Java 性能手册

JAVA 性能问题解决之道

张赛

目 录

1 原理篇

1.1 JVM 内存管理

1.1.1 内存组成

- 对象堆 heap：

  保存 java 对象。两个配置参数 Xmx － 最大值， Xms － 初始大小。

- 永久代：

  保存元数据，比如 class 定义等

  -XX:PermSize=<>, -XX:MaxPermSize=<>

- 热编译本地代码：

  热点代码经过 JIT 编译器编译所得的本地代码

- Socket 缓存：

  发送缓存，接收缓存

  java.util.Socket.setXXXBufferSize(int)

- 线程栈：

  -Xss<>

- 直接内存镜像：

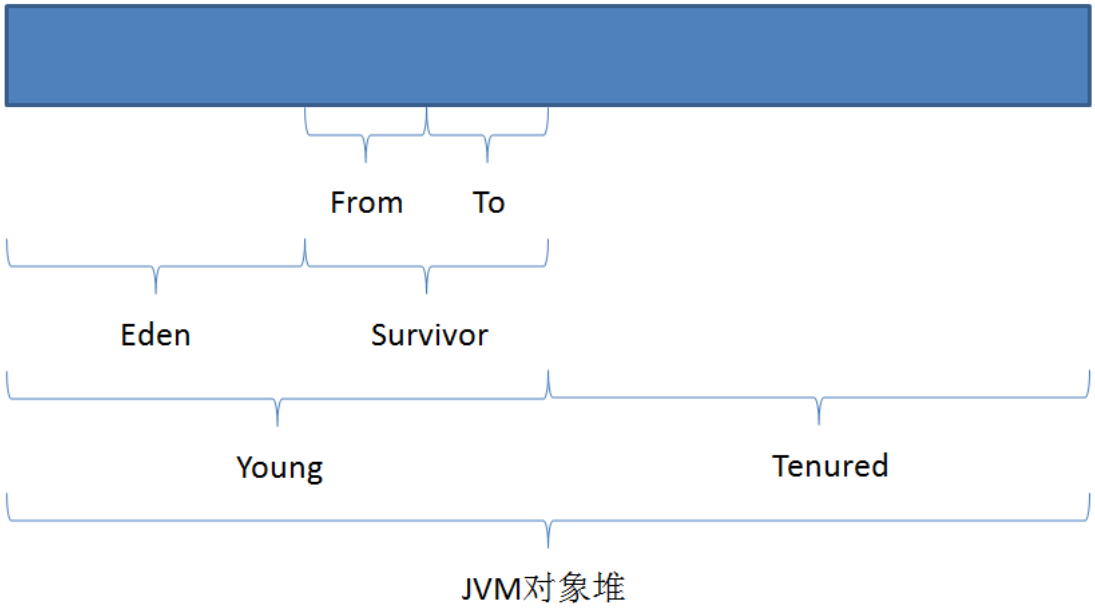
  经由 java.nio.ByteBuffer.allocateDirect(int)申请所得的内存

  -XX:MaxDirectMemorySize=<>

- 本地代码及其所申请的内存空间
- GC 所申请的内存空间

1.1.2 GC

图1-1 JVM 对象堆组成



- 1 新建对象实例保存在 Eden 区。
- 2 Eden 区发生内存分配失败时，触发 Minor GC。
- 3 Minor GC 后仍然存活的对象移到 To 区。（Eden 和 From 的都移）。对象 Age 加 1。
- 4 Minor GC 后，Survivor 区的 From、To 区角色对调。
- 5 Minor GC 时，若 To 区空间不够，则直接提升到 Tenured 代。
- 6 对象 Age 大于最大存活 Age 时，则直接提升到 Tenured 代。
- 7 Tenured 区空间不足时导致 Major GC。

使用 Jstat 工具可查看 GC 情况，详细请看 4.6 节 Jstat 工具使用。

1.1.3 参数

表1-1 性能相关参数列表

参数	注释
-server	一定要作为第一个参数，在多个 CPU 时性能佳
- Xms	初始 Heap 大小，使用的最小内存，cpu 性能高时此值应设的大一些

-Xmx	java heap 最大值，使用的最大内存。Xms、Xmx 两个值是分配 JVM 的最小和最大内存，取决于硬件物理内存的大小，建议均设为物理内存的一半。
-XX:PermSize	设定内存的永久保存区域
-XX:MaxPermSize	设定最大内存的永久保存区域
-XX:MaxNewSize	新生代占整个堆内存的最大值。
-Xss	每个线程的 Stack 大小
-verbose:gc	打印 GC 日志
-Xloggc:gc.log	指定 GC 日志文件
-Xmn	年轻代的 heap 大小，一般设置为 Xmx 的 3、4 分之一
-XX:+UseParNewGC	设置年轻代为并行收集，缩短 minor 收集的时间
-XX:+UseConcMarkSweepGC	使用 CMS 内存收集，缩短 major 收集的时间
-XX:+HeapDumpOnOutOfMemoryError	OOM 时会输出 java_pid.hprof 文件
-XX:+HeapDumpOnCtrlBreak	kill -3 时生成 HeapDump
-XX:MaxDirectMemorySize	最大本地直接内存

说明

java 启动参数共分为三类：

其一是标准参数（-），所有的 JVM 实现都必须实现这些参数的功能，而且向后兼容；

其二是非标准参数（-X），默认 jvm 实现这些参数的功能，但是并不保证所有 jvm 实现都满足，且不保证向后兼容；

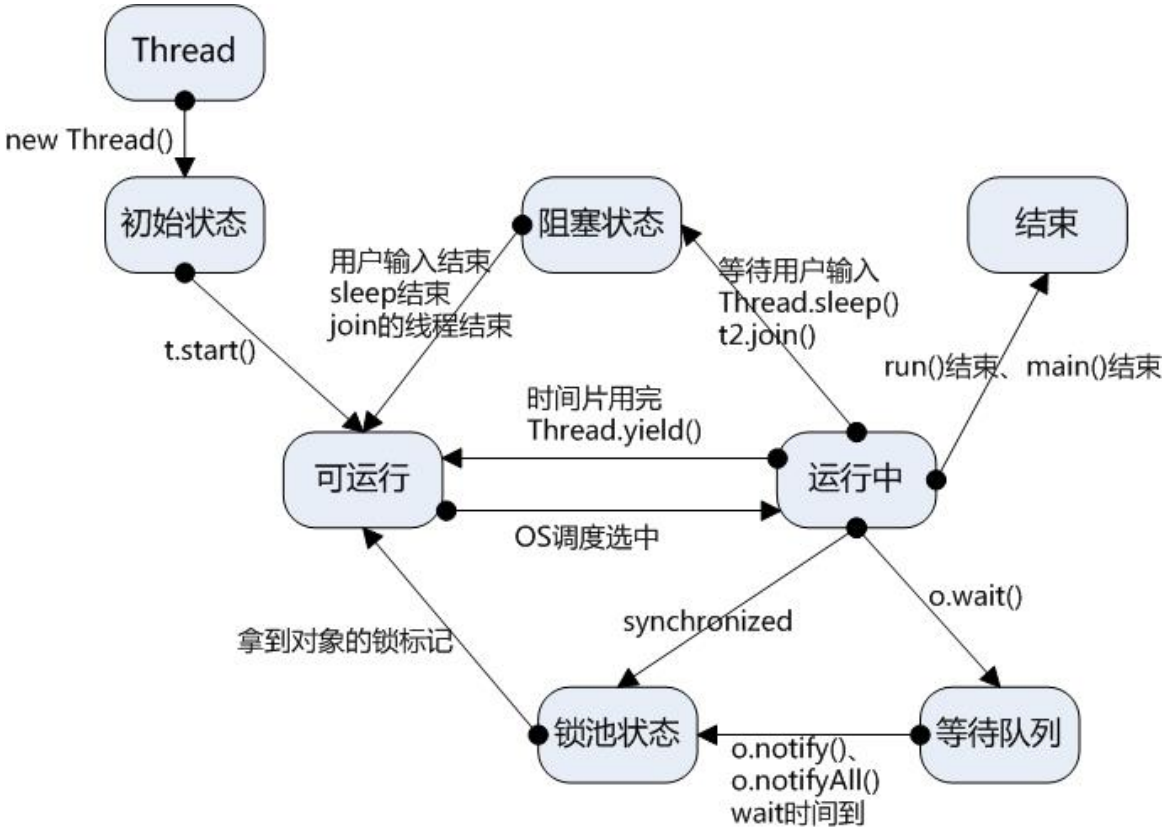
其三是非 Stable 参数（-XX），此类参数各个 jvm 实现会有所不同，将来可能会随时取消，需要慎重使用；

1.2 线程

线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中呈现出间断性。线程也有就绪、阻塞和运行三种基本状态。

线程是利用 CPU 的基本单位，是花费最小开销的实体。

线程状态转换如下图：



1.2.1 Runnable 状态

类型一：

该线程正在已经锁定了锁对象，并且正在运行中。

此时其它需要此锁对象的线程只能等待。

占用 CPU 的，一定是

- "RMI TCP Accept-1499" daemon prio=1 tid=0x97e29ec8 nid=0xdfd **runnable** [0xa3c91000..0xa3c92020]
- at java.net.PlainSocketImpl.socketAccept(Native Method)
- at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:384)
- - locked <0x5d892948> (a java.net.SocksSocketImpl)
- at java.net.ServerSocket.implAccept(ServerSocket.java:450)
- at java.net.ServerSocket.accept(ServerSocket.java:421)
- at sun.rmi.transport.tcp.TCPTransport.run(TCPTransport.java:340)
- at java.lang.Thread.run(Thread.java:595)

类型二：

正在执行中的线程。

- "Thread-41" - Thread t@78 java.lang.Thread.State: RUNNABLE

- at java.net.PlainSocketImpl.socketAccept(Native Method)
- at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:384)
- at java.net.ServerSocket.implAccept(ServerSocket.java:450)
- at java.net.ServerSocket.accept(ServerSocket.java:421)
- at com.xxx.oss.workflow.wes.util.WFSocketServer$ServerStarter.run(WFSocketServer.java:156)

1.2.2 Wait to 状态

waiting to lock 表示该锁正在被别的线程使用，并且那个线程不是 Wait on 状态。

由于在等待锁被释放，当前线程实际是处于被阻塞状态。

- "smpp02:Sender-108" daemon prio=5 tid=0x59a751a0 nid=0x13fc waiting for monitor entry [6066f000..6066fd88]
- at org.apache.log4j.Category.callAppenders(Category.java:185)
- - waiting to lock <0x14fdfe98> (a org.apache.log4j.spi.RootCategory)
- at org.apache.log4j.Category.forcedLog(Category.java:372)
- at org.apache.log4j.Category.log(Category.java:864)
- at org.apache.commons.logging.impl.Log4JLogger.debug(Log4JLogger.java:137)
- at com.xxx.uniportal.comm.base.server.AbstractHandler.send(AbstractHandler.java:407)
- at com.xxx.tellin.usr.uc.sendmessage.UCSMPPTransaction.send(UCSMPPTransaction.java:102)
- at com.xxx.tellin.usr.uc.sendmessage.UCServerProxy.synSend(UCServerProxy.java:134)
- at com.xxx.uniportal.comm.base.proxy.SendWorker.run(AbstractProxy.java:666)
- at com.xxx.uniportal.utilities.concurrent.PooledExecutor$Worker.run(PooledExecutor.java:748)
- at java.lang.Thread.run(Thread.java:534)

1.2.3 Wait on 状态

waiting on 和 locked(成对出现）

线程已经获得并锁定所需要的对象。

正在拿着这个锁等待。

如果其它线程需要，随时可以释放这个锁。

大部分闲置的端口监听线程就处于这个状态：

- "http-0.0.0.0-27443-Processor4" daemon prio=5 tid=0x599a7520 nid=0x1858 in Object.wait() [5c9ef000..5c9efd88]
- at java.lang.Object.wait(Native Method)

- - waiting on <0x1693d2f8> (a org.apache.tomcat.util.threads.ThreadPool$ControlRunnable)
- at java.lang.Object.wait(Object.java:429)
- at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPool.java:655)
- - locked <0x1693d2f8> (a org.apache.tomcat.util.threads.ThreadPool$ControlRunnable)
- at java.lang.Thread.run(Thread.java:534)

1.2.4 Sleep 状态

waiting on condition

因为某个条件而产生的等待。

同锁操作无关。

- "JonasClock" daemon prio=1 tid=0x08290040 nid=0xde1 waiting on condition [0xa3481000..0xa3482020]
- at java.lang.Thread.sleep(Native Method)
- at org.objectweb.jotm.TimerManager.clock(TimerManager.java:165)
- at org.objectweb.jotm.Clock.run(TimerManager.java:63)

1.2.5 Blocked 状态

因为等待某个锁对象而被阻塞。

- "Thread-1" prio=6 tid=0x02b6ac00 nid=0xda4 waiting for monitor entry [0x02ebf000..0x02ebfb14] **java.lang.Thread.State: BLOCKED** (on object **monitor**)
- at com.fox.pandora.DeadLockThreadTwo.run(DeadLockThread.java:44)
- waiting to lock <0x22a26e38> (a java.lang.Object)

1.2.6 Waiting 状态

正在处于空闲等待状态的线程。

- "http-8080-Processor150" - Thread t@256 **java.lang.Thread.State: WAITING on** org.apache.tomcat.util.threads.ThreadPool$ControlRunnable@2ce7cd
- at java.lang.Object.wait(Native Method)
- at java.lang.Object.wait(Object.java:474)
- at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPool.java:657)
- at java.lang.Thread.run(Thread.java:595)

1.2.7 TIMED_WAITING 状态

- 由 Object.wait(int)，Thread.sleep()等方法引起的等待。
- "JMX server connection timeout 1473" - Thread t@1473 java.lang.Thread.State: TIMED_WAITING on [I@19f9cde
- at java.lang.Object.wait(Native Method)
- at com.sun.jmx.remote.internal.ServerCommunicatorAdmin$Timeout.run(ServerCommunicatorAdmin.java:150)
- at java.lang.Thread.run(Thread.java:595)
- "Timer-1" - Thread t@21 java.lang.Thread.State: TIMED_WAITING on java.util.TaskQueue@15a498
- at java.lang.Object.wait(Native Method)
- at java.util.TimerThread.mainLoop(Timer.java:509)
- at java.util.TimerThread.run(Timer.java:462)

1.2.8 死锁

Kill -3 产生 javacore 的同时，也会自动进行死锁的检测：

Found one Java-level deadlock:

"Thread-1":

at com.fox.pandora.DeadLockThreadTwo.run(DeadLockThread.java:44)

- **waiting to** lock <**0x83a920a0**> (a java.lang.Object)

- locked <**0x83a920a8**> (a java.lang.Object)

"Thread-0":

at com.fox.pandora.DeadLockThread.run(DeadLockThread.java:25)

- **waiting to** lock <**0x83a920a8**> (a java.lang.Object)

- locked <**0x83a920a0**> (a java.lang.Object)

2 预防篇

2.1 调优

2.1.1 PT 数据库调优

PT 有很多可以设置的系统参数。其中对性能影响较大的几个参数如下表：

图2-1 PT 性能相关配置参数

配置项	注释
max_connections	最大连接数。默认是 100 个。在大系统中 100 个是比 较少的，一般可能都比 100 多，但是如果过大的话，系统性能反而不高。如果访问量确实很大的话，可以用 pgpool 连接池来管理。 还有就是应用中一些不是经常变化类似数据建议放到内存，以减少大量的数据库访问。
shared_buffers	设置数据库服务器内存共享内存缓冲区的使用量 。一般是物理内存的 20%左右。
wal_buffers	WAL 共享数据存储器使用的内存量。这个参数要求足够大，如果太小的话，log 关联的磁盘操作过频繁。
work_mem	指定的内存量由内部排序操作和哈希表切换到之前使用临时 磁盘文件。 这个参数比较重要的，复杂的 SQL 中如果访问磁盘过多的话，效率会比较低的。
effective_cache_size	设置用于一个查询的有效规模的计划的假设磁盘缓存大小。
checkpoint_segments	自动 WAL 的检查点之间的日志文件段的最大的数量（每段通常是 16MB）。
checkpoint_completion_target	指定检查对象的长度，作为检查点间隔的一小部分。 默认值为 0.5。增加大小能降低系统的不稳定现象。
maintenance_work_mem	称之为维护工作内存，主要是针对数据库的维护操作或者

