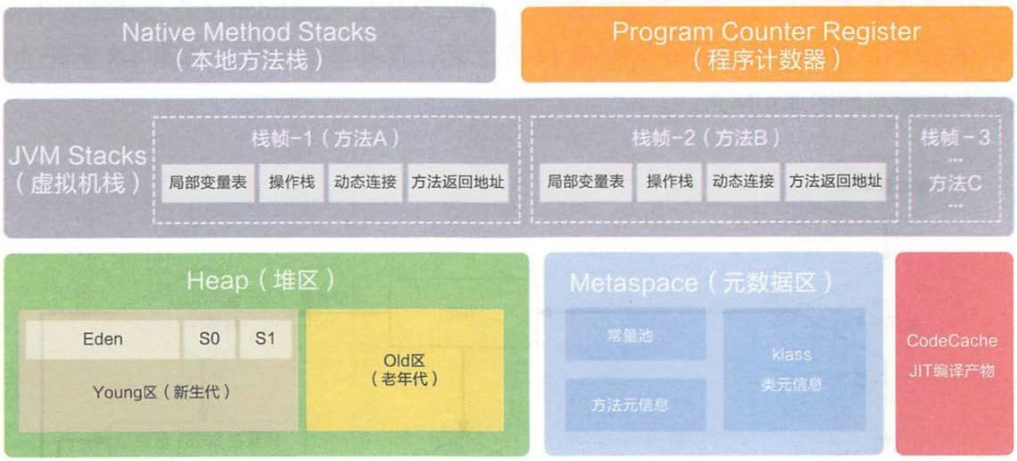
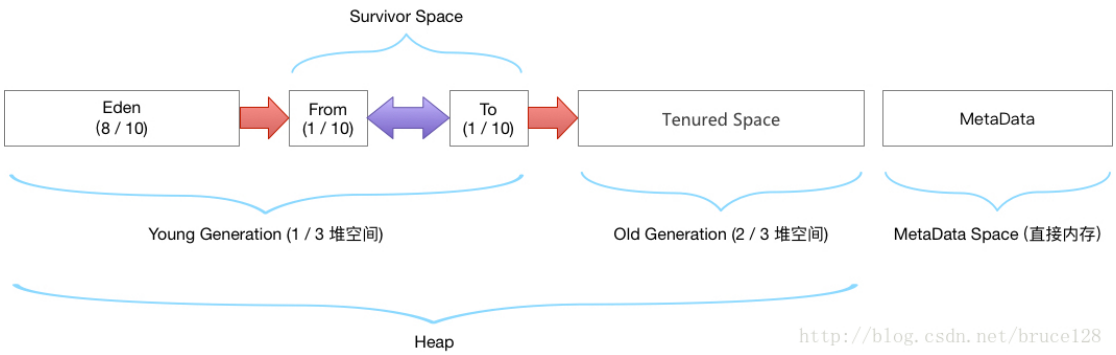
JVM 内存机制和垃圾回收机制

一、JVM 内存结构



1、堆内存（Heap）

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。堆内存是所有线程共有的，注意永久代并不属于堆内存中的一部分，同时 jdk1.8 之后永久代已经被移除。



a、新生代

新生代（Young）与老年代（Old）的比例的值为 1:2。默认的，Eden : from : to = 8 : 1 : 1 (可以通过参数 –XX:SurvivorRatio 来设定)，即： Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。

一般情况下，新创建的对象都会被分配到 Eden 区(一些大对象特殊处理),这些对象经过第一次 Minor GC 后，如果仍然存活，将会被移到 Survivor 区。对象在 Survivor 区中每熬过一次 Minor GC，年龄就会增加 1 岁，当它的年龄增加到一定程度时，就会被移动到年老代中。

因为年轻代中的对象基本都是朝生夕死的(80%以上)，所以在年轻代的垃圾回收算法使用的是复制算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

在 GC 开始的时候，对象只会存在于 Eden 区和名为“From”的 Survivor 区，Survivor 区“To”是空的。紧接着进行 GC，Eden 区中所有存活的对象都会被复制到“To”，而在“From”区中，仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过-XX:MaxTenuringThreshold 来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。经过这次 GC 后，Eden 区和 From 区已经被清空。这个时候，“From”和“To”会交换他们的角色，也就是新的“To”就是上次 GC 前的“From”，新的“From”就是上次 GC 前的“To”。不管怎样，都会保证名为 To 的 Survivor 区域是空的。Minor GC 会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。

a)、Eden 空间

因为 Eden 最大，所以新生成的对象都分配到 Eden 空间，当 Eden 空间快满时，进行一次 Minor GC，然后将存活的对象复制到 From Survivor 空间。这时，Eden 空间继续向外提供堆内存。

