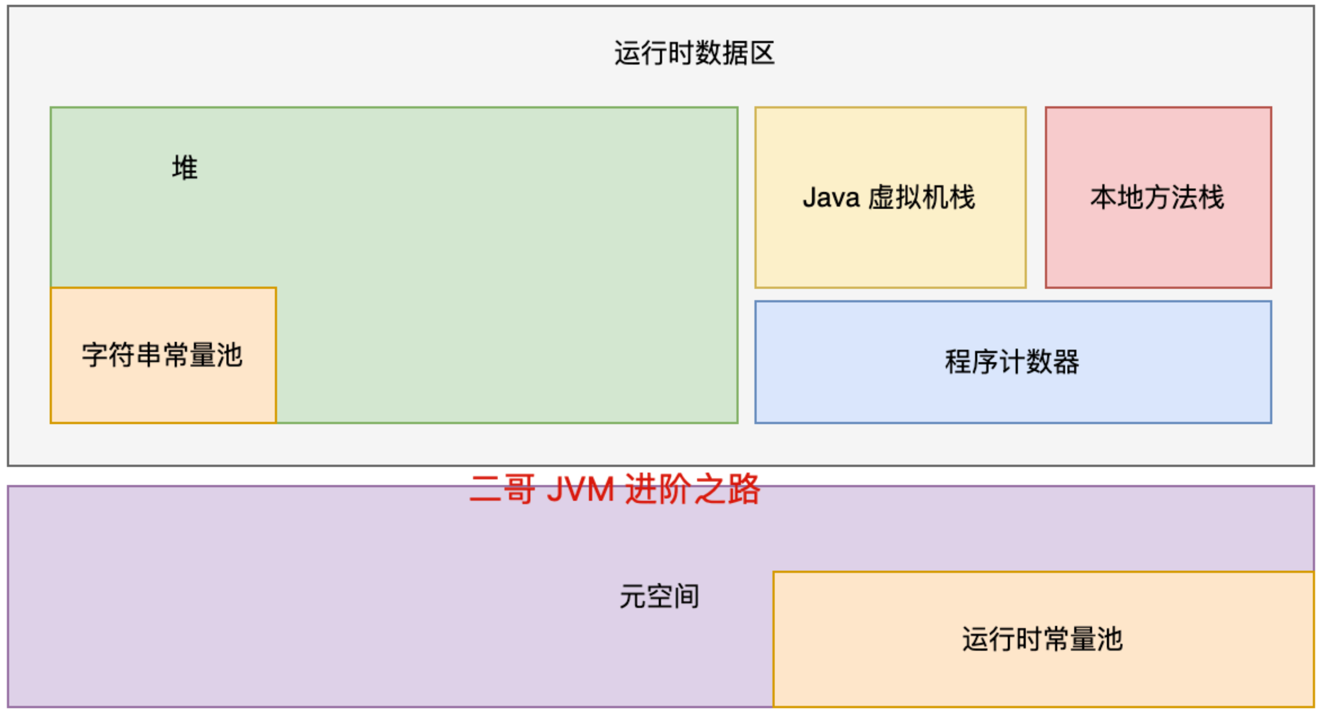
到 JDK 8 发行时，采用的就是集两者之长的 HotSpot VM。

我们可以在自己的电脑上使用 `java -version` 来获得 JDK 的信息：

```
~ git:(master) ±132 (0.252s)
java -version

java version "1.8.0_301"
Java(TM) SE Runtime Environment (build 1.8.0_301-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.301-b09, mixed mode)
```

二、Java 内存区域

Java 内存区域我们之前讲过，这里再盘一盘。



2.1 程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它可以看做是当前线程所执行的字节码的行号指示器。

字节码解释器通过改变程序计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要该计数器来完成。

每个线程都拥有一个独立的程序计数器，各个线程之间的计数器互不影响，独立存储。

2.2 虚拟机栈

虚拟机栈（Java Virtual Machine Stack）也是线程私有，它描述的是 Java 方法执行的线程内存模型：每个方法被执行的时候，Java 虚拟机都会同步创建一个栈帧，用于存储局部变量表、操作数栈、动态连接、方法出口等信息。

方法从调用到结束就对应着一个栈帧从入栈到出栈的过程。在《Java 虚拟机规范》中，对该内存区域规定了两类异常：

- 如果线程请求的栈深度大于虚拟机所允许的栈深度，将抛出 `StackOverflowError` 异常；
- 如果 Java 虚拟机栈的容量允许动态扩展，当栈扩展时如果无法申请到足够的内存会抛出 `OutOfMemoryError` 异常。

2.3 本地方法栈

本地方法栈（Native Method Stacks）与虚拟机栈类似，其区别在于：Java 虚拟机栈是为虚拟机执行 Java 方法（也就是字节码）服务的，而本地方法栈则是为 JVM 使用到的本地（Native）方法服务。

2.4 堆

堆（Java Heap）是虚拟机所管理的最大一块内存空间，它被所有线程所共享，用于存放对象实例。

Java 堆可以处于物理上不连续的内存空间中，但在逻辑上它应该被视为是连续的。Java 堆可以被实现成固定大小的，也可以是可扩展的。

当前大多数主流的虚拟机都是按照可扩展来实现的，即可以通过最大值参数 `-Xmx` 和最小值参数 `-Xms` 进行设定。

如果 Java 堆中没有足够的内存来完成对象实例分配，并且堆也无法再扩展时，Java 虚拟机将会抛出 `OutOfMemoryError` 异常。

2.5 方法区

方法区（Method Area）也是各个线程共享的内存区域，用于存储已被虚拟机加载的类信息、常量、静态变量、JIT 编译后的代码缓存等数据。

方法区也被称为 “非堆”，目的是与 Java 堆进行区分。《Java 虚拟机规范》规定，如果方法区无法满足新的内存分配需求时，将会抛出 `OutOfMemoryError` 异常。

JDK 8 以后的方法区实现已经不再是永久代（Permanent Generation）了，而是使用元空间（Metaspace）来实现。

运行时常量池（Runtime Constant Pool）也是方法区的一部分，用于存放常量池表（Constant Pool Table），常量池表中存放了编译期生成的各种符号字面量和符号引用。

> JDK 8 以后的运行时常量池在元空间中。

三、对象

3.1 对象的创建

当我们在代码中使用 `new` 关键字创建一个对象时，其在 JVM 中需要经过以下步骤：

1. 类加载过程

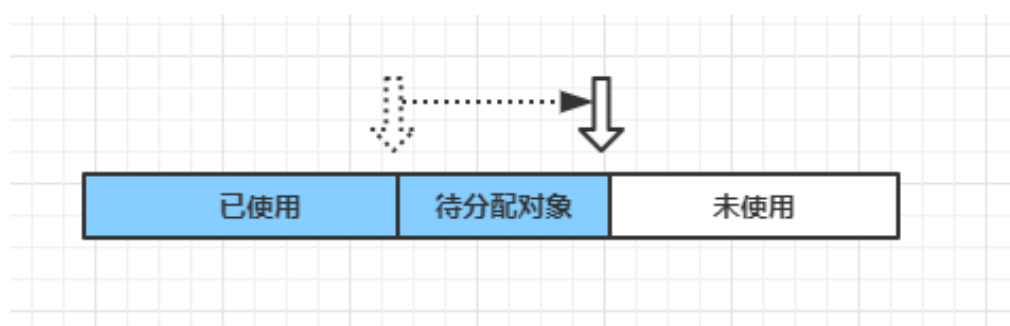
当虚拟机遇到一条字节码指令 `new` 时，首先将去检查这个指令的参数是否能在常量池中定位到一个符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，就必须先执行相应的类加载过程。

2. 分配内存

在类加载检查通过后，虚拟机需要给新生对象分配内存空间。根据 Java 堆是否规整，可以有以下两种分配方案：

①、**指针碰撞**：假设 Java 堆中内存是绝对规整的，所有使用的内存放在一边，所有未被使用的内存放在另外一边，中间以指针作为分界点指示器。

此时内存分配只是将指针向空闲方向偏移出对象大小的空间即可，这种方式被称为指针碰撞。



②、**空闲列表**：如果 Java 堆不是规整的，此时虚拟机需要维护一个列表，记录哪些内存块是可用的，哪些是不可用的。在进行内存分配时，只需要从该列表中选取出一块足够的内存空间划分给对象实例即可。

> 注：Java 堆是否规整取决于其采用的垃圾收集器是否带有空间压缩整理能力，前面讲过了。

除了分配方式外，由于对象创建在虚拟机中是一个非常频繁的行为，此时需要保证在并发环境下的线程安全：如果一个线程给对象 A 分配了内存空间，但指针还没来得及修改，此时就可能出现另外一个线程使用原来的指针来给对象 B 分配内存空间的情况。

想要解决这个问题有两个方案：

①、**方式一**：采用同步锁定，或采用 CAS 配上失败重试的方式来保证更新操作的原子性。

②、**方式二**：为每个线程在 Java 堆中预先分配一块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer，TLAB）。

线程在进行内存分配时优先使用本地缓冲，当本地缓冲使用完成后，再向 Java 堆申请分配，此时 Java 堆采用同步锁定的方式来保证分配行为的线程安全。

3. 对象头设置

将对象有关的元数据信息、对象的哈希码、分代年龄等信息存储到对象头中。

可以和 JIT 那节的内容关联起来。

4. 对象初始化

调用对象的构造方法，即 Class 文件中的 `<init>()` 来初始化对象，为相关字段赋值。

3.2 对象的内存布局

在 HotSpot 中，对象在堆内存中的存储布局可以划分为以下三个部分：

1. 对象头 (Header)

对象头包括两部分信息：

- **Mark Word**：对象自身的运行时数据，如哈希码、GC 分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等，官方统称为 Mark Word，我们曾在 synchronized 的四种锁状态讲过。
- **类型指针**：对象指向它类型元数据的指针，Java 虚拟机通过这个指针来确定该对象是哪个类的实例。需要说明的是，并非所有的虚拟机都必须要在对象数据上保留类型指针，这取决于对象的访问定位方式。

2. 实例数据 (Instance Data)

即我们在代码中定义的各种类型的字段，无论是从父类继承而来，还是子类中定义的都需要记录。

3. 对齐填充 (Padding)

主要起占位符的作用。HotSpot 要求对象起始地址必须是 8 字节的整倍数，即间接要求了任何对象的大小都必须是 8 字节的整倍数。对象头部分在设计上就是 8 字节的整倍数，如果对象的实例数据不是 8 字节的整倍数，则由对齐填充进行补全。

