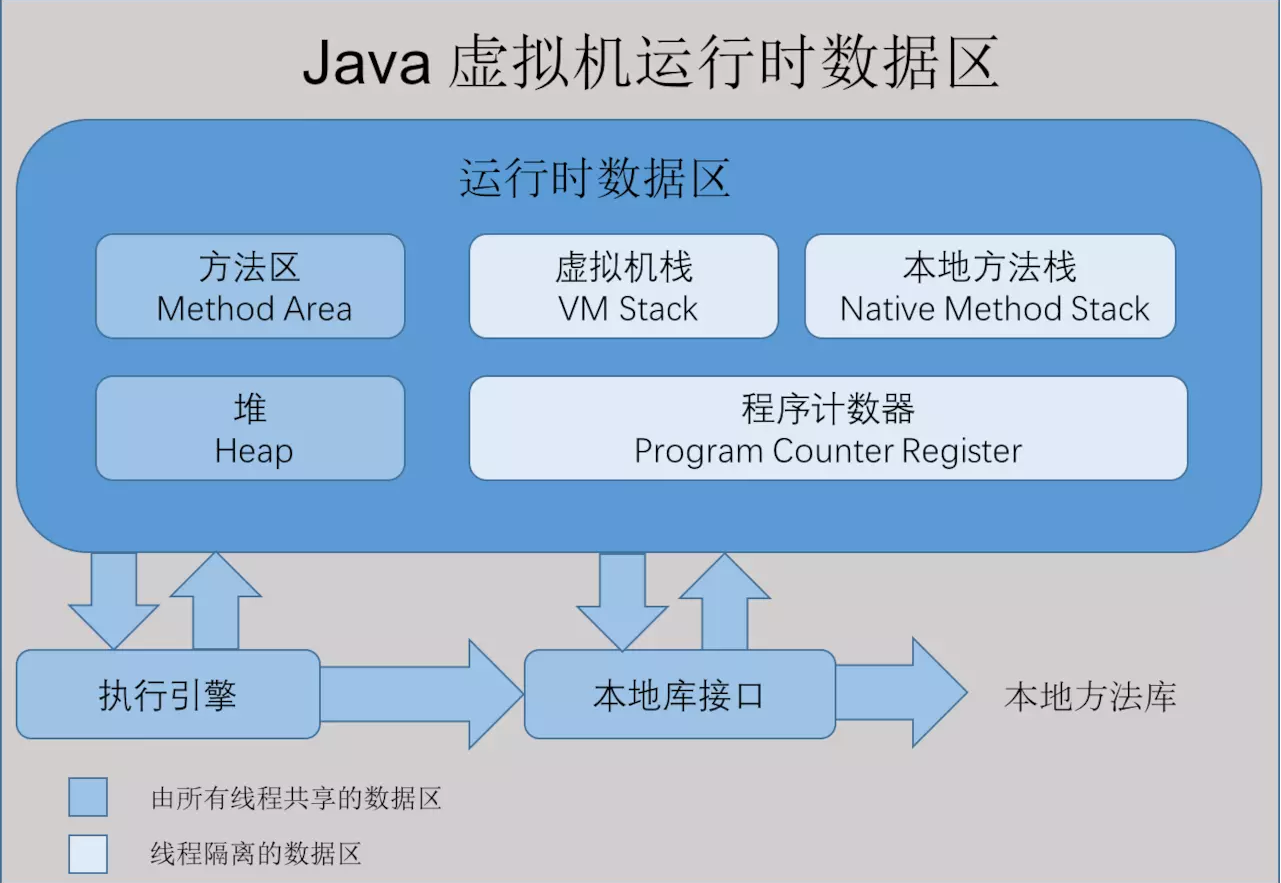
**JVM**

JVM是Java Virtual Machine（Java虚拟机）的缩写。

**一、JVM空间划分**



**（一）栈区**

**栈分为java虚拟机栈和本地方法栈**

1 、虚拟机栈：线程私有的，生命周期与线程相同。每个方法执行都会创建一个栈帧，用于存放局部变量表，操作栈，动态链接，方法出口等。每个方法从被调用，直到被执行完。对应着一个栈帧在虚拟机中从入栈到出栈的过程。通常说的栈就是指局部变量表部分，存放编译期间可知的8种基本数据类型，及对象引用和指令地址。局部变量表是在编译期间完成分配，当进入一个方法时，这个栈中的局部变量分配内存大小是确定的。

2、本地方法栈：本地方法栈为虚拟机使用到本地方法服务（native）

注：会有两种异常StackOverFlowError和 OutOfMemoneyError。当线程请求栈深度大于虚拟机所允许的深度就会抛出StackOverFlowError错误；虚拟机栈动态扩展，当扩展无法申请到足够的内存空间时候，抛出OutOfMemoneyError。

**（二）堆区**

所有线程共享区域，在虚拟机启动时创建，唯一目的存放对象实例；

堆区是gc的主要区域，通常情况下分为两个区块年轻代和年老代。更细一点年轻代又分为Eden区主要放新创建对象，From survivor 和 To survivor 保存gc后幸存下的对象，默认情况下各自占比8:1:1；

注：会有异常OutOfMemoneyError；

**（三）方法区**

被所有线程共享区域，用于存放已被虚拟机加载的类信息，常量，静态变量等数据。被Java虚拟机描述为堆的一个逻辑部分。习惯是也叫它永久代；

垃圾回收很少光顾这个区域，不过也是需要回收的，主要针对常量池回收，类型卸载；

常量池用于存放编译期生成的各种字节码和符号引用，常量池具有一定的动态性，里面可以存放编译期生成的常量；运行期间的常量也可以添加进入常量池中，比如string的intern()方法。

注：基于Hotspot虚拟机划分，把方法区算着为永久代来进行垃圾回收。但很难确定永久代的大小，因为它受到很多因素影响，并且每次 Full GC 之后永久代的大小都会改变，所以经常会抛出 OutOfMemoryError 异常。为了更容易管理方法区，从 JDK 1.8 开始，移除永久代，并把方法区移至元空间，它位于本地内存中，而不是虚拟机内存中。；