b)、From Survivor 空间

后面继续生成的对象还是放到 Eden 空间，当 Eden 空间又要满的时候，这时候 Eden 空间和 From Survivor 空间同时进行一次 Minor GC，然后把存活对象放到 To Survivor 空间。这时，Eden 空间继续向外提供堆内存。

c)、To Survivor 空间

b、老年代

2、**方法区（Method Area）**

《Java 虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。 同时大多数用的 JVM 都是 Sun 公司的 HotSpot。在 HotSpot 上把 GC 分代收集扩展至方法区，或者说使用永久代来实现方法区。因此，我们得到了结论，永久代是 HotSpot 的概念，方法区是 Java 虚拟机规范中的定义，是一种规范，而永久代是一种实现，一个是标准一个是实现。在 1.7 之前(JDK1.2 ~ JDK6)的实现中，HotSpot 使用永久代实现方法区。

对于 Java8， HotSpots 取消了永久代，而方法区是一个规范，规范没变，它就一直在。那么取代永久代的就是元空间（Metaspace）。永久代和元空间的区别：

a、存储位置不同，永久代物理是是堆的一部分，和新生代，老年代地址是连续的，而元空间属于本地内存；

b、存储内容不同，元空间存储类的元信息，静态变量和常量池等并入堆中。相当于永久代的数据被分到了堆和元空间中。

随着 JDK8 的到来，JVM 不再有 永久代(PermGen)。但类的元数据信息（metadata）还在，只不过不再是存储在连续的堆空间上，而是移动到叫做

“Metaspace”的本地内存（Native memory）。方法区理论上来说是堆的逻辑组成部分；

为什么用元空间取代永久代呢?

1. 字符串存在永久代中，容易出现性能问题和内存溢出。
2. 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，太小容易出现永久代溢出，太大则容易导致老年代溢出。
3. 永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

说明：运行时常量池——是方法区的一部分，是一块内存区域，用于存放编译期生成的各种字面量和符号引用；Class 文件常量池将在类加载后进入方法区的运行时常量池中存放。一个类加载到 JVM 中后对应一个运行时常量池，运行时常量池相对于 Class 文件常量池来说具备动态性，Class 文件常量只是一个静态存储结构，里面的引用都是符号引用。而运行时常量池可以在运行期间将符号引用解析为直接引用。可以说运行时常量池就是用来索引和查找字段和方法名称和描述符的。给定任意一个方法或字段的索引，通过这个索引最终可得到该方法或字段所属的类型信息和名称及描述符信息，这涉及到方法的调用和字段获取。

常量池随永久代的变化

几种常量池:

（1）静态常量池：即*.class 文件中的常量池，在 Class 文件结构中，最头的 4 个字节存储魔数，用于确定一个文件是否能被 JVM 接受，接着 4 个字节用于存储版本号，前 2 个为次版本号，后 2 个主版本号，再接着是用于存放常量的常量池，由于常量的数量是不固定的，所以常量池的入口放置一个 U2 类型的数据(constant_pool_count)存储常量池容量计数值。

这种常量池占用 class 文件绝大部分空间，主要用于存放两大类常量：字面量和符号引用量，字面量相当于 Java 语言层面常量的概念，如文本字符串、基础数据、声明为 final 的常值等；符号引用则属于编译原理方面的概念，包括了如下三种类型的常量：类和接口的全限定名、字段名称描述符、方法名称描述符。类的加载过程中的链接部分的解析步骤就是把符号引用替换为直接引用，即把那些描述符（名字）替换为能直接定位到字段、方法的引用或句柄（地址）。

（2）运行时常量池：虚拟机会将各个 class 文件中的常量池载入到运行时常量池中，即编译期间生成的字面量、符号引用，总之就是装载 class 文件。为什么它叫运行时　　常量池呢？因为这个常量池在运行时，里面的常量是可以增加的。

