垃圾回收相关概念

System.gc() 的理解

1. 在默认情况下，通过System.gc()者Runtime.getRuntime().gc() 的调用，**会显式触发Full GC**，同时对老年代和新生代进行回收，尝试释放被丢弃对象占用的内存。
2. 然而System.gc()调用附带一个免责声明，无法保证对垃圾收集器的调用(不能确保立即生效)
3. JVM实现者可以通过System.gc() 调用来决定JVM的GC行为。而一般情况下，垃圾回收应该是自动进行的，**无须手动触发，否则就太过于麻烦了。**在一些特殊情况下，如我们正在编写一个性能基准，我们可以在运行之间调用System.gc()

代码示例：手动执行 GC 操作

```
public class SystemGCTest {
    public static void main(String[] args) {
        new SystemGCTest();
        System.gc();//提醒jvm的垃圾回收器执行gc,但是不确定是否马上执行gc
        //与Runtime.getRuntime().gc();的作用一样。

//        System.runFinalization();//强制调用使用引用的对象的finalize()方法
    }
    //如果发生了GC，这个finalize()一定会被调用
    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("SystemGCTest 重写了finalize()");
    }
}
```

输出结果不确定：有时候会调用 finalize() 方法，有时候并不会调用

```
SystemGCTest 重写了finalize()
或
空
```

手动 GC 理解不可达对象的回收行为

```
//加上参数：  -XX:+PrintGCDetails
public class LocalVarGC {
    public void localvarGC1() {
        byte[] buffer = new byte[10 * 1024 * 1024];//10MB
        System.gc();
```

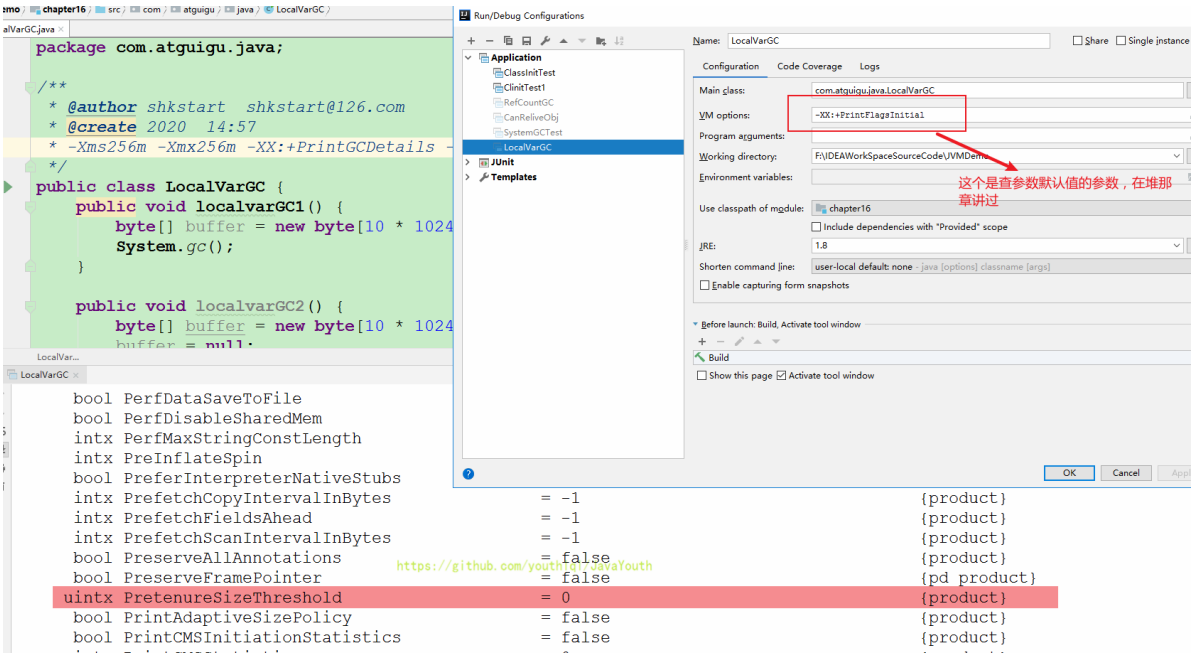
```
    }

    public void localvarGC2() {
        byte[] buffer = new byte[10 * 1024 * 1024];
        buffer = null;
        System.gc();
    }

    public void localvarGC3() {
        {
            byte[] buffer = new byte[10 * 1024 * 1024];
        }
        System.gc();
    }

    public void localvarGC4() {
        {
            byte[] buffer = new byte[10 * 1024 * 1024];
        }
        int value = 10;
        System.gc();
    }

    public void localvarGC5() {
        localvarGC1();
        System.gc();
    }

    public static void main(String[] args) {
        LocalVarGC local = new LocalVarGC();
        //通过在main方法调用这几个方法进行测试
        local.localvarGC1();
    }
}
```

JVM参数:

`-Xms256m -Xmx256m -XX:+PrintGCDetails -XX:PretenureSizeThreshold=15m`

1、第四个参数是设置大对象直接进入老年代的阈值，由于我的电脑8G和视频里老师的电脑16G不太一样。我测试的时候10M的数组都是直接进入到了老年代，为了保持一样的效果，我同时设置了堆内存和大对象阈值，尽量和宋红康老师保持一致

2、我也查过了大对象阈值的默认值



我不太懂这个默认值为啥是0，我猜测可能是代表什么比例，目前也没有搜到相关的东西。这个不太重要，暂时就没有太深究，希望读者有知道的可以告知我一声。

> 看不懂GC日志请看笔者的 **堆**那篇文章

1、调用 localvarGC1() 方法

执行 System.gc() 仅仅是将年轻代的 buffer 数组对象放到了老年代，buffer对象仍然没有回收

```
[GC (System.gc()) [PSYoungGen: 15492K->10728K(76288K)] 15492K->11000K(251392K),
0.0066473 secs] [Times: user=0.08 sys=0.02, real=0.01 secs]
[Full GC (System.gc()) [PSYoungGen: 10728K->0K(76288K)] [ParOldGen: 272K-
>10911K(175104K)] 11000K->10911K(251392K), [Metaspace: 3492K->3492K(1056768K)],
0.0097940 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
Heap
 PSYoungGen      total 76288K, used 655K [0x00000000fab00000,
0x0000000100000000, 0x0000000100000000)
  eden space 65536K, 1% used
[0x00000000fab00000,0x00000000faba3ee8,0x00000000feb00000)
  from space 10752K, 0% used
[0x00000000feb00000,0x00000000feb00000,0x00000000ff580000)
  to   space 10752K, 0% used
[0x00000000ff580000,0x00000000ff580000,0x0000000100000000)
 ParOldGen       total 175104K, used 10911K [0x00000000f0000000,
0x00000000fab00000, 0x00000000fab00000)
  object space 175104K, 6% used
[0x00000000f0000000,0x00000000f0aa7d08,0x00000000fab00000)
 Metaspace       used 3498K, capacity 4498K, committed 4864K, reserved 1056768K
  class space    used 387K, capacity 390K, committed 512K, reserved 1048576K
```

2、调用 localvarGC2() 方法

由于 buffer 数组对象没有引用指向它，执行 System.gc() 将被回收

```
[GC (System.gc()) [PSYoungGen: 15492K->808K(76288K)] 15492K->816K(251392K),
0.0294475 secs] [Times: user=0.00 sys=0.00, real=0.04 secs]
[Full GC (System.gc()) [PSYoungGen: 808K->0K(76288K)] [ParOldGen: 8K-
>640K(175104K)] 816K->640K(251392K), [Metaspace: 3385K->3385K(1056768K)],
0.0054210 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
Heap
 PSYoungGen      total 76288K, used 1966K [0x00000000fab00000,
0x0000000100000000, 0x0000000100000000)
  eden space 65536K, 3% used
[0x00000000fab00000,0x00000000faceb9e0,0x00000000feb00000)
  from space 10752K, 0% used
[0x00000000feb00000,0x00000000feb00000,0x00000000ff580000)
  to   space 10752K, 0% used
[0x00000000ff580000,0x00000000ff580000,0x0000000100000000)
 ParOldGen       total 175104K, used 640K [0x00000000f0000000,
0x00000000fab00000, 0x00000000fab00000)
  object space 175104K, 0% used
[0x00000000f0000000,0x00000000f00a01a8,0x00000000fab00000)
 Metaspace       used 3392K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 379K, capacity 388K, committed 512K, reserved 1048576K
```

