# JVM 调优

## 异常排查

### CPU 占用过高

引起 CPU 过高的原因大多数是由于长循环，例如在并发过高的情况下，利用 CAS 操作将导致大量空旋，导致 CPU 占用过高。

解决的方法是：

1. 通过 `top` 命令查看 CPU 占用最高的进程。
2. 通过 `top -Hp [pid]` 查看此进程中 CPU 占用最高的线程。

```
[root@VM-16-3-centos ~]# top -Hp 9869
top - 10:32:35 up 93 days, 15:57,  1 user,  load average: 0.04, 0.06, 0.06
Threads:  51 total,   0 running,  51 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.7 us,  1.3 sy,  0.0 ni, 97.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1882020 total,   107004 free,  1483680 used,   291336 buff/cache
KiB Swap:        0 total,        0 free,        0 used.   216360 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
31716 root      20   0  953488 321244   1436 S  0.3 17.1   0:00.55 nioEventLoopGro
 9869 root      20   0  953488 321244   1436 S  0.0 17.1   0:00.00 java
 9870 root      20   0  953488 321244   1436 S  0.0 17.1   0:03.57 java
```

3. 通过 `jstack [pid] | grep [xid] -A 30` 命令查看对应线程信息，注意 xid 是第二步中线程 ID 的 16 进制。


现在，可以定位到对应代码片段进行 review 排除错误，通过对 jstack 信息的查看，很显然我们是在 Main 函数中输出出现了问题，事实上也是如此，这段测试代码就是在一个 while 循环中不停的输出。

除了输出对应线程信息之外，还可以通过 `jstack -l pid > xxx.txt` 命令将对应 Java 进程的所有线程信息输出到 xxx.txt 中，然后可以通过在线网站进行分析，例如 Smart Java thread dump analyzer - thread dump analysis in seconds (fastthread.io) ，这个网站不仅能得出线程信息，还能得出是否发生死锁、是否占用 CPU 等。

## OOM 排查

这类错误通常需要排查 dump 文件，生成 dump 文件可以使用 jmap + pid 手动生成，但使用最多的还是让 Jvm 检查 OOM ERROR 自动转储 dump 文件，这需要添加如下 jvm 参数：

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=oom.hprof
```

oom.hprof 是存储 dump 信息的文件，为了更快出现 OOM，我们将堆的大小设置的小一点，添加参数 `-Xmx50m` 设置堆最大大小为 50mb 以便更快出现 OOM。

```
public static void main(String[] args) throws Exception {
    List<Object> list = new ArrayList<>();
    for (;;) {
        list.add(new HashMap<>());
```

