第五章：性能调优

调优步骤
- 衡量系统现状 —— 如请求次数、响应时间、资源消耗
- 设定调优目标 —— 根据用户能接受的响应速度和拥有的系统条件觉得
- 寻找性能瓶颈 —— 找出性能差的原因，通过工具找到相关代码
- 性能调优 —— 制定计划调优
- 衡量是否达到目标 —— 没有达到目标继续寻找性能瓶颈
- 达到目标，结束调优

寻找性能瓶颈
- 表象是资源消耗过多，外部处理系统性能不足；或者程序响应速度慢
- 寻找性能瓶颈的步骤
  - 1、分析资源消耗
  - 2、通过工具查找程序中造成资源消耗的代码
- 资源消耗的主要方面
  - CPU
  - 文件/网络IO
  - 内存
- 外部处理性能不够的原因
  - 其他系统提供的功能或者数据库操作响应速度不够
  - 如sql执行速度太慢，优化sql语句
- 响应速度不够的原因 —— 代码运行效率低、程序结构不合理、未充分使用资源

CPU消耗分析
- 三个概念
  - 上下文切换
    - 抢占式调度，时间片分完了要切换线程，恢复之前线程的状态
    - 在文件、网络IO、锁等待、线程Sleep时，当前线程会进入阻塞或休眠状态，触发上下文切换
    - 切换多了会占据CPU使用，响应速度下降
  - 运行队列 —— 每个CPU都有一个运行队列，存放准备运行的线程
  - 利用率
    - CPU利用率是CPU在用户进程、内核、中断处理、IO等待及空闲五个部分使用百分比
    - 一般在60~70%、30%~35%
    - linux中可通过top或者pidstat查看进程中CPU消耗情况，对Java应用来说，us、sy是CPU消耗严重的部分
      - us：us高表示运行应用消耗了大部分CPU，要找到具体消耗CPU的线程执行的代码
        - 查找步骤：
          - 通过linux命令找到CPU消耗严重的线程
          - 通过kill -3 [javapid]或jstack的方式dump出应用的java线程信息（或者jstack [pid] | grep 'nid=0x6849'）
          - 通过第一步找到的线程id找到对应的线程，即为消耗CPU的线程
          - 采样时多执行几次

```
"Thread-1" prio=10 tid=0x706cc400 nid=0x6849 runnable [0x6fd8d000]
   java.lang.Thread.State: RUNNABLE
        at chapter6.Demo$ConsumeCPUTask.run(Demo.java:36)
        at java.lang.Thread.run(Thread.java:619)
```

子主题

        - us高的原因
          - 线程一直处于Runnable状态，即通常线程在执行循环、正则、计算或者自旋等动作
          - 频繁GC，导致响应速度下降，堆积请求增加，消耗内存更严重，导致系统不断full GC —— 通过分析JVM内存消耗查找原因
      - sy:sy高表示花费更多时间线程切换
        - 主要原因是启动的线程太多，都处于不断地阻塞（锁等待、IO等待）和执行状态之间变化，导致操作系统不断切换执行线程，产生大量上下文切换
        - 要找出一直切换状态的原因
          - kill -3 [javapid]
          - jstack -l [javapid]
          - 通过这两种方式dump出Java线程信息，查看线程状态和锁信息，找到等待状态或锁竞争过多的线程

文件IO消耗分析
- 操作文件时先将数据放入文件缓冲区，直到内存不够或者系统要释放内存给用户使用 —— 提升文件IO速度，只有在写文件和第一次读取文件时才会产生真正的文件IO
- 跟踪文件IO消耗，通过pidstat查找
  - pidstat -d -t -p [pid] 1 100
  - 通过上面命令查看线程IO消耗状况

	TGID	TID	kB_rd/s	kB_wr/s	kB_ccwr/s	Command
11:44:41	2013	-	0.00	0.00	0.00	java
11:44:42	-	2013	0.00	0.00	0.00	|__java
11:44:42	-	2014	0.00	0.00	0.00	|__java
11:44:42	-	2015	0.00	0.00	0.00	|__java
11:44:42	-	2016	0.00	0.00	0.00	|__java
11:44:42	-	2017	0.00	0.00	0.00	|__java

图 5.8 pidstat 查看线程 IO 消耗

KB_rd/s 表示每秒读取的 KB 数，KB_wr/s 表示每秒写入的 KB 数。

- 或者iostat命令查看各个设备IO历史状况

avg-cpu:	%user	%nice	%sys	%iowait	%idle
	0.04	0.00	0.02	0.01	99.93

Device:	tps	Blk_read/s	Blk_wrtn/s	Blk_read	Blk_wrtn
xvda	0.51	0.21	5.91	1044056	29019296
xvda1	0.00	0.00	0.00	1568	30
xvda2	0.13	0.15	0.99	759506	4862144
xvda3	0.00	0.00	0.00	972	0
xvda5	0.04	0.02	0.32	103354	1558984
xvda6	0.00	0.01	0.01	28546	63096
xvda7	0.58	0.03	4.59	150070	22534970

