1.JVM内存分配与回收

1.1 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，虚拟机将发起一次Minor GC。我们来进行实际测试一下。

在测试之前我们先来看看 **Minor GC和Full GC 有什么不同呢?**

- **Minor GC/Young GC**：指发生新生代的的垃圾收集动作，Minor GC非常频繁，回收速度一般也比较快。
- **Major GC/Full GC**：一般会回收老年代，年轻代，方法区的垃圾，Major GC的速度一般会比Minor GC的慢10倍以上。

示例：

```
//添加运行JVM参数：  -XX:+PrintGCDetails
public class GCTest {
 public static void main(String[] args) throws InterruptedException {
 byte[] allocation1, allocation2/*, allocation3, allocation4, allocation5, allocation6*/;
 allocation1 = new byte[60000*1024];

 //allocation2 = new byte[8000*1024];

 /*allocation3 = new byte[1000*1024];
  allocation4 = new byte[1000*1024];
  allocation5 = new byte[1000*1024];
  allocation6 = new byte[1000*1024];*/
  }
}

运行结果：
Heap
  PSYoungGen total 76288K, used 65536K [0x000000076b400000, 0x0000000770900000, 0x00000007c0000000)
  eden space 65536K, 100% used [0x000000076b400000,0x000000076f400000,0x000000076f400000)
  from space 10752K, 0% used [0x000000076fe80000,0x000000076fe80000,0x0000000770900000)
  to space 10752K, 0% used [0x000000076f400000,0x000000076f400000,0x000000076fe80000)
```

```
 ParOldGen total 175104K, used 0K [0x00000006c1c00000, 0x00000006cc700000, 0x000000076b400000)
 object space 175104K, 0% used [0x00000006c1c00000,0x00000006c1c00000,0x00000006cc700000)
 Metaspace used 3342K, capacity 4496K, committed 4864K, reserved 1056768K
 class space used 361K, capacity 388K, committed 512K, reserved 1048576K
```

我们可以看出eden区内存几乎已经被分配完全（即使程序什么也不做，新生代也会使用至少几M内存）。**假如我们再为allocation2分配内存会出现什么情况呢?**

```
//添加运行JVM参数:  -XX:+PrintGCDetails
public class GCTest {
 public static void main(String[] args) throws InterruptedException {
 byte[] allocation1, allocation2/*, allocation3, allocation4, allocation5, allocation6*/;
 allocation1 = new byte[60000*1024];

 allocation2 = new byte[8000*1024];

 /*allocation3 = new byte[1000*1024];
 allocation4 = new byte[1000*1024];
 allocation5 = new byte[1000*1024];
 allocation6 = new byte[1000*1024];*/
 }
}

运行结果:
[GC (Allocation Failure) [PSYoungGen: 65253K->936K(76288K)] 65253K->60944K(251392K), 0.0279083 secs] [Times: user=0.13 sys=0.02, real=0.03 secs]
Heap
 PSYoungGen total 76288K, used 9591K [0x000000076b400000, 0x0000000774900000, 0x00000007c0000000)
 eden space 65536K, 13% used [0x000000076b400000,0x000000076bc73ef8,0x000000076f400000)
 from space 10752K, 8% used [0x000000076f400000,0x000000076f4ea020,0x000000076fe80000)
 to space 10752K, 0% used [0x0000000773e80000,0x0000000773e80000,0x0000000774900000)
 ParOldGen total 175104K, used 60008K [0x00000006c1c00000, 0x00000006cc700000, 0x000000076b400000)
```

```
  object space 175104K, 34% used [0x00000006c1c00000,0x00000006c569a010,0x00000006cc700000)
  Metaspace used 3342K, capacity 4496K, committed 4864K, reserved 1056768K
  class space used 361K, capacity 388K, committed 512K, reserved 1048576K
```