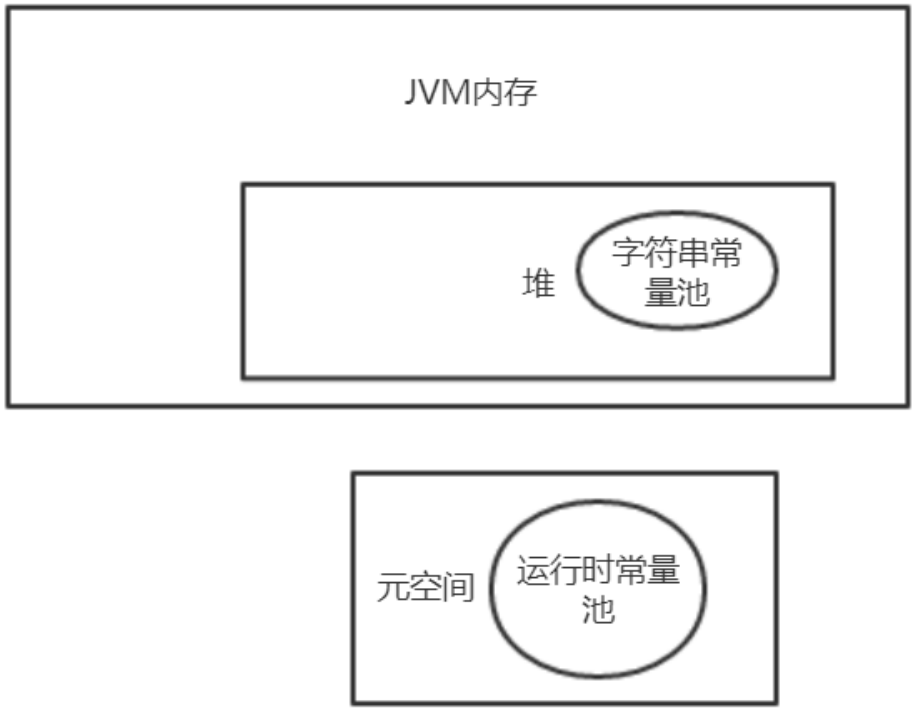
如：“+”连接字符生成新字符后调用 intern（）方法、生成基础数据的包装类型等等。

（3）字符串常量池 ： 字符串常量池可以理解为是分担了部分运行时常量池的工作。加载时，对于 class 文件的静态常量池，如果是字符串就会被装到字符串常量池中。

（4）整型常量池：Integer，类似字符串常量池。管理-128--127 的常量。类似的还有 Character、Long 等常量池（基本数据类型没有哦，Double、Float 也没有常量池）

总结就是:

class 文件有常量池存放这个类的信息，占用了大多数空间。但是运行时所有加载进来的 class 文件的常量池的东西都要放到运行时常量池，这个运行时常量池还可以在运行时添加常量。字符串常量池、Integer 等常量池则是分担了运行时常量池的工作，**在永久代移除后，字符串常量池也不再放在永久代了，但是也没有放到新的方法区---元空间里，而是留在了堆里（为了方便回收？）。运行时常量池当然是随着搬家到了元空间里，毕竟它是装类的重要等信息的，有它的地方才称得上是方法区。**

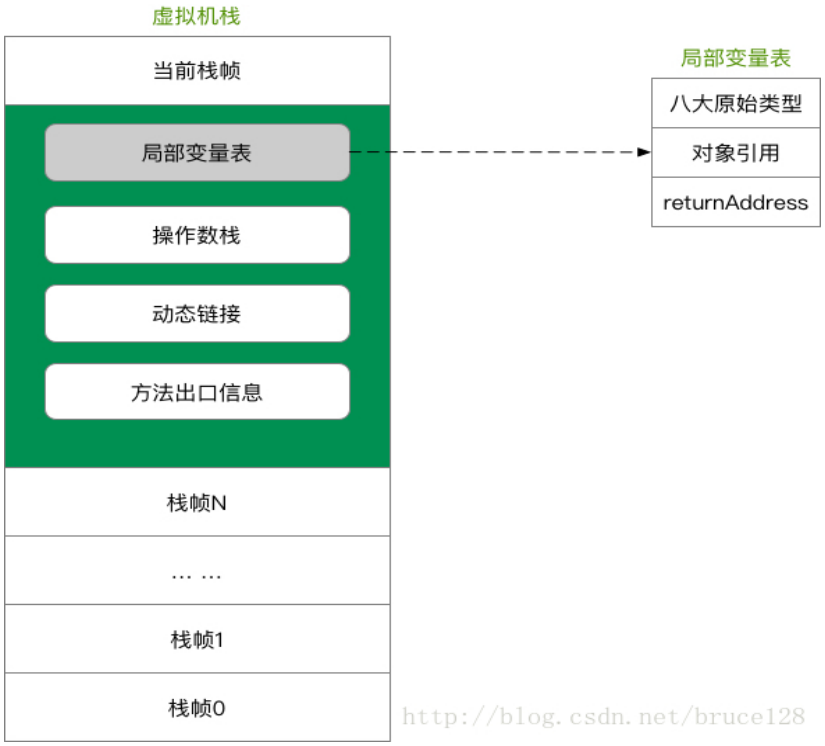


3、**栈内存**

a、虚拟机栈(JVM Stack)

描述的是 java 方法执行的内存模型：线程私有，每个线程对应一个 Java 虚拟机栈。每个方法被执行的时候都会创建一个"栈帧"并入栈,用于存储局部变量表(包括参数)、操作栈、方法出口等信息。每个方法被调用到执行完的过程，

就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。



b、本地方法栈(Native Stack)

