

# 性能分析之-- JAVA Thread Dump 分析综述

最近在做性能测试，需要对线程堆栈进行分析，在网上收集了一些资料，学习完后，将相关知识整理在一起，输出文章如下。

## 一、Thread Dump 介绍

### 1.1 什么是 Thread Dump ?

Thread Dump 是非常有用的诊断 Java 应用问题的工具。每一个 Java 虚拟机都有及时生成所有线程在某一点状态的 thread-dump 的能力，虽然各个 Java 虚拟机打印的 thread dump 略有不同，但是大多都提供了当前活动线程的快照，及 JVM 中所有 Java 线程的堆栈跟踪信息，堆栈信息一般包含完整的类名及所执行的方法，如果可能的话还有源代码的行数。

### 1.2 Thread Dump 特点

1. 能在各种操作系统下使用
2. 能在各种 Java 应用服务器下使用
3. 可以在生产环境下使用而不影响系统的性能
4. 可以将问题直接定位到应用程序的代码行上

### 1.3 Thread Dump 能诊断的问题

1. 查找内存泄露，常见的是程序里 load 大量的数据到缓存；
2. 发现死锁线程；

## 1.4 如何抓取 Thread Dump

一般当服务器挂起,崩溃或者性能底下时,就需要抓取服务器的线程堆栈(Thread Dump)用于后续的分析. 在实际运行中, 往往一次 dump 的信息, 还不足以确认问题。为了反映线程状态的动态变化, 需要接连多次做 threaddump, 每次间隔 10-20s, 建议至少产生三次 dump 信息, 如果每次 dump 都指向同一个问题, 我们才确定问题的典型性。

有很多方式可用于获取 ThreadDump, 下面列出一部分获取方式：

操作系统命令获取 ThreadDump:

*Windows:*

1.转向服务器的标准输出窗口并按下 Control + Break 组合键, 之后需要将线程堆栈复制到文件中；

*UNIX/ Linux :*

首先查找到服务器的进程号(process id), 然后获取线程堆栈.

1. ps -ef | grep java

2. kill -3 <pid>

注意：一定要谨慎, 一步不慎就可能让服务器进程被杀死。kill -9 命令会杀死进程。

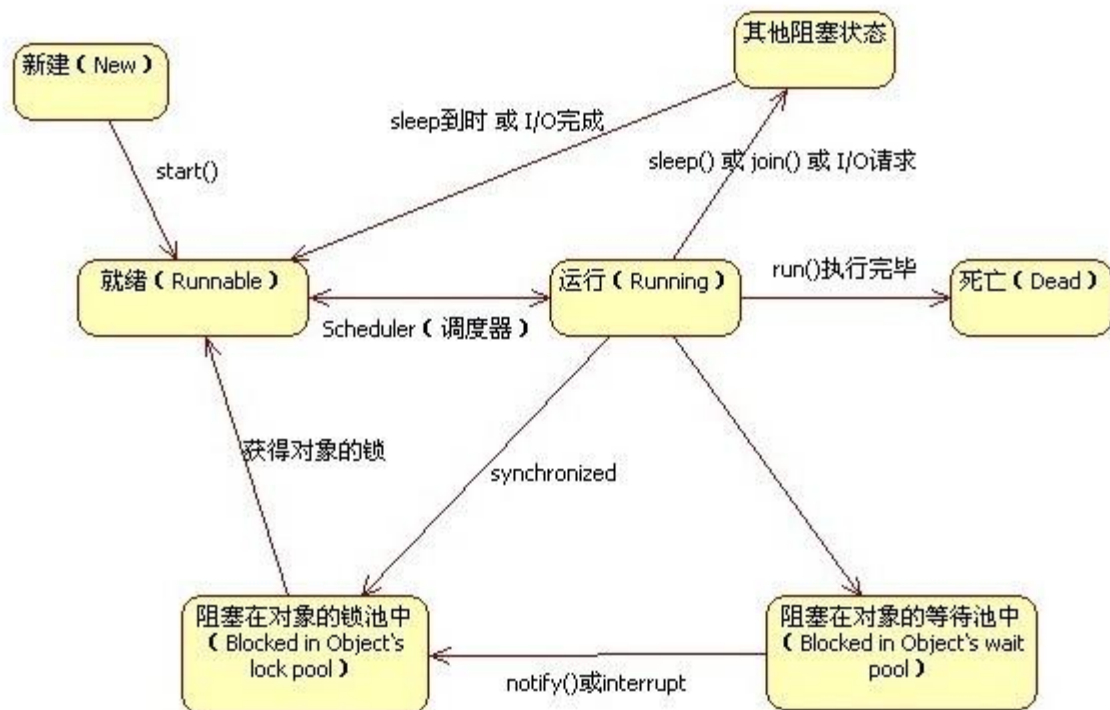
JVM 自带的工具获取线程堆栈:

JDK 自带命令行工具获取 PID , 再获取 ThreadDump:

1. jps 或 ps -ef|grepjava (获取 PID)

2. jstack [-l ]<pid> | tee -a jstack.log (获取 ThreadDump)

## 二、java 线程的状态转换介绍(为后续分析做准备)



### 2.1 新建状态 ( New )

用 new 语句创建的线程处于新建状态，此时它和其他 Java 对象一样，仅仅在堆区中被分配了内存。

### 2.2 就绪状态 ( Runnable )

当一个线程对象创建后，其他线程调用它的 start()方法，该线程就进入就绪状态，Java 虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得 CPU 的使用权。

### 2.3 运行状态 ( Running )

处于这个状态的线程占用 CPU，执行程序代码。只有处于就绪状态的线程才有机会转到运行状态。

## 2.4 阻塞状态 ( Blocked )

阻塞状态是指线程因为某些原因放弃 CPU，暂时停止运行。当线程处于阻塞状态时，Java 虚拟机不会给线程分配 CPU。直到线程重新进入就绪状态，它才有机会转到运行状态。

阻塞状态可分为以下 3 种：

1 ) 位于对象等待池中的阻塞状态 ( Blocked in object' s wait pool ) ：当线程处于运行状态时，如果执行了某个对象的 wait()方法，Java 虚拟机就会把线程放到这个对象的等待池中，这涉及到“线程通信”的内容。

2 ) 位于对象锁池中的阻塞状态 ( Blocked in object' s lock pool ) ：当线程处于运行状态时，试图获得某个对象的同步锁时，如果该对象的同步锁已经被其他线程占用，Java 虚拟机就会把这个线程放到这个对象的锁池中，这涉及到“线程同步”的内容。

3 ) 其他阻塞状态 ( Otherwise Blocked ) ：当前线程执行了 sleep()方法，或者调用了其他线程的 join()方法，或者发出了 I/O 请求时，就会进入这个状态。

## 2.5 死亡状态 ( Dead )

当线程退出 run()方法时，就进入死亡状态，该线程结束生命周期。

## 三、Thread Dump 分析

通过前面 1.4 部分的方法，获取 Thread Dump 信息后，对其进行分析；

### 3.1 首先介绍一下 Thread Dump 信息的各个部分

头部信息：

时间 , jvm 信息

2011-11-02 19:05:06

Full thread dump Java HotSpot(TM) Server VM (16.3-b01 mixed mode):

### 线程 info 信息块 :

1. "Timer-0" daemon prio=10tid=0xac190c00 nid=0xae7d0000 in Object.wait()  
[0xae7d0000]
2. java.lang.Thread.State: TIMED\_WAITING (on object monitor)
3. atjava.lang.Object.wait(Native Method)
4. -waiting on <0xb3885f60> (a java.util.TaskQueue) ###继续 wait
5. atjava.util.TimerThread.mainLoop(Timer.java:509)
6. -locked <0xb3885f60> (a java.util.TaskQueue) ###已经 locked
7. atjava.util.TimerThread.run(Timer.java:462)

\* 线程名称 : Timer-0

\* 线程类型 : daemon

\* 优先级: 10 , 默认是 5

\* jvm 线程 id : tid=0xac190c00 , jvm 内部线程的唯一标识 ( 通过 java.lang.Thread.getId()获取 , 通常用自增方式实现。 )

\* 对应系统线程 id ( NativeThread ID ) : nid=0xae7d0000 , 和 top 命令查看的线程 pid 对应 , 不过一个是 10 进制 , 一个是 16 进制。 ( 通过命令 : top -H -p pid , 可以查看该进程的所有线程信息 )

\* **线程状态** : in Object.wait().

\* 起始栈地址 : [0xae77d000]

\* **Java thread stack trace** : 是上面 2-7 行的信息。到目前为止这是最重要的数据，Java stack trace 提供了大部分信息来精确定位问题根源。

对于 thread dump 信息，主要关注的是线程的状态和其执行堆栈。现在针对这两个重点部分进行讲解：

### 1 ) Java thread stack trace 详解：

堆栈信息应该逆向解读：程序先执行的是第 7 行，然后是第 6 行，依次类推。

```
- locked <0xb3885f60> (a java.util.ArrayList)
```

```
- waiting on <0xb3885f60> (a java.util.ArrayList)
```

也就是说对象先上锁，锁住对象 0xb3885f60，然后释放该对象锁，进入 waiting 状态。

为啥会出现这样的情况呢？看看下面的 java 代码示例，就会明白：

```
synchronized(obj) {  
  
    .....  
  
    obj.wait();  
  
    .....  
  
}
```

在堆栈的第一行信息中，进一步标明了线程在代码级的状态，例如：

```
java.lang.Thread.State: TIMED_WAITING (parking)
```

解释如下：

### **|blocked|**

This thread tried to enter asynchronized block, but the lock was taken by another thread. This thread isblocked until the lock gets released.

### **|blocked (on thin lock)|**

This is the same state asblocked, but the lock in question is a thin lock.

### **|waiting|**

This thread calledObject.wait() on an object. The thread will remain there until some otherthread sends a notification to that object.

### **|sleeping|**

This thread calledjava.lang.Thread.sleep().

### **|parked|**

This thread calledjava.util.concurrent.locks.LockSupport.park().

### **|suspended|**

The thread's execution wassuspended by java.lang.Thread.suspend() or a JVMTI agent call.

