JVM 部分：

1. Java 语言的优势？平台无关，内存管理与访问机制相对安全，有完善的程序接口等。
2. JDK 和 JRE？JDK 用于支持 Java 程序开发的最小环境，JRE 是 Java SE API + JVM，它是 Java 程序运行的标准环境。
3. Java 内存区域的划分？哪些是线程私有的？哪些是内存公有的？它们分别的作用？
   程序计数器 PC - 当前线程所执行的字节码的行号指示器，不会抛出 OOM
   Java 虚拟机栈 - 存储基本数据类型和对象引用，生命周期和线程相同。调用方法的过程转换为**栈帧**入栈出栈的过程，64 位长度的 long 和 double 会占用 2 个局部变量的空间(Slot)，其余都占用一个。超出栈深度会出现 SOF 异常，无法申请内存则抛出 OOM
   本地方法栈 - 服务于 Native 方法的调用。HotSpot 虚拟机将本地方法栈和虚拟机栈合二为一，同样有 SOF 和 OOM 异常
   Java 堆 - 线程共享，虚拟机启动时创建，存储所有的对象实例及数组，垃圾回收器主要回收区域，可分为**新生代和老生代(Eden、From Survivor、 To Survivor)**，堆中可能划分多个线程私有的缓冲区(多线程不可见性？)，可以通过-Xmx -Xms 来调节堆的大小，没有内存空间则 OOM
   方法区 - 线程共享，存储已被虚拟机加载的**类信息**、**常量**、**静态变量**、即时编译器编译后的**代码。**JDK1.7 前，GC 扩展到方法区，使用永久代实现方法区。JDK7 及以后，已经将永久代移除，在 Heap 中开辟了一块区域存放运行时常量池。**JDK8 中，已经没有了永久代，方法区直接放在了与堆不相连的本地内存区域，这个区域叫做元空间。**默认为最大可分配内存空间。http://www.infoq.com/cn/articles/Java-PERMGEN-Removed?utm_campaign=infoq_content&，元空间最大的问题为碎片化问题，此区域也会 OOM
   运行时常量池 - 原来存储在方法区中，现在存储在元空间中，主要存放 Class 字节码文件中包含的编译期生成的**字面量和符号引用，**常量池中的内容不一定是编译期生成，也可以运行时增加，如 **String.intern()**，也会 OOM
   直接内存 - JDK1.4 新增的 NIO 引入了基于通道和缓冲区的 IO 方式，可以使用 Native 直接分配堆外内存，然后以存储在堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。**(文件复制的一种方法)** 它会受到本机内存总和的制约。
4. 创建对象的过程？
   遇到 new 指令
   -> 判断常量池中是否已经存在对应的 Class
   -> 不存在则加载、解析、初始化
   -> 为新生对象分配内存空间(**指针碰撞法**<Serial、ParNew 等带有 Compact 过程>和**空闲列表法**<CMS 等 Mark-Sweep>)，划分时还要考虑**同步**问题<使用 **CAS 锁和失败重试**，或使用**本地线程分配缓冲**，先放到缓冲中，满了再锁定和写入堆>
   -> 分配空间之后的初始化(默认初始化)
   -> 虚拟机对对象进行必要的设置(Object Header)
   -> 执行 init 方法，按照程序来进行初始化
5. 一个对象在内存中的存储布局？对象头、实例数据和对齐填充。
   对象头包含了 **HashCode、GC 分代年龄、锁状态标识、线程持有的锁**等，还包含**类型指针**、(数组的话还有**数组长度**)

实例数据会按照一定顺序排列，相同空间的字段放一起，父类变量在子类变量前**对齐填充**能保证对象的大小为 **8 字节的整数倍(对象的引用存放在栈中，为 4 字节，1Slot)，**仅仅只是起着占位符的作用

6. 对象的访问定位？**句柄**和**直接指针**两种。
   句柄方式：栈引用的是句柄，句柄中包含对象实例数据与类型数据各自的具体地址信息**(保存两个指针)**。----------在 reference 存储稳定的句柄地址
   直接指针方式：栈引用的是真正存放对象的地址，地址中包含对象类型的指针**(保存实例数据和类型指针)**。----------查询较快
7. 如何诊断和定位 OOM 异常？
   通用方法 - 运行时加入**-XX:PrintGCDetails** 可以查看 GC 的日志信息
   Java 堆内存 - 配置参数**-XX:+HeapDumpOnOutOfMemoryError** 可以在内存溢出时 Dump 出堆转储快照来分析。通过**-Xmx** 和**-Xms** 来改变堆大小，以避免溢出。导出快照后，导入到 Jvisualvm 中进行查看，可以看到各个类对象占用的空间百分比，根据这个来判断占用资源最多的对象是否需要，如果需要，则扩大堆大小，否则，查看程序的 BUG。
   虚拟机栈和本地方法栈 - 栈容量只由-Xss 参数来设定，无论增大栈帧大小还是缩小栈深度，都抛出 SOF 异常，**且栈分配越大，越容易出现内存溢出**。因为每个线程独享栈空间，多线程时很容易总和超过本机内存大小。**此时需要减小堆大小，减小栈容量。**
   方法区和运行时常量池溢出 - 可以使用-XX:PermSize 和-XX:MaxPermSize 来限制方法区的大小，间接限制常量池容量。JDK8 中应使用-XX:MetaspaceSize 来控制元空间大小。-XX:MinMetaspaceFreeRatio 和-XX:MaxMetaspaceFreeRatio 用来设置元空间空闲比例。String 的 intern()方法，首次出现的字符串会在常量池中添加指向堆的引用。**使用动态代里和 JSP 应用时，会动态产生类，经常导致方法区内存溢出。**
   本机直接内存溢出 - 使用-XX:MaxDirectMemorySize 来指定，默认-Xmx。直接内存溢出的特点是**堆 Dump 较小**。
8. 如何确定哪些内存需要回收？使用**引用计数算法**(很难解决相互循环引用的问题) -> **可达性分析算法**(GCRoot 对象包括虚拟机栈中引用的对象、方法区中类静态属性引用的对象、方法区中常量引用的对象、本地方法栈中 JNI 引用的对象)
9. Java 包含的引用类型？各自用在什么地方？区别是什么？
   强引用 - 程序代码中普遍存在的，只要强引用存在，GC 就不会回收。
   软引用 - 使用 **SoftReference** 来表示，在系统将要发生内存溢出前，会把这些对象列进回收范围之中进行第二次回收。**适合用来创建缓存。**
   弱引用 - 使用 **WeakReference** 类表示，弱引用关联的对象只能存活到下一次垃圾收集器发生之前。**应用有 WeakHashMap。**
   虚引用 - **PhantomReference** 类表示，不能通过该引用获得对象实例，但可以**在系统回收该对象时收到一个系统通知**（使用的唯一目的）。
10. 对象宣告死亡需要被标记两次，第一次是在判断 GCROOT 不可达之后标记一次，如果该对象没有覆盖 finalize 方法或已经调用过了 finalize 方法，则对象会在第二次回收。如果覆盖了 finalize 方法，则会将对象放到 F-Queue 队列中，等待 finalize 调用，这时，如果引用可达，则可以成功逃脱回收。但如果再来一次，则 JVM 不会调用 finalize 方法，则一定会被回收。
11. 永久代的垃圾回收主要针对**废弃常量**和**无用的类**。废弃常量即没有其他引用指向常量池中常量。无用的类必须所有实例被回收、加载类的 ClassLoader 被回收、该类

