# 走进java

## 1.5 展望Java技术的未来

* 无语言倾向：Graal VM，全栈虚拟机；
* 新一代即时编译器：Graal 即时编译器（C1，C2）

# JAVA内存区域与内存溢出异常

## 2.1 概述

Java与C++语言的不同是，java的内存管理是通过虚拟机完成的，所以不容易发生内存泄露，如果一旦发生，就要在了解虚拟机的基础上查找发生的问题。

## 2.2 运行时数据区

Java管理底层内存，不需要程序员去操作；JAVA虚拟机运行时数据区划分如下：



程序计数器，当前线程执行的字节码的行号指示器，为了保证线程切换，互不影响，每个线程一个独立的程序计数器，如果执行的是一个Native方法，则计数器为空，此区域没有规定OutOfMemoryError。

虚拟机栈也是线程私有，生命周期与线程相同，存储java方法的调用链形成的内存模型，执行方法的调用代表这入栈与出栈的过程，方法调用栈也叫作栈帧；栈里面的主要是函数使用到的内容，比如局部变量表，函数出口地址，操作数，对象引用等；局部变量表通过变量槽的方式存储局部变量，变量槽的大小在编译期就可以确定了；因为一个方法使用到的变量声明，在编译期就确定了；线程请求的栈深度大于虚拟机所允许的深度，抛出StackOverflowError异常，栈也可以扩展，如果没有内存可以扩展了，抛出OutOfMemoryError异常。

本地方法栈与虚拟机栈类似，只不过调用的方法是Native方法。

堆是所有线程共享的，在虚拟机启动时创建，存放对象实例与数组，堆也叫做GC堆，是垃圾收集器主要管理的区域，可以是物理不连续的空间，逻辑上是连续的，如果无法在堆上分配内存对象，抛出OutofMemoryError异常。

方法区也是线程共享的，存储虚拟机加载的类信息、常量、静态常量、即时编译器编译后的代码，是作为堆的一个部分（因为也是共享的），在HotSpot上称为永久代，此区域也是纳入内存管理的，常量池回收以及类型的卸载等会导致内存回收。方法区的无法满足内存分配需求时报OutofMemoryError异常；

运行时常量池是方法区的一部分，Class文件除了类的版本、字段、方法、接口等还有常量池定义，存放编译器生成的各种字面量与符号引用（变量名）；类加载后这些内容进入方法区的**运行时**常量池中存放，运行期间，这个常量池也可以加入内容，并非一定是编译期，常量池无法申请到内存时报OutofMemoryError异常；

直接内存不是虚拟机运行时数据区，是在java虚拟机之外的本机内存，不受java堆控制。

## 2.3 HotSpot虚拟机对象探秘

虚拟机遇到new指令时，检查指令的参数能否在常量池中定位一个类的符号引用，并检查这个类是否已加载，解析与初始化过，类加载后，为对象分配内存，对象所需的内存大小在类加载完后就确定了，分配对象的内存的方法有：指针碰撞（连续内存用一个指针做分界）与空闲列表（不连续内存使用标记的方式）2种方式，主要是看Java堆是不是连续的决定，并发情况下，这种分配对象是不安全的，需要进行同步处理，或者使用CAS重试处理，或者是内存的分配在线程划分在不同的空间内进行，每个线程预先在堆中占据一块内存，称为本地线程分配缓冲（TLAB），分配完成后，虚拟机将内存都初始化为0（不包括对象头）；保证不赋初值就可以使用；接下来，写入对象头信息，包括对象是哪个类的实例，类的元数据信息，对象的Hash码，对象的GC分代年龄等；之后，会调用对象的构造函数（Class对象的<init>()方法）进行初始化。

对象在内存中存储的布局可以分为：对象头、实例数据与对齐填充3块。对象头包含2部分数据：对象的运行时数据与类型指针；运行时数据包括哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等；运行时数据为一个字长度，比如32bit或者64bit；bit格式不固定如果对象未被锁定的话，32bit中25bit存hash码、4bit存储对象的分代年龄、2bit存锁标志位、1bit固定为0；类型指针确定对象是哪个类的；如果对象是数组，对象头还会存数组的长度；