## **2) 线程状态详解：**

### **Runnable**

\_The thread is either running or ready to run when it gets its CPU turn.\_

### **Wait on condition**

\_The thread is either sleeping or waiting to be notified by another

thread.\_

该状态出现在线程等待某个条件的发生或者 sleep。具体是什么原因，可以结合 stacktrace 来分析。最常见的情况是线程在等待网络的读写，比如当网络数据没有准备好读时，线程处于这种等待状态，而一旦有数据准备好读之后，线程会重新激活，读取并处理数据。在 Java 引入 New IO 之前，对于每个网络连接，都有一个对应的线程来处理网络的读写操作，即使没有可读写的数  
据，线程仍然阻塞在读写操作上，这样有可能造成资源浪费，而且给操作系统的线程调度也带来压力。在 New IO 里采用了新的机制，编写的服务器程序的性能和可扩展性都得到提高。

如果发现有大量的线程都处在 Wait on condition，从线程 stack 看，正等待网络读写，这可能是一个网络瓶颈的征兆。因为网络阻塞导致线程无法执行。一种情况是网络非常忙，几乎消耗了所有的带宽，仍然有大量数据等待网络读写；另一种情况也可能是网络空闲，但由于路由等问题，导致包无法正常的到达。所以要结合系统的一些性能观察工具来综合分析，比如 netstat 统计单位时间的发送包的数目，看是否很明显超过了所在网络带宽的限制；观察 cpu 的利用率，看系统态的 CPU 时间是否明显大于用户态的 CPU 时间；如果程序运行在 Solaris 10 平台上，可以用 dtrace 工具看系统调用的情况，如果观察到 read/write 的系统调用的次数或者运行时间遥遥领先；这些都指向由于网络带宽所限导致的网络瓶颈。另外一种出现 Wait on condition 的常见情况是该线程在 sleep，等待 sleep 的时间到了，将被唤醒。

Waiting for Monitor Entry and in Object.wait()



\_The thread is waiting to get the lock for an object (some other thread may be holding the lock). This happens if two or more threads try to execute synchronized code. Note that the lock is always for an object and not for individual methods.\_

在多线程的 JAVA 程序中，实现线程之间的同步，就要说说 Monitor。Monitor 是 Java 中用以实现线程之间的互斥与协作的主要手段，它可以看成是对象或者 Class 的锁。每一个对象都有，也仅有一个 monitor。每个 Monitor 在某个时刻，只能被一个线程拥有，该线程就是

“Active Thread”，而其它线程都是 “Waiting Thread”，分别在两个队列 “Entry Set” 和 “Wait Set” 里面等候。在 “Entry Set” 中等待的线程状态是 “Waiting for monitor entry”，而在 “Wait Set” 中等待的线程状态是 “in Object.wait()”。

先看 “Entry Set” 里面的线程。我们称被 synchronized 保护起来的代码段为临界区。当一个线程申请进入临界区时，它就进入了 “Entry Set” 队列。对应的 code 就像：

```
synchronized(obj) {  
    .....  
}
```

这时有两种可能性：

该 monitor 不被其它线程拥有，Entry Set 里面也没有其它等待线程。本线程即成为相应类或者对象的 Monitor 的 Owner，执行临界区的代码。

该 monitor 被其它线程拥有，本线程在 Entry Set 队列中等待。

在第一种情况下，线程将处于 “Runnable” 的状态，而第二种情况下，线程 DUMP 会显示处于 “waiting for monitor entry” 。

临界区的设置，是为了保证其内部的代码执行的原子性和完整性。但是因为临界区在任何时间只允许线程串行通过，这和我们多线程的程序的初衷是相反的。如果在多线程的程序中，大量使用 synchronized，或者不适当的使用了它，会造成大量线程在临界区的入口等待，造成系统的性能大幅下降。如果在线程 DUMP 中发现了这个情况，应该审查源码，改进程序。

再看 “Wait Set” 里面的线程。当线程获得了 Monitor，进入了临界区之后，如果发现线程继续运行的条件没有满足，它则调用对象（一般就是被 synchronized 的对象）的 wait() 方法，放弃 Monitor，进入 “Wait Set” 队列。只有当别的线程在该对象上调用了 notify() 或者 notifyAll()， “Wait Set” 队列中线程才得到机会去竞争，但是只有一个线程获得对象的 Monitor，恢复到运行态。在 “Wait Set” 中的线程， DUMP 中表现为：  
in Object.wait()。

一般，Cpu 很忙时，则关注 runnable 的线程，Cpu 很闲时，则关注 waiting for monitor entry 的线程。

### 3.2 JVM 线程介绍

在 Thread Dump 中，有一些 JVM 内部的后台线程，来执行譬如垃圾回收，或者低内存的检测等等任务，这些线程往往在 JVM 初始化的时候就存在，如下所示：

#### HotSpot VM Thread

被 HotSpot VM 管理的内部线程为了完成内部本地操作，一般来说不需要担心它们，除非 CPU 很高。

```
"VM Periodic Task Thread" prio=10tid=0xad909400 nid=0xaed waiting  
on condition
```

## HotSpot GC Thread

当使用 HotSpot parallel GC，HotSpot VM 默认创建一定数目的 GC thread。

```
"GC task thread#0 (ParallelGC)"prio=10 tid=0xf690b400 nid=0xade  
runnable
```

```
"GC task thread#1 (ParallelGC)"prio=10 tid=0xf690cc00 nid=0xadf  
runnable
```

```
"GC task thread#2 (ParallelGC)"prio=10 tid=0xf690e000 nid=0xae0  
runnable
```

```
.....
```

当面对过多 GC，内存泄露等问题时，这些是关键的数据。使用 native id，可以将从 OS/Java 进程观测到的高 CPU 与这些线程关联起来。

## JNI global references count

JNI global reference 是基本的对象引用，从本地代码到被 Java GC 管理的 Java 对象的引用。其角色是阻止仍然被本地代码使用的对象集合，但在 Java 代码中没有引用。在探测 JNI 相关内存泄露时，关注 JNI references 很重要。

如果你的程序直接使用 JNI 或使用第三方工具，如检测工具，检测本地内存泄露。

JNI global references: 832

## Java Heap utilization view

从 jdk1.6 开始在 thread dump 快照底部，可以找到崩溃点的内存空间利用情况:YongGen,OldGen 和 PermGen。目前我测试的系统导出的 thread dump，还未见到这一部分内容（sun jdk1.6）。以下例子，摘自他人文章：

Heap

PSYoungGen total 466944K, used 178734K [0xffffffff45c00000, 0xffffffff70800000, 0xffffffff70800000)

eden space 233472K, 76% used [0xffffffff45c00000,0xffffffff50ab7c50,0xffffffff54000000)

from space 233472K, 0% used [0xffffffff62400000,0xffffffff62400000,0xffffffff70800000)

to space 233472K, 0% used [0xffffffff54000000,0xffffffff54000000,0xffffffff62400000)

PSOldGen total 1400832K, used 1400831K [0xffffffffef0400000, 0xffffffff45c00000, 0xffffffff45c00000)

object space 1400832K, 99% used [0xffffffffef0400000,0xffffffff45bffb8,0xffffffff45c00000)

PSPermGen total 262144K, used 248475K [0xffffffffed0400000, 0xfffffffffee0400000, 0xffffffffef0400000)

object space 262144K, 94% used [0xffffffffed0400000,0xffffffffedf6a6f08, 0xfffffffffee0400000)

还有一些其他的线程（如下），不一一介绍了，有兴趣，可查看文章最后的附件信息。

"Low Memory Detector" daemon prio=10tid=0xad907400 nid=0xaecc runnable [0x00000000]

"CompilerThread1" daemon prio=10tid=0xad905400 nid=0xaebd waiting on condition [0x00000000]

"CompilerThread0" daemon prio=10tid=0xad903c00 nid=0xaeaa waiting on condition [0x00000000]

"Signal Dispatcher" daemon prio=10tid=0xad902400 nid=0xae9f runnable [0x00000000]

"Finalizer" daemon prio=10tid=0xf69eec00 nid=0xae8 in Object.wait() [0xaf17d000]

"Reference Handler" daemon prio=10tid=0xf69ed800 nid=0xae7 in Object.wait() [0xae1e7000]

"VM Thread" prio=10 tid=0xf69e9800nid=0xae6 runnable

四、案例分析：

## 4.1、使用方案

### cpu 飙高，load 高，响应很慢

方案：

- \* 一个请求过程中多次 dump
- \* 对比多次 dump 文件的 runnable 线程，如果执行的方法有较大变化，说明比较正常。如果在执行同一个方法，就有一些问题了。

### 查找占用 cpu 最多的线程信息

方案：

- \* 使用命令：top -H -p pid ( pid 为被测系统的进程号 )，找到导致 cpu 高的线程 id。

上述 Top 命令找到的线程 id，对应着 dump thread 信息中线程的 nid，只不过一个是十进制，一个是十六进制。

- \* 在 thread dump 中，根据 top 命令查找的线程 id，查找对应的线程堆栈信息。

### cpu 使用率不高但是响应很慢

方案：

- \* 进行 dump，查看是否有很多 thread stuck 在了 i/o、数据库等地方，定位瓶颈原因。

### 请求无法响应

方案：

- \* 多次 dump，对比是否所有的 runnable 线程都一直在执行相同的方法，如果是的，恭喜你，锁住了！

## 4.2 案例分析：

### 1.死锁：

死锁经常表现为程序的停顿，或者不再响应用户的请求。从操作系统上观察，对应进程的 CPU 占用率为零，很快会从 top 或 prstat 的输出中消失。

在 thread dump 中，会看到类似于这样的信息：

```
2012-09-21 14:58:36
Full thread dump Java HotSpot(TM) Client VM (1.6.0_02-b06 mixed mode, sharing):

"DestroyJavaVM" prio=6 tid=0x019f1400 nid=0x19d8 waiting on condition [0x00000000
0..0x002afd20]
  java.lang.Thread.State: RUNNABLE

"Thread-1" prio=6 tid=0x01a80000 nid=0x1b04 waiting for monitor entry [0x03e4f00
0..0x03e4fae8]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at T1.run(DeadLockTest.java:39)
    - waiting to lock <0x276ea028> (a java.lang.String)
    - locked <0x2c4c4f28> (a java.lang.String)
    at java.lang.Thread.run(Unknown Source)

"Thread-0" prio=6 tid=0x01a7f800 nid=0x1fbc waiting for monitor entry [0x03dff00
0..0x03dffb68]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at T1.run(DeadLockTest.java:39)
    - waiting to lock <0x2c4c4f28> (a java.lang.String)
    - locked <0x276ea028> (a java.lang.String)
    at java.lang.Thread.run(Unknown Source)

"Low Memory Detector" daemon prio=6 tid=0x01a6a000 nid=0x1024 runnable [0x000000
00..0x00000000]
  java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x01a5fc00 nid=0x16ac waiting on condition
[0x00000000..0x03d0f8f0]
  java.lang.Thread.State: RUNNABLE

"Attach Listener" daemon prio=10 tid=0x01a5ec00 nid=0x3b0 runnable [0x00000000..
```

(图 1)

```

Found one Java-level deadlock:
=====
"Thread-1":
  waiting to lock monitor 0x01a1bb84 (object 0x276ea028, a java.lang.String),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x01a1bb1c (object 0x2c4c4f28, a java.lang.String),
  which is held by "Thread-1"

Java stack information for the threads listed above:
=====
"Thread-1":
  at T1.run(DeadLockTest.java:39)
  - waiting to lock <0x276ea028> (a java.lang.String)
  - locked <0x2c4c4f28> (a java.lang.String)
  at java.lang.Thread.run(Unknown Source)
"Thread-0":
  at T1.run(DeadLockTest.java:39)
  - waiting to lock <0x2c4c4f28> (a java.lang.String)
  - locked <0x276ea028> (a java.lang.String)
  at java.lang.Thread.run(Unknown Source)

Found 1 deadlock.

Heap
def new generation      total 960K, used 263K [0x23690000, 0x23790000, 0x23b70000)
  eden space 896K,  29% used [0x23690000, 0x236d1f30, 0x23770000)
  from space 64K,   0% used [0x23770000, 0x23770000, 0x23780000)
  to   space 64K,   0% used [0x23780000, 0x23780000, 0x23790000)
tenured generation      total 4096K, used 0K [0x23b70000, 0x23f70000, 0x27690000)
  the space 4096K,   0% used [0x23b70000, 0x23b70000, 0x23b70200, 0x23f70000)
compacting perm gen     total 12288K, used 362K [0x27690000, 0x28290000, 0x2b690000)

```

(图2)

说明：

(图1)中有一个“Waiting formonitor entry”，可以看出，两个线程各持有一个锁，又在等待另一个锁，很明显这两个线程互相持有对方正在等待的锁。所以造成了死锁现象；

(图2)中对死锁的现象做了说明，可以看到，是“DeadLockTest.java”的39行造成的死锁现象。这样就能到相应的代码下去查看，定位问题。

## 2.热锁



热锁，也往往是导致系统性能瓶颈的主要因素。其表现特征为：由于多个线程对临界区，或者锁的竞争，可能出现：

\* 频繁的线程的上下文切换：从操作系统对线程的调度来看，当线程在等待资源而阻塞的时候，操作系统会将之切换出来，放到等待的队列，当线程获得资源之后，调度算法会将这个线程切换进去，放到执行队列中。

