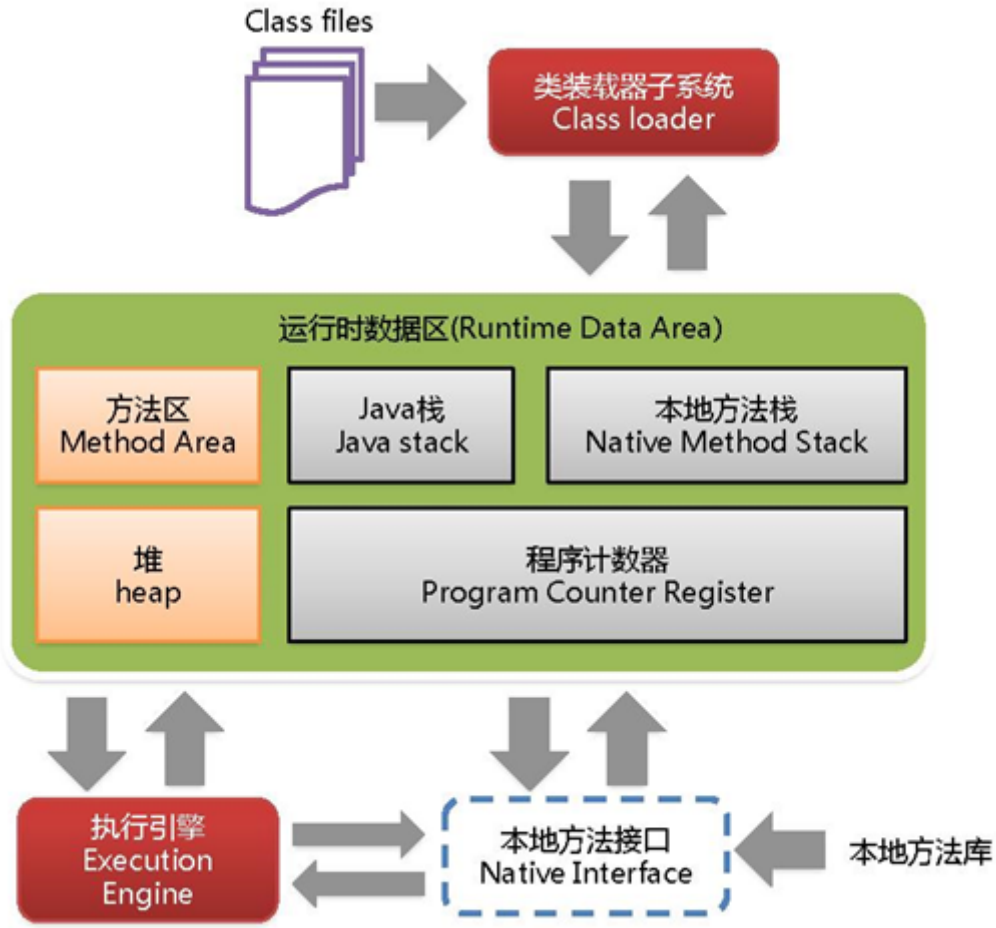
JVM 常考面试题

JVM知识点大纲

- 类的加载机制
- JVM内存结构
- GC算法，垃圾回收
- GC分析，命令调优

1. 什么是JVM内存结构?



jvm将虚拟机分为5大区域，程序计数器、虚拟机栈、本地方法栈、java堆、方法区；

- 程序计数器：线程私有的，是一块很小的内存空间，作为当前线程的行号指示器，用于记录当前虚拟机正在执行的线程指令地址；
- 虚拟机栈：线程私有的，每个方法执行的时候都会创建一个栈帧，用于**存储局部变量表**、**操作数**、**动态链接**和**方法返回**等信息，当线程请求的栈深度超过了虚拟机允许的最大深度时，就会抛出StackOverFlowError；
- 本地方法栈：线程私有的，保存的是native方法的信息，当一个jvm创建的线程调用native方法后，jvm不会在虚拟机栈中为该线程创建栈帧，**而是简单的动态链接并直接调用该方法**；
- 堆：java堆是所有线程共享的一块内存，几乎所有对象的实例和数组都要在堆上分配内存，因此该区域经常发生垃圾回收的操作；
- 方法区：存放**已被加载的类信息**、**常量**、**静态变量**、**即时编译器编译后的代码数据。**即永久代，在jdk1.8中不存在方法区了，被元数据区替代了，原方法区被分成两部分；1：加载的类信息，2：运

行时常量池；加载的类信息被保存在元数据区中，运行时常量池保存在堆中；

出来程序计数器，其余部分都会产生OOM

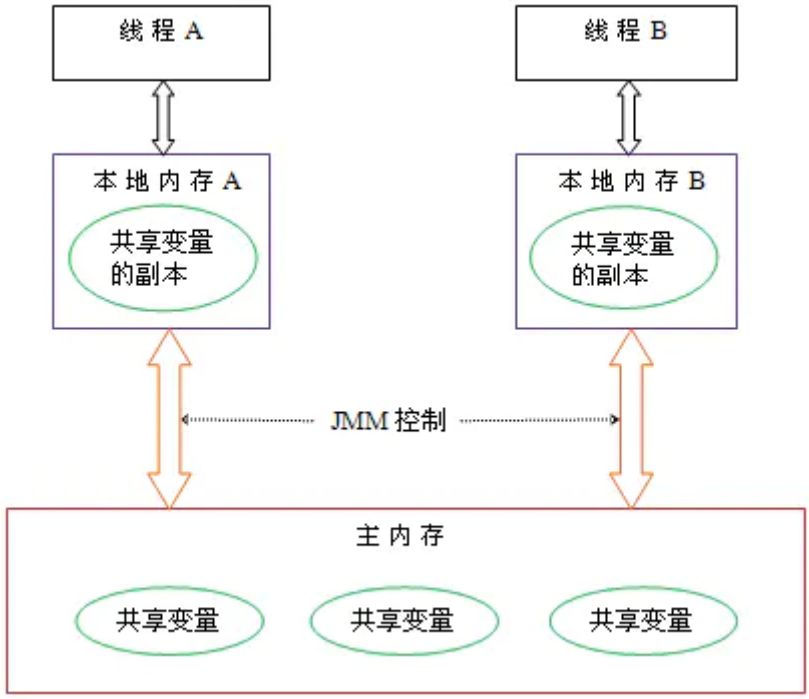
2. 什么是JVM内存模型?

Java 内存模型（下文简称 **JMM**）就是在底层处理器内存模型的基础上，定义自己的多线程语义。它明确指定了一组排序规则，来保证线程间的可见性。

这一组规则被称为 **Happens-Before**, JMM 规定，要想保证 B 操作能够看到 A 操作的结果（无论它们是否在同一个线程），那么 A 和 B 之间必须满足 **Happens-Before 关系**：

- **单线程规则**：一个线程中的每个动作都 happens-before 该线程中后续的每个动作
- **监视器锁定规则**：监听器的**解锁**动作 happens-before 后续对这个监听器的**锁定**动作
- **volatile 变量规则**：对 volatile 字段的写入动作 happens-before 后续对这个字段的每个读取动作
- **线程 start 规则**：线程 **start()** 方法的执行 happens-before 一个启动线程内的任意动作
- **线程 join 规则**：一个线程内的所有动作 happens-before 任意其他线程在该线程 **join()** 成功返回之前
- **传递性**：如果 A happens-before B, 且 B happens-before C, 那么 A happens-before C

怎么理解 happens-before 呢？如果按字面意思，比如第二个规则，线程（不管是不是同一个）的解锁动作发生在锁定之前？这明显不对。happens-before 也是为了保证可见性，比如那个解锁和加锁的动作，可以这样理解，线程1释放锁退出同步块，线程2加锁进入同步块，那么线程2就能看见线程1对共享对象修改的结果。



Java 提供了几种语言结构，包括 *volatile*, *final* 和 *synchronized*, 它们旨在帮助程序员向**编译器**描述程序的并发要求，其中：

- **volatile** - 保证**可见性**和**有序性**
- **synchronized** - 保证**可见性**和**有序性**; 通过**管程（Monitor）*保证一组动作的*原子性**
- **final** - 通过禁止**在构造函数初始化**和**给 final 字段赋值**这两个动作的重排序，保证**可见性**（如果 **this 引用逃逸**就不好说可见性了）

编译器在遇到这些关键字时，会插入相应的内存屏障，保证语义的正确性。

有一点需要**注意**的是，**synchronized 不保证**同步块内的代码禁止重排序，因为它通过锁保证同一时刻只有**一个线程**访问同步块（或临界区），也就是说同步块的代码只需满足 **as-if-serial** 语义 - 只要单线程的执行结果不改变，可以进行重排序。

所以说，Java 内存模型描述的是多线程对共享内存修改后彼此之间的可见性，另外，还确保正确同步的 Java 代码可以在不同体系结构的处理器上正确运行。

3. heap 和stack 有什么区别?

（**1**）申请方式

stack:由系统自动分配。例如，声明在函数中一个局部变量 int b; 系统自动在栈中为 b 开辟空间

heap:需要程序员自己申请，并指明大小，在 c 中 malloc 函数，对于Java 需要手动 new Object()的形式开辟

（**2**）申请后系统的响应

stack：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

heap：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

（**3**）申请大小的限制

stack：**栈是向低地址扩展的数据结构，是一块连续的内存的区域。**这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS 下，栈的大小是 2M（默认值也取决于虚拟内存的大小），如果申请的空间超过栈的剩余空间时，将提示 overflow。因此，能从**栈获得的空间较小**(受限于系统规定的栈的容量)。

heap：**堆是向高地址扩展的数据结构，是不连续的内存区域。**这是由于系统是用链表来存储的空闲内存地址的， 自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见， **堆获得的空间比较灵活，也比较**（受限于系统有效虚拟内存）。

（**4**）申请效率的比较

stack：由系统自动分配，速度较快。但程序员是无法控制的。

heap：由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。

（**5**）heap和stack中的存储内容

stack：在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址， 然后是函数的各个参数，在大多数的 C 编译器中，参数是由右往左入栈的，然后是函数中的局部变量。**注意静态变量是不入栈的。**

当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

heap：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

4. 什么情况下会发生栈内存溢出?