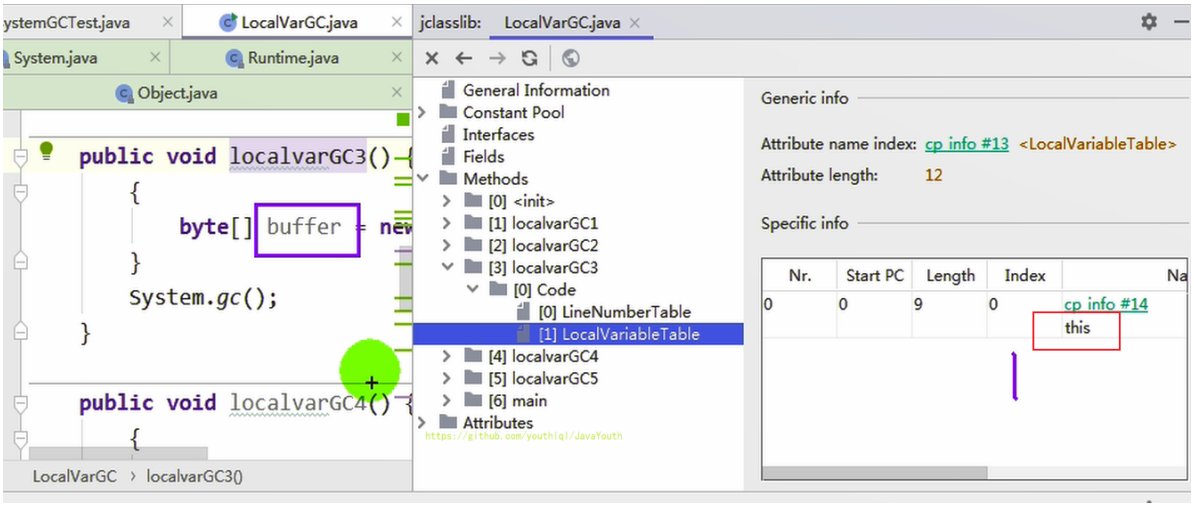
3、调用 localvarGC3() 方法

虽然出了代码块的作用域，但是 buffer 数组对象并没有被回收

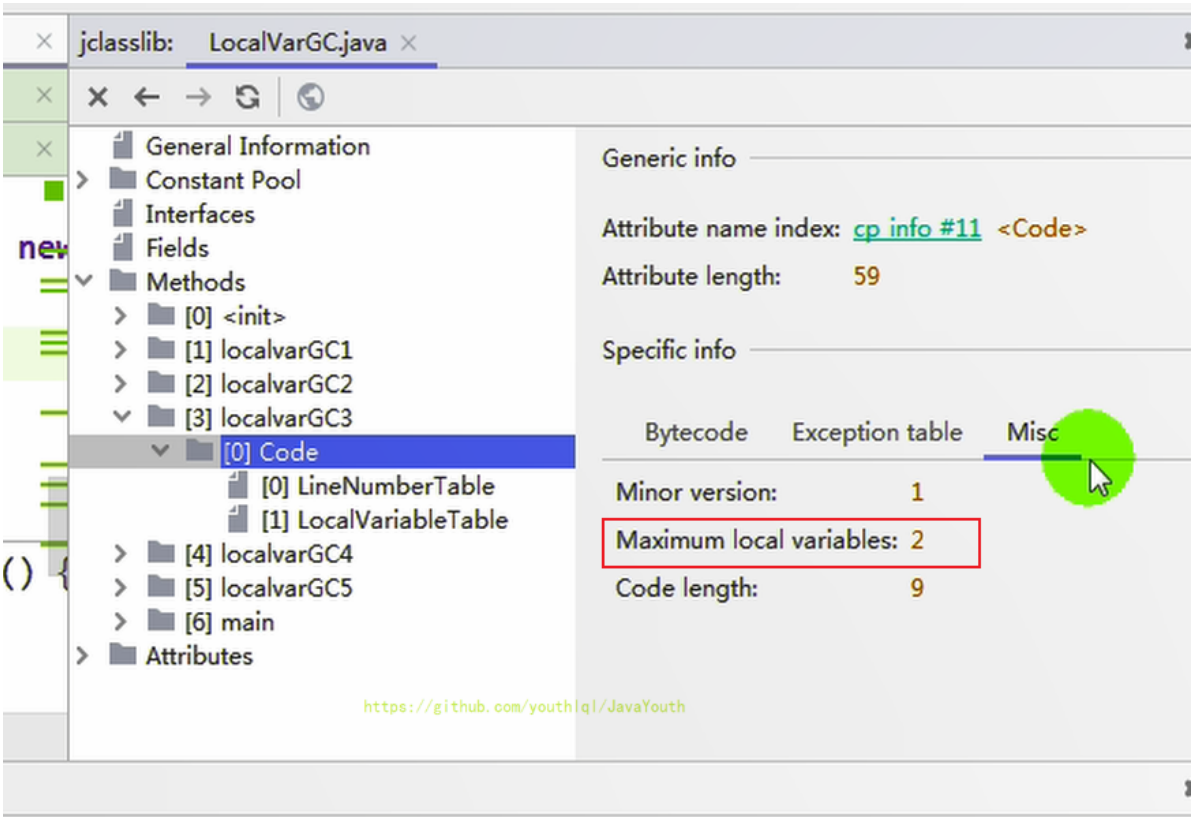
```
[GC (System.gc()) [PSYoungGen: 15492K->840K(76288K)] 15492K->11088K(251392K),
0.0070281 secs] [Times: user=0.08 sys=0.00, real=0.01 secs]
[Full GC (System.gc()) [PSYoungGen: 840K->0K(76288K)] [ParOldGen: 10248K-
>10900K(175104K)] 11088K->10900K(251392K), [Metaspace: 3386K->3386K(1056768K)],
0.0084464 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
Heap
 PSYoungGen      total 76288K, used 1966K [0x00000000fab00000,
0x0000000100000000, 0x0000000100000000)
  eden space 65536K, 3% used
[0x00000000fab00000,0x00000000faceb9e0,0x00000000feb00000)
  from space 10752K, 0% used
[0x00000000feb00000,0x00000000feb00000,0x00000000ff580000)
  to   space 10752K, 0% used
[0x00000000ff580000,0x00000000ff580000,0x0000000100000000)
 ParOldGen       total 175104K, used 10900K [0x00000000f0000000,
0x00000000fab00000, 0x00000000fab00000)
  object space 175104K, 6% used
[0x00000000f0000000,0x00000000f0aa52e8,0x00000000fab00000)
 Metaspace       used 3393K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 379K, capacity 388K, committed 512K, reserved 1048576K
```

原因：

1、来看看字节码：实例方法局部变量表第一个变量肯定是 this



2、你有没有看到，局部变量表的大小是 2。但是局部变量表里只有一个索引为0的啊？那索引为1的是哪个局部变量呢？实际上索引为1的位置是buffer在占用着，执行 System.gc() 时，栈中还有 buffer 变量指向堆中的字节数组，所以没有进行GC



