# 调优案例

到这里，我们了解了监控工具的使用以及会调整哪些参数，知道这些都是理论基础，接下来我们理论联系实际通过一些案例的分析，具体讲述怎么进行JVM调优。

# 1 内存优化示例

**1 监控分析**

当JVM运行稳定之后，通过以下命令得到dump日志

`jmap -dump:format=b,file=/home/hadoop/dump.dat 50125`

拿到如下FullGC信息:

```
2020-02-20T17:44:01.554+0800: 3.153:
[Full GC (Ergonomics) --jvm自己进行自适应调整引发的full gc , Allocation Failure 、System.gc()
[PSYoungGen: 37887K->0K(359424K)] --新生代垃圾收集器的名称，垃圾收集前和后新生代使用量 新生代总大小
[ParOldGen: 84645K->93168K(184832K)] --老生代垃圾收集器的名称，前后老生代使用量 老生代总空间大小
122533K->93168K(544256K), --前后堆内存的使用量 堆总空间大小
[Metaspace: 3135K->3135K(1056768K)],元空间区域垃圾收集，前后元空间的使用量，元空间大小
0.0773607 secs]  --GC事件持续的时间
[Times: user=1.25 sys=0.02, real=0.07 secs] --GC线程消耗的cpu时间，GC过程中操作系统调用和系统等待事件所消耗的时间 应用程序暂停的时间
```

以上gc日志中，在发生fullGC之时，整个应用的堆占用以及GC时间。为了更加精确需多次收集，计算平均值。或者是采用耗时最长的一次FullGC来进行估算。上图中，老年代空间占用在93168kb（约93MB），以此定为老年代空间的活跃数据。

**2 判断**

因FullGC的时间1.25秒大于1秒，故需要进行调优。

**3 确定目标**

则其他堆空间的分配，基于以下规则来进行。

老年代的空间大小为 93MB

- java heap：参数-Xms和-Xmx，建议扩大至3-4倍FullGC后的老年代空间占用。

  93 * (3-4) = (279-372)MB ， 设置heap大小为372MB；

- 元空间：参数-XX:MetaspaceSize=N和-XX:MaxMetaspaceSize=N，建议扩大至1.2-1.5倍FullGc后的永久带空间占用。

  3135K*(1.2-1.5)=(3762-4702)K，设置元空间大小为5MB;

- 新生代：参数-Xmn，建议扩大至1-1.5倍FullGC之后的老年代空间占用。

  93M*(1-1.5)=(93-139.5)M，设置新生代大小为140MB；

  **4 调整参数**

```
java -Xms373m -Xmx373m -Xmn140m -XX:PermSize=5m -XX:MaxPermSize=5m
```

**5 对比差异**

收集FullGC日志，发现FullGC的时间0.28秒，已经小于1秒，并且频率不高。已经达到调优目标，应用到所有服务器配置。

# 2 年轻代设定示例

确定young代的大小是通过评估垃圾回收的统计信息以及观察MinorGC的消耗时间和频率，下面举例说明如何通过垃圾回收的统计信息来确定young代的大小。

尽管MinorGC消耗的时间和young代里面的存活的对象数量有直接关系，但是一般情况下，更小young代空间，更短的MinorGC时间。如果不考虑MinorGC的时间消耗，减少young代的大小会导致MinorGC变得更加频繁，由于更小的空间，用完空间会用更少的时间。同理，提高young代的大小会降低MinorGC的频率。

当测试垃圾回收数据的时候，发现MinorGC的时间太长了，正确的做法就是减少young代的空间大小。如果MinorGC太频繁了就增加young代的空间大小。

**1 监控分析**

下图是一个展示了MinorGC的例子，这个例子是运行在如下的HotSpot VM命令参数下的。

```
 -Xms6144m -Xmx6144m -Xmn2048m -XX:PermSize=96m -XX:MaxPermSize=96m -
XX:+UserParallelOldGC
```

通过以配置以下参数生成dump文件

```
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=/home/hadoop/dump/
```

