摘自：《深入理解Java虚拟机：JVM高级特性与最佳实践.周志明.高清扫描版.pdf》

二、自动内存管理机制

JVM在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域。

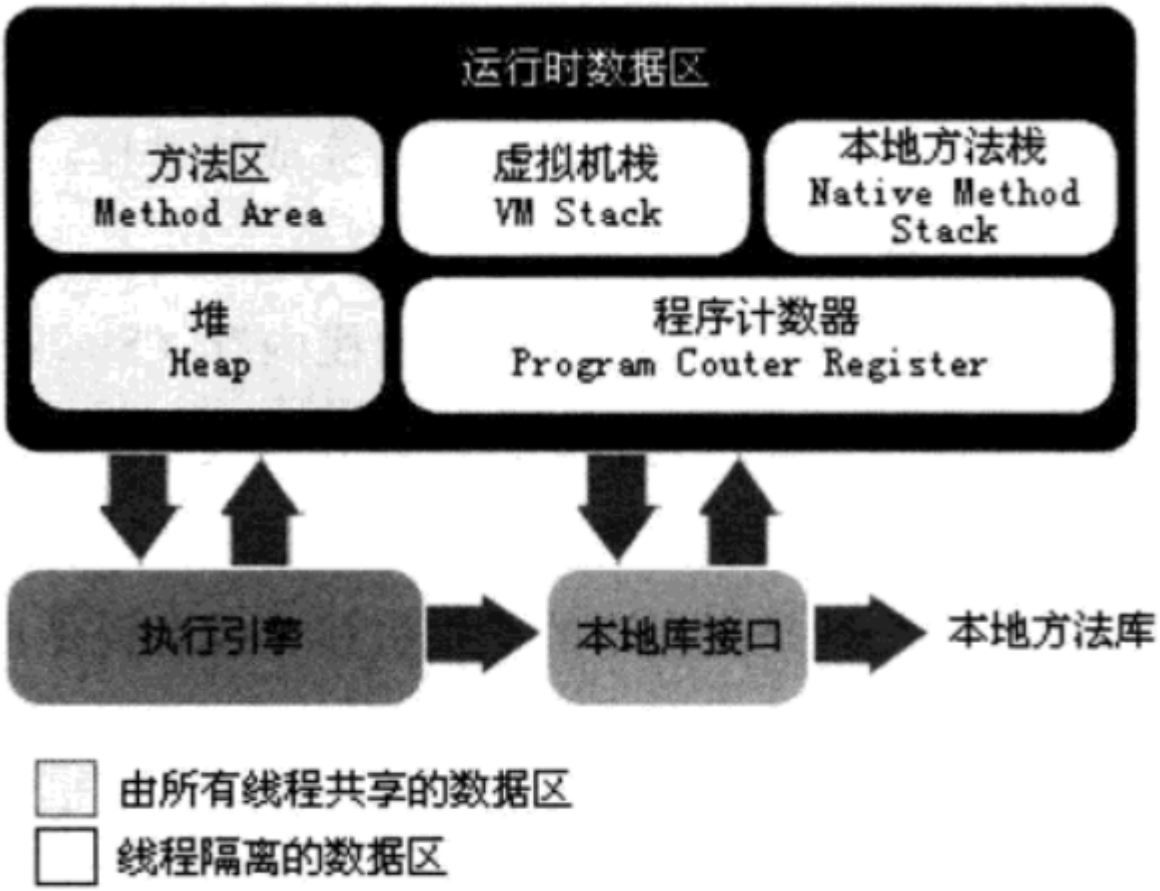
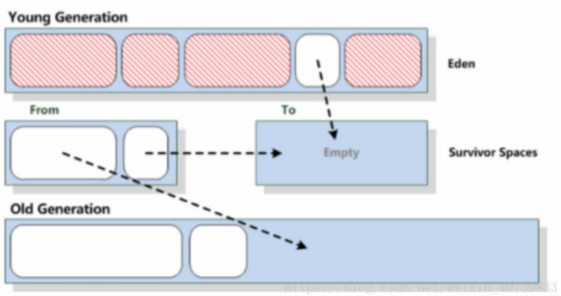


图 2-1　Java 虚拟机运行时数据区

运行时数据区

方法区
Method Area
存储已被虚拟机加载的类信息、常量、静态变量、即时编译器变异后的代码等数据
运行时常量池:

堆
Heap

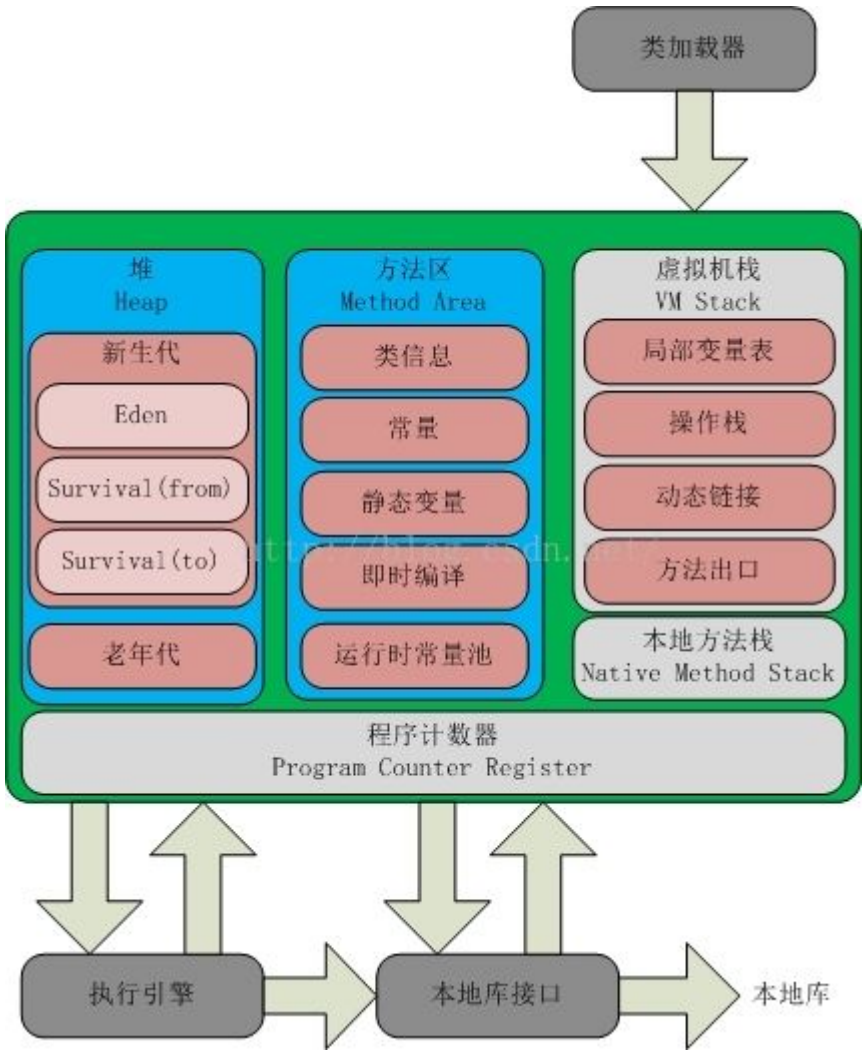


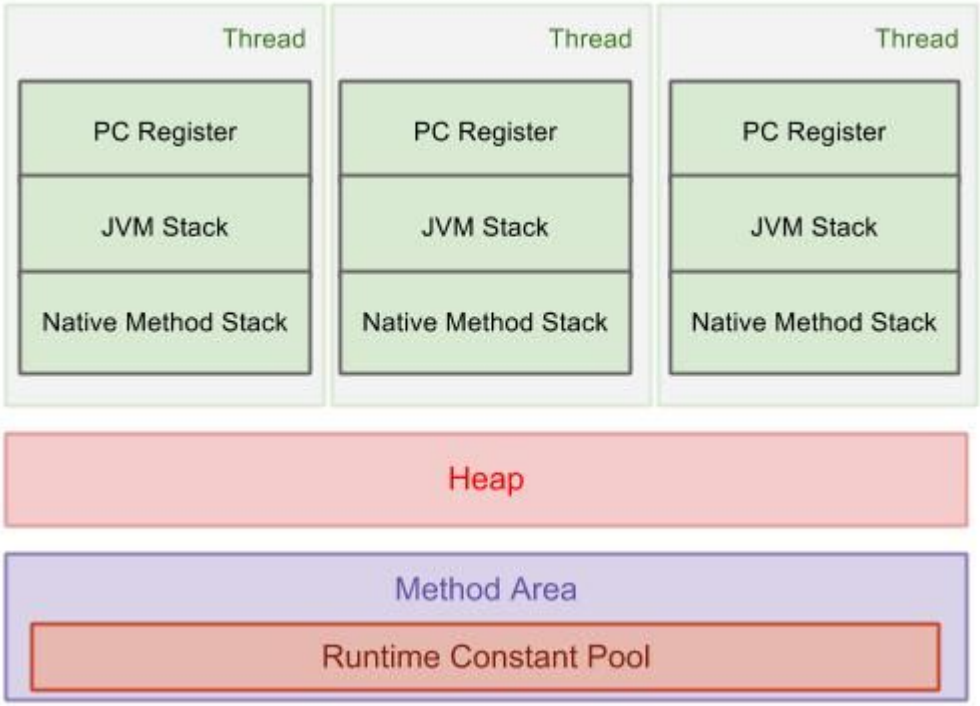
虚拟机栈
VM Stack

栈帧（Stack Frame）
局部变量表：存放基本数据类型、对象引用、returnAddress 类型
操作栈
动态链接
方法出口等信息

本地方法栈
Native Method Stack
（为执行 Native 方法服务）

程序计数器
Program Counter Register





1.程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节指令的地址；如果正在执行的是Native方法，这个计数器值则为空（Undefined）。**此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。**

2.虚拟机栈

和程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址引用指针，也可能是指向一个代表对象的句柄或者其他与对象相关的位置）和returnAddress类型（指向了一条字节码指令的地址）。

其中64位长度的long和double类型的数据会占用2个局部变量空间（Slot），其余的数据类型只占用1个。局部变量表所需的内存空间在编译期完成分配，当进入一个方法时，这个方法需要再帧中分配多大的局部变量空间完全是确定的，在方法运行期间不会改变局部变量表的大小。

在Java虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果虚拟机栈可以动态扩展（当前大部分的Java虚拟机都可以动态扩展，只不过Java虚拟机规范中也允许固定长度的虚拟机栈），当扩展时无法申请到足够的内存时会抛出OutOfMemoryError异常。

3.本地方法栈

本地方法栈（Native Method Stacks）与虚拟机所发挥的作用是非常相似的，其区别不过是虚拟机为虚拟机执行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。与虚拟机一样，本地方法栈区域会抛出StackOverflowError和OutOfMemoryError异常。

4.Java堆

对于大多数应用来说，Java堆（Java Heap）是Java虚拟机所管理的内存中最大的一块。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。在Java虚拟机规范中描述的是：所有对象实例以及数组都要在堆上分配，但是随着JIT编译器的发展和逃逸分析技术的逐渐成熟，栈分配、标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么“绝对”了。

Java堆是垃圾收集管理的主要区域，因此很多时候也被称作“GC堆”（Garbage Collected Heap）。如果从内存回收的角度来看，由于现在收集器基本都是采用的分代收集算法，所以Java堆中还可以细分为：新生代和老年代；再细致一点的有Eden空间、From Survivor空间、To Survivor空间等。如果从内存分配的角度看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer，TLAB）。不过，无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好地回收内存，或者更快地分配内存。

Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。在实现上，既可以固定大小，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过-Xmx和-Xms控制）。如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

5.方法区

方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，**它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器变异后的代码等数据。**虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但它却有一个别名叫做Non-Heap（非堆），目的应该是与Java堆区分开。

Java虚拟机规范对这个区域的限制非常宽松，除了和Java堆一样不需要连续的内存和可疑选择固定大小或者可扩展外，还可以选择不实现垃圾收集。相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这个区域的回收目标主要是针对常量池的回收和对类型的卸载，一般来说这个区域的回收“成绩”比较难令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是有必要的。

当方法区无法满足内存分配的需求时，将抛出OutOfMemoryError异常。

6.运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。

运行时常量池相对于Class文件常量池的另外一个重要特征是具备动态性，Java语言并不要求常量一定只能在编译期产生，也就是并非置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量池放入池中，这种特性被开发人员利用得比较多的便是String类的intern()方法。

> 关于String类的intern()方法：
>
> - 返回字符串对象的规范化表示形式。
> - 一个初始时为空的字符串池，它由类 String 私有地维护。
> - 当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（该对象由 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添加到池中，并且返回此 String 对象的引用。

> - 它遵循对于任何两个字符串 s 和 t，当且仅当 s.equals(t) 为 true 时，s.intern() == t.intern() 才为 true。
> - 所有字面值字符串和字符串赋值常量表达式都是内部的。

```
String str1 = "a";
String str2 = "b";
String str3 = "ab";
String str4_1 = "a" + "b";
String str4 = str1 + str2;
String str5 = new String("ab");
// 字符串中内容相同，比较方式是一位一位地比较字符
System.out.println(str5.equals(str3));
// str4不会进入到常量池中
System.out.println(str3 == str4);
// str4_1字符串相加的时候，都是静态字符串的结果会添加到字符串池
System.out.println(str3 == str4_1);
// new创建的String不会进入常量池
System.out.println(str5 == str3);
// intern()获取的是常量池中对象的引用
System.out.println(str5.intern() == str3);
System.out.println(str5.intern() == str4);
```

