JVM

Java内存分配

- **程序计数器(PC Program Counter Register)**

  在JVM规范中，每个线程都有它自己的程序计数器，并且任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器会存储当前线程正在执行的Java方法的JVM指令地址;或者，如果是在执行本地方法，则是未指定值（undefined）。（唯一不会抛出OutOfMemoryError)

  作用：`保证多线程能正常运行`

- **虚拟机栈(Java Virtual Machine Stack)**

  第二，Java虚拟机栈(Java Virtual Machine Stack），早期也叫Java栈。每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧(Stack Frame)，对应着一次次的Java方法调用

  前面谈程序计数器时，提到了当前方法；同理，在一个时间点，对应的只会有一个活动的栈帧，通常叫作当前帧，方法所在的类叫作当前类。如果在该方法中调用了其他方法，对应的新的栈帧会被创建出来，成为新的当前帧，一直到它返回结果或者执行结束。VM直接对Java栈的操作只有两个，就是对栈帧的压栈和出栈。

  栈帧中存储着`局部变量表(8字节)`、操作数（operand）栈(单线程，对应多线程中的程序计数器)、动态链接、方法正常退出或者异常退出的定义等。

- **堆(heap)**

  第三，堆（Heap），它是Java内存管理的核心区域，用来放置Java对象实例，几乎所有创建的Java对象实例都是被直接分配在堆上。堆被所有的线程共享，在虚拟机启动时，我们指定的"Xmx"之类参数就是用来指定最大堆空间等指标。（（编译器通过逃逸分析，确定对象是在栈上分配还是在堆上分配）理所当然，堆也是垃圾收集器重点照顾的区域，所以堆内空间还会被不同的垃圾收集器进行进一步的细分，最有名的就是新生代、老年代的划分。

- **方法区**

  第四，方法区（Method Area）。这也是所有线程共享的一块内存区域，用于存储所谓的元(Meta）数据，例如类结构信息，以及对应的运行时常量池、字段、方法代码等。由于早期的HotspotJVM实现，1.7叫做永久代(Permanent Generation）。Oracle JDK 8中将永久代移除，同时增加了元数据区（Metaspace）。

- **运行时常量池（Run-Time Constant Pool)**

  这是方法区的一部分。如果仔细分析过反编译的类文件结构，你能看到版本号、字段、方法、超类、接口等各种信息，还有一项信息就是常量池。Java的常量池可以存放各种常量信息，不管是编译期生成的各种字面量.是需要在运行时决定的符号引用，所以它比一般语言的符号表存储的信息更加宽泛。

- **本地方法栈**

  第六，本地方法栈(Native Method Stack）。它和Java虚拟机栈是非常相似的，支持对底层方法的调用（C++）通常用 navite 标记的方法

- **直接内存**

  Java 堆外的、直接向系统申请的内存空间。 可以这样理解，直接内存就是 JVM 以外的机器内存，比如，你有 4G 的内存，JVM 占用了 1G，则其余的 3G 就是直接内存。所以直接内存收到本机器内存的限制，也可能出现 OutOfMemoryError 的异常。 访问直接内存的速度会优于访问 Java 堆内存的速度，读写频繁的场景会考虑使用直接内存。NIO 使用的就是直接内存。

永久代和元数据的区别

1. 永久代必须指定大小限制 ，元数据可以设置，也可以不设置，无上限（受限于物理内存）
2. 字符串常量 1.7 - 永久代，1.8 - 堆

Java内存中哪些是共享的哪些是独有的

`堆、方法区、运行时常量池是线程共享的`

`栈、程序寄存器、本地方法栈是线程独享的`

`直接内存不受 JVM 管理`

Java中堆栈的区别

最主要的区别

栈内存用来存储局部变量和方法调用。而堆内存用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。

独有还是共享

栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。而堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。

空间大小

栈的内容空间远远小于堆，如果使用递归很容易造成StackOverFlowError

什么是Java虚拟机

Java虚拟机是一个可以执行Java字节码的虚拟机进程。Java源文件被编译成能被Java虚拟机执行的字节码文件。Java被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。Java虚拟机让这个变为可能，因为它知道底层硬件平台的指令长度和其他特性。

GC是什么?为什么要有GC?