1、栈是线程私有的，栈的生命周期和线程一样，每个方法在执行的时候就会创建一个栈帧，它包含局部变量表、操作数栈、动态链接、方法出口等信息，局部变量表又包括基本数据类型和对象的引用;
2、当线程请求的栈深度超过了虚拟机允许的最大深度时，会抛出StackOverFlowError异常，方法递归调用肯可能会出现该问题;
3、调整参数-xss去调整jvm栈的大小

5. 谈谈对 OOM 的认识？如何排查 OOM 的问题?

除了程序计数器，其他内存区域都有 OOM 的风险。

- 栈一般经常会发生 StackOverflowError，比如 32 位的 windows 系统单进程限制 2G 内存，无限创建线程就会发生栈的 OOM
- Java 8 常量池移到堆中，溢出会出 java.lang.OutOfMemoryError: Java heap space，设置最大元空间大小参数无效；
- 堆内存溢出，报错同上，这种比较好理解，GC 之后无法在堆中申请内存创建对象就会报错；
- 方法区 OOM，经常会遇到的是动态生成大量的类、jsp 等；
- 直接内存 OOM，涉及到 -XX:MaxDirectMemorySize 参数和 Unsafe 对象对内存的申请。

排查 OOM 的方法：

- 增加两个参数 -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/heapdump.hprof，当 OOM 发生时自动 dump 堆内存信息到指定目录；
- 同时 jstat 查看监控 JVM 的内存和 GC 情况，先观察问题大概出在什么区域；
- 使用 MAT 工具载入到 dump 文件，分析大对象的占用情况，比如 HashMap 做缓存未清理，时间长了就会内存溢出，可以把改为弱引用 。

6. 谈谈 JVM 中的常量池?

JVM常量池主要分为**Class文件常量池、运行时常量池，全局字符串常量池，以及基本类型包装类对象常量池。**

- **Class文件常量池。**class文件是一组以字节为单位的二进制数据流，在java代码的编译期间，我们编写的java文件就被编译为.class文件格式的二进制数据存放在磁盘中，其中就包括class文件常量池。
- **运行时常量池**：运行时常量池相对于class常量池一大特征就是具有动态性，java规范并不要求常量只能在运行时才产生，也就是说运行时常量池的内容并不全部来自class常量池，在运行时可以通过代码生成常量并将其放入运行时常量池中，这种特性被用的最多的就是String.intern()。
- **全局字符串常量池**：字符串常量池是JVM所维护的一个字符串实例的引用表，在HotSpot VM中，它是一个叫做StringTable的全局表。在字符串常量池中维护的是字符串实例的引用，底层C++实现就是一个Hashtable。这些被维护的引用所指的字符串实例，被称作”被驻留的字符串”或”interned string”或通常所说的”进入了字符串常量池的字符串”。
- 基本类型包装类对象常量池：java中基本类型的包装类的大部分都实现了常量池技术，这些类是Byte,Short,Integer,Long,Character,Boolean,另外两种浮点数类型的包装类则没有实现。另外上面这5种整型的包装类也只是在对应值小于等于127时才可使用对象池，也即对象不负责创建和管理大于127的这些类的对象。

7. 如何判断一个对象是否存活?

判断一个对象是否存活，分为两种算法1：引用计数法；2：可达性分析算法；

引用计数法:
给每一个对象设置一个引用计数器，当有一个地方引用该对象的时候，引用计数器就+1，引用失效时，引用计数器就-1；当引用计数器为0的时候，就说明这个对象没有被引用，也就是垃圾对象，等待回收；
缺点：无法解决循环引用的问题，当A引用B，B也引用A的时候，此时AB对象的引用都不为0，此时也就无法垃圾回收，所以一般主流虚拟机都不采用这个方法；

可达性分析法
从一个被称为GC Roots的对象向下搜索，如果一个对象到GC Roots没有任何引用链相连接时，说明此对象不可用，在java中可以作为GC Roots的对象有以下几种：

- 虚拟机栈中引用的对象

- 方法区类静态属性引用的变量
- 方法区常量池引用的对象
- 本地方法栈JNI引用的对象

但一个对象满足上述条件的时候，不会马上被回收，还需要进行两次标记；第一次标记：判断当前对象是否有finalize()方法并且该方法没有被执行过，若不存在则标记为垃圾对象，等待回收；若有的话，则进行第二次标记；第二次标记将当前对象放入F-Queue队列，并生成一个finalize线程去执行该方法，虚拟机不保证该方法一定会被执行，这是因为如果线程执行缓慢或进入了死锁，会导致回收系统的崩溃；如果执行了finalize方法之后仍然没有与GC Roots有直接或者间接的引用，则该对象会被回收；

8. 强引用、软引用、弱引用、虚引用是什么，有什么区别?

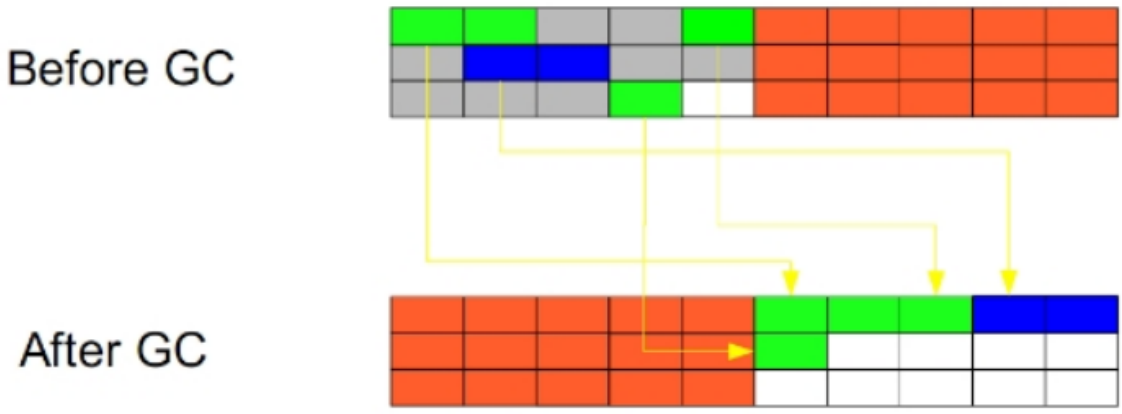
- 强引用，就是普通的对象引用关系，如 String s = new String("ConstXiong")
- 软引用，用于维护一些可有可无的对象。只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。SoftReference 实现
- 弱引用，相比软引用来说，要更加无用一些，它拥有更短的生命周期，当 JVM 进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。WeakReference 实现
- 虚引用是一种形同虚设的引用，在现实场景中用的不是很多，它主要用来跟踪对象被垃圾回收的活动。PhantomReference 实现

9. 被引用的对象就一定能存活吗?

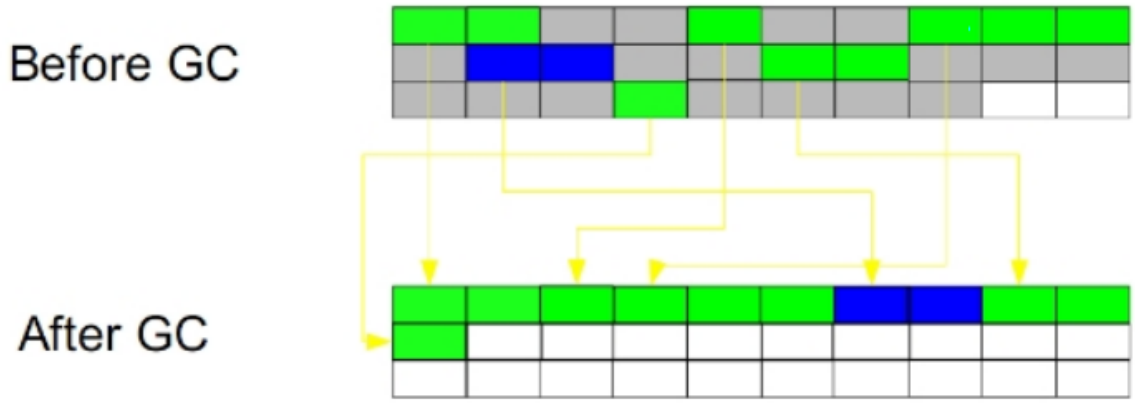
> 不一定，看 Reference 类型，弱引用在 GC 时会被回收，软引用在内存不足的时候，即 OOM 前会被回收，但如果没有在 Reference Chain 中的对象就一定会被回收。

10. Java中的垃圾回收算法有哪些?

java中有四种垃圾回收算法，分别是标记清除法、标记整理法、复制算法、分代收集算法；
标记清除法:
第一步：利用可达性去遍历内存，把存活对象和垃圾对象进行标记；
第二步：在遍历一遍，将所有标记的对象回收掉；
特点：效率不行，标记和清除的效率都不高；标记和清除后会产生大量的不连续的空间分片，可能会导致之后程序运行的时候需分配大对象而找不到连续分片而不得不触发一次GC；



标记整理法:
第一步：利用可达性去遍历内存，把存活对象和垃圾对象进行标记；
第二步：将所有的存活的对象向一段移动，将端边界以外的对象都回收掉；
特点：适用于存活对象多，垃圾少的情况；需要整理的过程，无空间碎片产生；



复制算法:
将内存按照容量大小分为大小相等的两块，每次只使用一块，当一块使用完了，就将还存活的对象移到另一块上，然后在把使用过的内存空间移除；
特点：不会产生空间碎片；内存使用率极低；

分代收集算法:
根据内存对象的存活周期不同，将内存划分成几块，java虚拟机一般将内存分成新生代和老生代，在新生代中，有大量对象死去和少量对象存活，所以采用复制算法，只需要付出少量存活对象的复制成本就可以完成收集；老年代中因为对象的存活率极高，没有额外的空间对他进行分配担保，所以采用标记清理或者标记整理算法进行回收；

