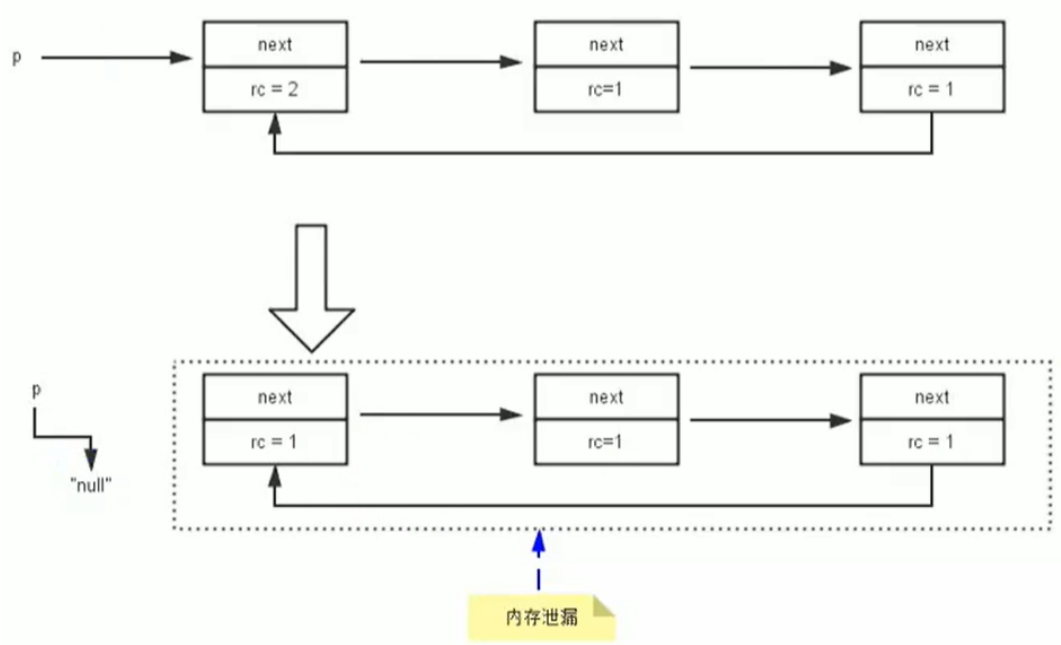
第15章 垃圾回收相关算法

1 标记阶段：引用计数算法

- 垃圾标记阶段：对象存活判断
  - 在堆里存放着几乎所有的Java对象实例，在GC执行垃圾回收之前，首先**需要区分出内存中哪些是存活对象，哪些是死亡的对象**。只有被标记为已经死亡的对象，GC才会在执行垃圾回收时，释放掉其所占的内存空间，因此这个过程我们可以称为**垃圾标记阶段**。
  - 那么在JVM中究竟是如何标记一个死亡对象呢？简单来说，当一个对象已经不再被任何存活对象继续引用时，就可以还盘为已经死亡。
  - 判断对象存活一般由两种方式：**引用计数算法**和**可达性分析算法**。

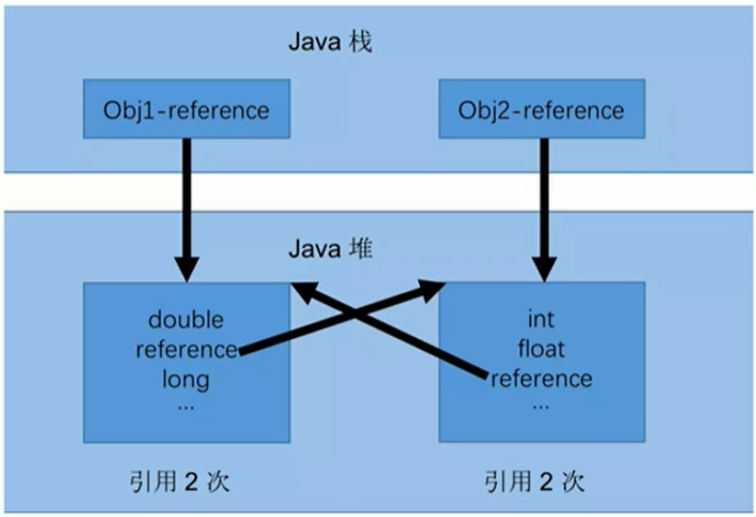
- 方式1：引用计数算法
  - 引用计数算法（Reference Counting）比较简单，对每个对象保存一个整形的**引用计数器属性。用于记录对象被引用的情况**。
  - 对于一个对象A，只要有任何一个对象引用了A，则A的引用计数器就加1；当引用失效时，引用计数器就减1.只要对象A的引用计数器的值为0，即表示对象A不可能再被使用，可进行回收。
  - 优点：**实现简单，垃圾对象便于辨识；判定效率高，回收没有延迟性**。
  - 缺点：
    - 它需要单独的字段存储计数器，这样的做法增加了**存储空间的开销**。
    - 没每次赋值都需要更新计数器，伴随着加法和减法的操作，这增加了**时间开销**。
    - 引用计数器有一个严重的问题，即无法处理**循环引用**的情况。这是一条致命缺陷，导致在Java的垃圾回收器中没有使用这类算法。



- 代码示例

```
/**
 * -XX:+PrintGCDetails
 * 证明：java使用的不是引用计数算法
 */
public class RefCountGC {
    // 这个成员属性唯一的作用就是占用一点内存
    private byte[] bigSize = new byte[5 * 1024 * 1024];  //
5MB

    Object reference = null;

    public static void main(String[] args) {
        RefCountGC obj1 = new RefCountGC();
        RefCountGC obj2 = new RefCountGC();

        obj1.reference = obj2;
        obj2.reference = obj1;

        obj1 = null;
        obj2 = null;
        // 显式的执行垃圾回收行为
        // 这里发生GC，obj1和obj2能否被回收？
        System.gc();
    }
}
```

如果采用循环引用会出现问题，如下图：



如果不下小心直接把 Obj1-reference 和 Obj2-reference 置 null。则在 Java 堆当中的两块内存依然保持着互相引用，无法回收。

如果System.gc();被注释掉，输出结果如下

```
RefCountGC
D:\Java\jdk1.8.0_231\bin\java.exe ...
Heap
 PSYoungGen      total 76288K, used 16808K [0x000000076b500000, 0x0000000770a00000, 0x00000007c0
  eden space 65536K, 25% used [0x000000076b500000,0x000000076c56a1b0,0x000000076f500000)
  from space 10752K, 0% used [0x000000076ff80000,0x000000076ff80000,0x0000000770a00000)
  to   space 10752K, 0% used [0x000000076f500000,0x000000076f500000,0x000000076ff80000)
 ParOldGen       total 175104K, used 0K [0x00000006c1e00000, 0x00000006cc900000, 0x000000076b500
  object space 175104K, 0% used [0x00000006c1e00000,0x00000006c1e00000,0x00000006cc900000)
 Metaspace       used 3145K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 343K, capacity 388K, committed 512K, reserved 1048576K
```

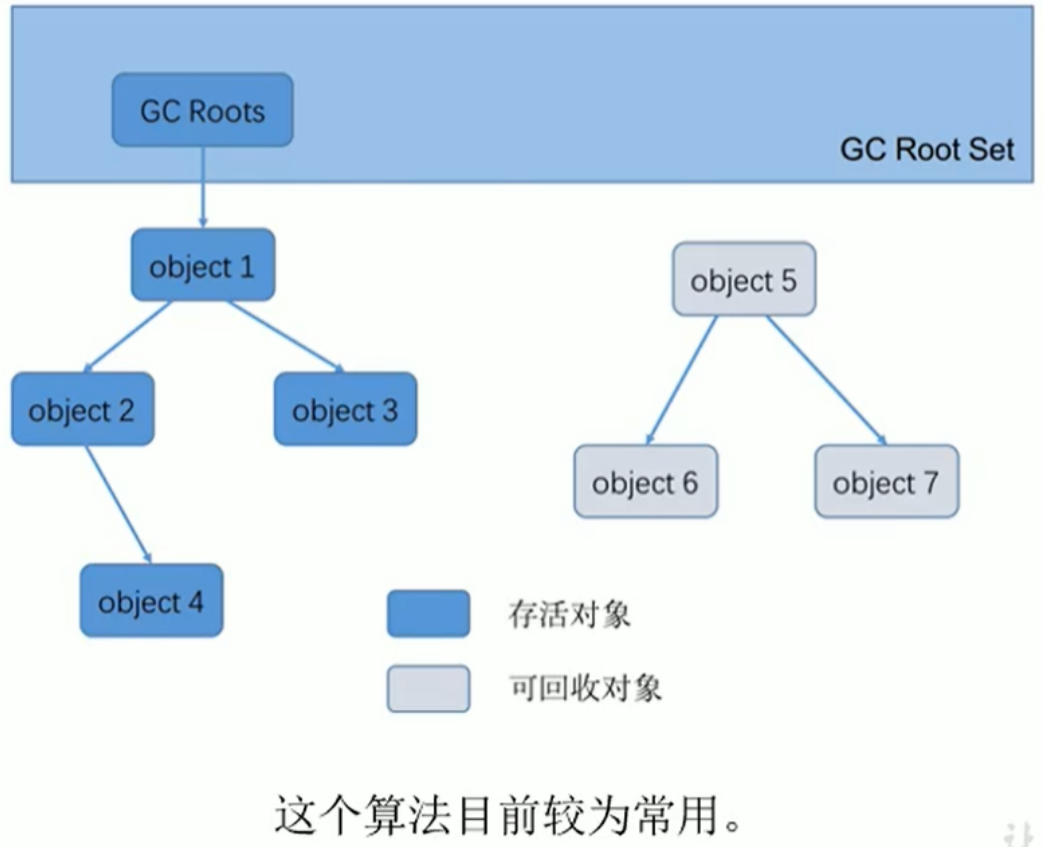
如果System.gc();没被注释掉，输出结果如下

```
RefCountGC
D:\Java\jdk1.8.0_231\bin\java.exe ...
[GC (System.gc()) [PSYoungGen: 15497K->712K(76288K)] 15497K->720K(251392K), 0.0010160 secs] [Tim
[Full GC (System.gc()) [PSYoungGen: 712K->0K(76288K)] [ParOldGen: 8K->601K(175104K)] 720K->601K(
Heap
 PSYoungGen      total 76288K, used 1966K [0x000000076b500000, 0x0000000770a00000, 0x00000007c00
  eden space 65536K, 3% used [0x000000076b500000,0x000000076b6eb9e0,0x000000076f500000)
  from space 10752K, 0% used [0x000000076f500000,0x000000076f500000,0x000000076ff80000)
  to   space 10752K, 0% used [0x000000076ff80000,0x000000076ff80000,0x0000000770a00000)
 ParOldGen       total 175104K, used 601K [0x00000006c1e00000, 0x00000006cc900000, 0x000000076b5
  object space 175104K, 0% used [0x00000006c1e00000,0x00000006c1e964a8,0x00000006cc900000)
 Metaspace       used 3224K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 349K, capacity 388K, committed 512K, reserved 1048576K
```