GC是垃圾收集的意思(Gabage Collection) ,内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的。

Java语言没有提供释放已分配内存的显示操作方法。

什么是垃圾

没有任何引用指向的一个对象或者多个对象（循环引用）就是垃圾

如何找到垃圾

引用计数 -- 无法解决循环垃圾的定位

引用计数算法很简单，它实际上是通过在对象头中分配一个空间来保存该对象被引用的次数。如果该对象被其它对象引用，则它的引用计数加一，如果删除对该对象的引用，那么它的引用计数就减一，当该对象的引用计数为0时，那么该对象就会被回收。

`根可达算法` **-- 根指向不到的都是垃圾**

根搜索算法的基本思路就是通过一系列名为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。

不过要注意的是被判定为不可达的对象不一定就会成为可回收对象。被判定为不可达的对象要成为可回收对象必须至少经历两次标记过程，如果在这两次标记过程中仍然没有逃脱成为可回收对象的可能性,则基本上就真的成为可回收对象了。

GC Roots都包含了哪些

Class、方法区内的静态引用变量、虚拟机栈、本地方法栈、运行时常量池

GC 的触发条件

1、程序调用 system.gc 时可以触发

2、系统自身来决定 GC 触发的时机

JVM 堆内存分代模型

新生代(Eden + survivor0 + survivor1) + 老年代

新生代

Eden 8

survivor0 1

survivor1 1

老年代

`新生代和老年代的比例 1:3`

`Eden和survivor0和survivor1的比例 8:1:1`

新生代中为什么要分为Eden和Survivor?

如果没有Survivor，Eden区每进行一次 `Minor GC`，存活的对象就会被送到老年代。老年代很快被填满，触发 `Major GC`. 老年代的内存空间远大于新生代，进行一次 `Full GC` 消耗的时间比 `Minor GC` 长得多,所以需要分为Eden和Survivor。Survivor的存在意义，就是减少被送到老年代的对象，进而减少Full GC的发生，Survivor的预筛选保证，只有经历 `16` 次Minor GC还能在新生代中存活的对象，才会被送到老年代。 设置两个Survivor区最大的好处就是 `解决了碎片化`，刚刚新建的对象在Eden中，经历一次Minor GC，Eden中的存活对象就会被移动到第一块survivor space S0，Eden被清空；等Eden区再满了，就再触发一次Minor GC，Eden和S0中的存活对象又会被复制送入第二块survivor space S1（这个过程非常重要，因为这种复制算法保证了S1中来自S0和Eden两部分的存活对象占用连续的内存空间，避免了碎片化的发生）

对象如何晋升为老年代

1、刚new出来的对象都在新生代的Eden中，如果对象过大（超过了 `3145728`）会直接进入老年代。

2、当Eden区满了会执行一次 YGC(MinorGC) 回收之后，大多数的对象会被回收，活着的进入s0 ，年龄加1

3、之后再执行 YGC，活着的进入s1，年龄加1

4、再执行 YGC，活着的进入s0，年龄加1

5、年龄足够，进入老年代（15次，CMS是6次）

6、s区装不下，进入老年代

GC 触发流程

新生代满了调用 YGC(MinorGC)，老年代满了调用 OGC(MajorGC)

Full GC 会清理整个堆内存，包括新生代和老年代，速度相当慢

Full GC、Major GC、Minor GC之间区别?

Minor GC

从新生代空间（包括Eden和Survivor区域）回收内存被称为Minor GC。

Major GC

清理Tenured区，用于回收老年代，出现Major GC通常会出现至少一次Minor GC.

Full GC

Full GC是针对整个新生代、老年代、元空间(metaspace，java8以上版本取代permgen）的全局范围的GC。

Major GC通常是跟full GC是等价的，收集整个GC堆。

YGC，Full GC触发条件

> Minor GC 触发条件：当Eden区满时，触发Minor GC。

> Full GC触发条件:
>
> 1. 调用System.gc时，系统建议执行Full Gc，但是不必然执行
> 2. 老年代空间不足
> 3. 方法区空间不足
> 4. 通过Minor GC后进入老年代的平均大小大于老年代的可用内存
> 5. 由Eden区、From Space区向To Sp3ace区复制时，对象大小大于To Space可存，该对象转存到老年代，且老年代的可用内存小于该对象大小

JVM 调优的目的

降低 Full GC 的次数

常见的垃圾回收算法

标记清楚 -- 位置不连续，会产生碎片

> “标记-清除”(Mark-Sweep）算法，如它的名字一样，算法分为“标记"和”清除"两个阶段:首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。

拷贝 -- 没有碎片，浪费空间

> “拷贝 ”(Copying）的收集算法，它将可用内存按容量划分为大小相等的两块每次只使用其中的一块。当这一块的内存用完了,就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

标记压缩 -- 效率太低

> 标记-压缩算法，标记过程仍然与"标记-清除"算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存分代收集算法，“分代收集”(Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

常见的垃圾回收器

1. Serial 年轻代，串行回收
2. PS 年轻代，并行回收
3. ParNew 年轻代，配合CMS的并行回收
4. SerialOld
5. ParallelOld
6. ConcurrentMarkSweep 老年代，并发的， 垃圾回收和应用程序同时运行，降低STW的时间(200ms)
7. G1(10ms)
8. ZGC (1ms) PK C++
9. Shenandoah
10. Eplison