\* 大量的系统调用：因为线程的上下文切换，以及热锁的竞争，或者临界区的频繁的进出，都可能导致大量的系统调用。

\* 大部分 CPU 开销用在“系统态”：线程上下文切换，和系统调用，都会导致 CPU 在“系统态”运行，换言之，虽然系统很忙碌，但是 CPU 用在“用户态”的比例较小，应用程序得不到充分的 CPU 资源。

\* 随着 CPU 数目的增多，系统的性能反而下降。因为 CPU 数目多，同时运行的线程就越多，可能就会造成更频繁的线程上下文切换和系统态的 CPU 开销，从而导致更糟糕的性能。

上面的描述，都是一个 scalability（可扩展性）很差的系统的表现。从整体的性能指标看，由于线程热锁的存在，程序的响应时间会变长，吞吐量会降低。

那么，怎么去了解“热锁”出现在什么地方呢？一个重要的方法还是结合操作系统的各种工具观察系统资源使用状况，以及收集 Java 线程的 DUMP 信息，看线程都阻塞在什么方法上，了解原因，才能找到对应的解决方法。

我们曾经遇到过这样的例子，程序运行时，出现了以上指出的各种现象，通过观察操作系统的资源使用统计信息，以及线程 DUMP 信息，确定了程序中热锁的存在，并发现大多数的线程状态都是 Waiting for monitor entry 或者

Wait on monitor , 且是阻塞在压缩和解压缩的方法上。后来采用第三方的压缩包 javalib 替代 JDK 自带的压缩包后 , 系统的性能提高了几倍。

五、附件：

JVM 运行过程中产生的一些比较重要的线程罗列如下：

线程名称	所属	解释说明
Attach Listener	JVM	Attach Listener 线程是负责接收到外部的命令，而对该命令进行执行的并且吧结果返回给发送者。通常我们会用一些命令去要求 jvm 给我们一些反馈信息，如：java -version、jmap、jstack 等等。 如果该线程在 jvm 启动的时候没有初始化，那么，则会在用户第一次执行 jvm 命令时，得到启动。
Signal Dispatcher	JVM	前面我们提到第一个 Attach Listener 线程的职责是接收外部 jvm 命令，当命令接收成功后，会交给 signal dispatcher 线程去进行分发到各个不同的模块处理命令，并且返回处理结果。signal dispatcher 线程也是在第一次接收外部 jvm 命令时，进行初始化工作。

CompilerThread0	JVM	用来调用 JITing，实时编译装卸 class。通常，jvm 会启动多个线程来处理这部分工作，线程名称后面的数字也会累加，例如： CompilerThread1
Concurrent Mark-Sweep GC Thread	JVM	并发标记清除垃圾回收器（就是通常所说的 CMS GC）线程，该线程主要针对老年代垃圾回收。ps：启用该垃圾回收器，需要在 jvm 启动参数中加上：-XX:+UseConcMarkSweepGC
DestroyJavaVM	JVM	执行 main()的线程在 main 执行完后调用 JNI 中的 jni_DestroyJavaVM() 方法唤起 DestroyJavaVM 线程。 JVM 在 Jboss 服务器启动之后，就会唤起 DestroyJavaVM 线程，处于等待状态，等待其它线程（java 线程和 native 线程）退出时通知它卸载 JVM。线程退出时，都会判断自己当前是否是整个 JVM 中最后一个非 daemon 线程，如果是，则通知 DestroyJavaVM 线程卸载 JVM。 ps：

		<p>扩展一下：</p> <p>1.如果线程退出时判断自己不为最后一个非 daemon 线程，那么调用 thread-&gt;exit(false) ，并在其中抛出 thread_end 事件，jvm 不退出。</p> <p>2.如果线程退出时判断自己为最后一个非 daemon 线程，那么调用 before_exit() 方法，抛出两个事件：</p> <p>事件 1：thread_end 线程结束事件；</p> <p>事件 2：VM 的 death 事件。</p> <p>然后调用 thread-&gt;exit(true) 方法，接下来把线程从 active list 卸下，删除线程等等一系列工作执行完成后，则通知正在等待的 DestroyJavaVM 线程执行卸载 JVM 操作。</p>
ContainerBackgroundProcessor 线程	JBOSS	<p>它是一个守护线程, 在 jboss 服务器在启动的时候就初始化了,主要工作是定期去检查有没有 Session 过期.过期则清除.</p> <p>参考：</p> <p><a href="http://liudeh-009.iteye.com/blog/1584876">http://liudeh-009.iteye.com/blog/1584876</a></p>

Dispatcher-Thread-3 线程	Log4j	<p>Log4j 具有异步打印日志的功能，需要异步打印日志的 Appender 都需要注册到 AsyncAppender 对象里面去，由 AsyncAppender 进行监听，决定何时触发日志打印操作。</p> <p>AsyncAppender 如果监听到它管辖范围内的 Appender 有打印日志的操作，则给这个 Appender 生成一个相应的 event，并将该 event 保存在一个 buffuer 区域内。</p> <p>Dispatcher-Thread-3 线程负责判断这个 event 缓存区是否已经满了，如果已经满了，则将缓存区内的所有 event 分发到 Appender 容器里面去，那些注册上来的 Appender 收到自己的 event 后，则开始处理自己的日志打印工作。 Dispatcher-Thread-3 线程是一个守护线程。</p>
Finalizer 线程	JVM	<p>这个线程也是在 main 线程之后创建的，其优先级为 10，主要用于在垃圾收集前，调用对象的 finalize()方法；</p> <p>关于 Finalizer 线程的几点：</p>

	<p>1) 只有当开始一轮垃圾收集时，才会开始调用 finalize()方法；因此并不是所有对象的 finalize()方法都会被执行；</p> <p>2) 该线程也是 daemon 线程，因此如果虚拟机中没有其他非 daemon 线程，不管该线程有没有执行完 finalize()方法，JVM 也会退出；</p> <p>3) JVM 在垃圾收集时会将失去引用的对象包装成 Finalizer 对象（Reference 的实现），并放入 ReferenceQueue，由 Finalizer 线程来处理；最后将该 Finalizer 对象的引用置为 null，由垃圾收集器来回收；</p> <p>4) JVM 为什么要单独用一个线程来执行 finalize()方法呢？如果 JVM 的垃圾收集线程自己来做，很有可能由于在 finalize()方法中误操作导致 GC 线程停止或不可控，这对 GC 线程来说是一种灾难；</p>
--	--

Gang worker#0	JVM	JVM 用于做新生代垃圾回收 (monir gc) 的一个线程。#号后面是线程编号，例如：Gang worker#1
GC Daemon	JVM	<p>GC Daemon 线程是 JVM 为 RMI 提供远程分布式 GC 使用的，GC Daemon 线程里面会主动调用 System.gc()方法，对服务器进行 Full GC。其初衷是当 RMI 服务器返回一个对象到其客户机（远程方法的调用方）时，其跟踪远程对象在客户机中的使用。当再没有更多的对客户机上远程对象的引用时，或者如果引用的“租借”过期并且没有更新，服务器将垃圾回收远程对象。</p> <p>不过，我们现在 jvm 启动参数都加上了-XX:+DisableExplicitGC 配置，所以，这个线程只有打酱油的份了。</p>
IdleRemover	JBOSS	Jboss 连接池有一个最小值，该线程每过一段时间都会被 Jboss 唤起，用于检查和销毁连接池中空闲和无效的连接，直到剩余的连接数小于等于它的最小值。

Java2D Disposer	JVM	<p>这个线程主要服务于 awt 的各个组件。说起该线程的主要工作职责前，需要先介绍一下 Disposer 类是干嘛的。Disposer 提供一个 addRecord 方法。如果你想在对象被销毁前再做一些善后工作，那么，你可以调用 Disposer#addRecord 方法，将这个对象和一个自定义的 DisposerRecord 接口实现类，一起传入进去，进行注册。</p> <p>Disposer 类会唤起 “Java2D Disposer” 线程，该线程会扫描已注册的这些对象是否要被回收了，如果是，则调用该对象对应的 DisposerRecord 实现类里面的 dispose 方法。</p> <p>Disposer 实际上不限于在 awt 应用场景，只是 awt 里面的很多组件需要访问很多操作系统资源，所以，这些组件在被回收时，需要先释放这些资源。</p>
InsttoolCacheScheduler_QuartzSchedulerThread	Quartz	<p>InsttoolCacheScheduler_QuartzSchedulerThread 是 Quartz 的主线程，</p>



	<p>它主要负责实时的获取下一个时间点要触发的触发器，然后执行触发器相关联的作业。</p> <p>原理大致如下：</p> <p>Spring 和 Quartz 结合使用的场景下，Spring IOC 容器初始化时会创建并初始化 Quartz 线程池（TreadPool），并启动它。刚启动时线程池中每个线程都处于等待状态，等待外界给他分配 Runnable（持有作业对象的线程）。</p> <p>继而接着初始化并启动 Quartz 的主线程（InsttoolCacheScheduler_QuartzSchedulerThread），该线程自启动后就会处于等待状态。等待外界给出工作信号之后，该主线程的 run 方法才实质上开始工作。run 中会获取 JobStore 中下一次要触发的作业，拿到之后会一直等待到该作业的真正触发时间，然后将该作业包装成一个 JobRunShell 对象（该对象实现了 Runnable 接口，其</p>
--	--

		<p>实看是上面 TreadPool 中等待外界分配给他的 Runnable ) , 然后将刚创建的 JobRunShell 交给线程池, 由线程池负责执行作业。</p> <p>线程池收到 Runnable 后, 从线程池一个线程启动 Runnable , 反射调用 JobRunShell 中的 run 方法, run 方法执行完成之后, TreadPool 将该线程回收至空闲线程中。</p>
InsttoolCacheScheduler_Worker-2	Quartz	<p>InsttoolCacheScheduler_Worker-2 线程就是 ThreadPool 线程的一个简单实现, 它主要负责分配线程资源去执行 InsttoolCacheScheduler_QuartzSchedulerThread 线程交给它的调度任务 ( 也就是 JobRunShell ) 。</p>
JBossLifeThread	JBoss	<p>Jboss 主线程启动成功, 应用程序部署完毕之后将 JBossLifeThread 线程实例化并且 start , JBossLifeThread 线程启动成功之后就处于等待状态, 以保持 Jboss Java 进程处于存活中。 所得比较通俗一点, 就是 Jboss 启动流程</p>

		<p>执行完毕之后，为什么没有结束？就是因为有这个线程 hold 主了它。</p>
JBoss System Threads(1)-1	Jboss	<p>该线程是一个 socket 服务，默认端口号为： 1099。</p> <p>主要用于接收外部 naming service ( Jboss JNDI ) 请求。</p>
JCA PoolFiller	Jboss	<p>该线程主要为 JBoss 内部提供连接池的托管。</p> <p>简单介绍一下工作原理：</p> <p>Jboss 内部凡是有远程连接需求的类，都需要实现</p> <p>ManagedConnectionFactory 接口，例如需要做 JDBC 连接的</p> <p>XAManagedConnectionFactory 对象，就实现了该接口。</p> <p>然后将</p> <p>XAManagedConnectionFactory 对象，</p> <p>还有其它信息一起包装到</p> <p>InternalManagedConnectionPool 对象里面，接着将</p> <p>InternalManagedConnectionPool</p>

		<p>交给 PoolFiller 对象里面的列队进行管理。</p> <p>JCA PoolFiller 线程会定期判断列队内是否有需要创建和管理的</p> <p>InternalManagedConnectionPool 对象，如果有的话，则调用该对象的 fillToMin 方法， 触发它去创建相应的远程连接，并且将这个连接维护到它相应的连接池里面去。</p>
JDWP Event Helper Thread	JVM	<p>JDWP 是通讯交互协议，它定义了调试器和被调试程序之间传递信息的格式。</p> <p>它详细完整地定义了请求命令、回应数据和错误代码，保证了前端和后端的 JVMTI 和 JDI 的通信通畅。 该线程主要负责将 JDI 事件映射成 JVMTI 信号，以达到调试过程中操作 JVM 的目的。</p>
JDWP Transport Listener: dt_socket	JVM	<p>该线程是一个 Java Debugger 的监听器线程，负责受理客户端的 debug 请求。 通常我们习惯将它的监听端口设置为 8787。</p>