- 小节
  - 引用计数算法，是很多语言的资源回收选择，例如隐人工智能而更加火热的Python，它更是同时支持引用计数和垃圾收集机制。
  - 具体哪种最优是要看场景的，业界有大规模时间中仅保留引用技术机制，以提高吞吐量的尝试。
  - Java并没有选择引用计数，是因为其存在一个基本难题，也就是很难处理循环引用关系。
  - Python如何解决循环引用？
    - 手动解除：很好理解，就是在合适的实际，解除引用关系。
    - 使用弱引用weakref，weakref是Python提供的标准，只在解决循环引用。

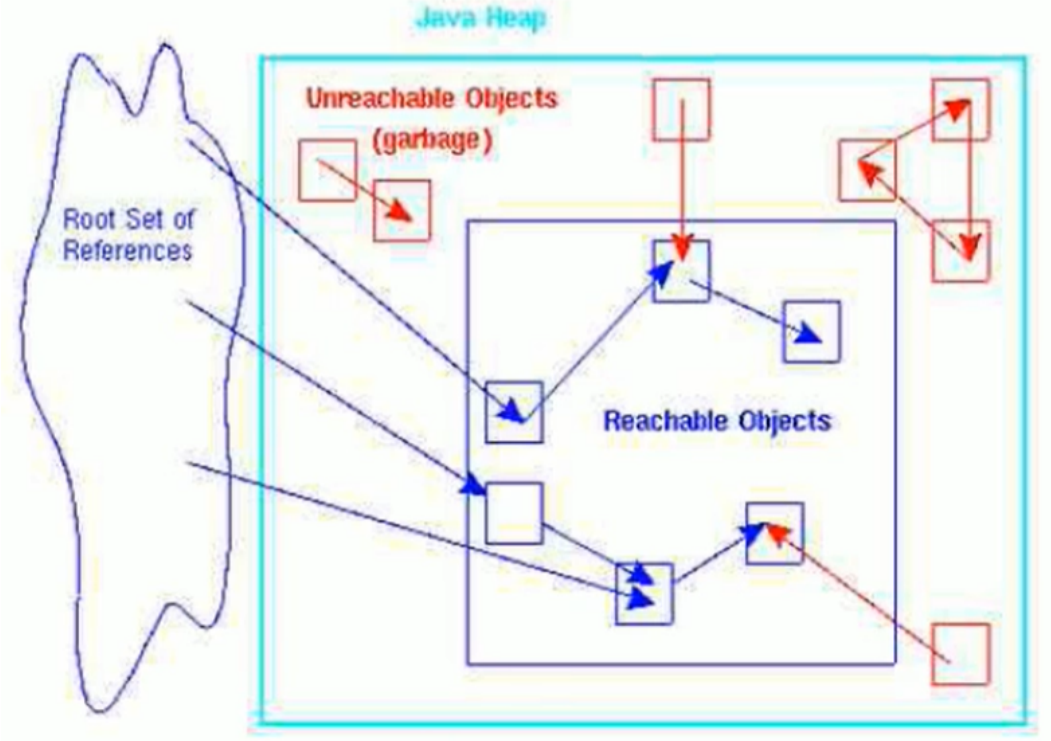
2 标记阶段：可达性分析算法

- 方式二：可达性分析（或跟搜索算法、追踪性垃圾收集）
  - 相对于引用计数算法而言，可达性分析算法不仅同样具备实现简单和执行高效等特点，更重要的是该算法可以有效地**解决在引用计数算法中循环引用的问题，防止内存泄露的发生**。
  - 相对于引用计数算法，这里的可达性分析就是**Java、C#**选择的。这种类型的垃圾收集通常也叫做**追踪性垃圾收集**（Tracing Garbage Collection）。
  - 所谓“GC Roots”根集合就是一组必须活跃的引用。
  - 基本思路：
    - 可达性分析算法是以根集合（GC Roots）为起始点，按照从上至下的方式**搜索被根对象集合所连接的目标是否可达**。
    - 使用可达性分析算法后，内存中存活对象都会被根对象集合直接或间接连接着，搜索走过的路径称为**引用链（Reference Chain）**。
    - 如果目标对象没有任何引用链相连，则是不可达的，就意味着该对象已经死亡，可以标记为垃圾对象。
    - 在可达性分析算法中，只有能被根对象集合直接或者间接连接的对象才是存活对象。



- 在Java语言中，GC Roots包括以下几类元素：
  - 虚拟机栈中引用的对象
    - 比如：各个线程被调用的方法中使用到的参数、局部变量等。
  - 本地方法栈JNI（通常说的本地方法）引用的对象
  - 方法区中静态属性引用的对象

        - 比如：Java类的引用类型静态变量
    - 方法区中常量引用的对象
        - 比如：字符串常量池（StringTable）里的引用
    - 所有被同步锁synchronized持有的对象
    - Java虚拟机内部的引用。
        - 基本数据类型对应的Class对象，一些常驻的异常对象（如：NullPointerException、OutOfMemoryError），系统类加载器。
    - 反应java虚拟机内部情况的JMXBean、JVMTI中注册的回调、本地代码缓存等。



    - 除了这些固定的GC Roots集合以外，根据用户选择的垃圾收集器以及当前回收的内存区域不同，还可以有其他对象“临时性”地加入，共同构成完整GC Roots集合，比如：分代收集和局部回收（Partial GC）。
        - 如果只针对Java堆中某一块区域进行垃圾回收（比如：典型的只针对新生代），必须考虑到内存区域是虚拟机字节的实现细节，更不是鼓励封闭的，这个区域的对象完全有可能被其他区域对象的对象所引用，这时候就需要一并将关联的区域对象也加入GC Roots集合中去考虑，才能保证可达性分析的准确定。
- 小技巧：
    - 由于Root采用栈方式存放变量和指针，所以如果一个指针，它保存了堆内存里面的对象，但是自己又不存放在堆内存里面，那么它就是一个Root。

- 注意：
    - 如果要使用可达性分析算法来判断内存是否可回收，那么分析工作必须在一个能保证一致性的快照中进行。这点不满足的话分析结果的准确性就无法保证。
    - 这点也是导致GC进行时必须“Stop The World”的一个重要原因。

    - 即使是号称（几乎）不会发生停顿的CMS收集器中，**枚举根节点时也是必须要停顿的**。

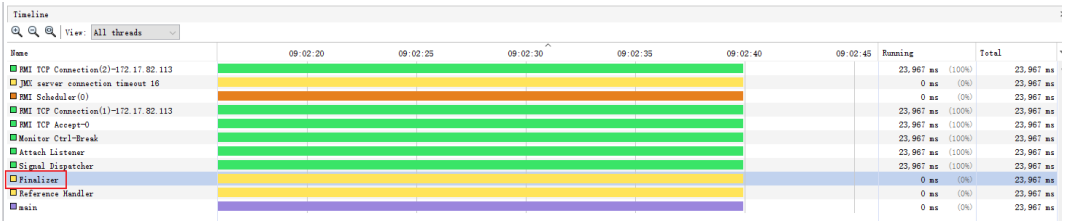
3 对象的finalization机制

- Java语言提供了对象终止（finalization）机制来允许开发人员提供能**对象被销毁之前的自定义处理逻辑**。
- 当垃圾回收器发现没有引用指向一个对象，即：垃圾回收此对象之前，总会先调用这个对象的finalize()方法。
- finalize()方法允许在子类中被重写，**用于在对象被回收之前进行资源释放**。通常在这个方法中进行一些资源释放和清理工作，比如关闭文件、套接字和数据库连接等。
- 永远不要主动调用某个对象的finalize()方法，应该交给垃圾回收机制调用。理由包括下面三点：
    - 在finalize()时可能会导致对象复活。
    - finalize()方法的执行时没有时间保障的，它完全由GC线程决定，极端情况下，若不发生GC，则finalize()方法将没有执行机会。
    - 一个糟糕的finalize()会严重影响GC的性能。
- 从功能上来说，finalize()方法与C++中的析构函数比较相似，但是Java采用的是基于垃圾回收器的自动内存管理机制，所以finalize()方法在本质上不同于C++中的析构函数。

