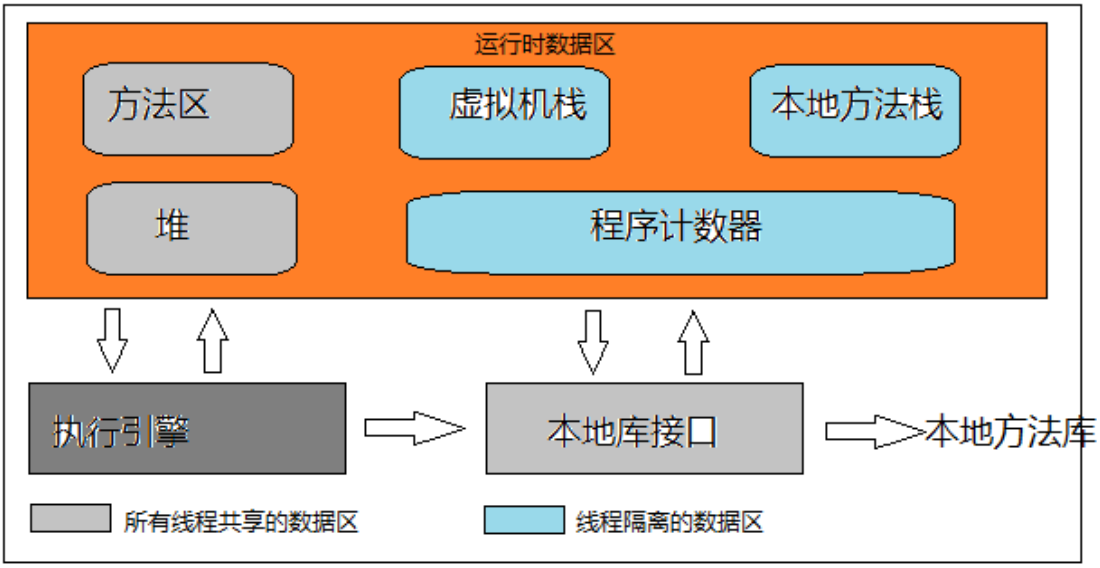
目录

Java 虚拟机

内存区域

Java 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以下若干个区域，以便于进行管理。



程序计数器

当前线程所之星的字节码的行号指示器。
如果线程正在执行的是 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果执行的是 Native 方法，则这个计数器为空（undefined）。此区域是 Java 虚拟机唯

一没有规定任何 OutOfMemoryError 的区域。
每个线程都有独立的程序计数器，各个线程互不影响，是“线程私有”的内存。

Java 虚拟机栈

线程私有的内存，生命周期与线程相同。描述的是 Java 方法执行的内存模型：每个方法在执行时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。
存放了编译期可知的各种基本数据类型、对象引用。long 和 double 占用两个局部变量空间（slot），其余的占用 1 个。
抛出两个异常：
StackOverflowError：线程请求的栈深度大于虚拟机允许的深度时，则抛出
OutOfMemoryError：虚拟机栈动态扩展时，如果无法申请到内存，则抛出

本地方法栈

类似于虚拟机栈，虚拟机执行 Native 方法时使用。

Java 堆

所有线程共享的内存，在虚拟机启动时创建。所有的对象实例、数组都需要在堆上分配。
Java 堆细分为：新生代、老年代。
再细分：Eden 空间、From Survivor 空间、To Survivor 空间等
堆内存大小由参数：-Xmx -Xms 控制

方法区

所有线程共享的内存。存储被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等。
也即：永久代（JDK1.8 没了）
参数：-XX:MaxPermSize 控制方法区大小

