链滴

Java 垃圾收集器与内存分配策略

作者：DongXiaokai0819

原文链接：https://ld246.com/article/1572929736453

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

垃圾收集器与内存分配策略

    Java内存运行时区域
各个部分，其中程序计数器、虚拟机栈、本地方法栈3个区域随线程而生，随线程而灭，栈中的栈帧
着方法的进入和退出而有条不紊地执行着出栈和入栈操作。

    每一个栈帧中分配多
内存基本上是在类结构确定下来就已知的，因此这几个区域的内存分配和回收都具备确定性，在这几
区域内就不需要过多的考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟着回收了。

    而Java堆和方法区的
存问题，我们只有在程序处于运行期间时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
的，垃圾收集器所关注的是这部分的内存。

1.如何判断对象已死?

    在堆里面存放着Java
界中几乎所有的对象实例，垃圾收集器在对堆进行回收前，第一件事情就是要确定这些对象之中哪些
“存活”着，哪些已经“死去”（即不可能再被任何途径使用的对象）。

1.1引用记数法

● 引用记数法就是给每个对象中添加一个引用计数器，每当有一个地方引用它，计数器就加 1；当引
失效，计数器就减 1；任何时候计数器为 0 的对象就是不可能再被使用的。

这个方法**实现简单**，**效率高**，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的
因是它很难解决对象之间**相互引用**的问题。

对象间的相互引用

所谓对象之间的相互引用问题，如下面代码所示：

除了对象 objA 和 objB 相互引用着对方之外，这两个对象之间再无任何引用。但是他们因为互相引
对方，导致它们的引用计数器都不为 0，于是引用计数算法无法通知 GC 回收器回收他们。

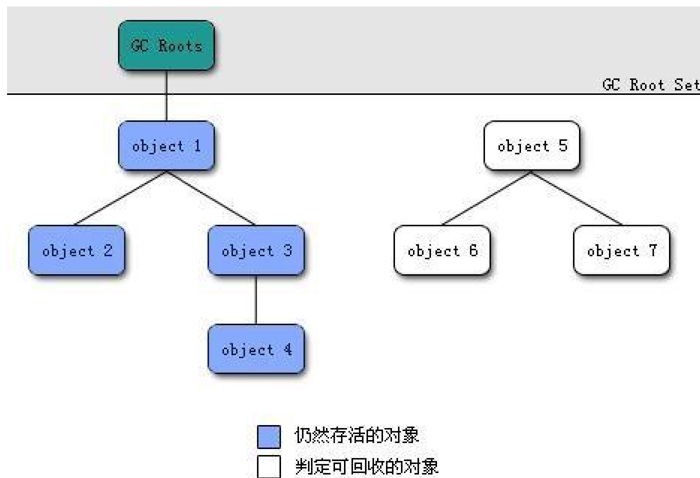
```
public class ReferenceCountingGc {
    Object instance = null;
    public static void main(String[] args) {
        ReferenceCountingGc objA = new ReferenceCountingGc();
        ReferenceCountingGc objB = new ReferenceCountingGc();
        objA.instance = objB;
        objB.instance = objA;
        objA = null;
        objB = null;

    }
}
```

1.2可达性分析算法

● 这个算法的基本思想就是通过一系列的称为 “ **GC Roots**” 的对象作为起点，从这些节点开始向
搜索，节点所走过的路径称为**引用链**，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此

原文链接：Java 垃圾收集器与内存分配策略

象是不可用的。



上图中的object5、6、7，没有任何的引用链与GC Roots相连，所以虚拟机会判定这些对象为不可达象（可回收对象）。

作为GC Roots的对象包括下面几种：

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象。
2. 方法区中类静态属性引用的对象。
3. 方法区中常量引用的对象。
4. 本地方法栈中JNI（即一般说的Native方法）引用的对象。

1.3对象间的引用

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的引用链是否可达判断对象是否存活都与“引用”有关。

Java1.2之前对引用的定义：如果 reference 类型的数据存储的数值代表的是另一块内存的起始地址就称这块内存代表一个引用。

JDK1.2 以后，Java 对引用的概念进行了扩充，将引用分为强引用、软引用、弱引用、虚引用四种（用强度逐渐减弱）

1.3.1强引用（StrongReference）

● 我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用，类似“Object obj = new Objet()”。

如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空间足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引的对象来解决内存不足问题。

1.3.2软引用（SoftReference）

● 软引用是用来描述一些还有用但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存出异常之前，将会把这些对象列进回收范围之中进行第二次回收，如果这次回收还没有足够的内存，

会抛出内存溢出异常。

软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，J
VA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