简单解释一下为什么会出现这种情况： 因为给allocation2分配内存的时候eden区内存几乎已经被分配完了，我们刚刚讲了当Eden区没有足够空间进行分配时，虚拟机将发起一次Minor GC，GC期间虚拟机又发现allocation1无法存入Survior空间，所以只好把新生代的对象提前转移到老年代中去，老年代上的空间足够存放allocation1，所以不会出现Full GC。执行Minor GC后，后面分配的对象如果能够存在eden区的话，还是会在eden区分配内存。可以执行如下代码验证：

```
public class GCTest {
 public static void main(String[] args) throws InterruptedException {
 byte[] allocation1, allocation2, allocation3, allocation4, allocation5, allocation6;
 allocation1 = new byte[60000*1024];

 allocation2 = new byte[8000*1024];

 allocation3 = new byte[1000*1024];
 allocation4 = new byte[1000*1024];
  allocation5 = new byte[1000*1024];
  allocation6 = new byte[1000*1024];
  }
 }

 运行结果:
 [GC (Allocation Failure) [PSYoungGen: 65253K->952K(76288K)] 65253K->60960K(251392K), 0.0311467 secs] [Times: user=0.08 sys=0.02, real=0.03 secs]
 Heap
  PSYoungGen total 76288K, used 13878K [0x000000076b400000, 0x0000000774900000, 0x00000007c0000000)
  eden space 65536K, 19% used [0x000000076b400000,0x000000076c09fb68,0x0000000076f400000)
  from space 10752K, 8% used [0x000000076f400000,0x000000076f4ee030,0x0000000076fe80000)
```

```
21   to space 10752K, 0% used [0x0000000773e80000,0x0000000773e80000,0x00000
00774900000)
22   ParOldGen total 175104K, used 60008K [0x00000006c1c00000, 0x00000006cc7
00000, 0x000000076b400000)
23   object space 175104K, 34% used [0x00000006c1c00000,0x00000006c569a010,0
x00000006cc700000)
24   Metaspace used 3343K, capacity 4496K, committed 4864K, reserved
1056768K
25   class space used 361K, capacity 388K, committed 512K, reserved 1048576K
```

1.2 大对象直接进入老年代

大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。JVM参数-XX:PretenureSizeThreshold可以设置大对象的大小，如果对象超过设置大小会直接进入老年代，不会进入年轻代

为什么要这样呢?

为了避免为大对象分配内存时的复制操作而降低效率。

1.3 长期存活的对象将进入老年代

既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别那些对象应放在新生代，那些对象应放在老年代中。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 出生并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，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，当它的年龄增加到一定程度（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 `-XX:MaxTenuringThreshold` 来设置。

1.4 对象动态年龄判断

当前放对象的Survivor区域里(其中一块区域，放对象的那块s区)，一批对象的总大小大于这块Survivor区域内存大小的50%，那么此时**大于等于**这批对象年龄最大值的对象，就可以直接进入老年代了，例如Survivor区域里现在有一批对象，年龄1+年龄2+年龄n的多个年龄对象总和超过了Survivor区域的50%，此时就

会把年龄n以上的对象都放入老年代。这个规则其实是希望那些可能是长期存活的对象，尽早进入老年代。对象动态年龄判断机制一般是在minor gc之后触发的。

1.5 Minor gc后存活的对象Survivor区放不下

这种情况会把存活的对象部分挪到老年代，部分可能还会放在Survivor区

1.6 老年代空间分配担保机制

年轻代每次minor gc，之前JVM都会计算下老年代**剩余可用空间**
如果这个可用空间小于年轻代里现有的所有对象大小之和(**包括垃圾对象**)
就会看一个“-XX:-HandlePromotionFailure”(jdk1.8默认就设置了)的参数是否设置了
如果有这个参数，就会看看老年代的可用内存大小，是否大于之前每一次minor gc后进入老年代的对象的**平均大小**。
如果上一步结果是小于或者之前说的参数没有设置，那么就是触发一次Full gc，对老年代和年轻代一起回收一次垃圾，如果回收完还是没有足够空间存放新的对象就会发生"OOM"

当然，如果minor gc之后剩余的存活对象大小还是大于老年代可用空间，那么也会触发full gc，full gc完之后如果还是没用空间放minor gc之后的存活对象，则也会发生“OOM”

