1.JVM的内存划分

JVM中的内存主要划分为5个区域，即**方法区，堆内存，程序计数器，虚拟机栈以及本地方法栈。**

方法区：方法区是一个**线程之间共享**的区域。**常量，静态变量以及JIT编译后的代码都在方法区。主要用于存储已被虚拟机加载的类信息**，也可以称为“永久代”，垃圾回

收效果一般，通过-XX：MaxPermSize控制上限。

堆内存：堆内存是**垃圾回收**的主要场所，也是线程之间共享的区域，主要用来**存储创建的对象实例**，通过-Xmx 和-Xms 可以控制大小。

虚拟机栈（栈内存）：栈内存中主要保存**局部变量、基本数据类型变量**以及**堆内存中某个对象的引用变量**。每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表，操作数栈，动态链接，方法出口等信息。栈中的栈帧随着方法的进入和退出有条不紊的执行着出栈和入栈的操作。

2. 对象创建的过程

1.类加载检查

当JVM检测到有一条new指令时，首先先检查该指令的参数是否在**常量池**中定位到一个类的符号引用，并检查这个符号引用所代表的类是否已被加载、解析和初始化过。如果存在的话，JVM将直接使用已有的信息对该类进行操作。

如果没有，则执行相应的类加载过程。

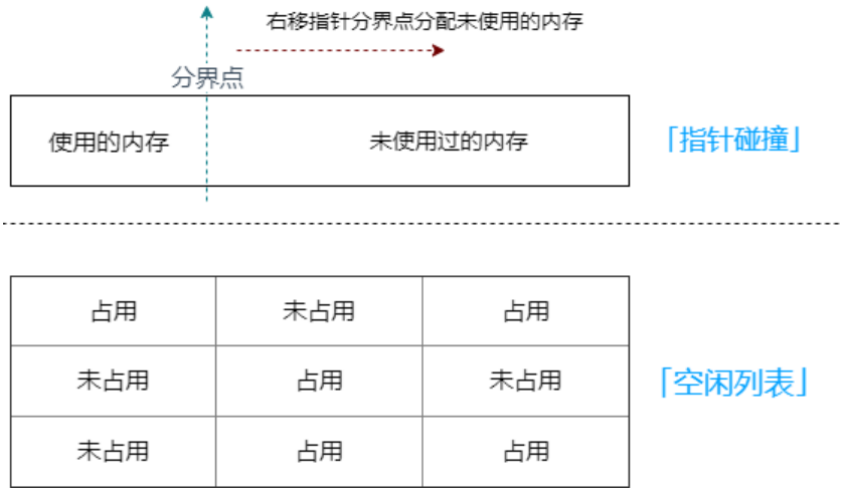
2.分配内存空间

类加载检查通过后，虚拟机为新生对象分配内存，对象所需内存大小在类加载完成后便可以完全确定，为对象分配空间无非就是从**Java堆中**划分出一块确定大小的内存而已。

不同的JVM垃圾收集器在分配内存时的表现也不相同，具体表现为两种：

（1）如果垃圾收集器选择的是Serial、ParNew这种基于压缩整理算法的，那么内存是规整的，虚拟机将采用的是**指针碰撞法**来为对象分配内存。意思是所有用过的内存在一边，空闲的内存在另外一边，中间放着一个指针作为分界点的指示器，分配内存就仅仅是把指针向空闲那边挪动一段与对象大小相等的距离罢了。

（2）如果垃圾收集器选择的是CMS这种基于标记-清除算法的，那么内存不是规整的，已使用的内存和未使用的内存相互交错，虚拟机将采用的是空闲列表法来为对象分配内存。意思是虚拟机维护了一个列表，记录上哪些内存块是可用的以及内存块的位置和大小，再分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的内容。