	语句。尽量的将这些操作在内存中进行。主要针对 VACUUM，CREATE INDEX，REINDEX 等操作。在对整个数据库进行 VACUUM 或者较大的 index 进行重建时，适当的调整该参数非常必要

2.1.2 JVM 调优

2.1.2.1 关键点

- 最大化可在 Minor GC 过程中回收的对象实例数量
- 不要设定超出可用物理内存的堆大小
- 通常意义上：
  - 越大的堆空间，越有利于达成 GC 优化目的
  - 越大的堆空间，发生的 GC 次数越少
  - 越小的堆空间，单次 GC 的效率越高
  - 应该设置最大、最小堆空间尺寸为相同值

    –Xms & -Xmx
  - 应该设置缺省永久区、最大永久区尺寸为相同值

    –XX:PermSize & -XX:MaxPermSize
  - 应该设置缺省 Young 区、最大 Young 区尺寸为相同值

    –XX:NewSize & -XX:MaxNewSize

    这些内存区域的增长或收缩都会导致一次 major GC。

2.1.2.2 优化概述

- Parallel
  - 首先优化 Young 区：尽量避免 major GC 或降低 major GC 频率
  - 优化 Old 区 ：减少或避免对象实例的 promotion 以降低 major GC 频率，甚至避免 major GC
  - 最大化对象堆
- CMS
  - 首先优化 Young 区
  - 着重避免 promotion 的提前发生，在 CMS 中 promotion 是一个代价昂贵的操作; promotion 发生越频繁，越容易导致碎片的产生
  - 应用调优以降低 major GC 的频率

2.1.2.3 Young 区优化

- 目的
  - 减少 GC 次数
  - 最大化在 Young 区被回收的对象实例数量
- 方法

  - 增大 eden 区的尺寸
  - -XX:TargetSurvirorRatio=<> “To”区可被使用的百分比
  - -XX:MaxTenuringThreshold 对象实例在 promotion 前可经历的 minor GC 次数
  - -XX:+AlwaysTenure –XX:+NeverTenure 一般不使用
- MaxTenuringThreshold 的平衡
  - 大

    更少的 promotion，使得对象实例更久的留存在 Young 区，在 Young 区被回收的可能性更大，从而降低 major GC 的频率
  - 小

    避免不必要的对象实例的拷贝操作(Trom -> To)
  - 建议

    拷贝好过 promotion
  - 监控

    --XX:+PrintTenuringDistribution

2.1.2.4 Tenured 区优化

- Parallel
  - 基本上依赖于手工操作
  - 调整 Tenured 区大小
  - -XX:ParallelGCThreads=<> 调整 GC 线程数
  - -XX:+PrintGCDetails –XX:+PrintGCTimestamps 查看 GC 统计信息
- CMS
  - 基本上依赖于手工操作
  - 调整 Tenured 区大小
  - -XX:CMSInitiatingOccupancyFraction=<> 调整触发 GC 时的 Tenured 占用比率
  - -XX:+CMSIncrementalMode 使能 incrementalMode
  - -XX:+CMSIncrementalPacing 允许虚拟机自动统计和调整 major GC 的触发时机
  - -XX:CMSIncrementalDutyCycle=<> 两次 minor GC 之间的间隔，可被用于 major GC 的时长百分比
  - -XX:+PrintGCDetails –XX:+PrintGCTimestamps 查看 GC 统计信息

2.1.3 Linux 环境调优

Linux 环境调优是个非常大的学问，这里只记录当前遇到的问题。

2.1.3.1 句柄

Linux 默认单个进程打开最大句柄数量是 1024，一般 JAVA 进程很容易超过这个限制。一旦进程的文件句柄数量超过了系统定义的值，就会报“too many files open”的错误提示。

几个重要命令：

- ulimit –n：查看当前系统单个进程可打开的最大句柄数
- lsof | grep pid | wc –l：查看某进程已打开句柄数

设置系统级句柄：

- echo 30720 > /proc/sys/fs/file-max
- cat /proc/sys/fs/file-nr 可查看设置结果。（第一个数值为系统已分配句柄，第二个为空闲句柄，第三个为系统最大句柄数）

设置用户级句柄：

- 临时设置－立即生效，但重启后失效
  - ulimit –HSn 4096
- 永久设置－重启才能生效，永不失效
  - vi /etc/security/limits.conf
  - 增加如下行:
  - * soft nofile 2048
  - * hard nofile 32768
  - 星号表示任何用户，soft/hard 表示软限制、硬限制。

说明

设置后需要重启。

用户级设置即为一个进程所能打开最大数量，也即 ulimit –n 所看内容。

2.1.3.2 端口

当客户端在网络连接 TCP 消息建链都不成功的时候，需要看下本地端口是否占满了。

使用 netstat –antp|grep port|wc –l 命令可以查看该类连接占用了多少端口。如果超过 2 万，且大量处于 TIME_WAIT 状态的话就比较危险了。因为 LINUX 系统总共端口有 65535 个，实际建链使用的端口是有范围的（/proc/sys/net/ipv4/ip_local_port_range 中配置），一般也就 2 万多个端口可用于建链。

如果系统确实需要建这么多的链接，那就需要优化环境了。从以下几方面优化：

- time_wait 连接重用： echo 1 > /proc/sys/net/ipv4/tcp_tw_reuse
- 快速回收 time_wait 连接： echo 0 > /proc/sys/net/ipv4/tcp_tw_recycle
- 最大 time_wait 连接长度： echo "12000" > /proc/sys/net/ipv4/tcp_max_tw_buckets
- 向外连接可用端口范围： echo "20000 65000" > /proc/sys/net/ipv4/ip_local_port_range

危险

以上参数非必要不要修改。

2.2 性能编程军规

2.2.1 SQL

推荐阅读：http://platformdoc.xxx.com/hedex/hdx.do?lib=885977612

2.2.1.1【创建必要的索引】

索引，使用索引可快速访问数据库表中的特定信息。

并非所有的数据库都以相同的方式使用索引。作为通用规则，只有当经常查询索引列中的数据时，才需要在表上创建索引。索引占用磁盘空间，并且降低添加、删除和更新行的速度。在多数情况下，索引用于数据检索的速度优势大大超过它的不足之处。但是，如果应用程序非常频繁地更新数据或磁盘空间有限，则可能需要限制索引的数量。

2.2.1.2【尽量避免在循环中调用 SQL】

能不在循环中调用同一个 SQL 就不要这么做。考虑用 SQL 替代循环。

能写成一个 SQL 的就不要写分开写成多个。

如：按条件循环查询一条 SQL，则可以使用 where in (xxx)以及 group by 合并成一个 SQL。

2.2.1.3【尽量避免索引失效场景】

以下场景会导致索引失效：

- 类型不匹配
- 对列使用了函数
- 使用索引实际会降低速度（如上百万条数据查询）
- 结果集返回比例较大
- SQL 中使用了 is null
- SQL 中的 LIKE 条件前加了%通配符（where name like ‘%zhangsai%’）
- 如果是复合索引，只有在它的第一列（leading column）被 where 子句引用时，优化器才会选择使用该索引。

2.2.1.4【避免使用 SELECT *】

尤其在列多的时候。

8 万多的数据表，60 列：

- Select * from table -- 耗时几十秒
- Select 列 1 from table --耗时几秒

另外也 count 时要这样写:

Select count(1) from table;

2.2.1.5【尽量避免使用不必要的函数】

select xxx from table where timezone(xx_time, xxx) >= condition;

把 condition 转换成 xx_time 列可直接比较的值，大数据量情况下性能会提升很多。

2.2.1.6【尽量避免使用 OFFSET】

实测 PT 数据库中，OFFSET 偏移量超过 10 万后，SQL 执行时间就会慢 1S。因为它会把偏移量前面的数据也往内存里加。

2.2.2 JAVA

以下从 JAVA 编程军规中摘取了性能相关部分。本手册只列举条例，详细例子请阅读《JAVA 编程军规 V1.0》。

2.2.2.1【确保程序不再持有无用对象的引用，避免程序内存泄露。】

Java 中的内存泄露更准确的提法是“无意识的引用保留”，GC 会在程序运行过程中对每一个对象进行检查，如果当前程序中不在引用此对象，则此对象被标识为垃圾对象，可以被回收。但是如果程序中保留了对无用对象的引用则会造成 GC 无法检测出垃圾对象，进而无法回收垃圾对象的内存。

2.2.2.2【在进行三个字符串(不包含三个)以上的串联操作时必须使用 StringBuilder 或 StringBuffer，禁止使用“+”。】

java 中字符串是不可变对象(immutable)，在进行字符串串联操作(字符串+)时会生成临时对象。这些对象没有任何意义，会增加 JVM 垃圾收集的负担。如果串联字符串比较多会严重影响程序性能。StringBuffer 和 StringBuilder 功能完全一致，唯一的区别只是 StringBuffer 中的每个方法都是线程安全的，在无需考虑线程同步的场景使用 StringBuilder 性能更高。

2.2.2.3【根据应用场景选择最适合的容器，避免因为容器选择不当造成程序性能问题。】

ArrayList：

- 1. ArrayList 内部是使用 Object 数组来实现，随机访问速度很快。
- 2.向 ArrayList 内添加元素如果遇到内部数组需要扩容，则需要内存拷贝，速度慢。
- 3.在 ArrayList 内插入，删除元素涉及内存拷贝，速度很慢。

LinkedList:

- 1.LinkedList 内部使用链表实现，随机访问其中的元素很慢。
- 2.在 LinkedList 内插入删除元素速度很快。

总结：如果需要频繁的随机访问容器中的数据，不需要频繁的对容器中的数据进行修改或者移动，那么考虑使用 ArrayList；如果你不需要频繁的随机访问容器中的数据，需要频繁的对容器中的数据进行修改或者移动，那么考虑使用 LinkedList；

HashSet：Set 是集合类，该集合不能有“重复”对象存在。HashSet 将持有对象映射到在哈希表中，可快速存取对象。由于使用了 Hash 算法对对象的 hashCode 方法强依赖。

HashMap：Map 是一组 key-value(键值对)集合，其中的 key(键)不能重复。HashMap 用 key 对象生成 hashcode 然后映射到 Entry<K,V>[]数组(键值对数组)中，其 get(Object key) 最佳时间复杂度为 O(1)，最坏则为 O(n)，因此高效的实现 key 对象的 hashCode 方法对 HashMap 的性能至关重要。在存在多线程并发访问 HashMap 的场景下必须考虑加/解锁。

Vector 和 Hashtable 是线程安全的。

2.2.2.4【必须在进行 I/O 操作时使用缓存。】

不使用缓冲的 I/O 操作会频繁的调用操作系统底层函数访问磁盘，使用缓冲机制能带来显著的性能提升。

2.2.2.5【在程序中必须考虑对象重用，避免创建不必要的垃圾对象。】

Java 中除了基本类型外一切皆对象。一个大型系统中肯定存在大量对象互相协作完成各种功能。但是我们必须确保每一个对象是必须的，不是多余的垃圾对象。如果存在大量垃圾对象会增大 GC 负载，对程序性能造成严重影响。例如：程序响应请求时间明显变长，程序吞吐率降低等。

变量定义放到循环外面，尽量少在循环中 new 对象。

2.2.2.6【对多线程访问的变量、方法，必须加锁保护，避免出现多线程并发访问引起的问题。】

目前在实际的产品中多线程的使用非常普遍，且问题较多，因此在进行编码实现的时候必须重点考虑多线程并发的问题。多线程并发问题的实质是多个线程间对共享数据进行同步的问题。根据 Java Language Specification 中对 Java 内存模型的定义, JVM 中存在一个主内存(Java Heap Memory)，Java 中所有变量都储存在主存中，对于所有线程都是共享的。每条线程都有自己的工作内存(Working Memory)，工作内存中保存的是主存中某些变量的拷贝，线程对所有变量的操作都是在工作内存中进行，线程之间无法相互直接访问，变量传递均需要通过主存完成。

根据上述内存模型的定义，要在多个线程间安全的同步共享数据就必须使用锁机制，将某线程中更新的数据从其工作内存中刷新至主内存，并确保其他线程从主内存获取此数据更新后的值再使用。

2.2.2.7【新起一个线程，都要使用 Thread.setName(“xx”)设置线程名。】

这样在打印日志的时候就会把线程名称打印出来。如果不设置，线程名称默认为 Thread-1 这样类型的，日志区分不出来那个线程打印的。这会给问题定位带来很大不方便。

2.2.2.8【线程使用时，要在代码框架中使用线程池，避免创建不可复用的线程。禁止在循环中创建新线程，否则会引起 JVM 资源耗尽。】

对于经常被调用的代码段中有线程的情况，也要使用线程池。如接口中新建线程，每个请求都要建个线程。

3 定位篇

一般说起性能问题，无非是 CPU、内存和 IO。下面从这三个方面入手说下性能问题定位思路。SQL 问题本身不属于性能问题范畴，但在大数据量情况下，往往很多性能问题是 SQL 不合理导致的,所以这里也讲下 SQL 问题定位。另外前台 JS 往往也有性能问题，这里也顺便提供下定位思路。

万事万物皆是联系的，所以定位性能问题时，不能完全割裂独立地看 CPU、内存、IO，要根据当时环境实际情况，全方面的排查分析。否则很容易走错方向。

JAVA 进程复位时，一般 JVM 会输出 hs_err_pidxxx.log 日志到工作目录，里面会详细记录复位原因、当时系统的状况（CPU、内存情况，GC 情况等等），文件最上面会写 Possible reason，那个只是 possible，真正原因不一定准。要仔细分析 GC、线程栈等，综合分析后再明确方向。复位时也可能有 hprof 文件生成，说明是堆内存溢出了。如果啥都没有，就得看日志了，包括系统日志（/var/log/message）。

3.1 CPU

3.1.1 监控线程

命令：top -Hp <pid> 即可查看<pid>进程所对应各线程的运行情况。

输出结果如下：

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
29348	root	20	0	4585m	1.8g	11m	R	8	23.5	0:22.79	java
29390	root	20	0	4585m	1.8g	11m	S	6	23.5	0:26.97	java
29427	root	20	0	4585m	1.8g	11m	S	4	23.5	0:08.37	java
753	root	20	0	4585m	1.8g	11m	S	2	23.5	1:50.87	java
761	root	20	0	4585m	1.8g	11m	S	2	23.5	0:27.91	java

其中的 pid 列即为该线程 ID。转换为 16 进制到，到线程堆栈中即可搜到相应线程信息。

3.1.2 获取堆栈

通过 Jvisaulvm、Jstack、kill -3 都可以获取。详见工具篇。

3.1.3 线程栈分析

主要关注线程数量、状态、相同执行堆栈的数量。线程状态及转换见 1.2 节。

1.如果 CPU 冲高后，停止业务后 CPU 使用率降下来，基本上就没有死循环；反之如果 CPU 居高不下，甚至更高，那么基本上肯定是有死循环了。

2.如果某个线程在连续的多个 Thread Dump 中状态都为 Runnable，则可能存在性能问题。

3.如果多个 Runnable 的线程都在执行同一个方法，则可能存在性能问题。

4.如果多次观察到多个线程 wait to 同一个对象（或被同一个对象 BLOCKED），则说明存在锁同步和 CPU 问题。

5.wait on 和 sleep 状态（或 WAITING 和 TIME_WAITING）的线程不耗费 CPU，不需要关注。

6.一般有 dead lock 时，进程已经挂死，想不注意到都不行。

7.可以使用图形化工具如 TDA（或集成在 VisualVM 中的 TDA 插件）进行分析。

8.如果遇到下面这样的线程，没有任何堆栈信息，可用 BTrace 工具跟踪分析。

- "Timer-89683" prio=10 tid=0x000000000295b000 nid=0x2ca6 runnable [0x0000000000000000]
- java.lang.Thread.State: RUNNABLE
-
- "Timer-89674" prio=10 tid=0x00007f51a3fd0800 nid=0x2ca5 runnable [0x0000000000000000]
- java.lang.Thread.State: RUNNABLE

3.2 Memory

3.2.1 判断溢出类型

表3-1 内存溢出类型

内存对象	对应存放区域	
JAVA 对象	JAVA 堆栈（Heap）	
JAVA 类	永久区（PermGen）	
线程	本地内存 （Native Memory）	线程栈（Stack)
本地代码/数据		动态库等
打开/共享文件		

- 如果程序自动生成 hprof 文件，一定是 Heap 内存溢出。否则要具体定位是哪类溢出。
- 根据 OOM 日志判断溢出类型：
  - **堆内存溢出**

    Exception in thread “main” java.lang.OutOfMemoryError: Java heap space

    Exception in thread “main” java.lang.OutOfMemoryError: Requested array size exceeds VM limit
  - **永久区内存溢出**

    Exception in thread “main” java.lang.OutOfMemoryError: PermGen space
  - **本地内存溢出**

    Exception in thread “main” java.lang.OutOfMemoryError: request <size> bytes for <reason>. Out of swap space?