3、那么这种代码块的情况，什么时候会被GC呢？我们来看第四个方法

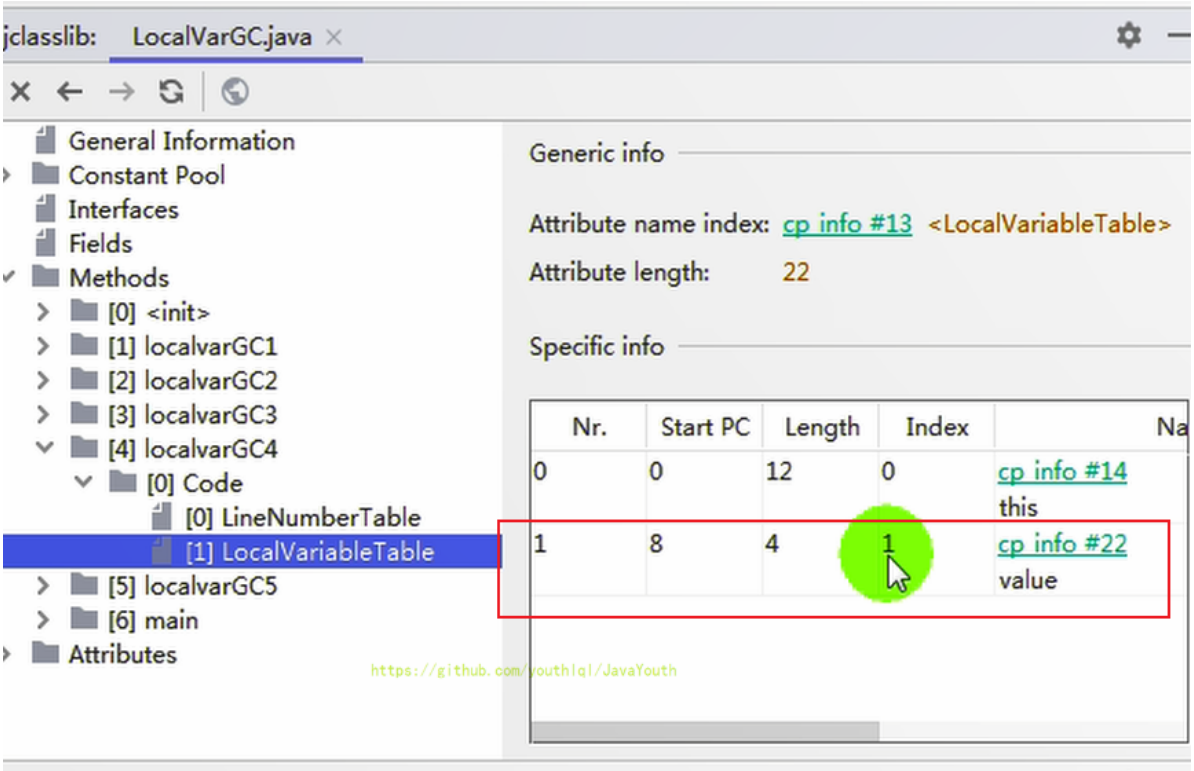
4、调用 localvarGC4() 方法

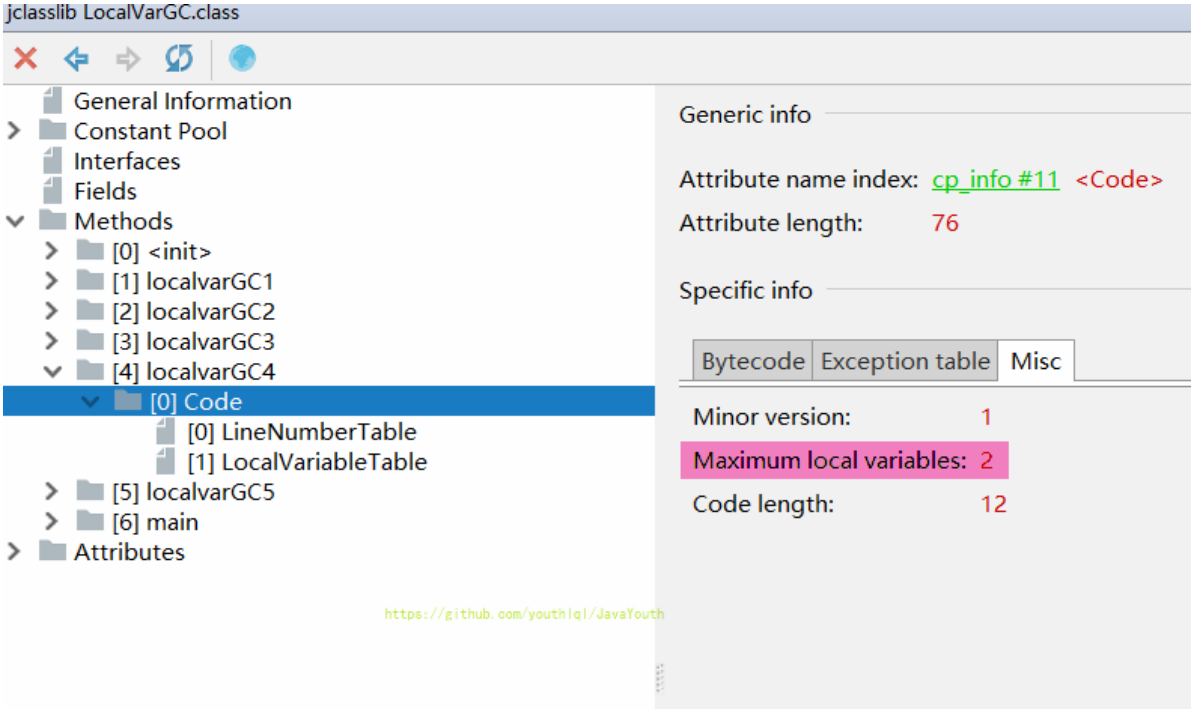
```
[GC (System.gc()) [PSYoungGen: 15492K->776K(76288K)] 15492K->784K(251392K),
0.0009430 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (System.gc()) [PSYoungGen: 776K->0K(76288K)] [ParOldGen: 8K-
>646K(175104K)] 784K->646K(251392K), [Metaspace: 3485K->3485K(1056768K)],
0.0065829 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
Heap
 PSYoungGen      total 76288K, used 1966K [0x00000000fab00000,
0x0000000100000000, 0x0000000100000000)
  eden space 65536K, 3% used
[0x00000000fab00000,0x00000000faceb9f8,0x00000000feb00000)
  from space 10752K, 0% used
[0x00000000feb00000,0x00000000feb00000,0x00000000ff580000)
  to   space 10752K, 0% used
[0x00000000ff580000,0x00000000ff580000,0x0000000100000000)
 ParOldGen       total 175104K, used 646K [0x00000000f0000000,
0x00000000fab00000, 0x00000000fab00000)
  object space 175104K, 0% used
[0x00000000f0000000,0x00000000f00a1b88,0x00000000fab00000)
 Metaspace       used 3498K, capacity 4498K, committed 4864K, reserved 1056768K
  class space    used 387K, capacity 390K, committed 512K, reserved 1048576K
```

Q：就多定义了一个局部变量 value ，就可以把字节数组回收了呢?

A：局部变量表长度为 2 ，这说明了出了代码块时，buffer 就出了其作用域范围，此时没有为 value 开启新的槽，value 变量直接占据了 buffer 变量的槽（Slot），导致堆中的字节数组没有引用再指向它，执行 System.gc() 时被回收。看，value 位于局部变量表中索引为 1 的位置。value这个局部变量把原本属于buffer的slot给占用了，这样栈上就没有buffer变量指向 `new byte[10 * 1024 * 1024]` 实例了。

> 这点看不懂的可以看我前面的文章：虚拟机栈 --> Slot的重复利用





调用 localvarGC5() 方法

局部变量除了方法范围就是失效了，堆中的字节数组铁定被回收

```
[GC (System.gc()) [PSYoungGen: 15492K->840K(76288K)] 15492K->11088K(251392K),
0.0070281 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
[Full GC (System.gc()) [PSYoungGen: 840K->0K(76288K)] [ParOldGen: 10248K-
>10911K(175104K)] 11088K->10911K(251392K), [Metaspace: 3492K->3492K(1056768K)],
0.0082011 secs] [Times: user=0.03 sys=0.03, real=0.01 secs]
[GC (System.gc()) [PSYoungGen: 0K->0K(76288K)] 10911K->10911K(251392K),
0.0004440 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (System.gc()) [PSYoungGen: 0K->0K(76288K)] [ParOldGen: 10911K-
>671K(175104K)] 10911K->671K(251392K), [Metaspace: 3492K->3492K(1056768K)],
0.0108555 secs] [Times: user=0.08 sys=0.02, real=0.01 secs]
Heap
 PSYoungGen      total 76288K, used 655K [0x00000000fab00000,
0x0000000100000000, 0x0000000100000000)
  eden space 65536K, 1% used
[0x00000000fab00000,0x00000000faba3ee8,0x00000000feb00000)
  from space 10752K, 0% used
[0x00000000ff580000,0x00000000ff580000,0x0000000100000000)
  to   space 10752K, 0% used
[0x00000000feb00000,0x00000000feb00000,0x00000000ff580000)
 ParOldGen       total 175104K, used 671K [0x00000000f0000000,
0x00000000fab00000, 0x00000000fab00000)
  object space 175104K, 0% used
[0x00000000f0000000,0x00000000f00a7cf8,0x00000000fab00000)
 Metaspace       used 3499K, capacity 4502K, committed 4864K, reserved 1056768K
  class space    used 387K, capacity 390K, committed 512K, reserved 1048576K
```

内存溢出与内存泄漏

内存溢出

1. 内存溢出相对于内存泄漏来说，尽管更容易被理解，但是同样的，内存溢出也是引发程序崩溃的罪魁祸首之一。
2. 由于GC一直在发展，所有一般情况下，除非应用程序占用的内存增长速度非常快，造成垃圾回收已经跟不上内存消耗的速度，否则不太容易出现OOM的情况。
3. 大多数情况下，GC会进行各种年龄段的垃圾回收，实在不行了就放大招，来一次独占式的Full GC操作，这时候会回收大量的内存，供应用程序继续使用。
4. Javadoc中对OutofMemoryError的解释是，没有空闲内存，并且垃圾收集器也无法提供更多内存。

内存溢出（OOM）原因分析

首先说没有空闲内存的情况：说明Java虚拟机的堆内存不够。原因有二：

1. Java虚拟机的堆内存设置不够。
    - 比如：可能存在内存泄漏问题；也很有可能就是堆的大小不合理，比如我们要处理比较可观的数据量，但是没有显式指定JVM堆大小或者指定数值偏小。我们可以通过参数-Xms 、-Xmx来调整。
2. 代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在被引用）
    - 对于老版本的Oracle JDK，因为永久代的大小是有限的，并且JVM对永久代垃圾回收（如，常量池回收、卸载不再需要的类型）非常不积极，所以当我们不断添加新类型的时候，永久代出现OutOfMemoryError也非常多见。尤其是在运行时存在大量动态类型生成的场合；类似intern字符串缓存占用太多空间，也会导致OOM问题。对应的异常信息，会标记出来和永久代相关：“java.lang.OutOfMemoryError:PermGen space"。
    - 随着元数据区的引入，方法区内存已经不再那么窘迫，所以相应的OOM有所改观，出现OOM，异常信息则变成了：“java.lang.OutofMemoryError:Metaspace"。直接内存不足，也会导致OOM。