实例数据是对象真正的内容，包含父类的；按照一定的顺序排列：longs/double、ints、.....oops，相同宽度的字段被分到一起，父类定义的变量会出现在子类前面。

对齐部分，占位符，虚拟机规定对象的大小是8字节的整数倍，当实例数据不足8的整数倍时，补位凑齐。

Java程序需要通过栈上的reference（引用或者指针）来操作堆上的对象，如何定位对象有2种方式：句柄（中间层模式），java堆划分出一块区域作为句柄池，句柄中包含对象实例数据的地址与类型数据的地址信息，reference是句柄的地址；直接指针访问（直接模式），refernce直接存储堆内的对象地址，句柄是当对象移动时，reference值不变，直接指针，refernece要变，但是直接指针的方式更加快。

## 2.4 实战：OutOfMemoryError异常

### 2.4.1 Java堆溢出

在Eclipse IDE或者java命令行上加入参数

-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+HeapDumpOnOutOfMemoryError；

Java对的大小为20M，最小为Xms=20M，最大Xmx=20M，会禁止虚拟机拓展堆大小。

只要创建的对象足够多，就会产生堆内存溢出异常。



使用Eclipse Memory Analyzer可以对转储的Heap文件进行分析，可以查看是因为什么内存溢出，是因为内存泄露还是内存不足。

### 2.4.2 虚拟机栈与本地方法栈溢出

HotSpot不区分虚拟机栈与本地方法栈，-Xss设置栈大小，栈会爆出2个异常，StackoverflowError与OutmemoryError2种错误，要注意被爆出的触发条件。

### 2.4.3 方法区与运行时常量池溢出

运行时常量池溢出需要注意的是，在java 8以后放到了堆中，而不是永久代，方法区溢出使用CGLIB技术，不断加入新的Class。

### 2.4.4 本机直接内存溢出

DirectMemory使用-XX:MaxDirectMemorySize指定，如果不指定，跟堆的大小是一样的，使用Unsafe抛出内存溢出异常。

# 垃圾收集器与内存分配策略

## 3.1概述

垃圾收集器（Grabage Collection，GC）：那些内存需要回收，什么时候回收，如何回收；当发生内存泄漏、内存溢出、高并发瓶颈出现时需要考虑垃圾收集内容，一般情况下不涉及；程序计数器、栈等内存分配回收等，不需要考虑，是作用域相关的编译期就确定的，方法区与堆是全局相关的，是在运行期才能确定的内存内容，主要就是考虑这里的内存分配与回收。

## 3.2对象已死吗？

堆中存在这所有的对象，回收前，需要确定那些还或者，那些不会在被使用到；

### 3.2.1 引用计数算法

给对象中添加一个引用计数器，如果多了一个引用就+1，少了就-1；简单直接，很多语言的实现方案，这种方法没法解决，2个对象相互引用的问题，他们都需要回收了，但是引用计数不为0，都得不到回收。Java虚拟机并不是用这种方法判断对象是否可以回收的。

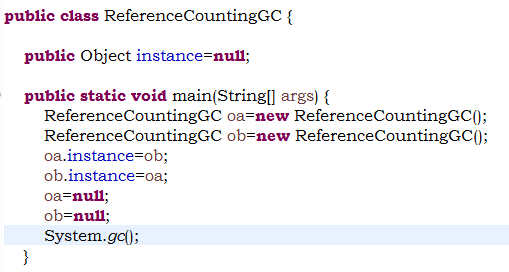


图3-1 相互引用的GC情况代码

### 3.2.2 可达性分析算法

判断对象存活，通过一些GC roots的对象作为起始点，向下搜索，搜索走过的路径为引用链，一个对象没有到GC roots的引用链，那么对象是不可用的。GC roots对象包括：虚拟机栈中引用的对象、方法区类静态属性引用的对象、方法区常量引用的对象、本地方法栈引用的对象、虚拟机内部引用、所有被Sychronized持有的对象、或者其他的堆的区域对象，因为具体的虚拟机把堆分为了不同区域，每次回收回收部分区域。