1.7 Eden与Survivor区默认8:1:1

大量的对象被分配在eden区，eden区满了后会触发minor gc，可能会有99%以上的对象成为垃圾被回收掉，剩余存活的对象会被挪到为空的那块survivor区，下一次eden区满了后又会触发minor gc，把eden区和survivor去垃圾对象回收，把剩余存活的对象一次性挪动到另外一块为空的survivor区，因为新生代的对象都是朝生夕死的，存活时间很短，所以JVM默认的8:1:1的比例是很合适的，**让eden区尽量的大，survivor区够用即可**

JVM默认有这个参数-XX:+UseAdaptiveSizePolicy，会导致这个比例自动变化，如果不想这个比例有变化可以设置参数-XX:-UseAdaptiveSizePolicy

2.如何判断对象可以被回收

堆中几乎放着所有的对象实例，对堆垃圾回收前的第一步就是要判断那些对象已经死亡（即不能再被任何途径使用的对象）。

2.1 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，计数器就加1；当引用失效，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，其最主要的原因是它很难解决对象之间相互循环引用的问题。 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用对方，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。

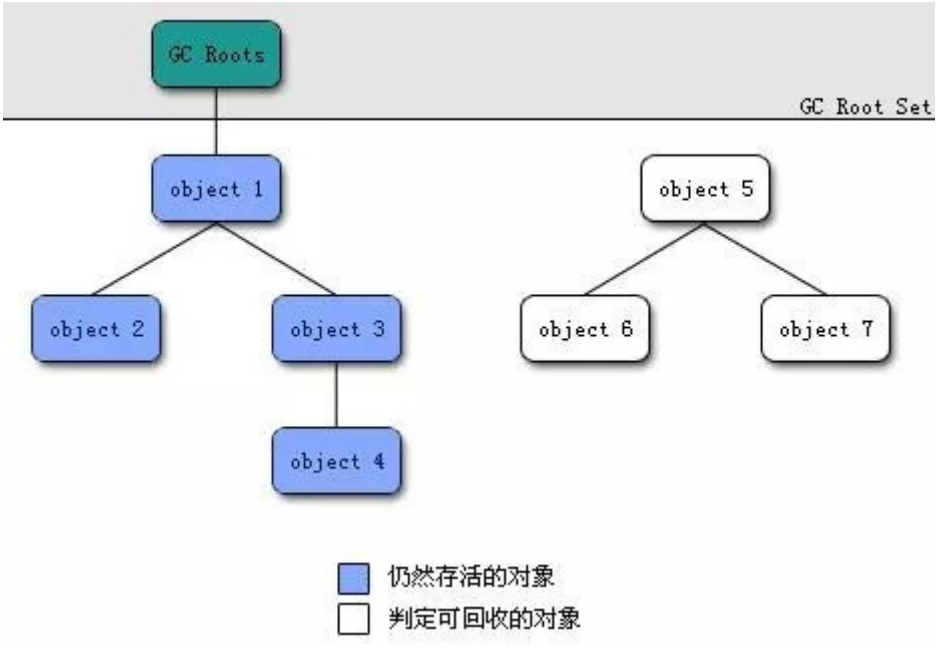
```
public class ReferenceCountingGc {
 Object instance = null;

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

2.2 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，从这些节点开始向下搜索，找到的对象都标记为非垃圾对象，其余未标记的对象都是垃圾对象

GC Roots根节点：线程栈的本地变量、静态变量、本地方法栈的变量等等



2.3 常见引用类型

java的引用类型一般分为四种：**强引用**、**软引用**、弱引用、虚引用

强引用：普通的变量引用

```
1 public static User user = new User();
```

软引用：将对象用SoftReference软引用类型的对象包裹，正常情况不会被回收，但是GC做完后发现释放不出空间存放新的对象，则会把这些软引用的对象回收掉。**软引用可用来实现内存敏感的高速缓存。**

```
1 public static SoftReference<User> user = new SoftReference<User>(new
User());
```

软引用在实际中有重要的应用，例如浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。
（1）如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建
（2）如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出

弱引用：将对象用WeakReference软引用类型的对象包裹，弱引用跟没引用差不多，GC会直接回收掉，很少用

