一.jdk内置命令行工具

1、开发命令行

工具	简介
java	Java 应用的启动程序
javac	JDK 内置的编译工具
javap	反编译 class 文件的工具
javadoc	根据 Java 代码和标准注释,自动生成相关的API说明文档
javah	JNI 开发时, 根据 java 代码生成需要的 .h文件。
extcheck	检查某个 jar 文件和运行时扩展 jar 有没有版本冲突，很少使用
jdb	Java Debugger；可以调试本地和远端程序, 属于 JPDA 中的一个 demo 实现, 供其他调试器参考。开发时很少使用
jdeps	探测 class 或 jar 包需要的依赖
jar	打包工具，可以将文件和目录打包成为 .jar 文件；.jar 文件本质上就是 zip 文件, 只是后缀不同。使用时按顺序对应好选项和参数即可。
keytool	安全证书和密钥的管理工具；（支持生成、导入、导出等操作）
jarsigner	JAR 文件签名和验证工具
policytool	实际上这是一款图形界面工具, 管理本机的 Java 安全策略

2、JVM分析调优命令行

工具	简介
jps/jinfo	查看 java 进程
jstat	查看 JVM 内部 gc 相关信息
jmap	查看 heap 或类占用空间统计
jstack	查看线程信息
jcmd	执行 JVM 相关分析命令（整合命令）
jrunscript/jjs	执行 js 命令

jps：示例：jps -l：主要用来获取进程PID，带包名

jstat：示例：jstat -gc pid 1000 10：查看某进程的gc情况，每1000ms打印依次，总共打印10次

jstat -gcutil pid 1000 10：按百分比显示上述

jmap：示例：jmap -heap pid：打印堆内存配置和使用信息

jmap -histo pid：直方图，看哪些类占用空间最多

jmap -histo pid > xxx.txt ： 输出信息到文件

jstack：示例：jstack -l pid：查看某进程的线程信息、锁的信息等

jcmd：整合上述命令

示例：jcmd pid help：展示进程的有授权的命令，例如GC.heap_info

jcmd pid GC.heap_info

3、JVM图形化工具：jdk自带：jconsole

jvisualvm

idea插件：visualGC

JMC：Java Mission Control

二、GC策略

1、回收算法

分代算法：年轻代内存满了触发Minor GC：使用标记复制算法+S0+S1

老年代内存满了触发Full GC：使用标记清除整理算法

	标记整理算法	复制算法
空间	内存空间利用率高	内存空间利用率低
时间	内存回收效率低	内存回收效率高
优势场景	单次回收对象少(在单次不回收对象时达到极致), 内存空间紧张	单次回收对象多(在单次回收全部对象时达到极致), 内存空间宽裕

2、多种GC策略

串行GC (Serial GC)：单线程，会触发全线暂停STW，适用于几百兆内存的单核服务器

改进后的多线程版本 ParNew GC，配后CMS(并发标记清除)使用

并行GC (Parallel GC)：适用于多核服务器

串行：按顺序一件一件做事

并发：一段时间内你交替做多件事

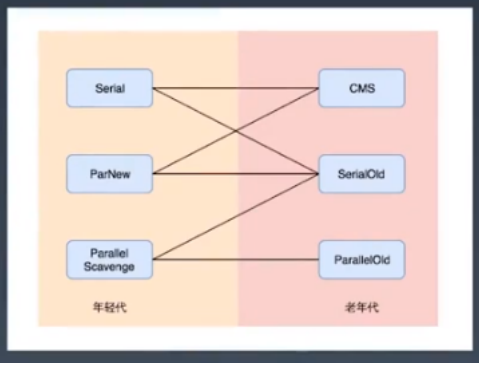
并行：两个人同时做事

G1 GC：Garbage-First，垃圾优先，CMS GC的升级版

各个 GC 对比

收集器	串行、并行or并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后备预案
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境时在Server模式下与CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网站或B/S系统服务端上的Java应用
G1	并发	both	标记-整理+复制算法	响应速度优先	面向服务端应用，将来替换CMS

常用的 GC 组合（重点）



常用的组合为：

（1）Serial+Serial Old 实现单线程的低延迟垃圾回收机制；

（2）ParNew+CMS，实现多线程的低延迟垃圾回收机制；

（3）Parallel Scavenge和Parallel Scavenge Old，实现多线程的高吞吐量垃圾回收机制；

GC 如何选择

选择正确的 GC 算法，唯一可行的方式就是去尝试，一般性的指导原则：

1. 如果系统考虑吞吐优先，CPU 资源都用来最大程度处理业务，用 Parallel GC；
2. 如果系统考虑低延迟有限，每次 GC 时间尽量短，用 CMS GC；
3. 如果系统内存堆较大，同时希望整体来看平均 GC 时间可控，使用 G1 GC。

对于内存大小的考量：

1. 一般 4G 以上，算是比较大，用 G1 的性价比较高。
2. 一般超过 8G，比如 16G-64G 内存，非常推荐使用 G1 GC。

最后讨论一个很多开发者经常忽视的问题，也是面试大厂常问的问题：JDK8 的默认 GC 是什么？JDK9，JDK10，JDK11...等等默认的是 GC 是什么？

GC 总结

可以看出 GC 算法和实现的演进路线：

1. 串行 -> 并行：重复利用多核 CPU 的优势，大幅降低 GC 暂停时间，提升吞吐量。
2. 并行 -> 并发：不只开多个 GC 线程并行回收，还将GC操作拆分为多个步骤，让很多繁重的任务和应用线程一起并发执行，减少了单次 GC 暂停持续的时间，这能有效降低业务系统的延迟。
3. CMS -> G1：G1 可以说是在 CMS 基础上进行迭代和优化开发出来的，划分为多个小堆块进行增量回收，这样就更进一步地降低了单次 GC 暂停的时间
4. G1 -> ZGC：ZGC 号称无停顿垃圾收集器，这又是一次极大的改进。ZGC 和 G1 有一些相似的地方，但是底层的算法和思想又有了全新的突破。

脱离场景谈性能都是耍流氓”。

目前绝大部分 Java 应用系统，堆内存并不大比如 2G-4G 以内，而且对 10ms 这种低延迟的 GC 暂停不敏感，也就是说处理一个业务步骤，大概几百毫秒都是可以接受的，GC 暂停 100ms 还是 10ms 没多大区别。另一方面，系统的吞吐量反而往往是我们追求的重点，这时候就需要考虑采用并行 GC。

如果堆内存再大一些，可以考虑 G1 GC。如果内存非常大（比如超过 16G，甚至是 64G、128G），或者是对延迟非常敏感（比如高频量化交易系统），就需要考虑使用本节提到的新 GC（ZGC/Shenandoah）。