JVM-垃圾回收

推荐博文

- https://blog.csdn.net/antony9118/article/details/51375662

三、垃圾回收

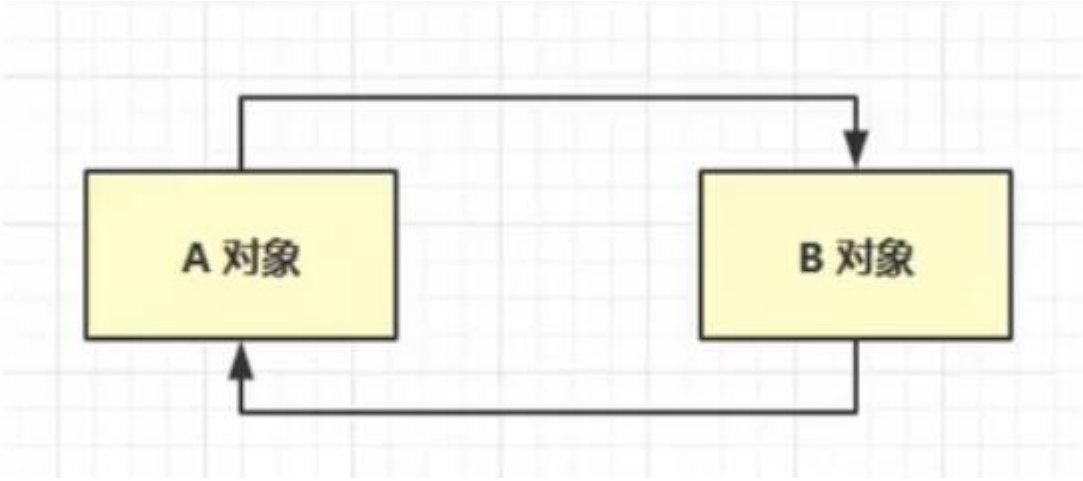
- [ ] 如何判断对象可以回收
- [ ] 垃圾回收算法
- [ ] 分代垃圾回收
- [ ] 垃圾回收器
- [ ] 垃圾回收调优

3.1 如何判断垃圾对象可以回收

3.1.1 引用计数法

早期Python使用

两个对象相互引用(引用变量均为1)，但是实际并未引用，程序无法通过计数值为0来垃圾回收

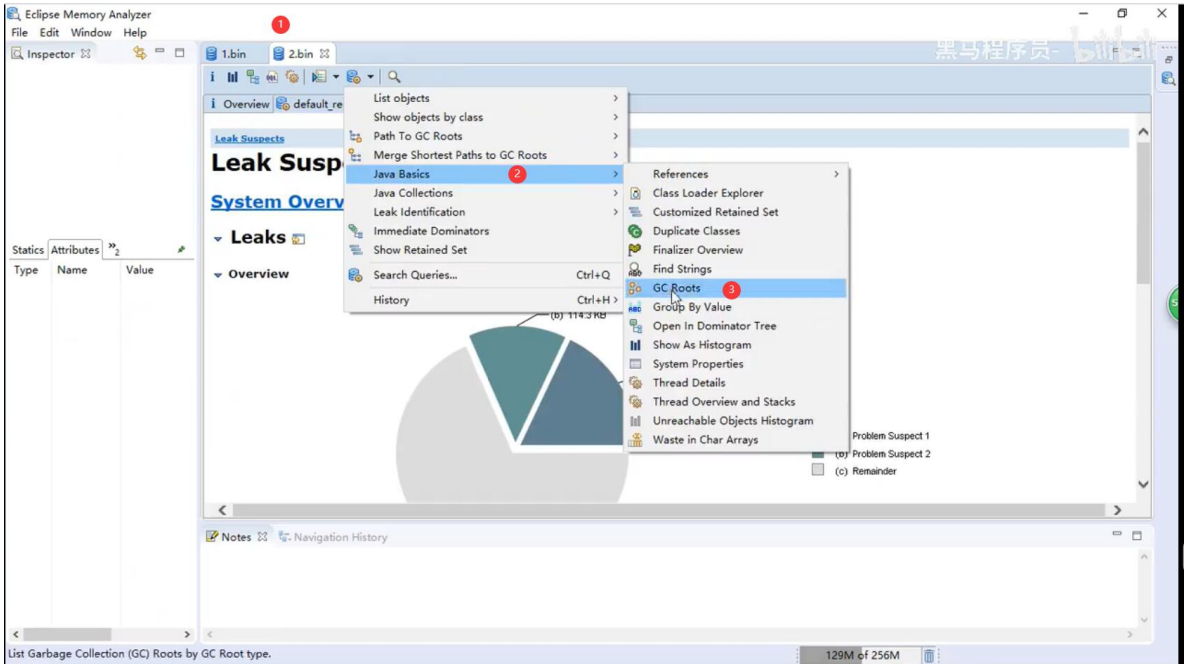


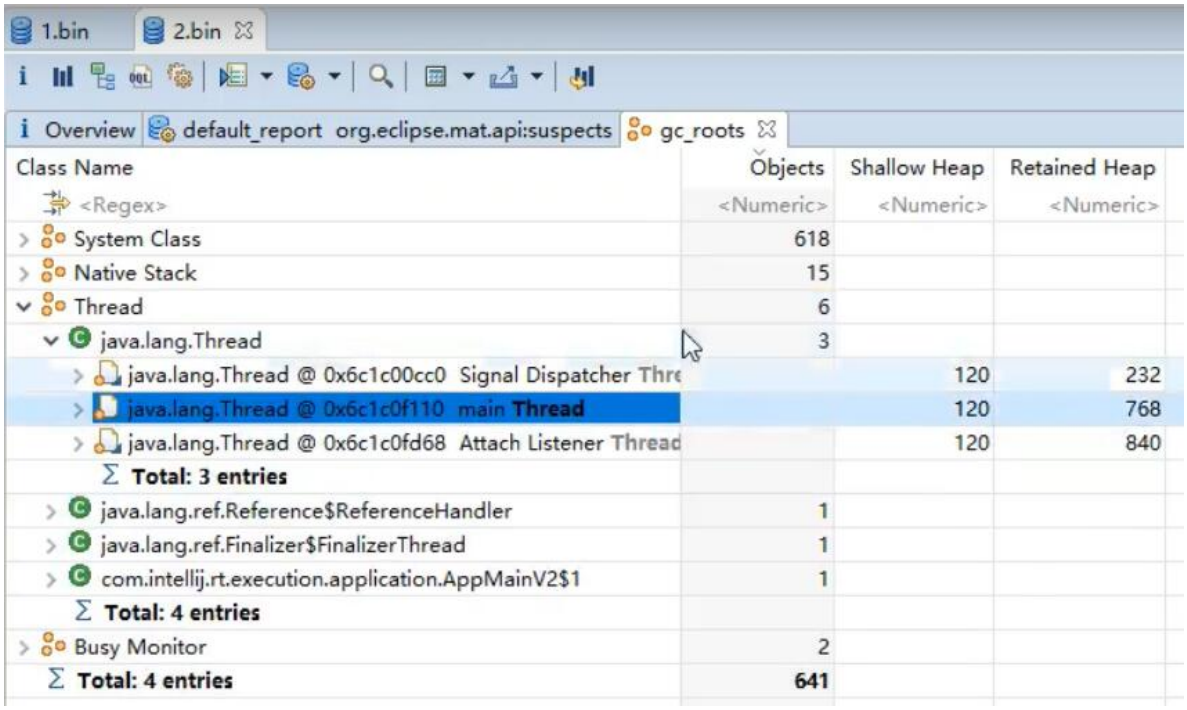
3.1.2 可达性分析算法

- Java 虚拟机中的垃圾回收器采用可达性分析来探索所有存活的对象
- 扫描堆中的对象，看是否能够沿着 GC Root对象 为起点的引用链找到该对象，找不到，表示可以回收；
- 根对象，肯定不能垃圾回收的；Java虚拟机中的垃圾回收器采用可达性分析来探索所有存活的对象；
- 扫描堆中的对象，看是否能够沿着GC Root对象为起点的引||用链找到该对象，找不到，表示可以回收哪些对象可以作为GC Root？

Eclipse Memory Analyzer

查看gc roots



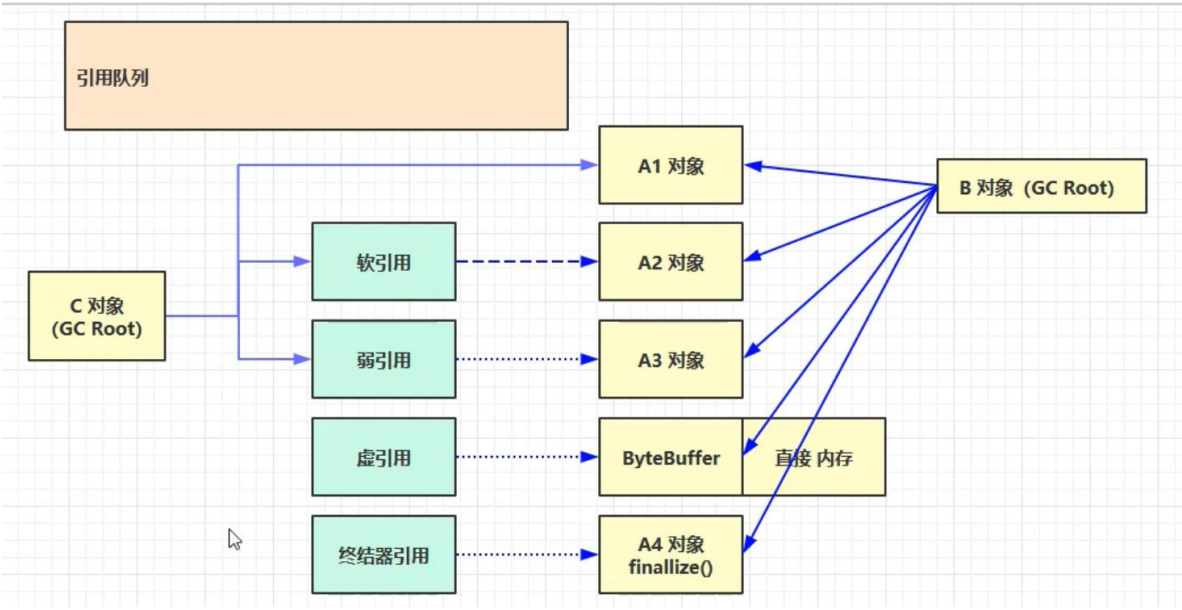


gc roots对象说明：

- SystemClass：系统类
- Native Stack：系统方法调用的Java对象
- Thread：活动线程调动的对象
- Busy Monitor：同步锁对象

3.1.3 四种引用

只有强引用是直接引用，其他引用都是间接引用，存在 ”中介对象“(软、弱、虚、终结器)



1 强引用

- 只有所有 GC Roots 对象都不通过【强引用】引用该对象，该对象才能被垃圾回收
- 举例：new一个对象，等号赋值（建立引用），沿着GC Root能找到此对象，则该对象不可被引用，与GC Root对象建立了强引用，如A1

2 软引用

- 内存不足时，且对象没有强引用，可以被回收
- 触发：对象被垃圾回收时&内存不够时

3 弱引用

- 对象没有强引用，即使内存充足也可被回收
- 触发：对象被垃圾回收时

4 引用队列：

- 目的：软、弱、虚、终这些中介对象占用一定内存，当无对象进行引用他们时，会利用引用队列，一次遍历即可释放这些中介对象。
- 触发：当引用软、弱、虚、终这些中介对象的对象被回收时