1.8默认的垃圾回收：PS + ParallelOld

JVM 参数分类

> 标准： - 开头，所有的HotSpot都支持
>
> 非标准：-X 开头，特定版本HotSpot支持特定命令
>
> 不稳定：-XX 开头，下个版本可能取消

常见的 JVM 跟踪参数

> 跟踪 GC 日志
>
> - -XX:+PrintGC：最简单的 GC 参数，每一行代表进行了一次 GC。
> - -XX:+PrintGCDetails：最常用的 GC 参数，可以在日志中打印 GC 详细信息。
> - -XX:+PrintHeapAtGC：详细的打印出 GC 前后的堆内存信息。
> - -XX:+PrintGCTimeStamps：输出每次 GC 发生的时间
> - -XX:+PrintGCApplicationConcurrentTime：记录 Java 程序执行时间，表示应用程序在两次垃圾回收之间运行多长时间
> - -XX:+PrintGCApplicationStoppedTime：记录 Java 程序因为 GC 而产生的停顿时间

> 跟踪类加载/卸载信息
>
> - verbose:class：跟踪类加载和卸载

> 跟踪查看虚拟机参数
>
> - -XX:+PrintVMOptions：查看 Java 程序启动时设置的参数。
> - -XX:+PrintCommandLineFlags：打印虚拟机的显示和隐藏参数。
> - -XX:+PrintFlagsFinal：打印所有系统参数的值。

常见的 JVM 调优命令

-XX:+UseParNewGC：设定年轻代垃圾回收器

-XX:+UseConcMarkSweepGC：设定年老代垃圾回收器

1.-Xms：初始堆大小。只要启动，就占用的堆大小。

2.-Xmx：最大堆大小。java.lang.OutOfMemoryError：Java heap这个错误可以通过配置-Xms和-Xmx参数来设置。

3.-Xss：栈大小分配。栈是每个线程私有的区域，通常只有几百K大小，决定了函数调用的深度，而局部变量、参数都分配到栈上。

> 当出现大量局部变量，递归时，会发生栈空间OOM（`java.lang.StackOverflowError`）之类的错误。

4.XX:NewSize：设置新生代大小的绝对值。

5.-XX:NewRatio：设置年轻代和年老代的比值。比如设置为3，则新生代：老年代=1:3，新生代占总heap的1/4。

6.-XX:MaxPermSize：设置元数据区大小。

> `java.lang.OutOfMemoryError:PermGenspace`这个OOM错误需要合理调大`PermSize`和`MaxPermSize`大小。

7.-XX:SurvivorRatio：年轻代中Eden区与两个Survivor区的比值。注意，Survivor区有form和to两个。比如设置为8时，那么eden:form:to=8:1:1。

8.-XX:HeapDumpOnOutOfMemoryError：发生OOM时转储堆到文件，这是一个非常好的诊断方法。

9.-XX:HeapDumpPath：导出堆的转储文件路径。

10.-XX:OnOutOfMemoryError：OOM时，执行一个脚本，比如发送邮件报警，重启程序。后面跟着一个脚本的路径。

常用的JVM性能调优工具

jps(Java Virtual Machine Process Status Tool)

jps主要用来输出JVM中运行的进程状态信息。语法格式如下：

```
jps [options] [hostid]
```

如果不指定hostid就默认为当前主机或服务器。

命令行参数选项说明如下：

```
-q 不输出类名、Jar名和传入main方法的参数

-m 输出传入main方法的参数

-l 输出main类或Jar的全限名

-v 输出传入JVM的参数
```

比如下面：

```
root@ubuntu:/# jps -m -l
2458 org.artifactory.standalone.main.Main /usr/local/artifactory-2.2.5/etc/jetty.xml
29920 com.sun.tools.hat.Main -port 9998 /tmp/dump.dat
3149 org.apache.catalina.startup.Bootstrap start
30972 sun.tools.jps.Jps -m -l
8247 org.apache.catalina.startup.Bootstrap start
25687 com.sun.tools.hat.Main -port 9999 dump.dat
21711 mrf-center.jar
```

jstack

jstack主要用来查看某个Java进程内的线程堆栈信息。语法格式如下：

```
jstack [option] pid
jstack [option] executable core
jstack [option] [server-id@]remote-hostname-or-ip
```

命令行参数选项说明如下：

```
-l long listings，会打印出额外的锁信息，在发生死锁时可以用jstack -l pid来观察锁持有情况-m mixed mode，不仅会输出Jav
```

jstack可以定位到线程堆栈，根据堆栈信息我们可以定位到具体代码，所以它在JVM性能调优中使用得非常多。下面我们来一个实例找出某个Java进程中最耗费CPU的Java线程并定位堆栈信息，用到的命令有ps、top、printf、jstack、grep。

