# JVM调优实战

## 逃逸分析

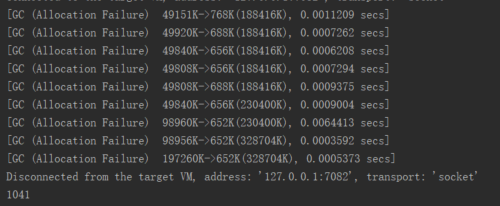


在idea的运行配置中填入：

-XX:-DoEscapeAnalysis -XX:-EliminateAllocations -XX:-UseTLAB -XX:+PrintGC

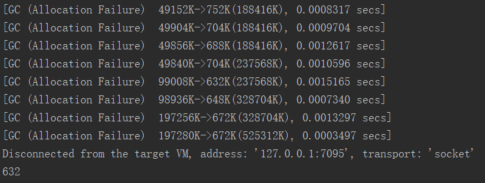
**结果分析：**

1. 无逃逸分析、无栈上分配、不使用线程本地内存：



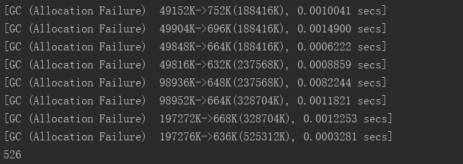
1. 使用线程本地内存，无需在eden区分配内存时加锁，效率变高

-XX:-DoEscapeAnalysis -XX:-EliminateAllocations -XX:+UseTLAB -XX:+PrintGC



1. 开启逃逸分析、使用标量替换、使用线程本地内存、效率变高

-XX:+DoEscapeAnalysis -XX:+EliminateAllocations -XX:+UseTLAB -XX:+PrintGC



注：逃逸分析是存在开销的，有时候反而不如关闭逃逸分析效率高。

## JVM参数

**内存设置：**

* -Xms 初始堆大小（超过这个值就开始垃圾回收）
* -Xmx 最大堆大小
* -Xss 线程栈大小
* -XX:NewSize=n 设置新生代大小
* -XX:NewRatio=n设置新生代和老年代的比值，如-XX:NewRatio=3，表示新生代：老年代= 1:3，新生代占整个新老年代和的1/4
* -XX:SurvivorRatio=n新生代中eden区与两个survivor区的比值，如-XX:SurvivorRatio=3，表示eden:survior =3:2，一个survivor区占整个新生代的1/5
* -XX:MaxPermSize=n 设置永久代大小

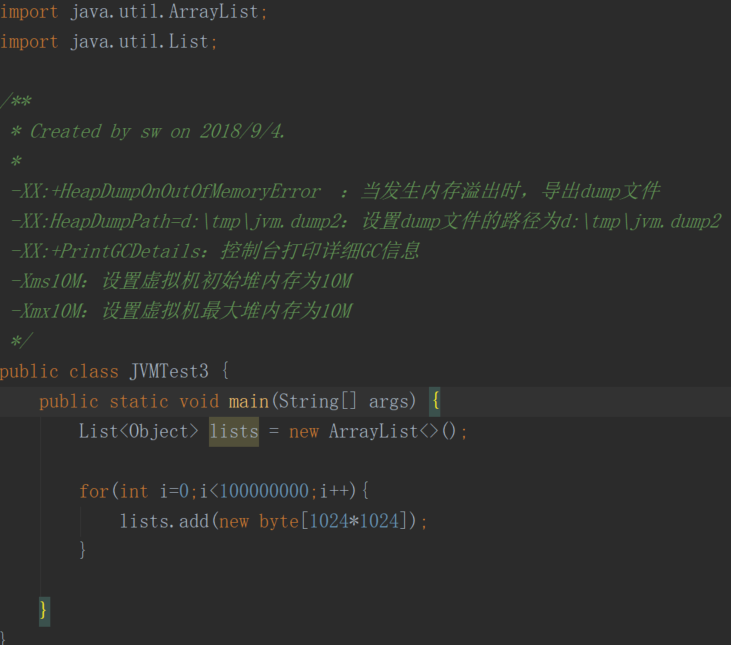
**收集器设置：**

* -XX:+UseSerialGC 设置使用串行收集器
* -XX:+UseParallelGC 设置并行收集器
* -XX:+UseConcMarkSweepGC 设置并发收集器