### 3.2.3 再谈引用

Java中引用的定义：reference数据中，存储的数值代表的是一块内存的起始地址，就称为引用；JVM后续对引用的状态进行了扩充：

强引用：GC roots引用链上的所有对象；

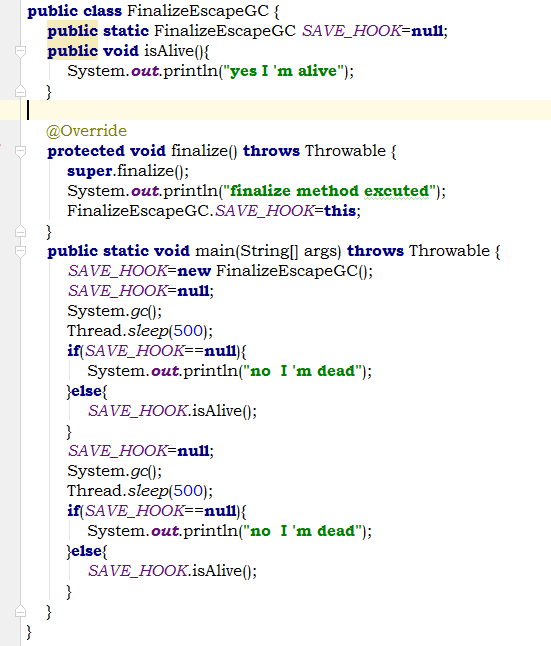
软引用：有用但是非必须的对象，在内存溢出前会回收，如果内存够，则保留；

弱引用：非必须对象，下一次垃圾回收器运行，一定回收的；

虚引用：不知道啥意思。

### 3.2.4 生存还是死亡

即使对象不能被引用到，也并非不能挽救回来；覆盖了finalize方法（没有执行过）的对象会经过2次标记，才会真正回收；如果对象不可达后，先判断对象的finalize()方法，如果没有被覆盖，或者已经执行过一次，那么直接收回该对象内存，如果不是这样，这个对象会被放入一个F-queue队列中，等待执行finalize方法，一个低优先级的Finalizer线程会取队列的对象来执行finalize方法，如果此时对象重新与引用链上连接上的话，gc在下一次执行时会将其移出回收集合，如果执行完finalize还没有，那么对象被回收。可以拿到的代码如下：



Finalize方法只会执行一次，

### 3.2.5 回收方法区

方法区回收的内容：废弃常量和无用的类；常量池中存在没有被引用的常量，那么就回收；类回收条件：所有类的实例被回收，加载该类的ClassLoader被回收，该类对应的Class对象没有在任何地方被引用，是否对类回收，虚拟机提供了参数控制。

## 3.3 垃圾收集算法

垃圾算法：实现哪些内存需要被回收、什么时候回收、怎么回收的整体算法；具体的虚拟机与实现方式是不同的，主要的思想就是分代思想。

### 3.3.1 分代收集理论

弱分代：绝大多数对象都是朝生夕灭的；

强分代：熬过多次收集的对象就越难以消亡；

根据这个理论，可以把判断对象的生命周期类型并划分到不同的区域里面，然后不同的区域运行不同的收集策略；一般的虚拟机管理堆内存都会把堆分为新生代与老年代。Minor GC：目标只是新生代的垃圾收集；Major GC：目标只是老年代的垃圾收集；Mixed GC：多个区域的垃圾收集，Full GC：堆。

### 3.3.2 标记-清除算法

最基础的核心算法，分为2个阶段：1.标记哪些内存要回收，2.清除；效率与回收的对象的数量成正比，对象越多，处理越慢；会导致内存碎片，无法分配大对象时会触发有一次的内存回收。

