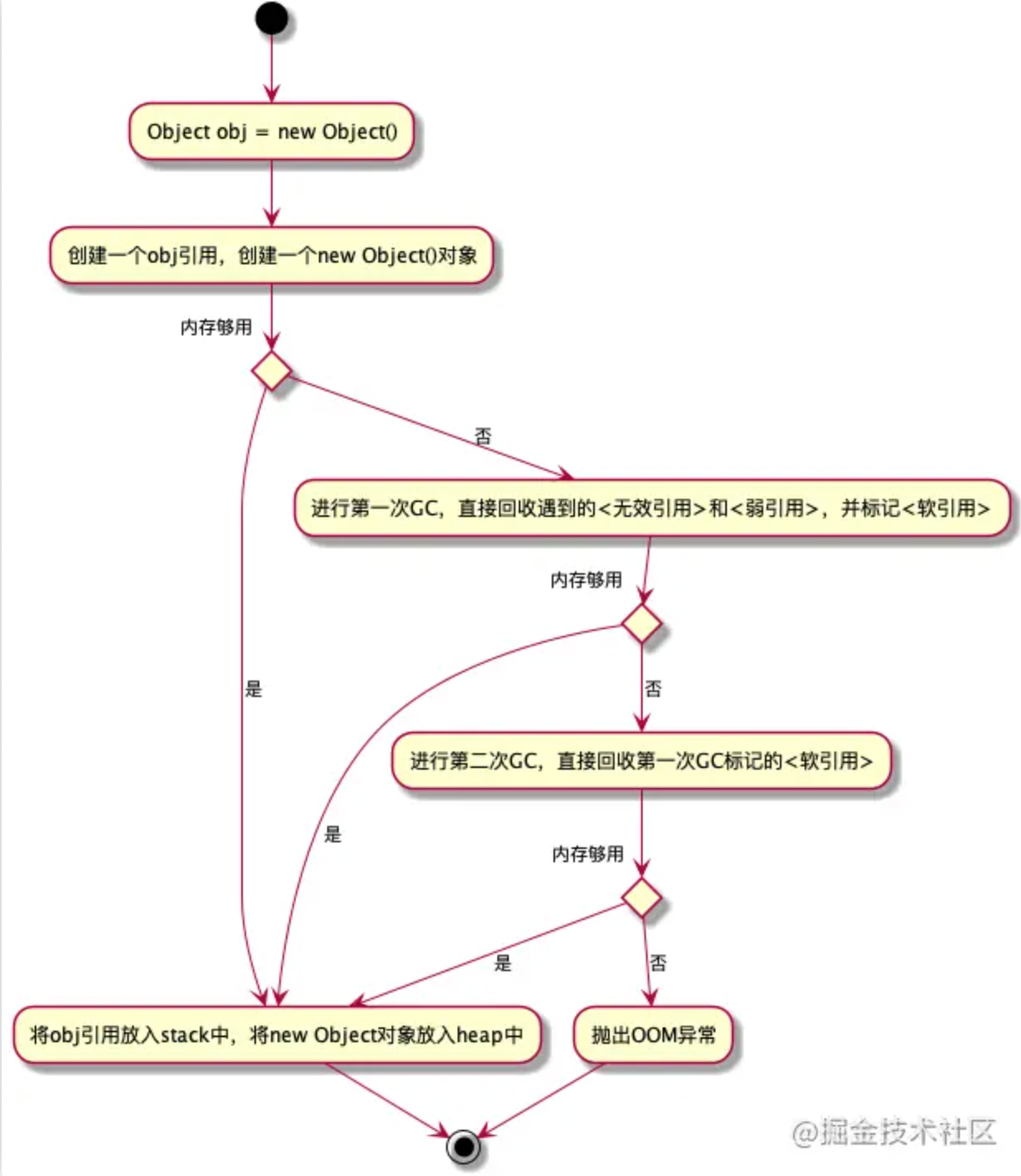
**回收时机**

垃圾回收时机，站在开发者的角度，有两个点:

* 1 主动回收，比如手动调用了System.gc();
* 2 被动回收，比如LargeObj large = new LargeObj();此时发现剩余内存放不下LargeObj()这个对象，就会触发垃圾回收机制。

大多数情况下都是被动回收的，我们就来分析下被动回收的流程:

* 1 LargeObj large = new LargeObj();这一行代码会创建两个对象，一个是large这个引用，放在方法栈里面的；一个是new LargeObj()这个对象，放在堆里面的，此时发现堆里面放不下LargeObj()这个对象，就会触发gc；
* 2 此时开始进行回收，gc会检测堆里面的所有对象，我们可以把堆里面为一个表格，gc会从上到下，从左到右，一行一行检测当前对象是否可被回收，如果是无效对象，或者是弱引用，则直接回收；如果是软引用，则仅标记；
* 3 经过第2步后，gc回收了一些无效对象和弱引用，然后看此时内存够用吗，够用就直接跳转到6步，不够则向下执行第4步；
* 4 此时发现回收过后内存还不够，于是就回收第2步标记的软引用；
* 5 经过第4步，回收了软引用后，内存够用吗，够用则跳转到第6步，不够则直接抛出OOM异常；
* 6 直接在堆空间给LargeObj()分配内存，并且将large放入方法栈中，指向LargeObj()这个对象；



**如何判定是否可回收**

**可达性分析算法**: 检测堆中的每个对象到GCRoot是否可达，如果可达，就是活对象，如果不可达，就是死对象；死对象可以直接回收。

**可作为GCRoot的点**:

1 方法区中静态属性 和 常量 引用的对象 2 虚拟机栈 和 本地方法栈中引用的对象 3 活跃线程的成员变量

public class Test2 {

public static int \_10M = 10 \* 1024 \* 1024;

// 持有一个10M的数组

private byte[] memory = new byte[\_10M];

public static void main(String[] args) {

System.out.println("方法未入栈");

// 打印内存

System.gc();

printMemory();

// 测试方法入栈

test();

// 测试方法出栈后，再次进行回收，再打印内存状态

System.gc();

System.out.println("方法已出栈");

printMemory();

}

public static void test() {

// 创建对象，就有了一个10M的成员变量，10M的成员变量被test2持有，也就是到test2是可达的

Test2 test2 = new Test2();

//test2 = null; // keyPoint，注释掉

// 在方法还没执行完(也就是还在栈中)，尝试回收并打印内存状态

System.gc();

System.out.println("创建局部变量，方法未出栈");

printMemory();

}

/\*\*

\* 打印当前总内存和可用内存

\*/

public static void printMemory() {

String total = Runtime.getRuntime().totalMemory() / 1024 / 1024 + "M";

System.out.println("total: " + total);

String free = Runtime.getRuntime().freeMemory() / 1024 / 1024 + "M";

System.out.println("free: " + free);

}

}

我们先打印当前的内存状态；然后进入测试代码(持有一个10M的内存块)打印内存状态，最后再等测试代码执行完(出栈)，再打印下内存状态。日志如下:

方法未入栈

total: 245M

free: 242M

创建局部变量，方法未出栈

total: 245M

free: 233M

方法已出栈

total: 245M

free: 243M

我们看到，正常情况下有243M可用，方法入栈后，创建了test2对象，而test2对象因为持有了10M的数组，也就是10M的数组到test2是可达的，所以即使gc()也不会回收，所以空闲空间打印出来就是233M，而等到test()方法出栈后，也就是test()方法已经不在虚拟机栈里面了，所以test2对象就不再是GCRoot了，所以10M的内存就可以被回收了，所以我们再次 执行gc()后，发现可用内存变为了243M，这就证明了**只有在虚拟机栈中的引用持有的对象才能是GCRoot**；现在我们把"keyPoint"点的注释打开，再次运行，如下:

方法未入栈

total: 245M

free: 242M

创建局部变量，方法未出栈

total: 245M

free: 242M

方法已出栈

total: 245M

free: 243M

**如何回收**

既然内存可以看成一个表格，那么怎么回收里面无用的格子呢，大概有两种策略:

* 1 先标记无用的，然后把无用的全部清理；
* 2 先标记有用的，然后把有用的复制下来，再一次清理整个格子。

可以细分为下面3种方法:

* 1 标记-清理

先从上到下从左到右扫描整个内存块，将无用的内存进行标记，然后再清理所有被标记的内存格，这样有两个缺点；1 如果回收的内存不连贯，就会造成大量的内存碎片 2 如果回收的内存比较多，则效率底，比如共有100个格子，结果其中99个格子都被标记了，那就需要回收99个格子，太慢了，效率低。



