通过 Java 串行收集器（SerialGC）深入理解分代 GC 过程

作者：Lord-X
原文链接：https://ld246.com/article/1564807710608
来源网站：链滴

□rose□rose 如果您觉得我的文章对您有帮助的话，记得在GitHub上star一波哈□ose□rose

□rose□rose GitHub_awesome-it-blog □rose□rose

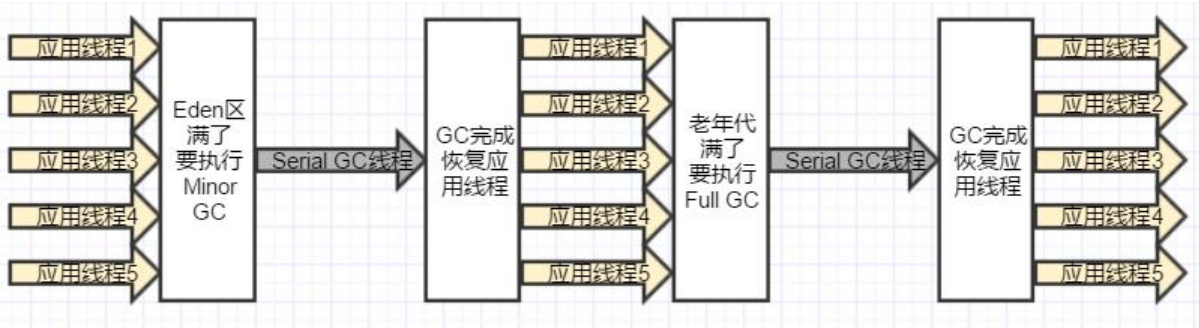
Java GC发展至今，已经推出了好几代收集器，包括Serial、ParNew、Parallel、CMS、G1以及Java1中最新的ZGC。每一代GC都对前一代存在的问题做出了很大的改善。

今天介绍一个古董收集器-Serial串行GC。

虽然此收集器的使用场景已经不多，但本文通过这个收集器，说明了如何分配每一块堆内存的大小，根据GC日志，详细说明了Serial GC在新生代和老年代的GC过程。

Serial GC的名字能很好地概括他的特点：串行。它与应用线程的执行是串行的，也就是说，执行应用程的时候，不会执行GC，执行GC的时候，不能执行应用线程。

所以，整个Java进程执行起来就行下面的样子：



Serial GC使用的是分代算法，在新生代上，Serial使用复制算法进行收集，在老年代上，Serial使用记-压缩算法进行收集。

分代算法、复制算法、标记-压缩请移步：

Java虚拟机-GC垃圾回收算法-标记清除法、复制算法、标记压缩法、分代算法

1 Serial存在的问题

如上图所示，在需要执行GC时，GC线程会阻塞所有用户线程（Stop-The-world，简称STW），等执行完，才会恢复用户线程。

这对我们的应用程序来说，每次GC是都会造成不同程度的卡顿，对用户是极为不友好的。

2 使用场景

个人观点：

首先，根据其特点，回收算法简单，所以回收效率高。

其次，它是单线程收集的，不存在GC线程之间的切换。由于Java的线程切换是需要系统内核来调度，在单线程下，可以很大程度的减少调度带来的系统开销。

所以，也许在单核CPU机器上，且业务场景为只对公司内部使用且可以忍受STW带来的卡顿的情况下有一些用武之地。

3 实战

环境：

- CPU：i7 4核
- 内存：16G
- JDK version：8

3.1 先来看一下默认情况下，使用的哪个GC

添加下面JVM参数并运行代码，观察GC日志

```
/**
 * JVM参数:
 * -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
 */
public static void main(String[] args) {
    System.out.println("Hello SerialGC");
}
```

程序输出如下

```
Hello SerialGC
// 下面是GC日志
Heap
PSYoungGen      total 76288K, used 6554K [0x000000076b180000, 0x0000000770680000, 0x0
000007c0000000)
  eden space 65536K, 10% used [0x000000076b180000,0x000000076b7e6930,0x000000076f1
```

```
0000)
  from space 10752K, 0% used [0x000000076fc00000,0x000000076fc00000,0x0000000770680
00)
  to   space 10752K, 0% used [0x000000076f180000,0x000000076f180000,0x000000076fc0000
)
ParOldGen       total 175104K, used 0K [0x00000006c1400000, 0x00000006cbf00000, 0x00000
076b180000)
  object space 175104K, 0% used [0x00000006c1400000,0x00000006c1400000,0x00000006cbf
0000)
Metaspace       used 3458K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 381K, capacity 388K, committed 512K, reserved 1048576K
```

- PSYoungGen：表示年轻代使用的是ParallelGC
- ParOldGen：表示老年代使用的是ParallelGC
- Metaspace：元数据区使用情况

可见，在多核情况下，JVM默认选用了支持多线程并发的ParallelGC。

3.2 Serial GC是运行在Client模式下的默认收集器?

周志明老师的书中提到过，SerialGC仍然是-client模式下默认的收集器。

下面来实验一下，刚才的JVM启动参数加上-client参数

```
/**
* JVM参数:
* -client -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
*/
public static void main(String[] args) {
    System.out.println("Hello SerialGC");
}
```