### 3.3.3 标记-复制算法

解决内存分片问题，内存分为2块，每次只在一块内存分配内存，收集时将活的对象复制到另一块，清除原来那一块里面的对象，然后再清理另一块，复制活的对象到原本的那块，就是一半作为备份空间使用；不会有内存碎片产生，每次都可以进行连续的内存分配，只是会浪费一半的内存，是新生代内存回收使用的方法。优化的策略是Eden-Survivor-Survivor方法；一块Survivor用来做备份空间，使用Eden与一个Survisor；备份空间占用降低了。

### 3.3.4 标记-整理算法

老年代的回收算法，让所有对象都向一端移动，然后清除掉边界意外的内存，前面是复制（存活的对象少），这个是移动（存活的对象多，不想浪费空间进行复制），移动对象需要虚拟机停止。移动对象回收时麻烦，不移动对象由于内存碎片，再次分配内存时麻烦。

## 3.4 HotSpot的算法实现

HotSpot在实现垃圾收集时，也要对执行效率有考虑。

### 3.4.1 枚举根节点

根据GC roots引用链来逐个检查确定对象的有效性，是低效的，特别是当跟节点对象（GC roots）很多的情况下，本身确定GC roots的内容就是很大的难题，因为GC roots的内容变化很大，所以所有的收集器在确定根节点的对象时都需要虚拟机暂停（stop the world），所以确定GC roots的过程必须在内存的一个快照上进行，就是不能变化，所以GC进行时，所有的线程将会停顿；HotSpot不会逐个检查所有区域内的对象是否是根节点还是基本类型的数据什么的，因为栈上的内存无法区分是对象地址还是数据，因为在类加载完成的时候，HotSpot把类内的数据对象做一个标记，标记了对象内的数据的类型信息与偏移地址等，这些标记组成了类的OoPMap的数据结构。GC扫描时可以直接得知这些信息，从而获取GC roots（参考文章：https://blog.csdn.net/woaigaolaoshi/article/details/51439227）。

### 3.4.2 安全点（可以简单理解为方法调用，JVM会把高聚合的块作为一个整体执行，类似与事务）

HotSpot并不会在每个指令处处都记录OopMap的数据，因为每个指令都可能导致栈内存变化，这样的空间消耗比较巨大，因而只是在特定的位置记录OopMap，这个位置就是安全点，GC只有在碰到安全点时才会触发GC roots枚举，使所有线程跑到最近的安全点停顿下来，有2种方案：抢先式中断：GC发生时，先中断所有线程，如果有不在安全点的，就让它执行直到到达安全点；主动式中断：GC发生时，设置一个所有线程中断标志，线程在执行时轮询这个标志，发现设置了，就自己中断挂起。

### 3.4.3 安全区域

安全点的扩充，由点到面，在这块代码区域中，栈内不会发生对象的引用关系的变化，所以任意地方开始GC roots都是可以的。

### 3.4.4 记忆集与卡表

记忆集是一种从非数据收集区域指向数据收集区域的指针集合的抽象数据结构，卡表是记忆集的具体实现，卡表有自己的记忆粒度控制。

### 3.4.5 写屏障

### 3.4.6 并发的可达性分析

可达性分析（追踪式垃圾收集）要求在进行分析停顿用户线程；带来的问题是堆的对象越多，停顿的时间就越多；如果不停顿，就会有并发扫描的问题；有的对象扫描完了，并发程序有引用了新的对象，这个对象并没有被标记成可达的（因为扫描完了），这种对象小时的问题，需要同时满足2个条件：

* 赋值器插入了一条或者多条从黑色对象到白色对象的新引用；
* 赋值器删除了全部从灰色对象到该白色对象的直接或者间接引用；

解决的办法就是分别破坏上面的条件：

增量更新：记录分析时的黑色对象新增加的引用的黑色对象，然后再分析一遍这些对象；

原始快照：把删除了白色对象引用的灰色对象记录下来，然后再分析一遍。

这些都是通过写屏障实现的

## 3.5 经典垃圾收集器

收集算法是方法论，垃圾收集器是实践者；Java对垃圾收集器如何实现没有做规定，各个厂商自己实现。

### 3.5.1 Serial收集器

历史最久，单线程，工作时，必须暂停掉所有线程。这个收集器是虚拟机运行在Client模式下的默认新生代收集器，一般来说桌面应用分配的内存都不会很大，停顿事件就会较少，优点：简单高效,额外内存消耗比较少，没有线程交互开销。新生代采用标记复制算法，老生代采用标记整理算法。

