

堆 Heap

堆体系结构

对象在堆中的生命周期

Minor GC的过程

HotSpot虚拟机的内存管理

永久区

堆参数调优

堆溢出 OutOfMemoryError

堆体系结构

一个JVM实例只存在一个堆内存，堆内存的大小是可以调节的。类加载器读取了类文件之后，需要把类、方法、常量变量放到堆内存中，保持所有引用类型的真实信息，方便执行器执行。

其中，堆内存分为3个部分：

1. Young Generation Space, 新生区、新生代
2. Tenure Generation Space, 老年区、老年代
3. Permanent Space, 永久区、元空间

Java7之前，堆结构图如下，而Java8则只将永久代变成了元空间。

堆内存**逻辑上**分为三部分：新生+养老+永久



总结一下，堆内存存在逻辑上分为新生+养老+元空间，而堆内存存在物理上分为新生+养老。

对象在堆中的生命周期

那么如何直观的了解对象在堆中的生命周期呢？

(1) 首先，新生区是类的诞生、成长、消亡的区域。一个类在这里被创建并使用，最后被垃圾回收器收集，结束生命。

(2) 其次，所有的类都是在 `Eden Space` 被 `new` 出来的。而当 `Eden Space` 的空间用完时，程序又需要创建对象，JVM的垃圾回收器则会将 `Eden Space` 中不再被其他对象所引用的对象进行销毁，也就是垃圾回收 (Minor GC)。此时的GC可以认为是轻量级GC，又叫YangGC，简称YGC。

(3) 然后将 `Eden Space` 中剩余的未被回收的对象，移动到 `Survivor 0 Space`，以此往复，直到 `Survivor 0 Space` 也满了的时候，再对 `Survivor 0 Space` 进行垃圾回收，剩余的未被回收的对象，则再移动到 `Survivor 1 Space`。`Survivor 1 Space` 也满了的话，再移动至 `Tenure Generation Space`。

(4) 最后，如果 `Tenure Generation Space` 也满了的话，那么这个时候就会被垃圾回收 (Major GC or Full GC) 并将该区的内存清理。此时的GC可以认为是重量级GC，又称FullGC，简称FGC。如果 `Tenure Generation Space` 被GC垃圾回收之后，依旧处于占满状态的话，就会产生我们场景的 `OOM` 异常，即 `OutOfMemoryError`。

新生区 (如下是首次讲解, 简单版, 先入门大致理解, 下一页ppt详细)

新生区是类的诞生、成长、消亡的区域, 一个类在这里产生, 应用, 最后被垃圾回收器收集, 结束生命。新生区又分为两部分: 伊甸区 (Eden space) 和幸存者区 (Survivor space), 所有的类都是在伊甸区被new出来的。幸存者区有两个: 0区 (Survivor 0 space) 和1区 (Survivor 1 space)。当伊甸区的空间用完时, 程序又需要创建对象, JVM的垃圾回收器将对伊甸区进行垃圾回收 (Minor GC), 将伊甸区中的不再被其他对象所引用的对象进行销毁。然后将伊甸区中的剩余对象移动到幸存者 0区。若幸存者 0区也满了, 再对该区进行垃圾回收, 然后移动到 1区。那如果1区也满了呢? 再移动到养老区。若养老区也满了, 那么这个时候将产生MajorGC (FullGC), 进行养老区的内存清理。若养老区执行了Full GC之后发现依然无法进行对象的保存, 就会产生OOM异常 “OutOfMemoryError”。

如果出现`java.lang.OutOfMemoryError: Java heap space`异常, 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, 可以通过参数`-Xms`、`-Xmx`来调整。
- (2) 代码中创建了大量对象, 并且长时间不能被垃圾收集器收集 (存在被引用)。

Minor GC的过程

Survivor 0 Space, 幸存者0区, 也叫 from 区;

Survivor 1 Space, 幸存者1区, 也叫 to 区。

其中, from 区和 to 区的区分不是固定的, 是互相交换的, 意思是说, 在每次GC之后, 两者会进行交换, 谁空谁就是 to 区。

GC之后有交换, 谁空谁是to!