运行时常量池

是方法区的一部分。存放编译期生成的各种字面量、符号引用

直接内存

不是虚拟机运行时的数据区域，也不是虚拟机规范中定义的内存区域。如 :DirectByteBuffer。

大小不受 Java 堆大小的限制，受到本机内存限制。

对象

对象创建

1. 遇到 new 指令，检查常量池中是否可以找到这个类的符号引用，并且检查这个类是否已被加载、解析、初始化过。
2. 为新生对象分配内存（虚拟机使用 CAS 的方式保证指针操作的原子性）
   指针碰撞：Serial、ParNew
   空闲列表：CMS
   TLAB（ThreadLocal Allocation Buffer）：先在线程的栈上分配内存，当内存不够时，需要分配新的 TLAB，此时需要同步锁定。参数：-XX:+/-UseTLAB
3. 将分配的内存初始化为零
4. 对对象进行必要的设置。如：对象属于哪个类的实例、类的元信息、GC 分代年龄等（保存在对象头中）
5. 执行 init（构造函数初始化）

内存布局

对象在内存中的布局包括：对象头、实例数据、对齐填充
对象头：

- 存储对象自身的运行时数据：哈希码、GC 分代年龄、锁状态标志、线程持有的锁、偏向线程 id、偏向时间戳。
- 指针，对象指向它的类元数据的指针

实例数据：

- 程序中定义的字段信息

对齐填充：占位符。HotSport VM 要求对象的初始地址为 8 字节的整数倍，即：对象大小必须是 8 字节的整数倍。对象头正好是 8 字节。

对象访问

方式：

- 句柄（通过句柄池间接访问对象数据）
- 指针

垃圾收集器和内存分配策略

3 个问题？

1. 哪些需要回收
2. 什么时候回收
3. 怎么回收

引用计数

每个对象被引用 1 次之后，对象头中的计数加 1，取消引用之后，计数减 1。

可达性分析算法

通过 GC Roots 对象作为起始点，遍历子树，当一个对象到 GC Roots 没有任何引用链时，证明此对象不可用，可以被回收。

可以作为 GC Roots 的对象：

- 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（Native 方法）引用的对象

是否覆写 finalize 方法作为判断对象是否被再次编辑的判断。

回收方法区：废弃的常量和无用的类。

判断类是否回收：

1. 该类的所有实例被回收
2. 加载该类的 ClassLoader 已被回收
3. 该类的 java.lang.Class 对象不再被引用

垃圾回收算法

标记清除

标记出所有需要回收的对象，标记完成后，统一回收。

问题：
1. 效率问题，标记和清除过程的效率都不高
2. 空间问题，产生大量不连续的内存碎片

复制算法

将内存分为大小相等的两块，每次只使用其中的一块。当该块用完了，就将还存活的对象复制到另外一块上，然后把已使用过的内存空间一次清理掉。

类似于 Eden：Survivor0 : Survivor1 = 8:1:1

当 Survivor1 空间不足时，需要老年代进行担保。

一般适用于年轻代的垃圾回收算法，因为年轻代的对象 98%都是朝生夕死的。

标记整理算法

老年代的垃圾回收算法。

1. 标记所有的待回收对象
2. 把所有的存活对象移动到另一端

HotSpot 算法实现

枚举根节点

使用 OopMap 的数据结构来知道在哪里存放了对象引用。

安全点

在 OopMap 的帮助下，可以快速完成 GCRoots 的枚举。
但是并不是为每条指令都生成 OopMap，仅在程序执行时，到达安全点才能暂停。
一般只有**方法调用**、**循环跳转**、**异常跳转**等功能的指令才会产生 safepoint。

如何到达 safepoint？
线程执行时，主动轮训 safepoint 标志位，发现中断标志为真时，就自己中断挂起，轮训标志位的地方和 safepoint 是重合的。

安全区域

当线程处于 sleep、blocked 状态时，需要安全区域来解决回收问题。
安全区域是：在一段代码片段中，引用关系不会发生变化，在这个区域胡总的任意位置开始 GC 都是安全的。

