GC 基础（Garbage Collector）

什么是垃圾?

没有引用指向的对象就叫垃圾

java vs c++

java GC 自动处理，开发效率高，执行效率低

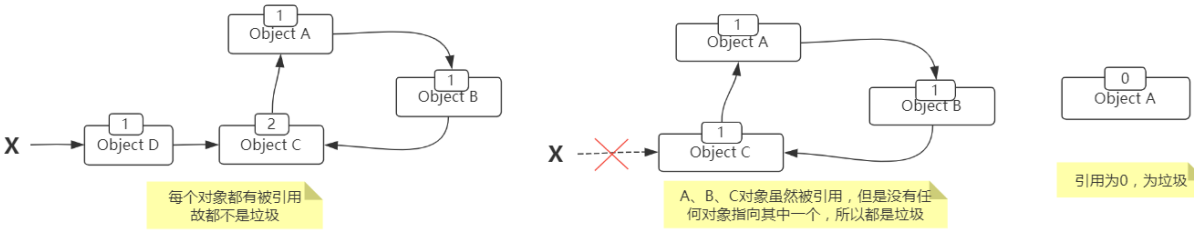
c++ 手工处理垃圾，容易忘记回收— 导致内存泄漏，对此回收—非法访问 开发效率低，执行效率高

找到垃圾有两种算法

- **Reference count 引用计数 —— python 采用的是这种**

  概念：有一个引用指向一个对象，在对象头写一个数字 ，有几个引用指向它，则标记为几，引用变成0的时候，则表示垃圾。

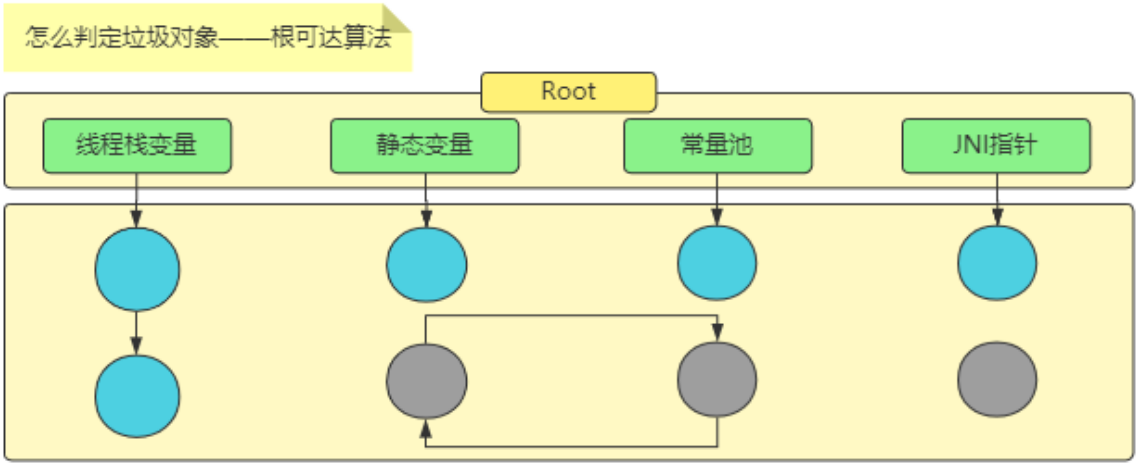
  弊端：不能解决的问题是 循环引用 A 引用B ，B引用C，C引用A，即他们三个互相引用，他们三个的引用标记都是1 但是没有别的引用指向他们，也就是说他们三个是一堆垃圾，这种情况GC回收不了