对比

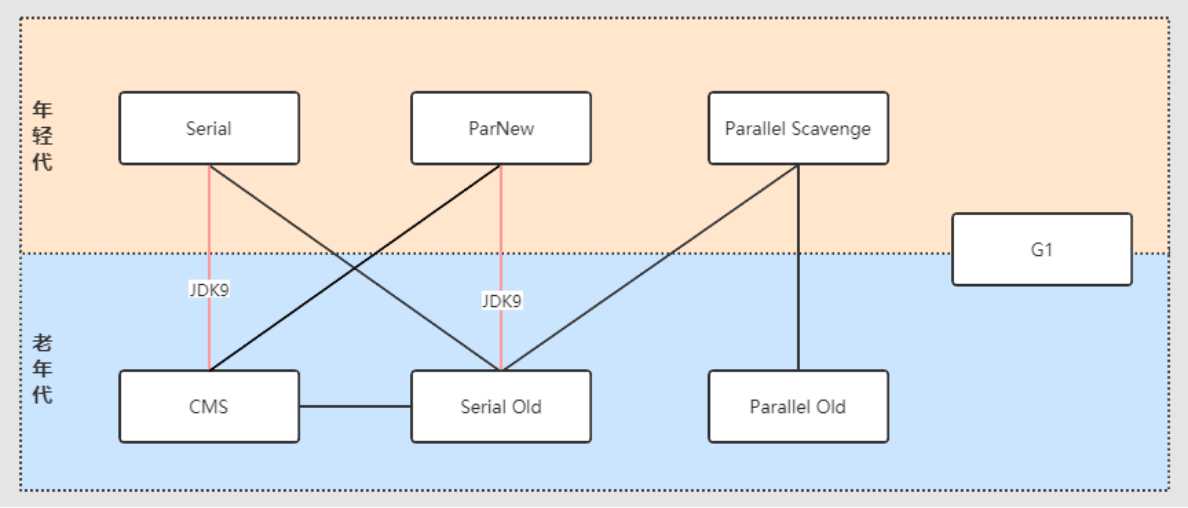
回收算法类型	优点	缺点	适用区域
标记-清除算法（Mark-Sweep）	不需要移动对象，简单高效	标记、清除过程效率低，产生内存碎片	老年代
标记-复制算法（Copying）	清理速度快，没有内存碎片产生	内存使用率低，有可能产生频繁复制的问题	年轻代
标记-整理算法（Mark-Compact）	简单高效、没有内存碎片产生 综了合前两种算法的优点	仍然需要移动局部对象	老年代

11. 有哪几种垃圾回收器，各自的优缺点是什么?

垃圾回收器主要分为以下几种：Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old、CMS、G1；

- Serial:单线程的收集器，收集垃圾时，必须stop the world，使用复制算法。它的最大特点是在进行垃圾回收时，需要对所有正在执行的线程暂停（stop the world），对于有些应用是难以接受的，但是如果应用的实时性要求不是那么高，只要停顿的时间控制在N毫秒之内，大多数应用还是可以接受的，是client级别的默认GC方式。
- ParNew:Serial收集器的多线程版本，也需要stop the world，复制算
- Parallel Scavenge:新生代收集器，复制算法的收集器，并发的多线程收集器，目标是达到一个可控的吞吐量，和ParNew的最大区别是GC自动调节策略；虚拟机会根据系统的运行状态收集性能监控信息，动态设置这些参数，以提供最优停顿时间和最高的吞吐量；
- Serial Old:Serial收集器的老年代版本，单线程收集器，使用标记整理算法。
- Parallel Old：是Parallel Scavenge收集器的老年代版本，使用多线程，标记-整理算法。
- CMS:是一种以获得最短回收停顿时间为目标的收集器，标记清除算法，运作过程：初始标记，并发标记，重新标记，并发清除，收集结束会产生大量空间碎片；
- G1:标记整理算法实现，运作流程主要包括以下：初始标记，并发标记，最终标记，筛选回收。不会产生空间碎片，可以精确地控制停顿；G1将整个堆分为大小相等的多个Region（区域），G1跟踪每个区域的垃圾大小，在后台维护一个优先级列表，每次根据允许的收集时间，优先回收价值最大的区域，已达到在有限时间内获取尽可能高的回收效率；

垃圾回收器间的配合使用图：



各个垃圾回收器对比：

垃圾回收器类型	使用的回收算法	特点	设置参数	适用区域
Serial	复制算法	单线程回收器，简单高效，停顿时间长。服务端程序几乎不用，一般用于客户端程序。	-XX:+UseSerialGC（年轻代：Serial，老年代：Serial Old）	年轻代
ParNew	复制算法	多线程回收器，降低了停顿时间，但增加了线程上下文切换的消耗。	-XX:+UseParNewGC（年轻代：ParNew，老年代：Serial Old，JDK9生效） -XX:ParallelGCThreads=4（设置并发线程数，默认等于CPU核心数）	年轻代
Parallel Scavenge	复制算法	多线程回收器，追求高吞吐量，高效利用CPU。可以控制最大垃圾回收停顿时间。	-XX:+UseParallelGC（年轻代：Parallel Scavenge，老年代：Parallel Old） -XX:ParallelGCThreads=4（设置并发线程数，默认等于CPU核心数） -XX:MaxGCPauseMillis=200（设置最大GC停顿时间） -XX:GCTimeRatio=99（设置吞吐量） -XX: +UseAdaptiveSizePolicy（启用自适应调解策略，默认开启）	年轻代
Serial Old	标记-整理算法	单线程回收器，简单高效，停顿时间长，主要用于客户端程序。或者与其它回收器配合使用。		老年代
Parallel Old	标记-整理算法	多线程回收器，追求高吞吐量。	-XX:+UseParallelOldGC（年轻代：Parallel Scavenge，老年代：Parallel Old）	老年代
CMS	标记-清除算法	并发回收器，可以与用户线程同时进行。高并发，低停顿，追求最短回收停顿时间，CPU占用比较高，响应时间快，停顿时间短。需要 Serial Old 来避免并发失败的风险。	-XX:+UseConcMarkSweepGC（年轻代：ParNew，老年代：CMS，Serial Old 备用） -XX:CMSInitiatingOccupancyFraction=92（老年代使用空间超过这个比例后触发CMS回收） -XX:+UseCMSInitiatingOccupancyOnly（让JVM使用设定的阀值） -XX:CMSWaitDuration=2000（CMS GC间隔时间，默认2000毫秒） -XX:+UseCMSCompactAtFullCollection（在必须 Full GC 时整理内存碎片，默认开启） -XX:CMSFullGCsBeforeCompaction=0（多少次 Full GC 后整理内存碎片） -XX:CMSBootstrapOccupancy=50（默认50%）	老年代
G1	标记-整理算法 + 复制算法	并发回收器，可以与用户线程同时进行。基于Region的内存布局形式，高并发、低停顿，可控的回收停顿时间。	-XX:+UseG1GC（年轻代：G1，老年代：G1） -XX:MaxGCPauseMillis=200（设置最大GC停顿时间，默认200毫秒） -XX:G1HeapRegionSize=1（设置每个 Region 的大小，1M~32M，且为2的N次冥） -XX:InitiatingHeapOccupancyPercent（老年代超过这个比例后触发MixedGC，默认45%） -XX:G1HeapWastePercent（空闲Region比例，超过这个阀值后停止MixedGC，默认5%） -XX:G1MixedGCLiveThresholdPercent（Region存活对象小于这个比例才会被回收，默认85%） -XX:G1NewSizePercent（新生代占用初始的堆内存比例，默认5%） -XX:G1MaxNewSizePercent（新生代占用最大的堆内存比例，默认60%） -XX:ParallelGCThreads=4（设置并发线程数，默认等于CPU核心数）	年轻代、老年代

12. 详细说一下CMS的回收过程？CMS的问题是什么？

CMS(Concurrent Mark Sweep，并发标记清除) 收集器是以获取最短回收停顿时间为目标的收集器（追求低停顿），它在垃圾收集时使得用户线程和 GC 线程并发执行，因此在垃圾收集过程中用户也不会感到明显的卡顿。

从名字就可以知道，CMS是基于“标记-清除”算法实现的。CMS 回收过程分为以下四步：

1. 初始标记 （CMS initial mark）：主要是标记 GC Root 开始的下级（注：仅下一级）对象，这个过程会 STW，但是跟 GC Root 直接关联的下级对象不会很多，因此这个过程其实很快。
2. 并发标记 (CMS concurrent mark)：根据上一步的结果，继续向下标识所有关联的对象，直到这条链上的最尽头。这个过程是多线程的，虽然耗时理论上会比较长，但是其它工作线程并不会阻塞，没有 STW。
3. 重新标记（CMS remark）：顾名思义，就是要再标记一次。为啥还要再标记一次？因为第 2 步并没有阻塞其它工作线程，其它线程在标识过程中，很有可能会产生新的垃圾。
4. 并发清除（CMS concurrent sweep）：清除阶段是清理删除掉标记阶段判断的已经死亡的对象，由于不需要移动存活对象，所以这个阶段也是可以与用户线程同时并发进行的。

CMS 的问题：

1. 并发回收导致CPU资源紧张：

在并发阶段，它虽然不会导致用户线程停顿，但却会因为占用了一部分线程而导致应用程序变慢，降低程序总吞吐量。CMS默认启动的回收线程数是：（CPU核数 + 3）/ 4，当CPU核数不足四个时，CMS对用户程序的影响就可能变得很大。

2. 无法清理浮动垃圾：

在CMS的并发标记和并发清理阶段，用户线程还在继续运行，就还会伴随有新的垃圾对象不断产生，但这一部分垃圾对象是出现在标记过程结束以后，CMS无法在当次收集中处理掉它们，只好留到下一次垃圾收集时再清理掉。这一部分垃圾称为"浮动垃圾"。

3. 并发失败（Concurrent Mode Failure）：

由于在垃圾回收阶段用户线程还在并发运行，那就还需要预留足够的内存空间提供给用户线程使用，因此CMS不能像其他回收器那样等到老年代几乎完全被填满了再进行回收，必须预留一部分空间供并发回收时的程序运行使用。默认情况下，当老年代使用了 92% 的空间后就会触发 CMS 垃圾回收，这个值可以通过 -XX**:** CMSInitiatingOccupancyFraction 参数来设置。

这里会有一个风险：要是CMS运行期间预留的内存无法满足程序分配新对象的需要，就会出现一次"并发失败"（Concurrent Mode Failure），这时候虚拟机将不得不启动后备预案：Stop The World，临时启用 Serial Old 来重新进行老年代的垃圾回收，这样一来停顿时间就很长了。

4.内存碎片问题：

CMS是一款基于"标记-清除"算法实现的回收器，这意味着回收结束时会有内存碎片产生。内存碎片过多时，将会给大对象分配带来麻烦，往往会出现老年代还有很多剩余空间，但就是无法找到足够大的连续空间来分配当前对象，而不得不提前触发一次 Full GC 的情况。

为了解决这个问题，CMS收集器提供了一个 -XX**:**+UseCMSCompactAtFullCollection 开关参数（默认开启），用于在 Full GC 时开启内存碎片的合并整理过程，由于这个内存整理必须移动存活对象，是无法并发的，这样停顿时间就会变长。还有另外一个参数 -XX**:**CMSFullGCsBeforeCompaction，这个参数的作用是要求CMS在执行过若干次不整理空间的 Full GC 之后，下一次进入 Full GC 前会先进行碎片整理（默认值为0，表示每次进入 Full GC 时都进行碎片整理）。

13. 详细说一下G1的回收过程?

