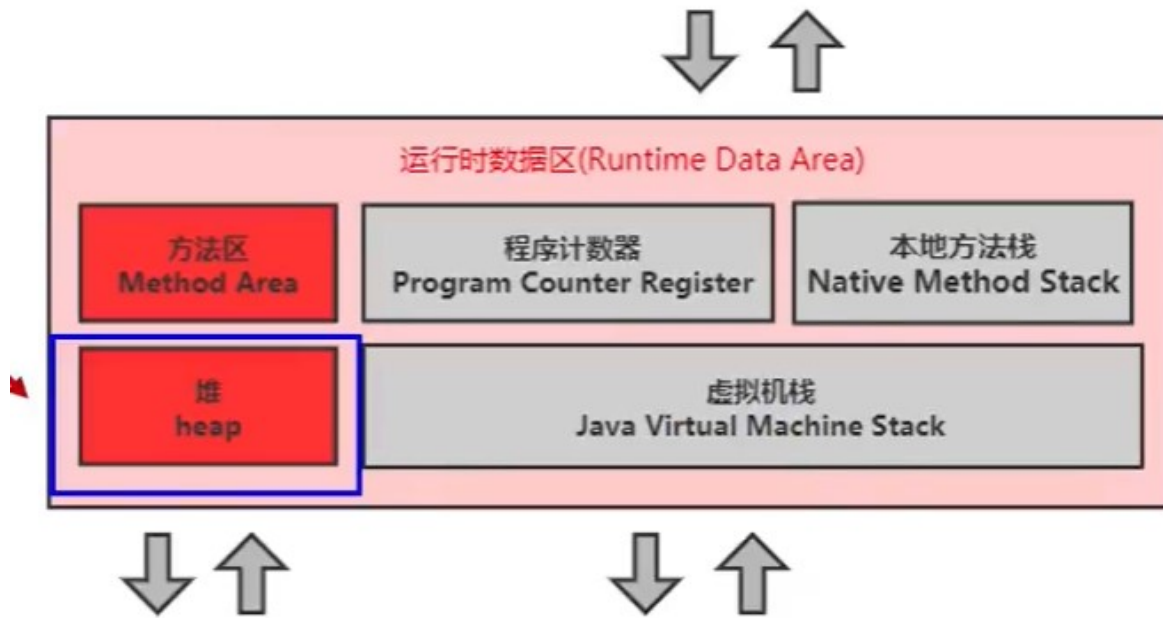


# 堆核心概述

## 核心概述



- 一个JVM实例只存在一个堆内存,堆也是Java内存管理的核心区域;
- Java堆区在 JVM启动的时候即被创建, 其空间大小也就确定了; 是JVM管理的最大一块内存空间  
堆内存的大小是可以调节的;
- <<Java虚拟机规范>> 规定, 堆可以在处于物理上不连续的内存空间中, 但在逻辑上它应该被视为连续的;
- 所有的线程共享Java堆, 在这里还可以划分线程私有的缓冲区(Thread Local Allocation Buffer, TLAB);  
也就是说, 堆空间并不是完全所有线程公有的, 堆里面会划分出缓冲区, 给每个线程私有也就是 TLAB
- <<Java虚拟机规范>>中对Java堆的描述是:所有的对象实例以及数组都应当在运行时分配在堆上;  
"几乎"所有的对象实例都在这里分配内存, 从实际角度来看;
- 数组和对象可能永远不会储存在栈上, 因为栈帧中保存引用, 这个引用指向对象或者数组

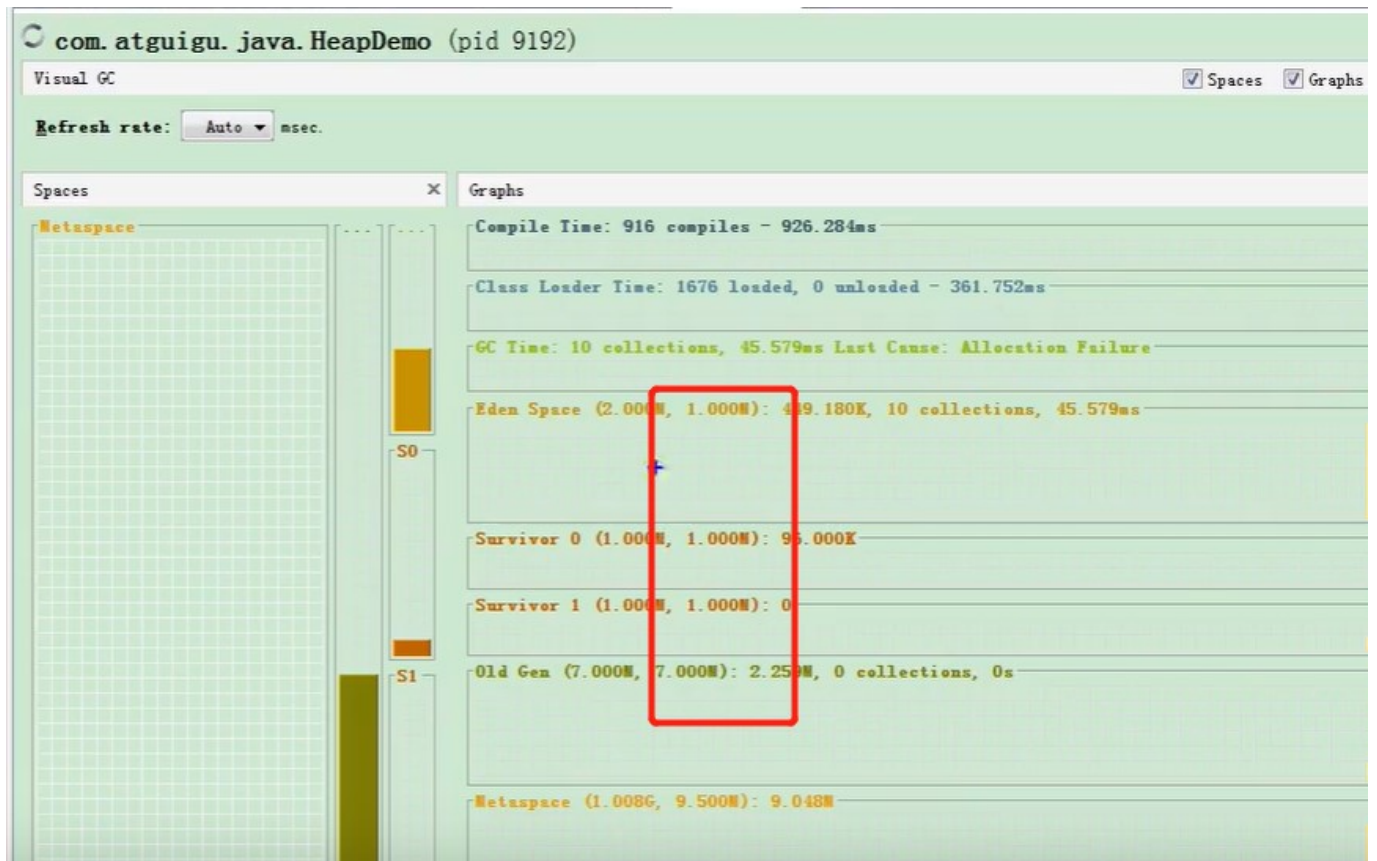
在堆中的位置

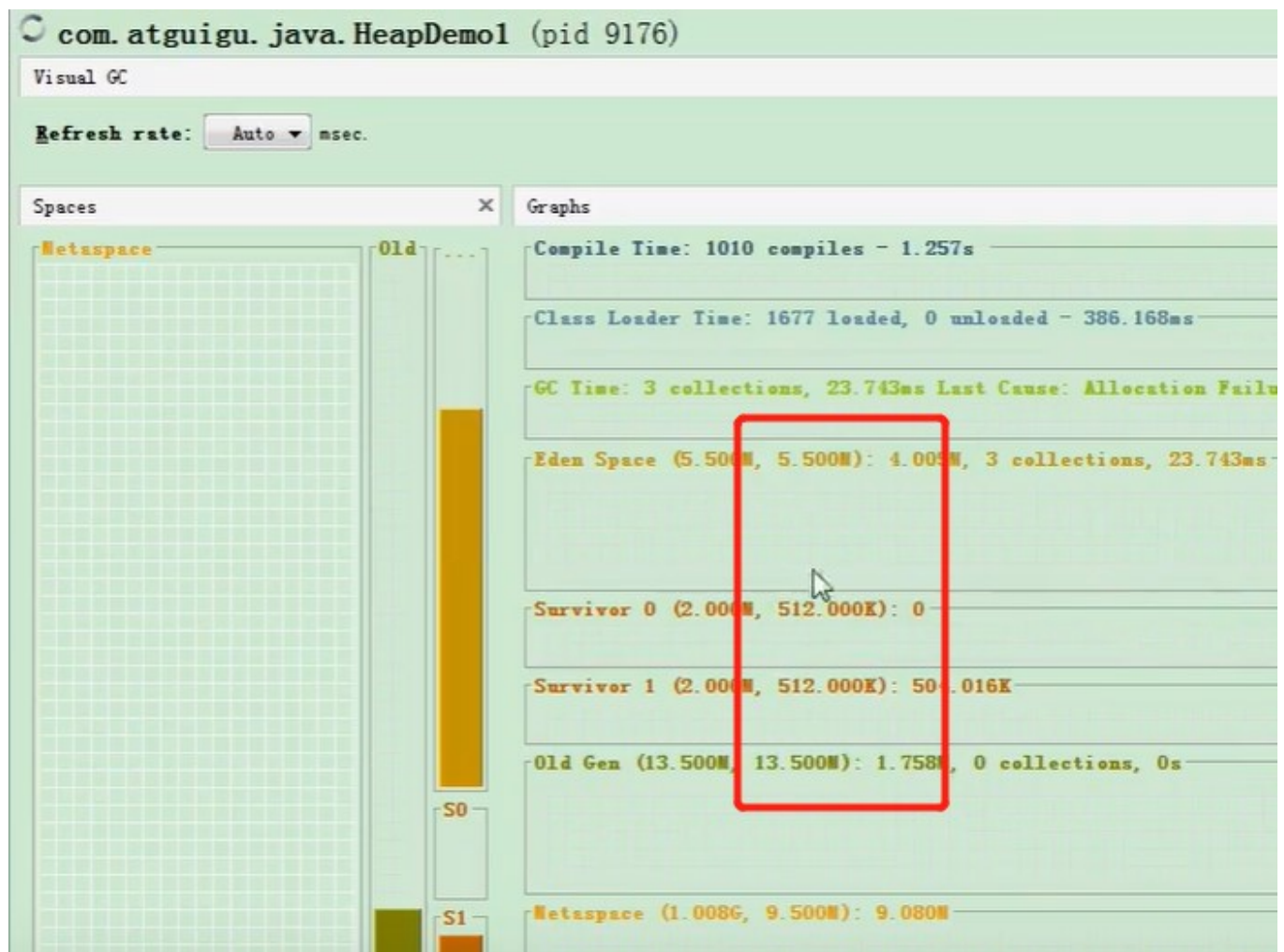
- 在方法结束后, 堆中的对象不会马上被移除, 仅仅在垃圾收集的时候才会被移除
- 堆, 是GC(Garbage Collection, 垃圾收集器) 执行垃圾回收的重点区域;

**证明: 每个进程都有自己独享的堆空间**

```
1  /**
2   * -Xms10m -Xmx10m
3   *
4   * @author shkstart shkstart@126.com
5   * @create 2020 16:41
6   */
7  public class HeapDemo {
8      public static void main(String[] args) {
9          System.out.println("start...");
10         try {
11             Thread.sleep(1000000);
12         } catch (InterruptedException e) {
13             e.printStackTrace();
14         }
15
16         System.out.println("end...");
17     }
18 }
19
20
21
22
23 /**
24  * -Xms20m -Xmx20m
25  * @author shkstart shkstart@126.com
26  * @create 2020 16:42
27  */
28 public class HeapDemo1 {
29     public static void main(String[] args) {
30         System.out.println("start...");
31         try {
32             Thread.sleep(1000000);
```

```
33     } catch (InterruptedException e) {
34         e.printStackTrace();
35     }
36
37     System.out.println("end...");
38 }
39 }
40
```





