

目录

JVM 与 Java 体系结构	17
前言	17
架构师每天都在思考什么？	18
为什么要学习 JVM	18
Java vs C++	19
推荐书籍	20
Java 生态圈	20
字节码	22
多语言混合编程	22
Java 发展的重大事件	22
虚拟机与 Java 虚拟机	24
虚拟机	24
Java 虚拟机	24
JVM 的位置	25
JVM 整体结构	26
Java 代码执行流程	28
JVM 的架构模型	28
举例	29
字节码反编译	29
总结	30
栈	30
JVM 生命周期	31
虚拟机的启动	31
虚拟机的执行	31

虚拟机的退出	31
JVM 发展历程.....	31
Sun Classic VM	31
Exact VM.....	32
HotSpot VM	32
JRockit	33
IBM 的 J9.....	33
KVM 和 CDC / CLDC Hotspot.....	33
Azul VM.....	34
Liquid VM.....	34
Apache Marmory	34
Micorsoft JVM.....	34
Taobao JVM	35
Dalvik VM	35
Graal VM	36
总结	36
类加载子系统	37
概述.....	37
类加载器子系统作用	38
类的加载过程.....	40
加载阶段	41
加载 class 文件的方式.....	42
链接阶段.....	42
准备 Prepare.....	44
解析 Resolve	44
初始化阶段	45

类加载器的分类.....	47
虚拟机自带的加载器.....	49
扩展类加载器（Extension ClassLoader）	49
应用程序类加载器（系统类加载器，AppClassLoader）	49
用户自定义类加载器.....	50
查看根加载器所能加载的目录.....	50
关于 ClassLoader	51
双亲委派机制.....	52
工作原理.....	53
双亲委派机制举例.....	53
沙箱安全机制.....	54
双亲委派机制的优势	54
其它.....	54
如何判断两个 class 对象是否相同	54
类的主动使用和被动使用	55
运行时数据区概述及线程	55
前言	55
线程.....	59
JVM 系统线程	59
程序计数器	60
介绍	60
作用	61
代码演示	62
使用 PC 寄存器存储字节码指令地址有什么用呢？	63
PC 寄存器为什么被设定为私有的？	63

CPU 时间片	64
虚拟机栈.....	65
虚拟机栈概述.....	65
Java 虚拟机栈是什么	66
生命周期.....	67
作用	67
栈的特点.....	67
开发中遇到哪些异常？	68
设置栈内存大小	69
栈的存储单位.....	69
栈中存储什么？	69
栈运行原理.....	71
栈帧的内部结构	72
局部变量表	73
关于 Slot 的理解.....	74
Slot 的重复利用.....	75
静态变量与局部变量的对比.....	76
操作数栈	77
概念	77
代码追踪.....	78
栈顶缓存技术.....	83
动态链接	84
方法调用：解析与分配.....	85
链接	85
绑定机制.....	85

早晚期绑定的发展历史	86
虚方法和非虚方法.....	86
invokedynamic 指令	87
动态类型语言和静态类型语言	87
方法重写的本质	87
方法的调用：虚方法表	88
方法返回地址.....	89
一些附加信息.....	90
栈的相关面试题.....	90
本地方法接口	92
什么是本地方法.....	92
为什么使用 Native Method?	93
与 Java 环境的交互.....	93
与操作系统的交互.....	94
Sun's Java	94
现状.....	94
本地方法栈	94
堆	96
堆的核心概念.....	96
堆内存细分	98
设置堆内存大小与 OOM.....	100
OutOfMemory 举例	101
年轻代与老年代.....	102
图解对象分配过程	104
概念	104

图解过程.....	105
思考：幸存区区满了后？	106
对象分配的特殊情况.....	107
代码演示对象分配过程.....	107
常用的调优工具	108
总结	109
Minor GC， MajorGC、 Full GC.....	109
Minor GC.....	110
Major GC	110
Full GC	111
GC 举例.....	111
堆空间分代思想.....	112
内存分配策略.....	113
为对象分配内存： TLAB.....	114
问题：堆空间都是共享的么？	114
为什么有 TLAB？	114
什么是 TLAB.....	114
TLAB 分配过程.....	115
小结：堆空间的参数设置.....	116
堆是分配对象的唯一选择么？	117
逃逸分析.....	117
栈上分配.....	120
同步省略.....	122
分离对象和标量替换.....	123
代码优化之标量替换.....	124

逃逸分析的不足	124
小结	125
方法区	125
前言	125
栈、堆、方法区的交互关系	126
方法区的理解	127
HotSpot 中方法区的演进	128
设置方法区大小与 OOM	129
jdk7 及以前	129
JDK8 以后	129
如何解决这些 OOM	130
方法区的内部结构	131
类型信息	131
域信息	132
方法 (Method) 信息	132
non-final 的类变量	132
全局常量	133
运行时常量池 VS 常量池	133
常量池	134
运行时常量池	135
方法区使用举例	136
方法区的演进细节	140
为什么永久代要被元空间替代?	142
StringTable 为什么要调整位置	143
静态变量存放在那里?	143

方法区的垃圾回收	144
总结	145
常见面试题	145
对象实例化内存布局与访问定位	146
对象实例化	146
面试题	146
对象创建方式	146
创建对象的步骤	147
初始化分配到的内存	148
设置对象的对象头	148
执行 init 方法进行初始化	148
对象实例化的过程	148
对象内存布局	149
对象头	149
实例数据（Instance Data）	150
小结	150
对象的访问定位	150
图示	150
对象访问的两种方式	151
直接内存 Direct Memory	152
非直接缓存区和缓存区	153
存在的问题	153
执行引擎	154
执行引擎概述	154
执行引擎的工作流程	156

Java 代码编译和执行过程	157
什么是解释器（Interpreter）	159
什么是 IT 编译器	159
为什么 Java 是半编译半解释型语言	159
机器码、指令、汇编语言	159
机器码	159
指令	160
指令集	160
汇编语言	160
高级语言	160
C、C++源程序执行过程	161
字节码	162
解释器	163
解释器分类	163
现状	164
JIT 编译器	164
Java 代码的执行分类	164
问题来了	164
HotSpot JVM 执行方式	165
案例	165
概念解释	166
热点探测技术	166
方法调用计数器	167
热点衰减	168
回边计数器	169

HotSpotVM 可以设置程序执行方法.....	169
HotSpotVM 中 JIT 分类.....	170
C1 和 C2 编译器不同的优化策略	171
分层编译策略	171
总结	171
AOT 编译器.....	171
写到最后	172
StringTable	172
String 的基本特性.....	172
为什么 JDK9 改变了结构.....	172
String 的不可变性	173
面试题	174
注意	175
String 的内存分配.....	175
为什么 StringTable 从永久代调整到堆中	177
String 的基本操作.....	177
字符串拼接操作	177
底层原理.....	178
拼接操作和 append 性能对比	179
intern()的使用	180
intern 的空间效率测试	181
面试题	182
new String("ab") 会创建几个对象	182
new String("a") + new String("b") 会创建几个对象	182
intern 的使用：JDK6 和 JDK7.....	183

扩展	184
总结	185
StringTable 的垃圾回收	187
G1 中的 String 去重操作	187
描述	187
实现	187
开启	188
垃圾回收概述	188
概念	188
什么是垃圾	189
什么是垃圾？	191
磁盘碎片整理	191
大厂面试题	191
为什么需要 GC	192
早期垃圾回收	193
Java 垃圾回收机制	193
优点	193
担忧	194
GC 主要关注的区域	194
垃圾回收相关算法	195
标记阶段：引用计数算法	195
循环引用	195
举例	196
小结	198
标记阶段：可达性分析算法	198

概念	198
思路	199
GC Roots 可以是哪些?	201
注意	203
对象的 finalization 机制	203
注意	203
生存还是死亡?	203
具体过程	204
代码演示	205
MAT 与 JProfiler 的 GC Roots 溯源	206
MAT 是什么?	206
命令行使用 jmap	207
使用 JVIsualVM	207
使用 MAT 打开 Dump 文件	207
JProfiler 的 GC Roots 溯源	208
如何判断什么原因造成 OOM	208
清除阶段: 标记-清除算法	210
执行过程	210
什么是清除?	211
缺点	211
清除阶段: 复制算法	212
背景	212
核心思想	212
优点	213
缺点	213

注意	213
清除阶段：标记-整理算法	214
背景	214
执行过程	214
标清和标整的区别	215
标整的优缺点	215
小结	216
分代收集算法	216
增量收集算法	217
概述	217
缺点	218
分区算法	218
写到最后	218
垃圾回收相关概念	219
System.gc()的理解	219
手动 GC 来理解不可达对象的回收	219
内存溢出	221
内存泄漏	222
举例	223
Stop The World	223
垃圾回收的并行与并发	224
并发	224
并行	224
并发和并行对比	225
垃圾回收的并行与并发	225

安全点与安全区域	226
安全点	226
安全区域.....	227
再谈引用	227
再谈引用： 强引用	228
举例	229
总结	230
再谈引用： 软引用	230
再谈引用： 弱引用	230
再谈引用： 虚引用	231
案例	232
终结器引用	234
垃圾回收器	234
GC 分类与性能指标.....	234
垃圾收集器分类	234
评估 GC 的性能指标	236
性能指标： 吞吐量.....	237
性能指标： 暂停时间.....	237
吞吐量 vs 暂停时间	238
不同的垃圾回收器概述.....	238
垃圾回收器发展史.....	239
7 种经典的垃圾收集器.....	239
7 款经典收集器与垃圾分代之间的关系	240
垃圾收集器的组合关系	241
如何查看默认垃圾收集器	241

Serial 回收器：串行回收	242
总结	243
ParNew 回收器：并行回收	243
Parallel 回收器：吞吐量优先	244
参数配置	245
CMS 回收器：低延迟	246
CMS 为什么不使用标记整理算法？	248
优点	248
缺点	248
设置的参数	248
小结	249
JDK 后续版本中 CMS 的变化	249
G1 回收器：区域化分代式	250
既然我们已经有了前面几个强大的 GC，为什么还要发布 Garbage First (G1)？	250
为什么名字叫 Garbage First(G1)呢？	250
G1 垃圾收集器的优点	250
G1 垃圾收集器的缺点	252
G1 参数设置	253
G1 收集器的常见操作步骤	253
G1 收集器的适用场景	253
分区 Region：化整为零	254
G1 垃圾回收器的回收过程	255
Remembered Set（记忆集）	256
G1 回收过程-年轻代 GC	257

G1 回收过程-并发标记过程.....	258
G1 回收过程 - 混合回收.....	259
G1 回收可选的过程 4 - Full GC	260
G1 回收的优化建议	260
垃圾回收器总结.....	261
怎么选择垃圾回收器.....	261
面试	262
GC 日志分析	262
verbose:gc.....	263
PrintGCDetails.....	263
补充	264
Young GC 图片	265
FullGC 图片、	265
GC 回收举例.....	265
垃圾回收器的新发展	267
Open JDK12 的 Shenandoash GC	268
革命性的 ZGC	269
AliGC.....	272

JVM 与 Java 体系结构

前言

作为 Java 工程师的你曾被伤害过吗？你是否也遇到过这些问题？

运行着的线上系统突然卡死，系统无法访问，甚至直接 OOMM！

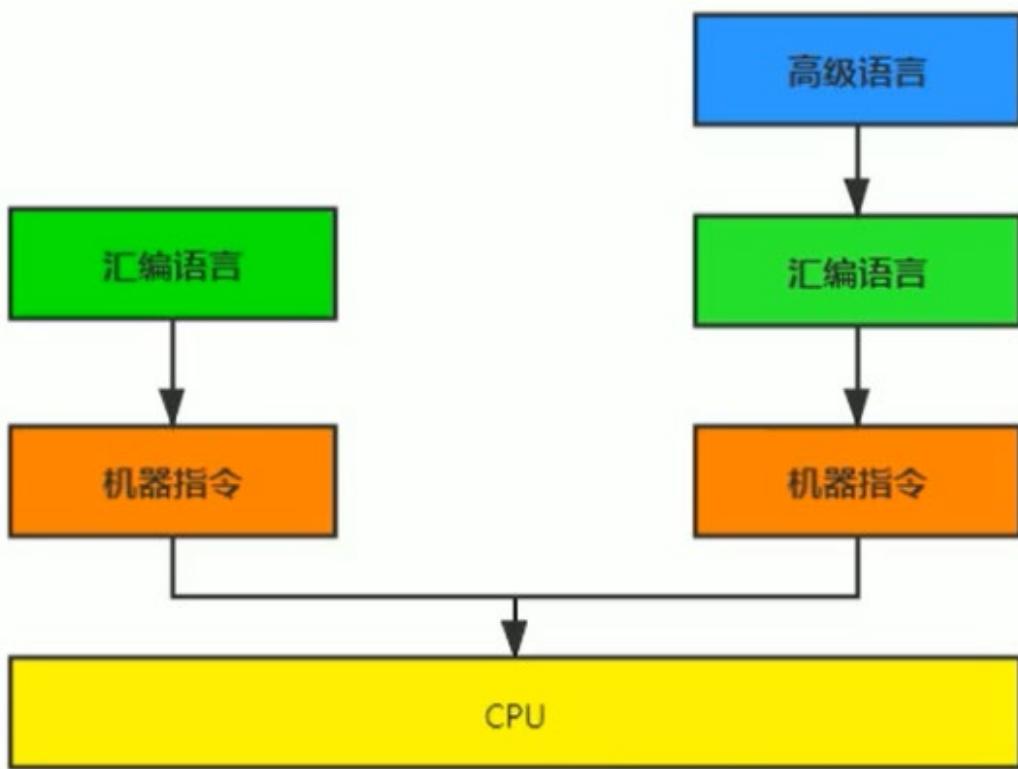
- 想解决线上 JVM GC 问题，但却无从下手。
- 新项目上线，对各种 JVM 参数设置一脸茫然，直接默认吧然后就 JJ 了
- 每次面试之前都要重新背一遍 JVM 的一些原理概念性的东西，然而面试官却经常问你在实际项目中如何调优 VM 参数，如何解决 GC、OOM 等问题，一脸懵逼。



大部分 Java 开发人员，除会在项目中使用到与 Java 平台相关的各种高精尖技术，对于 Java 技术的核心 Java 虚拟机了解甚少。

一些有一定工作经验的开发人员，打心眼儿里觉得 SSM、微服务等上层技术才是重点，基础技术并不重要，这其实是一种本末倒置的“病态”。如果我们把核心类库的 API 比做数学公式的话，那么 Java 虚拟机的知识就好比公式的推导过程。

计算机系统体系对我们来说越来越远，在不了解底层实现方式的前提下，通过高级语言很容易编写程序代码。但事实上计算机并不认识高级语言



架构师每天都在思考什么？

- 应该如何让我的系统更快？
- 如何避免系统出现瓶颈？

知乎上有条帖子：应该如何看招聘信息，直通年薪 50 万+？

- 参与现有系统的性能优化，重构，保证平台性能和稳定性
- 根据业务场景和需求，决定技术方向，做技术选型
- 能够独立架构和设计海量数据下高并发分布式解决方案，满足功能和非功能需求
- 解决各类潜在系统风险，核心功能的架构与代码编写
- 分析系统瓶颈，解决各种疑难杂症，性能调优等

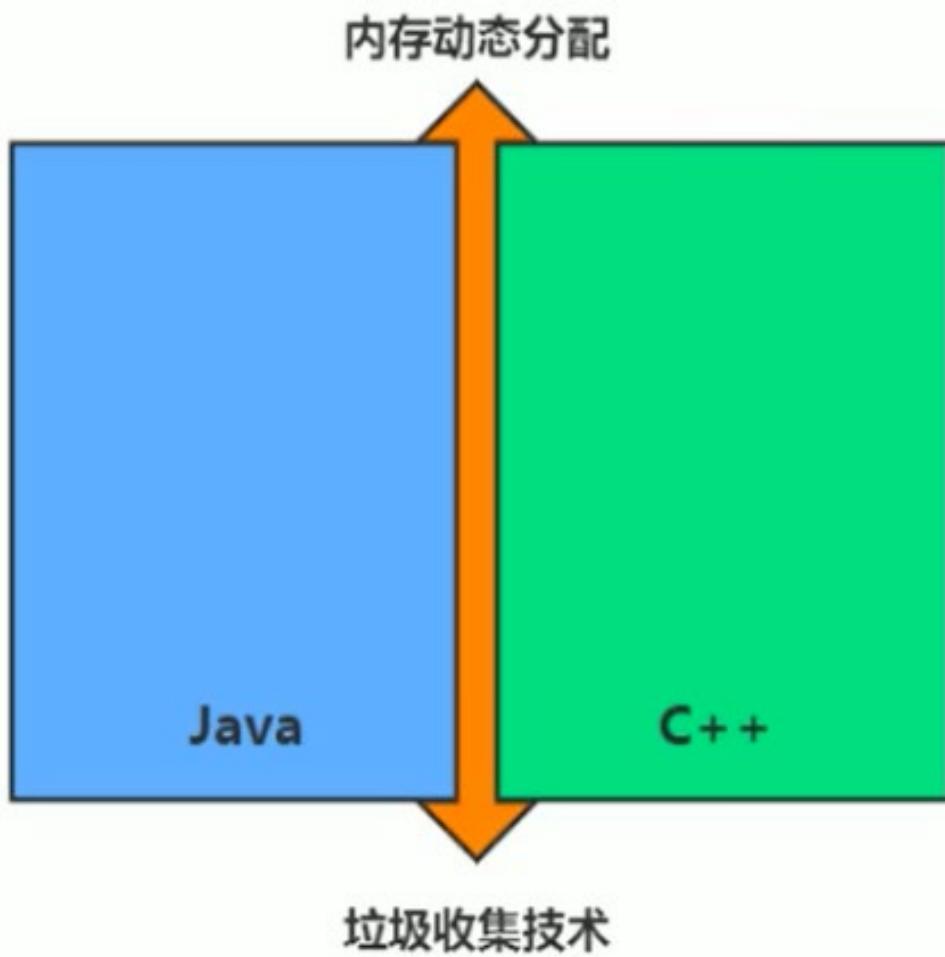
为什么要学习 JVM

- 面试的需要（BATJ、TMD、PKQ 等面试都爱问）

- 中高级程序员必备技能
 - 项目管理、调优的需求
- 追求极客的精神
 - 比如：垃圾回收算法、JIT（及时编译器）、底层原理

Java vs C++

垃圾收集机制为我们打理了很多繁琐的工作，大大提高了开发的效率，但是，垃圾收集也不是万能的，懂得 JVM 内部的内存结构、工作机制，是设计高扩展性应用和诊断运行时问题的基础，也是 Java 工程师进阶的必备能力。



C 语言需要自己来分配内存和回收内存，Java 全部交给 JVM 进行分配和回收。

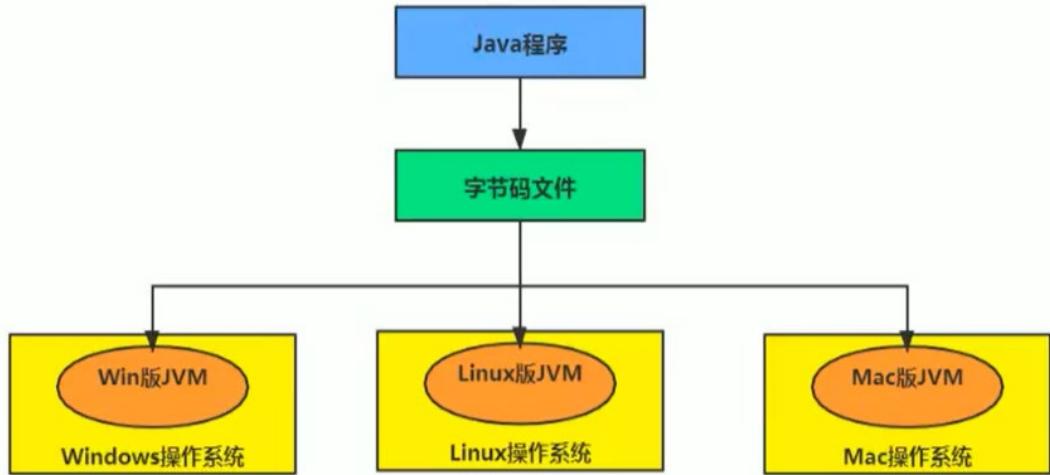
推荐书籍



Java 生态圈

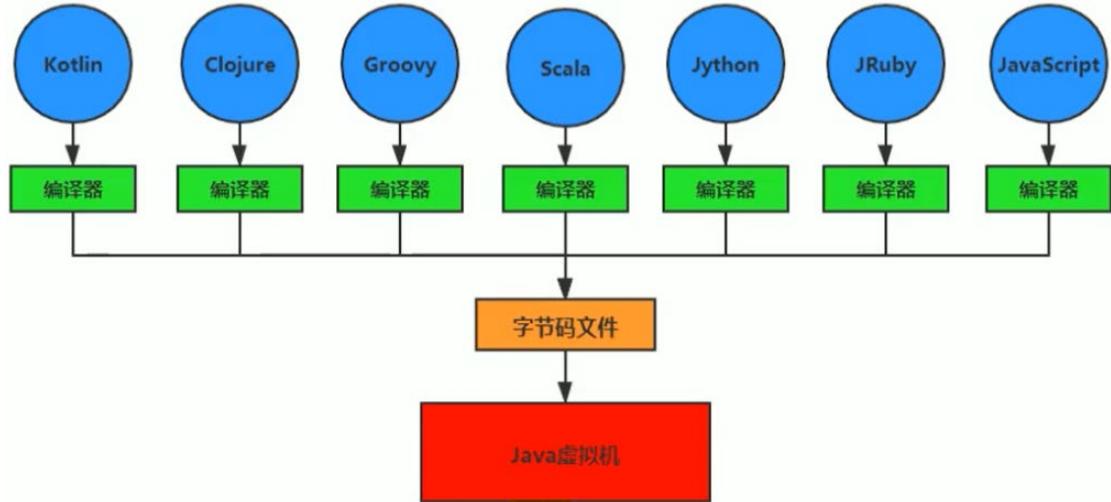
Java 是目前应用最为广泛的软件开发平台之一。随着 Java 以及 Java 社区的不断壮大 Java 也早已不再是简简单单的一门计算机语言了，它更是一个平台、一种文化、一个社区。

- 作为一个平台，Java 虚拟机扮演着举足轻重的作用
 - Groovy、Scala、JRuby、Kotlin 等都是 Java 平台的一部分
- 作为灯种文化，Java 几乎成为了“开源”的代名词。
 - 第三方开源软件和框架。如 Tomcat、Struts, MyBatis, Spring 等。
 - 就连 JDK 和 JVM 自身也有不少开源的实现，如 openJDK、Harmony。
- 作为一个社区，Java 拥有全世界最多的技拥拥护者和开源社区支持，有数不清的论坛和资料。从桌面应用软件、嵌入式开发到企业级应用、后台服务器、中间件，都可以看到 Java 的身影。其应用形式之复杂、参与人数之众多也令人咋舌。



“write once, run anywhere.”

每个语言都需要转换成字节码文件，最后转换的字节码文件都能通过 Java 虚拟机进行运行和处理



随着 Java7 的正式发布，Java 虚拟机的设计者们通过 JSR-292 规范基本实现在 Java 虚拟机平台上运行非 Java 语言编写的程序。

Java 虚拟机根本不关心运行在其内部的程序到底是使用何种编程语言编写的，它只关心“字节码”文件。也就是说 Java 虚拟机拥有语言无关性，并不会单纯地与 Java 语言“终身绑定”，只要其他编程语言的编译结果满足并包含 Java 虚拟机的内

部指令集、符号表以及其他辅助信息，它就是一个有效的字节码文件，就能够被虚拟机所识别并装载运行。

字节码

我们平时说的 java 字节码，指的是用 java 语言编译成的字节码。准确的说任何能在 jvm 平台上执行的字节码格式都是一样的。所以应该统称为：jvm 字节码。

不同的编译器，可以编译出相同的字节码文件，字节码文件也可以在不同的 JVM 上运行。

Java 虚拟机与 Java 语言并没有必然的联系，它只与特定的二进制文件格式—Class 文件格式所关联，Class 文件中包含了 Java 虚拟机指令集（或者称为字节码、Bytecodes）和符号表，还有一些其他辅助信息。

多语言混合编程

Java 平台上的多语言混合编程正成为主流，通过特定领域的语言去解决特定领域的问题是当前软件开发应对日趋复杂的项目需求的一个方向。

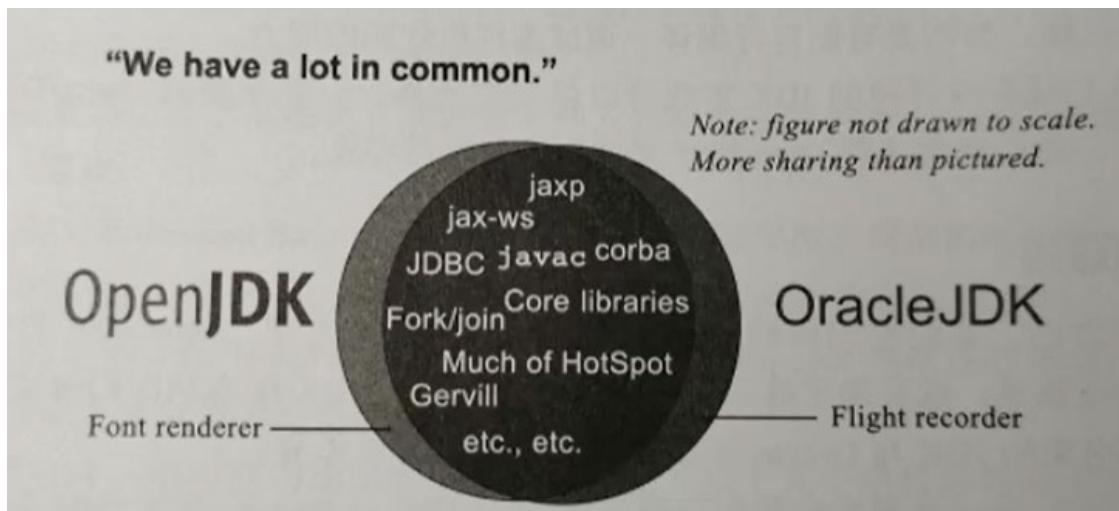
试想一下，在一个项目之中，并行处理用 clojure 语言编写，展示层使用 JRuby/Rails，中间层则是 Java，每个应用层都将使用不同的编程语言来完成，而且，接口对每一层的开发者都是透明的，各种语言之间的交互不存在任何困难，就像使用自己语言的原生 API 一样方便，因为它们最终都运行在一个虚拟机之上。

对这些运行于 Java 虚拟机之上、Java 之外的语言，来自系统级的、底层的支持正在迅速增强，以 JSR-292 为核心的一系列项目和功能改进（如 Da Vinci Machine 项目、Nashorn 引擎、InvokeDynamic 指令、java.lang.invoke 包等），推动 Java 虚拟机从“Java 语言的虚拟机”向“多语言虚拟机”的方向发展。

Java 发展的重大事件

- 1990 年，在 Sun 计算机公司中，由 Patrick Naughton、Mike Sheridan 及 James Gosling 领导的小组 Green Team，开发出新的程序语言，命名为 oak，后期命名为 Java
- 1995 年，Sun 正式发布 Java 和 HotJava 产品，Java 首次公开亮相。
- 1996 年 1 月 23 日 sun Microsystems 发布了 JDK 1.0。
- 1998 年，JDK1.2 版本发布。同时，sun 发布了 JSP/Servlet、EJB 规范，以及将 Java 分成了 J2EE、J2SE 和 J2ME。这表明了 Java 开始向企业、桌面应用和移动设备应用 3 大领域挺进。
- 2000 年，JDK1.3 发布，Java HotSpot Virtual Machine 正式发布，成为 Java 的默认虚拟机。

- 2002 年，JDK1.4 发布，古老的 Classic 虚拟机退出历史舞台。
- 2003 年年底，Java 平台的 scala 正式发布，同年 Groovy 也加入了 Java 阵营。
- 2004 年，JDK1.5 发布。同时 JDK1.5 改名为 JavaSE5.0。
- 2006 年，JDK6 发布。同年，Java 开源并建立了 openJDK。顺理成章，Hotspot 虚拟机也成为了 openJDK 中的默认虚拟机。
- 2007 年，Java 平台迎来了新伙伴 Clojure。
- 2008 年，oracle 收购了 BEA，得到了 JRockit 虚拟机。
- 2009 年，Twitter 宣布把后台大部分程序从 Ruby 迁移到 scala，这是 Java 平台的又一次大规模应用。
- 2010 年，oracle 收购了 sun，获得 Java 商标和最真价值的 HotSpot 虚拟机。此时，oracle 拥有市场占有率最高的两款虚拟机 HotSpot 和 JRockit，并计划在未来对它们进行整合：HotRockit
- 2011 年，JDK7 发布。在 JDK1.7u4 中，正式启用了新的垃圾回收器 G1。
- 2017 年，JDK9 发布。将 G1 设置为默认 Gc，替代 CMS
- 同年，IBM 的 J9 开源，形成了现在的 open J9 社区
- 2018 年，Android 的 Java 侵权案判决，Google 赔偿 oracle 计 88 亿美元
- 同年，oracle 宣告 JavagE 成为历史名词 JDBC、JMS、Servlet 赠予 Eclipse 基金会
- 同年，JDK11 发布，LTS 版本的 JDK，发布革命性的 zGc，调整 JDK 授权许可
- 2019 年，JDK12 发布，加入 RedHat 领导开发的 shenandoah GC



在 JDK11 之前，oracleJDK 中还会存在一些 openJDK 中没有的、闭源的功能。但在 JDK11 中，我们可以认为 openJDK 和 oracleJDK 代码实质上已经完全一致的程度。

虚拟机与 Java 虚拟机

虚拟机

所谓虚拟机（Virtual Machine），就是一台虚拟的计算机。它是一款软件，用来执行一系列虚拟计算机指令。大体上，虚拟机可以分为系统虚拟机和程序虚拟机。

- 大名鼎鼎的 Visual Box, Mware 就属于系统虚拟机，它们完全是对物理计算机的仿真，提供了一个可运行完整操作系统的软件平台。
- 程序虚拟机的典型代表就是 Java 虚拟机，它专门为执行单个计算机程序而设计，在 Java 虚拟机中执行的指令我们称为 Java 字节码指令。

无论是系统虚拟机还是程序虚拟机，在上面运行的软件都被限制于虚拟机提供的资源中。

Java 虚拟机

Java 虚拟机是一台执行 Java 字节码的虚拟计算机，它拥有独立的运行机制，其运行的 Java 字节码也未必由 Java 语言编译而成。

JVM 平台的各种语言可以共享 Java 虚拟机带来的跨平台性、优秀的垃圾回收器，以及可靠的即时编译器。

Java 技术的核心就是 Java 虚拟机（JVM, Java Virtual Machine），因为所有的 Java 程序都运行在 Java 虚拟机内部。

Java 虚拟机就是二进制字节码的运行环境，负责装载字节码到其内部，解释/编译为对应平台上的机器指令执行。每一条 Java 指令，Java 虚拟机规范中都有详细定义，如怎么取操作数，怎么处理操作数，处理结果放在哪里。

特点：

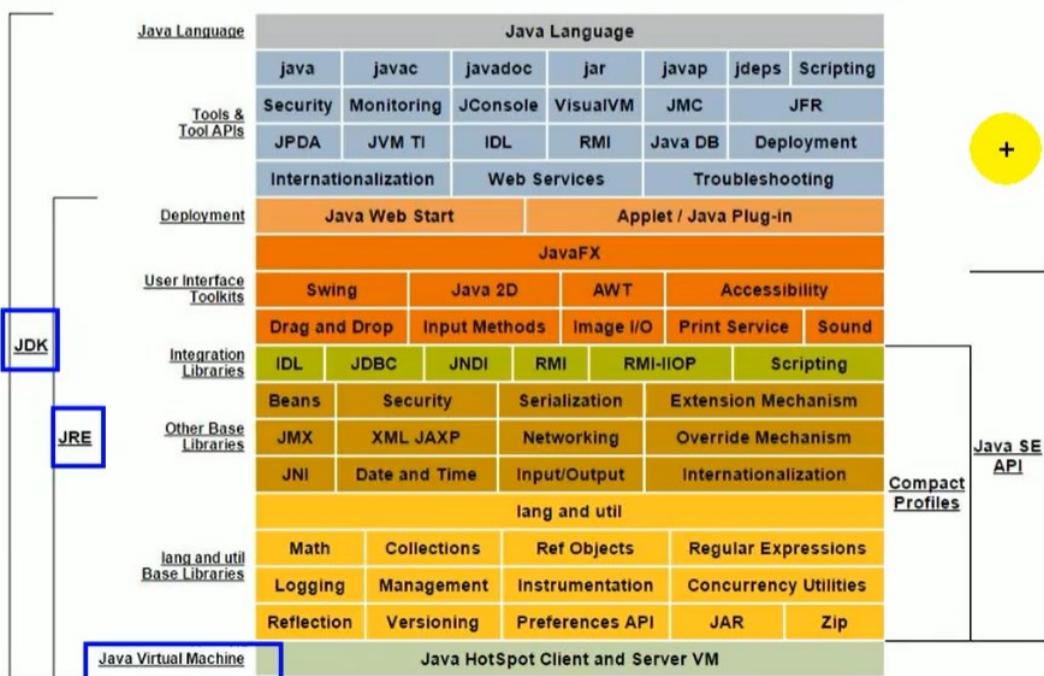
- 一次编译，到处运行
- 自动内存管理
- 自动垃圾回收功能

JVM 的位置

JVM 是运行在操作系统之上的，它与硬件没有直接的交互

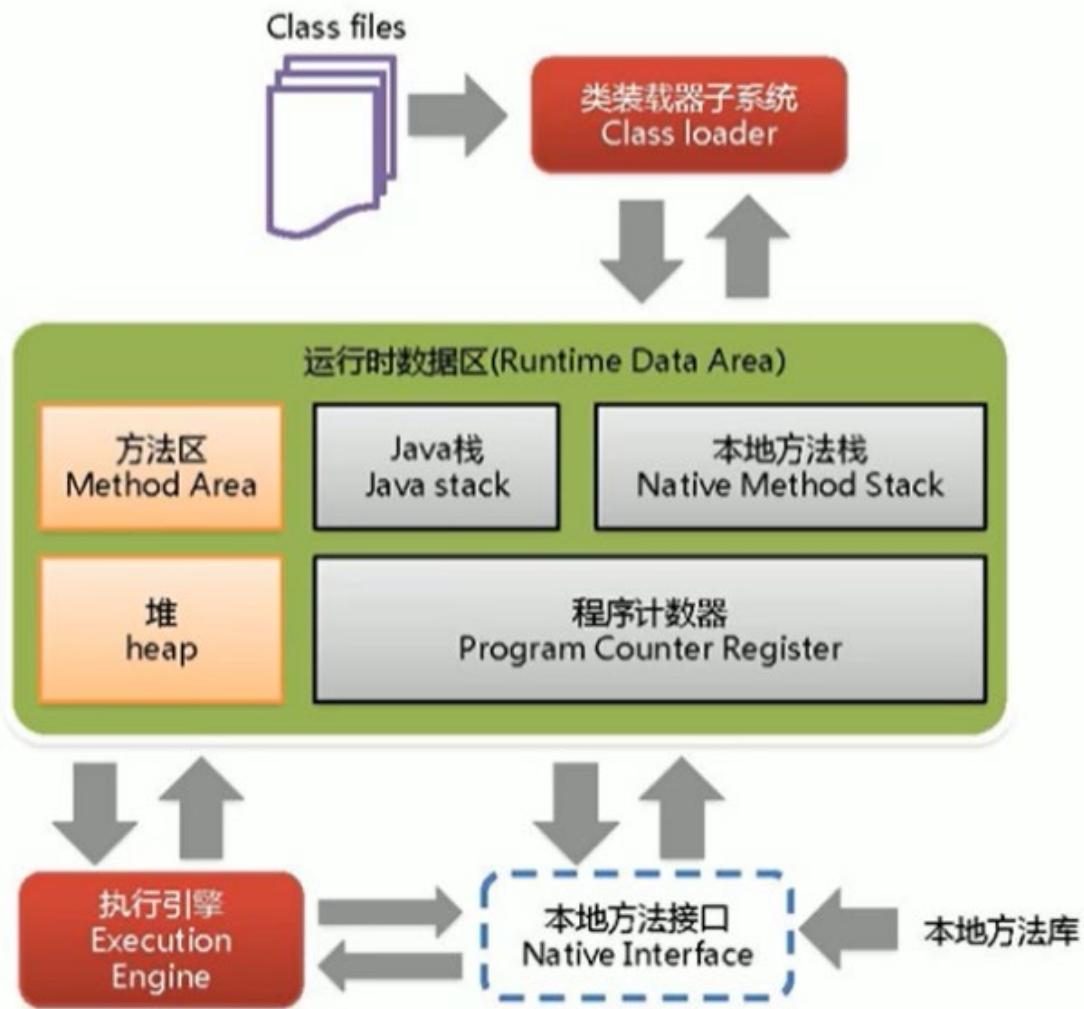


Java 的体系结构



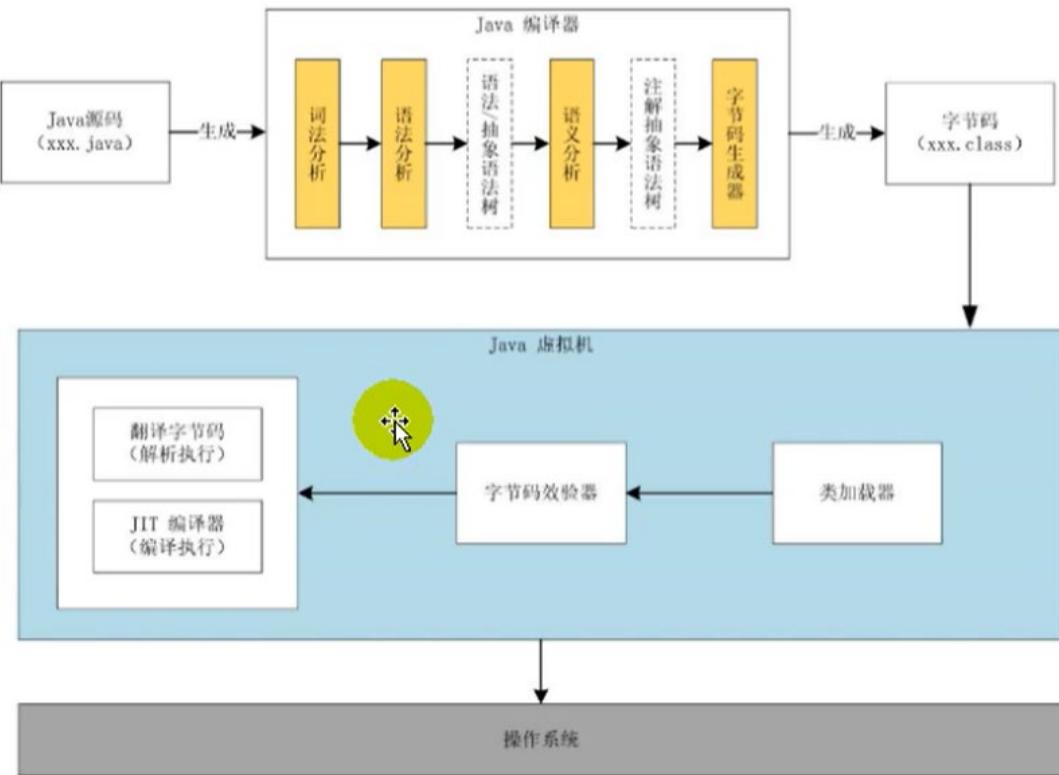
JVM 整体结构

- HotSpot VM 是目前市面上高性能虚拟机的代表作之一。
- 它采用解释器与即时编译器并存的架构。
- 在今天，Java 程序的运行性能早已脱胎换骨，已经达到了可以和 C/C++ 程序一较高下的地步。



执行引擎包含三部分：解释器，及时编译器，垃圾回收器

Java 代码执行流程



只是能生成被 Java 虚拟机所能解释的字节码文件，那么理论上就可以自己设计一套代码了

JVM 的架构模型

Java 编译器输入的指令流基本上是一种基于栈的指令集架构，另外一种指令集架构则是基于寄存器的指令集架构。具体来说：这两种架构之间的区别：

基于栈式架构的特点

- 设计和实现更简单，适用于资源受限的系统；
- 避开了寄存器的分配难题：使用零地址指令方式分配。
- 指令流中的指令大部分是零地址指令，其执行过程依赖于操作栈。指令集更小，编译器容易实现。
- 不需要硬件支持，可移植性更好，更好实现跨平台

基于寄存器架构的特点

- 典型的应用是 x86 的二进制指令集：比如传统的 PC 以及 Android 的 Davlik 虚拟机。
- 指令集架构则完全依赖硬件，可移植性差
- 性能优秀和执行更高效
- 花费更少的指令去完成一项操作。
- 在大部分情况下，基于寄存器架构的指令集往往都以一地址指令、二地址指令和三地址指令为主，而基于栈式架构的指令集却是以零地址指令为主方水洋

举例

同样执行 $2+3$ 这种逻辑操作，其指令分别如下：

基于栈的计算流程（以 Java 虚拟机为例）：

```
iconst_2 //常量 2 入栈
istore_1
iconst_3 // 常量 3 入栈
istore_2
iload_1
iload_2
iadd //常量 2/3 出栈，执行相加
istore_0 // 结果 5 入栈
```

而基于寄存器的计算流程

```
mov eax,2 //将 eax 寄存器的值设为 1
add eax,3 //使 eax 寄存器的值加 3
```

字节码反编译

我们编写一个简单的代码，然后查看一下字节码的反编译后的结果

```
/**
 * @author: 陌溪
 * @create: 2020-07-04-21:17
 */
public class StackStruTest {
    public static void main(String[] args) {
        int i = 2 + 3;
    }
}
```

然后我们找到编译后的 class 文件，使用下列命令进行反编译

```
javap -v StackStruTest.class
```

得到的文件为：

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=4, args_size=1
    0: iconst_2
    1: istore_1
    2: iconst_3
    3: istore_2
    4: iload_1
    5: iload_2
    6: iadd
    7: istore_3
    8: return
LineNumberTable:
  line 9: 0
  line 10: 2
  line 11: 4
  line 12: 8
LocalVariableTable:
  Start  Length  Slot  Name   Signature
    0        9      0  args   [Ljava/lang/String;
    2        7      1      i     I
    4        5      2      j     I
    8        1      3      k     I
```

总结

由于跨平台性的设计，Java 的指令都是根据栈来设计的。不同平台 CPU 架构不同，所以不能设计为基于寄存器的。优点是跨平台，指令集小，编译器容易实现，缺点是性能下降，实现同样的功能需要更多的指令。

时至今日，尽管嵌入式平台已经不是 Java 程序的主流运行平台了（准确来说应该是 HotSpotVM 的宿主环境已经不局限于嵌入式平台了），那么为什么不将架构更换为基于寄存器的架构呢？

栈

- 跨平台性
- 指令集小
- 指令多
- 执行性能比寄存器差

JVM 生命周期

虚拟机的启动

Java 虚拟机的启动是通过引导类加载器（bootstrap class loader）创建一个初始类（initial class）来完成的，这个类是由虚拟机的具体实现指定的。

虚拟机的执行

- 一个运行中的 Java 虚拟机有着一个清晰的任务：执行 Java 程序。
- 程序开始执行时他才运行，程序结束时他就停止。
- 执行一个所谓的 Java 程序的时候，真真正正在执行的是一个叫做 Java 虚拟机的进程。

虚拟机的退出

有如下的几种情况：

- 程序正常执行结束
- 程序在执行过程中遇到了异常或错误而异常终止
- 由于操作系统发现错误而导致 Java 虚拟机进程终止
- 某线程调用 Runtime 类或 system 类的 exit 方法，或 Runtime 类的 halt 方法，并且 Java 安全管理器也允许这次 exit 或 halt 操作。
- 除此之外，JNI（Java Native Interface）规范描述了用 JNI Invocation API 来加载或卸载 Java 虚拟机时，Java 虚拟机的退出情况。

JVM 发展历程

Sun Classic VM

- 早在 1996 年 Java1.0 版本的时候，Sun 公司发布了一款名为 sun classic VM 的 Java 虚拟机，它同时也是世界上第一款商用 Java 虚拟机，JDK1.4 时完全被淘汰。
- 这款虚拟机内部只提供解释器。现在还有及时编译器，因此效率比较低，而及时编译器会把热点代码缓存起来，那么以后使用热点代码的时候，效率就比较高。
- 如果使用 JIT 编译器，就需要进行外挂。但是一旦使用了 JIT 编译器，JIT 就会接管虚拟机的执行系统。解释器就不再工作。解释器和编译器不能配合工作。
- 现在 hotspot 内置了此虚拟机。

Exact VM

为了解决上一个虚拟机问题，jdk1.2 时，sun 提供了此虚拟机。Exact Memory Management: 准确式内存管理

- 也可以叫 Non-Conservative/Accurate Memory Management
- 虚拟机可以知道内存中某个位置的数据具体是什么类型。|

具备现代高性能虚拟机的维形

- 热点探测（寻找出热点代码进行缓存）
- 编译器与解释器混合工作模式

只在 solaris 平台短暂使用，其他平台上还是 classic vm，英雄气短，终被 Hotspot 虚拟机替换

HotSpot VM

HotSpot 历史

- 最初由一家名为“Longview Technologies”的小公司设计
- 1997 年，此公司被 sun 收购；2009 年，Sun 公司被甲骨文收购。
- JDK1.3 时，HotSpot VM 成为默认虚拟机

目前 Hotspot 占有绝对的市场地位，称霸武林。

- 不管是现在仍在广泛使用的 JDK6，还是使用比例较多的 JDK8 中，默认的虚拟机都是 HotSpot
- Sun/oracle JDK 和 openJDK 的默认虚拟机
- 因此本课程中默认介绍的虚拟机都是 HotSpot，相关机制也主要是指 HotSpot 的 Gc 机制。（比如其他两个商用虚机都没有方法区的概念）

从服务器、桌面到移动端、嵌入式都有应用。

名称中的 HotSpot 指的就是它的热点代码探测技术。

- 通过计数器找到最具编译价值代码，触发即时编译或栈上替换
- 通过编译器与解释器协同工作，在最优化的程序响应时间与最佳执行性能中取得平衡

JRockit

专注于服务器端应用

- 它可以不太关注程序启动速度，因此 JRockit 内部不包含解析器实现，全部代码都靠即时编译器编译后执行。

大量的行业基准测试显示，JRockit JVM 是世界上最快的 JVM。

- 使用 JRockit 产品，客户已经体验到了显著的性能提高（一些超过了 70%）和硬件成本的减少（达 50%）。

优势：全面的 Java 运行时解决方案组合

- JRockit 面向延迟敏感型应用的解决方案 JRockit Real Time 提供以毫秒或微秒级的 JVM 响应时间，适合财务、军事指挥、电信网络的需要
- MissionControl 服务套件，它是一组以极低的开销来监控、管理和分析生产环境中的应用程序的工具。

2008 年，JRockit 被 oracle 收购。

oracle 表达了整合两大优秀虚拟机的工作，大致在 JDK8 中完成。整合的方式是在 HotSpot 的基础上，移植 JRockit 的优秀特性。

高斯林：目前就职于谷歌，研究人工智能和水下机器人

IBM 的 J9

全称：IBM Technology for Java Virtual Machine，简称 IT4J，内部代号：J9

市场定位与 HotSpot 接近，服务器端、桌面应用、嵌入式等多用途 VM 广泛用于 IBM 的各种 Java 产品。

目前，有影响力的三大商用虚拟机之一，也号称是世界上最快的 Java 虚拟机。

2017 年左右，IBM 发布了开源 J9VM，命名为 openJ9，交给 Eclipse 基金会管理，也称为 Eclipse OpenJ9

OpenJDK -> 是 JDK 开源了，包括了虚拟机

KVM 和 CDC / CLDC Hotspot

oracle 在 Java ME 产品线上的两款虚拟机为：CDC/CLDC HotSpot Implementation
VM KVM（Kilobyte）是 CLDC-HI 早期产品目前移动领域地位尴尬，智能机被 Android 和 iOS 二分天下。

KVM 简单、轻量、高度可移植，面向更低端的设备上还维持自己的一片市场

- 智能控制器、传感器
- 老人手机、经济欠发达地区的功能手机

所有的虚拟机的原则：一次编译，到处运行。

Azul VM

前面三大“高性能 Java 虚拟机”使用在通用硬件平台上这里 Azu1VW 和 BEALiquid VM 是与特定硬件平台绑定、软硬件配合的专有虚拟机 I

- 高性能 Java 虚拟机中的战斗机。

Azul VM 是 Azu1Systems 公司在 HotSpot 基础上进行大量改进，运行于 Azul Systems 公司的专有硬件 Vega 系统上的 ava 虚拟机。

每个 Azu1VM 实例都可以管理至少数十个 CPU 和数百 GB 内存的硬件资源，并提供在巨大内存范围内实现可控的 GC 时间的垃圾收集器、专有硬件优化的线程调度等优秀特性。

2010 年，AzulSystems 公司开始从硬件转向软件，发布了自己的 zing JVM，可以在通用 x86 平台上提供接近于 Vega 系统的特性。

Liquid VM

高性能 Java 虚拟机中的战斗机。

BEA 公司开发的，直接运行在自家 Hypervisor 系统上 Liquid VM 即是现在的 JRockit VE（Virtual Edition），

Liquid VM 不需要操作系统的支持，或者说它自己本身实现了一个专用操作系统的必要功能，如线程调度、文件系统、网络支持等。

随着 JRockit 虚拟机终止开发，Liquid VM 项目也停止了。

Apache Harmony

Apache 也曾经推出过与 JDK1.5 和 JDK1.6 兼容的 Java 运行平台 Apache Harmony。

它是 IElf 和 Inte1 联合开发的开源 JVM，受到同样开源的 openJDK 的压制，Sun 坚决不让 Harmony 获得 JCP 认证，最终于 2011 年退役，IBM 转而参与 OpenJDK

虽然目前并没有 Apache Harmony 被大规模商用的案例，但是它的 Java 类库代码吸纳进了 Android SDK。

Micorsft JVM

微软为了在 IE3 浏览器中支持 Java Applets，开发了 Microsoft JVM。

只能在 window 平台下运行。但确是当时 Windows 下性能最好的 Java VM。

1997 年，sun 以侵犯商标、不正当竞争罪名指控微软成功，赔了 sun 很多钱。微软 windowsXPSP3 中抹掉了其 VM。现在 windows 上安装的 jdk 都是 HotSpot。

Taobao JVM

由 AliJVM 团队发布。阿里，国内使用 Java 最强大的公司，覆盖云计算、金融、物流、电商等众多领域，需要解决高并发、高可用、分布式的复合问题。有大量的开源产品。

基于 openJDK 开发了自己的定制版本 AlibabaJDK，简称 AJDK。是整个阿里 Java 体系的基石。

基于 openJDK Hotspot VM 发布的国内第一个优化、深度定制且开源的高性能服务器版 Java 虚拟机。

- 创新的 GCIH (GCinvisble heap) 技术实现了 off-heap，即将生命周期较长的 Java 对象从 heap 中移到 heap 之外，并且 Gc 不能管理 GCIH 内部的 Java 对象，以此达到降低 GC 的回收频率和提升 Gc 的回收效率的目的。
- GCIH 中的对象还能够在多个 Java 虚拟机进程中实现共享
- 使用 crc32 指令实现 JvM intrinsic 降低 JNI 的调用开销
- PMU hardware 的 Java profiling tool 和诊断协助功能
- 针对大数据场景的 ZenGc

taobao vm 应用在阿里产品上性能高，硬件严重依赖 intel 的 cpu，损失了兼容性，但提高了性能

目前已经在淘宝、天猫上线，把 oracle 官方 JvM 版本全部替换了。

Dalvik VM

谷歌开发的，应用于 Android 系统，并在 Android2.2 中提供了 JIT，发展迅猛。

Dalvik 只能称作虚拟机，而不能称作“Java 虚拟机”，它没有遵循 Java 虚拟机规范不能直接执行 Java 的 Class 文件

基于寄存器架构，不是 jvm 的栈架构。

执行的是编译以后的 dex (Dalvik Executable) 文件。执行效率比较高。

- 它执行的 dex (Dalvik Executable) 文件可以通过 class 文件转化而来，使用 Java 语法编写应用程序，可以直接使用大部分的 Java API 等。

Android 5.0 使用支持提前编译（Ahead of Time Compilation, AoT）的 ART VM 替换 Dalvik VM。

Graal VM

2018 年 4 月，oracle Labs 公开了 GraalVM，号称 "Run Programs Faster Anywhere"，勃勃野心。与 1995 年 java 的"write once, run anywhere"遥相呼应。

GraalVM 在 HotSpot VM 基础上增强而成的跨语言全栈虚拟机，可以作为“任何语言”的运行平台使用。语言包括：Java、Scala、Groovy、Kotlin；C、C++、Javascript、Ruby、Python、R 等

支持不同语言中混用对方的接口和对象，支持这些语言使用已经编写好的本地库文件

工作原理是将这些语言的源代码或源代码编译后的中间格式，通过解释器转换为能被 Graal VM 接受的中间表示。Graal VM 提供 Truffle 工具集快速构建面向一种新语言的解释器。在运行时还能进行即时编译优化，获得比原生编译器更优秀的执行效率。

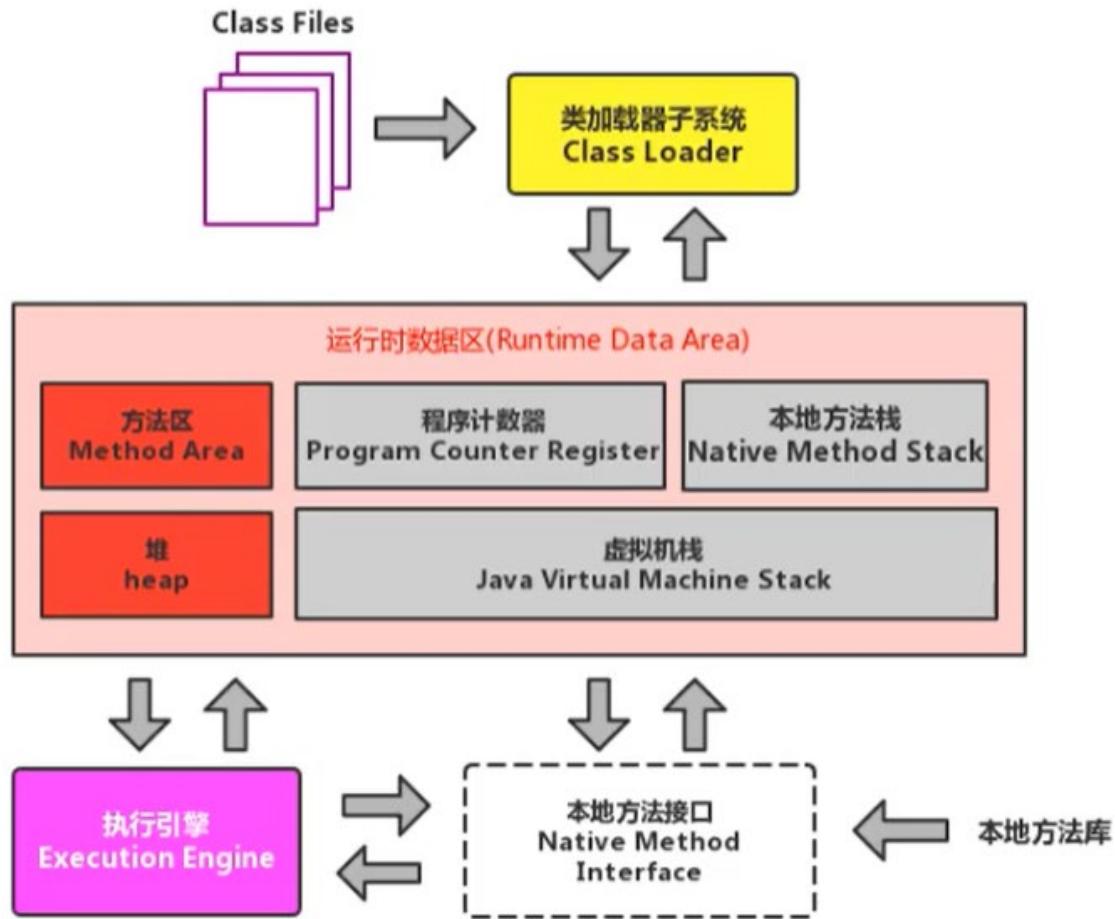
如果说 HotSpot 有一天真的被取代，Graalvm 希望最大。但是 Java 的软件生态没有丝毫变化。

总结

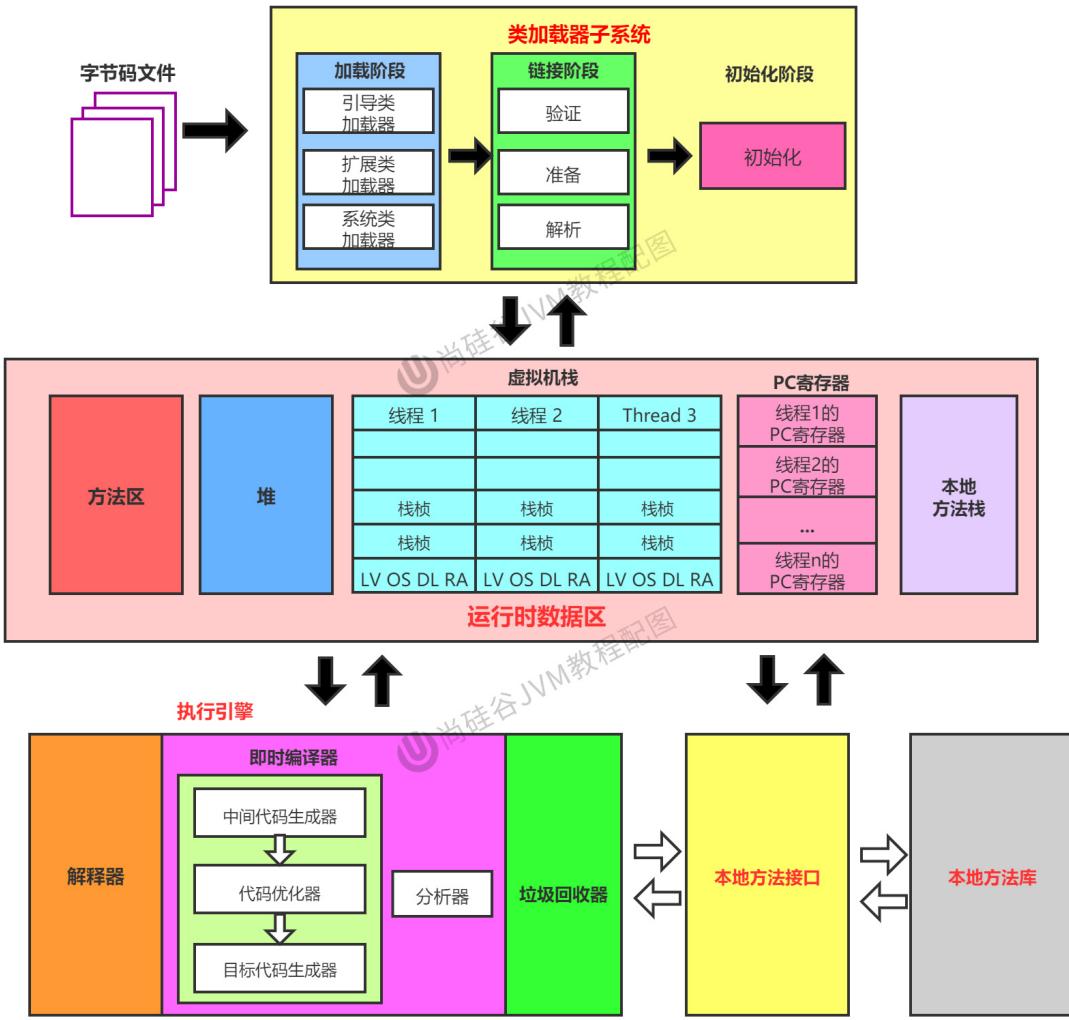
具体 JVM 的内存结构，其实取决于其实现，不同厂商的 JVM，或者同一厂商发布的不同版本，都有可能存在一定差异。主要以 oracle HotSpot VM 为默认虚拟机。

类加载子系统

概述



完整图如下



如果自己想手写一个 Java 虚拟机的话，主要考虑哪些结构呢？

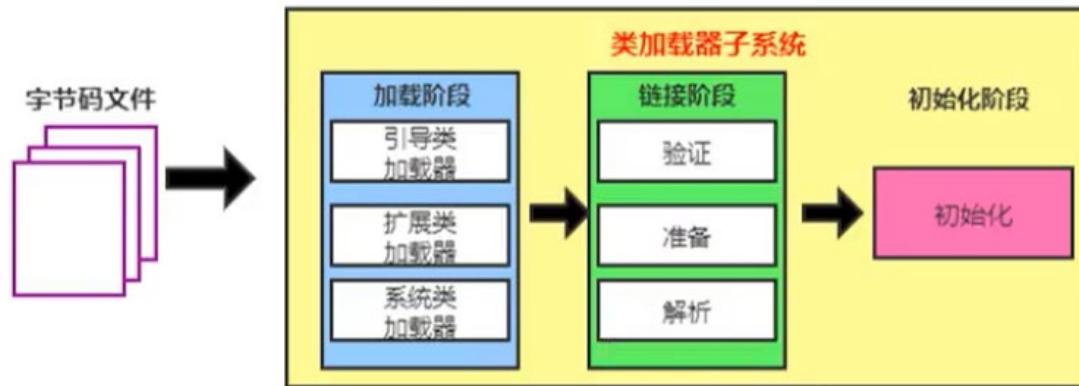
- 类加载器
- 执行引擎

类加载器子系统作用

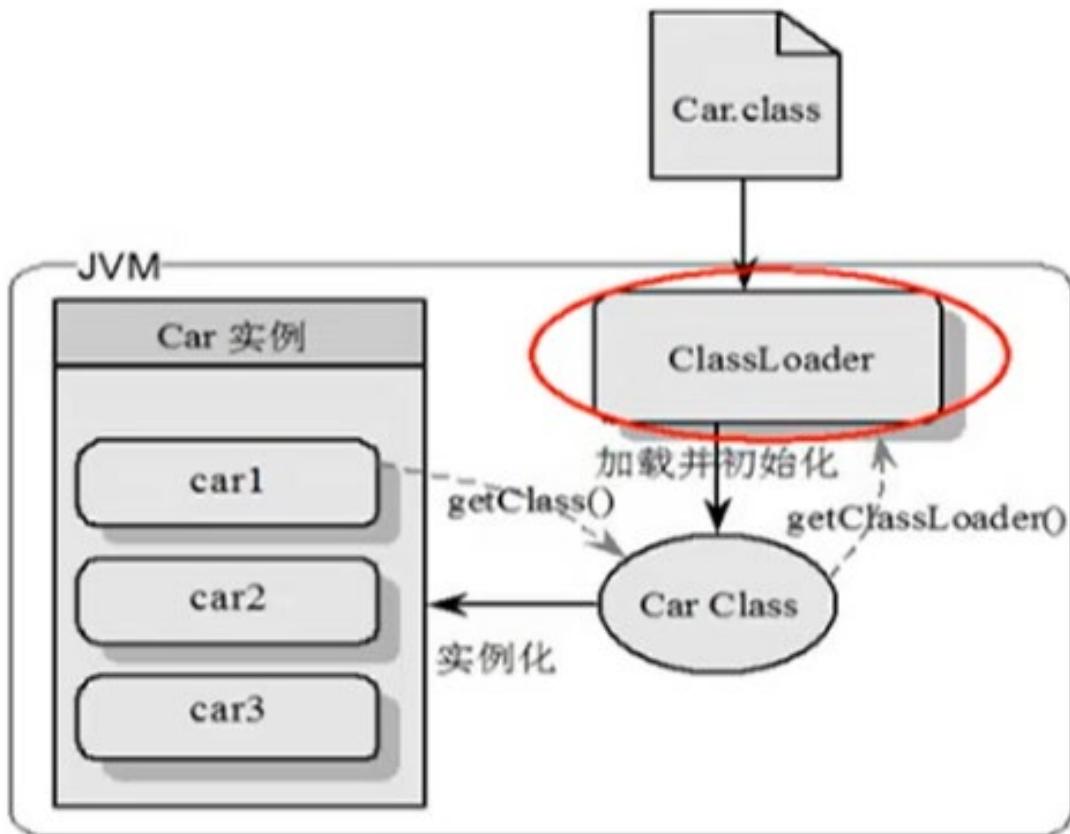
类加载器子系统负责从文件系统或者网络中加载 Class 文件，class 文件在文件开头有特定的文件标识。

ClassLoader 只负责 class 文件的加载，至于它是否可以运行，则由 Execution Engine 决定。

加载的类信息存放于一块称为方法区的内存空间。除了类的信息外，方法区中还会存放运行时常量池信息，可能还包括字符串字面量和数字常量（这部分常量信息是 Class 文件中常量池部分的内存映射）



- class file 存在于本地硬盘上，可以理解为设计师画在纸上的模板，而最终这个模板在执行的时候是要加载到 JVM 当中来根据这个文件实例化出 n 个一模一样的实例。
- class file 加载到 JVM 中，被称为 DNA 元数据模板，放在方法区。
- 在.class 文件->JVM->最终成为元数据模板，此过程就要一个运输工具（类装载器 Class Loader），扮演一个快递员的角色。

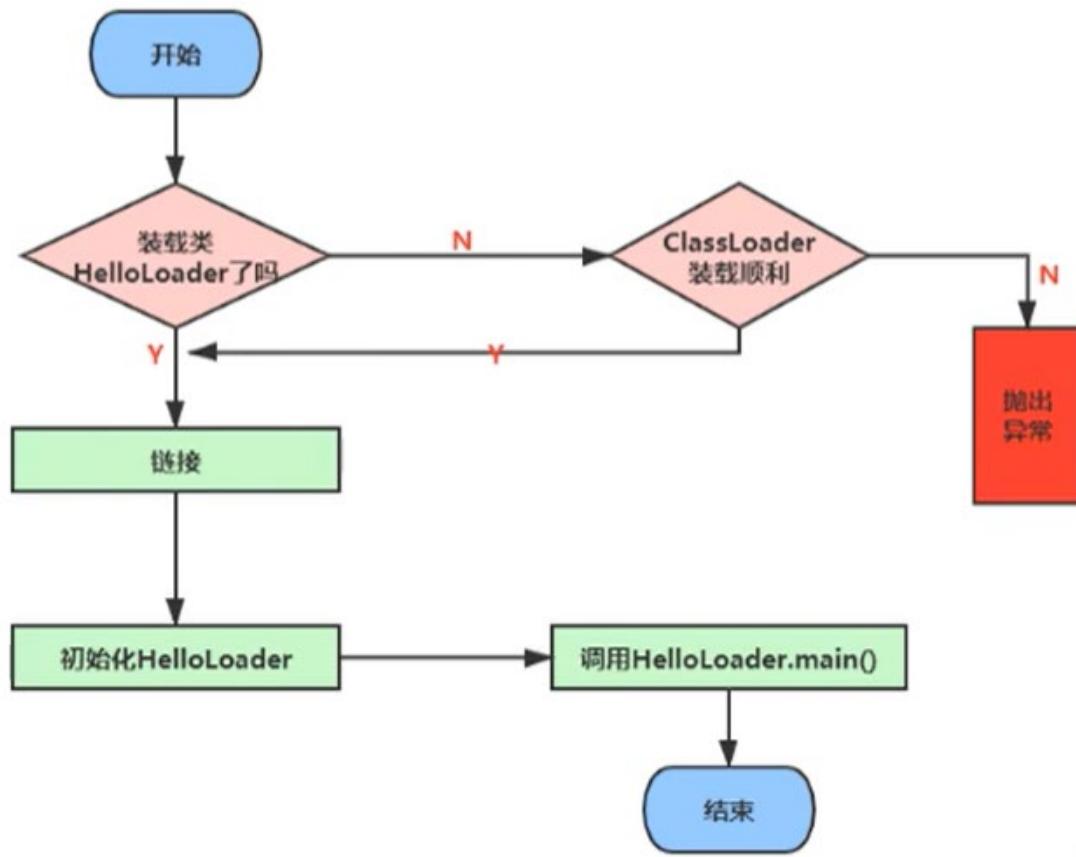


类的加载过程

例如下面的一段简单的代码

```
/*
 * 类加载子系统
 * @author: 陌溪
 * @create: 2020-07-05-8:24
 */
public class HelloLoader {
    public static void main(String[] args) {
        System.out.println("我已经被加载啦");
    }
}
```

它的加载过程是怎么样的呢?



完整的流程图如下所示



加载阶段

通过一个类的全限定名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化为方法区的运行时数据结构

在内存中生成一个代表这个类的 `java.lang.Class` 对象，作为方法区这个类的各种数据的访问入口

加载 class 文件的方式

- 从本地系统中直接加载
- 通过网络获取，典型场景：Web Applet
- 从 zip 压缩包中读取，成为日后 jar、war 格式的基础
- 运行时计算生成，使用最多的是：动态代理技术
- 由其他文件生成，典型场景：JSP 应用从专有数据库中提取.class 文件，比较少见
- 从加密文件中获取，典型的防 Class 文件被反编译的保护措施

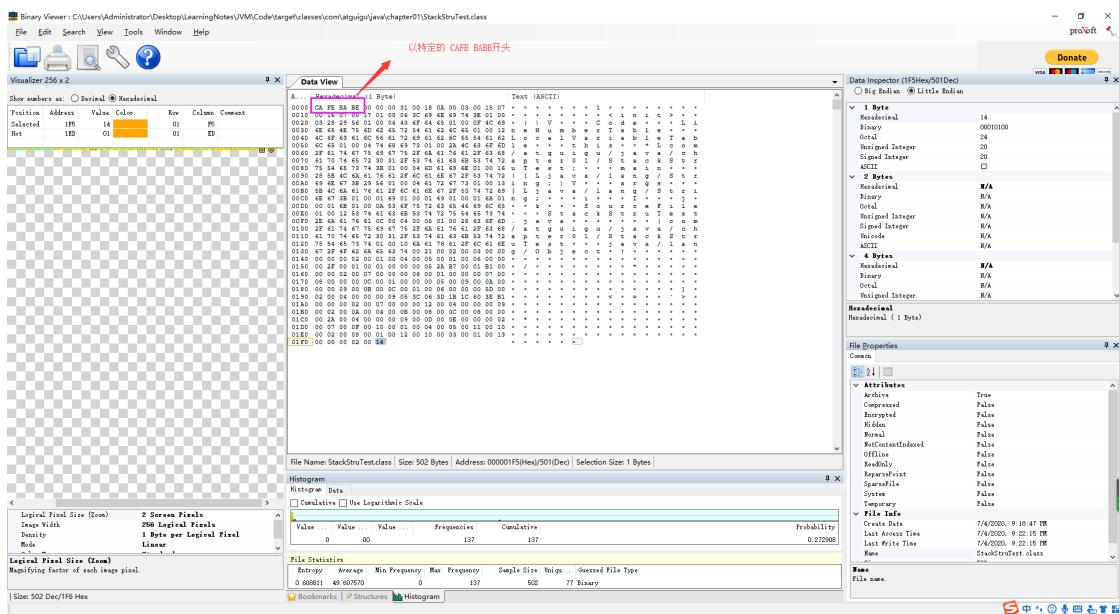
链接阶段

验证 Verify

目的在于确保 Class 文件的字节流中包含信息符合当前虚拟机要求，保证被加载类的正确性，不会危害虚拟机自身安全。

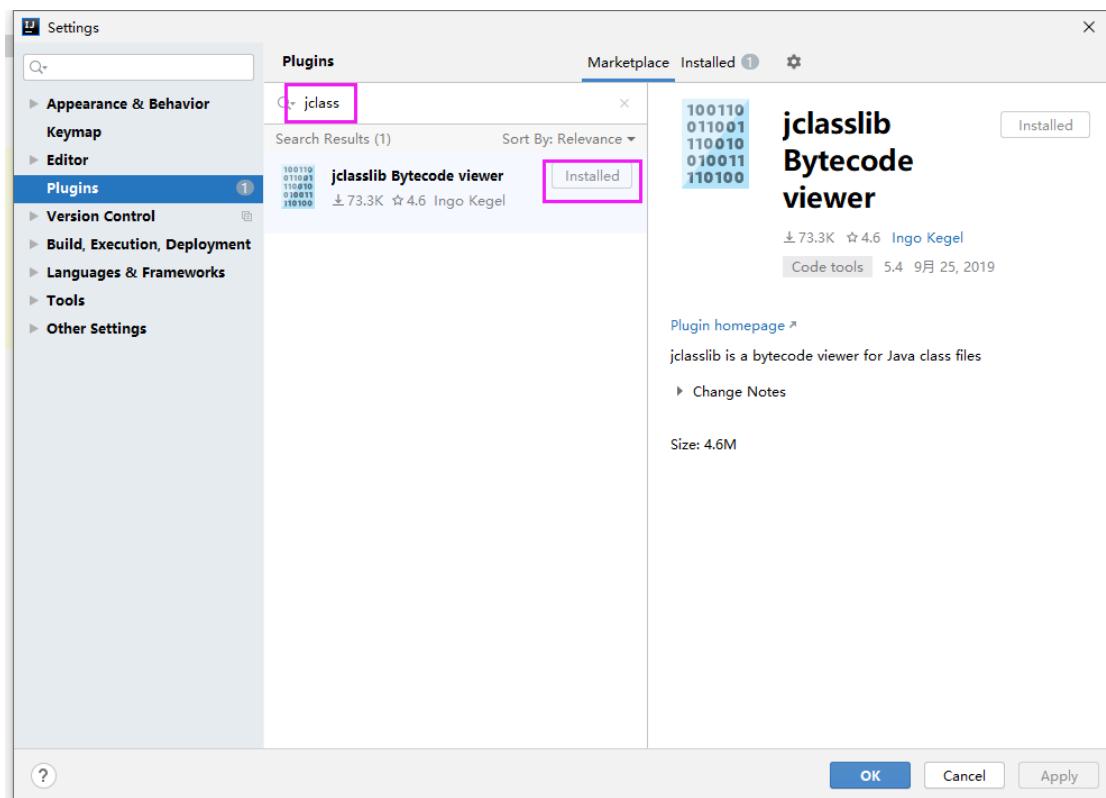
主要包括四种验证，文件格式验证，元数据验证，字节码验证，符号引用验证。

工具：Binary Viewer 查看

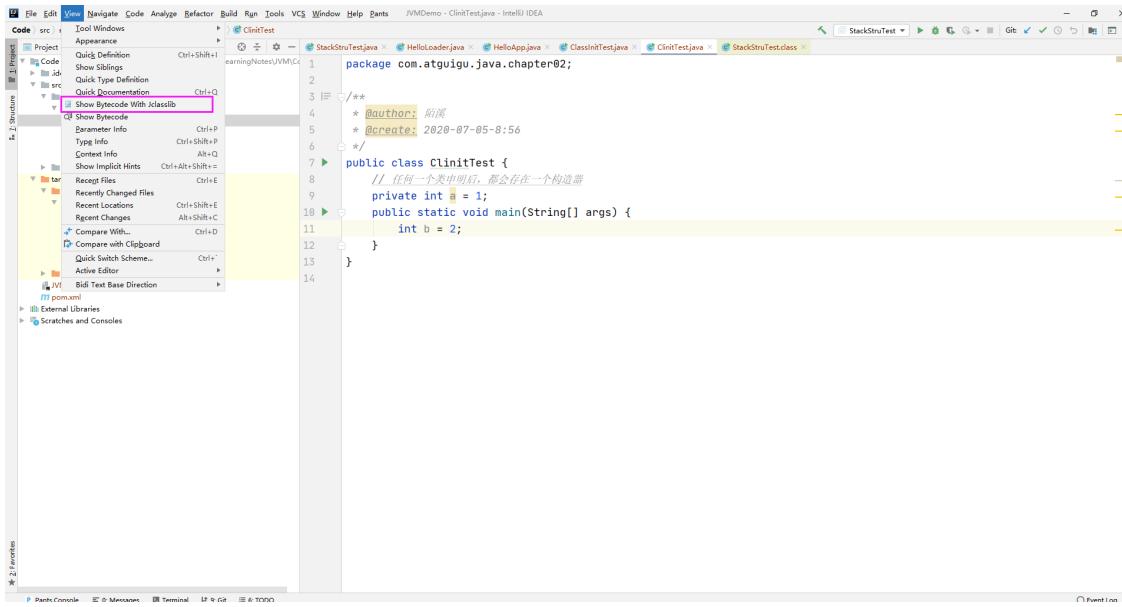


如果出现不合法的字节码文件，那么将会验证不通过

同时我们可以通过安装 IDEA 的插件，来查看我们的 Class 文件



安装完成后，我们编译完一个 class 文件后，点击 view 即可显示我们安装的插件来查看字节码方法了



准备 Prepare

为类变量分配内存并且设置该类变量的默认初始值，即零值。