7.直接内存

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但是这部分内存也被频繁地适用，而且也有可能导致OutOfMemoryError异常出现。

在JDK1.4中新加入了NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，它可以使用Native函数直接分配堆外内存，然后通过一个存储在Java里面的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回复制数据。

8.对象访问

```
Object obj = new Object();
```

假设这代码出现在方法体中：

- “Object obj”这部分的语义将会反映到Java栈的本地变量表中，作为一个reference类型数据出现。
- “new Object()”这部分的语义将会反映到Java堆中，形成一块存储了Object类型所有实例数据值（Instance Data，对象中各个实例字段的数据）的结构化内存，根据具体类型以及虚拟机实现的对象内存布局（Object Memory Layout）的不同，这块内存的长度是不固定的。
- 另外，在Java堆中还必须包含能查找到此对象类型数据（如对象类型、父类、实现的接口、方法等）的地址信息，这些类型数据则存储在方法区中。

由于reference类型访问到Java堆中的对象的具体位置，主流的访问方式有两种：使用句柄和直接指针。

- 使用句柄：Java堆中将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，句柄中包含了对象实例数据和类型数据各自的具体信息。

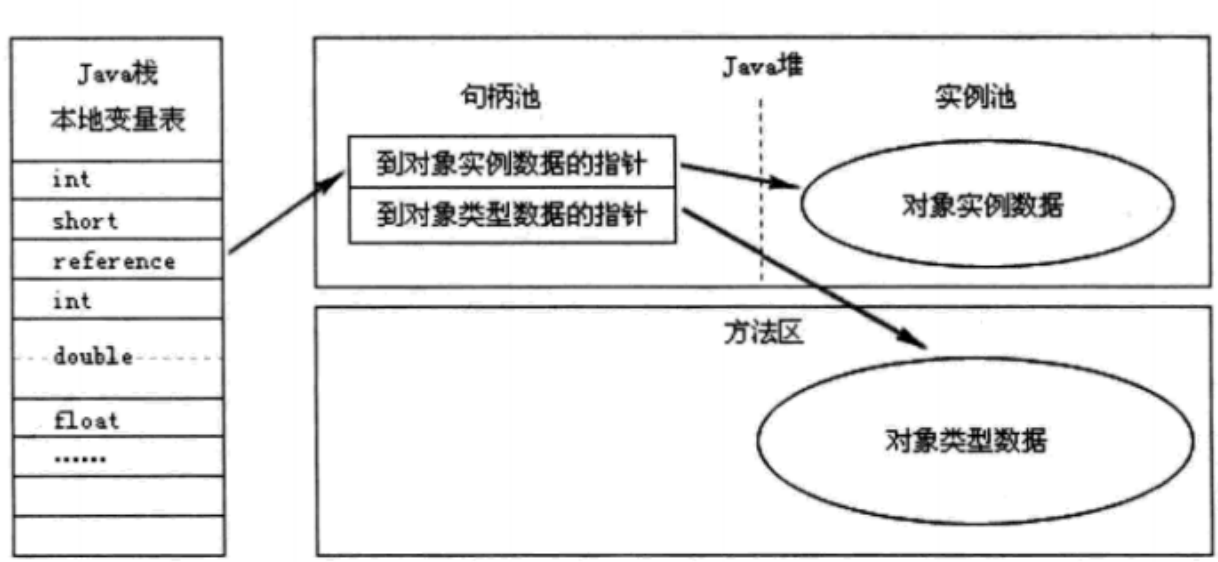


图 2-2　通过句柄访问对象

- 直接指针：Java堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference中直接存储的就是对象地址。

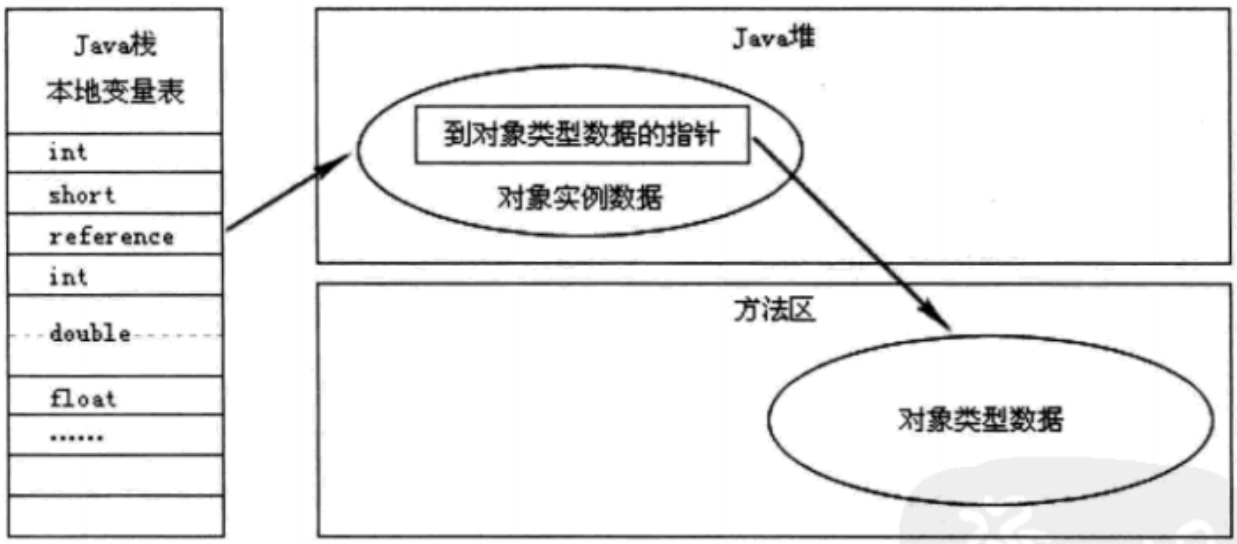


图 2-3　通过直接指针访问对象

两种对象的访问方式各有优势，使用句柄方式访问的最大好处就是reference中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时会改变句柄中的实例数据指针，而reference本身不需要被修改。

使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在Java中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。Sun HotSpot使用的是第二种方式进行对象的访问。

三、垃圾收集器与内存分配策略

程序计数器、虚拟机栈、本地方法栈三个区域随线程而生，随线程而灭，这几个区域的内存分配和回收都具备确定性，方法结束或线程结束时，内存自然就跟随着回收了。

Java堆和方法区只有在程序处于运行期间才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的就是这部分内存。

确定哪些对象“存活”，哪些已经“死去”（不可能再被任何途径使用的对象）的方法有两种：引用计数算法和根搜索法。

1.引用计数法

给对象添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器都为0的对象就是不可能再被使用的。

但是它很难解决对象之间的相互循环引用问题。

2.根搜索法（GC Roots Tracing)

算法思路是：通过一系列名为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连（用图论的话来说就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。