第一步先找出Java进程ID，我部署在服务器上的Java应用名称为mrf-center：

```
root@ubuntu:/# ps -ef | grep mrf-center | grep -v grep
root     21711     1  1 14:47 pts/3    00:02:10 java -jar mrf-center.jar
```

得到进程ID为21711，第二步找出该进程内最耗费CPU的线程，可以使用ps -Lfp pid或者ps -mp pid -o THREAD, tid, time或者top -Hp pid，我这里用第三个，输出如下：

```
top - 17:10:22 up 59 days,  1:56,  1 user,  load average: 0.05, 0.08, 0.06
Tasks:  36 total,   0 running,  36 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.8%us,  0.3%sy,  0.0%ni, 98.7%id,  0.2%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8075600k total,  7799876k used,   275724k free,    77260k buffers
Swap: 25061368k total, 15506616k used,  9554752k free,   865340k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
21742 root      20   0 4185m  95m  10m S    1  1.2   1:12.24 java
21711 root      20   0 4185m  95m  10m S    0  1.2   0:00.00 java
21712 root      20   0 4185m  95m  10m S    0  1.2   0:02.22 java
21713 root      20   0 4185m  95m  10m S    0  1.2   0:00.18 java
21714 root      20   0 4185m  95m  10m S    0  1.2   0:00.16 java
21715 root      20   0 4185m  95m  10m S    0  1.2   0:00.17 java
21716 root      20   0 4185m  95m  10m S    0  1.2   0:00.16 java
21717 root      20   0 4185m  95m  10m S    0  1.2   0:00.59 java
21718 root      20   0 4185m  95m  10m S    0  1.2   0:00.00 java
21719 root      20   0 4185m  95m  10m S    0  1.2   0:00.00 java
21720 root      20   0 4185m  95m  10m S    0  1.2   0:00.00 java
21721 root      20   0 4185m  95m  10m S    0  1.2   0:03.25 java
21722 root      20   0 4185m  95m  10m S    0  1.2   0:03.42 java
21723 root      20   0 4185m  95m  10m S    0  1.2   0:00.00 java
21724 root      20   0 4185m  95m  10m S    0  1.2   0:01.89 java
21727 root      20   0 4185m  95m  10m S    0  1.2   0:01.19 java
21728 root      20   0 4185m  95m  10m S    0  1.2   0:00.02 java
21730 root      20   0 4185m  95m  10m S    0  1.2   0:00.24 java
21731 root      20   0 4185m  95m  10m S    0  1.2   0:00.24 java
21732 root      20   0 4185m  95m  10m S    0  1.2   0:00.24 java
21733 root      20   0 4185m  95m  10m S    0  1.2   0:00.24 java
21734 root      20   0 4185m  95m  10m S    0  1.2   0:00.24 java
21735 root      20   0 4185m  95m  10m S    0  1.2   0:00.24 java
21736 root      20   0 4185m  95m  10m S    0  1.2   0:00.24 java
21737 root      20   0 4185m  95m  10m S    0  1.2   0:00.23 java
21738 root      20   0 4185m  95m  10m S    0  1.2   0:00.24 java
21739 root      20   0 4185m  95m  10m S    0  1.2   0:00.23 java
21740 root      20   0 4185m  95m  10m S    0  1.2   0:04.42 java
21741 root      20   0 4185m  95m  10m S    0  1.2   0:00.79 java
21743 root      20   0 4185m  95m  10m S    0  1.2   0:00.78 java
21744 root      20   0 4185m  95m  10m S    0  1.2   0:00.02 java
21745 root      20   0 4185m  95m  10m S    0  1.2   0:00.00 java
21746 root      20   0 4185m  95m  10m S    0  1.2   0:00.00 java
```

TIME列就是各个Java线程耗费的CPU时间，CPU时间最长的是线程ID为21742的线程，用

```
printf "%x\n" 21742
```

得到21742的十六进制值为54ee，下面会用到。

OK，下一步终于轮到jstack上场了，它用来输出进程21711的堆栈信息，然后根据线程ID的十六进制值grep，如下：

```
root@ubuntu:/# jstack 21711 | grep 54ee
"PollIntervalRetrySchedulerThread" prio=10 tid=0x00007f950043e000 nid=0x54ee in Object.wait() [0x00007f94c6eda
```