### 3.5.2 ParNew收集器

Serial收集器的多线程版本，各种参数几乎与Serial收集器一样。是运行的server模式下的首选的新生代收集器；能与老生代收集器CMS配合使用；在单CPU环境下，ParNew收集器不会比Serial收集器更好，CPU数量越多，ParNew收集器性能越好。

### 3.5.3 Parallel Scavenge收集器

新生代、复制算法、并行的多线程收集器；Parallel Scavenge收集器并不致力与缩短停顿时间，而是提高吞吐量，即用户代码运行时间/总运行时间；停顿时间越短越需要与用户交互多的程序，越长适合与后台计算任务。

### 3.5.4 Serial Old收集器

Serial收集器的老生代版本，差不多。

### 3.5.5 Parallel Old收集器

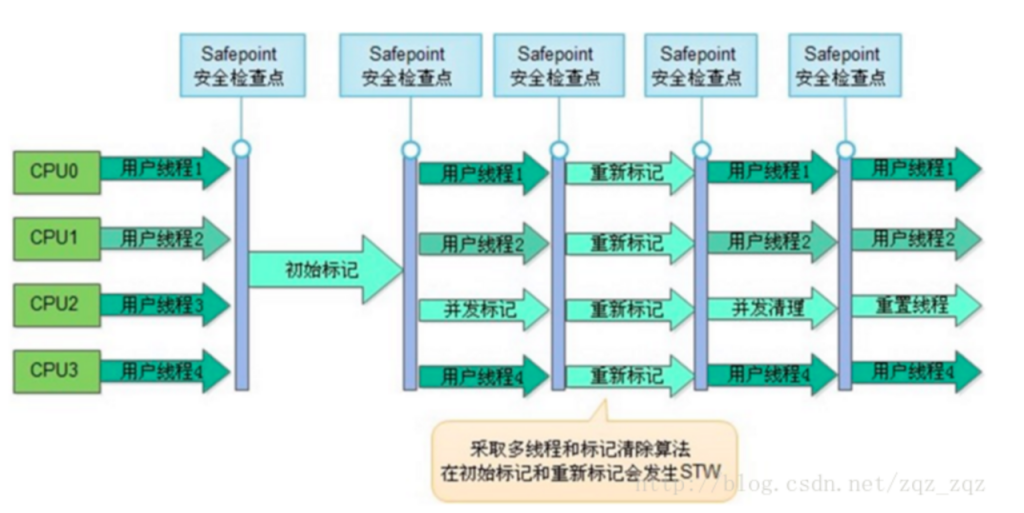
Parallel Scavenge收集的老生代版本。

### 3.5.6 CMS（Concurrent Mark Sweep）收集器

致力于最小的停顿时间，适合网站服务端；基于标记-清除算法实现，运作过程：

* + 初始标记（GC roots 需要STW）
  + 并发标记（查找链，并发，不需要STW）
  + 重新标记（第二轮次标记，上面的运行过程中发生了引用更改的重新标记）
  + 并发清除

运行示意图：



CMS收集器的缺点：

* CMS收集器因为是与用户程序并发的，抢占CPU资源，所以对CPU资源非常敏感；
* CMS无法处理浮动垃圾（回收期间产生的垃圾对象），需要预留空间做期间的对象分配， 目前是老年代的内存占用超过92%就会触发GC，如果过剩余的内存不够分配大对象，就会发生Concurrent Mode Failure，导致STW的Full GC产生，就会冻结用户线程，启用Serial模式进行Full GC；
* 内存碎片造成无法分配大对象，产生Failure，也会导致Full GC；

