Java Dump分析

研究院—侯春磊

概述

SINCE 1997

概述

- 问题一：什么是Java Dump？
- 问题二：Java Dump有什么用？

问题一：什么是Java Dump？

Java虚拟机的运行时快照。将Java虚拟机运行时的状态和信息保存到文件。

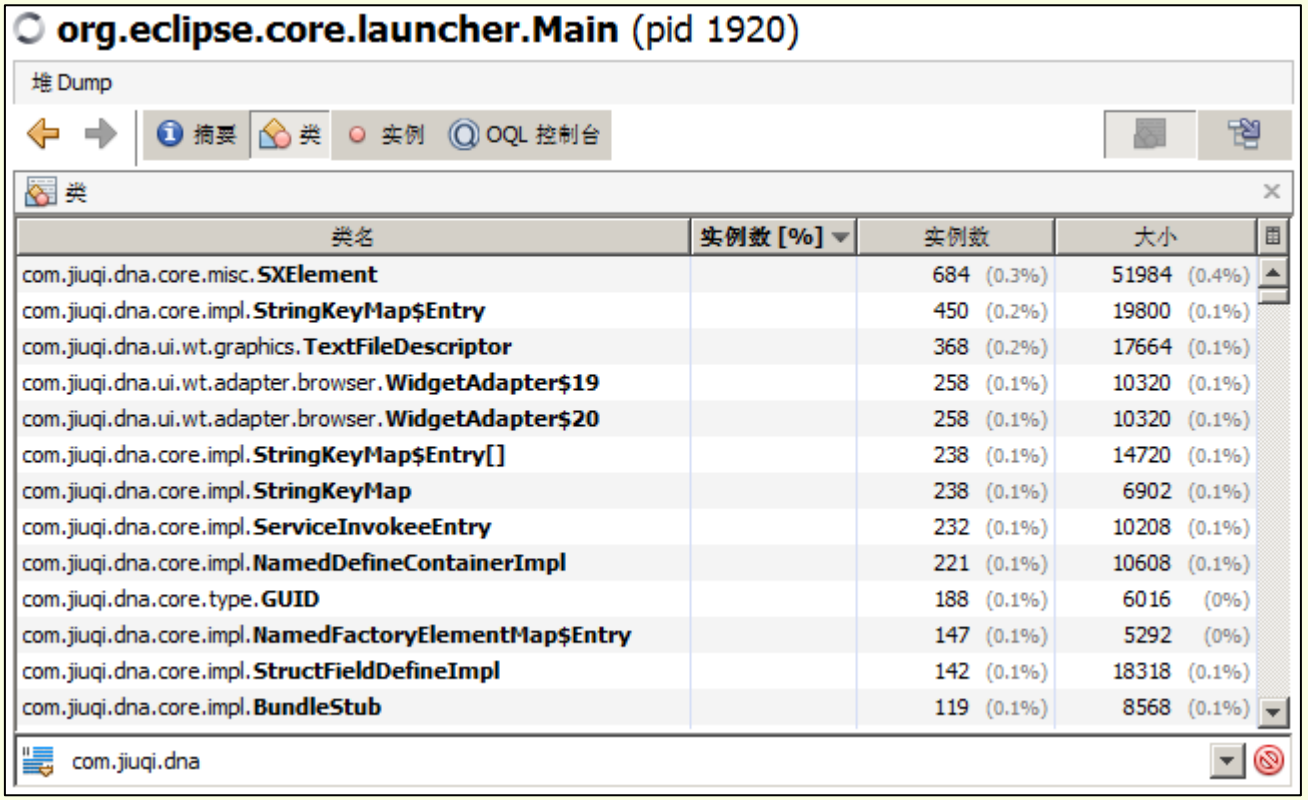
Java Dump文件

- 线程Dump，包含所有线程的运行状态。纯文本格式。
- 堆Dump，包含线程Dump，并包含所有堆对象的状态。二进制格式。

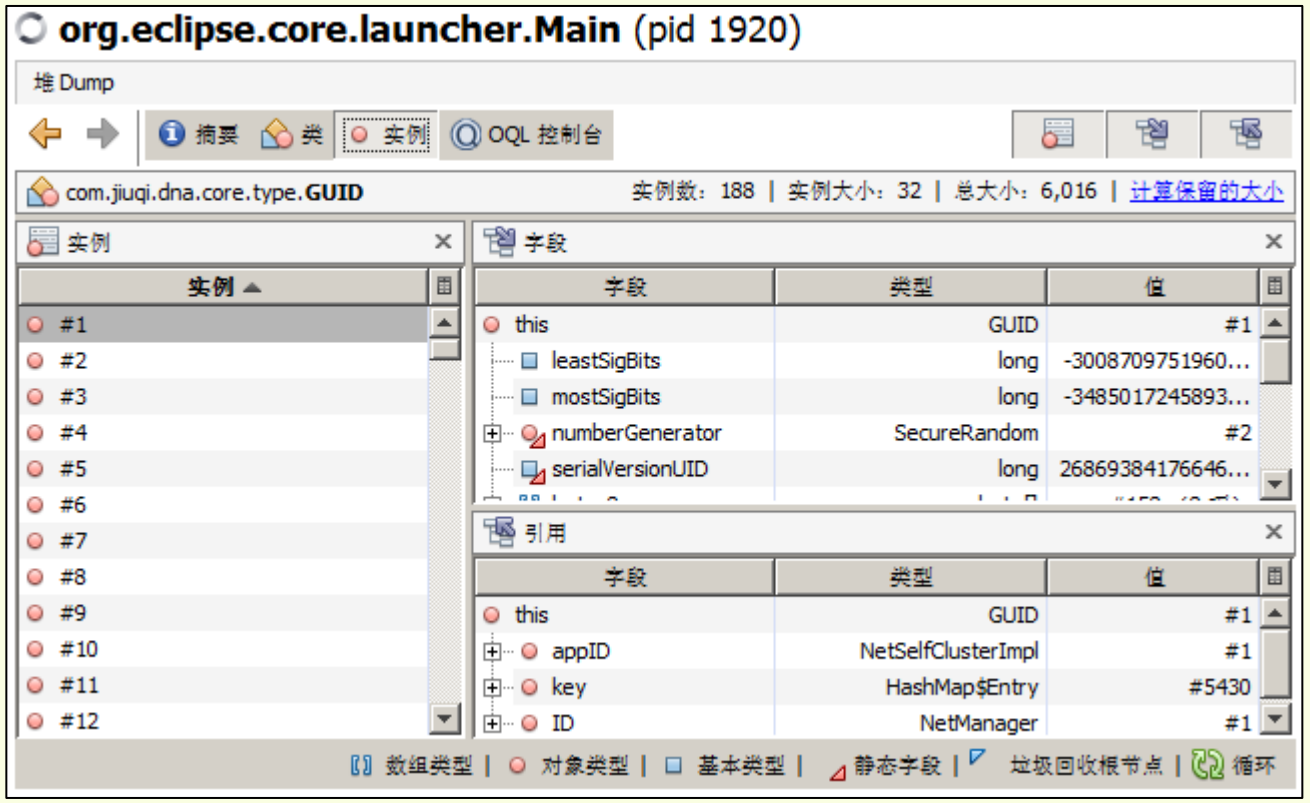
线程Dump示例

```
2011-05-25 19:30:16
Full thread dump Java HotSpot(TM) 64-Bit Server VM (19.1-b02 mixed mode):

"pool-3-thread-1" prio=6 tid=0x000000000711e800 nid=0x1274 runnable [0x000000000c1
   java.lang.Thread.State: RUNNABLE
  at java.net.SocketInputStream.socketRead0(Native Method)
  at java.net.SocketInputStream.read(SocketInputStream.java:129)
  at com.sun.net.ssl.internal.ssl.InputRecord.readFully(InputRecord.java:293)
  at com.sun.net.ssl.internal.ssl.InputRecord.read(InputRecord.java:331)
  at com.sun.net.ssl.internal.ssl.SSLSocketImpl.readRecord(SSLSocketImpl.java:798)
  - locked <0x00000000fb677450> (a java.lang.Object)
  at com.sun.net.ssl.internal.ssl.SSLSocketImpl.performInitialHandshake(SSLSocketI

"Image Fetcher 3" daemon prio=8 tid=0x000000000711d000 nid=0x1354 in Object.wait()
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
  at java.lang.Object.wait(Native Method)
  - waiting on <0x00000000f0521a60> (a java.util.Vector)

"Thread-3" daemon prio=6 tid=0x0000000008a84800 nid=0xd34 in Object.wait() [0x0000
   java.lang.Thread.State: WAITING (on object monitor)
  at java.lang.Object.wait(Native Method)
  - waiting on <0x00000000f04a85f8> (a java.util.LinkedList)

"VM Thread" prio=10 tid=0x00000000024d7000 nid=0xcd4 runnable