**（四）运行时常量池**

属于方法区一部分，用于存放编译期生成的各种字面量和符号引用。编译器和运行期(String 的 intern() )都可以将常量放入池中。内存有限，无法申请时抛出 OutOfMemoryError。

**（五）程序计数器**

当前线程所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成。

Java虚拟机多线程是通过线程轮流切换并分配处理器执行时间的方式实现的。

为了线程切换能恢复到正确的位置，每条线程都需要一个独立的程序计数器，所以它是线程私有的。

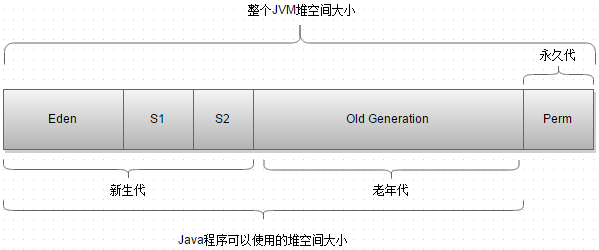
注：唯一一块Java虚拟机没有规定任何OutofMemoryError的区块。

**（六）直接内存**

非虚拟机运行时数据区的部分，在 JDK 1.4 中新加入 NIO (New Input/Output) 类，引入了一种基于通道(Channel)和缓存(Buffer)的 I/O 方式，它可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。可以避免在 Java 堆和 Native 堆中来回的数据耗时操作。

注：会有异常OutOfMemoryError：会受到本机内存限制，如果内存区域总和大于物理内存限制从而导致动态扩展时出现该异常。

**二、堆的分区**



1.JVM中堆空间可以分成三个大区，新生代、老年代、永久代（即方法区）。

2.新生代可以划分为三个区，Eden区，两个幸存区。即Eden，S1(From Survival)，S2(To Survival)

在JVM运行时，可以通过配置以下参数改变整个JVM堆的配置比例

JVM运行时堆的大小

-Xms堆的最小值

-Xmx堆空间的最大值

.新生代堆空间大小调整

-XX:NewSize新生代的最小值

-XX:MaxNewSize新生代的最大值

-XX:NewRatio设置新生代与老年代在堆空间的大小

-XX:SurvivorRatio新生代中Eden所占区域的大小

3.永久代大小调整

-XX:MaxPermSize

**（一）新生代**

Eden 区

IBM 公司的专业研究表明，有将近 98% 的对象是朝生夕死，所以针对这一现状，大多数情况下，对象会在新生代 Eden 区中进行分配。

当 Eden 区没有足够空间进行分配时，虚拟机会发起一次 Minor GC，Minor GC 相比 Major GC 更频繁，回收速度也更快。

通过 Minor GC 之后，Eden 会被清空，Eden 区中绝大部分对象会被回收，而那些无需回收的存活对象，将会进到 Survivor 的 From 区(若 From 区不够，则直接进入 Old 区)。

Survivor 区

Survivor 区相当于是 Eden 区和 Old 区的一个缓冲，类似于我们交通灯中的黄灯。

Survivor 又分为 2 个区：

From 区

To 区

每次执行 Minor GC，会将 Eden 区和 From 存活的对象放到 Survivor 的 To 区(如果 To 区不够，则直接进入 Old 区)。

①为啥需要?

不就是新生代到老年代么，直接 Eden 到 Old 不好了吗，为啥要这么复杂。

想想如果没有 Survivor 区，Eden 区每进行一次 Minor GC，存活的对象就会被送到老年代，老年代很快就会被填满。

而有很多对象虽然一次 Minor GC 没有消灭，但其实也并不会蹦跶多久，或许第二次，第三次就需要被清除。这时候移入老年区，很明显不是一个明智的决定。

所以，Survivor 的存在意义就是减少被送到老年代的对象，进而减少 Major GC 的发生。

Survivor 的预筛选保证，只有经历 16 次 Minor GC 还能在新生代中存活的对象，才会被送到老年代。

②为啥需要俩?

设置两个 Survivor 区最大的好处就是解决内存碎片化。我们先假设一下，Survivor 如果只有一个区域会怎样。

Minor GC 执行后，Eden 区被清空了，存活的对象放到了 Survivor 区，而之前 Survivor 区中的对象，可能也有一些是需要被清除的。

问题来了，这时候我们怎么清除它们?在这种场景下，我们只能标记清除，而我们知道标记清除最大的问题就是内存碎片，在新生代这种经常会消亡的区域，采用标记清除必然会让内存产生严重的碎片化。

因为 Survivor 有2个区域，所以每次 Minor GC，会将之前 Eden 区和 From 区中的存活对象复制到 To 区域。

第二次 Minor GC 时，From 与 To 职责兑换，这时候会将 Eden 区和 To 区中的存活对象再复制到 From 区域，以此反复。

这种机制最大的好处就是，整个过程中，永远有一个 Survivor space 是空的，另一个非空的 Survivor space 是无碎片的。

那么，Survivor 为什么不分更多块呢?比方说分成三个、四个、五个?显然，如果 Survivor 区再细分下去，每一块的空间就会比较小，容易导致 Survivor 区满，两块 Survivor 区可能是经过权衡之后的最佳方案。

**（二）老年代**

老年代占据着 2/3 的堆内存空间，只有在 Major GC 的时候才会进行清理，每次 GC 都会触发“Stop-The-World”。

内存越大，STW 的时间也越长，所以内存也不仅仅是越大就越好。由于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，所以老年代这里采用的是标记-整理算法。

除了上述所说，在内存担保机制下，无法安置的对象会直接进到老年代，以下几种情况也会进入老年代。

①大对象

大对象指需要大量连续内存空间的对象，这部分对象不管是不是“朝生夕死”，都会直接进到老年代。

这样做主要是为了避免在 Eden 区及 2 个 Survivor 区之间发生大量的内存复制。当你的系统有非常多“朝生夕死”的大对象时，得注意了。

②长期存活对象

虚拟机给每个对象定义了一个对象年龄(Age)计数器。正常情况下对象会不断的在 Survivor 的 From 区与 To 区之间移动，对象在 Survivor 区中每经历一次 Minor GC，年龄就增加1岁。