```
        }
    }
```

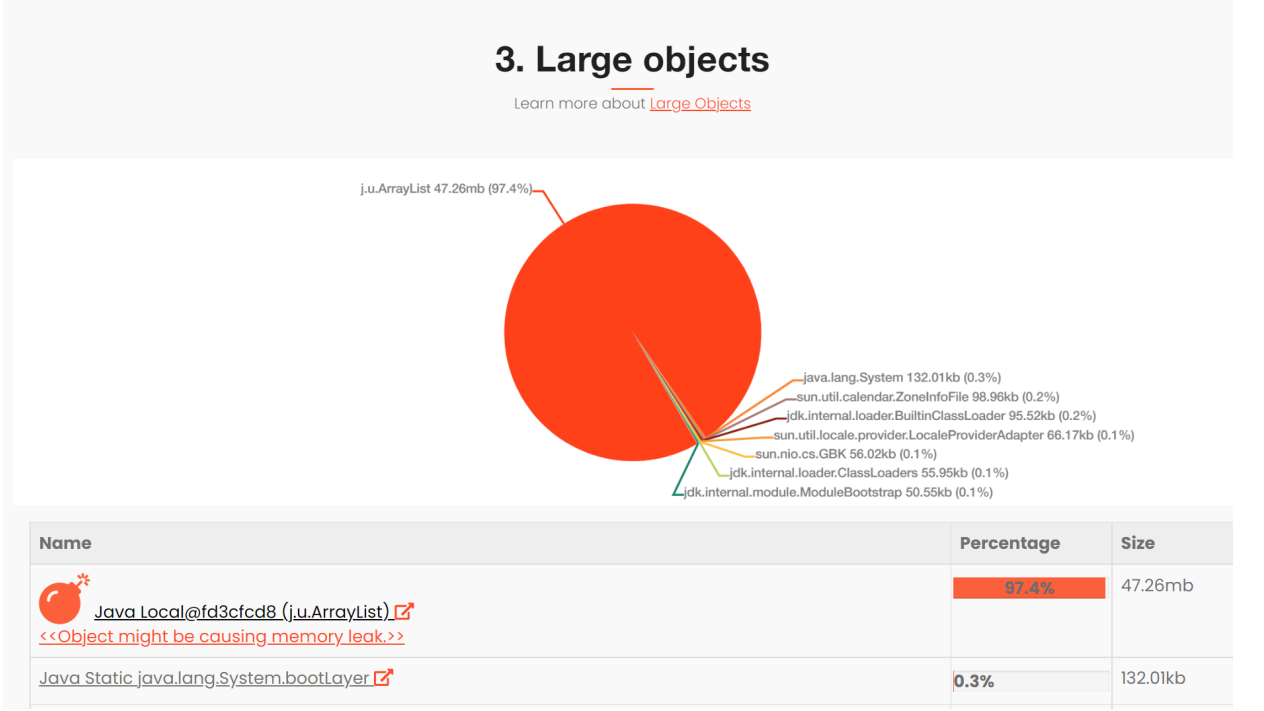
测试代码很简单，不停创建 HashMap 对象即可，运行代码：

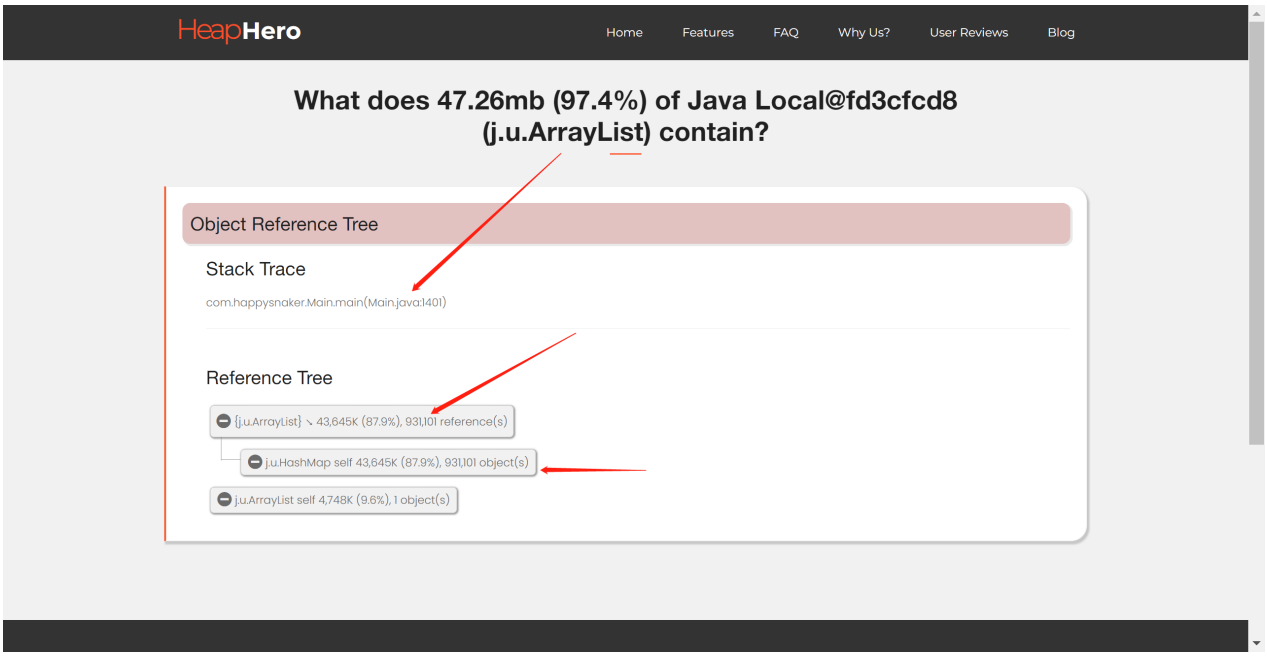
```
Dumping heap to oom.hprof ...
Heap dump file created [81166345 bytes in 0.495 secs]
Exception in thread "main" java.lang.OutOfMemoryError Create breakpoint : Java heap space
    at java.base/java.io.PrintStream.newLine(PrintStream.java:628)
    at java.base/java.io.PrintStream.println(PrintStream.java:883)
    at com.happysnaker.Main.main(Main.java:1401)
```

现在 dump 文件已经被转储，接下来得分析 dump 文件。

说实话，除非时间比较充足，否则自己去读懂 dump 文件是没有太大必要的，可以利用现有的工具进行分析，例如 jdk 自带的 Visual VM 工具，但我更喜欢在页面更加美观、操作更方便的在线网站上分析，例如 World-class heap Dump analysis - Java, Android memory dump analyzer (heaphero.io)。

将 dump 文件提交并分析，很快就能得出是哪里出错了：




# 调优实战

## 调优参数

一下列出几个重要的 JVM 参数（圈起来，要背的）：

- `-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=oom.hprof`

  表示开启 dump 文件自动转储，触发的条件是 OnOutOfMemoryError，存储的位置为 oom.hprof。

- `-XX:+PrintGCDetails` 或者 `-Xlog:gc*`

  表示打印出 GC 日志，有专门的工具对日志进行分析。

- `-Xms1024m`

  设置堆的初始大小，相当于懒加载，通常设置成等于 -Xmx。

- `-Xmx1024m`

