JVM

[1. 运行时数据区域 1](#_101)

[1.1. 虚拟机栈 1](#_102)

[1.2. 本地方法栈 1](#_103)

[1.3. 程序计数器 1](#_104)

[1.4. 堆 1](#_105)

[1.4.1. 新生代，对象招生梦死 1](#_110)

[1.4.2. 老年代 1](#_111)

[1.5. 方法区 1](#_107)

[1.5.1. 1.7永久代 1](#_108)

[1.5.2. 1.8元空间 1](#_109)

[1.6. 分配实例 2](#_188)

[1.6.1. 分配结论 3](#_189)

[1.7. 常量池分配 3](#_190)

[1.7.1. 字符串常量池 3](#_191)

[1.7.2. class文件常量池 4](#_192)

[1.7.3. 运行时常量池 4](#_193)

[2. 垃圾收集 4](#_112)

[2.1. 判断是否可被收集 4](#_115)

[2.1.1. 引用计数 4](#_116)

[2.1.2. GC Root可达性分析 5](#_117)

[2.2. 方法区回收，主要是类卸载 5](#_122)

[2.2.1. 该类实例被回收 5](#_123)

[2.2.2. ClassLoader被回收 5](#_124)

[2.2.3. 没有对Class对象的引用 5](#_125)

[2.3. 堆回收 5](#_127)

[2.3.1. 垃圾收集算法 5](#_132)

[2.3.2. 垃圾收集器 5](#_133)

[2.3.3. GC 8](#_152)

[3. 对象内存布局 9](#_169)

[3.1. 对象头 9](#_170)

[3.1.1. synchronized 10](#_173)

[3.1.2. static synchronized 11](#_179)

[3.2. 实例数据 11](#_171)

[3.3. 对齐 11](#_172)

[4. 类加载器 11](#_162)

[4.1. 普通Java中 11](#_174)

[4.1.1. 启动类加载器 11](#_163)

[4.1.2. 扩展类加载器 11](#_164)

[4.1.3. 应用类加载器 11](#_165)

[4.1.4. 自定义类加载器 12](#_166)

[4.2. OSGi 12](#_176)

[5. 问题 12](#_167)

[5.1. JVM调优 12](#_168)

[5.2. CAS问题 12](#_185)

[5.3. 重排序 12](#_186)

[5.4. 单例模式 13](#_187)

[5.5. 默认垃圾回收器 13](#_200)

[6. JVM调优 13](#_194)

[6.1. Java内存泄露 13](#_195)

[6.2. OOM发生后情形 14](#_196)

[6.3. jps在win10上不能生效 14](#_197)

[6.4. OOM常见原因 14](#_198)

[6.5. jmap使用 15](#_199)

[7. 类加载 15](#_153)

[7.1. 加载 15](#_154)

[7.2. 验证 15](#_155)

[7.3. 准备 15](#_156)

[7.4. 解析 15](#_157)

[7.5. 初始化 15](#_158)

[7.6. 使用 15](#_160)

[7.7. 卸载 15](#_161)

[8. 引用类型 16](#_126)

[9. JMM 16](#_180)

[9.1. 何时刷新工作内存 16](#_181)

[9.2. volatile 17](#_182)

[9.3. happens-before规则 18](#_183)

[9.4. 处理器实现原子操作 18](#_184)

[10. 调优 18](#_201)

[10.1. jstat 18](#_202)

[10.2. jvisualvm 19](#_203)

[10.3. jinfo 19](#_206)

[10.4. jstack 19](#_207)

[10.5. 定位OOM 19](#_208)

# 运行时数据区域

## 虚拟机栈

JAVA方法执行的内存模型；每个方法在执行时会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等。

## 本地方法栈

为Native方法服务。

## 程序计数器

当前线程所执行的字节码的行号指示器。

## 堆

存放对象实例。

### 新生代，对象招生梦死

#### 1个Eden

#### 2个Survivor

### 老年代

老年代和新生代内存比例为2:1

## 方法区

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。

### 1.7永久代

### 1.8元空间

移除“元空间”的原因：

方法区大小难以设定，容易发生内存溢出。永久代会存放Class的相关信息，一般这些信息在编译期间就能确定大小。但是如果是在一些需要动态生成大量Class的应用中，如：Spring的动态代理、大量的JSP页面或动态生成JSP页面等，由于方法区的大小在一开始就要分配好，因此就能难确定大小，容易出现内存溢出