当年龄增加到 15 岁时，这时候就会被转移到老年代。当然，这里的 15，JVM 也支持进行特殊设置。

③动态对象年龄

虚拟机并不重视要求对象年龄必须到 15 岁，才会放入老年区，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去老年区，无需等你“成年”。

这其实有点类似于负载均衡，轮询是负载均衡的一种，保证每台机器都分得同样的请求。

看似很均衡，但每台机的硬件不同，健康状况不同，我们还可以基于每台机接受的请求数，或每台机的响应时间等，来调整我们的负载均衡算法。、

**三、垃圾对象判定标准**

**（一）引用计数法**

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器就加1；当引用失效的时候，计数器就减1；任何时刻计数器为0的对象就是不可能再被使用的。

引用计数算法是将垃圾回收分摊到整个应用程序的运行当中了，而不是在进行垃圾收集时，不需要挂起整个应用的运行，直到对堆中所有对象的处理都结束。因此，采用引用计数的垃圾收集不属于严格意义上的"Stop-The-World"的垃圾收集机制。

客观说：引用计数器实现简单，判定效率也足够高，在部分情况下是一个不错的算法。但JVM并没有使用引用计数法来管理内存。问题在于若对象之间存在相互引用，就无法回收。

**（二）可达性分析算法**

可达性分析算法(Reachability Analysis)的基本思路是，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 GC Roots 节点到该节点不可达)，则证明该对象是不可用的。

通过可达性算法，成功解决了引用计数所无法解决的问题-“循环依赖”，只要你无法与 GC Root 建立直接或间接的连接，系统就会判定你为可回收对象。那这样就引申出了另一个问题，哪些属于 GC Root。

在 Java 语言中，可作为 GC Root 的对象包括以下 4 种：

①虚拟机栈(栈帧中的本地变量表)中引用的对象**:**

此时的 s，即为 GC Root，当s置空时，localParameter 对象也断掉了与 GC Root 的引用链，将被回收

②方法区中类静态属性引用的对象:

s 为 GC Root，s 置为 null，经过 GC 后，s 所指向的 properties 对象由于无法与 GC Root 建立关系被回收。

而 m 作为类的静态属性，也属于 GC Root，parameter 对象依然与 GC root 建立着连接，所以此时 parameter 对象并不会被回收。

③方法区中常量引用的对象:

m 即为方法区中的常量引用，也为 GC Root，s 置为 null 后，final 对象也不会因没有与 GC Root 建立联系而被回收。

④本地方法栈中JNI(即一般说的 Native 方法)引用的对象:

任何 Native 接口都会使用某种本地方法栈，实现的本地方法接口是使用 C 连接模型的话，那么它的本地方法栈就是 C 栈。当线程调用 Java 方法时，虚拟机会创建一个新的栈帧并压入 Java 栈。

然而当它调用的是本地方法时，虚拟机会保持 Java 栈不变，不再在线程的 Java 栈中压入新的帧，虚拟机只是简单地动态连接并直接调用指定的本地方法。

**四、垃圾回收算法**

**（一）**复制算法：

把空间分成两块，每次只对其中一块进行 GC。当这块内存使用完时，就将还存活的对象复制到另一块上面。

缺点：空间利用率低下

注意：大多数新生代对象都不会熬过第一次GC。没必要 1 : 1 划分空间。可以分一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 空间和其中一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活的对象一次性复制到另一块 Survivor 上，最后清理 Eden 和 Survivor 空间。大小比例一般是 8 : 1 : 1，每次浪费 10% 的 Survivor 空间。但是如果存活的大于 10%，采用一种分配担保策略将多出来的对象直接进入老年代。

**（**二）标记-清除算法

直接标记清除就可。

缺点：①效率不高 ②空间会产生大量碎片③应用需要挂起，也就是stop the world

（三）标记-整理算法

不同于针对新生代的复制算法，针对老年代的特点，创建该算法。主要是把存活对象移到内存的一端。

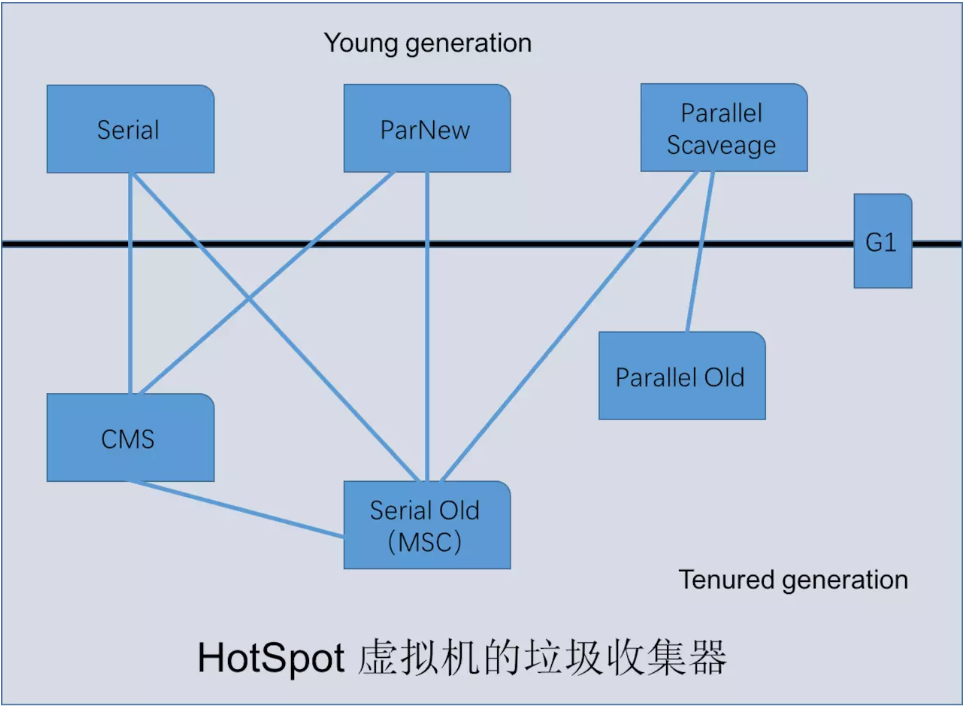
**（四）分代收集算法**

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期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）和新生代（Young Generation），老年代的特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要被回收，那么就可以根据不同代的特点采取最适合的收集算法。

