一、运行时数据区域

结构：

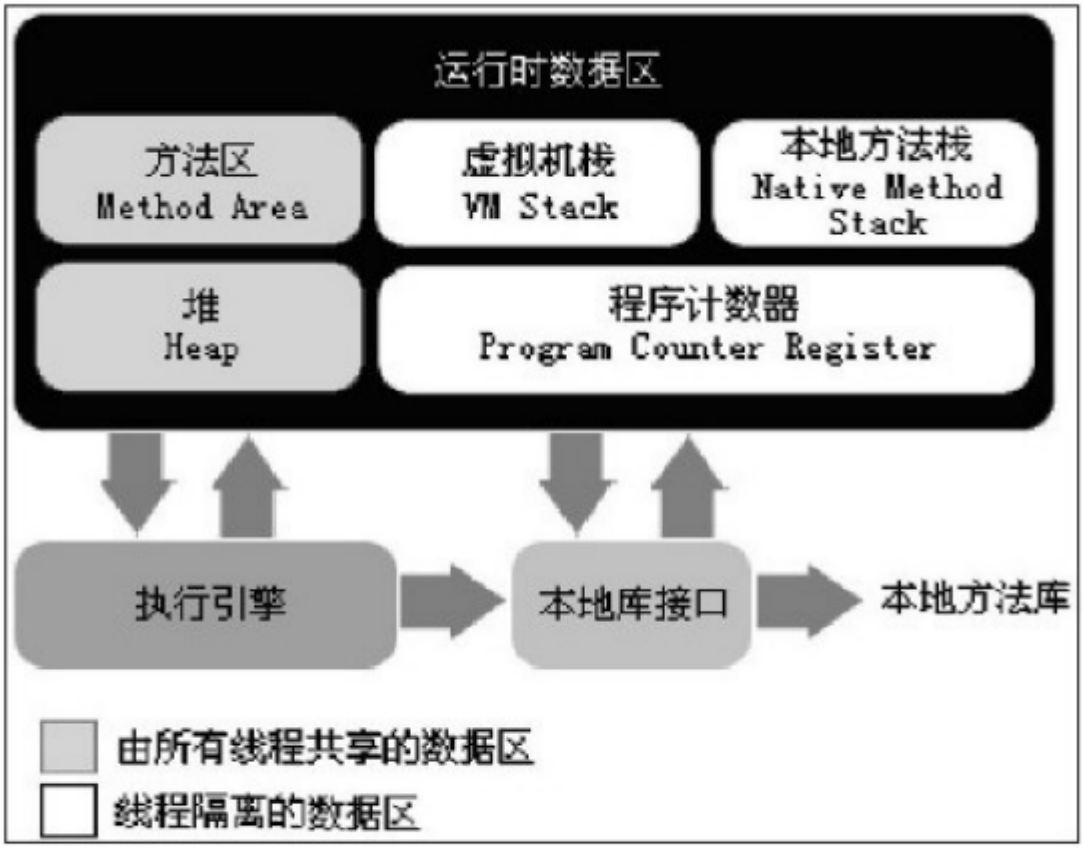


图　2-1　Java虚拟机运行时数据区

程序计数器

- 指向当前线程所执行的字节码指令
- 如果正在执行Native方法，则计数器为空
- 因为需要和线程一对一，所以是线程私有

Java 虚拟机栈

- 线程私有，生命周期与线程相同
- 描述Java 方法执行的内存模型：存放栈帧
- 栈帧：局部变量表、操作数栈、动态链接、方法出口
    1. 局部变量表：包含方法执行过程中的所有变量
    2. 操作数栈：入栈、出栈、复制、交换、产生消费变量
- 每一个方法从调用到执行完毕的过程对应一个栈帧在虚拟机栈中入栈到出栈的过程

- 异常有两种：
    1. 如果线程请求的栈深度大于虚拟机栈所允许的深度，抛出StackOverflowError，如递归层数过大
    2. 如果栈动态扩展时无法申请到足够的内存，抛出OutOfMemoryError

本地方法栈

与虚拟机栈类似，区别在于虚拟机栈为虚拟机执行Java 方法服务，而本地方法栈为虚拟机使用的Native 方法服务

Java 堆

- Java 堆被所有线程共享
- 几乎所有的对象实例都在这里分配内存
- 是垃圾收集器的主要工作区域
- 现代收集器都采用分代收集算法，Java 堆还分为：新生代、老年代
- 可扩展，通过-Xmx 和-Xms 控制

方法区

- 被所有线程共享
- 用于存储已被虚拟机加载的类信息、常量、静态变量等

运行时常量池

运行时常量池是方法去的一部分。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，用于存放编译器生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池存放。

直接内存

直接内存并不是虚拟机运行时数据区的一部分。 在JDK 1.4中新加入了NIO（New Input/Output）类，引入一种基于通道（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。在一些场景中可以显著提高性能，因为避免 了在Java 堆和Native 堆中来回复制数据。 直接内存会受到物理内存的限制。

对象分配内存

- 什么是符号引用？在说符号引用之前我们先来看看直接引用，直接引用是什么，比如就是你拥有你所需要数据的地址值，可以直接根据地址值获取到数据。但是 java语言是解释性的语言，然后由于总总原因（我也不知道对不对的原因）在某些时刻有些东西的直接地址还并不存在，是无法使用直接引用。这时候就可以用到符号引用了。
- 符号引用：符号引用是一个字符串，它给出了被引用的内容的名字并且可能会包含一些其他关于这个被引用项的信息——这些信息必须足以唯一的识别一个类、字段、方法。这样，对于其他类的符号引用必须给出类的全名。对于其他类的字段，必须给出类名、字段名以及字段描述符。对于其他类的方法的引用必须给出类名、方法名以及方法的描述符。这样我们就能根据符号引用锁定唯一的类，方法或字段了。
- 分配内存并不是线程安全的，有两种处理方法：