```
/**
 * @author: 陌溪
 * @create: 2020-07-05-8:42
 */
public class HelloApp {
    private static int a = 1; // 准备阶段为0，在下个阶段，也就是初始化的时候才是1
    public static void main(String[] args) {
        System.out.println(a);
    }
}
```

上面的变量 a 在准备阶段会赋初始值，但不是 1，而是 0。

这里不包含用 final 修饰的 static，因为 final 在编译的时候就会分配了，准备阶段会显式初始化；

这里不会为实例变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到 Java 堆中。

例如下面这段代码

解析 Resolve

将常量池内的符号引用转换为直接引用的过程。

事实上，解析操作往往伴随着 JVM 在执行完初始化之后再执行。

符号引用就是一组符号来描述所引用的目标。符号引用的字面量形式明确定义在《java 虚拟机规范》的 class 文件格式中。直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型等。对应常量池中的 CONSTANT Class info、CONSTANT Fieldref info、CONSTANT Methodref info 等

初始化阶段	成员变量 类变量（带 static） 实例变量（不带 static） 局部变量	有默认的初始值 无默认的初始值
-------	--	--------------------

初始化阶段就是执行类构造器方法<clinit>()的过程。

此方法不需定义，是 javac 编译器自动收集类中的所有类变量的赋值动作和静态代码块中的语句合并而来。Java类变量：静态变量，必须以static修饰。

- 也就是说，当我们代码中包含 static 变量的时候，就会有<clinit>方法
构造器方法中指令按语句在源文件中出现的顺序执行。

<clinit>()不同于类的构造器。（关联：构造器是虚拟机视角下的<init>()）若该类具有父类，JVM 会保证子类的<clinit>()执行前，父类的<clinit>()已经执行完毕。

- 任何一个类在声明后，都有生成一个构造器，默认是空参构造器

```
/*
 * @author: 陌溪
 * @create: 2020-07-05-8:47
 */
public class ClassInitTest {
    private static int num = 1;
    static {
        num = 2;
        number = 20;
        System.out.println(num);
        System.out.println(number); // 报错，非法的前向引用
    }

    private static int number = 10;

    public static void main(String[] args) {
        System.out.println(ClassInitTest.num); // 2
        System.out.println(ClassInitTest.number); // 10
    }
}
```

关于涉及到父类时候的变量赋值过程

```

/**
 * @author: 陌溪
 * @create: 2020-07-05-9:06
 */
public class ClinitTest1 {
    static class Father {
        public static int A = 1;
        static {
            A = 2;
        }
    }

    static class Son extends Father {
        public static int b = A;
    }

    public static void main(String[] args) {
        System.out.println(Son.b);
    }
}

```

我们输出结果为 2，也就是说首先加载 ClinitTest1 的时候，会找到 main 方法，然后执行 Son 的初始化，但是 Son 继承了 Father，因此还需要执行 Father 的初始化，同时将 A 赋值为 2。我们通过反编译得到 Father 的加载过程，首先我们看到原来的值被赋值成 1，然后又被复制成 2，最后返回

```

iconst_1
putstatic #2 <com/atguigu/java/chapter02/ClinitTest1$Father.A>
iconst_2
putstatic #2 <com/atguigu/java/chapter02/ClinitTest1$Father.A>
return

```

虚拟机必须保证一个类的 () 方法在多线程下被同步加锁。

```

/**
 * @author: 陌溪
 * @create: 2020-07-05-9:14
 */
public class DeadThreadTest {
    public static void main(String[] args) {
        new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + "\t"
线程 t1 开始);
            new DeadThread();
        }, "t1").start();

        new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + "\t"
线程 t2 开始);
    }
}

```

```

        new DeadThread();
    }, "t2").start();
}
}
class DeadThread {
    static {
        if (true) {
            System.out.println(Thread.currentThread().getName() + "\t
初始化当前类");
            while(true) {

            }
        }
    }
}

```

上面的代码，输出结果为

```

线程 t1 开始
线程 t2 开始
线程 t2 初始化当前类

```

从上面可以看出初始化后，只能够执行一次初始化，这也就是同步加锁的过程

类加载器的分类

JVM 支持两种类型的类加载器。分别为引导类加载器（Bootstrap ClassLoader）和自定义类加载器（User-Defined ClassLoader）。

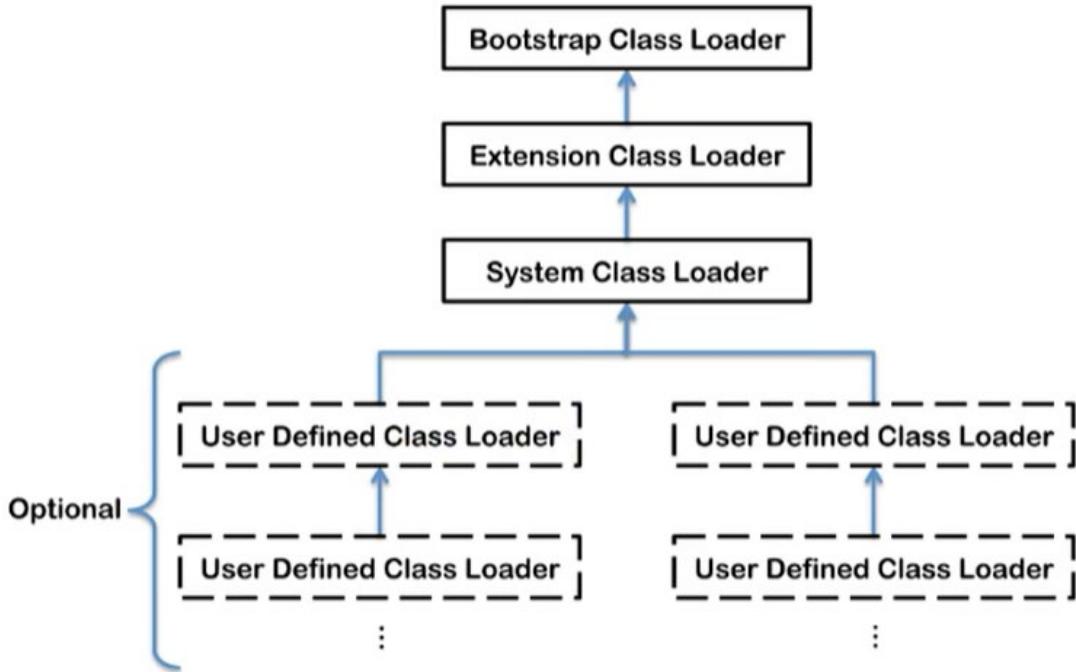
从概念上来讲，自定义类加载器一般指的是程序中由开发人员自定义的一类类加载器，但是 Java 虚拟机规范却没有这么定义，而是将所有派生于抽象类 ClassLoader 的类加载器都划分为自定义类加载器。

无论类加载器的类型如何划分，在程序中我们最常见的类加载器始终只有 3 个，如下所示：

引导类加载器	Bootstrap ClassLoader	JAVAHOME/jre/lib/rt.jar、resources.jar、sun.boot.class.path
--------	-----------------------	---

自定义加载器	扩展类加载器	jre/lib/ext
--------	--------	-------------

	系统类加载器	
--	--------	--



这里的四者之间是包含关系，不是上层和下层，也不是子系统的继承关系。

我们通过一个类，获取它不同的加载器

```

/**
 * @author: 陌溪
 * @create: 2020-07-05-9:47
 */
public class ClassLoaderTest {
    public static void main(String[] args) {
        // 获取系统类加载器
        ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
        System.out.println(systemClassLoader);

        // 获取其上层的：扩展类加载器
        ClassLoader extClassLoader = systemClassLoader.getParent();
        System.out.println(extClassLoader);

        // 试图获取 根加载器
        ClassLoader bootstrapClassLoader = extClassLoader.getParent();
        System.out.println(bootstrapClassLoader);

        // 获得自定义加载器
        ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();
    }
}

```

```

        System.out.println(classLoader);

        // 获取 String 类型的加载器
        ClassLoader classLoader1 = String.class.getClassLoader();
        System.out.println(classLoader1);
    }
}

```

得到的结果，从结果可以看出根加载器无法直接通过代码获取，同时目前用户代码所使用的加载器为系统类加载器。同时我们通过获取 String 类型的加载器，发现是 null，那么说明 String 类型是通过根加载器进行加载的，也就是说 Java 的核心类库都是使用根加载器进行加载的。

```

sun.misc.Launcher$AppClassLoader@18b4aac2
sun.misc.Launcher$ExtClassLoader@1540e19d
null
sun.misc.Launcher$AppClassLoader@18b4aac2
null

```

虚拟机自带的加载器

启动类加载器（引导类加载器，Bootstrap ClassLoader）

- 这个类加载使用 C/C++语言实现的，嵌套在 JVM 内部。
- 它用来加载 Java 的核心库（JAVAHOME/jre/1ib/rt.jar、resources.jar 或 sun.boot.class.path 路径下的内容），用于提供 JVM 自身需要的类
- 并不继承自 ava.lang.ClassLoader，没有父加载器。
- 加载扩展类和应用程序类加载器，并指定为他们的父类加载器。
- 出于安全考虑，Bootstrap 启动类加载器只加载包名为 java、javax、sun 等开头的类

扩展类加载器（Extension ClassLoader）

- Java 语言编写，由 sun.misc.Launcher\$ExtClassLoader 实现。
- 派生于 ClassLoader 类
- 父类加载器为启动类加载器
- 从 java.ext.dirs 系统属性所指定的目录中加载类库，或从 JDK 的安装目录的 jre/1ib/ext 子目录（扩展目录）下加载类库。如果用户创建的 JAR 放在此目录下，也会自动由扩展类加载器加载。

应用程序类加载器（系统类加载器，AppClassLoader）

- javI 语言编写，由 sun.misc.LaunchersAppClassLoader 实现

- 派生于 ClassLoader 类
- 父类加载器为扩展类加载器
- 它负责加载环境变量 classpath 或系统属性 java.class.path 指定路径下的类库
- 该类加载是程序中默认的类加载器，一般来说，Java 应用的类都是由它来完成加载
- 通过 classLoader#getSystemClassLoader() 方法可以获取到该类加载器

用户自定义类加载器

在 Java 的日常应用程序开发中，类的加载几乎是由上述 3 种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，来定制类的加载方式。为什么要自定义类加载器？

- 隔离加载类：避免类的冲突
- 修改类加载的方式：需要的时候动态加载
- 扩展加载源：从别处加载
- 防止源码泄漏：解决容易被篡改的问题

用户自定义类加载器实现步骤：

- 开发人员可以通过继承抽象类 `java.lang.ClassLoader` 类的方式，实现自己的类加载器，以满足一些特殊的需求
- 在 JDK1.2 之前，在自定义类加载器时，总会去继承 `ClassLoader` 类并重写 `loadClass()` 方法，从而实现自定义的类加载类，但是在 JDK1.2 之后已不再建议用户去覆盖 `loadclass()` 方法，而是建议把自定义的类加载逻辑写在 `findclass()` 方法中
- 在编写自定义类加载器时，如果没有太过于复杂的需求，可以直接继承 `URIClassLoader` 类，这样就可以避免自己去编写 `findclass()` 方法及其获取字节码流的方式，使自定义类加载器编写更加简洁。

查看根加载器所能加载的目录

刚刚我们通过概念了解到了，根加载器只能够加载 `java /lib` 目录下的 `class`，我们通过下面代码验证一下

```
/**  
 * @author: 陌溪  
 * @create: 2020-07-05-10:17
```

```

/*
public class ClassLoaderTest1 {
    public static void main(String[] args) {
        System.out.println("*****启动类加载器*****");
        // 获取BootstrapClassLoader 能够加载的API 的路径
        URL[] urls = sun.misc.Launcher.getBootstrapClassPath().getURLs
();
        for (URL url : urls) {
            System.out.println(url.toExternalForm());
        }
        // 从上面路径中，随意选择一个类，来看看他的类加载器是什么：得到的是null，说明是 根加载器
        ClassLoader classLoader = Provider.class.getClassLoader();
    }
}

```

得到的结果

```

*****启动类加载器*****
file:/E:/Software/JDK1.8/Java/jre/lib/resources.jar
file:/E:/Software/JDK1.8/Java/jre/lib/rt.jar
file:/E:/Software/JDK1.8/Java/jre/lib/sunrsasign.jar
file:/E:/Software/JDK1.8/Java/jre/lib/jsse.jar
file:/E:/Software/JDK1.8/Java/jre/lib/jce.jar
file:/E:/Software/JDK1.8/Java/jre/lib/charsets.jar
file:/E:/Software/JDK1.8/Java/jre/lib/jfr.jar
file:/E:/Software/JDK1.8/Java/jre/classes
null

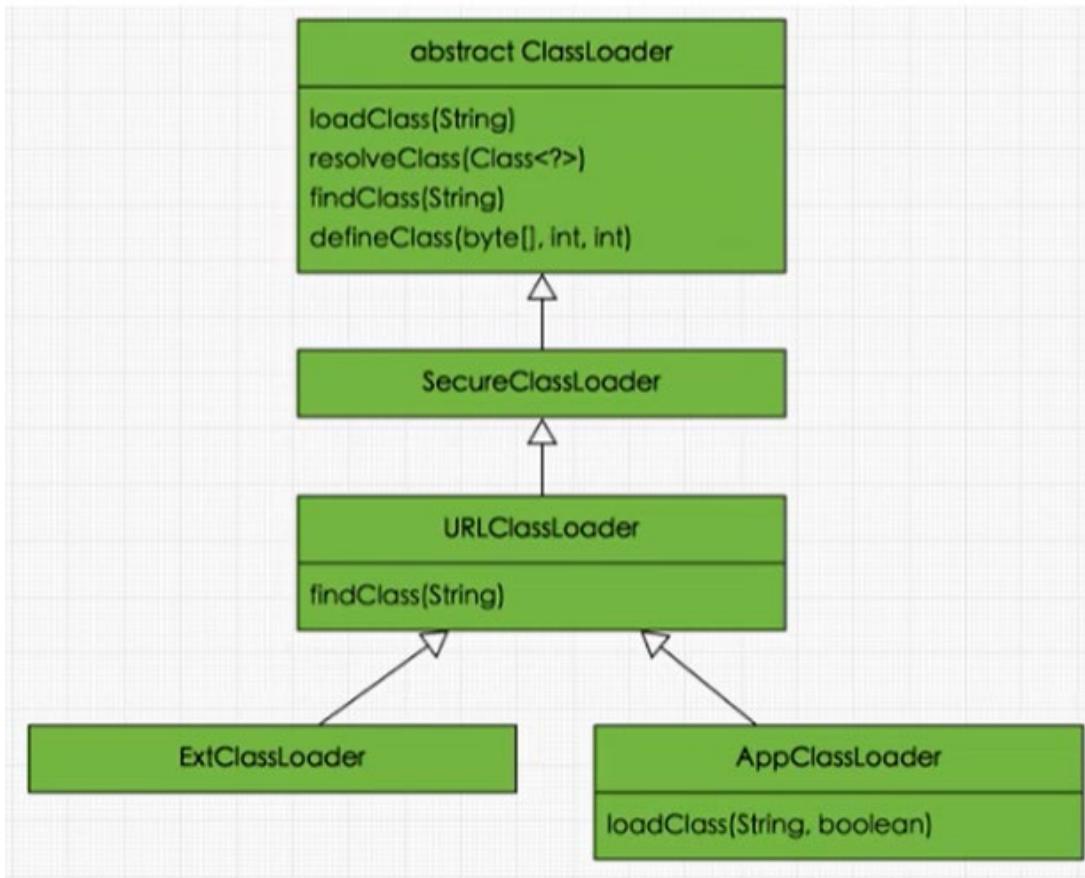
```

关于 ClassLoader

ClassLoader 类，它是一个抽象类，其后所有的类加载器都继承自 ClassLoader（不包括启动类加载器）

方法名称	描述
getParent()	返回该类加载器的超类加载器
loadClass(String name)	加载名称为 name 的类，返回结果为 java.lang.Class 类的实例
findClass(String name)	查找名称为 name 的类，返回结果为 java.lang.Class 类的实例
findLoadedClass(String name)	查找名称为 name 的已经被加载过的类，返回结果为 java.lang.Class 类的实例
defineClass(String name,byte[] b,int off,int len)	把字节数组 b 中的内容转换为一个 Java 类，返回结果为 java.lang.Class 类的实例
resolveClass(Class<?> c)	连接指定的一个 Java 类

sun.misc.Launcher 它是一个 java 虚拟机的入口应用



获取 ClassLoader 的途径

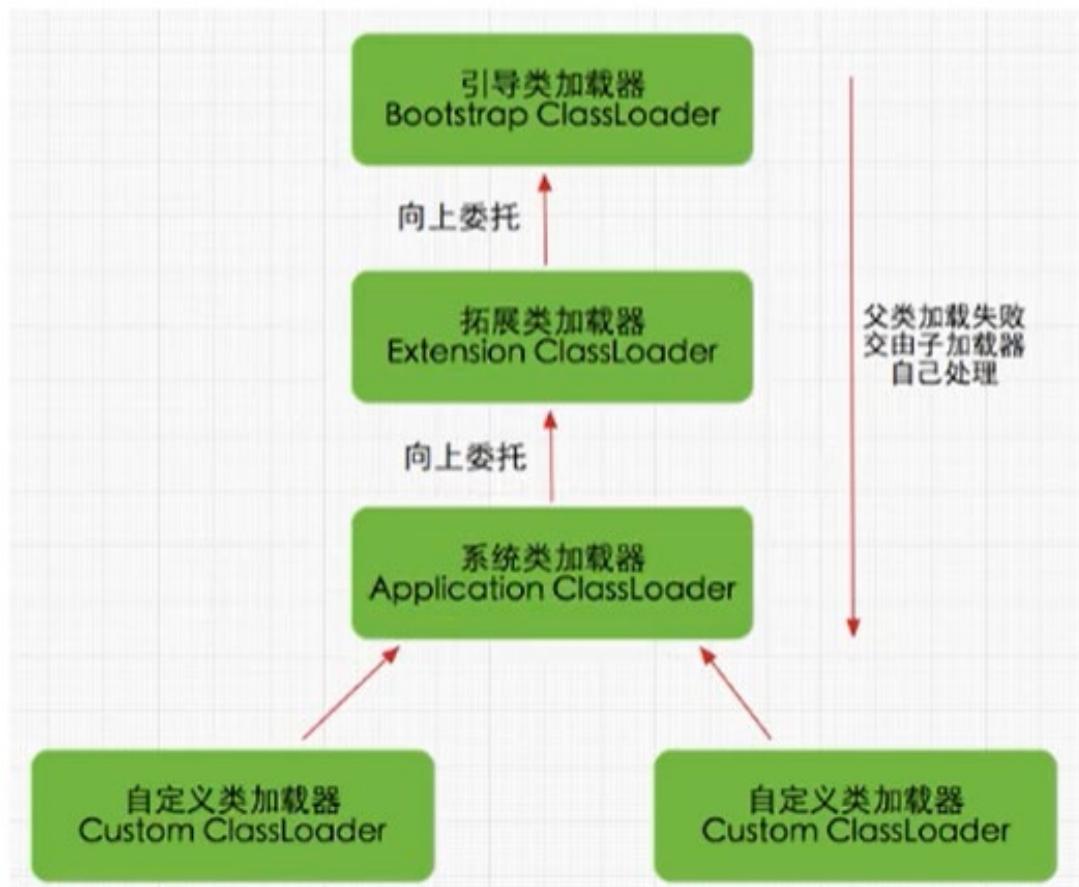
- 获取当前 ClassLoader: clazz.getClassLoader()
- 获取当前线程上下文的 ClassLoader:
Thread.currentThread().getContextClassLoader()
- 获取系统的 ClassLoader: ClassLoader.getSystemClassLoader()
- 获取调用者的 ClassLoader: DriverManager.getCallerClassLoader()

双亲委派机制

Java 虚拟机对 class 文件采用的是按需加载的方式，也就是说当需要使用该类时才会将它的 class 文件加载到内存生成 class 对象。而且加载某个类的 class 文件时，Java 虚拟机采用的是双亲委派模式，即把请求交由父类处理，它是一种任务委派模式。

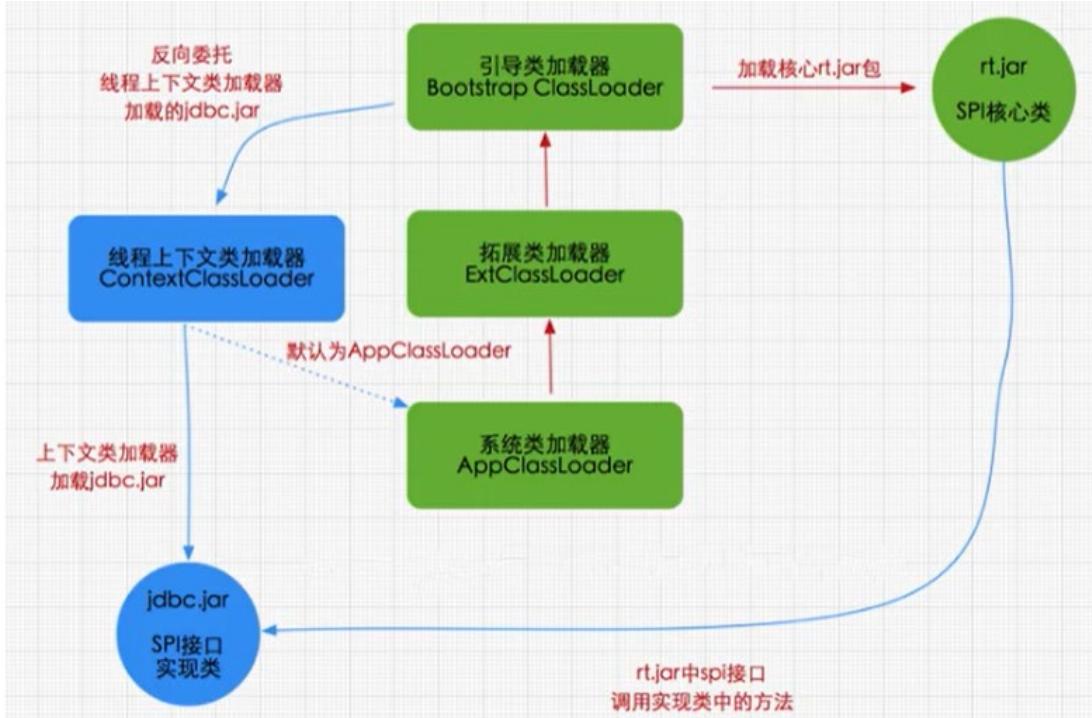
工作原理

- 如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行；
- 如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器；
- 如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式。



双亲委派机制举例

当我们加载 `jdbc.jar` 用于实现数据库连接的时候，首先我们需要知道的是 `jdbc.jar` 是基于 SPI 接口进行实现的，所以在加载的时候，会进行双亲委派，最终从根加载器中加载 SPI 核心类，然后在加载 SPI 接口类，接着在进行反向委派，通过线程上下文类加载器进行实现类 `jdbc.jar` 的加载。



沙箱安全机制

自定义 string 类，但是在加载自定义 String 类的时候会率先使用引导类加载器加载，而引导类加载器在加载的过程中会先加载 jdk 自带的文件（rt.jar 包中 java\lang\String.class），报错信息说没有 main 方法，就是因为加载的是 rt.jar 包中的 string 类。这样可以保证对 java 核心源代码的保护，这就是沙箱安全机制。

双亲委派机制的优势

通过上面的例子，我们可以知道，双亲机制可以

- 避免类的重复加载
- 保护程序安全，防止核心 API 被随意篡改
 - 自定义类：java.lang.String
 - 自定义类：java.lang.ShkStart (报错：阻止创建 java.lang 开头的类)

其它

如何判断两个 class 对象是否相同

在 JVM 中表示两个 class 对象是否为同一个类存在两个必要条件：

- 类的完整类名必须一致，包括包名。
- 加载这个类的 ClassLoader（指 ClassLoader 实例对象）必须相同。

换句话说，在 JVM 中，即使这两个类对象（class 对象）来源同一个 Class 文件，被同一个虚拟机所加载，但只要加载它们的 ClassLoader 实例对象不同，那么这两个类对象也是不相等的。

JVM 必须知道一个类型是由启动加载器加载的还是由用户类加载器加载的。如果一个类型是由用户类加载器加载的，那么 JVM 会将这个类加载器的一个引用作为类型信息的一部分保存在方法区中。当解析一个类型到另一个类型的引用的时候，JVM 需要保证这两个类型的类加载器是相同的。

类的主动使用和被动使用

Java 程序对类的使用方式分为：主动使用和被动使用。主动使用，又分为七种情况：

- 创建类的实例
- 访问某个类或接口的静态变量，或者对该静态变量赋值
- 调用类的静态方法 I
- 反射（比如：Class.forName ("com.atguigu.Test") ）
- 初始化一个类的子类
- Java 虚拟机启动时被标明为启动类的类
- JDK7 开始提供的动态语言支持：
- java.lang.invoke.MethodHandle 实例的解析结果 REF getStatic、REF putStatic、REF invokeStatic 句柄对应的类没有初始化，则初始化

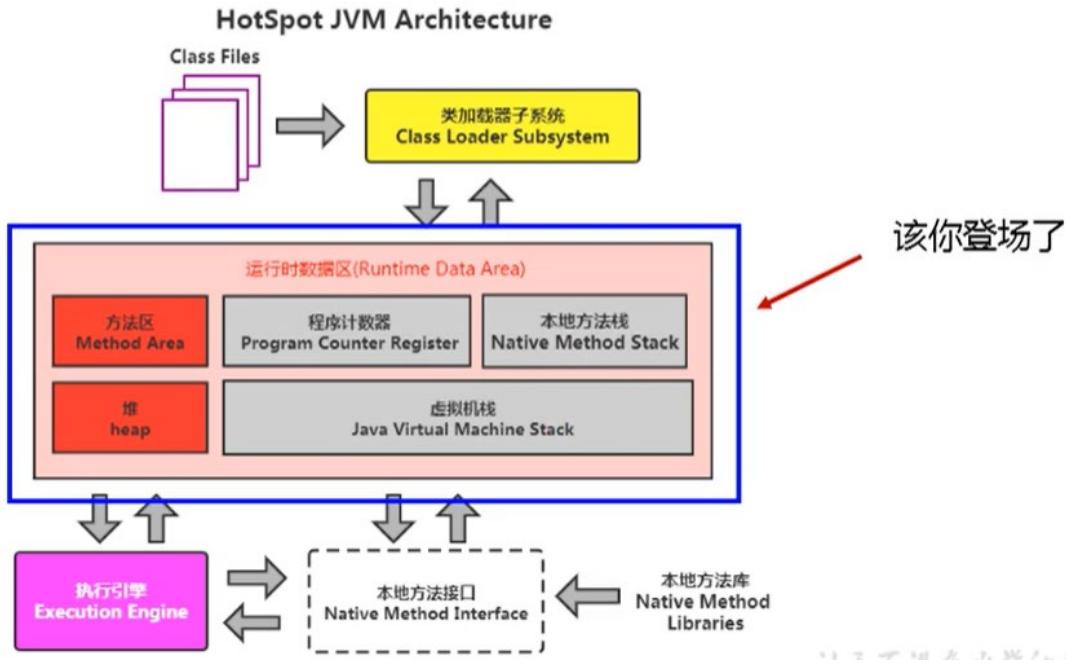
除了以上七种情况，其他使用 Java 类的方式都被看作是对类的被动使用，都不会导致类的初始化。

被动使用不会导致类的初始化，而主动使用会进行类的初始化。

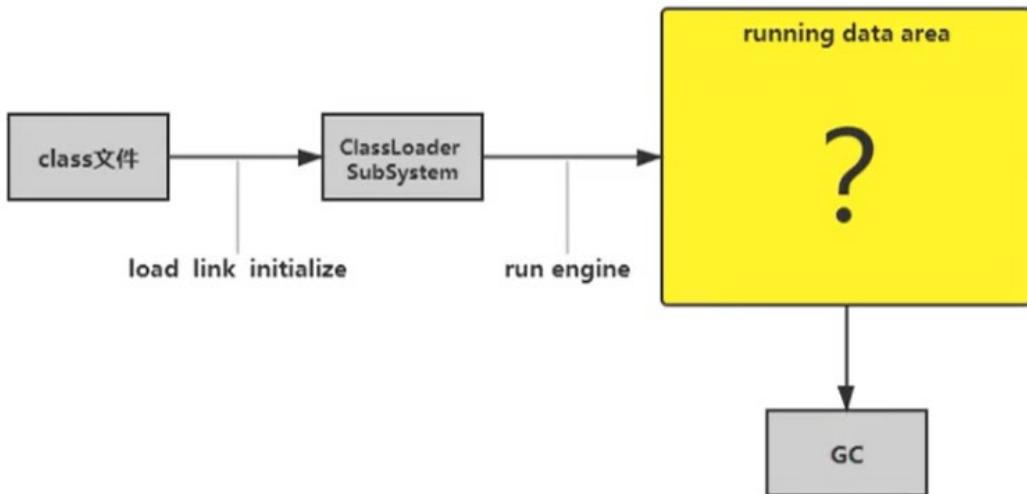
运行时数据区概述及线程

前言

本节主要讲的是运行时数据区，也就是下图这部分，它是在类加载完成后的阶段



当我们通过前面的：类的加载-> 验证 -> 准备 -> 解析 -> 初始化 这几个阶段完成后，就会用到执行引擎对我们的类进行使用，同时执行引擎将会使用到我们运行时数据区



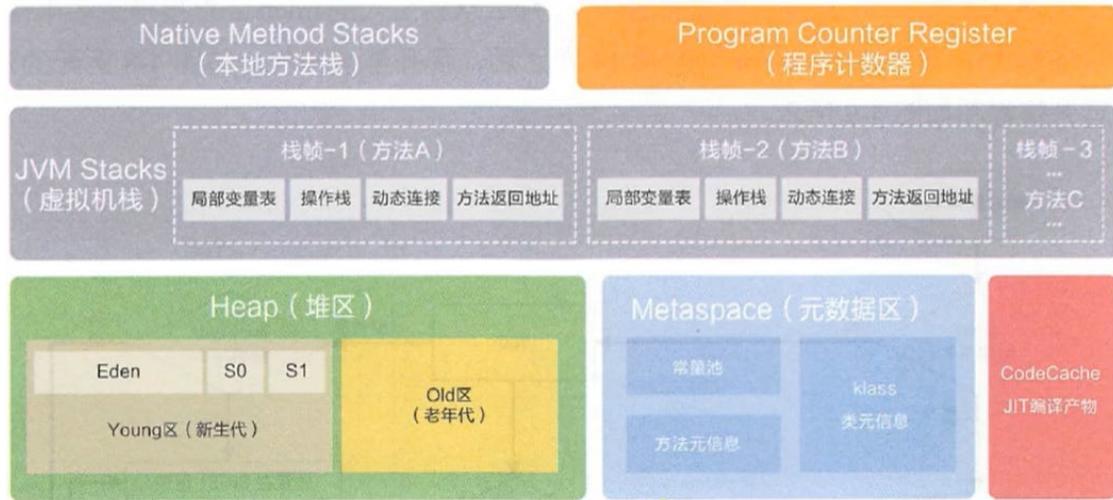
也就是大厨做饭，我们把大厨后面的东西（切好的菜，刀，调料），比作是运行时数据区。而厨师可以类比于执行引擎，将通过准备的东西进行制作成精美的菜品



内存是非常重要的系统资源，是硬盘和 CPU 的中间仓库及桥梁，承载着操作系统和应用程序的实时运行。JVM 内存布局规定了 Java 在运行过程中内存申请、分配、管理的策略，保证了 JVM 的高效稳定运行。不同的 JVM 对于内存的划分方式和管理机制存在着部分差异。结合 JVM 虚拟机规范，来探讨一下经典的 JVM 内存布局。

我们通过磁盘或者网络 IO 得到的数据，都需要先加载到内存中，然后 CPU 从内存中获取数据进行读取，也就是说内存充当了 CPU 和磁盘之间的桥梁。

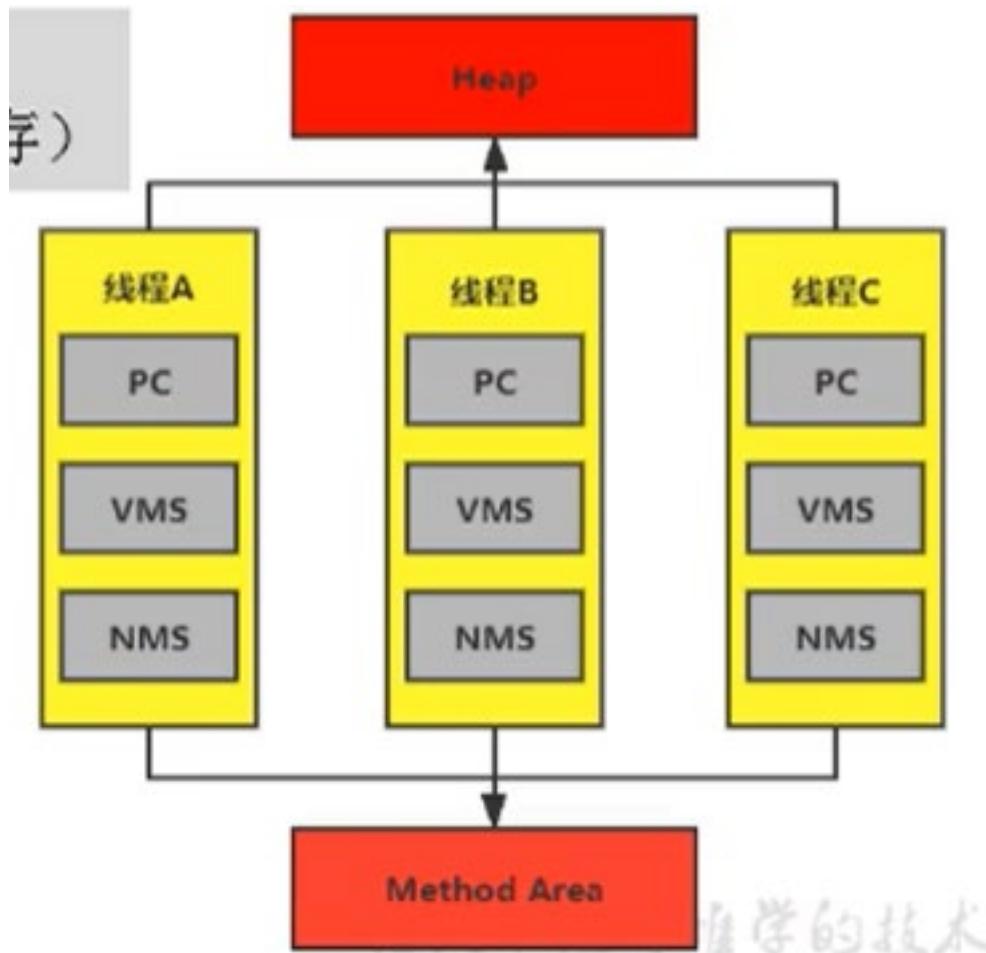
运行时数据区的完整图



Java 虚拟机定义了若干种程序运行期间会使用到的运行时数据区，其中有一些会随着虚拟机启动而创建，随着虚拟机退出而销毁。另外一些则是与线程一一对应的，这些与线程对应的数据区域会随着线程开始和结束而创建和销毁。

灰色的为单独线程私有的，红色的为多个线程共享的。即：

- 每个线程：独立包括程序计数器、栈、本地栈。
- 线程间共享：堆、堆外内存（永久代或元空间、代码缓存）



线程

线程是一个程序里的运行单元。JVM 允许一个应用有多个线程并行的执行。在 Hotspot JVM 里，每个线程都与操作系统的本地线程直接映射。

- 当一个 Java 线程准备好执行以后，此时一个操作系统的本地线程也同时创建。Java 线程执行终止后，本地线程也会回收。

操作系统负责所有线程的安排调度到任何一个可用的 CPU 上。一旦本地线程初始化成功，它就会调用 Java 线程中的 run () 方法。

JVM 系统线程

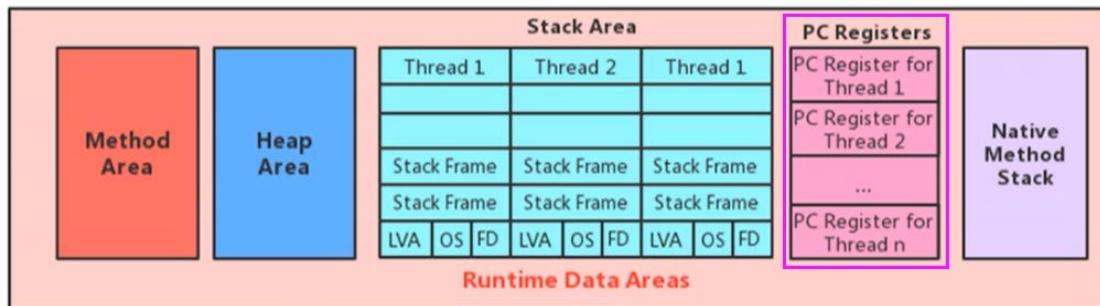
如果你使用 console 或者是任何一个调试工具，都能看到在后台有许多线程在运行。这些后台线程不包括调用 public static void main (String[]) 的 main 线程以及所有这个 main 线程自己创建的线程。| 这些主要的后台系统线程在 Hotspot JVM 里主要是以下几个：

- 虚拟机线程：这种线程的操作是需要 JVM 达到安全点才会出现。这些操作必须在不同的线程中发生的原因是他们都需要 JVM 达到安全点，这样堆才不会变化。这种线程的执行类型包括"stop-the-world"的垃圾收集，线程栈收集，线程挂起以及偏向锁撤销。
- 周期任务线程：这种线程是时间周期事件的体现（比如中断），他们一般用于周期性操作的调度执行。
- GC 线程：这种线程对在 JVM 里不同种类的垃圾收集行为提供了支持。
- 编译线程：这种线程在运行时会将字节码编译成本地代码。
- 信号调度线程：这种线程接收信号并发送给 JVM，在它内部通过调用适当的方法进行处理。

程序计数器

介绍

JVM 中的程序计数器（Program Counter Register）中，Register 的命名源于 CPU 的寄存器，寄存器存储指令相关的现场信息。CPU 只有把数据装载到寄存器才能够运行。这里，并非是广义上所指的物理寄存器，或许将其翻译为 PC 计数器（或指令计数器）会更加贴切（也称为程序钩子），并且也不容易引起一些不必要的误会。JVM 中的 PC 寄存器是对物理 PC 寄存器的一种抽象模拟。



它是一块很小的内存空间，几乎可以忽略不记。也是运行速度最快的存储区域。

在 JVM 规范中，每个线程都有它自己的程序计数器，是线程私有的，生命周期与线程的生命周期保持一致。

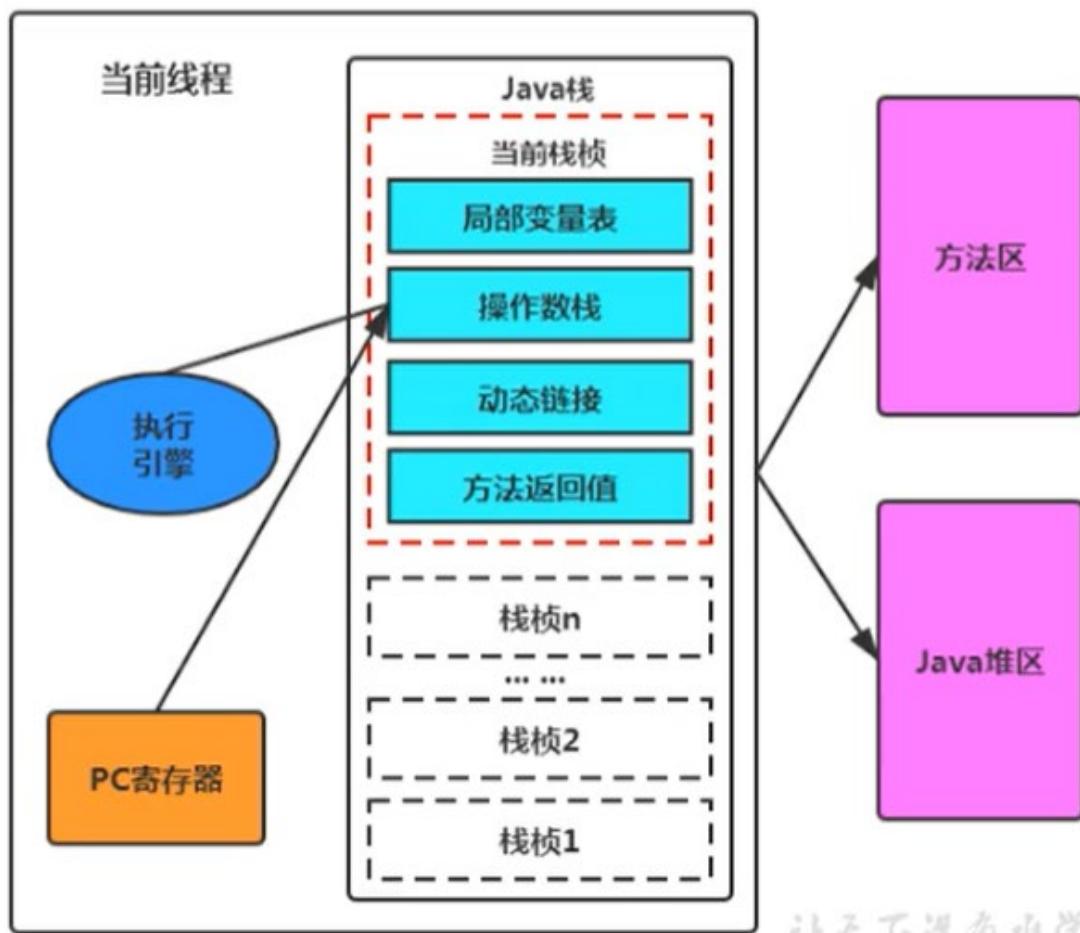
任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器会存储当前线程正在执行的 Java 方法的 JVM 指令地址；或者，如果是在执行 native 方法，则是未指定值（undefined）。

它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。字节码解释器工作时就是通过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。PC不存在OOM和GC的问题。

它是唯一一个在 Java 虚拟机规范中没有规定任何 `outofMemoryError` 情况的区域。

作用

PC 寄存器用来存储指向下一条指令的地址，也即将要执行的指令代码。由执行引擎读取下一条指令。



让天下没有难学的

代码演示

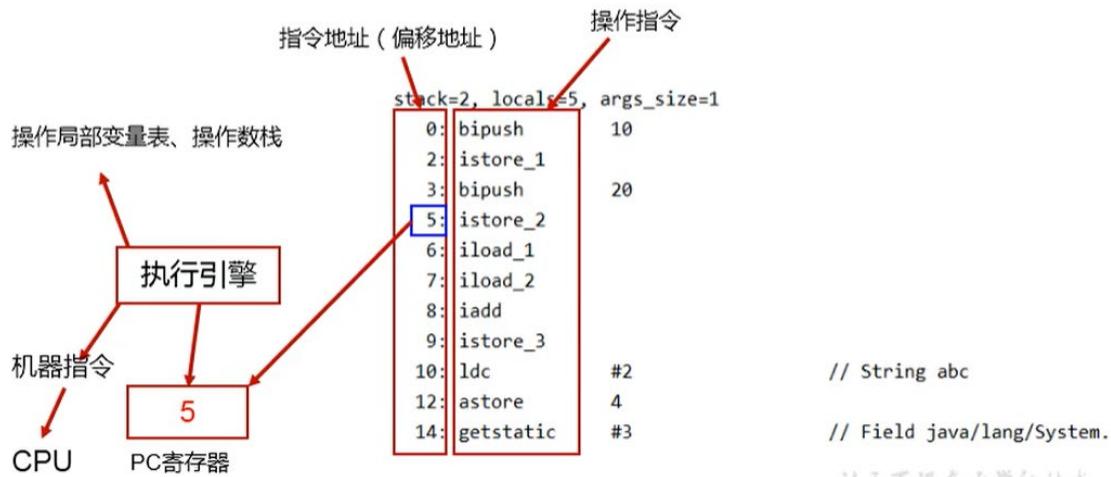
我们首先写一个简单的代码

```
/**程序计数器
 * @author: 陌溪
 * @create: 2020-07-05-16:01
 */
public class PCRegisterTest {
    public static void main(String[] args) {
        int i = 10;
        int j = 20;
        int k = i + j;
    }
}
```

然后将代码进行编译成字节码文件，我们再次查看，发现在字节码的左边有一个行号标识，它其实就是要指令地址，用于指向当前执行到哪里。

```
0: bipush      10
2: istore_1
3: bipush      20
5: istore_2
6: iload_1
7: iload_2
8: iadd
9: istore_3
10: return
```

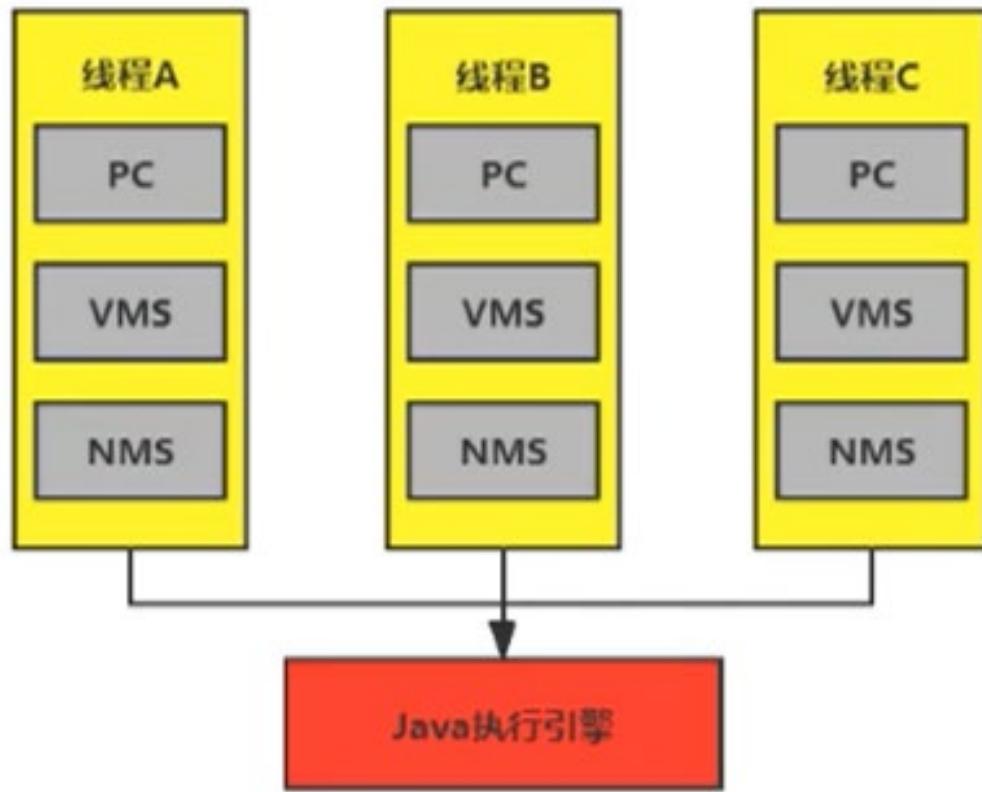
通过 PC 寄存器，我们就可以知道当前程序执行到哪一步了



使用 PC 寄存器存储字节码指令地址有什么用呢？

因为 CPU 需要不停的切换各个线程，这时候切换回来以后，就得知道接着从哪开始继续执行。

JVM 的字节码解释器就需要通过改变 PC 寄存器的值来明确下一条应该执行什么样的字节码指令。

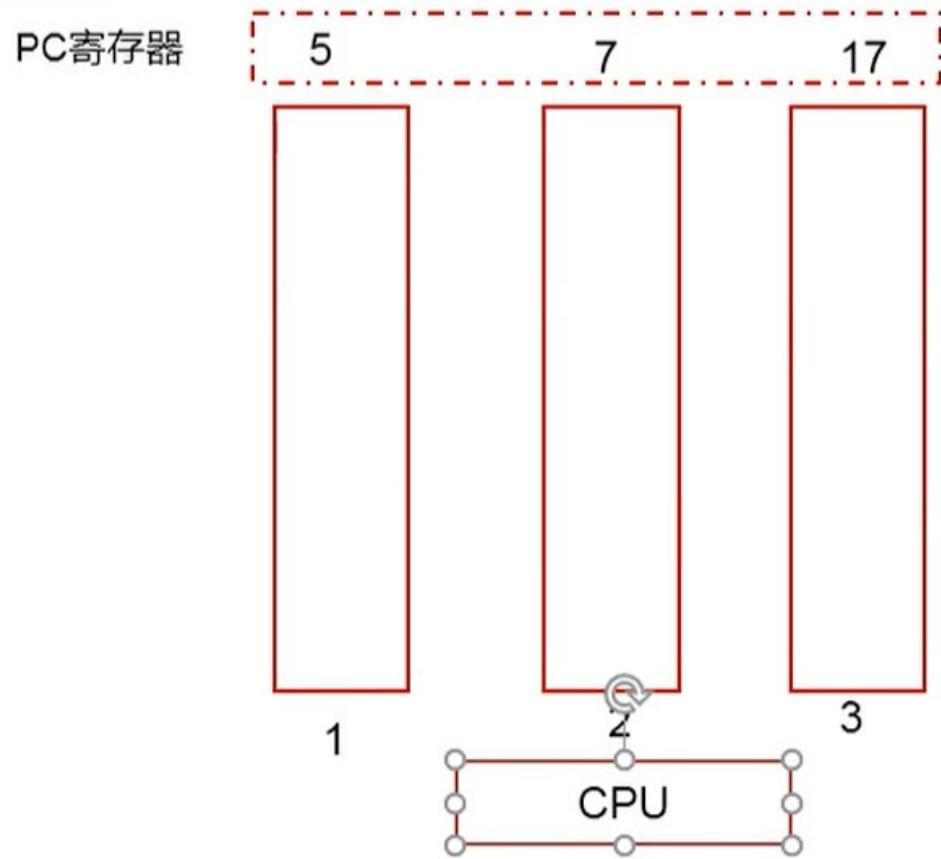


PC 寄存器为什么被设定为私有的？

我们都知道所谓的多线程在一个特定的时间段内只会执行其中某一个线程的方法，CPU 会不停地做任务切换，这样必然导致经常中断或恢复，如何保证分毫无差呢？为了能够准确地记录各个线程正在执行的当前字节码指令地址，最好的办法自然是为每一个线程都分配一个 PC 寄存器，这样一来各个线程之间便可以进行独立计算，从而不会出现相互干扰的情况。

由于 CPU 时间片轮限制，众多线程在并发执行过程中，任何一个确定的时刻，一个处理器或者多核处理器中的一个内核，只会执行某个线程中的一条指令。

这样必然导致经常中断或恢复，如何保证分毫无差呢？每个线程在创建后，都会产生自己的程序计数器和栈帧，程序计数器在各个线程之间互不影响。



CPU 时间片

CPU 时间片即 CPU 分配给各个程序的时间，每个线程被分配一个时间段，称作它的时间片。

在宏观上：我们可以同时打开多个应用程序，每个程序并行不悖，同时运行。

但在微观上：由于只有一个 CPU，一次只能处理程序要求的一部分，如何处理公平，一种方法就是引入时间片，每个程序轮流执行。



虚拟机栈

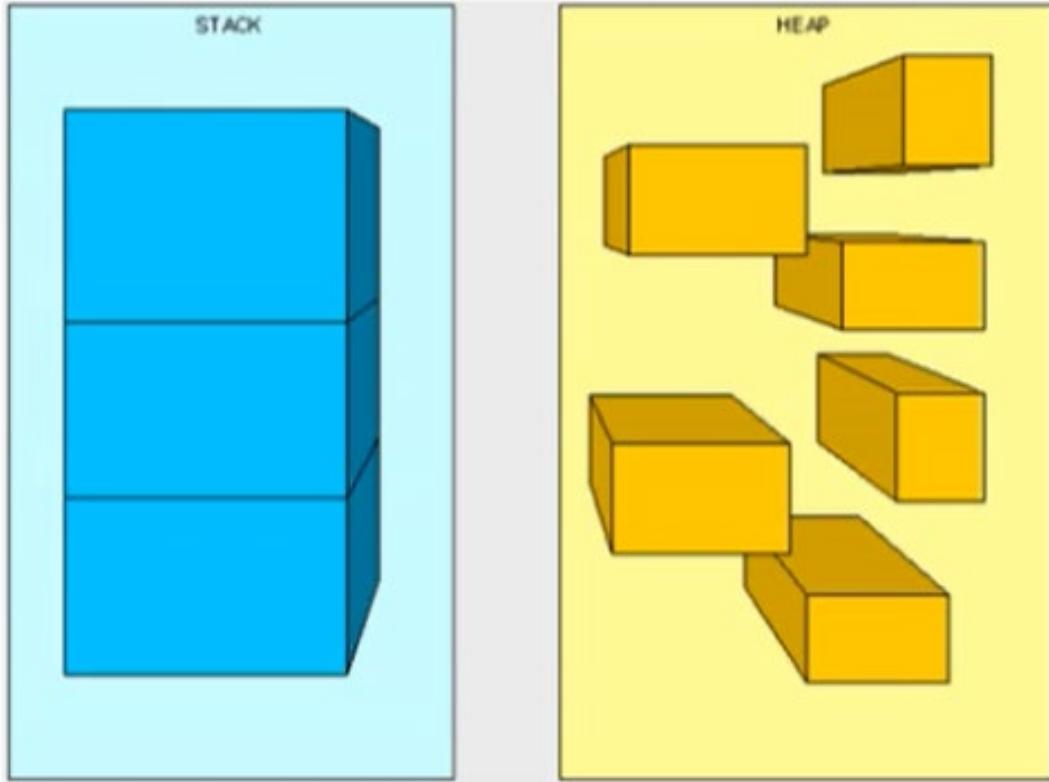
虚拟机栈概述

由于跨平台性的设计，Java 的指令都是根据栈来设计的。不同平台 CPU 架构不同，所以不能设计为基于寄存器的。优点是跨平台，指令集小，编译器容易实现，缺点是性能下降，实现同样的功能需要更多的指令。

有不少 Java 开发人员一提到 Java 内存结构，就会非常粗粒度地将 JVM 中的内存区理解为仅有 Java 堆（heap）和 Java 栈（stack）？为什么？

首先栈是运行时的单位，而堆是存储的单位

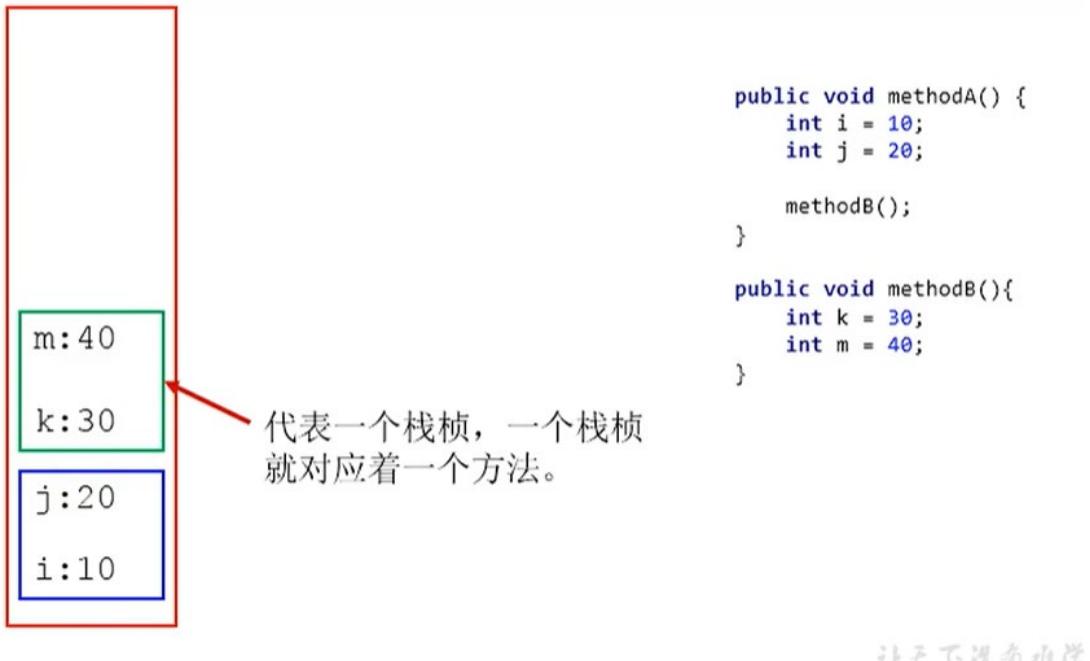
- 栈解决程序的运行问题，即程序如何执行，或者说如何处理数据。
- 堆解决的是数据存储的问题，即数据怎么放，放哪里



Java 虚拟机栈是什么

Java 虚拟机栈（Java Virtual Machine Stack），早期也叫 Java 栈。每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（Stack Frame），对应着一次次的 Java 方法调用。

是线程私有的



补充 不是名函数

生命周期

生命周期和线程一致，也就是线程结束了，该虚拟机栈也销毁了

作用

主管 Java 程序的运行，它保存方法的局部变量、部分结果，并参与方法的调用和返回。

局部变量，它是相比于成员变量来说的（或属性）	成员变量	类变量（带static）	实例变量（不带static）	有默认的初始值
基本数据类型变量 vs 引用类型变量（类、数组、接口）	局部变量			无默认的初始值

栈的特点

栈是一种快速有效的分配存储方式，访问速度仅次于程序计数器。JVM 直接对 Java 栈的操作只有两个：

- 每个方法执行，伴随着进栈（入栈、压栈）
- 执行结束后的出栈工作

对于栈来说不存在垃圾回收问题（栈存在溢出的情况），**栈存在OOM不需要GC**



开发中遇到哪些异常？

栈中可能出现的异常

Java 虚拟机规范允许 Java 栈的大小是动态的或者是固定不变的。

如果采用固定大小的 Java 虚拟机栈，那每一个线程的 Java 虚拟机栈容量可以在线程创建的时候独立选定。如果线程请求分配的栈容量超过 Java 虚拟机栈允许的最大容量，Java 虚拟机将会抛出一个 StackoverflowError 异常。

如果 Java 虚拟机栈可以动态扩展，并且在尝试扩展的时候无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的虚拟机栈，那 Java 虚拟机将会抛出一个 `outofMemoryError` 异常。

```
/**  
 * 演示栈中的异常: StackOverflowError  
 * @author: 陌溪  
 * @create: 2020-07-05-17:11  
 */  
public class StackErrorTest {  
    private static int count = 1;  
    public static void main(String[] args) {  
        System.out.println(count++);  
        main(args);  
    }  
}
```

当栈深度达到 9803 的时候，就出现栈内存空间不足

设置栈内存大小

我们可以使用参数 `-Xss` 选项来设置线程的最大栈空间，栈的大小直接决定了函数调用的最大可达深度

```
-Xss1m  
-Xss1k
```

栈的存储单位

每个线程都有自己的栈，栈中的数据都是以栈帧（Stack Frame）的格式存在。

在这个线程上正在执行的每个方法都各自对应一个栈帧（Stack Frame）。

栈帧是一个内存区块，是一个数据集，维系着方法执行过程中的各种数据信息。

栈中存储什么？

每个线程都有自己的栈，栈中的数据都是以栈帧（Stack Frame）的格式存在。在这个线程上正在执行的每个方法都各自对应一个栈帧（Stack Frame）。栈帧是一个内存区块，是一个数据集，维系着方法执行过程中的各种数据信息。

OOP 的基本概念：类和对象

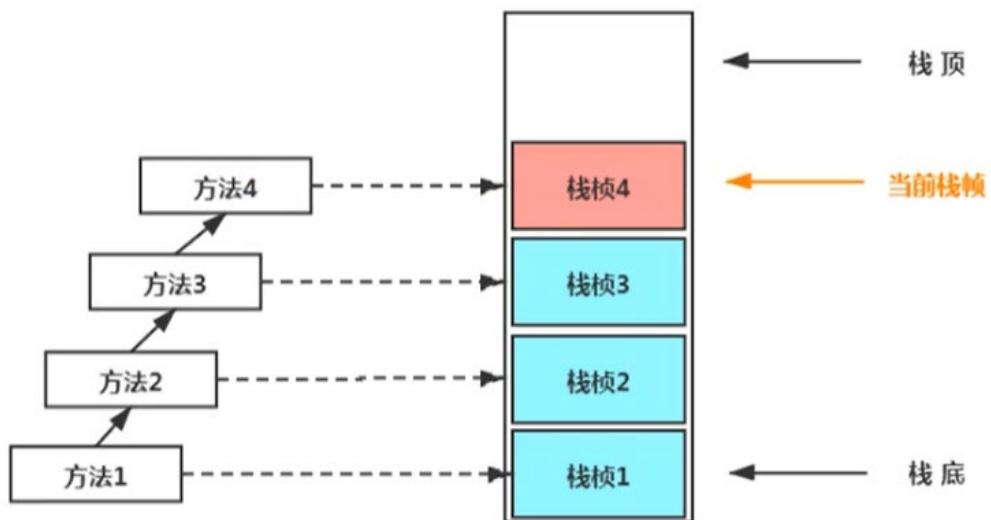
类中基本结构：field（属性、字段、域）、method

JVM 直接对 Java 栈的操作只有两个，就是对栈帧的压栈和出栈，遵循“先进后出”/“后进先出”原则。

在一条活动线程中，一个时间点上，只会有一个活动的栈帧。即只有当前正在执行的方法的栈帧（栈顶栈帧）是有效的，这个栈帧被称为当前栈帧（Current Frame），与当前栈帧相对应的方法就是当前方法（Current Method），定义这个方法的类就是当前类（Current Class）。

执行引擎运行的所有字节码指令只针对当前栈帧进行操作。

如果在该方法中调用了其他方法，对应的新的栈帧会被创建出来，放在栈的顶端，成为新的当前帧。



下面写一个简单的代码

```
/**  
 * 栈帧  
 *  
 * @author: 陌溪  
 * @create: 2020-07-05-20:33  
 */  
public class StackFrameTest {  
    public static void main(String[] args) {  
        method01();  
    }  
  
    private static int method01() {  
        System.out.println("方法 1 的开始");  
        int i = method02();  
        System.out.println("方法 1 的结束");  
        return i;  
    }  
}
```

```

private static int method02() {
    System.out.println("方法 2 的开始");
    int i = method03();
    System.out.println("方法 2 的结束");
    return i;
}
private static int method03() {
    System.out.println("方法 3 的开始");
    int i = 30;
    System.out.println("方法 3 的结束");
    return i;
}
}

```

输出结果为

方法 1 的开始
方法 2 的开始
方法 3 的开始
方法 3 的结束
方法 2 的结束
方法 1 的结束

满足栈先进后出的概念，通过 Idea 的 DEBUG，能够看到栈信息



栈运行原理

不同线程中所包含的栈帧是不允许存在相互引用的，即不可能在一个栈帧之中引用另外一个线程的栈帧。

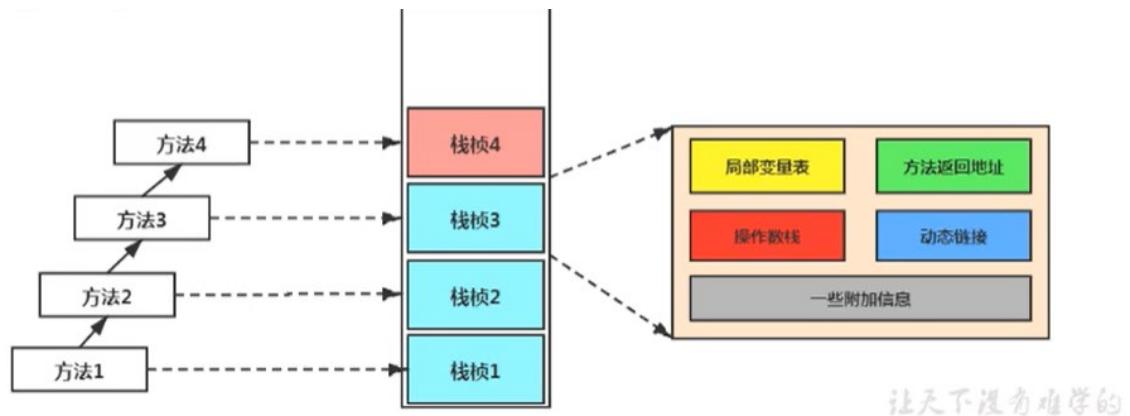
如果当前方法调用了其他方法，方法返回之际，当前栈帧会传回此方法的执行结果给前一个栈帧，接着，虚拟机会丢弃当前栈帧，使得前一个栈帧重新成为当前栈帧。

Java 方法有两种返回函数的方式，一种是正常的函数返回，使用 `return` 指令；另外一种是抛出异常。不管使用哪种方式，都会导致栈帧被弹出。

栈帧的内部结构

每个栈帧中存储着：

- 局部变量表（Local Variables）
- 操作数栈（operand Stack）（或表达式栈）
- 动态链接（Dynamic Linking）（或指向运行时常量池的方法引用）
- 方法返回地址（Return Address）（或方法正常退出或者异常退出的定义）
- 一些附加信息



并行每个线程下的栈都是私有的，因此每个线程都有自己各自的栈，并且每个栈里面都有很多栈帧，栈帧的大小主要由局部变量表 和 操作数栈决定的



局部变量表

局部变量表：Local Variables，被称之为局部变量数组或本地变量表

定义为一个数字数组，主要用于存储方法参数和定义在方法体内的局部变量这些数据类型包括各类基本数据类型、对象引用（reference），以及 returnAddress 类型。

由于局部变量表建立在线程的栈上，是线程的私有数据，因此不存在数据安全问题（主要说的是数据共享的问题）

局部变量表所需的容量大小是在编译期确定下来的，并保存在方法的 Code 属性的 maximum local variables 数据项中。在方法运行期间是不会改变局部变量表的大小的。

方法嵌套调用的次数由栈的大小决定。一般来说，栈越大，方法嵌套调用次数越多。对一个函数而言，它的参数和局部变量越多，使得局部变量表膨胀，它的栈帧就越

大，以满足方法调用所需传递的信息增大的需求。进而函数调用就会占用更多的栈空间，导致其嵌套调用次数就会减少。

局部变量表中的变量只在当前方法调用中有效。在方法执行时，虚拟机通过使用局部变量表完成参数值到参数变量列表的传递过程。当方法调用结束后，随着方法栈帧的销毁，局部变量表也会随之销毁。

关于 Slot 的理解

参数值的存放总是在局部变量数组的 index0 开始，到数组长度-1 的索引结束。

局部变量表，最基本的存储单元是 Slot（变量槽）局部变量表中存放编译期可知的各种基本数据类型（8 种），引用类型（reference），returnAddress 类型的变量。

在局部变量表里，32 位以内的类型只占用一个 slot（包括 returnAddress 类型），64 位的类型（long 和 double）占用两个 slot。

byte、short、char 在存储前被转换为 int，boolean 也被转换为 int，0 表示 false，非 0 表示 true。
long 和 double 则占据两个 slot。

JVM 会为局部变量表中的每一个 Slot 都分配一个访问索引，通过这个索引即可成功访问到局部变量表中指定的局部变量值

当一个实例方法被调用的时候，它的方法参数和方法体内部定义的局部变量将会按照顺序被复制到局部变量表中的每一个 slot 上

如果需要访问局部变量表中一个 64bit 的局部变量值时，只需要使用前一个索引即可。（比如：访问 long 或 double 类型变量）

如果当前帧是由构造方法或者实例方法创建的，那么该对象引用 this 将会存放在 index 为 0 的 slot 处，其余的参数按照参数表顺序继续排列。

这也解释了为什么静态方法中不能使用 this 关键字。

因为静态方法的 0 的 slot 位置处不是 this。

索引	类型	参数
0	int	int k
1	long	long m
3	float	float p
4	double	double q
6	reference	Object t

Slot 的重复利用

栈帧中的局部变量表中的槽位是可以重用的，如果一个局部变量过了其作用域，那么在其作用域之后申明的新的局部变就很有可能会复用过期局部变量的槽位，从而达到节省资源的目的。下面的函数localVar2的局部变量表中有2个变量，分别是this、b。

```
public class SlotTest {  
    public void localVar1() {  
        int a = 0;  
        System.out.println(a);  
        int b = 0;  
    }  
  
    public void localVar2() {  
        {  
            int a = 0;  
            System.out.println(a);  
        }  
        //此时的b就会复用a的槽位  
        int b = 0;  
    }  
}
```

静态变量与局部变量的对比

变量的分类：

- 按数据类型分：基本数据类型、引用数据类型
- 按类中声明的位置分：成员变量（类变量，实例变量）、局部变量
 - 类变量：linking 的 prepare 阶段，给类变量赋对应类型的默认值，init 阶段给类变量显示赋值即静态代码块
 - 实例变量：随着对象创建，会在堆空间中分配实例变量空间，并进行默认赋值
 - 局部变量：在使用前必须进行显式赋值，不然编译不通过。

参数表分配完毕之后，再根据方法体内定义的变量的顺序和作用域分配。

我们知道类变量表有两次初始化的机会，第一次是在“准备阶段”，执行系统初始化，对类变量设置零值，另一次则是在“初始化”阶段，赋予程序员在代码中定义的初始值。

和类变量初始化不同的是，局部变量表不存在系统初始化的过程，这意味着一旦定义了局部变量则必须人为的初始化，否则无法使用。

在栈帧中，与性能调优关系最为密切的部分就是前面提到的局部变量表。在方法执行时，虚拟机使用局部变量表完成方法的传递。

局部变量表中的变量也是重要的垃圾回收根节点，只要被局部变量表中直接或间接引用的对象都不会被回收。

操作数栈

概念

操作数栈：Operand Stack

每一个独立的栈帧除了包含局部变量表以外，还包含一个后进先出（Last - In - First - Out）的操作数栈，也可以称之为表达式栈（Expression Stack）

操作数栈，在方法执行过程中，根据字节码指令，往栈中写入数据或提取数据，即入栈（push）和出栈（pop）

- 某些字节码指令将值压入操作数栈，其余的字节码指令将操作数取出栈。使用它们后再把结果压入栈
- 比如：执行复制、交换、求和等操作



代码举例

代码举例

```
public void testAddOperation() {  
    byte i = 15;  
    int j = 8;  
    int k = i + j;  
}
```

字节码指令信息

```
public void testAddOperation();  
Code:  
0: bipush      15  
2: istore_1  
3: bipush      8  
5: istore_2  
6: iload_1  
7: iload_2  
8: iadd  
9: istore_3  
10: return
```

操作数栈，主要用于保存计算过程的中间结果，同时作为计算过程中变量临时的存储空间。

操作数栈就是 JVM 执行引擎的一个工作区，当一个方法刚开始执行的时候，一个新的栈帧也会随之被创建出来，这个方法的操作数栈是空的。.

这个时候数组是有长度的，因为数组一旦创建，那么其长度就是不可变的

每一个操作数栈都会拥有一个明确的栈深度用于存储数值，其所需的最大深度在编译期就定义好了，保存在方法的 Code 属性中，为 maxstack 的值。

栈中的任何一个元素都是可以任意的 Java 数据类型

- 32bit 的类型占用一个栈单位深度
- 64bit 的类型占用两个栈单位深度

操作数栈并非采用访问索引的方式来进行数据访问的，而是只能通过标准的入栈和出栈操作来完成一次数据访问

如果被调用的方法带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新 PC 寄存器中下一条需要执行的字节码指令。

操作数栈中元素的数据类型必须与字节码指令的序列严格匹配，这由编译器在编译器期间进行验证，同时在类加载过程中的类检验阶段的数据流分析阶段要再次验证。

|

另外，我们说 Java 虚拟机的解释引擎是基于栈的执行引擎，其中的栈指的就是操作数栈。

代码追踪

我们给定代码

```
public void testAddOperation() {  
    byte i = 15;  
    int j = 8;  
    int k = i + j;  
}
```

使用 javap 命令反编译 class 文件: javap -v 类名.class