Low Memory Detector	JVM	这个线程是负责对可使用内存进行检测，如果发现可用内存低，分配新的内存空间。
process reaper	JVM	该线程负责去执行一个 OS 命令行的操作。
Reference Handler	JVM	JVM 在创建 main 线程后就创建 Reference Handler 线程，其优先级最高，为 10，它主要用于处理引用对象本身（软引用、弱引用、虚引用）的垃圾回收问题。
Surrogate Locker Thread (CMS)	JVM	<p>这个线程主要用于配合 CMS 垃圾回收器使用，它是一个守护线程，其主要负责处理 GC 过程中，Java 层的 Reference（指软引用、弱引用等等）与 jvm 内部层面的对象状态同步。这里对它们的实现稍微做一下介绍：这里拿 WeakHashMap 做例子，将一些关键点先列出来（我们后面会将这些关键点全部串起来）：</p> <p>1.我们知道 HashMap 用 Entry[]数组来存储数据的，WeakHashMap 也不例外，内部有一个 Entry[]数组。</p>

	<p>2. WeakHashMap 的 Entry 比较特殊，它的继承体系结构为 Entry-&gt;WeakReference-&gt;Reference。</p> <p>3.Reference 里面有一个全局锁对象：Lock，它也被称为 pending_lock.注意：它是静态对象。</p> <p>4. Reference 里面有一个静态变量：pending。</p> <p>5. Reference 里面有一个静态内部类：ReferenceHandler 的线程，它在 static 块里面被初始化并且启动，启动完成后处于 wait 状态，它在一个 Lock 同步锁模块中等待。</p> <p>6.另外，WeakHashMap 里面还实例化了一个 ReferenceQueue 列队，这个列队的作用，后面会提到。</p> <p>7.上面关键点就介绍完毕了，下面我们把他们串起来。</p> <p>假设，WeakHashMap 对象里面已经保存了很多对象的引用。</p>
--	---

	<p>JVM 在进行 CMS GC 的时候，会创建一个 ConcurrentMarkSweepThread ( 简称 CMST ) 线程去进行 GC ,</p> <p>ConcurrentMarkSweepThread 线程被创建的同时会创建一个 SurrogateLockerThread ( 简称 SLT ) 线程并且启动它，SLT 启动之后，处于等待阶段。CMST 开始 GC 时，会发一个消息给 SLT 让它去获取 Java 层 Reference 对象的全局锁：Lock。直到 CMS GC 完毕之后，JVM 会将 WeakHashMap 中所有被回收的对象所属的 WeakReference 容器对象放入到 Reference 的 pending 属性当中（每次 GC 完毕之后，pending 属性基本上都不会为 null 了），然后通知 SLT 释放并且 notify 全局锁:Lock。此时激活了 ReferenceHandler 线程的 run 方法，使其脱离 wait 状态，开始工作了。</p> <p>ReferenceHandler 这个线程会将 pending 中的所有 WeakReference</p>
--	---

		<p>对象都移动到它们各自的列队当中，比如当前这个 WeakReference 属于某个 WeakHashMap 对象，那么它就会被放入相应的 ReferenceQueue 列队里面（该列队是链表结构）。当我们下次从 WeakHashMap 对象里面 get、put 数据或者调用 size 方法的时候，WeakHashMap 就会将 ReferenceQueue 列队中的 WeakReference 依依 poll 出来去和 Entry[]数据做比较，如果发现相同的，则说明这个 Entry 所保存的对象已经被 GC 掉了，那么将 Entry[]内的 Entry 对象剔除掉。</p>
taskObjectTimerFactory	JVM	<p>顾名思义，该线程就是用来执行任务的。当我们把一个任务交给 Timer 对象，并且告诉它执行时间，周期时间后，Timer 就会将该任务放入任务列队，并且通知 taskObjectTimerFactory 线程去处理任务，taskObjectTimerFactory 线程会将状态为取消的任务从任务列队中移</p>



		除，如果任务是非重复执行类型的，则在执行完该任务后，将它从任务列队中移除，如果该任务是需要重复执行的，则计算出它下一次执行的时间点。
VM Periodic Task Thread	JVM	该线程是 JVM 周期性任务调度的线程，它由 WatcherThread 创建，是一个单例对象。该线程在 JVM 内使用得比较频繁，比如：定期的内存监控、JVM 运行状况监控，还有我们经常需要去执行一些 jstat 这类命令查看 gc 的情况，如下：  jstat -gcutil 23483 250 7 这个命令告诉 jvm 在控制台打印 PID 为：23483 的 gc 情况，间隔 250 毫秒打印一次，一共打印 7 次。
VM Thread	JVM	这个线程就比较牛 b 了，是 jvm 里面的线程母体，根据 hotspot 源码（vmThread.hpp）里面的注释，它是一个单例的对象（最原始的线程）会产生或触发所有其他的线程，这个单个的 VM 线程是会被其他线程所使用来做一些 VM 操作（如，清扫垃圾等）。

	<p>在 VMThread 的结构体里有一个 VMOperationQueue 列队，所有的 VM 线程操作(vm_operation)都会被保存到这个列队当中，VMThread 本身就是一个线程，它的线程负责执行一个自轮询的 loop 函数(具体可以参考：VMThread.cpp 里面的 void VMThread::loop())，该 loop 函数从 VMOperationQueue 列队中按照优先级取出当前需要执行的操作对象(VM_Operation)，并且调用 VM_Operation-&gt;evaluate 函数去执行该操作类型本身的业务逻辑。</p> <p>ps：VM 操作类型被定义在 vm_operations.hpp 文件内，列举几个：ThreadStop、ThreadDump、PrintThreads、GenCollectFull、GenCollectFullConcurrent、</p>
--	---

		CMS_Initial_Mark、 CMS_Final_Remark.....
--	--	--

参考文章：

<http://jameswxx.iteye.com/blog/1041173>

<http://blog.csdn.net/wanyanxgf/article/details/6944987>

<http://blog.csdn.net/fanshadoop/article/details/8509218>

<http://www.7dtest.com/site/article-80-1.html>

<http://blog.csdn.net/a43350860/article/details/8134234>

[https://weblogs.java.net/blog/mandychung/archive/2005/11/thread\\_dump\\_and\\_1.html](https://weblogs.java.net/blog/mandychung/archive/2005/11/thread_dump_and_1.html)

## 性能分析工具之-- Eclipse Memory Analyzer tool(MAT) (一)

### 前言

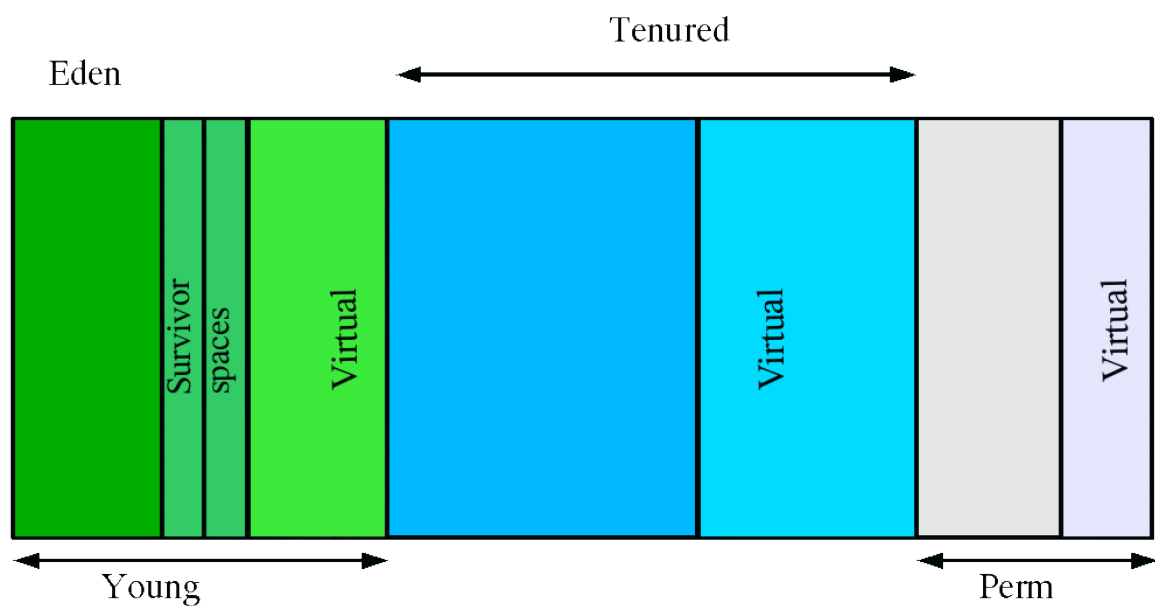
在平时工作过程中，有时会遇到 OutOfMemoryError，我们知道遇到 Error 一般表明程序存在着严重问题，可能是灾难性的。所以找出是什么原因造成 OutOfMemoryError 非常重要。现在向大家引荐 Eclipse Memory Analyzer tool(MAT)，来化解我们遇到的难题。如未说明，本文均使用 Java 5.0 on Windows XP SP3 环境。

### 为什么用 MAT

之前的观点，我认为使用实时 profiling/monitoring 之类的工具，用一种非常实时的方式来分析哪里存在内存泄漏是很正确的。年初使用了某 profiler 工具测试消息中间件中存在的内存泄漏，发现在吞吐量很高的时候 profiler 工具自己也无法响应，这让人很头痛。后来了解到这样的工具本身就要消耗性能，且在某些条件下还发现不了泄漏。所以，分析离线数据就非常重要了，MAT 正是这样一款工具。

## 为何会内存溢出

我们知道 JVM 根据 generation(代)来进行 GC，根据下图所示，一共被分为 young generation(年轻代)、tenured generation(老年代)、permanent generation(永久代, perm gen)，perm gen (或称 Non-Heap 非堆) 是个异类，稍后会讲到。注意，heap 空间不包括 perm gen。



绝大多数的对象都在 young generation 被分配，也在 young generation 被收回，当 young generation 的空间被填满，GC 会进行 minor collection(次回收)，这次回收不涉及到 heap 中的其他 generation，minor collection 根据 weak generational hypothesis(弱年代假设)来假设 young generation 中大量的对象都是垃圾需要回收，minor collection 的过程会非常快。young generation 中未被回收的对象被转移到 tenured generation，然而 tenured generation 也会被填满，最终触发 major collection(主回收)，这次回收针对整个 heap，由于涉及到大量对象，所以比 minor collection 慢得多。

JVM 有三种垃圾回收器，分别是 throughput collector，用来做并行 young generation 回收，由参数-XX:+UseParallelGC 启动；concurrent low pause collector，用来做 tenured generation 并发回收，由参数-XX:+UseConcMarkSweepGC 启动；incremental low pause collector，可以认为是默认的垃圾回收器。不建议直接使用某种垃圾回收器，最好让 JVM 自己决断，除非自己有足够的把握。

Heap 中各 generation 空间是如何划分的？通过 JVM 的-Xmx=n 参数可指定最大 heap 空间，而-Xms=n 则是指定最小 heap 空间。在 JVM 初始化的时候，如果最小 heap 空间小于最大 heap 空间的话，如上图所示 JVM 会把未用到的空间标注为 Virtual。除了这两个参数还有-XX:MinHeapFreeRatio=n 和 -XX:MaxHeapFreeRatio=n 来分别控制最大、最小的剩余空间与活动对象之比

例。在 32 位 Solaris SPARC 操作系统下，默认值如下，在 32 位 windows xp 下，默认值也差不多。

参数	默认值
MinHeapFreeRatio	40
MaxHeapFreeRatio	70
-Xms	3670k
-Xmx	64m

由于 tenured generation 的 major collection 较慢，所以 tenured generation 空间小于 young generation 的话，会造成频繁的 major collection，影响效率。Server JVM 默认的 young generation 和 tenured generation 空间比例为 1:2，也就是说 young generation 的 eden 和 survivor 空间之和是整个 heap（当然不包括 perm gen）的三分之一，该比例可以通过-XX:NewRatio=n 参数来控制，而 Client JVM 默认的-XX:NewRatio 是 8。至于调整 young generation 空间大小的 NewSize=n 和 MaxNewSize=n 参数就不讲了，请参考后面的资料。

young generation 中幸存的对象被转移到 tenured generation，但不幸的是 concurrent collector 线程在这里进行 major collection，而在回收任务结束前空间被耗尽了，这时将会发生 Full Collections(Full GC)，整个应用程序都会停止下来直到回收完成。Full GC 是高负载生产环境的噩梦.....

