目录

[专有名词解释： 1](#_Toc11513968)

[一 . JVM 内存结构 1](#_Toc11513969)

[1.作用 1](#_Toc11513970)

[2.内存结构 3](#_Toc11513971)

[2.1 共享堆空间 : 3](#_Toc11513972)

[2.2 私有空间： 5](#_Toc11513973)

[3. new()对象 6](#_Toc11513974)

[3.1 对象的创建 6](#_Toc11513975)

[3.2 对象的内存布局 6](#_Toc11513976)

[3.3 访问对象的方式 6](#_Toc11513977)

[二．JVM的 控制 8](#_Toc11513978)

[1. 参数类型划分： 8](#_Toc11513979)

[A: 标准参数 8](#_Toc11513980)

[B: X参数,非标准化参数 8](#_Toc11513981)

[C: XX参数 8](#_Toc11513982)

[2. 垃圾回收算法 8](#_Toc11513983)

[2.1 标记清除算法 9](#_Toc11513984)

[2.2 复制算法 9](#_Toc11513985)

[2.3 标记整理算法 9](#_Toc11513986)

[2.4 分代收集算法 9](#_Toc11513987)

[3. 垃圾收集器 9](#_Toc11513988)

[3.1 并行收集器： 10](#_Toc11513989)

[3.2 并发收集器 Concurrent： 11](#_Toc11513990)

[3.3 垃圾收集器的选择规则 13](#_Toc11513991)

[3.4 URL备注： 13](#_Toc11513992)

[4.垃圾回收日志 13](#_Toc11513993)

[4.1 日志解析 13](#_Toc11513994)

[4.2 例子 14](#_Toc11513995)

[5 Hotspot 参数配置 14](#_Toc11513996)

[5.1 -XX:+DisableExplicitGC 与 NIO的direct memory 14](#_Toc11513997)

[5.2 -XX:MaxDirectMemorySize 14](#_Toc11513998)

[5.3 线程栈大小设置 15](#_Toc11513999)

[5.4 . 年轻代设置 15](#_Toc11514000)

[5.5 Metaspace空间设置 15](#_Toc11514001)

[5.4. 缓存设置 15](#_Toc11514002)

[三．内存溢出案例分析 15](#_Toc11514003)

[1.栈溢出 15](#_Toc11514004)

[2.堆溢出 16](#_Toc11514005)

[2.1 堆内存溢出 16](#_Toc11514006)

[2.2 堆外内存溢出 16](#_Toc11514007)

[3.JVM 统计命令 16](#_Toc11514008)

[四．类加载机制 21](#_Toc11514009)

[1. 类文件结构 21](#_Toc11514010)

[1.1 类文件结构 21](#_Toc11514011)

[2. 类的初始化 25](#_Toc11514012)

[2.1 初始化机制 25](#_Toc11514013)

[2.2通过子类引用父类中的静态字段，这时对子类的引用为被动引用，因此不会初始化子类，只会初始化父类 25](#_Toc11514014)

[2.3常量在编译阶段会存入调用它的类的常量池中 26](#_Toc11514015)

[2.4通过数组定义来引用类，不会触发类的初始化 26](#_Toc11514016)

[2.5接口的初始化过程与类初始化过程的不同 27](#_Toc11514017)

[3.类加载机制（重点） 28](#_Toc11514018)

[3.1类加载过程 28](#_Toc11514019)

[4. Javac编译与JIT编译 31](#_Toc11514020)

[4.1. 编译的作用 31](#_Toc11514021)

[4.2.字节码生成 31](#_Toc11514022)

[4.3. JIT编译(重点) 31](#_Toc11514023)

[五．字节码编译及引用类型 34](#_Toc11514024)

[1. 反编译 34](#_Toc11514025)

[5. 软引用、弱引用、虚引用、final引用、JNI引用 35](#_Toc11514026)

[六．javap 编译结果对照（.class） 35](#_Toc11514027)

[1. 常量压入栈操作 35](#_Toc11514028)

[2. 从栈中局部变量中装载值的命令 36](#_Toc11514029)

[3. 将栈中的值存入局部变量的指令 36](#_Toc11514030)

[4. wide指令 36](#_Toc11514031)

[5. 通用(无类型）栈操作 37](#_Toc11514032)

[6. 整数运算 37](#_Toc11514033)

[7. 按位布尔运算 37](#_Toc11514034)

[8. 浮点运算 38](#_Toc11514035)

[9. 对象操作指令 38](#_Toc11514036)

[10. 数组操作指令 38](#_Toc11514037)

[11. 条件分支指令 38](#_Toc11514038)

[12. 比较指令 39](#_Toc11514039)

[13. 无条件转移指令 39](#_Toc11514040)

[14. 方法调用指令 39](#_Toc11514041)

[15. 方法返回指令 40](#_Toc11514042)

[16. 线程同步 40](#_Toc11514043)

[17 JVM指令助记符( 重点 ) 40](#_Toc11514044)

[八．Tomcat 配置文件调优 41](#_Toc11514045)

[1. Tomcat提供的文档说明 41](#_Toc11514046)

[1.1．conf/server.xml 关于http配置参数说明： 41](#_Toc11514047)

[1.2 bin目录catalina.sh/setenv.sh 41](#_Toc11514048)

备注：参考地址

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2002-02 | 2004-09 | 2006-04 | 2011-07 | 2014-03 | 2017-09 | 2018-03 | 2018-09 |
| 版本名称 | 1.4.0 | 5.0(1.5.0) | 6.0 (1.6.0) | 7.0 (1.7.0) | 8.0 (1.8.0) | 9.0 | 10.0 | 11.0 |
| 收集器 | Par new | CMS | Parallel Old |  | G1 |  |  |  |
| Parallel scanvenge | Serial old:MSC |  |  |  |  |  |  |
| JVM 参数生效 |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

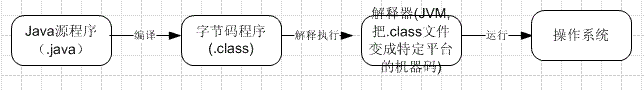
PretenureSizeThreshold : 只对Serial 和ParNew 两款收集器有效；

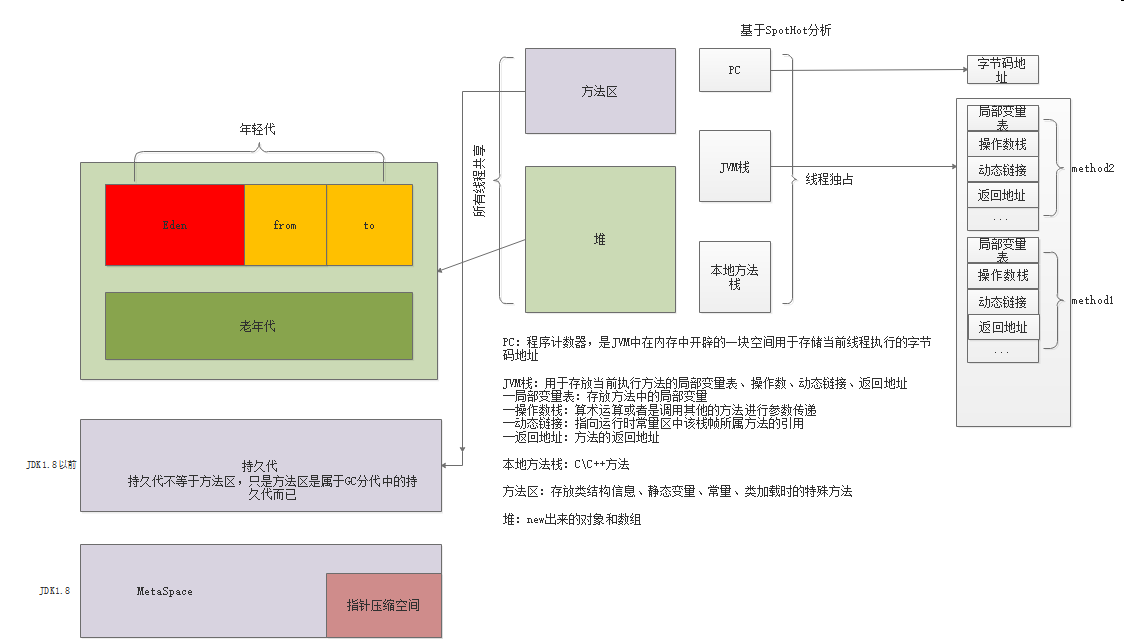
# 专有名词解释：

1. **OSR**：栈上替换，JVM 在程序循环运行时，开始执行此循环已被JIT编译的版本
2. **JIT**：即时编译编译器 just in time ，在JDK1.6版本之后为打破解释执行对字节码生成的影响
3. HSC：**热点代码 Hot Spot Code 。即某个方法或代码块运行特别频繁，用于JIT处理**
4. HSD：**热点探测 Hot Spot Detection，用于JIT**识别出热点代码
5. EA：逃逸分析 Escape analysis，识别对象动态范围的程序分析
6. PEA：部分逃逸分析partial escape analysis

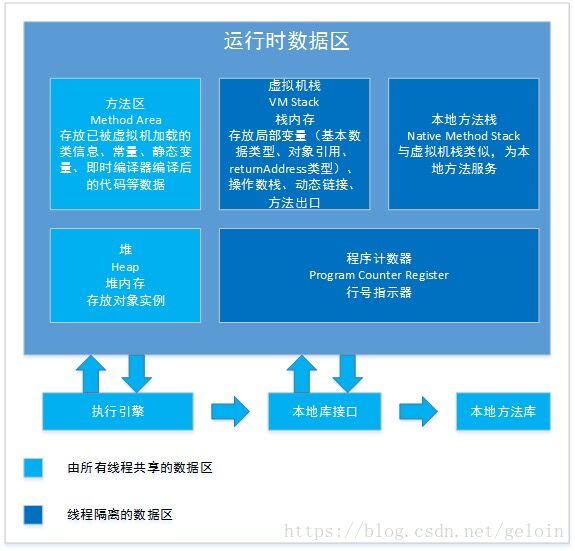
# 一 . JVM 内存结构

## 1.作用





## 2.内存结构



备注：JDK 在1.8 之后 永久内存PermanentGeneration从堆内存中移到了本地内存(native memory)中，即 MateSpace空间 .原持久代这块内存区域被完全移除。 PermSize和MaxPermSize JVM 参数会被忽略

上图( 不包括程序计数器,栈,本地方法栈 )

### 2.1 共享堆空间 :

它是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域唯一的目的是**存放对象实例**，几乎所有的对象实例都在这里分配内存。Java虚拟机规范中描述道：所有的对象实例以及数组都要在堆上分配，但是随着JIT编译器的发展和逃逸分析技术逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化发生，所有的对象都在堆上分配的定论也并不“绝对”了。

堆空间又拆分成：

#### **2.1.1 堆区**

存放java对象，Java堆可以处于物理上不连续的内存中，只要逻辑上是连续的即可，就像磁盘空间。在实现时，可以实现成固定大小或可扩展的，不过当前主流虚拟机是按照可扩展进行实现的（通过-Xmx和 -Xms控制）。