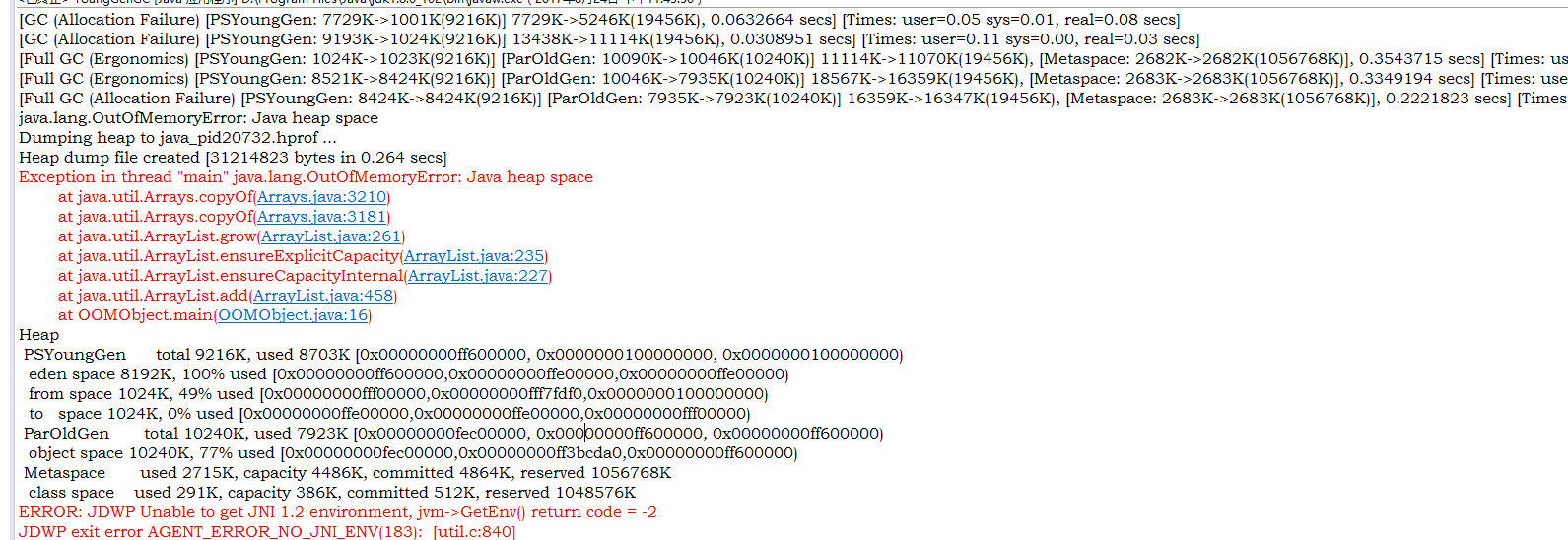
### 3.5.7 G1（Garbage First）收集器

G1收集器开创了局部收集的设计思路；经过不断的实验，最终成为可以商用的收集器；G1是面向服务端的垃圾收集器；取代了CMS收集器，并且之后设计出了统一收集器接口完成，收集的定义与具体实现分离；G1使用region作为堆的布局；有点像是页式内存管理；G1收集器运作过程：

* 初始标记：标记GC roots；
* 并发标记：遍历对象图；
* 最终标记：并发问题处理；
* 筛选回收：统计region区域的回收价值与成本，结合JVM设定的停顿事件来计算回收的region的集合，清空，并把存活的对象移动到空的region；

### 3.5.8 理解GC日志

每一种收集器的日志形式都由它们自身决定；如下一段：



最前面的数字代表哦GC发生的事件，一般是虚拟机启动以来的秒数。[GC与[Full GC说明这次垃圾收集的停顿类型，Full说明发生了全部停顿。调用System.gc()会导致Full；[ParOldGen与[PSYoungGen等标识GC发生的区域，与GC收集器密切相关；[ParOldGen说明老生代使用的ParNew收集器；[PSYoungGen说明新生代使用的是Parallel Scavenge收集器；7729K->1001K(9216K) ，7729表示回收前的使用容量，1001K表示回收后的使用容量，9216K表示这个区域的总容量； [PSYoungGen: 7729K->1001K(9216K)] 7729K->5246K(19456K), 0.0632664 secs]；7729K->5246K(19456K)这个方括号之外的代表堆的容量信息；0.0632664标识内存GC所占用的时间。