现在来说说异类 perm gen，它是 JVM 用来存储无法在 Java 语言级描述的对象，这些对象分别是类和方法数据（与 class loader 有关）以及 interned strings(字符串驻留)。一般 32 位 OS 下 perm gen 默认 64m，可通过参数-XX:MaxPermSize=n 指定，[JVM Memory Structure](#) 一文说，对于这块区域，没有更详细的文献了，神秘。

回到问题“为何会内存溢出？”。

要回答这个问题又要引出另外一个话题，既什么样的对象 GC 才会回收？当然是 GC 发现通过任何 reference chain(引用链)无法访问某个对象的时候，该对象即被回收。名词 GC Roots 正是分析这一过程的起点，例如 JVM 自己确保了对象的可达性(那么 JVM 就是 GC Roots)，所以 GC Roots 就是这样在内存中保持对象可达性的，一旦不可到达，即被回收。通常 GC Roots 是一个在 current thread(当前线程)的 call stack(调用栈)上的对象（例如方法参数和局部变量），或者是线程自身或者是 system class loader(系统类加载器)加载的类以及 native code(本地代码)保留的活动对象。所以 GC Roots 是分析对象为何还存活于内存中的利器。知道了什么样的对象 GC 才会回收后，再来学习下对象引用都包含哪些吧。

从最强到最弱，不同的引用（可达性）级别反映了对象的生命周期。

I Strong Ref(强引用)：通常我们编写的代码都是 Strong Ref，于此对应的是强可达性，只有去掉强可达，对象才被回收。

I Soft Ref(软引用)：对应软可达性，只要有足够的内存，就一直保持对象，直到发现内存吃紧且没有 Strong Ref 时才回收对象。一般可用来实现缓存，通过 `java.lang.ref.SoftReference` 类实现。

I Weak Ref(弱引用)：比 Soft Ref 更弱，当发现不存在 Strong Ref 时，立刻回收对象而不必等到内存吃紧的时候。通过 `java.lang.ref.WeakReference` 和 `java.util.WeakHashMap` 类实现。

I Phantom Ref(虚引用)：根本不会在内存中保持任何对象，你只能使用 Phantom Ref 本身。一般用于在进入 `finalize()` 方法后进行特殊的清理过程，通过 `java.lang.ref.PhantomReference` 实现。

有了上面的种种我相信很容易就能把 heap 和 perm gen 撑破了吧，是的利用 Strong Ref，存储大量数据，直到 heap 撑破；利用 interned strings ( 或者 class loader 加载大量的类 ) 把 perm gen 撑破。

## 关于 shallow size、retained size

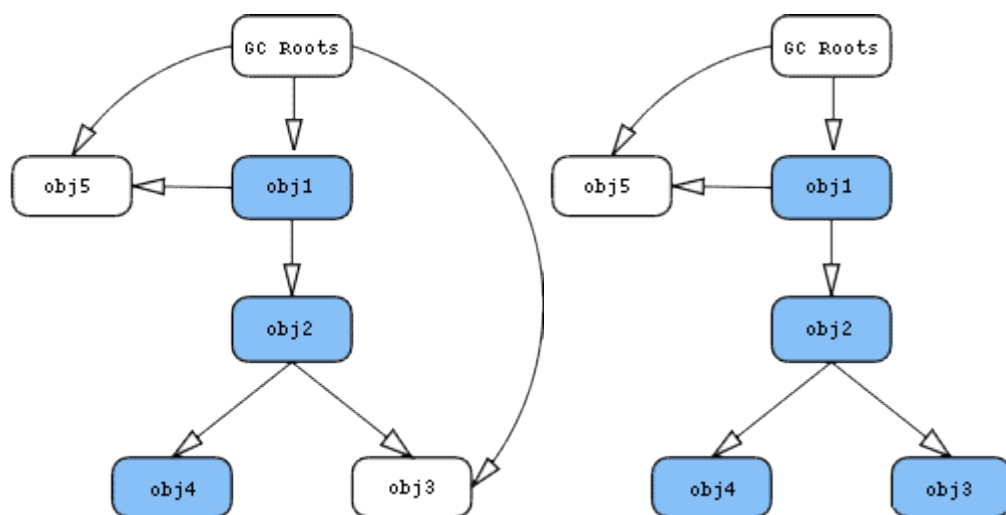
Shallow size 就是对象本身占用内存的大小，不包含对其他对象的引用，也就是对象头加成员变量（不是成员变量的值）的总和。在 32 位系统上，对象头占用 8 字节，int 占用 4 字节，不管成员变量（对象或数组）是否引用了其他对象（实例）或者赋值为 null 它始终占用 4 字节。故此，对于 String 对象实例来说，它有三个 int 成员（ $3 \times 4 = 12$  字节）、一个 char[] 成员（ $1 \times 4 = 4$  字节）以及一个对象头（8 字节），总共  $3 \times 4 + 1 \times 4 + 8 = 24$  字节。根据这一原



则，对 `String a=" rosen jiang"` 来说，实例 `a` 的 shallow size 也是 24 字节（很多人对此有争议，请看官甄别并留言给我）。

Retained size 是该对象自己的 shallow size，加上从该对象能直接或间接访问到对象的 shallow size 之和。换句话说，retained size 是该对象被 GC 之后所能回收到内存的总和。为了更好的理解 retained size，不妨看个例子。

把内存中的对象看成下图中的节点，并且对象和对象之间互相引用。这里有一个特殊的节点 GC Roots，正解！这就是 reference chain 的起点。



从 `obj1` 入手，上图中蓝色节点代表仅仅只有通过 `obj1` 才能直接或间接访问的对象。因为可以通过 GC Roots 访问，所以左图的 `obj3` 不是蓝色节点；而在右图却是蓝色，因为它已经被包含在 retained 集合内。

所以对于左图，`obj1` 的 retained size 是 `obj1`、`obj2`、`obj4` 的 shallow size 总和；右图的 retained size 是 `obj1`、`obj2`、`obj3`、`obj4` 的 shallow size 总和。`obj2` 的 retained size 可以通过相同的方式计算。

## Heap Dump

heap dump 是特定时间点，java 进程的内存快照。有不同的格式来存储这些数据，总的来说包含了快照被触发时 java 对象和类在 heap 中的情况。由于快照只是一瞬间的事情，所以 heap dump 中无法包含一个对象在何时、何地（哪个方法中）被分配这样的信息。

在不同平台和不同 java 版本有不同的方式获取 heap dump，而 MAT 需要的是 HPROF 格式的 heap dump 二进制文件。

如何获取 heap dump 文件？

通常来说，只要你设置了如下所示的 JVM 参数：

`-XX:+HeapDumpOnOutOfMemoryError`

JVM 就会在发生内存泄露时抓拍下当时的内存状态，也就是我们想要的堆转储文件。

如果你不想等到发生崩溃性的错误时才获得堆转储文件，也可以通过设置如下 JVM 参数来按需获取堆转储文件。

`-XX:+HeapDumpOnCtrlBreak`

除此之外，还有很多工具，例如 [JMap](#)，JConsole 都可以帮助我们得到一个堆转储文件。使用 jmap 获取 heap dump 的命令如下：

`jmap -dump:format=b,file=<dumpfile> <pid>`

解释：format=b-->指定格式为二进制；file=<dumpfile>-->指定文件名称，自定义；<pid> -->进程 id

由于我是 windows+JDK5，所以选择了 `-XX:-HeapDumpOnOutOfMemoryError` 这种方式，更多配置请参考 [MAT Wiki](#)。

## 参考资料

[MAT Wiki](#)

[Interned Strings](#)

[Strong,Soft,Weak,Phantom Reference](#)

[Tuning Garbage Collection with the 5.0 Java\[tm\] Virtual Machine](#)

[Permanent Generation](#)

[Understanding Weak References 译文](#)

[Java HotSpot VM Options](#)

[Shallow and retained sizes](#)

[JVM Memory Structure](#)

[GC roots](#)

## 性能分析工具之-- Eclipse Memory Analyzer tool(MAT) (二)

本文结合网络上比较优秀的文章，及自己的实践，做了一些修改和补充

### 前言

[性能分析工具之-- Eclipse Memory Analyzer tool\(MAT\) \(一\)](#) 中介绍了内存泄漏的前因后果。在本文中，将介绍 MAT 如何根据 heapdump 分析泄漏根源。由于测试范例可能过于简单，很容易找出问题，但我期待借此举一反三。

一开始不得不说说 ClassLoader，本质上，它的工作就是把磁盘上的类文件读入内存，然后调用 `java.lang.ClassLoader.defineClass` 方法告诉系统把内存镜像处理成合法的字节码。Java 提供了抽象类 `ClassLoader`，所有用户自定义类装载器都实例化自 `ClassLoader` 的子类。`systemclass loader` 在没有指定装载器的情况下默认装载用户类，在 Sun Java 1.5 中既 `sun.misc.Launcher$AppClassLoader`。更详细的内容请参看下面的资料。

### 准备 heap dump

请看下面的 Pilot 类，没啥特殊的。

```
/**
 * Pilot class
 * @author rosen jiang
 */
package org.rosenjiang.bo;

public class Pilot{

    String name;

    int age;

    public Pilot(String a, int b){

        name = a;

        age = b;

    }

}
```

然后再看 OOMHeapTest 类，它是如何撑破 heapdump 的。

```
/**
 * OOMHeapTest class
 * @author rosen jiang
 */
```

```
package org.rosenjiang.test;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import org.rosenjiang.bo.Pilot;

public class OOMHeapTest {

    public static void main(String[] args){

        oom();

    }

    private static void oom(){

        Map<String, Pilot> map = new HashMap<String, Pilot>();

        Object[] array = new Object[1000000];

        for(int i=0; i<1000000; i++){

            String d = new Date().toString();

            Pilot p = new Pilot(d, i);

            map.put(i+"rosen jiang", p);

            array[i]=p;

        }

    }

}
```

```
}  
  
}
```