另外一个问题是new对象时的线程安全性，也就是内存分配时的同步问题。因为可能出现虚拟机正在给对象A分配内存，指针还没有来得及修改，对象B又同时使用了原来的指针来分配内存的情况。这种情况下虚拟机会通过两种方式进行同步：

a、**CAS和失败重试机制**：对分配内存空间的动作进行同步处理，虚拟机采用CAS配上失败重试的方式保证更新操作的原子性。CAS简单解释就是：比较并交换，通过3\操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

b、**TLAB方式**：把内存的分配动作按照线程划分在不同的空间之中进行，即每个线程在Java堆中先预留一块本地线程分配缓冲（TLAB）。哪个线程分配内存时，就在哪个线程的TLAB分配，只有当TLAB用完并分配新的TLAB时，才需要同步锁定。

3.设置对象头

对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息。这些信息存放在对象的对象头之中

4.执行init，初始化对象

https://www.cnblogs.com/haitaofeiyang/p/7767919.html

2.对象的内存布局

对象的内存布局分为三个区域：

a、对象头，b、实例数据，c、对齐填充。

- 对象头：非固定的数据结构。一来是用来存储对象自身的运行时数据，如哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等。二来是类型指针，即对象指向它的类元数据的指针、JVM通过这个指针来确定这个对象是哪个类的实例。如果对象是一个Java数组，则在对象头中还需要有一块记录数组长度数据。

- 实例数据：存储对象真正有效的信息，也就是程序代码中所定义的各种类型的字段内容。不论是从父类继承下来的，还是在子类中定义的。这部分的存储顺序会受到Java源码中定义顺序的影响。
- 对齐填充：不一定必须存在。启到占位符的作用。因为JVM的自动内存管理系统要求对象的起始地址必须是8字节的整数倍，即对象的大小必须是8字节的整数倍。故当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

3.对象访问机制

所以对象访问方式也是由虚拟机实现而定的，主流的访问方式主要有**「使用句柄」**和**「直接指针」**两种

- 使用句柄访问的话，Java 堆中将可能会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自具体的地址信息，其结构如下图所示。
- 使用直接指针访问的话，Java 堆中对象的内存布局就必须考虑如何放置访问类型数据的相关信息，reference 中存储的直接就是对象地址，如果只是访问对象本身的话，就不需要多一次间接访问的开销，如下图所示。

