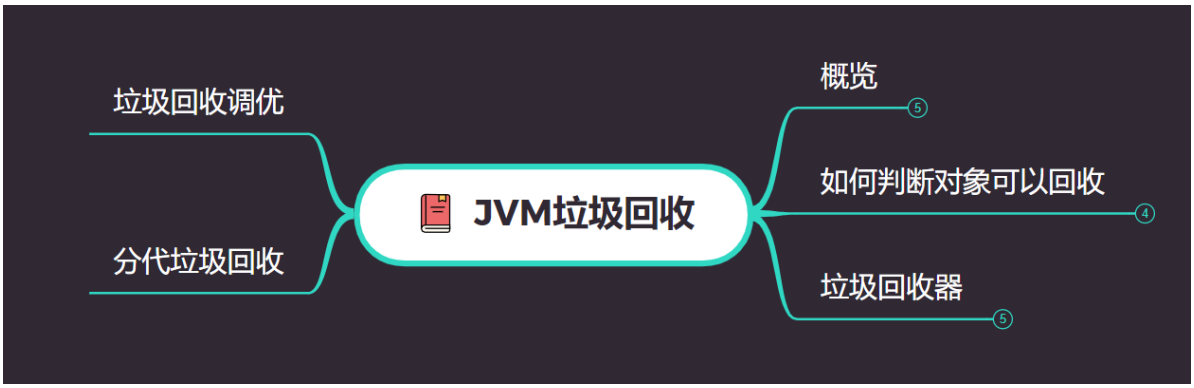
垃圾回收



上一篇我们知道了JVM的运行时区域，其中**程序计数器**、**虚拟机栈**、**本地方法栈** 三个区域随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。每一个栈帧中分配多少内存基本上是在类结构确定下来时就已知的，因此这几个区域的**内存分配和回收**都具备**确定性**。在这几个区域内不需要过多考虑回收的问题，**因为方法结束或线程结束时，内存自然就跟随着回收了。**

而 `Java` **堆** 和 **方法区** 则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样。我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分**内存的分配和回收**都是**动态的**，**垃圾收集器** 所关注的是这部分内存。

在这里我们主要学习

- 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

1. 如何判断对象可以回收

如何判断 `Java` 中一个对象应该 **“存活”** 还是 **“死去”**，这是 垃圾回收器要做的第一件事

1.1 引用计数法

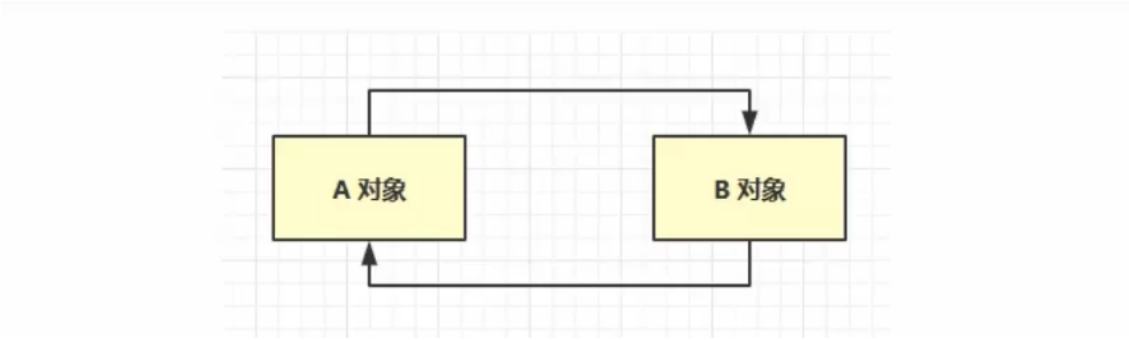
`Java` **堆** 中每个具体对象（**不是引用**）都有一个**引用计数器。**当一个对象被创建并初始化赋值后，该变量计数设置为 `1` 。每当有一个地方引用它时，计数器值就**加1。**当引用**失效**时，即一个对象的某个引用超过了生命周期（出作用域后）或者被设置为一个新值时，计数器值就**减1。**任何引用计数为 `0` 的对象可以被当作**垃圾收集。**当一个对象被垃圾收集时，它引用的任何对象计数减1。

- **优点**:

  引用计数收集器执行简单，判定效率高，交织在程序运行中。对程序不被长时间打断的实时环境比较有利。

- **缺点**:

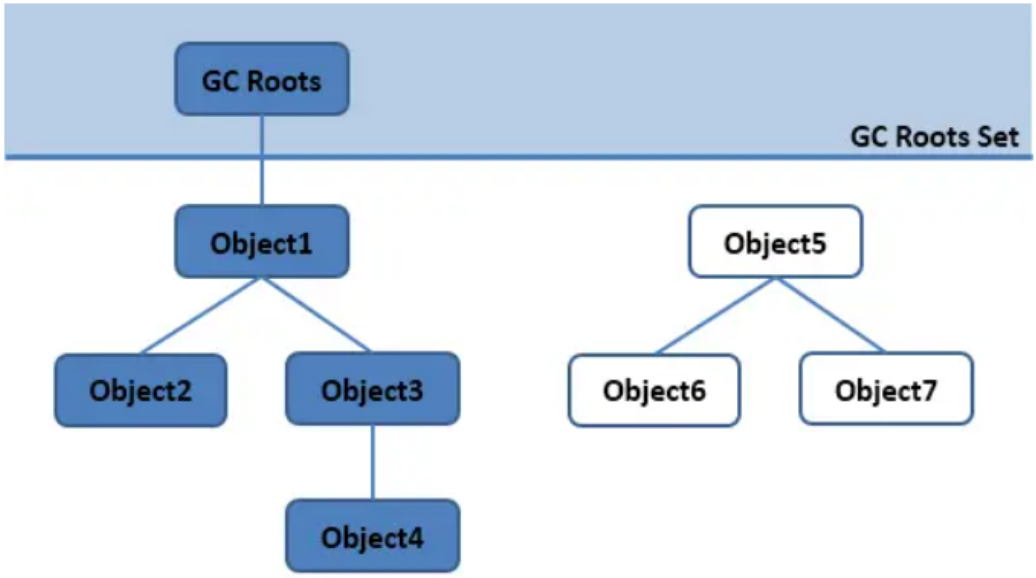
  难以检测出对象之间的循环引用。同时，引用计数器增加了程序执行的开销。**所以Java语言并没有选择这种算法进行垃圾回收。**



1.2 可达性分析法

可达性分析算法也叫**根搜索算法**，通过一系列的称为 `GC Roots` 的对象作为起点，然后向下搜索。搜索所走过的**路径**称为引用链 （`Reference Chain`），当一个**对象**到 `GC Roots` 没有任何**引用链**相连时，即该对象**不可达**，也就说明此对象是 **不可用的。**

如下图所示: `Object5`、`Object6`、`Object7` 虽然互有关联, 但它们到 `GC Roots` 是不可达的, 因此也会被判定为可回收的对象



那么那些对象是 `GC root` 对象呢？我们可以使用eclipse家的一个工具 `Menory Analyzer(MAT)` 来分析，地址：https://www.eclipse.org/mat/

我们先要有一段测试代码：

```
1  public class TestGCRoot {
2      public static void main(String[] args) throws IOException {
3          List<Object> list = new ArrayList<>();
4          list.add("a");
5          list.add("b");
6          System.out.println(1);
7          System.in.read();
8
9          list = null;
10         System.out.println(2);
11         System.in.read();
12         System.out.println("end...");
13     }
14 }
```

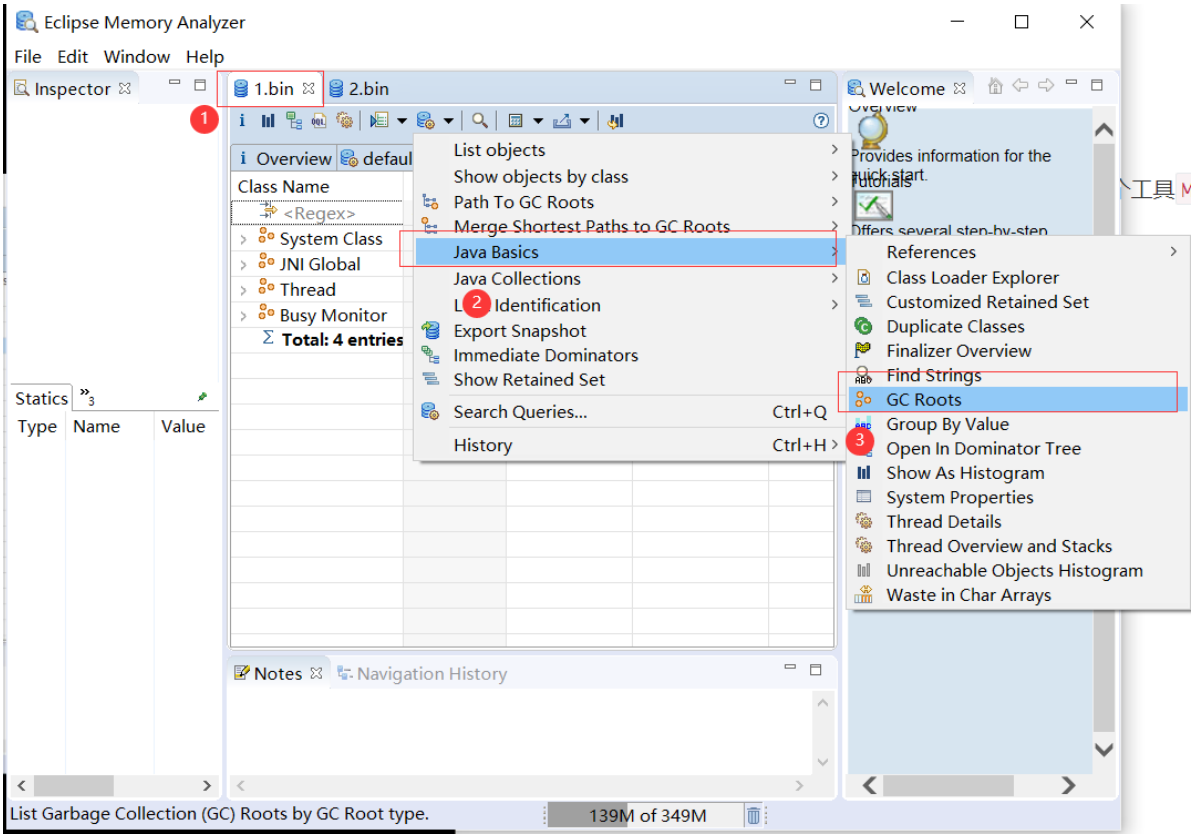
mat工具的使用需要使用jmap工具抓取的内存快照

首先我们使用命令抓取堆空间的内存快照

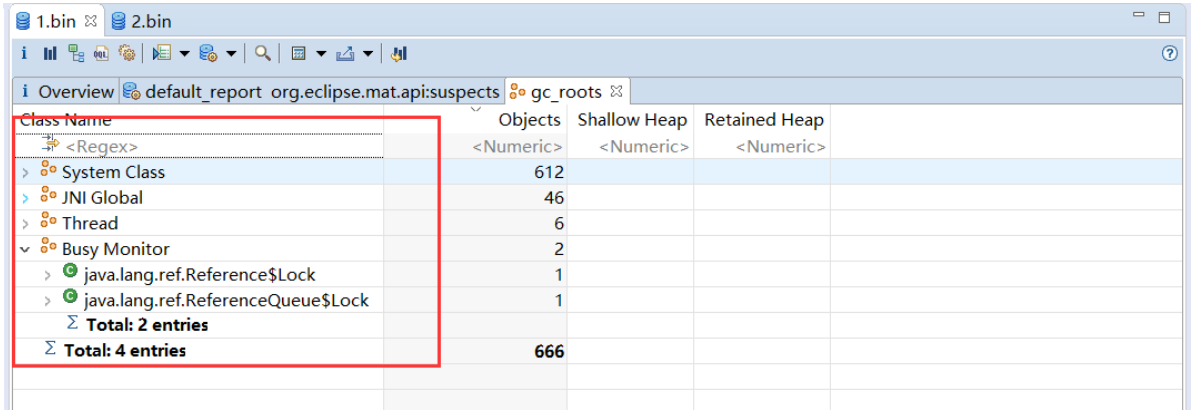
```
// b表示二进制 live表示先进行一次gc并查看存活的对象  输出路径  线程id
jmap -dump:format=b,live,file=D:\codeTools\mat\testdata\1.bin 21088
```