被 native 关键字修饰的方法叫做本地方法，本地方法和其它方法不一样，本地方法意味着和平台有关，因此使用了 native 的程序可移植性都不太高。另外 native 方法在 JVM 中运行时数据区也和其它方法不一样，它有专门的本地方法栈。native 方法主要用于加载文件和动态链接库，由于 Java 语言无法访问操作系统底层信息（比如：底层硬件设备等），这时候就需要借助 C 语言来完成了。被 native 修饰的方法可以被 C 语言重写。

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。

二、Gc 性能指标

1、吞吐量

cpu 花在垃圾回收的时间和花在应用程序上的时间的占比。比如：虚拟机总共运行了 100 分钟，其中垃圾收集花掉 1 分钟，那吞吐量就是 99%。

2、 gc 负荷

与吞吐量相反，指应用花在 GC 上的时间百分比

3、 gc 频率

与吞吐量相反，指应用花在 GC 上的时间百分比

4、停顿时间

中断应用程序来做垃圾回收的应用停顿（stop-the-world）时间。

5、反应速度

从一个对象变成垃圾道这个对象被回收的时间。

总结:

一个交互式的应用要求暂停时间越少越好，然而，一个非交互性的应用，当然是希望 GC 负荷越低越好。

一个实时系统对暂停时间和 GC 负荷的要求，都是越低越好。

如果你想要比较好的**吞吐量**和**延迟**，那就得在 **CPU 消耗**上有所牺牲

如果你想要比较好的**吞吐量**和 **CPU 消耗**，那就得在**延迟**上有所牺牲

如果你想要比较好的**延迟**和 **CPU 消耗**，那就得在**吞吐量**上有所牺牲

三、gc 的算法

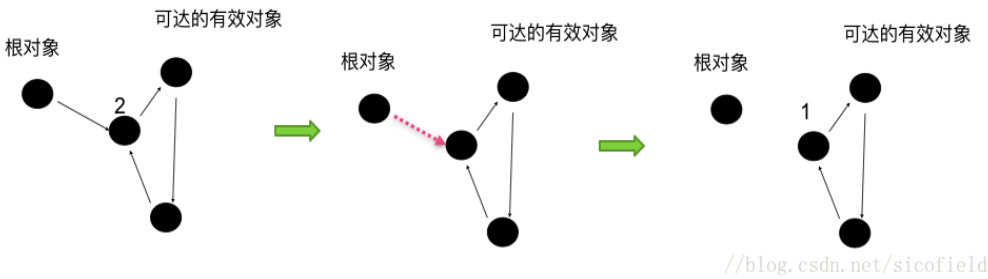
1、引用计数法

对于一个对象 A，只要有任何一个对象引用了 A，则 A 的引用计数器就加 1，当引用失效时，引用计数器就减 1。只要对象 A 的引用计数器的值为 0，则对象 A 就不可能再被使用。

缺点:

a、引用和去引用伴随加法和减法，影响性能;

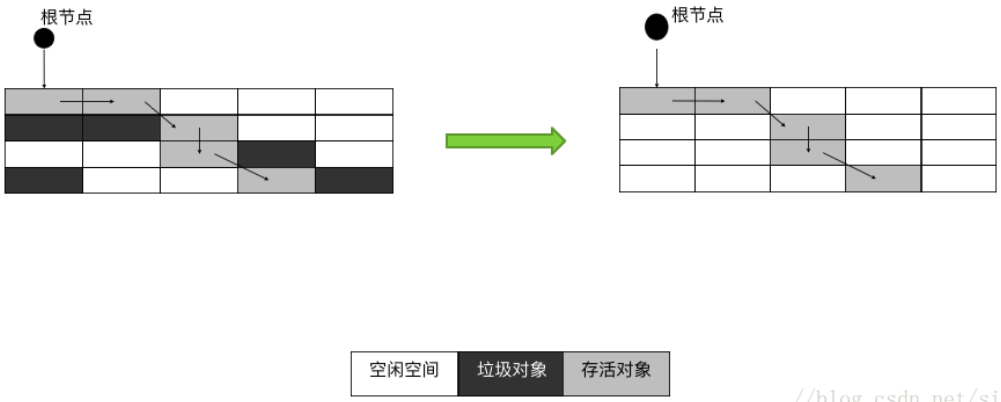
b、很难处理循环引用



如图循环引用因为引用计数不为 0，所以永远不会被 GC。

2、标记-清除

标记-清除算法是现代垃圾回收算法的思想基础。标记-清除算法将垃圾回收分为两个阶段：**标记阶段**和**清除阶段**。一种可行的实现是，在标记阶段，首先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。