5 虚引用

- 软弱可以不借用引用队列，虚引用及终结器引用必须使用引用队列，虚引用主要配合 ByteBuffer 使用，被引用对象回收时，会将虚引用入队。
- 例如：创建ByteBuffer的时候会创建一个名为Cleaner的虚引用对象，当ByteBuffer没有被强引用所引用就会被jvm垃圾回收，虚引用Cleaner就会进入引用队列，会有专门的线程扫描引用队列，被发现后会调用直接内存地址的方法将直接内存释放掉，保证直接内存不会导致内存泄漏
- 由 Reference Handler 线程调用虚引用相关方法释放直接内存

6 终结器引用

- 如上A4对象，创建的时候会关联一个引用队列，且没有被强引用，当A4被垃圾回收的时候，会将终结器引用放入到一个引用队列（被引用对象暂时还没有被垃圾回收），有专门的线程（优先级较低，可能会造成对象迟迟不被回收）扫描引用队列并调用finallize()方法，第二次GC的时候才能回收掉被引用对象
- 无需手动编码，但其内部配合引用队列使用，在垃圾回收时，终结器引用入队（被引用对象暂时没有被回收），再由 Finalizer 线程通过终结器引用找到被引用对象并调用它的 finalize方法，第二次 GC 时才能回收被引用对象。

演示强引用、软引用

```
/**
 * @Description 演示软引用：网上浏览图片，会存储到缓存中，此时若图片对象采用了强引用，会造成内存溢出
 * @Setting -Xmx20m -XX:+PrintGCDetails -verbose:gc
 */
public class Demo2 {
    private static final int _4MB = 4 * 1024 * 1024;

    public static void main(String[] args) throws IOException {
        //hard();//java.lang.OutOfMemoryError: Java heap space
        System.in.read();
        soft();
    }

    /***
     * @Description 演示强引用
     */
    public static void hard() {
        List<byte[]> list = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            list.add(new byte[_4MB]);
        }
    }

    /***
     * @Description 演示弱引用
     */
    public static void soft() {
        List<SoftReference<byte[]>> list = new ArrayList<>();//list -> SoftReference -> byte[]，引用过程
        for (int i = 0; i < 5; i++) {
            //byte对象包装成软引用类型
            SoftReference<byte[]> reference = new SoftReference<>(new byte[_4MB]);
            System.out.println(reference.get());//打印软引用包装的对象
            list.add(reference);
            System.out.println(list.size());
        }
        System.out.println("循环结束：" + list.size());
        for (SoftReference<byte[]> reference : list) {
            System.out.println(reference.get());//打印软引用包装的对象，部分为null，被垃圾回收了
        }
    }
}
```

```
#打印日志如下
[B@1b6d3586
1
[B@4554617c
2
[B@74a14482
3
```

```
#第三次触发一次垃圾回收；PSYoungGen新生代，ParOldGen老年代
#Full GC,更全面的垃圾回收
#当一次Full GC回收后仍无足够内存，会释放引用软引用的对象。
[GC (Allocation Failure) [PSYoungGen: 2785K->480K(6144K)] 15073K-
>13360K(19968K), 0.0018255 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Ergonomics) [PSYoungGen: 480K->0K(6144K)] [ParOldGen: 12880K-
>13248K(13824K)] 13360K->13248K(19968K), [Metaspace: 3716K->3716K(1056768K)],
0.0112937 secs] [Times: user=0.11 sys=0.00, real=0.01 secs]
[B@1540e19d
4
[Full GC (Ergonomics) [PSYoungGen: 4208K->0K(6144K)] [ParOldGen: 13248K-
>5000K(11776K)] 17457K->5000K(17920K), [Metaspace: 3716K->3716K(1056768K)],
0.0077984 secs] [Times: user=0.02 sys=0.01, real=0.01 secs]
[B@677327b6
5

循环结束：5
null
null
null
[B@1540e19d
[B@677327b6
Heap
 PSYoungGen      total 6144K, used 4376K [0x00000000ff980000,
0x0000000100000000, 0x0000000100000000)
  eden space 5632K, 77% used
[0x00000000ff980000,0x00000000ffdc6200,0x00000000fff00000)
  from space 512K, 0% used
[0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)
  to   space 512K, 0% used
[0x00000000fff80000,0x00000000fff80000,0x0000000100000000)
 ParOldGen       total 11776K, used 5000K [0x00000000fec00000,
0x00000000ff780000, 0x00000000ff980000)
  object space 11776K, 42% used
[0x00000000fec00000,0x00000000ff0e2040,0x00000000ff780000)
 Metaspace       used 3722K, capacity 4540K, committed 4864K, reserved 1056768K
  class space    used 409K, capacity 428K, committed 512K, reserved 1048576K
```

配合引用队列

```
/**
 * @Description 演示软引用，配合软引用队列;
 * @PS 引用队列的作用：对象被垃圾回收时，没人引用软、弱、虚、终这些中介对象，则进入引用队列
 * @Setting -Xmx20m -XX:+PrintGCDetails -verbose:gc
 */
public class Demo3 {
    private static final int _4MB = 4 * 1024 * 1024;

    public static void main(String[] args) {
        soft();
    }

    /***
     * @Description 演示软引用
     */
    public static void soft() {
```

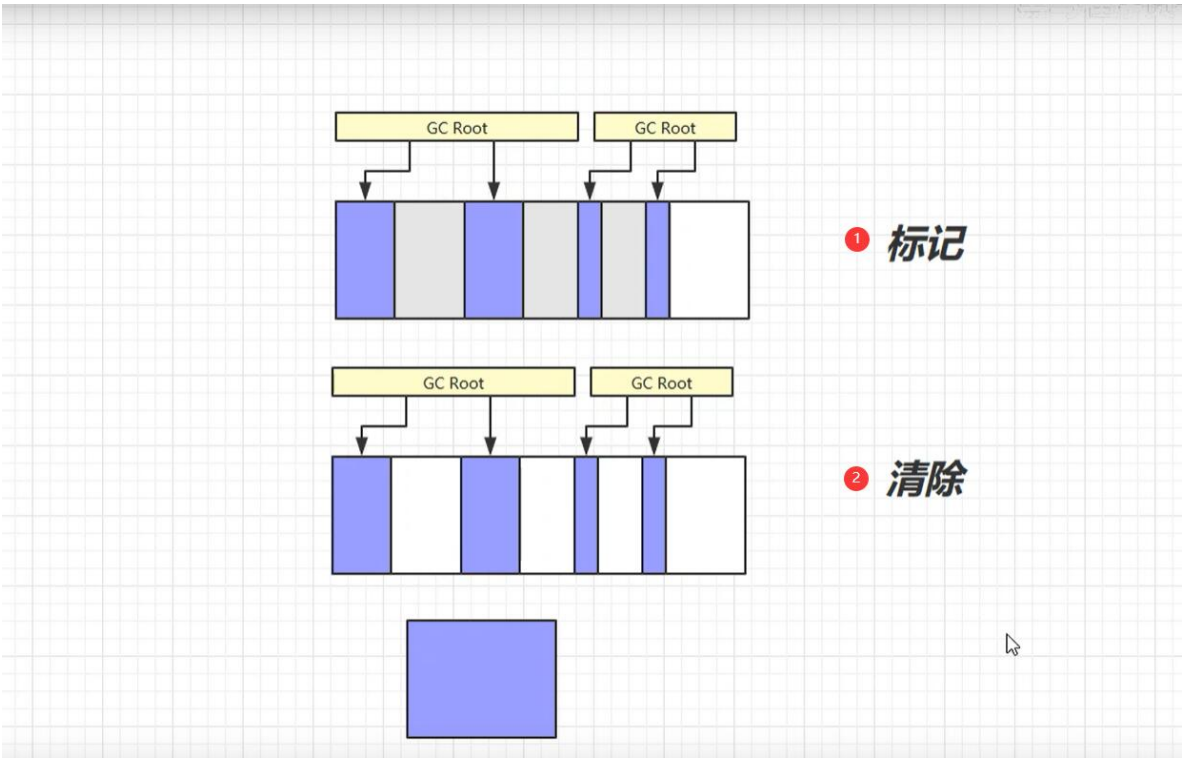
```
        List<SoftReference<byte[]>> list = new ArrayList<>();//list ->
SoftReference -> byte[]，引用过程
        ReferenceQueue<byte[]> queue = new ReferenceQueue<>();//引用队列
        for (int i = 0; i < 5; i++) {
            //byte对象包装成软引用类型
            SoftReference<byte[]> reference = new SoftReference<>(new
byte[_4MB], queue);
            System.out.println(reference.get());//打印软引用包装的对象
            list.add(reference);
            //期间会触发垃圾回收，使得软引用对象(中介)进入引用队列
            System.out.println(list.size());
        }
        Reference<? extends byte[]> poll = queue.poll();//获得最先进入队列的软引用对
象
        //poll不为空，说明引用队列不为空
        while (poll != null) {
            list.remove(poll);//对象集合中，移除没被引用的软引用对象
            poll = queue.poll();
        }
        System.out.println("循环结束: " + list.size());
        //配合引用队列后，不会打印null值了，因为软引用对象被回收了，只有没被垃圾回收的对象及其
引用的软引用对象
        for (SoftReference<byte[]> reference : list) {
            System.out.println(reference.get());//打印软引用包装的对象，部分为null，
被垃圾回收了
        }
    }
}
```

弱引用演示

上面代码Soft改为Weak

3.2 垃圾回收算法

3.2.1 标记清除算法



步骤说明：

- 标记GC Root不引用的对象
- 清除对象

优缺点：

- 速度快
- 清除垃圾后，不整理，易产生页内碎片(OS)

> 假如需要创造一段连续空间（数组），则无法装入

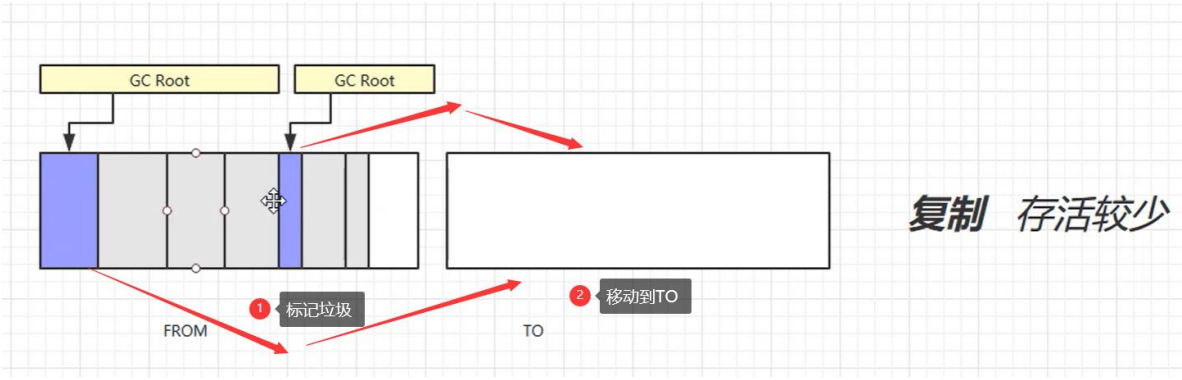
3.2.2 标记整理

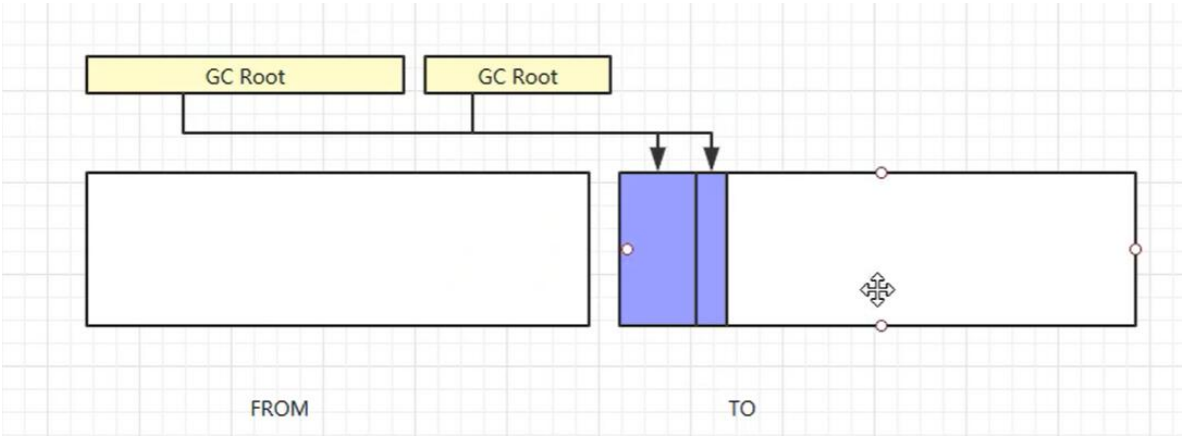
区别于**标记清除**，在清除的过程中，会整理空间，使后面的引用对象向前移动，从而使得空间连续。