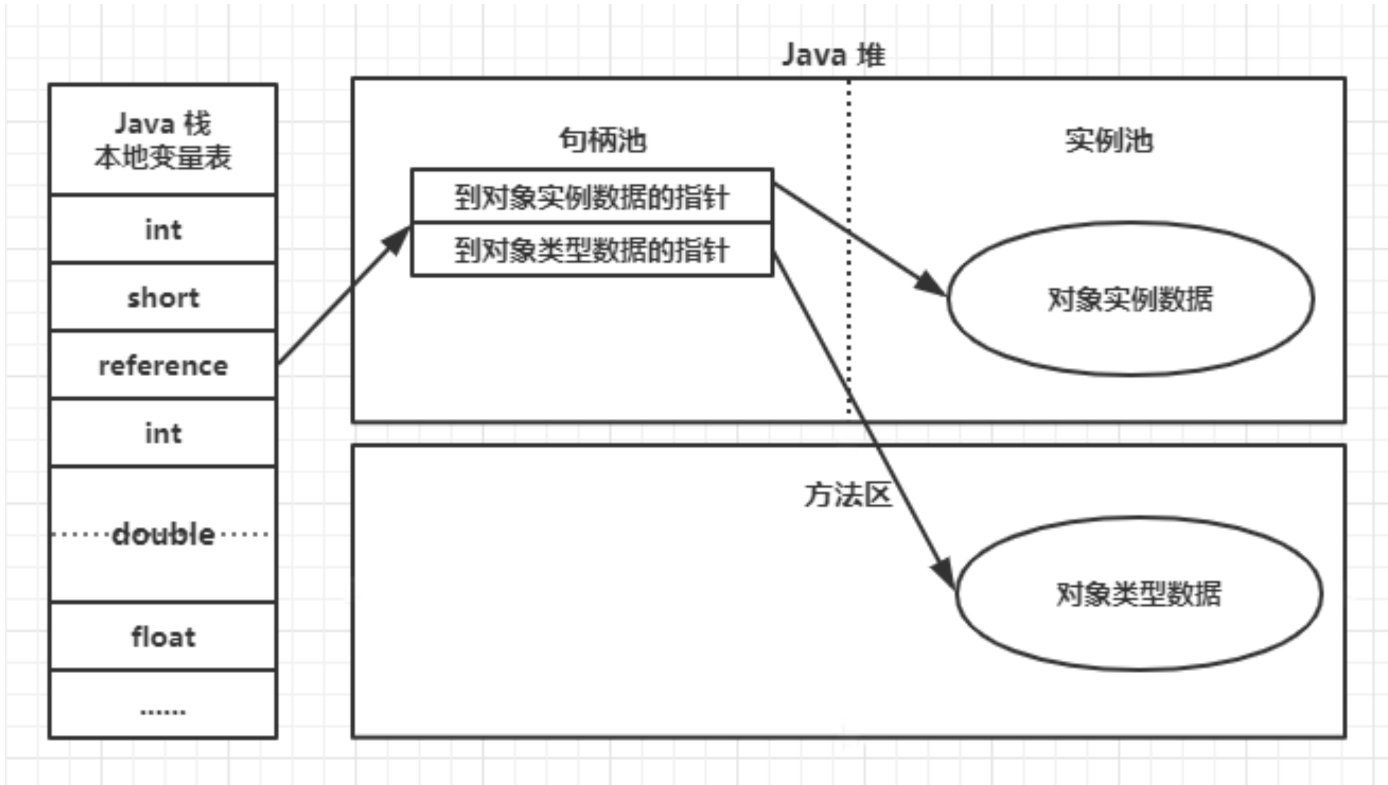
3.3 对象的访问定位

对象创建后，Java 程序就可以通过栈上的 `reference`（也就是引用）来操作堆上的具体对象。

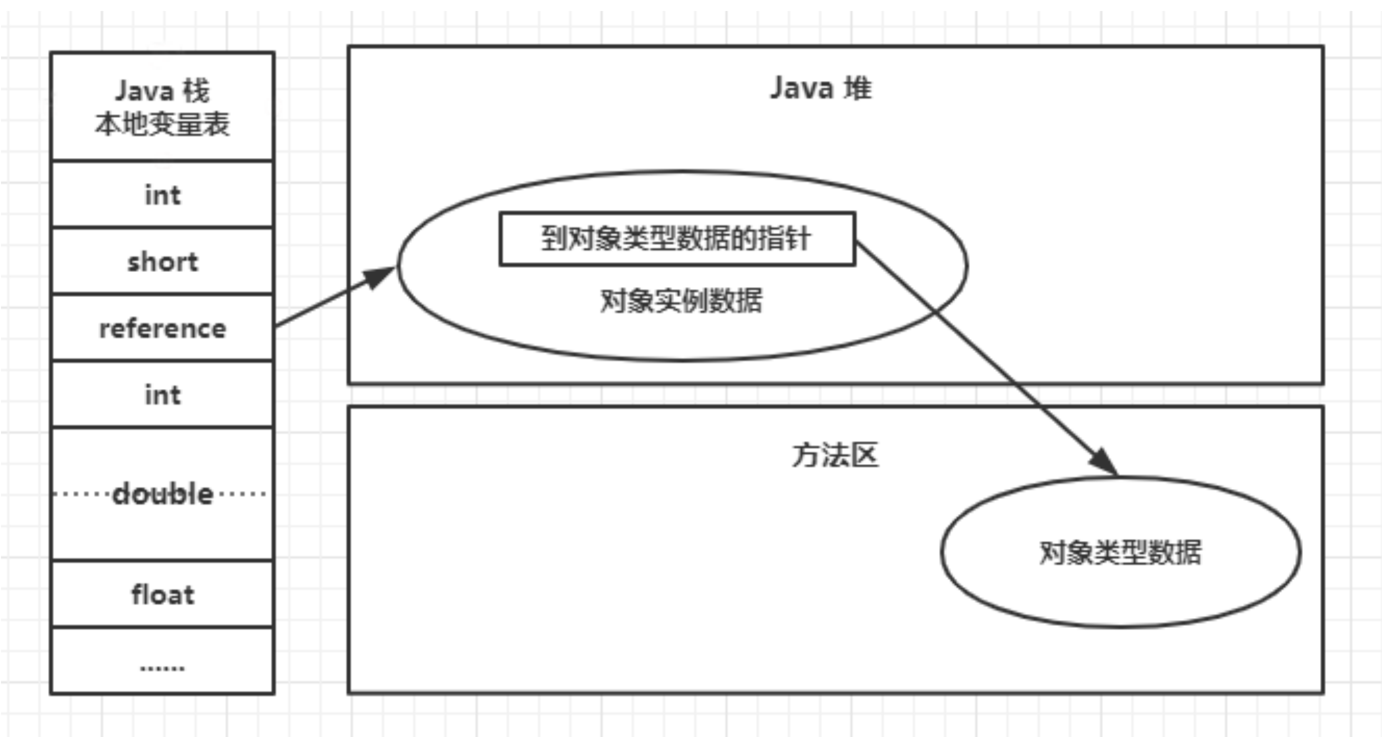
《Java 虚拟机规范》规定 `reference` 是一个指向对象的引用，但并未规定其具体实现方式。主流的方式方式有以下两种：

- **句柄访问**：Java 堆将划分出一块内存来作为句柄池， `reference` 中存储的是对象的句柄地址，而句柄则包含了对象实例数据和类型数据的地址信息。
- **指针访问**： `reference` 中存储的直接就是对象地址，而对象的类型数据则由上文介绍的对象头中的类型指针来指定。

通过句柄访问对象：



通过直接指针访问对象：



句柄访问的优点在于对象移动时（垃圾收集时移动对象是非常普遍的行为）只需要改变句柄中实例数据的指针，而 `reference` 本身并不需要修改；

指针访问则反之，由于其 `reference` 中存储的直接就是对象地址，所以当对象移动时， `reference` 需要被修改。但针对只需要访问对象本身的场景，指针访问则可以减少一次定位开销。由于对象访问是一项非常频繁的操作，所以这类减少的效果会非常显著，基于这个原因，**HotSpot 主要使用的是指针访问的方式**。

四、垃圾收集机制

在 JVM 内存模型中，程序计数器、虚拟机栈、本地方法栈这 3 个区域都是线程私有的，会随着线程的结束而销毁，因此在这 3 个区域当中，无需过多考虑垃圾回收问题。垃圾回收问题主要发生在 Java 堆上。

在 Java 堆上，垃圾回收的主要内容是死亡的对象（不可能再被任何途径使用的对象）。

判断对象是否死亡有以下两种方法：

4.1 引用计数法

在对象中添加一个引用计数器，对象每次被引用时，该计数器加一；当引用失效时，计数器的值减一；只要计数器的值为零，则代表对应的对象不可能再被使用。该方法的缺点在于无法避免相互引用的问题：

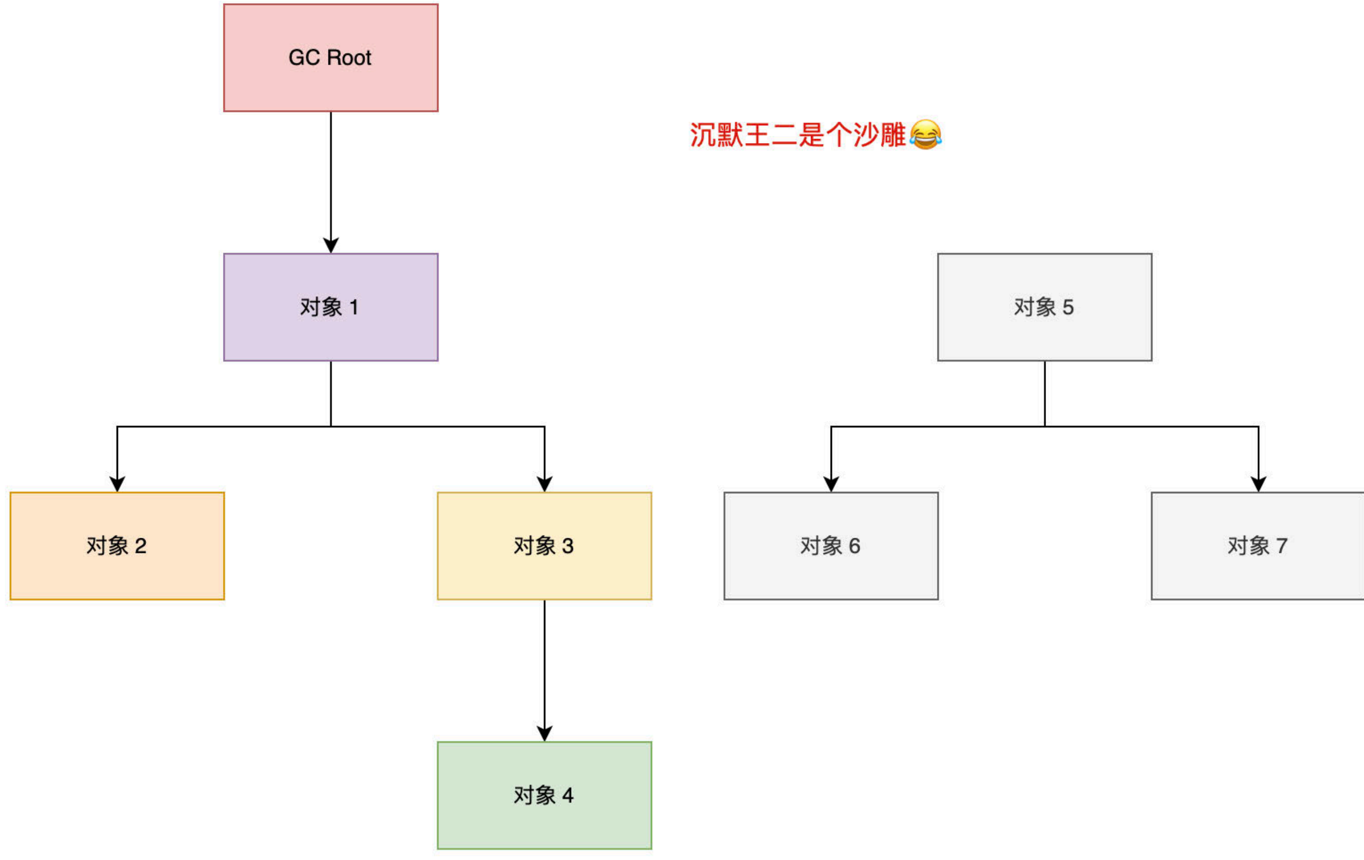
```
objA.instance = objB
objB.instance = objA
objA = null;
objB = null;
System.gc();
```

如上所示，此时两个对象已经不能再被访问，但其互相持有对对方的引用，如果采用引用计数法，则两个对象都无法被回收。

4.2 可达性分析

但上面的代码在大多数虚拟机中都能被正确的回收，因为大多数主流的虚拟机都是采用的可达性分析方法来判断对象是否死亡。

可达性分析是通过一系列被称为 `GC Roots` 的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程所走过的路径被称为引用链（Reference Chain），如果某个对象到 `GC Roots` 间没有任何引用链相连，这代表 `GC Roots` 到该对象不可达， 此时证明该对象不可能再被使用。