接着我们用mat打开二进制文件，file -> open heap dump

我这里打开了两个，一个gc前的一个gc后的，现在我们看一下gc root对象

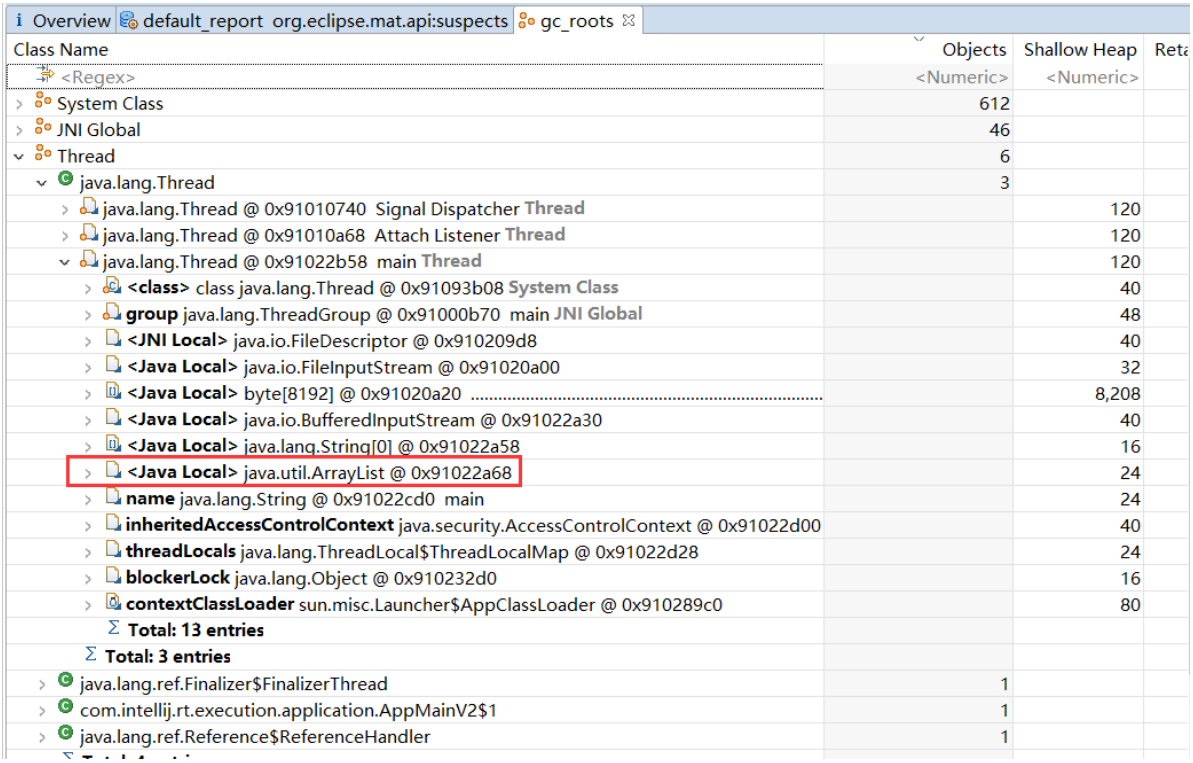


我们可以看到gc root对象被分为了四类：



- System Class（系统核心类）
- JNI Global（**本地方法栈** 中（`Native` 方法）引用的变量）
- Thread（活动线程对象）
- Busy Monitor（被加锁的对象，synchronized锁住的对象）

我们来分析**Thread（活动线程中的对象）**的主线程，我们进行查看可以看到：



这里我们需要注意：

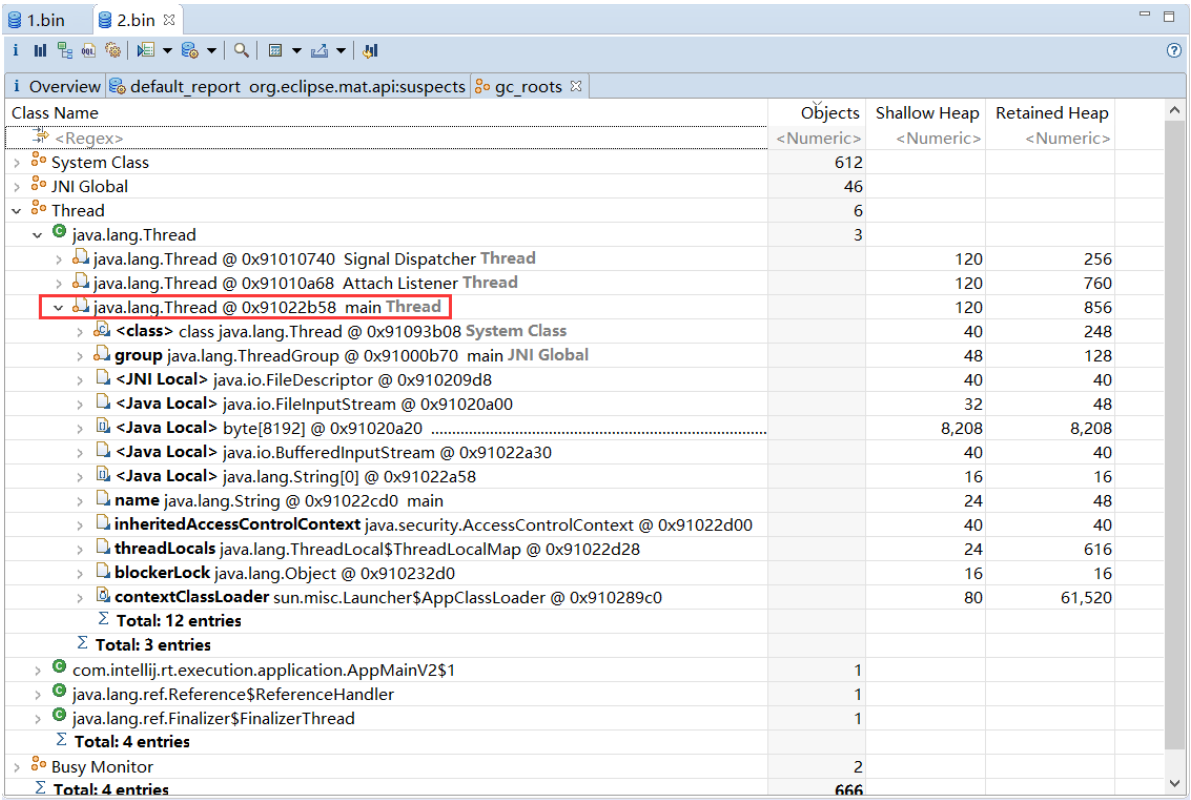
```
List<Object> list = new ArrayList<>();
```

等号前面的是局部变量，是存放在栈帧中的，后面new出来的对象是存在与堆空间里面的，所以在堆里的对象才是gc root，而不是引用变量

即在程序活动过程中，局部变量所引用的变量是GC root对象

包括方法参数也是一样的，在红色记号的上一行，main函数方法参数中引用的对象也是gc root

接下来我们查看进行了一次垃圾回收之后的信息



可以看到因为我们让本地变量的引用指向了null，并且经过了一次gc，所以`ArrayList`这个对象已经被清理了

```
list = null;
```

> 这里千万要区分开：GC ROOT是指的引用的对象，不是引用变量

总结：gc root有：

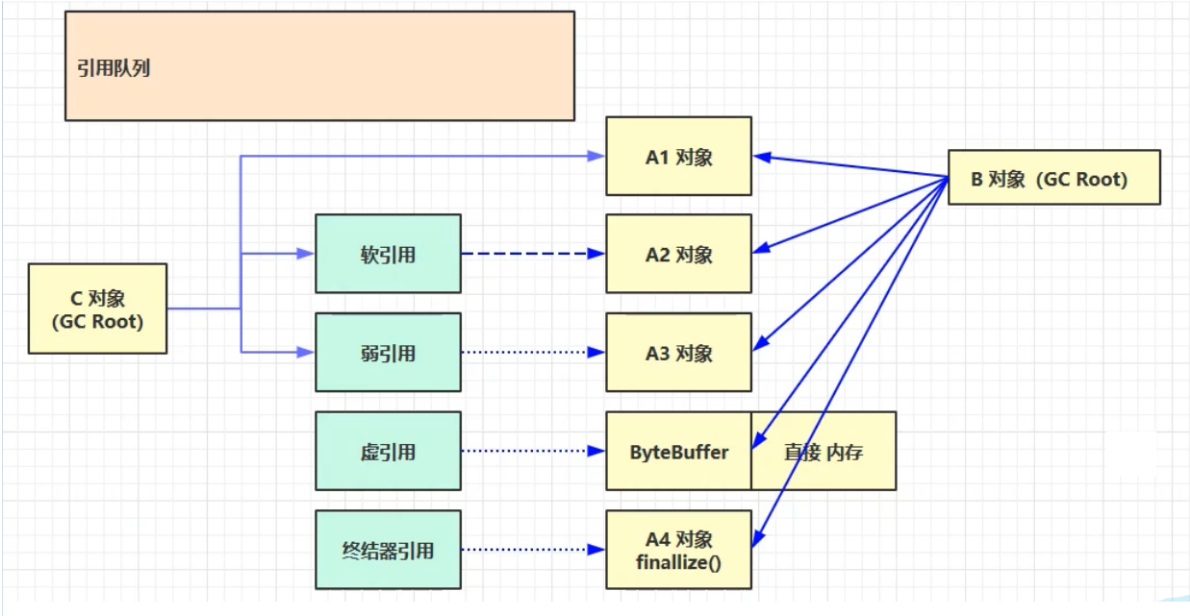
- 虚拟机栈中引用的对象
  - ➢比如：各个线程被调用的方法中使用到的参数、局部变量等。
- 本地方法栈内JNI（通常说的本地方法）引用的对象
- 方法区中类静态属性引用的对象
  - ➢比如：Java类的引用类型静态变量
- 方法区中常量引用的对象
  - ➢比如：字符串常量池（string Table） 里的引用
- 所有被同步锁synchronized持有的对象
- Java虚拟机内部的引用。
  - ➢基本数据类型对应的Class对象，一些常驻的异常对象（如：NullPointerException、OutOfMemoryError），系统类加载器。

1.3 四种引用类型

1. 强引用（StrongReference）
2. 软引用（SoftReference）
3. 弱引用（WeakReference）
4. 虚引用（PhantomReference）
5. 终结器引用（FinalReference）

强、软、弱、虚、终结器引用

引用类型	GC时JVM内存充足	GC时JVM内存不足
强引用	不被回收	不被回收
弱引用	被回收	被回收
软引用	不被回收	被回收



上面的实现代表强引用，虚线箭头表示软弱虚、终结器引用

1.3.1 强引用

我们平时new一个对象就是强引用

```
List<Object> list = new ArrayList<>();
```

1.3.2 软、弱引用

没有被GC root对象直接引用，而是被GC root对象引用的对象引用，这样的应用就是软引用或弱引用

- 软引用和弱引用的特性基本一致， 主要的区别在于软引用在内存不足时才会被回收。如果一个对象只具有软引用，Java GC在内存充足的时候不会回收它，内存不足时才会被回收。
- **如果一个对象只具有弱引用**，无论内存充足与否，Java GC后对象如果只有弱引用将会被自动回收。

这里要注意的是，当软、弱引用引用的对象被清理后，引用就会进入引用队列，等待被清理，当然软、弱引用可以配合引用队列使用，也可以不配合引用队列使用

我们来举一个软引用的栗子：

```
/**
 * 测试软引用
 * -Xmx20m  -XX:+PrintGCDetails -verbose:gc
 *
 * @since: 2022/6/1 17:18
 * @author: 梁峰源
 */
public class TestSoftReference {
    private static final int _4MB = 4 * 1024 * 1024;

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

    /**
     * list -> SoftReference -> byte[]
     */
    public static void soft(){
```

```
        List<SoftReference<byte[]>> list = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            SoftReference<byte[]> ref = new SoftReference<>(new byte[_4MB]);
            System.out.println(ref.get());
            list.add(ref);
            System.out.println(list.size());
        }
        System.out.println("循环结束：" +list.size());
        for(SoftReference<byte[]> ref : list){
            System.out.println(ref.get());
        }
    }
}
```

打印结果：

```
[B@1b6d3586
1
[B@4554617c
2
[B@74a14482
3
[GC (Allocation Failure) [PSYoungGen: 2081K->488K(6144K)] 14369K->13114K(19968K), 0.0025689 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[B@1540e19d
4
[GC (Allocation Failure) --[PSYoungGen: 4809K->4809K(6144K)] 17435K->17435K(19968K), 0.0019384 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Ergonomics) [PSYoungGen: 4809K->4531K(6144K)] [ParOldGen: 12626K->12586K(13824K)] 17435K->17118K(19968K), [Metaspace: 3329K->3329K(1056768K)], 0.0074210 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
[GC (Allocation Failure) --[PSYoungGen: 4531K->4531K(6144K)] 17118K->17126K(19968K), 0.0010017 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Allocation Failure) [PSYoungGen: 4531K->0K(6144K)] [ParOldGen: 12594K->716K(9216K)] 17126K->716K(15360K), [Metaspace: 3329K->3329K(1056768K)], 0.0056801 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[B@677327b6
5
循环结束：5
null
null
null
null
[B@677327b6
Heap
 PSYoungGen      total 6144K, used 4433K [0x00000000ff980000, 0x0000000100000000, 0x0000000100000000)
  eden space 5632K, 78% used [0x00000000ff980000,0x00000000ffdd4650,0x00000000fff00000)
  from space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)
  to   space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)
 ParOldGen       total 9216K, used 716K [0x00000000fec00000, 0x00000000ff500000, 0x00000000ff980000)
```

```
  object space 9216K, 7% used [0x00000000fec00000,0x00000000fecb32a8,0x00000000ff500000)
 Metaspace       used 3349K, capacity 4500K, committed 4864K, reserved 1056768K
  class space    used 358K, capacity 388K, committed 512K, reserved 1048576K
```