```
1 public static WeakReference<User> user = new WeakReference<User>(new
User());
```

2.4 finalize()方法最终判定对象是否存活

即使在可达性分析算法中不可达的对象，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，至少要经历再次标记过程。

标记的前提是对象在进行可达性分析后发现没有与GC Roots相连接的引用链。

1. 第一次标记并进行一次筛选。

筛选的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，对象将直接被回收。

2. 第二次标记

如果这个对象覆盖了finalize方法，finalize方法是对象脱逃死亡命运的最后一次机会，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的一个对象建立关联即可，譬如把自己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移除出“即将回收”的集合。如果对象这时候还没逃脱，那基本上它就真的被回收了。

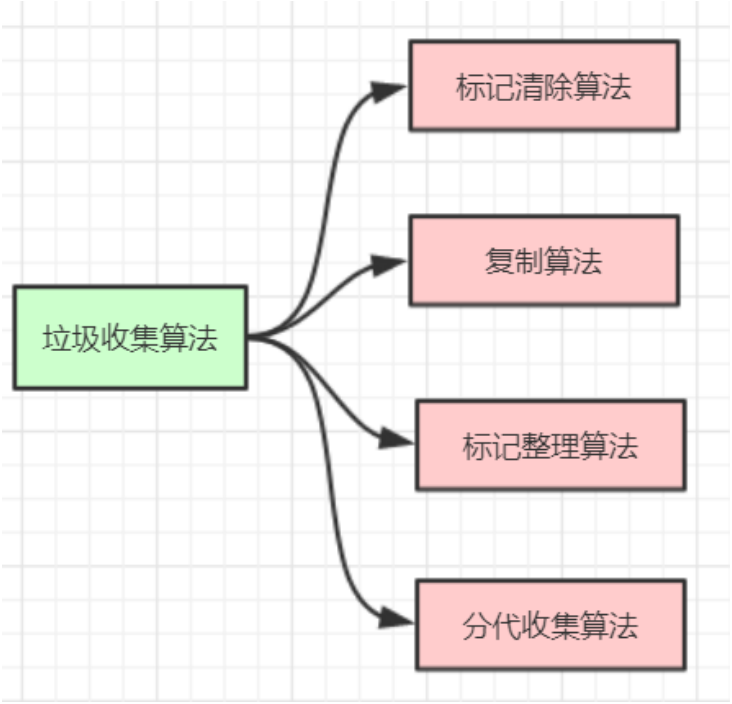