### 3.5.9 垃圾收集器参数总结

一些设置虚拟机垃圾收集的相关参数。

## 3.6 内存分配与回收策略

Java的自动内存管理解决了：给对象分配内存与回收内存2件事；大多数情况下，对象在堆上的Eden区分配内存，大对象直接进入老年代；长期存活的对象将直接进入老年代。

# 虚拟机性能监控与故障处理工具

知识经验是关键基础，数据是依据，工具是运用知识处理数据的手段。

## 4.2 JDK的命令行工具

/bin目录下，sun提供了很多的java工具，这些工具多种多样，非常小，里面主要是调用/lib/tools.jar里面的类来实现功能，linux下这些工具是由shell写成的，用于监控虚拟机与故障处理的主要工具有：

Jps:JVM Process Status Toll 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；

Jmap:Memory Map for java 生成虚拟机的内存转储快照文件；

Jhat:JVM Heap Dump Browser 用于分析heapdump文件，可以用浏览器查看；

Jstack:Stack Trace for Java 显示虚拟机的线程快照；

Jps命令与Linux下的ps命令类似，列出正在运行的虚拟机进程，并显示本地虚拟机唯一ID --LVVMID也就是PID；并显示虚拟机执行的主类；支持一些参数。

## 4.3 JDK的可视化工具

# 第六章 类文件结构

虚拟机上的语言允许可以将代码编译成平台无关的格式。JVM不只是平台无关的，也是语言无关的，JVM可以运行JAVA、Scala、Groovy等其他几种语言的程序。

## 6.2 无关性的基石

越来越多的语言选择了与操作系统和机器指令集无关的平台中立的格式作为程序编译后的存储格式，这种格式就是字节码Class文件，其他语言编写的程序都可以编译成字节码文件拿到JVM上执行；通常字节码文件的策程序特性要比Java还要多很多，java语言并不能完全表现出字节码所表示的程序结构。

## 6.3 Class类文件结构

任何一个Class文件都对应着唯一一个类或者接口；Class文件是8字节倍数大小的二进制流，数据项目紧密排列在Class文件中，没有分隔符；整体结构类似于C语言中的结构体；结构体中只有2种数据类型：无符号数和表。

无符号数是基本类型，u1、u2、u4、u8等表示，分别代表1字节...8字节；无符号数描述数字、索引引用、数量值、或者是UTF-8编码构成的字符串值。表是由多个无符号数与其他表组成，所有表都已\_info结尾。Class文件就是一张表，用于描述层次关系的符合结构数据。

### 6.3.1 魔数与Class文件的版本

每个Class文件的头4个字节是魔数，这个数的作用是确定Class文件是否是一个能被JVM运行的Class文件；通常为0XCAFFBABE；再4个字节是Class文件的版本号。5-6是次版本号，7-8是主版本号。

### 6.3.2 常量池

挨着版本号的是常量池，资源仓库，占用空间很大，通常是表类型的；由于常量池的数据量不固定，所以开头处有u2类型2个字节标识常量池计数值；从1开始计数，0表示不引用任何一个常量池项目的含义，也包含在内，但是是有特殊用途的一个索引；主要存储2大类：字面量（java语法上的常量）与符号引用（编译层面，类与接口的全限定名、字段的名称与描述符、方法的名称与描述符）；Class文件编译后并不能在程序获得方法字段等真实内存信息，这些都是在JVM加载Class文件时动态链入的，与传统的C语言的在编译阶段就链接程序不同，所以只有在运行期才能获得真正的内存地址。