MinorGC的过程 (复制->清空->互换)

- (1) Eden Space、from 复制到 to, 年龄+1。

首先，当 **Eden Space** 满时，会触发第一次GC，把还活着的对象拷贝到 **from** 区。而当 **Eden Space** 再次触发GC时，会扫描 **Eden Space** 和 **from**，对这两个区进行垃圾回收，经过此次回收后依旧存活的对象，则直接复制到 **to** 区（如果对象的年龄已经达到老年的标准，则移动至老年代区），同时把这些对象的年龄+1。

(2) 清空 **Eden Space**、**from**

然后，清空 **Eden Space** 和 **from** 中的对象，此时的 **from** 是空的。

(3) **from** 和 **to** 互换

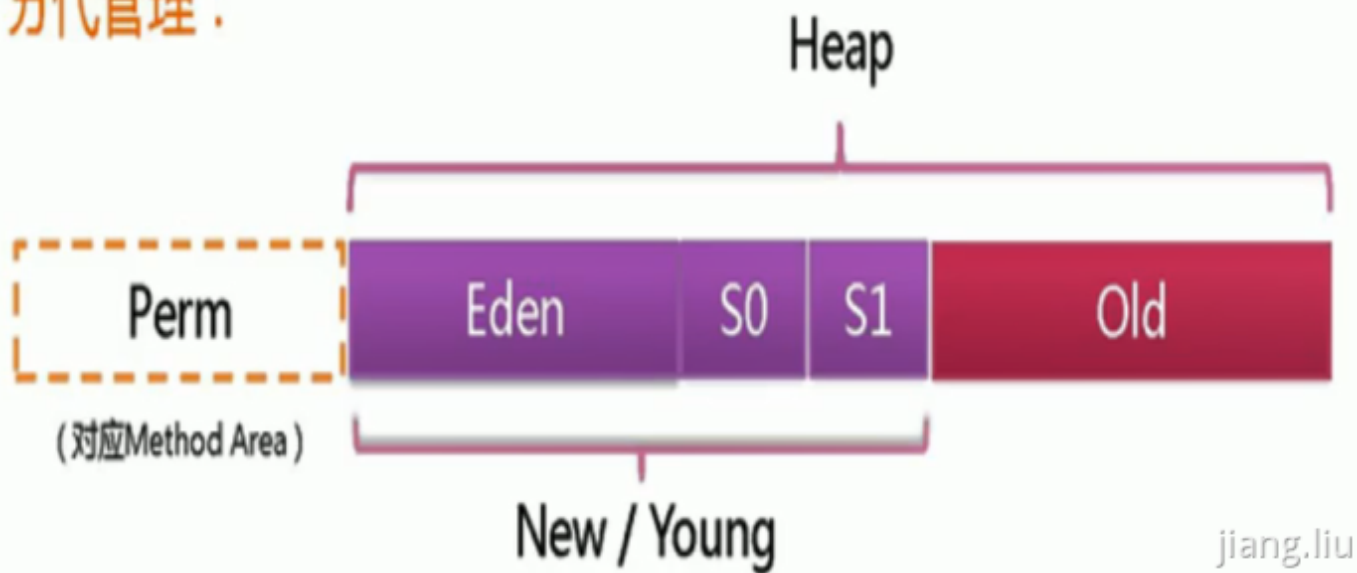
最后，**from** 和 **to** 进行互换，原 **from** 成为下一次GC时的 **to**，原 **to** 成为下一次GC时的 **from**。部分对象会在 **from** 和 **to** 中来回进行交换复制，如果交换15次（由JVM参数 **MaxTenuringThreshold** 决定，默认15），最终依旧存活的对象就会移动至老年代。

总结一句话，**GC之后有交换，谁空谁是 to**。

这样也是为了保证内存中没有碎片，所以 **Survivor 0 Space** 和 **Survivor 1 Space** 有一个要是空的。

HotSpot虚拟机的内存管理

分代管理：



不同对象的生命周期不同，其中98%的对象都是临时对象，即这些对象的生命周期大多只存在于Eden区。

实际而言，方法区（**Method Area**）和堆一样，是各个线程共享的内存区域，它用于存储虚拟机加载的：类信息+普通常量+静态常量+编译器编译后的代码等等。虽然JVM规范将方法区描述为堆的一个逻辑