- 由于finalize()方法的存在，**虚拟机中的对象一般处于三种可能的状态**。
    - 如果从所有的根节点都无法访问到某个对象，说明对象已经不再使用了。一般来说，次对象需要被回收，但事实上，也并不是“非死不可”的，这时候他们展示处于“缓刑”阶段。**一个无法触及的对象有可能在某一个条件下“复活”自己**，如果这样，那么对它的回收就是不合理的，为此，定义虚拟机中的对象可能的三种状态。如下：
        - **可触及的**：从根节点开始，可以到达这个对象。
        - **可复活的**：对象的所有引用都被释放，但是对象有可能在finalize()中复活。
        - **不可触及的**：对象的finalize()被调用，并且没有复活，那么就会进入不可触及状态。不可触及的对象不可能被复活，因为**finalize()只会被调用一次**。
    - 以上三种状态中，是由于finalize()方法的存在，进行的区分。只有在对象不可触及是才可能被回收。
- 回收对象的具体过程：判断一个对象objA是否可回收，至少要经历两次编辑过程：
    1. 如果对象objA到GC Roots没有引用链，则进行第一次标记。
    2. 进行筛选，判断次对象是否有必要执行finalize()方法

       ① 如果对象objA没有重写finalize()方法，或者finalize()方法已经被虚拟机调用过，则虚拟机将会把finalize()方法视为“没有必要执行”，objA被判定为不可触及的。

② 如果对象onjA重写了finalize()方法，且还未执行过，那么objA会被插入F-Queue队列中，由一个虚拟机自动创建的、低优先级的Finalizer线程触发finalize()方法执行。



③ **finalize()方法是对象逃脱死亡的最后机会**，稍后GC会对F-Queue队列中的对象进行第二次标记。**如果objA在finalize()方法中与引用链上的任何一个对象建立了联系**，那么在第二次标记时，objA会被移出“即将回收”集合。之后，对象会再次出现没有引用的情况。这种情况下，finalize()方法不会再次被调用，对象会直接编程不可触及的状态，也就是说，一个对象的finalize()方法只会被调用一次。

- 代码演示

```
/**
 * 测试Object类中finalize()方法，即对象的finalization机制。
 */
public class CanReliveObj {
    public static CanReliveObj obj;  // 类变量，属于 GC Root

    // 此方法只能被调用一次
    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("调用当前类重写的finalize()方法");
        obj = this;  // 当前待回收的对象在finalize()方法中与引用链上的一个对象obj建立了联系
    }

    public static void main(String[] args) {
        try {
            obj = new CanReliveObj();
            // 对象第一次成功拯救自己
            obj = null;
            System.gc();  // 调用垃圾回收器
            System.out.println("第1次 gc");
            // 因为Finalizer线程优先级很低，暂停2秒，以等待它
            Thread.sleep(2000);