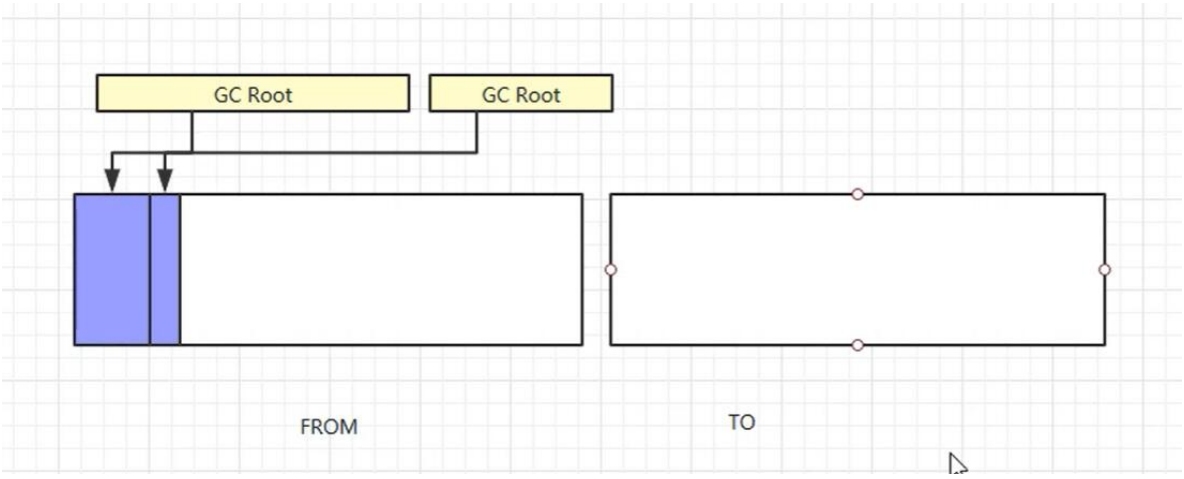
优缺点：

- 速度慢，因为涉及对象移动
- 清除垃圾后，不整理，不易产生页内碎片(OS)