是的，上面构造了很多的 Pilot 类实例，向数组和 map 中放。由于是 StrongRef，GC 自然不会回收这些对象，一直放在 heap 中直到溢出。当然在运行前，先要在 Eclipse 中配置 VM 参数-  
XX:+HeapDumpOnOutOfMemoryError。好了，一会儿功夫内存溢出，控制台打出如下信息。

```
java.lang.OutOfMemoryError: Java heap space  
Dumping heap to java_pid3600.hprof  
Heap dump file created [78233961 bytes in 1.995 secs]  
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  
e
```

java\_pid3600.hprof 既是 heap dump，可以在 OOMHeapTest 类所在的工程根目录下找到。

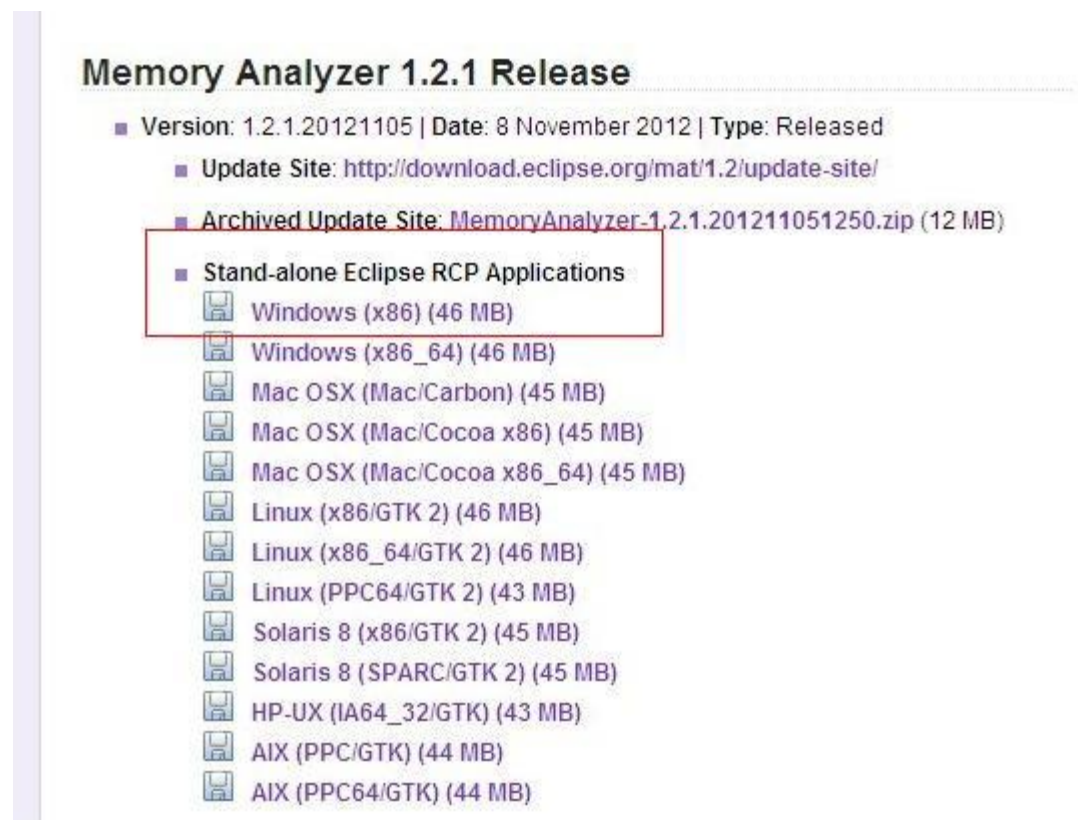
## MAT 安装

话分两头说，有了 heap dump 还得安装 MAT。

MAT 支持两种安装方式，一种是“独立版本”，用户不必安装 EclipseIDE 环境，MAT 作为一个独立的 EclipseRCP 应用运行；另一种是“插件版本”，也就是说 MAT 可以作为 EclipseIDE 的一个插件，和 Eclipse 开发平台集成。

独立版本，下载地址：<http://www.eclipse.org/mat/downloads.php>

下载的 zip 包，解压即可使用。下载页图示如下：



与 eclipse IDE 集成安装过程，可参看以下文章：

<http://www.ibm.com/developerworks/cn/opensource/os-cn-ecl-ma/index.html>

安装完成后，为了更有效率的使用 MAT，我们可以配置一些环境参数。因为通常而言，分析一个堆转储文件需要消耗很多的堆空间，为了保证分析的效率和

性能，在有条件的情况下，我们会建议分配给 MAT 尽可能多的内存资源。你可以采用如下两种方式来分配内存更多的内存资源给 MAT。

一种是修改启动参数 `MemoryAnalyzer.exe-vmargs -Xmx4g`

另一种是编辑文件 `MemoryAnalyzer.ini`，在里面添加类似信息 `-vmargs-Xmx4g`。

说明：

1. `MemoryAnalyzer.ini` 中的参数一般默认为 `-vmargs- Xmx1024m`，这就够用了。假如你机器的内存不大，改大该参数的值，会导致 `MemoryAnalyzer` 启动时，报错：`Failed to create the Java Virtual Machine`。

2. 当你导出的 dump 文件的大小大于你配置的 1024m（说明 1 中，提到的配置：`-vmargs- Xmx1024m`），MAT 输出分析报告的时候，会报错：`An internal error occurred during: "Parsing heap dump from XXX"`。适当调大说明 1 中的参数即可。

至此，MAT 就已经成功地安装配置好了，在 Eclipse 的左上角有 `Open Heap Dump` 按钮，按照刚才说的路径找到 `java_pid3600.hprof` 文件并打开。

先检查一下 MAT 生成的一系列文件：（截图来自另一个例子）





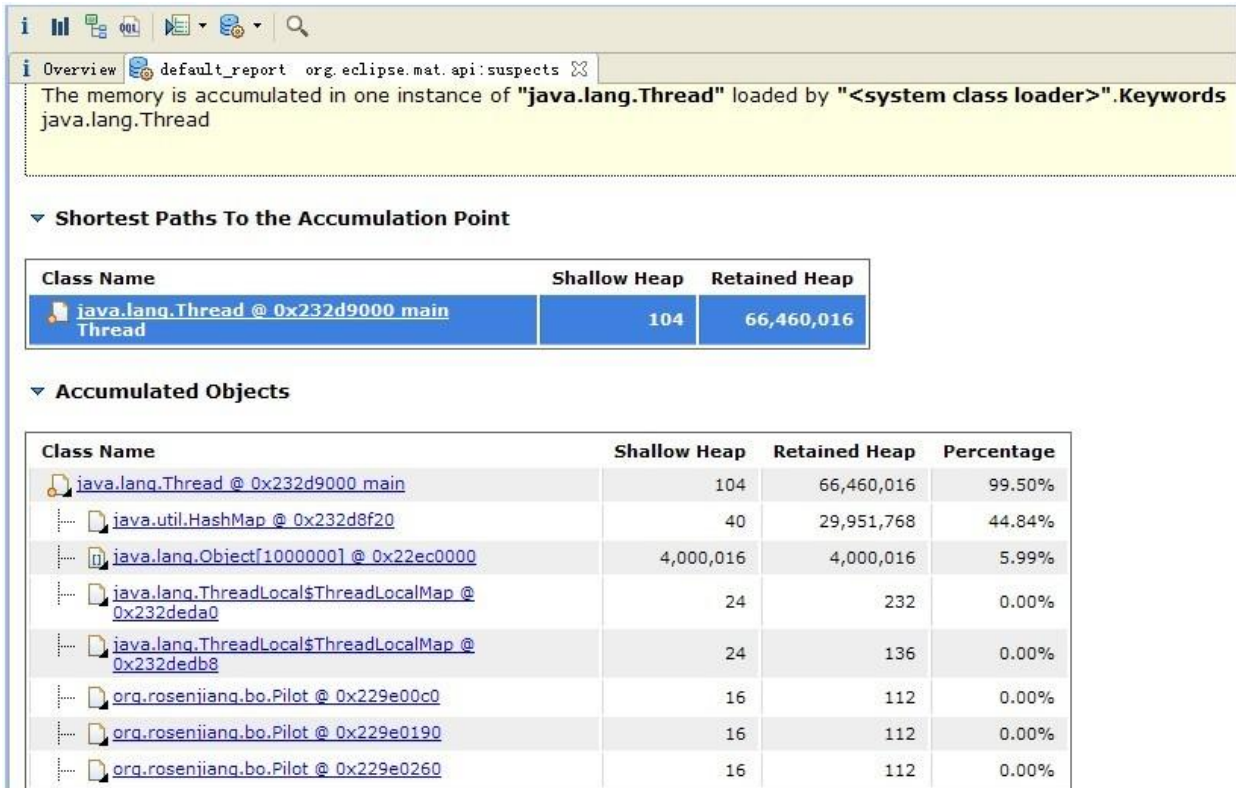
可以看到 MAT 工具提供了一个很贴心的功能，将报告的内容压缩打包到一个 zip 文件，并把它存放到原始堆转储文件的存放目录下，这样如果您需要和同事一起分析这个内存问题的话，只需要把这个小小的 zip 包发给他就可以了，不需要把整个堆文件发给他。并且整个报告是一个 HTML 格式的文件，用浏览器就可以轻松打开。

使用 MAT 打开 dump 文件，等待一会后，会弹出向导界面，保持默认设置，直接点 Finish 即是分析内存泄露问题。在点击 Finish 后，会出现 overview 界面，您可以点击工具栏上的 Leak Suspects 菜单项来生成内存泄露分析报告，也可以直接点击饼图下方的 Reports->Leak Suspects 链接来生成报告。如图：



MAT 工具分析了 heap dump 后在界面上非常直观的展示了一个饼图，该图深色区域被怀疑有内存泄漏，可以发现整个 heap 才 64M 内存，深色区域就占了 99.5%。接下来是一个简短的描述，告诉我们 main 线程占用了大量内存，并且明确指出 system class loader 加载的"java.lang.Thread"实例有内存聚集，并建议用关键字"java.lang.Thread"进行检查。所以，MAT 通过简单的两句话就说明了问题所在，就算使用者没什么处理内存问题的经验。在下面还有一个"Details"链接，在点开之前不妨考虑一个问题：为何对象实例会聚集在内存

中，为何存活（而未被 GC）？是的——Strong Ref，那么再走近一些吧。如图：



The screenshot shows the Eclipse MAT tool interface. The top bar indicates the current report is 'default\_report' for 'org.eclipse.mat.api:suspects'. The main text area states: 'The memory is accumulated in one instance of "java.lang.Thread" loaded by "<system class loader>". Keywords: java.lang.Thread'. Below this, there are two expandable sections. The first, 'Shortest Paths To the Accumulation Point', contains a table with three columns: 'Class Name', 'Shallow Heap', and 'Retained Heap'. The second, 'Accumulated Objects', contains a table with four columns: 'Class Name', 'Shallow Heap', 'Retained Heap', and 'Percentage'.

