# Java问题定位技术

2009 年 7 月 19 日

# 目 录

# 表格目录

# 插图目录

# 前言

目前已经出版了许多关于Java的书籍，但绝大多数书籍着重于介绍开发方面的主题。甚至同一主题的书籍，在市面上可以找到许多。与此形成鲜明对比的是，对于系统地介绍Java问题定位类的书籍却是少之又少，即使有这方面的内容，往往也是一笔带过。本书系统地介绍Java问题定位技术，我相信有一些很少公开的定位技术，在正确使用时，可以产生令人惊讶的效果。

采用Java开发的大型应用系统越来越大、越来越复杂；很多系统甚至是将很多第三方系统集成在一起，整个系统看起来像一个黑盒子。系统运行遭遇问题(系统停止响应，运行越来越慢，或者性能低下，甚至系统core dump)，如何迅速命中问题的根本原因是颇具挑战性的任务。这类问题的定位技巧是本文介绍的重点，借助这些技巧可以快速找到这些问题的突破口。

功能性的问题定位往往有很清晰的套路（如单步跟踪等），因此本书对此不做介绍。本书着重介绍稳定性和可靠性方面的问题定位技术，特别是那些在实验室难以发现的深层次的问题。本书将Java问题定位的方法体系化，提供一种以黑盒子方式进行问题定位的思路：如何使用线程堆栈进行性能瓶颈分析？如何分析内存泄漏？如何分析系统挂死？在掌握本书所介绍的方法后，很多情况下无需对系统了解就可以对这类问题进行定位。

对于可靠性和稳定性等非功能需求很多时候难以验证，我们不可能写出"万能的"测试用例来发现系统所有的可靠性和稳定性问题，所以非功能特性缺陷带来的灾难和难以验证的特点需要我们采取一切可能的办法进行提前预防，将非功能性需求表现在代码中，是一个优秀程序工程师的功底的一个体现。因此本书除了介绍"事后"定位技术以外，同时还介绍了大量的事前预防技术，对一些严重影响稳定性或者可靠性问题相关的陷阱进行了深入分析,它们正是大型系统容易忽略但对系统稳定性和可靠性有巨大影响的暗礁，如果能在系统的设计和编码阶段就防止埋上这些"地雷"，那么就不需要事后补救这种代价极高的维护成本。

本书旨在让软件能表现出"工业强度"，即一个"产品级别"的软件，"产品级别"与"功能完备"是完全不同的两个概念。验收测试的通过与系统能否承受实际应用中的压力完全是两回事。经过验收测试的系统有可能在真实的环境下表现得一塌糊涂，也有可能非常地棒。举个例子，我们能保证不存在内存泄漏吗？没人会在测试服务器中在完全模拟实际的负载的情况下对系统进行一个周或者一个月的测试。因此，通过QA并不能保证没有内存泄漏的发生，因此它很容易就被带入到产品中。内存泄漏情况大多是与流量相关的，也就是说，流量越大，内存泄漏的速度就越快。这意味着你根本无法预测什么时候要重启程序，问题往往发生在系统最忙的时候，墨菲法则往往就在这个时候生效。"产品级别"的另一个方面是系统对所谓"瞬时峰值"的应对能力，也就是应对系统的短暂性冲击的能力。经过短暂峰值的冲击的系统能否自动恢复？很多系统经过短暂的峰值冲击，往往不能恢复，这常常是由于异常情况没有很好地进行"善后处理",导致大量资源泄漏，比如数据库连接泄露，一旦衰退开始，系统崩溃就只是迟早的问题了。要想将功能完备软件变为产品级软件，系统要有一套完整的异常处理机制，对异常进行了合适的"善后处理",避免由于异常导致的资源泄漏等问题。本书对这些具有坏味道的代码也进行了深入剖析。

本书介绍的定位技术主要有：内存泄漏定位，线程堆栈分析等。内存定位套路比较固定，但线程堆栈分析需要一定的火候，它需要一定的悟性和长期的修炼。在可靠性和稳定性问题的定位中，线程堆栈分析是最有力的武器，掌握了这个定位工具，会大大增强自己的"内功"。

本书适合如下人员阅读：

- 开发的应用属于7*24的应用，并要求99.999%(俗称5个9)的高稳定性高可靠性。
- 开发的应用属于大型应用，每个人只熟悉系统的一小部分。
- 对下面一些问题模糊不清的开发人员：
  - 将不用的对象设为null,就可以避免内存泄漏.
  - 由于JVM自动进行内存管理，因此java中不会有内存泄漏.
  - unix/linux下使用top观察到内存上升，可以断定程序存在内存泄漏.
  - 线程不安全的HashMap并发读是不会出现问题的.
  - 字符集和字符编码的关系。
  - "系统挂死"、"宕机"感觉比较抽象。
- 负责对系统进行优化维护的开发人员。
- 致力于开发大型可靠系统的开发人员。

  本书实用性强，定位疑难问题命中率高。

——张民卫——

## 作者简介

# §1 Java线程堆栈分析

如果您是从C++/C转到Java上的程序员，那么线程堆栈应该不是陌生的技术，但对于原生的Java程序员来说，很多人不清楚还有这个这个密门绝技。什么是线程堆栈[1]？线程堆栈也称作线程调用堆栈。Java线程堆栈是虚拟机中线程（包括锁）状态的一个瞬间快照，即系统在某个时刻所有线程的运行状态,包括每一个线程的调用堆栈，锁的持有情况等信息。每一种Java虚拟机(SUN JVM、IBM JVM、JRokit、GNU JVM等等)都提供了线程转储(thread dump)的后门，通过这个后门可以将那个时刻的线程堆栈打印出来。虽然各种Java虚拟机在线程堆栈的打印输出格式上有一些不同，但是线程堆栈的信息都包含：

1. 线程的名字，ID，线程的数量等。
2. 线程的运行状态，锁的状态（锁被哪个线程持有，哪个线程再等待锁等）。
3. 调用堆栈（即函数的调用层次关系）。调用堆栈包含完整的类名，所执行的方法，源代码的行数。

具体打印出的堆栈信息内容多少依赖于你的系统的复杂程度，也许从几十行到上万行。借助线程堆栈，可以分析许多问题，如线程死锁、锁争用、死循环、识别耗时操作等等。在多线程场合下的稳定性问题分析和性能问题分析，线程堆栈分析是最有效的方法，在多数情况下甚至无需对系统了解就可以进行相应的分析。

由于线程堆栈是系统当时某个时刻的线程运行状况（即瞬间快照），对于已经消失而又没留有痕迹的信息，线程堆栈是无法进行历史追踪的。这种情况下，只能结合日志进行分析。如连接池中的连接被哪些线程使用了而没有释放这类问题。尽管如此，总的来说，线程堆栈是多线程类应用程序非功能型问题定位的最有效手段，可以说是杀手锏。线程堆栈最善于分析如下类型的问题:

- 系统无缘无故CPU过高。
- 系统挂起，无响应。
- 系统运行越来越慢。
- 性能瓶颈（如无法充分利用CPU等）
- 线程死锁、死循环，饿死等。
- 由于线程数量太多导致系统失败（如无法创建线程等）。

借助线程堆栈会帮助我们迅速地缩小问题的范围，找到突破口，命中目标。本章对线程堆栈进行详细的介绍，包括如下内容：

- 如何输出线程堆栈?

[1]也叫做Thread dump或者trace stack

- 如何解读线程堆栈?
- 如何借助线程堆栈进行问题分析?
- 线程堆栈不能分析什么类型的问题?

本书只所以先开门见山地首先介绍线程堆栈技术，是因为该技术是分析可靠性、稳定性、性能问题的最有力的技术，以笔者的经验，大约有50%以上的问题可以通过堆栈分析得以快速精确定位。同时线程堆栈分析很多时候并不需要源代码，这在很多场合，具有无可比拟的优势。笔者采用该技术，曾经定位/解决了多个几乎不可能完成的任务(详见第 §13.22节第 216)，下面我们就开始我们的线程堆栈之旅。

### §1.1 如何输出线程堆栈?

Java虚拟机提供了线程转储(Thread dump)的后门，通过这个后门，可以将线程堆栈打印出来。这个后门就是通过向Java进程发送一个QUIT信号，Java虚拟机收到该信号之后，将系统当前的JAVA线程调用堆栈打印出来。有的虚拟机实现（如SUN JDK）堆栈信息将打印在屏幕上。另外有的虚拟机实现（如IBM JDK）直接将线程堆栈打印到一个文件中，从当前的运行目录下可以找到该文件。如果JDK将线程堆栈打印在屏幕上，由于信息量太大（一般的系统都有几千行或者几万行），经常会超出控制台缓冲区的最大行数限制造成信息丢失，因此最好手工进行重定向到一个文件中。在Windows下和Unix/Linux下，通过如下的命令行方式向Java进程请求堆栈输出：

**windows** 在运行java的控制台窗口上按<ctrl> + <break>组合键。

**unix/Linux** 使用kill -3 <java pid>[2]

在AIX上用IBM的JVM，需要进行以下设置,kill -3才可以有效进行线程转储：

```
export IBM_HEAPDUMP=true
export IBM_HEAP_DUMP=true
export IBM_HEAPDUMP_OUTOFMEMORY=true
export IBM_HEAPDUMPDIR=<directory path>
```

同时请确保Java命令行中没有DISABLE_JAVADUMP运行选项。按照上面介绍的方法之后，就可以打印线程堆栈了。

在Unix下如果是以后台方式启动的java进程，打印的线程堆栈会和其它屏幕输出一样，在控制台已经被关闭的情况下，这些信息你无法"捡"回它们。因此为了避免这种情况，在启动时系统时最好做一下重定向。重定向符号有如下

两个，

\> 将屏幕输出写入到文件中，重写文件内容。

\>\> 将屏幕输出添加到文件末尾。

特别地，在linux/unix下使用如下的方式进行重定向：

```
myrun.sh > run.log  2>&1
```

在操作系统中，0,1,2分别表示输入/输出流，含义如下：

**0-** 标准输入，即C中的stdin，或者C++中的cin，或者Java中的System.in

**1-** 标准输出，即C中的stdout，或者C++中的cout，或者Java中的System.out

**2-** 错误输出，即C中的stderr，或者C++中的cerr，或者Java中的System.err

[2]即Java进程ID

2>&1表示将错误输出重定向到标准输出流中，即将标准输出和错误输出都重定向到一个文件中。

> ☞ 提示：
>
> 在JDK1.5以上的版本中，可以在Java程序中通过Thread.getStackTrace()控制堆栈自动打印.通过这种方式，线程堆栈的打印时机可编程。通过手工编程，可以在满足某些条件时，将线程堆栈自动打印。

## §1.2 如何解读线程堆栈?

下面通过一个实际的例子对线程堆栈的解读进行详细介绍。掌握了线程堆栈解读的方法，就可以庖丁解牛，对线程堆栈进行深入剖析。

### §1.2.1 线程的解读

如下面一段Java源代码程序：

```
public class MyTest {
    Object obj1 = new Object();
    Object obj2 = new Object();
    public void fun1()
    {
        synchronized(obj1){
            fun2();
        }
    }
    public void fun2()
    {
        synchronized(obj2){
            while(true){ //为了演示需要，该函数永不退出
                System.out.print("");
            }
        }
    }
    public static void main(String[] args) {
        MyTest aa = new MyTest();
        aa.fun1();
    }
```

运行该程序: java MyTest,通过上节介绍的方法打印线程堆栈，打印的线程堆栈如下(Linux下)[3]：

```
Full thread dump Java HotSpot(TM) Client VM (1.5.0_08-b03 mixed mode, sharing):

"Low Memory Detector" daemon prio=1 tid=0x080a5848 nid=0xd2e runnable  //第(1)个线程
"CompilerThread0" daemon prio=1 tid=0x080a42a0 nid=0xd2d waiting on condition//(2)
"Signal Dispatcher" daemon prio=1 tid=0x080a31d8 nid=0xd2c runnable          //(3)
"Finalizer" daemon prio=1 tid=0x0809c660 nid=0xd2b in Object.wait()          //(4)
    at java.lang.Object.wait(Native Method)
    - waiting on <0xc8bf06c8> (a java.lang.ref.ReferenceQueue$Lock)
```

[3]为了排版需要，将某些线程的地址信息给删除掉了，这些信息在问题分析中用处不大，如："Reference Handler" daemon prio=1 tid=0x0809b970 nid=0xd2a in Object.wait()[0xf26d9000..0xf26da0b0]中后面的[]中的信息给略掉了。

```
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
    - locked <0xc8bf06c8> (a java.lang.ref.ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
    at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=1 tid=0x0809b970 nid=0xd2a in Object.wait()  //(5)
    at java.lang.Object.wait(Native Method)
    - waiting on <0xc8bf05d8> (a java.lang.ref.Reference$Lock)
    at java.lang.Object.wait(Object.java:474)
    at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
    - locked <0xc8bf05d8> (a java.lang.ref.Reference$Lock)

"main" prio=1 tid=0x0805c988 nid=0xd28 runnable [0xfff65000..0xfff659c8]     //(6)
    at java.lang.String.indexOf(String.java:1352)
    at java.io.PrintStream.write(PrintStream.java:460)
    - locked <0xc8bf87d8> (a java.io.PrintStream)
    at java.io.PrintStream.print(PrintStream.java:602)
    at MyTest.fun2(MyTest.java:16)
    - locked <0xc8c1a098> (a java.lang.Object)
    at MyTest.fun1(MyTest.java:8)
    - locked <0xc8c1a090> (a java.lang.Object)
    at MyTest.main(MyTest.java:26)

"VM Thread" prio=1 tid=0x08098d88 nid=0xd29 runnable                          //(7)

"VM Periodic Task Thread" prio=1 tid=0x080a6d30 nid=0xd2f waiting on condition//(8)
```

在这段堆栈输出中可以看出，我们看出系统当前共有如下线程：Low Memory Detector、CompilerThread0、Signal Dispatcher、Finalizer、Reference Handler、main、VM Thread、VM Periodic Task Thread共八个，其中只有main线程属于Java用户线程，其它七个都是由虚拟机自动创建的,如果是java界面程序，虚拟机还会自动创建事件分发线程awt-eventqueue等，我们在实际分析的过程中，只关心Java用户线程即可。

从上面的main线程中看，线程堆栈里面的最直观的信息是当前线程的调用上下文，即从哪个函数中调用到哪个函数中（从下往上看），正执行到哪个类的哪一行，借助这些信息，我们就对当前系统正在做什么就一目了然。线程堆栈在分析问题中的作用请见后面的章节。其中一个线程的某一层调用含义如下[4]：

[4]如果括号中没有显示Java源代码文件名，可能是由于系统运行期间启动了JIT,JIT详见第 142页第 §7.3节。

```
    at MyTest.fun1(MyTest.java:8)
         |     |         |       |
         |     |         |       +-----当前正在调用的函数所在的源代码文件的行号
         |     |         +------------当前正在调用的函数所在的源代码文件
         |     +----------------------当前正在调用的方法名
         +----------------------------当前正在调用的类名
```

另外，从main线程的堆栈中，有"- locked <0xc8c1a090> (a java.lang.Object)"语句,这表示该线程(即main线程)已经占有了锁<0xc8c1a090>，其中0xc8c1a090表示锁ID，这个锁的ID是系统自动产生的，我们只需要知道每次打印的堆栈，同一个ID表示是同一个锁即可[5]。每一个线程堆栈的第一行含义如下：

```
"main" prio=1 tid=0x0805c988 nid=0xd28 runnable [0xfff65000..0xfff659c8]
    |    |       |              |          |        |
    |    |       |              |          |        +--线程占用内存地址
    |    |       |              |          +-----------线程的状态
    |    |       |              +----线程对应的本地线程id号
    |    |       +--------------------线程id
    |    +--------------------------线程优先级
    +-------------------------------线程名称
```

其中"线程对应的本地线程id号"所指的"本地线程"是指该Java线程所对应的虚拟机中的本地线程。我们知道Java是解析型语言，执行的实体是Java虚拟机，因此Java语言中的线程是依附于Java虚拟机中的本地线程来运行的，实际上是本地线程在执行Java线程代码。Java代码中创建一个thread，虚拟机在运行期就会创建一个对应的本地线程，而这个本地线程才是真正的线程实体。为了更加深入得理解本地线程和Java线程的关系，在Unix/Linux下，我们可以通过如下方式把Java虚拟机的本地线程打印出来：

1. 使用ps -ef | grep java 获得Java进程ID。
2. 使用pstack <java pid>获得Java虚拟机的本地线程的堆栈[6]。

本例中，我们获取的本地线程堆栈如下：

```
Thread 8 (Thread 4067802000 (LWP 3369)):
#0  0xffffe402 in __kernel_vsyscall ()
#1  0x0082042c in pthread_cond_timedwait@@GLIBC_2.3.2 ()
#2  0x008208d5 in pthread_cond_timedwait@GLIBC_2.0 () from /lib/libpthread.so.0
#3  0xf7ab9e4c in os::Linux::safe_cond_timedwait ()
#4  0xf7aa5d71 in Monitor::wait ()
#5  0xf7b5c25b in VMThread::loop ()
```

[5]在有的虚拟机实现中，即使是同一个锁变量，当多次打印堆栈时，每次堆栈打印的锁ID也是不同的。

[6]pstack可以打印出本地程序的线程堆栈，有的操作系统下，打印本地堆栈的命令是gstack

```
#6  0xf7b5bec0 in VMThread::run ()
#7  0xf7ababe8 in _start ()
#8  0x0081c3db in start_thread () from /lib/libpthread.so.0
#9  0x0077c06e in clone () from /lib/libc.so.6
Thread 7 (Thread 4067273616 (LWP 3370)):
#0  0xffffe402 in __kernel_vsyscall ()
#1  0x008201a6 in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x0082085e in pthread_cond_wait@GLIBC_2.0 () from /lib/libpthread.so.0
#3  0xf7ab9cee in os::Linux::safe_cond_wait ()
#4  0xf7aafcef in ObjectMonitor::wait ()
#5  0xf7b06176 in ObjectSynchronizer::wait ()
#6  0xf7a02b03 in JVM_MonitorWait ()
#7  0xf287a4db in ?? ()
#8  0x0809ba30 in ?? ()
#9  0xf26d9fcc in ?? ()
#10 0x00000000 in ?? ()
Thread 6 (Thread 4066745232 (LWP 3371)):
#0  0xffffe402 in __kernel_vsyscall ()
#1  0x008201a6 in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x0082085e in pthread_cond_wait@GLIBC_2.0 () from /lib/libpthread.so.0
#3  0xf7ab9cee in os::Linux::safe_cond_wait ()
#4  0xf7aafcef in ObjectMonitor::wait ()
#5  0xf7b06176 in ObjectSynchronizer::wait ()
#6  0xf7a02b03 in JVM_MonitorWait ()
#7  0xf287a4db in ?? ()
#8  0x0809c720 in ?? ()
#9  0xf2658f1c in ?? ()
#10 0x00000000 in ?? ()
Thread 5 (Thread 4063869840 (LWP 3372)):
#0  0xffffe402 in __kernel_vsyscall ()
#1  0x008221ae in sem_wait@GLIBC_2.0 () from /lib/libpthread.so.0
#2  0xf7abb046 in check_pending_signals ()
#3  0xf7ab7f4d in os::signal_wait ()
#4  0xf7ab5285 in signal_thread_entry ()
#5  0xf7b233d3 in JavaThread::run ()
#6  0xf7ababe8 in _start ()
#7  0x0081c3db in start_thread () from /lib/libpthread.so.0
#8  0x0077c06e in clone () from /lib/libc.so.6
Thread 4 (Thread 4063341456 (LWP 3373)):
#0  0xffffe402 in __kernel_vsyscall ()
#1  0x008201a6 in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x0082085e in pthread_cond_wait@GLIBC_2.0 () from /lib/libpthread.so.0
#3  0xf7ab9cee in os::Linux::safe_cond_wait ()
```

```
#4  0xf7aa5e34 in Monitor::wait ()
#5  0xf793658e in CompileQueue::get ()
#6  0xf7938242 in CompileBroker::compiler_thread_loop ()
#7  0xf7b28da6 in compiler_thread_entry ()
#8  0xf7b233d3 in JavaThread::run ()
#9  0xf7ababe8 in _start ()
#10 0x0081c3db in start_thread () from /lib/libpthread.so.0
#11 0x0077c06e in clone () from /lib/libc.so.6
Thread 3 (Thread 4062813072 (LWP 3374)):
#0  0xffffe402 in __kernel_vsyscall ()
#1  0x008201a6 in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x0082085e in pthread_cond_wait@GLIBC_2.0 () from /lib/libpthread.so.0
#3  0xf7ab9cee in os::Linux::safe_cond_wait ()
#4  0xf7aa5cc1 in Monitor::wait ()
#5  0xf7a8e31f in LowMemoryDetector::low_memory_detector_thread_entry ()
#6  0xf7b233d3 in JavaThread::run ()
#7  0xf7ababe8 in _start ()
#8  0x0081c3db in start_thread () from /lib/libpthread.so.0
#9  0x0077c06e in clone () from /lib/libc.so.6
Thread 2 (Thread 4062284688 (LWP 3375)):
#0  0xffffe402 in __kernel_vsyscall ()
#1  0x0082042c in pthread_cond_timedwait@@GLIBC_2.3.2 ()
#2  0x008208d5 in pthread_cond_timedwait@GLIBC_2.0 () from /lib/libpthread.so.0
#3  0xf7ab8b38 in os::sleep ()
#4  0xf7b22418 in WatcherThread::run ()
#5  0xf7ababe8 in _start ()
#6  0x0081c3db in start_thread () from /lib/libpthread.so.0
#7  0x0077c06e in clone () from /lib/libc.so.6
Thread 1 (Thread 4160560000 (LWP 3368)):
#0  0xf28fc863 in ?? ()
#1  0x00000000 in ?? ()
#0  0xf28fc863 in ?? ()
```

从操作系统打印出的虚拟机的本地线程看，本地线程数量和Java线程堆栈中的线程数量相同，都是8个。说明二者是一一对应的。其中本地线程各项含义如下：

```
Thread 1 (Thread 4160560000 (LWP 3368)):
    |          |                |
    |          |                +----本地线程id(另一种表示,LWP-light weight process)
    |          +-------------------本地线程id
    +------------------------------线程名称
```

但是这个本地线程号如何与Java Thread Dump文件中对应起来呢？很简单，在Java Thread Dump文件中，每个线程都有tid=...nid=...的属性，通过这些属性可以对应到相应的本地线程,

我们先看Java线程的第一行，里面有一个属性为"nid=",如：

```
"main" prio=1 tid=0x0805c988 nid=0xd28 runnable [0xfff65000..0xfff659c8]
                                |
                                +-----线程对应的本地线程id号
```

其中nid就是native thread id，也就是指的本地线程中的LWPID，二者是相同的，只不过java线程中的nid中用16进制来表示，而本地线程中的id用十进制表示。例如上面的例子中3368的十六进制表示为0xd28.在Java线程中查找nid=0xd28即是本地线程对应Java线程。即：

```
"main" prio=1 tid=0x0805c988 nid=0xd28 runnable [0xfff65000..0xfff659c8]
    at java.lang.String.indexOf(String.java:1352)
    at java.io.PrintStream.write(PrintStream.java:460)
    - locked <0xc8bf87d8> (a java.io.PrintStream)
    at java.io.PrintStream.print(PrintStream.java:602)
    at MyTest.fun2(MyTest.java:16)
    - locked <0xc8c1a098> (a java.lang.Object)
    at MyTest.fun1(MyTest.java:8)
    - locked <0xc8c1a090> (a java.lang.Object)
    at MyTest.main(MyTest.java:26)
```

二者是表示是同一个线程。在本例中，Java线程和本地线程的映射关系(即java线程中nid与本地线程中lwp属性相等的)见图1。

3368（十进制）=0xd28(十六进制) 说明二者是同一个线程

| 本地线程堆栈 | Java线程堆栈 |
|---|---|
| Thread 1 (Thread 4160560000 (LWP 3368)) | "main" prio=1 tid=0x0805c988 nid=0xd28 runnable ... |
| ♮0 0xf28fc863 in ?? () | at java.lang.String.indexOf(String.java:1352) |
| ♮1 0x00000000 in ?? () | at java.io.PrintStream.write(PrintStream.java:460) |
| ♮0 0xf28fc863 in ?? () | - locked <0xc8bf87d8> (a java.io.PrintStream) |
| | at java.io.PrintStream.print(PrintStream.java:602) |
| | at MyTest.fun2(MyTest.java:16) |
| | - locked <0xc8c1a098> (a java.lang.Object) |
| | at MyTest.fun1(MyTest.java:8) |
| | - locked <0xc8c1a090> (a java.lang.Object) |
| | at MyTest.main(MyTest.java:26) |

图 1 本地线程和Java线程的映射

上面例子中，本地线程和Java线程的映射关系如下：

**表 1 Java线程和本地线程的映射关系**

| Native Thread(LWP) | JavaThread(nid) |
|---|---|
| 3368 | 0xd28 |
| 3369 | 0xd29 |
| 3370 | 0xd2a |
| 3371 | 0xd2b |
| 3372 | 0xd2c |
| 3373 | 0xd2d |
| 3374 | 0xd2e |
| 3375 | 0xd2f |

从上面的分析可以看出，Java线程实际上和本地线程指的是同一个东西，只有本地线程才是真正的线程实体，Java线程实际上就是指这个本地线程，它并不是一个另外存在的的实体。关于本地线程的作用，在后面的章节有介绍。下面我们继续介绍线程堆栈中的其它标识：

```
"main" prio=1 tid=0x0805c988 nid=0xd28 runnable [0xfff65000..0xfff659c8]
```

中的"runnable"表示当前线程处于运行状态。这个runnable状态是从虚拟机的角度来看的,表示这个线程正在运行。但是处于Runnable状态的线程不一定真地消耗CPU. 处于Runnable的线程只能说明该线程没有阻塞在java的wait或者sleep方法上，同时也没等待在锁上面。但是如果该线程调用了本地方法[7]，而本地方法处于等待状态，这个时候虚拟机是不知道本地代码中发生了什么[8]，此时尽管当前线程实际上也是阻塞的状态，但实际上显示出来的还是runnable状态，这种情况下是不消耗CPU的。如下面的线程堆栈：

```
"Thread-243" prio=1 tid=0xa58f2048 nid=0x7ac2 runnable [0xaeedb000..0xaeedc480]
    at java.net.SocketInputStream.socketRead0(Native Method)
    at java.net.SocketInputStream.read(SocketInputStream.java:129)
    at oracle.net.ns.Packet.receive(Unknown Source)
    at oracle.net.ns.DataPacket.receive(Unknown Source)
    at oracle.net.ns.NetInputStream.getNextPacket(Unknown Source)
    at oracle.net.ns.NetInputStream.read(Unknown Source)
    at oracle.jdbc.driver.T4CMAREngine.getNBytes(T4CMAREngine.java:1520)
    at oracle.jdbc.driver.T4CMAREngine.unmarshalNBytes()
    at oracle.jdbc.driver.T4CLongRawAccessor.readStreamFromWire()
    at oracle.jdbc.driver.T4CLongRawAccessor.readStream()
    at oracle.jdbc.driver.T4CInputStream.getBytes(T4CInputStream.java:70)
```

[7]有两种可能会调用本地方法，一种是调用到用户手工写的JNI本地代码中,另一种Java自身提供的API调用到了本地代码中，像at java.net.SocketInputStream.socketRead0(Native Method)中的"Native Method"就表示当前调用正在本地方法中.

[8]但操作系统是知道的，pstack就是操作提供的一个命令，它知道当前线程正在执行的本地代码上下文。

```
    - locked <0x934f4258> (a oracle.jdbc.driver.T4CInputStream)
    - locked <0x6b0dd600> (a oracle.jdbc.driver.T4CConnection)
    at oracle.jdbc.driver.OracleInputStream.needBytes()
    ... ...
    at org.hibernate.loader.Loader.list(Loader.java:1577)
    at org.hibernate.loader.hql.QueryLoader.list()
    at com.wes.timer.TimerTaskImpl.execute(TimerTaskImpl.java:627)
    - locked <0x80df8ce8> (a com.wes.timer.TimerTaskImpl)
    at com.wes.threadpool.RunnableWrapper.run(RunnableWrapper.java:209)
    at com.wes.threadpool.PooledExecutorEx$Worker.run()
    at java.lang.Thread.run(Thread.java:595)
```

该线程处于runnable状态，而它正在调用如下的本地方法:

```
    at java.net.SocketInputStream.socketRead0(Native Method)
```

但实际上像读socket的本地方法大多数时间是阻塞的。除非socket的缓冲区中有数据，底层的TCP/IP协议栈将唤醒阻塞的线程。这里仅想说明"runnable"状态不意味这个线程正在消耗CPU。因此我们在分析哪个线程在消耗大量CPU时，不能以这个"runnable"字样作为判断该线程是否消耗CPU的依据。

另外，我们常在线程堆栈中发现".<init>"或者".<clinit>"字样的函数，比如下面两个堆栈信息:

```
"Thread-5"  prio=1 tid=0xa58f2048 nid=0x7ac2 runnable [...]
    at java.lang.UNIXProcess.forkAndExec(Native Method)
    at java.lang.UNIXProcess.<init>;(UNIXProcess.java:156)
    at java.lang.Runtime.execInternal(Native Method)
    at java.lang.Runtime.exec(Runtime.java:568)
    at java.lang.Runtime.exec(Runtime.java:433)
    at TestApply.main(TestApply.java:14)
```

又如：

```
"main" prio=10 tid=0x08074680 nid=0x1 waiting for monitor entry [...]
    at java.util.logging.LogManager.addLogger(LogManager.java:322)
    - waiting to lock <0xb5627710> (a java.util.logging.LogManager)
    at java.util.logging.LogManager$1.run(LogManager.java:180)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.util.logging.LogManager.<clinit>(LogManager.java:156)
    at test.main(test.java:14)
```

那么".<clinit>"和".<init>"各表示什么含义呢？实际上，".<clinit>"表示当前正在执行类的初始化。".<init>"正在执行对象的构造函数。如下：

```
    at java.lang.UNIXProcess.<init>;(UNIXProcess.java:156)
                                 |
                                 +----正在执行UNIXProcess对象的构造函数
```

```
        at java.util.logging.LogManager.<clinit>(LogManager.java:156)
                                         |
                                         +--正在执行LogManager类的初始化
```

下面详细介绍一下类的初始化和对象的初始化。

**类初始化** 类"初始化"阶段，它是一个类或接口被首次使用的前阶段中的最后一项工作，本阶段负责为类变量赋予正确的初始值。Java 编译器把所有的类变量初始化语句和类型的静态初始化器通通收集到<clinit> 方法内，该方法只能被Jvm 调用，专门承担初始化工作。除接口以外，初始化一个类之前必须保证其直接超类已被初始化，并且该初始化过程是由Jvm 保证线程安全的。另外，并非所有的类都会拥有一个<clinit>() 方法，在以下条件中该类不会拥有<clinit>() 方法：

- 该类既没有声明任何类变量，也没有静态初始化语句；
- 该类声明了类变量，但没有明确使用类变量初始化语句或静态初始化语句初始化；
- 该类仅包含静态final 变量的类变量初始化语句，并且类变量初始化语句是编译时常量表达式。

**对象初始化** 对象实例化和初始化是就是对象生命的起始阶段的活动，在这里我们主要讨论对象的初始化工作的相关特点。Java 编译器在编译每个类时都会为该类至少生成一个实例初始化方法–即"<init>()" 方法。此方法与源代码中的每个构造方法相对应，如果类没有明确地声明任何构造方法，编译器则为该类生成一个默认的无参构造方法，这个默认的构造器仅仅调用父类的无参构造器，与此同时也会生成一个与默认构造方法对应的"<init>()" 方法. 通常来说，<init>() 方法内包括的代码内容大概为：调用另一个<init>() 方法；对实例变量初始化；与其对应的构造方法内的代码。如果构造方法是明确地从调用同一个类中的另一个构造方法开始，那它对应的<init>() 方法体内包括的内容为：一个对本类的<init>() 方法的调用；对应用构造方法内的所有字节码。如果构造方法不是通过调用自身类的其它构造方法开始，并且该对象不是Object 对象，那<init>() 法内则包括的内容为：一个对父类<init>() 方法的调用；对实例变量初始化方法的字节码；最后是对应构造子的方法体字节码。如果这个类是Object，那么它的<init>() 方法则不包括对父类<init>() 方法的调用。

另外，还有的时候，我们会发现堆栈信息里面包含"Native Method"，或者"Compiled Code"。

```
      at java.lang.UNIXProcess.forkAndExec(Native Method)
                                                |
              该方法是一个本地方法（JNI） ------+

      at org/apache/axis/client/Call.invoke(Call.java:2467)(Compiled Code)
                                                            |
                该class的方法已经被JIT编译成了本地代码------+
```

### §1.2.2 锁的解读

在介绍线程堆栈的解读方法之前，先介绍一点关于多线程的背景知识。即wait()和sleep()的重大区别.wait()和sleep() 有一个共同点，就是二者都会把当前的线程阻塞住（时长为函数参数指定的时间），我们称之为睡眠或者等待。但二者实际上是完全不同的两个函数，二者有着最为本质的区别：

**wait()** 当线程执行到wait()方法上，当前线程会释放监视锁，此时其它线程可以占有该锁，一旦wait()方法执行完成，当前线程又继续持有该锁，直到执行完该锁的作用域。可以说wait()是多线程场合下用得最多的一个方法。结合notify(),可以实现两个线程之间的通信,一个线程可以通过这种方法通知另一个线程继续执行，完成线程之间的配合。wait()和锁的示意图如下:

占有锁

```
synchronized(lock){
    map.put(new String("miller"),new Object());
    map.put(new String("mike"),new Object());
    ... ...
```

释放锁

```
    lock.wait(5000);
```

占有锁

```
    ... ...
    map.remove(new String("mike"))
    map.remove(new String("miller"))
}
```

图 2 含有wait(5000)的代码段锁的占用情况

在wait(5000)这5秒（5000毫秒）期间，当前线程会释放它占有的锁，此时其它线程有机会获得该锁。当wait(5000)执行完成后，当前线程继续获得该锁的使用权。满足如下条件之一，wait()方法退出：

- 达到了等待的时间之后，自动退出。如wait(5000),5秒后wait方法退出。
- 其它的线程调用了该锁的notify()方法。当如果多个线程在等待同一个锁，只有一个线程会被通知到。

> ☞ 提示：
>
> 正是由于wait()的这个特性（一旦执行到一个锁的wait()方法，该线程就会释放这个锁），所以可以有多个线程一起进入到同步块。

**sleep()** 与锁操作无关，如果该方法恰好在一个锁的保护范围之内，当前线程即使在执行sleep()的时候，仍然继续保持监视锁。该方法实际上仅仅是完成等待或者睡眠的语义。示意图如

下：

占有锁

占有锁

占有锁

```
synchronized(lock){
    map.put(new String("miller"),new Object());
    map.put(new String("mike"),new Object());
    ... ...

    Thread.sleep(5000);

    ... ...
    map.remove(new String("mike"))
    map.remove(new String("miller"))
}
```

图 3 含有sleep(5000)的代码段锁的占用情况

从上面的代码Thread.sleep(5000)可以看出，sleep()方法并不是锁上面的一个方法，而是线程的一个静态方法。也就是说该方法实际上是和锁操作无关的。如果sleep()方法恰好在一个锁的保护范围之内，那么当前线程即使执行到该sleep方法，也不会产生特别的锁操作(持有锁或者释放锁)，如果原来持有，现在仍然持有。如果原来没有持有，那么现在仍然不持有。

从上面介绍的线程堆栈看，线程堆栈中包含的直接信息为：线程的个数、每个线程调用的方法堆栈、当前锁的状态。线程的个数可以直接数出来；线程调用的方法堆栈，从下向上看，即表示当前的线程调用了哪个类上的哪个方法。而锁的状态看起来稍微有一点技巧。与锁相关的三个重要信息如下：

- 当一个线程占有一个锁的时候，线程堆栈中会打印—locked <0x22bffb60>
- 当一个线程正在等待其它线程释放该锁，线程堆栈中会打印—waiting to lock <0x22bffb60>
- 当一个线程占有一个锁，但又执行到该锁的wait()上，线程堆栈中首先打印locked,然后又会打印—waiting on <0x22c03c60>

例如下面的源代码：

```
package MyPackage;

public class ThreadTest {

    public static void main(String[] args) {
        Object shareobj = new Object();
        TestThread_Locked thread1 = new TestThread_Locked(shareobj);
        thread1.start();  //启动第一个线程
```

```

        TestThread_WaitingTo thread2 = new TestThread_WaitingTo(shareobj);
        thread2.start();  //启动第二个线程

        TestThread_WaitingOn thread3 = new TestThread_WaitingOn();
        thread3.start();  //启动第三个线程
    }
}

package MyPackage;

public class TestThread_Locked  extends Thread{
    Object lock = null;
    public TestThread_Locked(Object lock_)
    {
        lock = lock_;
        this.setName(this.getClass().getName());
    }
    public void run()
    {
        fun();
    }
    public void fun(){
        synchronized(lock){
            fun_longtime();
        }
    }
    public void fun_longtime(){
        try{
            Thread.sleep(20000);  //<---打印线程堆栈时，该线程运行到这里
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}

package MyPackage;

public class TestThread_WaitingOn extends Thread{
    Object lockobj1 = new Object();
    public TestThread_WaitingOn()
    {
```

```
            this.setName(this.getClass().getName());
        }
        public void run()
        {
            fun();
        }

        public void fun(){
            synchronized(lockobj1){
                fun_wait();
            }
        }
        public void fun_wait(){
            try{
                lockobj1.wait(100000);//<--- 打印线程堆栈时，该线程运行到这里
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
    }

    package MyPackage;

    public class TestThread_WaitingTo  extends Thread{
        Object lock = null;
        public TestThread_WaitingTo(Object lock_)
        {
            lock = lock_;
            this.setName(this.getClass().getName());
        }
        public void run()
        {
            fun();
        }
        public void fun()
        {
            synchronized(lock){  //<--打印线程堆栈时，该线程运行到这里
                fun_longtime();
            }
        }
        public void fun_longtime(){
```

```
        try{
            Thread.sleep(20000);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

运行该程序，打印堆栈如下：

```
"MyPackage.TestThread_WaitingOn" prio=6 tid=0x00a85ab8 nid=0xb04 in
Object.wait() [0x02d6f000..0x02d6fae8]
        at java.lang.Object.wait(Native Method)
        //此时wait方法会导致该锁被释放,其它线程又可以占有该锁。
        - waiting on <0x22c03c60> (a java.lang.Object)
               |                            |
               |                            +--0x22c03c60锁的类型是Object
               +--表示该线程执行到了锁0x22c03c60的wait()方法上
        at MyPackage.TestThread_WaitingOn.fun_wait(TestThread_WaitingOn.java:22)
        at MyPackage.TestThread_WaitingOn.fun(TestThread_WaitingOn.java:16)
        - locked <0x22c03c60> (a java.lang.Object)
            |
            +--locked表示该线程占有了锁0x22c03c60（即已经进入synchronized代码块中了）
        at MyPackage.TestThread_WaitingOn.run(TestThread_WaitingOn.java:11)

"MyPackage.TestThread_WaitingTo" prio=6 tid=0x00a855c0 nid=0xb08
waiting for monitor entry [0x02d2f000..0x02d2fb68]
        at MyPackage.TestThread_WaitingTo.fun(TestThread_WaitingTo.java:17)
        - waiting to lock <0x22bffb60> (a java.lang.Object)
            |
            +--waiting to lock表示锁0x22bffb60已经被其它线程占有，该线程只能等待该锁
        at MyPackage.TestThread_WaitingTo.run(TestThread_WaitingTo.java:12)

"MyPackage.TestThread_Locked" prio=6 tid=0x00a862e8 nid=0xb00
waiting on condition [0x02cef000..0x02cefbe8]
        at java.lang.Thread.sleep(Native Method)
        at MyPackage.TestThread_Locked.fun_longtime(TestThread_Locked.java:22)
        at MyPackage.TestThread_Locked.fun(TestThread_Locked.java:17)
        - locked <0x22bffb60> (a java.lang.Object)
            |
            +--该线程占有了锁0x22bffb60（已经进入synchronized代码块）
        at MyPackage.TestThread_Locked.run(TestThread_Locked.java:12)
```

从上面这个例子中，可以很清晰地看出，在线程堆栈中与锁相关的三个最重要的特征字: *locked,waiting to lock,waiting on*,了解这三个特征字，就能够对锁进行分析了。

一般情况下，当一个(些)线程在等待一个锁时，应该有一个线程占用这个锁，即如果有的线程在等待一个锁，该锁必然被另一个线程占有了，也就是说，从打印的堆栈中如果能看到waiting to lock <0x22bffb60>,应该也应该能找到一个线程locked <0x22bffb60>, 大多数情况确实如此，但在有些情况下，你会发现堆栈中可能根本就没有locked <0x22bffb60>，而只有wainting to. 这是什么原因呢？ 实际上，在一个线程释放锁和另一个线程被唤醒之间有一个时间窗，在这期间，如果恰巧进行了堆栈转储，那么就会发生上面所介绍的堆栈，只能找到一个锁的wainting to,但找不到locked该锁的线程。另外，当通过kill -3 <java pid>(unix/linux)或者<ctrl>+<break>(windows)向虚拟机进程发送信号，请求输出线程堆栈时，有的虚拟机有不同的实现策略，并不一定立即响应该请求，也许会等待正在执行的线程执行完成，然后才打印堆栈。在实际的应用中看，IBM的JDK打印出的堆栈，经常能找到一个锁的wainting to线程,但找不到locked该锁的线程；而SUN的JDK绝大多数都是配对出现的。

### §1.2.3 线程状态的解读

借助线程堆栈，可以分析很多类型的问题，CPU的消耗分析即是线程堆栈分析的一个重要内容。本节介绍如何解决线程堆栈的状态信息。

Java线程状态有如下几类：

**RUNNABLE** 从虚拟机的角度看，线程处于正在运行状态。

那么处于RUNNABLE的线程是不是一定消耗CPU呢？实际上不一定。下面的线程堆栈表示该线程正在从网络读取数据，尽管下面这个线程显示为RUNNABLE状态，但实际上网络IO,线程绝大多数时间是被挂起，只有当数据到达之后，线程才被重新唤醒。挂起发生在本地代码(Native)中，虚拟机根本不知道，不像显式调用了Java的sleep()或者wait()等方法，虚拟机能知道线程的真正状态，但对于本地代码中的挂起，虚拟机无法真正地知道线程状态，因此它一概显示为RUNNABLE。像这种socket IO操作，不会消耗大量的CPU,因为大多时间在等待，只有数据到来之后，才消耗一点点CPU.

```
Thread-39" daemon prio=1 tid=0x08646590 nid=0x666d runnable [5beb7000..5beb88b8]
  java.lang.Thread.State: RUNNABLE
    at java.net.SocketInputStream.socketRead0(Native Method)
    at java.net.SocketInputStream.read(SocketInputStream.java:129)
    at java.io.BufferedInputStream.fill(BufferedInputStream.java:183)
    at java.io.BufferedInputStream.read(BufferedInputStream.java:201)
    - locked <0x47bfb940> (a java.io.BufferedInputStream)
    at org.postgresql.PG_Stream.ReceiveChar(PG_Stream.java:141)
    at org.postgresql.core.QueryExecutor.execute(QueryExecutor.java:68)
    - locked <0x47bfb758> (a org.postgresql.PG_Stream)
    at org.postgresql.Connection.ExecSQL(Connection.java:398)
```

下面的线程正在执行纯Java代码指令，实实在在是消耗CPU的线程。

```
"Thread-444" prio=1 tid=0xa4853568 nid=0x7ade runnable [0xafcf7000..0xafcf8680]
  java.lang.Thread.State: RUNNABLE
    //实实在在再对应CPU运算指令
    at org.apache.commons.collections.ReferenceMap.getEntry(Unknown Source)
    at org.apache.commons.collections.ReferenceMap.get(Unknown Source)
    at org.hibernate.util.SoftLimitMRUCache.get(SoftLimitMRUCache.java:51)
    at org.hibernate.engine.query.QueryPlanCache.getNativeSQLQueryPlan()
    at org.hibernate.impl.AbstractSessionImpl.getNativeSQLQueryPlan()
    at org.hibernate.impl.AbstractSessionImpl.list()
    at org.hibernate.impl.SQLQueryImpl.list(SQLQueryImpl.java:164)
    at com.mogoko.struts.logic.user.LeaveMesManager.getCommentByShopId()
    at com.mogoko.struts.action.shop.ShopIndexBaseInfoAction.execute()
    ......
```

下面的线程正在进行JNI本地方法调用，具体是否消耗CPU，要看TcpRecvExt的实现，如果TcpRecvExt 是纯运算代码，那么是实实在在消耗CPU,如果TcpRecvExt()中存在挂起的代码，那么该线程尽管显示为RUNNABLE,但实际上也是不消耗CPU的。

```
"ClientReceiveThread" daemon prio=1 tid=0x99dbacf8 nid=0x7988 runnable [...]
  java.lang.Thread.State: RUNNABLE
    at com.pangu.network.icdcomm.htcpapijni.TcpRecvExt(Native Method)
    at com.pangu.network.icdcomm.IcdComm.receive(IcdComm.java:60)
    at com.msp.client.MspFactory$ClientReceiveThread.task(MspFactory.java:333)
    at com.msp.system.TaskThread.run(TaskThread.java:94)
```

**TIMED_WAITING(on object monitor)** 表示当前线程被挂起一段时间,说明该线程正在执行obj.wait(int time)方法.

下面的线程堆栈表示当前线程正处于TIMED_WAITING状态，当前正在被挂起，时长为参数中指定的时长，如obj.wait(2000)。因此该线程当前不消耗CPU。

```
"JMX server" daemon prio=6 tid=0x0ad2c800 nid=0xdec in Object.wait() [...]
  java.lang.Thread.State: TIMED_WAITING (on  object  monitor)
   at java.lang.Object.wait(Native Method)
   - waiting on <0x03129da0> (a [I)
   at com.sun.jmx.remote.internal.ServerComm$Timeout.run(ServerComm.java:150)
   - locked <0x03129da0> (a [I)
   at java.lang.Thread.run(Thread.java:620)
```

**TIMED_WAITING(sleeping)** 表示当前线程被挂起一段时间,即正在执行Thread.sleep(int time)方法.

下面的线程正处于TIMED_WAITING状态，表示当前被挂起一段时间，时长为参数中指定的时长，如Thread.sleep(100000)。因此该线程当前不消耗CPU。

```
"Comm thread" daemon prio=10 tid=0x00002aaad4107400 nid=0x649f waiting on condition
[0x000000004133b000..0x000000004133ba00]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at org.apache.hadoop.mapred.Task$1.run(Task.java:282)
        at java.lang.Thread.run(Thread.java:619)
```

**TIMED_WAITING(parking)** 当前线程被挂起一段时间,即正在执行Thread.sleep(int time)方法.

下面的线程正处于TIMED_WAITING状态，表示当前被挂起一段时间，时长为参数中指定的时长，如LockSupport.parkNanos(blocker, l10000) 。因此该线程当前不消耗CPU。

```
"RMI TCP" daemon prio=6 tid=0x0ae3b800 nid=0x958 waiting on condition [0x17eff000..0x17effa94]
```

```
java.lang.Thread.State: TIMED_WAITING (parking)
 at sun.misc.Unsafe.park(Native Method)
 - parking to wait for  <0x02f49f58> (a java.util.concurrent.SynchronousQueue$TransferStack)
 at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:179)
 at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill(SynchronousQueue.java:424)
 at java.util.concurrent.SynchronousQueue$TransferStack.transfer(SynchronousQueue.java:323)
 at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:871)
 at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:495)
 at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:693)
 at java.lang.Thread.run(Thread.java:620)
```

**WAINTING(on object monitor)** 当前线程被挂起，即正在执行obj.wait()方法(无参数的wait()方法).

下面的线程正处于WAITING状态，表示当前线程被挂起，如obj.wait()（只能通过notify()唤醒）。因此该线程当前不消耗CPU。

```
"IPC Client" daemon prio=10 tid=0x00002aaad4129800 nid=0x649d in Object.wait() [0x039000..0x039d00]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00002aaab3acad18>; (aorg.apache.hadoop.ipc.Client$Connection)
        at java.lang.Object.wait(Object.java:485)
        at org.apache.hadoop.ipc.Client$Connection.waitForWork(Client.java:234)
        - locked <0x00002aaab3acad18> (aorg.apache.hadoop.ipc.Client$Connection)
        at org.apache.hadoop.ipc.Client$Connection.run(Client.java:273)
```

☞总结:

处于TIMED_WAITING、WAINTING状态的线程一定不消耗CPU. 处于RUNNABLE的线程，要结合当前线程代码的性质判断，是否消耗CPU.

- 如果是纯Java运算代码，则消耗CPU.
- 如果是网络IO,很少消耗CPU.
- 如果是本地代码，结合本地代码的性质判断(可以通过pstack/gstack获取本地线程堆栈)，如果是纯运算代码，则消耗CPU, 如果被挂起，则不消耗CPU,如果是IO,则不怎么消耗CPU。

## §1.3 如何借助线程堆栈进行问题分析?

大的应用程序中，线程堆栈打印出来的行数特别多（依赖于线程的数量和调用层次的多少），如何从众多的信息中找到真正有价值的信息，需要一定的技巧，本节对此进行详细的介绍。

线程堆栈反映了系统在当前时间正在执行什么代码。根据这些信息就可以知道系统当前到底再做什么。看堆栈一般是从三个视角来分析：堆栈的局部信息、一次堆栈的统计信息（全局信息）、多个堆栈的对比信息。

**视角一** 从一次的堆栈信息中，我们能直接获取以下直接的信息：

- 当前每一个线程的调用层次关系（即调用上下文），即每个线程当前正在调用哪些函数。
- 当前每个线程当前的状态：持有了哪些锁？在等待哪些锁？

**视角二** 从一次的堆栈信息中，我们还可以获得下面的统计方面的信息：

- 当前锁的争用情况：
  - 是不是很多线程在等待同一个锁,如果很多线程在等待同一个锁，那么说明这个系统已经出现了性能瓶颈，并导致了锁竞争。还可能是某个线程长时间持有一个锁不释放（比如这个线程正陷入了死循环的代码或者正在请求一个资源，很长时间得不到唤醒）。
  - 是否有死锁，哪些线程形成了锁环？
- 当前大多数线程正在干什么，即正在执行什么代码？
- 当前线程总的数量。

**视角三** 从多次（即前后打印多次堆栈进行对比）的堆栈信息中，我们还可以获得下面的统计对比方面的信息：

- 一个线程是否在长期执行。如果每次打印的堆栈，某一个线程一直处于同样的调用上下文中，那么说明这个线程一直在执行这段代码，此时就要根据代码逻辑检查，这种长期执行是否是合理的？
- 某个线程是否存在长期获取不到锁的情况？线程是不是永远得不到唤醒？如果每次打印的堆栈，某一个线程一直在等待一个锁，那么就需要检查占有这个锁的线程为什么不释放锁？

打印一次堆栈，是一个切面，如果打印多次堆栈，那么就是立体的了。通过以上多个视角进行观察，线程堆栈在定位如下类型的问题上非常有帮助：

※ 线程死锁分析（视角一）

⋇ Java代码导致的CPU过高分析（视角三）[9]

⋇ 死循环分析（视角三）[10]

⋇ 资源不足分析（视角二）。

⋇ 性能瓶颈分析(视角二和视角三)[11]

线程堆栈在很多类型的问题分析上，非常有帮助，本章就一些典型的场景进行介绍，原理都是类似的。

[9]导致CPU过高还有其它的可能原因，详见第 205页第 §13.9节
[10]详见第 198页第 §13.5节
[11]使用线程堆栈分析性能瓶颈单独一章进行介绍，请参考第 39页 2章。

### §1.3.1 线程死锁分析

**线程死锁的原因** 当两个或多个线程正在等待被对方占有的锁，死锁就会发生。死锁会导致两个线程无法继续运行，被永远挂起。下图描述了两个线程死锁的场景：

在时间点0的时候，线程0占有了lock0，线程1占有了lock1。在时间点1二者又做了一些其它操作(此处略去). 在时间点2的时候，线程0企图获取lock1，由于此时lock1已经被线程1锁住，因此此时只能等待对方释放lock1。线程1同时企图获取lock0，由于此时lock0已经被线程0锁住，因此此时只能等待对方释放锁。由于这两个线程互相要等待被对方占有的锁，自己才能继续，因此这就造成了死锁。二者永远没有机会继续运行下去。

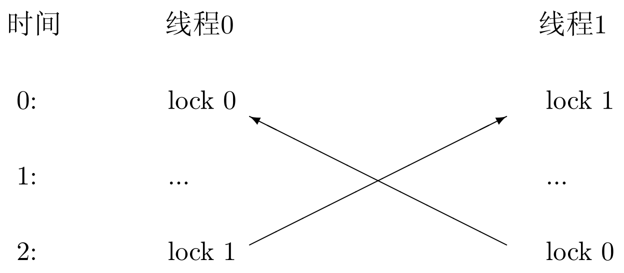



图 4 线程死锁

两个或超过两个线程因为环路的锁依赖关系而形成的锁环，就形成了真正的死锁。一个简单的死锁例子代码如下：

```
package MyPackage;
public class Main {
    public static void main(String[] args) {
        Object lockobj1 = new Object();
        Object lockobj2 = new Object();
        TestThread1 thread1 = new TestThread1(lockobj1,lockobj2);
        thread1.start();

        TestThread2 thread2 = new TestThread2(lockobj1,lockobj2);
        thread2.start();
    }
}

package MyPackage;
public class TestThread1  extends Thread{
    Object lock1 = null;
    Object lock2 = null;
    public TestThread1(Object lock1_,Object lock2_)
    {
        lock1 = lock1_;
```

```
            lock2 = lock2_;
            this.setName(this.getClass().getName());
        }
        public void run()
        {
            fun();
        }
        public void fun(){
            synchronized(lock1){
                try{
                    Thread.sleep(2);
                }
                catch(Exception e){
                    e.printStackTrace();
                }
                synchronized(lock2){
                }
            }
        }
    }

    package MyPackage;
    public class TestThread2  extends Thread{
        Object lock1 = null;
        Object lock2 = null;
        public TestThread2(Object lock1_,Object lock2_)
        {
            lock1 = lock1_;
            lock2 = lock2_;
            this.setName(this.getClass().getName());
        }
        public void run()
        {
            fun();
        }

        public void fun(){
            synchronized(lock2){
                try{
                    Thread.sleep(2);
                }
                catch(Exception e){
```

```
                    e.printStackTrace();
                }
                synchronized(lock1){
                }
            }
        }
    }
```

执行该程序，并打印堆栈，结果如下：

```
Found one Java-level deadlock:
=============================
"MyPackage.TestThread2":
  waiting to lock monitor 0x0003f04c (object 0x22bffb08, a java.lang.Object),
  which is held by "MyPackage.TestThread1"
"MyPackage.TestThread1":
  waiting to lock monitor 0x0003f06c (object 0x22bffb10, a java.lang.Object),
  which is held by "MyPackage.TestThread2"

Java stack information for the threads listed above:
===================================================
"MyPackage.TestThread2":
        at MyPackage.TestThread2.fun(TestThread2.java:25)
        - waiting to lock <0x22bffb08> (a java.lang.Object)-------+
        - locked <0x22bffb10> (a java.lang.Object)   <-------+    |
        at MyPackage.TestThread2.run(TestThread2.java:14)    |    |
"MyPackage.TestThread1":                                     |    |
        at MyPackage.TestThread1.fun(TestThread1.java:25)    |    |
        - waiting to lock <0x22bffb10> (a java.lang.Object)--+    |
        - locked <0x22bffb08> (a java.lang.Object)  <-------------+
        at MyPackage.TestThread1.run(TestThread1.java:14)
```

从打印的线程堆栈中我们能看到"Found one Java-level deadlock"，即如果存在线程死锁情况，堆栈中会直接给出死锁的分析结果。

对于上面提到的死锁，是真正的死锁。每个线程都在等待一个被对方占用的锁，结果造成了死锁。对于真正的死锁而言，虚拟机从锁的持有和请求情况就能够判断出来，因此打印堆栈时虚拟机会自动给出死锁的提示。但在实际中，很多人把系统无响应的问题统称为死锁，这种称谓实际是不恰当的。真正的死锁就是指上面所介绍的真正含义上的死锁，即是由于代码的引入的错误而导致的死锁。

当一组Java线程发生死锁的时候，那么意味着Game Over，这些线程永远得被挂在那里了，永远不能继续运行下去。当发生死锁的线程正在执行系统的关键功能时，那么这个死锁可能会导致整个系统的瘫痪，具体的严重程度取决于这些线程执行的是什么性质的功能代码，要想恢复系统，临时也是唯一的规避办法是将系统重启。然后赶快去修改导致这个死锁的Bug。

与其它并发的危险相同，死锁很少能够立即被发现，也就是说在实验室测试，能否及时发现这类问题，依赖于你的运气和你准备的测试用例的有效性。代码如果有发生死锁的潜在可能并不意味着死锁每次都发生，它只发生在该发生的时候,当死锁出现的时候，往往是遇到了最不幸的时候－在高负载的生产环境之下。

要避免死锁的问题，唯一的办法是修改代码。一个可靠的并发系统可以说是设计出来的，而不是通过改Bug改出来的，这一点与其它类型的

Bug有很大的不同[12]。另外，死锁的两个或多个线程是不消耗CPU的，有的人认为CPU 100%的使用率是线程死锁导致的，这个说法是完全错误的。无限循环（即死循环），并且在循环中代码都是CPU密集型，才有可能导致CPU的100%使用率，像socket或者数据库等IO操作是不怎么消耗CPU的。

[12]关于并发介绍，请参考第 89页第 §4.1节

### §1.3.2 Java代码死循环等导致的CPU过高分析

当系统负载大的时候,CPU的使用率会较高，但是不正确的代码也会导致CPU过高，比如死循环。当发生CPU过高的问题，我们需要能够分析CPU高的真正原因。既然CPU过高可能是死循环导致的，那么如何从线程堆栈中找到死循环的线程呢？方法是多次打印堆栈，通过前后堆栈对比找到一直在运行的线程，这些线程都是可疑的线程[13]，具体的步骤如下：

1. 通过前面介绍的堆栈获取方法获取第一次堆栈信息(详细请参考第 3页第 §1.1节)。

2. 等待一定的时间，再获取第二次堆栈信息。

3. 预处理两次堆栈信息，首先去掉处于sleeping或者waiting状态的线程，因为这种线程是不消耗CPU的。

4. 比较第一次堆栈和第二次堆栈预处理后的线程，找出这段时间一直活跃的线程，如果两次堆栈中同一个线程处于同样的调用上下文，那么就应该列为重点怀疑对象。结合代码逻辑检查该线程的执行上下文所对应的代码段是否属于应该长期运行的代码。如果不属于，那么就要仔细检查，为什么这个线程长期执行不完那段代码，这段代码是否可能存在一个死循环。

如果通过堆栈定位，没有发现热点代码段，那么CPU过高可能是不恰当的内存设置导致的频繁GC,从而导致CPU过高（请参考第 81页第 1节）。其它原因导致的CPU过高，请参考第 204页第 §13.9节。下面的线程在间隔为5分钟，分两次堆栈打印，发现该线程一直在执行同一段代码，因此怀疑这个代码段中存在死循环。其中第一次堆栈：

```
"Thread-444" prio=1 tid=0xa4853568 nid=0x7ade runnable [0xafcf7000..0xafcf8680]
    at org.apache.commons.collections.ReferenceMap.getEntry(Unknown Source)
    at org.apache.commons.collections.ReferenceMap.get(Unknown Source)
    at org.hibernate.util.SoftLimitMRUCache.get(SoftLimitMRUCache.java:51)
    at org.hibernate.engine.query.QueryPlanCache.getNativeSQLQueryPlan()
    at org.hibernate.impl.AbstractSessionImpl.getNativeSQLQueryPlan()
    at org.hibernate.impl.AbstractSessionImpl.list()
    at org.hibernate.impl.SQLQueryImpl.list(SQLQueryImpl.java:164)
    at com.mogoko.struts.logic.user.LeaveMesManager.getCommentByShopId()
    at com.mogoko.struts.action.shop.ShopIndexBaseInfoAction.execute()
    ......
```

第二次堆栈，该线程仍在那儿：

```
"Thread-444" prio=1 tid=0xa4853568 nid=0x7ade runnable [0xafcf7000..0xafcf8680]
    at org.apache.commons.collections.ReferenceMap.getEntry(Unknown Source)
    at org.apache.commons.collections.ReferenceMap.get(Unknown Source)
    at org.hibernate.util.SoftLimitMRUCache.get(SoftLimitMRUCache.java:51)
```

[13]即前面介绍的视角三，请参考第 23页第 §1.3节

```
at org.hibernate.engine.query.QueryPlanCache.getNativeSQLQueryPlan()
at org.hibernate.impl.AbstractSessionImpl.getNativeSQLQueryPlan()
at org.hibernate.impl.AbstractSessionImpl.list()
at org.hibernate.impl.SQLQueryImpl.list(SQLQueryImpl.java:164)
at com.mogoko.struts.logic.user.LeaveMesManager.getCommentByShopId()
at com.mogoko.struts.action.shop.ShopIndexBaseInfoAction.execute()
... ...
```

在长达5分钟的时间里，这个线程一直在执行org.apache.commons.collections.ReferenceMap .getEntry() 方法，说明这个函数执行一直没有结束。在有些场合下，有的函数永远不退出，这是正常的代码逻辑。这时候，具体这个函数是否属于正常还是属于Bug导致的死循环，需要结合源代码进行判断。像上面的函数，在一个Map中获取一个元素在长达几分钟的时间内还不返回，这种函数明显属于不正常情况。因此首先怀疑该函数是否存在死循环。

导致死循环的代码属于代码的Bug，这种类型的问题，重现比较难，但一旦重现问题，这类问题解决起来就比较容易。一般通过分析代码就可以发现问题。导致死循环的原因大致有如下几个：

- HashMap等线程不安全的容器，用在多线程读/写的场合，导致HashMap的方法调用形成死循环[14]。
- 多线程场合，对共享变量没有进行保护，导致数据混乱，从而使循环退出的条件永远不满足，导致死循环的发生，如
  - for,while循环中的退出条件永远不满足导致的死循环。
  - 链表等数据结构首尾相接，导致遍历永远无法停止。
- 其它错误的编码。

对于死循环导致的CPU 过高问题，通过下节介绍的方法能够一次性得到定位。下节介绍的方法要借助一些操作系统工具，这对操作系统有一定的依赖。因此实际问题的定位，可以根据情况选择合适的定位手段。

[14]将HashMap用在多线程场合下，发生死循环是很常见的现象。

### §1.3.3 高消耗CPU代码的常用分析方法

**借助操作系统提供的性能分析工具进行CPU消耗分析** 死循环可能导致CPU持续过高，对于非死循环的CPU密集型代码，也可能由于算法过于复杂，也会导致CPU过高。上面介绍的方法仅适用于死循环导致的CPU过高分析，对于非死循环导致的CPU过高，分析起来就不那么方便了，只能寻找其它更有效的定位方法。我们知道在Linux/Unix下都提供了相应的性能统计工具，通过该工具可以获得一个进程中的每一个线程所消耗的CPU比例。如在Linux下，可以通过top，在Solaris下，可以通过prstat -L <pid>获取每个线程的CPU占用的时间百分比。不同的操作系统，线程CPU统计的命令见下表：

| 操作系统 | solaris | linux | aix |
|---|---|---|---|
| 命令名称 | prstat -L <pid> | top -p <pid> | ps -emo THREAD |

该工具统计的是Java虚拟机本地线程的CPU使用情况，如果通过某种方法找到本地线程对应的Java线程，那么结合Java的线程堆栈就可以找到消耗CPU的Java代码段。实际上，在 §1.2.1节第 9页我们介绍了本地线程和Java线程的映射关系，二者是一一对应的。

具体步骤如下（假设当前的java进程id为3368）：

1. top -p 3368[15]
2. 输入'H'查看该进程所有线程的统计情况(CPU等)

```
 PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
3368 zmw2      25   0  256m 9620 6460 R 93.3  0.7   5:42.06 java
3369 zmw2      15   0  256m 9620 6460 S  0.0  0.7   0:00.00 java
3370 zmw2      15   0  256m 9620 6460 S  0.0  0.7   0:00.00 java
3371 zmw2      15   0  256m 9620 6460 S  0.0  0.7   0:00.00 java
3372 zmw2      15   0  256m 9620 6460 S  0.0  0.7   0:00.00 java
3373 zmw2      15   0  256m 9620 6460 S  0.0  0.7   0:00.00 java
3374 zmw2      15   0  256m 9620 6460 S  0.0  0.7   0:00.00 java
3375 zmw2      15   0  256m 9620 6460 S  0.0  0.7   0:00.00 java
```

通过top中的'H'命令可以获取如下详细信息：每个线程(在'H'命令下面，PID列是指线程ID,即LWPID)消耗了多少CPU。但是这个线程号如何与Java Thread Dump文件中对应起来呢[16]？很简单，在Java Thread Dump文件中，每个线程都有tid=...nid=...的属性，其中nid就是native thread id，只不过nid中用16进制来表示。例如上面的例子中3368的十六进制表示为0xd28.在Java线程中查找nid=0xd28即是本地线程对应Java线程[17]：

[15]早期版本的top不支持对线程的统计，可以使用ps命令，如：ps H -eo user,pid.ppid,tid,time,%cpu,cmd --sort=%cpu

[16]详见 §1.2.1 第 9页

[17]完整的Java堆栈请参考第 §1.2.1节第 5页

```
"main" prio=1 tid=0x0805c988 nid=0xd28 runnable [0xfff65000..0xfff659c8]
    at java.lang.String.indexOf(String.java:1352)
    at java.io.PrintStream.write(PrintStream.java:460)
    - locked <0xc8bf87d8> (a java.io.PrintStream)
    at java.io.PrintStream.print(PrintStream.java:602)
    at MyTest.fun2(MyTest.java:16)
    - locked <0xc8c1a098> (a java.lang.Object)
    at MyTest.fun1(MyTest.java:8)
    - locked <0xc8c1a090> (a java.lang.Object)
    at MyTest.main(MyTest.java:26)
```

具体导致问题的代码可能是：

1. 纯Java代码导致的CPU过高。
2. Java代码中调用的JNI代码导致的CPU过高
3. 虚拟机自身的代码导致的CPU过高，比如GC的bug等。

无论是哪个地方引起的问题，通过线程堆栈（Java线程堆栈或者本地线程堆栈）分析可以一次命中问题：

1. 通过top -p <jvm pid> [18]获取最消耗CPU的本地线程ID。
2. 通过kill -3打印Java线程堆栈。
3. 通过pstack <java pid>（有的操作系统下命令为gstack）打印本地线程堆栈。
4. 在Java线程堆栈中查找nid=<第1步获得的最耗CPU时间的线程id>。
   (a) 如果在Java线程堆栈中找到了对应的线程ID,并且该线程正在执行纯Java代码，说明是该Java代码导致的CPU过高。如：

   ```
   "Thread-444" prio=1 tid=0xa4853568 nid=0x7ade runnable [0xafcf7000..0xafcf8680]
       //当前正在执行的代码是纯Java代码
       at org.apache.commons.collections.ReferenceMap.getEntry(Unknown Source)
       at org.apache.commons.collections.ReferenceMap.get(Unknown Source)
       at org.hibernate.util.SoftLimitMRUCache.get(SoftLimitMRUCache.java:51)
       at org.hibernate.engine.query.QueryPlanCache.getNativeSQLQueryPlan()
       at org.hibernate.impl.AbstractSessionImpl.getNativeSQLQueryPlan()
       at org.hibernate.impl.AbstractSessionImpl.list()
       at org.hibernate.impl.SQLQueryImpl.list(SQLQueryImpl.java:164)
       at com.mogoko.struts.logic.user.LeaveMesManager.getCommentByShopId()
       at com.mogoko.struts.action.shop.ShopIndexBaseInfoAction.execute()
       ... ...
   ```

   (b) 如果在Java线程堆栈中找到了对应的线程ID,并且该Java线程正在执行Native code,说明导致CPU过高的问题代码在JNI调用中。如：

[18]Solaris下使用prstat -L

```
"Thread-609" prio=5 tid=0x01583d88 nid=0x280 runnable [7a680000..7a6819c0]
    //CheckLicense是Native方法，说明导致CPU过高的问题代码在本地代码中。
    at meetingmgr.conferencemgr.Operation.CheckLicense(Native method)
    at meetingmgr.MeetingAdapter.prolongMeeting(MeetingAdapter.java:171)
    at meetingmgr.timer.OnMeetingExec.execute(OnMeetingExec.java:189)
    at util.threadpool.RunnableWrapper.run(RunnableWrapper.java:131)
    at EDU.oswego.cs.dl.util.concurrent.PooledExecutor$Worker.run(...)
    at java.lang.Thread.run(Thread.java:534)
```

此时可以根据第三步获取到所有的本地线程堆栈，根据之前获得的最耗CPU时间的线程id，在本地线程堆栈中找到对应线程，即为高CPU消耗的线程。借助该本地线程堆栈信息，可以直接定位到本地代码中的死循环等问题,当然，如果是JDK的问题，只能通过JDK来解决。

```
#0  0x00000037e1e324aa in checksum ()
#1  0x00000037dcacbd66 in calculate()
... ...
#5  0x0000000000428f27 in CheckLicense ()
Thread 1 (Thread 46912546288176 (LWP 640)):
```

(c) 如果在Java线程堆栈中找不到对应的线程ID, 有如下两种可能：

i. JNI调用中重新创建的线程来执行，那么在Java线程堆栈中就不存在该线程的信息。

ii. 虚拟机自身代码导致的CPU过高，如堆内存枯竭导致的频繁FULL GC,或者虚拟机的Bug等。

此时同样可以根据第三步获取到所有的本地线程堆栈，根据之前获得的最耗CPU时间的线程id，在本地线程堆栈中找到对应线程，即为高CPU消耗的线程。借助该本地线程堆栈信息，可以直接定位到本地代码中的死循环等问题。

这种定位方式由于能够直接定位到特定的线程ID，因此基本上能够一次命中问题。是最为有效的一种方式。不管什么原因导致的CPU过高，通过这种方式都能查出来。这种方式对系统的消耗最小，非常适合在生产环境使用。

**Xrunprof协助分析** 虚拟机自身也提供了一些CPU剖析工具，借助这些工具，可以获得哪些代码段消耗了更多的CPU,借助这些信息我们也可以找到可疑的性能点。runprof的详细使用请参考第 §7.4节第 144页的介绍。

**JProfiler或者OptimizeIt等工具** 一些商业化的剖析工具，如JProfiler或者OptimizeIt等也提供了CPU剖析的能力，借助这些工具，也可以分析出消耗CPU比较多的代码段。详细请参考随机软件资料。

**多次打印堆栈** 对于耗时多的代码段，通过多次打印堆栈，该代码段在堆栈中被命中的频率相应较高，通过这种方式，也可以找出消耗CPU比较高的代码段。

上面介绍的四种方式，总的来说，第一种和第四种对系统影响最小，特别适合在生产环境下定位问题使用。而第二种和第三种通过剖析工具进行定位，由于剖析工具的极度消耗CPU,会导致整个系统的性能急剧下降，因此不太适合生产环境使用。

### §1.3.4 资源不足等导致的性能下降分析

这里所说的资源包括数据库连接等。大多时候资源不足和性能瓶颈是同一类问题。当资源不足，就会导致资源争用，请求该资源的线程会被阻塞或者挂起(wait)，自然就导致性能下降。系统对于资源，一般的设计模式是：当需要资源的时候，获取资源，当不需要的时候，就把资源释放，如果暂时没有可用资源，那么就等待（即阻塞）在那里（即等在一个资源锁上面），如果有别的线程释放资源，那么等待的线程被notify() ,等待的线程获得资源继续运行(一般资源的设计都是遵循wait/notify的模式，详见 §4.8 第 95页)。如果资源不足，那么有大量的线程在等待资源，打印的线程堆栈如果具有这个特征，那么就说明该系统资源是瓶颈。对于资源不足的导致的性能瓶颈，打印出的线程堆栈有如下特点：

- 大量的线程停在同样的调用上下文上。

如：下面的堆栈[19]大量的http-8082-Processor线程都停止在org.apache.commons.pool.impl .GenericObjectPool.borrowObject上面，说明大量的线程正在等待该资源。这就说明了该系统资源是瓶颈。

```
http-8082-Processor84" daemon prio=10 tid=0x0887c000 nid=0x5663 in Object.wait()
  java.lang.Thread.State: WAITING (on object monitor)
at java.lang.Object.wait(Object.java:485)
at org.apache.commons.pool.impl.GenericObjectPool.borrowObject(Unknown Source)
   - locked <0x75132118> (a org.apache.commons.dbcp.AbandonedObjectPool)
at org.apache.commons.dbcp.AbandonedObjectPool.borrowObject()
   - locked <0x75132118> (a org.apache.commons.dbcp.AbandonedObjectPool)
at org.apache.commons.dbcp.PoolingDataSource.getConnection()
at org.apache.commons.dbcp.BasicDataSource.getConnection(BasicDataSource.java:312)
at dbAccess.FailSafeConnectionPool.getConnection(FailSafeConnectionPool.java:162)
at servlets.ControllerServlet.doGet(ObisControllerServlet.java:93)
... ...
http-8082-Processor85" daemon prio=10 tid=0x0887c000 nid=0x5663 in Object.wait()
  java.lang.Thread.State: WAITING (on object monitor)
at java.lang.Object.wait(Object.java:485)
at org.apache.commons.pool.impl.GenericObjectPool.borrowObject(Unknown Source)
   - locked <0x75132118> (a org.apache.commons.dbcp.AbandonedObjectPool)
at org.apache.commons.dbcp.AbandonedObjectPool.borrowObject()
   - locked <0x75132118> (a org.apache.commons.dbcp.AbandonedObjectPool)
at org.apache.commons.dbcp.PoolingDataSource.getConnection()
at org.apache.commons.dbcp.BasicDataSource.getConnection(BasicDataSource.java:312)
at dbAccess.FailSafeConnectionPool.getConnection(FailSafeConnectionPool.java:162)
at servlets.ControllerServlet.doGet(ObisControllerServlet.java:93)
... ...(以下相同的调用上下文略)
```

[19]排版需要，没有将所有的线程堆栈印刷出来，这里只列了二个以作说明

导致资源不足的原因可能如下，结合堆栈就能够判断：

- 资源数量配置太少（如连接池连接配置过少等），而系统当前的压力比较大，资源不足导致了某些线程不能及时获得资源而等待在那里(即挂起)。
- 获得资源的线程把持资源时间太久，导致资源不足。如下是一种过分的的资源使用代码:

```
void fun1()
{
   Connection conn = ConnectionPool.getConnection();//获取一个数据库连接
   ... ...         //使用该数据库连接访问数据库
   ... ...         //数据库返回结果，访问完成
   ... ...         //做其它耗时操作,但这些耗时操作数据库访问无关，
   conn.close();   //释放连接回池
}
```

这段代码，在数据库访问完成后，无谓地在占用连接没释放，会导致一个线程长时间占有这个连接，而在这么长的时间里其它线程只能等待。从而导致整体性能下降。应该修改为：

```
void fun1()
{
   Connection conn = ConnectionPool.getConnection();//获取一个数据库连接
   ... ...         //使用该数据库连接访问数据库
   ... ...         //数据库返回结果，访问完成
   conn.close();   //数据库连接一旦使用完，马上释放连接回池。
   ... ...         //做其它耗时操作,但这些耗时操作数据库访问无关，
}
```

- 设计不合理导致资源占用时间过久，如SQL语句设计不恰当，或者没有索引导致的数据库访问太慢等。
- 资源用完后，在某种异常情况下，没有关闭或者回池，导致可用资源泄漏或者减少，从而导致资源竞争。(详见 §5.1第 101页)

资源不足或者资源使用不恰当，表现出来往往是一个性能问题[20]。系统越来越慢，并最终停止响应。遇到系统变慢等问题，打印堆栈是最为有效的定位方式。

[20]使用线程堆栈分析性能瓶颈单独一章进行介绍，请参考第 39页第 2章。

### §1.3.5 线程不退出导致的系统挂死分析

导致系统挂死的原因有很多，其中有一个最常见的原因是线程挂死。既然是线程挂死，那么每次打印线程堆栈，该线程必然都在同一个调用上下文上，因此定位该类型的问题原理是，通过打印多次堆栈，找出对应业务逻辑使用的线程，通过对比前后打印的堆栈确认该线程执行的代码段是否一直没有执行完成。通过打印多次堆栈，找到挂起的线程（即不退出）。步骤如下：

1. 通过前面介绍的堆栈获取方法获取第一次堆栈信息请参考第 3页第 §1.1节
2. 等待一定的时间，再获取第二次堆栈信息
3. 比较第一次堆栈和第二次线程堆栈，找出这段时间一直活跃的线程，那么就应该列为重点分析对象。

如果通过堆栈定位，没有发现不退出的线程，可能是其它原因导致系统的挂死。请参考第207页第 §13.11节.

下面的线程在间隔为5分钟，分两次打印，发现该线程一直未执行完，因此怀疑对应的代码有死循环：

```
"Thread-444" prio=1 tid=0xa4853568 nid=0x7ade runnable [0xafcf7000..0xafcf8680]
    at org.apache.commons.collections.ReferenceMap.getEntry(Unknown Source)
    at org.apache.commons.collections.ReferenceMap.get(Unknown Source)
    at org.hibernate.util.SoftLimitMRUCache.get(SoftLimitMRUCache.java:51)
    at org.hibernate.engine.query.QueryPlanCache.getNativeSQLQueryPlan()
    at org.hibernate.impl.AbstractSessionImpl.getNativeSQLQueryPlan()
    at org.hibernate.impl.AbstractSessionImpl.list()
    at org.hibernate.impl.SQLQueryImpl.list(SQLQueryImpl.java:164)
    at com.mogoko.struts.logic.user.LeaveMesManager.getCommentByShopId()
    at com.mogoko.struts.action.shop.ShopIndexBaseInfoAction.execute()
    ......
```

具体导致线程无法退出的原因有很多，如：

- 线程正在执行死循环的代码。
- 资源不足或者资源泄漏，造成当前线程阻塞在锁对象上（即wait在锁对象上），长期得不到唤醒(notify)。
- 如果当前程序和外部通信，当外部程序挂起无返回时，也会导致当前线程挂起。

总之，通过线程堆栈找到线程组塞的代码位置，很容易分析相关问题。另外，有的时候线程不是永远不结束，而是比较长的时间内不结束，这往往是一个性能问题，如长时间的锁争用，借助线程堆栈，也很容易对这种问题进行分析。

### §1.3.6 多个锁导致的锁链分析

有的时候打印出的堆栈，很多线程在等待不同的锁，有的锁竞争可能是由于另一个锁对象竞争导致，这时候要找到根源。

如下堆栈信息，等待锁0xbef17078的线程有40多个，等待0xbc7b4110有10多个。