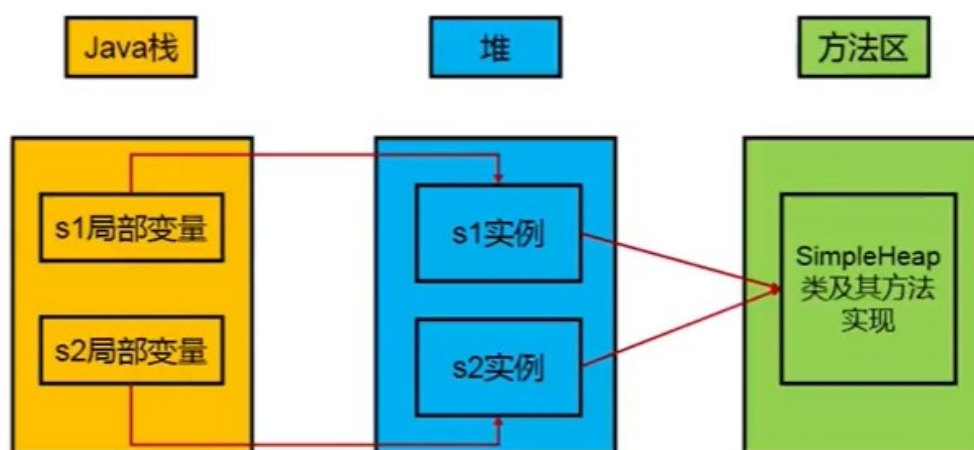
使用工具 jvisualvm.exe 可以看到每个进程的堆, 是不一样的

```
1  /**
2   * @author shkstart shkstart@126.com
3   * @create 2020 17:28
4   */
5  public class SimpleHeap {
6      private int id;//属性、成员变量
7
8      public SimpleHeap(int id) {
9          this.id = id;
10     }
11
12     public void show() {
13         System.out.println("My ID is " + id);
14     }
15     public static void main(String[] args) {
16         SimpleHeap s1 = new SimpleHeap(1);
```

```

17     SimpleHeap s2 = new SimpleHeap(2);
18
19     int[] arr = new int[10];
20
21     Object[] arr1 = new Object[10];
22 }
23 }
24

```



## 内存细分

现代垃圾收集器大部分都基于分代收集理论设计, 堆空间细分为

**现代垃圾收集器大部分都基于分代收集理论设计, 堆空间细分为：**

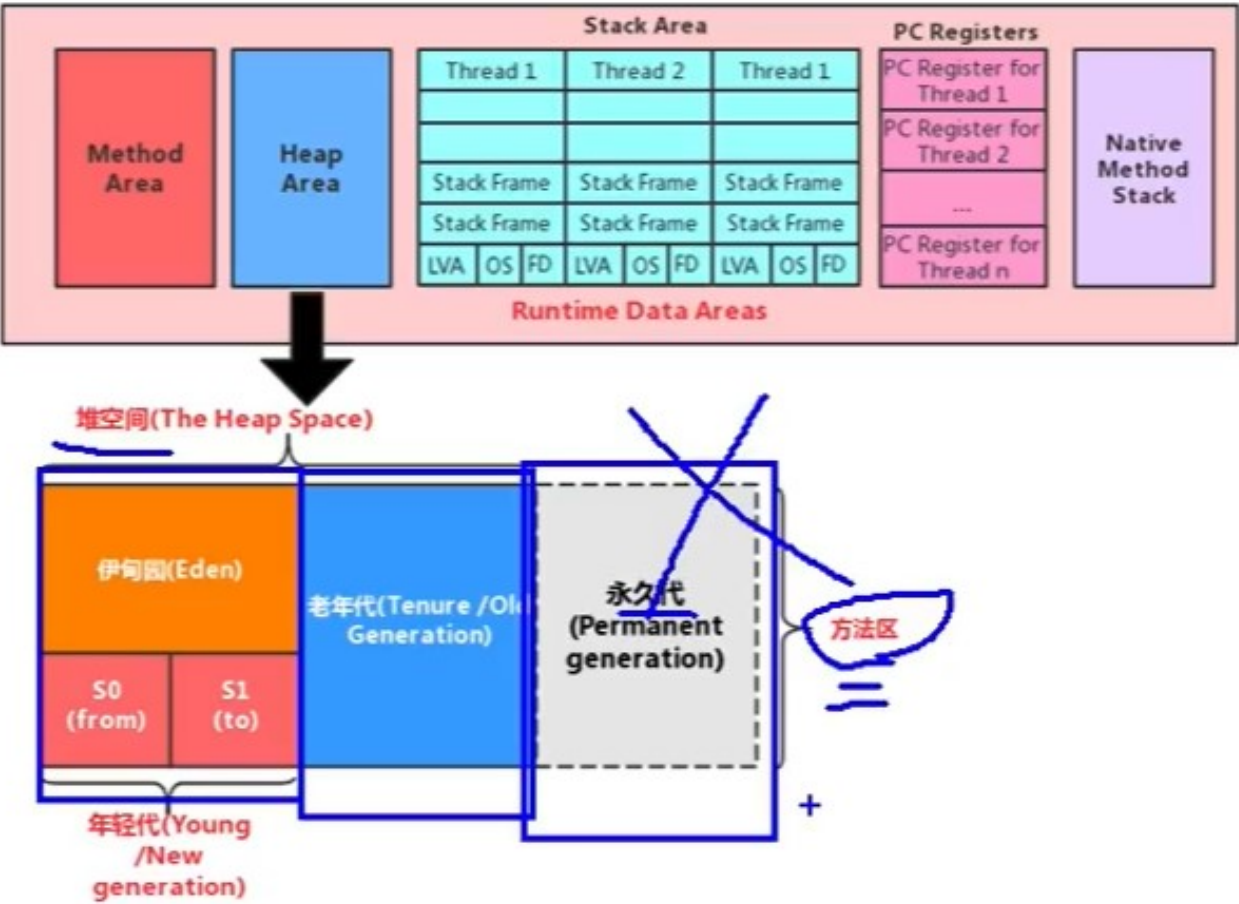
- Java 7及之前堆内存逻辑上分为三部分：新生区+养老区+永久区
  - Young Generation Space 新生区 Young/New
    - ✓ 又被划分为Eden区和Survivor区
  - Tenure generation space 养老区 Old/Tenure
  - Permanent Space 永久区 Perm

- Java 8及之后堆内存逻辑上分为三部分：新生区+养老区+元空间
  - Young Generation Space 新生区 Young/New
    - ✓ 又被划分为Eden区和Survivor区
  - Tenure generation space 养老区 Old/Tenure
  - Meta Space 元空间 Meta

约定：新生区 ⇄ 新生代 ⇄ 年轻代 养老区 ⇄ 老年区 ⇄ 老年代 永久区 ⇄ 永久代

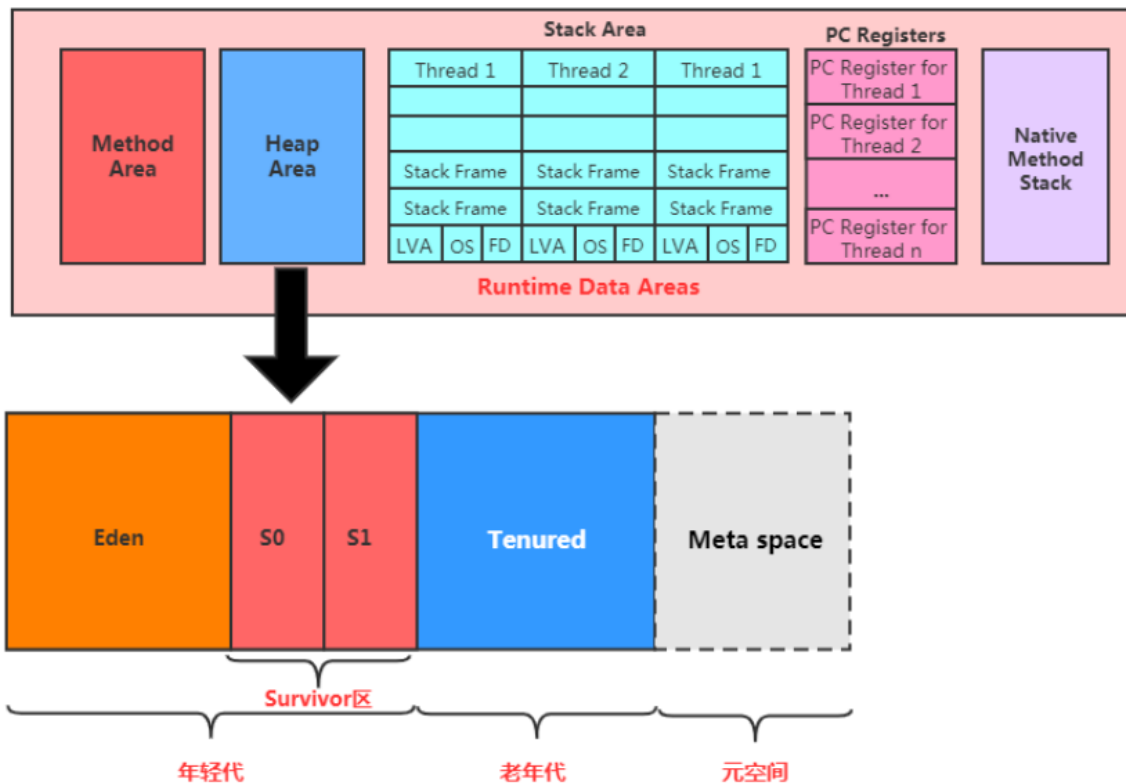
让天下没有难存的技术

java7



java 8





使用添加参数, 可以打印出 GC的细节

```
1 -XX:+PrintGCDetails
```

```
Heap
PSYoungGen      total 6144K, used 2187K [0x00000000ff980000, 0x0000000010000000, 0x0000000010000000)
eden space 5632K, 38% used [0x00000000ff980000,0x00000000ffba2cc0,0x00000000fff00000)
from space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000010000000)
to space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)
ParOldGen       total 13824K, used 0K [0x00000000fec00000, 0x00000000ff980000, 0x00000000ff980000)
object space 13824K, 0% used [0x00000000fec00000,0x00000000fec00000,0x00000000ff980000)
Metaspace       used 3454K, capacity 4496K, committed 4864K, reserved 1056768K
class space     used 381K, capacity 388K, committed 512K, reserved 1048576K

Process finished with exit code 0
```

from to, 幸存者 1,2区

## 设置堆内存大小与OOM

## 参数与指令

- **所指的堆区内存, 堆空间指的都是 新生代+老年代的大小**

- Java 堆区用于存储Java对象实例, 那么堆的大小在JVM启动时就已经设定好了, 大家可以通过选项

"-Xms" "-Xmx" 两个参数来进行设置;

"-Xms": 用于表示堆区的初始内存, 等价于 -XX:InitialHeapSize

"-Xmx": 用于表示堆区的最大内存, 等价于 -XX:MaxHeapSize

-X 是jvm的运行参数

ms 是 memory start

mx 是 memory max

- 一旦堆区中的内存大小超过了 "-Xmx" 所制定的最大内存时, 将会抛出 OutOfMemoryError异常
- 通常会将 -Xms 和 -Xmx 两个参数配置相同的值,其目的是为了能够在java垃圾回收机制清理完堆区后不需要重新分隔计算堆区的大小, 从而提高性能;
- 默认情况下,

初始内存大小: 物理电脑内存大小 / 64

最大内存大小: 物理电脑内存大小 / 4

```
1  /**
2   * 1. 设置堆空间大小的参数
3   * -Xms 用来设置堆空间（年轻代+老年代）的初始内存大小
4   *      -X 是jvm的运行参数
5   *      ms 是memory start
6   * -Xmx 用来设置堆空间（年轻代+老年代）的最大内存大小
7   *
8   * 2. 默认堆空间的大小
9   *      初始内存大小: 物理电脑内存大小 / 64
10  *      最大内存大小: 物理电脑内存大小 / 4
11  * 3. 手动设置: -Xms600m -Xmx600m
12  *      开发中建议将初始堆内存和最大的堆内存设置成相同的值。
13  *
14  * 4. 查看设置的参数: 方式一: jps / jstat -gc 进程id
15  *      方式二: -XX:+PrintGCDetails 需要程序执行完毕
```



```

16  *
17  * @create 2020 20:15
18  */
19  public class HeapSpaceInitial {
20      public static void main(String[] args) {
21
22          //返回Java虚拟机中的堆内存总量
23          long initialMemory = Runtime.getRuntime().totalMemory() / 1024 / 1024;
24          //返回Java虚拟机试图使用的最大堆内存量
25          long maxMemory = Runtime.getRuntime().maxMemory() / 1024 / 1024;
26
27          System.out.println("-Xms : " + initialMemory + "M");
28          System.out.println("-Xmx : " + maxMemory + "M");
29
30          //      System.out.println("系统内存大小为: " + initialMemory * 64.0 / 1024 +
31          //      System.out.println("系统内存大小为: " + maxMemory * 4.0 / 1024 + "G");
32
33          try {
34              Thread.sleep(1000000);
35          } catch (InterruptedException e) {
36              e.printStackTrace();
37          }
38      }
39  }

```

为什么获得的堆内存, 不是设置的 600M呢

The screenshot shows an IDE with a project structure on the left and a code editor on the right. The code editor displays the same Java code as the first block. A red box highlights the comment '3. 手动设置: -Xms600m -Xmx600m' in the code. Below the code editor, the 'Run' configuration is visible, showing the command 'D:\developer\_tools\jdk\jdk1.8.0\_131\bin\java.exe ...'. A red box highlights the JVM arguments '-Xms : 575M' and '-Xmx : 575M' in the run configuration.

```
1 jstat -gc pidid
```

```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>jps
9344 Launcher
6004 Jps
3564
9228 HeapSpaceInitial 600M
C:\Users\Administrator>jstat -gc 9228
S0C  S1C  S0U  S1U   EC    EU    OC    OU    MC    MU    CCSC  CCSU    YGC    YGCT  FGC    FGCT    GCT
25600.0 25600.0 0.0  0.0  153600.0 12288.1 409600.0 0.0  4480.0 770.4 384.0 75.9    0    0.000 0    0.000 0.000
C:\Users\Administrator>
```

C就是分配总量, U就是使用的总量

因为 S0 和 S1 也就是幸存者区, 总有一个是空的, 并不是两个都可以进行存储对象!