在Java语言中，可作为GC Roots的对象包括下面几种：

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

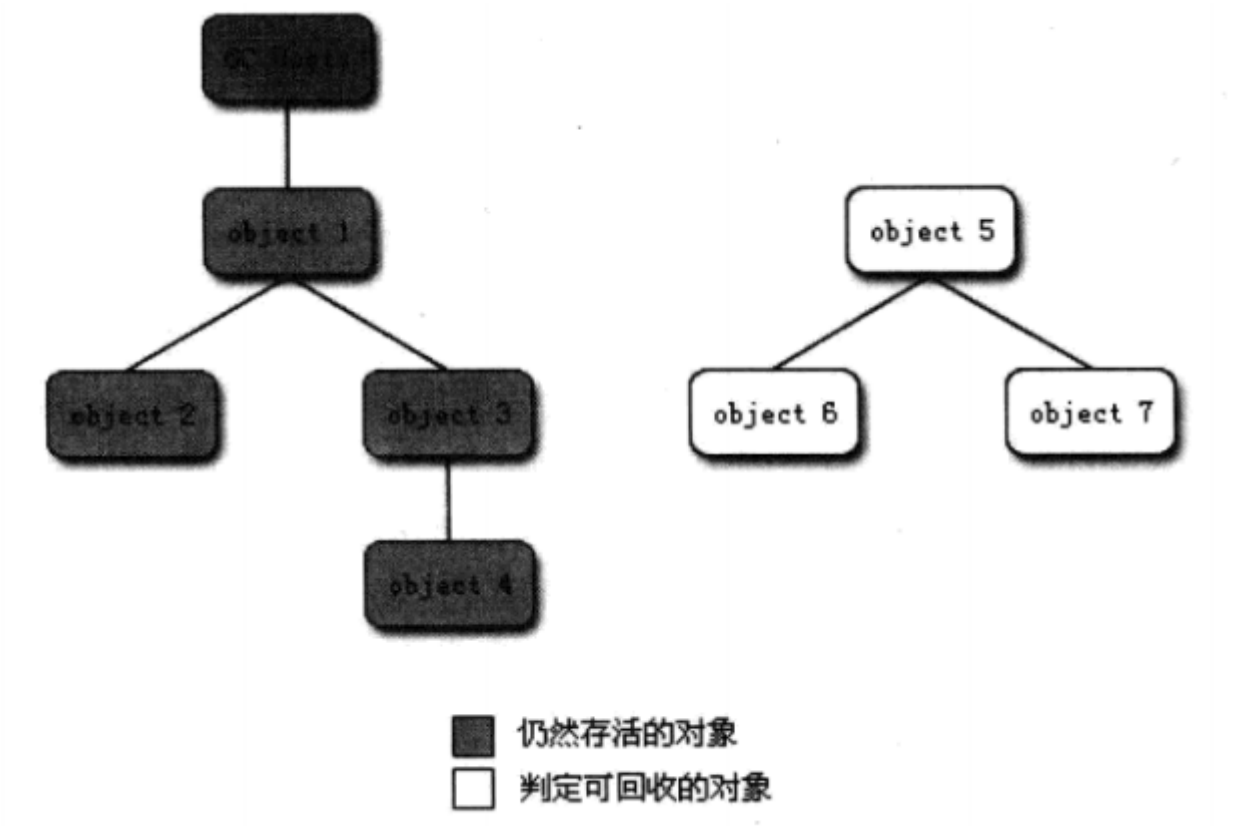


图 3-1　根搜索算法判定对象是否可回收

在JDK1.2之后，Java对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）四中，这四种引用强度一次逐渐减弱。

- **强引用**就是在程序代码中普遍存在的，类似“Object obj=new Object()”这类的引用，只要强引用还在，垃圾收集器永远不会回收掉被引用的对象。
- **软引用**用来描述一些还有用，但并非必须的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中并进行第二次回收。如果这次回收还是没有足够的内存，才回抛出内存溢出异常。在JDK1.2之后，提供了SoftReference类来实现软引用。可以用来做内存敏感的高速缓存。
- **弱引用**也是用来描述非必须对象的，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。无论当前内存是否足够，都会回收掉只被弱引用关联的对象。
- **虚引用也被称为幽灵引用或者幻影引用**，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是希望能在这个对象被收集器回收时收到一个系统通知。虚引用必须和引用队列关联使用，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之 关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

3.生存还是死亡

在根搜索算中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在跟搜索后发现没有与GC Roots相连接的引用链，那么它将会被第一次标记并且进行一次筛选，筛选条件是此对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行finalize()方法，那么这个对象将会被放置在一个名为F-Queue的队列中，并在稍后由一条虚拟机自动建立的、低优先级的Finalizer线程去执行。这里所谓的“执行”是指虚拟机会出发这个方法，但并不承诺会等待它运行结束。这样的原因是，如果一个对象在finalize()方法中执行缓慢，或者发生了死循环（更极端的情况），将很可能会导致F-Queue队列中的其他对象永久处于等待状态，甚至导致整个内存回收系统崩溃。finalize()方法是对象逃脱死亡的最后一次机会，稍后GC将对F-Queue重的对象进行第二次小规模的标记，如果对象要在finalize()中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this关键字）赋值给某个类变量或对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这个时候还没有逃脱，那它就真的离死不远了。

4.回收方法区

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。

回收废弃常量是没有任何对象引用常量池中的常量时，也没有其他地方引用这个字面量时，如果发生内存怒回收，此常量会被“请”出常量池。常量池中的其他类（接口）、方法、字段的符号引用也与此类似。

判定一个类是否是“无用的类”的条件相对苛刻许多。类需要同时满足下面3个条件才能算是“无用的类”。

- 该类所有的实例都已经被回收，Java对重不存在该类的任何实例
- 加载该类的ClassLoader已经被回收
- 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类。

在大量使用反射、动态代理、CGLib等bytecode框架的场景，以及动态生成JSP和OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

5.垃圾收集算法

（1）标记-清除法

最基础的收集算法是“标记-清除”（Mark-Sweep）算法，算法分为”标记“和”清除“两个阶段：首先标记出所有需要回收的对象，标记完成后统一回收掉所有被标记的对象。标记过程前面已经介绍了。

它存在连个主要的缺点：一是效率问题，标记和清除过程的效率都不高；另一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存空间而不得不提前出发另一次垃圾收集动作。