- **Root Searching 根可达算法 hotspot 采用这种**

  从根对象一直往下找，找不到的就是垃圾 下图最后三个就是垃圾

  根对象：程序起来马上要用到的对象就是根对象

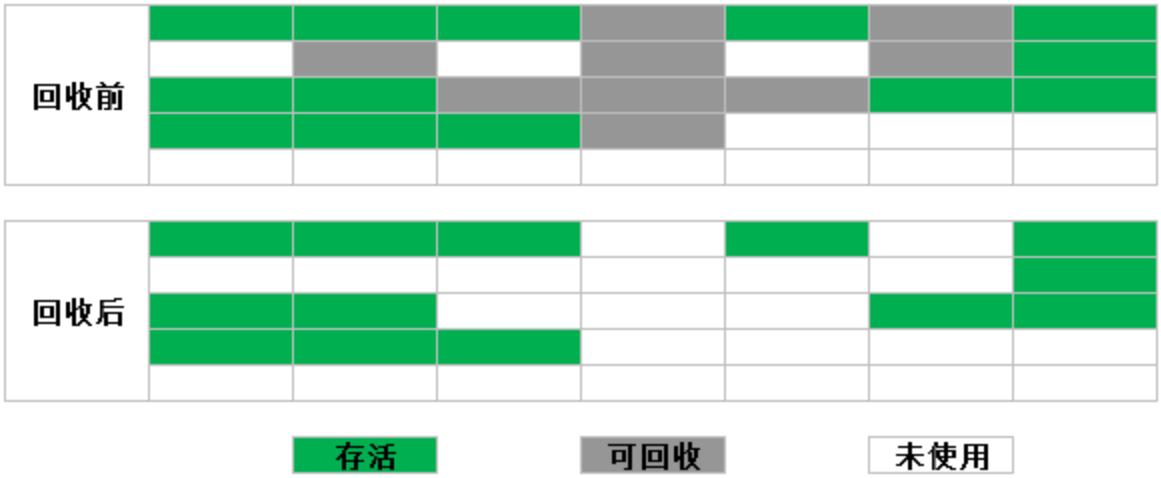


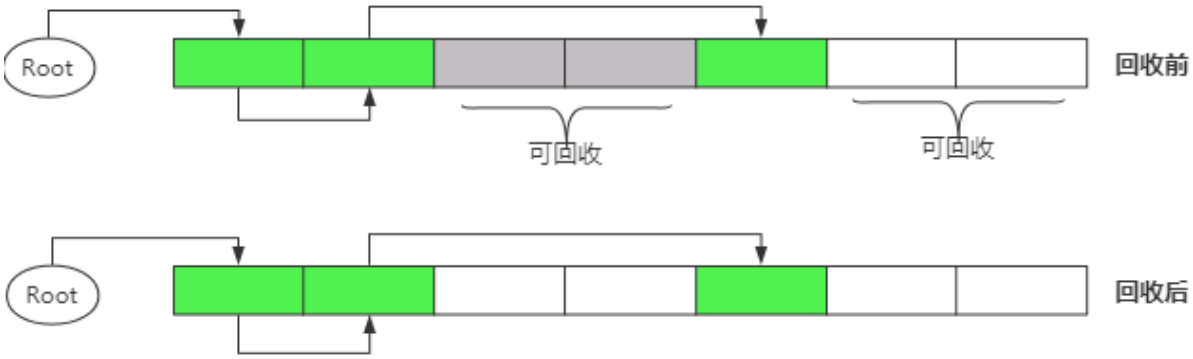
1. 线程变量：main方法开始，会启用一个线程，线程会启用一个线程栈，栈里面会有很多栈帧，栈帧里面开始的这些对象 算 根对象
2. 静态变量：class被load进内存的时候，会对这些静态变量初始化，这些静态变量能够访问到的对象也算是根对象
3. 常量值：当前class 会用到其它class 对象的 常量也算根对象
4. 调用了 java 写的 C / C++ 本地的方法 也算 根对象