垃圾收集器

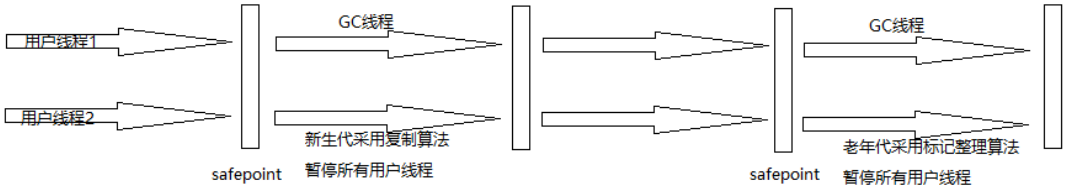
Serial 收集器

单线程收集器，只会使用 1 个 CPU 或 1 个收集线程去完成垃圾回收操作。

client 模式的默认收集器

ParNew 收集器

Serial 收集器的多线程版本。新生代收集器
参数：
-XX:SurvivorRatio、-XX:PretenureSizeThreshold、-XX:HandlePremotionFailure



只能与 CMS 配合工作
参数：-XX:UseParNewGC
参数：-XX:ParallelGCThreads 参数限制垃圾收集的线程数

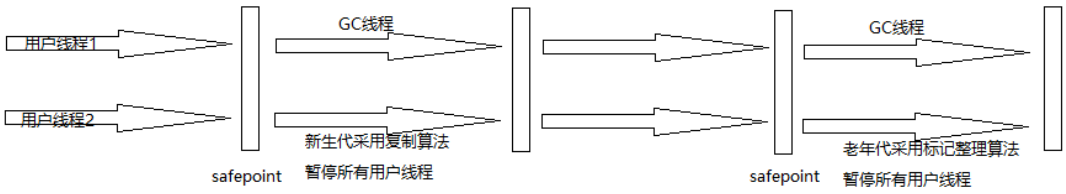
Parallel Scavenge 收集器

也是使用复制算法，新生代收集器
这是一个与吞吐量相关的收集器。

参数：-XX:MaxGCPauseMillis 停顿时间
参数：-XX:GCTimeRatio 控制吞吐量（默认: 99v）

Serial Old 收集器

单线程、老年代收集器。



Parallel Old 收集器

是 Parallel Scavenge 收集器的老年代版本。使用多线程和标记整理算法。

CMS 收集器

以获取最短回收停顿为目标的收集器。
基于**标记清除算法**。
过程：

- 初始标记（Stop The World）
- 并发标记
- 重新标记（Stop The World）
- 并发清除

参数：
-XX:*CMSInitiatingOccupancyFraction*=N（默认 70，垃圾占堆空间 70%时，开始收集）
-XX:UseCmsInitiatingOccupancyOnly（默认 false，开启之后，只看老年代的空间占用）
-XX:*ConcGCThreads*=N（CMS 的 GC 线程数）

缺点：

- 对 CPU 资源敏感
- 无法处理浮动垃圾
- 产生内存碎片

G1 收集器

调优目标：避免发生并发模式失败或疏散失败
特点：
- 并行与并发：充分利用多核，利用并发的方式让 Java 线程继续执行
- 分代收集：不需要其它垃圾收集器就能管理整个堆
- 空间整合：基于标记整理算法，不会产生内存碎片
- 可预测的停顿

参数：-XX:MaxGCPauseMills=N（200ms，如果 stw 超过 0.2s
G1 中每个分区的大小计算：1 << log(初始堆大小/2048)（Min=1MB,Max=32MB）
参数：-XX:G1HeapRegionSize=N（分区大小）
调整方法：
1. 新生代和老年代比例
2. 调整堆大小
3. 更早的启用后台处理（-XX:ConcGCThreads=N CPU 充足时考虑）
4. 调整 G1 垃圾收集器的运行频率（-XX:InitiatingHeapOccupancyPrecent=N（45%））
5. 改变晋升阀值或者在混合式垃圾收集周期中处理更多或更少的老年代分区
   参数：-XX:G1MixedGCLiveThresholdPercent=N（分区的垃圾比例）
   参数：-XX:G1MixedGCCountTarget=N（8，混合式 GC 周期数，减小时解决晋升失败问题，增大时通过减少每次 GC 的工作量，解决 GC 停顿时间过长问题，）
   参数：-XX:MaxGCPauseMills（GC 停顿的可忍受最长时间）

步骤：
- 初始标记（Stop The World）
- 并发标记
- 最终标记
- 筛选回收

高级调优

对象移动到老年代的条件：
1. Survivor 空间太小，新生代垃圾收集时，如果目标 Survivor 空间被填满，则直接进入老年代
2. 对象在 Survivor 空间中经历的 GC 周期数有上限