常量池中每一项都是一个表的结构，表开始的第一个u1字节都是一个标志位，代表当前的常量是哪种常量类型

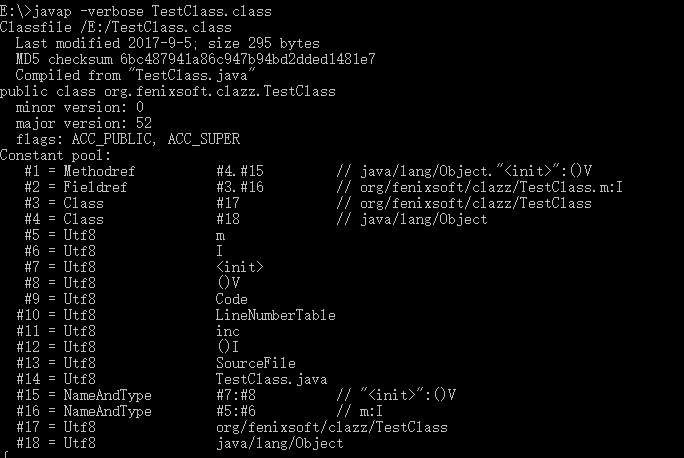
|  |  |  |
| --- | --- | --- |
| 类型 | 标志 | 描述 |
| CONSTANT\_utf8\_info | 1 | UTF-8编码的字符串 |
| CONSTANT\_Integer\_info | 3 | 整形字面量 |
| CONSTANT\_Float\_info | 4 | 浮点型字面量 |
| CONSTANT\_Long\_info | 5 | 长整型字面量 |
| CONSTANT\_Double\_info | 6 | 双精度浮点型字面量 |
| CONSTANT\_Class\_info | 7 | 类或接口的符号引用 |
| CONSTANT\_String\_info | 8 | 字符串类型字面量 |
| CONSTANT\_Fieldref\_info | 9 | 字段的符号引用 |
| CONSTANT\_Methodref\_info | 10 | 类中方法的符号引用 |
| CONSTANT\_InterfaceMethodref\_info | 11 | 接口中方法的符号引用 |
| CONSTANT\_NameAndType\_info | 12 | 字段或方法的符号引用 |
| CONSTANT\_MothodType\_info | 16 | 标志方法类型 |
| CONSTANT\_MethodHandle\_info | 15 | 表示方法句柄 |
| CONSTANT\_InvokeDynamic\_info | 18 | 表示一个动态方法调用点 |

每种常量类型都有自己的数据结构。



常量池容量为0x0013也就是19个。

Class文件中方法字段等都需要CONSTANT\_utf8\_info的常量来存储符号引用，所以名字的最大长度就是CONSTANT\_utf8\_info的长度决定，也就是65535个字符；javap命令可以自动分析Class文件字节码内容。

yuhandaod

### 6.3.3 访问标志

常量池之后是2个字节的类或者接口层次的访问信息。

|  |  |  |  |
| --- | --- | --- | --- |
| 标志名 | 标志值 | 标志含义 | 针对的对像 |
| ACC\_PUBLIC | 0x0001 | public类型 | 所有类型 |
| ACC\_FINAL | 0x0010 | final类型 | 类 |
| ACC\_SUPER | 0x0020 | 使用新的invokespecial语义 | 类和接口 |
| ACC\_INTERFACE | 0x0200 | 接口类型 | 接口 |
| ACC\_ABSTRACT | 0x0400 | 抽象类型 | 类和接口 |
| ACC\_SYNTHETIC | 0x1000 | 该类不由用户代码生成 | 所有类型 |
| ACC\_ANNOTATION | 0x2000 | 注解类型 | 注解 |

### 6.3.4 类索引、父类索引、与接口集合

在访问标志位之后。

### 6.3.5 字段表集合

### 6.3.6 方法表集合

### 6.3.7 属性表集合

## 6.4 字节码指令简介

Java虚拟机的指令由一个字节长度的操作码与后面多个字节的操作数的参数组成；所以操作码不会超过256个，操作数没有长度对齐，数据在指令里是无格式存储的，也没有边界符。

操作码（指令）本身就包含了所操作数据的类型信息，比如iload，在局部表中加载int类型数据到操作数栈；

# 虚拟机类加载机制

Yuhan捣蛋