第三版：JVM 32 道

运行时数据区是什么?

虚拟机在执行 Java 程序的过程中会把它所管理的内存划分为若干不同的数据区，这些区域有各自的用途、创建和销毁时间。

线程私有：程序计数器、Java 虚拟机栈、本地方法栈。

线程共享：Java 堆、方法区。

程序计数器是什么?

程序计数器是一块较小的内存空间，可以看作当前线程所执行字节码的行号指示器。字节码解释器工作时通过改变计数器的值选取下一条执行指令。分支、循环、跳转、线程恢复等功能都需要依赖计数器完成。是唯一在虚拟机规范中没有规定内存溢出情况的区域。

如果线程正在执行 Java 方法，计数器记录正在执行的虚拟机字节码指令地址。如果是本地方法，计数器值为 Undefined。

Java 虚拟机栈的作用?

Java 虚拟机栈来描述 Java 方法的内存模型。每当有新线程创建时就会分配一个栈空间，线程结束后栈空间被回收，栈与线程拥有相同的生命周期。栈中元素用于支持虚拟机进行方法调用，每个方法在执行时都会创建一个栈帧存储方法的局

部变量表、操作栈、动态链接和方法出口等信息。每个方法从调用到执行完成，就是栈帧从入栈到出栈的过程。

有两类异常：① 线程请求的栈深度大于虚拟机允许的深度抛出 StackOverflowError。② 如果 JVM 栈容量可以动态扩展，栈扩展无法申请足够内存抛出 OutOfMemoryError（HotSpot 不可动态扩展，不存在此问题）。

本地方法栈的作用?

本地方法栈与虚拟机栈作用相似，不同的是虚拟机栈为虚拟机执行 Java 方法服务，本地方法栈为虚本地方法服务。调用本地方法时虚拟机栈保持不变，动态链接并直接调用指定本地方法。

虚拟机规范对本地方法栈中方法的语言与数据结构无强制规定，虚拟机可自由实现，例如 HotSpot 将虚拟机栈和本地方法栈合二为一。

本地方法栈在栈深度异常和栈扩展失败时分别抛出 StackOverflowError 和 OutOfMemoryError。

堆的作用是什么?

堆是虚拟机所管理的内存中最大的一块，被所有线程共享的，在虚拟机启动时创建。堆用来存放对象实例，Java 里几乎所有对象实例都在堆分配内存。堆可以处于物理上不连续的内存空间，逻辑上应该连续，但对于例如数组这样的大对象，多数虚拟机实现出于简单、存储高效的考虑会要求连续的内存空间。

堆既可以被实现成固定大小，也可以是可扩展的，可通过 -Xms 和 -Xmx 设置堆的最小和最大容量，当前主流 JVM 都按照可扩展实现。如果堆没有内存完成实例分配也无法扩展，抛出 OutOfMemoryError。

方法区的作用是什么?

方法区用于存储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数据。

JDK8 之前使用永久代实现方法区，容易内存溢出，因为永久代有 -XX:MaxPermSize 上限，即使不设置也有默认大小。JDK7 把放在永久代的字符串常量池、静态变量等移出，JDK8 中永久代完全废弃，改用在本地内存中实现的元空间代替，把 JDK 7 中永久代剩余内容（主要是类型信息）全部移到元空间。

虚拟机规范对方法区的约束宽松，除和堆一样不需要连续内存和可选择固定大小/可扩展外，还可以不实现垃圾回收。垃圾回收在方法区出现较少，主要目标针对常量池和类型卸载。如果方法区无法满足新的内存分配需求，将抛出 OutOfMemoryError。

运行时常量池的作用是什么?

运行时常量池是方法区的一部分，Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池表，用于存放编译器生成的各种字面量与符号引用，这部分内容在类加载后存放到运行时常量池。一般除了保存 Class 文件中描述的符号引用外，还会把符号引用翻译的直接引用也存储在运行时常量池。

运行时常量池相对于 Class 文件常量池的一个重要特征是动态性，Java 不要求常量只有编译期才能产生，运行期间也可以将新的常量放入池中，这种特性利用较多的是 String 的 intern 方法。

运行时常量池是方法区的一部分，受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError。

直接内存是什么?