    Exception in thread "main" java.lang.OutOfMemoryError: <reason>

    <stack trace>(Native method)

    Native memory allocation (malloc) failed to allocate…
  - **本地线程资源不足**

    Exception in thread <thread>/Caused by java.lang.OutOfMemoryError: unable to create new native thread
  - **本地直接内存溢出**

    java.lang.OutOfMemoryError: Direct buffer memory
  - **推荐阅读：https://eyalsch.wordpress.com/2009/06/17/oome/**

📖 说明

首先我们要区分是内存泄漏还是内存溢出，两者抛出的异常都是：

java.lang.OutOfMemoryError;内存溢出并不是真正的内存泄漏，泄漏通常指的是某些 JAVA 对象预期应该被回收，但是实际还被引用，导致不断的被创建但老的又没有被回收，最终导致内存达到可用内存上限发生内存溢出；内存溢出往往是由于一次需要开辟很大的堆内存，但是由于虚拟机可以使用的最大堆内存不足，导致内存申请和分配失败；这类问题通常都是由于虚拟机启动参数内存配置不合理导致的。

修改方式为：

在 JAVA 启动项中修改-Xmx 参数，通常对于一个大型应用系统来说，该值为 4096M，如果是 32 位操作系统和虚拟机，建议设置为 2048M;

内存泄漏的易发地带：

缓存策略不合理：有些缓存的老化算法不合理，导致缓存策略不生效，无法及时的将缓存中的数据交换到磁盘中；还有些缓存设计不合理，缓存的内存上限超过了系统可用的最大内存；

容器类对象：数据库数据映射到内存、性能数据统计、数据库批量提交等等功能都会大量使用容器类做数据管理，在一些情况下，如果容器类中的数据没有被及时清除，可能就潜在存储内存泄露；

一次加载大量数据到内存中，可能导致内存溢出，之前很多管理系统都发生过类似问题，在将某些数据展示到前台页面的时候，由于过滤条件无法有效过滤所有数据，导致瞬间系统内存溢出；

全局的静态类，特别是做数据管理类包含容器对象的；

第三方库导致的，例如对象池、缓存池等，如果使用不当或者对其本身理解不透彻，很容易引发问题；因此，并不建议使用太多这类缓存池。

3.2.2 溢出分析

3.2.2.1 堆内存分析

- 1 查看 GC 信息（详见 jstat 工具使用说明），如果年轻代和老化代使用率将近 100%，且多次 GC 依然没有降低，就需要 DUMP 下内存进一步分析了。
- 2 使用 jmap 或 jvisualvm 工具 dump 内存，使用 MAT 工具打开。
- 3 从 MAT 工具打开的图形界面上查看是哪个对象占用了最多的内存，然后通过 java basics/open in dominator tree 查看引用关系，结合代码分析 GC 回收不了该对象的原因。有没有内存泄漏，也就看得很清楚了。

3.2.2.2 永久区内存分析

通过 VisualVM 和 JConsole 等工具可以观察到 PermGen 使用情况和加载类的数量。

3.2.2.3 本地内存溢出分析

使用 pmap 观察本地内存的主要内容（详见 4.7.8 节，pmap 使用。）

根据这些信息，可以分析出大部分内存块的用途：

- 加载的动态库

  动态库可以很明显地根据其名称确认，如下面蓝色框中的一系列动态库。对应一个动态库肯定有两个内存块，其中一块属性为 r-x--（可读可执行），为该动态库的代码区；另一块为 rw---（可读可写），为该动态库的数据区。

- 线程栈

  由两块背靠背的内存块构成：其中一块大小在 4K－20K 之间，属性为------，为隔离区，用途是防止线程栈内存块被其它线程等侵占；另一块即为真正的线程栈。如下图黑色框中部分。

- JVM 内存

  即 JVM 堆内存、永久区等由 JVM 控制的内存，其属性为 rwx--（可读可写可执行），并且前后没有隔离区，有时会连续出现。如下图中绿色框中部分。

- 打开的文件

  文件所占用的内存块非常容易辨认，其属性一定为 r—s-（可读共享），并且其文件名也会被清楚地标识出来。如下图中红框中部分。

图3-1 本地内存使用明细

3.2.2.4 本地直接内存溢出

解决方案：

- 1 增大 SWAP 分区
- 2 检查是否同时有使用 Direct ByteBuffer（例如用到了 NIO 框架 mina），且 JVM 参数中配置了 DisableExplicitGC。

推荐阅读：http://iamzhongyong.iteye.com/blog/1743718

如果上述方案不能解决，只能使用终极大招，使用 perftools 监测内存泄漏：

http://yuncode.net/article/a_517e7cc033ed396

3.3 IO

1 观察 TOP 信息中的 wa 的百分比。

2 如果 wa 一直比较高的话，IO 可能就有问题了。使用 iostat 详细观察。详见 4.8.7 节。

3 确定 IO 有问题的话，就要定位问题出在哪了。

- 首先要根据业务场景，看下是否能明显判断出来业务本身确实有耗大量 IO 的地方。
- 其次，如果看不出来的话，排查下可能是哪个进程导致的，然后用 strace 命令（详见 4.8.4）抓一下系统调用，用 lsof 命令（详见 4.8.2）抓一下当时业务进程打开的文件，再抓一下当时的线程堆栈，基本上能看出来 IO 高的这会是在干吗。

- 最后，结合业务场景、代码分析具体原因。如果是第三方软件导致的，那 strace 抓到的系统调用就是铁证。

说明

Strace 信息中 lseek、fstat、read、write 等几个文件相关操作要重点关注。结合执行时间，就能找到耗 CPU 的地方，再反推下业务代码差不多就找到问题所在了。

3.4 SQL

这里主要讲下 GaussDB 的手段。其它数据库思路也大同小异。

1 在性能环境上多次抓取当前在执行的 SQL。每次都能被抓到的，就需要重点关注下了。

方法：在数据库客户端 SQL 执行命令行中执行以下 SQL

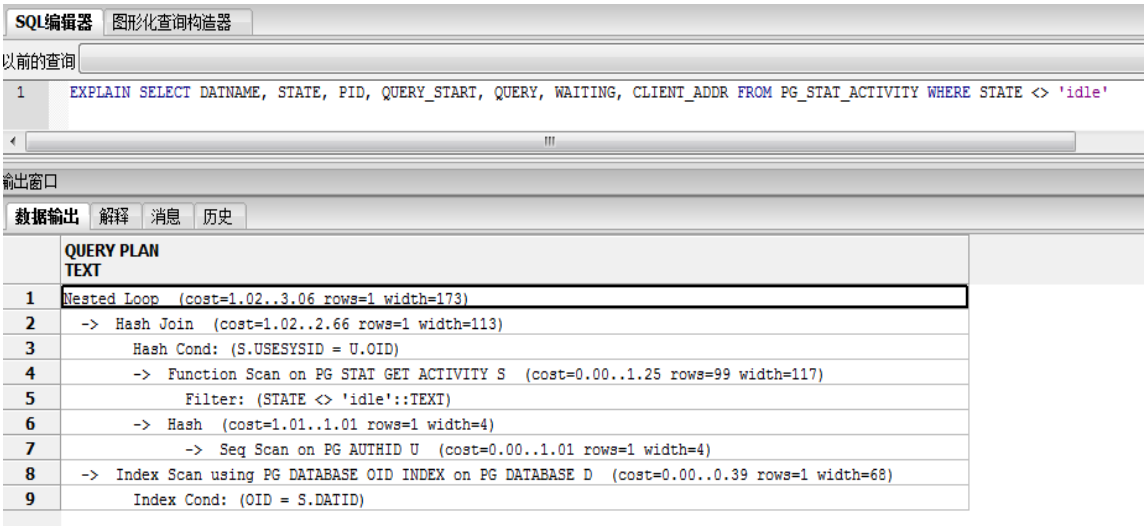
SELECT DATNAME, STATE, PID, QUERY_START, QUERY, WAITING, CLIENT_ADDR FROM PG_STAT_ACTIVITY WHERE STATE <> 'idle'

2 将上述抓到的 SQL 在数据库客户端 SQL 执行命令行中执行一下，看下执行时间，如果时间较长，那就有问题了。

3 对于比较耗时的 SQL，可以分析一下其执行计划

方法：在数据库客户端 SQL 执行命令行中，将要分析的 SQL 前加个 explain，然后执行。如下图

图3-2 GaussDB 执行计划分析



说明

Seq Scan 和 Index Scan 分别表示全表扫描和索引扫描。

3.5 JS

对于前台响应慢的问题，主要定位思路是 Java Script 的加载和执行时间。一般当前服务器性能上，加载不会太慢。可以在源码中加入以下时间戳打印函数，然后在 js 文件的各处打印，可以观察出执行时间，就能锁定具体出问题的脚本了。再根据业务功能进行优化了。

```
log = function(str) {
    try {
        var date = new Date();
        console.info(date.getMinutes() + ':' + date.getSeconds() + '.' + date.getMilliseconds() + ' : ' + str);
    }
    catch(e){}
}
example: log('Debug start');
```

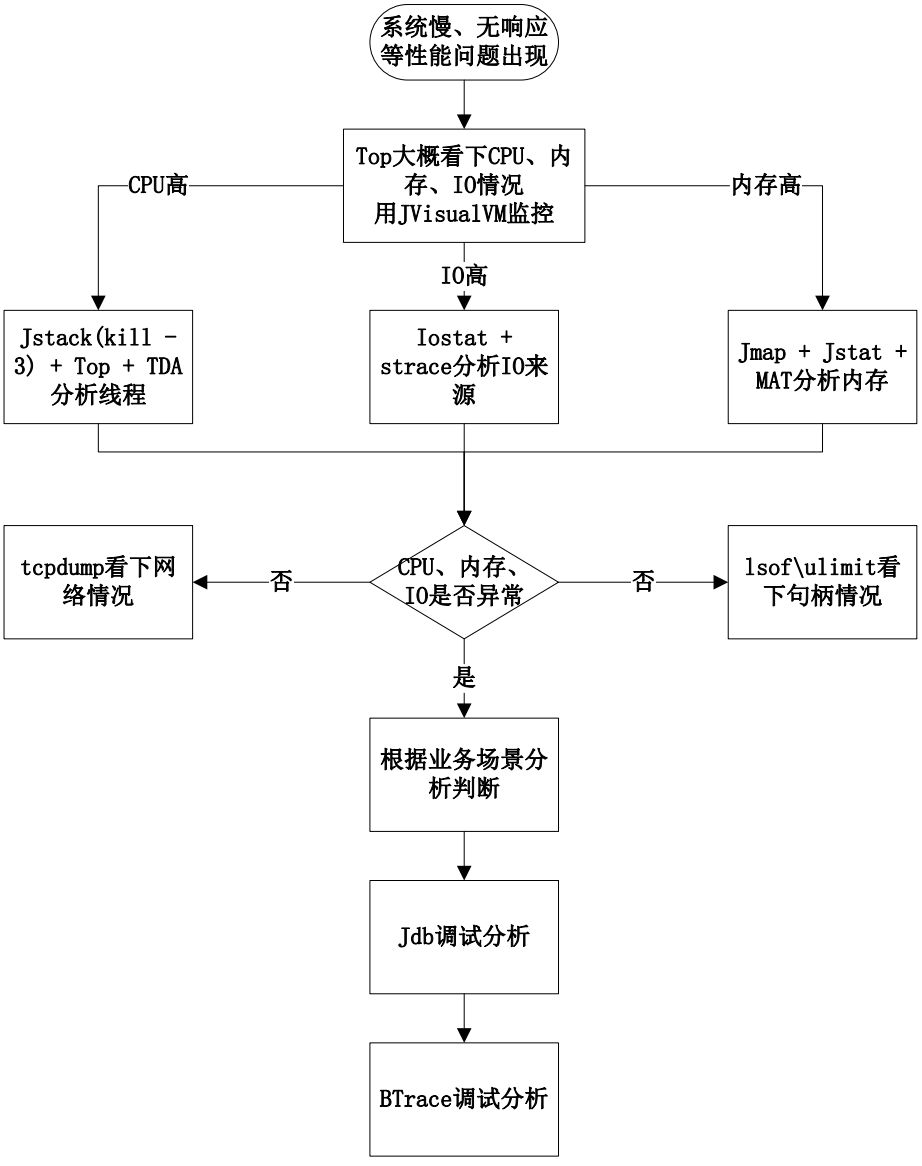
说明

该函数是将时间戳打印到控制台上。调试前需要打开 JS 调试控制台。（IE 下按 F12 打开）

前台页面中能不加载的 JS 文件尽量不要加载。

4 工具篇

用好工具，事半功倍。但工具也不是万能的，关键是掌握原理。工具使用大致如下：



说明

1 JDK 自带工具访问进程，需要切到启动该进程的用户下执行（JVisualVM 连接用户使用该用户）。如，OC 当前使用的是 appuser 用户启动，所以要切到 appuser 用户下才能正常使用 JDK 自带的工具。

2 JDK 自带工具使用，建议 Jconsole 和 JvisualVM 在客户端连接服务器来监控，减少工具本身耗的资源性能。

3 Jstat, Jstack, Jmap, Jdb 等工具使用，将 JDK 拷贝到要操作的 LINUX 服务器上，将 JDK 上述工具建立映射，如: sudo ln –s ${JAVA_HOME}/bin/jstack /bin/jstack，就可以直接敲命令用了。

4 如果 JDK 自带工具访问进程被拒，主要看下/tmp/hsperfdata_$username/这个目录下有没有该 JAVA 进程号的为名的文件。不要随意删除这里的文件。

4.1 MAT

MAT：Memory Analyzer，内存分析工具。JMap dump 出来的 hprof 文件用此工具分析。

4.1.1 工具安装

- 方式一：在现有 ECLIPSE 里安装
  - 1 设置 Eclipse 的上网代理

    默认的 Eclipse 是不用代理上网，但在一些公司的局域网，需要使用代理上网，因而需要手工设置 eclipse 的上网设置

    window-->preferences-->general-->network connections
    选中 manual proxy configuration: 依次填入 http proxy , port 就 ok 了。