G1（Garbage First）回收器采用面向局部收集的设计思路和基于Region的内存布局形式，是一款主要面向服务端应用的垃圾回收器。G1设计初衷就是替换 CMS，成为一种全功能收集器。G1 在JDK9 之后成为服务端模式下的默认垃圾回收器，取代了 Parallel Scavenge 加 Parallel Old 的默认组合，而 CMS 被声明为不推荐使用的垃圾回收器。G1从整体来看是基于 标记-整理 算法实现的回收器，但从局部（两个Region之间）上看又是基于 标记-复制 算法实现的。

G1 回收过程，G1 回收器的运作过程大致可分为四个步骤：

1. 初始标记（会STW）：仅仅只是标记一下 GC Roots 能直接关联到的对象，并且修改TAMS指针的值，让下一阶段用户线程并发运行时，能正确地在可用的Region中分配新对象。这个阶段需要停顿线程，但耗时很短，而且是借用进行Minor GC的时候同步完成的，所以G1收集器在这个阶段实际并没有额外的停顿。
2. 并发标记：从 GC Roots 开始对堆中对象进行可达性分析，递归扫描整个堆里的对象图，找出要回收的对象，这阶段耗时较长，但可与用户程序并发执行。当对象图扫描完成以后，还要重新处理在并发时有引用变动的对象。
3. 最终标记（会STW）：对用户线程做短暂的暂停，处理并发阶段结束后仍有引用变动的对象。
4. 清理阶段（会STW）：更新Region的统计数据，对各个Region的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收计划，可以自由选择任意多个Region构成回收集，然后把决定回收的那一部分Region的存活对象复制到空的Region中，再清理掉整个旧Region的全部空间。这里的操作涉及存活对象的移动，必须暂停用户线程，由多条回收器线程并行完成的。

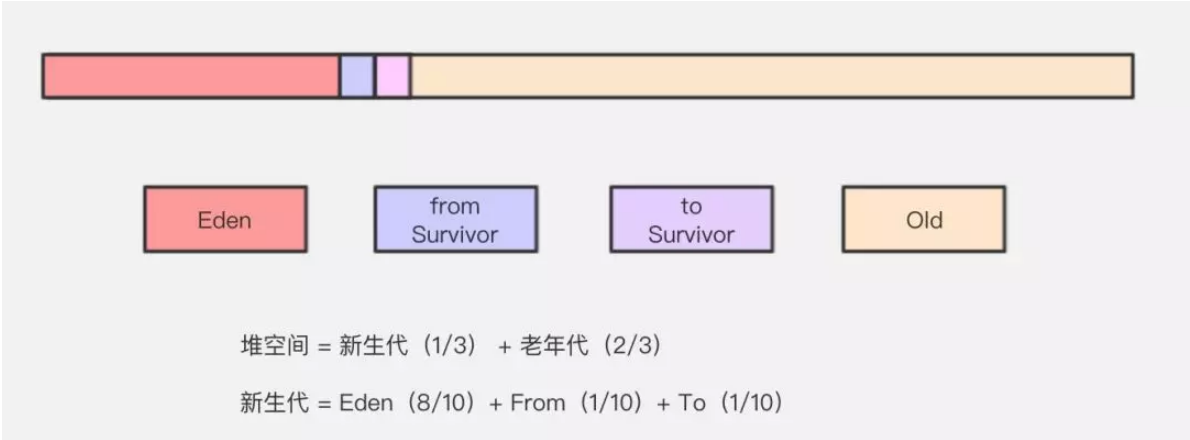
14. JVM中一次完整的GC是什么样子的?

先描述一下Java堆内存划分。

在 Java 中，堆被划分成两个不同的区域：新生代 (Young)、老年代 (Old)，新生代默认占总空间的 1/3，老年代默认占 2/3。
新生代有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1。

新生代的垃圾回收（又称Minor GC）后只有少量对象存活，所以选用复制算法，只需要少量的复制成本就可以完成回收。

老年代的垃圾回收（又称Major GC）通常使用“标记-清理”或“标记-整理”算法。



再描述它们之间转化流程：

- 对象优先在Eden分配。当 eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。
    - 在 Eden 区执行了第一次 GC 之后，存活的对象会被移动到其中一个 Survivor 分区；
    - Eden 区再次 GC，这时会采用复制算法，将 Eden 和 from 区一起清理，存活的对象会被复制到 to 区；
    - 移动一次，对象年龄加 1，对象年龄大于一定阈值会直接移动到老年代。GC年龄的阈值可以通过参数 -XX:MaxTenuringThreshold 设置，默认为 15；
    - 动态对象年龄判定：Survivor 区相同年龄所有对象大小的总和 > (Survivor 区内存大小 * 这个目标使用率)时，大于或等于该年龄的对象直接进入老年代。其中这个使用率通过 -XX:TargetSurvivorRatio 指定，默认为 50%；
    - Survivor 区内存不足会发生担保分配，超过指定大小的对象可以直接进入老年代。
- 大对象直接进入老年代，大对象就是需要大量连续内存空间的对象（比如：字符串、数组），为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。
- 老年代满了而**无法容纳更多的对象**，Minor GC 之后通常就会进行Full GC，Full GC 清理整个内存堆 – **包括年轻代和老年代。**

15. Minor GC 和 Full GC 有什么不同呢?

Minor GC：只收集新生代的GC。

Full GC: 收集整个堆，包括 新生代，老年代，永久代(在 JDK 1.8及以后，永久代被移除，换为 metaspace 元空间)等所有部分的模式。

Minor GC触发条件：当Eden区满时，触发Minor GC。

Full GC触发条件:

- 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。如果发现统计数据说之前Minor GC的平均晋升大小比目前old gen剩余的空间大，则不会触发Minor GC而是转为触发full GC。
- 老年代空间不够分配新的内存（或永久代空间不足，但只是JDK1.7有的，这也是用元空间来取代永久代的原因，可以减少Full GC的频率，减少GC负担，提升其效率）。

- 由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把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。
- 调用System.gc时，系统建议执行Full GC，但是不必然执行。

16. 介绍下空间分配担保原则?

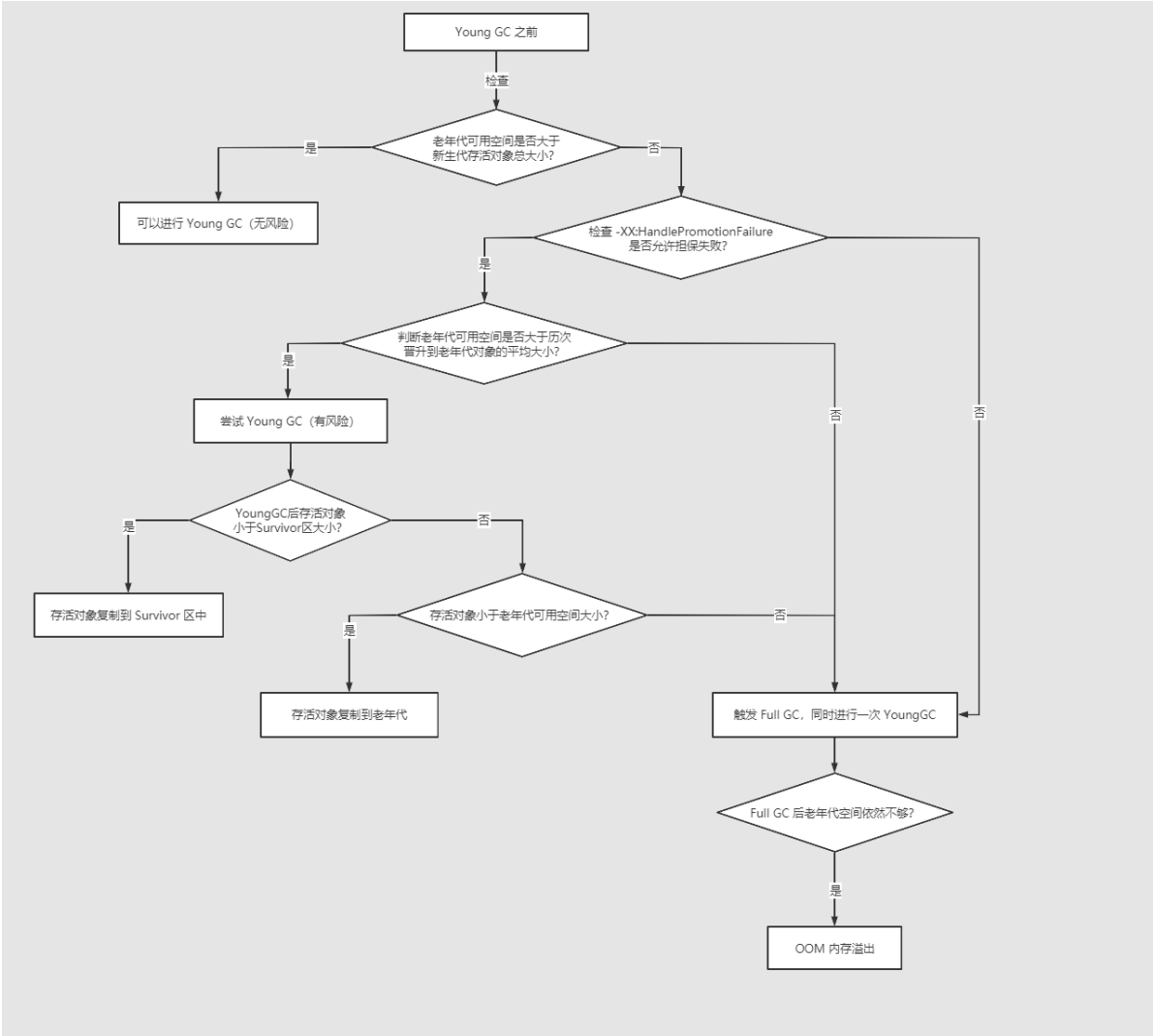
如果YougGC时新生代有大量对象存活下来，而 survivor 区放不下了，这时必须转移到老年代中，但这时发现老年代也放不下这些对象了，那怎么处理呢？其实JVM有一个老年代空间分配担保机制来保证对象能够进入老年代。