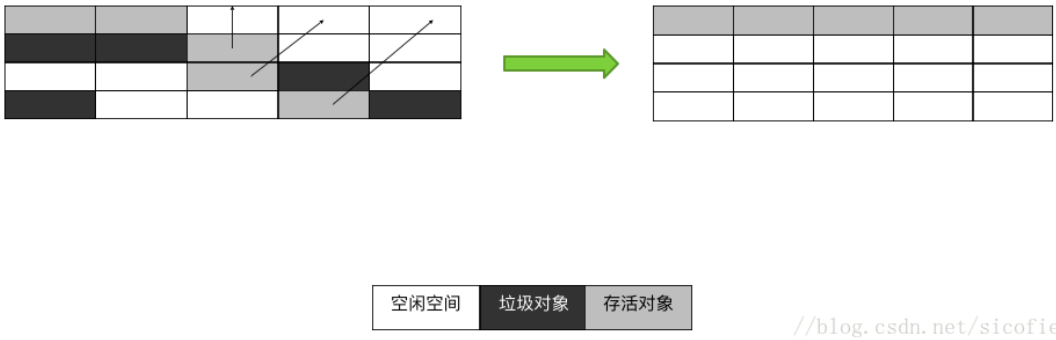


问题：

多次标记清楚过后容易形成过多的内存碎片

3、标记-压缩

标记-压缩算法适合用于存活对象较多的场合，如老年代。它在标记-清除算法的基础上做了一些优化。和标记-清除算法一样，标记-压缩算法也首先需要从根节点开始，对所有可达对象做一次标记。但之后，它并不简单的清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。



这种方式避免的内存碎片的产生。

4、复制算法

复制算法将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。

与标记-清除算法相比，复制算法是一种相对高效的回收方法，但是复制算法需要多一倍的内存。

复制算法比较适用于新生代。因为在新生代，垃圾队形通常会多于存货对象，复制算法的效果会比价好。不适用于存活对象较多的场合如老年代。

总结:

1、 分代思想

对于复制、标记清除、标记压缩等 GC 算法各自都有优势和缺陷，在所有的算法中没有一种算法能够替代其他算法。因此根据垃圾回收对象的特性，使用合适的算法回收，才是明智的选择。

分代算法就是这种思想，它将内存区间根据对象的特点分成几块，根据每块内存区间的特点，使用不同的回收算法，以提高垃圾回收的效率。

通常情况下 JVM 虚拟机的堆分为新生代和老年代。新生代的特点是对象存活时间一般不长，大约 90%的新建对象会被很快回收，因此，新生代比较适合使用复制算法。当一个对象经过几次回收之后依然存活，对象就会被放入老年代的内存空间。在老年代中，几乎所有对象都是经过几次 GC 之后依然得以存活的。因此，可以任务这些对象，在一定时期内，甚至在应用程序的整个生命周期中，将是常驻内存的。所以使用标记清除或者标记压缩算法比较合适。

2、 分区思想

分代算法按照对象的生命周期长短划分为两个部分，分区算法将整个堆空间划分成连续的不同小区间，每个小区间独立使用独立回收。这种算法的好处是可以控制一次回收多少个区间。这样可以让 GC 导致停顿时间拆短，如此 GC 多次每次时间变短。

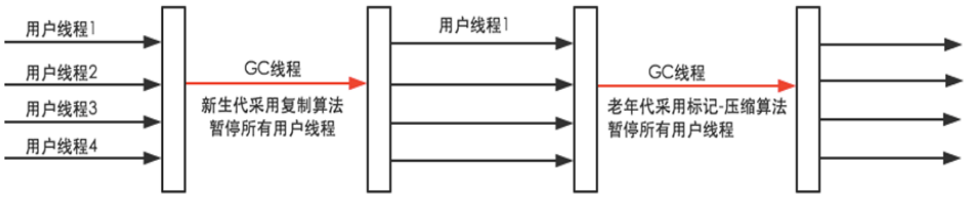
四、gc 的种类

JVM 给了三种选择：串行收集器、并行收集器、并发收集器，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，

JDK5.0 以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0 以后，JVM 会根据当前系统配置进行判断。吞吐量优先的并行收集器

1、serial 收集器（串行）

Serial 收集器指的是是串行 GC 单线程收集器，在进行垃圾回收时会暂停其他所有的工作线程（stop the world，简称 STW）直至回收结束，因此会影响用户的正常使用体验。Client 模式下，新生代默认收集器。

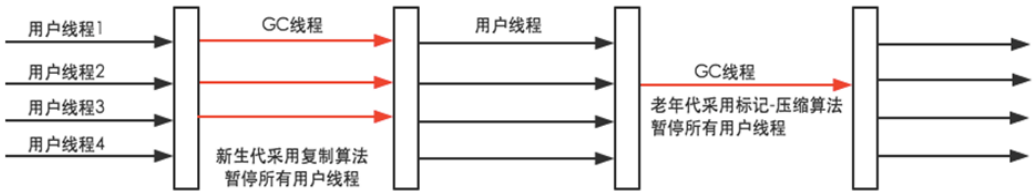


2、ParNew 收集器（并行）

并行收集器，Serial 收集器的多线程版本

Server 模式下 Jvm 默认的新生代收集器

默认开启的垃圾回收线程与 cpu 核数一致



3、CMS 收集器（并发）

并发收集器（ConcurrentMarkSweep）

采用了标记-清除算法

并发收集、低停顿

缺点:

消耗 cpu

会产生内存碎片

浮动垃圾（Concurrent Mode Failure

4、G1

a、特性

a)、并行与并发

G1 能充分利用多 CPU、多核环境下的硬件优势，使用多个 CPU 来缩短 Stop-The-World 停顿的时间，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。

b)、分代收集

与其他收集器一样，分代概念在 G1 中依然得以保留。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。

c)、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（两个 Region 之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着 G1 运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次 GC。

d)、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，消耗在垃圾收集上的时间不得超过 N 毫秒。

在 G1 之前的其他收集器进行收集的范围都是整个新生代或者老年代，而 G1 不再是这样。使用 G1 收集器时，Java 堆的内存布局就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离的了，它们

都是一部分 Region（不需要连续）的集合。

G1 收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个 Java 堆中进行全区域的垃圾收集。G1 跟踪各个 Region 里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region（这也就是 Garbage-First 名称的来由）。这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 G1 收集器在有限的时间内可以获取尽可能高的收集效率。

b、执行过程

G1 收集器的运作大致可划分为以下几个步骤：

a)、初始标记（Initial Marking）

初始标记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 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，这阶段需要停顿线程，但耗时很短。

b)、并发标记（Concurrent Marking）

并发标记阶段是从 GC Root 开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行。

c)、最终标记（Final Marking）

最终标记阶段是为了修正在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标记记录，虚拟机将这段时间对象变化记录在线程 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中，这阶段需要停顿线程，但是可并行执行。

d)、筛选回收（Live Data Counting and Evacuation）

筛选回收阶段首先对各个 Region 的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划，这个阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅提高收集效率。

c、的

d、的

concurrent 又叫响应时间优先的收集器，它是说 GC 线程可以和应用线程并行执行。CMS 和 G1 都是 concurrent 的垃圾收集器，JDK9 里面 G1 已经是默认的垃圾收集器了。

G1 收集器运行示意图

5、parallel

parallel 又叫吞吐量优先的收集器，指多个 GC 线程并行，但是 GC 线程与应用线程不是并行，用户线程仍然处于等待状态。

在早期 JDK 8 等版本中，它是 server 模式 JVM 的默认 GC 选择，也被称作是吞吐量优先的 GC。它的算法和 Serial GC 比较相似，尽管实现要复杂的多，其特点是新生代和老年代 GC 都是并行进行的，在常见的服务器环境中更加高效。

开启选项是：

-XX:+UseParallelGC

另外，Parallel GC 引入了开发者友好的配置项，我们可以直接设置暂停时间或吞吐量等目标，JVM 会自动进行适应性调整，例如下面参数：

-XX:MaxGCPauseMillis=value -XX:GCTimeRatio=N // GC 时间和用户时间比例 = 1 / (N+1)

注：每一种 GC 的日志格式都是不一样的。

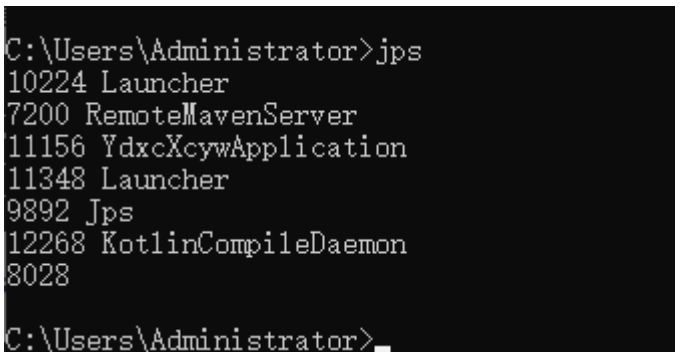
五、JVM 参数

`-Xms`:初始堆大小,默认为物理内存的 `1/64(<1GB)`;默认(`MinHeapFreeRatio` 参数可以调整)空余堆内存小于 `40%`时，`JVM` 就会增大堆直到`-Xmx` 的最大限制

六、Jdk 常用命令

1、jps

jps:与 unix 上的 ps 类似，用来显示本地的 java 进程，可以查看本地运行着几个 java 程序（因为每一个 java 程序都会独占一个 java 虚拟机实例），并显示他们的进程号。不过 jps 有个缺点是只能显示当前用户的进程 id，要显示其他用户的还只能用 linux 的 ps 命令。