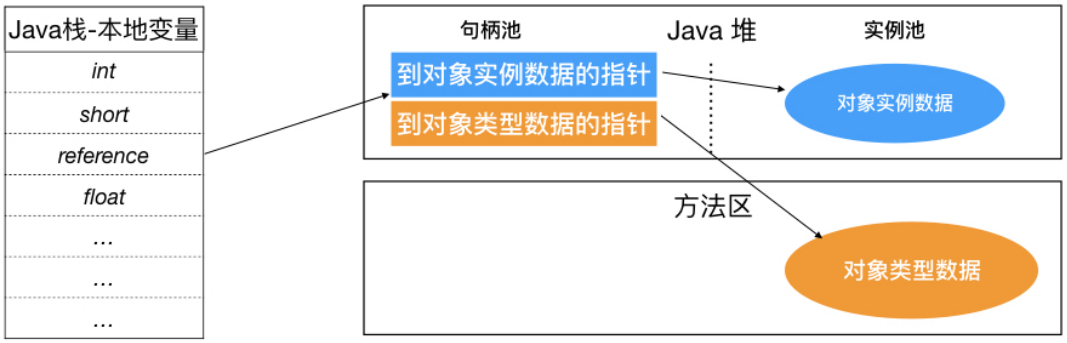


图-通过句柄访问对象

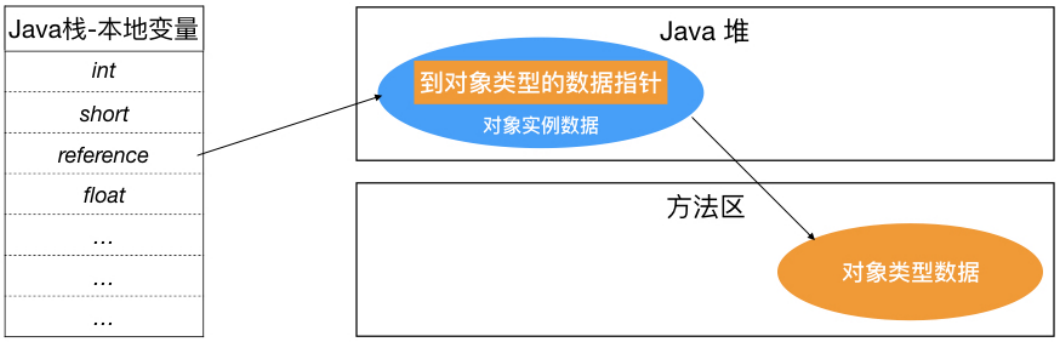


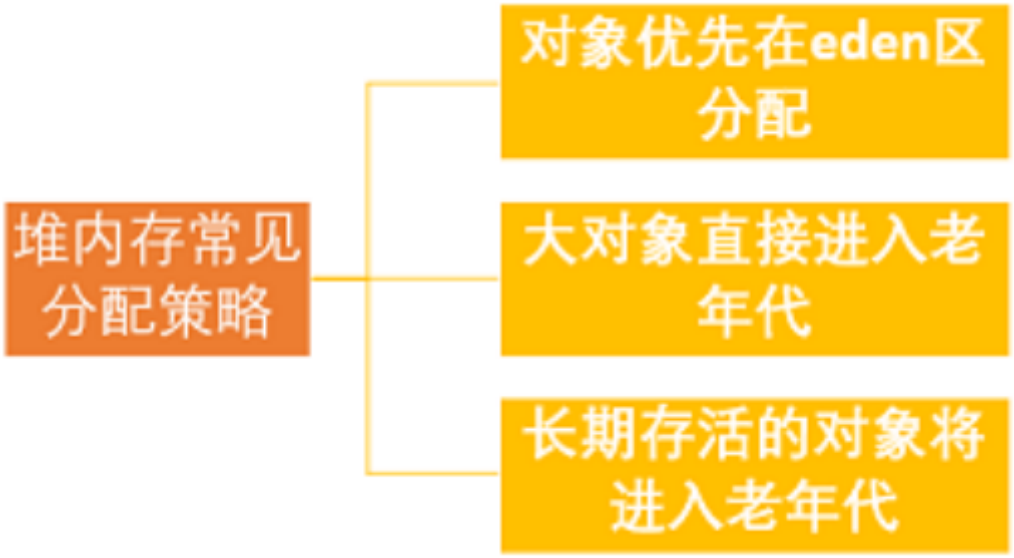
图-通过直接指针访问对象

访问方式比较：

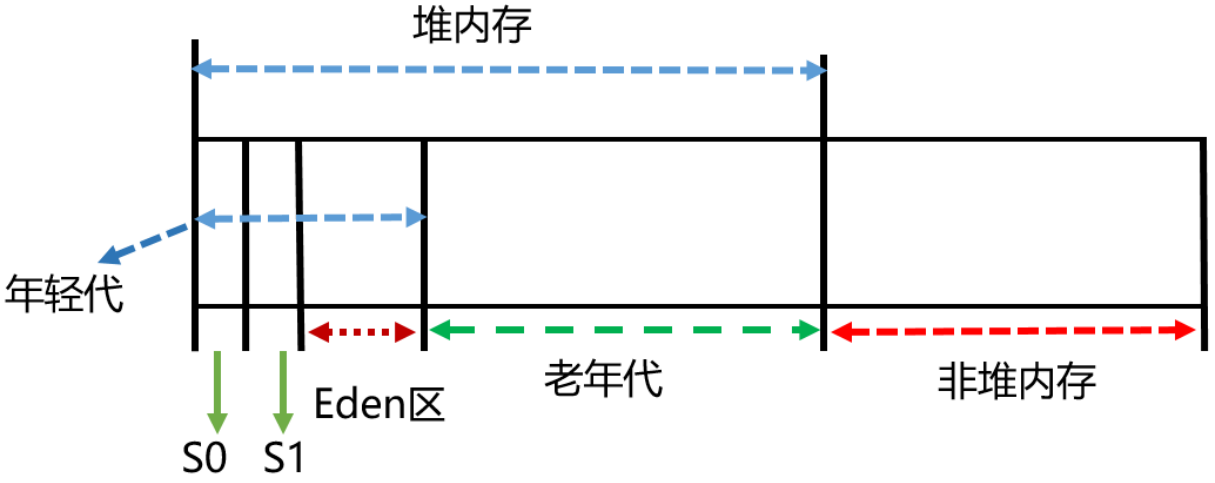
句柄访问的最大好处就是 reference 中存储的是稳定句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而 reference 本身不需要被修改。