2.5 如何判断一个类是无用的类

方法区主要回收的是无用的类，那么如何判断一个类是无用的类的呢?

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，也就是 Java 堆中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的方法。

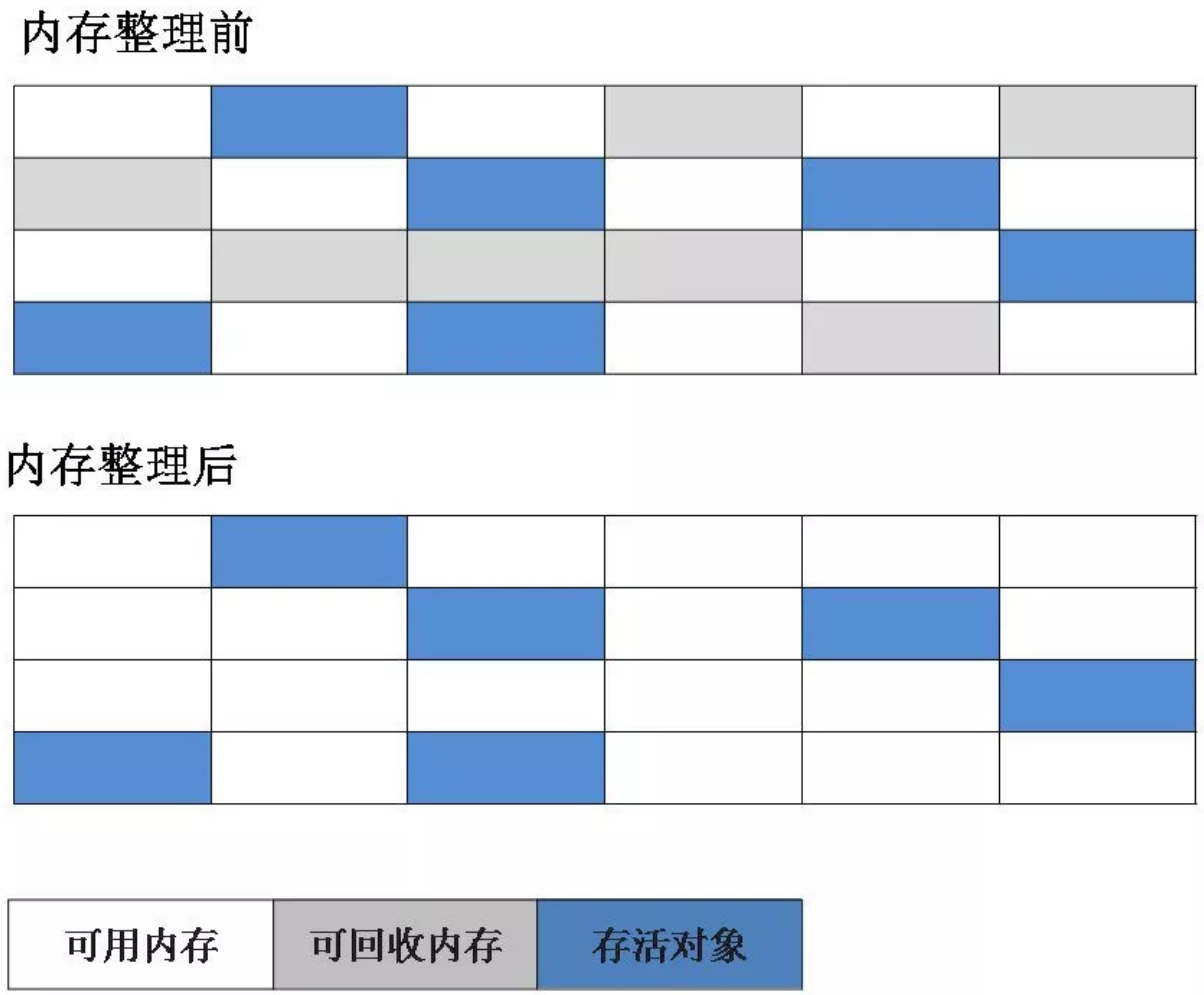
3.垃圾收集算法



3.1 标记-清除算法

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被标记的对象。它是最基础的收集算法，效率也很高，但是会带来两个明显的问题：

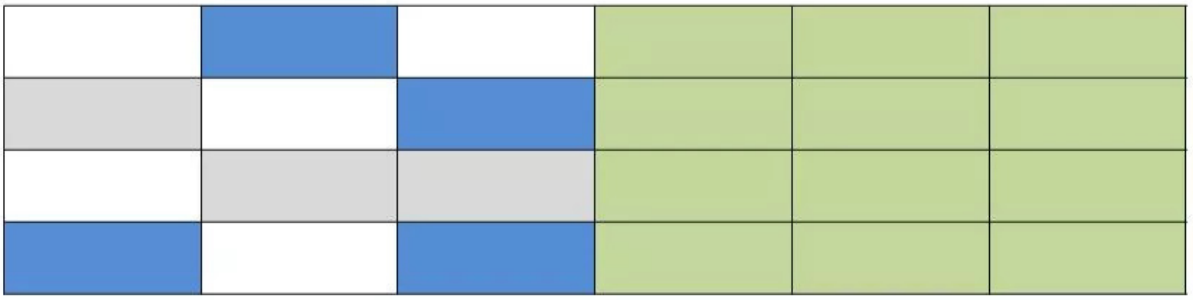
1. **效率问题**
2. **空间问题（标记清除后会产生大量不连续的碎片）**



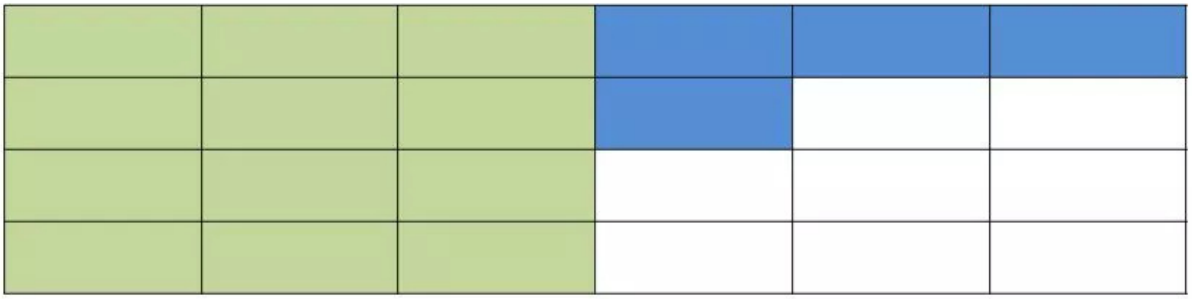
3.2 复制算法

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，每次使用其中的一块。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