3.2.3 复制







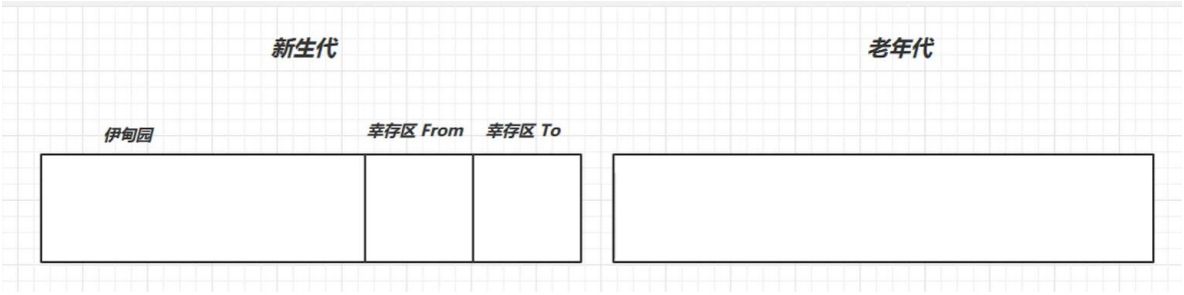
步骤说明：

- 开辟FROM、TO两个内存空间
- 进行垃圾标记
- 将存活对象移动到TO
- FROM中执行垃圾回收
- 交换FROM、TO，恢复初态。

优缺点：

- 不产生页内碎片(OS)
- 需开辟两个内存空间FROM、TO

3.3 分代垃圾回收



3.3.1 分代的概念

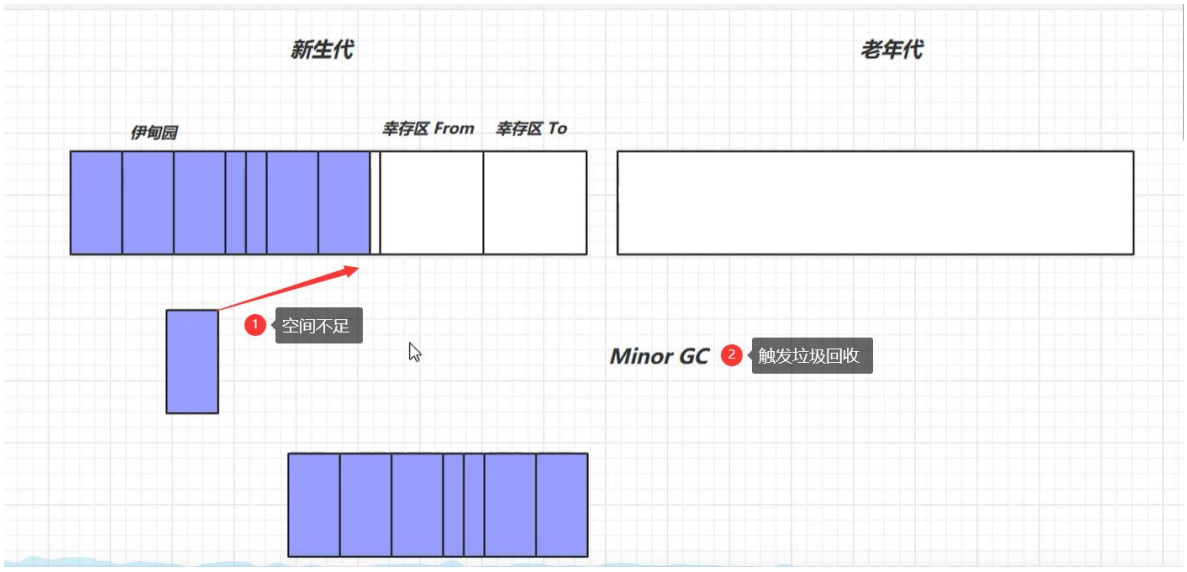
新生代

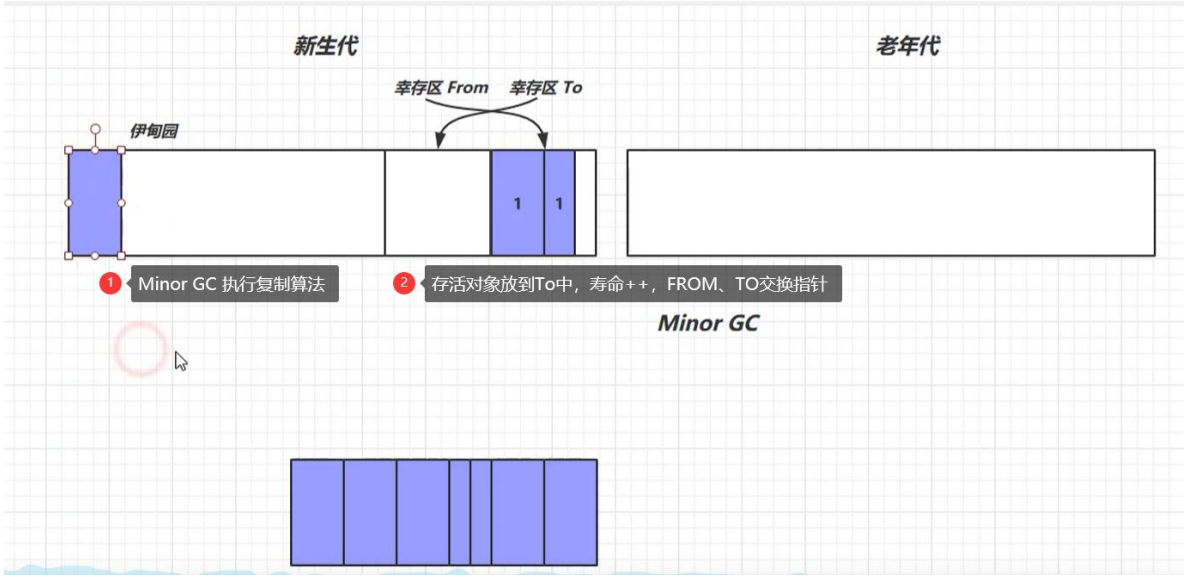
- 用完就可以丢弃、生命周期较短、随时需要回收的对象
- 手纸，饭盒

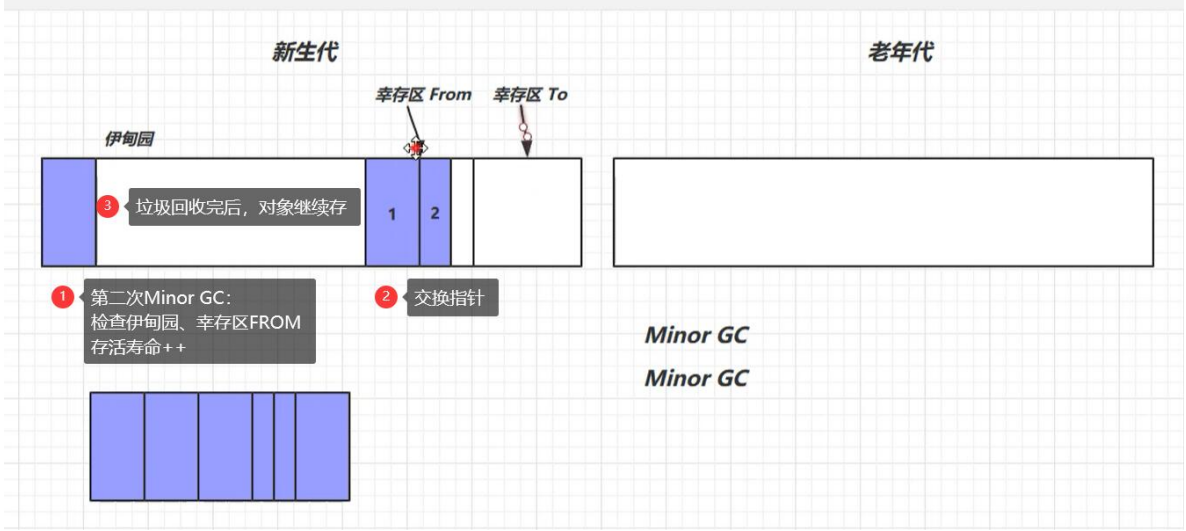
老年代

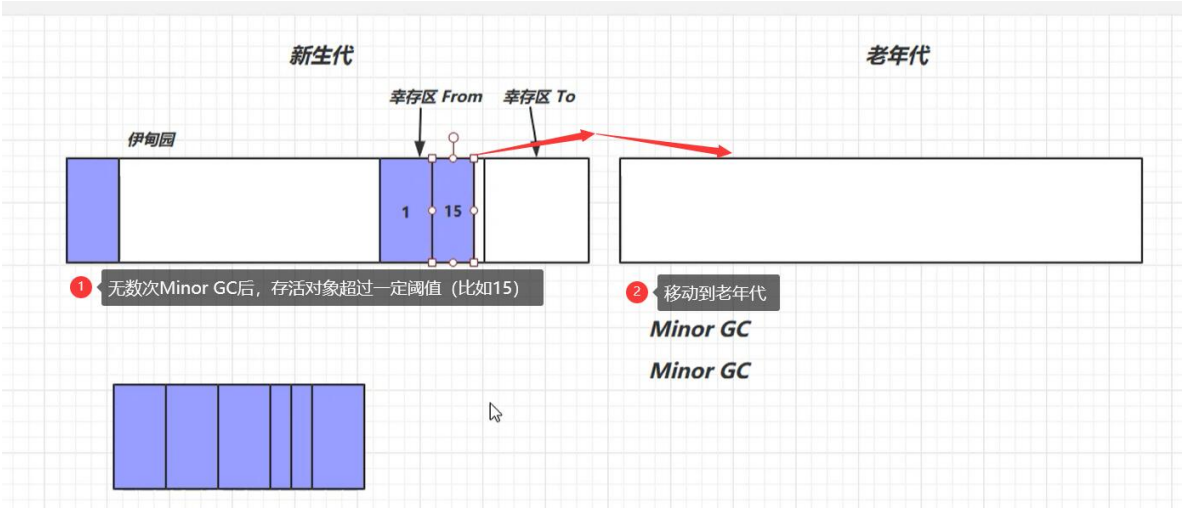
- 长时间使用的对象，更有价值，长时间存活
- 老物件

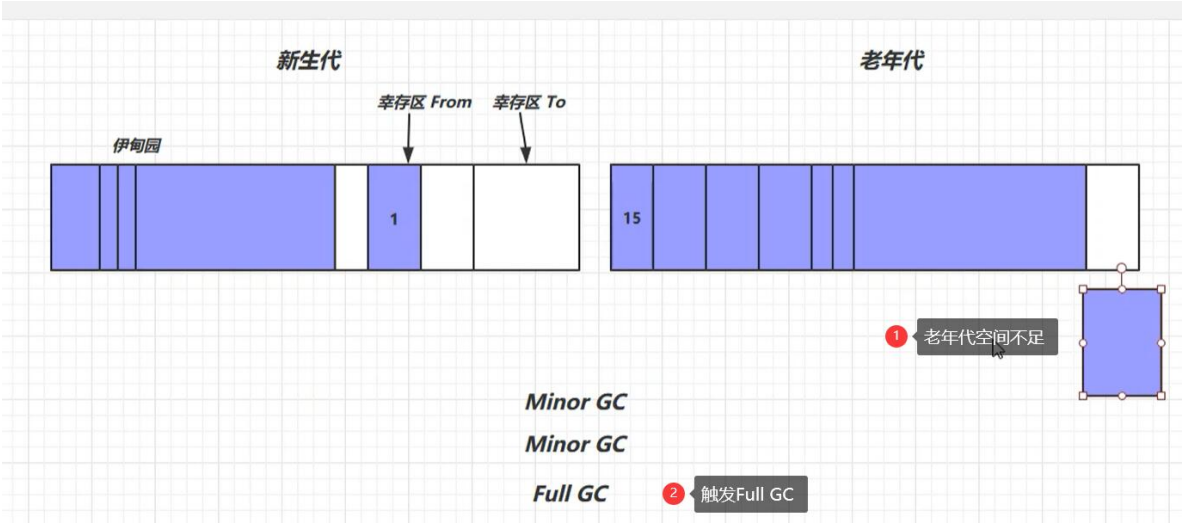
3.3.2 新生代-老年代的垃圾回收


新生代
老年代
伊甸园
幸存区 From
幸存区 To
1 空间不足
Minor GC
2 触发垃圾回收


新生代
老年代
幸存区 From
幸存区 To
伊甸园
1
1
1 Minor GC 执行复制算法
2 存活对象放到To中，寿命++，FROM、TO交换指针
Minor GC


新生代
老年代
幸存区 From
幸存区 To
伊甸园
3 垃圾回收完后，对象继续存
1
2
1 第二次Minor GC:
检查伊甸园、幸存区FROM
存活寿命++
2 交换指针
Minor GC
Minor GC





步骤说明：

- 对象首先分配在伊甸园区域
- 新生代空间不足时，触发 minor gc，伊甸园和 from 存活的对象使用 copy 复制到 to 中，存活的对象年龄加 1并且交换 from to
- minor gc 会引发 stop the world，暂停其它用户的线程，等垃圾回收结束，用户线程才恢复运行，当对象寿命超过阈值时，会晋升至老年代，最大寿命是15（4bit）
- 当老年代空间不足，会先尝试触发 minor gc，如果之后空间仍不足，那么触发 full gc，STW的时间更长

> 当要加入的内存对象太大，新生代内存不够，若老年代符合，则放入老年代。

代码演示：

```
/**
 * @Description 测试子线程内存溢出时是否会导致整个Java程序报错
 */
public class Demo4 {
    private static final int _8MB = 8 * 1024 *1024;
    //-Xms20M -Xmx20M -Xmn10M -XX:+UseSerialGC -XX:+PrintGCDetails -verbose:gc
    public static void main(String[] args) throws InterruptedException {
        new Thread(()->{
            ArrayList<byte[]> list= new ArrayList<>();
            list.add(new byte[_8MB]);
            list.add(new byte[_8MB]);
        }).start();
```

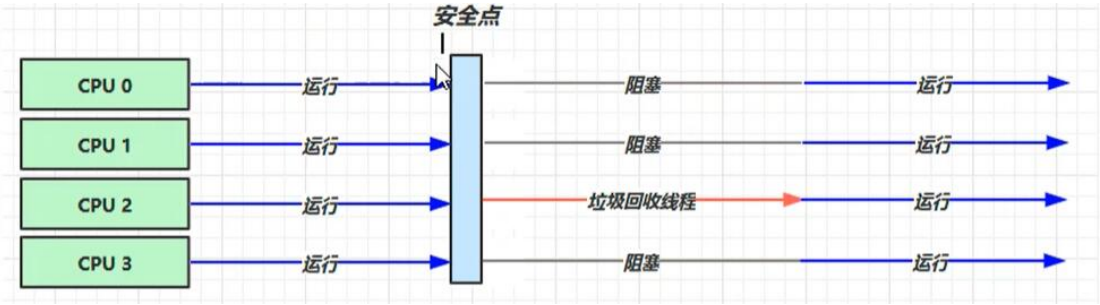
```
        //下述代码若取消睡眠后，则不影响整个程序
        System.out.println("sleep.........");
        Thread.sleep(1000000000L);
    }
}
```

3.3.3 垃圾回收参数说明

堆初始大小	-Xms
堆最大大小	-Xmx 或 -XX:MaxHeapSize=size
新生代大小	-Xmn 或 (-XX:NewSize=size + -XX:MaxNewSize=size)
幸存区比例（动态）	-XX:InitialSurvivorRatio=ratio 和 -XX:+UseAdaptiveSizePolicy
幸存区比例	-XX:SurvivorRatio=ratio
晋升阈值	-XX:MaxTenuringThreshold=threshold
晋升详情	-XX:+PrintTenuringDistribution
GC详情	-XX:+PrintGCDetails -verbose:gc
FullGC 前 MinorGC	-XX:+ScavengeBeforeFullGC

3.4 垃圾回收器

3.4.1 串行

- 单线程
- 堆内存较小，适合个人电脑
- 虚拟机参数：`-XX:+UseSerialGC= Serial + SerialOld`
  - `Serial`：工作在新生代，复制算法
  - `SerialOld`：工作在老年代，标记+整理算法
- 

  - 安全点：4个线程同时运行，期间内存不够触发垃圾回收，之后4个线程同时在**安全点**停下来。因为垃圾回收过程中，会改变内存地址，为了安全的使用这些地址，所以需要当前运行的线程在安全点停下来，此时完成垃圾回收工作就不会被干扰了。
  - 期间垃圾回收线程运行，其他线程阻塞。

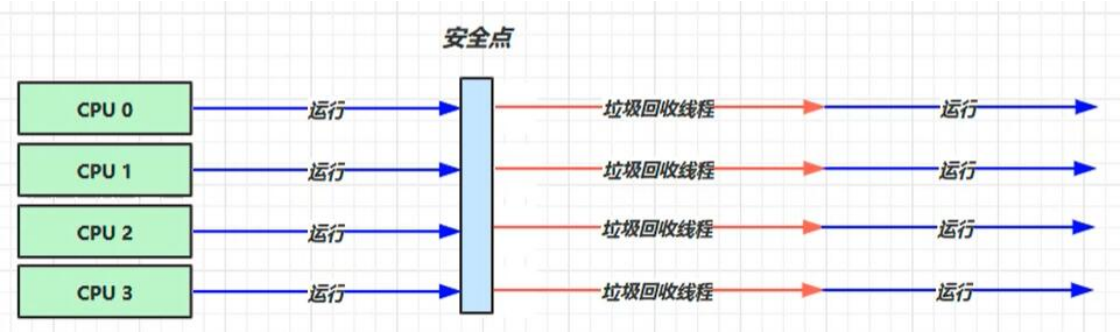
3.4.2 吞吐量优先

- 多线程
- 堆内存较大，多核cpu
- 让单位时间内，STW（响应时间）的时间最短 （少餐多食）

    - 0.2 0.2 = 0.4
- 虚拟机参数：
    - `-XX:+UseParallelGC ~ -XX:+UseParallelOldGC` （1.8默认开启）
        - 新生代，复制算法；老年代，标记整理算法
        - `Parallel`：并行，多线程
    - `-XX:+UseAdaptiveSizePolicy`
        - 动态调整伊甸园与survivor(幸存)区，以及其他堆内存
    - `-XX:GCTimeRatio=ratio`
        - 设置吞吐量大小，它的值是一个 0-100 之间的整数。假设 GCTimeRatio 的值为 n，那么系统将花费不超过 1/(1+n) 的时间用于垃圾收集（一般设置为19）
    - `-XX:MaxGCPauseMillis=ms`
        - 最大暂停毫秒数（默认200ms）

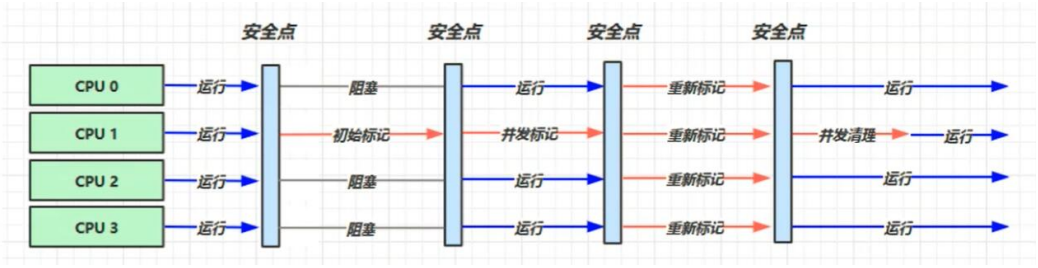
> 上面两个参数所达到的效果是冲突的：`GCTimeRatio`增加，吞吐量增加，堆内存增加，暂停时间增加，`MaxGCPauseMillis`即最大暂停时间减小，吞吐量也就限制了。