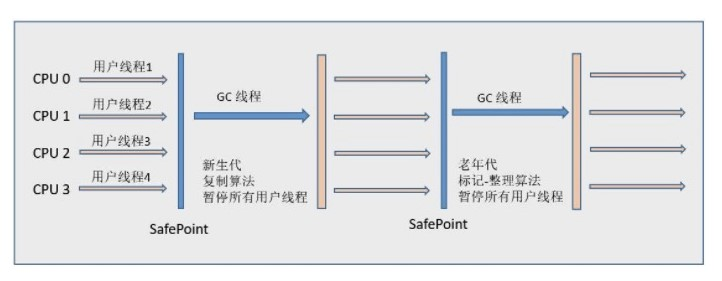
目前大部分垃圾收集器对于新生代都采取复制算法，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，但是实际中并不是按照1：1的比例来划分新生代的空间的，一般来说是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将Eden和Survivor中还存活的对象复制到另一块Survivor空间中，然后清理掉Eden和刚才使用过的Survivor空间。

而由于老年代的特点是每次回收都只回收少量对象，一般使用的是标记-整理算法（压缩法）。

**五、垃圾收集器**



1. Serial 收集器



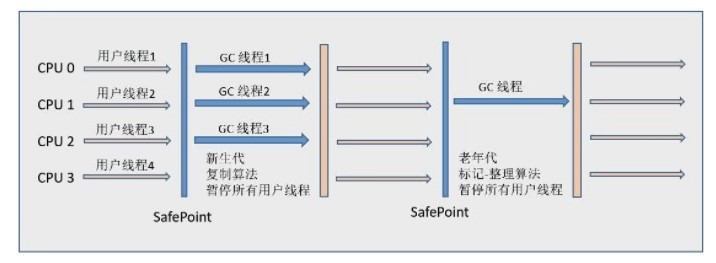
Serial 翻译为串行，也就是说它以串行的方式执行。

它是单线程的收集器，只会使用一个线程进行垃圾收集工作。

它的优点是简单高效，在单个 CPU 环境下，由于没有线程交互的开销，因此拥有最高的单线程收集效率。

它是 Client 场景下的默认新生代收集器，因为在该场景下内存一般来说不会很大。它收集一两百兆垃圾的停顿时间可以控制在一百多毫秒以内，只要不是太频繁，这点停顿时间是可以接受的。

1. ParNew 收集器



它是 Serial 收集器的多线程版本。

它是 Server 场景下默认的新生代收集器，除了性能原因外，主要是因为除了 Serial 收集器，只有它能与 CMS 收集器配合使用。

3. Parallel Scavenge 收集器

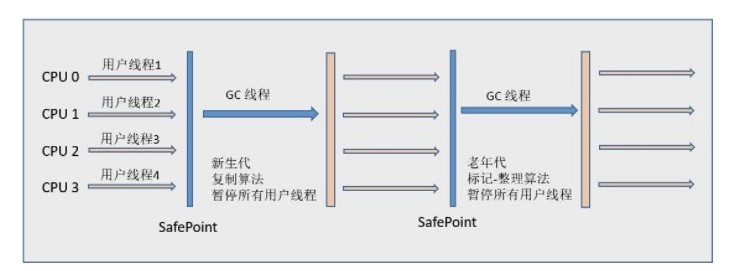
与 ParNew 一样是多线程收集器。

其它收集器目标是尽可能缩短垃圾收集时用户线程的停顿时间，而它的目标是达到一个可控制的吞吐量，因此它被称为“吞吐量优先”收集器。这里的吞吐量指 CPU 用于运行用户程序的时间占总时间的比值。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验。而高吞吐量则可以高效率地利用CPU 时间，尽快完成程序的运算任务，适合在后台运算而不需要太多交互的任务。

缩短停顿时间是以牺牲吞吐量和新生代空间来换取的：新生代空间变小，垃圾回收变得频繁，导致吞吐量下降。可以通过一个开关参数打开 GC 自适应的调节策略（GC Ergonomics），就不需要手工指定新生代的大小（-Xmn）、Eden 和 Survivor 区的比例、晋升老年代对象年龄等细节参数了。虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。