Survivor 空间大小
初始：-XX:InitialSurvivorRatio=N（8）
最大：-XX:MaxSurvivorRatio=N（3）

Survivor 空间调整：-XX:TargetSurvivorRatio=N（50，Survivor 调整之后，保证 50%可用）
对象需要在 Survivor 经历多少个 GC 周期：
-XX:InitialTenuringThreshold=N（PS、G1=7，CMS=6）
-XX:MaxTenuringThreshold=N（PS、G1=15，CMS=6）

特殊参数：
-XX:+AlwaysTenure（false，对象直接在老年代）
-XX:+NeverTenure（false，JVM 会认为初始和最大的晋升阀值无限大。即：只要 Survivor 空间有容量，对象就不会晋升到老年代）

-XX:+PrintTenuringDistribution（false，在 GC 日志中看到晋升信息）

TLAB

ThreadLocalAllocateBuffer
受决定的因素：应用线程数、Eden 空间的大小、线程的分配率
受益的程序：需要分配大量巨型对象的程序、需要创建大量线程的程序
参数：
-XX:-UseTLAB（默认开启，加此参数则关闭）
-XX:+PrintTLAB（监控 TLAB 分配情况）
-XX:TLABSize=N（0，动态调整）
-XX:-ResizeTLAB（默认开启，避免每次 GC 时都调整该值）
-XX:TLABWasteTargetPercent=N（决定应该在堆上分配还是回收当前的 TLAB）
-XX:MinTLABSize=N（2KB,最大小于 1G）

GC 日志

参考：理解 CMS GC 日志

5.505: [GC (GCLocker Initiated GC) 7.694: 5.505: [ParNew: 50403K->31946K(1887488K), 0.0135559 secs] 1188549K->1170178K(3984640K), 0.0137616 secs] [Times: user=0.19 sys=0.00, real=0.01 secs]
时间戳:[GC(原因) 发生时间:[GC算法:回收前垃圾大小->回收后垃圾大小(总垃圾大小),耗时] 回收前堆大小->回收后堆大小(总堆大小),耗时][Times: user=0 sys=0 real=0]

[GC (Allocation Failure) 2017-10-11T11:23:41.118+0800: 1615948.929: [ParNew: 1698351K->21547K(1887488K), 2.9402005 secs] 3122030K->1446553K(3984640K), 2.9404803 secs] [Times: user=3.52 sys=0.19, real=2.94 secs]

"Allocation Failure" is a cause of GC cycle to kick.
"Allocation Failure" means that no more space left in Eden to allocate object. So, it is normal cause of young GC.
Older JVM were not printing GC cause for minor GC cycles.
"Allocation Failure" is almost only possible cause for minor GC. Another reason for minor GC to kick could be CMS remark phase (if +XX:+ScavengeBeforeRemark is enabled).

内存分配和回收策略

对象优先在 Eden 分配，当 Eden 空间不足时，发生 Minor GC。

参数：-Xms20M -Xmx20M -Xmn10M
Java 堆大小为 20M，10M 分配给新生代，剩下 10M 给老年代。
-XX:SurvivorRatio=8
大对象直接在老年代分配
参数：-XXPretenureSizeThreshold，当对象大于这个值时，直接在老年代分配。
避免 Eden 和 Survivor 区之间的内存复制。

长期存活的对象将进入老年代
每个对象都有个年龄的计数器。
对象在 Survivor 每熬过 1 次 MinorGC，年龄就增加 1 岁，当年龄到一定成都（默认 15）时，晋升到老年代。
由参数：-XX:MaxTenuringThreshold 设置。