辑部分，但它却还有一个别名叫做 **Non-Heap**（非堆内存），目的就是要和堆区分开。

对于HotSpot虚拟机而言，很多开发者习惯将方法区称为“永久代（**Permanent Gen**）”。但严格来说两者是不同的，或者说只是使用永久代来实现方法区而已，**永久代是方法区（可以理解为一个接口 **interface**）的一个实现**，JDK1.7的版本中，已经将原本放在永久代的字符串常量池移走。（字符串常量池，JDK1.6在方法区，JDK1.7之后一直在堆中）



如果没有明确指明，Java虚拟机的名字就叫做 **HotSpot**。


```
C:\WINDOWS\system32\cmd.exe
```

```
Microsoft Windows [版本 10.0.18363.997]  
(c) 2019 Microsoft Corporation. 保留所有权利。
```

```
C:\Users\Administrator>java -version
```

```
java version "1.8.0_91"
```

```
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
```

```
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)
```

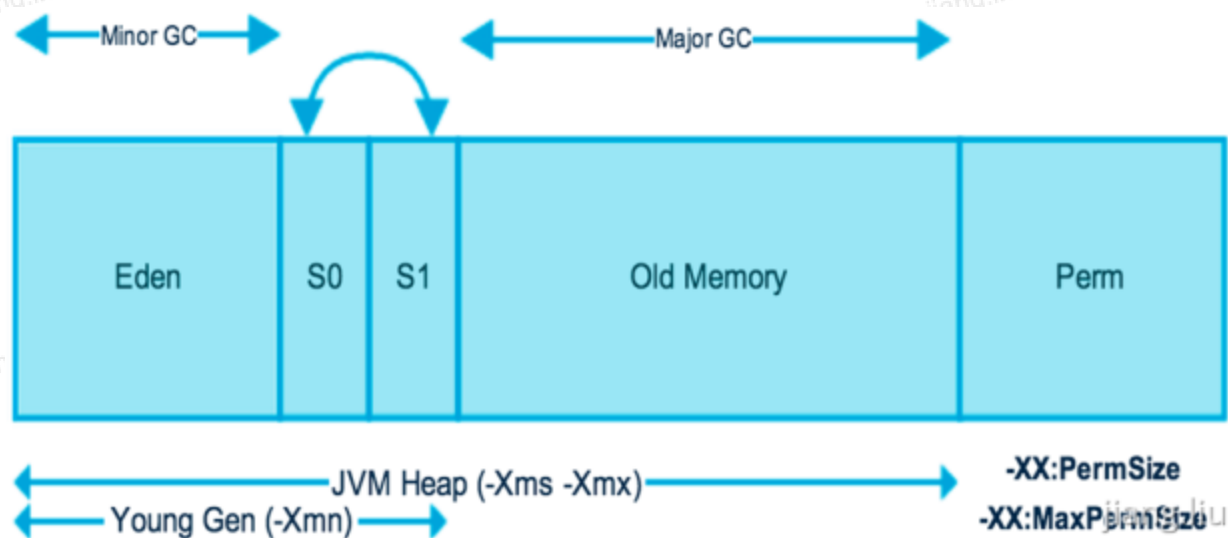