运行结果如下：

```
Hello SerialGC in client mode
Heap
PSYoungGen      total 76288K, used 6554K [0x000000076b180000, 0x0000000770680000, 0x0
000007c0000000)
  eden space 65536K, 10% used [0x000000076b180000,0x000000076b7e6930,0x000000076f1
0000)
  from space 10752K, 0% used [0x000000076fc00000,0x000000076fc00000,0x0000000770680
00)
  to   space 10752K, 0% used [0x000000076f180000,0x000000076f180000,0x000000076fc0000
)
ParOldGen       total 175104K, used 0K [0x00000006c1400000, 0x00000006cbf00000, 0x00000
076b180000)
  object space 175104K, 0% used [0x00000006c1400000,0x00000006c1400000,0x00000006cbf
0000)
Metaspace       used 3513K, capacity 4498K, committed 4864K, reserved 1056768K
  class space    used 387K, capacity 390K, committed 512K, reserved 1048576K
```

可见依然是ParallelGC。

这个原因应该是由于，在JDK1.8下，-client和-server参数默认都是失效的，所以指定-client也无济事。

其实笔者也在相同的环境下尝试了JDK6和JDK7，也同样不是SerialGC，所以猜想可能是老版本的单CPU情况下，JVM会默认选择SerialGC，但这一点笔者尚未查证。

PS：-client和-server

-client和-server参数在之前版本的JDK中是用来选择JVM运行过程中使用的编译器的。对启动性能有求的程序，可使用-client，对应的编译器为编译效率较快C1，对峰值性能有要求的程序，可使用-serv r，对应生成代码执行效率较快的C2（参考了郑雨迪老师在极客时间推出的课程）。

Java8会默认使用分层编译的机制，会自动选择在何时使用哪个编译器，所以client和server参数在默情况下失效。相对之前的JDK版本，JDK8的这种机制很大程度地提升了代码的编译执行效率。

3.3 Serial GC实战 - JVM参数

本小节说明了如何配置堆内存中每一块内存的大小。

首先我们要明确需要指定哪几块内存。因为Serial GC是分代收集，所以要确认新生代和老年代的大小其中，新生代又需要确认Eden区和Survivor区的大小。

- 定义整个堆内存的大小

```
// -Xmx：最大堆内存，-Xms：最小堆内存，这里设置为一样的，表示堆内存固定200M
-Xmx200M -Xms200M
```

- 定义新生代和老年代的大小

```
// NewRatio表示老年代和新生代的比例，3表示3：1
// 即把整个堆内存分为4份，老年代占3份，新生代1份
// 目前堆内存为200M，NewRatio=3时，新生代=50M，老年代=150M
-XX:NewRatio=3
```

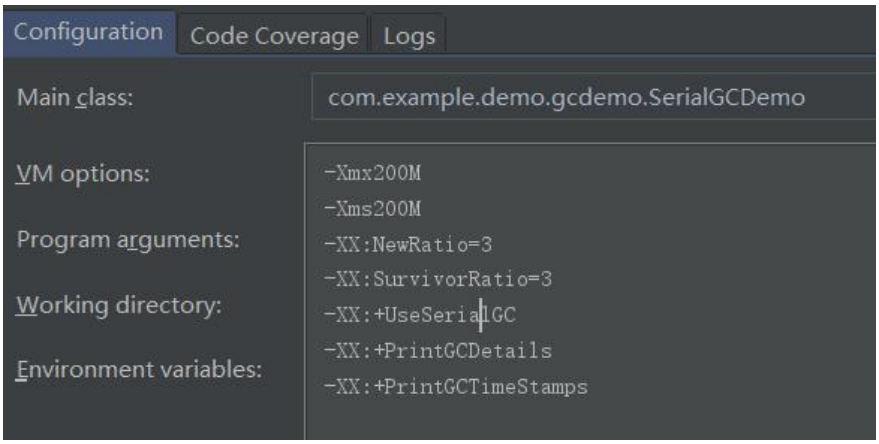
- 定义Eden区和Survivor区的大小

```
// SurvivorRatio表示Eden区和两个Survivor区的比例，3表示3：2（注意是两个Survivor区）
// 即把新生代分为5份，Eden占3份，Survivor区占2份
// 目前新生代为50M，Survivor=3时，Eden=30M，Survivor=20M（from=10M, to=10M）
-XX:SurvivorRatio=3
```

- 配置GC日志打印参数

```
// -XX:+UseSerialGC：显示指定使用Serial GC
// -XX:+PrintGCDetails：打印GC详细日志
// -XX:+PrintGCTimeStamps：打印GC发生的时间
-XX:+UseSerialGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
```

- 实践



依然用上面的Hello SerialGC程序，运行结果如下

```
Hello SerialGC
Heap
// def new generation表明新生代使用SerialGC，total:40M，已使用：4302K
// total少了10M？这是因为新生代使用复制算法，From区和to区实际上每次只能使用1个，所以是ed
n的30M + from或to的10M = 40M
def new generation   total 40960K, used 4302K [0x00000000f3800000, 0x00000000f6a00000,
x00000000f6a00000)
  // eden区30M
  eden space 30720K,  14% used [0x00000000f3800000, 0x00000000f3c33b78, 0x00000000f56
0000)
  // from区10M
  from space 10240K,   0% used [0x00000000f5600000, 0x00000000f5600000, 0x00000000f60
0000)
  // to区10M
  to   space 10240K,   0% used [0x00000000f6000000, 0x00000000f6000000, 0x00000000f6a0
000)
// 老年代使用 SerialGC ，总大小150M，已使用0K
tenured generation   total 153600K, used 0K [0x00000000f6a00000, 0x0000000100000000, 0x
000000100000000)
   the space 153600K,   0% used [0x00000000f6a00000, 0x00000000f6a00000, 0x00000000f6a
0200, 0x0000000100000000)
// 元数据区大小，暂不关注
Metaspace       used 3450K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 380K, capacity 388K, committed 512K, reserved 1048576K
```