1. 这里面隐含着一层意思是，在抛出OutofMemoryError之前，通常垃圾收集器会被触发，尽其所能去清理出空间。
    - 例如：在引用机制分析中，涉及到JVM会去尝试**回收软引用指向的对象**等。
    - 在java.nio.Bits.reserveMemory()方法中，我们能清楚的看到，System.gc()会被调用，以清理空间。
2. 当然，也不是在任何情况下垃圾收集器都会被触发的
    - 比如，我们去分配一个超大对象，类似一个超大数组超过堆的最大值，JVM可以判断出垃圾收集并不能解决这个问题，所以直接抛出OutofMemoryError。

内存泄漏

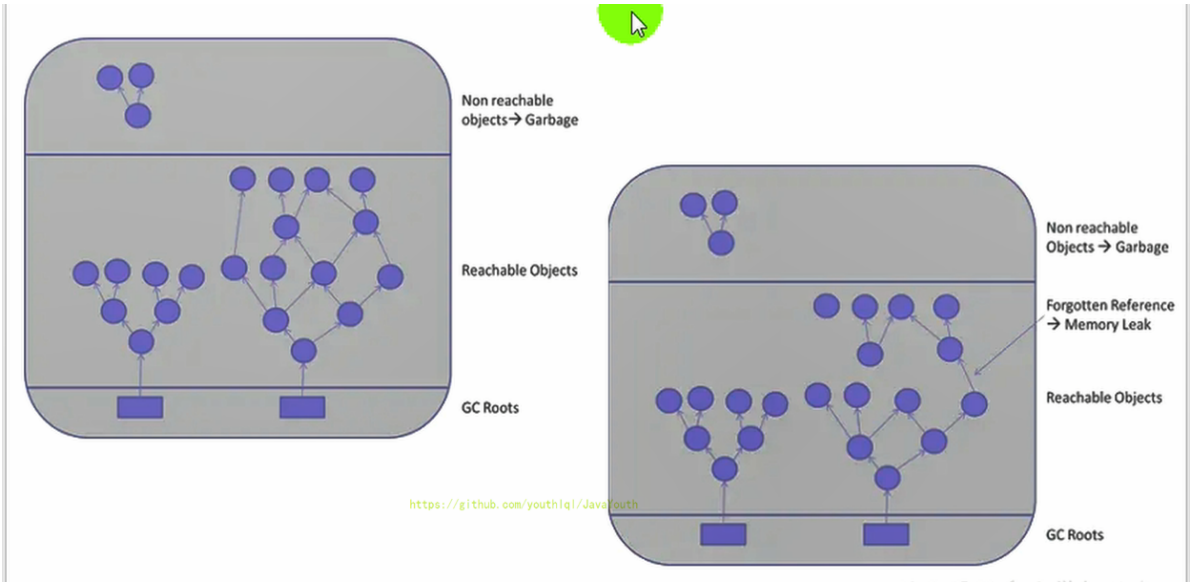
1. 也称作“存储渗漏”。严格来说，**只有对象不会再被程序用到了，但是GC又不能回收他们的情况，才叫内存泄漏。**

2. 但实际情况很多时候一些不太好的实践（或疏忽）会导致对象的生命周期变得很长甚至导致OOM，也可以叫做宽泛意义上的“内存泄漏”。
3. 尽管内存泄漏并不会立刻引起程序崩溃，但是一旦发生内存泄漏，程序中的可用内存就会被逐步蚕食，直至耗尽所有内存，最终出现OutofMemory异常，导致程序崩溃。
4. 注意，这里的存储空间并不是指物理内存，而是指虚拟内存大小，这个虚拟内存大小取决于磁盘交换区设定的大小。

内存泄露官方例子

左边的图：Java使用可达性分析算法，最上面的数据不可达，就是需要被回收的对象。

右边的图：后期有一些对象不用了，按道理应该断开引用，但是存在一些链没有断开（图示中的Forgotten Reference Memory Leak），从而导致没有办法被回收。



常见例子

1. 单例模式
    - 单例的生命周期和应用程序是一样长的，所以在单例程序中，如果持有对外部对象的引用的话，那么这个外部对象是不能被回收的，则会导致内存泄漏的产生。
2. 一些提供close()的资源未关闭导致内存泄漏
    - 数据库连接 dataSourse.getConnection()，网络连接socket和io连接必须手动close，否则是不能被回收的。

Stop the World

1. Stop-the-World，简称STW，指的是GC事件发生过程中，会产生应用程序的停顿。**停顿产生时整个应用程序线程都会被暂停，没有任何响应**，有点像卡死的感觉，这个停顿称为STW。
2. 可达性分析算法中枚举根节点（GC Roots）会导致所有Java执行线程停顿，为什么需要停顿所有Java 执行线程呢?
    - 分析工作必须在一个能确保一致性的快照中进行

- 一致性指整个分析期间整个执行系统看起来像被冻结在某个时间点上
- **如果出现分析过程中对象引用关系还在不断变化，则分析结果的准确性无法保证**

3. 被STW中断的应用程序线程会在完成GC之后恢复，频繁中断会让用户感觉像是网速不快造成电影卡带一样，所以我们需要减少STW的发生。

1. STW事件和采用哪款GC无关，所有的GC都有这个事件。
2. 哪怕是G1也不能完全避免Stop-the-world情况发生，只能说垃圾回收器越来越优秀，回收效率越来越高，尽可能地缩短了暂停时间。
3. STW是JVM在**后台自动发起和自动完成**的。在用户不可见的情况下，把用户正常的工作线程全部停掉。
4. 开发中不要用System.gc()，这会导致Stop-the-World的发生。

代码感受 Stop the World

```
public class StopTheWorldDemo {
    public static class WorkThread extends Thread {
        List<byte[]> list = new ArrayList<byte[]>();

        public void run() {
            try {
                while (true) {
                    for(int i = 0;i < 1000;i++){
                        byte[] buffer = new byte[1024];
                        list.add(buffer);
                    }

                    if(list.size() > 10000){
                        list.clear();
                        System.gc();//会触发full gc，进而会出现STW事件

                    }
                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }

    public static class PrintThread extends Thread {
        public final long startTime = System.currentTimeMillis();

        public void run() {
            try {
                while (true) {
                    // 每秒打印时间信息
                    long t = System.currentTimeMillis() - startTime;
                    System.out.println(t / 1000 + "." + t % 1000);
                    Thread.sleep(1000);
                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
```

```
    }

    public static void main(String[] args) {
        WorkThread w = new WorkThread();
        PrintThread p = new PrintThread();
        w.start();
        p.start();
    }
}
```

关闭工作线程 w，观察输出：当前时间间隔与上次时间间隔**基本**是每隔1秒打印一次

```
0.1
1.1
2.2
3.2
4.3
5.3
6.3
7.3

Process finished with exit code -1
```