```
C:\Users\Administrator>_
```

jiang.liu

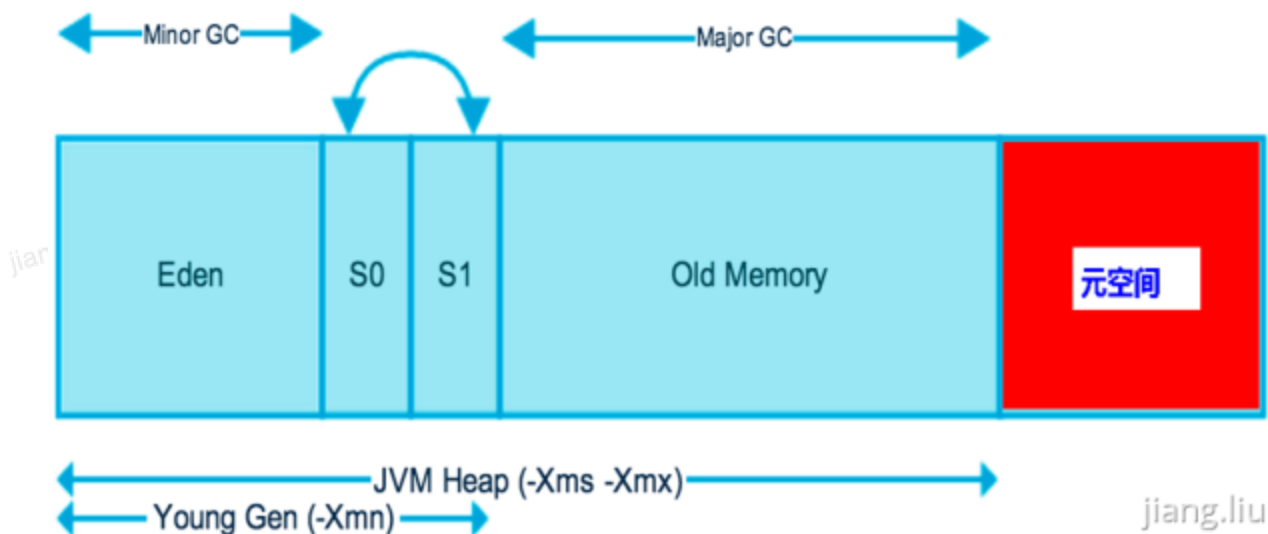
永久区

永久区是一个常驻内存区域，用于存放JDK自身所携带的 `Class`，`Interface` 的元数据（也就是上面文章提到的 `rt.jar` 等），也就是说它存储的是运行环境必须的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭JVM才会释放此区域所占用的内存。

(1) JDK1.7



(2) JDK1.8



在JDK1.8中，永久代已经被移除，被一个称为元空间的区域所取代。元空间的本质和永久代类似。

元空间与永久代之间最大的区别在于：**永久代使用的JVM的堆内存，但是java8以后的元空间并不在虚拟机中而是使用本机物理内存。**

因此，默认情况下，元空间的大小仅受本地内存限制。

类的元数据放入 `native memory`，**字符串池和类的静态变量放入Java堆中**，这样可以加载多少类的元数据就不再由 `MaxPermSize` 控制，而由系统的实际可用空间来控制。

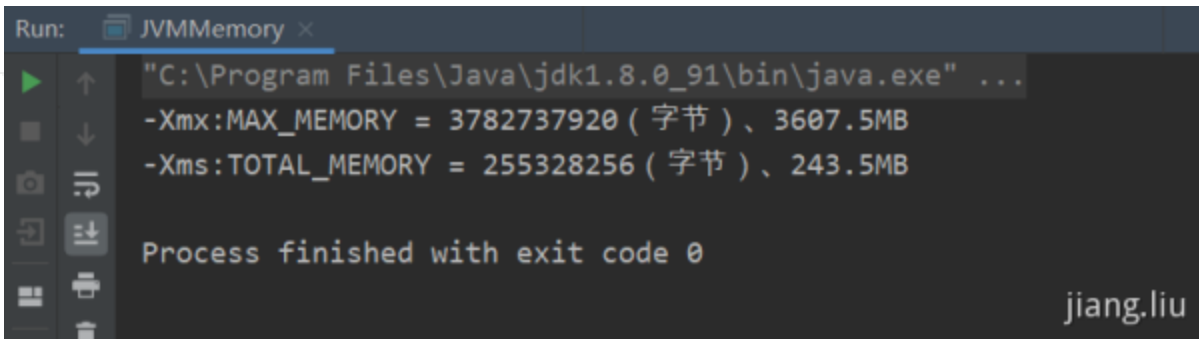
注：JDK1.8将类的静态变量从永久代移到了堆中，JDK1.7将字符串常量池移到了堆中。原本这两货都是方法区(永久代)的东西。