1. 采用CAS 配上失败重试的方式保证更新操作的原子性
2. 把内存分配的动作按照线程划分在不同的空间之中进行，每个线程在Java 堆中预先分配一小块内存，称为本地线程分配缓冲（TLAB），只有TLAB 用完b并分配新的TLAB 时，才需要同步锁定。

- 内存分配完成后，虚拟机需要将分配到的内存都初始化为零值。 上面的工作完成后，从虚拟机的视角看，一个新的对象已经产生了，但从Java 程序的视角看，对象创建才刚刚开始---方法还没有执行，所有的字段都还为

零。所以，一般来说，执行new 指令之后会接着执行方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完成。

对象的内存布局

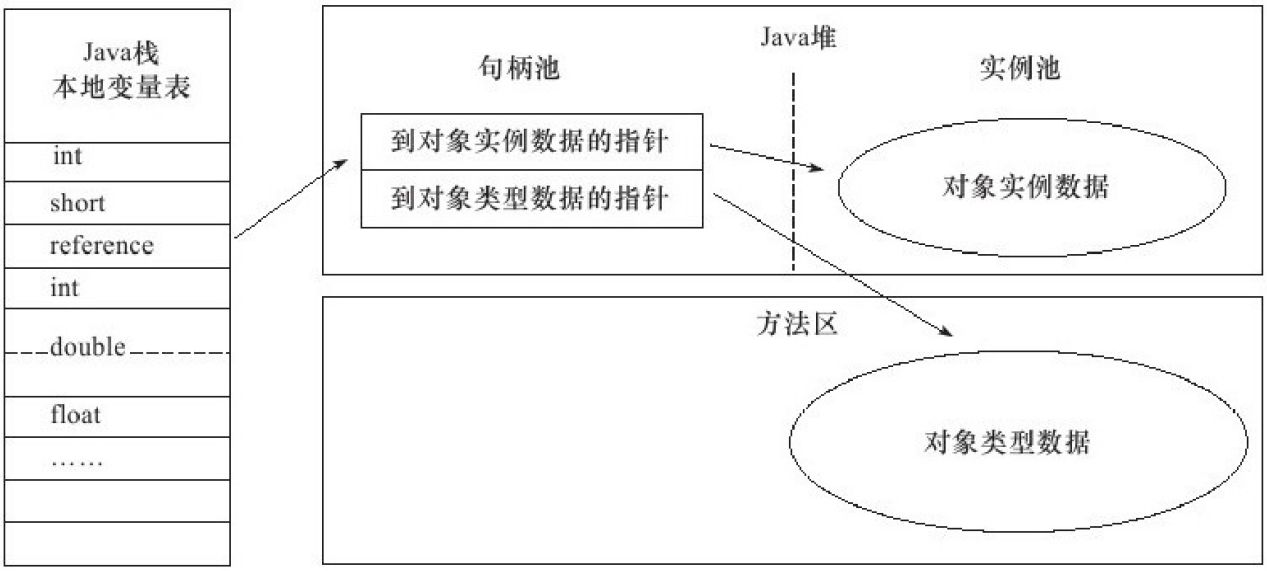
对象在内存中的布局分为3个区域：

1. 对象头（Header）
    - 存储对象自身的运行数据，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
    - 类型指针，对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例
2. 实例数据（Instance Data）
3. 对齐填充（Padding）