若堆中没有内存完成实例分配，并且堆也无法扩展时，将会抛出**OutOfMemoryError**异常。

为便于内存的回收通常又划分成年轻代，老年代，各区引用对象生命周期的不同会采用不同的回收策略

#### **2.1.2** 方法区

**A．方法区（Method Area）**与Java堆一样

是各个线程共享的内存区域，它用于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等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它有一个别名叫做 Non-Heap（非堆），目的是为了和Java堆区分开来。

**B．虚拟机规范记录（有关异常）**

根据Java虚拟机规范的规定，当方法无法满足内存需求时，将会抛出**OutOfMemoryError**异常

##### **2.1.2.1** 运行时常量池

**A.运行时常量池是方法区的一部分（Runtime Constant Pool）**

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（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池存放

**B．虚拟机规范记录（有关异常）**

**运行时常量池**是方法区的一部分，自然受到方法区的内存限制，当常量池无法再申请到内存时会抛出**OutOfMemoryError**异常。

**C .** **运行时常量池和Class文件**

Java虚拟机对Class文件每一部分（自然包括常量池）的格式有严格规定，每一个字节用于存储那种数据都必须符合规范上的要求才会被虚拟机认可、装载和执行。但对于**运行时常量池**，Java虚拟机规范没有做任何有关细节的要求，不同的提供商实现的虚拟机可以按照自己的需求来实现此内存区域。不过一般而言，除了保存Class文件中的描述符号引用外，还会把翻译出的直接引用也存储在**运行时常量池**中。

**运行时常量池**相对于Class文件常量池的另外一个重要特征是具备**动态性**，Java语言并不要求常量一定只有编译器才能产生，也就是并非置入Class文件中的常量池的内容才能进入方法区**运行时常量池**，运行期间也可能将新的常量放入池中，此特性被开发人员利用得比较多的便是String类的intern() 方法。

MateSpace区域(本地内存来存储类元数据)：

Metaspace: 用于存放class ，package ， Method 字节码，常量池，符号引用等

A: CCS:32位指针的class

B: CodeCache：JIT 动态编译后的本地代码，JNI使用的C代码 。通过（jinfo -flag ReservedCodeCacheSize PID ）查询大小。

### 2.2 私有空间：

#### 2.2.1 Java虚拟机栈

同程序计数器相同，**Java虚拟机栈（**[**Java**](http://lib.csdn.net/base/java)**Virtual Machine Stacks）也是线程私有的**，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个**栈帧**用于存储**局部变量表、操作数栈、动态链接、方法出口**等信息。每一个方法从调用直至执行完成的过程，会对应一个栈帧在虚拟机栈中入栈到出栈的过程。

A．局部变量表中存放了编译期可知的

* **八大数据类型**（boolean、byte、char、short、int、float、long、double）。
* **对象引用**（reference类型，它不等于对象本身，可能是一个指向对象起始地址的指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）
* **returnAddress类型**（指向了一条字节码指令的地址）

**B．操作数栈：是指**同局部变量表一样,操作数栈的最大深度也在编译的时候写入到Code属性的  
max\_stacks数据项中。操作数栈的每一个元素可以是任意的Java数据类型,包括long和double。32位数据类型所占的栈容量为1,64位数据类型所占的栈容量为2。

**C．动态链接：是指**每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接(Dynamic Linking)

**D．方法返回地址：是指**当一个方法开始执行后,只有两种方式可以退出，一种是遇到方法返回的字节码指令；一种是遇见异常，并且这个异常没有在方法体内得到处理。

**无论采用何种退出方式,在方法退出之后,都需要返回到方法被调用的位置,程序才能继续执行,方法返回时可能需要在栈帧中保存一些信息,用来帮助恢复它的上层方法的执行状态。一般来说,方法正常退出时,调用者的PC计数器的值可以作为返回地址,栈帧中很可能会保存这个计数器值。而方法异常退出时,返回地址是要通过异常处理器表来确定的,栈帧中一般不会保存这部分信息。**

在Java虚拟机规范中，对这个区域规定了两种异常状况：

* 若线程请求的栈深度大于虚拟机所允许的深度，将抛出**StackOverflowError**异常。
* 若虚拟机可以动态扩展（当前大部分Java虚拟机都可动态扩展，只不过Java虚拟机规范也允许固定长度的虚拟机栈），当扩展时无法申请到足够的内存，就会抛出**OutOfMemoryError**异常。

#### 2.2.2 本地方法栈

**A．本地方法栈（Native Method Stack）**

与虚拟机栈所发挥的作用类似，它们之间的区别是：虚拟机栈为虚拟机执行**Java**方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的**Native**方法服务。

**B．虚拟机规范记录（有关异常）**

与虚拟机一样，本地方法栈会抛出**StackOverflowError**和**OutOfMemoryError**异常。

#### **2.2.3 程序计数器**

## 3. new()对象

### 3.1 对象的创建

当虚拟机遇到一个new指令时首先会检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并检查这个符号引用代表的类是否已被加载，解析并初始化过。如果没有则必须先执行类的加载过程，当加载后JVM才会为其分配内存，内存大小在类完成加载后便是已知的了

### 3.2 对象的内存布局

对象在内存中存储的布局可分为3块区域：对象头header ，实例数据InstanceData ，对齐填充Padding

#### 3.2.1 对象头header

对象头包括两部分信息：

A 对象自身运行时的数据：哈希码hashCode ，GC分代年龄，锁状态标志 ，线程持有的锁，偏向线程ID以上即为：Mark Word在32位和64位JVM中分别为32bit 和64bit

B 另一部分是类型指针，即对象指向它的类元数据的指针，JVM通过这个指针来确定这个对象是哪个类的实例。

**数组是个例外，通过元数据无法确定数组的大小。**

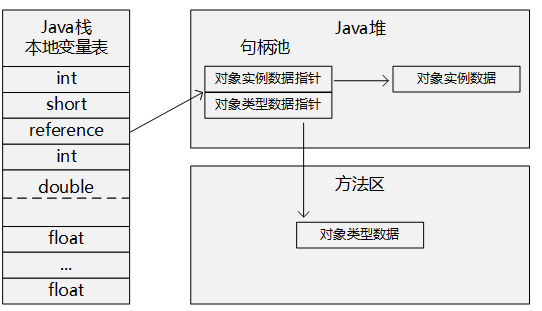
### 3.3 访问对象的方式

**3.3.1 句柄访问**

**由于java栈中存放的reference引用没有定义该通过哪种方式去访问，定位在JVM堆中的对象具体位置，其访问**

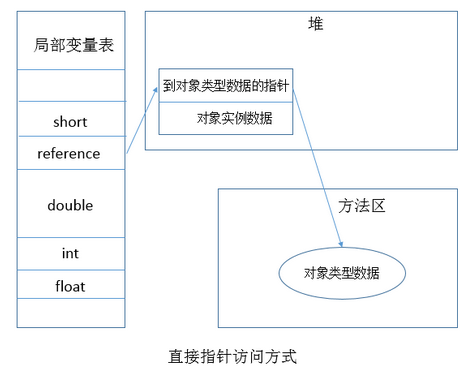
**方式跟JVM实现有很大关系。**

**句柄访问需要在JVM堆中划分出一块内存来作为句柄池，栈中reference存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的地址信息**



**3.3.2 指针访问**

**如果是指针访问则JVM堆中的布局就必须考虑如何放置访问类型的数据信息，reference中存储的就是对象的地址**



**直接指针的方式最大的好处就是快，节省了一次指针定位的开销**

# 二．JVM的 控制

## 1. 参数类型划分：

### A: 标准参数

( 在jvm ) –server –client -version -showversion

### B: X参数,非标准化参数

（ 在各个JDK版本中变动比较小 )

Eg: -Xint：完全解释执行

-Xcomp:第一次使用就编译成本地代码 （ 速度慢 ）

-Xmixed:混合模式 （ jvm自己决定编译类型 ）

### C: XX参数

1 .（Boolean类型）格式：-XX:[+-]<属性的name> 表示（+启用）或者（-禁用）name属性

eg.： -XX:+UseConcMarkSweepGC ( 启用 )

-XX:+UseG1GC (启用G1 垃圾收集器 )

2.（非Boolean类型）格式：-XX:<name>=<value>表示name属性的值是value.

eg: -XX:MaxGcPauseMillis=500

特例：

-Xmx 和-Xms 最大和最小内存参数 ，

-Xss 设置每个线程的堆栈大小JDK5.0以后每个线程堆栈大小为1M，减小这个值能生成更多的线程

-Xms

-Xmx 等价于-XX:MaxHeapSize

**-Xmn2g**：设置年轻代大小为2G

-Xss 设置虚拟机栈内存大小：线程栈的大小是个双刃剑，如果设置过小，可能会出现栈溢出，特别是在该线程内有递归、大的循环时出现溢出的可能性更大，如果该值设置过大，就有影响到创建栈的数量，如果是多线程的应用，就会出现内存溢出的错误

eg: java -Xss=128K HackTheJava

3. 参数控制的内存区域

## 2. 垃圾回收算法

java创建对象后其指针一直被JVM持有，需要JVM判断对象是否可被回收，eg：如果class对象没有被任何引用持有后jvm认为其可被回收。枚举根节点，可达性分析。