空间分配担保

在发生 Minor 之前，虚拟机会先检查老年代最大可用连续空间是否大于新生代所有对象总空间。如果成立，则 MinorGC 确保安全。

参数：-XX:HandlePromotionFailure=false 控制是否允许担保失败

故障处理

JVM 相关工具

JVM Tools

工具名称	作用
jps	jvm process status tool　显示知道系统内所有的虚拟机进程
jstat	jvm statistics monitoring tool　手机虚拟机运行时数据
jinfo	configuration info for java　显示虚拟机配置信息
jmap	memory map for java　生成虚拟机内存转储快照（heapdump 文件）
jhat	jvm heap dump browser　用于分析 heapdump 文件
jstack	stack trace for java　显示虚拟机的线程快照
jcmd	console command　控制台操作

jps -mlvV 显示虚拟机启动参数

jstat -gc pid 1s 10

jmap -clstats pid class 信息
jmap -histo pid 对象信息
jmap -dump:live,format=b,file=heap.bin <pid>
jcmd -pid help 查看所有支持的命令
jcmd -pic GC.run 执行一次 full gc

GC 相关参数

```
-XX:ErrorFile=/apps/logs/osp/hs_err_%p.log
-XX:+HeapDumpOnOutOfMemoryError 发生 OOM 时，创建堆转储文件
-XX:HeapDumpPath=/apps/logs/osp/
-XX:OnOutOfMemoryError=$CATALINA_HOME/restart.sh 发生 OOM 后的操作
-XX:+AlwaysPreTouch
-XX:AutoBoxCacheMax=20000
-XX:CMSInitiatingOccupancyFraction=75 老年代堆空间的使用率,第一次 CMS 垃圾收集会在老年代被占用 75%时被触发.
-XX:+ExplicitGCInvokesConcurrent 用来替代 DisableExplicitGC,这两个参数只能配合 CMS 使用（-XX:+UseConcMarkSweepGC），而且 System.gc()还是会触发 GC 的，只不过不是触发一个 完全 stop-the-world 的 full GC，而是一次并发 GC 周期
-XX:+DisableExplicitGC 禁用 System.gc()
-XX:+ManagementServerJMX
-XX:InitialTenuringThreshold=4
-XX:MaxTenuringThreshold=4 垃圾最大年龄
-XX:InitialHeapSize=4294967296
-XX:MaxDirectMemorySize=2147483648
-XX:MaxHeapSize=4294967296
-XX:MaxMetaspaceSize=536870912
-XX:MetaspaceSize=536870912
-XX:NewRatio=1 新生代:老年代=1:1,年轻代占整个堆的 1/2
-XX:SurvivorRatio=8 Survivor:Eden=2:8
-XX:OldPLABSize=16
-XX:-OmitStackTraceInFastThrow 强制要求 JVM 始终抛出含堆栈的异常
-XX:+ParallelRefProcEnabled 使用多线程处理对象引用
-XX:+PerfDisableSharedMem 禁止 JVM 写 statistics 数据,代价是 VisualVM 和 jstat 用不了,只能用 JMX 取数据(原来每次进入安全点（比如 GC,JVM 都会默默的在/tmp/hperf 目录写上一点 statistics 数据,如果刚好遇到 PageCache 刷盘,把文件阻塞了,就不能结束这个 Stop the World 的安全点了)
-XX:ReservedCodeCacheSize=251658240 用于设置 Code Cache 大小,JIT 编译的代码都放在 Code Cache 中,若 Code Cache 空间不足则 JIT 无法继续编译,并且会去优化,比如编译执行改为解释执行,由此,性能会降低.(240M)
```

```
-XX:-TieredCompilation 不使用多层编译
-XX:-UseBiasedLocking 不使用偏向锁
-XX:+UseCMSInitiatingOccupancyOnly 使用手动定义初始化定义开始 CMS 收集,禁止 hostspot 自行触发 CMS GC
-XX:+UseCompressedClassPointers 开启 class 指针压缩
-XX:+UseCompressedOops 压缩普通对象指针
-XX:-UseCounterDecay 禁止 JIT 调用计数器衰减
-XX:+UseConcMarkSweepGC 老年代 GC 算法为 CMS
-XX:+UseParNewGC 年轻代 GC 算法为 ParNew 算法
-XX:+PrintCommandLineFlags
-XX:+PrintGC
-XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCDateStamps
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
```