    - `-XX:ParallelGCThreads=n
        - 垃圾回收线程数（默认是cpu核数）
-



3.4.3 响应时间优先

- 多线程
- 堆内存较大,多核cpu
- 尽可能让单次STW的时间最短（少食多餐）
    - 0.1 0.1 0.1 0.1 0.1 = 0.5
    - 虚拟机参数：
        - `-XX:+UseConcMarkSweepGC ~ -XX:+UseParNewGC ~ SerialOld`
            - `concurrent`:并发，`MarkSweep`：标记清除，简称`CMS`：https://www.jianshu.com/p/86e358afdf17
            - 用户进程与垃圾回收进程并发执行，抢占资源。
            - `-XX:+UseConcMarkSweepGC ~ -XX:+UseParNewGC`：成对执行，一个老年代，一个新生代。若并发失败，则退化成串行`SerialOld`回收方式。
        - `-XX:ParallelGCThreads=n ~ -XX:ConcGCThreads=threads`
            - 并行线程数 ~ 并发线程数
            - 并发线程数建议设置成并行线程数n的1/4
        - `-XX:CMSInitiatingOccupancyFraction=percent`
            - 执行CMS垃圾回收的内存占比(控制何时进行CMS垃圾回收)

        - 例：`percent`=60%，老年代内存达到60%，执行CMS垃圾回收（剩余的40%，留给浮动垃圾，若`percent`=100%，则浮动垃圾空间不够）
    - `-XX:+CMSScavengeBeforeRemark`
        - 前言：新生代会引用老年代对象，为了重新标记，期间会扫描整个堆，但是许多新生代本身是作为垃圾要回收的，因此标记了"垃圾"，做了无用功。
        - 在CMS GC前启动一次新生代gc（`XX:+UseParNewGC`），目的在于减少old gen对ygc gen的引用，降低remark时的开销，一般CMS的GC耗时 80%都在remark阶段。
- 

  > 初始标记非常快，暂停时间短。
  >
  > 重新标记，因为第二个并发运行阶段可能造成对象内存改变，所以重新标记。期间会线程阻塞。
  >
  > 只有初试标记和重新标记会造成其他线程阻塞，其余均并发执行。
  >
  > 第四个安全点：垃圾回收线程**执行并发清理**的过程中，其他用户线程处于运行状态，在次期间，产生的垃圾无法被回收，只能等到下一次垃圾回收时才可以进行回收，我们称这些垃圾为**浮动垃圾**。下一次的垃圾回收，需要内存空间不足，才可清除垃圾，期间浮动垃圾会存在没有空间存放的尴尬局面；因此浮动垃圾需要预留一块空间来存储。
  >
  > 缺点：CMS是标记清除算法，会产生内存碎片，当内存碎片太多时，会造成并发失败，并发失败则`XX:+UseParNewGC`会退化成`SerialOld`串行老年代回收，清理碎片，时间消耗多。

3.4.4 G1 **

定义: Garbage First、Garbage One

- 2004论文发布
- 2009 JDK 6u14体验
- 2012 JDK 7u4官方支持
- 2017JDK9默认

适用场景：

- 同时注重吞吐量(Throughput) 和低延迟(Low latency)，默认的暂停目标是200 ms
- 超大堆内存,会将堆划分为多个大小相等的Region
- 整体上是标记+整理算法,两个区域之间是复制算法