```
"Thread-1021" prio=5 tid=0x0164eac0 nid=0x41e waiting for monitor entry[...]
    at meetingmgr.timer.OnMeetingExec.monitorExOverNotify(OnMeetingExec.java:262)
    - waiting to lock <0xbef17078> (a [B) //等待锁0xbef17078 ----------------------+
    at meetingmgr.timer.OnMeetingExec.execute(OnMeetingExec.java:189)               |
    at util.threadpool.RunnableWrapper.run(RunnableWrapper.java:131)                |
    at EDU.oswego.cs.dl.util.concurrent.PooledExecutor$Worker.run(...)              |
    at java.lang.Thread.run(Thread.java:534)                                        |
                                                                                    |
"Thread-196" prio=5 tid=0x01054830 nid=0xe1 waiting for monitor entry[...]         |
    at meetingmgr.conferencemgr.Operation.prolongResource(Operation.java:474)       |
    - waiting to lock <0xbc7b4110> (a [B)  //等待锁0xbc7b4110 ---------------------|-+
    at meetingmgr.MeetingAdapter.prolongMeeting(MeetingAdapter.java:171)            | |
    at meetingmgr.FacadeForCallBean.applyProlongMeeting(FacadeFroCallBean.java:190)| |
    at meetingmgr.timer.OnMeetingExec.monitorExOverNotify(OnMeetingExec.java:278)  | |
    - locked <0xbef17078> (a [B)           //占有锁0xbef17078 <--------------------+ |
    at meetingmgr.timer.OnMeetingExec.execute(OnMeetingExec.java:189)                 |
    at util.threadpool.RunnableWrapper.run(RunnableWrapper.java:131)                  |
    at EDU.oswego.cs.dl.util.concurrent.PooledExecutor$Worker.run(...)                |
    at java.lang.Thread.run(Thread.java:534)                                          |
                                                                                      |
"Thread-609" prio=5 tid=0x01583d88 nid=0x280 runnable [7a680000..7a6819c0]           |
    at java.net.SocketInputStream.socketRead0(Native method)                          |
    ... ...                                                                           |
    at oracle.jdbc.ttc7.Oall7.recieve(Oall7.java:369)                                 |
    ... ...                                                                           |
    at net.sf.hiberante.impl.QueryImpl.list(QueryImpl.java:39)                        |
    at meetingmgr.conferencemgr.Operation.prolongResource(Operation.java:481)         |
    - locked <0xbc7b4110> (a [B)           //占有锁0xbc7b4110 <----------------------+
    at meetingmgr.MeetingAdapter.prolongMeeting(MeetingAdapter.java:171)
    at meetingmgr.timer.OnMeetingExec.execute(OnMeetingExec.java:189)
    at util.threadpool.RunnableWrapper.run(RunnableWrapper.java:131)
    at EDU.oswego.cs.dl.util.concurrent.PooledExecutor$Worker.run(...)
    at java.lang.Thread.run(Thread.java:534)
```

1. 看到有40多个线程再等待锁0xbef17078,首先找到已经占有这把锁的线程，即"Thread-196"

2. 看到"Thread-196"占有了锁0xbef17078,但又在等待锁<0xbc7b4110>,那么此时需要再找出占有<0xbc7b4110>这个锁的线程，即"Thread-609"

3. 那么占有锁<0xbc7b4110>的线程是问题的根源，下一步就要查到底为什么这个线程长时间占有这个锁。可能的原因是持有这把锁的线程正在执行的代码性能比较低，导致锁占用时间过长。

### §1.3.7 通过线程堆栈进行性能瓶颈分析

线程堆栈对于多线程场合下的性能瓶颈定位非常有效。单独一章对性能瓶颈分析进行介绍。请参考第 45页 §2.2.2

### §1.3.8 线程堆栈不能分析什么问题?

线程堆栈定位问题，只能定位在当前线程上留下痕迹的问题，如线程死锁，线程挂死。另外，定位由于锁的设计不恰当导致的性能问题，线程堆栈也是最有效的工具，因为性能问题时时刻刻反映在当前的线程统计状况上。但线程堆栈对于不留痕迹的问题，就无能为力的。例如下列问题使用线程堆栈定位问题就没有什么帮助:

- 线程为什么跑飞的问题（请参考第 208页 §13.12节）。
- 并发的Bug导致的数据混乱,这种问题在线程堆栈中没有任何痕迹，所以这种问题线程堆栈无法提供任何帮助。
- 数据库锁表的问题(请参考第 176页 §11.1.2节),表被锁，往往是由于某个事务没有提交/回滚，但这些信息无法在堆栈中表现出来，所以堆栈分析这类问题毫无帮助。
- 其它。

总得来说，像前面提到的这种在线程上不留痕迹的问题只能通过其它手段来进行定位。在实际的系统中，系统的问题分为几种类型:

- 在堆栈中能够表现出问题的，使用线程堆栈进行定位。
- 无法在线程中留下痕迹的问题定位，需要依赖于一个好的日志设计。
- 非常隐蔽的问题，只能依赖于丰富的代码经验，如多线程导致的数据混乱、以及后面提到的幽灵代码(请参考第 101页 §5.1)。

> ☞小技巧:
>
> (new Throwable()).printStackTrace()可以在运行期打印调用该函数的线程堆栈信息，借助这个信息可以知道调用流程。

# §2 通过Java线程堆栈进行性能瓶颈分析

改善性能意味这用更少的资源做更多的事情。"资源"的概念很广泛。对于给定的活动而言，一些特定的资源通常非常缺乏，无论是CPU周期，数据库连接的数量，系统对端的处理能力等。当线程的运行因某个特定资源受阻时，我们称之为受限于该资源:受限于数据库，受限于对端的处理能力等。

为了利用并发来提高系统性能，我们需要更有效地利用现有的处理器资源，这意味着我们期望使CPU尽可能处于忙碌状态(当然，这并不是让CPU周期忙于应付无用的计算，而是让CPU做有用的事情而忙)。如果程序是受限于当前的CPU计算能力，那么我们通过增加更多的处理器或者通过集群就能提高总的性能。如果程序不能令现有的处理器处于忙碌工作的状态，那么增加处理器也无济于事。线程通过分解应用程序，总是让空闲的处理器进行未完成的工作，从而保持所有的CPU周期"热火朝天"地工作。总的来说，性能提高，需要且仅需要解决当前的受限资源，当前受限资源可能是:

**CPU** 如果当前CPU已经能够接近100%的利用率，并且代码业务逻辑无法再简化，那么说明该系统的已经达到了性能最大化，如果再想提高性能，只能增加处理器(增加更多的机器或者安装更多的CPU)。

**其它资源** 如数据库连接数量等，如果CPU利用率没有接近100%，那么通过修改代码(当然是有用代码)尽量提高CPU的使用率，那么整体性能也会获得极大提高。

关于一个系统到底需要多少线程，请参考第 93页第 §4.6节。本节着重介绍多线程场合下的性能瓶颈定位。特别是锁的使用不当，导致的性能瓶颈。对于笨拙的算法导致的性能低下不在本节讨论范围之内。如果一个系统中如下特点，说明这个系统有性能提升的空间，换句话说，如果你的系统具有如下特点，说明你这个系统存在性能瓶颈：

- 随着系统逐步增加压力，但是CPU的使用率无法趋近于100%。 如下图：

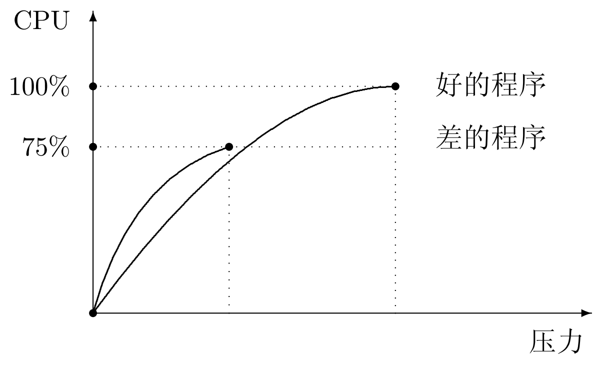



图 5 性能好和差的程序CPU利用率曲线对比

一个好的程序，应该是能够充分利用CPU。如果一个程序在单CPU的机器上无论在多大的压力下都无法令CPU的使用率接近100% ,说明这个程序设计地有问题。一个系统的性能瓶颈分析过程大致是这样的:

1. 先进行单流程的性能瓶颈分析，首先让单流程的性能达到最优.如采用更加简单的算法等。单流程的性能分析可以借助OptimizeIt或则通过增加时间戳等方式进行代码片断的分析[21]，分析哪段代码耗时比较多。这种场合下没有太多的通用技巧，只能具体问题具体分析，因此本文对此不多做介绍。

2. 进行整体性能瓶颈分析。因为单流程性能最优，不一定整个系统性能也一定很高。在多线程场合下，锁争用等也会导致性能低下，尽管单流程的性能已经达到最优了。本节着重介绍这种场合下的性能调优。

在继续介绍之前，我们先介绍一下高性能的概念。高性能在不同的应用场合下，有不同的含义：

1. 有的场合高性能意味着用户速度的体验，如界面操作，如在word中，点击一个菜单，响应很快我们就说性能很高。

2. 有的场合，高吞吐量意味着高性能，如短信或者彩信，系统更看重吞吐量，而对每一个消息的处理时间不敏感。

3. 有的场合，是二者的结合，如基于sip的软交换系统，不但要求系统有很大的吞吐量，还要求每个消息在指定的时间内完成处理，不允许有延迟。

性能调优的终极目标是：系统的CPU利用率接近100%.如果你的CPU没有被充分利用，那么有如下几个可能：

**施加的压力不足** 可能被测试的程序没有被加入足够的的压力（负载），这时候可以通过增加压力，检测系统的响应时间，服务失败率，和CPU的使用率情况。如果增加压力，系统开始出现部分服务失败，系统的响应时间变慢，或者CPU的使用率无法再上升，那么此时的压力应该是系统的饱和压力。即此时的能力是系统当前的最大能力。

**系统存在瓶颈** 当系统在饱和压力下，如果CPU的使用率没有接近100%,那么说明这个系统的性能还有提升的空间。

开发期间，请按照上面所述的方法检查系统是否存在不当编码导致性能无法最大化。在运行期间，如果系统存在如下问题，那么也可以使用本节所介绍的线程堆栈查找性能瓶颈的方法进行问题定位：

- 持续运行缓慢。时常发现应用程序运行缓慢。通过改变环境因子（如负载量、数据库连接数等）也无法有效提升整体响应时间。

- 系统性能随时间的增加逐渐下降。在负载稳定的情况下，系统运行时间越长速度越慢。可能是由于超出某个阈值范围，系统运行频繁出错从而导致系统死锁或崩溃。

[21]其它定位方法请参考第 31页第 §1.3.3节介绍。

- 系统性能随负载的增加逐渐下降。随着用户数目的增多，应用程序的运行越发缓慢。若干个用户退出系统后，应用程序便能够恢复正常运行状态。

## §2.1 常见的性能瓶颈

### 由于不恰当的同步导致的资源争用

1. 不相关的两个函数，共用了一个锁,或者不同的共享变量共用了同一个锁，无谓地制造出了资源争用。下面是多线程新手常见的一种错误：

```
class MyClass
{
    Object sharedObj;
    synchronized void fun1() {...} //访问共享变量sharedObj
    synchronized void fun2() {...} //访问共享变量sharedObj
    synchronized void fun3() {...} //不访问共享变量sharedObj
    synchronized void fun4() {...} //不访问共享变量sharedObj
    synchronized void fun5() {...} //不访问共享变量sharedObj
}
```

上面的代码将sychronized加在类的每一个方法上面，违背了保护什么锁什么的原则。对于无共享资源的两个方法，使用了同一个锁，人为造成了不必要的锁等待。Java缺省提供了this锁，这样就很多人喜欢直接在方法上使用synchronized加锁，很多情况下这样做是不恰当的，如果不考虑清楚就这样做，很容易造成锁粒度过大：

- 两个不相干的方法（即压根没有使用同一个共享变量），共用了this锁，导致人为的资源竞争.
- 即使一个方法中的代码也不是处处需要锁保护的。如果整个方法使用了synchronized的，那么很可能就把synchronized的作用域给人为扩大了。在方法级别上加锁，是一种粗犷的锁使用习惯。

上面的例子应将不访问共享变量的synchronized去掉：

```
class MyClass
{
    Object sharedObj;
    synchronized void fun1() {...} //访问共享变量sharedObj
    synchronized void fun2() {...} //访问共享变量sharedObj
    void fun3() {...}              //不访问共享变量sharedObj
    void fun4() {...}              //不访问共享变量sharedObj
    void fun5() {...}              //不访问共享变量sharedObj
}
```

> ☞ 提示：
>
> 只要访问共享变量的代码段才需要使用锁保护，而且每一个共享变量对应一个自己的锁，而不要让所有的共享变量使用同一把锁。同时，如果可能尽量避免将synchronized加在整个方法上面，而是将synchronized加在尽量少的代码上。

2. 锁的粒度过大，对共享资源访问完成后，没有将后续的代码放在synchronized同步代码块之外。这样会导致当前线程长时间无谓的占有该锁，其它争用该锁的线程只能等待，最终导致性能受到极大影响。

```
void fun1()
{
    synchronized(lock){
        ... ...   //正在访问共享资源
        ... ...   //做其它耗时操作,但这些耗时操作与共享资源无关
    }
}
```

上面的代码，会导致一个线程过长地占有锁，而在这么长的时间里其它线程只能等待，这种写法在不同的场合下有不同的提升余地：

**单CPU场合** 将耗时操作拿到同步块之外，有的情况下可以提升性能，有的场合则不能：

- 同步块中的耗时代码是CPU密集型代码（如纯CPU运算等）,不存在磁盘IO/网络IO等低CPU消耗的代码，这种情况下，由于CPU执行这段代码是100%的使用率，因此缩小同步块也不会带来任何性能上的提升。但是，同时缩小同步块也不会带来性能上的下降。
- 同步块中的耗时代码属于磁盘/网络IO等低CPU消耗的代码，当当前线程正在执行不消耗CPU的代码时，这时候CPU是空闲的，如果此时让CPU忙起来，可以带来整体性能上的提升，所在在这种场景下，将耗时操作的代码放在同步块中，肯定是可以提高整个性能的。

**多CPU场合** 将耗时操作拿到同步块之外，总是可以提升性能

- 同步块中的耗时代码是纯CPU运算,不存在磁盘IO/网络IO等可能不消耗CPU的代码，这种情况下，由于是多CPU,其它CPU也许是空闲的，因此缩小同步块可以让其它线程马上得到执行这段代码，可以带来性能的提升。
- 同步块中的耗时代码存在磁盘/网络IO等不消耗CPU的代码，当当前线程正在执行不消耗CPU的代码时，这时候总有CPU是空闲的，如果此时让CPU忙起来，可以带来整体性能上的提升，所在在这种场景下，将耗时操作的代码放在同步块中，肯定是可以提高整个性能的。

但不管如何，缩小同步范围，对系统没有任何不好的影响，大多数情况下，会带来性能的提升，所以一定要缩小同步范围。因此上面的代码应该改为：

```
void fun1()
{
    synchronized(lock)
    {
        ... ...   //正在访问共享资源
    }
    ... ...       //其它耗时操作代码拿到synchronized代码外面
}
```

> ☞ 提示：
>
> 只将访问共享资源的代码放在同步块中。以确保"快进快出"

**sleep的滥用**　sleep只适合用在等待固定时长的场合，如果轮询代码中夹杂着sleep()调用，这种设计必然是一种糟糕的设计。这种设计在某些场合下会导致严重的性能瓶颈，如果是用户交互的系统，那么用户会必然会直接感觉系统变慢。如果是后台消息处理系统，那么必然消息处理会很慢。这种设计肯定可以使用notify()和wait()来完成同样的功能，详见第 §4.8节第 95页。

**String +的滥用**。

```
String c = new String("abc") + new String("efg") + new String("12345");
```

每一次+操作都会产生一个临时对象，并伴随着数据拷贝，这个对性能是一个极大的消耗。这个写法常常成为系统的瓶颈，如果这个地方恰好是一个性能瓶颈，修改成StringBuffer之后，性能会有大幅的提升。

**不恰当的线程模型**　在多线程场合下，如果线程模型不恰当，也会使性能低下。如在网络IO的场合，我们一定要使用消息发送队列和消息接收队列来进行异步IO. 这种修改之后，性能可能会有几十倍的上升。详见第 §10.3节第 169页。

**效率低下的SQL语句或者不恰当的数据库设计**　详见第 §11.2节第 178页。

**不恰当的GC参数设置导致的性能低下**　不恰当的GC参数设置会导致严重的性能问题。详见第 §3.5.1节第 81页。

**线程数量不足**　在使用线程池的场合，如果线程池的线程配置太少，也会导致性能低下。

**内存泄漏导致的频繁GC**　内存泄漏会导致GC越来越频繁，而GC操作是CPU密集型操作，频繁GC会导致系统整体性能严重下降。

### 其它

上面介绍的几种性能瓶颈除了GC参数导致的性能瓶颈外，通过线程堆栈都可以可以找到系统的性能瓶颈，具体性能瓶颈的分析请见下一节.

## §2.2 性能瓶颈分析的手段和工具

上面提到的所有这些原因形成的性能瓶颈，都可以通过线程堆栈分析，找到根本的原因。

### §2.2.1 如何去模拟，发现性能瓶颈?

性能瓶颈的几个特征：

- 当前的性能瓶颈只有一处，只有当解决的这一处，才知道下一处。没有解决当前的性能瓶颈，下一处性能瓶颈是不会出现的。在公路上，最窄的一处决定了该道路的通车能力。只有拓宽了最窄的地方，整个的交通的通车能力才能上去，而如果直接拓宽次窄（即第二窄）的路段，整个路段的通车能力不会有任何的提升。程序中的性能瓶颈和交通道路上的性能瓶颈是类似的。只有找到真正性能瓶颈，才能使整个性能得到真正的提升。如下图：

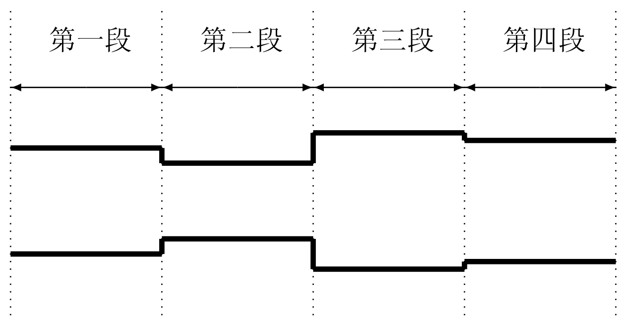



图 6 总的性能决定于最差的那一段的能力

第二段是系统最窄的地方，只有找到这个地方，将这一段拓宽，整体能力才能上去，但第一段又会成为下一个瓶颈，如此往复找到所有的性能瓶颈。

- 性能瓶颈是动态的，低负载下不是瓶颈的地方，在高负载下可能成为瓶颈。在高压力下才能出现的瓶颈，由于JProfiler等性能剖析工具依附在JVM上带来的开销，使系统根本就无法达到该瓶颈出现时需要的性能。因此这种类型的性能瓶颈在JProfiler 或者OptimizeIt等性能剖析工具下压根无法出现，也就无法找到这个性能瓶颈。在这种场合下，进行线程堆栈分析才是一个真正有效的办法。 (这也就是为什么JProfiler和OptimizeIt性能分析工具在某些情况不能真正有帮助的原因)，请参考附录 A第 225页。

鉴于性能瓶颈的以上特点，进行性能模拟的时候，一定要使用比系统当前稍高的压力下进行模拟，否则性能瓶颈不会现形。具体的步骤如下：

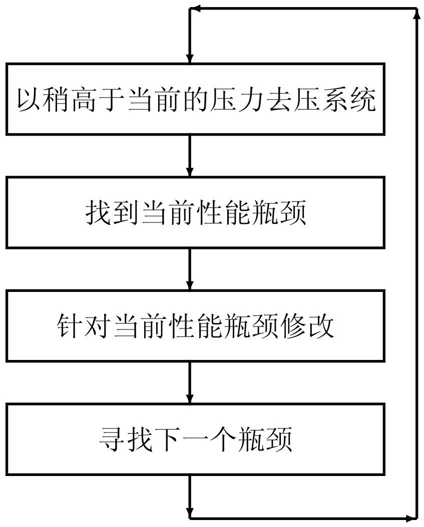



图 7 性能调优的过程

### §2.2.2 如何通过线程堆栈识别性能瓶颈？

通过线程堆栈，可以很容易地识别多线程场合下高负载的时候才出现的性能瓶颈。一旦一个系统出现性能问题，最重要的就是识别性能瓶颈，然后根据识别的性能瓶颈进行修改。一般的话，一个多线程的系统，先按照线程的功能进行归类（组），把执行相同功能代码的线程作为一组进行分析。当使用堆栈进行分析的时候，以这一组线程进行统计学分析。下面所提到的就是指执行相同代码的同一类线程的统计规律。另外，如果一个线程池为不同的功能代码服务，那么将整个线程池的线程作为一组分析即可。一般一个系统一旦出现性能瓶颈，从堆栈上分析，有如下三种最为典型的堆栈特征[22]：

1. 绝大多数线程的堆栈都表现为在同一个调用上下文上，且只剩下非常少的空闲线程。可能的原因如下：

   (a) 线程的数量过少（具体一个系统应该有多少线程请参考第 §4.6节第 93页）。

   (b) 锁的粒度过大导致的锁竞争。

   (c) 资源竞争（如数据库连接池中连接不足，导致有些企图获取连接的线程被阻塞）

   (d) 锁范围内有大量耗时操作（如大量的磁盘IO），导致锁争用。

   (e) 远程通信的对方处理缓慢（甚至导致socket缓冲区写满），如数据库侧的SQL代码性能低下。

[22]如果打印出来的堆栈不是这三种情形之一，可能被测试的程序没有被加入足够多的压力（负载），瓶颈尚未出现，你可以增加压力，直到使应用程序负荷达到饱和，此时性能瓶颈就会出现，并从堆栈里面露头。如何让系统压力达到饱和，请参考第 2节第 40页。

2. 绝大多数线程处于等待状态，只有几个工作的线程，总体性能上不去。可能的原因是，系统存在关键路径，在该关键路径上没有足够的能力给下个阶段输送大量的任务，导致其它地方空闲。如在消息分发系统，消息分发一般是一个线程，而消息处理是多个线程，这时候消息分发是瓶颈的话，那么从线程堆栈就会观察到上面提到的现象：即该关键路径没有足够的能力给下个阶段输送大量的任务，导致其它地方空闲。

3. 线程总的数量很少。导致性能瓶颈的原因与上面的类似。这里线程很少，是由于某些线程池实现使用另一种设计思路，当任务来了之后才new出线程来，这种实现方式下，线程的数量上不去，就意味有在某处关键路径上没有足够的能力给下个阶段输送大量的任务，从而不需要更多的线程来处理。关于线程池的设计思路请参考 §4.7 第 94页）。

   下面是一个出现了性能瓶颈的堆栈的例子：

```
"Thread-243" prio=1 tid=0xa58f2048 nid=0x7ac2 runnable
[0xaeedb000..0xaeedc480]
    at java.net.SocketInputStream.socketRead0(Native Method)
    at java.net.SocketInputStream.read(SocketInputStream.java:129)
    at oracle.net.ns.Packet.receive(Unknown Source)
    ... ...
    at oracle.jdbc.driver.LongRawAccessor.getBytes()
    at oracle.jdbc.driver.OracleResultSetImpl.getBytes()
    - locked <0x9350b0d8> (a oracle.jdbc.driver.OracleResultSetImpl)
    at oracle.jdbc.driver.OracleResultSet.getBytes(O)
    ... ...
    at org.hibernate.loader.hql.QueryLoader.list()
    at org.hibernate.hql.ast.QueryTranslatorImpl.list()
    ... ...
    at com.wes.NodeTimerOut.execute(NodeTimerOut.java:175)
    at com.wes.timer.TimerTaskImpl.executeAll(TimerTaskImpl.java:707)
    at com.wes.timer.TimerTaskImpl.execute(TimerTaskImpl.java:627)
    - locked <0x80df8ce8> (a com.wes.timer.TimerTaskImpl)
    at com.wes.threadpool.RunnableWrapper.run(RunnableWrapper.java:209)
    at com.wes.threadpool.PooledExecutorEx$Worker.run()
    at java.lang.Thread.run(Thread.java:595)

"Thread-248" prio=1 tid=0xa58f2048 nid=0x7ac2 runnable
[0xaeedb000..0xaeedc480]
    at java.net.SocketInputStream.socketRead0(Native Method)
    at java.net.SocketInputStream.read(SocketInputStream.java:129)
    at oracle.net.ns.Packet.receive(Unknown Source)
    ... ...
    at oracle.jdbc.driver.LongRawAccessor.getBytes()
    at oracle.jdbc.driver.OracleResultSetImpl.getBytes()
    - locked <0x9350b0d8> (a oracle.jdbc.driver.OracleResultSetImpl)
```

```
    at oracle.jdbc.driver.OracleResultSet.getBytes(O)
    ... ...
    at org.hibernate.loader.hql.QueryLoader.list()
    at org.hibernate.hql.ast.QueryTranslatorImpl.list()
    ... ...
    at com.wes.NodeTimerOut.execute(NodeTimerOut.java:175)
    at com.wes.timer.TimerTaskImpl.executeAll(TimerTaskImpl.java:707)
    at com.wes.timer.TimerTaskImpl.execute(TimerTaskImpl.java:627)
    - locked <0x80df8ce8> (a com.wes.timer.TimerTaskImpl)
    at com.wes.threadpool.RunnableWrapper.run(RunnableWrapper.java:209)
    at com.wes.threadpool.PooledExecutorEx$Worker.run()
    at java.lang.Thread.run(Thread.java:595)

... ...
"Thread-238" prio=1 tid=0xa4a84a58 nid=0x7abd in Object.wait()
[0xaec56000..0xaec57700]
    at java.lang.Object.wait(Native Method)
    at com.wes.collection.SimpleLinkedList.poll(SimpleLinkedList.java:104)
    - locked <0x6ae67be0> (a com.wes.collection.SimpleLinkedList)
    at com.wes.XADataSourceImpl.getConnection_internal(XADataSourceImpl.java:1642)
    ... ...
    at org.hibernate.impl.SessionImpl.list()
    at org.hibernate.impl.SessionImpl.find()
    at com.wes.DBSessionMediatorImpl.find()
    at com.wes.ResourceDBInteractorImpl.getCallBackObj()
    at com.wes.NodeTimerOut.execute(NodeTimerOut.java:152)
    at com.wes.timer.TimerTaskImpl.executeAll()
    at com.wes.timer.TimerTaskImpl.execute(TimerTaskImpl.java:627)
    - locked <0x80e08c00> (a com.facilities.timer.TimerTaskImpl)
    at com.wes.threadpool.RunnableWrapper.run(RunnableWrapper.java:209)
    at com.wes.threadpool.PooledExecutorEx$Worker.run()
    at java.lang.Thread.run(Thread.java:595)

"Thread-233" prio=1 tid=0xa4a84a58 nid=0x7abd in Object.wait()
[0xaec56000..0xaec57700]
    at java.lang.Object.wait(Native Method)
    at com.wes.collection.SimpleLinkedList.poll(SimpleLinkedList.java:104)
    - locked <0x6ae67be0> (a com.wes.collection.SimpleLinkedList)
    at com.wes.XADataSourceImpl.getConnection_internal(XADataSourceImpl.java:1642)
    ... ...
    at org.hibernate.impl.SessionImpl.list()
    at org.hibernate.impl.SessionImpl.find()
    at com.wes.DBSessionMediatorImpl.find()
```

```
    at com.wes.ResourceDBInteractorImpl.getCallBackObj()
    at com.wes.NodeTimerOut.execute(NodeTimerOut.java:152)
    at com.wes.timer.TimerTaskImpl.executeAll()
    at com.wes.timer.TimerTaskImpl.execute(TimerTaskImpl.java:627)
    - locked <0x80e08c00> (a com.facilities.timer.TimerTaskImpl)
    at com.wes.threadpool.RunnableWrapper.run(RunnableWrapper.java:209)
    at com.wes.threadpool.PooledExecutorEx$Worker.run()
    at java.lang.Thread.run(Thread.java:595)
... ...
```

从堆栈中看，有51个(socket)访问，其中有50个是JDBC数据库访问占用的。这说明把50个链接已经耗尽，其它所有http请求由于获取不到链接，而被阻塞在java.lang.Object.wait()方法上. 从这个堆栈中看，性能瓶颈出现在数据库访问上，数据库访问耗尽了所有的连接。找到瓶颈后，下一步结合源代码分析，具体是什么原因导致了数据库的访问需要过长的时间？ 没有创建索引，还是使用了效率过低的SQL语句？

> ☞ 提示：
>
> 一定要在一定的压力下面进行模拟，性能瓶颈才会出现。需要特别注意的是，压力工具的性能一定要高于被测试的应用程序，

### §2.2.3 其它提高性能的方法

有的时候，如果一个队列太长，单次存取时间较长，一个队列使用一个锁的话，会造成锁的激烈竞争，这个时候可以考虑把一个锁拆成多个。例如ConcrrentHashMap的实现使用了一个包含默认16个锁的Array,每一个锁都守护Hash Bucket的1/16;Bucket N由第N mod 16个锁来守护。假设hash提供了合理的拓展特性，并且关键字能够以统一的方式进行访问，这将会把对于锁的请求减少到原来的1/16,这项技术使得ConcrrentHashMap能够同时支持16个并发的Writer [23]，(为了对多处理器系统的大负荷访问提供更好的并发性，这里锁的数量还可以增加，但是只有当你有足够的证据证明并发Writer的竞争强度够大时，你可以增大所的数量超过默认的16个，合理突破这个限制，但无论如何应该是$2^n$)

分离锁的一个负面作用：对整个容器独占访问更加昂贵，通常一个操作可以通过获取最多不超过一个锁来进行，但如果有个别情况需要对整个容器加锁，如ConcurrentHashMap的值需要被扩展，重排，放入一个更大的Bucket时，就需要获取所有的内部锁[24]。这种场合下也可以考虑增加一个整个容器的锁，当需要对整个容器加锁的话，就采用这个锁。

对于像ConcurrentHashMap的实现，size是一个全局性的指标，如果把size实现为全局的，那么每一个操作可能都会访问到它，并且是同步的，这样这个地方就可能形成了"热点"，这样可能会造成严重的性能瓶颈。为了解决这种情况，ConcurrrentHashMap通过枚举每一个条目获得size,而不是维护一个全局的计数。

---

[23]对于HashMap这种纯CPU消耗操作的代码来说，分离锁在多处理器系统上能带来很大的性能提升，但在单CPU没什么价值。

[24]获得内部锁的任意集合的唯一方式是递归。

### §2.2.4 性能调优的终结条件

性能调优的过程总是有一个止点，那么满足什么条件，就说明已经没有优化的空间？总的原则是在系统中算法已经足够简化，即从算法的角度无法提升性能时，当增加压力时，CPU上升，随着压力的增加，CPU的使用率能趋向于100%，此时说明系统的性能已经榨乾，性能调优即告结束。即系统满足了如下两个条件，那么性能调优即告结束：

1. 算法足够优化
2. 没有线程/资源的使用不当而导致的CPU利用不足。

如果达到上面的条件，性能仍然无法满足应用的要求，只能通过考虑购买更好的机器，或者集群来实现更大的容量支持。

### §2.2.5 性能调优工具

目前市场上有一些性能分析工具，如JProfiler,OptimizeIt等。甚至JDK自身也提供的相应的分析工具。 但这些分析工具一旦挂到系统上之后，会导致整体性能的大幅下降，在多线程场合下，由于整体的压力无法上去，导致性能瓶颈根本就不会出现，因此这种场合下进行性能分析，这些工具基本上是没有帮助的。这些性能剖析工具比较适合于单线程下的代码段分析，找到比较耗时的算法和代码，但对于多线程场合下锁使用不当的分析，往往无能为力。

### §2.2.6 跟性能相关的JVM参数

对于由于不恰当的JVM参数设置导致的性能低下，定位方法如下：详见 §3.5.3 第 83页

## §2.3 性能分析的手段总结

通过前面几节的介绍，我们知道有多种方式可以对性能瓶颈进行分析，但每种方式往往只适合特定场合下的性能分析。没有"万能"的手段。实际情况下的性能分析，要灵活组合。

### §2.3.1 借助操作系统提供的CPU统计工具

该种方法借助于操作系统提供的一些CPU统计工具，如top, prstat等。请参考第 §1.3.3节第 31页, 该方法的特点是：

- 获取信息时，对系统几乎没有任何影响。
- 工具都是现成的，不需要搭建环境

适合分析的问题：

- 可以分析哪些代码消耗的过多的CPU

不适合分析的问题：

- 锁的粒度不合理，由于占有了锁并一定消耗CPU(如一些等待，或者IO，访问数据库等)
- 系统各种任务采用了同一个线程池，由于一个线程可能执行不同的任务，因此没有办法分离出具体的执行代码。

### §2.3.2 通过Java线程堆栈进行性能瓶颈分析

第 2节第 39页介绍的线程堆栈性能分析方法，该种方法借助于线程堆栈分析性能瓶颈。该方法的特点是：

- 获取信息时，对系统整体性能影响非常小，只有在打印堆栈的时候，会导致系统性能有少许的增加。
- 工具都是现成的，不需要搭建环境。

适合分析的问题：

- 多线程场合下，锁的粒度不合理。
- 多线程场合下，资源竞争。

不适合分析的问题：

- 非多线程型应用。

这种性能分析方法，适合于多线程场合下的性能瓶颈分析。

### §2.3.3 runhprof

第 §7.4节第 144页和第 §1.3.3节第 31页绍的runhprof性能分析方法，该种方法借助于虚拟机自带的性能剖析工具进行性能分析。请参考,该方法的特点是：

- 系统启动时，就要启动代理，对系统性能影响非常大，可能会引起几倍甚至几十倍的性能下降
- 工具都是JDK自带的，容易获得

适合分析的问题：

- 分析哪些代码块比较消耗CPU
- 资源竞争等

不适合分析的问题：

- 由于自身会带来性能的极大下降，因此在多线程场合下作整体性能分析基本没用，因为瓶颈压根不会出现。

### §2.3.4 JProfiler、JBuilder等工具

该方法的特点是：

- 系统启动时，就要启动代理，对系统性能影响非常大，可能会引起几倍甚至几十倍的性能下降
- 工具提供了更加直观的分析界面，使用起来非常地直观。

适合分析的问题：

- 非多线程下，分析哪些代码块比较消耗CPU。

不适合分析的问题：

- 由于自身会带来性能的极大下降，因此在多线程场合下做整体性能分析基本没用，因为瓶颈压根不会出现。

### §2.3.5 手工打印时间戳

通过在代码中增加System.out.println()打印时间戳，通过分析每一段代码的执行时间来分析性能瓶颈. 这种方法比较原始，只能告诉你哪些代码执行地慢，无法告诉你为什么慢，因此这种方法一般只作为一种补充手段。

# §3 Java内存泄漏分析和堆内存设置

Java语言的一个重要优点就是通过垃圾收集器(Garbage Collection，GC)自动管理内存的回收，程序员不需要通过调用函数来释放内存。因此，很多程序员认为Java不存在内存泄漏问题，或者认为即使有内存泄漏也不是程序的责任，而是GC或JVM的问题。其实，这种说法是不正确的，因为Java也存在内存泄露，但它的表现与C++不同。Java虚拟机的垃圾收集器(GC)自动回收内存垃圾，但是Java应用系统中还是有可能出现内存泄漏，Java代码写法不当，同样会造成内存泄漏。本章着重介绍Java内存泄漏的原因和定位方法。

## §3.1 Java内存泄漏的背景知识

在大型企业系统中，Java代码中的内存泄漏是常见而且比较隐蔽的问题。这些泄漏问题通常是在最不愿意它发生的正式生产环境中发现的，而且它也很难在开发与测试环境中得到重现。避免内存泄漏的第一步是要弄清楚它是如何发生的，然后对症下药。那究竟是什么导致了Java 程序中的内存泄漏呢？难道Java 虚拟机的垃圾收集器没有自动管理内存吗？

为了判断Java中是否有内存泄露，我们首先必须了解Java是如何管理内存的。Java的内存管理就是对象的分配和释放问题。在Java中，程序员需要通过关键字new为每个对象申请内存空间(基本类型除外)，所有的对象都在堆(Heap)中分配空间。另外，对象的释放是由GC决定和执行的。在Java中，内存的分配是由程序完成的，而内存的释放是有GC完成的，这种收支两条线的方法确实简化了程序员的工作。GC为了能够正确释放对象，GC必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等，GC都需要进行监控。监视对象状态是为了更加准确地、及时地释放对象，而释放对象的根本原则就是该对象不再被引用。

到此我们就明白了，Java虚拟机会对内存进行管理，但是垃圾收集的对象只能是不再被引用的对象。但是，某些在业务逻辑上已经不再需要的对象，却在系统的某个地方仍然不经意地被引用，这样垃圾回收器就不能对这些对象进行垃圾收集。

下面给出了一个简单的内存泄露的例子。在这个例子中，我们循环申请Object对象，并将所申请的对象放入一个HashMap中，如果我们仅仅释放引用本身，那么HashMap仍然引用该对象，所以这个对象对GC来说是不可回收的。因此，如果对象加入到HashMap后，还必须从HashMap中删除，这样就保证了这个对象没有再被HashMap引用到。

```
    HashMap mapobj = new HashMap();  //全局变量

    public void myfun()
    {
        String obj1 = new String("abcd");
        ... ...
        mapobj.put(obj1,obj1);
        ... ...
        obj1 = null; //此时，obj1所指向的物理内存没有被释放，
                     //因为变量mapobj仍然引用了到了这块String物理内存。
```

```
        }

```

☞ 提示:

"不再引用"和"不再需要"是两个不同的概念。"不再引用"是从虚拟机的角度看对象，而"不再需要"却是从"人"(程序员)的角度来看。不再需要的对象，在某些场合下，需要清晰地告诉虚拟机，那么对象才会不被引用到。

JVM可以自动垃圾回收，但它只能回收满足垃圾回收条件的对象。而在某些时候，需要我们自己去保证回收条件的满足。在写代码的过程中，内存泄漏往往是由于无意识造成的。系统中往往有些对象，对虚拟机来说，是被引用的，但是对应用来讲，实际上已经不再有用。这部分对象就需要我们程序员去保证垃圾回条件的满足。

Java的垃圾回收算法要做两件事情,首先它必须能检测出垃圾对象。其次它必须回收垃圾对象所使用的堆空间。垃圾对象的检测是建立在一个根对象的集合并且检查从这些根对象开始的可触及性的基础上来实现。如果根对象和某个对象之间存在引用路径，则这个对象就是可触及的，对于程序来说，根对象总是可以访问到的。从这些根对象开始，任何可以被触及的对象都被标记为"活动"对象，无法触及的对象就被认为是垃圾。

如果系统中，存在越来越多的不再影响程序未来执行的对象(即程序不再使用这些对象)，且这些对象和根对象之间存在引用路径，那么内存泄漏就产生了。即对于不再需要的java对象，由于继续被外部引用，导致虚拟机仍然认为这些对象不是垃圾，而程序却永远不会再用到它们。

内存泄漏常发生于如下场景：

- 全局的容器类（如HashMap,或者自定义的容器类等），在对象不再需要时，忘记从容器中remove，这样这个对象就会仍然被HashMap等引用到，造成这个对象不满足垃圾回收的条件，从而造成内存泄漏。特别地，在抛出异常的时候，一定要确保remove被执行到。详见第 §5.1节第 102页介绍的幽灵代码。

- 像Runnable对象等被java虚拟机自身管理的对象，没有正确的释放渠道。runnable对象必须交给一个Thread去run,否则该对象就永远不会消亡。因为像这种对象，尽管不被应用程序中的其它用户对象访问，但是这种对象会被虚拟机内部所引用。

☞ 提示:

JVM能够自动进行垃圾回收，但是要我们的代码保证无用的对象没有被继续引用。内存泄漏往往是在无意识下发生的。对于C++，程序员需要自己释放所有new出来的内存，而对于Java程序员需要确保不需要的对象一定不要被引用到。

另外，很多资料介绍了弱引用，强引用等概念，实际上这些概念都不重要，不论是弱引用，强引用，只要是一块物理内存被引用到了，这块内存必然不被当做垃圾。在深入讨论之前，我们先了解一下java对象在内存中所占的大小。

### §3.1.1 Java对象的size（32位平台）

在32位的平台上，Java对象占用的大小如下表：

表 2 JRE 1.4.2 Windows上的对象的大小

| 类型 | 尺寸(bytes) |
|---|---|
| java.lang.Object | 8 |
| java.lang.Float | 16 |
| java.lang.Double | 16 |
| java.lang.Integer | 16 |
| java.lang.Long | 16 |
| java.math.BigInteger | 56 (*) |
| java.lang.BigDecimal | 72 (*) |
| java.lang.String | 2*(Length) + 38 ±2 |
| empty java.util.Vector | 80 |
| object reference | 4 |
| float array | 4*(Length) + 14 ±2 |

- 在32位的平台上，一个对象引用占用四个字节[25]。这里要特别注意，对象引用也是一种数据类型，并且在32位平台上长度为四个字节。
- Object占用8 个字节.
- 对象的大小看起来都是8字节的倍数，这可能是基于字节对齐考虑的。对于余数不满8个字节的，自动延伸到8个字节。
- 数组的长度是数据元素的长度加14±2
- Strings的长度为内容的长度(每个字符两个字节)加上38±2 .
- 关于BigDecimal和BigInteger: 这些类型的对象大小一般是变化的。这里仅列出一个在long的范围之内的整数值。具体请参考[19]

在本书中，请特别关注对象引用是一种数据类型就够了。下面我们继续讨论，Java对象和Java对象引用。

### §3.1.2 Java对象及其引用

在许多Java书中，把对象和对象引用混为一谈。可是，如果分不清对象与对象引用，那实在没法很好地理解Java的自动垃圾回收。为便于说明，我们先定义一个简单的类：

```
class Person {
    String name;
```

[25]与C/C++中的指针含义差不多‡

```
    int    age;
}
```

有了这个类，就可以用它来创建对象：

```
Person p1 = new Person();
```

通常把这条语句的动作称之为创建一个对象，其实，它包含了四个动作。

1. 右边的"new Person"，在堆空间分配一块内存，创建一个Person类对象（也简称为Person对象）。
2. 末尾的()意味着，在对象创建后，立即调用Person类的构造函数，对刚生成的对象进行初始化。构造函数是肯定有的。如果没写，Java会给你补上一个默认的构造函数。
3. 左边的"Person p1"创建了一个Person类引用变量(在32位系统下是四个字节)。所谓Person类引用，就是以后可以用来指向Person对象的对象引用。
4. "="操作符使对象引用指向刚创建的那个Person对象。

上面的代码对应的内存分配如下：

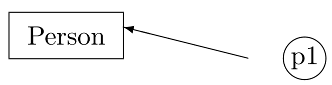



图 8 引用关系映射图(一)

从图中看出，Person p1 = new Person(); 分配了两块内存，一块是Person大小的内存（图中用矩形来标识），另一块是对象引用（四个字节，图中用圆圈来表示），该对象引用指向Person的内存。此时Person的内存被一个地方引用到。为了表达更为直观，图中使用圆圈表示对象引用，它实际上也是一块内存，长度等于机器的地址字长。

我们可以把这条语句拆成两部分：

```
Person p1;
p1 = new Person();
```

效果是一样的。这样写，就比较清楚了，有两个实体：一是对象引用变量，一是对象本身。再来一句：

```
Person p2;
```

这里又创建了一个对象引用。如果再加一句：

```
p2 = p1;
```

这里，发生了复制行为。但是，要说明的是，对象本身并没有被复制，被复制的只是对象引用。结果是，p2也指向了p1所指向的对象。两个对象引用指向了同一块内存。如下图所示：

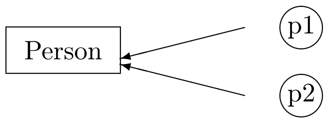



图 9 引用关系映射图(二)

此时Person对象，被p1,p2两个对象引用所引用。如果用下句再创建一个对象：

```
p2 = new Person();
```

则引用变量p2改指向第二个对象，对应的内存映射图如下：

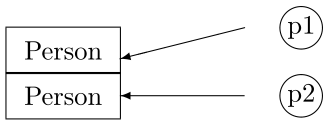



图 10 引用关系映射图(三)

从以上叙述再推演下去，我们可以获得以下结论：

1. 一个对象引用可以指向0个或1个对象（当=null的时候，指向0个对象）；
2. 一个对象可以有N个引用指向它。

如果再来下面语句：

```
p1 = p2;
```

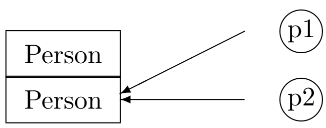



图 11 引用关系映射图(四)

按上面的推断，p1也指向了第二个对象。这个没问题。问题是第一个对象呢？没有一个地方引用它，至此它成为垃圾回收的对象。我们再看看单纯下面的语句会发生什么？

```
new Person();
```

它是合法的，而且可用的。一旦该构造函数执行完成，那么这个对象的生命周期也就结束了。这种对象我们称之为临时对象。

> ☞ 提示：
>
> 原子数据类型没有对象引用，因为所有的原子数据类型是放在一些对象里的，或者都是暂态临时对象。原子对象的复制，执行的是拷贝操作，而不是指向操作。

一个对象可能被两种类型的引用所引用，一种是单纯的对象引用，如:

```
Person p1 = new Person()
```

中的p1是一个单纯的对象引用。另一种是另一个对象中的类成员变量引用。如:

```
class MyClass{
    private int i;
    private Person person;
};
MyClass aa = new MyClass();
aa.person = new Person();
```

### §3.1.3 虚拟机自动垃圾回收机制

对象引用遍历从一组根对象开始，沿着整个对象图上的每条链接，递归确定可到达（reachable）的对象。如果某对象不能从这些根对象的一个（至少一个）到达，则将它作为垃圾收集。在对象遍历阶段，gc必须记住哪些对象可以到达，以便删除不可到达的对象，这称为标记（marking）对象。下一步，gc要删除不可到达的对象。删除时，有些gc的实现只是简单的扫描堆栈，删除未标记的对象，并释放它们的内存以生成新的对象，这叫做清除（sweeping）。这种方法的问题在于内存会分成好多小段，而它们不足以用于新的对象，但是组合起来却很大。因此，许多gc可以重新组织内存中的对象，并进行压缩（compact），形成可利用的空间。Java语言规范没有明确地说明JVM使用哪种垃圾回收算法，但是任何一种垃圾收集算法一般要做2件基本的事情：

1. 发现无用信息对象；
2. 回收被无用对象占用的内存空间。

大多数垃圾回收算法使用了根集(root set)这个概念；垃圾收集首先需要确定从根开始哪些对象是可达的和哪些是不可达的，从根集可达的对象都是活动对象，它们不能作为垃圾被回收，这也包括从根集间接可达的对象。而根集通过任意路径不可达的对象符合垃圾收集的条件，应该被回收。如下图，所有从根集不可达的对象就变成了垃圾（图中灰色框）。

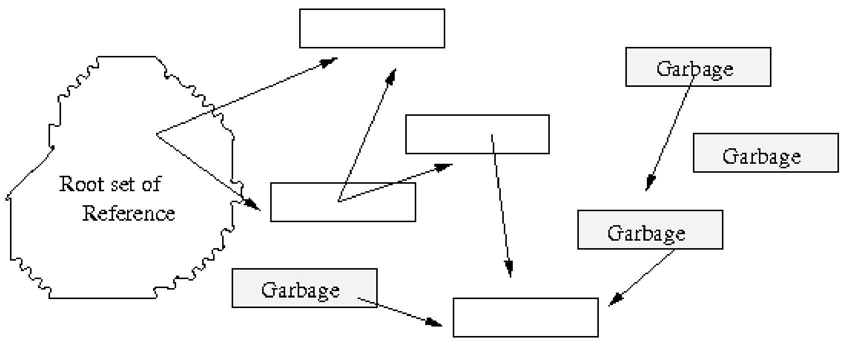



图 12 根集

Java的根集对象包括哪些？

- 没有被任何外部对象引用的栈中的对象，即系统内运行的所有线程分配在栈中的变量，该对象就是"根"，一旦线程跑到某一个变量所在的作用域之外，那么该变量就变成了垃圾，如果线程还在作用域内运行，那么该对象就是"根"；如果被其它对象引用，那么该对象不再是"根","根"变为它的上一级。具体哪个对象是"根"，依赖于垃圾回收那个时刻，每一个变量是否已经出了作用域。

- 静态变量.

在如下代码中，变量map由于是静态变量，因此在任何时候，它都是"根"对象。但对于p1对象，要看正在GC的时刻正在运行的线程是否还在p1的作用域内。p1的作用域是fun1()函数，如果线程正在fun1()函数，那么p1就是一个"根对象"，如果正在GC的时刻，正在运行的线程已经执行到fun2()，那么此时已经出了p1的作用域，此时p1就变成了垃圾。

```
class Person {
    String name;
    int    age;
}

class RunBasicThread implements Runnable{
     static HashMap map = new HashMap();

     RunBasicThread() {     }

    public void run()
    {
        fun1();
        fun2();
```

```
    }

    private void fun1(Person p)
    {
        Person p1 = new Person();
        ... ...
    }

    private void fun2()
    {
        ... ...
    }

    public static void main(String[] args) {
        RunBasicThread bt = new RunBasicThread();

        //创建一个独立的线程去执行
        new Thread(bt).start();
    }
}

```

### §3.1.4 如何告诉虚拟机不再需要这块内存?

> ☞ 提示:
>
> Java虚拟机的自动垃圾回收，回收的是已经是垃圾的对象。但具体是不是垃圾，有时需要我们的代码来告诉虚拟机，有的时候需要显式代码来通知虚拟机，在有的场景下，则不需要显式代码。

- 作用域之内的对象没有被外部对象引用，那么该作用域内new出的对象只有被本作用域内的对象引用到，不需要特别告诉虚拟机这块内存不需要。如下代码:

```
    class Person {
        String name;
        int    age;
    }

    private void fun1()
    {
       Person p1 = new Person();
```

```
        ... ...
    }
```

对象引用p1所指向的对象没有被外部其它对象引用，因此当fun1()执行完成后，局部变量p1出了作用域之后，new Person()所创建的对象就没有再被其它的地方引用到，因此该对象就真正地变成了垃圾。

- 作用域之内new出的对象被外部常态对象（长期存在的对象[26]）引用到了，此时如果不再需要该对象，需要手工写代码对该对象进行清理，虚拟机才会把它当作是垃圾对象。特别是要关注的是容器类对象，容器类往往是提供了某个函数，将一个对象放入到一个容器对象中,同时提供了另一个函数将对象移除。如HashMap.put()是向HashMap中放入一个对象（实际上就是将该HashMap指向（即引用）这个物理对象），同时HashMap.remove() 是从该HashMap中删除一个对象（即该HashMap不再引用到该物理对象），put和remove二者一定是要配对出现，一旦遗漏remove()调用，则必然造成被put的对象被引用，造成该对象无法获得回收，从而造成内存泄露。只要在函数内部new 出的对象,并且没有被外部对象引用到,都不需要关注对象生命周期.但是一旦被外部对象引用到，就要特别小心。

```
class Person {
    String name;
    int    age;
}

class MyClass{
    public static HashMap mapobj = new HashMap();  //长期存在的全局变量
    private void fun1()
    {
        Person p1 = new Person();
        mapobj.put("p1",p1);
        ... ...
    }
}
```

当正在执行fun1()函数时，Person被引用的图如下：

[26]长期存在的对象有两种：一种是静态全局变量，这种变量在整个程序生命周期都是一直有效的。另外一种当永不退出的线程中的栈对象

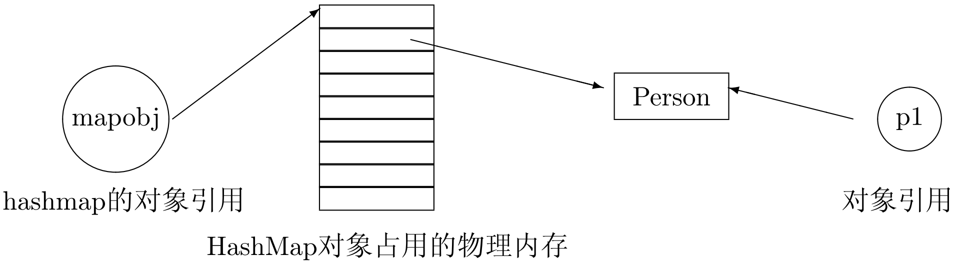



图 13 引用关系映射图(五)

当函数fun1()执行完成时，Person被引用的图如下(p1由于作用域已经结束，因此不再存在)：

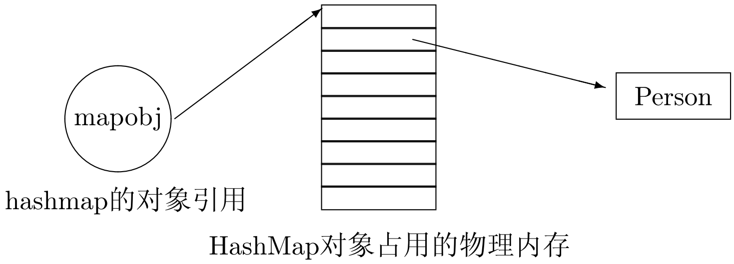



图 14 引用关系映射图(六)

对象Person仍然被mapobj所指向的HashMap引用到。而该HashMap对象是static的，生命周期是长期有效的。因此Person对象仍然不满足垃圾回收的条件，即到此位置，该对象还不是"垃圾"。如果该对象确实不再需要了，需要手工调用HashMap.remove()方法将该对象从HashMap中删除掉，之后该对象不再被HashMap引用到，该对象就真正地变成了垃圾。

- 作用域之内new出的对象被外部暂态对象引用到了，如果不再需要，不需要特别的代码告诉虚拟机这块内存不再需要。这样会形成一个孤岛，整个这个孤岛就变成了垃圾。

```
class Person {
    String name;
    int    age;
}

class MyClass{
    public HashMap mapobj = new HashMap();
```

```
        private void fun1()
        {
            Person p1 = new Person();
            mapobj.put("p1",p1);
            ... ...
        }

        private void fun2()
        {
            fun1();
            mapobj = null;   //mapobj不再指向new HashMap()所创建的对象
        }
    }
```

当执行fun1()完时，但尚未执行mapobj = null，Person被引用的图如下：

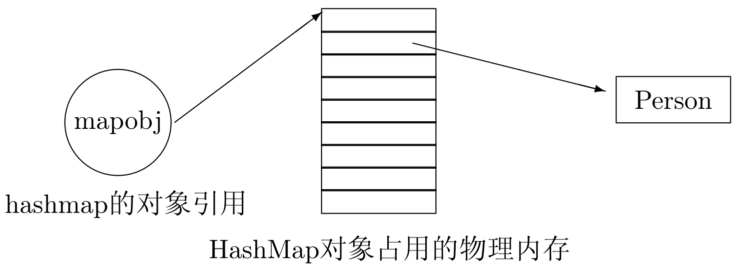



图 15 引用关系映射图(七)

当fun2()执行完成，mapobj = null;之后对应的图如下,HashMap尽管引用了Person内存,但由于HashMap对象自身已经不被任何外部对象引用到,因此HashMap对象和Person对象二者形成了孤岛，是不可达的，变成了真正的垃圾对象：

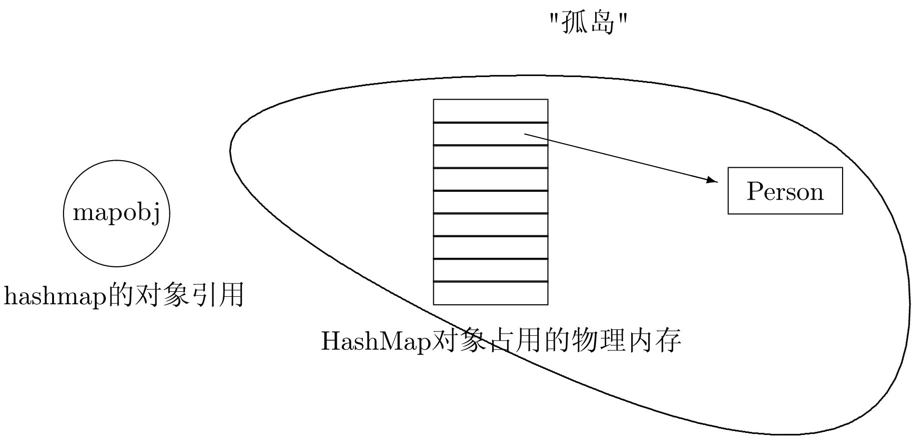



图 16 引用关系映射图(八)

上面我们介绍了Java垃圾回收的机制，在某些情况下，如果处理不得当，就会出现无用对象[27]无法回收的情况，也就是我们所说的内存泄漏，下面我们就介绍Java内存泄漏的症状和定位。

> ☞ 警告：
>
> 对集合对象（系统提供的或者自己实现的）只添加而不删除元素，在其它地方并保持了对集合对象的引用，是一种最常见的内存泄漏。

### §3.1.5 将对象设为null就可以避免内存泄漏吗？

有的Java程序员认为通过将变量设为null可以保证对象变为垃圾，从而避免内存泄漏，因此代码中充斥了大量的"ojb=null;"这种代码，如果obj是局部变量，实际上这种代码对解决内存泄漏没有任何价值，因为一旦该变量的作用域结束，该变量自然会得到垃圾回收。如果本来就存在内存泄漏，即使增加了这种代码，内存泄漏还会静静地呆在哪里。局部变量引用设为null，没有任何意义。通常不需要为了解决内存泄漏特别地将一个对象引用设为null.在解释这个问题之前，我们先介绍一下语句在运行期间会发生什么。

```
String obj1 = new String("abcd");
```

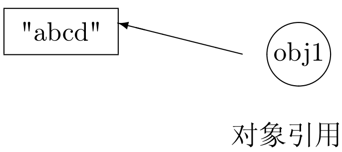



图 17 引用关系映射图(九)

[27]这里所说的无用对象是指从业务逻辑上讲，这个对象不会再被使用到，但从虚拟机来看，它根本不是垃圾对象，因为它仍然被其它对象引用到。

上面的代码，对虚拟机来说，分为了如下几个步骤：

1. 给obj1引用分配一个4个字节（32位的系统下）。即引用本身也占用4个字节。
2. 分配一块针对String类型的真正的物理内存，并使用"abcd"对该处内存进行初始化。
3. 给obj1赋值，将它指向所分配String对象的真正的物理内存地址，也就是说该处内存被obj1引用，此时String对象的内存引用计数为1。

在*String obj1 = new String("abcd");*一句代码中，最重要的是我们要清楚obj1是一个对象引用，该对象引用指向了使用new运算符分配的内存。这一行代码，对虚拟机而言，实际上是对应了两块内存分配：

- obj1对象引用的地址(在32位的系统下为四个字节，64位系统下为八个字节)。
- 使用new操作符分配的物理内存。

我们再看如下的代码到底发生了什么？

```
    HashMap mapobj = new HashMap();  //全局变量

    public void myfun()
    {
        String obj1 = new String("abcd");
        ......
        mapobj.put(obj1,obj1);
    }
```

这段代码对应的对象之间的引用映射图如下，即new String("abcd")对应是一块独立的内存，对象引用obj1指向这块内存。同时HashMap对象mapobj也指向这块内存。也就是说这块内存被两个地方引用：

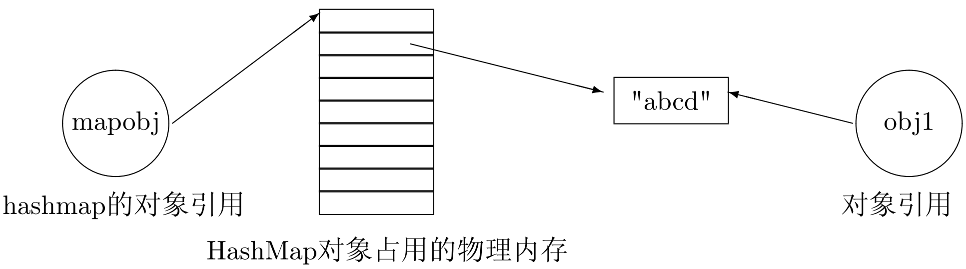



图 18 引用关系映射图(十)

如果增加一句"obj1 = null;"的语句

```
    HashMap mapobj = new HashMap();  //全局变量

    public void myfun()
    {
        String obj1 = new String("abcd");
        ... ...
        mapobj.put(obj1,obj1);
        ... ...
        obj1 = null;
    }
```

那么对应的引用图如下，obj1指向new String("abcd")对应的引用已经断开，此时这块内存只被mapobj引用，此时这块内存由于仍然被引用，因此仍然不满足垃圾回收的条件。当执行到obj1=null代码时，实际上是将obj1与矩形框对应的物理内存之间的引用断掉。obj1=null起到的惟一作用是把obj1不指向任何内存。但该矩形框锁表示的内存仍然被mapobj引用到。因此在这种情况，JVM是不会认为这是一块待回收的垃圾内存。这里的代码很简单，但在一些大系统中，代码看起来就没有这样清晰。

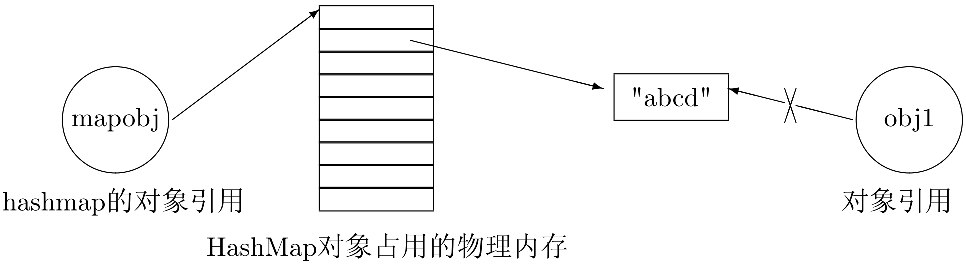



图 19 设为null的对象引用图

如果我们代码增加"mapobj.remove(obj1);"，那么内存引用关系如下，即new String("abcd")对应的物理内存只被obj1引用，当函数的作用域结束时，obj1消失，不再引用到该物理内存，此时这块物理内存就变成了垃圾。因此这里，是否有"obj1=null;"语句，对内存回收没有任何影响。

```
    HashMap mapobj = new HashMap();  //全局变量

    public void myfun()
    {
        String obj1 = new String("abcd");
        ... ...
        mapobj.put(obj1,obj1);
```

```
            ... ...
            mapobj.remove(obj1);
        }
```

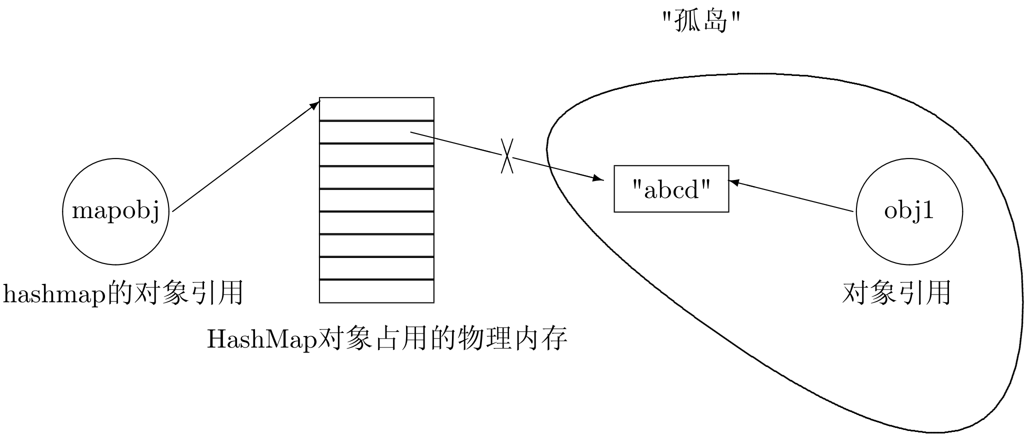



图 20 从hashmap移去对象的对象引用图

当一个全局变量指向Hashmap,当该全局变量设为null之后，那么hashmap和引用的对象就会成为一个孤岛，最终会全部释放掉。

```
        HashMap mapobj = new HashMap();  //全局变量

        public void myfun()
        {
            String obj1 = new String("abcd");
            ... ...
            mapobj.put(obj1,obj1);
            ... ...
            mapobj = null;
        }
```

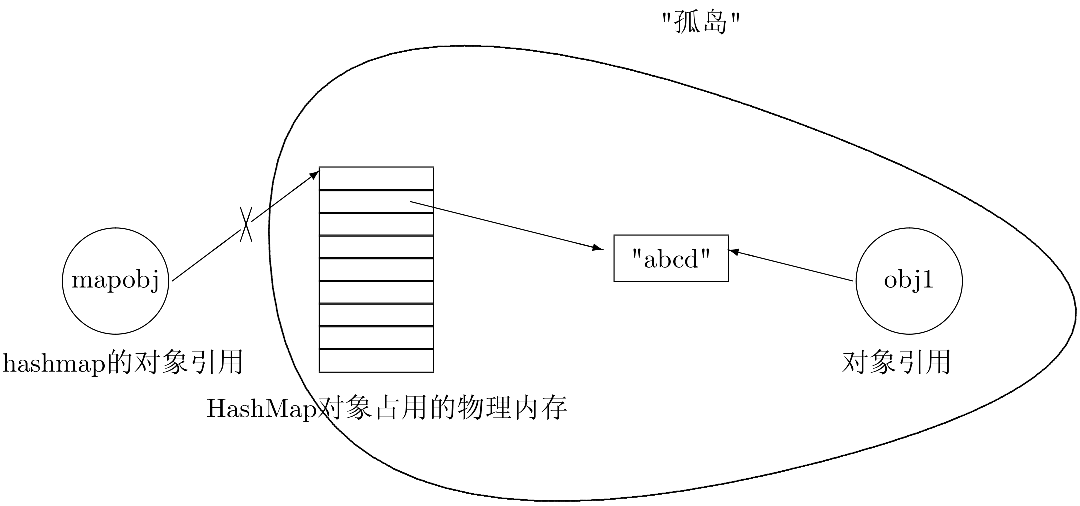



图 21 将指向hashmap对象的的引用只能置空的对象引用图

把一个引用设为null的动机往往是业务逻辑的需要，而不是释放内存的需要。如果出发点是为了释放内存，那么请问一下自己，这个是否真地需要？如下HashMap的代码片断中的tab[i] = null;是为了让tab不要指向任何对象，这个是业务逻辑需要，而不是为了垃圾回收需要.在JDK自带的代码中，我们会发现，将一个引用设为null,都是业务逻辑的需要，而不是为了内存关系的需要。将对象引用设为null，对内存泄露是没有实际价值的。

```
HashMap.java
public void clear() {
    modCount++;
    Entry[] tab = table;
    for (int i = 0; i < tab.length; i++)
        tab[i] = null;
    size = 0;
}
```

☞ 提示：

将一个对象引用设为null,是表示让该引用不指向任何其它物理内存，仅此而已。问问自己，是不是潜意识里面把对象引用当做了它所指向的对象的物理内存了？要特别注意的是，将对象引用设为'空'，而不是把对象引用所指向的物理对象设为'空'。很多时候，往往是这种概念混淆造成了理解错误。

### §3.1.6 JVM内存类型

Java进程内存, 指整个Java进程占用的内存。等于Java堆内存＋Perm内存＋本地内存：进程大小是java堆内存、Perm内存、本地内存与加载的可执行文件和库所占用内存的总和。在32位操作系统上，进程的地址空间理论上最大可达到4 GB。从这4 GB 内存中，操作系统内核为自己保留一部分内存（通常为1-2 GB）。剩余内存可用于应用程序。Windows缺省情况下，2 GB 可用于应用程序，剩余2 GB 保留供内核使用。但是，在Windows 的一些变化版本中，有一个/3GB 开关可用于改变该分配比率，使应用程序能够获得3 GB[28]。RH Linux AS 2.1-3 GB 可用于应用程序。对于其它操作系统，请参考操作系统文档了解有关配置。

1. Java堆内存,这是JVM 用来分配java 对象的内存。即通过-Xmx -Xms设置的用来分配给Java对象的内存。当执行一句java分配对象的代码：new String();那么就是从这块内存进行分配的。如果未指定最大的堆大小，那么该极限值由JVM 根据诸如计算机中的物理内存量和该时刻的可用空闲内存量这类因素来决定。始终建议您指定最大的java 堆值

2. Perm内存(PermGen space的全称是Permanent Generation space,是指内存的永久保存区域), 即通过-XX:PermSize设置的内存,这块内存是虚拟机用来加载class字节码文件的内存。这块内存在系统运行期一般比较固定。因为类文件是有限的。这里所说的一般，还有一些不一般的情况，目前在有些面向方面的编程中，会动态进行代码织入[29]操作，即系统在运行期间会修改或者增加字节码，这种情况下会导致类改变或者增加新的类，类需要重新加载，如果持续地有新类产生，可能会导致这块内存一直增加，直到这块内存溢出.

3. Java进程本地内存,这是JVM 用于其内部操作的内存。JVM 将使用的本地内存数量取决于生成的代码量、创建的线程、GC 期间用于保存java 对象信息的内存，以及在代码生成、优化等过程中使用的临时空间。如果有一个第三方本地模块，那么它也可能使用本地内存。例如，本地JDBC 驱动程序将分配本地内存。最大本地内存量受到任何特定操作系统上的虚拟进程大小限制的约束，也受到用-Xmx 标志指定用于java 堆的内存量的限制。例如，如果应用程序能分配总计为3 GB 的内存量，并且java 堆的大小设为1 GB，那么本地内存量的最大值可能在2 GB 左右。即JVM使用的本地内存由如下几部分组成：

   (a) Java.exe是C/C++写的程序，运行过程中自然需要内存,包括操作系统加载该程序，和java.exe运行过程中自己分配的内存。

   (b) JNI调用动态库使用的内存，即JNI中调用new或者malloc的内存等。

---

[28]有关/3GB 开关的详细信息,可以在以下网址中找到：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ddtools/hh/ddtools/bootini_1fcj.asp

[29]详细可以参考aspectj编译器的介绍

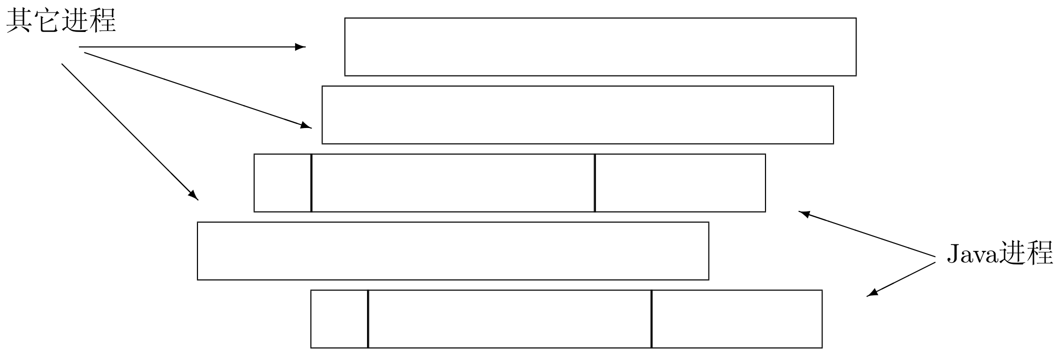



图 22 操作系统下的进程

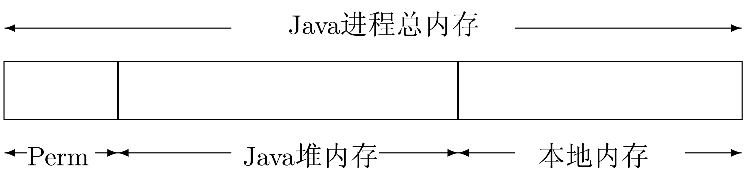



图 23 Java进程的内存占用情况

其中Java堆内存可以通过命令行指定：

**指定初使堆内存大小：** -Xms:<value>[k|m|g]

**指定最大堆内存大小：** -Xmx:<value>[k|m|g]

其中Perm内存可以通过命令行指定：

**指定初使Perm内存大小：** -XX:PermSize=<value>[k|m|g]

**指定最大Perm内存大小：** -XX:MaxPermSize=<value>[k|m|g]

☞ 提示：

Java进程内存=Java堆内存+本地内存+Perm内存，其中堆内存和Perm内存都可以通过参数设置它的大小。而本地内存的大小是不需要设置的，就像传统的程序，内存的分配直接由操作系统从总的内存中分配，需要多大就分配多大。除非大到操作系统允许的最大值。

## §3.2 Java内存泄漏的症状

### §3.2.1 为什么会发生OOM(OutOfMemroy) 问题?

第 68页的 §3.1.6节介绍了Java使用的内存种类包含三种，这三种类型的内存都可能发生内存泄漏：

- 堆内存不足,如果JVM 不能在java 堆中获得更多内存来分配更多java 对象，将会抛出java 堆内存不足(java OOM) 错误。如果java 堆充满了活动对象，并且JVM 无法再扩展java 堆，那么它将不能分配更多java 对象。

- 本地内存不足, 如果JVM 无法获得更多本地内存，它将抛出本地内存不足（本地OOM）错误。当进程用到的内存到达操作系统的最大限值，或者当计算机用完RAM 和交换空间时，通常会发生这种情况。当发生这种情况时，JVM 处于本地内存OOM状态，此时虚拟机会打印相关信息并退出。通常情况下，JVM 收到sigabort 信号时将会生成一个核心文件。

- 加载类(字节码)的Perm内存不足.即指定的Permsize不足以加载系统运行使用的.class字节码文件，就发发生Perm内存不足的错误。

导致java OOM 大多是应用程序的问题。应用程序可能在不断泄漏一些java 内存，而这可能导致出现上述问题。或者，应用程序使用更多的活动对象，因此它需要更多java 堆内存。当系统发生内存溢出，在应用程序中需要检查以下方面：

**应用程序中的缓存功能** 如果应用程序在内存中缓存java 对象，则应确保此缓存并没有不断增大。对缓存中的对象数应有一个限值。我们可以尝试减少此限值，来观察其是否降低java 堆使用量。

**大量的长期活动对象** 如果应用程序中有长期活动对象,而且占用的内存比较大，则可以尝试尽可能减少这些对象的存在期，或者通过更改设计避免这种需要大量内存的长期对象[30]。

**堆内存泄漏** 堆内存泄漏导致的OOM,内存泄漏导致的OOM定位方法请参考 71页第 §3.3节。

**本地内存泄漏** 本地内存泄漏导致的OOM,一般原因有如果几个可能:

- 如果系统中存在JNI调用，本地内存泄漏可能存在于JNI代码中。
- JDK的Bug
- 操作系统的Bug.

[30]像EJB容器中，为了避免这种可能大量存在的长期对象，采取了所谓的"钝化"技术，设置一个阈值，一旦EJB的数量超过这个阈值，则将不活跃的ejb对象从内存中持久化到数据库或者文件中，并从内存中将该ejb删除，一旦该ejb被重新调用，那么就从磁盘中重新读入构造对象。

### §3.2.2 Java内存泄漏的症状

如果Java应用程序存在内存泄漏，往往伴随着如下的现象：

1. 系统越来越慢，并伴随CPU使用率过高。这主要是因为随着内存的泄漏，可用的内存越来越小，垃圾回收器频频进行垃圾回收(完全垃圾回收(FULL GC)一次接一次，每次耗时几秒，甚至几十秒)，而垃圾回收一个CPU密集型操作，频繁的GC会导致CPU持续居高不下，在有内存泄漏的场合，到了最后必然是伴随着CPU使用率几乎为100

2. 系统运行一段时间，系统抛OutOfMemory异常，至此整个系统完全不工作

3. 虚拟机core dump

## §3.3 Java内存泄漏的定位和分析

内存泄漏的分析过程并不复杂。但往往需要很大的耐心，因为内存泄漏的分析只能是事后分析，问题重现后才可以进行分析，在某些场合下，重现问题是非常考验耐心的活动。泄漏快的地方，重现时间短，就容易进行分析。反之，时间就长。如果正常流程下的泄漏，相对来说用例容易构造，因此重现也比较容易，如果异常流程下的泄漏，由于用例往往难以"命中"，因此如果内存泄漏恰恰是异常流程导致的，那么这种情况下的内存泄漏依赖于你的耐心和你的一点点运气。结合第 70页的 §3.2.1节介绍的三种类型的内存泄漏，本节依次介绍他们的定位方法。首先介绍Java代码导致的堆内存泄漏的定位方法，然后再介绍本地内存泄漏和Perm内存（永久内存）泄漏的定位方法（即非Java代码导致的内存泄漏）。

### §3.3.1 堆内存泄漏定位

当出现*java.lang.OutOfMemoryError: Java heap space*异常,说明当前的堆内存不足，无法创建更多的Java对象。发生堆内存不足的原因有如下两种可能:

1. 设置的堆内存太小，而系统运行需要的内存要超过这个设置值。定位方法请参考 81页第 §3.5.1节
   - 使用-Xmx 参数增加虚拟机最大堆内存的大小
   - 使用-XX:MaxPermSize参数增加Perm段的最大值。

2. Java代码内存泄漏导致的内存不足，这种情况属于代码的bug, 不再使用的对象，本应该被垃圾回收器识别为垃圾而被回收掉，但由于代码的bug，这些实际上无用的对象不能被标识为垃圾，导致这种无用对象占用了过多的内存，最后内存耗尽。定位方法请参考 73页第 §3.3.1节。

3. 由于设计原因导致系统需要过多的内存，如系统中过多地缓存了数据库中的数据，这属于设计问题，需要通过设计减少内存的使用。

有的时候，内存泄漏不明显，或者怀疑系统有内存泄漏，我们可以通过下面介绍的方法初步确认系统是否存在内存泄漏。首先在Java命令行中增加-verbose:gc参数，然后重新启动java进程。当系统运行过程中,JVM进行垃圾回收的时候，会将垃圾回收的日志打印出来，通过分析这些GC日志，我们可以初步判断系统是否存在堆内存泄漏，如下GC的信息输出：

```
8190.813: [GC 164675K->251016K(1277056K), 0.0117749 secs]
8190.825: [Full GC 251016K->164654K(1277056K), 0.8142190 secs]
8191.644: [GC 164678K->251214K(1277248K), 0.0123627 secs]
8191.657: [Full GC 251214K->164661K(1277248K), 0.8135393 secs]
8192.478: [GC 164700K->251285K(1277376K), 0.0130357 secs]
8192.491: [Full GC 251285K->164670K(1277376K), 0.8118171 secs]
8193.311: [GC 164726K->251182K(1277568K), 0.0121369 secs]
8193.323 : [Full GC 251182K->164644K(1277568K), 0.8186925 secs]
8194.156: [GC 164766K->251028K(1277760K), 0.0123415 secs]
8194.169: [Full GC 251028K->164660K(1277760K), 0.8144430 secs]
```

在这段GC输出中，每一项的含义如下（更详细的GC输出信息的解读，请参考第 135页第§7.1节）：