根节点包含：a. 类加载器

b. Thread

c. 虚拟机栈的本地变量表

d static 成员

e．常量引用

f. 本地方法栈的变量

JVM 从根节点做可达性分析，

### 2.1 标记清除算法

该算法分两个阶段：标记，清除。标记完成后统一回收，其效率不高，且易产生碎片导致提前GC

### 2.2 复制算法

将内存划分成大小相等的两块，当一块用完了就将还存活的对象复制到另一块，再把原来的一块全部清空。

碎片是没有了，但空间利用率低，如果大部分都是存活的会导致JVM回收过于频繁

### 2.3 标记整理算法

JVM在垃圾回收时采用的是分带垃圾回收，即每个区域采用的回收算法是不一样的，有些对象时在young区分配，生命周期短，复制的效率高些 ；old 区生命周期长，

Eg: young 区用复制算法

Old区用标记清除算法 或者标记整理

JVM的对象优先在Eden区分配，只有大对象才会直接进入老年代。

长期存活的对象直接进入老年代。

备注：大对象的控制命令：-XX:PretenureSizeThreshold

长期存活的对象控制命令：-XX:MaxTenuringThreshold

### 2.4 分代收集算法

就是根据对象存活周期不同将内存划分成几块，按照各个块的存活特点采用适当的收集算法

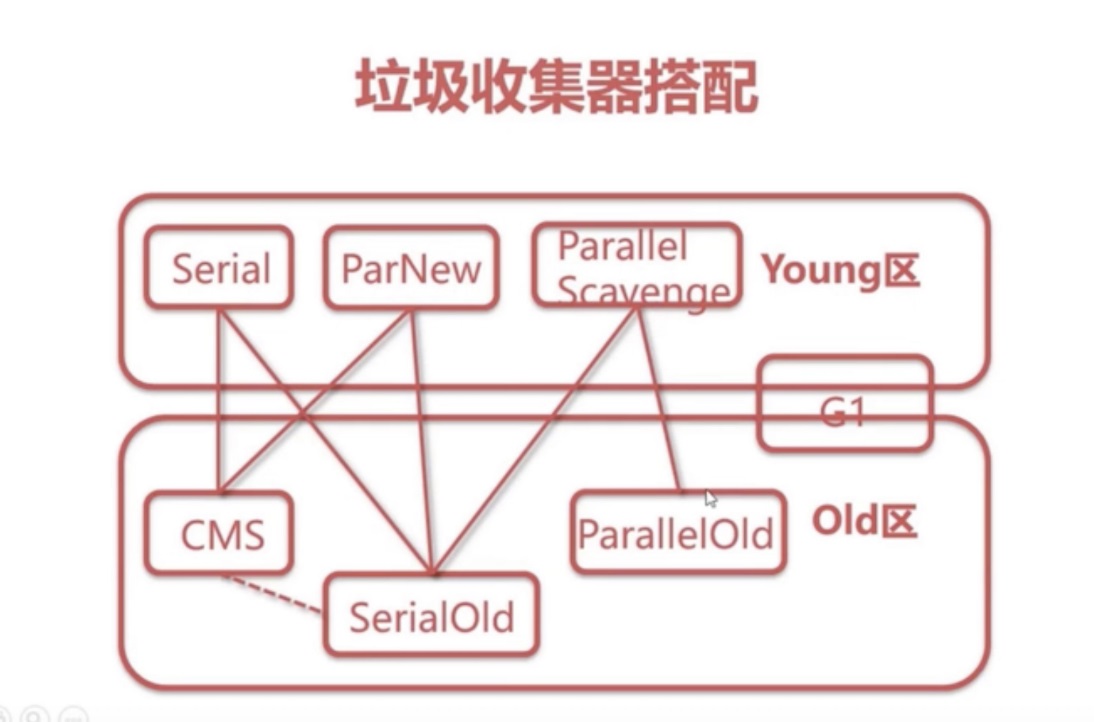
## 3. 垃圾收集器

收集器的分类：

串行收集器Serial （ 单线程收集，内存较小 ）：Serial ， Serial Old 。回收时需要暂停所有工作线程

并行收集器 Parallel ：Parallel Scavenge 新生代收集器 ， Parallel Old 老年代收集器 ，吞吐量

并发收集器Concurrent ： CMS（Concurrent Mark Sweep ） ，G1 （ Garbage First ） ，停顿时间



### 3.1 并行收集器：

并行：是指在一个时刻有多个进程同时执行

3.1.1

吞吐量优先。指多条垃圾收集线程并行工作时用户的线程需要处于停顿状态，该收集器适用于交互操作弱的应用实例。JVM开启选项： (可互相激活)

A：启用收集器设置

-XX:+UseParallelGC

-XX:+UseParallelOldGC

B：并发收集器的参数配合设置

-XX:ParallelGCThreads=<N> 设置GC线程的个数 (CPU > 8 = 5/8 个 CPU < 8 = CPU的实际个数 )

Tomcat 默认采用的就是并行收集器（ jinfo –flag UseParallelGC PID ）并行收集器与串行收集器工作模式相似，只是暂停时并行地进行垃圾收集。年轻代采用复制算法，老年代采用标记-整理，在回收的同时还会对内存进行压缩。关注吞吐量主要指年轻代的Parallel Scavenge收集器，通过两个目标参数-XX:MaxGCPauseMills和-XX:GCTimeRatio，调整新生代空间大小，来降低GC触发的频率。并行收集器适合对吞吐量要求远远高于延迟要求的场景，并且在满足最差延时的情况下，并行收集器将提供最佳的吞吐量

备注：**吞吐量是指**JVM在做垃圾回收和运行应用的时间比值。Eg: 虚拟机总共运行100分钟，垃圾回收耗时1分钟，则吞吐量就是 100 / ( 100 + 1 ) = 99% 参数控制：

-XX:MaxGCPauseMillis , 控制最大垃圾收集停顿时间

-XX:GCTimeTatio=< n > , 垃圾收集时间是： 1 / ( 1+ n ) ，其参考值是一个介于0 到100 之间的数值

吞吐量和停顿时间本身就是一个矛盾体，吞吐量高必然要求停顿时间短，我们在JVM调优时主要针对这两个指标操作。

3.1.2 优势：

3.1.3 缺点：

由于并发进行，CMS在收集与应用线程会同时会增加对堆内存的占用，也就是说，CMS必须要在老年代堆内存用尽之前完成垃圾回收，否则CMS回收失败时，将触发担保机制，串行老年代收集器将会以STW的方式进行一次GC，从而造成较大停顿时间；

标记清除算法无法整理空间碎片，老年代空间会随着应用时长被逐步耗尽，最后将不得不通过担保机制对堆内存进行压缩。CMS也提供了参数-XX:CMSFullGCsBeForeCompaction(默认0，即每次都进行内存整理)来指定多少次CMS收集之后，进行一次压缩的Full GC。

### 3.2 并发收集器 Concurrent：

**并发：是指在一段时间多个进程一起执行**

响应时间优先。指用户线程与垃圾收集器同时执行（也可能交替执行），收集器执行回收时不会停顿用户的程序，适用于交互频繁的应用。但该收集器CPU敏感（） ，会产生动态垃圾容易产生磁盘碎片，因为JVM在回收时用户的进程还在不停的运行。

JVM 收集器设置：

#### 3.2.1 CMS / ( iCMS适用于单核或双核 )

A：启用收集器设置 CMS （ old区的回收器 ）

-XX:+UseConcMarkSweepGC ( old 区 )

-XX:+UseParNewGC ( young 区 )

B：并发收集器的参数配合设置

-XX:ConcGCThreads:并发GC线程数

-XX:+UseCMSCompactAtFullCollection ( 产生磁盘碎片后做内存压缩 )

-XX:CMSFullGCsBeforeCompaction ( N次fullGC 后压缩一次内存 )

-XX:+CMSScavengeBeforeRemark ( 在FullGC 前是否启用youngGC，这样可以事先回收一部分垃圾 )

-XX:+UseConcMarkSweepGC ( 设置年老代为并发收集，年轻代并行，适合于响应要求高的系统 , 开启CMS 收集器 )

-XX:CMSInitiatingOccupancyFrantion: ( 提高回收老年代触发的内存使用空间 FullGC )

-XX:+UseCMSInitiatingOccupancyOnly： （ 是否动态调整 ）

-XX:+CMSScavengeBeforeRemark: ( FullGC 之前先做 YGC )

-XX:+CMSClassUnloadingEnabledL ( 启动回收Perm区 )

-XX:+CMSFullGCsBeforeCompaction ( 设置进行几次fullGC后 ，进行一次碎片整理 , 如果值为0 表示：代表每次都压缩, 把对象移动到内存的最左边，可能会影响性能,但是可以消除碎片)

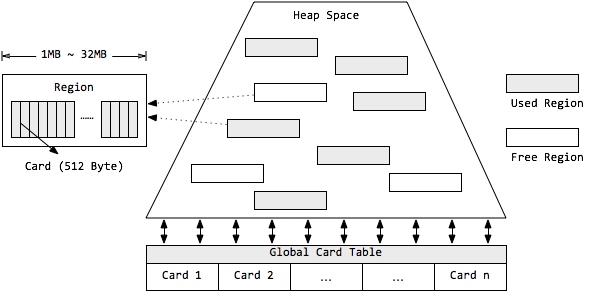
##### 3.2.1.1 CMS 缺点

CMS 是以回收停顿时间为目标的，基于标记—清除算法实现，

1. 在其并发回收阶段虽然不会停止用户线程，但因为回收时占用了一部分线程资源，同样会导致应用程序变慢，总吞吐量降低，CMS默认回收的线程数是（ CPU数量+ 3 ）/4 , 当CPU在4个以上时，会随着CPU数量增加性能下降
2. CMS无法处理浮动垃圾，因为并发回收所以会导致垃圾碎片，CMS无法在本次回收时处理掉它们。
3. 由于垃圾回收阶段用户线程仍在运行，也就还需要留有足够的内存给用户线程使用，所以CMS不能像其他收集器那样等到老年代几乎完全填满了再回收，需要预留一部分空间提供并发收集时的程序使用。上面的CMSInitiatingOccupancyFrantion 提高触发百分比，降低内存回收次数

#### 3.2.2 G1

内存模型：



逻辑组成概念：

1. Region: 分区，将整个堆空间分成若干个大小相等的内存区域
2. Card：在每个分区内部又被分成了若干个大小为512 Byte卡片(Card)
3. Lab ( Local allocation buffer ):本地分配缓冲
4. RSet: ( Remember Set )已记忆集合 , (谁引用了我的对象) 内部类似一个反向指针，记录引用分区内对象的卡片索引

在G1收集器中 没有FullGC的概念，改为了MixedGC：不是FullGC ， 不会全部回收old区，而是会回收young 和 部分old 区

同样也有全局并发标记 ，

##### 3.2.2.1 G1特点

A 并行与并发：回收时仍不需要停顿用户线程

B 空间整合：虽然整体仍采用标记—整理 算法的收集器，但局部的两个Region 之间是基于复制算法实现的，其运作期间不会产生空间碎片

C 可预测的停顿：G1跟踪各个Region里面占用空间大小及回收所需要的时间经验值，其在后台维护一个优先列表，优先回收价值最大的Region

G1 的空间划分仍保留了年轻代老年代，逻辑上采用的是Region，但每个Region都有一个与之对应的Remebered Set 用于存放Region之间的引用关系，避免栈中的引用调用对象时Region还要进行全部区域的扫描，

##### 3.2.2.2 G1 的设置

A：启用收集器设置

-XX:+UseG1GC ( old + young 区 )

B：并发收集器的参数配合设置

**-XX:GCTimeRatio**即为GC与应用的耗费时间比，G1默认为9，而CMS默认为99

-XX:G1HeapReginSize=n ( regin 的大小设置， 在 1---32 M 之间， 小于2048 个 )

-XX:MaxGCPauseMillis=200 ( 最大停顿时间 )

-XX:G1NewSizePercent

-XX:G1MaxNewSizePercent

-XX:concGCThreads=n ( 并发线程数 = 1/4 \* 并行 )

C．关于MixGC的调优：

-XX:InitiatingHeapOccupancyPercent

-XX:G1MixedGCLiveThresholdPercent

-XX:G1HeapWastePercent

-XX:G1MixedGCCountTarget

-XX:G1OldCSetRegionThresholdPercent

#### 设置规则：

1. 停顿时间 是指收集器回收时中断应用的执行时间。可通过参数控制：

-XX:MaxGCPauseMillis

2. 年轻代大小：避免使用-Xmn ,-XX:NewRatio 等显式的设置 Young区大小，会覆盖暂停时间目标。

3.暂停时间目标：时间不要太短，会影响吞吐量

备注: **JVM 的调优主要是针对吞吐量和停顿时间，但这两个参数又是互斥的。**

JVM的每个内存区域采用的收集器搭配使用，

## 3.3 垃圾收集器的选择规则

1. 优先调整堆的大小，让服务器自己来选择

2.如果允许停顿时间大于1秒，则选择并行或者JVM默认的配置

3.如果响应时间要求高并且需要小于1秒，则使用并发收集器

## 3.4 URL备注：

GC管网调优地址：

官方文档：

<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html>

如何选择垃圾收集器

<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html>

G1最佳实践

<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc_tuning.html#recommendations>