## OOM举例

```
public class OOMTest {
    public static void main(String[] args) {
        ArrayList<Picture> list = new ArrayList<>();
        while(true){
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            list.add(new Picture(new Random().nextInt(1024 * 1024)));
        }
    }
}
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at com.atguigu.java.Picture.<init>(OOMTest.java:25)
    at com.atguigu.java.OOMTest.main(OOMTest.java:16)
```

..... 下没有难学的技术

```
1 /**
2  * -Xms600m -Xmx600m
3  *
4  * @create 2020 21:12
5  */
6 public class OOMTest {
7     public static void main(String[] args) {
8         ArrayList<Picture> list = new ArrayList<>();
9         while(true){
10             try {
11                 Thread.sleep(20);
12             } catch (InterruptedException e) {
```

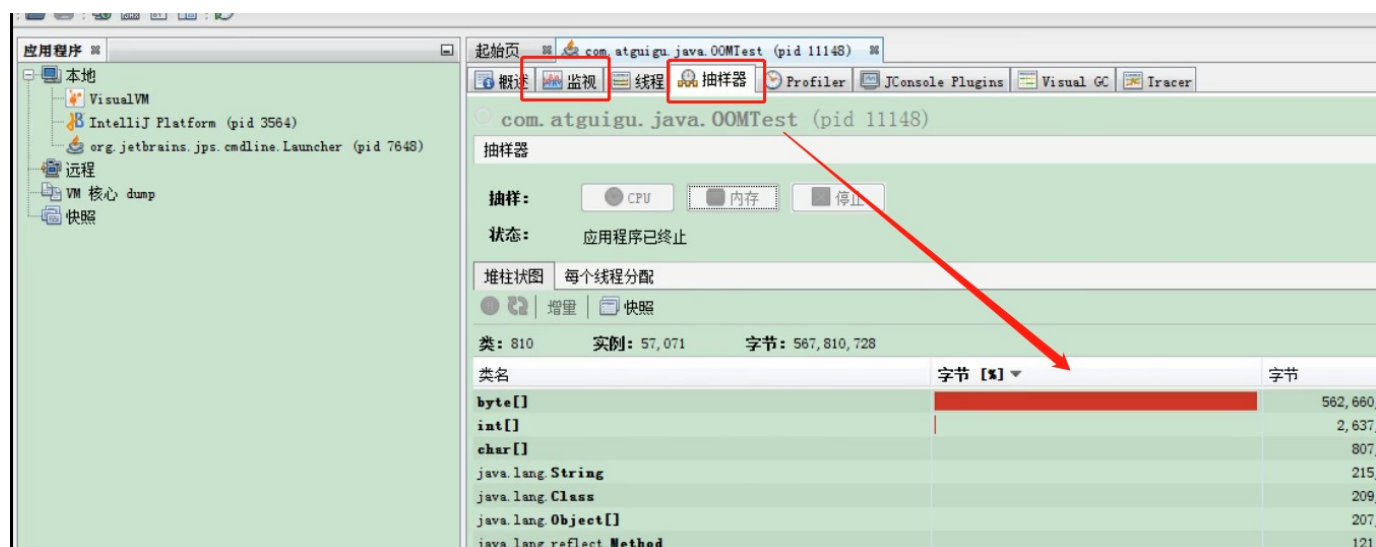
```

13         e.printStackTrace();
14     }
15     list.add(new Picture(new Random().nextInt(1024 * 1024)));
16 }
17 }
18 }
19
20 class Picture{
21     private byte[] pixels;
22
23     public Picture(int length) {
24         this.pixels = new byte[length];
25     }
26 }
27

```

## 工具分析

可以通过 jVisualVm 工具查看到, 监视就是 新生代和老年代的情况, 抽样器里面就可以看到是哪个 对象造成的内存过大的问题

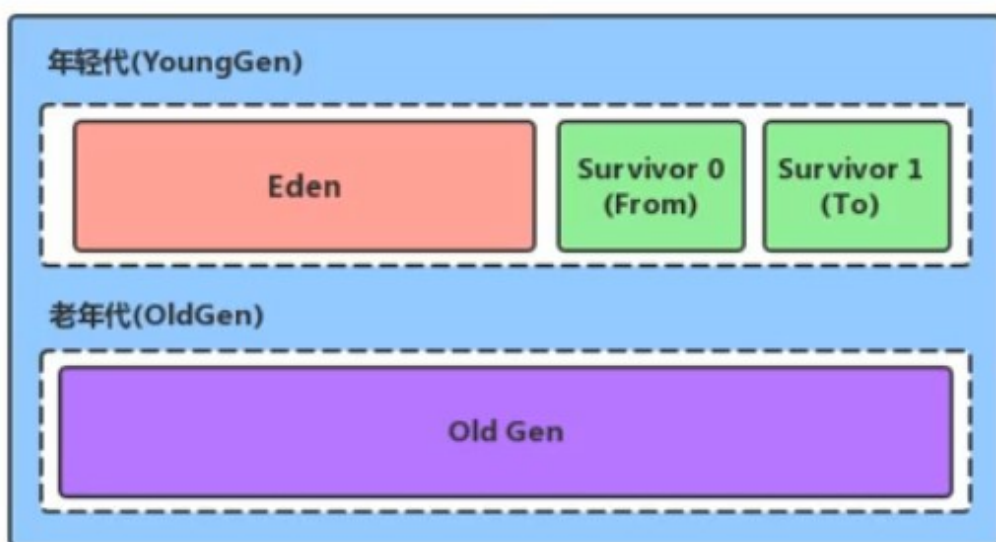


## 年轻代与老年代

### 概述

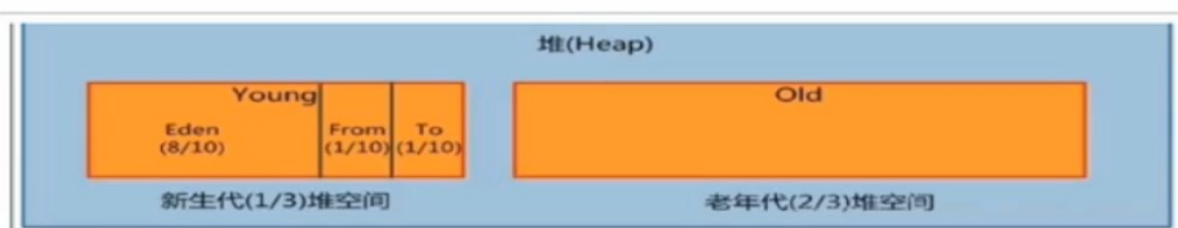
- 存储在JVM中的Java对象可以被划分为两类:

1. 一类生命周期较短的瞬时对象, 这类对象的创建和消亡都非常的迅速
  2. 一类对象的生命周期却非常长, 在某些极端的情况下还能够与JVM的生命周期保持一致
- Java堆区进一步细分的话, 可以分为年轻代 YoungGen 和 老年代 OldGen
  - 其中年轻代又可以划分为 Eden空间, Survivor0空间和 Survivor1空间 (有时也可称之为 from区和 to区)



## 年轻代和老年代比例

下面这参数开发中一般不会调:



- 配置新生代与老年代在堆结构的占比。
  - 默认 `-XX:NewRatio=2`, 表示新生代占1, 老年代占2, 新生代占整个堆的1/3
  - 可以修改 `-XX:NewRatio=4`, 表示新生代占1, 老年代占4, 新生代占整个堆的1/5

```

1  /**
2   * -Xms600m -Xmx600m
3   *
4   * -XX:NewRatio : 设置新生代与老年代的比例。默认值是2.
5   * -XX:SurvivorRatio : 设置新生代中Eden区与Survivor区的比例。默认值是8
6   * -XX:-UseAdaptiveSizePolicy : 关闭自适应的内存分配策略 （暂时用不到）
7   * -Xmn:设置新生代的空间的大小。 （一般不设置）
8   *
9   * @author shkstart shkstart@126.com
10  * @create 2020 17:23
11  */
12 public class EdenSurvivorTest {
13     public static void main(String[] args) {
14         System.out.println("我只是来打个酱油~");
15         try {
16             Thread.sleep(1000000);
17         } catch (InterruptedException e) {
18             e.printStackTrace();
19         }
20     }
21 }

```

可以通过jps来观察, 也可以通过 jVisualVM里面的 Visual GC插件来观察, 算下来比例 默认 1:2

```

C:\Users\Administrator>jps
10416 Main
4308 KotlinCompileDaemon
3564
6988 EdenSurvivorTest
8476 Launcher
9948 Jps

C:\Users\Administrator>jinfo -flag SurvivorRatio 6988
-XX:SurvivorRatio=8

C:\Users\Administrator>jinfo -flag NewRatio 6988
-XX:NewRatio=2

```

# 年轻代内部比例

- 在HotSpot中, Eden空间和另外两个Survivor空间缺省所占的比例是8:1:1
- 当然开发人员可以通过选项, "-XX:SurvivorRatio" 调整整个空间比例; 比如 -XX:SurvivorRatio=8
- 几乎所有的Java对象都是在Eden去被new出来的;
- 绝大部分的Java对象的销毁都在新生代进行了;  
IBM 公司的专门研究表明, 新生代中 80% 的对象都是 "朝生夕死"的;
- 可以使用选项 "-Xmn" 设置新生代最大内存大小  
这个参数一般使用默认值就可以了

注意: 在使用的时候 并不是 8:1:1 有一个默认的应用比例

-XX:-UseAdaptiveSizePolicy "-"号是关闭的意思, 但是关闭了依旧不是 8:1:1

要想真的是 8:1:1 需要显示的设置 -XX:SurvivorRatio=8

-Xmn:设置新生代的空间的大小。(一般不设置) 优先级 高于 -XX:NewRatio 也就是 晚加载

## 图解对象分配过程

### 概述

1. new的对象先放伊甸园区, 此区有大小限制;
2. 当伊甸园的空间填满时, 程序又需要创建对象, JVM的垃圾回收器将对伊甸园区进行垃圾回收(Minor GC), 将伊甸园区中不再被其他对象所引用的对象进行销毁; 再加载新的对象放到伊甸园区;
3. 然后将伊甸园中的剩余对象移动到幸存者0区
4. 如果再次触发垃圾回收, 此时上次幸存下来的放到幸存者0区的, 如果没有回收, 就会放到幸存者1区;
5. 如果再次经历垃圾回收, 此时会重新放回幸存者0区, 接着再去幸存者1区;