直接指针来访问最大的好处就是速度更快，它节省了一次指针定位的时间开销，由于对象访问在 Java 中非常频繁，因此这类开销积少成多也是一项极为可观的执行成本，

4.内存分配与垃圾回收



堆内存分为**年轻代和老年代。**年轻代又可以进一步划分为**一个Eden（伊甸）区和两个Survivor（幸存）区**组成。如下图所示：



JVM内存简单分配示意图

4.1JVM堆内存的分配：

JVM初始分配的堆内存由**-Xms**指定，默认是物理内存的1/64。JVM最大分配的堆内存由**-Xmx**指定，默认是物理内存的1/4。

扩容机制:

默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制。空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制。因此我们一般设置-Xms和-Xmx相等以避免在每次GC 后调整堆的大小。

长期存活的对象将进入老年代，每一次**MinorGC（年轻代GC）**，对象年龄就大一岁，默认15岁晋升到老年代

4.2JVM堆区新生代为什么有两个Survivor

from to 过程

young gc过程:

年轻代中分为一个Eden区和两个Surviver区，比例为8：1：1，两个Surviver区分别称为“From”区和“To”区。对象在Eden区创建，经过一次Yong GC后，还存活的对象将会被复制到Surviver区的“From”区，此时“To”区是空的。到了下一次GC的时候，Eden区还存活的对象会直接移动到Surviver区的“To”区，而“Form”区的对象有两个去处，“From”区的对象会根据经过的GC次数计算年龄，如果年龄到达了阈值（默认15），则会被移动到老年代中，否则就复制到“To”区，此时“From”区变成了空的，然后“From”区和“To”区进行角色互换，到下一次进行GC时，还是有一块空的“To”区，用来存放从eden区和“From”区移动过来的对象。

每次YGC发生之后，S1和S2总会有一个是空的，这样子的目的是避免内存碎片化带来的空间与性能损失.

4.3 Minor GC和Full GC(Major GC)

大多数情况下,内存分配在Eden区,当Eden区内存不够时,jvm将发起一次Minor GC

新生代GC(Minor GC):指发生在新生代的垃圾收集,Minor GC非常频繁,回收速度也很快

老年代GC(Major GC):指发生在老年代的垃圾回收,一般伴随一次Minor GC,频率很低,并且回收速度慢很多

4.4 GC发生的时间

也就是GC会在什么时候触发，主要有以下几种触发条件：

- 执行System.gc()的时候：建议执行Full GC，但是JVM并不保证一定会执行
- 新生代空间不足（下面会详细展开）
- 老年代空间不足（下面会详细展开）

对象大都在Eden区分配内存，如果某个时刻JVM需要给某一个对象在Eden区上分配一块内存，但是此时Eden区剩余的连续内存小于该对象需要的内存，Eden区空间不足会触发minor GC。触发minor GC前会检查之前每次Minor GC时晋升到老年代的平均对象大小是否大

于老年代剩余空间大小，如果大于，则直接触发Full GC；否则，查看HandlePromotionFailure参数的值，如果为false，则直接触发Full GC；如果为true（默认为true，表示允许担保失败，虽然剩余空间大于之前晋升到老年代的平均大小，但是依旧可能担保失败），则仅触发Minor GC，如果期间发生老年代不足以容纳新生代存活的对象，此时会触发Full GC 。