  - 2 安装插件

    插件地址：http://download.eclipse.org/mat/1.3/update-site/

- 方式二：直接下载工具

地址：http://www.eclipse.org/mat/downloads.php

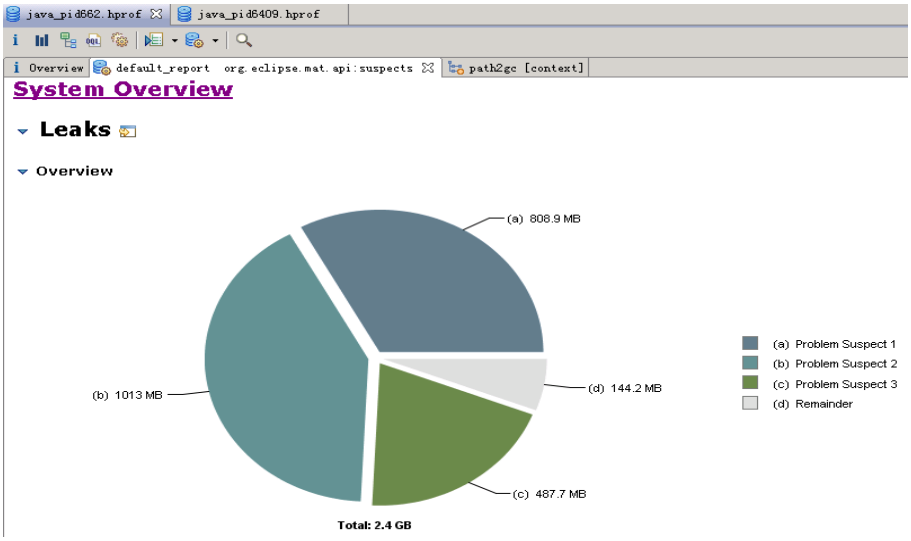
注意：

- 1 如果是 WIN7 系统，要下载 64 位的哦，否则打不开。
- 2 如果文件特别大，只能到一个内存足够大的执行机上打开。

4.1.2 分析文件

工具打开 phrof 文件后如下图：

图4-1 进程 heap 分析图



扇形图中，显示了问题内存占用比重。占用的越多，问题越大。点击“Leak Suspects”可查看问题明细列表。再点“details”可查看每个问题的统计信息。包括回收引用路径、内存中对象统计等。如下图：

图4-2 详细图

The class **"javax.crypto.JceSecurity"**, loaded by **"<system class loader>"**, occupies **1,000,534,360 (41.16%)** bytes. The memory is accumulated in one instance of **"java.lang.Object[]"** loaded by **"<system class loader>"**.

Keywords
java.lang.Object[]
javax.crypto.JceSecurity

Shortest Paths To the Accumulation Point

Class Name	Shallow Heap	Retained Heap
java.lang.Object[16384] @ 0x791f93c08	65,552	1,000,532,912
table java.util.IdentityHashMap @ 0x751cbfae8	40	1,000,532,952
verificationResults class javax.crypto.JceSecurity @ 0x751cb1488 System Class	**40**	**1,000,534,360**

Accumulated Objects

Class Name	Shallow Heap	Retained Heap	Percentage
class javax.crypto.JceSecurity @ 0x751cb1488	40	1,000,534,360	41.16%
java.util.IdentityHashMap @ 0x751cbfae8	40	1,000,532,952	41.16%
java.lang.Object[16384] @ 0x791f93c08	65,552	1,000,532,912	41.16%
org.bouncycastle.jce.provider.BouncyCastleProvider @ 0x77c85cf58	96	302,256	0.01%
org.bouncycastle.jce.provider.BouncyCastleProvider @ 0x77c85d078	96	302,256	0.01%

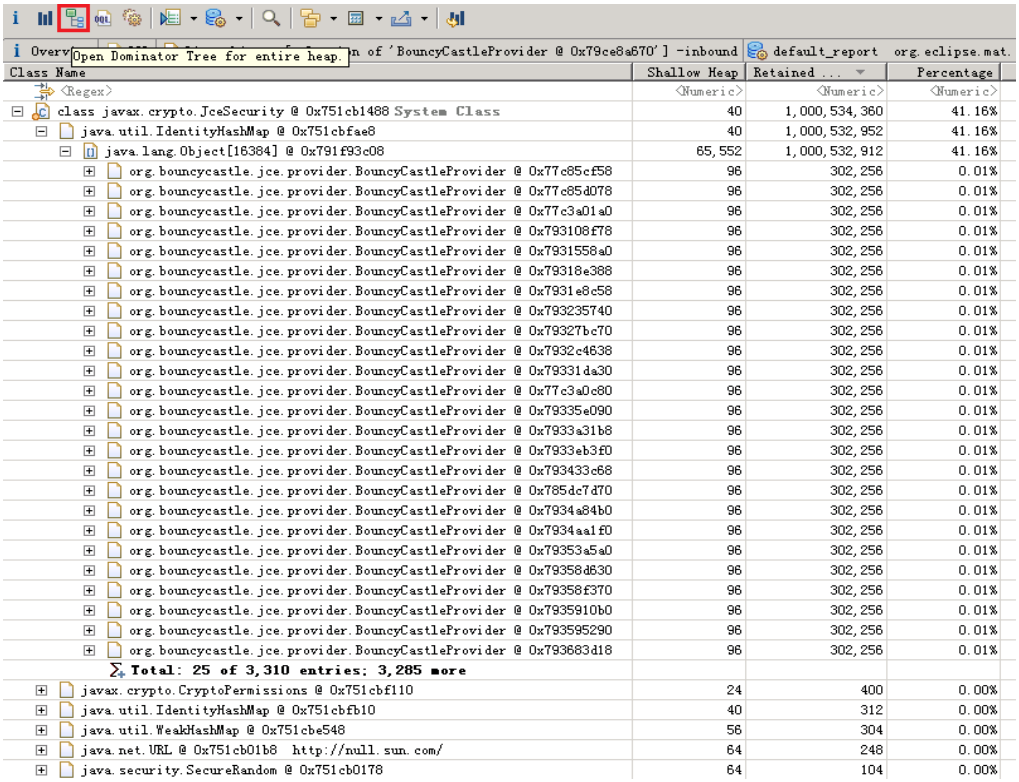
说明

Shallow Heap：指当前对象占用的内存大小。

Relastiond Heap：指当前对象大小 + 所有引用的对象大小。

MAT 功能很强大，DOMINATOR TREE 查看调用关系更直接。OQL 提供查询等等。

图4-3 DOMINATOR TREE 视图

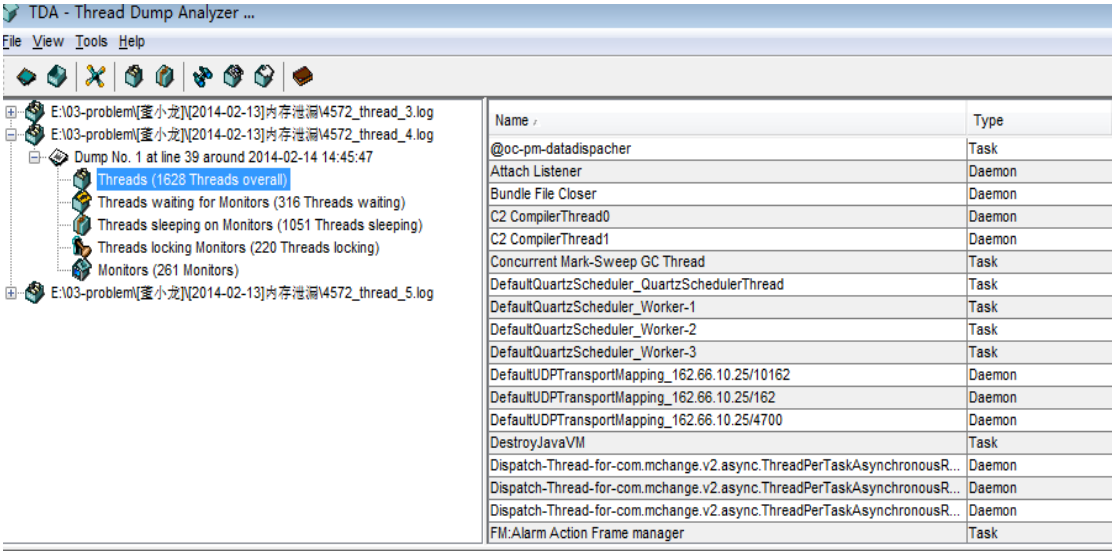


4.2 TDA

TDA：Thread Dump Ayalyzer

下载地址：https://java.net/projects/tda/downloads

TDA 就是线程栈的图形化查看工具。可独立运行，也可下载其插件在 Jvisualvm 中使用。



4.3 JvisualVM

JAVA 性能监控的工具非常多，其中 JDK6.0 之后携带的可视化工具 VisualVM 是一款多合一的故障定位和分析工具，经过实际测试，其对性能的影响非常小，这是 Jprofiler、JProbe 等工具无法比拟的。

4.3.1 VisualVM 安装

VisualVM 基于 NetBeans 平台开发，因此其具备插件扩展能力，通过插件扩展，可以支持如下能力：

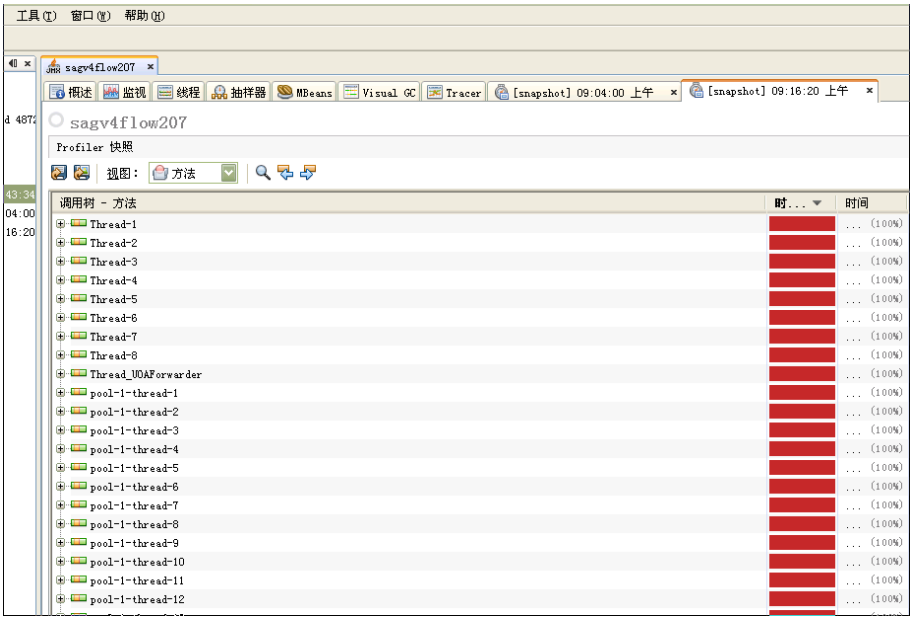
- 虚拟机进程配置信息，包括 JVM 参数和系统属性；
- 监控 CPU、内存、装载的类和线程信息；
- 线程的实时运行信息，可以 dump 线程堆栈；
- 通过 Profile 抽样器可以采集线程的调用次数、耗时，统计热点方法；

插件的安装方式如下：

启动 VisualVM，点击【工具】栏，选择【插件】，会弹出如下页面：

图4-4 插件安装

勾选需要安装的插件，点击安装即可，安装之后的页面如下：



4.3.2 运行连接

运行前提：在 JVM 启动参数中添加 JMX 配置：

-Dcom.sun.management.jmxremote.port=18950
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false

运行：本地安装 JDK 后，配置 JAVA_HOME 等环境变量。

在命令行输入：jvisualvm

打开后添加 JMX 连接，如下图：

图4-5 连接 JMX

4.3.3 监控

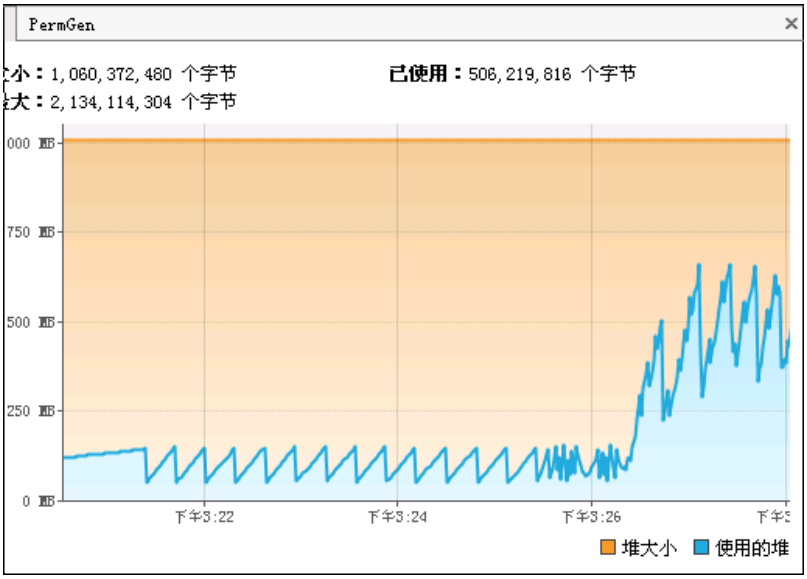
比较实用的功能是：

- 【线程】标签页的线程监控功能；
- 【抽样器】标签页的性能数据收集功能；
- 【监视】标签页的堆内存数据收集功能

下面分别讲解用法：

内存监控如下：

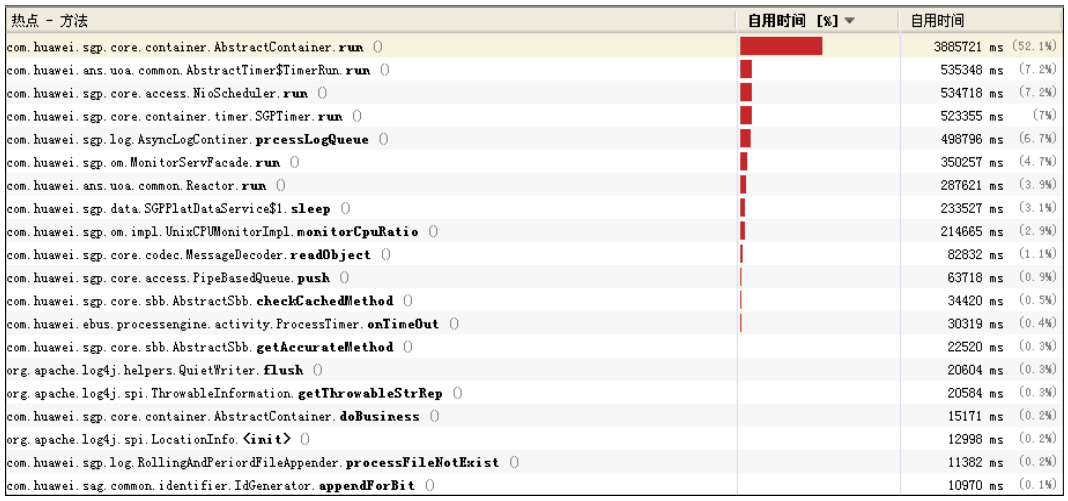
图4-6 内存监控



通过对内存的使用趋势，来判断是否存在内存泄露，如果存在，使用 1.1 中的方式来定位问题；

性能瓶颈分析：

点击【抽样器】标签，切换到性能统计视图，点击【CPU】按钮开始对 CPU 调用进行统计，如下图所示：



热点 - 方法	自用时间 [%]	自用时间
com.huawei.sgp.core.container.AbstractContainer.**run** ()		3885721 ms (52.1%)
com.huawei.ans.uoa.common.AbstractTimer$TimerRun.**run** ()		535348 ms (7.2%)
com.huawei.sgp.core.access.NioScheduler.**run** ()		534718 ms (7.2%)
com.huawei.sgp.core.container.timer.SGPTimer.**run** ()		523355 ms (7%)
com.huawei.sgp.log.AsyncLogContiner.**prcessLogQueue** ()		498796 ms (6.7%)
com.huawei.sgp.om.MonitorServFacade.**run** ()		350257 ms (4.7%)
com.huawei.ans.uoa.common.Reactor.**run** ()		287621 ms (3.9%)
com.huawei.sgp.data.SGPPlatDataService$1.**sleep** ()		233527 ms (3.1%)
com.huawei.sgp.om.impl.UnixCPUMonitorImpl.**monitorCpuRatio** ()		214665 ms (2.9%)
com.huawei.sgp.core.codec.MessageDecoder.**readObject** ()		82832 ms (1.1%)
com.huawei.sgp.core.access.PipeBasedQueue.**push** ()		63718 ms (0.9%)
com.huawei.sgp.core.sbb.AbstractSbb.**checkCachedMethod** ()		34420 ms (0.5%)
com.huawei.ebus.processengine.activity.ProcessTimer.**onTimeOut** ()		30319 ms (0.4%)
com.huawei.sgp.core.sbb.AbstractSbb.**getAccurateMethod** ()		22520 ms (0.3%)
org.apache.log4j.helpers.QuietWriter.**flush** ()		20604 ms (0.3%)
org.apache.log4j.spi.ThrowableInformation.**getThrowableStrRep** ()		20584 ms (0.3%)
com.huawei.sgp.core.container.AbstractContainer.**doBusiness** ()		15171 ms (0.2%)
org.apache.log4j.spi.LocationInfo.**<init>** ()		12998 ms (0.2%)
com.huawei.sgp.log.RollingAndPeriordFileAppender.**processFileNotExist** ()		11382 ms (0.2%)
com.huawei.sag.common.identifier.IdGenerator.**appendForBit** ()		10970 ms (0.1%)