关于复制算法，请移步：

复制算法

3.4 Serial GC实战 - 通过GC日志理解新生代老年代的GC过程

此实验在上述JVM参数配置条件下运行。

下面通过一个实例程序，来观察一下

```
public class SerialGCDemo {

    /**
     * 堆内存：-Xmx200M -Xms200M
```

```
     * 新生代：-XX:NewRatio=3 -XX:SurvivorRatio=3
     * GC参数：-XX:+UseSerialGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
     * 堆空间：200M，新生代：50M，老年代：150M，新生代eden区：30M，新生代from区：10
，新生代to区：10M
     * -Xmx200M -Xms200M -XX:NewRatio=3 -XX:SurvivorRatio=3 -XX:+UseSerialGC -XX:+Prin
GCDetails -XX:+PrintGCTimeStamps
     * @param args
     */
    public static void main(String[] args) {
        byte[][] useMemory = new byte[1000][];
        Random random = new Random();
        for (int i = 0; i < useMemory.length; i++) {
            useMemory[i] = new byte[1024 * 1024 * 10]; // 创建10M的对象
            // 20%的概率将创建出来的对象变为可回收对象
            if (random.nextInt(100) < 20) {
                System.out.println("created byte[] and set to null: " + i);
                useMemory[i] = null;
            } else {
                System.out.println("created byte[]: " + i);
            }
        }
    }
}
```

整体日志输入如下：

```
created byte[]: 0
created byte[]: 1
0.236: [GC (Allocation Failure) 0.236: [DefNew: 24807K->870K(40960K), 0.0132148 secs] 2480
K->21350K(194560K), 0.0132618 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
created byte[]: 2
created byte[] and set to null: 3
0.252: [GC (Allocation Failure) 0.252: [DefNew: 21941K->717K(40960K), 0.0060942 secs] 4242
K->31437K(194560K), 0.0061231 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
created byte[]: 4
created byte[]: 5
0.259: [GC (Allocation Failure) 0.259: [DefNew: 22408K->717K(40960K), 0.0114560 secs] 5312
K->51917K(194560K), 0.0114856 secs] [Times: user=0.00 sys=0.02, real=0.02 secs]
created byte[]: 6
created byte[]: 7
0.285: [GC (Allocation Failure) 0.285: [DefNew: 21788K->717K(40960K), 0.0122524 secs] 7298
K->72397K(194560K), 0.0122868 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
created byte[]: 8
created byte[]: 9
0.299: [GC (Allocation Failure) 0.299: [DefNew: 21790K->717K(40960K), 0.0115042 secs] 9347
K->92877K(194560K), 0.0115397 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
created byte[]: 10
created byte[]: 11
0.312: [GC (Allocation Failure) 0.312: [DefNew: 21791K->717K(40960K), 0.0120174 secs] 1139
2K->113357K(194560K), 0.0120525 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
created byte[]: 12
created byte[]: 13
0.328: [GC (Allocation Failure) 0.328: [DefNew: 21792K->717K(40960K), 0.0162437 secs] 1344
2K->133837K(194560K), 0.0162844 secs] [Times: user=0.00 sys=0.01, real=0.02 secs]
```

```
created byte[]: 14
created byte[]: 15
0.347: [GC (Allocation Failure) 0.347: [DefNew: 21793K->21793K(40960K), 0.0000201 secs]0.3
7: [Tenured: 133120K->143360K(153600K), 0.0103885 secs] 154913K->154316K(194560K), [M
taspace: 3350K->3350K(1056768K)], 0.0104608 secs] [Times: user=0.02 sys=0.00, real=0.01 se
s]
Exception in thread "main" created byte[]: 16
0.361: [Full GC (Allocation Failure) 0.361: [Tenured: 143360K->143360K(153600K), 0.0028089
ecs] 165153K->164556K(194560K), [Metaspace: 3350K->3350K(1056768K)], 0.0028543 secs] [
imes: user=0.00 sys=0.00, real=0.00 secs]
0.364: [Full GC (Allocation Failure) 0.364: [Tenured: 143360K->143360K(153600K), 0.0050038
ecs] 164556K->164538K(194560K), [Metaspace: 3350K->3350K(1056768K)], 0.0050390 secs] [
imes: user=0.00 sys=0.00, real=0.00 secs]
Disconnected from the target VM, address: '127.0.0.1:57881', transport: 'socket'
java.lang.OutOfMemoryError: Java heap space
Heap
    at com.example.demo.gcdemo.SerialGCDemo.main(SerialGCDemo.java:28)
def new generation   total 40960K, used 22281K [0x00000000f3800000, 0x00000000f6a00000,
0x00000000f6a00000)
  eden space 30720K,  72% used [0x00000000f3800000, 0x00000000f4dc27c0, 0x00000000f56
0000)
  from space 10240K,   0% used [0x00000000f6000000, 0x00000000f6000000, 0x00000000f6a
0000)
  to   space 10240K,   0% used [0x00000000f5600000, 0x00000000f5600000, 0x00000000f600
000)
tenured generation   total 153600K, used 143360K [0x00000000f6a00000, 0x00000001000000
0, 0x0000000100000000)
   the space 153600K,  93% used [0x00000000f6a00000, 0x00000000ff6000e0, 0x00000000ff6
0200, 0x0000000100000000)
Metaspace       used 3381K, capacity 4568K, committed 4864K, reserved 1056768K
  class space    used 364K, capacity 392K, committed 512K, reserved 1048576K
```

