链滴

个人整理 - Java 后端面试题 - JVM 篇

作者：valarchie
原文链接：https://ld246.com/article/1583743114636
来源网站：链滴

```
<ul>
<li>标 ★ 号为重要知识点</li>
</ul>
<h2 id="-JVM回收算法和回收器-CMS采用哪种回收算法-怎么解决内存碎片问题-">★JVM 回收算
和回收器，CMS 采用哪种回收算法，怎么解决内存碎片问题？</h2>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">  CMD采用的是标记-清除算法。会导致内存碎片。
</span></span><span class="highlight-line"><span class="highlight-cl">  可打开-XX：+Us
CMSCompactAtFullCollection开关参数（默认打开）在进行Full GC之前整理内存碎片（称为“压
”）；
</span></span><span class="highlight-line"><span class="highlight-cl">  使用-XX:CMSFull
CsBeforeCompaction参数（默认0）设置多少次不带压缩的Full CG之后才进行一次带压缩的Full G
。
</span></span><span class="highlight-line"><span class="highlight-cl">  内存整理无法并
，还需要STW，需要适当调整内存整理的频率，在GC性能与空间利用率之间平衡。
</span></span></code></pre>
<h2 id="-哪些情况会导致Full-GC-">★ 哪些情况会导致 Full GC？</h2>
<p>老年代满、永久代满、CMS 回收失败、从新生代要放入老年代的对象超过剩余空间。</p>
<h2 id="-JVM内存区域如何划分-">★JVM 内存区域如何划分？</h2>
<ul>
<li>
<p>内存区域只是一个划分规范，并不是所有虚拟机都是按照这样做的</p>
</li>
<li>
<p>最新的 java8 内存模型为：程序计数器、本地方法栈、java 虚拟机栈、堆。以及放置在本地内
的元数据区。元数据区即 java7 的永久代。</p>
</li>
<li>
<p>堆：Java 中的堆是用来存储对象本身的以及数组（当然，数组引用是存放在 Java 栈中的），是 J
va 垃圾收集器管理的主要区域。堆是被所有线程共享的，在 JVM 中只有一个堆。</p>
</li>
<li>
<p>虚拟机栈：虚拟机栈中存放的是一个个的栈帧，每个栈帧对应一个被调用的方法，在栈帧中包括
部变量表、操作数栈、指向当前方法所属的类的运行时常量池的引用、方法返回地址和一些额外的附
信息。当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈。当方法执行完
之后，便会将栈帧出栈。</p>
</li>
<li>
<p>本地方法栈：本地方法栈则是为执行本地方法（Native Method）服务的，在 HotSopt 虚拟机
直接就把本地方法栈和 Java 栈合二为一</p>
</li>
<li>
<p>方法区：方法区与堆一样，是被线程共享的区域。方法区存储了类的信息（包括类的名称、方法
息、字段信息）、静态变量、常量以及编译器编译后的代码等。在方法区中有一个非常重要的部分就
运行时常量池，它是每一个类或接口的常量池的运行时表示形式，在类和接口被加载到 JVM 后，对
的运行时常量池就被创建出来。当然并非 Class 文件常量池中的内容才能进入运行时常量池，在运行
间也可将新的常量放入运行时常量池中，比如 String 的 intern 方法。当方法区无法满足内存分配需
时，则抛出 OutOfMemoryError 异常。在 HotSpot 虚拟机中，用永久代来实现方法区，将 GC 分
收集扩展至方法区，但是这样容易遇到内存溢出的问题。JDK1.7 中，已经把放在永久代的字符串常
池移到堆中。JDK1.8 撤销永久代，引入元空间。</p>
</li>
<li>
<p>程序计数器（线程私有）：是当前线程所执行的字节码的行号指示器，每条线程都要有一个独立
程序计数器，这类内存也称为“线程私有”的内存。正在执行 java 方法的话，计数器记录的是虚拟
```

字节码指令的地址（当前指令的地址）。如果还是 Native 方法，则为空。

直接内存：在 JDK1.4 中新加入的 NOI 类，引入了一种基于通道与缓冲区的 I/O 方式，它可以用 Native 函数直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这内存的引用进行操作。

<h2 id="-介绍JVM中7个区域-然后把每个区域可能造成内存的溢出的情况说明">★ 介绍 JVM 中 7 区域，然后把每个区域可能造成内存的溢出的情况说明</h2>

程序计数器：不会出现内存溢出
线程栈：每个线程栈需要一定的内存空间，虚拟机内存极小的情况下可能出现内存溢出。大部分况出现 stackOverFlow 的异常。
本地方法栈：一样有可能出现内存溢出。
方法区：类信息或者静态变量过多的话会导致溢出。
常量池：字符串过多的情况下溢出。
直接内存区：

<h2 id="-哪些属于GC-Root不正常引用或者哪些情况会出现内存泄露-">★ 哪些属于 GC Root 不正引用或者哪些情况会出现内存泄露？</h2>

静态集合中我们已经不需要但没有删除的数据。
未关闭的 IO 流。
内部类持有外部类，由于内部类隐式持有外部类的引用，但此时已经不再需要外部类时，外部类未被回收。
改变哈希值，如果在哈希集合中已存好一个对象，之后修改对象的属性导致哈希值发生改变后，定位不到该对象了，造成内存泄露。（比如 student 一开始 hashcode 为 1，修改属性之后 hashcod 为 2，但是 hash 表中位置没有变，到时候 remove student 的时候是按 hashcode 为 2 去删除，现删除不掉 hashcode 为 1 的位置）
对象的生命周期超过它的使用周期。