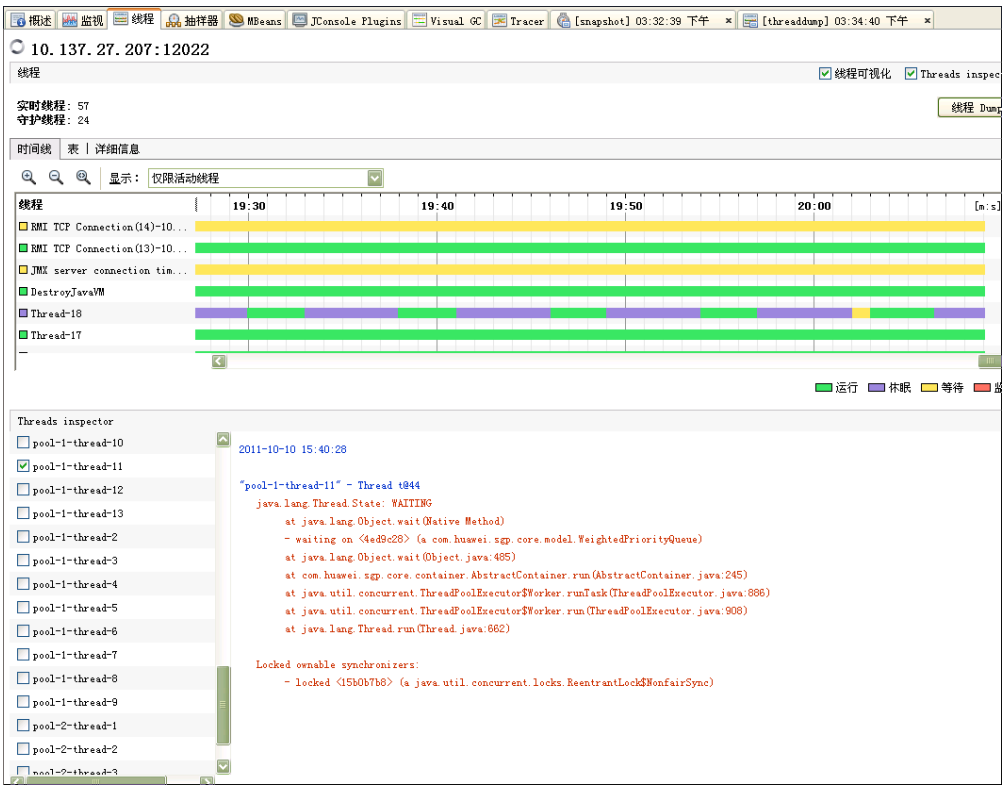
通过【自用时间】我们可以一目了然的定位到热点区域，如果想精确定位到堆栈信息，点击【线程 dump】标签，dump 出当前的线程堆栈，可以做详细的调用分析：

```
"pool-1-thread-3" - Thread t@35
   java.lang.Thread.State: RUNNABLE
        at java.io.FileOutputStream.writeBytes(Native Method)
        at java.io.FileOutputStream.write(FileOutputStream.java:282)
        at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:65)
        at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:123)
        - locked <4968536e> (a java.io.BufferedOutputStream)
        at java.io.PrintStream.write(PrintStream.java:432)
        - locked <20ae29f3> (a java.io.PrintStream)
        at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:202)
        at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:272)
        at sun.nio.cs.StreamEncoder.flushBuffer(StreamEncoder.java:85)
        - locked <603d8068> (a java.io.OutputStreamWriter)
        at java.io.OutputStreamWriter.flushBuffer(OutputStreamWriter.java:168)
        at java.io.PrintStream.write(PrintStream.java:477)
        - locked <20ae29f3> (a java.io.PrintStream)
        at java.io.PrintStream.print(PrintStream.java:619)
        at java.io.PrintStream.println(PrintStream.java:756)
        - locked <20ae29f3> (a java.io.PrintStream)
        at com.huawei.sag.framework.flow.AbstractSagSbb.onResponseEvent(AbstractSagSbb.java:139)
        at com.huawei.sag.sms.unisms.UniSmsProcessSbb.onOutwardSmpp34SubmitSmRspEvent(UniSmsProcessSbb.java:79)
        at sun.reflect.GeneratedMethodAccessor139.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at com.huawei.sgp.core.sbb.AbstractSbb.onEvent(AbstractSbb.java:488)
        at com.huawei.sgp.core.container.ServiceContainer.doInvokeSbb(ServiceContainer.java:358)
        at com.huawei.sgp.core.container.ServiceContainer.processLogic(ServiceContainer.java:308)
        at com.huawei.sgp.core.container.AbstractContainer.doBusiness(AbstractContainer.java:358)
        at com.huawei.sgp.core.container.AbstractContainer.run(AbstractContainer.java:269)
        at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
        at java.lang.Thread.run(Thread.java:662)
```

通过间隔性的取【快照】，可以精确的分析出系统调用的热点区域，定位出系统瓶颈。

线程监控：

切换到【线程】标签页，可以实时监控线程信息，如下：



4.3.4 动态日志跟踪

BTrace 是 VisualVM 的一个动态日志跟踪插件，它的强大之处在于不停止目标程序的前提下，通过 HotSpot 虚拟机的 HotSwap 技术动态加入原本不存在的调试代码。

该功能的现实意义是：在现网环境中，如果发现由于缺乏关键日志而导致无法精确定位出问题的时候，该如何办？传统的做法是等系统停机的时候，更新一个携带日志的版本用来定位。因为缺乏日志导致问题无法定位的情况在公司的各产品普遍存在，我们迫切需要改变这种现状。

BTrace 的出现解决了我们面临的难题。

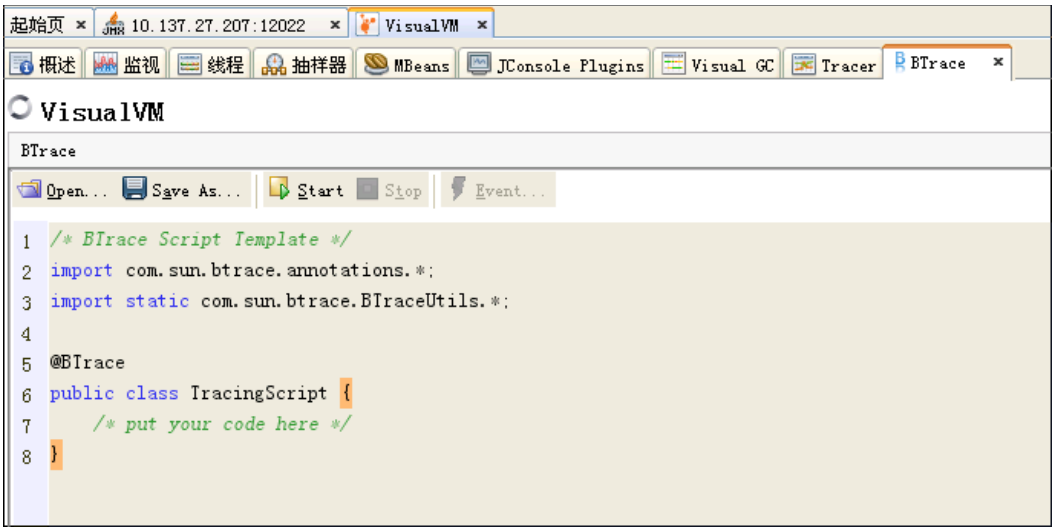
BTrace 的作用不仅仅用于动态日志跟踪，还可以进行性能监控、定位内存泄露等，感兴趣的话可以访问官方网站：

http://kenai.com/projects/btrace

关于 Btrace 的详细使用，这里不再赘述，感兴趣的可以自己访问其官方网站查看样例。在使用 Btrace 的时候，需要注意以下几点：

BTrace 只能连接本地应用程序，因此，如果是在生产环境或者测试环境中，需要安装 Linux 版本的 Btrace 插件；

由于 BTrace 本身也是可视化的，需要通过 Xmanager 等可视化终端连接服务端，启动本地的 VisualVM，如果本地安装了 Btrace 插件，就可以直接使用其功能，样例如下：



4.4 Jstack

Jstack 用于打印进程的线程堆栈。

命令：jstack pid

一般在判断线程死锁、CPU 高等情况下，要看线程堆栈，可用此命令查看输出结果。

配套使用 top –Hp pid 可查看该进程的线程 CPU 使用情况。

4.5 Jdb

Jdb 用来对 core 文件和正在运行的 Java 进程进行实时地调试。一般使用 ECLIPSE 开发时，其调试功能非常强大，基本上用不上 Jdb。但有一种情况下，Jdb 会让你受益匪浅：当调试的代码是第三方或其它来源的代码，没有源码时，Jdb 就派上用场了。

4.5.1 Jdb 的启动

4.5.1.1 调试已启动进程

被调试进程的 JVM 参数配置了远程 DEBUG 端口：

-Xdebug -Xrunjdwp:transport=dt_socket,address=32041,server=y,suspend=n

命令:

jdb –attach 32041

这里有 32041 即为远程调试端口。

4.5.1.2 调试 Java 类

通过在命令行窗口中执行 jdb（确保你的 jdk 安装目录下的 bin 目录在你的环境变量 PATH 中），你就可以启动 Java 调试器：

d:\temp\java\swing>jdb

Initializing jdb ...

>

出现“>”后，jdb 就进入与用户的交互状态，准备接收用户输入的命令。然后我们可以使用 run classname 来加载需要调试的 Java 类

也可以在启动 jdb 时执行要调试得 Java 类名：

d:\temp\java\swing>jdb ProgressMonitorTest

Initializing jdb ...

>

在启动 jdb 时我们可以指定一些参数，比较有用的参数是：

-classpath：指定加载 Java 类的路径

-sourcepath：指定 Java 源代码所在的目录，多个目录之间使用分号“;”分隔

4.5.2 Jdb 命令简介

启动 jdb 后，就进入了交互状态，用户可以键入命令对代码进行调试，下面是 jdb 的命令简介，其中

[]表示可选参数，| 表示二者选一。（下面只列出了比较常用的一部分，完成的 jdb 命令可以输入 help 获取）

run [class [args]] -- start execution of application's main class

运行一个 Java 类，class 指定了类名，args 为运行这个类需要指定参数

threads [threadgroup] -- list threads

列出当前的线程（线程组），对于调试多线程程序或窗口程序很有用

thread <thread id> -- set default thread

设置当前缺省的线程，已 threads 命令显示的线程 ID 为参数。一般调试命令如设置断点、查看变量都是在缺省线程上操作的。

suspend [thread id(s)] -- suspend threads (default: all)

挂起一个或多个线程，不指定线程 ID 时，暂停所有线程

resume [thread id(s)] -- resume threads (default: all)

恢复一个或多个被挂起的线程，不知丁线程 ID 时，恢复所有被挂起的线程

where [<thread id> | all] -- dump a thread's stack

显示一个线程的堆栈，不指导线程 ID 时，显示缺省线程的堆栈

interrupt <thread id> -- interrupt a thread

中止一个线程，必须指定线程 ID

print <expr> -- print value of expression

打印表达式/变量的值，如果 expr 指定的是一个对象，输出的是这个对象 toString 返回的值

dump <expr> -- print all object information

显示 Java 对象所有的信息，

eval <expr> -- evaluate expression (same as print)

执行一个表达式并显示结果，与 print 效果类似

set <lvalue> = <expr> -- assign new value to field/variable/array element

调试过程中动态修改变量的值

locals -- print all local variables in current stack frame

打印所有局部变量信息

classes -- list currently known classes

列出当前已知的所有类，包括 Java 核心类和加载的用户类。由于打印的信息太多，这个命令反而没有什么用。

class <class id> -- show details of named class

显示指定类的信息，注意包括继承的基类和实现的接口信息

methods <class id> -- list a class's methods

列出指定类所有的方法，包括从基类和接口继承的方法

fields <class id> -- list a class's fields

列出指定类的成员变量信息

stop in <class id>.<method>[(argument_type,...)] -- set a breakpoint in a method

stop at <class id>:<line> -- set a breakpoint at a line

设置一个断点，可以指定函数名或行号，这是 jdb 最有用的命令之一

clear <class id>.<method>[(argument_type,...)] -- clear a breakpoint in a method

clear <class id>:<line> -- clear a breakpoint at a line

clear -- list breakpoints

不带任何参数时，显示所有断点信息。指定参数时可以清除函数或特定行上设置的断点

catch [uncaught|caught|all] <class id>|<class pattern> -- break when specified exception occurs

设置捕获特定的异常。当捕获的异常发生时，jdb 将中止程序的运行，就好像在产生异常的那条语句上设置了断点一样

ignore [uncaught|caught|all] <class id>|<class pattern> -- cancel 'catch' for the specified exception

取消捕获异常

watch [access|all] <class id>.<field name> -- watch access/modifications to a field

设置观察项，在对应的类成员变量被修改或访问时，jdb 中止程序的运行，与设置断点的效果一样

unwatch [access|all] <class id>.<field name> -- discontinue watching access/modifications to a field

取消 watch 的设置

trace methods [thread] -- trace method entry and exit

跟踪指定线程的方法进入和退出

untrace methods [thread] -- stop tracing method entry and exit

取消 trace 的设置

step -- execute current line

单步执行，会进入子函数内部

step up -- execute until the current method returns to its caller

函数内部执行只到返回调用者

stepi -- execute current instruction

执行当前语句，注意一行中可能包含多条语句，step 是执行一行代码

next -- step one line (step OVER calls)