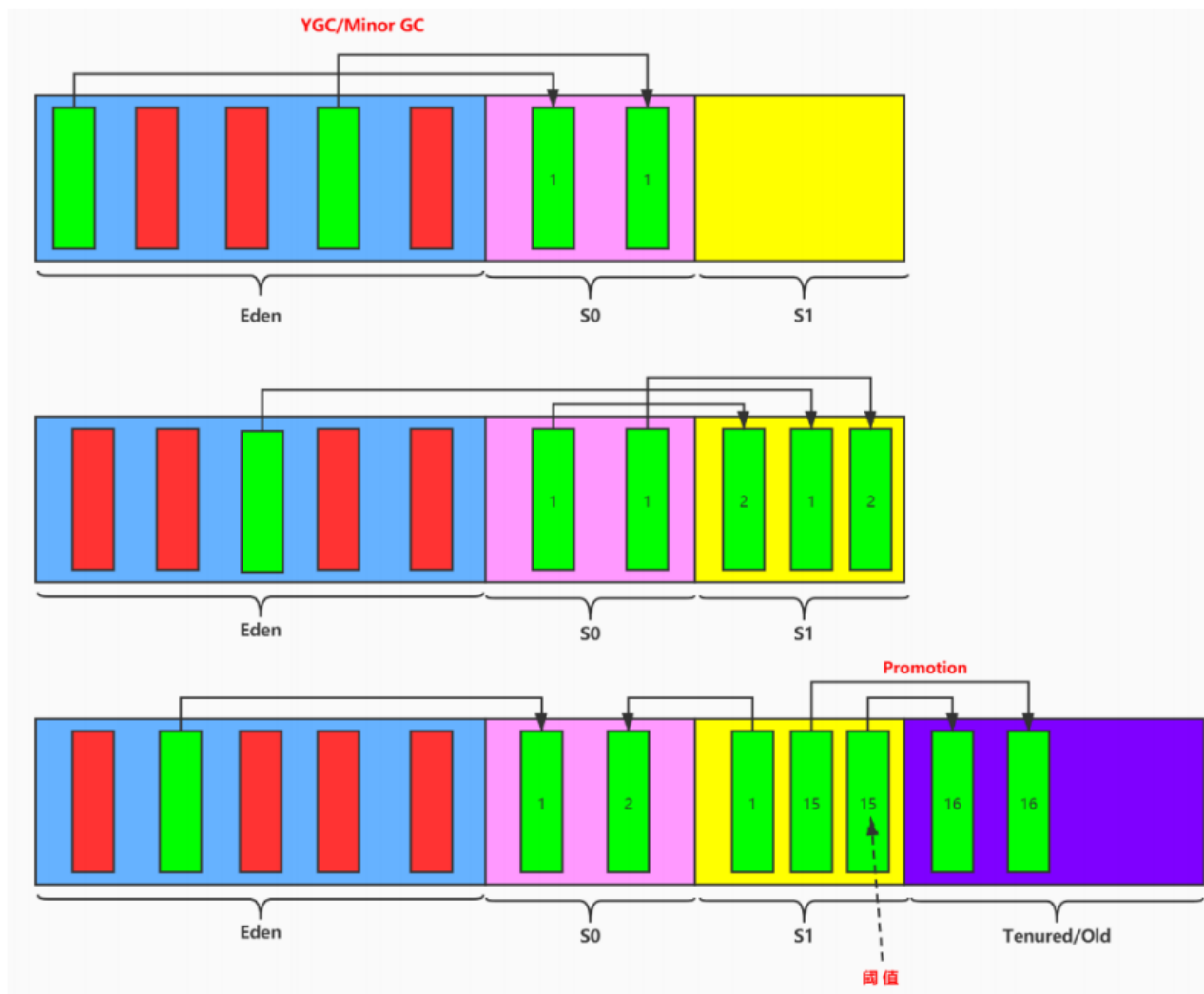
6. 啥时候去老年区? 可以设置次数, 默认是15次

可以设置参数 `-XX:MaxTenuringThreshold=<N>` 进行设置;

7. 在老年代, 相对悠闲; 当老年代内存不足时, 再次触发GC: Major GC, 进行老年代的内存清理;

8. 若老年代执行了Major GC之后发现依然无法进行对象的保存, 就会产生OOM异常

**java.lang.OutOfMemoryError: Java heap space**

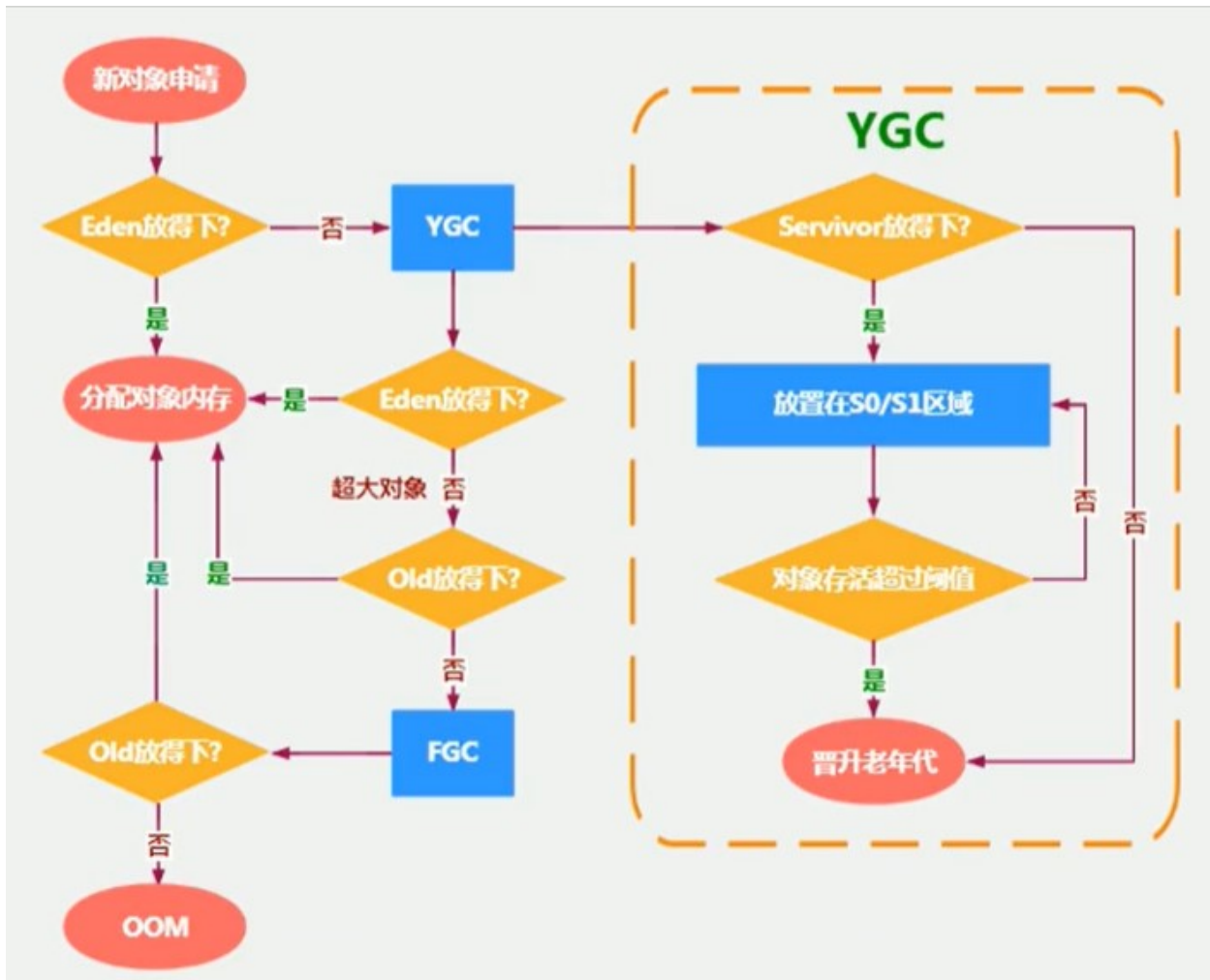


在 survivor区, 会有一个年龄技术器, 默认到达15 进去老年代

**当 Eden区满的时候, 触发 YGC, 会同时回收 Eden 和 S0,1; 但是幸存者区满的时候, 不会触发GC, 但不代表不能GC ;**

## 小结

- 针对幸存者S0,S1区的总结: 复制之后有交换,谁空谁是to
- 关于垃圾回收: 频繁在新生代手机, 很少在老年代收集, 几乎不在永久区/元空间收集;



## 代码举例

```

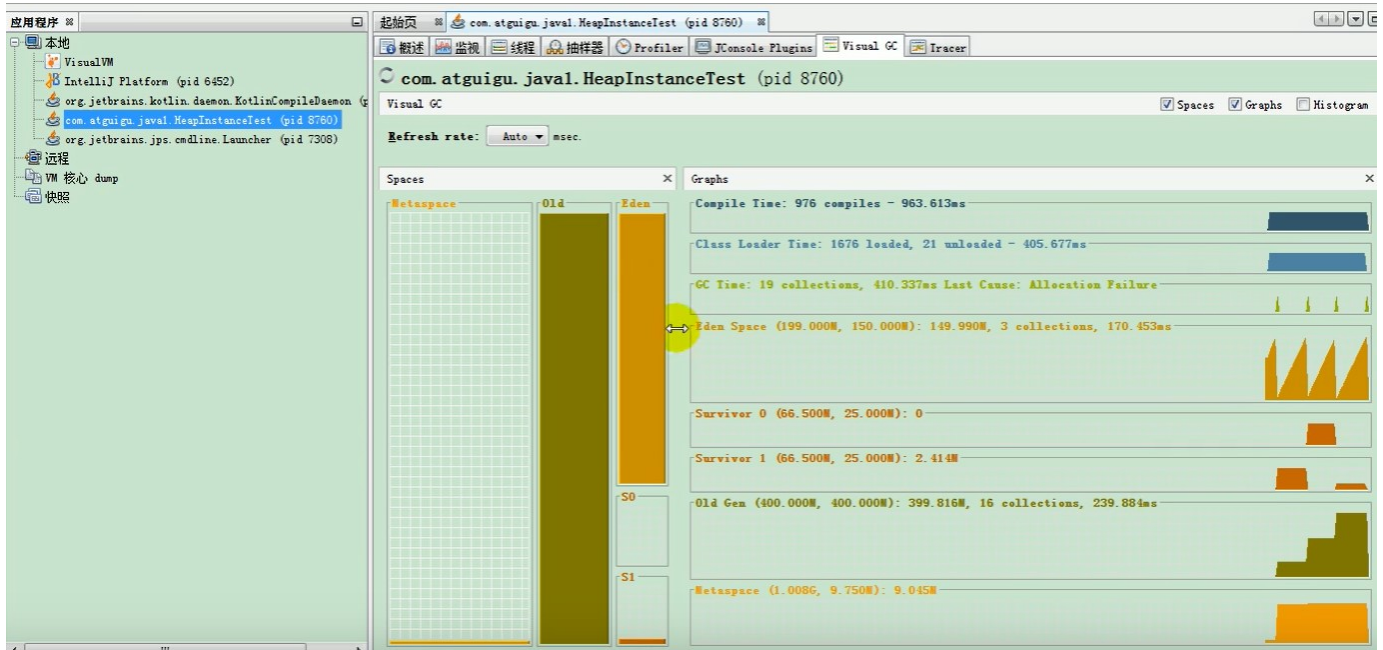
1  /**
2   * -Xms600m -Xmx600m
3   * @author shkstart shkstart@126.com
4   * @create 2020 17:51
5   */
6  public class HeapInstanceTest {
7      byte[] buffer = new byte[new Random().nextInt(1024 * 200)];
8
9      public static void main(String[] args) {
10         ArrayList<HeapInstanceTest> list = new ArrayList<HeapInstanceTest>();
11         while (true) {
12             list.add(new HeapInstanceTest());
13             try {
14                 Thread.sleep(10);
15             } catch (InterruptedException e) {

```

```

16         e.printStackTrace();
17     }
18 }
19 }
20 }
21

```



可以看到 GC的实际过程, 坡度图, 以及增长情况

## Minor GC、Major GC、Full GC

### 概述

- JVM在进行GC时, 并非每次都对上面三个内存区域(**新生代,老年代,方法区**)一起回收的, **大部分时候回收的都是指新生代;**
- 针对HotSpot VM的实现, 它里面的GC按照回收区域又分为两大种类型: 一种是部分收集 (Partial GC), 一种是整堆收集(Full GC)
- 部分收集: 不是完整收集整个Java堆的垃圾收集;其中又分为:
  - 新生代收集(Minor GC / Young GC): 只是新生代的垃圾收集;
  - 老年代收集(Major GC / Old GC): 只是老年代的垃圾收集;

目前, 只有 CMS GC会有单独收集老年代的行为;

注意, 很多时候 Major GC 会和Full GC 混淆使用, 需要具体分辨是老年代回收还是整堆回收;

混合收集 (Mixed GC): 手机整个新生代以及部分老年代的垃圾收集;