```
public void testAddOperation();  
  Code:  
  0: bipush      15  
  2: istore_1  
  3: bipush      8  
  5: istore_2  
  6: iload_1  
  7: iload_2  
  8: iadd  
  9: istore_3  
 10: return
```

byte、short、char、boolean 内部都是使用 int 型来进行保存的

从上面的代码我们可以知道，我们都是通过 bipush 对操作数 15 和 8 进行入栈操作

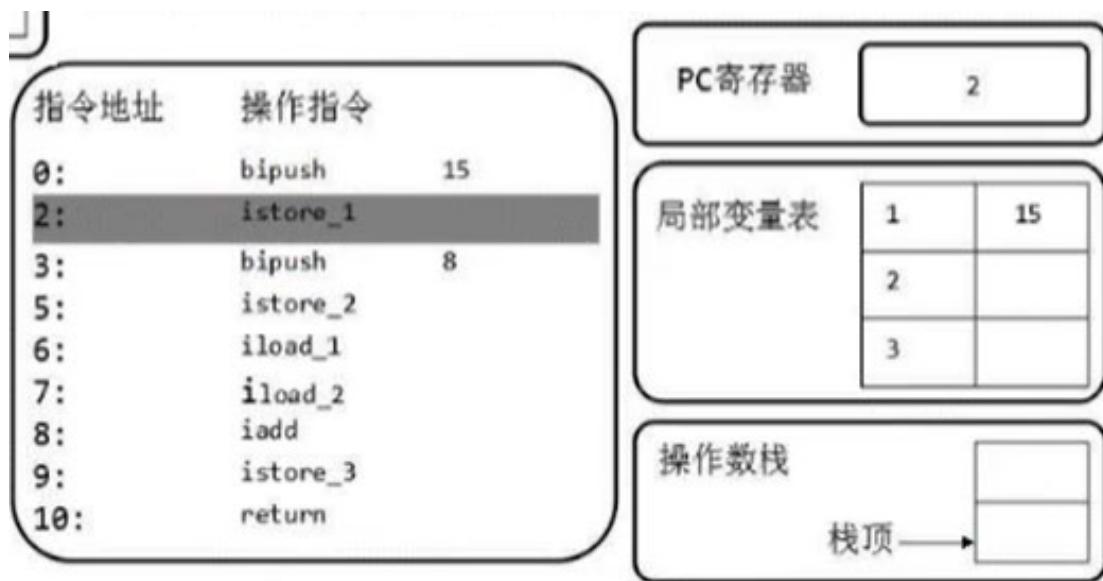
同时使用的是 iadd 方法进行相加操作，i-> 代表的就是 int，也就是 int 类型的加法操作

执行流程如下所示：

首先执行第一条语句，PC 寄存器指向的是 0，也就是指令地址为 0，然后使用 bipush 让操作数 15 入栈。



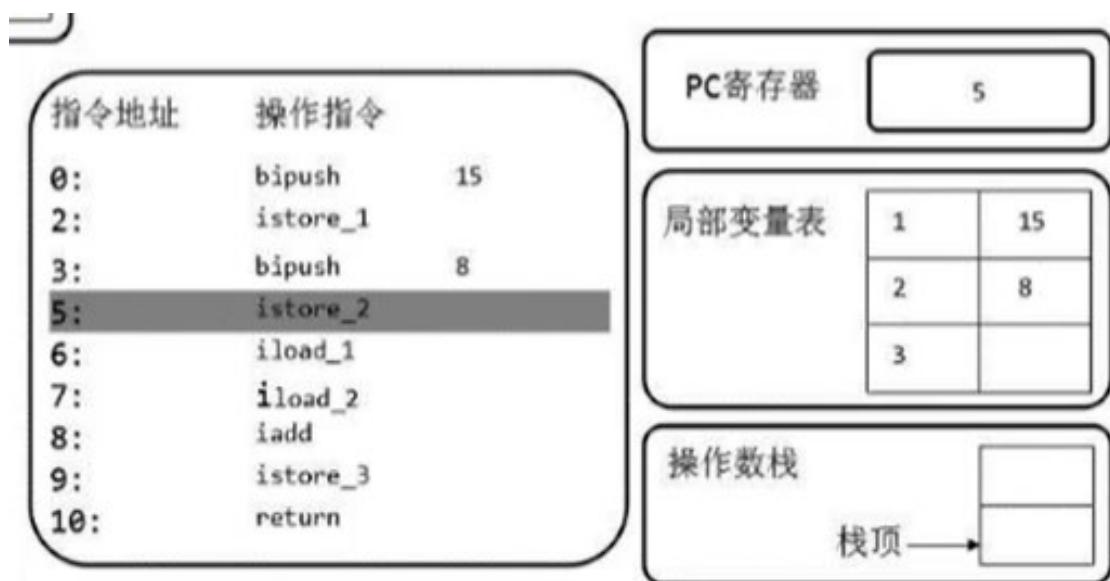
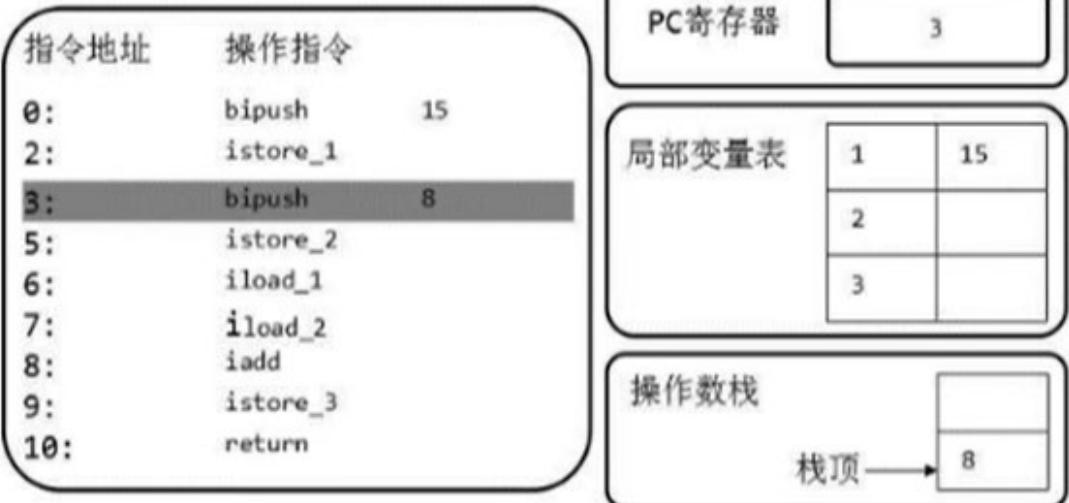
执行完后，让 $PC + 1$ ，指向下一行代码，下一行代码就是将操作数栈的元素存储到局部变量表 1 的位置，我们可以看到局部变量表已经增加了一个元素



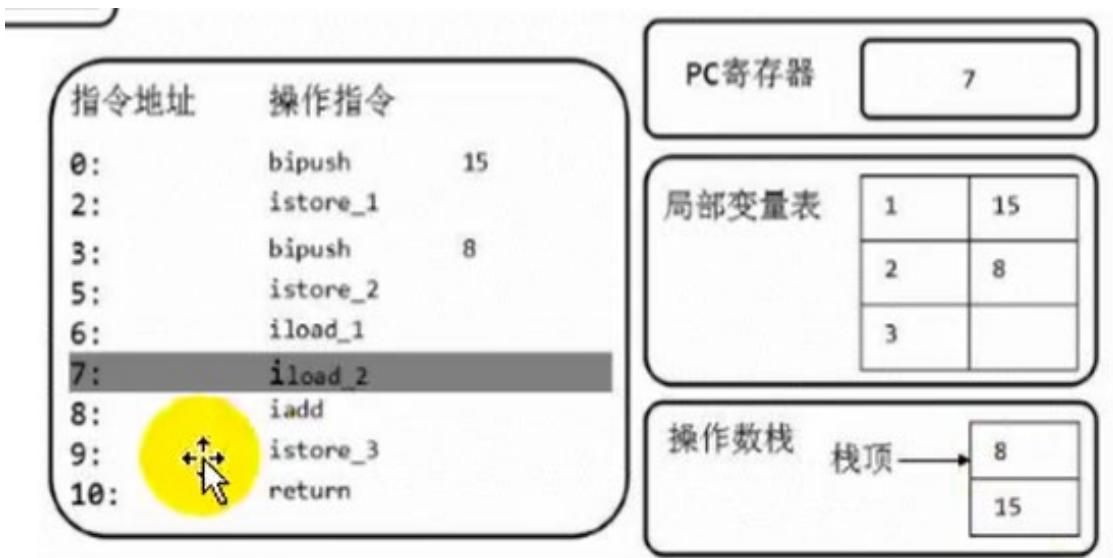
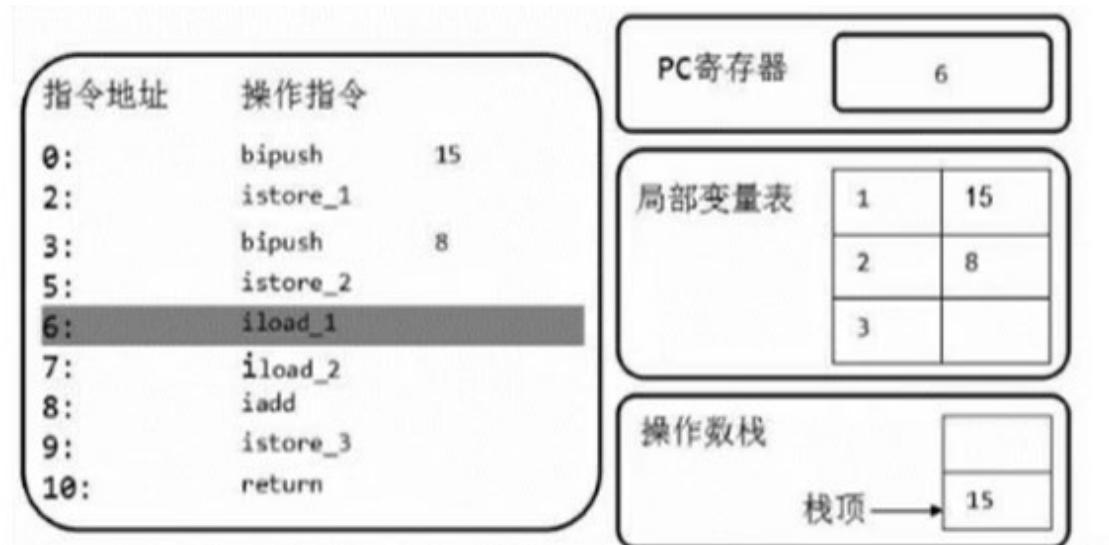
为什么局部变量表不是从 0 开始的呢？

其实局部变量表也是从 0 开始的，但是因为 0 号位置存储的是 this 指针，所以说就直接省略了~

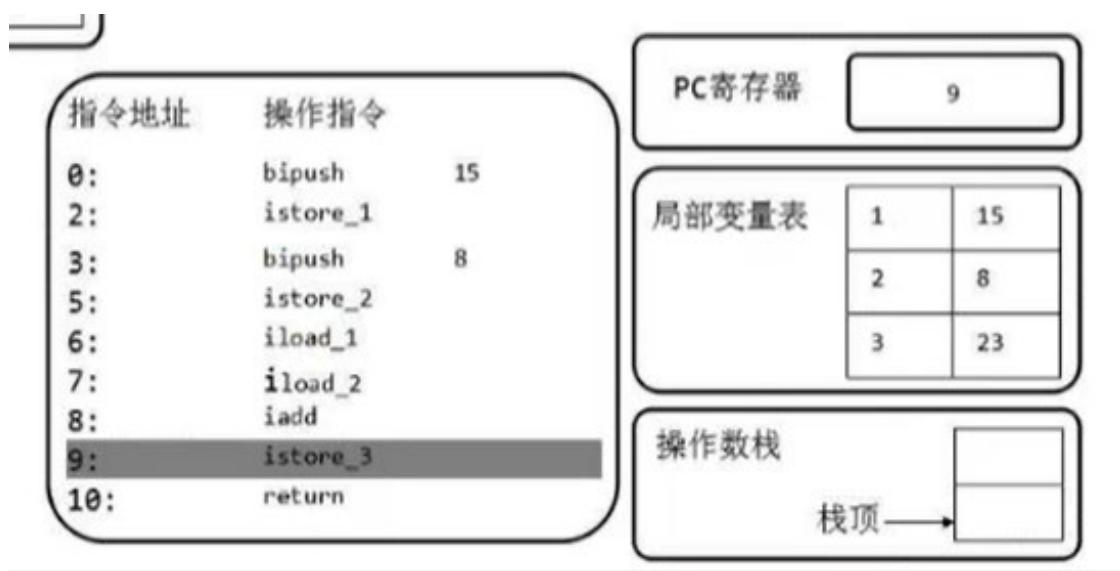
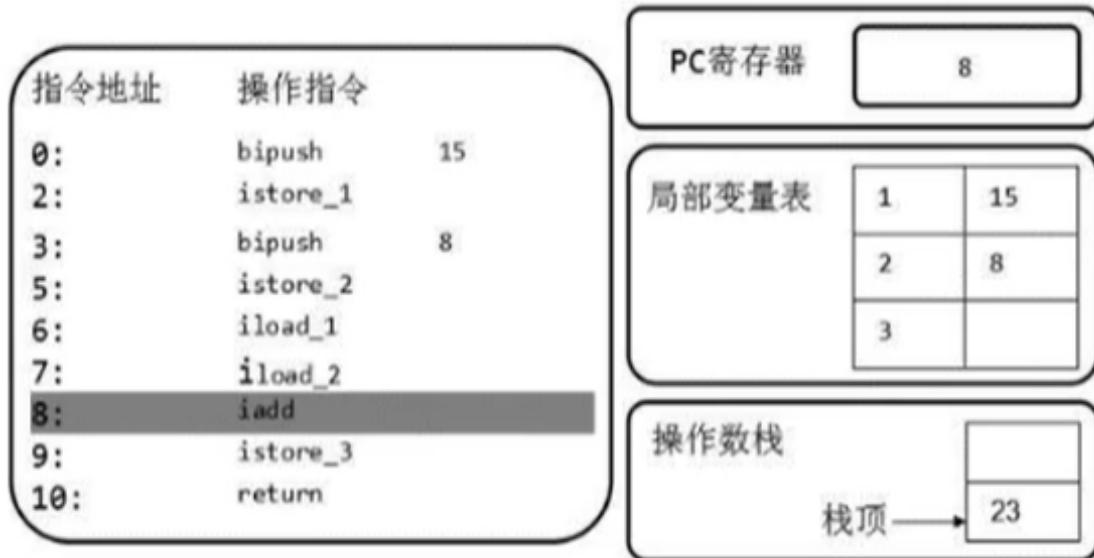
然后 $PC+1$ ，指向的是下一行。让操作数 8 也入栈，同时执行 store 操作，存入局部变量表中



然后从局部变量表中，依次将数据放在操作数栈中



然后将操作数栈中的两个元素执行相加操作，并存储在局部变量表 3 的位置



最后 PC 寄存器的位置指向 10，也就是 return 方法，则直接退出方法

栈顶缓存技术

栈顶缓存技术：Top Of Stack Cashing

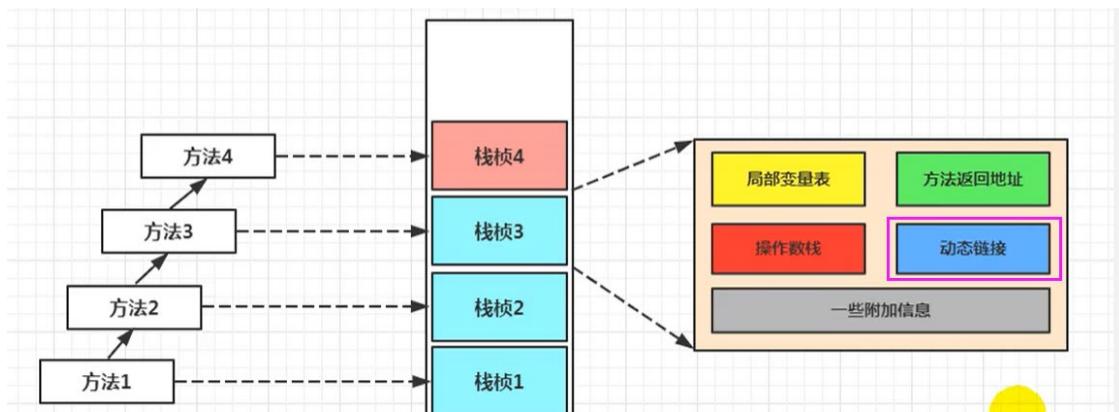
前面提过，基于栈式架构的虚拟机所使用的零地址指令更加紧凑，但完成一项操作的时候必然需要使用更多的入栈和出栈指令，这同时也意味着将需要更多的指令分派（instruction dispatch）次数和内存读/写次数。

由于操作数是存储在内存中的，因此频繁地执行内存读/写操作必然会影响执行速度。为了解决这个问题，HotSpot JVM 的设计者们提出了栈顶缓存（Tos，Top-of-Stack Cashing）技术，将栈顶元素全部缓存在物理 CPU 的寄存器中，以此降低对内存的读/写次数，提升执行引擎的执行效率。

寄存器：指令更少，执行速度快

动态链接

动态链接：Dynamic Linking



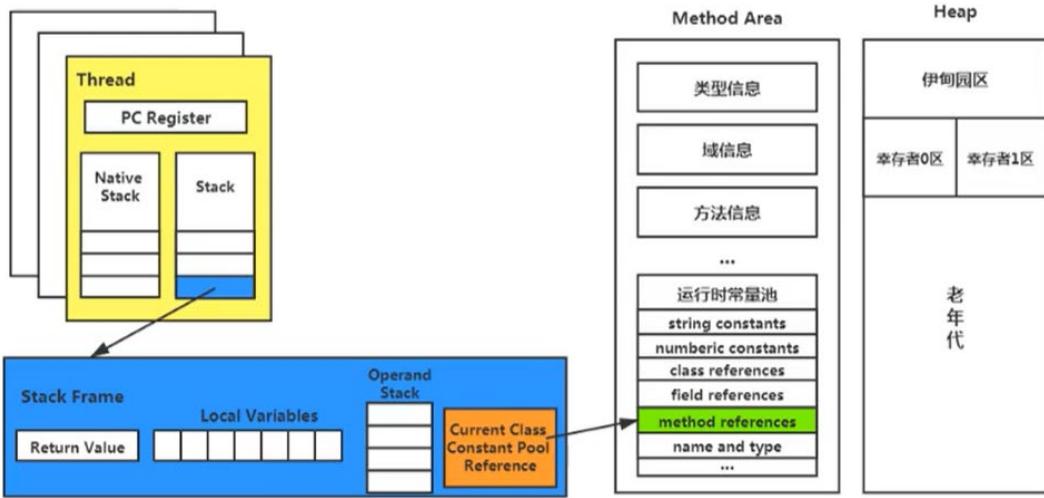
动态链接、方法返回地址、附加信息：有些地方被称为帧数据区

每一个栈帧内部都包含一个指向运行时常量池中该栈帧所属方法的引用，包含这个引用的目的就是为了支持当前方法的代码能够实现动态链接（Dynamic Linking）。比如：invokedynamic 指令

在 Java 源文件被编译到字节码文件中时，所有的变量和方法引用都作为符号引用（symbolic Reference）保存在 class 文件的常量池里。

比如：描述一个方法调用了另外的其他方法时，就是通过常量池中指向方法的符号引用来表示的，那么动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用。

符号引用就是一个类中（当然不仅是类，还包括类的其他部分，比如方法，字段等），引入了其他的类，可是JVM并不知道引入的其他类在哪里，所以就用唯一符号来代替，等到类加载器去解析的时候，就把符号引用找到那个引用类的地址，这个地址也就是直接引用。



为什么需要运行时常量池？

因为在不同的方法，都可能调用常量或者方法，所以只需要存储一份即可，节省了空间

常量池的作用：就是为了提供一些符号和常量，便于指令的识别

方法调用：解析与分配

在 JVM 中，将符号引用转换为调用方法的直接引用与方法的绑定机制相关

链接

静态链接

当一个字节码文件被装载进 JVM 内部时，如果被调用的目标方法在编译期克制，且运行期保持不变时，这种情况下降调用方法的符号引用转换为直接引用的过程称之为静态链接

动态链接

如果被调用的方法在编译期无法被确定下来，也就是说，只能够在程序运行期将调用的方法的符号转换为直接引用，由于这种引用转换过程具备动态性，因此也称之为动态链接。

绑定机制

对应的方法的绑定机制为：早期绑定（Early Binding）和晚期绑定（Late Binding）。绑定是一个字段、方法或者类在符号引用被替换为直接引用的过程，这仅仅发生一次。

早期绑定

早期绑定就是指被调用的目标方法如果在编译期可知，且运行期保持不变时，即可将这个方法与所属的类型进行绑定，这样一来，由于明确了被调用的目标方法究竟是哪一个，因此也就可以使用静态链接的方式将符号引用转换为直接引用。

晚期绑定

如果被调用的方法在编译期无法被确定下来，只能够在程序运行期根据实际的类型绑定相关的方法，这种绑定方式也就被称之为晚期绑定。比如多态

早晚期绑定的发展历史

随着高级语言的横空出世，类似于 Java 一样的基于面向对象的编程语言如今越来越多，尽管这类编程语言在语法风格上存在一定的差别，但是它们彼此之间始终保持着一个共性，那就是都支持封装、继承和多态等面向对象特性，既然这一类的编程语言具备多态特悄，那么自然也就具备早期绑定和晚期绑定两种绑定方式。

Java 中任何一个普通的方法其实都具备虚函数的特征，它们相当于 C++语言中的虚函数（C++中则需要使用关键字 `virtual` 来显式定义）。如果在 Java 程序中不希望某个方法拥有虚函数的特征时，则可以使用关键字 `final` 来标记这个方法。

虚方法和非虚方法

- 如果方法在编译期就确定了具体的调用版本，这个版本在运行时是不可变的。这样的方法称为非虚方法。主要指编译期就能确定的方法。
- 静态方法、私有方法、`final` 方法、实例构造器、父类方法都是非虚方法。
- 其他方法称为虚方法。

子类对象的多态的使用前提

- 类的继承关系
- 方法的重写

虚拟机中提供了以下几条方法调用指令：

普通调用指令：

- `invokestatic`: 调用静态方法，解析阶段确定唯一方法版本（非虚方法）
- `invokespecial`: 调用方法、私有及父类方法，解析阶段确定唯一方法版本（非虚）
- `invokevirtual`: 调用所有虚方法（虚方法）
- `invokeinterface`: 调用接口方法（虚方法）

动态调用指令:

- **invokedynamic**: 动态解析出需要调用的方法，然后执行

前四条指令固化在虚拟机内部，方法的调用执行不可人为干预，而 **invokedynamic** 指令则支持由用户确定方法版本。其中 **invokestatic** 指令和 **invokespecial** 指令调用的方法称为非虚方法，其余的（**final** 修饰的除外）称为虚方法。

invokedynamic 指令

JVM 字节码指令集一直比较稳定，一直到 Java7 中才增加了一个 **invokedynamic** 指令，这是 Java 为了实现动态类型语言】支持而做的一种改进。

但是在 Java7 中并没有提供直接生成 **invokedynamic** 指令的方法，需要借助 ASM 这种底层字节码工具来产生 **invokedynamic** 指令。直到 Java8 的 Lambda 表达式的出现，**invokedynamic** 指令的生成，在 Java 中才有了直接的生成方式。

Java7 中增加的动态语言类型支持的本质是对 Java 虚拟机规范的修改，而不是对 Java 语言规则的修改，这一块相对来讲比较复杂，增加了虚拟机中的方法调用，最直接的受益者就是运行在 Java 平台的动态语言的编译器。

动态类型语言和静态类型语言

动态类型语言和静态类型语言两者的区别就在于对类型的检查是在编译期还是在运行期，满足前者就是静态类型语言，反之是动态类型语言。

说的再直白一点就是，静态类型语言是判断变量自身的类型信息；动态类型语言是判断变量值的类型信息，变量没有类型信息，变量值才有类型信息，这是动态语言的一个重要特征。

Java: String info = "abcefg"; (Java 是静态类型语言的，会先编译就进行类型检查)

JS: var name = "shkstart"; var name = 10; (运行时才进行检查)

方法重写的本质

Java 语言中方法重写本质:

- 找到操作数栈顶的第一个元素所执行的对象的实际类型，记作 C。
- 如果在类型 C 中找到与常量中的描述符合简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；如果不通过，则返回 **java.lang.IllegalAccessError** 异常。
- 否则，按照继承关系从下往上依次对 C 的各个父类进行第 2 步的搜索和验证过程。
- 如果始终没有找到合适的方法，则抛出 **java.lang.AbstractMethodError** 异常。

IllegalAccessError 介绍

程序试图访问或修改一个属性或调用一个方法，这个属性或方法，你没有权限访问。一般的，这个会引起编译器异常。这个错误如果发生在运行时，就说明一个类发生了不兼容的改变。

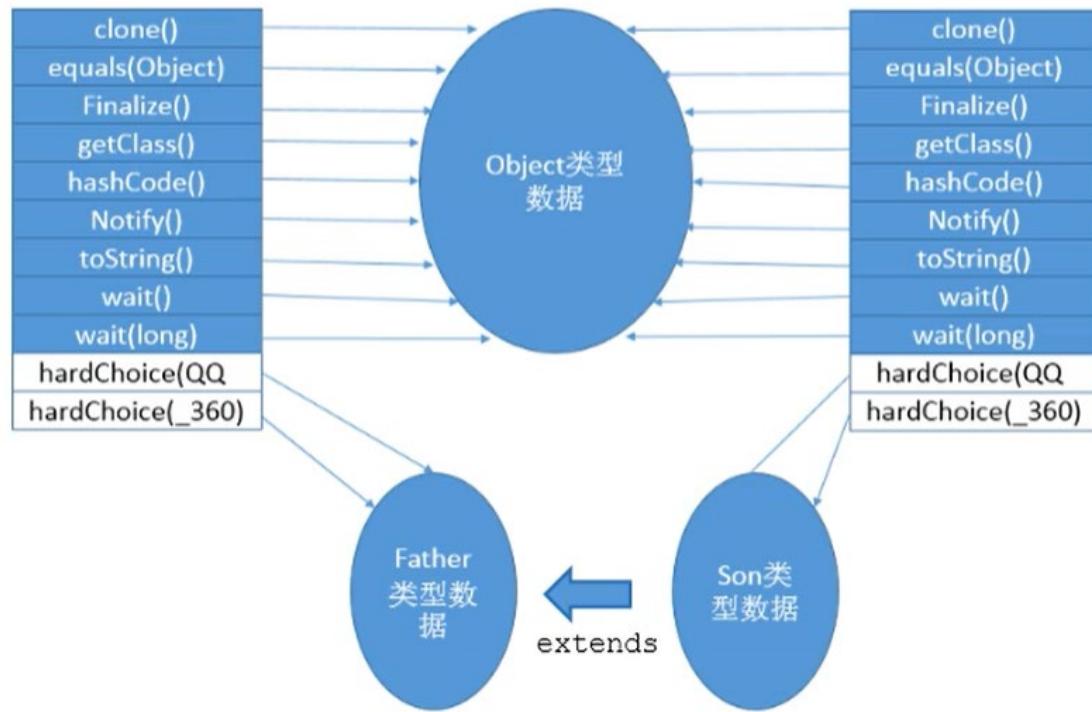
方法的调用：虚方法表

在面向对象的编程中，会很频繁的使用到动态分派，如果在每次动态分派的过程中都要重新在类的方法元数据中搜索合适的目标的话就可能影响到执行效率。因此，为了提高性能，JVM 采用在类的方法区建立一个虚方法表（virtual method table）（非虚方法是确定的，不会出现在表中）来实现。使用索引表来代替查找。

每个类中都有一个虚方法表，表中存放着各个方法的实际入口。

虚方法表是什么时候被创建的呢？

虚方法表会在类加载的链接阶段被创建并开始初始化，类的变量初始值准备完成之后，JVM 会把该类的方法表也初始化完毕。



如上图所示：如果类中重写了方法，那么调用的时候，就会直接在虚方法表中查找，否则将会直接连接到 Object 的方法中。虚方法表的创建提高了方法调用的效率。

方法返回地址

存放调用该方法的 pc 寄存器的值。一个方法的结束，有两种方式：

- 正常执行完成
- 出现未处理的异常，非正常退出

无论通过哪种方式退出，在方法退出后都返回到该方法被调用的位置。方法正常退出时，调用者的 pc 计数器的值作为返回地址，即调用该方法的指令的下一条指令的地址。而通过异常退出的，返回地址是要通过异常表来确定，栈帧中一般不会保存这部分信息。

当一个方法开始执行后，只有两种方式可以退出这个方法：

执行引擎遇到任意一个方法返回的字节码指令（return），会有返回值传递给上层的方法调用者，简称正常完成出口；

- 一个方法在正常调用完成之后，究竟需要使用哪一个返回指令，还需要根据方法返回值的实际数据类型而定。
- 在字节码指令中，返回指令包含 ireturn（当返回值是 boolean, byte, char, short 和 int 类型时使用），lreturn（Long 类型），freturn（Float 类型），dreturn（Double 类型），areturn。另外还有一个 return 指令声明为 void 的方法，实例初始化方法，类和接口的初始化方法使用。

在方法执行过程中遇到异常（Exception），并且这个异常没有在方法内进行处理，也就是只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，简称异常完成出口。

方法执行过程中，抛出异常时的异常处理，存储在一个异常处理表，方便在发生异常的时候找到处理异常的代码

Exception		table:	
from	to	target	type
4	16	19	any
19	21	19	any

本质上，方法的退出就是当前栈帧出栈的过程。此时，需要恢复上层方法的局部变量表、操作数栈、将返回值压入调用者栈帧的操作数栈、设置 PC 寄存器值等，让调用者方法继续执行下去。

正常完成出口和异常完成出口的区别在于：通过异常完成出口退出的不会给他的上层调用者产生任何的返回值。

一些附加信息

栈帧中还允许携带与 Java 虚拟机实现相关的一些附加信息。例如：对程序调试提供支持的信息。

栈的相关面试题

- 举例栈溢出的情况？（StackOverflowError）
 - 通过 -Xss 设置栈的大小
- 调整栈大小，就能保证不出现溢出么？
 - 不能保证不溢出
- 分配的栈内存越大越好么？
 - 不是，一定时间内降低了 OOM 概率，但是会挤占其它的线程空间，因为整个空间是有限的。
- 垃圾回收是否涉及到虚拟机栈？
 - 不会
- 方法中定义的局部变量是否线程安全？
 - 具体问题具体分析

```
/**  
 * 面试题  
 * 方法中定义局部变量是否线程安全？具体情况具体分析  
 * 何为线程安全？  
 * 如果只有一个线程才可以操作此数据，则必是线程安全的  
 * 如果有多个线程操作，则此数据是共享数据，如果不考虑共享机制，则为线程不  
 * 安全  
 * @author: 陌溪  
 * @create: 2020-07-06-16:08  
 */  
public class StringBuilderTest {  
  
    // s1 的声明方式是线程安全的
```

```

public static void method01() {
    // 线程内部创建的，属于局部变量
    StringBuilder s1 = new StringBuilder();
    s1.append("a");
    s1.append("b");
}

// 这个也是线程不安全的，因为有返回值，有可能被其它的程序调用
public static StringBuilder method04() {
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.append("a");
    stringBuilder.append("b");
    return stringBuilder;
}

// stringBuilder 是线程不安全的，操作的是共享数据
public static void method02(StringBuilder stringBuilder) {
    stringBuilder.append("a");
    stringBuilder.append("b");
}

/**
 * 同时并发的执行，会出现线程不安全的问题
 */
public static void method03() {
    StringBuilder stringBuilder = new StringBuilder();
    new Thread(() -> {
        stringBuilder.append("a");
        stringBuilder.append("b");
    }, "t1").start();

    method02(stringBuilder);
}

// StringBuilder 是线程安全的，但是 String 也可能线程不安全的
public static String method05() {
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.append("a");
    stringBuilder.append("b");
    return stringBuilder.toString();
}
}

```

总结一句话就是：如果对象是在内部产生，并在内部消亡，没有返回到外部，那么它就是线程安全的，反之则是线程不安全的。（没有影响到外部环境就安全）

运行时数据区，是否存在 Error 和 GC？

运行时数据区	是否存在 Error	是否存在 GC
程序计数器	否	否
虚拟机栈	是	否
本地方法栈	是	否
方法区	是 (OOM)	是
堆	是	是

本地方法接口

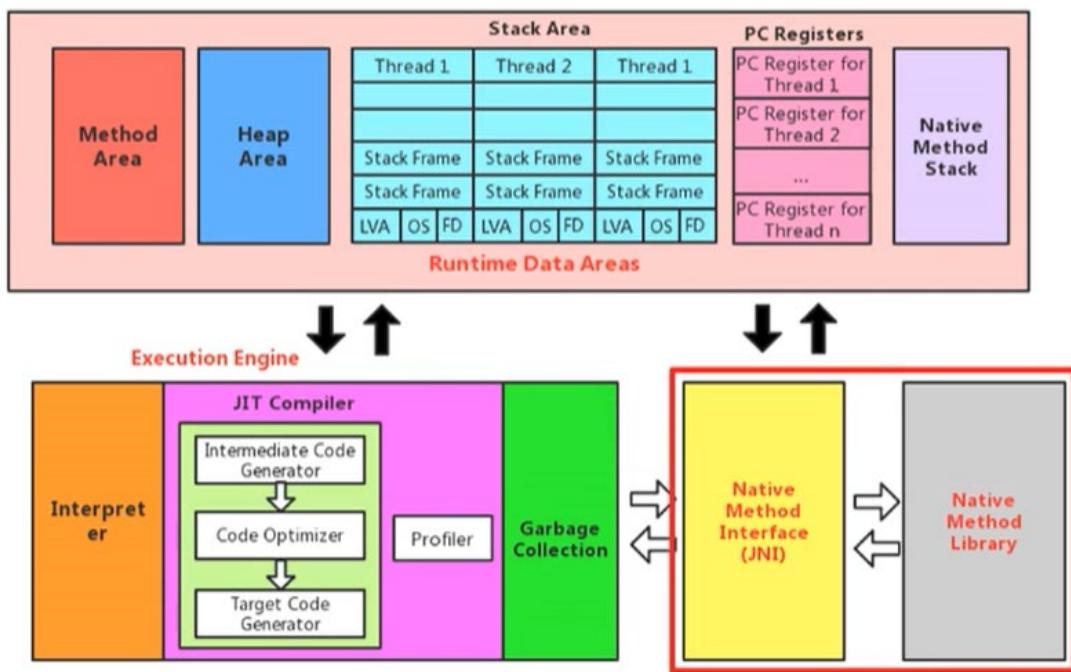
什么是本地方法

简单地讲，一个 Native Method 是一个 Java 调用非 Java 代码的接口。一个 Native Method 是这样一个 Java 方法：该方法的实现由非 Java 语言实现，比如 C。这个特征并非 Java 所特有，很多其它的编程语言都有这一机制，比如在 C++ 中，你可以用 `extern "c"` 告知 c++ 编译器去调用一个 c 的函数。

"A native method is a Java method whose implementation is provided by non-Java code." (本地方法是一个非 Java 的方法，它的具体实现是非 Java 代码的实现)

在定义一个 native method 时，并不提供实现体（有些像定义一个 Java interface），因为其实现体是由非 Java 语言在外面实现的。

本地接口的作用是融合不同的编程语言为 Java 所用，它的初衷是融合 C/C++ 程序。



代码举例说明 Native 方法是如何编写的

```
/**
 * 本地方法
 *
 * @author: 陌溪
 * @create: 2020-07-06-16:45
 */
public class IhaveNatives {
    public native void Native1(int x);
    native static public long Native2();
    native synchronized private float Native3(Object o);
    native void Natives(int[] ary) throws Exception;
}
```

需要注意的是：标识符 `native` 可以与其它 java 标识符连用，但是 `abstract` 除外

为什么使用 Native Method?

Java 使用起来非常方便，然而有些层次的任务用 Java 实现起来不容易，或者我们对程序的效率很在意时，问题就来了。

与 Java 环境的交互

有时 Java 应用需要与 Java 外面的环境交互，这是本地方法存在的主要原因。你可以想想 Java 需要与一些底层系统，如操作系统或某些硬件交换信息时的情况。本

地方法正是这样一种交流机制：它为我们提供了一个非常简洁的接口，而且我们无需去了解 Java 应用之外的繁琐的细节。

与操作系统的交互

JVM 支持着 Java 语言本身和运行时库，它是 Java 程序赖以生存的平台，它由一个解释器（解释字节码）和一些连接到本地代码的库组成。然而不管怎样，它毕竟不是一个完整的系统，它经常依赖于底层系统的支持。这些底层系统常常是强大的操作系统。通过使用本地方法，我们得以用 Java 实现了 jre 的与底层系统的交互，甚至 JVM 的一些部分就是用 c 写的。还有，如果我们要使用一些 Java 语言本身没有提供封装的操作系统的特性时，我们也需要使用本地方法。

Sun's Java

Sun 的解释器是用 C 实现的，这使得它能像一些普通的 C 一样与外部交互。jre 大部分是用 Java 实现的，它也通过一些本地方法与外界交互。例如：类 `java.lang.Thread` 的 `setPriority()` 方法是用 Java 实现的，但是它实现调用的是该类里的本地方法 `setPriority0()`。这个本地方法是用 C 实现的，并被植入 JVM 内部，在 Windows 95 的平台上，这个本地方法最终将调用 Win32 `SetPriority()` API。这是一个本地方法的具体实现由 JVM 直接提供，更多的情况是本地方法由外部的动态链接库（external dynamic link library）提供，然后被 JVW 调用。

现状

目前该方法使用的越来越少了，除非是与硬件有关的应用，比如通过 Java 程序驱动打印机或者 Java 系统管理生产设备，在企业级应用中已经比较少见。因为现在的异构领域间的通信很发达，比如可以使用 Socket 通信，也可以使用 Web Service 等等，不多做介绍。

本地方法栈

Java 虚拟机栈于管理 Java 方法的调用，而本地方法栈用于管理本地方法的调用。

本地方法栈，也是线程私有的。

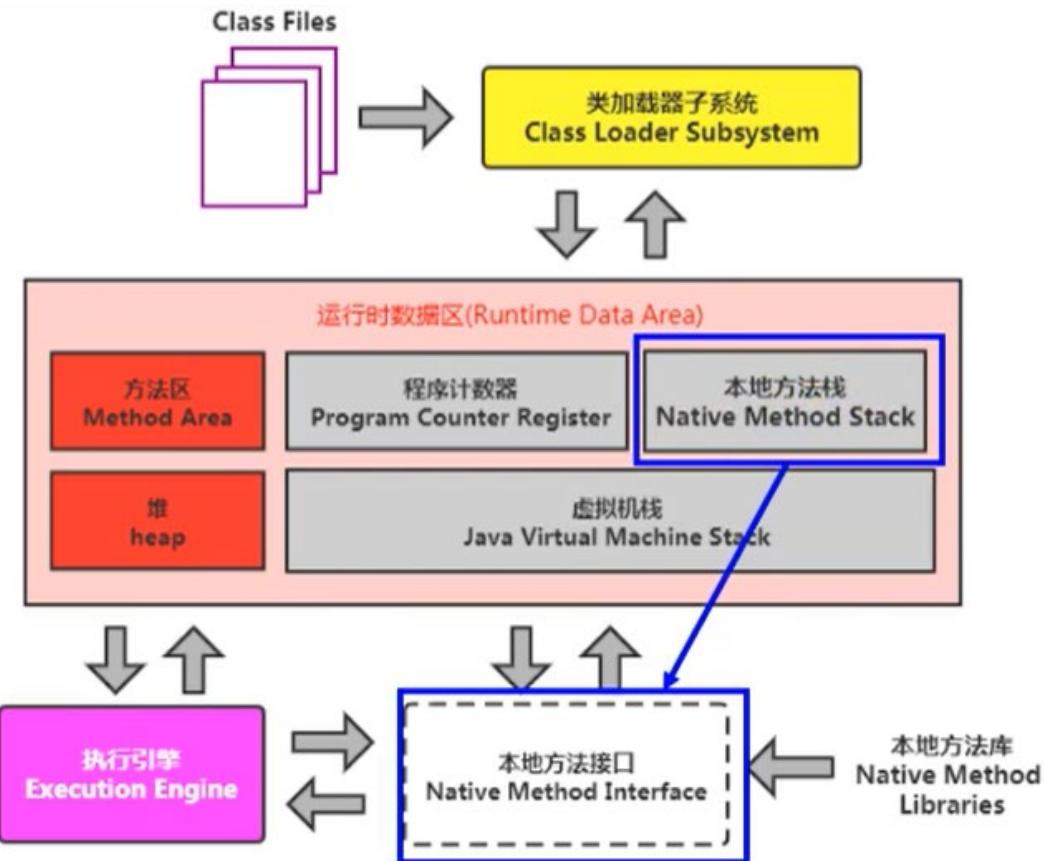
允许被实现成固定或者是可动态扩展的内存大小。（在内存溢出方面是相同的）

- 如果线程请求分配的栈容量超过本地方法栈允许的最大容量，Java 虚拟机将会抛出一个 `StackOverflowError` 异常。

- 如果本地方法栈可以动态扩展，并且在尝试扩展的时候无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的本地方法栈，那么 Java 虚拟机将会抛出一个 `outofMemoryError` 异常。

本地方法是使用 C 语言实现的。

它的具体做法是 Native Method Stack 中登记 native 方法，在 Execution Engine 执行时加载本地方法库。



当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界。它和虚拟机拥有同样的权限。

- 本地方法可以通过本地方法接口来访问虚拟机内部的运行时数据区。
- 它甚至可以直接使用本地处理器中的寄存器
- 直接从本地内存的堆中分配任意数量的内存。

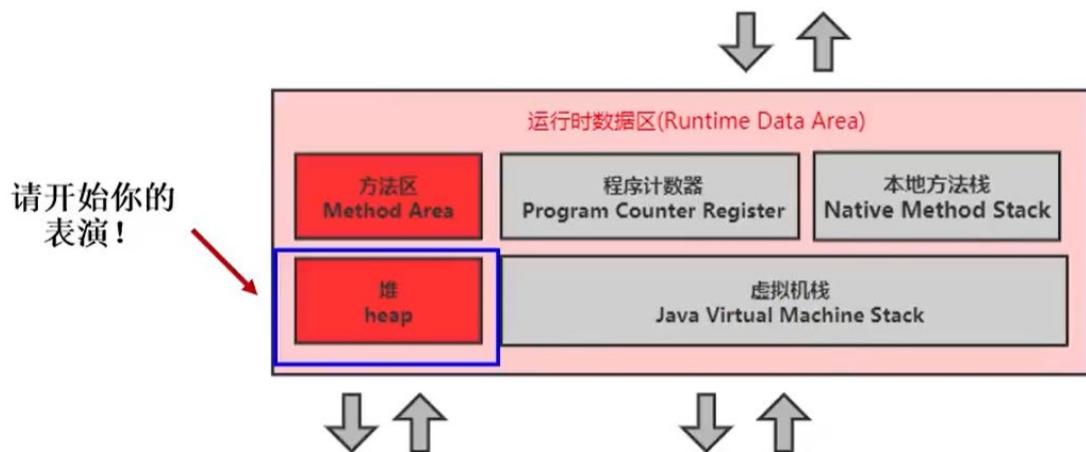
并不是所有的 JVM 都支持本地方法。因为 Java 虚拟机规范并没有明确要求本地方方法栈的使用语言、具体实现方式、数据结构等。如果 JVM 产品不打算支持 native 方法，也可以无需实现本地方方法栈。

在 Hotspot JVM 中，直接将本地方法栈和虚拟机栈合二为一。

堆

堆的核心概念

堆针对一个 JVM 进程来说是唯一的，也就是一个进程只有一个 JVM，但是进程包含多个线程，他们是共享同一堆空间的。



一个 JVM 实例只存在一个堆内存，堆也是 Java 内存管理的核心区域。

Java 堆区在 JVM 启动的时候即被创建，其空间大小也就确定了。是 JVM 管理的最大一块内存空间。

- 堆内存的大小是可以调节的。

《Java 虚拟机规范》规定，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连续的。

所有的线程共享 Java 堆，在这里还可以划分线程私有的缓冲区（Thread Local Allocation Buffer，TLAB）。

-Xms10m: 最小堆内存

-Xmx10m: 最大堆内存

下图就是使用：Java VisualVM 查看堆空间的内容，通过 jdk bin 提供的插件



《Java 虚拟机规范》中对 Java 堆的描述是：所有的对象实例以及数组都应当在运行时分配在堆上。（The heap is the run-time data area from which memory for all class instances and arrays is allocated）

我要说的是：“几乎”所有的对象实例都在这里分配内存。—从实际使用角度看的。

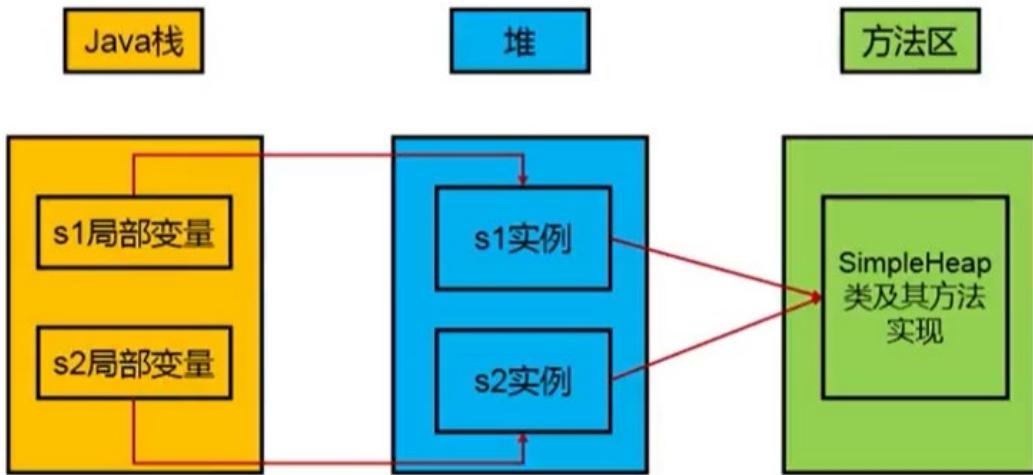
- 因为还有一些对象是在栈上分配的

数组和对象可能永远不会存储在栈上，因为栈帧中保存引用，这个引用指向对象或者数组在堆中的位置。

在方法结束后，堆中的对象不会马上被移除，仅仅在垃圾收集的时候才会被移除。

- 也就是触发了 GC 的时候，才会进行回收
- 如果堆中对象马上被回收，那么用户线程就会受到影响，因为有 stop the word

堆，是 GC (Garbage Collection，垃圾收集器) 执行垃圾回收的重点区域。



堆内存细分

Java 7 及之前堆内存逻辑上分为三部分：新生区+养老区+永久区

- Young Generation Space 新生区 Young/New 又被划分为 Eden 区和 Survivor 区
- Tenure generation space 养老区 Old/Tenure
- Permanent Space 永久区 Perm

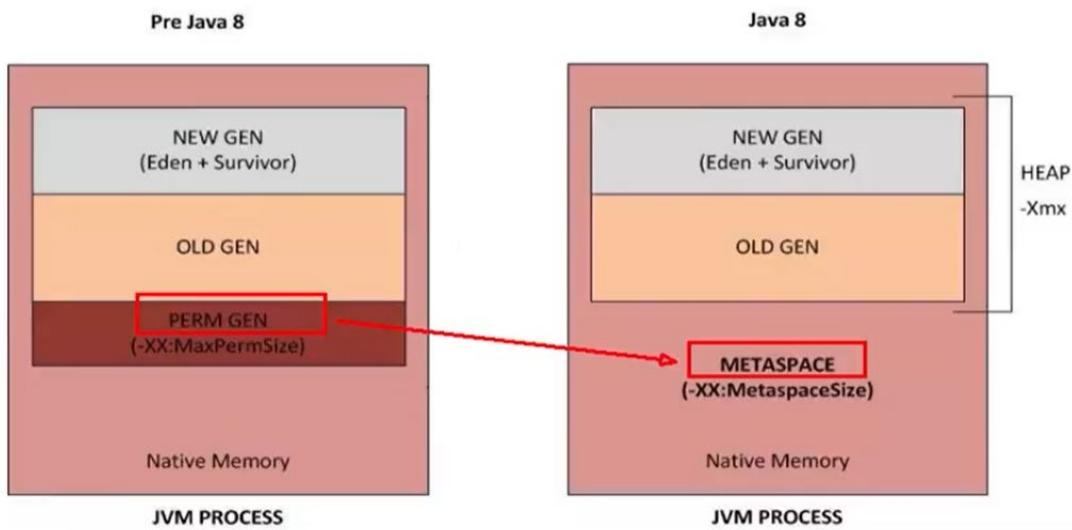
Java 8 及之后堆内存逻辑上分为三部分：新生区+养老区+元空间

- Young Generation Space 新生区 Young/New 又被划分为 Eden 区和 Survivor 区
- Tenure generation space 养老区 Old/Tenure
- Meta Space 元空间 Meta

约定：新生区 -> 新生代 -> 年轻代、 养老区 -> 老年区 -> 老年代、 永久区 -> 永久代



堆空间内部结构, JDK1.8 之前从永久代替换成元空间



设置堆内存大小与 OOM

Java 堆区用于存储 Java 对象实例，那么堆的大小在 JVM 启动时就已经设定好了，大家可以通过选项"-Xmx"和"-Xms"来进行设置。

- "-Xms"用于表示堆区的起始内存，等价于-xx:InitialHeapSize
- "-Xmx"则用于表示堆区的最大内存，等价于-XX:MaxHeapSize

一旦堆区中的内存大小超过"-xmx"所指定的最大内存时，将会抛出 OutOfMemoryError 异常。

通常会将-Xms 和-Xmx 两个参数配置相同的值，其目的是为了能够在 Java 垃圾回收机制清理完堆区后不需要重新分隔计算堆区的大小，从而提高性能。

默认情况下

- 初始内存大小：物理电脑内存大小/64
- 最大内存大小：物理电脑内存大小/4

```
/*
 * -Xms 用来设置堆空间（年轻代+老年代）的初始内存大小
 * -X: 是 jvm 运行参数
 * ms: memory start
 * -Xmx: 用来设置堆空间（年轻代+老年代）的最大内存大小
 *
 * @author: 陌溪
 * @create: 2020-07-06-20:44
 */
public class HeapSpaceInitial {
    public static void main(String[] args) {
        // 返回 Java 虚拟机中的堆内存总量
        long initialMemory = Runtime.getRuntime().totalMemory() / 1024
        / 1024;
        // 返回 Java 虚拟机试图使用的最大堆内存
        long maxMemory = Runtime.getRuntime().maxMemory() / 1024 / 1024;
        System.out.println("-Xms:" + initialMemory + "M");
        System.out.println("-Xmx:" + maxMemory + "M");
    }
}
```

输出结果

```
-Xms:245M
-Xmx:3614M
```

如何查看堆内存的内存分配情况

```
jps -> jstat -gc 进程 id
```

```
::\Users\Administrator>jstat -gc 9228
S0C   S1C   S0U   S1U     EC     EU     OC     OU      MC      MU    CCSC    CCSU    VGC      VGCT    FGC      F
5600.0 25600.0 0.0    0.0 153600.0 12288.1 409600.0    0.0    4480.0 770.4 384.0   75.9      0    0.000    0
```

-XX:+PrintGCDetails

```
Heap
PSYoungGen      total 179200K, used 12288K [0x00000000f3800000, 0x0000000100000000, 0x0000000100000000)
  eden space 153600K, 8% used [0x00000000f3800000,0x00000000f44001b8,0x00000000fce00000)
  from space 25600K, 0% used [0x00000000fe700000,0x00000000fe700000,0x0000000100000000)
  to   space 25600K, 0% used [0x00000000fce00000,0x00000000fce00000,0x00000000fe700000)
ParOldGen       total 409600K, used 0K [0x00000000da800000, 0x00000000f3800000, 0x00000000f3800000)
  object space 409600K, 0% used [0x00000000da800000,0x00000000da800000,0x00000000f3800000)
Metaspace        used 3474K, capacity 4496K, committed 4864K, reserved 1056768K
  class space   used 384K, capacity 388K, committed 512K, reserved 1048576K
```

OutOfMemory 举例

```
public class OOMTest {
    public static void main(String[] args) {
        ArrayList<Picture> list = new ArrayList<>();
        while(true){
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            list.add(new Picture(new Random().nextInt(1024 * 1024)));
        }
    }
}
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  at com.atguigu.java.Picture.<init>(OOMTest.java:25)
  at com.atguigu.java.OOMTest.main(OOMTest.java:16)
```

我们简单的写一个 OOM 例子

```
/*
 * OOM 测试
 *
 * @author: 陌溪
 */
```

```

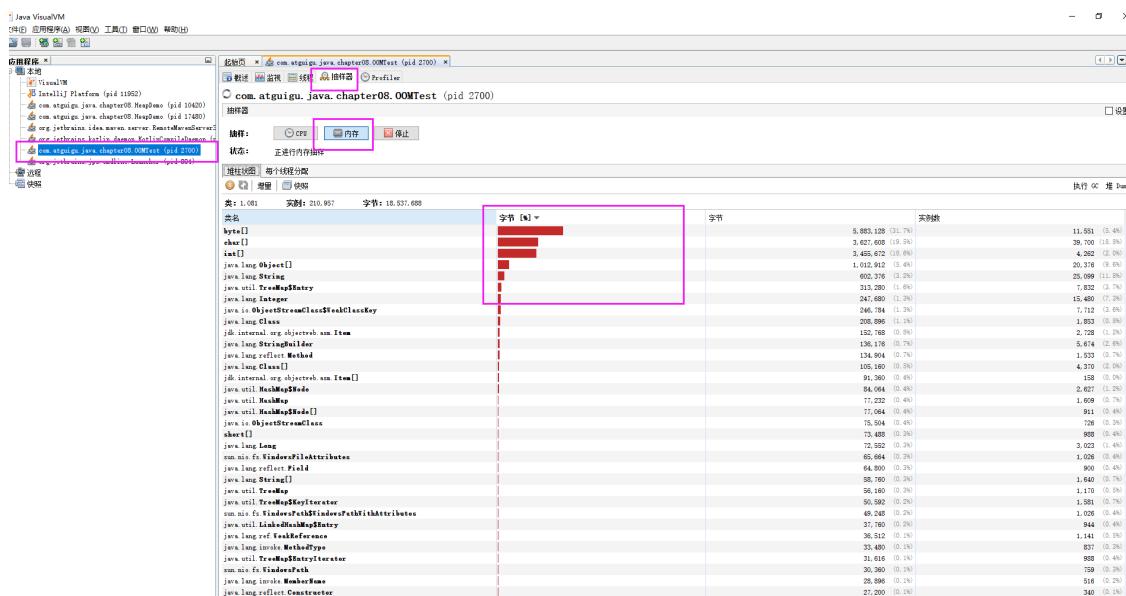
* @create: 2020-07-06-21:11
*/
public class OOMTest {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        while(true) {
            list.add(999999999);
        }
    }
}

```

然后设置启动参数

-Xms10m -Xmx:10m

运行后，就出现 OOM 了，那么我们可以通过 VisualVM 这个工具查看具体是什么参数造成的 OOM



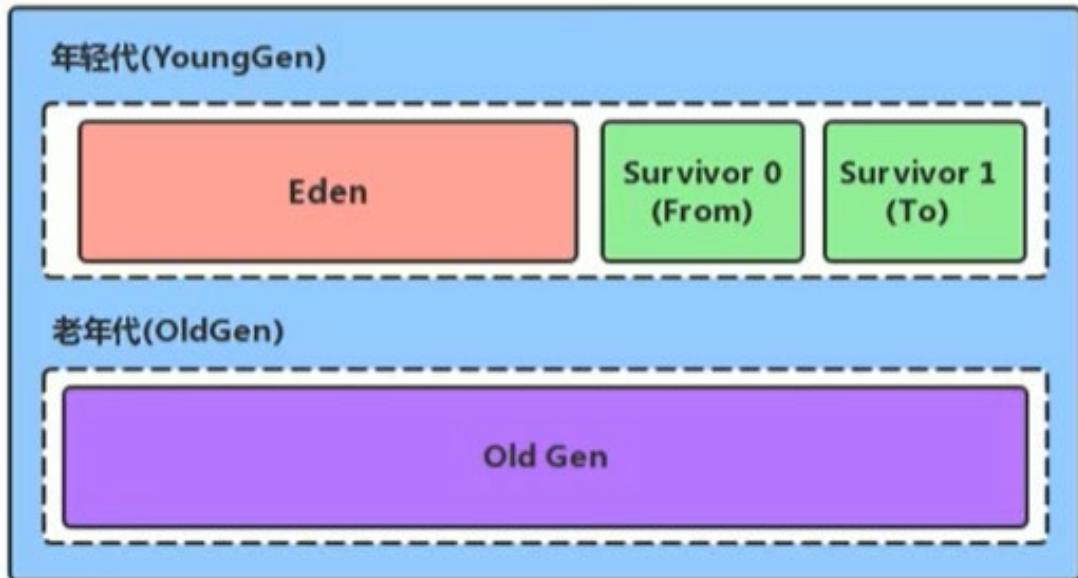
年轻代与老年代

存储在 JVM 中的 Java 对象可以被划分为两类：

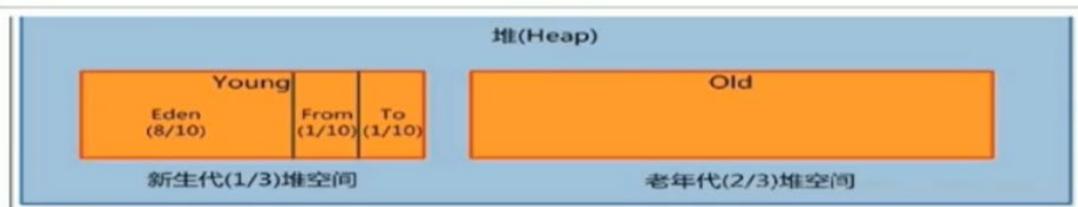
- 一类是生命周期较短的瞬时对象，这类对象的创建和消亡都非常迅速
 - 生命周期短的，及时回收即可
- 另外一类对象的生命周期却非常长，在某些极端的情况下还能够与 JVM 的生命周期保持一致

Java 堆区进一步细分的话，可以划分为年轻代（YoungGen）和老年代（oldGen）

其中年轻代又可以划分为 Eden 空间、Survivor0 空间和 Survivor1 空间（有时也叫做 from 区、to 区）



下面这参数开发中一般不会调：



- Eden: From: to -> 8:1:1
- 新生代: 老年代 -> 1 : 2

配置新生代与老年代在堆结构的占比。

- 默认-XX:NewRatio=2，表示新生代占 1，老年代占 2，新生代占整个堆的 1/3
- 可以修改-XX:NewRatio=4，表示新生代占 1，老年代占 4，新生代占整个堆的 1/5

当发现在整个项目中，生命周期长的对象偏多，那么就可以通过调整老年代的大小，来进行调优

在 HotSpot 中，Eden 空间和另外两个 survivor 空间缺省所占的比例是 8: 1: 1 当然开发人员可以通过选项“-xx:SurvivorRatio”调整这个空间比例。比如-
xx:SurvivorRatio=8

几乎所有的 Java 对象都是在 Eden 区被 new 出来的。绝大部分的 Java 对象的销毁都在新生代进行了。（有些大的对象在 Eden 区无法存储时候，将直接进入老年代）

IBM 公司的专门研究表明，新生代中 80% 的对象都是“朝生夕死”的。

可以使用选项"-Xmn"设置新生代最大内存大小

这个参数一般使用默认值就可以了。



图解对象分配过程

概念

为新对象分配内存是一件非常严谨和复杂的任务，JM 的设计者们不仅需要考虑内存如何分配、在哪里分配等问题，并且由于内存分配算法与内存回收算法密切相关，所以还需要考虑 GC 执行完内存回收后是否会在内存空间中产生内存碎片。

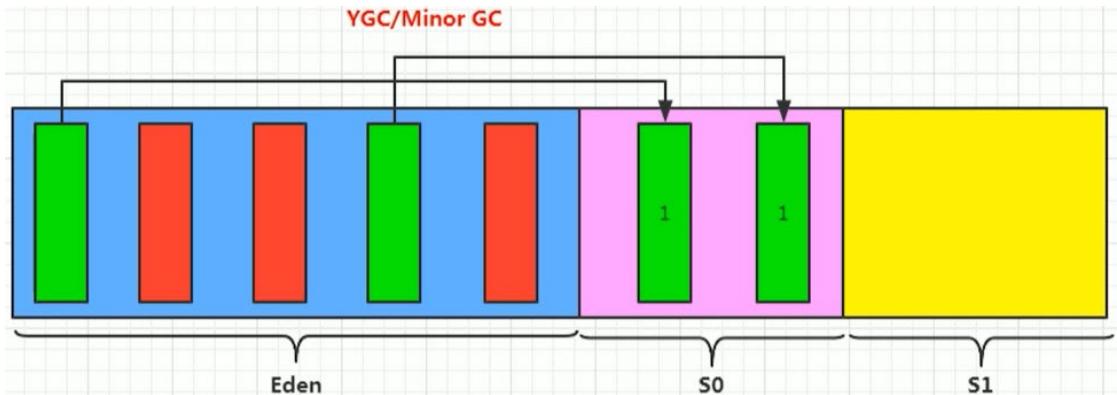
- new 的对象先放伊甸园区。此区有大小限制。
- 当伊甸园的空间填满时，程序又需要创建对象，JVM 的垃圾回收器将对伊甸园区进行垃圾回收（MinorGC），将伊甸园区中的不再被其他对象所引用的对象进行销毁。再加载新的对象放到伊甸园区
- 然后将伊甸园中的剩余对象移动到幸存者 0 区。
- 如果再次触发垃圾回收，此时上次幸存下来的放到幸存者 0 区的，如果没有回收，就会放到幸存者 1 区。
- 如果再次经历垃圾回收，此时会重新放回幸存者 0 区，接着再去幸存者 1 区。
- 啥时候能去养老区呢？可以设置次数。默认是 15 次。

- 在养老区，相对悠闲。当养老区内存不足时，再次触发 GC： Major GC，进行养老区的内存清理
- 若养老区执行了 Major GC 之后，发现依然无法进行对象的保存，就会产生 OOM 异常。

可以设置参数： -Xx:MaxTenuringThreshold= N 进行设置

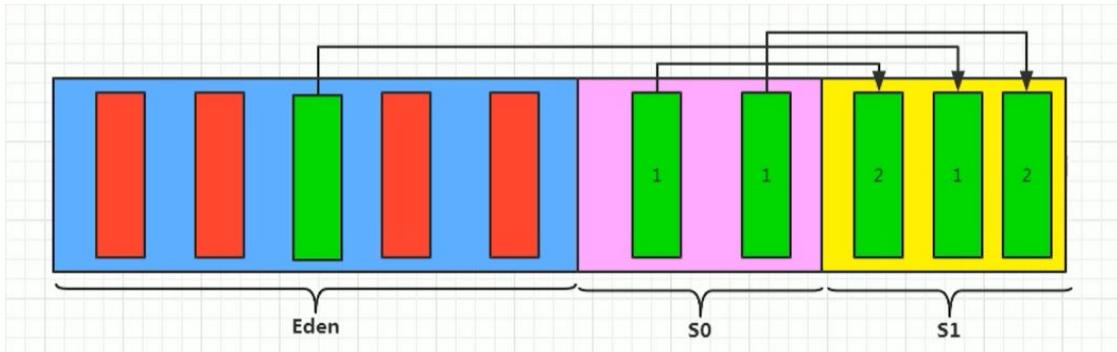
图解过程

我们创建的对象，一般都是存放在 Eden 区的，当我们 Eden 区满了后，就会触发 GC 操作，一般被称为 YGC / Minor GC 操作

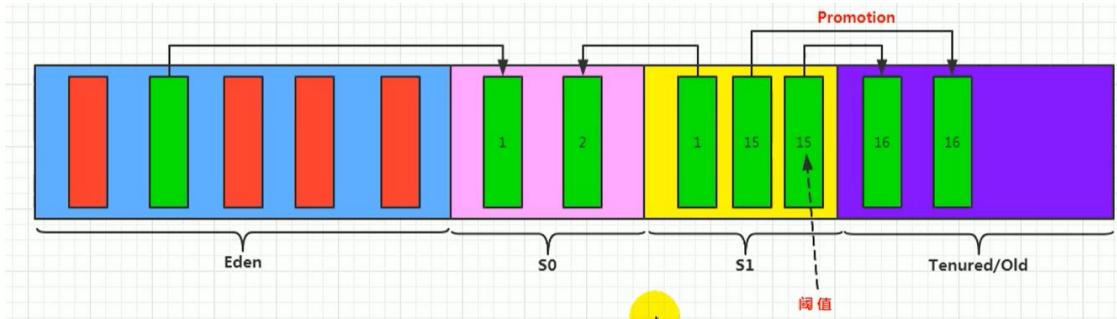


当我们进行一次垃圾收集后，红色的将会被回收，而绿色的还会被占用着，存放在 S0(Survivor From)区。同时我们给每个对象设置了一个年龄计数器，一次回收后就是 1。

同时 Eden 区继续存放对象，当 Eden 区再次存满的时候，又会触发一个 MinorGC 操作，此时 GC 将会把 Eden 和 Survivor From 中的对象 进行一次收集，把存活的对象放到 Survivor To 区，同时让年龄 + 1



我们继续不断的进行对象生成和垃圾回收，当 Survivor 中的对象的年龄达到 15 的时候，将会触发一次 Promotion 晋升的操作，也就是将年轻代中的对象晋升到老年代中



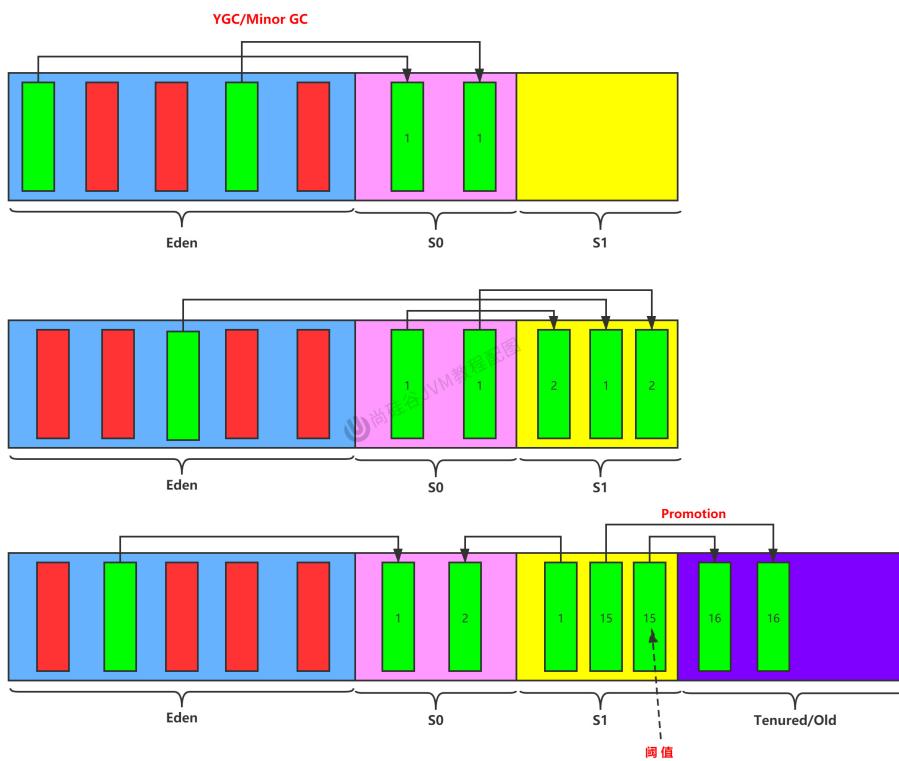
思考：幸存区满了后？

特别注意，在 Eden 区满了的时候，才会触发 MinorGC，而幸存者区满了后，不会触发 MinorGC 操作

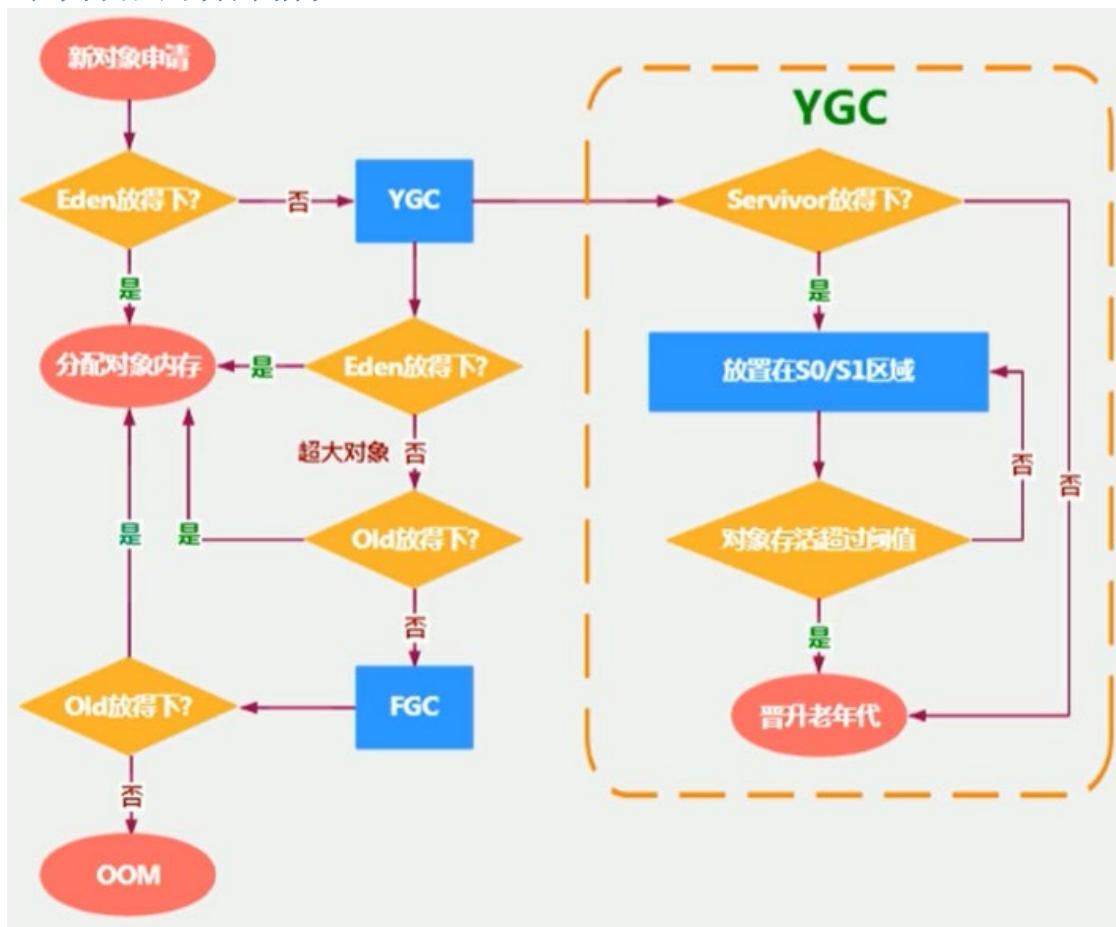
如果 Survivor 区满了后，将会触发一些特殊的规则，也就是可能直接晋升老年代

举例：以当兵为例，正常人的晋升可能是：新兵 -> 班长 -> 排长 -> 连长

但是也有可能有些人因为做了非常大的贡献，直接从 新兵 -> 排长



对象分配的特殊情况



代码演示对象分配过程

我们不断的创建大对象

```
/*
 * 代码演示对象创建过程
 *
 * @author: 陌溪
 * @create: 2020-07-07-07-9:16
 */
public class HeapInstanceTest {
    byte [] buffer = new byte[new Random().nextInt(1024 * 200)];
    public static void main(String[] args) throws InterruptedException
    {
        ArrayList<HeapInstanceTest> list = new ArrayList<>();
        把对象都放入List中阻止了垃圾回收器回收
        while (true) {
            list.add(new HeapInstanceTest());
            Thread.sleep(10);
        }
    }
}
```

```
        }  
    }  
}
```

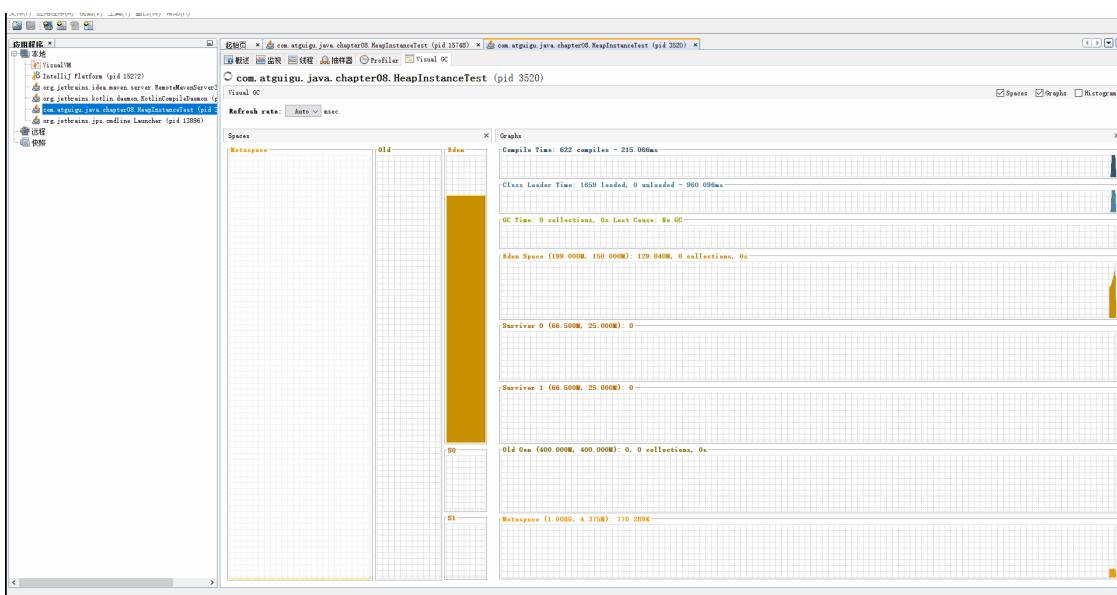
然后设置 JVM 参数

```
-Xms600m -Xmx600m
```

然后 cmd 输入下面命令，打开 VisualVM 图形化界面

```
jvisualvm
```

然后通过执行上面代码，通过 VisualGC 进行动态化查看



最终，在老年代和新生代都满了，就出现 OOM

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  
        at com.atguigu.java.chapter08.HeapInstanceTest.<init>(HeapInstanc  
eTest.java:13)  
        at com.atguigu.java.chapter08.HeapInstanceTest.main(HeapInstanc  
est.java:17)
```

常用的调优工具

- JDK 命令行
- Eclipse: Memory Analyzer Tool
- Jconsole
- Visual VM (实时监控 推荐~)

- Jprofiler (推荐~)
- Java Flight Recorder (实时监控)
- GCViewer
- GCEasy

总结

- 针对幸存者 s0, s1 区的总结：复制之后有交换，谁空谁是 to
- 关于垃圾回收：频繁在新生区收集，很少在老年代收集，几乎不再永久代和元空间进行收集
- 新生代采用复制算法的目的：是为了减少内碎片

Minor GC, MajorGC、Full GC

- Minor GC: 新生代的 GC
- Major GC: 老年代的 GC
- Full GC: 整堆收集，收集整个 Java 堆和方法区的垃圾收集

我们都知道，JVM 的调优的一个环节，也就是垃圾收集，我们需要尽量的避免垃圾回收，因为在垃圾回收的过程中，容易出现 STW 的问题

而 Major GC 和 Full GC 出现 STW 的时间，是 Minor GC 的 10 倍以上

JVM 在进行 GC 时，并非每次都对上面三个内存区域一起回收的，大部分时候回收的都是指新生代。针对 Hotspot VM 的实现，它里面的 GC 按照回收区域又分为两种类型：一种是部分收集（Partial GC），一种是整堆收集（Full GC）

部分收集：不是完整收集整个 Java 堆的垃圾收集。其中又分为：

- 新生代收集（MinorGC/YoungGC）：只是新生代的垃圾收集
- 老年代收集（MajorGC/oldGC）：只是老年代的圾收集。
 - 目前，只有 CMSGC 会有单独收集老年代的行为。
 - 注意，很多时候 Major GC 会和 Full GC 混淆使用，需要具体分辨是老年代回收还是整堆回收。
- 混合收集（MixedGC）：收集整个新生代以及部分老年代的垃圾收集。
 - 目前，只有 G1 GC 会有这种行为

整堆收集（FullGC）：收集整个 java 堆和方法区的垃圾收集。

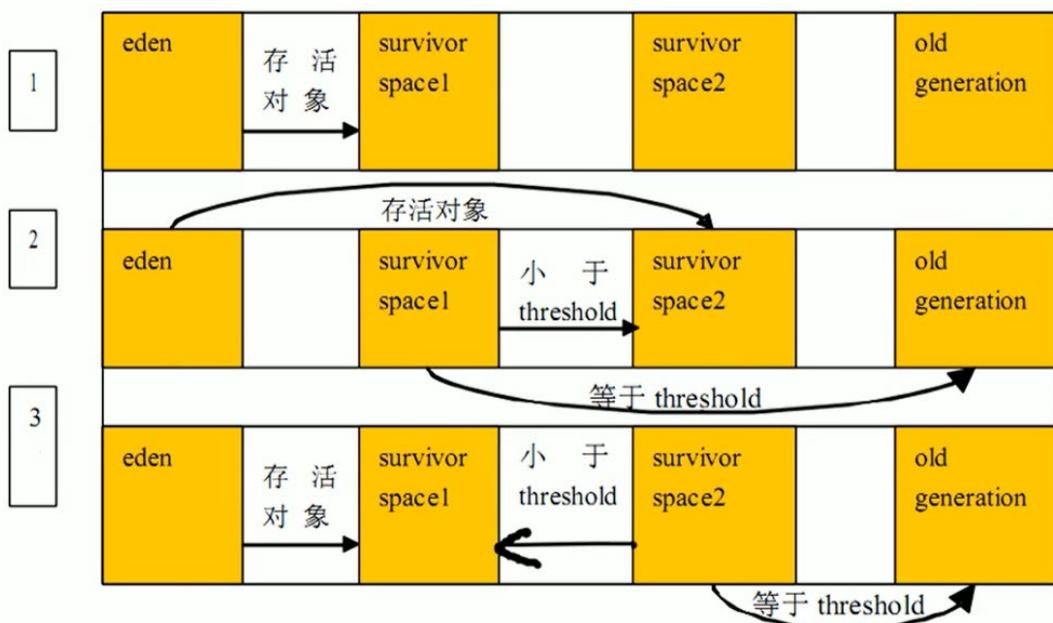
Minor GC

当年轻代空间不足时，就会触发 MinorGC，这里的年轻代满指的是 Eden 代满，Survivor 满不会引发 GC。（每次 Minor GC 会清理年轻代的内存。）

因为 Java 对象大多都具备 **朝生夕灭** 的特性，所以 Minor GC 非常频繁，一般回收速度也比较快。这一定义既清晰又易于理解。

Minor GC 会引发 STW，暂停其它用户的线程，等垃圾回收结束，用户线程才恢复运行

STW: stop the word



Major GC

指发生在老年代的 GC，对象从老年代消失时，我们说“Major Gc”或“Full GC”发生了

出现了 MajorGc，经常会伴随至少一次的 Minor GC（但非绝对的，在 Parallel Scavenge 收集器的收集策略里就有直接进行 MajorGC 的策略选择过程）

- 也就是在老年代空间不足时，会先尝试触发 MinorGc。如果之后空间还不足，则触发 Major GC

Major GC 的速度一般会比 MinorGc 慢 10倍以上，STW 的时间更长，如果 Major GC 后，内存还不足，就报 OOM 了

Full GC

触发 FullGC 执行的情况有如下五种：

- 调用 `System.gc()` 时，系统建议执行 FullGC，但是不必然执行
- 老年代空间不足
- 方法区空间不足
- 通过 Minor GC 后进入老年代的平均大小大于老年代的可用内存
- 由 Eden 区、survivor spacee (From Space) 区向 survivor space (To Space) 区复制时，对象大小大于 To Space 可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

说明：Full GC 是开发或调优中尽量要避免的。这样暂时时间会短一些

GC 举例

我们编写一个 OOM 的异常，因为我们在不断的创建字符串，是存放在元空间的

```
/*
 * GC 测试
 *
 * @author: 陌溪
 * @create: 2020-07-07-10:01
 */
public class GCTest {
    public static void main(String[] args) {
        int i = 0;
        try {
            List<String> list = new ArrayList<>();
            String a = "mogu blog";
            while(true) {
                list.add(a);
                a = a + a;
                i++;
            }
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

设置 JVM 启动参数