单步执行，但不进入子函数

cont -- continue execution from breakpoint

继续执行只到遇到下一个断点和程序结束

list [line number|method] -- print source code

显示源代码，需要 jdb 启动时设置 sourcepath 参数或使用 use 命令设置源代码路径

use (or sourcepath) [source file path] -- display or change the source path

不带参数时显示当前查找源代码的路径，指定参数可修改当前的源代码路径

classpath -- print classpath info from target VM

显示当前的 Java 类加载路径，感觉此命令有缺陷，无法修改当前的 classpath，只能在 jdb 启动时指定或在环境变量中指定

monitor <command> -- execute command each time the program stops

在每次触发一个断点或程序执行结束时执行 command 指定的命令，如：monitor locals，则在每次程序中止时都字段执行 locals 命令

monitor -- list monitors

列出所有 monitor 执行的 command

unmonitor <monitor#> -- delete a monitor

取消一个 monitor 的设置，参数为 monitor 不断参数时列出的 num

lock <expr> -- print lock info for an object

列出一个对象锁的信息

threadlocks [thread id] -- print lock info for a thread

列出线程的锁信息

!! -- repeat last command

执行上一个命令

<n> <command> -- repeat command n times

重复执行一个 command 命令 n 次

help (or ?) -- list commands

列出所有的命令

version -- print version information

显示版本信息

exit (or quit) -- exit debugger

中止 jdb 调试器的执行

4.6 Jstat

Jstat 主要利用 JVM 内建的指令对 Java 应用程序的资源和性能进行实时的命令行的监控，包括了对 Heap size 和垃圾回收状况的监控。可见，Jstat 是轻量级的、专门针对 JVM 的工具，非常适用。由于 JVM 内存设置较大，图中百分比变化不太明显一个极强的监视 VM 内存工具。可以用来监视 VM 内存内的各种堆和非堆的大小及其内存使用量。

jstat 工具特别强大，有众多的可选项，详细查看堆内各个部分的使用量，以及加载类的数量。使用时，需加上查看进程的进程 id，和所选参数。以下详细介绍各个参数的意义。

- jstat -class pid:显示加载 class 的数量，及所占空间等信息。
- jstat -compiler pid:显示 VM 实时编译的数量等信息。
- jstat -gc pid:可以显示 gc 的信息，查看 gc 的次数，及时间。其中最后五项，分别是 young gc 的次数，young gc 的时间，full gc 的次数，full gc 的时间，gc 的总时间。
- jstat -gccapacity:可以显示，VM 内存中三代（young,old,perm）对象的使用和占用大小，如：PGCMN 显示的是最小 perm 的内存使用量，PGCMX 显示的是 perm 的内存最大使用量，PGC 是当前新生成的 perm 内存占用量，PC 是但前 perm 内存占用量。其他的可以根据这个类推， OC 是 old 内纯的占用量。
- jstat -gcnew pid:new 对象的信息。
- jstat -gcnewcapacity pid:new 对象的信息及其占用量。
- jstat -gcold pid:old 对象的信息。
- jstat -gcoldcapacity pid:old 对象的信息及其占用量。
- jstat -gcpermcapacity pid: perm 对象的信息及其占用量。
- jstat -util pid:统计 gc 信息统计。
- jstat -printcompilation pid:当前 VM 执行的信息。

除了以上一个参数外，还可以同时加上 两个数字，如：jstat -printcompilation 3024 250 6 是每 250 毫秒打印一次，一共打印 6 次，还可以加上-h3 每三行显示一下标题。

语法结构：

Usage: jstat -help|-options

jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]

jstat -gcutil pid

参数解释：

- Options — 选项，我们一般使用 -gcutil 查看 gc 情况
- vmid — VM 的进程号，即当前运行的 java 进程号
- interval – 间隔时间，单位为秒或者毫秒
- count — 打印次数，如果缺省则打印无数次

说明

GC 信息查看：

- S0 — Heap 上的 Survivor space 0 区已使用空间的百分比
- S1 — Heap 上的 Survivor space 1 区已使用空间的百分比
- E — Heap 上的 Eden space 区已使用空间的百分比
- O — Heap 上的 Old space 区已使用空间的百分比
- P — Perm space 区已使用空间的百分比
- YGC — 从应用程序启动到采样时发生 Young GC 的次数
- YGCT - 从应用程序启动到采样时 Young GC 所用的时间(单位秒)
- FGC — 从应用程序启动到采样时发生 Full GC 的次数
- FGCT - 从应用程序启动到采样时 Full GC 所用的时间(单位秒)
- GCT — 从应用程序启动到采样时用于垃圾回收的总时间(单位秒)

4.7 Jmap

Jmap 用于打印出某个 java 进程（使用 pid）内存使用情况，所有‘对象’的情况（如：产生那些对象，及其数量）。

命令：jmap -dump:format=b,file=outfile pid

可以将 pid 进程的内存 heap 输出出来到 outfile 文件里，再配合 MAT 或 Jhat 工具进行分析。

基本参数：

- -dump:[live,]format=b,file=<filename> 使用 hprof 二进制形式,输出 jvm 的 heap 内容到文件=. live 子选项是可选的，假如指定 live 选项，那么只输出活的对象到文件。
- -finalizerinfo 打印正等候回收的对象的信息.
- -heap 打印 heap 的概要信息，GC 使用的算法，heap 的配置及 wise heap 的使用情况。
- -histo[:live] 打印每个 class 的实例数目，内存占用，类全名信息。 VM 的内部类名字开头会加上前缀”*”。 如果 live 子参数加上后,只统计活的对象数量。
- -permstat 打印 classload 和 jvm heap 长久层的信息. 包含每个 classloader 的名字,活泼性,地址,父 classloader 和加载的 class 数量. 另外,内部 String 的数量和占用内存数也会打印出来.
- -F 强迫.在 pid 没有相应的时候使用-dump 或者-histo 参数. 在这个模式下,live 子参数无效.
- -h | -help 打印辅助信息
- -J 传递参数给 jmap 启动的 jvm

说明

除上述工具外，JDK 自带的工具还有 Jconsole, Jhat 等等，有这里不再一一罗列，有兴趣的同学可以自己研究。

4.8 Linux 重要命令

4.8.1 top

top 命令用于监控 Linux 整体性能。

几条实用的命令：

- 查看某进程的线程情况： top –Hp pid <-d 1 –n 1>
- 查看所有进程信息：top -c -b -n1
- 按内存使用百分比排序： top 后按 m

Top 输出结果详解：

```
linux:/opt/OperationCenter/OC/bin # top
top - 15:40:13 up 17 days,  6:00,  5 users,  load average: 21.39, 19.89, 18.41
Tasks: 206 total,  16 running, 190 sleeping,   0 stopped,   0 zombie
Cpu(s): 66.4%us,  2.2%sy,  0.0%ni, 30.6%id,  0.4%wa,  0.0%hi,  0.4%si,  0.0%st
Mem:   8064872k total,  7254628k used,   810244k free,    47656k buffers
Swap:  2104472k total,  1301152k used,   803320k free,  2399608k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
31680 root      20   0 4601m 1.4g  11m S   61 18.1  24:30.72 java
16481 gaussdba  20   0 1175m 530m 526m R   38  6.7   3:16.49 gaussdb
11479 gaussdba  20   0 1175m 514m 510m R   30  6.5   1:15.51 gaussdb
  981 gaussdba  20   0 1176m 524m 520m R   26  6.7   2:27.53 gaussdb
16285 gaussdba  20   0 1176m 513m 509m R   26  6.5   0:54.68 gaussdb
```

- 第一行：分别是当前时间、系统运行时间（格式为时:分）、当前登录用户数、系统负载（即任务队列的平均长度。 三个数值分别为 1 分钟、5 分钟、15 分钟前到现在的平均值。）
- 第二行：分别是进程总数、正在运行的进程数、睡眠的进程数、停止的进程数、僵尸进程数。
- 第三行：分别是用户空间占用 CPU 百分比、内核空间占用 CPU 百分比、用户进程空间内改变过优先级的进程占用 CPU 百分比、空闲 CPU 百分比、等待输入输出的 CPU 时间百分比、硬中断的 CPU 百分比，软中断的 CPU 百分比。
- 第四行：分别是物理内存总量、使用的物理内存总量、空闲内存总量、用作内核缓存的内存量。
- 第五行：分别是交换区总量、使用的交换区总量、空闲交换区总量、缓冲的交换区总量。 内存中的内容被换出到交换区，而后又被换入到内存，但使用过的交换区尚未被覆盖， 该数值即为这些内容已存在于内存中的交换区的大小。相应的内存再次被换出时可不必再对交换区写入。

进程信息如下表

表4-1 进程详情

列名	含义
PID	进程 id
PPID	父进程 id
RUSER	Real user name
UID	进程所有者的用户 id
USER	进程所有者的用户名
GROUP	进程所有者的组名
TTY	启动进程的终端名。不是从终端启动的进程则显示为 ？
PR	优先级
NI	nice 值。负值表示高优先级，正值表示低优先级
P	最后使用的 CPU，仅在多 CPU 环境下有意义
%CPU	上次更新到现在的 CPU 时间占用百分比
TIME	进程使用的 CPU 时间总计，单位秒
TIME+	进程使用的 CPU 时间总计，单位 1/100 秒
%MEM	进程使用的物理内存百分比
VIRT	进程使用的虚拟内存总量，单位 kb。VIRT=SWAP+RES
SWAP	进程使用的虚拟内存中，被换出的大小，单位 kb。
RES	进程使用的、未被换出的物理内存大小，单位 kb。RES=CODE+DATA
CODE	可执行代码占用的物理内存大小，单位 kb
DATA	可执行代码以外的部分(数据段+栈)占用的物理内存大小，单位 kb
SHR	共享内存大小，单位 kb
nFLT	页面错误次数
nDRT	最后一次写入到现在，被修改过的页面数。
S	进程状态。 D=不可中断的睡眠状态 R=运行 S=睡眠 T=跟踪/停止 Z=僵尸进程
COMMAND	命令名/命令行

WCHAN	若该进程在睡眠，则显示睡眠中的系统函数名
Flags	任务标志，参考 sched.h

4.8.2 lsof

lsof 是一个列出当前系统打开文件的工具。在定位句柄使用、文件读写、磁盘 IO 等问题时比较常用。

常用的参数列表

- lsof filename 显示打开指定文件的所有进程
- lsof -a 表示两个参数都必须满足时才显示结果
- lsof -c string 显示 COMMAND 列中包含指定字符的进程所有打开的文件
- lsof -u username 显示所属 user 进程打开的文件
- lsof -g gid 显示归属 gid 的进程情况
- lsof +d /DIR/ 显示目录下被进程打开的文件
- lsof +D /DIR/ 同上，但是会搜索目录下的所有目录，时间相对较长
- lsof -d FD 显示指定文件描述符的进程
- lsof -n 不将 IP 转换为 hostname，缺省是不加上-n 参数
- lsof -i 用以显示符合条件的进程情况

📖 说明

查看某进程打开句柄数量：lsof | grep pid | wc –l

查看所有进程句柄数量，从大到小排序：lsof -n |awk '{print $2}'|sort|uniq -c |sort -nr|more

第一列是句柄数量，第二列是进程号。

4.8.3 netstat

Netstat 命令用于显示各种网络相关信息，如网络连接，路由表，接口状态 (Interface Statistics)，masquerade 连接，多播成员 (Multicast Memberships) 等等。

常见参数

- -a (all)显示所有选项，默认不显示 LISTEN 相关
- -t (tcp)仅显示 tcp 相关选项
- -u (udp)仅显示 udp 相关选项
- -n 拒绝显示别名，能显示数字的全部转化成数字。
- -l 仅列出有在 Listen (监听) 的服务状态
- -p 显示建立相关链接的程序名
- -r 显示路由信息，路由表
- -e 显示扩展信息，例如 uid 等
- -s 按各个协议进行统计
- -c 每隔一个固定时间，执行该 netstat 命令。

实用命令：

- 列出所有 tcp 端口 netstat –at
- 只列出所有监听 tcp 端口 netstat –lt
- 显示所有端口的统计信息 netstat –s
- 查看进程信息 netstat -p
- 显示核心路由信息 netstat –r
- 输出中不显示主机，端口和用户名 netstat -n
- 查看连接某服务端口最多的的 IP 地址 netstat -nat | grep "192.168.1.15:22" |awk '{print $5}'|awk -F: '{print $1}'|sort|uniq -c|sort -nr|head -20

说明

LISTEN 和 LISTENING 的状态只有用-a 或者-l 才能看到

netstat -p 可以与其它开关一起使用，就可以添加“PID/进程名称”到 netstat 输出中，这样 debugging 的时候可以很方便的发现特定端口运行的程序。

4.8.4 strace

strace 用于跟踪系统调用和信号。在定位 IO、CPU 性能问题时比较有用。

命令：

strace -ttTp 进程号 –ff –o 文件名

图4-7 strace 输出

```
[pid  2268] 16:17:01.054848 futex(0x7fa6ca7f3dc0, FUTEX_WAKE_PRIVATE, 1 <unfinished ...>
[pid   885] 16:17:01.054876 <... futex resumed> ) = 1 <0.000265>
[pid   873] 16:17:01.054902 <... futex resumed> ) = 0 <0.233493>
[pid   761] 16:17:01.054930 setsockopt(857, SOL_TCP, TCP_NODELAY, [1], 4 <unfinished ...>
[pid   753] 16:17:01.054964 <... futex resumed> ) = 1 <0.000295>
[pid 31688] 16:17:01.054991 <... futex resumed> ) = 0 <0.000294>
[pid 29383] 16:17:01.055070 futex(0x7fa6c877e928, FUTEX_WAIT_PRIVATE, 2, NULL <unfinished ...>
[pid  6199] 16:17:01.055095 futex(0x7fa6d2186c54, FUTEX_WAIT_PRIVATE, 301, NULL <unfinished ...>
[pid  3985] 16:17:01.055117 <... futex resumed> ) = -1 ETIMEDOUT (Connection timed out) <0.010064>
[pid  3985] 16:17:01.055144 futex(0x1d73928, FUTEX_WAKE_PRIVATE, 1) = 0 <0.000015>
[pid  2698] 16:17:01.055184 <... futex resumed> ) = -1 ETIMEDOUT (Connection timed out) <0.010028>
[pid  3985] 16:17:01.055209 futex(0x1d70a54, FUTEX_WAIT_PRIVATE, 3249, NULL <unfinished ...>
[pid  2698] 16:17:01.055236 futex(0x7fa6d1674a28, FUTEX_WAKE_PRIVATE, 1) = 0 <0.000015>
```

从上图可以看出每个线程每个系统调用的执行时间以及执行结果，以辅助定位问题。

参数说明：

- -c 统计每一系统调用的所执行的时间,次数和出错的次数等。
- -d 输出 strace 关于标准错误的调试信息。
- -f 跟踪由 fork 调用所产生的子进程。
- -ff 如果提供-o filename,则所有进程的跟踪结果输出到相应的 filename。pid 中,pid 是各进程的进程号。
- -F 尝试跟踪 vfork 调用。在-f 时,vfork 不被跟踪。
- -h 输出简要的帮助信息。