            if (obj == null) {
                System.out.println("obj is dead");
            } else {
                System.out.println("obj is still alive");
            }
            System.out.println("第2次 gc");
```

```
            // 下面这段代码与上面的完全相同，但是这次自救却失败了
            obj = null;
            System.gc();
            // 因为Finalizer线程优先级很低，暂停2秒，以等待它
            Thread.sleep(2000);
            if (obj == null) {
                System.out.println("obj is dead");
            } else {
                System.out.println("obj is still alive");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

结果：

调用当前类重写的finalize()方法

第1次 gc

obj is still alive

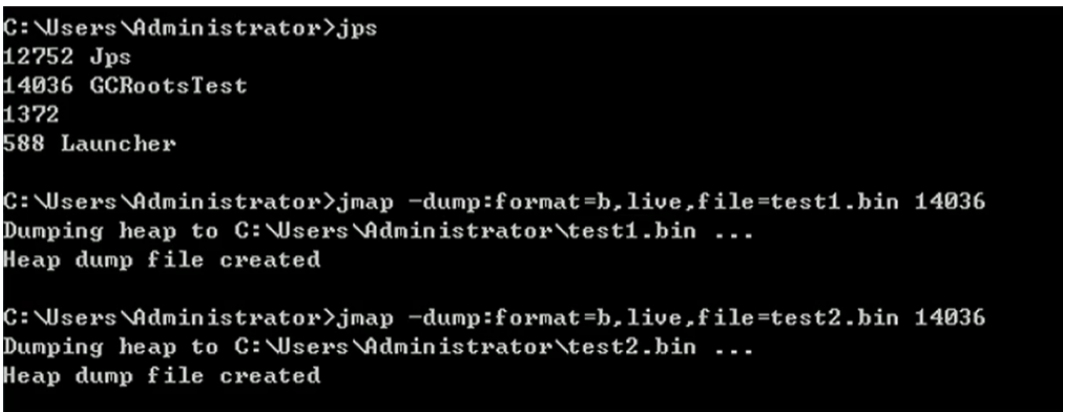
第2次 gc

obj is dead

4 MAT与JProfiler的GC Roots溯源

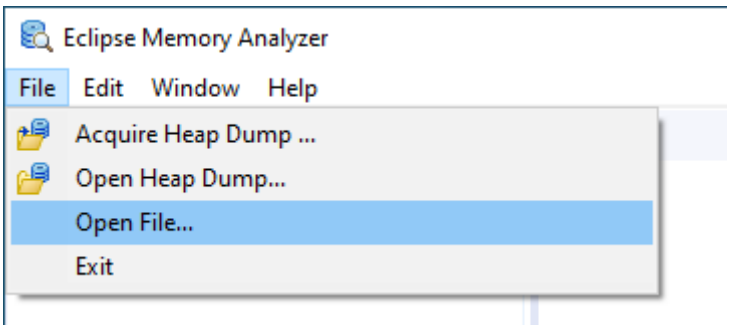
- MAT简介
  - MAT是Memory Analyzer的简称，它是一款功能强大的Java堆内存分析器。用于查找内存泄露以及查看内存消耗情况。
  - MAT是基于Eclipse开发的，是一款免费的性能分析工具。
  - 大家可以在http://www.eclipse.org/mat/下载并使用MAT。

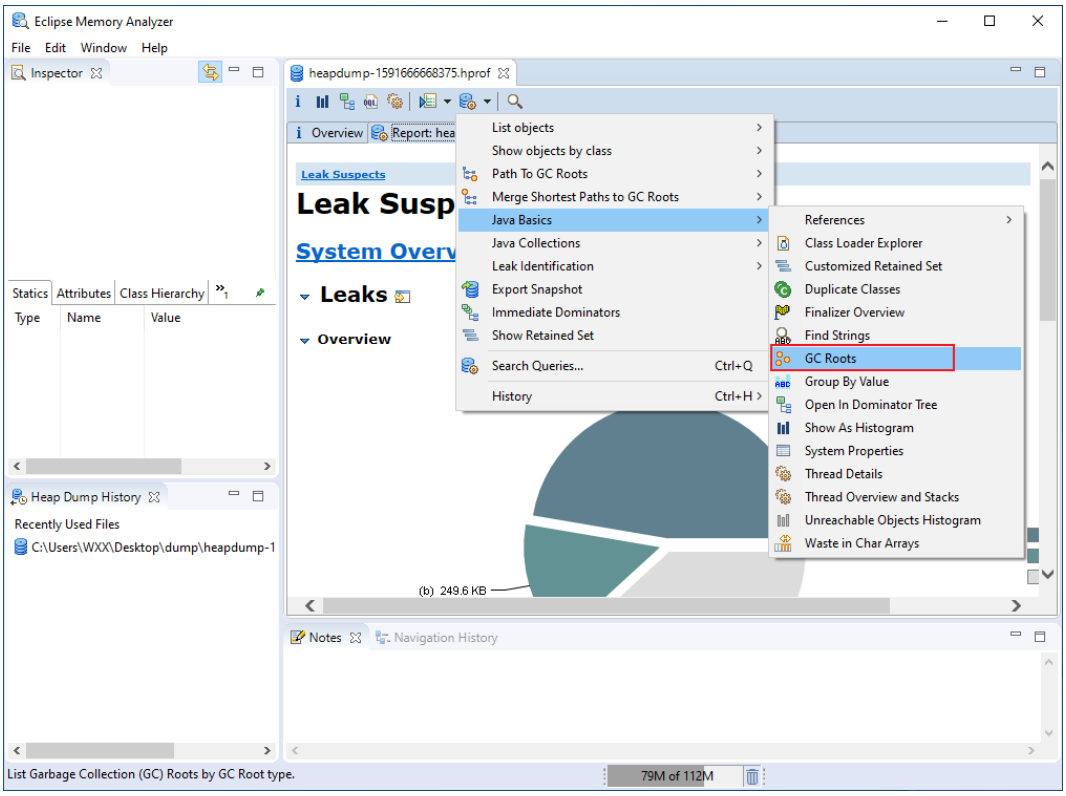
- 获取dump文件
  - 方式1：命令行使用jmap

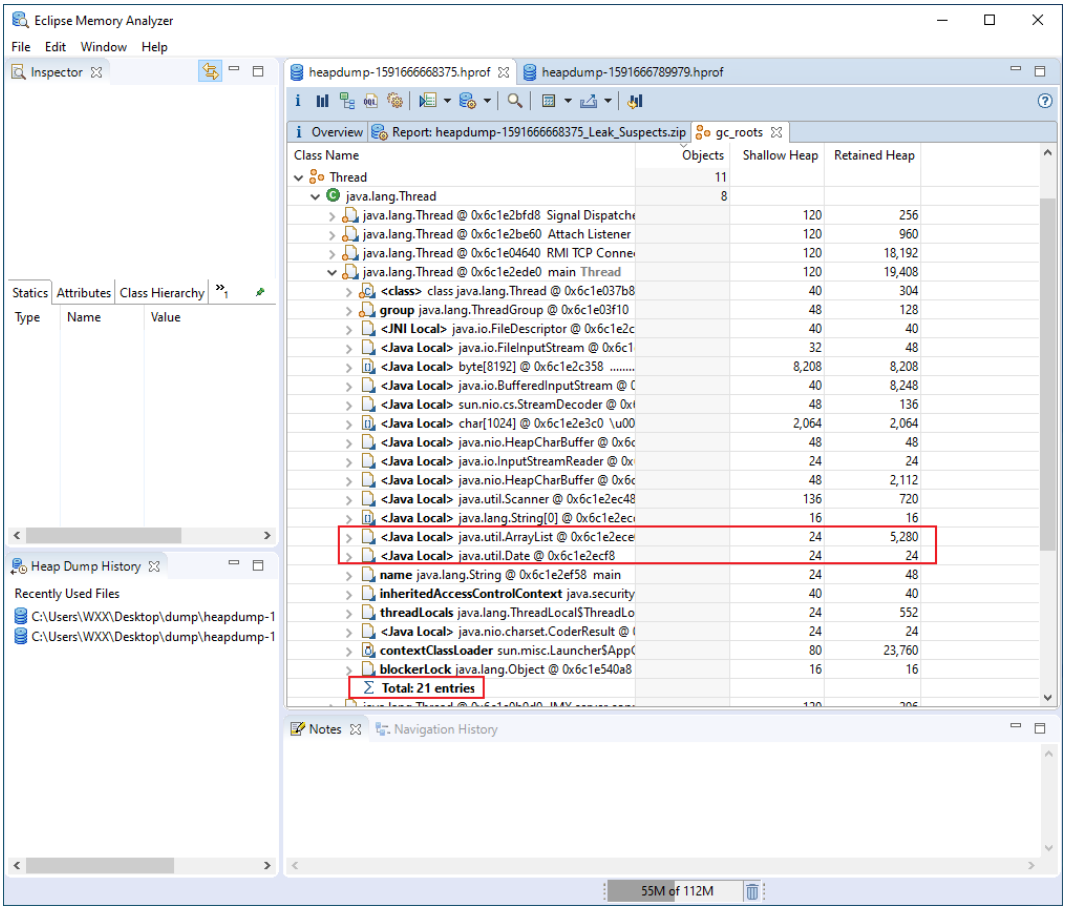


- 方式2：使用JVisualVM导出
  - 捕获的heap dump文件是一个临时文件，关闭JVisualVM后自动删除，若要保留，需要将其另存为文件。
  - 可通过以下方法捕获heap dump：
    - 在左侧“Application”（应用程序）子窗口中左击相应的应用程序，选择Heap Dump（堆DUmp）。
    - 在Monitor（监视）字标签页中点击Heap Dump（堆Dump）按钮。
  - 本地应用程序的Heap dumps作为应用程序标签页的一个标签页打开。同时，heap dump在左侧的Application（应用程序）栏中对应一个含有时间戳的节点。右击这个节点的save as（另存为）即可将heap dump保存到本地。
- dump文件分析

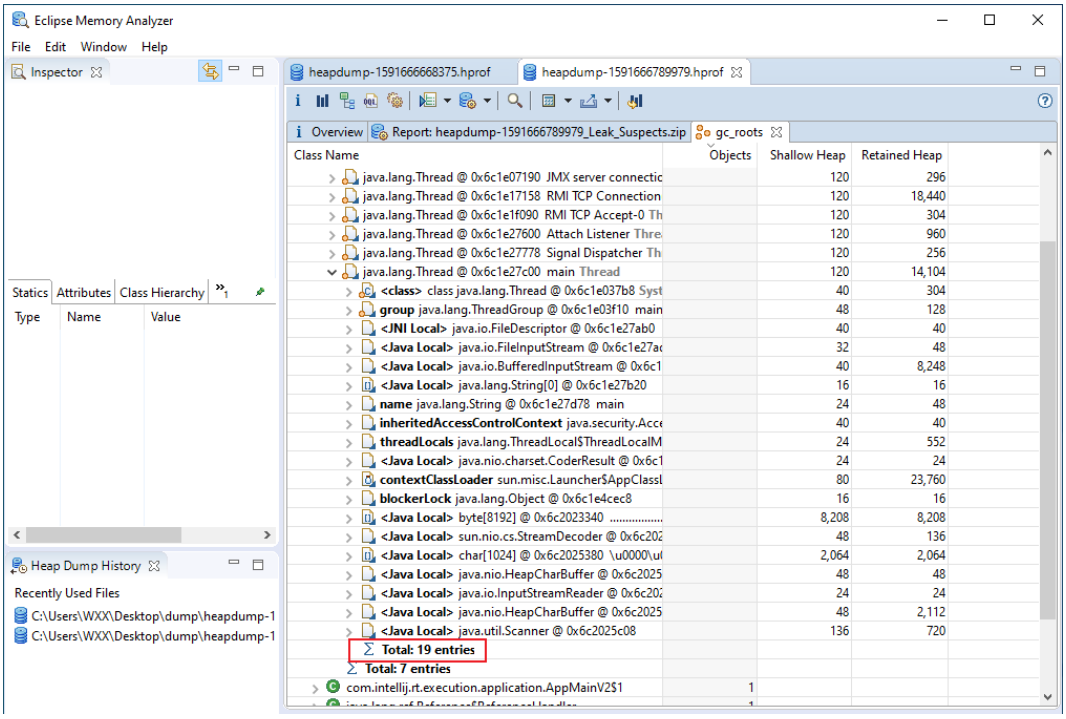
```
public class GCRootsTest {
    public static void main(String[] args) {
        List<Object> numList = new ArrayList<>();
        Date birth = new Date();