在执行每次 YoungGC 之前，JVM会先检查老年代最大可用连续空间是否大于新生代所有对象的总大小。因为在极端情况下，可能新生代 YoungGC 后，所有对象都存活下来了，而 survivor 区又放不下，那可能所有对象都要进入老年代了。这个时候如果老年代的可用连续空间是大于新生代所有对象的总大小的，那就可以放心进行 YoungGC。但如果老年代的内存大小是小于新生代对象总大小的，那就有可能老年代空间不够放入新生代所有存活对象，这个时候JVM就会先检查 -XX:HandlePromotionFailure 参数是否允许担保失败，如果允许，就会判断老年代最大可用连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试进行一次YoungGC，尽快这次YoungGC是有风险的。如果小于，或者 -XX:HandlePromotionFailure 参数不允许担保失败，这时就会进行一次 Full GC。

在允许担保失败并尝试进行YoungGC后，可能会出现三种情况：

- ① YoungGC后，存活对象小于survivor大小，此时存活对象进入survivor区中
- ② YoungGC后，存活对象大于survivor大小，但是小于老年大可用空间大小，此时直接进入老年代。
- ③ YoungGC后，存活对象大于survivor大小，也大于老年大可用空间大小，老年代也放不下这些对象了，此时就会发生“Handle Promotion Failure”，就触发了 Full GC。如果 Full GC后，老年代还是没有足够的空间，此时就会发生OOM内存溢出了。

通过下图来了解空间分配担保原则：



17. 什么是类加载？类加载的过程？类加载的生命周期？

虚拟机把描述类的数据加载到内存里面，并对数据进行校验、解析和初始化，最终变成可以被虚拟机直接使用的class对象；（类的加载一直到初始化过程都是在运行期间完成的， 虽然会损失一点性能，但是却使Java 应用程序具有高度灵活的特性。Java 可以动态扩展的特性就是依赖于运行期动态加载和动态连接这个特点实现的。 ）

类的整个生命周期包括：加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)和卸载(Unloading)7个阶段。其中准备、验证、解析3个部分统称为连接（Linking）。如图所示：

加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，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再开始，这是为了支持Java语言的运行时绑定（也称为动态绑定或晚期绑定）

类加载过程如下：

- 加载，加载分为三步：
  1、通过类的全限定性类名获取该类的二进制流；
  2、将该二进制流的静态存储结构转为方法区的运行时数据结构；
  3、在内存中为该类生成一个java.lang.Class对象（ Java规范中并没有规定 Class 对象的存放位置，对于Hot Spot 虚拟机来说，Class 对象虽然是对象，但却是存放在方法区中。 ），作为方法区中这个类数据结构的访问入口
- 验证：验证该class文件中的字节流信息复合虚拟机的要求，不会威胁到jvm的安全；

  验证总体上分为四个阶段：

  1. 文件格式的验证；
  2. 元数据验证；
  3. 字节码验证；
  4. 符号引用验证；
- 准备：为Class对象的静态变量分配内存，初始化其初始值(这些变量所使用的内存都将在方法区中进行)；
- 解析： 虚拟机将常量池内的**符号引用替换为直接引用** 的过程；
  - 符号引用： 即用 **一组符号** 来描述所引用的目标。它与虚拟机的内存布局无关，引用的目标不一定已经加载到内存中。
  - 直接引用： 直接引用可以是指向目标的 **指针、相对偏移量** 或是一个能简介定位到目标的句柄。它是和虚拟机内存布局相关的。

  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 调用限定符 7类符号引用进行。
- 初始化：到了初始化阶段，才开始执行类中定义的java代码；初始化阶段是调用类构造器的过程；

17. 获取类的二进制流的方式

- 从压缩包中获取，比如 JAR包、EAR、WAR包等
- 从网络中获取，比如红极一时的Applet技术
- 从运行过程中动态生成，最出名的便是动态代理技术，在java.lang.reflect.Proxy 中，就是用了 ProxyGenerator.generateProxyClass 来为特定接口生成形式为“$Proxy”的代理类的二进制流
- 从其它文件生成，如JSP文件生成Class 类
- 从数据库中读取，比如说有些中间件服务器，通过数据库完成程序代码在集群之间的分发

17. 数组类怎么加载的

对于数组类来说，它不是由虚拟机加载得来的，而是在运行过程中直接创建的。其创建过程遵循以下规则：

- 如果数组的组件类型（数组去掉一个维度的数据类型）是引用类型，就递归使用这些引用类型的类加载器进行加载。
- 如果组件类型不是引用类型，例如 int[] 数组，Java 虚拟机会将数组标记为与引导类加载器管理
- 数组的可见性与它的组件类型可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认为public。

17. 类加载、初始化的时机

Java虚拟机规范并没有规定什么时候需要进行类的加载阶段，但是却规定5中情况必须对类进行初始化。

1. 实例化对象、读写类静态字段、调用静态方法的时候
2. 使用反射调用的时候
3. 初始化类时，需要先触发父类的初始化
4. 虚拟机启动时，会先初始化包含 main() 方法的主类
5. 使用JDK1.7 的动态语言支持的时候，如果 java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getStatic 、REF_putStatic、REF_invokeStatic的方法句柄，句柄对应的类会被初始化

以上 五种场景中会触发类的初始化， 也成为对类的主动引用

除此之外，所有引用类的方式都不会触发初始化，称为**被动引用。**

1. 通过子类引用父类中定义的静态字段，只会触发父类的初始化。至于是否会触发子类的加载和验证，取决于虚拟机的具体实现（HotSpot不会加载）。
2. 通过数组定义来引用类，如 A[] ints = new A[10] ， 不会触发A 类的初始化。而是会会触发名为 LA 的类初始化。它是一个由虚拟机自动生成的、直接继承于Object 的子类，创建动作由字节码指令 newarray 触发。这个类代表了一个元素类型为 A 的一位数组，数组中的属性和方法都实现在这个类中。Java 语言中数组的访问比C/C++ 安全是因为这个类封装了数组元素的访问方法。
3. 常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。

17. 接口加载的时机?

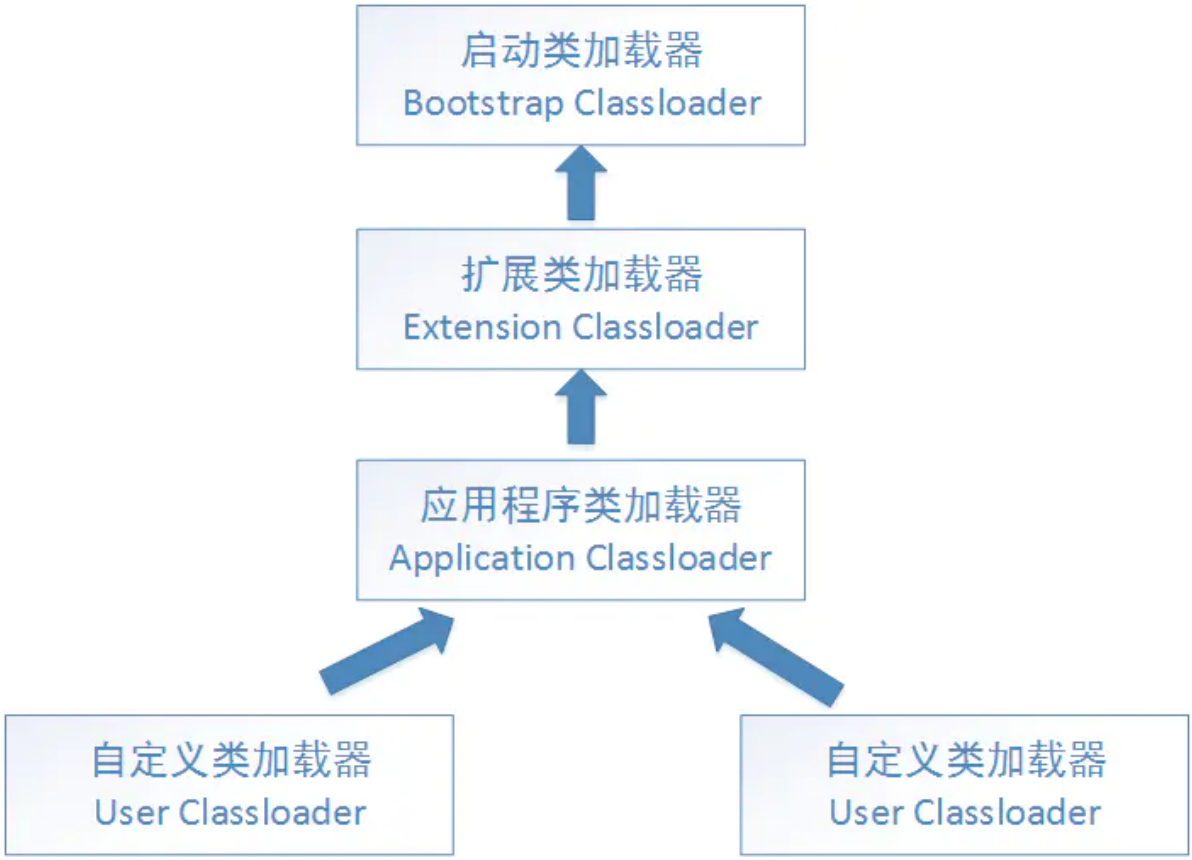
接口也有初始化过程，但是接口中不能使用static 语句块，但是编译器仍然会为接口生成 类构造器初始化接口中定义的成员变量。 接口与类真正的

区别是初始化场景中的第三种情况：当一类在初始化的时候，会要求其所有的父类都初始化完成，但是接口在初始化的时候，并不要求其父接口全部完成了初始化，只有在真正用到了父接口的时候（如引用接口中定义的常量）才会初始化。

18. 什么是类加载器，常见的类加载器有哪些?