G1GC 的关键技术

<https://zhuanlan.zhihu.com/p/22591838>

MetaSpace

https://ifeve.com/jvm-troubleshooting-guide-4/

IBM 关于JVM内存模型的介绍说明：

<https://www.ibm.com/developerworks/library/j-nativememory-linux/>

JVM官方分析统计命令使用说明：

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/>

JIT 即时编译原理

<https://www.ibm.com/developerworks/cn/java/j-lo-just-in-time/index.html>

## 4.垃圾回收日志

### 4.1 日志解析

235.741: [GC (Allocation Failure) [PSYoungGen: 845088K->288K(959488K)] 891328K->46528K(3056640K), 0.0032235 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]

Eg: A. 最前面的数字表示从JVM启动以来经过的秒数

B. GC 和FullGC 表示：这次垃圾收集的停顿类型，如果有Full 说明这次GC发生了Stop-The-word ,

C. 收集器类型：ParNew： Allocation ：

D．接下来表示GC发生的区域：PSYoungGen 年轻代

E．方括号内部的数字：845088K->288K(959488K) 表示：GC前该内存区域已使用容量->GC 后该内存区域已使用容量，最后括号内表示**该区域的总容量**；

F. 方括号外：891328K->46528K(3056640K) 表示：GC前java堆已使用容量 –> GC 后java堆已使用容量，括号表示java堆总容量

G. 0.0032235 secs 表示：GC占用的时间单位是秒

H. user:用户态消耗的时间 ；sys: 内核态消耗时间

### 4.2 例子

2018-06-04T15:32:53.442+0800: 0.690: [GC (**Allocation Failure**) [PSYoungGen: 10240K->1527K(11776K)] 10240K->2494K(39424K), 0.0039626 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

#GC发生的时间戳：从JVM启动到第一次GC的时间：[GC-**年轻代GC(内存分配失败触发GC**) [PS（Parallel Scanvenge）年轻代收集：收集前大小-收集后大小（总大小）] 堆收集前大小-堆收集后大小（对总大小），GC耗时],[耗时：GC应用程序耗时，GC系统耗时，GC实际耗时]

2018-06-04T15:32:54.375+0800: 1.623: [Full GC (Ergonomics) [PSYoungGen: 1536K->0K(22016K)] [ParOldGen: 25538K->21620K(47616K)] 27074K->21620K(69632K), [Metaspace: 8384K->8384K(1056768K)], 0.0952506 secs] [Times: user=0.38 sys=0.00, real=0.10 secs]

#格式都一样，只是内容变了一些，这次是Full GC，不仅收集了年轻代，还收集了年老代和元数据空间（如果JDK小于1.8，则是PermGen-持久代）

## 5 Hotspot 参数配置

### 5.1 -XX:+DisableExplicitGC 与 NIO的direct memory

首先DisableExplicitGC的作用是：限制用户显式的调用System.gc()后JVM置为无效处理。

而System.gc()的默认效果是引发一次stop-the-world的full GC，对整个GC堆做收集

见程序：

同样的程序，不带-XX:+DisableExplicitGC时能正常完成运行，而带上这个参数后却出现了OOM

你可以看到 ByteBuffer.allocateDirect的原代码里面 reserveMemory 方法显式的调用了 System.gc();

### 5.2 **-XX:MaxDirectMemorySize**

**-XX:MaxDirectMemorySize** 是用来配置NIO direct memory上限用的VM参数。但如果不配置它的话，direct memory默认最多能申请多少内存呢？

当MaxDirectMemorySize参数没被显式设置时它的值就是-1，在Java类库初始化时maxDirectMemory()被java.lang.System的静态构造器调用，走的路径就是这条：

Java代码 复制代码 收藏代码

if (s.equals("-1")) {

// -XX:MaxDirectMemorySize not given, take default

directMemory = Runtime.getRuntime().maxMemory();

}

这个max\_capacity()实际返回的是 -Xmx减去一个survivor space的预留大小（G1除外）。

结论：MaxDirectMemorySize没显式配置的时候，NIO direct memory可申请的空间的上限就是-Xmx减去一个survivor space的预留大小。

### 5.3 线程栈大小设置

-Xss 与 -XX:ThreadStackSize

以上两个参数无功能上的区别，后者是HotSpot的内部VM标志。实际上，是前者在许多其他jvm中都支持，但是后者是HotSpot特有的。在JDK的后续版本中，-Xss和-Xoss不赞成使用这些参数。

### 5.4 . 年轻代设置

### 5.5 Metaspace空间设置

A．- -XX:MinMetaspaceFreeRatio设定的过小，会影响内存增长，导致比较频繁的GC。而过大怎会导致内存单次增长过多，造成不必要的浪费

#### 备注：参看openJDK源码

See hotspot/src/share/vm/runtime/arguments.cpp

### 5.4. 缓存设置

对于HotSpot的JIT即时编译设置，分client和server两种模式，如果是server模式，jvm会把编译后的部分机器码存放到寄存器中，即便系统不至于每次都向主存去要数据，但寄存器大小有限，且容量小，一个通常的做法是将代码缓存变成默认大小的两倍或四倍

–XX:ReservedCodeCacheSize=Nflag（N 就是之前提到的默认大小）来最大化代码缓存大小

-XX:InitialCodeCacheSize=N 来声明初始大小（代码缓存的管理类似于 JVM 中的内存管理）

重定义代码缓存的大小并不会真正影响性能，所以设置 ReservedCodeCacheSize 的大小一般是必要的。

# 三．内存溢出案例分析

## 1.栈溢出

Java.lang.StackOverflowError

栈中只保存基础数据类型的对象（比如int i=1中1就是基础类型的对象）和自定义对象的引用,栈中所有内容都是私有的不能共享，且生命周期固定。

## 2.堆溢出

java整个进程占用的内存：   
a. 堆内存   
b. metaspace(堆内) JDK8使用metaspace来替代了permsize:永久代大小   
c. 堆外内存使用   
d. 线程栈空间

### 2.1 堆内存溢出

就是指： -Xmx 堆最大内存不够用了

### 2.2 堆外内存溢出

JIT编译后的本地代码，JNI使用的C代码

DirectByteBuffer ,异步传输在native空间里会有很大的占用空间

### 2.3 占用内存的几种情况

A．Direct Memory ：可以通过-XX:MaxDirectMemorySize 调整大小。虚拟机回收时会对Direct Memory进行回收，但它不会像新生代和老年代那样发现空间不足时通知收集器进行回收，只能等待老年代满了后Full GC，然后顺便帮Direct Memory清理垃圾，否则只能等到抛内存溢出异常，在cache里面做System.gc() .

B． 线程堆栈：-Xss 调整大小

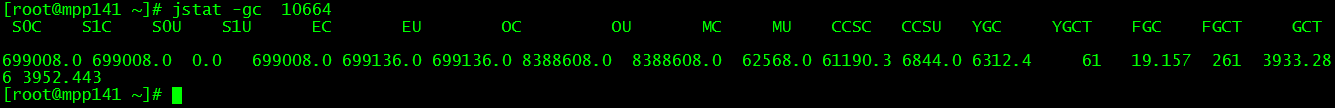
C. Socket 缓存区：每个Socket连接都有Receive 和send 两个缓存区，分别占37和25KB ，如果无法分配了，则可能抛出：IOException :Too many open files

D．JNI代码调用本地库，本地库的内存也不在堆中

## 3. JVM 统计命令

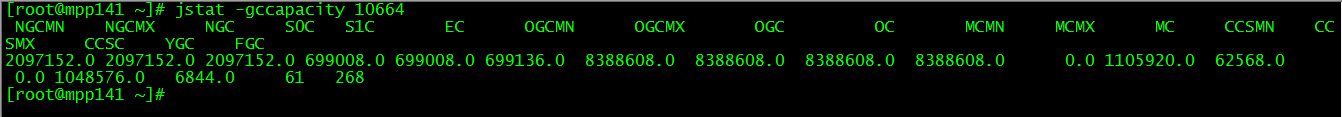
#### 3.1 jstat 主要是监控JVM 类加载信息，jvm垃圾回收信息，运行时，jit 编译信息

##### A . 垃圾回收统计 ： jstat –gc PID



* **S0C：**第一个幸存区的大小
* **S1C：**第二个幸存区的大小
* **S0U：**第一个幸存区的使用大小
* **S1U：**第二个幸存区的使用大小
* **EC：**伊甸园区的大小
* **EU：**伊甸园区的使用大小
* **OC：**老年代大小
* **OU：**老年代使用大小
* **MC：**方法区大小
* **MU：**方法区使用大小
* **CCSC:**压缩类空间大小
* **CCSU:**压缩类空间使用大小
* **YGC：**年轻代垃圾回收次数
* **YGCT：**年轻代垃圾回收消耗时间
* **FGC：**老年代垃圾回收次数
* **FGCT：**老年代垃圾回收消耗时间
* **GCT：**垃圾回收消耗总时间

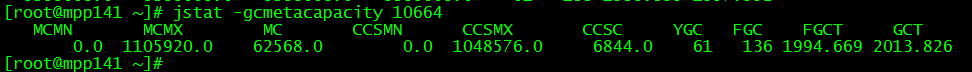
##### B．堆内存统计



jstat -gccapacity

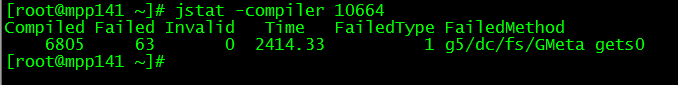
* **NGCMN：**新生代最小容量
* **NGCMX：**新生代最大容量
* **NGC：**当前新生代容量
* **S0C：**第一个幸存区大小
* **S1C：**第二个幸存区的大小
* **EC：**伊甸园区的大小
* **OGCMN：**老年代最小容量
* **OGCMX：**老年代最大容量
* **OGC：**当前老年代大小
* **OC:**当前老年代大小
* **MCMN:**最小元数据容量
* **MCMX：**最大元数据容量
* **MC：**当前元数据空间大小
* **CCSMN：**最小压缩类空间大小
* **CCSMX：**最大压缩类空间大小
* **CCSC：**当前压缩类空间大小
* **YGC：**年轻代gc次数
* **FGC：**老年代GC次数

##### C．元空间统计：jstat -gcmetacapacity PID



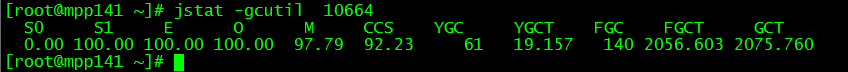
* **MCMN:** 最小元数据容量
* **MCMX：**最大元数据容量
* **MC：**当前元数据空间大小
* **CCSMN：**最小压缩类空间大小
* **CCSMX：**最大压缩类空间大小
* **CCSC：**当前压缩类空间大小
* **YGC：**年轻代垃圾回收次数
* **FGC：**老年代垃圾回收次数
* **FGCT：**老年代垃圾回收消耗时间
* **GCT：**垃圾回收消耗总时间