日志说明：

```
0.236: [GC (Allocation Failure) 0.236: [DefNew: 24807K->870K(40960K), 0.0132148 secs] 2480
K->21350K(194560K), 0.0132618 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
```

- 0.236：GC发生的时间（秒），从程序启动开始计算
- [GC：GC类型，另外还有Full GC，GC不会造成STW，Full GC会。
- (Allocation Failure)：GC原因，申请内存失败
- [DefNew：说明新生代用Serail GC回收，即default new generation之意。
- 24087K -> 870K(40960K)：GC前该区域内存已使用容量 -> GC后该区域内存已使用容量(该区域存总容量)
- 0.0132148 secs：该内存区域GC所占用的时间（秒）
- 24807K->21350K(194560K)：GC前堆内存已使用容量 -> GC后堆内存已使用容量(堆内存总容量190M，这里要减去from或to的10M)
- 0.0132618 secs：本次回收整体占用时间（秒）
- [Times: user=0.02 sys=0.00, real=0.01 secs]：占用时间具体数据。user：用户态消耗的CPU时，sys：内核态消耗的CPU时间，real：从操作开始到操作结束所经历的墙钟时间。

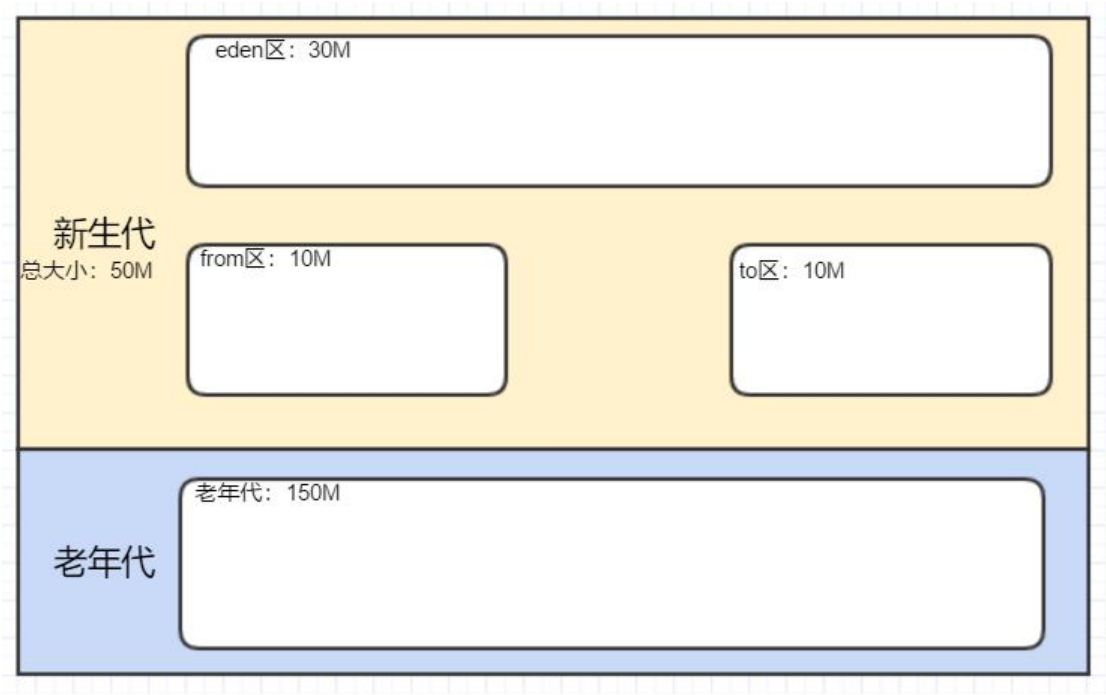
0.361: [Full GC (Allocation Failure) 0.361: [Tenured: 143360K->143360K(153600K), 0.0028089 ecs] 165153K->164556K(194560K), [Metaspace: 3350K->3350K(1056768K)], 0.0028543 secs] [imes: user=0.00 sys=0.00, real=0.00 secs]

这里只说明一下与上面有区别的地方

- [Full GC：GC类型，会造成STW
- [Tenured：老年代回收
- 143360K->143360K(153600K)：老年代GC前已使用内存容量 -> 老年代GC后已使用内存容量(老代总容量)
- 165153K->164556K(194560K)：堆内存GC前已使用内存容量 -> 堆内存GC后已使用内存容量(堆存总容量)
- Metaspace：元数据区内存回收情况