GC清除算法

- **Mark-Sweep（标记清除）**

  先标记，后清除





适用场景：适用于存活对象比较多的情况下效率较高

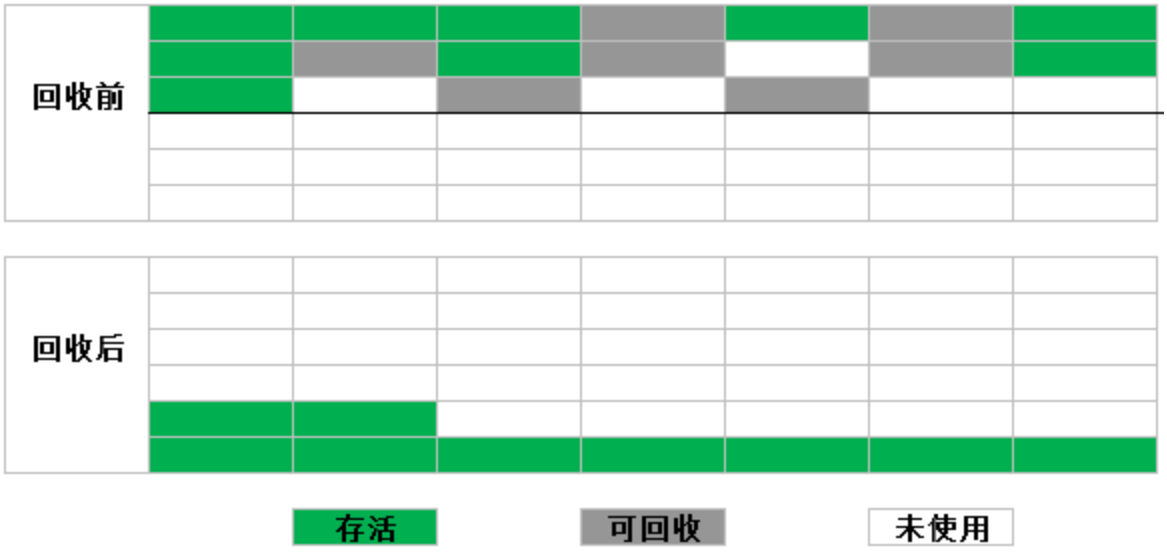
优点：算法相对简单

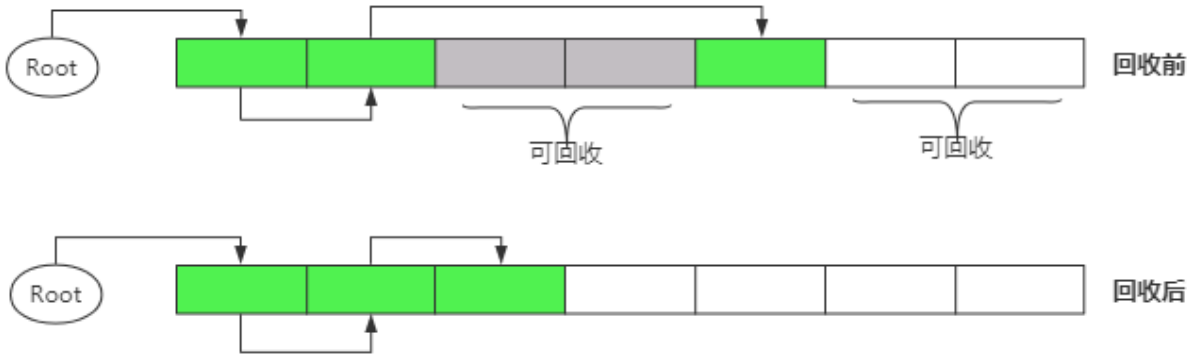
缺点：要经过两边扫描，效率偏低，第一遍找出有用的，第二遍找出没用的垃圾

容易产生空间碎片——即 内存之间会产生很多的空闲块 很多窟窿

- **Copying （拷贝）**

  把内存一分为二，吧有用的拷贝到一块，没用的拷贝到另一块，然后把没用的那一块清掉





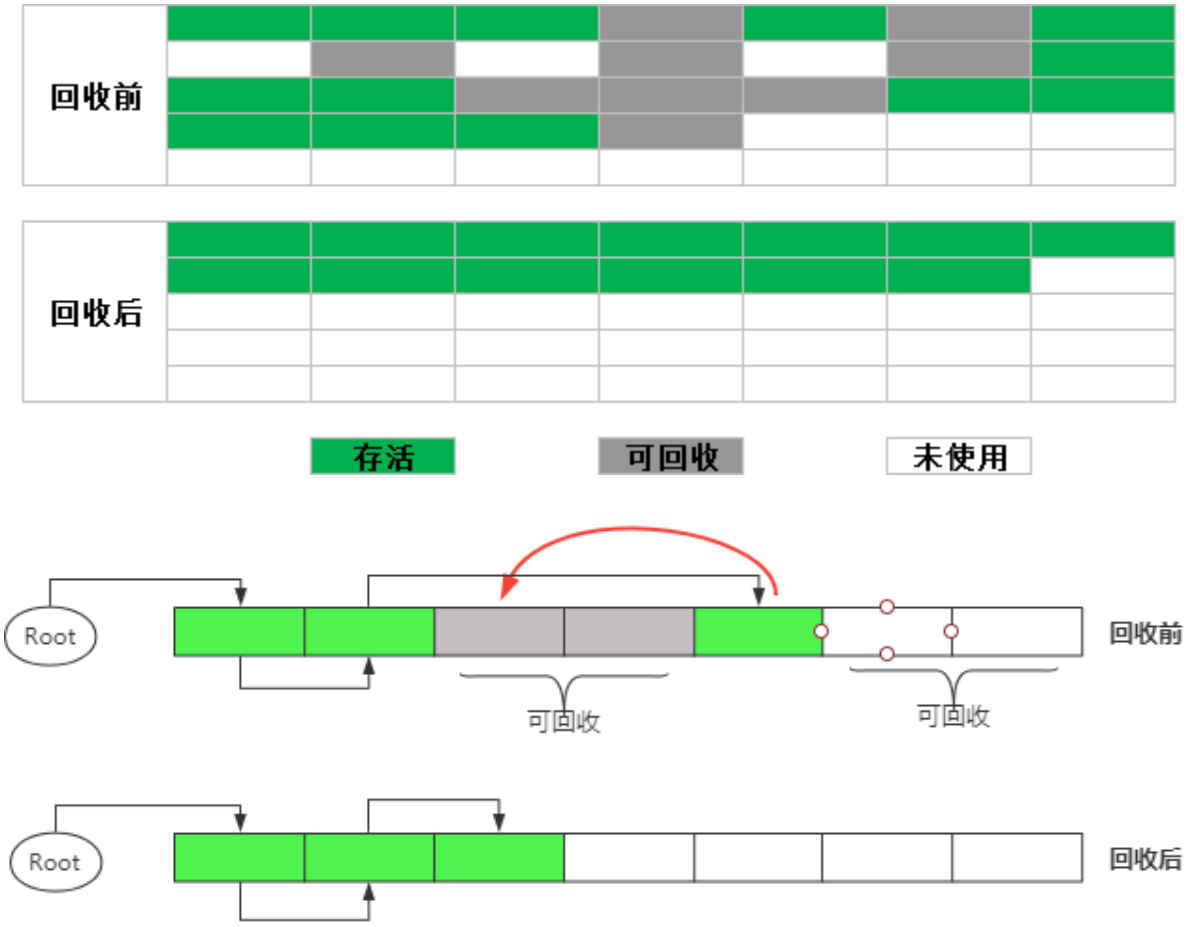
适用场景：存活对象较少的情况

缺点：对象复制过程，需要调整对象引用位置，浪费空间

优点：只扫描一次,内存分布清晰，可以紧挨在一起，不会产生碎片

- **Mark-Compact（标记压缩）**

  有用的全部压缩到头上去，剩下的大块东西全部空闲



缺点：效率低，扫描两次，第一遍扫描，找到不可回收的，然后从前面扫描空闲位置或者要回收的位置，再移动该不可回收的对象到前面，这样后面的整个区域空闲且连续

优点：不会产生碎片，方便对象分配，且不会产生内存减半

JVM内存分代模型，堆内的逻辑分区（不适用不分代的GC）

JVM 的 hotspot 实际是用的分代算法，分代存在于ZGC之前，现在基本用的GC基本都是分代的

查看 新生代和老年代的内存比例：

java -XX:+PrintFlagsFinal -version | grep NewRatio

```
[root@wwwwwfp ~]# java -XX:+PrintFlagsFinal -version | grep NewRatio
    uintx NewRatio                                 = 2                                   {product}