GC复杂且效率低。方法区存储了类的元数据信息和各种常量，它的内存回收目标理应当是对这些类型的卸载和常量的回收。但由于这些数据被类的实例引用，卸载条件变得复杂且严格，回收不当会导致堆中的类实例失去元数据信息和常量信息。因此，回收方法区内存不是一件简单高效的事情。

促进HotSpot JVM与JRockit VM的融合。JRockit没有方法区，移除永久代可以促进HotSpot JVM与JRockit VM的融合。

元空间的特点：

每个加载器有专门的存储空间。

不会单独回收某个类。

元空间里的对象的位置是固定的。

如果发现某个加载器不再存活了，会把相关的空间整个回收

## 分配实例

class Fruit {

static int x = 10;

static BigWaterMelon bigWaterMelon\_1 = new BigWaterMelon(x);

int y = 20;

BigWaterMelon bigWaterMelon\_2 = new BigWaterMelon(y);

public static void main(String[] args) {

final Fruit fruit = new Fruit();

int z = 30;

BigWaterMelon bigWaterMelon\_3 = new BigWaterMelon(z);

new Thread() {

@Override

public void run() {

int k = 100;

setWeight(k);

}

void setWeight(int waterMelonWeight) {

fruit.bigWaterMelon\_2.weight = waterMelonWeight;

}

}.start();

}

}

class BigWaterMelon {

public BigWaterMelon(int weight) {

this.weight = weight;

}

public int weight;

}

### 分配结论

首先需要区分两类变量的分配情况：

原始数据类型变量：其“变量分配”和“数据分配”是在一起的。

引用数据类型变量：其“变量分配”和“数据分配”不一定是在一起的。

所有的对象实例都会放在堆中（无论是静态变量、局部变量、成员变量引用的），当然也有可能经过逃逸分析发现不会出栈的，会直接在栈中进行分配。

## 常量池分配

### 字符串常量池

在 HotSpot VM 里实现的 string pool 功能的是一个 StringTable 类，它是一个 Hash 表，默认值大小长度是1009；里面存的是驻留字符串的引用（而不是驻留字符串实例自身）。也就是说某些普通的字符串实例被这个 StringTable 引用之后就等同被赋予了“驻留字符串”的身份。这个 StringTable 在每个 HotSpot VM 的实例里只有一份，被所有的类共享。

注意：字符串常量池和其中引用的字符串实例都在堆中

具体的创建过程是：

在hotspot中，运行时常量池中：

CONSTANT\_Utf8：是Symbol\*类型的

CONSTANT\_String：java.lang.String，是一个实际Java对象的引用

其中CONSTANT\_Utf8会在类加载的过程中就全部创建出来，而CONSTANT\_String则是懒加载的。例如在第一次引用该项的ldc指令才会被resolve。resolve过后，这个项的才会最终变为JVM\_CONSTANT\_String。

另外对于Hotspot VM来说，加载类的时候，字符串字面量会进入到当前类的运行时常量池，不会进入全局字符串常量池（即在StringTable中没有相应的引用，在堆中也没有对应的对象产生）。在resolve时，才会去查询全局字符串池，最后把符号引用替换为直接引用，可能会创建对象，并在字符串常量池中驻留引用。

### class文件常量池

我们都知道，class 文件中除了包含类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池(constant pool table)，用于存放编译器生成的各种字面量(Literal)和符号引用(Symbolic References)。

字面量：比较接近 Java 语言层面常量的概念，如文本字符串、被声明为 final 的常量值等。

符号引用：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（它与直接引用区分一下，直接引用一般是指向方法区的本地指针，相对偏移量或是一个能间接定位到目标的句柄）