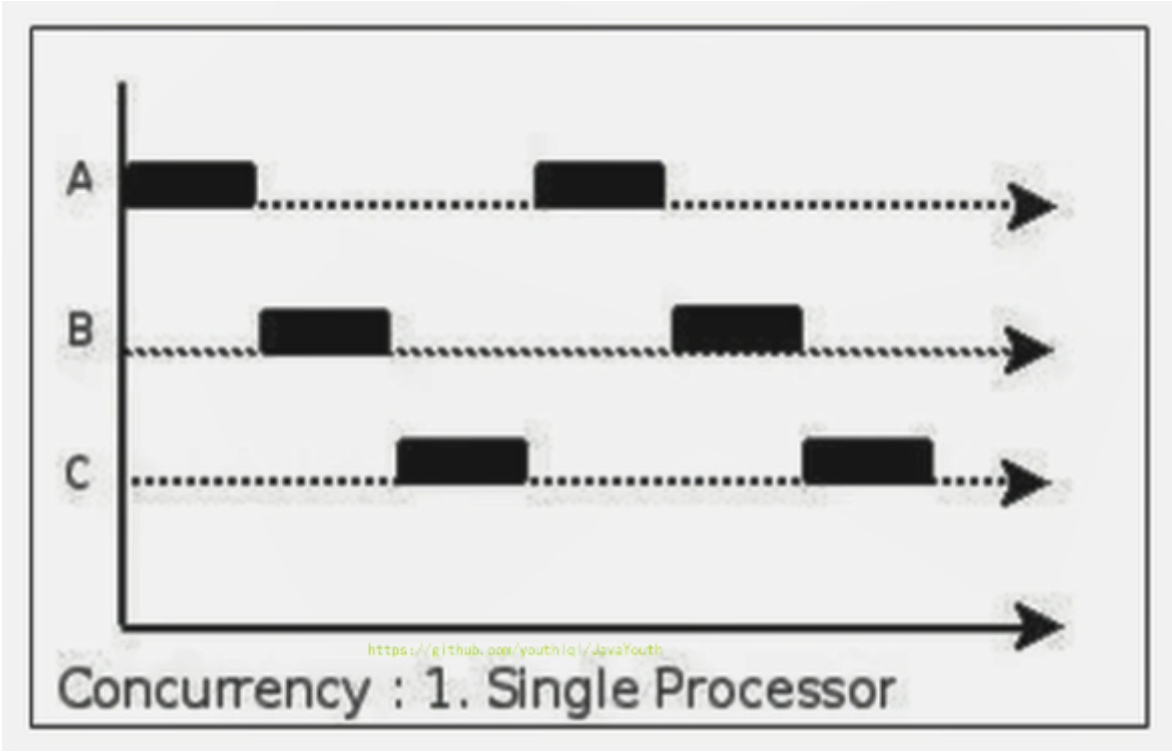
开启工作线程 w，观察打印输出：当前时间间隔与上次时间间隔相差 1.3s，可以明显感受到 Stop the World 的存在

```
0.1
1.4
2.7
3.8
4.12
5.13

Process finished with exit code -1
```

垃圾回收的并行与并发

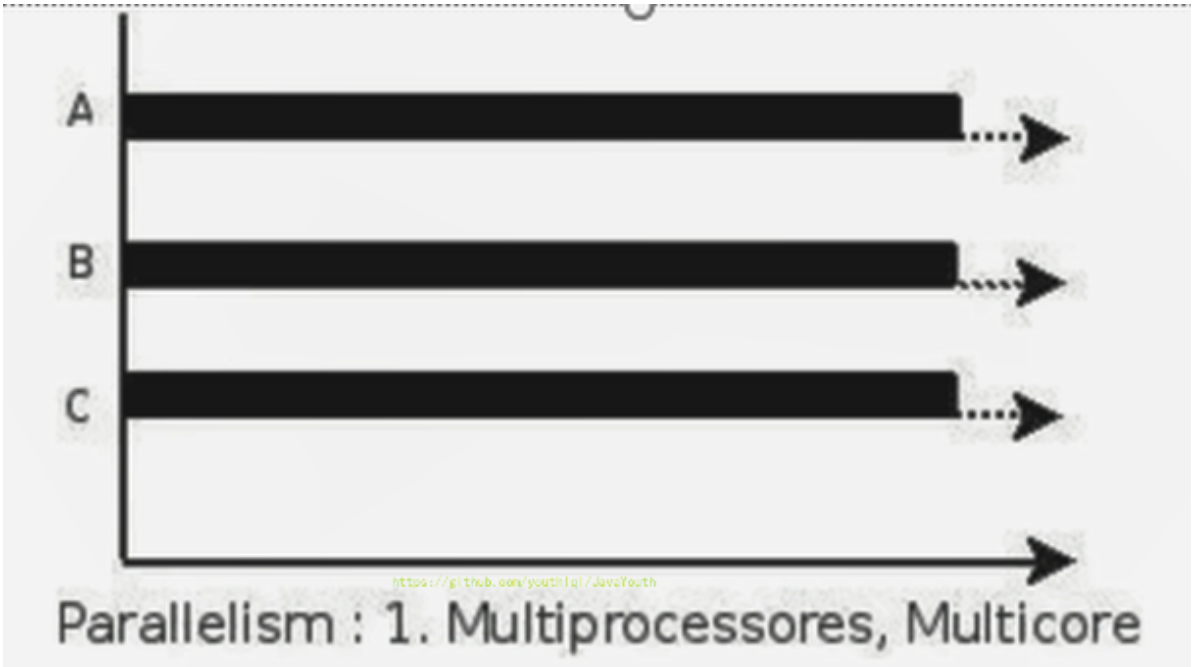
并发的概念

1. 在操作系统中，是指**一个时间段**中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理器上运行
2. 并发不是真正意义上的“同时进行”，只是CPU把一个时间段划分成几个时间片段（时间区间），然后在这几个时间区间之间来回切换。由于CPU处理的速度非常快，只要时间间隔处理得当，即可让用户感觉是多个应用程序同时在进行



并行的概念

1. 当系统有一个以上CPU时，当一个CPU执行一个进程时，另一个CPU可以执行另一个进程，两个进程互不抢占CPU资源，可以**同时**进行，我们称之为并行（Parallel）
2. 其实决定并行的因素不是CPU的数量，而是CPU的核心数量，比如一个CPU多个核也可以并行
3. 适合科学计算，后台处理等弱交互场景



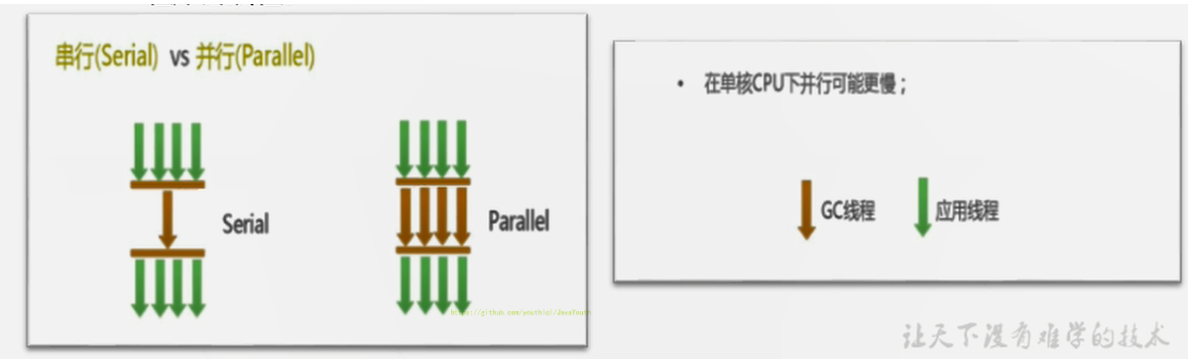
> **并发与并行的对比**

1. 并发，指的是多个事情，在同一时间段内同时发生了。
2. 并行，指的是多个事情，在同一时间点上（或者说同一时刻）同时发生了。
3. 并发的多个任务之间是互相抢占资源的。并行的多个任务之间是不互相抢占资源的。

4. 只有在多CPU或者一个CPU多核的情况中，才会发生并行。否则，看似同时发生的事情，其实都是并发执行的。

垃圾回收的并发与并行

1. 并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍处于等待状态。
    - 如ParNew、Parallel Scavenge、Parallel Old
2. 串行（Serial）
    - 相较于并行的概念，单线程执行。
    - 如果内存不够，则程序暂停，启动JVM垃圾回收器进行垃圾回收（单线程）



并发和并行，在谈论垃圾收集器的上下文语境中，它们可以解释如下：

1. 并发（Concurrent）：指**用户线程与垃圾收集线程同时执行**（但不一定是并行的，可能会交替执行），垃圾回收线程在执行时不会停顿用户程序的运行。
    - 比如用户程序在继续运行，而垃圾收集程序线程运行于另一个CPU上；
2. 典型垃圾回收器：CMS、G1

HotSpot的算法实现细节

根节点枚举

1、固定可作为GC Roots的节点主要在全局性的引用（例如常量或类静态属性）与执行上下文（例如栈帧中的本地变量表）中，尽管目标明确，但查找过程要做到高效并非一件容易的事情，现在Java应用越做越庞大，光是方法区的大小就常有数百上千兆，里面的类、常量等更是恒河沙数，若要逐个检查以这里为起源的引用肯定得消耗不少时间。

2、迄今为止，**所有收集器在根节点枚举这一步骤时都是必须暂停用户线程的**，因此毫无疑问根节点 枚举与之前提及的整理内存碎片一样会面临相似的"Stop The World"的困扰。现在可达性分析算法耗时 最长的查找引用链的过程已经可以做到与用户线程一起并发，**但根节点枚举始终还 是必须在一个能保障一致性的快照中才得以进行**——这里"一致性"的意思是整个枚举期间执行子系统 看起来就像被冻结在某个时间点上，不会出现分析过程中，根节点集合的对象引用关系还在不断变化 的情况，若这点不能满足的话，分析结果准确性也就无法保证。这是导致垃圾收集过程必须停顿所有 用户线程的其中一个重要原因，即使是号称停顿时间可控，或者（几乎）不会发生停顿的CMS、G1、 ZGC等收集器，枚举根节点时也是必须要停顿的。