"GC task thread#0 (ParallelGC)" prio=6 tid=0x00000000004ec000 nid=0x11a4 runnable
```

堆Dump示例


org.eclipse.core.launcher.Main (pid 1920)
堆 Dump
摘要
类
实例
OQL 控制台
类
类名
实例数 [%]
实例数
大小
com.jiuqi.dna.core.misc.SXElement 684 (0.3%) 51984 (0.4%)
com.jiuqi.dna.core.impl.StringKeyMap$Entry 450 (0.2%) 19800 (0.1%)
com.jiuqi.dna.ui.wt.graphics.TextFileDescriptor 368 (0.2%) 17664 (0.1%)
com.jiuqi.dna.ui.wt.adapter.browser.WidgetAdapter$19 258 (0.1%) 10320 (0.1%)
com.jiuqi.dna.ui.wt.adapter.browser.WidgetAdapter$20 258 (0.1%) 10320 (0.1%)
com.jiuqi.dna.core.impl.StringKeyMap$Entry[] 238 (0.1%) 14720 (0.1%)
com.jiuqi.dna.core.impl.StringKeyMap 238 (0.1%) 6902 (0.1%)
com.jiuqi.dna.core.impl.ServiceInvokeeEntry 232 (0.1%) 10208 (0.1%)
com.jiuqi.dna.core.impl.NamedDefineContainerImpl 221 (0.1%) 10608 (0.1%)
com.jiuqi.dna.core.type.GUID 188 (0.1%) 6016 (0%)
com.jiuqi.dna.core.impl.NamedFactoryElementMap$Entry 147 (0.1%) 5292 (0%)
com.jiuqi.dna.core.impl.StructFieldDefineImpl 142 (0.1%) 18318 (0.1%)
com.jiuqi.dna.core.impl.BundleStub 119 (0.1%) 8568 (0.1%)
com.jiuqi.dna

堆Dump示例


org.eclipse.core.launcher.Main (pid 1920)
堆 Dump
摘要
类
实例
OQL 控制台
com.jiuqi.dna.core.type.GUID
实例数：188 | 实例大小：32 | 总大小：6,016 | 计算保留的大小
实例
字段
引用
实例
#1
#2
#3
#4
#5
#6
#7
#8
#9
#10
#11
#12
字段
类型
值
this
GUID
#1
leastSigBits
long
-3008709751960...
mostSigBits
long
-3485017245893...
numberGenerator
SecureRandom
#2
serialVersionUID
long
26869384176646...
this
GUID
#1
appID
NetSelfClusterImpl
#1
key
HashMap$Entry
#5430
ID
NetManager
#1
数组类型 | 对象类型 | 基本类型 | 静态字段 | 垃圾回收根节点 | 循环