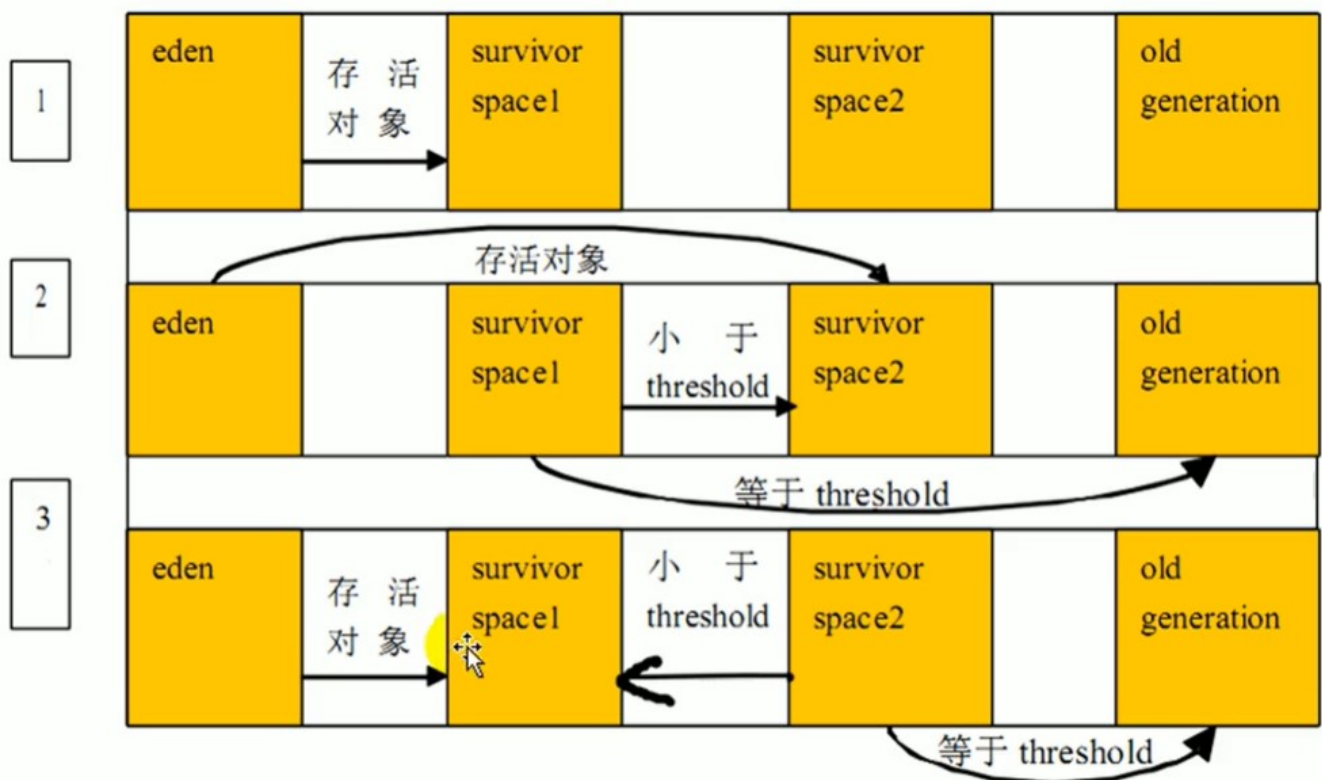
目前, 只有 G1 GC 会有着这种行为

- 整堆收集(Full GC): 收集整个java堆和方法区的垃圾收集;

## Minor GC

新生代GC (Minor GC)触发机制:

- 当年轻代空间不足时, 就会出发Minor GC, 这里的年轻代满指的是Eden代满, Survivor满不会引发GC; (每次 Minor GC 会清理新生代的内存;)
- 因为 Java 对象大多都具备朝生夕灭的特性, 所以 Minor GC 非常频繁, 一般回收速度也比较快; 这一定义既清晰又易于理解;
- Minor GC会引发 STW, 暂停其他用户的线程, 等垃圾回收结束, 用户线程才恢复运行



## Major GC

老年代GC (Major GC)触发机制:

- 指发生在老年代的GC, 对象从老年代消失时, 我们说"Major GC" 或 "Full GC" 发生了;
- 出现了Major GC, 经常会伴随至少一次的Minor GC(但非绝对的, 在 Parallel Scavenge 收集器的收集策略里就有直接进行Major GC的策略选择过程)

也就是在老年代空间不足时, 会先尝试触发Minor GC, 如果之后空间还不足, 则触发Major GC

- Major GC的速度一般会比Minor GC慢10倍以上, STW的时间更长;
- 如果Major GC后, 内存还不足, 就报OOM了;

## Full GC

Full GC触发机制:

触发Full GC执行的情况有如下五种:

1. 调用System.gc()时,系统建议执行Full GC, 但是不必然执行
2. 老年代空间不足
3. 方法区空间不足
4. 通过Minor GC后进入老年代的平均大小大于老年代的可用内存
5. 由Eden区、survivor space0(From Space)区 向 survivor space1(To Space)区复制时, 对象大小大于To Space可用内存, 则把该对象转存到老年代, 且老年代的可用内存小于该对象大小

**说明: Full GC 是开发或调优中尽量要避免的; 这样暂时时间会短一些;**

**并且 Full GC 条件比较多, 所以用GC 日志来判断是不是 老年代的问题**

## GC日志, 代码举例

```
1 -XX:+PrintGCDetails 参数查看GC日志
2
```

```

3  /**
4   * 测试MinorGC、MajorGC、FullGC
5   * -Xms9m -Xmx9m -XX:+PrintGCDetails
6   * @author shkstart shkstart@126.com
7   * @create 2020 14:19
8   */
9  public class GCTest {
10     public static void main(String[] args) {
11         int i = 0;
12         try {
13             List<String> list = new ArrayList<>();
14             String a = "atguigu.com";
15             while (true) {
16                 list.add(a);
17                 a = a + a;
18                 i++;
19             }
20
21         } catch (Throwable t) {
22             t.printStackTrace();
23             System.out.println("遍历次数为: " + i);
24         }
25     }
26 }
27

```

```

[GC (Allocation Failure) [PSYoungGen: 2039K->502K(2560K)] 2039K->842K(9728K), 0.0008309 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 2388K->508K(2560K)] 2728K->2080K(9728K), 0.0007249 secs] [Times: user=0.00 sys=0.00, real=0.00 sec]
[GC (Allocation Failure) [PSYoungGen: 1977K->498K(2560K)] 3549K->2841K(9728K), 0.0006675 secs] [Times: user=0.00 sys=0.00, real=0.00 sec]
[Full GC (Ergonomics) [PSYoungGen: 1297K->0K(2560K)] [ParOldGen: 6567K->4849K(7168K)] 7864K->4849K(9728K), [Metaspace: 3432K->3432K(1056K)] 0.0013918 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] 4849K->4849K(9728K), 0.0013918 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] [ParOldGen: 4849K->4831K(7168K)] 4849K->4831K(9728K), [Metaspace: 3432K->3432K(1056K)] 0.0013918 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
遍历次数为: 16
Heap
 PSYoungGen      total 2560K, used 90K [0x00000000ffdb0000, 0x0000000010000000, 0x0000000010000000)
  eden space 2048K, 4% used [0x00000000ffdb0000, 0x00000000ffdb16a8, 0x00000000ffdb0000)
  from space 512K, 0% used [0x00000000ffdb8000, 0x00000000ffdb8000, 0x0000000010000000)
  to   space 512K, 0% used [0x00000000ffdb0000, 0x00000000ffdb0000, 0x00000000ffdb8000)
 ParOldGen       total 7168K, used 4831K [0x00000000ff600000, 0x00000000ffdb0000, 0x00000000ffdb0000)
  object space 7168K, 67% used [0x00000000ff600000, 0x00000000ffab7d40, 0x00000000ffdb0000)
 Metaspace       used 3469K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 379K, capacity 388K, committed 512K, reserved 1048576K
java.lang.OutOfMemoryError: Java heap space

```

GC前后, 新生代大小变化      堆总空间大小

**GC 就是 youngGC; Full GC 指的是 Full GC**

**OOM之前一定经历过 Full GC**



# 堆空间分代思想

## 原因

为什么需要把Java堆分代?不分代就不能正常工作了吗?

- 经研究,不同对象的生命周期不同; 70%-90%的对象是临时对象;
- 新生代: 有Eden、两块大小相同的Survivor(又称为from/to, S0/S1)构成,to总为空;
- 老年代: 存放新生代中经历多次GC仍然存活的对象

其实不分代完全可以, 分代的唯一理由就是优化GC性能;如果没有分代, 那所有的对象都在一块, 就如同把一个学校的人都关在了一个教室; GC的时候要找到哪些对象没用, 这样就会对堆的所有区域进行扫描;而很多对象都是朝生夕死的, 如果分代的话,把新创建的对象放到某一地方, 当GC 的时候先把这块存储"朝生夕死"对象的区域进行回收,这样就会腾出很大的空间出来

## 内存分配策略

### 对象提升(Promotion)规则

- 如果对象在Eden 出生并经过第一次Minor GC后仍然存活, 并且能被Survivor容纳的话, 将被移动到Survivor 空间中, 并将对象年龄设为1; 对象在Survivor 区中每熬过一次 MinorGC, 年龄就增加1岁, 当它的年龄增加到一定程度(默认为15岁, 其实每个JVM、每个GC都有所不同)时,就会被晋升到老年代中;
- 对象晋升老年代的年龄阈值, 可以通过选项 -XX:MaxTenuringThreshold来设置;

### 不同年龄段处理

针对不同年龄段的对象分配原则如下所示:

- 优先分配到Eden
- 大对象直接分配到老年代

尽量避免程序中出现过多的大对象

- 长期存活的对象分配到老年代

- 动态对象年龄判断

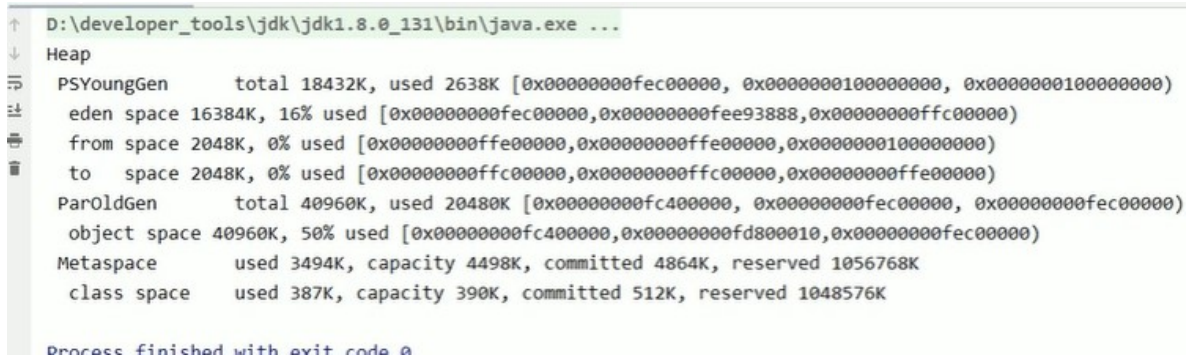
如果Survivor 区中相同年龄的所有对象大小的总和大于Survivor空间的一半, 年龄大于或者等于改年龄的对象可以直接进入老年代, 无须等到 `MaxTenuringThreshold` 中要求的年龄;

- 空间分配担保

`-XX:HandlePromotionFailure`

## 代码示例

```
1  /** 测试：大对象直接进入老年代
2   *  -Xms60m -Xmx60m -XX:NewRatio=2 -XX:SurvivorRatio=8 -XX:+PrintGCDetails
3   *  @author shkstart  shkstart@126.com
4   *  @create 2020  21:48
5   */
6  public class YoungOldAreaTest {
7      public static void main(String[] args) {
8          byte[] buffer = new byte[1024 * 1024 * 20]; //20m
9
10     }
11 }
12
```



The screenshot shows a Windows command prompt window with the following content:

```
D:\developer_tools\jdk\jdk1.8.0_131\bin\java.exe ...
Heap
PSYoungGen      total 18432K, used 2638K [0x00000000fec00000, 0x0000000010000000, 0x0000000010000000)
 eden space 16384K, 16% used [0x00000000fec00000,0x00000000fee93888,0x00000000ffc00000)
  from space 2048K, 0% used [0x00000000ffe00000,0x00000000ffe00000,0x0000000010000000)
 to   space 2048K, 0% used [0x00000000ffc00000,0x00000000ffc00000,0x00000000ffe00000)
ParOldGen       total 40960K, used 20480K [0x00000000fc400000, 0x00000000fec00000, 0x00000000fec00000)
 object space 40960K, 50% used [0x00000000fc400000,0x00000000fd800010,0x00000000fec00000)
Metaspace       used 3494K, capacity 4498K, committed 4864K, reserved 1056768K
 class space    used 387K, capacity 390K, committed 512K, reserved 1048576K

Process finished with exit code 0
```

