一、总览

1. JVM体系（分为四大块）

(1) 类的加载机制

(2) jvm内存结构

(4) GC算法 垃圾回收

(5) GC分析 命令调优

二、类的加载机制

主要内容：

(1) 什么是类的加载

(2) 类的生命周期

(3) 类加载器

(4) 双亲委派模型

1. 定义

(1) 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。

(2) 类加载器并不需要等到某个类被“首次主动使用”时再加载它，JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误）如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误

1.1 加载.class文件的方式

(1) 从本地系统中直接加载

(2) 通过网络下载.class文件

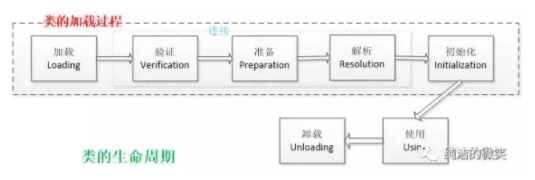
(3) 从zip，jar等归档文件中加载.class文件

(4) 从专有数据库中提取.class文件

(5) 将Java源文件动态编译为.class文件

2. 类的声明周期

2.1 类的生命周期包括这几个部分，加载、连接、初始化、使用和卸载，其中前三部是类的加载的过程,如下图；



(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

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。

④ 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。

(2) 连接：连接又包含三块内容：验证、准备、初始化。

① 验证：确保被加载的类的正确性。

四个步骤：文件格式、元数据、字节码、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；

a. 这时候进行内存分配的仅包括类变量（static），在方法区分配，而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在Java堆中。

b. 这里所设置的初始值通常情况下是数据类型默认的零值（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值。

c. 如果类字段的字段属性表中存在 ConstantValue属性，即同时被final和static修饰，那么在准备阶段变量value就会被初始化为ConstValue属性所指定的值。

③ 解析，把类中的符号引用转换为直接引用

a. 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。

b. 符号引用就是一组符号来描述目标，可以是任何字面量。

c. 直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

(3) 初始化：为类的静态变量赋予正确的初始值

① 声明类变量是指定初始值

② 使用静态代码块为类变量指定初始值

（JVM初始化步骤）

① 假如这个类还没有被加载和连接，则程序先加载并连接该类

② 假如该类的直接父类还没有被初始化，则先初始化其直接父类

③ 假如类中有初始化语句，则系统依次执行这些初始化语句

（类的初始化时机 - 只有当对类的主动使用的时候才会导致类的初始化）

① 创建类的实例，也就是new的方式

② 访问某个类或接口的静态变量，或者对该静态变量赋值

③ 调用类的静态方法

④ 反射（如 Class.forName(“com.shengsiyuan.Test”)）

⑤ 初始化某个类的子类，则其父类也会被初始化

⑥ Java虚拟机启动时被标明为启动类的类（ JavaTest），直接使用 java.exe命令来运行某个主类。

(4) 使用：new出对象程序中使用

(5) 卸载：执行垃圾回收

2.2 Java虚拟机结束生命周期的情况

(1) 执行了 System.exit()方法

(2) 程序正常执行结束

(3) 程序在执行过程中遇到了异常或错误而异常终止

(4) 由于操作系统出现错误而导致Java虚拟机进程终止

2.3 类的加载顺序

(1) 父类静态代码块(包括静态初始化块，静态属性，但不包括静态方法)

(2) 子类静态代码块(包括静态初始化块，静态属性，但不包括静态方法 )

(3) 父类非静态代码块( 包括非静态初始化块，非静态属性 )

(4) 父类构造函数

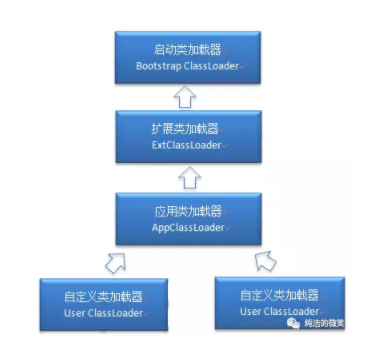
(5) 子类非静态代码块 ( 包括非静态初始化块，非静态属性 )