4.Serial Old 收集器

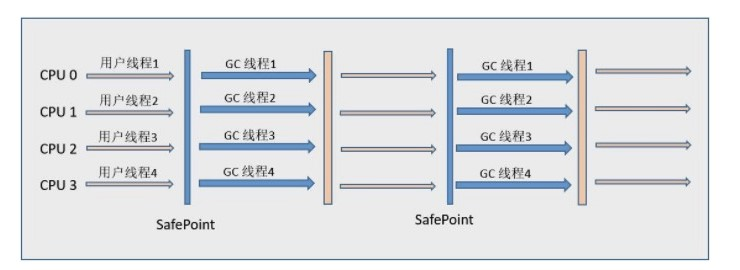


是 Serial 收集器的老年代版本，也是给 Client 场景下的虚拟机使用。如果用在 Server 场景下，它有两大用途：

在 JDK 1.5 以及之前版本（Parallel Old 诞生以前）中与 Parallel Scavenge 收集器搭配使用。

作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 时使用。

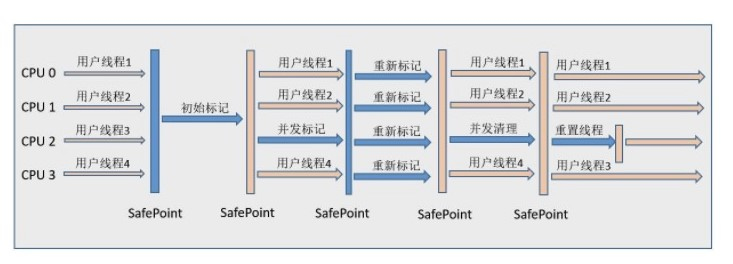
5. Parallel Old 收集器



是 Parallel Scavenge 收集器的老年代版本。

在注重吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

6. CMS 收集器



CMS（Concurrent Mark Sweep），Mark Sweep 指的是标记 - 清除算法。

分为以下四个流程：

初始标记：仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿。

并发标记：进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，不需要停顿。

重新标记：为了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，需要停顿。

并发清除：不需要停顿。

在整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，不需要进行停顿。

具有以下缺点：

吞吐量低：低停顿时间是以牺牲吞吐量为代价的，导致 CPU 利用率不够高。

无法处理浮动垃圾，可能出现 Concurrent Mode Failure。浮动垃圾是指并发清除阶段由于用户线程继续运行而产生的垃圾，这部分垃圾只能到下一次 GC 时才能进行回收。由于浮动垃圾的存在，因此需要预留出一部分内存，意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。

标记 - 清除算法导致的空间碎片，往往出现老年代空间剩余，但无法找到足够大连续空间来分配当前对象，不得不提前触发一次 Full GC。

7. G1 收集器

G1（Garbage-First），它是一款面向服务端应用的垃圾收集器，在多 CPU 和大内存的场景下有很好的性能。

HotSpot 开发团队赋予它的使命是未来可以替换掉 CMS 收集器。

堆被分为新生代和老年代，其它收集器进行收集的范围都是整个新生代或者老年代，而 G1 可以直接对新生代和老年代一起回收。

G1 把堆划分成多个大小相等的独立区域（Region），新生代和老年代不再物理隔离。

通过引入 Region 的概念，从而将原来的一整块内存空间划分成多个的小空间，使得每个小空间可以单独进行垃圾回收。这种划分方法带来了很大的灵活性，使得可预测的停顿时间模型成为可能。通过记录每个 Region 垃圾回收时间以及回收所获得的空间（这两个值是通过过去回收的经验获得），并维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。

每个 Region 都有一个 Remembered Set，用来记录该 Region 对象的引用对象所在的 Region。通过使用Remembered Set，在做可达性分析的时候就可以避免全堆扫描。

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：

**初始标记**

**并发标记**

**最终标记：**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的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。

**筛选回收：**首先对各个 Region 中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。此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度提高收集效率。

具备如下特点：

**空间整合**：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，这意味着运行期间不会产生内存空间碎片。

可预测的停顿：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 GC 上的时间不得超过 N 毫

秒。

针对HotSpot VM的实现，它里面的GC其实准确分类只有两大种：

Partial GC：并不收集整个GC堆的模式

Young GC：只收集young gen的GC

Old GC：只收集old gen的GC。只有CMS的concurrent collection是这个模式

Mixed GC：收集整个young gen以及部分old gen的GC。只有G1有这个模式

Full GC：收集整个堆，包括young gen、old gen、perm gen（如果存在的话）等所有部分的模式。

Major GC通常是跟full GC是等价的，收集整个GC堆。但因为HotSpot VM发展了这么多年，外界对各种名词的解读已经完全混乱了，当有人说“major GC”的时候一定要问清楚他想要指的是上面的full GC还是old gen。

**六、调优工具及命令**

**（一）调优命令**

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。

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，JVM Memory Map命令用于生成heap dump文件

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

jstack，用于生成java虚拟机当前时刻的线程快照。

jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

-X 开头的参数都是非标准的参数（不是所有的JVM都实现了）

-XX 都是不稳定的并且不推荐在生产环境中使用

\* 布尔类型的参数

格式 -XX:+<option> 打开/-XX:-<option> 关闭

例如: -XX:PrintGCDetails 打开GC信息(调优和定位内存回收问题经常使用)

\* 数字型参数

格式 -XX:<option>=<number>

例如 -XX:NewRatio=2

\* 字符型参数

格式 -XX:<option>=<string>

例如 -XX:HeapDumpPath=./java\_pid.hprof

**（二）调优工具**

常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控

jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。

MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗

GChisto，一款专业分析gc日志的工具

# 七、HotSpot 虚拟机对象

**（一）对象的创建**

遇到 new 指令时，首先检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化过。如果没有，执行相应的类加载。

类加载检查通过之后，为新对象分配内存(内存大小在类加载完成后便可确认)。在堆的空闲内存中划分一块区域(‘指针碰撞-内存规整’或‘空闲列表-内存交错’的分配方式)。

前面讲的每个线程在堆中都会有私有的分配缓冲区(TLAB)，这样可以很大程度避免在并发情况下频繁创建对象造成的线程不安全。

内存空间分配完成后会初始化为 0(不包括对象头)，接下来就是填充对象头，把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 分代年龄等信息存入对象头。

执行 new 指令后执行 init 方法后才算一份真正可用的对象创建完成。

**（二）对象的内存布局**

1、对象头(Header)：包含两部分，第一部分用于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，32 位虚拟机占 32 bit，64 位虚拟机占 64 bit。官方称为 ‘Mark Word’。第二部分是类型指针，即对象指向它的类的元数据指针，虚拟机通过这个指针确定这个对象是哪个类的实例。另外，如果是 Java 数组，对象头中还必须有一块用于记录数组长度的数据，因为普通对象可以通过 Java 对象元数据确定大小，而数组对象不可以。