得到以下GC日志

```
2019-12-21T14:40:29.564+0800: 1.280: --GC事件开始的时间，相对于jvm开始启动的间隔秒数
[GC (Allocation Failure) --区分GC 类型，触发gc原因
[PSYoungGen: 2045989K->249795K(2097152K)] --垃圾收集器名称、垃圾收集前和后新生代使用量 新生代总大小
3634533K->1838430K(6291456K),垃圾收集 前后堆内存的使用量 堆总空间大小
0.0543798 secs]  GC事件持续的时间
```

```
[Times: user=0.38 sys=0.01, real=0.05 secs] GC线程消耗的cpu时间，GC过程中操作系统调用和系统等待事件所消耗的时间 应用程序暂停的时间

2019-12-21T14:40:31.949+0800: 3.665:
[GC (Allocation Failure)
[PSYoungGen: 2047896K->247788K(2097152K)]
3655319K->1859216K(6291456K),
0.0539614 secs]
[Times: user=0.35 sys=0.01, real=0.05 secs]

2019-12-21T14:40:34.346+0800: 6.062:
[GC (Allocation Failure)
[PSYoungGen: 2045889K->248993K(2097152K)]
3677202K->1881099K(6291456K),
0.0532377 secs]
[Times: user=0.39 sys=0.01, real=0.05 secs]

2019-12-21T14:40:36.815+0800: 8.531:
[GC (Allocation Failure)
[PSYoungGen: 2047094K->247765K(2097152K)]
3696985K->1900882K(6291456K),
0.054332 secs]
[Times: user=0.37 sys=0.01, real=0.05 secs]
```

上图显示了MinorGC平均的消耗时间是0.05秒，平均的频率是2.417秒1次。当计算MinorGC的消耗时间和频率的时候，越多的数据参与计算，准确性会越高。并且应用要处于稳定运行状态下来收集MinorGC信息也是非常重要的。

下一步是比较MinorGC的平均时间和系统对延迟的要求，如果MinorGC的平均时间大于了系统的要求，减少young代的空间大小，然后继续测试，再收集数据以及重新评估。 如果MinorGC的频率大于了系统的要求，就增加young代的空间大小，然后继续测试，再收集以及重新评估。也许需要数次重复才能够让系统达到延迟要求。当你改变young代的空间大小的时候，尽量保持old代的空间大小不要改变。

**2 判断**

从上图的垃圾回收信息来看，如果应用的延迟要求是40毫秒的话，观察到的MinorGC的延迟是50毫秒，比系统的要求高出了不少。

**3 调整参数**

意味着old代的空间大小是4096M，减小young代的空间大小的10%而且要保持old代的空间大小不变，可以使用如下选项。

```
-Xms5939m -Xmx5939m -Xmn1843m -XX:PermSize=96 -XX:MaxPermSize=96 -XX:+UserParallelOldGC
```

注意的是young代的空间大小从2048M减少到1843M，整个Java堆的大小从6144M减少到5939M，两者都是减少了205m。

**4 重复调整**

无论是young的空间调大还是调小，都需要重新收集垃圾回收信息和重新计算MinorGC的平均时间和频率，以达到应用的延迟要求，可能需要几个轮回来达到这个要求。

另外一些调整young代的空间需要注意的事项：

1、old代的空间一定不能小于活动对象的大小的1.5倍。

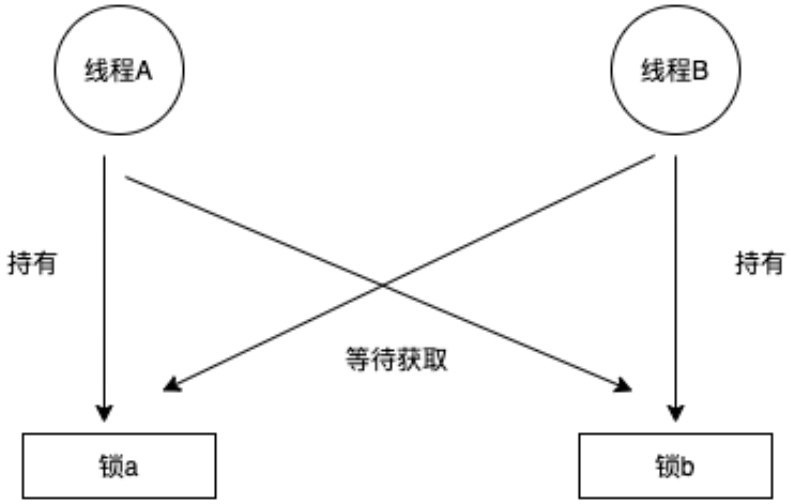
2、young代的空间至少要有Java堆大小的10%，太小的Java空间会导致过于频繁的MinorGC。