的 Class 对象也没有被引用。大量反射、动态代理等场景需要虚拟机具备类卸载功能，配置-Xnoclassgc 来控制类是否可以 GC。

12. 垃圾回收算法？

标记清除算法 - 先标记再回收**(标记和清除效率都不高，且回收后会产生大量的内存碎片)**

复制算法 - 使用一半，需要回收时将需要的部分移动到另一半，然后直接清除**(实现简单，运行高效，但运行时内存少了一半)** 改进版本：**新生代分割为 Eden、From Survivor、To Survivor**，比例为 8:1:1，这样只有 10%的空间被浪费。**当 Survivor 不够用时，需要老年代来分配担保。**

标记整理算法 - **老年代选用该算法**，先标记，然后将还存活的对象向一端移动，并清理掉端边界以外的内存。

分代收集算法 - **即新生代使用复制算法，老年代使用标记整理算法。**

13. HotSpot 算法实现？

枚举根节点 - 由于全局引用和执行上下文的 GCROOT 太多，而由于 GC 的存在，引用都是变化的，所以为了在枚举根节点过程中，需要 GC Stop the World。一般在编译时就存入 **OopMap** 中。

安全点 - 指程序到达安全点之后才能开始 GC。到达安全点后，采用主动式中断，置标志位的方式来通知 GC。

安全区域 - 有些线程 Sleep 或 Blocked 时，线程无法响应 JVM 请求。这种情况使用安全区域来解决，保证 GC 安全。

14. 垃圾回收器？

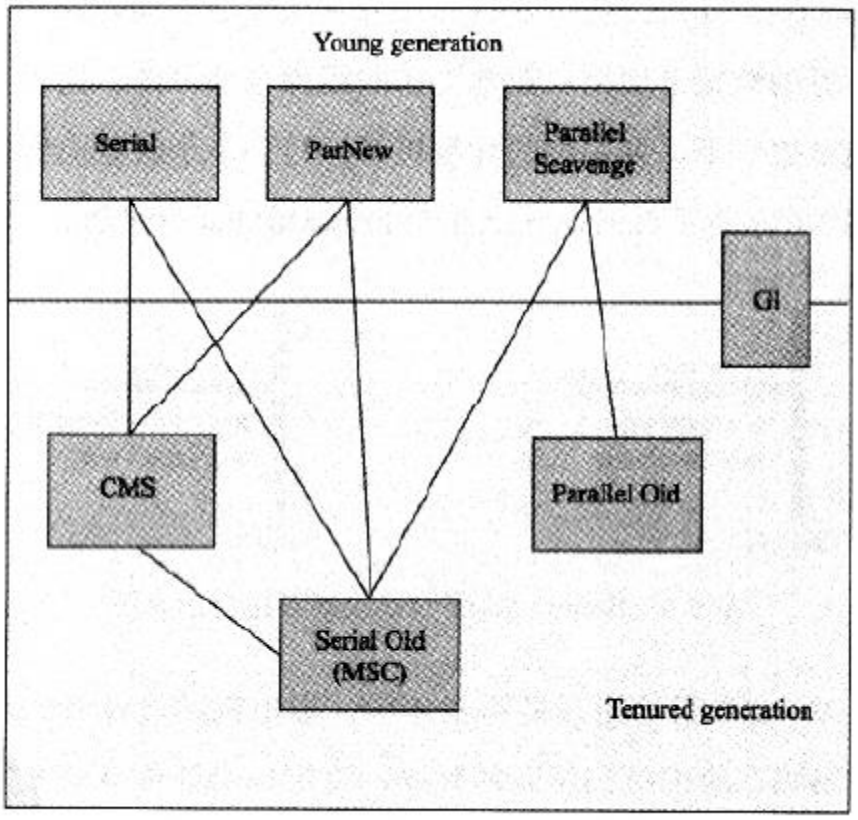


图 3-5 HotSpot 虚拟机的垃圾收集器[⊖]

Serial 收集器 - (Stop the World)单线程，收集时必须暂停所有工作线程。采用复

制算法。它是 Client 模式下默认的新生代收集器。
ParNew 收集器 - (Stop the World)Serial 的多线程版本。它是 Server 模式下默认的新生代收集器，除 Serial 收集器外，只有它可以与 CMS 收集器配合。在单 CPU 的环境下不会比 Serial 效果更好，存在线程切换的开销。
Parallel Scavenge 收集器 - **吞吐量优先的收集器**。(用户时间/总时间)。可以使用-XX:MaxGCPauseMillis 和-XX:GCTimeRatio 来指定停顿时间和吞吐量。另外，还可以提供 GC **自适应调节策略**，这是和 ParNew 最大的区别。

Serial Old 收集器 - 老年代单线程收集器，采用标记整理。可以在 client 模式配合 serial，也可以 server 模式配合 Parallel Scavenge。
Parallel Old 收集器 - 多线程，标记整理，可以和 Parallel Scavenge 组合，成为多线程收集组合。
CMS 收集器 - **并发收集、低停顿**的标记清除收集器，以**获取最短回收停顿时间**为目标。使用在重视服务响应速度，希望系统停顿时间最短，用户体验较好的地方。运作过程分四步：**初始标记(单线程) -> 并发标记(与用户线程并发) -> 重新标记(stw) -> 并发清除(与用户线程并发)**，1,3 步骤需要 Stop the World。重新标记是为了修正并发标记时期用户程序运作导致的标记变动，比初始标记长，但远比并发标记时间短。CMS 的三个缺点：1. 对 CPU 资源敏感。 2. 无法处理并发清理阶段产生的垃圾。 3. 标记清除会产生碎片，需要 Full GC 来合并碎片。
G1 收集器 - 利用并行和并发，尽量缩短 Stop the World 时间，分代收集，使用标记整理算法，停顿时间模型可预测。运作过程：**初始标记(单线程) -> 并发标记(与用户线程并发) -> 最终标记(stw) -> 筛选回收(stw)**。追求低停顿可以选择 G1，追求吞吐量则不要选择 G1。

a) Serial New 收集器是针对新生代的收集器，采用的是复制算法
b) ParNew（并行）收集器，新生代采用复制算法，老年代采用标记整理
c) Parallel Scavenge（并行）收集器，针对新生代，采用复制收集算法
d) Serial Old（串行）收集器，新生代采用复制，老年代采用标记清理
e) Parallel Old（并行）收集器，针对老年代，标记整理
f) CMS 收集器，基于标记清理
g) G1 收集器：整体上是基于标记清理，局部采用复制

15. GC 日志的组成？
GC 类型：GC 表示没发生停顿，FULL GC 表示 STW。调用 system.gc（）会执行 fullgc
GC 发生区域：[DefNew 新生代、[Tenured 老生代、 [Perm 永久代
垃圾回收空间和时间数据： 回收前使用 -> 回收后使用(总量)，时间
堆垃圾回收空间和时间数据：回收前使用 -> 回收后使用(总量)，时间
[Times: user=0.01, sys=0.00, real=0.02secs]：用户态消耗 CPU 时间、内核态消耗 CPU 时间和操作开始到结束的时间。
16. 垃圾回收器参数：
-XX:SurvivorRatio：新生代 Eden 和 Survivor 区域的比例比较
-Xmx -Xms:堆内存的最大值和最小值
-Xmn：新生代大小

-Xss：栈内存大小
-XX:PermSize：方法区大小
-XX:MaxPermSize：最大方法区大小

17. 内存分配和回收策略：

对象优先在 Eden 分配 - **-XX:+PrintGCDetails** 在发生垃圾收集行为是打印内存回收日志

大对象直接进入老年代 - **-XX:PretenureSizeThreshold** 参数指定大小阈值，单位为字节

长期存活的对象将进入老年代 - **-XX:MaxTenuringThreshold** 指定年龄阈值，默认 15 岁，即经历 15 次 GC 之后依然存活的对象