类加载器是指：通过一个类的全限定性类名获取该类的二进制字节流叫做类加载器；类加载器分为以下四种：

- 启动类加载器（BootStrapClassLoader）：用来加载java核心类库，无法被java程序直接引用（这个类加载器用 C++ 语言实现， 是虚拟机自身的一部分： ）；
- 扩展类加载器（Extension ClassLoader）：用来加载java的扩展库，java的虚拟机实现会提供一个扩展库目录，该类加载器在扩展库目录里面查找并加载java类；
- 系统类加载器（AppClassLoader）：它根据java的类路径来加载类，一般来说，java应用的类都是通过它来加载的；
- 自定义类加载器：由java语言实现，继承自ClassLoader；



19. 什么是双亲委派模型？为什么需要双亲委派模型？

当一个类加载器收到一个类加载的请求，他首先不会尝试自己去加载，而是将这个请求委派给父类加载器去加载，只有父类加载器在自己的搜索范围类查找不到给类时，子加载器才会尝试自己去加载该类；

双亲委派机制的作用：

1、防止重复加载同一个`.class`。通过委托去向上面问一问，加载过了，就不用再加载一遍。保证数据安全。
2、保证核心`.class`不能被篡改。如果没有双亲委派的话，用户就可以自己定义一个java.lang.String类，那么就无法保证类的唯一性。

补充：**那怎么打破双亲委派模型**？

自定义类加载器，继承ClassLoader类，重写loadClass方法和findClass方法。

20. 列举一些你知道的打破双亲委派机制的例子，为什么要打破？

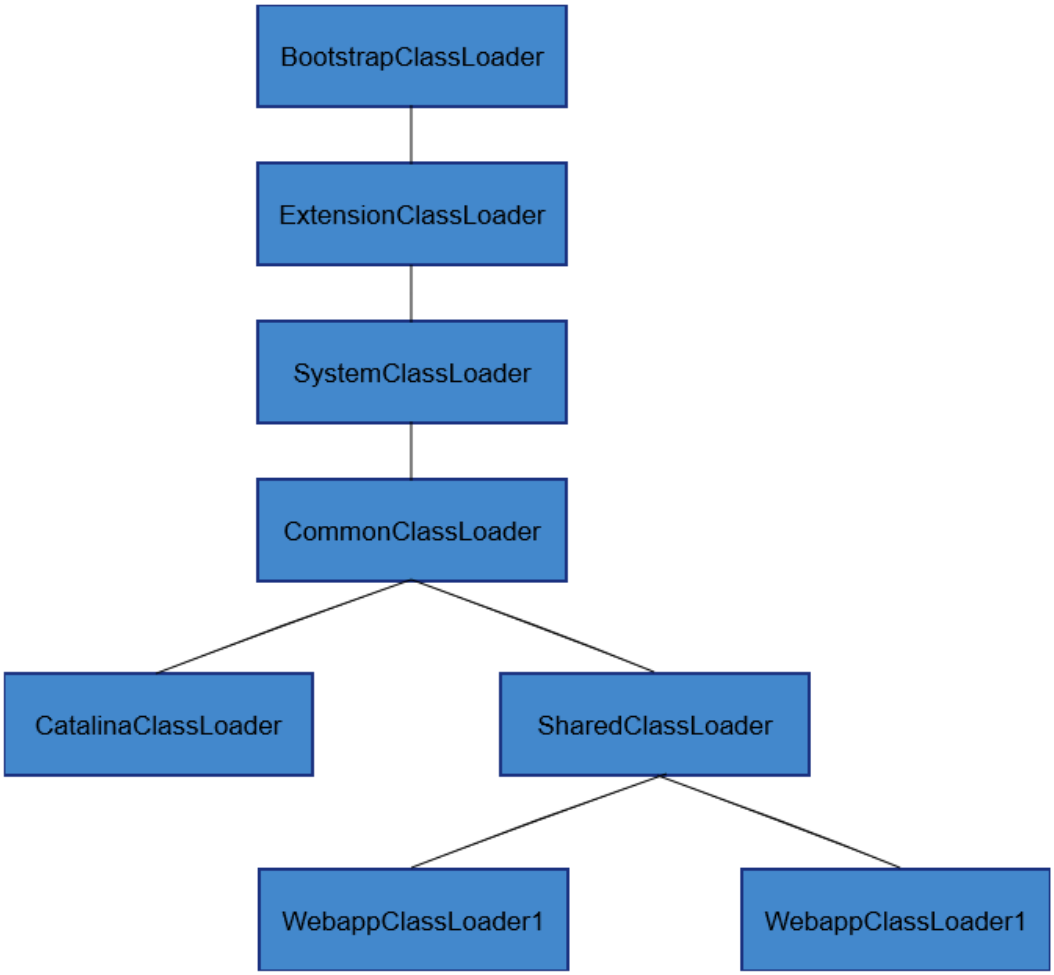
- JNDI 通过引入线程上下文类加载器，可以在 Thread.setContextClassLoader 方法设置，默认是应用程序类加载器，来加载 SPI 的代码。有了线程上下文类加载器，就可以完成父类加载器请求子类加载器完成类加载的行为。打破的原因，是为了 JNDI 服务的类加载器是启动器类加载，为了完成高级类加载器请求子类加载器（即上文中的线程上下文加载器）加载类。
- Tomcat，应用的类加载器优先自行加载应用目录下的 class，并不是先委派给父加载器，加载不了才委派给父加载器。

  tomcat之所以造了一堆自己的classloader，大致是出于下面三类目的：

  - 对于各个`webapp`中的`class`和`lib`，需要相互隔离，不能出现一个应用中加载的类库会影响另一个应用的情况，而对于许多应用，需要有共享的lib以便不浪费资源。
  - 与`jvm`一样的安全性问题。使用单独的`classloader`去装载`tomcat`自身的类库，以免其他恶意或无意的破坏；

- 热部署。

tomcat类加载器如下图：



- OSGi，实现模块化热部署，为每个模块都自定义了类加载器，需要更换模块时，模块与类加载器一起更换。其类加载的过程中，有平级的类加载器加载行为。打破的原因是为了实现模块热替换。
- JDK 9，Extension ClassLoader 被 Platform ClassLoader 取代，当平台及应用程序类加载器收到类加载请求，在委派给父加载器加载前，要先判断该类是否能够归属到某一个系统模块中，如果可以找到这样的归属关系，就要优先委派给负责那个模块的加载器完成加载。打破的原因，是为了添加模块化的特性。

21. 说一下 JVM 调优的命令?

- jps：JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。
- jstat：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。
- jmap：jmap(JVM Memory Map)命令用于生成heap dump文件，如果不使用这个命令，还阔以使用-XX:+HeapDumpOnOutOfMemoryError参数来让虚拟机出现OOM的时候·自动生成dump文件。
  jmap不仅能生成dump文件，还阔以查询finalize执行队列、Java堆和永久代的详细信息，如当前使用率、当前使用的是哪种收集器等。
- jhat：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。在此要注意，一般不会直接在服务器上进行分析，因为jhat是一个耗时并且耗费硬件资源的过程，一般把服务器生成的dump文件复制到本地或其他机器上进行分析。
- jstack：jstack用于生成java虚拟机当前时刻的线程快照。jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情，或者等待什么资源。 如果java程序崩溃生成core文

件，jstack工具可以用来获得core文件的java stack和native stack的信息，从而可以轻松地知道java程序是如何崩溃和在程序何处发生问题。

22. Java对象创建过程

1. JVM遇到一条新建对象的指令时首先去检查这个指令的参数是否能在常量池中定义到一个类的符号引用。然后加载这个类（类加载过程在后边讲）
2. 为对象分配内存。一种办法"指针碰撞"、一种办法"空闲列表"，最终常用的办法"本地线程缓冲分配(TLAB)"
3. 将除对象头外的对象内存空间初始化为0
4. 对对象头进行必要设置

23. JDK新特性

JDK8

支持 Lamda 表达式、集合的 stream 操作、提升HashMap性能

JDK9

```
//Stream API中iterate方法的新重载方法，可以指定什么时候结束迭代
IntStream.iterate(1, i -> i < 100, i -> i + 1).forEach(System.out::println);
```