类文件结构

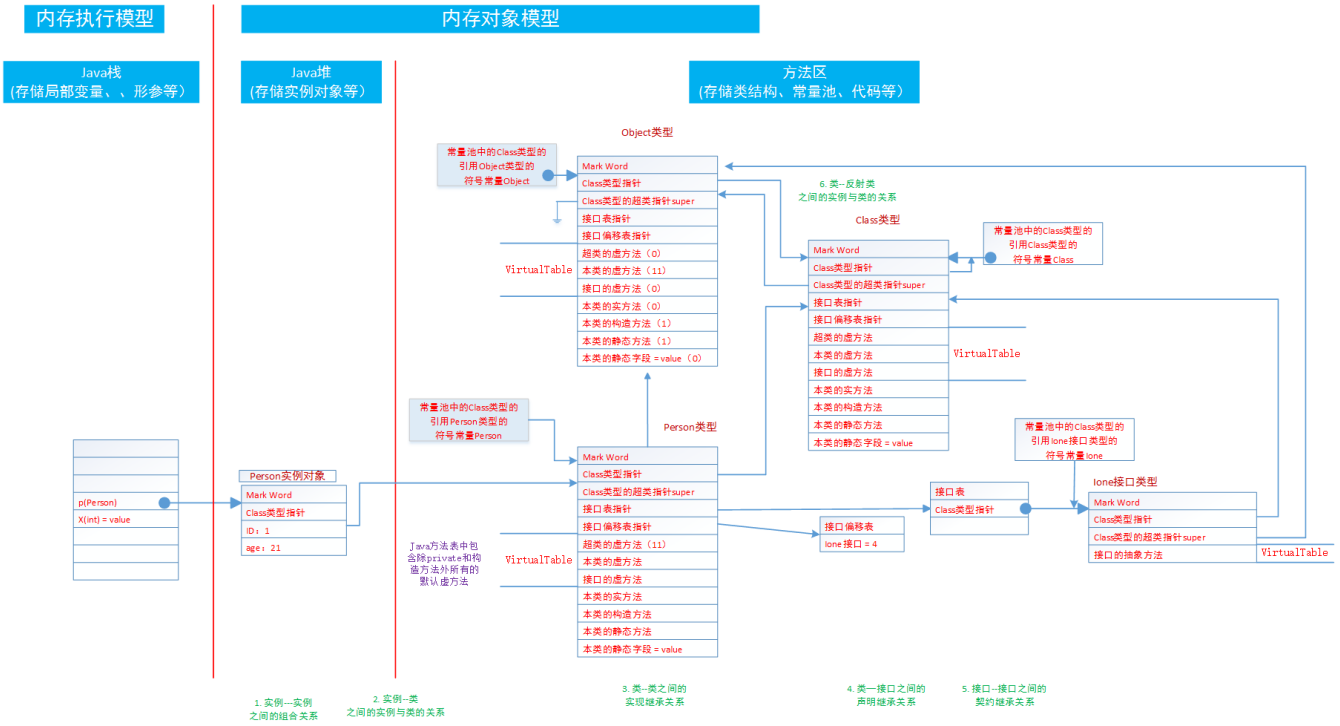
Class 文件是以 8 位字节为基础单位的二进制流。

4 字节魔数：版本号
常量池：计数+info
访问标志：识别 Class 是类或接口，是否 public，是否 abstract，是否 final 等
类索引、父类索引、接口索引集合：
字段表集合
方法表集合
属性表集合

方法调用指令：
invokevirtual：实例方法
invokeinterface：接口方法
invokespecial：初始化、私有方法、父类方法
invokestatic：类方法
invokedynamic：动态解析的方法，如：lambda

同步指令：
方法级别同步和方法内同步，都使用同步结构：Monitor 实现
synchronized 关键字通过两条指令：monitorenter、monitorexit 来实现

对象内存模型



虚拟机类加载机制

生命周期

加载-> 校验 -> 准备 -> 解析 -> 初始化 -> 使用 -> 卸载

必须初始化的情况：

1. new（new Object）、getstatic（读取、设置静态字段）、putstatic、invokestatic（调用静态方法） 4 条指令
2. 使用 java.lang.reflect 包的方法进行反射时
3. 初始化一个类时，如果它的父类没有初始化，则首先初始化它的父类
4. 虚拟机启动时，要执行的主类（main 方法）
5. 动态语言