问题二：Java Dump有什么用？

如何分析程序错误（Bug）的产生原因？

- 调试、远程调试

  能否搭建调试环境？允许调试？能否捕捉到异常？

- 控制台输出，结合代码阅读

  信息较少，经常不足以分析问题。

从程序的开发阶段来看待Bug

开发	功能测试	性能测试	生产环境

功能正确性的Bug	非功能性的Bug

调试	控制台输出、代码阅读	JavaDump分析

Java Dump的意义

- Java进程的快照，保存状态和信息可用于Bug的分析等。
- 补足传统Bug分析手段的不足：
  - 可在任何Java环境使用；信息量充足。
  - 针对非功能正确性的Bug，主要为：多线程并发、内存泄漏。

Java Dump的使用者

- 制作Dump：实施、测试、开发。
- 分析Dump：主要面向开发人员，部分测试人员。

制作DUMP

SINCE 1997

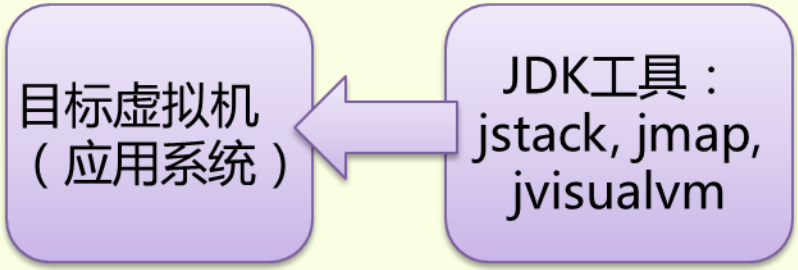
制作Java Dump

在不同的操作系统平台、不同的Java虚拟机环境下，使用图形化或命令行工具，生成指定Java进程的Dump并保存到文件。

制作Java Dump

1. 制作Java Dump的原理
2. 常见的Java虚拟机
3. Java Dump的格式
4. 制作Java Dump
   - 虚拟机参数
   - 图形化环境
   - 命令行环境

制作Dump的原理



- 使用JDK提供的工具，连接目标虚拟机，制作Dump。
- 目标和工具都使用1.6或以上的Java；发行版相同。

常见的Java虚拟机

不同Java虚拟机的Dump规范不完全相同。

- HotSpot VM：Sun官方的Java虚拟机实现。
- OpenJDK：开源版本的虚拟机实现。
- JRockit：BEA开发的，Weblogic使用的虚拟机。
- IBM J9 VM：AIX平台的Java虚拟机。

Dump的格式

- 线程Dump：纯文本，各虚拟机发行版略微不同。
- 堆Dump：二进制格式，
  - HotSpot VM与OpenJDK：HPROF格式。
  - JRockit：没有堆Dump，使用飞行记录。
  - IBM J9 VM：IBM Portable Heap Dump（PHD）格式。