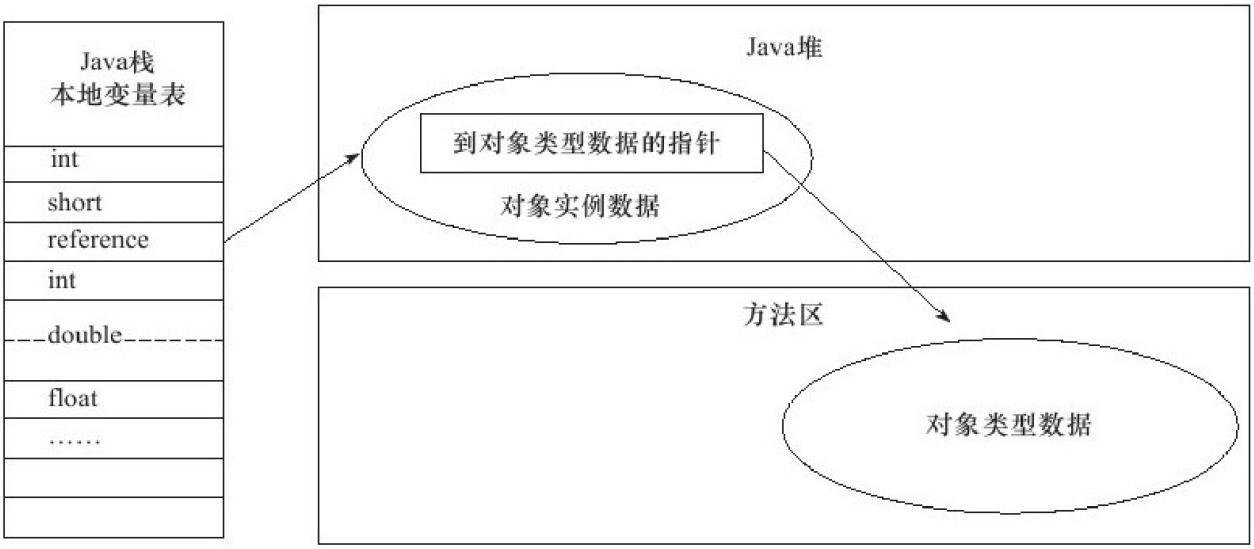
对象的访问定位

Java 程序通过栈上的reference 数据来操作堆上的具体对象。有句柄和直接指针两种。

1. 句柄，Java 堆中会划分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和对象类型数据。



2. 直接指针，reference 中存储的直接就是对象地址



String.intern() 是一个Native 方法，它的作用是：如果字符串常量池中已经包含一个等于此String 对象的字符串，则返回代表池中这个字符串的String 对象；否则，将此String 对象包含的字符串添加到常量池中，并且返回此String 对象的引用。

二、垃圾收集器

引用计数算法

给对象添加一个引用计数，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；任何时刻计数器值为0 的对象就是不可能再被使用的。

- 缺陷：很难解决对象间 的相互循环引用问题；如：objA.instance = objB; objB.instance = objA;

可达性分析算法

基本思路是通过一系列称为“GC Roots” 的对象作为起始点，从这些节点开始向下搜索，搜索所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，则证明此对象是不可用的。

引用

1. 强引用

类似“Object o = new Object()” 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。

2. 软引用

用来描述一些还有用但是并非必需的对象。对于软引用关联者的对象，在系统将要发生内存溢出之前，将会把这些对象列进回收范围进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。

3. 弱引用

用来描述非必需对象的，强度比软引用更弱，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。当垃圾回收器工作时，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。

4. 虚引用

是最弱的一种引用关系，为对象设置虚引用的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。强度依次减弱

回收方法区

在方法区中进行垃圾收集“性价比”一般比较低：在堆中，尤其在新生代中，常规应用进行一次垃圾收集一般可以回收70%~95%的空间，而永久代的垃圾收集效率远低于此。 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。

```
- 废弃常量的回收：如：如果字符串“abc”已经进入常量池，但是没有任何String 对象引用常量池中的“abc” 常量，也没有其他地方引用了这个字面量，如果这时发生内存回收，这个“abc”常量就会被系统清理出常量池。常量池中的其他类（接口）、方法、字段的符号引用也于此类似。
- 无用的类的回收：该类的所有实例都已经被回收、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。
```