默认G1垃圾回收器

JDK10

其重点在于通过完全GC并行来改善G1最坏情况的等待时间。

JDK11

ZGC (并发回收的策略) 4TB

用于 Lambda 参数的局部变量语法

JDK12

Shenandoah GC (GC 算法)停顿时间和堆的大小没有任何关系，并行关注停顿响应时间。

JDK13

增加ZGC以将未使用的堆内存返回给操作系统，16TB

JDK14

删除cms垃圾回收器、弃用ParallelScavenge+SerialOldGC垃圾回收算法组合

将ZGC垃圾回收器应用到macOS和windows平台

线上故障排查

1、硬件故障排查

如果一个实例发生了问题，根据情况选择，要不要着急去重启。如果出现的CPU、内存飙高或者日志里出现了OOM异常

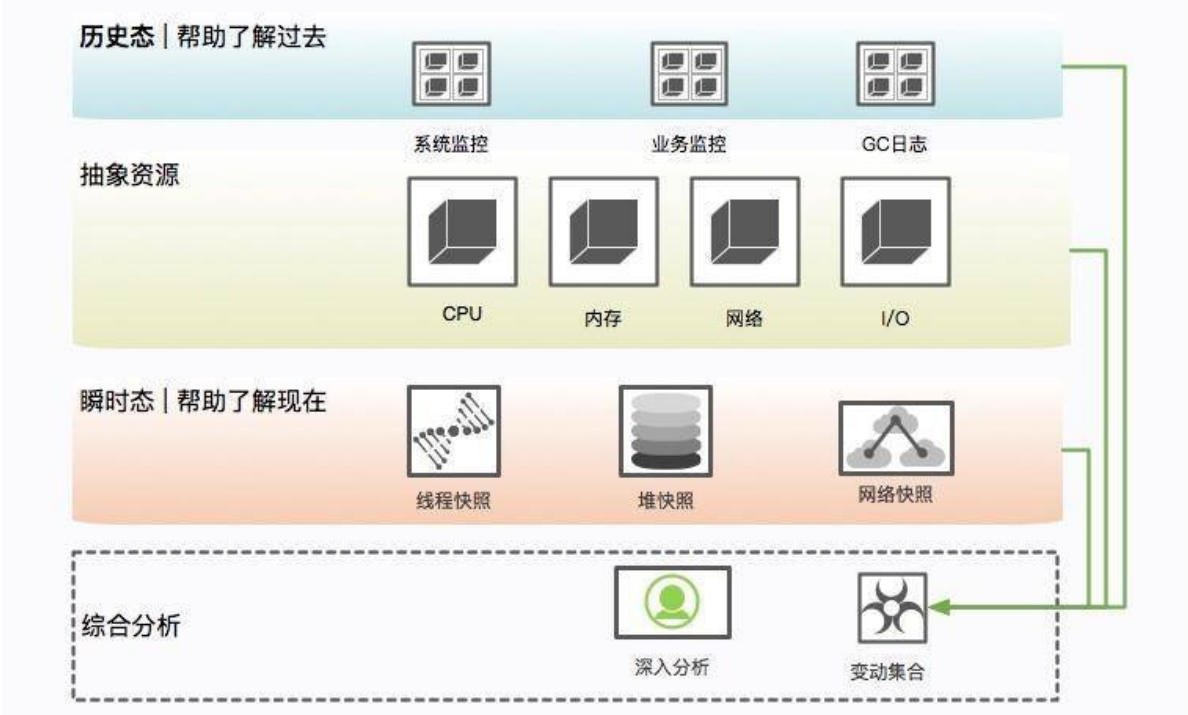
第一步是隔离，第二步是**保留现场**，第三步才是**问题排查。**

隔离

就是把你的这台机器从请求列表里摘除，比如把 nginx 相关的权重设成零。

现场保留

瞬时态和历史态



查看比如 CPU、系统内存等，通过历史状态可以体现一个趋势性问题，而这些信息的获取一般依靠监控系统的协作。

保留信息

（1）**系统当前网络连接**

```
ss -antp > $DUMP_DIR/ss.dump 2>&1
```

使用 ss 命令而不是 netstat 的原因，是因为 netstat 在网络连接非常多的情况下，执行非常缓慢。

后续的处理，可通过查看各种网络连接状态的梳理，来排查 TIME_WAIT 或者 CLOSE_WAIT，或者其他连接过高的问题，非常有用。

（2）**网络状态统计**

```
netstat -s > $DUMP_DIR/netstat-s.dump 2>&1
```

它能够按照各个协议进行统计输出，对把握当时整个网络状态，有非常大的作用。

```
sar -n DEV 1 2 > $DUMP_DIR/sar-traffic.dump 2>&1
```

在一些速度非常高的模块上，比如 Redis、Kafka，就经常发生跑满网卡的情况。表现形式就是网络通信非常缓慢。

（3）**进程资源**

```
lsof -p $PID > $DUMP_DIR/lsof-$PID.dump
```

通过查看进程，能看到打开了哪些文件，可以以进程的维度来查看整个资源的使用情况，包括每条网络连接、每个打开的文件句柄。同时，也可以很容易的看到连接到了哪些服务器、使用了哪些资源。这个命令在资源非常多的情况下，输出稍慢，请耐心等待。

（4）**CPU 资源**

```
mpstat > $DUMP_DIR/mpstat.dump 2>&1
vmstat 1 3 > $DUMP_DIR/vmstat.dump 2>&1
sar -p ALL  > $DUMP_DIR/sar-cpu.dump  2>&1
uptime > $DUMP_DIR/uptime.dump 2>&1
```

主要用于输出当前系统的 CPU 和负载，便于事后排查。

（5）**I/O 资源**

```
iostat -x > $DUMP_DIR/iostat.dump 2>&1
```

一般，以计算为主的服务节点，I/O 资源会比较正常，但有时也会发生问题，比如**日志输出过多，或者磁盘问题**等。此命令可以输出每块磁盘的基本性能信息，用来排查 I/O 问题。在第 8 课时介绍的 GC 日志分磁盘问题，就可以使用这个命令去发现。

（6）**内存问题**

```
free -h > $DUMP_DIR/free.dump 2>&1
```

free 命令能够大体展现操作系统的内存概况，这是故障排查中一个非常重要的点，比如 SWAP 影响了 GC，SLAB 区挤占了 JVM 的内存。

（7）**其他全局**

```
ps -ef > $DUMP_DIR/ps.dump 2>&1
dmesg > $DUMP_DIR/dmesg.dump 2>&1
sysctl -a > $DUMP_DIR/sysctl.dump 2>&1
```

dmesg 是许多静悄悄死掉的服务留下的最后一点线索。当然，ps 作为执行频率最高的一个命令，由于内核的配置参数，会对系统和 JVM 产生影响，所以我们也输出了一份。

（8）**进程快照**，最后的遗言（jinfo）

```
${JDK_BIN}jinfo $PID > $DUMP_DIR/jinfo.dump 2>&1
```

此命令将输出 Java 的基本进程信息，包括**环境变量和参数配置**，可以查看是否因为一些错误的配置造成了 JVM 问题。

（9）dump 堆信息

```
${JDK_BIN}jstat -gcutil $PID > $DUMP_DIR/jstat-gcutil.dump 2>&1
${JDK_BIN}jstat -gccapacity $PID > $DUMP_DIR/jstat-gccapacity.dump 2>&1
```

jstat 将输出当前的 gc 信息。一般，基本能大体看出一个端倪，如果不能，可将借助 jmap 来进行分析。

（10）堆信息

```
${JDK_BIN}jmap $PID > $DUMP_DIR/jmap.dump 2>&1
${JDK_BIN}jmap -heap $PID > $DUMP_DIR/jmap-heap.dump 2>&1
${JDK_BIN}jmap -histo $PID > $DUMP_DIR/jmap-histo.dump 2>&1
${JDK_BIN}jmap -dump:format=b,file=$DUMP_DIR/heap.bin $PID > /dev/null  2>&1
```

jmap 将会得到当前 Java 进程的 dump 信息。如上所示，其实最有用的就是第 4 个命令，但是前面三个能够让你初步对系统概况进行大体判断。因为，第 4 个命令产生的文件，一般都非常的大。而且，需要下载下来，导入 MAT 这样的工具进行深入分析，才能获取结果。这是分析内存泄漏一个必经的过程。

（11）JVM 执行栈

```
${JDK_BIN}jstack $PID > $DUMP_DIR/jstack.dump 2>&1
```

jstack 将会获取当时的执行栈。一般会多次取值，我们这里取一次即可。这些信息非常有用，能够还原 Java 进程中的线程情况。

```
top -Hp $PID -b -n 1 -c >  $DUMP_DIR/top-$PID.dump 2>&1
```

为了能够得到更加精细的信息，我们使用 top 命令，来获取进程中所有线程的 CPU 信息，这样，就可以看到资源到底耗费在什么地方了。

（12）高级替补

```
kill -3 $PID
```

有时候，jstack 并不能够运行，有很多原因，比如 Java 进程几乎不响应了等之类的情况。我们会尝试向进程发送 kill -3 信号，这个信号将会打印 jstack 的 trace 信息到日志文件中，是 jstack 的一个替补方案。

```
gcore -o $DUMP_DIR/core $PID
```

对于 jmap 无法执行的问题，也有替补，那就是 GDB 组件中的 gcore，将会生成一个 core 文件。我们可以使用如下的命令去生成 dump：

```
${JDK_BIN}jhsdb jmap --exe ${JDK}java  --core $DUMP_DIR/core --binaryheap
```

3. **内存泄漏的现象**

稍微提一下 jmap 命令，它在 9 版本里被干掉了，取而代之的是 jhsdb，你可以像下面的命令一样使用。

```
jhsdb jmap  --heap --pid  37340
jhsdb jmap  --pid  37288
jhsdb jmap  --histo --pid  37340
jhsdb jmap  --binaryheap --pid  37340
```

一般内存溢出，表现形式就是 Old 区的占用持续上升，即使经过了多轮 GC 也没有明显改善。比如 ThreadLocal里面的GC Roots，内存泄漏的根本就是，这些对象并没有切断和 GC Roots 的关系，可通过一些工具，能够看到它们的联系。

2、报表异常 | JVM调优

有一个报表系统，频繁发生内存溢出，在高峰期间使用时，还会频繁的发生拒绝服务，由于大多数使用者是管理员角色，所以很快就反馈到研发这里。

业务场景是由于有些结果集的字段不是太全，因此需要对结果集合进行循环，并通过 HttpClient 调用其他服务的接口进行数据填充。使用 Guava 做了 JVM 内缓存，但是响应时间依然很长。

初步排查，JVM 的资源太少。接口 A 每次进行报表计算时，都要涉及几百兆的内存，而且在内存里驻留很长时间，有些计算又非常耗 CPU，特别的“吃”资源。而我们分配给 JVM 的内存只有 3 GB，在多人访问这些接口的时候，内存就不够用了，进而发生了 OOM。在这种情况下，没办法，只有升级机器。把机器配置升级到 4C8G，给 JVM 分配 6GB 的内存，这样 OOM 问题就消失了。但随之而来的是频繁的 GC 问题和超长的 GC 时间，平均 GC 时间竟然有 5 秒多。

进一步，由于报表系统和高并发系统不太一样，它的对象，存活时长大得多，并不能仅仅通过增加年轻代来解决；而且，如果增加了年轻代，那么必然减少了老年代的大小，由于 CMS 的碎片和浮动垃圾问题，我们可用的空间就更少了。虽然服务能够满足目前的需求，但还有一些不太确定的风险。

第一，了解到程序中有很多缓存数据和静态统计数据，为了减少 MinorGC 的次数，通过分析 GC 日志打印的对象年龄分布，把 MaxTenuringThreshold 参数调整到了 3（特殊场景特殊的配置）。这个参数是让年轻代的这些对象，赶紧回到老年代去，不要老呆在年轻代里。

第二，我们的 GC 时间比较长，就一块开了参数 CMSScavengeBeforeRemark，使得在 CMS remark 前，先执行一次 Minor GC 将新生代清掉。同时配合上个参数，其效果还是比较好的，一方面，对象很快晋升到了老年代，另一方面，年轻代的对象在这种情况下是有限的，在整个 MajorGC 中占的时间也有限。

第三，由于缓存的使用，有大量的弱引用，拿一次长达 10 秒的 GC 来说。我们发现在 GC 日志里，处理 weak refs 的时间较长，达到了 4.5 秒。这里可以加入参数 ParallelRefProcEnabled 来并行处理 Reference，以加快处理速度，缩短耗时。