Jdk1.2之后，提供了SoftReference类来实现软引用。

```
User u = new User(1,"Test");
SoftReference<User> userSoft = new SoftReference<>(u);//SoftReference使用示例
u = null;//保证new User(1,"Test")这个实例只有userSoft在软引用
```

1.3.3弱引用（WeakReference)

● 弱引用也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生
道下一次垃圾收集之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联
对象。

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，J
va 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

Jdk1.2之后，提供了WeakReference类来实现弱引用。

软引用和弱引用的用法

● 软引用 SoftReference和弱引用 WeakReference，可以用在内存资源紧张的情况下以及创建不是
重要的数据缓存。当系统内存不足的时候，缓存中的内容是可以被释放的。

    例如，一个程序用来
理用户提供的图片。如果将所有图片读入内存，这样虽然可以很快的打开图片，但内存空间使用巨大
一些使用较少的图片浪费内存空间，需要手动从内存中移除。如果每次打开图片都从磁盘文件中读取
内存再显示出来，虽然内存占用较少，但一些经常使用的图片每次打开都要访问磁盘，代价巨大。这
时候就可以用软引用构建缓存。

1.3.4虚引用（PhantomReference)

● "虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。
果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

为一个对象设置虚引用的关联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。虚引
主要用来跟踪对象被垃圾回收的活动。

Jdk1.2之后，提供了PhantomReference类来实现虚引用。

虚引用与软引用和弱引用的区别

● 虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果
现它还有虚引用，就会在回收对象的内存之前，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。程

如果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行动。

特别注意，在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为软引用可加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等题的产生。

1.4不可达的对象并非“非死不可”

    即使在可达性分析法
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亡，至少要经历两次标记过程；可达性分析法中不可达的对象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 finalize 方法已经被拟机调用过时，虚拟机将这两种情况视为没有必要执行。

任何一个对象的finalize()方法都只会被系统自动调用一次。

    由于finalize()方法运
代价高昂，不确定性大，无法保证各个对象的调用顺序，应**尽量避免使用finalize()方法。**

    被判定为需要执行的
象将会被放在一个队列中进行第二次标记，除非这个对象与引用链上的任何一个对象建立关联，否则会被真的回收。

1.5回收方法区

回收废弃常量

假如在常量池中存在字符串 "abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池。

如何判断一个类是无用的类

    判定一个常量是否是
废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足面 3 个条件才能算是 “无用的类” ：

1. 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
2. 加载该类的 ClassLoader 已经被回收。
3. 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的方
。

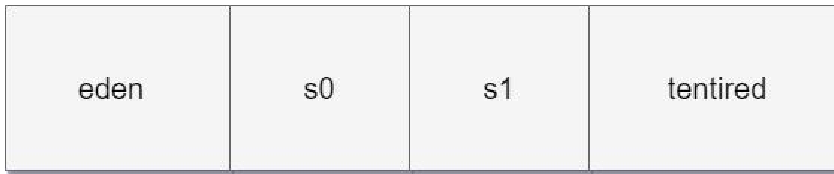
虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样使用了就会必然被回收。

2.Java内存分配与回收策略

    java技术体系中所提
的自动内存管理最终可以归结为自动化的解决了俩个问题：**给对象分配内存**以及**回收分配给对象的内存。**

对象的内存分配，往大方向讲，就是堆上分配。

Java堆空间的基本结构：



上图所示的 eden 区、s0("From") 区、s1("To") 区都属于新生代，tentired 区属于老年代。

    大部分情况，对象都
首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s1("To")，并且对象年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 5 岁），就会被晋升到老年代中。

2.1分配策略

对象优先在 eden 区分配

● 大多数情况下，对象在新生代中 eden 区分配。当 eden 区没有足够空间进行分配时，虚拟机将发一次 Minor GC。

Minor GC 和 Full GC

● 新生代 GC（Minor GC）:指发生新生代的的垃圾收集动作，Minor GC 非常频繁，回收速度一般也较快。

● 老年代 GC（Major GC/Full GC）:指发生在老年代的 GC，出现了 Major GC 经常会伴随至少一次的 Minor GC（并非绝对），Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。

下面我们来看一个测试：

```
private static final int SIZE = 1024*1024;//1M

public static void main(String[] args) {
    byte[] allocation1,allocation2,allocation3,allocation4;
    allocation1 = new byte[2*SIZE];
    allocation2 = new byte[2*SIZE];
    allocation3 = new byte[2*SIZE];
    allocation4 = new byte[2*SIZE];//GC出现
}
```