制作Dump

- Java虚拟机参数
- 图形化工具
- 命令行工具

使用Java虚拟机制作Dump

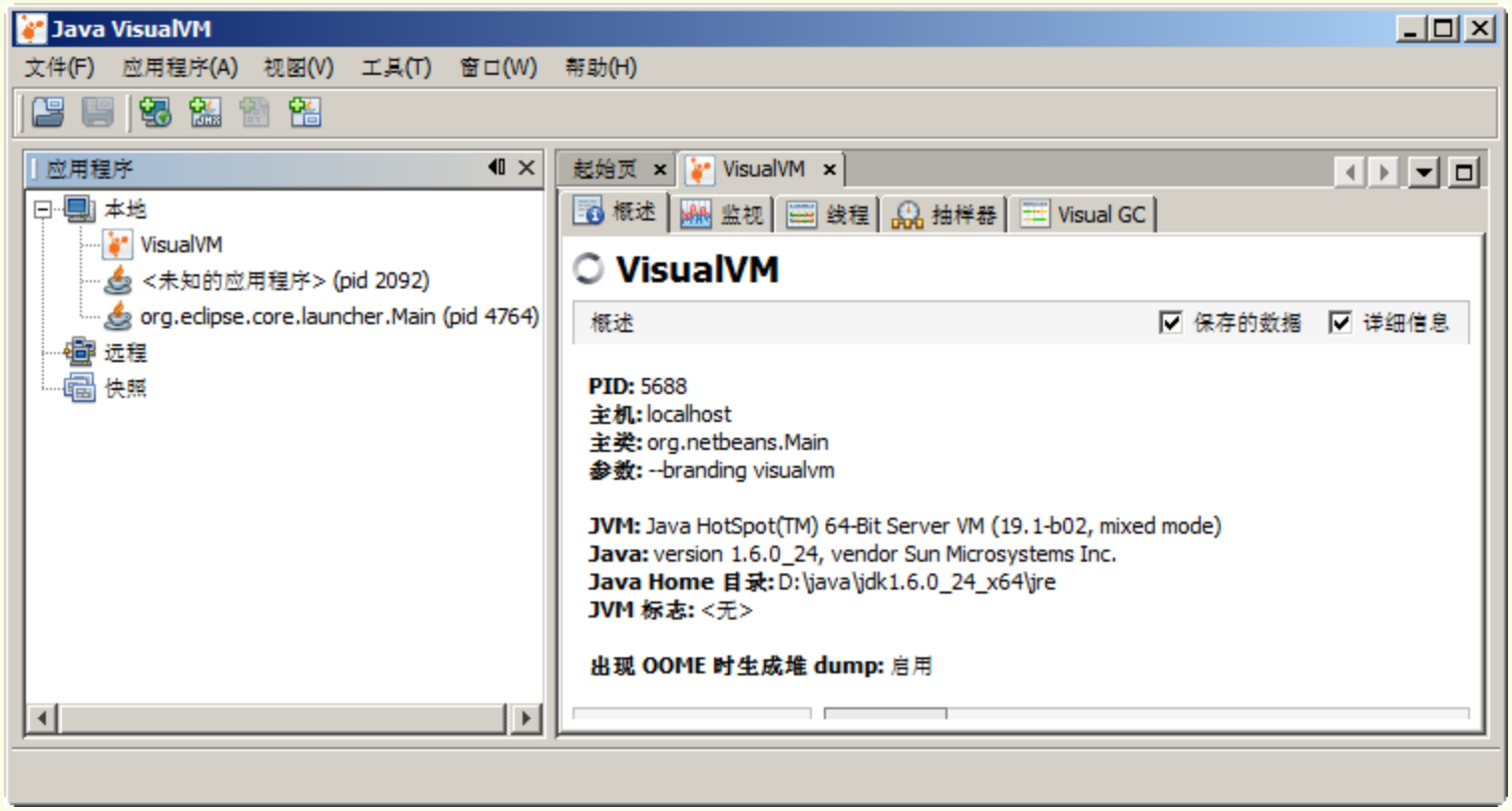
- 指示虚拟机在发生内存不足错误时，自动生成堆Dump

  `-XX:+HeapDumpOnOutOfMemoryError`

- 只在HotSpot支持；1.5_07之后支持。
- 生产环境都需要添加该参数，保留现场。

使用图形化工具制作Dump

使用JDK（1.6）自带的工具：Java VisualVM。



使用图形化工具制作Dump

- 使用JDK，而非JRE，且版本在1.6或以上。
- Linux平台下同样包含该工具，且操作与Windows一致。
- 远程连接Java进程涉及到比较复杂的权限配置，一般不建议使用。所以尽可能的从本机制作Dump。
- 生成Dump后，需要另存到文件。
- Java虚拟机内存较大时，制作堆Dump前先GC。

使用命令行制作Dump

在JDK的bin目录下，包含了java命令及其他实用工具。

- jps：查看本机的Java进程信息。
- jstack：打印线程的栈信息，制作线程Dump。
- jmap：打印内存映射，制作堆Dump。
- jconsole：简易的可视化控制台。
- jvisualvm：功能强大的控制台。
- jstat：性能监控工具。
- jhat：内存分析工具。

使用命令行制作Dump

1. 检查虚拟机版本
2. 找出目标Java应用的进程ID
3. 使用jstack命令制作线程Dump
    - Linux环境下使用kill命令制作线程Dump
4. 使用jmap命令制作堆Dump

使用命令行制作Dump

- 使用 java -version 查看虚拟机版本

```
C:\>java -version
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07)
Java HotSpot(TM) 64-Bit Server VM (build 19.1-b02, mixed mode)
```

```
[dev@localhost bin]$ java -version
java version "1.6.0_22"
OpenJDK Runtime Environment (IcedTea6 1.10.6) (rhel-
1.43.1.10.6.el6_2-x86_64)
OpenJDK 64-Bit Server VM (build 20.0-b11, mixed mode)

[dev@localhost bin]$ ./java -version
java version "1.6.0_31"
Java(TM) SE Runtime Environment (build 1.6.0_31-b04)
Java HotSpot(TM) 64-Bit Server VM (build 20.6-b01, mixed mode)
```

使用命令行制作Dump

- 使用jps查看Java进程ID（PID）；Linux下还可以使用ps命令。

  jps 或者 jps -l 或者 jps -v

  ps -ef|grep java

```
[dev@localhost bin]$./jps -l
1018 org.netbeans.Main
1223 sun.tools.jps.Jps
2316 /home/dev/dna/srvmgr/thr/com.jiuqi.dna.launcher_1.0.0.jar
```

使用命令行制作线程Dump

- 使用jstack制作线程Dump。

  jstack <进程ID> >> <输出文件>

```
C:\>jstack 2316 >> c:\thread.txt
```

```
[dev@localhost bin]$ ./jstack 2316 >> thread.txt
```

Linux下使用Kill命令制作线程Dump

- 输出线程Dump到目标Java进程的标准输出。

  kill -quit <进程ID>

  kill -3 <进程ID>

```
[dev@localhost bin]$ kill -quit 2316
```

```
[dev@localhost bin]$ kill -3 2316
```

使用命令行制作堆Dump

- 使用jmap命令制作堆Dump

```
jmap -dump:format=b,file=<输出文件> <进程ID>
```

```
C:\>jmap -dump:format=b,file=c:\heap.hprof 2316
Dumping heap to C:\heap.hprof ...
Heap dump file created
```

```
[dev@localhost bin]$ ./jmap -dump:format=b,file=heap.hprof
2316
Dumping heap to /usr/java/1.6.0_31/bin/heap.hprof ...
Heap dump file created
```