可以看到当空间不足时，共触发了三次GC，前两次GC并没有回收多少空间，所以触发了第三次，将软连接指向的对象回收了，但是从最后的结果我们可以看出，其实软引用并没有被回收

现在我们来手动回收一下软引用，如何清理呢？我们需要配合`引用队列`

```java
/**
 * list -> SoftReference -> byte[]
 */
public static void soft(){
    List<SoftReference<byte[]>> list = new ArrayList<>();
    // 引用队列
    ReferenceQueue<byte[]> queue = new ReferenceQueue<>();
    for (int i = 0; i < 5; i++) {
        // 关联软引用队列和软引用，当软引用关联的byte数组被回收时，软引用自己会被加入到queue中
        SoftReference<byte[]> ref = new SoftReference<>(new byte[_4MB],queue);
        System.out.println(ref.get());
        list.add(ref);
        System.out.println(list.size());
    }
    System.out.println("循环结束："+list.size());
    // 手动清理软引用
    Reference<? extends byte[]> poll;
    while((poll = queue.poll()) != null){
        //清理无效引用
        list.remove(poll);
    }
    for(SoftReference<byte[]> ref : list){
        System.out.println(ref.get());
    }
}
```

从结果可以看到无效的引用关系以及被清理了

> 弱引用栗子和软引用一样，只是将`SoftReference`变为了`WeakReference`

1.3.3 虚引用、终结器引用

当虚、终结器引用被创建的时候，一定会关联一个应用队列

我们在之间的ByteBuffer的源码中看到在申请空间时，创建了一个虚引用对象`cleaner`

```
cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
```

> 虚引用和引用队列同时存在，betybuffer没有强引用了 虚引用就会进入虚引用队列，一个检查虚引用队列的线程（ReferenceHandler）会启用unsafe把直接内存释放掉，采用虚引用的目的就是释放直接引用

1.3.4 终结器引用

我们知道所有的对象都继承自`Object`类，`Object`类有一个终结方法`finallize()`，

当我们的对象重写了`finallize()`终结方法并且没有被强引用的时候就可以被垃圾回收了

我们重写了终结方法，是希望该类在被回收之前能够执行一次此方法

其实当没有强引用引用该对象时，会由我们的VM为其创建一个对应`终结器引用`，当该对象被垃圾回收时，会将终结器引用放入引用队列，并由一个优先级很低的线程`finalise Thread`将终结器引用回收，再由JVM回收该类

> 虚引用是将被强引用的对象附带的那部分内存也回收了去，但是回收方式不同。终结器引用是当终结器引用进入引用队列并调用finallize方法后才将对象回收掉的

1.3.5 总结

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
2. 软引用（SoftReference） 仅有软引用引用该对象时，在垃圾回收后，内存仍不足时会再次出发垃圾回收，回收软引用 对象 可以配合引用队列来释放软引用自身
3. 弱引用（WeakReference） 仅有弱引用引用该对象时，在垃圾回收时，无论内存是否充足，都会回收弱引用对象 可以配合引用队列来释放弱引用自身
4. 虚引用（PhantomReference） 必须配合引用队列使用，主要配合 ByteBuffer 使用，被引用对象回收时，会将虚引用入队， 由 Reference Handler 线程调用虚引用相关方法释放直接内存
5. 终结器引用（FinalReference） 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

2. 垃圾清除算法

[垃圾回收Hotspot jdk1.8 oracle官方文档]

常见的垃圾清除算法有：

- 标记清除
- 标记整理
- 复制

2.1 标记清除

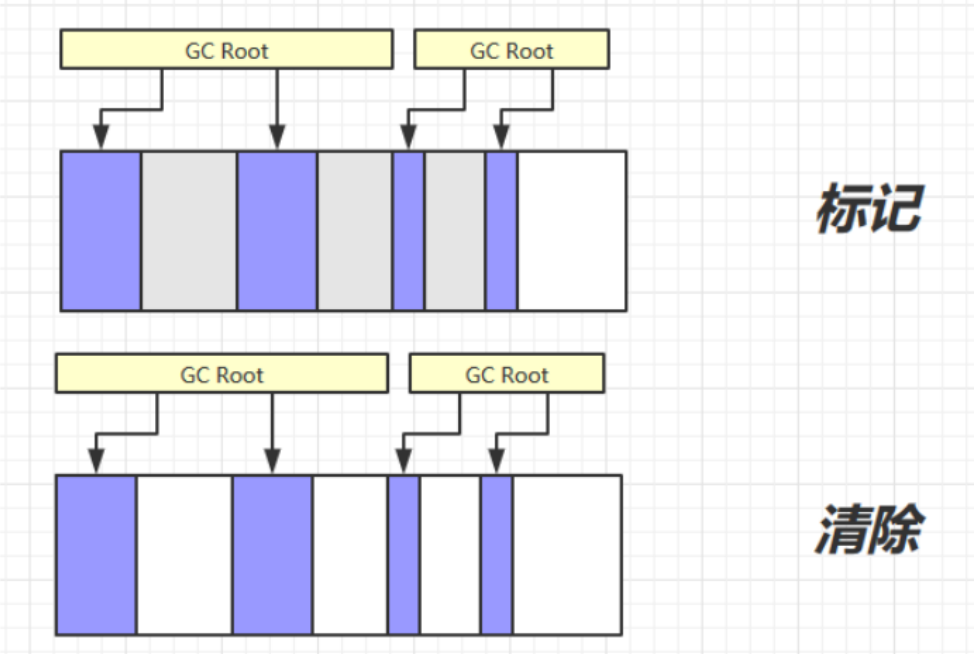
标记-清除算法对**根集合**进行扫描，对**存活**的对象进行标记。标记完成后，再对整个空间内**未被标记**的对象扫描，进行回收。（注意这里的回收并不是置为空，只是标记无用，下一次直接用其他对新覆盖）

- **优点**：

  实现简单，不需要进行对象进行移动。

- **缺点**：

  标记、清除过程效率低，产生大量不连续的内存碎片，提高了垃圾回收的频率。



2.2 标记整理

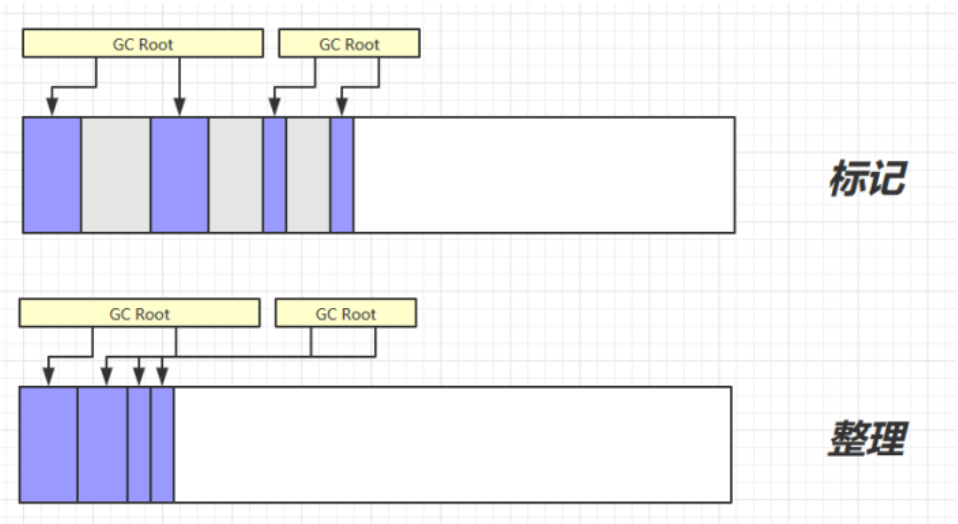
标记-整理算法 采用和 **标记-清除算法** 一样的方式进行对象的标记，但后续不直接对可回收对象进行清理，而是将所有的**存活对象**往一端**空闲空间**移动，然后清理掉端边界以外的内存空间。

- **优点**:

  解决了标记-清理算法存在的内存碎片问题。

- **缺点**:

  仍需要进行局部对象移动，一定程度上降低了效率。



2.3 复制

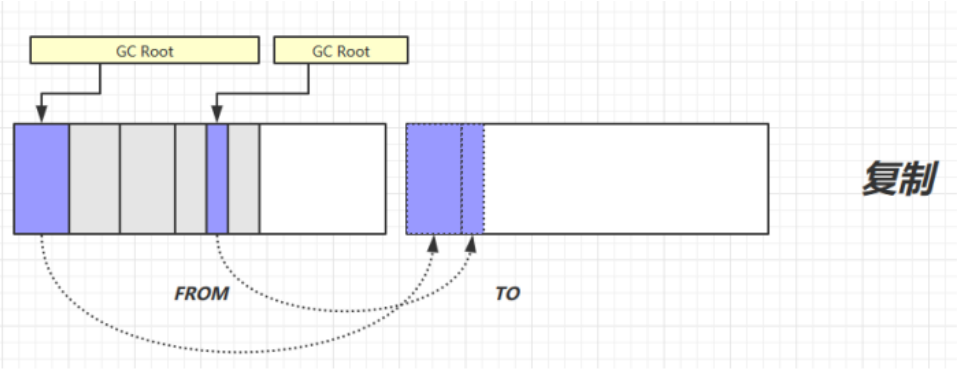
这种收集算法解决了标记清除算法存在的效率问题。它将内存区域划分成相同的两个**内存块**。每次仅使用一半的空间，`JVM`生成的新对象放在一半空间中。当一半空间用完时进行`GC`，把可到达对象复制到另一半空间，然后把使用过的内存空间一次清理掉。

- **优点**:

  按顺序分配内存即可，实现简单、运行高效，不用考虑内存碎片。

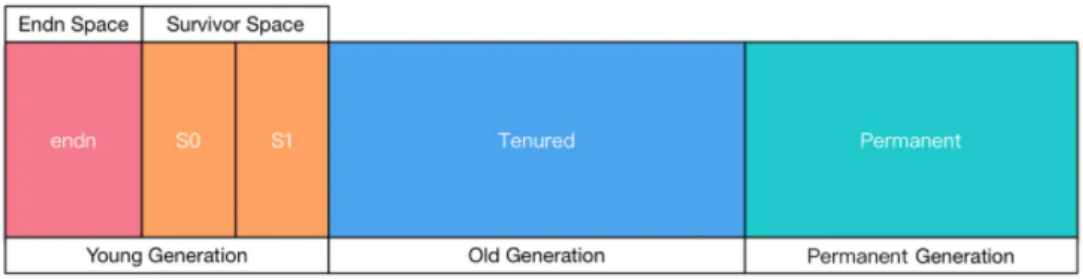
- **缺点**:

  可用的内存大小缩小为原来的一半，对象存活率高时会频繁进行复制。



3. 分代垃圾回收

实际中 `JVM` 不会只采用一种垃圾回收算法，而是采用分代垃圾回收，分代收集算法，顾名思义是根据对象的**存活周期**将内存划分为几块。一般包括**年轻代**、**老年代** 和 **永久代**，如图所示



3.1 新生代、老年代GC

新生代（Young generation）

绝大多数最新被创建的对象会被分配到这里，由于**大部分对象**在创建后会很快变得**不可达**，所以很多对象被创建在**新生代**，然后**消失**。对象从这个区域消失的过程我们称之为 `minor GC` 。

新生代 中存在一个 `Eden` 区和两个 `Survivor` 区。新对象会首先分配在 `Eden` 中（如果新对象过大，会直接分配在老年代中）。在 `GC` 中， `Eden` 中的对象会被移动到 `Survivor` 中，直至对象满足一定的年纪（定义为熬过 `GC` 的次数），会被移动到**老年代**。

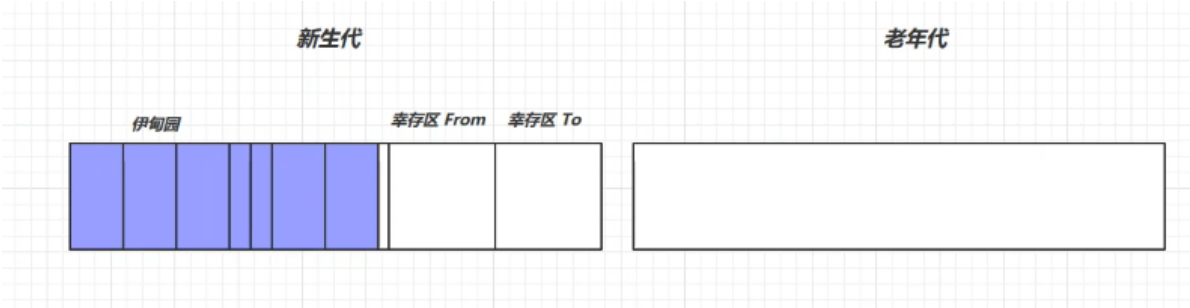
可以设置**新生代**和**老年代**的相对大小。这种方式的优点是新生代大小会随着整个**堆**大小**动态扩展**。参数 `-XX:NewRatio` 设置**老年代**与**新生代**的比例。例如 `-XX:NewRatio=8` 指定 **老年代/新生代** 为 `8/1` . **老年代** 占堆大小的 `7/8` ， **新生代** 占堆大小的 `1/8` （默认即是 `1/8` ）