<h2 id="-内存溢出的原因">★ 内存溢出的原因</h2>
过多使用了 static 变量；大量的递归或者死循环；大数据项的查询，如返回表的所有记录，应该用分页查询。
栈过大会导致内存占用过多，频繁页交换阻碍效率。

A，HashMap,vector 等容易（静态集合类）， 和应用程序生命周期一样，所引用的所有对象 Object 也不能释放。
B，当集合类里面的对象属性被修改后，再调用 remove()不起作用，hashcode 值发生了改变</l
>
C，其对象 add 监听器，但是往往释放对象时忘记去删除这些监听器
D，各种连接记得关闭
E，内部类的引用
F，调用其他模块，对象作用参数
G，单例模式，持有外部对象引用无法收回。

内存泄露例子
<pre><code class="highlight-chroma"> Vector<String> A = new Vector<String>();
  for(int i = 0; i < 100; i++){
    Object o = new Object ();

    A.add(o);
    o = null;
  }
</code></pre>
内存溢出的例子
<pre><code class="highlight-chroma">StringBuffer b = new StringBuffer ();
for(int i =0; i <
00; i++){
  for(int j =0; i <
100; j++){
    b.append(*);
  }
}
</code></pre>
<h2 id="内存溢出可能原因和解决-">内存溢出可能原因和解决。</h2>
原因可能是：

A，数据加载过多，如 1 次从数据库中取出过多数据
B，集合类中有对对象的引用，用完后没有清空或者集合对象未置空导致引用存在等，使得 JVM
法回收
C，死循环，过多重复对象
D，第三方软件的 bug
E，启动参数内存值设定的过小。

例如方法：修改 JVM 启动参数，加内存(-Xms，-Xmx)；错误日志，是否还有其他错误；代码
查
<h2 id="-请问java中内存泄漏是什么意思-什么场景下会出现内存泄漏的情况-">★ 请问 java 中内存
漏是什么意思？什么场景下会出现内存泄漏的情况？</h2>

内存泄漏定义（memory leak）：一个不再被程序使用的对象或变量还在内存中占有存储空间。
次内存泄漏似乎不会有大的影响，但内存泄漏堆积后的后果就是内存溢出。
内存溢出（out of memory）：指程序申请内存时，没有足够的内存供申请者使用，或者说，给
你一块存储 int 类型数据的存储空间，但是你却存储 long 类型的数据，那么结果就是内存不够用，
时就会报错 OOM,即所谓的内存溢出。

静态集合类：

如 HashMap、LinkedList 等等。如果这些容器为静态变量，那么它们的生命周期与程序一致，
容器中的对象在程序结束之前将不能被释放，从而造成内存泄漏。简单而言，长生命周期的对象持有
生命周期对象的引用，尽管短生命周期的对象不再使用，但是因为长生命周期对象持有它的引用而导
不能被回收。
<ol start="2">
各种连接：

如数据库连接、网络连接和 IO 连接等。在对数据库进行操作的过程中，首先需要建立与数据库
连接，当不再使用时，需要调用 close 方法来释放与数据库的连接。只有连接被关闭后，垃圾回收器
会回收对应的对象。否则，如果在访问数据库的过程中，对 Connection、Statement 或 ResultSet
显性地关闭，将会造成大量的对象无法被回收，从而引起内存泄漏。
<ol start="3">
变量不合理的作用域：

一个变量的定义的作用范围大于其使用范围，很有可能会造成内存泄漏。另一方面，如果没有及

地把对象设置为 null，很有可能导致内存泄漏的发生。
<ol start="4">
内部类持有外部类：

如果一个外部类的实例对象的方法返回了一个内部类的实例对象，这个内部类对象被长期引用了即使那个外部类实例对象不再被使用，但由于内部类持有外部类的实例对象，这个外部类对象将不会垃圾回收，这也会造成内存泄露。
<ol start="5">
改变哈希值：

当一个对象被存储进 HashSet 集合中以后，就不能修改这个对象中的那些参与计算哈希值的字了，否则，对象修改后的哈希值与最初存储进 HashSet 集合中时的哈希值就不同了，在这种情况下即使在 contains 方法使用该对象的当前引用作为的参数去 HashSet 集合中检索对象，也将返回找不对象的结果，这也会导致无法从 HashSet 集合中单独删除当前对象，造成内存泄露。
<ol start="6">
单例模式：

不正确使用单例模式是引起内存泄漏的一个常见问题，单例对象在初始化后将在 JVM 的整个生周期中存在（以静态变量的方式），如果单例对象持有外部的引用，那么这个对象将不能被 JVM 正回收，导致内存泄漏。
<h2 id="-如何避免内存泄漏">★ 如何避免内存泄漏</h2>
未对作废数据内存单元置为 null，尽早释放无用对象的引用，使用临时变量时，让引用变量在推活动域后自动设置为 null，暗示垃圾收集器收集；程序避免用 String 拼接，用 StringBuffer，因为个 String 会占用内存一块区域；尽量少用静态变量（全局不会回收）；不要集中创建对象尤其大对，可以使用流操作；尽量使用对象池，不再循环中创建对象，优化配置；创建对象到单例 getInstance 中，对象无法回收被单例引用；服务器 session 时间设置过长也会引起内存泄漏。
<h2 id="-请写出几段可以导致内存溢出-内存泄漏-栈溢出的代码-">★ 请写出几段可以导致内存溢出内存泄漏、栈溢出的代码？</h2>