3、由于目前主流Java虚拟机使用的都是**准确式垃圾收集**，所以当用户线程停顿下来之后，其实并不需要一个不漏地检查完所有 执行上下文和全局的引用位置，虚拟机应当是有办法直接得到哪些地方存放着对象引用的。在HotSpot 的解决方案里，是使用一组称为**OopMap的数据结构**来达到这个目的。一旦类加载动作完成的时候， HotSpot就会把对象内什么偏移量上是什么类型的数据计算出来，在即时编译过程中，也 会在特定的位置记录下栈里和寄存器里哪些位置是引用。这样收集器在扫描时就可以直接得知这些信 息了，**并不需要真正一个不漏地从方法区等GC Roots开始查找。**

4、Exact VM因它使用**准确式内存管理**（Exact Memory Management，也可以叫Non-Conservative/Accurate Memory Management）而得名。准确式内存管理是指虚拟机可以知道内存中某个位 置的数据具体是什么类型。譬如内存中有一个32bit的整数123456，虚拟机将有能力分辨出它到底是一 个指向了123456的内存地址的引用类型还是一个数值为123456的整数，准确分辨出哪些内存是引用类 型，这也是在垃圾收集时准确判断堆上的数据是否还可能被使用的前提。**【这个不是特别重要，了解一下即可】**

> 常考面试：**在OopMap的协助下，HotSpot可以快速准确地完成GC Roots枚举**

安全点与安全区域

安全点（Safepoint）

1. 程序执行时并非在所有地方都能停顿下来开始GC，只有在特定的位置才能停顿下来开始GC，这些位置称为"安全点（Safepoint）"。
2. Safe Point的选择很重要，**如果太少可能导致GC等待的时间太长，如果太频繁可能导致运行时的性能问题。**大部分指令的执行时间都非常短暂，通常会根据"**是否具有让程序长时间执行的特征**"为标准。比如：选择一些执行时间较长的指令作为Safe Point，**如方法调用、循环跳转和异常跳转等。**

如何在GC发生时，检查所有线程都跑到最近的安全点停顿下来呢?

1. 抢先式中断：（目前没有虚拟机采用了）首先中断所有线程。如果还有线程不在安全点，就恢复线程，让线程跑到安全点。
2. 主动式中断：设置一个中断标志，各个线程运行到Safe Point的时候**主动轮询**这个标志，如果中断标志为真，则将自己进行中断挂起。

安全区域（Safe Region）

1. Safepoint 机制保证了程序执行时，在不太长的时间内就会遇到可进入GC的Safepoint。但是，程序“不执行”的时候呢?
2. 例如线程处于Sleep状态或Blocked 状态，这时候线程无法响应JVM的中断请求，“走”到安全点去中断挂起，JVM也不太可能等待线程被唤醒。对于这种情况，就需要安全区域（Safe Region）来解决。
3. **安全区域是指在一段代码片段中，对象的引用关系不会发生变化，在这个区域中的任何位置开始GC都是安全的。**我们也可以把Safe Region看做是被扩展了的Safepoint。

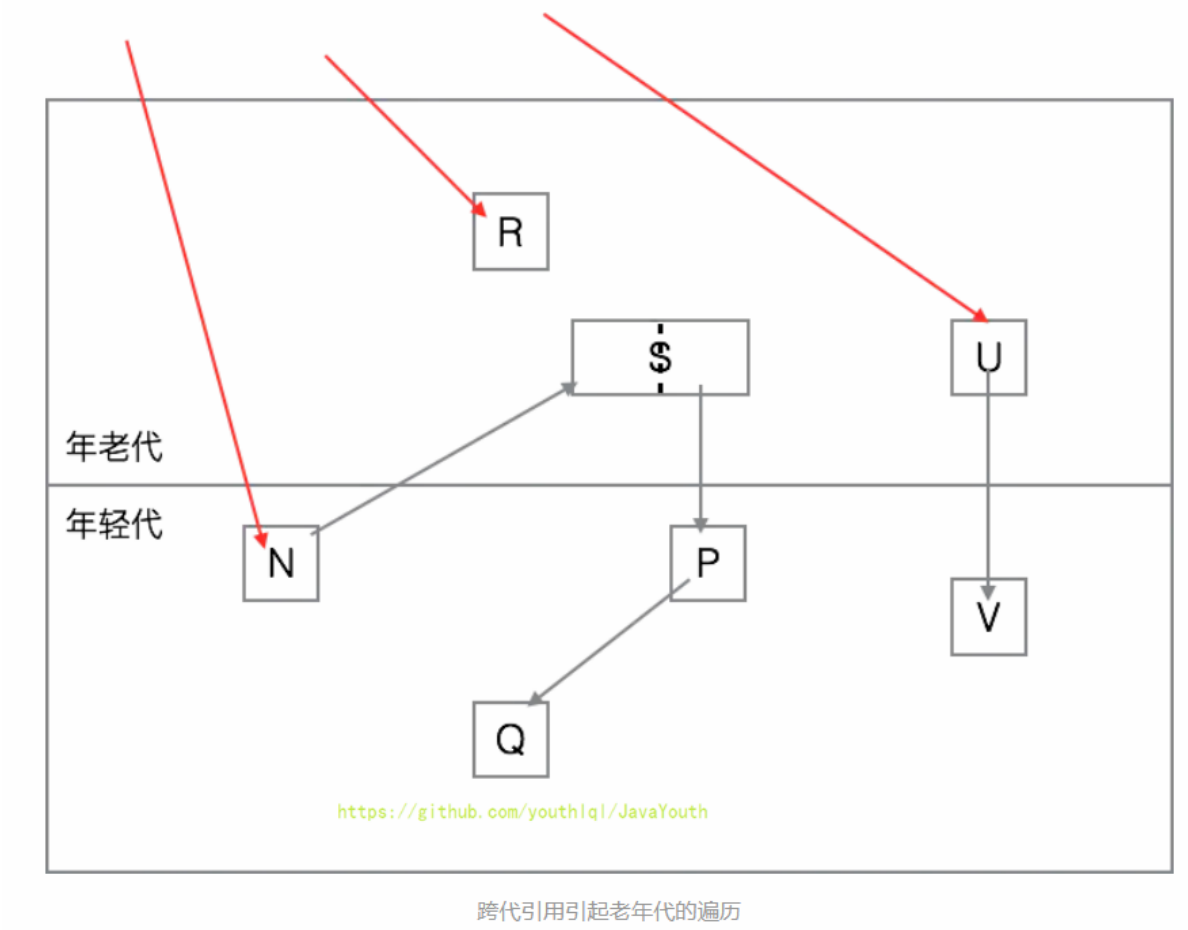
安全区域的执行流程

1. 当线程运行到Safe Region的代码时，首先标识已经进入了Safe Region，如果这段时间内发生GC，JVM会忽略标识为Safe Region状态的线程
2. 当线程即将离开Safe Region时，会检查JVM是否已经完成根节点枚举（即GC Roots的枚举），如果完成了，则继续运行，否则线程必须等待直到收到可以安全离开Safe Region的信号为止；

记忆集与卡表

什么是跨代引用?

1、一般的垃圾回收算法至少会划分出两个年代，年轻代和老年代。但是单纯的分代理论在垃圾回收的时候存在一个巨大的缺陷：为了找到年轻代中的存活对象，却不得不遍历整个老年代，反过来也是一样的。



跨代引用引起老年代的遍历

2、如果我们从年轻代开始遍历，那么可以断定N, S, P, Q都是存活对象。但是，V却不会被认为是存活对象，其占据的内存会被回收了。这就是一个惊天的大漏洞！因为U本身是老年代对象，而且有外部引用指向它，也就是说U是存活对象，而U指向了V，也就是说V也应该是存活对象才是！而这都是因为我们只遍历年轻代对象！

3、所以，为了解决这种跨代引用的问题，最笨的办法就是遍历老年代的对象，找出这些跨代引用来。这种方案存在极大的性能浪费。因为从两个分代假说里面，其实隐含了一个推论：跨代引用是极少的。也就是为了找出那么一点点跨代引用，我们却得遍历整个老年代！从上图来说，很显然的是，我们根本不必遍历R。

4、因此，为了避免这种遍历老年代的性能开销，通常的分代垃圾回收器会引入一种称为**记忆集**的技术。**简单来说，记忆集就是用来记录跨代引用的表。**

记忆集与卡表

1、为解决对象跨代引用所带来的问题，垃圾收集器在新生代中建 立了名为**记忆集（Remembered Set）的数据结构**，用以避免把整个老年代加进GC Roots扫描范围。事实上并不只是新生代、老年代之间才有跨代引用的问题，所有涉及部分区域收集（Partial GC）行为的 垃圾收集器，典型的如G1、ZGC和Shenandoah收集器，都会面临相同的问题，因此我们有必要进一步 理清记忆集的原理和实现方式，以便在后续章节里介绍几款最新的收集器相关知识时能更好地理解。