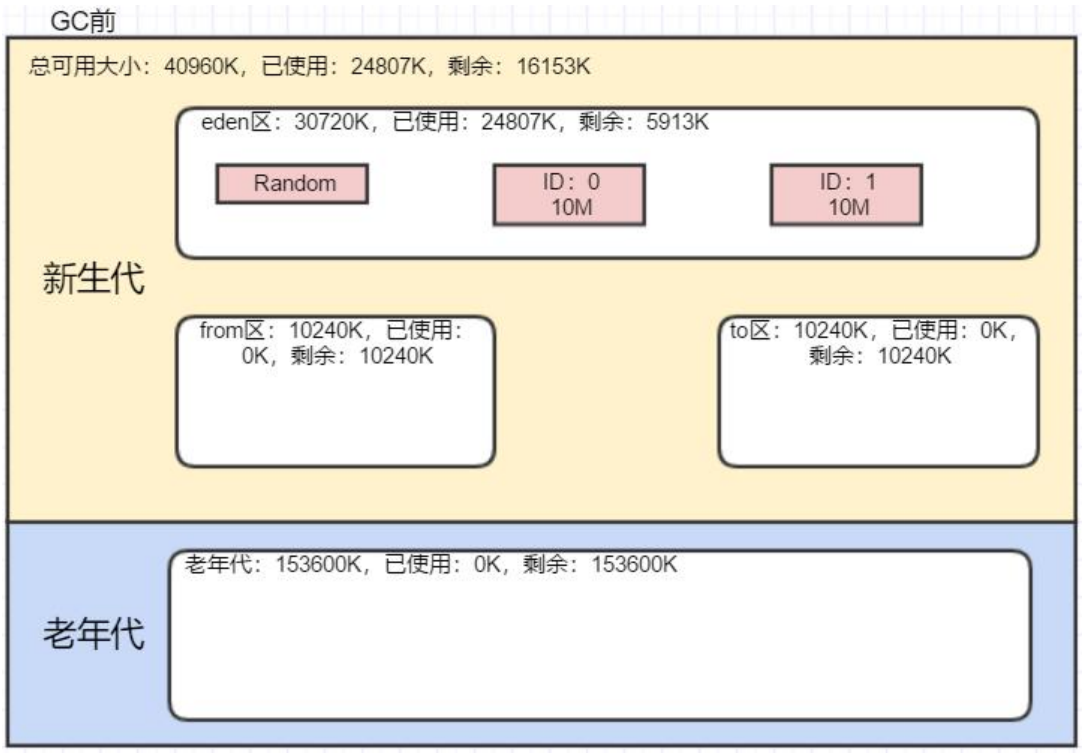
下面分步骤详细看一下从程序开始到结束，对内存的变化过程

整个内存初始状态如下：

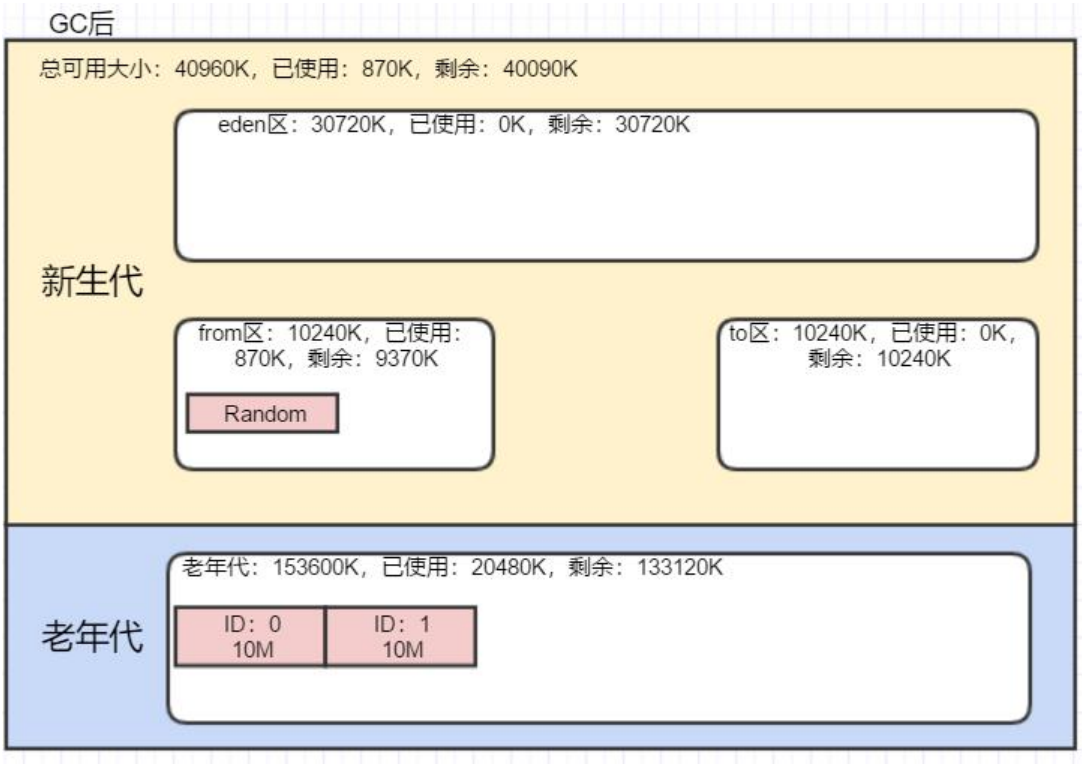


created byte[]: 0
created byte[]: 1
0.236: [GC (Allocation Failure) 0.236: [DefNew: 24807K->870K(40960K), 0.0132148 secs] 2480 K->21350K(194560K), 0.0132618 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]