直接内存不属于运行时数据区，也不是虚拟机规范定义的内存区域，但这部分内存被频繁使用，而且可能导致内存溢出。

JDK1.4 中新加入了 NIO 这种基于通道与缓冲区的 IO，它可以使用 Native 函数库直接分配堆外内存，通过一个堆里的 DirectByteBuffer 对象作为内存的引用进行操作，避免了在 Java 堆和 Native 堆来回复制数据。

直接内存的分配不受 Java 堆大小的限制，但还是会受到本机总内存及处理器寻址空间限制，一般配置虚拟机参数时会根据实际内存设置 -Xmx 等参数信息，但经常忽略直接内存，使内存区域总和大于物理内存限制，导致动态扩展时出现 OOM。

由直接内存导致的内存溢出，一个明显的特征是在 Heap Dump 文件中不会看见明显的异常，如果发现内存溢出后产生的 Dump 文件很小，而程序中又直接或间接使用了直接内存（典型的间接使用就是 NIO），那么就可以考虑检查直接内存方面的原因。

内存溢出和内存泄漏的区别?

内存溢出 OutOfMemory，指程序在申请内存时，没有足够的内存空间供其使用。

内存泄露 Memory Leak，指程序在申请内存后，无法释放已申请的内存空间，内存泄漏最终将导致内存溢出。

堆溢出的原因?

堆用于存储对象实例，只要不断创建对象并保证 GC Roots 到对象有可达路径避免垃圾回收，随着对象数量的增加，总容量触及最大堆容量后就会 OOM，例如在 while 死循环中一直 new 创建实例。

堆 OOM 是实际应用中最常见的 OOM，处理方法是通过内存映像分析工具对 Dump 出的堆转储快照分析，确认内存中导致 OOM 的对象是否必要，分清到底是内存泄漏还是内存溢出。

如果是内存泄漏，通过工具查看泄漏对象到 GC Roots 的引用链，找到泄露对象是通过怎样的引用路径、与哪些 GC Roots 关联才导致无法回收，一般可以准确定位到产生内存泄漏代码的具***置。

如果不是内存泄漏，即内存中对象都必须存活，应当检查 JVM 堆参数，与机器内存相比是否还有向上调整的空间。再从代码检查是否存在某些对象生命周期过长、持有状态时间过长、存储结构设计不合理等情况，尽量减少程序运行期的内存消耗。

栈溢出的原因?

由于 HotSpot 不区分虚拟机和本地方法栈，设置本地方法栈大小的参数没有意义，栈容量只能由 -Xss 参数来设定，存在两种异常：

StackOverflowError： 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError，例如一个递归方法不断调用自己。该异常有明确错误堆栈可供分析，容易定位到问题所在。

OutOfMemoryError： 如果 JVM 栈可以动态扩展，当扩展无法申请到足够内存时会抛出 OutOfMemoryError。HotSpot 不支持虚拟机栈扩展，所以除非在创建线程申请内存时就因无法获得足够内存而出现 OOM，否则在线程运行时是不会因为扩展而导致溢出的。

运行时常量池溢出的原因?

String 的 intern 方法是一个本地方法，作用是如果字符串常量池中已包含一个等于此 String 对象的字符串，则返回池中这个字符串的 String 对象的引用，否则将此 String 对象包含的字符串添加到常量池并返回此 String 对象的引用。

在 JDK6 及之前常量池分配在永久代，因此可以通过 -XX:PermSize 和 -XX:MaxPermSize 限制永久代大小，间接限制常量池。在 while 死循环中调用 intern 方法导致运行时常量池溢出。在 JDK7 后不会出现该问题，因为存放在永久代的字符串常量池已经被移至堆中。

方法区溢出的原因?