##### D . 编译统计 jstat –compiler PID



* **Compiled：**编译数量。
* **Failed：**失败数量
* **Invalid：**不可用数量
* **Time：**时间
* **FailedType：**失败类型
* **FailedMethod：**失败的方法

##### E . 总结垃圾回收统计 jstat –gcutil PID



* **S0：**幸存1区当前使用比例
* **S1：**幸存2区当前使用比例
* **E：**伊甸园区使用比例
* **O：**老年代使用比例
* **M：**元数据区使用比例
* **CCS：**压缩使用比例
* **YGC：**年轻代垃圾回收次数
* **FGC：**老年代垃圾回收次数
* **FGCT：**老年代垃圾回收消耗时间
* **GCT：**垃圾回收消耗总时间

#### 3.2 jinfo 查看JVM运行时的参数值

Eg: jinfo -flag ThreadStackSize PID( 线程堆栈大小 )

jinfo -flag UseG1Gc PID ( 是否启用了G1Gc 垃圾回收 )

jinfo -flag UseParallelGC PID ( parallel scavenge + serial Old 组合收集器进行内存回收 )

jinfo -flag UseSerialGC PID （ 使用Serial + Serial Old 组合收集内存 ）

jinfo -flags PID ( 可以查看全部手动赋值的全部参数设置：)

#### 3.3 jstack : JVM提供的堆栈跟踪（线程分析）工具

Eg：查询CPU占用过高的进程

**查询线程栈空间：jinfo -flag ThreadStackSize PID**

1. 获取Java进程的pid ，jps -l , 或者 先top 查询出占用过高的进程ID

命令：top 查询结果的Java进程ID是：8183

2. 查询 8183 进程下占用过高的子线程ID：

命令：top -H -p 8183

比如查询结果获取到的子线程ID：8228 ，8193 ，8186

3.将子进程的ID转换成16进制数字：

命令：printf %x 28837

printf %x 28838

printf %x 28839

4. 打印当前进程ID的所有线程信息 ，（ 可以输出到指定文件中 ）

命令： jstack 8183 > jincheng.8183

5. 最后筛选出线程ID的信息 （ -B10 ： 前10 行信息 ，-A10 后10行信息 ）

cat jincheng.8183 | grep 1ffa -A10 -B10

#### 3.4 jmap:打印共享对象内存映射或堆内存详细信息

Eg: 手动生成内存快照dump：jmap -F -dump:live,file=jmap.hprof PID

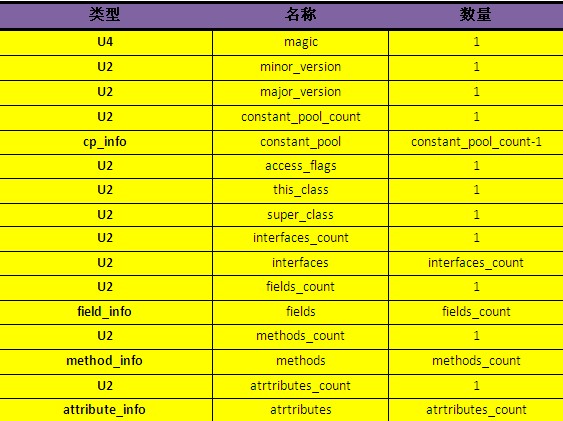
#### 3.5 jcmd：向正在运行的Java虚拟机(JVM)发送诊断命令请求

# 四．类加载机制

## 1. 类文件结构

### 1.1 类文件结构

Class文件格式采用一种类似于C语言结构体的伪结构来存储，这种伪结构中只有两种数据类型：无符号数和表。无符号数属于基本数据类型，以u1、u2、u4、u8来分别代表1、2、4、8个字节的无符号数。表是由多个无符号数或其他表作为数据项构成的符合数据类型，所有的表都习惯性地以“\_info”结尾



#### 1.1.1 magic与version

每个Class文件的头4个字节称为魔数（magic），它的唯一作用是判断该文件是否为一个能被虚拟机接受的Class文件。它的值固定为0xCAFEBABE。紧接着magic的4个字节存储的是Class文件的次版本号和主版本号，高版本的JDK能向下兼容低版本的Class文件

#### 1.1.2 constant\_pool 常量池

它是Class文件中与其他项目关联最多的数据类型，也是占用Class文件空间最大的数据项目之一。

##### A常量池中主要存放两大类常量：

字面量和符号引用。字面量比较接近于Java层面的常量概念，如文本字符串、被声明为final的常量值等。而符号引用总结起来则包括了下面三类常量：

类和接口的全限定名（即带有包名的Class名，如：org.lxh.test.TestClass）

字段的名称和描述符（private、static等描述符）

方法的名称和描述符（private、static等描述符）

##### B．虚拟机在加载Class文件时才会进行动态连接

也就是说，Class文件中不会保存各个方法和字段的最终内存布局信息，因此，这些字段和方法的符号引用不经过转换是无法直接被虚拟机使用的。当虚拟机运行时，需要从常量池中获得对应的符号引用，再在类加载过程中的解析阶段将其替换为直接引用，并翻译到具体的内存地址中。

这里说明下符号引用和直接引用的区别与关联：

符号引用：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到了内存中。

直接引用：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那说明引用的目标必定已经存在于内存之中了。

#### 1.1.3 access\_flag

常量池结束之后，紧接着的2个字节代表访问标志（access\_flag），这个标志用于识别一些类或接口层次的访问信息，包括：这个Class是类还是接口，是否定义为public类型，abstract类型，如果是类的话，是否声明为final

#### 1.1.4 this\_class、super\_class、interfaces

类索引（this\_class）和父类索引（super\_class）都是一个u2类型的数据，而接口索引集合（interfaces）则是一组u2类型的数据集合，Class文件中由这三项数据来确定这个类的继承关系。类索引、父类索引和接口索引集合都按照顺序排列在访问标志之后，类索引和父类索引两个u2类型的索引值表示，它们各自指向一个类型为COMNSTANT\_Class\_info的类描述符常量，通过该常量中的索引值找到定义在COMNSTANT\_Utf8\_info类型的常量中的全限定名字符串。而接口索引集合就用来描述这个类实现了哪些接口，这些被实现的接口将按implements语句（如果这个类本身是个接口，则应当是extend语句）后的接口顺序从左到右排列在接口的索引集合中。

#### 1.1.5 . fields

字段表（field\_info）用于描述接口或类中声明的变量。字段包括了类级变量或实例级变量，但不包括在方法内声明的变量。字段的名字、数据类型、修饰符等都是无法固定的，只能引用常量池中的常量来描述。下面是字段表的最种格式：



#### 1.1.6 methods

方法表（method\_info）的结构与属性表的结构相同，不过多赘述。方法里的Java代码，经过编译器编译成字节码指令后，存放在方法属性表集合中一个名为“Code”的属性里. 在Java语言中，要重载一个方法，除了要与原方法具有相同的简单名称外，还要求必须拥有一个与原方法不同的特征签名，特征签名就是一个方法中各个参数在常量池中的字段符号引用的集合，也就是因为返回值不会包含在特征签名之中，因此Java语言里无法仅仅依靠返回值的不同来对一个已有方法进行重载。

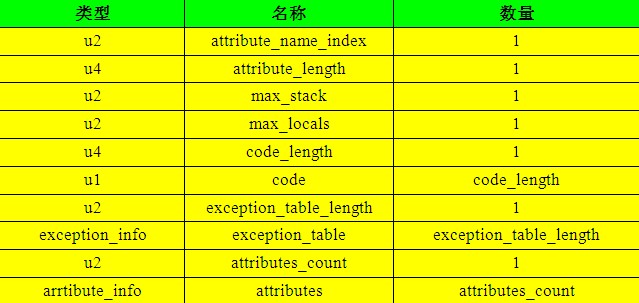
#### 1.1.7     attributes

属性表（attribute\_info）在前面已经出现过多系，在Class文件、字段表、方法表中都可以携带自己的属性表集合，以用于描述某些场景专有的信息. 对于每个属性，它的名称都需要从常量池中引用一个CONSTANT\_Utf8\_info类型的常量来表示，每个属性值的结构是完全可以自定义的，只需说明属性值所占用的位数长度即可。一个符合规则的属性表至少应具有“attribute\_name\_info”、“attribute\_length”和至少一项信息属性。



##### 1.1.7.1 Code属性

Java程序方法体中的代码讲过Javac编译后，生成的字节码指令便会存储在Code属性中，但并非所有的方法表都必须存在这个属性，比如接口或抽象类中的方法就不存在Code属性。如果方法表有Code属性存在，那么它的结构将如下表所示：



ttribute\_name\_index是一项指向CONSTANT\_Utf8\_info型常量的索引，常量值固定为“Code”，它代表了该属性的名称。attribute\_length指示了属性值的长度，由于属性名称索引与属性长度一共是6个字节，所以属性值的长度固定为整个属性表的长度减去6个字节。

max\_stack代表了操作数栈深度的最大值，max\_locals代表了局部变量表所需的存储空间，它的单位是Slot，并不是在方法中用到了多少个局部变量，就把这些局部变量所占Slot之和作为max\_locals的值，原因是局部变量表中的Slot可以重用。

code\_length和code用来存储Java源程序编译后生成的字节码指令。code用于存储字节码指令的一系列字节流，它是u1类型的单字节，因此取值范围为0x00到0xFF，那么一共可以表达256条指令，目前，Java虚拟机规范已经定义了其中200条编码值对应的指令含义。code\_length虽然是一个u4类型的长度值，理论上可以达到2^32-1，但是虚拟机规范中限制了一个方法不允许超过65535条字节码指令，如果超过了这个限制，Javac编译器将会拒绝编译字节码指令之后是这个方法的显式异常处理表集合（exception\_table），它对于Code属性来说并不是必须存在的。它的格式如下表所示：



它包含四个字段，这些字段的含义为：如果字节码从第start\_pc行到第end\_pc行之间（不含end\_pc行）出现了类型为catch\_type或其子类的异常（catch\_type为指向一个CONSTANT\_Class\_info型常量的索引），则转到第handler\_pc行继续处理，当catch\_pc的值为0时，代表人和的异常情况都要转到handler\_pc处进行处理。异常表实际上是Java代码的一部分，编译器使用异常表而不是简单的跳转命令来实现Java异常即finally处理机制，也因此，finally中的内容会在try或catch中的return语句之前执行，并且在try或catch跳转到finally之前，会将其内部需要返回的变量的值复制一份副本到最后一个本地表量表的Slot中

#### 1.1.8. 简要说明下final、static、static final修饰的字段赋值的区别：

static修饰的字段在类加载过程中的准备阶段被初始化为0或null等默认值，而后在初始化阶段（触发类构造器<clinit>）才会被赋予代码中设定的值，如果没有设定值，那么它的值就为默认值。

final修饰的字段在运行时被初始化（可以直接赋值，也可以在实例构造器中赋值），一旦赋值便不可更改；

static final修饰的字段在Javac时生成ConstantValue属性，在类加载的准备阶段根据ConstantValue的值为该字段赋值，它没有默认值，必须显式地赋值，否则Javac时会报错。可以理解为在编译期即把结果放入了常量池中。