### 运行时常量池

JVM 在执行某个类的时候，必须经过加载、连接、初始化，而连接又包括验证、准备、解析（resolve）三个阶段。而当类加载到内存中后，JVM 就会将 class 文件常量池中的内容存放到运行时常量池中，由此可知，运行时常量池也是每个类都有一个。在上面也说了，class 常量池中存的是字面量和符号引用，也就是说它们存的并不是对象的实例，而是对象的符号引用值。而经过resolve 之后，也就是把符号引用替换为直接引用，解析的过程会去查询全局字符串池，也就是上面所说的 StringTable，以保证运行时常量池所引用的字符串与全局字符串池中所引用的是一致的。

注意：runtime constant pool（而不是interned string pool / StringTable之类的其他东西）的话，其中的引用类型常量（例如CONSTANT\_String、CONSTANT\_Class、CONSTANT\_MethodHandle、CONSTANT\_MethodType之类）都存的是引用，实际的对象还是存在Java heap上的。

# 垃圾收集

## 判断是否可被收集

### 引用计数

### GC Root可达性分析

#### 虚拟机栈中的对象

#### 本地方法栈中的对象

#### 方法区静态属性引用对象

所引用的对象具体存放在堆中。

#### 方法区常量对象

## 方法区回收，主要是类卸载

### 该类实例被回收

### ClassLoader被回收

### 没有对Class对象的引用

## 堆回收

### 垃圾收集算法

#### 复制

#### 标记-整理

#### 标记-清除

#### 分代收集

### 垃圾收集器

#### 新生代

##### Serial：需要STW

##### ParNew：Serial的多线程版本

##### Parallel Scavenge收集器

吞吐量优先

吞吐量 = 运行用户代码时间 / (运行用户代码时间 + 垃圾收集时间）

#### 老年代

##### Serial Old

##### Parallel Old

##### CMS：最短停顿时间

1. 初始阶段：

标记老年代中所有GC Root对象

标记年轻代中活着的对象引用到的老年代对象

2. 并发标记：执行GC Root Tracing的过程。

标记变更的对象

预清理阶段：处理前一根阶段因为引用关系改变导致没有标记到的存活对象的。

可终止的预处理：尝试最终标记阶段足够多的工作，期待能够发生一次ygc，清理年轻代的引用，减少最终标记阶段花费的时间。

3. 重新标记：修正并发标记期间因用户程序继续运行而导致标记产生变动的那部分对象的标记记录，完成标记整个老年代的所有存活对象

4. 并发清理：清除没有标记的对象并且回收空间。

使用场景为互联网站或者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。

###### 初始阶段：STW

初始标记仅标记一下GC Roots能直接关联到的对象，速度很快

###### 并发标记

(收集垃圾跟用户线程一起执行) 初始标记和重新标记任然需要“stop the world”，并发标记过程就是进行GC Roots Tracing的过程；

###### 重新标记：STW

重新标记 (Stop the World事件 CPU停顿，比初始标记稍微长，远比并发标记短)修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记时间短

###### 并发清除

###### 优缺点

**优点**

1. 并发收集，低停顿

**缺点**

1. CMS收集器对CPU资源非常敏感

在并发阶段，虽然不会导致用户线程停顿，但是会因为占用一部分线程使应用程序变慢，总吞吐量会降低

2. CMS无法处理浮动垃圾

CMS在并发清理阶段线程还在运行，伴随着程序的运行自然也会产生新的垃圾，这一部分垃圾产生在标记过程之后，CMS无法在当次过程中处理，只有等到下次GC的时候清理。

3. CMS基于标记-清除

在收集结束时候会有大量空间碎片产生。空间碎片太多，会给大对象的分配带来很大麻烦。

#### G1

##### 以Region为单位

##### 初始标记

标记一下GC Roots能直接关联到的对象，并且修改TAMS(Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的Region中创建新对象。

##### 并发标记

##### 最终标记

##### 筛选回收，STW

对各个Region的回收价值和成本来进行排序，根据**用户所期望的GC停顿时间**来指定回收计划。

##### 特点

1、并行于并发

G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。

2、分代收集

虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。它能够采用不同的方式去处理新创建的对象和已经存活了一段时间，熬过多次GC的旧对象以获取更好的收集效果。