老年代（Old generation）

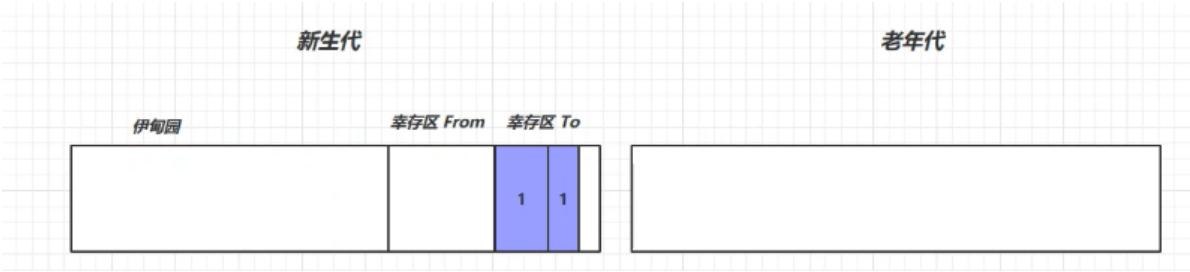
对象没有变得不可达，并且从新生代中**存活**下来，会被**拷贝**到这里。其所占用的空间要比新生代多。也正由于其相对**较大的空间**，发生在**老年代**上的 `GC` 要比**新生代**要**少得多**。对象从**老年代**中消失的过程，可以称之为 `major GC` （或者 `full GC` ）。

> 下面来演示一下新生代产生和消亡的过程

当我们创建一个对象时，默认会使用 `Eden` 的空间

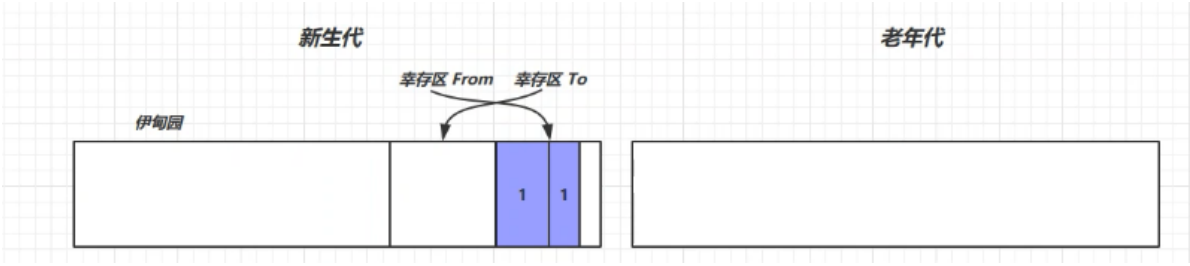


但是当我们的对象不断创建，`Eden`中的空间很快就会不够用，这是就会触发一次`Minor GC`，`Minor GC`触发后会采用可达性分析算法寻找可以清理的对象，然后采取`复制`垃圾回收算法将存活的对象复制到`幸存区To`，并且将幸存的对象（即在Minor GC后存活下来的对象）的**寿命加一**

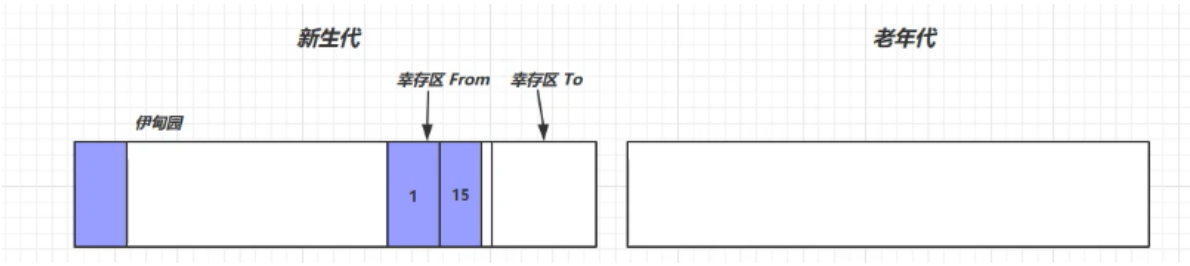


根据`复制`内存算法，伊甸园中的其他内存就会被整段清除，这样可以防止内存碎片的产生，然后交换幸存区From和幸存区To的指向（物理上不会变化，只会逻辑上改变）

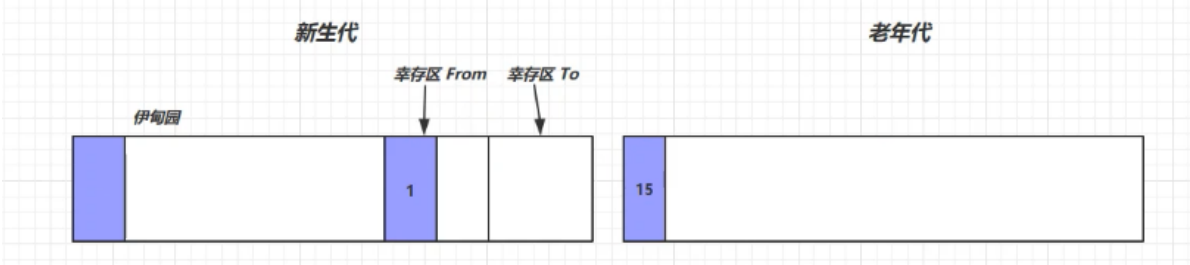
注意: from与to只是两个指针，它们是变动的，to指针指向的Survivor区是空的



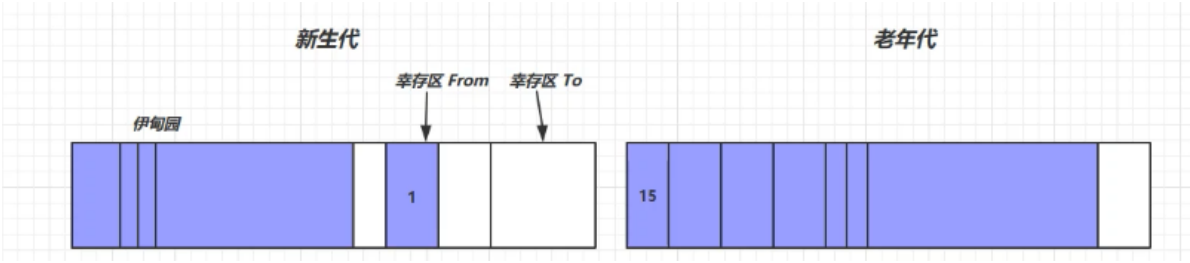
接着伊甸园中空间又充足了，可以继续分配空间了，当伊甸园又满了之后，会触发`第二次Minor GC`，这次是将伊甸园和幸存区From中的对象清理，存活下来的对象又会被保存在幸存区To中，接着又会交换幸存区From和To的指向，并将存活下来的对象寿命加一



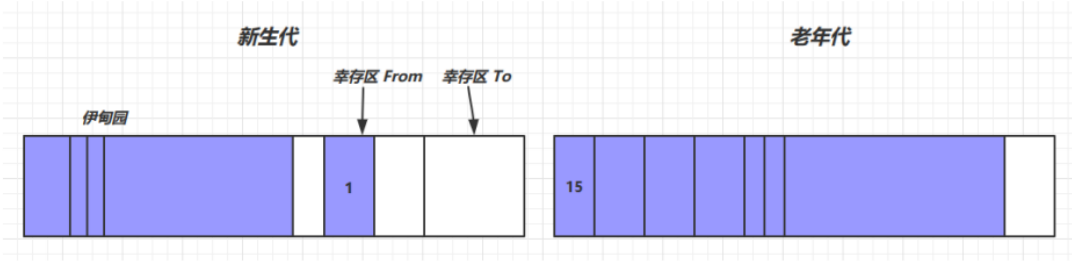
如果对象的寿命达到了`15`，也就是经历了`15次GC`还没被清理，它就会被**晋升到老年代中**，老年代的GC频率会低一些



当我们的老年代空间满后（可能这个时候新生代的空间也会满），会触发一次`Full GC`，`Full GC`会对整个堆空间进行清理，包括新生代和老年代的空间，所以`Full GC`称之为重量级GC，`Full GC`采用的垃圾回收策略有两种，分别是**标记+清除和标记+整理**，比较消耗时间



3.2 新生代、老年代GC总结



我们总结下新生代、老年代GC的特点：

- 对象首先分配在伊甸园区域
- 新生代空间不足时，触发 **Minor GC**，伊甸园和幸存区from存活的对象使用copy算法复制到幸存区to中，存活的对象年龄加一，并交换幸存区from to
- **Minor GC**会引发`stop the world`，即暂停所有的线程，仅当 **Minor GC**的线程执行完后其他的线程才能继续运行。产生stw是因为在 **Minor GC**的过程中会产生对象的复制（对象的地址发生改变），这是如果有线程正在工作可能会导致访问的对象突然消失的情况，所以会产生stw
- 当对象寿命超过阈值时，会晋升至老年代，最大寿命为15（对象头中只有4bit空间）
- 当老年代空间不足，会先尝试触发 **Minor GC**，如果之后空间仍然不足，那么就会触发 **full GC**

3.3 相关VM参数

含义	参数
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

其中`SurvivorRatio`非常有文章，参考：

- JVM 参数解析： SurvivorRatio
- oracle官方调优文档

3.4 垃圾回收过程

测试下面的代码：

```
/**
 * 测试minorGC
 * -Xms20m -Xmx20m -Xmn10m -XX:+UseSerialGC -XX:+PrintGCDetails -verbose:gc
 *
 * @since: 2022/6/1 20:26
 * @author: 梁峰源
 */
public class TestMinorGC {
    private static final int _512KB = 512 << 10;
    private static final int _1MB = 2 << 20;
    private static final int _6MB = 6 << 20;
    private static final int _7MB = 7 << 20;
    private static final int _8MB = 8 << 20;

    public static void main(String[] args) {
        ArrayList<byte[]> list = new ArrayList<>();
        list.add(new byte[_7MB]);
    }
}
```

list未添加元素之前：

```
Heap
 def new generation   total 9216K, used 2087K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  eden space 8192K,  25% used [0x00000000fec00000, 0x00000000fee09f40, 0x00000000ff400000)
  from space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000)
  to   space 1024K,   0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000)
 tenured generation   total 10240K, used 0K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
   the space 10240K,   0% used [0x00000000ff600000, 0x00000000ff600000, 0x00000000ff600200, 0x0000000100000000)
 Metaspace       used 3256K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 346K, capacity 388K, committed 512K, reserved 1048576K
```

之后：

```
[GC (Allocation Failure) [DefNew: 1923K->724K(9216K), 0.0033241 secs] 1923K->724K(19456K), 0.0042289 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 8302K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  eden space 8192K,  92% used [0x00000000fec00000, 0x00000000ff366830, 0x00000000ff400000)
  from space 1024K,  70% used [0x00000000ff500000, 0x00000000ff5b5118, 0x00000000ff600000)
  to   space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000)
 tenured generation   total 10240K, used 0K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
   the space 10240K,   0% used [0x00000000ff600000, 0x00000000ff600000, 0x00000000ff600200, 0x0000000100000000)
 Metaspace       used 3281K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 347K, capacity 388K, committed 512K, reserved 1048576K
```

可以看到`eden`和`from`占了一部分空间

当空间超出新生代的容量后就会往老年代晋升（这里是在原来的基础上又增加了1MB）

```
[GC (Allocation Failure) [DefNew: 1923K->729K(9216K), 0.0019081 secs] 1923K->729K(19456K), 0.0019757 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [DefNew: 8900K->518K(9216K), 0.0058807 secs] 8900K->8411K(19456K), 0.0059344 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
Heap
 def new generation   total 9216K, used 1196K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  eden space 8192K,   8% used [0x00000000fec00000, 0x00000000feca9700, 0x00000000ff400000)
  from space 1024K,  50% used [0x00000000ff400000, 0x00000000ff481ad0, 0x00000000ff500000)
  to   space 1024K,   0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000)
 tenured generation   total 10240K, used 7892K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
   the space 10240K,  77% used [0x00000000ff600000, 0x00000000ffdb5258, 0x00000000ffdb5400, 0x0000000100000000)
 Metaspace       used 3319K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 354K, capacity 388K, committed 512K, reserved 1048576K
```