```
-Xms10m -Xmx10m -XX:+PrintGCDetails
```

打印出的日志

```
[GC (Allocation Failure) [PSYoungGen: 2038K->500K(2560K)] 2038K->797K(9728K), 0.3532002 secs] [Times: user=0.01 sys=0.00, real=0.36 secs]
[GC (Allocation Failure) [PSYoungGen: 2108K->480K(2560K)] 2405K->1565K(9728K), 0.0014069 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Ergonomics) [PSYoungGen: 2288K->0K(2560K)] [ParOldGen: 6845K->5281K(7168K)] 9133K->5281K(9728K), [Metaspace: 3482K->3482K(1056768K)], 0.0058675 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] 5281K->5281K(9728K), 0.0002857 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] [ParOldGen: 5281K->5263K(7168K)] 5281K->5263K(9728K), [Metaspace: 3482K->3482K(1056768K)], 0.0058564 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
Heap
    PSYoungGen      total 2560K, used 60K [0x00000000fffd00000, 0x00000000100000000, 0x00000000100000000)
        eden space 2048K, 2% used [0x00000000fffd00000, 0x00000000fffd0f138, 0x00000000fffd00000)
        from space 512K, 0% used [0x00000000ffff00000, 0x00000000ffff00000, 0x00000000ffff80000)
        to   space 512K, 0% used [0x00000000ffff80000, 0x00000000ffff80000, 0x00000000ffff80000)
    ParOldGen      total 7168K, used 5263K [0x00000000ff600000, 0x00000000fffd00000, 0x00000000fffd00000)
        object space 7168K, 73% used [0x00000000ff600000, 0x00000000fffb23cf0, 0x00000000fffd00000)
    Metaspace      used 3514K, capacity 4498K, committed 4864K, reserved 1056768K
        class space     used 388K, capacity 390K, committed 512K, reserved 1048576K

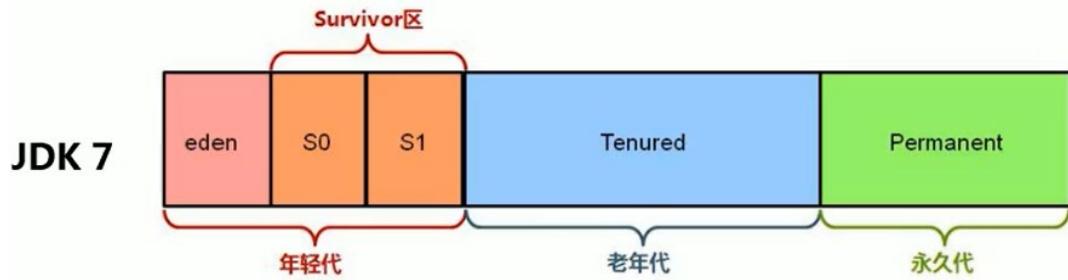
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOfRange(Arrays.java:3664)
    at java.lang.String.<init>(String.java:207)
    at java.lang.StringBuilder.toString(StringBuilder.java:407)
    at com.atguigu.java.chapter08.GCTest.main(GCTest.java:20)
```

触发 OOM 的时候，一定是进行了一次 Full GC，因为只有在老年代空间不足时候，才会爆出 OOM 异常

堆空间分代思想

为什么要把 Java 堆分代？不分代就不能正常工作了吗？经研究，不同对象的生命周期不同。70%-99%的对象是临时对象。

新生代：有 Eden、两块大小相同的 survivor（又称为 from/to, s0/s1）构成，to 总为空。老年代：存放新生代中经历多次 GC 仍然存活的对象。



其实不分代完全可以，分代的唯一理由就是优化 GC 性能。如果没有分代，那所有的对象都在一块，就如同把一个学校的人都关在一个教室。GC 的时候要找到哪些对象没用，这样就会对堆的所有区域进行扫描。而很多对象都是朝生夕死的，如果分代的话，把新创建的对象放到某一地方，当 GC 的时候先把这块存储“朝生夕死”对象的区域进行回收，这样就会腾出很大的空间出来。



内存分配策略

如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 survivor 空间中，并将对象年龄设为 1。对象在 survivor 区中每熬过一次 MinorGC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁，其实每个 JVM、每个 GC 都有所不同）时，就会被晋升到老年代。

对象晋升老年代的年龄阀值，可以通过选项-xx:MaxTenuringThreshold 来设置。

针对不同年龄段的对象分配原则如下所示：

- 优先分配到 Eden

- 开发中比较长的字符串或者数组，会直接存在老年代，但是因为新创建的对象都是朝生夕死的，所以这个大对象可能也很快被回收，但是因为老年代触发 Major GC 的次数比 Minor GC 要更少，因此可能回收起来就会比较慢
- 大对象直接分配到老年代
 - 尽量避免程序中出现过多的大对象
- 长期存活的对象分配到老年代
- 动态对象年龄判断
 - 如果 survivor 区中相同年龄的所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代，无须等到 MaxTenuringThreshold 中要求的年龄。

空间分配担保: -Xx:HandlePromotionFailure

- 也就是经过 Minor GC 后，所有的对象都存活，因为 Survivor 比较小，所以就需要将 Survivor 无法容纳的对象，存放到老年代中。

为对象分配内存: TLAB

问题: 堆空间都是共享的么?

不一定，因为还有 TLAB 这个概念，在堆中划分出一块区域，为每个线程所独占

为什么有 TLAB?

TLAB: Thread Local Allocation Buffer，也就是为每个线程单独分配了一个缓冲区
堆区是线程共享区域，任何线程都可以访问到堆区中的共享数据

由于对象实例的创建在 JVM 中非常频繁，因此在并发环境下从堆区中划分内存空间是线程不安全的

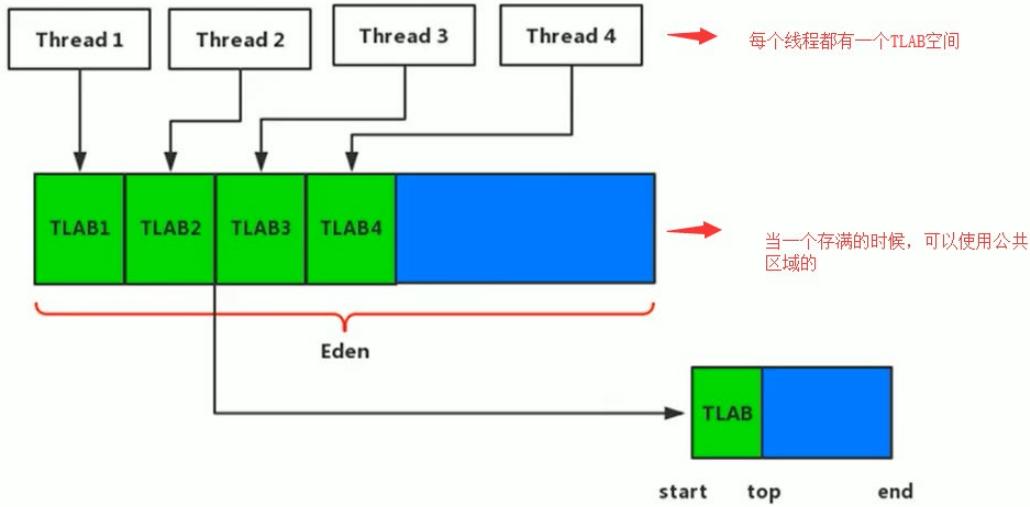
为避免多个线程操作同一地址，需要使用加锁等机制，进而影响分配速度。

什么是 TLAB

从内存模型而不是垃圾收集的角度，对 Eden 区域继续进行划分，JVM 为每个线程分配了一个私有缓存区域，它包含在 Eden 空间内。

多线程同时分配内存时，使用 TLAB 可以避免一系列的非线程安全问题，同时还能够提升内存分配的吞吐量，因此我们可以将这种内存分配方式称之为快速分配策略。

据我所知所有 OpenJDK 衍生出来的 JVM 都提供了 TLAB 的设计。



尽管不是所有的对象实例都能够再 TLAB 中成功分配内存，但 JVM 确实是将 TLAB 作为内存分配的首选。

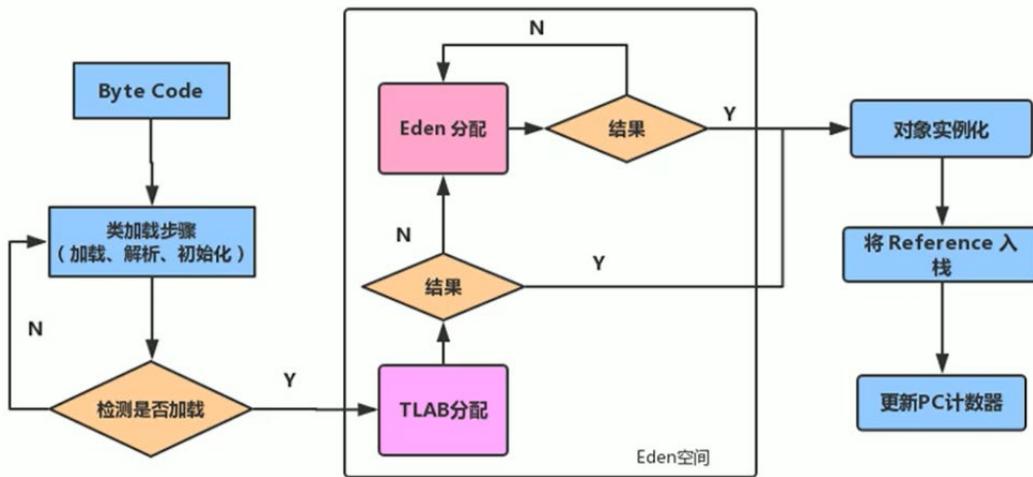
在程序中，开发人员可以通过选项“-Xx:UseTLAB”设置是否开启 TLAB 空间。

默认情况下，TLAB 空间的内存非常小，仅占有整个 Eden 空间的 1%，当然我们可以通过选项“-Xx:TLABWasteTargetPercent”设置 TLAB 空间所占用 Eden 空间的百分比大小。

一旦对象在 TLAB 空间分配内存失败时，JVM 就会尝试着通过使用加锁机制确保数据操作的原子性，从而直接在 Eden 空间中分配内存。

TLAB 分配过程

对象首先是通过 TLAB 开辟空间，如果不能放入，那么需要通过 Eden 来进行分配



小结：堆空间的参数设置

- `-XX: +PrintFlagsInitial`: 查看所有的参数的默认初始值
- `-XX: +PrintFlagsFinal`: 查看所有的参数的最终值（可能会存在修改，不再是初始值）
- `-Xms`: 初始堆空间内存（默认为物理内存的 1/64）
- `-Xmx`: 最大堆空间内存（默认为物理内存的 1/4）
- `-Xmn`: 设置新生代的大小。（初始值及最大值）
- `-XX:NewRatio`: 配置新生代与老年代在堆结构的占比
Survivor区域太大，会导致YGC失去意义
Survivor区域太小，会导致YGC太过频繁
- `-XX:SurvivorRatio`: 设置新生代中 Eden 和 S0/S1 空间的比例
- `-XX:MaxTenuringThreshold`: 设置新生代垃圾的最大年龄
- `-XX: +PrintGCDetails`: 输出详细的 GC 处理日志
 - 打印 gc 简要信息: ①`-Xx: +PrintGC` ②`-verbose:gc`
- `-XX:HandlePromotionFailure`: 是否设置空间分配担保

在发生 Minor GC 之前，虚拟机会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，则此次 Minor GC 是安全的

- 如果小于，则虚拟机会查看-xx:HandlePromotionFailure 设置值是否允担保失败。
 - 如果 HandlePromotionFailure=true，那么会继续检查老年代最大可用连续空间是否大于历次晋升到老年代的对像的平均大小。
 - 如果大于，则尝试进行一次 Minor GC，但这次 Minor GC 依然有风险的；
 - 如果小于，则改为进行一次 Full GC。
 - 如果 HandlePromotionFailure=false，则改为进行一次 Full GC。

在 JDK6 Update24 之后，HandlePromotionFailure 参数不会再影响到虚拟机的空间分配担保策略，观察 openJDK 中的源码变化，虽然源码中还定义了 HandlePromotionFailure 参数，但是在代码中已经不会再使用它。JDK6 Update 24 之后的规则变为只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小就会进行 Minor GC，否则将进行 FullGC。

堆是分配对象的唯一选择么？

逃逸分析

在《深入理解 Java 虚拟机》中关于 Java 堆内存有这样一段描述：

随着 JIT 编译期的发展与逃逸分析技术逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化，所有的对象都分配到堆上也渐渐变得不那么“绝对”了。

在 Java 虚拟机中，对象是在 Java 堆中分配内存的，这是一个普遍的常识。但是，有一种特殊情况，那就是如果经过 **逃逸分析** (Escape Analysis) 后发现，一个对象并没有逃逸出方法的话，那么就可能被优化成栈上分配。这样就无需在堆上分配内存，也无须进行垃圾回收了。这也是最常见的堆外存储技术。

此外，前面提到的基于 openJdk 深度定制的 TaoBaoVM，其中创新的 GCIH (GC invisible heap) 技术实现 off-heap，将生命周期较长的 Java 对象从 heap 中移至 heap 外，并且 GC 不能管理 GCIH 内部的 Java 对象，以此达到降低 GC 的回收频率和提升 GC 的回收效率的目的。

如何将堆上的对象分配到栈，需要使用逃逸分析手段。

这是一种可以有效减少 Java 程序中同步负载和内存堆分配压力的跨函数全局数据流分析算法。通过逃逸分析，Java Hotspot 编译器能够分析出一个新的对象的引用的使用范围从而决定是否要将这个对象分配到堆上。逃逸分析的基本行为就是分析对象动态作用域：

没有发生逃逸，就在栈上分配

- 当一个对象在方法中被定义后，对象只在方法内部使用，则认为没有发生逃逸。

- 当一个对象在方法中被定义后，它被外部方法所引用，则认为发生逃逸。例如作为调用参数传递到其他地方中。

逃逸分析举例

没有发生逃逸的对象，则可以分配到栈上，随着方法执行的结束，栈空间就被移除，每个栈里面包含了很多栈帧，也就是发生逃逸分析

```
public void my_method() {
    V v = new V();
    // use v
    // ....
    v = null;
}
```

针对下面的代码

```
public static StringBuffer createStringBuffer(String s1, String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb;
}
```

如果想要 StringBuffer sb 不发生逃逸，可以这样写

```
public static String createStringBuffer(String s1, String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb.toString();
}
```

完整的逃逸分析代码举例

```
/**
 * 逃逸分析
 * 如何快速的判断是否发生了逃逸分析，就看 new 的对象是否在方法外被调用。
 * @author: 陌溪
 * @create: 2020-07-07-20:05
 */
public class EscapeAnalysis {

    public EscapeAnalysis obj;

    /**
     * 方法返回 EscapeAnalysis 对象，发生逃逸
     * @return
     */
    public EscapeAnalysis getInstance() {
```

```

        return obj == null ? new EscapeAnalysis():obj;
    }

    /**
     * 为成员属性赋值，发生逃逸
     */
    public void setObj() {
        this.obj = new EscapeAnalysis();
    }

    /**
     * 对象的作用于仅在当前方法中有效，没有发生逃逸
     */
    public void useEscapeAnalysis() {
        EscapeAnalysis e = new EscapeAnalysis();
    }

    /**
     * 引用成员变量的值，发生逃逸
     */
    public void useEscapeAnalysis2() {
        EscapeAnalysis e = getInstance();      发生逃逸
        // getInstance().XXX  发生逃逸
    }
}

```

参数设置

在 JDK 1.7 版本之后，HotSpot 中默认就已经开启了逃逸分析

如果使用的是较早的版本，开发人员则可以通过：

- 选项“-xx: +DoEscapeAnalysis”显式开启逃逸分析
- 通过选项“-xx: +PrintEscapeAnalysis”查看逃逸分析的筛选结果

结论

开发中能使用局部变量的，就不要使用在方法外定义。

使用逃逸分析，编译器可以对代码做如下优化： 栈上分配+同步省略+标量替换

- **栈上分配**：将堆分配转化为栈分配。如果一个对象在子程序中被分配，要使指向该对象的指针永远不会发生逃逸，对象可能是栈上分配的候选，而不是堆上分配
- **同步省略**：如果一个对象被发现只有一个线程被访问到，那么对于这个对象的操作可以不考虑同步。

- 分离对象或标量替换：有的对象可能不需要作为一个连续的内存结构存在也可以被访问到，那么对象的部分（或全部）可以不存储在内存，而是存储在CPU寄存器中。

栈上分配

JIT编译器在编译期间根据逃逸分析的结果，发现如果一个对象并没有逃逸出方法的话，就可能被优化成栈上分配。分配完成后，继续在调用栈内执行，最后线程结束，栈空间被回收，局部变量对象也被回收。这样就无须进行垃圾回收了。

常见的栈上分配的场景

在逃逸分析中，已经说明了。分别是给成员变量赋值、方法返回值、实例引用传递。

举例

我们通过举例来说明开启逃逸分析和未开启逃逸分析时候的情况

```
/**
 * 栈上分配
 * -Xmx1G -Xms1G -XX:-DoEscapeAnalysis -XX:+PrintGCDetails
 * @author: 陌溪
 * @create: 2020-07-07-20:23
 */
class User {
    private String name;
    private String age;
    private String gender;
    private String phone;
}
public class StackAllocation {
    public static void main(String[] args) throws InterruptedException
    {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 100000000; i++) {
            alloc();
        }
        long end = System.currentTimeMillis();
        System.out.println("花费的时间为: " + (end - start) + " ms");

        // 为了方便查看堆内存中对象个数，线程sleep
        Thread.sleep(10000000);
    }

    private static void alloc() {
        // 未发生逃逸
        User user = new User();
    }
}
```

```
}
```

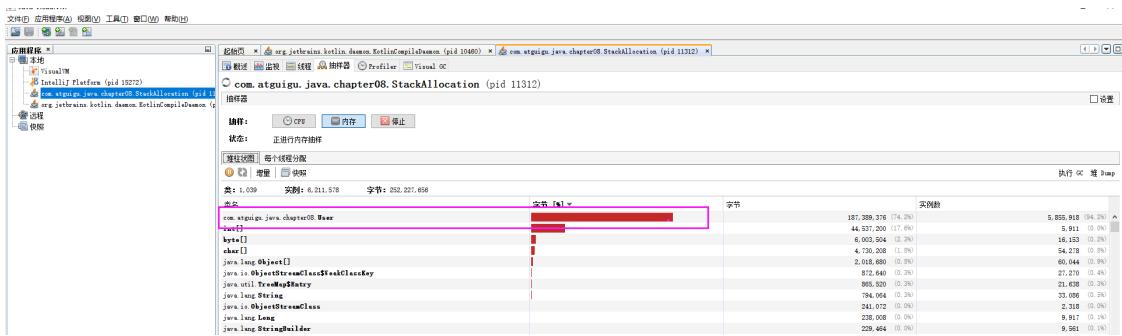
设置 JVM 参数，表示未开启逃逸分析

```
-Xmx1G -Xms1G -XX:-DoEscapeAnalysis -XX:+PrintGCDetails
```

运行结果，同时还触发了 GC 操作

花费的时间为： 664 ms

然后查看内存的情况，发现有大量的 User 存储在堆中



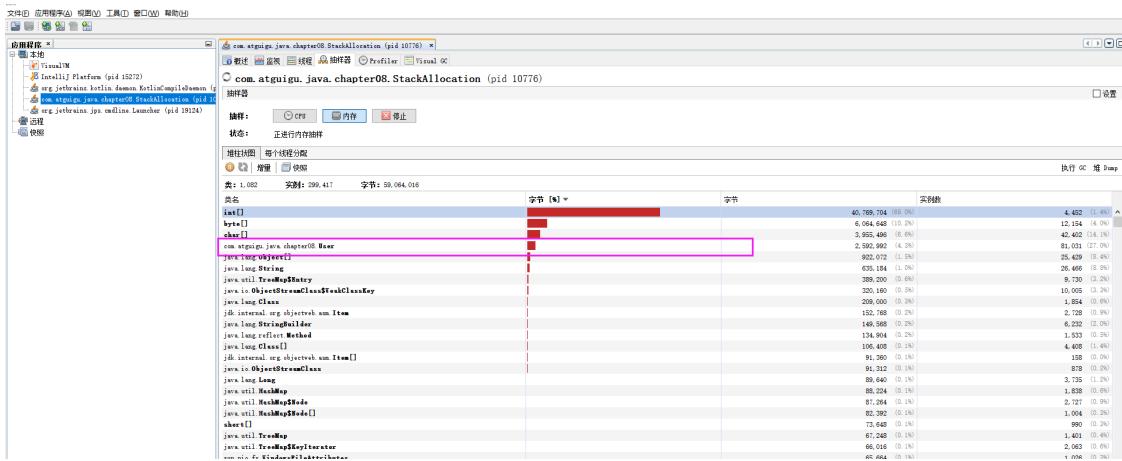
我们在开启逃逸分析

```
-Xmx1G -Xms1G -XX:+DoEscapeAnalysis -XX:+PrintGCDetails
```

然后查看运行时间，我们能够发现花费的时间快速减少，同时不会发生 GC 操作

花费的时间为： 5 ms

再看内存情况，我们发现只有很少的 User 对象，说明 User 未发生逃逸，因为它存储在栈中，随着栈的销毁而消失



同步省略

线程同步的代价是相当高的，同步的后果是降低并发性和性能。

在动态编译同步块的时候，JIT 编译器可以借助逃逸分析来判断同步块所使用的锁对象是否只能够被一个线程访问而没有被发布到其他线程。如果没有，那么 JIT 编译器在编译这个同步块的时候就会取消对这部分代码的同步。这样就能大大提高并发性和性能。这个取消同步的过程就叫同步省略，也叫锁消除。

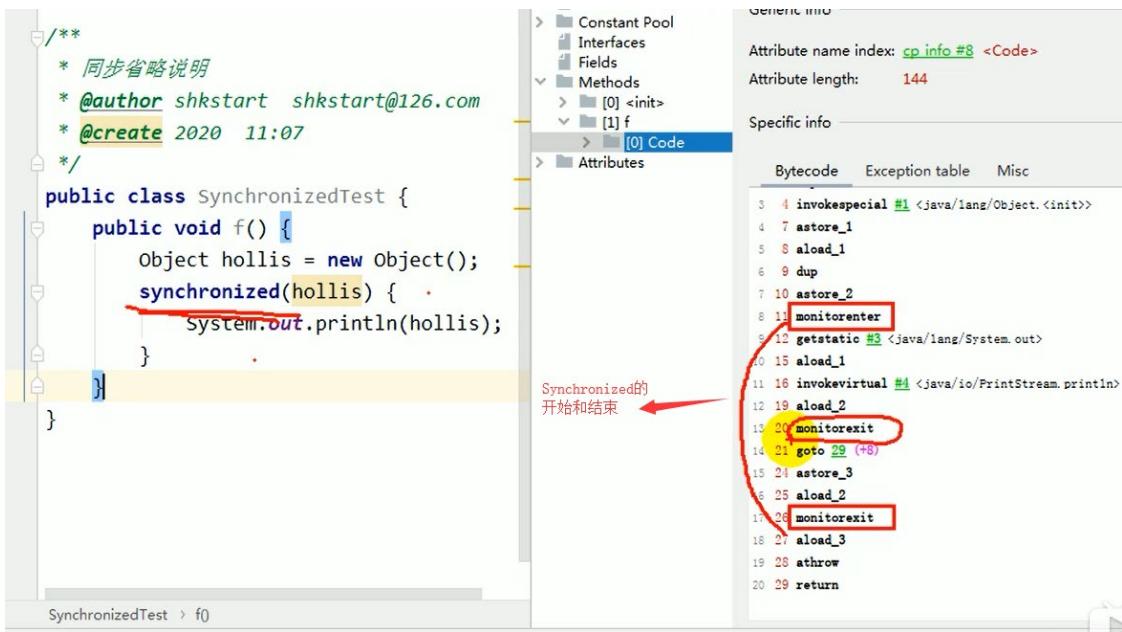
例如下面的代码

```
public void f() {
    Object hellis = new Object();
    synchronized(hillis) {
        System.out.println(hillis);
    }
}
```

代码中对 hellis 这个对象加锁，但是 hellis 对象的生命周期只在 f() 方法中，并不会被其他线程所访问到，所以在 JIT 编译阶段就会被优化掉，优化成：

```
public void f() {
    Object hellis = new Object();
    System.out.println(hillis);
}
```

我们将其转换成字节码



分离对象和标量替换 有的对象可能不需要作为一个连续的内存结构存在也能被访问到，那么对象的部分或全部可以不存储在内存（堆），而是存储到CPU寄存器（栈）中

标量（scalar）是指一个无法再分解成更小的数据的数据。Java 中的原始数据类型就是标量。

相对的，那些还可以分解的数据叫做聚合量（Aggregate），Java 中的对象就是聚合量，因为他可以分解成其他聚合量和标量。

在 JIT 阶段，如果经过逃逸分析，发现一个对象不会被外界访问的话，那么经过 JIT 优化，就会把这个对象拆解成若干个其中包含的若干个成员变量来代替。这个过程就是标量替换。

```
public static void main(String args[]) {
    alloc();
}
class Point {
    private int x;
    private int y;
}
private static void alloc() {
    Point point = new Point(1,2);
    System.out.println("point.x" + point.x + ";point.y" + point.y);
}
```

以上代码，经过标量替换后，就会变成

```
private static void alloc() {
    int x = 1;
```

```
    int y = 2;
    System.out.println("point.x = " + x + "; point.y=" + y);
}
```

可以看到，`Point` 这个聚合量经过逃逸分析后，发现他并没有逃逸，就被替换成两个标量了。那么标量替换有什么好处呢？就是可以大大减少堆内存的占用。因为一旦不需要创建对象了，那么就不再需要分配堆内存了。标量替换为栈上分配提供了很好的基础。

代码优化之标量替换

上述代码在主函数中进行了 1 亿次 alloc。调用进行对象创建，由于 `User` 对象实例需要占据约 16 字节的空间，因此累计分配空间达到将近 1.5GB。如果堆空间小于这个值，就必然会发生 GC。使用如下参数运行上述代码：

```
-server -Xmx100m -Xms100m -XX:+DoEscapeAnalysis -XX:+PrintGC -XX:+EliminateAllocations
```

这里设置参数如下：

- 参数-server：启动 Server 模式，因为在 server 模式下，才可以启用逃逸分析。
- 参数-XX:+DoEscapeAnalysis：启用逃逸分析
- 参数-Xmx10m：指定了堆空间最大为 10MB
- 参数-XX:+PrintGC：将打印 Gc 日志
- 参数-XX:+EliminateAllocations：开启了标量替换（默认打开），允许将对象打散分配在栈上，比如对象拥有 id 和 name 两个字段，那么这两个字段将被视为两个独立的局部变量进行分配

逃逸分析的不足

关于逃逸分析的论文在 1999 年就已经发表了，但直到 JDK1.6 才有实现，而且这项技术到如今也并不是十分成熟。

其根本原因就是无法保证逃逸分析的性能消耗一定能高于他的消耗。虽然经过逃逸分析可以做标量替换、栈上分配、和锁消除。但是逃逸分析自身也是需要进行一系列复杂的分析的，这其实也是一个相对耗时的过程。一个极端的例子，就是经过逃逸分析之后，发现没有一个对象是不逃逸的。那这个逃逸分析的过程就白白浪费掉了。

虽然这项技术并不十分成熟，但是它也是即时编译器优化技术中一个十分重要的手段。注意到有一些观点，认为通过逃逸分析，JVM 会在栈上分配那些不会逃逸的对象，这在理论上是可行的，但是取决于 JVM 设计者的选择。据我所知，oracle

Hotspot JVM 中并未这么做，这一点在逃逸分析相关的文档里已经说明，所以可以明确所有的对象实例都是创建在堆上。

目前很多书籍还是基于 JDK7 以前的版本，JDK 已经发生了很大变化，`intern` 字符串的缓存和静态变量曾经都被分配在永久代上，而永久代已经被元数据区取代。但是，`intern` 字符串缓存和静态变量并不是被转移到元数据区，而是直接在堆上分配，所以这一点同样符合前面一点的结论：对象实例都是分配在堆上。

小结

年轻代是对象的诞生、成长、消亡的区域，一个对象在这里产生、应用，最后被垃圾回收器收集、结束生命。

老年代放置长生命周期的对象，通常都是从 survivor 区域筛选拷贝过来的 Java 对象。当然，也有特殊情况，我们知道普通的对象会被分配在 TLAB 上；如果对象较大，JVM 会试图直接分配在 Eden 其他位置上；如果对象太大，完全无法在新生代找到足够长的连续空闲空间，JVM 就会直接分配到老年代。当 GC 只发生在年轻代中，回收年轻代对象的行为被称为 MinorGc。

当 GC 发生在老年代时则被称为 MajorGc 或者 FullGC。一般的，MinorGc 的发生频率要比 MajorGC 高很多，即老年代中垃圾回收发生的频率将大大低于年轻代。

方法区

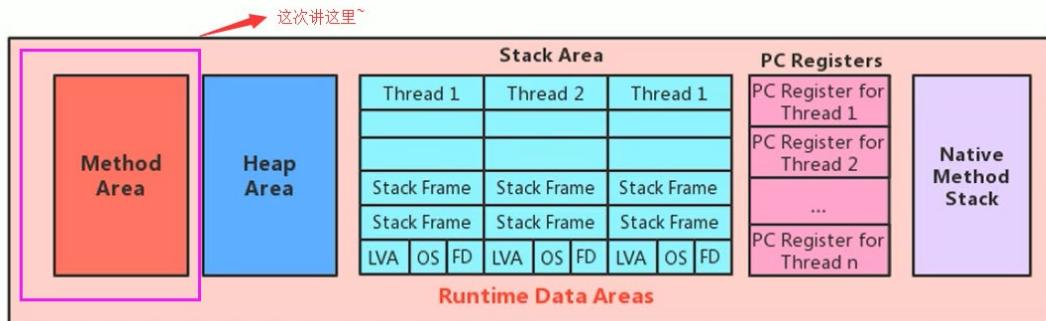
永久代和元空间是HotSpot虚拟机中对JVM规范中方法区的实现，而永久代和元空间的区别在JVM不同版本中对方法区的实现。

前言

在JDK1.7之前：永久代是方法区的实现，存放了运行时常量池，包含字符串常量池，静态变量等。

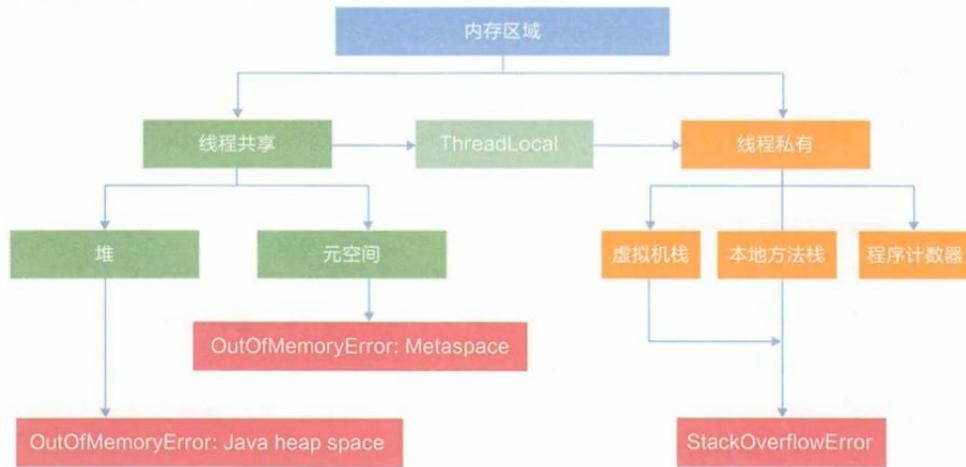
在JDK1.7：永久代是方法区的实现，存放的字符串常量池、静态变量等移出至堆内存。运行时常量池等剩下的东西还在永久代（方法区）。

这次所讲述的是运行时数据区的最后一个部分



从线程共享与否的角度来看

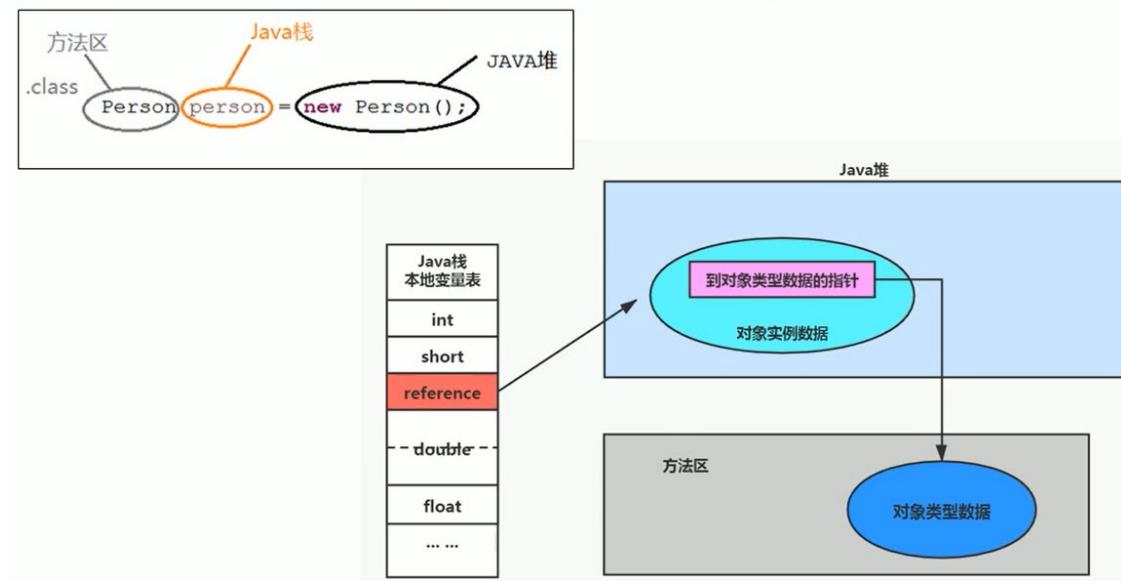
从线程共享与否的角度来看



ThreadLocal: 如何保证多个线程在并发环境下的安全性? 典型应用就是数据库连接管理, 以及会话管理

栈、堆、方法区的交互关系

下面就涉及了对象的访问定位



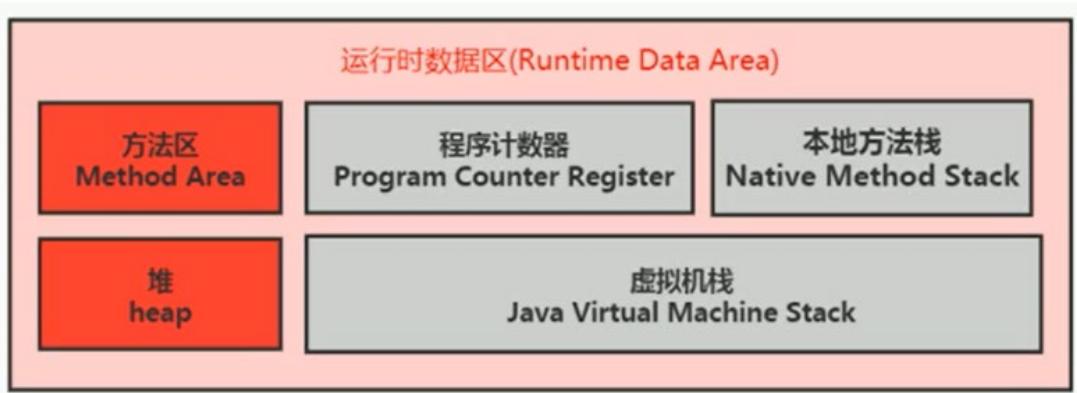
- Person: 存放在元空间, 也可以说方法区
- person: 存放在 Java 栈的局部变量表中

- new Person(): 存放在 Java 堆中

方法区的理解

《Java 虚拟机规范》中明确说明：“尽管所有的方法区在逻辑上是属于堆的一部分，但一些简单的实现可能不会选择去进行垃圾收集或者进行压缩。”但对于 HotSpotJVM 而言，方法区还有一个别名叫做 Non-Heap（非堆），目的就是要和堆分开。

所以，方法区看作是一块独立于 Java 堆的内存空间。



方法区主要存放的是 Class，而堆中主要存放的是实例化的对象

- 方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域。
- 方法区在 JVM 启动的时候被创建，并且它的实际的物理内存空间中和 Java 堆区一样都可以是不连续的。
- 方法区的大小，跟堆空间一样，可以选择固定大小或者可扩展。
- 方法区的大小决定了系统可以保存多少个类，如果系统定义了太多的类，导致方法区溢出，虚拟机同样会抛出内存溢出错误：java.lang.OutOfMemoryError: PermGen space 或者 java.lang.OutOfMemoryError:Metaspace
 - 加载大量的第三方的 jar 包
 - Tomcat 部署的工程过多（30~50 个）
 - 大量动态的生成反射类
- 关闭 JVM 就会释放这个区域的内存。

HotSpot 中方法区的演进

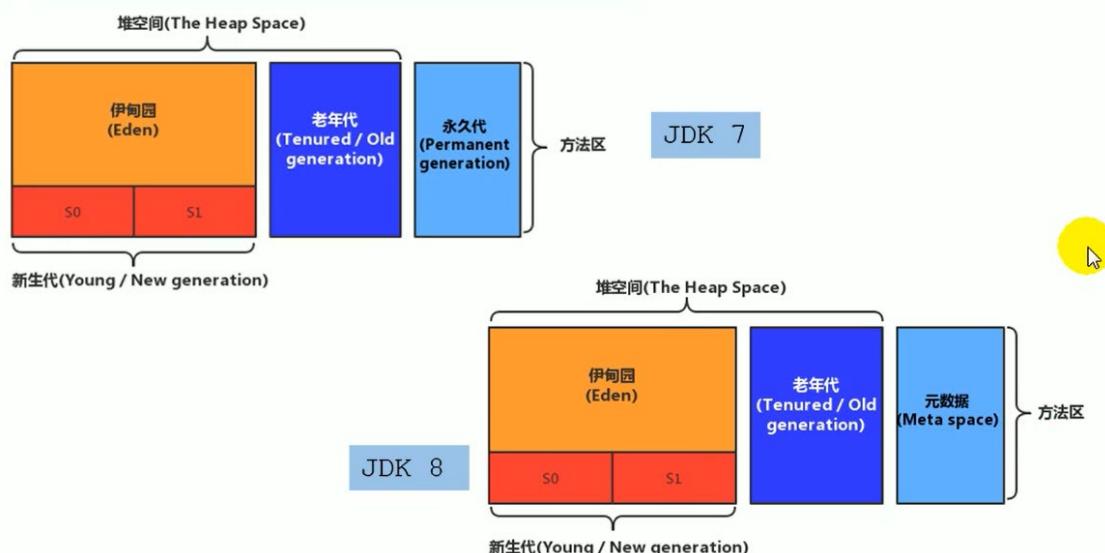
在 jdk7 及以前，习惯上把方法区，称为永久代。jdk8 开始，使用元空间取代了永久代。

- JDK 1.8 后，元空间存放在堆外内存中

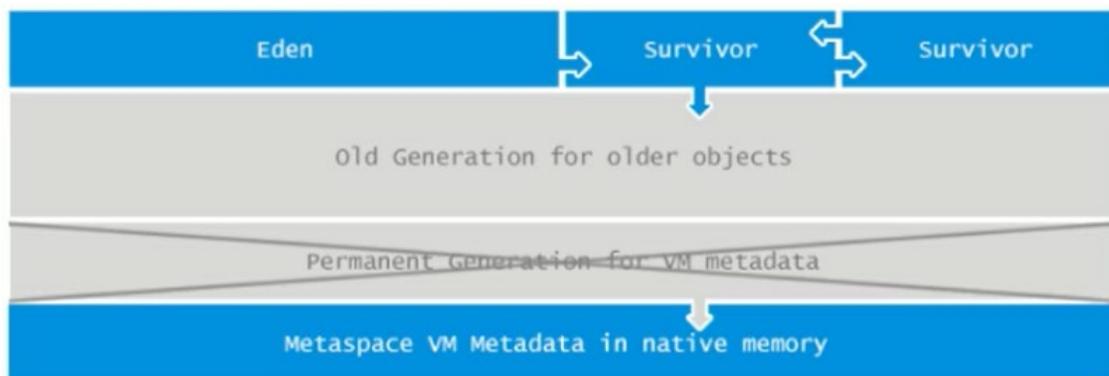
对于Hotspot来说，方法区和永久代等价

本质上，方法区和永久代并不等价。仅是对 hotspot 而言的。《Java 虚拟机规范》对如何实现方法区，不做统一要求。例如：BEA JRockit / IBM J9 中不存在永久代的概念。

现在来看，当年使用永久代，不是好的 idea。导致 Java 程序更容易 oom（超过-XX:MaxPermSize 上限）



而到了 JDK8，终于完全废弃了永久代的概念，改用与 JRockit、J9 一样在本地内存中实现的元空间（Metaspace）来代替



元空间的本质和永久代类似，都是对 JVM 规范中方法区的实现。不过元空间与永久代最大的区别在于：元空间不在虚拟机设置的内存中，而是使用本地内存

永久代、元空间二者并不只是名字变了，内部结构也调整了

根据《Java 虚拟机规范》的规定，如果方法区无法满足新的内存分配需求时，将抛出 OOM 异常

设置方法区大小与 OOM

方法区的大小不必是固定的，JVM 可以根据应用的需要动态调整。

jdk7 及以前

- 通过-xx:Permsize 来设置永久代初始分配空间。默认值是 20.75M
- -XX:MaxPermsize 来设定永久代最大可分配空间。32 位机器默认是 64M，64 位机器模式是 82M
- 当 JVM 加载的类信息容量超过了这个值，会报异常 OutofMemoryError:PermGen space。

```
C:\Users\Administrator>jps  
7940  
8856 StaticFieldTest  
12252 Jps  
12268 Launcher  
  
C:\Users\Administrator>jinfo -flag PermSize 8856  
-XX:PermSize=21757952  
  
C:\Users\Administrator>jinfo -flag MaxPermSize 8856  
-XX:MaxPermSize=85983232
```

JDK8 以后

元数据区大小可以使用参数 -XX:MetaspaceSize 和 -XX:MaxMetaspaceSize 指定

默认值依赖于平台。windows 下，-XX:MetaspaceSize 是 21M，-XX:MaxMetaspaceSize 的值是-1，即没有限制。

与永久代不同，如果不指定大小，默认情况下，虚拟机会耗尽所有的可用系统内存。如果元数据区发生溢出，虚拟机一样会抛出异常 OutOfMemoryError:Metaspace

-XX:MetaspaceSize：设置初始的元空间大小。对于一个 64 位的服务器端 JVM 来说，其默认的-xx:MetaspaceSize 值为 21MB。这就是初始的高水位线，一旦触及这个水

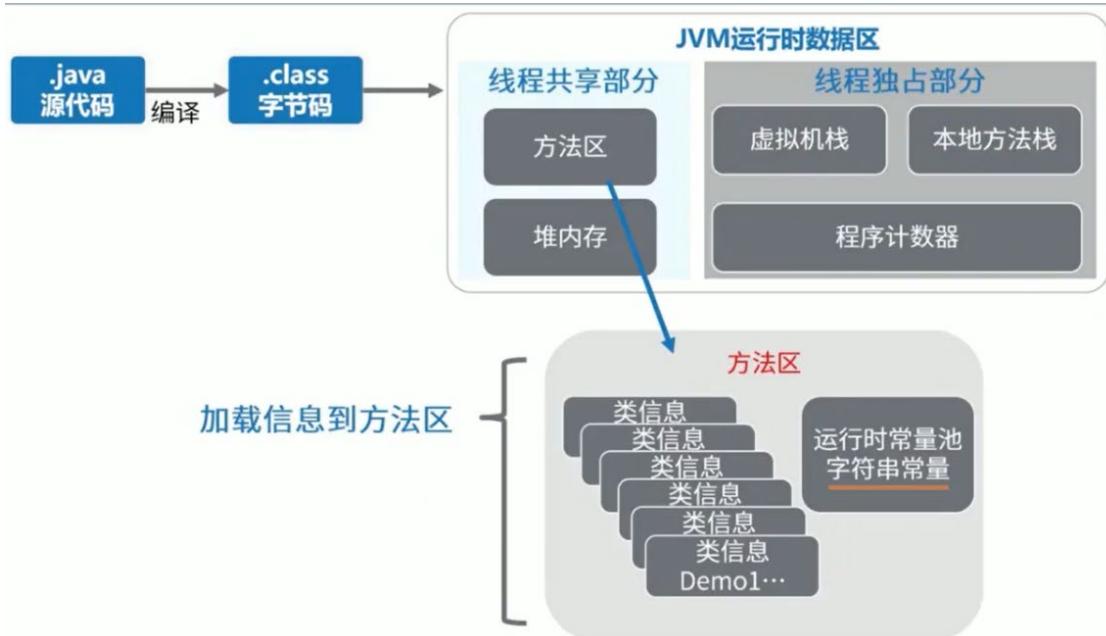
位线，FullGC 将会被触发并卸载没用的类（即这些类对应的类加载器不再存活）然后这个高水位线将会重置。新的高水位线的值取决于 GC 后释放了多少元空间。如果释放的空间不足，那么在不超过 MaxMetaspaceSize 时，适当提高该值。如果释放空间过多，则适当降低该值。

如果初始化的高水位线设置过低，上述高水位线调整情况会发生很多次。通过垃圾回收器的日志可以观察到 FullGC 多次调用。为了避免频繁地 GC，建议将-XX:MetaspaceSize 设置为一个相对较高的值。

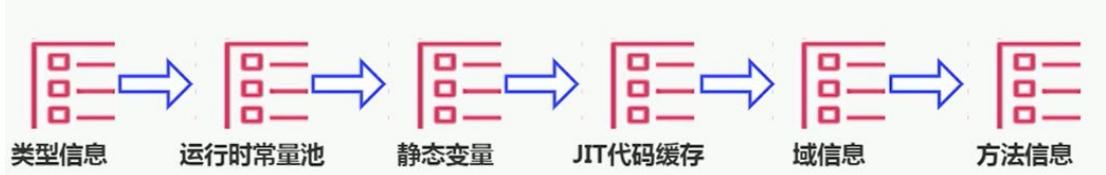
如何解决这些 OOM

- 要解决 OOM 异常或 heap space 的异常，一般的手段是首先通过内存映像分析工具（如 Eclipse Memory Analyzer）对 dump 出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了内存泄漏（Memory Leak）还是内存溢出（Memory Overflow）泄露是因为没办法对对象进行回收，是导致溢出的原因之一
 - 内存泄漏就是有大量的引用指向某些对象，但是这些对象以后不会使用了，但是因为它们还和 GC ROOT 有关联，所以导致以后这些对象也不会被回收，这就是内存泄漏的问题
- 如果是内存泄漏，可进一步通过工具查看泄漏对象到 GC Roots 的引用链。于是就能找到泄漏对象是通过怎样的路径与 GCRoots 相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息，以及 GCRoots 引用链的信息，就可以比较准确地定位出泄漏代码的位置。
- 如果不存在内存泄漏，换句话说就是内存中的对象确实都还必须存活者，那就应当检查虚拟机的堆参数（-Xmx 与-Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

方法区的内部结构



《深入理解 Java 虚拟机》书中对方法区（Method Area）存储内容描述如下：它用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等。



类型信息

对每个加载的类型（类 class、接口 interface、枚举 enum、注解 annotation），JVM 必须在方法区中存储以下类型信息：

- 这个类型的完整有效名称（全名=包名.类名）
- 这个类型直接父类的完整有效名（对于 interface 或是 java.lang.object，都没有父类）
- 这个类型的修饰符（public, abstract, final 的某个子集）
- 这个类型直接接口的一个有序列表

域信息 (属性)

JVM 必须在方法区中保存类型的所有域的相关信息以及域的声明顺序。

域的相关信息包括：域名称、域类型、域修饰符（public, private, protected, static, final, volatile, transient 的某个子集）

方法（Method）信息

JVM 必须保存所有方法的以下信息，同域信息一样包括声明顺序：

- 方法名称
- 方法的返回类型（或 void）
- 方法参数的数量和类型（按顺序）
- 方法的修饰符（public, private, protected, static, final, synchronized, native, abstract 的一个子集）
- 方法的字节码（bytecodes）、操作数栈、局部变量表及大小（abstract 和 native 方法除外）
- 异常表（abstract 和 native 方法除外）

每个异常处理的开始位置、结束位置、代码处理在程序计数器中的偏移地址、被捕获的异常类的常量池索引

non-final 的类变量 final衡量一个变量是否是常量，类变量衡量它是不是静态的，这两个关键字不矛盾。

静态变量和类关联在一起，随着类的加载而加载，他们成为类数据在逻辑上的一部分。

类变量被类的所有实例共享，即使没有类实例时，你也可以访问它

```
/**  
 * non-final 的类变量  
 *  
 * @author: 陌溪  
 * @create: 2020-07-08-16:54  
 */  
public class MethodAreaTest {  
    public static void main(String[] args) {  
        Order order = null;  
        order.hello();  
        System.out.println(order.count);  
    }  
}  
class Order {
```

```

public static int count = 1;
public static final int number = 2;
public static void hello() {
    System.out.println("hello!");
}
}

```

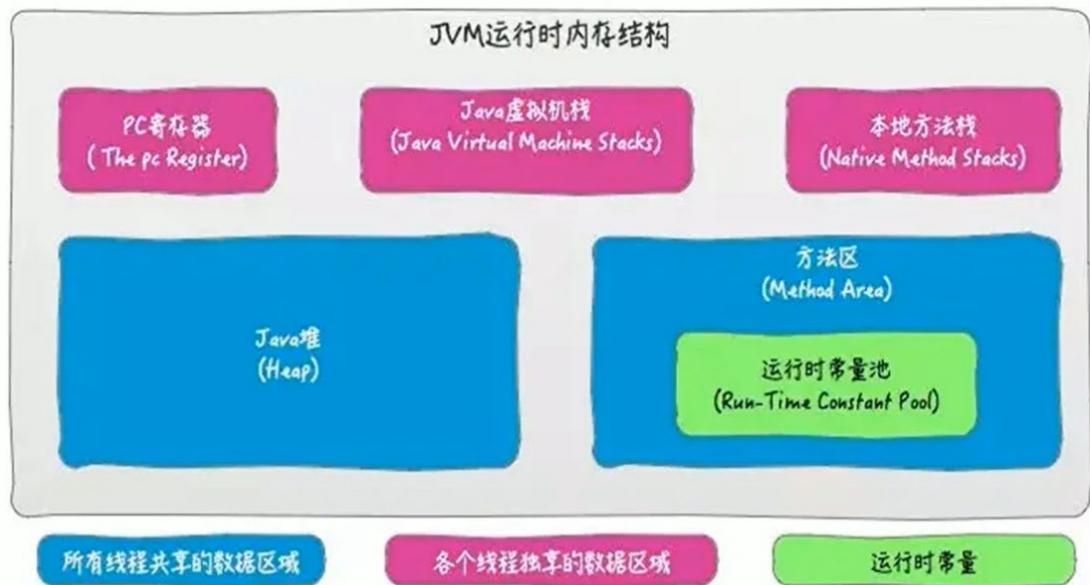
如上代码所示，即使我们把 order 设置为 null，也不会出现空指针异常

全局常量

全局常量就是使用 static final 进行修饰，被声明为 final 的类变量的处理方法在编译的时候就会被分配。普通的类变量是在类加载器子系统的linking-prepare阶段才被赋予默认值，在随后<clinit>()阶段才被初始化为给定值。

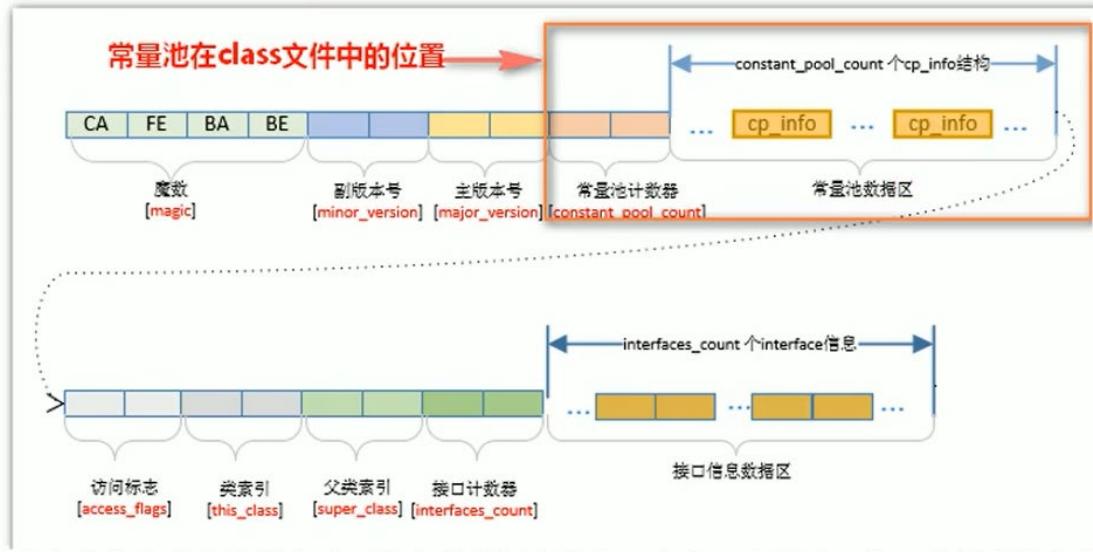
运行时常量池 VS 常量池

运行时常量池，就是运行时常量池



- 方法区，内部包含了运行时常量池
- 字节码文件，内部包含了常量池
- 要弄清楚方法区，需要理解清楚 ClassFile，因为加载类的信息都在方法区。
- 要弄清楚方法区的运行时常量池，需要理解清楚 classFile 中的常量池。

常量池



一个有效的字节码文件中除了包含类的版本信息、字段、方法以及接口等描述符信息外，还包含一项信息就是常量池表（Constant Pool Table），包括各种字面量和对类型、域和方法的符号引用。

为什么需要常量池

一个 java 源文件中的类、接口，编译后产生一个字节码文件。而 Java 中的字节码需要数据支持，通常这种数据会很大以至于不能直接存到字节码里，换另一种方式，可以存到常量池，这个字节码包含了指向常量池的引用。r 在动态链接的时候会用到运行时常量池，之前有介绍。

比如：如下的代码：

```
public class SimpleClass {  
    public void sayHello() {  
        System.out.println("hello");  
    }  
}
```

虽然上述代码只有 194 字节，但是里面却使用了 String、System、PrintStream 及 Object 等结构。这里的代码量其实很少了，如果代码多的话，引用的结构将会更多，这里就需要用到常量池了。

常量池中有什么

- 数量值
- 字符串值

- 类引用
- 字段引用
- 方法引用

例如下面这段代码

```
public class MethodAreaTest2 {
    public static void main(String args[]) {
        Object obj = new Object();
    }
}
```

将被翻译成如下字节码

```
new #2
dup
invokespecial
```

小结

常量池、可以看做是一张表，虚拟机指令根据这张常量表找到要执行的类名、方法名、参数类型、字面量等类型

运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。

常量池表（Constant Pool Table）是 Class 文件的一部分，用于存放编译期生成的各种字面量与符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。

运行时常量池，在加载类和接口到虚拟机后，就会创建对应的运行时常量池。

JVM 为每个已加载的类型（类或接口）都维护一个常量池。池中的数据项像数组项一样，是通过索引访问的。

运行时常量池中包含多种不同的常量，包括编译期就已经明确的数值字面量，也包括到运行期解析后才能够获得的方法或者字段引用。此时不再是常量池中的符号地址了，这里换为真实地址。

运行时常量池，相对于 Class 文件常量池的另一重要特征是：具备动态性。

运行时常量池类似于传统编程语言中的符号表（symboltable），但是它所包含的数据却比符号表要更加丰富一些。可以使用 String. intern() 将字符串放入运行时常量池

当创建类或接口的运行时常量池时，如果构造运行时常量池所需的内存空间超过了方法区所能提供的最大值，则 JVM 会抛出 OutOfMemoryError 异常。

方法区使用举例

如下代码

```
public class MethodAreaDemo {  
    public static void main(String args[]) {  
        int x = 500;  
        int y = 100;  
        int a = x / y;  
        int b = 50;  
        System.out.println(a+b);  
    }  
}
```

字节码执行过程展示



首先现将操作数 500 放入到操作数栈中



然后存储到局部变量表中



然后重复一次，把 100 放入局部变量表中，最后再将变量表中的 500 和 100 取出，进行操作



将 500 和 100 进行一个除法运算，在把结果入栈



在最后就是输出流，需要调用运行时常量池的常量



最后调用 invokevirtual（虚方法调用），然后返回



返回时



程序计数器始终计算的都是当前代码运行的位置，目的是为了方便记录 方法调用后能够正常返回，或者是进行了 CPU 切换后，也能回到原来的代码进行执行。

方法区的演进细节

首先明确：只有 Hotspot 才有永久代。BEA JRockit、IBM J9 等来说，是不存在永久代的概念的。原则上如何实现方法区属于虚拟机实现细节，不受《Java 虚拟机规范》管束，并不要求统一

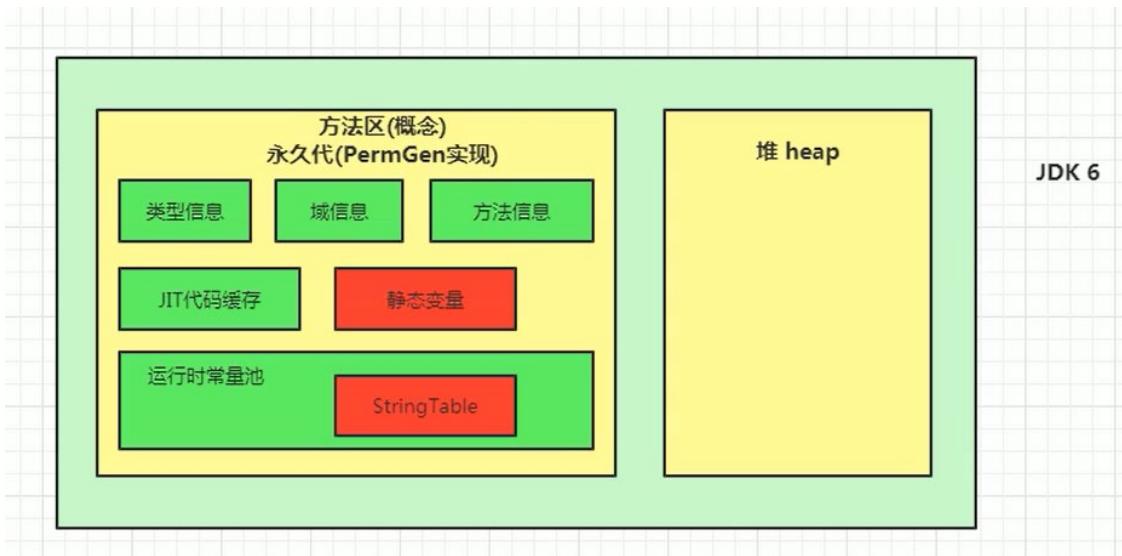
Hotspot 中方法区的变化：

JDK1.6 及以前 有永久代，静态变量在永久代上，字符串常量池在运行时常量池上

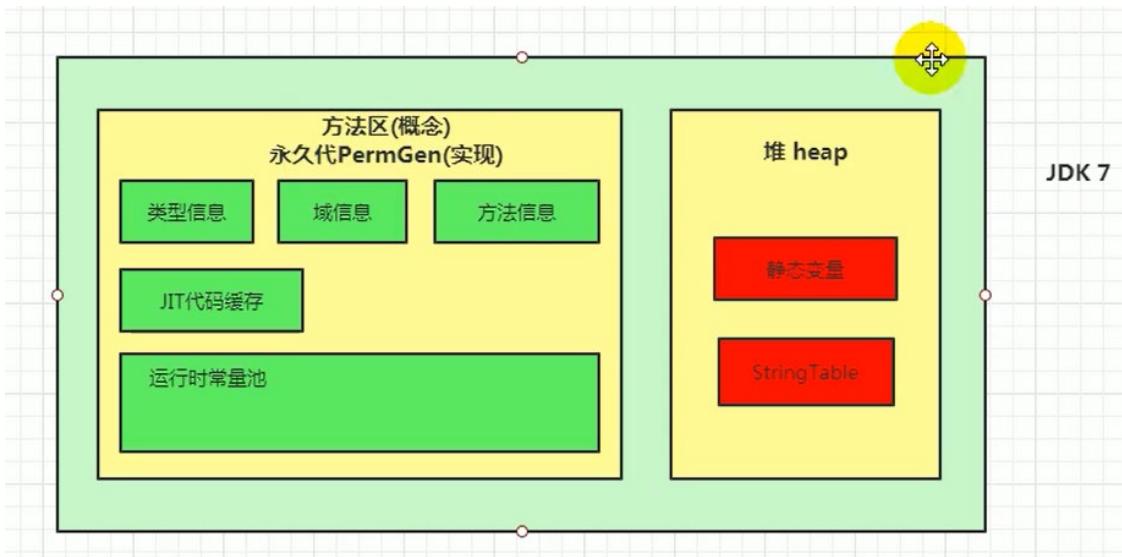
JDK1.7 有永久代，但已经逐步“去永久代”，字符串常量池、静态变量移除，保存在堆中

JDK1.8 无永久代，类型信息，字段，方法，常量保存在本地内存的元空间，但字符串常量池、静态变量仍然在堆中。

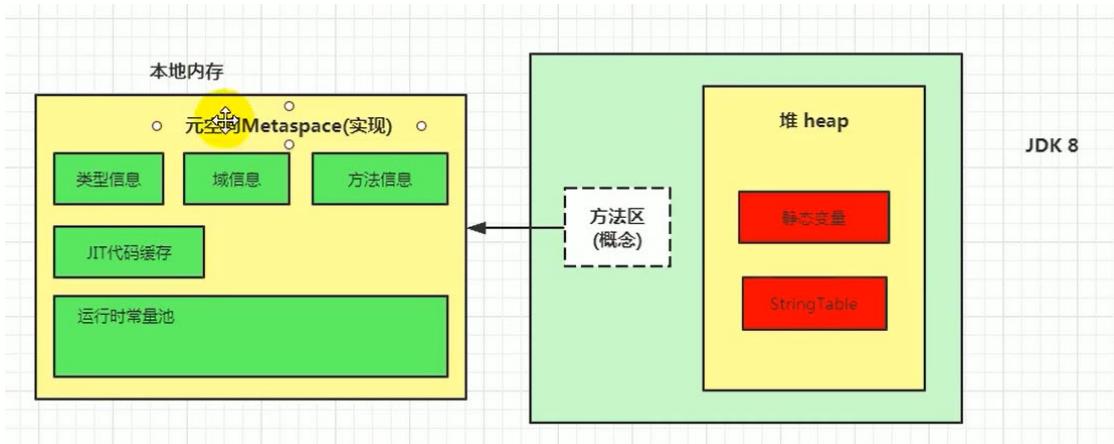
JDK6 的时候



JDK7 的时候



JDK8 的时候，元空间大小只受物理内存影响



为什么永久代要被元空间替代?

JRockit 是和 HotSpot 融合后的结果，因为 JRockit 没有永久代，所以他们不需要配置永久代。

随着 Java8 的到来，HotSpot VM 中再也见不到永久代了。但是这并不意味着类的元数据信息也消失了。这些数据被移到了一个与堆不相连的本地内存区域，这个区域叫做元空间（Metaspace）。

由于类的元数据分配在本地内存中，元空间的最大可分配空间就是系统可用内存空间，这项改动是很有必要的，原因有：

- 为永久代设置空间大小是很难确定的。

在某些场景下，如果动态加载类过多，容易产生 Perm 区的 oom。比如某个实际 Web 工程中，因为功能点比较多，在运行过程中，要不断动态加载很多类，经常出现致命错误。

`"Exception in thread dubbo client x.x
connector'java.lang.OutOfMemoryError:PermGen space"`

而元空间和永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。

- 对永久代进行调优是很困难的。

- 主要是为了降低 Full GC

有些人认为方法区（如 HotSpot 虚拟机中的元空间或者永久代）是没有垃圾收集行为的，其实不然。《Java 虚拟机规范》对方法区的约束是非常宽松的，提到过可以不要求虚拟机在方法区中实现垃圾收集。事实上也确实有未实现或未能完整实现方法区类型卸载的收集器存在（如 JDK11 时期的 ZGC 收集器就不支持类卸载）。

一般来说这个区域的回收效果比较难令人满意，尤其是类型的卸载，条件相当苛刻。但是这部分区域的回收有时又确实是必要的。以前 sun 公司的 Bug 列表中，曾出现过的若干个严重的 Bug 就是由于低版本的 HotSpot 虚拟机对此区域未完全回收而导致内存泄漏

方法区的垃圾收集主要回收两部分内容：常量池中废弃的常量和不再使用的类型

StringTable 为什么要调整位置

jdk7 中将 StringTable 放到了堆空间中。因为永久代的回收效率很低，在 full gc 的时候才会触发。而 full gc 是老年代的空间不足、永久代不足时才会触发。

这就导致 stringTable 回收效率不高。而我们开发中会有大量的字符串被创建，回收效率低，导致永久代内存不足。放到堆里，能及时回收内存。

静态变量存放在那里？ 这个静态变量指的是静态变量名，而不是对象，对象在堆内存中

静态引用对应的对象实体始终都存在堆空间，首先这个静态变量指的是变量名

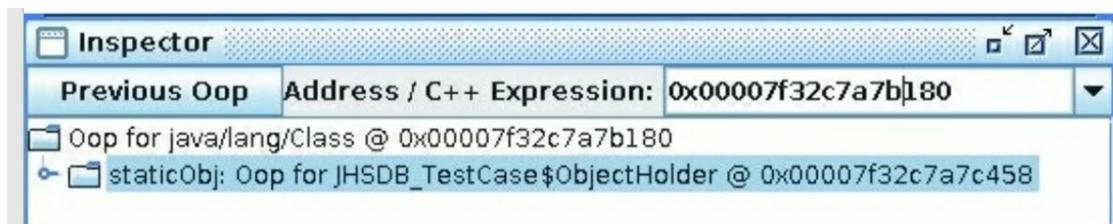
可以使用 jhsdb.ext，需要在 jdk9 的时候才引入的

staticobj 随着 Test 的类型信息存放在方法区，instanceobj 随着 Test 的对象实例存放在 Java 堆，localobject 则是存放在 foo () 方法栈帧的局部变量表中。