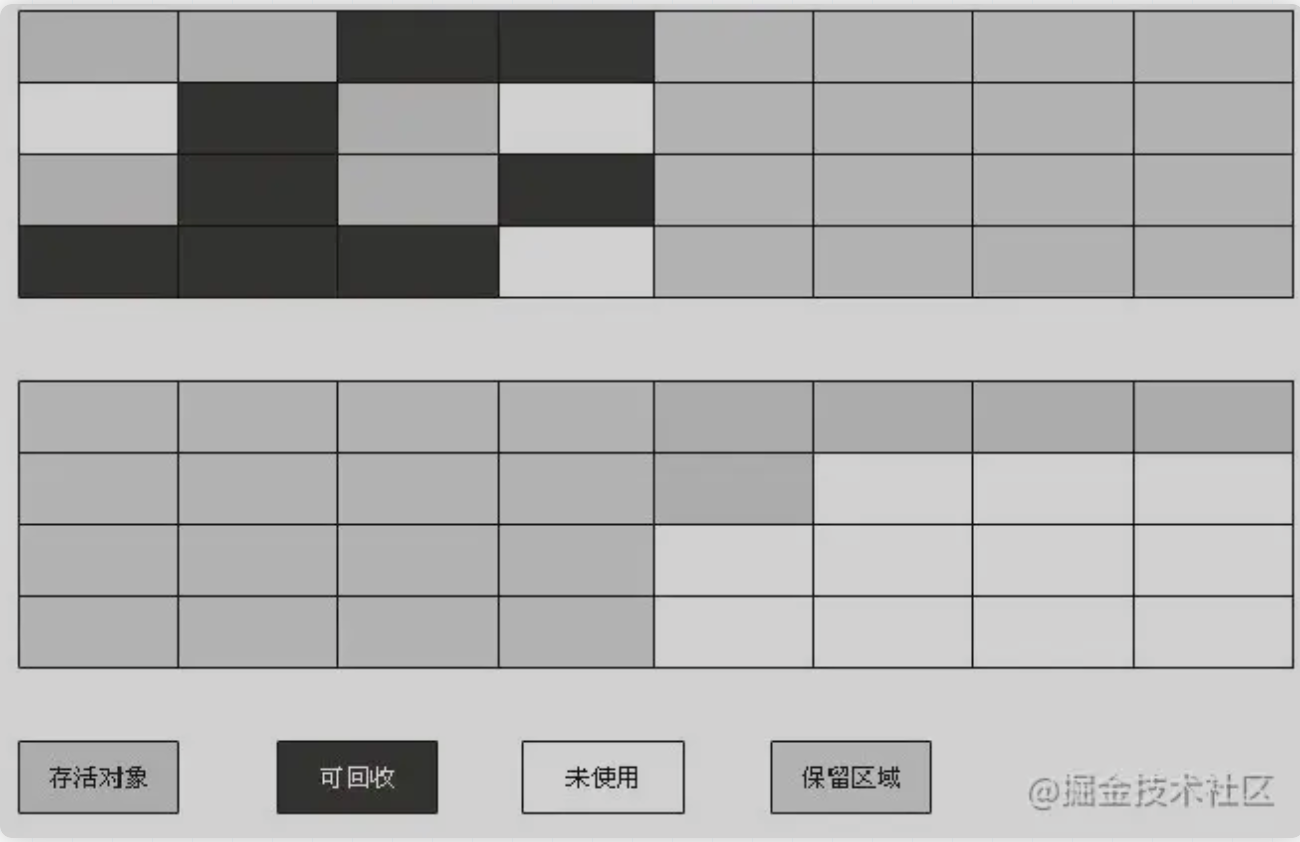
* 2 标记-整理

先从上到下从左到右扫描整个内存快，将无用内存进行标记，然后将无用内存都移动到一端，有用内存就在另一端，然后直接对无用内存的一端清除即可。优点就是避免了内存碎片，缺点还是效率低，回收的越多越慢。



* 3 复制算法

将内存分为大小相同的A、B两块，每次使用其中一块，比如A(我们将A简称为使用块，将B简称为存活块)，满了则对A进行扫描，然后将A上面存活的对象复制到B上，然后再将A全部清除，然后交换A、B的角色。优点就是快，不用遍历回收，而是一次直接清空A，缺点就是浪费内存，一次只能用一半内存，这样变相提高了gc()的频率，不划算。假如我们知道每次回收都能回收很多的对象，比如回收4/5， 也就是只有1/5存活，那么我们就可以将A、B的比列调整为4:1，也就是4/5给A，用来放置创建的对象，1/5给B，用来放置回收后剩下的对象，那么万一有几次存活的对象超过了1/5怎么办呢，我们就需要**内存担保**来处理这种情况了，这就涉及到分代策略了。



现在我们知道上述所有内存分配方法都有缺点，要么就是效率低，要么就是浪费空间，那么我们就可以根据不同场景来选择不同的回收算法。总的来说，**我们可以将对象分为两大类: 长工对象和短工对象**。

* 长工对象: 生命周期长，存活时间长，经过多次gc()还存活的对象，比如成员变量指向的对象。
* 短工对象: 生命周期短，存活时间短，通常一次gc()就把它干掉了，比如局部变量指向的对象，在方法出栈就GG了。