可以看到CPU消耗在PollIntervalRetrySchedulerThread这个类的Object.wait()，我找了下我的代码，定位到下面的代码：

```
// Idle wait
getLog().info("Thread [" + getName() + "] is idle waiting...");
schedulerThreadState = PollTaskSchedulerThreadState.IdleWaiting;
long now = System.currentTimeMillis();
long waitTime = now + getIdleWaitTime();
long timeUntilContinue = waitTime - now;
synchronized(sigLock) {    try {
        if(!halted.get()) {
            sigLock.wait(timeUntilContinue);
        }
    }     catch (InterruptedException ignore) {
    }
}
```

它是轮询任务的空闲等待代码，上面的sigLock.wait(timeUntilContinue)就对应了前面的Object.wait()。

jmap（Memory Map）和jhat（Java Heap Analysis Tool）

jmap用来查看堆内存使用状况，一般结合jhat使用。

jmap语法格式如下：

```
jmap [option] pid
jmap [option] executable core
jmap [option] [server-id@]remote-hostname-or-ip
```

如果运行在64位JVM上，可能需要指定-J-d64命令选项参数。

```
jmap -permstat pid
```

打印进程的类加载器和类加载器加载的持久代对象信息，输出：类加载器名称、对象是否存活（不可靠）、对象地址、父类加载器、已加载的类大小等信息，如下图：

```
root@ubuntu:/# jmap -permstat 21711
Attaching to process ID 21711, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 20.10-b01
8331 intern Strings occupying 761376 bytes.
finding class loader instances ..Finding object size using Printezis bits and skipping over...
Finding object size using Printezis bits and skipping over...
Finding object size using Printezis bits and skipping over...
done.
computing per loader stat ..done.
please wait.. computing liveness.....................................liveness analysis may be inaccurate ...
class_loader    classes bytes   parent_loader   alive?  type

<bootstrap>     1418    8882640   null          live    <internal>
0x0000000784d14628      1       1944      null          dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1410      1       3112    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1368      1       3168    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef14b8      1       3152    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef13a0      1       3184    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1560      1       3216    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784d4b500      0       0       0x0000000784c000b0      dead    java/util/ResourceBundle$RBClassLoader@0x000000077fdd35d0
0x0000000784ef1528      1       1944      null          dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef14f0      1       3336    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784d14548      1       1944      null          dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef12c0      1       1944      null          dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1598      1       1944      null          dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784f87b08      1       3200    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784d14580      1       1944      null          dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784c000f8      8       49544     null          live    sun/misc/Launcher$ExtClassLoader@0x000000077fbd1ff8
0x0000000784fa5e98      1       3112    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784fa5b28      1       3128    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1480      1       3112    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784c000b0      2362    14145024        0x0000000784c000f8      live    sun/misc/Launcher$AppClassLoader@0x000000077fc40b90
0x0000000784d14698      1       1944    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1260      1       1960      null          dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1330      1       3104      null          dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1448      1       3136    0x0000000784c000b0      dead    sun/reflect/DelegatingClassLoader@0x000000077fa675e8
```

使用jmap -heap pid查看进程堆内存使用情况，包括使用的GC算法、堆配置参数和各代中堆内存使用情况。比如下面的例子：

```
root@ubuntu:/# jmap -heap 21711
Attaching to process ID 21711, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 20.10-b01

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
MinHeapFreeRatio = 40
MaxHeapFreeRatio = 70
MaxHeapSize      = 2067791872 (1972.0MB)
NewSize          = 1310720 (1.25MB)
MaxNewSize       = 17592186044415 MB
OldSize          = 5439488 (5.1875MB)
NewRatio         = 2
SurvivorRatio    = 8
PermSize         = 21757952 (20.75MB)
MaxPermSize      = 85983232 (82.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 6422528 (6.125MB)
   used     = 5445552 (5.1932830810546875MB)
   free     = 976976 (0.9317169189453125MB)
   84.78829520089286% used
From Space:
   capacity = 131072 (0.125MB)
   used     = 98304 (0.09375MB)
   free     = 32768 (0.03125MB)
   75.0% used
To Space:
   capacity = 131072 (0.125MB)
   used     = 0 (0.0MB)
   free     = 131072 (0.125MB)
   0.0% used
PS Old Generation
   capacity = 35258368 (33.625MB)
   used     = 4119544 (3.928703308105468 8MB)
   free     = 31138824 (29.69629669189453MB)
```

```
   11.683876009235595% used
PS Perm Generation
   capacity = 52428800 (50.0MB)
   used     = 26075168 (24.867218017578125MB)
   free     = 26353632 (25.132781982421875MB)
   49.7344603515625% used
   ....
```