```
hsdb>scanoops 0x00007f32c7800000 0x00007f32c7b50000 JHSDB_TestCase$ObjectHolder  
0x00007f32c7a7c458 JHSDB_TestCase$ObjectHolder  
0x00007f32c7a7c480 JHSDB_TestCase$ObjectHolder  
0x00007f32c7a7c490 JHSDB_TestCase$ObjectHolder
```

测试发现：三个对象的数据在内存中的地址都落在 Eden 区范围内，所以结论：只要是对象实例必然会在 Java 堆中分配。

接着，找到了一个引用该 staticobj 对象的地方，是在一个 java.lang.Class 的实例里，并且给出了这个实例的地址，通过 Inspector 查看该对象实例，可以清楚看到这确实是一个 java.lang.Class 类型的对象实例，里面有一个名为 staticobj 的实例字段：



从《Java 虚拟机规范》所定义的概念模型来看，所有 Class 相关的信息都应该存放在方法区之中，但方法区该如何实现，《Java 虚拟机规范》并未做出规定，这就成了一件允许不同虚拟机自己灵活把握的事情。JDK7 及其以后版本的 HotSpot 虚拟机选择把静态变量与类型在 Java 语言一端的映射 class 对象存放在一起，存储于 Java 堆之中，从我们的实验中也明确验证了这一点

方法区的垃圾回收

有些人认为方法区（如 Hotspot 虚拟机中的元空间或者永久代）是没有垃圾收集行为的，其实不然。《Java 虚拟机规范》对方法区的约束是非常宽松的，提到过可以不要求虚拟机在方法区中实现垃圾收集。事实上也确实有未实现或未能完整实现方法区类型卸载的收集器存在（如 JDK11 时期的 zGC 收集器就不支持类卸载）。

一般来说这个区域的回收效果比较难令人满意，尤其是类型的卸载，条件相当苛刻。但是这部分区域的回收有时又确实是必要的。以前 sun 公司的 Bug 列表中，曾出现过的若干个严重的 Bug 就是由于低版本的 HotSpot 虚拟机对此区域未完全回收而导致内存泄漏。

方法区的垃圾收集主要回收两部分内容：常量池中废弃的常量和不再使用的类型。

先来说说方法区内常量池之中主要存放的两大类常量：字面量和符号引用。字面量比较接近 Java 语言层次的常量概念，如文本字符串、被声明为 final 的常量值等。而符号引用则属于编译原理方面的概念，包括下面三类常量：

- 类和接口的全限定名
- 字段的名称和描述符
- 方法的名称和描述符

HotSpot 虚拟机对常量池的回收策略是很明确的，只要常量池中的常量没有被任何地方引用，就可以被回收。

回收废弃常量与回收 Java 堆中的对象非常类似。（关于常量的回收比较简单，重点是类的回收）

判定一个常量是否“废弃”还是相对简单，而要判定一个类型是否属于“不再被使用的类”的条件就比较苛刻了。需要同时满足下面三个条件：

- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类及其任何派生子类的实例。
- 加载该类的类加载器已经被回收，这个条件除非是经过精心设计的可替换类加载器的场景，如 osgi、JSP 的重加载等，否则通常很难达成的。
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

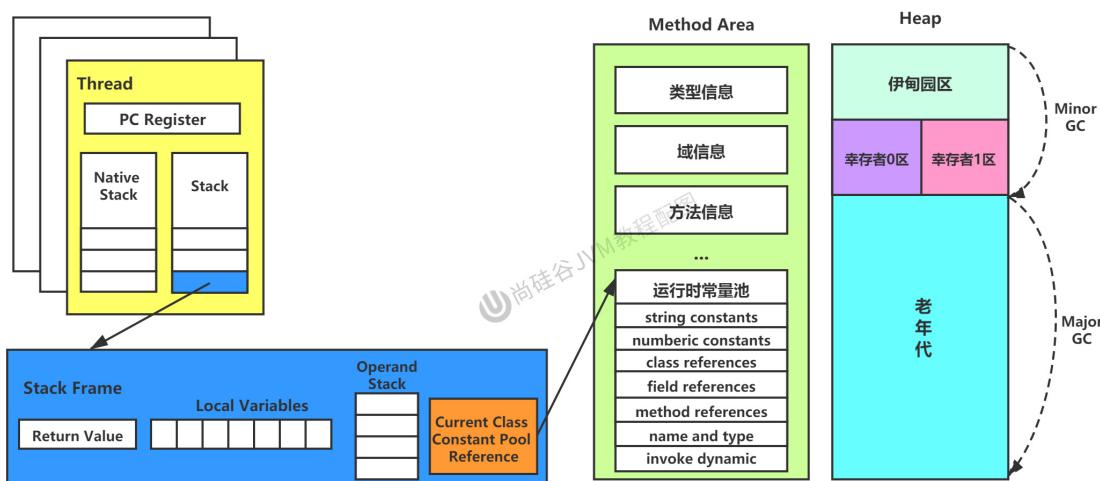
虚拟机对常量的垃圾回收比较简单

虚拟机对类的垃圾回收比较困难

Java 虚拟机被允许对满足上述三个条件的无用类进行回收，这里说的仅仅是“被允许”，而并不是和对象一样，没有引用了就必然会回收。关于是否要对类型进行回收，HotSpot 虚拟机提供了-Xnoclassgc 参数进行控制，还可以使用-verbose:class 以及 -XX: +TraceClass-Loading、-XX: +TraceClassUnLoading 查看类加载和卸载信息

- 在大量使用反射、动态代理、CGLib 等字节码框架，动态生成 JSP 以及 oSGi 这类频繁自定义类加载器的场景中，通常都需要 Java 虚拟机具备类型卸载的能力，以保证不会对方法区造成过大的内存压力。

总结



常见面试题

百度三面：说一下 JVM 内存模型吧，有哪些区？分别干什么的？

蚂蚁金服：Java8 的内存分代改进 JVM 内存分哪几个区，每个区的作用是什么？

一面：JVM 内存分布/内存结构？栈和堆的区别？堆的结构？为什么两个 survivor 区？二面：Eden 和 survivor 的比例分配

小米：jvm 内存分区，为什么要有新生代和老年代

字节跳动：二面：Java 的内存分区 二面：讲讲 vm 运行时数据库区 什么时候对象会进入老年代？

京东：JVM 的内存结构，Eden 和 Survivor 比例。JVM 内存为什么要分成新生代，老年代，持久代。新生代中为什么要分为 Eden 和 survivor。

天猫：一面：Jvm 内存模型以及分区，需要详细到每个区放什么。一面：JVM 的内存模型，Java8 做了什么改

拼多多：JVM 内存分哪几个区，每个区的作用是什么？

美团：java 内存分配 jvm 的永久代中会发生垃圾回收吗？一面：jvm 内存分区，为什么要有新生代和老年代？

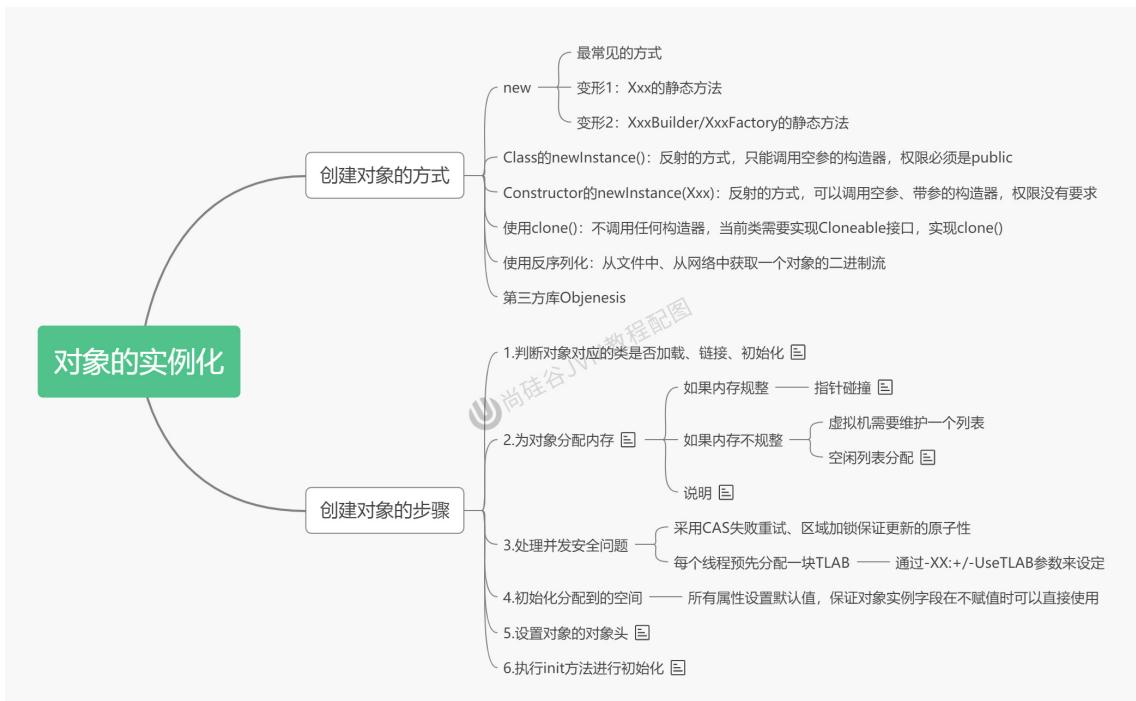
对象实例化内存布局与访问定位

对象实例化

面试题

- 对象在 JVM 中是怎么存储的？
- 对象头信息里面有哪些东西？
- Java 对象头有什么？

从对象创建的方式 和 步骤开始说



对象创建方式

- `new`: 最常见的方式、单例类中调用 `getInstance` 的静态类方法, `XXXFactory` 的静态方法
- `Class` 的 `newInstance` 方法: 在 JDK9 里面被标记为过时的方法, 因为只能调用空参构造器 (反射)

- Constructor 的 newInstance(XXX): 反射的方式，可以调用空参的，或者带参的构造器
- 使用 clone(): 不调用任何的构造器，要求当前的类需要实现 Cloneable 接口中的 clone 接口
- 使用序列化：序列化一般用于 Socket 的网络传输
- 第三方库 Objenesis

创建对象的步骤

判断对象对应的类是否加载、链接、初始化

虚拟机遇到一条 new 指令，首先去检查这个指令的参数能否在 Metaspace 的常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载，解析和初始化。（即判断类元信息是否存在）。如果没有，那么在双亲委派模式下，使用当前类加载器以 ClassLoader + 包名 + 类名为 key 进行查找对应的 .class 文件，如果没有找到文件，则抛出 ClassNotFoundException 异常，如果找到，则进行类加载，并生成对应的 Class 对象。

为对象分配内存

首先计算对象占用空间的大小，接着在堆中划分一块内存给新对象。如果实例成员变量是引用变量，仅分配引用变量空间即可，即 4 个字节大小

- 如果内存规整：指针碰撞
- 如果内存不规整
 - 虚拟机需要维护一个列表
 - 空闲列表分配

如果内存是规整的，那么虚拟机将采用的是指针碰撞法（Bump The Point）来为对象分配内存。

意思是所有用过的内存放在一边，空闲的内存放另外一边，中间放着一个指针作为分界点的指示器，分配内存就仅仅是把指针指向空闲那边挪动一段与对象大小相等的距离罢了。如果垃圾收集器选择的是 Serial，ParNew 这种基于压缩算法的，虚拟机采用这种分配方式。一般使用带 Compact（整理）过程的收集器时，使用指针碰撞。

如果内存不是规整的，已使用的内存和未使用的内存相互交错，那么虚拟机将采用的是空闲列表来为对象分配内存。意思是虚拟机维护了一个列表，记录上那些内存

块是可用的，再分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的内容。这种分配方式成为了“空闲列表（Free List）”

选择哪种分配方式由 Java 堆是否规整所决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

处理并发问题

- 采用 CAS 配上失败重试保证更新的原子性
- 每个线程预先分配 TLAB - 通过设置 -XX:+UseTLAB 参数来设置（区域加锁机制）
 - 在 Eden 区给每个线程分配一块区域

初始化分配到的内存

所有属性设置默认值，保证对象实例字段在不赋值时可以直接使用。

一般地，给对象属性赋值的操作包括如下四步：这个环节仅仅进行第一步

- 属性的默认初始化
- 显示初始化
- 代码块中的初始化
- 构造器初始化

设置对象的对象头

将对象的所属类（即类的元数据信息）、对象的HashCode 和对象的 GC 信息、锁信息等数据存储在对象的对象头中。这个过程的具体设置方式取决于 JVM 实现。

执行 init 方法进行初始化

这个环节进行了上面的剩余三步

在 Java 程序的视角看来，初始化才正式开始。初始化成员变量，执行实例化代码块，调用类的构造方法，并把堆内对象的首地址赋值给引用变量

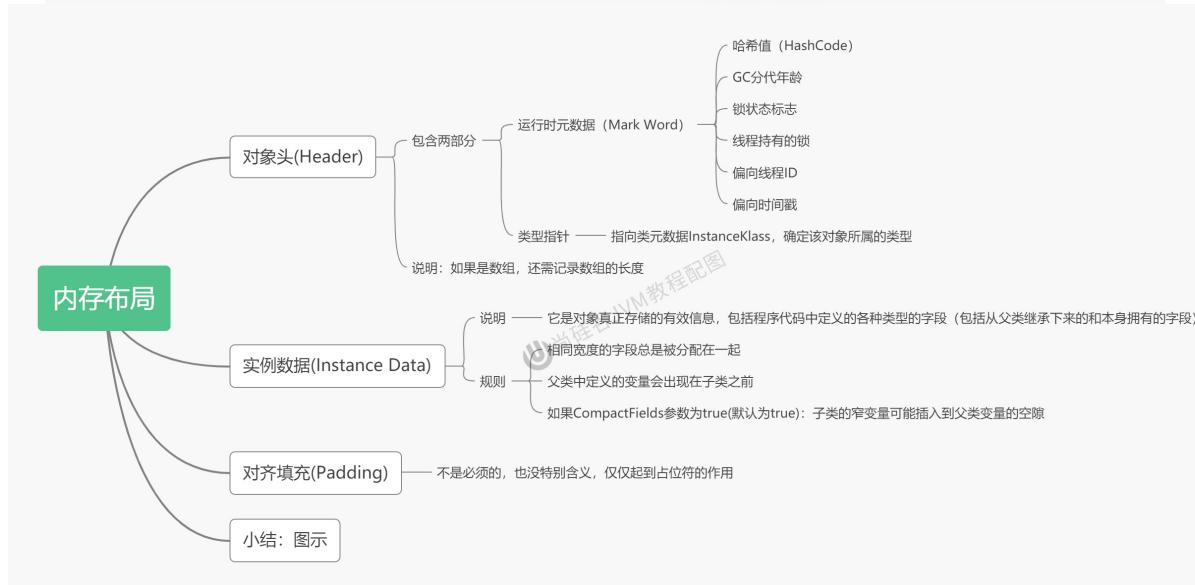
因此一般来说（由字节码中跟随 invokespecial 指令所决定），new 指令之后会接着就是执行方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完成创建出来。

对象实例化的过程

- 加载类元信息
- 为对象分配内存

- 处理并发问题
- 属性的默认初始化（零值初始化）
- 设置对象头信息
- 属性的显示初始化、代码块中初始化、构造器中初始化

对象内存布局



对象头

对象头包含了两部分，分别是 运行时元数据（Mark Word） 和 类型指针

如果是数组，还需要记录数组的长度

运行时元数据

- 哈希值 (HashCode)
- 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

类型指针

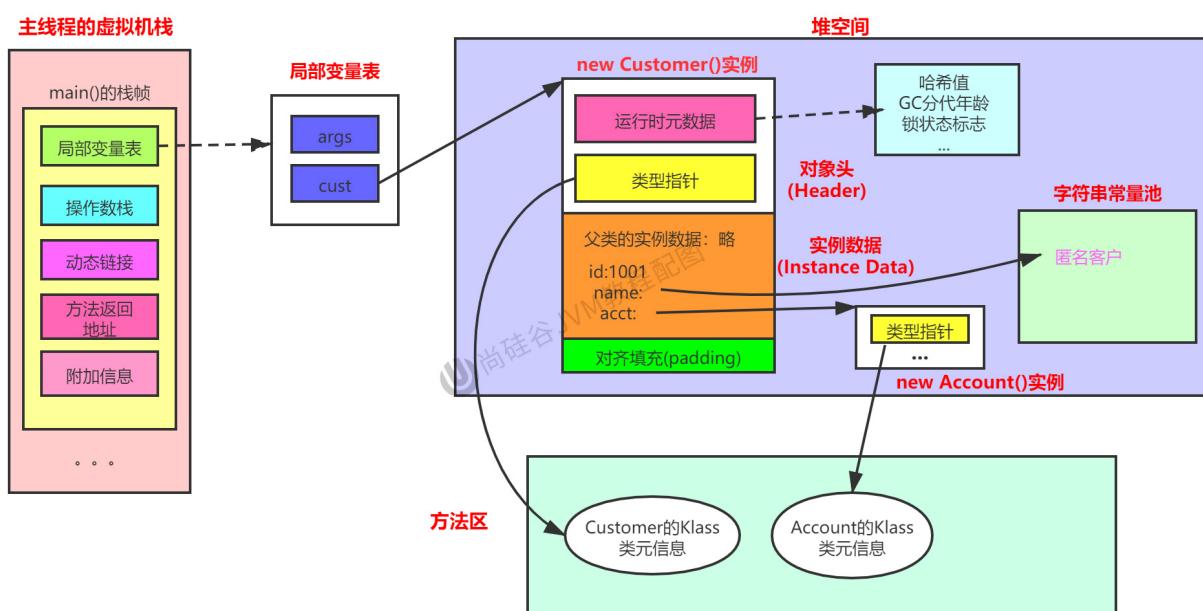
指向类元数据 InstanceKlass，确定该对象所属的类型。指向的其实是方法区中存放的类元信息

实例数据（Instance Data）

说明

不是必须的，也没有特别的含义，仅仅起到占位符的作用

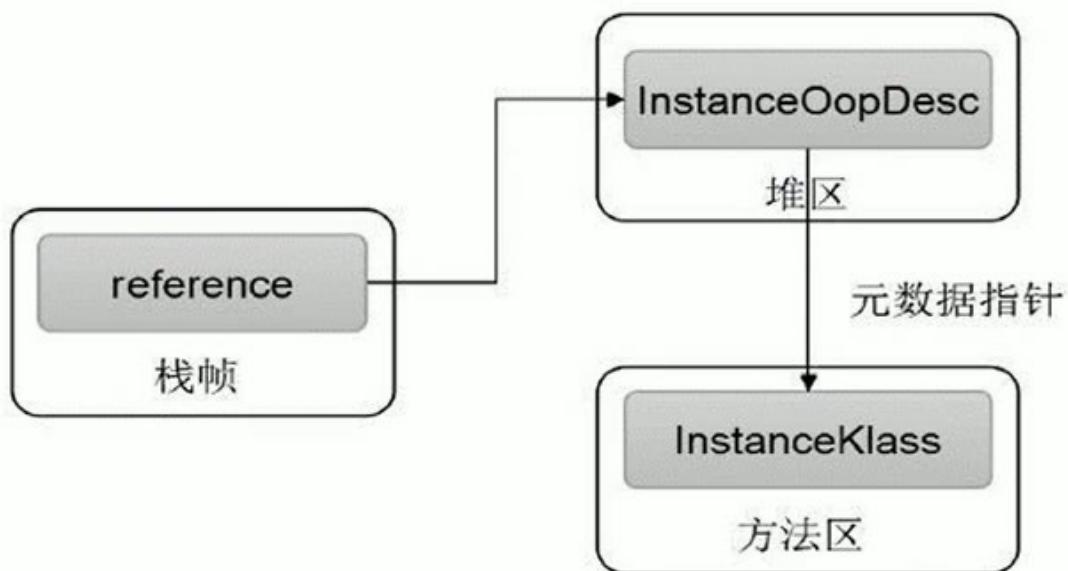
小结



对象的访问定位

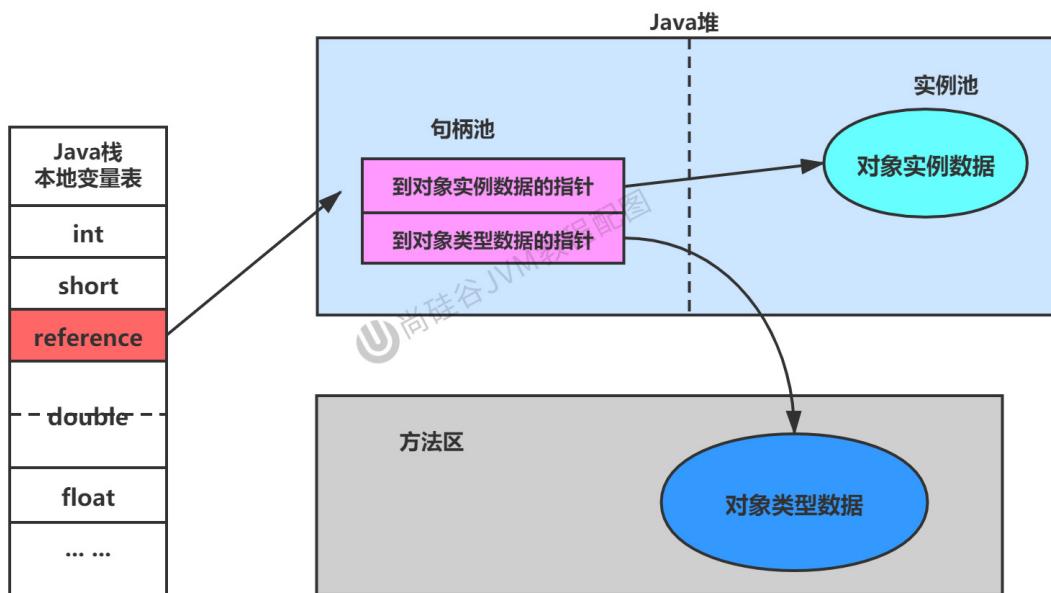
图示

JVM 是如何通过栈帧中的对象引用访问到其内部的对象实例呢？



对象访问的两种方式

句柄访问

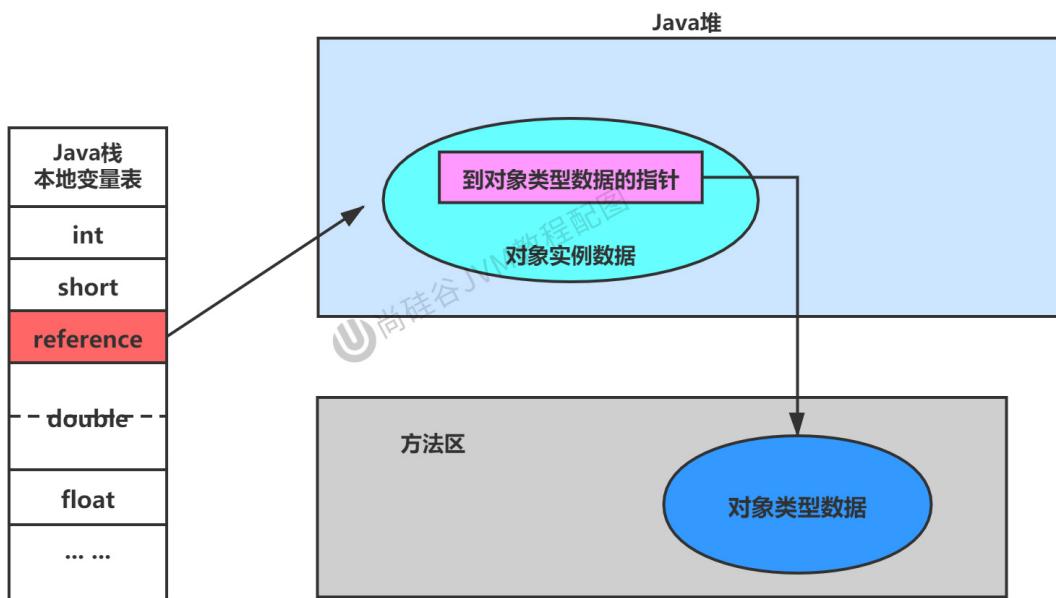


句柄访问就是说栈的局部变量表中，记录的对象的引用，然后在堆空间中开辟了一块空间，也就是句柄池

优点

reference 中存储稳定句柄地址，对象被移动（垃圾收集时移动对象很普遍）时只会改变句柄中实例数据指针即可，reference 本身不需要被修改

直接指针（HotSpot 采用）



直接指针是局部变量表中的引用，直接指向堆中的实例，在对象实例中有类型指针，指向的是方法区中的对象类型数据，好处：节省空间且速度快

直接内存 Direct Memory

JDK8元空间的实现使用直接内存

不是虚拟机运行时数据区的一部分，也不是《Java 虚拟机规范》中定义的内存区域。

直接内存是在 Java 堆外的、直接向系统申请的内存区间。

来源于 NIO，通过存在堆中的 DirectByteBuffer 操作 Native 内存

通常，访问直接内存的速度会优于 Java 堆。即读写性能高。

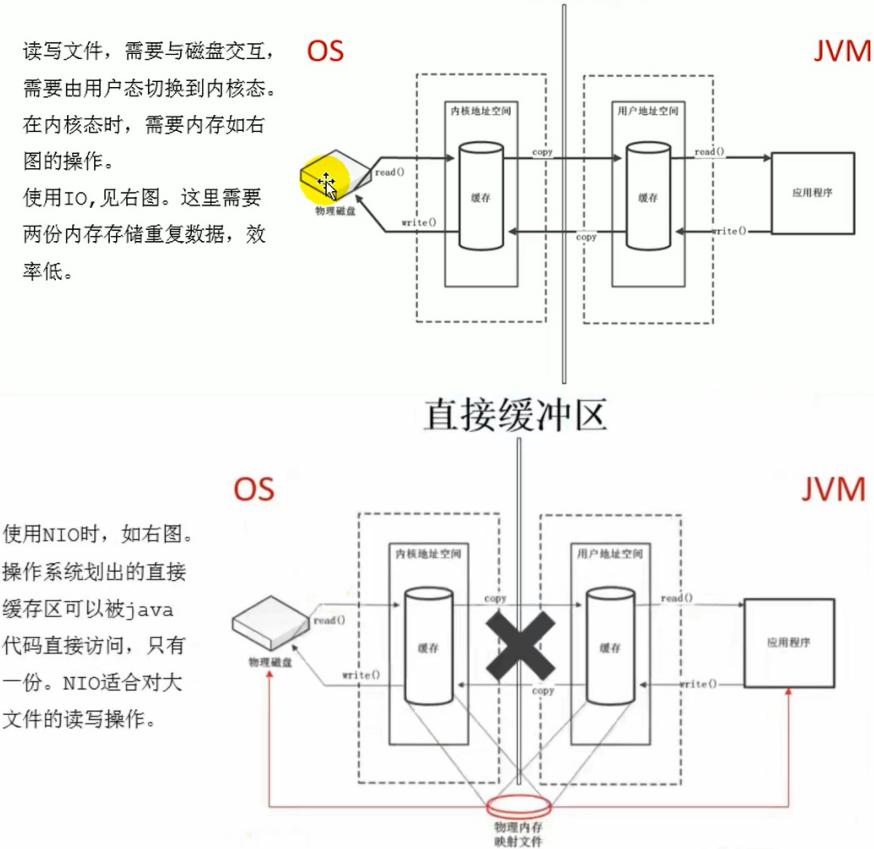
- 因此出于性能考虑，读写频繁的场合可能会考虑使用直接内存。
- Java 的 NIO 库允许 Java 程序使用直接内存，用于数据缓冲区

使用下列代码，直接分配本地内存空间

```
int BUFFER = 1024*1024*1024; // 1GB  
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(BUFFER);
```

非直接缓存区和缓存区

原来采用 BIO 架构，我们需要从用户态切换成内核态；NIO 借助直接缓存区的概念



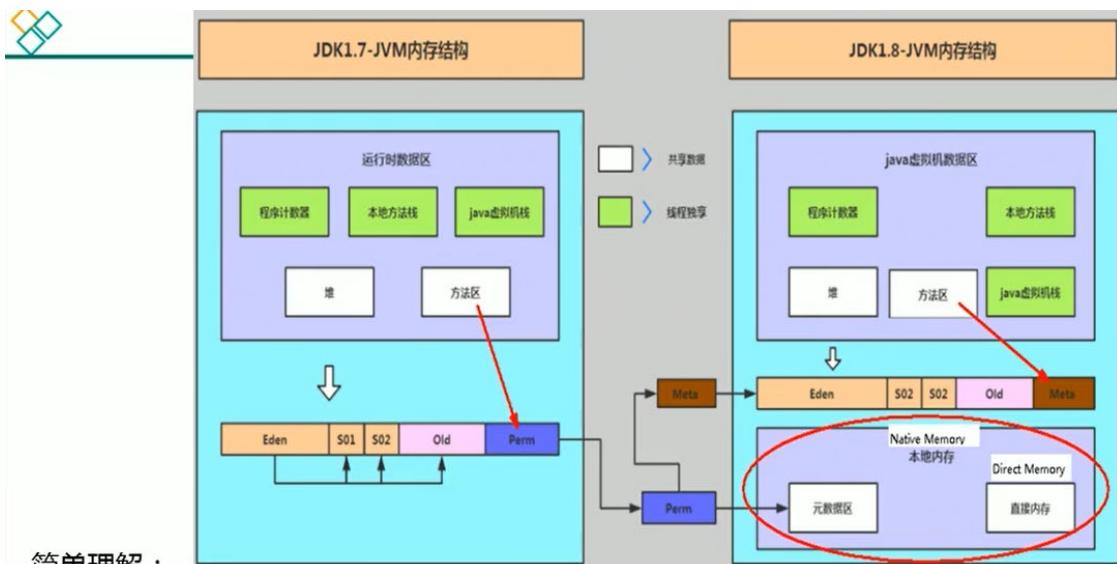
存在的问题

也可能导致 `outofMemoryError` 异常：OOM: Direct Buffer Memory

由于直接内存不在 Java 堆外，因此它的大小不会直接受限于`-xmx` 指定的最大堆大小，但是系统内存是有限的，Java 堆和直接内存的总和依然受限于操作系统能给出的最大内存。缺点

- 分配回收成本较高
 - 不受 JVM 内存回收管理，监控工具难以监控
- 直接内存大小可以通过 `MaxDirectMemorySize` 设置

如果不指定，默认与堆的最大值`-xmx` 参数值一致



简单理解：

java process memory = java heap + native memory

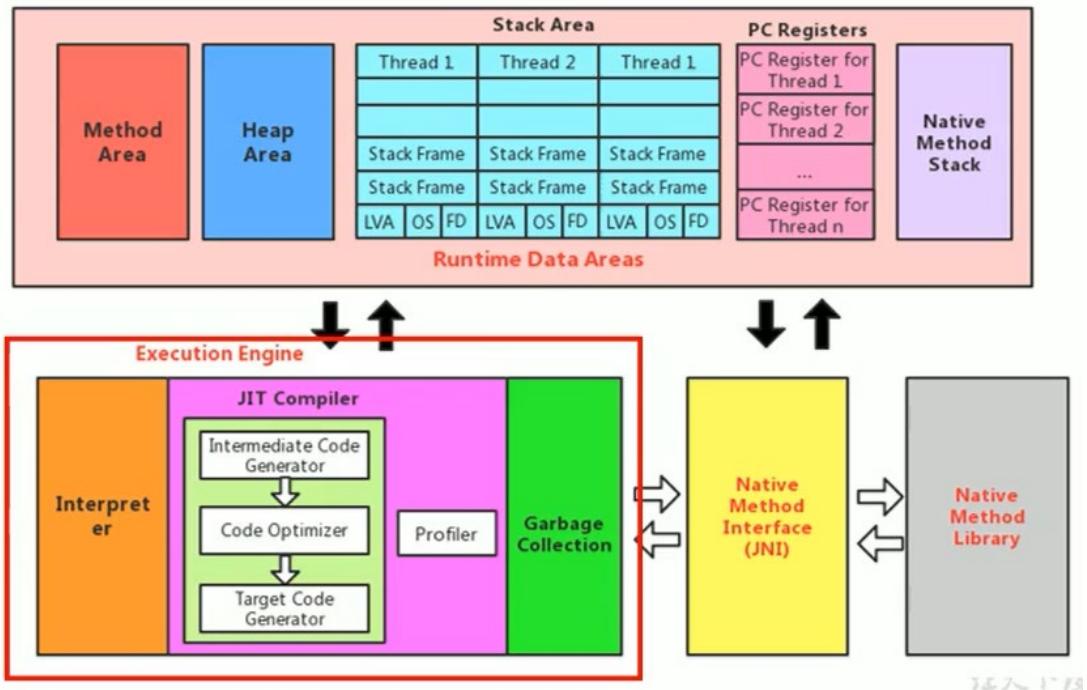
java程序进程中占用的空间=java堆、本地内存

让天下没有难学的技术

执行引擎

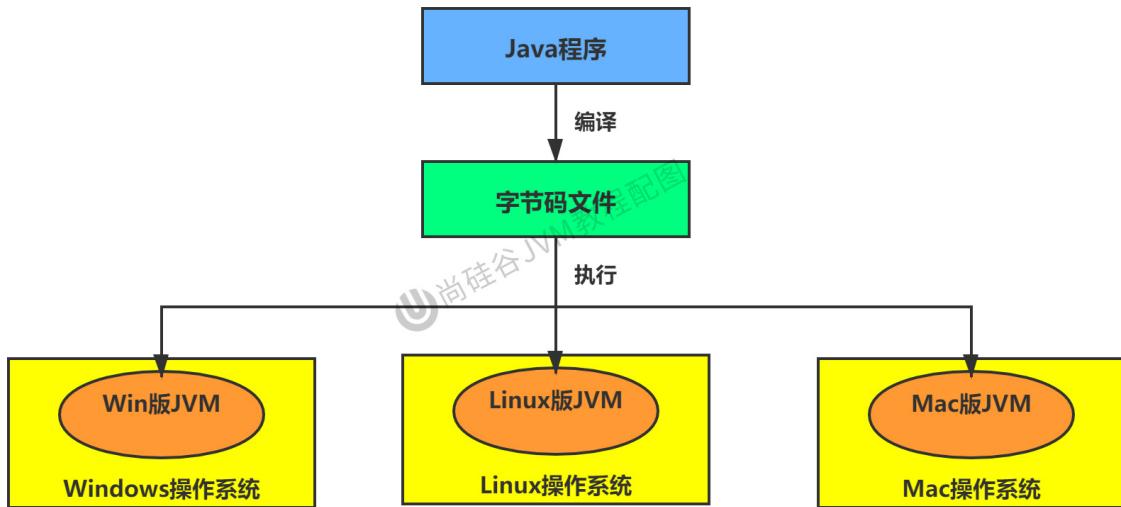
执行引擎概述

执行引擎属于 JVM 的下层，里面包括 解释器、及时编译器、垃圾回收器



执行引擎是 Java 虚拟机核心的组成部分之一。“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体系，能够执行那些不被硬件直接支持的指令集格式。

JVM 的主要任务是负责装载字节码到其内部，但字节码并不能够直接运行在操作系统之上，因为字节码指令并非等价于本地机器指令，它内部包含的仅仅只是一些能够被 JVM 所识别的字节码指令、符号表，以及其他辅助信息。

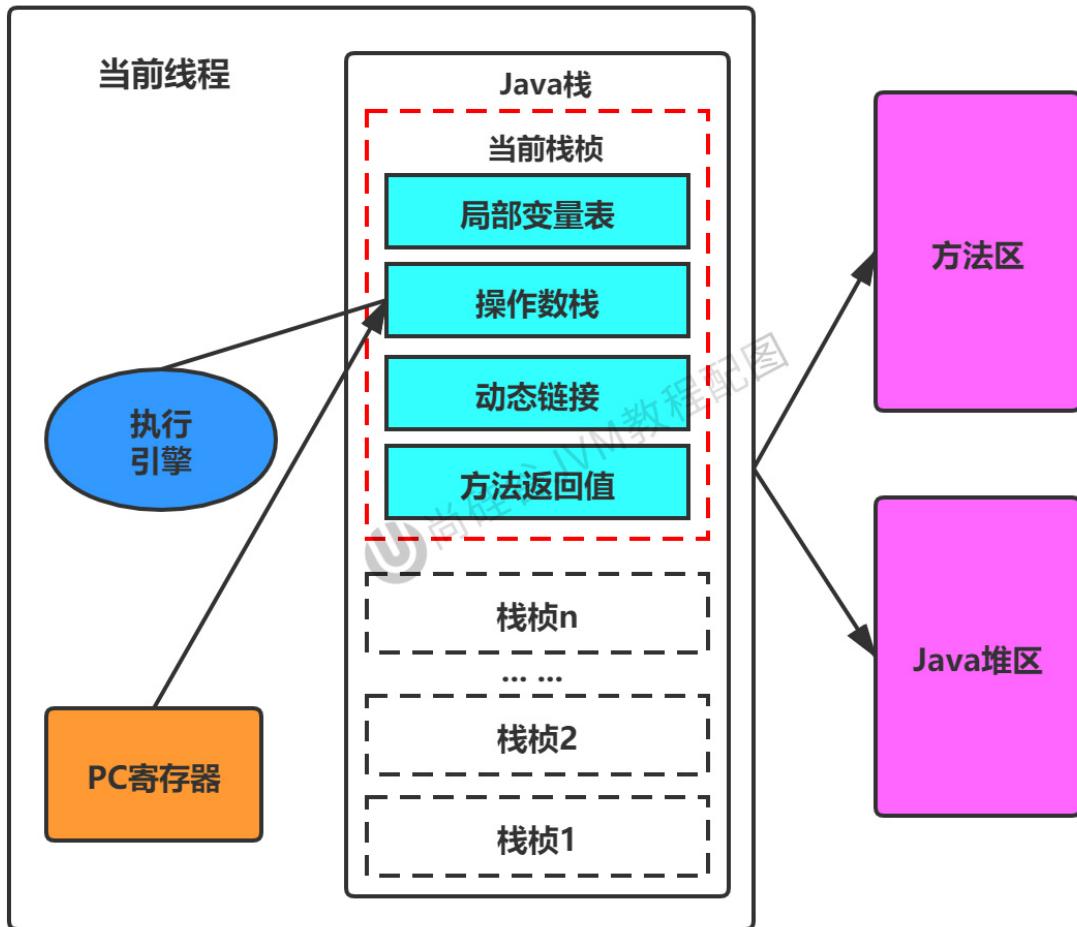


那么，如果想要让一个 Java 程序运行起来，执行引擎（Execution Engine）的任务就是将字节码指令解释/编译为对应平台上的本地机器指令才可以。简单来说，JVM 中的执行引擎充当了将高级语言翻译为机器语言的译者。

执行引擎的工作流程

- 执行引擎在执行的过程中究竟需要执行什么样的字节码指令完全依赖于 PC 寄存器。
- 每当执行完一项指令操作后，PC 寄存器就会更新下一条需要被执行的指令地址。

- 当然方法在执行的过程中，执行引擎有可能会通过存储在局部变量表中的对象引用准确定位到存储在 Java 堆区中的对象实例信息，以及通过对对象头中的元数据指针定位到目标对象的类型信息。

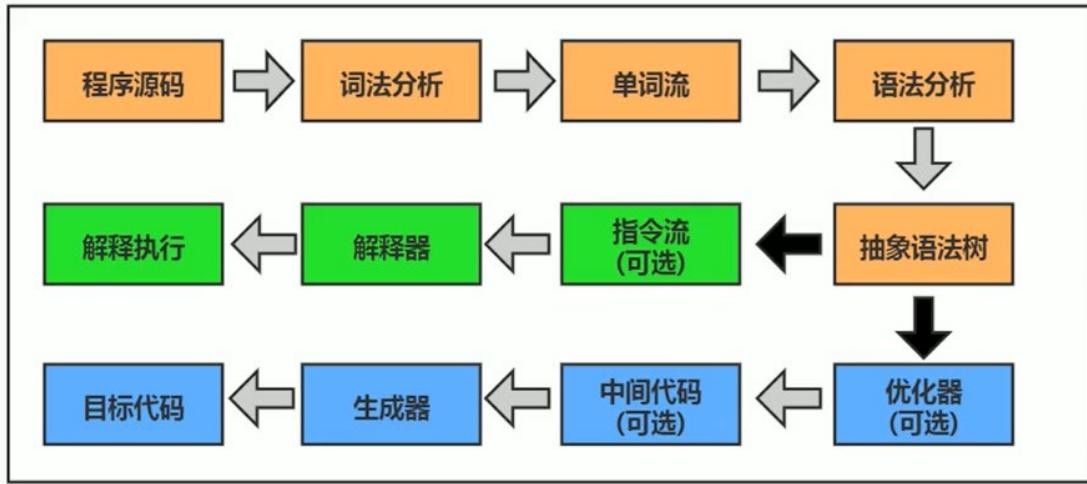


从外观上来看，所有的 Java 虚拟机的执行引擎输入，输出都是一致的：输入的是字节码二进制流，处理过程是字节码解析执行的等效过程，输出的是执行过程。

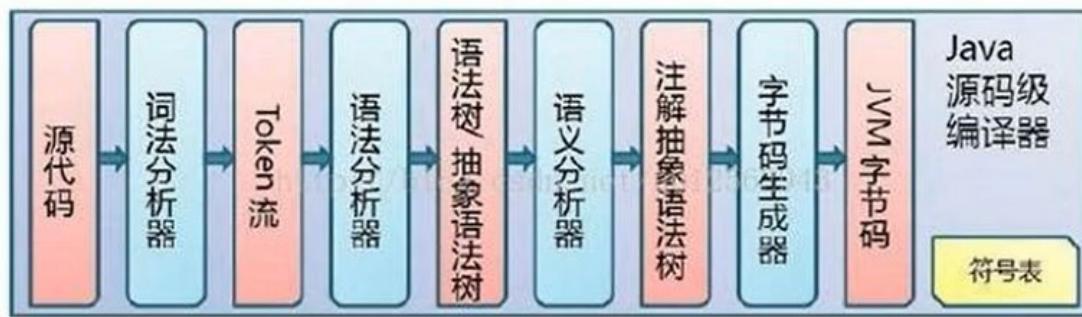
Java 代码编译和执行过程

大部分的程序代码转换成物理机的目标代码或虚拟机能执行的指令集之前，都需要经过上图中的各个步骤

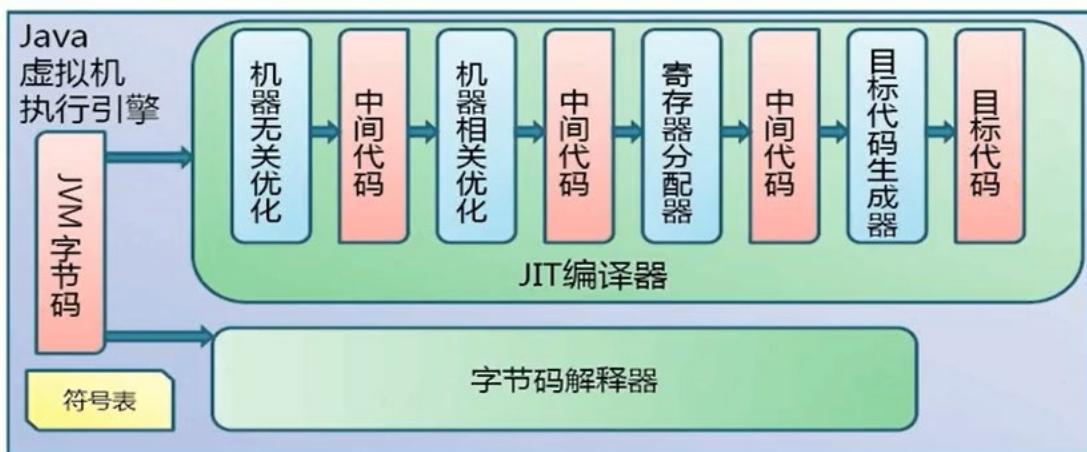
- 前面橙色部分是生成字节码文件的过程，和 JVM 无关
- 后面蓝色和绿色才是 JVM 需要考虑的过程



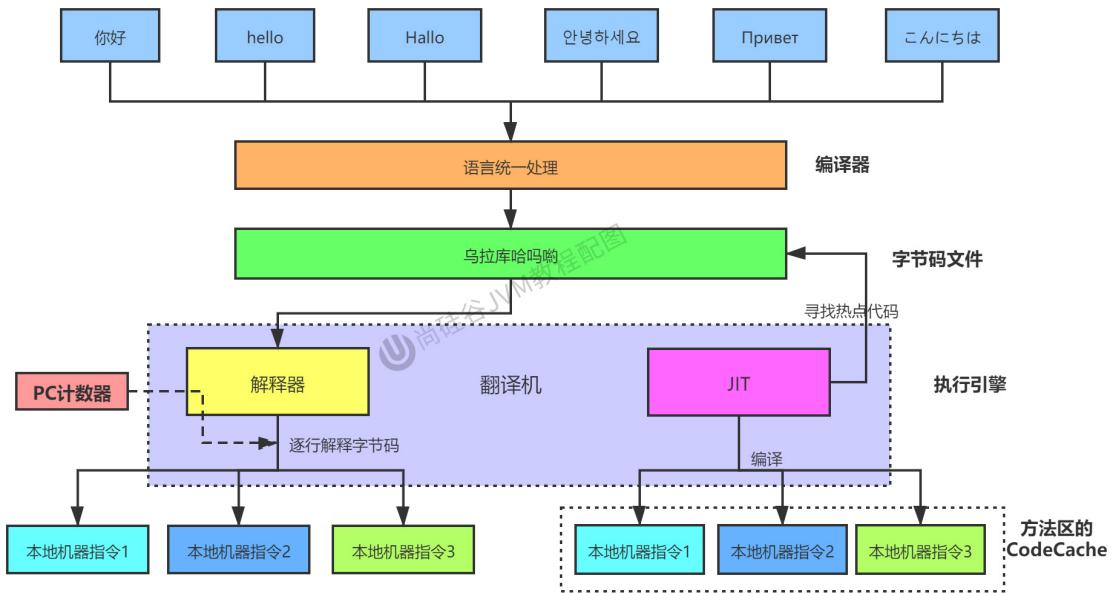
Java 代码编译是由 Java 源码编译器来完成，流程图如下所示（上图橙色部分）：



Java 字节码的执行是由 JVM 执行引擎来完成，流程图如下所示



我们用一个总的图，来说说解释器和编译器



什么是解释器（Interpreter）

当 Java 虚拟机启动时会根据预定义的规范对字节码采用逐行解释的方式执行，将每条字节码文件中的内容“翻译”为对应平台的本地机器指令执行。

什么是 JIT 编译器

JIT (Just In Time Compiler) 编译器：就是虚拟机将源代码直接编译成和本地机器平台相关的机器语言。

为什么 Java 是半编译半解释型语言

JDK1.4 时代，将 Java 语言定位为“解释执行”还是比较准确的。再后来，Java 也发展出可以直接生成本地代码的编译器。现在 JVM 在执行 Java 代码的时候，通常都会将解释执行与编译执行二者结合起来进行。

翻译成本地代码后，就可以做一个缓存操作，存储在方法区中

机器码、指令、汇编语言

机器码

各种用二进制编码方式表示的指令，叫做机器指令码。开始，人们就用它来编写程序，这就是机器语言。

机器语言虽然能够被计算机理解和接受，但和人们的语言差别太大，不易被人们理解和记忆，并且用它编程容易出差错。

用它编写的程序一经输入计算机，CPU 直接读取运行，因此和其他语言编的程序相比，执行速度最快。

机器指令与 CPU 紧密相关，所以不同种类的 CPU 所对应的机器指令也就不同。

指令

由于机器码是有 0 和 1 组成的二进制序列，可读性实在太差，于是人们发明了指令。

指令就是把机器码中特定的 0 和 1 序列，简化成对应的指令（一般为英文简写，如 mov, inc 等），可读性稍好

由于不同的硬件平台，执行同一个操作，对应的机器码可能不同，所以不同的硬件平台的同一种指令（比如 mov），对应的机器码也可能不同。

指令集

不同的硬件平台，各自支持的指令，是有差别的。因此每个平台所支持的指令，称之为对应平台的指令集。如常见的

- x86 指令集，对应的是 x86 架构的平台
- ARM 指令集，对应的是 ARM 架构的平台

汇编语言

由于指令的可读性还是太差，于是人们又发明了汇编语言。

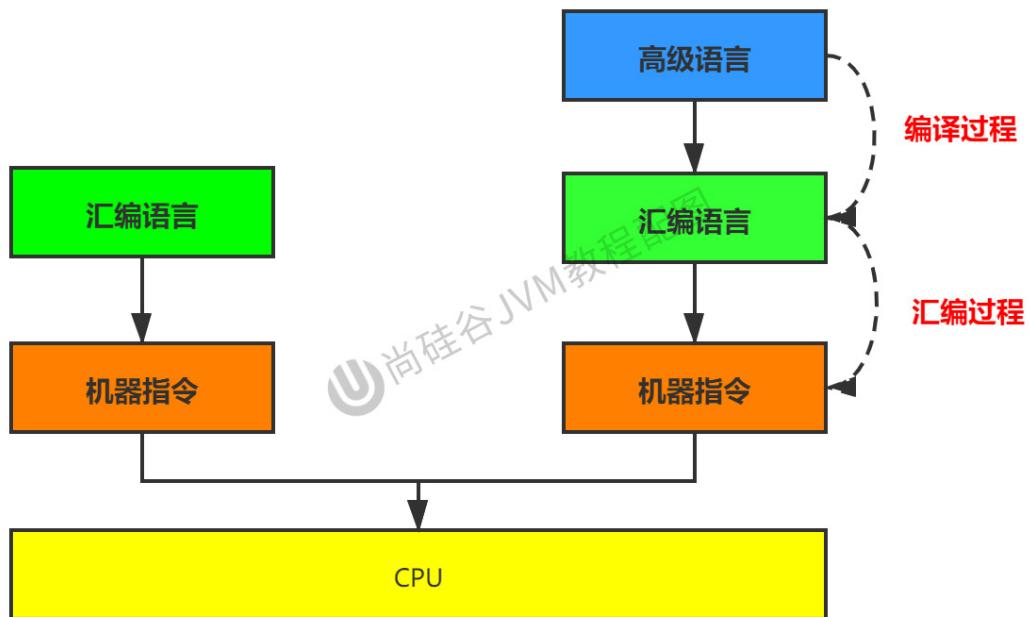
在汇编语言中，用助记符（Mnemonics）代替机器指令的操作码，用地址符号（Symbol）或标号（Label）代替指令或操作数的地址。在不同的硬件平台，汇编语言对应着不同的机器语言指令集，通过汇编过程转换成机器指令。

由于计算机只认识指令码，所以用汇编语言编写的程序还必须翻译成机器指令码，计算机才能识别和执行。

高级语言

为了使计算机用户编程序更容易些，后来就出现了各种高级计算机语言。

高级语言比机器语言、汇编语言更接近人的语言当计算机执行高级语言编写的程序时，仍然需要把程序解释和编译成机器的指令码。完成这个过程的程序就叫做解释程序或编译程序。



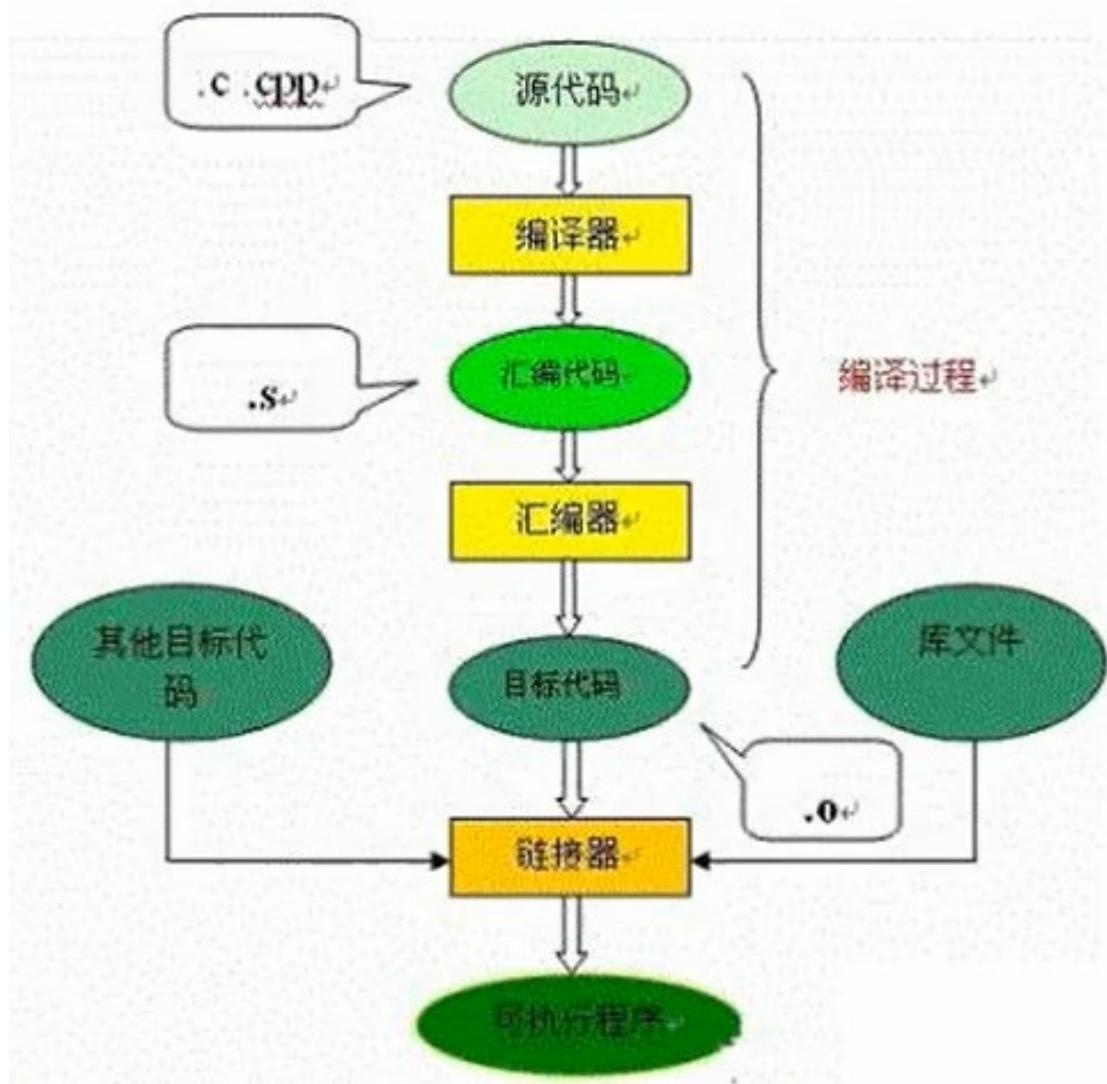
高级语言也不是直接翻译成机器指令，而是翻译成汇编语言吗，如下面说的 C 和 C++

C、C++源程序执行过程

编译过程又可以分成两个阶段：编译和汇编。

编译过程：是读取源程序（字符流），对之进行词法和语法的分析，将高级语言指令转换为功能等效的汇编代码

汇编过程：实际上指把汇编语言代码翻译成目标机器指令的过程。



字节码

字节码是一种中间状态（中间码）的二进制代码（文件），它比机器码更抽象，需要直译器转译后才能成为机器码。

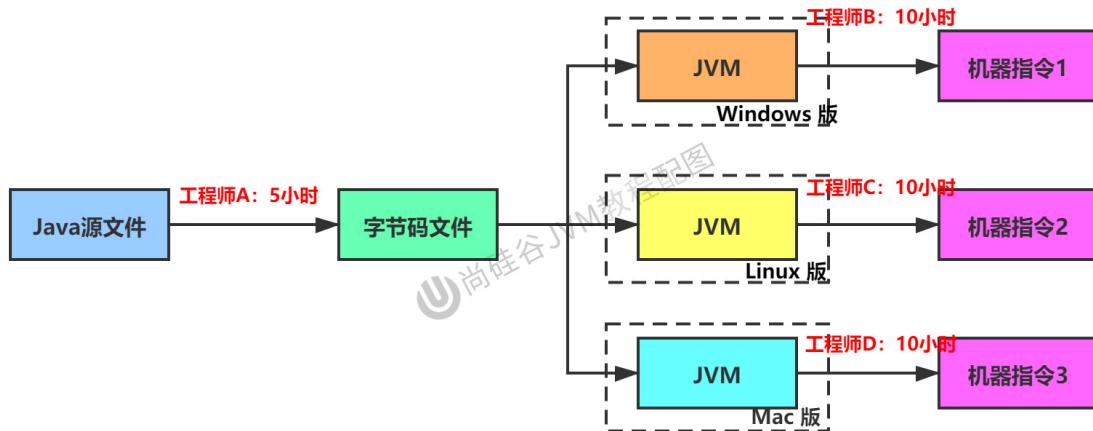
字节码主要为了实现特定软件运行和软件环境、与硬件环境无关。

字节码的实现方式是通过编译器和虚拟机器。编译器将源码编译成字节码，特定平台上的虚拟机器将字节码转译为可以直接执行的指令。

- 字节码典型的应用为：Java bytecode

解释器

JVM 设计者们的初衷仅仅只是单纯地为了满足 Java 程序实现跨平台特性，因此避免采用静态编译的方式直接生成本地机器指令，从而诞生了实现解释器在运行时采用逐行解释字节码执行程序的想法。



为什么 Java 源文件不直接翻译成 JVM，而是翻译成字节码文件？可能是因为直接翻译的代码是比较大的

解释器真正意义上所承担的角色就是一个运行时“翻译者”，将字节码文件中的内容“翻译”为对应平台的本地机器指令执行。

当一条字节码指令被解释执行完成后，接着再根据 PC 寄存器中记录的下一条需要被执行的字节码指令执行解释操作。

解释器分类

在 Java 的发展历史里，一共有两套解释执行器，即古老的字节码解释器、现在普遍使用的模板解释器。

字节码解释器在执行时通过纯软件代码模拟字节码的执行，效率非常低下。

而模板解释器将每一条字节码和一个模板函数相关联，模板函数中直接产生这条字节码执行时的机器码，从而很大程度上提高了解释器的性能。

在 HotSpot VM 中，解释器主要由 Interpreter 模块和 Code 模块构成。

- Interpreter 模块：实现了解释器的核心功能
- Code 模块：用于管理 HotSpot VM 在运行时生成的本地机器指令

现状

由于解释器在设计和实现上非常简单，因此除了 Java 语言之外，还有许多高级语言同样也是基于解释器执行的，比如 Python、Perl、Ruby 等。但是在今天，基于解释器执行已经沦落为低效的代名词，并且时常被一些 C/C++ 程序员所调侃。

为了解决这个问题，JVM 平台支持一种叫作即时编译的技术。即时编译的目的是避免函数被解释执行，而是将整个函数体编译成为机器码，每次函数执行时，只执行编译后的机器码即可，这种方式可以使执行效率大幅度提升。

不过无论如何，基于解释器的执行模式仍然为中间语言的发展做出了不可磨灭的贡献。

JIT 编译器

Java 代码的执行分类

第一种是将源代码编译成字节码文件，然后在运行时通过解释器将字节码文件转为机器码执行

第二种是编译执行（直接编译成机器码）。现代虚拟机为了提高执行效率，会使用即时编译技术（JIT，Just In Time）将方法编译成机器码后再执行

HotSpot VM 是目前市面上高性能虚拟机的代表作之一。它采用解释器与即时编译器并存的架构。在 Java 虚拟机运行时，解释器和即时编译器能够相互协作，各自取长补短，尽力去选择最合适的方式来权衡编译本地代码的时间和直接解释执行代码的时间。

在今天，Java 程序的运行性能早已脱胎换骨，已经达到了可以和 C/C++ 程序一较高下的地步。

问题来了

有些开发人员会感觉到诧异，既然 HotSpot VM 中已经内置 JIT 编译器了，那么为什么还需要再使用解释器来“拖累”程序的执行性能呢？比如 JRockit VM 内部就不包含解释器，字节码全部都依靠即时编译器编译后执行。

- JRockit 虚拟机是砍掉了解释器，也就是只采及时编译器。那是因为 JRockit 只部署在服务器上，一般已经有时间让他进行指令编译的过程了，对于响应来说要求不高，等及时编译器的编译完成后，就会提供更好的性能

首先明确：当程序启动后，解释器可以马上发挥作用，省去编译的时间，立即执行。编译器要想发挥作用，把代码编译成本地代码，需要一定的执行时间。但编译为本地代码后，执行效率高。

服务器端更关注性能。
客户端更关注响应时间

所以：尽管 JRockit VM 中程序的执行性能会非常高效，但程序在启动时必然需要花费更长的时间来进行编译。对于服务端应用来说，启动时间并非是关注重点，但对于那些看中启动时间的应用场景而言，或许就需要采用解释器与即时编译器并存的架构来换取一个平衡点。

在此模式下，当 Java 虚拟机启动时，解释器可以首先发挥作用，而不必等待即时编译器全部编译完成后再执行，这样可以省去许多不必要的编译时间。随着时间的推移，编译器发挥作用，把越来越多的代码编译成本地代码，获得更高的执行效率。

同时，解释执行在编译器进行激进优化不成立的时候，作为编译器的“逃生门”。

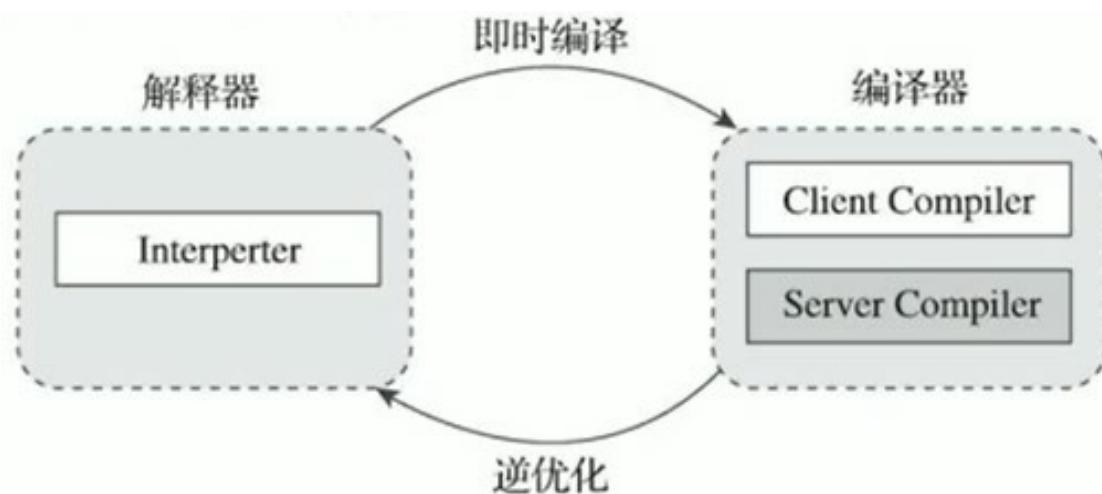
HotSpot JVM 执行方式

当虚拟机启动的时候，解释器可以首先发挥作用，而不必等待即时编译器全部编译完成再执行，这样可以省去许多不必要的编译时间。并且随着程序运行时间的推移，即时编译器逐渐发挥作用，根据热点探测功能，将有价值的字节码编译为本地机器指令，以换取更高的程序执行效率。

案例

注意解释执行与编译执行在线上环境微妙的辩证关系。机器在热机状态可以承受的负载要大于冷机状态。如果以热机状态时的流量进行切流，可能使处于冷机状态的服务器因无法承载流量而假死。

在生产环境发布过程中，以分批的方式进行发布，根据机器数量划分成多个批次，每个批次的机器数至多占到整个集群的 $1/8$ 。曾经有这样的故障案例：某程序员在发布平台进行分批发布，在输入发布总批数时，误填写成为两批发布。如果是热机状态，在正常情况下一半的机器可以勉强承载流量，但由于刚启动的 JVM 均是解释执行，还没有进行热点代码统计和 JIT 动态编译，导致机器启动之后，当前 $1/2$ 发布成功的服务器马上全部宕机，此故障说明了 JIT 的存在。—阿里团队



概念解释

- Java 语言的“编译期”其实是一段“不确定”的操作过程，因为它可能是指一个前端编译器（其实叫“编译器的前端”更准确一些）把.java 文件转变成.class 文件的过程；也可能是指虚拟机的后端运行期编译器（JIT 编译器，Just In Time Compiler）
- 把字节码转变成机器码的过程。
- 还可能是指使用静态提前编译器（AOT 编译器，Ahead of Time Compiler）直接把.java 文件编译成本地机器代码的过程。

前端编译器：Sun 的 Javac、Eclipse JDT 中的增量式编译器（ECJ）。

JIT 编译器：HotSpot VM 的 C1、C2 编译器。

AOT 编译器：GNU Compiler for the Java（GCJ）、Excelsior JET。

热点探测技术

一个被多次调用的方法，或者是一个方法体内部循环次数较多的循环体都可以被称之为“热点代码”，因此都可以通过 JIT 编译器编译为本地机器指令。由于这种编译方式发生在方法的执行过程中，因此被称之为栈上替换，或简称为 OSR（On Stack Replacement）编译。

一个方法究竟要被调用多少次，或者一个循环体究竟需要执行多少次循环才可以达到这个标准？必然需要一个明确的阈值，JIT 编译器才会将这些“热点代码”编译为本地机器指令执行。这里主要依靠热点探测功能。

目前 HotSpot VM 所采用的热点探测方式是基于计数器的热点探测。

采用基于计数器的热点探测，HotSpot V 将会为每一个方法都建立 2 个不同类型的计数器，分别为方法调用计数器（Invocation Counter）和回边计数器（Back Edge Counter）。

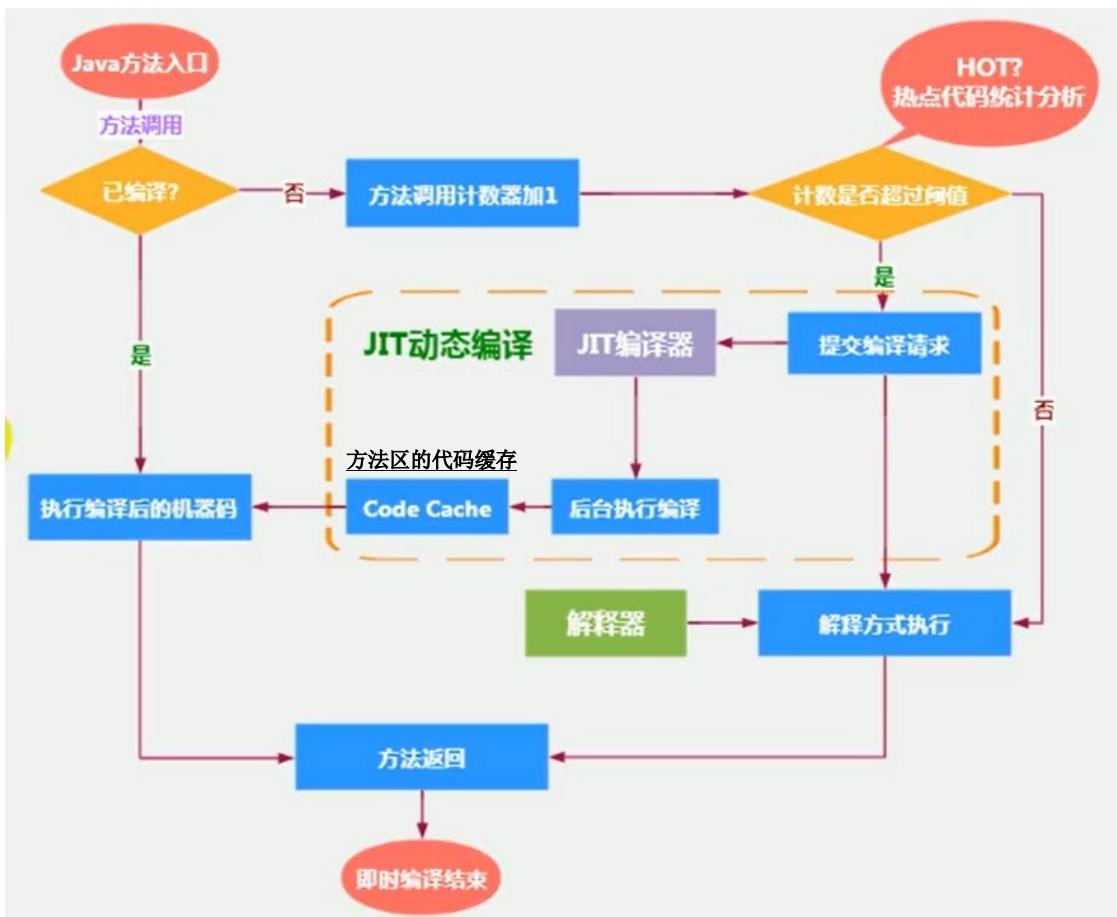
- 方法调用计数器用于统计方法的调用次数
- 回边计数器则用于统计循环体执行的循环次数

方法调用计数器

这个计数器就用于统计方法被调用的次数，它的默认阈值在 Client 模式下是 1500 次，在 Server 模式下是 10000 次。超过这个阈值，就会触发 JIT 编译。

这个阈值可以通过虚拟机参数 -XX:CompileThreshold 来人为设定。

当一个方法被调用时，会先检查该方法是否存在被 JIT 编译过的版本，如果存在，则优先使用编译后的本地代码来执行。如果不存在已被编译过的版本，则将此方法的调用计数器值加 1，然后判断方法调用计数器与回边计数器值之和是否超过方法调用计数器的阈值。如果已超过阈值，那么将会向即时编译器提交一个该方法的代码编译请求。



热点衰减

如果不做任何设置，方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间之内方法被调用的次数。当超过一定的时间限度，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器热度的衰减（Counter Decay），而这段时间就称为此方法统计的半衰周期（Counter Half Life Time）

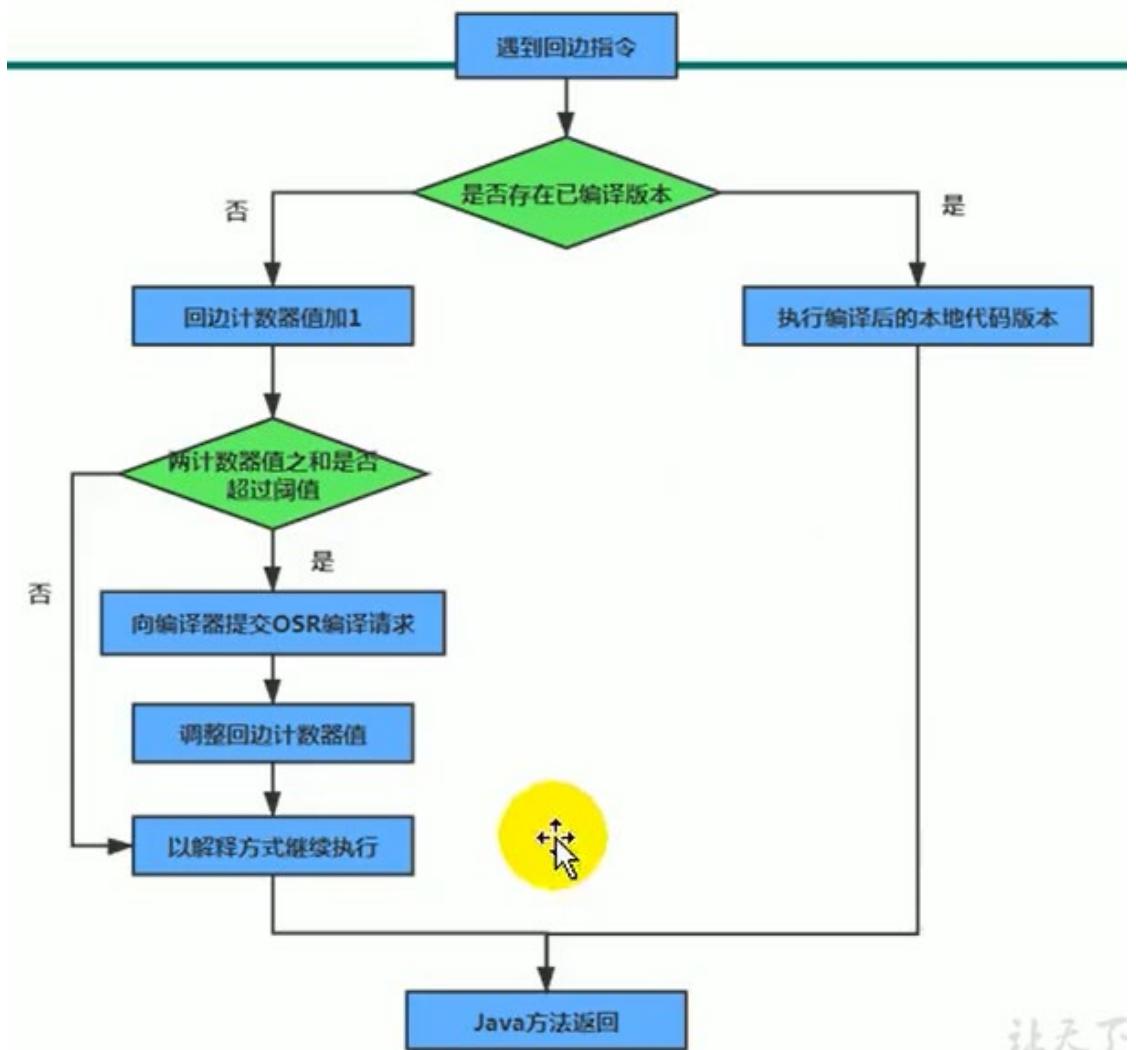
- 半衰周期是化学中的概念，比如出土的文物通过查看 C60 来获得文物的年龄

进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的，可以使用虚拟机参数 `-XX:-UseCounterDecay` 来关闭热度衰减，让方法计数器统计方法调用的绝对次数，这样，只要系统运行时间足够长，绝大部分方法都会被编译成本地代码。

另外，可以使用 `-XX:CounterHalfLifeTime` 参数设置半衰周期的时间，单位是秒。

回边计数器

它的作用是统计一个方法中循环体代码执行的次数，在字节码中遇到控制流向后跳转的指令称为“回边”（Back Edge）。显然，建立回边计数器统计的目的就是为了触发 OSR 编译。



让天下

HotSpotVM 可以设置程序执行方法

缺省情况下 HotSpot VM 是采用解释器与即时编译器并存的架构，当然开发人员可以根据具体的应用场景，通过命令显式地为 Java 虚拟机指定在运行时到底是完全采用解释器执行，还是完全采用即时编译器执行。如下所示：

- -Xint：完全采用解释器模式执行程序；

- **-Xcomp:** 完全采用即时编译器模式执行程序。如果即时编译出现问题，解释器会介入执行
- **-Xmixed:** 采用解释器+即时编译器的混合模式共同执行程序。

```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 © 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)

C:\Users\Administrator>java -Xint -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, interpreted mode)

C:\Users\Administrator>java -Xcomp -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, compiled mode)

C:\Users\Administrator>java -Xmixed -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode) →

C:\Users\Administrator>
```

HotSpotVM 中 JIT 分类

JIT 的编译器还分为了两种，分别是 C1 和 C2，在 HotSpot VM 中内嵌有两个 JIT 编译器，分别为 Client Compiler 和 Server Compiler，但大多数情况下我们简称为 C1 编译器和 C2 编译器。开发人员可以通过如下命令显式指定 Java 虚拟机在运行时到底使用哪一种即时编译器，如下所示：

- **-client:** 指定 Java 虚拟机运行在 Client 模式下，并使用 C1 编译器；
 - C1 编译器会对字节码进行简单和可靠的优化，耗时短。以达到更快的编译速度。
- **-server:** 指定 Java 虚拟机运行在 server 模式下，并使用 C2 编译器。
 - C2 进行耗时较长的优化，以及激进优化。但优化的代码执行效率更高。（使用 C++），64位系统默认server模式，改不成client模式。

C1 和 C2 编译器不同的优化策略

在不同的编译器上有不同的优化策略，C1 编译器上主要有方法内联，去虚拟化、冗余消除。

- 方法内联：将引用的函数代码编译到引用点处，这样可以减少栈帧的生成，减少参数传递以及跳转过程
- 去虚拟化：对唯一的实现类进行内联
- 冗余消除：在运行期间把一些不会执行的代码折叠掉

C2 的优化主要是在全局层面，逃逸分析是优化的基础。基于逃逸分析在 C2 上有如下几种优化：

- 标量替换：用标量值代替聚合对象的属性值
- 栈上分配：对于未逃逸的对象分配对象在栈而不是堆
- 同步消除：清除同步操作，通常指 `synchronized`

分层编译策略

分层编译（Tiered Compilation）策略：程序解释执行（不开启性能监控）可以触发 C1 编译，将字节码编译成机器码，可以进行简单优化，也可以加上性能监控，C2 编译会根据性能监控信息进行激进优化。

不过在 Java7 版本之后，一旦开发人员在程序中显式指定命令“-server”时，默认将会开启分层编译策略，由 C1 编译器和 C2 编译器相互协作共同来执行编译任务。

总结

- 一般来讲，JIT 编译出来的机器码性能比解释器高
- C2 编译器启动时长比 C1 慢，系统稳定执行以后，C2 编译器执行速度远快于 C1 编译器

AOT 编译器

jdk9 引入了 AOT 编译器（静态提前编译器，Ahead of Time Compiler）

Java 9 引入了实验性 AOT 编译工具 jaotc。它借助了 Graal 编译器，将所输入的 Java 类文件转换为机器码，并存放至生成的动态共享库之中。

所谓 AOT 编译，是与即时编译相对立的一个概念。我们知道，即时编译指的是在程序的运行过程中，将字节码转换为可在硬件上直接运行的机器码，并部署至托管环境中的过程。而 AOT 编译指的则是，在程序运行之前，便将字节码转换为机器码的过程。

.java -> .class -> (使用 jaotc) -> .so

最大的好处：Java 虚拟机加载已经预编译成二进制库，可以直接执行。不必等待及时编译器的预热，减少 Java 应用给人带来“第一次运行慢”的不良体验

缺点：

- 破坏了 java “一次编译，到处运行”，必须为每个不同的硬件，OS 编译对应的发行包
- 降低了 Java 链接过程的动态性，加载的代码在编译器就必须全部已知。
- 还需要继续优化中，最初只支持 Linux X64 java base

写到最后

- 自 JDK10 起，HotSpot 又加入了一个全新的及时编译器：Graal 编译器
- 编译效果短短几年时间就追平了 G2 编译器，未来可期
- 目前，带着实验状态标签，需要使用开关参数去激活才能使用
`-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler`

StringTable

String 的基本特性

- String：字符串，使用一对""起来表示
 - `String s1 = "atguigu";` // 字面量的定义方式
 - `String s2 = new String("atguigu");`
- String 被声明为 final，不可被继承
- String 实现了 Serializable 接口：表示字符串是支持序列化的。跨进程通信String对象无压力
- 实现了 Comparable 接口：表示 string 可以比较大小
- String 在 jdk8 及以前内部定义了 final char[] value 用于存储字符串数据。JDK9 时改为 byte[]

为什么 JDK9 改变了结构

String 类的当前实现将字符存储在 char 数组中，每个字符使用两个字节(16 位)。从许多不同的应用程序收集的数据表明，字符串是堆使用的主要组成部分，而且，

大多数字符串对象只包含拉丁字符。这些字符只需要一个字节的存储空间，因此这些字符串对象的内部 char 数组中有一半的空间将不会使用。

我们建议改变字符串的内部表示 class 从 utf-16 字符数组到字节数组+一个 encoding-flag 字段。新的 String 类将根据字符串的内容存储编码为 ISO-8859-1/Latin-1(每个字符一个字节)或 UTF-16(每个字符两个字节)的字符。编码标志将指示使用哪种编码。

结论：String 再也不用 char[] 来存储了，改成了 byte [] 加上编码标记，节约了一些空间

```
// 之前  
private final char value[];  
// 之后  
private final byte[] value
```

同时基于 String 的数据结构，例如 StringBuffer 和 StringBuilder 也同样做了修改

String 的不可变性

String：代表不可变的字符序列。简称：不可变性。

当对字符串重新赋值时，需要重写指定内存区域赋值，不能使用原有的 value 进行赋值。当对现有的字符串进行连接操作时，也需要重新指定内存区域赋值，不能使用原有的 value 进行赋值。当调用 string 的 replace () 方法修改指定字符或字符串时，也需要重新指定内存区域赋值，不能使用原有的 value 进行赋值。通过字面量的方式（区别于 new）给一个字符串赋值，此时的字符串值声明在字符串常量池中。

代码

```
/**  
 * String 的不可变性  
 *  
 * @author: 陌溪  
 * @create: 2020-07-11-8:57  
 */  
public class StringTest1 {  
  
    public static void test1() {  
        // 字面量定义的方式，“abc”存储在字符串常量池中  
        String s1 = "abc";  
        String s2 = "abc";  
        System.out.println(s1 == s2); // 这里的等于判断的是地址是否相等  
        s1 = "hello";  
        System.out.println(s1 == s2);  
        System.out.println(s1);  
        System.out.println(s2);  
        System.out.println("-----");  
    }  
}
```

```

public static void test2() {
    String s1 = "abc";
    String s2 = "abc";
    // 只要进行了修改，就会重新创建一个对象，这就是不可变性
    s2 += "def";
    System.out.println(s1);
    System.out.println(s2);
    System.out.println("-----");
}

public static void test3() {
    String s1 = "abc";
    String s2 = s1.replace('a', 'm');
    System.out.println(s1);
    System.out.println(s2);
}

public static void main(String[] args) {
    test1();
    test2();
    test3();
}
}

```

运行结果

```

true
false
hello
abc
-----
abc
abcdef
-----
abc
mbc

```

面试题

```

/**
 * 面试题
 *
 * @author: 陌溪
 * @create: 2020-07-11-9:05
 */
public class StringExer {
    String str = new String("good");
    char [] ch = {'t','e','s','t'};

```

```

public void change(String str, char ch []) {
    str = "test ok";
    ch[0] = 'b';
}

public static void main(String[] args) {
    StringExer ex = new StringExer();
    ex.change(ex.str, ex.ch);
    System.out.println(ex.str);
    System.out.println(ex.ch);
}
}

```

输出结果

```
good
best
```

注意

HashTable底层就是数组（导致了固定大小的特征）+链表的结构，String的底层是HashMap，哈希表的key是Set，不能重复

字符串常量池是不会存储相同内容的字符串的

注意：默认大小是指对应的数组的大小，如果字符串越来越多，则在后面的链表处添加

String 的 string Pool 是一个固定大小的 Hashtable，默认值大小长度是 1009。如果放进 string Pool 的 string 非常多，就会造成 Hash 冲突严重，从而导致链表会很长，而链表长了后直接会造成的影响就是当调用 string.intern 时性能会大幅下降。

使用-XX:StringTablesize 可设置 stringTable 的长度

在 jdk6 中 stringTable 是固定的，就是 1009 的长度，所以如果常量池中的字符串过多就会导致效率下降很快。stringTablesize 设置没有要求。

在 jdk7 中，stringTable 的长度默认值是 60013。

String.intern():如果String在字符串常量池中不存在，则将String加入到字符串常量池中，否则将String指向字符串在字符串常量池中的地址。

在 JDK8 中，StringTable 可以设置的最小值为 1009。

String 的内存分配

在 Java 语言中有 8 种基本数据类型和一种比较特殊的类型 string。这些类型为了使它们在运行过程中速度更快、更节省内存，都提供了一种常量池的概念。

常量池就类似一个 Java 系统级别提供的缓存。8 种基本数据类型的常量池都是系统协调的，string 类型的常量池比较特殊。它的主要使用方法有两种。

直接使用双引号声明出来的 String 对象会直接存储在常量池中。

- 比如：string info="atguigu.com";

如果不是用双引号声明的 string 对象，可以使用 string 提供的 intern() 方法。

Java中只有值引用，也就是将传入方法的参数复制一份后传入方法。对于ex对象，传递的是ex对象引用的复制，也就是ex对象的物理地址，这样在修改的时候我们发现还是针对ex对象本身进行了修改，我们可能会误以为这就是引用传递的例子，然而在这个例子中我们真实传入的实参是ex对象的地址，实参本身没有改变！对于String对象，我们传入的也是原“good”的引用，但是由于字符串的不可变性，导致change过程中的“=”实际上就是修改了地址。

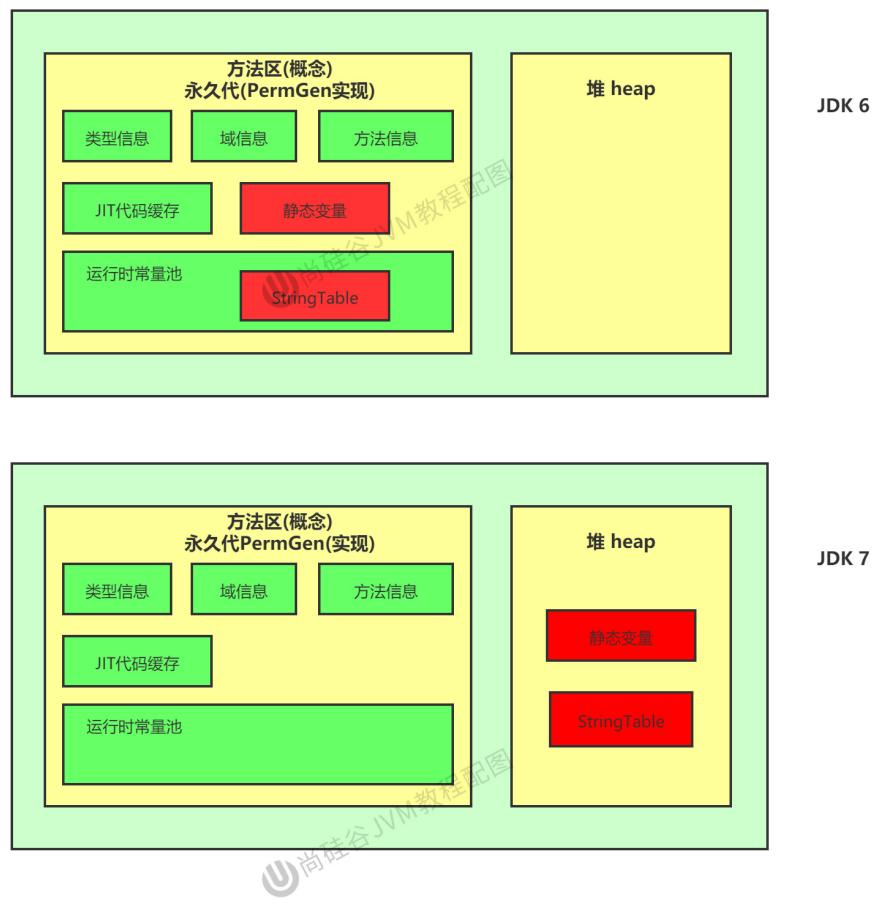
Java 6 及以前，字符串常量池存放在永久代

Java 7 中 oracle 的工程师对字符串池的逻辑做了很大的改变，即将字符串常量池的位置调整到 Java 堆内

所有的字符串都保存在堆（Heap）中，和其他普通对象一样，这样可以让你在进行调优应用时仅需要调整堆大小就可以了。

字符串常量池概念原本使用得比较多，但是这个改动使得我们有足够的理由让我们重新考虑在 Java 7 中使用 `String.intern()`。

Java8 元空间，字符串常量在堆



为什么 StringTable 从永久代调整到堆中

在 JDK 7 中，interned 字符串不再在 Java 堆的永久生成中分配，而是在 Java 堆的主要部分(称为年轻代和年老代)中分配，与应用程序创建的其他对象一起分配。此更改将导致驻留在主 Java 堆中的数据更多，驻留在永久生成中的数据更少，因此可能需要调整堆大小。由于这一变化，大多数应用程序在堆使用方面只会看到相对较小的差异，但加载许多类或大量使用字符串的较大应用程序会出现这种差异。`intern()`方法会看到更显著的差异。

- 永久代的默认Size比较小
- 永久代垃圾回收频率低

String 的基本操作

Java 语言规范里要求完全相同的字符串字面量，应该包含同样的 Unicode 字符序列（包含同一份码点序列的常量），并且必须是指向同一个 String 类实例。

字符串拼接操作

- 常量与常量的拼接结果在常量池，原理是编译期优化
- 常量池中不会存在相同内容的变量
- 只要其中有一个是变量，结果就在堆中。变量拼接的原理是 `StringBuilder`
- 如果拼接的结果调用 `intern()` 方法，则主动将常量池中还没有的字符串对象放入池中，并返回此对象地址