在 Java 语言中，固定可作为 `GC Roots` 的对象包括以下几种：

- 在虚拟机栈（栈帧中的本地变量表）中引用的对象，譬如各个线程被调用的方法堆栈中使用到的**参数**、**局部变量**、**临时变量**等；
- 在方法区（元空间）中类静态变量引用的对象，譬如 Java 类中引用类型的静态变量；
- 在方法区（元空间）中常量引用的对象，譬如字符串常量池（String Table）里的引用；
- 在本地方法栈中的 JNI（Native 方法）引用的对象；
- Java 虚拟机内部的引用，如基本数据类型对应的 Class 对象，一些常驻的异常对象（如 NullPointException，OutOfMemoryError 等）及系统类加载器；
- 所有被同步锁（synchronized 关键字）持有的对象；

除了这些固定的 `GC Roots` 集合以外，根据用户所选用的垃圾收集器以及当前回收的内存区域的不同，还可能会有其他对象 “临时性” 地加入，共同构成完整的 `GC Roots` 集合。

4.3 对象引用

可达性分析是基于引用链进行判断的，在 JDK 1.2 之后，Java 将引用关系分为以下四类：

强引用 (Strongly Reference)

最传统的引用，如 `Object obj = new Object()` 。无论任何情况下，只要强引用关系还存在，垃圾收集器就永远不会回收掉被引用的对象。

软引用 (Soft Reference)

用于描述一些还有用，但非必须的对象。只被软引用关联着的对象，在系统将要发生内存溢出之前，会被列入回收范围内进行第二次回收，如果这次回收后还没有足够的内存，才会抛出内存溢出异常。

下面是一个使用 Java 中 SoftReference 类的示例代码：

```
class SoftReferenceExample {
    public static void main(String[] args) {
        // 创建一个强引用的对象
        String strongReference = new String("二哥，我是个强引用");

        // 创建一个软引用，指向上面的对象
        SoftReference<String> softReference = new SoftReference<>(strongReference);

        // 干掉强引用
        strongReference = null;

        // 现在只有软引用指向 "二哥，我是个强引用" 对象

        // 尝试通过软引用获取对象
        String retrievedString = softReference.get();
        System.out.println(retrievedString); // 输出 "二哥，我是个强引用"

        // 强制进行垃圾回收，可能会清除软引用的对象
        System.gc();

        // 再次尝试通过软引用获取对象
        retrievedString = softReference.get();
        if (retrievedString != null) {
            System.out.println(retrievedString);
        } else {
            System.out.println("软引用的对象已被垃圾回收");
        }
    }
}
```

这个例子中，我们首先创建了一个字符串对象的强引用，然后通过 SoftReference 创建了这个对象的软引用。在取消了强引用后，这个对象只剩下软引用。当我们尝试通过软引用获取对象时，如果对象还存在，软引用会返回它；如果对象已被垃圾收集器回收，则返回 null。

需要注意的是，第二次回收时，如果这次回收后还没有足够的内存，才会抛出内存溢出异常。这里的 “足够” 是指在抛出内存溢出异常之前，系统会进行最后一次尝试，如果这次回收后还是没有足够的内存，才会抛出内存溢出异常。

通常情况下，上面代码在执行 gc 后软引用不会被回收，因为此时内存还是足够的。

```
二哥，我是个强引用
二哥，我是个强引用
```

弱引用 (Weak Reference)

用于描述那些非必须的对象，强度比软引用弱。被弱引用关联的对象只能生存到下一次垃圾收集发生时，无论当前内存是否足够，弱引用对象都会被回收。

来看这段代码：

```
class WeakReferenceExample {
    public static void main(String[] args) {
        // 创建一个强引用的对象
        String strongReference = new String("二哥，我是强引用");

        // 创建一个弱引用，指向上面的对象
        WeakReference<String> weakReference = new WeakReference<>(strongReference);

        // 取消强引用
        strongReference = null;

        // 强制进行垃圾回收
        System.gc();

        // 尝试通过弱引用获取对象
        String retrievedString = weakReference.get();
        if (retrievedString != null) {
            System.out.println(retrievedString);
        } else {
            System.out.println("弱引用的对象已被垃圾回收");
        }
    }
}
```

这个例子中，我们首先创建了一个字符串对象的强引用，然后通过 WeakReference 创建了这个对象的弱引用。在取消了强引用后，这个对象只剩下弱引用。当我们尝试通过弱引用获取对象时，如果对象还存在，弱引用会返回它；如果对象已被垃圾收集器回收，则返回 null。

运行结果就和软引用不一样了，gc 后弱引用被回收了。

```
弱引用的对象已被垃圾回收
```

虚引用 (Phantom Reference)

最弱的引用关系。为一个对象设置虚引用关联的唯一目的只是为了能在这个对象被回收时收到一个系统通知。

虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾收集器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象后，将这个虚引用加入到与之关联的引用队列中。

来看这个示例代码：

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

```
        // 创建一个强引用的对象
        String strongReference = new String("二哥，我是强引用");

        // 创建一个引用队列
        ReferenceQueue<String> referenceQueue = new ReferenceQueue<>();

        // 创建一个虚引用，指向上面的对象，并与引用队列关联
        PhantomReference<String> phantomReference = new PhantomReference<>
(strongReference, referenceQueue);

        // 取消强引用
        strongReference = null;

        // 强制进行垃圾回收
        System.gc();

        // 检查引用队列，看是否有通知
        if (referenceQueue.poll() != null) {
            System.out.println("虚引用的对象已被垃圾回收，且收到了通知");
        } else {
            System.out.println("虚引用的对象未被垃圾回收，或未收到通知");
        }
    }
}
```

这个例子中，我们首先创建了一个字符串对象的强引用，然后通过 PhantomReference 创建了这个对象的虚引用，并与引用队列关联。在取消了强引用后，这个对象只剩下虚引用。当我们强制进行垃圾回收时，如果对象还存在，虚引用不会返回它；如果对象已被垃圾收集器回收，则返回 null。同时，如果对象被回收，虚引用会被加入到引用队列中。

运行结果如下：

```
虚引用的对象已被垃圾回收，且收到了通知
```

4.4 对象真正死亡

要真正宣告一个对象死亡，需要经过至少两次标记过程：

①、如果对象在进行可达性分析后发现 `GC Roots` 不可达，将会进行第一次标记；

②、随后进行一次筛选，筛选的条件是此对象是否有必要执行 `finalized()` 方法。

如果对象没有覆盖 `finalized()` 方法，或者 `finalized()` 已经被虚拟机调用过，这两种情况都会视为没有必要执行。

如果判定结果是有必要执行，此时对象会被放入名为 `F-Queue` 的队列，等待 Finalizer 线程执行其 `finalized()` 方法。

在这个过程中，收集器会进行第二次小规模的标记，如果对象在 `finalized()` 方法中重新将自己与引用链上的任何一个对象进行了关联，如将自己（this 关键字）赋值给某个类变量或者对象的成员变量，此时它就实现了自我拯救，则第二次标记会将其移除 “即将回收” 的集合，否则该对象就将被真正回收，走向死亡。

4.5 垃圾收集算法

1. 分代收集理论

当前大多数虚拟机都遵循 “分代收集” 的理论进行设计，它建立在强弱两个分代假说下：

- **弱分代假说 (Weak Generational Hypothesis)**： 绝大多数对象都是朝生夕灭的。
- **强分代假说 (Strong Generational Hypothesis)**： 熬过越多次垃圾收集过程的对象就越难以消亡。
- **跨带引用假说 (Intergenerational Reference Hypothesis)**： 基于上面两条假说还可以得出的一条隐含推论：存在相互引用关系的两个对象，应该倾向于同时生存或者同时消亡。

强弱分代假说奠定了垃圾收集器的设计原则：**收集器应该将 Java 堆划分出不同的区域**，然后将回收对象依据其年龄（年龄就是对象经历垃圾收集的次数）分配到不同的区域中进行存储。

之后如果一个区域中的对象都是朝生夕灭的，那么收集器只需要关注少量对象的存活而不是去标记那些大量将要被回收的对象，此时就能以较小的代价获取较大的空间。

最后再将难以消亡的对象集中到一块，根据强分代假说，它们是很难消亡的，因此虚拟机可以使用较低的频率进行回收，这就兼顾了时间和内存空间的开销。

2. 回收类型

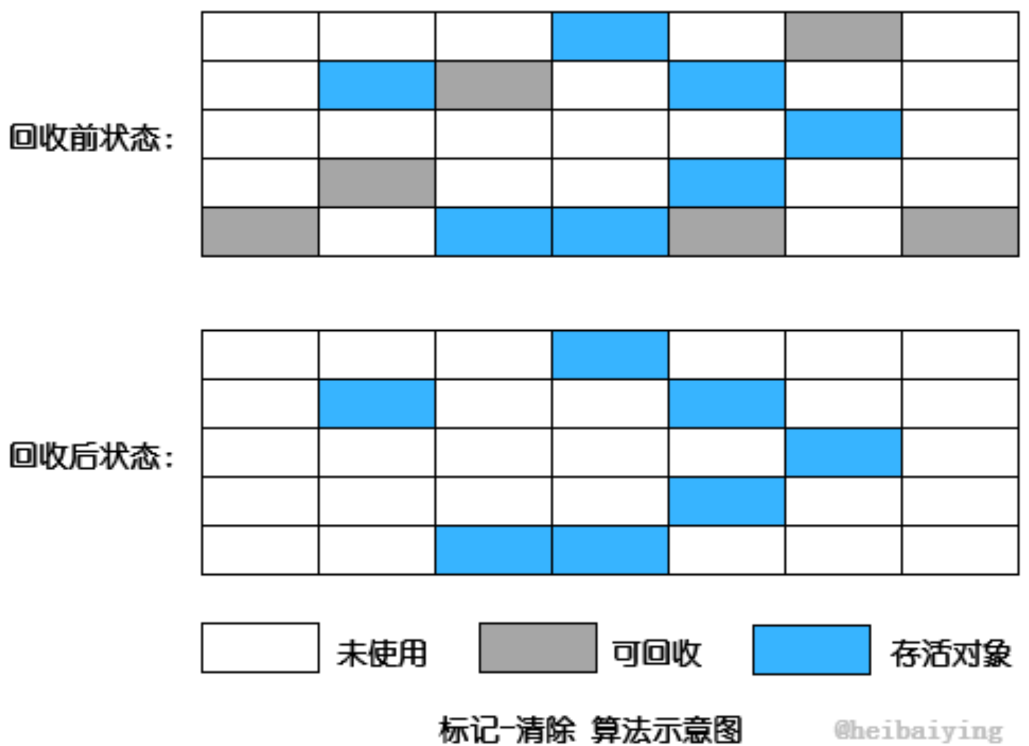
根据分代收集理论，收集范围可以分为以下几种类型：

①、**部分收集 (Partial GC)**： 具体分为：

- 新生代收集（Minor GC / Young GC）： 只对新生代进行垃圾收集；
- 老年代收集（Major GC / Old GC）： 只对老年代进行垃圾收集。需要注意的是 Major GC 在有的语境中也用于指代整堆收集；
- 混合收集（Mixed GC）： 对整个新生代和部分老年代进行垃圾收集。
- **整堆收集 (Full GC)**： 收集整个 Java 堆和方法区。

3. 标记-清除算法

它是最基础的垃圾收集算法，收集过程分为两个阶段：首先标记出所有需要回收的对象，在标记完成后，统一回收掉所有被标记的对象；也可以反过来，标记存活对象，统一回收所有未被标记的对象。