> 这里有一个点，就是大空间晋升策略，当对象的容量超出`eden`的大小并且from也放不下的话，就会去查看老年代是否能够放下，如果能放下就可以直接晋升到老年代
>
> 注意：这种情况下不会触发垃圾回收！

修改测试代码：

```
list.add(new byte[_8MB]);
```

结果：

```
Heap
 def new generation   total 9216K, used 2419K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  eden space 8192K,  29% used [0x00000000fec00000, 0x00000000fee5cff8, 0x00000000ff400000)
  from space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000)
  to   space 1024K,   0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000)
 tenured generation   total 10240K, used 8192K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
   the space 10240K,  80% used [0x00000000ff600000, 0x00000000ffe00010, 0x00000000ffe00200, 0x0000000100000000)
 Metaspace       used 3331K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 357K, capacity 388K, committed 512K, reserved 1048576K
```

可以看到并没有`GC`的信息出现，`tenured generation`中的空间占了80%

当然如果申请的空间过大，超过了堆空间的大小，就会导致OOM

但如果是主线程的子线程OOM了会导致主线程OOM吗？答案是不会

我们改写测试用例：

```
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();
    Thread.sleep(1000);
}
```

结果：

显然子线程的OOM并没有影响到主线程

```
[GC (Allocation Failure) [DefNew: 4338K->969K(9216K), 0.0031336 secs] [Tenured: 8192K->9159K(10240K), 0.0037985 secs] 12531K->9159K(19456K), [Metaspace: 4246K->4246K(1056768K)], 0.0070174 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
[Full GC (Allocation Failure) [Tenured: 9159K->9103K(10240K), 0.0035054 secs] 9159K->9103K(19456K), [Metaspace: 4246K->4246K(1056768K)], 0.0035468 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
Exception in thread "Thread-0" java.lang.OutOfMemoryError: Java heap space
    at com.fx.gc.MinorGC.TestMinorGC.lambda$main$0(TestMinorGC.java:24)
    at com.fx.gc.MinorGC.TestMinorGC$$Lambda$1/1324119927.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:748)
Heap
 def new generation   total 9216K, used 1293K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  eden space 8192K,  15% used [0x00000000fec00000, 0x00000000fed43588, 0x00000000ff400000)
  from space 1024K,   0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000)
```

```
  to   space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000)
 tenured generation   total 10240K, used 9103K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
   the space 10240K,  88% used [0x00000000ff600000, 0x00000000ffee3f58, 0x00000000ffee4000, 0x0000000100000000)
 Metaspace       used 4768K, capacity 4880K, committed 4992K, reserved 1056768K
  class space    used 524K, capacity 560K, committed 640K, reserved 1048576K
```

4. 垃圾回收器

垃圾回收器的分类可以分为：

1. 串行
   - 单线程。串行的垃圾回收器是单线程的，当GC时需要STW
   - 堆内存较小，适合个人电脑。因为是串行的所以线程多了也没有用
2. 吞吐量优先
   - 适合多线程（适合工作在服务器上）
   - 堆内存较大，需要多核CPU支持
   - 要在单位时间内，STW的时间最短（例如一分钟内发生 0.2 +0.2 = 0.4）
3. 响应时间优先
   - 适合多线程（适合工作在服务器上）
   - 堆内存较大，需要多核CPU支持
   - 尽可能让单次的STW时间最短（例如一分钟内发生 0.1 + 0.1 +0.1 +0.1 +0.1 = 0.5）

4.1 串行垃圾回收器

开启串行垃圾回收的虚拟机参数为：

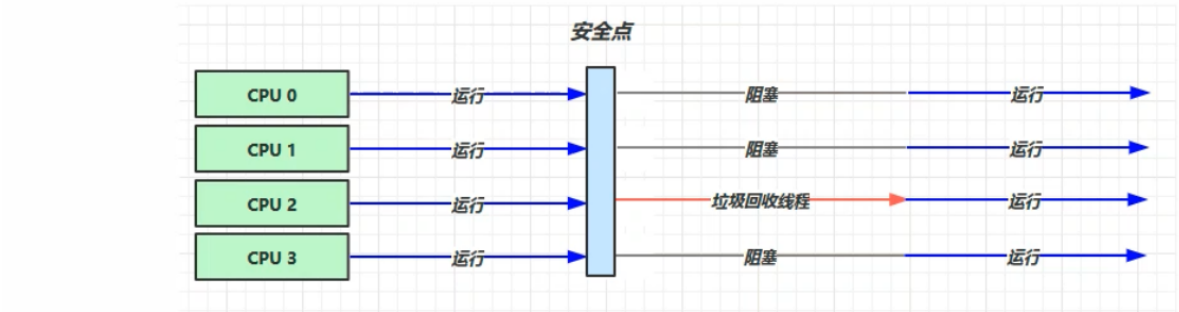
```
-XX:+UseSerialGC=Serial + SerialOld
```

串行垃圾回收器对应的有：

- Serial：作用在新生代，采用复制的垃圾回收算法
- SerialOld：作用在老年代，采用的是标记+整理的垃圾回收算法

接下来演示一下串行垃圾器回收垃圾的过程：

当内存不够时，需要所有线程在一个安全点暂定下来（STW），以为垃圾回收的过程可能会让线程内对象指向的地址发生改变，为了安全的完成GC工作



4.2 吞吐量优先垃圾回收器

开启的JVM参数，1.8下默认就是这个垃圾回收器，所以也可以不用开启

```
-XX:+UseParallelGC ~ -XX:+UseParallelOldGC //这两个开关只需要开启一个，另一个会连带开启
-XX:+UseAdaptiveSizePolicy
-XX:GCTimeRatio=ratio
-XX:MaxGCPauseMillis=ms
-XX:ParallelGCThreads=n
```

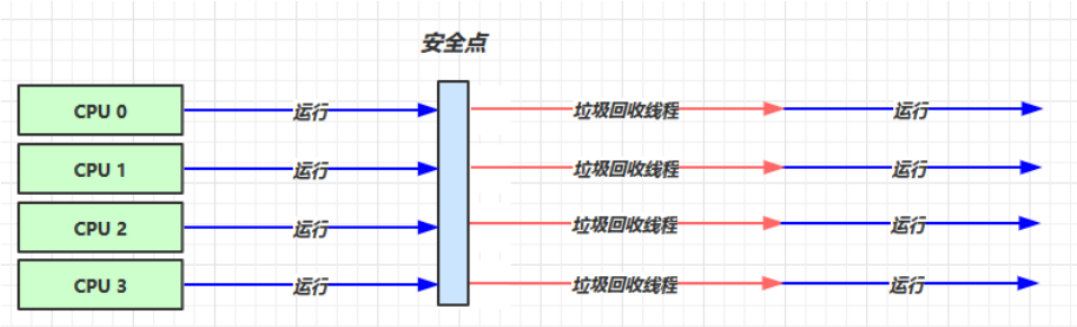
吞吐量优先垃圾回收器对应的有：

- UseParallelGC：新生带垃圾回收器，采用复制算法
- UseParallelOldGC：老年代，标记+整理算法

工作流程：

当内存不够时，需要所有线程在一个安全点暂定下来（STW），然后开启多个线程进行垃圾回收，默认的多线程线程数和CPU的核数保持一致。

它的特点是当GC的时候，CPU的利用率会飙升，接近100%



线程数我们也可以根据一个虚拟机参数进行设置：

```
-XX:ParallelGCThreads=n
```

这个参数还可以跟下面几个参数（一开关、两目标）配合使用：

```
-XX:+UseAdaptiveSizePolicy
-XX:GCTimeRatio=ratio
-XX:MaxGCPauseMillis=ms // 最大暂定毫秒数，默认200ms，这个参数和上面的参数其实是相互矛盾的
```

- -XX:+UseAdaptiveSizePolicy（开关）

  自适应的大小调整策略，调整新生代的大小，动态调整伊甸园和幸存区的内存比例

- -XX:GCTimeRatio=ratio（目标1）

  GC时间占比，垃圾回收的时间和总时间的占比，为 1/(1+ratio)，如果没有达到预期会调整堆空间的大小，默认是调大，因为堆调大后gc的次数会减少

  ratio默认是99，即一百分钟类有一分钟在垃圾回收，有点难达到，我们一般设置为19

- -XX:MaxGCPauseMillis=ms（目标2）

  最大暂停毫秒数，和目标1对立，需要设置一个折中的数字

4.3 响应时间优先

对应虚拟机参数有：

```
-XX:+UseConcMarkSweepGC ~ -XX:+UseParNewGC ~ SerialOld
-XX:ParallelGCThreads=n ~ -XX:ConcGCThreads=threads
-XX:CMSInitiatingOccupancyFraction=percent
-XX:+CMSScavengeBeforeRemark
```

- -XX:+UseConcMarkSweepGC （CMS）

  Con：concurrent 并发，Mark 标记，Sweep清除（**并发标记清除**）

  CMS是并发执行的垃圾回收器，和之前的`Parallel`不同，`Parallel`是并行的，执行GC的时候其他进程只能堵塞等待。

  `CMS`它在工作的同时，用户线程也能执行，`CMS`和用户线程是并发执行的，都要去抢占CPU，当然在GC的某些阶段还是需要STW，但是另外一些阶段是并发执行的，从这里就可以看出`CMS`的优势和特点

  它工作在`老年代`，与之配合的是`UseParNewGC`，在新生代工作。但有的时候`CMS`会出现并发失败的情况，这是`CMS`会退化为`SerialOld`

- -XX:ParallelGCThreads=n

  并行线程数，默认为CPU的核数

- -XX:ConcGCThreads=threads

  并发线程数，一般这个参数我们会设置为并行线程数的四分之一，例如下图为一个线程垃圾回收，其他三个线程留给用户线程执行。注意这样其实会影响到吞吐量

- -XX:CMSInitiatingOccupancyFraction=percent

  由于在`CMS`工作的时候，其他线程还在运行，就可能会产生新的垃圾，这些垃圾我们称之为`浮动垃圾`，`CMS`并不能清理这些新产生的垃圾，只能下次再进行清理，但是这又带来一个问题，因为GC就是因为空间不足了才导致的，如果现在又产生了新的垃圾，那么这些垃圾往哪里放呢？

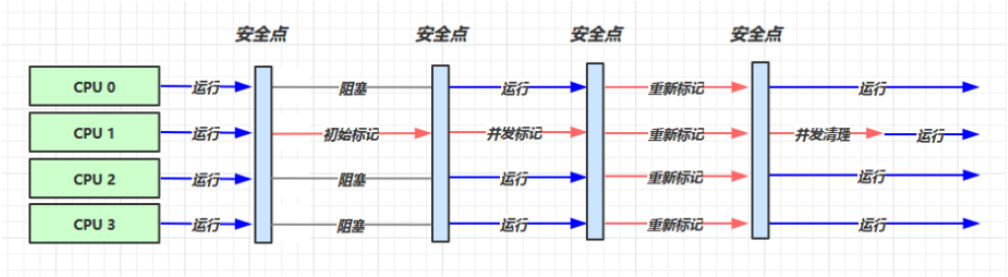
  所以我们必须预留一些空间来保证用户现在在GC过程中的内存开销

  这个参数就表示何时触发垃圾回收。不能内存占100%的时候去GC，因为并发清理时候会产生浮动垃圾，这些浮动垃圾没地方存。在一些JVM中默认为65%

- -XX:+CMSScavengeBeforeRemark

  在重新标记的过程中，有可能新生代的对象会引用老年代的对象，这是重新标记时需要通过新生代的应用扫描老年代的对象，但其实这样很浪费性能，因为在新生代里许多都是垃圾，是要清除的，过多的可达性分析其实是没有必要的

  我们可以使用这个参数表示在重新标记前先用`UseParNewGC`对新生代垃圾进行清理

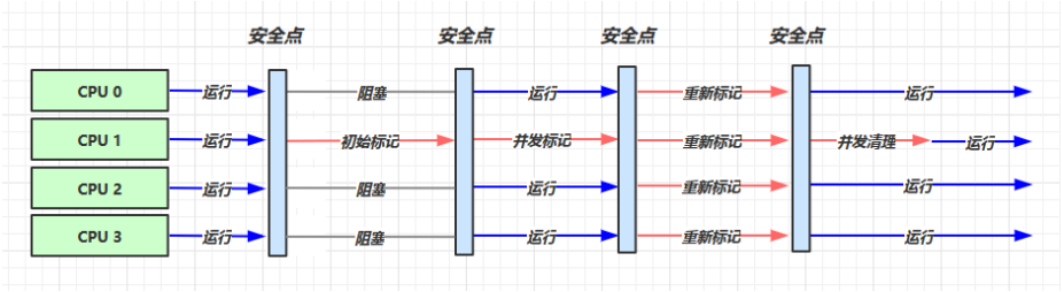


我们看下`CMS`的工作流程：

首先老年代发生了内存不足，现在所有的线程都到达了安全点并暂停下来，这是后 `CMS` 会进行 `初始标记` ，在标记时任然需要STW，但是因为 `初始标记` 只会去标记根对象所以非常快。