        for (int i = 0; i < 100; i++) {
            numList.add(String.valueOf(i));
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("数据添加完毕，请操作：");
        new Scanner(System.in).next();
        numList = null;
        birth = null;

        System.out.println("numList、birth已置空，请操作：");
        new Scanner(System.in).next();

        System.out.println("结束");
    }
}
```


Eclipse Memory Analyzer
File
Edit
Window
Help
Acquire Heap Dump ...
Open Heap Dump...
Open File...
Exit

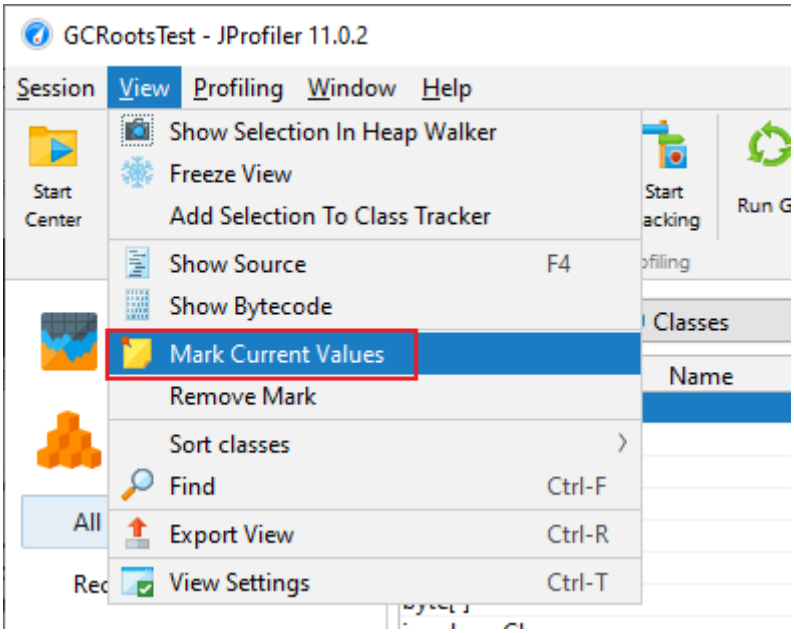

Eclipse Memory Analyzer
File Edit Window Help
Inspector
heapdump-1591666668375.hprof
Overview
Report: hea
Leak Suspects
Leak Susp
System Overv
Leaks
Overview
List objects
Show objects by class
Path To GC Roots
Merge Shortest Paths to GC Roots
Java Basics
Java Collections
Leak Identification
Export Snapshot
Immediate Dominators
Show Retained Set
Search Queries... Ctrl+Q
History Ctrl+H
References
Class Loader Explorer
Customized Retained Set
Duplicate Classes
Finalizer Overview
Find Strings
GC Roots
Group By Value
Open In Dominator Tree
Show As Histogram
System Properties
Thread Details
Thread Overview and Stacks
Unreachable Objects Histogram
Waste in Char Arrays
(b) 249.6 KB
Statics
Attributes
Class Hierarchy
Type
Name
Value
Heap Dump History
Recently Used Files
C:\Users\WXX\Desktop\dump\heapdump-1
Notes
Navigation History
List Garbage Collection (GC) Roots by GC Root type.
79M of 112M

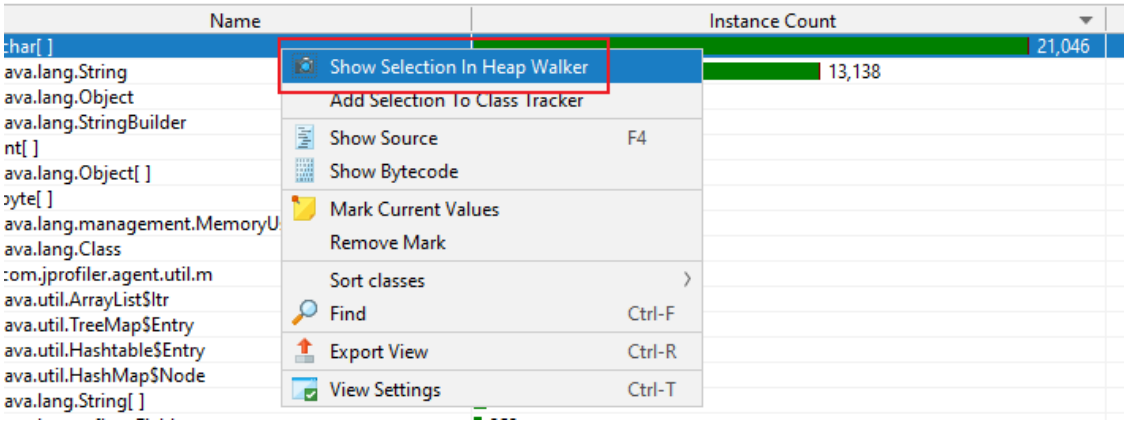

Eclipse Memory Analyzer
File Edit Window Help
Inspector
heapdump-1591666668375.hprof
heapdump-1591666789979.hprof
Overview
Report: heapdump-1591666668375_Leak_Suspects.zip
gc_roots
Class Name
Objects
Shallow Heap
Retained Heap
Thread 11
java.lang.Thread 8
java.lang.Thread @ 0x6c1e2bfd8 Signal Dispatche 120 256
java.lang.Thread @ 0x6c1e2be60 Attach Listener 120 960
java.lang.Thread @ 0x6c1e04640 RMI TCP Conne 120 18,192
java.lang.Thread @ 0x6c1e2ede0 main Thread 120 19,408
<class> class java.lang.Thread @ 0x6c1e037b8 40 304
group java.lang.ThreadGroup @ 0x6c1e03f10 48 128
<JNI Local> java.io.FileDescriptor @ 0x6c1e2c 40 40
<Java Local> java.io.FileInputStream @ 0x6c1 32 48
<Java Local> byte[8192] @ 0x6c1e2c358 ..... 8,208 8,208
<Java Local> java.io.BufferedInputStream @ 0 40 8,248
<Java Local> sun.nio.cs.StreamDecoder @ 0x 48 136
<Java Local> char[1024] @ 0x6c1e2e3c0 \u00 2,064 2,064
<Java Local> java.nio.HeapCharBuffer @ 0x6c 48 48
<Java Local> java.io.InputStreamReader @ 0x 24 24
<Java Local> java.nio.HeapCharBuffer @ 0x6c 48 2,112
<Java Local> java.util.Scanner @ 0x6c1e2ec48 136 720
<Java Local> java.lang.String[0] @ 0x6c1e2ec 16 16
<Java Local> java.util.ArrayList @ 0x6c1e2ece 24 5,280
<Java Local> java.util.Date @ 0x6c1e2ecf8 24 24
name java.lang.String @ 0x6c1e2ef58 main 24 48
inheritedAccessControlContext java.security 40 40
threadLocals java.lang.ThreadLocal$ThreadLo 24 552
<Java Local> java.nio.charset.CoderResult @ 24 24
contextClassLoader sun.misc.Launcher$App 80 23,760
blockerLock java.lang.Object @ 0x6c1e540a8 16 16
Total: 21 entries
Statics
Attributes
Class Hierarchy
Type
Name
Value
Heap Dump History
Recently Used Files
C:\Users\WXX\Desktop\dump\heapdump-1
C:\Users\WXX\Desktop\dump\heapdump-1
Notes
Navigation History
55M of 112M

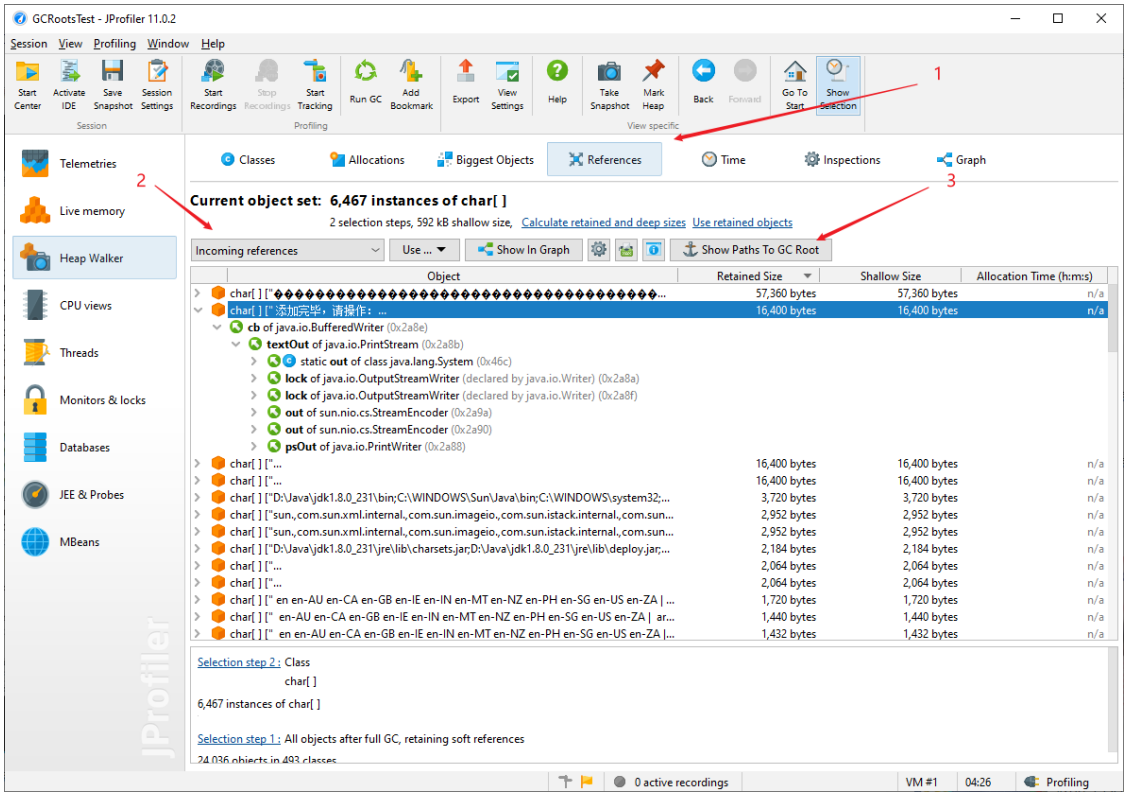
置空numList和birth后通过分析dump文件可以看到ArrayList和Date没有了



- 使用JProfiler查找某对象的引用链（GC 溯源）







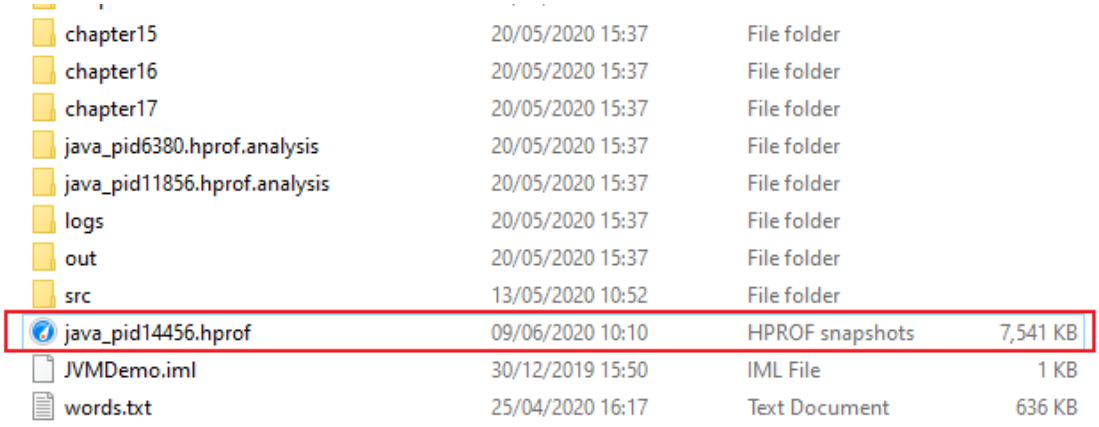
- 使用JProfiler分析OOM