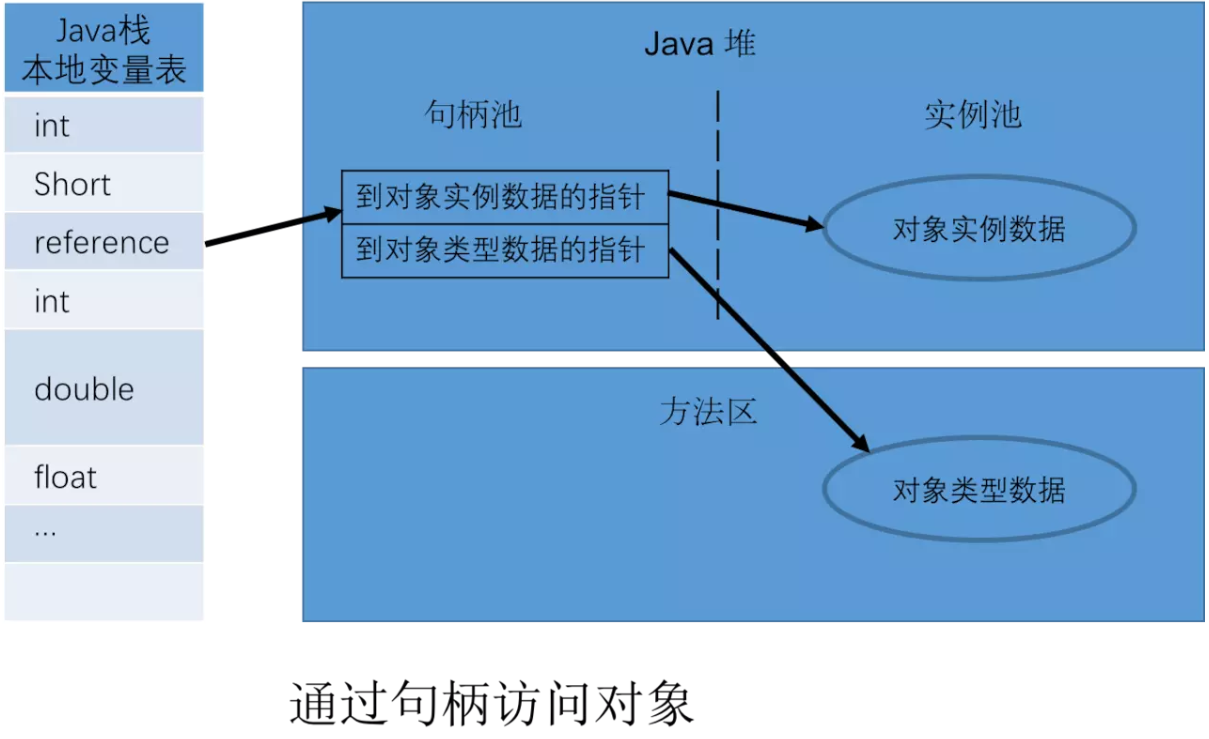
2、 实例数据(Instance Data)：程序代码中所定义的各种类型的字段内容(包含父类继承下来的和子类中定义的)。

3、对齐填充(Padding)：不是必然需要，主要是占位，保证对象大小是某个字节的整数倍。

**（三）对象的访问定位**

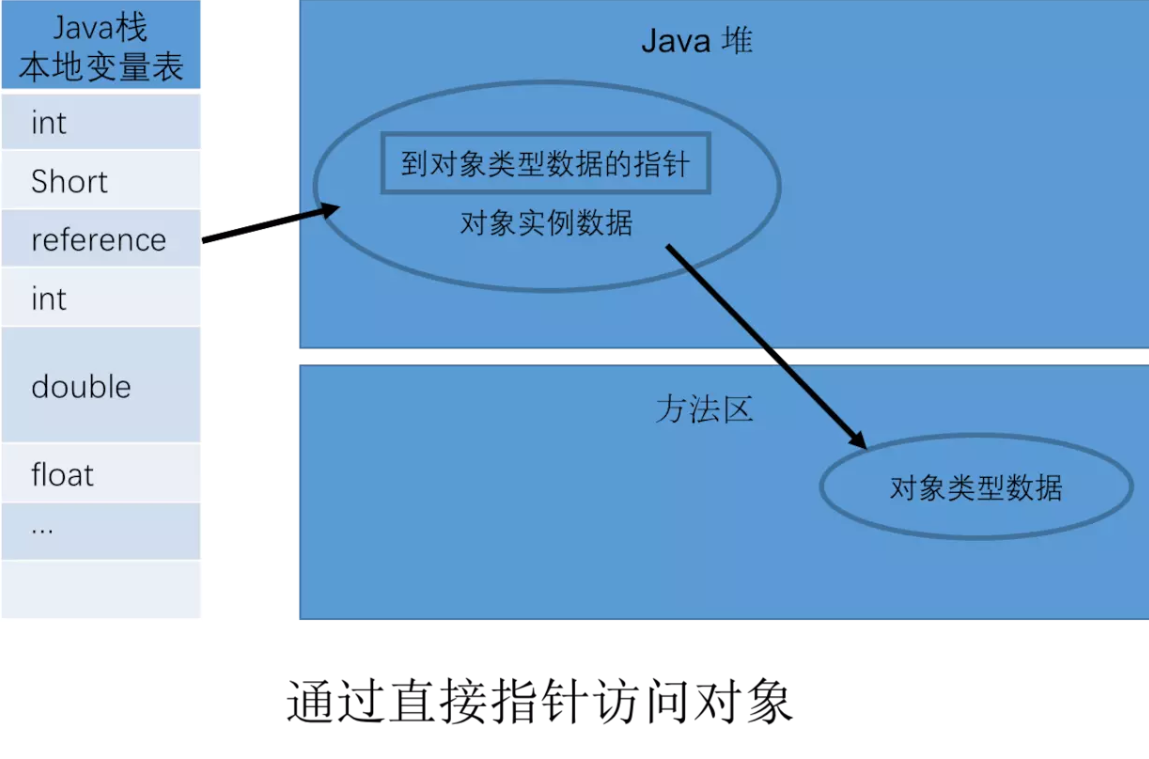
1、通过句柄访问

Java 堆中会分配一块内存作为句柄池。reference 存储的是句柄地址。详情见图。



2、使用直接指针访问

reference 中直接存储对象地址



比较：使用句柄的最大好处是 reference 中存储的是稳定的句柄地址，在对象移动(GC)是只改变实例数据指针地址，reference 自身不需要修改。直接指针访问的最大好处是速度快，节省了一次指针定位的时间开销。如果是对象频繁 GC 那么句柄方法好，如果是对象频繁访问则直接指针访问好。