  设置堆的最大大小，当增大此参数时，可有效防止 OOM 的发生，同时因为堆的总大小增大，GC 发生的频繁程度下降，但由于内存增大，一次 GC 的停顿时间增多。

- `-Xmn1024m`

  设置年轻代的初始大小，当减小新生代时，由于 GC 主要发生在新生代，新生代空间减小，GC 发生次数更频繁，虽然 GC 频繁，影响 JVM 性能，但都是新生代的 GC，停顿时间还是比较短的。由于新生代空间减小，可能会导致新生代提前进入老年代（条件是分代年龄和达到空间阈值），这就使得老年代可能会提前触发 GC。

- `-Xss128k`

  设置线程栈空间的大小，当程序有明显递归行为时，可适当增加比例。

- `-XX:SurvivorRatio=8`

  Eden 和 S 区的比例，大多数情况下只会调小此参数，如果程序不符合“大多数对象都是朝生夕灭”的分代假说的话，可能 S 区将无法存放存活的对象，于是将不得不借用老年代的空间，这就导致一部分新生代被移动至老年代区间中，这是违背初衷的，这将会更频繁的触发 FULL GC，这个时候需要调小此参数，但是但此参数越来越小时，E 区变得更小，新生代 GC 也会更加频繁，影响 JVM 性能。

- `-XX:MaxTenuringThreshold=10`

  设置晋升老年代的年龄条件，如果程序中对象大多数都是存活很久的对象，可以调小此参数，让对象更快的进入老年代，不然对象将持续占用新生代的空间，变相减少新生代的空间，导致 GC 频繁。

速记规则：

| 参数 | 拆分 | 含义 |
| --- | --- | --- |
| -Xms | -X、memory、size | 内存大小 |
| -Xmx | -X、memory、max | 内存最大 |
| -Xmn | -X、memory、new | 新生代内存 |
| -Xss | -X、stack、size | 栈大小 |



还有一些使用不同垃圾收集器的参数这个不需要背，它们的参数都符合 `-XX:+Use[GC_Name]` 命名规范，例如：`-XX:+UseConcMarkSweepGC` 表示使用 CMS 老年代垃圾收集器。

## 调优实战

我们使用最古老的串行化 Serial 垃圾收集器，对 JVM 添加如下参数：`-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=oom.hprof -Xmx50m -Xlog:gc* -XX:+UseSerialGC`。

测试程序如下：