**GC统计信息：**

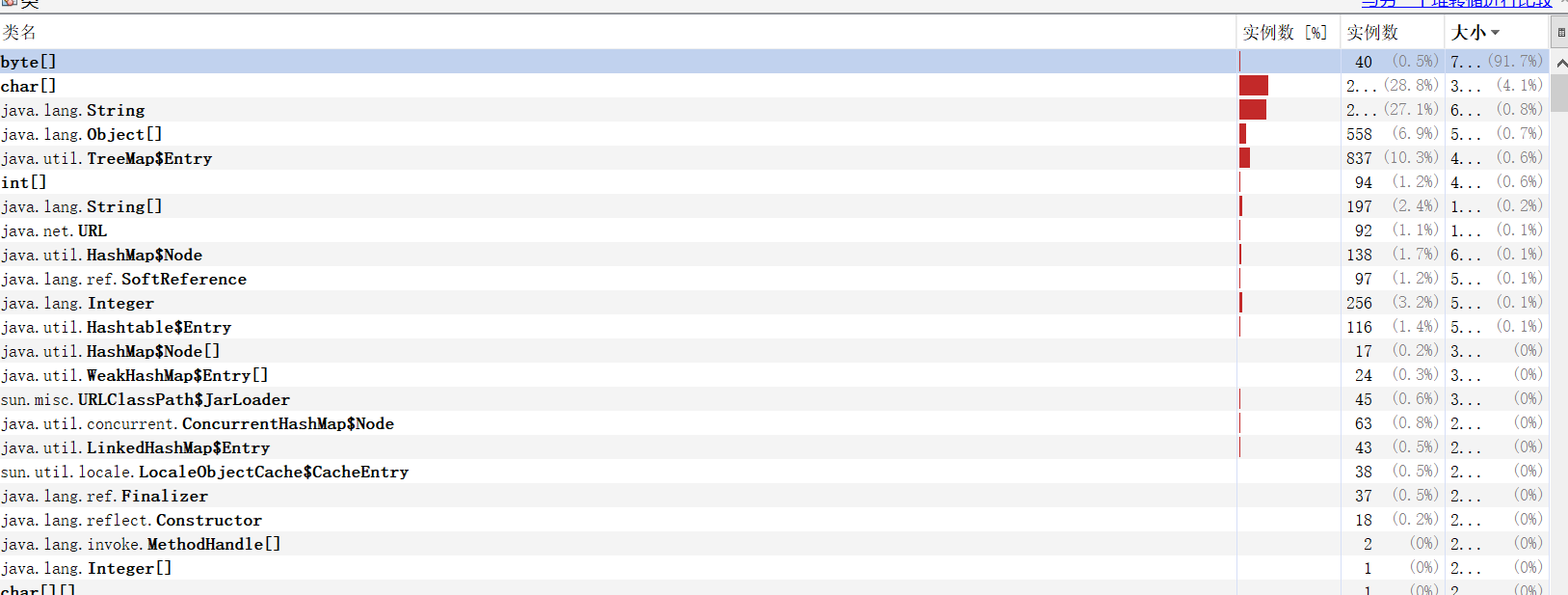
* -XX:+PrintGC 打印GC信息
* -XX:+PrintGCDetails 打印详细GC信息
* -Xloggc:filename 打印GC信息到日志文件中

## OOM实例

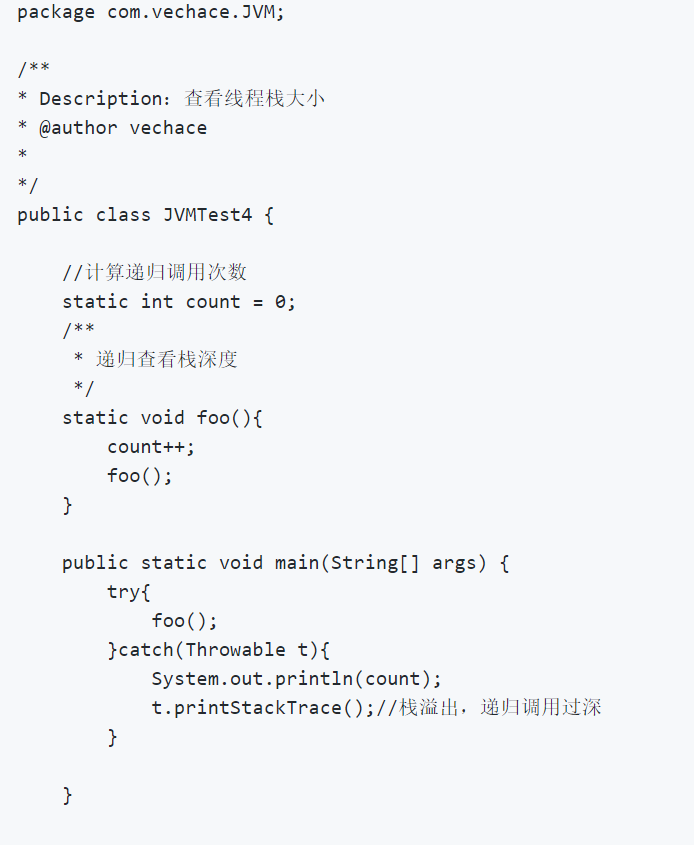


* 一般情况下，虚拟机的初始堆内存会比最大堆内存要小，而调优时往往会把初始值-Xms调至最大值-Xmx或者接近最大值,目的是减少中间的GC内存计算过程。
* 例如，设置-Xmx1G，-Xms256M，当程序运行时，虚拟机会不断地进行GC、申请新内存用于存新对象，且进行一次GC的效率较低，耗时。而直接设置-Xms1G时，初始内存开始就分配1G，与最大内存相等，程序运行时就省去了中间的内存计算及GC过程，进而提高了效率，这是调优的小技巧。

使用visualVM查看dump文件：



从图中可以看出，程序中出现了一个字节数组byte[]，占用了92.7%的内存，对应程序中的for循环，在实际开发过程中，可以通过visualVM查看内存分配情况，再回过来检查代码，找出问题所在。



 可见，递归调用次数明显增多了，在JVM调优时，-Xss也是一个非常重要的调优参数，当-Xss调的值较小时，线程的并发数就多（总内存不变，每个线程分的内存少，线程数自然变多）

 而当-Xss调的比较大，则线程递归深度就深（内存分得多，调用栈深度越深，同时线程数变少），该值属于经验值，需要结合业务来进行分析。