方法区主要存放类型信息，如类名、访问修饰符、常量池、字段描述、方法描述等。只要不断在运行时产生大量类，方法区就会溢出。例如使用 JDK 反射或 CGLib 直接操作字节码在运行时生成大量的类。很多框架如 Spring、Hibernate 等对类增强时都会使用 CGLib 这类字节码技术，增强的类越多就需要越大的方法区保证动态生成的新类型可以载入内存，也就更容易导致方法区溢出。

JDK8 使用元空间取代永久代，HotSpot 提供了一些参数作为元空间防御措施，例如 -XX:MetaspaceSize 指定元空间初始大小，达到该值会触发 GC 进行类型卸载，同时收集器会对该值进行调整，如果释放大量空间就适当降低该值，如果释放很少空间就适当提高。

创建对象的过程是什么?

字节码角度

NEW

如果找不到 Class 对象则进行类加载。加载成功后在堆中分配内存，从 Object 到本类路径上的所有属性都要分配。分配完毕后进行零值设置。最后将指向实例对象的引用变量压入虚拟机栈顶。

DUP:

在栈顶复制引用变量，这时栈顶有两个指向堆内实例的引用变量。两个引用变量的目的不同，栈底的引用用于赋值或保存局部变量表，栈顶的引用作为句柄调用相关方法。

INVOKESPECIAL

通过栈顶的引用变量调用 init 方法。

执行角度

1. 当 JVM 遇到字节码 new 指令时，首先将检查该指令的参数能否在常量池中定位到一个类的符号引用，并检查引用代表的类是否已被加载、解析和初始化，如果没有就先执行类加载。
2. 在类加载检查通过后虚拟机将为新生对象分配内存。
3. 内存分配完成后虚拟机将成员变量设为零值，保证对象的实例字段可以不赋初值就使用。

4. 设置对象头，包括哈希码、GC 信息、锁信息、对象所属类的类元信息等。

5. 执行 init 方法，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。

对象分配内存的方式有哪些?

对象所需内存大小在类加载完成后便可完全确定，分配空间的任务实际上等于把一块确定大小的内存块从 Java 堆中划分出来。

指针碰撞： 假设 Java 堆内存规整，被使用过的内存放在一边，空闲的放在另一边，中间放着一个指针作为分界指示器，分配内存就是把指针向空闲方向挪动一段与对象大小相等的距离。

空闲列表： 如果 Java 堆内存不规整，虚拟机必须维护一个列表记录哪些内存可用，在分配时从列表中找到一块足够大的空间划分给对象并更新列表记录。

选择哪种分配方式由堆是否规整决定，堆是否规整由垃圾收集器是否有空间压缩能力决定。使用 Serial、ParNew 等收集器时，系统采用指针碰撞；使用 CMS 这种基于清除算法的垃圾收集器时，采用空间列表。

对象分配内存是否线程安全?

对象创建十分频繁，即使修改一个指针的位置在并发下也不是线程安全的，可能正给对象 A 分配内存，指针还没来得及修改，对象 B 又使用了指针来分配内存。

解决方法：① CAS 加失败重试保证更新原子性。② 把内存分配按线程划分在不同空间，即每个线程在 Java 堆中预先分配一小块内存，叫做本地线程分配缓冲 TLAB，哪个线程要分配内存就在对应的 TLAB 分配，TLAB 用完了再进行同步。

对象的内存布局了解吗?

对象在堆内存的存储布局可分为对象头、实例数据和对齐填充。

对象头占 12B，包括对象标记和类型指针。对象标记存储对象自身的运行时数据，如哈希码、GC 分代年龄、锁标志、偏向线程 ID 等，这部分占 8B，称为 Mark Word。Mark Word 被设计为动态数据结构，以便在极小的空间存储更多数据，根据对象状态复用存储空间。

类型指针是对象指向它的类型元数据的指针，占 4B。JVM 通过该指针来确定对象是哪个类的实例。

实例数据是对象真正存储的有效信息，即本类对象的实例成员变量和所有可见的父类成员变量。存储顺序会受到虚拟机分配策略参数和字段在源码中定义顺序的影响。相同宽度的字段总是被分配到一起存放，在满足该前提条件的情况下父类中定义的变量会出现在子类之前。

对齐填充不是必然存在的，仅起占位符作用。虚拟机的自动内存管理系统要求任何对象的大小必须是 8B 的倍数，对象头已被设为 8B 的 1 或 2 倍，如果对象实例数据部分没有对齐，需要对齐填充补全。

对象的访问方式有哪些?