创建了两个10M的对象（记为ID：0，ID：1），并且没有设置成可回收对象，由于Eden区目前最起还有一个Random对象，所以在给第三个对象申请内存时，发现Eden区内存不足，触发了GC。

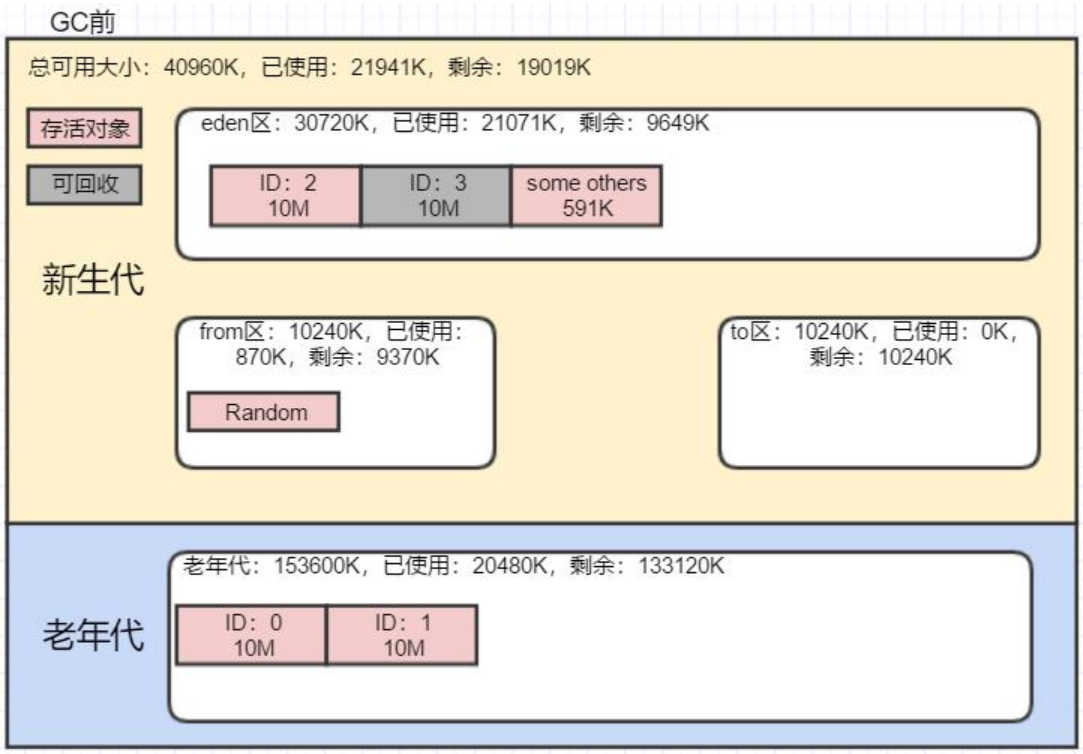


新生代在GC后变为870K，说明Random对象被复制到from区，而两个10M的对象都直接晋升到了老代。

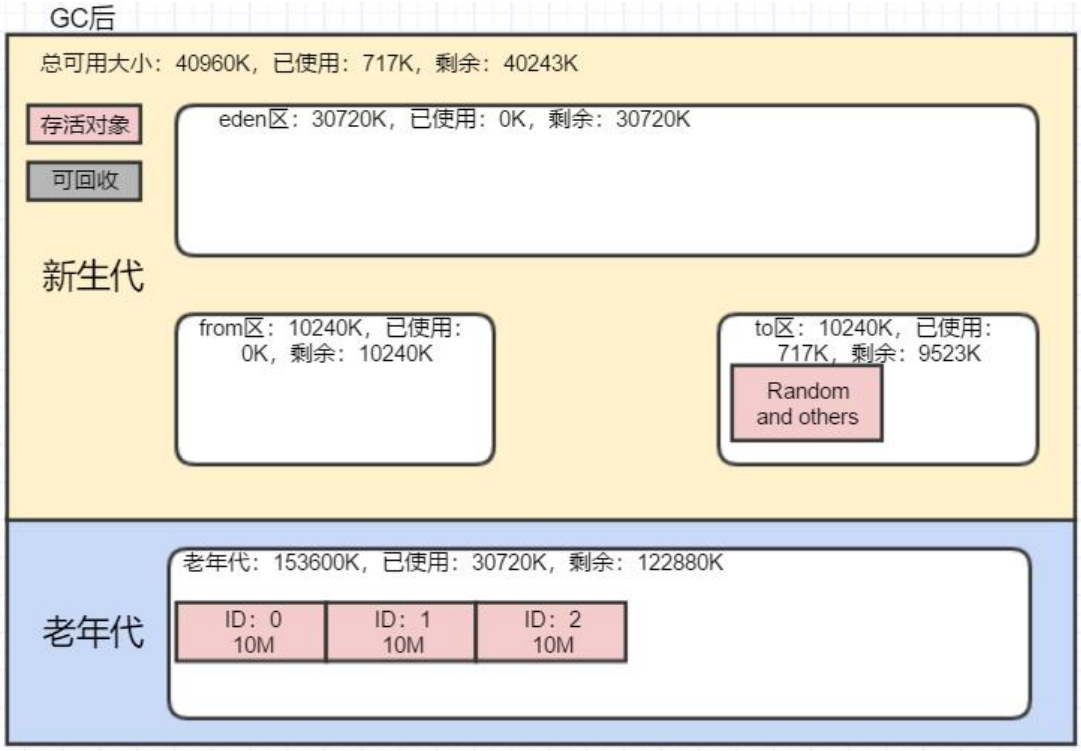



```
created byte[]: 2
created byte[] and set to null: 3
0.252: [GC (Allocation Failure) 0.252: [DefNew: 21941K->717K(40960K), 0.0060942 secs] 4242
K->31437K(194560K), 0.0061231 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

创建了ID：2和ID：3对象，并把ID：3设置为可回收对象



GC会将Eden区的对象和from区的对象尝试复制到to区，ID：3对象直接回收（通过堆空间的容量变可以看出：42421K->31437K），ID：2对象在to区中放不下，晋升老年代



一直到创建ID：12，ID：13，都与上述过程类似，并且没有产生过垃圾对象，但创建完ID：13对象，老年代的已使用内存达到了130M+，如下：

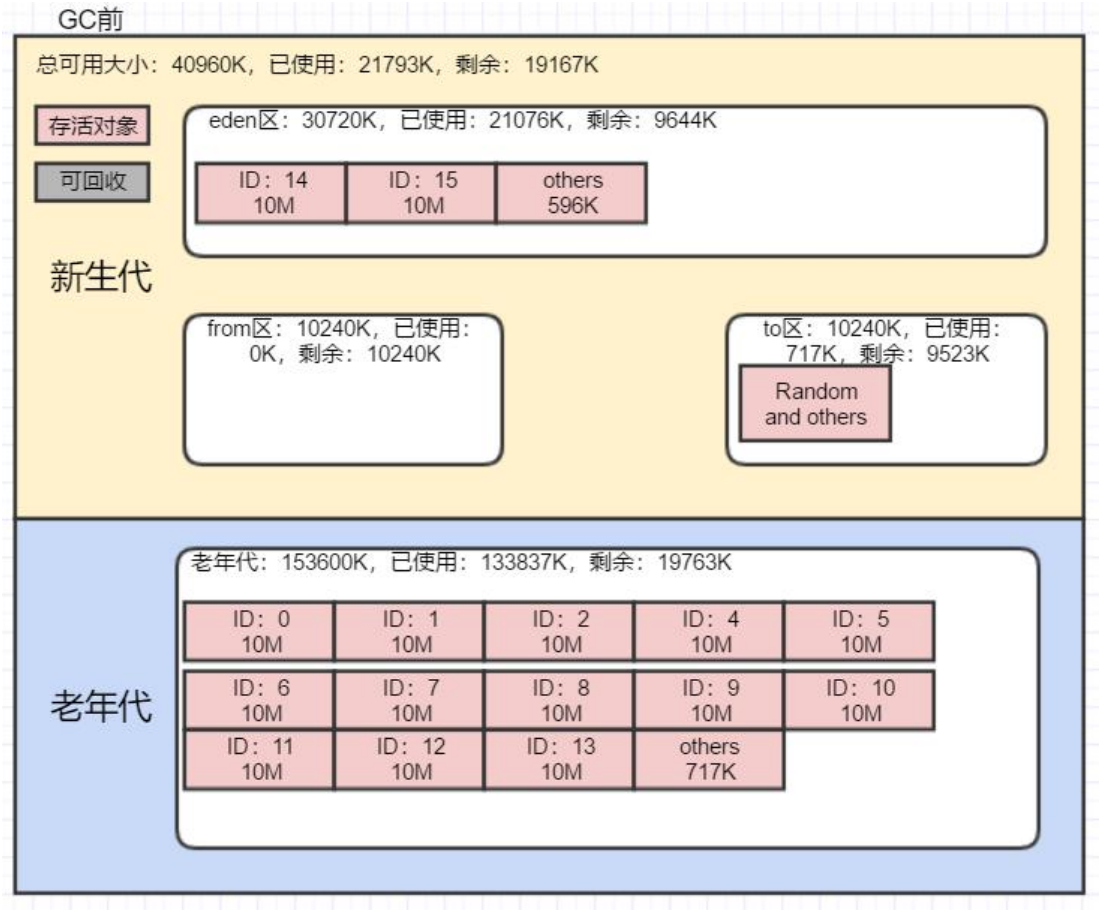
```
created byte[]: 12
created byte[]: 13