```
/**
 * -Xms8m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError
 */
public class HeapOOM {
    byte[] buffer = new byte[1 * 1024 * 1024];  // 1MB

    public static void main(String[] args) {
        ArrayList<HeapOOM> list = new ArrayList<>();

        int count = 0;
        try{
            while(true){
                list.add(new HeapOOM());
                count++;
            }
        }catch (Throwable e){
            System.out.println("count = " + count);
            e.printStackTrace();
        }
    }
}
```

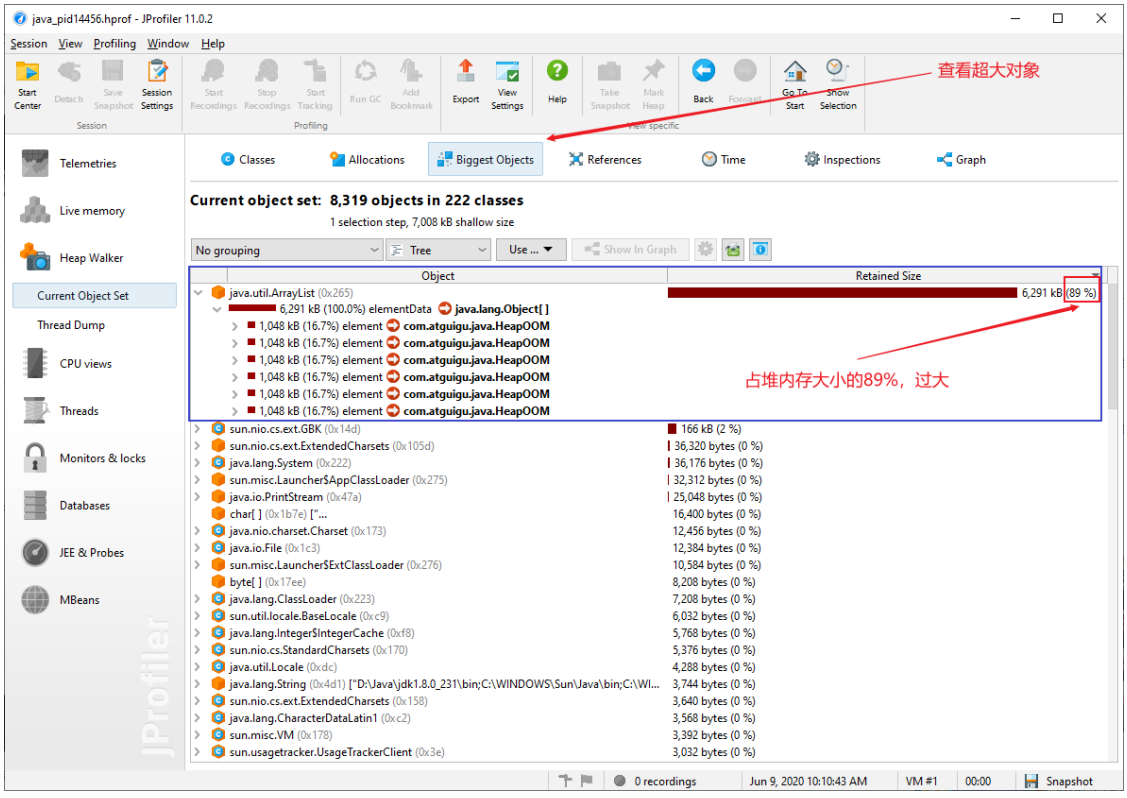
HeapOOM

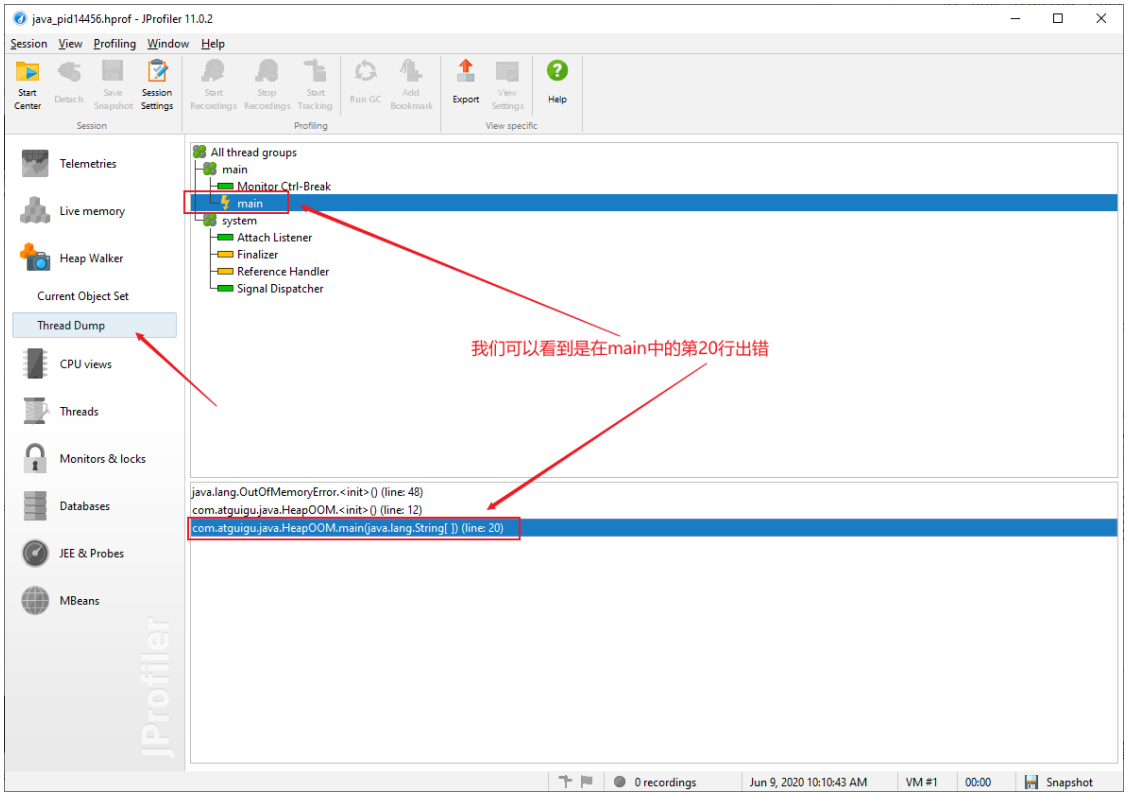
```
D:\Java\jdk1.8.0_231\bin\java.exe ...
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid14456.hprof ...
Heap dump file created [7721316 bytes in 0.117 secs]
count = 6
java.lang.OutOfMemoryError: Java heap space
    at com.atguigu.java.HeapOOM.<init>(HeapOOM.java:12)
    at com.atguigu.java.HeapOOM.main(HeapOOM.java:20)
```

-XX:+HeapDumpOnOutOfMemoryError：出现OOM是会生成Dump文件



双击用JProfiler打开此文件

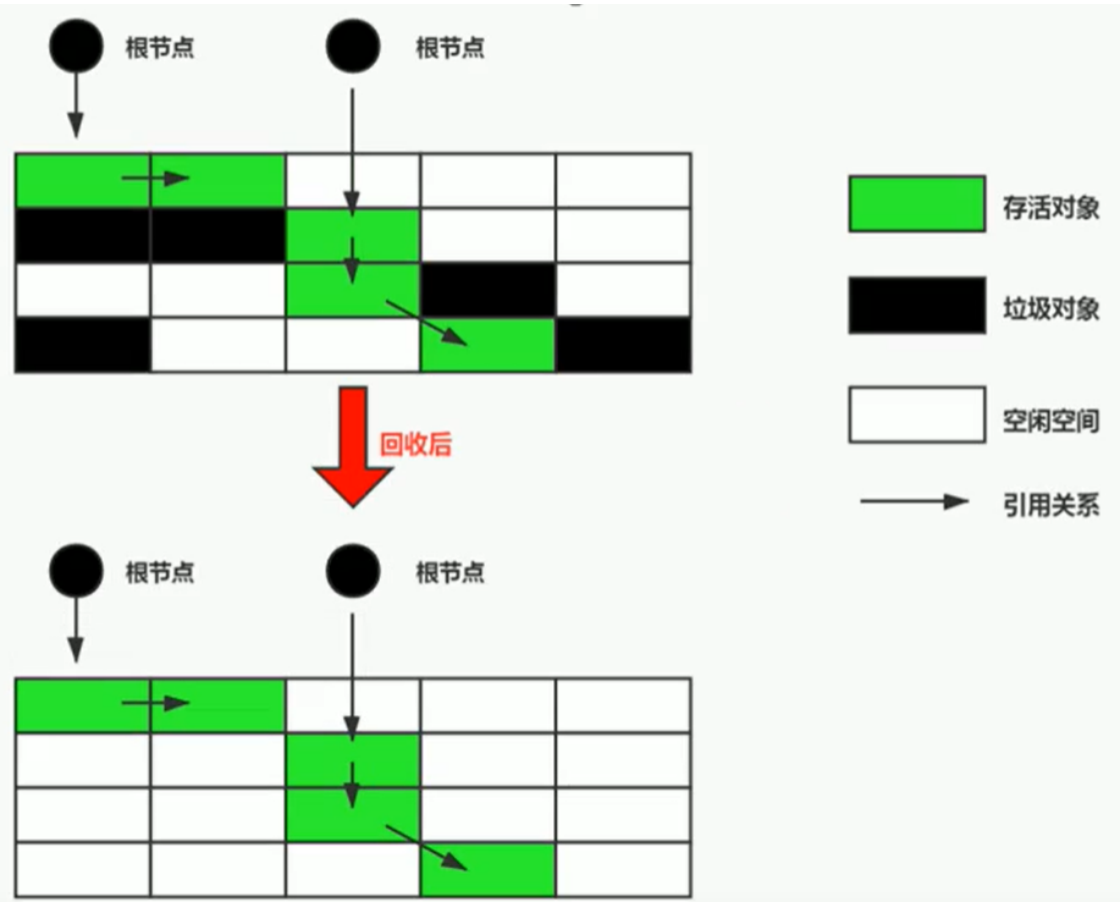




5 清除阶段：标记-清除算法

- 垃圾清除阶段
  - 当成功区分出内存中存活对象和死亡对象后，GC接下来的任务就是执行垃圾回收，释放掉无用对象所占用的内存空间，以便有足够的可用内存空间为新对象分配内存。
  - 目前在JVM中比较常见的三种垃圾收集算法是**标记-清除算法（Mark-Sweep）、复制算法（Copying）、标记-压缩算法（Mark-Compact）**。

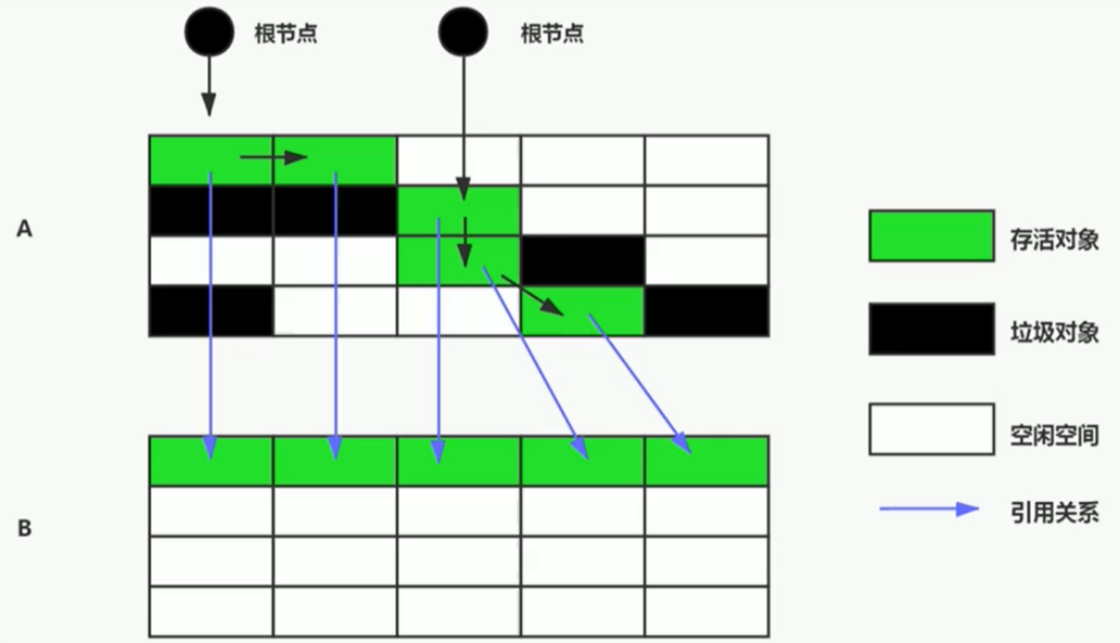
- 标记-清除（Mark-Sweep）算法
  - 背景：标记-清除（Mark-Sweep）算法是一种非常基础和常见的垃圾收集算法，该算法被J.McCarthy等人在1960年提出并应用于Lisp语言。
  - 执行过程：当中的有效内存空间（avaiable memory）被耗尽的时候，就会停止整个程序（也被称为stop the world），然后进行两项工作，第一项是标记，第二项是清除。
    - **标记**：Collector从引用根节点开始比那里，标记所有被引用的对象。一般是在对象的Header中记录为可达对象。
    - **清除**：Collector对堆内存从头到尾进行线性的遍历，如果发现某个对象在Header中没有标记为可达对象，则将其回收。



- 缺点
  - 效率不算高
  - 在进行GC的时候，需要停止整个应用程序，导致用户体验差
  - 这种方式清理出来的空闲内存是不连续的，产生内存碎片。需要维护一个空闲列表，大对象可能放不下
- 注意：何为清除？
  - 这里所谓的清除并不是真的置空，而是把需要清除的对象地址保存在空闲的地址列表中。下次有新对象需要加载时，判断垃圾的位置空间是否够，如果够，就存放。

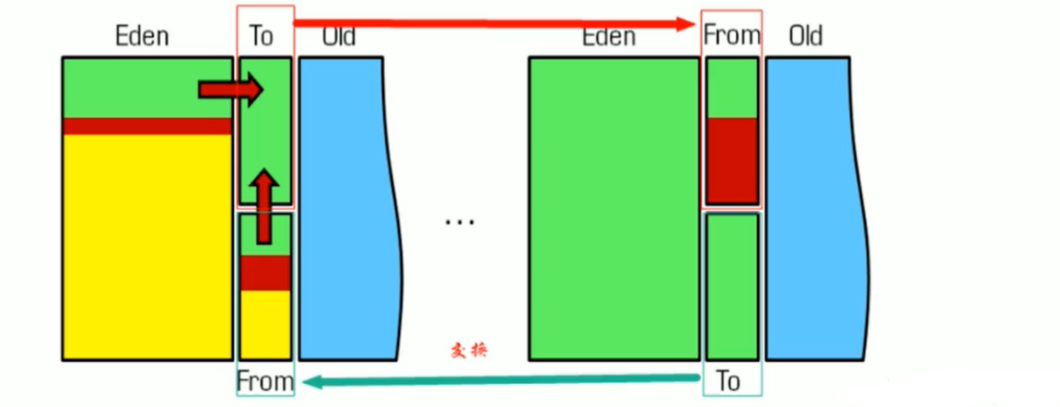
6 清除阶段：复制算法

- 复制（Copying）算法