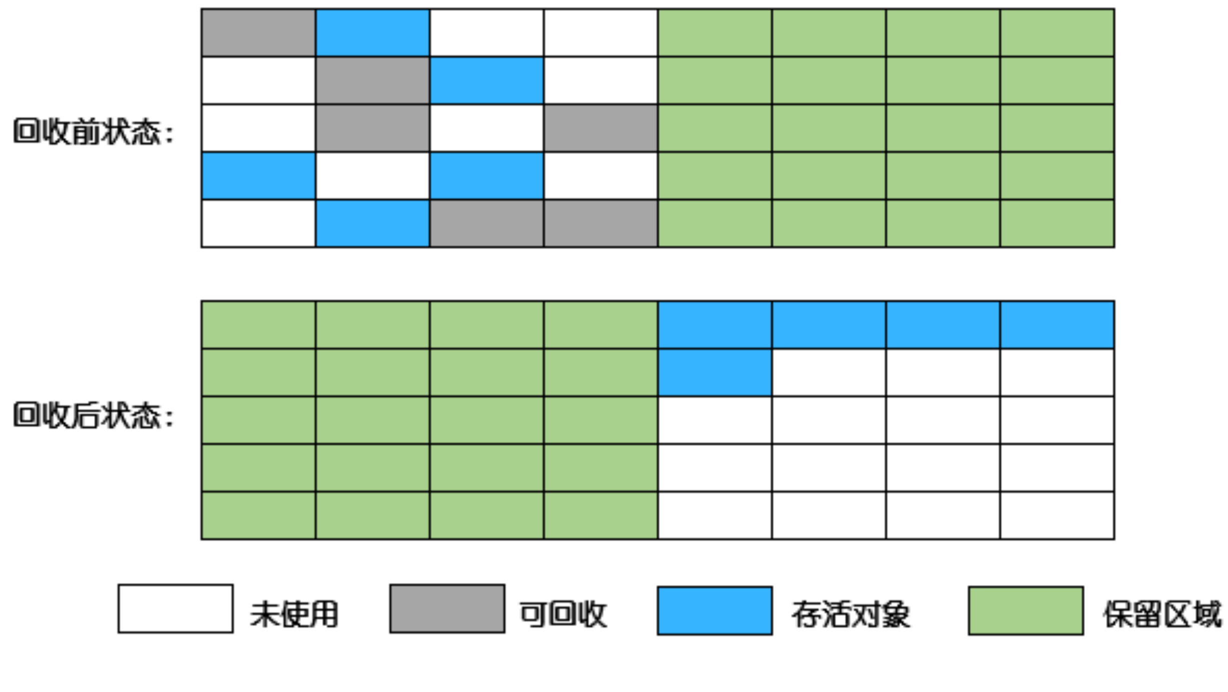
它主要有以下两个缺点：

- 执行效率不稳定：如果 Java 堆上包含大量需要回收的对象，则需要进行大量标记和清除动作；
- 内存空间碎片化：标记清除后会产生大量不连续的空间，从而导致无法为大对象分配足够的连续内存。

4. 标记-复制算法

标记-复制算法基于 "半区复制" 算法：它将可用内存按容量划分为大小相等的两块，每次只使用其中一块，当这一块的内存使用完了，就将还存活着的对象复制到另外一块，然后再把已经使用过的那块内存空间一次性清理掉。其优点在于避免了内存空间碎片化的问题，其缺点如下：

- 如果内存中多数对象都是存活的，这种算法将产生大量的复制开销；
- 浪费内存空间，内存空间变为了原有的一半。



基于新生代 “朝生夕灭” 的特点，大多数虚拟机都不会按照 1:1 的比例来进行内存划分，例如 HotSpot 会将内存空间划分为一块较大的 `Eden` 和两块较小的 `Survivor` 空间，它们之间的比例是 8:1:1 。

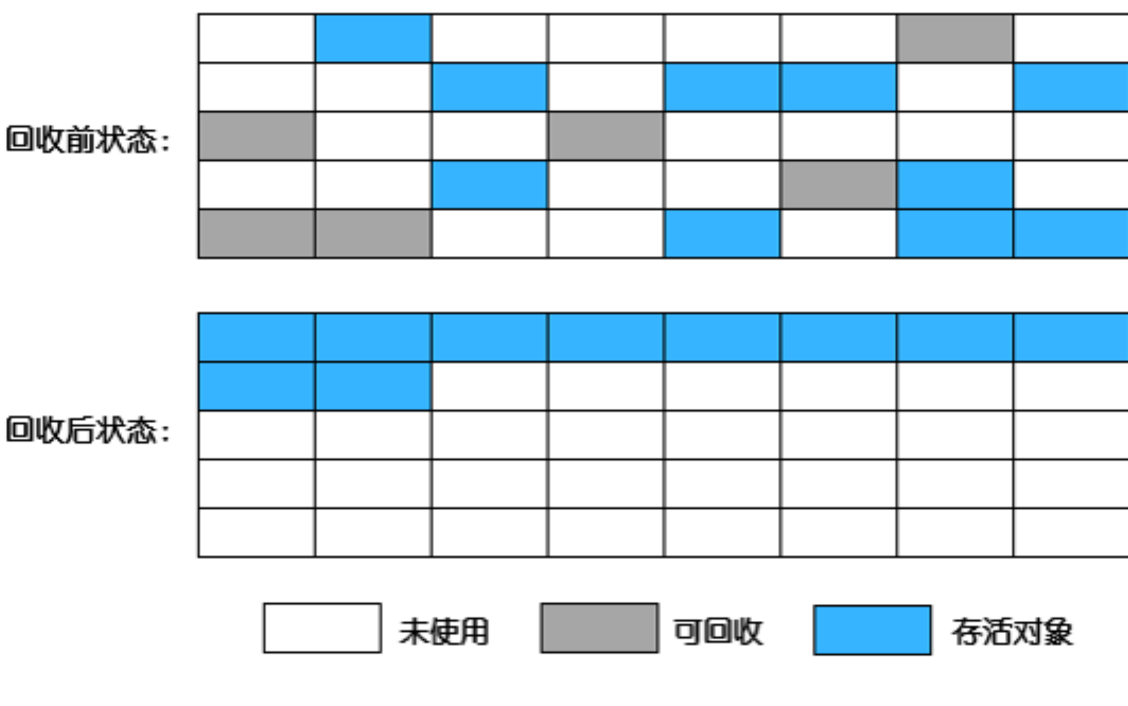
每次分配时只会使用 `Eden` 和其中的一块 `Survivor` ，发生垃圾回收时，只需要将存活的对象一次性复制到另外一块 `Survivor` 上，这样只有 10% 的内存空间会被浪费掉。

当 `Survivor` 空间不足以容纳一次 `Minor GC` 时，此时由其他内存区域（通常是老年代）来进行分配担保。

5. 标记-整理算法

标记-整理算法是在标记完成后，让所有存活对象都向内存的一端移动，然后直接清理掉边界以外的内存。

其优点在于可以避免内存空间碎片化的问题，也可以充分利用内存空间；其缺点在于根据所使用的收集器的不同，在移动存活对象时可能要全程暂停用户程序：



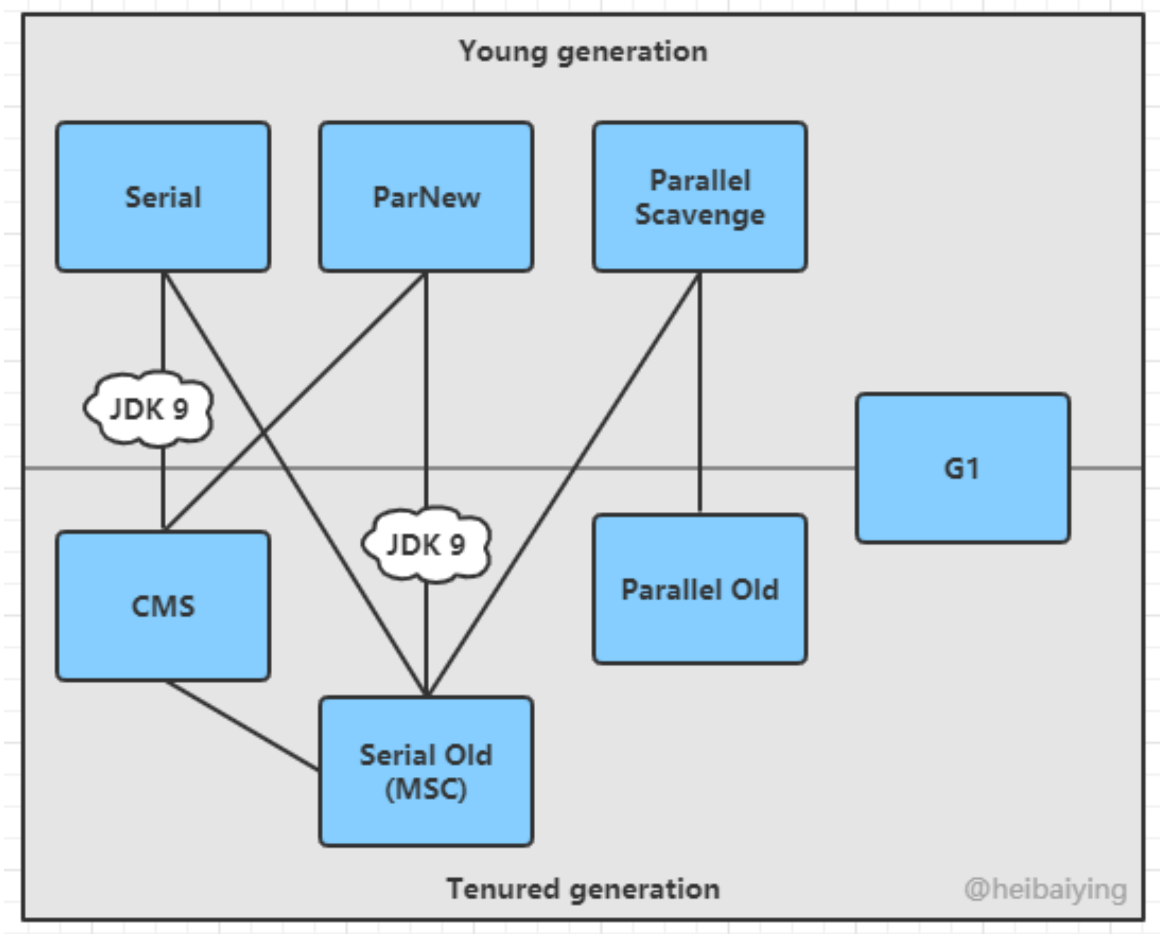
五、垃圾收集器

并行与并发是并发编程中的专有名词，在谈论垃圾收集器的上下文语境中，它们的含义如下：

①、**并行 (Parallel)**： 并行描述的是多条垃圾收集器线程之间的关系，说明同一时间有多条这样的线程在协同工作，此时通常默认用户线程是处于等待状态。

②、**并发 (Concurrent)**： 并发描述的是垃圾收集器线程与用户线程之间的关系，说明同一时间垃圾收集器线程与用户线程都在运行。但由于垃圾收集器线程会占用一部分系统资源，所以程序的吞吐量依然会受到一定影响。