相关帖子：

https://www.jianshu.com/p/548c67aa1bc0

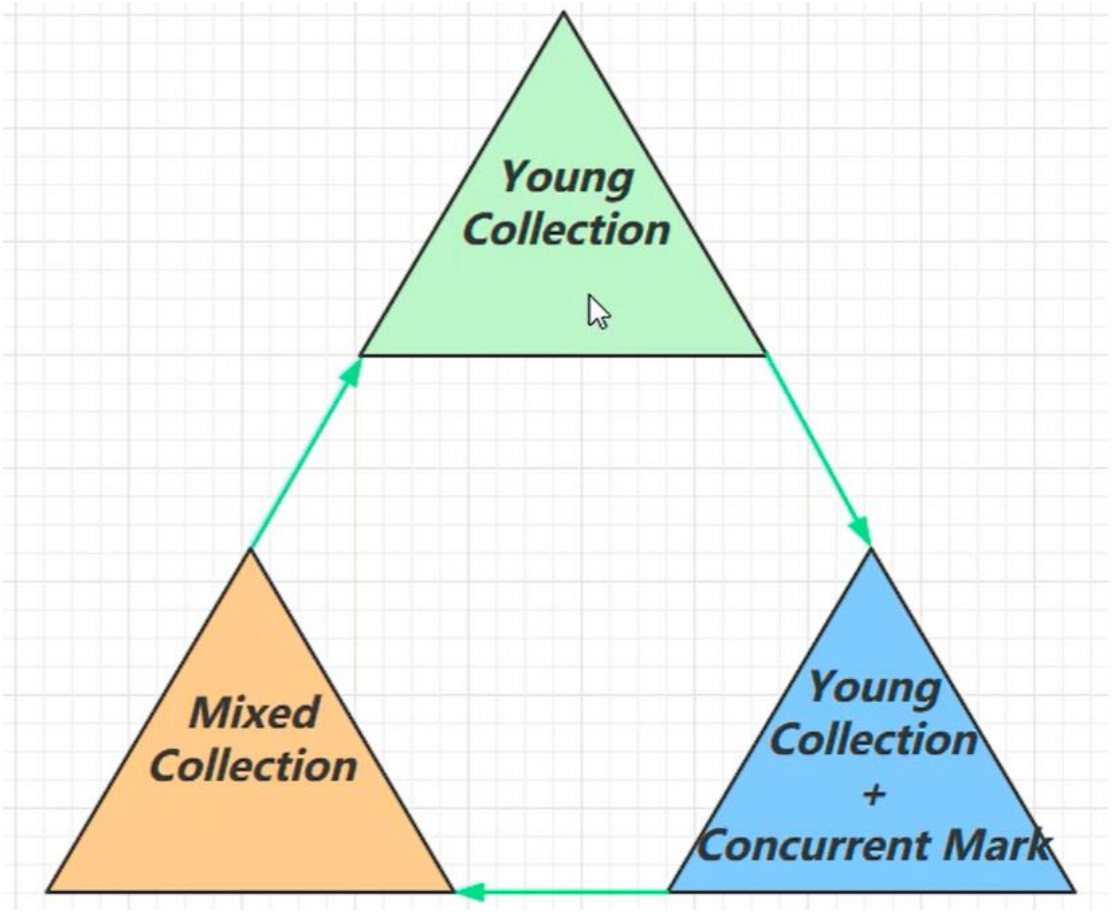
相关JVM参数

`-XX: +UseG1GC`

`-XX:G1HeapRegionSize=size`

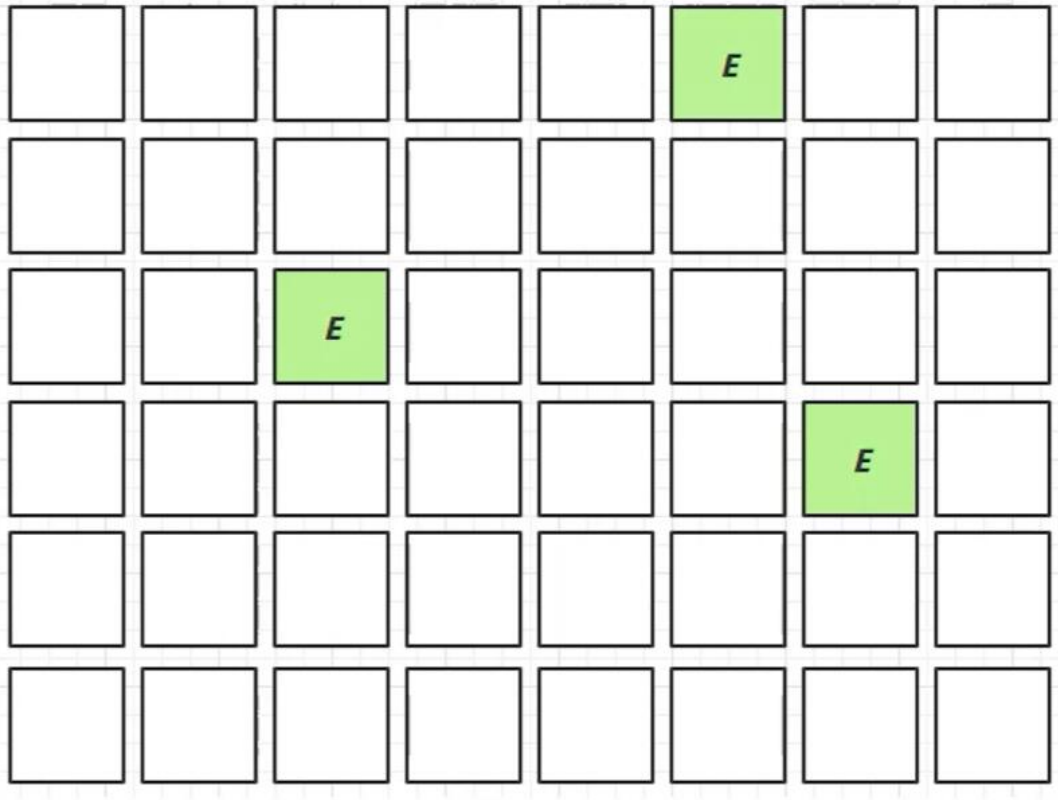
`-XX :MaxGCPauseMillis=time`

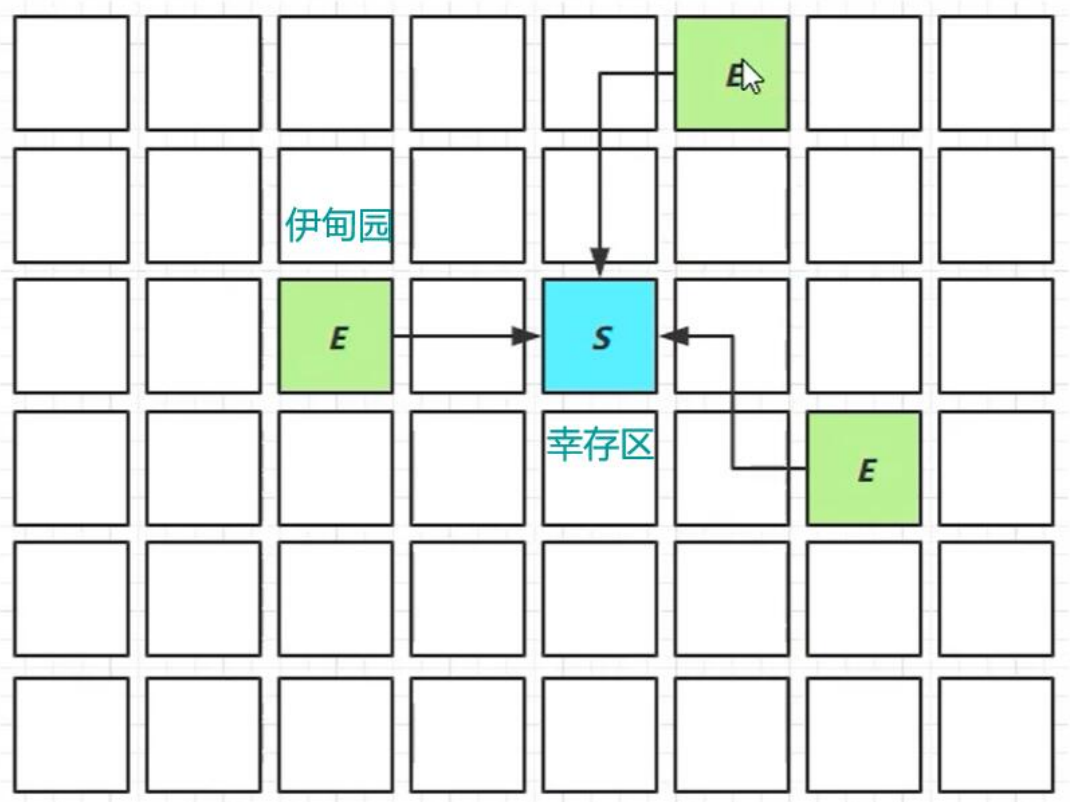
(1) 垃圾回收阶段

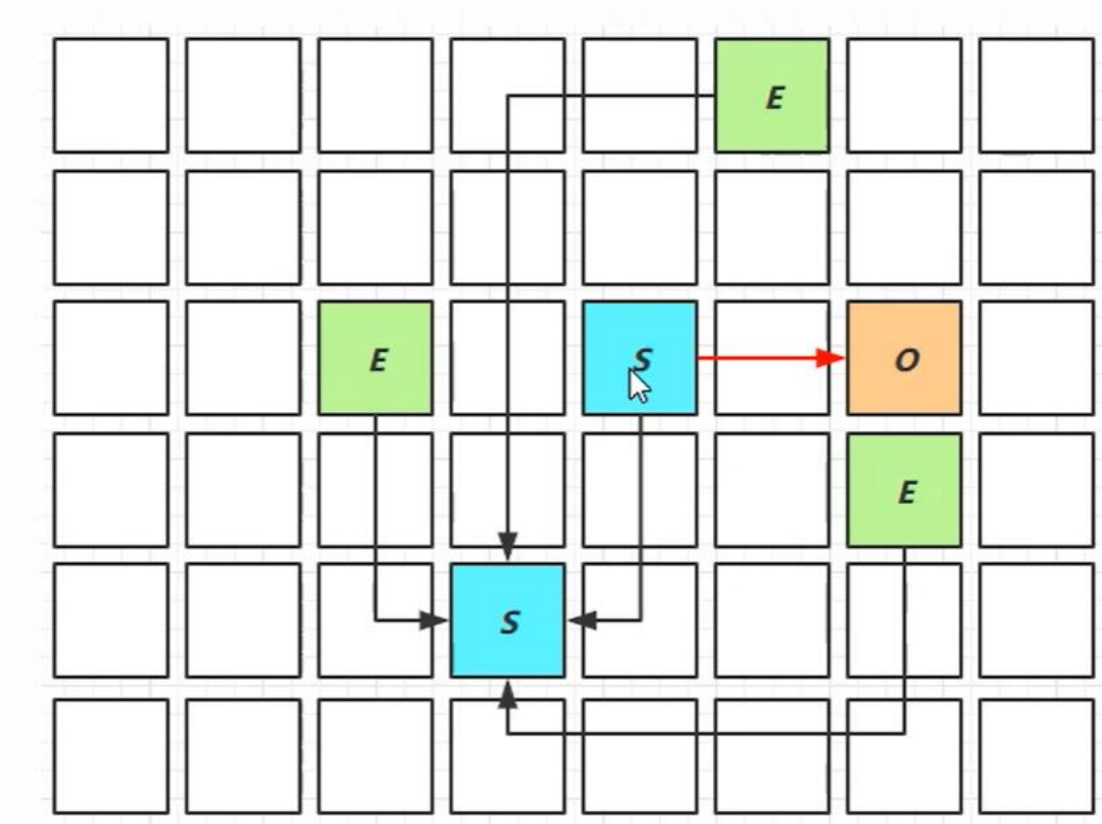


(2) Young Collection

- 会 STW

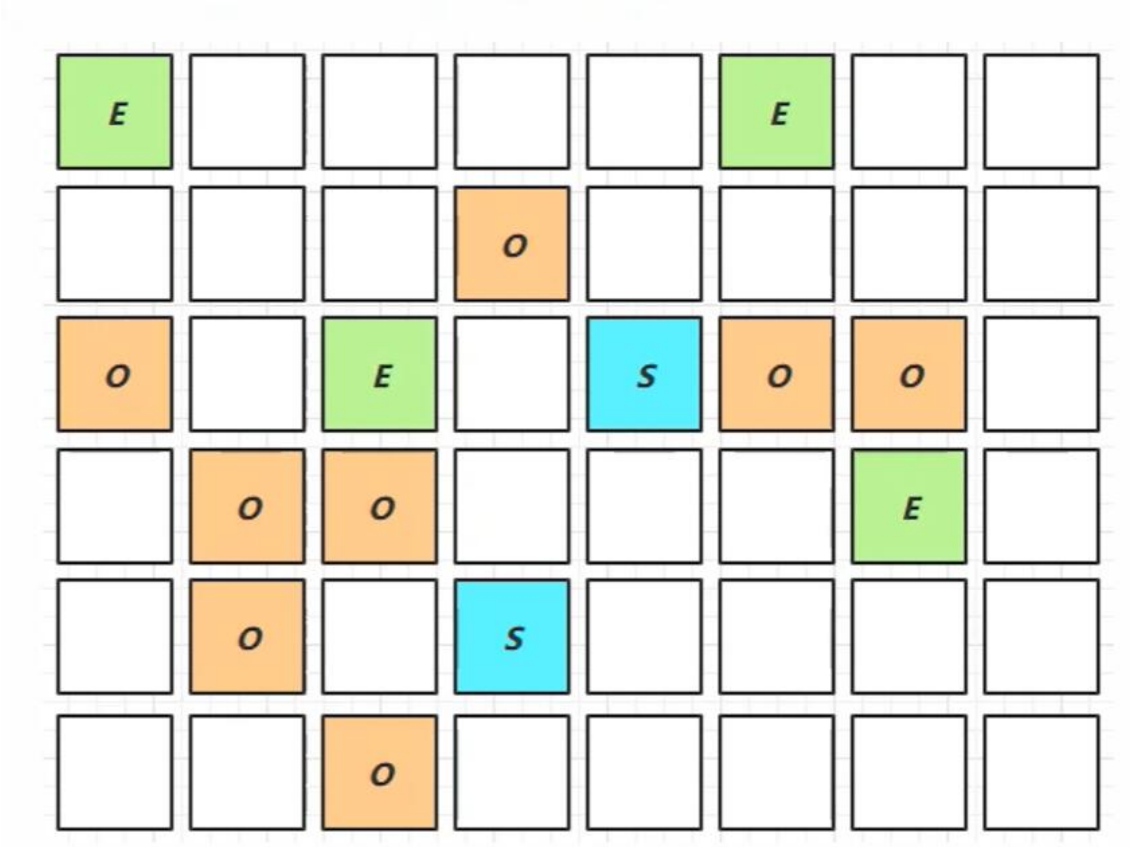






(3) Young Collection + CM

- 在Young GC时会进行GC Root的初始标记
- 老年代占用堆空间比例达到阈值时，进行并发标记(不会STW)，由下面的JVM参数决定
  - `-Xx: InitiatingHeapOccupancyPercent=percent` (默认45%)