堆参数调优

在进行堆参数调优前，我们可以通过下面的代码来获取虚拟机的相关内存信息。

```
1 public class JVMMemory {
2     public static void main(String[] args) {
3         // 返回 Java 虚拟机试图使用的最大内存量
4         long maxMemory = Runtime.getRuntime().maxMemory();
5         System.out.println("MAX_MEMORY = " + maxMemory + " (字节)、" +
6             (maxMemory / (double) 1024 / 1024) + "MB");
7         // 返回 Java 虚拟机中的内存总量
8         long totalMemory = Runtime.getRuntime().totalMemory();
9         System.out.println("TOTAL_MEMORY = " + totalMemory + " (字节)、" +
10             (totalMemory / (double) 1024 / 1024) + "MB");
11     }
12 }
```

运行结果如下：

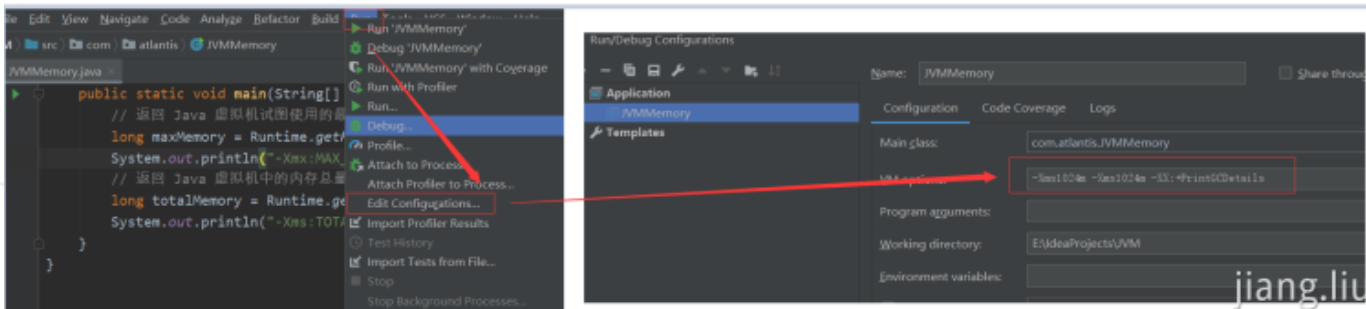


那么，这个 3607.5MB 和 243.5MB 是怎么算出来的？看下图就明白了，虚拟机最大内存为物理内存的1/4，而初始分配的内存为物理内存的1/64。

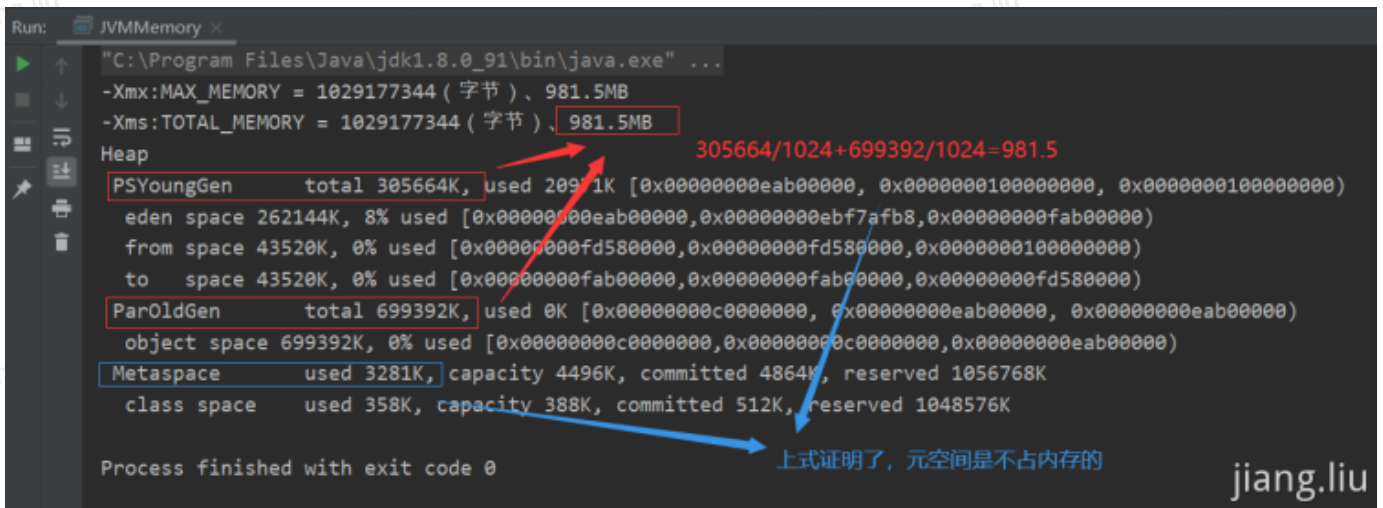
-Xms	设置初始分配大小，默认为物理内存的“1 / 64”
-Xmx	最大分配内存，默认为物理内存的 “1 / 4”
-XX:+PrintGCDetails	输出详细的GC处理日志