```
8194.169: [Full GC 251028K->164660K(1277760K), 0.8144430 secs]
    |         |       |         |      |            |
    |         |       |         |      |            +-本次垃圾回收消耗的时间
    |         |       |         |     +--当前堆内存总的大小
    |         |       |        +-----垃圾回收之后, Java对象占用的内存大小
    |         |      +--------------垃圾回收之前, Java对象占用的内存大小
    |        +----------------------垃圾回收的类型(普通GC或者完全GC)
   +--------------------------------垃圾回收的时间点(秒为单位)
```

我们知道，Java虚拟机的垃圾回收有两种类型：

**普通GC** 在GC信息的输出中，[GC 164726K->251182K(1277568K), 0.0121369 secs]中的"GC"就代表的普通GC,普通GC只回收部分垃圾对象，因此回收完毕后，系统中仍存在大量的垃圾对象

**完全GC** 即FULL GC,在GC信息的输出中，[Full GC 251285K->164670K(1277376K), 0.8118171 secs]的"FULL GC"就代表的完全GC,完全GC，系统彻底的对垃圾对象进行回收，回收完毕后，垃圾对象所占用的内存得到彻底的回收，此时系统中存在的对象都是真正在使用的活动对象，这时候的Java内存真实地反映了Java对象所占用的内存的大小。

在分析系统是否存在内存泄漏时，我们关注的是在当时真正有用的对象所占用的内存的大小。如果随着系统的运行，真正的Java对象所占用的内存越来越大，那么基本上能够确认系统存在内存泄漏（此时要排除系统是否设计了大量的缓存）。因此在做内存泄漏的分析时，我们只需要分析Full GC的行(非完全垃圾回收，由于并没有将所有的垃圾都回收，因此对我们的分析没有价值)。以下面的例子为例进行说明：

[Full GC 251285K->164670K(1277376K), 0.8118171 secs]

- 251285K 完全垃圾回收之前Java对象占用的内存大小，这个值包含两部分，一部分是正在使用的Java对象占用的空间，另一部分是垃圾对象占用的空间。

- 164670K 完全垃圾回收之后Java对象占用的内存大小，这个值是真正的活动Java对象占用的内存。
- 1277376K 堆的设置最大值。
- 0.8118171 secs 表示本次完全垃圾回收占用的时间。

判断系统是否存在内存泄漏的依据是：如果系统存在内存泄漏，那么完全垃圾回收完之后的内存值应该持续上升。如果在现场能观察到这个现象，说明系统存在内存泄漏。 当怀疑一个系统存在内存泄漏的时候，首先使用FULL GC信息对内存泄漏进行一个初步确认，确认系统是否存在内存泄漏。只检查完全垃圾回收后的可用内存值是否一直再增大，步骤如下[31]：

1. 首先截取系统稳定运行以后的GC信息（如初始化已经完成），这个非常重要，非稳定运行期的信息无分析价值，因为你无法确认内存的增长是正常的增长还是由于内存泄漏导致的非正常增长。
2. 过滤出FULL GC的行。只有FULL GC的行才有分析价值。因为完全GC后的内存是当前Java对象真正使用的内存数量。一般系统会有两种可能：
   (a) 如果完全垃圾回收后的内存持续增长[32]，大有一直增长到Xmx设定值的趋势，那么这个时候基本上就可以断定系统存在内存泄漏。
   (b) 如果当前完全垃圾回收后内存增长到一个值之后，又能回落，总体上处于一个动态平衡，那么内存泄漏基本可以排除。

通过如上内存使用趋势分析之后，基本上就能确定系统是否存在堆内存泄漏。当然这种GC信息分析只能告诉你系统是否存在堆内存泄漏，但具体哪里泄漏，它是无法告诉你的。内存泄漏的的精确定位，是要找到内存泄漏的具体位置，需要借助如下工具/手段之一可以找到真正导致内存泄漏的类或者对象。

- JProfiler
- OptimizeIt
- JProbe
- JConsole
- -Xrunhprof (runhprof使用详见 §7.4 第 144页)
- JDK1.6自带工具（详见 附录 B.3.2 第 262页）
- 其它工具，如：MDD4J、BEA JRockit虚拟机自带分析工具等。

[31]存在一些GC分析工具，如gcviewer等可以对这些GC输出进行分析，结果就更加直观，不过隐藏在背后的原理和手工分析是相同的。

[32]需要特别注意的是，如果垃圾回收前的值一直增长，这个本身不是任何问题，从垃圾收集信息来看，垃圾回收之前的值总是逐渐接近设定的堆内存最大值，尽管有的时候堆内存设置的很大，这个是由于JVM启动垃圾回收的时间点来决定的，当-Xmx设置的很大，那么JVM气都启动垃圾回收的时间点也要晚一些。

这些工具从JVM获得系统内存信息的方法有如下两种：JVMTI和字节码技术(byte code instrumentation)。Java虚拟机工具接口(Java Virtual Machine Tools Interface，JVMTI)及其前身Java虚拟机监视程序接口(Java Virtual Machine Profiling Interface，JVMPI) 是外部工具与JVM通信并从JVM收集信息的标准化接口。字节码技术是指使用探测器处理字节码以获得工具所需的信息的技术。对于内存泄漏检测来说，这两种技术有两个缺点，这使它们不太适合用于生产环境。首先，它们在内存占用和性能降低方面的开销不可忽略。有关堆使用量的信息必须以某种方式从JVM导出，并收集到工具中进行处理。这意味着要为工具分配内存。信息的导出对JVM的性能影响极大。目前存在一些更好的分析工具，如JRockit Memory Leak Detector,SUN JDK1.6自带的一些命令行选项等，通过将对象数量分析内置到虚拟机中，即将内存的使用情况直接背负(piggyback)在垃圾收集器本身上而进行，相对对系统的性能影响更小。其次，只要JVM是使用-Xmanagement选项（允许通过远程JMX接口监控和管理JVM）启动的，Memory Leak Detector就可以与运行中的JVM进行连接或断开。当该工具断开时，没有任何东西遗留在JVM中，JVM又将以全速运行代码，正如工具连接之前一样。同样的，JConsole是SUN JDK自带的分析工具，挂接该工具后，对系统的影响也很小。

尽管选用的工具不同，但分析思路是相同的。这里仅介绍内存泄露精确分析的通用思路，进行内存泄漏精确定位的步骤如下(JProfiler使用详见附录 A 第 221页)：

1. 系统稳定运行一段时间，即按照业务逻辑来讲，不应该再有大的内存需求波动，这个条件非常重要

2. 点击工具条上的垃圾回收按钮，然后立即点击mark按钮，对当前对象的真实数量进行mark

3. 等待一段时间（如一个小时，或者一个晚上）

4. 点击工具条上的垃圾回收按钮，检查是否有大量的对象增加，将增加最大的那些对象挑出来，确定可疑范围。

5. 结合源代码，看是否这些可疑对象被外部引用了。

在查找哪个对象有泄漏时，最重要的是重现问题。而问题的重现依赖于有效的测试用例，也就是测试用例必须能触发内存泄漏，然后才能进行分析。内存泄漏的问题，越容易重现越容易定位。泄漏越快越容易定位。反之，定位起来就需要更长的时间，因为绝大多时间再等待问题的出现，而不是在分析问题。如果是Web程序，可以借助LoadRunner工具进行压力测试模拟，如果是纯Java应用代码，要自己编写测试用例进行模拟（在开发过程中，你是否把测试代码也作为正式代码给维护了起来？如果是，那这个工作也就容易；如果你有一套自动测试用例，那么就更容易了。）使用JProfiler等工具进行精确内存分析过程中，可以同时将-verbose:gc开关打开，观察内存的使用情况，当内存的使用接近-Xmx设置的值的$\frac{2}{3}$时候，此时挂上分析工具进行内存泄漏分析。根据经验，一旦内存的使用量接近最大的堆内存，系统非常容易core dump.此时一切的重现努力就付之东流。同时，在使用JProfiler等内存工具分析的时候，如果Java对象的内存特征不明显，找不到很明显的嫌疑对象，那么让系统多运行一段时间，嫌疑对象的特征会越来越明显，它的diff值会和正常的对象差别越来越大，直到鹤立鸡群，现形于天下。找到内存泄漏的对象之后，下一步就要结合原代码进行分析，是什么代码造成了对象的泄漏。具体的

定位思路如下：通过JProfiler找到泄漏对象的内存分配点，根据内存分配点查找代码，该对象是否需要手工进行释放？

☞ 提示：

由于JProfiler等工具一旦启动，将消耗大量的CPU和内存，因此找一台性能好的机器(更大的内存和更好的CPU)，能极大地节约时间，否则系统及其容易core dump,而使好不容易重现的努力付之东流，直到你完全丧失了信心。特别是在32位的系统上，由于JProfiler的代理也消耗大量的内存,往往操作系统对一个进程最大2G左右的的内存限制根本就不够，这时候，最好使用64位的机器进行模拟。

在JProfiler或者OptimizeIt等内存剖析工具中，发现可能存在多个类的对象存在泄漏，这种现象又两种可能：

1. 从内存分析工具中看，尽管泄漏的对象不止一个类，但可能系统只存在一处内存泄漏，即只有一个类的对象有泄漏问题，由于这个类的对象引用了其它的类的对象，而造成了被引用的对象也泄漏了，此时要在泄漏的源头进行消除，问题即可解决。

2. 系统存在不止一处内存泄漏。此时耐心地一处一处进行分析定位。

另外，上面介绍的方法适合于常态下的内存泄露，即系统一旦运行就会有缓慢的内存泄露，这种情况下通过JProfiler工具等在实验室很快就能重现。但如果在常态下无内存泄露，只有在满足特定条件下才有内存泄露的时候，由于不容易构造重现条件（或者不容易想到），因此在实验室也许根本无法重现问题，因此问题定位也就无从谈起。另外，还有的时候系统在实验室无论如何测试也没有内存泄露，但在真实环境上运行，内存泄露就会冒出来，像这种情况往往没有条件给现场挂上JProfiler等分析工具（现场不允许操作，或者由于JProfile等带来的性能急剧下降，现场无法接受）。因此在这种情况下，我们需要找到一个更有效的定位手段。JDK1.6或者更新的JDK版本提供了一系列有效的运行期选项，借助这些运行期选项，问题一旦重现，马上可以将分析信息dump出来,详细请参考 226页第 附录 B.1节）。

### §3.3.2 本地内存泄漏的定位

本地内存泄漏的症状是:从GC信息输出来看,不存在Java堆内存的泄漏,但使用内存工具(如windows下的资源管理器,unix下的prstat等), 发现Java进程的总内存越来越大，并且无停止上涨的迹象[33]，直到整个系统崩溃。如果是本地内存泄漏，定位起来相对比堆内存的问题定位要复杂一些。由于本地内存泄漏的原因有如下几个：

1. 如果系统中存在JNI调用，本地内存泄漏可能存在于JNI代码中。

2. JDK的Bug.

3. 操作系统的Bug.

[33]Java虚拟机一旦启动，通过进程的内存查看工具，你会发现在很长的时间内，Java进程的总内存一直在增长，这个本身不一定是问题，此时需要多观察一段时间，如果到了某个点增长停止，那么说明这个不是问题。如果永无停止，那么就说明系统存在内存溢出的问题。

如果JNI中存在内存泄漏，那么通过C/C++的内存泄漏分析方法可以进行定位（可以结合pmap等进行初步确认,详细请参考 161页第 §9.3.4节）。如果JNI中不存在内存泄漏，那么就更新JDK版本，通过这种排除法逐步确认问题的所在。

另外，本地内存泄漏可能还会引发如下异常：*java.lang.OutOfMemoryError: unable to create new native thread* ,即：

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to
create new native thread
        at java.lang.Thread.start0(Native Method)
        at java.lang.Thread.start(Thread.java:574)
        at TestThread.main(TestThread.java:34)
```

出现这个异常，不是由于堆内存泄漏造成的内存不足，这往往是创建的线程过多或者堆内存设置过大。我们知道，当在Java中创建一个java线程的时候，同时也创建一个操作系统的线程，而实际工作的是这个操作系统的线程。Java中的线程实际上是一个虚拟的。操作系统创建的线程对象是在本地内存上创建的(关于Java线程和Java本地线程的关系请参考第 7页，第 §1.2.1节)，如果由于某些原因，本地内存不足就会造成上面提到的这个异常。造成这个问题的原因有如下几个：

1. 系统当前有过多的线程，操作系统无法再创建更多的线程。可以通过打印线程堆栈，查看总的线程数量. 这个问题可以通过第 5页，第 §1.2节介绍的方法，通过分析线程堆栈很容易分析出来。造成这个状况的可能原因如下：

   - 系统创建的线程被阻塞或者死锁，导致系统中的线程越来越多，直到超过最大限制。
   - 操作系统自身能创建的线程数量太少。如：

     **HP安腾** 缺省情况下，一个进程创建最大的线程数为256,可用通过sam命令修改内核参数

     **2.4内核的Linux** 如Suse8,内核不支持线程，通过进程模拟线程，资源消耗非常大，一般最大只能创建300线程。最好通过升级操作系统来解决。当然通过-Xss行将每个线程堆栈的尺寸设小，也可以稍稍增大线程的数量限制[34]。

2. swap分区不足。一般情况下，交换分区要设到4G到8G比较安全。交换分区的说明请参考第 164页，第 §9.7节

3. 在32位的系统下，过大的堆内存设置，导致本地内存不足。在32位的操作系统下，每个进程理论上的最大地址空间是4G($2^{32}$).但实际上由于操作系统内存管理的原因，实际上每个进程的最大地址空间远远小于这个理论值。一般情况在2G左右(不同的操作系统这个值稍有不同，但都在2G上下)具体请参考第 81页第 §3.5.2节。。从Java程序员的角度来看，这2G左右的空间包含如下三部分：

[34]2.6内核的Linux创建的线程数量请参考第 295页，附录 H

**堆内存** Java虚拟机为Java对象（即Java代码中通过new操作符创建的对象）预留[35]的空间。这块内存通过-Xmx和-Xms指定。

**Perm内存** Java虚拟机为加载class预留的空间，这块空间通过-XX:PermSize指定。

**本地内存** Java虚拟机本身是C/C++写的，运行期间自然也需要内存，另外如果系统中有JNI调用，那么Native 代码中使用的内存也是这块。这块内存没有参数进行指定。

这三块内存中，其中Java堆内存可以通过-Xms和-Xmx设置（最小最大值），Perm内存可以通过-XX:PermSize和-XX:MaxPermSize来设置，这两块内存设置后，虚拟机会自动将给二者预留出来。如果二者相加之和过大，那么势必会挤压Native的大小（因为三者加起来不能超过整个进程总的内存极限，即上面所说的2G左右），在这种情况下，Native内存不足，系统也回抛出OutOfMemory的错误。这就是为什么在实际环境中，这两个可设的值不是设得越大越好。Xmx或者PermSize设置太大，导致Native本地内存的大小受到挤压（原因请参考第 82页，第 §3.5.2节）。解决方法如下：

(a) 减少Xmx或者PermSize的设置

(b) 如果系统需要的堆内存确实很大，无法减少Xmx的设置，可以通过设置-Xss强行将每个线程堆栈的尺寸设小，一旦线程堆栈过长，则自动截断，从而可以让线程堆栈占用的内存不过渡膨胀。但这个效果往往有限的。

但在64位的系统下，进程的地址空间要大得多，基本上不用考虑最大内存的因素，只要系统有足够大的物理内存。

### §3.3.3 Perm内存泄漏精确定位

出现*java.lang.OutOfMemoryError: PermGen space*异常，说明虚拟机的Perm内存（即永久区内存）不足。PermGen space的全称是Permanent Generationspace, 是指内存的永久保存区域[36], 这块内存主要是被JVM用作存放Class和Meta信息的,这些信息一经载入，就很少发生变化，class就属于这种类型的数据，当Class在被Loader时就会被放到PermGen space中, 它和存放类实例(Instance)的Heap区域不同,GC(Garbage Collection)不会在主程序运行期对PermGenspace进行清理，所以如果你的应用中有很多CLASS的话,就很可能出现PermGen space错误, 这种错误常见在web服务器对JSP进行precompile的时候。如果你的WEB APP下都用了大量的第三方jar, 其大小超过了jvm默认的大小(4M)那么就会产生此错误信息了。通过手动设置MaxPermSize大小可以解决这个问题，比如修改-XX:PermSize=64M为-XX:MaxPermSize=128M（但是不要设置太大，太大会挤压本地内存的空间，经过几次测试，没有问题基本上就可以了）。

Java支持动态修改类，和动态生成类，如果使用不恰当，会导致Java虚拟机装载越来越多的类，最终造成Perm内存耗尽。具体虚拟机加载了哪些类，可以通过在启动命令行中增加-verbase:class，例如：

[35]这里之所以说预留，是因为预留的主体是虚拟机，理论上说，java class自身是不会运行的，而真正执行的是虚拟机里的机器码，通俗地说，class就是脚本，指导虚拟机来进行运行。

[36]关于Java中的内存区域细节请参考 68页第 §3.1.6节

```
java -verbose:class -classpath . MyPackage.ThreadTest
[Opened C:\Program Files\Java\jdk1.5.0_12\jre\lib\rt.jar]
[Opened C:\Program Files\Java\jdk1.5.0_12\jre\lib\jsse.jar]
[Opened C:\Program Files\Java\jdk1.5.0_12\jre\lib\jce.jar]
[Opened C:\Program Files\Java\jdk1.5.0_12\jre\lib\charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.ThreadDeath from shared objects file]
[Loaded java.lang.Exception from shared objects file]
[Loaded java.lang.RuntimeException from shared objects file]
[Loaded java.security.ProtectionDomain from shared objects file]
[Loaded java.security.AccessControlContext from shared objects file]
[Loaded java.lang.ClassNotFoundException from shared objects file]
[Loaded java.lang.LinkageError from shared objects file]
```

如果不断地打印出正在加载某些类，就需要多关注一下了。

另外,有一些JDK可以禁止class做GC,比如SUN JDK的命令行选项为-noclassgc,如果启动命令行中增加了该选项，那么系统在运行期间就不会做class的GC. 这在一般情况下不会有问题，毕竟系统的class的数量有限。但在某些情况下，该选项会造成永久内存区内存溢出。

需要特别注意的是，目前随着依赖注入等技术的广泛使用，代码中可能大量地使用了反射技术。在反射的过程中，会有一些新的类被动态创建出来，如果系统中频繁地有新的类被动态创建出来，并且将禁止了class的GC，此时很容易导致永久内存区溢出。比如如果代码中有通过方法的发射机制进行方法的调用时，虚拟机就会自动创建一个新的类。

```
MyClass.class.getMethod("foo",null).invoke(null,null);
```

在反射过程，对于普通的方法，JVM会动态产生如下的类:

```
sun.reflect.GeneratedMethodAccessorN(N是一个数字)
```

如果反射的是构造函数，那么JVM会动态产生：

```
sun.reflect.GeneratedConstructorAccessorN(N是一个数字)。
```

通过在命令行中增加-verbose:class很容易观察到如下的class输出：

```
[Loaded sun.reflect.GeneratedMethodAccessor551 from __JVM_DefineClass__]
[Loaded sun.reflect.GeneratedMethodAccessor566 from __JVM_DefineClass__]
[Loaded sun.reflect.GeneratedMethodAccessor567 from __JVM_DefineClass__]
[Loaded sun.reflect.GeneratedMethodAccessor570 from __JVM_DefineClass__]
[Loaded sun.reflect.GeneratedMethodAccessor571 from __JVM_DefineClass__]
[Loaded sun.reflect.GeneratedMethodAccessor572 from __JVM_DefineClass__]
[Loaded sun.reflect.GeneratedMethodAccessor573 from __JVM_DefineClass__]
[Loaded sun.reflect.GeneratedMethodAccessor574 from __JVM_DefineClass__]
[Loaded sun.reflect.GeneratedMethodAccessor585 from __JVM_DefineClass__]
```

如果系统的代码中存在这种用法，在命令行中最好不要加-noclassgc选项，加上这个选项后，势必需要更大的永久内存，很容易造成永久内存区溢出。如果将-noclassgc从命令行中删除，那么从class的输出中可以看到这些类会被卸载掉：

```
[Unloading class sun.reflect.GeneratedMethodAccessor570]
[Unloading class sun.reflect.GeneratedMethodAccessor565]
[Unloading class sun.reflect.GeneratedMethodAccessor551]
[Unloading class sun.reflect.GeneratedMethodAccessor566]
[Unloading class sun.reflect.GeneratedMethodAccessor567]
[Unloading class sun.reflect.GeneratedMethodAccessor585]
```

### §3.3.4 真实环境下内存泄漏的定位（生僻场合下的内存泄漏定位）

如果内存泄漏的位置比较隐蔽，正常的测试用例无法覆盖，那么在实验室往往难以命中；另外，对于非常缓慢的内存泄漏，很难观察出是否存在内存泄漏，在这些情况下，只能依赖于真实环境下的重现（要想找到问题的根源，只能再让真实环境下再发生一次事故，老板也许在下一次事故发生之后马上就解雇我了，My God!）。在SUN的JDK[37]下，可以采用如下的方式来收集内存分配数据进行分析。

1. jmap -histo <java pid> > objhist.log, 可以打印出当前对象的个数和大小
2. 如果系统已经OutOfMemory异常并停止工作，可以通过jmap -heap:format=b <java pid>获取内存信息
3. 在启动期间增加-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath="具体的路径"，当系统一旦OutOfMemory之后，就会将内存信息和堆信息收集下来。

在IBM的JDK[34]下,通过设置如下环境变量，可以通过kill -3 <pid>打印堆转储(heap dump)和线程转储(thread dump):

```
export IBM_HEAP_DUMP=true
export IBM_HEAPDUMP=true
export IBM_HEAPDUMP_OUTOFMEMORY=true
export IBM_JAVACORE_OUTOFMEMORY=true
export IBM_JAVA_HEAPDUMP_TEXT=true
```

然后通过Heap Analysis工具对堆输出文件进行分析[38]。

## §3.4 Java堆内存泄漏的解决

本地内存泄漏和Perm(永久区)内存泄漏通过修改Java运行期参数，可以得到解决（请参考前面章节）。本节着重介绍堆内存泄漏的发生原因和解决方法。

我们知道，造成堆内存泄漏的原因是由于：已经无用的对象仍然被其它对象引用就造成了内存泄漏。只有从最根部清除引用才能从根本上解决内存泄漏的问题。一句话，就是要将指

[37]只有在JDK1.5之上的JDK才支持

[38]可以从http://www.alphaworks.ibm.com/tech/heapanalyzer下载该工具

向第一级无用对象的引用清除。通常，处理一处内存泄漏的问题仅需一两行代码就可搞定，找出根因是问题的关键。盲目导致忙碌，治标不如治本。常见的内存泄漏代码有如下几种：

- 全局变量（特别是容器类）引用了一个对象，在不需要的使用没有释放。特别是有的函数要成对出现，如HashMap.put()和HashMap.remove(), 如果一个把一个对象引用放入了一个全局的HashMap,在不需要的时候，没有从HashMap中remove掉，就会造成一个泄漏。

- 虽然正常情况对象进行了释放，但是在异常情况下，由于释放代码没有被执行到导致的缓慢内存泄漏。这种只有在异常情况下才会导致内存泄漏，一般比较隐蔽，在实验室往往由于难以模拟而成为漏网之鱼。如果系统在生产环境下运行出现缓慢的内存泄漏，那么这种情况可能性就比较大。如下代码就属于这种典型的场景：

```
 HashMap.put(key,myoject);
 ... ...              //其它可能抛出异常的代码
 HashMap.remove(key); //上面的异常导致这句关键代码没有被执行
```

  应该修改为如下代码，才更加健壮：

```
try{
    HashMap.put(key,myoject);
    ... ...         //其它可能抛出异常的代码
}
catch(MyException e){ //自定义的异常
    ... ...
}finally{    //通过finally,确保任何异常下，资源清理代码都会得到执行。
    ... ... //资源清理代码
    HashMap.remove(key);
}
```

  或者：

```
try{
    HashMap.put(key,myoject);
    ... ...              //其它可能抛出异常的代码
}
catch(Throwable t){
    HashMap.remove(key); //在任何异常情况下，资源清理代码都会得到执行。
}
```

- runnable类型的对象被new了，但是没有按照正常的逻辑提交给线程去执行。runnable这种特殊对象一旦new出来，会被虚拟机自身所引用，尽管用户代码中没有显式引用[39]。

[39]你可以尝试一下，new 了runnable对象，如果不提交给线程执行，你会发现该对象尽管没被显式引用，但是仍然永远不能被回收。如果提交这个runnable对象给一个线程去执行时，执行完后，虚拟机会自动回收该垃圾对象。

## §3.5 java内存和垃圾回收设置

虚拟机提供了一系列内存和垃圾回收的命令行选项，本节就相关选项进行介绍。

### §3.5.1 堆内存的设置原则

启动命令行中，我们一般会通过-Xmx和-Xms进行对堆内存进行设置。堆内存的设置很有讲究，要根据应用和环境的实际情况进行设置。如果堆内存设置太小会造成如下问题：

1. 堆内存设置太小，很多的CPU时间片被用作垃圾回收，导致频繁GC，CPU过高，浪费很多CPU.严重的时候，会导致性能有几倍或者几十倍的下降。

   ```
   8190.813: [GC 164675K->251016K(1277056K), 0.0117749 secs]
   8190.825: [Full GC 251016K->164654K(1277056K), 0.8142190 secs]
   8191.644: [GC 164678K->251214K(1277248K), 0.0123627 secs]
   8191.657: [Full GC 251214K->164661K(1277248K), 0.8135393 secs]
   8192.478: [GC 164700K->251285K(1277376K), 0.0130357 secs]
   8192.491: [Full GC 251285K->164670K(1277376K), 0.8118171 secs]
   8193.311: [GC 164726K->251182K(1277568K), 0.0121369 secs]
   8193.323 : [Full GC 251182K->164644K(1277568K), 0.8186925 secs]
   8194.156: [GC 164766K->251028K(1277760K), 0.0123415 secs]
   8194.169: [Full GC 251028K->164660K(1277760K), 0.8144430 secs]
   ```

   从GC输出中看出，大约每一秒左右发生一次完全GC(FULL GC),频率过高，说明系统可用内存不多，导致地频繁GC. 通过加大Xmx可以避免频繁FULL GC.

2. OutOfMemory，堆内存设置过小，正常的内存分配也无法满足，造成事实的内存不足。

堆内存设置太大，在32位JDK下面也会导致本地内存不足，详见第 §3.5.2节第 81页。

### §3.5.2 在32位下如何设置堆内存?

理论上，32位组成的数字最大为$2^{32}$,即4G.因此在32位的机器上，32位地址最大能指向4G的位置。也就是说，在32位操作系统下理论上的寻址空间是4G.在这种机器下，即使安装了更多的物理内存，由于地址字长的限制，系统也无法访问到。也就是说，机器即使安装了更多的物理内存，实际上最终被使用到的内存理论上最大也就4G.其它的内存都是无用的。因此我们说32位的系统下，一个进程最大能使用的内存理论上是4G。

这里往往有一个思维直觉误区，如果每个进程可以使用4G内存，那么是不是2个进程就可以使用8G内存，3个进程就可以使用12G内存了，这种直觉是错误的。由于字长的限制，系统在任何时候都无法访问到4G之外的内存，这里跟进程的个数没有关系。如果进程多了，那么所有这些进程，加起来只能使用到这4G内存。

这一点从汇编语言的角度更容易理解一些。以汇编语言的直接寻址为例：

```
mov  ax [直接地址]
[直接地址]是32位长的一个数字，它最大只能指向4G的位置。
```

注意，上面所说的是"理论上"，实际上由于操作系统内存管理的实现，对进程来讲，并不是可以真得占用到4G内存。一般情况在2G左右。超出了这个限制，会导致内存分配失败。各种操作系统下进程的最大地址空间如下[40]：

| 操作系统 | 进程最大地址空间 |
|---|---|
| Redhat Linux 32 bit | 2 GB |
| Redhat Linux 64 bit | 3 GB |
| Windows 98/2000/NT/Me/XP | 2 GB |
| Solaris x86 (32 bit) | 4 GB |
| Solaris 32 bit | 4 GB |
| Solaris 64 bit | Terabytes |

那么64位的操作系统安装32位的应用程序，情况是怎么样的？答案是以最小的为准。32位的应用程序由于自身的字长限制，最大只能使用4G内存。

在64位的环境上，堆内存的大小几乎没有限制，只要机器有足够大的物理内存。但在32位的环境上[41]，限制如下：

**正常的windows** 最大可以设置大约1.5G的堆内存

**使用/3G启动的windows** 最大可设置大约2.8G的堆内存

**有大内存支持的Linux** 最大可设置大约2.8G的堆内存

☞ 提示：

英特尔的CPU支持物理地址扩展(PAE),32位的操作系统可以访问4G之外的物理地址。但这个同时需要操作系统的支持。SuseLinux提供了一个内核编译选项，重新编译内核，可以让32位的操作系统支持超过4G的内存寻址。

**32的环境下堆内存应该如何设置设置多大？** 既然进程总的内存大小有上限，那么如果把堆内存设置很大，必然导致本地内存的空间受挤压。所以在32位的系统下面，由于支持的最大内存有限，因此设置Xmx要非常小心。如果设置的过大，系统一样会抛出OutOfMemory异常，不过这次内存不足是由于本地内存不足造成的。如何找到本地内存和堆内存边界的黄金分割点，需要通过测试来完成，这里没有黄金玉律。有的应用需要本地内存大一些，有些应用需要的本地内存小一些。根据经验，32位下面堆内存的大小一般不要超过1.2G.

[40]该表摘自[33]
[41]CPU,操作系统，JVM只要有一个是32位的，那么就受到32位的限制

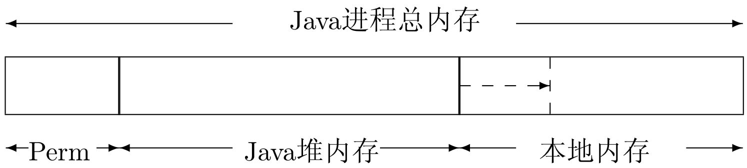



图 24 堆内存过大会直接挤压本地内存的大小

同样的道理,Perm内存设置也不要过大，Perm内存的使用量在一个系统中一般比较固定，设大了实际上是对内存的一种浪费。如果该内存设置过大，也会侵占本地内存的最大范围，如果本地内存空间太小，也会导致本地内存溢出（参考第 76页，第 §3.3.2节）。

**64位的环境下堆内存应该如何设置设置多大？** 64位的机器上，理论上可以支持的内存为$2^{64}$,即4G*4G,这个值是非常非常大了,大到我们可以认为是无限的。因此在64位的机器上，堆内存的大小设置基本上依赖于你到底装了多少的物理内存。但是不要超过物理内存的大小。

**内存的大小堆性能的影响有多大？** 我们要记住的是内存跟计算能力匹配是非常关键的，在JVM这种自己管理内存的系统尤其如此。曾经有这样一个案例，在8 个CPU 的AIX 机器，配4GB 内存200 CAPS不到就开始出问题，配16GB 的时候竟然能支持1000 CAPS。有足够的内存将导致系统内存使用的和回收的效率更高，所以直接影响系统的吞吐量。

### §3.5.3 特殊场合下JVM参数调优

原理：观察垃圾回收情况，对Xmx进行调整，是JVM的垃圾回收更加平滑和高效率。方法：在Java命令行中增加-verbose:gc 参数。分析方法：如果每次回收完成后，可用的内存持续减少，则说明可能存在内存泄漏。

设置最大的堆尺寸。在实时的场合，一般建议将-Xms和-Xmx设得一样大，这样在系统启动期间可以将整个内存给预留下来，避免内存增长过程巨大开销带来得性能暂时受影响，如sip电话场合。

### §3.5.4 Java 完全垃圾回收

完全垃圾回收(FULL GC),是指虚拟机进行一次彻底的垃圾回收。由于完全垃圾回收计算量非常大，是高耗CPU的操作，当堆内存设置很大时，由于垃圾对象非常多，每次对象扫描用的时间非常可观，常常需要几秒钟的时间才能完成。在极端的情况下，甚至需要几十秒。如果采用串行垃圾回收，那么这个系统在完全垃圾回收期间，java代码时不运行的。在实时场合，这个时间会造成很严重的问题。因此在系统内存设置很大的场合，采用并行/并发垃圾回收会有更佳的效果。并行和并发垃圾回收的区别如下：

- 并行垃圾回收,这里的并行相对于垃圾回收线程自己，即垃圾回收线程有多个，垃圾回收线程是并行运行的。

- 并发垃圾回收,这里的并发是相对于Java应用程序而言的，垃圾回收和应用程序并发运行。

在不同的场合下有不同的垃圾回收参数设置，而这个设置某些情况下堆性能影响极大。

**并行垃圾回收** 垃圾回收线程有多个，因此叫做并行垃圾回收。但此时进行垃圾回收的时候，Java应用程序的运行要完全停下来。垃圾回收期间，所有的Java线程停止下来，将所有的CPU时间给并行垃圾回收器线程使用，以尽可能的加快垃圾回收过程，这个策略可以保证系统有大的吞吐量。如果多CPU的环境下，JVM在每一个CPU上都会启动一个垃圾回收线程，因此在多CPU（或者多核）的环境下，并行垃圾回收可以获得更好的性能。在单CPU上，并行垃圾回收并不能带来本质上的好处。如下图：

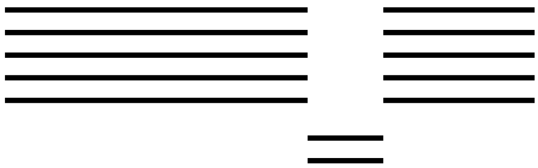



图 25 并行垃圾回收

**并发垃圾回收** 此时进行垃圾回收的时候，Java应用程序的运行不需要停下来，因此这里叫做并发垃圾回收。在并发垃圾回收模型下，一个专门的线程负责垃圾回收工作，与Java线程并发运行，这种方式可以极大得避免垃圾回收导致的系统暂停。在某些实时应用中，并发垃圾回收可以避免系统由于垃圾回收而导致的暂时停顿。如sip电话应用，系统暂停是会导致用户感受迟钝，或者重发消息加大等导致的接通率下降，如下图：

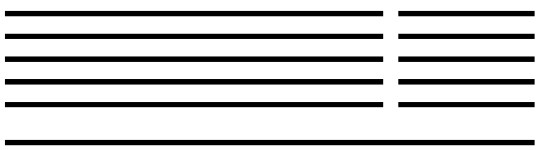



图 26 并发垃圾回收

#### §3.5.5 top陷阱：top真得能告诉你系统是否存在内存泄漏吗？

top是unix/linux下用来观察进程状态的工具，不同的操作系统下可能还有其它类似的工具，如prstat。从top中观察到的内存是整个Jvm所占用的总的内存，包括三部分之和：Java 堆内存（即真正java代码new使用的内存）,permSize (存放class的内存空间)，JVM自身运行需要的本地内存。同时垃圾回收的策略是，当满足一定的使用百分比才真正地进行回收，最关键的是，即使进行了垃圾回收，JVM也不会把这块内存归还给操作系统。从top中看，可能随着系统的运行，使用的内存在一定的时间段内一直在增长。因此无法通过top判断Java堆内存的真

实使用情况，也就无法判断系统是否存在内存泄漏。如果要观察堆内存的占用情况，只能通过java命令行中增加-verbose:gc，观察FULL GC的输出,详细请参考 71页第 §3.3.1节。

### §3.5.6 实时虚拟机

最近几年，随着Java在实时场合下的使用，实时虚拟机也应运而生。那么实时虚拟机到底提供了哪些普通虚拟机所不具备的特性？在介绍实时虚拟机之前，先介绍一下普通虚拟机在实时领域的不足。

- 非实时虚拟机完全垃圾回收时，要将Java应用程序完全停下来，在极端情况下，Java应用程序会有几十秒甚至更多的停顿[42]。

- 非实时虚拟机对何时进行垃圾回收完全无法预测。

在用于竞争性较高的环境中，在这种环境中，性能非常关键，每一毫秒都很重要。例如，特定的行业（比如电信或保险业）要求事务在给定的时间帧中以极低的延迟执行。然而，如果尝试用标准的Java来实现，则很可能会因为由垃圾收集过程所产生的无法预测的暂停时间而失败。实时虚拟机引入了一种确定性垃圾收集机制，提供了一个用于执行这些关键型应用的J2EE运行时。确定性垃圾收集确保程序执行期间的暂停时间非常短，而且挂起的请求会在定义的时间帧中得到处理从而允许购建高性能和确定性的应用程序。

常规的完全垃圾收集对Java性能有着很大的影响。在完全垃圾收集期间，Java进程会完全停止。当垃圾收集完成后，进程才会继续。从堆中清理废弃对象以及为新对象释放空间的过程需要进行高度优化，以便确保有效的内存管理。

实时虚拟机可以使用一种动态的"确定性"垃圾收集优先级。该策略被优化以确保暂停时间非常短，并限制定义良好的时间帧（也称为"滑动窗口"（sliding window））中的这些暂停的总次数。这对特定的应用程序来说很有用，尤其是对事务延迟有严格要求的应用程序。然而，即使较短的确定性暂停时间也不一定能保证较高的应用程序吞吐量。确定性垃圾收集的目标是降低在执行垃圾收集时运行的应用程序的延迟。与常规垃圾收集相比，确定性垃圾收集产生的暂停时间会短得多。

**关于实时** 首先思考一下该产品中的两个单词的意思："real"和"time"。关于"实时"（real time)，存在着许多种定义，许多文章也描述了不同的概念－－第一个Java Specification Request (JSR 1) 甚至就是专门针对此主题的。然而，实时的定义并不是一成不变的。没有人可以给出一个确定的定义，说明到底什么是实时以及如何确定它，为了更好地说明实时的概念，我们将引入几个常见的定义。根据Douglas Jenson对实时的讨论，存在两种类型的实时："软"实时和"硬"实时。

**硬实时** 定义了一个系统，其中所有可调度和不可调度的实体的执行都要遵守规定的完成时间约束。其它时间约束（也称为"上界"）可能也必须满足。实体的行为和运行时间是可预测的、确定的。

[42]在一个堆内存为16G的系统上，曾经遇到过一次完全垃圾回收消耗了五分钟!

**软实时** 表示不属于硬实时的所有其它实时情况。所有时间约束都是软性的。基本上，这就意味着所有可调度和不可调度的实体都可能被优化以便以最佳状态执行，但是执行时间不可预测。

在一个硬实时系统中，必须遵守时间期限，否则计算结果就会无效。例如，考虑汽车中的嵌入式系统。如果您要加速，而电子加速器的响应延迟了，那么就会得到无法预料的行为。而如果刹车系统延迟了，就会导致可怕的后果。软实时系统中的定时约束没这么严格，甚至在超出了时间约束之后，计算结果也可能仍然有用。音频流就是一个软实时系统的例子。如果一个数据包延迟或丢失了，音频的质量会降低，但是流可能依然有效。

为了保证满足实时系统的定时需求，底层计算系统的行为和计时必须是可预测的。一个可预测的定时系统，其所有操作所需的时间必须是有限制的。这意味着所有操作在最坏情况下的定时是已知的。实时系统通常都是用多个异步线程实现的。这是因为他们通常需要响应事件以及控制异步设备。

另一个常见需求是优先级。因为特定事件的紧急程度以及需要处理的事件的数目都不同，因此必须支持优先级的概念，以便确保时间关键型的任务不会由于非时间关键型的任务而被延迟。让非关键型的任务以与时间关键型的任务相同的优先级运行就有可能导致此问题。由于这种优先级倒置，任务需要具有互相通信的能力。因此，实时系统必须提供同步和通信功能。简而言之，实时系统必须以最小的开销实现可预测性。

下面讨论一些相关概念时所使用的其它术语和定义：

- 实时是一种计算机响应等级，在这种等级下，用户可感知到足够快，或者计算机能够跟得上某个外部流程。

- 延迟是指系统花在从一个指定点传输数据到另一个指定点的时间。

- 抖动是延迟偏差。一个具有确定性的应用程序抖动应该较低。该术语基本上描述了一种度量确定性的方法。

- 吞吐量是计算机在给定的时间帧中所能处理的工作量。

- 确定性垃圾收集是一个执行概念，用于描述快速的、可预测暂停时间的内存堆垃圾收集。垃圾收集是从堆中清理废弃对象以便收回空间用于新对象的过程。

**什么是"实时"？** 根据SUN实时Java规范工程师, 实时意味着"the ability to reliably and predictably reason about and control the temporal behavior of program logic."（可靠的可以预测地控制程序逻辑地临时行为），实时不意味着"快"。它意味着对一个实时事件的响应是可靠的，可预测的。实时计算机意味着在你给予的死限之内进行响应. 大量的应用领域不能忍受哪怕是一秒的延迟，如飞机控制软件，核电厂软件等等。实时系统并不都是速度上的要求，尽管实时系统设计者尽量使系统更快。很明显，标准的java虚拟机不能满足实时的要求，在java的license中已经明确说明了这一点，java不能用在核电系统，也不能用在军事防御系统等等。

**实时Java** Java作为实时应用最大的障碍是它的垃圾回收，实时的垃圾回收器是实时Java的核心部分。同时，实时Java在线程调度，同步，锁，类的初始化，最大响应延迟等方面也做了相应的增强。

其中不同的虚拟机提供商有不同的实现。像SUN的实时Java需要用户进行相应的代码修改才能做到实时（如使用RealtimeThread 来代替Thread等）。而Weblogic的实时虚拟机完全是透明的，用户代码不需要做任何修改。在这里很难说哪一种更好一些。代码透明并不意味着更好。

**非实时虚拟机** 非实时虚拟机的垃圾回收情况如下：

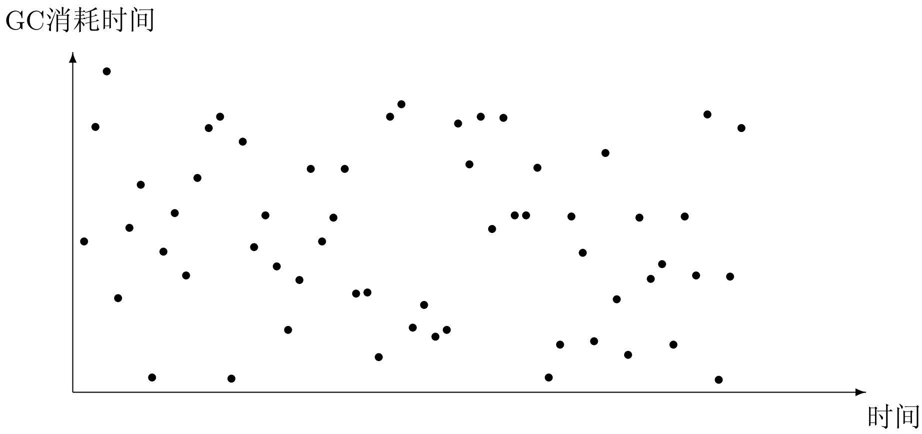



图 27 非实时虚拟机的垃圾回收占用时间

**实时虚拟机** 实时虚拟机的垃圾回收情况如下：

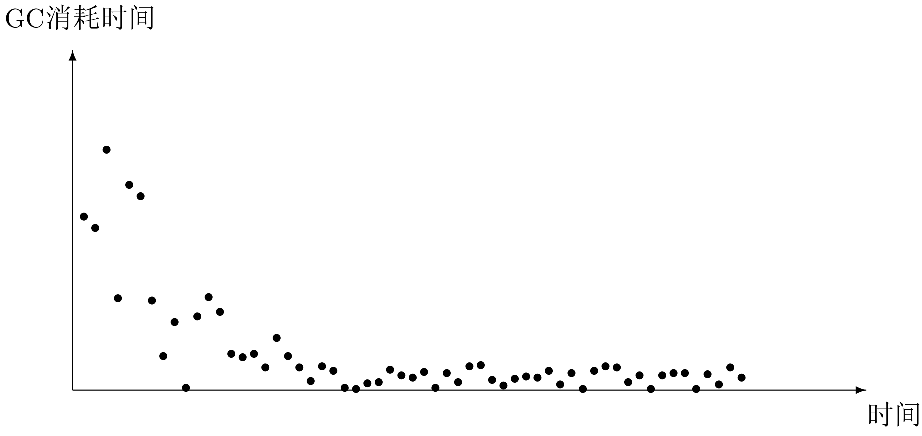



图 28 实时虚拟机的垃圾回收占用时间

## §3.6 关于JavaScript的内存泄漏

相关文档请参考[3]、[4]、[5]、[6]、[7]

# §4 关于并发和多线程

本章介绍了在Java 程序中使用线程需要注意的事项。

是否应该使用多线程在很大程度上取决于手头的应用程序的类型。如果应用程序是计算密集型（如纯数学运算）的，并受CPU 功能的制约，则只有多CPU(或者多个内核) 机器能够从更多的线程中受益，单CPU下，多线程不会带来任何性能上的提升，反而有可能由于线程切换等额外开销而导致性能下降。当应用程序必须等待缓慢的资源（如网络连接或数据库连接上的数据）时，多线程会让系统的CPU充分利用起来，当一个线程被阻塞时，另一个线程可以继续利用CPU。总之，使用多线程不会增加CPU 的处理能力。但在某些场景下可以更加充分地利用CPU。

由于同一进程的多个线程共享同一片存储空间，在带来方便的同时，在编程角度也带来了复杂性，如：如何保证多线程访问的数据一致性等。多线程编程属于编程中容易犯错误的地方。而且多线程编程问题测试定位比较困难。总的来说，好的多线程程序是写出来的而不是测出来的。将多线程问题寄希望于测试中发现，无疑是极度不可靠的。

Java api与多线程编程相关的三大关键字：synchronized、wait、notify. 理解这三个关键字，就可以编写多线程代码了，但是，多线程代码有许多要注意的地方。本章将详细介绍这些注意事项。

## §4.1 在什么情况下需要加锁？

在多线程场合，最重要的是确保多线程访问的数据一致性，而要保证数据的一致性，就需要借助于锁。在多线程的场合，首先要搞清楚到底什么需要保护？并不是所有的数据都需要加锁保护——只有那些被多个线程访问的共享数据才需要加锁保护。锁的本质上是确保同一时刻只能有一个线程访问到共享变量，那么该共享变量就能得到有效保护[43]。

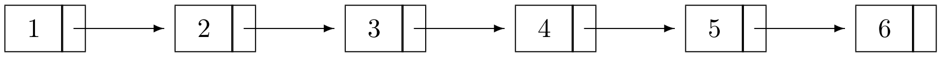
总得来说，首先要搞清楚什么变量需要进行多线程保护，然后在访问这个变量的代码段加上锁。

> ☞ 提示：
>
> 你保护的一定是变量，而不是代码。保护变量是通过代码段上加锁来实现的。而占有锁的则是线程。

下面以单向列表为例说明，假设我们要构造一个在多线程环境下线程安全的链表：

[43] HashMap被多线程访问下，可能导致put或者get方法block(或者死循环)

原始链表




插入一个元素

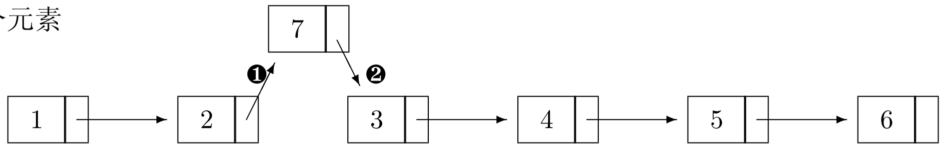



图 29 链表添加一个元素

假设有两个线程在同时操作这个链表(next成员变量指向下一个元素的对象引用)，一个线程插入一个元素（我们称之为写线程），另一个线程遍历该链表（我们称之为读线程）.其中读线程遍历整个链表，取出所有的链表数据，写线程向链表中插入一个元素7。如果不使用任何锁，那么可能会恰好导致了下面的执行序列：

| 时间点 | 写线程 | 读线程 |
|---|---|---|
| 0: | 修改元素2的next，即图中第❶步 | ... ... |
| 1: | ... ... | 遍历到元素7,next为空，遍历完成 |
| 2: | 修改元素7的next，即图中第❷步 | ... ... |

图 30 并发存取同一个链表

从图中可以看出，由于两个线程在同时操作这个链表，由于数据的不完整性，导致读线程并没有取到所有的元素，而只取从链表开始到元素7的位置. 由此可见不加任何保护的多线程访问，势必会造成混乱。为了避免多线程造成的数据不一致，需要对操作该队列的代码放入同步块中（锁对象就是这个链表实例），确保同一个时刻只有一个线程访问该链表。

加锁是为了保证数据的一致性，但同时可能引入了死锁的问题。死锁是一个经典的多线程问题，因为不同的线程都在等待那些根本不可能被释放的锁，从而导致所有的工作都无法完成[44]。

但只要按照下面几条规则去设计系统，就能够避免死锁问题。

- 让所有的线程按照同样的顺序获得一组锁。这种方法消除了X 和Y 的拥有者分别等待被对方占有的锁。

[44]死锁问题的定位请参考第 25页第 §1.3.1节

- 将多个锁组成一组并放到同一个锁下。只有一个锁，就不会存在死锁的问题。

## §4.2 如何加锁？

加锁的关键字synchronized有三种用法:

1. synchronized 方法：通过在方法声明中加入synchronized关键字来声明synchronized 方法. 每个类实例对应一把锁，因此实际上是所有的synchronized方法都使用这个实例(即this对象)作为锁对象。每个synchronized 方法都必须获得调用该方法的类实例的锁方能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为synchronized 的成员函数中至多只有一个处于可执行状态（因为至多只有一个线程能够获得该类实例对应的锁），从而有效避免了类成员变量的访问冲突。例如:

```
    public synchronized void synMethod() {
        //方法体
    }

```

   这也就是同步方法，那这时synchronized锁定的是哪个对象呢？它锁定的是调用这个同步方法的对象。也就是说，当不同的线程执行这个对象的这个同步方法时，它们之间会形成互斥。但是这个对象所属的Class所产生的另一对象却可以任意调用这个被加了synchronized关键字的方法，即该类的不同对象实例之间无任何互斥关系。

2. synchronized 块：通过synchronized关键字来声明synchronized 块,一次只有一个线程进入该锁的代码块.例如:

```
class MyClass
{
    private Object lock = new Object();   // 锁对象
    Public void methodA()
    {
        synchronized(lock) { ... ... }
    }
    ... ...
}
```

   谁拿到这个锁谁就可以运行该锁所在的那段代码（当然，每个需要访问该变量的地方都加上这种锁）。

3. synchronized在this对象上,此时,线程获得的是对象锁.例如:

```
public void synMethod()
{
```

```
    synchronized (this)      //this对象作为锁对象
    {
        //... ...
    }
}
```

等同于：

```
public synchronized void synMethod()
{
    //... ...
}
```

this指的就是调用这个方法的类实例对象，在对象级使用锁通常是一种比较粗糙的方法。这个需要慎重考虑，一不小心就会导致锁范围认为扩大，造成性能下降。

## §4.3 多线程编程中易犯的错误

- 锁范围过大，详见 §2.1 第 41页。
- 多把锁使用造成死锁，详见 §1.3.1 第 25页。
- 多个共享变量共用一把锁。特别是在方法级别上使用synchronized关键字，人为造成的锁竞争，详见 §2.1 第 41页。
- 无意识地启动线程过多，超过最大限制。如在某个时刻，一个任务一个线程，如果任务成千上万，需要同时启动成千上万个线程，那么就会有大量失败。
- 调用线程的interrupted()方法硬性去停止线程，而不是通过run()的正常退出(return)来结束线程。

## §4.4 i++这种仅有原子操作是否需要同步保护

i++表面上看时一个原子操作，但实际上不完全是，这个语句仍然需要同步保护，原因有如下两个：

- 一条累加语句对于risc CPU而言，对应多条指令，而不是一条指令，是非原子的，因此必须加以同步。
- 尽管long这种类型在32位的系统下面是原子变量，但在64位下面的long等原子数据类型实际上是非原子的。

## §4.5 进程线程多，是否就意味着我的程序可以获得更多的CPU？

只有当CPU成为整个系统的瓶颈，那么这句话就是成立的，也就是说，如果CPU一直在高位运行，那么线程多的进程，被执行到的几率就更高一些。在CPU不忙的时候，每个程序都能够得到及时的服务，该问题也就不存在。

### §4.6 线程的数量一般设为多少比较合理？

我们知道，多线程在大多数场合可以提高整个系统的性能或者吞吐量，但一个系统中到底多少个线程才是合理的？总的来说，线程数量过大过少都不好。过大导致线程切换开销过大，反而导致整个系统性能下降。过小导致CPU不能充分被利用，性能仍然上不去。系统到底使用多少线程，依据系统线程运行是否充分利用了CPU.如果每个线程都100%的使用CPU的话，那么系统一个线程就够了，但实际情况是，在如下情况下是不消耗CPU的：

- 磁盘IO
- 网络IO
- 带有3D加速卡的图形运算
- 等待输入

特别是磁盘IO,网络IO相比CPU的速度，是非常慢的，也就是说，在这很长的时间CPU是空闲的，此时系统如果有多个线程那么其它线程可以在该线程空闲的时候利用CPU,从而提高CPU的利用率，系统总的吞吐量也就上去了。当实际的应用系统中，如果只有一个线程的话，该线程有访问远程数据库的行为，那么在等待数据返回的期间，这期间是不消耗CPU的。示意图如下，从下图可以看出，如果系统只有一个线程，那么CPU绝大多数时间是空闲的，因此整个系统的性能肯定很低。

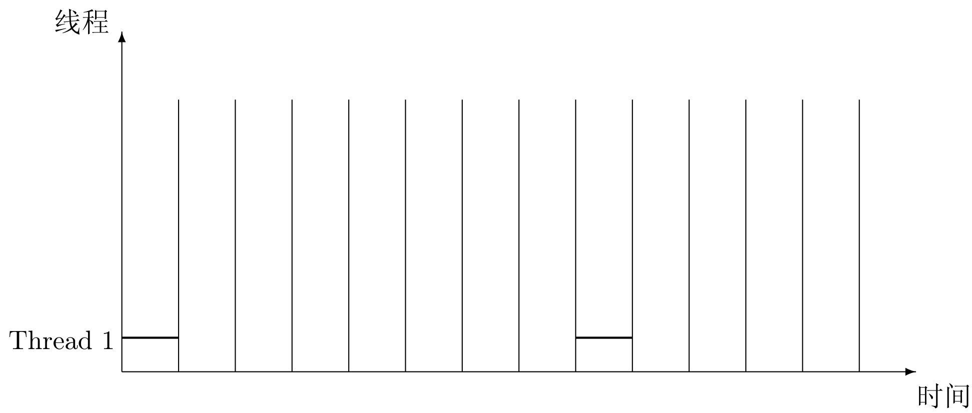



图 31 单线程下CPU的使用情况

如果我们采用多线程，那么其它线程就会把这段空闲时间充分利用起来，性能会有大幅的提升，如下图：

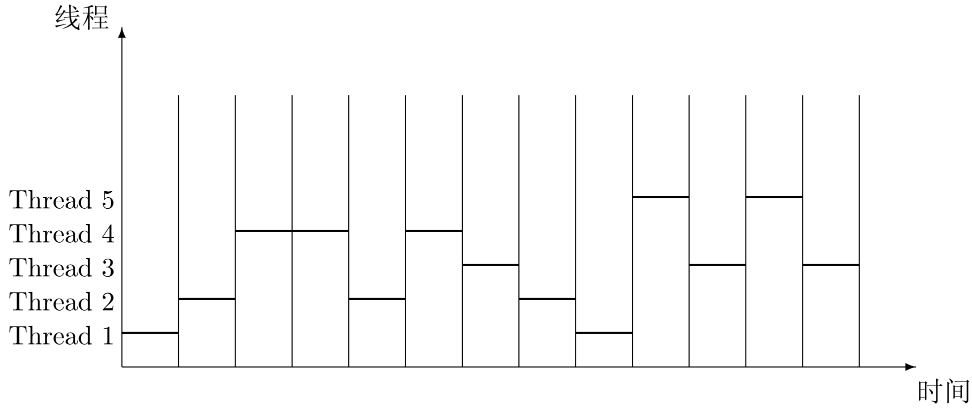



图 32 多线程下CPU的使用情况

当然，理想情况是把整个CPU全部给利用起来，但实际上，可能做不到这一点，只要将CPU的利用趋近于饱和就可以了。同样的，从上图可以看出，如果一段代码是高CPU消耗的代码（如数据运算），那么一个线程就足够了，线程多了，反而由于线程上下文切换的开销，会降低系统的性能。

总得来说，一段代码导致CPU空闲的比例越大(空闲的CPU周期/总的CPU周期)，那么线程的数量就应该加大，当一个线程被阻塞时，其它线程可以继续执行业务代码，这样可以充分利用CPU。不能简单得说线程多性能就好，或者说线程少性能就好。在一种应用下到底需要多少线程，不是取决于线程数量本身，而是取决于你的具体应用类型。如果执行线程不消耗CPU的时间片越大，那么线程数量大对性能就好。当然，一个系统只通过调整线程的数量，不一定能带来性能的真正提高，比如，设计/编码不合理导致系统中存在资源争用（比如长期锁等待），此时只靠调整线程的数量，可能根本不会有任何效果，在这种情况下,随着压力的加大，CPU的使用率并不能一直上升并趋近于饱和(即100%),往往只能达到某一个中间值，随后随着压力的增大，系统的失败率开始上升。因此一个设计良好的系统，需要考虑各种因素，才能将性能调到最大。

## §4.7 关于线程池

线程池设计一般有两种思路：

- 线程池初始化时，即将必要数量的线程创建出来，并一直存在，直到整个系统shutdown.
- 线程池在初始化时，即创建很少数量的线程，当系统压力上去时，并导致当前线程数量不足时，那么会创建新的线程，一旦系统压力降下去，那么部分线程将被销毁。这种线程池的线程数量是在动态变化的。

如果系统有可能在某个时刻任务过多，那么使用线程池要特别小心，因为这些任务可能将线程池中的所有线程耗光，同时将任务队列塞满，从而造成任务提交失败，因此将一个任务提交给线程池的时候，已经要对提交是否成功进行检查，如果提交不成功，就需要进行日志纪录，以方便定位问题，否则这种问题非常难以定位。

## §4.8 notify和wait的组合

notify和wait的结合使得我们可以实现线程间通信，用于解决各种复杂的线程时序问题。关于wait()[45] 和notify() 方法再说明两点：

1. 调用notify() 方法导致解除阻塞的线程是从因调用该对象的wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

2. 除了notify()，还有一个方法notifyAll() 也可起到类似作用，唯一的区别在于，调用notifyAll() 方法将把因调用该对象的wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态，其它继续回到等待状态。

多线程之间需要协调工作。例如，浏览器的一个显示图片的线程displayThread想要执行显示图片的任务，必须等待下载线程downloadThread将该图片下载完毕。如果图片还没有下载完，displayThread可以暂停，当downloadThread完成了任务后，再通知displayThread“图片准备完毕，可以显示了”，这时，displayThread继续执行。以上逻辑简单的说就是：如果条件不满足，则等待。当条件满足时，等待该条件的线程将被唤醒。在Java中，这个机制的实现依赖于wait/notify。等待机制与锁机制是密切关联的。

```
    synchronized(obj) {
        while(!condition) {
            obj.wait();
        }
        obj.doSomething();
```

当线程A获得了obj锁后，发现条件condition不满足，无法继续下一处理，于是线程A就wait()。在另一线程B中，如果B更改了某些条件，使得线程A的condition条件满足了，就可以唤醒线程A：

```
    synchronized(obj) {
        condition = true;
        obj.notify();
    }
```

另外如果唤醒的代码如下（唤醒之后的运行期代码仍然在同步块中）：

```
    synchronized(obj) {
        condition = true;
        obj.notify();
        ... ...        //其它代码
    }
```

[45]关于wait和sleep的区别请参考第 §1.2.2节第 14页。

当其它代码执行完成时，obj.wait()才能被唤醒。也就是说obj.notify()并不是立即能将obj.wait()唤醒，只有当正在执行obj.notify()的线程将锁释放，obj.notify()才能被真正地唤醒。例如：

```
package MyPackage;

public class TestThread_Notify extends Thread{
    Object lock = null;
    public TestThread_Notify(Object lock_)
    {
        lock = lock_;
        this.setName(this.getClass().getName());
    }
    public void run()
    {
        fun();
    }

    public void fun(){
        synchronized(lock){
            lock.notify();
            System.out.println("Have notified");
            try{
                Thread.sleep(2000);
            }catch(Exception e){
                e.printStackTrace();
            }
            System.out.println("sleep complete");
        }
    }
}

package MyPackage;

public class TestThread_Wait  extends Thread{
    Object lock = null;
    public TestThread_Wait(Object lock_)
    {
        lock = lock_;
        this.setName(this.getClass().getName());
    }
    public void run()
    {
        fun();
```

```
    }

    public void fun(){
        synchronized(lock){
            try{
                lock.wait();
            }catch(Exception e){
                e.printStackTrace();
            }
            System.out.println("is notified");

        }
    }
}

package MyPackage;

public class ThreadTest {
    public static void main(String[] args) {
        Object shareobj = new Object();
        TestThread_Wait thread1 = new TestThread_Wait(shareobj);
        thread1.start();

        TestThread_Notify thread3 = new TestThread_Notify(shareobj);
        thread3.start();
    }

}
```

打印结果如下：

```
Have notified
sleep complete
is notified
```

从打印的结果来看，lock.notify()执行之后，并没有马上唤醒等待该锁的线程，而是该同步块完全执行完之后，等待该锁的线程才被真正的唤醒。需要注意的概念是：

1. 调用obj的wait(), notify()方法前，必须获得obj锁，即wait()方法必须写在synchronized(obj) ... 代码段内
2. 调用obj.wait()后，线程A就释放了obj的锁，否则线程B无法获得obj锁，也就无法在synchronized(obj) ... 代码段内唤醒A。
3. 当obj.wait()方法返回后，线程A将再次获得obj锁，才能继续执行。

4. 如果A1,A2,A3都在obj.wait()，则B调用obj.notify()只能唤醒A1,A2,A3中的一个（具体哪一个由JVM决定）。

5. obj.notifyAll()则能全部唤醒A1,A2,A3，但是要继续执行obj.wait()的下一条语句，必须获得obj锁，因此，A1,A2,A3只有一个有机会获得锁继续执行，例如A1，其余的需要等待A1释放obj锁之后才能继续执行。

6. 当B调用obj.notify/notifyAll的时候，此时B正持有obj锁，因此，A1,A2,A3虽被唤醒，但是仍无法获得obj锁。直到B退出synchronized块，释放obj锁后，A1,A2,A3中的一个才有机会获得锁继续执行。

下面谈一谈一些常用的方法: wait(),wait(long),notify(),notifyAll()等方法是当前类的实例方法,

- wait()是使持有对象锁的线程释放锁;
- wait(long)是使持有对象锁的线程释放锁时间为long(毫秒)后,再次获得锁,wait()和wait(0)等价;
- notify()是唤醒一个正在等待该对象锁的线程,如果等待的线程不止一个,那么被唤醒的线程由jvm确定;
- notifyAll是唤醒所有正在等待该对象锁的线程.我们应该优先使用notifyAll()方法,因为唤醒所有线程比唤醒一个线程更容易让jvm找到最适合被唤醒的线程.
- 有synchronized的地方不一定有wait,notify
- 有wait,notify的地方必有synchronized.这是因为wait和notify不是属于线程类，而是每一个对象都具有的方法，而且，这两个方法都和对象锁有关，有锁的地方，必有synchronized。

synchronized和wait,notify没有绝对的关系,在synchronized声明的方法、代码块中,你完全可以不用wait,notify等方法,但是, 如果当线程对某一资源存在某种争用的情况下,你必须适时得将线程放入等待或者唤醒.

> ❏思考：当前线程wait()时间已到，而其它线程正在占有锁执行一个耗时操作，当前线程能否及时获得该锁？

## §4.9 线程的阻塞

为了解决对共享存储区的访问冲突，Java 引入了同步机制，现在让我们来考察多个线程对共享资源的访问，显然同步机制已经不够了，因为在任意时刻所要求的资源不一定已经准备好了被访问，反过来，同一时刻准备好了的资源也可能不止一个。为了解决这种情况下的访问控制问题，Java 引入了对阻塞机制的支持。

阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。

1. sleep() 方法：sleep() 允许指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到CPU 时间，指定的时间一过，线程重新进入可执行状态。典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。

2. suspend() 和resume() 方法：两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用resume() 使其恢复。

3. yield() 方法：yield() 使得线程放弃当前分得的CPU 时间，但是不使线程阻塞，即线程仍处于可执行状态，随时可能再次分得CPU 时间。调用yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程。

4. wait() 和notify() 方法：两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的notify () 被调用。

初看起来它们与suspend() 和resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。述的核心区别导致了一系列的细节上的区别。首先，前面叙述的所有方法都隶属于Thread 类，但是这一对却直接隶属于Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用任意对象的notify()方法则导致因调用该对象的wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在synchronized 方法或块中调用，理由也很简单，只有在synchronized 方法或块中当前线程才占有锁，才有锁可以释放。

同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现IllegalMonitorStateException 异常。

wait() 和notify() 方法的上述特性决定了它们经常和synchronized 方法或块一起使用，将它们和操作系统的进程间通信机制作一个比较就会发现它们的相似性：synchronized方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于block 和wakeup 原语（这一对方法均声明为synchronized）。

它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。关于wait() 和notify() 方法最后再说明两点：

1. 调用notify() 方法导致解除阻塞的线程是从因调用该对象的wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

2. 除了notify()，还有一个方法notifyAll() 也可起到类似作用，唯一的区别在于，调用notifyAll() 方法将把因调用该对象的wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

## §4.10 Java线程的优先级

Java 线程模型支持线程优先级。本质上，线程的优先级是从1 到10 之间的一个数字，数字越大表明任务越紧急。JVM 首先调用优先级较高的线程，然后才调用优先级较低的线程。但是，该标准对具有相同优先级的线程的处理是随机的。如何处理这些线程取决于基层的操作系统策略。在某些情况下，优先级相同的线程分时运行；在另一些情况下，线程将一直运行到结束。请记住，Java 支持10 个优先级，基层操作系统支持的优先级可能要少得多，这样会造成一些混乱。因此，只能将优先级作为一种很粗略的工具使用。通常情况下，请不要依靠线程优先级来控制线程的状态。

但是，不要指望利用线程优先级来解决系统中优先级高低的问题。系统中存在更重要的任务，但重要并一定意味着真正的高优先级。当系统负荷不高的时候，设置优先级基本上没有什么价值。在负荷比较高的情况下，设置优先级基本上解决不了你的问题。

实际上，Java的GC会对实时性造成极大的削弱，通过控制线程优先级即使能获得一点点的实时好处，但是也被GC特性给大大削弱了。因此设置线程优先级不会从根本上获得你所需要的效果。

## §4.11 关于多线程的一些错误观点

**线程数量多会导致性能下降，并导致系统不稳定** 每个人都有一个线程数量上的心理"价位",有的人潜意识里面认为一个系统几十个线程比较合理，有的人认为几百个也是合理的。目前的大型应用系统，如果一个系统中存在2000个线程，不要大惊小怪。目前很多大型的Java应用程序运行在具有多CPU(或者多核)的硬件上，而这种硬件随着多核CPU的普及，这种硬件变得越来越普通。

**过多的线程处于等待状态，是系统设计不合理，或者线程过渡使用**

**从JConsole中看到，线程工作量的均匀分布才是合理的**

# §5 幽灵代码

本章介绍一些常见的幽灵代码模式。之所以说这些类型的代码是"幽灵",是因为他们往往来无踪去无影。代码在外面看起来,似乎无懈可击,但正在运行的系统无缘无故突然不可用,也许重启后很长时间又不再出现,当我们认为事情已经过去的时候[46],某天像幽灵一样又再次出现,安静的生活时不时被这些幽灵打乱。

这些代码模式,潜藏在系统中,是一个一个的地雷。

## §5.1 异常退出幽灵代码

在Java中,引入了异常处理机制。函数通过抛出异常,可以把控制还给它的调用者,由于异常的存在,我们必须意识到控制可能永远无法达到函数的"正式"的结束点,而是直接跳转至调用者。在某些场合下,我们应该确保异常不被忽略。这种异常处理机制的这个处理规则特别隐蔽,往往踏水无痕,藏于无形之间,如果函数没有对一种异常进行捕获,一旦发生异常,那么将自动退出函数,从而造成应该执行到的关键代码遗漏。这一个自动退出,在某些场合下会导致人为无意识的疏忽。而这种疏忽往往让系统存在很大的隐患。特别是与资源处理相关的代码,这种疏忽造成的问题会严重影响系统的稳定性。而这种问题由于在正常测试下(正常的功能测试很少抛异常)一般不会暴露,它的暴露往往是在生产环境下才会暴露,而且往往运行了很长时间才会暴露。之所以称这种代码模式为幽灵代码,是因为它常常来无影去无踪(常常午夜凶铃惊醒你),非常难以抓到现形。

下面就介绍一下这种非常隐蔽的幽灵代码模式。这种代码模式,是指由于函数有未捕获的异常,导致函数异常退出,而函数中的重要代码没有得到执行。总之,如果由于中间的代码抛了异常,导致后面的关键代码没有被执行到,都属于严重影响稳定性的问题。具体影响有多大,依赖于这个关键代码在系统中的影响有多大[47]。比如,我们知道,对于一些资源(系统资源或者自定义资源),使用与关闭需要成对出现,如果由于异常退出导致关闭资源的代码没有被执行到,那么则会造成资源泄漏。资源泄漏多了,就会导致整个系统无法工作。再具体一点,比如打开一个文件,由于异常退出导致关闭文件代码没有被执行,则会造成文件句柄泄漏。打开一个socket,不需要的时候没有close也会造成文件句柄泄漏,打开一个数据库连接,使用完后没有关闭,就会造成连接泄漏,等等。

下面我们就介绍一下这种由于异常退出幽灵代码模式:

```
    ... ... //其它可能抛出异常的代码

    ... ... //资源清理等重要必须要执行到的代码，当上面的代码抛出异常等，
            //导致该语句得不到执行，那么资源就造成了泄漏

    ... ...
```

[46]有的程序员喜欢把这种问题归为环境问题,用来"忽悠"自己的上司和老板,其实心里是很愧疚和忐忑不安的。

[47]第 §11.1.2节第 176页是一个这种幽灵代码导致的致命问题案例

这种代码在很多场合，会导致严重的系统挂死等致命问题，我们称之为幽灵代码。大家也会看到，这种幽灵代码[48]会不时地在本文中出现。如上幽灵代码应修改为：

```
try{
    ... ...             //其它可能抛出异常的代码
}
catch(Throwable t){  //最好使用更低级别的Throwable而不是Exception
    ... ...             //抓住这些异常，确保后面的资源清理代码任何时候都可以得到执行
}
... ...                 //资源清理代码
```

或者：

```
try{
    ... ...             //其它可能抛出异常的代码
}
catch(MyException e){ //自定义的异常
    ... ...
}finally{    //通过finally,确保任何情况下，资源清理代码都会得到执行。
    ... ... //资源清理代码
}
```

这种幽灵代码模式，会导致如下问题：

- 由于异常没有抓住导致后面的关键代码得不到执行，资源得不到释放,常见的资源有
    1. 文件句柄（文件或者socket）
    2. 数据库连接
- 由于不可预知的异常抛出，导致永久生命周期的线程异常退出。
- 其它

从上面的介绍看，避免这个幽灵代码，修改的方法有两种，其中之一是将关键代码放在catch(Throwable t)异常处理代码中，确保任何异常情况下，关键代码仍然可以被执行，那为什么在关键场合需要catch(Throwable),而不是catch(Exception)呢? 在一般情况，catch(Exception)基本上能够捕捉到绝大多数异常，但是在苛刻的运行环境下，仍然有漏网之鱼，这在某些应用下，一次遗漏也会导致致命的问题。Throwable比Exception更为低级一些，可以保证所有的异常都能被捕获，从而使得在任何情况下，"善后"代码都能得到执行。为了说明在极端苛刻的情况下，系统可能抛出Throwable，而不是Exception级别的异常，下面列出了Java中定义的Throwable级别的异常。从下面的列表可以看出，java.lang.Exception继承自java.lang.Error,而还有很多其

[48] 在C++中也支持异常这种概念，但并不是每个人都喜欢这种错误处理模式，当然在C++中不使用异常机制是基于效率的考虑，但还有一个很重要的原因是这种不可预知的代码返回点导致的系统问题，即上面所提到的幽灵代码模式。在代码中使用错误码返回，确可以容易避免这种错误的情况，这一点再次验证了"简单就是美"的哲学

它的异常继承自java.lang.Error，如sun.management.AgentConfigurationError等。如果代码中，只catch了java.lang.Exception,那么其它的异常将无法被捕获。如在某些极端情况下，不恰当的代码有可能存在StackOverflowError的问题，同样，瞬间的高访问量，也可能导致暂时的内存不足（java.lang.OutOfMemoryError）。如果因为这一两次失败，导致整个系统不可用，那么整个系统可靠性是比较低的。一个好的系统需要确保能够自动恢复。因此在关键的代码处，捕获Throwable相比捕获Exception更加安全可靠。

```
java.lang.Object
  |--java.lang.Throwable
  |       |--java.lang.Error
  |       |    |--sun.management.AgentConfigurationError
  |       |    |--java.lang.annotation.AnnotationFormatError
  |       |    |--java.lang.AssertionError
  |       |    |--sun.awt.DebugHelperImpl.AssertionFailure
  |       |    |--java.awt.AWTError
  |       |    |--java.nio.charset.CoderMalfunctionError
  |       |    |--javax.xml.transform.FactoryFinder.ConfigurationError
  |       |    |--... ...
  |       |    |--java.lang.LinkageError
  |       |    |    |--java.lang.ClassCircularityError
  |       |    |    |--java.lang.ClassFormatError
  |       |    |    |    |--java.lang.reflect.GenericSignatureFormatError
  |       |    |    |    |--java.lang.UnsupportedClassVersionError
  |       |    |    |--java.lang.ExceptionInInitializerError
  |       |    |    |--java.lang.IncompatibleClassChangeError
  |       |    |    |    |--java.lang.AbstractMethodError
  |       |    |    |    |--java.lang.IllegalAccessError
  |       |    |    |    |--java.lang.InstantiationError
  |       |    |    |    |--java.lang.NoSuchFieldError
  |       |    |    |    |--java.lang.NoSuchMethodError
  |       |    |    |--java.lang.NoClassDefFoundError
  |       |    |    |--java.lang.UnsatisfiedLinkError
  |       |    |    |--java.lang.VerifyError
  |       |    |--sun.nio.ch.Reflect.ReflectionError
  |       |    |--sun.misc.ServiceConfigurationError
  |       |    |--javax.swing.text.StateInvariantError
  |       |    |--java.lang.ThreadDeath
  |       |    |--com.sun.jmx.snmp.IPAcl.TokenMgrError
  |       |    |--javax.xml.transform.TransformerFactoryConfigurationError
  |       |    |--java.lang.VirtualMachineError
  |       |    |    |--java.lang.InternalError
  |       |    |    |--java.lang.OutOfMemoryError
  |       |    |    |--java.lang.StackOverflowError
  |       |    |    |--java.lang.UnknownError
  |       |    |--java.lang.Exception //只捕获Exception并不能保证捕获了所有的异常
  |       |    |    |-- ... ...       //该异常上面的异常都不属于java.lang.Exception
```

### §5.1.1 异常退出幽灵代码导致的资源泄漏

异常幽灵代码可能会隐藏很多可靠性问题在系统中，但该幽灵代码导致的资源泄漏是最常见的问题，本节将就资源泄漏进行详细的说明。当申请的资源没有得到释放时，就造成了资

源泄漏。资源的泄漏是程序申请了资源，但由于程序的缺陷，最终失去了对资源的控制而无法释放。常见的资源有：

- 自定义资源，如：连接池，线程池。
- 系统资源，如：文件句柄（TCP/IP连接或者文件）。
- 线程。
- 对象内存，等等。
- 锁对象无法被释放。在JDK1.5中，JDK提供了除sychronized之外的锁实现机制,这些锁要求显式去获取锁，释放锁。如ReentrantLock,ReentrantLock.lock()和ReentrantLock.unlock()必须成对出现。在下面所介绍的幽灵代码模式下，一旦处理不当就会造成执行unlock()遗漏，那么其它所有请求该锁的线程，会永远地被阻塞在那里。
- 等等... ...

对于上面提到的资源，当使用完成后，一定要显式调用关闭资源的代码。一旦遗漏，势必造成资源的耗尽，最终导致系统无法工作。

有的程序员认为，在对象离开作用域之后，Java启动垃圾回收，而垃圾回收会自动将资源释放掉，这种说法实际上是不成立的。Java的自动垃圾回收只能保证内存被回收，不能保证资源被回收，二者是不同的概念。不同的资源回收有不同的方法，总而言之，必须显式地调用资源的回收代码，资源才能被真正地回收。我们分别以文件和数据库连接池资源为例来说明一下这个问题：

```
Context  ctx  =  new  InitialContext();
DataSource  ds  =  (DataSource)ctx.lookup(dataSourceName);
Connection  con  =  ds.getConnection();
... ...
conn.close();  //将连接回池(即资源回收)
```

在这里的资源回收含义是将数据库连接释放回池，该连接可以再次被其它地方申请使用。Connection的close()方法执行的是将连接回池操作，只有Connection.close()方法被调用，才能确保数据库连接资源被回收。有兴趣的可以参考一下tomcat的数据库连接池的close()源代码实现。在这里自动垃圾回收对资源回收没有任何关系，自动垃圾回收只做了它应该做的事情，该我们自己做的还需要我们自己做。

```
FileInputStream infile =  FileInputStream("c:\\test.txt");
int n = infile.read(buff); //从文件读取数据
infile.close();            //关闭文件
```

在这里资源回收的含义是告诉操作系统将文件关闭，接着操作系统会清空该文件内存缓冲，释放文件句柄等相关操作。只有FileInputStream.close()方法被调用，才能确保文件句柄资源被回收清理，否则就会造成文件句柄泄漏，当泄漏到一定程度之后，由于打开的文件句柄的数量超过了操作系统下对每一个进程的最大限制数量，那么该系统无法再打开文件。在这里，自动垃圾也不能帮助我们自动清理资源。再如下代码段：

```
    stream.read();
    ... ... //这里其它的代码，如果抛出异常，那么下面的stream.close()
            //就无法执行到，导致stream永远得不到关闭。
    stream.close();
```

应该修改成如下代码:

```
    stream.read();
    try{
        ... ...  //其它可能抛出异常的代码
    }
    catch(Throwable t){  //最好使用更低级别的Throwable而不是Exception
        ... ...          //抓住这些异常，确保后面的资源清理代码任何时候都可以得到执行
    }
    stream.close();      //资源清理
```

或者

```
    stream.read();
    try{
        ... ...  //其它可能抛出异常的代码
    }
    catch(){
        ... ...
    }
    finally{
      stream.close();    //资源清理
    }
```

实际上，异常的这种代码模式（一旦异常，如果没有捕获，自动退到上一级），在资源管理方面存在先天的不足。当外层代码无法预知所调用的函数抛出什么类型的异常时，为了避免出现以上所提到的资源泄漏，往往只有两个选择：

- 捕捉所有异常，在异常代码中加入资源清理的代码，确保资源获取与释放成对出现[49]。
- 增加finally语句，将异常清理的代码放在其中。

C++中也提供了异常机制，但很多C++程序中根本没有使用语言级别提供的异常处理机制，反而使用类似错误码的机制，手工控制return 的时机，这里面除了性能的考量因素之外，上面提到的关于异常处理的代码结构容易隐藏问题也是原因之一。毕竟自己控制return点所有都在自己控制之下，不可预知的资源清理问题（如内存释放）就会得到很好的控制。

[49]在某些情况下,系统抛出的可能是Throwable异常，而不是Exception异常，所以在一些极端的场合，需要catch(Throwable t) 才更加安全

## §5.2 wait()与循环

下面是一种常见的代码模式：

```
public synchronized void myfun()
{
    if(!available){
        try{
           lock.wait();
        }catch(Exception e){}
    }
    ... ...
}
```

上面的代码有如下问题：

1. 正常情况wait()被唤醒，是由于持有锁的另一个线程调用了notify()或者notifyAll(),但另一个线程也可能调用了Thread.interrupted(), 此时该线程被唤醒，但实际上条件可能并不满足，即available仍然是false。

2. 在这个线程被唤醒并获得锁(当前线程调用到wait()上会释放所持有的锁，一旦退出wait()，由于所执行代码仍然在synchronized代码块中，因此该线程又一次占有锁，详细请参考第§1.2.2节第 14页。)之间有一个时间窗，在这个期间，available变量可能被其它线程修改，仍然为false.

正确的应该将wait()放在一个循环中，当该线程被唤醒重新检查条件，如果满足，则继续，如果不满足则继续等待。

```
public synchronized void myfun()
{
    while(!available){
        try{
           lock.wait();
        }catch(Exception e){}
    }
    ... ...
}
```

将wait()放在循环中，就避免了上面提到的两种情况，在任何情况下，都能执行正确的代码逻辑。

## §5.3 Double-Checked Locking单例模式

Double-Checked Locking(双检查锁)是一种非常广泛使用的一种代码模式，用在多线程场合下的懒初始化(lazy initialization)。

```
public MyInstance getInstance()
{
    if(this.instance == null){
        synchronized(this){
            this.instance = new MyInstance();
        }
    }
    return this.instance;
}
```

有可能两个或者两个以上的线程同时执行到语句if(this.instance == null)，此时this.instance为null,这样的话，就会这几个线程都会执行new MyInstance()语句，如下的执行时序必然造成一个实例被创建了多次,如下图：

| 时间点 | 线程A | 线程B |
|---|---|---|
| 0: | 检查this.instance，为空 | 检查this.instance，为空 |
| 1: | 执行new MyInstance()创建一个实例 | ... ... |
| 2: | ... ... | 执行new MyInstance()创建一个实例 |

图 33 Double-Checked Locking单例模式多线程场合下可能的执行时序

为了避免这个由于多线程导致的多次创建对象，直接将this.instance == null 的检查放在同步块中即可，如：

```
public synchronized MyInstance getInstance()
{
    if(this.instance == null){
        this.instance = new MyInstance();
    }
    return this.instance;
}
```

详细请参考[23]

## §5.4 另一种异常陷阱-连续的关键接口调用

在某些场合下，要求在任何情况下某些代码一定要被执行。这里以文件句柄为例，两个文件流使用完毕后，一定要关闭，否则会造成文件句柄泄漏。如果恰好有超过两个的文件流要关闭，习惯上我们可能将多个文件流的关闭放在一起，如下：

```
try{
```

```
        stream1.close(); //如果这句代码如果抛出异常，
        stream2.close(); //那么这一行将得不到执行，造成一个文件句柄泄漏
      }  catch(Exception e){}
```

这种代码写法隐藏着一些问题，因为第一句可能失败，并抛出异常造成第二句没有得到执行，从而造成一个文件句柄泄漏，如果这种情况发生多次，势必会耗尽所有的文件句柄。

安全的代码应该改为,将每一个文件的关闭放在各自的try catch块中，避免句柄泄漏.

```
      try{
        stream1.close();
      }catch(Exception e){}
      try{
        stream2.close();  //stream2的关闭不受stream1的影响。
      }catch(Exception e){}
```

在实际应用中，有很多资源的关闭或释放可能有同样的问题，比如：

1. 连续关闭多个文件或者socket
2. 连续关闭多个数据库连接
3. 连续调用多个Parlay接口,当拆呼叫时，要求每一个Parlay拆话接口都要调用到，如果一个抛出异常，其它接口没有调用到的话，由于相关对象被引用，常常会导致内存泄漏等问题
4. 等等

# §6 常见的Java泥潭

本章介绍常见的Java陷阱和泥潭。

## §6.1 不稳定的Runtime.getRuntime().exec()

Runtime.getRuntime().exec()可以执行一个外部程序。该API创建一个外部进程，并返回一个Process的子类对该进程进行控制。类Process提供了执行进程输出，输入，等待完成，获取错误码，以及杀死进程的方法。

但JDK提供的这个API在实际运行中有很多不稳定性。JDK中对java.lang.Process有如下描述[10]：

| | |
|---|---|
| The methods that create processes may not work well for special processes on certain native platforms, such as native windowing processes, daemon processes, Win16/DOS processes on Microsoft Windows, or shell scripts. The created subprocess does not have its own terminal or console. All its standard io (i.e. stdin, stdout, stderr) operations will be redirected to the parent process through three streams (getOutputStream(), getInputStream(), getErrorStream()). The parent process uses these streams to feed input to and get output from the subprocess. Because some native platforms only provide limited buffer size for standard input and output streams, failure to promptly write the input stream or read the output stream of the subprocess may cause the subprocess to block, and even deadlock. | 该方法在某些本地平台上， 特定的进程也许不能很好地工作，如本地的windows进程,daemon进程， windows上的16位或者Dos程序，或者脚本程序。由于创建的子进程没有自己的终端或者控制台，所有的标准IO(stdin,stdout,stderr)的输出将被通过三个流(getOutputStream(), getInputStream(), getErrorStream())重定向到父进程中，父进程通过这些流来进行io。由于某些本地平台对这三个流仅提供有限大小的缓冲区，对子进程输入流的写或者对输出流的读上的失败可能导致子进程被阻塞，甚至导致死锁。 |

在实际使用过程中，往往会发现如下问题：

- 在windows平台下可用的外部进程调用，在Unix/Linux确不可用，或者导致了程序挂起
- 调用exe程序没有问题，调用脚本程序却没有成功。

- 正常退出的脚本执行没有问题，异常退出的脚本却导致了exec()的挂起
- 返回结果不正确等。

对此，参考文献[9]对此有非常深入的分析。

通过java.lang.Runtime.getRuntime()可以获取Java运行期对象, 借助该引用可以通过exec()函数执行外部程序，如开发者经常使用这种方法启动一个浏览器来显示html帮助，exec()共有如下四个版本：

- public Process exec(String command);
- public Process exec(String [] cmdArray);
- public Process exec(String command, String [] envp);
- public Process exec(String [] cmdArray, String [] envp);

通过这几个方法，可以将命令和相应的参数传给操作系统，操作系统创建一个进程（一个运行的程序），并将Process类的对象引用，返回给java虚拟机。Process是一个抽象类，每一个操作系统对应一个Process的一个子类。这些方法有如下集中参数形式：

1. 包含了程序名称和参数一个完整的字符串（空格作为分隔符）
2. 程序和参数分离的一个String数组。
3. 一组环境变量形成的数组（环境变量格式为name=value）

Runtime.exec() 的第一个陷阱是*IllegalThreadStateException.* 下面是第一个例子，例子中调用javac外部程序：

```
public class ExitPitfall {
    public static void main(String[] args) {
        try
        {
            Runtime rt = Runtime.getRuntime();
            Process proc = rt.exec("javac");
            int exitcode = proc.exitValue();
            System.out.println("exit code: " + exitcode);
        }
        catch (Throwable t){
            t.printStackTrace();
        }
    }
}
```

如上程序产生如下的输出:

```
C:\work\sketch\Java\pitfall_exec>java -classpath bin ExitPitfall
java.lang.IllegalThreadStateException: process has not exited
        at java.lang.ProcessImpl.exitValue(Native Method)
        at ExitPitfall.main(ExitPitfall.java:8)
```

如果外部进程没有完成，exitValue() 方法将回抛出IllegalThreadStateException异常; 那么如何来解决这个问题呢？通过使用waitFor()可以避免该异常。实际上通过waitFor()也可以获取返回值。这意味着不需要将exitValue() 和waitFor() 联合使用, 任何一个都可以完成该功能.但是有一种情况二者有明显的差异：当外部程序永远不退出时，如果你不希望你的java程序被阻塞，这个时候应该使用exitValue() 而不是waitFor().