动态对象年龄判断 - 若 Survivor 空间中某年龄的对象过半，则大于或等于该年龄的对象都会进入老年代。

18. 新生代 GC 和老生代 GC 的区别？新生代更频繁，老生代速度比新生代慢 10 倍以上。
19. 空间分配担保失败下，会被迫进行 FullGC。另外，如果老年代连续空间小于新生代对象总大小或者晋升平均大小也会进行 FullGC，否则会进行 Minor GC。
20. 虚拟机性能监控和故障处理工具：

表 4-1　Sun JDK 监控和故障处理工具

名　　称	主要作用
jps	JVM Process Status Tool，显示指定系统内所有的 HotSpot 虚拟机进程
jstat	JVM Statistics Monitoring Tool，用于收集 HotSpot 虚拟机各方面的运行数据
jinfo	Configuration Info for Java，显示虚拟机配置信息

名　　称	主要作用
jmap	Memory Map for Java，生成虚拟机的内存转储快照（heapdump 文件）
jhat	JVM Heap Dump Browser，用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果
jstack	Stack Trace for Java，显示虚拟机的线程快照

jps - 显示系统所有的 JVM 进程并显示虚拟机执行主类名称和这些进程在本地虚拟机唯一 ID

jstat - 指定 pid 收集虚拟机各方面运行数据(compile、GC 等)，类装载，内存垃圾回收，JIT 编译等运行数据

jinfo - 查看指定 pid 的配置信息，

jmap - 指定 pid 转储堆快照，还可以查询 finalize 执行队列，javad 堆和永久代的信息

jhat - 分析堆转储快照，并生成 HTML，不如用 VisualVM

jstack - 指定 pid 分析当前线程快照。调试线程死锁必备！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，定位长时间停顿的原因。

HSDIS - JIT 生成代码反编译

jvisual、jConsole - JDK 可视化工具，可以检测死锁，查看 GC 的次数等。BTrace 还可以动态加入调试程序。打印堆栈，参数，返回值只是最基本的应用，它还可以进行性能监控，定位连接泄漏和内存泄漏，解决多线程竞争的问题。

21. JVM 调优的过程思路：
    - 首先考虑升级 JDK,
    - 打开 jvisualvm，查看 GC 占用时间，堆内存占用情况，视情况调整堆大小。(首先解决 Minor GC，如果新生代 GC 频发，每次 GC 操作，都跑到安全点挂起等待回收，动作太多，这很有可能是因为虚拟机给新生代的空间太小而导致的，通过-Xmx 调整新生代大小。Full GC 查看 GC 日志如果每次 GC 都扩容，那么会浪费时间，这时可以将堆和永久代的容量固定下来，避免运行时自动扩展，将新生代和堆和老年代都固定在合适的大小)。
    - **编译代码时间过长**，可以禁用字节码验证来加快编译速度。-Xverify:none。
    - **垃圾回收次数过多**？使用-Xmn 调整新生代大小。老年代的回收次数太多，可以固定老年代和永久代的空间，避免自动扩展。另外，可以显式屏蔽 System.gc()，使用-XX:+DisableExplicitGC。
    - **垃圾回收停顿时间过长**？CPU 资源丰富->使用 CMS 收集器和 ParNew 收集器来减少停顿时间。
    - 如果查看 GC，发现 FullGC 时间过长，很可能是由于堆太大导致。此时可以考虑通过 64 位 JDK 来使用，或者分割为若干 32 位虚拟机，建立逻辑集群方式来利用资源，尽量保证绝大多数对象都由新生代回收，尽量不要进入老年代。
    - OOM 类似错误使用堆 Dump 日志来查看错误，进而确定溢出的位置，推测溢出的原因。
    - 由于本地内存引起的 OOM，通过堆 Dump 看不出来，可以通过异常堆栈来确定。
    - 线程堆栈引起的 OOM，在内存不足时会 SOF，如果无法创建新的线程，往往是本机内存不足，不能创建新的栈了。
    - Socket 缓冲区占用也会很多，可能会抛出 IOException: Too many open files 异常。
22. Class 类文件结构？魔数 CAFEBABE + 主次版本号(50 - > JDK1.6) + 常量池(使用 javap -verbose 可以查看 class 文件的常量信息) + 访问标志(判断是否公有、接口、abstract 等) + 类索引(类的全限定名)、父类索引(父类的全限定名)和接口索引集合(类定义时的继承关系引用) + 字段表集合(类中字段，包括访问规则等，不会出现父类继承的字段，但可能会增加一些代码中不存在的字段，如内部类保持对外部类的访问) + 方法表集合(访问标志、方法表，，不包含继承方法，不包含实现代码，注意 init 实例构造器和 cinit 类构造器) + 属性表集合(存储方法的实现)
23. 常量池中的信息？
    字面量-接近 java 语言层面的常量的概念
    符号引用（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） 虚拟机加载 Class 文件的时候进行**动态连接**，在 class 文件中不会保存各个方法，字段的最终内存布局信息，这些方法和字段不经过运行期的转换无法得到真正的内存入口，需要先从常量池中获取符号引用，再在类创建和运行时解析，翻译到具体的内存地址中。
24. 9 大字节码指令：(编译器会在编译期将 byte 和 short 类型扩展为 int，boolean，char 也同理)
    加载和存储指令：iload、fload、aload; istore... ; iconst ...;
    运算指令：iadd、isub、irem、ineg、ishl、ixor、iand....
    类型转换指令：**数值窄化不会产生异常！**

对象创建与访问指令：new、newarray、arraylength、instanceof、getfield....
操作数栈管理指令：pop、dup、swap...
控制转移指令：ifeq、lookupswitch、goto...
方法调用和返回指令：invokevirtual、invokeinterface、Invokespecial、invokestatic、invokedynamic

25. 异常处理指令：idiv（异常采用异常表实现）
    同步指令：使用管程 monitorenter、monitorexit
26. 虚拟机类加载是在什么时期完成的？类加载的过程？
    运行期完成。
    加载 -> 连接(检验、准备、解析) -> 初始化
    初始化过程触发的条件：
    1. 遇到 new、getstatic、putstatic、invokestatic4 条指令时，没初始化需要触发。**(常量池除外)**
    2. 使用反射包对类进行反射调用时，没初始化需要先初始化。
    3. 初始化一个类时，先要初始化其父类。**(接口引用父接口除外)**
    4. 当虚拟机启动时，包含 main 方法的类会先初始化。
    5. 如果 java.lang.invoke.MethodHandle 实例需要 getStatic、putStatic、invoketatic 方法时，优先初始化。

    由此，可以了解到，父类静态属性由子类来访问时，不会加载子类；new 了一个父类的数组，也不会加载父类；类访问常量字段，也不会加载类。
27. **加载类的过程：首先通过全类名获取类的二进制字节流，然后转换为方法区的运行时数据结构，最后生成 Class 对象，作为类数据的访问入口。**
    注意：二进制流的来源可以通过 jar 包载入，也可以动态生成，如动态代理，JSP，数据库等。
    一个非数组的类可以通过自定义类加载器，重写 loadClass()方法来完成。
28. **验证阶段的目的是为了验证字节码是否符合要求。**(文件格式(组成顺序是否正确)、元数据(语言规范)、字节码(语法校验)、符号引用(引用完整性)四个方面来验证)。**可以配置-Xverify:none 跳过验证。**
29. **准备阶段主要为类变量分配内存并设置类变量初始值。这里的初始值不是初始化，只是赋值为 0。**
30. **解析阶段只要将常量池内的符号引用替换为直接引用。即把代码中的名称引用转变成对于具体类的引用。**
31. **初始化阶段是执行类构造器 cinit 方法的过程。cinit 方法可以修改后声明的变量，但不能访问，它能保证父类类构造器先加载，但不是显式调用关系。cinit 方法不是必须的。接口中也包含 cinit 方法，但接口的父类 clinit 方法不需要先执行。执行 clinit 方法是线程安全的，同时只能有一个线程执行该方法。**
32. **两个类是否相等，只有在同一个类加载器加载的前提下才有意义。**
33. 类加载的双亲委派机制？
    启动类加载器 - 加载存放 JAVA_HOME\lib 中的 jar 文件
    扩展类加载器 - 加载 JAVA_HOME\lib\ext 中的 jar 文件
    应用程序类加载器 - 加载用户类路径 classpath 下的 jar 文件，默认的类加载器
34. 双亲委派模型：如果一个类加载器收到了类加载的请求，它首先交给父类加载器去完成，每一层加载器都是如此。只有当父类加载器无法加载时，子加载器才会尝试