jiang.liu

IDEA中如何配置JVM内存参数？在【Run】->【Edit Configuration...】->【VM options】中，输入参数 -Xms1024m -Xmx1024m -XX:+PrintGCDetails，然后保存退出。



运行结果如下：



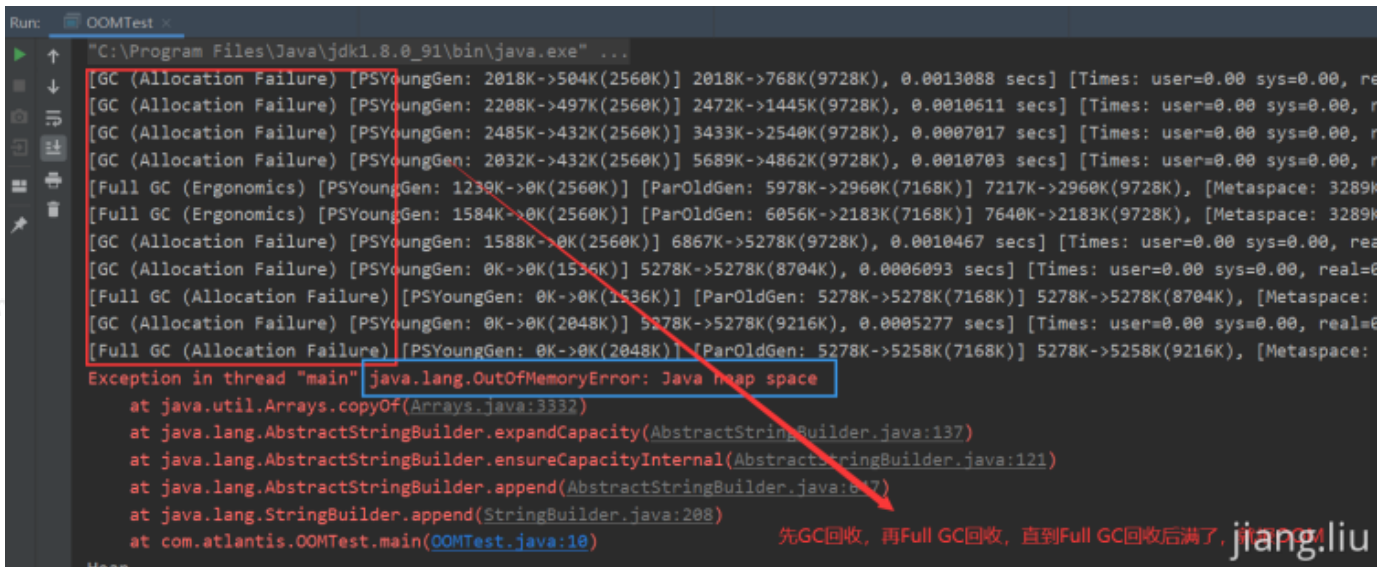
JVM的初始内存和最大内存一般怎么配?