HotSpot 中一共存在七款经典的垃圾收集器：

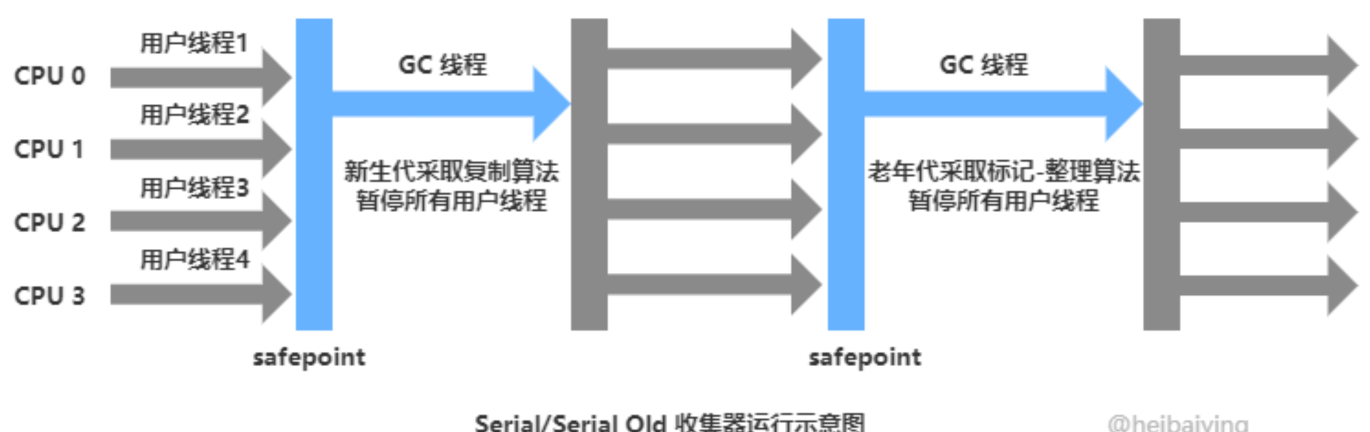


> 注：收集器之间存在连线，代表它们可以搭配使用。

5.1 Serial 收集器

Serial 收集器是最基础、历史最悠久的收集器，它是一个单线程收集器，在进行垃圾回收时，必须暂停其他所有的工作线程，直到收集结束，这是其主要缺点。

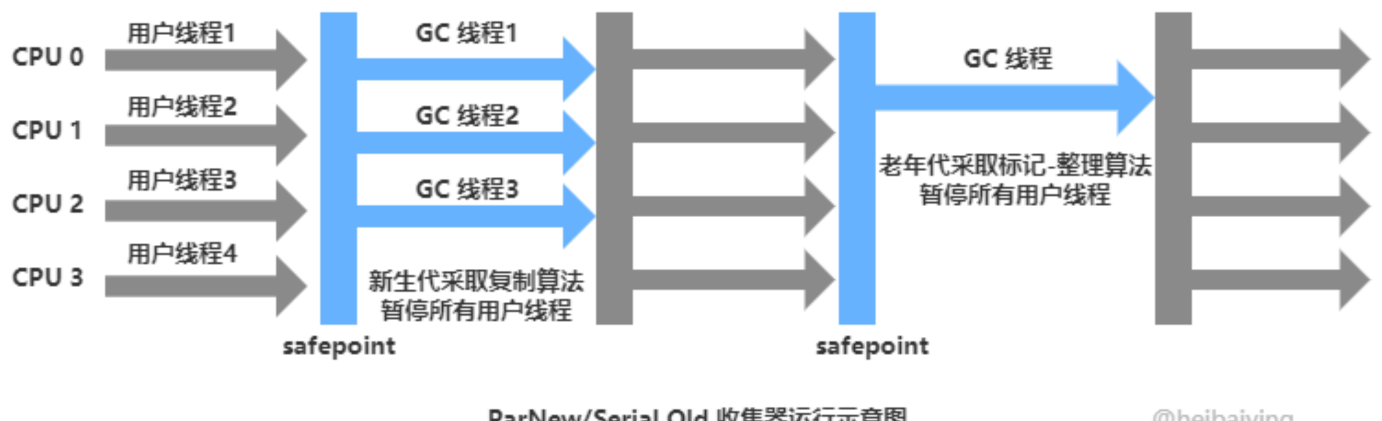
它的优点在于单线程避免了多线程复杂的上下文切换，因此在单线程环境下收集效率非常高，由于这个优点，迄今为止，其仍然是 HotSpot 虚拟机在客户端模式下默认的新生代收集器：



Serial/Serial Old 收集器运行示意图

5.2 ParNew 收集器

它是 Serial 收集器的多线程版本，可以使用多条线程进行垃圾回收：



ParNew/Serial Old 收集器运行示意图

5.3 Parallel Scavenge 收集器

Parallel Scavenge 也是新生代收集器，基于 标记-复制 算法进行实现，它的目标是达到一个可控的吞吐量。这里的吞吐量指的是处理器运行用户代码的时间与处理器总消耗时间的比值：

```
吞吐量 = 运行用户代码时间 \ (运行用户代码时间 + 运行垃圾收集时间)
```

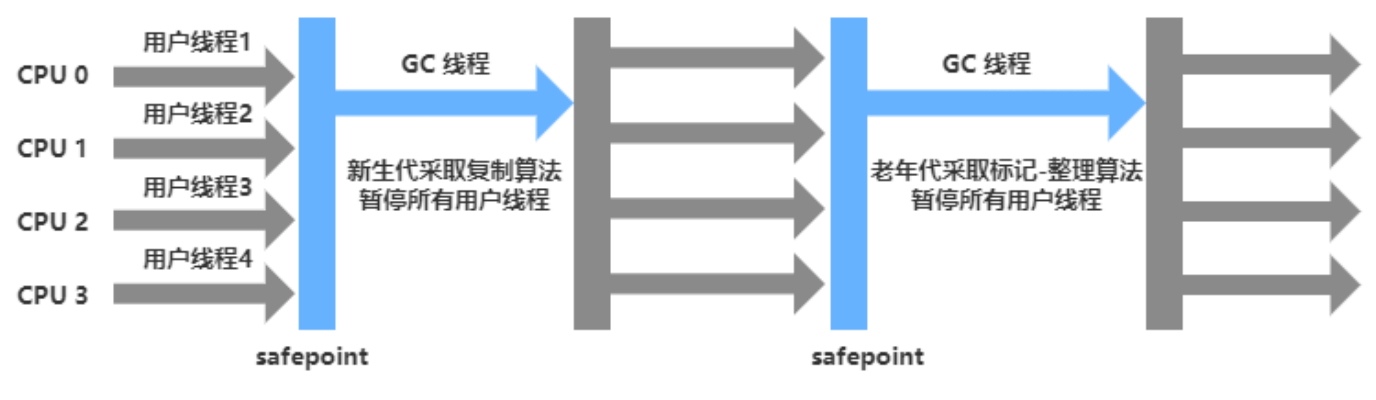
Parallel Scavenge 收集器提供两个参数用于精确控制吞吐量：

①、`-XX:MaxGCPauseMillis`：控制最大垃圾收集时间，假设需要回收的垃圾总量不变，那么降低垃圾收集的时间就会导致收集频率变高，所以需要将其设置为合适的值，不能一味减小。

②、`-XX:MaxGCTimeRatio`：直接用于设置吞吐量大小，它是一个大于 0 小于 100 的整数。假设把它设置为 19，表示此时允许的最大垃圾收集时间占总时间的 5%（即 1/(1+19)）；默认值为 99，即允许最大 1%（1/(1+99)）的垃圾收集时间。

5.4 Serial Old 收集器

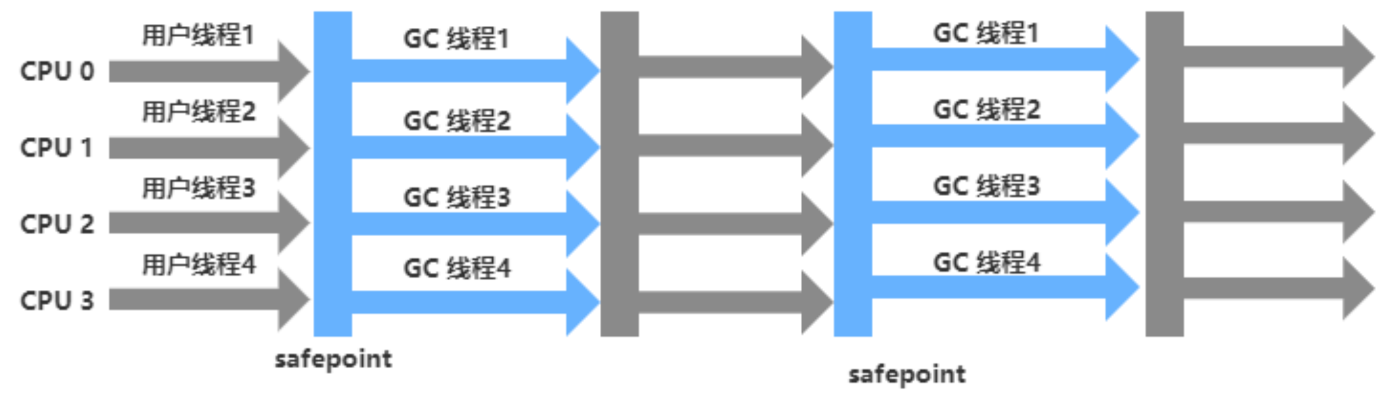
从名字也能看出来，它是 Serial 收集器的老年代版本，同样是一个单线程收集器，采用 标记-整理 算法，主要用于给客户端模式下的 HotSpot 使用：



Serial/Serial Old 收集器运行示意图

5.5 Paralled Old 收集器

Paralled Old 是 Parallel Scavenge 收集器的老年代版本，支持多线程并发收集，采用 标记-整理 算法实现：

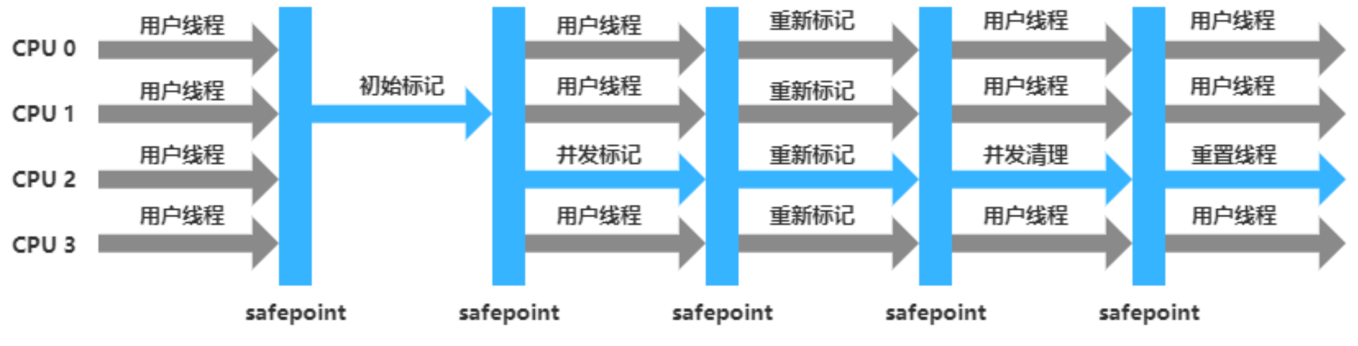


Parallel Scavenge/Parallel Old 收集器运行示意图

5.6 CMS 收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器，基于 标记-清除 算法实现，整个收集过程分为以下四个阶段：

1. **初始标记 (inital mark)**： 标记 `GC Roots` 能直接关联到的对象，耗时短但需要暂停用户线程；
2. **并发标记 (concurrent mark)**： 从 `GC Roots` 能直接关联到的对象开始遍历整个对象图，耗时长但不需要暂停用户线程；
3. **重新标记 (remark)**： 采用增量更新算法，对并发标记阶段因为用户线程运行而产生变动的那部分对象进行重新标记，耗时比初始标记稍长且需要暂停用户线程；
4. **并发清除 (inital sweep)**： 并发清除掉已经死亡的对象，耗时长但不需要暂停用户线程。