openjdk version "1.8.0_161"
OpenJDK Runtime Environment (build 1.8.0_161-b14)
OpenJDK 64-Bit Server VM (build 25.161-b14, mixed mode)
[root@wwwwwfp ~]#
```

逻辑分代是把堆内存区分为 新生代和老年代两个部分

新生代：刚new 出来的对象

新生代大量死去，少量存活，使用Coping算法

新生代有分了三个区域　一个 eden区域和两个 survivor区域 默认比例8 1 1

eden：new 出来对象往里面仍的区域

sruvivor：GC回收一次，跑到survivor，再回收一次，进入下一个survivor

老年代：GC回收了好多次都没有回收的对象

老年代存活率高，回收较少，采用MC或者MS

一个对象从出生到死亡

MinorGC/YGC ： 年轻代空间耗尽时触发——也就是年轻代满的时候进行回收

MajorGC/FullGC：在老年代无法继续分配空间时出发，新生代老年代同时进行回收———也就是老年代满的时候

new 对象之后，首先尝试往栈上仍，栈扔不下，先仍进对堆中的 Eden区域，垃圾回收一次后，进入 Survival S0, 再回收 进入 S1，下一次回收循环进入S0，循环复制年龄超过限制时进入old。通过参数：-XX:MaxTenuringThreshold 配置 ，可参考 http://www.yayihouse.com/yayishuwu/chapter/1623

虚拟机并不是永远地要求对象的年龄必须达到了MaxTenuringThreshold才能晋升老年代，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就可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄。