因此，可以通过catch IllegalThreadStateException 异常或者通过使用waitFor()等待外部程序完成可以避免这个陷阱。

现在我们来修改上面的程序，等待外部进程完成：

```
public class ExitPitfall1 {
    public static void main(String[] args) {
        try
        {
            Runtime rt = Runtime.getRuntime();
            Process proc = rt.exec("javac");
            int exitcode = proc.waitFor();
            System.out.println("exit code: " + exitcode);
        }
        catch (Throwable t){
            t.printStackTrace();
        }
    }
}
```

不幸的是，程序仍然没有任何输出，程序被挂起，无法结束，为什么javac.exe进程无法完成呢？

为什么*Runtime.exec()* 挂起

JDK文档有如下描述：

该方法在某些本地平台上，特定的进程也许不能很好地工作，如本地的*windows*进程,*daemon*进程，*windows*上的*16*位或者*Dos*程序，或者脚本程序。由于创建的子进程没有自己的终端或者控制台，所有的标准*IO(stdin,stdout,stderr)*的输出将被通过三个流*(getOutputStream(), getInputStream(), getErrorStream())*重定向到父进程中，父进程通过这些流来进行*io*。由于某些本地平台对这三个流仅提供有限大小的缓冲区，对子进程输入流的写或者对输出流的读上的失败可能导致子进程被阻塞，甚至导致死锁。

JDK文档告诉了可能存在问题，但是没有告诉该如何做。

尽管Runtime.exec() 看起来相当简单,但这个API在用起来却特别容易犯错误。现在让我们按着JDK的指导来处理javac的输出。当运行javac时，如果没有任何参数，那么它将产生一系列的关于如何运行该程序的输出以及可用的输入参数。这些输出是通过stderr流进行输出的。在等待进程退出之前，你很容易写一个程序来耗尽该流。下面的程序，可以工作，但不是一个好的通用程序。更好的解决方案是清空标准输出流和错误输出流。最好的解决方案是同步清空这些流（后面将有介绍）。

```java
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;

public class ExitPitfall2 {
    public static void main(String args[])
    {
        try
        {
            Runtime runtime = Runtime.getRuntime();
            Process proc = runtime.exec("javac");
            InputStream stderr = proc.getErrorStream();
            InputStreamReader isr = new InputStreamReader(stderr);
            BufferedReader reader = new BufferedReader(isr);
            String line = null;
            while ( (line = reader.readLine()) != null)
                System.out.println(line);
            int exitcode = proc.waitFor();
            System.out.println("exit code: " + exitcode);
        } catch (Throwable t)
          {
            t.printStackTrace();
          }
    }
}
```

运行结果如下:

```
C:\work\sketch\Java\pitfall_exec>java -classpath bin ExitPitfall2
用法: javac <选项> <源文件>
其中，可能的选项包括:
  -g                           生成所有调试信息
  -g:none                      不生成任何调试信息
  -g:{lines,vars,source}       只生成某些调试信息
  -nowarn                      不生成任何警告
  -verbose                     输出有关编译器正在执行的操作的消息
```

```
    -deprecation              输出使用已过时的 API 的源位置
    -classpath <路径>         指定查找用户类文件的位置
    -cp <路径>                指定查找用户类文件的位置
    -sourcepath <路径>        指定查找输入源文件的位置
    -bootclasspath <路径>     覆盖引导类文件的位置
    -extdirs <目录>           覆盖安装的扩展目录的位置
    -endorseddirs <目录>      覆盖签名的标准路径的位置
    -d <目录>                 指定存放生成的类文件的位置
    -encoding <编码>          指定源文件使用的字符编码
    -source <版本>            提供与指定版本的源兼容性
    -target <版本>            生成特定 VM 版本的类文件
    -version                  版本信息
    -help                     输出标准选项的提要
    -X                        输出非标准选项的提要
    -J<标志>                  直接将 <标志> 传递给运行时系统

exit code: 2
```

ExitPitfall2可以运行并且可以获得返回值2.正常情况下，返回值0意味着成功，其它非0值意味着发生了错误。因此为了规避第二个陷阱-如果你启动的程序会产生输出或者需要输入，确保你处理了输入和输出流。

假设一个命令在windows操作系统下可以运行，如dir,copy这种内部命令,许多新程序员会使用Runtime.exec() 后果是，他们落入了Runtime.exec的第三个陷阱。请见下面的例子：

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;

public class WinCommandPitfall {
    public static void main(String args[])
    {
        try
        {
            Runtime rt = Runtime.getRuntime();
            Process proc = rt.exec("dir");
            InputStream stdin = proc.getInputStream();
            InputStreamReader isr = new InputStreamReader(stdin);
            BufferedReader br = new BufferedReader(isr);
            String line = null;
            while ( (line = br.readLine()) != null)
                System.out.println(line);
            int exitcode = proc.waitFor();
            System.out.println("exit code: " + exitcode);
```

```
            } catch (Throwable t)
            {
                t.printStackTrace();
            }
        }
    }
```

运行结果为：

```
C:\work\sketch\Java\pitfall_exec>java -classpath bin WinCommandPitfall
java.io.IOException: CreateProcess: dir error=2
        at java.lang.ProcessImpl.create(Native Method)
        at java.lang.ProcessImpl.<init>(ProcessImpl.java:81)
        at java.lang.ProcessImpl.start(ProcessImpl.java:30)
        at java.lang.ProcessBuilder.start(ProcessBuilder.java:451)
        at java.lang.Runtime.exec(Runtime.java:591)
        at java.lang.Runtime.exec(Runtime.java:429)
        at java.lang.Runtime.exec(Runtime.java:326)
        at WinCommandPitfall.main(WinCommandPitfall.java:11)
```

返回值为2,意味着dir.exe文件未找到。这是因为dir是windows的内部解析命令，不是一个独立的可执行应用程序。运行windows下面的command内部命令，需要借助commond.com（windows95）或者com.exe(windows NT,2000,xp).例子代码如下：

```
import java.util.*;
import java.io.*;

class MyStreamThread extends Thread {
    InputStream is;
    String type;

    MyStreamThread(InputStream is, String type) {
        this.is = is;
        this.type = type;
    }

    public void run() {
        try {
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            String line = null;
            while ((line = br.readLine()) != null)
                System.out.println(type + ">" + line);
        } catch (IOException ioe) {
```

```
                ioe.printStackTrace();
            }
        }
    }
    public class WinCommandPitfall1 {
        public static void main(String args[]) {
            if (args.length < 1) {
                System.out.println("USAGE: java CommandPitfall1 <cmd>");
                System.exit(1);
            }

            try {
                String osName = System.getProperty("os.name");
                String[] cmd = new String[3];

                if (osName.equals("Windows NT") || osName.equals("Windows XP")) {
                    cmd[0] = "cmd.exe";
                    cmd[1] = "/C";
                    cmd[2] = args[0];
                } else if (osName.equals("Windows 95")) {
                    cmd[0] = "command.com";
                    cmd[1] = "/C";
                    cmd[2] = args[0];
                }

                Runtime rt = Runtime.getRuntime();
                System.out.println("Execing " + cmd[0] + " " + cmd[1] + " "
                        + cmd[2]);
                Process proc = rt.exec(cmd);
                MyStreamThread errorstream = new MyStreamThread(proc
                        .getErrorStream(), "ERR");
                MyStreamThread outputstream = new MyStreamThread(proc
                        .getInputStream(), "OUTPUT");

                errorstream.start();
                outputstream.start();

                int exitcode = proc.waitFor();
                System.out.println("exit code: " + exitcode);
            } catch (Throwable t) {
                t.printStackTrace();
            }
        }
```

```
}
```

运行GoodWindowsExec，输出如下:

```
C:\work\sketch\Java\pitfall_exec>java -classpath bin WinCommandPitfall1 "dir *"

Execing cmd.exe /C dir *
OUTPUT> 驱动器 C 中的卷没有标签。
OUTPUT> 卷的序列号是 84FE-C588
OUTPUT>
OUTPUT> C:\work\sketch\Java\pitfall_exec 的目录
OUTPUT>
OUTPUT>2007-12-31  09:48    <DIR>          .
OUTPUT>2007-12-31  09:48    <DIR>          ..
OUTPUT>2007-12-31  09:48               388 .project
OUTPUT>2007-12-31  09:48    <DIR>          src
OUTPUT>2007-12-31  09:48    <DIR>          bin
OUTPUT>2007-12-31  09:48               232 .classpath
OUTPUT>2007-12-31  09:51                33 run_ExitPitfall.bat
OUTPUT>2007-12-31  09:54                34 run_ExitPitfall1.bat
exit code: 0
OUTPUT>2007-12-31  09:58                34 run_ExitPitfall2.bat
OUTPUT>2007-12-31  10:14                45 复件 run_CommandPitfall1.bat
OUTPUT>2007-12-31  10:31                39 run_WinCommandPitfall.bat
OUTPUT>2007-12-31  10:31                48 run_WinCommandPitfall1.bat
OUTPUT>              15 个文件          1,107 字节
OUTPUT>               4 个目录  3,082,616,832 可用字节
```

使用关联文档类型运行将启动对应该文档类型的应用程序. 如启动word(.doc扩展名), 键入:

```
>java WinCommandPitfall1 "test.doc"
```

CommandPitfall1使用os.name系统属性决定操作系统的类型，然后决定合适的命令解析程序，使用MyStreamThread处理错误输出，标准输出。MyStreamThread清空独立线程传给它的任何流。

因此，为了避免Runtime.exe()的第三个陷阱，首先不能假设一个命令必然是可执行程序。要知道你所执行的命令是可执行程序还是一个内部命令。在下面的例子中，我们将做详细的介绍

getInputStream()用来获取进程的输出流，注意这里的InputStream是从Java的角度来看,而不是从外部程序的角度来看。外部程序的输出即是Java程序的输入。同样地，外部程序的输入流，从java角度来看，确是一个输出流。

*Runtime.exec() 不是命令行或者shell* Runtime.exec()最后一个陷阱是错误认为exec()可以接受你命令行或者shell中能接受的任何命令。Runtime.exec()能力是非常有限的，并且不能跨平台。Runtime.exec()的这个陷阱往往是被误解成可以接受作为命令行的任何字符串。

> ☞ 提示：
>
> 我们在命令行下输入一个命令时，往往包含命令的名称，命令的参数，以及相关的重定向控制等。因此我们常常认为把这样一个完整的命令行传给Runtime.exec(), Runtime.exec()也能按照我们的设想去做事，这个是大错而特错的。

下面这个例子介绍了一个使用exec()重定向的例子：

```
import java.util.*;
import java.io.*;

class MyStreamThread extends Thread {
    InputStream is;
    String type;

    MyStreamThread(InputStream is, String type) {
        this.is = is;
        this.type = type;
    }

    public void run() {
        try {
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            String line = null;
            while ((line = br.readLine()) != null)
                System.out.println(type + ">" + line);
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}

public class WinRedirectPitfall {
    public static void main(String args[])
    {
        try
        {
            Runtime rt = Runtime.getRuntime();
            Process proc = rt.exec("netstat -an > a.txt");
```

```java

            MyStreamThread errorGobbler = new
            MyStreamThread(proc.getErrorStream(), "ERROR");

            MyStreamThread outputGobbler = new
            MyStreamThread(proc.getInputStream(), "OUTPUT");

            errorGobbler.start();
            outputGobbler.start();

            int exitcode = proc.waitFor();
            System.out.println("exit code: " + exitcode);
        } catch (Throwable t)
        {
            t.printStackTrace();
        }
    }
}

```

运行该例程:

```
C:\work\sketch\Java\pitfall_exec>java -classpath bin WinRedirectPitfall
ERROR>
ERROR>显示协议统计信息和当前 TCP/IP 网络连接。
ERROR>
ERROR>NETSTAT [-a] [-b] [-e] [-n] [-o] [-p proto] [-r] [-s] [-v] [interval]
ERROR>
ERROR>  -a            显示所有连接和监听端口。
ERROR>  -b            显示包含于创建每个连接或监听端口的
ERROR>                可执行组件。在某些情况下已知可执行组件
ERROR>                拥有多个独立组件，并且在这些情况下
ERROR>                包含于创建连接或监听端口的组件序列
ERROR>                被显示。这种情况下，可执行组件名
ERROR>                在底部的 [] 中，顶部是其调用的组件，
ERROR>                等等，直到 TCP/IP 部分。注意此选项
ERROR>                可能需要很长时间，如果没有足够权限
ERROR>                可能失败。
ERROR>  -e            显示以太网统计信息。此选项可以与 -s
ERROR>                选项组合使用。
exit code: 1
ERROR>  -n            以数字形式显示地址和端口号。
ERROR>  -o            显示与每个连接相关的所属进程 ID。
ERROR>  -p proto      显示 proto 指定的协议的连接; proto 可以是
```

```
ERROR>                  下列协议之一：TCP、UDP、TCPv6 或 UDPv6。
ERROR>                  如果与 -s 选项一起使用以显示按协议统计信息，proto 可以是下
列协议之一：
ERROR>                  IP、IPv6、ICMP、ICMPv6、TCP、TCPv6、UDP 或 UDPv6。
ERROR>  -r              显示路由表。
ERROR>  -s              显示按协议统计信息。默认地，显示 IP、
ERROR>                  IPv6、ICMP、ICMPv6、TCP、TCPv6、UDP 和 UDPv6 的统计信息;
ERROR>                  -p 选项用于指定默认情况的子集。
ERROR>  -v              与 -b 选项一起使用时将显示包含于
ERROR>                  为所有可执行组件创建连接或监听端口的
ERROR>                  组件。
ERROR>  interval        重新显示选定统计信息，每次显示之间
ERROR>                  暂停时间间隔(以秒计)。按 CTRL+C 停止重新
ERROR>                  显示统计信息。如果省略，netstat 显示当前
ERROR>                  配置信息(只显示一次)
```

程序WinRedirectPitfall一个简单的netstat程序的输出重定向到a.txt文件中，然而我们a.txt根本没有产生。其中jecho程序简单地把命令行参数值直接输出。然在这种方式行不通。因为这里把exec()当成了shell解析器，实际上它完全不是。exec()仅能执行单个可执行程序（程序或者脚本）。如果需要处理流，如重定向或者输入到其它程序中，必须用程序来做。

```java
import java.util.*; import java.io.*;
class MyStreamThreadWithRedirect extends Thread {
    InputStream is;
    String type;
    OutputStream os;

    MyStreamThreadWithRedirect(InputStream is, String type)
    {
        this(is, type, null);
    }

    MyStreamThreadWithRedirect(InputStream is, String type, OutputStream redirect)
    {
        this.is = is;
        this.type = type;
        this.os = redirect;
    }

    public void run()
    {
        try{
```

```
            PrintWriter pw = null;
            if (os != null)
                pw = new PrintWriter(os);

            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            String line=null;
            while ( (line = br.readLine()) != null){
                if (pw != null){
                    pw.println(line);
                }
            }
            if (pw != null)
                pw.flush();
        } catch (IOException ioe){
            ioe.printStackTrace();
        }
    }
}
import java.io.FileOutputStream;

public class WinRedirectPitfall1 {
    public static void main(String args[])
    {
        try{
            FileOutputStream fos = new FileOutputStream("a.txt");
            Runtime rt = Runtime.getRuntime();
            Process proc = rt.exec("netstat -an");
            MyStreamThreadWithRedirect errorGobbler = new
            MyStreamThreadWithRedirect(proc.getErrorStream(), "ERROR", fos);

            MyStreamThreadWithRedirect outputGobbler = new
            MyStreamThreadWithRedirect(proc.getInputStream(), "OUTPUT", fos);

            errorGobbler.start();
            outputGobbler.start();

            int exitcode = proc.waitFor();
            System.out.println("exit code: " + exitcode);
            fos.flush();
            fos.close();
        } catch (Throwable t){
```

```
                t.printStackTrace();
            }
        }
    }
```

运行WinRedirectPitfall1 产生如下输出:

```
C:\work\sketch\Java\pitfall_exec>java -classpath bin WinRedirectPitfall1
exit code: 0
```

运行WinRedirectPitfall1, a.txt被创建了，说明该程序执行成功。解决这个exec陷阱，主要是通过控制外部进程的标准输出流来控制重定向。创建一个独立的OutputStream, 接收外部进程的标准输出然后写到指定文件名的文件中，用这种方式完成外部进程重定向的功能。

既然Runtime.exec()参数是和操作系统相关的，不同的操作系统下，输入的命令也随之不同。在写代码之前，最好测试输入的参数是否合法，然后再确定参数的写法，下面提供了一个命令行工具用来检测这个有效性。

避免Runtime.exec()的陷阱总结如下：

1. 只有外部进程退出，才能获取到返回值。
2. 必须马上处理外部程序的输入，输出以及错误流
3. 必须使用Runtime.exec()执行程序（指可执行程序）
4. 不能像命令行一样使用Runtime.exec()

在复杂的场合下，可以使用JNI来完成类似的功能。使用C/C++的system() 函数完成外部进程的调用，而使用java来调用该C/C++编译而成的JNI动态库。这样就很少遇到各种各样的怪问题了。

## §6.2 JDK自带的几个Timer的适用场合

### §6.2.1 java.util.Timer

```
----------------Main_UtilTimer.java----------------
package MyPackage; import java.util.Timer;

public class Main_UtilTimer {
    public static void main(String[] args) {
        Timer timer = new Timer();

        MyTimerTask task1 = new MyTimerTask(1);
        MyTimerTask task2 = new MyTimerTask(2);
        MyTimerTask task3 = new MyTimerTask(3);
        timer.schedule(task1, 5000);
        timer.schedule(task2, 5000);
        timer.schedule(task3, 5000);
    }
}

----------------Main_UtilTimer.java----------------
package MyPackage; import java.util.TimerTask;

public class MyTimerTask extends TimerTask {

    int taskid;
    public MyTimerTask(int _taskid){
        taskid = _taskid;
    }
    public void run()
    {
        try{
            System.out.println("execute timer task:"
                + taskid + " at :" + System.currentTimeMillis());;
            Thread.sleep(5000);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

5秒之前打印的堆栈是:

```
Full thread dump Java HotSpot(TM) Client VM (1.5.0_13-b05 mixed mode, sharing):
```

```
"DestroyJavaVM" prio=6 tid=0x00035b78 nid=0xe68 waiting on condition

"Timer-0" prio=6 tid=0x00a861a8 nid=0xec0 in Object.wait()
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22c011e8> (a java.util.TaskQueue)
        at java.util.TimerThread.mainLoop(Unknown Source)
        - locked <0x22c011e8> (a java.util.TaskQueue)
        at java.util.TimerThread.run(Unknown Source)

"Low Memory Detector" daemon prio=6 tid=0x00a58a80 nid=0xe70 runnable

"CompilerThread0" daemon prio=10 tid=0x00a57660 nid=0xe88 waiting on condition

"Signal Dispatcher" daemon prio=10 tid=0x00a93830 nid=0xe8c waiting on condition

"Finalizer" daemon prio=8 tid=0x0003f7a8 nid=0xe7c in Object.wait()
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22bd0ad0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        - locked <0x22bd0ad0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        at java.lang.ref.Finalizer$FinalizerThread.run(Unknown Source)

"Reference Handler" daemon prio=10 tid=0x0003e328 nid=0xe04 in Object.wait()
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22bd09e0> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Unknown Source)
        at java.lang.ref.Reference$ReferenceHandler.run(Unknown Source)
        - locked <0x22bd09e0> (a java.lang.ref.Reference$Lock)

"VM Thread" prio=10 tid=0x00a49f28 nid=0xe5c runnable

"VM Periodic Task Thread" prio=10 tid=0x00a81cd8 nid=0xe00 waiting on condition
```

timer任务执行时打印的堆栈（即5秒之后）：

```
Full thread dump Java HotSpot(TM) Client VM (1.5.0_13-b05 mixed mode, sharing):

"DestroyJavaVM" prio=6 tid=0x00035b78 nid=0xe68 waiting on condition

"Timer-0" prio=6 tid=0x00a861a8 nid=0xec0 waiting on condition
        at java.lang.Thread.sleep(Native Method)
```

```
        at MyPackage.MyTimerTask.run(MyTimerTask.java:15)
        at java.util.TimerThread.mainLoop(Unknown Source)
        at java.util.TimerThread.run(Unknown Source)

"Low Memory Detector" daemon prio=6 tid=0x00a58a80 nid=0xe70 runnable

"CompilerThread0" daemon prio=10 tid=0x00a57660 nid=0xe88 waiting on condition

"Signal Dispatcher" daemon prio=10 tid=0x00a93830 nid=0xe8c waiting on condition

"Finalizer" daemon prio=8 tid=0x0003f7a8 nid=0xe7c in Object.wait()
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22bd0ad0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        - locked <0x22bd0ad0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        at java.lang.ref.Finalizer$FinalizerThread.run(Unknown Source)

"Reference Handler" daemon prio=10 tid=0x0003e328 nid=0xe04 in Object.wait()
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22bd09e0> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Unknown Source)
        at java.lang.ref.Reference$ReferenceHandler.run(Unknown Source)
        - locked <0x22bd09e0> (a java.lang.ref.Reference$Lock)
```

执行的结果为：

```
execute timer task:1 at :1197985233784
execute timer task:3 at :1197985248786
execute timer task:2 at :1197985263787
```

从第一个堆栈可以看出，定时器的调度线程为"Timer-0"。但触发用户的定时任务时，执行线程仍然是"Timer-0"，也就是说调度线程和执行线程是同一个。同时，从打印出的结果来看，三个定时任务之间是依次执行的，而第三个任务比第一个晚了十秒钟，但它俩本应该是同时执行的。

这个在某些要求精确的场合下会造成严重的问题。解决的办法是用户的任务代码中，启动另一个线程来执行。当然，当任务的数量很多时，这样可能会导致某个瞬间线程过多，超过了系统所允许的最大值。这时候，最好使用线程池来执行定时任务代码。这样线程池就可以把最大的线程数量给控制在一定的范围内，避免了线程超过系统的最大数量而导致的失败。这是一个很完美的解决方案。

比较完善的代码如下：

```
----------------Main_AsyncUtilTimer.java----------------
package MyPackage;
import java.util.Timer;
public class Main_AsyncUtilTimer {
    public static void main(String[] args) {
        Timer timer = new Timer();

        MyAsyncTimerTask task1 = new MyAsyncTimerTask(1);
        MyAsyncTimerTask task2 = new MyAsyncTimerTask(2);
        MyAsyncTimerTask task3 = new MyAsyncTimerTask(3);
        timer.schedule(task1, 5000);
        timer.schedule(task2, 5000);
        timer.schedule(task3, 5000);
    }
}
----------------MyAsyncTimerTask.java----------------
package MyPackage;
import java.util.TimerTask;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class MyAsyncTimerTask extends TimerTask {

    int taskid;
    public MyAsyncTimerTask(int _taskid){
        taskid = _taskid;
    }
    public void run()
    {
        int nTasks = 50;
        int tpSize = 100;
        ThreadPoolExecutor threadpool = new ThreadPoolExecutor(tpSize, tpSize,
          50000L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());

        MyTimerExecutorThread exethread = new MyTimerExecutorThread(taskid);

        exethread.start();
    }
}

----------------MyAsyncTimerTask.java----------------
```

```
package MyPackage;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

public class MyTimerExecutorThread extends Thread{
    int taskid;
    public MyTimerExecutorThread(int _taskid){
        taskid = _taskid;
    }
    public void run(){
        try{
            System.out.println("execute timer task:"
                + taskid + " at :" + System.currentTimeMillis());;
            Thread.sleep(15000);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

```
execute timer task:1 at :1197995590286
execute timer task:3 at :1197995590286
execute timer task:2 at :1197995590286
```

从结果可以看出来，三个定时任务同时触发，没有互相影响，这正式我们想要的结果。这种实现有如下优点：

- 定时任务之间无任何影响，保证了触发精度
- 由于使用了线程池，因此避免了大量任务同时触发导致的线程数量超过系统限制。

这个方案既保证了触发精度，要保证了系统的可靠性。

### §6.2.2 java.swing.Timer

java.swing.Timer是swing包中带的一个定时器。这个定时器与java.util.Timer的差别在于，swing的timer的任务执行，使用的是swing事件分发线程。定时器事件与键盘鼠标等事件使用的是同一个事件分发器和执行线程。同样的，如果使用了这个Timer实现，那么任务和键盘鼠标事件的处理是在同一个线程中进行的，定时任务之是一个一个来执行的，因此会影响定时任务触发的精度。

一般情况，用户的定时任务不要使用这个Timer实现。

## §6.3 池的合理设计

常见的池有对象池，线程池，连接池。

### §6.3.1 对象池

恰当地使用对象池化技术，可以有效地减少对象生成和初始化时的消耗，提高系统的运行效率。创建新的对象并初始化的操作，可能会消耗很多的时间。在这种对象的初始化工作包含了一些费时的操作（例如在sip电话场合，频繁地去分配一些大对象）的时候，尤其是这样。在需要大量生成这样的对象的时候，就可能会对性能造成一些不可忽略的影响。要缓解这个问题，除了选用更好的硬件和更棒的虚拟机以外，适当地采用一些能够减少对象创建次数的编码技巧，也是一种有效的对策。对象池化技术（Object Pooling）就是这方面的常用技巧。

对象池化的基本思路是：将用过的对象保存起来，等下一次需要这种对象的时候，再拿出来重复使用，从而在一定程度上减少频繁创建对象所造成的开销。用于充当保存对象的“容器”的对象，被称为“对象池”（Object Pool，或简称Pool）。对于没有状态的对象（例如String），在重复使用之前，无需进行任何处理；对于有状态的对象（例如StringBuffer），在重复使用之前，就需要把它们恢复到等同于刚刚生成时的状态。并非所有对象都适合拿来池化－－因为维护对象池也要造成一定开销。对生成时开销不大的对象进行池化，反而可能会出现“维护对象池的开销”大于“生成新对象的开销”，从而使性能降低的情况。但是对于生成时开销可观的对象，池化技术就是提高性能的有效策略了。

基本上，只在重复生成某种大对象的操作成为影响性能的关键因素的时候，才适合进行对象池化。对一些小对象，使用池化技术不能带来任何性能的提升，反而会导致一定的性能下降。如果进行池化所能带来的性能提高并不重要的话，还是不采用对象池化技术，以保持代码的简明，而使用更好的硬件和更棒的虚拟机来提高性能为佳。

**关于对象池的使用场景** 在早期的JVM版本中，对象的创建(new)和垃圾回收是非常耗时的操作，但目前的版本，它们的性能有了本质的提高，事实上，Java中的分配现在已经比C语言在中的malloc更快了。在HotSpot1.4.x以后的版本中，new Object的代码几乎只有十几个机器指令。针对"慢"的对象创建，很多系统使用了对象池。对象在系统启动的初始阶段就将对象大量地给创建出来并放入对象池中，以后使用该对象就不直接去new,而是从对象池中取一个，当不用的时候，就释放回对象池，这样通过手工管理对象的生命周期，看起来避免了JVM对对象的频繁创建和销毁。

对象池的使用会带来如下不便：

- 多线程场合，对象生命周期不清晰，当一个线程将一个对象回池后，另一个线程也许仍然持有该对象的引用，如果该线程继续访问该对象引用，那么势必造成混乱. 当然可以通过代理设计模式来避免这种问题，但这又引入了大量的synchronized操作，是整个代码变得更加复杂，并且直接影响了性能。

- 对象回池时，要确保所有成员变量被重新初始化（对象重置），这些初始化操作是必须的，否则后期被使用，容易误使用这些实际上没有经过初始化的数据。而这些操作也需要消耗大量的代码，并且容易漏掉，这种bug非常隐蔽,难以察觉和定位。

- 当线程分配新的对象时，需要线程内部非常细微的协调，因为分配运算通常使用线程本地的分配来消除对象堆中的大部分同步。但是使用对象池化计数，这些线程从池中请求对象，那么协调访问池的数据结构的同步就是必须的了，这便产生了阻塞的可能性，又因为锁的竞争产生的阻塞，其代价比直接分配的代价多上百倍，这个地方甚至会成为整个系统的瓶颈。

另外，使用了对象池还有另外一个副作用，正确设定池的大小在很多场合下是一个巨大的挑战，太小，池会失去效率，太大会对垃圾回收造成压力。只所以会对垃圾回收造成影响，首先我们从垃圾回收的过程说起，垃圾回收分为三个阶段：

**mark阶段** 即扫描对象，将是垃圾的对象标识出来。这个阶段是最耗时间的。

**sweep阶段** 即将垃圾对象进行回收

**compact阶段** 即将内存碎片重整连成片，以避免大对象分配失败。

如果使用对象池，由于系统中存在大量的对象，这大大增加了需要mark的对象，尽管几乎每次mark它们的状态都几乎不变。而垃圾回收的mark阶段是最花时间的。实际上，JVM分配和回收对象都是非常快的，但是为什么我们有时发现申请新对象比重用老对象性能更差呢？关键在于对象申请下来之后，需要初始化，如果初始化过程比较复杂，包括构造函数或者调用初始化函数，所以在高性能的场合[50]，对于普通的Java 对象我们尽量采用clone的方式进行初始化，而不是采用构造函数或者初始化函数去初始化，这样的话，采用新分配对象逻辑简单，性能又好。

因此，目前使用对象池化技术请慎重考虑，是否你真得能控制住这批经常脱缰的野马。

> ☞ 提示：
>
> 在现代的虚拟机中，分配对象通常比引入同步要"便宜"得多.

### §6.3.2 线程池

由于一个进程内的线程不是无限的资源。操作系统对每一个进程都有一个最大线程数量的限制。为了避免系统在高峰期达到了最大线程数量而导致的应用失败，引入了线程池的设计。通过引入线程池，系统获得了如下好处：

- 将系统最大线程的数量给控制住，避免因线程数量超过系统限制而导致的系统不稳定。
- 避免了频繁去new Thread这种耗时操作，因此对系统的性能有一定的价值。

一般系统的最大线程数限制在几百到几千个，依赖于不同的系统而不同。

### §6.3.3 连接池

数据库连接资源是有限的，一般是几百个到几千个左右。超过了这个阈值，数据库会拒绝连接的建立。为了避免系统在高峰期达到了最大连接数量而导致的应用失败，引入了连接线程池的设计。通过引入连接池，系统获得了如下好处：

[50] 这里所说的高性能场合是指实时性很高的场合，比如sip电话系统，对系统短暂的GC也无法忍受的场合

- 将单机数据库应用程序的最大连接的数量给控制住，避免因连接数量超过系统限制而导致的系统不稳定。
- 避免了频繁去创建连接这种耗时操作，因此对系统的性能有一定的价值。
- 在多个数据库客户端的情况下，可以针对每一个数据库客户端分配指定数量的连接，因此是数据库服务器的的总的连接数可以被控制住，使整个系统稳定运行。

当某一个时段访问量比较大时，使用的连接数达到了连接池的最大值，那么获取连接的线程将被暂时挂起，直到池中有可用的连接。同时，引入了连接池可以处理暂时的过高请求，而整个系统还是稳定的。如果不采用这个连接池技术，瞬间的峰值会导致大量的请求处理由于创建不了数据库连接而导致处理失败。

但连接池的设计需要严重关注连接失效的问题，如果连接池无法自动处理失效的问题，那么连接池将无效，导致系统瘫机。数据库连接一般预先将连接建立好，应用使用时直接从池中获取连接，使用完后将连接释放回池。但有时从池中拿的连接已经失效(即死连接)，即socket物理连接已经不存在，造成这种情况的原因有如下几个：

- 物理数据库和应用程序之间跨防火墙，防火墙自行将socket连接关闭，此时连接池中的数据库socket连接已经变成死连接。一般一个连接超过一定的时间，无数据流量，防火墙就会自行关闭物理socket连接，一般情况下，不要将二者跨防火墙。
- 当一个连接长期无请求时，连接被数据库自动关闭。如MySQL,oracle等通过配置，都有这种模式。
- 数据库被重起过，导致连接池中已创建的连接失效。
- 网络闪断而导致socket无效。

因此一个好的连接池设计，需要综合考虑连接的失效问题(即死连接检测的问题)，确保连接是激活有效的，才能是一个稳定可靠的连接池。

☞让MySQL连接长时间生效:
MySQL连接如果8小时未使用，再使用该连接进行数据库操作，会抛出如下异常：

```
com.mysql.jdbc.CommunicationsException: Communications link failure
due to underlying exception
```

如果是MySQL5以前的版本，需要修改连接池配置中的URL，添加autoReconnect=true
如果是MySQL5以后的版本，需要修改my.cnf(或者my.ini)文件，在[mysqld]后面添加(单位都是秒，记得必须都添加，否则不起作用，通过show variables查看wait_timeout的值)：wait_timeout = 172800 interactive-timeout = 172800

**池设计的陷阱** 特别对于连接池和对象池，必须采用委托的模式，避免外边的代码捣乱。否则就会出现典型对象池模型的“对象过早归还”现象。另外，对于对象生命周期不清晰的场合，

最好不要用对象池，否则会有大量的空指针。空指针本身并不可怕，可怕的是导致其它的重要"善后"代码没有执行，导致内存泄漏，连接泄漏等严重影响稳定性的问题，请参考幽灵代码一节。

> ☞ 提示：
>
> 只有在生命周期非常清晰的场合才适合使用对象池，而且对象仅限定在内部使用，不要暴露给模块之外。

## §6.4 JDK1.5线程池的陷阱

JDK1.5自带的线程池中有如下特点，开始new出最少数量的线程，当有新任务添加到该线程池中，会有一个线程去执行，如果当时没有可用的线程，那么这个任务就放在队列中待执行，如果新到的任务太多，导致积压的任务过多，多到任务队列都装不下的时候，就会创建出新的线程。这种实现放在在非实时的场合，可能不会导致太大的问题，但是在实时应用场合，如sip,如果一个任务在队列中等待时间过长（如超过500毫秒），会导致大量消息重发或者呼叫失败。在这种场合下，需要将线程池的最小线程数量指定大一些，避免线程不足导致的任务处理延迟而造成的超时。

## §6.5 Timer的使用陷阱

在Timer的用法过程中，每new一个Timer，就会产生一个线程。如果创建了过多的Timer就会导致线程耗尽。

```
Timer timer = new Timer();
timer.schedule(new Task(), 60 * 1000);
```

如上的代码运行后，打印堆栈会发现对应一个MyTask线程。

```
at com.MyTask$1.run()
at java.util.TimerThread.mainLoop(Timer.java:512)
at java.util.TimerThread.run(Timer.java:462)
```

# §7 JVM

本章介绍JVM命令行参数的使用[51]。其中JVM命令行参数分为三种：

- 标准的运行期参数
- -X扩展参数
- -XX扩展参数

## §7.1 java运行期参数

直接输入java命令行(请参考[14])，打印如下：

```
C:\Documents and Settings\Admin>java Usage: java [-options] class
[args...]
           (to execute a class)
   or  java [-options] -jar jarfile [args...]
           (to execute a jar file)

where options include:
    -client       to select the "client" VM
    -server       to select the "server" VM
    -hotspot      is a synonym for the "client" VM  [deprecated]
                  The default VM is client.

    -cp <class search path of directories and zip/jar files>
    -classpath <class search path of directories and zip/jar files>
                  A ; separated list of directories, JAR archives,
                  and ZIP archives to search for class files.
    -D<name>=<value>
                  set a system property
    -verbose[:class|gc|jni]
                  enable verbose output
    -version      print product version and exit
    -version:<value>
                  require the specified version to run
    -showversion  print product version and continue
    -jre-restrict-search | -jre-no-restrict-search
                  include/exclude user private JREs in the version search
    -? -help      print this help message
```

[51]本章以SUN的JDK为蓝本进行介绍

```
-X            print help on non-standard options
-ea[:<packagename>...|:<classname>]
-enableassertions[:<packagename>...|:<classname>]
              enable assertions
-da[:<packagename>...|:<classname>]
-disableassertions[:<packagename>...|:<classname>]
              disable assertions
-esa | -enablesystemassertions
              enable system assertions
-dsa | -disablesystemassertions
              disable system assertions
-agentlib:<libname>[=<options>]
              load native agent library <libname>, e.g. -agentlib:hprof
                see also, -agentlib:jdwp=help and -agentlib:hprof=help
-agentpath:<pathname>[=<options>]
              load native agent library by full pathname
-javaagent:<jarpath>[=<options>]
              load Java programming language agent, see java.lang.instrument
```

**-client** 选择client模式下的运行模式。

虚拟机为了满足不同场合下运行要求，提供了两种模式，一种是server模式，另一种是client模式。有的场合下，内存很大，对性能要求苛刻。有的场合下，内存很小，但对性能要求不高。如果内存比较大，对性能要求苛刻的场合，建议运行在server模式下，在这种模式下，虚拟机通过使用很大的内存来换取更高的性能。

如果内存比较小，但对性能要求不高的场合，建议运行在client模式下，在这种模式下，虚拟机使用有限的内存，来正常运行。这种方式适合于小内存，短期运行的程序，牺牲速度，换取内存。

从1.5以来，当应用程序启动时，如果用户没有在命令行中显式指明虚拟机运行的模式，但起动器会自动尝试检查该应用是运行在一个server类型的机器上，还是client类型的机器上，如果是server类型的机器，则自动使用server模式启动虚拟机。这主要是基于一个性能考虑。server模式尽管启动过程比client慢，但运行期性能却比client模式的运行期性能高。

> ☞ 注意：
>
> J2SE 5.0中, 只有两个CPU以上，2G内存以上的物理机器才会被看做是server类型的机器。但不同的操作系统下面，server类型的标准也有不同，详细请参考SUN的官方文档(请参考[14])。为了保险起见，尽量手工在命令行中指定虚拟机的运行模式。

**举例** java -cient -classpath classes MyClass

**-server** 选择client模式下的运行模式。

适合于大内存长期运行的程序，以更大的内存换取更快的速度. 详细请参考-client

**举例** java -server -classpath classes MyClass

**-hotspot** -hotspot和-client是相同的含义同

**举例** java -hotspot -classpath classes MyClass

**-cp**

**-classpath** 设置虚拟机运行的classpath.在该路径（或者jar,或者zip文件）下面搜索class文件。如果存在多个路径（或者jar,zip文件），在windows下面采用";"作为分隔符，在Unix下面采用":"作为分割符。在命令行中指定-classpath或者-cp可以覆盖CLASSPATH环境变量的设置。如果没指定classpath,缺省是当前目录。

**举例** java -classpath classes;lib/mylib.jar MyClass

**-D<name>=<value>** 定义运行期变量(或者系统属性)，功能与环境变量相同。

**举例** -DROOTPATH=c:\myprogram

**-verbose:class** Java虚拟机运行期间，打印将class的加载情况。如：

```
[Loaded sun.net.util.IPAddressUtil from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded java.util.regex.MatchResult from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded java.util.regex.Matcher from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded java.util.SubList from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded java.util.RandomAccessSubList from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded java.util.ListIterator from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded java.util.SubList$1 from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded java.util.AbstractList$ListItr from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded org.xml.sax.EntityResolver from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded org.xml.sax.DTDHandler from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded org.xml.sax.ContentHandler from D:\jdk1.5.0_13\jre\lib\rt.jar]
[Loaded org.xml.sax.ErrorHandler from D:\jdk1.5.0_13\jre\lib\rt.jar]
```

从上面的例子中可以看出该系统中加载了哪些库文件中的哪些类。根据这个信息可以对类的加载情况进行分析。实际的应用中，有如下作用：

- 查看哪个jar文件被使用了，当一个类替换了jar无效之后，可以根据这个信息判断自己期望的jar文件被使用了。特别是当系统中存在多个版本的jar包是时，这个信息特别有用。

- 检查Permsize的内存溢出原因。在某些动态修改产生类的情况，不恰当的编程会出现不断有类被加载而又不卸载的情况，最后造成perm内存耗尽。通过-verbose:gc可以检查出这类问题[52]。

**举例** java -classpath classes;lib/mylib.jar -verbose:class MyClass

**-verbose:gc** GC的输出解读

```
8190.813:[GC 164675K->251016K(1277056K), 0.0117749 secs]
8190.825:[Full GC 251016K->164654K(1277056K), 0.8142190 secs]
8191.644:[GC 164678K->251214K(1277248K), 0.0123627 secs]
8191.657:[Full GC 251214K->164661K(1277248K), 0.8135393 secs]
8192.478:[GC 164700K->251285K(1277376K), 0.0130357 secs]
8192.491:[Full GC 251285K->164670K(1277376K), 0.8118171 secs]
8193.311:[GC 164726K->251182K(1277568K), 0.0121369 secs]
8193.323:[Full GC 251182K->164644K(1277568K), 0.8186925 secs]
8194.156:[GC  164766K->251028K(1277760K), 0.0123415 secs]
8194.169:[Full GC 251028K->164660K(1277760K), 0.8144430 secs]
```

各项含义如下：

```
8194.169:[Full GC 251028K->164660K(1277760K), 0.8144430 secs]
     |      |        |         |        |           |
     |      |        |         |        |           +--本次垃圾回收消耗的时间
     |      |        |         |        +-------------当前整个堆内存的大小
     |      |        |         +---垃圾回收后内存对象占有的内存。
     |      |        +------------垃圾回收前内存对象占有的内存。
     |      +----------------------本次垃圾回收的类型(FULL GC,和普通GC)
     +----------------------------垃圾回收的时间点(第几秒)
```

其中完全垃圾回收(FULL GC)表示本次垃圾回收器对所有的垃圾对象进行了回收。既然有完全回收，那么就有不完全回收，平时大多数情况下垃圾回收器进行的是不完全垃圾回收，此时回收后的内存实际上还有部分垃圾在里面，因此从这个数据我们无法分析我们的程序中的对象到底占用了多少的内存。因此在实际问题的分析过程中，只有完全垃圾回收的行，才有分析价值，因为它真实地反映了Java对象真正占用的内存大小。

通过分析FULL GC信息，可以进行内存泄漏分析等。详细请见：请参考第 71页第 §3.3节

**举例** java -classpath classes:lib/mylib.jar -verbose:gc MyClass

**-verbose:jni** 打印详细的JNI本地接口的使用情况。

```
[Dynamic-linking native method java.lang.StrictMath.pow ... JNI]
[Dynamic-linking native method java.lang.Float.intBitsToFloat ... JNI]
```

[52]javassist[12]等可以动态修改类，这些可能会导致类重复被加载的情况。

```
[Dynamic-linking native method java.lang.Double.longBitsToDouble ... JNI]
[Dynamic-linking native method java.lang.Float.floatToIntBits ... JNI]
[Dynamic-linking native method java.lang.Double.doubleToLongBits ... JNI]
[Dynamic-linking native method java.lang.Object.registerNatives ... JNI]
[Registering JNI native method java.lang.Object.hashCode]
[Registering JNI native method java.lang.Object.wait]
[Registering JNI native method java.lang.Object.notify]
[Registering JNI native method java.lang.Object.notifyAll]
[Registering JNI native method java.lang.Object.clone]
```

**举例** java -classpath classes;lib/mylib.jar -verbose:jni MyClass

**-version** 打印当前的JDK版本。
如：

```
E:\apache-tomcat-5.5.25\bin>java -version
java version "1.5.0_13"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_13-b05)
Java HotSpot(TM) Client VM (build 1.5.0_13-b05, mixed mode, sharing)
```

**举例** java -version

**-version:<value>** 以指定版本的虚拟机来运行java程序。

**举例** java -version:1.4 -classpath classes;lib/mylib.jar MyClass

**-showversion** 打印产品的版本，并继续。

**举例** java -showversion -classpath classes;lib/mylib.jar MyClass

**-jre-restrict-search | -jre-no-restrict-search** 在版本搜索中包含/排除用户私有JRE

**举例** java -jre-restrict-search -classpath classes;lib/mylib.jar MyClass

**-? -help** 打印帮助信息。

**举例** java -help

**-X** 打印扩展（即-X参数）帮助

**举例** java -X

**-ea[:<packagename>...|:<classname>]**

**-enableassertions[:<packagename>...|:<classname>]** 激活断言，缺省情况为禁止。如果-enableassertions 或者-ea没有跟参数，则表示激活所有断言。如果跟随参数,则表示仅对指定的包或者类激活断言。如果跟随的参数是"...", 则表示只激活当前工作目录下面的未命名包的断言。如果不是以"..."为结尾，则表示，激活指定类中的断言。如果一个命令行中包含多个选项（即多个-ea或者-enableassertions）,则以加载这些类的顺序为准。例如：如果一个运行一个程序仅想激活com.wombat.fruitbat包中的断言(and any subpackages), 命令行如下:

```
java -ea:com.wombat.fruitbat... <Main Class>
```

-enableassertions 和-ea 可以应用于所有的class以及系统class（系统类没有classloader）.只有一个例外: 如果没有参数的情况下，这个开关对系统classloader无效，这样很容易激活所有的用户类的断言除了系统类。对于系统类激活断言，请使用-enablesystemassertions .

**举例** java -enableassertions -classpath classes;lib/mylib.jar MyClass

**-da[:<packagename>...|:<classname>]**

**-disableassertions[:<packagename>...|:<classname>]** 关闭断言。(缺省值也是关闭)

disableassertions 或者-da如果没有参数的话，表示关闭所有断言。如果跟随着参数，并且用"..."结尾，表示关闭参数中指定的包的断言，并且同时关闭这些包的子包中的断言。如果参数仅仅是"...", 表示只关闭当前工作目录下的无名包(unamed package)，如果只跟随参数但没有"..."结尾，表示只关闭指定参数中指定类的断言。如果运行一个程序，打开com.wombat.fruitbat包中的断言，但关闭类com.wombat.fruitbat.Brickbat 中的断言，命令行如下：

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat <Main Class>
```

-disableassertions或者-da开关适用于所有的类加载器，包括系统类（系统类无类加载器），但有一个例外：无参数的格式，开关对系统类无效，之所以这样设计，是因为这样很容易打开所有类的断言除了系统类，对系统类断言的控制，虚拟机提供了一个独立的命令行选项：-disablesystemassertions. 详细，请参考后面的介绍。

**举例** java -disablesystemassertions -classpath classes;lib/mylib.jar MyClass

**-esa | -enablesystemassertions** 激活所有系统类中的断言（即设置系统类的缺省断言状态为true）

**举例** java -enablesystemassertions -classpath classes;lib/mylib.jar MyClass

**-dsa | -disablesystemassertions** 禁止所有系统类中的断言（即设置系统类的缺省断言状态为false）

**举例** java -disablesystemassertions -classpath classes;lib/mylib.jar MyClass

**-agentlib:<libname>[=<options>]** 加载本地代理库，如：

- -agentlib:hprof
- -agentlib:jdwp=help
- -agentlib:hprof=help

什么是JVM Tool Interface?

JVM TI接口是开发和监控工具的可编程接口。它提供了检测虚拟机的状态的机制，同时也提供了控制虚拟机中运行程序的能力。JVM TI目的是提供一个VM接口，通过这个接口，监测工具可以完全了解JVM内部的状态，包括但不限于:CPU剖析，debug,监控，线程分析，代码覆盖分析等。但JVM TI 不是在所有厂商的Java虚拟机中都是可用的。JVM TI是一个双向的接口，JVM TI的client端，这里称作是agent,当感兴趣的事件在JVM发生时,agent可以以事件的方式被JVM通知到。同时JVM TI可以通过接口或者事件的方式主动查询或者控制JVM应用程序。

代理和虚拟机运行在同一个进程中，并和虚拟机直接通信。通信是通过本地接口进行的(JVM TI).本地进程内接口允许以最小的代价堆虚拟机进行最大的控制。典型地，代理一般相对比较小而精悍，他们可以被包含了大量功能的外部独立进程进行控制，这样的话，对java应用程序的干扰最小。下面用到的术语"命令行选项"表示在JavaVMInitArgs参数提供的选项，用在JNI调用JNI_CreateJavaVM函数中。下面两个命令行选项，确保虚拟机启动期间正确装载和运行代理。命令行选项除了指明动态库的名字之外，同时还包含启动期间传给代理的选项。

- -agentlib:<agent-lib-name>=<options> -agentlib: 后面跟的要加载的动态库的名字.动态库可以是指明全路径的，也可以放在动态库缺省路径下。典型地，<agent-lib-name> 被扩展为操作系统下特定的文件名. 在系统启动间<options>将会传给代理。例如：-agentlib:foo=opt1,opt2, 在windows下，虚拟机将尝试从系统PATH环境变量所指向的路径装载foo.dll，在unix下, 虚拟机将尝试从系统LD_LIBRARY_PATH环境变量所指向的路径装载libfoo.so.
- -agentpath:<path-to-agent>=<options> -agentpath:跟的路径表示要装载的动态库的绝对路径。<options>指明的选项在系统启动期间传给代理，例如：-agentpath:c:\myLibs\foo.dll =opt1,opt2 表示装载c:\myLibs\foo.dll.

启动期间代理动态库的Agent_OnLoad函数将会被调用到。

## §7.2 java -X扩展运行参数

```
C:\Documents and Settings\Admin>java -X
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
    -Xbootclasspath:<directories and zip/jar files separated by ;>
                      set search path for bootstrap classes and resources
    -Xbootclasspath/a:<directories and zip/jar files separated by ;>
                      append to end of bootstrap class path
```

```
    -Xbootclasspath/p:<directories and zip/jar files separated by ;>
                      prepend in front of bootstrap class path
    -Xnoclassgc       disable class garbage collection
    -Xincgc           enable incremental garbage collection
    -Xloggc:<file>    log GC status to a file with time stamps
    -Xbatch           disable background compilation
    -Xms<size>        set initial Java heap size
    -Xmx<size>        set maximum Java heap size
    -Xss<size>        set java thread stack size
    -Xprof            output cpu profiling data
    -Xfuture          enable strictest checks, anticipating future default
    -Xrs              reduce use of OS signals by Java/VM (see documentation)
    -Xcheck:jni       perform additional checks for JNI functions
    -Xshare:off       do not attempt to use shared class data
    -Xshare:auto      use shared class data if possible (default)
    -Xshare:on        require using shared class data, otherwise fail.

The -X options are non-standard and subject to change without
notice.
```

**-Xmixed** 混合模式执行(缺省值)。即解释模式和JIT混合执行（JIT请参考第 142页第 §7.3节）。

**举例** java -Xmixed -classpath classes;lib/mylib.jar MyClass

**-Xint** 解释模式执行。禁止将类中的方法编译成本地代码，所有的字节码以解析方式进行。在该模式下，Java HotSpot虚拟机可适配编译器在性能方面的优势将无法体现。

**举例** java -Xint -classpath classes;lib/mylib.jar MyClass

**-Xbootclasspath:<directories and zip/jar files separated by ;>** 将路径,jar,zip库加增加到启动class的搜索路径，以";"作为分隔符。这样做违反了Java 2 Runtime Environment binary code license.

**举例** java -Xbootclasspath:classes;lib/mylib.jar MyClass

**-Xbootclasspath/a:<directories and zip/jar files separated by ;>** 将路径,jar,zip库加增加到缺省的bootclasspath，以";"作为分隔符。

**举例** java -Xbootclasspath/a:classes;lib/mylib.jar MyClass

**-Xbootclasspath/p:<directories and zip/jar files separated by ;>** 将路径,jar,zip库加增加到缺省的bootclasspath的前面，以";"作为分隔符。注意，应用程序使用该命令的目的是为了覆盖rt.jar中的类，这样做违反了Java 2 Runtime Environment binary code license.

**举例** java -Xbootclasspath/p:classes;lib/mylib.jar MyClass

prepend in front of bootstrap class path

**-Xnoclassgc** 不进行class的垃圾收集

**举例** java -Xnoclassgc -classpath classes;lib/mylib.jar MyClass

**-Xincgc** 打开增量垃圾收集

增量垃圾收集开关在缺省情况下是关闭的。增量垃圾收集可以减少程序运行期偶发的长时间垃圾收集。增量垃圾收集器将在一定的时间与程序并发执行，在这段垃圾收集的时间内，对正在执行的程序有一定的性能影响。

**举例** java -Xincgc -classpath classes;lib/mylib.jar MyClass

**-Xloggc:<file>** 将GC信息打印到指定的文件中。与-verbose:gc类似，-Xloggc:<file>将GC信息中直接打印到一个文件中。如果两个都提供了，那么以-Xloggc:<file>为准

**举例** java -Xloggc:c:\mylog.txt -classpath classes;lib/mylib.jar MyClass

**-Xbatch** 关闭后台编译

正常情况下，JVM将以后台的方式编译class中的方法，一直按解析模式运行代码，直到后台编译完成。-Xbatch标记关闭后台编译，所有方法的编译作为前台任务完成，直到编译完成。

**举例** java -Xbatch -classpath classes;lib/mylib.jar MyClass

**-Xms<size>** 指明堆内存的初使大小。该值必须是1024的倍数，并且大于1MB. 可以通过k或者M后缀表示是以KB字节为单位，m或者M表示以MB字节为单位。缺省值是2M,如：

```
-Xms6291456
-Xms6144k
-Xms6m
```

**举例** java -Xms512M -classpath classes;lib/mylib.jar MyClass

**-Xmx<size>** 指明最大的堆内存大小，该值必须是1204字节的倍数，k或者K表示KB, m或者M表示MB. 缺省值为64MB.

```
-Xmx83886080
-Xmx81920k
-Xmx80m
```

在Solaris 7和Solaris 8平台下面，该值的上限是4000M减去本地代码内存开销。在Solaris 2.6 和x86 平台下面, 该值的上限是2000m去本地代码内存开销。在Linux平台下面，该值的上限是2000m去本地代码内存开销。

**举例** java -Xmx512M -classpath classes;lib/mylib.jar MyClass

**-Xss<size>** 设置线程堆栈的大小。

**举例** java -Xss512K -classpath classes;lib/mylib.jar MyClass

**-Xprof** 打开CPU剖析功能。

剖析正在运行的程序，发送剖析数据到标准输出。这个选项作为程序开发期的一个有效工具，由于对性能的巨大影响因此不建议在生产环境下面使用。

-Xrunhprof[:help][:<suboption>=<value>,...]

打开CPU、堆、监视器（锁）的剖析。可以跟随一系列的"<suboption>=<value>"列表，元素之间使用,作为分隔符。运行java -Xrunhprof:help 可以获取子选项的列表和缺省值。

**举例** java -Xprof -classpath classes;lib/mylib.jar MyClass

**-Xfuture** 执行严格的类文件格式检查。考虑到后向兼容性，Java 2.x版本的JDK虚拟机的缺省类文件格式检查相比1.1.x版本的JDK要松一些。-Xfuture选项打开，将执行更严格的类文件格式检查，这样可以确保类文件与标准中定义的类文件格式更加顺从。鼓励开发者在新开发的代码中打开该选项，因为将来版本的java应用程序起动器缺省将启动严格的类文件格式检查。

**举例** java -Xfuture -classpath classes;lib/mylib.jar MyClass

**-Xrs** 减少由JVM使用的操作系统信号量。

为了确保Java应用程序的正确停止，JVM提供了shutdown钩子策略。In a previous release, the Shutdown Hooks facility was added to allow orderly shutdown of a Java application. 目的是为了确保系统退出之前，用户代码的善后代码可以得到执行(如关闭一个数据连接的)。SUN的虚拟机提供了对某些信号量的处理。

如果使用了-Xrs命令行选项，有如下两个后果：

1. SIGQUIT 线程堆栈打印不可用，即无法使用kill -3打印线程堆栈.
2. 已有的shutdown钩子函数调用的用户代码当虚拟机停止的时候将不会被自动调用，如当JVM将要停止的时候，手工调用System.exit()来确保原有的钩子函数被执行。

**用法**

**举例** java -Xrs -classpath classes;lib/mylib.jar MyClass

**-Xcheck:jni** 对JNI函数执行附加的检查特别地，虚拟机在处理JNI请求之前，会对传给JNI调用的参数进行校验，同时对运行期环境数据也进行校验。对于验证非法的数据，虚拟机将作为一个致命错误而终止。但该选项打开后，性能会有一定的下降。

**举例** java -Xcheck:jni -classpath classes;lib/mylib.jar MyClass

**-Xshare:off** 关于使用共享类数据。

**举例** java -Xshare:off -classpath classes;lib/mylib.jar MyClass

**-Xshare:auto** 在可能的情况下，使用共享的类数据

**举例** java -Xshare:off -classpath classes;lib/mylib.jar MyClass

**-Xshare:on** 一定使用共享的类数据，否则失败

**举例** java -Xshare:off -classpath classes;lib/mylib.jar MyClass

## §7.3 关于JIT

它允许实时地将Java解释型程序自动编译成本机机器语言，以使程序执行的速度更快。有些JVM包含JIT编译器。为了能做到平台无关，Java先把源代码编译成字节码，由JVM的interpreter轉转换成适合该平台的机器码。但解析运行很慢，于是把比较常用的bytecode先用JIT compiler处理，下次遇到相同的代码可以直接调用，以加快运行速度。

字节码解析很慢，JVM运行时，解析器必须读取字节码，译码，然后执行，循环往复，这个过程非常消耗事件。JIT就是为了解决这个解析执行的性能问题而引入了即时编译技术(Just-In-Time compilation)

在IBM JDK中，JIT不是JVM的组成部分，但它是一个Java运行环境的一个标准组件。

为了使得Java 的性能和可伸缩性尽可能的好，您应当使用最新可用版本的操作系统和Java，以及Just-In-Time（JIT）编译器。

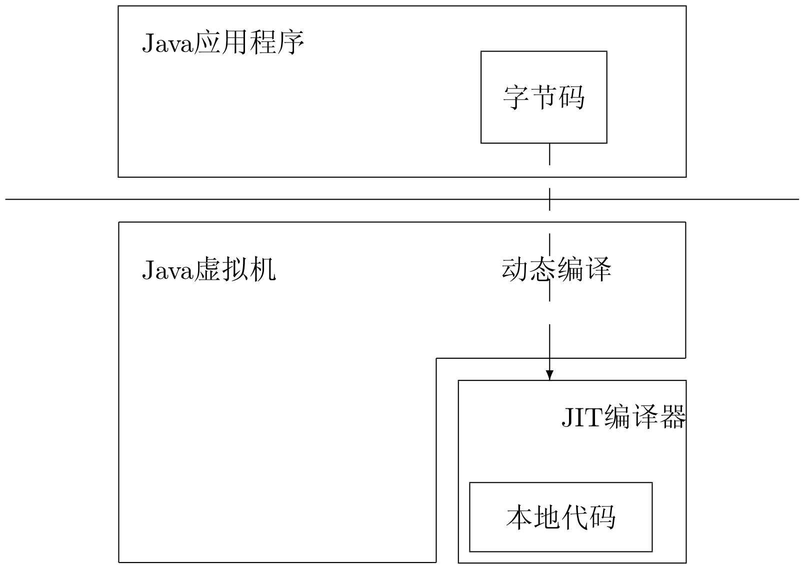



图 34 JIT

☞ 提示：

当JIT生效的时候，打印的线程堆栈可能看不到线程执行到的源代码类名和行号等信息。

如果一个方法被JIT编译器编译过之后，打印出的堆栈最后有"(Compiled Code)"字样，如下：

```
"Task-10" (TID:0x359B9600, sys_thread_t:0x3588E928, state:R, nativeID:0x0009105B) prio=5
    at org.apache.commons.el.Logger.logError(Logger.java:481)
    at org.apache.commons.el.Logger.logError(Logger.java(Inlined Compiled Code))
    at org.apache.commons.el.Logger.logError(Logger.java(Inlined Compiled Code))
    at org.apache.commons.el.ArraySuffix.evaluate(ArraySuffix.java(Compiled Code))
    at org.apache.commons.el.ComplexValue.evaluate(ComplexValue.java(Compiled Code))
                                                                                   |
                                           表示该方法被JIT编译器编译过<--------------+
    ... ...
    at com.ibm._jsp._more._jspx_meth_page_param_18(_more.java:11511)
    at com.ibm._jsp._more._jspService(_more.java:687)
    ... ...
    at com.ibm.ws.util.ThreadPool$Worker.run(ThreadPool.java(Compiled Code))
```

系统使用了JIT会带来性能的极大提升，极端情况下，可能会有上十倍的提升。

## §7.4 -Xrunhprof

我们都知道，OptimizeIt和JProfiler都可以进行内存和CPU进行剖析。如使用OptimizeIt 进行内存剖析的命令行如下：

```
-Xbootclasspath/p:"c:\OptimizeitEntSuite60\lib\bootcp\oibcp_1.4.2_06.jar"
```

但实际上SUN的虚拟机已经自带一套剖析工具:runhprof,下面介绍一下它的使用：

```
D:\>java -Xrunhprof:help

     HPROF: Heap and CPU Profiling Agent (JVMTI Demonstration Code)

hprof usage: java -agentlib:hprof=[help]|[<option>=<value>, ...]

Option Name and Value  Description                    Default
---------------------  -----------                    -------
heap=dump|sites|all    heap profiling                 all
cpu=samples|times|old  CPU usage                      off
monitor=y|n            monitor contention             n
format=a|b             text(txt) or binary output     a
file=<file>            write data to file             java.hprof[.txt]
net=<host>:<port>      send data over a socket        off
depth=<size>           stack trace depth              4
interval=<ms>          sample interval in ms          10
cutoff=<value>         output cutoff point            0.0001
lineno=y|n             line number in traces?         y
thread=y|n             thread in traces?              n
doe=y|n                dump on exit?                  y
msa=y|n                Solaris micro state accounting n
force=y|n              force output to <file>         y
verbose=y|n            print messages about dumps     y

Obsolete Options
----------------
gc_okay=y|n

Examples
--------
  - Get sample cpu information every 20 millisec, with a stack depth of 3:
      java -agentlib:hprof=cpu=samples,interval=20,depth=3 classname
  - Get heap usage information based on the allocation sites:
      java -agentlib:hprof=heap=sites classname
```

```
Notes
-----
  - The option format=b cannot be used with monitor=y.
  - The option format=b cannot be used with cpu=old|times.
  - Use of the -Xrunhprof interface can still be used, e.g.
       java -Xrunhprof:[help]|[<option>=<value>, ...]
    will behave exactly the same as:
       java -agentlib:hprof=[help]|[<option>=<value>, ...]

Warnings
--------
  - This is demonstration code for the JVMTI interface and use of BCI,
    it is not an official product or formal part of the J2SE.
  - The -Xrunhprof interface will be removed in a future release.
  - The option format=b is considered experimental, this format may change
    in a future release.
```

### §7.4.1 Java虚拟机运行期剖析接口介绍

Java虚拟机运行期剖析接口（Java Virtual Machine Profiler Interface）简称JVMPI,是介于java虚拟机和它的剖析器之间的双向接口函数，一方面，Java虚拟机在发生各种事件时会通知Java虚拟机剖析器代理，这些事件包括：

1. 堆栈分配
2. 线程的启动
3. 线程的停止
4. 方法调用
5. 对象的分配等

另一方面，Java虚拟机的剖析代理会通过JVMPI发送命令来请求更多的信息，例如如果剖析器的终端(如OptimizeIt或者JProfiler) 需要，Java虚拟机剖析器代理可以请求关闭或者打开指定事件的通知。

剖析器终端与剖析器代理可能会，也可能会不在同一个进程中，也可能时位于同一台机器的不同进程中，或者通过网络连接的不同机器上。

基于JVMPI的剖析工具能够获得到Java虚拟机运行时的很多信息，如：

1. 内存分配点(Heap memory allocation sites)
2. CPU使用的高频区(CPU usage hot-spots)
3. 没有回收的不必要的对象(unnecessary object retention)

4. 监视资源竞争(monitor contention)等

通过这些信息，我们可以对java进程进程全面的分析。

进行上面的信息分析，市场上有很多类似的分析工具，如JProfiler,OptimizeIt等，这些工具其实都是借助于JVMPI来实现的。 当然JDK自身带的剖析器代理输出同样可以做类似的分析。下面就对JDK自带的剖析器代理的使用做详细的介绍。实际上图形化的剖析工具看起来确实比较直观一些，但实际上只要掌握了原理，JDK自带的剖析器在分析问题的时候也非常容易[53]，特别在某些不方便或者商业化的剖析工具的场合下，使用JDK自带剖析器是非常方便的。

### §7.4.2 运行虚拟机期剖析器代理的原理及HProf代理的使用

运行：java -agentlib:hprof=heap=all myclass
java -agentlib:hprof=cpu=y myclass 当执行kill -3或者System.exit()时，相关信息就会被打印出来。

### §7.4.3 信息分析

**内存分析** 在产生的java.hprof.txt文件中有如下信息,这些信息是关于对象内存的信息，这些这些信息可以分析内存泄漏等问题：

```
          percent          live          alloc'ed  stack class
 rank   self  accum     bytes objs     bytes  objs trace name
    1  9.14%  9.14%    506768 7295    506768  7295 300000 char[]
    2  3.14% 12.27%    173960 7232    173960  7232 300000 java.lang.String
    3  2.85% 15.12%    157920 1974    228720  2859 300978 java.lang.reflect.Method
    4  2.80% 17.92%    155496 2815    177480  3245 303706 char[]
    5  2.74% 20.66%    151856   70   2326592   573 300077 byte[]
    6  2.31% 22.98%    128368 3139    274512  4968 303802 char[]
    7  1.48% 24.46%     82080   10    155952    19 301361 byte[]
    8  1.35% 25.80%     74808 3117    119232  4968 303801 java.lang.String
    9  1.30% 27.10%     71888  730     72160   732 302169 char[]
   10  1.22% 28.32%     67560 2815     77880  3245 303705 java.lang.String
   11  1.18% 29.50%     65552    1     65552     1 303607 byte[]
   12  0.97% 30.47%     53608    9     53608     9 311207 byte[]
   13  0.89% 31.36%     49368 2057    145992  6083 307496 java.lang.String
   14  0.89% 32.25%     49248    6     65664     8 313545 byte[]
   15  0.84% 33.09%     46864  730     47008   732 302167 char[]
   16  0.72% 33.81%     40000 1000    181520  4538 307502 java.util.TreeMap
   17  0.71% 34.53%     39600 1650     56136  2339 300166 java.util.ArrayList
   18  0.67% 35.19%     37008    9    193264    47 303605 char[]
   19  0.66% 35.85%     36680  655     36680   655 304042 java.lang.Object[]
   20  0.65% 36.50%     36080  451     36160   452 301479 java.util.HashMap$Entry[]
```

[53]命令行输出看起来比较恐怖，但实际上命令行信息非常集中

```
   21  0.59% 37.10%     32800    2     32800     2 312424 char[]
   22  0.58% 37.67%     32000 1000    132288  4134 307512 java.util.TreeMap$Entry
   23  0.58% 38.25%     32000 1000    145216  4538 307511 java.util.TreeMap$Entry

  .......
  416  0.02% 77.25%      1048    5      2136    10 313732 char[]
  417  0.02% 77.27%      1048    5      4272    20 312169 char[]
  418  0.02% 77.29%      1040    1      1040     1 312645 int[]
  419  0.02% 77.31%      1040    8      1040     8 309281 char[]
  420  0.02% 77.33%      1040    1      1040     1 307630 java.util.HashMap$Entry[]
  421  0.02% 77.34%      1040   65      1040    65 301962 java.util.HashSet
  422  0.02% 77.36%      1040   13      1040    13 301030 java.util.HashMap$Entry[]
   |     |      |          |     |         |      |   |      |
   |     |      |          |     |         |      |   |      +-类名
   |     |      |          |     |         |      |   +---分配点
   |     |      |          |     |         |      +-------已分配的对象数
   |     |      |          |     |         +--------------已分配的字节数
   |     |      |          |     +----------------------当前分配的对象数
   |     |      |          +----------------------------当前分配的字节数
   |     |      +---------------------------------------累计百分比
   |     +---------------------------------------当时占分配内存总数的百分比
   +---------------------------------------------名次


TRACE 307627:
    java.util.AbstractMap.<init>(AbstractMap.java:53)
    java.util.HashMap.<init>(HashMap.java:164)
    java.util.HashMap.<init>(HashMap.java:193)
    org.apache.tomcat.util.buf.StringCache.<clinit>(StringCache.java:64)
TRACE 307628:
    java.util.HashMap.<init>(HashMap.java:181)
    java.util.HashMap.<init>(HashMap.java:193)
    org.apache.tomcat.util.buf.StringCache.<clinit>(StringCache.java:64)
    org.apache.catalina.core.StandardServer.initialize(StandardServer.java:774)
TRACE 307629:
    java.util.AbstractMap.<init>(AbstractMap.java:53)
    java.util.HashMap.<init>(HashMap.java:164)
    java.util.HashMap.<init>(HashMap.java:193)
    org.apache.tomcat.util.buf.StringCache.<clinit>(StringCache.java:82)
TRACE 307630:
    java.util.HashMap.<init>(HashMap.java:181)
    java.util.HashMap.<init>(HashMap.java:193)
    org.apache.tomcat.util.buf.StringCache.<clinit>(StringCache.java:82)
```

```
        org.apache.catalina.core.StandardServer.initialize(StandardServer.java:774)
    TRACE 307631:
        org.apache.tomcat.util.buf.StringCache.<init>(StringCache.java:30)
        org.apache.catalina.core.StandardServer.initialize(StandardServer.java:774)
        org.apache.catalina.startup.Catalina.load(Catalina.java:504)
        org.apache.catalina.startup.Catalina.load(Catalina.java:524)
    TRACE 307632:
        java.lang.String.<init>(String.java:520)
        java.lang.String.substring(String.java:1770)
        org.apache.commons.modeler.Registry.findDescriptor(Registry.java:957)
        org.apache.commons.modeler.Registry.findManagedBean(Registry.java:666)
```

列名含义如下：

**rank** 根据已经分配内存多少进行排序所得的名次（将同一个线程堆栈分配的对象所占内存的大小进行排名）

**percent self** 特定方法为指定的类对象分配内存的大小所占分配内存总数的百分比（打印该hprof时刻，该对象所占用当时内存的百分比）

**percent accum** 截止到最后一次采样，self部分的累计值。（有些已经被回收，因此比percent self要大）

**live bytes** 当前分配的字节数(the number of bytes currently in use("live"))

**live objs** 当前分配的对象数(the number of objects currently in use("live"))

**alloc'ed bytes** 已经分配的字节数（the number of bytes originally allocated("allocéd")）

**alloc'ed objs** 已经分配的对象数（the number of objects originally allocated("allocéd")）

**trace** 代表内存分配动态上下文(即线程堆栈)中的Trace id

**name** 类名

```
420  0.02% 77.33%      1040    1      1040      1 307630 java.util.HashMap$Entry[]
```

表示：0.02%是由307630所指的方法来分配的，即该方法中分配的java.util.HashMap$Entry

对象大约占0.02%，而307630所指的动态上下文(即线程堆栈)是：

```
    TRACE 307630:
        java.util.HashMap.<init>(HashMap.java:181)
        java.util.HashMap.<init>(HashMap.java:193)
```

```
org.apache.tomcat.util.buf.StringCache.<clinit>(StringCache.java:82)
org.apache.catalina.core.StandardServer.initialize(StandardServer.java:774)
```

"TRACE 308086:" 表示对象分配点，即是由哪个方法分配的。这实际上是一个堆栈，表示该堆栈所指的代码流程分配了该对象。动态上下文的每一帧都包含类名，方法名，原代码文件名和行号，用户可以为HProf设置动态上下文的最大帧数，默认是4.动态上下文中不仅指明了哪一些方法进行了内存分配，而且还指明了哪些方法调用了这些方法进行了内存分配。

通过hprof输出进行内存泄漏分析(借助JProfile的分析请参考第 221页第 A节)的方法如下：

1. 从内存上下文中，找出占用内存特别多的对象
2. 从内存分配上下文中，看一下哪段代码分配了这块内存，清理出可疑的代码范围。
3. 结合对应的原代码，分析为什么内存没有释放

因此在手头没有OptimizeIt或者JProfiler等内存分析工具，借助hprof同样可以进行内存泄漏问题定位。

**CPU分析** 同时，hprof也提供了CPU剖析的能力，借助这些信息可以对热点函数进行分析，定位性能问题或者其它死锁等问题：

```
CPU  SAMPLES BEGIN (total = 703) Wed Nov 14 21:11:32 2007
rank   self accum    count trace method
   1 78.95% 78.95%     555 300525 java.net.PlainSocketImpl.socketAccept
   2  1.99% 80.94%      14 300098 java.lang.ClassLoader.defineClass1
   3  0.85% 81.79%       6 300369 java.io.FileOutputStream.writeBytes
   4  0.57% 82.36%       4 300573 java.lang.Shutdown.halt0
   5  0.43% 82.79%       3 300295 java.util.zip.Inflater.inflateBytes
   6  0.43% 83.21%       3 300336 java.lang.Class.getDeclaredConstructors0
   7  0.28% 83.50%       2 300406 java.io.WinNTFileSystem.getBooleanAttributes
   8  0.28% 83.78%       2 300423 java.lang.String$CaseInsensitiveComparator.compare
   9  0.28% 84.07%       2 300509 java.lang.Thread.start0
  10  0.28% 84.35%       2 300513 org.apache.tomcat.util.http.mapper.Mapper.addHost
  ......
  70  0.14% 92.89%       1 300558 java.net.PlainSocketImpl.socketBind
  71  0.14% 93.03%       1 300352 java.util.jar.Attributes.read
  72  0.14% 93.17%       1 300345 java.text.DateFormat$Field.<clinit>
  73  0.14% 93.31%       1 300338 org.apache.tomcat.util.digester.Digester.endDocument
  74  0.14% 93.46%       1 300337 java.io.Win32FileSystem.resolve
  75  0.14% 93.60%       1 300570 java.util.EventObject.<init>
  76  0.14% 93.74%       1 300335 java.lang.Throwable.fillInStackTrace
  77  0.14% 93.88%       1 300334 java.util.zip.InflaterInputStream.<init>
  78  0.14% 94.03%       1 300333 java.io.WinNTFileSystem.getBooleanAttributes
  79  0.14% 94.17%       1 300332 java.net.URLClassLoader$1.run
```

```
  80  0.14% 94.31%       1 300331 java.lang.Throwable.fillInStackTrace
  81  0.14% 94.45%       1 300330 org.util.IntrospectionUtils.setProperty
   |    |     |          |    |          |
   |    |     |          |    |          +--类名
   |    |     |          |    +-------------分配点，即动态上下文中的Trace id
   |    |     |          +------------------该堆栈采样过程中被命中的次数
   |    |     +-----------------------------累计占用的CPU百分比
   |    +-----------------------------------最后采样的时刻，所占用的CPU百分比
   +----------------------------------------排名

... ...
TRACE 300525:
    java.net.PlainSocketImpl.socketAccept(PlainSocketImpl.java:Unknown line)
    java.net.PlainSocketImpl.accept(PlainSocketImpl.java:384)
    java.net.ServerSocket.implAccept(ServerSocket.java:450)
    java.net.ServerSocket.accept(ServerSocket.java:421)
TRACE 300098:
    java.lang.ClassLoader.defineClass1(ClassLoader.java:Unknown line)
    java.lang.ClassLoader.defineClass(ClassLoader.java:620)
    java.security.SecureClassLoader.defineClass(SecureClassLoader.java:124)
    java.net.URLClassLoader.defineClass(URLClassLoader.java:260)
TRACE 300369:
    java.io.FileOutputStream.writeBytes(FileOutputStream.java:Unknown line)
    java.io.FileOutputStream.write(FileOutputStream.java:260)
    java.io.BufferedOutputStream.write(BufferedOutputStream.java:105)
    java.io.PrintStream.write(PrintStream.java:412)
TRACE 300573:
    java.lang.Shutdown.halt0(Shutdown.java:Unknown line)
    java.lang.Shutdown.halt(Shutdown.java:145)
    java.lang.Shutdown.exit(Shutdown.java:219)
    java.lang.Terminator$1.handle(Terminator.java:35)
TRACE 300295:
    java.util.zip.Inflater.inflateBytes(Inflater.java:Unknown line)
    java.util.zip.Inflater.inflate(Inflater.java:215)
    java.util.zip.InflaterInputStream.read(InflaterInputStream.java:128)
    sun.misc.Resource.getBytes(Resource.java:77)
TRACE 300336:
    java.lang.Class.getDeclaredConstructors0(Class.java:Unknown line)
    java.lang.Class.privateGetDeclaredConstructors(Class.java:2357)
    java.lang.Class.getConstructor0(Class.java:2671)
    java.lang.Class.newInstance0(Class.java:321)
```

列名含义如下：

**rank** 根据已经采用过程中，占用CPU的的排名（与命中次数是吻合的）

**self** 特定方法堆栈最后一次活动，所占用的CPU百分比。（一个堆栈可能结束后，又被调起）

**accum** 累计占用的CPU百分比。（该堆栈可能有多次活动，如果accum=self,意味这该堆栈一直处于活动状态）

**count** 该堆栈采样过程中被命中的次数

**trace** 代表动态上下文(即线程堆栈)中的Trace id

**thread** 线程堆栈的方法名（完整的堆栈从trace point对应的详细trace中可以找到）

HProf代理定期从当前所有运行现程中采样，记录堆栈活动信息，上面片断的count一列显示了每个线程堆栈在采样过程中被命中的次数，这些信息可以给出哪些线程使用CPU最频繁。被命中的次数越多，说明这个线程运行的时间越长。

通过hprof输出进行性能瓶颈分析，导致系统缓慢的原因很多（详见第 212页第 §13.14节），对于由于Java代码导致的系统缓慢，可以借助hprof的输出进行分析：

1. 首先从占用CPU时间最多的方法进行分析，执行该方法时间太长的原因可能如下：
   - 该方法是不是出现了死循环？
   - 方法是不是在长时间等待一个锁？
   - 本来该方法就应该长期运行，根本就不是一个问题。
2. 根据代码逻辑找到可疑的堆栈，进行分析

## §7.5 正确的视角看虚拟机

虚拟机实际上就是一个程序，当程序启动时，就开始执行保存在.class文件中的字节码指令。.class中的指令的执行是由虚拟机程序来执行的。更直观地说，.class相当于是脚本，java.exe是脚本执行程序。就像perl脚本执行程序，perl是可执行程序，perl脚本是脚本语言，perl可执行程序根据perl脚本指令运行。perl脚本与class文件的差别在于，perl脚本是可读的，而class文件是二进制不可读的。但二者的地位是一样的。理解了.class和虚拟机的关系, 我们就会知道在执行.class文件的时候究竟发生了什么事情，这样更有助于我们理解一些内在行为, class是JVM的运行脚本。把class看成是脚本更容易理解如下问题：

- 哪些操作会涉及本地代码的调用？
- 哪些操作会涉及本地内存的使用？
- Java线程和本地线程什么关系？

比如：

- 在Java中创建一个socket，触发了哪些系统调用?
- 我们都知道Java代码中可以调用本地代码接口(JNI)，即可以实现Java调用C++，那么在一个C++程序中是否可以调用Java代码?