加载

1. 通过类的全限定名来获取定义此类的二进制字节流（文件、网络等方式）

2. 将字节流所代表的静态存储结构转化为方法区内的运行时数据结构
3. 在内存中生成代表这个类的 Class 对象，作为方法区这个类的各种数据访问入口

验证

1. 文件格式验证（保证字节流被正确解析，保存在方法区中）
2. 元数据验证（对字节码描述的信息进行语义分析）
3. 字节码验证（通过数据流和控制流分析，确定程序语义时合法的、符合逻辑的。并且对类的方法体进行验证分析，保证方法不危害虚拟机）
4. 符号引用验证（验证类名、对象的方法和字段）

准备

正式为**类变量（static 修饰的）**分配内存并设置初始化值的阶段，变量使用的内存都在方法区分配。
如果被 final 修饰，则直接显式初始化，不会默认初始化。

解析

将常量池内的符号引用替换为直接引用的过程。
符号引用：任意形式的字面量
直接引用：可以直接指向目标的指针、相对偏移量、间接定位到目标的句柄。

主要针对：类或接口、字段、类方法、接口方法、方法类型、方法句柄、调用点限定符。

初始化

1. 父类静态变量
2. 父类静态代码块
3. 子类静态变量
4. 子类静态代码块
5. 父类属性
6. 父类代码块
7. 父类构造函数
8. 子类属性

9. 子类代码块
10. 子类构造函数

类加载器

分类

1. BootStrap ClassLoader 使用 C++实现，虚拟机自身的一部分。
   加载 <JAVA_HOME>/lib 路径或 -Xbootclasspath 指定的路径
2. java.lang.ClassLoader 使用 Java 实现

双亲委派

当一个类收到了类加载的请求，它首先委托给父类的加载器去完成，当父类的加载器无法完成时，子加载器才会尝试自己去加载。

好处：天然具备带有优先级的层次关系。

自定义的类加载器一般只需要实现 **findClass** 方法即可

执行引擎

Java 虚拟机的解释执行称为：基于栈的执行引擎。

如何找到正确的方法？如何执行方法内的字节码？

优化

早期（编译期）优化

Javac 编译过程：

1. 解析与填充符号表（词法、语法分析 生成抽象语法树）
2. 插入式注解处理器的注解处理过程（读取、修改、添加抽象语法树中的元素）

3. 语义分析与字节码生成过程
    a. 标注检查
    b. 数据以及控制流分析
    c. 解语法糖
    d. 字节码生成

晚期（运行期）优化

jit 编译

热点代码
基于计数器的热点探测方法，为每个方法都准备了两类计数器：方法调用计数器和回边计数器。

参数：-XX:CompileThreshold 设置调用次数

栈上分配
同步消除
标量替换

Java 内存模型与线程

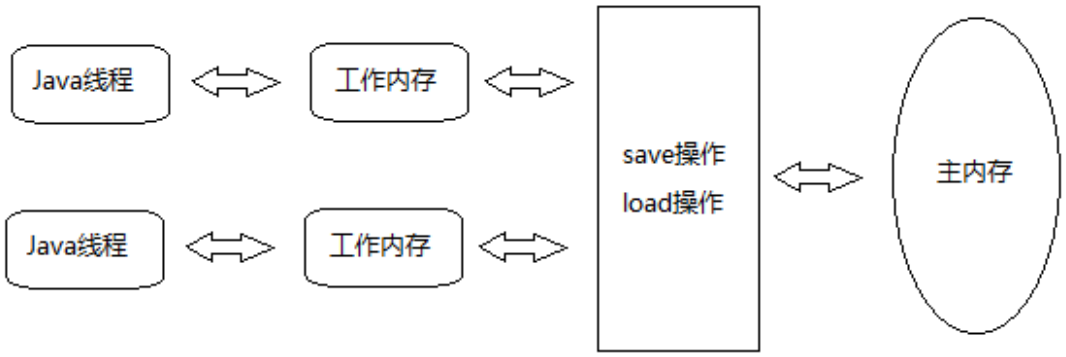
内存模型

Java 内存模型：定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量。

Java 内存模型规定：所有变量都存储在主内存中，每条线程有自己的工作内存，线程的工作内存保存了该线程使用的变量的主内存副本拷贝，线程对该变量的所有操作（读取、写入）都必须在工作内存中进行，而不能直接读写主内存中的变量。