内存溢出（数组中不断添加对象）
内存泄漏 (静态变量引用对象、长字符串 Intern、未关闭流)
栈溢出 (无线递归)

<h2 id="-JVM对象的结构-">★JVM 对象的结构？</h2>
对象头（哈希值、gc 年龄、一些锁相关数据）、对象实例数据、对齐填充。
<h2 id="JVM双亲委派机制-">JVM 双亲委派机制？</h2>

类加载每次都递归交给父类去加载，是在加载不到才会交给下层加载
Application ClassLoader --> Extension ClassLoader --> Bootstrap ClassLoader

<h2 id="数组多大放在-JVM-老年代-">数组多大放在 JVM 老年代？</h2>
-XX:PretenureSizeThreshold 参数可以设置超过这个值直接进入老年代。
或者年轻代放不下时就直接进入老年代了。
<h2 id="-GC-有环怎么处理-">★GC 有环怎么处理？</h2>
从 GC roots 分析可达性解决循环引用的问题。
<h2 id="-如果想不被-GC-怎么办-JVM可以作为GC-Root的对象有哪些-">★ 如果想不被 GC 怎么办 JVM 可以作为 GC Root 的对象有哪些？</h2>
虚拟机栈中的对象、本地方法栈中的对象、以及方法区中静态属性引用对象、常量引用对象。</
>
<h2 id="-如果想在-GC-中生存-1-次怎么办">★ 如果想在 GC 中生存 1 次怎么办</h2>
finalize 方法重写,持有自身对象,即可逃逸一次(因 finalize 方法只执行一次)
<h2 id="jvm-如何分配直接内存--new-对象如何不分配在堆而是栈上">jvm 如何分配直接内存， ne 对象如何不分配在堆而是栈上</h2>

-XX:+DoEscapeAnalysis： 表示开启逃逸分析
逃逸分析指的是，在方法中创建的对象被传递出去后，就产生方法逃逸。
而非方法逃逸的变量，有可能在编译器的优化下直接分配到栈上。

<h2 id="-类加载过程--">★ 类加载过程？</h2>
<h4 id="1-加载">1.加载</h4>
<pre><code class="highlight-chroma"><span class="highlight
cl">通过类的全限定名查找到该类的字节码文件，将该字节码文件装载到jvm中，jvm将文件中静态
节码结构转换成
运行时动态数据结
，并在方法区生成一个定义该类的Class对象，作为方法区中该类的各种数据访问的入口。
</code></pre>
<h4 id="2-验证">2.验证</h4>
<pre><code class="highlight-chroma"><span class="highlight
cl">确保该类的字节码文件中所包含的信息是否符合当前虚拟机的要求，不包含有危害虚拟机的信息
主要有四种验证，
文件格式验证，元
据验证排（语义）、字节码验证（防止危害虚拟机），符号引用验证）
</code></pre>
<h4 id="3-准备">3.准备</h4>
<pre><code class="highlight-chroma"><span class="highlight
cl">为类变量分配内存，并设置一个初始值。被final修饰的类变量，该类型会在编译期就已经被分配
确定
</code></pre>
<h4 id="4-解析">4.解析</h4>
<pre><code class="highlight-chroma"><span class="highlight
cl">将常量池中符号间接引用替换成直接引用
</code></pre>
<h4 id="5-初始化">5.初始化</h4>
<pre><code class="highlight-chroma"><span class="highlight
cl">为类变量、静态代码块进行真正初始化（赋值操作）（类的初始化顺序，如果有父类先初始化父
中类变量
和静态代码块，在
始化子类的静态变量、静态代码块’）
</code></pre>
<h2 id="JVM堆中对象是如何创建的-">JVM 堆中对象是如何创建的?</h2>
当遇到 new 指令的时候，检查这个类是否被加载，没被加载的话加载，然后为对象分配内存空
并进行默认初始化，执行方法。
<h2 id="-eden区-Survivor区-">★eden 区，Survivor 区？</h2>
这两个区都是属于新生代。eden 区用于存放那些刚被 new 出来的对象（因为大部分对象都是
生夕死的，所以 eden 区的对象生命周期都比较短暂）。
survivor 区分为两个，一个是 s1,一个是 s2。存在对象的区标记为 from，另外一个空的区标记为
to。对象会在这两区中倒腾，所以 s1 这轮是 from,下一次就是 to。
当 eden 区满的时候，开始清理 eden 区和 from 区，将剩下存活的对象移入 to 区当中去。</p

<h2 id="java虚拟机的主要作用-">java 虚拟机的主要作用？</h2>
主要作用就是解释运行 java 字节码程序消除平台相关性。
<h2 id="-GC中如何判断对象需要被回收-">★GC 中如何判断对象需要被回收？</h2>