垃圾收集算法

标记-清除算法（Mark - Sweep）

分为标记、清除两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

- 不足：一、效率低，标记和清除的效率都不高；二、空间问题，标记清除后产生大量不连续的内存碎片

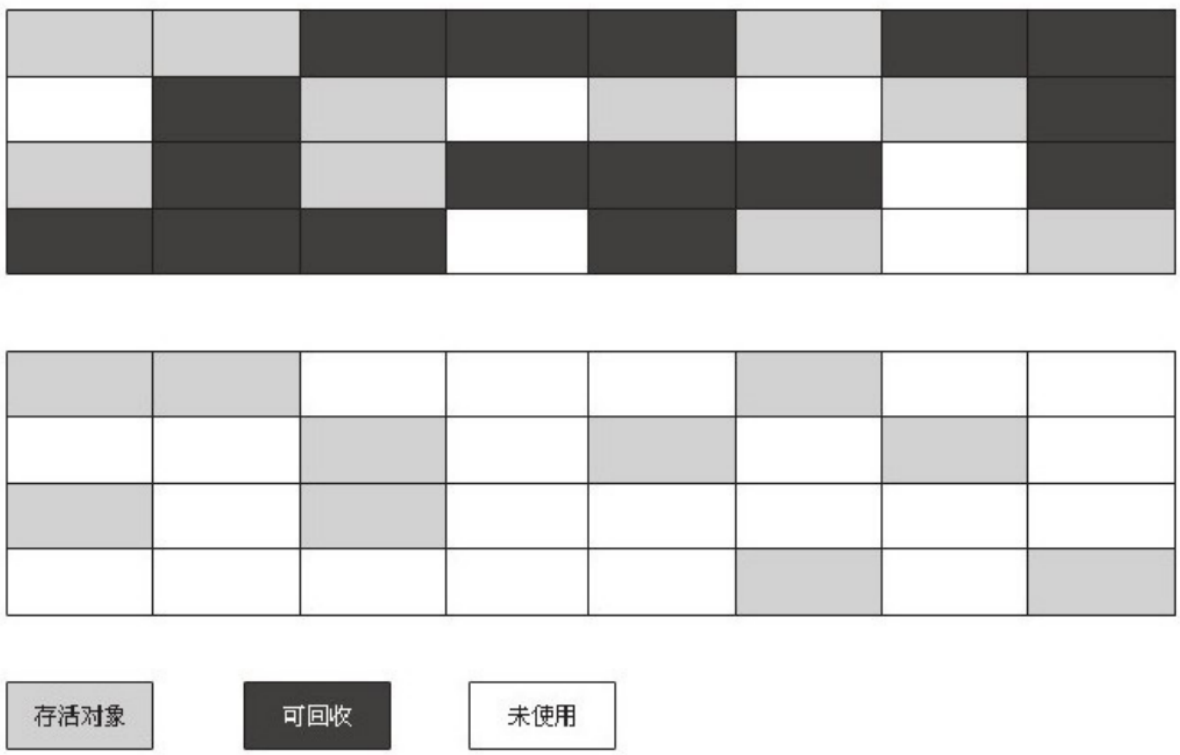
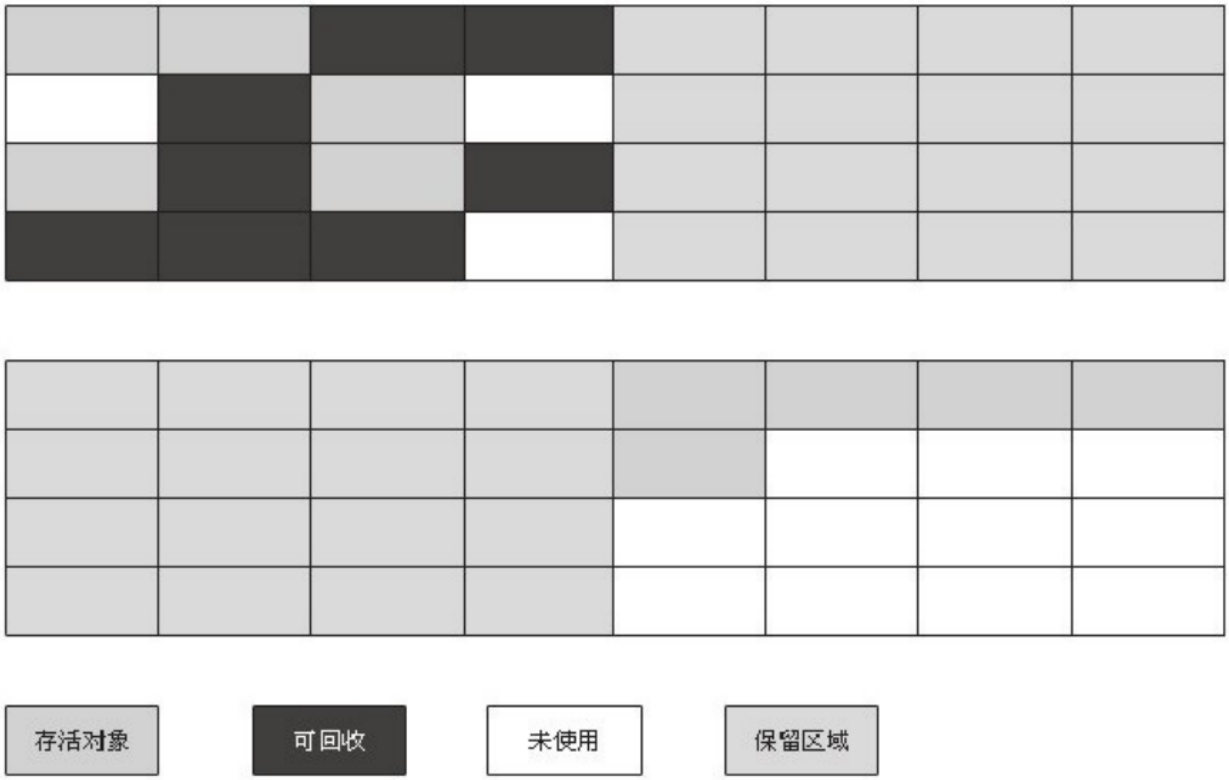


图 3-2 “标记-清除”算法示意图

复制算法（Copying）

将可用内存按容量划分为大小想等的两块，每次只使用其中的一块。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，然后把已使用过的内存空间 一次清理掉。