自己去加载。
好处：防止编程人员恶意修改 JDK 代码，处于安全的考虑。同时，可以明确加载任务，不会发生重复加载的事情。

35. 破坏双亲委派机制：
    JNDI 的设计：启动类加载器加载的 JNDI 需要加载用户程序，引入了线程上下文类加载器，可以传入一个类加载器。
    代码热替换：OSGi，每个模块都有一个自己的类加载器，更换模块时，只要把 Bundle 和加载器一起换掉就可以了。
36. **栈帧**用于支持虚拟机进行方法调用和执行，它存储了**局部变量表、操作数栈、动态链接和方法返回地址**等信息。**编译代码时，栈帧需要分配的内存就已经确定。**
    局部变量表 - 用来存储方法内部的局部变量。一般局部变量表 0 位 SLOP 是 this 指针。**局部变量没有准备过程，不存在初始值，如果不显式初始化，则会抛出异常。**
    操作数栈 - 用来存储临时变量，进行程序解释执行的栈。
    动态链接 - 指向该栈帧所属方法的引用。
    方法返回地址 - 方法退出包括正常退出和异常退出，正常退出在栈帧中有计数值。而异常退出信息保存在异常处理器中，不保存在栈帧中。
    后两者和附加信息统称为**栈帧信息**。
37. 调用目标方法的确定，包括静态方法和私有方法两大类。5 条方法调用字节码指令：invokestatic、invokespecial、invokevirtural、invokeinterface、invokedynamic。invokestatic 和 invokespecial 都可以**在解析阶段(编译期)**确定唯一的调用版本，包括静态方法、私有方法、实例构造器和父类方法和 final 标记的方法。
38. **解析调用在编译期间就完全确定，在类装载的解析阶段就转换为直接引用，不会在运行期完成。**
39. 分派机制。
    静态分派(方法重载) - **在编译期就已经决定了调用的哪个重载方法，而编译时根据变量的静态类型(外观类型)来确定的重载方法，所以并不会在运行时改变调用方法**。所以尹老师那题 Set Collection 和 List 最后都是 Collection。
    另外，重载时，首先考虑最匹配的类型，其次是向上转型，如 char -> int -> long -> float -> double，然后发生自动装箱，然后会匹配父类的类型，如 Serialize，或 Object 等。最弱的重载是可变长参数。

    动态分派(方法多态) - 查看字节码，可以找到 invokevirtural 指令，它会得到当前调用对象的实际类型，然后从该类及其父类寻找对应方法，如果找不到，则抛出 AbstractMethodError 异常。
    我们把这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

    Java 语言是一门静态多分派，动态单分派的语言。

    虚拟机的动态分派实现 - 使用了虚方法表的手段，为类在方区中建立了方法表，以优化方法的搜索。方法表一般在类加载的连接阶段进行初始化，准备了类的变量初始值后，虚拟机会把该类的方法表也初始化。(除了建立方法表，还可以使用内联缓存和守护内联两种非稳定的激进优化手段)

40. **动态类型语言：类型检查的过程在运行期完成。**
    Java 是静态类型语言，但是 JVM 希望支持动态类型语言的运行，而已有 4 种方法调用指令并没有办法做到，所以引入 Invokedynamic 和 java.lang.invoke 包，解决这个问题。

    MethodHandle 和反射的区别？
    两者都可以模拟方法的调用。
    1. 反射是模拟 Java 代码层次的方法调用，而 MethodHandle 是在模拟字节码层次的方法调用。反射不需要关心字节码实现，MethodHandle 可以指定实现指令。
    2. 反射是重量级的，MethodHandle 是轻量级的。MethodHandle 应该享有虚拟机的优化，如方法内联。
    3. 反射服务于 Java，MethodHandle 服务于所有 JVM 语言。

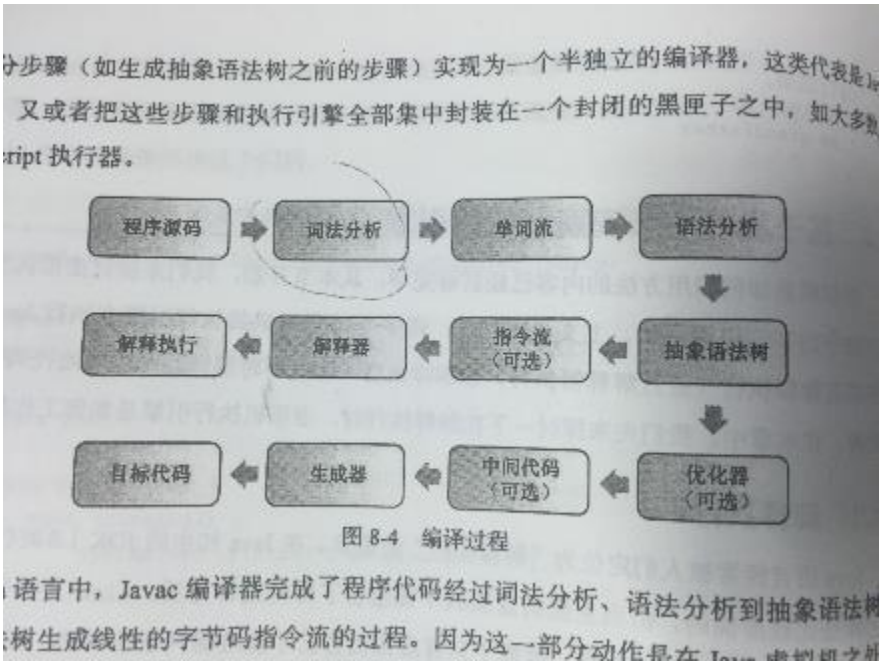
    invokedynamic 指令：
    使用该指令可以在程序中掌握分派的逻辑，比如可以在子类中调用祖父类中的指定方法。
41. Java 字节码的解释执行流程？
    

    图 8-4 编译过程
42. **基于栈的指令集**和**基于寄存器的指令集**的优缺点？
    栈的指令集优点是可移植，代码相对紧凑，编译器实现简单，但执行速度稍慢，完成相同功能所需指令数量会更多。
    寄存器容易受到硬件的约束，执行速度较快。