(6) 子类构造函数

(7) 静态域最先初始化，静态域包括：静态变量，静态方法，静态块，加载顺序由声明顺序决定。

其中：类中静态块按照声明顺序执行，并且(1)和(2)不需要调用new类实例的时候就执行了(意思就是在类加载到方法区的时候执行的)

3. 类加载器



注意：这里父类加载器并不是通过继承关系来实现的，而是采用组合实现的。

(1) 启动类加载器（Bootstrap ClassLoader）

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。

(2) 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\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。

(3) 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

3.1 类加载机制

(1) 全盘负责，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入

(2) 父类委托，先让父类加载器试图加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类

(3) 缓存机制，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效

4. 类的加载方式

(1) 命令行启动应用时候由JVM初始化加载

(2) 通过Class.forName()方法动态加载

(3) 通过ClassLoader.loadClass()方法动态加载

4.1 Class.forName()和ClassLoader.loadClass()区别

(1) Class.forName()：将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static块；

(2) ClassLoader.loadClass()：只干一件事情，就是将.class文件加载到jvm中，不会执行static中的内容,只有在newInstance才会去执行static块

(3) Class.forName(name,initialize,loader)带参函数可控制是否加载static块,指定ClassLoader，初始化时不执行静态块.并且只有调用了newInstance()方法采用调用构造函数，创建类的对象 。

5. 双亲委派模型

5.1 工作流程：

双亲委派模型的工作流程是：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。

5.2 双亲委派机制

(1) 当 AppClassLoader加载一个class时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器ExtClassLoader去完成。

(2) 当 ExtClassLoader加载一个class时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给BootStrapClassLoader```去完成。

(3) 如果 BootStrapClassLoader加载失败（例如在 $JAVA\_HOME/jre/lib里未查找到该class），会使用 ExtClassLoader来尝试加载；

(4) 若ExtClassLoader也加载失败，则会使用 AppClassLoader来加载，如果 AppClassLoader也加载失败，则会报出异常 ClassNotFoundException。

5.3 作用：

(1) 系统类防止内存中出现多份同样的字节码

(2) 保证Java程序安全稳定运行

三、JVM内存结构

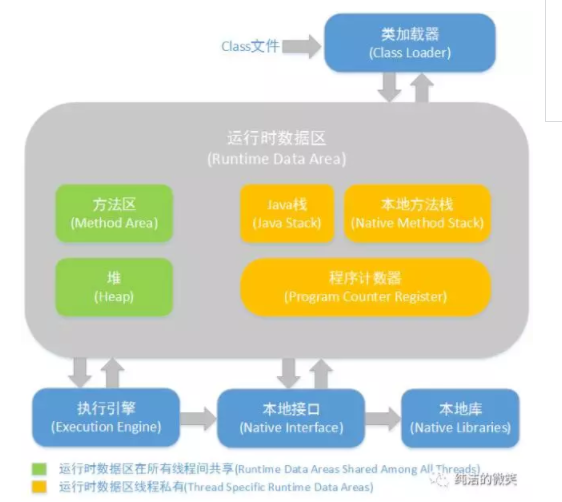
主要内容：

(1) jvm内存结构都是什么

(2) 对象分配规则

1. jvm内存结构

JVM内存结构主要有三大块：堆内存、方法区和栈。



(1) Java堆（Heap）,是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

(2) 方法区（Method Area）,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。方法区会被称为永久代，持久代。

(3) 程序计数器（Program Counter Register）,程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。

此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

(4) JVM栈（JVM Stacks）,与程序计数器一样，Java虚拟机栈（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：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

(5) 本地方法栈（Native Method Stacks）,本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

1.1 堆的内存分配

(1) 堆内存是JVM中最大的一块由年轻代和老年代组成，而年轻代内存又被分成三部分，Eden（伊甸区）空间、From Survivor（幸存者区）空间、To Survivor（幸存者区）空间,默认情况下年轻代按照8:1:1的比例来分配。在年轻代中使用的回收算法是复制算法。