    上述main方法中，尝
分配3个2MB大小和一个4MB大小的对象，在运行时通过Xms20M -Xmx20M -Xmn10M 三个参数限制堆大小为20M，新生代大小为10M，剩下的10M给老年代。-XX:SurvivorRatio=8决定了新生代Eden区与一个Survivor区的空间比例是8:1。

    main方法中在分配第
个allocation4对象的语句时会发生一次MinorGC，发生原因为Eden区已经被占用了6M（即使程序么也不做，新生代也会使用 2000 多 k 内存），剩余空间已经不足以分配allocation4所需要的内存，C期间，虚拟机又发现已有的3个2MB大小的对象全部无法放入Survivor空间，所以只好通过空间担保制，将新生代对象提前转移到老年代去。

原文链接：Java 垃圾收集器与内存分配策略

    这里发生的 Minor GC
和 Full GC ，由于我是用的jdk1.8，跟书上不太一样。。。。我也不太懂，待以后再试试。

    我觉得可能是，发生M
norGC后，由于无法将对象放入Survivor空间，所以通过空间担保机制，将新生代对象提前转移到了
年代，但是又由于转移进来的对象占了老年代很大的空间，达到了一个阈值，所以触发了FullGC。

输出如下：

```
[GC (Allocation Failure) [PSYoungGen: 7803K->808K(9216K)] 7803K->6960K(19456K), 0.0050
83 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Ergonomics) [PSYoungGen: 808K->0K(9216K)] [ParOldGen: 6152K->6727K(10240K)]
6960K->6727K(19456K), [Metaspace: 2931K->2931K(1056768K)], 0.0073063 secs] [Times: use
=0.00 sys=0.00, real=0.01 secs]
Heap
 PSYoungGen      total 9216K, used 2212K [0x00000000ff600000, 0x0000000100000000, 0x00
0000100000000)
  eden space 8192K, 27% used [0x00000000ff600000,0x00000000ff8290f0,0x00000000ffe0000
)
  from space 1024K, 0% used [0x00000000ffe00000,0x00000000ffe00000,0x00000000fff00000)
  to   space 1024K, 0% used [0x00000000fff00000,0x00000000fff00000,0x0000000100000000)
 ParOldGen       total 10240K, used 6727K [0x00000000fec00000, 0x00000000ff600000, 0x000
0000ff600000)
  object space 10240K, 65% used [0x00000000fec00000,0x00000000ff291d38,0x00000000ff60
000)
 Metaspace       used 2937K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 318K, capacity 388K, committed 512K, reserved 1048576K
```

大对象直接进入老年代

- 大对象直接进入老年代

大对象：需要大量连续内存空间的Java对象，比如很长的字符串和大型数组

    最好不要弄大对象（
当避免使用大对象），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
的连续空间来“安置”它们。

-XX:PretenureSizeThreshold 参数， 大于这个数量直接在老年代分配，缺省为0 ，表示绝不会直接
配在老年代。

长期存活的对象将进入老年代

    既然虚拟机采用了分
收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应放在新生代，哪些对象应放在老年
中。为了做到这一点，虚拟机给每个对象一个对象年龄（Age）计数器。

    如果对象在 Eden 出
并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间
，并将对象年龄设为 1.对象在 Survivor 中每熬过一次 MinorGC,年龄就增加 1 岁，当它的年龄增加
一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 -
X:MaxTenuringThreshold 来设置。

2.2动态对象年龄判定

    为了更好的适应不同
序的内存情况，虚拟机不是永远要求对象年龄必须达到了某个值才能进入老年代，如果 Survivor 空
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就可以直
进入老年代，无需达到要求的年龄。

2.3空间分配担保

    新生代中有大量的对
存活，survivor空间不够，当出现大量对象在MinorGC后仍然存活的情况（最极端的情况就是内存回
后新生代中所有对象都存活），就需要老年代进行分配担保，把Survivor无法容纳的对象直接进入老
代.只要老年代的连续空间大于新生代对象的总大小或者历次晋升的平均大小，就进行Minor GC，否则
ullGC。

3垃圾收集算法

3.1标记-清除算法

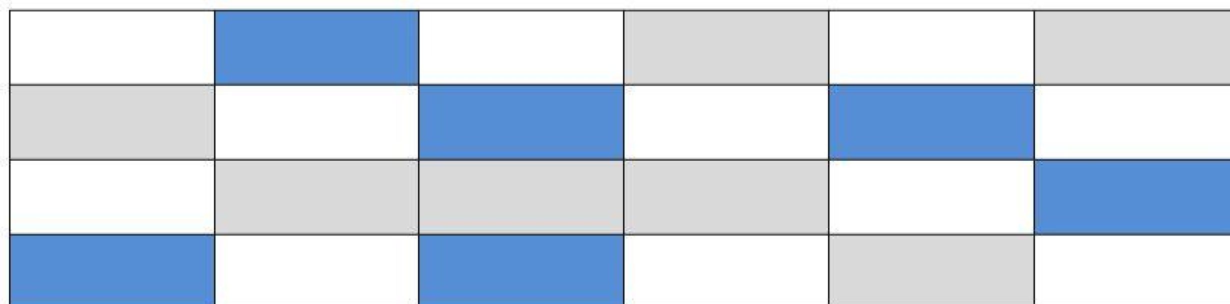
该算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有
标记的对象。