    Java 解释执行整体是基于栈的指令集，整个运算过程中间变量都以操作数栈的入栈和出栈作为信息交换途径。

43. Tomcat 类加载过程：

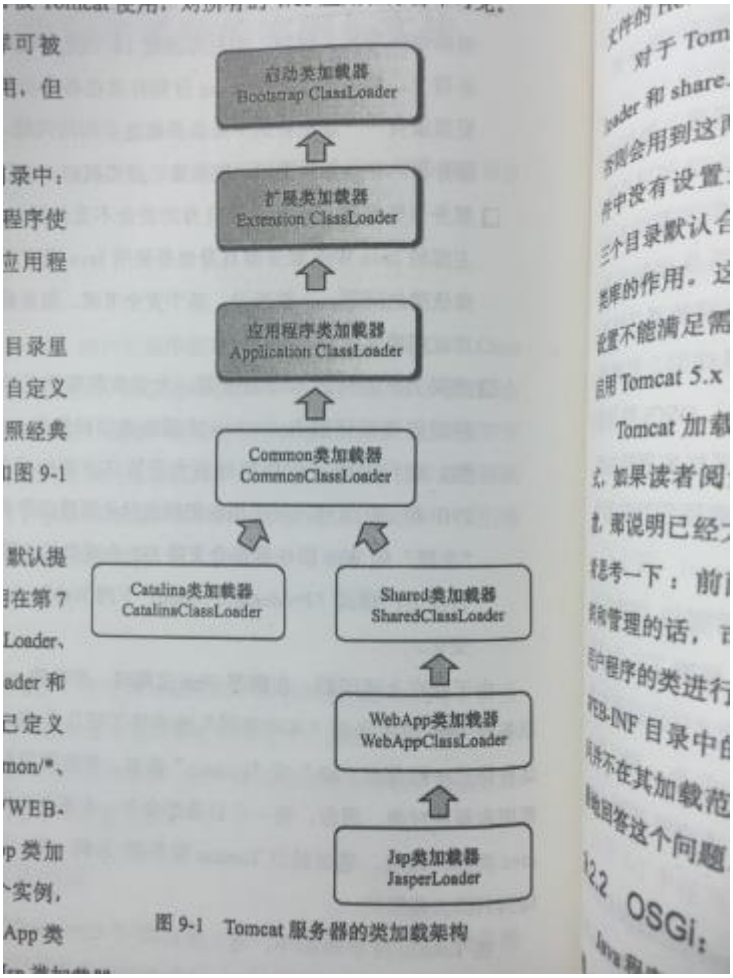


图 9-1 Tomcat 服务器的类加载架构

默认在 conf/catalina.properties 没有配置 server.loader 和 share.loader，所以默认这两个加载器和目录合并到 lib 目录下，简化了部署场景。

44. OSGi 可以实现模块级的热插拔功能，当程序升级更新或调试除错时，可以只停用、重新安装然后重启程序的其中一部分。OSGi 的 Bundle 类加载器之间只有规则，没有固定的委派关系。OSGi 的加载器模型已经不再是双亲委派模型，发展成了复杂的网状结构。**复杂的类加载模型，提供强大的功能，也带来了线程死锁和内存泄露的风险。**
45. **字节码生成技术和动态代理的实现。**
    动态代理是典型的运行时动态生成字节码的技术，Proxy.newInstance 就是根据传入的接口和类加载器，动态生成接口类的实现类，并继承 Proxy 类，最后将传入的 InvocationHandler 子类传入构造的新类。在创建的新类中，接口对应的方法会调用传入 InvocationHandler 的 invoke 方法，通过这种方式完成了动态代理的实现。具体代码生成使用的 sun.misc.ProxyGenerator 类的 generateProxyClass()方法。
46. JDK 逆向工程，使高版本 JDK 的代码兼容低版本 JDK 代码。对应工具为 Retrotranslator。
47. 实现远程执行功能。自定义类加载器，将 Byte[]变为 Class 对象。接着将 System 替换为 HackSystem 类，直接修改字节码文件。然后实现 ByteUtils 来类型转换，最后实现 HackSystem，重定向 System.out 到 ByteArrayOutputStream，就可以实现远程执行的功能。首先编写程序，然后将编译好的字节码传到服务器端，服务器端拿

到字节码，对字节码的 System 进行替换，成 HackSystem，这样实现了输出重定向，然后通过类加载器，将 Byte 数组变成真正的 Class 对象，再调用 main 方法，获得返回值后返回，再获得执行结果，响应请求即可。

48. 早期编译期优化，主要针对.java 编译成.class 的过程，Javac 对于代码运行的效率并没有很大改善，但针对程序编码来说增加了许多新功能。
49. Javac 的编译过程？

    解析与填充符号表过程。

    插入式注解处理器的注解处理过程。

    分析与字节码生成过程。
50. Javac 编译过程的主要 8 个处理步骤：

    准备过程 - 初始化插入式注解处理器

    执行注解处理(输入到符号表、词法分析、语法分析) - 将每个标记取出，构成**抽象语法树**，每个节点代表语法结构。完成了语法和词法的分析，接着就需要填充符号表，符号表的内容主要用于语义检查和产生中间代码。

    语义分析及字节码生成 - 根据上步骤得到的抽象语法树，进行源程序的上下文审查，如类型审查等。

    标注检查 - 这里很重要的一个过程是**常量折叠(a=1+2 => a=3)**。

    数据流分析 - 检查局部变量在使用前是否复制、方法每条路径是否有返回值、异常是否已被处理等。将局部变量声明为 final，对运行期是没有影响的，变量的不变性仅仅由编译器在编译期间保障。

    解语法糖 - JDK1.5 新特性在编译过后都变成了基础语法结构，只是为了程序员方便使用。

    生成字节码 - 不但需要把前面各个步骤生成的信息(语法树、符号表)转化成字节码写到磁盘中，编译器还进行少量代码增加和转换**(如 init 构造方法和 clinit 方法)**。另外，**此阶段还会将 String 加操作转换为 StringBuffer 或 StringBuilder 的 append()操作。**
51. 解语法糖的典型案例：

    泛型擦除。List<String>和 List<Integer>在运行时没有区别，所以不能以这两个参数不同来重载方法。此外，**擦除只是方法的 Code 属性进行了擦除，而元数据中还保留泛型信息，所以还是可以通过反射获取泛型。**

    自动装箱、拆箱和 Foreach。这些语法会在编译时期进行解语法糖，还原成基础语法。

    条件编译。有一些 if(true) else 的代码会直接省略，擦除这个分支语句。
52. 可以利用插入式注解处理器，在 javac 编译时对源代码的规范进行检查，可以自己定义插入式注解处理器。
53. 晚期运行期优化指在运行时，虚拟机会将代码编译成本地平台相关的机器码，并进行各层次的优化，完成这个任务的编译器称为即时编译器(JIT)。
54. 为什么 JVM 采用解释器和编译器并存的架构？**考虑内存限制和效率并存。**

    为了在程序启动响应速度和运行效率之间达到最佳平衡，HotSpot 虚拟机采用**分层编译**的策略。

    虚拟机在运行过程中，会将被多次调用的方法和多次执行的循环体认定为**热点代码**。判断代码是否是热点代码，这种行为称为**热点探测**。判定方法分为：**基于采样的热点探测(采样栈顶方法，简单高效，但是不准确，容易受干扰)和基于计数器的热点探测(比较麻烦，每个方法都需要计数器，但是准确)。HotSpot 使用后者。计**

数器分为方法调用计数器(定期半衰减)和回边计数器(统计循环体代码执行次数，计算调回的次数，可以触发 OSR 编译)。OSR 栈上替换

55. Client 编译器采用典型的三段式优化(HIR -> LIR -> 窥孔优化)，主要关注局部优化。

    Server 编译器则是充分优化的高级编译器，如**无用代码消除、循环展开、循环表达式外提、消除公共子表达式、常量传播、基本块重排序**等，还有一些**范围检查消除、空值检查消除**和一些激进优化，守护内联和分支频率预测等。

56. 即时编译器优化的四种代表技术：

    语言无关的**公共子表达式消除：运算时可能会做代数化简，提取公因式之类的。**

    语言相关的**数组范围检查消除：根据数据流判断数组不会越界，就可以在数组访问时去除边界检查。**

    方法内联：两个方法，不内联可能都有意义，但内联之后便可以更深层次的优化。（不能优化实例方法，只能静态方法，解决方案是类型继承关系分析，对代码进行激进优化，并留逃生门(守护内联))。