优化之后，效果不错，但并不是特别明显。经过评估，针对高峰时期的情况进行调研，我们决定再次提升机器性能，改用 8core16g 的机器。但是，这带来另外一个问题。

高性能的机器带来了非常大的服务吞吐量，通过 jstat 进行监控，能够看到年轻代的分配速率明显提高，但随之而来的 MinorGC 时长却变的不可控，有时候会超过 1 秒。累积的请求造成了更加严重的后果。

这是由于堆空间明显加大造成的回收时间加长。为了获取较小的停顿时间，我们在堆上**改用了 G1 垃圾回收器**，把它的目标设定在 200ms。G1 是一款非常优秀的垃圾收集器，不仅适合堆内存大的应用，同时也简化了调优的工作。通过主要的参数初始和最大堆空间、以及最大容忍的 GC 暂停目标，就能得到不错的性能。修改之后，虽然 GC 更加频繁了一些，但是停顿时间都比较小，应用的运行较为平滑。

到目前为止，也只是勉强顶住了已有的业务，但是，这时候领导层面又发力，**要求报表系统可以支持未来两年业务10到100倍的增长**，并保持其可用性，但是这个“千疮百孔”的报表系统，稍微一压测，就宕机，那如何应对十倍百倍的压力呢？硬件即使可以做到动态扩容，但是毕竟也有极限。

使用 MAT 分析堆快照，发现很多地方可以通过代码优化，那些占用内存特别多的对象：

1、select * 全量排查，只允许获取必须的数据

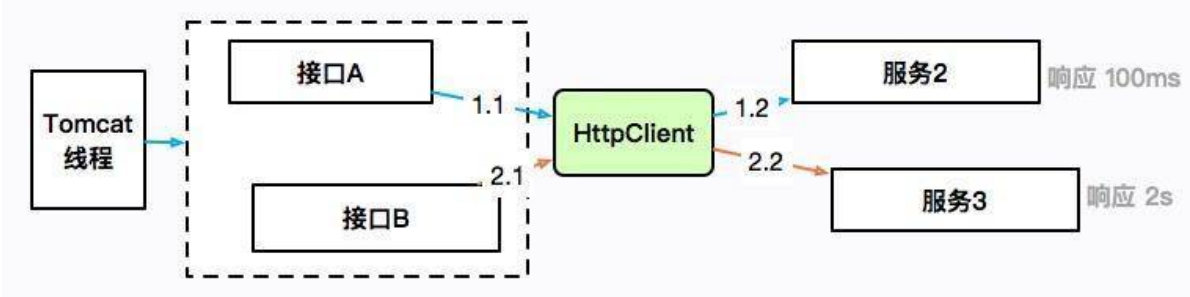
2、报表系统中cache实际的命中率并不高，将Guava 的 Cache 引用级别改成弱引用（WeakKeys）

3、限制报表导入文件大小，同时拆分用户超大范围查询导出请求。

每一步操作都使得JVM使用变得更加可用，一系列优化以后，机器相同压测数据性能提升了数倍。

3、大屏异常 | JUC调优

有些数据需要使用 HttpClient 来获取进行补全。提供数据的服务提供商有的响应时间可能会很长，也有可能会造成服务整体的阻塞。



接口 A 通过 HttpClient 访问服务 2，响应 100ms 后返回；接口 B 访问服务 3，耗时 2 秒。HttpClient 本身是有一个最大连接数限制的，如果服务 3 迟迟不返回，就会造成 HttpClient 的连接数达到上限，**概括来讲，就是同一服务，由于一个耗时非常长的接口，进而引起了整体的服务不可用**

这个时候，通过 jstack 打印栈信息，会发现大多数竟然阻塞在了接口 A 上，而不是耗时更长的接口 B，这个现象起初十分具有迷惑性，不过经过分析后，我们猜想其实是因为接口 A 的速度比较快，在问题发生点进入了更多的请求，它们全部都阻塞住的同时被打印出来了。

为了验证这个问题，我搭建了一个demo 工程，模拟了两个使用同一个 HttpClient 的接口。fast 接口用来访问百度，很快就能返回；slow 接口访问谷歌，由于众所周知的原因，会阻塞直到超时，大约 10 s。利用ab对两个接口进行压测，同时使用 jstack 工具 dump 堆栈。首先使用 jps 命令找到进程号，然后把结果重定向到文件（可以参考 10271.jstack 文件）。

过滤一下 nio 关键字，可以查看 tomcat 相关的线程，足足有 200 个，这和 Spring Boot 默认的 maxThreads 个数不谋而合。更要命的是，有大多数线程，都处于 BLOCKED 状态，说明线程等待资源超时。通过grep fast | wc -l 分析，确实200个中有150个都是blocked的fast的进程。

问题找到了，解决方式就顺利成章了。

1、fast和slow争抢连接资源，通过线程池限流或者熔断处理

2、有时候slow的线程也不是一直slow，所以就得加入监控

3、使用带countdownLaunch对线程的执行顺序逻辑进行控制

4、接口延迟 | SWAP调优

有一个关于服务的某个实例，经常发生服务卡顿。由于服务的并发量是比较高的，每多停顿 1 秒钟，几万用户的请求就会感到延迟。

我们统计、类比了此服务其他实例的 CPU、内存、网络、I/O 资源，区别并不是很大，所以一度怀疑是机器硬件的问题。

接下来我们对比了节点的 GC 日志，发现无论是 Minor GC，还是 Major GC，这个节点所花费的时间，都比其他实例长得多。

通过仔细观察，我们发现在 GC 发生的时候，vmstat 的 si、so 飙升的非常严重，这和其他实例有着明显的不同。

使用 free 命令再次确认，发现 SWAP 分区，使用的比例非常高，引起的具体原因是什么呢?

更详细的操作系统内存分布，从 /proc/meminfo 文件中可以看到具体的逻辑内存块大小，有多达 40 项的内存信息，这些信息都可以通过遍历 /proc 目录的一些文件获取。我们注意到 slabtop 命令显示的有一些异常，dentry（目录高速缓冲）占用非常高。

问题最终定位到是由于某个运维工程师删除日志时，定时执行了一句命令：

find / | grep "xxx.log"

他是想找一个叫做 要被删除 的日志文件，看看在哪台服务器上，结果，这些老服务器由于文件太多，扫描后这些文件信息都缓存到了 slab 区上。而服务器开了 swap，操作系统发现物理内存占满后，并没有立即释放 cache，导致每次 GC 都要和硬盘打一次交道。

解决方式就是关闭 SWAP 分区。

swap 是很多性能场景的万恶之源，建议禁用。在高并发 SWAP 绝对能让你体验到它魔鬼性的一面：进程倒是死不了了，但 GC 时间长的却让人无法忍受。

5、内存溢出 | Cache调优

> 有一次线上遇到故障，重新启动后，使用 jstat 命令，发现 Old 区一直在增长。我使用 jmap 命令，导出了一份线上堆栈，然后使用 MAT 进行分析，通过对 GC Roots 的分析，发现了一个非常大的 HashMap 对象，这个原本是其他同事做缓存用的，但是做了一个无界缓存，没有设置超时时间或者 LRU 策略，在使用上又没有重写key类对象的hashcode和equals方法，对象无法取出也直接造成了堆内存占用一直上升，后来，将这个缓存改成 guava 的 Cache，并设置了弱引用，故障就消失了。
>
> 关于文件处理器的应用，在读取或者写入一些文件之后，由于发生了一些异常，**close 方法又没有放在 finally** 块里面，造成了文件句柄的泄漏。由于文件处理十分频繁，产生了严重的内存泄漏问题。

内存溢出是一个结果，而**内存泄漏**是一个原因。内存溢出的原因有**内存空间不足、配置错误**等因素。一些错误的编程方式，不再被使用的对象、没有被回收、没有及时切断与 GC Roots 的联系，这就是内存泄漏。

举个例子，有团队使用了 HashMap 做缓存，但是并没有设置超时时间或者 LRU 策略，造成了放入 Map 对象的数据越来越多，而产生了内存泄漏。

再来看一个经常发生的内存泄漏的例子，也是由于 HashMap 产生的。代码如下，由于没有重写 Key 类的 hashCode 和 equals 方法，造成了放入 HashMap 的所有对象都无法被取出来，它们和外界失联了。所以下面的代码结果是 null。

```
//leak example
import java.util.HashMap;
import java.util.Map;
public class HashMapLeakDemo {
    public static class Key {
        String title;
    public Key(String title) {
        this.title = title;
    }
}

public static void main(String[] args) {
    Map<Key, Integer> map = new HashMap<>();
    map.put(new Key("1"), 1);
```

```
    map.put(new Key("2"), 2);
    map.put(new Key("3"), 2);
    Integer integer = map.get(new Key("2"));
    System.out.println(integer);
    }
}
```

即使提供了 equals 方法和 hashCode 方法，也要非常小心，尽量避免使用自定义的对象作为 Key。

再看一个例子，关于文件处理器的应用，在读取或者写入一些文件之后，由于发生了一些异常，**close 方法又没有放在 finally** 块里面，造成了文件句柄的泄漏。由于文件处理十分频繁，产生了严重的内存泄漏问题。

6、CPU飙高 | 死循环

我们有个线上应用，单节点在运行一段时间后，CPU 的使用会飙升，一旦飙升，一般怀疑某个业务逻辑的计算量太大，或者是触发了死循环（比如著名的 HashMap 高并发引起的死循环），但排查到最后其实是 GC 的问题。

（1）使用 top 命令，查找到使用 CPU 最多的某个进程，记录它的 pid。使用 Shift + P 快捷键可以按 CPU 的使用率进行排序。

```
top
```

（2）再次使用 top 命令，加 -H 参数，查看某个进程中使用 CPU 最多的某个线程，记录线程的 ID。

```
top -Hp $pid
```

（3）使用 printf 函数，将十进制的 tid 转化成十六进制。

```
printf %x $tid
```

（4）使用 jstack 命令，查看 Java 进程的线程栈。

```
jstack $pid >$pid.log
```

（5）使用 less 命令查看生成的文件，并查找刚才转化的十六进制 tid，找到发生问题的线程上下文。

```
less $pid.log
```

我们在 jstack 日志搜关键字DEAD，以及中找到了 CPU 使用最多的几个线程id。

可以看到问题发生的根源，是我们的堆已经满了，但是又没有发生 OOM，于是 GC 进程就一直在那里回收，回收的效果又非常一般，造成 CPU 升高应用假死。接下来的具体问题排查，就需要把内存 dump 一份下来，使用 MAT 等工具分析具体原因了。