当标记完所有GC root后，其他线程就可以恢复运行了，此时 `CMS` 会负责继续标记其他垃圾对象，这里的用户线程会进行并发执行，不用暂停

当达到第三个安全点时，因为用户线程执行可能会打乱标记，所以需要重新标记一次，这里是并行进行的，标记完后，最后又让用户线程恢复执行



> `CMS` 其实有一些问题，因为 `CMS` 是采用标记+清除的垃圾回收算法，所以可能会产生碎片内存过多的情况，进而导致并发失败的现象产生（并发失败是因为预留给用户进程的内存空间不足）
>
> 这时候 `CMS` 会退化成 `SerialOld` ，会让停顿时间突然增加

4.4 G1

定义：Garbage First

- 2004 论文发布
- 2009 JDK 6u14 体验
- 2012 JDK 7u4 官方支持
- 2017 JDK 9 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 JVM 参数

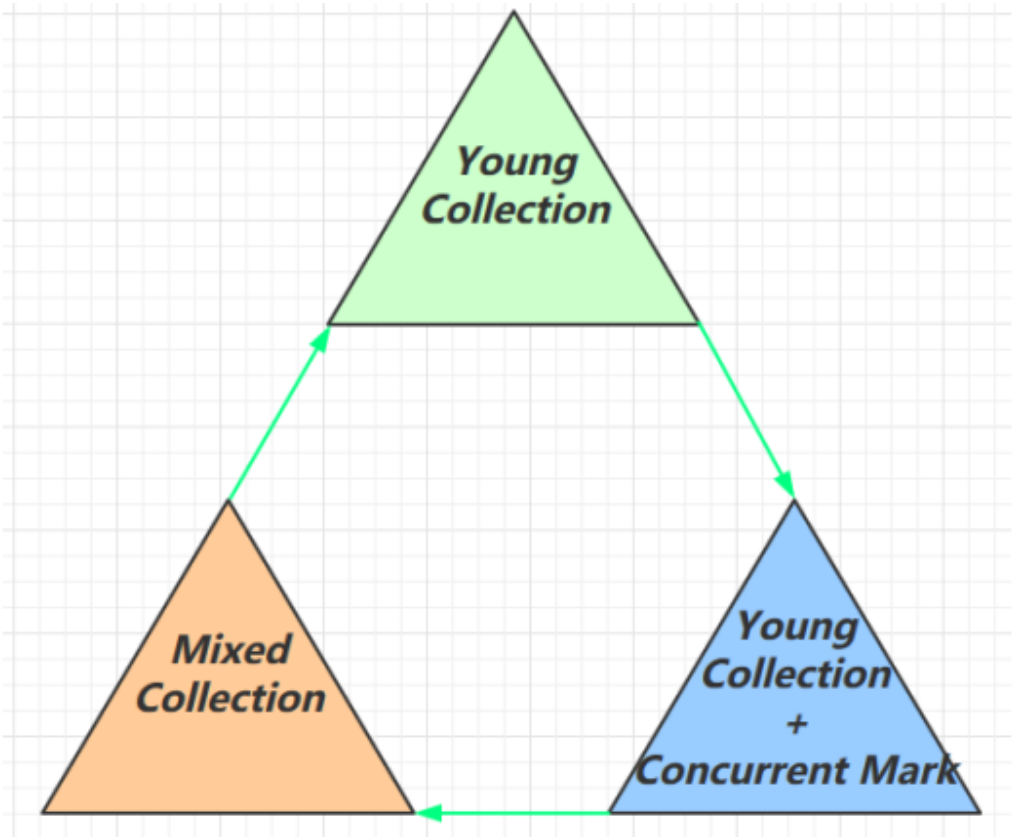
```
-XX:+UseG1GC
-XX:G1HeapRegionSize=size
-XX:MaxGCPauseMillis=time
```

4.4.1 G1垃圾回收阶段

G1回收一共有三个阶段

- Young Collection
- Young Collection + Concurrent Mark
- Mixed Collection

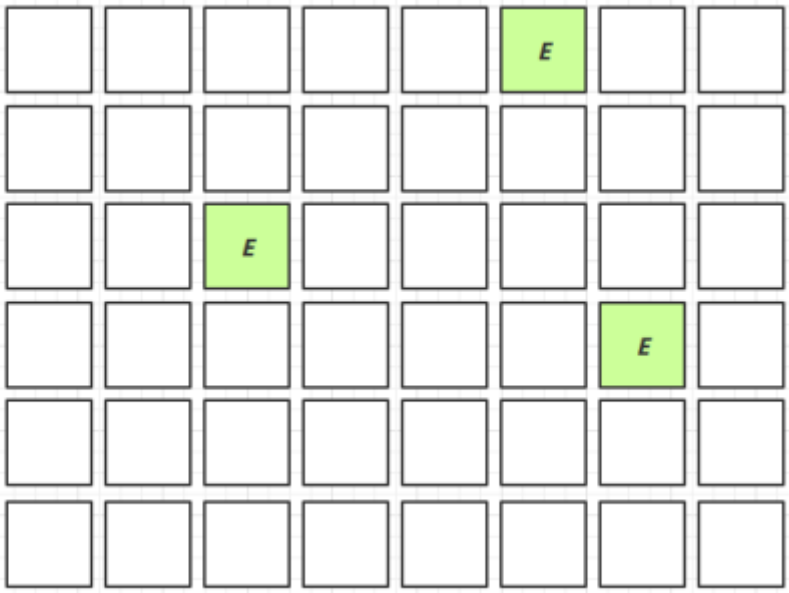
这三个阶段是循环进行的



4.4.2 Young Collection

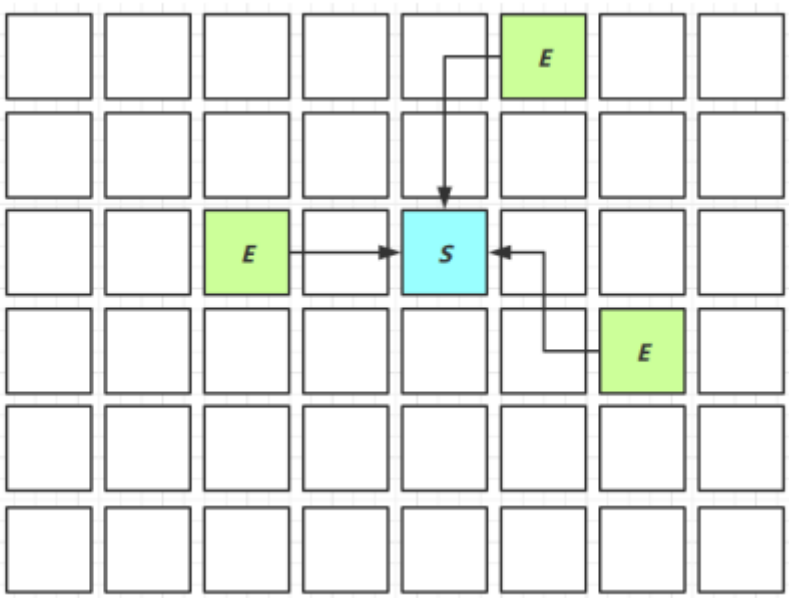
阶段一：新生代垃圾回收

G1开创的基于Region的堆内存布局是它能够实现这个目标的关键。虽然G1也仍是遵循分代收集理论设计的，但其堆内存的布局与其他收集器有非常明显的差异：G1不再坚持固定大小以及固定数量的 分代区域划分，而是把连续的Java堆划分为多个大小相等的独立区域（Region），每一个Region都可以根据需要，扮演新生代的 `Eden` 空间、 `Survivor` 空间，或者老年代空间。收集器能够对扮演不同角色的Region采用不同的策略去处理，这样无论是新创建的对象还是已经存活了一段时间、熬过多次收集的 旧对象都能获取很好的收集效果

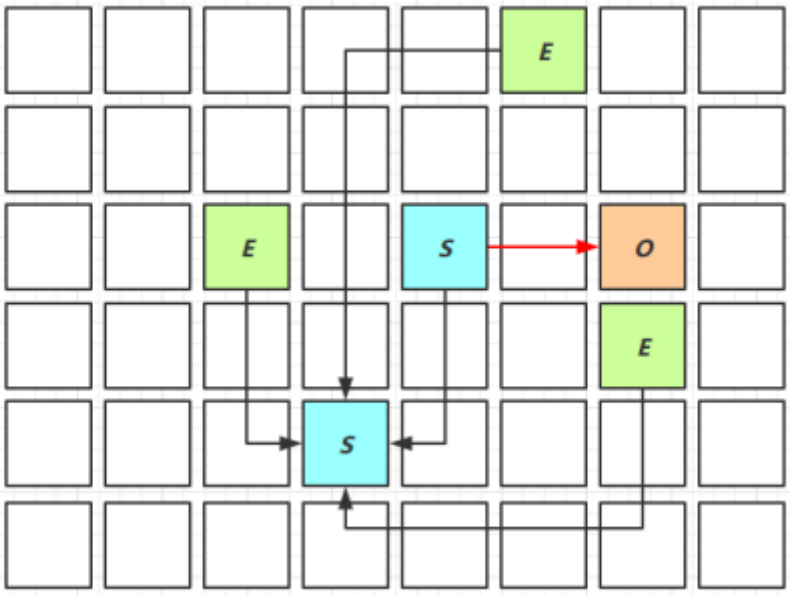


其中新生代垃圾回收的过程为：

当内存紧张的时候就会进行GC，并将 `Eden` 中存活的对象复制到 `Survivor`



当`Survivor`中的空间满后，就会将年龄满了的对象放入到老年区，将年龄不足的对象复制到另外的幸存区

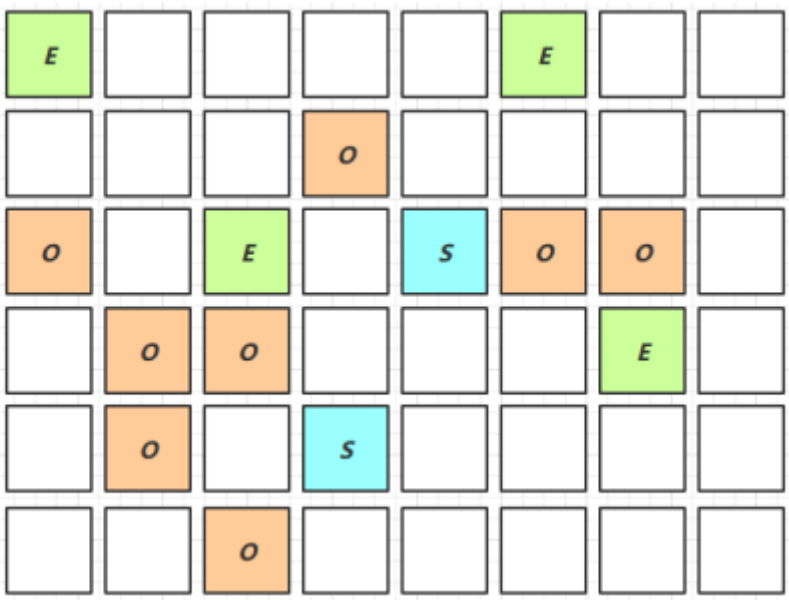


4.4.3 Young Collection + CM

当阶段一结束后就会进入到第二个阶段，新生代垃圾回收和标记阶段

- 在Young GC时会进行 GC Root 的初试标记
- 老年代占用堆空间比例达到了阈值时（45%），进行并发标记（不会STW），有下面的JVM参数决定

```
-XX:InitiatingHeapOccupancyPercent=percent  (默认45%)
```