可达性分析：通过一些被称为引用链（GC Roots）的对象作为起点，从这些节点开始向下搜索
搜索走过的路径被称为（Reference Chain），当一个对象到 GC Roots 没有任何引用链相连时（即从
C Roots 节点到该节点不可达），则证明该对象是不可用的。
引用计数分析:对象每被引用一次就 +1，这个规则比较简单，但是会出现两个对象互相引用。但
不可达的情况，却没有被回收。

<h2 id="-JVM内存模型是什么-">★JVM 内存模型是什么？ </h2>

JMM(Java Memory Model) 是线程间通信的机制 。线程间共享变量存储在主内存，每个线程都自己的本地内存，存储的是共享变量在本地的副本。
对应于 cpu 中的寄存器（主内存）与高速缓存（本地内存），可以这么理解。

<h2 id="-JVM的线程模型是什么-">★JVM 的线程模型是什么？ </h2>

内核线程(Kernel-Level Thread, KLT) 就是由操作系统内核支持的线程，内核通过操纵调度器(Sceduler)对线程进行调度。程序一般不会直接使用内核线程，而是去使用内核线程的一种高级接口-轻级进程(Light Weight Process, LWP)，轻量级进程就是我们通常意义上所讲的线程。这种轻量级进与内核线程之间 1：1 的关系称为一对一的线程模型

用户线程(User Thread, UT)指的是完全建立在用户空间的线程库上，系统内核不能感知线程存的实现。用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。因此操作比核线程更快速，并可以支持更大的线程数量。这种进程(不是轻量级进程)与用户线程之间 1：N 的关称为一对多的线程模型。

在早期的 JVM 实现当中使用的是用户线程 1：N 的一对多线程模型，在当时 cpu 的核数普遍较。随着 cpu 的性能演进，核数越来越多了，如果继续采用用户线程模型的话，就很难利用 cpu 的多优势。

如果某几个 JVM 线程被映射到一个内核线程后，（1：N 或者 M:N 内存模型）,如果这里面的一 JVM 线程发起系统调用导致内核线程阻塞，那么剩下的几个线程依旧会被阻塞。

所以现今的 JVM 实现采用的是 1:1 的内核线程模型。

<h2 id="-JVM的最大内存限制-">★JVM 的最大内存限制。 </h2>
首先 JVM 内存限制于实际的最大物理内存了 假设物理内存无限大的话 JVM 内存的最大值跟操系统有很大的关系。简单的说就 32 位处理器虽然可控内存空间有 4GB,但是具体的操作系统会给一个制，这个限制一般是 2GB-3GB（一般来说 Windows 系统下为 1.5G-2G Linux 系统 下为 2G-3G） 而 64bit 以上的处理器就不会有限制了。
但是，这里会有一个问题在默认情况下，堆大小在 32G 以下的话 JVM 中的引用会占用 4 个字。这是 JVM 在启动的时候就已经决定了的。如果你去掉了-XX:-UseCompressedOops 选项的话，然也可以在较小的堆上使用 8 字节的引用（但在生产系统中这么做是毫无意义的！）。一旦堆超过了 2G，你就进入到 64 位的世界里了，因此对象引用就只能是 8 字节而非 4 字节了。此时，Java 程序堆中平均会有 20% 的空间是被对象引用占据了。相当于非常浪费，而且垃圾回收时间也会很长。所每个 JVM 实例分配的内存最好控制在 32G 以内。
<h2 id="为什么Java被称作是-平台无关的编程语言--">为什么 Java 被称作是“平台无关的编程语”？ </h2>
<pre><code class="highlight-chroma">因为JVM针对不同的操作系统进行了的编译，编译的结果统一了对字节码文件的执行。
通俗的解释就是：一个中国人（windows平台），还有一个日本人（linux平台），如果要与他们交流的话，我（开发）必须
会说中国话，还得说日本话。这时候一个英国人（JVM虚拟机）同时会中国话和日本话，而这时我让这个英国人当翻译

我只需要会英语就了，英国人会自动将英文翻译成中文和日文。（JVM虚拟机会自动将字节码翻译成平台能理解的操作令）。
</code></pre>
<h2 id="-JVM加载class文件的原理机制-">★JVM 加载 class 文件的原理机制？</h2>

BootstrapLoader：

BootstrapLoard 是用 C++ 语言写的，它是在 Java 虚拟机启动后初始化的，它主要负责加载 %AVA_HOME%/jre/lib,-Xbootclasspath 参数指定的路径以及 %JAVA_HOME%/jre/classes 中的类。t.jar
<ol start="2">
ExtClassLoader：

Bootstraploader 加载 ExtClassLoader,并且将 ExtClassLoader 的父加载器设置为 Bootstrploaer，ExtClassLoader 是用 Java 写的，具体来说就是 sun.misc.Launcher$ExtClassLoader，ExtClassoader 主要加载 %JAVA_HOME%/jre/lib/ext，此路径下的所有 classes 目录以及 java.ext.dirs 系统量指定的路径中类库。
<ol start="3">
AppClassLoader：

Bootstrploader 加载完 ExtClassLoader 后，就会加载 AppClassLoader,并且将 AppClassLoadr 的父加载器指定为 ExtClassLoader。 AppClassLoader 也是用 Java 写成的，它的实现类是 sun.mic.Launcher$AppClassLoader，另外我们知道 ClassLoader 中有个 getSystemClassLoader 方法,此法返回的正是 AppclassLoader.AppClassLoader 主要负责加载 classpath 所指定的位置的类或者是jr 文档，它也是 Java 程序默认的类加载器。
双亲委派机制的工作流程：