**八、引用类型**

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象是否可达，判定对象是否可被回收 都与引用有关。

Java 提供了四种强度不同的引用类型。

1. 强引用 被强引用关联的对象不会被回收。

使用 new 一个新对象的方式来创建强引用。

Object obj = new Object();

2. 软引用

被软引用关联的对象只有在内存不够的情况下才会被回收。 使用 SoftReference 类来创建软引用。

Object obj = new Object();

SoftReference<Object> sf = new SoftReference<Object>(obj);

obj =null; // 使对象只被软引用关联

3. 弱引用

被弱引用关联的对象一定会被回收，也就是说它只能存活到下一次垃圾回收发生之前。 使用 WeakReference 类来创建弱引用。

Object obj = new Object();

WeakReference<Object> wf = new WeakReference<Object>(obj);

obj = null;

4. 虚引用

又称为幽灵引用或者幻影引用，一个对象是否有虚引用的存在，不会对其生存时间造成影响，也无法通过虚引用得到 一个对象。

为一个对象设置虚引用的唯一目的是能在这个对象被回收时收到一个系统通知。 使用 PhantomReference 来创建虚引用。

Object obj = new Object();

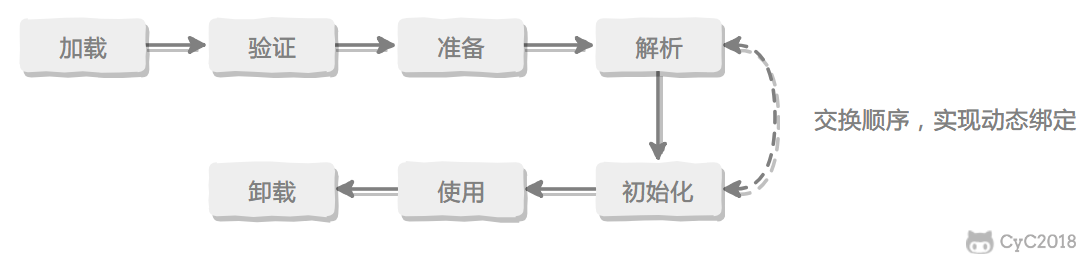
PhantomReference<Object> pf = new PhantomReference<Object>(obj, null);

obj = null;

**九、类加载机制**

类是在运行期间第一次使用时动态加载的，而不是一次性加载所有类。因为如果一次性加载，那么会占用很多的内存。

**类的生命周期**



包括以下 7 个阶段：

加载（Loading）

验证（Verification）

准备（Preparation）

解析（Resolution）

初始化（Initialization）

使用（Using）

卸载（Unloading）

类加载过程

包含了加载、验证、准备、解析和初始化这 5 个阶段。

1. 加载

加载是类加载的一个阶段，注意不要混淆。

加载过程完成以下三件事：

通过类的完全限定名称获取定义该类的二进制字节流。

将该字节流表示的静态存储结构转换为方法区的运行时存储结构。

在内存中生成一个代表该类的 Class 对象，作为方法区中该类各种数据的访问入口。其中二进制字节流可以从以下方式中获取：

从 ZIP 包读取，成为 JAR、EAR、WAR 格式的基础。从网络中获取，最典型的应用是 Applet。运行时计算生成，例如动态代理技术，在 java.lang.reflect.Proxy 使用 ProxyGenerator.generateProxyClass的代理类的二进制字节流。由其他文件生成，例如由 JSP 文件生成对应的 Class 类。

2. 验证

确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

3. 准备

类变量是被 static 修饰的变量，准备阶段为类变量分配内存并设置初始值，使用的是方法区的内存。

实例变量不会在这阶段分配内存，它会在对象实例化时随着对象一起被分配在堆中。应该注意到，实例化不是类加载的一个过程，类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次。初始值一般为 0 值，例如下面的类变量 value 被初始化为 0 而不是 123。

public static int value = 123;

如果类变量是常量，那么它将初始化为表达式所定义的值而不是 0。例如下面的常量 value 被初始化为 123 而不是0。

public static final int value = 123;

4. 解析

将常量池的符号引用替换为直接引用的过程。

其中解析过程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 的动态绑定。

5. 初始化

初始化阶段才真正开始执行类中定义的 Java 程序代码。初始化阶段是虚拟机执行类构造器 <clinit>() 方法的过程。在准备阶段，类变量已经赋过一次系统要求的初始值，而在初始化阶段，根据程序员通过程序制定的主观计划去初始化类变量和其它资源。

<clinit>() 是由编译器自动收集类中所有类变量的赋值动作和静态语句块中的语句合并产生的，编译器收集的顺序由语句在源文件中出现的顺序决定。特别注意的是，静态语句块只能访问到定义在它之前的类变量，定义在它之后的类变量只能赋值，不能访问。由于父类的 <clinit>() 方法先执行，也就意味着父类中定义的静态语句块的执行要优先于子类。

接口中不可以使用静态语句块，但仍然有类变量初始化的赋值操作，因此接口与类一样都会生成 <clinit>() 方法。但接口与类不同的是，执行接口的 <clinit>() 方法不需要先执行父接口的 <clinit>() 方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的 <clinit>() 方法。

虚拟机会保证一个类的 <clinit>() 方法在多线程环境下被正确的加锁和同步，如果多个线程同时初始化一个类，只会有一个线程执行这个类的 <clinit>() 方法，其它线程都会阻塞等待，直到活动线程执行 <clinit>() 方法完毕。如果在一个类的 <clinit>() 方法中有耗时的操作，就可能造成多个线程阻塞，在实际过程中此种阻塞很隐蔽。类初始化时机

1. 主动引用

虚拟机规范中并没有强制约束何时进行加载，但是规范严格规定了有且只有下列五种情况必须对类进行初始化（加载、验证、准备都会随之发生）：

遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类没有进行过初始化，则必须先触发其初始化。最常见的生成这 4 条指令的场景是：使用 new 关键字实例化对象的时候；读取或设置一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）的时候；以及调用一个类的静态方法的时候。

使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行初始化，则需要先触发其初始化。当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