Java 程序会通过栈上的 reference 引用操作堆对象，访问方式由虚拟机决定，主流访问方式主要有句柄和直接指针。

句柄： 堆会划分出一块内存作为句柄池，reference 中存储对象的句柄地址，句柄包含对象实例数据与类型数据的地址信息。优点是 reference 中存储的是稳定

句柄地址，在 GC 过程中对象被移动时只会改变句柄的实例数据指针，而 reference 本身不需要修改。

直接指针： 堆中对象的内存布局就必须考虑如何放置访问类型数据的相关信息，reference 存储对象地址，如果只是访问对象本身就不需要多一次间接访问的开销。优点是速度更快，节省了一次指针定位的时间开销，HotSpot 主要使用直接指针进行对象访问。

如何判断对象是否是垃圾?

引用计数：在对象中添加一个引用计数器，如果被引用计数器加 1，引用失效时计数器减 1，如果计数器为 0 则被标记为垃圾。原理简单，效率高，但是在 Java 中很少使用，因为存在对象间循环引用的问题，导致计数器无法清零。

可达性分析：主流语言的内存管理都使用可达性分析判断对象是否存活。基本思路是通过一系列称为 GC Roots 的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程走过的路径称为引用链，如果某个对象到 GC Roots 没有任何引用链相连，则会被标记为垃圾。可作为 GC Roots 的对象包括虚拟机栈和本地方法栈中引用的对象、类静态属性引用的对象、常量引用的对象。

Java 的引用有哪些类型?

JDK1.2 后对引用进行了扩充，按强度分为四种：

强引用： 最常见的引用，例如 Object obj = new Object() 就属于强引用。只要对象有强引用指向且 GC Roots 可达，在内存回收时即使濒临内存耗尽也不会被回收。

软引用： 弱于强引用，描述非必需对象。在系统将发生内存溢出前，会把软引用关联的对象加入回收范围以获得更多内存空间。用来缓存服务器中间计算结果及不需要实时保存的用户行为等。

弱引用： 弱于软引用，描述非必需对象。弱引用关联的对象只能生存到下次 YGC 前，当垃圾收集器开始工作时无论当前内存是否足够都会回收只被弱引用关联的对象。由于 YGC 具有不确定性，因此弱引用何时被回收也不确定。

虚引用： 最弱的引用，定义完成后无法通过该引用获取对象。唯一目的就是为了能在对象被回收时收到一个系统通知。虚引用必须与引用队列联合使用，垃圾回收时如果出现虚引用，就会在回收对象前把这个虚引用加入引用队列。

有哪些 GC 算法?

标记-清除算法

分为标记和清除阶段，首先从每个 GC Roots 出发依次标记有引用关系的对象，最后清除没有标记的对象。

执行效率不稳定，如果堆包含大量对象且大部分需要回收，必须进行大量标记清除，导致效率随对象数量增长而降低。

存在内存空间碎片化问题，会产生大量不连续的内存碎片，导致以后需要分配大对象时容易触发 Full GC。

标记-复制算法

为了解决内存碎片问题，将可用内存按容量划分为大小相等的两块，每次只使用其中一块。当使用的这块空间用完了，就将存活对象复制到另一块，再把已使用过的内存空间一次清理掉。主要用于进行新生代。

实现简单、运行高效，解决了内存碎片问题。代价是可用内存缩小为原来的一半，浪费空间。

HotSpot 把新生代划分为一块较大的 Eden 和两块较小的 Survivor，每次分配内存只使用 Eden 和其中一块 Survivor。垃圾收集时将 Eden 和 Survivor 中仍然存活的对象一次性复制到另一块 Survivor 上，然后直接清理掉 Eden 和已用过的那块 Survivor。HotSpot 默认 Eden 和 Survivor 的大小比例是 8:1，即每次新生代中可用空间为整个新生代的 90%。