```
public static void test1() {  
    String s1 = "a" + "b" + "c"; // 得到 abc 的常量池  
    String s2 = "abc"; // abc 存放在常量池，直接将常量池的地址返回  
    /**  
     * 最终 java 编译成.class，再执行.class  
     */  
    System.out.println(s1 == s2); // true，因为存放在字符串常量池  
    System.out.println(s1.equals(s2)); // true  
}  
  
public static void test2() {  
    String s1 = "javaEE";  
    String s2 = "hadoop";  
    String s3 = "javaEEhadoop";  
    String s4 = "javaEE" + "hadoop";  
    String s5 = s1 + "hadoop";  
    String s6 = "javaEE" + s2;  
    String s7 = s1 + s2;
```

```

        System.out.println(s3 == s4); // true
        System.out.println(s3 == s5); // false
        System.out.println(s3 == s6); // false
        System.out.println(s3 == s7); // false
        System.out.println(s5 == s6); // false
        System.out.println(s5 == s7); // false
        System.out.println(s6 == s7); // false

        String s8 = s6.intern();
        System.out.println(s3 == s8); // true
    }

```

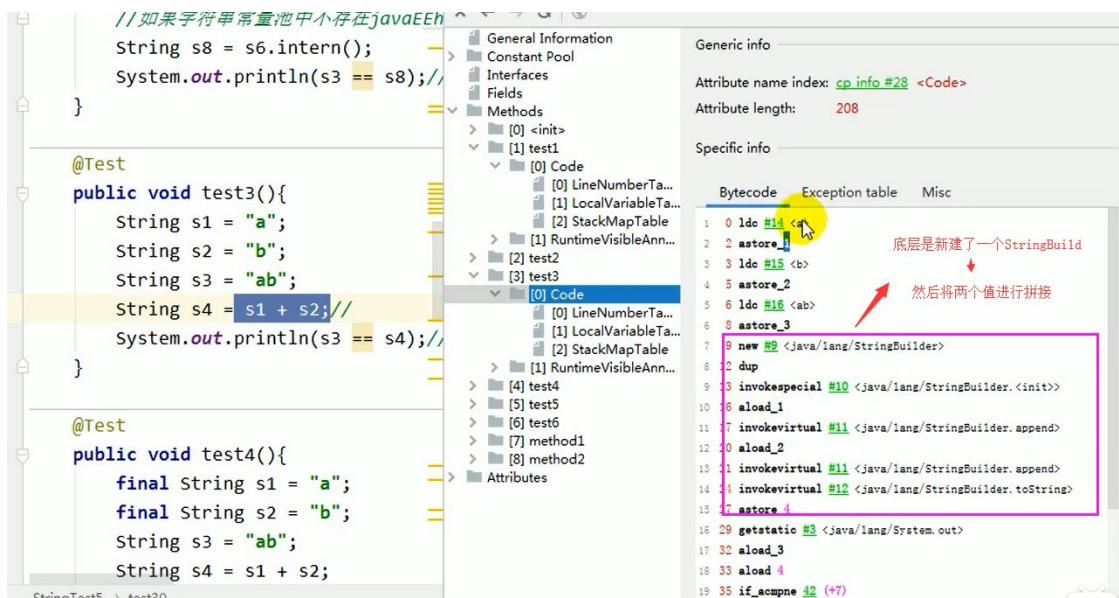
从上述的结果我们可以知道：

如果拼接符号的前后出现了变量，则相当于在堆空间中 new String(), 具体的内容为拼接的结果

而调用 intern 方法，则会判断字符串常量池中是否存在 JavaEEhadoop 值，如果存在则返回常量池中的地址值，否则就在常量池中创建

底层原理

拼接操作的底层其实使用了 StringBuilder



s1 + s2 的执行细节

- `StringBuilder s = new StringBuilder();`
- `s.append(s1);`

- `s.append(s2);`
- `s.toString(); -> 类似于 new String("ab");`

在 JDK5 之后，使用的是 `StringBuilder`，在 JDK5 之前使用的是 `StringBuffer`

<code>String</code>	<code>StringBuffer</code>	<code>StringBuilder</code>
<code>String</code> 的值是不可变的，这就导致每次对 <code>String</code> 的操作都会生成新的 <code>String</code> 对象，不仅效率低下，而且浪费大量优先的内存空间	<code>StringBuffer</code> 是可变类，和线程安全的字符串操作类，任何对它指向的字符串的操作都不会产生新的对象。每个 <code>StringBuffer</code> 对象都有一定的缓冲区容量，当字符串大小没有超过容量时，不会分配新的容量，当字符串大小超过容量时，会自动增加容量	可变类，速度更快
不可变	可变 线程安全 多线程操作字符串	可变 线程不安全 单线程操作字符串

注意，我们左右两边如果是变量的话，就是需要 `new StringBuilder` 进行拼接，但是如果使用的是 `final` 修饰，则是从常量池中获取。所以说拼接符号左右两边都是字符串常量或常量引用则仍然使用编译器优化。也就是说被 `final` 修饰的变量，将会变成常量，类和方法将不能被继承、

- 在开发中，能够使用 `final` 的时候，建议使用上

```
public static void test4() {
    final String s1 = "a";
    final String s2 = "b";
    String s3 = "ab";
    String s4 = s1 + s2;
    System.out.println(s3 == s4);
} 如果拼接符号两边都是字符串常量或常量引用 (final)，则仍然使用编译期优化
```

运行结果

true

拼接操作和 `append` 性能对比

```
public static void method1(int highLevel) {
    String src = "";
    for (int i = 0; i < highLevel; i++)
    {
        src += "a"; // 每次循环都会创建一个StringBuilder 对象
    }
}
```

```
public static void method2(int highLevel) {  
    StringBuilder sb = new StringBuilder();  
    for (int i = 0; i < highLevel; i++) {  
        sb.append("a");  
    }  
}
```

方法 1 耗费的时间：4005ms，方法 2 消耗时间：7ms

结论：

- 通过 `StringBuilder` 的 `append()` 方式添加字符串的效率，要远远高于 `String` 的字符串拼接方法

好处

- `StringBuilder` 的 `append` 的方式，自始至终只创建一个 `StringBuilder` 的对象
- 对于字符串拼接的方式，还需要创建很多 `StringBuilder` 对象和调用 `toString` 时候创建的 `String` 对象
- 内存中由于创建了较多的 `StringBuilder` 和 `String` 对象，内存占用过大，如果进行 `GC` 那么将会耗费更多的时间

改进的空间

- 我们使用的是 `StringBuilder` 的空参构造器，默认的字符串容量是 16，然后将原来的字符串拷贝到新的字符串中，我们也可以默认初始化更大的长度，减少扩容的次数
- 因此在实际开发中，我们能够确定，前前后后需要添加的字符串不高于某个限值，那么建议使用构造器创建一个阈值的长度 `new StringBulider(100000)`

intern()的使用

`intern` 是一个 `native` 方法，调用的是底层 C 的方法

字符串池最初是空的，由 `String` 类私有地维护。在调用 `intern` 方法时，如果池中已经包含了由 `equals(object)` 方法确定的与该字符串对象相等的字符串，则返回池中的字符串。否则，该字符串对象将被添加到池中，并返回对该字符串对象的引用。

如果不是用双引号声明的 `string` 对象，可以使用 `string` 提供的 `intern` 方法：`intern` 方法会从字符串常量池中查询当前字符串是否存在，若不存在就会将当前字符串放入常量池中。

比如：

```
String myInfo = new string("I love atguigu").intern();
```

也就是说，如果在任意字符串上调用 `String.intern` 方法，那么其返回结果所指向的那个类实例，必须和直接以常量形式出现的字符串实例完全相同。因此，下列表达式的值必定是 `true`

```
( "a"+ "b"+ "c" ) .intern () == "abc"
```

通俗点讲，Interned string 就是确保字符串在内存里只有一份拷贝，这样可以节约内存空间，加快字符串操作任务的执行速度。注意，这个值会被存放在字符串内部池（String Intern Pool）

intern 的空间效率测试

我们通过测试一下，使用了 `intern` 和不使用的时候，其实相差还挺多的

```
/**  
 * 使用 Intern() 测试执行效率  
 * @author: 陌溪  
 * @create: 2020-07-11-15:19  
 */  
public class StringIntern2 {  
    static final int MAX_COUNT = 1000 * 10000;  
    static final String[] arr = new String[MAX_COUNT];  
  
    public static void main(String[] args) {  
        Integer [] data = new Integer[]{1,2,3,4,5,6,7,8,9,10};  
        long start = System.currentTimeMillis();  
        for (int i = 0; i < MAX_COUNT; i++) {  
            arr[i] = new String(String.valueOf(data[i%data.length])).in  
tern();  
        }  
        long end = System.currentTimeMillis();  
        System.out.println("花费的时间为: " + (end - start));  
  
        try {  
            Thread.sleep(1000000);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

结论：对于程序中大量使用存在的字符串时，尤其存在很多已经重复的字符串时，使用 `intern()` 方法能够节省内存空间。

大的网站平台，需要内存中存储大量的字符串。比如社交网站，很多人都存储：北京市、海淀区等信息。这时候如果字符串都调用 `intern()` 方法，就会很明显降低内存的大小。

面试题

`new String("ab")`会创建几个对象 2个

```
/*
 * new String("ab") 会创建几个对象? 看字节码就知道是 2 个对象
 *
 * @author: 陌溪
 * @create: 2020-07-11-11:17
 */
public class StringNewTest {
    public static void main(String[] args) {
        String str = new String("ab");
    }
}
```

我们转换成字节码来查看

```
0 new #2 <java/lang/String>
3 dup
4 ldc #3 <ab>
6 invokespecial #4 <java/lang/String.<init>>
9 astore_1
10 return
```

这里面就是两个对象

- 一个对象是：`new` 关键字在堆空间中创建
- 另一个对象：字符串常量池中的对象

`new String("a") + new String("b")` 会创建几个对象

```
/*
 * new String("ab") 会创建几个对象? 一共6个
 *
 * @author: 陌溪
 * @create: 2020-07-11-11:17
 */
public class StringNewTest {
    public static void main(String[] args) {
        String str = new String("a") + new String("b");
    }
}
```

字节码文件为

```
0 new #2 <java/lang/StringBuilder>
3 dup
4 invokespecial #3 <java/lang/StringBuilder.<init>>
7 new #4 <java/lang/String>
```

```
10 dup
11 ldc #5 <a>
13 invokespecial #6 <java/lang/String.<init>>
16 invokevirtual #7 <java/lang/StringBuilder.append>
19 new #4 <java/lang/String>
22 dup
23 ldc #8 <b>
25 invokespecial #6 <java/lang/String.<init>>
28 invokevirtual #7 <java/lang/StringBuilder.append>
31 invokevirtual #9 <java/lang/StringBuilder.toString>
34 astore_1
35 return
```

我们创建了 6 个对象

- 对象 1: new StringBuilder()
- 对象 2: new String("a")
- 对象 3: 常量池的 a
- 对象 4: new String("b")
- 对象 5: 常量池的 b
- 对象 6: toString 中会创建一个 new String("ab")
 - 调用 toString 方法, 不会在常量池中生成 ab

intern 的使用: JDK6 和 JDK7

JDK6 中

```
String s = new String("1"); // 在常量池中已经有了
s.intern(); // 将该对象放入到常量池。但是调用此方法没有太多的区别, 因为已经存在了1
String s2 = "1";
System.out.println(s == s2); // false

String s3 = new String("1") + new String("1");
s3.intern();
String s4 = "11";
System.out.println(s3 == s4); // true
```

输出结果

```
false
true
```

为什么对象会不一样呢?

- 一个是 new 创建的对象，一个是常量池中的对象，显然不是同一个
如果是下面这样的，那么就是 true

```
String s = new String("1");
s = s.intern();
String s2 = "1";
System.out.println(s == s2); // true
```

而对于下面的来说，因为 s3 变量记录的地址是 new String("11")，然后这段代码执行完以后，常量池中不存在 "11"，这是 JDK6 的关系，然后执行 s3.intern()后，就会在常量池中生成 "11"，最后 s4 用的就是 s3 的地址

为什么最后输出的 s3 == s4 会为 false 呢？

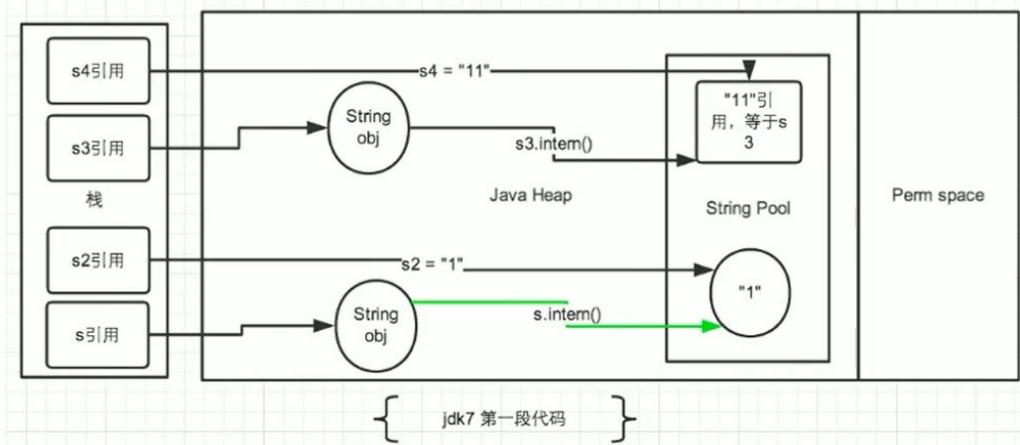
这是因为在 JDK6 中创建了一个新的对象 "11"，也就是有了新的地址， s2 = 新地址

而在 JDK7 中，在 JDK7 中，并没有创新一个新对象，而是指向常量池中的新对象。

JDK7 中

```
String s = new String("1");
s.intern();
String s2 = "1";
System.out.println(s == s2); // true

String s3 = new String("1") + new String("1");
s3.intern();      JDK7中为了节省堆空间，此intern的含义是在字符串常量池中记录一个地址，该地址指向之前创建过的new String("11")
String s4 = "11"; 记录一个地址，该地址指向之前创建过的new String("11")
System.out.println(s3 == s4); // true
```



扩展

```
String s3 = new String("1") + new String("1");
String s4 = "11"; // 在常量池中生成的字符串
```

```
s3.intern(); // 然后 s3 就会从常量池中找，发现有了，就什么事情都不做  
System.out.println(s3 == s4);
```

我们将 s4 的位置向上移动一行，发现变化就会很大，最后得到的是 false

总结

总结 string 的 intern () 的使用：

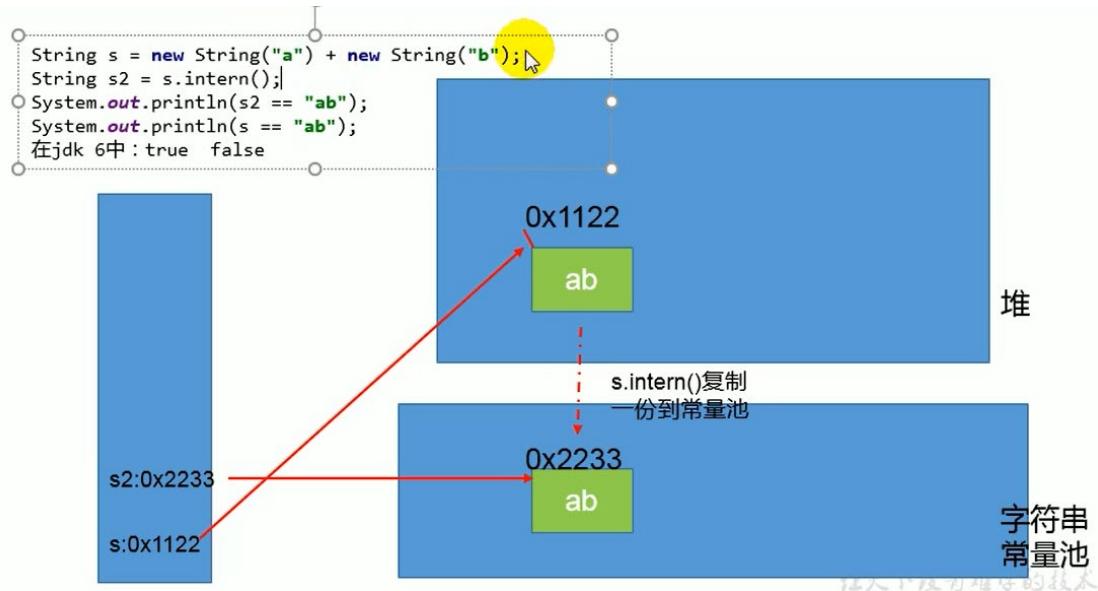
JDK1.6 中，将这个字符串对象尝试放入串池。

- 如果串池中有，则并不会放入。返回已有的串池中的对象的地址
- 如果没有，会把此**对象**复制一份，放入串池，并返回串池中的对象地址

JDK1.7 起，将这个字符串对象尝试放入串池。

- 如果串池中有，则并不会放入。返回已有的串池中的对象的地址
- 如果没有，则会把**对象的引用地址**复制一份，放入串池，并返回串池中的引用地址

练习：



- 在 JDK6 中，在字符串常量池中创建一个字符串 “ab”
- 在 JDK8 中，在字符串常量池中没有创建 “ab”，而是将堆中的地址复制到串池中。

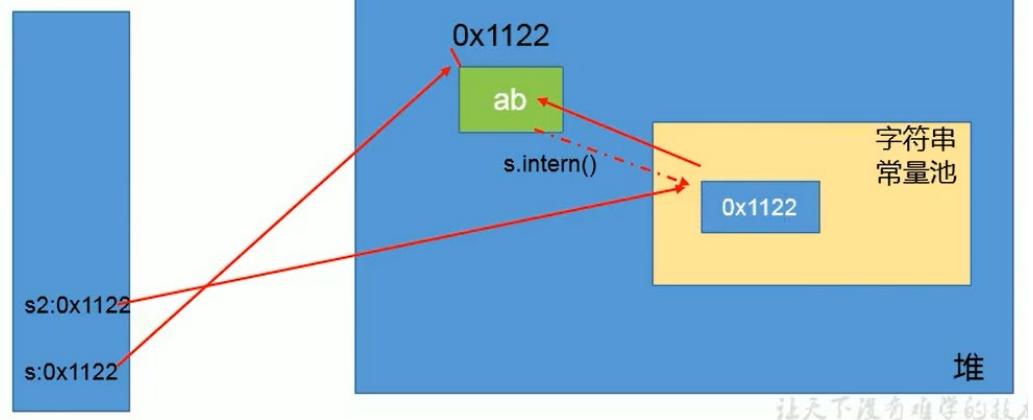
所以上述结果，在 JDK6 中是：

```
true  
false
```

在 JDK8 中是

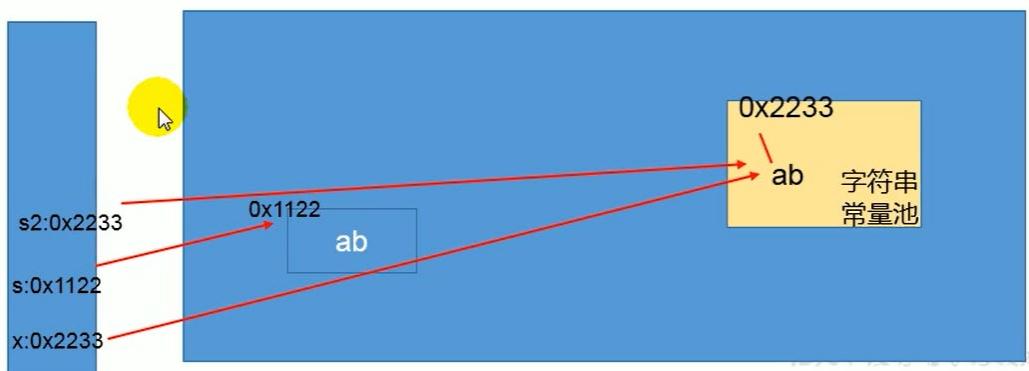
```
false  
true
```

```
String s = new String("a") + new String("b");  
String s2 = s.intern();  
System.out.println(s2 == "ab");  
System.out.println(s == "ab");  
在jdk 7,8中: true true
```



针对下面这题，在 JDK6 和 8 中表现的是一样的

```
String x = "ab";  
String s = new String("a") + new String("b");  
String s2 = s.intern(); //因为常量池已经有ab,则不会放入  
System.out.println(s2 == x);  
System.out.println(s == x);  
在jdk6,7,8中执行都是true false
```



StringTable 的垃圾回收

```
/*
 * String 的垃圾回收
 * -Xms15m -Xmx15m -XX:+PrintStringTableStatistics -XX:+PrintGCDetails
 * @author: 陌溪
 * @create: 2020-07-11-16:55
 */
public class StringGCTest {
    public static void main(String[] args) {
        for (int i = 0; i < 100000; i++) {
            String.valueOf(i).intern();
        }
    }
}
```

G1 中的 String 去重操作

注意这里说的重复，指的是在堆中的数据，而不是常量池中的，因为常量池中的本身就不会重复

描述

背景：对许多 Java 应用（有大的也有小的）做的测试得出以下结果：

- 堆存活数据集合里面 string 对象占了 25%
- 堆存活数据集合里面重复的 string 对象有 13.5%
- string 对象的平均长度是 45

许多大规模的 Java 应用的瓶颈在于内存，测试表明，在这些类型的应用里面，Java 堆中存活的数据集合差不多 25% 是 string 对象。更进一步，这里面差不多一半 string 对象是重复的，重复的意思是说：`string1.equals(string2) = true`。堆上存在重复的 string 对象必然是一种内存的浪费。这个项目将在 G1 垃圾收集器中实现自动持续对重复的 string 对象进行去重，这样就能避免浪费内存。

实现

- 当垃圾收集器工作的时候，会访问堆上存活的对象。对每一个访问的对象都会检查是否是候选的要去重的 string 对象。
- 如果是，把这个对象的一个引用插入到队列中等待后续的处理。一个去重的线程在后台运行，处理这个队列。处理队列的一个元素意味着从队列删除这个元素，然后尝试去重它引用的 string 对象。

- 使用一个 hashtable 来记录所有的被 string 对象使用的不重复的 char 数组。当去重的时候，会查这个 hashtable，来看堆上是否已经存在一个一模一样的 char 数组。
- 如果存在， string 对象会被调整引用那个数组，释放对原来的数组的引用，最终会被垃圾收集器回收掉。
- 如果查找失败， char 数组会被插入到 hashtable，这样以后的时候就可以共享这个数组了。

开启

命令行选项

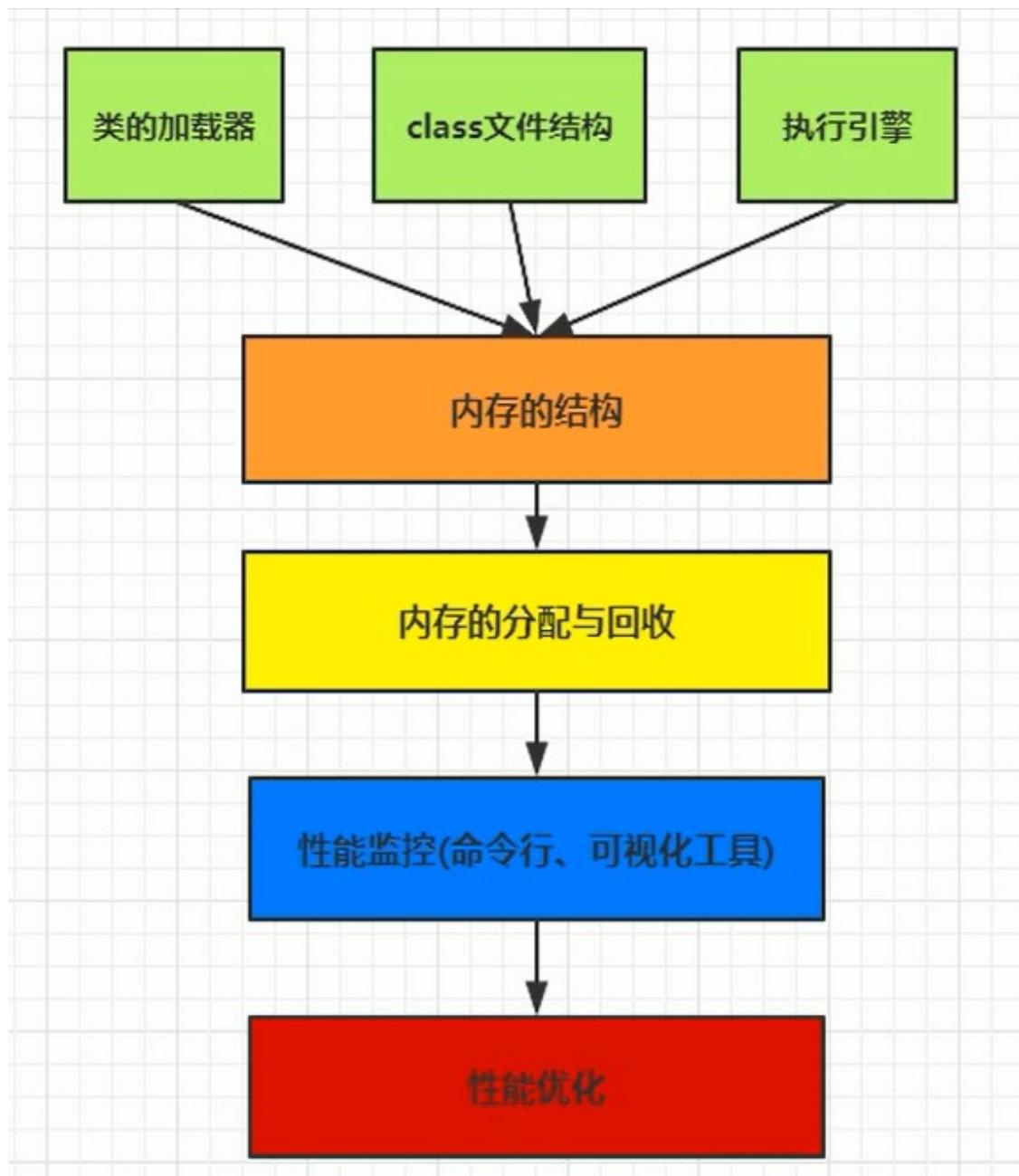
`UseStringDeduplication (bool)`：开启 string 去重，默认是不开启的，需要手动开启。

`PrintStringDeduplicationStatistics (bool)`：打印详细的去重统计信息
`StringDeduplicationAgeThreshold (uintx)`：达到这个年龄的 string 对象被认为是去重的候选对象

垃圾回收概述

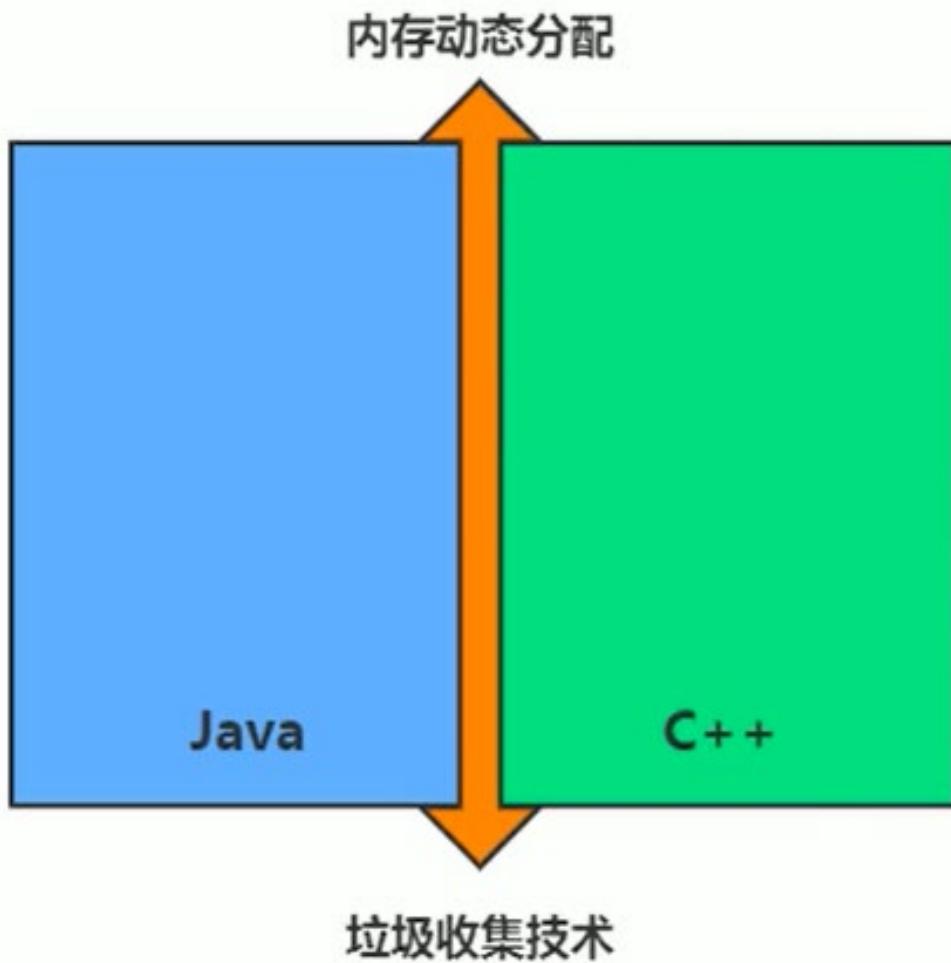
概念

这次我们主要关注的是黄色部分，内存的分配与回收



什么是垃圾

在提到什么是垃圾之前，我们先看下面一张图



从上图我们可以很明确的知道，Java 和 C++语言的区别，就在于垃圾收集技术和内存动态分配上，C 语言没有垃圾收集技术，需要我们手动的收集。

垃圾收集，不是 Java 语言的伴生产物。早在 1960 年，第一门开始使用内存动态分配和垃圾收集技术的 Lisp 语言诞生。关于垃圾收集有三个经典问题：

- 哪些内存需要回收？
- 什么时候回收？
- 如何回收？

垃圾收集机制是 Java 的招牌能力，极大地提高了开发效率。如今，垃圾收集几乎成为现代语言的标配，即使经过如此长时间的发展，Java 的垃圾收集机制仍然在

不断的演进中，不同大小的设备、不同特征的应用场景，对垃圾收集提出了新的挑战，这当然也是面试的热点。

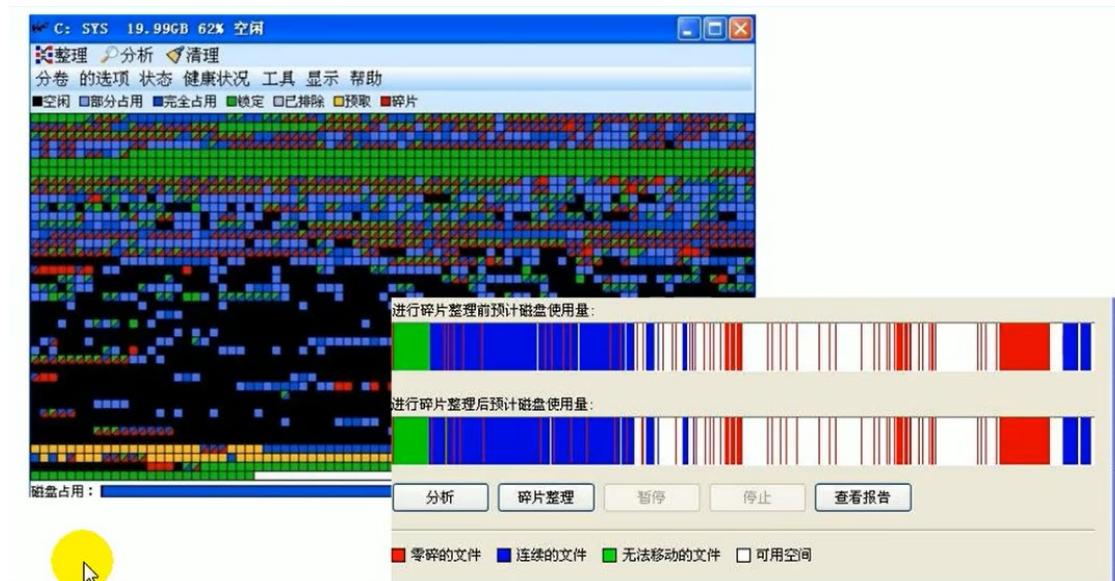
什么是垃圾？

垃圾是指在运行程序中没有任何指针指向的对象，这个对象就是需要被回收的垃圾。

如果不及时对内存中的垃圾进行清理，那么，这些垃圾对象所占的内存空间会一直保留到应用程序的结束，被保留的空间无法被其它对象使用，甚至可能导致内存溢出。

磁盘碎片整理

机械硬盘需要进行磁盘整理，同时还有坏道



大厂面试题

蚂蚁金服

- 你知道哪几种垃圾回收器，各自的优缺点，重点讲一下 CMS 和 G1？
- JVM GC 算法有哪些，目前的 JDK 版本采用什么回收算法？
- G1 回收器讲下回收过程 GC 是什么？为什么要有 GC？
- GC 的两种判定方法？CMS 收集器与 G1 收集器的特点

百度

- 说一下 GC 算法，分代回收说下

- 垃圾收集策略和算法

天猫

- JVM GC 原理，JVM 怎么回收内存
- CMS 特点，垃圾回收算法有哪些？各自的优缺点，他们共同的缺点是什么？

滴滴

Java 的垃圾回收器都有哪些，说下 g1 的应用场景，平时你是如何搭配使用垃圾回收器的

京东

- 你知道哪几种垃圾收集器，各自的优缺点，重点讲下 cms 和 G1，
- 包括原理，流程，优缺点。垃圾回收算法的实现原理

阿里

- 讲一讲垃圾回收算法。
- 什么情况下触发垃圾回收？
- 如何选择合适的垃圾收集算法？
- JVM 有哪三种垃圾回收器？

字节跳动

- 常见的垃圾回收器算法有哪些，各有什么优劣？
- System.gc () 和 Runtime.gc () 会做什么事情？
- Java GC 机制？GC Roots 有哪些？
- Java 对象的回收方式，回收算法。
- CMS 和 G1 了解么，CMS 解决什么问题，说一下回收的过程。
- CMS 回收停顿了几次，为什么要停顿两次？

为什么需要 GC

对于高级语言来说，一个基本认知是如果不进行垃圾回收，内存迟早都会被消耗完，因为不断地分配内存空间而不进行回收，就好像不停地生产生活垃圾而从来不打扫一样。

除了释放没用的对象，垃圾回收也可以清除内存里的记录碎片。碎片整理将所占用的堆内存移到堆的一端，以便 JVM 将整理出的内存分配给新的对象。

随着应用程序所应付的业务越来越庞大、复杂，用户越来越多，没有 GC 就不能保证应用程序的正常进行。而经常造成 STW 的 GC 又跟不上实际的需求，所以才会不断地尝试对 GC 进行优化。

早期垃圾回收

在早期的 C/C++时代，垃圾回收基本上是手工进行的。开发人员可以使用 new 关键字进行内存申请，并使用 delete 关键字进行内存释放。比如以下代码：

```
MibBridge *pBridge= new cmBaseGroupBridge();
//如果注册失败，使用Delete 释放该对象所占内存区域
if (pBridge->Register (kDestroy) !=NO_ERROR)
    delete pBridge;
```

这种方式可以灵活控制内存释放的时间，但是会给开发人员带来频繁申请和释放内存的管理负担。倘若有一处内存区间由于程序员编码的问题忘记被回收，那么就会产生内存泄漏，垃圾对象永远无法被清除，随着系统运行时间的不断增长，垃圾对象所耗内存可能持续上升，直到出现内存溢出并造成应用程序崩溃。

有了垃圾回收机制后，上述代码极有可能变成这样

```
MibBridge *pBridge=new cmBaseGroupBridge();
pBridge->Register(kDestroy);
```

现在，除了 Java 以外，C#、Python、Ruby 等语言都使用了自动垃圾回收的思想，也是未来发展趋势，可以说这种自动化的内存分配和来及回收方式已经成为了现代开发语言必备的标准。

Java 垃圾回收机制

优点

自动内存管理，无需开发人员手动参与内存的分配与回收，这样降低内存泄漏和内存溢出的风险

没有垃圾回收器，java 也会和 cpp 一样，各种悬垂指针，野指针，泄露问题让你头疼不已。

自动内存管理机制，将程序员从繁重的内存管理中释放出来，可以更专心地专注于业务开发

oracle 官网关于垃圾回收的介绍

<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html>

担忧

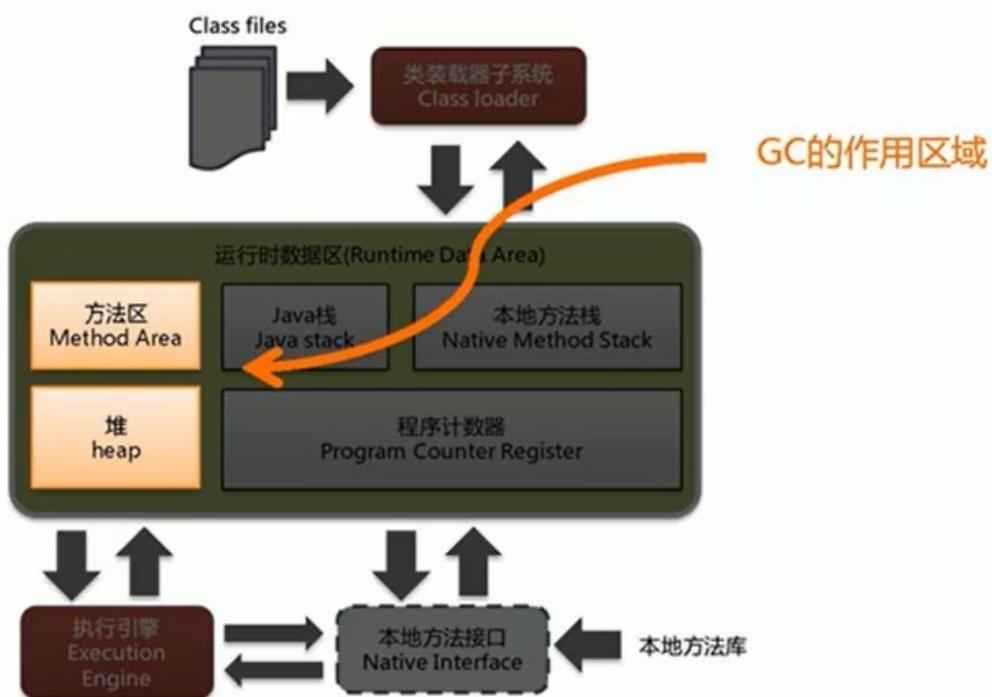
对于 Java 开发人员而言，自动内存管理就像是一个黑匣子，如果过度依赖于“自动”，那么这将会是一场灾难，最严重的就会弱化 Java 开发人员在程序出现内存溢出时定位问题和解决问题的能力。

此时，了解 JVM 的自动内存分配和内存回收原理就显得非常重要，只有在真正了解 JVM 是如何管理内存后，我们才能够在遇见 `outofMemoryError` 时，快速地根据错误异常日志定位问题和解决问题。

当需要排查各种内存溢出、内存泄漏问题时，当垃圾收集成为系统达到更高并发量的瓶颈时，我们就必须对这些“自动化”的技术实施必要的监控和调节。

GC 主要关注的区域

GC 主要关注于 方法区 和堆中的垃圾收集



垃圾收集器可以对年轻代回收，也可以对老年代回收，甚至是全栈和方法区的回收

- 其中，**Java 堆**是垃圾收集器的工作重点

从次数上讲：

- 频繁收集 Young 区
- 较少收集 Old 区
- 基本不收集 Perm 区（元空间）

垃圾回收相关算法

标记阶段：引用计数算法

在堆里存放着几乎所有的 Java 对象实例，在 GC 执行垃圾回收之前，首先需要区分出内存中哪些是存活对象，哪些是已经死亡的对象。只有被标记为已经死亡的对象，GC 才会在执行垃圾回收时，释放掉其所占用的内存空间，因此这个过程我们可以称为垃圾标记阶段。

那么在 JVM 中究竟是如何标记一个死亡对象呢？简单来说，当一个对象已经不再被任何的存活对象继续引用时，就可以宣判为已经死亡。

判断对象存活一般有两种方式：引用计数算法和可达性分析算法。

引用计数算法（Reference Counting）比较简单，对每个对象保存一个整型的引用计数器属性。用于记录对象被引用的情况。

对于一个对象 A，只要有任何一个对象引用了 A，则 A 的引用计数器就加 1；当引用失效时，引用计数器就减 1。只要对象 A 的引用计数器的值为 0，即表示对象 A 不可能再被使用，可进行回收。

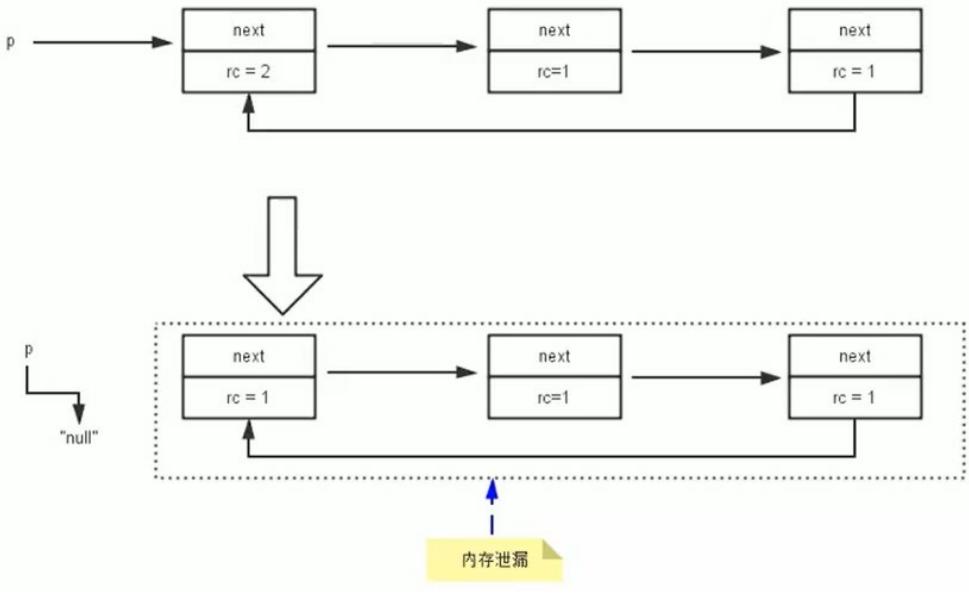
优点：实现简单，垃圾对象便于辨识；判定效率高，回收没有延迟性。

缺点：它需要单独的字段存储计数器，这样的做法增加了存储空间的开销。

每次赋值都需要更新计数器，伴随着加法和减法操作，这增加了时间开销。引用计数器有一个严重的问题，即无法处理循环引用的情况。这是一条致命缺陷，导致在 Java 的垃圾回收器中没有使用这类算法。

循环引用

当 p 的指针断开的时候，内部的引用形成一个循环，这就是循环引用，从而造成内存泄漏



举例

我们使用一个案例来测试 Java 中是否采用的是引用计数算法

```
/**
 * 引用计数算法测试
 *
 * @author: 陌溪
 * @create: 2020-07-12-10:26
 */
public class RefCountGC {
    // 这个成员属性的唯一作用就是占用一点内存
    private byte[] bigSize = new byte[5*1024*1024];
    // 引用
    Object reference = null;

    public static void main(String[] args) {
        RefCountGC obj1 = new RefCountGC();
        RefCountGC obj2 = new RefCountGC();
        obj1.reference = obj2;
        obj2.reference = obj1;
        obj1 = null;
        obj2 = null;
        // 显示的执行垃圾收集行为，判断obj1 和 obj2 是否被回收？
        System.gc();
    }
}
```

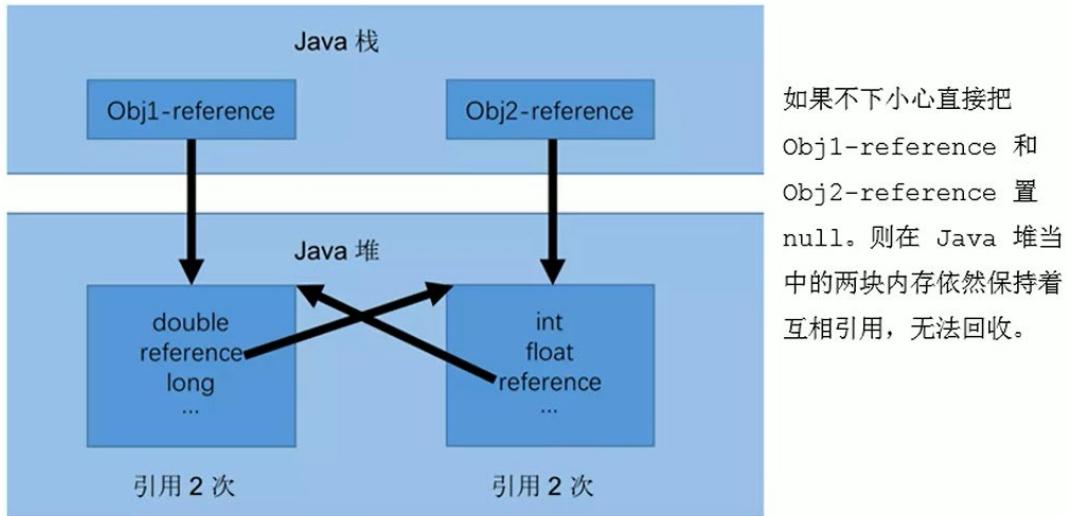
运行结果

```
[GC (System.gc()) [PSYoungGen: 15490K->808K(76288K)] 15490K->816K(25139
2K), 0.0061980 secs] [Times: user=0.00 sys=0.00, real=0.36 secs]
[Full GC (System.gc()) [PSYoungGen: 808K->0K(76288K)] [ParOldGen: 8K->6
72K(175104K)] 816K->672K(251392K), [Metaspace: 3479K->3479K(1056768K)],
0.0045983 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
    PSYoungGen      total 76288K, used 655K [0x000000076b500000, 0x00000000
770a00000, 0x00000007c000000)
        eden space 65536K, 1% used [0x000000076b500000,0x000000076b5a3ee8,0x0
00000076f500000)
        from space 10752K, 0% used [0x000000076f500000,0x000000076f500000,0x0
00000076ff80000)
        to   space 10752K, 0% used [0x000000076ff80000,0x000000076ff80000,0x0
000000770a00000)
    ParOldGen      total 175104K, used 672K [0x00000006c1e00000, 0x000000
06cc900000, 0x000000076b500000)
        object space 175104K, 0% used [0x00000006c1e00000,0x00000006c1ea8070,
0x00000006cc900000)
    Metaspace      used 3486K, capacity 4496K, committed 4864K, reserved
1056768K
    class space     used 385K, capacity 388K, committed 512K, reserved 104
8576K
```

我们能够看到，上述进行了 GC 收集的行为，将上述的新生代中的两个对象都进行回收了

PSYoungGen: 15490K->808K(76288K)] 15490K->816K(251392K)

如果使用引用计数算法，那么这两个对象将会无法回收。而现在两个对象被回收了，说明 Java 使用的不是引用计数算法来进行标记的。



小结

引用计数算法，是很多语言的资源回收选择，例如因人工智能而更加火热的 Python，它更是同时支持引用计数和垃圾收集机制。

具体哪种最优是要看场景的，业界有大规模实践中仅保留引用计数机制，以提高吞吐量的尝试。

Java 并没有选择引用计数，是因为其存在一个基本的难题，也就是很难处理循环引用关系。Python 如何解决循环引用？

手动解除：很好理解，就是在合适的时机，解除引用关系。使用弱引用 `weakref`, `weakref` 是 Python 提供的标准库，旨在解决循环引用。

标记阶段：可达性分析算法

概念

可达性分析算法：也可以称为根搜索算法、追踪性垃圾收集

相对于引用计数算法而言，可达性分析算法不仅同样具备实现简单和执行高效等特点，更重要的是该算法可以有效地解决在引用计数算法中循环引用的问题，防止内存泄漏的发生。

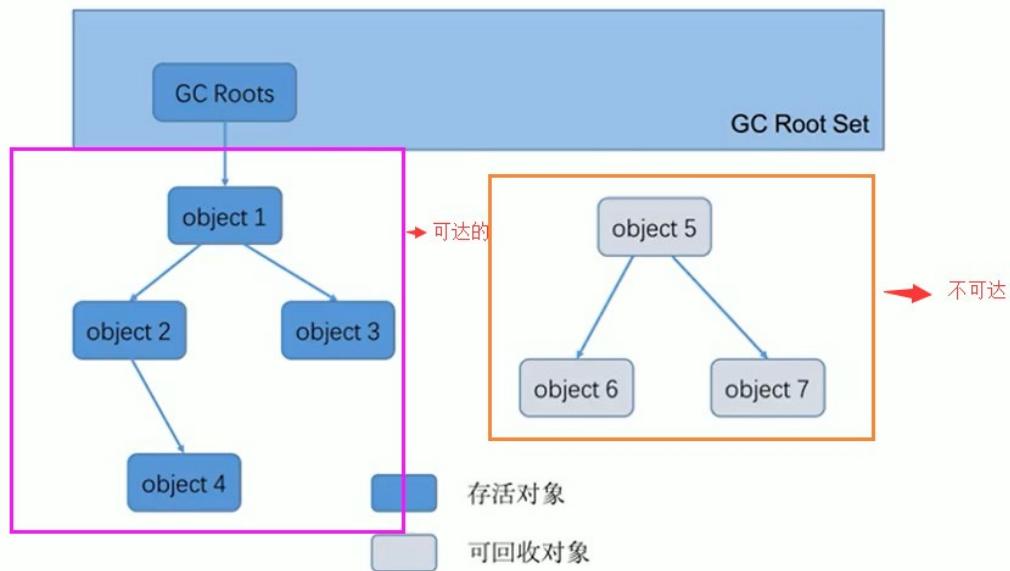
相较于引用计数算法，这里的可达性分析就是 Java、C#选择的。这种类型的垃圾收集通常也叫作追踪性垃圾收集（Tracing Garbage Collection）

思路

所谓“GC Roots”根集合就是一组必须活跃的引用。

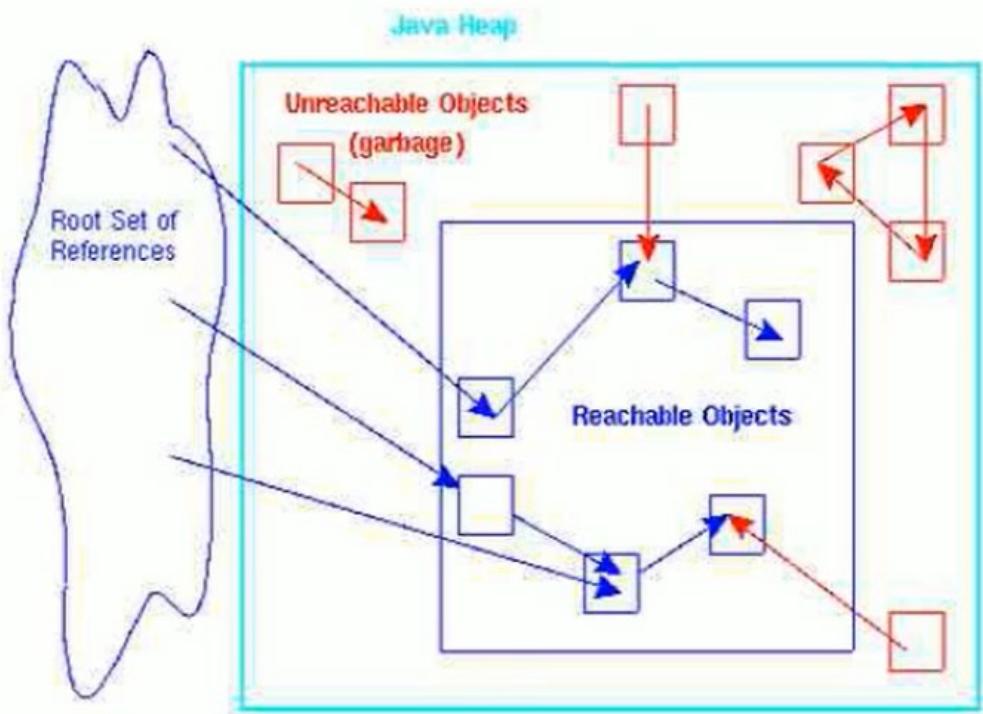
基本思路：

- 可达性分析算法是以根对象集合（GC Roots）为起始点，按照从上至下的方式搜索被根对象集合所连接的目标对象是否可达。
- 使用可达性分析算法后，内存中的存活对象都会被根对象集合直接或间接连接着，搜索所走过的路径称为引用链（Reference Chain）
- 如果目标对象没有任何引用链相连，则是不可达的，就意味着该对象已经死亡，可以标记为垃圾对象。
- 在可达性分析算法中，只有能够被根对象集合直接或者间接连接的对象才是存活对象。



GC Roots 可以是哪些? 面试题

- 虚拟机栈中引用的对象
 - 比如：各个线程被调用的方法中使用到的参数、局部变量等。
- 本地方法栈内 JNI（通常说的本地方法）引用的对象
- 方法区中类静态属性引用的对象
 - 比如：Java 类的引用类型静态变量
- 方法区中常量引用的对象
 - 比如：字符串常量池（string Table）里的引用
- 所有被同步锁 synchronized 持有的对象
- Java 虚拟机内部的引用。
 - 基本数据类型对应的 Class 对象，一些常驻的异常对象（如：NullPointerException、outofMemoryError），系统类加载器。
- 反映 java 虚拟机内部情况的 JMXBean、JVMTI 中注册的回调、本地代码缓存等。



总结

总结一句话就是，除了堆空间外的一些结构，比如虚拟机栈、本地方法栈、方法区、字符串常量池等地方对堆空间进行引用的，都可以作为 GC Roots 进行可达性分析。

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不同，还可以有其他对象“临时性”地加入，共同构成完整 GC Roots 集合。比如：分代收集和局部回收（PartialGC）。

如果只针对 Java 堆中的某一块区域进行垃圾回收（比如：典型的只针对新生代），必须考虑到内存区域是虚拟机自己的实现细节，更不是孤立封闭的，这个区域的对象完全有可能被其他区域的对象所引用，这时候就需要一并将关联的区域对象也加入 GC Roots 集合中去考虑，才能保证可达性分析的准确性。

小技巧

由于 Root 采用栈方式存放变量和指针，所以如果一个指针，它保存了堆内存里面的对象，但是自己又不存放在堆内存里面，那它就是一个 Root。

注意

如果要使用可达性分析算法来判断内存是否可回收，那么分析工作必须在一个能保障**一致性的快照**中进行。这点不满足的话分析结果的准确性就无法保证。

这点也是导致 GC 进行时必须“stop The World”的一个重要原因。

即使是号称（几乎）不会发生停顿的 CMS 收集器中，枚举根节点时也是必须要停顿的。

对象的 finalization 机制

Java 语言提供了对象终止（finalization）机制来允许开发人员提供对象被销毁之前的自定义处理逻辑。

当垃圾回收器发现没有引用指向一个对象，即：垃圾回收此对象之前，总会先调用这个对象的 finalize() 方法。

finalize() 方法允许在子类中被重写，用于在对象被回收时进行资源释放。通常在这个方法中进行一些资源释放和清理的工作，比如关闭文件、套接字和数据库连接等。

注意

永远不要主动调用某个对象的 finalize() 方法，它应该交给垃圾回收机制调用。理由包括下面三点：

- 在 finalize() 时可能会导致对象复活。
- finalize() 方法的执行时间是没有保障的，它完全由 GC 线程决定，极端情况下，若不发生 GC，则 finalize() 方法将没有执行机会。
 - 因为优先级比较低，即使主动调用该方法，也不会因此就直接进行回收
- 一个糟糕的 finalize() 会严重影响 Gc 的性能。

从功能上来说，finalize() 方法与 C++ 中的析构函数比较相似，但是 Java 采用的是基于垃圾回收器的自动内存管理机制，所以 finalize() 方法在本质上不同于 C++ 中的析构函数。

由于 finalize() 方法的存在，虚拟机中的对象一般处于三种可能的状态。

生存还是死亡？ 面试题

如果从所有的根节点都无法访问到某个对象，说明对象已经不再使用了。一般来说，此对象需要被回收。但事实上，也并非是“非死不可”的，这时候它们暂时处于“缓

刑”阶段。一个无法触及的对象有可能在某一个条件下“复活”自己，如果这样，那么对它的回收就是不合理的，为此，定义虚拟机中的对象可能的三种状态。如下：

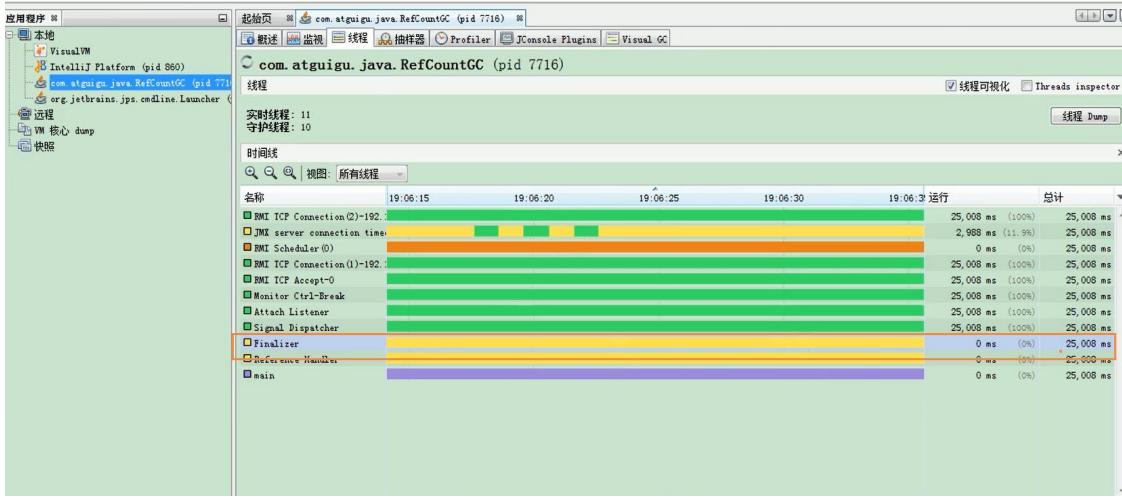
- 可触及的：从根节点开始，可以到达这个对象。
- 可复活的：对象的所有引用都被释放，但是对象有可能在 `finalize()` 中复活。
- 不可触及的：对象的 `finalize()` 被调用，并且没有复活，那么就会进入不可触及状态。不可触及的对象不可能被复活，因为 `finalize()` 只会被调用一次。

以上 3 种状态中，是由于 `finalize()` 方法的存在，进行的区分。只有在对象不可触及时才可以被回收。

具体过程

判定一个对象 `objA` 是否可回收，至少要经历两次标记过程：

- 如果对象 `objA` 到 GC Roots 没有引用链，则进行第一次标记。
- 进行筛选，判断此对象是否有必要执行 `finalize()` 方法
 - 如果对象 `objA` 没有重写 `finalize()` 方法，或者 `finalize()` 方法已经被虚拟机调用过，则虚拟机视为“没有必要执行”，`objA` 被判定为不可触及的。
 - 如果对象 `objA` 重写了 `finalize()` 方法，且还未执行过，那么 `objA` 会被插入到 F-Queue 队列中，由一个虚拟机自动创建的、低优先级的 Finalizer 线程触发其 `finalize()` 方法执行。
 - `finalize()` 方法是对对象逃脱死亡的最后机会，稍后 GC 会对 F-Queue 队列中的对象进行第二次标记。如果 `objA` 在 `finalize()` 方法中与引用链上的任何一个对象建立了联系，那么在第二次标记时，`objA` 会被移出“即将回收”集合。之后，对象会再次出现没有引用存在的情况。在这个情况下，`finalize` 方法不会被再次调用，对象会直接变成不可触及的状态，也就是说，一个对象的 `finalize` 方法只会被调用一次。



上图就是我们看到的 Finalizer 线程

代码演示

我们使用重写 finalize()方法，然后在方法的内部，重写将其存放到 GC Roots 中

```
/**
 * 测试 Object 类中 finalize()方法
 * 对象复活场景
 *
 * @author: 陌溪
 * @create: 2020-07-12-11:06
 */
public class CanReliveObj {
    // 类变量, 属于GC Roots 的一部分
    public static CanReliveObj canReliveObj;

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("调用当前类重写的 finalize()方法");
        canReliveObj = this;
    }

    public static void main(String[] args) throws InterruptedException
    {
        canReliveObj = new CanReliveObj();
        canReliveObj = null;
        System.gc();
        System.out.println("-----第一次 gc 操作-----");
        // 因为Finalizer 线程的优先级比较低，暂停2秒，以等待它
        Thread.sleep(2000);
    }
}
```

```

    if (canReliveObj == null) {
        System.out.println("obj is dead");
    } else {
        System.out.println("obj is still alive");
    }

    System.out.println("-----第一次 gc 操作-----");
    canReliveObj = null;
    System.gc();
    // 下面代码和上面代码是一样的，但是 canReliveObj 却自救失败了
    Thread.sleep(2000);
    if (canReliveObj == null) {
        System.out.println("obj is dead");
    } else {
        System.out.println("obj is still alive");
    }

}

}

```

最后运行结果

```

-----第一次 gc 操作-----
调用当前类重写的 finalize()方法
obj is still alive
-----第二次 gc 操作-----
obj is dead

```

在进行第一次清除的时候，我们会执行 finalize 方法，然后对象进行了一次自救操作，但是因为 finalize()方法只会被调用一次，因此第二次该对象将会被垃圾清除。

MAT 与 JProfiler 的 GC Roots 溯源

MAT 是什么？

MAT 是 Memory Analyzer 的简称，它是一款功能强大的 Java 堆内存分析器。用于查找内存泄漏以及查看内存消耗情况。

MAT 是基于 Eclipse 开发的，是一款免费的性能分析工具。

大家可以在 <http://www.eclipse.org/mat/> 下载并使用 MAT

命令行使用 jmap

```
C:\Users\Administrator>jps
12752 Jps
14036 GCRootsTest
1372
588 Launcher

C:\Users\Administrator>jmap -dump:format=b, live, file=test1.bin 14036
Dumping heap to C:\Users\Administrator\test1.bin ...
Heap dump file created

C:\Users\Administrator>jmap -dump:format=b, live, file=test2.bin 14036
Dumping heap to C:\Users\Administrator\test2.bin ...
Heap dump file created
```

使用 JVVisualVM

捕获的 heap dump 文件是一个临时文件，关闭 JVVisualVM 后自动删除，若要保留，需要将其另存为文件。可通过以下方法捕获 heap dump：

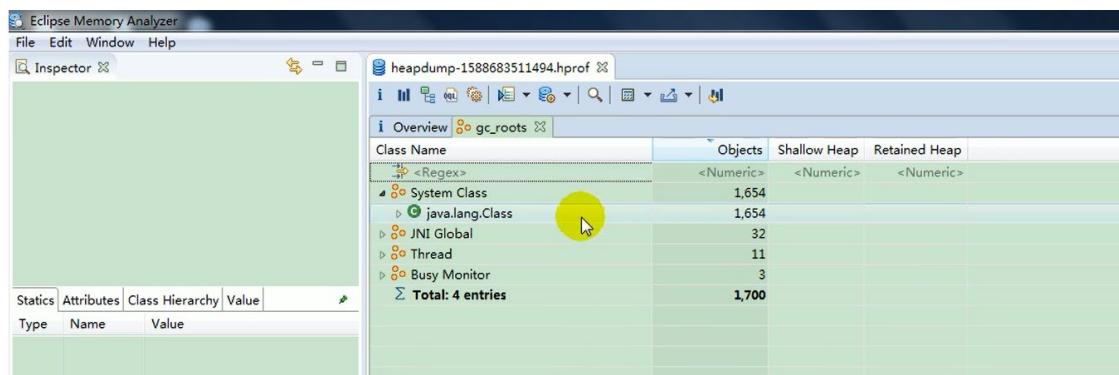
在左侧“Application”（应用程序）子窗口中右击相应的应用程序，选择 Heap Dump（堆 Dump）。

在 Monitor（监视）子标签页中点击 Heap Dump（堆 Dump）按钮。本地应用程序的 Heap dumps 作为应用程序标签页的一个子标签页打开。同时，heap dump 在左侧的 Application（应用程序）栏中对应一个含有时间戳的节点。

右击这个节点选择 save as（另存为）即可将 heap dump 保存到本地。

使用 MAT 打开 Dump 文件

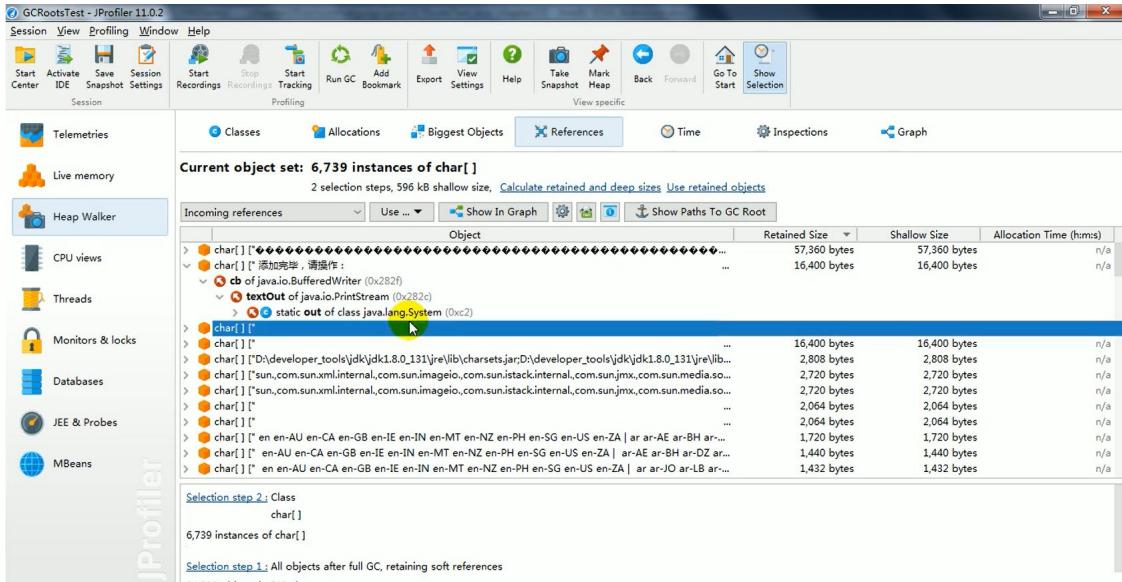
打开后，我们就可以看到有哪些可以作为 GC Roots 的对象



里面我们能够看到有一些常用的 Java 类，然后 Thread 线程。

JProfiler 的 GC Roots 源溯

我们在实际的开发中，一般不会查找全部的 GC Roots，可能只是查找某个对象的整个链路，或者称为 GC Roots 源溯，这个时候，我们就可以使用 JProfiler



如何判断什么原因造成 OOM

当我们程序出现 OOM 的时候，我们就需要进行排查，我们首先使用下面的例子进行说明

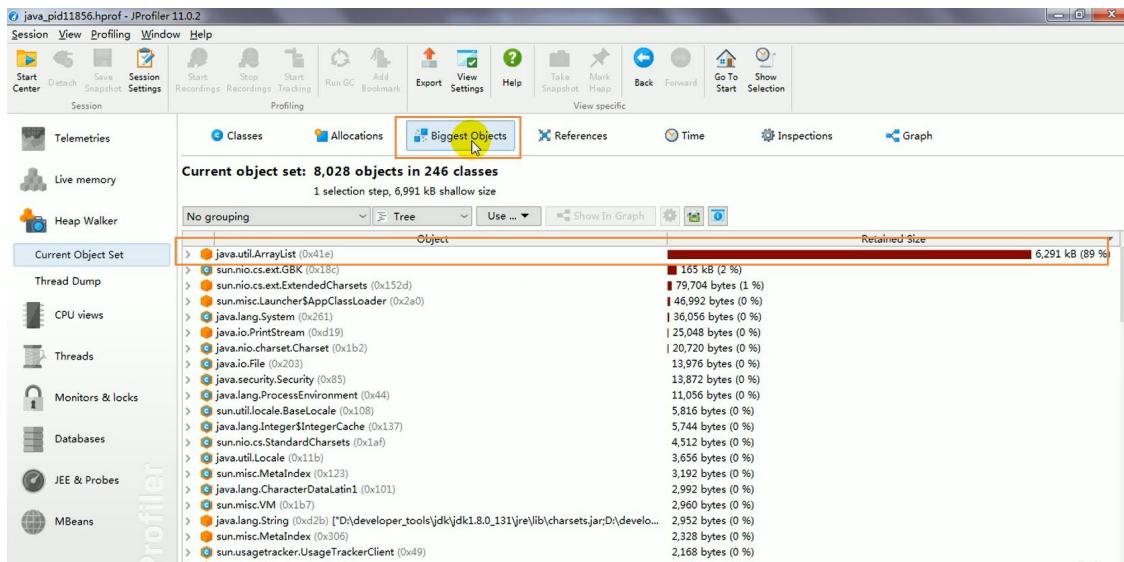
```
/**  
 * 内存溢出排查  
 * -Xms8m -Xmx8m -XX:HeapDumpOnOutOfMemoryError  
 * @author: 陌溪  
 * @create: 2020-07-12-14:56  
 */  
  
public class HeapOOM {  
    // 创建1M的文件  
    byte [] buffer = new byte[1 * 1024 * 1024];  
  
    public static void main(String[] args) {  
        ArrayList<HeapOOM> list = new ArrayList<>();  
        int count = 0;  
        try {  
            while (true) {  
                list.add(new HeapOOM());  
                count++;  
            }  
        } catch (Exception e) {  
            System.out.println("Exception caught: " + e.getMessage());  
        }  
    }  
}
```

```
        }
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("count:" + count);
    }
}
```

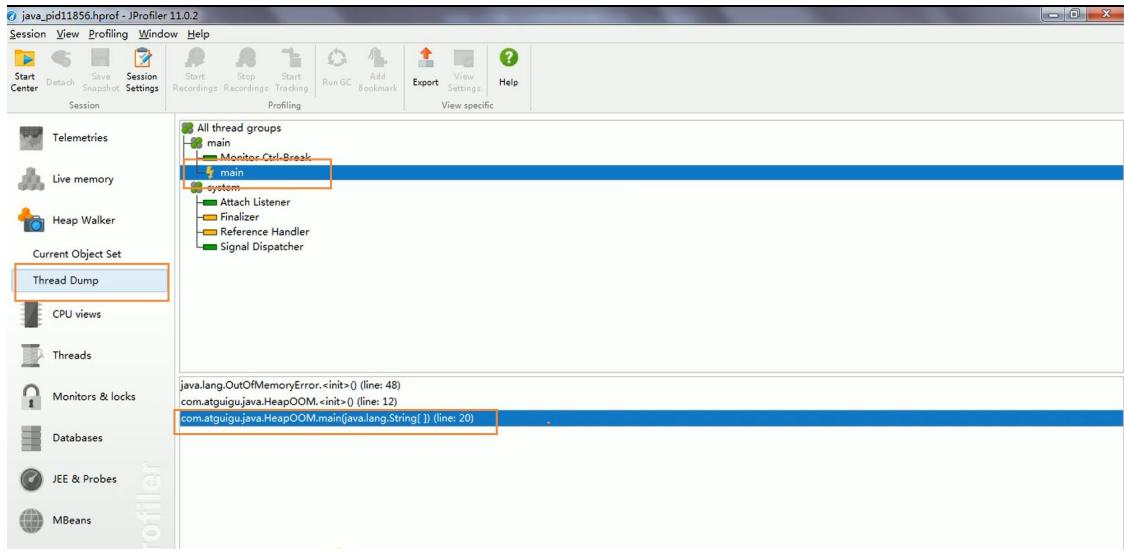
上述代码就是不断的创建一个 1M 小字节数组，然后让内存溢出，我们需要限制一下内存大小，同时使用 `HeapDumpOnOutOfMemoryError` 将出错时候的 dump 文件输出

`-Xms8m -Xmx8m -XX:HeapDumpOnOutOfMemoryError`

我们将生成的 dump 文件打开，然后点击 Biggest Objects 就能够看到超大对象



然后我们通过线程，还能够定位到哪里出现 OOM



清除阶段：标记-清除算法

当成功区分出内存中存活对象和死亡对象后，GC 接下来的任务就是执行垃圾回收，释放掉无用对象所占用的内存空间，以便有足够的可用内存空间为新对象分配内存。目前在 JVM 中比较常见的三种垃圾收集算法是

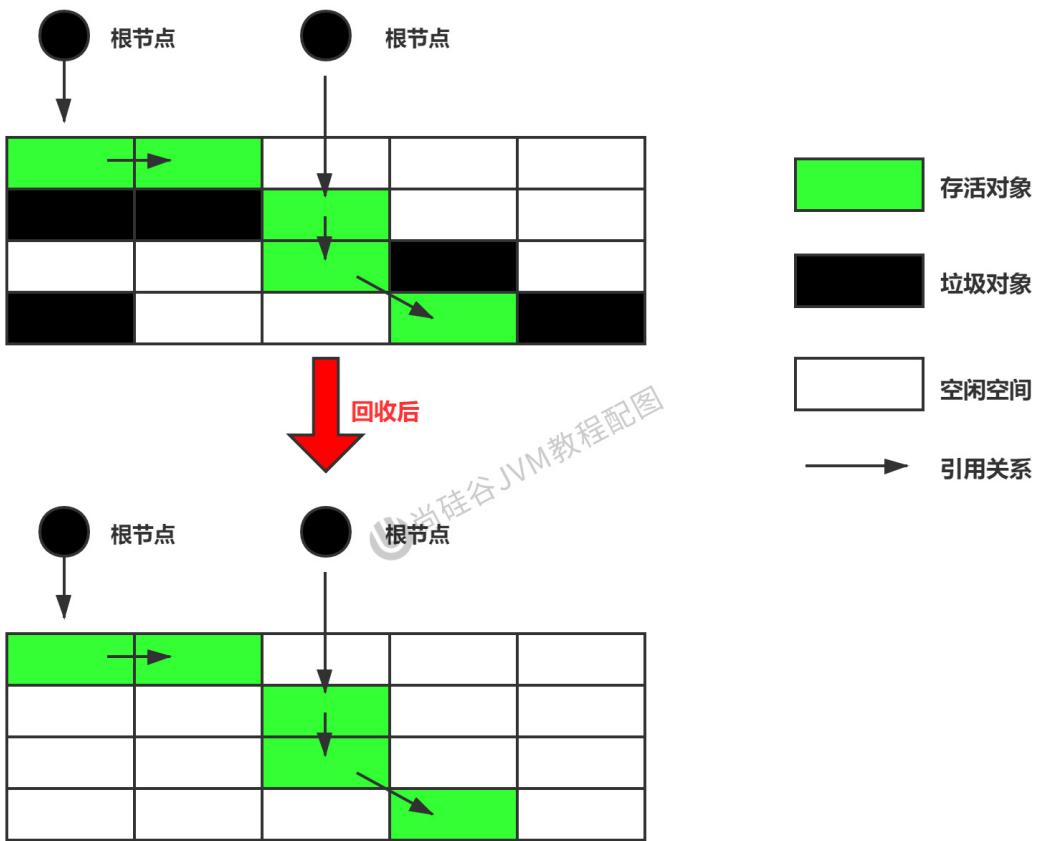
- 标记—清除算法（Mark-Sweep）
- 复制算法（copying）
- 标记-压缩算法（Mark-Compact）