    逃逸分析：一个对象不会逃出方法外，可以在栈上分配对象空间。如果确定一个对象不会逃出线程，无法被其他线程访问，那么关于该对象的同步可以消除。

57. Java 与 C++的编译器对比？

    坏处：

    1. 即时编译器终究占用用户程序的运行时间，即时编译器也不敢引入大规模的优化技术。
    2. Java 语言是动态的类型安全语言，虚拟机必须频繁的动态检查。
    3. Java 多态的使用频率大于 C++，导致优化很困难。
    4. Java 的动态扩展性，导致很多全局优化都很难做。
    5. 手动回收内存，手动选择内存创建位置，效率比 GC 高。

    好处：

    1. 开发效率高，动态可扩展性高，垃圾回收。
    2. 可以使用运行期优化。

58. 硬件的效率和一致性：

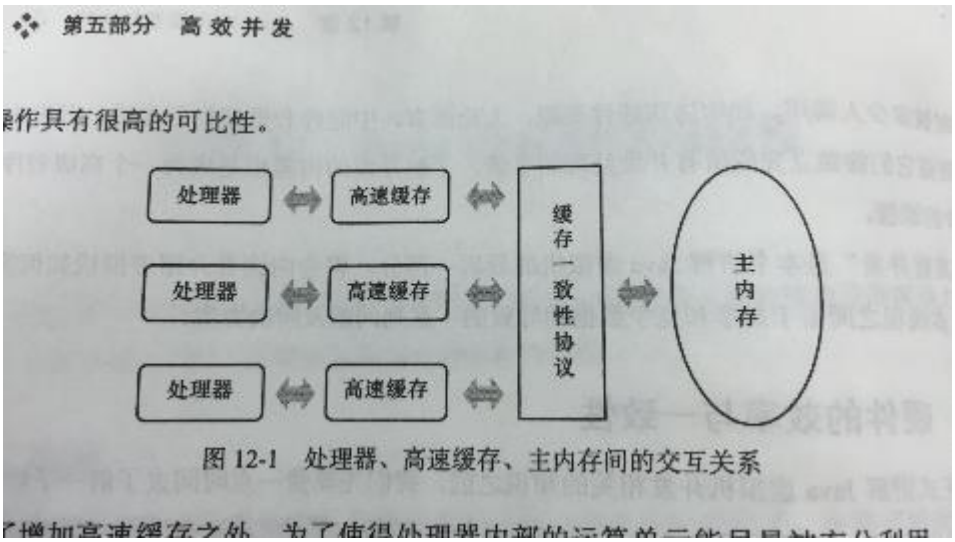


图 12-1 处理器、高速缓存、主内存间的交互关系

59. Java 内存模型：目的是使得各种平台下的 Java 程序都能达到一致内存访问的效果。

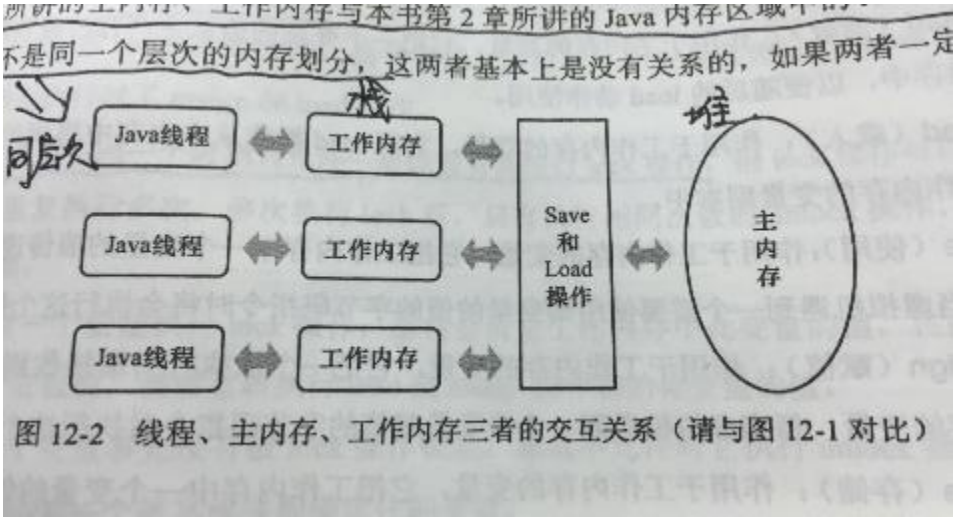
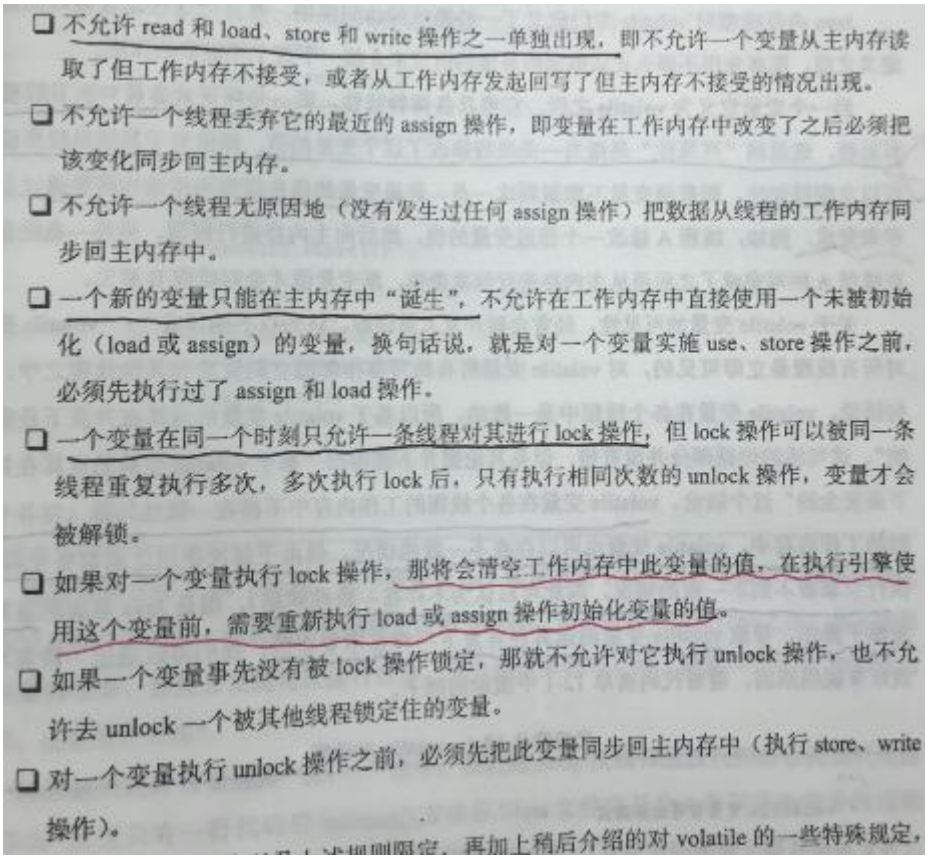


图 12-2　线程、主内存、工作内存三者的交互关系（请与图 12-1 对比）

60. Java 内存模型规定所有变量都存储在主内存中，每个线程还有自己的工作内存，线程工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作必须在工作内存中进行，而不能直接读写主内存的变量。**为了获得更好的性能，虚拟机会让工作内存优先存储在寄存器和高速缓存中，因为程序运行时主要访问读写的是工作内存。**
61. Java 内存模型定义了以下 8 中操作来完成主内存和工作内存的交互，虚拟机必须保证这 8 个操作是原子的：lock 锁定、unlock 解锁、read 读取、load 载入、use 使用、assign 赋值、store 存储、write 写入。