- -i 输出系统调用的入口指针。
- -q 禁止输出关于脱离的消息。
- -r 打印出相对时间关于,,每一个系统调用。
- -t 在输出中的每一行前加上时间信息。
- -tt 在输出中的每一行前加上时间信息,微秒级。
- -ttt 微秒级输出,以秒表示时间。
- -T 显示每一调用所耗的时间。
- -v 输出所有的系统调用。一些调用关于环境变量,状态,输入输出等调用由于使用频繁,默认不输出。
- -V 输出 strace 的版本信息。
- -x 以十六进制形式输出非标准字符串
- -xx 所有字符串以十六进制形式输出。
- -a column 设置返回值的输出位置。默认为 40。
- -e expr 指定一个表达式,用来控制如何跟踪。格式如下: [qualifier=][!]value1[,value2]。。。qualifier 只能是 trace,abbrev,verbose,raw,signal,read,write 其中之一。value 是用来限定的符号或数字。默认的 qualifier 是 trace。感叹号是否定符号。例如: -eopen 等价于 -e trace=open,表示只跟踪 open 调用。而-etrace!=open 表示跟踪除了 open 以外的其他调用。有两个特殊的符号 all 和 none。注意有些 shell 使用!来执行历史记录里的命令,所以要使用\\。
- -e trace=set
- 只跟踪指定的系统调用。例如:-e trace=open,close,rean,write 表示只跟踪这四个系统调用。默认的为 set=all。
- -e trace=file
- 只跟踪有关文件操作的系统调用。
- -e trace=process 只跟踪有关进程控制的系统调用。
- -e trace=network 跟踪与网络有关的所有系统调用。
- -e strace=signal 跟踪所有与系统信号有关的系统调用
- -e trace=ipc 跟踪所有与进程通讯有关的系统调用
- -e abbrev=set 设定 strace 输出的系统调用的结果集。-v 等与 abbrev=none。默认为 abbrev=all。
- -e raw=set 将指定的系统调用的参数以十六进制显示。
- -e signal=set 指定跟踪的系统信号。默认为 all。如 signal=!SIGIO(或者 signal=!io),表示不跟踪 SIGIO 信号。
- -e read=set 输出从指定文件中读出的数据。例如: -e read=3,5
- -e write=set 输出写入到指定文件中的数据。
- -o filename 将 strace 的输出写入文件 filename
- -p pid 跟踪指定的进程 pid。
- -s strsize 指定输出的字符串的最大长度。默认为 32。文件名一直全部输出。
- -u username 以 username 的 UID 和 GID 执行被跟踪的命令。

说明

对于不打日志，又抓不到堆栈的线程，不妨从这里抓下看看。

对于底层系统调用的错误，从这里可以抓到充足的证据。

strace -ttTp 进程号 –ff –o 文件名 会生成文件名.线程名文件，如果是-f 的话，会输出到一个文件中。

Strace 信息解读需要对 linux、C 语言等有一定基础。有兴趣的同学可以再深入了解。

4.8.5 kill

Kill 一般用于杀进程，但 kill -3 pid 可以输出该进程的线程堆栈。

说明

输出的线程堆栈信息，不同的系统会保存到不同的地方。OC 使用的 iemp 平台会输出到 var/iemp/log/stack.log 中。

4.8.6 tcpdump

tcpdump 是一个免费的网络分析工具。

如：tcpdump -i any -s 0 -w /home/hdd/dump.cap

参数：

- -a 将网络地址和广播地址转变成名字；
- -d 将匹配信息包的代码以人们能够理解的汇编格式给出；
- -dd 将匹配信息包的代码以 c 语言程序段的格式给出；
- -ddd 将匹配信息包的代码以十进制的形式给出；
- -e 在输出行打印出数据链路层的头部信息；
- -f 将外部的 Internet 地址以数字的形式打印出来；
- -l 使标准输出变为缓冲行形式；
- -n 不把网络地址转换成名字；
- -t 在输出的每一行不打印时间戳；
- -v 输出一个稍微详细的信息，例如在 ip 包中可以包括 ttl 和服务类型的信息；
- -vv 输出详细的报文信息；
- -c 在收到指定的包的数目后，tcpdump 就会停止；
- -F 从指定的文件中读取表达式,忽略其它的表达式；
- -i 指定监听的网络接口；
- -r 从指定的文件中读取包(这些包一般通过-w 选项产生)；
- -w 直接将包写入文件中，并不分析和打印出来；
- -T 将监听到的包直接解释为指定的类型的报文，常见的类型有 rpc （远程过程调用）和 snmp（简单网络管理协议；）

说明

A 想要截获所有 210.27.48.1 的主机收到的和发出的所有的数据包：

#tcpdump host 210.27.48.1

B 想要截获主机 210.27.48.1 和主机 210.27.48.2 或 210.27.48.3 的通信，使用命令：(在命令行中使用括号时，一定要添加'\')

#tcpdump host 210.27.48.1 and \ (210.27.48.2 or 210.27.48.3 \)

C 如果想要获取主机 210.27.48.1 除了和主机 210.27.48.2 之外所有主机通信的 ip 包，使用命令：

#tcpdump ip host 210.27.48.1 and ! 210.27.48.2

D 如果想要获取主机 210.27.48.1 接收或发出的 telnet 包，使用如下命令：

#tcpdump tcp port 23 host 210.27.48.1

E 对本机的 udp 123 端口进行监视 123 为 ntp 的服务端口

tcpdump udp port 123

F 系统将只对名为 hostname 的主机的通信数据包进行监视。主机名可以是本地主机，也可以是网络上的任何一台计算机。下面的命令可以读取主机 hostname 发送的所有数据：

#tcpdump -i eth0 src host hostname

G 下面的命令可以监视所有送到主机 hostname 的数据包：

#tcpdump -i eth0 dst host hostname

H 我们还可以监视通过指定网关的数据包：

#tcpdump -i eth0 gateway Gatewayname

I 如果你还想监视编址到指定端口的 TCP 或 UDP 数据包，那么执行以下命令：

#tcpdump -i eth0 host hostname and port 80

J 如果想要获取主机 210.27.48.1 除了和主机 210.27.48.2 之外所有主机通信的 ip 包，使用命令：

#tcpdump ip host 210.27.48.1 and ! 210.27.48.2

K 想要截获主机 210.27.48.1 和主机 210.27.48.2 或 210.27.48.3 的通信，使用命令：

#tcpdump host 210.27.48.1 and \ (210.27.48.2 or 210.27.48.3 \)

L 如果想要获取主机 210.27.48.1 除了和主机 210.27.48.2 之外所有主机通信的 ip 包，使用命令：

#tcpdump ip host 210.27.48.1 and ! 210.27.48.2

M 如果想要获取主机 210.27.48.1 接收或发出的 telnet 包，使用如下命令：

#tcpdump tcp port 23 host 210.27.48.1

4.8.7 iostat

4.8.7.1 安装

找到 sysstat 包，执行命令：

rpm -ivh sysstat-8.1.5-7.9.56.x86_64.rpm

4.8.7.2 监控

 # iostat -x

Linux 2.6.32.12-0.7-default (linux-162-66-15-107) 02/12/14 _x86_64_

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           7.82    0.00    1.58    0.14    0.00   90.46

Device:            rrqm/s   wrqm/s     r/s     w/s    rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda                  0.23     1.17    0.03    1.22      2.09     19.22     17.00      0.00    3.70   1.42   0.18
sdb                  0.04     8.08    0.08    4.47     12.60     99.56     24.67      0.06   13.99   0.51   0.23
```

- rrqm/s: 每秒进行 merge 的读操作数目。即 delta(rmerge)/s
- wrqm/s: 每秒进行 merge 的写操作数目。即 delta(wmerge)/s
- r/s: 每秒完成的读 I/O 设备次数。即 delta(rio)/s
- w/s: 每秒完成的写 I/O 设备次数。即 delta(wio)/s
- rsec/s: 每秒读扇区数。即 delta(rsect)/s
- wsec/s: 每秒写扇区数。即 delta(wsect)/s
- rkB/s: 每秒读 K 字节数。是 rsect/s 的一半，因为每扇区大小为 512 字节。
- wkB/s: 每秒写 K 字节数。是 wsect/s 的一半。
- avgrq-sz: 平均每次设备 I/O 操作的数据大小 (扇区)。即 delta(rsect+wsect)/delta(rio+wio)
- avgqu-sz: 平均 I/O 队列长度。即 delta(aveq)/s/1000 (因为 aveq 的单位为毫秒)。
- await: 平均每次设备 I/O 操作的等待时间 (毫秒)。即 delta(ruse+wuse)/delta(rio+wio)
- svctm: 平均每次设备 I/O 操作的服务时间 (毫秒)。即 delta(use)/delta(rio+wio)
- %util: 一秒中有百分之多少的时间用于 I/O 操作，或者说一秒中有多少时间 I/O 队列是非空的。即 delta(use)/s/1000 (因为 use 的单位为毫秒)

说明

如果 %util 接近 100%，说明产生的 I/O 请求太多，I/O 系统已经满负荷，该磁盘可能存在瓶颈。

IO 高时(不一直是 100%)，如果跟业务特征是对得上的，而且 await 和 svctm 差别不大，不用太过关注。

svctm 一般要小于 await (因为同时等待的请求的等待时间被重复计算了)，svctm 的大小一般和磁盘性能有关，CPU/内存的负荷也会对其有影响，请求过多也会间接导致 svctm 的增加。await 的大小一般取决于服务时间(svctm) 以及 I/O 队列的长度和 I/O 请求的发出模式。如果 svctm 比较接近 await，说明 I/O 几乎没有等待时间；如果 await 远大于 svctm，说明 I/O 队列太长，应用得到的响应时间变慢，如果响应时间超过了用户可以容许的范围，这时可以考虑更换更快的磁盘，调整内核 elevator 算法，优化应用，或者升级 CPU。

队列长度(avgqu-sz)也可作为衡量系统 I/O 负荷的指标，但由于 avgqu-sz 是按照单位时间的平均值，所以不能反映瞬间的 I/O 洪水。

4.8.8 pmap

命令：usage: pmap [options] pid

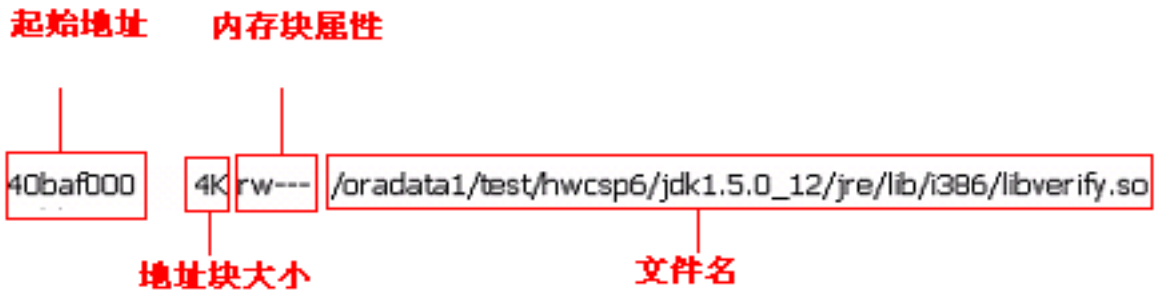
- -d, --device display offset and device numbers
- -q, --quiet hide header and memory statistics
- -V, --version display version information

-h, --help display this help

图4-8 范例

```
OCHost1:/opt/OperationCenter/AppBase/OC/AppBase/var/iemp/log # pmap -d 12020
12020: java
START               SIZE     RSS     PSS   DIRTY    SWAP PERM OFFSET   DEVICE MAPPING
0000000000400000      4K      0K      0K      0K      0K r-xp 0000000000000000 08:11  /opt/OperationCenter/AppBase/OC/mttools/jre/bin/java
0000000000600000      4K      4K      4K      4K      0K rw-p 0000000000000000 08:11  /opt/OperationCenter/AppBase/OC/mttools/jre/bin/java
0000000000601000  52400K  51620K  51620K  51620K      0K rw-p 0000000000000000 00:00  [heap]
00000000c8000000 917504K 210756K 210756K 210756K      0K rw-p 0000000000000000 00:00  [anon]
00007f0fc4000000  22844K  14260K  14260K  14260K      0K rw-p 0000000000000000 00:00  [anon]
00007f0fc564f000  42692K      0K      0K      0K      0K ---p 0000000000000000 00:00  [anon]
00007f0fc8283000     12K      0K      0K      0K      0K ---p 0000000000000000 00:00  [anon]
00007f0fc8286000   1016K     24K     24K     24K      0K rw-p 0000000000000000 00:00  [anon]
00007f0fc8384000     12K      0K      0K      0K      0K ---p 0000000000000000 00:00  [anon]
00007f0fc8387000   1016K     24K     24K     24K      0K rw-p 0000000000000000 00:00  [anon]
00007f0fc8485000     12K      0K      0K      0K      0K ---p 0000000000000000 00:00  [anon]
00007f0fc8488000   1016K     24K     24K     24K      0K rw-p 0000000000000000 00:00  [anon]
00007f0fc8586000     12K      0K      0K      0K      0K ---p 0000000000000000 00:00  [anon]
00007f0fc8589000   1016K     24K     24K     24K      0K rw-p 0000000000000000 00:00  [anon]
00007f0fc8687000     12K      0K      0K      0K      0K ---p 0000000000000000 00:00  [anon]
00007f0fc868a000   1016K    100K    100K    100K      0K rw-p 0000000000000000 00:00  [anon]
```

图4-9 每列含义



4.9 BTrace

当环境没开 DEBUG 端口，或者一个临时对象抓不到调用栈时，BTrace 是个很好的选择!

BTrace 是 sun 提供的 Java 动态跟踪工具，相当于 DTrace（一个 Solari 的工具，用于跟踪内核调用等等） for Java， BTrace 的工作的基本原理是把跟踪的代码动态替换到被跟踪的 Java 程序内（通过动态的修改运行时的 java 字节码，在运行时代码中插入监控动作，输出相关信息），它借助动态字节码注入技术 ，实现优雅且功能强大。

官方文档：

- BTrace 用户指南 http://kenai.com/projects/btrace/pages/UserGuide
- BTrace 开发者指南 http://kenai.com/projects/btrace/pages/DeveloperGuide

4.9.1 下载部署

下载地址：https://kenai.com/projects/btrace/downloads/directory/releases/release-1.2.4

将 btrace-bin.tar.gz 上传 linux 服务器，解压到任意目录。（本文以 LINUX 为例）

前提：环境有可用的 JDK

4.9.2 编写脚本

本文以调试 OC 图片管理中查询图片为例。

4.9.2.1 跟踪函数入参和返回值

```
@BTrace
public class TraceVarRtn
{
    @OnMethod(clazz = "com.xxx.oc.as.portal.service.ImageServiceImpl", method = "queryImageInfo", location = @Location(Kind.RETURN))
    public static void traceExecute(Map<String, Object> conditionMap, @Return List<Map<String, Object>> result)
    {
        BTraceUtils.println("Call ImageServiceImpl.queryImageInfo");
        BTraceUtils.println(BTraceUtils.strcat("Input paramter size is:", BTraceUtils.str(BTraceUtils.size(conditionMap))));
        BTraceUtils.println(BTraceUtils.strcat("Return value size is:", BTraceUtils.str(BTraceUtils.size(result))));
    }
}
```

4.9.2.2 跟踪函数运行时间

@BTrace

```
public class TraceExecuteTime
{
    @TLS
    static long beginTime;

    @OnMethod(clazz = "com.xxx.oc.as.portal.service.ImageServiceImpl", method =
"queryImageInfo")
    public static void traceExecuteBegin()
    {
        beginTime = BTraceUtils.timeMillis();
    }

    @OnMethod(clazz = "com.xxx.oc.as.portal.service.ImageServiceImpl", method =
"queryImageInfo", location = @Location(Kind.RETURN))
    public static void traceExecute(Map<String, Object> conditionMap, @Return
List<Map<String, Object>> result)
    {

BTraceUtils.println(BTraceUtils.strcat(BTraceUtils.strcat("ImageServiceImpl.queryImageInf
o time is:", BTraceUtils.str(BTraceUtils.timeMillis() - beginTime)), "ms"));
    }
}
```

4.9.2.3 跟踪函数调用栈

```
import static com.sun.btrace.BTraceUtils.*;
import com.sun.btrace.annotations.*;
@BTrace
public class TraceStack
{
    @OnMethod(clazz = "com.xxx.oc.as.portal.service.ImageServiceImpl", method =
"queryImageInfo")
    public static void traceExecute()
    {
        println("The stack of ImageServiceImpl.queryImageInfo :");
        jstack();
    }
```

```
}
```

4.9.2.4 按行跟踪

```
@BTrace
public class TraceLine
{
    @OnMethod(clazz = "com.xxx.oc.as.portal.service.ImageServiceImpl", location = @Location(value = Kind.LINE, line = 150))
    public static void traceExecute(@ProbeClassName String pcn, @ProbeMethodName String pmn, int line)
    {
        println(strcat(strcat(strcat("Call ", pcn), "."), pmn));
    }
}
```

4.9.3 运行调试

将编写好的脚本上传到 BTrace 的 bin 目录，执行以下命令：

btrace　java 进程号　脚本文件名