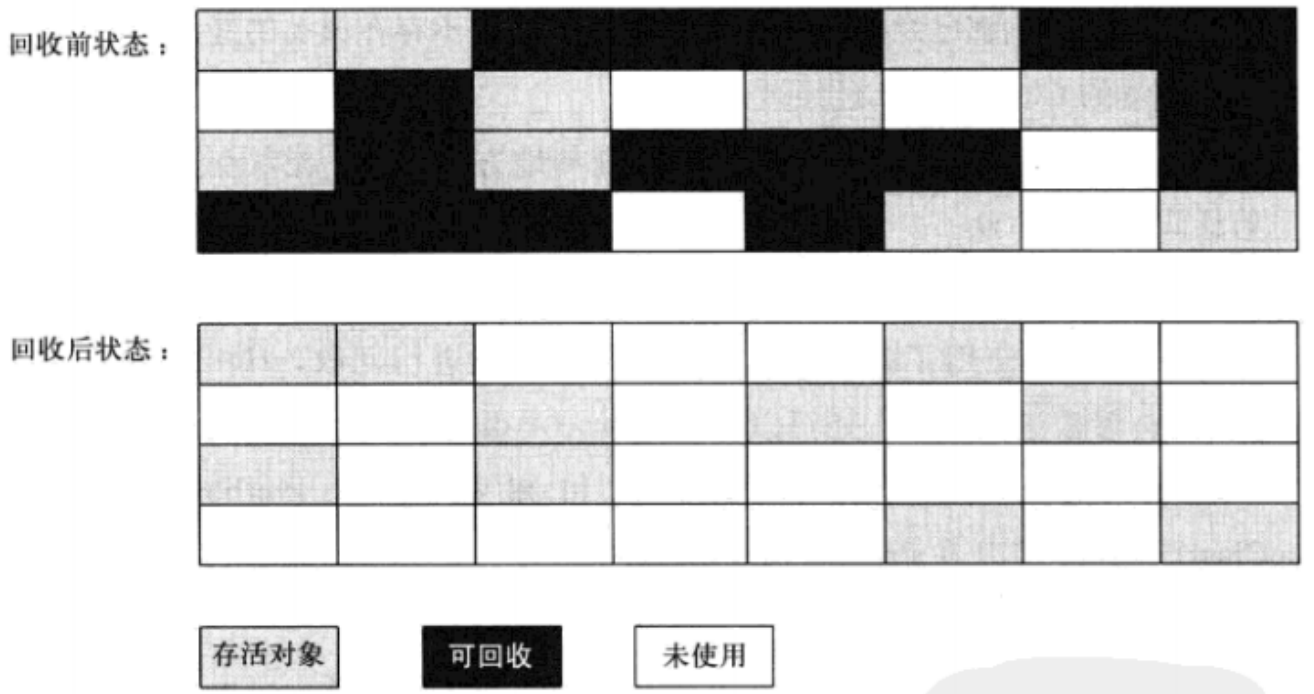


图 3-2 “标记－清除”算法示意图

（2）复制算法

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它将可用内存按容量划分为大小相等的两块，每次只是用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对其中的一块进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为原来的一般。

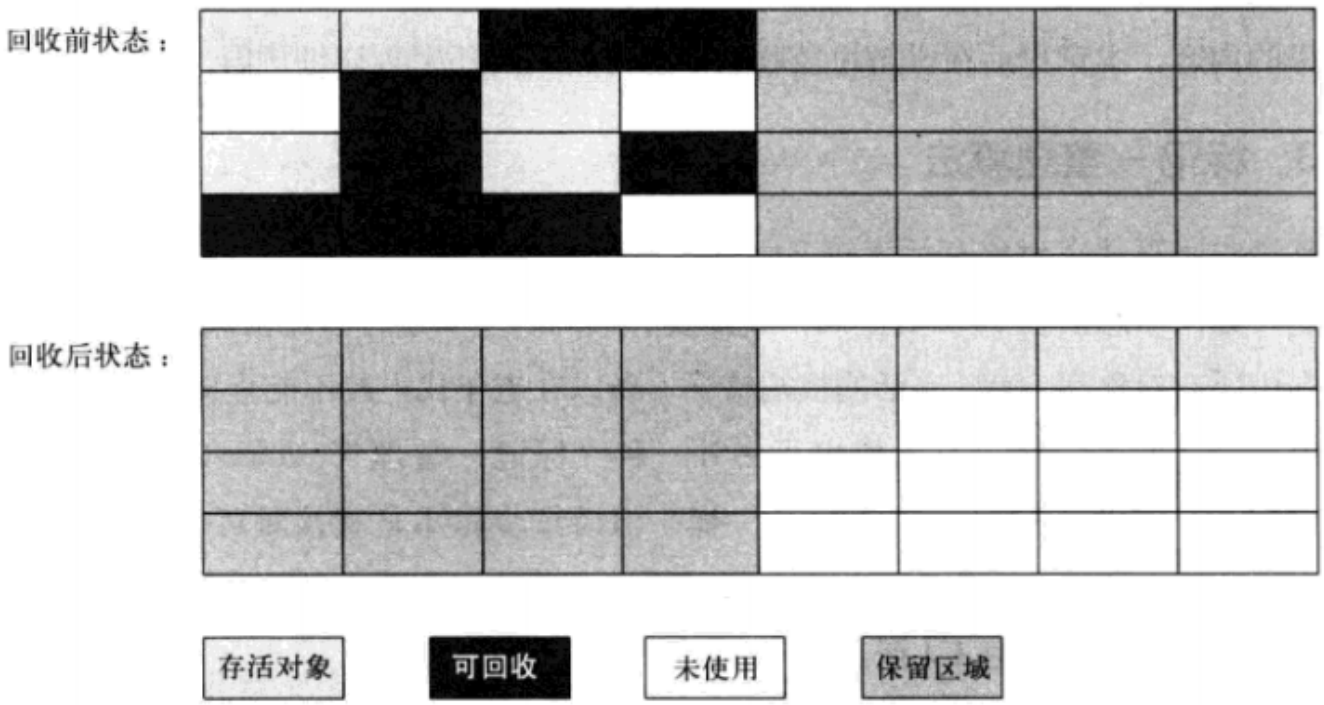


图 3-3　复制算法示意图

现在的商业虚拟机都采用这种收集算法来回收**新生代**，IBM的专门研究表明，新生代中的对象98%是朝生夕死的，所以并不需要按照1:1的比例来划分内存空间，而是将**内存分为一块较大的Eden空间和两块较小的Survivor空间**，每次使用Eden和其中的一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地拷贝到另一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor的空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%，只有10%的内存是会被“浪费”的。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

（3）标记-整理法

复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，随后让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