HotSpot VM制作Dump的总结

	线程Dump	堆Dump
图形化	Java VisualVM	Java VisualVM
命令行	jstack	jmap

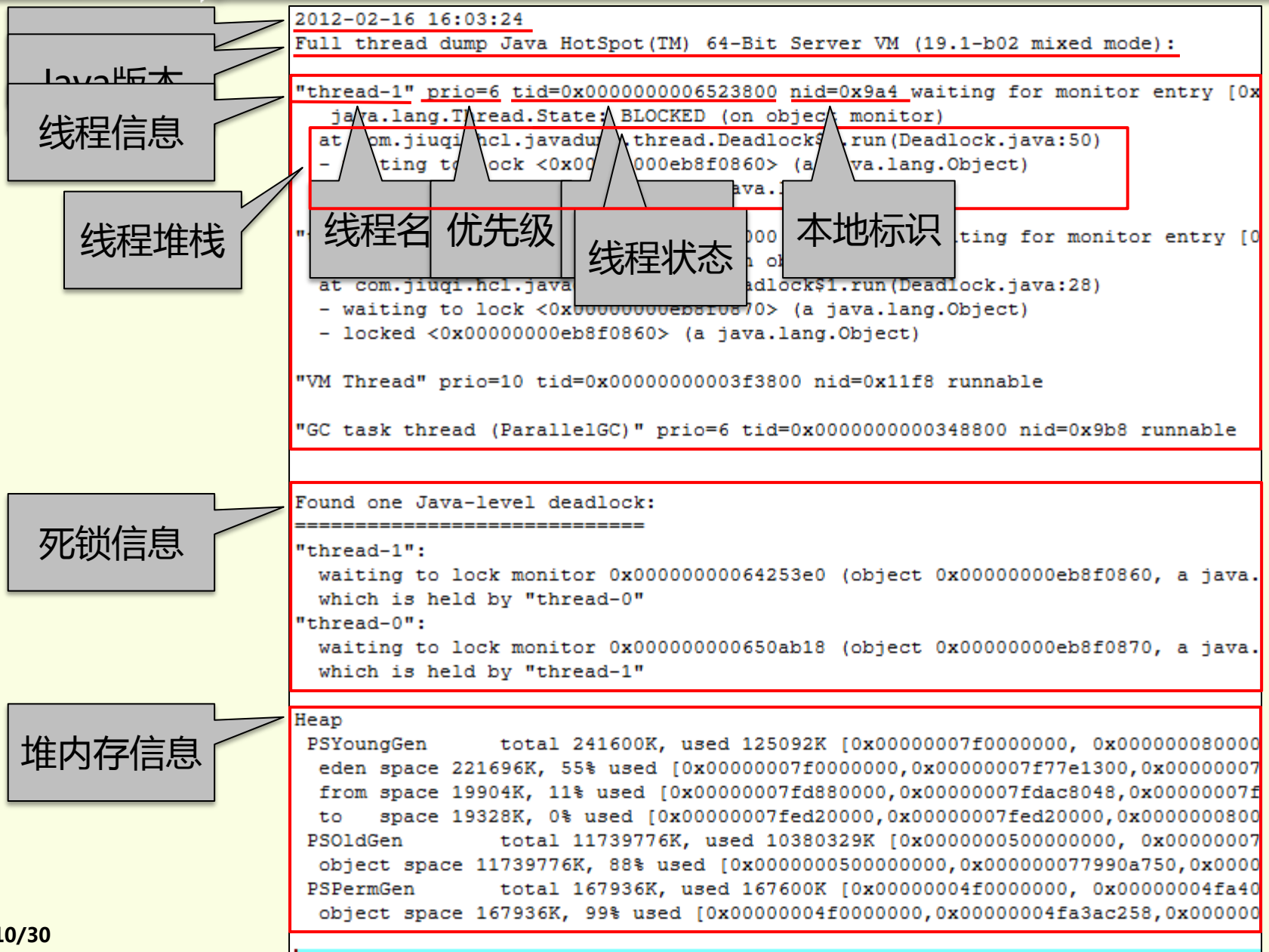
分析线程DUMP

SINCE 1997

分析线程Dump

1. 线程Dump的内容
2. 使用场景、针对的问题
3. 相关的Java机制
4. 常用的工具介绍
5. 分析模式

线程Dump的内容


Java版本
线程信息
线程堆栈
死锁信息
堆内存信息
线程名
优先级
线程状态
本地标识
2012-02-16 16:03:24
Full thread dump Java HotSpot(TM) 64-Bit Server VM (19.1-b02 mixed mode):
"thread-1" prio=6 tid=0x0000000006523800 nid=0x9a4 waiting for monitor entry [0x
java.lang.Thread.State: BLOCKED (on object monitor)
at com.jiuqi.hcl.javadu thread.Deadlock$ .run(Deadlock.java:50)
- ting to ock <0x00 000eb8f0860> (a va.lang.Object)
ava.
ting for monitor entry [0
at com.jiuqi.hcl.java adlock$1.run(Deadlock.java:28)
- waiting to lock <0x 70> (a java.lang.Object)
- locked <0x00000000eb8f0860> (a java.lang.Object)
"VM Thread" prio=10 tid=0x00000000003f3800 nid=0x11f8 runnable
"GC task thread (ParallelGC)" prio=6 tid=0x0000000000348800 nid=0x9b8 runnable
Found one Java-level deadlock:
==========
"thread-1":
waiting to lock monitor 0x00000000064253e0 (object 0x00000000eb8f0860, a java.
which is held by "thread-0"
"thread-0":
waiting to lock monitor 0x000000000650ab18 (object 0x00000000eb8f0870, a java.
which is held by "thread-1"
Heap
PSYoungGen total 241600K, used 125092K [0x00000007f0000000, 0x000000080000
eden space 221696K, 55% used [0x00000007f0000000,0x00000007f77e1300,0x00000007
from space 19904K, 11% used [0x00000007fd880000,0x00000007fdac8048,0x00000007f
to space 19328K, 0% used [0x00000007fed20000,0x00000007fed20000,0x0000000800
PSOldGen total 11739776K, used 10380329K [0x0000000500000000, 0x00000007
object space 11739776K, 88% used [0x0000000500000000,0x000000077990a750,0x0000
PSPermGen total 167936K, used 167600K [0x00000004f0000000, 0x00000004fa40
object space 167936K, 99% used [0x00000004f0000000,0x00000004fa3ac258,0x000000