(2) 控制参数

-Xms： 设置堆的最小空间大小。

-Xmx： 设置堆的最大空间大小。

-XX:NewSize： 设置年轻代最小空间大小。

-XX:MaxNewSize： 设置年轻代最大空间大小。

-XX:PermSize： 设置永久代最小空间大小。

-XX:MaxPermSize： 设置永久代最大空间大小。（永久代是方法区）

Xss： 设置每个线程的堆栈大小。

(3) 可以设置堆空间大小和年轻代空间大小两个参数来间接控制老年代大小：

老年代空间大小=堆空间大小-年轻代大空间大小

2. 对象分配规则

(1) 对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。

(2) 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。

(3) 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。

(4) 动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。

(5) 空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Monitor GC,如果false则进行Full GC。

3. OutOfMemoryError异常

(1) Exception in thread “main”: java.lang.OutOfMemoryError: Java heap space

原因：对象不能被分配到堆内存中

(2) Exception in thread “main”: java.lang.OutOfMemoryError: PermGen space

原因：类或者方法不能被加载到老年代。它可能出现在一个程序加载很多类的时候，比如引用了很多第三方的库；

(3) Exception in thread “main”: java.lang.OutOfMemoryError: Requested array size exceeds VM limit

原因：创建的数组大于堆内存的空间

(4) Exception in thread “main”: java.lang.OutOfMemoryError: request <size> bytes for <reason>. Out of swap space

原因：分配本地分配失败。JNI、本地库或者Java虚拟机都会从本地堆中分配内存空间。

(5) Exception in thread “main”: java.lang.OutOfMemoryError: <reason> <stack trace>（Native method

原因：同样是本地方法内存分配失败，只不过是JNI或者本地方法或者Java虚拟机发现

四、GC算法 垃圾回收

主要内容：

(1) 对象存活判断

(2) GC算法

(3) 垃圾回收器

1. 对象存活判断

1.1 判断对象是否存活一般有两种方式：

(1) 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。

(2) 可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索所走过的路径称为引用链。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，不可达对象。

(3) 在Java语言中，GC Roots包括：

① 虚拟机栈中引用的对象。

② 方法区中类静态属性实体引用的对象。

③ 方法区中常量引用的对象。

④ 本地方法栈中JNI引用的对象。

2. GC算法

GC最基础的算法有三种：标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

2.1 标记 -清除算法：“标记-清除”（Mark-Sweep）算法，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。缺点如下：

(1) 一个是效率问题，标记和清除过程的效率都不高；

(2) 另外一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

2.2 复制算法：“复制”（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。这种算法的代价是将内存缩小为原来的一半，持续复制长生存期的对象则导致效率降低。

2.3 标记-压缩算法（老年代采用）：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。

2.4 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。

(1) 在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

3. 垃圾回收器

(1) 如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现。

(2) Minor GC 会清理年轻代的内存

Major GC 是清理老年代。

Full GC 是清理整个堆空间—包括年轻代和老年代。

3.1 Serial收集器，

串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。新生代、老年代使用串行回收；新生代复制算法、老年代标记-压缩；垃圾收集的过程中会Stop The World（服务暂停）

(1) 参数控制： -XX:+UseSerialGC 串行收集器

3.2 ParNew收集器，

(1) ParNew收集器其实就是Serial收集器的多线程版本。

(2) 新生代并行，老年代串行；新生代复制算法、老年代标记-压缩

(3) 参数控制：

-XX:+UseParNewGC ParNew收集器

-XX:ParallelGCThreads 限制线程数量

3.3 Parallel收集器，

(1) Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的吞吐量。

(2) 可以通过参数来打开自适应调节策略，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量；也可以通过参数控制GC的时间不大于多少毫秒或者比例；新生代复制算法、老年代标记-压缩

(3) 参数控制： -XX:+UseParallelGC 使用Parallel收集器+ 老年代串行

3.4 Parallel Old 收集器

(1) Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法。

(2) 参数控制： -XX:+UseParallelOldGC 使用Parallel收集器+ 老年代并行

3.5 CMS收集器（基于“标记-清除”）

(1) CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用都集中在互联网站或B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。

(2) 步骤：

① 初始标记（CMS initial mark）

② 并发标记（CMS concurrent mark）

③ 重新标记（CMS remark）

④ 并发清除（CMS concurrent sweep）

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行。老年代收集器（新生代使用ParNew）

(3) 总结

优点：并发收集、低停顿

缺点： 产生大量空间碎片、并发阶段会降低吞吐量

3.6 G1收集器（标记-整理算法）

(1) G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足GC停顿时间要求的同时,还具备高吞吐量性能特征