  - 为了解决标记-清除算法在垃圾收集效率方面的缺陷，M.L.Minsky于1963年发表了著名的论文，“使用双存储区的Lisp语言垃圾收集器（CA LISP Garbage Collector Algorithm Using Serial Secondary Storage）”。M.L.Minsky本人成功地将该算法引入到Lisp语言的一个实现版本中。
  - 核心思想：将活着的内存空间分为两块，每次使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后清除正在使用的内存块中的所有对象，交换两个内存的角色，最后完成垃圾回收。



- 优点：
  - 没有标记和清除过程（我感觉这句话不对），实现简单，运行高效
  - 复制过去后保证了空间的连续性，不会出现“碎片”问题
- 缺点：
  - 此算法的缺点也是非常明显的，就是需要两倍的内存空间。
  - 对于G1这种分拆称为大量region的GC，赋值而不是移动，意味着GC需要维护region之间对象引用关系，不管内存占用还是时间，开销也不小。
- **特别的：**
  - 如果系统中的垃圾对象很多，复制算法不会很理想。因为复制算法需要复制的存活对象数量数量非常低效率才高。
- 应用场景

在新生代，对常规应用的垃圾回收，一次通常可以回收70%-99%的内存空间。回收性价比很高。所以现在的商业虚拟机都是用这种收集算法回收新生代。

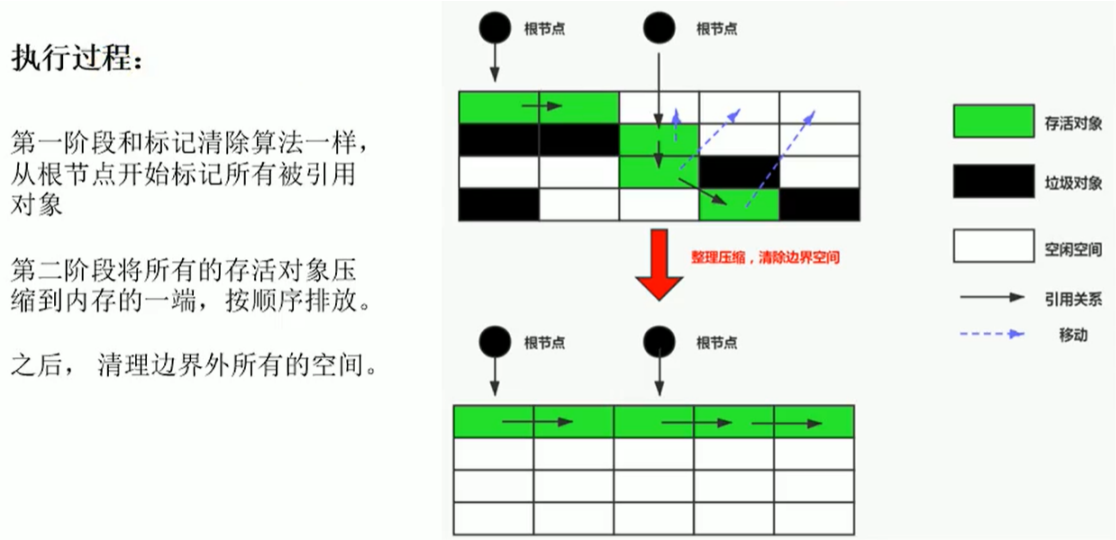


7 清除阶段：标记-压缩算法

- 标记-压缩（或标记-整理、Mark-Compact）算法
  - 背景：复制算法的高效性是建立在存活对象少、垃圾对象多的前提下的。这种情况在新生代进程发生，但是在老年代，更常见的情况是大部分对象都是存活对

象。如果依然使用复制算法，由于存活对象较多，复制的成本也将很高。因此，**基于老年代垃圾回收的特性，需要使用其他的算法**。

- 标记-清除算法的确可以应用在老年代中，但是该算法不仅执行效率低下，而且在执行完内存回收后还会产生碎片，所以JVM的设计者需要在此基础之上进行改进。标记-压缩（Mark-Compact）算法由此诞生。
- 1970年前后，G.L.Steele、C.J.Chene和D.S.Wise等研究者发布标记-压缩算法。在许多现代的垃圾收集器中，人们都是用了标记-压缩算法或其他改进版本。



- 标记-压缩算法的最终效果等同于标记-清除算法执行完成后，再进行一次内存碎片整理，因此，也可以把它称为**标记-清除-压缩（Mark-Sweep-Compact）算法**。
- 标记-压缩算法 和 标记-清除算法本质区别
  - 标记-压缩算法是**移动式的**。
  - 标记-清除算法是一种**非移动式的回收算法**。
  - 是否移动回收后的存活对象是一项优缺点并存的风险决策。
- 可以看到，标记的存活对象将会被整理，按照内存地址一次排列，而未被标记的内存会被清理掉。如此以来，当我们需要给新对象分配内存时，JVM只需要持有一个内存的起始地址即可，这比维护一个空闲列表显然少了许多开销。
- 优点：
  - 消除了标记-清除算法中，内存区域分散的缺点，我们需要给新对象分配内存是，JVM只需要持有一个内存的起始地址即可。
  - 消除了赋值算法当中，内存减半的高额代价。
- 缺点：
  - 从效率上来说，标记-整理算法要低于另外两种算法（标记-清除算法、复制算法）
  - 移动对象的同时，如果对象被其他对象引用，则还需调整引用的地址
  - 移动的过程中，需要全称暂停用户应用程序。即STW。（其他两种算法也有这个问题）

8 小节

	Mark-Sweep	Mark-Compact	Copying
速度	中等	最慢	最快
空间开销	少（但会堆积碎片）	少（不堆积碎片）	通常需要活对象的2倍大小（不堆积碎片）
移动对象	否	是	是

效率上来说，复制算法是当之无愧的老大，但是却浪费了太多内存。

而为了尽量兼顾上面的三个指标，标记-整理算法相对来说更平滑一些，但是效率上不尽如人意，它比复制算法多了一个标记的阶段，比标记-清除算法多了一个整理内存的阶段。

9 分代收集算法

- **难道就没有一种最优的算法吗？**
  - 回答：没有，没有最好的算法，只有最适合的算法。
- 分代收集算法
  - 前面所有这些算法中，并没有一种算法可以完全替代其他算法，他们都具有自己独特的优势和特点。分代收集算法应运而生。
  - 分代收集算法，是基于这样一个事实：不同的对象的生命周期是不一样的。因此，**不同生命周期的对象可以采取不同的收集方式，一遍提高回收效率**。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点使用不同的回收算法，以提高垃圾回收的效率。