线程Dump的内容

- 制作时间
- Java 版本
- 线程信息：

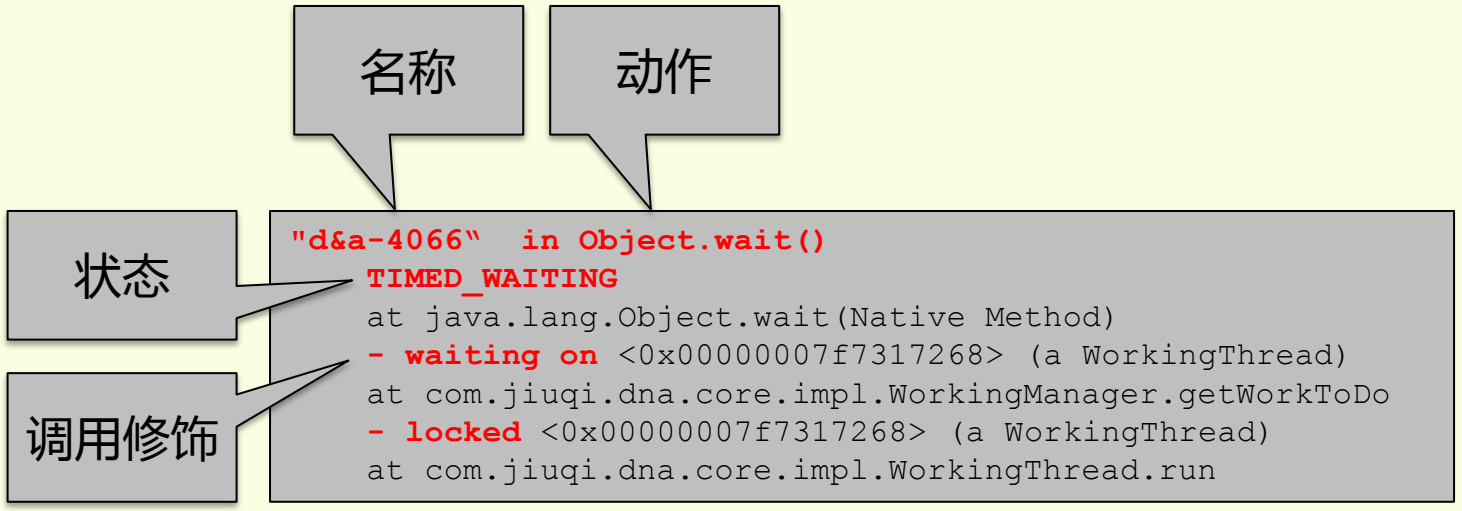
  名称、优先级、标识、状态、堆栈
- 死锁信息：存在直接Java线程的死锁时才包含。
- 内存信息：使用`kill`制作时才包含。

线程Dump使用场景

- 系统无响应或响应慢，不知道系统状态时。
- 多线程并发相关的问题，如：死锁、阻塞等。

线程信息

```
"d&a-4066" daemon prio=10 tid=0x000000004b12d000 nid=0x231b in Object.wait()
[0x000000005b1cb000]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
          at java.lang.Object.wait(Native Method)
          - waiting on <0x00000007f7317268> (a com.jiuqi.dna.core.impl.WorkingThread)
          at com.jiuqi.dna.core.impl.WorkingManager.getWorkToDo(WorkingManager.java:322)
          - locked <0x00000007f7317268> (a com.jiuqi.dna.core.impl.WorkingThread)
          at com.jiuqi.dna.core.impl.WorkingThread.run(WorkingThread.java:40)
```



相关Java机制

- 线程状态
- 监视器
- 调用修饰
- 线程动作

线程状态

- NEW，未启动的。不会出现在Dump中。
- RUNNABLE，在虚拟机内执行的。
- BLOCKED，受阻塞并等待监视器锁。
- WATING，无限期等待另一个线程执行特定操作。
- TIMED_WATING，有时限的等待另一个线程的特定操作。
- TERMINATED，已退出的。