Concurrent Mark Sweep 收集器运行示意图

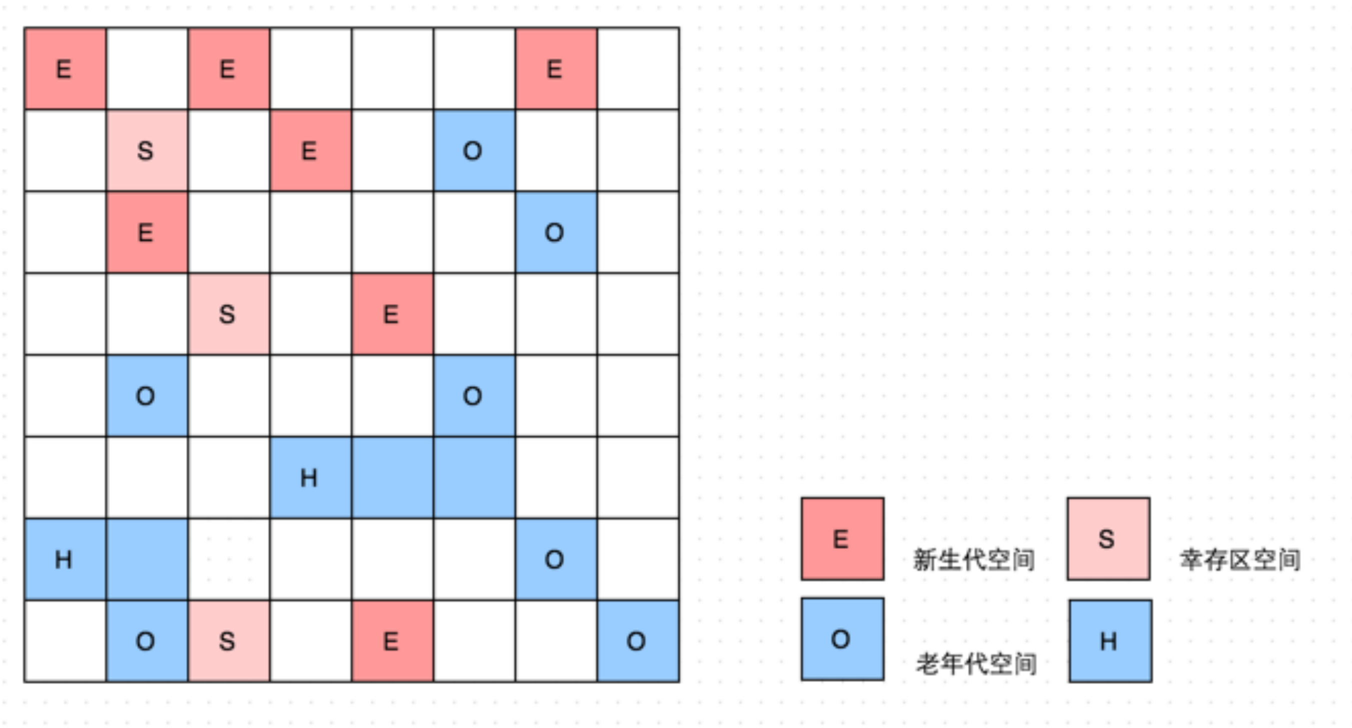
其优点在于耗时长的 并发标记 和 并发清除 阶段都不需要暂停用户线程，因此其停顿时间较短，其主要缺点如下：

- 由于涉及并发操作，因此对处理器资源比较敏感。
- 由于是基于 标记-清除 算法实现的，因此会产生大量空间碎片。
- 无法处理浮动垃圾（Floating Garbage）：由于并发清除时用户线程还是在继续，所以此时仍然会产生垃圾，这些垃圾就被称为浮动垃圾，只能等到下一次垃圾收集时再进行清理。

5.7 Garbage First 收集器

Garbage First（简称 G1）是一款面向服务端的垃圾收集器，也是 JDK 9 服务端模式下默认的垃圾收集器，它的诞生具有里程碑式的意义。

G1 虽然也遵循分代收集理论，但不再以固定大小和固定数量来划分分代区域，而是把连续的 Java 堆划分为多个大小相等的独立区域（Region）。每一个 Region 都可以根据不同的需求来扮演新生代的 `Eden` 空间、`Survivor` 空间或者老年代空间，收集器会根据其扮演角色的不同而采用不同的收集策略。



上面还有一些 Region 使用 H 进行标注，它代表 Humongous，表示这些 Region 用于存储大对象（humongous object，H-obj），即大小大于等于 region 一半的对象。

G1 收集器的运行大致可以分为以下四个步骤：

①、**初始标记 (Inital Marking)**：标记 `GC Roots` 能直接关联到的对象，并且修改 TAMS（Top at Mark Start）指针的值，让下一阶段用户线程并发运行时，能够正确的在 Reigin 中分配新对象。

G1 为每一个 Reigin 都设计了两个名为 TAMS 的指针，新分配的对象必须位于这两个指针位置以上，位于这两个指针位置以上的对象默认被隐式标记为存活的，不会纳入回收范围；

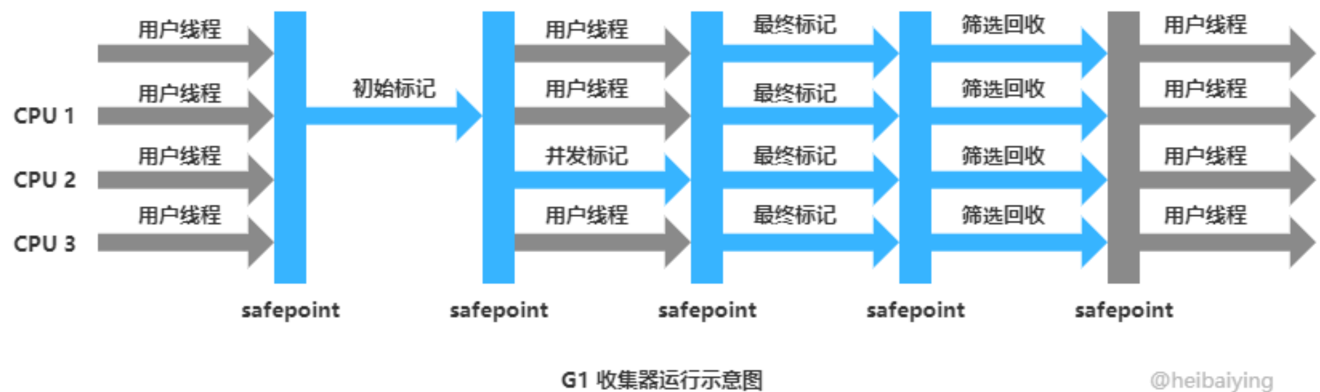
②、**并发标记 (Concurrent Marking)**：从 `GC Roots` 能直接关联到的对象开始遍历整个对象图。遍历完成后，还需要处理 SATB 记录中变动的对象。

SATB（snapshot-at-the-beginning，开始阶段快照）能够有效的解决并发标记阶段因为用户线程运行而导致的对象变动，其效率比 CMS 重新标记阶段所使用的增量更新算法效率更高；

③、**最终标记 (Final Marking)**：对用户线程做一个短暂的暂停，用于处理并发阶段结束后仍遗留下来的少量的 STAB 记录。虽然并发标记阶段会处理 SATB 记录，但由于处理时用户线程依然是运行中的，因此依然会有少量的变动，所以需要最终标记来处理；

④、**筛选回收 (Live Data Counting and Evacuation)**：负责更新 Regin 统计数据，按照各个 Regin 的回收价值和成本进行排序，在根据用户期望的停顿时间进行来指定回收计划，可以选择任意多个 Regin 构成回收集。

然后将回收集中 Regin 的存活对象复制到空的 Regin 中，再清理掉整个旧的 Regin 。此时因为涉及到存活对象的移动，所以需要暂停用户线程，并由多个收集线程并行执行。



G1 收集器运行示意图

5.8 内存分配原则

1. 对象优先在 Eden 分配

大多数情况下，对象在新生代的 `Eden` 区中进行分配，当 `Eden` 区没有足够空间时，虚拟机将进行一次 Minor GC。

2. 大对象直接进入老年代

大对象就是指需要大量连续内存空间的 Java 对象，最典型的就是超长的字符串或者元素数量很多的数组，它们将直接进入老年代。

主要是因为如果在新生代分配，因为其需要大量连续的内存空间，可能会导致提前触发垃圾回收；并且由于新生代的垃圾回收本身就很频繁，此时复制大对象也需要额外的性能开销。

3. 长期存活的对象将进入老年代

虚拟机会给每个对象在其对象头中定义一个年龄计数器。对象通常在 `Eden` 区中诞生，如果经历第一次 Minor GC 后仍然存活，并且能够被 Survivor 容纳的话，该对象就会被移动到 Survivor 中，并将其年龄加 1。

对象在 Survivor 中每经过一次 Minor GC，年龄就加 1，当年龄达到一定程度后（由 `-XX:MaxTenuringThreshold` 设置，默认值为 15）就会进入老年代中。

4. 动态年龄判断

如果在 Survivor 空间中相同年龄的所有对象大小的总和大于 Survivor 空间的一半，那么年龄大于或等于该年龄的对象就可以直接进入老年代，而无需等待年龄到达 `-XX:MaxTenuringThreshold` 设置的值。

5. 空间担保分配

在发生 Minor GC 之前，虚拟机必须先检查老年代最大可用的连续空间是否大于新生代所有对象的总空间，如果条件成立，那么这一次的 Minor GC 可以确认是安全的。

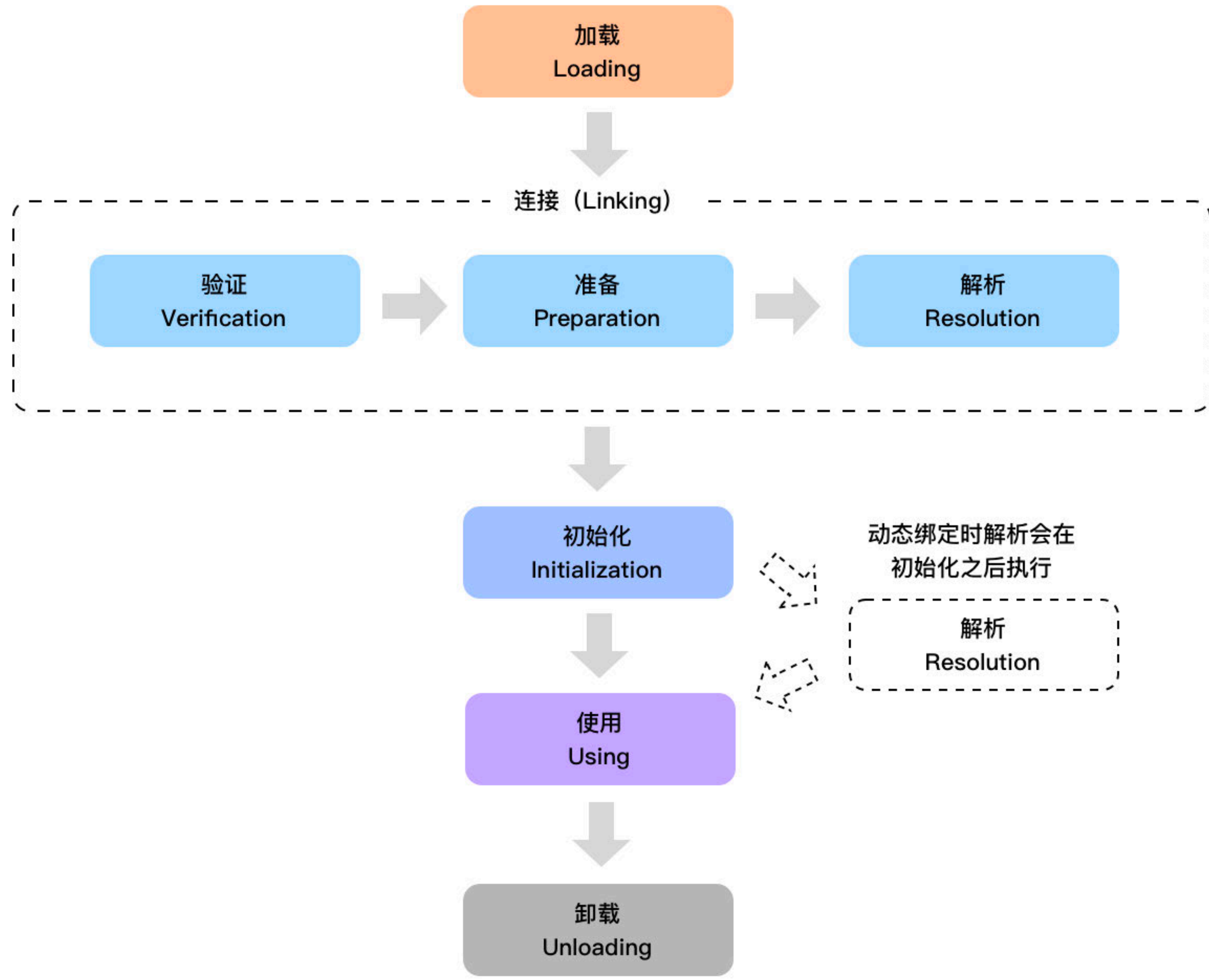
如果不成立，虚拟机会查看 `-XX:HandlePromotionFailure` 的值是否允许担保失败，如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于或者 `-XX:HandlePromotionFailure` 的值设置不允许冒险，那么就要改为进行一次 Full GC 。