执行 jps 命令，会列出所有正在运行的 java 进程，其中 jps 命令也是一个 java 程序，前面的数字就是对应的进程 id，这个 id 的作用非常大，后面会有相关介绍。

jps -help：

```
usage: jps [-help]
       jps [-q] [-mlvV] [<hostid>]

Definitions:
    <hostid>:      <hostname>[:<port>]
```

jps -l 输出应用程序 main.class 的完整 package 名或者应用程序 jar 文件完整路径名

```
C:\Users\Administrator>jps -l
Picked up _JAVA_OPTIONS: -Xmx512M
4384 C:\Users\ADMINI~1\AppData\Local\Temp\pulDA28.tmp\PULSEI~1.JAR
6888 sun.tools.jps.Jps
3704 JpsDemo
```

jps -v 输出传递给 JVM 的参数

```
C:\Users\Administrator>jps -v
Picked up _JAVA_OPTIONS: -Xmx512M
4384 PULSEI~1.JAR -Xmx512m -XX:MaxPermSize=512m -XX:ReservedCodeCacheSize=64m -D
osgi.nls.warnings=ignore -Xmx512M
3704 JpsDemo -Xms64m -Xmx2048m -Dfile.encoding=GBK -Xmx512M
3320 Jps -Denv.class.path=.;C:\Program Files (x86)\Java\jdk1.6.0_12\jre\lib\rt.j
ar;C:\Program Files (x86)\Java\jdk1.6.0_12\lib\tools.jar;C:\Program Files (x86)\
Java\jdk1.6.0_12\lib\dt.jar -Dapplication.home=C:\Program Files (x86)\Java\jdk1.
6.0_12 -Xms8m -Xmx512M
```

jps 失效

我们在定位问题过程会遇到这样一种情况，用 jps 查看不到进程 id，用 ps -ef | grep java 却能看到启动的 java 进程。

要解释这种现象，先来了解下 jps 的实现机制：

java 程序启动后，会在目录/tmp/hsperfdata_{userName}/下生成几个文件，文件名就是 java 进程的 pid，因此 jps 列出进程 id 就是把这个目录下的文件名列一下而已，至于系统参数，则是读取文件中的内容。

我们来思考下：**如果由于磁盘满了，无法创建这些文件，或者用户对这些文件没有读的权限。又或者因为某种原因这些文件或者目录被清除，出现以上这些情况，就会导致 jps 命令失效。**

如果 jps 命令失效，而我们又要获取 pid，还可以使用以下两种方法：

1、top | grep java

2、ps -ef |grep java

2、jstack

主要用于生成指定进程当前时刻的线程快照，线程快照是当前 java 虚拟机每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是用于定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致长时间等待。

```
C:\Users\Administrator>jstack 6220
Picked up _JAVA_OPTIONS: -Xmx512M
2016-09-28 17:31:42
Full thread dump Java HotSpot(TM) Client VM (11.2-b01 mixed mode, sharing):

"Low Memory Detector" daemon prio=6 tid=0x024a6800 nid=0x1660 runnable [0x000000
00..0x00000000]
   java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x0249bc00 nid=0x14e4 waiting on condition
[0x00000000..0x04b1f914]
   java.lang.Thread.State: RUNNABLE

"Attach Listener" daemon prio=10 tid=0x024a6400 nid=0x1394 waiting on condition
[0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=10 tid=0x024a5c00 nid=0x12ac runnable [0x0000000
0..0x00000000]
   java.lang.Thread.State: RUNNABLE

"Finalizer" daemon prio=8 tid=0x02453000 nid=0xa5c in Object.wait() [0x0496f000.
.0x0496fa68]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x0a8802a0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
        - locked <0x0a8802a0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=10 tid=0x02451c00 nid=0x1a50 in Object.wait() [0
x048df000..0x048dfae8]
```

3、jmap

主要用于打印指定 java 进程的共享对象内存映射或堆内存细节。

堆 Dump 是反映堆使用情况的内存镜像，其中主要包括系统信息、虚拟机属性、完整的线程 Dump、所有类和对象的状态等。一般在内存不足，GC 异常等情况下，我们会去怀疑内存泄漏，这个时候就会去打印堆 Dump。

4、jstat

a、jstat -gc pid

垃圾回收统计。

```
C:\Users\Administrator>jstat -gc 11156
 SOC    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC    YGCT    FGC    FGCT     GCT
2048.0 24576.0  0.0    0.0   1019392.0 48530.4   216576.0   47770.8   73424.0 69756.5 9728.0 9046.5     14    0.162   4      0.374    0.536
```

- S0C：第一个幸存区的大小

- S1C：第二个幸存区的大小

- S0U：第一个幸存区的使用大小

- S1U：第二个幸存区的使用大小
- EC：伊甸园区的大小
- EU：伊甸园区的使用大小
- OC：老年代大小
- OU：老年代使用大小
- MC：方法区大小
- MU：方法区使用大小
- CCSC:压缩类空间大小
- CCSU:压缩类空间使用大小
- YGC：年轻代垃圾回收次数
- YGCT：年轻代垃圾回收消耗时间
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

b、jstat -gcutil pid

总结垃圾回收统计

```
C:\Users\Administrator>jstat -gcutil 4112
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00   8.39  24.31  93.81  91.17     19    0.186     7    1.423    1.609
```