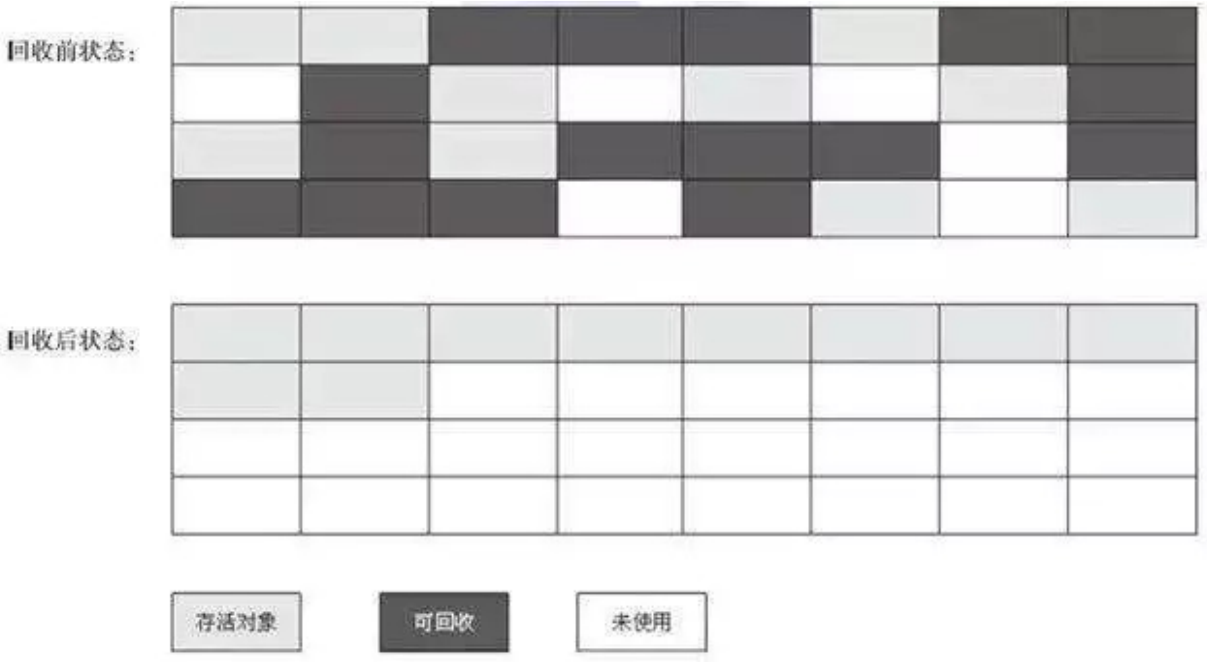
内存整理后



可用内存	可回收内存	存活对象	保留内存

3.3 标记-整理算法

根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一段移动，然后直接清理掉端边界以外的内存。



3.4 分代收集算法

当前虚拟机的垃圾收集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。一般将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。

比如在新生代中，每次收集都会有大量对象(近99%)死去，所以可以选择复制算法，只需要付出少量对象的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。注意，“标记-清除”或“标记-整理”算法会比复制算法慢10倍以上

通过上面这些内容介绍，大家应该对JVM优化有些概念了，就是尽可能让对象都在新生代里分配和回收，尽量别让太多对象频繁进入老年代，避免频繁对老年代进行垃圾回收，同时给系统充足的内存大小，避免新生代频繁的进行垃圾回收。