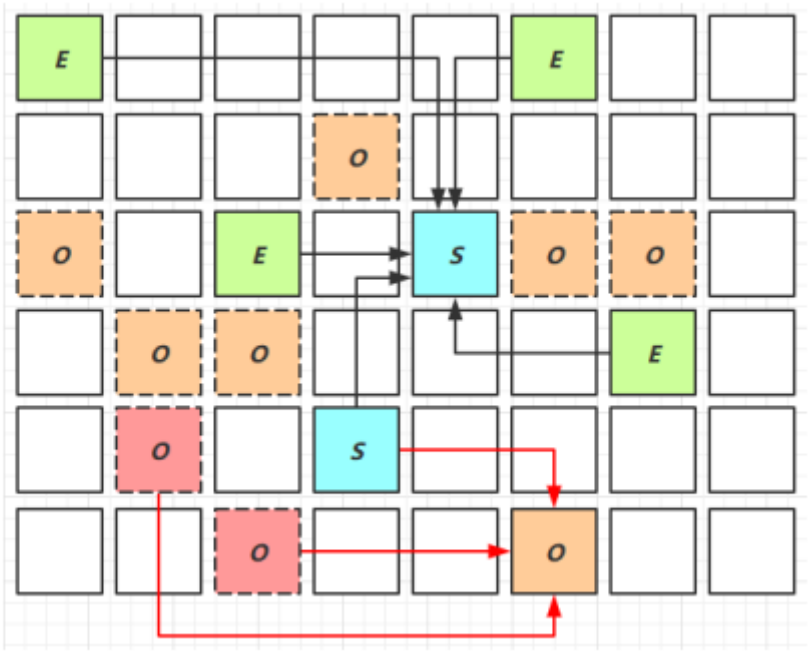


4.4.4 Mixed Collection

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
```



G1会根据用户设置的暂停时间，优先回收垃圾较多的区域，而不是回收所有的区域

4.4.5 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

G1

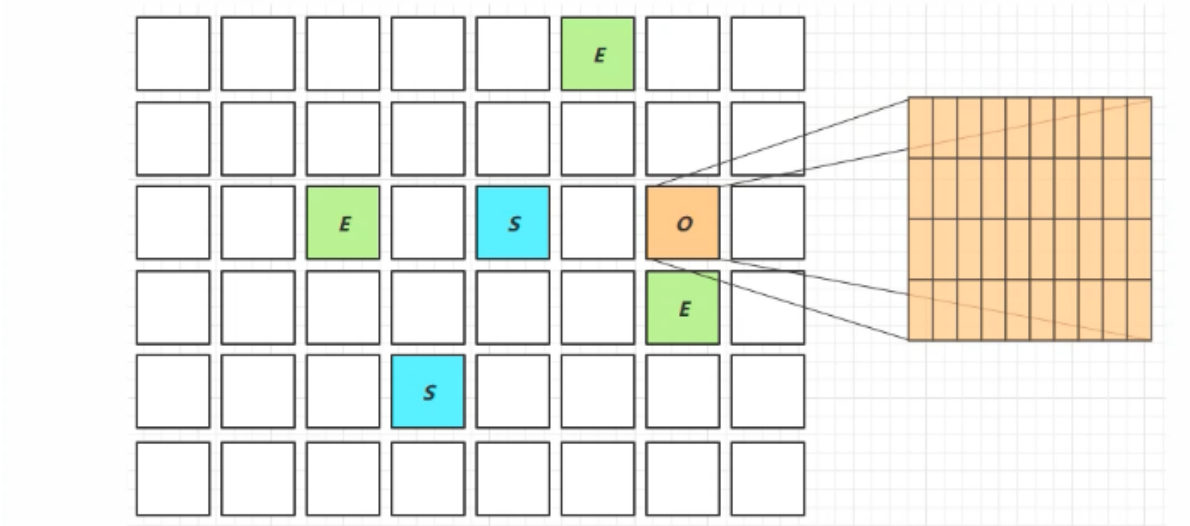
- 新生代内存不足发生的垃圾收集 - minor gc
- 老年代内存不足

4.4.6 Young Collection跨代应用

新生代回收的跨代引用（老年代引用新生代）问题

如果我们遍历老年代，效率是非常低的，所以我们使用一种叫 `卡表` 的技术

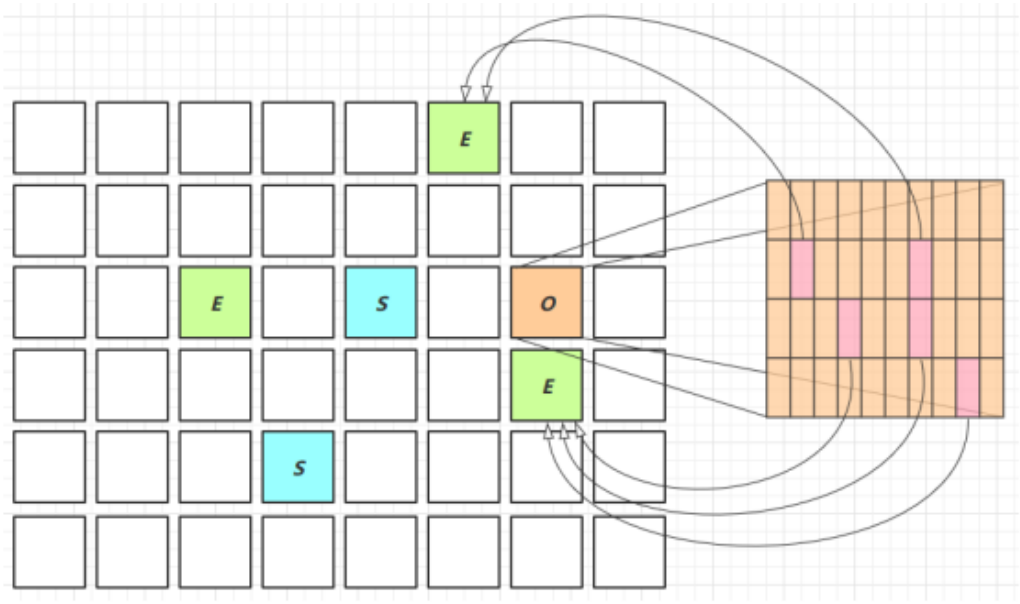
它会将老年代的空间进一步细分，分为许多小块空间，如果这些小块的空间引用了新生代的对象，就会被标记为 脏位，这样做的好处就是减少了扫描范围



- 卡表与 Remembered Set
- 在引用变更时通过 post-write barrier + dirty card queue
- concurrent refinement threads 更新 Remembered Set

卡表是老年代标记新生代的对象， `Remembered Set` 是新生代对象标记自己被那些老年代的对象所引用

在每次对象的引用变更时（比如引用的对象进入了老年代），会在标记 `dirty card` 的时候会加一条写屏障，然后去标记。这是一个异步操作，不会立即去执行，而是会将更新的任务放入 `dirty card queue` 队列之中，再由一个单独的线程去完成这些任务



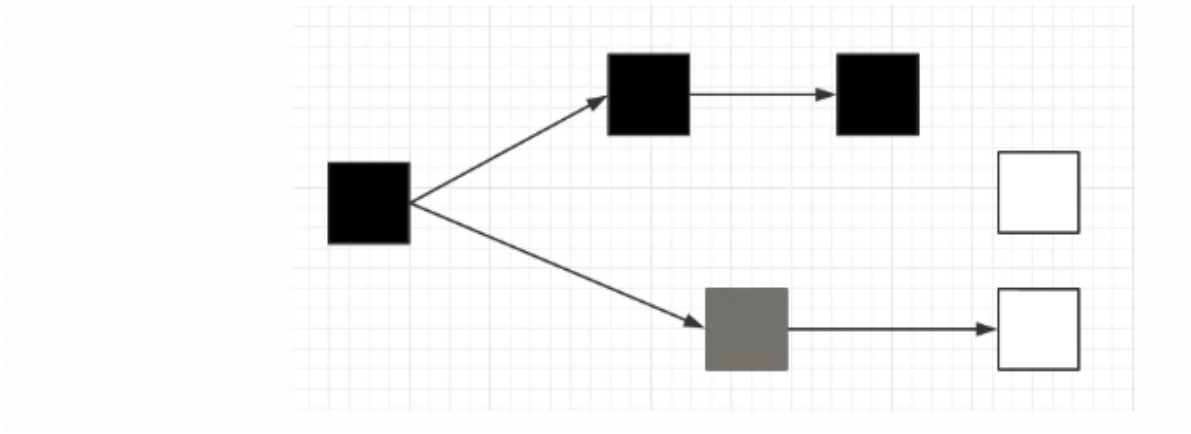
4.4.7 Remark（三色标记法）

pre-write barrier + satb_mark_queue

那么如何标记呢？常用的方法是`三色标记法`

三色标记（Tri-color Marking）作为工具来辅助推导，把遍历对象图过程中遇到的对象，按照“是否访问过”这个条件标记成以下三种颜色：

- 白色：表示对象**尚未被垃圾收集器访问过**。显然在可达性分析刚刚开始的阶段，所有的对象都是白色的，若在分析结束的阶段，仍然是白色的对象，即代表不可达。
- 黑色：表示对象已经被垃圾收集器访问过，且这个对象的所有引用都已经扫描过。黑色的对象代表已经扫描过，它是安全存活的，如果有其他对象引用指向了黑色对象，无须重新扫描一遍。黑色对象不可能直接（不经过灰色对象）指向某个白色对象。
- 灰色：表示对象已经被垃圾收集器访问过，但这个对象上至少存在一个引用还没有被扫描过。



三色标记的过程：

1. 在 `GC` 标记开始的时候，所有的对象均为白色；
2. 在将所有的 `GC Roots` 直接引用的对象标记为灰色集合；
3. 如果判断灰色集合中的对象不存在子引用，则将其放入黑色集合，若存在子引用对象，则将其所有的子引用对象存放到灰色集合，当前对象放入黑色集合。
4. 按照此步骤 3 ，依此类推，直至灰色集合中所有的对象变黑后，本轮标记完成，并且在白色集合内的对象称为不可达对象，即垃圾对象。
5. 标记结束后，为白色的对象为 GC Roots 不可达，可以进行垃圾回收。

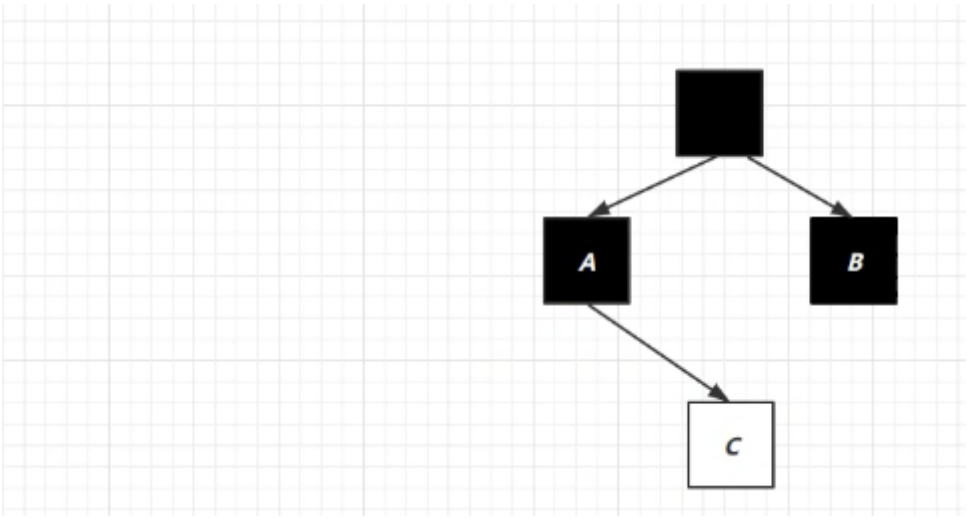
误表情况：

三色标记的过程中，标记线程和用户线程是并发执行的，那么就有可能在我们标记过程中，用户线程修改了引用关系，把原本应该回收的对象错误标记成了存活。(简单来说就是 `GC` 已经标黑的对象，在并发过程中用户线程引用链断掉，导致实际应该是垃圾的白色对象但却依旧是黑的，也就是**浮动垃圾**)。这时产生的垃圾怎么办呢?

答案是本次不处理，留给下次垃圾回收处理。

而**误标**问题，意思就是把本来应该存活的垃圾，标记为了死亡。这就会导致非常严重的错误。那么这类垃圾是怎么产生的呢?

其实也很简单，因为标记的过程是并发执行的，如果现在有用户线程将标记白色的连接断掉，让另外的黑色块连接它，在连接的过程中因为该黑色块可能已经被扫描过了，所以不会再次扫描它，这就导致了误标，并且会将误标的对象删除，这是很严重的问题



什么是误标？当下面两个条件同时满足，会产生误标：

1. 赋值器插入了一条或者多条黑色对象到白色对象的引用
2. 赋值器删除了全部从灰色对象到白色对象的直接引用或者间接引用

如何解决误标呢?

要解决误标的问题，只需要破坏这两个条件中的任意一种即可，分别有两种解决方案：

- 增量更新（Incremental Update）
- 原始快照（Snapshot At The Beginning, STAB）

当对象的应用发生改变时，JVM 会为其加入一个写屏障

增量更新

增量更新要破坏的是第一个条件，当黑色对象插入新的指向白色对象的引用关系时，就将这个新插入的引用记录下来，等并发扫描结束之后，再将这些记录过的引用关系中的黑色对象为根，重新扫描一次。这可以简化理解为，黑色对象一旦新插入了指向白色对象的引用之后，它就变回灰色对象了。

原始快照 (STAB)

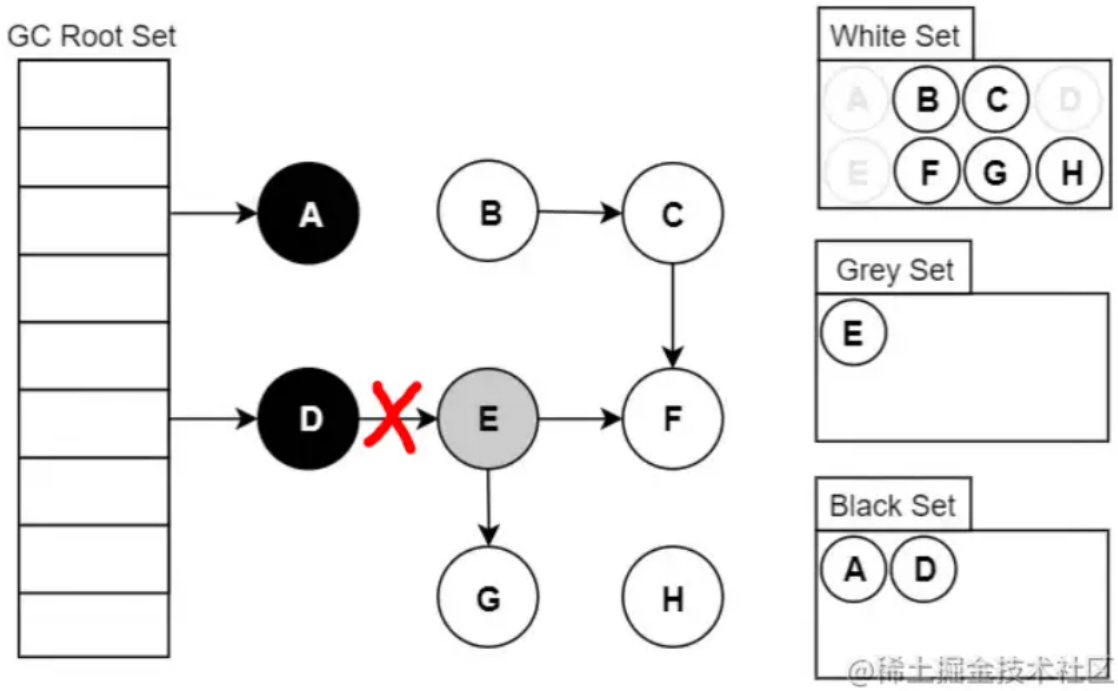
原始快照要破坏的是第二个条件，当灰色对象要删除指向白色对象的引用关系时，就将这个要删除的引用记录下来，在并发扫描结束之后，再将这些记录过的引用关系中的灰色对象为根，重新扫描一次。这也可以简化理解为，无论引用关系删除与否，都会按照刚刚开始扫描那一刻的对象图快照来进行搜索

漏标和多标

对于错标其实细分出来会有两种情况，分别是：漏标和多标

多标-浮动垃圾

如果标记执行到 E 此刻执行了 `object.E = null`



在这个时候， E/F/G 理论上是可以被回收的。但是由于 E 已经变为了**灰色**了，那么它就会继续执行下去。最终的结果就是不会将他们标记为垃圾对象，在本轮标记中存活。

在本轮应该被回收的垃圾没有被回收，这部分被称为“浮动垃圾”。浮动垃圾并不会影响程序的正确性，这些“垃圾”只有在下次垃圾回收触发的时候被清理。

还有在标记过程中产生的新对象，默认被标记为黑色，但是可能在标记过程中变为“垃圾”。这也算是浮动垃圾的一部分。

> 这里不展开了，东西太多了，详情看《深入理解Java虚拟机》周志明

接下来我们看G1的一些优化，这里只看8、9版本的jdk

4.4.8 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

对应虚拟机参数：