当前 ClassLoader 首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返原来已经加载的类。每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下加载的时候就可以直接返回了。

当前 classLoader 的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到 bootstrp ClassLoader.</p

当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中以便下次有加载请求的时候直接返回。

<h2 id="-minor-gc如果运行的很频繁-可能是什么原因引起的-minor-gc如果运行的很慢-可能是什原因引起的-">★minor gc 如果运行的很频繁，可能是什么原因引起的，minor gc 如果运行的很慢可能是什么原因引起的?</h2>
频繁的原因：

新生代的内存空间分配过小。
程序中 new 太多声明周期短的对象。
threshold 值太高，新生代中的对象迟迟不进入老年代，使得一直占用新生代空间。

很慢的原因：

新生代内存空间太大，扫描时间过长。
对象引用链较长，进行可达性分析时间较长。
新生代 survivor 区设置的比较小，清理后剩余的对象不能装进去需要移动到老年代，造成移动销。
内存分配担保失败，由 minor gc 转化为 full gc。
采用的垃圾收集器效率较低，比如新生代使用 serial 收集器。

<h2 id="-频繁GC问题或内存溢出问题-如何定位-">★ 频繁 GC 问题或内存溢出问题，如何定位？</2>

GC:GC 前后进行 dump, dump 出来后分析到底是哪些大对象造成 full GC。
内存溢出: 在内存溢出的时候进行 Dump

<h2 id="-JVM垃圾收集器有哪些-">★JVM 垃圾收集器有哪些？</h2>
<h5 id="新生代中">新生代中</h5>

Serial (Client，复制，串行)
parNew(serial 的并行版本，可和 CMS 配合)
parallel Scavenge(并行回收，致力于吞吐量，不适合交互频繁的服务器)

<h5 id="老年代中">老年代中</h5>

serial old(跟 serial 差不多，区别是使用标记整理，适合 Client)
parallel old(和 parallel Scavenge 差不多，区别是标记整理,不适合交互频繁的服务器)
cms(低停顿，并发收集，标记两次,适合 Server)

<h5 id="不分新生代和老年代">不分新生代和老年代</h5>
G1(将内存分为一个一个 region,性能良好，低停顿，双清理算法,适合 Server)
<h2 id="-简述GC算法">★ 简述 GC 算法</h2>
GC 算法分成四种：

标记清除算法：首先先标记，然后统一把标记的对象依次清除，缺点是 CPU 消耗大，极易出现存碎片，所以一般用于老年代。

复制算法：把内存区域分成俩块，每次只使用其中一块，然后把还存活的对象放在另一块中，清原先的块，这样的话不会出现内存碎片。新生代常用的。

复制整理：指针碰撞，将使用过的对象移动到内存的一段，不用的放在另一端。

分代收集：根据不同代的区别，使用符合不同代的算法。

简单来说 minorGC 发生在新生代，gc 频繁而且需要开销小，如果采用整理算法的话，频繁整效率低，所以采取复制算法。

老年代：对象相较于新生代 gc 不频繁且对象少，采取标记清除或者标记整理算法。

<h2 id="垃圾回收算法的实现原理-">垃圾回收算法的实现原理。</h2>
1.复制算法
复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第块。但是这种方式，内存的代价太高，每次基本上都要浪费一半的内存。
2.标记清除算法
是 JVM 垃圾回收算法中最古老的一个，该算法共分成两个阶段，第一阶段从引用根节点开始标记所被引用的对象，第二阶段遍历整个堆，清除未被标记的对象。该算法的缺点是需要暂停整个应用，并在回收以后未使用的空间是不连续，即内存碎片，会影响到存储。
3.标记整理算法
此算法结合了标记-清楚算法和复制算法的优点，也分为两个阶段，第一阶段从引用根节点开始标记有被引用的对象，第二阶段遍历整个堆，在回收不存活的对象占用的空间后，会将所有的存活对象往端空闲空间移动，并更新对应的指针。标记-整理算法是在标记-清除算法的基础上，又进行了对象的动，因此成本更高，但是却解决了内存碎片的问题，按顺序排放，同时解决了复制算法所需内存空间大的问题。
4.分代收集
分代收集算法是目前大部分 JVM 的垃圾收集器采用的算法。它的核心思想是根据对象存活的生命周将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Tenured Generation）和新生（Young Generation），在堆区之外还有一个代就是永久代（Permanet Generation）。老年代的点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需被回收，那么就可以根据不同代的特点采取最适合的收集算法。
<h2 id="-GC是什么--为什么要有GC-">★GC 是什么? 为什么要有 GC?</h2>

GC 即垃圾回收，回收的是不再使用的对象的内存空间。
在 java 语言当中封装了内存分配的操作，程序员不需要关心开辟内存空间和释放内存空间。（C + 语言就需要）就可以把更多精力放在与内存无关的编码上。
所以 java 语言需要设计一套方法用于回收程序中不再使用的对象的内存空间，即 GC。

<h2 id="垃圾回收器的基本原理是什么-垃圾回收器可以马上回收内存吗-有什么办法主动通知虚拟机行垃圾回收-">垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通虚拟机进行垃圾回收？</h2>