# §8 关于字符集与编码

本节主要介绍如下内容：

- 什么叫字符集
- 什么叫编码
- 代码页
- locale
- 宽字节(wide char)与双字节(double byte char)
- DBCS与MBCS
- GBK与GB2312是什么关系
- unicode在计算机中占几个字符
- 将一个xml文件的编码从GB2312改为UTF,是否只需要修改声明？（encoding="utf-8"）

## §8.1 字符集

注意这里提到的是字符集而不是字符编码

字符集顾名思义就是字符的集合。这个集合中定义了每个字符的编码, 这里的编码就像每个员工给一个编号一样，跟计算机没有关系。unicode中称这个叫代码点。最初的ASCII字符集只包含128个字符（含一些控制字符）。常见的字符集有如下：

- ISO-8859-1 ISO-8859-16属于西欧语系
- GB2312,共收录6763个简体汉字
- GBK
- BIG5,台湾香港地区使用的繁体字
- GB18030
- Unicode,Unicode字符集试图囊括地球上所有的文字和符号，如：西欧文字，阿拉伯文字，中日韩，64卦等。其它字符集无法解决如下问题：
  - 在同一段文字中同时包含中日韩的文字
  - 采用Big5的字符集在以GB2312为内码的机器上进行正确显示

而Unicode由于囊括了所有的字符，因此上面两个问题在Unicode都能得到彻底的解决。

> ☞ 提示：
>
> 字符集只定义了每个符号对应的编号，这个编号与计算机没有任何关系。字符集不规定每个字符在计算机中用几个字节表示，这个属于编码（encoding）的范畴[a]。
>
> [a]字符集和编码是两码事，理解这一点是非常重要的，否则很容易被一些概念搞得晕头转向。

## §8.2 编码

编码是指一个字符在计算机中怎样存放，是采用一个字节，还是采用两个字节，还是采用不定长的字节？在介绍编码之前，我们先介绍几个名词：

- 单字节，即在计算机中用一个字节表示（字符集里面定义的）一个字符。如英文和欧洲语系
- 多字节(Multiple Byte Character Set,MBCS),即在计算机中用多个字节表示一个字符。
- 双字节（Double Byte Character Set,DBCS）,即在计算机中用两个字节表示一个字符。

GB2312,GBK,Big5一般既是字符集，又是编码。而unicode字符集是字符集，编码方式（即在计算机内表示）则有很多种.

- 最初采用两字节编码（ascii也采用两字节）
- UTF-32采用固定的4自字节编码。

- UTF-16和UCS-2相似，带有扩展机制，也是变长编码（2字节或者4字节）
- UTF-8采用变长编码(1-6字节不等)
- GB18030实际上是另一种Unicode的编码方式

关于UTF-8：

- UTF-8采用变长编码(1-6字节不等)
- UTF-8通过haffman编码区分后面有几个字节.
  优点:
  1. 对英文而言，仍然采用单字节编码，这样保证了后向的兼容性，对英文文档不存在转换的问题，原有的程序也没问题，同时比较节省空间
  2. 编码中不会出现0x00，这样在C/C++这种以0x00作为字符串结束的语言下，不会导致混乱
  3. 容易找到字符的边界，不容易出现字被截断的情况
  4. 逐字节编码，不存在大头/小头的问题(Big endian/Little endian)

  缺点：
  1. 判断字符数等需要从头开始遍历，效率较低，所以一般程序内部采用UCS-2定长字节表示（定长字符，字符串长度等运算非常容易），而保存在磁盘中则以UTF-8，这样可以保证空间的节省。

## §8.3 Unicode和UTF-8的关系

Unicode是字符集，而UTF-8则是Unicode字符集对应的计算机的存储格式。Unicode编码一般是指原生编码，尽管这里也叫编码。而UTF-8则特指计算机内部的存储编码。即Unicode是字符集名称，而UCS-2,UTF-8这些才是编码的名称。

**表 3 Unicode与UTF-8的映射关系**

| Unicode | UTF-8 |
|---|---|
| U-00000000 - U-0000007F: | 0xxxxxxx |
| U-00000080 - U-000007FF: | 110xxxxx 10xxxxxx |
| U-00000800 - U-0000FFFF: | 1110xxxx 10xxxxxx 10xxxxxx |
| U-00010000 - U-001FFFFF: | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx |
| U-00200000 - U-03FFFFFF: | 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx |
| U-04000000 - U-7FFFFFFF: | 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx |

注意xxx的位置由字符编码数的二进制表示的位填入. 第一个字节的开头"1"的数目就是整个串中字节的数目.

例如: Unicode 字符U+00A9 = 1010 1001 (版权符号) 在UTF-8 里的编码为:

11000010 10101001 = 0xC2 0xA9

而字符U+2260 = 0010 0010 0110 0000 (不等于) 编码为:

11100010 10001001 10100000 = 0xE2 0x89 0xA0

## §8.4 编码的识别

既然有那么多的编码，那么如何知道数据文件中，采用的是哪种编码方式呢？答案是没有准确的方法。在继续介绍数据文件和编码的关系之前，我们先做一个小试验：在windows xp下打开记事本程序，键入"联通"，并保存。然后再打开该文件，看看你看到了什么？

你应该没有看到"联通"两字，那这是为什么呢？

在windows xp下的记事本文件，在文件头部纪录了编码的类型，编码是FFFE,表示该文件是UTF-8的编码,而"联通"的ascii编码恰好是FFFE, 因此使得记事本程序误认为这是一个文件头。

实际上，具体数据文件中采用哪种编码方式，这个是依赖于各个系统自己的设计，没有通用的方法来判断，但在某些常用的通用的文件格式中，有如下约定：

**http** 使用Content-Type(HTTP header中或者meta tag)中进行表示

**e-mail** 使用Content-Type,Content-Transfer-Encoding进行表示

**BOM-Byte Order Mark** 在文件的头部增加一个标记来标明文件编码类型。

**00 00 FE EF：** UTF-32,big endian

**FF FE 00 00：** UTF-32,little endian

**FE FF：** UTF-16, big endian

**FF FE：** UTF-16, little endian

**EF BB BF：** UTF-8

xml解析器如何判断xml文件的编码？

1. 先检查文件头是否有BOM编码，有，则根据BOM编码确定编码
2. 没有BOM编码，则按ascii格式读取xml声明，根据xml声明中的encoding属性确定编码，如:<?xml version="1.0" encoding="UTF-8"?>
3. 如果二者都存在，且BOM声明和xml声明的不一致，或者声明的与文件中实际的数据不一致，那么均不能正确完成读取

## §8.5 关于编码的转换

- 编码是可以转换的，但必须两个字符集都存在的字符才可以相互转换。
- iconv -f UTF-8 -t GB2312 file.txt > file-gb.txt，将utf转换成gb2312

- C/C++中byte就是char，可以将多字节转换为宽字节，或者将宽字节转换为多字节。前面已经介绍过，由于UTF-8可以节省存储空间，因此在文件的持久化方面，采用得比较广泛。但由于UTF-8每个字符不定长度，因此在程序处理方面很不方便，如计算字符串的长度，搜索一个子串等都非常困难。因此在程序中一般采用的是固定字长的方式。因此多字节一般是指UTF-8编码方式，而宽字节一般是unicode的编码方式（在计算机内存中可能采用四个字节）。一般的处理过程是这样的:

  1. 从磁盘读入内存后，是UTF-8编码，即多字节方式。
  2. c/c++程序通过调用mbstowcs()将多字节转换为宽字节，即每个字符用unicode的四字节表示。(64位的机器可能是8字节)。具体4个字节或者8个字节对系统没有影响，关键看wchat_t的长度。
  3. 当需要写磁盘的时候，通过wcstombs()将宽字节转换为多字节，然后保存。

  在C++中是不直接支持各种字符集的，需要手工编写代码进行转换，在不同的操作系统下函数名称亦有不同。

  Linux下：

  - mbstowcs 多字节转换为宽字节，
  - wcstombs 宽字节转换为多字节

  windows下：

  - MultiByteToWideChar() 多字节转换为宽字节
  - WideCharToMultiByte 宽字节转换为多字节

- Java中，如果手工处理字符集的话，

  - String a = new String(bytes,"encodeing") 将bytes数组作为encoding的编码来处理
  - bytes=String.getBytes("encodeing") 将String中的字符串中按指定的编码方式转变为字节数组。

其它

- windows对编码的支持

  - windows2000只支持UCS-2
  - XP开始支持UTF-8

- 在运行程序时可以指定编码类型（程序根据该环境变量指定的编码类型进行数据的处理，当然前提时程序支持）。

  - windows下通过控制面板/区域和语言选项/非unicode程序的语言进行设置。
  - Unix/Linux下通过LC_ALL,LC_CTYPE,LANG等进行设置。

- 不能随机获取一个文件的一个字符，要结合上下文一起判断。

  native2ascii

# §9 常用的工具

本节介绍定位过程中用到的其它辅助工具。将这些使用工具结合起来，可以找到系统性能瓶颈，调试故障，判断出错的原因等。但是哪个工具适合手头的任务呢？关键在于需要的获得的信息以及与之匹配的工具。

## §9.1 远程调试

通过在Java启动的命令行中增加如下参数：
*-Xdebug -Xrunjdwp:transport=dt.socket,server=y,address=3333,suspend=n*,
可以启动远程Debug功能，进行远程调试程序。

## §9.2 Java自带工具

### §9.2.1 jconsole

请参考第 296页，第 I节。

### §9.2.2 jstack

打印Java进程的线程堆栈，详细请参考：http://blogs.sun.com/alanb/entry/jstack

## §9.3 Unix下的进行分析利器proc

/proc文件系统不是普通意义上的文件系统，它是一个到运行中进程进程地址空间的访问接口。通过/proc, 借助这些工具，可以对进程进行剖析，从而定位相关问题[54]。
proc 常用工具列表

- pstack
- pfiles
- pldd
- pmap
- ptree
- pwdx
- plimit

### §9.3.1 pstack

**用法：** pstack [java pid]

打印当前进程的每个线程的调用堆栈。

[54]windows下可以使用taskinfo工具进行类似的分析。

| | | | | |
|---|---|---|---|---|
| 操作系统 | solaris | linux | aix | HP |
| 命令 | pstack | lsstack/pstack | procstack | NA |

如在Linux下使用pstack：

```
$pstack 12323
Thread 2 (Thread 1084229968 (LWP 3369)):
#0  0x00000037dcacbd66 in poll () from /lib64/libc.so.6
#1  0x00002aaaab6c05e4 in wxapp_poll_func ()
#2  0x00000037e1e31fee in virtual thunk to IceDelegateM::Ice::Object:: ... ...
#3  0x00000037e1e324aa in g_main_loop_run () from /lib64/libglib-2.0.so.0
#4  0x00000037e635b0c3 in gtk_main () from /usr/lib64/libgtk-x11-2.0.so.0
#5  0x00002aaaab6d854d in wxEventLoop::Run ()
#6  0x00002aaaab767d4b in wxAppBase::MainLoop ()
#7  0x00002aaaaab40e4c in wxEntry ()
#8  0x0000000000428f50 in work ()
#9  0x00000037dd60baa3 in pthread_once () from /lib64/libpthread.so.0
#10 0x00002aaaad100992 in boost::call_once ()
#11 0x0000000000428f27 in mythread_proc ()
#12 0x000000000042728d in boost::detail::function:: ... ...
#13 0x00002aaaad101c9e in boost::function0<void, std::allocator ... ...
#14 0x00002aaaad101712 in virtual thunk to IceDelegateM::Ice::Object::ice_ids(...)
#15 0x00000037dd606407 in start_thread () from /lib64/libpthread.so.0
#16 0x00000037dcad4b0d in clone () from /lib64/libc.so.6
Thread 1 (Thread 46912546288176 (LWP 3368)):
#0  0x00000037dcacddf2 in select () from /lib64/libc.so.6
#1  0x00002aaaabfe4c4f in processEventsAndTimeouts ()
#2  0x00002aaaabfe58aa in glutMainLoop ()
#3  0x0000000000426cf5 in main ()
#0  0x00000037dcacddf2 in select () from /lib64/libc.so.6
```

pstack命令对诊断进程的挂起或者内存转储的状态非常有用。它默认显示进程中所有线程的堆栈情况，也可以作为一种原始的性能分析技巧，通过对进程堆栈的取样观察可以确定进程把主要时间花在了哪些部分。

### §9.3.2 pfiles

**用法：** pfiles [java pid]

列出该进程打开的所有文件和socket.如：

```
# pfiles 349 349: /usr/sbin/syslogd Current rlimit:65536
filedescriptors
0: S_IFDIR mode:0755 dev:102,3 ino:2 uid:0 gid:0 size:1536 O_RDONLY /
1: S_IFDIR mode:0755 dev:102,3 ino:2 uid:0 gid:0 size:1536 O_RDONLY /
```

```
2: S_IFDIR mode:0755 dev:102,3 ino:2 uid:0 gid:0 size:1536 O_RDONLY /
3: S_IFCHR mode:0000 dev:270,0
   ino:50368 uid:0 gid:0 rdev:41,53 O_RDWR /devices/pseudo/udp@0:udp
4: S_IFDOOR mode:0444 dev:279,0 ino:57 uid:0 gid:0 size:0
   O_RDONLY|O_LARGEFILE FD_CLOEXEC door to nscd[132]
   /var/run/name_service_door
5: S_IFCHR mode:0600 dev:270,0 ino:50855940 uid:0 gid:3 rdev:97,0
   O_WRONLY|O_APPEND|O_NOCTTY|O_LARGEFILE /devices/pseudo/sysmsg@0:sysmsg
6: S_IFREG mode:0644 dev:102,3 ino:3056 uid:0 gid:0 size:358
   O_WRONLY|O_APPEND|O_NOCTTY|O_LARGEFILE /var/adm/messages
7: S_IFREG mode:0644 dev:102,3 ino:2538 uid:0
   gid:3 size:8316 O_WRONLY|O_APPEND|O_NOCTTY|O_LARGEFILE /var/log/syslog
                                                              |
                                                              +-打开的是文件
8: S_IFSOCK mode:0666 dev:276,0 ino:8748 uid:0 gid:0 size:0 O_RDWR SOCK_STREAM
   SO_REUSEADDR,SO_KEEPALIVE,SO_SNDBUF(49152),SO_RCVBUF(49189),IP_NEXTHOP(37.192.0.0)
   sockname: AF_INET 192.168.127.3 port: 9103
       |                 |                |
       |                 |                |
       |                 |                +-----本地端口号
       |                 +----------------------本地IP
       +-----------------------------------------打开的是socket
   peername: AF_INET 192.168.127.4 port: 32863
                         |                |
                         |                +-----对端端口号
                         +----------------------对端IP
```

在不同的操作系统下，这个命令稍有不同：

| 操作系统 | solaris | linux | aix | HP |
|---|---|---|---|---|
| 命令名称 | pfiles | lsof | NA | NA |

有的时候打印不出文件名，如：

1018: S_IFREG mode:0644 dev:291,0 ino:335047 uid:3221 gid:102 size:2425
O_RDONLY|O_LARGEFILE 如果打开的文件已经被删除，那么就无法打印出文件名，此时可以借助文件结点通过如下方式找到文件名：
*find . -type f -exec ls -i   -print | grep 335047*

借助于该命令，找到泄漏的文件或者socket，就很容易缩小问题的范围了。如果是文件，根据文件的名字，如果是socket,就根据端口号，很容易确定哪个功能模块造成的句柄泄漏，然后检查相关的源代码，问题很快就能得到定位。

### §9.3.3 pldd

**用法：** pldd [pid]

列出本进程使用的动态库。如果程序中有JNI调用，那么可以使用该命令找到该进程到底调用了哪些动态库。

```
# pldd 1239
1239:   /usr/sbin/syslogd
/usr/lib/libnsl.so.1
/usr/lib/libpthread.so.1
/usr/lib/libdoor.so.1
/usr/lib/libc.so.1
```

### §9.3.4 pmap

**用法：** pmap [pid]

可以使用pmap命令来显示组成进程内存空间的各个内存映射。也可以使用pmap来查看进程占用物理内存的大小(即RSS)并收集有关进程使用内存的更多信息。因为进程通过共享库的使用和其它共享内存映射的方式与其它进程共享这些内存，我们可能会由于把同一共享内存统计多次而高估系统范围内的内存使用情况。要环节这种状况，把那些非共享的匿名内存的数量当作进程独有内存使用的一个估计数（即Anon列）. 进程的内存分为两类：

**虚拟内存** 指分配给进程的虚拟空间的数量

**物理内存** 也叫做驻留内存(RSS)，是指分配给进程的真实内存页面的数量。

```
#pmap   x 3368
3368:   ./sketchv
Address           Kbytes     RSS    Anon  Locked Mode   Mapping
0000000000400000     316       -       -       - r-x--  sketchv
000000000064f000      20       -       -       - rw---  sketchv
0000000000654000    3568       -       -       - rw---    [ anon ]
0000000040000000       4       -       -       - -----    [ anon ]
0000000040001000   10240       -       -       - rw---    [ anon ]
00000037db800000     108       -       -       - r-x--  ld-2.7.so
00000037dba1a000       4       -       -       - r----  ld-2.7.so
00000037dba1b000       4       -       -       - rw---  ld-2.7.so
... ...
00007ffff126c000      84       -       -       - rw---    [ stack ]
00007ffff13fe000       8       -       -       - r-x--    [ anon ]
ffffffffff600000       4       -       -       - r-x--    [ anon ]
----------------  ------  ------  ------  ------
total kB          421820       -       -       -
```

通过pmap -x 可以看到哪个动态库分配了多少内存，如果在一个JNI库中存在本地内存泄漏，那么借助pmap可以将范围缩小到一个动态库上。

### §9.3.5 ptree

**用法：** ptree [pid]

显示指定进程相关的血统关系，即进程的父子关系。

```
# ptree 1838
1933  /usr/dt/bin/dtlogin -daemon
  6359  /usr/dt/bin/dtlogin -daemon
    6380  /bin/ksh /usr/dt/bin/Xsession
6390       /usr/openwin/bin/fbconsole
… …
```

### §9.3.6 pwdx

**用法：** pwdx [pid]

显示指定进程的运行目录。

```
# pwdx 213
213:    /export/home/zmw/pp/bin
```

### §9.3.7 plimit

**用法：** plimit [pid]

显示指定进程的限制。

```
# pwdx 4100
4100: myprocc
   resource              current         maximum
  time(seconds)         unlimited       unlimited
  file(blocks)          unlimited       unlimited
  data(kbytes)          unlimited       unlimited
  stack(kbytes)         8192            unlimited
  coredump(blocks)      unlimited       unlimited
  nofiles(descriptors)  30000           30000
  vmemory(kbytes)       unlimited       unlimited
```

## §9.4 Unix下的进程统计工具prstat

进程统计实用工具(prstat)显示了一个正在实用系统资源的进程的概要信息，prstat按一定的时间间隔统计一次信息，并打印到屏幕上。

```
$ prstat
    PID  USERNAME   SIZE    RSS   STATE  PRI  NICE     TIME    CPU   PROCESS/NLWP
  24543  zmw       1323M   187M   cpu0     0    10  0:54:33  78.2%  java/23
  26555  zmw         23M     7M   cpu0     0    10  7:56:22   0.2%  ftpd/1
  14543  zmw        323M    37M   cpu0     0    10  2:45:73   0.1%  initd/1
  ... ...
Total: 91 processes,521 lwps,load averages: 39.06, 28.24, 6.68
```

默认地，prstat用一列显示每个进程的输出，每项根据CPU消耗量进行排序，各个列的含义如下：

**PID** 进程ID

**USERNAME** 进程的所有者名称（用户名）

**SIZE** 所有的映射虚拟内存大小，包括所有的映射文件及设备

**RSS** 驻留集合的大小，表示映射到进程的物理内存的总量，包括共享给其它进程的物理内存。进程的内存占用可以划分为两大类型：虚拟大小和驻留集大小。虚拟大小是指进程占用的虚拟内存的全部大小，即组成地址空间的单个映射的虚拟大小的总和，进程的虚拟内存的某些或者全部是在物理内存的，我们把这个大小称为进程的驻留集大小即RSS.

**STATE** 进程状态

**PRI** 进程优先级

**NICE** 用于优先级计算的精确数字

**TIME** 进程的累计执行时间

**CPU** CPU使用时间的百分比

**PROCESS/NLWP** 进程名以及进程的线程数

另外，prstat还提供了另一个重要的选项：-L，使用-L选项，prstat显示每行是一个线程，而不是一个进程。如：

```
$ prstat
    PID  USERNAME   SIZE    RSS   STATE  PRI  NICE     TIME    CPU   PROCESS/LWPID
  24543  zmw       1323M   187M   cpu0     0    10  0:54:33  40.2%  java/1
  26555  zmw       1321M     7M   cpu2     0    10  7:56:22   2.2%  java/2
  14543  zmw       1223M    37M   cpu3     0    10  2:45:73   0.8%  java/3
  ... ...
```

在这里只有最后一列和上面是不同的：

**PROCESS/LWPID** 进程名以及对应的轻型进程ID(即lwp)

这个选项再分析某些线程有帮助，详细请见 §1.3.3节第 31页.

## §9.5 Unix下的剖析工具truss/strace/dtrace/sotrace

**truss** solaris下跟踪本进程使用的操作系统调用和信号量. 如: truss -p 2343 -v all 1

**strace** Linux下跟踪本进程使用的操作系统调用和信号量.

**dtrace** solaris10下跟踪本进程使用的操作系统调用和信号量,功能更为强大.

**sotruss** solaris下跟踪共享库的系统调用，如：sotruss date 将列出date用到的所有动态库各自调用的系统调用

## §9.6 网络工具

### §9.6.1 路由跟踪命令traceroute/tracert

显示路由到目的地址所经过的路由器，可以诊断网络阻塞。例如：

```
C:\>tracert www.google.com
Tracing route to www-china.l.google.com [64.233.189.104]
over a maximum of 30 hops:

  1    <1 ms    <1 ms    <1 ms  192.168.0.1
  2     *        *        *     Request timed out.
  3    18 ms    18 ms    17 ms  58.60.17.93
  4    17 ms    18 ms    17 ms  58.60.24.97
  5    17 ms    18 ms    18 ms  58.60.24.53
  6    19 ms    17 ms    17 ms  202.97.64.18
```

## §9.7 swap交换分区管理

交换分区在Unix下一个非常重要，有的时候系统莫名其妙出错，交换分区是一个可疑点之一。 一般情况下，交换分区至少要保证要有4G. 在网上广为流传的，交换分区的大小是RAM的两倍，只是一个经验值，很多场合是不恰当的。比如RAM很大的情况下。

在Solaris下，可以通过如下的命令来增加交换分区：

```
mkfile 4000M /myswapfile   //创建一个4G的交换文件
swap -a myswapfile         //将该文件增加为交换文件
swap -l                    //列出所有的交换分区
swap -s                    //系统中总的交换分区大小
```

## §9.8 其它

| | |
|---|---|
| file core | 检查core文件是有哪个进程产生 |
| nm | 查看,obj,so等文件里面的符号表，如果是C++,可以使用nm a.so \| c++filt |

# §10 Java最佳实践

## §10.1 J2EE的潜在难点和最佳实践

- 如果你不了解或者没有经验，就有可能错误地使用，或者冒项目失败的风险。
- 一个糟糕的经验可能使你的系统瘫痪。
- 你很容易构建一个十分缓慢的J2EE应用。

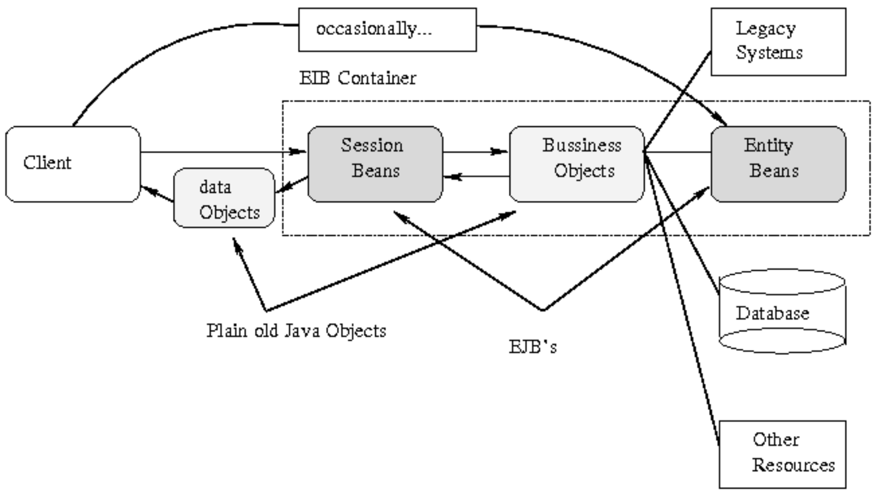



图 35 Session Bean Facade

### §10.1.1 架构上的问题

关于模型

- 表示与逻辑分开。见图36

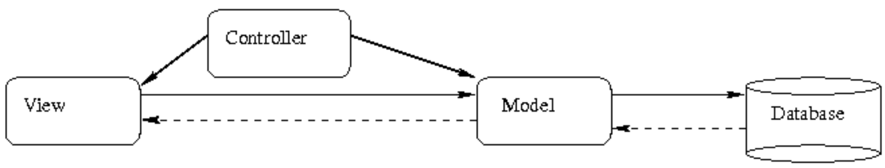



图 36 MVC

**View-表示** 显示输入输出数据（可以是HTML,可以是一个windows应用）

**Model-业务或者逻辑数据** 基于输入和定义的业务流程执行计算或其它操作

**Controller** 协调View和Model,在它们之间交换数据。

Controller有两个选择：

**以JSP为中心** 所有的请求直接发往下一个JSP,用以处理和输出

**以Servlet为中心** 所有的请求发送到servlet进行处理，输出到JSP

☞ 提示：

在复杂的情况下，应该以Servlet为中心，当简单应用可以以JSP为中心。典型情况不JSP中不宜有太多的Java代码。在业务处理之后最好让serverlet决定使用哪一个JSP.虽然可以使用JSP重定向，但是容易引起混乱。

- 两种典型的应用。

**基于单纯的Servlet** 见图37. HTTPServlet的子类近作servlet分内的工作（即管理request, response,及HttpSession对象）。将业务逻辑写在传统的Java类中。这种方式，可以带来测试，和重用的极大便利。

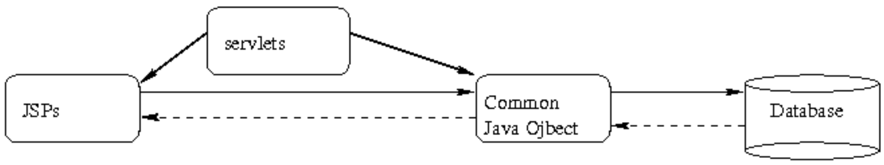



图 37 基于servlet的MVC

**基于Servlet和EJB** 见图38。 同样将将业务逻辑写在传统的Java类中。这种方式，可以带来测试，和重用的极大便利。

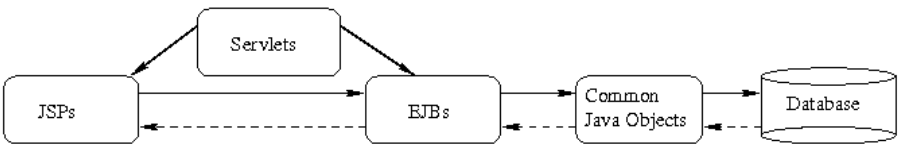



图 38 基于EJB的MVC

**关于EJB** EJB的类型：

- Entity Bean

  – 代表数据
  – 在服务器重启后仍然存在

- Session Bean
  – 执行动作
  – 等价于一般的JavaObject,只不过在远程
  – 可以是stateful原子操作，无上下文关系，也可以是stateless(在一个context中执行一系列的操作)

当从用户端存取一个Entity Bean:

- 每一个调用都是远程的，甚至get/set操作也是远程的.
- 如果不小心，从一个表中取一行有20个字段的纪录，可能有多大20次的数据库访问。如果将这些方法设置为readonly能避免这部分开销。

多数人的第一反应是将现有的每一个数据库表映射成一个EntityBean,该方法在某些场景下可行，但在大多数场景下不应该这样。特别当数据随着时间的推移逐步增长的场合尤其不合适。这会导致系统启动的时间延长至不可忍受的程度（长达几十分钟，甚至几个小时）。另外，在内存的需求方面也存在过分的要求。要记住，在不真正需要EJB的场合，使用EJB会带来相当的复杂性和时间开销。

直接映射数据库表的entity bean,不能提供一个有用的抽象，如果想直接和数据通信，为什么不用JDBC或者一个data bean来避免Ejb的开销？如，使用session bean,返回简单的java对象。分离出逻辑是EJB中设计的关键：

- Session Bean只做EJB分内的事情，也就是说使用session bean作为一个facade. 见图35
  – 将实际的业务逻辑放在传统的java类中，可以独立于EJB Container之外进行测试开发
  – 从这些类存取entity bean或者session bean能获得更好的性能。
    * 容器内的本地调用比远程调用快至少一个数量级
    * 所有对entity bean的调用可以在一个交易中完成。
- 面向方面的设计，将Ejb作为一层很薄的层
- 避免实体Bean的过分使用。

### §10.1.2 关于Servlet技巧

## §10.2 Java应用程序的基本准则

在编写Java应用程序时，以下是基本的准则：

- 当进行字符串连接的时候，使用StringBuffer，而不是使用String,以避免不必要大量临时对象，已经拷贝操作发生。
- 尽可能分组本地操作以减少Java 本地接口（JNI）的调用。
- 除非必要避免调用垃圾回收器。如果您必须调用它，只有在空闲时间或一些非关键阶段再这样做。
- 避免不必要的"cast"和"instanceof"引用，因为在Java 中销毁操作不是在编译时而是在运行时执行的。
- 当数组可以满足要求时尽可能避免使用vector。
- 使用缓冲区I/O 并调优缓冲区大小。
- 使用连接池和准备缓存声明进行数据库访问。
- 使用连接池连接到数据库并重用连接而不是重复打开和关闭连接。
- 最大化线程生存期并最小化线程创建和销毁次数。
- 最小化共享资源的争用。
- 最小化短生存期对象的创建。
- 尽量减少远程方法调用次数。
- 复杂系统下，尽量使用异步调用（即回调的方式）以避免阻塞远程方法调用。
- 尽可能保持同步方法处于循环外。
- 在数据库中以Unicode 形式存储字符串和字符数据。

## §10.3 消息系统的设计模型和关键点

使用消息分发模型是获得最佳性能的关键。

### §10.3.1 设计模型

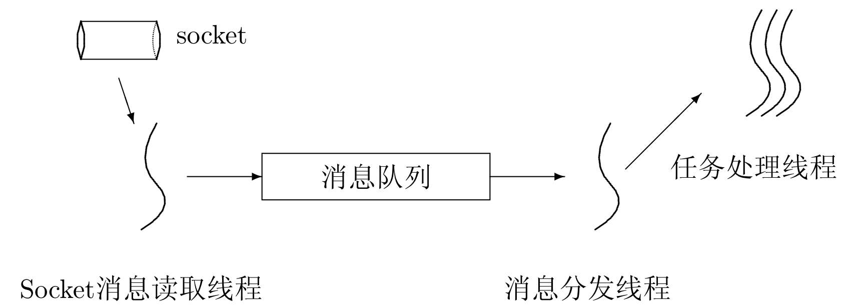



图 39 接收消息模型(NIO)

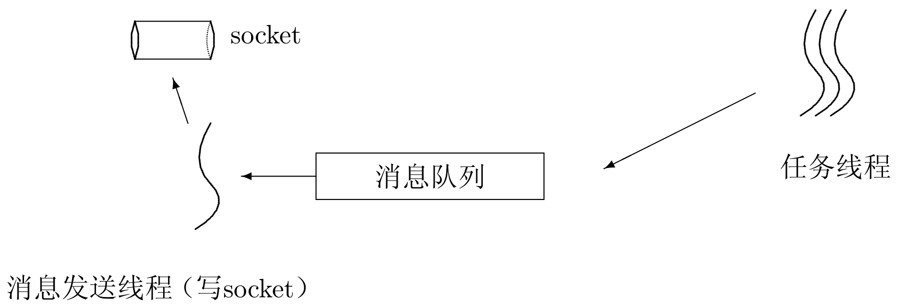



图 40 发送消息模型(NIO)

消息队列：

```
package com.example.queuemodel;

public class CircularQueue {
    /** Array of references to the objects being queued. */
```

```
    protected Object queue[] = null;

    /** The array index for the next object to be stored in the queue. */
    protected int sIndex;

    /** The array index for the next object to be removed from the queue. */
    protected int rIndex;

    /** Number of objects currently stored in the queue. */
    protected int count;

    /** The number of objects in the array (Queue size + 1). */
    protected int qSize;

    /**
     * Creates a circular queue of size s (s objects).
     * @param s The maximum number of elements to be queued.
     */
    public CircularQueue(int s) {
        qSize = s + 1;
        sIndex = 0;
        rIndex = qSize;
        count = 0;
        queue = new Object[qSize];
    }

    /**
     * Stores an object in the queue.
     *
     * @param x The object to be stored in the queue.
     * @return true if successful, false otherwise.
     * @exception ArrayIndexOutOfBoundsException
     */
    public boolean put(Object x) throws ArrayIndexOutOfBoundsException {

        synchronized(this)
        {
            if ((sIndex + 1 == rIndex) ||
                ((sIndex + 1 == qSize ) && (rIndex == 0))) {
```

```
                // queue is full
                return false;
            } else {
                // insert object into queue.
                queue[sIndex++] = x;
                count++;
                if (sIndex == qSize) {
                    // loop back
                    sIndex = 0;
                }
            }
            this.notify();
        }
        return true;
    }

    /**
     * Removes an object from the queue.
     *
     * @return a reference to the object being retrieved.
     * @exception ArrayIndexOutOfBoundsException
     */
    public Object get() throws ArrayIndexOutOfBoundsException {
        synchronized(this)
        {
            if (rIndex == qSize) {
                // loop back
                rIndex = 0;
            }
            if (rIndex == sIndex) {
                // queue is empty
                //return null;
                try{
                    this.wait();
                }
                catch(Exception e){
                    e.printStackTrace();
                }
                // return object
```

```
                count--;
                Object obj = queue[rIndex];
                queue[rIndex] = null;
                rIndex++;
                return obj;
            } else  {
                // return object
                count--;
                Object obj = queue[rIndex];
                queue[rIndex] = null;
                rIndex++;
                return obj;
            }
        }
    }

    /**
     * Returns the total number of objects stored in the queue.
     *
     * @return The total number of objects in the queue.
     */
    public int getCount() {
        return count;
    }

    /**
     * Checks to see if the queue is empty.
     *
     * @return true if queue is empty, false otherwise.
     */
    public boolean isEmpty() {
        return (count == 0 ? true : false);
    }
}
```

消息分发线程:

```

package com.example.queuemodel.receiver;

import java.util.concurrent.*; import java.util.concurrent.atomic.*;
```

```
import com.example.queuemodel.CircularQueue;
import com.example.queuemodel.MyMessage;
import com.example.queuemodel.TestTask;

public class ReceiverThread extends Thread{
    CircularQueue msgQueue;
    ThreadPoolExecutor threadpool;
    public ReceiverThread(CircularQueue _msgQueue,ThreadPoolExecutor _threadpool){
        msgQueue = _msgQueue;
        threadpool = _threadpool;
    }
    public void run(){
        while(true){
            MyMessage  msg = (MyMessage)msgQueue.get();
            TestTask task = new TestTask(msg);
            threadpool.execute(task);
        }
    }
}
```

消息发送线程：

```
package com.example.queuemodel.sender;
import com.example.queuemodel.CircularQueue;
import com.example.queuemodel.MyMessage;
public class SenderThread extends Thread{
    CircularQueue msgQueue;
    public SenderThread(CircularQueue _msgQueue){
        msgQueue = _msgQueue;
    }
    public void run(){
        while(true){
            MyMessage  msg = (MyMessage)msgQueue.get();
            System.out.println("11");

        }
    }
}
```

该消息模型可以带来最高的性能。 对于消息系统的其它关注因素请参考：第 169页 §10.3节。

### §10.3.2 其它设计关键点

- 一个消息的完整处理使用同一个线程做完，中间不要切换线程，便于问题定位和分析。以保证一个好的可调测性（方便打印日志，以及debug）。在sip 这种场合下，消息分发给一个线程还不够，应该先分发给一个call对应的内部队列，当内部队列中不为空的时候，则启动一个新的线程，否则该线程自动退出（回池或者消亡），以便避免过多的线程被浪费。如果内部队列有多个消息（任务），该该线程一直处理到完为止。FastInvoker有类似的功能。这里可以补充进来.

- 接收/发送消息使用消息分发机制，及任务队列的机制。

  1. 接收消息队列的消息分发线程，只负责将原始消息分发到另外的消息处理线程，一定不要进行耗时的parse操作。
  2. 对于有优先级区别的消息要分别使用不同的队列，避免使用同一个队列。如心跳消息的优先级要高，最好使用一个独立的队列，以避免在高峰时段，普通消息队列满而导致心跳这种高优先级的消息得不到及时处理，而影响稳定性。即对于有高优先级的消息，需要使用独立的队列进行处理，而不能和普通消息共用队列的方式，否则容易导致系统不稳定，甚至整个系统僵死。在消息系统的设计中，首先要检查整个系统的消息是否是对等的？如果有的消息优先级高，则应该为高优先级的消息使用专用的队列，以避免被普通消息耗尽队列导致整个系统僵死。如短消息系统中的login消息，这种消息至关重要，如果和普通的消息共用一个队列，由于对端异常（如重新启动，需要重新登陆），等待发送的消息填满整个发送队列，而login消息无法放入该发送队列，由于login无法发到对端，同时队列中的消息由于没有login，因此永远无法发送成功，从而导致系统永远没有机会继续运行。又如sip系统中的心跳消息，如果心跳消息得不到优先处理，当超过一定的时间后，可靠性系统会误判断，并作出一些有破坏性的操作，如清理整个会场等。反而是系统更加不稳定。如：

- 另外，消息队列要注意控制长度，因为消息系统中普遍有超时，当一个消息的响应在指定的时间内没有返回的话，那么对方一般会重发，当重发超过一定的次数，仍然没有返回的话，将停止再发。在这种机制下，我们要注意消息队列的长度。 新来的消息排在后面待处理，如果消息队列的长度过长，那么该消息要等待很长的时间才能被处理到，如果等待的时间超过了超时重发的总值，此时即使再处理该消息对方也不再理会。这样导致整个系统就死掉了，因为排在后面的每一个消息都会遇到这个情况。

- 进行消息分发的消息处理线程，要确保永不退出。 请参考第 208页第 §13.12节。

# §11 关于数据库

本节介绍数据库相关的注意事项。

## §11.1 关于数据库表死锁与锁表的问题

死锁与锁表有一些关联，又有相当大的不同。

**死锁** 死锁是由于两个事务互相等待被对方占有的锁，而导致的真正含义的死锁。这种死锁模式理论上说，是两个事务永远无法结束，但由于这种死锁数据库能够检测出来，与其二者都无法进行，还不如壮士断腕，让一个事务失败，这样另一个事务可以正常进行，失败的事务可以回滚，由应用程序决定下一步的动作，使系统不至于出现整个系统挂死的情况。

**锁表** 当一个事务在操作数据库（update等）的时候，在事务commit/rollback之前表的相关行是会被一直锁住的，但如果事务在某些原因下一直没有被提交，那么相关的表是一直被锁住的，而其它的事务由于该表的相关行被锁住，导致一直单向等待对方释放，这样导致所有的事务都无法进行，导致系统最终挂死。

### §11.1.1 关于表死锁

关于集群模式下数据库访问，如何保证时序（像工作流，如果保证两个机器上的工作项以一定的顺序执行）。事务提交缓慢比锁表更为严重。单机结合QueuedLock可以避免。

**表死锁的原因** 当两个或多个用户正在等待被对方锁定的数据的时候，死锁就会发生。死锁会导致两个事务不能继续运行。下图描述了两个事务死锁的场景：

在时间点0的时候，事务0要更新第0行，事务1要更新第1行，因此二者分别给自己要更新的行进行了锁行操作，即事务0锁住了第0行，事务1锁住了第1行。在时间点1二者又做了一些操作(此处不再赘叙). 在时间点2的时候，事务0企图更新第1行数据，由于此时第1行已经被事务1锁住，因此此时只能等待对方释放行锁。事务1同时企图更新第0行数据，由于此时第0行已经被事务0锁住，因此此时只能等待对方释放行锁。由于这两个事务互相要等待对方被对方占有的锁，自己才能继续，因此这就造成了死锁。二者永远没有机会继续运行下去。

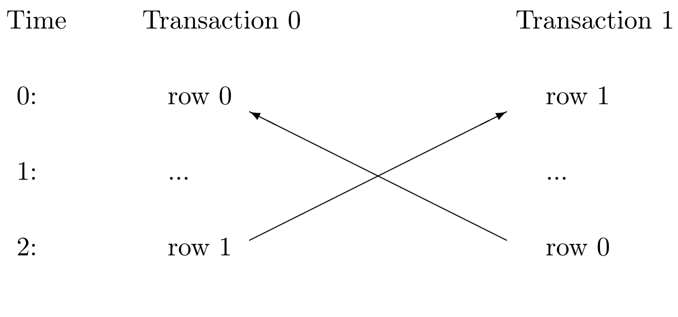



图 41 表死锁

**表死锁的检测** Oracle可以自动检测出死锁，并将死锁事务中的一个回滚，这样来释放一个发生冲突的行锁。在分布式事务下，这种死锁发生的机理是相同的。

**表死锁的避免** 多表死锁一般可以通过调整事务访问表的顺序进行避免。即以相同的顺序来访问表。或者手工来精确控制锁也可以。所有的开发者通过先锁主表，后锁详细表的方式更新数据，主表首先被锁住，然后锁住详细表，这样就避免的死锁。如果在整个应用中，这个原则能够很好的遵守的话，那么死锁就能避免。当你知道一个事务中需要一系列的锁的话，首先申请申请排他锁。

**手工加锁** 数据库一般会自动根据判断进行加锁以确保数据的完整性，以及语句级的读一致性。但也可以手工对锁进行控制。在下面几种情况下，需要使用手工来控制锁。

- 应用程序需要事务级的读一致性或者重复读。换句话说，在事务期间，查询必须得到一致的数据，而不能被其它事务干扰。通过手工使用显式的锁（或者只读事务、可序列化的事务）来保证事务级的读一致性。

- 应用程序需要对一个资源进行排他存取，从而不需要等待其它事务完成。

  在事务级，下面的sql语句可以实现加锁功能

- The SET TRANSACTION ISOLATION LEVEL statement

- The LOCK TABLE statement (which locks either a table or, when used with views, the underlying base tables)

- The SELECT ... FOR UPDATE statement

当这些事务提交或者回滚的时候，这些语句占有的锁，就会自动释放。一旦用户使用了事务级的手工加锁，要自己保证事务的完整性。

表一旦发生死锁，由于从逻辑上可以由数据库检测出来，因此为了避免事务永不结束锁带来的严重后果，数据库一般会自动让一个事务失败，而让另一个事务继续进行。因此表死锁不至于导致很严重的问题[55]，至多导致一个事务失败。如果程序对失败进行了处理，那么对系统可能不会造成任何影响。

### §11.1.2 关于锁表

死锁是由于两个事务在特定的关联行上锁表导致的。当一个事务在操作数据库（update等）的时候，在事务commit/rollback之前表的相关行是会被一直锁住的，当然在一定的时间内，表被锁住是保证事务一致性的正确和惟一手段。但如果事务在某些原因下一直没有被提交，那么相关的表是一直被锁住的，而其它的事务在继续进行之前只能等待对方释放，这样导致所有的事务都无法进行，导致系统最终挂死。更严重的是这种情况下，由于数据库无法判断事务不提交的真正原因，因此也就无法擅自做出事务失败回滚的操作，因此该问题一旦发生，对整个系统的影响是致命的，这个系统无法恢复。

[55]而线程的死锁往往是致命的，详见第 25页第 §1.3.1节

☞事务既没有提交也没有回滚，是锁表的唯一原因。如果事务由于某种代码缺陷导致的永远没有提交/回滚，那么这个锁表是永久的，直到java 进程退出，或者数据库连接被关闭。如果事务只是提交慢，那么在提交之前，表是一直锁住的，一旦提交，那么就被解锁，这往往是一个性能问题。

这种问题是系统真正严重关注的问题。要避免这种情况，惟一的办法是要保证所有的事务都要保证能够提交。特别是在如下的幽灵代码[56]：

1. 由于异常导致提交或者回滚代码没有得到执行，如：

```
transaction.start();
...... //这里其它的代码，如果抛出异常，
       //那么下面的事务提交或者回滚就无法执行到，
       //导致事务永远无法提交/回滚，从而导致永久的锁表。
transaction.commit(); 或者 transaction.rollback();

```

应该修改成如下代码

```
transaction.start();
try{
    ......          //其它可能抛出异常的代码
}
catch(Throwable t){ //注意抓住Throwable比Exception更安全
    ......          //抓住所有异常，确保后面的资源清理代码可以执行到
}
transaction.commit(); 或者 transaction.rollback();
```

或者

```
transaction.start();
try{
    ......          //其它可能抛出异常的代码
}
catch(){
    ......
}
finally{
  transaction.commit();  或者
  transaction.rollback();  //任何情况下保证事务的结束
}
```

[56]可怕的幽灵代码再次出现⊗,在这里一旦出现，那么对整个系统的影响将是致命的。

2. 由于不满足if/else条件等而导致提交或者回滚代码没有得到执行。

3. 事务代码非常分散，在不满足某些条件下，导致提交或者回滚代码没有得到执行。

4. 由于病灶转移带来的锁表假象（或者属于性能问题，只是时间提交地太慢了，那么从数据库来看，表是长时间被锁住）,请参考第 219页第 §13.24节介绍的案例。

这种代码导致的锁表问题，一般隐藏都很深，在实验室测试，这种问题不容易暴露。而这种问题一旦在生产环境发生，影响将是致命的。很不幸的是，这种问题通过前面介绍的定位手段几乎没有帮助，因为这种问题是由于应该执行的代码而没有执行，因此在系统中留不下任何痕迹，因此问题发生时，由于痕迹太少很难进行定位。写代码时进行注意是惟一比较有效的办法。另外，如果生产环境许可的话，可以从数据库侧查找是哪个SQL导致了表被锁，通过这个信息也能逐步找到嫌疑代码。但由于这种问题是致命的，因此生产环境发生问题，往往是马上进行重启，而没有时间留给我们定位的。

> ☞ 警告：
>
> 在真实的生产环境下，锁表会影响这个系统，锁表比死锁更为可怕。死锁只会导致一个事务失败，而锁表影响往往是全局而且致命的。

## §11.2 关于数据库SQL的性能

### §11.2.1 union语句

如下的SQL性能极差：

```
SELECT TASKID FROM TASK_DETAIL WHERE TASKID NOT IN (SELECT TASKID
FROM  TBL_PERSIST_CALLBACK UNION SELECT TASKID FROM WF_TBL_SYSTEM_TIMERS)
```

而修改成如下的语句，则性能非常好，性能提高几百倍：

```
 SELECT TASKID
  FROM TASK_DETAIL a
 WHERE NOT EXISTS
       (SELECT 1
          FROM WF_TBL_PERSIST_CALLBACK b
         WHERE a.taskid = b.taskid)
   AND NOT EXISTS
       (SELECT 1 FROM WF_TBL_SYSTEM_TIMERS c
         WHERE a.taskid = c.taskid)
```

通过topas pid查询sql

```
select sql_text
from v$session a, v$process b, v$sqltext c
where a.SQL_HASH_VALUE=c.HASH_VALUE and
```

```
        a.PADDR = b.ADDR and
        b.SPID = 2420858
order by c.piece
```

监控锁表的sql

```
select o.owner,
       o.object_name,
       o.object_type,
       o.last_ddl_time,
       o.status,
       l.session_id,
       l.oracle_username,
       l.locked_mode
from dba_objects o,gv$locked_object l
where o.object_id=l.object_id
```

性能监控的sql

```
select round(fetches/(executions+1)),
       sql_text,
       t.EXECUTIONS,
       t.VERSION_COUNT,
       t.CPU_TIME,
       t.ELAPSED_TIME,
       t.DISK_READS,
       fetches,
       hash_value,
       t.BUFFER_GETS
from v$sqlarea t
order by DESC

select sql_text
from V$sqltext
where hash_value=1118282882 order by pieces;
```

## §11.3 关于高性能场合下数据库的设计模式

两个大的原则:

- 尽量避免有历史积累效应的代码逻辑。即本次的运行尽量不依赖于之前的运行，这样的代码不容易出问题。

- 尽量避免将数据查到客户端进行遍历，尽量传回最少的数据。以workflow的定时器实现为例。

## §11.4 必须使用事务吗？

使用事务会保证数据的一致性。这是事务带来的最大价值。在数据要求绝对正确的场合，如银行，事务是不二选择。但事务不一定适合用在所有的场合。在数据完整性要求不是绝对高的情况下，不使用事务也许是一个更好的选择。使用事务会带来如下的负作用：

- 会带来一定的复杂性和可靠性的问题，如没有catch异常的话，导致rollback/commit没有提交，导致锁表，其它等待该锁的所有事务挂起。前面已经介绍过，这种由于事务没有提交/回滚而导致的锁表对系统的影响是致命的。会使整个系统停止工作。
- 在设计不合理的情况下，事务时间过长，会导致数据库锁竞争，导致整体性能急剧下降
- 完全依赖回滚处理出错情况，会使系统对容错处理自然而然降低注意，而过渡依赖于回滚，数据的一致性是保证了，但是用户操作会失败，用户操作频频受阻，用户感受很不好。正确的处理是找到为什么出错，而不是为了数据一致性而进行回滚。在绝大多数的情况下，是不应该出错的，出错情况下，绝大多数是由于代码的问题导致的，由于外部环境导致的出错往往是少数的。

## §11.5 确保Java代码不要依赖于数据库表字段的顺序

程序免不了进行升级，而基于数据库的程序，现场数据库的升级只能进行增量升级，遇到增加字段等，就在现场数据库上进行增加，而这种字段增加的位置很可能和开发环境不同，因此代码不要依赖于字段的顺序，要特别避免select * from tableA where .... 这种写法。

## §11.6 一种更简单的逻辑与数据分析-Named SQL

将SQL放在程序的外部，即配置文件中。Named SQL有如下优点：

- 通过更换SQL就可以实现多种跨数据库。(因为在现场select的接口是不会因为性能的调整该修改）

# §12 工程实践

本节介绍一些最佳的工程实践。

## §12.1 在高端机器上，一个JVM好还是多个JVM好？

在高端的机器上（很大的内存以及多核/多CPU），如何部署系统？单个进程还是多个进程更好一些？首先我们先分析一下单个进程和多个进程之间的差别。

单个进程的好处:

- 避免了多个进程带来的内存额外开销。

多进程的好处:

- 如果系统的线程模型设计不合理，无法充分利用CPU，那么多进程部署总的性能会高一些。
- 如果系统采用的是32位JDK，但跑在64位的操作系统上,并且系统物理内存足够大，因为32位的JDK最大的寻址空间为4G（实际能使用的地址空间2G左右）,因此即使机器有更多的物理内存，也不能充分利用，在这种情况下如果采用多进程会充分利用内存资源，更多地利用内存资源，使整体性能可能会更高。

总的来说，如果系统线程设计比较合理，能充分利用CPU，那么单个进程的性能不会比多个进程性能低，并且单个进程占用更少的系统资源。如果线程模型设计不合理，无法充分利用CPU,那么多个进程总的处理能力可能会更高。如果使用了32位的JDK ,但运行在64位的操作系统上，而恰好内存是系统的受限瓶颈，此时启动多个进程可以利用更多的内存，从而总的性能可能会更高。

## §12.2 关于Java进程监控-watchdog

watchdog又叫看门狗。负责监控程序是否正常运行。watchdog是系统在网上运行的最后一根救命稻草。它能避免你午夜12点被急促支持电话叫醒。如何让watchdog能真得检测出系统的问题，是watchdog中最重要的一部分。 下面着重介绍一下这一部分。

### §12.2.1 如何检测系统异常

传统的检测方式，往往检测进程是否还存在，这个在用C/C++的程序中用得比较多，但在Java下面，这种单纯监控Java进程是否存在的监控方式不是很恰当。这种方式只能检测出系统是否core dump。但在Java应用程序下面，系统不正常工作，往往有以下几个原因：

1. 系统core dump，进程异常退出。
2. 内存溢出，OutOfMemory，Java进程仍然存在
3. 资源泄漏导致无可用资源，如无数据库连接，达到了最大的文件句柄数，导致文件或socket无法创建

4. 系统线程被长期刮起（正在等待获取资源等），导致线程池无可用线程。

5. 其它未知的异常。

上面所提到的2,3,4三种场合，虽然系统不工作，但Java进程仍然存在，我们称这种情况为假死，这些异常也是watdog最应该关注的情况，只有能够检测出绝大多数的系统异常情况，watchdog才更有价值。

真正的系统异常要根据不同的场景来设计。最佳的设计是驱动真正的业务，这样可以真正的检测出系统是否异常。只有设计一个仿真的检测机制，才是最可靠的检测方法。

## §12.3 关于class Loader

class loader是一个负责加载类的对象。类ClassLoader是一个抽象类。每一个对象都有一个定义它的ClassLoader的引用。对于数组类的类对象，不是由类加载器创建的，而是在java运行期自动创建的。对一个数组类的类加载器，Class.getClassLoader() 返回的是它的基本类的类加载器。如果基本类型是原始类型，数组类则没有类加载器。

当搜索一个类或者资源的时候，类加载器采用的是委托模式。每一个ClassLoader都有一个相应的父类加载器。当类加载器要加载一个类时，会先请求其Parent加载(依次递归),如果在其父加载器树中都没有搜索到该类，则由当前类加载器加载。虚拟机内嵌的的类加载器, 叫做"bootstrap class loader",该类加载器不再有父类加载器。

由类加载器创建的对象的构造函数或者方法可能会引用到其它的类。这时也是从当前类的加载器的父类加载器开始搜索。

详细请参考[16]。

## §12.4 关于负载控制-动态过负荷还是静态过负荷?

在大容量的场合，进行负载控制是保证稳定性的一个重要手段。当系统当前的压力超过了指定的阈值，系统对部分请求会进行拒绝，以确保系统仍能正常工作，不至于因为过负荷而导致系统异常。系统当前的压力很容易根据当前的运行情况进行统计计算，但系统真正能承受的压力，如何去判断。一般有两种思路：

- 自适应的判断方法，来动态获得系统的能力数据，即动态过负荷
  - CPU的使用情况，如CPU的使用率达到80%,认为系统压力达到了最大的允许压力。
  - 消息队列的长度，如消息队列中消息堆积达到80%,就认为系统压力达到了最大的允许压力。
  - 线程池线程的使用情况，如空闲线程少于20%，就认为系统忙了。
- 使用固定的阈值作为系统的能力数据，即静态过负荷

从表面上看，自适应的判断方法优点是显而易见的的：可以根据机器的好坏，自动得出系统的能力。设置固定的阈值的方法缺点也是显而易见的的：无法对性能高低的机器进行自适应，好的机器可能由于不恰当的阈值设置，导致能力不能充分发挥。同时这种方式特别也会带来管理上的问题，如何确保安装人员能够根据机型进行正确的设置。

但实际情况确不是这样的。自适应依赖于一些外部参数判断系统当前的压力，在嵌入式硬件可能比较适用，因为在这种系统下，系统一般是独占的（应用程序自己独占），而操作系统任务调度往往也是比较单纯的，CPU的空闲基本反映了系统的忙闲。但在工作站或者服务器上，由于机器上运行的程序往往有多个，另外，系统可能也存在定时任务，会不定期启动，因此CPU使用率过高可能是其它外部程序导致的。另外，由于多线程设计不当，或者参数配置不当（如线程数量配置过小），往往CPU还没达到80%,系统已经无法正常工作。除非自己写的程序可以随着压力的加大可以达到100%的使用率，但这实际上是很难的。同样的，消息队列的长度仍然不能反映系统当前的空闲状态，除非瓶颈在消息队列上，消息队列的处理能力才能反映系统的能力。因为消息分发线程处理能力一般很快，会很快将消息产生一个任务，并扔给线程池，但最终线程中也有队列，实际上可能是在线程池中产生任务堆积而不是在消息队列中堆积。更为严重的是，如果消息队列中出现堆积，说明系统压力已经远远超过了自身的能力。线程池中有任务堆积亦然，正常情况线程池和消息队列中都不能产生堆积，一旦产生堆积，说明系统压力已经超过了系统的极限，此时检测出来，已经晚矣，因此根据消息队列和线程池队列都无法进行系统压力的判断。最为可靠的办法还是人为设定一个阈值，这个是非常安全可靠。虽然缺乏自适应能力，但在可靠性高的场合，可靠比方便更为重要。

## §12.5 机器设多个IP的原理？

关于计算机网络方面的知识领域非常广，但作为程序开发人员理解到如下的层面就能很好得理解其它东西了。

- 在网卡的层面，所有局域网内部都是广播的(与TCP/IP的广播不同，这里的广播实际上是电路级的广播，即最底层的向一个导线施加一个电压，那么Hub会同时施加到所有连接的网线上)，即当前局域网的所有机器都可以收到数据包，这也就是为什么在局域网的任何一个机器上都可以抓到整个网络上的包的原因。
- TCP/IP协议会忽略所有的不属于本机IP的包。因此，从外面看起来是点对点。
- 当一个机器有多个IP的时候，TCP/IP层会把所有属于本机IP的包都收下来，并通知应用层。这个就是多个IP的本质。

## §12.6 关于日志

### §12.6.1 关于java日志的几大恶劣设计

在电信级或者银行级的软件系统，在稳定性和可靠性上要求要苛刻得多，出现问题后要能够进行快速定位。这就依赖于一个好的日志系统。研发人员虽然对系统比较熟悉，对功能如何实现的也有一定的把握，但一个系统往往是庞大的，更为可能的是系统已经经过"几代人"接管，到最后，可能整个项目组内没有对整个系统的每一个角落都了如指掌，这时候，问题就出现了。对于研发人员来说，最擅长的就是通过问题现象去代码里进行分析，现场产品因为在运行，一般来说是不被允许使用调试工具直接在现场进行调试的。而问题只能通过日志进行分析。也就是说，日志对于问题的定位至关重要。系统日志设计的好坏会直接影响你解决问题的效率和质量。下面就是一些具有"坏口味"的日志实现：

1. 吞掉异常。

```
try{
    ... ...
}
catch(Exception e){
     //什么都没做
}
```

吞掉异常是最恶劣的代码习惯。如果发生问题，无人直到发生了什么问题。现场支持人员能做的惟一的事情是去猜到底系统发生了什么？

2. 吃掉原始异常，抛出另外一个自定义的异常。

```
try{
    ... ...
}
catch(Exception e){
    //吞掉原始异常，再抛出一个自定义异常
    MyException myE = new MyException();
    logger.log(LogLevel.ERROR,myE);
    throw myE;
}
```

原始异常最能反映问题的实际情况，里面的错误信息是最全的，包括发生问题的调用上下文，以及行号等。而自己再抛出另外一个异常，无疑是将这些最重要的信息给隐藏掉了，无端地给问题定位带来难度。

3. 多此一举的自定义错误码

```
try{
    ... ...
```

```
        }
        catch(Exception e){
            logger.log(LogLevel.ERROR,ERROR_CODE,"Error reason");
            return ERROR_CODE; //吞掉原生异常，直接返回错误码
        }
```

Java对于错误，缺省的情况下是通过异常来表达，包括Java自带库也是通过这种方式实现的。使用异常方式是Java代码的最佳选择，这样不但保证了整个系统的一致性，同时能保证原始的错误信息毫无遗漏地暴露在我们的面前。如果自己的系统使用错误码，不但多此一举，而且容易将最有用的信息给屏蔽掉，实在是无一点价值。如下面的异常，如果不直接打印出来，会将最有用的信息*unable to create new native thread*给漏掉，定位问题时还以为是普通的堆内存溢出。

```
Exception in thread "main"
java.lang.OutOfMemoryError: unable to create new native thread
        at java.lang.Thread.start0(Native Method)
        at java.lang.Thread.start(Thread.java:574)
        at TestThread.main(TestThread.java:34)
```

4. 不正确的日志级别

```
        try{
            ... ...
        }
        catch(Exception e){
            logger.log(LogLevel.DEBUG,e);   //日志级别为DEBUG
        }
```

在真实的生产环境，基于性能的考虑，一般日志的运行级别只会设置为ERROR/WARN,如果代码中将这种出错情况的日志级别设为DEBUG, 那么这种日志在现场根本不会打印出来。

### §12.6.2 什么是好的日志？

满足了如下条件就是好的日志：

1. 打印的是最原始的错误信息，没有经过任何转换
2. 给出了正确的日志级别，以保证在出错情况下，关心的日志能够真得打印出来
3. 在异常发生时，日志中有明确的调用上下文。

遇到异常，下面的日志打印就非常有效，既简单，又实用。

```
    try{
        ... ...
```

```
    }
    catch(Exception e){
        logger.log(LogLevel.DEBUG,e);   //日志级别为DEBUG
    }
```

## §12.7 异常处理的原则?

Java中的异常大致分成三类:

**JVM 异常** 这种类型的异常由JVM 抛出。OutOfMemoryError 就是JVM 异常的一个常见示例。对JVM 异常我们无能为力。它们表明一种致命的情况。 唯一的办法退出办法是停止应用程序服务器，然后重新启动系统。

**应用程序异常** 应用程序异常是一种定制异常，由应用程序或第三方的库抛出。这种异常往往是不满足某个应用条件，由应用抛出。

**系统异常** 在大多数情况下，系统异常由JVM 作为RuntimeException 的子类抛出。这种异常往往是编码错误，例如，NullPointerException 或ArrayOutOfBoundsException 将因代码中的错误而被抛出。另一种类型的系统异常在系统碰到配置不当的资源（例如，拼写错误的JNDI 查找（JNDI lookup））时发生。 在这种情况下，系统就将抛出系统异常。最重要的规则是，如果您对某个异常无能为力，那么它就是一个系统异常并且向上抛出。

以下是一些普遍接受的异常处理原则:

1. 如果无法处理某个异常，那就不要捕获它。
2. 如果捕获了一个异常，请不要胡乱处理它。
3. 尽量在靠近异常被抛出的地方捕获异常。
4. 在捕获异常的地方将它记录到日志中，除非您打算将它重新抛出。而不是把它吞掉。
5. 需要用几种类型的异常就用几种，尤其是对于应用程序异常
6. 如果系统使用了异常，那么就不要再使用错误码

## §12.8 基于限制的系统部署/设计

是什么决定了系统需要部署多少台机器？答案只有一个，系统的限制。部署/设计一个系统首先标识出，该系统的所有限制。根据这些限制去计算系统的能力。在设计期间，标识限制，是最影响设计的因素之一。

如果不知道系统的限制，那么也就无法正确评估系统到底有多少台机器。系统的限制就像系统的性能瓶颈，只能有一处。

在一个系统中，有如下限制:

- CPU计算量。如果CPU运算量不能满足要求，则需要增加新的机器。
- 内存。如果内存不能满足要求，可以通过增加内存或者机器来解决。
- 数据库的连接数。

### §12.9 String的值为什么不能改变？

我们都知道String里面的内容不能修改，但很多时候我们下意识的认为这是java对这个类做了什么特殊处理。其实不然，String类是一个普通类，虚拟机并没有对这个类进行区别对待，只所以有这个限制，是因为String类没有提供修改内容的接口。 如String + String 返回一个新的String,而不是修改原来的String. 其它方法也是这个道理，没有提供修改内容的方法，因此String里面的内容永远不会被改变。

### §12.10 系统出现问题需要收集的信息

当真实环境发生问题的时候，第一要务是恢复服务；然后在不破坏服务品质协议（SLA）的前提下，做一些力所能及的数据收集工作。更深层次的研究只能等灾难再次发生才能进行了。最重要的信息如下： 日志，堆栈，CPU信息，内存信息.

### §12.11 Web Failover集群的方案

Web集群一般前端放一个F5或者apache作为负载均衡器，负责到来的请求分发到后端不同的web服务器上。这样就实现了负载的均衡，并能容易地进行扩容。为了避免一个机器core dump导致用户当前操作失败，系统往往需要考虑failover(失败转移)。实现方式大约有如下两种：

**使用JGroup作为Session共享机制** 一旦一个用户登陆到系统中来，则将该Session信息复制到全局的JBossCache中，如果该机器core dump,那么负载均衡器将会将到来的请求自动发送到另一个机器上，由于另一台机器在内存中无该Session,因此可以从全局的JGroup中获取该session.这样用户就根本感觉不到后台是另一台机器再为他服务。这样就实现了失败转移的能力。但这种实现有如下缺点：

- 机器量很大的时候，网络流量会相当大。当机器多到一定的程度的时候，这个地方会成为整个系统的瓶颈，导致容量无法再扩大。

- JGroup的设计缺陷容易造成系统挂死。下面就是一个实际的例子，JGroup的对端一直无返回，挂死，导致本进程直接挂死。

```
"main" prio=1 tid=0x0805df38 nid=0x497b in Object.wait()
[0xbfffb000..0xbfffc808]
at java.lang.Object.wait(Native Method)
- waiting on <0x5ec84590> (a org.jgroups.util.Promise)
at java.lang.Object.wait(Object.java:474)
at org.jgroups.util.Promise.doWait(Promise.java:100)
at org.jgroups.util.Promise._getResultWithTimeout(Promise.java:52)
at org.jgroups.util.Promise.getResultWithTimeout(Promise.java:28)
- locked <0x5ec84590> (a org.jgroups.util.Promise)
at org.jgroups.util.Promise.getResult(Promise.java:77)
at org.jgroups.JChannel.connect(JChannel.java:423)
- locked <0x5ebf7030> (a org.jgroups.JChannel)
```

```
at org.jboss.cache.TreeCache.startService(TreeCache.java:1424)
at com.service.impl.GlobalCacheServiceImpl.afterPropertiesSet()
... ...
at org.springframework.web.context.ContextLoader.createWebApplicationContext()
at org.springframework.web.context.ContextLoader.initWebApplicationContext()
at com.container.control.ContainerContextLoaderListener.contextInitialized()
at org.apache.catalina.core.StandardContext.listenerStart()
at org.apache.catalina.core.StandardContext.start()
- locked <0x58f6ea98> (a org.apache.catalina.core.StandardContext)
at org.apache.catalina.core.ContainerBase.addChildInternal()
- locked <0x576943c8> (a java.util.HashMap)
at org.apache.catalina.core.ContainerBase.addChild()
at org.apache.catalina.core.StandardHost.addChild()
... ...
at org.apache.catalina.util.LifecycleSupport.fireLifecycleEvent()
at org.apache.catalina.core.ContainerBase.start()
- locked <0x58f6e2f0> (a org.apache.catalina.core.StandardHost)
at org.apache.catalina.core.StandardHost.start()
- locked <0x58f6e2f0> (a org.apache.catalina.core.StandardHost)
at org.apache.catalina.core.ContainerBase.start()
- locked <0x58fe1d98> (a org.apache.catalina.core.StandardEngine)
at org.apache.catalina.core.StandardEngine.start()
at org.apache.catalina.core.StandardService.start()
- locked <0x58fe1d98> (a org.apache.catalina.core.StandardEngine)
at org.apache.catalina.core.StandardServer.start()
- locked <0x5769de10> (a [Lorg.apache.catalina.Service;)
at org.apache.catalina.startup.Catalina.start()
at sun.reflect.NativeMethodAccessorImpl.invoke0()
at sun.reflect.NativeMethodAccessorImpl.invoke()
at sun.reflect.DelegatingMethodAccessorImpl.invoke()
at java.lang.reflect.Method.invoke()
at org.apache.catalina.startup.Bootstrap.start()
at org.apache.catalina.startup.Bootstrap.main()
```

- 系统复杂，当机器数量大时，问题定位的复杂度指数级增加。

**使用cookie携带token的方式实现跨机器共享** 每一个cookie携带自身的一些标识信息，如：用户名等标识用户的相关信息（如用户名或者Session的上下文信息等）。当前机器core dump时（或者所请求的业务放在其它节点），负载均衡器将该用户后续的请求转发到另一台机器上，但这台新接管的机器并无法找到该用户的Session信息。这时可以根据用户名和密码重建该用户的信息。这种方式实际上是将session信息放在IE端了，每次请求都携带该信息，走到哪里带到哪里。这样就解决了用户重复登陆的问题和跨节点的用户身份识别问题。但这种使用cookie携

带session信息有如下缺点：

- cookie携带的用户信息，那么一旦别人拿到这个cookie文件或者直到用户名自造一个cookie文件，可以直接冒名登陆。这会带来一定的安全问题。要避免这个问题，需要做一定的处理。即通过一个在cookie中携带一个经过加密的令牌，令牌中包含客户端的IP等信息，在server端处理请求的时候，检查它的合法性。从而确保该用户不是被冒名。
- 如果有Session的上下文信息需要保存，那么只能保存在cookie中，这样比较复杂。

## §12.12 关于可靠性设计

可靠性设计有如下两个特点：

- 如果你不能做到最好，还不如不做，因为它会在正常情况下会做坏事。
- 尽量地简单再简单，复杂地检测机制不但帮不了你，还会害了你。比如，检测系统的当前压力，很多人会想到通过检测下面两个参数作为系统是否过负荷的标记，但实际上是大错而特错的。
  - CPU
  - 队列长度

最佳的设计是通过配置，将最大负荷配置在配置文件中。

## §12.13 如何实现JVM Shutdown钩子函数？

通常JVM的关闭是由用户通过Unix上的kill命令，或者Windows上的<ctrl>+c信号启动的。从JDK1.3以来，应用程序可以使用java.lang.Runtime的addShutdownHook()方法安装自己的钩子函数，以确保当Java虚拟机退出时，做一些必要的业务清理操作。关闭钩子函数是已经初始化但是还没有启动的java.lang.Thread的示例。当JVM收到退出信号时，JVM将启动该钩子线程。

> ☞ 提示：
>
> 钩子函数不应该执行任何耗时的操作，而且应该是线程安全的，不应该依赖于其它任何服务，因为整个系统都在关闭自己的过程中，不能将自己的命运寄托于其它可能状态已经不正常的服务上。

在某些业务场合，当系统退出时，一定要执行一些清理工作，这时候钩子函数就是一个非常重要手段。

当然有的系统自己提供了关闭的功能实现，如通过一个socket或者其它手段向主java进程发送一个命令,当主java进程收到该命令后，调用System.exit()方法退出系统，可以在System.exit()之前执行相应的清理工作。

```java
public class Main {
    public static void main(String[] args) {
        Runtime.getRuntime().addShutdownHook(new Thread(){
            public void run(){
                MyJVMShutdownhook();
            }
        });

        try{
            Thread.sleep(10000000);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }

    public static void MyJVMShutdownhook()
    {
        System.out.println("Hello, this is my shutdown hook function");
    }
}
```

当按下<ctrl>+c键时，输出如下：

```
C:\work\sketch\Java\shutdownhook>java -classpath bin Main
```

```
Hello, this is my shutdown hook function
终止批处理操作吗(Y/N)? y
```

### §12.14 如何截取输出流？

在Java开发中，控制台输出仍是一个重要的工具，但默认的控制台输出有着各种各样的局限。本文介绍如何用Java管道流截取控制台输出，分析管道流应用中应该注意的问题，提供了截取Java程序和非Java程序控制台输出的实例对于使用javaw这个启动程序的开发者来说，控制台窗口尤其宝贵。因为用javaw启动java程序时，根本不会有控制台窗口出现。如果程序遇到了问题并抛出异常，根本无法查看Java运行时环境写入到System.out或System.err的调用堆栈跟踪信息。为了捕获堆栈信息，一些人采取了用try/catch()块封装main()的方式，但这种方式不一定总是有效，在Java运行时的某些时刻，一些描述性错误信息会在抛出异常之前被写入System.out和System.err；除非能够监测这两个控制台流，否则这些信息就无法看到。

因此，有些时候检查Java运行时环境（或第三方程序）写入到控制台流的数据并采取合适的操作是十分必要的。本文讨论的主题之一就是创建这样一个输入流，从这个输入流中可以读入以前写入Java控制台流（或任何其他程序的输出流）的数据。我们可以想象写入到输出流的数据立即以输入的形式“回流”到了Java程序。

要在文本框中显示控制台输出，我们必须用某种方法“截取”控制台流。换句话说，我们要有一种高效地读取写入到System.out和System.err 所有内容的方法。如果你熟悉Java的管道流PipedInputStream和PipedOutputStream，就会相信我们已经拥有最有效的工具。Class System提供了如下三个与流相关的函数：

```
static void     setErr(PrintStream err)
          Reassigns the "standard" error output stream.
static void     setIn(InputStream in)
          Reassigns the "standard" input stream.
static void     setOut(PrintStream out)
          Reassigns the "standard" output stream.
```

对于输出流的两个函数setOut和setErr:

```
PrintStream ps = new PrintStream(pipedOS);
System.setOut(ps);
System.setErr(ps);
```

### §12.15 Linux下如何将进程绑定在特定的CPU上运行？

以root用户执行如下命令

```
#bind <进程id> <cpu 掩码>
```

其中CPU掩码为十进制的形势。如果机器有4个CPU,那么用4位二进制数字中的每一位表示一个CPU,其中’0’表示不使用该CPU,1表示使用该CPU. 如: 0101(十进制为5) 表示使用第一个(从左边数第一位)和第三个CPU（从左边数第三位），0001(十进制为1)表示只使用第一个CPU.如果进程id为6000,那么：

```
#bind 6000 5
```

就表示使用第一个和第三个CPU.

在某些特殊场合下这个非常有用。比如内核态的死循环，如果所有的CPU都在内核态死循环，那么整个系统将挂死，不会对用户的任何命令进行响应，包括telnet等。如果通过绑定CPU的方式，留出一个CPU，那么可以保证这个系统在异常的时候，仍然被控制，此时可以收集有用的相关信息。

## §12.16 关于Java和C++的互通

Java和C++的互通有如下几种方式：

**JNI** Java通过JNI调用C++的动态库，C++也可以通过JNI调用Java代码。

**corba** 通过IDL接口，实现C++和Java的互相调用，Java和C++在不同的进程中，通过socket进行互通。

**JMS** JMS属于Java规范，但目前有C++实现的JMS客户端，Java和C++通过JMS进行通信。

**其它** 如ICE等非标准的第三方中间件完成通信。

在Java和C++的通信中，根据实际情况选用不同的调用方式。

### §12.16.1 Java代码中调用C++

```
public class JNISampleJava2C {

    //Native method declaration
      native String printMe(String str);
    //Load the library
      static {
        System.loadLibrary("JNIJava2C");
      }

    public static void main(String[] args) {
      //Create class instance
        JNISampleJava2C mappedFile=new JNISampleJava2C();
      //Call native method
      mappedFile.printMe("Hello World");
    }
}
```

#### Native Method Declaration

#### Compile the Program

**Generate the Header File** javah -jni JNISampleJava2C

### §12.16.2 C++代码中调用Java

C++代码中也可以调用Java,详细请参考：[24]和[25]

# §13 常见的案例

本节介绍一些常见的典型经验案例。

## §13.1 Too many open files

在操作系统中，每打开一个文件或者每创建一个socket，那么就会产生一个文件描述符（或者叫做文件句柄）。而操作系统对每个进程可以打开的文件socket都有一个数量限制。如果打开了，忘记关闭就会造成泄漏，特别在第 102页，第 §5.1节中提到的幽灵代码模式所造成的文件句柄泄漏更加隐蔽，需要特别关注。

### 情形一

```
java.net.SocketException: Too many open files
    at java.net.PlainSocketImpl.accept(Compiled Code)
    at java.net.ServerSocket.implAccept(Compiled Code)
    at java.net.ServerSocket.accept(Compiled Code)
    at weblogic.t3.srvr.ListenThread.run(Compiled Code)
```

操作系统的中打开文件的最大句柄数受限所致，关于文件句柄请参考第 194页第 §13.1节，常常发生在很多个并发用户访问服务器的时候. 因为为了执行每个用户的应用服务器都要加载很多文件(new一个socket就需要一个文件句柄),这就会导致打开文件的句柄的缺乏.造成这个异常的可能原因有如下几个：

- 打开的文件socket没有关闭，导致文件句柄泄漏，最终超过允许操作系统的最大句柄极限。特别是无意识的句柄泄漏，具体分析方法请参考pfiles，通过pfiles命令查出系统打开了哪些文件和socket,如果一个文件被打开很多次，那么很可能代码中存在文件句柄泄漏。(第 159页)[57]：

```
    File f = new File("c:/test/StoreTest-1.xml");
    java.io.BufferedReader br = null;
    try {
        br = new java.io.BufferedReader(new java.io.FileReader(f));
    } catch (FileNotFoundException e) {}
    ... ...
    //这里的代码抛出了异常，那么下面的close就无法调用到，
    //导致一个文件句柄泄漏
    br.close()
```

正确的写法应该是：

[57]又是幽灵代码在作祟

```
        File f = new File("c:/test/StoreTest-1.xml");
        java.io.BufferedReader br = null;
        try {
            br = new java.io.BufferedReader(new java.io.FileReader(f));
        } catch (FileNotFoundException e) {}
        ... ...
        finally{
          br.close() //在任何情况下，这句代码都可以得到执行。
       }
```

- 操作系统支持的文件句柄的数量有限.不能满足进程要求，这个可以通过修改操作系统内核参数来搞定。

**情形二** 但有的时候会出现如下的异常：

```
java.io.IOException: Too many open files
    at java.lang.UNIXProcess.forkAndExec(Native Method)
    at java.lang.UNIXProcess.(UNIXProcess.java:54)
    at java.lang.UNIXProcess.forkAndExec(Native Method)
    at java.lang.UNIXProcess.(UNIXProcess.java:54)
    at java.lang.Runtime.execInternal(Native Method)
    at java.lang.Runtime.exec(Runtime.java:551)
    at java.lang.Runtime.exec(Runtime.java:477)
    at java.lang.Runtime.exec(Runtime.java:443)
```

这个是由于交换分区不足造成的。关于交换分区请见请参考第 164页第 §9.7节
文件句柄泄漏的定位方法如下：

1. 使用pfiles等操作系统命令查出有哪些文件或者socket被打开（pfiles的使用方法请参考第159页第 §9.3.2节）

2. 根据文件名找到相应的模块，检查相关的代码.如果是socket，则根据端口号找到相应的模块，最后检查相关的代码。

## §13.2 java.lang.StackOverflowError

出现这种情况，是由于代码种存在递归调用导致的调用层次太多，超过了系统的限制。属于原代码bug. 通过观察异常堆栈，很容易发现问题所在。

```
public class Main {
    static long sum2(long a) {
        if (a == 1) {
          return 1;
        } else {
          return sum2(a - 1) + a;
        }
      }

    public static void main(String[] args) {
         System.out.println(sum2(10000));
    }
}
```

产生如下的结果：

```
E:\sketch\Java\overflow>java -classpath bin Main Exception in thread
"main" java.lang.StackOverflowError
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
        at Main.sum2(Main.java:7)
```

## §13.3 java.net.SocketException: Broken pipe

```
ClientAbortException:  java.net.SocketException: Broken pipe
    at org.apache.catalina.connector.OutputBuffer.realWriteBytes(OutputBuffer.java:358)
    at org.apache.tomcat.util.buf.ByteChunk.flushBuffer(ByteChunk.java:434)
    at org.apache.tomcat.util.buf.ByteChunk.append(ByteChunk.java:349)
    at org.apache.catalina.connector.OutputBuffer.writeBytes(OutputBuffer.java:381)
```

```
        at org.apache.catalina.connector.OutputBuffer.write(OutputBuffer.java:370)
    ... ...
```

往已经关闭的管道或者socket写数据就会抛这个异常。

## §13.4 HashMap的ConcurrentModificationException

如果hashmap在迭代的同时，被其他线程修改，则会抛出一个ConcurrentModification-Exception异常。同样提供了线程安全的HashTable也有同样的问题，HashTable的线程安全是指HashTable提供的方法进行了线程安全的处理，但是对于迭代访问，由于迭代的方法不属于HashTable的内部方法，因此HashTable 的内部进行方法同步是没有用的。

要理解HashMap/HashTable的内部实现就可以理解其中的缘由。Iterator指向HashMap内部数据结构的一个元素，当遍历下一个元素的时候，还要借助于当前元素进行下一个遍历，这样如果有其它线程修改了这些元素，势必造成遍历混乱。(可以分析一下HashMap和Iterator的内部代码来进行讲解)

```
package hashmap;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.Map.Entry;
public class HashMapTest {
    private static final Map map = new HashMap();

    public static void main(String[] args) {
        try {
            for (int i = 0; i < 10; i++) {
                map.put(i, i);
            }
            new Thread() {
                public void run() {
                    Set> set = map.entrySet();
                    synchronized (map) {
                        Iterator> it = set.iterator();
                        while (it.hasNext()) {
                           Entry en = it.next();
                            System.out.println(en.getKey());
                        }
                    }
                }
            }.start();
            for (int i = 0; i < 10; i++) {
```

```
                    map.put(i, i);
                }
            } catch (Throwable e) {
                e.printStackTrace();
            }
        }
    }
```

为了解决这个问题，jdk5.0以前的版本提供了Collections.SynchronizedXX()方法，对已有容器进行同步实现，这样容器在迭代时其他线程就不能同时进行修改了，但是由于Collections.SynchronizedXX()生成的容器把所有方法都进行了同步，其他线程如果只是读取数据也必须等待迭代结束。于是JDK5.0中新加了ConcurrentHashMap类，这个类通过内部独立锁的机制对写操作和读操作分别进行了同步，当一个线程在进行迭代操作时，其他线程也可以同步的读写，Iterator返回的只是某一时点上的，不会抛ConcurrentModificationException异常，比起hashmap效率也不会有太大损失。

## §13.5 多线程场合下HashMap导致的死循环

JDK针对HashMap有如下说明：

> Note that this implementation is not synchronized. If multiple threads access this map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.)