标记-整理算法

标记-复制算法在对象存活率高时要进行较多复制操作，效率低。如果不想浪费空间，就需要有额外空间分配担保，应对被使用内存中所有对象都存活的极端情况，所以老年代一般不使用此算法。

老年代使用标记-整理算法，标记过程与标记-清除算法一样，但不直接清理可回收对象，而是让所有存活对象都向内存空间一端移动，然后清理掉边界以外的内存。

标记-清除与标记-整理的差异在于前者是一种非移动式算法而后者是移动式的。如果移动存活对象，尤其是在老年代这种每次回收都有大量对象存活的区域，是一种极为负重的操作，而且移动必须全程暂停用户线程。如果不移动对象就会导致空间碎片问题，只能依赖更复杂的内存分配器和访问器解决。

你知道哪些垃圾收集器?

序列号

最基础的收集器，使用复制算法、单线程工作，只用一个处理器或一条线程完成垃圾收集，进行垃圾收集时必须暂停其他所有工作线程。

Serial 是虚拟机在客户端模式的默认新生代收集器，简单高效，对于内存受限的环境它是所有收集器中额外内存消耗最小的，对于处理器核心较少的环境，Serial 由于没有线程交互开销，可获得最高的单线程收集效率。

新品

Serial 的多线程版本，除了使用多线程进行垃圾收集外其余行为完全一致。

ParNew 是虚拟机在服务端模式的默认新生代收集器，一个重要原因是除了 Serial 外只有它能与 CMS 配合。自从 JDK 9 开始，ParNew 加 CMS 不再是官方推荐的解决方案，官方希望它被 G1 取代。

并行清理

新生代收集器，基于复制算法，是可并行的多线程收集器，与 ParNew 类似。

特点是它的关注点与其他收集器不同，Parallel Scavenge 的目标是达到一个可控制的吞吐量，吞吐量就是处理器用于运行用户代码的时间与处理器消耗总时间的比值。

串行旧

Serial 的老年代版本，单线程工作，使用标记-整理算法。

Serial Old 是虚拟机在客户端模式的默认老年代收集器，用于服务端有两种用途：① JDK5 及之前与 Parallel Scavenge 搭配。② 作为 CMS 失败预案。

平行老

Parallel Scavenge 的老年代版本，支持多线程，基于标记-整理算法。JDK6 提供，注重吞吐量可考虑 Parallel Scavenge 加 Parallel Old。

不育系

以获取最短回收停顿时间为目标，基于标记-清除算法，过程相对复杂，分为四个步骤：初始标记、并发标记、重新标记、并发清除。

初始标记和重新标记需要 STW（Stop The World，系统停顿），初始标记仅是标记 GC Roots 能直接关联的对象，速度很快。并发标记从 GC Roots 的直接关联对象开始遍历整个对象图，耗时较长但不需要停顿用户线程。重新标记则是为了修正并发标记期间因用户程序运作而导致标记产生变动的那部分记录。并发清除清理标记阶段判断的已死亡对象，不需要移动存活对象，该阶段也可与用户线程并发。

缺点：① 对处理器资源敏感，并发阶段虽然不会导致用户线程暂停，但会降低吞吐量。② 无法处理浮动垃圾，有可能出现并发失败而导致 Full GC。③ 基于标记-清除算法，产生空间碎片。

G1

开创了收集器面向局部收集的设计思路和基于 Region 的内存布局，主要面向服务端，最初设计目标是替换 CMS。

G1 之前的收集器，垃圾收集目标要么是整个新生代，要么是整个老年代或整个堆。而 G1 可面向堆任何部分来组成回收集进行回收，衡量标准不再是分代，而是哪块内存中存放的垃圾数量最多，回收受益最大。

跟踪各 Region 里垃圾的价值，价值即回收所获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级列表，每次根据用户设定允许的收集停顿时间优先