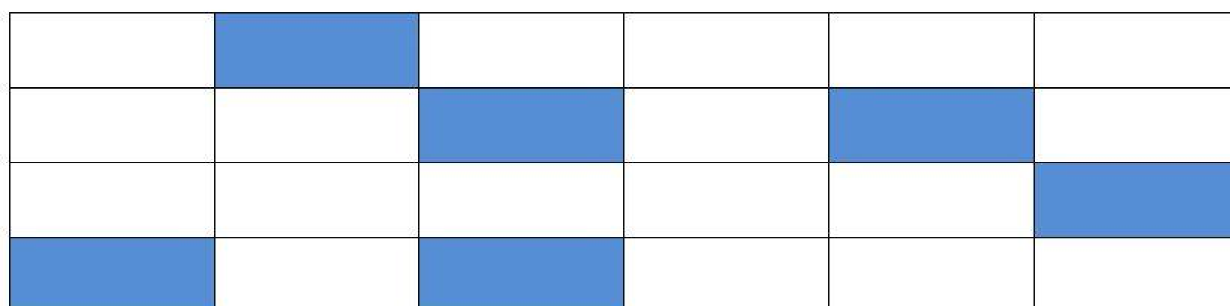
它是最基础的收集算法，后续的算法都是对其不足进行改进得到。这种垃圾收集算法会带来两个明显
问题：

1. 效率问题:标记和清除俩个过程的效率都不高

2. 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以后在程序运行
程中需要分配大对象时，无法找到足够的连续内存，而不得不提前触发另一次垃圾收集动作。

标记-清除算法的执行过程如下图所示：





3.2复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉这样就使每次的内存回收都是对内存区间的一半进行回收，内存分配时也就不用考虑内存碎片等复杂况。

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半，太过于浪费。

复制算法的执行过程如下图所示：

内存整理前

内存整理后

可用内存	可回收内存	存活对象	保留内存

    现在的商业虚拟机都用这种收集算法来回收新生代，研究表明，新生代中的对象98%都是“朝生夕死”的，所以不需要按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空间和俩块较小的Survivor空间，每次用Eden和其中一块Survivor空间。

    当回收时，将Eden和Srvivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survvor空间。

3.3标记-整理算法

根据老年代的特点提出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

3.4把算法们都用上--分代收集算法

    当前虚拟机的垃圾收都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。**般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。**

为什么要分为新生代和老年代?

● 在新生代中，每次垃圾收集时都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象复制成本就可以完成每次垃圾收集。