3、空间整合

与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。

4、可预测的停顿

这是G1相对于CMS的另一个大优势，降低停顿时间是G1和CMS共同的关注点，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M毫秒的时间片段内，

### GC

#### GC分类

针对HotSpot VM的实现，它的GC其实只有两种：

1. Partial GC：并不收集整个堆

Young GC：只收集年轻代

Old GC：只收集老年代，只有CMS的concurrent collection是这个模式

Mixed GC：收集整个年轻代和部分老年代。只有G1有这个模式。

2. Full GC：收集年轻代、老年代和永久代（如果存在的话）等所有部分的模式。

#### GC流程

1. 当新对象被创建时，首先会被分配到Eden区，若Eden区空间不够，则触发Minor GC。

2. 在正式Minor GC前，会检查新生代中对象是否比老年代中剩余空间大还是小。

老年代剩余空间大于新生代对象大小，直接Minor GC，如果Survivor区不够放，也有老年代担着。

老年代剩余空间小于新生代对象大小，就要看是否启用了“老年代空间分配担保规则”。这个规则是：如果老年代中剩余空间大小大于历次Minor GC后剩余下来的对象大小，那就允许进行Minor GC（以前放得下，那么这次应该也能放得下）；如果剩余空间小于历次Minor GC后剩余下来的对象大小，那么先进行Full GC，把老年代空出来再检查。

3. Minor GC后会有三种情况：

Minor GC后的对象足够放到Survivor，则皆大欢喜，GC结束。

Minor GC后的对象不够放到Survivor，但是老年代能放下，那也可以。

Minor GC后的对象Survivor和老年代都放不下，只能Full GC。如果Full GC之后，老年代仍然放不下，就只能OOM。

当然，如果在Minor GC检查时进行了两次Full GC，那么也会OOM。

# 对象内存布局

## 对象头

对象头信息是与对象自身定义的数据无关的额外存储成本，主要包括自身对象运行时数据（如哈希码，GC分代年龄等，这部分称为Mark Word）和类型指针（指向方法区类型数据的指针），另外如果对象是数组的话，还会额外存储数组的长度。

在64bit机器上，Mark Word（8B） + 实例指针（4B） + [数组长度（4B）]

以下是无锁情况下的Mark Word：

unused(25b)+hashcode(31b)+unused(1b)+age(4b)+biased\_lock(1b)+lock(2b)

对于锁情况主要关注倒数3个位：

0 00 轻量级锁

0 01 无锁

0 10 重量级锁

0 11 gc标志

1 01 偏向锁

### synchronized

锁升级过程依赖于Mark Word，主要依赖于biased\_lock和lock。

偏向锁：大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低，引入了偏向锁。通过改变biased\_lock为1，并将获取到该锁的Thread Id记录到Mark Word中。当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程。适用于没有锁竞争的场合。 具体的Mark Word如下

thread(54bit)+epoch(2b)+unused(1b)+age(4)+biased\_lock(1b,此时为1)+lock(2b,此时为01)

偏向锁秘密：因为jvm 在启动的时候需要加载资源，这些对象加上偏向锁没有任何意义，可以减少了大量偏向锁撤销的成本；所以默认就把偏向锁延迟了4000ms。

-XX:BiasedLockingStartupDelay=0修改偏向锁延迟

-XX:-UseBiasedLocking=false关闭偏向锁

偏向锁撤销：偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态。如果处于锁定状态，那么将其升级为轻量级锁（00），然后唤醒原持有偏向锁线程；如果未锁定，则恢复到无锁（01）状态，由当前申请锁的线程重新进行CAS。