HashMap是线程没有进行同步的，如果多个线程并行存取HashMap,只要有线程修改这个map,那么必须在外部对HashMap进行同步. 这里所说的修改是指添加或者删除元素等操作，如果仅仅是改变一个已有key的值不在这个范畴。

从JDK的说明中，我们可以看出HashMap是线程不安全的，如果在多线程场合下使用HashMap要自己手工在外面进行同步，如果不做同步，可能会有如下不确定的问题发生：

1. 数据混乱。
2. 未知的行为，如无限循环。

在实际中，我们遇到的未加保护的HashMap访问导致的无限循环（死循环）最多。这种问题一旦发生，整个系统基本瘫痪。下面是一个实际的案例,整个系统瘫痪，现场收集回的多个堆栈(前后持续几分钟)，每次堆栈都包含如下的线程:

```
"Thread-131" prio=5 tid=0x0164eac0 nid=0x41e waiting for monitor entry[...]
    at java.util.HashMap.removeEntryForKey(Unknown Source)
    at java.util.HashMap.remove(Unknown Source)
    at meetingmgr.timer.OnMeetingExec.monitorExOverNotify(OnMeetingExec.java:262)
    -  locked <0xccf27372> (a [B)       //占有锁0xccf27372 <----------------------+
    at meetingmgr.timer.OnMeetingExec.execute(OnMeetingExec.java:189)            |
    at util.threadpool.RunnableWrapper.run(RunnableWrapper.java:131)             |
    at EDU.oswego.cs.dl.util.concurrent.PooledExecutor$Worker.run(...)           |
    at java.lang.Thread.run(Thread.java:534)                                     |
                                                                                 |
"Thread-1021" prio=5 tid=0x0164eac0 nid=0x41e waiting for monitor entry[...]     |
    at meetingmgr.timer.OnMeetingExec.monitorExOverNotify(OnMeetingExec.java:262)  |
    - waiting to lock <0xccf27372> (a [B) //等待锁0xccf27372 <---------------------+
    at meetingmgr.timer.OnMeetingExec.execute(OnMeetingExec.java:189)            |
    at util.threadpool.RunnableWrapper.run(RunnableWrapper.java:131)             |
    at EDU.oswego.cs.dl.util.concurrent.PooledExecutor$Worker.run(...)           |
    at java.lang.Thread.run(Thread.java:534)                                     |
                                                                                 |
"Thread-196" prio=5 tid=0x01054830 nid=0xe1 waiting for monitor entry[...]       |
    at meetingmgr.conferencemgr.Operation.prolongResource(Operation.java:474)    |
    at meetingmgr.MeetingAdapter.prolongMeeting(MeetingAdapter.java:171)         |
    at meetingmgr.FacadeForCallBean.applyProlongMeeting(FacadeFroCallBean.java:190)|
    at meetingmgr.timer.OnMeetingExec.monitorExOverNotify(OnMeetingExec.java:278)  |
    - waiting to lock <0xccf27372> (a [B) //等待锁0xccf27372 <---------------------+
    at meetingmgr.timer.OnMeetingExec.execute(OnMeetingExec.java:189)
    at util.threadpool.RunnableWrapper.run(RunnableWrapper.java:131)
    at EDU.oswego.cs.dl.util.concurrent.PooledExecutor$Worker.run(...)
    at java.lang.Thread.run(Thread.java:534)
... ...     //共有三百多线程再等待锁0xccf27372
```

从堆栈中分析，有如下几个疑点：

- 里面有三百多个线程在等待锁<0xccf27372>，在正常压力下，不应该出现如此激烈的锁争用。
- 占有锁的<0xccf27372>线程"Thread-131",在每次打印的堆栈中都存在，说明这个调用一直没有完成(即java.util.HashMap.removeEntryForKey())，按照正常来说，HashMap.remove()操作是一个非常快的操作，无论如何也不需要几分钟的时间，这里似乎已经陷入了一个无限循环。

结合代码逻辑分析，由于当前正在java.util.HashMap.removeEntryForKey()函数中死循环，导致线程"Thread-131"永不退出，因此这个线程占用的锁<0xccf27372>永远得不到释放，从而使得所有等待在这个锁上面的线程被饿死，永远无法获取这个锁，从而导致整个系统限于瘫痪。根本原因就是由于HashMap陷入了死循环！限于死循环的原因是因为HashMap被多线程

访问导致了数据混乱，从而造成了死循环。为了更深入说明这个问题，剖析一下HashMap中的removeEntryForKey()函数的源代码：

```
Entry<K,V> removeEntryForKey(Object key) {
    Object k = maskNull(key);
    int hash = hash(k.hashCode());
    int i = indexFor(hash, table.length);
    Entry<K,V> prev = table[i];
    Entry<K,V> e = prev;

    while (e != null) {
        Entry<K,V> next = e.next;
        if (e.hash == hash && eq(k, e.key)) {
            modCount++;
            size--;
            if (prev == e)
                table[i] = next;
            else
                prev.next = next;
            e.recordRemoval(this);
            return e;
        }
        prev = e;
        e = next;
    }

    return e;
}
```

从上面的代码看出，removeEntryForKey()函数存在一处while(e != null)可能导致无限循环的相关代码。从代码中看,Entry<K,V>是一个链表结构，通过遍历这个链表，找到合适的元素，如果这个链表的访问不加保护，很容易造成一个首尾相接的闭环链表。一旦造成首尾相接，那么无限循环的土壤也就生成了。

HashMap是线程不安全的，这一点一定要牢记于心，不但要记住，还要一定不要去冒险。你在google上使用"HashMap infinite loop" 搜索一下，你就会发现这种用法导致的问题数量触目惊心。不要侥幸这个问题不发生，而是它一定会发生，也许三天，也许三个月，一旦问题发生，带来的影响往往是致命性的。

## §13.6 Web系统吊死（挂死）的定位思路

web系统吊死的原因有如下几个：

1. 系统无可用线程 - 通过打印线程堆栈可以分析线程的使用情况。

(a) 系统当前的请求数量超过系统的最大能力，压力过大导致系统“忙”不过来，从而线程池(tomcat和Webshpere等Web容器都使用了线程池)中线程都在忙于处理请求，没有更多的可用线程来处理新的请求，当这种过负荷累积到一定的程度，处理一个请求的时间越来越长，最终达到不可忍受的程度。当然导致这种问题，可能是访问量真得太大了，此时只能通过购买更好的机器，或者集群来提升系统的处理能力。但实际上，很多情况下，并不是这样，而是拙劣的程序设计或者不恰当的配置导致的人为的性能瓶颈，常见的原因有：

- 锁竞争导致线程被长期阻塞(block)，导致线程耗尽。
- 资源竞争导致的线程长时间block，导致线程耗尽。
- 线程池数量设置太小，导致所有线程被耗尽。

(b) 系统存在死锁，导致所有线程耗尽。

(c) 系统存在死循环等。一个线程遭遇死循环，往往会有如下两种连锁反应：

- 由于该死循环的线程占有一把锁，而且永不释放，从而导致其它所有请求该锁的线程挂起，请求得不到继续处理，造成整个系统挂死。
- 在单CPU的机器上，该死循环的线程可能耗尽CPU,导致其它线程的处理非常慢，直到有大量超时发生，从外面看，整个程序挂死。

2. 系统无可用内存，请求得不到处理，通过-verbose:gc可以产看内存的使用情况。

   (a) 系统内存设置太小,-Xmx，导致内存不足

   (b) 系统存在内存泄漏，导致内存耗尽

3. 系统有Java异常抛出，导致业务不能正确完成 - 通过查看日志可以确定。

4. 系统其它代码错误，比如发生了异常，这种情况下通过单步跟踪确认问题。

5. 系统存在文件句柄泄漏（文件句柄泄漏的定位方法请见第 194页第 §13.1节），导致和web server的socket无法建立。

   web系统吊死的定位步骤如下：

1. 如果在浏览器中表现出滚动条一直在滚动，但无页面返回，说明Web Server端已经再处理该请求，但迟迟不结束，说明该问题是和线程有关系的问题，问题重现时，通过打印线程堆栈，检查线程的情况即可快速定位问题。具体请参考第 5页第 §1.2节。

   如果浏览器中出现空白页面，说明Web Server端给浏览器端返回的是一个空白页面，如果浏览器其它错误页面，说明Web Server端给浏览器返回的是错误页面，定位思路如下：

2. 在java命令行中增加-verbose:gc并启动系统，通过观察GC的使用情况，确定是否存在内存泄漏。具体请参考第 71页第 §3.3.1节。

3. 问题重现时，查看日志，检查是否有异常抛出，结合原代码检查该异常是否导致了实际问题。

4. 问题重现时，停止压力测试，单独使用一个请求去访问，单步跟踪确认问题。

同时注意观察浏览器的表现出来的症状：

1. 浏览器的响应迟迟不返回，进度滚动条一直再滚动。说明http线程一直不结束，重点关注死锁，资源不足造成的等待（资源太少或者资源泄漏等）

2. 出现页面不能访问的错误。说明http的socket都无法建立

3. 返回的页面是空白。说明系统有异常抛出，页面没有生成。重点查看日志中的异常。

## §13.7 基于消息系统（如sip）吊死的定位思路

消息系统吊死的原因一般有如下几个原因：

- 关键线程异常退出。

- 状态混乱，导致系统无法正常运行。

## §13.8 多线程读写socket导致的数据混乱

socket I/O函数都是线程不安全的，如果二个或多个线程同时对一个socket写，由于线程不安全的缘故，最终形成的数据流可能交叉混合在一起。导致对方编解码错误。

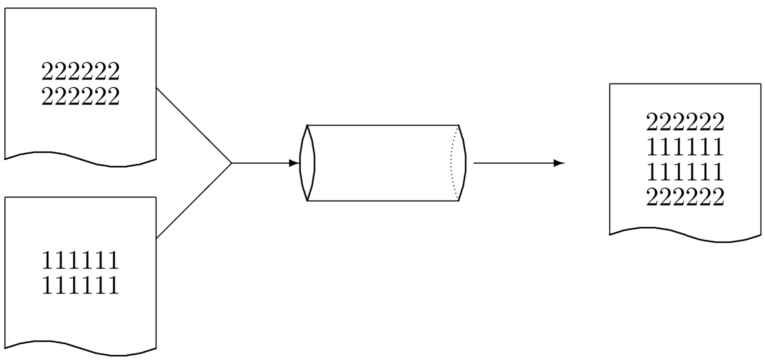



图 42 不加保护的消息发送

从图中可见，不加多线程保护写socket,会导致两个消息的内容打乱混在一起。

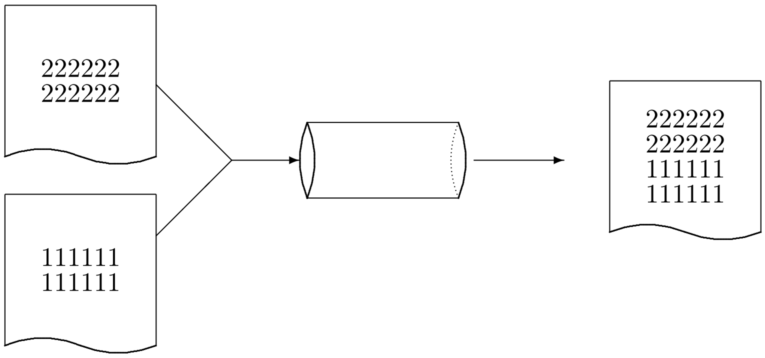



图 43 加保护的消息发送

从图中可见，通过多线程保护写socket,可以保证两个消息顺序地被写到socket中。

线程不安全的例子代码如下：

```
class MyClient{
  Socket server=new Socket(InetAddress.getLocalHost(),5678);
  BufferedReader in=new BufferedReader(new InputStreamReader(server.getInputStream()));
  PrintWriter    out=new PrintWriter(server.getOutputStream());
  BufferedReader wt=new BufferedReader(new InputStreamReader(System.in));

  public void sendmsg(String str)
  {
     out.println(str); //PrintWriter是线程不安全的
  }
}
```

PrintWriter是线程不安全的,如果多个线程同时调用sendmsg()方法，有两个选择：

1. 在out.println(str)语句上加锁保护，确保同时只能有一个线程调用PrintWriter上的方法，如:将上面的代码修改为：

```
        public void sendmsg(String str)
        {
            //通过手工同步，确保同一个时刻只有一个线程调用PrintWriter上的方法
            synchronized(out){
                out.println(str);
            }
        }
    }
```

2. 专门提供一个消息队列，多线程向这个消息队列写，专门一个消息发送线程，操作PrintWriter,向外发送消息，这种就是所谓的异步IO(即NIO),这种是性能最高的一种消息发送方式。

## §13.9 关于CPU过高问题的定位思路

首先要清楚为什么CPU会过高？我们是不是真正得理解了CPU过高得本质。从CPU的角度看，如果CPU正在执行代码，那么它在那个时刻是100%占用CPU,也就是说在执行机器码的过程中，CPU的使用率就很高。但实际上一大段代码，并不是所有的代码都消耗CPU。详细请参考第 20页第 §1.2.3节

线程的状态分为两种：

1. runnable 线程处于runnable状态.但处于runnable状态并不意味着一定消耗CPU,比如正在磁盘IO或者网络IO, 从线程的角度来看，它虽然处于运行状态，但由于挂起在本地代码中，因此实际上仍然处于等待状态。此时是不消耗CPU的。

2. Object.wait() 处于等待状态。正在调用sleep或者wait方法时，会处于这个状态，此时也不消耗CPU.

处于runnable线程，但在网络IO，这个几乎是不消耗CPU的:

```
"Thread-271" prio=1 tid=0xa4853568 nid=0x7ade runnable ...
    at java.net.SocketInputStream.socketRead0(Native Method)
    at java.net.SocketInputStream.read(SocketInputStream.java:129)
    at oracle.net.ns.Packet.receive(Unknown Source)
    at oracle.net.ns.DataPacket.receive(Unknown Source)
    at oracle.net.ns.NetInputStream.getNextPacket(Unknown Source)
    at oracle.net.ns.NetInputStream.read(Unknown Source)
    at oracle.net.ns.NetInputStream.read(Unknown Source)
    at oracle.net.ns.NetInputStream.read(Unknown Source)
    at oracle.jdbc.driver.T4CMAREngine.unmarshalUB1()
    at oracle.jdbc.driver.T4CMAREngine.unmarshalSB1()
    at oracle.jdbc.driver.T4C8Oall.receive(T4C8Oall.java:478)
    at oracle.jdbc.driver.T4CPreparedStatement.doOall8()
    at oracle.jdbc.driver.T4CPreparedStatement.executeForRows()
    at oracle.jdbc.driver.OracleStatement.executeMaybeDescribe()
    at oracle.jdbc.driver.T4CPreparedStatement.executeMaybeDescribe()
    at oracle.jdbc.driver.OracleStatement.doExecuteWithTimeout()
    at oracle.jdbc.driver.OraclePreparedStatement.executeInternal()
    at oracle.jdbc.driver.OraclePreparedStatement.executeQuery()
    - locked <0x93632280> (a oracle.jdbc.driver.T4CPreparedStatement)
    - locked <0x6b103258> (a oracle.jdbc.driver.T4CConnection)
    ......
```

处于wait的线程，这个是完全不消耗CPU的：

```
"Thread-269" prio=1 tid=0xa4851aa8 nid=0x7adc in Object.wait() ...
    at java.lang.Object.wait(Native Method)
```

```
    at com.util.collection.SimpleLinkedList.poll()
    - locked <0x6ae67be0> (a com.util.collection.SimpleLinkedList)
    at com.impl.XADataSourceImpl.getConnection_internal()
    at com.impl.XADataSourceImpl.getConnection()
    at org.hibernate.connection.DatasourceConnectionProvider.getConnection()
    at org.hibernate.jdbc.ConnectionManager.openConnection()
    at org.hibernate.jdbc.ConnectionManager.getConnection()
    at org.hibernate.jdbc.AbstractBatcher.prepareQueryStatement()
    at org.hibernate.loader.Loader.prepareQueryStatement()
    at org.hibernate.loader.Loader.doQuery(Loader.java:390)
    at org.hibernate.loader.Loader.doQueryAndInitializeNonLazyCollections()
    at org.hibernate.loader.Loader.doList(Loader.java:1593)
    at org.hibernate.loader.Loader.list(Loader.java:1577)
    at org.hibernate.loader.hql.QueryLoader.list()
    at org.hibernate.hql.ast.QueryTranslatorImpl.list()
    at org.hibernate.impl.SessionImpl.list()
    at org.hibernate.impl.SessionImpl.find()
    ......
```

处于runnable的线程，却真正消耗CPU的线程：

```
"Thread-444" prio=1 tid=0xa4853568 nid=0x7ade runnable ...
    at org.apache.commons.collections.ReferenceMap.getEntry(Unknown Source)
    at org.apache.commons.collections.ReferenceMap.get(Unknown Source)
    at org.hibernate.util.SoftLimitMRUCache.get(SoftLimitMRUCache.java:51)
    at org.hibernate.engine.query.QueryPlanCache.getNativeSQLQueryPlan()
    at org.hibernate.impl.AbstractSessionImpl.getNativeSQLQueryPlan()
    at org.hibernate.impl.AbstractSessionImpl.list()
    at org.hibernate.impl.SQLQueryImpl.list(SQLQueryImpl.java:164)
    at com.mogoko.struts.logic.user.LeaveMesManager.getCommentByShopId()
    at com.mogoko.struts.action.shop.ShopIndexBaseInfoAction.execute()
```

CPU过高一般是有如下原因造成的：

1. java代码中存在死循环导致CPU过高[58]
2. 系统存在不恰当的代码，尽管没有死循环，但仍然CPU过高。
3. JNI中有死循环代码，

[58]像HashMap这种线程不安全的容器类，在多线程同时访问时，很容易造成死循环。自己写的多线程代码需要特别关注这个问题。

4. 堆内存设置太小造成的频繁GC. (堆内存设置过小，或者存在内存泄漏)具体的定位方法请参考：第 71页第 §3.3.1节
5. 32位JDK下，堆内存设置太大造成的频繁GC. 具体请参考：第 81页第 §3.5.2节。
6. JDK自身存在死循环的Bug.

CPU过高问题定位的第一步就是要找到CPU高消耗的线程。对于由于代码导致的CPU过高，可以通过第 31页第 §1.3.3介绍了通用的方法来定位根本原因。

## §13.10 系统运行越来越慢的定位思路

系统运行越来越慢有如下几个可能原因造成的：

1. 系统存在内存泄漏，当内存越来越少的时候，FULL GC越来越频繁，并且每次GC的时间都很长，系统越来越少的时间在执行用户代码，整个系统越来越慢，直到整个系统停止工作。通过如下GC的输出可以看出，每大约5秒进行一次Full GC,每次GC的时间大约在4.5秒左右，也就是说像这种情况下，每5秒只有0.5在执行Java代码，因此系统看起来会非常慢。系统按这个趋势发展下去，就会越来越慢。

```
8190.825: [Full GC 1272056K->1217654K(1277056K), 4.3142190 secs]
8195.657: [Full GC 1274322K->1214535K(1277056K), 4.5135393 secs]
8200.491: [Full GC 1277684K->1225488K(1277056K), 4.1118171 secs]
8205.323: [Full GC 1278602K->1211545K(1277056K), 4.8186925 secs]
8211.169: [Full GC 1274576K->1216755K(1277056K), 4.4144430 secs]
```

2. 系统存在资源泄漏，慢慢导致了资源争用，随着资源泄漏的增多，更加加剧了资源争用，系统在外面看起来越来越慢。比如系统存在数据库连接泄漏的Bug, 慢慢地，当数据库连接池中可用的连接下降到一定的程度，必然导致资源争用，大多获取连接的线程会被暂时阻塞在获取连接的代码中，直到其它线程释放连接，当前被挂起的线程才可能继续运行。如:

```
http-8082-Processor84" daemon prio=10 tid=0x0887c000 nid=0x5663 in
Object.wait() [0x6c1ad000..0x6c1ae030]
  java.lang.Thread.State: WAITING (on object monitor)
  //当连接池没有可用的连接资源时，该线程会被挂起在wait()上面，直到
  //有新的可用连接，才能会唤醒。
at java.lang.Object.wait(Object.java:485)
at org.apache.commons.pool.impl.GenericObjectPool.borrowObject(Unknown Source)
   - locked <0x75132118> (a org.apache.commons.dbcp.AbandonedObjectPool)
at org.apache.commons.dbcp.AbandonedObjectPool.borrowObject()
   - locked <0x75132118> (a org.apache.commons.dbcp.AbandonedObjectPool)
at org.apache.commons.dbcp.PoolingDataSource.getConnection()
at org.apache.commons.dbcp.BasicDataSource.getConnection(BasicDataSource.java:312)
at dbAccess.FailSafeConnectionPool.getConnection(FailSafeConnectionPool.java:162)
at servlets.ControllerServlet.doGet(ObisControllerServlet.java:93)
```

## §13.11 系统挂死问题的定位思路

经常听到系统挂死的说法，但系统挂死只是一个表面现象。如何来分析系统挂死的真正原因？

系统挂死，从表面上来看，是系统不处理响应。对于Web系统来说，http请求无页面返回，对于消息系统来说，系统无响应消息，总之来说，系统像死了一样。导致系统挂死的原因很多，不同的系统有不同可能，具体的问题需要在特定的场景下进行分析，但总的归结原因有如下几种：

① 线程死锁

② 线程永远得不到唤醒(wait/notify)

③ 资源不足导致线程挂死在获取资源的代码中(如获取数据库连接)

④ 无限死循环

⑤ 内存溢出

⑥ 关键线程异常退出，导致消息得不到处理等。

其中①、②、③、④ 通过第 36页第 §1.3.5节介绍的线程堆栈可以得到定位。

对于一个线程挂死，本身不是抽象的，最终要落到具体的代码上去，从线程的角度来看，线程挂死属于下列情况的某一种：

1. 线程wait在一个monitor对象上，一直没有被唤醒。
2. 线程正在执行sleep。
3. 远程调用，对方一直没有返回。一般表现是长期处于读socket状态，如：

```
  "Thread-248" prio=1 tid=0xa58f2048 nid=0x7ac2 runnable
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.read(SocketInputStream.java:129)
```

4. 无限死循环。
5. 长期等待一个锁。

从线程堆栈中，通过多次打印堆栈，很容易判断出所谓的挂死的线程正在执行的具体的代码。

在极少的情况，可能是虚拟机的僵死导致系统无响应。作者曾经遇到过一个项目，在SUN JDK下开发并测试正常的程序，偶然在某个商业局点上使用了JRockit虚拟机，运行三天到四天左右就会出现Java进程僵死的情况。它的典型特征是，该java进程无任何响应，即使通过kill -3这种发信号的方式，进程也不做任何响应。另外通过top进程查看工具，进程状态可能是T状

态(跟踪/停止)。出现这种情况的原因有很多，作者通过各种手段首先排除了自身代码的可能因素（如内存泄露等），但问题仍然得不到解决。说明问题的可能原因在JDK上，后来将这个局点的JDK换成了SUN的JDK，一切正常。因此建议在什么JDK下开发的程序，商业局点部署的时候，最好采用同样的JDK,即使要更换JDK,最好做充分的稳定性测试。

### §13.12 关于线程死亡/线程跑飞

编写地不严密的线程池会"泄漏"线程，直到最终丢失所有线程。大多数线程池实现通过捕获抛出的异常或重新启动死亡的线程来防止这一点，

在server端的应用程序，系统往往会有一些全生命周期的线程，这些线程一旦运行，则永不退出，如处理消息队列的线程，线程池中的线程等。这些线程在系统中往往处于最关键的位置，一旦这些线程异常退出，常常造成的是整个系统的瘫痪。因此这种问题严重影响系统的稳定性和可靠性，同时这种问题具有很深的隐蔽性，一般只有在大压力或者极端的情况下，问题才会暴露，这样就带来了很大的隐患。正是由于这种问题的深度隐藏性，在开发期间这种问题如果不关注，往往问题会直接遗留到现网运行环境。下面就介绍一下导致这种漏网之鱼的可能原因。

- 未捕获的异常导致线程退出：只catch了已知的异常，没有Catch所有级别的异常

```
public void execute ( ) {
  do{
     try{
     ......
     }
     //当Throwable或者其它其它没有被捕获的异常抛出时，该while会退出，
     //这肯定是违背了初衷。
     catch( MyException e){
     }
  }while(true)
}
```

比上面的代码更加正确的应该为：

```
public void execute ( ) {
  do{
     try{
     ......
     }
     //捕获所有未知异常，确保while(true)永不退出
     catch( Throwable t){
         ... ...  //异常处理代码
     }
  }while(true)
}
```

但上面的代码仍然不够安全， 因为catch( Throwable t)的处理代码仍然有可能抛出异常，如果此处抛出异常仍然可以导致该线程退出。 更加完善的代码应该如下：

```
public void execute ( ) {
  do{
     try{
     ......
     }
     //捕获所有未知异常，确保while(true)永不退出
     catch( Throwable t){
         try{
           ... ... //异常处理代码
         }
         //在异常处理代码中再捕获所有未知异常，确保while(true)永不退出
         catch(Throwable t){
              //这里面什么都不要做了
         }
     }
  }while(true)
}
```

1. catch(Exception e)级别的所有异常
2. catch(Throwable t)级别的所有异常，这种异常往往在极端下才会发生，如(这里举一些例子)：
   – Cannot create Native thread
3. 异常处理代码中仍然要防止新的异常发生。

- 如果代码的流程依赖状态，在异常情况下，如果没有考虑相应的状态复位或者状态刷新，会导致业务级别的状态混乱，最终导致代码不能按照预先设计的思路进行运转，从而导致系统无法正常工作。
- 关键任务提交给线程池，没有对提交结果结果进行检查并处理，如果交给线程池不成功，那么该关键任务得不到执行，从而导致系统无法工作。线程池有如下几个原因，可能任务提交失败：
  – 系统繁忙，导致线程池任务队列满，从而将新提交的任务抛弃。此时提交任务代码需要做容错处理，等待重新提交，以确保该关键任务得到执行。

    *void java.util.concurrent.ThreadPoolExecutor.execute(Runnable command)*

    *Executes the given task sometime in the future. The task may execute in a new thread or in an existing pooled thread. If the task cannot be submitted for execution, either because this executor has been shutdown or because its capacity has been reached, the task is handled by the current RejectedExecutionHandler.*

*Throws:*

- *RejectedExecutionException - at discretion of RejectedExecutionHandler, if task cannot be accepted for execution*
- *NullPointerException - if command is null*

```
public void submitTask ( TestTask task ) {
    threadpool.execute(task);
}
```

正确的应该为：

```
public void submitTask ( TestTask task ) {
    try{
        threadpool.execute(task);
    }
    //检查是否提交成功
    catch(RejectedExecutionException e){
        //做相应的容错处理
    }
}
```

线程跑飞这种问题本身并不复杂，但在编码期间，很容易忽略该问题，并且这种问题一般隐藏都很深。在实验室测试，这种问题并不容易暴露，因此这就给日后的故障留下了伏笔，这种问题只能在编码期间进行严重关注。

### §13.13 关于虚拟机Core Dump

几乎每个人都遇到过Java进程core dump的痛苦经历。Java虚拟机Coredump的原因一般有如下几个：

1. 内存问题

   - Java 堆内存不足，导致虚拟机core dump，理论上说，如果虚拟机有足够的本地内存保证自身运行需要，并且虚拟机足够健壮，堆内存不足不应该导致虚拟机core dump,毕竟堆内存是在虚拟机的管理之下。但实际上虚拟机也有bug，往往表现出来的症状是虚拟机进程core dump. 况且出现了堆内存不足，系统本来就无法继续工作，虚拟机core dump反而更容易把问题给暴露出来.特别在系统在watchdog (关于watchdog的介绍请参考第 181页第 §12.2节) 监控之下的话，core dump很容易被检测出来，反而能帮助系统快速进行恢复。但堆内存不足，不一定一定会导致虚拟机core dump, 二者之间没有必然联系。导致堆内存不足的可能性有如下几个可能（内存不足问题的定位请参考第 71页第 §3.3.1节）：

     (a) java内存泄漏

     (b) Xmx设置太小

- 32位内存下，Xmx设置太大，导致Java进程自身需要的本地内存不足，详细请参考第81页第 §3.5.2节

2. JNI问题. JNI代码中的野指针等,定位方法如下：当错误发生时，使用gdb 等类似的调试器来寻找错误地址，这取决于您的操作系统。

   (a) 在产生core 文件的目录中启动gdb 调试器: gdb java core

   (b) 如果gdb 调试器可以读取core 文件，它就可以分析出失效发生的地址（例如，Segmentation fault in function_name at 0x10001234）。

   (c) 要获得一个堆栈跟踪信息，可以使用bt 命令：(gdb) bt

3. 虚拟机的Bug. 在接触大型系统之前，我对IBM,SUN,HP等国际一流公司有一种崇拜，对他们提供的JDK实现也是百分百信任。但当我接触到大型系统之后，发现情形不是这样的，这些系统中也存在严重或者致命的Bug，但这不是这些公司的错误，而是"人都会犯错误"。程序都是人写的，既然每个人都会犯错误，那么质量控制良好的公司仍然不能幸免，只是他们能控制到一个更好的层面。针对SUN JDK1.5的bug修改，请参考[17]. 这里仅针对死循环、内存泄漏、虚拟机core dump各举一例：

```
 Bug ID      Description
4879522   REGRESSION: infinite loop in ISO2022_JP$Decoder.decodeArrayLoop()
4839069   Huge LightweightDispatcher memory leak when JPopupMenu is recycled
4794360   REGRESSION: HotSpot server  core dumps with signal 11
```

   解决的办法是升级到当前最新的子版本。如你当前1.5.0_06版本，那么请尽快升级到1.5.0_15(直到当前最新的)，注意只升级到"_"所表示的子版本。这个子版本升级基本上只解决了bug,虚拟机特性没有很大的变化，可以大胆地升级。但是如果进行大版本的升级，就需要重新进行全面的测试才可以。如从1.4升级到1.5，从1.5升级到1.6等，这种大版本可能升级会遇到各种各样的兼容问题，需要进行专门的测试。但是子版本升级基本上不用考虑升级带来的兼容性问题。

4. 虚拟机通过<ctrl>+c停止时，线程没有正常停止，强行停止导致的core dump

5. JIT引入的core dump.在将一个Java 程序从使用Sun JDK 的平台迁移到使用IBM JDK 的平台上时，这两个供应商的JVM 中使用的优化技术之间可能存在很大差异，这些差异可能会对程序产生影响.JIT 对于Java 程序的执行流程会产生很大的影响。在将程序从一个平台上迁移到另外一个平台上可能碰到的问题如下：

   - 死锁挂起
   - 一直产生不正确的结果
   - 结果不一致
   - 不正常结束
   - 无限循环

- 内存泄漏
- 虚拟机莫名其妙地Core dump

如果在一个平台上系统运行正常，而到了另一个系统运行不正常，首先要怀疑的就是JIT。尽快在判断问题原因时，确定JIT 是否是问题的根源非常重要. 但是没有必要花费太多的努力在JIT 调试上，这可能会耗费大量的时间，最终可能是由于JIT 的一个小问题，而这个问题可能已经在相同版本的JVM 的一个最新的修正包中解决了。JIT 一直处于不断的更新之中，您所碰到的问题很可能早已在最新的修正包中解决了。如果怀疑是JIT导致的这个问题，很简单，把JIT禁调之后，再检查问题是否还存在，如果存在，则不是JIT导致的，若问题消失，说明是JIT相关的[59]。

## §13.14 系统运行运行越来越慢问题的定位思路

系统缓慢一般是由于如下几个原因造成的：

1. 堆内存泄漏造成的内存不足，导致系统越来越慢，直到停止。定位方法请参考：第 71页第 §3.3.1节
2. Xmx设置太小造成的堆内存不足，导致系统越来越慢，直到停止。第 71页第 §3.3.1节
3. 系统出现死循环，消耗了过多的CPU。具体的定位方法请参考：第 29页第 §1.3.2节
4. 系统资源竞争（如使用了数据库连接池中连接，获取连接会导致竞争），导致锁等待。具体的定位方法请参考：第 34页第 §1.3.4节以及第 151页第 §7.4.3节

## §13.15 代码GC导致的性能低下

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4867874

```
import javax.imageio.*;
import javax.imageio.stream.*;
import java.util.*;
import java.io.*;

public class JPEGImageReaderTest {

  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("USAGE: java JPEGImageReaderTest <jpegfile>");
      System.exit(1);
    }

    try {
      ImageReader reader =
```

[59] 关于IBM JIT问题的定位请参考第 297页第 K节，相关文档请参考[26]、[27]

```
          (ImageReader) ImageIO.getImageReadersByFormatName("jpeg").next();

        long start = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
          System.out.print('.');
          reader.setInput(new FileImageInputStream(new File(args[0])));
          reader.read(0);
          reader.reset();
        }
        System.out.println("\ntook " + (System.currentTimeMillis() - start) +
                           " ms");
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
```

```
[GC 399K->355K(1984K), 0.0275450 secs]
[GC 691K->462K(1984K), 0.0188700 secs]
[Full GC 857K->194K(1984K), 0.0989880 secs]
[Full GC 594K->203K(1984K), 0.0943940 secs]
[Full GC 602K->211K(1984K), 0.0866380 secs]
[Full GC 610K->194K(1984K), 0.1035630 secs]
```

## §13.16 java.lang.OutOfMemoryError: unable to create new native thread

请参考第 76页第 §3.3.2节

## §13.17 java.lang.OutOfMemoryError: PermGen space

请参考第 77页第 §3.3.3节

## §13.18 java.lang.OutOfMemoryError: Java heap space

请参考第 71页第 §3.3.1节

## §13.19 Connection Pool exhausted

系统出现下面的异常意味着连接池耗尽：

```
java.sql.SQLException: DBCP could not obtain an idle db connection, pool exhausted
  at   org.apache.commons.dbcp.AbandonedObjectPool.borrowObject(AbandonedObjectPool.java:123)
  at   org.apache.commons.dbcp.PoolingDataSource.getConnection(PoolingDataSource.java:110)
  at   org.apache.commons.dbcp.BasicDataSource.getConnection(BasicDataSource.java:312)
  at   org.mart.dbapi.dao.CCommonDAO.getConnection(CCommonDAO.java:46)
  at   org.mart.dbapi.dao.CCommonDAO.select(CCommonDAO.java:183)
  at   org.apache.jsp.todaynew_jsp._jspService(todaynew_jsp.java:304)
```

```
at   org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:133)
at   javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
... ...
at   org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPool.java:649)
at   java.lang.Thread.run(Thread.java:536)
```

造成连接池耗尽可能有如下的原因：

- 当前的访问量超过了配置连接数量能够支撑的压力,其中有如下几个可能：
  - 连接池的配置数量过少，导致在一定的访问压力下，配置的连接数量不足以支持当前的访问量。
  - 数据库访问过慢，导致连接被较长时间占用，得不到及时释放。
  - 尽管连接数不少，但当前的压力过大，仍然导致连接无法满足压力要求。

  上面三种可能归根结底属于性能范畴，下面的原因则是由于代码Bug导致的连接耗尽：

- 不恰当的代码导致的某些连接没有被关闭，这个属于Bug的范畴。这种Bug一般比较隐蔽，难以定位和分析。并且往往只在某些情况下才发生，往往得不到及时发现，而更具有危害性。总的来说，导致链接泄露的原因是由于某些链接的Connection.close()没有被执行，从而造成了这些链接的蒸发。
  - 第 §5.1.1节第 105介绍的幽灵代码模式，由于异常导致的Connection.close()方法遗漏，造成的链接泄露。
  - 关闭链接的代码在不同的if/switch分支中，导致在分支条件不满足的情况下，这句关键代码也得不到执行。

这种类型的问题，一般难以定位，只能依赖于检视代码查找错误。总之，这种open/close配对模式的代码，一定要确保配对执行。

## §13.20 系统时间更改导致的系统无法正常工作

```
3XMTHREADINFO "http-0.0.0.0:8680" (TID:0x0000000120F1E33,sys_thread_t:0x0000000120F078B0
              state: CW, native ID:0x0000000000232343) prio=8
4XESTACKTRACE at java/lang/Thread.sleep(Native Method)
4XESTACKTRACE at java/lang/Thread.sleep(Thread.java:938(Compiled code))
4XESTACKTRACE at org/apache/tomcat/util/net/PoolTcpEndpoint.run(PoolTcpEndpoint.java:639)
4XESTACKTRACE at java/lang/Thread.run(Thread.java:810)
```

```
    /**
     * The background thread that listens for incoming TCP/IP connections and
     * hands them off to an appropriate processor.
     */
    public void run() {

```

```
            // Loop until we receive a shutdown command
            while (running) {

                // Loop if endpoint is paused
                while (paused) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // Ignore
                    }
                }

                // Allocate a new worker thread
                MasterSlaveWorkerThread workerThread = createWorkerThread();
                if (workerThread == null) {
                    try {
                        // Wait a little for load to go down: as a result,
                        // no accept will be made until the concurrency is
                        // lower than the specified maxThreads, and current
                        // connections will wait for a little bit instead of
                        // failing right away.
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // Ignore
                    }
                    continue;
                }

                // Accept the next incoming connection from the server socket
                Socket socket = acceptSocket();

                // Hand this socket off to an appropriate processor
                workerThread.assign(socket);

                // The processor will recycle itself when it finishes

            }

            // Notify the threadStop() method that we have shut ourselves down
            synchronized (threadSync) {
                threadSync.notifyAll();
            }

```

```
        }
```

系统时间更改，虚拟机内部所有正在执行的时间敏感函数都会受到影响，如wait方法（带时间参数的方法）,sleep()。会导致wait或者sleep的阻塞时间提前或者延后(依赖于如何更改地系统时间)。影响尽管仅仅如此，但在实际的系统中，依赖于wait()或者sleep()的逻辑导致的衍生影响远远不止如此。如上面的例子，如果时间修改一天，那么sleep可能需要延迟一天才能完成，导致整个socket根本无法处理。导致的影响就是整个系统停止处理。

## §13.21 瞬间内存泄露的定位思路

如果系统存在瞬间内存泄漏或者只是高压力的环境下才出现内存泄漏，那么这种情况通过前面介绍的挂载JProfile 没有帮助，因此一旦挂载上这些剖析工具，整个系统性能急剧下降，导致问题不会重现，这种情况下就要借助虚拟机提供的事后信息收集进行定位。通过设置-XX:+HeapDumpOnOutOfMemoryError，当系统OutOfMemory的时候，会自动对内存进行dump,借助输出的dump文件进行内存分析(可以手工分析或者借助jhat等工具。)，虚拟机提供的这些事后分析工具可以说是定位类似问题的唯一有效手段。

## §13.22 第三方系统能力分析

在现网上，与第三方系统对接，但是发现性能根本无法上去，大量线程在等待，完全满足不了要求。从堆栈里面分析，等待0xc0967e20锁的线程共有1922个，说明这个系统有严重的瓶颈。

```
"worker-70809027279" daemon prio=5 tid=0x00a9bd30 nid=0x127 waiting for monitor entry
  at com.inprise.vbroker.GIOP.OutputStream.writeUnfragmented(OutputStream.java:163)
  - waiting to lock <0xc0967e20>( a com.inprise.vbroker.IIOP.Connection)
  at com.inprise.vbroker.GIOP.OutputStream.writeFragmented(OutputStream.java:86)
  at com.inprise.vbroker.GIOP.Message.write(Message.java:113)
  at com.inprise.vbroker.GIOP.GiopConnection.send_message(GiopConnection.java:286)
  at com.inprise.vbroker.GIOP.GiopConnection.send_message(GiopConnection.java:248)
  at com.inprise.vbroker.GIOP.ProtocolConnector.invoke(ProtocolConnector.java:778)
  at com.inprise.vbroker.orb.DelegateImpl.invoke(DelegateImpl.java:664)
  at com.omg.CORBA.portable.ObjectImpl._invoke(ObjectImpl.java:457)
  at de.payplugin.Clearing._ClearingStub.rechargeAmount3(_ClearingStub.java:343)
  at de.payplugin.clearingimpl.ClearingImpl.rechargeAmount3(ClearingImpl.java:250)
  at de.payplugin.processing.RechargeAmount3Req.execute(RechargeAmount3Req.java:109)
  at de.payplugin.processing.SendCorbaClearingRequest.run(SendCorbaClearingRequest.java:92)
  at de.payplugin.processing.PaymentProcessor$CorbaSender.run(PaymentProcessor.java:653)

"worker-70809027277" daemon prio=5 tid=0x00a9a098 nid=0x126 runnable
  at java.net.SocketOutputStream.socketWrite0(Native Method)
  at java.net.SocketOutputStream.socketWrite(SocketOutputStream.java:92)
  at java.net.SocketOutputSteam.write(SocketOutputStream.java:136)
  at com.inprise.vbroker.IIOP.Connection.write(Connection.java:251)
```

```
  at com.inprise.vbroker.GIOP.OutputStream.write(OutputStream.java:196)
  at com.inprise.vbroker.GIOP.OutputStream.writeUnfragmented(OutputStream.java:164)
  - locked <0xc0967e20>( a com.inprise.vbroker.IIOP.Connection)
  at com.inprise.vbroker.GIOP.OutputStream.writeFragmented(OutputStream.java:86)
  at com.inprise.vbroker.GIOP.Message.write(Message.java:113)
  at com.inprise.vbroker.GIOP.GiopConnection.send_message(GiopConnection.java:286)
  at com.inprise.vbroker.GIOP.GiopConnection.send_message(GiopConnection.java:248)
  at com.inprise.vbroker.GIOP.ProtocolConnector.invoke(ProtocolConnector.java:778)
  at com.inprise.vbroker.orb.DelegateImpl.invoke(DelegateImpl.java:664)
  at com.omg.CORBA.portable.ObjectImpl._invoke(ObjectImpl.java:457)
  at de.payplugin.Clearing._ClearingStub.rechargeAmount3(_ClearingStub.java:343)
  at de.payplugin.clearingimpl.ClearingImpl.rechargeAmount3(ClearingImpl.java:250)
  at de.payplugin.processing.RechargeAmount3Req.execute(RechargeAmount3Req.java:109)
  at de.payplugin.processing.SendCorbaClearingRequest.run(SendCorbaClearingRequest.java:92)
  at de.payplugin.processing.PaymentProcessor$CorbaSender.run(PaymentProcessor.java:653)

"http-8080-Processor88" daemon prio=5 tid=0x0010d838 nid=0xc9 in Object.wait()
  //在该锁上等待
  at java.lang.Object.wait(Native Method)
  at de.siemens.payplugin.processing.PaymentProcessor.execute(PaymentProcessor.java:425)
  //启动了一个新的线程向server请求，因此这里是该线程的专用锁，以备将来唤醒
  - locked <0xbdc238c8>  (a java.lang.Object)
  at de.payplugin.processing.PaymentProcessor.execute(PaymentProcessor.java:226)
  at de.payplugin.processing.PaymentConnection.execute(PaymentConnection.java:91)
  at de.payplugin.servlet.PaymentPluginServlet.processRequest(PaymentPluginServlet.java:192)
  at de.payplugin.servlet.PaymentPluginServlet.doGet(PaymentPluginServlet.java:350)
  at javax.servlet.http.HttpServlet.service(HttpServlet.java:740)
  at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
  ... ...
  at org.apache.tomcat.util.net.TcpWorkerThread.runIt(PoolTcpEndpoint.java:577)
  at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPool.java:683)
  at java.lang.Thread.run(Thread.java:534)
```

由于第三方不配合，只能通过堆栈分析，猜测第三方提供的lib库的内部实现，从上面堆栈来看，每一个http线程都wait在自己的锁上（每个http线程lock的锁id都不相同），同时根据PaymentProcessor.execute字样猜测，这里应该是又起了一个新的线程来继续处理请求，结合worker-的数量发现，worker线程的数量和http线程的数量是相同的，因此worker线程应该是http线程创建的，一旦worker线程执行完毕，http线程将被唤醒。结合这个分析，那么问题就转移到了worker线程上，为什么worker线程处理慢，结合worker线程的堆栈，发现worker线程正在通过corba调用到远端，而远端返回比较慢，因此造成了锁（0xc0967e20）的竞争，原因应该在远端的实现上。通过对远端堆栈分析，问题得到定位。

### §13.23 系统性能过低

在AIX操作系统下，一系统性能过低。经过打印堆栈，堆栈情况如下：

```
"http-8080-Processor47" (TID:0x0000000116612C00 native ID:0x00000000001260DD)
  at .../axis/encoding/DeserializerImpl.onStartElement(DeserializerImpl.java:444)
  at .../axis/encoding/DeserializerImpl.startElement(DeserializerImpl.java:393)
  at .../axis/encoding/DeserializeationContext.startElement(DeserializeationContext.java:1048)
  at .../axis/message/SAX2EventRecorder.replay(SAX2EventRecorder.java:165)
  at org/apache/axis/message/MessageElement.publishToHandler(MessageElement.java:1141)
  at org/apache/axis/message/RPCElement.deserialize(RPCElement.java:166)
  at org/apache/axis/message/RPCElement.getParams(RPCElement.java:384)
  at org/apache/axis/client/Call.invoke(Call.java:2467)
  ... ...
  at com/myspace/IShareWSWHHandlerStub.getShareToMe(IShareWSWHHandlerStub.java:221)
  ... ...
  at javax/servlet/http/HttpServlet.service(HttpServlet.java:709)
  ... ...
  at com/myspace/RightFilter.doFilter(RightFilter.java:335)
  ... ...
  at org.apache/coyote/http11/Http1BaseProtocol1$Http11ConnectionHandler
                             .processConnectin(Http1BaseProtocol1.java:664)

"http-8080-Processor48" (TID:0x0000000116660000 native ID:0x000000000243019)
  at org/.../XMLBSDocumentScannerImpl.scanEndElement(XMLBSDocumentScannerImpl.java:164)
  at org/.../XMLDocumentFragmetScannerImpl
             $FragmetnContentDispatcher.dispatch(XMLDocumentFragmetScannerImpl:365)
  at org/a.../XMLDocumentFragmetScannerImpl.scanDocument(XMLDocumentFragmetScannerImpl:26)
  at org/apache/xerces/parsers/XML11Configuration.parse(XML11Configuration:26)
  ... ...
  at org/apache/axis/client/AxisClient.invoke(AxisClient.java:206)
  ... ...
  at javax/servlet/http/HttpServlet.service(HttpServlet.java:709)
  ... ...
  at com/myspace/RightFilter.doFilter(RightFilter.java:335)
  ... ...
  at org.apache/coyote/http11/Http1BaseProtocol1$Http11ConnectionHandler
                             .processConnectin(Http1BaseProtocol1.java:664)

"http-8080-Processor49" (TID:0x0000000116633000 native ID:0x00000000017805D)
  at org/.../SAX2EventRecorder.replay(SAX2EventRecorder.java:162)
  at org/a.../MessageElement.publishToHandler(MessageElement.java:1141)
  at org/apache/axis/message/RPCElement.deserialize(RPCElement.java:166)
  at org/apache/axis/message/RPCElement.getParams(RPCElement.java:384)
```

```
  at org/apache/axis/client/Call.invoke(Call.java:2467)
  ... ...
  at com/myspace/IShareWSWHHandlerStub.getShareToMe(IShareWSWHHandlerStub.java:221)
  ... ...
  at javax/servlet/http/HttpServlet.service(HttpServlet.java:709)
  ... ...
  at com/myspace/RightFilter.doFilter(RightFilter.java:335)
  ... ...
  at org.apache/coyote/http11/Http1BaseProtocol1$Http11ConnectionHandler
                              .processConnectin(Http1BaseProtocol1.java:664)


"http-8080-Processor50" (TID:0x0000000113256097O native ID:0x00000000014D047)
  at java.net.PlainSocketImpl.socketAccept(Native Method)
  at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:353)
  at java.net.ServerSocket.implAccept(ServerSocket.java:448)
  at java.net.ServerSocket.accept(ServerSocket.java:419)
  at org.apache.tomcat.util...acceptSocket(DefaultServerSocketFactory.java:60)
  at org.apache.tomcat.util.net.PoolTcpEndpoint.acceptSocket(PoolTcpEndpoint.java:368)
  at org.apache.tomcat.util.net.TcpWorkerThread.runIt(PoolTcpEndpoint.java:549)
  at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPool.java:683)
  at java.lang.Thread.run(Thread.java:534)
```

从多次打印堆栈看，正在"干活"的线程（即执行用户代码的线程）始终是3到4个，其它线程都处于在线程池中空闲状态。同时发现该进程占用的CPU非常高，几乎接近于饱和。说明很有可能这几个线程消耗了大量的CPU。该环境有4个CPU, 除非是CPU密集型操作，每个线程长期占有一个CPU的计算量。因此元凶基本锁定在这四个线程正在实行的代码上。从堆栈来看，这三四个线程执行的代码都是解析soap消息，因此怀疑soap消息的解析上导致了CPU过高，从而导致系统整体性能低。

至此，下一步的目标就是查找到底是什么原因导致了soap解析计算量大？是解析算法太复杂？还是soap消息太大？首先经过抓包分析，发现一个soap消息大约是0.5M, 按照这个消息量计算，百兆网络上只能传递大约25个消息(0.5*25=12.5M)。而解析这个消息消耗大量的CPU,系统慢也就是情理之中了。至此问题得到了定位。

## §13.24 病灶转移-Java程序内存溢出(OutOfMemory)导致的数据库锁表

## §13.25 AIX下如何定位IO 100%的问题？

1. topas -P

2. filemon -o out.txt -O all

3. truss -p <pid>

## §13.26 高性能UDP程序

通过修改sysctl.conf，可以增大UDP缓冲区，对于处理能力特别强的机器，系统的瓶颈可能处在UDP socket上，

```
#最大的接受UDP缓冲区大小
net.inet.udp.sendspace=65535
#最大的发送UDP数据缓冲区大小
net.inet.udp.maxdgram=65535
```

# 附录 A JProfiler内存泄漏精确定位

通过JProfiler能够对内存泄漏进行精确定位。JProfiler可以列出每一种对象的数量，通过观察对象数量的变化，可以确定泄漏的对象。JProfiler精确内存泄漏定位的本质是通过JProfiler找到非正常增长的对象，然后结合分析源代码，找到泄漏的根本原因并进行消除。总的思路是，启动模拟压力测试代码或者压力测试工具，等待系统达到一个稳定的运行状态，这里所说的稳定，是指系统的内存使用应该达到一个动态平衡，此时压力均匀，系统不应该有大的内存使用波动，同时系统如果有缓存设计，要确保缓存已经达到饱和。满足这个条件之后，如果观察到的某一类型的对象数量不断再增大，那么这种类型的对象就是内存泄漏的重点嫌疑对象。当然由于JVM的垃圾回收时间不确定，在JProfiler中观察到的某一类对象的数量可能包含两部分，一部分是真正在使用的对象，另一部分是不再使用的垃圾对象，但由于垃圾回收尚未启动，因此这部分垃圾对象仍然包括总数里面。而我们真正关心的是真正正在使用的对象的数量，为此，JProfiler提供了一个垃圾回收的按钮，通过该垃圾回收按钮可以让虚拟机启动完全垃圾回收(FULL GC)，完全垃圾回收之后，我们看到的数量就是真正正在使用的对象的数量。通过观察真正在用的对象的数量的变化情况，可以很容易得找到泄漏的对象。

使用JProfiler进行内存泄漏分析，有如下几个注意事项：

1. 由于系统一旦挂上JProfiler，JProfiler自身进行对象信息收集是非常消耗内存，因此，一般情况下要通过Xmx把堆内存设置的大一些，根据经验，一般设到800M到1000M是比较可行的（当然依赖于应用对内存需求的不同而不同）。但是也不能设置太大，一旦设置太大，会挤压本地内存的空间，导致本地内存不足。如果挂载JProfiler的JVM启动没多久，就出现了OutOfMemory，那么首先要怀疑的就是-Xmx设置不合理导致的，此时尝试调整-Xmx问题一般会得到解决。

2. 另外，JProfiler自身进行对象信息收集也是非常消耗CPU的，挂上JProfiler之后，系统会变得异常缓慢，此时压力一定要低，否则会有很多其它错误出现，反而对分析会造成干扰和视线转移。

3. 同时,挂上JProfiler之后，虚拟机容易core dump,此时最好打开-verbose:gc, 观察GC的情况，当内存使用的比较多的时候，停下压力测试工具，然后进行分析。否则如果内存用得比较大时再分析，虚拟机容易core dump.辛勤的劳动付之东流，只能重新开始。

具体过程如下：

**远程启动JProfiler** 请参考联机手册 /reference/Managing Sessions/Starting remote sessions[60]

**内存泄漏分析**

1. 启动模拟压力测试代码或者压力测试工具，等待系统达到一个稳定的运行状态

[60]远程启动JProfiler的方法如下："-agentlib:jprofilerti=port=8849 -Xbootclasspath/a:/home/jprofiler5/bin/agent.jar",命令行的具体含义，请参考第 138页，第 §7.1节

2. 点击Memory View进入，内存视图,见图：44

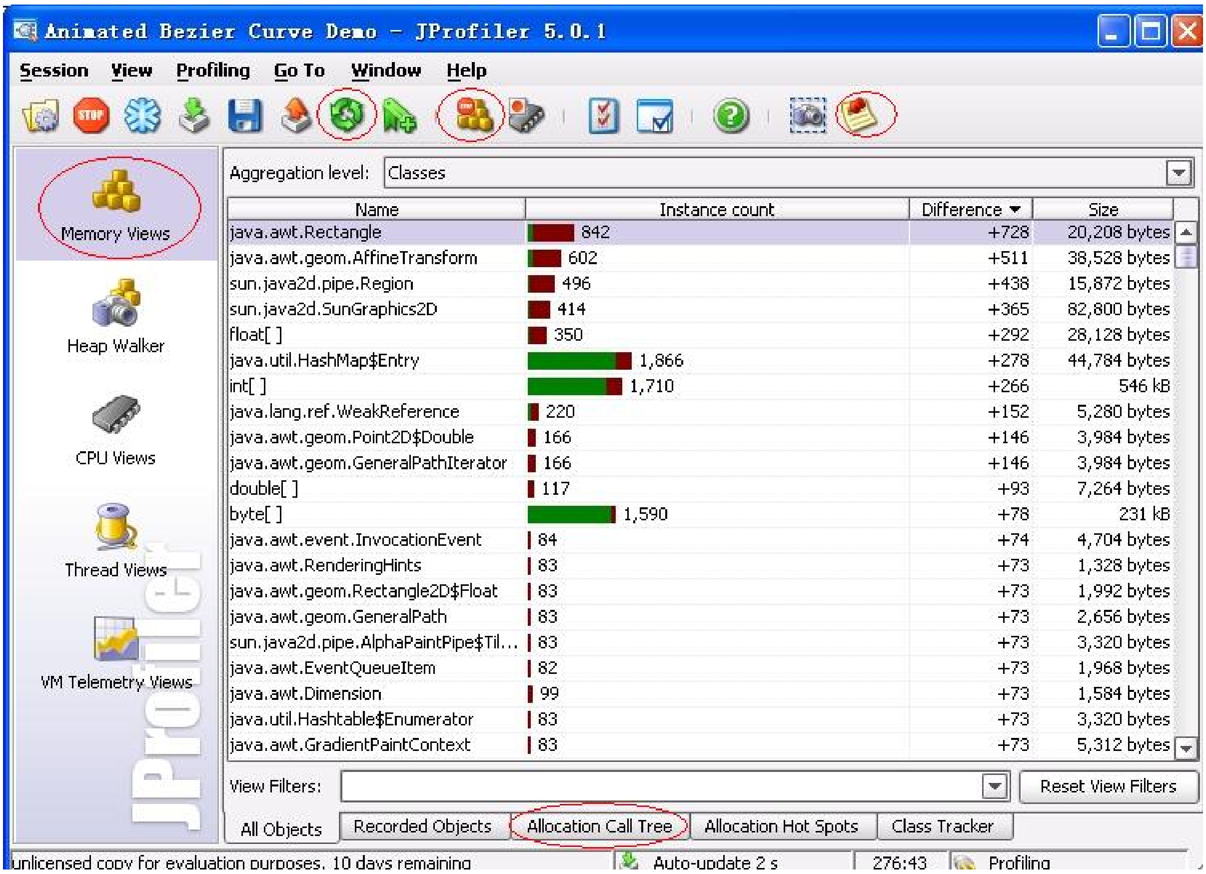



图 44 使用JProfile进行内存泄漏定位－找到内存泄漏的对象

3. 点击垃圾回收按钮，然后马上点击mark按钮，对当前的内存中对象数量进行标记

4. 等待一段时间（几个小时或者一个晚上，依据系统内存泄漏的快慢）

5. 停止压力，只所以要停止压力主要是为了观察的方便，停止压力后，系统中的对象数量基本保持不变。如果不停止压力进行观察，对象的数量一直处于变化之中，观察起来不是很方便。

6. 然后再点击垃圾垃圾回收按钮，观察一下difference的值，经过多轮之后，如果某些类的对象持续增加，那么基本能确认存在内存泄漏的对象。

   在JProfile中可能发现很多对象有泄漏，其中会发现一些Java自带类的对象泄漏，这些其实往往是由于我们自己对象的内存泄漏导致了这些对象的泄漏，即我们自己的对象引用了JDK的自带的类的对象，我们自身实现的对象有内存泄漏，那么必然也导致JDK自带对象也有泄漏，因此在定位过程中可以直接忽略这些JDK自带类的对象泄漏分析，而将关注点放在自己的实现类上，一旦自己的实现类内存泄漏解决，由于引用已断，因此Java自带类的泄漏也会自行消失。为此，JProfiler提供了"View

Filters"输入框，在这个框中输入自身包名作为过滤条件，如：com.XX.*,此时将只显示与此相关的类的对象的数量，此时通过过滤之后，很容易找到哪些对象泄漏。

7. 然后点击"Allocation call Tree" Tab页进入对象分配调用树分析页面。见图45。从这个页面中，我们可以看到这些泄漏的对象在哪里分配的，然后结合我们自身的代码逻辑检查，这些泄漏的对象是否被其它地方引用到了而没有被释放掉？比如一个外部的HashMap对象引用了该对象，但是当该对象不需要的时候，是否忘记了调用HashMap.remove()方法将该对象从HashMap中清除掉？

   从这个视图中看，泄漏的对象可能不止一种，在这种情况下有如下几种可能：

   - 确实存在多处内存泄漏。
   - 存在的内存泄漏只有一处，由于泄漏的对象又引用了其它的对象，因此其它的对象也造成了泄漏，但根源只有一处。在分析的过程中，需要注意找这些对象之间的规律.

   如果实在不清楚这些泄漏的对象到底被哪些对象引用了，还有一个小技巧，就是将过滤条件设为java.util.*,然后再观察是否有一些HashMap$Entry,TreeMap$Entry等容器类的对象也在增加，如果再增加，说明泄漏的那些对象被这些容器类引用到了，说明忘记从容器类中删除了。

8. 右键点击窗口，弹出右键菜单，点击: "Calculate Allocation Call Tree" 菜单，出现图46,输入最大嫌疑的类名。

9. 出现对象分配树，根据对象分配树中指出的对象分配点，结合源代码，检查对象是否被长久引用。

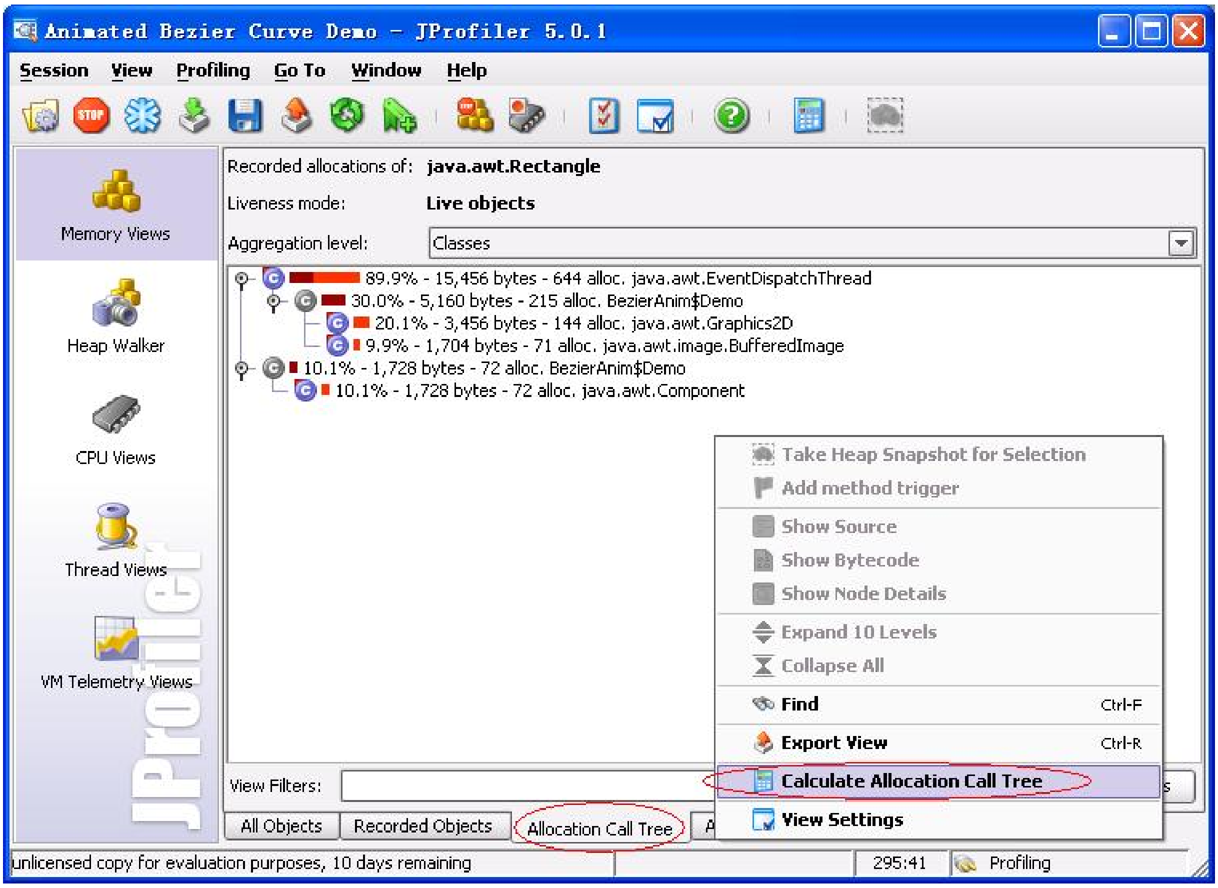



图 45 使用JProfile进行内存泄漏定位—找到泄漏对象的分配树

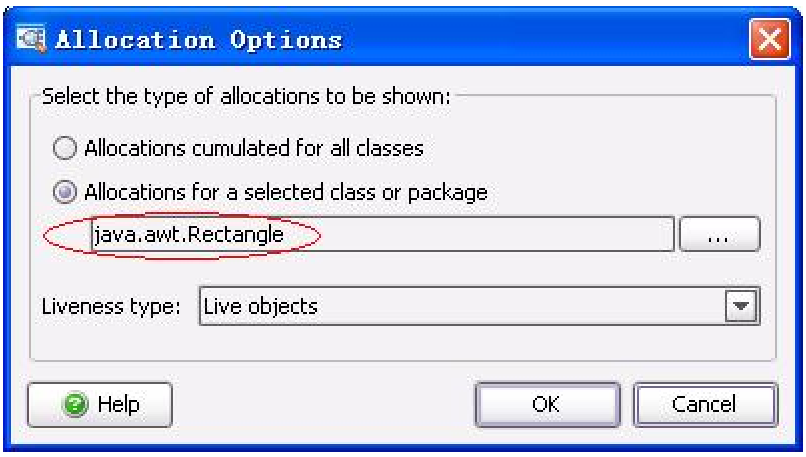



图 46 使用JProfile进行内存泄漏定位—指定对应类的对象分配树

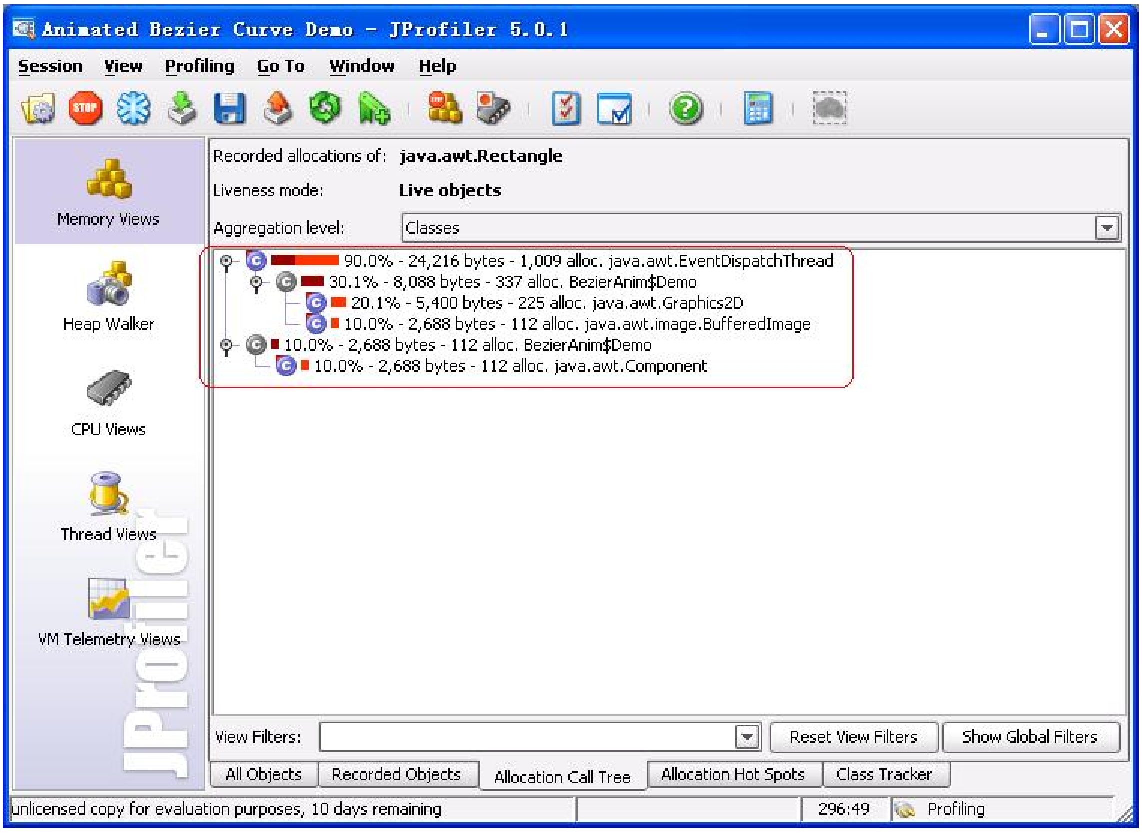



图 47 使用JProfile进行内存泄漏定位－泄漏对象的分配树

**在哪些场合下JProfile看似能实际上确不能？** JProfiler除了分析内存之外，还可以分析每个函数执行的时间，进行性能分析。但实际上在多线程的后台程序中，往往只有在高压力下才能出现的瓶颈，由于JProfiler依附在JVM上带来的开销，使系统根本就无法达到该瓶颈出现时需要的性能。因此这种类型的性能瓶颈无法出现，也就无法找到这个性能瓶颈。在这种场合下，进行线程堆栈分析才是一个真正有效的办法。

另外，在使用了线程池的场合，由于在观察期间，每个线程可能执行了多段不同的代码段，由于JProfiler只能给出执行的所有代码段的分析结果，因此这种情况下，用Jprofiler提供的性能剖析数据很难进行分析。

另外，如果在JVM中启动了JProfiler代理，该代理自身对内存有相当大的消耗，可能比你自己的程序还要消耗更多的内存。启动JProfiler的代理命令行参数如下：

```
-agentlib:jprofilerti -Xbootclasspath/a:/opt/JProfiler/bin/agent.jar
                |                                               |
                |                                               +-- java库
                +--------JProfiler代理的本地库jprofilerti.dll(.so)
```

从命令行参数中可以看出，JProfiler代理包含两部分，一部分是本地动态库，另一部分为java库，因此在启动了JProfiler代理的JVM中，JProfiler本身要消耗的内存也分为如下两部分：

1. java堆内存

2. java本地内存

相比没有挂载JProfiler代理的JVM,需要消耗多得多的本地内存和Java堆内存，因此在32位的机器上(或者32位的JDK), 必须小心设置-Xmx的大小。-Xmx设置太大会导致本地内存不足（详细请参考第 82页第 §3.5.2节），-Xms设置太小，会很快导致Java堆内存不足，从而影响问题的定位。因此-Xmx不能设置过大，同时也不能设置过小。

# 附录 B SUN JDK自带故障定位

## 附录 B.1 SUN JDK命令行选项

本章介绍了JDK1.6中各种诊断和监测工具，可用于Java平台标准版开发工具包6[61]。

### 附录 B.1.1 诊断工具和选项

**事后诊断工具** 下面介绍事后诊断工具：

| | |
|---|---|
| Fatal Error Log | 当致命错误发生时，致命日志信息将被写到一个文件中。 |
| -XX:+HeapDumpOnOutOfMemoryError | 当虚拟机检测到本地内存溢出时，产生堆栈文件 |
| -XX:OnError | 该命令行选项指定致命错误发生时需要运行的脚本。例如，在windows系统上，在致命问题发生时强制进行堆栈转储。当事后调试器没有打开时，该命令特别有用。 |
| -XX:+ShowMessageBoxOnError | 当致命错误发生时，JVM将被挂起，通过这个选项，此时可以将一些调试工具（如：gdb,dbx等）挂接到JVM进程进行分析。 |
| Java VisualVM | 借助该工具可以对转储文件进行可视化分析。 |
| jdb | 借助该工具，可以分析问题发生时线程正在做什么。 |
| jhat | 借助该工具，可以从堆转储文件中分析对象分配情况。 |
| jinfo | 借助该工具，可以分析转储文件的配置信息。 |
| jmap | 借助该工具，可以分析转储文件的内存映射情况。 |
| jstack | 借助该工具，可以从正在运行的虚拟机中或者转储文件中获取本地和Java堆栈信息。 |
| Native tools | 操作系统自带的分析工具 |

[61] 详细请参考[28]

**挂起进程的在线诊断工具** 下面介绍挂起进程的在线诊断工具：

| | |
|---|---|
| Ctrl-Break[62] | 通过将当前挂起进程的堆栈进行转储，以分析死锁等问题。 |
| jdb | 通过jdb，可以将该工具attach到正在运行的虚拟机上，可以分析当前的线程状况 |
| jhat | 通过jhat，可以对当前正在运行的虚拟机进行堆内存分配分析. |
| jinfo | 通过jinfo,可以观察当前正在运行的虚拟机的信息。 |
| jmap | 通过jmap,可以获取当前进程的内存信息。在Solaris和Linux上，对于已经挂起的进程，可以使用-F选项。 |
| jsadebugd | 通过jsadebugd,可以作为一个Debug代理挂接到一个Java进程或者core文件上，相当于一个debug server. |
| jstack | jstack,可以获取当前进程的堆栈信息。在Solaris和Linux上，对于已经挂起的进程，可以使用-F选项。 |
| Native tools | 操作系统自带的分析工具 |

**监控工具和选项** 下面介绍挂监控工具和选项：

| | |
|---|---|
| Java VisualVM | 该工具可以对正在运行的虚拟机进行监控，提供可视化的界面观察虚拟机的详细信息。 |
| JConsole | 该工具是基于JMX的监控工具，他使用了JVM中内嵌的JMX指令，监控正在运行程序的性能和资源消耗情况。 |
| jmap | 该工具可以获取当前正在运行进程或者core文件的内存映射信息。 |
| jps | 该工具列出目标系统上的侵入式虚拟机，当在VM是内嵌式的环境下（即虚拟机被JNI调用启动，而不是被虚拟机launcher启动），该工具非常有用。 |
| jstack | 借助该工具，可以从正在运行的虚拟机中或者转储文件中获取本地和Java堆栈信息。 |
| jstat | 该工具使用HotSpot VM内嵌的指令来获取当前程序的性能和资源消耗情况。该工具可以用来分析性能问题，特别是和堆大小以及垃圾收集相关的性能问题。 |
| jstatd | 该工具是一个RMI server类型的应用程序，监控内嵌式虚拟机的创建和停止等，他提供了被远程工具连接上来的接口。 |
| visualgc | 该工具提供了一个垃圾回收系统的图形化监控工具，它使用了HotSpot VM的内嵌指令。 |
| Native tools | 操作系统自带的分析工具 |

**其它工具和选项等** 下面介绍其它工具和选项：

| | |
|---|---|
| HPROF profiler | 它可以统计CPU使用状况，堆内存分配情况，以及所有的锁和线程。HPROF在分析性能，锁，内存泄露等非常有用。 |
| jhat | 这个工具在诊断内存泄露方面非常有用，同时可以通过它来浏览堆对象的分配情况，找到所有的可达对象，显示哪些对象是活动对象。 |
| -Xcheck:jni | 该选项在诊断JNI问题时非常有用。 |
| -verbose:class | 通过开选项，JVM在加载和卸载类时，会打印出日志。 |
| -verbose:gc | 通过开选项，JVM在做垃圾回收时，会打印出日志。 |
| -verbose:jni | 通过开选项，JVM在做JNI相关操作时，会打印出日志。 |

## 附录 B.2 诊断工具详细介绍

### 附录 B.2.1 HPROF - Heap Profiler

The Heap Profiler (HPROF) tool 是一个随JDK一同发布的简单剖析工具。它是采用Java虚拟机工具接口( Java Virtual Machine Tools Interface-JVM TI)实现的动态库。这个工具可以将剖析信息以二机制或者ascii码方式写到文件或者socket中。这些信息也可以被一些前端剖析分析工具进一步处理。

HPROF工具能够显示CPU使用情况，对内存分配情况，以及monitor(锁)的使用的情况，另外，它可以转储虚拟机中完整的堆使用情况以及monitor，和线程。在分析性能问题，锁竞争，内存泄漏等问题上，这个HPROF工具非常有用。在JDK发布自带的HPROF库里面，包含了HPROF的JVM TI的演示代码，这些代码放在了$JAVA_HOME/demo/jvmti/hprof目录下。HPROF工具使用方法如下：

```
$ java -agentlib:hprof ToBeProfiledClass
```

根据请求命令的类型，HPROF指示虚拟机发送给它相关的事件，然后该工具处理这些事件，形成剖析信息，例如下面的命令获取堆分配的剖析信息：

```
$ java -agentlib:hprof=heap=sites ToBeProfiledClass
```

下面列出了HPROF命令行选项。

```
$ java -agentlib:hprof=help
     HPROF: Heap and CPU Profiling Agent (JVMTI Demonstration Code)
hprof usage: java -agentlib:hprof=[help]|[<option>=<value>, ...]
Option Name and Value  Description                 Default
---------------------  -----------                 -------
heap=dump|sites|all    heap profiling              all
cpu=samples|times|old  CPU usage                   off
monitor=y|n            monitor contention          n
format=a|b             text(txt) or binary output  a
file=<file>            write data to file          java.hprof[{.txt}]
```

```
net=<host>:<port>      send data over a socket        off
depth=<size>           stack trace depth              4
interval=<ms>          sample interval in ms          10
cutoff=<value>         output cutoff point            0.0001
lineno=y|n             line number in traces?         y
thread=y|n             thread in traces?              n
doe=y|n                dump on exit?                  y
msa=y|n                Solaris micro state accounting n
force=y|n              force output to <file>         y
verbose=y|n            print messages about dumps     y
Obsolete Options
----------------
gc_okay=y|n
<>
Examples
--------
  - Get sample cpu information every 20 millisec, with a stack depth of 3:
      java -agentlib:hprof=cpu=samples,interval=20,depth=3 classname
  - Get heap usage information based on the allocation sites:
      java -agentlib:hprof=heap=sites classname
Notes
-----
  - The option format=b cannot be used with monitor=y.
  - The option format=b cannot be used with cpu=old|times.
  - Use of the -Xrunhprof interface can still be used, e.g.
       java -Xrunhprof:[help]|[<option>=<value>, ...]
    will behave exactly the same as:
       java -agentlib:hprof=[help]|[<option>=<value>, ...]
Warnings
--------
  - This is demonstration code for the JVMTI interface and use of BCI,
    it is not an official product or formal part of the JDK.
  - The -Xrunhprof interface will be removed in a future release.
  - The option format=b is considered experimental, this format may change
    in a future release.
```

缺省情况下，堆信息将被写入当前目录的java.hprof.txt文件中(Ascii) 当虚拟机退出时，堆信息将会被打印出来，如果在退出时，不需要堆信息，可以通过将"dump on exit" 选线设为"n" (doe=n)。另外，在windows下，通过<ctrl>+<break>组合键可以将堆信息打印出来，在Solaris/Linux下，通过kill -QUIT pid来完成输出。

在大多数情况下，输出包括跟踪ID,线程ID,对象ID等。典型地，每一种类型的ID以不同的数字作为开头，比如trace id也许以300000作为开头。

**堆分配点剖析(heap=sites)** 下面的输出是由Java编译器(Javac)在编译一系列源代码文件时产生的堆分配信息，这里仅列出一部分：

```
$ javac -J-agentlib:hprof=heap=sites Hello.java
SITES BEGIN (ordered by live bytes) Wed Oct 4 13:13:42 2006
          percent          live          alloc'ed  stack class
 rank   self  accum     bytes objs     bytes  objs trace name
    1 44.13% 44.13%   1117360 13967  1117360 13967 301926 java.util.zip.ZipEntry
    2  8.83% 52.95%    223472 13967   223472 13967 301927 com.sun.tools.javac.util.List
    3  5.18% 58.13%    131088     1   131088     1 300996 byte[]
    4  5.18% 63.31%    131088     1   131088     1 300995 com.sun.tools.javac.util.Name[]
```

在堆分配剖析文件里，最关键的信息是程序每一部分分配的对象的数量。上面显示为44.13%的SITES记录表示java.util.zip.ZipEntry对象占用了总空间的44.13%.