0.328: [GC (Allocation Failure) 0.328: [DefNew: 21792K->717K(40960K), 0.0162437 secs] 1344
2K->133837K(194560K), 0.0162844 secs] [Times: user=0.00 sys=0.01, real=0.02 secs]
```

原文链接：通过 Java 串行收集器（SerialGC）深入理解分代 GC 过程

再创建ID：14，ID：15对象后，又需要新生代GC

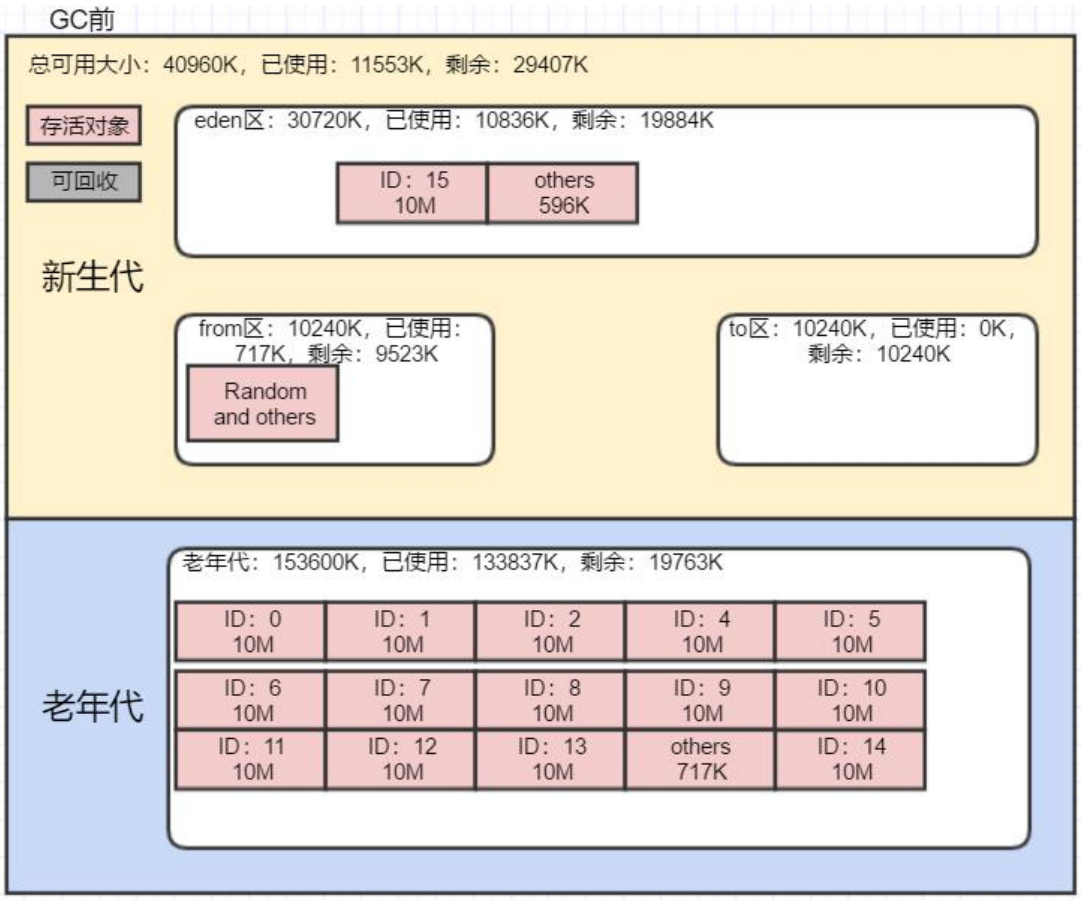
```
created byte[]: 14
created byte[]: 15
0.347: [GC (Allocation Failure) 0.347: [DefNew: 21793K->21793K(40960K), 0.0000201 secs]0.3
7: [Tenured: 133120K->143360K(153600K), 0.0103885 secs] 154913K->154316K(194560K), [M
taspace: 3350K->3350K(1056768K)], 0.0104608 secs] [Times: user=0.02 sys=0.00, real=0.01 se
s]
```

GC前如下所示



在新生代GC时，要把ID：14，ID：15的对象复制到老年代，但此时老年代已经不足以容纳这两个对，此时会触发老年代的GC。

即日志中的Tenured部分。但发现没有任何对象可以回收，然后尝试复制了Eden区的一个对象到老年代



然后继续创建对象，会继续尝试Full GC，Full GC无果，最终发生内存溢出。

```
Exception in thread "main" created byte[]: 16
0.361: [Full GC (Allocation Failure) 0.361: [Tenured: 143360K->143360K(153600K), 0.0028089
ecs] 165153K->164556K(194560K), [Metaspace: 3350K->3350K(1056768K)], 0.0028543 secs] [
imes: user=0.00 sys=0.00, real=0.00 secs]
0.364: [Full GC (Allocation Failure) 0.364: [Tenured: 143360K->143360K(153600K), 0.0050038
ecs] 164556K->164538K(194560K), [Metaspace: 3350K->3350K(1056768K)], 0.0050390 secs] [
imes: user=0.00 sys=0.00, real=0.00 secs]
Disconnected from the target VM, address: '127.0.0.1:57881', transport: 'socket'
java.lang.OutOfMemoryError: Java heap space
Heap
    at com.example.demo.gcdemo.SerialGCDemo.main(SerialGCDemo.java:28)
```

4 总结

首先介绍了Serial的特点以及存在的问题，SerialGC是串行收集器，在收集时会产生STW，停顿时间长导致用户体验差。

然后通过实战，介绍了如何指定JVM的每一块堆内存。

最后通过一个案例，详细描述了SerialGC的整个过程以及内存变化。