通过可达性分析，回收确定不可达的对象的内存空间。
不能，调用 System.gc()并不一点执行。
System.gc()可以通知虚拟机进行垃圾回收，但不保证执行。

<h2 id="-运行时异常与受检查异常有何异同-">★ 运行时异常与受检查异常有何异同？</h2>

检查异常是在程序中最经常碰到异常，所有继承自 Exception 并且不是运行时异常的异常都是检异常，比如咱们最常见的 IO 异常和 SQL 异常。这种异常都发生在编译的阶段，Java 编译器强制程去捕获此类型的异常，即它会把可能会出现这些异常的代码放到 try 块中，把对异常的处理代码放到 c tch 块中。受检异常跟程序运行的上下文环境有关，即使程序设计无误，仍然可能因使用的问题而引。

运行时异常不同于检查异常，编译器没有强制对其进行捕获并处理，如果不对异常进行处理，那当出现这种异常的时候，会由 JVM 来处理，比如 NullPointerException 异常，它就是运行时异常。要程序设计得没有问题通常就不会发生运行时异常。

以下是一些关于异常的优良实践（出自 Effective Java）

不要将异常处理用于正常的控制流（设计良好的 API 不应该强迫它的调用者为了正常的控制流而

用异常）
对可以恢复的情况使用受检异常，对编程错误使用运行时异常
避免不必要的使用受检异常（可以通过一些状态检测手段来避免异常的发生）
优先使用标准的异常
每个方法抛出的异常都要有文档
保持异常的原子性
不要在 catch 中忽略掉捕获到的异常

<h2 id="-解释内存中的栈-stack--堆-heap-和静态区-static-area-的用法-">★ 解释内存中的栈(stac
)、堆(heap)和静态区(static area)的用法。</h2>
通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用
存中的栈空间；而通过 new 关键字和构造器创建的对象放在堆空间；程序中的字面量（literal）如直
书写的 100、"hello"和常量都是放在静态区中。栈空间操作起来最快但是栈很小，通常大量的对象都
放在堆空间，理论上整个内存没 有被其他进程使用的空间甚至硬盘上的虚拟内存都可以被当成堆空间
使用。
<h2 id="-JVM是如何分代的-">★JVM 是如何分代的?</h2>
java 堆，分新生代老年代，新生代有 Eden，from surviver，to surviver 三个空间，堆被所有
程共。eden 内存不足时，发生一次 minor GC，会把 from survivor 和 eden 的对象复制到 to survi
or，这次的 to survivor 就变成了下次的 from survivor，经过多次 minor GC，默认 15 次，达到次
的对象会从 survivor 进行老年代。1 次 new 如果新生代装不下，则直接进入老年代。还有动态年龄
定。 年龄从小到大的累加和超过 survivor 一半的话，大于或者等于这个最大年龄的直接进入老年代

堆的年轻代大则老年代小，GC 少，但是每次时间会比较长。年轻代小则老年代大，会缩短每次
C 的时间，但是次数频繁。可以让老年代尽量缓存常用对象，JVM 默认年轻代和老年代的大小比例为
:2,。观察峰值老年代内存，不影响 full GC，加大老年代可调 1:1，但是要给老年代预留三分之一的空
。减少使用全局变量和大对象 ，调整新生代，老年代到最合适。
<h2 id="-jvm-YGC和FGC发生的具体场景-">★jvm YGC 和 FGC 发生的具体场景？</h2>
YGC ： 对新生代堆进行 gc。频率比较高，因为大部分对象的存活寿命较短，在新生代里被回收
性能耗费较小。
FGC ： 全堆范围的 gc。默认堆空间使用到达 80%(可调整)的时候会触发 fgc。以我们生产环境为例
一般比较少会触发 fgc，有时 10 天或一周左右会有一次。
YGC 发生场景：edn 空间不足
FGC 发生场景：old 空间不足，perm 空间不足，调用方法 System.gc()，ygc 时的悲观策略, dump l
ve 的内存信息时(jmap –dump:live)
<h2 id="-Java-8的内存分代改进">★Java 8 的内存分代改进</h2>

方法区是虚拟机规范，而永久代是 HotSpot 的实现，其他虚拟机可能没有永久代这个概念。</li

在 java8 中将永久代从虚拟机堆内存移到了本地内存。称之为元空间。
不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。（字符
常量移至 java 堆中）因此，默认情况下，元空间的大小仅受本地内存限制，但可以通过参数来指定
空间的大小。
为什么要将永久代替换成 Metaspace？可能的原因有：

字符串存在永久代中，容易出现性能问题和内存溢出。
类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久
溢出，太大则容易导致老年代溢出。
永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。
Oracle 可能会将 HotSpot 与 JRockit 合二为一。

<h2 id="-新生代和老生代的内存回收策略">★ 新生代和老生代的内存回收策略</h2>
复制算法(新生代算法)：

复制算法是针对 Java 堆中的新生代内存垃圾回收所使用的回收策略，解决了”标记-清理”的效问题。

复制算法将堆中可用的新生代内存按容量划分成大小相等的两块内存区域，每次只使用其中的一区域。当其中一块内存区域需要进行垃圾回收时，会将此区域内还存活着的对象复制到另一块上面，后再把此内存区域一次性清理掉。这样做的好处是每次都是对整个新生代一半的内存区域进行内存回，内存分配时也就不需要考虑内存碎片等复杂情况，只需要移动堆顶指针，按顺序分配即可。此算法现简单，运行高效。
<pre><code class="highlight-chroma"> 新生代的对象大多数gc完没剩多少，没必要使用整理算法
</code></pre>