3、当提高Java堆大小的时候，不要超过JVM可以使用的物理内存大小。如果使用过多的物理内存，会导致使用交换区，这个会严重影响性能。

如果在仅仅是MinorGC导致了延迟的情况下，你无法通过调整young代的空间来满足系统的需求，那么你需要重新修改应用程序、修改JVM部署模型把应用部署到多个JVM上面（通常得要多机器了）或者重新评估系统的需求。

# 3 死锁案例

所谓死锁，是指多个进程(线程)在运行过程中因争夺资源而造成的一种僵局，当进程(线程)处于这种僵持状态时，若无外力作用，它们都将无法再向前推进。 因此我们举个例子来描述，如果此时有一个线程A，按照先锁a再获得锁b的的顺序获得锁，而在此同时又有另外一个线程B，按照先锁b再锁a的顺序获得锁。如下图所示：




## 3.1 死锁产生条件

1.互斥条件：进程(线程)要求对所分配的资源进行排它性控制，即在一段时间内某资源仅为一进程(线程)所占用。

2.请求和保持条件：当进程(线程)因请求资源而阻塞时，对已获得的资源保持不放。

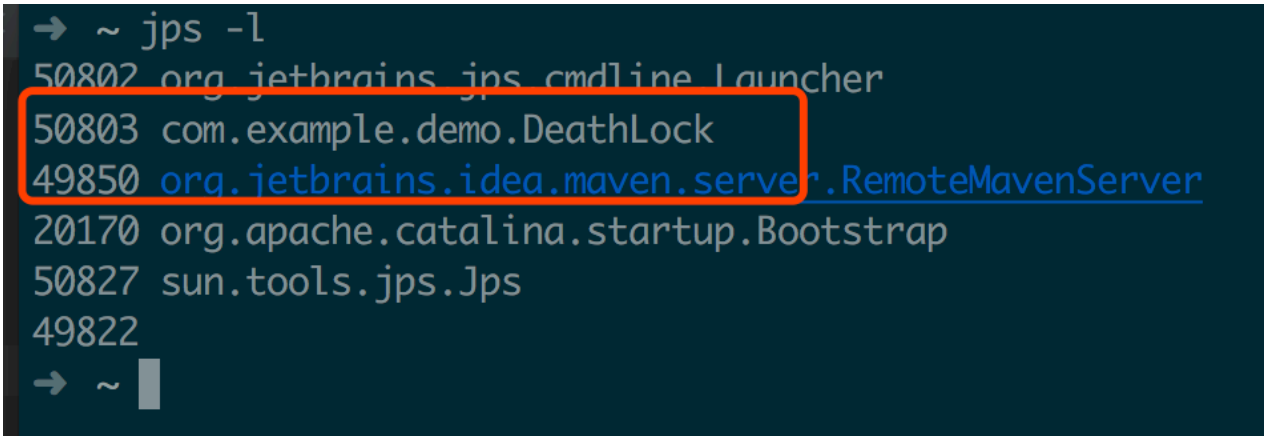
3.不剥夺条件：进程(线程)已获得的资源在未使用完之前，不能剥夺，只能在使用完时由自己释放。