图 5.9 iostat 查看 IO 消耗

在上面的几项指标中，其中 Device 表示设备卷标名或分区名；tps 是每秒的 IO 请求数，这也是 IO 消耗情况中值得关注的数字；Blk_read/s 是指每秒读的块数量，通常块的大小为 512 字节；Blk_wrtn/s 是指每秒写的块数量；Blk_read 是指总共读取的块数量；Blk_wrtn 是指总共写入的块数量。

  - 首先关注iowait%所占百分比，当占据主要百分比时，就要关注IO消耗了
- 文件IO消耗高时，最重要是找到造成消耗高的代码
  - 通过pidstat直接找到文件IO操作多的线程
  - 结合jstack找到对应java代码，或者直接使用jstack得到线程信息分析文件IO操作多的线程
- 文件IO消耗多的主要原因是多个线程进行大量内容写入，或者磁盘处理速度慢，或者文件系统慢，或者文件太大
- 示例

执行以上代码，通过 iostat 可看到类似图 5.10 的信息：

avg-cpu:	%user	%nice	%sys	%iowait	%idle
	9.50	0.00	6.50	83.50	0.50

Device:	rrqm/s	wrqm/s	r/s	w/s	rsec/s	wsec/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	svctm	%util
xvda	0.00	27.50	0.50	9884.00	4.00	79276.00	2.00	39638.00	8.02	8206.79	805.18	0.10	100.00

avg-cpu:	%user	%nice	%sys	%iowait	%idle
	24.75	0.00	14.25	52.75	8.25

Device:	rrqm/s	wrqm/s	r/s	w/s	rsec/s	wsec/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	svctm	%util
xvda	0.00	31.50	0.00	5496.00	0.00	11444.00	0.00	5722.00	2.08	3803.63	899.55	0.13	73.50

avg-cpu:	%user	%nice	%sys	%iowait	%idle
	13.75	0.00	12.25	54.00	20.00

Device:	rrqm/s	wrqm/s	r/s	w/s	rsec/s	wsec/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	svctm	%util
xvda	0.00	253.50	0.50	6537.50	4.00	87164.00	2.00	43582.00	13.33	5329.84	467.50	0.10	65.00

图 5.10 iostat 查看示例代码的 IO 消耗

从上面可看出，iowait 占据了很多的 CPU，结合 iostat 的信息来看，主要是写的消耗，并且花在 await 的时间上要远大于 svctm 的时间。至于具体是什么动作导致了 iowait，仍然要通过对应用的线程 dump 来分析，找出其中 IO 操作相关的动作。对上面的操作进行线程的 dump，可看到如下信息：

```
"Thread-1" prio=10 tid=0x08111800 nid=0x5c97 runnable [0x9157b000..0x9157bea0]
 java.lang.Thread.State: RUNNABLE
     at java.io.FileOutputStream.writeBytes(Native Method)
     at java.io.FileOutputStream.write(FileOutputStream.java:260)
     at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:202)
```

分布式 Java 应用：基础与实践

36 第 5 章 性能调优

```
at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:263)
at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:106)
- locked <0x9338da60> (a java.io.FileWriter)
at java.io.OutputStreamWriter.write(OutputStreamWriter.java:190)
at java.io.BufferedWriter.flushBuffer(BufferedWriter.java:111)
- locked <0x9338da60> (a java.io.FileWriter)
at java.io.BufferedWriter.write(BufferedWriter.java:212)
- locked <0x9338da60> (a java.io.FileWriter)
at java.io.Writer.write(Writer.java:140)
at chapter6.fileio.demo.IOWaitHighDemo$Task.run(IOWaitHighDemo.java:59)
at java.lang.Thread.run(Thread.java:619)
```

从上面的线程堆栈中，可看到线程停留在了 FileOutputStream.writeBytes 这个 Native 方法上，这方法所做的动作为将数据写入文件中，也就是所要寻找的 IO 操作相关的动作。继续跟踪堆栈往上查找，直到查找到系统中的代码，例如上面的例子中则为 IOWaitHighDemo.java:59。

使用 pidstat 则简单很多，直接通过 pidstat 找到 IO 读写量大的线程 ID，然后结合上面的线程 dump 即可找到相应的耗文件 IO 多的动作。

网络IO消耗分析
- 采用sar分析网络IO消耗状况
- 输入sar -n FULL 1 2 —— 执行1秒为频率，共输出两次网络IO消耗情况
- 主要关注sockets上的统计信息中的：tcpsck和updsck
- 当网络IO高时，先对线程dump分析，查找产生大量网络IO的线程
- 线程特征是读取或写入网络流，将对象序列化为字节流发送或者读取字节流反序列化为对象 —— 这个过程消耗堆内存
- Java应用一般不会造成网络IO消耗严重