- 不足：将内存缩小为原来的一半

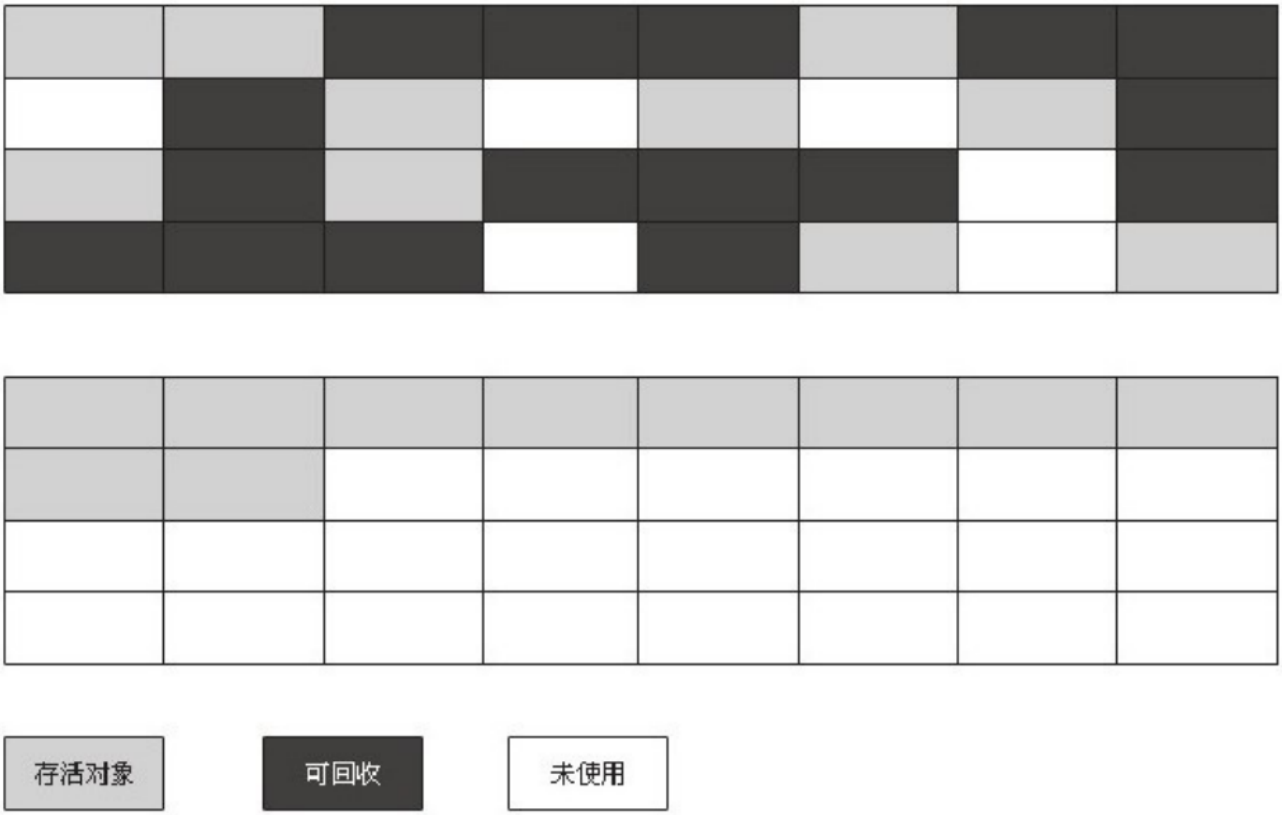


一般商用虚拟机的新生代都采用复制算法来回收。将内存分为较大的一块Eden空间和两块较小的Survivor 空间，每次使用Eden 和其中一块Survivor 。当回收时，将Eden 和Survivor 中还存活的对象一次性复制到另外一块Survivor 空间中，最后清理掉Eden 和刚才用过的Survivor 空间。

- 内存的分配担保：如果另外一块Survivor 没有足够空间存放上一次新生代收集下来的存活对象，这些对象将通过分配担保机制进行老年代。

标记-整理算法（Mark - Compact）

标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象清理，而是让所有存活对象都向一端移动，然后清理掉端边界以外的内存。



分代收集算法（Generational Collection）

当前商业虚拟机都采用“分代收集”，根据对象存活周期的不同将内存划分为几块。一般是把Java 堆分为新生代和老年代。新生代只有少量对象存活，采用复制算法；老年代对象存活率高，没有额外空间 进行担保分配，采用标记-清除或标记-整理算法。