20m的对象, 直接存在了 Old Gen区;

## 为对象分配内存: TLAB

# 作用

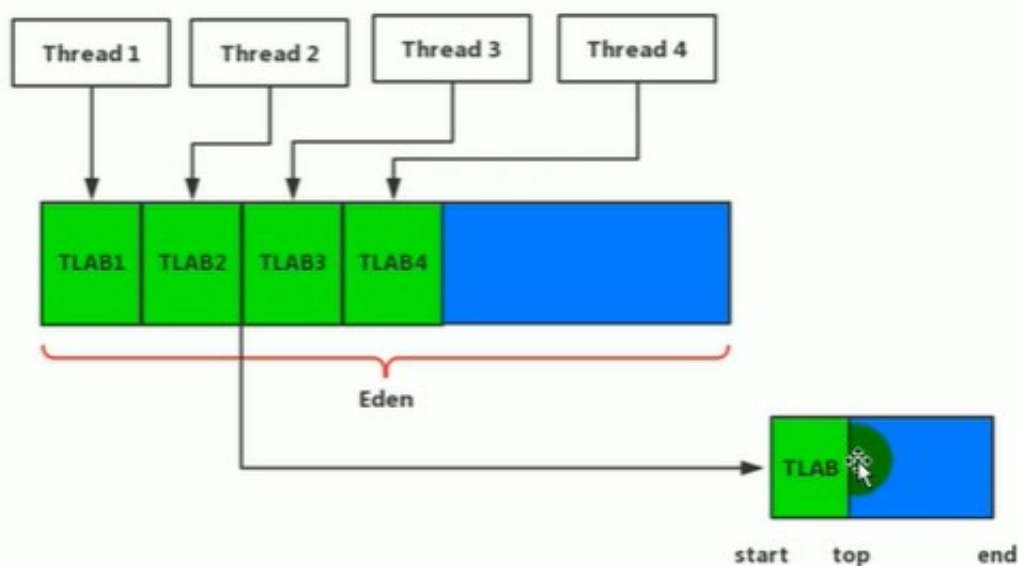
为什么有 TLAB (Thread Local Allocation Buffer)?

- 堆区是线程共享区域, 任何线程都可以访问到堆区中的共享数据;
- 由于对象实例的创建在JVM中非常频繁, 因此在并发环境下从堆区中划分内存空间是线程不安全的;
- 为避免多个线程操作同一地址, 需要使用加锁等机制, 进而影响分配速度;

# 解释

什么是 TLAB?

- 从内存模型而不是垃圾收集的角度, 堆Eden区域进行划分, JVM为每个线程分配了一个私有缓存区域, 它包含在Eden空间内;
- 多线程同时分配内存时, 使用 TLAB 可以避免一系列的非线程安全问题, 同时还能够提升内存分配的吞吐量, 因此我们可以将这种内存分配方式称之为快速分配策略;
- 几乎所有OpenJDK衍生出来的JVM都提供了TLAB的设计;



# 说明

TLAB的在说明:

- 尽管不是所有的对象实例都能够在TLAB中成功分配内存, 但JVM确实是将TLAB作为内存分配的首选;
- 在程序中,开发人员可以通过选项 "-XX:UseTLAB" 设置是否开启TLAB空间; **默认情况开启**
- 默认情况下, TLAB空间的内存非常小, 仅占有整个Eden空间的1%, 当然我们可以通过选项 "-XX:TLABWasteTargetPercent"设置TLAB空间所占有Eden空间的百分比大小
- 一旦对象在TLAB空间分配内存失败时, JVM就会尝试着通过使用加锁机制确保数据操作的原子性, 从而直接在Eden空间中分配内存;

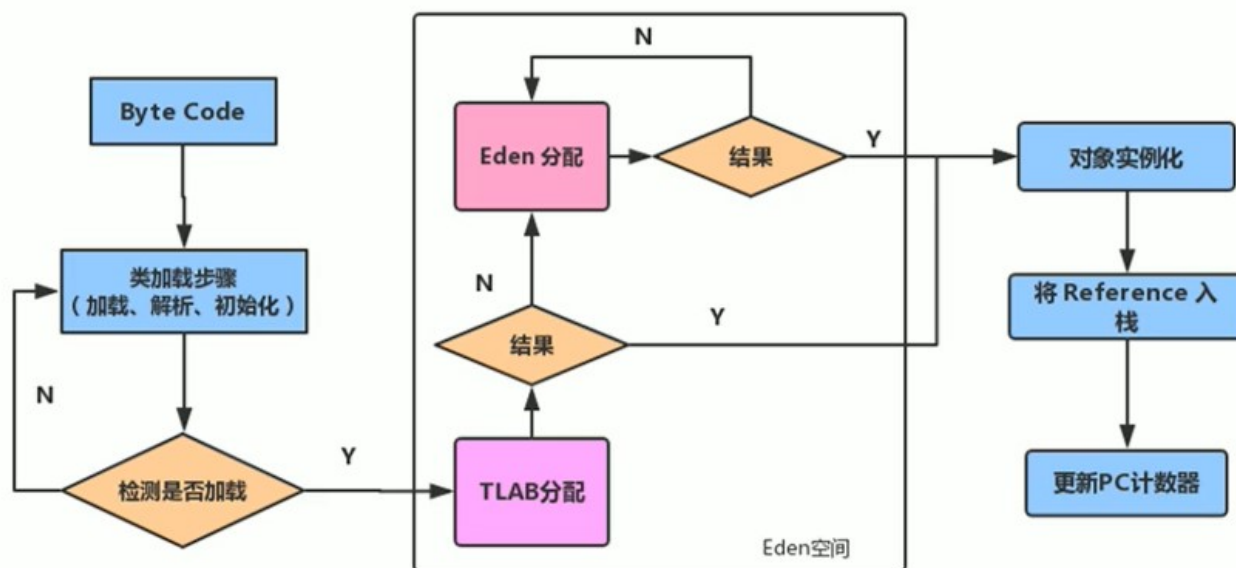
```
1  /**
2   * 测试-XX:UseTLAB参数是否开启的情况:默认情况是开启的
3   *
4   * @author shkstart  shkstart@126.com
5   * @create 2020  16:16
6   */
7  public class TLABArgsTest {
8      public static void main(String[] args) {
9          System.out.println("我只是来打个酱油~");
10         try {
11             Thread.sleep(1000000);
12         } catch (InterruptedException e) {
13             e.printStackTrace();
14         }
15     }
16 }
```

```
C:\Users\Administrator>jps
6276 TLABArgsTest
3800 Jps
4440
5816 KotlinCompileDaemon
4060 Launcher

C:\Users\Administrator>jinfo -flag UseTLAB 6276
-XX:+UseTLAB

C:\Users\Administrator>
```

## 整体上TLAB图示



## 小结堆空间的参数设置

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>

测试堆空间常用的jvm参数：

-XX:+PrintFlagsInitial：查看所有的参数的默认初始值

-XX:+PrintFlagsFinal：查看所有的参数的最终值（可能会存在修改，不再是初始值）

具体查看某个参数的指令：jps：查看当前运行中的进程

jinfo -flag SurvivorRatio 进程id

-Xms：初始堆空间内存（默认为物理内存的1/64）

-Xmx：最大堆空间内存（默认为物理内存的1/4）

-Xmn：设置新生代的大小。（初始值及最大值）

-XX:NewRatio：配置新生代与老年代在堆结构的占比

-XX:SurvivorRatio：设置新生代中Eden和S0/S1空间的比例

新生代过大, minor GC就失去了意义

新生代过小, minor GC频率太高

-XX:MaxTenuringThreshold：设置新生代垃圾的最大年龄

-XX:+PrintGCDetails: 输出详细的GC处理日志

打印gc简要信息: ① -XX:+PrintGC ② -verbose:gc

-XX:HandlePromotionFailure: 是否设置空间分配担保

```
1  /**
2   * 测试堆空间常用的jvm参数:
3   * -XX:+PrintFlagsInitial : 查看所有的参数的默认初始值
4   * -XX:+PrintFlagsFinal : 查看所有的参数的最终值 (可能会存在修改, 不再是初始值)
5   * 具体查看某个参数的指令: jps: 查看当前运行中的进程
6   * jinfo -flag SurvivorRatio 进程id
7   *
8   * -Xms: 初始堆空间内存 (默认为物理内存的1/64)
9   * -Xmx: 最大堆空间内存 (默认为物理内存的1/4)
10  * -Xmn: 设置新生代的大小。(初始值及最大值)
11  * -XX:NewRatio: 配置新生代与老年代在堆结构的占比
12  * -XX:SurvivorRatio: 设置新生代中Eden和S0/S1空间的比例
13  * -XX:MaxTenuringThreshold: 设置新生代垃圾的最大年龄
14  * -XX:+PrintGCDetails: 输出详细的GC处理日志
15  * 打印gc简要信息: ① -XX:+PrintGC ② -verbose:gc
16  * -XX:HandlePromotionFailure: 是否设置空间分配担保
17  *
18  * @author shkstart shkstart@126.com
19  * @create 2020 17:18
20  */
21 public class HeapArgsTest {
22     public static void main(String[] args) {
23
24     }
25 }
```

0001 StartAttachListener	= false	{product}
intx StarvationMonitorInterval	= 200	{product}
bool StressLdcRewrite	= false	{product}
uintx StringDeduplicationAgeThreshold	= 3	{product}
uintx StringTableSize	= 60013	{product}
bool SuppressFatalErrorMessage	= false	{product}
uintx SurvivorPadding	= 3	{product}
uintx SurvivorRatio	:= 5	{product}
intx SuspendRetryCount	= 50	{product}

冒号重新赋值



## 空间分配担保

在发生Minor GC之前, 虚拟机会检查老年代最大可用的连续空间是否大于新生代所有对象的总空间

如果大于, 则此次Minor GC是安全的

如果小于, 则虚拟机会查看 `-XX:HandlePromotionFailure` 设置值是否允许担保失败;

如果`-XX:HandlePromotionFailure=true`, 那么会继续检查老年代最大可用连续空间是否大于历次晋升到老年代的对象的平均大小;

如果大于, 则尝试进行一次Minor GC,但这次Minor GC依然是有风险的;

如果小于, 则改为进行一次Full GC;

如果`-XX:HandlePromotionFailure=false`, 则改为进行一次 Full GC

在JDK6 Update24之后(可以认为JDK7),

`HandlePromotionFailure`参数不会再影响到虚拟机的空间分配担保策略, 观察OpenJDK中的源码变化,

虽然源码中还定义了`HandlePromotionFailure`参数, 但是在代码中已经不会在使用它;

JDK6 Update24之后的

**规则变为只要老年代的连续空间大于新生代对象总大小 或者 历次晋升的平均大小就会进行 Minor GC, 否则将进行Full GC;**

## 逃逸分析

**堆是分配对象的唯一选择吗?**

### 解释

在《深入理解Java虚拟机》中关于Java堆内存有这样一段描述:

- 随着JIT编译期的发展与逃逸分析技术逐渐成熟, 栈上分配、标M替换优化技术将会导致一些微妙的变化, 所有的对象都分配到堆上也渐渐变得不那么“绝对”了。

- 在Java虚拟机中，对象是在Java堆中分配内存的，这是一个普遍的常识。但是，有一种特殊情况，那就是如果经过逃逸分析 (Escape Analysis)后发现，一个对象并没有逃逸出方法的话，那么就可能被优化成栈上分配。这样就无需在堆上分配内存，也无须进行垃圾回收了。这也是最常见的堆外存储技术。
- 此外，前而提到的基于OpenJDK深度定制的TaoBaoVM，其中创新的GCIH (GC invisible heap)技术实现off-heap，将生命周期较长的Java对象从heap中移至 heap外，并且GC不能管理GCIH内部的Java对象，以此达到降低GC的回收频率和提升 GC的回收效率的目的。

## 逃逸分析

- 如何将堆上的对象分配到栈，需要使用逃逸分析手段。
- 这是一种可以有效减少Java程序中同步负载和内存堆分配压力的跨函数 全局数据流分析算法。
- 通过逃逸分析，Java Hotspot编译器能够分析出一个新的对象的引用的 使用范围从而决定是否要将这个对象分配到堆上。
- 逃逸分析的基本行为就是分析对象动态作用域：