轻量级锁：偏向锁失败后，升级为轻量级锁（1.6后加入）核心思想是“对绝大部分的锁，在整个同步周期内都不存在竞争”。虚拟机首先在当前线程的栈帧中建立一个名为锁记录的空间，用于存放锁对象当前Mark Word的拷贝，然后使用CAS尝试将锁对象的Mark Word更新为指向Lock Record的指针。如果更新成功，则轻量级锁获取成功，记录锁状态为轻量级锁。具体的Mark Word如下：

ptr\_to\_lock\_record(62b) + lock(2b，此时为00）

自旋：轻量级锁获取失败后，虚拟机为了避免线程真实地在操作层面挂起，还会进行自旋锁优化。基于“在绝大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失”。JVM会让当前线程执行50-100个循环，没办法就只能升级为重量级锁了。

重量级锁：创建一个monitor对象（本质依赖于操作系统的mutex），把这个对象的地址更新到Mark Word当中。其Mark Word如下：

ptr\_to\_heavyweight\_monitor(62b) + lock(2b，此时为10）

锁消除：JIT编译时，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间。

### static synchronized

锁住的是类，而不是对象

## 实例数据

## 对齐

# 类加载器

1. 隔离，如OSGi

2. Tomcat容器，每个WebApp都有自己的ClassLoader，加载每个WebApp的ClassPath路径上的类，一旦遇到Tomcat自带的Jar包就委托给CommonClassLoader。

3. 对于一些核心类库，可能会把字节码加密，加载的时候就必须进行解密。

默认情况下的ClassLoader：

1. 类A中使用new创建B对象，则使用类A的类加载器去加载类B

2. 可以使用线程上下文类加载器加载类，但是需要显示调用。

## 普通Java中

### 启动类加载器

将存放在<JAVA\_HOME>/lib目录下的，或者被-Xbootclasspath参数所制定的路径中的，并且被虚拟机识别的类库加载到虚拟机内存中去。

### 扩展类加载器

负责加载<JAVA\_HOME>/lib/ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库。

### 应用类加载器

负责加载用户类路径上所指定的类库。

### 自定义类加载器

重写loadClass，会破坏双亲委派模式。

重写findClass，则不会破坏双亲委派模式。

## OSGi

OSGi：Java动态模块化技术，提供热插拔功能，源于其灵活的类加载器架构。

OSGi的类加载器之间只有规则，没有固定的委派关系。各Bundle都会声明依赖的Package和发布的Package。如果有其他的Bundle依赖这个Package，则会由发布该Package的Bundle的类加载器完成加载动作。

# 问题

## JVM调优

jvisualvm可以监控本地进程，观察线程等变化。也可以通过抽样器查看CPU或内存使用情况。

jstack PID：可以得到线程的dump信息（死锁情况等）。

## CAS问题

1. ABA问题：可通过版本号解决

2. 循环时间长开销大。

3. 只能保证一个共享变量的原子操作。

## 重排序

从源代码到指令序列共有三种排序：

1. 编译器优化的重排序：在不改变单线程语义的前提下，可以重新安排语句的执行顺序。

2. 指令级并行重排序：现代处理器采用了指令级并行技术（Instruction-Level Parallelism, ILP)来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

当两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作存在数据依赖性。

3. 内存系统的重排序：由于处理器使用缓存和读/写缓冲区，使得加载和存储操作看上去可能是在乱序执行。

1属于编译器重排序，2和3属于处理器重排序。

## 单例模式

*private static volatile* TestSingleton *instance* = *null*; *private static* TestSingleton getInstance() {  *if* (*instance* == *null*) {  *synchronized*(TestSingleton.*class*) {  *if* (*instance* == *null*) {  *instance* = *new* TestSingleton();  }  }  }  *return instance*; }

以上通过volatile防止初始化的时候指令重排序带来的问题。

*private static class* InstanceHolder {  *private static* TestSingleton *instance* = *new* TestSingleton(); } *private static* TestSingleton getInstance() {  *return* InstanceHolder.*instance*; }