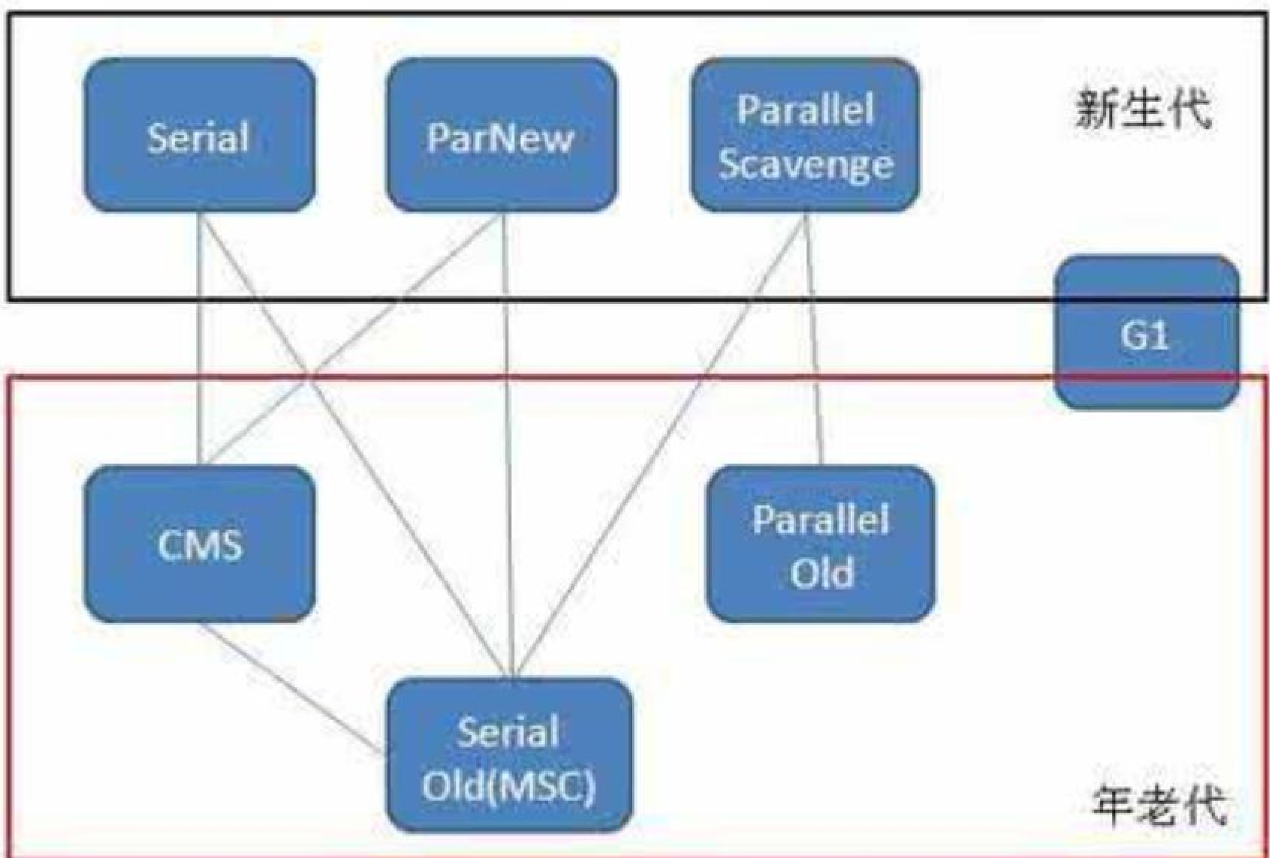
● 而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择标记-清除”或“标记-整理”算法进行垃圾收集。

4垃圾收集器

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

    虽然我们对各个收集
进行比较，但并非要挑选出一个最好的收集器。因为直到现在为止还没有最好的垃圾收集器出现，更没有万能的垃圾收集器，我们能做的就是根据具体应用场景选择适合自己的垃圾收集器。试想一下：果有一种四海之内、任何场景下都适用的完美收集器存在，那么我们的 HotSpot 虚拟机就不会实现么多不同的垃圾收集器了。

分代收集以及各种收集器对应关系：



收集器列表：

收集器	收集对象和算法	收集器类型	说明	适用场景
Serial	新生代，复制算法	单线程	进行垃圾收集时，必须暂停所有工作线程，直到完成；(stop the world)	简单高效；适合内存不大的情况；
ParNew	新生代，复制算法	并行的多线程收集器	ParNew垃圾收集器是Serial收集器的多线程版本	搭配CMS垃圾回收器的首选
Parallel Scavenge 吞吐量优先收集器	新生代，复制算法	并行的多线程收集器	类似ParNew，更加关注吞吐量，达到一个可控制的吞吐量；	本身是Server级别多CPU机器上的默认GC方式，主要适合后台运算不需要太多交互的任务；
Serial Old	老年代，标记整理算法	单线程	jdk7/8默认的老生代垃圾回收器	Client模式下虚拟机使用
Parallel Old	老年代，标记整理算法	并行的多线程收集器	Parallel Scavenge收集器的老年代版本，为了配合Parallel Scavenge的面向吞吐量的特性而开发的对应组合；	在注重吞吐量以及CPU资源敏感的场合采用
CMS	老年代，标记清除算法	并行与并发收集器	尽可能的缩短垃圾收集时用户线程停止时间；缺点在于： 1.内存碎片 2.需要更多cpu资源 3.浮动垃圾问题，需要更大的堆空间	重视服务的响应速度、系统停顿时间和用户体验的互联网网站或者B/S系统。互联网后端目前cms是主流的垃圾回收器；
G1	跨新生代和老年代；标记整理 + 化整为零	并行与并发收集器	JDK1.7才正式引入，采用分区回收的思维，基本不牺牲吞吐量的前提下完成低停顿的内存回收；可预测的停顿是其最大的优势；	面向服务端应用的垃圾回收器，目标为取代CMS

注：

吞吐量 = 运行用户代码时间/(运行用户代码时间 + 垃圾收集时间)