- 不允许 read 和 load、store 和 write 操作之一单独出现，即不允许一个变量从主内存读取了但工作内存不接受，或者从工作内存发起回写了但主内存不接受的情况出现。
- 不允许一个线程丢弃它的最近的 assign 操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。
- 不允许一个线程无原因地（没有发生过任何 assign 操作）把数据从线程的工作内存同步回主内存中。
- 一个新的变量只能在主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load 或 assign）的变量，换句话说，就是对一个变量实施 use、store 操作之前，必须先执行过了 assign 和 load 操作。
- 一个变量在同一个时刻只允许一条线程对其进行 lock 操作，但 lock 操作可以被同一条线程重复执行多次，多次执行 lock 后，只有执行相同次数的 unlock 操作，变量才会被解锁。
- 如果对一个变量执行 lock 操作，那将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行 load 或 assign 操作初始化变量的值。
- 如果一个变量事先没有被 lock 操作锁定，那就不允许对它执行 unlock 操作，也不允许去 unlock 一个被其他线程锁定住的变量。
- 对一个变量执行 unlock 操作之前，必须先把此变量同步回主内存中（执行 store、write 操作）。

62. volatile 关键字的特性？Java 提供个最轻量级的同步机制。它具备两种特性：**第一是保证此变量对所有线程的可见性(一个线程修改了这个变量的值，其他线程立即可知)**，而普通变量必须将修改的结果回写到主内存，其他线程才能通过主内存得到该值。

    volatile 变量在各个线程工作内存中不存在一致性问题，因为在每次改变并将 volatile 变量存入到内存时，会通知其他的缓存中的 volatile 变量失效，因此可以认为不存在不一致的问题，但是 Java 的运算并非原子操作，可能导致 volatile 变量的运算在并发下也不安全，如 i++;因此，volatile 变量仍要通过加锁来保证原子性。

    volatile 的第二语义是禁止指令重排序优化，普通的变量仅仅保证执行过程中所有依赖赋值结果的地方都能得到正确的结果，但不保证执行顺序和程序代码的顺序一致。

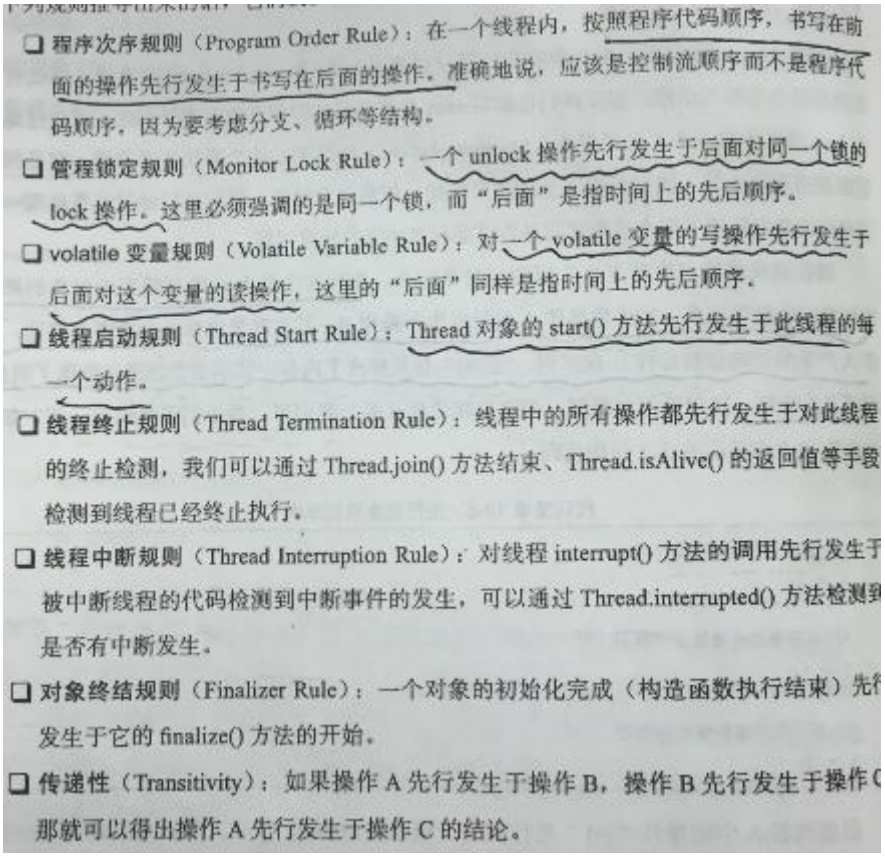
63. Java 内存模型是围绕着在并发过程中如何处理**原子性、可见性和有序性**这三个特征来建立的。

    原子性 - 由 Java 内存模型来直接保证的原子性变量操作包括 read、load、assign、use、store 和 write，我们大致可以认为基本数据类型的访问读写是具备原子性的。如果需要更大范围的原子性，可以使用 lock 和 unlock，对应字节码指令 monitorenter 和 monitorexit。

    可见性 - 指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。volatile 保证了新值能立即同步主内存，使用前立即从主内存刷新。除了 volatile，synchronized 和 final 也可以实现可见性：同步块在 unlock 前，会将更改同步到主内存中。final 变量在构造完成后，能被其他线程正确访问。

    有序性 - 如果本线程内观察，所有的操作都有序。如果一个线程观察另一个线程，所有操作无序。synchronized 和 volatile 可以禁止指令重排序。

64. Java 内存模型的先行发生的原则(happens-before)。



- 程序次序规则（Program Order Rule）：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说，应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构。
- 管程锁定规则（Monitor Lock Rule）：一个 unlock 操作先行发生于后面对同一个锁的 lock 操作。这里必须强调的是同一个锁，而“后面”是指时间上的先后顺序。
- volatile 变量规则（Volatile Variable Rule）：对一个 volatile 变量的写操作先行发生于后面对这个变量的读操作，这里的“后面”同样是指时间上的先后顺序。
- 线程启动规则（Thread Start Rule）：Thread 对象的 start() 方法先行发生于此线程的每一个动作。
- 线程终止规则（Thread Termination Rule）：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过 Thread.join() 方法结束、Thread.isAlive() 的返回值等手段检测到线程已经终止执行。
- 线程中断规则（Thread Interruption Rule）：对线程 interrupt() 方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过 Thread.interrupted() 方法检测到是否有中断发生。
- 对象终结规则（Finalizer Rule）：一个对象的初始化完成（构造函数执行结束）先行发生于它的 finalize() 方法的开始。
- 传递性（Transitivity）：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那就可以得出操作 A 先行发生于操作 C 的结论。

总而言之，时间先后顺序与先行发生原则之间基本没有太大的关系，所以我们衡量并发安全问题的时候不要受到时间顺序的干扰，一切必须以先行发生原则为准。

65. **线程是比进程更轻量级的调度执行单位，线程的引入，可以把一个进程的资源分配和执行调度分开，各个线程既可以共享进程资源(内存地址、文件 IO 等)，又可以独立调度(线程是 CPU 调度的基本单位)。**

66. **线程的实现可以分为三种方式：内核线程实现，使用用户线程实现和使用用户线程加轻量级进程的混合实现。**

内核线程(KLT) - 直接有操作系统内核支持的线程，这种线程有内核来完成线程切换，内核通过操纵调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。程序一般使用内核线程的高级接口 - 轻量级进程(LWP)来操作。**局限：每个线程操作都需要系统调用，代价较高，需要用户态和内核态来回切换；其次，每个轻量级进程都需要一个内核线程的支持，因此需要消耗一定的内核资源，数量有限。**

用户线程实现 - 狭义讲，完全建立在用户控件的线程库，用户线程的建立、同步、销毁和调度都在用户态完成，不需要内核的帮助。优点：操作快速和低消耗，可以支持规模更大的线程数量，部分高性能数据库中的多线程就是由用户线程实现的。劣势：没有系统内核的制约，所有线程操作都要用户程序来处理，使用用户线程实现的程序一般都比较复杂。