4.环路等待条件：在发生死锁时，必然存在一个进程(线程)--资源的环形链。

## 3.2 源代码

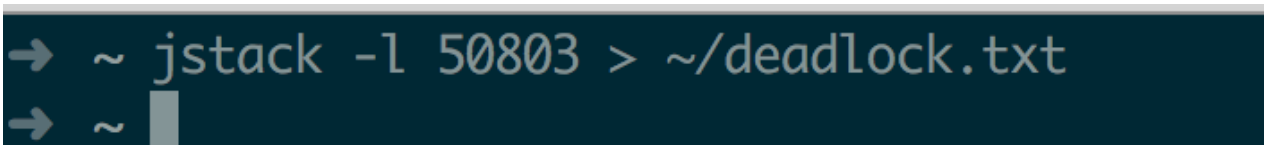
```
package com.example.demo;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class DeathLock {

    private static Lock lock1 = new ReentrantLock();
    private static Lock lock2 = new ReentrantLock();

    public static void deathLock() {
        Thread t1 = new Thread() {
            @Override
            public void run() {
                try {
                    lock1.lock();
                    TimeUnit.SECONDS.sleep(1);
                    lock2.lock();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        Thread t2 = new Thread() {
            @Override
            public void run() {
                try {
                    lock2.lock();
                    TimeUnit.SECONDS.sleep(1);
                    lock1.lock();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        t1.setName("mythread1");
        t2.setName("mythread2");
        t1.start();
        t2.start();
    }

    public static void main(String[] args) {
        deathLock();
    }
}
```