通过类特性（在初始化时会加锁，基于此同步多个线程对同一个类的初始化）

## 默认垃圾回收器

JDK8默认的垃圾回收器组合是Parallel Scavenge + Parallel Old

# JVM调优

## Java内存泄露

Java中虽然有垃圾回收机制，它能够将不再使用的对象，自动从内存中清除。即使这样，java中也存在着内存泄露的情况：

1. 长生命周期的对象持有短生命周期对象的引用

尽管短生命周期的对象已经不再需要，但是长生命周期的对象一直持有它的引用导致其无法被回收。例如：缓存系统，加载一个对象放在缓存系统中，一直不去使用这个对象。

2. 各种连接

比如数据库连接、网络连接和io连接，除非使用close()方法将其连接关闭，不然不会自动被GC回收。

3. 特别注意一些像HashMap、ArrayList集合对象，经常会引发内存泄露

当他们被声明为static时，它们的生命周期就会和应用程序一样长。

4. 单例对象持有外部对象的引用

那么外部对象将不能被jvm正常回收，导致泄露。

## OOM发生后情形

1. 一个进程有3个线程，如果一个线程抛出OOM，其他两个线程还能运行么？

答案是**还能运行**

原因是当一个线程抛出OOM异常后，它所占据的内存资源会全部被释放掉，从而不会影响其他线程的运行。

2. 如果主线程异常抛出了，子线程还能运行么？

答案是还可以，因为线程之间没有父子关系。当然如果其他线程是守护线程，则JVM终止。

## jps在win10上不能生效

在C:\Users\%USER%\AppData\Local\Temp\hsperfdata\_%USER%目录下添加用户，并设置完全控制权限。

## OOM常见原因

1. Java Heap Space溢出

堆内存没有足够空间存放新创建的对象时，就会抛出OOM错误。

请求创建一个超大对象，通常是一个大数组。

超出预期的访问量/数据量，通常是上游系统请求流量飙升，常见于各类促销/秒杀活动，可以结合业务流量指标排查是否有尖状峰值

过度使用Finalizer，对象没有被马上GC

内存泄露

2. GC overhead limit exceeded

当JVM花费98%以上的时间执行GC，但只恢复了不到2%的内存，且该动作连续重复了5次，则会抛出该错误。

3. Metaspace

通常是加载的class数目太多或体积太大

4. Unable to create new native thread

每个线程都需要占用一定的内存空间，当JVM向操作系统底层请求创建一个新的native线程时，如果没有足够的资源分配就会报此错误。

## jmap使用

1. jmap -heap pid

查看pid对应jvm进程的heap映像，会显示Eden, from, to和老年代的内存占用情况。

2. jmap -histo[:live] pid

显示堆中对象的统计信息，如果加了:live，则只计算活动对象

3. jmap -clstats pid

打印类加载器信息

4. jmap -dump:file=lll.dump pid

生成dump文件

# 类加载

## 加载

1. 获取类的二进制字节流。

2. 将字节流表示的静态存储结构转化为方法区运行时存储结构

3. 在内存中生成代表该类的Class对象

## 验证

保证安全

## 准备

为类变量分配内存并“初始化”，而类常量则是直接初始化

## 解析

将常量池的符号引用转化为直接引用

## 初始化

调用手机类变量赋值和静态语句块生成的cinit方法

## 使用

## 卸载

# 引用类型

强引用：最普遍的引用。如果对象具有强引用，那垃圾回收期绝不会回收它。当内存空间不够时，虚拟机宁愿抛出OOM，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

软引用：当内存空间足够时，就不会回收它；如果内存空间不足了，就会回收这些对象的内存。

弱引用：只能存活到下次GC前。

虚引用：虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

通过与引用队列联合使用，当引用对象被GC后，引用会放到引用队列里

ReferenceQueue<Object> queue = *new* ReferenceQueue<>(); *List*<SoftReference<Object>> list = *new* ArrayList<>(); *for* (*int* i = 0; i < 1024 \* 1024; i++) {  list.add(*new* SoftReference<>(*new* Object(), queue));  *if* (queue.poll() != *null*) {  System.*out*.println("Appear reference from " + i + "th");  } }