垃圾收集时间 = 垃圾收集频率 * 单次垃圾回收时间

jvm中的并发与并行

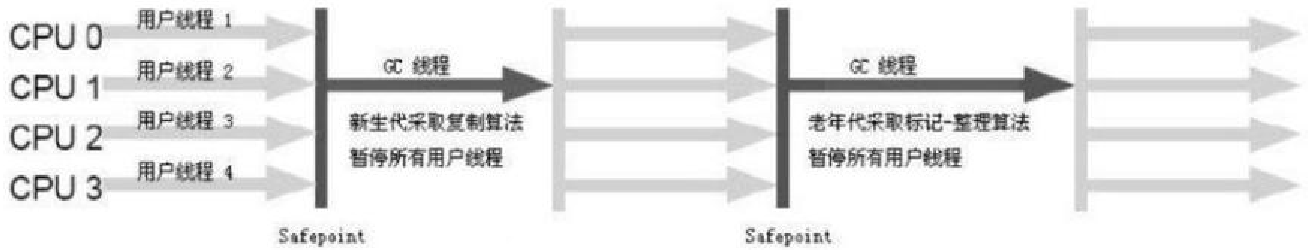
并发与并行都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，可以解释如下：

- 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，可能会交替执行）用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。

原文链接：Java 垃圾收集器与内存分配策略

4.1Serial & Serial Old

Serial & Serial Old 收集器运行示意图



新生代采用复制算法，老年代采用标记-整理算法。

Serial 收集器

    Serial（串行）收集器
集器是最基本、历史最悠久的垃圾收集器了。大家看名字就知道这个收集器是一个单线程收集器了。
的 “单线程” 的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是
在进行垃圾收集工作的时候必须暂停其他所有的工作线程（ "Stop The World" ），直到它收集结束。

    虚拟机的设计者们当
知道 Stop The World 带来的不良用户体验，所以在后续的垃圾收集器设计中停顿时间在不断缩短（
然还有停顿，寻找最优秀的垃圾收集器的过程仍然在继续）。

- Serial 收集器简单而高效（与其他收集器的单线程相比）。Serial 收集器由于没有线程交互的开销自然可以获得很高的单线程收集效率。Serial 收集器对于运行在 Client 模式下的虚拟机来说是个不错选择。

Serial Old收集器

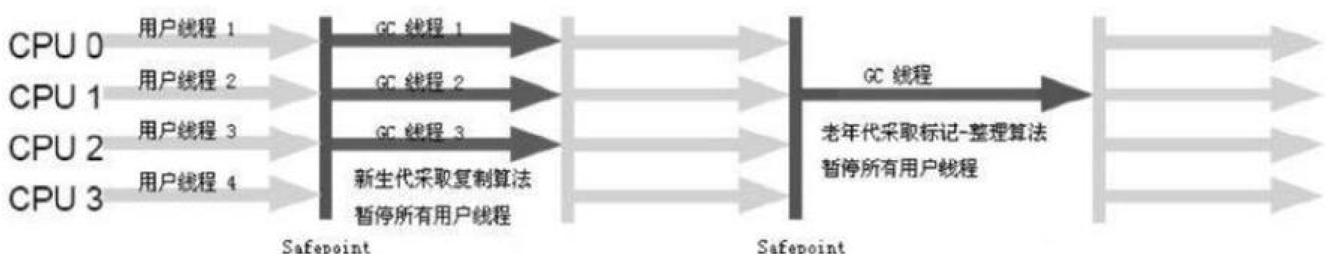
    Serial 收集器的老年
版本，它同样是一个单线程收集器。它主要有两大用途：一种用途是在 JDK1.5 以及以前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途是作为 CMS 收集器的后备方案。

参数

- -XX:+UseSerialGC 新生代和老年代都用串行收集器
- -XX:+UseParNewGC 新生代使用ParNew，老年代使用Serial Old
- -XX:+UseParallelGC 新生代使用ParallerGC，老年代使用Serial Old