六、类加载机制

Java 虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这个过程被称为虚拟机的类加载机制。

6.1 类加载时机

一个类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期将会经历加载、验证、准备、卸载、解析、初始化、使用、卸载七个阶段，其中验证、准备、解析三个部分统称为连接：



《Java 虚拟机规范》严格规定了有且只有六种情况必须立即对类进行初始化：

①、遇到 `new`、 `getstatic`、 `putstatic`、 `invokestatic` 这四条字节码指令，能够生成这四条指令码的典型 Java 代码场景有：

- 使用 `new` 关键字实例化对象时；
- 读取或设置一个类型的静态字段时（被 final 修饰，已在编译期把结果放入常量池的静态字段除外）；
- 调用一个类的静态方法时。

②、使用 `java.lang.reflect` 包的方法对 Class 进行反射调用时，如果类型没有进行过初始化、则需要触发其初始化；

③、当初始化类时，如发现其父类还没有进行过初始化、则需要触发其父类进行初始化；

④、当虚拟机启动时，用户需要指定一个要执行的主类（包含 main() 方法的那个类），虚拟机会先初始化这个主类；

⑤、当使用 JDK 7 新加入的动态语言支持时，如果一个 `java.lang.invoke.MethodHandle` 实例最后解析的结果为 `REF_getStatic`，`REF_putStatic`，`REF_invokeStatic`，`REF_newInvokeSpecial` 四种类型的方法句柄，并且这个方法句柄对应的类没有进行过初始化，则需要先触发其初始化；

⑥、当一个接口中定义了 JDK 8 新加入的默认方法（被 default 关键字修饰的接口方法）时，如果有这个接口的实现类发生了初始化，那么该接口要在其之前被初始化。

6.2 类加载过程

1. 加载

在加载阶段，虚拟机需要完成以下三件事：

- 通过一个类的全限定名来获取定义此类的二进制字节流；
- 将这个字节流所代表的静态存储结构转换为运行时数据结构；
- 在内存中生成一个代表这个类的 `java.lang.Class` 对象，作为这个类的各种数据的访问入口。

《Java 虚拟机规范》并没有限制从何处获取二进制流，因此可以从 JAR 包、WAR 包获取，也可以从 JSP 生成的 Class 文件等处获取。

2. 验证

这一阶段的目的是确保 Class 文件的字节流中包含的信息符合《Java 虚拟机规范》的全部约束要求，从而保证这些信息被当做代码运行后不会危害虚拟机自身的安全。

验证阶段大致会完成下面四项验证：

- **文件格式验证**：验证字节流是否符合 Class 文件格式的规范；
- **元数据验证**：对字节码描述的信息进行语义分析，以保证其描述的信息符合《Java 语言规范》的要求（如除了 `java.lang.Object` 外，所有的类都应该有父类）；
- **字节码验证**：通过数据流分析和控制流分析，确定程序语义是合法的，符合逻辑的（如允许把子类对象赋值给父类数据类型，但不能把父类对象赋值给子类数据类型）；
- **符号引用验证**：验证类是否缺少或者被禁止访问它依赖的某些外部类、方法、字段等资源。如果无法验证通过，则会抛出一个 `java.lang.IncompatibleClassChangeError` 的子类异常，如 `java.lang.NoSuchFieldError`、`java.lang.NoSuchMethodError` 等。

3. 准备

准备阶段是正式为类中定义的变量（即静态变量，被 static 修饰的变量）分配内存并设置类变量初始值的阶段。

4. 解析

解析是 Java 虚拟机将常量池内的符号引用替换为直接引用的过程：

- **符号引用**：符号引用用一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。
- **直接引用**：直接引用是指可以直接指向目标的指针、相对偏移量或者一个能间接定位到目标的句柄。

整个解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符这 7 类符号引用进行解析。

5. 初始化

初始化阶段就是执行类构造器的 `<clinit>()` 方法的过程，该方法具有以下特点：

- `<clinit>()` 方法由编译器自动收集类中所有类变量的赋值动作和静态语句块中的语句合并产生，编译器收集顺序由语句在源文件中出现的顺序决定。
- `<clinit>()` 方法与类的构造方法（即在虚拟机视角中的实例构造器 `<init>()`方法）不同，它不需要显示的调用父类的构造器，Java 虚拟机会保证在子类的 `<clinit>()` 方法执行前，父类的 `<clinit>()` 方法已经执行完毕。
- 由于父类的 `<clinit>()` 方法先执行，也就意味着父类中定义的静态语句块要优先于子类变量的赋值操作。
- `<clinit>()` 方法对于类或者接口不是必须的，如果一个类中没有静态语句块，也没有对变量进行赋值操作，那么编译器可以不为这个类生成 `<clinit>()` 方法。
- 接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成 `<clinit>()` 方法。
- Java 虚拟机必须保证一个类的 `<clinit>()` 方法在多线程环境中被正确的加锁同步，如果多个线程同时去初始化一个类，那么只会有其中一个线程去执行这个类的 `<clinit>()` 方法，其他线程都需要阻塞等待。

6.3 类加载器

能够通过一个类的全限定名来获取描述该类的二进制字节流的工具称为类加载器。

每一个类加载器都拥有一个独立的类名空间，因此对于任意一个类，都必须由加载它的类加载器和这个类本身来共同确立其在 Java 虚拟机中的唯一性。

这意味着要想比较两个类是否相等，必须在同一类加载器加载的前提下；如果两个类的类加载器不同，则它们一定不相等。

6.4 双亲委派模型

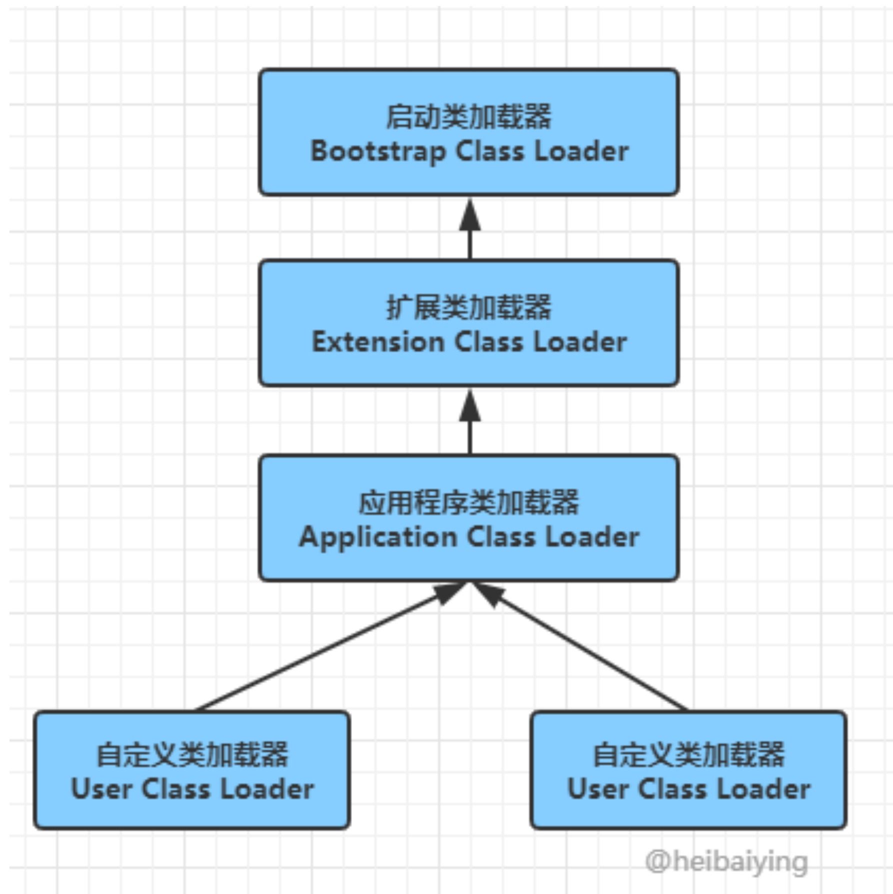
从 Java 虚拟机角度而言，类加载器可以分为以下两类：

- **启动类加载器**：启动类加载器（Bootstrap ClassLoader）由 C++ 语言实现（以 HotSpot 为例），它是虚拟机自身的一部分；
- **其他所有类的类加载器**：由 Java 语言实现，独立存在于虚拟机外部，并且全部继承自 `java.lang.ClassLoader` 。

从开发人员角度而言，类加载器可以分为以下三类：

- **启动类加载器 (Boostrap Class Loader)**：负责把存放在 `<JAVA_HOME>\lib` 目录中，或被 `-Xbootclasspath` 参数所指定的路径中存放的能被 Java 虚拟机识别的类库加载到虚拟机的内存中；
- **扩展类加载器 (Extension Class Loader)**：负责加载 `<JAVA_HOME>\lib\ext` 目录中，或被 `java.ext.dirs` 系统变量所指定的路径中的所有类库。
- **应用程序类加载器 (Application Class Loader)**：负责加载用户类路径（ClassPath）上的所有的类库。

JDK 9 之前的 Java 应用都是由这三种类加载器相互配合来完成加载：



上图所示的各种类加载器之间的层次关系被称为类加载器的 “双亲委派模型”，“双亲委派模型” 要求除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器，需要注意的是这里的加载器之间的父子关系一般不是以继承关系来实现的，而是使用组合关系来复用父类加载器的代码。

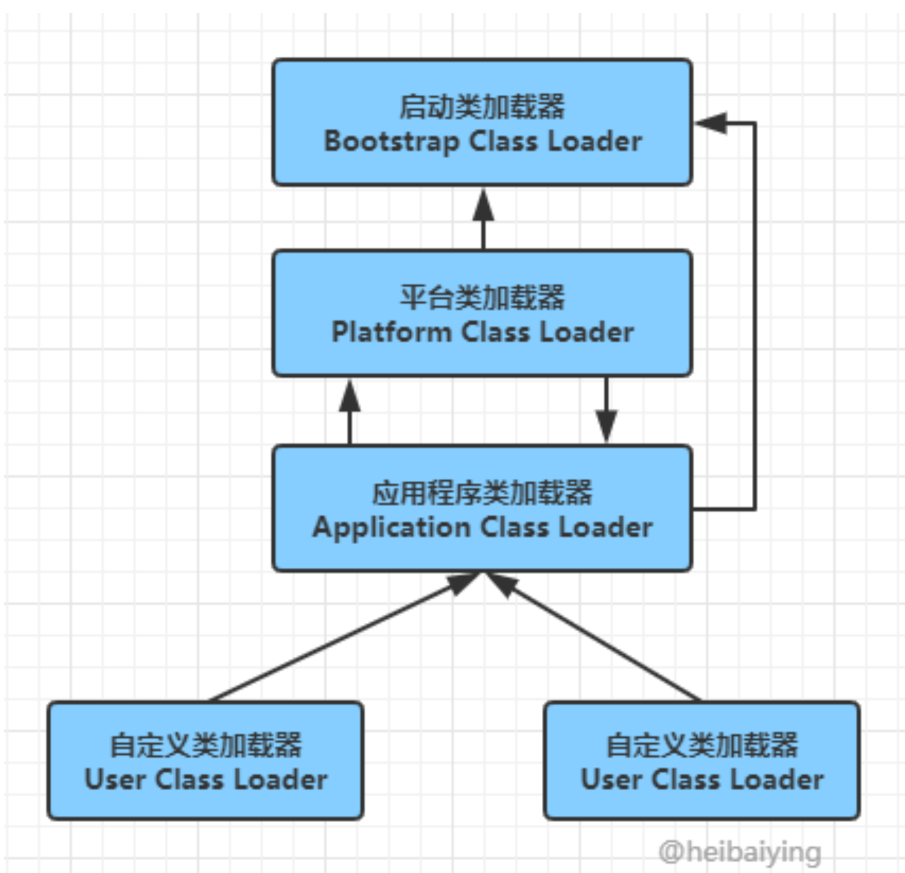
双亲委派模型的工作过程如下：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一层的类加载器都是如此，因此所有的加载请求最终都应该传送到最顶层的启动类加载器，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去完成加载。