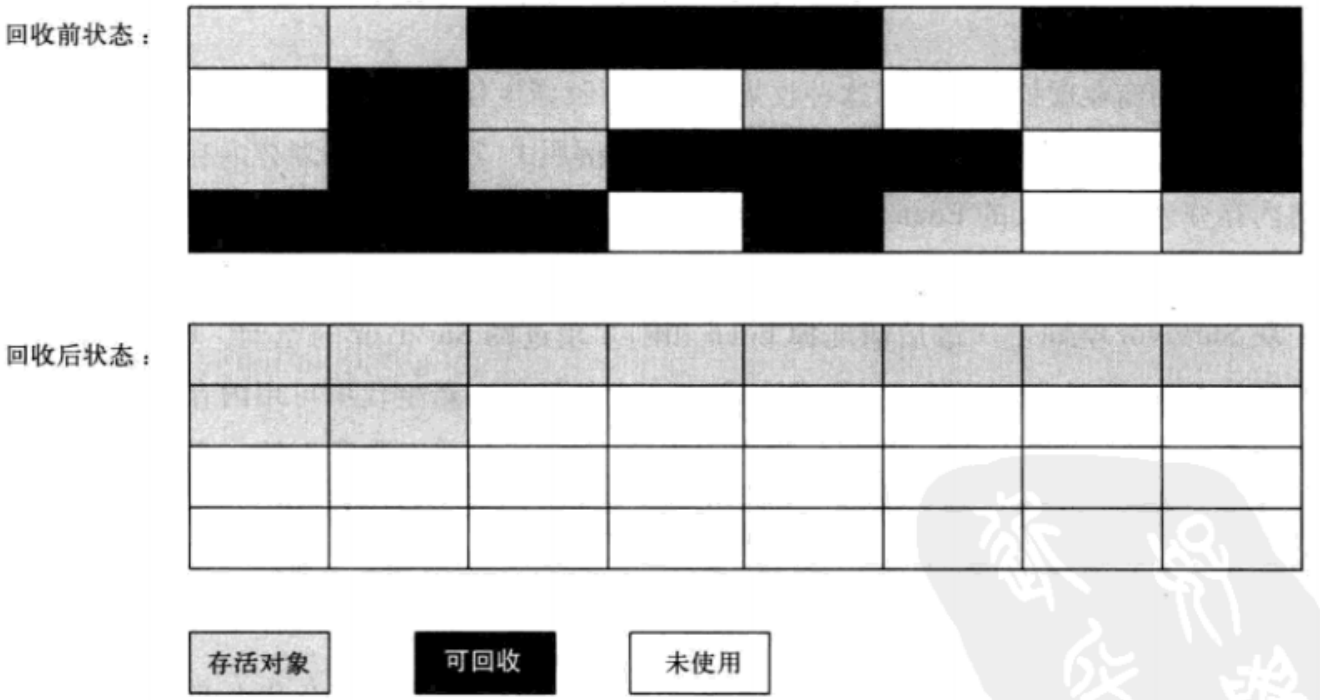


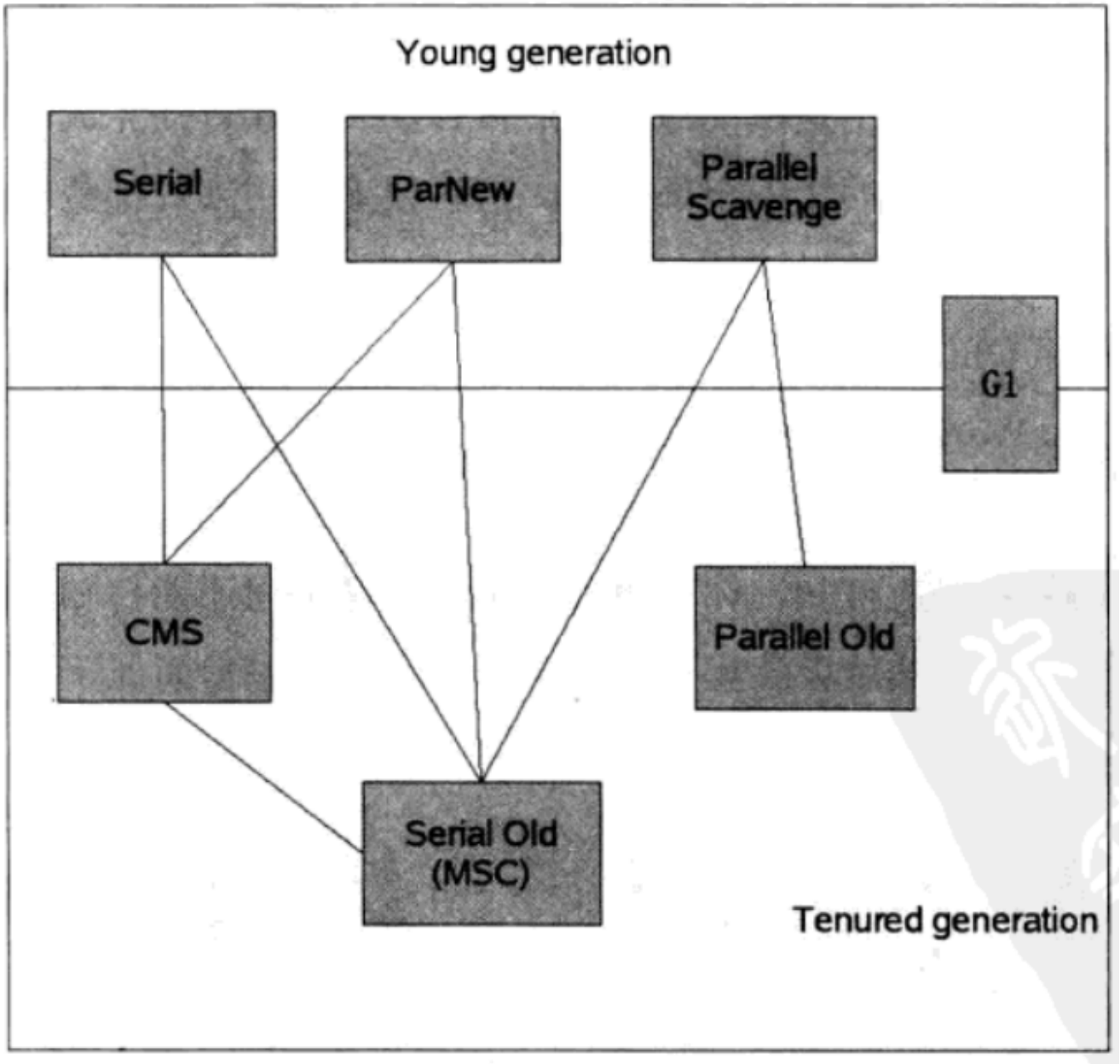
图 3-4 “标记－整理”算法示意图

（4）分代手机算法

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，根据对象的存活周期的不同将内存划分为几块，一般是将Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最合适的手机算法。对于新生代，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法进行垃圾回收。