标记整理算法(老年代回收算法)：

复制算法在对象存活率较高的老年代会进行很多次的复制操作，效率很低，所以在栈的老年代不用复制算法。

针对老年代对象存活率高的特点，提出了一种称之为”标记-整理算法”。标记过程仍与”标记-除”过程一致，但后续步骤不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然直接清理掉端边界以外的内存。

有一个参数控制对象的年龄多少进入老年代，还有一个参数是控制对象多大直接进入老年代
<pre><code class="highlight-chroma"> 老年代的对象大多回收比较少，如果采用复制算法的话，复制效率低。
</code></pre>

<h2 id="-JVM的编译优化">★JVM 的编译优化</h2>

语言无关的经典优化技术之一：公共子表达式消除。
语言相关的经典优化技术之一：数组范围检查消除。
最重要的优化技术之一：方法内联。
最前沿的优化技术之一：逃逸分析。（栈上分配，标量替换，消除同步）

<h2 id="指令重排序-内存栅栏等">指令重排序，内存栅栏等</h2>

大多数现代微处理器都会采用将指令乱序执行（out-of-order execution，简称 OoOE 或 OOE 的方法，在条件允许的情况下，直接运行当前有能力立即执行的后续指令，避开获取下一条指令所需据时造成的等待。

通过乱序执行的技术，处理器可以大大提高执行效率。除了处理器，常见的 Java 运行时环境的 JI 编译器也会做指令重排序操作，即生成的机器指令与字节码指令顺序不一致。

内存屏障（Memory Barrier，或有时叫做内存栅栏，Memory Fence）是一种 CPU 指令，用于制特定条件下的重排序和内存可见性问题。Java 编译器也会根据内存屏障的规则禁止重排序。java

译器在生成指令序列的适当位置会插入内存屏障指令来禁止特定类型的处理器重排序。

StoreLoad Barriers 是一个“全能型”的屏障，它同时具有其他三个屏障的效果。现代的多处理大都支持该屏障（其他类型的屏障不一定被所有处理器支持）。执行该屏障开销会很昂贵，因为当前理器通常要把写缓冲区中的数据全部刷新到内存中（buffer fully flush）。

<h2 id="JVM常用参数">JVM 常用参数</h2>
https://cloud.tencent.com/eveloper/article/1198524
一、堆设置
<pre><code class="highlight-chroma">-Xms:初始堆大小
-Xmx:最大堆大小
-XX:NewSize=n:置年轻代大小
-XX:NewRatio=n:置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和1/4
-XX:SurvivorRation:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=：2，
一个Survivor区占个年轻代的1/5
-XX:MaxPermSizen:设置持久代大小
</code></pre>
二、收集器设置
<pre><code class="highlight-chroma">-XX:+UseSerialGC:设置串行收集器
-XX:+UseParallelC:设置并行收集器
-XX:+UseParalleldOldGC:设置并行年老代收集器
-XX:+UseConcMakSweepGC:设置并发收集器
</code></pre>
三、垃圾回收统计信息
<pre><code class="highlight-chroma">-XX:+PrintGC
-XX:+PrintGCDetais
-XX:+PrintGCTimStamps
-Xloggc:filename
</code></pre>
四、并行收集器设置
<pre><code class="highlight-chroma">-XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。
-XX:MaxGCPauseillis=n:设置并行收集最大暂停时间
-XX:GCTimeRatio

n:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)
</code></pre>
五、并发收集器设置
<pre><code class="highlight-chroma">-XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。
-XX:ParallelGCThrads=n:设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。
</code></pre>
<h2 id="-你知道哪几种垃圾收集器-各自的优缺点-重点讲下cms和G1-包括原理-流程-优缺点-">★知道哪几种垃圾收集器，各自的优缺点，重点讲下 cms 和 G1，包括原理，流程，优缺点。</h2>
<h5 id="CMS-以获取最短回收停顿时间为目标的收集器-基于并发-标记清理-实现">CMS：以获取短回收停顿时间为目标的收集器，基于并发“标记清理”实现</h5>
<h5 id="过程-">过程：</h5>

初始标记：独占 PUC，仅标记 GCroots 能直接关联的对象

并发标记：可以和用户线程并行执行，标记所有可达对象

重新标记：独占 CPU(STW)，对并发标记阶段用户线程运行产生的垃圾对象进行标记修正

并发清理：可以和用户线程并行执行，清理垃圾

<h5 id="优点-">优点:</h5>
并发，低停顿
<h5 id="缺点-">缺点：</h5>

对 CPU 非常敏感：在并发阶段虽然不会导致用户线程停顿，但是会因为占用了一部分线程使应程序变慢

无法处理浮动垃圾：在最后一步并发清理过程中，用户线程执行也会产生垃圾，但是这部分垃圾在标记之后，所以只有等到下一次 gc 的时候清理掉，这部分垃圾叫浮动垃圾

3, CMS 使用“标记-清理”法会产生大量的空间碎片，当碎片过多，将会给大对象空间的分配带很大的麻烦，往往会出现老年代还有很大的空间但无法找到足够大的连续空间来分配当前对象，不得提前触发一次 FullGC，为了解决这个问题 CMS 提供了一个开关参数，用于在 CMS 顶不住，要进行ullGC 时开启内存碎片的合并整理过程，但是内存整理的过程是无法并发的，空间碎片没有了但是停时间变长了
<h5 id="CMS-出现FullGC的原因-">CMS 出现 FullGC 的原因：</h5>