老年代满了，会触发Full GC（回收整个堆内存）。关于老年代：

1. 分配很大的对象：大对象直接进入老年代，经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够多的连续空间；
2. 长期存活的对象将进入老年代；
3. 如果survivor空间中相同年龄所有对象大小的总和大于survivor空间的一半，年龄大于等于该年龄的对象就可以直接进入老年代；
4. CMS GC在出现promotion failure和concurrent mode failure的时候

5.JVM如何判断对象能否被回收

1.引用计数算法

无法解决循环引用的问题,已经弃用

2.root根搜索方法

root搜索方法的基本思路就是通过一系列可以做为root的对象作为起始点，从这些节点开始向下搜索。当一个对象到root节点没有任何引用链接时，则证明此对象是可以被回收的。以下对象会被认为是root对象.

- 虚拟机栈（栈帧中的本地变量表）中引用的对象；
- 方法区中类静态属性引用的对象；
- 方法区常量引用的对象；
- 本地方法栈中JNI（即一般说的Native方法）引用的对象；

3.JVM内存溢出

1. 堆内存溢出

   OutOfMemoryError

2. 虚拟机栈/本地方法栈溢出

   StackOverflowError 虚拟机栈中的栈帧数量过多（一个线程嵌套调用的方法数量过多）时，就会抛出StackOverflowError异常

   OutOfMemoryError：如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError。

6.JVM垃圾回收算法

标记-清除算法，复制算法和标记整理算法

标记-清除算法（Mark-Sweep）：

标记-清除算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，并且会产生内存碎片。

复制算法：

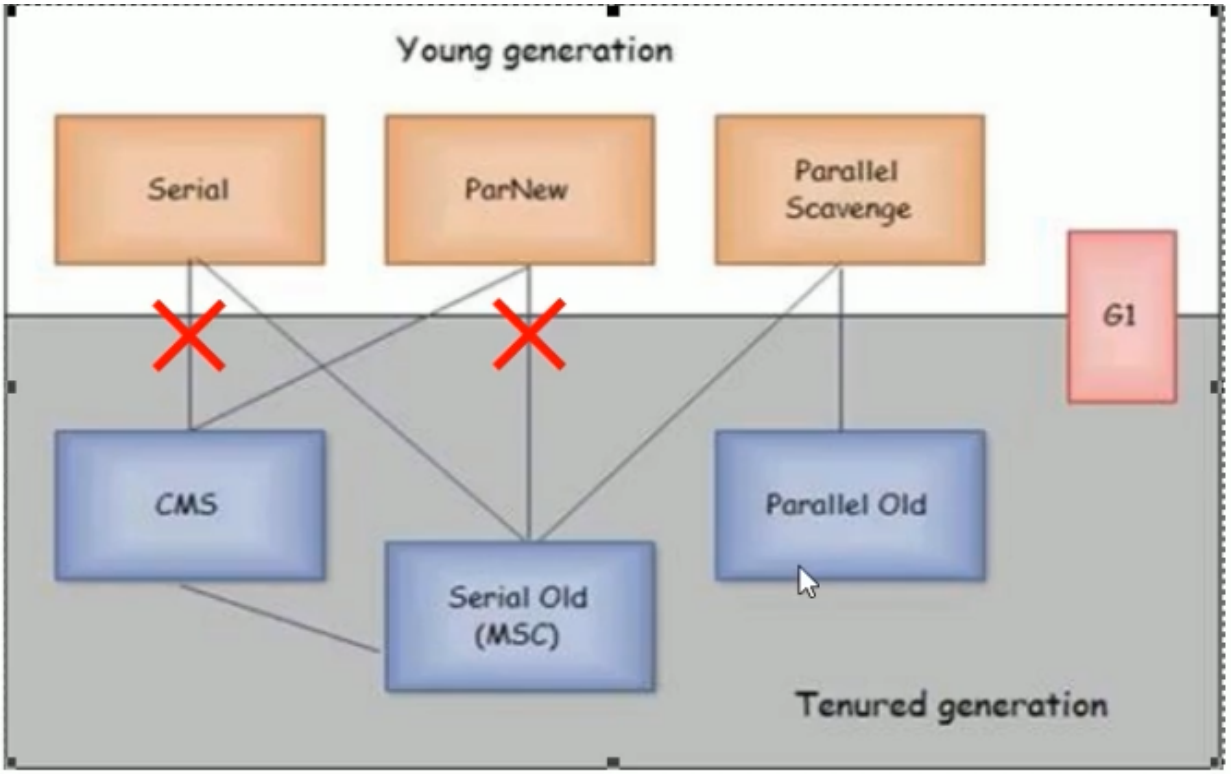
复制算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。复制算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。