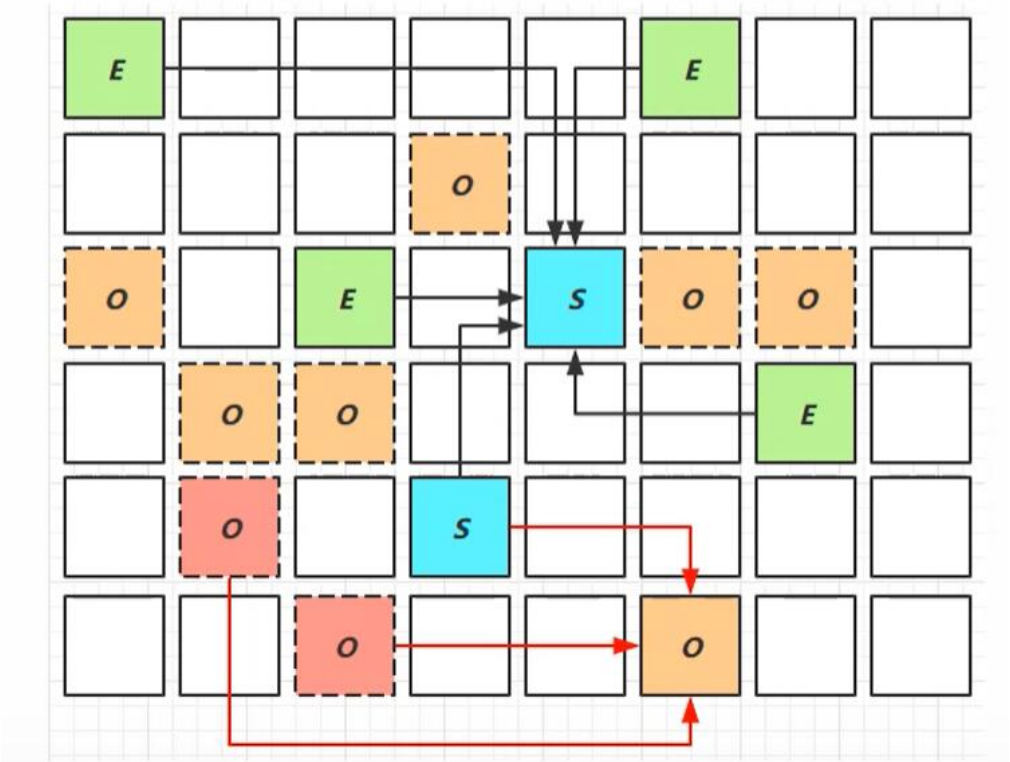


(4) Mixed Collection

会对E、S、O进行全面垃圾回收

- 最终标记(Remark) 会 STW
- 拷贝存活(Evacuation) 会 STW

`-XX :MaxGCPauseMillis=ms`



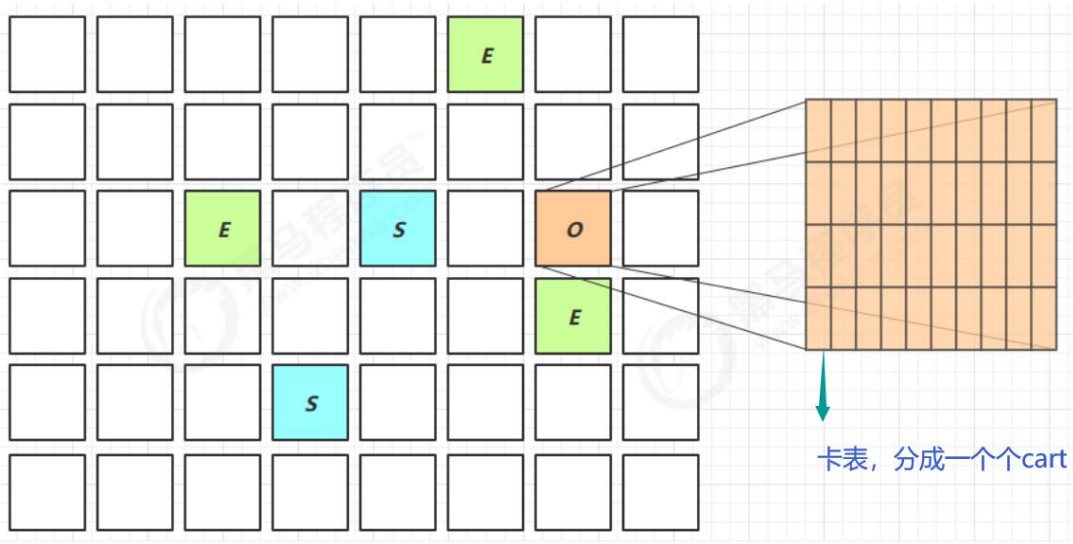
> 红色是垃圾多的部分，G1优先回收，以至使最大暂停时间尽量少

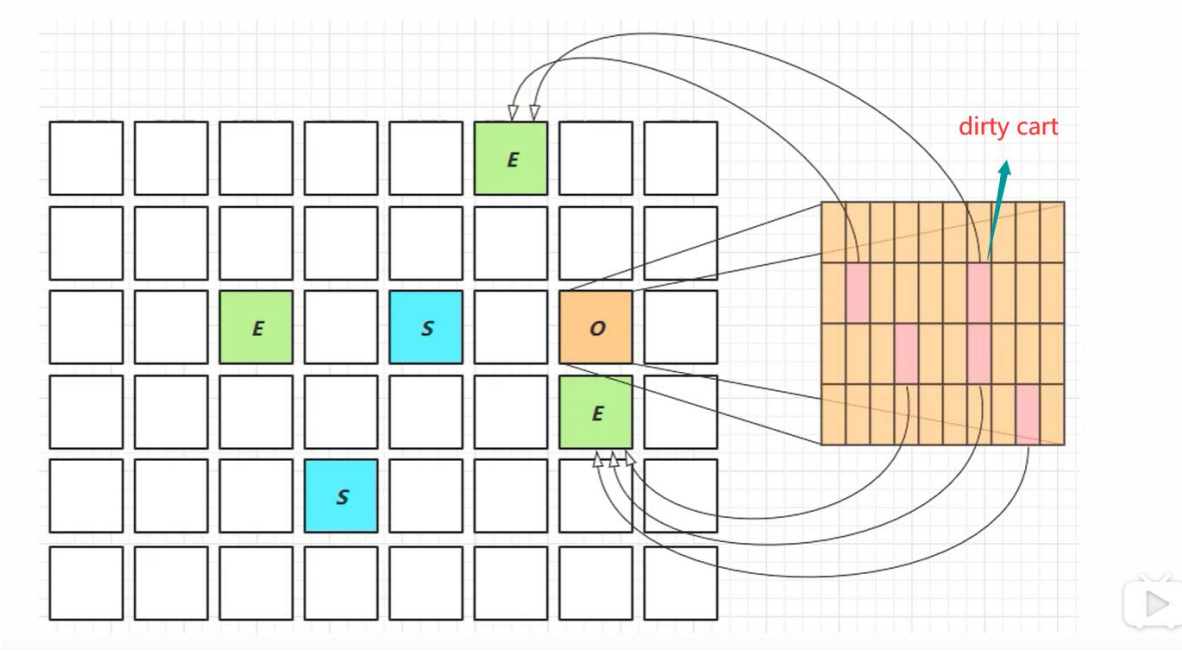
(5) Full GC

- SerialGC
  - 新生代内存不足发生的垃圾收集- minor gc
  - 老年代内存不足发生的垃圾收集- full gc
- ParallelGC
  - 新生代内存不足发生的垃圾收集- minor gc
  - 老年代内存不足发生的垃圾收集- full gc
- CMS
  - 新生代内存不足发生的垃圾收集- minor gc
  - 老年代内存不足
    - 当并发失败时=> 退化成SerialGC，执行Full GC
- G1
  - 新生代内存不足发生的垃圾收集- minor gc
  - 老年代内存不足
    - 当并发垃圾收集的速度 < 其他线程垃圾产生的速度 => 退化成SerialGC，执行Full GC

(6) Young Collection跨代引用

1. 新生代回收的跨代引用(老年代引用新生代)问题
2. 新生代垃圾回收，首先找到根对象，根对象一部分存在老年代中，根对象进行可达性分析，找到存活对象，存活对象进行复制，到幸存区
3. 若有一个cart引用了新生代对象，则标记为脏卡(`dirty cart`)
4. 之后gc root遍历时只需遍历脏cart即可，不用遍历整个老年代，提高了效率





- 卡表与 `Remembered Set`（记录incoming reference，即外部对自身的引用，即脏卡）
- 在引用变更时通过 `post-write barrier` + `dirty card queue`
- `concurrent refinement threads` 更新 `Remembered Set`

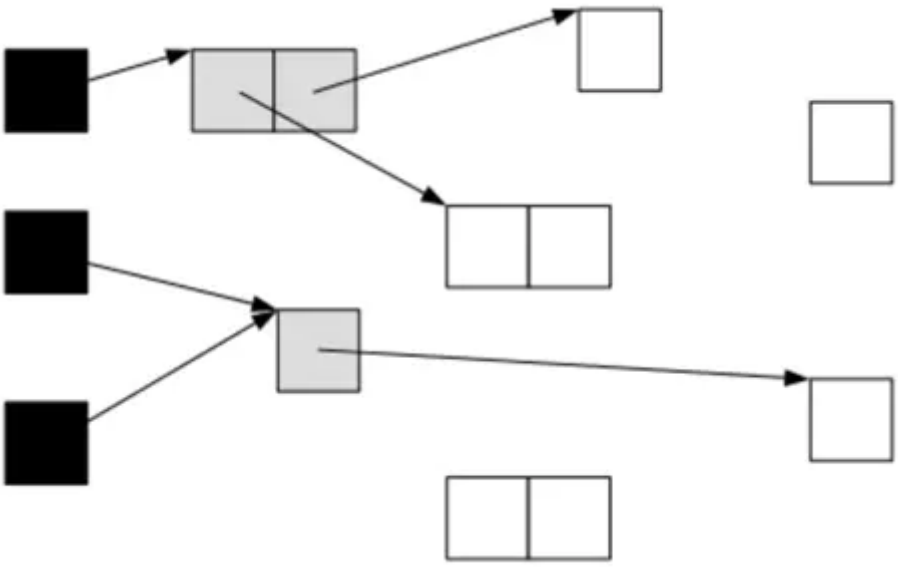
(7) Remark

- 定义：重新标记阶段
- 参数：`pre-write barrier+ satb_mark_queue`

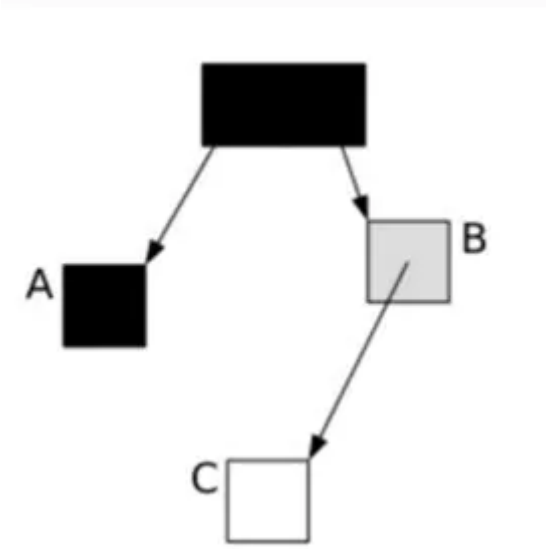
SATB的全称（Snapshot At The Beginning）字面意思是开始GC前存活对象的一个快照。SATB的作用是保证在并发标记阶段的正确性。如何理解这句话?

首先要介绍三色标记算法

图为并发标记阶段，对象的处理状态图，黑色处理完，灰色处理中，白色未处理，最终白色的没有箭头指向的（没有强引用）则不会存活



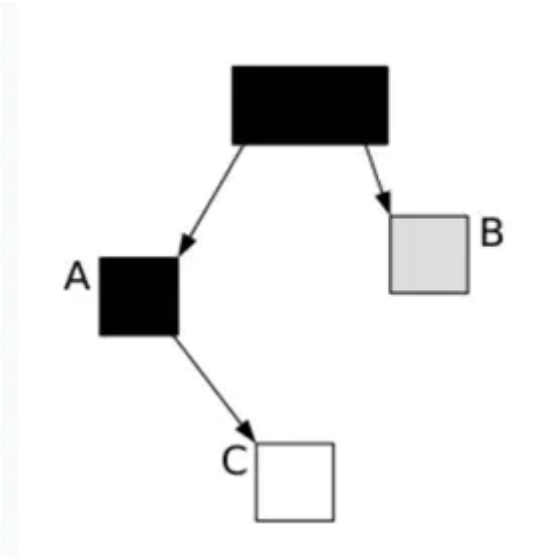
在GC扫描C之前的颜色如下：



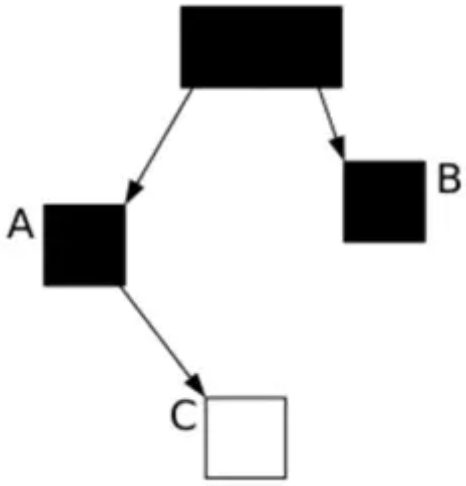
假设在并发标记阶段，应用线程改变了这种引用关系

```
A.c=C
B.c=null
```

得到如下结果

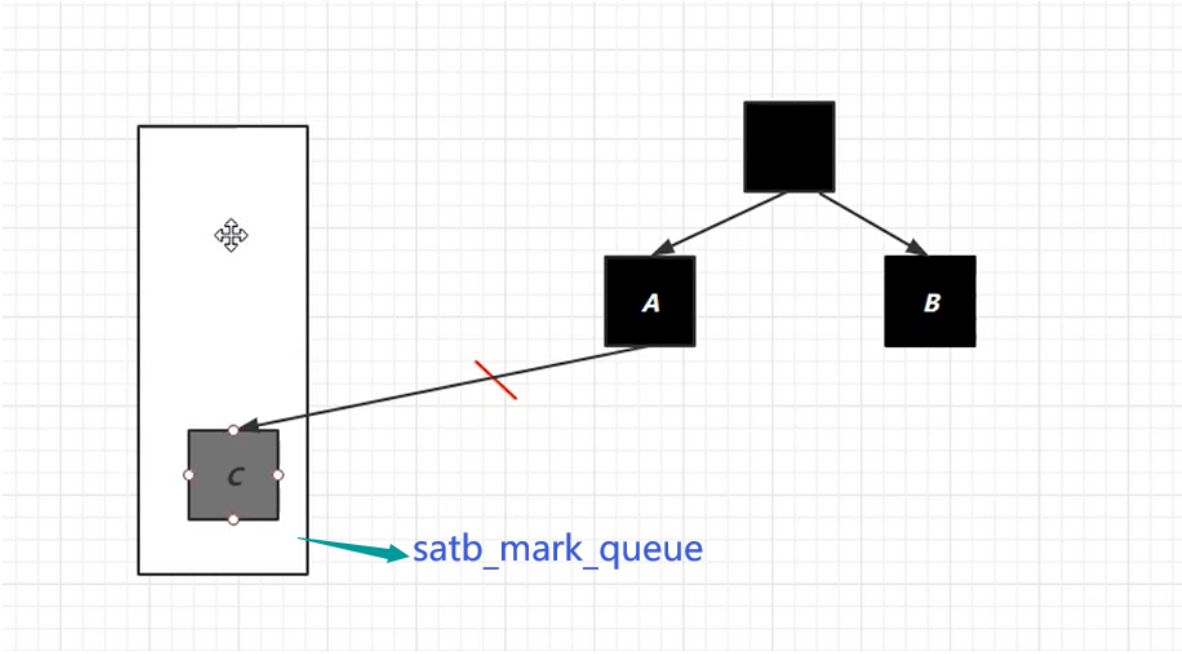


在重新标记阶段扫描后



这种情况下C会被当做垃圾进行回收。Snapshot的存活对象原来是A、B、C，现在变成A、B了，Snapshot的完整遭到破坏了，显然这个做法是不合理。

G1采用的是`pre-write barrier`（写屏障）解决这个问题。简单说就是在并发标记阶段，当引用关系发生变化的时候，通过`pre-write barrier`函数会把这种这种变化记录并保存在一个队列里，在JVM源码中这个队列叫`satb_mark_queue`。在remark阶段会扫描这个队列，通过这种方式，旧的引用所指向的对象就会被标记上，其子孙也会被递归标记上，这样就不会漏标记任何对象，snapshot的完整性也就得到了保证。