使用jmap -histo[:live] pid查看堆内存中的对象数目、大小统计直方图，如果带上live则只统计活对象，如下：

```
root@ubuntu:/# jmap -histo:live 21711 | more
num     #instances         #bytes  class name----------------------------------------------
   1:         38445        5597736  <constMethodKlass>
   2:         38445        5237288  <methodKlass>
   3:          3500        3749504  <constantPoolKlass>
   4:         60858        3242600  <symbolKlass>
   5:          3500        2715264  <instanceKlassKlass>
   6:          2796        2131424  <constantPoolCacheKlass>
   7:          5543        1317400  [I
   8:         13714        1010768  [C
   9:          4752        1003344  [B
  10:          1225         639656  <methodDataKlass>
  11:         14194         454208  java.lang.String
  12:          3809         396136  java.lang.Class
  13:          4979         311952  [S
  14:          5598         287064  [[I
  15:          3028         266464  java.lang.reflect.Method
  16:           280         163520  <objArrayKlassKlass>
  17:          4355         139360  java.util.HashMap$Entry
  18:          1869         138568  [Ljava.util.HashMap$Entry;
  19:          2443          97720  java.util.LinkedHashMap$Entry
  20:          2072          82880  java.lang.ref.SoftReference
  21:          1807          71528  [Ljava.lang.Object;
  22:          2206          70592  java.lang.ref.WeakReference
  23:           934          52304  java.util.LinkedHashMap
  24:           871          48776  java.beans.MethodDescriptor
  25:          1442          46144  java.util.concurrent.ConcurrentHashMap$HashEntry
  26:           804          38592  java.util.HashMap
  27:           948          37920  java.util.concurrent.ConcurrentHashMap$Segment
  28:          1621          35696  [Ljava.lang.Class;
  29:          1313          34880  [Ljava.lang.String;
  30:          1396          33504  java.util.LinkedList$Entry
  31:           462          33264  java.lang.reflect.Field
  32:          1024          32768  java.util.Hashtable$Entry
  33:           948          31440  [Ljava.util.concurrent.ConcurrentHashMap$HashEntry;
```

class name是对象类型，说明如下：

```
B  byte
C  char
D  double
F  float
I  int
J  long
Z  boolean
[  数组，如[I表示int[]
[L+类名 其他对象
```

还有一个很常用的情况是：用jmap把进程内存使用情况dump到文件中，再用jhat分析查看。jmap进行dump命令格式如下：

```
jmap -dump:format=b,file=dumpFileName pid
```

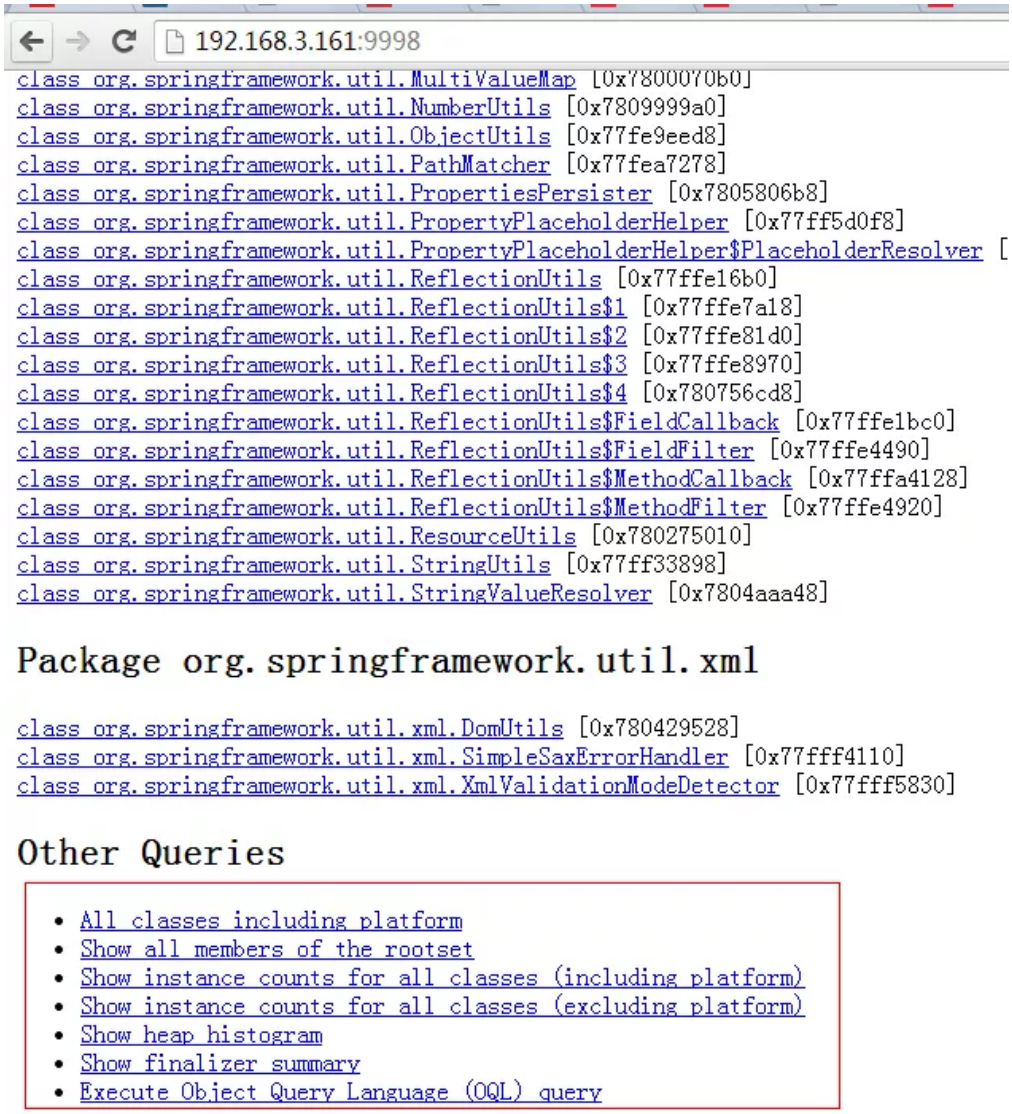
我一样地对上面进程ID为21711进行Dump：

```
root@ubuntu:/# jmap -dump:format=b,file=/tmp/dump.dat 21711
Dumping heap to /tmp/dump.dat ...
Heap dump file created
```