标记-清除算法（Mark-Sweep）是一种非常基础和常见的垃圾收集算法，该算法被 J.McCarthy 等人在 1960 年提出并应用于 Lisp 语言。

执行过程

当堆中的有效内存空间（available memory）被耗尽的时候，就会停止整个程序（也被称为 stop the world），然后进行两项工作，第一项则是标记，第二项则是清除

- **标记：**Collector 从引用根节点开始遍历，标记所有被引用的对象。一般是在对象的 Header 中记录为可达对象。
 - 标记的是引用的对象，不是垃圾！！
- **清除：**Collector 对堆内存从头到尾进行线性的遍历，如果发现某个对象在其 Header 中没有标记为可达对象，则将其回收



什么是清除？

这里所谓的清除并不是真的置空，而是把需要清除的对象地址保存在空闲的地址列表里。下次有新对象需要加载时，判断垃圾的位置空间是否够，如果够，就将数据存放在原有的地址实现**数据覆盖**。

关于空闲列表是在为对象分配内存的时候 提过

- 如果内存规整
 - 采用指针碰撞的方式进行内存分配
- 如果内存不规整
 - 虚拟机需要维护一个列表
 - 空闲列表分配

缺点

- 标记清除算法的效率不算高 主要是遍历过程，两次全局扫描遍历

- 在进行 GC 的时候，需要停止整个应用程序，用户体验较差
- 这种方式清理出来的空闲内存是不连续的，产生内存碎片，需要维护一个空闲列表

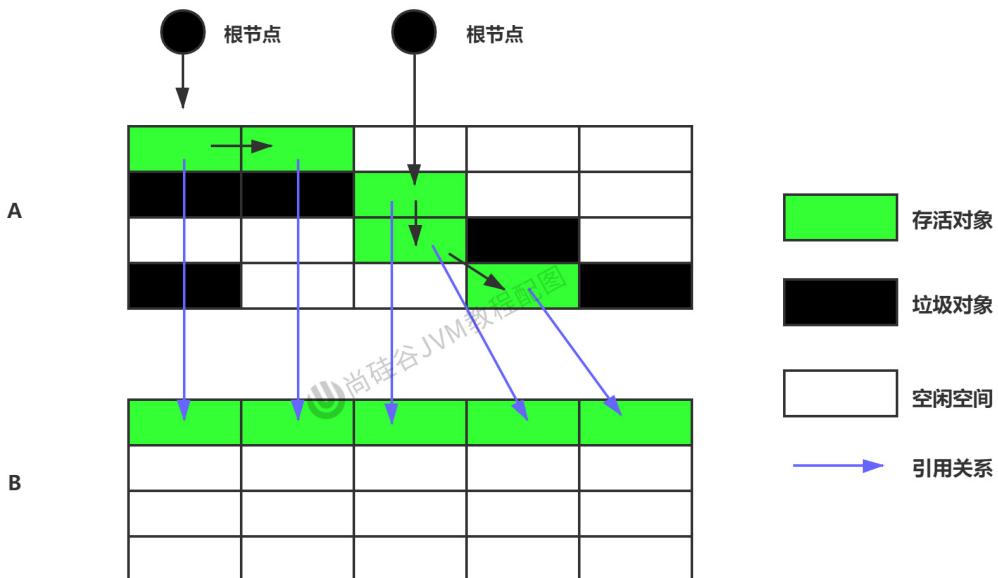
清除阶段：复制算法

背景

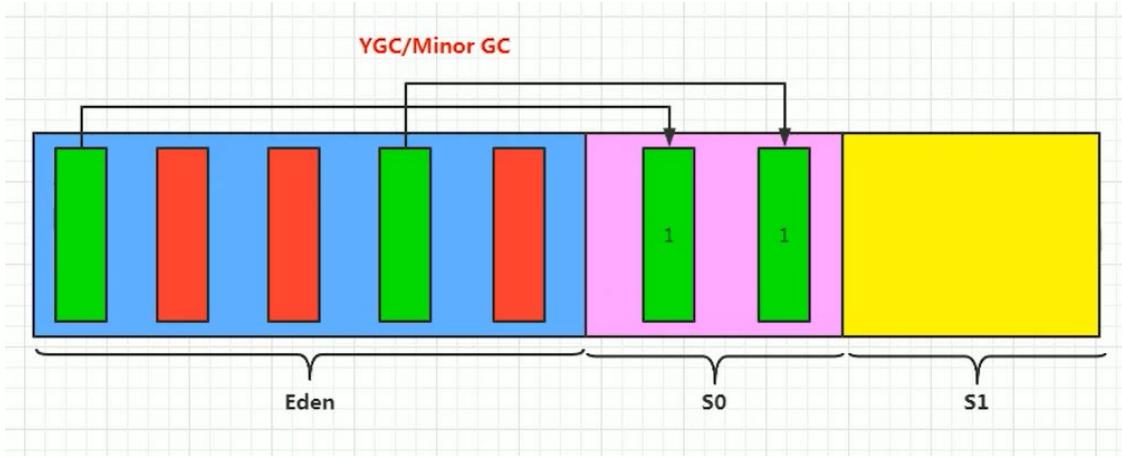
为了解决标记-清除算法在垃圾收集效率方面的缺陷，M.L.Minsky 于 1963 年发表了著名的论文，“使用双存储区的 Lisp 语言垃圾收集器 CA LISP Garbage Collector Algorithm Using Serial Secondary Storage）”。M.L.Minsky 在该论文中描述的算法被人们称为复制（Copying）算法，它也被 M.L.Minsky 本人成功地引入到了 Lisp 语言的一个实现版本中。

核心思想

将活着的内存空间分为两块，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后清除正在使用的内存块中的所有对象，交换两个内存的角色，最后完成垃圾回收



把可达的对象，直接复制到另外一个区域中复制完成后，A 区就没有用了，里面的对象可以直接清除掉，其实里面的新生代里面就用到了复制算法



优点

- 没有标记和清除过程，实现简单，运行高效
- 复制过去以后保证空间的连续性，不会出现“碎片”问题。

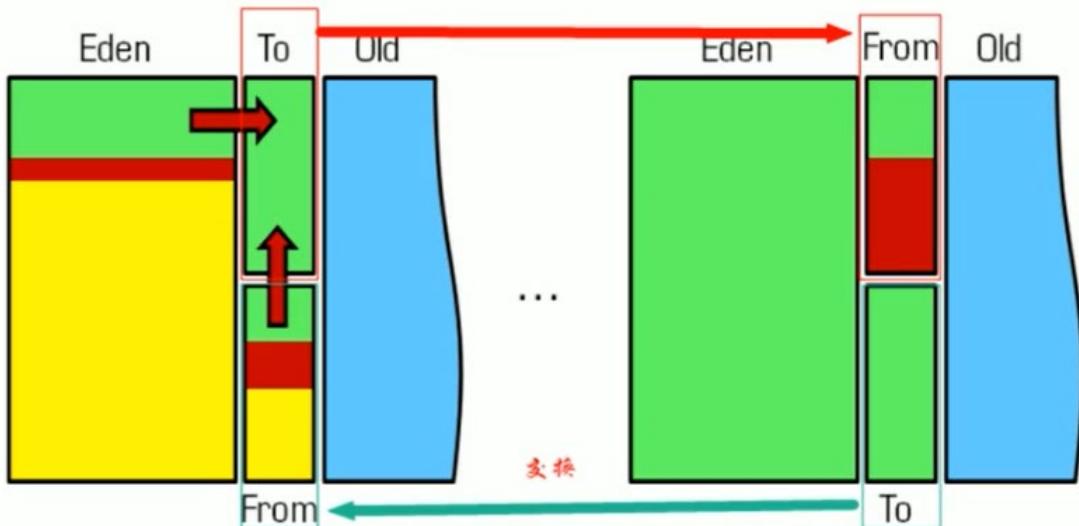
缺点

- 此算法的缺点也是很明显的，就是需要两倍的内存空间。
- 对于 G1 这种分拆成为大量 region 的 GC，复制而不是移动，意味着 GC 需要维护 region 之间对象引用关系，不管是内存占用或者时间开销也不小

注意

如果系统中的垃圾对象很多，复制算法需要复制的存活对象数量并不会太大，或者说非常低才行（老年代大量的对象存活，那么复制的对象将会有很多，效率会很低）

在新生代，对常规应用的垃圾回收，一次通常可以回收 70% - 99% 的内存空间。回收性价比很高。所以现在的商业虚拟机都是用这种收集算法回收新生代。



清除阶段：标记-整理算法

背景

复制算法的高效性是建立在存活对象少、垃圾对象多的前提下的。这种情况在新生代经常发生，但是在老年代，更常见的情况是大部分对象都是存活对象。如果依然使用复制算法，由于存活对象较多，复制的成本也将很高。因此，基于老年代垃圾回收的特性，需要使用其他的算法。

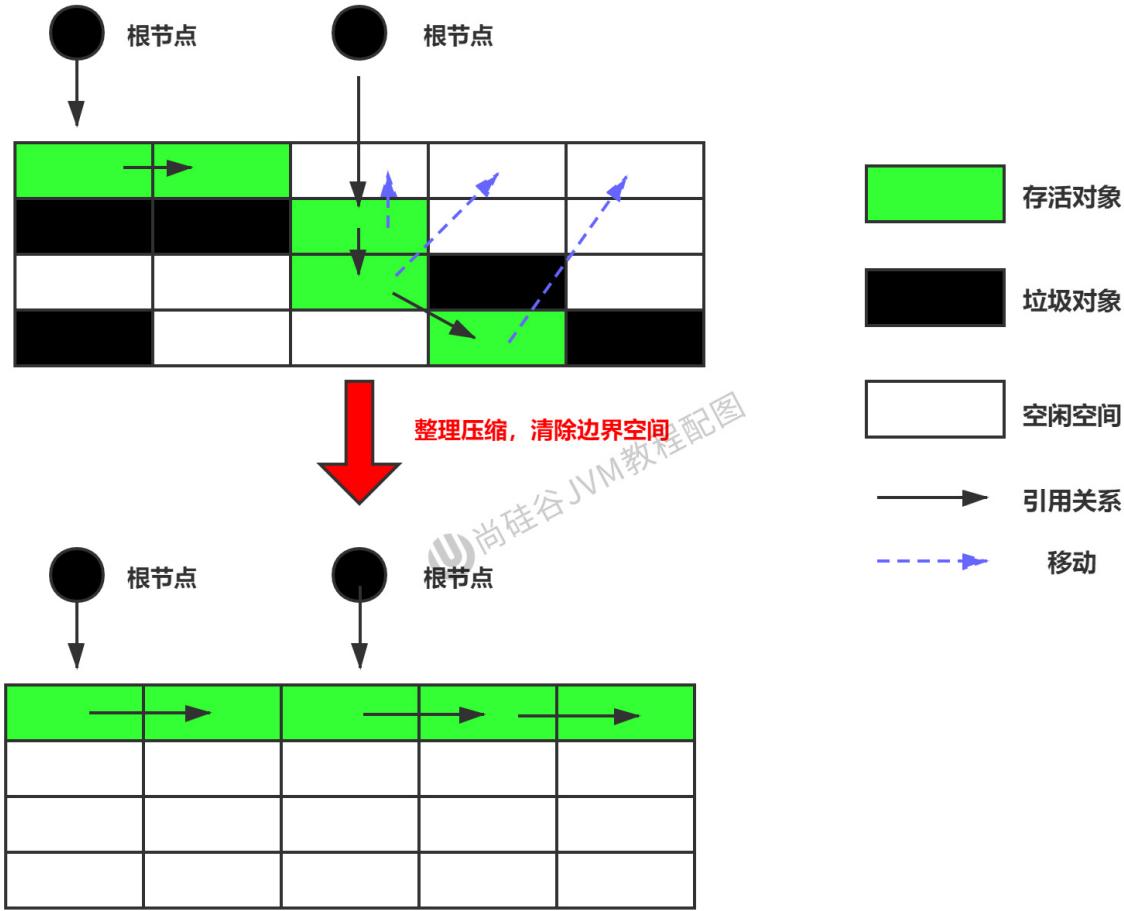
标记一清除算法的确可以应用在老年代中，但是该算法不仅执行效率低下，而且在执行完内存回收后还会产生内存碎片，所以 JVM 的设计者需要在此基础之上进行改进。标记-压缩（Mark-Compact）算法由此诞生。

1970 年前后，G.L.Steele、C.J.Chene 和 D.s.Wise 等研究者发布标记-压缩算法。在许多现代的垃圾收集器中，人们都使用了标记-压缩算法或其改进版本。

执行过程

第一阶段和标记清除算法一样，从根节点开始标记所有被引用对象

第二阶段将所有的存活对象压缩到内存的一端，按顺序排放。之后，清理边界外所有的空间。



标清和标整的区别

标记-压缩算法的最终效果等同于标记-清除算法执行完成后，再进行一次内存碎片整理，因此，也可以把它称为**标记-清除-压缩**（Mark-Sweep-Compact）算法。

二者的本质差异在于标记-清除算法是一种非移动式的回收算法，标记-压缩是移动式的。是否移动回收后的存活对象是一项优缺点并存的风险决策。可以看到，标记的存活对象将会被整理，按照内存地址依次排列，而未被标记的内存会被清理掉。如此一来，当我们需要给新对象分配内存时，JVM 只需要持有一个内存的起始地址即可，这比维护一个空闲列表显然少了许多开销。

标整的优点

优点 节省空间、没有碎片

- 消除了标记-清除算法当中，内存区域分散的缺点，我们需要给新对象分配内存时，JVM 只需要持有一个内存的起始地址即可。
- 消除了复制算法当中，内存减半的高额代价。

缺点

- 从效率上来说，标记—整理算法要低于复制算法和标记清除—算法。
- 移动对象的同时，如果对象被其他对象引用，则还需要调整引用的地址
- 移动过程中，需要全程暂停用户应用程序。即：STW

小结

效率上来说，复制算法是当之无愧的老大，但是却浪费了太多内存。

而为了尽量兼顾上面提到的三个指标，标记-整理算法相对来说更平滑一些，但是效率上不尽如人意，它比复制算法多了一个标记的阶段，比标记-清除多了一个整理内存的阶段。

	标记清除	标记整理	复制
速率	中等	最慢	最快
空间开销	少（但会堆积碎片）	少（不堆积碎片）	通常需要活对象的 2 倍空间（不堆积碎片）
移动对象	否	是	是

综合我们可以找到，没有最好的算法，只有最合适的方法

分代收集算法

前面所有这些算法中，并没有一种算法可以完全替代其他算法，它们都具有自己独特的优势和特点。分代收集算法应运而生。

分代收集算法，是基于这样一个事实：不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的收集方式，以便提高回收效率。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使用不同的回收算法，以提高垃圾回收的效率。

在 Java 程序运行的过程中，会产生大量的对象，其中有些对象是与业务信息相关，比如 Http 请求中的 Session 对象、线程、Socket 连接，这类对象跟业务直接挂钩，因此生命周期比较长。但是还有一些对象，主要是程序运行过程中生成的临时变量，这些对象生命周期会比较短，比如：string 对象，由于其不变类的特性，系统会产生大量的这些对象，有些对象甚至只用一次即可回收。

目前几乎所有的 GC 都采用分代收集算法执行垃圾回收的

在 HotSpot 中，基于分代的概念，GC 所使用的内存回收算法必须结合年轻代和老年代各自的特点。

- 年轻代 (Young Gen)

年轻代特点：区域相对老年代较小，对象生命周期短、存活率低，回收频繁。

这种情况复制算法的回收整理，速度是最快的。复制算法的效率只和当前存活对象大小有关，因此很适用于年轻代的回收。而复制算法内存利用率不高的问题，通过 hotspot 中的两个 survivor 的设计得到缓解。

- 老年代 (Tenured Gen)

老年代特点：区域较大，对象生命周期长、存活率高，回收不及年轻代频繁。

这种情况存在大量存活率高的对象，复制算法明显变得不合适。一般是由标记-清除或者是标记-清除与标记-整理的混合实现。

- Mark 阶段的开销与存活对象的数量成正比。
- Sweep 阶段的开销与所管理区域的大小成正相关。
- compact 阶段的开销与存活对象的数据成正比。

以 HotSpot 中的 CMS 回收器为例，CMS 是基于 Mark-Sweep 实现的，对于对象的回收效率很高。而对于碎片问题，CMS 采用基于 Mark-Compact 算法的 Serial old 回收器作为补偿措施：当内存回收不佳（碎片导致的 Concurrent Mode Failure 时），将采用 serial old 执行 FullGC 以达到对老年代内存的整理。

分代的思想被现有的虚拟机广泛使用。几乎所有的垃圾回收器都区分新生代和老年代

增量收集算法

概述

上述现有的算法，在垃圾回收过程中，应用软件将处于一种 stop the World 的状态。在 stop the World 状态下，应用程序所有的线程都会挂起，暂停一切正常的工作，等待垃圾回收的完成。如果垃圾回收时间过长，应用程序会被挂起很久，将严重影响用户体验或者系统的稳定性。为了解决这个问题，即对实时垃圾收集算法的研究直接导致了增量收集 (Incremental Collecting) 算法的诞生。

如果一次性将所有的垃圾进行处理，需要造成系统长时间的停顿，那么就可以让垃圾收集线程和应用程序线程交替执行。每次，垃圾收集线程只收集一小片区域的内存空间，接着切换到应用程序线程。依次反复，直到垃圾收集完成。

总的来说，增量收集算法的基础仍是传统的标记-清除和复制算法。**增量收集算法通过对线程间冲突的妥善处理，允许垃圾收集线程以分阶段的方式完成标记、清理或复制工作**

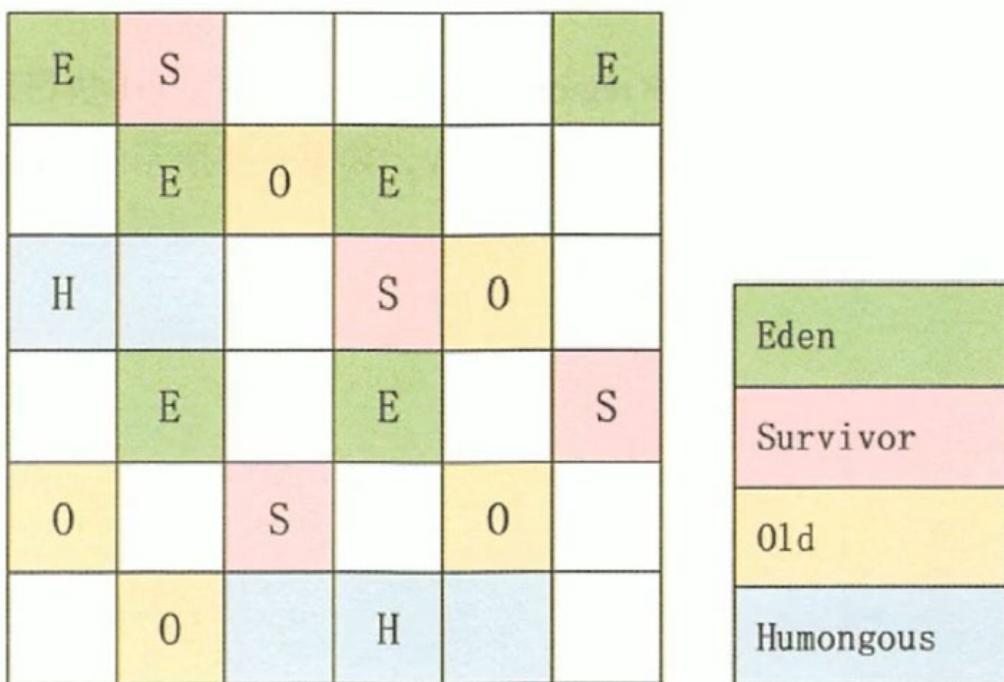
缺点

使用这种方式，由于在垃圾回收过程中，间断性地还执行了应用程序代码，所以能减少系统的停顿时间。但是，因为线程切换和上下文转换的消耗，会使得垃圾回收的总体成本上升，造成系统吞吐量的下降。

分区算法

一般来说，在相同条件下，堆空间越大，一次 GC 时所需要的时间就越长，有关 GC 产生的停顿也越长。为了更好地控制 GC 产生的停顿时间，将一块大的内存区域分割成多个小块，根据目标的停顿时间，每次合理地回收若干个小区间，而不是整个堆空间，从而减少一次 GC 所产生的停顿。

分代算法将按照对象的生命周期长短划分成两个部分，分区算法将整个堆空间划分成连续的不同小区间。每一个小区间都独立使用，独立回收。这种算法的好处是可以控制一次回收多少个小区间。



写到最后

注意，这些只是基本的算法思路，实际 GC 实现过程要复杂的多，目前还在发展的前沿 GC 都是复合算法，并且并行和并发兼备。

垃圾回收相关概念

System.gc()的理解

在默认情况下，通过 system.gc() 或者 Runtime.getRuntime().gc() 的调用，会显式触发 FullGC，同时对老年代和新生代进行回收，尝试释放被丢弃对象占用的内存。

然而 system.gc() 调用附带一个免责声明，无法保证对垃圾收集器的调用。(不能确保立即生效)

JVM 实现者可以通过 system.gc() 调用来决定 JVM 的 GC 行为。而一般情况下，垃圾回收应该是自动进行的，无须手动触发，否则就太过于麻烦了。在一些特殊情况下，如我们正在编写一个性能基准，我们可以在运行之间调用 System.gc()

代码演示是否触发 GC 操作

```
/*
 * System.gc()
 *
 * @author: 陌溪
 * @create: 2020-07-12-19:07
 */
public class SystemGCTest {
    public static void main(String[] args) {
        new SystemGCTest();
        // 提醒JVM 进行垃圾回收
        System.gc();
        //System.runFinalization();
    }

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("SystemGCTest 执行了 finalize 方法");
    }
}
```

运行结果，但是不一定会触发销毁的方法，调用 System.runFinalization() 会强制调用 **失去引用对象的 finalize()**

SystemGCTest 执行了 finalize 方法

手动 GC 来理解不可达对象的回收

代码如下所示：

```
/*
 * 局部变量回收
```

```

/*
 * @author: 陌溪
 * @create: 2020-07-12-19:12
 */
public class LocalVarGC {

    /**
     * 触发 Minor GC 没有回收对象，然后在触发 Full GC 将该对象存入 old 区
     */
    public void localvarGC1() {
        byte[] buffer = new byte[10*1024*1024];
        System.gc();
    }

    /**
     * 触发 YoungGC 的时候，已经被回收了
     */
    public void localvarGC2() {
        byte[] buffer = new byte[10*1024*1024];
        buffer = null;
        System.gc();
    }

    /**
     * 不会被回收，因为它还存放在局部变量表索引为 1 的槽中
     */
    public void localvarGC3() {
        {
            byte[] buffer = new byte[10*1024*1024];
        }
        System.gc();
    }

    /**
     * 会被回收，因为它还存放在局部变量表索引为 1 的槽中，但是后面定义的 value
     * 把这个槽给替换了
     */
    public void localvarGC4() {
        {
            byte[] buffer = new byte[10*1024*1024];
        }
        int value = 10;      槽被替换了
        System.gc();
    }

    /**
     * localvarGC5 中的数组已经被回收
     */
}

```

```
public void localvarGC5() {
    localvarGC1();
    System.gc();
}

public static void main(String[] args) {
    LocalVarGC localVarGC = new LocalVarGC();
    localVarGC.localvarGC3();
}
}
```

内存溢出

内存溢出相对于内存泄漏来说，尽管更容易被理解，但是同样的，内存溢出也是引发程序崩溃的罪魁祸首之一。

由于 GC 一直在发展，所以一般情况下，除非应用程序占用的内存增长速度非常快，造成垃圾回收已经跟不上内存消耗的速度，否则不太容易出现 OOM 的情况。

大多数情况下，GC 会进行各种年龄段的垃圾回收，实在不行了就放大招，来一次独占式的 FullGC 操作，这时候会回收大量的内存，供应用程序继续使用。

javadoc 中对 `outofMemoryError` 的解释是，没有空闲内存，并且垃圾收集器也无法提供更多内存。

首先说没有空闲内存的情况：说明 Java 虚拟机的堆内存不够。原因有二：

- Java 虚拟机的堆内存设置不够。

比如：可能存在内存泄漏问题；也很有可能就是堆的大小不合理，比如我们要处理比较可观的数据量，但是没有显式指定 JVM 堆大小或者指定数值偏小。我们可以通过参数-Xms、-Xmx 来调整。

- 代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在被引用）

对于老版本的 oracle JDK，因为永久代的大小是有限的，并且 JVM 对永久代垃圾回收（如，常量池回收、卸载不再需要的类型）非常不积极，所以当我们不断添加新类型的时候，永久代出现 `OutOfMemoryError` 也非常多见，尤其是在运行时存在大量动态类型生成的场合；类似 `intern` 字符串缓存占用太多空间，也会导致 OOM 问题。对应的异常信息，会标记出来和永久代相关：

“`java.lang.OutOfMemoryError:PermGen space`”。

随着元数据区的引入，方法区内存已经不再那么窘迫，所以相应的 OOM 有所改观，出现 OOM，异常信息则变成了：“`java.lang.OutOfMemoryError:Metaspace`”。直接内存不足，也会导致 OOM。

这里面隐含着一层意思是，在抛出 OutofMemoryError 之前，通常垃圾收集器会被触发，尽其所能去清理出空间。

例如：在引用机制分析中，涉及到 JVM 会去尝试回收软引用指向的对象等。在 `java.nio.Bits.reserveMemory()` 方法中，我们能清楚的看到，`System.gc()` 会被调用，以清理空间。

当然，也不是在任何情况下垃圾收集器都会被触发的

比如，我们去分配一个超大对象，类似一个超大数组超过堆的最大值，JVM 可以判断出垃圾收集并不能解决这个问题，所以直接抛出 OutofMemoryError。

内存泄漏

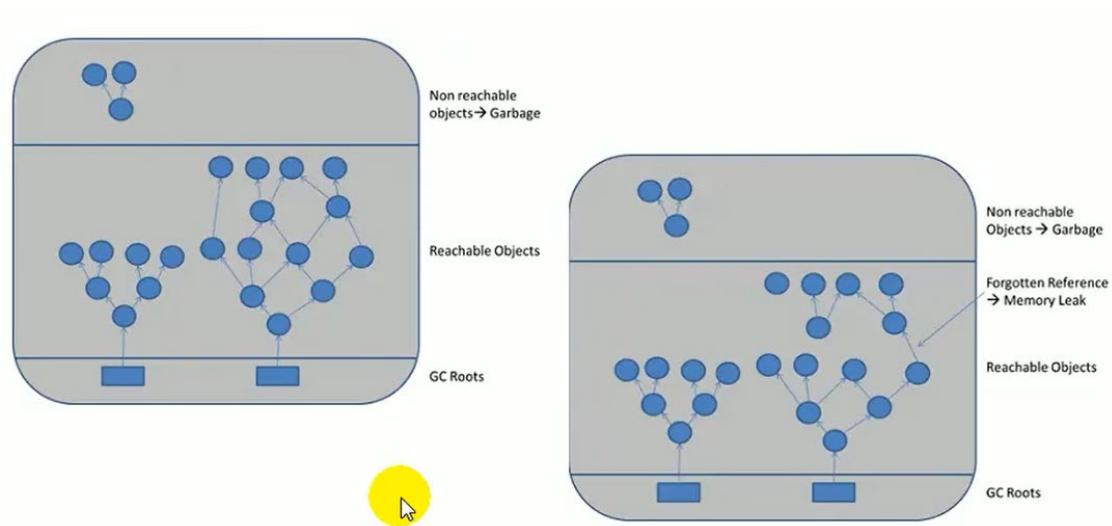
也称作“存储渗漏”。严格来说，只有对象不会再被程序用到了，但是 GC 又不能回收他们的情况，才叫内存泄漏。

但实际情况很多时候一些不太好的实践（或疏忽）会导致对象的生命周期变得很长甚至导致 OOM，也可以叫做宽泛意义上的“内存泄漏”。

尽管内存泄漏并不会立刻引起程序崩溃，但是一旦发生内存泄漏，程序中的可用内存就会被逐步蚕食，直至耗尽所有内存，最终出现 outofMemory 异常，导致程序崩溃。

注意，这里的存储空间并不是指物理内存，而是指虚拟内存大小，这个虚拟内存大小取决于磁盘交换区设定的大小。

买房子：80 平的房子，但是有 10 平是公摊的面积，我们是无法使用这 10 平的空间，这就是所谓的内存泄漏



Java 使用可达性分析算法，最上面的数据不可达，就是需要被回收的。后期有一些对象不用了，按道理应该断开引用，但是存在一些链没有断开，从而导致没有办法被回收。

举例

- 单例模式

单例的生命周期和应用程序是一样长的，所以单例程序中，如果持有对外部对象的引用的话，那么这个外部对象是不能被回收的，则会导致内存泄漏的产生。

- 一些提供 close 的资源未关闭导致内存泄漏

数据库连接（`dataSource.getConnection()`），网络连接（`socket`）和 `io` 连接必须手动 `close`，否则是不能被回收的。

Stop The World

`stop-the-world`，简称 STW，指的是 GC 事件发生过程中，会产生应用程序的停顿。停顿产生时整个应用程序线程都会被暂停，没有任何响应，有点像卡死的感觉，这个停顿称为 STW。

可达性分析算法中枚举根节点（GC Roots）会导致所有 Java 执行线程停顿。

- 分析工作必须在一个能确保一致性的快照中进行
- 一致性指整个分析期间整个执行系统看起来像被冻结在某个时间点上
- 如果出现分析过程中对象引用关系还在不断变化，则分析结果的准确性无法保证

被 STW 中断的应用程序线程会在完成 GC 之后恢复，频繁中断会让用户感觉像是网速不快造成电影卡带一样，所以我们需要减少 STW 的发生。

STW 事件和采用哪款 GC 无关所有的 GC 都有这个事件。

哪怕是 G1 也不能完全避免 Stop-the-world 情况发生，只能说垃圾回收器越来越优秀，回收效率越来越高，尽可能地缩短了暂停时间。

STW 是 JVM 在后台自动发起和自动完成的。在用户不可见的情况下，把用户正常的工作线程全部停掉。

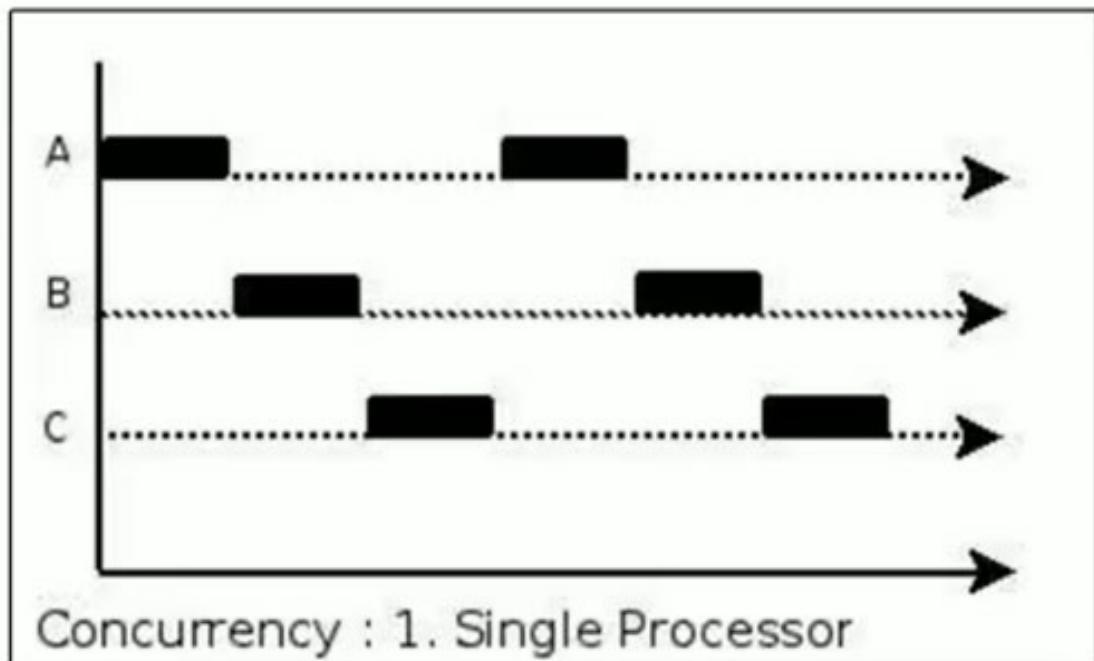
开发中不要用 `System.gc()` 会导致 stop-the-world 的发生。

垃圾回收的并行与并发

并发(concurrent)

在操作系统中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理器上运行。

并发不是真正意义上的“同时进行”，只是 CPU 把一个时间段划分成几个时间片段（时间区间），然后在这几个时间区间之间来回切换，由于 CPU 处理的速度非常快，只要时间间隔处理得当，即可让用户感觉是多个应用程序同时在进行。

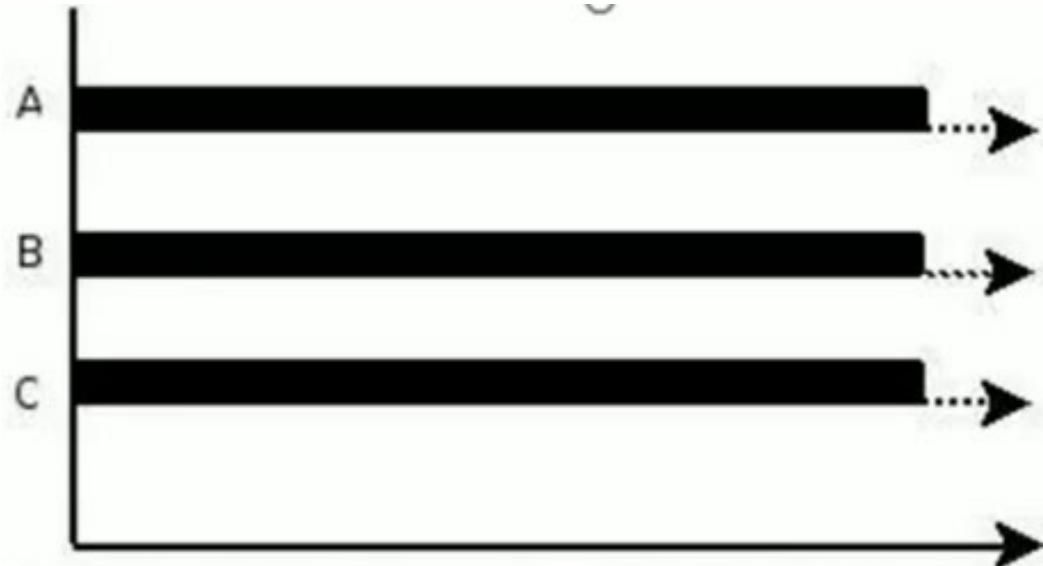


并行

当系统有一个以上 CPU 时，当一个 CPU 执行一个进程时，另一个 CPU 可以执行另一个进程，两个进程互不抢占 CPU 资源，可以同时进行，我们称之为并行（Parallel）。

其实决定并行的因素不是 CPU 的数量，而是 CPU 的核心数量，比如一个 CPU 多个核也可以并行。

适合科学计算，后台处理等弱交互场景



Parallelism : 1. Multiprocessores, Multicore

并发和并行对比

并发，指的是多个事情，在同一时间段内同时发生了。

并行，指的是多个事情，在同一时间点上同时发生了。

并发的多个任务之间是互相抢占资源的。并行的多个任务之间是不互相抢占资源的。

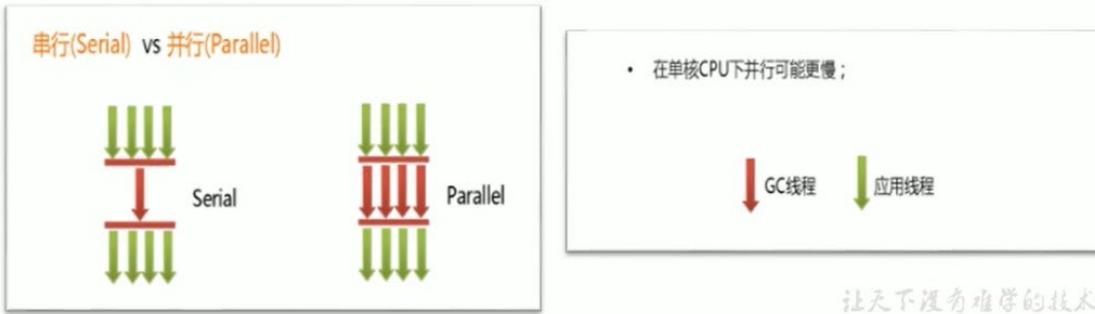
只有在多 CPU 或者一个 CPU 多核的情况下，才会发生并行。

否则，看似同时发生的事情，其实都是并发执行的。

垃圾回收的并行与并发

并发和并行，在谈论垃圾收集器的上下文语境中，它们可以解释如下：

- 并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍处于等待状态。如 ParNew、Parallel Scavenge、Parallel old；注意此处说的是垃圾回收线程并行工作，而不是垃圾回收与用户线程之间的关系
 - 相较于并行的概念，单线程执行。
 - 如果内存不够，则程序暂停，启动 JM 垃圾回收器进行垃圾回收。回收完，再启动程序的线程。
- 串行（Serial）
 - 相较于并行的概念，单线程执行。
 - 如果内存不够，则程序暂停，启动 JM 垃圾回收器进行垃圾回收。回收完，再启动程序的线程。



并发和并行，在谈论垃圾收集器的上下文语境中，它们可以解释如下：

并发 (Concurrent)：指**用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行）**，垃圾回收线程在执行时不会停顿用户程序的运行。

用户程序在继续运行，而垃圾收集程序线程运行于另一个 CPU 上；

如：CMS、G1



安全点与安全区域

安全点

程序执行时并非在所有地方都能停顿下来开始 GC，只有在特定的位置才能停顿下来开始 GC，这些位置称为“安全点（Safepoint）”。

Safepoint 的选择很重要，如果太少可能导致 GC 等待的时间太长，如果太频繁可能导致运行时的性能问题。大部分指令的执行时间都非常短暂，通常会根据“是否

具有让程序长时间执行的特征”为标准。比如：选择一些执行时间较长的指令作为 Safe Point，如方法调用、循环跳转和异常跳转等。

如何在 GC 发生时，检查所有线程都跑到最近的安全点停顿下来呢？

- **抢先式中断：**（目前没有虚拟机采用了）首先中断所有线程。如果还有线程不在安全点，就恢复线程，让线程跑到安全点。
- **主动式中断：**设置一个中断标志，各个线程运行到 Safe Point 的时候主动轮询这个标志，如果中断标志为真，则将自己进行中断挂起。（有轮询的机制）

安全区域（Safe Region）

Safepoint 机制保证了程序执行时，在不太长的时间内就会遇到可进入 GC 的 Safepoint。但是，程序“不执行”的时候呢？例如线程处于 sleep-状态或 Blocked 状态，这时候线程无法响应 JVM 的中断请求，“走”到安全点去中断挂起，JVM 也不太可能等待线程被唤醒。对于这种情况，就需要安全区域（Safe Region）来解决。

安全区域是指在一段代码片段中，对象的引用关系不会发生变化，在这个区域中的任何位置开始 GC 都是安全的。我们也可以把 Safe Region 看做是被扩展了的 Safepoint。

执行流程：

- 当线程运行到 Safe Region 的代码时，首先标识已经进入了 Safe Region，如果这段时间内发生 GC，JVM 会忽略标识为 Safe Region 状态的线程
- 当线程即将离开 Safe Region 时，会检查 JVM 是否已经完成 GC，如果完成了，则继续运行，否则线程必须等待直到收到可以安全离开 Safe Region 的信号为止；

再谈引用

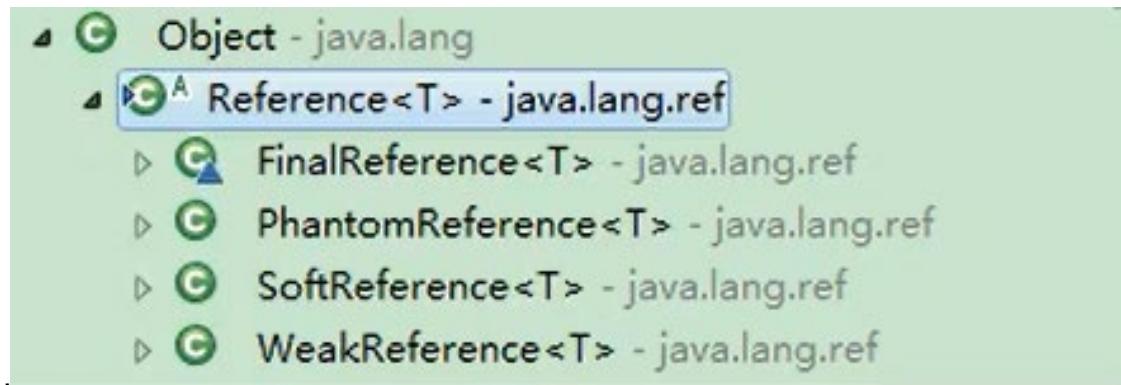
我们希望能描述这样一类对象：当内存空间还足够时，则能保留在内存中；如果内存空间在进行垃圾收集后还是很紧张，则可以抛弃这些对象。

【既偏门又非常高频的面试题】强引用、软引用、弱引用、虚引用有什么区别？具体使用场景是什么？在 JDK1.2 版之后，Java 对引用的概念进行了扩充，将引用分为：

- 强引用（Strong Reference）
- 软引用（Soft Reference）
- 弱引用（Weak Reference）
- 虚引用（Phantom Reference）

强软弱虚

这 4 种引用强度依次逐渐减弱。除强引用外，其他 3 种引用均可以在 `java.lang.ref` 包中找到它们的身影。如下图，显示了这 3 种引用类型对应的类，开发人员可以在应用程序中直接使用它们。



`Reference` 子类中只有终结器引用是包内可见的，其他 3 种引用类型均为 `public`，可以在应用程序中直接使用

- 强引用（`StrongReference`）：最传统的“引用”的定义，是指在程序代码之中普遍存在的引用赋值，即类似“`object obj=new Object()`”这种引用关系。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（`SoftReference`）：在系统将要发生内存溢出之前，将会把这些对象列入回收范围之中进行第二次回收。如果这次回收后还没有足够的内存，才会抛出内存流出异常。 内存不足即回收
- 弱引用（`WeakReference`）：被弱引用关联的对象只能生存到下一次垃圾收集之前。当垃圾收集器工作时，无论内存空间是否足够，都会回收掉被弱引用关联的对象。 垃圾回收必收集
- 虚引用（`PhantomReference`）：一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用获得一个对象的实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

再谈引用：强引用

在 Java 程序中，最常见的引用类型是强引用（普通系统 99% 以上都是强引用），也就是我们最常见的普通对象引用，也是默认的引用类型。

当在 Java 语言中使用 `new` 操作符创建一个新的对象，并将其赋值给一个变量的时候，这个变量就成为指向该对象的一个强引用。

强引用的对象是可触及的，垃圾收集器就永远不会回收掉被引用的对象。

对于一个普通的对象，如果没有其他的引用关系，只要超过了引用的作用域或者显式地将相应（强）引用赋值为 null，就是可以当做垃圾被收集了，当然具体回收时机还是要看垃圾收集策略。

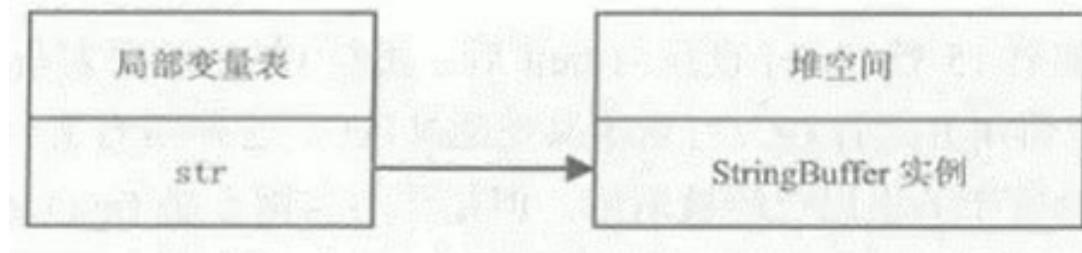
相对的，软引用、弱引用和虚引用的对象是软可触及、弱可触及和虚可触及的，在一定条件下，都是可以被回收的。所以，强引用是造成 Java 内存泄漏的主要原因之一。

举例

强引用的案例说明

```
StringBuffer str = new StringBuffer("hello mogublog");
```

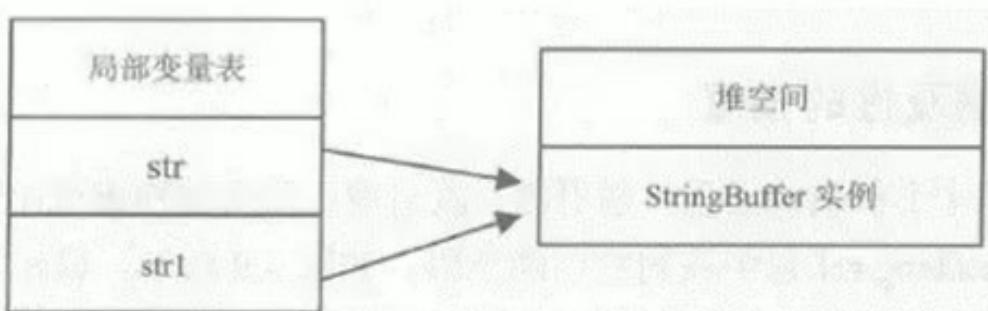
局部变量 str 指向 stringBuffer 实例所在堆空间，通过 str 可以操作该实例，那么 str 就是 stringBuffer 实例的强引用对应内存结构：



如果此时，在运行一个赋值语句

```
StringBuffer str = new StringBuffer("hello mogublog");
StringBuffer str1 = str;
```

对应的内存结构为：



那么我们将 str = null; 则 原来堆中的对象也不会被回收，因为还有其它对象指向该区域

总结

本例中的两个引用，都是强引用，强引用具备以下特点：

- 强引用可以直接访问目标对象。
- 强引用所指向的对象在任何时候都不会被系统回收，虚拟机宁愿抛出 OOM 异常，也不会回收强引用所指向对象。
- 强引用可能导致内存泄漏。

再谈引用： 软引用

内存足够的时候也不会回收软引用

软引用是用来描述一些还有用，但非必需的对象。只被软引用关联着的对象，在系统将要发生内存溢出异常前，会把这些对象列进回收范围之中进行第二次回收，如果这次回收还没有足够的内存，才会抛出内存溢出异常。

[软引用不导致OOM](#)

注意，这里的第一次回收是不可达的对象

软引用通常用来实现内存敏感的缓存。比如：高速缓存就有用到软引用。如果还有空闲内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。

垃圾回收器在某个时刻决定回收软可达的对象的时候，会清理软引用，并可选地把引用存放到一个引用队列（Reference Queue）。

类似弱引用，只不过 Java 虚拟机会尽量让软引用的存活时间长一些，迫不得已才清理。

一句话概括：当内存足够时，不会回收软引用可达的对象。内存不够时，会回收软引用的可达对象

在 JDK1.2 版之后提供了 SoftReference 类来实现软引用

```
// 声明强引用
Object obj = new Object();
// 创建一个软引用
SoftReference<Object> sf = new SoftReference<>(obj);
obj = null; // 销毁强引用，这是必须的，不然会存在强引用和软引用
```

再谈引用： 弱引用

发现即回收

弱引用也是用来描述那些非必需对象，被弱引用关联的对象只能生存到下一次垃圾收集发生为止。在系统 GC 时，只要发现弱引用，不管系统堆空间使用是否充足，都会回收掉只被弱引用关联的对象。

但是，由于垃圾回收器的线程通常优先级很低，因此，并不一定能很快地发现持有弱引用的对象。在这种情况下，弱引用对象可以存在较长的时间。

弱引用和软引用一样，在构造弱引用时，也可以指定一个引用队列，当弱引用对象被回收时，就会加入指定的引用队列，通过这个队列可以跟踪对象的回收情况。

软引用、弱引用都非常适合来保存那些可有可无的缓存数据。如果这么做，当系统内存不足时，这些缓存数据会被回收，不会导致内存溢出。而当内存资源充足时，这些缓存数据又可以存在相当长的时间，从而起到加速系统的作用。

在 JDK1.2 版之后提供了 WeakReference 类来实现弱引用

```
// 声明强引用
Object obj = new Object();
// 创建一个弱引用
WeakReference<Object> sf = new WeakReference<>(obj);
obj = null; // 销毁强引用，这是必须的，不然会存在强引用和弱引用
```

弱引用对象与软引用对象的最大不同就在于，当 GC 在进行回收时，需要通过算法检查是否回收软引用对象，而对于弱引用对象，GC 总是进行回收。弱引用对象更容易、更快被 GC 回收。

面试题：你开发中使用过 WeakHashMap 吗？ 弱引用

WeakHashMap 用来存储图片信息，可以在内存不足的时候，及时回收，避免了 OOM

再谈引用：虚引用

对象回收跟踪

也称为“幽灵引用”或者“幻影引用”，是所有引用类型中最弱的一个

一个对象是否有虚引用的存在，完全不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它和没有引用几乎是一样的，随时都可能被垃圾回收器回收。

它不能单独使用，也无法通过虚引用来获取被引用的对象。**当试图通过虚引用的 get() 方法取得对象时，总是 null**

为一个对象设置虚引用关联的唯一目的在于跟踪垃圾回收过程。比如：能在这个对象被收集器回收时收到一个系统通知。

虚引用必须和引用队列一起使用。虚引用在创建时必须提供一个引用队列作为参数。当垃圾回收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，将这个虚引用加入引用队列，以通知应用程序对象的回收情况。

由于虚引用可以跟踪对象的回收时间，因此，也可以将一些资源释放操作放置在虚引用中执行和记录。

虚引用无法获取到我们的数据

在 JDK1.2 版之后提供了 PhantomReference 类来实现虚引用。

```
// 声明强引用
Object obj = new Object();
// 声明引用队列
ReferenceQueue phantomQueue = new ReferenceQueue();
// 声明虚引用（还需要传入引用队列）
PhantomReference<Object> sf = new PhantomReference<>(obj, phantomQueue);
obj = null;
```

案例

我们使用一个案例，来结合虚引用，引用队列， finalize 进行讲解

```
/*
 * @author: 陌溪
 * @create: 2020-07-12-21:42
 */
public class PhantomReferenceTest {
    // 当前类对象的声明
    public static PhantomReferenceTest obj;
    // 引用队列
    static ReferenceQueue<PhantomReferenceTest> phantomQueue = null;

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("调用当前类的 finalize 方法");
        obj = this;
    }

    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            while(true) {
                if (phantomQueue != null) {
                    PhantomReference<PhantomReferenceTest> objt = null;
                    try {
                        objt = (PhantomReference<PhantomReferenceTest>)
phantomQueue.remove();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    if (objt != null) {
                        System.out.println("追踪垃圾回收过程: PhantomRefe
```

```
renceTest 实例被 GC 了");
        }
    }
}, "t1");
thread.setDaemon(true);
thread.start();

phantomQueue = new ReferenceQueue<>();
obj = new PhantomReferenceTest();
// 构造了 PhantomReferenceTest 对象的虚引用，并指定了引用队列
PhantomReference<PhantomReferenceTest> phantomReference = new PhantomReference<>(obj, phantomQueue);
try {
    System.out.println(phantomReference.get());
    // 去除强引用
    obj = null;
    // 第一次进行GC，由于对象可复活，GC 无法回收该对象
    System.out.println("第一次 GC 操作");
    System.gc();
    Thread.sleep(1000);
    if (obj == null) {
        System.out.println("obj 是 null");
    } else {
        System.out.println("obj 不是 null");
    }
    System.out.println("第二次 GC 操作");
    obj = null;
    System.gc();
    Thread.sleep(1000);
    if (obj == null) {
        System.out.println("obj 是 null");
    } else {
        System.out.println("obj 不是 null");
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
}
```

最后运行结果

null

第一次 GC 操作

调用当前类的 finalize 方法

`obj` 不是 `null`

第二次 GC 操作

追踪垃圾回收过程: `PhantomReferenceTest` 实例被 GC 了

`obj` 是 `null`

从上述运行结果我们知道, 第一次尝试获取虚引用的值, 发现无法获取的, 这是因为虚引用是无法直接获取对象的值, 然后进行第一次 `gc`, 因为会调用 `finalize` 方法, 将对象复活了, 所以对象没有被回收, 但是调用第二次 `gc` 操作的时候, 因为 `finalize` 方法只能执行一次, 所以就触发了 GC 操作, 将对象回收了, 同时将会触发第二个操作就是将回收的值存入到引用队列中。

终结器引用

`FinalReference`

它用于实现对象的 `finalize()` 方法, 也可以称为终结器引用

无需手动编码, 其内部配合引用队列使用

在 GC 时, 终结器引用入队。由 `Finalizer` 线程通过终结器引用找到被引用对象调用它的 `finalize()` 方法, 第二次 GC 时才回收被引用的对象

垃圾回收器

GC 分类与性能指标

垃圾收集器没有在规范中进行过多的规定, 可以由不同的厂商、不同版本的 JVM 来实现。

由于 JDK 的版本处于高速迭代过程中, 因此 Java 发展至今已经衍生了众多的 GC 版本。

从不同角度分析垃圾收集器, 可以将 GC 分为不同的类型。

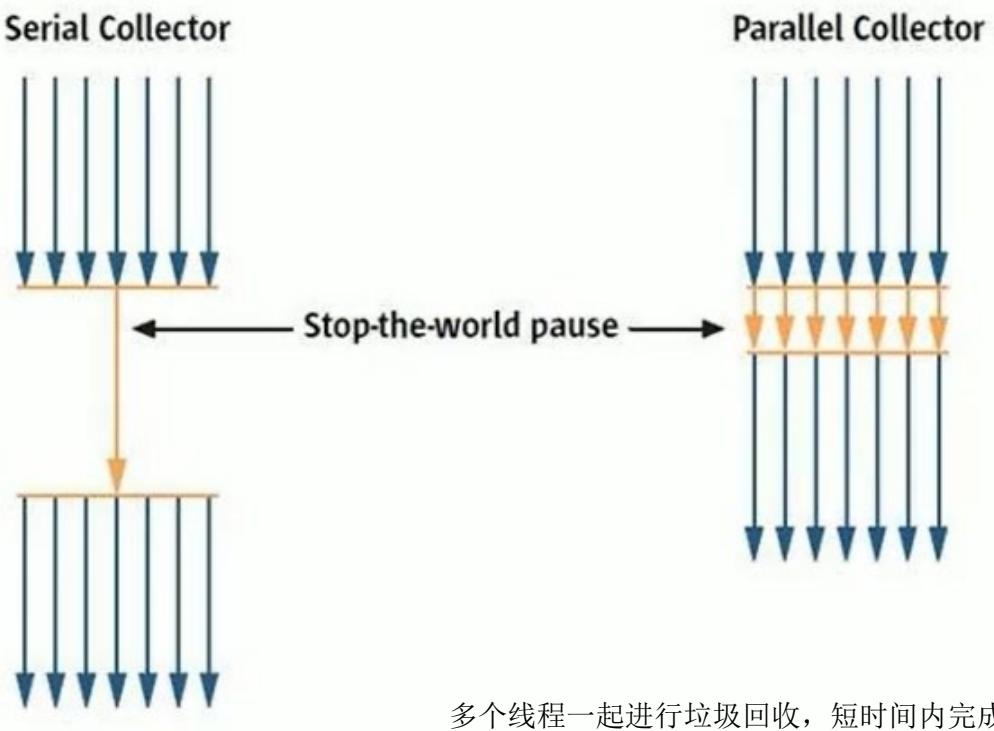
Java 不同版本新特性

- 语法层面: `Lambda` 表达式、`switch`、自动拆箱装箱、`enum`
- API 层面: `Stream API`、新的日期时间、`Optional`、`String`、集合框架
- 底层优化: JVM 优化、GC 的变化、元空间、静态域、字符串常量池位置变化

垃圾收集器分类

按线程数分

按线程数分（垃圾回收线程数），可以分为串行垃圾回收器和并行垃圾回收器。



串行回收指的是在同一时间段内只允许有一个 CPU 用于执行垃圾回收操作，此时工作线程被暂停，直至垃圾收集工作结束。

- 在诸如单 CPU 处理器或者较小的应用内存等硬件平台不是特别优越的场合，串行回收器的性能表现可以超过并行回收器和并发回收器。所以，串行回收默认被应用在客户端的 Client 模式下的 JVM 中
- 在并发能力比较强的 CPU 上，并行回收器产生的停顿时间要短于串行回收器。

和串行回收相反，并行收集可以运用多个 CPU 同时执行垃圾回收，因此提升了应用的吞吐量，不过并行回收仍然与串行回收一样，采用独占式，使用了“stop-the-world”机制。

按工作模式分

按照工作模式分，可以分为并发式垃圾回收器和独占式垃圾回收器。

- 并发式垃圾回收器与应用程序线程交替工作，以尽可能减少应用程序的停顿时间。

- 独占式垃圾回收器（Stop the world）一旦运行，就停止应用程序中的所有用户线程，直到垃圾回收过程完全结束。



按碎片处理方式分

按碎片处理方式分，可分为压缩式垃圾回收器和非压缩式垃圾回收器。

- 压缩式垃圾回收器会在回收完成后，对存活对象进行压缩整理，消除回收后的碎片。再分配对象空间时使用：指针碰撞
- 非压缩式的垃圾回收器不进行这步操作。再分配对象空间时使用：空闲列表。

按工作的内存区间分，又可分为年轻代垃圾回收器和老年代垃圾回收器。

评估 GC 的性能指标

- 吞吐量：**运行用户代码的时间占总运行时间的比例（总运行时间 = 程序的运行时间 + 内存回收的时间）
- 垃圾收集开销：**吞吐量的补数，垃圾收集所用时间与总运行时间的比例。
- 暂停时间：**执行垃圾收集时，程序的工作线程被暂停的时间。STW
- 收集频率：**相对于应用程序的执行，收集操作发生的频率。
- 内存占用：**Java 堆区所占的内存大小。
- 快速：**一个对象从诞生到被回收所经历的时间。

吞吐量、暂停时间、内存占用 这三者共同构成一个“不可能三角”。三者总体的表现会随着技术进步而越来越好。一款优秀的收集器通常最多同时满足其中的两项。

这三项里，暂停时间的重要性日益凸显。因为随着硬件发展，内存占用多些越来越能容忍，硬件性能的提升也有助于降低收集器运行时对应用程序的影响，即提高了吞吐量。而内存的扩大，对延迟反而带来负面效果。简单来说，主要抓住两点：

这三项里，暂停时间的重要性日益凸显。因为随着硬件发展，内存占用多些越来越能容忍，硬件性能的提升也有助于降低收集器运行时对应用程序的影响，即提高了吞吐量。而内存的扩大，对延迟反而带来负面效果。简单来说，主要抓住两点：

- 吞吐量
- 暂停时间

性能指标：吞吐量

吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量=运行用户代码时间 / (运行用户代码时间+垃圾收集时间)

比如：虚拟机总共运行了 100 分钟，其中垃圾收集花掉 1 分钟，那吞吐量就是 99%。

这种情况下，应用程序能容忍较高的暂停时间，因此，高吞吐量的应用程序有更长的时间基准，快速响应是不必考虑的

吞吐量优先，意味着在单位时间内，STW 的时间最短： $0.2+0.2=0.4$



性能指标：暂停时间

“暂停时间”是指一个时间段内应用程序线程暂停，让 Gc 线程执行的状态

例如，GC 期间 100毫秒的暂停时间意味着在这 100 毫秒期间内没有应用程序线程是活动的。暂停时间优先，意味着尽可能让单次 STW 的时间最短： $0.1+0.1 + 0.1+ 0.1+ 0.1=0.5$



吞吐量 vs 暂停时间

高吞吐量较好因为这会让应用程序的最终用户感觉只有应用程序线程在做“生产性”工作。直觉上，吞吐量越高程序运行越快。

低暂停时间（低延迟）较好因为从最终用户的角度来看不管是 GC 还是其他原因导致一个应用被挂起始终是不好的。这取决于应用程序的类型，有时候甚至短暂的 200 毫秒暂停都可能打断终端用户体验。因此，具有低的较大暂停时间是非常重要的，特别是对于一个交互式应用程序。

不幸的是“高吞吐量”和“低暂停时间”是一对相互竞争的目标（矛盾）。

因为如果选择以吞吐量优先，那么必然需要降低内存回收的执行频率，但是这样会导致 GC 需要更长的暂停时间来执行内存回收。

相反的，如果选择以低延迟优先为原则，那么为了降低每次执行内存回收时的暂停时间，也只能频繁地执行内存回收，但这又引起了年轻代内存的缩减和导致程序吞吐量的下降。

在设计（或使用）GC 算法时，我们必须确定我们的目标：一个 GC 算法只可能针对两个目标之一（即只专注于较大吞吐量或最小暂停时间），或尝试找到一个二者的折衷。

现在标准：**在最大吞吐量优先的情况下，降低停顿时间**

不同的垃圾回收器概述

垃圾收集机制是 Java 的招牌能力，极大地提高了开发效率。这当然也是面试的热点。

那么，Java 常见的垃圾收集器有哪些？

GC 垃圾收集器是和 JVM 一脉相承的，它是和 JVM 进行搭配使用，在不同的使用场景对应的收集器也是有区别

垃圾回收器发展史

有了虚拟机，就一定需要收集垃圾的机制，这就是 Garbage Collection，对应的产品我们称为 Garbage Collector。

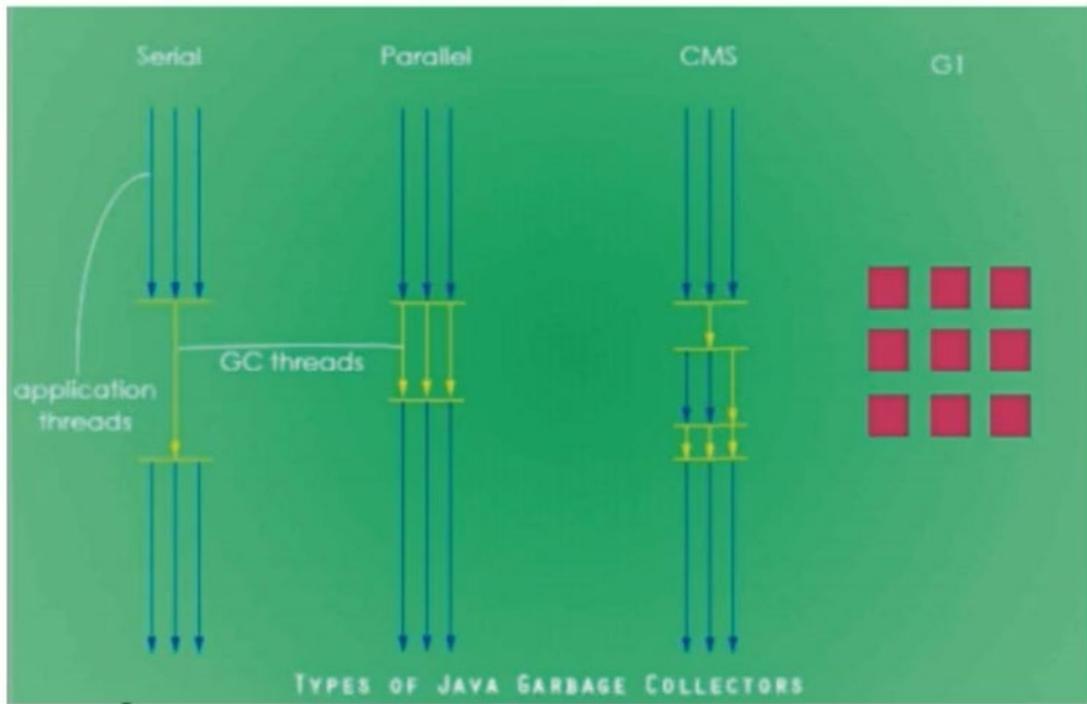
- 1999 年随 JDK1.3.1 一起来的是串行方式的 serialGc，它是第一款 GC。ParNew 垃圾收集器是 Serial 收集器的多线程版本
- 2002 年 2 月 26 日，Parallel GC 和 Concurrent Mark Sweep GC 跟随 JDK1.4.2 一起发布。
- Parallel GC 在 JDK6 之后成为 HotSpot 默认 GC。
- 2012 年，在 JDK1.7u4 版本中，G1 可用。
- 2017 年，JDK9 中 G1 变成默认的垃圾收集器，以替代 CMS。
- 2018 年 3 月，JDK10 中 G1 垃圾回收器的并行完整垃圾回收，实现并行性来改善最坏情况下的延迟。
- 2018 年 9 月，JDK11 发布。引入 Epsilon 垃圾回收器，又被称为 "No-Op(无操作)" 回收器。同时，引入 ZGC：可伸缩的低延迟垃圾回收器（Experimental）
- 2019 年 3 月，JDK12 发布。增强 G1，自动返回未用堆内存给操作系统。同时，引入 Shenandoah GC：低停顿时间的 GC（Experimental）。·2019 年 9 月，JDK13 发布。增强 zGC，自动返回未用堆内存给操作系统。
- 2020 年 3 月，JDK14 发布。删除 cMs 垃圾回收器。扩展 zGC 在 macos 和 Windows 上的应用

7 种经典的垃圾收集器

- 串行回收器：Serial、Serial old
- 并行回收器：ParNew、Parallel Scavenge、Parallel old
- 并发回收器：CMS、G11

并行回收器：多个垃圾回收线程一起并行执行

并发回收器：垃圾回收线程与用户线程交替执行



7 款经典收集器与垃圾分代之间的关系

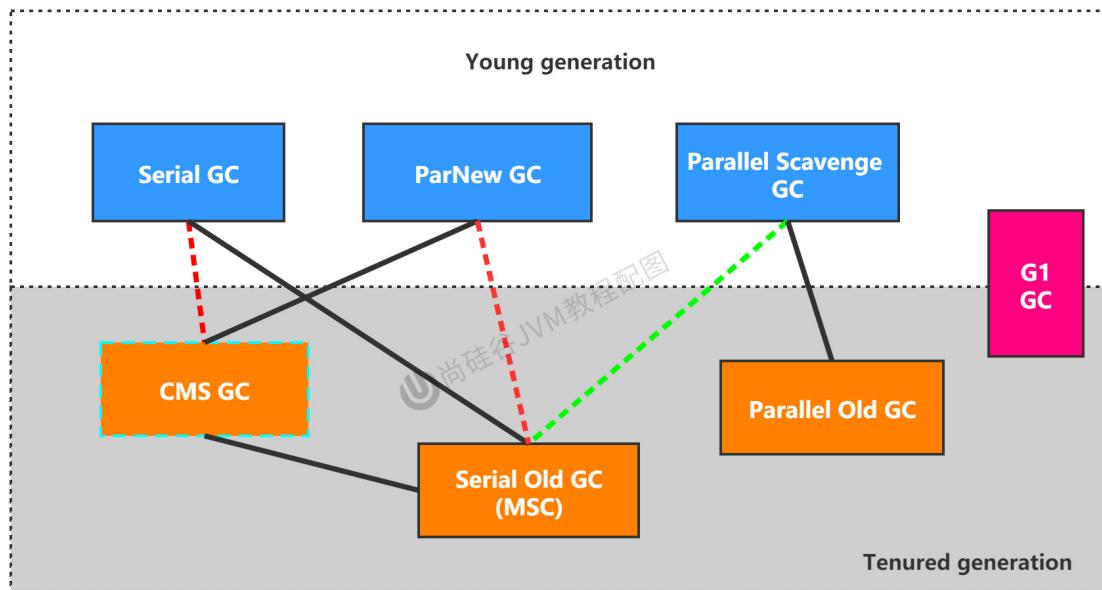


新生代收集器: Serial、ParNew、Parallel Scavenge;

老年代收集器: Serial old、Parallel old、CMS;

整堆收集器: G1;

垃圾收集器的组合关系



- 两个收集器间有连线，表明它们可以搭配使用：Serial/Serial old、Serial/CMS、ParNew/Serial old、ParNew/CMS、Parallel Scavenge/Serial old、Parallel Scavange/Parallel old、G1；
- 其中 Serial old 作为 CMS 出现"Concurrent Mode Failure"失败的后备预案。
- (红色虚线) 由于维护和兼容性测试的成本，在 JDK 8 时将 Serial+CMS、ParNew+Serial old 这两个组合声明为废弃 (JEP173)，并在 JDK9 中完全取消了这些组合的支持 (JEP214)，即：移除。
- (绿色虚线) JDK14 中：弃用 Parallel Scavange 和 Serialold GC 组合 (JEP366)
- (青色虚线) JDK14 中：删除 CMs 垃圾回收器 (JEP363)

为什么要有很多收集器，一个不够吗？因为 Java 的使用场景很多，移动端，服务器等。所以就需要针对不同的场景，提供不同的垃圾收集器，提高垃圾收集的性能。

虽然我们会对各个收集器进行比较，但并非为了挑选一个最好的收集器出来。没有一种放之四海皆准、任何场景下都适用的完美收集器存在，更加没有万能的收集器。所以我们选择的只是对具体应用最合适的收集器。

如何查看默认垃圾收集器

-XX:+PrintCommandLineFlags: 查看命令行相关参数（包含使用的垃圾收集器）

使用命令行指令: jinfo -flag 相关垃圾回收器参数 进程 ID

Serial 回收器: 串行回收

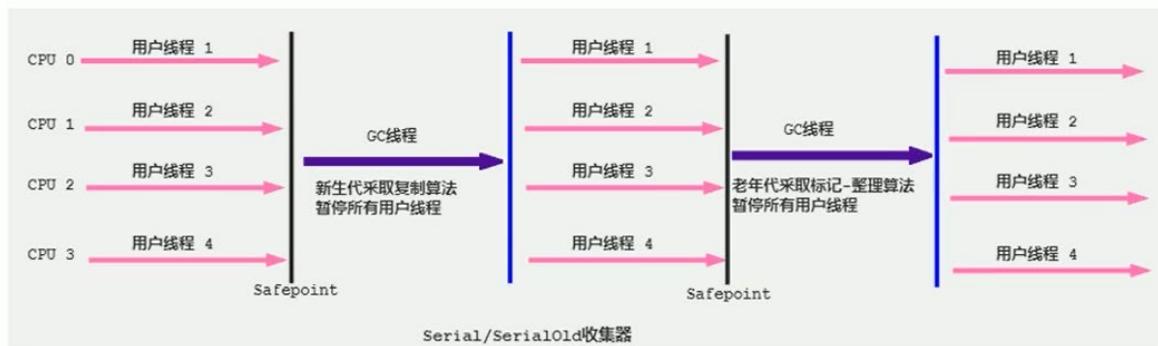
Serial 收集器是最基本、历史最悠久的垃圾收集器了。JDK1.3 之前回收新生代唯一的选择。

Serial 收集器作为 HotSpot 中 client 模式下的默认新生代垃圾收集器。

Serial 收集器采用复制算法、串行回收和"stop-the-World"机制的方式执行内存回收。

除了年轻代之外，Serial 收集器还提供用于执行老年代垃圾收集的 Serial old 收集器。Serial old 收集器同样也采用了串行回收和"stop the World"机制，只不过内存回收算法使用的是标记-压缩算法。

- Serial old 是运行在 Client 模式下默认的老年代的垃圾回收器
- Serial Old 在 Server 模式下主要有两个用途：
 - 与新生代的 Parallel scavenge 配合使用
 - 作为老年代 CMS 收集器的后备垃圾收集方案



这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个 CPU 或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束（Stop The World）

优势：简单而高效（与其他收集器的单线程比），对于限定单个 CPU 的环境来说，Serial 收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

运行在 client 模式下的虚拟机是个不错的选择。

在用户的桌面应用场景中，可用内存一般不大（几十 MB 至一两百 MB），可以在较短时间内完成垃圾收集（几十 ms 至一百多 ms），只要不频繁发生，使用串行回收器是可以接受的。

在 HotSpot 虚拟机中，使用-XX: +UseSerialGC 参数可以指定年轻代和老年代都使用串行收集器。

等价于新生代用 Serial GC，且老年代用 Serial old GC

总结

这种垃圾收集器大家了解，现在已经不用串行的了。而且在限定单核 cpu 才可以用。现在都不是单核的了。

对于交互较强的应用而言，这种垃圾收集器是不能接受的。一般在 Java web 应用程序中是不会采用串行垃圾收集器的。

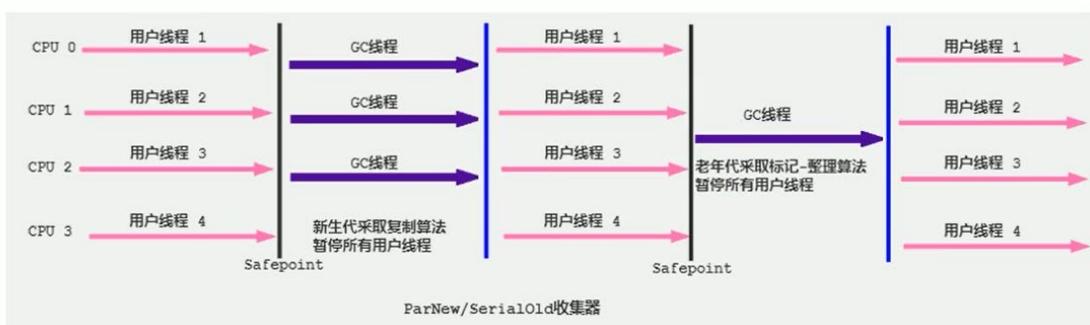
ParNew 回收器：并行回收

如果说 serialGC 是年轻代中的单线程垃圾收集器，那么 ParNew 收集器则是 serial 收集器的多线程版本。

- Par 是 Parallel 的缩写，New: 只能处理的是新生代

ParNew 收集器除了采用并行回收的方式执行内存回收外，两款垃圾收集器之间几乎没有丝毫区别。ParNew 收集器在年轻代中同样也是采用复制算法、"stop-the-World"机制。

ParNew 是很多 JVM 运行在 Server 模式下新生代的默认垃圾收集器。



- 对于新生代，回收次数频繁，使用并行方式高效。
- 对于老年代，回收次数少，使用串行方式节省资源。（CPU 并行需要切换线程，串行可以省去切换线程的资源）

由于 ParNew 收集器是基于并行回收，那么是否可以断定 ParNew 收集器的回收效率在任何场景下都会比 serial 收集器更高效？

- ParNew 收集器运行在多CPU的环境下，由于可以充分利用多CPU、多核心等物理硬件资源优势，可以更快速地完成垃圾收集，提升程序的吞吐量。
- 但是在单个CPU的环境下，ParNew 收集器不比 Serial 收集器更高效。Serial 收集器基于串行回收，CPU 不需要频繁地做任务切换，可以有效避免多线程交互过程中产生的一些额外开销。

因为除 Serial 外，目前只有 ParNew GC 能与 CMS 收集器配合工作

在程序中，开发人员可以通过选项"-XX: +UseParNewGC"手动指定使用 ParNew 收集器执行内存回收任务。它表示年轻代使用并行收集器，不影响老年代。

-XX:ParallelGCThreads 限制线程数量，默认开启和 CPU 数据相同的线程数。

Parallel 回收器：吞吐量优先

HotSpot 的年轻代中除了拥有 ParNew 收集器是基于并行回收的以外，Parallel Scavenge 收集器同样也采用了复制算法、并行回收和"Stop the World"机制。