年轻带晋升到老年带没有足够的连续空间，很有可能是内存碎片导致的

在并发过程中 JVM 觉得在并发过程结束之前堆就会满，需要提前触发 FullGC

<h5 id="G1-是一款面向服务端应用的垃圾收集器">G1：是一款面向服务端应用的垃圾收集器</h5>
<h5 id="特点-">特点：</h5>

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）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。

分代收集：分代概念在 G1 中依然得以保留。虽然 G1 可以不需要其它收集器配合就能独立管理个 GC 堆，但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 的旧象以获取更好的收集效果。也就是说 G1 可以自己管理新生代和老年代了。

空间整合：由于 G1 使用了独立区域（Region）概念，G1 从整体来看是基于“标记-整理”算实现收集，从局部（两个 Region）上来看是基于“复制”算法实现的，但无论如何，这两种算法都味着 G1 运作期间不会产生内存空间碎片。

可预测的停顿：这是 G1 相对于 CMS 的另一大优势，降低停顿时间是 G1 和 CMS 共同的关注，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用这明确指定一个长度为 M 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。

与其它收集器相比，G1 变化较大的是它将整个 Java 堆划分为多个大小相等的独立区域（Regio），虽然还保留了新生代和来年代的概念，但新生代和老年代不再是物理隔离的了它们都是一部分 Reion（不需要连续）的集合。同时，为了避免全堆扫描，G1 使用了 Remembered Set 来管理相关的象引用信息。当进行内存回收时，在 GC 根节点的枚举范围中加入 Remembered Set 即可保证不对堆扫描也不会有遗漏了。

如果不计算维护 Remembered Set 的操作，G1 收集器的运作大致可划分为以下几个步骤：</p

初始标记（Initial Making）

并发标记（Concurrent Marking）

最终标记（Final Marking）

筛选回收（Live Data Counting and Evacuation）

看上去跟 CMS 收集器的运作过程有几分相似，不过确实也这样。初始阶段仅仅只是标记一下 GC Roots 能直接关联到的对象，并且修改 TAMS（Next Top Mark Start）的值，让下一阶段用户程序发运行时，能在正确可以用的 Region 中创建新对象，这个阶段需要停顿线程，但耗时很短。并发标阶段是从 GC Roots 开始对堆中对象进行可达性分析，找出存活对象，这一阶段耗时较长但能与用户程并发运行。而最终标记阶段需要吧 Remembered Set Logs 的数据合并到 Remembered Set 中，阶段需要停顿线程，但可并行执行。最后筛选回收阶段首先对各个 Region 的回收价值和成本进行排，根据用户所期望的 GC 停顿时间来制定回收计划，这一过程同样是需要停顿线程的，但 Sun 公司

露这个阶段其实也可以做到并发，但考虑到停顿线程将大幅度提高收集效率，所以选择停顿。
<h2 id="当出现了内存溢出-你怎么排错-">当出现了内存溢出，你怎么排错。</h2>
使用一个参数使得内存溢出的时候生成堆快照，然后用 jvisualvm 等内存分析工具进行分析
<h2 id="-简单说说你了解的类加载器-可以打破双亲委派么-怎么打破--tomcat如何实现多版本及热署-">★ 简单说说你了解的类加载器，可以打破双亲委派么，怎么打破。 tomcat 如何实现多版本及部署?</h2>

双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，他首先不会自己去尝试加这个类，而是把这个请求委派父类加载器去完成。每一个层次的类加载器都是如此，因此所有的加载求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个请求（他的搜索围中没有找到所需的类）时，子加载器才会尝试自己去加载。
为什么要这么做呢？
如果没有使用双亲委派模型，由各个类加载器自行加载的话，如果用户自己编写了一个称为 java.lng.Object 的类，并放在程序的 ClassPath 中，那系统将会出现多个不同的 Object 类， Java 类型系中最基础的行为就无法保证。应用程序也将会变得一片混乱。

线程上下文类加载器可打破。tomcat 为了实现类的多版本以及热部署，使用自定义的类加载器打破双亲委托机制。
<h2 id="怎么打出线程栈信息-">怎么打出线程栈信息。</h2>
jstack 命令
<h2 id="请解释如下jvm参数的含义-">请解释如下 jvm 参数的含义：</h2>
<pre><code class="highlight-chroma">-server 服务器启动模式
-Xms512m 最小大小
-Xmx512m 最大大小
-Xss1024K 栈空大小
-XX:PermSize=25m 永久代大小
-XX:MaxPermSize 512m 最大永久代大小
-XX:MaxTenuring hreshold=20 晋升年龄
-XX:CMSInitiating ccupancyFraction=80 当内存达到的阈值进行gc
-XX:+UseCMSInitiitingOccupancyOnly。是否一直使用最开始的阈值，后期不动态调整。
</code></pre>
转自我的 github
<h5 id="技术讨论群QQ-1398880">技术讨论群 QQ:1398880</h5>