处理回收价值最大的 Region。这种方式保证了 G1 在有限时间内获取尽可能高的收集效率。

G1 运作过程：

1. 初始标记：标记 GC Roots 能直接关联到的对象，让下一阶段用户线程并发运行时能正确地在可用 Region 中分配新对象。需要 STW 但耗时很短，在 Minor GC 时同步完成。
2. 并发标记：从 GC Roots 开始对堆中对象进行可达性分析，递归扫描整个堆的对象图。耗时长但可与用户线程并发，扫描完成后要重新处理 SATB 记录的在并发时有变动的对象。
3. 最终标记：对用户线程做短暂暂停，处理并发阶段结束后仍遗留下来的少量 SATB 记录。
4. 筛选回收：对各 Region 的回收价值排序，根据用户期望停顿时间制定回收计划。必须暂停用户线程，由多条收集线程并行完成。

可由用户指定期望停顿时间是 G1 的一个强大功能，但该值不能设得太低，一般设置为 100~300 ms。

ZGC 了解吗?

JDK11 中加入的具有实验性质的低延迟垃圾收集器，目标是尽可能在不影响吞吐量的前提下，实现在任意堆内存大小都可以把停顿时间限制在 10ms 以内的低延迟。

基于 Region 内存布局，不设分代，使用了读屏障、染色指针和内存多重映射等技术实现可并发的标记-整理，以低延迟为首要目标。

ZGC 的 Region 具有动态性，是动态创建和销毁的，并且容量大小也是动态变化的。

你知道哪些内存分配与回收策略?

对象优先在 Eden 区分配

大多数情况下对象在新生代 Eden 区分配，当 Eden 没有足够空间时将发起一次 Minor GC。

大对象直接进入老年代

大对象指需要大量连续内存空间的对象，典型是很长的字符串或数量庞大的数组。大对象容易导致内存还有不少空间就提前触发垃圾收集以获得足够的连续空间。

HotSpot 提供了 -XX:PretenureSizeThreshold 参数，大于该值的对象直接在老年代分配，避免在 Eden 和 Survivor 间来回复制。

长期存活对象进入老年代

虚拟机给每个对象定义了一个对象年龄计数器，存储在对象头。如果经历过第一次 Minor GC 仍然存活且能被 Survivor 容纳，该对象就会被移动到 Survivor 中并将年龄设置为 1。对象在 Survivor 中每熬过一次 Minor GC 年龄就加 1，当增加到一定程度（默认 15）就会被晋升到老年代。对象晋升老年代的阈值可通过 -XX:MaxTenuringThreshold 设置。

动态对象年龄判定

为了适应不同内存状况，虚拟机不要求对象年龄达到阈值才能晋升老年代，如果在 Survivor 中相同年龄所有对象大小的总和大于 Survivor 的一半，年龄不小于该年龄的对象就可以直接进入老年代。

空间分配担保

MinorGC 前虚拟机必须检查老年代最大可用连续空间是否大于新生代对象总空间，如果满足则说明这次 Minor GC 确定安全。

如果不满足，虚拟机会查看 -XX:HandlePromotionFailure 参数是否允许担保失败，如果允许会继续检查老年代最大可用连续空间是否大于历次晋升老年代对象的平均大小，如果满足将冒险尝试一次 Minor GC，否则改成一次 FullGC。

冒险是因为新生代使用复制算法，为了内存利用率只使用一个 Survivor，大量对象在 Minor GC 后仍然存活时，需要老年代进行分配担保，接收 Survivor 无法容纳的对象。

你知道哪些故障处理工具?

jps：虚拟机进程状况工具

功能和 ps 命令类似：可以列出正在运行的虚拟机进程，显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一 ID（LVMID)。LVMID 与操作系统的进程 ID（PID）一致，使用 Windows 的任务管理器或 UNIX 的 ps 命令也可以查询到虚拟机进程的 LVMID，但如果同时启动了多个虚拟机进程，必须依赖 jps 命令。

jstat：虚拟机统计信息监视工具

用于监视虚拟机各种运行状态信息。可以显示本地或远程虚拟机进程中的类加载、内存、垃圾收集、即时编译器等运行时数据，在没有 GUI 界面的服务器上是运行期定位虚拟机性能问题的常用工具。