6.垃圾收集器

如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现。



如果两个收集器之间存在连线，就说明他们可以搭配使用。

（1）Serial收集器

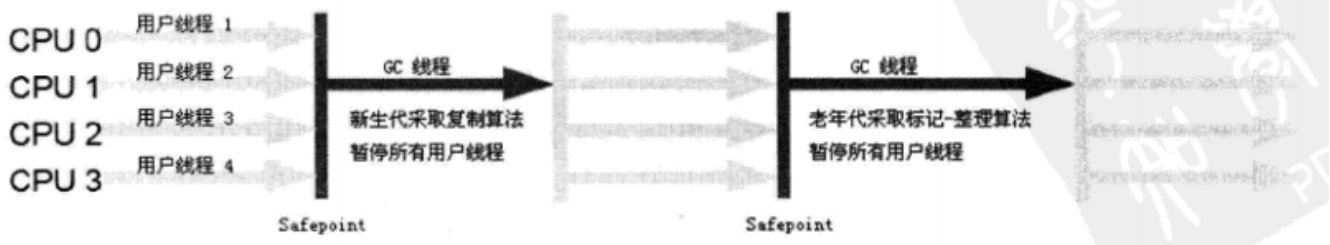


图 3-6　Serial / Serial Old 收集器运行示意图

Serial收集器是最基本、历史最悠久的收集器。这个收集器是一个单线程的收集器，它在进行垃圾收集时，必须暂停其他所有工作线程（Sun将这件事情称之为“Stop The World”），直到它收集结束。“Stop The World”这项工作时有虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户的正常工作的线程全部停掉。它依然是虚拟机运行在Client模式下的默认新生代收集器。

它的优点是：简单而高效（与其他收集器的单线程比）。

要点：

- 针对新生代
- 采用复制算法
- 单线程收集
- Stop The World
- HotSpot在Client模式下的默认新生代收集器

(2) ParNew收集器

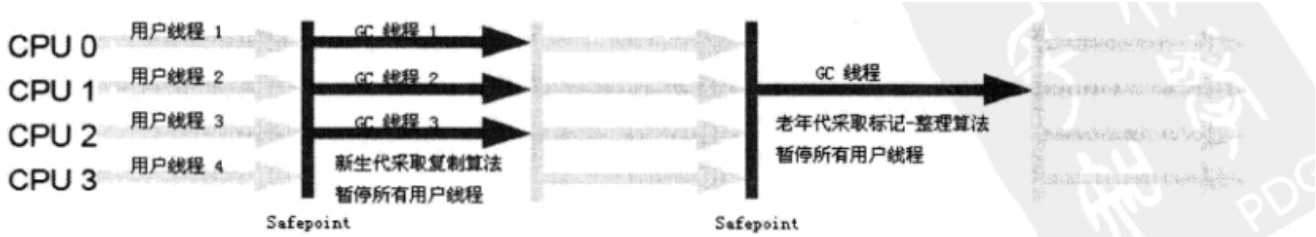


图 3-7　ParNew / Serial Old 收集器运行示意图

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数（例如：-XX:SurvivorRatio、-XX:PretenureSizeThreadhold、-XX:HandlePromotionFailure等）、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样。

ParNew是许多运行Server模式下的虚拟机中首选的新生代收集器。ParNew收集器也是使用-XX:+UseConcMarkSweepGC选项后的默认新生代收集器，也可以使用-XX:+UseParNewGC选项来强制指定它。可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

> 并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态
>
> 并发（Concurrent）：指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户程序继续运行，而垃圾收集程序运行与另一个CPU上。

要点：

- 除了多线程外，其他的和Serial收集器一样

(3) Parallel Scavenge收集器

Parallel Scanvenge收集器也是一个新生代收集器，它也是复制算法的收集器，又是并行的多线程收集器。Parallel Scavenge与其他收集器的关注点不同，Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput）。所谓吞吐量就是CPU用于用户代码的实现与CPU总消耗时间的比值，即 $吞吐量 = \frac{运行用户代码时间}{(运行用户代码时间+垃圾收集时间)}$，虚拟机总共运行了100分钟，其中垃圾收集划掉1分钟，那吞吐量就是99%。

停顿时间越短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；而高吞吐量则可以最高效率地利用CPU时间，尽快地完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Scaveng通过下面两个参数精确控制吞吐量：

- 参数-XX:MaxGCPauseMills控制最大垃圾收集停顿时间
- -XX:GCTimeRatio参数设置吞吐量的大小。第一个参数设置的越小，

GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的，系统将新生代调小，垃圾收集速度就会变得更快，但是同样也会变得更频繁，吞吐量也就会下降。

GCTimeRatio参数的值应当是一个大于0小于100的整数，也就是垃圾收集时间占总时间的比率，相当于是吞吐量的倒数。如果把参数设置为19，那允许的最大GC时间就占总时间的5%（即1/(1+99)），默认值为99，就是允许最大1%（即1/(99+1)）的垃圾收集时间。