```
public static void main(String[] args) throws Exception {
    class Test {
        String val;
        public Test(String _val) {
            this.val = _val;
        }
    }

    List<Object> oldList = new ArrayList<>();
    for (int i = 0; i < 1e6; i++) {
        for (int j = 0; j < 10; j++) {
            Test young = new Test("y-i == " + i);
        }
        Test old = new Test("o-i == " + i);
        oldList.add(old);
        if (i > 1e5) {
            // 移除老年代，让他们被 GC 回收
            oldList.remove(0);
        }
    }
}
```

在这个例子很简单，不停创建新生代和老年代，当老年代存活的够久时，移除它们，同时为了保证“大多数对象都是新生代”的假说，我们按照 10：1 的比例创造它们。

Total GC stats

| | |
|---|---|
| Total GC count | 54 |
| Total reclaimed bytes | 616 mb |
| Total GC time | 2 sec 90 ms |
| Avg GC time | 38.7 ms |
| GC avg time std dev | 22.7 ms |
| GC min/max time | 0 / 60.0 ms |
| GC Interval avg time | 517 ms |

Minor GC stats

| | |
|---|---|
| Minor GC count | 21 |
| Minor GC reclaimed | 245 mb |
| Minor GC total time | 320 ms |
| Minor GC avg time | 15.2 ms |
| Minor GC avg time std dev | 18.9 ms |
| Minor GC min/max time | 0 / 60.0 ms |
| Minor GC Interval avg | 17.0 ms |

Full GC stats

| | |
|---|---|
| Full GC Count | 33 |
| Full GC reclaimed | 334 mb |
| Full GC total time | 1 sec 770 ms |
| Full GC avg time | 53.6 ms |
| Full GC avg time std dev | 6.43 ms |
| Full GC min/max time | 30.0 ms / 60.0 ms |
| Full GC Interval avg | 843 ms |

在上面这个例子中，新生代内存：老年代内存 = 15 ：33.8，在这个条件下触发了 33 次 FULL GC，程序运行时能感受到明显的卡顿，现在我们考虑降低 FULL GC 次数。

想要降低 FULL GC，一般得从两个方面入手：

1. 调大新生代的大小，当新生代过小时，更多的新生代对象由于空间不够进入老年代，导致老年代压力过大，触发 FULL GC，因此调大新生代大小是一个解决方案。
2. 调大老年代的大小，这很显然，当老年代大小变大时，FULL GC 频率自然会下降，但一次会需要更多的时间。
3. 调整进入老年代的年龄限制。一方面，如果这个年龄限制过小，将会导致很多新生代进入老年代；另一方面，当限制过大时，本身为老年代的对象可能不得不存留在新生代中，导致新生代空间压力过大，这又回到了条件一。事实上，多数垃圾收集器都会动态的选择年龄限制。

你可能会觉得条件 1 和条件 2 不是冲突的嘛，的确是冲突的，这个例子告诉我们，没有一个绝对正确的决定，改变任意一个参数可能使程序变得更好，也可能使程序变的更坏，这完全取决于程序的性质，因此，像 G1 这种根据历史回收的数据启发式决定回收哪些垃圾的垃圾收集器也许是更好的选择。

回过头来分析，FULL GC 触发频繁，我们可以肯定老年代空间是不够的，但是，这究竟是由于 新生代空间小导致新生代对象进入老年代空间 还是 老年代对象本身过多 导致的，如果是前一种，我们必须调大新生代空间，而如果是后一种，我们把必须调小新生代空间以增大老年代空间，这完全取决于具体的应用程序。

仔细分析代码，程序中新生代是严格意义上的 “朝生夕灭”，young 对象被创建之后在下一个循环中就被抛弃了，因此考虑不太可能是新生代空间小导致的问题，因为存活的新生代对象非常少，GC 总是能清理掉哪些已死的对象。

因此考虑是第二种原因，尝试一下调小新生代大小：`-Xmn10m` ：

Total GC stats

| | |
|---|---|
| Total GC count | 77 |
| Total reclaimed bytes | n/a |
| Total GC time | 1 sec 280 ms |
| Avg GC time | 16.6 ms |
| GC avg time std dev | 22.5 ms |
| GC min/max time | 0 / 60.0 ms |
| GC Interval avg time | 343 ms |

Minor GC stats

| | |
|---|---|
| Minor GC count | 77 |
| Minor GC reclaimed | 579 mb |
| Minor GC total time | 1 sec 280 ms |
| Minor GC avg time | 16.6 ms |
| Minor GC avg time std dev | 22.5 ms |
| Minor GC min/max time | 0 / 60.0 ms |
| Minor GC Interval avg | 343 ms |

Full GC stats

| | |
|---|---|
| Full GC Count | 0 |
| Full GC reclaimed | n/a |
| Full GC total time | n/a |
| Full GC avg time | n/a |
| Full GC avg time std dev | n/a |
| Full GC min/max time | n/a |
| Full GC Interval avg | n/a |

完美的决策，竟然没有一次 FULL GC 的发生！由于新生代空间减小，Minor GC 发生次数增多是必然的，但由于没有 FULL GC 发生，总时间是第一次的一半不到。

来试试调大新生代大小：`-Xmn20m` ：

Total GC stats

| | |
|---|---|
| Total GC count | 52 |