基于双亲委派模型可以保证程序中的类在各种类加载器环境中都是同一个类，否则就有可能出现一个程序中存在两个不同的 `java.lang.Object` 的情况。

6.5 模块化下的类加载器

JDK 9 之后为了适应模块化的发展，类加载器做了如下变化：

- 仍维持三层类加载器和双亲委派的架构，但扩展类加载器被平台类加载器所取代；
- 当平台及应用程序类加载器收到类加载请求时，要首先判断该类是否能够归属到某一个系统模块中，如果可以找到这样的归属关系，就要优先委派给负责那个模块的加载器完成加载；
- 启动类加载器、平台类加载器、应用程序类加载器全部继承自 `java.internal.loader.BuiltinClassLoader` ，BuiltinClassLoader 中实现了新的模块化架构下类如何从模块中加载的逻辑，以及模块中资源可访问性的处理。



七、程序编译

7.1 编译器分类

- **前端编译器**：把 `*.java` 文件转变成 `.class` 文件的过程；如 JDK 的 Javac，Eclipse JDT 中的增量式编译器。
- **即时编译器**：常称为 JIT 编译器（Just In Time Complier），在运行期把字节码转变成本地机器码的过程；如 HotSpot 虚拟机中的 C1、C2 编译器，Graal 编译器。
- **提前编译器**：直接把程序编译成目标机器指令集相关的二进制代码的过程。如 JDK 的 jaotc，GUN Compiler for the Java（GCJ），Excelsior JET 。

7.2 解释器与编译器

在 HotSpot 中，Java 程序最初都是通过解释器（Interpreter）进行解释执行的，其优点在于可以省去编译时间，让程序快速启动。

当程序启动后，如果虚拟机发现某个方法或代码块的运行特别频繁，就会使用编译器将其编译为本地机器码，并使用各种手段进行优化，从而提高执行效率，这就是即时编译器。

HotSpot 内置了两个（或三个）即时编译器：

- **客户端编译器 (Client Complier)**： 简称 C1；
- **服务端编译器 (Servier Complier)**： 简称 C2，在有的资料和 JDK 源码中也称为 Opto 编译器；
- **Graal 编译器**： 在 JDK 10 时才出现，长期目标是替代 C2。

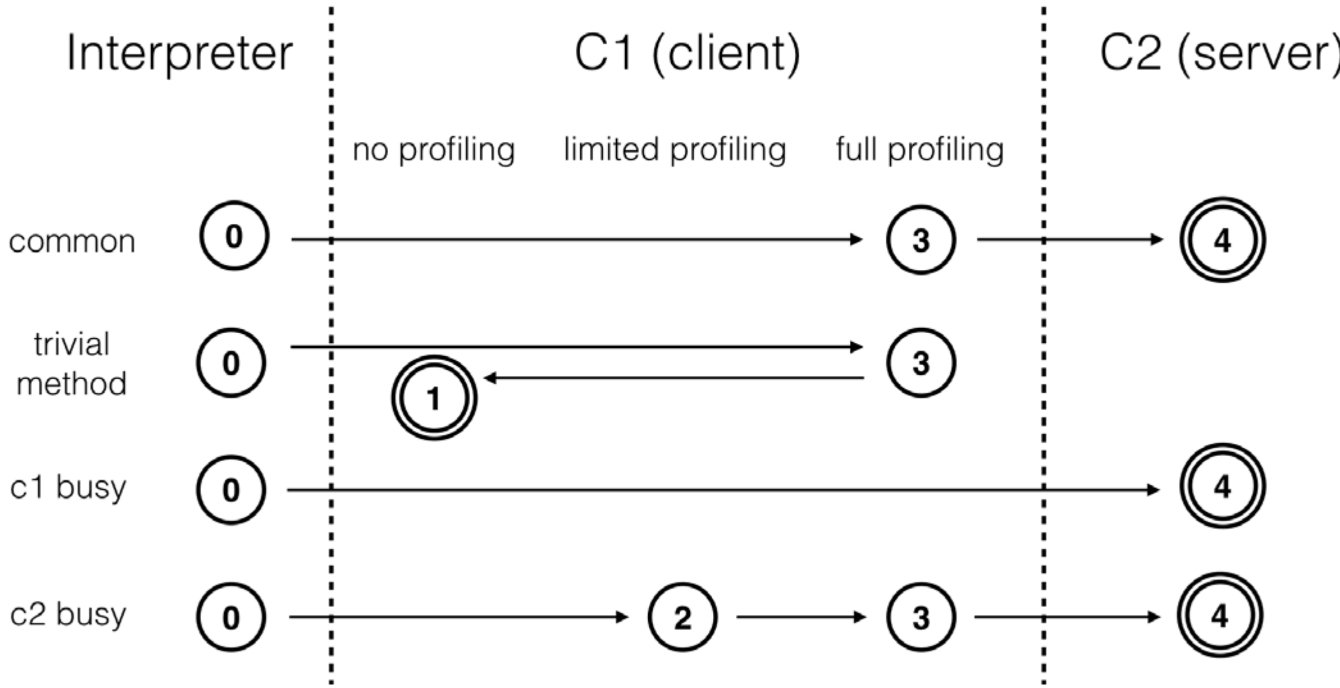
在分层编译的工作模式出现前，不管是采用客户端编译器还是服务端编译器完全取决于虚拟机是运行在客户端模式还是服务端模式下，可以在启动时通过 `-client` 或 `-server` 参数进行指定，也可以让虚拟机根据自身版本和宿主机性能来自主选择。

7.3 分层编译

要编译出优化程度越高的代码通常都需要越长的编译时间，为了在程序启动速度与运行效率之间达到最佳平衡，HotSpot 在编译子系统中加入了分层编译（Tiered Compilation）：

- **第 0 层**：程序纯解释执行，并且解释器不开启性能监控功能；
- **第 1 层**：使用客户端编译器将字节码编译为本地代码来运行，进行简单可靠的稳定优化，不开启性能监控功能；
- **第 2 层**：仍然使用客户端编译执行，仅开启方法及回边次数统计等有限的性能监控；
- **第 3 层**：仍然使用客户端编译执行，开启全部性能监控；
- **第 4 层**：使用服务端编译器将字节码编译为本地代码，其耗时更长，并且会根据性能监控信息进行一些不可靠的激进优化。

以上层次并不是固定不变的，根据不同的运行参数和版本，虚拟机可以调整分层的数量。各层次编译之间的交互转换关系如下图所示：



实施分层编译后，解释器、客户端编译器和服务端编译器就会同时工作，可以用客户端编译器获取更高的编译速度、用服务端编译器来获取更好的编译质量。

7.4 热点探测

即时编译器编译的目标是 “热点代码”，它主要分为以下两类：

- 被多次调用的方法。
- 被多次执行循环体。这里指的是一个方法只被少量调用过，但方法体内部存在循环次数较多的循环体，此时也认为是热点代码。但编译器编译的仍然是循环体所在的方法，而不会单独编译循环体。

判断某段代码是否是热点代码的行为称为 “热点探测” （Hot Spot Code Detection），主流的热点探测方法有以下两种：

- **基于采样的热点探测 (Sample Based Hot Spot Code Detection)**： 采用这种方法的虚拟机会周期性地检查各个线程的调用栈顶，如果发现某个（或某些）方法经常出现在栈顶，那么就认为它是 “热点方法”。
- **基于计数的热点探测 (Counter Based Hot Spot Code Detection)**： 采用这种方法的虚拟机会为每个方法

（甚至是代码块）建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值就认为它是 “热点方法”。

八、代码优化

即时编译器除了将字节码编译为本地机器码外，还会对代码进行一定程度的优化，它包含多达几十种优化技术，这里选取其中代表性的四种进行介绍：

8.1 方法内联

最重要的优化手段，它会将目标方法中的代码原封不动地 “复制” 到发起调用的方法之中，避免发生真实的方法调用，并采用名为类型继承关系分析（Class Hierarchy Analysis，CHA）的技术来解决虚方法（Java 语言中默认的实例方法都是虚方法）的内联问题。

8.2 逃逸分析

逃逸行为主要分为以下两类：

- **方法逃逸**：当一个对象在方法里面被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他方法中，此时称为方法逃逸；
- **线程逃逸**：当一个对象在方法里面被定义后，它可能被外部线程所访问，例如赋值给可以在其他线程中访问的实例变量，此时称为线程逃逸，其逃逸程度高于方法逃逸。

```
public static StringBuilder concat(String... strings) {
    StringBuilder sb = new StringBuilder();
    for (String string : strings) {
        sb.append(string);
    }
    return sb; // 发生了方法逃逸
}

public static String concat(String... strings) {
    StringBuilder sb = new StringBuilder();
    for (String string : strings) {
        sb.append(string);
    }
    return sb.toString(); // 没有发生方法逃逸
}
```

如果能证明一个对象不会逃逸到方法或线程之外，或者逃逸程度比较低（只逃逸出方法而不会逃逸出线程），则可以对这个对象实例采取不同程序的优化：

- **栈上分配 (Stack Allocations)**： 如果一个对象不会逃逸到线程外，那么将会在栈上分配内存来创建这个对象，而不是 Java 堆上，此时对象所占用的内存空间就会随着栈帧的出栈而销毁，从而可以减轻垃圾回收的压力。
- **标量替换 (Scalar Replacement)**： 如果一个数据已经无法再分解成为更小的数据类型，那么这些数据就称为标量（如 int、long 等数值类型及 reference 类型等）；反之，如果一个数据可以继续分解，那它就被称为聚合量（如对象）。如果一个对象不会逃逸外方法外，那么就可以将其改为直接创建若干个被这个方法使用的成员变量来替代，从而减少内存占用。
- **同步消除 (Synchronization Elimination)**： 如果一个变量不会逃逸出线程，那么对这个变量实施的同步措

施就可以消除掉。

8.3 公共子表达式消除

如果一个表达式 E 之前已经被计算过了，并且从先前的计算到现在 E 中所有变量的值都没有发生过变化，那么 E 这次的出现就称为公共子表达式。对于这种表达式，无需再重新进行计算，只需要直接使用前面的计算结果即可。

8.4 数组边界检查消除

对于虚拟机执行子系统来说，每次数组元素的读写都带有一次隐含的上下文检查以避免访问越界。如果数组的访问发生在循环之中，并且使用循环变量来访问数据，即循环变量的取值永远在 `[0, list.length)` 之间，那么此时就可以消除整个循环的数据边界检查，从而避免多次无用的判断。

小结

这篇内容我们系统地总结了 JVM 最重要的知识点，比如说 JVM 的内存结构、垃圾回收算法、垃圾回收器、类加载机制、类加载器、程序编译、代码优化等等，希望能对大家在学习 JVM 的时候有所帮助。

> - 参考链接：https://github.com/heibaiying/Full-Stack-Notes
> - 整理：沉默王二