2、记忆集是一种用于记录**从非收集区域指向收集区域的指针集合的抽象数据结构**。如果我们不考虑效率和成本的话，最简单的实现可以用非收集区域中所有含跨代引用的对象数组来实现这个数据结构。

> 比如说我们有老年代（非收集区域）和年轻代（收集区域）的对象之间有一条引用链

3、这种记录全部含跨代引用对象的实现方案，无论是空间占用还是维护成本都相当高昂。而在垃圾 收集的场景中，收集器只需要通过记忆集判断出某一块非收集区域是否存在有指向了收集区域的指针 就可以了，并不需要了解这些跨代指针的全部细节。那设计者在实现记忆集的时候，便可以选择更为 粗犷的记录粒度来节省记忆集的存储和维护成本，下面列举了一些可供选择（当然也可以选择这个范 围以外的）的记录精度：

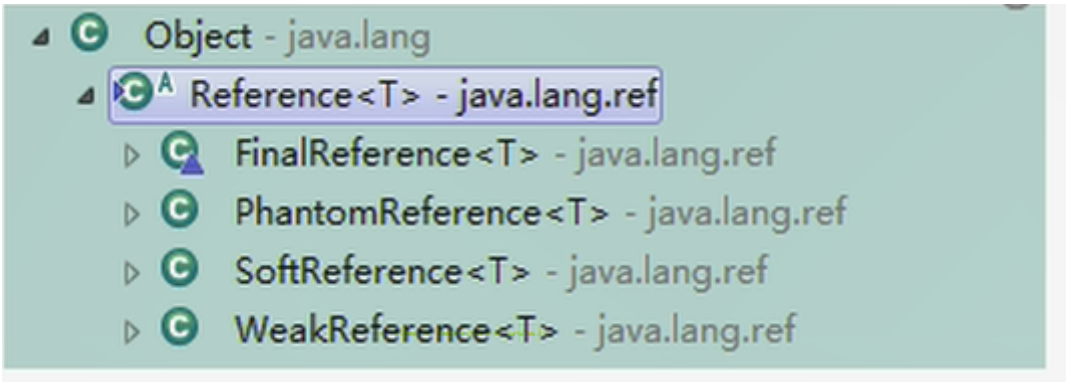
- 字长精度：每个记录精确到一个机器字长（就是处理器的寻址位数，如常见的32位或64位，这个精度决定了机器访问物理内存地址的指针长度），该字包含跨代指针。
- 对象精度：每个记录精确到一个对象，该对象里有字段含有跨代指针。
- 卡精度：每个记录精确到一块内存区域，该区域内有对象含有跨代指针。

4、其中，第三种“卡精度”所指的是用一种称为“卡表”（Card Table）的方式去实现记忆集，这也是 目前最常用的一种记忆集实现形式，一些资料中甚至直接把它和记忆集混为一谈。前面定义中提到记 忆集其实是一种“抽象”的数据结构，抽象的意思是只定义了记忆集的行为意图，并没有定义其行为的 具体实现。卡表就是记忆集的一种具体实现，它定义了记忆集的记录精度、与堆内存的映射关系等。 关于卡表与记忆集的关系，读者不妨按照Java语言中HashMap与Map的关系来类比理解。 卡表最简单的形式可以只是一个字节数组，而HotSpot虚拟机确实也是这样做的

> 读者只需要知道有这个东西，面试的时候能说出来，再细致一点的就需要看周志明老师的第三版书了

再谈引用概述

1. 我们希望能描述这样一类对象：当内存空间还足够时，则能保留在内存中；如果内存空间在进行垃圾收集后还是很紧张，则可以抛弃这些对象。
2. 既偏门又非常高频的面试题：强引用、软引用、弱引用、虚引用有什么区别？具体使用场景是什么？
3. 在JDK1.2版之后，Java对引用的概念进行了扩充，将引用分为：
    - 强引用（Strong Reference）
    - 软引用（Soft Reference）
    - 弱引用（Weak Reference）
    - 虚引用（Phantom Reference）
4. 这4种引用强度依次逐渐减弱。除强引用外，其他3种引用均可以在java.lang.ref包中找到它们的身影。如下图，显示了这3种引用类型对应的类，开发人员可以在应用程序中直接使用它们。



Reference子类中只有终结器引用是包内可见的，其他3种引用类型均为public，可以在应用程序中直接使用

1. 强引用（StrongReference）：最传统的“引用”的定义，是指在程序代码之中普遍存在的引用赋值，即类似“`object obj=new Object()`”这种引用关系。无论任何情况下，只要强引用关系还存在，垃圾收集器就永远不会回收掉被引用的对象。宁可报OOM，也不会GC强引用
2. 软引用（SoftReference）：在系统将要发生内存溢出之前，将会把这些对象列入回收范围之中进行第二次回收。如果这次回收后还没有足够的内存，才会抛出内存溢出异常。
3. 弱引用（WeakReference）：被弱引用关联的对象只能生存到下一次垃圾收集之前。当垃圾收集器工作时，无论内存空间是否足够，都会回收掉被弱引用关联的对象。
4. 虚引用（PhantomReference）：一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来获得一个对象的实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

再谈引用：强引用

1. 在Java程序中，最常见的引用类型是强引用（普通系统99%以上都是强引用），也就是我们最常见的普通对象引用，**也是默认的引用类型。**
2. 当在Java语言中使用new操作符创建一个新的对象，并将其赋值给一个变量的时候，这个变量就成为指向该对象的一个强引用。
3. **只要强引用的对象是可触及的，垃圾收集器就永远不会回收掉被引用的对象。**只要强引用的对象是可达的，jvm宁可报OOM，也不会回收强引用。
4. 对于一个普通的对象，如果没有其他的引用关系，只要超过了引用的作用域或者显式地将相应（强）引用赋值为null，就是可以当做垃圾被收集了，当然具体回收时机还是要看垃圾收集策略。
5. 相对的，软引用、弱引用和虚引用的对象是软可触及、弱可触及和虚可触及的，在一定条件下，都是可以被回收的。所以，强引用是造成Java内存泄漏的主要原因之一。

强引用代码举例

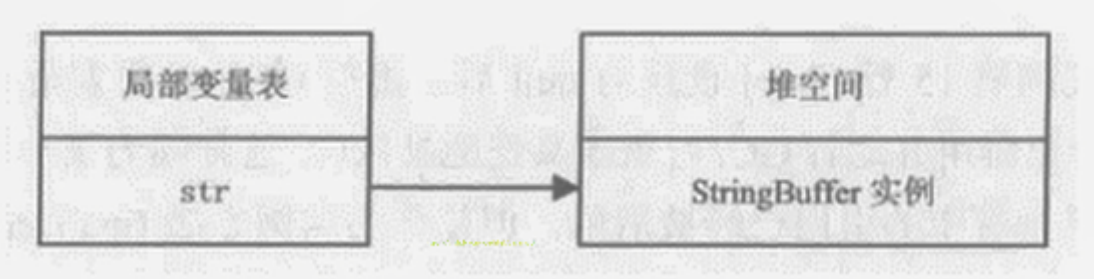
```
public class StrongReferenceTest {
    public static void main(String[] args) {
        StringBuffer str = new StringBuffer ("Hello,尚硅谷");
        StringBuffer str1 = str;

        str = null;
        System.gc();

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(str1);
    }
}
```

输出

```
Hello,尚硅谷
```

局部变量str指向stringBuffer实例所在堆空间，通过str可以操作该实例，那么str就是stringBuffer实例的强引用对应内存结构：

`StringBuffer str = new StringBuffer("hello,尚硅谷");`



总结

本例中的两个引用，都是强引用，强引用具备以下特点：

1. 强引用可以直接访问目标对象。
2. 强引用所指向的对象在任何时候都不会被系统回收，虚拟机宁愿抛出OOM异常，也不会回收强引用所指向对象。
3. 强引用可能导致内存泄漏。

再谈引用：软引用

软引用（Soft Reference）：内存不足即回收

1. 软引用是用来描述一些还有用，但非必需的对象。只被软引用关联着的对象，在系统将要发生内存溢出异常前，会把这些对象列进回收范围之中进行第二次回收，如果这次回收还没有足够的内存，才会抛出内存溢出异常。注意，这里的第一次回收是不可达的对象
2. 软引用通常用来实现内存敏感的缓存。比如：高速缓存就有用到软引用。如果还有空闲内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。
3. 垃圾回收器在某个时刻决定回收软可达的对象的时候，会清理软引用，并可选地把引用存放到一个引用队列（Reference Queue）。
4. 类似弱引用，只不过Java虚拟机会尽量让软引用的存活时间长一些，迫不得已才清理。
5. 一句话概括：当内存足够时，不会回收软引用可达的对象。内存不够时，会回收软引用的可达对象