那么 Parallel 收集器的出现是否多此一举？

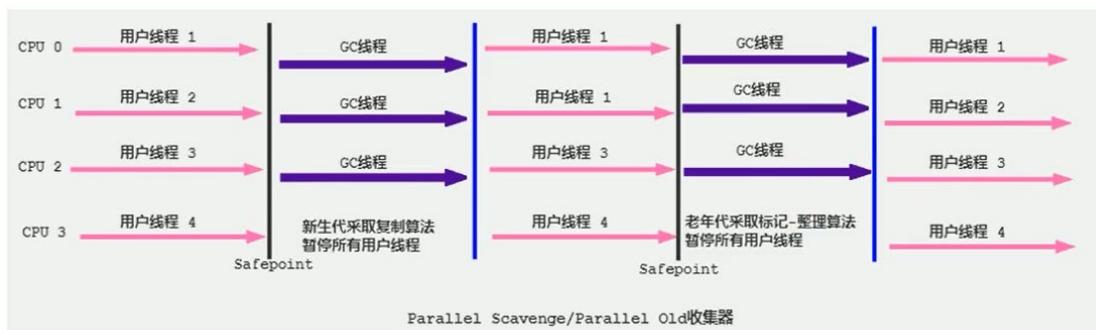
- 和 ParNew 收集器不同，Parallel Scavenge 收集器的目标则是达到一个可控的吞吐量（Throughput），它也被称为吞吐量优先的垃圾收集器。
- 自适应调节策略也是 Parallel Scavenge 与 ParNew 一个重要区别。

高吞吐量则可以高效率地利用 CPU 时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。因此，常见在服务器环境中使用。例如，那些执行批量处理、订单处理、工资支付、科学计算的应用程序。

高吞吐量——
后台运行

Parallel 收集器在 JDK1.6 时提供了用于执行老年代垃圾收集的 Parallel old 收集器，用来代替老年代的 Serial old 收集器。

Parallel old 收集器采用了标记-压缩算法，但同样也是基于并行回收和"stop-the-World"机制。



在程序吞吐量优先的应用场景中，Parallel 收集器和 Parallel old 收集器的组合，在 server 模式下的内存回收性能很不错。在 Java8 中，默认是此垃圾收集器。

参数配置

-XX: +UseParallelGC 手动指定年轻代使用 Parallel 并行收集器执行内存回收任务。

-XX: +UseParalleloldGC 手动指定老年代都是使用并行回收收集器。

- 分别适用于新生代和老年代。默认 jdk8 是开启的。
- 上面两个参数，默认开启一个，另一个也会被开启。（互相激活）

-XX:ParallelGcThreads 设置年轻代并行收集器的线程数。一般地，最好与 CPU 数量相等，以避免过多的线程数影响垃圾收集性能。

在默认情况下，当 CPU 数量小于 8 个，ParallelGcThreads 的值等于 CPU 数量。

当 CPU 数量大于 8 个，ParallelGcThreads 的值等于 $3+[5*\text{CPU Count}]/8]$

-XX:MaxGCPauseMillis 设置垃圾收集器最大停顿时间（即 STW 的时间）。单位是毫秒。

为了尽可能地把停顿时间控制在 MaxGCPauseMillis 以内，收集器在工作时会调整 Java 堆大小或者其他一些参数。对于用户来讲，停顿时间越短体验越好。但是在服务器端，我们注重高并发，整体的吞吐量。所以服务器端适合 Parallel，进行控制。该参数使用需谨慎。

-XX:GCTimeRatio 垃圾收集时间占总时间的比例（= $1/(N+1)$ ）。用于衡量吞吐量的大小。

取值范围（0, 100）。默认值 99，也就是垃圾回收时间不超过 1。

与前一个-xx:MaxGCPauseMillis 参数有一定矛盾性。暂停时间越长，Radio 参数就容易超过设定的比例。

-XX:+UseAdaptiveSizePolicy 设置 Parallel scavenge 收集器具有自适应调节策略

在这种模式下，年轻代的大小、Eden 和 Survivor 的比例、晋升老年代的对象年龄等参数会被自动调整，已达到在堆大小、吞吐量和停顿时间之间的平衡点。

在手动调优比较困难的场合，可以直接使用这种自适应的方式，仅指定虚拟机的最大堆、目标的吞吐量（GCTimeRatio）和停顿时间（MaxGCPauseMillis），让虚拟机自己完成调优工作。

CMS 回收器：低延迟

在 JDK1.5 时期，Hotspot 推出了一款在强交互应用中几乎可认为有划时代意义的垃圾收集器：CMS（Concurrent-Mark-Sweep）收集器，这款收集器是 HotSpot 虚拟机中第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程同时工作。

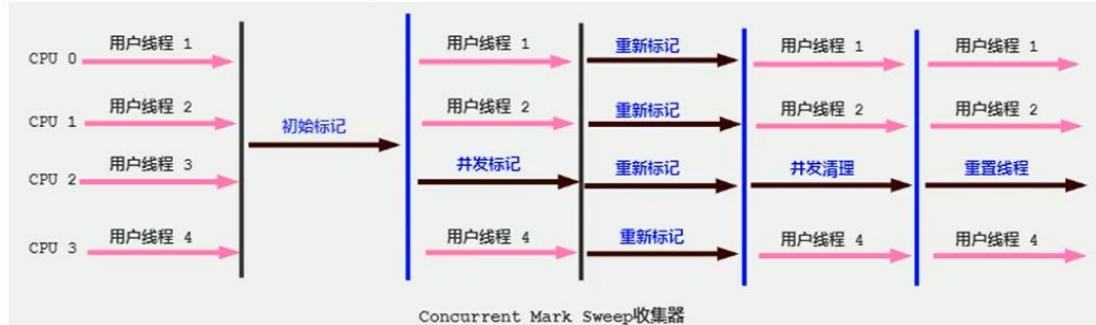
CMS 收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间。停顿时间越短（低延迟）就越适合与用户交互的程序，良好的响应速度能提升用户体验。

目前很大一部分的 Java 应用集中在互联网站或者 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS 收集器就非常符合这类应用的需求。

CMS 的垃圾收集算法采用标记-清除算法，并且也会"stop-the-world"

不幸的是，CMS 作为老年代的收集器，却无法与 JDK1.4.0 中已经存在的新生代收集器 Parallel Scavenge 配合工作，所以在 JDK1.5 中使用 CMS 来收集老年代的时候，新生代只能选择 ParNew 或者 Serial 收集器中的一个。

在 G1 出现之前，CMS 使用还是非常广泛的。一直到今天，仍然有很多系统使用 CMS GC。



CMS 整个过程比之前的收集器要复杂，整个过程分为 4 个主要阶段，即初始标记阶段、并发标记阶段、重新标记阶段和并发清除阶段。（涉及 STW 的阶段主要是：初始标记 和 重新标记）

- **初始标记（Initial-Mark）阶段：**在这个阶段中，程序中所有的工作线程都将会因为“stop-the-world”机制而出现短暂的暂停，这个阶段的主要任务仅仅只是标记出 **GCRoots** 能直接关联到的对象。一旦标记完成之后就会恢复之前被暂停的所有应用线程。由于直接关联对象比较小，所以这里的速度非常快。

- **并发标记 (Concurrent-Mark)** 阶段：从 Gc Roots 的直接关联对象开始遍历整个对象图的过程，这个过程耗时较长但是不需要停顿用户线程，可以与垃圾收集线程一起并发运行。
- **重新标记 (Remark)** 阶段：由于在并发标记阶段中，程序的工作线程会和垃圾收集线程同时运行或者交叉运行，因此为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间通常会比初始标记阶段稍长一些，但也远比并发标记阶段的时间短。
- **并发清除 (Concurrent-Sweep)** 阶段：此阶段清理删除掉标记阶段判断的已经死亡的对象，释放内存空间。由于不需要移动存活对象，所以这个阶段也是可以与用户线程同时并发的

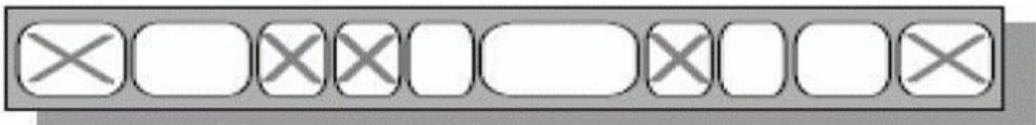
尽管 CMS 收集器采用的是并发回收（非独占式），但是在其初始化标记和再次标记这两个阶段中仍然需要执行“Stop-the-World”机制暂停程序中的工作线程，不过暂停时间并不会太长，因此可以说明目前所有的垃圾收集器都做不到完全不需要“stop-the-World”，只是尽可能地缩短暂停时间。

由于最耗费时间的并发标记与并发清除阶段都不需要暂停工作，所以整体的回收是低停顿的。

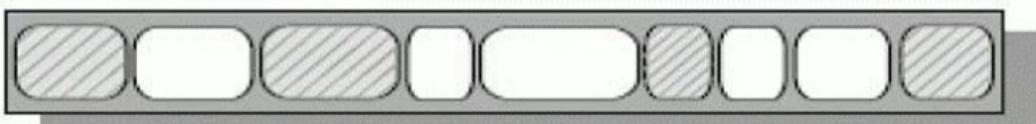
另外，由于在垃圾收集阶段用户线程没有中断，所以在 CMS 回收过程中，还应该确保应用程序用户线程有足够的内存可用。因此，CMS 收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，而是当堆内存使用率达到某一阈值时，便开始进行回收，以确保应用程序在 CMS 工作过程中依然有足够的空间支持应用程序运行。要是 CMS 运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用 Serial old 收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

CMS 收集器的垃圾收集算法采用的是**标记清除算法**，这意味着每次执行完内存回收后，由于被执行内存回收的无用对象所占用的内存空间极有可能是不连续的一些内存块，不可避免地将会产生一些内存碎片。那么 CMS 在为新对象分配内存空间时，将无法使用指针碰撞（Bump the Pointer）技术，而只能选择空闲列表（Free List）执行内存分配。

a) Start of Sweeping



b) End of Sweeping



CMS 为什么不使用标记整理算法？

答案其实很简答，因为当并发清除的时候，用 Compact 整理内存的话，原来的用户线程使用的内存还怎么用呢？要保证用户线程能继续执行，前提的它运行的资源不受影响嘛。Mark Compact 更适合“stop the world”这种场景下使用

优点

- 并发收集
- 低延迟

缺点

- 会产生内存碎片，导致并发清除后，用户线程可用的空间不足。在无法分配大对象的情况下，不得不提前触发 Full GC。
- CMS 收集器对 CPU 资源非常敏感。在并发阶段，它虽然不会导致用户停顿，但是会因为占用了一部分线程而导致应用程序变慢，总吞吐量会降低。
- CMS 收集器无法处理浮动垃圾。可能出现“Concurrent Mode Failure”失败而导致另一次 Full GC 的产生。在并发标记阶段由于程序的工作线程和垃圾收集线程是同时运行或者交叉运行的，那么在并发标记阶段如果产生新的垃圾对象，CMS 将无法对这些垃圾对象进行标记，最终会导致这些新产生的垃圾对象没有被及时回收，从而只能在下一次执行 GC 时释放这些之前未被回收的内存空间。
浮动垃圾：并发标记和并发清理阶段，用户线程产生的新垃圾

设置的参数

- -XX: +UseConcMarkSweepGC 手动指定使用 CMS 收集器执行内存回收任务。

开启该参数后会自动将-xx: +UseParNewGC 打开。即：ParNew（Young 区用）+CMS（old 区用）+Serial old 的组合。

- `-XX:CMSInitiatingoccupanyFraction` 设置堆内存使用率的阈值，一旦达到该阈值，便开始进行回收。

JDK5 及以前版本的默认值为 68，即当老年代的空间使用率达到 68% 时，会执行一次 CMS 回收。JDK6 及以上版本默认值为 92%

如果内存增长缓慢，则可以设置一个稍大的值，大的阈值可以有效降低 CMS 的触发频率，减少老年代回收的次数可以较为明显地改善应用程序性能。反之，如果应用程序内存使用率增长很快，则应该降低这个阈值，以避免频繁触发老年代串行收集器。因此通过该选项便可以有效降低 Full GC 的执行次数。

- `-XX: +UseCMSCompactAtFullCollection` 用于指定在执行完 Full GC 后对内存空间进行压缩整理，以此避免内存碎片的产生。不过由于内存压缩整理过程无法并发执行，所带来的问题就是停顿时间变得更长了。
- `-XX:CMSFullGCsBeforecompaction` 设置在执行多少次 FullGC 后对内存空间进行压缩整理。
- `-XX:ParallelcMSThreads` 设置 CMS 的线程数量。

CMS 默认启动的线程数是 $(Parallel\ GCThreads+3) / 4$ ，`ParallelGCThreads` 是年轻代并行收集器的线程数。当 CPU 资源比较紧张时，受到 CMS 收集器线程的影响，应用程序的性能在垃圾回收阶段可能会非常糟糕。

小结

HotSpot 有这么多的垃圾回收器，那么如果有人问，Serial GC、Parallel GC、Concurrent Mark Sweep GC 这三个 Gc 有什么不同呢？

请记住以下口令：

- 如果你想要最小化地使用内存和并行开销，请选 Serial GC；
- 如果你想要最大化应用程序的吞吐量，请选 Parallel GC；
- 如果你想要最小化 GC 的中断或停顿时间，请选 CMS GC。

JDK 后续版本中 CMS 的变化

JDK9 新特性： CMS 被标记为 deprecated 了 (JEP291) >如果对 JDK9 及以上版本的 HotSpot 虚拟机使用参数 `-XX: +UseConcMarkSweepGC` 来开启 CMS 收集器的话，用户会收到一个警告信息，提示 CMS 未来将会被废弃。

JDK14 新特性： 删除 CMS 垃圾回收器 (JEP363) 移除了 CMS 垃圾收集器，如果在 JDK14 中使用 `XX: +UseConcMarkSweepGC` 的话，JVM 不会报错，只是给出一个 warning 信息，但是不会 exit。JVM 会自动回退以默认 GC 方式启动 JVM

G1 回收器：区域化分代式

既然我们已经有了前面几个强大的 GC，为什么还要发布 Garbage First (G1) ？

原因就在于应用程序所应对的业务越来越庞大、复杂，用户越来越多，没有 GC 就不能保证应用程序正常进行，而经常造成 STW 的 GC 又跟不上实际的需求，所以才会不断地尝试对 GC 进行优化。G1 (Garbage-First) 垃圾回收器是在 Java7 update4 之后引入的一个新的垃圾回收器，是当今收集器技术发展的最前沿成果之一。

与此同时，为了适应现在不断扩大的内存和不断增加的处理器数量，进一步降低暂停时间 (pause time)，同时兼顾良好的吞吐量。

官方给 G1 设定的目标是在延迟可控的情况下获得尽可能高的吞吐量，所以才担当起“全功能收集器”的重任与期望。

为什么名字叫 Garbage First(G1) 呢？

因为 G1 是一个并行回收器，它把堆内存分割为很多不相关的区域 (Region) (物理上不连续的)。使用不同的 Region 来表示 Eden、幸存者 0 区，幸存者 1 区，老年代等。

G1 GC 有计划地避免在整个 Java 堆中进行全区域的垃圾收集。G1 跟踪各个 Region 里面的垃圾堆积的价值大小 (回收所获得的空间大小以及回收所需时间的经验值)，在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。

由于这种方式的侧重点在于回收垃圾最大量的区间 (Region)，所以我们给 G1 一个名字：垃圾优先 (Garbage First)。

G1 (Garbage-First) 是一款面向服务端应用的垃圾收集器，主要针对配备多核 CPU 及大容量内存的机器，以极高概率满足 GC 停顿时间的同时，还兼具高吞吐量的性能特征。

在 JDK1.7 版本正式启用，移除了 Experimental 的标识，是 JDK9 以后的默认垃圾回收器，取代了 CMS 回收器以及 Parallel+Parallel old 组合。被 oracle 官方称为“全功能的垃圾收集器”。

与此同时，CMS 已经在 JDK9 中被标记为废弃 (deprecated)。在 jdk8 中还不是默认的垃圾回收器，需要使用 -xx: +UseG1GC 来启用。

G1 垃圾收集器的优点

与其他 GC 收集器相比，G1 使用了全新的分区算法，其特点如下所示：

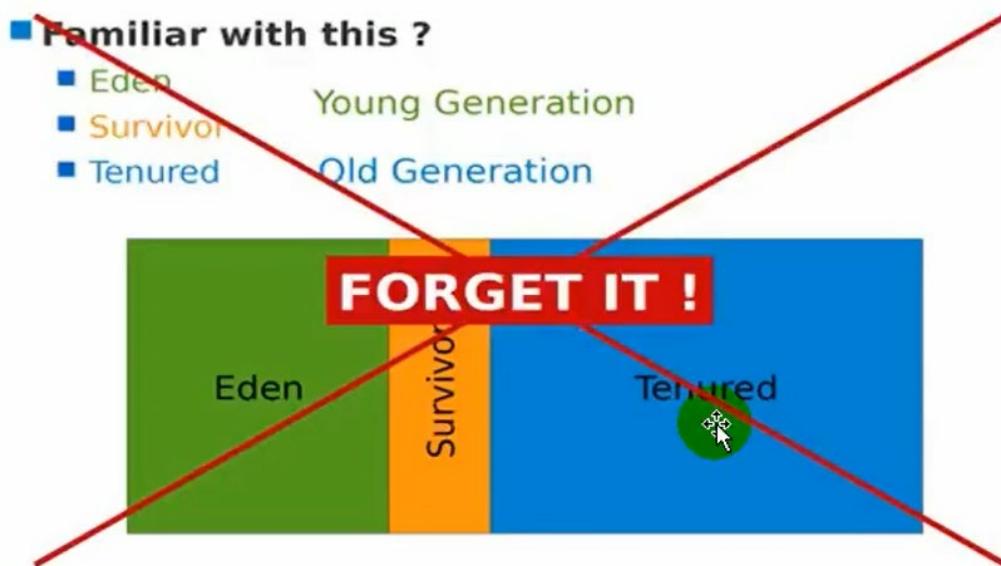
并行与并发

- 并行性：G1 在回收期间，可以有多个 GC 线程同时工作，有效利用多核计算能力。此时用户线程 STW
- 并发性：G1 拥有与应用程序交替执行的能力，部分工作可以和应用程序同时执行，因此，一般来说，不会在整个回收阶段发生完全阻塞应用程序的情况

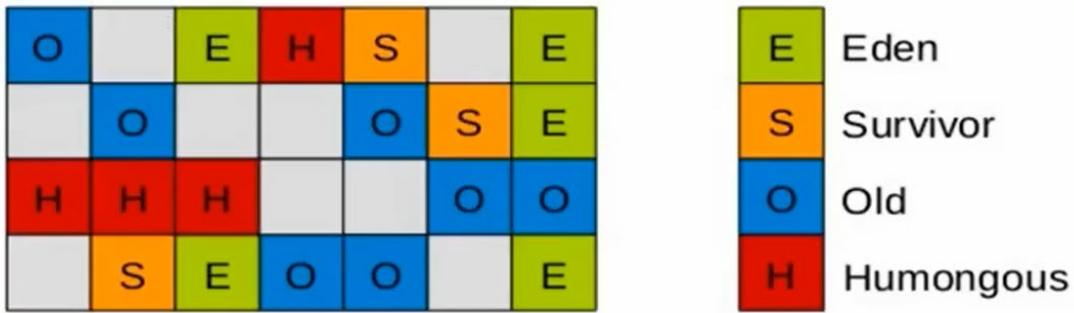
分代收集

- 从分代上看，G1 依然属于分代型垃圾回收器，它会区分年轻代和老年代，年轻代依然有 Eden 区和 Survivor 区。但从堆的结构上看，它不要求整个 Eden 区、年轻代或者老年代都是连续的，也不再坚持固定大小和固定数量。
- 将堆空间分为若干个区域（Region），这些区域中包含了逻辑上的年轻代和老年代。
- 和之前的各类回收器不同，它同时兼顾年轻代和老年代。对比其他回收器，或者工作在年轻代，或者工作在老年代；

G1 所谓的分代，已经不是下面这样的了



而是这样的一个区域



空间整合

- CMS：“标记-清除”算法、内存碎片、若干次 GC 后进行一次碎片整理
- G1 将内存划分为一个个的 region。内存的回收是以 region 作为基本单位的。Region 之间是复制算法，但整体上实际可看作是标记-压缩（Mark-Compact）算法，两种算法都可以避免内存碎片。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次 GC。尤其是当 Java 堆非常大的时候，G1 的优势更加明显。
实时：要求10ms完成GC就必须做到，
软实时：大概率在10ms内完成

可预测的停顿时间模型（即：软实时 soft real-time）这是 G1 相对于 CMS 的另一大优势，G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒。

- 由于分区的原因，G1 可以只选取部分区域进行内存回收，这样缩小了回收的范围，因此对于全局停顿情况的发生也能得到较好的控制。
- G1 跟踪各个 Region 里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。保证了 G1 收集器在有限的时间内可以获取尽可能高的收集效率。
- 相比于 CMSGC，G1 未必能做到 CMS 在最好情况下的延时停顿，但是最差情况要好很多。

G1 垃圾收集器的缺点

相较于 CMS，G1 还不具备全方位、压倒性优势。比如在用户程序运行过程中，G1 无论是为了垃圾收集产生的内存占用（Footprint）还是程序运行时的额外执行负载（overload）都要比 CMS 要高。

从经验上来说，在小内存应用上 CMS 的表现大概率会优于 G1，而 G1 在大内存应用上则发挥其优势。平衡点在 6-8GB 之间。

G1 参数设置

- -XX:+UseG1GC: 手动指定使用 G1 垃圾收集器执行内存回收任务
- -XX:G1HeapRegionSize 设置每个 Region 的大小。值是 2 的幂，范围是 1MB 到 32MB 之间，目标是根据最小的 Java 堆大小划分出约 2048 个区域。默认是堆内存的 1/2000。
- -XX:MaxGCPauseMillis 设置期望达到的最大 GC 停顿时间指标（JVM 会尽力实现，但不保证达到）。默认值是 200ms
- -XX:+ParallelGcThread 设置 STW 工作线程数的值。最多设置为 8
- -XX:ConcGCThreads 设置并发标记的线程数。将 n 设置为并行垃圾回收线程数（ParallelGCThreads）的 1/4 左右。
- -XX:InitiatingHeapoccupancyPercent 设置触发并发 GC 周期的 Java 堆占用率阈值。超过此值，就触发 GC。默认值是 45。

G1 收集器的常见操作步骤

G1 的设计原则就是简化 JVM 性能调优，开发人员只需要简单的三步即可完成调优：

- 第一步：开启 G1 垃圾收集器
- 第二步：设置堆的最大内存
- 第三步：设置最大的停顿时间

G1 中提供了三种垃圾回收模式：YoungGC、Mixed GC 和 FullGC，在不同的条件下被触发。

G1 收集器的适用场景

面向服务端应用，针对具有大内存、多处理器的机器。（在普通大小的堆里表现并不惊喜）

最主要的应用是需要低 GC 延迟，并具有大堆的应用程序提供解决方案；

如：在堆大小约 6GB 或更大时，可预测的暂停时间可以低于 0.5 秒；（G1 通过每次只清理一部分而不是全部的 Region 的增量式清理来保证每次 GC 停顿时间不会过长）。用来替换掉 JDK1.5 中的 CMS 收集器；在下面的情况时，使用 G1 可能比 CMS 好：

- 超过 50% 的 Java 堆被活动数据占用；

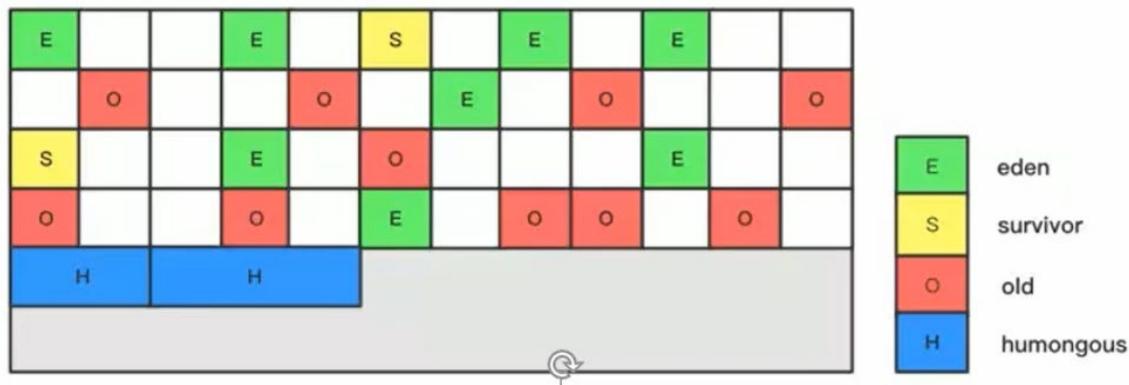
- 对象分配频率或年代提升频率变化很大；
- GC 停顿时间过长（长于 0.5 至 1 秒）

HotSpot 垃圾收集器里，除了 G1 以外，其他的垃圾收集器使用内置的 JVM 线程执行 GC 的多线程操作，而 G1GC 可以采用应用线程承担后台运行的 GC 工作，即当 JVM 的 GC 线程处理速度慢时，系统会调用应用程序线程帮助加速垃圾回收过程。

分区 Region：化整为零

使用 G1 收集器时，它将整个 Java 堆划分成约 2048 个大小相同的独立 Region 块，每个 Region 块大小根据堆空间的实际大小而定，整体被控制在 1MB 到 32MB 之间，且为 2 的 N 次幂，即 1MB, 2MB, 4MB, 8MB, 16MB, 32MB。可以通过 XX:G1HeapRegionsize 设定。所有的 Region 大小相同，且在 JVM 生命周期内不会被改变。

虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分 Region（不需要连续）的集合。通过 Region 的动态分配方式实现逻辑上的连续。



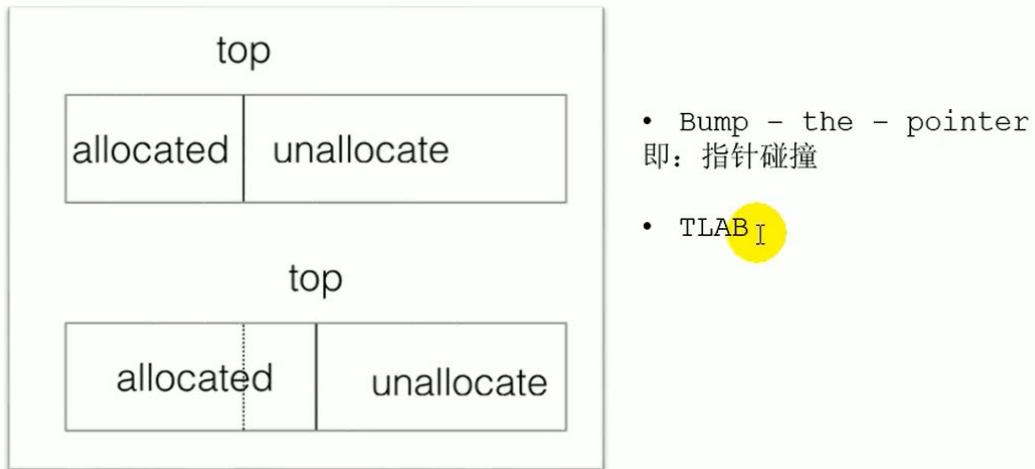
一个 region 有可能属于 Eden, Survivor 或者 old/Tenured 内存区域。但是一个 region 只可能属于一个角色。图中的 E 表示该 region 属于 Eden 内存区域，S 表示属于 survivor 内存区域，O 表示属于 old 内存区域。图中空白的表示未使用的内存空间。

G1 垃圾收集器还增加了一种新的内存区域，叫做 Humongous 内存区域，如图中的 H 块。主要用于存储大对象，如果超过 1.5 个 region，就放到 H。

设置 H 的原因：对于堆中的大对象，默认直接会被分配到老年代，但是如果它是一个短期存在的大对象，就会对垃圾收集器造成负面影响。为了解决这个问题，G1划分了一个 Humongous 区，它用来专门存放大对象。如果一个 H 区装不下一个大对

象，那么 G1 会寻找连续的 H 区来存储。为了能找到连续的 H 区，有时候不得不启动 FullGC。G1 的大多数行为都把 H 区作为老年代的一部分来看待。

每个 Region 都是通过指针碰撞来分配空间

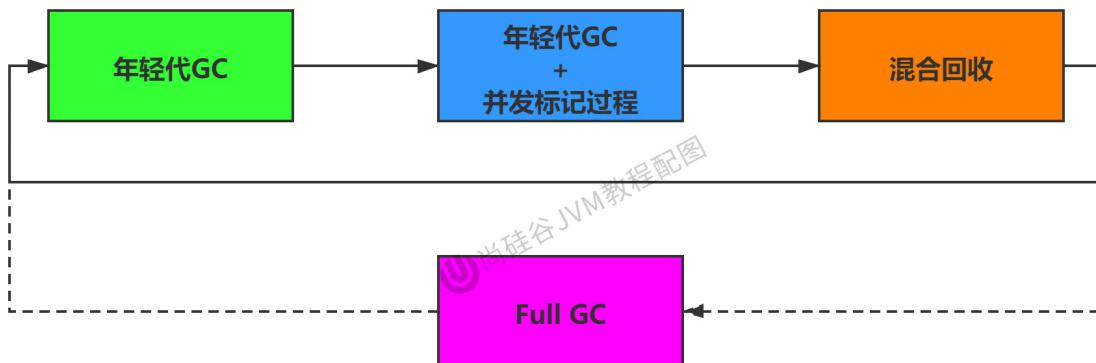


G1 垃圾回收器的回收过程

G1GC 的垃圾回收过程主要包括如下三个环节：

- 年轻代 GC (Young GC)
- 老年代并发标记过程 (Concurrent Marking)
- 混合回收 (Mixed GC)

(如果需要，单线程、独占式、高强度的 FullGC 还是继续存在的。它针对 GC 的评估失败提供了一种失败保护机制，即强力回收。)



顺时针， young gc->young gc+concurrent mark->Mixed GC 顺序，进行垃圾回收。

应用程序分配内存，当年轻代的 Eden 区用尽时开始年轻代回收过程； G1 的年轻代收集阶段是一个并行的独占式收集器。在年轻代回收期， G1GC 暂停所有应用程序线程，启动多线程执行年轻代回收。然后从年轻代区间移动存活对象到 Survivor 区间或者老年区间，也有可能是两个区间都会涉及。

当堆内存使用达到一定值（默认 45%）时，开始老年并发标记过程。

标记完成马上开始混合回收过程。对于一个混合回收期， G1GC 从老年区间移动存活对象到空闲区间，这些空闲区间也就成为了老年的一部分。和年轻代不同，老年 G1 回收器和其他 GC 不同， G1 的老年回收器不需要整个老年被回收，一次只需要扫描/回收一小部分老年 Region 就可以了。同时，这个老年 Region 是和年轻代一起被回收的。

举个例子：一个 Web 服务器， Java 进程最大堆内存为 4G，每分钟响应 1500 个请求，每 45 秒钟会新分配大约 2G 的内存。 G1 会每 45 秒钟进行一次年轻代回收，每 31 个小时整个堆的使用率会达到 45%，会开始老年并发标记过程，标记完成后开始四到五次的混合回收。

Remembered Set (记忆集)

一个对象被不同区域引用的问题

一个 Region 不可能是孤立的，一个 Region 中的对象可能被其他任意 Region 中对象引用，判断对象存活时，是否需要扫描整个 Java 堆才能保证准确？

在其他的分代收集器，也存在这样的问题（而 G1 更突出）

回收新生代也不得不同时扫描老年代？

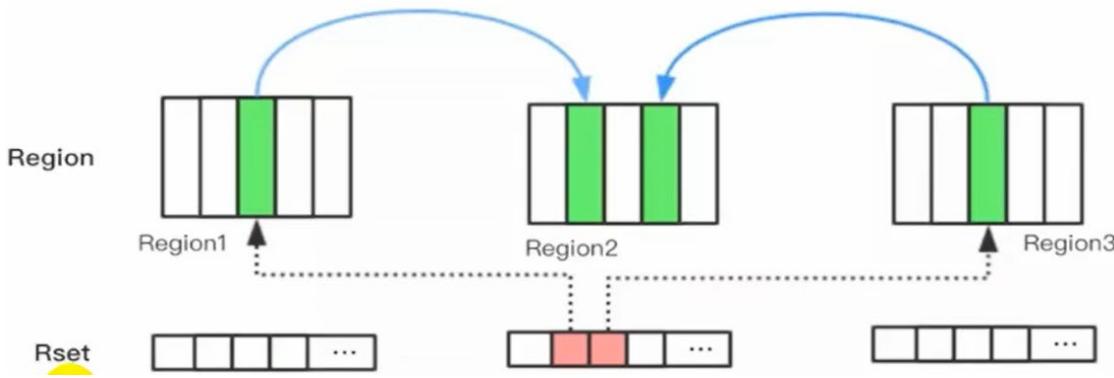
这样的话会降低 MinorGC 的效率；

解决方法：

无论 G1 还是其他分代收集器， JVM 都是使用 Remembered Set 来避免全局扫描：

每个 Region 都有一个对应的 Remembered Set；每次 Reference 类型数据写操作时，都会产生一个 Write Barrier 暂时中断操作；

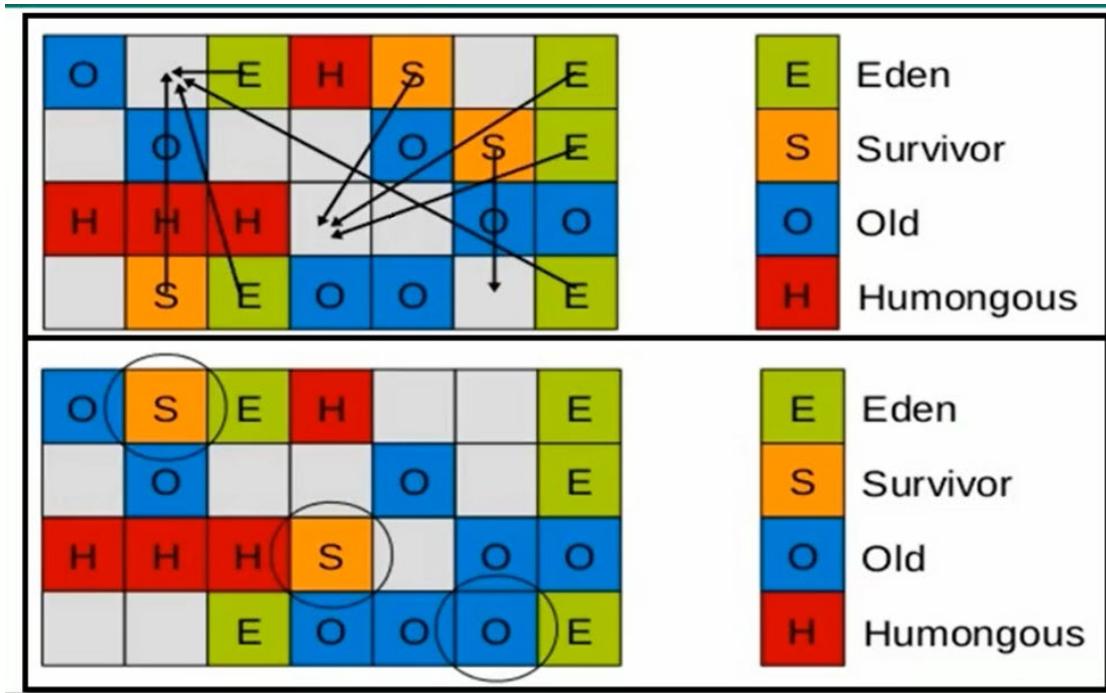
然后检查将要写入的引用指向的对象是否和该 Reference 类型数据在不同的 Region (其他收集器：检查老年对象是否引用了新生代对象)；如果不同，通过 cardTable 把相关引用信息记录到引用对象的所在 Region 对应的 Remembered Set 中；当进行垃圾收集时，在 GC 根节点的枚举范围加入 Remembered Set；就可以保证不进行全局扫描，也不会有遗漏。



G1 回收过程-年轻代 GC

JVM 启动时，G1 先准备好 Eden 区，程序在运行过程中不断创建对象到 Eden 区，当 Eden 空间耗尽时，G1 会启动一次年轻代垃圾回收过程。

YGC 时，首先 G1 停止应用程序的执行（Stop-The-World），G1 创建回收集（Collection Set），回收集是指需要被回收的内存分段的集合，年轻代回收过程的回收集包含年轻代 Eden 区和 Survivor 区所有的内存分段。



然后开始如下回收过程：

- 第一阶段，扫描根

根是指 static 变量指向的对象，正在执行的方法调用链条上的局部变量等。根引用连同 RSet 记录的外部引用作为扫描存活对象的入口。

- 第二阶段，更新 RSet

处理 dirty card queue（见备注）中的 card，更新 RSet。此阶段完成后，RSet 可以准确的反映老年代对所在的内存分段中对象的引用。

- 第三阶段，处理 RSet

识别被老年代对象指向的 Eden 中的对象，这些被指向的 Eden 中的对象被认为是存活的对象。

- 第四阶段，复制对象。

此阶段，对象树被遍历，Eden 区内存段中存活的对象会被复制到 Survivor 区中空的内存分段，Survivor 区内存段中存活的对象如果年龄未达阈值，年龄会加 1，达到阈值会被复制到 old 区中空的内存分段。如果 Survivor 空间不够，Eden 空间的部分数据会直接晋升到老年代空间。

- 第五阶段，处理引用

处理 Soft, Weak, Phantom, Final, JNI Weak 等引用。最终 Eden 空间的数据为空，GC 停止工作，而目标内存中的对象都是连续存储的，没有碎片，所以复制过程可以达到内存整理的效果，减少碎片。

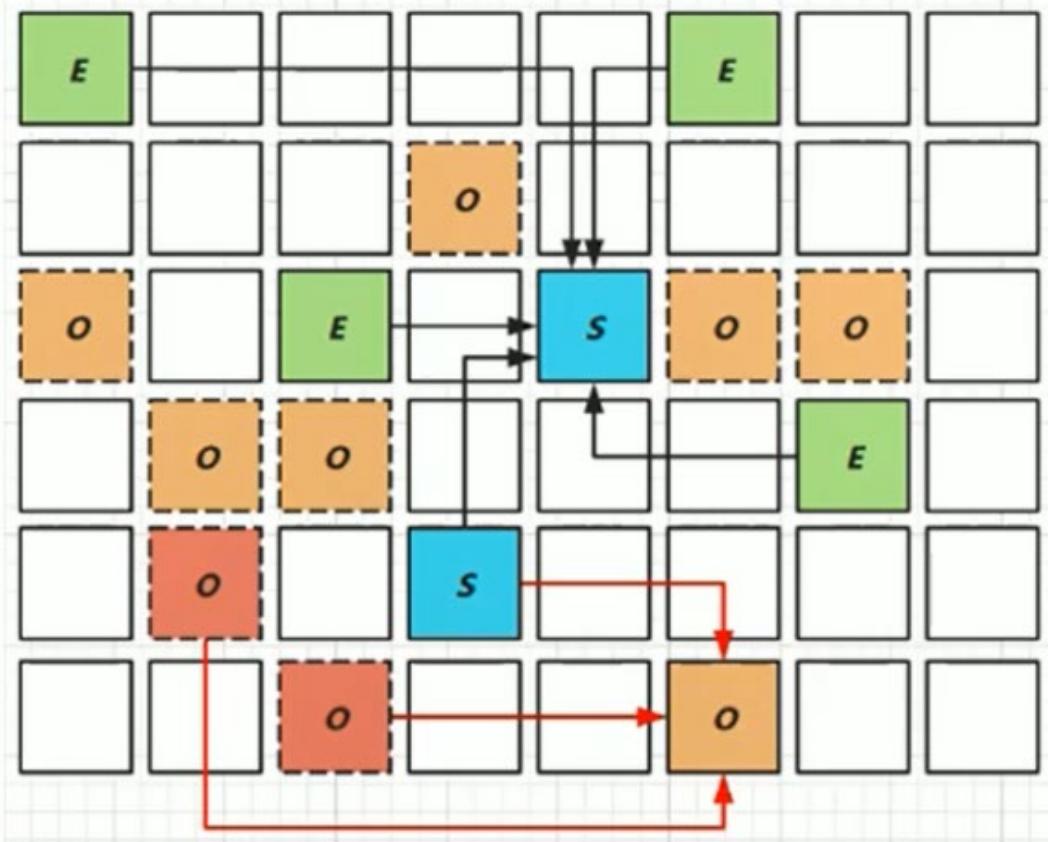
G1 回收过程-并发标记过程

- 初始标记阶段：标记从根节点直接可达的对象。这个阶段是 STW 的，并且会触发一次年轻代 GC。
- 根区域扫描（Root Region Scanning）：G1 GC 扫描 survivor 区直接可达的老年代区域对象，并标记被引用的对象。这一过程必须在 youngGC 之前完成。
- 并发标记（Concurrent Marking）：在整个堆中进行并发标记（和应用程序并发执行），此过程可能被 youngGC 中断。在并发标记阶段，若发现区域对象中的所有对象都是垃圾，那这个区域会被立即回收。同时，并发标记过程中，会计算每个区域的对象活性（区域中存活对象的比例）。
- 再次标记（Remark）：由于应用程序持续进行，需要修正上一次的标记结果。是 STW 的。G1 中采用了比 CMS 更快的初始快照算法：snapshot-at-the-beginning（SATB）。
- 独占清理（cleanup, STW）：计算各个区域的存活对象和 GC 回收比例，并进行排序，识别可以混合回收的区域。为下阶段做铺垫。是 STW 的。这个阶段并不会实际上去做垃圾的收集。

- 并发清理阶段：识别并清理完全空闲的区域。

G1 回收过程 - 混合回收

当越来越多的对象晋升到老年代 old region 时，为了避免堆内存被耗尽，虚拟机会触发一个混合的垃圾收集器，即 Mixed GC，该算法并不是一个 old GC，除了回收整个 Young Region，还会回收一部分的 old Region。这里需要注意：**是一部分老年代，而不是全部老年代**。可以选择哪些 old Region 进行收集，从而可以对垃圾回收的耗时时间进行控制。也要注意的是 Mixed GC 并不是 Full GC。



并发标记结束以后，老年代中百分百为垃圾的内存分段被回收了，部分为垃圾的内存分段被计算了出来。默认情况下，这些老年代的内存分段会分 8 次（可以通过-XX:G1MixedGCCountTarget 设置）被回收

混合回收的回收集（Collection Set）包括八分之一的老年代内存分段，Eden 区内存分段，Survivor 区内存分段。混合回收的算法和年轻代回收的算法完全一样，只是回收集多了老年代的内存分段。具体过程请参考上面的年轻代回收过程。

由于老年代中的内存分段默认分 8 次回收，G1 会优先回收垃圾多的内存分段。垃圾占内存分段比例越高的，越会被先回收。并且有一个阈值会决定内存分段是否被回收，

`XX:G1MixedGCLiveThresholdPercent`, 默认为 65%，意思是垃圾占内存分段比例要达到 65% 才会被回收。如果垃圾占比太低，意味着存活的对象占比高，在复制的时候会花费更多的时间。

混合回收并不一定要进行 8 次。有一个阈值`-XX:G1HeapWastePercent`, 默认值为 1e%，意思是允许整个堆内存中有 10% 的空间被浪费，意味着如果发现可以回收的垃圾占堆内存的比例低于 10%，则不再进行混合回收。因为 GC 会花费很多的时间但是回收到的内存却很少。

G1 回收可选的过程 4 - Full GC

G1 的初衷就是要避免 FullGC 的出现。但是如果上述方式不能正常工作，G1 会停止应用程序的执行（stop-The-world），使用单线程的内存回收算法进行垃圾回收，性能会非常差，应用程序停顿时间会很长。

要避免 FullGC 的发生，一旦发生需要进行调整。什么时候会发生 FullGC 呢？比如堆内存太小，当 G1 在复制存活对象的时候没有空的内存分段可用，则会回退到 full GC，这种情况可以通过增大内存解决。导致 full FullGC 的原因可能有两个：

- EVacuation (回收) 的时候没有足够的 to-space 来存放晋升的对象；
- 并发处理过程完成之前空间耗尽。

G1 回收的优化建议

从 oracle 官方透露出来的信息可知，回收阶段（Evacuation）其实本也有想过设计成与用户程序一起并发执行，但这件事情做起来比较复杂，考虑到 G1 只是回一部分 Region，停顿时间是用户可控制的，所以并不迫切去实现，而选择把这个特性放到了 G1 之后出现的低延迟垃圾收集器（即 ZGC）中。另外，还考虑到 G1 不是仅仅面向低延迟，停顿用户线程能够最大幅度提高垃圾收集效率，为了保证吞吐量所以才选择了完全暂停用户线程的实现方案。

优化建议一：年轻代大小

- 避免使用`-Xmn` 或 `-XX:NewRatio` 等相关选项显式设置年轻代大小
- 固定年轻代的大小会覆盖暂停时间目标

优化建议二：暂停时间目标不要太过严苛

- G1 GC 的吞吐量目标是 90% 的应用程序时间和 10% 的垃圾回收时间

- 评估 G1GC 的吞吐量时，暂停时间目标不要太严苛。目标太过严苛表示你愿意承受更多的垃圾回收开销，而这些会直接影响到吞吐量。

垃圾回收器总结

截止 JDK1.8，一共有 7 款不同的垃圾收集器。每一款的垃圾收集器都有不同的特点，在具体使用的时候，需要根据具体的情况选用不同的垃圾收集器。

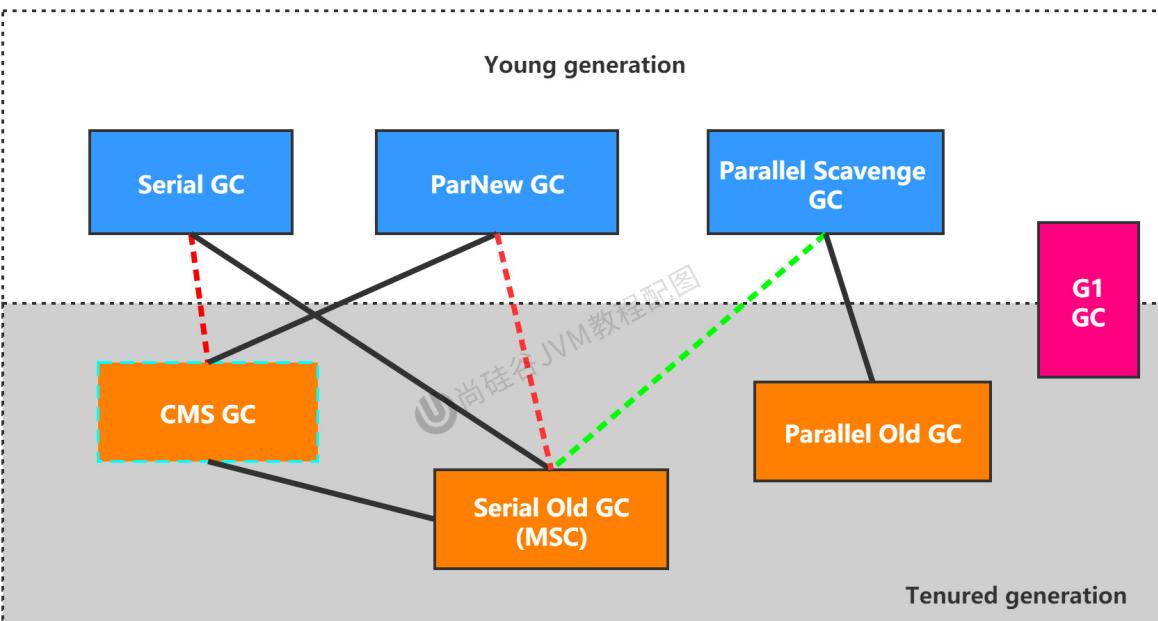
新生代都是复制算法 老年代都是标记整理，只有CMS是标记清除

垃圾收集器	分类	作用位置	使用算法	特点	适用场景
Serial	串行运行	作用于新生代	复制算法	响应速度优先	适用于单CPU环境下的client模式
ParNew	并行运行	作用于新生代	复制算法	响应速度优先	多CPU环境Server模式下与CMS配合使用
Parallel	并行运行	作用于新生代	复制算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
Serial Old	串行运行	作用于老年代	标记-压缩算法	响应速度优先	适用于单CPU环境下的Client模式
Parallel Old	并行运行	作用于老年代	标记-压缩算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
CMS	并发运行	作用于老年代	标记-清除算法	响应速度优先	适用于互联网或B/S业务
G1	并发、并行运行	作用于新生代、老年代	标记-压缩算法、复制算法	响应速度优先	面向服务端应用

G1同时针对老年代和新生代进行垃圾回收，所以有两个

GC 发展阶段： Serial=> Parallel（并行） => CMS（并发） => G1 => ZGC

不同厂商、不同版本的虚拟机实现差距比较大。HotSpot 虚拟机在 JDK7/8 后所有收集器及组合如下图



怎么选择垃圾回收器

Java 垃圾收集器的配置对于 JVM 优化来说是一个很重要的选择，选择合适的垃圾收集器可以让 JVM 的性能有一个很大的提升。怎么选择垃圾收集器？

- 优先调整堆的大小让 JVM 自适应完成。
- 如果内存小于 100M，使用串行收集器
- 如果是单核、单机程序，并且没有停顿时间的要求，串行收集器
- 如果是多 CPU、需要高吞吐量、允许停顿时间超过 1 秒，选择并行或者 JVM 自己选择
- 如果是多 CPU、追求低停顿时间，需快速响应（比如延迟不能超过 1 秒，如互联网应用），使用并发收集器
- 官方推荐 G1，性能高。
- 现在互联网的项目，基本都是使用 G1。

最后需要明确一个观点：

- 没有最好的收集器，更没有万能的收集
- 调优永远是针对特定场景、特定需求，不存在一劳永逸的收集器

面试

对于垃圾收集，面试官可以循序渐进从理论、实践各种角度深入，也未必是要求面试者什么都懂。但如果你懂得原理，一定会成为面试中的加分项。这里较通用、基础性的部分如下：

垃圾收集的算法有哪些？如何判断一个对象是否可以回收？

垃圾收集器工作的基本流程。

另外，大家需要多关注垃圾回收器这一章的各种常用的参数

GC 日志分析

通过阅读 GC 日志，我们可以了解 Java 虚拟机内存分配与回收策略。内存分配与垃圾回收的参数列表

- -XX:+PrintGc 输出 GC 日志。类似：-verbose:gc
- -XX:+PrintGcDetails 输出 Gc 的详细日志
- -XX:+PrintGcTimestamps 输出 Gc 的时间戳（以基准时间的形式）
- -XX:+PrintGCDetails 输出 Gc 的时间戳（以日期的形式，如 2013-05-04T21: 53: 59.234+0800）
- -XX:+PrintHeapAtGC 在进行 Gc 的前后打印出堆的信息

- -Xloggc:.../logs/gc.log 日志文件的输出路径

verbose:gc

打开 GC 日志

-verbose:gc

这个只会显示总的 GC 堆的变化，如下：

```
[GC (Allocation Failure) 80832K->19298K(227840K), 0.0084018 secs]
[GC (Metadata GC Threshold) 109499K->21465K(228352K), 0.0184066 secs]
[Full GC (Metadata GC Threshold) 21465K->16716K(201728K), 0.0619261 secs]
```

参数解析

GC、Full GC: GC的类型， GC只在新生代上进行， Full GC包括永生代， 新生代， 老年代。

Allocation Failure: GC发生的原因。

80832K->19298K: 堆在GC前的大小和GC后的大小。

228840k: 现在的堆大小。

0.0084018 secs: GC持续的时间。

PrintGCDetails

打开 GC 日志

-verbose:gc -XX:+PrintGCDetails

输入信息如下

```
[GC (Allocation Failure) [PSYoungGen: 70640K->10116K(141312K)] 80541K->20017K(227328K), 0.0172573
secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
[GC (Metadata GC Threshold) [PSYoungGen: 98859K->8154K(142336K)] 108760K->21261K(228352K),
0.0151573 secs] [Times: user=0.00 sys=0.01, real=0.02 secs]
[Full GC (Metadata GC Threshold) [PSYoungGen: 8154K->0K(142336K)] [ParOldGen: 13107K-
>16809K(62464K)] 21261K->16809K(204800K), [Metaspace: 20599K->20599K(1067008K)], 0.0639732 secs]
[Times: user=0.14 sys=0.00, real=0.06 secs]
```

参数解析

```
GC, Full GC: 同样是GC的类型
Allocation Failure: GC原因
PSYoungGen: 使用了Parallel Scavenge并行垃圾收集器的新生代GC前后大小的变化
ParOldGen: 使用了Parallel Old并行垃圾收集器的老年代GC前后大小的变化
Metaspace: 元数据区GC前后大小的变化, JDK1.8中引入了元数据区以替代永久代
xxx secs: 指GC花费的时间
Times: user: 指的是垃圾收集器花费的所有CPU时间, sys: 花费在等待系统调用或系统事件的时间, real:
GC从开始到结束的时间, 包括其他进程占用时间片的实际时间。
```

补充

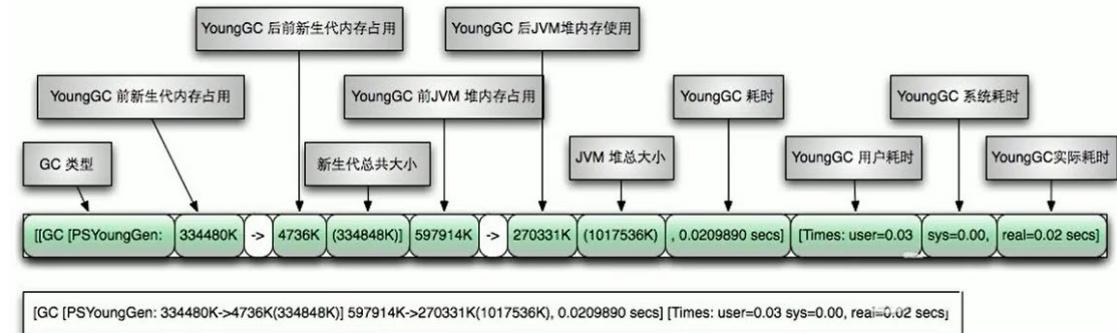
- [GC]和[FullGC]说明了这次垃圾收集的停顿类型, 如果有"Full"则说明 GC发生了"stop The World"
- 使用 Serial 收集器在新生代的名字是 Default New Generation, 因此显示的是 "[DefNew]"
- 使用 ParNew 收集器在新生代的名字会变成"[ParNew]", 意思是"Parallel New Generation"
- 使用 Parallel scavenge 收集器在新生代的名字是"[PSYoungGen]"
- 老年代的收集和新生代道理一样, 名字也是收集器决定的
- 使用 G1 收集器的话, 会显示为"garbage-first heap"

Allocation Failure 表明本次引起 GC 的原因是因为在年轻代中没有足够的空间能够存储新的数据了。

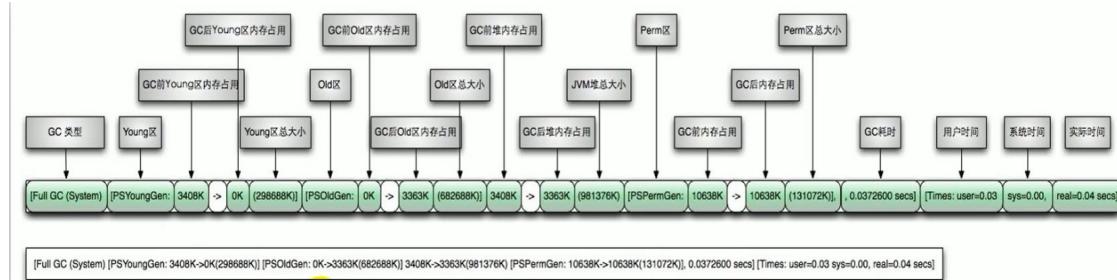
[PSYoungGen: 5986K->696K (8704K)]5986K->704K (9216K) 中括号内: GC
回收前年轻代大小, 回收后大小, (年轻代总大小) 括号外: GC 回收前年轻代和
老年代大小, 回收后大小, (年轻代和老年代总大小)

user 代表用户态回收耗时, sys 内核态回收耗时, real 实际耗时。由于多核的原因,
时间总和可能会超过 real 时间

Young GC 图片



FullGC 图片



GC 回收举例

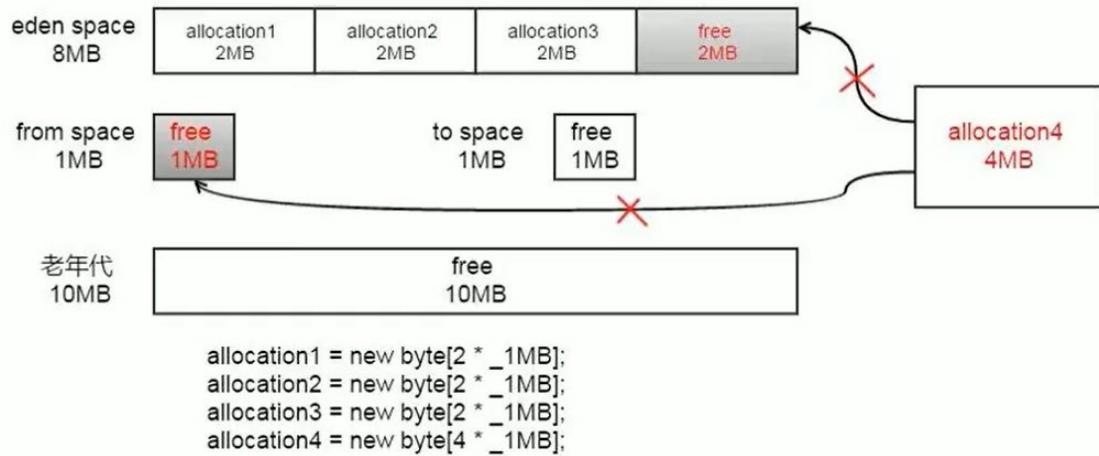
我们编写一个程序，用来说明 GC 收集的过程

```
/*
 * GC 垃圾收集过程
 * @author: 陌溪
 * @create: 2020-07-14-8:35
 */
public class GCUseTest {
    static final Integer _1MB = 1024 * 1024;
    public static void main(String[] args) {
        byte [] allocation1, allocation2, allocation3, allocation4;
        allocation1 = new byte[2 *_1MB];
        allocation2 = new byte[2 *_1MB];
        allocation3 = new byte[2 *_1MB];
        allocation4 = new byte[4 *_1MB];
    }
}
```

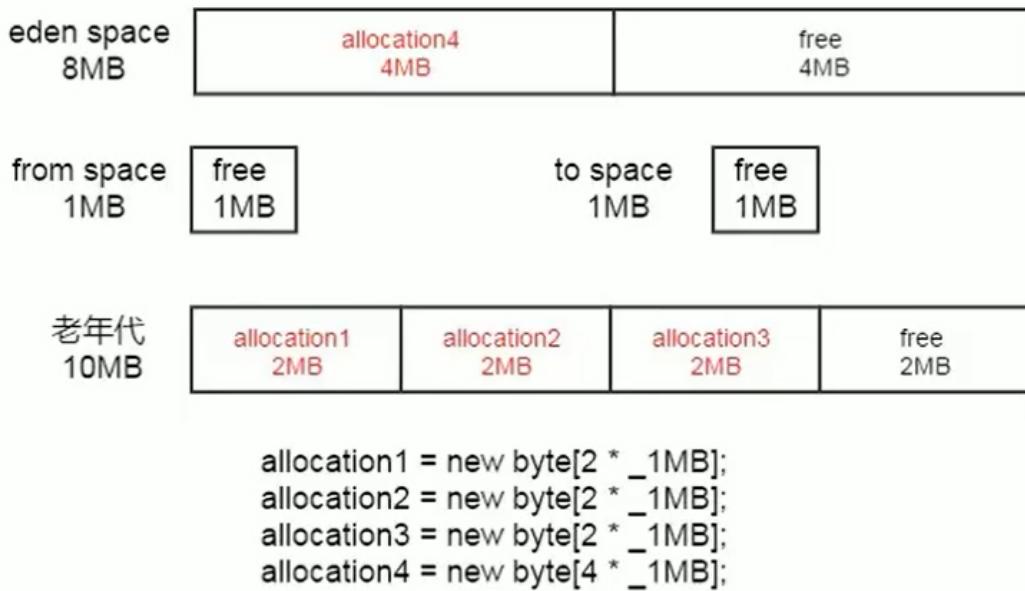
我们设置 JVM 启动参数

```
-Xms10m -Xmx10m -XX:+PrintGCDetails
```

首先我们会将 3 个 2M 的数组存放到 Eden 区，然后后面 4M 的数组来了后，将无法存储，因为 Eden 区只剩下 2M 的剩余空间了，那么将会进行一次 Young GC 操作，将原来 Eden 区的内容，存放到 Survivor 区，但是 Survivor 区也存放不下，那么就会直接晋级存入 Old 区



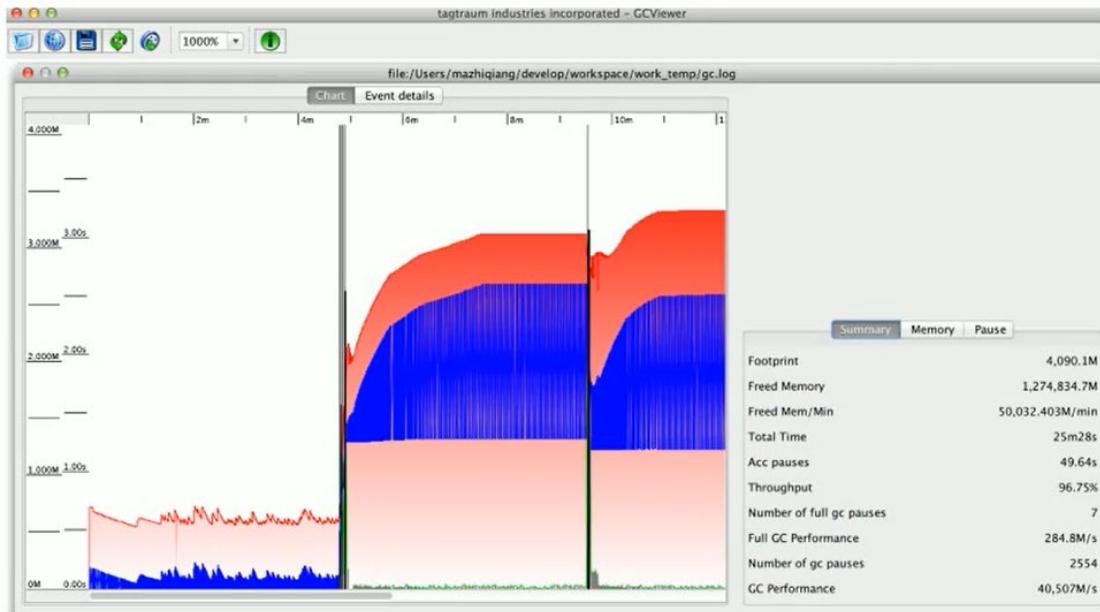
然后我们将 4M 对象存入到 Eden 区中



可以用一些工具去分析这些 GC 日志

常用的日志分析工具有：GCViewer、GCEasy、GCHisto、GCLogViewer、Hpjmeter、garbagecat 等

GCViewer



GC easy

The screenshot shows the homepage of the GCeasy website. The header includes links for HOME, BLOG, FEATURES, PRICING, USER REVIEWS, FAQ, and SIGN IN. The main content features a large banner with the text "Universal GC Log Analyzer" and a yellow circular button. Below the banner, a list of features is provided:

- ✓ Solve Memory & GC problems in seconds
- ✓ Get JVM Heap settings recommendations
- ✓ Machine Learning Algorithms
- ✓ Trusted by 4,000+ enterprises
- ✓ Free

To the right, there is a form for uploading a GC log file, with options for "Upload" and "Raw". A tip for compressing logs is displayed, along with a file selection input field and an "Analyze" button.

垃圾回收器的新发展

GC 仍然处于飞速发展之中，目前的默认选项 G1GC 在不断的进行改进，很多我们原来认为的缺点，例如串行的 FullGC、Card Table 扫描的低效等，都已经被大幅改进，例如，JDK10 以后，FullGC 已经是并行运行，在很多场景下，其表现还略优于 ParallelGC 的并行 FullGC 实现。

即使是 SerialGC，虽然比较古老，但是简单的设计和实现未必就是过时的，它本身的开销，不管是 GC 相关数据结构的开销，还是线程的开销，都是非常小的，所以随着云计算的兴起，在 serverless 等新的应用场景下，Serial Gc 找到了新的舞台。

比较不幸的是 CMSGC，因为其算法的理论缺陷等原因，虽然现在还有非常大的用户群体，但在 JDK9 中已经被标记为废弃，并在 JDK14 版本中移除

Epsilon:A No-Op GarbageCollector (Epsilon 垃圾回收器，"No-Op (无操作)"回收器) <http://openjdk.java.net/iep/s/318>

ZGC:A Scalable Low-Latency Garbage Collector (Experimental) (ZGC: 可伸缩的低延迟垃圾回收器，处于实验性阶段)

现在 G1 回收器已成为默认回收器好几年了。我们还看到了引入了两个新的收集器：ZGC (JDK11 出现) 和 Shenandoah (Open JDK12)

主打特点：低停顿时间

Open JDK12 的 Shenandoah GC

Open JDK12 的 shenandoash GC：低停顿时间的 GC (实验性)

Shenandoah，无疑是众多 GC 中最孤独的一个。是第一款不由 oracle 公司团队领导开发的 Hotspot 垃圾收集器。不可避免的受到官方的排挤。比如号称 openJDK 和 OracleJDk 没有区别的 Oracle 公司仍拒绝在 oracleJDK12 中支持 Shenandoah。

Shenandoah 垃圾回收器最初由 RedHat 进行的一项垃圾收集器研究项目 Pauseless GC 的实现，旨在针对 JVM 上的内存回收实现低停顿的需求。在 2014 年贡献给 OpenJDK。

Red Hat 研发 Shenandoah 团队对外宣称，Shenandoah 垃圾回收器的暂停时间与堆大小无关，这意味着无论将堆设置为 200MB 还是 200GB，99.9% 的目标都可以把垃圾收集的停顿时间限制在十毫秒以内。不过实际使用性能将取决于实际工作堆的大小和工作负载。

Shenandoah开发团队在实际应用中的测试数据

收 集 器	运 行 时 间	总 停 转	最 大 停 转	平 均 停 转
Shenandoah	387.602s	320ms	89.79ms	53.01ms
G1	312.052s	11.7s	1.24s	450.12ms
CMS	285.264s	12.78s	4.39s	852.26ms
Parallel Scavenge	260.092s	6.59s	3.04s	823.75ms

这是 RedHat 在 2016 年发表的论文数据，测试内容是使用 Es 对 200GB 的维基百科数据进行索引。从结果看：

停顿时间比其他几款收集器确实有了质的飞跃，但也未实现最大停顿时间控制在十毫秒以内的目标。而吞吐量方面出现了明显的下降，总运行时间是所有测试收集器里最长的。

总结

- shenandoah Gc 的弱项：高运行负担下的吞吐量下降。
- shenandoah GC 的强项：低延迟时间。

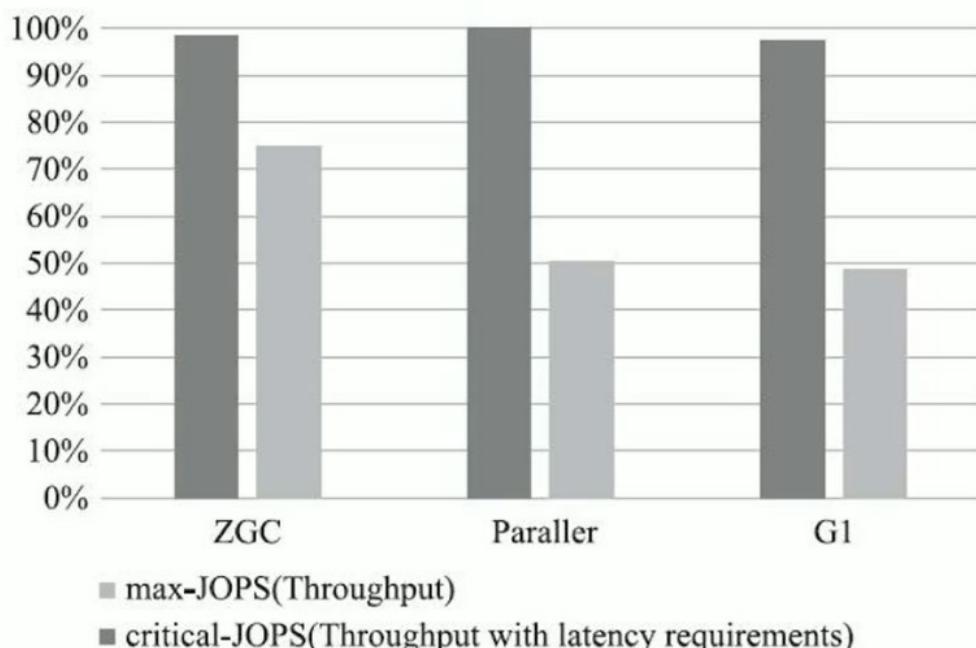
革命性的 ZGC

ZGC 与 shenandoah 目标高度相似，在尽可能对吞吐量影响不大的前提下，实现在任意堆内存大小下都可以把垃圾收集的停顿时间限制在十毫秒以内的低延迟。

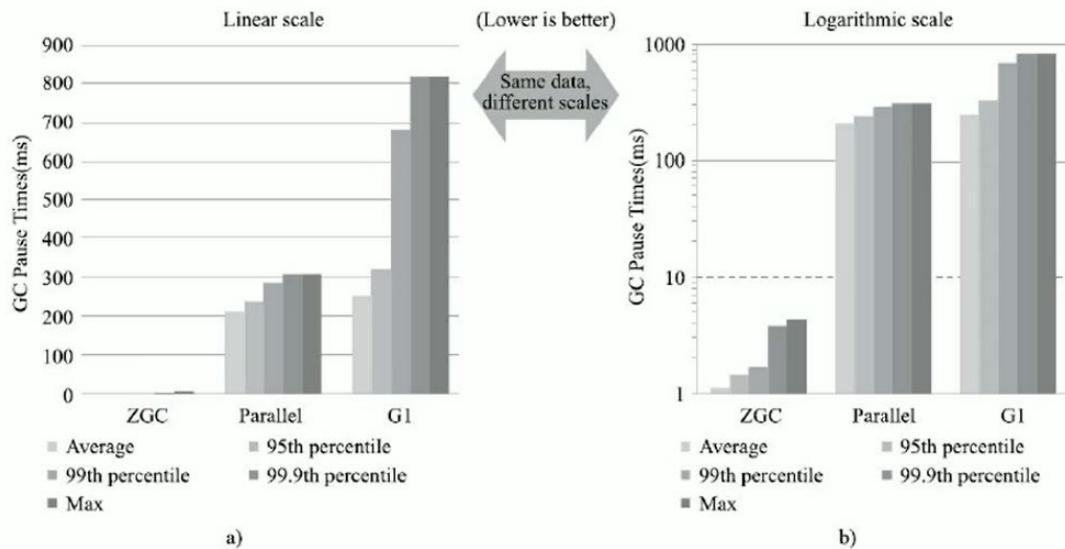
《深入理解 Java 虚拟机》一书中这样定义 ZGC：ZGC 收集器是一款基于 Region 内存布局的，（暂时）不设分代的，使用了读屏障、染色指针和内存多重映射等技术来实现可并发的标记-压缩算法的，以低延迟为首要目标的一款垃圾收集器。

ZGC 的工作过程可以分为 4 个阶段：并发标记 - 并发预备重分配 - 并发重分配 - 并发重映射 等。

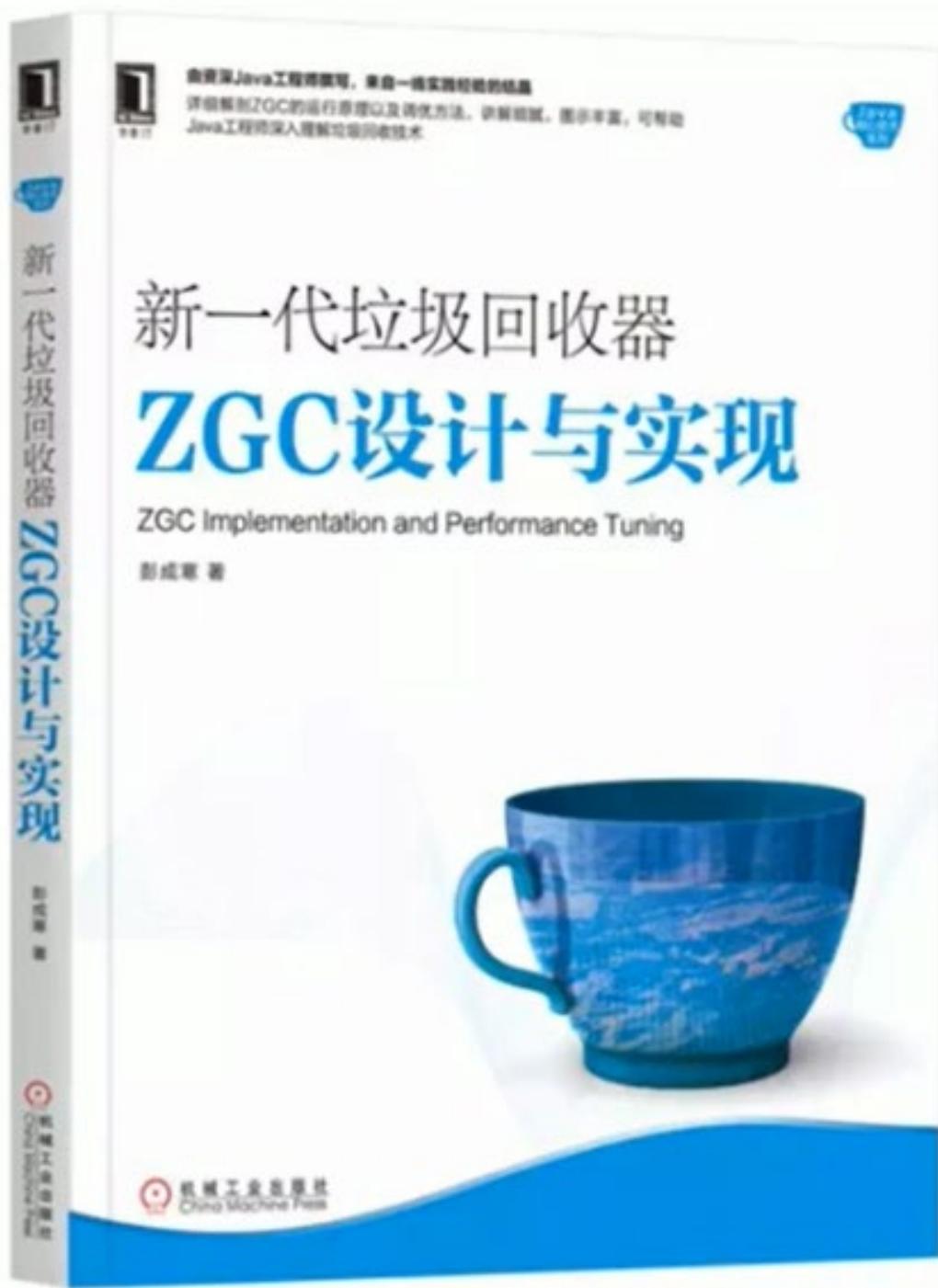
ZGC 几乎在所有地方并发执行，除了初始标记的是 STW 的。所以停顿时间几乎就耗费在初始标记上，这部分的实际时间是非常少的。



停顿时间对比



虽然 ZGC 还在试验状态，没有完成所有特性，但此时性能已经相当亮眼，用“令人震惊、革命性”来形容，不为过。未来将在服务端、大内存、低延迟应用的首选垃圾收集器。



JDK14 之前，ZGC 仅 Linux 才支持。

尽管许多使用 ZGC 的用户都使用类 Linux 的环境，但在 Windows 和 macOS 上，人们也需要 ZGC 进行开发部署和测试。许多桌面应用也可以从 ZGC 中受益。因此，ZGC 特性被移植到了 Windows 和 macOS 上。

现在 mac 或 Windows 上也能使用 zGC 了，示例如下：

-XX:+UnlockExperimentalVMOptions-XX: +UseZGC

AliGC

AliGC 是阿里巴巴 JVM 团队基于 G1 算法，面向大堆（LargeHeap）应用场景。指定场景下的对比：



当然，其它厂商也提供了各种别具一格的 GC 实现，例如比较有名的低延迟 GC Zing