当虚拟机启动时，用户需要指定一个要执行的主类（包含 main() 方法的那个类），虚拟机会先初始化这个主类；当使用 JDK 1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果为REF\_getStatic, REF\_putStatic, REF\_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化；

2. 被动引用

以上 5 种场景中的行为称为对一个类进行主动引用。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。被动引用的常见例子包括：

通过子类引用父类的静态字段，不会导致子类初始化。

System.out.println(SubClass.value); // value 字段在 SuperClass 中定义通过数组定义来引用类，不会触发此类的初始化。该过程会对数组类进行初始化，数组类是一个由虚拟机自动生成的、直接继承自 Object 的子类，其中包含了数组的属性和方法。

SuperClass[] sca = new SuperClass[10];

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。

System.out.println(ConstClass.HELLOWORLD);

类与类加载器

两个类相等，需要类本身相等，并且使用同一个类加载器进行加载。这是因为每一个类加载器都拥有一个独立的类名称空间。

这里的相等，包括类的 Class 对象的 equals() 方法、isAssignableFrom() 方法、isInstance() 方法的返回结果为true，也包括使用 instanceof 关键字做对象所属关系判定结果为 true。

**类加载器分类**

从 Java 虚拟机的角度来讲，只存在以下两种不同的类加载器：

1.启动类加载器（Bootstrap ClassLoader），使用 C++ 实现，是虚拟机自身的一部分；

2.所有其它类的加载器，使用 Java 实现，独立于虚拟机，继承自抽象类 java.lang.ClassLoader。

从 Java 开发人员的角度看，类加载器可以划分得更细致一些：

1.启动类加载器（Bootstrap ClassLoader）此类加载器负责将存放在 <JRE\_HOME>\lib 目录中的，或者被-Xbootclasspath 参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被 Java程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给启动类加载器，直接使用 null 代替即可。

2.扩展类加载器（Extension ClassLoader）这个类加载器是由ExtClassLoader（sun.misc.Launcher$ExtClassLoader）实现的。它负责将<JAVA\_HOME>/lib/ext 或者被java.ext.dir 系统变量所指定路径中的所有类库加载到内存中，开发者可以直接使用扩展类加载器。

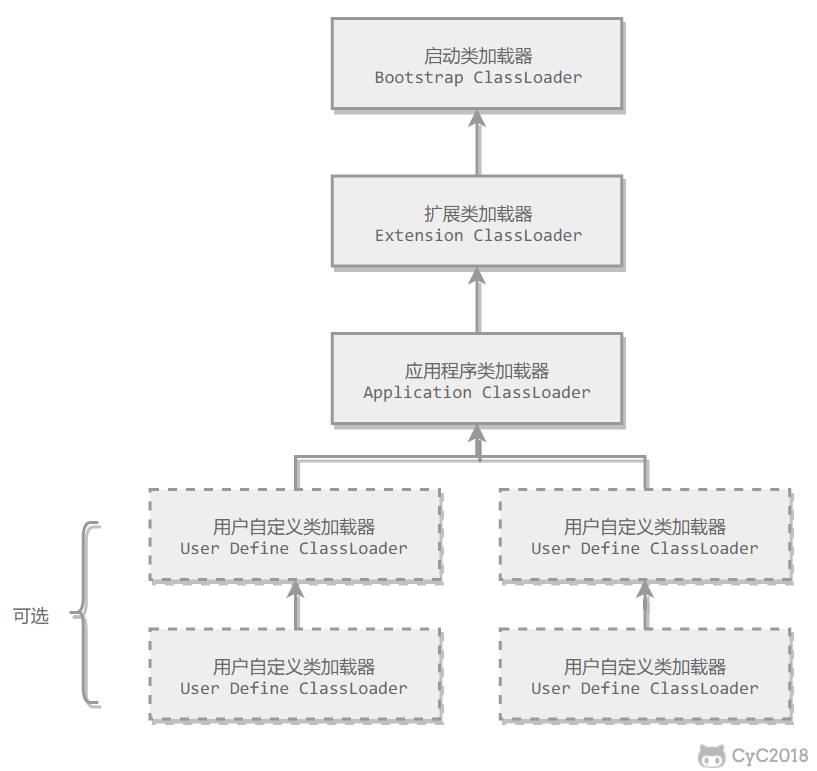
3.应用程序类加载器（Application ClassLoader）这个类加载器是由AppClassLoader（sun.misc.Launcher$AppClassLoader）实现的。由于这个类加载器是 ClassLoader 中的getSystemClassLoader() 方法的返回值，因此一般称为系统类加载器。它负责加载用户类路径（ClassPath）

上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

**双亲委派模型**

应用程序是由三种类加载器互相配合从而实现类加载，除此之外还可以加入自己定义的类加载器。

下图展示了类加载器之间的层次关系，称为双亲委派模型（Parents Delegation Model）。该模型要求除了顶层的启动类加载器外，其它的类加载器都要有自己的父类加载器。这里的父子关系一般通过组合关系（Composition）来实现，而不是继承关系（Inheritance）。



1. 工作过程

一个类加载器首先将类加载请求转发到父类加载器，只有当父类加载器无法完成时才尝试自己加载。

2. 好处

使得 Java 类随着它的类加载器一起具有一种带有优先级的层次关系，从而使得基础类得到统一。

例如 java.lang.Object 存放在 rt.jar 中，如果编写另外一个 java.lang.Object 并放到 ClassPath 中，程序可以编译通过。由于双亲委派模型的存在，所以在 rt.jar 中的 Object 比在 ClassPath 中的 Object 优先级更高，这是因为 rt.jar中的 Object 使用的是启动类加载器，而 ClassPath 中的 Object 使用的是应用程序类加载器。rt.jar 中的 Object 优先级更高，那么程序中所有的 Object 都是这个 Object。

3. 实现

以下是抽象类 java.lang.ClassLoader 的代码片段，其中的 loadClass() 方法运行过程如下：先检查类是否已经加载过，如果没有则让父类加载器去加载。当父类加载器加载失败时抛出 ClassNotFoundException，此时尝试自己去加载。