> -------剩余详细解释见如上G1帖子第一个

（8）JDK 8u20字符串去重

- 优点：节省大量内存
- 缺点：略微多占用了 cpu 时间，新生代回收时间略微增加
- 参数：`-XX:+UseStringDeduplication` 】
  - Deduplication，去除重复

```
String s1 = new String("hello"); // char[]{'h','e','l','l','o'}
String s2 = new String("hello"); // char[]{'h','e','l','l','o'}
```

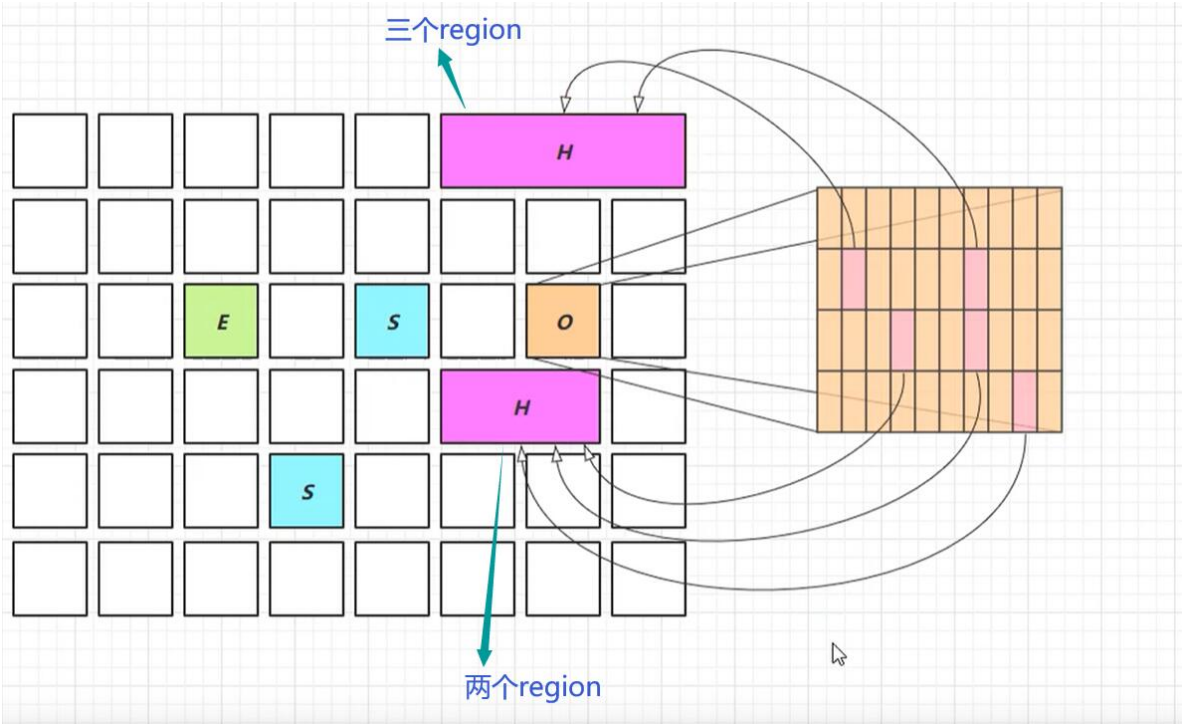
- 将所有新分配的字符串放入一个队列
- 当新生代回收时，G1并发检查是否有字符串重复
- 如果它们值一样，让它们引用同一个 char[]
  - 如上s1、s2会引用同一个char数组
- 注意，与 `String.intern()` 不一样
  - `String.intern()` 关注的是字符串对象
  - 而G1的字符串去重关注的是 `char[]`
  - 在 JVM 内部，使用了不同的字符串表

（9）JDK 8u40 并发标记类卸载

所有对象都经过并发标记后，就能知道哪些类不再被使用，当一个类加载器的所有类都不再使用，则卸
载它所加载的所有类 `-XX:+ClassUnloadingWithConcurrentMark` 默认启用

（10）JDK 8u60 回收巨型对象

定义：类比`Eden、survivor、old`区域，一个对象大于`region`的一半时，称之为巨型对象



- G1 不会对巨型对象进行拷贝
- 回收时被优先考虑
- G1 会跟踪老年代所有 incoming 引用，这样老年代 incoming 引用为0 的巨型对象就可以在新生代垃圾回收时处理掉
  - 如上图：若老年代`cart`没有对巨型对象进行引用，则巨型对象会在新生代就会被回收。

（11）JDK 9 并发标记起始时间的调整

- 并发标记必须在堆空间占满前完成，否则退化为 FullGC
- JDK 9 之前需要使用`-XX:InitiatingHeapOccupancyPercent`
- JDK 9 可以动态调整
  - `-XX:InitiatingHeapOccupancyPercent`用来设置初始值
  - 进行数据采样并动态调整
  - 总会添加一个安全的空档空间

（12）JDK 9 更高效的回收

- 250+增强
- 180+bug修复
- https://docs.oracle.com/en/java/javase/12/gctuning

3.5 垃圾回收调优

3.5.1 预备知识

- 掌握 GC 相关的 VM 参数，会基本的空间调整

> 查看虚拟机运行参数
> "C: Program Files \Javaljdk1.8.0_91\bin\java" -XX:+PrintFlagsFinal -version | findstr "GC"

- 掌握相关工具

- 明白一点：调优跟应用、环境有关，没有放之四海而皆准的法则

3.5.2 调优领域

- 内存
- 锁竞争
- cpu 占用
- io

3.5.3 确定调优目标

- 【低延迟】还是【高吞吐量】，选择合适的回收器
  - 低延迟：快速响应，适用互联网项目。
  - 高吞吐量：适用科学计算。
- CMS，G1，ZGC
  - 低延迟
  - CMS（JDK9已不推荐，JDK14完全移除，取而代之是G1）
  - ZGC（非常低延迟，体验级，JDK12引入）
- ParallelGC
- Zing
  - 超低的STW

3.5.4 最快的 GC

答案是不发生 GC

- 查看 `Full GC` 前后的内存占用，考虑下面几个问题
  - 数据是不是太多?
    - `resultSet = statement.executeQuery("select * from 大表 limit n")`
  - 数据表示是否太臃肿?
    - 对象图
    - 对象大小 16，Integer 24（对象16，包装int 4，对齐4）所以能用 int 就用int
  - 是否存在内存泄漏?
    - `static Map map =`
    - 软
    - 弱
    - 第三方缓存实现（redis等）

3.5.5 新生代调优

在排除了程序不合理造成的性能瓶颈之后，接下来就是内存调优，内存调优先从新生代开始

- 新生代的特点
  - 所有的 new 操作的内存分配非常廉价
    - `TLAB` `thread-local allocation buffer`（线程局部分配缓冲区）

      > 多个线程进行内存分配时，为了防止多个线程内存占用，所以就出现了TLAB，每个线程私有化的访问自身分配的内存
  - 死亡对象的回收代价是零
  - 大部分对象用过即死
  - Minor GC 的时间远远低于 Full GC

- 越大越好吗?

  > -Xmn Sets the **initial and maximum size (in bytes)** of the heap for the young generation (nursery).GC is performed in this region more often than in other regions. If the size for the young
  > generation is **too small, then a lot of minor garbage** collections are performed. If the size is **too**
  > **large, then only full garbage collections are performed**, which can take a long time to complete.Oracle recommends that you keep the size for the young generation greater **than 25% and less than 50%** of the overall heap size.

  - 综上文档所述，新生代尽可能大，但不能太大，一般在25%-50%
  - 新生代主要是复制算法，复制涉及内存的移动，所以耗费较长
- 合理的young内存分配：新生代能容纳所有【并发量 * (请求-响应)】的数据
  - 一次请求就可以把大部分对象都回收，使得很少触发young gc
- 合理的幸存区分配：大到能保留【当前活跃对象+需要晋升对象】
  - 幸存区可分为，将要被回收的对象、即将晋升但是年龄(阈值)不够，两者都因为被使用中所以还没被回收

- 晋升阈值配置得当，让长时间存活对象尽快晋升
  - `-XX:MaxTenuringThreshold=threshold`
  - `-XX:+PrintTenuringDistribution`

    ```
    Desired survivor size 48286924 bytes, new threshold 10 (max 10)
    - age 1: 28992024 bytes, 28992024 total
    - age 2: 1366864 bytes, 30358888 total
    - age 3: 1425912 bytes, 31784800 total
    ...
    ```

3.5.6 老年代调优

以 CMS 为例

- CMS 的老年代内存越大越好
- 先尝试不做调优，如果没有 Full GC 那么已经...，否则先尝试调优新生代
- 观察发生 Full GC 时老年代内存占用，将老年代内存预设调大 1/4 ~ 1/3
  - `-XX:CMSInitiatingOccupancyFraction=percent`

    > percent是指的留下1-percent给浮动垃圾

3.5.7 调优案例

案例1：Full GC 和 Minor GC频繁

- 现象：程序运行期间，程序GC频繁，尤其是Minor GC达到了100/min
- 原因：GC频繁，空间紧张，因为幸存区空间紧张，其对象晋升阈值降低，导致本来生存周期很短的对象晋升到老年代
- 解决：调整新生代空间大小

案例2：请求高峰期发生 Full GC，单次暂停时间特别长 （CMS）

- 原因
  - 业务需求追求低延时，所以用CMS

    - 重新标记（见3.4.3）需要扫描整个堆内存的对象，高峰期的新生代对象多，可能造成STW时间长
- 解决
    - 重新标记前先做一次垃圾回收，回收新生代对象，减少重新标记的时间（见3.4.3第四个参数）

案例3： 老年代充裕情况下，发生 Full GC （CMS jdk1.7）

- 原因
    - 1.8+是元空间作为方法区的实现；1.7及之前是永久代，永久代空间不足也会导致Full GC；
    - 1.8+元空间，其垃圾回收就不是Java组件所控制的了，元空间默认使用操作系统使用的内存空间，较为充裕
- 解决：1.7-需要增加永久代的内存空间，1.8+不会出现这个问题。