内存消耗分析
- 主要在堆内存消耗，通常设置Xms和Xmx相等避免申请内存
- 在创建线程和使用直接内存时会操作堆外内存，可能导致溢出
- 内存消耗导致GC频繁、CPU消耗增加、线程执行速度下降，OOM
- **可通过vmstat、sar、top、pidstat等方式查看内存消耗**

资源消耗不多，但程序执行慢的原因
- 多出现于访问量不大的情况下
- 1、锁竞争激烈
- 2、未充分使用硬件资源
- 3、数据量增加

JVM调优（为了降低GC导致的应用暂停时间）
- 内存管理方面的调优，包括堆的各个分区大小、GC垃圾回收期
- 代大小的调优
  - XMS
  - Xmx
  - Xmn —— 决定新生代空间大小
  - XX：SurvivorRatio —— 决定新生区三个分区的比例
  - XX：MaxTenuringTheshold —— 控制经历多少次Minor GC转入老年代，默认15
  - 避免新生代大小设置过小
    - 设置过小minor GC更频繁
    - 可能使minor GC对象直接进入老年代，甚至老年代内存不够导致Full GC
  - 避免新生代大小设置过大
    - 使老年代变小，Full GC频繁
    - 使minor GC耗时增加
  - 避免Survivor区过小或过大
    - 过小可能导致minor GC后的对象太大直接进入老年代，导致Full GC提前
    - 大点是好的，避免直接进入老年代，但是不能太大
  - 合理设置新生代存活周期
    - 比如设置为16，多活了一次，可能就能让对象在新生代回收
    - 缺点是会让survivor区被占用更多的内存
- GC策略调优
  - CMS GC可以认为是和应用并发进行的，确实可以减小GC给应用造成的暂停，但是也会导致吞吐量下降，因为需要减少一个线程为GC使用
  - Web应用需要一个暂停时间短的GC，CMS GC是不错的选择，G1是加强版CMS更好
- 案例： —— P200~202

程序调优
- CPU消耗严重
  - us高
    - 原因：线程无挂起动作一直执行，导致其他线程线程饿死
    - 解决：对线程动作加Thread.sleep，释放CPU执行权
  - sy高
    - 原因：
      - 线程太多需要经常切换线程
      - 线程之间锁竞争激烈，线程状态经常切换。
      - 应用中有较多的网络IO操作或确实需要锁竞争机制
    - 解决：
      - 减少线程数
      - 降低线程之间的锁竞争
      - 采用协程支持更高并发量，避免并发量上涨后CPU的sy消耗严重、系统load上涨和系统性能下降
        - 仅需要启动CPU核数的线程数和大量Task支持高并发量
        - 使线程使用率提升
        - 但是需要在堆中保存Task上下文信息，消耗更多内存
        - 并且因为协程要求所有操作都不阻塞原生线程，所以应用中不能使用同步、锁机制，还需要解决同步访问数据库、操作文件等问题
- 文件IO消耗严重
  - 原因 —— 多个线程在写大量的数据到同一文件，使文件变得很大，使写入越来越慢，各线程激烈争抢文件锁
  - 解决：
    - 同步写文件改为异步写文件，减少写文件满导致性能下降太多
    - 批量读写，提高IO操作性能
    - 限流，如出现大量异常时，统计异常的log.error执行频率，超过某一频率时一段时间内不在写入log或者塞入一个队列缓慢写
    - 减少文件大小
- 网络IO消耗严重
  - 原因：同时需要发送或接受的包太多
  - 解决：限流，限制发送packet频率
- 内存消耗严重
  - 原因
    - 消耗了过多的堆内存，GC频繁
    - 反射或者动态加载导致方法区内存压力太大
  - 解决
    - 释放不必要引用，比如ThreadLocal中存放的对象，如果没有主动释放不会被GC
    - 使用对象缓存池，创建对象实例要耗费一定CPU和内存
    - 采用合理缓存失效算法
      - 即缓存满了后采用什么样的淘汰策略
      - FIFO
      - LRU
      - LFU
      - LinkedHashMap可实现支持FIFO和LRU策略的缓存池
    - 合理使用软引用和弱引用

资源消耗不多，程序执行慢
- 原因
  - 锁竞争激烈 —— 线程多了后，很容易处于等待锁的状况，性能下降，并使CPU sy上升 —— 解决
    - 围绕减少锁竞争来解决
    - 使用并发包中的类，采用CAS自旋减少线程切换
    - 使用读写锁，读不加锁
    - 拆分锁，使用锁粒度更小的数据结构，如ConcurrentHashMap
  - 未充分发挥硬件资源
    - 并发场景下未充分使用CPU —— 解决
      - 增加线程数到CPU核心数
      - 增加线程数并发处理原单线程的计算，加快计算速度
    - 未充分使用内存