# JMM

物理内存--主内存

高速缓存--工作内存

处理器--Java线程

## 何时刷新工作内存

只有在以下情况下，才能触发一个线程对字段的更改其他线程可见：

1. 写入线程释放同步锁，会将所有工作内存中的变量写回到主内存中；读取线程获取同步锁，则会刷新所有工作内存中的变量，即重新从主内存获取。synchronized和lock均可，lock通过对volatile修饰的state变量读写实现。（锁的获取-释放）

2. 如果一个字段被申明为volatile，则写入线程会立即将修改后的值（不只是volatile修饰的变量）同步到主内存；读取时，会使工作内存置为无效，接下来从主内存中读取共享变量。（volatile的读-写）

3. 线程第一次访问一个对象的某个字段时，它会看到字段的初始值或来自某个其他线程写入的值。

4. 当一个线程终止时，所有写入的变量都被刷新到主内存。如，现有线程A、B，在B线程中调用A.join()，那么在B中可以保证看到A线程产生的影响。

Java内存模型规定unlock操作前，需要把改变的副本同步回主存，在lock操作之后会清空此变量在工作内存中原先的副本，需要再次从主内存中read-load新的值。volatile的读-写和锁的获取-释放具有相同的内存语义。另外CAS也具有volatile读和volatile写的语义。

## volatile

当变量定义为volatile时，将具备两种特性：可见性和有序性。

instance = new Singleton(); // instance是volatile变量

上述代码在x86处理器下会多出第二行汇编代码(lock addl $0 \* 0, (%esp))，lock前缀的指令在多核处理器下会引发两件事情：

1. 将当前处理器缓存行的数据写回到系统内存中。

2. 将这个写回内存的操作会使在其他CPU里缓存了该内存地址的数据无效。

在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是否过期，当处理器发现自己的缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态。当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存中。

可见性：当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。实际上是通过CPU的lock指令实现的，即强制使本cpu的缓存写入主存中，该写入动作也会引起别的CPU或者别的内核无效化相应的缓存。

有序性：通过禁止指令重排序来实现。具体的是使用StoreStore、StoreLoad等内存屏障（上面的lock指令不是内存屏障，但有着内存屏障的作用，等同StoreLoad）确保指令重排序时不会把后面的指令排到内存屏障之前，也不会把之前的排到后面。

另外volatile还应用在单例双重检查的重排序上。

volatile写开销大于读开销。

## happens-before规则

程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作；

锁定规则：一个unLock操作先行发生于后面对同一个lock操作；

volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作；

传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C；

线程启动规则：Thread对象的start()方法先行发生于此线程的每个动作前；

线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生；

线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行；

对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始；

## 处理器实现原子操作

基本操作：

处理器会自动保证基本内存操作的原子性

复杂操作（跨总线宽度、跨多个缓存行和跨页表等）：

1. 使用总线锁：会使得其他处理器不能操作其他内存地址的数据，开销较大。

2. 使用缓存锁：由缓存一致性机制阻止同时修改两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时，会使缓存行撤销。

3. 当操作的数据不能被缓存在处理器内部，或跨多个缓存行时；当处理器不支持缓存锁定时。这两种情况都会使用总线锁。

# 调优

## jstat

jstat [ option vmid [interval[s|ms] [count]] ]

1. jstat -gcutil 2766

监视内容与-gc基本相同，但输出主要关注已使用空间占总空间的百分比。

## jvisualvm

可分析dump文件，也可监控正在运行的jvm进程，通过抽样器可进行内存和CPU的分析。

## jinfo

jinfo [ option ] pid

1. jinfo -flag CMSInitiatingOccupancyFraction 1444

查询CMSInitiatingOccupancyFraction参数值

## jstack

用于生成虚拟机当前时刻的线程快照。

## 定位OOM

1. 有可能是内存分配确实过小，而正常业务使用了大量内存

使用jmap -heap vmid

2. 某一个对象被频繁申请，却没有释放，内存不断泄漏，导致内存耗尽

jmap -histo:live 10765

3. 某一个资源被频繁申请，系统资源耗尽，例如：不断创建线程，不断发起网络连接

查看进程创建的线程数，以及网络连接数，如果资源耗尽，也可能出现OOM。