那么，如果是针对长工对象的回收，因为存活的太多了，一次必然只能回收一点点，那么使用复制算法就行不通，存活那么多，等价于几乎需要全部复制；而且存活的多，意味着存活块占的比例大，那么就太浪费内存了；所以复制算法不适合； 那么如果用标记-整理呢，因为存活的很多，所以死的就少，标记整理是回收死对象，那么回收的就少，反而提高了效率，正适合！

同理，对于短工对象的回收，存活的少，死的多，那么用标记-整理，则要回收很多，不合适，那么用复制算法呢，因为存活的少，所以要复制的就少；而且存活块栈的比例小，也就不太浪费内存了，所以正适合复制算法。

基于此，我们就可以: **针对长工对象采用标记-整理算法，针对短工对象，采用复制算法，这就称为分代算法**。其中长工对象放置的地方称为老年代，短工对象放置的地方称为新生代，然后针对不同的代使用不同的回收算法。

至此，可以总结为一句话: 老年代采用标记-整理算法，新生代采用复制算法。

我们先来看对象的分配流程(现在假设新生代和老年代都是空的):

* 1 创建一个对象User user = new User();
* 2 新创建的new User()对象首先会分配在新生代，如果新生代能放得下的话。
* 3 如果新生代放不下，则尝试对新生代进行一次gc()，称为MinorGC，此次gc()过后，所有存活的对象的年龄都+1，如果对象+1后年龄达到了15，则这个对象会直接移到老年代，我们可以简称为"年龄达标"。
* 4 如果此次gc()后，新生代还是放不下new User()对象，则直接放入老年代，我们可以简称称为"体积达标"。
* 5 如果老年代也放不下的话，则会对老年代进行一次gc()，称为MajorGC。
* 6 如果gc()后，还放不下，则进行二次gc()，也就是回收软引用的过程。
* 7 如果还是放不下，则抛出OOM异常。

综上，有两个条件可以进入老年代:

* 1 年龄达标: 年龄达到15的对象(gc发生了15次还存活下来的对象)被放到老年代。这是属于时间层面的。
* 2 体积达标: 新生代在MinorGC后还放不下，则直接进入老年代。这是属于空间层面的

上面我们说过，复制算法需要有内存担保，来防止存活块放不下的情况，这里的内存担保就是老年代，也就是说，当新生代的存活块B放不下存活的对象时，那么就放在老年代。也就是我们上面说的"体积达标"的情况。而且我们还可以知道，如果创建了一个大对象，导致新生代放不下，那么就会触发新生代的MinorGC，换言之，如果我们频繁的创建大对象， 就会导致频繁的MinorGC，而发生gc时，会有个stop world的过程，也就是停止所有线程，这就会造成卡顿，所以我们要尽量避免创建大对象。

新生代采用复制算法，我们上面说到，为了避免浪费内存，复制算法的使用块和存活块不是1:1的，HotSpot虚拟机内部的分配比例是9:1的，更详细的说是8:1:1的，我们把新生代分为3块，一个eden区和两个survivor区，其中eden占8份，两个survivor区各占1份，每次对象过来时，我们使用eden和其中一块survivor区来存放对象，使用另一块survivor 块来存放gc()后存活的对象。我们将使用的survivor块成为survivor from，将存放存活对象的survivor块成为survivor to。下面来模拟一下流程:

* 1 创建对象User user = new User();
* 2 使用eden + survivor from 来存放new User()对象，看是否能放得下。
* 3 如果放不下，我们对eden + survivor from块进行MinorGC: 存活的对象复制到survivor to块上去，并且年龄+1，然后将eden和survivor from清空。
* 4 此时eden和survivor from是空的，survivor to则存放了存活的对象，此时交换survivor from 和 survivor to的身份，也就是说，下次分配内存使用eden + survivor to，而存活块则使用survivor from。

后续分配策略跟上述相同。

那么我们为什么要使用8:1:1三块内存呢，直接使用9:1不可以吗？

我们知道，复制算法的原理就是**使用两块大小一样的内存块，一个作为使用块，一个作为存活块**，但是这样会浪费空间，才使用了9:1的策略，但是这样的话，复制的时候就有风险，所以需要额外的担保，那么我们能不能想个办法: 内存在使用时是9:1，复制时候是1:1呢？有!

现在我们将内存分为三块: A:B1:B2 = 8:1:1，我们先使用A+B1，来存放对象，此时我们用了9份(使用时是9:1)，等待gc()后要复制的时候，我们就等价于让B1和B2来互相复制(复制时是1:1)，前提是本次存活对象小于等于10%(大部分情况是满足的)。完事后，我们下次就使用:A+B2来使用，使用B1来作为存活块了。 可以看到，使用8:1:1这种方式，等价于提供一个主块A和两个挂载块B1和B2，更灵活。而使用9:1的话，根本无法达到上述目标。