Class Name	Shallow Heap	Retained Heap
java.lang.Thread @ 0x232d9000 main Thread	104	66,460,016












Class Name	Shallow Heap	Retained Heap	Percentage
java.lang.Thread @ 0x232d9000 main	104	66,460,016	99.50%
java.util.HashMap @ 0x232d8f20	40	29,951,768	44.84%
java.lang.Object[1000000] @ 0x22ec0000	4,000,016	4,000,016	5.99%
java.lang.ThreadLocal\$ThreadLocalMap @ 0x232deda0	24	232	0.00%
java.lang.ThreadLocal\$ThreadLocalMap @ 0x232dedb8	24	136	0.00%
org.rosenjiang.bo.Pilot @ 0x229e00c0	16	112	0.00%
org.rosenjiang.bo.Pilot @ 0x229e0190	16	112	0.00%
org.rosenjiang.bo.Pilot @ 0x229e0260	16	112	0.00%

点击了"Details"链接之后，除了在上一页看到的描述外，还有 Shortest Paths To the Accumulation Point 和 Accumulated Objects 部分，这里说明了从 GC root 到聚集点的最短路径，以及完整的 reference chain。观察 Accumulated Objects 部分，java.util.HashMap 和 java.lang.Object[1000000]实例的 retained heap(size)最大，在上一篇文章中我们知道 retained heap 代表从该类实例沿着 reference chain 往下所能收集到的其他类实例的 shallow heap(size)总和，所以明显类实例都聚集在 HashMap 和 Object 数组中了。这里我们发现一个有趣的现象，既 Object 数组的 shallow heap 和 retained heap 竟然一样，通过 Shallow and retained

[sizes](#)一文可知，数组的 shallow heap 和一般对象（非数组）不同，依赖于数组的长度和里面的元素的类型，对数组求 shallow heap，也就是求数组集合内所有对象的 shallow heap 之和。好，再来看 org.rosenjiang.bo.Pilot 对象实例的 shallow heap 为何是 16，因为对象头是 8 字节，成员变量 int 是 4 字节、String 引用是 4 字节，故总共 16 字节。

在 Accumulated Objects 视图中，retained heap 占用最多的是 hashMap 和 object 数组，为啥它们会占用这么大的 heap 呢？这个时候需要分析 hashMap 和 object 数组中存放了一些什么对象？接着往下看，来到了 Accumulated Objects by Class 区域，顾名思义，这里能找到被聚集的对象实例的类名。org.rosenjiang.bo.Pilot 类上头条了，被实例化了 290,325 次，再返回去看程序，我承认是故意这么干的。还有很多有用的报告可用来协助分析问题，只是本文中的例子太简单，也用不上。

#### ▼ Accumulated Objects by Class

Label	Number Of Objects	Used Heap Size	Retained Heap Size
 <a href="#">org.rosenjiang.bo.Pilot</a>	290,235	4,643,760	32,506,320
 <a href="#">java.util.HashMap</a>	1	40	29,951,768
 <a href="#">java.lang.Object[]</a>	1	4,000,016	4,000,016
 <a href="#">java.lang.String[]</a>	38	1,200	1,200
 <a href="#">java.lang.ThreadLocal\$ThreadLocalMap</a>	2	48	368
 <a href="#">sun.util.calendar.Gregorian\$Date</a>	1	96	96
 <a href="#">java.lang.StringBuilder</a>	1	16	88
 <a href="#">java.security.AccessControlContext</a>	1	24	24
 <a href="#">char[]</a>	1	24	24
 <a href="#">java.lang.Object</a>	1	8	8
 <a href="#">java.lang.Class</a>	1	0	0
Σ Total: 11 entries	290,283	8,645,232	66,459,912

为了更多的了解 MAT 的功能，再举一些例子（不提供对应的代码）：

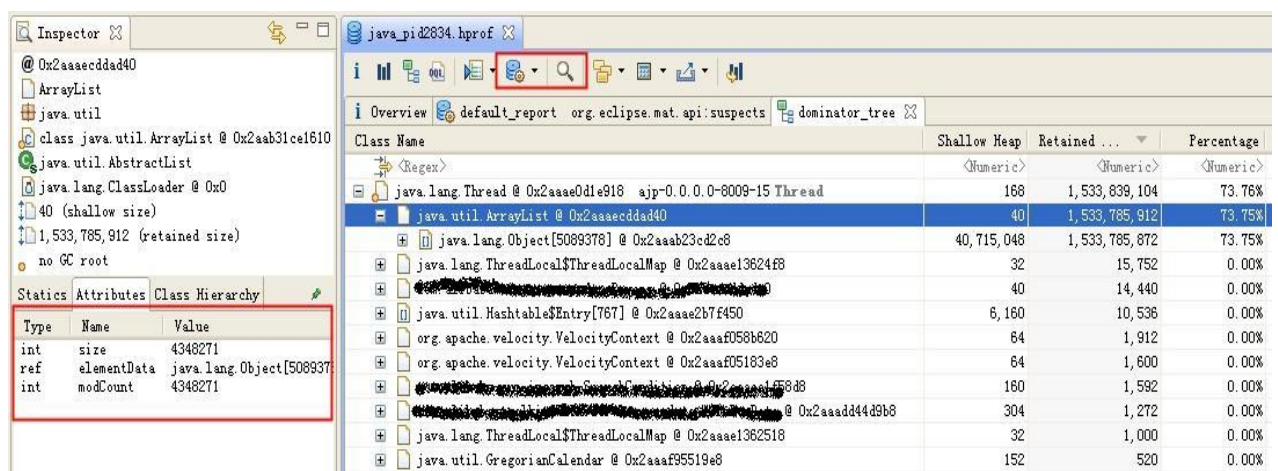
例子二：

通过 MAT 发现 heap dump 问题所在，就需要寻找导致内存泄漏的代码点。

这时往往需要打开对象依赖关系树形视图，点击如图按钮即可。



这时会看到如下视图：



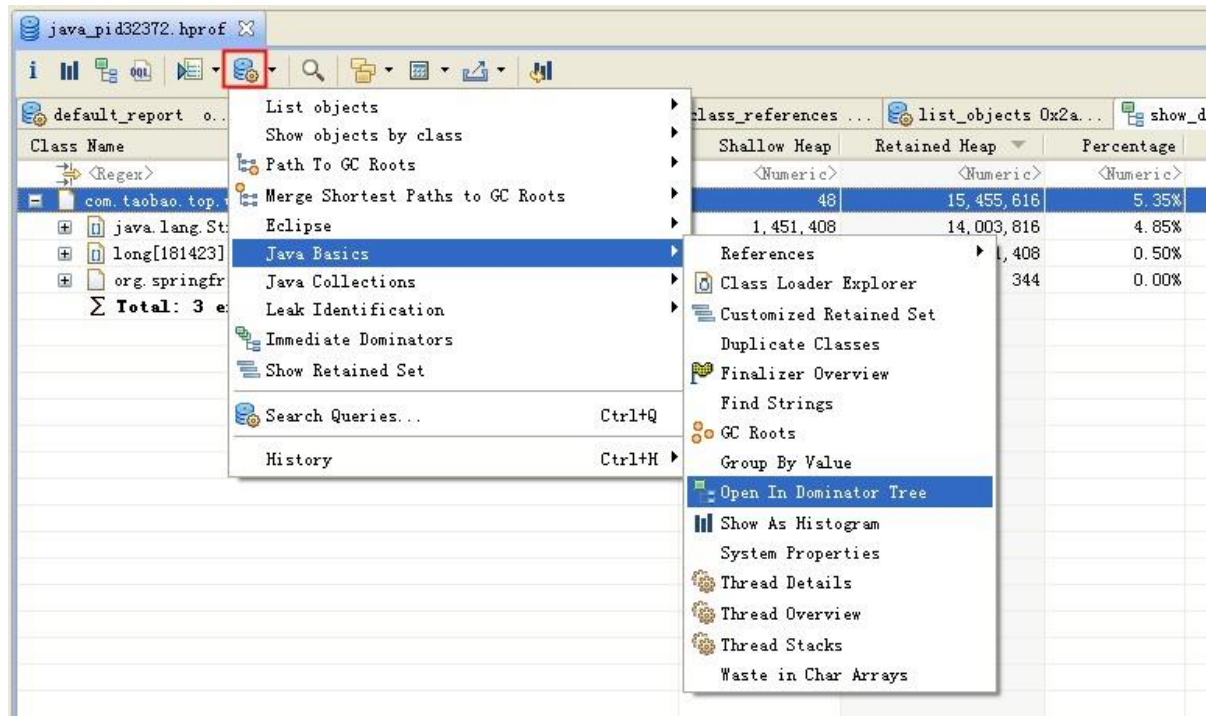
这个视图的右边大区域可以看到对象的依赖关系，选中某个对象以后可以在左边小窗口查看对象的一些属性。如果属性的值是一些内存地址你还可以点击工具栏的搜索按钮来搜索具体的对象信息。在进行具体分析的时候 MAT 只是起了帮助你进行分析的工具的功能，OOM 问题分析没有固定方法和准则。只能发挥你敏锐的洞察力，结合源代码，对内存中的对象进行分析从而找到代码中的 BUG.

例子三：

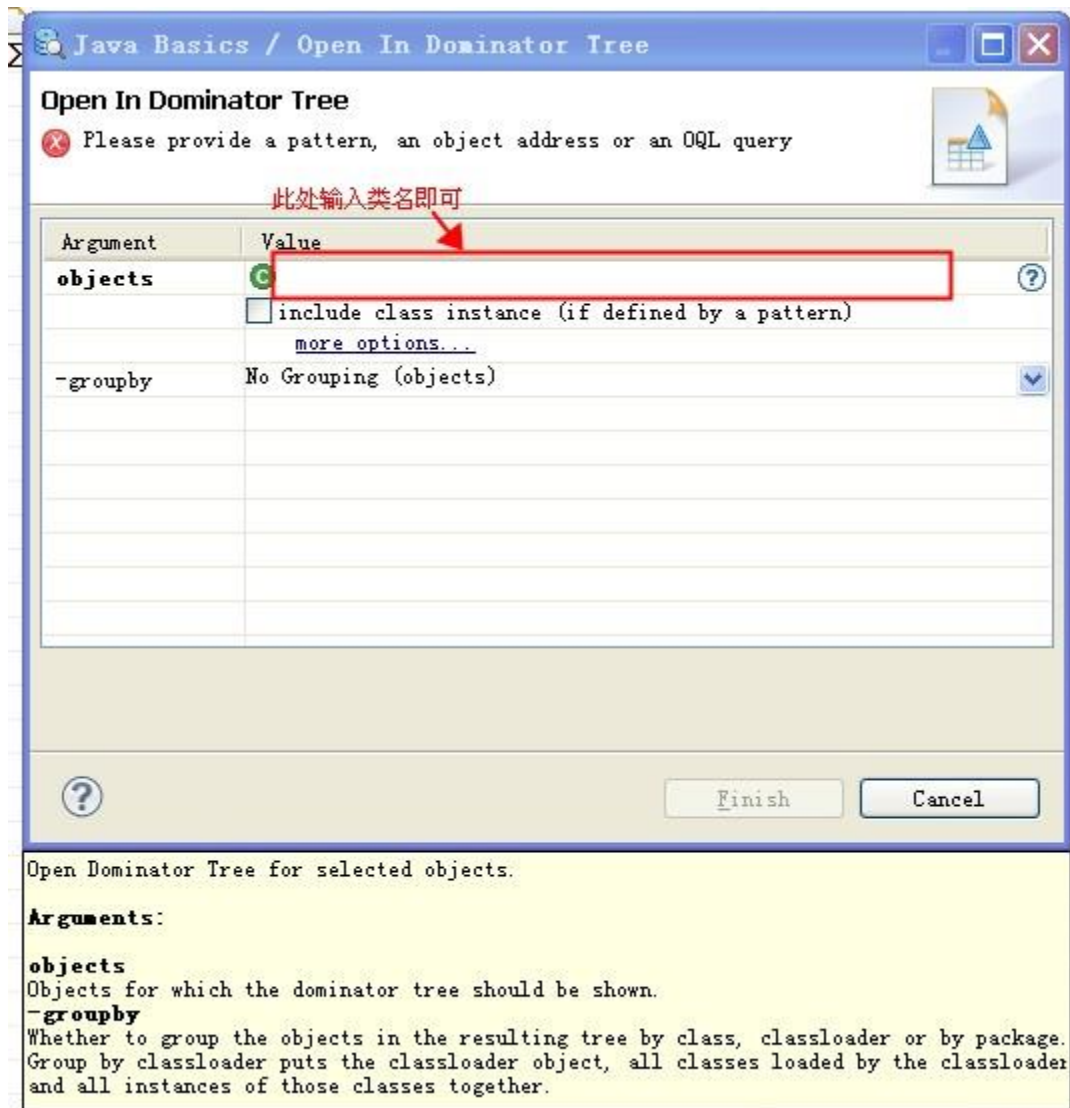
如何查看某一个对象占用的内存空间

1.按以下方式打开新窗口即可，如图：





2.输入类名（输入类的全名），如图：



参考资料：

<http://www.blogjava.net/rosen/archive/2010/06/13/323522.html>

<http://seanhe.iteye.com/blog/898277>

# 性能分析工具之-- Eclipse Memory Analyzer tool(MAT) ( 三 )

继 [性能分析工具之-- Eclipse Memory Analyzer tool\(MAT\) \( 一 \)](#) , [性能分析工具之-- Eclipse Memory Analyzer tool\(MAT\) \( 二 \)](#) 两篇文章之后, 接下来该讲述 Perm gen 引起的内存泄露问题的分析过程。

## perm gen

我们在上 2 篇文章中知道, perm gen 是个异类, 里面存储了类和方法数据(与 class loader 有关)以及 interned strings (字符串驻留)。在 heap dump 中没有包含太多的 perm gen 信息。那么我们就用这些少量的信息来解决问题吧。

看下面的代码, 利用 interned strings 把 perm gen 撑破了。

```
/**
 * OOMPermTest class
 * @author rosen jiang
 */
package org.rosenjiang.test;

public class OOMPermTest {

    public static void main(String[] args){

        oom();

    }

    private static void oom(){

        Object[] array = new Object[100000000];

        for(int i=0; i<100000000; i++){
```



```

        String d = String.valueOf(i).intern();

        array[i]=d;

    }

}

}

```

控制台打印如下的信息，然后把 java\_pid1824.hprof 文件导入到 MAT。其实在 MAT 里，看到的状况应该和 “OutOfMemoryError: Java heap space” 差不多（用了数组），因为 heap dump 并没有包含 interned strings 方面的任何信息。只是在这里需要强调，使用 intern()方法的时候应该多加注意。

```

java.lang.OutOfMemoryError: PermGen space

Dumping heap to java_pid1824.hprof ...

Heap dump file created [121273334 bytes in 2.845 secs]

Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
... ..
... ..
... ..
... ..