在JDK1.2版之后提供了SoftReference类来实现软引用

```
Object obj = new Object();// 声明强引用
SoftReference<Object> sf = new SoftReference<>(obj);
obj = null; //销毁强引用
```

软引用代码举例

代码

```
public class SoftReferenceTest {
    public static class User {
        public User(int id, String name) {
            this.id = id;
            this.name = name;
        }

        public int id;
        public String name;

        @Override
        public String toString() {
            return "[id=" + id + ", name=" + name + "] ";
        }
    }

    public static void main(String[] args) {
        //创建对象，建立软引用
//        SoftReference<User> userSoftRef = new SoftReference<User>(new User(1,
"songhk"));
        //上面的一行代码，等价于如下的三行代码
        User u1 = new User(1,"songhk");
        SoftReference<User> userSoftRef = new SoftReference<User>(u1);
        u1 = null;//取消强引用


        //从软引用中重新获得强引用对象
        System.out.println(userSoftRef.get());

        System.out.println("---目前内存还不紧张---");
        System.gc();
```

```
        System.out.println("After GC:");
//        //垃圾回收之后获得软引用中的对象
        System.out.println(userSoftRef.get());//由于堆空间内存足够，所有不会回收软引用的可达对象。
        System.out.println("---下面开始内存紧张了---");
        try {
            //让系统认为内存资源紧张、不够
//            byte[] b = new byte[1024 * 1024 * 7];
            byte[] b = new byte[1024 * 7168 - 635 * 1024];
        } catch (Throwable e) {
            e.printStackTrace();
        } finally {
            //再次从软引用中获取数据
            System.out.println(userSoftRef.get());//在报OOM之前，垃圾回收器会回收软引用的可达对象。
        }
    }
}
```

JVM参数

`-Xms10m -Xmx10m`

在 JVM 内存不足时，会清理软引用对象

输出结果：

```
[id=1, name=songhk]
---目前内存还不紧张---
After GC:
[id=1, name=songhk]
---下面开始内存紧张了---
null
java.lang.OutOfMemoryError: Java heap space
    at com.atguigu.java1.SoftReferenceTest.main(SoftReferenceTest.java:48)

Process finished with exit code 0
```

再谈引用：弱引用

> **弱引用（Weak Reference）发现即回收**

1. 弱引用也是用来描述那些非必需对象，**只被弱引用关联的对象只能生存到下一次垃圾收集发生为止。在系统GC时，只要发现弱引用，不管系统堆空间使用是否充足，都会回收掉只被弱引用关联的对象。**
2. 但是，由于垃圾回收器的线程通常优先级很低，因此，并不一定能很快地发现持有弱引用的对象。在这种情况下，弱引用对象可以存在较长的时间。
3. 弱引用和软引用一样，在构造弱引用时，也可以指定一个引用队列，当弱引用对象被回收时，就会加入指定的引用队列，通过这个队列可以跟踪对象的回收情况。

4. 软引用、弱引用都非常适合来保存那些可有可无的缓存数据。如果这么做，当系统内存不足时，这些缓存数据会被回收，不会导致内存溢出。而当内存资源充足时，这些缓存数据又可以存在相当长的时间，从而起到加速系统的作用。

在JDK1.2版之后提供了WeakReference类来实现弱引用

```
// 声明强引用
Object obj = new Object();
WeakReference<Object> sf = new WeakReference<>(obj);
obj = null; //销毁强引用
```

弱引用对象与软引用对象的最大不同就在于，当GC在进行回收时，需要通过算法检查是否回收软引用对象，而对于弱引用对象，GC总是进行回收。弱引用对象更容易、更快被GC回收。

面试题：你开发中使用过WeakHashMap吗?

弱引用代码举例

```
public class WeakReferenceTest {
    public static class User {
        public User(int id, String name) {
            this.id = id;
            this.name = name;
        }

        public int id;
        public String name;

        @Override
        public String toString() {
            return "[id=" + id + ", name=" + name + "] ";
        }
    }

    public static void main(String[] args) {
        //构造了弱引用
        WeakReference<User> userWeakRef = new WeakReference<User>(new User(1,
"songhk"));
        //从弱引用中重新获取对象
        System.out.println(userWeakRef.get());

        System.gc();
        // 不管当前内存空间足够与否，都会回收它的内存
        System.out.println("After GC:");
        //重新尝试从弱引用中获取对象
        System.out.println(userWeakRef.get());
    }
}
```

执行垃圾回收后，软引用对象必定被清除

```
[id=1, name=songhk]
After GC:
null

Process finished with exit code 0
```

再谈引用：虚引用

虚引用（Phantom Reference）：对象回收跟踪

1. 也称为“幽灵引用”或者“幻影引用”，是所有引用类型中最弱的一个
2. 一个对象是否有虚引用的存在，完全不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它和没有引用几乎是一样的，随时都可能被垃圾回收器回收。
3. 它不能单独使用，也无法通过虚引用来获取被引用的对象。当试图通过虚引用的get()方法取得对象时，总是null 。**即通过虚引用无法获取到我们的数据**
4. **为一个对象设置虚引用关联的唯一目的在于跟踪垃圾回收过程。比如：能在这个对象被收集器回收时收到一个系统通知。**
5. 虚引用必须和引用队列一起使用。虚引用在创建时必须提供一个引用队列作为参数。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象后，将这个虚引用加入引用队列，以通知应用程序对象的回收情况。
6. 由于虚引用可以跟踪对象的回收时间，因此，也可以将一些资源释放操作放置在虚引用中执行和记录。

在JDK1.2版之后提供了PhantomReference类来实现虚引用。

```
// 声明强引用
Object obj = new Object();
// 声明引用队列
ReferenceQueue phantomQueue = new ReferenceQueue();
// 声明虚引用（还需要传入引用队列）
PhantomReference<Object> sf = new PhantomReference<>(obj, phantomQueue);
obj = null;
```

虚引用代码示例

```
public class PhantomReferenceTest {
    public static PhantomReferenceTest obj;//当前类对象的声明
    static ReferenceQueue<PhantomReferenceTest> phantomQueue = null;//引用队列

    public static class CheckRefQueue extends Thread {
        @Override
        public void run() {
            while (true) {
                if (phantomQueue != null) {
```

```
                    PhantomReference<PhantomReferenceTest> objt = null;
                    try {
                        objt = (PhantomReference<PhantomReferenceTest>)
phantomQueue.remove();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    if (objt != null) {
                        System.out.println("追踪垃圾回收过程：PhantomReferenceTest实
例被GC了");
                    }
                }
            }
        }
    }

    @Override
    protected void finalize() throws Throwable { //finalize()方法只能被调用一次!
        super.finalize();
        System.out.println("调用当前类的finalize()方法");
        obj = this;
    }

    public static void main(String[] args) {
        Thread t = new CheckRefQueue();
        t.setDaemon(true);//设置为守护线程：当程序中没有非守护线程时，守护线程也就执行结
束。
        t.start();

        phantomQueue = new ReferenceQueue<PhantomReferenceTest>();
        obj = new PhantomReferenceTest();
        //构造了 PhantomReferenceTest 对象的虚引用，并指定了引用队列
        PhantomReference<PhantomReferenceTest> phantomRef = new
PhantomReference<PhantomReferenceTest>(obj, phantomQueue);

        try {
            //不可获取虚引用中的对象
            System.out.println(phantomRef.get());
            System.out.println("第 1 次 gc");
            //将强引用去除
            obj = null;
            //第一次进行GC,由于对象可复活，GC无法回收该对象
            System.gc();
            Thread.sleep(1000);
            if (obj == null) {
                System.out.println("obj 是 null");
            } else {
                System.out.println("obj 可用");
            }
            System.out.println("第 2 次 gc");
            obj = null;
            System.gc(); //一旦将obj对象回收，就会将此虚引用存放到引用队列中。
            Thread.sleep(1000);
            if (obj == null) {
                System.out.println("obj 是 null");
```

```
            } else {
                System.out.println("obj 可用");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

1、第一次尝试获取虚引用的值，发现无法获取的，这是因为虚引用是无法直接获取对象的值，然后进行第一次GC，因为会调用finalize方法，将对象复活了，所以对象没有被回收

2、但是调用第二次GC操作的时候，因为finalize方法只能执行一次，所以就触发了GC操作，将对象回收了，同时将会触发第二个操作就是将待回收的对象存入到引用队列中。

输出结果：

```
null
第 1 次 gc
调用当前类的finalize()方法
obj 可用
第 2 次 gc
追踪垃圾回收过程：PhantomReferenceTest实例被GC了
obj 是 null

Process finished with exit code 0
```

再谈引用：终结器引用（了解）

1. 它用于实现对象的finalize() 方法，也可以称为终结器引用
2. 无需手动编码，其内部配合引用队列使用
3. 在GC时，终结器引用入队。由Finalizer线程通过终结器引用找到被引用对象调用它的finalize()方法，第二次GC时才回收被引用的对象