dump出来的文件可以用MAT、VisualVM等工具查看，这里用jhat查看：

```
root@ubuntu:/# jhat -port 9998 /tmp/dump.dat
Reading from /tmp/dump.dat...
Dump file created Tue Jan 28 17:46:14 CST 2014Snapshot read, resolving...
Resolving 132207 objects...
Chasing references, expect 26 dots..........................
Eliminating duplicate references..........................
Snapshot resolved.
Started HTTP server on port 9998Server is ready.
```

注意如果Dump文件太大，可能需要加上-J-Xmx512m这种参数指定最大堆内存，即jhat -J-Xmx512m -port 9998 /tmp/dump.dat。然后就可以在浏览器中输入主机地址:9998查看了：



上面红线框出来的部分大家可以自己去摸索下，最后一项支持OQL（对象查询语言）。

什么是类的加载

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。类的加载的最终产品是位于堆区中的Class对象， Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

类加载器有哪些

启动类加载器

> Bootstrap ClassLoader，负责加载存放在JDK\ire\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

扩展类加载器

> Extension ClassLoader，该加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载DK\ire\Vliblext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。

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

类加载机制

启动类加载器

该加载器使用C++语言实现，属于虚拟机自身的一部分

`加载自带的各种jar包的` （rt等 -- > 核心包）

扩展类加载器

加载javax之类的工具包

其他类加载器(扩展、应用程序、自定义)

Java语言实现，独立于VM外部，并且全部继承抽象类java.lang.ClassLoader，一般包含Extension ClassLoader，AppClassLoader

`加载我们自己写的类` -- > 应用程序加载器

类加载流程

加载

`查找并加载类的二进制数据`

加载是类加载过程的第一个阶段，虚拟机在这一阶段需要完成以下三件事情：

- 通过类的全限定名来获取其定义的二进制字节流
- 将字节流所代表的静态存储结构转化为方法区的运行时数据结构
- 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

验证

`确保被加载的类的正确性`

这一阶段是确保Class文件的字节流中包含的信息符合当前虚拟机的规范，并且不会损害虚拟机自身的安全。包含了四个验证动作：文件格式验证，元数据验证，字节码验证，符号引用验证。

准备

`为类的静态变量分配内存，并将其初始化为默认值`

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。

解析

`把类中的符号引用转换为直接引用`

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

初始化

`类变量进行初始化`

双亲委派模型

如果一个类加载器收到了类请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父加载器去完成，每一层都是如此，因此所有类加载的请求都会传到启动类加载器，只有当父加载器无法完成该请求时，子加载器才去白己加载。

为什么要有双亲委派模型

保证安全：防止修改系统类或者有同包同名类

如何打破双亲委派模型

`打破双亲委派机制则不仅要继承ClassLoader类，还要重写loadClass和 findClass方法。`

Tomcat就打破了双亲委派模型，jsp类加载器，每次修改jsp都会把对应的jsp类加载器整个清除再重新加载，达到不需要重启jvm就能动态的修改jsp代码

Java中类的生命周期

1、加载，查找并加载类的二进制数据，在Java堆中也创建一个java.lang.Class类的对象

2、连接，连接又包含三块内容:验证、准备、初始化。

1、验证，文件格式、元数据节码、符号引用验证

2、准备，为类的静态变量分配内存，并将其初始化为默认值

3、解析，把类中的符号引用转换为直接引用

3、初始化，为类的静态变量赋予正确的初始值

4、使用，new出对象程序中使用

5、卸载，执行垃圾回收

Java中的引用类型有几种?