  - 在Java程序运行的过程中，会产生大量的对象，其中有些对象是与业务信息相关，比如**Http请求中的Session对象、线程、Socket连接**，这类对象跟业务直接挂钩，因此生命周期比较长。但是还有一些对象，主要是程序运行过程中生成的临时变量，这些对象生命周期比较短，比如：**String对象**，由于其不变性的特性，系统会产生大量这些对象，有些对象甚至只用一次即可回收。
  - **目前几乎所有的GC都是采用分代收集（Generational Collecting）算法执行垃圾回收的**。
  - 在Hotspot中，基于分代的概念，GC所使用的内存回收算法必须结合年轻代和老年代各自的特点。
    - 年轻代（Young Gen）

      年轻代特点：区域相对于老年代较小，对象声明周期短、存活率低，回收频繁。这种情况**复制算法**是回收整理速度最快的。赋值算法的效率之和当前存活对象大小有关，因此很适用于年轻代的回收。而赋值算法内存利用率不高的问题，通过hotspot中的两个survivor的设计得到缓解。
    - 老年代（Tenured Gen）

      老年代特点：区域较大，对象生命周期长、存活率高，回收不及年轻代频繁。这种情况下存在大量存活率高的对象，赋值算法变得明显不合适。一般是由**标记-清除算法 与 标记-压缩算法**的混合实现。
      - Mark阶段的开销与存货对象的数量成正比。
      - Sweep阶段的开销与所管理区域的大小成正相关。

        - Compact阶段的开销与存货对象的数据成正比。

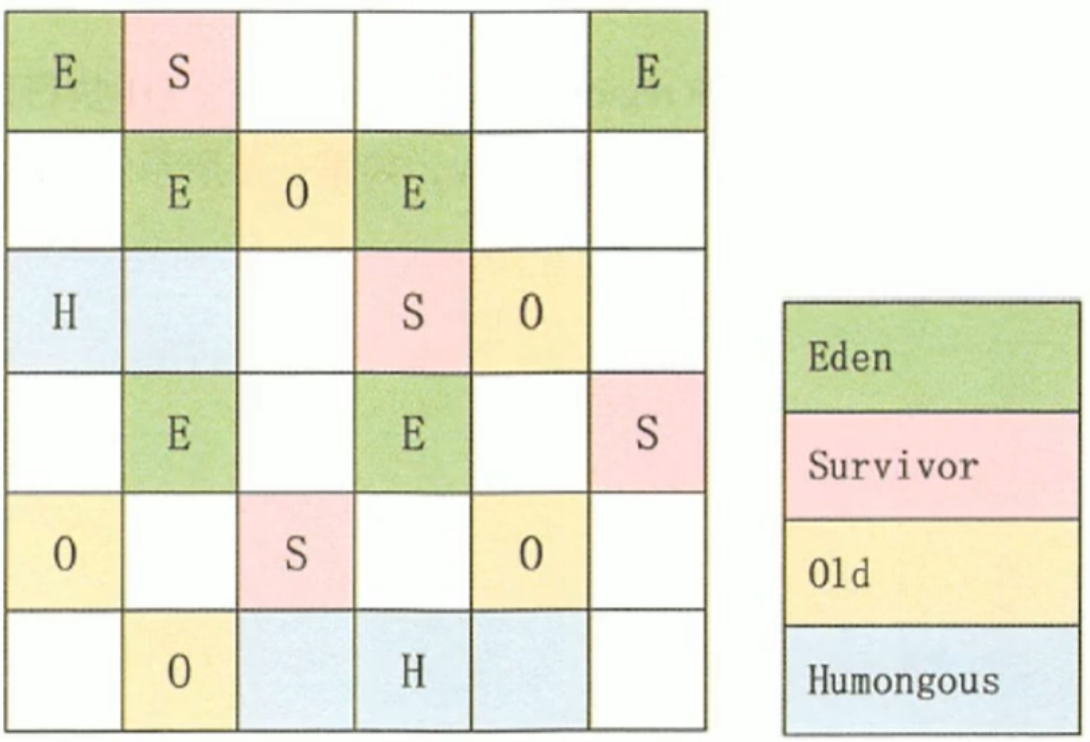
      以Hotspot中的CMS回收器为例，CMS是基于Mark-Sweep实现的，对于对象的回收效率很高。而对于碎片问题，CMS采用基于Mark-Compact算法的Serial Old回收器作为补偿措施：当内存回收不佳（碎片导致Concurrent Mode Failure时），将采用Serial Old执行Full GC以达到对老年代内存的整理。

- 分代的思想被现有的虚拟机广泛使用。几乎所有的垃圾回收器都区分新生代和老年代。

10 增量收集算法、分区算法

- 这两种算法都是为了解决STW的问题
- 增量收集算法（从时间角度提高低延迟）
    - 上述现有的算法，在垃圾回收过程中，应用软件将处于一种Stop the World的状态。在**Stop the World**状态下，应用程序所有的线程都会挂起，暂停一切正常的工作，等待垃圾回收的完成。如果垃圾回收时间过长，应用程序会被挂起很久，**将严重影响用户体验或者系统的稳定性**。为了解决这个问题，即对实时垃圾收集算法的研究导致了增量收集（Incremental Collecting）算法的诞生。
    - 基本思想：如果一次性将所有的垃圾进行处理，需要造成系统长时间的停顿，那么就可以让垃圾收集线程和应用程序交替执行。每次，**垃圾收集线程只收集一小片区域的内存空间，接着切换到应用程序线程。依次反复，直到垃圾收集完成**。
    - 总的来说，增量收集算法的基础仍然是传统的标记-清除算法和复制算法。增量收集算法通过**对线程间冲突的妥善处理，允许垃圾线程以分阶段的方式完成标记、清理或复制工作**。
    - 缺点：
        - 使用这种方式，由于在垃圾回收过程中，间断性地还执行了应用程序代码，所以能减少系统的停顿时间。但是，因为线程切换和上下文转换的消耗，会使得垃圾回收的总成本上升，**造成系统吞吐量的下降**。

- 分区算法（从空间角度提高低延迟）
    - 一般来说，在相同条件下，堆空间越大，一次GC所需要的时间越长，有关GC产生的停顿也越长。为了更好地控制GC产生的停顿时间，将一块大的内存区域分割成多个小块，格局目标的停顿时间，每次合理地回收若干小区间，而不是整个堆空间，从而减少一次GC锁产生的停顿。
    - 分代算法将按照对象的生命周期长短划分为两个部分，分区算法将整个堆空间划分成连续的不同小区间region。
    - 每一个小区间都独立使用，独立回收。这种算法的好处是可以控制一次回收多少峰小区间。



- 写在最后：
  - 注意，这些只是基本的算法思路，实际GC实现过程要复杂的多，目前还在发展中的前言GC都是符合算法，并且并行和并发兼备。