#### 1.1.9 InnerClasses

该属性用于记录内部类与宿主类之间的关联。如果一个类中定义了内部类，那么编译器将会为它及它所包含的内部类生成InnerClasses属性。

## 2. 类的初始化

### 2.1 初始化机制

类初始化是类加载过程的最后一个阶段，到初始化阶段，才真正开始执行类中的Java程序代码。虚拟机规范严格规定了有且只有四种情况必须立即对类进行初始化：

遇到new、getstatic、putstatic、invokestatic这四条字节码指令时，如果类还没有进行过初始化，则需要先触发其初始化。生成这四条指令最常见的Java代码场景是：使用new关键字实例化对象时、读取或设置一个类的静态字段（static）时（被static修饰又被final修饰的，已在编译期把结果放入常量池的静态字段除外）、以及调用一个类的静态方法时。

使用Java.lang.refect包的方法对类进行反射调用时，如果类还没有进行过初始化，则需要先触发其初始化。

当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。

当虚拟机启动时，用户需要指定一个要执行的主类，虚拟机会先执行该主类。

虚拟机规定只有这四种情况才会触发类的初始化，称为对一个类进行主动引用，除此之外所有引用类的方式都不会触发其初始化，称为被动引用。下面举一些例子来说明被动引用

### 2.2通过子类引用父类中的静态字段，这时对子类的引用为被动引用，因此不会初始化子类，只会初始化父类

class Father{

public static int m = 33;

static{

System.out.println("父类被初始化");

}

}

class Child extends Father{

static{

System.out.println("子类被初始化");

}

}

public class StaticTest{

public static void main(String[] args){

System.out.println(Child.m);

}

}

执行后输出的结果如下：

父类被初始化

33

对于静态字段，只有直接定义这个字段的类才会被初始化，因此，通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

### 2.3常量在编译阶段会存入调用它的类的常量池中

本质上没有直接引用到定义该常量的类，因此不会触发定义常量的类的初始化

class Const{

public static final String NAME = "我是常量";

static{

System.out.println("初始化Const类");

}

}

public class FinalTest{

public static void main(String[] args){

System.out.println(Const.NAME);

}

}

执行后输出的结果如下：

我是常量

虽然程序中引用了const类的常量NAME，但是在编译阶段将此常量的值“我是常量”存储到了调用它的类FinalTest的常量池中，对常量Const.NAME的引用实际上转化为了FinalTest类对自身常量池的引用。也就是说，实际上FinalTest的Class文件之中并没有Const类的符号引用入口，这两个类在编译成Class文件后就不存在任何联系了。

### 2.4通过数组定义来引用类，不会触发类的初始化

class Const{

static{

System.out.println("初始化Const类");

}

}

public class ArrayTest{

public static void main(String[] args){

Const[] con = new Const[5];

}

}

执行后不输出任何信息，说明Const类并没有被初始化。

但这段代码里触发了另一个名为“LLConst”的类的初始化，它是一个由虚拟机自动生成的、直接继承于java.lang.Object的子类，创建动作由字节码指令newarray触发，很明显，这是一个对数组引用类型的初初始化，而该数组中的元素仅仅包含一个对Const类的引用，并没有对其进行初始化。如果我们加入对con数组中各个Const类元素的实例化代码，便会触发Const类的初始化，如下：

class Const{

static{

System.out.println("初始化Const类");

}

}

public class ArrayTest{

public static void main(String[] args){

Const[] con = new Const[5];

for(Const a:con)

a = new Const();

}

}

这样便会得到如下输出结果：

初始化Const类

根据四条规则的第一条，这里的new触发了Const类。

### 2.5接口的初始化过程与类初始化过程的不同

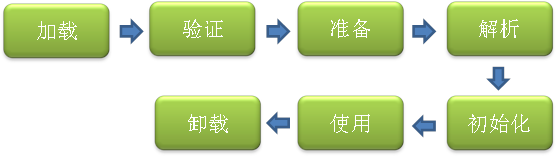
接口也有初始化过程，上面的代码中我们都是用静态语句块来输出初始化信息的，而在接口中不能使用“static{}”语句块，但编译器仍然会为接口生成<clinit>类构造器，用于初始化接口中定义的成员变量（实际上是static final修饰的全局常量）。

二者在初始化时最主要的区别是：当一个类在初始化时，要求其父类全部已经初始化过了，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量），才会初始化该父接口。这点也与类初始化的情况很不同，回过头来看第2个例子就知道，调用类中的static final常量时并不会 触发该类的初始化，但是调用接口中的static final常量时便会触发该接口的初始化。

## 3.类加载机制（重点）

### 3.1类加载过程

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段。它们开始的顺序如下图所示：



其中类加载的过程包括了加载、验证、准备、解析、初始化五个阶段。在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序是确定的，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始，这是为了支持Java语言的运行时绑定（也成为动态绑定或晚期绑定）。 对java来说，绑定分为静态绑定和动态绑定：

A.静态绑定：即前期绑定。在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。针对java，简单的可以理解为程序编译期的绑定。java当中的方法只有final，static，private和构造方法是前期绑定的。

B.动态绑定：即晚期绑定，也叫运行时绑定。在运行时根据具体对象的类型进行绑定。在java中，几乎所有的方法都是后期绑定的。

下面详细讲述类加载过程中每个阶段所做的工作。

#### 3.1.1 加载

加载时类加载过程的第一个阶段，在加载阶段，虚拟机需要完成以下三件事情：

1、通过一个类的全限定名来获取其定义的二进制字节流。

2、将这个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结构。

3、在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口。

注意，这里第1条中的二进制字节流并不只是单纯地从Class文件中获取，比如它还可以从Jar包中获取、从网络中获取（最典型的应用便是Applet）、由其他文件生成（JSP应用）等。

相对于类加载的其他阶段而言，加载阶段（准确地说，是加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，因为开发人员既可以使用系统提供的类加载器来完成加载，也可以自定义自己的类加载器来完成加载。(extends ClassLoader)

加载阶段完成后，**虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中**，而且在Java**堆中也创建一个java.lang.Class类的对象**，这样便可以**通过该对象访问方法区中的这些数据**。

对于任意一个类，都需要由它的类加载器和这个类本身一同确定其在就Java虚拟机中的唯一性，也就是说，即使两个类来源于同一个Class文件，只要加载它们的类加载器不同，那这两个类就必定不相等。这里的“相等”包括了代表类的Class对象的equals（）、isAssignableFrom（）、isInstance（）等方法的返回结果，也包括了使用instanceof关键字对对象所属关系的判定结果。

#### 3.1.2 加载类型

站在Java开发人员的角度来看，类加载器可以大致划分为以下三类：

A．启动类加载器：Bootstrap ClassLoader，跟上面相同。它负责加载存放在JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库（如rt.jar，所有的java.\*开头的类均被Bootstrap ClassLoader加载）。启动类加载器是无法被Java程序直接引用的。

B．扩展类加载器：Extension ClassLoader，该加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载JDK\jre\lib\ext目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.\*开头的类），开发者可以直接使用扩展类加载器。

C．应用程序类加载器：Application ClassLoader，该类加载器由sun.misc.Launcher$AppClassLoader来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

#### 3.1.3 自定义的类加载器作用

因为JVM自带的ClassLoader只是懂得从本地文件系统加载标准的java class文件，因此如果编写了自己的ClassLoader，便可以做到如下几点：

A．）在执行非置信代码之前，自动验证数字签名。

B．）动态地创建符合用户特定需要的定制化构建类。

C．）从特定的场所取得java class，例如数据库中和网络中。

#### 3.1.4 .双亲委派模型的工作流程

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。

使用双亲委派模型来组织类加载器之间的关系，有一个很明显的好处，就是Java类随着它的类加载器（说白了，就是它所在的目录）一起具备了一种带有优先级的层次关系，这对于保证Java程序的稳定运作很重要。例如，类java.lang.Object类存放在JDK\jre\lib下的rt.jar之中，因此无论是哪个类加载器要加载此类，最终都会委派给启动类加载器进行加载，这边保证了Object类在程序中的各种类加载器中都是同一个类。

#### 3.1.5 验证

**验证的目的是为了确保Class文件中的字节流包含的信息符合当前虚拟机的要求**，而且不会危害虚拟机自身的安全。不同的虚拟机对类验证的实现可能会有所不同，但大致都会完成以下四个阶段的验证：文件格式的验证、元数据的验证、字节码验证和符号引用验证。

A．文件格式的验证：验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理，该验证的主要目的是保证输入的字节流能正确地解析并存储于方法区之内。经过该阶段的验证后，字节流才会进入内存的方法区中进行存储，后面的三个验证都是基于方法区的存储结构进行的。

B．元数据验证：对类的元数据信息进行语义校验（其实就是对类中的各数据类型进行语法校验），保证不存在不符合Java语法规范的元数据信息。

C．字节码验证：该阶段验证的主要工作是进行数据流和控制流分析，对类的方法体进行校验分析，以保证被校验的类的方法在运行时不会做出危害虚拟机安全的行为。

D.符号引用验证：这是最后一个阶段的验证，它发生在虚拟机将符号引用转化为直接引用的时候（解析阶段中发生该转化，后面会有讲解），主要是对类自身以外的信息（常量池中的各种符号引用）进行匹配性的校验。

#### 3.1.6 备注

这里还需要注意如下几点：

对基本数据类型来说，对于类变量（static）和全局变量，如果不显式地对其赋值而直接使用，则系统会为其赋予默认的零值，而对于局部变量来说，在使用前必须显式地为其赋值，否则编译时不通过。

对于同时被static和final修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过；而只被final修饰的常量则既可以在声明时显式地为其赋值，也可以在类初始化时显式地为其赋值，总之，在使用前必须为其显式地赋值，系统不会为其赋予默认零值。

对于引用数据类型reference来说，如数组引用、对象引用等，如果没有对其进行显式地赋值而直接使用，系统都会为其赋予默认的零值，即null。

如果在数组初始化时没有对数组中的各元素赋值，那么其中的元素将根据对应的数据类型而被赋予默认的零值。

#### 3.1.7 解析

解析阶段是虚拟机将常量池中的符号引用转化为直接引用的过程。 解析阶段可能开始于初始化之前，也可能在初始化之后开始，虚拟机会根据需要来判断，

**解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行，分别对应于常量池中的CONSTANT\_Class\_info、**

**CONSTANT\_Fieldref\_info、**

**CONSTANT\_Methodref\_info、**

**CONSTANT\_InterfaceMethodref\_info**

**四种常量类型。**

**A、类或接口的解析：**判断所要转化成的直接引用是对数组类型，还是普通的对象类型的引用，从而进行不同的解析

**B、字段解析：**对字段进行解析时，会先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束；如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束