对象如何晋身到老年代：

- 经历一定Minor 次数依然存活的对象
- Survivor 区中存放不下的对象

垃圾收集器

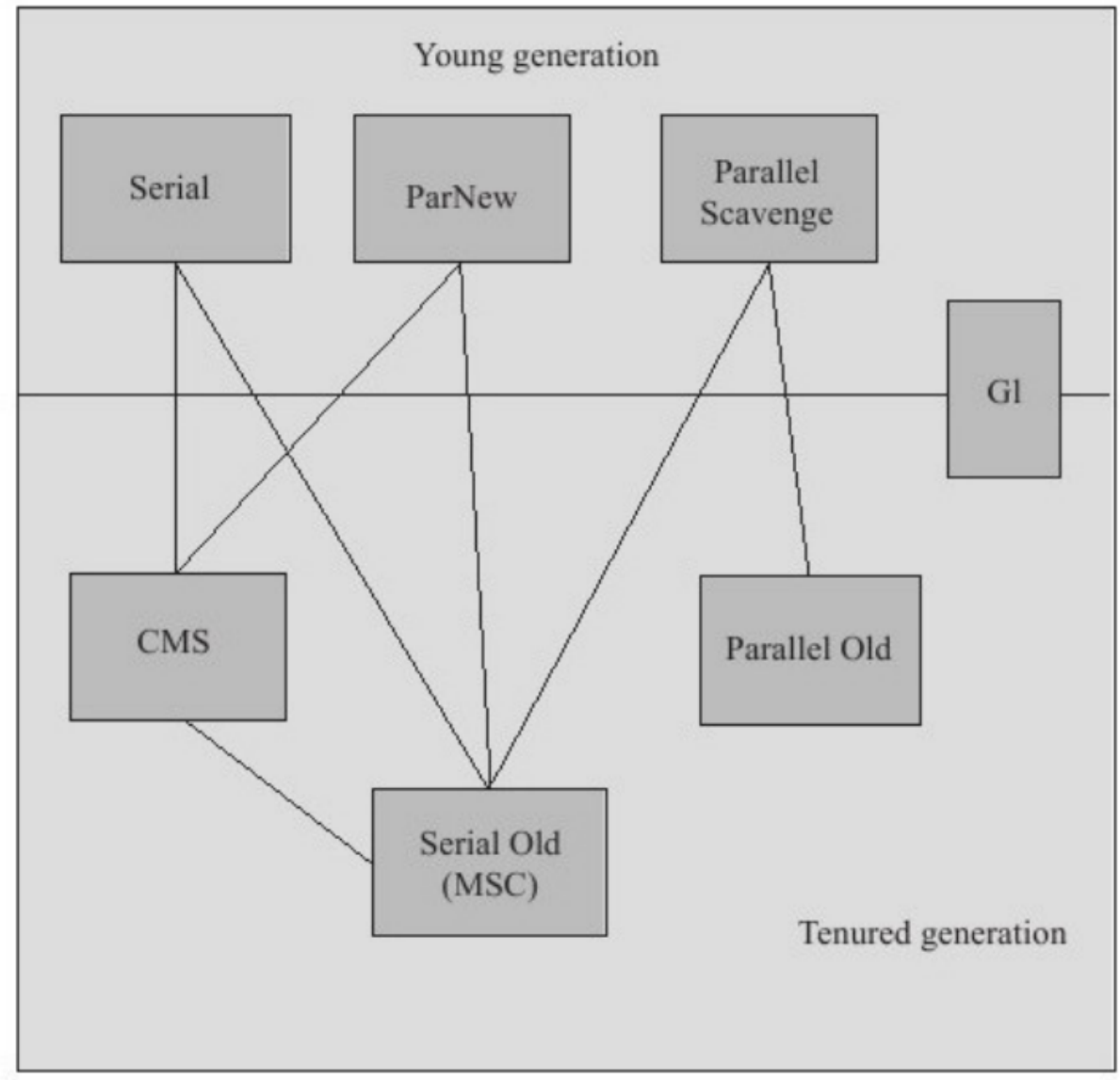


图　3-5　HotSpot虚拟机的垃圾收集器[1]

Serial 收集器

进行垃圾收集时，必须暂停其他所有的工作线程，直到收集结束。

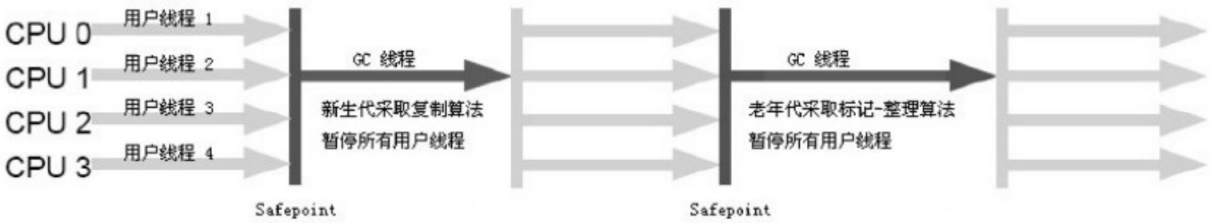


图　3-6　Serial/Serial Old收集器运行示意图

ParNew 收集器

ParNew 收集器是Serial 收集器的多线程版本。

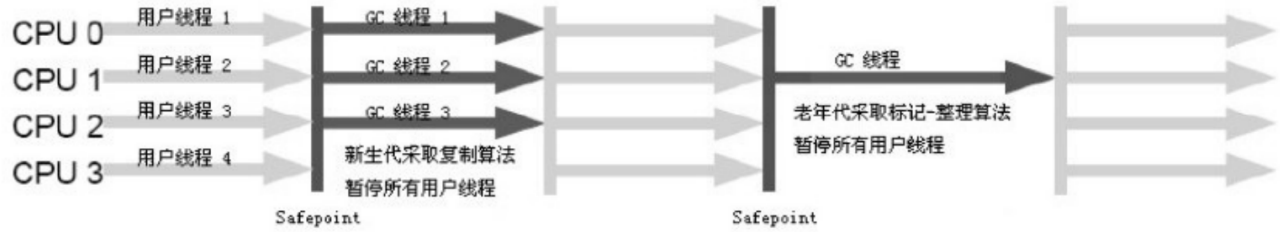


图　3-7　ParNew/Serial Old收集器运行示意图

Parallel Scavenge 收集器

Parallel Scavenge 收集器的特点是它的关注点与其他收集器不同，CMS 等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，而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 用于运行用户代码的时间与CPU 总消耗时间的比值。