参数含义：S0 和 S1 表示两个 Survivor，E 表示新生代，O 表示老年代，YGC 表示 Young GC 次数，YGCT 表示 Young GC 耗时，FGC 表示 Full GC 次数，FGCT 表示 Full GC 耗时，GCT 表示 GC 总耗时。

jinfo：Java 配置信息工具

实时查看和调整虚拟机各项参数，使用 jps 的 -v 参数可以查看虚拟机启动时显式指定的参数，但如果想知道未显式指定的参数值只能使用 jinfo 的 -flag 查询。

jmap：Java 内存映像工具

用于生成堆转储快照，还可以查询 finalize 执行队列、Java 堆和方法区的详细信息，如空间使用率，当前使用的是哪种收集器等。和 jinfo 一样，部分功能在 Windows 受限，除了生成堆转储快照的 -dump 和查看每个类实例的 -histo 外，其余选项只能在 Linux 使用。

jhat：虚拟机堆转储快照分析工具

JDK 提供 jhat 与 jmap 搭配使用分析 jmap 生成的堆转储快照。jhat 内置了一个微型的 HTTP/Web 服务器，生成堆转储快照的分析结果后可以在浏览器查看。

jstack：Java 堆栈跟踪工具

用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的目的通常是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间挂起等。线程出现

停顿时通过 jstack 查看各个线程的调用堆栈，可以获知没有响应的线程在后台做什么或等什么资源。

Java 程序是怎样运行的?

1. 首先通过 Javac 编译器将 .java 转为 JVM 可加载的 .class 字节码文件。
2. Javac 是由 Java 编写的程序，编译过程可以分为：① 词法解析，通过空格分割出单词、操作符、控制符等信息，形成 token 信息流，传递给语法解析器。② 语法解析，把 token 信息流按照 Java 语法规则组装成语法树。③ 语义分析，检查关键字使用是否合理、类型是否匹配、作用域是否正确等。④ 字节码生成，将前面各个步骤的信息转换为字节码。
3. 字节码必须通过类加载过程加载到 JVM 后才可以执行，执行有三种模式，解释执行、JIT 编译执行、JIT 编译与解释器混合执行 (主流 JVM 默认执行的方式)。混合模式的优势在于解释器在启动时先解释执行，省去编译时间。
4. 之后通过即时编译器 JIT 把字节码文件编译成本地机器码。
5. Java 程序最初都是通过解释器进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁，就会认定其为"热点代码"，热点代码的检测主要有基于采样和基于计数器两种方式，为了提高热点代码的执行效率，虚拟机会把它们编译成本地机器码，尽可能对代码优化，在运行时完成这个任务的后端编译器被称为即时编译器。
6. 还可以通过静态的提前编译器 AOT 直接把程序编译成与目标机器指令集相关的二进制代码。

类加载是什么?

Class 文件中描述的各类信息都需要加载到虚拟机后才能使用。JVM 把描述类的数据从 Class 文件加载到内存，并对数据进行校验、解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这个过程称为虚拟机的类加载机制。

与编译时需要连接的语言不同，Java 中类型的加载、连接和初始化都是在运行期间完成的，这增加了性能开销，但却提供了极高的扩展性，Java 动态扩展的语言特性就是依赖运行期动态加载和连接实现的。

一个类型从被加载到虚拟机内存开始，到卸载出内存为止，整个生命周期经历加载、验证、准备、解析、初始化、使用和卸载七个阶段，其中验证、解析和初始化三个部分称为连接。加载、验证、准备、初始化阶段的顺序是确定的，解析则不一定：可能在初始化之后再开始，这是为了支持 Java 的动态绑定。

类初始化的情况有哪些?