**C、类方法解析：**对类方法的解析与对字段解析的搜索步骤差不多，只是多了判断该方法所处的是类还是接口的步骤，而且对类方法的匹配搜索，是先搜索父类，再搜索接口。

#### 3.1.8 初始化

初始化是类加载过程的最后一步，到了此阶段，才真正开始执行类中定义的Java程序代码。在准备阶段，类变量已经被赋过一次系统要求的初始值

从另一个角度来表达：初始化阶段是执行类构造器<clinit>()方法的过程。

这里简单说明下<clinit>（）方法的执行规则:

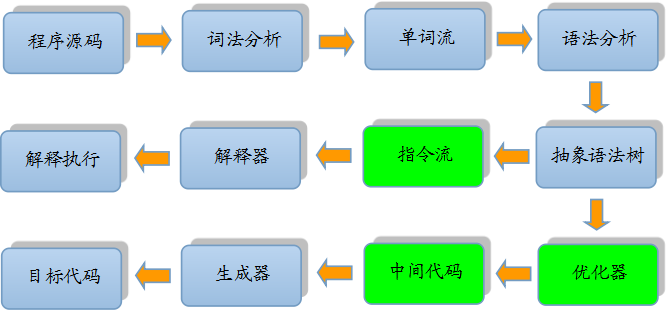
1、<clinit>（）方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句中可以赋值，但是不能访问。

2、<clinit>（）方法与实例构造器<init>（）方法（类的构造函数）不同，它不需要显式地调用父类构造器，虚拟机会保证在子类的<clinit>（）方法执行之前，父类的<clinit>（）方法已经执行完毕。因此，在虚拟机中第一个被执行的<clinit>（）方法的类肯定是java.lang.Object。

整个类加载过程中，除了在加载阶段用户应用程序可以自定义类加载器参与之外，其余所有的动作完全由虚拟机主导和控制。到了初始化才开始执行类中定义的Java程序代码（亦及字节码），但这里的执行代码只是个开端，它仅限于<clinit>（）方法。类加载过程中主要是**将Class文件（准确地讲，应该是类的二进制字节流）加载到虚拟机内存中，真正执行字节码的操作**，在加载完成后才真正开始

## 4. Javac编译与JIT编译

不论是物理机还是虚拟机，大部分的程序代码从开始编译到最终转化成物理机的目标代码或虚拟机能执行的指令集之前，都会按照如下图所示的各个步骤进行：



### 4.1. 编译的作用

为什么要编译？javac编译在程序运行期间将字节码转变成机器码（现在的Java程序在运行时基本都是解释执行加编译执行），如HotSpot虚拟机自带的JIT（Just In Time Compiler）编译器（分Client端和Server端）。另外，有时候还有可能会碰到静态提前编译器（AOT，Ahead Of Time Compiler）直接把\*.java文件编译成本地机器代码，如GCJ、Excelsior JET等 。

### 4.2.字节码生成

字节码生成是Javac编译过程的最后一个阶段。**字节码生成阶段不仅仅是把前面各个步骤所生成的信息转化成字节码写到磁盘中**，编译器还进行了少量的代码添加和转换工作。 实例构造器<init>（）方法和类构造器<clinit>（）方法就是在这个阶段添加到语法树之中的（这里的实例构造器并不是指默认的构造函数，而是指我们自己重载的构造函数，如果用户代码中没有提供任何构造函数，那编译器会自动添加一个没有参数、访问权限与当前类一致的默认构造函数

### 4.3. JIT编译(重点)

JIT的由来: Java程序最初是仅仅通过解释器解释执行的，即对字节码逐条解释执行，这种方式的执行速度相对会比较慢，尤其当某个方法或代码块运行的特别频繁时，这种方式的执行效率就显得很低。于是后来在虚拟机中引入了**JIT编译器（即时编译器）**，当虚拟机发现某个方法或代码块运行特别频繁时，就会把这些代码认定为“Hot Spot Code”（**热点代码**），为了提高热点代码的执行效率，**在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各层次的优化，完成这项任务的正是JIT编译器**。

现在主流的商用虚拟机（如Sun HotSpot、IBM J9）中几乎都同时包含解释器和编译器（三大商用虚拟机之一的JRockit是个例外，它内部没有解释器，因此会有启动相应时间长之类的缺点，但它主要是面向服务端的应用，这类应用一般不会重点关注启动时间）。二者各有优势：当程序需要迅速启动和执行时，解释器可以首先发挥作用，省去编译的时间，立即执行；当程序运行后，随着时间的推移，编译器逐渐发挥作用 。把越来越多的代码编译成本地代码后，可以获取更高的执行效率。解释执行可以节约内存，而编译执行（必须把机器码文件加载到内存）可以提升效率

“.class”文件里面实际是字节码，而不是可以直接执行的机器码。Java 通过字节码和 Java 虚拟机（JVM）这种跨平台的抽象，屏蔽了操作系统和硬件的细节，这些是实现“一次编译，到处执行”的基础。

在运行时，**JVM 会通过类加载器（Class-Loader）加载字节码，解释或者编译执行**。就像我前面提到的，主流 Java 版本中，如 JDK 8 实际是解释和编译混合的一种模式，即所谓的混合模式（-Xmixed）。通常运行在 server 模式的 JVM，会进行上万次调用以收集足够的信息进行高效的编译，client 模式这个门限是 1500 次。Oracle Hotspot JVM 内置了两个不同的 JIT编译(JIT compiler)，C1 对应前面说的 client 模式，适用于对于启动速度敏感的应用，比如普通 Java 桌面应用；C2 对应 server 模式，它的优化是为长时间运行的服务器端应用设计的。默认是采用所谓的分层编译（TieredCompilation）。

JIT 编译在部分商用虚拟机中（如HotSpot），Java程序最初是通过解释器（Interpreter）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时，就会把这些代码认定为“热点代码”(**1、被多次调用的方法,2、被多次执行的循环体**, 在HotSpot虚拟机中使用的是——基于计数器的热点探测方法，因此它为每个方法准备了两个计数器：**方法调用计数器和回边计数器**。在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发JIT编译。)。为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器称为即时编译器（Just In Time Compiler，下文统称JIT编译器）

#### 备注： 回边计数器：

用于统计一个方法中循环体代码执行的次数。回边计数器触发JIT编译的流程与方法调用计数器极其类似。

当解释器遇到一条回边指令（编译原理的相关知识，可以粗略理解为循环）时，会先检查将要执行的代码片段是否存在被JIT编译过的版本，如果存在，则优先使用编译后的本地代码来执行。如果不存在已经被编译的版本，则将此方法的回边计数器加1，然后判断方法调用计数器与回边计数器之和是否超过回边调用计数器的阀值，如果已经超过阀值，那么将会向即时编译器提交一个OSR编译请求，并且会把回边计数器的值降低一些，以便继续在解释器中执行循环

#### 4.3.1 解释器和编译器互补

解释器与编译器两者各有优势：当程序需要迅速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即执行。在程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率。当程序运行环境中内存资源限制较大（如部分嵌入式系统中），可以使用解释器执行节约内存，反之可以使用编译执行来提升效率。

HotSpot虚拟机中内置了两个JIT编译器：Client Complier和Server Complier，简称为C1、C2编译器，分别用在客户端和服务端。目前**主流的HotSpot虚拟机中默认是采用解释器与其中一个编译器直接配合的方式工作**。程序使用哪个编译器，取决于虚拟机运行的模式。HotSpot虚拟机会根据自身版本与宿主机器的硬件性能自动选择运行模式，**用户也可以使用“-client”或“-server”参数去强制指定虚拟机运行在Client模式或Server模式**。用Client Complier获取更高的编译速度，用Server Complier 来获取更好的编译质量。为什么提供多个即时编译器与为什么提供多个垃圾收集器类似，都是为了适应不同的应用场景。

备注： Client Complier获取更高的编译速度： ？？？

Server Complier 来获取更好的编译质量：？？？

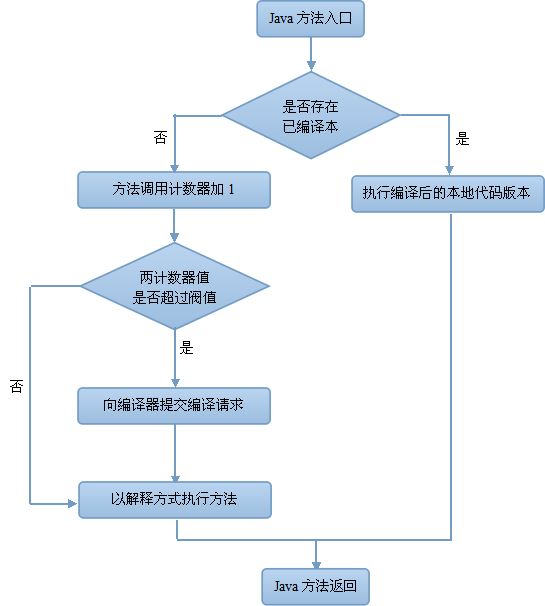
两种情况，编译器都是以整个方法作为编译对象，这种编译也是虚拟机中标准的编译方式。要知道一段代码或方法是不是热点代码，是不是需要触发即时编译，需要进行Hot Spot Detection（热点探测）。目前主要的热点 判定方式有以下两种：

A．基于采样的热点探测：采用这种方法的虚拟机会周期性地检查各个线程的栈顶，如果发现某些方法经常出现在栈顶，那这段方法代码就是“热点代码”。这种探测方法的好处是实现简单高效，还可以很容易地获取方法调用关系，缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。

B．基于计数器的热点探测：采用这种方法的虚拟机会为每个方法，甚至是代码块建立计数器，统计方法的执行次数，如果执行次数超过一定的阀值，就认为它是“热点方法”。这种统计方法实现复杂一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系，但是它的统计结果相对更加精确严谨。

在确定虚拟机运行参数的前提下，这**两个计数器都有一个确定的阀值**，当计数器的值超过了阀值，就会触发JIT编译。触发了JIT编译后，在默认设置下，执行引擎并不会同步等待编译请求完成，而是**继续进入解释器按照解释方式执行字节码，直到提交的请求被编译器编译完成为止**（编译工作在后台线程中进行）。当编译工作完成后，下一次调用该方法或代码时，就会使用已编译的版本。

由于方法计数器触发即时编译的过程与回边计数器触发即时编译的过程类似，因此这里仅给出方法调用计数器触发即时编译的流程：



**Javac字节码编译器与虚拟机内的JIT编译器的执行过程合起来其实就等同于一个传统的编译器所执行的编译过程。**

# 五．字节码编译及引用类型

## 1. 反编译

1.1 **javap**

**A．-c 反编译的方式**

**B．-s命令可以查看class反编译结果中所有函数的签名**

**C．** -p  -private 显示所有类和成员

## 5. 软引用、弱引用、虚引用、final引用、JNI引用

a. 软引用:

b. 弱引用

c. 虚引用

d.强引用：

# 六．javap 编译结果对照（.class）

https://blog.csdn.net/qq\_32783577/article/details/82848180

## 1. 常量压入栈操作

aconst\_null 将null对象引用压如栈

iconst\_1 将int类型常量 1压入栈

iconst\_m1 将int类型常量-1压入栈

lconst\_1 将long类型常量1压入栈

fconst\_1 将float类型常量1压入栈

dconst\_1 将double类型常量压入栈

bipush 将一个8位带符号整数压入栈中

sipush 将一个16位带符号整数压入栈中

idc 把常量池中的项压入栈

idc\_w 把常量池中的项压入栈(使用宽索引

idc2\_w 把常量池中long类型或者double类型的项压入栈(使用款索引)