关联源代码和分配点的最好方式是记录导致内存分配的线程堆栈。下面的是剖析输出的另外一部分信息，下面四个分配点说明了是由哪个调用堆栈产生的：

```
TRACE 301926:
        java.util.zip.ZipEntry.<init>(ZipEntry.java:101)
        java.util.zip.ZipFile+3.nextElement(ZipFile.java:417)
        com.sun.tools.javac.jvm.ClassReader.openArchive(ClassReader.java:1374)
        com.sun.tools.javac.jvm.ClassReader.list(ClassReader.java:1631)
TRACE 301927:
        com.sun.tools.javac.util.List.<init>(List.java:42)
        com.sun.tools.javac.util.List.<init>(List.java:50)
        com.sun.tools.javac.util.ListBuffer.append(ListBuffer.java:94)
        com.sun.tools.javac.jvm.ClassReader.openArchive(ClassReader.java:1374)
TRACE 300996:
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:379)
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:481)
        com.sun.tools.javac.util.Name$Table.make(Name.java:332)
        com.sun.tools.javac.util.Name$Table.instance(Name.java:349)
TRACE 300995:
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:378)
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:481)
        com.sun.tools.javac.util.Name$Table.make(Name.java:332)
        com.sun.tools.javac.util.Name$Table.instance(Name.java:349)
```

线程堆栈的每一帧包含了类名，方法名，源代码文件和行号，用户可以设置最大帧的层数，缺省是4. 线程堆栈指明了是那个方法触发的内存分配。

**堆转储(heap=dump)** 堆转储是通过heap=dump选项获得的，该输出文件可以是ASCII也可以是二进制，取决于format选项的设置。如果这些输出文件要被jhat工具分析，可以通过format=b指定输出格式为二进制。当指定了二进制格式，转储文件包括原子类型字段和原子数组内容。

下面的转储片段是由javac编译器产生的：

```
$ javac -J-agentlib:hprof=heap=dump Hello.java
```

转储文件非常大，它包括了如下信息：

- 由垃圾收集器所分析的对象根集(root set)
- 对于每一个Java对象，从根集达到该对象的对象引用路径(entry)

下面是一个例子：

```
HEAP DUMP BEGIN (39793 objects, 2628264 bytes) Wed Oct 4 13:54:03 2006
ROOT 50000114 (kind=<thread>, id=200002, trace=300000)
ROOT 50000006 (kind=<JNI global ref>, id=8, trace=300000)
ROOT 50008c6f (kind=<Java stack>, thread=200000, frame=5)
:
CLS 50000006 (name=java.lang.annotation.Annotation, trace=300000)
    loader        90000001
OBJ 50000114 (sz=96, trace=300001, class=java.lang.Thread@50000106)
    name        50000116
    group        50008c6c
    contextClassLoader    50008c53
    inheritedAccessControlContext    50008c79
    blockerLock    50000115
OBJ 50008c6c (sz=48, trace=300000, class=java.lang.ThreadGroup@50000068)
    name        50008c7d
    threads    50008c7c
    groups        50008c7b
ARR 50008c6f (sz=16, trace=300000, nelems=1,
     elem type=java.lang.String[]@5000008e)
    [0]        500007a5
CLS 5000008e (name=java.lang.String[], trace=300000)
    super        50000012
    loader        90000001
:
HEAP DUMP END
```

每一个记录是一个根(Root). OBJ表示对象实例，CLS表示class，ARR表示数组。16进制数字是由HPROF分配的标识符，这些数字用来表示从一个对象到另一个对象的引用。例如，在上面的例子里java.lang.Thread实例50000114有一个到它的线程组(thread group:50008c6c)和另一个对象的引用。

一般情况下，这个输出文件非常大，非常有必要使用可视化工具（比如jhap）来阅读这个文件,详细请参考相关章节。

**CPU使用率采样剖析(cpu=samples)** HPROF工具可以通过对线程进行周期采样收集CPU的使用信息，下面是从运行javac编译器获得的采样信息的部分输出：

```
$ javac -J-agentlib:hprof=cpu=samples Hello.java
CPU SAMPLES BEGIN (total = 462) Wed Oct 4 13:33:07 2006
rank   self  accum   count trace method
   1 49.57% 49.57%     229 300187 java.util.zip.ZipFile.getNextEntry
   2  6.93% 56.49%      32 300190 java.util.zip.ZipEntry.initFields
   3  4.76% 61.26%      22 300122 java.lang.ClassLoader.defineClass2
   4  2.81% 64.07%      13 300188 java.util.zip.ZipFile.freeEntry
   5  1.95% 66.02%       9 300129 java.util.Vector.addElement
   6  1.73% 67.75%       8 300124 java.util.zip.ZipFile.getEntry
   7  1.52% 69.26%       7 300125 java.lang.ClassLoader.findBootstrapClass
   8  0.87% 70.13%       4 300172 com.sun.tools.javac.main.JavaCompiler.<init>
   9  0.65% 70.78%       3 300030 java.util.zip.ZipFile.open
  10  0.65% 71.43%       3 300175 com.sun.tools.javac.main.JavaCompiler.<init>

...
CPU SAMPLES END
```

HPROF代理周期性地对所有正在运行线程栈进行采样，并记录最活跃的线程栈。上面的count字段表示在采样时，特定的线程栈被命中为激活的次数。这些线程栈正是应用程序中的热点区。

**CPU使用时间剖析(cpu=times)** HPROF工具可以通过在每个方法进入和退出时注入代码来收集CPU的使用情况，因此可以知道每个方法的调用次数和运行的时间。这种方法叫做字节码注入(Byte Code Injection-BCI)，运行起来比cpu=samples要慢一些，下面是运行javac编译器获得的部分信息：

```
$ javac -J-agentlib:hprof=cpu=times Hello.java
CPU TIME (ms) BEGIN (total = 2082665289) Wed oct 4 13:43:42 2006
rank   self  accum   count trace method
   1  3.70%  3.70%       1 311243 com.sun.tools.javac.Main.compile
   2  3.64%  7.34%       1 311242 com.sun.tools.javac.main.Main.compile
   3  3.64% 10.97%       1 311241 com.sun.tools.javac.main.Main.compile
   4  3.11% 14.08%       1 311173 com.sun.tools.javac.main.JavaCompiler.compile
   5  2.54% 16.62%       8 306183 com.sun.tools.javac.jvm.ClassReader.listAll
   6  2.53% 19.15%      36 306182 com.sun.tools.javac.jvm.ClassReader.list
```

```
    7  2.03% 21.18%       1 307195 com.sun.tools.javac.comp.Enter.main
    8  2.03% 23.21%       1 307194 com.sun.tools.javac.comp.Enter.complete
    9  1.68% 24.90%       1 306392 com.sun.tools.javac.comp.Enter.classEnter
   10  1.68% 26.58%       1 306388 com.sun.tools.javac.comp.Enter.classEnter
...
CPU TIME (ms) END
```

这种信息收集精确地收集了一个方法被进入的次数，以及花费的CPU时间。

### 附录 B.2.2 Java VisualVM

VisualVM是随JDK一起发布的工具之一，该工具用来故障定位，以及监控提升应用程序性能。通过VisualVM可以产生堆栈转储(Heap Dump)并进行分析，分析性能和内存泄漏已经监控垃圾回收，以及进行轻量级的内存和CPU使用率剖析。同时，该工具在调整堆大小(Heap size),离线分析,事后分析非常有用。

同时，在VisualVM中，可以直接使用已经存在的pluginl来扩展VisualVM的能力,比如借助MBeans tab页和JConsole Wrapper tab页，JConsole中的大多数功能在VisualVM都是可用的。你可以通过在VisualVM的tool菜单中的VisualVM plugin列表中选择你要使用的功能。更详细的VisualVM文档在http://java.sun.com/javase/6/docs/technotes/guides/visualvm/index.html

通过VisualVM,可以执行如下的故障定位：

- 查看本地或者远程Java应用程序列表.
- 查看应用程序配置和运行期环境. 对于每一个应用程序，该工具可以显示基本的运行期信息: 进程id, host, main class, 进程启动参数, JVM 版本, JDK 主目录, JVM flags, JVM参数, 系统属性.
- 打开或者关闭针对应用程序遭遇OutOfMemoryError异常时的堆转储功能。
- 监控应用程序内存消耗，正在运行的线程，以及加载的类.
- 立即触发一次垃圾回收.
- 立即触发一次堆转储，可以从几个视角来观察堆转储文件：总结, 根据类, 根据实例. 同时可以保存堆转储到一个本地文件中.
- 剖析应用程序性能以及分析内存分配(仅适用于本地应用程序). 同时可以保存这些数据.
- 立即触发创建一个线程堆栈.
- 分析core转储文件(仅Solaris OS 和Linux下支持).
- 通过应用程序快照，离线分析应用程序.
- 获取由社区发布的plug-ins.
- 编写并共享你自己写的plug-ins.

- 与MBeans进行交互(需要安装MBeans tab plug-in).

当启动Java VisualVM,主应用程序窗口打开，显示本机运行正在运行的Java应用程序列表，以及连接到远程机器上的应用程序列表，以及任何虚拟机core转储文件列表（仅Solaris和Linux下支持），以及保存的应用程序快照。

Java VisualVM 将自动检测并链接运行在Java SE6.0上的应用程序的JMX代理，或者链接到通过正确参数启动的Java SE5.0上的JMX代理。为了确保该工具能够检测并链接到远程机器上的代理，远程机器必须运行jstatd。如果Java VisualVM 不能自动检测并链接，该工具也提供了通过显式创建链接的方式。

### 附录 B.2.3 JConsole Utility

在JDK中另一个很有用的工具是JConsole监控工具，这个工具与JMX兼容，通过JDK中内置的JMX指令与虚拟机进行交互获取应用程序的性能和资源消耗情况。

JConsole工具可以attach到Java SE应用程序，获取诸如线程，内存消耗，class加载，运行期编译，及操作系统等相关信息。这些信息的输出可以帮助分析内存泄漏，class过度加载，以及线程问题，同时这些信息对于调整heap大小也很有帮助。

除了监控功能外，JConsole还可以动态改变运行期参数。例如，通过设置-verbose:gc可以即时打开垃圾回收信息开关。下面的列表提供了JConsole的思路。

- Overview

  这个pane以图像化的方式显示堆内存使用，线程数量，类的数量，以及CPU的使用情况。

- Memory
  - 对于选定的内存区域(堆内存, 非堆内存, 以及各种各样的内存池)，JConsole可以显示:
    * 整个时间段内，内存使用的情况
    * 当前内存的大小
    * Amount of committed memory
    * 最大内存大小
  - 垃圾收集信息，包括垃圾收集的次数，垃圾收集总共花费的时间
  - 当前堆内存或者非堆内存的使用百分比。

    另外，在这一个Tab页，你还可以强制马上执行垃圾回收操作。

- Threads
  - 整个时间段内，线程的使用情况
  - 活动线程-当前活动线程的数量。
  - 峰值k - 从虚拟机启动开始，活动线程的最大峰值数量。

  - 选定线程的线程名称，调用堆栈。同时，对于阻塞线程，可以显示该线程正在请求或者占有的锁。
  - 死锁检测按钮- 发送请求到目标应用程序，执行死锁检测。

- Classes
  - 整个时间段内加载类的数量图。
  - 当前加载到内存中类的数量.
  - 从虚拟机启动到现在，加载到内存所有类的总数, 包括期间又被卸载的类.
  - 从虚拟机启动以来，卸载类的总数。

- VM Summary
  - 一般信息，比如JConsole链接, 虚拟机启动的时长, 编译器名称，总的编译时间等等.
  - 线程和类总结信息.
  - 内存和垃圾收集信息，包括finalizaion挂起的对象数量等等.
  - 操作系统信息，包括物理特点，正在运行进程的虚拟内存数量，交换空间等等.
  - 虚拟机自身信息，如运行期参数，类路径等等.

- MBeans

  这个页面显示一个树状结构，列出所有注册到链接到JMS代理上的平台和应用MBeans. 当你选择树中的一个MBean,它的属性，操作，通知以及其它信息将被显示。

  - 你可以调用任何操作，如用来HotSpotDianostic（热点诊断）的Mbean上的dumpHeap,该操作位于com.sun.management域之内，执行堆转储，输入参数是目标虚拟机所在机器上的堆转储文件的路径
  - 作的另一个例子，是你可以设置可写属性的值，例如，你可以通过HotSpotDiagnostic MBean上的setVMOption操作来设置，取消设置修改特定虚拟机flags的值。
  - 你可以通过Subscribe和Unsubscribe按钮订阅通知.

JConsole可以监控本地或者远程程序，如果你通过一个参数指定要链接的JMX代理启动该工具，该工具将自动监控指定的应用。监控本地应用程序，直接通过执行命令jconsole pid,其中pid是进程id. 监控远程程序，通过执行命令jconsole hostname:portnumber，hostname为远程机器的名称(或者ip),portnumber是JMX代理的端口号。如果直接输入不带任何参数的jconsole，该工具显示一个新链接窗口，通过该窗口中的菜单，你可以选择要链接的本地或者远程程序。

在J2SE 1.5,必须通过-Dcom.sun.management.jmxremote选项启动要被监控的应用程序。在Java SE 6中，不需要指定任何选项。

作为一个监控工具的列子，下面展示了一个堆内存使用的图表:

该指南不是一个完整的JConsole 指导，更详细信息请参考：

- 监控和管理Java应用程序：http://java.sun.com/javase/6/docs/technotes/guides/management/index.html

- 通过JMX监控和管理：http://java.sun.com/javase/6/docs/technotes/guides/management/agent.html
- 使用JConsole: http://java.sun.com/javase/6/docs/technotes/guides/management/jconsole.html
- jconsole手册：http://java.sun.com/javase/6/docs/technotes/tools/share/jconsole.html

### 附录 B.2.4 jdb Utility

gdb包含在JDK中，作为命令行debugger的一个例子，该工具使用Java Debug接口(JDI)启动或者链接到目标JVM上，jdb的源代码放在$JAVA_HOME/demo/jpda/examples.jar中。Java Debug Interface(JDI)是一个高层Java API,它给调试器提供有用的信息，JDI是Java Platform Debugger Architecture (JPDA)的一个组件。在JDI中，通过连接器可以将调试器链接到目标虚拟机中，同样连接器可以用作远程调试（通过TCP/IP或者共享内存传输），

在Solaris上，JDK同时发布了几个可服务性代理连接器(several Serviceability Agent (SA)),通过这些代理可以将调试器attach到崩溃转储文件或者挂起的进程上，在确定系统崩溃或者挂起时，系统正在做什么是非常有用的。这些可服务性代理连接器在windows上或者Linux上是不可用的，这些连接器是：SACoreAttachingConnector, SADebugServerAttachingConnector, and SAPIDAttachingConnector.

这些连接器一般是随着企业版的调试器使用，比如NetBeans IDE 或者其他商业IDEs，下面将介绍如果与jdb命令行调试器中使用连接器。

连接器的相信信息请参考：
http://java.sun.com/javase/6/docs/technotes/guides/jpda/conninv.html#Connectors. 命令jdb -listconnectors 可以打印出可用的连接器. 更详细的信息请参考jdb用户手册，如：

- Solaris OS and Linux: jdb man page

  http://java.sun.com/javase/6/docs/technotes/tools/solaris/jdb.html

- Windows: jdb man page

  http://java.sun.com/javase/6/docs/technotes/tools/windows/jdb.html

**Attaching to a Process** 下面这个jdb的例子采用了将SA PID绑定连接器到一个进程上，目标进程实际上是不需要使用特别的选项来启动，即使-agentlib:jdwp选项是不需要的。当连接器绑定到jvm进程时，它将进入只读模式，这样调试器可以测试线程以及正在运行的程序，但是它不能改变任何东西，当调试器绑定上之后，进程将被frozen.下面的例子中的命令指导jdb使用一个名字为sun.jvm.hotspot.jdi.SAPIDAttachingConnector的连接器，这是一个连接器的名字，而不是一个类名，连接器带一个pid的参数，你目标进程的pid(本例中为9302)

```
$ jdb -connect sun.jvm.hotspot.jdi.SAPIDAttachingConnector:pid=9302

Initializing jdb ...
> threads
Group system:
  (java.lang.ref.Reference$ReferenceHandler)0xa Reference Handler unknown
```

```
  (java.lang.ref.Finalizer$FinalizerThread)0x9  Finalizer         unknown
  (java.lang.Thread)0x8                         Signal Dispatcher running
  (java.lang.Thread)0x7                         Java2D Disposer   unknown
  (java.lang.Thread)0x2                         TimerQueue        unknown
Group main:
  (java.lang.Thread)0x6                         AWT-XAWT          running
  (java.lang.Thread)0x5                         AWT-Shutdown      unknown
  (java.awt.EventDispatchThread)0x4             AWT-EventQueue-0  unknown
  (java.lang.Thread)0x3                         DestroyJavaVM     running
  (sun.awt.image.ImageFetcher)0x1               Image Animator 0  sleeping
  (java.lang.Thread)0x0                         Intro             running
> thread 0x7
Java2D Disposer[1] where
  [1] java.lang.Object.wait (native method)
  [2] java.lang.ref.ReferenceQueue.remove (ReferenceQueue.java:116)
  [3] java.lang.ref.ReferenceQueue.remove (ReferenceQueue.java:132)
  [4] sun.java2d.Disposer.run (Disposer.java:125)
  [5] java.lang.Thread.run (Thread.java:619)
Java2D Disposer[1] up 1
Java2D Disposer[2] where
  [2] java.lang.ref.ReferenceQueue.remove (ReferenceQueue.java:116)
  [3] java.lang.ref.ReferenceQueue.remove (ReferenceQueue.java:132)
  [4] sun.java2d.Disposer.run (Disposer.java:125)
  [5] java.lang.Thread.run (Thread.java:619)
```

在这个例子中，threads命令用来获取进程的当前线程列表，然后一个特定的线程0x7被选择，thread 0x7用来获得该0x7线程的调用栈，up 1用来上移一个帧，where用来重新获取线程调用栈。

**Attaching to a Core File on the Same Machine** SA Core用来将调试器绑定到一个core文件上，core文件也许系统崩溃时创建的，在Solaris可以通过gcore，Linux上通过gdb中的gcore命令获得。因为core文件是系统当时的一个快照，因此链接系是以只读方式绑定的，调试器可以测试系统core文件产生时的线程。

下面是一个例子：

```
$ jdb -connect sun.jvm.hotspot.jdi.SACoreAttachingConnector:\
  javaExecutable=$JAVA_HOME/bin/java,core=core.20441
```

该命令指明jdb采用名字为sun.jvm.hotspot.jdi.SACoreAttachingConnector的连接器,连接器的参数为javaExecutable和core文件的名称，javaExecutable参数指明java二进制库的名称，core参数为core文件的名称（在这个例子中，core文件的名字为core.20441）

**Attaching to a Core File or a Hung Process from a Different Machine** 为了调试一个从其它机器上传来的core文件，OS版本和库版本必须是匹配的。这种情况下，你可以首先运行一个称作SA Debug Server的proxy server, 然后，在安装调试器的机器上，你可以通过SA debug server连接器链接到debug server上。在下面的例子中，有两个机器，机器1和机器2,core文件在机器1上，调试器在机器2上，在机器1上按照如下方式启动SA Debug server：

```
$ jsadebugd $JAVA_HOME/bin/java core.20441
```

jsadebugd命令有两个参数.第一个是可执行程序的名字. 大多数情况下就是java, 但它也可以是其它名字（比如内嵌的VM虚拟机），第二个参数是core文件的名字，这个例子中core文件的名字是core.20441。在机器2中，调试器使用SA Debug Server Attaching Connector链接到远程SA Debug Server上，命令如下：

```
$ jdb -connect sun.jvm.hotspot.jdi.SADebugServerAttachingConnector:debugServerName=machine1
```

这个命令指示jdb采用名字为sun.jvm.hotspot.jdi.SADebugServerAttachingConnector的连接器进行连接. 连接器有一个参数debugServerName,即SA Debug Server所运行的机器名称或者IP.

注意，SA Debug Server 也可以用作远程调试挂起的进程，在这种情况下，他只带一个参数，即这个进程的进程id, 另外，如果在同一个机器上运行多个SA Debug Server ，每一个必须提供系统唯一的ID,在SA Debug Server Attaching Connector连接器上，ID作为一个附件参数，细节请参考JPDA文档。

### 附录 B.2.5 jhat Utility

通过jhat工具，可以很方便地浏览堆快照中对象拓扑，这个工具是Java SE 6中新引入的，用来代替堆分析工具(Heap Analysis Tool-HAT). 关于HAT工具，可以参考J2se5.0中的故障诊断和处理指导（Troubleshooting and Diagnostic Guide）。

更详细的jhat帮助，请参考jhat的手册页(jhat- Java Heap Analysis Tool). 这个工具用来解析二进制格式的堆转储文件，例如，通过jmap -dump产生的堆转储文件。这个工具可以帮助分析不期望的对象持有，即那些本来已经不再需要，但是仍然被保持的对象，即我们所称的内存泄漏。这个工具提供了一个标准的查询，根查询显示所有从根集(rootset)到指定对象的引用路径，这在分析不期望的对象持有特别有用。除了这个标准的功能之外，你可以通过对象查询语言接口（the Object Query Language (OQL) interface）来开发自己的定制查询。

当你启动jhat命令，这个工具在指定的端口上启动HTTP server,你可以通过浏览器链接到这个server上，在指定的堆转储上进行查询。下面的例子展示了如何分析名字为snapshot.hprof的堆转储文件。

```
$ jhat snapshot.hprof
Started HTTP server on port 7000
Reading from java_pid2278.hprof...
Dump file created Fri May 19 17:18:38 BST 2006
Snapshot read, resolving...
```

```
Resolving 6162194 objects...
Chasing references, expect 12324 dots..............................
Eliminating duplicate references...................................
Snapshot resolved.
Server is ready.
```

上面输出表示，jhat在7000端口上启动了一个http server,可以通过在浏览器中输入地址：http://localhost:7000链接到该http server上。一旦连接到jhat server,就可以执行标准查询命令或者定制查询。

### 标准查询

- 所有被加载的类。

  缺省页面中显示出去平台的类之外所有的类，按照类名排序，通过点击类名可以进入类查询，第二种查询方式，可以包含平台类，如java. sun. javax.swint. char[(字符数组)开头的类。

- Class Query

  类查询显示类的信息，包括它的父类和子类，数据成员，静态数据成员，在该页中，可以查看该类引用的任意的类。

- Object Query

  对象查询提供堆中对象的信息，可以查看该对象的类，对象成员的值，以及引用到当前对象的对象，最用的是根对象查询，即根集引用链。

  同时，对象查询还提供该对象分配点的调用堆栈跟踪信息（backtrace）。

- Instances Query

  实例查询可以显示一个给定类的所有实例，同时包括父类的实例数量，这样可以后向跟踪源类。

- Roots Query

  根集查询显示一个给定对象的根集引用链，它提供了从指定对象可达的根集引用链（即那个对象引用了这个对象）。该同居通过深度优先搜索，提供最小长度的引用链。

  有两种类型的根集查询：一种为不包括弱引用(Roots)，另一种为包括弱引用(所有的根)。There are two kinds of Roots query: one that excludes weak references (Roots), and one that includes them (All Roots). A weak reference is a reference object that does not prevent its referent from being made finalizable, finalized, and then reclaimed. If an object is only referred to by a weak reference, it usually isn't considered to be retained, because the garbage collector can collect it as soon as it needs the space.

  This is probably the most valuable query in jhat for debugging unintentional object retention. Once you find an object that is being retained, this query tells you why it is being retained.

- Reachable Objects Query

  可达对象查询显示从一个指定对象到所有对象的传递闭包，这在运行期分析内存非常有用，只是它提供了一个简单的对象topo关系图。

- Instance Counts for All Classes Query

  所有类的实例数查询显示该系统中每一个类的实例数。定位一个系统内存泄漏一个非常有效的方法是，长时间运行程序，然后请求一次堆转储，查看所有类的实例数量，可以很容易识别出那些类的实例数量超大，从而进一步分析是否系统存在内存泄漏，比如通过根集的引用关系确定这些实例是否被意外引用？

- All Roots Query 所有根集查询显示根集的所有成员

- New Instances Query

  新实例查询当你在调用jhat两次堆转储是有用，类似于实例数查询，只是它只显示第二次新创建的实例。

- Histogram Queries

  内置的柱状图查询也可以提供有用的信息

**定制查询** You can develop your own custom queries with the built-in Object Query Language (OQL) interface. Click on the Execute OQL Query button on the first page to display the OQL query page, where you can create and execute your custom queries. The OQL Help facility describes the built-in functions, with examples.

The syntax of the select statement is as follows:

```
select JavaScript-expression-to-select
   [ from [instanceof] classname identifier
   [ where JavaScript-boolean-expression-to-filter ] ]
```

The following is an example of a select statement:

```
select s from java.lang.String s where s.count >= 100
```

**堆分析提示** 从jhat分析中获取有用的信息，往往需要一些背景知识，比如关于该进程使用的库以及它使用的API，一般情况下，该工具可以回答下面两个重要问题：

- 那些对象是活着的?

  当查看对象实例时，你可以在“References to this object”部分中检查列出的对象，看看是哪些对象引用了这个对象。更重要的是，你可以使用Root查询来确定从根集到指定对象的引用链，这些引用链显示了从根对象到这个对象之间的引用路径，通过这个引用链，你可以快速确定一个对象是怎样从根集被引用下来的。

As noted earlier, there are two kinds of Roots queries: one that excludes weak references (Roots), and one that includes them (All Roots). A weak reference is a reference object that does not prevent its referent from being made finalizable, finalized, and then reclaimed. If an object is only referred to by a weak reference, it usually is not considered to be retained, because the garbage collector can collect it as soon as it needs the space.

jhat工具可以按照如下方式对引用链进行排序：

- Java类的静态数据成员.
- 本地变量，对于root而言，负责他们的线程将被显示。因为一个线程即是一个Java对象，这个链接是可触及的，这允许你可以很容易地知道线程的名字
- 本地静态值.
- 本地局部变量. 同样，roots是由他们的线程来标识的。

• 这些对象是在哪里被分配的?

当对象实例被显示时，有一个标题为"Objects allocated from"的部分显示了该实例在调用栈中的分配点，根据此信息，可以看到该对象是在哪里被创建的。注意，仅当heap=all选项打开时收集到的HPROF堆转储才能看到分配点信息。

当通过单次对象转储不能标识出泄漏点时，可以通过一系列的转储，重点关注每一次相比上一次新创建的对象上面，jhat工具通过-baseline选项提供了这种能力。-baseline选项允许两次转储进行比较，如果同一个对象同时出现在两次转储中，则不在新对象报告中。第一次转储作为基线，将分析的重点放在基线获取后的转储中新创建的对象。

使用方法如下：

```
$ jhat -baseline snapshot.hprof#1 snapshot.hprof#2
```

在上面的例子中，在文件snapshot.hprof有两个转储，他们通过#1 and #2 来区分

当jhat是以两次堆转储的方式启动，对所有类的实例查询包括一个附加列，这个附加列是该类的新实例的个数。当在基线转储中不存在，但是在第二次转储中存在的实例即认为是该类的新实例。对每一个实例，你可以看到它在哪里被分配的，该对象引用的其它对象有哪些，以及被哪些对象所引用。

一般情况下，当在两次连续转储中，想了解这段时间间隔内新创建的对象，-baseline选项非常有用。

### 附录 B.2.6 jinfo Utility

jinfo命令行可以获取正在运行的Java进程或者崩溃转储文件的相关配置信息，打印系统属性已经启动虚拟机的命令行参数。借助jsadebugd daemon也可以获取远程机器上的信息。通过-flag选项，可以动态改变正在运行虚拟机的参数。

例如：

```
$ jinfo 29620
Attaching to process ID 29620, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100
Java System Properties:

java.runtime.name = Java(TM) SE Runtime Environment
sun.boot.library.path = /usr/jdk/instances/jdk1.6.0/jre/lib/sparc
java.vm.version = 1.6.0-rc-b100
java.vm.vendor = Sun Microsystems Inc.
java.vendor.url = http://java.sun.com/
path.separator = :
java.vm.name = Java HotSpot(TM) Client VM
file.encoding.pkg = sun.io
sun.java.launcher = SUN_STANDARD
sun.os.patch.level = unknown
java.vm.specification.name = Java Virtual Machine Specification
user.dir = /home/js159705
java.runtime.version = 1.6.0-rc-b100
java.awt.graphicsenv = sun.awt.X11GraphicsEnvironment
java.endorsed.dirs = /usr/jdk/instances/jdk1.6.0/jre/lib/endorsed
os.arch = sparc
java.io.tmpdir = /var/tmp/
line.separator =

java.vm.specification.vendor = Sun Microsystems Inc.
os.name = SunOS
sun.jnu.encoding = ISO646-US
java.library.path = /usr/jdk/instances/jdk1.6.0/jre/lib/sparc/client:
/usr/jdk/instances/jdk1.6.0/jre/lib/sparc:/usr/jdk/instances/jdk1.6.0/jre/../lib/sparc:
/net/gtee.sfbay/usr/sge/sge6/lib/sol-sparc64:/usr/jdk/packages/lib/sparc:/lib:/usr/lib
java.specification.name = Java Platform API Specification
java.class.version = 50.0
sun.management.compiler = HotSpot Client Compiler
os.version = 5.10
user.home = /home/js159705
user.timezone = US/Pacific
java.awt.printerjob = sun.print.PSPrinterJob
file.encoding = ISO646-US
java.specification.version = 1.6
java.class.path = /usr/jdk/jdk1.6.0/demo/jfc/Java2D/Java2Demo.jar
```

```
user.name = js159705
java.vm.specification.version = 1.0
java.home = /usr/jdk/instances/jdk1.6.0/jre
sun.arch.data.model = 32
user.language = en
java.specification.vendor = Sun Microsystems Inc.
java.vm.info = mixed mode, sharing
java.version = 1.6.0-rc
java.ext.dirs = /usr/jdk/instances/jdk1.6.0/jre/lib/ext:/usr/jdk/packages/lib/ext
sun.boot.class.path = /usr/jdk/instances/jdk1.6.0/jre/lib/resources.jar:
/usr/jdk/instances/jdk1.6.0/jre/lib/rt.jar:/usr/jdk/instances/jdk1.6.0/jre/lib/sunrsasign.jar:
/usr/jdk/instances/jdk1.6.0/jre/lib/jsse.jar:
/usr/jdk/instances/jdk1.6.0/jre/lib/jce.jar:/usr/jdk/instances/jdk1.6.0/jre/lib/charsets.jar:
/usr/jdk/instances/jdk1.6.0/jre/classes
java.vendor = Sun Microsystems Inc.
file.separator = /
java.vendor.url.bug = http://java.sun.com/cgi-bin/bugreport.cgi
sun.io.unicode.encoding = UnicodeBig
sun.cpu.endian = big
sun.cpu.isalist =
```

如果你启动的虚拟机采用了-classpath 和-Xbootclasspath 选项, jinfo 能够输出for java.class.path 和sun.boot.class.path，这在定位class Loader非常有用。

同时jinfo可以采用core文件做为输入。Solaris OS, gcore可以或者一个正在运行进程的core文件，在上面的例子中core文件名字为core.29620，在jinfo中必须同时制定java可执行程序和core文件的名字。如：

```
$ jinfo $JAVA_HOME/bin/java core.29620
```

有的时候，而进程程序的名字并不是java,比如当虚拟机是在JNI中被启动的话。

### 附录 B.2.7 jmap Utility

jmap可以打印core文件或者正在运行的JVM的内存统计相关信息。该工具也可以使用jsadebugd daemon来请求远程机器上的进程或者core文件。如果运行jmap没有任何选项，它打印装载的共享对象的列表（输出与Solaris上的pmap相似）,对于一些特别信息，可以通过选项-heap, -histo, or -permstat获得,下面详细描述这些选项。

另外，Java SE 6 引入了-dump:format=b,file=filename 选项，该选项可以将Java堆以二进制的方式打印到指定的文件中，然后可以通过jhat对该文件进行分析。如果由于进程挂起而导致jmap pid命令没有任何响应，可以通过-F选项（仅Solaris和Linux支持）强迫使用Serviceability代理。该工具随JDK版本一起发布，在windows下的JDK 6中也包括该工具，但只支持jmap -dump:format=b,file=file pid 与jmap -histo[:live] pid. 更详细的信息，请参考手册。

**Heap Configuration and Usage** -heap 选项用来获取如下的Java堆信息：

- 垃圾收集算法的信息，包括GC算法的名字（比如并行垃圾回收算法）以及特定细节（比如并行GC的线程数量）
- 堆配置信息
- 堆使用总结，针对每一个堆区域，该工具打印总的堆容量，在使用的内存以及可用内存，如果一个区域被作为收集区域（如新生代），相应的内存大小的总结也会被打印出来。

下面的例子显示了jmap -heap命令的输出：

```
$ jmap -heap 29620
Attaching to process ID 29620, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100

using thread-local object allocation.
Mark Sweep Compact GC

Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70
   MaxHeapSize      = 67108864 (64.0MB)
   NewSize          = 2228224 (2.125MB)
   MaxNewSize       = 4294901760 (4095.9375MB)
   OldSize          = 4194304 (4.0MB)
   NewRatio         = 8
   SurvivorRatio    = 8
   PermSize         = 12582912 (12.0MB)
   MaxPermSize      = 67108864 (64.0MB)

Heap Usage:
New Generation (Eden + 1 Survivor Space):
   capacity = 2031616 (1.9375MB)
   used     = 70984 (0.06769561767578125MB)
   free     = 1960632 (1.8698043823242188MB)
   3.4939673639112905% used
Eden Space:
   capacity = 1835008 (1.75MB)
   used     = 36152 (0.034477233886718750MB)
   free     = 1798856 (1.7155227661132812MB)
   1.9701276506696428% used
```

```
From Space:
   capacity = 196608 (0.1875MB)
   used     = 34832 (0.0332183837890625MB)
   free     = 161776 (0.154281616210937 5MB)
   17.716471354166668% used
To Space:
   capacity = 196608 (0.1875MB)
   used     = 0 (0.0MB)
   free     = 196608 (0.1875MB)
   0.0% used
tenured generation:
   capacity = 15966208 (15.2265625MB)
   used     = 9577760 (9.134063720703125MB)
   free     = 6388448 (6.092498779296875MB)
   59.98769400974859% used
Perm Generation:
   capacity = 12582912 (12.0MB)
   used     = 1469408 (1.401336669921875MB)
   free     = 11113504 (10.598663330078125MB)
   11.677805582682291% used
```

**Heap Histogram of Running Process** -histo 选项可以获取相关类的柱状图。当在正在运行的进程上执行该命令时，该工具将打印对象的数量，内存大小，以及类名。内部类使用尖括号括起来，柱状对对分析堆是如何使用的非常有用，比如通过一个类的对象占用总内存除以该类型对象的数量可以获得一个对象的大小。下面例子显示了对正在运行的进程执行jmap -histo命令的结果。

```
$ jmap -histo 29620
num   #instances    #bytes  class name
--------------------------------------
  1:      1414     6013016  [I
  2:       793      482888  [B
  3:      2502      334928  <constMethodKlass>
  4:       280      274976  <instanceKlassKlass>
  5:       324      227152  [D
  6:      2502      200896  <methodKlass>
  7:      2094      187496  [C
  8:       280      172248  <constantPoolKlass>
  9:      3767      139000  [Ljava.lang.Object;
 10:       260      122416  <constantPoolCacheKlass>
 11:      3304      112864  <symbolKlass>
 12:       160       72960  java2d.Tools$3
 13:       192       61440  <objArrayKlassKlass>
```

```
 14:       219       55640  [F
 15:      2114       50736  java.lang.String
 16:      2079       49896  java.util.HashMap$Entry
 17:       528       48344  [S
 18:      1940       46560  java.util.Hashtable$Entry
 19:       481       46176  java.lang.Class
 20:        92       43424  javax.swing.plaf.metal.MetalScrollButton
... more lines removed here to reduce output...
1118:         1           8  java.util.Hashtable$EmptyIterator
1119:         1           8  sun.java2d.pipe.SolidTextRenderer
Total    61297    10152040
```

**Heap Histogram of Core File** 当在core文件上执行-histo命令时，该工具打印每一个类的对象的数量，大小，类名，内部类使用*作为前缀。

```
& jmap -histo /net/onestop/jdk/6.0/promoted/all/b100/binaries/solaris-sparcv9/bin/java core
Attaching to core core from executable /net/koori.sfbay/onestop/jdk/6.0/
promoted/all/b100/binaries/solaris-sparcv9/bin/java, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 1.6.0-rc-b100
Iterating over heap. This may take a while...
Heap traversal took 8.902 seconds.

Object Histogram:

Size    Count    Class description
-------------------------------------------------------
4151816    2941    int[]
2997816    26403    * ConstMethodKlass
2118728    26403    * MethodKlass
1613184    39750    * SymbolKlass
1268896    2011    * ConstantPoolKlass
1097040    2011    * InstanceKlassKlass
882048    1906    * ConstantPoolCacheKlass
758424    7572    char[]
733776    2518    byte[]
252240    3260    short[]
214944    2239    java.lang.Class
177448    3341    * System ObjArray
176832    7368    java.lang.String
137792    3756    java.lang.Object[]
121744    74    long[]
```

```
72960    160    java2d.Tools$3
63680    199    * ObjArrayKlassKlass
53264    158    float[]
... more lines removed here to reduce output...
```

**Getting Information on the Permanent Generation** 永久区是指虚拟机自身放置反射数据的区域，比如类，方法对象（也称作方法区）,这个区同时放置内部字符串。对于可能动态产生或者加载大量类的应用程序（比如，JSP页面或者web容器），设置永久区的大小是非常重要的。如果一个应用程序加载过多的类，或者内部字符串，可能会产生OutOfMemoryError错误，错误的格式为"in thread XXXX java.lang.OutOfMemoryError: PermGen space." 详细请参考第261页第 附录 B.3.1节. 通过使用-permstat选项可以打印永久区的对象统计信息，例子如下：

```
$ jmap -permstat 29620
Attaching to process ID 29620, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100
12674 intern Strings occupying 1082616 bytes.
finding class loader instances ..Unknown oop at 0xd0400900
Oop's klass is 0xd0bf8408
Unknown oop at 0xd0401100
Oop's klass is null
done.
computing per loader stat ..done.
please wait.. computing liveness.......................................done.
class_loader    classes bytes   parent_loader   alive?  type

<bootstrap>      1846 5321080  null        live   <internal>
0xd0bf3828  0     0      null         live    sun/misc/Launcher$ExtClassLoader@0xd8c98c78
0xd0d2f370  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0c99280  1   1440      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0b71d90  0     0   0xd0b5b9c0    live java/util/ResourceBundle$RBClassLoader@0xd8d042e8
0xd0d2f4c0  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0b5bf98  1    920   0xd0b5bf38      dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0c99248  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f488  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0b5bf38  6   11832  0xd0b5b9c0      dead    sun/reflect/misc/MethodUtil@0xd8e8e560
0xd0d2f338  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f418  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f3a8  1    904     null          dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0b5b9c0  317  1397448 0xd0bf3828      live    sun/misc/Launcher$AppClassLoader@0xd8cb83d8
0xd0d2f300  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f3e0  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
```

```
0xd0ec3968  1   1440      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0e0a248  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0c99210  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f450  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f4f8  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0e0a280  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50

total = 22      2186    6746816   N/A   alive=4, dead=18       N/A
```

对每一个类加载对象，包括了如下的细节信息：

- 工具正在运行时，快照时刻类加载的地址。
- 被加载类的数量。
- 这个类加载器所加载的所有类的元数据所占用的近似字节数。
- 父类加载器的地址（如果有的话）
- “live”或者“dead”指示该加载对象将来是否会被垃圾收集。
- 类名。

### 附录 B.2.8 jps Utility

jps工具用来列出目标系统上的当前用户启动的虚拟机，特别当虚拟机是内嵌的，即虚拟机是通过JNI被启动的而不是通过java启动器启动的（即java命令行），这个工具非常用用，在这种内置启动虚拟机的情况下，通常是不容易在进程列表中识别出虚拟机。

下面是一个例子：

```
$ jps
16217 MyApplication
16342 jps
```

这个工具列出该用户有存储权限的所有虚拟机，具体是否有存取权限，依赖于操作系统的权限机制，在Solaris上，如果非root用户使用该工具，只能列出该用户id启动的虚拟机。除了列出进程id, 该工具同时还提供了选项输出传给引用程序main方法的参数,已经应用程序main class的全包名。如果jstatd运行在远程机器上，该jps工具还可以列出远程机器上的java进程。

如果一个机器上运行了几个通过Web启动的虚拟机，显示可能如下：

```
$ jps
1271 jps
     1269 Main
     1190 Main
```

在这种情况下，jps -m可以对他们进行区分：

```
$ jps -m
1271 jps -m
     1269 Main http://bugster.central.sun.com/bugster.jnlp
     1190 Main http://webbugs.sfbay/IncidentManager/incident.jnlp
```

该工具随JDK一起发布.

---

注意：这些指令在Windows 98和Windows ME上是不可用的，同时，在采用了FAT32的Windows NT, 2000, 或者XP也是不可用的。

---

### 附录 B.2.9 jrunscript Utility

jrunscript是一个命令行脚本的shell,它支持交互模式或者批处理模式执行脚本，缺省情况下，该工具使用JavaScript,但是你也可以执行其他脚本语言。详细信息请参考相关手册。

### 附录 B.2.10 jsadebugd Daemon

Serviceability Agent Debug Daemon (jsadebugd)可以绑定到一个进程上，或者core文件上。该工具目前仅Solaris OS 和Linux上可用，远程客户端如jstack, jmap, 和jinfo 可以绑定到采用RMI方式的服务器上。

详细信息请参考相关手册。

### 附录 B.2.11 jstack Utility

jstack命令可以绑定到指定的进程或者core文件，并打印所有线程的栈跟踪，包括java线程和虚拟机内部线程，该工具同样可以检测死锁。该工具同样可以使用jsadebugd daemon查询远程机器上的进程或者core文件，不过需要注意的是，这种情况下输出需要更长的时间。所有线程的线程跟踪在诊断诸如死锁以及挂起的问题非常有用。

该工具直接包含在Solaris的操作系统中，或者JDK的Linux版本中，同样在windows上的JDK 6也包含该工具，只是仅提供了jstack pid 和jstack -l pid 选项。在Java SE 6中引入了-l选项，该选项侵入工具查看堆中的owable的同步器（synchronizers）以及答应关于ava.util.concurrent.locks的信息，如果不使用该选项，线程转储进包含监视器(monitors)的信息。

在Java SE 6中，jstack pid 的输出等同于在控制台中输入Ctrl-
或者通过kill -3(windows下Ctrl-Break)发送一个QUIT信号给JVM进程线程转储同样也可以通过可编程Thread.getAllStackTraces Java接口进行输出，或者在debubber中通过各种选项进行输出(如jdb)

**强行调用栈转储** 当由于进程挂起，而导致jstack pid命令没有任何响应时，可以使用-F选项强行进行栈转储(仅Solaris和Linux支持)，如：

```
$ jstack -F 8321
Attaching to process ID 8321, please wait...
```

```
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100
Deadlock Detection:

Found one Java-level deadlock:
=============================

"Thread2":
  waiting to lock Monitor@0x000af398 (Object@0xf819aa10, a java/lang/String),
  which is held by "Thread1"
"Thread1":
  waiting to lock Monitor@0x000af400 (Object@0xf819aa48, a java/lang/String),
  which is held by "Thread2"

Found a total of 1 deadlock.

Thread t@2: (state = BLOCKED)

Thread t@11: (state = BLOCKED)
 - Deadlock$DeadlockMakerThread.run() @bci=108, line=32 (Interpreted frame)

Thread t@10: (state = BLOCKED)
 - Deadlock$DeadlockMakerThread.run() @bci=108, line=32 (Interpreted frame)

Thread t@6: (state = BLOCKED)

Thread t@5: (state = BLOCKED)
 - java.lang.Object.wait(long) @bci=-1107318896 (Interpreted frame)
 - java.lang.Object.wait(long) @bci=0 (Interpreted frame)
 - java.lang.ref.ReferenceQueue.remove(long) @bci=44, line=116 (Interpreted frame)
 - java.lang.ref.ReferenceQueue.remove() @bci=2, line=132 (Interpreted frame)
 - java.lang.ref.Finalizer$FinalizerThread.run() @bci=3, line=159 (Interpreted frame)

Thread t@4: (state = BLOCKED)
 - java.lang.Object.wait(long) @bci=0 (Interpreted frame)
 - java.lang.Object.wait(long) @bci=0 (Interpreted frame)
 - java.lang.Object.wait() @bci=2, line=485 (Interpreted frame)
 - java.lang.ref.Reference$ReferenceHandler.run() @bci=46, line=116 (Interpreted frame)
```

**从core转储中打印调用栈** 命令如下：

```
$ jstack $JAVA_HOME/bin/java core
```

**打印混合调用栈** jstack可以打印混合调用栈，既可以打印本地方法调用栈，同时还可以打印Java调用栈，本地栈是VM代码或者JNI C/C++调用栈。

采用-m选项，例如：To print a mixed stack, use the -m option, as in the following example:

```
$ jstack -m 21177
Attaching to process ID 21177, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100
Deadlock Detection:

Found one Java-level deadlock:
=============================

"Thread1":
  waiting to lock Monitor@0x0005c750 (Object@0xd4405938, a java/lang/String),
  which is held by "Thread2"
"Thread2":
  waiting to lock Monitor@0x0005c6e8 (Object@0xd4405900, a java/lang/String),
  which is held by "Thread1"

Found a total of 1 deadlock.

----------------- t@1 -----------------
0xff2c0fbc    __lwp_wait + 0x4
0xff2bc9bc    _thrp_join + 0x34
0xff2bcb28    thr_join + 0x10
0x00018a04    ContinueInNewThread + 0x30
0x00012480    main + 0xeb0
0x000111a0    _start + 0x108
----------------- t@2 -----------------
0xff2c1070    ___lwp_cond_wait + 0x4
0xfec03638    bool Monitor::wait(bool,long) + 0x420
0xfec9e2c8    bool Threads::destroy_vm() + 0xa4
0xfe93ad5c    jni_DestroyJavaVM + 0x1bc
0x00013ac0    JavaMain + 0x1600
0xff2bfd9c    _lwp_start
----------------- t@3 -----------------
0xff2c1070    ___lwp_cond_wait + 0x4
0xff2ac104    _lwp_cond_timedwait + 0x1c
0xfec034f4    bool Monitor::wait(bool,long) + 0x2dc
0xfece60bc    void VMThread::loop() + 0x1b8
0xfe8b66a4    void VMThread::run() + 0x98
```

```
0xfec139f4    java_start + 0x118
0xff2bfd9c    _lwp_start
----------------- t@4 -----------------
0xff2c1070    ___lwp_cond_wait + 0x4
0xfec195e8    void os::PlatformEvent::park() + 0xf0
0xfec88464    void ObjectMonitor::wait(long long,bool,Thread*) + 0x548
0xfe8cb974    void ObjectSynchronizer::wait(Handle,long long,Thread*) + 0x148
0xfe8cb508    JVM_MonitorWait + 0x29c
0xfc40e548    * java.lang.Object.wait(long) bci:0 (Interpreted frame)
0xfc40e4f4    * java.lang.Object.wait(long) bci:0 (Interpreted frame)
0xfc405a10    * java.lang.Object.wait() bci:2 line:485 (Interpreted frame)
... more lines removed here to reduce output...
----------------- t@12 -----------------
0xff2bfe3c    __lwp_park + 0x10
0xfe9925e4    AttachOperation*AttachListener::dequeue() + 0x148
0xfe99115c    void attach_listener_thread_entry(JavaThread*,Thread*) + 0x1fc
0xfec99ad8    void JavaThread::thread_main_inner() + 0x48
0xfec139f4    java_start + 0x118
0xff2bfd9c    _lwp_start
----------------- t@13 -----------------
0xff2c1500    _door_return + 0xc
----------------- t@14 -----------------
0xff2c1500    _door_return + 0xc
```

以'*'开头的帧表示是Java帧，否则是本地C/C++帧。

这个工具的输出可以作为c++filt的输入，进行C++符号解码(demangle),因为HotSpot虚拟机采用的C++语言开发，jstack工具打印的C++符号名是内部函数的符号名（即C++被编译后生成的函数名），c++filt可以将它们转换成对应的C++函数名。

### 附录 B.2.12 jstat Utility

jstat工具采用HotSpot VM的内置指令提供正在运行程序的性能和资源消耗信息，该工具一般用作性能分析，或者一些特定情况下堆内存以及垃圾回收分析。该工具不需要虚拟机特别的启动选项，缺省情况下，虚拟机内置的指令是打开的。

---

注意：这些指令在Windows 98和Windows ME上是不可用的，同时，在采用了FAT32的Windows NT, 2000, 或者XP也是不可用的。

---

jstat工具有如下选项：

- class - 打印class loader的统计或者状态信息。

- compiler - 打印HotSpot compiler的统计信息.
- gc - 打印堆内存回收的统计信息.
- gccapacity - 打印代(generations)的容量统计信息等。
- gccause - 打印垃圾回收的总结信息(同-gcutil), 已经最后和当前垃圾回收事件的原因(如果有的话) .
- gcnew - 打印新生代的统计信息.
- gcnewcapacity - 打印新生代大小，空间等统计信息.
- gcold - 打印老生代的统计信息.
- gcoldcapacity - 打印老生代大小，空间等统计信息.
- gcpermcapacity - 打印持久代的大小统计信息.
- gcutil - 打印垃圾回收的统计信息
- printcompilation - 打印HotSpot compilation method 统计信息.

jstat 提供的数据类似于Solaris或者Linux上的vmstat 和iostat
通过visualgc工具，你可以可视化地观察这些数据。

**Example of -gcutil Option** 下面是一个采用-gcutil 选项的例子，该工具绑定到进程id为2834的进程上，250毫秒采样9次。

```
$ jstat -gcutil 2834 250 9
  S0     S1     E      O      P     YGC    YGCT    FGC    FGCT     GCT
  0.00   0.00  87.14  46.56  96.82     54    1.197   140   86.559   87.757
  0.00   0.00  91.90  46.56  96.82     54    1.197   140   86.559   87.757
  0.00   0.00 100.00  46.56  96.82     54    1.197   140   86.559   87.757
  0.00  27.12   5.01  54.60  96.82     55    1.215   140   86.559   87.774
  0.00  27.12  11.22  54.60  96.82     55    1.215   140   86.559   87.774
  0.00  27.12  13.57  54.60  96.82     55    1.215   140   86.559   87.774
  0.00  27.12  18.05  54.60  96.82     55    1.215   140   86.559   87.774
  0.00  27.12  23.85  54.60  96.82     55    1.215   140   86.559   87.774
  0.00  27.12  27.32  54.60  96.82     55    1.215   140   86.559   87.774
```

这个输出显示新生代的回收发生在第三第四次采用之间，回收用了0.017秒，将对象从eden space (E) 搬到old space (O)，导致老生代空间利用从46.56% 上升到54.60%.

**Example of -gcnew Option** 下面的例子解释了-gcnew选项，jstat工具绑定到进程id为2834的进程上，250毫秒采样间隔，并显示输出，另外-h3用来控制每三行显示一次列的头部。

```
$ jstat -gcnew -h3 2834 250
S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
 192.0  192.0    0.0    0.0 15  15   96.0   1984.0   942.0    218    1.999
 192.0  192.0    0.0    0.0 15  15   96.0   1984.0  1024.8    218    1.999
 192.0  192.0    0.0    0.0 15  15   96.0   1984.0  1068.1    218    1.999
 S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
 192.0  192.0    0.0    0.0 15  15   96.0   1984.0  1109.0    218    1.999
 192.0  192.0    0.0  103.2  1  15   96.0   1984.0     0.0    219    2.019
 192.0  192.0    0.0  103.2  1  15   96.0   1984.0    71.6    219    2.019
 S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
 192.0  192.0    0.0  103.2  1  15   96.0   1984.0    73.7    219    2.019
 192.0  192.0    0.0  103.2  1  15   96.0   1984.0    78.0    219    2.019
 192.0  192.0    0.0  103.2  1  15   96.0   1984.0   116.1    219    2.019
```

从该例子中，可以看出在第四次和第五次采样之间，发生了一次新生代回收（young generation collection），持续了0.02秒，这次收集发现了监控空间0(S1U)的利用率超过了期望的监控大小（survivor size ），因此部分对象被搬到了老生代（这里没有显示），同时tenuring阈值（tenuring threshold (TT)）从15降低到1.

**Example of -gcoldcapacity Option** 下面的例子解释了-gcoldcapacity 选项，jstat绑定到进程id为21891的进程上，250毫秒中采样三次，-t选项用来控制时间戳。

```
$ jstat -gcoldcapacity -t 21891 250 3
Timestamp    OGCMN     OGCMX       OGC        OC    YGC   FGC     FGCT     GCT
    150.1   1408.0   60544.0   11696.0   11696.0    194    80    2.874   3.799
    150.4   1408.0   60544.0   13820.0   13820.0    194    81    2.938   3.863
    150.7   1408.0   60544.0   13820.0   13820.0    194    81    2.938   3.863
```

时间戳从虚拟机启动的时间开始计时，另外, -gcoldcapacity 输出显示堆正在扩展满足更多的分配，导致老生代能力（old generation capacity (OGC)）和老生代空间（space capacity (OC)）由于也在增长，第81次完全垃圾回收（Full GC (FGC)）老生代能力（old generation capacity (OGC)）从11696 KB 增长到13820 KB。代的最大能力（the generation (and space)）是60544 KB (OGCMX), 因此还有空间进行扩展。

### 附录 B.2.13 jstatd Daemon

jstatd daemon 是一个RMI（Remote Method Invocation ）服务器应用程序，用来监控内置式的(instrumented)虚拟机的启动和停止，同时为远程监控工具提供接口绑定到运行在本机的Java虚拟机上，例如，jstatd允许jps工具列出远程机器上的java进程。

注意：这些指令在Windows 98和Windows ME上是不可用的，同时，在采用了FAT32的Windows NT, 2000, 或者XP也是不可用的。

---

更详细的信息，请参考相关手册。

### 附录 B.2.14 visualgc Tool

visualgc工具是jstat的关联工具，通过visualgc，可以可视化观察垃圾回收情况，正如jstat,它采用虚拟机中的内置指令。

visualgc没有被含在JDK的随机发布包中，可以在jvmstat3.0网站上独立下载。

### 附录 B.2.15 Ctrl-Break Handler

在Solaris或者Linux操作系统上，同时按<Ctrl>+\ 键可以让JVM打印线程转储到标准输出上。在windows下等价的键是Ctrl+Break。在Solaris或者Linux上，当Java进程受到QUIT信号时，就会进行线程转储，因此kill -QUIT pid启动相同的结果。

下面详细介绍相关的Ctrl+Break:

- 线程转储
- 死锁检测
- 堆总结

**线程转储** 线程转储包括线程调用栈，线程状态，所有虚拟机中的Java线程，例如：

```
Full thread dump Java HotSpot(TM) Client VM (1.6.0-rc-b100 mixed mode):

"DestroyJavaVM" prio=10 tid=0x00030400 nid=0x2 waiting on condition [0x00000000..0xfe77fbf0]
   java.lang.Thread.State: RUNNABLE

"Thread2" prio=10 tid=0x000d7c00 nid=0xb waiting for monitor entry [0xf36ff000..0xf36ff8c0]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
        - waiting to lock <0xf819a938> (a java.lang.String)
        - locked <0xf819a970> (a java.lang.String)

"Thread1" prio=10 tid=0x000d6c00 nid=0xa waiting for monitor entry [0xf37ff000..0xf37ffbc0]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
        - waiting to lock <0xf819a970> (a java.lang.String)
        - locked <0xf819a938> (a java.lang.String)

"Low Memory Detector" daemon prio=10 tid=0x000c7800 nid=0x8 runnable [0x00000000..0x00000000]
```

```
  java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x000c5400 nid=0x7 waiting on condition [0x00000000..0x00000000]
  java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=10 tid=0x000c4400 nid=0x6 waiting on condition [0x00000000..0x00000000]
  java.lang.Thread.State: RUNNABLE

"Finalizer" daemon prio=10 tid=0x000b2800 nid=0x5 in Object.wait() [0xf3f7f000..0xf3f7f9c0]
  java.lang.Thread.State: WAITING (on object monitor)
       at java.lang.Object.wait(Native Method)
       - waiting on <0xf4000b40> (a java.lang.ref.ReferenceQueue$Lock)
       at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
       - locked <0xf4000b40> (a java.lang.ref.ReferenceQueue$Lock)
       at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
       at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=10 tid=0x000ae000 nid=0x4 in Object.wait() [0xfe57f000..0xfe57f940]
  java.lang.Thread.State: WAITING (on object monitor)
       at java.lang.Object.wait(Native Method)
       - waiting on <0xf4000a40> (a java.lang.ref.Reference$Lock)
       at java.lang.Object.wait(Object.java:485)
       at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
       - locked <0xf4000a40> (a java.lang.ref.Reference$Lock)

"VM Thread" prio=10 tid=0x000ab000 nid=0x3 runnable

"VM Periodic Task Thread" prio=10 tid=0x000c8c00 nid=0x9 waiting on condition
```

线程转储输出中包含一个头，每个线程的调用栈，线程之间用空行分开，Java线程首先被打印出来，之后是虚拟机内部线程。头包含如下信息：

- 线程名称
- 线程是否是daemon线程
- 线程优先级(prio)
- 线程id(tid), 即内存中线程结构的地址
- 本地线程id (nid)
- 线程状态, 指出在转储的那个时刻线程正在做什么
- 地址范围, 给出该线程在合法栈区的估计范围。

下面列出可能的线程状态：

| 线程状态 | 含义 |
|---|---|
| NEW | 线程尚未启动. |
| RUNNABLE | Java虚拟机正在执行该线程 |
| BLOCKED | 线程正在等待监视锁被阻塞. |
| WAITING | 这个线程正在无限期地等待另一个线程执行一个特定操作. |
| TIMED_WAITING | 这个线程正在等待指定的时间. |
| TERMINATED | 线程已退出. |

**死锁检测** 除了线程调用栈之外，转储还可以进行死锁检测，一旦发现死锁，它将打印附加信息指明该死锁，如：

```
Found one Java-level deadlock:
=============================
"Thread2":
  waiting to lock monitor 0x000af330 (object 0xf819a938, a java.lang.String),
  which is held by "Thread1"
"Thread1":
  waiting to lock monitor 0x000af398 (object 0xf819a970, a java.lang.String),
  which is held by "Thread2"

Java stack information for the threads listed above:
===================================================
"Thread2":
        at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
        - waiting to lock <0xf819a938> (a java.lang.String)
        - locked <0xf819a970> (a java.lang.String)
"Thread1":
        at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
        - waiting to lock <0xf819a970> (a java.lang.String)
        - locked <0xf819a938> (a java.lang.String)

Found 1 deadlock.
```

如果在Java虚拟机启动命令行中设置了-XX:+PrintConcurrentLocks,线程转储同时还可以打印被每一个线程所拥有的并发锁列表。

**Heap Summary** Java SE6版本，线程转储还可以打印堆总结。该输出显示不同的代内存（堆的区域），大小，使用的数量，地址范围，当在使用诸如pmap时，地址范围非常有用。

```
Heap
 def new generation   total 1152K, used 435K [0x22960000, 0x22a90000, 0x22e40000
)
```

```
  eden space 1088K,  40% used [0x22960000, 0x229ccd40, 0x22a70000)
  from space 64K,   0% used [0x22a70000, 0x22a70000, 0x22a80000)
  to   space 64K,   0% used [0x22a80000, 0x22a80000, 0x22a90000)
 tenured generation   total 13728K, used 6971K [0x22e40000, 0x23ba8000, 0x269600
00)
   the space 13728K,  50% used [0x22e40000, 0x2350ecb0, 0x2350ee00, 0x23ba8000)
 compacting perm gen  total 12288K, used 1417K [0x26960000, 0x27560000, 0x2a9600
00)
   the space 12288K,  11% used [0x26960000, 0x26ac24f8, 0x26ac2600, 0x27560000)
    ro space 8192K,  62% used [0x2a960000, 0x2ae5ba98, 0x2ae5bc00, 0x2b160000)
    rw space 12288K,  52% used [0x2b160000, 0x2b79e410, 0x2b79e600, 0x2bd60000)
```

如果设置了选项-XX:+PrintClassHistogram , 同时堆的直方图也会被显示.

### 附录 B.2.16 操作系统工具

该部分给出了故障定位的操作系统工具列表，每一个工具都给出了简要描述，更详细的信息可以参考操作系统文档。(Linux/Unix下的man手册)

**Linux Operating System** 表 4是Linux上提供的工具。

**表 4 Linux下工具列表**

| Tool | Description |
|---|---|
| c++filt | 转换成C++格式的符号表. |
| gdb | GNU debugger. |
| libnjamd | Memory allocation tracking. |
| lsstack | 打印线程堆栈(与Solaris下的pstack类似). |
| ltrace | 库调用用跟踪器(等价于Solaris下的truss ). |
| mtrace and muntrace | GNU malloc tracer. |
| proc tools(pmap,pstack) | 进程工具. |
| strace | System call tracer (equivalent to truss -t in Solaris OS). |
| top | Display most CPU-intensive processes. |
| vmstat | 报告进程，内存，IO，trap，CPU活动等信息 |

**Windows Operating System** 表 5是windows 操作系统下，提供了的工具，另外，你可以查阅MSDN库，搜索debug支持。

**Solaris Operating System** 表 6是Solaris操作系统下的工具，一些工具是Solaris 10下面才有：

表 5 windows下工具列表

| Tool | Description |
| --- | --- |
| dumpchk | 验证一个内存转储文件是否被正确创建. |
| msdev | 通过该命令可以启动VC++或者Win32调试器. |
| userdump | 用户模式进程转储工具. |
| windbg | windows下的调试器，可以调试进程，或者转储文件. |
| /Md and /Mdd 编译选项 | 自动跟踪内存分配的编译器选项. |

### 附录 B.3 内存泄漏问题定位

如果你的应用执行的时间越来越长，或者操作系统看起来越来越慢，这可能暗示系统存在内存泄漏，换句话说，虚拟内存正在被分配，但是当不再需要的时候没有归还，极端情况下，系统内存溢出，或者应用程序异常终止。该节提供针对内存泄漏的一些建议和诊断方法。

#### 附录 B.3.1 Meaning of OutOfMemoryError

内存泄漏最常见的一个提示是java.lang.OutOfMemoryError,当堆中或者堆中一个特定区域没有足够的空间分配给一个对象的时候，将抛出这个错误。这个异常表明：垃圾回收器不能够再回收出更多的内存给该对象，同时堆空间无法再扩展。当java.lang.OutOfMemoryError异常抛出时，伴随着一个线程堆栈被同步打印出来。

当本地分配无法完成时，本地库代码也可能抛出java.lang.OutOfMemoryError 异常，例如：交换分区太低。早期OutOfMemoryError的诊断主要是确定内存溢出的类新，是java堆内存溢出，还是本地堆内存溢出，下面的章节介绍了诊断细节。

- Exception in thread "main": java.lang.OutOfMemoryError: Java heap space

  详细请参考第 265页 附录 B.3.2节

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

  详细请参考第 261页 附录 B.3.1节

- Exception in thread "main": java.lang.OutOfMemoryError: Requested array size exceeds VM limit

  详细请请参考第 261页 附录 B.3.1节

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

  详细请参考第 261页 附录 B.3.1节

- Exception in thread "main": java.lang.OutOfMemoryError: <reason> <stack trace> (Native method)

  详细请参考第 262页第附录 B.3.1节

表 6 Solaris下工具列表

| Tool | Description |
|---|---|
| coreadm | 指明虚拟机产生的core文件的名字和位置. |
| cpustat | Monitor system behavior using CPU performance counters. |
| cputrack | Per-process monitor process, LWP behavior using CPU performance counters. |
| c++filt | 转换成C++格式的符号表. 该工具在Solaris上随C++编译器一起发布. |
| DTrace | Solaris新引入的工具: 动态跟踪内核函数，系统调用以及用户功能. |
| gcore | Force a core dump of a process. The process continues after the core dump is written. |
| intrstat | Report statistics on CPU consumed by interrupt threads. |
| iostat | Report I/O statistics. |
| libumem | Solaris 9 OS update 3引入的工具: 用来定位和修正内存管理Bug. |
| mdb | 内核模块级调试器 |
| netstat | Display the contents of various network-related data structures. |
| pargs | 打印进程参数,环境变量等. |
| pfiles | 打印进程打开的文件句柄信息. |
| pldd | Print shared objects loaded by a process. |
| pmap | 打印进程或者core文件的内存布局信息，包括heap, data, text段. |
| prstat | Report statistics for active Solaris OS processes. (Similar to top.). |
| prun | Set the process to running mode (reverse of pstop). |
| ps | List all processes. |
| psig | List the signal handlers of a process. |
| pstack | 打印指定进程或者core文件的线程堆栈 |
| pstop | Stop the process (suspend). |
| ptree | Print process tree containing the given pid. |
| sar | System activity reporter. |
| sdtprocess | Display most CPU-intensive processes. (Similar to top.). |
| sdtperfmeter | 显示操作系统的性能图，如CPU, disks, network等. |
| top | 显示进程的CPU相关信息. |
| trapstat | Display runtime trap statistics. (SPARC only) |
| truss | 跟踪系统调用的进入或者退出事件 |
| vmstat | Report system virtual memory statistics. |
| watchmalloc | Track memory allocations. |

**Detail Message: Java heap space** Java heap space信息表示在Java堆中无法再分配新的对象，这个错误不能断定系统一定存在内存泄漏，也许是堆内存的配置不当导致的（比如Xmx设置太小）。但对于长期运行正常的系统，如果出现该错误，很多时候意味着系统存在内存泄漏。另外，finalizers的过度使用也可能导致该问题。如果一个类有一个finalize方法，这意味着垃圾回收时，该类型的对象不能像常规对象一样被回收，而是将这些对象排队等待finalization,真正的垃圾回收发生在更晚的时候（即finalize执行完成之后）。在SUN的实现中，专门有一个daemon线程（称之为finalizer线程）负责执行这些排队对象的finalize方法，如果这个finalizer线程无法跟上finalization 队列的增长速度,那么Java堆将被填满，导致OutOfMemoryError. 还有一种场景可以导致该错误，当一个应用程序创建了高优先级线程，这样就会导致finalizer线程由于优先级太低而导致抢不到足够多的CPU，导致finalization 队列处理很慢而一直增长下去，最终导致OutOfMemoryError,定位方法请参考 265页第 附录 B.3.2节。

**Detail Message: PermGen space** PermGen的OutOfMemoryError表示永久区内存满了，永久区内存是用来存放类的地方，如果一个程序加载了过多的类，永久区的内存也许需要通过-XX:MaxPermSize设置来增加空间。

内部的java.lang.String类型的对象也存放在永久区内，java.lang.String类维护一个string池，当intern方法被调用，这个方法检查池中是否存在相同的字符串，如果存在，则内部方法直接范围永久区内的字符串，否则它增加这个字符串到池中，更精确一点讲，java.lang.String.intern方法用来获取典型的字符串，如果一个字符串作为文字（literal）出现，那么同一个类实例将被返回。当应用中存在大量的字符串，缺省大小的永久区也许会不够用，必要的时候要通过-XX:MaxPermSize来增加。

当这种类型的错误出现时，文本String.intern或者ClassLoader.defineClass出现在打印的堆栈的顶部。

jmap -permgen 命令可以打印永久区内对象的统计信息，包括内部String实例。详细请参考 247页第 附录 B.2.7节

**Detail Message: Requested array size exceeds VM limit** 该信息表示应用程序（或者该应用使用的API）企图去分配一块比堆尺寸还大的数组。比如当一个应用程序企图在堆尺寸只有256M的空间上申请512M 的数组，就会发生该类型的OutOfMemoryError异常。在大多数情况下是设置问题（堆尺寸设置太小），或者应用程序中存在申请大内存的bug,例如，数组元素的个数计算错误导致巨大的内存申请。

**Detail Message: request <size> bytes for <reason>. Out of swap space?** 信息"request <size> bytes for <reason>. Out of swap space?"表面上看是一个OutOfMemoryError错误，然而,该异常一般表示表示HotSpot VM从本地堆中申请内存失败，或者本地堆接近耗尽，该错误信息指出请求多少个字节失败，并给出原因，大多数情况下<reason>部分指出报告失败的源模块的名称，有的时候也确实告诉你原因。

当该错误发生时，VM调用致命错误处理机制，产生一个致命错误日志文件，包括线程信息，进程信息，以及宕机的时间。当本地堆耗尽时，堆内存和内存映射信息是非常有用的。详

细请参考：269页第 附录 B.4节. 该错误发生时，你也许需要借助操作系统上的相关诊断工具进行进一步定位，详细请参考 258页第 附录 B.2.16节.

该问题也许和应用程序没有任何关系，比如:

- 操作系统配置了很小的交换空间。
- 该机上的其它进程消耗了过多的内存。

如果不是上面原因导致的，也许是用于本地内存泄漏导致的该问题，比如，本地库或者应用持续的从系统申请内存，却从不释放它。

**Detail Message: <reason> <stack trace> (Native method)** 如果类型的类型是" <reason> <stack trace> (Native method)" 并且本地方法的顶层帧(frame)被打印出来，这个表示本地方法调用遇到了内存分配错误，该错误与上面的错误的区别在于，本错误表示错误发生在JNI或者本地方法中而不是发生在Java代码中。

发生该类型的错误，你也许需要使用操作系统提供的诊断工具进行进一步定位，详细请参考 258页第 附录 B.2.16节.

**Crash Instead of OutOfMemoryError** 有的时候，当本地堆内存分配失败后不久，系统就会崩溃(Crash),一般情况是由于本地代码没有检查内存分配函数返回的错误而导致的。

比如，当malloc系统调用在无可用内存时返回NULL，如果该返回值没有被检查，应用程序就会由于尝试存取非法内存位置而导致崩溃。不用的环境上，现象也许不一样，这种问题一般比较难以定位。

然而，在某些情况下，致命错误日志或者崩溃转储文件对定位这种问题是足够的。如果确认了崩溃时由于未检查内存分配失败导致的，那么必须再进一步检查为什么分配会失败。正如其它本地堆问题一样，系统也许是由于没有充分的交换分区，或者其它进程消耗了过大的内存，或者由于应用程序内存泄漏等导致的内存耗尽。

### 附录 B.3.2 Java代码中的内存泄漏诊断

诊断Java代码中的内存泄漏是一项困难任务，大多数情况下需要这个应用程序的非常具体的知识，另外，这个诊断过程往往需要漫长地往复多次。本节分为如下几个小节：

- NetBeans Profiler
- 使用jhat
- 创建堆转储(Heap Dump)
- 在正在运行的程序上获取堆直方图(Heap Histogram)。
- 在OutOfMemoryError上获取堆直方图(Heap Histogram)。
- 监控正在等待Finalization的(Pending Finalization)的对象数量
- 第三方的内存Debuggers

**使用jhat** Once the dump file is created, it can be used as input to jhat, as described in 2.5 jhat Utility. jhat工具在定位内存泄漏时非常有用，它能够浏览对象堆，查看堆中虽有可达的对象，哪个引用把持了激活对象。为了有效使用jhat，你必须要获得多个正在运行程序的堆转储，并且必须使用二进制的方式，一旦堆转储文件被创建，它也作为jhat的输入。请参考第 238页，第附录 B.2.5节

**Creating a Heap Dump** 堆转储提供了堆内存分配的相信信息，下面描述了几种堆转储的方法：

- HPROF剖析器
- jmap工具
- JConsole 工具
- -XX:+HeapDumpOnOutOfMemoryError 命令行选项

**HPROF Profiler** 正在运行的程序可以通过HPROF剖析器代理创建堆转储，例如：

```
$ java -agentlib:hprof=file=snapshot.hprof,format=b application
```

如果虚拟机是被嵌入的，或者没有采用附加的命令行选项启动的虚拟机，可以通过设置环境变量JAVA_TOOLS_OPTIONS，-agentlib将自动将这些环境变量加进去，

一旦应用程序启动时打开了HPROF,当在启动控制台上kill -3 pid或者<ctrl>+<break>,<ctrl>+(依赖于操作系统)，堆转储文件将会被打印出来。在这个例子中snapshot.hprof文件被创建。该堆转储文件包括所有原始数据和调用堆栈。

该转储文件可以包括多次转储，每一次转储都被添加到该文件中。

**jmap Utility** 通过jmap工具同样可以获得堆转储，详见 243页 附录 B.2.7，下面是一个例子：

```
$ jmap -dump:format=b,file=snapshot.jmap process-pid
```

不管虚拟机是怎样被启动的，jmap工具可以产生一个堆转储快照，在上面的例子中，snapshot.jmap的文件将产生，jmap输出文件包含所有的原始数据，但是不包含对象创建的调用堆栈这一信息。

**JConsole Utility** 另一个获得堆转储的工具是JConsole,在MBean tab页中，选择HotSpotDiagnostic MBean，点击dumpHeap。

**-XX:+HeapDumpOnOutOfMemoryError Command-line Option** 如果在JVM启动命令行中指定了-XX:+HeapDumpOnOutOfMemoryError 选项, 当OutOfMemoryError 异常发生时，虚拟机将进行堆转储。

**Obtaining a Heap Histogram on a Running Process** 你可以通过检查堆柱状图，快速缩小内存泄漏问题的可疑范围，可以通过如下几个方式获取该信息：

- 运行期运行jmap -histo pid. 该命令输出堆中每个类的对象的实例数量和总大小。如果收集了一系列的柱状图(比如每两分钟收集一次)，你可以观察到泄漏的趋势，便于进一步深入分析。
- 在Solaris或者Linux上，jmap可以从core文件中获得堆使用的柱状图。
- 如果虚拟机启动时采用了命令行选项：-XX:+PrintClassHistogram command-line option, <Ctrl> + <Break> 时也会产生一个堆的柱状图。

**Obtaining a Heap Histogram at OutOfMemoryError** 如果你在命令中指定选项：-XX:+HeapDumpOnOutOfMemoryError, 一旦OutOfMemoryError被抛出,则VM就会产生一个堆转储. 然后你可以使用jmap工具从堆转储中获取柱状图。当OutOfmemoryError时产生core文件，你可以通过在该core文件上执行jmap获取一个柱状图，例子如下：

```
$ jmap -histo \ /java/re/javase/6/latest/binaries/solaris-sparc/bin/java core.27421

Attaching to core core.27421 from executable
/java/re/javase/6/latest/binaries/solaris-sparc/bin/java, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 1.6.0-beta-b63
Iterating over heap. This may take a while...
Heap traversal took 8.902 seconds.

Object Histogram:

Size      Count   Class description
-------------------------------------------------------
86683872  3611828 java.lang.String
20979136  204     java.lang.Object[]
403728    4225    * ConstMethodKlass
306608    4225    * MethodKlass
220032    6094    * SymbolKlass
152960    294     * ConstantPoolKlass
108512    277     * ConstantPoolCacheKlass
104928    294     * InstanceKlassKlass
68024     362     byte[]
65600     559     char[]
31592     359     java.lang.Class
27176     462     java.lang.Object[]
25384     423     short[]
```

```
17192      307      int[]
:
```

该例子显示OutOfMemoryError是由java.lang.String (共3611828 )对象数量导致的，但是没有提供是哪里创建的这些对象的进一步信息。不管如何，这些信息仍然是非常有用的，进一步的分析可以借助HPROF或者jhat等工具，确定strings对象被分配的位置，以及是什么对象引用了这些对象而导致没有释放。

**Monitoring the Number of Objects Pending Finalization** 正如 82页第 §3.5.2节提到的，finalizers的过度使用也可以导致OutOfMemoryError, 可以通过如下几个方式监控这些等待finalization的对象数量。

- The JConsole management tool (见 234页第 附录 B.2.3节). 该工具在Summary tab页里面报告了正等待finalization的对象数量，这个数量是近似的，但是它对于分析应用的特点和确定是否该应用依赖很多finalization非常有用。
- 在Solaris和Linux上，jmap -finalizerinfo 可以打印正在等待finialization的对象信息。
- 可以通过java.lang.management.MemoryMXBean 类上getObjectPendingFinalizationCount方法获取正在等待finalization的对象。

**Third Party Memory Debuggers** 除了前面提到的工具，同样市面上有大量的第三方内存分析工具，如JProbe,OptimizeIt等。

**Diagnosing Leaks in Native Code** 有几种技术可以定位本地内存泄漏，一般来讲，没有针对所有平台的独立的理想解决方案。

**Tracking All Memory Allocation and Free Calls** 一个最常用的方法是跟踪所有本地内存分配和释放操作，这种方法非常简单，但是非常有效，这些年来，许多产品已经开发了本地堆内存分配和使用的跟踪工具。比如Purify,Sun的DBX运行期检查工具，详见第 附录 B.3.2节第 267页。这些工具可以发现本地代码的内存泄漏以及访问未经分配的本地内存。所有这些类型的工具同样可以用于使用了本地代码的java应用，一般情况下，这些工具是平台相关的，比如由于虚拟机可以在运行期动态创建代码（如JIT技术)，因此这些工具也会造成错误解释，请确认工具版本和你使用的虚拟机版本是匹配的。

许多简单的本地内存检测例子在http://sourceforge.net/有介绍，这些工具的库假设你已经修改了你的源代码，并且在内存分配函数上放入了wrapper函数，并进行了重新编译。更强大的工具是不侵入这些动态内存分配函数，比如Solaris 9 Update 3引入的libumem.so( 见 269页第 附录 B.3.2节)。

**Tracking Memory Allocation in a JNI Library** 如果你来写一个JNI库，采用简单的wrapper方式预先创建一些本地化的方法确保无内存泄漏是非常明智的，下面例程是一个简单的内存跟踪方式，在源代码中定义如下的行：

```
#include <stdlib.h>
#define malloc(n) debug_malloc(n, __FILE__, __LINE__)
#define free(p) debug_free(p, __FILE__, __LINE__)

Then you can use the following functions to watch for leaks.

/* Total bytes allocated */
static int total_allocated;
/* Memory alignment is important */
typedef union { double d; struct {size_t n; char *file; int line;} s; } Site;
void *
debug_malloc(size_t n, char *file, int line)
{
    char *rp;
    rp = (char*)malloc(sizeof(Site)+n);
    total_allocated += n;
    ((Site*)rp)->s.n = n;
    ((Site*)rp)->s.file = file;
    ((Site*)rp)->s.line = line;
    return (void*)(rp + sizeof(Site));
}
void
debug_free(void *p, char *file, int line)
{
    char *rp;
    rp = ((char*)p) - sizeof(Site);
    total_allocated -= ((Site*)rp)->s.n;
    free(rp);
}
```

JNI库需要周期性（或者在shutdown时）检查total_allocated变量的值，确保问题能及时发现。上面的代码也可以扩展一下，将导致内存分配的代码位置保存到分配链表中，用作报告泄漏的内存是哪里分配的。你需要确保debug_free()和debug_malloc成对使用，同样对realloc(),calloc(),strdup()也可以采取同样的处理方式。这种方式仅适用于局部的内存泄漏分析。

更加全局的方式是对整个进程的库调用进行干预。

**Tracking Memory Allocation With OS Support** 大多数操作系统提供了全局内存分配跟踪支持。

- 在Windows下，站点http://msdn.microsoft.com/library/default.asp 可以搜索到debug支持。Microsoft C++ 编译器提供了/Md 和/Mdd 编译选项，直接包含特别的内存分配跟踪支持。
- Linux 有一些诸如mtrace,libnjamd 等工具进行内存分配跟踪。

- Solaris 提供了watchmalloc tool. Solaris 9 OS update 3 提供了libumem tool (详见第 269页附录 B.3.2).