1. 遇到 new、getstatic、putstatic 或 invokestatic 字节码指令时，还未初始化。典型场景包括 new 实例化对象、读取或设置静态字段、调用静态方法。
2. 对类反射调用时，还未初始化。
3. 初始化类时，父类还未初始化。
4. 虚拟机启动时，会先初始化包含 main 方法的主类。
5. 使用 JDK7 的动态语言支持时，如果 MethodHandle 实例的解析结果为指定类型的方法句柄且句柄对应的类还未初始化。
6. 口定义了默认方法，如果接口的实现类初始化，接口要在其之前初始化。

7. 其余所有引用类型的方式都不会触发初始化，称为被动引用。被动引用实例：① 子类使用父类的静态字段时，只有父类被初始化。② 通过数组定义使用类。③ 常量在编译期会存入调用类的常量池，不会初始化定义常量的类。

8. 接口和类加载过程的区别：初始化类时如果父类没有初始化需要初始化父类，但接口初始化时不要求父接口初始化，只有在真正使用父接口时（如引用接口中定义的常量）才会初始化。

类加载的过程是什么？

加载

该阶段虚拟机需要完成三件事：① 通过一个类的全限定类名获取定义类的二进制字节流。② 将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区。③ 在内存中生成对应该类的 Class 实例，作为方法区这个类的数据访问入口。

验证

确保 Class 文件的字节流符合约束。如果虚拟机不检查输入的字节流，可能因为载入有错误或恶意企图的字节流而导致系统受攻击。验证主要包含四个阶段：文件格式验证、元数据验证、字节码验证、符号引用验证。

验证重要但非必需，因为只有通过与否的区别，通过后对程序运行期没有任何影响。如果代码已被反复使用和验证过，在生产环境就可以考虑关闭大部分验证缩短类加载时间。

准备

为类静态变量分配内存并设置零值，该阶段进行的内存分配仅包括类变量，不包括实例变量。如果变量被 final 修饰，编译时 Javac 会为变量生成 ConstantValue 属性，准备阶段虚拟机会将变量值设为代码值。

解析

将常量池内的符号引用替换为直接引用。

符号引用以一组符号描述引用目标，可以是任何形式的字面量，只要使用时能无歧义地定位目标即可。与虚拟机内存布局无关，引用目标不一定已经加载到虚拟机内存。

直接引用是可以直接指向目标的指针、相对偏移量或能间接定位到目标的句柄。和虚拟机的内存布局相关，引用目标必须已在虚拟机的内存中存在。

初始化

直到该阶段 JVM 才开始执行类中编写的代码。准备阶段时变量赋过零值，初始化阶段会根据程序员的编码去初始化类变量和其他资源。初始化阶段就是执行类构造方法中的 `` 方法，该方法是 Javac 自动生成的。

有哪些类加载器?

自 JDK1.2 起 Java 一直保持三层类加载器：

启动类加载器

在 JVM 启动时创建，负责加载最核心的类，例如 Object、System 等。无法被程序直接引用，如果需要把加载委派给启动类加载器，直接使用 null 代替即可，因为启动类加载器通常由操作系统实现，并不存在于 JVM 体系。

平台类加载器

从 JDK9 开始从扩展类加载器更换为平台类加载器，负载加载一些扩展的系统类，比如 XML、加密、压缩相关的功能类等。

应用类加载器

也称系统类加载器，负责加载用户类路径上的类库，可以直接在代码中使用。如果没有自定义类加载器，一般情况下应用类加载器就是默认的类加载器。自定义类加载器通过继承 ClassLoader 并重写 findClass 方法实现。

双亲委派模型是什么?

类加载器具有等级制度但非继承关系，以组合的方式复用父加载器的功能。双亲委派模型要求除了顶层的启动类加载器外，其余类加载器都应该有自己的父加载器。

一个类加载器收到了类加载请求，它不会自己去尝试加载，而将该请求委派给父加载器，每层的类加载器都是如此，因此所有加载请求最终都应该传送到启动类加载器，只有当父加载器反馈无法完成请求时，子加载器才会尝试。

类跟随它的加载器一起具备了有优先级的层次关系，确保某个类在各个类加载器环境中都是同一个，保证程序的稳定性。

如何判断两个类是否相等?

任意一个类都必须由类加载器和这个类本身共同确立其在虚拟机中的唯一性。

两个类只有由同一类加载器加载才有比较意义，否则即使两个类来源于同一个 Class 文件，被同一个 JVM 加载，只要类加载器不同，这两个类就必定不相等。