使用用户线程和轻量级进程混合实现 - 两者都存在，既可以低消耗，也能支持大规模用户线程的并发，而轻量级线程作为内核线程和用户线程之间的桥梁，大大降低了整个进程被完全阻塞的风险。

67. Java 线程调度的方式？

    协同式线程调度和**抢占式线程调度**。协同式的好处是实现简单，坏处是线程执行时间不可控制，如果一个线程编写有问题，一直不告知系统进行线程切换，那么程序会阻塞在哪里。

    抢占式调度的多线程系统，每个线程由系统来分配执行时间，线程的切换不由线程本身决定，Java 的线程调度方式就是抢占式调度。可以通过线程优先级来让某个线程多执行，但是线程调度取决于操作系统，但 Java 线程的优先级不一定和操作系统的优先级一一对应。另外，优先级可能会被系统自动改变，如果操作系统发现一个线程执行的很勤奋努力，则会为他多分配时间。

68. Java 线程的几种状态？5 种状态：

    新建 - 创建后尚未启动的线程处于这种状态。

    运行 - Runnable 包括操作系统线程状态中的 Running 和 Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待 CPU 为他分配执行时间。

    无限期等待 - 处于这种状态的线程不会被分配 CPU 执行时间，他们要等待被其他线程显示的唤醒。没有设置 Timeout 的 Object.wait 和没有设置 TimeOut 的 Thread.join()和 LockSupport.park()可以导致无限期等待。

    有限期等待 - 处于这种状态的线程也不会被分配 CPU 执行时间，不过不需要等待被其他线程显式地唤醒，在一定时间内它们会由系统自动唤醒。Thread.sleep()方法，设置时间的 Object.wait()和 Thread.join 方法，LockSupport.parkNanos 和 LockSupport.parkUntil 方法。

    阻塞 - 线程被阻塞了，阻塞和等待的区别是阻塞状态是等待获取到一个排他锁，这个事件将在另一个线程放弃这个锁的时候发生；而等待状态则是在等待一段时间，或者唤醒的动作发生时触发。在程序等待进入同步区域的时候，线程会进入这种状态。

    结束 - 已终止线程的线程状态，线程已经结束执行。

69. 什么是线程安全？当多个线程同时访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替运行，也不需要考虑进行额外的同步，或者在调用方法进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那么这个对象是线程安全的。

70. Java 语言中各种操作共享的数据分为：不可变、绝对线程安全、相对线程安全、线程兼容和线程对立：

    不可变 - 由于 String 声明了 final，可以保证值不可变，而它的方法都是返回一个新的字符串对象，所以是线程安全的。**除了 String,还有枚举类型，Number 类的部分子类，BigInteger 和 BigDecimal。AtomicInteger 和 AtomicLong 不是不可变的。**

    绝对线程安全 - Vector 类，虽然 add、get、size 都是 synchronized 的，但是如果不在方法调用端做额外的同步，代码仍然不安全，会出现这边删除了元素，那边抛出数组越界。

    相对线程安全 - 可以保证这个对象单独操作时安全的，我们再调用时不需要做额外的保护措施，但对于特定顺序的连续调用，就需要在调用端使用额外的同步手段来保证调用的正确性。Vector、HashTable、Collections.synchronizedCollection()都是这种类型。

    线程兼容 - 本身不是线程安全的，但是可以通过调用端正确的使用同步手段来保证对象在并发环境中可以安全地使用，相当于平常说的不是线程安全的。如

ArrayList 和 HashMap。

线程对立 - 无论采取什么样的同步措施，都没有办法在并发的情况下使用。如 Thread.suspend 和 Thread.resume。System.setIn 和 System.setOut 和 System.runFinalizersOnExit();

71. 线程安全的实现方法？

互斥同步 - 同步是指在多个线程并发访问共享数据时，保证共享数据在同一时刻只被一个线程使用。临界区、互斥量和信号量都是主要的互斥实现方式。Java 中使用 Synchronized 关键字。

注意点：Synchronized 是可重入的。每次阻塞和唤醒线程，都需要操作系统的介入，用户态和内核态的转换很耗费时间，所以它是一个重量级的锁。（虚拟机的优化：在阻塞线程之前加一段自旋等待的过程，避免频繁切入内核态）

另外，可以使用重入锁来实现同步。synchronized 和 ReentrantLock 的区别？

ReentrantLock 增加了等待可中断、公平锁和锁绑定 Condition 条件。

非阻塞同步 - 互斥同步最主要的问题就是进行线程阻塞和唤醒所带来的性能问题，因此这种同步模式也成为阻塞同步。而非阻塞同步是一种基于冲突检测的乐观并发策略，基本思想：先进性操作，如果没有其他线程征用共享数据，则操作成功，如果有争用，则采取补偿措施(不断重试，直到成功)。这种方式需要硬件指令集的支持：

这类指令常用有：

测试并设置 TestandSet

获取并增加 FetchandIncrement

交换 Swap

比较并交换 CompareandSwap 下文称 CAS

加载链接/条件存储 Load-Linked/StoreConditional

无同步方案 -

可重入代码，如果一个方法，返回结果可预测，只要输入了相同的数据，就能返回相同的结果，那就满足可冲入性的要求。

线程本地存储(ThreadLocal)，一个请求对应一个线程的时候，可以通过使用线程本地存储，实现一个线程之间的数据共享。每个 Thread 对象中都有一个 ThreadLocalMap 对象，这个对象存储了一组以 ThreadLocal.threadLocalHashCode 为键，本地线程变量为值的 K-V 值对，ThreadLocal 就是 Map 的访问入口，每一个 ThreadLocal 对象都包含了一个独一无二的 threadLocalHashCode 值，使用这个值就能找到对应的本地线程变量。

72. CAS 操作使用 sun.misc.Unsafe 类，最终编译出一条平台相关的 CAS 指令。Unsafe 类只能由类加载器加载，如果不采用反射，只能通过其他的 API 间接使用它，如 atomic 包。

如果在方法中使用自增，可以通过原子 Integer 类实现并发的自增，不需要添加额外的同步。

实质上，CAS 操作就是无限循环，直到 CAS 操作成功。CAS 操作的漏洞是 ABA 问题，就是两个线程分别增加和减少了值，第三个线程拿到这个值，并不知道其他线程已经修改了。

73. 锁优化：适应性自旋、锁消除、锁粗化、轻量级锁、偏向锁。

自旋：自旋默认次数 10 次，用于循环等待线程是否马上可以得到执行权限。自适应自旋锁可以自动调整循环次数。

锁消除：根据逃逸分析，消除不可能产生竞争的锁。

锁粗化：合并频繁互斥同步操作，减少不必要性能损失。

轻量级锁：首先创建一个轻量锁，然后利用 CAS 操作将对象头信息中的锁指针指向生成的锁，如果失败，检测是否已经指向当前线程的堆栈，如果不是，则锁对象已经被其他线程抢占了，那么此时就要对**锁进行膨胀，转换为重量级锁。解锁也通过 CAS 来操作。如果整个同步过程没有出现竞争，则不需要互斥，否则，会比重量级锁慢一个 CAS 操作。CAS 实质是在同步过程没有竞争时使用 CAS 操作进行优化**

偏向锁：在无竞争的情况下把整个同步都消除掉。Mark Word 操作和轻量锁很像，不同的是如果 CAS 操作成功，则持有偏向锁的线程每次进入这个锁相关的同步块时，都不再进行任何同步操作。当有另外一个线程尝试获取这个锁时，偏向模式宣告结束，根据锁对象目前是否处于锁定状态，撤销偏向后恢复到未锁定或轻量锁定状态，然后使用轻量锁的方式来处理竞争关系。