答: 初始内存和最大内存一定是一样大, 理由是避免GC和应用程序争抢内存, 进而导致内存忽高忽低产生停顿。

堆溢出 OutOfMemoryError

现在我们来演示一下 OOM, 首先把堆内存调成10M后, 再一直new对象, 导致Full GC也无法处理, 直至撑爆堆内存, 进而导致 OOM 堆溢出错误, 程序及结果如下:

```
1 import java.util.Random;
2 public class OOMTest {
3     public static void main(String[] args) {
4         String str = "Atlantis";
5         while (true) {
6             // 每执行下面语句, 会在堆里创建新的对象
7             str += str + new Random().nextInt(88888888) + new
Random().nextInt(999999999);
8         }
9     }
10 }
```



```
Run: OOMTest
"C:\Program Files\Java\jdk1.8.0_91\bin\java.exe" ...
[GC (Allocation Failure) [PSYoungGen: 2018K->504K(2560K)] 2018K->768K(9728K), 0.0013088 secs] [Times: user=0.00 sys=0.00, real=0.0013088 secs]
[GC (Allocation Failure) [PSYoungGen: 2208K->497K(2560K)] 2472K->1445K(9728K), 0.0010611 secs] [Times: user=0.00 sys=0.00, real=0.0010611 secs]
[GC (Allocation Failure) [PSYoungGen: 2485K->432K(2560K)] 3433K->2540K(9728K), 0.0007017 secs] [Times: user=0.00 sys=0.00, real=0.0007017 secs]
[GC (Allocation Failure) [PSYoungGen: 2032K->432K(2560K)] 5689K->4862K(9728K), 0.0010703 secs] [Times: user=0.00 sys=0.00, real=0.0010703 secs]
[Full GC (Ergonomics) [PSYoungGen: 1239K->0K(2560K)] [ParOldGen: 5978K->2960K(7168K)] 7217K->2960K(9728K), [Metaspace: 3289K->3289K(11360K)] 0.0010611 secs] [Times: user=0.00 sys=0.00, real=0.0010611 secs]
[Full GC (Ergonomics) [PSYoungGen: 1584K->0K(2560K)] [ParOldGen: 6056K->2183K(7168K)] 7640K->2183K(9728K), [Metaspace: 3289K->3289K(11360K)] 0.0010611 secs] [Times: user=0.00 sys=0.00, real=0.0010611 secs]
[GC (Allocation Failure) [PSYoungGen: 1588K->0K(2560K)] 6867K->5278K(9728K), 0.0010467 secs] [Times: user=0.00 sys=0.00, real=0.0010467 secs]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(1536K)] 5278K->5278K(8704K), 0.0006093 secs] [Times: user=0.00 sys=0.00, real=0.0006093 secs]
[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(1536K)] [ParOldGen: 5278K->5278K(7168K)] 5278K->5278K(8704K), [Metaspace: 3289K->3289K(11360K)] 0.0006093 secs] [Times: user=0.00 sys=0.00, real=0.0006093 secs]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(2048K)] 5278K->5278K(9216K), 0.0005277 secs] [Times: user=0.00 sys=0.00, real=0.0005277 secs]
[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2048K)] [ParOldGen: 5278K->5258K(7168K)] 5278K->5258K(9216K), [Metaspace: 3289K->3289K(11360K)] 0.0005277 secs] [Times: user=0.00 sys=0.00, real=0.0005277 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Arrays.java:3332)
    at java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:137)
    at java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:121)
    at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:647)
    at java.lang.StringBuilder.append(StringBuilder.java:208)
    at com.atlantis.OOMTest.main(OOMTest.java:10)
```

如果出现java.lang.OutOfMemoryError: Java heap space异常，说明Java虚拟机的堆内存不够，造成堆内存溢出。原因有两点：

- ①Java虚拟机的堆内存设置太小，可以通过参数-Xms和-Xmx来调整。
- ②代码中创建了大量对象，并且长时间不能被GC回收（存在被引用）。