Serial Old 收集器

是Serial 收集器的老年代版本。

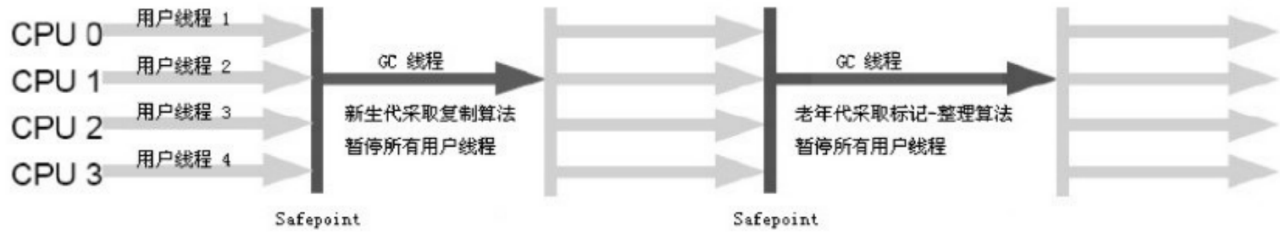


图　3-8　Serial/Serial Old收集器运行示意图

Parallel Old 收集器

是Parallel Scavenge 收集器的老年代版本

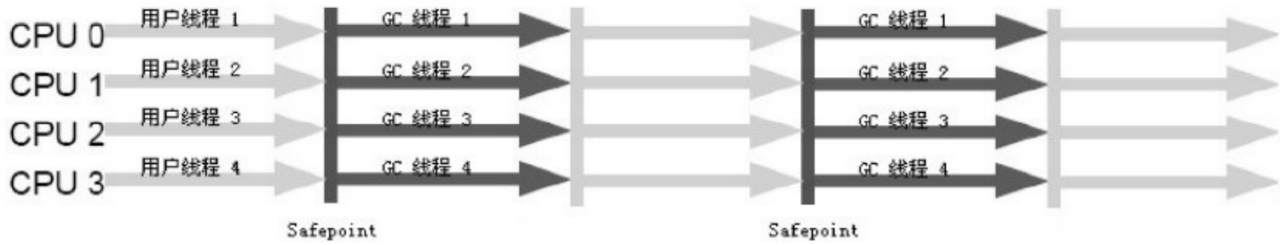


图　3-9　Parallel Scavenge/Parallel Old收集器运行示意图

CMS 收集器（Concurrent Mark Sweep）

是一种以获取最短回收停顿时间为目标的收集器，基于标记-清除算法实现，整个过程包含四个步骤：

1. 初始标记（CMS inital mark）

2. 并发标记（CMS concurrent mark）
3. 重新标记（CMS remark）
4. 并发清除（CMS concurrent sweep）

其中，初始标记、重新标记这两个步骤仍然需要“Stop The World”，初始标记只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC RootsTracing的过程，而重新标记阶段则是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，这一阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

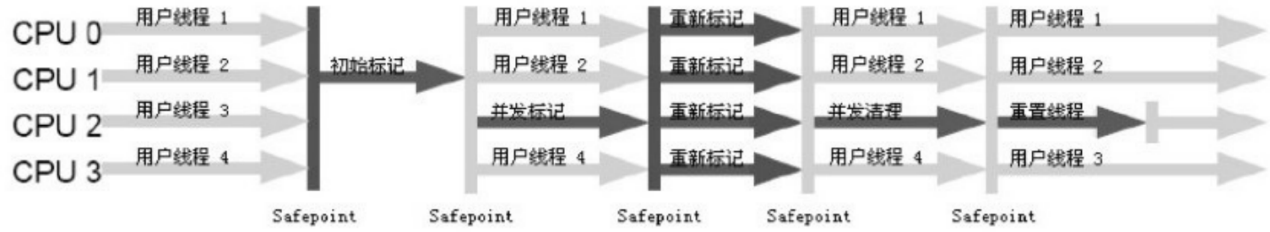


图　3-10　Concurrent Mark Sweep收集器运行示意图

G1 收集器（Garbage-First）

G1的特点：

- 并行与并发：充分利用多CPU、多核心的硬件优势，来缩短Stop The World 时间
- 分代收集：
- 空间整合：从整体看采用标记-整理算法，从局部看采用复制算法，意味着G1 运作期间不会产生空间碎片
- 可预测的停顿：

在G1 之前的收集器收集的范围都是整个新生代和老年代，而G1 不再是这样，G1 将整个Java 堆分成多个大小想等的 独立区域（Region），虽然也保留了新生代和老年代，但新生代和老年代不再是物理隔离的了。　　G1 之所以可以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j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，在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值大的Region。 步骤：

1. 初始标记
2. 并发标记
3. 最终标记
4. 筛选标记

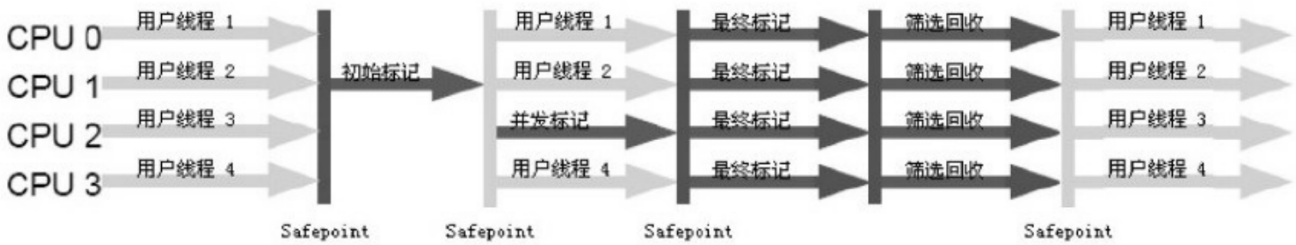


图　3-11　G1收集器运行示意图

虚拟机类加载机制

整个生命周期包括：

1. 加载

- 通过一个类的全限定名来获取定义此类的二进制字节流
- 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
- 在内存中生成一个代表这个类的java.lang.Class 对象，作为方法区这个类的各种数据的访问入口

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

- 为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配，分配的变量仅包括被static 修饰的变量，初始值是指零值。

4. 解析

- 将常量池中的符号引用转化为直接引用

5. 初始化

- 真正执行类中定义的Java 程序代码（字节码）
- 初始化阶段是执行类构造器()方法的过程
- ()方法是由编译器自动收集类中的所有类变量的赋值操作和静态语句块中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的。