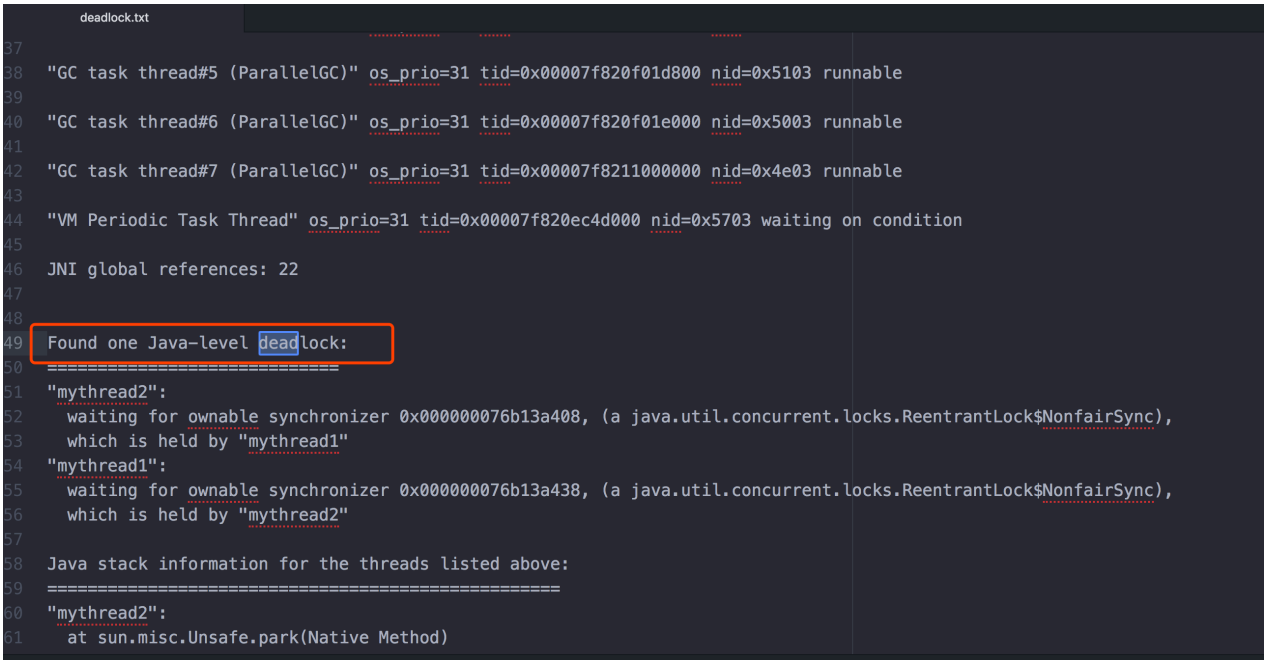
## 3.3 查看进程号




## 3.4 jstack检测

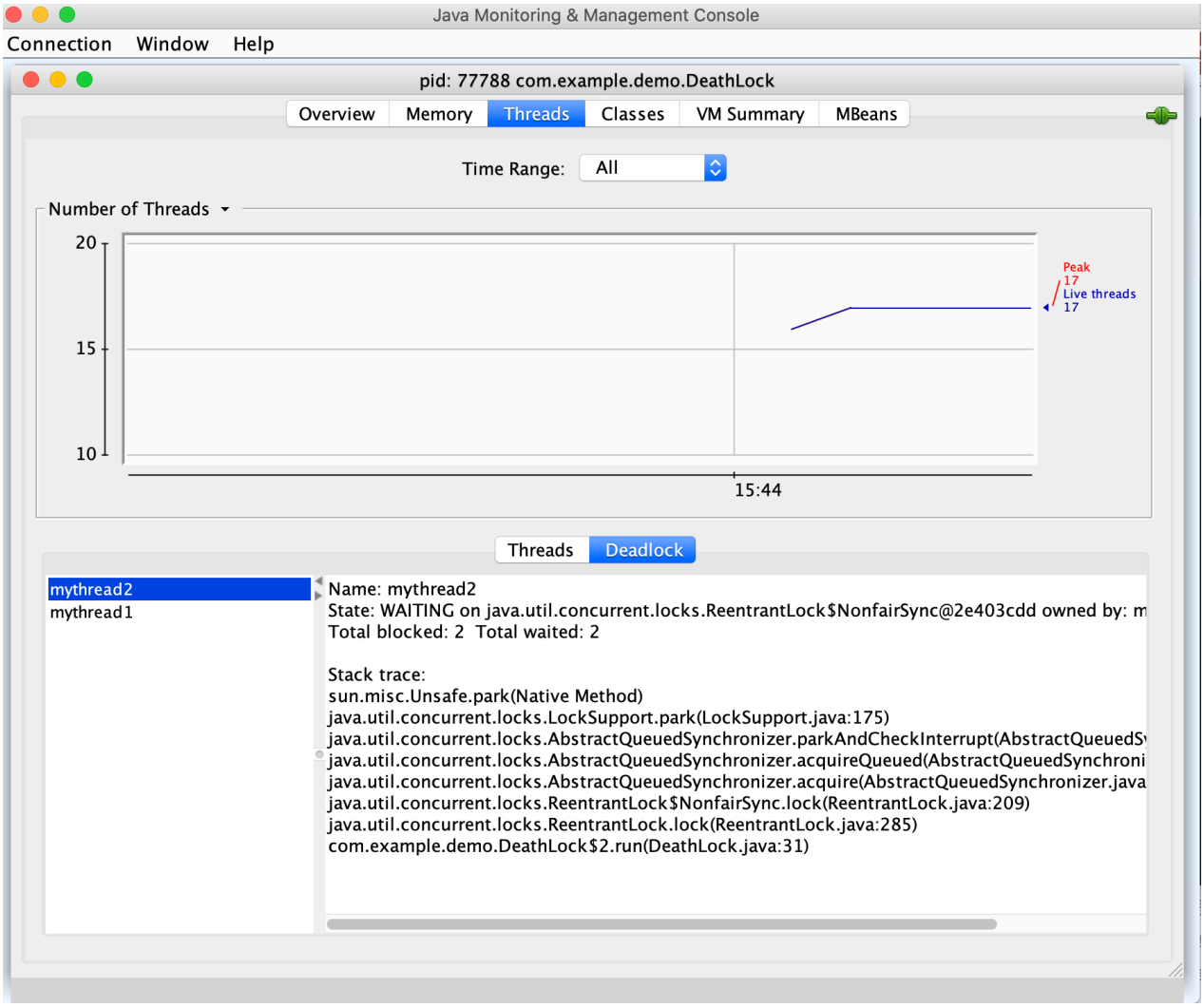
jstack是java虚拟机自带的一种堆栈跟踪工具。Jstack工具可以用于生成java虚拟机当前时刻的线程快照。线程快照是当前java虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如`线程间死锁`、`死循环`、`请求外部资源导致的长时间等待`等。 线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情，或者等待什么资源。