```
OCHost1:/opt/btrace/bin # ./btrace 5321 TraceVarRtn.java
Call ImageServiceImpl.queryImageInfo
Input paramter size is:0
Return value size is:5
```

小心

1 执行 btrace 需要切到启动该 JAVA 进程的用户下执行

2 跟踪构造函数，方法名写 <init>

4.10 Jprofiler

Jprofiler 对热点监控做的非常好，CPU、内存、SQL 的热点都可以监控到。这样一个版本转测后，通过监控热点可以有效的提前识别性能瓶颈点，提早预防解决。

该软件唯一不好的地方是需要在业务软件上装个 Agent，一定程度上影响了业务进程本身的性能。

4.10.1 下载部署

共享地址：\\10.146.155.87\share\05-常用软件\jprofiler

客户端安装 jprofiler_windows_8_0_7.exe。

服务端安装：jprofiler_linux_8_0_7.sh

安装前需要设置 JAVA_HOME 环境变量

- 在~/.bash_profile 中添加
- export JAVA_HOME=/opt/jdk
- export PATH=${JAVA_HOME}/bin:${PATH}
- export CLASSPATH=./JAVA_HOME/lib;$JAVA_HOME/jre/lib:${CLASSPATH}
- 后重新 SSH 登录生效。

如果 JAVA 程序运行用户与安装用户非同一个用户，安装后更改下属主和权限。

4.10.2 运行连接

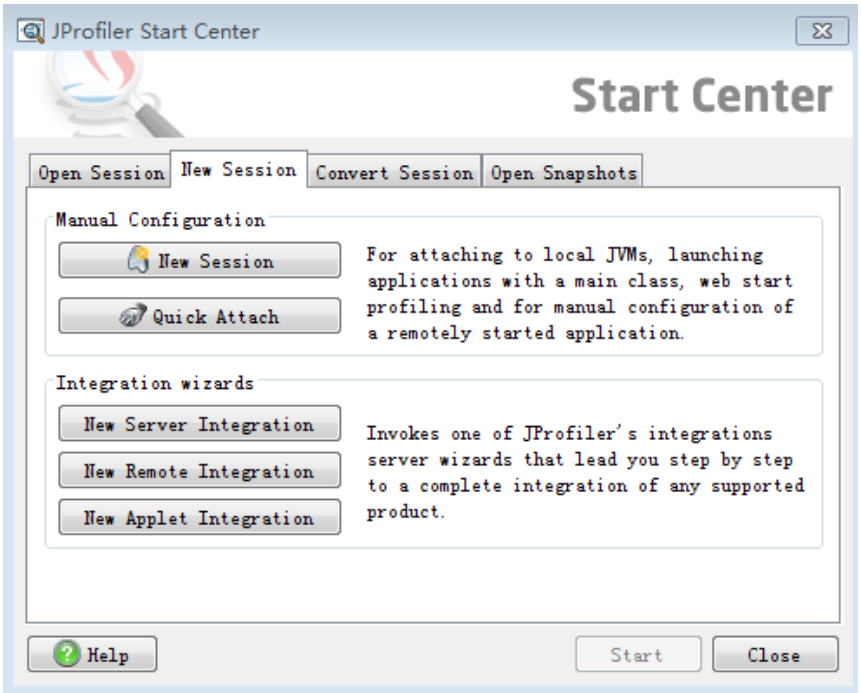
服务端

在 JVM 参数中添加 JVM_OPT="$JVM_OPT
-agentpath:/opt/jprofiler8/bin/linux-x64/libjprofilerti.so=nowait,port=8849"

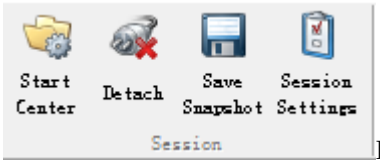
重启进程

客户端

- 运行 Jprofile，打开 Start Center，如下图



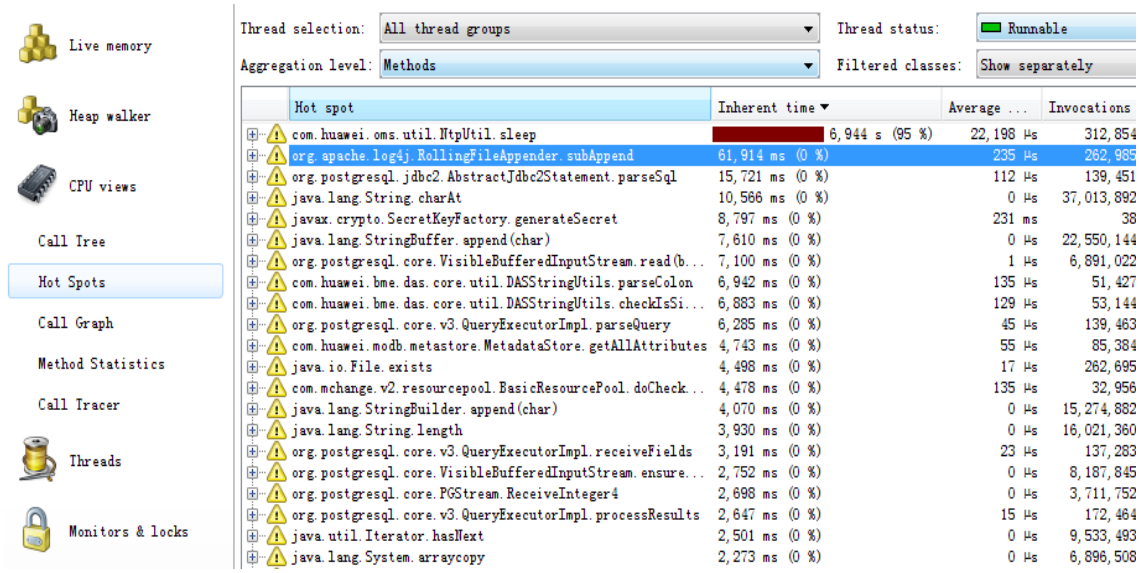
- 首次监控，选择“New Remote Integration”。 以后”Open Session”中会记录之前的登录信息，可直接从中打开。
- 打开连接向导后，“Local or remote”页选择“On a remote computer”，“Linux x86/amd64”。
- “Profiled JVM”页选择“Oracle”，“1.7.0”，“hotspot”，勾上“64bit jvm”。
- “Remote Address”页填写监控服务器的 IP 地址。
- “Remote Installation Directory”页填写服务端安装路径。“/opt/jprofiler8”
- “Chose profiling port”页填写 JVM 参数中的 PORT，默认 8849
- 其它页按默认下一步。

点 

Detach 断开连接。

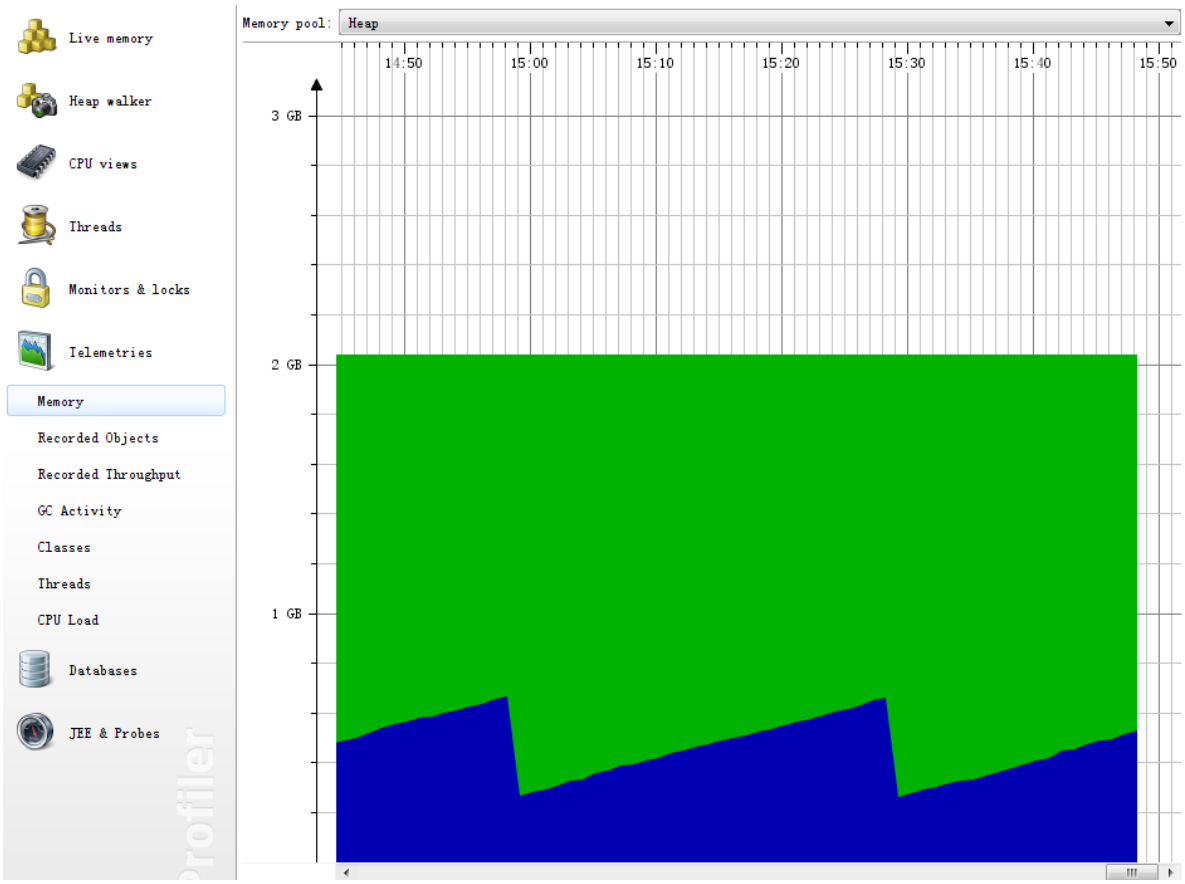
4.10.3 监控

4.10.3.1 监控热点（CPU）

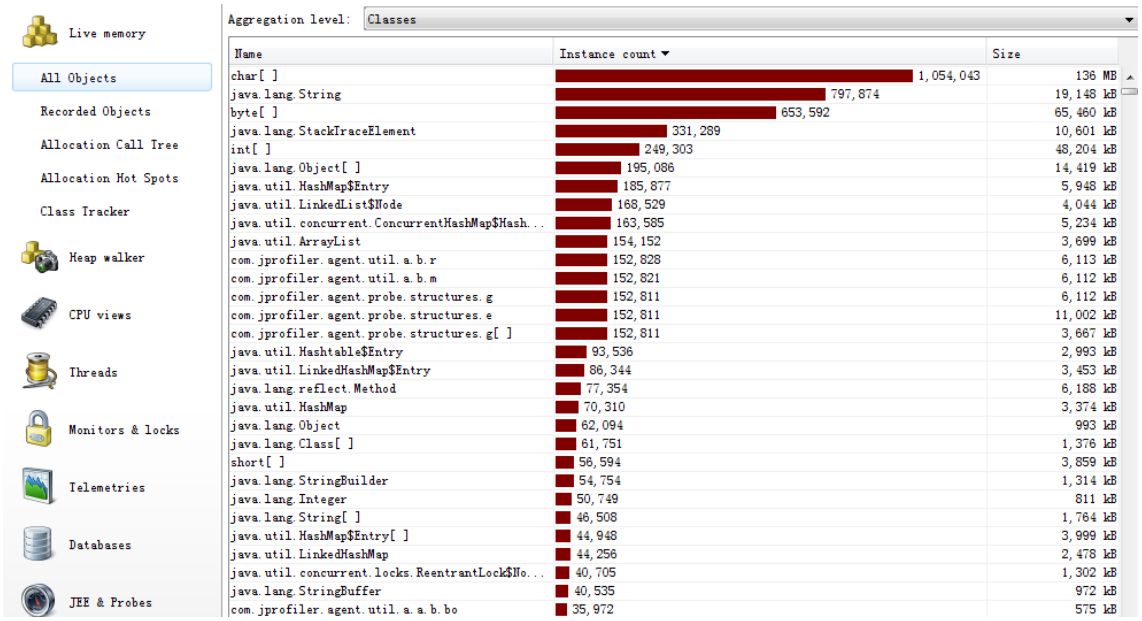


选择 CPU views/Hot Spots，要以观察到当前系统运行的热点函数（CPU 消耗在哪），以及其运行堆栈、时长等信息。

4.10.3.2 监控内存

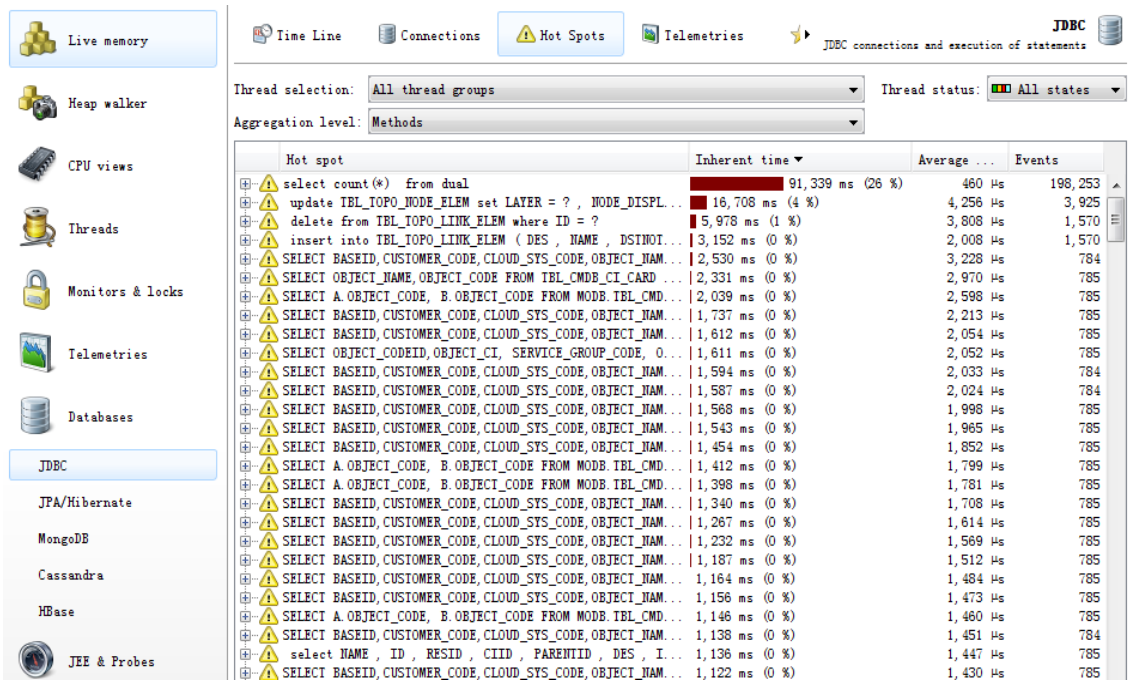


选择“Telemetries/Memory/Heap”可以监控堆内存使用/回收情况，非堆内存通过上面的下拉框选择后查看。



选择“Live memory/All Objects”可以查看所有对象内存使用情况，如果发现某对象在不断增长，可通过 Call Tree 等手段看是否有内存泄漏。

4.10.3.3 监控 SQL



选择“Databases/JDBC/Hot Spots”可以监控到所有运行的 SQL 执行时间、频率等信息。

A 参考文献

童志刚 《JVM 内存管理》

李林锋 《Java 性能问题定位参考手册》

胡志云 《Java 性能问题定位手段优化优秀实践》

贺忆东 《使用 JDB 调试 Java 程序》