## 2. 从栈中局部变量中装载值的命令

iload 从局部变量中装载int类型值

lload 从局部变量中装载long类型值

fload 从局部变量中装载float类型值

dload 从局部变量中装载double类型值

aload 从局部变量中装载引用类型的值(refernce)

iload\_0 从局部变量0中装载int类型值

lload\_0 从局部变量0中装载long类型值

fload\_0 从局部变量0中装载float类型值

dload\_0 从局部变量0中装载double类型值

aload\_0 从局部变量0中装载引用类型值

iaload 从数组中装载int类型值

laload 从数组中装载long类型值

faload 从数组中装载float类型值

daload 从数组中装载double类型值

aaload 从数组中装载引用类型值

baload 从数组中装载byte类型或boolean类型值

caload 从数组中装载char类型值

saload 从数组中装载short类型值

## 3. 将栈中的值存入局部变量的指令

istore 将int类型值存入局部变量

lstore 将long类型值存入局部变量

fstore 将float类型值存入局部变量

dstore 将double类型值存入局部变量

astore 将将引用类型或returnAddress类型值存入局部变量

istore\_0 将int类型值存入局部变量0

lstore\_1 将long类型值存入局部变量1

fstore\_0 将float类型值存入局部变量0

dstore\_0 将double类型值存入局部变量0

astore\_3 将引用类型或returnAddress类型值存入局部变量3

astore 将int类型值存入数组中

lastore 将long类型值存入数组中

fastore 将float类型值存入数组中

dastore 将double类型值存入数组中

aastore 将引用类型值存入数组中

bastore 将byte类型或者boolean类型值存入数组中

castore 将char类型值存入数组中

sastore 将short类型值存入数组中

## 4. wide指令

wide 使用附加字节扩展局部变量索引

## 5. 通用(无类型）栈操作

nop 不做任何操作

pop 弹出栈顶端一个字长的内容

pop2 弹出栈顶端两个字长的内容

dup 复制栈顶部一个字长内容

dup\_x1 复制栈顶部一个字长的内容，然后将复制内容及原来弹出的两个字长的内容压入栈

dup\_x2 复制栈顶部一个字长的内容，然后将复制内容及原来弹出的三个字长的内容压入栈

dup2 复制栈顶部两个字长内容

dup2\_x1 复制栈顶部两个字长的内容，然后将复制内容及原来弹出的三个字长的内容压入栈

dup2\_x2 复制栈顶部两个字长的内容，然后将复制内容及原来弹出的四个字长的内容压入栈

swap 交换栈顶部两个字长内容类型转换

i2l 把int类型的数据转化为long类型

i2f 把int类型的数据转化为float类型

i2d 把int类型的数据转化为double类型

l2i 把long类型的数据转化为int类型

l2f 把long类型的数据转化为float类型

l2d 把long类型的数据转化为double类型

f2i 把float类型的数据转化为int类型

f2l 把float类型的数据转化为long类型

f2d 把float类型的数据转化为double类型

d2i 把double类型的数据转化为int类型

d2l 把double类型的数据转化为long类型

d2f 把double类型的数据转化为float类型

i2b 把int类型的数据转化为byte类型

i2c 把int类型的数据转化为char类型

i2s 把int类型的数据转化为short类型

## 6. 整数运算

iadd 执行int类型的加法

ladd 执行long类型的加法

isub 执行int类型的减法

lsub 执行long类型的减法

imul 执行int类型的乘法

lmul 执行long类型的乘法

idiv 执行int类型的除法

ldiv 执行long类型的除法

irem 计算int类型除法的余数

lrem 计算long类型除法的余数

ineg 对一个int类型值进行取反操作

lneg 对一个long类型值进行取反操作

iinc 把一个常量值加到一个int类型的局部变量上

## 7. 按位布尔运算

iand 对int类型值进行“逻辑与”操作

land 对long类型值进行“逻辑与”操作

ior 对int类型值进行“逻辑或”操作

lor 对long类型值进行“逻辑或”操作

ixor 对int类型值进行“逻辑异或”操作

lxor 对long类型值进行“逻辑异或”操作

## 8. 浮点运算

fadd 执行float类型的加法

dadd 执行double类型的加法

fsub 执行float类型的减法

dsub 执行double类型的减法

fmul 执行float类型的乘法

dmul 执行double类型的乘法

fdiv 执行float类型的除法

ddiv 执行double类型的除法

frem 计算float类型除法的余数

drem 计算double类型除法的余数

fneg 将一个float类型的数值取反

dneg 将一个double类型的数值取反

## 9. 对象操作指令

new 创建一个新对象

checkcast 确定对象为所给定的类型

getfield 从对象中获取字段

putfield 设置对象中字段的值

getstatic 从类中获取静态字段

putstatic 设置类中静态字段的值

instanceof 判断对象是否为给定的类型

## 10. 数组操作指令

newarray 分配数据成员类型为基本上数据类型的新数组

anewarray 分配数据成员类型为引用类型的新数组

arraylength 获取数组长度

multianewarray 分配新的多维数组

## 11. 条件分支指令

ifeq 如果等于0，则跳转

ifne 如果不等于0，则跳转

iflt 如果小于0，则跳转

ifge 如果大于等于0，则跳转

ifgt 如果大于0，则跳转

ifle 如果小于等于0，则跳转

if\_icmpcq 如果两个int值相等，则跳转

if\_icmpne 如果两个int类型值不相等，则跳转

if\_icmplt 如果一个int类型值小于另外一个int类型值，则跳转

if\_icmpge 如果一个int类型值大于或者等于另外一个int类型值，则跳转

if\_icmpgt 如果一个int类型值大于另外一个int类型值，则跳转

if\_icmple 如果一个int类型值小于或者等于另外一个int类型值，则跳转

ifnull 如果等于null，则跳转

ifnonnull 如果不等于null，则跳转

if\_acmpeq 如果两个对象引用相等，则跳转

if\_acmpnc 如果两个对象引用不相等，则跳转

## 12. 比较指令

lcmp 比较long类型值

fcmpl 比较float类型值（当遇到NaN时，返回-1）

fcmpg 比较float类型值（当遇到NaN时，返回1）

dcmpl 比较double类型值（当遇到NaN时，返回-1）

dcmpg 比较double类型值（当遇到NaN时，返回1）

## 13. 无条件转移指令

goto 无条件跳转

goto\_w 无条件跳转（宽索引）

--表跳转指令

tableswitch 通过索引访问跳转表，并跳转

lookupswitch 通过键值匹配访问跳转表，并执行跳转操作

--异常

athrow 抛出异常或错误

--finally子句

jsr 跳转到子例程

jsr\_w 跳转到子例程（宽索引）

rct 从子例程返回

## 14. 方法调用指令

invokcvirtual 运行时按照对象的类来调用实例方法

invokespecial 根据编译时类型来调用实例方法 ==>调用超类构造方法、实例初始化方法、私有方法

invokestatic 调用类（静态）方法

invokcinterface 调用接口方法

## 15. 方法返回指令

ireturn 从方法中返回int类型的数据

lreturn 从方法中返回long类型的数据

freturn 从方法中返回float类型的数据

dreturn 从方法中返回double类型的数据

areturn 从方法中返回引用类型的数据

return 从方法中返回，返回值为void

## 16. 线程同步

monitorenter 进入并获取对象监视器 (获取对象锁)

monitorexit 释放并退出对象监视器 (释放对象锁)

## 17 JVM指令助记符( 重点 )

变量到操作数栈： iload,iload\_<n>,lload,lload\_<n>,fload,fload\_<n>,dload,dload\_<n>,aload,aload\_<n>

操作数栈到变量： istore,istore\_<n>,lstore,lstore\_<n>,fstore,fstore\_<n>,dstore,dstor\_<n>,astore,astore\_<n>

常数到操作数栈bipush,sipush,ldc,ldc\_w,ldc2\_w,aconst\_null,iconst\_ml,iconst\_<i>,lconst\_<l>,fconst\_<f>,dconst\_<d>

加： iadd,ladd,fadd,dadd

减： isub,lsub,fsub,dsub

乘： imul,lmul,fmul,dmul

除： idiv,ldiv,fdiv,ddiv

余数： irem,lrem,frem,drem

取负： ineg,lneg,fneg,dneg

移位： ishl,lshr,iushr,lshl,lshr,lushr

按位或： ior,lor

按位与： iand,land

按位异或： ixor,lxor

类型转换： i2l,i2f,i2d,l2f,l2d,f2d(放宽数值转换)

i2b,i2c,i2s,l2i,f2i,f2l,d2i,d2l,d2f(缩窄数值转换)

创建类实便： new

创建新数组： newarray,anewarray,multianwarray

访问类的域和类实例域： getfield,putfield,getstatic,putstatic

把数据装载到操作数栈： baload,caload,saload,iaload,laload,faload,daload,aaload

从操作数栈存存储到数组： bastore,castore,sastore,iastore,lastore,fastore,dastore,aastore

获取数组长度： arraylength

检相类实例或数组属性： instanceof,checkcast

操作数栈管理： pop,pop2,dup,dup2,dup\_xl,dup2\_xl,dup\_x2,dup2\_x2,swap

有条件转移： ifeq,iflt,ifle,ifne,ifgt,ifge,ifnull,ifnonnull,if\_icmpeq,if\_icmpene,

if\_icmplt,if\_icmpgt,if\_icmple,if\_icmpge,if\_acmpeq,if\_acmpne,lcmp,fcmpl

fcmpg,dcmpl,dcmpg

复合条件转移： tableswitch,lookupswitch

无条件转移： goto,goto\_w,jsr,jsr\_w,ret

调度对象的实便方法： invokevirtual

调用由接口实现的方法： invokeinterface

调用需要特殊处理的实例方法:invokespecial

调用命名类中的静态方法： invokestatic

方法返回： ireturn,lreturn,freturn,dreturn,areturn,return

异常： athrow

finally关键字的实现使用： jsr,jsr\_w,ret

# 八．Tomcat 配置文件调优

## 1. Tomcat提供的文档说明

官方说明：apache-tomcat-8.5.34/webapps/docs/config/http.html

### 1.1．conf/server.xml 关于http配置参数说明：

maxConnections :最大连接数，8版本之后都是NIO 多路复用 默认给的链接数：10000

备注：此处的连接数是受限于OS允许建立的最大连接数，eg：Linux下的

acceptCount: 超出最大值之后配置的队列长度，

maxThreads: 单位时间可处理的并发线程数，工作线程 默认200

minSpareThreads: 最小空闲的工作线程

备注：1. server.xml 中的autoDeploy=true表示Tomcat会周期性的检查是否有新的应用部署，严重影响性能。

2. enableLookups: DNS 查询

3.reloadable:监控webapp下的web 的变化，会重新载入 ：默认关闭

Executor

### 1.2 bin目录catalina.sh/setenv.sh