当一个对象在方法中被定义后，对象只在方法内部使用，则认为没有 发生逃逸。

当一个对象在方法中被定义后，它被外部方法所引用，则认为发生逃逸。例如作为调用参数传递到其他地方中。

```
1 // 没有发生逃逸的对象
2 public void test_methodI(){
3     X x = new X();
4     // use x
5     // ...
6     x = null;
7 }
```

**没有发生逃逸的对象，则可以分配到栈上，随着方法执行的结束，栈空间就被移除；**

```
public static StringBuffer createStringBuffer(String s1, String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb;
}
```

上述代码如果想要StringBuffer sb不逃出方法，可以这样写：

```
public static String createStringBuffer(String s1, String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb.toString();
}
```

## 逃逸分析代码示例

如何快速的判断是否发生了逃逸分析，就看new的对象实体是否有可能在方法外被调用。

```
1  /**
2   * 逃逸分析
3   *
4   * 如何快速的判断是否发生了逃逸分析，大家就看new的对象实体是否有可能在方法外被调用。
5   * @author shkstart
6   * @create 2020 下午 4:00
7   */
8  public class EscapeAnalysis {
9
10     public EscapeAnalysis obj;
11
12     /**
13     方法返回EscapeAnalysis对象，发生逃逸
14     */
15     public EscapeAnalysis getInstance(){
16         return obj == null? new EscapeAnalysis() : obj;
17     }
18     /**
19     为成员属性赋值，发生逃逸
20     */
21     public void setObj(){
```

```

22         this.obj = new EscapeAnalysis();
23     }
24     //思考：如果当前的obj引用声明为static的？仍然会发生逃逸。
25
26     /*
27     对象的作用域仅在当前方法中有效，没有发生逃逸
28     */
29     public void useEscapeAnalysis(){
30         EscapeAnalysis e = new EscapeAnalysis();
31     }
32     /*
33     引用成员变量的值，发生逃逸
34     对象本身就是来自外部的的方法；
35     */
36     public void useEscapeAnalysis1(){
37         EscapeAnalysis e = getInstance();
38         //getInstance().xxx()同样会发生逃逸
39     }
40 }

```

## 参数设置

- 在JDK 6u23版本之后，HotSpot中默认就已经开启了逃逸分析。
- 如见使用的是较早的版本，开发人员则可以通过：

选项“-XX: +DoEscapeAnalysis”显式开启逃逸分析

通过选项“-XX: +PrintEscapeAnalysis ” 查看逃逸分析的筛选结果

## 逃逸分析代码优化

使用逃逸分析，编译器可以对代码做如下优化：

一、**栈上分配**。将堆分配转化为栈分配。如果一个对象在子程序中被分配，要使指向该对象的指针永远不会逃逸，对象可能是栈分配的候选，而不是堆分配。

二、**同步省略**。如见一个对象被发现只能从一个线程被访问到，那么对于这个对象的操作可以不考虑同步。

三、 **分离对象或标量替换**。有的对象可能不需要作为一个连续的内存结构存在也可以被访问到，那么对象的部分（或全部）可以不存储在内存，而是存储在CPU寄存器中。

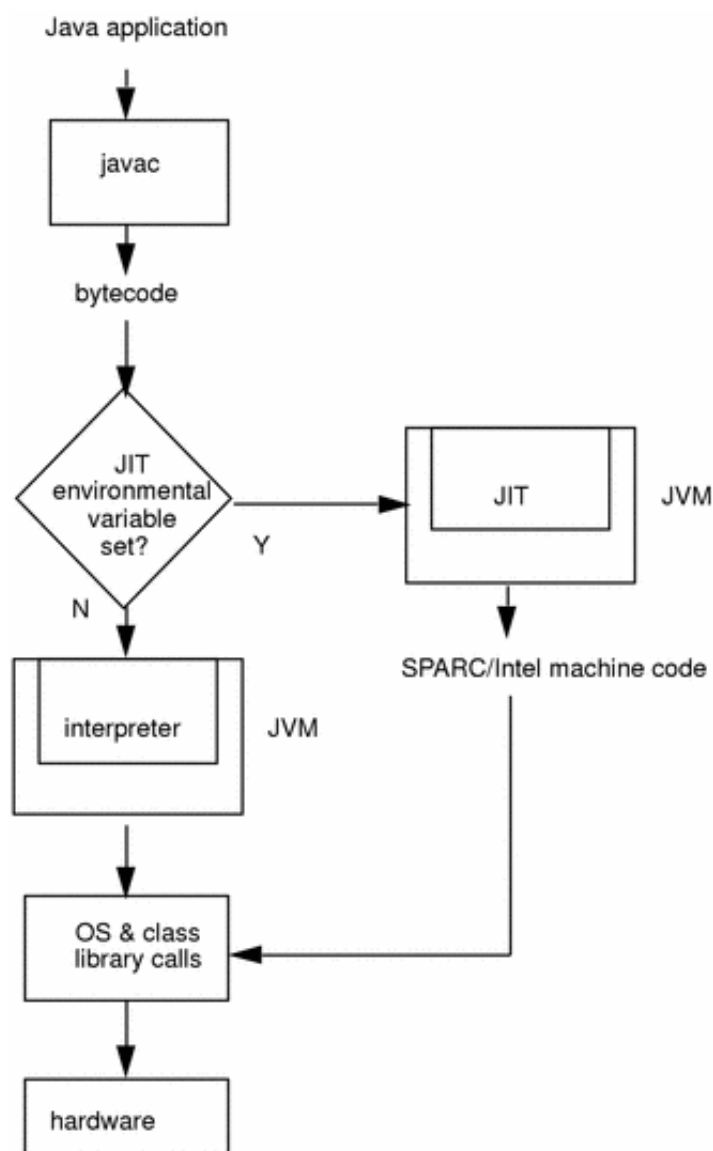
## JIT概念

JIT: Just In Time Compiler，一般翻译为即时编译器，这是是针对解释型语言而言的，而且并非虚拟机必须，是一种优化手段，Java的商用虚拟机HotSpot就有这种技术手段，Java虚拟机标准对JIT的存在没有作出任何规范，所以这是虚拟机实现的自定义优化技术。

HotSpot虚拟机的执行引擎在执行Java代码是可以采用【解释执行】和【编译执行】两种方式的，如果采用的是编译执行方式，那么就会使用到JIT，而解释执行就不会使用到JIT，所以，早期说Java是解释型语言，是没有任何问题的，而在拥有JIT的Java虚拟机环境下，说Java是解释型语言严格意义上已经不正确了。

HotSpot中的编译器是javac，他的工作是将源代码编译成字节码，这部分工作是完全独立的，完全不需要运行时参与，所以Java程序的编译是半独立的实现。有了字节码，就有解释器来进行解释执行，这是早期虚拟机的工作流程，后来，虚拟机会将执行频率高的方法或语句块通过JIT编译成本地机器码，提高了代码执行的效率，至此你已经了解了JIT在Java虚拟机中所处的地位和工作的主要内容。

### 1.JIT的工作原理图



## 工作原理

当JIT编译启用时（默认是启用的），JVM读入.class文件解释后，将其发给JIT编译器。JIT编译器将字节码编译成本机机器代码。

通常javac将程序源码编译，转换成java字节码，JVM通过解释字节码将其翻译成相应的机器指令，逐条读入，逐条解释翻译。非常显然，经过解释运行，其运行速度必定会比可运行的二进制字节码程序慢。为了提高运行速度，引入了JIT技术。

在执行时JIT会把翻译过的机器码保存起来，以备下次使用，因此从理论上来说，采用该JIT技术能够，能够接近曾经纯编译技术。

## 2.相关知识

JIT是just in time,即时编译技术。使用该技术，可以加速java程序的运行速度。

JIT并不总是奏效，不能期望JIT一定可以加速你代码运行的速度，更糟糕的是她有可能减少



代码的运行速度。这取决于你的代码结构，当然非常多情况下我们还是可以如愿以偿的。

从上面我们知道了之所以要关闭JIT `java.lang.Compiler.disable();` 是由于加快运行的速度。由于JIT对每条字节码都进行编译，造成了编译过程负担过重。为了避免这样的情况，当前的JIT仅仅对常常运行的字节码进行编译，如循环等

## 栈上分配

- JIT编译器在编译期间根据逃逸分析的结果，发现如果一个对象并没有逃逸出方法的话，就可能被优化成栈上分配。分配完成后，继续在调用栈内执行，最后线程结束，栈空间被回收，局部变量对象也被回收。这样就无须进行垃圾回收了。
- 常见的栈上分配的场景

在[逃逸分析代码示例](#)中，已经说明了。分别是给成员变量赋值、方法返回值、实例引用传递。

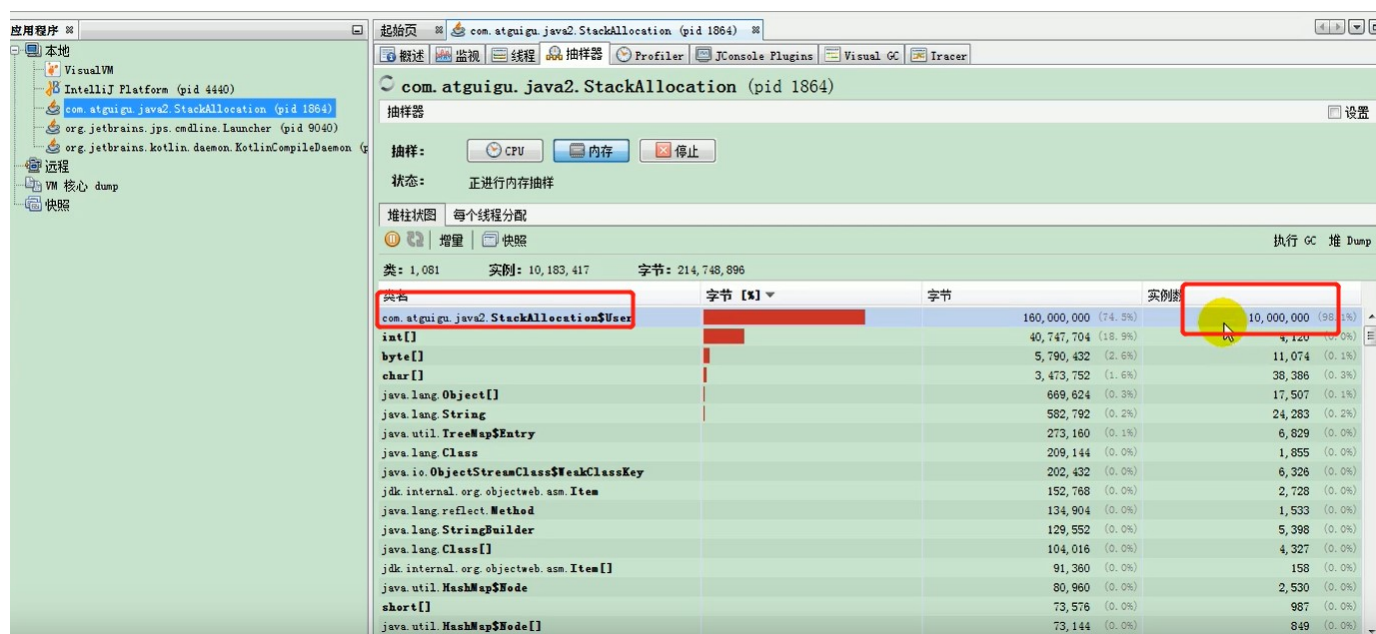
```
1  /**
2   * 栈上分配测试
3   * -Xmx1G -Xms1G -XX:-DoEscapeAnalysis -XX:+PrintGCDetails
4   * @author shkstart shkstart@126.com
5   * @create 2020 10:31
6   */
7  public class StackAllocation {
8      public static void main(String[] args) {
9          long start = System.currentTimeMillis();
10
11         for (int i = 0; i < 10000000; i++) {
12             alloc();
13         }
14         // 查看执行时间
15         long end = System.currentTimeMillis();
16         System.out.println("花费的时间为: " + (end - start) + " ms");
17         // 为了方便查看堆内存中对象个数，线程sleep
18         try {
19             Thread.sleep(1000000);
20         } catch (InterruptedException e1) {
21             e1.printStackTrace();
22         }
23     }
24 }
```

```

22     }
23 }
24
25 private static void alloc() {
26     User user = new User(); //未发生逃逸
27 }
28
29 static class User {
30
31 }
32 }

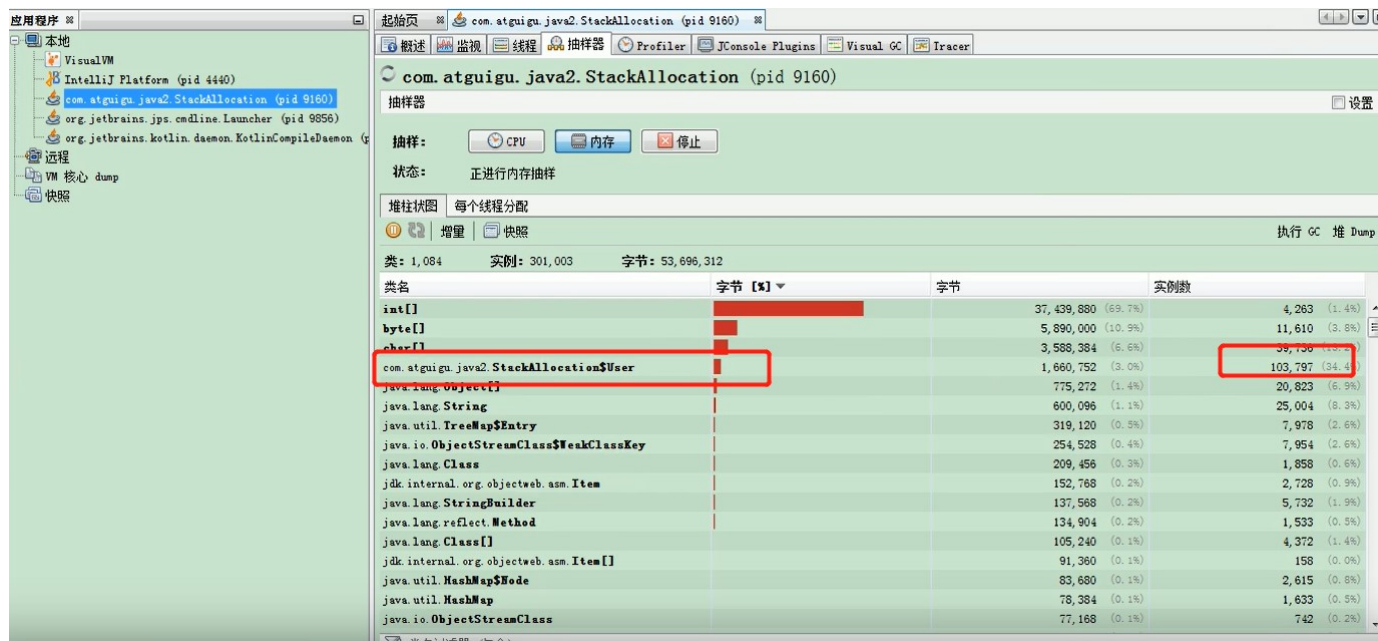
```