标记-整理算法：

标记-整理算法结合了**“标记-清除”和“复制”**两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

7.JVM中的垃圾收集器

JVM中的垃圾收集器主要包括7种，即**Serial，Serial Old，ParNew，Parallel Scavenge，Parallel Old以及CMS，G1收集器。**



Serial	新生代	单线程复制	**Client 模式（桌面应用）；单核服务器。**
ParNew	新生代	多线程复制	**多核服务器；与 CMS 收集器搭配使用**
Parallel Scavenge	新生代		**注重吞吐量，高效利用 CPU，需要高效运算且不需要太多交互**
Serial Old	老年代	单线程标记整理	**弃用**
Parallel Old	老年代	多线程标记整理	**与Parallel Scavenge 收集器搭配使用；注重吞吐量。jdk7、jdk8 默认**
CMS	老年代	多线程标记清除	**重视服务器响应速度，要求系统停顿时间最短。**
G1	不区分	多线程标记清除	

新生代收集器：Serial/ParNew/Parallel Scavenge

老年代收集器： Serial Old/CMS/Parallel Old

堆内存垃圾收集器： G1

新生代收集器

（1）Serial 收集器

Serial 是一款用于新生代的单线程收集器，采用复制算法进行垃圾收集。Serial 进行垃圾收集时，不仅只用一条线程执行垃圾收集工作，它在收集的同时，所有的用户线程必须暂停（Stop The World）。

适用场景：Client 模式（桌面应用）；单核服务器。

（2）ParNew 收集器

ParNew 就是一个 Serial 的多线程版本，其它与Serial并无区别。ParNew 在单核 CPU 环境并不会比 Serial 收集器达到更好的效果，它默认开启的收集线程数和 CPU 数量一致，可以通过 -XX:ParallelGCThreads 来设置垃圾收集的线程数。

适用场景：多核服务器；与 CMS 收集器搭配使用

（3）Parallel Scavenge 收集器

Parallel Scavenge 也是一款用于新生代的多线程收集器，与 ParNew 的不同之处是ParNew 的目标是尽可能缩短垃圾收集时用户线程的停顿时间，Parallel Scavenge 的目标是达到一个可控制的吞吐量。

适用场景：注重吞吐量，高效利用 CPU，需要高效运算且不需要太多交互。

老年代收集器

（1）Serial Old 收集器

Serial Old 收集器是 Serial 的老年代版本，同样是一个单线程收集器，采用标记-整理算法。

适用场景：Client 模式（桌面应用）；单核服务器；与 Parallel Scavenge 收集器搭配；作为 CMS 收集器的后备预案。

（2）CMS(Concurrent Mark Sweep) 收集器

多线程收集

CMS 收集器是一种以最短回收停顿时间为目标的收集器，以 “ 最短用户线程停顿时间 ” 著称。整个垃圾收集过程分为 4 个步骤：

① **初始标记**：标记一下 GC Roots 能直接关联到的对象，速度较快。

② **并发标记**：进行 GC Roots Tracing，标记出全部的垃圾对象，耗时较长。

③ **重新标记**：修正并发标记阶段引用户程序继续运行而导致变化的对象的标记记录，耗时较短。

④ **并发清除**：用标记-清除算法清除垃圾对象，耗时较长。

适用场景：重视服务器响应速度，要求系统停顿时间最短。

（3）Parallel Old 收集器

Parallel Old 收集器是 Parallel Scavenge 的老年代版本，是一个多线程收集器，采用标记-整理算法。可以与 Parallel Scavenge 收集器搭配，可以充分利用多核 CPU 的计算能力。