对象在各个区域的分配

- 栈上分配 —— 无需调整
  - 线程私有小对象
  - 无逃逸对象：只在某一段代码中使用，出了这一段代码没人认识它
  - 支持标量替换：比如有一个对象T，里面只有一个int m 和int n，这种情况完全可以用 两个 int 就可以代替这个对象，没必要把它当作一个完整的对象，即用普通类型代替整个对象
- 线程本地分配TLAB（Thread Local Allocation Buffer）—— 无需调整
  - 占用eden，默认 1%
  - 多线程的时候不用竞争eden就可以申请空间，提高效率
  - 小对象
- 老年代（大对象、持久对象）
- Eden（新对象）

线程本地分配 TLAB

栈上分配不下的会优先进行TLAB分配，每个线程在 eden区域分配一个1%的空间，这个空间只是某个线程独有，分配空间的时候，首先往线程独有的这个空间分配，这样就不会和其它线程产生争用，所以效率就会变高

```
//-XX:-DoEscapeAnalysis   -  去掉  逃逸分析   - 号标识去掉
//-XX:-EliminateAllocations - 去掉 标量替换
//-XX:-UseTLAB          - 去掉 TRAB
//-Xlog:c5_gc*
// 逃逸分析 标量替换 线程专有对象分配     运行的时候通过配置这些参数查看运行效率
public class TestTLAB {
    //User u;
    class User {
        int id;
        String name;
        public User(int id, String name) {
            this.id = id;
            this.name = name;
        }
    }

    void alloc(int i) {
        new User(i, "name " + i);   // 这个就表示无逃逸
        // u = new User(i,"");      // 这个就是 有 逃逸
    }

    public static void main(String[] args) {
        TestTLAB t = new TestTLAB();
        long start = System.currentTimeMillis();
        for(int i=0; i<1000_0000; i++) t.alloc(i);
        long end = System.currentTimeMillis();
        System.out.println(end - start);
        //for(;;);
    }
}
```

对象分配过程：

new 一个对象后，首先尝试到栈上分配（栈有一个好处 pop 完后就结束 无需 GC）

栈分配不下，根据指定的参数查看是否装下，比较对象大小，过大，直接进入old FGC 回收

如果对象大小小于配置参数值，先看是否满足TLAB，然后分配到Eden区

在Eden区域 进行GC清除，如果清除成功　结束

如果Eden GC没有清理，S1 再GC清除　如果年龄够进入old区域，如果年龄不够进入S2

1. 动态年龄：（不重要）
   https://www.jianshu.com/p/989d3b06a49d
2. 分配担保：（不重要）
   YGC期间 survivor区空间不够了 空间担保直接进入老年代
   参考：https://cloud.tencent.com/developer/article/1082730

1. 部分垃圾回收器使用的模型

> 除Epsilon ZGC Shenandoah之外的GC都是使用逻辑分代模型G1是逻辑分代，物理不分代除此之外不仅逻辑分代，而且物理分代

2. 新生代 + 老年代 + 永久代（1.7）Perm Generation/ 元数据区(1.8) Metaspace
   1. 永久代 元数据 - Class
   2. 永久代必须指定大小限制 ，元数据可以设置，也可以不设置，无上限（受限于物理内存）
   3. 字符串常量 1.7 - 永久代，1.8 - 堆
   4. MethodArea逻辑概念 - 永久代、元数据（不同版本不同叫法，永久代/元数据区）

      存的数据　Class 数据、方法编译完的数据、代码编译完的数据

   新生代 + 老年代　是堆　　MethodArea 是堆之外的空间

   区别：　方法区被所有线程共享

   1. 方法区是是一个概念，具体实现1.7和1.8 不同
   2. 1.8之前 Perm Generation 不能无限大，必须指定大小，指定完也不能改，当各种各样的类产生的比较多的时候容易产生溢出（如用一些框架类，容易产生各种各样的代理类，这些也是放到这个区的）

3. 1.8之后 mata space 没有大小限制，但是受限于物理内存
4. 字符串常量存储区域不同，1.7之前在Perm Generation 里面，1.8之后放在堆内存里面

1. 新生代 = Eden + 2个suvivor区
    1. YGC回收之后，大多数的对象会被回收，活着的进入s0
    2. 再次YGC，活着的对象eden + s0 → s1
    3. 再次YGC，eden + s1 → s0
    4. 年龄足够 → 老年代 （15 CMS 6）
    5. s区装不下 → 老年代
2. 老年代
    1. 顽固分子
    2. 老年代满了FGC Full GC
3. GC Tuning (Generation)
    1. 尽量减少FGC
    2. MinorGC = YGC
    3. MajorGC = FGC