## 3.5 jconsole 检测

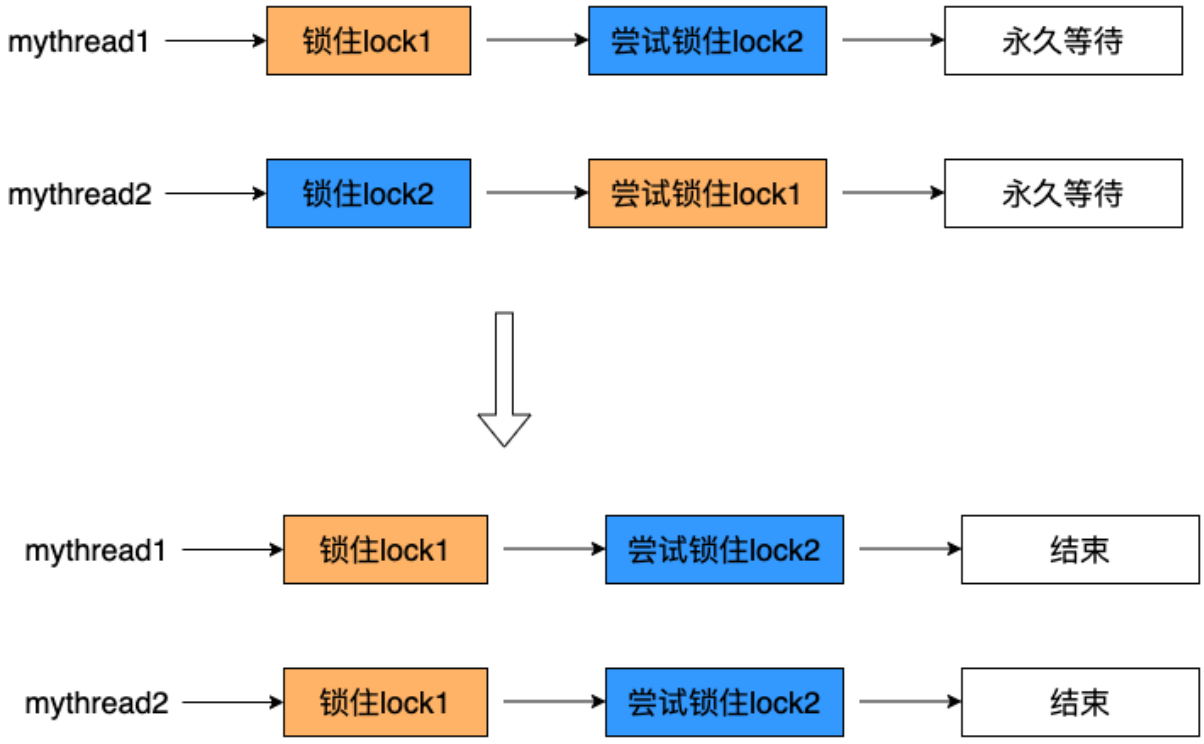
Jconsole是JDK自带的监控工具，在JDK/bin目录下可以找到。它用于连接正在运行的本地或者远程的JVM，对运行在Java应用程序的资源消耗和性能进行监控，并画出大量的图表，提供强大的可视化界面。而且本身占用的服务器内存很小，甚至可以说几乎不消耗。




## 3.6 死锁预防

### 1、以确定的顺序获得锁

如果必须获取多个锁，那么在设计的时候需要充分考虑不同线程之前获得锁的顺序。按照上面的例子，两个线程获得锁的时序图如下：




**2、超时放弃**

当使用synchronized关键词提供的内置锁时，只要线程没有获得锁，那么就会永远等待下去，然而Lock接口提供了`boolean tryLock(long time, TimeUnit unit) throws InterruptedException`方法，该方法可以按照固定时长等待锁，因此线程可以在获取锁超时以后，主动释放之前已经获得的所有的锁。通过这种方式，也可以很有效地避免死锁。 还是按照之前的例子，时序图如下：