4.2ParNew 收集器

ParNew & Serial Old 收集器运行示意图



原文链接：Java 垃圾收集器与内存分配策略

新生代采用复制算法，老年代采用标记-整理算法。

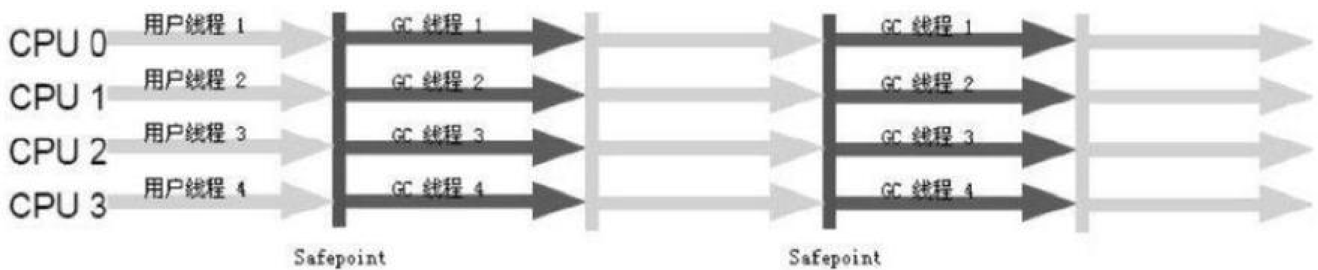
    ParNew 收集器其实
是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、收策略等等）和 Serial 收集器完全一样。

    它是许多运行在 Server
模式下的虚拟机的首要选择，除了 Serial 收集器外，只有它能与 CMS 收集器（真正意义上的并发收器，后面会介绍到）配合工作。

参数:

- -XX:+UseParNewGC 新生代使用ParNew，老年代使用Serial Old

4.3Parallel Scavenge & Parallel Old



新生代采用复制算法，老年代采用标记-整理算法。

Parallel Old 收集器

    Parallel Scavenge 收
器的老年代版本。使用多线程和“标记-整理”算法。在注重吞吐量以及 CPU 资源的场合，都可以优考虑 Parallel Scavenge 收集器和 Parallel Old 收集器。

Parallel Scavenge收集器

    Parallel Scavenge 收
器也是使用复制算法的多线程收集器，它看上去几乎和ParNew都一样。

    Parallel Scavenge 收
器关注点是吞吐量（高效率的利用 CPU）。CMS 等垃圾收集器的关注点更多的是用户线程的停顿时（提高用户体验）。所谓吞吐量就是 CPU 中用于运行用户代码的时间与 CPU 总消耗时间的比值。 Paallel Scavenge 收集器提供了很多参数供用户找到最合适的停顿时间或最大吞吐量，如果对于收集器作不太了解的话，手工优化存在困难的话可以选择把内存管理优化交给虚拟机去完成也是一个不错的择。

参数:

- -XX:+UseParallerOldGC：新生代使用ParallerGC，老年代使用Parallel Old

- -XX:MaxGCPauseMills ：参数允许的值是一个大于0的毫秒数，收集器将尽可能地保证内存回收费的时间不超过设定值。不过大家不要认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾集速度变得更快，GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些收集300MB新生代肯定比收集500MB快吧，这也直接导致垃圾收集发生得更频繁一些，原来10秒收

一次、每次停顿100毫秒，现在变成5秒收集一次、每次停顿70毫秒。停顿时间的确在下降，但吞吐
也降下来了。

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

● -XX:+UseAdaptiveSizePolicy 当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、
den与Survivor区的比例（-XX：SurvivorRatio）、晋升老年代对象年龄（-XX：PretenureSizeThres
old）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
最合适的停顿时间或者最大的吞吐量，这种调节方式称为GC自适应的调节策略。

    如果对于收集器运作
来不太了解，手工优化存在困难的时候，使用Parallel Scavenge收集器配合自适应调节策略，把内存
理的调优任务交给虚拟机去完成将是一个不错的选择。只需要把基本的内存数据设置好（如-Xmx设
最大堆），然后使用MaxGCPauseMillis参数（更关注最大停顿时间）或GCTimeRatio（更关注吞吐
）参数给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。自适应调节策略
是Parallel Scavenge收集器与ParNew收集器的一个重要区别。

4.4CMS 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。它非常
合在注重用户体验的应用上使用。

    CMS（Concurrent M
rk Sweep）收集器是 HotSpot 虚拟机第一款真正意义上的**并发收集器**，它第一次实现了让垃圾收集
程与用户线程（基本上）同时工作。

    从名字中的Mark Swe
p这两个词可以看出，CMS 收集器是一种 “**标记-清除**”算法实现的，它的运作过程相比于前面几种
圾收集器来说更加复杂一些。整个过程分为四个步骤：