```

倒是在思考如何把 class loader 撑破废了些心思。经过尝试，发现使用 ASM 来动态生成类才能达到目的。ASM(<http://asm.objectweb.org>)的主要作用是处理已编译类(compiled class)，能对已编译类进行生成、转换、分析（功能之一是实现动态代理），而且它运行起来足够的快和小巧，文档也全面，实属居家必备之良品。ASM 提供了 core API 和 tree API，前者是基于事件的方式，后者是基于对象的方式，类似于 XML 的 SAX、DOM 解析，但是

使用 tree API 性能会有损失。既然下面要用到 ASM，这里不得不啰嗦下已编译类的结构，包括：

- 1、修饰符（例如 public、private）、类名、父类名、接口和 annotation 部分。
- 2、类成员变量声明，包括每个成员的修饰符、名字、类型和 annotation。
- 3、方法和构造函数描述，包括修饰符、名字、返回和传入参数类型，以及 annotation。

当然还包括这些方法或构造函数的具体 Java 字节码。

4、常量池(constant pool)部分，constant pool 是一个包含类中出现的数字、字符串、类型常量的数组。

Modifiers, name, super class, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code

已编译类和原来的类源码区别在于，已编译类只包含类本身，内部类不会在已编译类中出现，而是生成另外一个已编译类文件；其二，已编译类中没有注释；其三，已编译类没有 package 和 import 部分。

这里还得说说已编译类对 Java 类型的描述，对于原始类型由单个大写字母表示，Z 代表 boolean、C 代表 char、B 代表 byte、S 代表 short、I 代表 int、F 代表 float、J 代表 long、

D 代表 double ;而对类类型的描述使用内部名(internal name)外加前缀 L 和后面的分号共同表示来表示 , 所谓内部名就是带全包路径的表示法 , 例如 String 的内部名是 java/lang/String ; 对于数组类型 , 使用单方括号加上数据元素类型的方式描述。最后对于方法的描述 , 用圆括号来表示 , 如果返回是 void 用 V 表示 , 具体参考下图。

Java type	Type descriptor
boolean	Z
char	C
byte	B
short	S
int	I
float	F
long	J
double	D
Object	Ljava/lang/Object;
int []	[I
Object [] []	[[Ljava/lang/Object;

Method declaration in source file	Method descriptor
void m(int i, float f)	(IF)V
int m(Object o)	(Ljava/lang/Object;)I
int [] m(int i, String s)	(ILjava/lang/String;)[I
Object m(int [] i)	([I)Ljava/lang/Object;

下面的代码中会使用 ASM core API , 注意接口 ClassVisitor 是核心 , FieldVisitor、MethodVisitor 都是辅助接口。ClassVisitor 应该按照这样的方式来调用 visit visitSource? visitOuterClass? ( visitAnnotation | visitAttribute )\*( visitInnerClass | visitField | visitMethod )\* visitEnd。就是说 visit 方法必须首先调用 ,再调用最多一次的 visitSource , 再调用最多一次的 visitOuterClass 方法 , 接下来再多次调用 visitAnnotation 和 visitAttribute 方法 , 最后是多次调用 visitInnerClass、visitField 和 visitMethod 方法。调用完后再调用 visitEnd 方法作为结尾。

注意 ClassWriter 类 , 该类实现了 ClassVisitor 接口 , 通过 toByteArray 方法可以把已编译类直接构建成二进制形式。由于我们要动态生成子类 , 所以这里只对 ClassWriter 感兴趣。

首先是抽象类原型 :

```
/**
 * @author rosen jiang
 * MyAbsClass class
 */
package org.rosenjiang.test;

public abstract class MyAbsClass {

    int LESS = -1;

    int EQUAL = 0;

    int GREATER = 1;

    abstract int absTo(Object o);

}
```

其次是自定义类加载器 , 实在没法 , ClassLoader 的 defineClass 方法都是 protected 的 , 要加载字节数组形式 ( 因为 toByteArray 了 ) 的类只有继承一下自己再实现。

```
/**
 * @author rosen jiang
```

```
* MyClassLoader class

*/

package org.rosenjiang.test;

public class MyClassLoader extends ClassLoader {

    public Class defineClass(String name, byte[] b) {

        return defineClass(name, b, 0, b.length);

    }

}
```

最后是测试类。

```
/**

 * @author rosen jiang

 * OOMPermTest class

 */

package org.rosenjiang.test;

import java.util.ArrayList;

import java.util.List;

import org.objectweb.asm.ClassWriter;

import org.objectweb.asm.Opcodes;
```

```
public class OOMPermTest {

    public static void main(String[] args) {

        OOMPermTest o = new OOMPermTest();

        o.oom();

    }

    private void oom() {

        try {

            ClassWriter cw = new ClassWriter(0);

            cw.visit(Opcodes.V1_5, Opcodes.ACC_PUBLIC + Opcodes.ACC_ABSTRACT,

                "org/rosenjiang/test/MyAbsClass", null, "java/lang/Object",

                new String[] {});

            cw.visitField(Opcodes.ACC_PUBLIC + Opcodes.ACC_FINAL + Opcodes.ACC_STATIC, "LESS", "I",

                null, new Integer(-1)).visitEnd();

            cw.visitField(Opcodes.ACC_PUBLIC + Opcodes.ACC_FINAL + Opcodes.ACC_STATIC, "EQUAL", "I",

                null, new Integer(0)).visitEnd();

            cw.visitField(Opcodes.ACC_PUBLIC + Opcodes.ACC_FINAL + Opcodes.ACC_STATIC, "GREATER", "I",

                null, new Integer(1)).visitEnd();

            cw.visitMethod(Opcodes.ACC_PUBLIC + Opcodes.ACC_ABSTRACT, "absTo",
```

```

"(Ljava/lang/Object;)I", null, null).visitEnd();

cw.visitEnd();

byte[] b = cw.toByteArray();


List<ClassLoader> classLoaders = new ArrayList<ClassLoader>();

while (true) {

    MyClassLoader classLoader = new MyClassLoader();

    classLoader.defineClass("org.rosenjiang.test.MyAbsClass", b);

    classLoaders.add(classLoader);

}

} catch (Exception e) {

    e.printStackTrace();

}

}

}

```

不一会儿，控制台就报错了。

```

java.lang.OutOfMemoryError: PermGen space

Dumping heap to java_pid3023.hprof ...

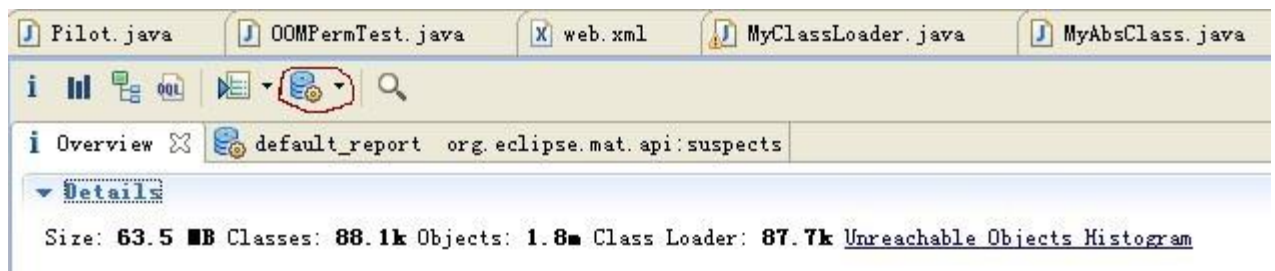
Heap dump file created [92593641 bytes in 2.405 secs]

Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
...

```

...

打开 java\_pid3023.hprof 文件，注意看下图的 Classes: 88.1k 和 Class Loader: 87.7k 部分，从这点可看出 class loader 加载了大量的类。



更进一步分析，点击上图中红框线圈起来的按钮，选择 Java Basics——Class Loader Explorer 功能。打开后能看到下图所示的界面，第一列是 class loader 名字；第二列是 class loader 已定义类(defined classes)的个数，这里要说一下已定义类和已加载类(loaded classes)了，当需要加载类的时候，相应的 class loader 会首先把请求委派给父 class loader，只有当父 class loader 加载失败后，该 class loader 才会自己定义并加载类，这就是 Java 自己的“双亲委派加载链”结构；第三列是 class loader 所加载的类的实例数目。



i Overview default_report org.eclipse.mat.api:suspects classloaderexplorerquery		
Class Name	Defined Cla...	No. of Instances
<Regex>	<Numeric>	<Numeric>
<system class loader>	444	1,670,445
sun.misc.Launcher\$AppClassLoader @ 0x22efe608	11	87,657
org.rosenjiang.test.MyClassLoader @ 0x229e0148	1	0
parent sun.misc.Launcher\$AppClassLoader @ 0x22efe608	11	87,657
org.rosenjiang.test.MyAbsClass		0
Σ Total: 2 entries		
org.rosenjiang.test.MyClassLoader @ 0x229e0428	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e0708	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e09e8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e0cc8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e0fa8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1288	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1568	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1848	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1b28	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1e08	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e20e8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e23c8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e26a8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e2988	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e2c68	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e2f48	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3228	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3508	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e37e8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3ac8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3da8	1	0
Σ Total: 24 of 87,659 entries	88,110	1,758,102

在 Class Loader Explorer 这里，能发现 class loader 是否加载了过多的类。另外，还有 Duplicate Classes 功能，也能协助分析重复加载的类，在此就不再截图了，可以肯定的是 MyAbsClass 被重复加载了 N 多次。

最后

其实 MAT 工具非常的强大，上面故弄玄虚的范例代码根本用不上 MAT 的其他分析功能，

所以就不再描述了。其实对于 OOM 不只我列举的两种溢出错误，还有多种其他错误，但我想说的是，对于 perm gen，如果实在找不出问题所在，建议使用 JVM 的 -verbose 参数，该参数会在后台打印出日志，可以用来查看哪个 class loader 加载了什么类，例：“[Loaded org.rosenjiang.test.MyAbsClass from org.rosenjiang.test.MyClassLoader]”。

全文完。

## 参考资料

[memoryanalyzer Blog](#)

[java 类加载器体系结构](#)

[ClassLoader](#)