① **空间整合**，G1收集器采用标记整理算法，不会产生内存空间碎片。分配大对象时不会因为无法找到连续空间而提前触发下一次GC。

② **可预测停顿**，这是G1的另一大优势，降低停顿时间是G1和CMS的共同关注点，但G1除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为N毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒，这几乎已经是实时Java（RTSJ）的垃圾收集器的特征了。

(2) 使用G1收集器时，Java堆的内存布局与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔阂了，它们都是一部分（可以不连续）Region的集合。G1的新生代收集跟ParNew类似，当新生代占用达到一定比例的时候，开始出发收集。和CMS类似，G1收集器收集老年代对象会有短暂停顿。

4. 常用垃圾收集器组合

|  |  |  |
| --- | --- | --- |
| 新生代GC策略 | 老年老代GC策略 | 说明 |
| Serial | Serial Old | Serial和Serial Old都是单线程进行GC，特点就是GC时暂停所有应用线程。 |
| Serial | CMS+Serial Old | CMS（Concurrent Mark Sweep）是并发GC，实现GC线程和应用线程并发工作，不需要暂停所有应用线程。另外，当CMS进行GC失败时，会自动使用Serial Old策略进行GC。 |
| ParNew | CMS | 使用 -XX:+UseParNewGC选项来开启。ParNew是Serial的并行版本，可以指定GC线程数，默认GC线程数为CPU的数量。可以使用-XX:ParallelGCThreads选项指定GC的线程数。如果指定了选项 -XX:+UseConcMarkSweepGC选项，则新生代默认使用ParNew GC策略。 |
| ParNew | Serial Old | 使用 -XX:+UseParNewGC选项来开启。新生代使用ParNew GC策略，年老代默认使用Serial Old GC策略。 |
| Parallel Scavenge | Serial Old | Parallel Scavenge策略主要是关注一个可控的吞吐量：应用程序运行时间 / (应用程序运行时间 + GC时间)，可见这会使得CPU的利用率尽可能的高，适用于后台持久运行的应用程序，而不适用于交互较多的应用程序。 |
| Parallel Scavenge | Parallel Old | Parallel Old是Serial Old的并行版本 |
| G1GC | G1GC | -XX:+UnlockExperimentalVMOptions -XX:+UseG1GC #开启； -XX:MaxGCPauseMillis=50 #暂停时间目标； -XX:GCPauseIntervalMillis=200 #暂停间隔目标； -XX:+G1YoungGenSize=512m #年轻代大小； -XX:SurvivorRatio=6 #幸存区比例 |

五、GC分析 命令调优

主要内容：

1. GC日志分析

2. 调优命令

3. 调优工具

1. GC日志分析

(1) Young GC回收日志:

2016-07-05T10:43:18.093+0800: 25.395:

[GC[PSYoungGen: 274931K->10738K(274944K)] 371093K->147186K(450048K), 0.0668480 secs] [Times: user=0.17 sys=0.08, real=0.07 secs]

(2) Full GC回收日志:

2016-07-05T10:43:18.160+0800: 25.462:

[FullGC[PSYoungGen: 10738K->0K(274944K)]

[ParOldGen: 136447K->140379K(302592K)] 147186K->140379K(577536K)

[PSPermGen: 85411K->85376K(171008K)], 0.6763541 secs]

[Times: user=1.75 sys=0.02, real=0.68 secs]

(3) PSYoungGen、ParOldGen、PSPermGen属于Parallel收集器。其中PSYoungGen表示gc回收前后年轻代的内存变化；ParOldGen表示gc回收前后老年代的内存变化；PSPermGen表示gc回收前后永久区的内存变化。

(4) Young GC 主要是针对年轻代进行内存回收比较频繁，耗时短；

Full GC 会对整个堆内存进行回收，耗时长，因此一般尽量减少full gc的次数

2. 调优命令

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

(1) jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。

(2) jstat，JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

(3) jmap，JVM Memory Map命令用于生成heap dump文件

(4) jhat，JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看

(5) jstack，用于生成java虚拟机当前时刻的线程快照。

(6) jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

3. 调优工具

常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

(1) jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控

(2) jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。

(3) MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗

(4) GChisto，一款专业分析gc日志的工具