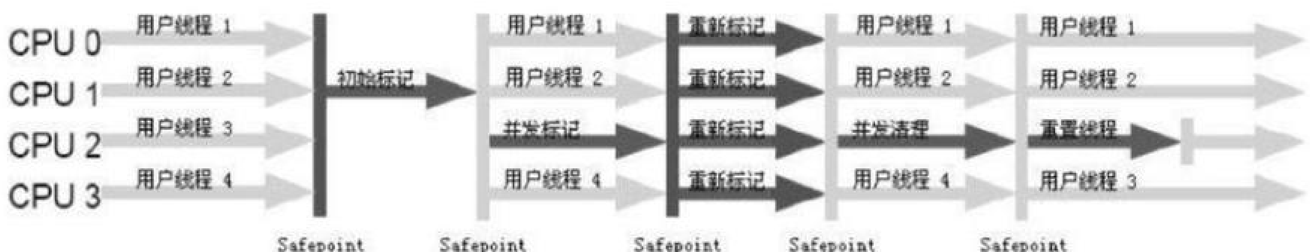
1. 初始标记(STW)： 暂停所有的其他线程，并记录下直接与 root 相连的对象，速度很快；

2. 并发标记： 同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这
闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线
无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。

3. 重新标记(STW)： 重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生
动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发
记阶段时间短

4. 并发清除： 开启用户线程，同时 GC 线程开始对为标记的区域做清扫。

Concurrent Mark Sweep 收集器运行示意图：



CMS主要优点：**并发收集、低停顿。**

但是它有下面三个明显的缺点：

1. 对 CPU 资源敏感；
2. 无法处理浮动垃圾；
3. 它使用的回收算法-“标记-清除”算法会导致收集结束时会有大量空间碎片产生。

参数：

- -XX:+UseConcMarkSweepGC，表示新生代使用ParNew，老年代的用CMS

4.5G1 收集器

G1 (Garbage-First) 是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征.

被视为 JDK1.7 中 HotSpot 虚拟机的一个重要进化特征。它具备一下特点：

1. 并行与并发：G1 能充分利用 CPU、多核环境下的硬件优势，使用多个 CPU（CPU 或者 CPU 核心来缩短 Stop-The-World 停顿时间。部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 集器仍然可以通过并发的方式让 java 程序继续执行。
2. 分代收集：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，但是还是保留了分代的念。
3. 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”算法实现的集器；从局部上来看是基于“复制”算法实现的。
4. 可预测的停顿：这是 G1 相对于 CMS 的另一个大优势，降低停顿时间是 G1 和 CMS 共同的关注，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。

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，它们都是部分Region（不需要连续）的集合。

G1的内存布局

    G1收集器之所以能建
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堆中进行全区域地垃圾收集。

    G1 收集器在后台维护
一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbge-First 的由来)。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 收集器有限时间内可以尽可能高的收集效率（把内存化整为零）。

在G1收集器中，Region之间地对象引用以及其他收集器中地新生代与老年代之间地对象引用，虚拟都是使用Remembered Set 来避免全栈扫描的。

G1中每个Region都有一个与之对应的Remembered Set，虚拟机发现程序在堆Reference类型的数进行写操作时，会产生一个Write Barrier暂时中断写操作，检查Reference引用的对象是否处于不用Region之中，如果是，便通过CardTable 把相关引用信息记录到被引用对象所属的Region的Rememered Set之中。

当进行内存回收时，在GC根节点的枚举范围中加入Remembered Set 即可保证不对全栈扫描也不会遗漏。

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：

1. 初始标记（STW）：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，好让下一阶段的用户程序并发运行时，能在正确可用的Region中创建新对象耗时很短。
2. 并发标记：从GC Roots开始对堆中对象进行可达性分析，找出存活的对象，耗时较长，但可与用程序并发执行。
3. 最终标记（STW）：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记记录，可并行执行。
4. 筛选回收：先对各个Region的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收划，可以与用户程序并发执行。

G1的执行步骤：

参数：

- -XX:+UseG1GC

垃圾收集常用参数总结

原文链接：Java 垃圾收集器与内存分配策略

参数	描述
UseSerialGC	虚拟机运行在Client模式下的默认，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 + Srial Old的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用PaNew+ 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Concrrent Mode Failure失败后的后备收集器使用
UseParallelGC	虚拟机运行在Server模式下的默值，打开此开关后，使用Parallel Scavenge + Serial Old (PS Mark Sweep)的收集器组合进行内存收
UserParallelOldGC	打开此开关后，使用Parallelcavenge + 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。每个对象在坚持过一次Minor GC之后，年龄就增加1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	动态调整Java堆中各个域的大小以及进入老年代的年龄
HandlePromotionFailure	是否允许分配担保失，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时进行内存回收线程数
GCTimeRatio	GC时间占总时间的比率，默认值是9， 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。仅使用Parallel Scavenge收集器时生效
CMSInitiatingOccupancyFraction	设置CMS收器在老年代时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

参考

《深入理解Java虚拟机》

Java Guide：https://snailclimb.top/JavaGuide/#/?id=java