要点：

- 新生代收集器
- 采用的是复制算法
- 多线程
- 关注吞吐量，高效率地适用CPU时间

（4）Serial Old收集器

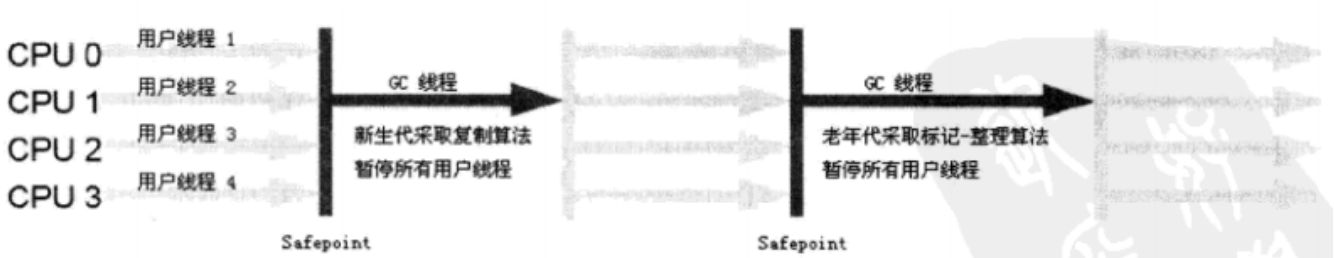


图 3-8　Serial / Serial Old 收集器运行示意图

Serial Old是Serial收集器的老年代版本，它同样是一个单线程的收集器，使用的是“标记-整理”算法。

要点：

- 老年代收集器
- 单线程收集器
- 使用“标记-整理”算法

（5）Parallel Old收集器

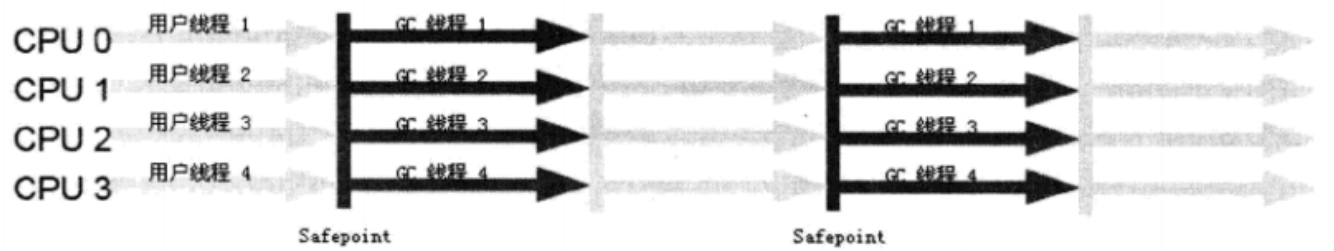


图 3-9　Parallel Scavenge / Parallel Old 收集器运行示意图

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。

要点：

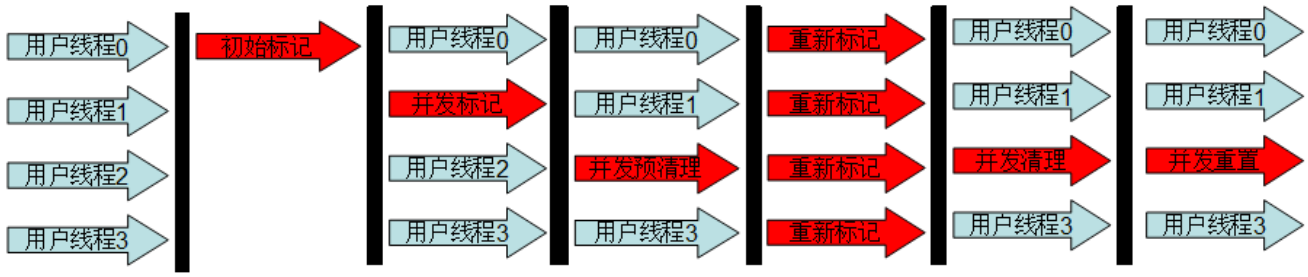
- 老年代收集器
- 多线程
- 使用“标记-整理”算法

（6）CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用都集中在互联网站或B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。

CMS收集器是基于“标记-清除”算法实现的。一般包括6个阶段：

- 初始标记
- 并发标记
- 并发预清理
- 重新标记
- 并发清理
- 并发重置



初始标记：在这个阶段，需要虚拟机停顿正在执行的任务，STW（Stop The World）。这个过程从垃圾回收的“根对象”开始，**只扫描到能够和“根对象”直接关联的对象**，并作为标记。所以这个过程虽然暂停了整个JVM，但是很快就完成了。

并发标记：这个阶段紧随着初始标记阶段，在初始标记的基础上继续向下追溯标记。并发标记阶段，应用程序和并发标记线程并发执行，所以用户不会感受到停顿。

并发预清理：并发预清理阶段仍然是并发的。在这个阶段，虚拟机查找在执行并发标记阶段新进入老年代的对象（可能会有一些对象从新生代晋升到老年代，或者有一些对象被分配到老年代）。通过重新扫描，减少下一个阶段“重新标记”的工作，因为下一个阶段会Stop The World。

重新标记：这个阶段会暂停虚拟机，收集器扫描在CMS堆中剩余的对象。扫描从“根对象”开始向下追溯，并处理对象关联。

并发清理：清理垃圾对象，这个阶段收集器线程和应用程序线程并发执行

并发重置：这个阶段，重置CMS收集器的数据结构，等待下一次垃圾回收。

CMS的缺点：

- CMS采取的是“标记-清理”算法，不会整理空间，会产生空间碎片。
- 需要更多的CPU资源
- 需要更大的堆空间。CMS标记阶段应用程序的线程还是在执行的，那么就会有堆空间继续分配的情况，为了保证在CMS回收完堆之前还有空间分配给正在运行的应用程序，就必须预留一部分空间。

要点：

- 它的目标是：获取最短回收停顿时间为目标
- 使用“标记-清除”算法实现
- 回收老年代

（7）G1收集器

G1（Garbage First）收集器是当前收集器技术发展的最前沿成果，在JDK 1.6 Update14中提供了Early Access版本的G1收集器以供试用。

G1收集器较前面的CMS有两个显著的改进：一是G1收集器是基于"标记-整理"算法实现的收集器，也就是说它不会产生空间碎片。二是它可以非常精确地控制停顿，既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，这几乎已经是实时Java（RTSJ）的垃圾收集器的特征了。

G1将整个Java堆（包括新生代、老年代）划分为多个大小固定的独立区域（Region），并且跟踪这些区域里面的垃圾堆积程度，在后台维护一个优先列表，每次根据允许的收集时间，优先回收垃圾最多的区域（这就是Garbage First名称的来由）。

7.垃圾收集参数整理

表 3-1　垃圾收集相关的常用参数

参　数	描　述
UseSerialGC	虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial + Serial Old 的收集器组合进行内存回收
UseParNewGC	打开此开关后，使用 ParNew + Serial Old 的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用 ParNew + CMS + Serial Old 的收集器组合进行内存回收。Serial Old 收集器将作为 CMS 收集器出现 Concurrent Mode Failure 失败后的后备收集器使用
UseParallelGC	虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge + Serial Old（PS MarkSweep）的收集器组合进行内存回收
UseParallelOldGC	打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行内存回收
SurvivorRatio	新生代中 Eden 区域与 Survivor 区域的容量比值，默认为 8，代表 Eden：Survivor=8：1
PretenureSizeThreshold	直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
MaxTenuringThreshold	晋升到老年代的对象年龄。每个对象在坚持过一次 Minor GC 之后，年龄就加 1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	动态调整 Java 堆中各个区域的大小以及进入老年代的年龄
HandlePromotionFailure	是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个 Eden 和 Survivor 区的所有对象都存活的极端情况
ParallelGCThreads	设置并行 GC 时进行内存回收的线程数
GCTimeRatio	GC 时间占总时间的比率，默认值为 99，即允许 1% 的 GC 时间。仅在使用 Parallel Scavenge 收集器时生效
MaxGCPauseMillis	设置 GC 的最大停顿时间。仅在使用 Parallel Scavenge 收集器时生效
CMSInitiatingOccupancyFraction	设置 CMS 收集器在老年代空间被使用多少后触发垃圾收集。默认值为 68%，仅在使用 CMS 收集器时生效
UseCMSCompactAtFullCollection	设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用 CMS 收集器时生效
CMSFullGCsBeforeCompaction	设置 CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用 CMS 收集器时生效

8.内存分配和回收策略

（1）对象优先在Eden空间中分配

（2）大对象直接进入老年代

（3）长期存活的对象进入老年代

如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并将对象年龄设为1。对象在Survivor区中每熬过一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁）时，就会被晋升到老年代。可以通过-XX:MaxTenuringThreshold来设置。

> 新生代GC（Minor GC）：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕死的特性，所以Minor GC非常频繁，一般回收速度也比较快。
>
> 老年代GC（Major GC / Full GC）：指发生在老年代的GC，出现Major GC，经常会伴随至少一次的Minor GC（但非绝对的，在ParallelScavenge收集器的收集策略里就有直接进行Major GC的策略选择过程）。MajorGC的速度一般会比Major GC慢十倍以上。

（4）动态年龄判断

如果在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，年龄大于或等于该年龄的对象可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄。

（5）空间分配担保

在发生Minor GC时，虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则改为直接进行一次Full GC。如果小于，则查看HandlePromotionFailure设置是否允许担保失败；如果允许，那只会进行Minor GC；如果不允许，则也要改为进行一次Full GC。