| Total reclaimed bytes | 598 mb |
| Total GC time | 2 sec 200 ms |
| Avg GC time | 42.3 ms |
| GC avg time std dev | 21.9 ms |
| GC min/max time | 0 / 80.0 ms |
| GC Interval avg time | 552 ms |

Minor GC stats

| | |
|---|---|
| Minor GC count | 16 |
| Minor GC reclaimed | 225 mb |
| Minor GC total time | 300 ms |
| Minor GC avg time | 18.7 ms |
| Minor GC avg time std dev | 24.7 ms |
| Minor GC min/max time | 0 / 80.0 ms |
| Minor GC Interval avg | 19.0 ms |

Full GC stats

| | |
|---|---|
| Full GC Count | 36 |
| Full GC reclaimed | 341 mb |
| Full GC total time | 1 sec 900 ms |
| Full GC avg time | 52.8 ms |
| Full GC avg time std dev | 8.03 ms |
| Full GC min/max time | 30.0 ms / 80.0 ms |
| Full GC Interval avg | 795 ms |

这导致 FULL GC 增多了！FULL GC 次数增多 15 次，总的 FULL GC 时间相较于第一次增加 130ms 了，停顿的时间增加了，但执行的越频繁，平均停顿时间会相对少一点，这里平均停顿时间比第一次少了 1ms 左右。

我们试着调大一下最大堆大小：`-Xmx75m`：

Total GC stats

| | |
|---|---|
| Total GC count | 30 |
| Total reclaimed bytes | n/a |
| Total GC time | 480 ms |
| Avg GC time | 16.0 ms |
| GC avg time std dev | 16.0 ms |
| GC min/max time | 0 / 70.0 ms |
| GC Interval avg time | 1 sec 104 ms |

Minor GC stats

| | |
|---|---|
| Minor GC count | 30 |
| Minor GC reclaimed | 569 mb |
| Minor GC total time | 480 ms |
| Minor GC avg time | 16.0 ms |
| Minor GC avg time std dev | 16.0 ms |
| Minor GC min/max time | 0 / 70.0 ms |
| Minor GC Interval avg | 1 sec 104 ms |

Full GC stats

| | |
|---|---|
| Full GC Count | 0 |
| Full GC reclaimed | n/a |
| Full GC total time | n/a |
| Full GC avg time | n/a |
| Full GC avg time std dev | n/a |
| Full GC min/max time | n/a |
| Full GC Interval avg | n/a |

GC Pause Statistics

| | |
|---|---|
| Pause Count | 30 |
| Pause total time | 480 ms |
| Pause avg time | 16.0 ms |
| Pause avg time std dev | 0.0 |
| Pause min/max time | 0 / 70.0 ms |

这非常优秀，看来调大堆大小是个不错的选择，当然，毕竟“钞能力”嘛，但是不是内存越大越好呢？

我们来试试：`-Xmx512m`：

Total GC stats

| | |
|---|---|
| Total GC count | 9 |
| Total reclaimed bytes | n/a |
| Total GC time | 290 ms |
| Avg GC time | 32.2 ms |
| GC avg time std dev | 7.86 ms |
| GC min/max time | 20.0 ms / 50.0 ms |
| GC Interval avg time | 3 sec 88 ms |

Minor GC stats

| | |
|---|---|
| Minor GC count | 9 |
| Minor GC reclaimed | 539 mb |
| Minor GC total time | 290 ms |
| Minor GC avg time | 32.2 ms |
| Minor GC avg time std dev | 7.86 ms |
| Minor GC min/max time | 20.0 ms / 50.0 ms |
| Minor GC Interval avg | 3 sec 88 ms |

Full GC stats

| | |
|---|---|
| Full GC Count | 0 |
| Full GC reclaimed | n/a |
| Full GC total time | n/a |
| Full GC avg time | n/a |
| Full GC avg time std dev | n/a |
| Full GC min/max time | n/a |
| Full GC Interval avg | n/a |

的确，总时间变小了，但你发现了吗，这次实验一次 GC 的平均时间是最多的，如果这是 FULL GC，这意味着一次停顿的时间是最大的，这是一个弊端。

这真是矛盾啊，例如你想降低一次 FULL GC 的停顿时间，增加 FULL GC 的执行次数可以有效的降低时间，毕竟时间等于路程除以速度，于是你想让 FULL GC 执行的更频繁些：

- 减少老年代空间大小似乎达到这个目的，毕竟空间小了，GC 必然要频繁点。
- 但是减少老年代空间的同时又增大了新生代的空间，一旦新生代空间增大，对象可能在新生代就被回收了，FULL GC 触发次数会变少，当然这是我们乐于接受的结果；但另一方面，一旦新生代空间增大，触发 Minor GC 的次数就少了，别忘了，FULL GC 的触发某些情况下是依赖于 Minor GC 的，当 Minor GC 的次数少了，FULL GC 次数自然也少了。

JVM 调优就是如此矛盾，没有一个绝对的最优解，必须要根据你的程序选择恰当的参数。

索性对于一些明显极端的参数我们可以进行调整，例如堆空间或新生代过小，官方也推出了 G1 这种启发式的垃圾回收器来帮我们管理。