强引用

> 如果一个对象具有强引用，它就不会被垃圾回收器回收。即使当前内存空间不足，JVM也不会回收它，而是抛出OutOfMemoryError错误，使程序异常终止。如果想中断强引用和某个对象之间的关联，可以显式地将引用赋值为null，这样一来的话，JVM在合适的时间就会回收该对象。

Java中默认声明的就是强引用，比如：

```
Object obj = new Object(); //只要obj还指向Object对象，Object对象就不会被回收
obj = null;  //手动置null
```

软引用

软引用是用来描述一些非必需但仍有用的对象。**在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。**这种特性常常被用来实现缓存技术，比如网页缓存，图片缓存等。

> 在使用软引用时，如果内存的空间足够，软引用就能继续被使用，而不会被垃圾回收器回收;只有在内存空间不足时，软引用才会被垃圾回收器回收

弱引用

> 具有弱引用的对象拥有的生命周期更短暂。因为当JVM进行垃圾回收，一旦发现弱引用对象，无论当前内存空间是否充足，都会将弱引用回收。不过由于垃圾回收器是一个优先级较低的线程，所以并不一定能迅速发现弱引用对象

虚引用

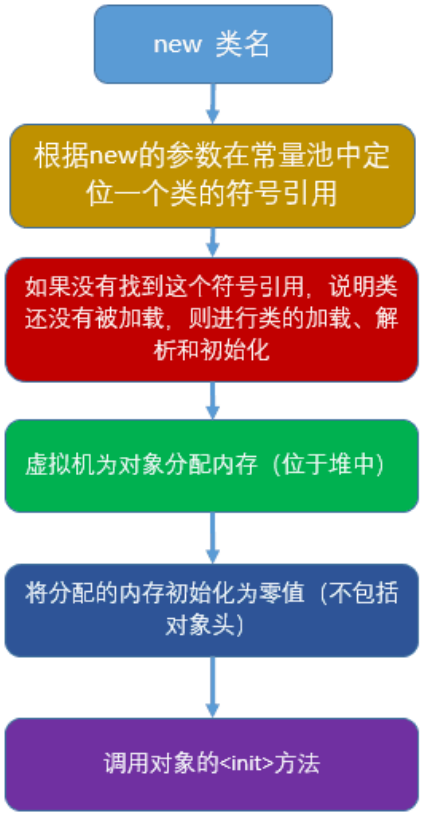
> 顾名思义，就是形同虚设，如果一个对象仅持有虚引用，那么它相当于没有引用，在任何时候都可能被垃圾回收器回收。
>
> 虚引用必须和引用队列关联使用，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

Java对象的创建

关于对象的创建过程一般是从new指令(我说的是JVM的层面)开始的(具体请看图1)，JVM首先对符号引用进行解析，如果找不到对应的符号引用，那么这个类还没有被加载，因此JVM便会进行类加载过程（具体加载过程可参见我的另一篇博文）。符号引用解析完毕之后，JVM会为对象在堆中分配内存，HotSpot虚拟机实现的JAVA对象包括三个部分：对象头、实例字段和对齐填充字段（具体内容请看图2），其中要注意的是，实例字段包括自身定义的和从父类继承下来的（即使父类的实例字段被子类覆盖或者被private修饰，都照样为其分配内存）。相信很多人在刚接触面向对象语言时，总把继承看成简单的“复制”，这其实是完全错误的。JAVA中的继承仅仅是类之间的一种逻辑关系（具体如何保存记录这种逻辑关系，则设计到Class文件格式的知识，具体请看我的另一篇博文），唯有创建对象时的实例字段，可以简单的看成“复制”。

为对象分配完堆内存之后，JVM会将该内存（除了对象头区域）进行零值初始化，这也就解释了为什么JAVA的属性字段无需显示初始化就可以被使用，而方法的局部变量却必须要显示初始化后才可以访问。最后，JVM会调用对象的构造函数，当然，调用顺序会一直上溯到Object类。

至此，一个对象就被创建完毕，此时，一般会有一个引用指向这个对象。


new 类名
根据new的参数在常量池中定位一个类的符号引用
如果没有找到这个符号引用，说明类还没有被加载，则进行类的加载、解析和初始化
虚拟机为对象分配内存（位于堆中）
将分配的内存初始化为零值（不包括对象头）
调用对象的<init>方法