```
-XX:+UseStringDeduplication //默认是开启的
```

接下来我们来看一下下面的代码：

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

在jdk1.8中，String的底层实现是char数组，现在new了两个相同的字符串，其实创建了两个char数组，产生了两个垃圾

在G1中对此进行了优化：

- 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[]（只有一个垃圾了）
- 注意，与 String.intern() 不一样
  - String.intern() 关注的是字符串对象
  - 而字符串去重关注的是 char[]
  - 在 JVM 内部，使用了不同的字符串表

4.4.9 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

对于框架程序G1会卸载所有无使用情况的自定义类加载器的所有类

对应虚拟机参数：

```
XX:+ClassUnloadingWithConcurrentMark //默认启用
```

4.4.10 JDK 8u60 回收巨型对象区

- 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

4.4.11 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

4.4.12 JDK 9 更高效的回收

- 250+增强
- 180+bug修复
- G1官方调优文档：https://docs.oracle.com/en/java/javase/12/gctuning/

5. 垃圾回收调优

调优之前需要先了解调优参数：

- 官网调优参数：https://docs.oracle.com/en/java/javase/11/tools/java.html
- 打印系统自带的虚拟机参数

  ```
  java -XX:+PrintFlagsFinal -version | findstr "GC"
  ```

  1.8的参数有：

  ```
  uintx AdaptiveSizeMajorGCDecayTimeScale         = 10
                {product}
  ```

```
    uintx AutoGCSelectPauseMillis                  = 5000
                {product}
     bool BindGCTaskThreadsToCPUs                  = false
                 {product}
    uintx CMSFullGCsBeforeCompaction               = 0
                 {product}
    uintx ConcGCThreads                            = 0
                 {product}
     bool DisableExplicitGC                        = false
                 {product}
     bool ExplicitGCInvokesConcurrent              = false
                 {product}
     bool ExplicitGCInvokesConcurrentAndUnloadsClasses  = false
                     {product}
    uintx G1MixedGCCountTarget                     = 8
                 {product}
    uintx GCDrainStackTargetSize                   = 64
                {product}
    uintx GCHeapFreeLimit                          = 2
                 {product}
    uintx GCLockerEdenExpansionPercent             = 5
                 {product}
     bool GCLockerInvokesConcurrent                = false
                 {product}
    uintx GCLogFileSize                            = 8192
                {product}
    uintx GCPauseIntervalMillis                    = 0
                 {product}
    uintx GCTaskTimeStampEntries                   = 200
                 {product}
    uintx GCTimeLimit                              = 98
                {product}
    uintx GCTimeRatio                              = 99
                {product}
     bool HeapDumpAfterFullGC                      = false
                 {manageable}
     bool HeapDumpBeforeFullGC                     = false
                 {manageable}
    uintx HeapSizePerGCThread                      = 87241520
                {product}
    uintx MaxGCMinorPauseMillis                    = 4294967295
                {product}
    uintx MaxGCPauseMillis                         = 4294967295
                {product}
    uintx NumberOfGCLogFiles                       = 0
                 {product}
     intx ParGCArrayScanChunk                      = 50
                {product}
    uintx ParGCDesiredObjsFromOverflowList         = 20
                {product}
     bool ParGCTrimOverflow                        = true
                {product}
     bool ParGCUseLocalOverflow                    = false
                 {product}
    uintx ParallelGCBufferWastePct                 = 10
                {product}
    uintx ParallelGCThreads                        = 8
                 {product}
```

```
    bool ParallelGCVerbose                          = false
                {product}
    bool PrintClassHistogramAfterFullGC             = false
                {manageable}
    bool PrintClassHistogramBeforeFullGC            = false
                {manageable}
    bool PrintGC                                    = false
                {manageable}
    bool PrintGCApplicationConcurrentTime           = false
                {product}
    bool PrintGCApplicationStoppedTime              = false
                {product}
    bool PrintGCCause                               = true
               {product}
    bool PrintGCDateStamps                          = false
                {manageable}
    bool PrintGCDetails                             = false
                {manageable}
    bool PrintGCID                                  = false
                {manageable}
    bool PrintGCTaskTimeStamps                      = false
                {product}
    bool PrintGCTimeStamps                          = false
                {manageable}
    bool PrintHeapAtGC                              = false
                {product rw}
    bool PrintHeapAtGCExtended                      = false
                {product rw}
    bool PrintJNIGCStalls                           = false
                {product}
    bool PrintParallelOldGCPhaseTimes               = false
                {product}
    bool PrintReferenceGC                           = false
                {product}
    bool ScavengeBeforeFullGC                       = true
               {product}
    bool TraceDynamicGCThreads                      = false
                {product}
    bool TraceParallelOldGCTasks                    = false
                {product}
    bool UseAdaptiveGCBoundary                      = false
                {product}
    bool UseAdaptiveSizeDecayMajorGCCost            = true
               {product}
    bool UseAdaptiveSizePolicyWithSystemGC          = false
                {product}
    bool UseAutoGCSelectPolicy                      = false
                {product}
    bool UseConcMarkSweepGC                         = false
                {product}
    bool UseDynamicNumberOfGCThreads                = false
                {product}
    bool UseG1GC                                    = false
                {product}
    bool UseGCLogFileRotation                       = false
                {product}
    bool UseGCOverheadLimit                         = true
               {product}
```

```
60      bool UseGCTaskAffinity                          = false
                {product}
61      bool UseMaximumCompactionOnSystemGC             = true
                {product}
62      bool UseParNewGC                                = false
                {product}
63      bool UseParallelGC                             := true
                {product}
64      bool UseParallelOldGC                           = true
                {product}
65      bool UseSerialGC                                = false
                {product}
66  openjdk version "1.8.0_302"
67  OpenJDK Runtime Environment Corretto-8.302.08.1 (build 1.8.0_302-b08)
68  OpenJDK 64-Bit Server VM Corretto-8.302.08.1 (build 25.302-b08, mixed
    mode)
69
```

5.1 调优方向

内存 锁竞争 cpu 占用 io

5.2 调优目标

- 【低延迟】还是【高吞吐量】，选择合适的回收器
- CMS，G1，ZGC
- ParallelGC

也可以从虚拟机下手，例如不使用Hotspot虚拟机，使用Zing虚拟机

5.3 最快的GC是不发生GC

我们在查看FullGC前后的内存暂用后，需要考虑下面几个问题

- 数据是不是太多了？

  例如JDBC中的resultSet，不能读太多的数据，需要limit限制

- 数据表示是否太臃肿？

  例如：对象图、对象大小（Java中最小的Object占用16bit，包装类最小16bit）

- 是否存在内存泄露？

  例如：定义了一个静态的Map，一直往里面添加东西

  我们可以用软弱引用管理这些东西

5.4 新生代调优

因为新生代中对象的产生和消亡比较频繁，所以先从这里开始调优

新生代的特点

- 所有的 new 操作的内存分配非常廉价

  所有的对象的产生都是在`Eden`中，new的时候会先检查在`tlab(thread location buffer)`线程局部缓冲区，这块内存其实是为了避免内存分配时的线程安全问题

- 死亡对象的回收代价是零（新生代中的gc算法都是复制算法）

- 大部分对象用过即死
- Minor GC 的时间远远低于 Full GC

是不是我们直接将新生代的空间设置的越大就越好呢?

-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

oracle官方建议:

Oracle recommends that you keep the size for the young generation greater than 25% and less than 50% of the overall heap size

新生代占比25%到50%

> 所以新生代第一步调优就是调整新生代和老年代的占比

我们可以估算新生代的内存大概占比

- 新生代能容纳所有【并发量 * (请求-响应)】的数据
- 幸存区大到能保留【当前活跃对象+需要晋升对象】
- 新生代空间尽量大

初试之外我们还应该:

- 晋升阈值配置得当,让长时间存活对象尽快晋升

  ```
  -XX:MaxTenuringThreshold=threshold // 调整晋升阈值
  XX:+PrintTenuringDistribution //打印幸存区中对象信息
  ```

  例如:

  ```
  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
  ...
  ```

5.5 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

  该参数表示比例,控制老年代空间占用达到全部的多少时进行full gc

  值越低,触发gc的时间就越早,一般设置在75%-80%之间,因为需要留空间给浮动垃圾

  ```
  -XX:CMSInitiatingOccupancyFraction=percent
  ```

在垃圾回收的某些阶段需要stw(初始标记，重新标记)，在某些阶段不需要stw(并发标记，并发清理阶段)，用户线程可以和垃圾收集器线程同时执行，降低用户线程的暂停时间

5.6 案例

- 案例1 Full GC 和 Minor GC频繁
- 案例2 请求高峰期发生 Full GC，单次暂停时间特别长 （CMS）
- 案例3 老年代充裕情况下，发生 Full GC （CMS jdk1.7）

> 案例1 Full GC 和 Minor GC频繁

如果我们现在的项目发生gc非常频繁，甚至达到了每分钟上百次，这是我们需要先查看gc是发生在新生代还是老年代

如果是新生代，试着先调大空间

增大了新生代的内存导致minorgc更少触发，并且survivor区增大，就不会让本不是生命周期那么长的对象进入老年区，从而给老年区节省空间，进一步就减少了老年区出发fullGC

> 案例2 请求高峰期发生 Full GC，单次暂停时间特别长 （CMS）

首先先查看CMS在标记的时候是否时间消耗异常

如果新生对象比较多，会导致老年代CMS重新标记的时间过长，可以在CMS标记之前对新生代的垃圾做一次清理

```
-XX:+CMSScavengeBeforeRemark //在标记前先minorGC
```

初始标记我只标记GCRoot对象 其他对象我可以让并发标记的线程慢慢标记且不影响其他线程执行执行，对于并发标记期间产生变更的对象加入队列，重新标记仅扫描队列中的对象，效率自然就快了

- 初始标记：仅仅标记GC ROOTS的直接关联对象，并且世界暂停
- 并发标记：使用GC ROOTS TRACING算法，进行跟踪标记，世界不暂停
- 重新标记，因为之前并发标记，其他用户线程不暂停，可能产生了新垃圾，所以重新标记，世界暂停

> 案例3 老年代充裕情况下，发生 Full GC （CMS jdk1.7）