不开启 -XX:-DoEscapeAnalysis



75毫秒

开启 -XX:+DoEscapeAnalysis



4毫秒

甚至根本没有发生 GC, 通过打印GC日志发现, 没有发生 GC

## 同步省略

- 线程同步的代价是相当高的, 同步的后果是降低并发性和性能。
- 在动态编译同步块的时候, JIT编译器可以借助逃逸分析来判断同步块所 使用的锁对象是否只能够被一个线程访问而没有被发布到其他线程。

如果没有, 那么JIT编译器在编译这个同步块的时候就会取消对这部分代码的 同步。

这样就能大大提高并发性和性能。这个取消同步的过程就叫同步省略, 也叫锁消除。

如以下代码:

```
public void f() {
    Object hollis = new Object();
    synchronized(hollis) {
        System.out.println(hollis);
    }
}
```

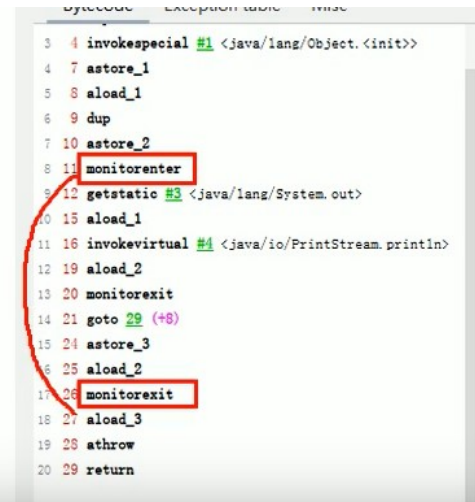
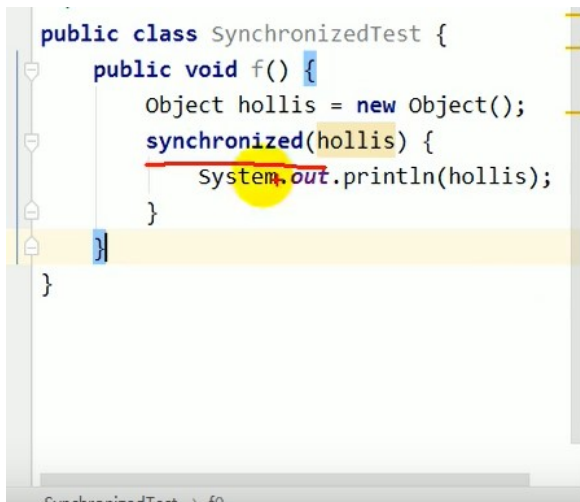
代码中对hollis这个对象进行加锁, 但是hollis对象的生命周期只在f()方法中, 并不会被其他线程所访问到, 所以在JIT编译阶段就会被优化掉。优化成:

```
public void f() {
    Object hollis = new Object();
    System.out.println(hollis);
}
```

```

1  /**
2   * 同步省略说明
3   * @author shkstart shkstart@126.com
4   * @create 2020 11:07
5   */
6  public class SynchronizedTest {
7      public void f() {
8          Object hollis = new Object();
9          synchronized(hollis) {
10             System.out.println(hollis);
11         }
12     }
13 }

```



**加载的时候, 字节码文件, 依旧可以看到 同步锁, 只是在运行的时候, 会逃逸分析**

## 标量替换

- 标量 (Scalar)是指一个无法再分解成更小的数据的数据。Java中的原始数据类型就是标量。
- 相对的, 那些还可以分解的数据叫做聚合量 (Aggregate), Java中的对象就是聚合量, 因为他可以分解成其他聚合量和标量。
- 在JIT阶段, 如果经过逃逸分析, 发现一个对象不会被外界访问的话, 那么经过JIT优化, 就会把这个对象拆解成若干个其中包含的若干个成员变量来代替。这个过程就是标量替换。

```

1 public static void main(String[] args) {
2     alloc ();
3 }
4
5 private static void alloc() {
6     Point point = new Point (1,2);
7     System.out.println("point.x="+point.x+"; point.y="+point.y);
8 }
9
10 class Point{
11     private int x;
12     private int y;
13 }

```

以上代码，经过标量替换后，就会变成:

```

1 private static void alloc() {
2     int x = 1;
3     int y = 2;
4     System.out.println("point.x="+x+"; point.y="+y);
5 }

```

可以看到，Point这个聚合量经过逃逸分析后，发现他并没有逃逸，就被替换成两个聚合量了。那么标量替换有什么好处呢？就是可以大大减少堆内存的占用。因为一旦不需要创建对象了，那么就不再需要分配堆内存了。

标量替换为栈上分配提供了很好的基础。

## 标量替换参数设置

参数-XX: +EliminateAllocations: 开启了标量替换（默认打开），允许将对象打散分配在栈上。

## 代码示例

```
1  /**
2   * 标量替换测试
3   *   -Xmx100m -Xms100m -XX:+DoEscapeAnalysis -XX:+PrintGC -XX:-EliminateAllocati
4   * @author shkstart shkstart@126.com
5   * @create 2020 12:01
6   */
7  public class ScalarReplace {
8      public static class User {
9          public int id;
10         public String name;
11     }
12
13     public static void alloc() {
14         User u = new User(); //未发生逃逸
15         u.id = 5;
16         u.name = "www.atguigu.com";
17     }
18
19     public static void main(String[] args) {
20         long start = System.currentTimeMillis();
21         for (int i = 0; i < 10000000; i++) {
22             alloc();
23         }
24         long end = System.currentTimeMillis();
25         System.out.println("花费的时间为: " + (end - start) + " ms");
26     }
27 }
28
29 /*
30 class Customer{
31     String name;
32     int id;
33     Account acct;
34
35 }
36
37 class Account{
38     double balance;
39 }
```



## 关闭 标量替换, 开启 逃逸分析 时间长了, GC多

```
ScalarReplace x
D:\developer_tools\jdk\jdk1.8.0_131\bin\java.exe ...
[GC (Allocation Failure) 25600K->736K(98304K), 0.0030894 secs]
[GC (Allocation Failure) 26336K->752K(98304K), 0.0013182 secs]
[GC (Allocation Failure) 26352K->712K(98304K), 0.0007486 secs]
[GC (Allocation Failure) 26312K->728K(98304K), 0.0007786 secs]
[GC (Allocation Failure) 26328K->792K(98304K), 0.0009105 secs]
[GC (Allocation Failure) 26392K->792K(101376K), 0.0016472 secs]
[GC (Allocation Failure) 32536K->692K(101376K), 0.0022003 secs]
[GC (Allocation Failure) 32436K->692K(100352K), 0.0003235 secs]
花费的时间为: 57 ms
```

## 开启 标量替换, 开启 逃逸分析, 时间短 没 GC

```
D:\developer_tools\jdk\jdk1.8.0_131\bin\java.exe ...
花费的时间为: 4 ms
Process finished with exit code 0
```

上述代码在主函数中进行了 1亿次alloc。调用进行对象创建，由于User对象实例需要占据约 16字节的空间，因此累计分配空间达到将近1.5GB。如果堆空间小于这个值，就必然会发生 GC。使用如下参数运行上述代码：

```
1 -server -Xmx100m -Xms100m -XX:+DoEscapeAnalysis -XX:+PrintGC -XX:+EliminateAllo
```

这里使用参数如下：

- 参数 -server: 启动Server模式，因为在Server模式下，才可以启用逃逸分析。
- 参数 -XX:+DoEscapeAnalysis: 启用逃逸分析

- 参数 -Xmx10m: 指定了堆空间最大为10MB
- 参数 -XX: +PrintGC: 将打印GC 日志。
- 参数 -XX:+EliminateAllocations: 开启了标量替换（默认打开），允许将对象打散分配在栈上，比如对象拥有id和name两个字段，那么这两个字段将会被视为两个 独立的局部变量进行分配。

## 逃逸分析小结

- 关于逃逸分析的论文在1999年就已经发表了，但直到JDK 1.6才有实现，而且这项技术到如今也并不是十分成熟的。
- 其根本原因就是无法保证逃逸分析的性能消耗一定能高于他的消耗。虽然经过逃逸分析可以做标量替换、栈上分配、和锁消除。但是逃逸分析自身也是需要进行一系列复杂的分析的，这其实也是一个相对耗时的过程。
- 一个极端的例子，就是经过逃逸分析之后，发现没有一个对象是不逃逸的。那这个逃逸分析的过程就白白浪费掉了。
- 虽然这项技术并不十分成熟，但是它也是即时编译器优化技术中一个十分重要的手段。
- 注意到有一些观点，认为通过逃逸分析，JVM会在栈上分配那些不会逃逸的对象，这在理论上是可行的，但是取决于JVM设计者的选择。据我所知，Oracle Hotspot JVM中并未这么做，这一点在逃逸分析相关的文档里已经说明，所以可以明确所有的对象实例都是创建在堆上。

-目前很多书籍还是基于JDK 7以前的版本，JDK已经发生了很大变化，intern字符串的缓存和静态变量曾经都被分配在永久代上，而永久代已经被元数据区取代。但是，intern字符串缓存和静态变量并不是被转移到元数据区，而是直接在堆上分配，所以 这一点同样符合前面一点的结论：

**对象实例都是分配在堆上。**

## 堆的小结

- 年轻代是对象的诞生、成长、消亡的区域，一个对象在这里产生、应用，最后被垃圾回收器收集、结束生命。

- 老年代放置长生命周期的对象，通常都是从Survivor区域筛选拷贝过来的Java对象。当然，也有特殊情况，我们知道普通的对象会被分配在TLAB上；如果对象较大，JVM会试图直接分配在Eden其他位置上；如果对象太大，完全无法在新生代找到足够长的连续空闲空间，JVM就会直接分配到老年代；
- 当GC只发生在年轻代中，回收年轻代对象的行为被称为MinorGC。当GC发生在老年代时则被称为MajorGC或者FullGC。一般的，MinorGC的发生频率要比MajorGC高很多，即老年代中垃圾回收发生的频率将大大低于年轻代。