- S0：幸存 1 区当前使用比例
- S1：幸存 2 区当前使用比例
- E：伊甸园区使用比例
- O：老年代使用比例
- M：元数据区使用比例
- CCS：压缩使用比例
- YGC：年轻代垃圾回收次数
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

c、jstat -gcnew pid

新生代垃圾回收统计

```
C:\Users\Administrator>jstat -gcnew 4112
 S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
25088.0 25088.0    0.0    0.0  1  15 25088.0 965632.0  80977.6     19    0.186
```

- S0C：第一个幸存区大小
- S1C：第二个幸存区的大小
- S0U：第一个幸存区的使用大小
- S1U：第二个幸存区的使用大小
- TT:对象在新生代存活的次数
- MTT:对象在新生代存活的最大次数
- DSS:期望的幸存区大小
- EC：伊甸园区的大小
- EU：伊甸园区的使用大小
- YGC：年轻代垃圾回收次数
- YGCT：年轻代垃圾回收消耗时间

d、jstat -gccapacity pid

堆内存统计

```
C:\Users\Administrator>jstat -gccapacity 4112
 NGCMN    NGCMX     NGC     S0C   S1C       EC      OGCMN      OGCMX       OGC         OC       MCMN     MCMX      MC     CCSMN    CCSMX     CCSC    YGC    FGC
 87040.0 1390592.0 1015808.0 25088.0 25088.0 965632.0   175104.0  2781184.0   196608.0   196608.0      0.0 1120256.0  82328.0      0.0 1048576.0  10920.0    19     7
```

- NGCMN：新生代最小容量
- NGCMX：新生代最大容量
- NGC：当前新生代容量
- S0C：第一个幸存区大小
- S1C：第二个幸存区的大小
- EC：伊甸园区的大小
- OGCMN：老年代最小容量
- OGCMX：老年代最大容量
- OGC：当前老年代大小
- OC:当前老年代大小
- MCMN:最小元数据容量
- MCMX：最大元数据容量

- MC：当前元数据空间大小
- CCSMN：最小压缩类空间大小
- CCSMX：最大压缩类空间大小
- CCSC：当前压缩类空间大小
- YGC：年轻代 gc 次数
- FGC：老年代 GC 次数

e、jstat -gcmetacapacity pid

元数据空间统计

```
C:\Users\Administrator>jstat -gcmetacapacity  4112
   MCMN       MCMX        MC       CCSMN      CCSMX       CCSC     YGC   FGC    FGCT     GCT
      0.0  1120256.0    82328.0        0.0  1048576.0    10920.0    19     7    1.423    1.609
```

- MCMN:最小元数据容量
- MCMX：最大元数据容量
- MC：当前元数据空间大小
- CCSMN：最小压缩类空间大小
- CCSMX：最大压缩类空间大小
- CCSC：当前压缩类空间大小
- YGC：年轻代垃圾回收次数
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

f、jstat -gcnewcapacity pid

新生代内存空间统计

```
C:\Users\Administrator>jstat -gcnewcapacity 4112
  NGCMN       NGCMX        NGC       S0CMX     S0C     S1CMX     S1C       ECMX        EC      YGC   FGC
   87040.0  1390592.0  1015808.0 463360.0  25088.0 463360.0  25088.0  1389568.0   965632.0    19     7
```

- NGCMN：新生代最小容量
- NGCMX：新生代最大容量
- NGC：当前新生代容量
- S0CMX：最大幸存 1 区大小
- S0C：当前幸存 1 区大小

- S1CMX：最大幸存 2 区大小
- S1C：当前幸存 2 区大小
- ECMX：最大伊甸园区大小
- EC：当前伊甸园区大小
- YGC：年轻代垃圾回收次数
- FGC：老年代回收次数

g、jstat -gcoldcapacity pid

老年代内存空间统计

```
C:\Users\Administrator>jstat -gcoldcapacity 4112
   OGCMN       OGCMX        OGC         OC       YGC   FGC    FGCT     GCT
   175104.0   2781184.0    196608.0    196608.0    19     7    1.423    1.609
```

- NGCMN：新生代最小容量
- NGCMX：新生代最大容量
- NGC：当前新生代容量
- S0CMX：最大幸存 1 区大小
- S0C：当前幸存 1 区大小
- S1CMX：最大幸存 2 区大小
- S1C：当前幸存 2 区大小
- ECMX：最大伊甸园区大小
- EC：当前伊甸园区大小
- YGC：年轻代垃圾回收次数
- FGC：老年代回收次数