6. 使用
7. 卸载

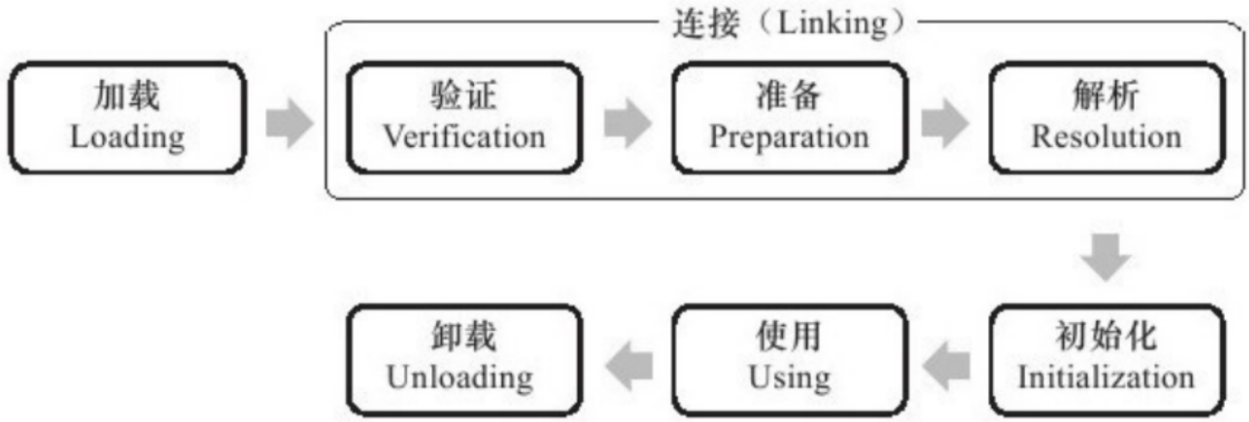


图　7-1　类的生命周期

类加载器

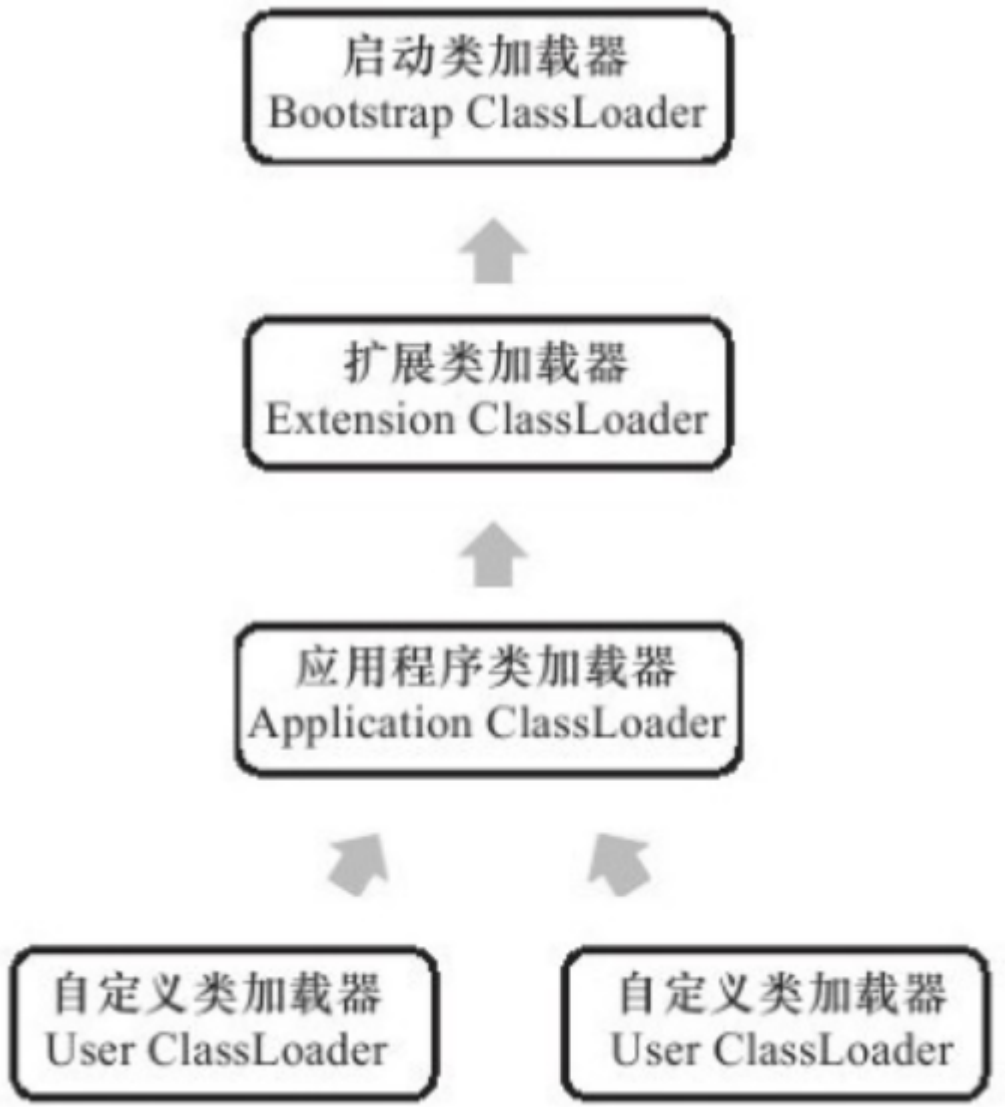
实现 “通过一个类的全限定名来获取描述此类的二进制字节流” 行为的代码模块称为“类加载器”

双亲委派模型

类加载器分为两种：

1. 启动类加载器
2. 所有的其他加载器

- 扩展类加载器
- 应用程序类加载器 ， 一般情况下是程序中默认的类加载器
- 自定义加载器



双亲委派模型的工作过程：如果一个类加载器收到了类加载请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器不能完成这个加载请求，子加载器才会自己尝试加载。

JAVA 内存模型与线程

volatile： 是JAVA 虚拟机提供的最轻量级的同步机制，有两种作用：

1. 保证此变量对所有线程的可见性，”可见性“指：当一条线程修改了这个变量的值，新值对于其他线程来说是立即得知的
2. 禁止指令重排序优化

只能保证可见性，没有原子性

参考

原文：https://blog.csdn.net/u014296316/article/details/83066436