适用场景：与Parallel Scavenge 收集器搭配使用；注重吞吐量。jdk7、jdk8 默认使用该收集器作为老年代收集器

堆收集器

G1收集器的特点：

- **并行与并发：G1能充分利用多CPU，多核环境下的硬件优势，来缩短Stop the World，是并发的收集器。**
- **分代收集：G1不需要其他收集器就能独立管理整个GC堆，能够采用不同的方式去处理新建对象、存活一段时间的对象和熬过多次GC的对象。**
- **空间整合：G1从整体来看是基于标记-整理算法，从局部（两个Region）上看基于复制算法实现，G1运作期间不会产生内存空间碎片。**
- **可预测的停顿：能够建立可以预测的停顿时间模型，预测停顿时间。**

和CMS收集器类似，G1收集器的垃圾回收工作也分为了四个阶段：

- 初始标记
- 并发标记
- 最终标记
- 筛选回收

其中，筛选回收阶段首先对各个Region的回收价值和成本进行计算，根据用户期望的GC停顿时间来制定回收计划。

-XX:+UseSerialGC：在新生代和老年代使用串行收集器
-XX:+UseParNewGC：在新生代使用并行收集器
-XX:+UseParallelGC ：新生代使用并行回收收集器，更加关注吞吐量
-XX:+UseParallelOldGC：老年代使用并行回收收集器
-XX:ParallelGCThreads：设置用于垃圾回收的线程数
-XX:+UseConcMarkSweepGC：新生代使用并行收集器，老年代使用CMS+串行收集器
-XX:ParallelCMSThreads：设定CMS的线程数量
-XX:+UseG1GC：启用G1垃圾回收器

8. JVM调优指令

JVM在内存调优方面，提供了几个常用的命令，分别为jps，jinfo，jstack，jmap以及jstat命令

jps：主要用来输出JVM中运行的进程状态信息，一般使用jps命令来查看进程的状态信息，包括JVM启动参数等。

jinfo：主要用来观察进程运行环境参数等信息。 **jinfo ${pid}**

jstack：主要用来查看某个Java进程内的线程堆栈信息。jstack pid 可以看到当前进程中各个线程的状态信息，包括其持有的锁和等待的锁。 **jstack -l ${pid}**

jmap：用来查看堆内存使用状况。jmap -heap pid可以看到当前进程的堆信息和使用的GC收集器，包括年轻代和老年代的大小分配等 **jmap -heap ${pid}**

jstat：进行实时命令行的监控，包括堆信息以及实时GC信息等。可以使用jstat -gcutil pid1000来每隔一秒来查看当前的GC信息。 **jstat -gcutil ${pid} 1000**

类加载机制

Java中的类加载机制指虚拟机**把描述类的数据从 Class 文件加载到内存**，并对数据进行**校验、 转换、解析和初始化**，最终形成可以被虚拟机直接使用的 Java 类型。

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括了：**加载、验证、准备、解析、初始化、使用、卸载七个阶段。**类加载机制的保持则包括前面五个阶段。

- **加载：**

加载是指**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。**

ss

- **验证：**

验证的作用是确保被加载的类的正确性，包括文件格式验证，元数据验证，字节码验证以及符号引用验证。

- **准备：**

准备阶段为类的静态变量分配内存，并将其初始化为默认值。假设一个类变量的定义为public static int val = 3;那么变量val在准备阶段过后的初始值不是3而是0。

- **解析：**

解析阶段将类中**符号引用转换为直接引用。**符号引用以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，只要使用时能够无歧义的定位到目标即可。

- **初始化：**

初始化阶段**为类的静态变量赋予正确的初始值**，JVM负责对类进行初始化，主要对类变量进行初始化。