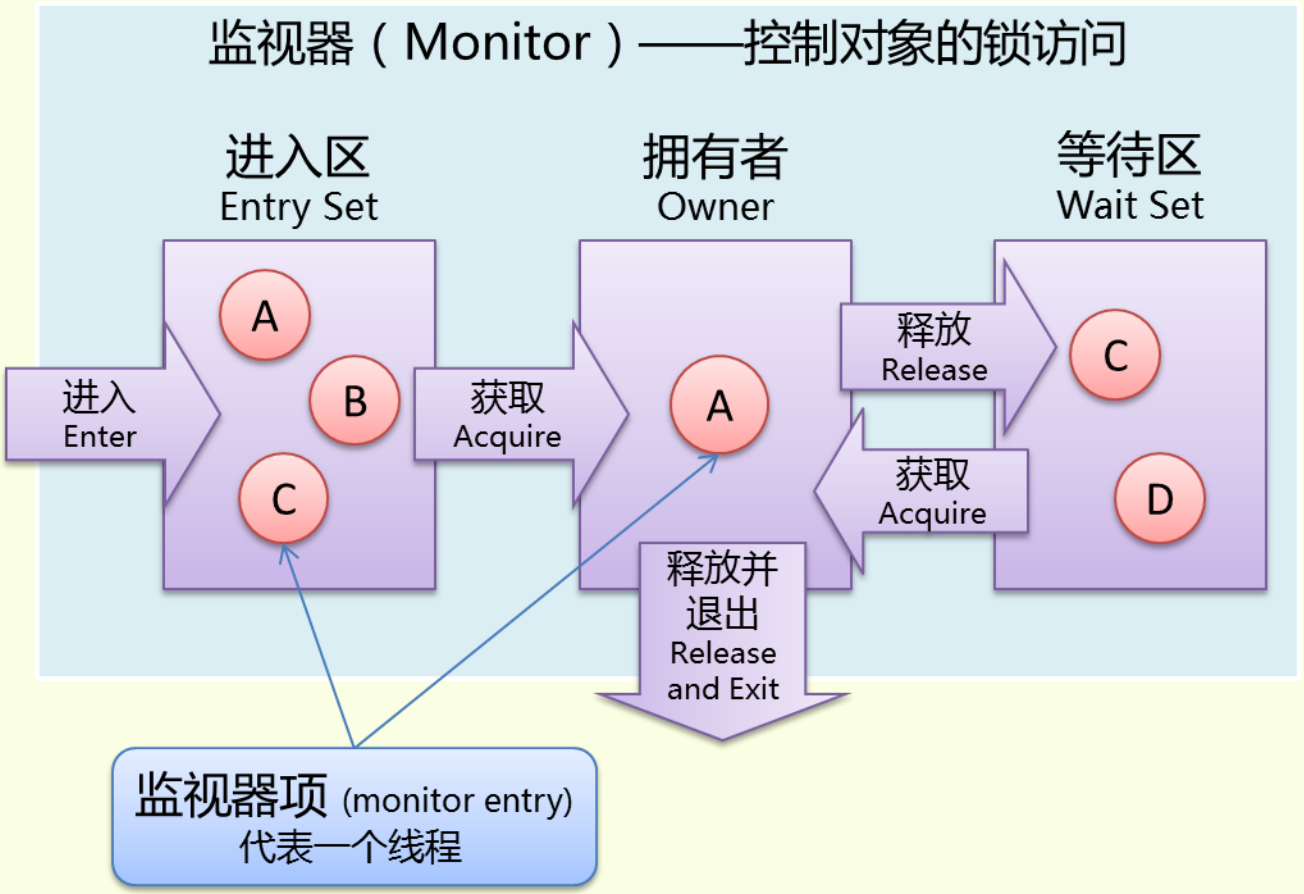
监视器（Monitor）

- 当使用synchronized定义同步块时时，监视器是用来控制对象的锁的并发访问的结构。

```
synchronized(obj){
    // 同步块，只允许一个线程进入
}
```

```
synchronized void method(){
    // 同步块，只允许一个线程进入
}
```

监视器（Monitor）


监视器（Monitor）——控制对象的锁访问
进入区
Entry Set
拥有者
Owner
等待区
Wait Set
A
B
C
进入
Enter
获取
Acquire
A
释放
Release
C
获取
Acquire
D
释放并
退出
Release
and Exit
监视器项 (monitor entry)
代表一个线程

监视器

- 监视器：对象锁的访问控制结构。也指对象的锁。
- 监视器项：线程的代理人。
- 进入区：表示线程通过synchronized要求获取对象的锁。如果对象未被锁住，则进入拥有者；否则则在进入区等待。一旦对象锁被其他线程释放，立即参与竞争。
- 拥有者：表示某一线程成功竞争到对象锁。
- 等待区：表示线程通过对象的wait方法，释放对象的锁，并在等待区等待被唤醒。

调用修饰

```
"d&a-4050" daemon in Object.wait()
  java.lang.Thread.State: TIMED_WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x00000007f6fba430> (a WorkingThread)
    at com.jiuqi.dna.core.impl.WorkingManager.getWorkToDo(WorkingManager.java)
    - locked <0x00000007f6fba430> (a WorkingThread)
    at com.jiuqi.dna.core.impl.WorkingThread.run(WorkingThread.java)
```

表示线程在方法调用时，额外的重要的操作。线程Dump分析的重要信息。修饰上方的方法调用。

调用修饰

- locked <地址> 目标
- waiting to lock <地址> 目标
- waiting on <地址> 目标
- parking to wait for <地址> 目标

实例锁： (a 类名)——synchronized对象。

类锁：(a Class for 类名)——静态synchronized方法。

调用修饰：locked

```
at oracle.jdbc.driver.PhysicalConnection.prepareStatement
- locked <0x00002aab63bf7f58> (a oracle.jdbc.driver.T4CConnection)
at oracle.jdbc.driver.PhysicalConnection.prepareStatement
- locked <0x00002aab63bf7f58> (a oracle.jdbc.driver.T4CConnection)
at com.jiuqi.dna.core.internal.db.datasource.PooledConnection.prepareStatement
```

```
synchronized (conn) { // conn的类型是T4CConnection
    // 同步块操作
}
```

通过synchronized关键字，成功获取到了对象的锁，成为监视器的拥有者，在临界区内操作。对象锁是可以线程重入的。

调用修饰：waiting to lock

```
at com.jiuqi.dna.core.impl.CacheHolder.isVisibleIn(CacheHolder.java:165)
- waiting to lock <0x0000000097ba9aa8> (a CacheHolder)
at com.jiuqi.dna.core.impl.CacheGroup$Index.findHolder
at com.jiuqi.dna.core.impl.ContextImpl.find
at com.jiuqi.dna.bap.basedata.common.util.BaseDataCenter.findInfo
```

```
synchronized (holder) { // holder的类型是CacheHolder
    // 临界区操作
}
```

通过synchronized关键字，没有获取到了对象的锁，线程在监视器的进入区等待。在调用栈顶出现，线程状态为Blocked。

调用修饰：waiting on

```
at java.lang.Object.wait(Native Method)
- waiting on <0x00000000da2defb0> (a WorkingThread)
at com.jiuqi.dna.core.impl.WorkingManager.getWorkToDo
- locked <0x00000000da2defb0> (a WorkingThread)
at com.jiuqi.dna.core.impl.WorkingThread.run
```

```
synchronized(thread) { // thread的类型是WorkingThread
    // 同步块操作……
    try {
        thread.wait();
    catch(InterruptException e) { /* 中断异常处理 */ }
}
```