本地内存是 JMM 的一个抽象概念，并不真实存在。它涵盖了缓存、写缓冲区、寄存器以及其它硬件和编译器优化。

内存模型如图：



一个变量如何在主内存和工作内存中存、取？
虚拟机需要保证以下操作是原子的

- lock 作用于主内存的变量，标识一个变量为线程独占的状态
- unlock 锁释放之后，其它线程才可以锁定
- read 作用于主内存，将主内存中变量传输到工作内存中
- load 作用于工作内存，将 read 传输的变量保存在工作内存的变量副本中
- use 工作内存，把变量传递给执行引擎
- assign 工作内存，将执行引擎返回的值赋给工作内存中的变量
- store 作用于工作内存，将工作内存中的变量传输到主内存
- write 将 store 传输的变量，放入主内存

3 个规则：
原子性、可见性、有序性

重排序：

1. 编译器优化重排序 编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序
2. 指令级重排序 处理器可以改变语句对应机器指令的执行顺序
3. 内存系统重排序

过程：
源代码 -> 编译器优化重排序 -> 指令级并行重排序 -> 内存系统重排序 -> 最终指令

为了保证内存可见性，Java 编译器在生成指令序列的适当位置插入内存屏障来禁止特定类型的处理器重排序。

对 volatile 关键字做了特殊处理
1. 保证可见性
2. 屏蔽指令重排序

遵循 as-if-serial 语义
无论怎么重排序，单线程程序的执行结果不能被改变。

顺序一致性时内存模型的一个理论参考。

规则：happens-before
1. 程序顺序规则：线程中的任意操作，happens-before 于该线程的任意后续操作
2. 监视器锁规则：对一个锁的解锁，happens-before 于后续任意对这个锁的加锁
3. volatile 变量规则：对变量的写，happens-before 于后续对这个变量的读
4. 传递性规则：a happens-before b, b happens-before c 则 a happens-before c
5. start 规则
6. join 规则

线程

Java 的线程与操作系统中的用户线程是 1:1 关系。

线程调度交由操作系统完成。

线程状态：

new、running、waiting、timed waiting、blocked、terminated

锁优化

互斥、同步时，挂机和恢复线程都需要进入内核态，由内核完成。

适应性自旋锁

参数：-XX:+UseSpinning 启用自旋锁
参数：-XX:+PerBlockSpin 自旋次数（默认 10）

自旋不能替代阻塞，自旋虽然避免了线程切换，但是会浪费 CPU

锁消除

由 JIT 编译来确定。比如：方法内部使用 StringBuffer 类

锁粗化

由虚拟机自动确定

轻量级锁

对象头中的数据
1. 用于存储对象自身的运行时数据：哈希码、GC 分代年龄等
2. 存储指向方法区对象类型数据的指针。

对象头：

存储内容	标志位	状态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定
指向重量级锁的指针	10	重量级锁定
空	11	GC 标识
偏向线程 ID、时间戳、对象分代年龄	01	可偏向

1. 在当前线程的栈帧中建立锁记录（Lock Record）存储锁对象的 mark word 拷贝
2. 使用 cas 尝试将对象的 mark word 更新为指向 Lock Record 的指针，成功则是轻量级
3. 膨胀为重量级

偏向锁

参数：-XX:+UseBiasedLocking

当锁对象第一次被线程获取时，虚拟机会把对象头中的标志位设为“01”，即偏向模式，同时使用 CAS 操作把线程 id 记录在对象的 mark word 中，成功则表示偏向锁获取成功，之后这个线程再次进入时，不需要加锁。