**Using dbx to Find Leaks** Sun的debugger dbx包含了运行期检测能力，可以检测内存泄漏，同时dbx也提供了linux版本

```
$ dbx ${java_home}/bin/java
Reading java
Reading ld.so.1
Reading libthread.so.1
Reading libdl.so.1
Reading libc.so.1
(dbx) dbxenv rtc_inherit on
(dbx) check -leaks
leaks checking - ON
(dbx) run HelloWorld
Running: java HelloWorld
(process id 15426)
Reading rtcapihook.so
Reading rtcaudit.so
Reading libmapmalloc.so.1
Reading libgen.so.1
Reading libm.so.2
Reading rtcboot.so
Reading librtc.so
RTC: Enabling Error Checking...
RTC: Running program...
dbx: process 15426 about to exec("/net/bonsai.sfbay/j2se/build/solaris-i586/bin/java")
dbx: program "/net/bonsai.sfbay/export/j2se/build/solaris-i586/bin/java"
just exec'ed
dbx: to go back to the original program use "debug $oprog"
RTC: Enabling Error Checking...
RTC: Running program...
t@1 (l@1) stopped in main at 0x0805136d
0x0805136d: main       :        pushl    %ebp
(dbx) when dlopen libjvm { suppress all in libjvm.so; }
(2) when dlopen libjvm { suppress all in libjvm.so; }
(dbx) when dlopen libjava { suppress all in libjava.so; }
(3) when dlopen libjava { suppress all in libjava.so; }
(dbx) cont
Reading libjvm.so
Reading libsocket.so.1
Reading libsched.so.1
```

```
Reading libCrun.so.1
Reading libm.so.1
Reading libnsl.so.1
Reading libmd5.so.1
Reading libmp.so.2
Reading libhpi.so
Reading libverify.so
Reading libjava.so
Reading libzip.so
Reading en_US.ISO8859-1.so.3
hello world
hello world
Checking for memory leaks...

Actual leaks report    (actual leaks:          27  total size:      46851 bytes)

  Total      Num of  Leaked     Allocation call stack
  Size       Blocks  Block
                     Address
==========  ====== =========== =======================================
     44376       4      -      calloc < zcalloc
      1072       1  0x8151c70  _nss_XbyY_buf_alloc < get_pwbuf < _getpwuid <
                               GetJavaProperties < Java_java_lang_System_initProperties <
                               0xa740a89a< 0xa7402a14< 0xa74001fc
       814       1  0x8072518  MemAlloc < CreateExecutionEnvironment < main
       280      10      -      operator new < Thread::Thread
       102       1  0x8072498  _strdup < CreateExecutionEnvironment < main
        56       1  0x81697f0  calloc < Java_java_util_zip_Inflater_init < 0xa740a89a<
                               0xa7402a6a< 0xa7402aeb< 0xa7402a14< 0xa7402a14< 0xa7402a14
        41       1  0x8072bd8  main
        30       1  0x8072c58  SetJavaCommandLineProp < main
        16       1  0x806f180  _setlocale < GetJavaProperties <
                               Java_java_lang_System_initProperties < 0xa740a89a< 0xa7402a14<
                               0xa74001fc< JavaCalls::call_helper < os::os_exception_wrapper
        12       1  0x806f2e8  operator new < instanceKlass::add_dependent_nmethod <
                               nmethod::new_nmethod < ciEnv::register_method <
                               Compile::Compile #Nvariant 1 < C2Compiler::compile_method <
                               CompileBroker::invoke_compiler_on_method <
                               CompileBroker::compiler_thread_loop
        12       1  0x806ee60  CheckJvmType < CreateExecutionEnvironment < main
        12       1  0x806ede8  MemAlloc < CreateExecutionEnvironment < main
        12       1  0x806edc0  main
         8       1  0x8071cb8  _strdup < ReadKnownVMs < CreateExecutionEnvironment < main
```

```
        8       1  0x8071cf8  _strdup < ReadKnownVMs < CreateExecutionEnvironment < main
```

上面的输出显示，当进程将要退出时，部分内存仍然没有被释放，dbx报告了这一嫌疑内存泄漏，之所以说是嫌疑，是因为初始化期间分配的内存也许在整个应用程序生命期间是一直需要的，这种情况下，dbx尽管报告了是内存泄漏，但真实情况下却不是内存泄漏。注意，该例子使用了两个抑制命令，抑制了虚拟机libjvm.so和支持库libjava.so报告的泄漏。

**Using libumem to Find Leaks** Solaris 9 update 3, ibumem.so 库和modular debugger (mdb) 可以用来调试内存泄漏. 在使用libumem之前, 你必须预先加载libumem 并按如下方式设置环境变量:

```
$ LD_PRELOAD=libumem.so
$ export LD_PRELOAD

$ UMEM_DEBUG=default
$ export UMEM_DEBUG
```

现在，运行java应用程序，并在退出之前先将虚拟机停下来，下面的例子采用truss命令使虚拟机在_exit系统调用时停止进程。

```
$ truss -f -T _exit java MainClass arguments
```

在这时，可以将mdb attach到虚拟机上：

```
$ mdb -p pid
>::findleaks
```

::findleaks 是mdb用来发现内存泄漏的命令. 如果发现了泄漏, findleaks 命令将打印内存分配调到的地址，缓冲区地址，以及最近的符号。

通过转储bufctl结构，还可以得到导致该内存泄漏的调用堆栈，该结构的地址可以通过::findleaks命令的输出获得，更详细的信息请参考：http://access1.sun.com/techarticles/libumem.html.

### 附录 B.4 Troubleshooting System Crashes

本章描述系统崩溃时的定位方法。系统崩溃，有几种可能的原因，如，虚拟机的bug,系统库的bug,Java SE库或者API的bug,应用程序本地代码bug,或者操作系统bug,外部因素也可以导致程序崩溃，比如操作系统资源耗尽。

一般来说虚拟机或者java SE库的bug较为少见。该章提供了如何测试崩溃的几种建议，有些情况下，通过某些手段要绕过崩溃直到bug被排除为止。

任何崩溃的第一步是查找fatal error日志，这个文件是虚拟机在崩溃时产生的，Fatal error请参考:第 277页appendix:FatalErrorLog.

#### 附录 B.4.1 Sample Crashes

本节采用了几个例子来讲述如何分析error log以及相关建议。

**Determining Where the Crash Occurred** Error日志的文件头指明了导致系统崩溃的问题帧, 见第 279appendix:headerformat

如果最顶端的帧是本地帧，而且不是操作系统的本地帧，这表明问题可能在本地库中，而不在虚拟机中。解决该问题的第一步是调查发生崩溃处的本地库的源代码。

1. 如果应用程序提供的本地库，则直接分析你的本地库，选项-Xcheck:jni可以帮你发现许多本地bug.
2. 如果本地库是由其它开发商提供的，这说明是这个第三方的库导致。
3. 通过观察jre/lib或者jre/bin目录检查这个库是否是JRE提供的，如果是，则将相关错误报告发给相应的实现商。

**Crash in Native Code** 如果致命错误log致命该崩溃发生在本地库中，也许是本地代码或者JNI代码的bug, 当然也可能是其它因素导致的。通过分析这个库或者core文件，或者崩溃转储是一个好的开始，例如下面的日志：

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
#  SIGSEGV (0xb) at pc=0x417789d7, pid=21139, tid=1024
#
# Java VM: Java HotSpot(TM) Server VM (6-beta2-b63 mixed mode)
# Problematic frame:
# C  [libApplication.so+0x9d7]

In this case a SIGSEGV occurred with a thread executing in the library libApplication.so.

In some cases a bug in a native library manifests itself as a crash in Java VM code.
Consider the following crash where a JavaThread fails while in the _thread_in_vm state
(meaning that it is executing in Java VM code) :

# An unexpected error has been detected by HotSpot Virtual Machine:
#
#  EXCEPTION_ACCESS_VIOLATION (0xc0000005) at pc=0x08083d77, pid=3700, tid=2896
#
# Java VM: Java HotSpot(TM) Client VM (1.5-internal mixed mode)
# Problematic frame:
# V  [jvm.dll+0x83d77]

---------------  T H R E A D  ---------------

Current thread (0x00036960):  JavaThread "main" [_thread_in_vm, id=2896]
 :
Stack: [0x00040000,0x00080000),  sp=0x0007f9f8,  free space=254k
```

```
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [jvm.dll+0x83d77]
C  [App.dll+0x1047]          <========= C/native frame
j  Test.foo()V+0
j  Test.main([Ljava/lang/String;)V+0
v  ~StubRoutines::call_stub
V  [jvm.dll+0x80f13]
V  [jvm.dll+0xd3842]
V  [jvm.dll+0x80de4]
V  [jvm.dll+0x87cd2]
C  [java.exe+0x14c0]
C  [java.exe+0x64cd]
C  [kernel32.dll+0x214c7]
```

在这种情况下，调用堆栈表明，App.dll的本地例程已经调到VM里面了。如果在本地应用程序库中崩溃了（比如上面的例子），你也许可以attatch本地程序调试器到core文件或者崩溃转储文件上，比如: dbx, gdb, windbg等。另一个方式是通过在命令行中增加选项-Xcheck:jni,这个选项不能确保找到所有与JNI相关的问题，但它确实能够识别非常多的问题。如果导致崩溃的本地库属于Java runtime environment(如awt.dll,net.dll等等)，表明也许你遇到了一个库或者API bug，如果经过进一步分析得出这个库确实是一个JVM库bug，那么你就可以提交一个错误单。

**Crash due to Stack Overflow** Java语言中的栈溢出正常情况下会导致java.lang.StackOverflowError，另外C/C++会写过栈的最后，触发一个栈溢出，这是一个致命错误，会导致进程终止。

在HotSpot实现中，java方法和C/C++本地代码共享栈帧(stack frame),即用户本地代码和虚拟机本地代码。Java方法产生代码检查到栈的尾部固定长度的空间确保栈空间是可用的，这样本地代码被调用时就不会超过栈空间。到栈的尾部的长度被称作为影子页(shadow pages),影子页的大小大约在3到20页（依赖于操作系统类型），这个长度是可调的。如果带有本地代码的应用程序需要更大的缺省长度，可以适度的调大影子空间的大小，通过-XX:StackShadowPages=n来调整影子空间的大小，n要大约这个平台下的缺省值。

如果一个应用程序发生segmentation fault,但是同时没有产生core文件或者错误日志文件，或者windows上没有产生STACK_OVERFLOW_ERROR或“An irrecoverable stack overflow has occurred,”错误消息，这表明StackShadowPages被超过了，意味着需要更多的空间。如果你增大了StackShadowPages，也许你同时需要通过-Xss增大缺省线程的栈大小(thread stack),增加缺省的线程栈大小也许会导致系统能创建的线程数量下降，因此需要小心选择这个值，不同的操作系统上，线程栈的大小从256k到1024k不等。

下面是windows系统上的一个段异常的致命错误日志，一个线程导致了本地代码栈溢出

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
#  EXCEPTION_STACK_OVERFLOW (0xc00000fd) at pc=0x10001011, pid=296, tid=2940
#
```

```
# Java VM: Java HotSpot(TM) Client VM (1.6-internal mixed mode, sharing)
# Problematic frame:
# C  [App.dll+0x1011]
#

---------------  T H R E A D  ---------------

Current thread (0x000367c0):  JavaThread "main" [_thread_in_native, id=2940]
:
Stack: [0x00040000,0x00080000),  sp=0x00041000,  free space=4k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
C  [App.dll+0x1011]
C  [App.dll+0x1020]
C  [App.dll+0x1020]
:
C  [App.dll+0x1020]
C  [App.dll+0x1020]
...<more frames>...

Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
j  Test.foo()V+0
j  Test.main([Ljava/lang/String;)V+0
v  ~StubRoutines::call_stub
```

注意上面的输出的如下信息：

- 异常是EXCEPTION_STACK_OVERFLOW.
- 线程状态是_thread_in_native, 意味着线程正在执行本地或者JNI代码。
- 在栈信息中，自由空间仅4K (Windows上的一个页). 另外，stack pointer (sp) is at 0x00041000, 它非常接近于栈的尾部(0x00040000).
- 打印的本地帧显示，存在一个递归本地函数。
- ...<more frames>... 表示存在额外的帧没有被打印. 输出仅限定为100 帧.

**Crash in the HotSpot Compiler Thread** 如果致命错误输出显示当前线程是CompilerThread0, CompilerThread1, adapterCompiler,这表明你可能遇到的是编译bug,这时候你可以尝试临时切换编译器（比如，从HotSpot Server切换到HotSpot Client，或者相反），或者对导致该crash的编译方法进行排除，方法见下。

**Crash in Compiled Code** 如果crash发生在被编译的代码中，说明这个错误可能是一个编译器bug,导致了产生了错误的代码。这种情况下，问题帧有J符号作为标识（即Java编译代码帧，这里指JIT技术）,例如：

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
#  SIGSEGV (0xb) at pc=0x0000002a99eb0c10, pid=6106, tid=278546
#
# Java VM: Java HotSpot(TM) 64-Bit Server VM (1.6.0-beta-b51 mixed mode)
# Problematic frame:
# J  org.foobar.Scanner.body()V
#
:
Stack: [0x0000002aea560000,0x0000002aea660000),  sp=0x0000002aea65ddf0,
  free space=1015k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
J  org.foobar.Scanner.body()V

[error occurred during error reporting, step 120, id 0xb]
```

注意完全的堆栈是不可用的. 输出行"error occurred during error reporting"意味着当获取堆栈时问题又上升了（即由遇到了其它问题，在这个例子中，也许是栈已经被破坏了）。通过切换编译器（如从HotSpot Client VM切换到HotSpot Server VM，或者反之），或者将引起崩溃的方法从编译中去掉，也许能够临时规避这个问题。

**Crash in VMThread** 如果致命错误日志输出显示当前线程是VMThread，然后在THREAD段中查找包含VM_Operation的行，VMThread是虚拟机中特殊的线程，它执行特别的任务，比如垃圾回收。如果VM_Operation 指示当前的操作是垃圾回收操作，说明你可能遇到了一个诸如堆内存破坏的问题。

崩溃也许是一个GC问题，也可能是其它什么的（比如编译器或者运行期bug）如一些对象引用状态不一致或者不正确。通过更改GC参数，也许能临时规避错误。

### 附录 B.4.2 Finding a Workaround

如果一个关键应用发生了崩溃，而且这个崩溃是由于虚拟机中的bug导致的，我们期望快速的找到一个临时的规避方法，下面主要介绍常见的规避方法。

---

注意：如果下面介绍的规避方法消除了系统崩溃，这种规避方法只能作为临时之用，后续需要继续定位以找到问题的根因.

---

**Crash in HotSpot Compiler Thread or Compiled Code** 如果致命的错误日志显示，崩溃发生在一个编译线程，那么有可能（但并非总是如此）您所遇到的是一个编译错误。同样，如果崩溃放生在编译的代码中，则有可能是编译器产生了不正确的代码。如果虚拟机是HotSpot Client VM (-client option),在日志中的编译器线程显示为CompilerThread0。如果虚

拟机是HotSpot Server VM ,在日志中的编译器线程会有多个，显示为CompilerThread0, CompilerThread1, and AdapterThread.

以下错误日志片段的是一个编译错误（J2SE 5.0已经修正）。该日志文件显示，使用的虚拟机HotSpot Server VM，崩溃发生在线程CompilerThread1 。此外，日志文件表明，目前CompileTask是编译java.lang.Thread.setPriority方法。

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
:
# Java VM: Java HotSpot(TM) Server VM (1.5-internal-debug mixed mode)
:
--------------- T H R E A D ---------------

Current thread (0x001e9350): JavaThread "CompilerThread1" daemon [_thread_in_vm, id=20]

Stack: [0xb2500000,0xb2580000), sp=0xb257e500, free space=505k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [libjvm.so+0xc3b13c]
:

Current CompileTask:
opto: 11      java.lang.Thread.setPriority(I)V (53 bytes)

--------------- P R O C E S S ---------------

Java Threads: ( => current thread )
  0x00229930 JavaThread "Low Memory Detector" daemon [_thread_blocked, id=21]
=>0x001e9350 JavaThread "CompilerThread1" daemon [_thread_in_vm, id=20]
```

In this case there are two potential workarounds:

- 蛮力方法: 使用-client参数启动虚拟机，让虚拟机以HotSpot Client VM方式运行。
- 假设该bug只出现在setPriority编译中，则排除该方法的编译.

第一种方法（采用-client命令行选项），在某些环境下也许很容易，但在另外一些环境上，如果配置很复杂，或者本身命令行不允许修改配置，实施起来相对困难。一般情况下，从HotSpot Server VM 切换到the HotSpot Client VM，会导致峰值处理能力下降。

第二种方法需要在工作目录下创建.hotspot_compiler，例如：

```
exclude    java/lang/Thread    setPriority
```

这个文件的格式为：exclude CLASS METHOD , CLASS是全名的类名（包含包名），METHOD是方法的名称，构造函数的函数名为<init> ，静态初始化名称为<clinit>。

注意：.hotspot_compiler 是一个不支持的接口. 仅仅是为了故障定位，而将该它单独放在这里。

一旦程序重启，编译器将不再编译.hotspot_compiler文件中指定的方法。为了验证这个文件是否生效，可以在运行期的日志文件中查找相关字样，如：

```
### Excluding compile:    java.lang.Thread::setPriority
```

注意名称分隔符使用的是'.',而不是'/'.

**Crash During Garbage Collection** 如果崩溃发生在垃圾回收期间，致命错误日志报告了VM_Operation正在进行，为了讨论方便，假设大多并发GC(-XX:+UseConcMarkSweep)没有启用。日志中的THREAD段中显示了VM_Operation，指出了下面几种情形之一：

- Generation collection for allocation（新生代垃圾回收）
- Full generation collection（完全垃圾回收）
- Parallel gc failed allocation（并行GC分配失败）
- Parallel gc failed permanent allocation（并行GC永久区分配失败）
- Parallel gc system gc(并行系统垃圾回收)

最可能的是当前线程是VMThread,这个线程专门用来执行虚拟机中的特别任务。下面的日志显示崩溃发生在串行垃圾回收中：

```
---------------  T H R E A D  ---------------

Current thread (0x002cb720):  VMThread [id=3252]

siginfo: ExceptionCode=0xc0000005, reading address 0x00000000

Registers:
EAX=0x0000000a, EBX=0x00000001, ECX=0x00289530, EDX=0x00000000
ESP=0x02aefc2c, EBP=0x02aefc44, ESI=0x00289530, EDI=0x00289530
EIP=0x0806d17a, EFLAGS=0x00010246

Top of Stack: (sp=0x02aefc2c)
0x02aefc2c:   00289530 081641e8 00000001 0806e4b8
0x02aefc3c:   00000001 00000000 02aefc9c 0806e4c5
0x02aefc4c:   081641e8 081641c8 00000001 00289530
0x02aefc5c:   00000000 00000000 00000001 00000001
0x02aefc6c:   00000000 00000000 00000000 08072a9e
```

```
0x02aefc7c:   00000000 00000000 00000000 00035378
0x02aefc8c:   00035378 00280d88 00280d88 147fee00
0x02aefc9c:   02aefce8 0806e0f5 00000001 00289530
Instructions: (pc=0x0806d17a)
0x0806d16a:   15 08 83 3d c0 be 15 08 05 53 56 57 8b f1 75 0f
0x0806d17a:   0f be 05 00 00 00 00 83 c0 05 a3 c0 be 15 08 8b

Stack: [0x02ab0000,0x02af0000),  sp=0x02aefc2c,  free space=255k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [jvm.dll+0x6d17a]
V  [jvm.dll+0x6e4c5]
V  [jvm.dll+0x6e0f5]
V  [jvm.dll+0x71771]
V  [jvm.dll+0xfd1d3]
V  [jvm.dll+0x6cd99]
V  [jvm.dll+0x504bf]
V  [jvm.dll+0x6cf4b]
V  [jvm.dll+0x1175d5]
V  [jvm.dll+0x1170a0]
V  [jvm.dll+0x11728f]
V  [jvm.dll+0x116fd5]
C  [MSVCRT.dll+0x27fb8]
C  [kernel32.dll+0x1d33b]

VM_Operation (0x0373f71c): generation collection for allocation, mode:
 safepoint, requested by thread 0x02db7108
```

---

注意：垃圾回收过程中发生崩溃并不意味着垃圾回收实现一定存在Bug,也许它是一个编译器或者运行期bug,或者其它什么问题。

---

当你重复遇到垃圾回收过程导致的崩溃是，你可以尝试如下的规避方法：

- 修改GC参数，例如：如果你正在使用串行垃圾回收，可以修改成并行垃圾回收，或者反之。
- 如果你正在使用HotSpot Server VM, 尝试修改成HotSpot Client VM.

如果你不确定你使用的哪一种垃圾回收模式，如果有core文件，你可以使用jmap(Solaris或者Linux)工具从core文件中获取堆信息。一般情况下，GC参数如果不配置的话，windows上缺省使用的是串行垃圾回收，solaris和Linux上依赖于机器的配置，如果机器的内存2G以上，超过两个处理器，那么将使用并行GC,否则将使用串行GC. 命令行选项-XX:+UseSerialGC 表示

采用串行GC,-XX:+UseParallelGC.表示采用并行GC,如果在将并行GC修改成串行GC,那么在多处理器上，可能会导致性能下降。

**Class Data Sharing** 类数据共享是J2SE 5.0引入的一个新特性，当使用Sun提供的installer在32位系统上安装JRE时，installer从JAR中以私有的内部格式装载一系列的类，并将它们存储到一个称作共享档案的文件中。当虚拟机启动时，共享档案被内存映射，这样可以多虚拟机共享这些类，从而节省类的加载，并且共享加载类的元数据。在J2SE 5.0上，只有HotSpot client VM才使用了该特性。另外，共享只有在串行垃圾回收才支持。

致命错误日志在日志的头部打印了版本号，如果类共享被打开了，有专门的字符串进行了标注，如：

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
#  EXCEPTION_ACCESS_VIOLATION (0xc0000005) at pc=0x08083d77, pid=3572, tid=784
#
# Java VM: Java HotSpot(TM) Client VM (1.5-internal mixed mode, sharing)
 CompilerThread0# Problematic frame:
# V  [jvm.dll+0x83d77]
```

在命令行中通过选项-Xshare:off可以关闭共享. 如果共享类特性被禁掉之后，问题不复存在，那么说明这个bug可能是类共享特性的bug,此时你需要提交bug报告单给JVM的实现者。

**Microsoft Visual C++ Version Considerations** 32位的JDK 6是在windows Server 2003 SP1上采用的是Microsoft Studio .NET2003(专业版)编译的,四月份在windows Server 2005上编译了64位的JDK. 如果你经历了Java SE版本的崩溃，而且你采用了不同的编译器版本编译了你自己的本地库(比如有些代码是在一个运行期环境上编译的，而另一些代码则是在另一类型的运行期环境上编译的)，你必须要怀疑是兼容性导致了这个崩溃。比如你使用一个运行期库分配内存，你必须使用同一类型运行期库对它进行释放。如果分配和释放采用了不同的运行期库，故障行为是不确定的。

### 附录 B.5 Fatal Error Log

当致命错误发生时，虚拟机会创建相应的错误日志，该文件格式在不同的发布里可能稍有不同。该日志中如含如下内容：

- 发生致命错误的位置。
- 致命错误的描述。
- 文件头。
- 线程信息。
- 进程信息。
- 系统信息。

### 附录 B.5.1 Location of Fatal Error Log

-XX:ErrorFile=file用来指定错误日志文件创建的目录，这里file采用全路径名，其中子串%%表示'%',%p表示进程id

在下面的例子中，错误日志将被写到/var/log/java and will be named java_errorpid.log中。

```
java -XX:ErrorFile=/var/log/java/java_error%p.log
```

如果-XX:ErrorFile=file没有被指定，缺省文件名为hs_err_pidpid.log，日志文件将被创建在工作目录下。

### 附录 B.5.2 Description of Fatal Error Log

当致命错误发生时，错误日志可能包含如下信息：

- 操作异常或信号，引起了致命错误
- 版本和配置信息
- 引起该致命错误的线程细节，以及线程调用栈
- 正在运行的线程列表以及它们的状态。
- 堆使用的总结信息
- 加载的本地库列表
- 命令行参数
- 环境变量
- 操作系统或者CPU细节

---

注意：在某些情况下，仅有一部分信息被打印出来，这是由于这个致命错误是如此严重，以至于错误处理句柄不能够报告或者恢复所有的细节。

---

错误日志是文本文件格式，包括了下面的段：

- 引起系统崩溃的简要描述. 见第 279 附录 B.5.3.
- 线程信息. 见第 281 附录 B.5.4
- 进程信息. 见第 283 附录 B.5.5
- 系统信息. 见第 288 附录 B.5.5

---

注意：这里描述的致命错误日志的格式是Java SE 6的，在不同的Java版本上可以稍有不同。

---

### 附录 B.5.3 Header Format

致命日志文件的头包含了问题的一个简要描述，该文件头同时被输出到标准输出和日志文件中。例如：

```
#
# An unexpected error has been detected by Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x417789d7, pid=21139, tid=1024
#
# Java VM: Java HotSpot(TM) Client VM (1.6.0-rc-b63 mixed mode, sharing)
# Problematic frame:
# C  [libNativeSEGV.so+0x9d7]

#
# If you would like to submit a bug report, please visit:
#   http://java.sun.com/webapps/bugreport/crash.jsp
#
```

下面的例子显示虚拟机崩溃在一个不期望的信号量上，下面一行描述了信号量类型，程序计数器(PC),进行ID,线程ID

```
#  SIGSEGV (0xb) at pc=0x417789d7, pid=21139, tid=1024
      |      |           |             |         +--- thread id
      |      |           |             +------------- process id
      |      |           +--------------------------- program counter
      |      |                                        (instruction pointer)
      |      +--------------------------------------- signal number
      +---------------------------------------------- signal name
```

在下一行包含了虚拟机的版本，以及运行模式，如混合模式还是解释模式，类共享是否打开了等等。

```
# Java VM: Java HotSpot(TM) Client VM (1.6.0-rc-b63 mixed mode, sharing)
```

接着显示导致崩溃的函数帧，如：

```
# Problematic frame:
# C  [libNativeSEGV.so+0x9d7]
  |              +-- Same as pc, but represented as library name and offset.
  |                  For position-independent libraries (JVM and most shared
  |                  libraries), it is possible to inspect the instructions
  |                  that caused the crash without a debugger or core file
  |                  by using a disassembler to dump instructions near the
  |                  offset.
  +----------------- Frame type
```

在这个例子中,"C"帧表示这是一个本地帧，帧的类型如下：

**表 7 Thread Types**

| Frame Type | Description |
|---|---|
| C | Native C frame |
| j | Interpreted Java frame |
| V | VM frame |
| v | VM generated stub frame |
| J | Other frame types, including compiled Java frames |

内部错误将导致虚拟机错误处理句柄产生类似的错误转储，只是头格式是不同的，内部错误的例子一般包含了guarantee()失败，assertion failure，ShouldNotReachHere()等，下面是一个例子：

```
#
# An unexpected error has been detected by HotSpot Virtual Machine:
#
# Internal Error (4F533F4C494E55583F491418160E43505000F5), pid=10226, tid=16384
#
# Java VM: Java HotSpot(TM) Client VM (1.6.0-rc-b63 mixed mode)
```

在上面的文件头中，没有signal name 或者signal number. 而是包含了“Internal Error”和一个十六进制文本串. 这个十六进制文本串，实际上是错误发生的地方所在的文件和行号的编码，一般情况下，这个串信息只对虚拟机的开发者有用。

---

注意- 针对同样的错误串的临时解决方法在一种情况下工作，在另一种情况下也许不能工作。即：

- 有同样根因的错误也许会产生不同的错误串。
- 有同样错误串的两个错误也许根因完全不同。

因此，在定位问题时，错误串不能当作唯一的判断依据。

### 附录 B.5.4 Thread Section Format

这个段包含导致崩溃的线程的信息，如果多个线程在同时崩溃，仅仅一个线程被打印。

**Thread Information** 线程段的第一部分线程导致崩溃的线程信息，如：

```
Current thread (0x0805ac88):  JavaThread "main" [_thread_in_native, id=21139]
                    |             |         |            |          +-- ID
                    |             |         |            +------------- state
                    |             |         +-------------------------- name
                    |             +------------------------------------ type
                    +-------------------------------------------------- pointer
```

线程指针是一个指向虚拟机内部线程结构体的指针，如果你不是在调试一个活动的虚拟机或者core文件，这个一般情况下是没用的。下面是可能的线程类型：

- JavaThread
- VMThread
- CompilerThread
- GCTaskThread
- WatcherThread
- ConcurrentMarkSweepThread

下面是线程的状态：

**表 8 Thread States**

| Thread | State Description |
|---|---|
| _thread_uninitialized | 线程未创建，只有当内存被破坏，才会出现这个情况。 |
| _thread_new | 线程已经创建，但是尚未启动. |
| _thread_in_native | 线程正在执行本地代码. 这也许意味着本地代码有bug |
| _thread_in_vm | 线程正在执行虚拟机代码. |
| _thread_in_Java | 线程正在执行Java解释代码，或者被编译的Java代码. |
| _thread_blocked | 线程被阻塞. |
| ..._trans | 如果上面任一个状态包含_trans, 这意味着线程正在进行状态切换. |

线程ID是本地线程标识符。

如果线程是一个daemon线程，则“daemon”字样会打印出来。

**Signal Information** 错误日志中的下一段信息是导致虚拟机异常终止的不期望的信号。

```
siginfo: ExceptionCode=0xc0000005, reading address 0xd8ffecf1
```

在上面的例子，异常代码为0xc0000005 (ACCESS_VIOLATION)，当线程尝试读地址0xd8ffecf1时，异常发生了。

Solaris OS 和Linux 分别使用信号si_signo 和si_code表示异常, 如：

```
siginfo:si_signo=11, si_errno=0, si_code=1, si_addr=0x00004321
```

**Register Context** 错误日志的再下一段是寄存器上下文。这部分信息的格式是处理器相关的，下面是一个Intel (IA32)的输出：

```
Registers:
EAX=0x00004321, EBX=0x41779dc0, ECX=0x080b8d28, EDX=0x00000000
ESP=0xbfffc1e0, EBP=0xbfffc1f8, ESI=0x4a6b9278, EDI=0x0805ac88
EIP=0x417789d7, CR2=0x00004321, EFLAGS=0x00010216
```

当结合指令是，寄存器的值是非常有用的，正像下面所描述的：

**Machine Instructions** 在寄存器值的后面，错误日志包含了栈的顶部，同时包含了崩溃时程序计数器（PC）附近的32个字节的指令。这些程序指令可以被反编译器进行反编译，获得crash附近位置的指令，注意IA32 和AMD64在指令的长度上是不同的，它并不总是能够可靠解码的。

```
Top of Stack: (sp=0xbfffc1e0)
0xbfffc1e0:   00000000 00000000 0818d068 00000000
0xbfffc1f0:   00000044 4a6b9278 bfffd208 41778a10
0xbfffc200:   00004321 00000000 00000cd8 0818d328
0xbfffc210:   00000000 00000000 00000004 00000003
0xbfffc220:   00000000 4000c78c 00000004 00000000
0xbfffc230:   00000000 00000000 00180003 00000000
0xbfffc240:   42010322 417786ec 00000000 00000000
0xbfffc250:   4177864c 40045250 400131e8 00000000
Instructions: (pc=0x417789d7)
0x417789c7:   ec 14 e8 72 ff ff ff 81 c3 f2 13 00 00 8b 45 08
0x417789d7:   0f b6 00 88 45 fb 8d 83 6f ee ff ff 89 04 24 e8
```

**Thread Stack** 在可能的时候，错误日志会输出线程调用栈，包括栈的基地址和顶部地址，当前栈的指针，该线程未用的栈变量的大小，可能的话，还包括线程调用栈，最大100个帧被打印。对于C/C++的帧而言，库的名称也许会被打印。特别需要注意的是，在一些致命错误的情况下，栈也许被破坏了，这种情况下，详细信息往往不可用。

```
Stack: [0x00040000,0x00080000),  sp=0x0007f9f8,  free space=254k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [jvm.dll+0x83d77]
C  [App.dll+0x1047]
j  Test.foo()V+0
j  Test.main([Ljava/lang/String;)V+0
v  ~StubRoutines::call_stub
V  [jvm.dll+0x80f13]
V  [jvm.dll+0xd3842]
V  [jvm.dll+0x80de4]
C  [java.exe+0x14c0]
C  [java.exe+0x64cd]
C  [kernel32.dll+0x214c7]

Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
j  Test.foo()V+0
j  Test.main([Ljava/lang/String;)V+0
v  ~StubRoutines::call_stub
```

这个线程包含两个线程调用栈

- 本地调用栈, 及本地线程的所有函数调用，但是不包括Java函数。本地调用栈提供了重要线索，通过自上而下分析库，你可以初步得知到底是那个库导致了这个问题。
- 第二个线程栈是Java线程调用栈, 跳过了本地栈. 依赖于崩溃的类型，Java栈也许能打印，也许不打印。.

**Further Details** 如果错误发生在虚拟机线程，或者编译器线程，更详细的信息会被打印。例如，如果是虚拟机线程，同时会打印在致命错误时虚拟机线程正在执行的操作。下面的例子中，是编译线程导致了致命错误：

```
Current CompileTask:
HotSpot Client Compiler:754   b
nsk.jvmti.scenarios.hotswap.HS101.hs101t004Thread.ackermann(IJ)J (42 bytes)
```

对于HotSpot Server VM,输出也许稍微不同（因为方法可能被编译成了本地代码），但仍然包含了全路径类名和方法。

### 附录 B.5.5 Process Section Format

线程部分之后是进程段，它包括整个进程的信息，线程列表以及进程的内存使用情况。

**Thread List** 线程列表包括虚拟机能感知到的线程，包括所有的虚拟机线程，已经内部线程，不包括还没有attach到进程的用户创建的本地线程，输出格式如下：

```
=>0x0805ac88 JavaThread "main" [_thread_in_native, id=21139]
|      |          |        |             |             +----- ID
|      |          |        |             +------------------- state
|      |          |        |                                  (JavaThread only)
|      |          |        +--------------------------------- name
|      |          +------------------------------------------ type
|      +-------------------------------------------------- pointer
+------------------------------------------------------ "=>" current thread
```

例子如下：

```
Java Threads: ( => current thread )
  0x080c8da0 JavaThread "Low Memory Detector" daemon [_thread_blocked, id=21147]
  0x080c7988 JavaThread "CompilerThread0" daemon [_thread_blocked, id=21146]
  0x080c6a48 JavaThread "Signal Dispatcher" daemon [_thread_blocked, id=21145]
  0x080bb5f8 JavaThread "Finalizer" daemon [_thread_blocked, id=21144]
  0x080ba940 JavaThread "Reference Handler" daemon [_thread_blocked, id=21143]
=>0x0805ac88 JavaThread "main" [_thread_in_native, id=21139]
```

其它线程:

```
  0x080b6070 VMThread [id=21142]
  0x080ca088 WatcherThread [id=21148]
```

**VM State** 再下一部分信息是虚拟机状态，指明整个虚拟机的状态，状态分为如下几种类型：

**表 9 VM States**

| 一般的VM状态 | 描述 |
|---|---|
| not at a safepoint | 正常执行. |
| at safepoint | 所有的线程被阻塞在VM中，等待一个特定的VM操作完成. |
| synchronizing | 需要一个特定的VM操作，该VM正在等待所有阻塞的线程 |

The VM state output is a single line in the error log, as follows.

```
VM state:not at safepoint (normal execution)
```

**Mutexes and Monitors** 错误日志中的再下一段是线程当前拥有的互斥量和监视器，互斥量是虚拟机内部的锁，而不是和Java对象相关的监视器。下面的例子显示了当崩溃发生时锁相关的信息。对每一个锁，信息包含了锁名称，拥有者，内部互斥量的地址，以及OS的锁。一般情况下，只有特别熟悉虚拟机的人才有用。

```
VM Mutex/Monitor currently owned by a thread:
([mutex/lock_event])[0x007357b0/0x0000031c] Threads_lock - owner thread: 0x00996318
[0x00735978/0x000002e0] Heap_lock - owner thread: 0x00736218
```

**Heap Summary** 错误日志中的再下一段是堆内存总结，这部分信息的输出依赖于GC的配置参数，在这个例子中，使用了串行垃圾回收器，类共享被禁止。

```
Heap
 def new generation   total 576K, used 161K [0x46570000, 0x46610000, 0x46a50000)
  eden space 512K,  31% used [0x46570000, 0x46598768, 0x465f0000)
  from space 64K,   0% used [0x465f0000, 0x465f0000, 0x46600000)
  to   space 64K,   0% used [0x46600000, 0x46600000, 0x46610000)
 tenured generation   total 1408K, used 0K [0x46a50000, 0x46bb0000, 0x4a570000)
   the space 1408K,   0% used [0x46a50000, 0x46a50000, 0x46a50200, 0x46bb0000)
 compacting perm gen  total 8192K, used 1319K [0x4a570000, 0x4ad70000, 0x4e570000)
   the space 8192K,  16% used [0x4a570000, 0x4a6b9d48, 0x4a6b9e00, 0x4ad70000)
No shared spaces configured.
```

**Memory Map** 日志中的下一段信息是崩溃时虚拟内存区域列表。当应用程序较大时，这段列表可能非常长，内存映射在定位某些类型的崩溃非常有用，因为它可以告诉内实际上使用了那些库，它们在内存中的位置，还包括堆，栈，和保护页的位置。

内存映射的格式与操作系统相关，在Solaris上，每个库的基地址和结束地址被打印出来。在Linux系统下面，进程内存映射被打印（/proc/pid/maps）. 在Windows系统上，每一个库的基地址和结束地址被打印，下面的例子是Linux/X86上的输出，注意为了简洁起见，大多数行被截取了。

```
Dynamic libraries:
08048000-08056000 r-xp 00000000 03:05 259171     /h/jdk6/bin/java
08056000-08058000 rw-p 0000d000 03:05 259171     /h/jdk6/bin/java
08058000-0818e000 rwxp 00000000 00:00 0
40000000-40013000 r-xp 00000000 03:0a 400046     /lib/ld-2.2.5.so
40013000-40014000 rw-p 00013000 03:0a 400046     /lib/ld-2.2.5.so
40014000-40015000 r--p 00000000 00:00 0
Lines omitted.
4123d000-4125a000 rwxp 00001000 00:00 0
4125a000-4125f000 rwxp 00000000 00:00 0
```

```
4125f000-4127b000 rwxp 00023000 00:00 0
4127b000-4127e000 ---p 00003000 00:00 0
4127e000-412fb000 rwxp 00006000 00:00 0
412fb000-412fe000 ---p 00083000 00:00 0
412fe000-4137b000 rwxp 00086000 00:00 0
Lines omitted.
44600000-46570000 rwxp 00090000 00:00 0
46570000-46610000 rwxp 00000000 00:00 0
46610000-46a50000 rwxp 020a0000 00:00 0
46a50000-46bb0000 rwxp 00000000 00:00 0
46bb0000-4a570000 rwxp 02640000 00:00 0
Lines omitted.
```

格式说明如下：

```
40049000-4035c000 r-xp 00000000 03:05 824473  /jdk1.5/jre/lib/i386/client/libjvm.so
|<------------->|  ^      ^       ^     ^    |<----------------------------------->|
  Memory region    |      |       |     |                            |
                   |      |       |     |                            |
  Permission   --- +      |       |     |                            |
    r: read               |       |     |                            |
    w: write              |       |     |                            |
    x: execute            |       |     |                            |
    p: private            |       |     |                            |
    s: share              |       |     |                            |
                          |       |     |                            |
  File offset   ----------+       |     |                            |
                                  |     |                            |
  Major ID and minor ID of -------+     |                            |
  the device where the file             |                            |
  is located (i.e. /dev/hda5)           |                            |
                                        |                            |
  inode number  ------------------------+                            |
                                                                     |
  File name  --------------------------------------------------------+
```

在内存映射输出上，每一个库有两个内存区间:一个是数据段区间，一个是代码段区间。代码段的权限使用r-xp(readable,executable,private)进行标识，数据段的权限由rw-p(readable,writable,private)段进行标识。Java堆已经被包含在输出的前面总结部分，验证被保留的实际内存区域是否与堆总结部分的值匹配，这部分信息非常有用。这部分的属性被设置rwxp.

线程栈常常作为两块back-to-back区域在内存映射中进行显示。一块权限为—p(保护页，guard page), 另一块权限为rwxp(实际栈空间)。通过这块信息可以知道保护页的尺寸或者栈的尺寸，在这个例子中，栈位于4127b000 - 412fb000区间。

在windows系统上，内存映射输出是每一个模块被加载的地址和结束地址，例如：

```
Dynamic libraries:
0x00400000 - 0x0040c000     c:\jdk6\bin\java.exe
0x77f50000 - 0x77ff7000     C:\WINDOWS\System32\ntdll.dll
0x77e60000 - 0x77f46000     C:\WINDOWS\system32\kernel32.dll
0x77dd0000 - 0x77e5d000     C:\WINDOWS\system32\ADVAPI32.dll
0x78000000 - 0x78087000     C:\WINDOWS\system32\RPCRT4.dll
0x77c10000 - 0x77c63000     C:\WINDOWS\system32\MSVCRT.dll
0x08000000 - 0x08183000     c:\jdk6\jre\bin\client\jvm.dll
0x77d40000 - 0x77dcc000     C:\WINDOWS\system32\USER32.dll
0x7e090000 - 0x7e0d1000     C:\WINDOWS\system32\GDI32.dll
0x76b40000 - 0x76b6c000     C:\WINDOWS\System32\WINMM.dll
0x6d2f0000 - 0x6d2f8000     c:\jdk6\jre\bin\hpi.dll
0x76bf0000 - 0x76bfb000     C:\WINDOWS\System32\PSAPI.DLL
0x6d680000 - 0x6d68c000     c:\jdk6\jre\bin\verify.dll
0x6d370000 - 0x6d38d000     c:\jdk6\jre\bin\java.dll
0x6d6a0000 - 0x6d6af000     c:\jdk6\jre\bin\zip.dll
0x10000000 - 0x10032000     C:\bugs\crash2\App.dll
```

**VM Arguments and Environment Variables** 错误日志的下一段信息是虚拟机参数，之后是环境变量列表，如：

```
VM Arguments:
java_command: NativeSEGV 2

Environment Variables:
JAVA_HOME=/h/jdk
PATH=/h/jdk/bin:.:/h/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:
     /usr/dist/local/exe:/usr/dist/exe:/bin:/usr/sbin:/usr/ccs/bin:
     /usr/ucb:/usr/bsd:/usr/etc:/etc:/usr/dt/bin:/usr/openwin/bin:
     /usr/sbin:/sbin:/h:/net/prt-web/prt/bin
USERNAME=user
LD_LIBRARY_PATH=/h/jdk6/jre/lib/i386/client:/h/jdk6/jre/lib/i386:
     /h/jdk6/jre/../lib/i386:/h/bugs/NativeSEGV
SHELL=/bin/tcsh
DISPLAY=:0.0
HOSTTYPE=i386-linux
OSTYPE=linux
ARCH=Linux
```

```
MACHTYPE=i386
```

注意，环境变量列表并不是全集，而仅仅是应到到该虚拟机的环境变量子集。

**Signal Handlers** 在Solaris和Linux上，错误日志中的下一段信息则是信号处理句柄(signal handler)列表.

```
Signal Handlers:
SIGSEGV: [libjvm.so+0x3aea90], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGBUS: [libjvm.so+0x3aea90], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGFPE: [libjvm.so+0x304e70], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGPIPE: [libjvm.so+0x304e70], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGILL: [libjvm.so+0x304e70], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGUSR1: SIG_DFL, sa_mask[0]=0x00000000, sa_flags=0x00000000
SIGUSR2: [libjvm.so+0x306e80], sa_mask[0]=0x80000000, sa_flags=0x10000004
SIGHUP: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGINT: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGQUIT: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGTERM: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGUSR2: [libjvm.so+0x306e80], sa_mask[0]=0x80000000, sa_flags=0x10000004
```

**System Section Format** 错误日志中的最后一部分信息是系统信息，包括操作系统版本，CPU信息，内存配置总结信息，这些数据都和操作系统相关。下面是Solaris 9上的一段输出：

```
--------------- S Y S T E M ---------------

OS:                       Solaris 9 12/05 s9s_u5wos_08b SPARC
           Copyright 2005 Sun Microsystems, Inc.  All Rights Reserved.
                        Use is subject to license terms.
                           Assembled 21 November 2005

uname:SunOS 5.9 Generic_112233-10 sun4u  (T2 libthread)
rlimit: STACK 8192k, CORE infinity, NOFILE 65536, AS infinity
load average:0.41 0.14 0.09

CPU:total 2 has_v8, has_v9, has_vis1, has_vis2, is_ultra3

Memory: 8k page, physical 2097152k(1394472k free)

vm_info: Java HotSpot(TM) Client VM (1.5-internal) for solaris-sparc,
built on Aug 12 2005 10:22:32 by unknown with unknown Workshop:0x550
```

在Solaris或者Linux上，操作系统信息包含在/etc/*release文件中，这个文件描述了系统的类型，已经补丁号等。但在Linux上，这个文件也许反映不了任何操作系统升级，因为它允许用户任意编译某一部分。

在Solaris上，uname可以打印内核的名字，已经线程库(T1或者T2) 在Linux上，uname 也可以打印内核的名字，已经libc的版本，线程库的类型，如：

```
uname:Linux 2.4.18-3smp #1 SMP Thu Apr 18 07:27:31 EDT 2002 i686
libc:glibc 2.2.5 stable linuxthreads (floating stack)
    |<- glibc version ->|<--  pthread type       -->|
```

在Linux上，可能有三种线程类型，即linuxthreads (fixed stack), linuxthreads (floating stack), and NPTL.一般情况下被安装在/lib, /lib/i686, and /lib/tls 目录下.

知道线程的类型非常重要，比如如果崩溃发生在pthread中，你也许可以选择不同的pthread库来规避这个问题，可以通过LD_LIBRARY_PATH 或者LD_ASSUME_KERNEL来选择不同的库。

Solaris 和Linux上, 下面的信息是rlimit信息. 注意虚拟机的缺省栈的大小要小于系统的limit,例如：

```
rlimit: STACK 8192k, CORE 0k, NPROC 4092, NOFILE 1024, AS infinity
             |          |         |           |           virtual memory (-v)
             |          |         |           +--- max open files (ulimit -n)
             |          |         +----------- max user processes (ulimit -u)
             |          +------------------------- core dump size (ulimit -c)
             +---------------------------------------- stack size (ulimit -s)
load average:0.04 0.05 0.02
```

在下面的信息是CPU架构，以及能力：

```
CPU:total 2 family 6, cmov, cx8, fxsr, mmx, sse
  |    |              <----- CPU features ---->
  |    |
  |    |
  |    +------------ Total number of CPUs
  |
  +--- processor family (IA32 only):
       3 - i386
       4 - i486
       5 - Pentium
       6 - PentiumPro, PII, PIII
       15 - Pentium 4
```

下面的表列出了SPARC系统上可能的CPU特性：

表 10 SPARC Features

| SPARC Feature | Description |
|---|---|
| has_v8 | Supports v8 instructions. |
| has_v9 | Supports v9 instructions. |
| has_vis1 | Supports visualization instructions. |
| has_vis2 | Supports visualization instructions. |
| is_ultra3 | UltraSparc III. |
| no-muldiv | No hardware integer multiply and divide. |
| no-fsmuld | No multiply-add and multiply-subtract instructions. |

下面的表列出了Intel/IA32系统上可能的CPU特性：

表 11 Intel/IA32 Features

| | |
|---|---|
| cmov | Supports cmov instruction. |
| cx8 | Supports cmpxchg8b instruction. |
| fxsr | Supports fxsave and fxrstor. |
| mmx | Supports MMX. |
| sse | Supports SSE extensions. |
| sse2 | Supports SSE2 extensions. |
| ht | Supports Hyper-Threading Technology. |

下面的表列出了AMD64/EM64T系统上可能的CPU特性：

表 12 AMD64/EM64T Features

| AMD64/EM64T Feature | Description |
|---|---|
| amd64 | AMD Opteron, Athlon64, and so forth. |
| em64t | Intel EM64T processor. |
| 3dnow | Supports 3DNow extension. |
| ht | Supports Hyper-Threading Technology. |

再下面的信息是内存信息：

```
                                                            unused swap space
                                   total amount of swap space      |
                           unused physical memory         |         |
 total amount of physical memory      |                   |         |
     page size            |           |                   |         |
```

```
              v              v        v            v        v
Memory: 4k page, physical 513604k(11228k free), swap 530104k(497504k free)
```

有些系统需要交换空间至少是物理内存的两倍，而另外一些系统则没有这方面的要求。一般来讲，如果物理内存和交换空间几乎是满的情况下，内存问题导致的崩溃非常值得怀疑。

在Linux上，内核也许将未用的物理内存转变成文件缓冲区，当系统需要更多的内存时，内核将这些缓冲区内存归还应用程序，这些处理是透明的，但这并不意味着致命错误报告的未用物理内存一定接近0

错误日志的SYSTEM部分最后的信息是vm_info，即内嵌于libjvm.so/jvm.dll的版本字符串，每一个虚拟机有它唯一的vm_info字符串。

# 附录 C　在Solaris下，查找占用指定的端口的进程

在solaris结合pfiles可以查出指定的端口被哪个进程占用。脚本如下：

```
  \#!/bin/sh
  if[ \$\# -gt 1]; then
    echo "Usage:port number"
    exit 1
  fi \\
  if[ \$\# -gt 0]; then
    port = 0
  else \\
    port = \$1
  fi
ps -e >./.process
procss=‘awk ’{print \$1}’ ‘ ./.process
for p in \$process; do
  if[\$p != "PID"]; then
     res=‘pfiles \$p 2>\&1 | grep "port:\$port" ’
     if["\$res"] \&\& [-n "\$res"]; then
        echo "\$p:\$res"
     fi
  fi
done
```

类似地，在Linux下可以使用lsof -i:port 找出使用该端口的进程。如lsof -i:8080.

# 附录 D 如何在solaris下面分析IO瓶颈?

Solaris10提供了Dtrace的分析工具，可以通过如下的dtrace脚本来分析（将下面的内容放在一个脚本中，如io.d）：

```
#!/usr/sbin/dtrace -s
dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}
io:::start
{
        @files[pid, execname, args[2]->fi_pathname] = sum(args[0]->b_bcount);
}
dtrace:::END
{
        normalize(@files, 1024);
        printf("\n");
        printf("%6s %-12s %6s %s\n", "PID", "CMD", "KB", "FILE");
        printa("%6d %-12.12s %@6d %s\n", @files);
}
```

然后运行io.d(记得之前先chmod +x io.d增加可执行权限)，输出结果类似如下：

```
   PID CMD              KB FILE
     3 fsflush           0 <none>
   493 db2fmcd           0 <none>
   493 db2fmcd           1 /var/db2/.fmcd.lock
```

根据这个输出结果可以知道哪个进行消耗了太多的IO.

然后再通过truss -p <消耗IO较多的进程号> 这个命令可以打印出所有的IO细节(调用的哪个IO函数，正在处理哪个文件等)，可以查出到底是哪些IO操作很频繁。

这个dtrace命令在solaris 10下可用。

# 附录 E AIX操作系统下，32位进程的最大内存占有情况

32位应用程序最大的寻址空间为4G,但由于AIX虚拟内存模型，应用程序并不能真正地存取整个4G地址空间。AIX将地址空间分为16段，每段256M,进程的寻址空间是以段为单位进行管理，这样每一个段可以为进程私有，或者进程间共享。segment map如下：

**segment 0** 内核级

**segment 1** 应用程序数据

**segment 2** 线程堆栈和私有数据

**segment 3-C** 共享内存（对所有进程）

**segment D,F** 共享库text区以及相应的data区

**segment E** 共享内存以及其它内核使用

基本上，对于JVM而言，只有从段3到段C使可用的(大约为2.5G)。这块内存用来分配给本地内存和java堆内存。Java堆内存使可以通过-Xmx进行设置，你可以分配2.25G作为Java堆内存，而只有0.25G的作为本地内存，这种设置会导致本地内存不足，因此如何设置堆内存的大小，往往需要进行权衡和评估。svmon工具可以帮助做这个评估。

# 附录 F 关于TCP/IP

做应用层网络程序开发的，手头都有一把利器：抓包工具，即协议分析工具。常用的且功能强大的抓包工具有tcpdump(windows下面叫windump)、ethereal等。工作中常常会遇到因应用层程序在协议字段发送和接收解析上不一致出现"纠纷"的情况，这时我们一般采用TCP协议层用协议分析工具抓取该层原始数据包作为证据。还有的就是在客户端或者服务器端连接的问题上的一些现象也需要到TCP协议层出发，然后逐步溯源到真正的问题的根因。在抓包文件中有几个最重要的标记:

**SYN** (Synchronize sequence numbers),用来建立连接，在连接请求中，SYN=1,ACK=0,连接响应时,SYN=1 ACK=1. SYN和ACK区分Connection Request和Connection Accepted

**ACK** (Acknowlegement field significant) 值1表示确认号(Acknowledgment number)为合法，为0的时候表示数据段不包含确认信息，确认号被忽略。

**PSH** (PUSH function)push标记的数据如果被置1,表示接收端应尽快将数据提交给应用层,而不必等到缓冲区满才传送。

**RST** (Reset the connection) 用来复位因某种原因引起而出现的错误连接，也用来拒绝非常数据和请求，如果连接收到RST位的时候，通常表示发生了某种错误。

**FIN** (No more data from sender)用来释放连接，表示发送方已经没有数据发送了。

**FIN，ACK** 释放连接确认。

# 附录 G windows 2003/XP下，一个端口可以多个监听

在介绍之前，我们先运行下面一个例子：

```
import java.net.InetSocketAddress; import
java.nio.channels.ServerSocketChannel;

public class Main {
    public static void main(String[] args) {
        ServerSocketChannel serverChannel = null;
        try{
            serverChannel = ServerSocketChannel.open();
            serverChannel.socket().setReuseAddress(true);
        }catch(Exception e){
            e.printStackTrace();
        }
        try{
            serverChannel.socket().bind(new InetSocketAddress(8080));
        }catch(Exception e){
            e.printStackTrace();
        }

        System.out.println("bin success");
        try{
            Thread.currentThread().join();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

分别在两个控制台下面执行两次该程序，你会发现在window2003/XP下面没有任何异常产生，两次执行都可以成功，也就是说可以在同一个端口上建立两个监听。windows XP下产生的结果如下：

```
C:\Documents and Settings\Admin>netstat -a

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:990            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:2869           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1044         127.0.0.1:9100         ESTABLISHED
  TCP    127.0.0.1:1121         127.0.0.1:1122         ESTABLISHED
```

```
TCP    127.0.0.1:1122         127.0.0.1:1121         ESTABLISHED
```

从netstat的输出中可以看出，在8080上建立了两个监听会话。但同样的代码在linux，第二次执行抛出如下的异常：

```
java.net.BindException: Address already in use
 at gnu.java.net.PlainSocketImpl.bind(libgcj.so.7rh)
 at java.net.ServerSocket.bind(libgcj.so.7rh)
 at java.net.ServerSocket.bind(libgcj.so.7rh)
 at Main.main(Main.java:17)
```

在windows NT4.0中，如果使用bind()函数，对同一个listening port做多次bind()调用，后面的bind()调用会失败，而且会遇到"error 10048"的错误信息。为了避免出现这样的错误，新版的windows会在关闭前面的会话后在侦听端口将setsockopt()和SO_REUSADDR一起使用。这不是系统的bug,而是设计的原因。

在windows上，如果一个侦听套接字没有关闭，则在同一个端口上调用bind()函数时，除非使用SO_REUSADDR，否则会收到"error 10048"消息，如果使用SO_REUSADDR来同时将多个服务器绑定到同一个端口，则只有一个随机侦听套接字接受连接请求。

# 附录 H Suse9.0下，线程创建的数量和堆内存/永久内存的关系

在不同的操作系统下，JVM的运行参数之间是互相影响的，比如Xmx设置过大，可能系统能创建的线程数量就下降。在Suse9.0下面测试数据如下表：

表 13 Linux下线程创建的数量和堆内存/永久内存的关系

| -XX:Permsize | -XX:MaxPermsize | -Xms | -Xmx | -Xss | 可以创建的最大线程数 |
|---|---|---|---|---|---|
| 64M | 512M | 1024M | 1024M | - | 760 |
| 256M | 512M | 1024M | 1024M | - | 759 |
| 512M | 512M | 1024M | 1024M | - | 759 |
| 64M | 64M | 1024M | 1024M | - | 1650 |
| - | - | - | - | - | 1611 |
| - | - | - | - | 256K | 3299 |
| - | - | - | - | 512K | 1661 |
| - | - | - | - | 1024K | 837 |
| 256M | 256M | 1024M | 1024M | - | 1264 |
| 512M | 512M | 1024M | 1024M | 1024K | 632 |
| 512M | 512M | 1024M | 1024M | - | 755 |
| - | - | 256M | 1024M | - | 1647 |
| - | - | 1024M | 1024M | - | 1644 |

从如上测试可以得出一下结论：

1. Linux下，Xss的默认值为512K.
2. 虚拟机能创建的线程数量只与最大设置选项有关，如与-Xmx有关，与-Xms无关，与-XX:MaxPermsize有关，与-XX:Permsize无关。

# 附录 I JConsole

J2SE 5.0或者以上版本提供了一个综合的监控和管理工具JConsole,通过它可以获取应用运行的性能和资源消耗等信息。JConsole提供了可视化的界面，可以分析Java线程状态或者GC日志. JConsole能提供如下好用的功能:

- Java内存使用情况监测。
- 动态打开或关闭GC选项以及类加载的详细跟踪选项.
- 监测线程等（如用来发现死锁等）
- 存取SUN平台下的OS资源

JConsole本质上是JMX兼容的GUI工具，它可以连上启动了管理代理(management agent)的Java虚拟机。启动带有管理代理的虚拟机，请参考com.sun.management.jmxremote中描述的系统属性。如启动J2SE Java2Demo例子代码，命令行如下：

```
JDK_HOME/bin/java -Dcom.sun.management.jmxremote -jar Java2Demo.jar
```

启动JConsole,命令行如下(详细请参考[18])：

```
JDK_HOME/bin/jconsole
```

JDK1.4或者以下的版本不支持该工具，要分析相关问题，只能通过本书前面介绍的方法进行分析，手工打印线程堆栈，手工分析GC输出等（当然，手工分析和自动分析背后的原理是相同的）。

# 附录 J gcviewer

gcviewer是一个用来观察垃圾回收信息的可视化工具,支持IBM或SUN JDK的日志信息输出格式。前面已经介绍过，有些性能问题能够从GC信息中观察出来，如：系统是否存在内存泄漏？系统的内存参数(Xmx)是否设置合理，垃圾回收参数是否设置恰当等。由于这个工具提供可视化的显示，因此在分析这类问题，比较直观。实际上，这个工具只是将GC数据显示可视化，本质上并没有带来额外的分析能力，在掌握了前面介绍的方法之后，和手工分析没有本质上的区别。但由于它的直观性，因此不失为一个好助手。

可以从http://www.tagtraum.com/下载gcviewer工具，这个工具实际上就是一个jar文件，下载后，可以直接通过如下命令启动：

```
java -jar gcviewer.jar
```

运行该命令后，就可以在窗口中打开要分析的gc文件[63]。

# 附录 K IBM JDK下定位引起CoreDump的JIT方法

如果你遇到了JIT代码导致的Core dump,你也许能看到类似下面的错误输出：

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=4148bf20
Signal_Code=00000001 Handler1=00000100002ADB14 Handler2=00000100002F480C
R0=0000000000000006 R1=0000000000000006 R2=0000000000000000 R3=0000000000000006
R4=0000000000000001 R5=0000000080056808 R6=0000010002BCCA20 R7=0000000000000000
......
Compiled_method=java/lang/String.trim()Ljava/lang/String;
Target=2_30_20071004_14218_bHdSMR (AIX 5.3)
CPU=ppc (4 logical CPUs)(0xf5000000 RAM)
... ...
```

在这段输出中，最重要的两个信息是：

[63]通过在Java命令行中增加-Xloggc:mygc.txt，虚拟机在运行期间，会自动将GC信息中收集到mygc.txt中。

- vmState=0x00000000 指出引起错误的代码不是JVM运行期代码（JVM runtime code）.
- Compiled_method= 指出产生JIT代码的Java方法，这个方法导致了JVM core dump.

有了这两个信息，下一步的方向就很清楚了。首先禁掉JIT,检查系统是否仍然存在问题，如果问题消失，那么就是这段代码引起的问题。如果问题仍然存在，那么就是JDK存在BUG, 通过升级小版本号进行继续观察，如果仍然存在问题，那么大胆地给JDK开发商提BUG单。

# 附录 L 如何解读Java Core 文件？

## 附录 L.1 SUN JDK

## 附录 L.2 IBM JDK

```
1STSEGTYPE Object Memory
NULL       segment          start            alloc            end               type     bytes
1STSEGMENT 00000001153D5A30 0700000000000000 0700000060000000 0700000060000000  00000009 60000000
1STSEGMENT 00000001153D5850 0700000060000000 07000000F0000000 07000000F0000000  0000000A 90000000
1STSEGMENT 00000001153D5940 07000000F0000000 0700000100000000 0700000100000000  0000000A 90000000
```

60000000+90000000+90000000(16进制)为实际分配的内存

另外，专门有一个工具可以查看core 文件jca22：http:/www.ibm.com/developerworks/forums/forum.jspa?forumID=858

# 附录 M 几个奇怪的现象

下面几个是真实的堆栈，很是奇怪

## 附录 M.1 等锁的线程也可以处于runnable状态？

```
"UDPMessageProcessorThread" daemon prio=5 tid=0x02e62f20 nid=0x5c9 runnable [51581000...515819c0]
at java.net.PlainDatagramSocketImpl.receive(Native Method)
-waiting to lock<0xbc4a4678>(a java.net.PlainDatagramSocketImpl)
at java.net.DatagramSocket.receive(DatagramSocket.java:711)
-locked<0xbbe0f7c8>(a java.net.DatagramPacket)
-locked<0xbbe0f7c8>(a java.net.DatagramSocket)
at gov.nist.javax.sip.statck.UDPMessageProcessor.run(UDPMessageProcessor.java:257)
at java.lang.Thread.run(Thread.java:534)
```

## 附录 M.2 没锁的也可以waiting for？

```
"Timer-13" prio=1 tid=0x08725b80 nid=0x2430 waiting for monitor entry [0xa6a80000...0xa6a80480]
at org.jboss.cache.Fqn.parse(Fqn.java:219)
at org.jboss.cache.Fqn.fromString(Fqn.java:215)
at com.huawei.csp.adapter.cache.jboss.NativeCacheServiceImpl.hasData(NativeCacheServiceImpl.java:294)
```

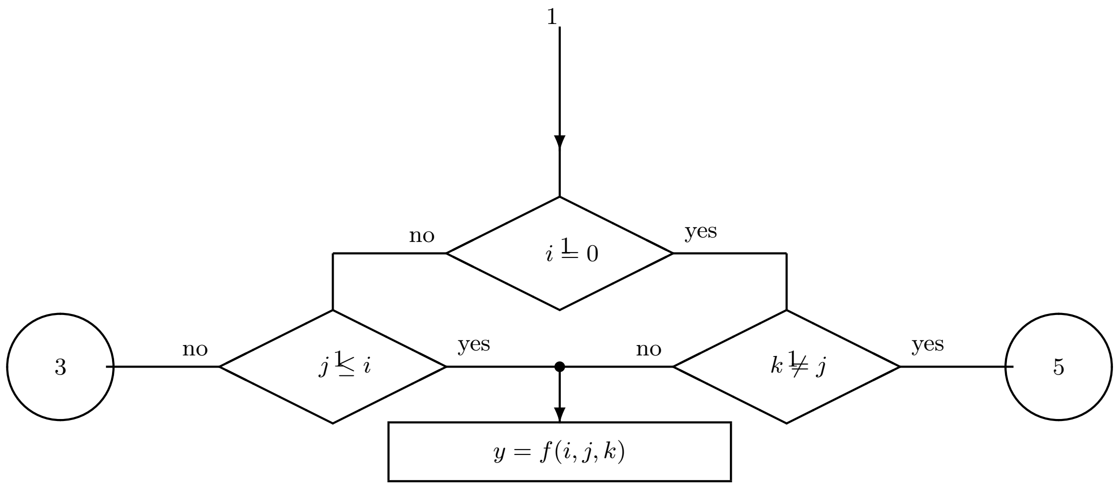
```
at com.bsf.iface.CommonServiceExtendImpl.getSystemParamFromModule(CommonServiceExtendImpl.java:357)
... ...
at java.util.TimerThread.mainLoop(Timer.java:512)
at java.util.TimerThread.run(Timer.java:462)

"http-8091-Processor8" daemon prio=1 tid=0xa3d2e740 nid=0x24ba waiting for monitor entry
   [0xa9987e000...0x99880600]
at sun.reflect.GeneratedMethodAccessor886.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:585)
at org.hibernate.property.BasicPropertyAccessor$BasicGetter.get(BasicPropertyAccessor.java:145)
... ...
at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPool.java:685)
at java.lang.Thread.run(Thread.java:595)
```

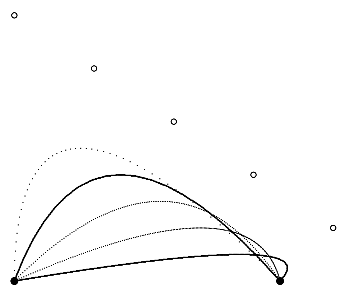
s

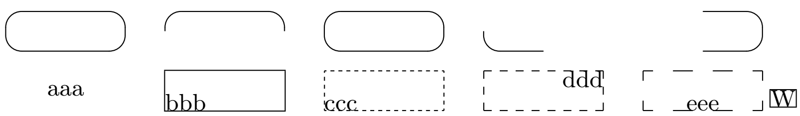
# 附录 N　感谢TEX

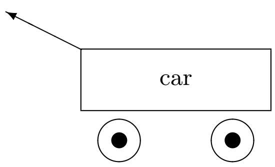
| | |
|---|---|
| `\thispagestyle{empty}` | 不打印页眉页脚 |
| `\begin{figure}[H]` | [H]表示图的位置不动 |
| `\subsubsection*{}` | 可以让该级别不出现在目录中 |

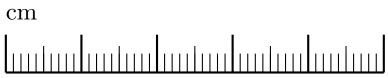
有有有有有有






| | |
|---|---|
| -Xnoclassgc | 禁止class垃圾回收 |
| -Xincgc | 增量垃圾回收 |
| -Xloggc:<file> | 垃圾回收日志 |

| | |
|---|---|
| -Xms<size> | 最小堆内存 |
| -Xmx<size> | 最大堆内存 |
| -Xss<size> | 线程堆栈大小 |
| -XX:PermSize | Perm内存的大小，即存放class的空间的大小 |

| | |
|---|---|
| -XX:-AllowUserSignalHandlers | 允许用户安装自己的信号处理句柄. (仅Solaris，Linux下适用.) |
| -XX:-UseSerialGC | 适用串行垃圾回收. (5.0版本引入) |
| -XX:-UseSpinning | Enable naive spinning on Java monitor before entering operating system thread synchronizaton code. (Relevant to 1.4.2 and 5.0 only.) [1.4.2, multi-processor Windows platforms: true] |
| -XX:+UseTLAB | Use thread-local object allocation (Introduced in 1.4.0, known as UseTLE prior to that.) [1.4.2 and earlier, x86 or with -client: false] |
| -XX:+UseSplitVerifier | Use the new type checker with StackMapTable attributes. (Introduced in 5.0.)[5.0: false] |
| -XX:+UseThreadPriorities | 使用本地线程优先级. |
| -XX:+UseVMInterruptibleIO | Thread interrupt before or with EINTR for I/O operations results in OS_INTRPT. (Introduced in 6. Relevant to Solaris only.) |

**CPU** 如果当前CPU已经能够接近100%的利用率，并且代码业务逻辑无法再删减，那么说明该系统的已经达到了性能最大化，如果再想提高性能，只能增加处理器(增加更多的机器或者安装更多的CPU)

**其它资源** 如数据库连接数量等，如果CPU利用率没有接近100%，那么通过修改代码(当然是有用代码)尽量提高CPU的使用率，那么整体性能也会获得提高。

**表 14 Hibernate与JDBC的对比**

| 角度 | Hibernate | JDBC |
|---|---|---|
| 性能 | dsdddd | dddd |
| 可维护 | ffff | |

许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄
许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄
许用户安装自333333333

许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄

许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄许用户安装自己的信号处理句柄

下面是另一个例子：

结果结果结果结

结果结果结果结果结果结果结果结果结果结果结果
结果结果结果结果结果结果结果结果结果结果结果
结果结果结果结果结果结果结果结果结果结果结果
结果结果结果结果结果结果结果结果结果结果结果
结果结果结果结果结果结果结果结果结果结果结果
结果结果结果结果结果结果结果结果结果结果结果
结果结果结果结果结果结果结果结果结果

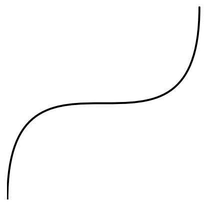


图 48 曲线

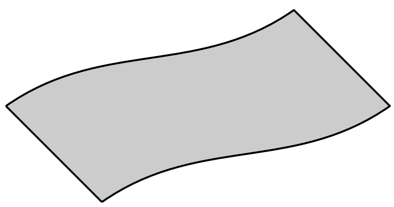
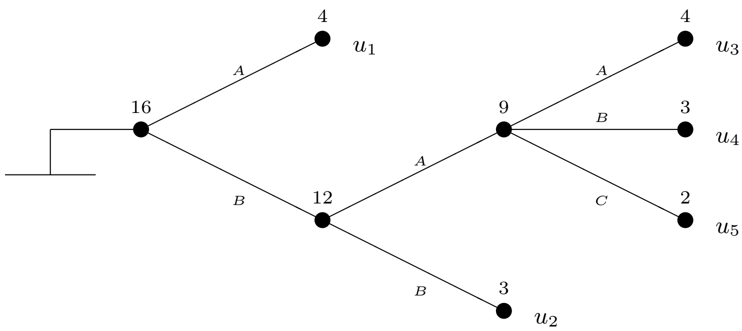


图 49 测试



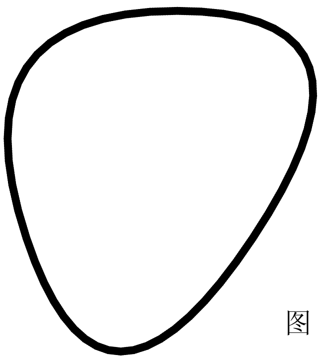
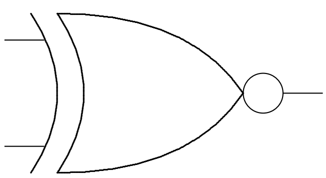



图 50 This is a box.



```
function sqrt(x: integer): integer;
```

```
(* sqrt(x) = √x *)
function pow(x,y: real): real;
(* pow(x,y) = x^y *)
            while( k > 1)

                int j = k >> 1;   //locate its father

                if(m_queue[j].value < m_queue[k].value)
                    break;

                Item tmp = m_queue[j];
                m_queue[j] = m_queue[k];
                m_queue[j].index = j;

                m_queue[k] = tmp;
                m_queue[k].index = k;

                k=j;

```

```
            while( k > 1)
            {
                int j = k >> 1;   //locate its father

                if(m\_queue[j].value < m\_queue[k].value)
                    break;

                Item tmp = m\_queue[j];
                m\_queue[j] = m\_queue[k];
                m\_queue[j].index = j;

                m\_queue[k] = tmp;
                m\_queue[k].index = k;

                k=j;
            }
```

```
            while( k > 1)
            {
                int j = k >> 1;   //locate its father

                if(m\_queue[j].value < m\_queue[k].value)
                    break;

                Item tmp = m\_queue[j];
                m\_queue[j] = m\_queue[k];
                m\_queue[j].index = j;

                m\_queue[k] = tmp;
```

```
            m\_queue[k].index = k;

            k=j;
        }
```

```
CREATE OR REPLACE FUNCTION
    add(x int , y int) RETURNS int
AS $add$
DECLARE
    sum integer;      -- local variable
BEGIN
    sum := x + y;
    return sum;       -- return result
END;
$add$ LANGUAGE plpgsql
#\HL#IMMUTABLE#\HLoff#
#\HL#RETURNS NULL ON NULL INPUT#\HLoff#;
```

| | |
|---|---|
| oneffffffffffffffffffffffffff | jjjjjjjjjjjj |
| three | four |

[ GC]Garbage Collector This is Garbage

[ GC]Garbage Collector This is Garbage

[ GC]Garbage Collector This is Garbage

[ GC]Garbage Collector This is Garbage

**GC** Garbage Collector

**GC** Garbage Collector

**GC** Garbage Collector

☞ 提示:

钩子函数不应该执行任何耗时的操作，而且应该是线程安全的，不应该依赖于其它任何服务，因为整个系统都在关闭自己的过程中，不能将自己的命运寄托于其它可能状态已经不正常的服务上。

☞钩子函数不应该执行任何耗时的操作，而且应该是线程安全的，不应该依赖于其它任何服务，因为整个系统都在关闭自己的过程中，不能将自己的命运寄托于其它可能状态已经不正常的服务上。

Project: Total Requirements = \$900 000.00
of which 2003 = \$450 000.00
2004 = \$350 000.00
2005 = \$100 000.00

2003 approved: \$350 000.00 Deficiency: \$100 000.00

| | | | | |
|---|---|---|---|---|
| 2004 | | $300 000.00 | | $150 000.00 |
| 2005 | | $250 000.00 | Surplus: | $150 000.00 |

| | | | | | |
|---|---|---|---|---|---|
| | tentative | 2004 | = $100 000.00 | for deficiency | 2003 |
| | | 2005 | = $ 50 000.00 | | 2004 |
| | | | + $100 000.00 | excess for | 2003 in 2004 |
| Commitments | | 2003 | = $100 000.00 | | |
| | | 2004 | = $150 000.00 | | |

signed: H. André

<table>
<tr><td rowspan="2">Day</td><td colspan="2">6.15–7.15 pm</td><td colspan="2">7.20–8.20 pm</td><td colspan="2">8.30–9.30 pm</td></tr>
<tr><td>Subj.</td><td>Teacher<br>Room</td><td>Subj.</td><td>Teacher<br>Room</td><td>Subj.</td><td>Teacher<br>Room</td></tr>
<tr><td>Mon.</td><td>UNIX</td><td>Dr. Smith<br>Comp. Ctr</td><td>Fortran</td><td>Ms. Clark<br>Hall A</td><td>Math.</td><td>Mr. Mills<br>Hall A</td></tr>
<tr><td>Tues.</td><td>LaTeX</td><td>Miss Baker<br>Conf. Room</td><td>Fortran</td><td>Ms. Clark<br>Conf Room</td><td>Math.</td><td>Mr. Mill<br>Hall A</td></tr>
<tr><td>Wed.</td><td>UNIX</td><td>Dr. Smith<br>Comp. Ctr</td><td>C</td><td>Dr. Jones<br>Hall A</td><td>ComSci.</td><td>Dr. Jones<br>Hall A</td></tr>
<tr><td>Fri.</td><td>LaTeX</td><td>Miss Baker<br>Conf. Room</td><td>C++</td><td>Ms. Clark<br>Conf. Room</td><td colspan="2">canceled</td></tr>
</table>

| Course and Date | Brief Description | Prerequisites |
|---|---|---|
| Introduction to LSEDIT<br>14.3. – 16.3. | Logging on — Explanation to the VMS fils system — Explanation and intensive application of the VMS editor LSEDIT — user modifications | none |
| Introlduction to LaTeX<br>21.3. – 25.3. | Word processors and formatting programs — text and commands — environments — dokument and page styles — displayed text — math equations — simple user defined structures | LSEDIT |

| | | | |
|---|---|---|---|
| Project: | Total Requirements | | = $900 000.00 |
| | of which | 2003 | = $450 000.00 |
| | | 2004 | = $350 000.00 |
| | | 2005 | = $100 000.00 |

| | | | |
|---|---|---|---|
| 2003 approved: | $350 000.00 | Deficiency: | $100 000.00 |
| 2004 | $300 000.00 | | $150 000.00 |
| 2005 | $250 000.00 | Surplus: | $150 000.00 |

| | | | | | |
|---|---|---|---|---|---|
| | tentative | 2004 | = $100 000.00 | for deficiency | 2003 |
| | | 2005 | = $ 50 000.00 | | 2004 |
| | | | + $100 000.00 | excess for | 2003 in 2004 |
| Commitments | | 2003 | = $100 000.00 | | |
| | | 2004 | = $150 000.00 | | |

signed: H. André

**Primary Energy Consumption**

| Energy Source | | 1975 | 1980 | 1986 |
|---|---|---|---|---|
| Total Consumption | | | | |
| (in million tons of BCU)[64] | | 347.7 | 390.2 | 385.0 |
| of which | (percentages) | | | |
| | petroleum | 52.1 | 47.6 | 43.2 |
| | bituminous coal | 19.1 | 19.8 | 20.0 |
| | brown coal | 9.9 | 10.0 | 8.6 |
| | natural gas | 14.2 | 16.5 | 15.1 |
| | nuclear Energy | 2.0 | 3.7 | 10.1 |
| | other[65] | 2.7 | 2.3 | 3.0 |

*Source:* Energy Balance Study Group, Essen 1987.

**Answers to the questions from exercise 4.15**:

1. With `@{}` at the beginning and end of the formatting definition the additional space of half of the column gap size in front and behind the table which occures by default, will be suppressed.

2. With `@{\extracolsep{\hfill}}` at the beginning of the formatting definition additional space of equal width would be inserted between all following columns so that the total tabular with becomes exatly the ordered width of 118 mm. Try this modification as special exersice.

3. `@{\extracolsep{\hfill}}` is to put behind the second column parameter and its countermanding `@{\hspace{1em}}@\extracolsep{1em}}` is to put behind the third column parameter. With `@{\extracolsep{1em}}` as countermanding expression the gap between the third and fourth column would be smaller than those for the following column. Try this as alternative exercise.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **1st Regional Soccer League — Final Results 2002/03** | | | | | | | | |
| | *Club* | *W* | *T* | *L* | *Goals* | *Points* | *Remarks* |
| 1 | Amesville Rockets | 19 | 13 | 1 | 66:31 | 51:15 | League Champs |
| 2 | Borden Comets | 18 | 9 | 6 | 65:37 | 45:21 | Trophy Winners |
| 3 | Clarkson Chargers | 17 | 7 | 9 | 70:44 | 41:25 | Candidates for National League |
| 4 | Daysdon Bombers | 14 | 10 | 9 | 66:50 | 38:28 | |
| 5 | Edgartowns Devils | 16 | 6 | 11 | 63:53 | 38:28 | |
| 6 | Freeburg Fighters | 15 | 7 | 11 | 64:47 | 37:29 | |
| 7 | Gadsby Tigers | 15 | 7 | 11 | 52:37 | 35:31 | Medium Teams |
| 8 | Harrisville Hotshots | 12 | 11 | 10 | 62:58 | 35:31 | |
| 9 | Idlleton Shovers | 14 | 9 | 11 | 49:51 | 35:31 | |
| 10 | Jamestown Horntets | 11 | 11 | 11 | 48:47 | 33:33 | |
| 11 | Kingston Cowboys | 13 | 5 | 14 | 54:45 | 32:34 | |
| 12 | Lonsdale Stompers | 12 | 8 | 13 | 50:57 | 32:34 | |
| 13 | Marsdon Heroes | 9 | 12 | 11 | 50:42 | 31:35 | |
| 14 | Norburg Flames | 10 | 8 | 15 | 60:68 | 28:38 | |
| 15 | Ollison Champions | 8 | 9 | 16 | 42:49 | 25:41 | |
| 16 | Petersville Lancers | 6 | 8 | 19 | 31:77 | 20:46 | Disbanding |
| 17 | Quincy Giants | 7 | 5 | 21 | 40:89 | 19:47 | Denoted |
| 19 | Ralstone Regulars | 3 | 11 | 19 | 37:74 | 17:49 | |

This is my red color This is red color

参考椭球面的法线

---

注意:

---

一行对齐: 左对齐

居中

右对齐

多行对齐:

...

,

...

,

...

# 参考文献


[1] http://www.ibm.com/developerworks/cn/java/j-jtp11253/index.html

[2] http://www.ibm.com/developerworks/cn/java/j-jtp0924/index.html

[3] http://www.codeproject.com/jscript/leakpatterns.asp

[4] http://msdn2.microsoft.com/en-us/library/bb250448.aspx

[5] http://www-128.ibm.com/developerworks/web/library/wa-memleak/?ca=dgr-btw01Javascriptleaks

[6] http://laurens.vd.oever.nl/weblog/items2005/closures

[7] http://www.jibbering.com/faq/faq_notes/closures.html

[8] http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf

[9] http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html

[10] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Process.html

[11] http://download-uk.oracle.com/docs/cd/B19306_01/server.102/b14220/consist.htm#i5337

[12] http://www.ibm.com/developerworks/cn/java/j-dyn0203/

[13] http://dev2dev.bea.com/pub/a/2004/04/jrockit142_hirt.html

[14] http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/java.html

[15] http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp

[16] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html

[17] http://java.sun.com/j2se/1.5.0/fixedbugs/fixedbugs.html

[18] http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html

[19] http://martin.nobilitas.com/java/sizeof.html

[20] http://java.sun.com/developer/technicalArticles/programming/hprof.html

[21] http://www.bea.com.cn/support_pattern/Investigating_Out_of_Memory_Memory_Leak_Pattern.html

[22] http://www-128.ibm.com/developerworks/cn/java/l-JavaMemoryLeak/

[23] http://www.cs.umd.edu/ pugh/java/memoryModel/DoubleCheckedLocking.html

[24] http://tech.ccidnet.com/art/1077/20050413/237901_2.html

[25] http://www.chinaitpower.com/A200508/2005-08-10/189843.html

[26] http://www.ibm.com/developerworks/cn/linux/es-JITDiag.html

[27] http://publib.boulder.ibm.com/infocenter/javasdk/v5r0/index.jsp?
topic=/com.ibm.java.doc.diagnostics.50/html/jitpd_failing_method.html

[28] http://java.sun.com/javase/6/webnotes/trouble/TSG-VM/html/docinfo.html

[29] http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag142.pdf

[30] http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf

[31] http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag60.pdf

[32] http:/www.innovatedigital.com

[33] http://docs.sun.com/source/819-0084/pt_tuningjava.html

[34] http://www.ibm.com/developerworks/aix/library/au-javaonaix_memory.html