通过synchronized关键字，成功获取到了对象的锁后，调用了wait方法，进去对象的等待区等待。在调用栈顶出现，线程状态为WAITING或TIMED_WATING。

synchronized模型相关的调用修饰

- `locked <对象地址> (a 类名)`

  使用synchronized申请对象锁成功，监视器的拥有者。

- `waiting to lock <对象地址> (a 类名)`

  使用synchronized申请对象锁未成功，在进入区等待。

- `waiting on <对象地址> (a 类名)`

  使用synchronized申请对象锁成功后，释放锁并在等待区等待。

调用修饰：parking to wait for

```
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <0x00000000eb8f35c8> (a FutureTask$Sync)
at java.util.concurrent.locks.LockSupport.park(LockSupport:156)
...
at java.util.concurrent.locks.ReentrantReadWriteLock$WriteLock.lock()
```

park是基本的线程阻塞原语，不通过监视器在对象上阻塞。

随concurrent包会出现的新的机制，与synchronized体系不同。

线程动作

```
"d&a-4066"  in Object.wait()
    TIMED_WAITING
    at java.lang.Object.wait(Native Method)
    - waiting on <0x00000007f7317268> (a WorkingThread)
    at com.jiuqi.dna.core.impl.WorkingManager.getWorkToDo
    - locked <0x00000007f7317268> (a WorkingThread)
    at com.jiuqi.dna.core.impl.WorkingThread.run
```

- 线程状态产生的原因
- 线程Dump分析中，非常重要的信息。

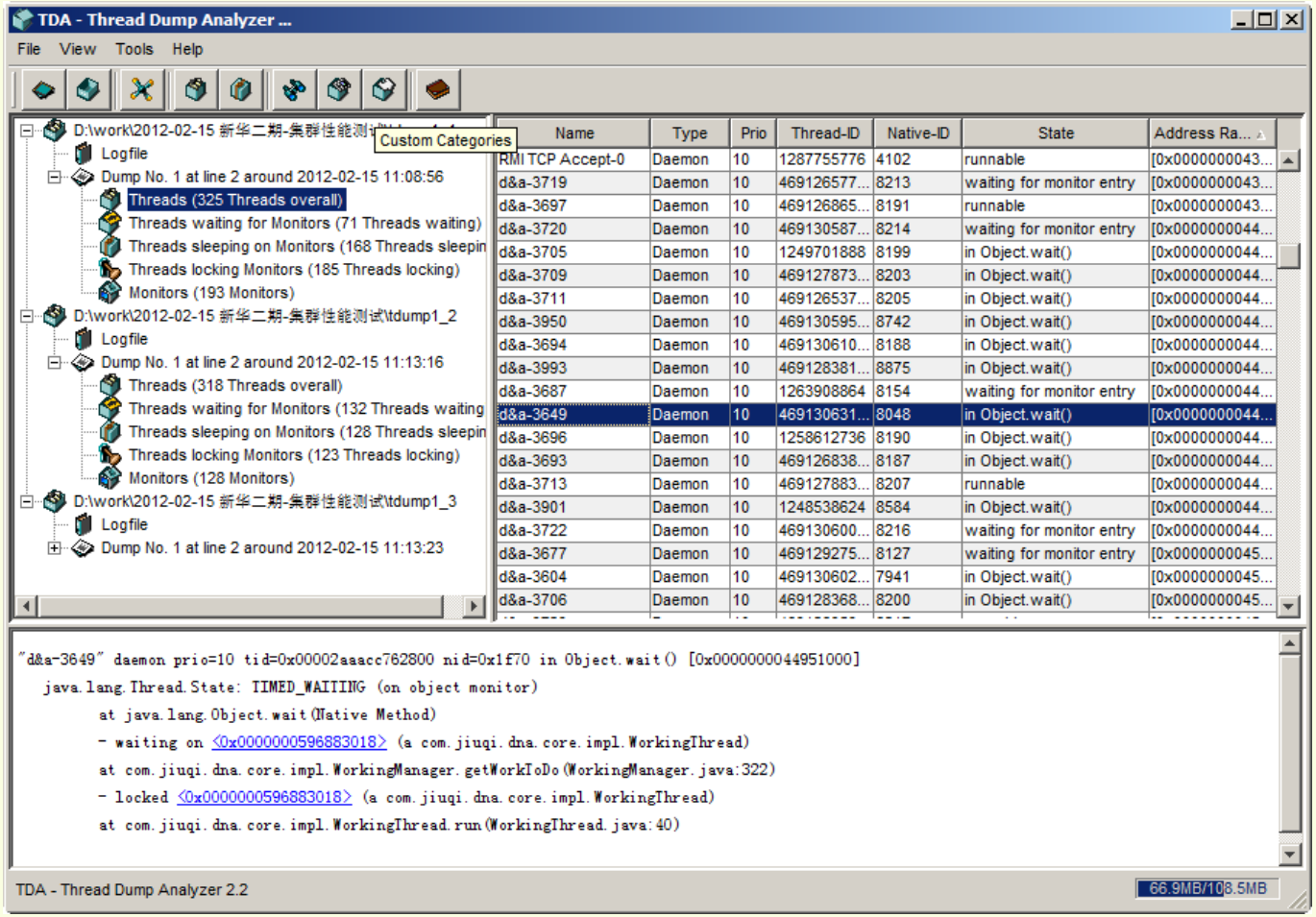
线程动作

- runnable：状态一般为RUNNABLE。
- in Object.wait()：等待区等待，状态为WAITING或TIMED_WAITING。
- waiting for monitor entry：进入区等待，状态为BLOCKED。
- waiting on condition：等待区等待、被park。
- sleeping：休眠的线程，调用了Thread.sleep()。

分析线程Dump的工具

- Thread Dump Analyzer (TDA)
- IBM Thread and Monitor Dump Analyzer (TMDA)

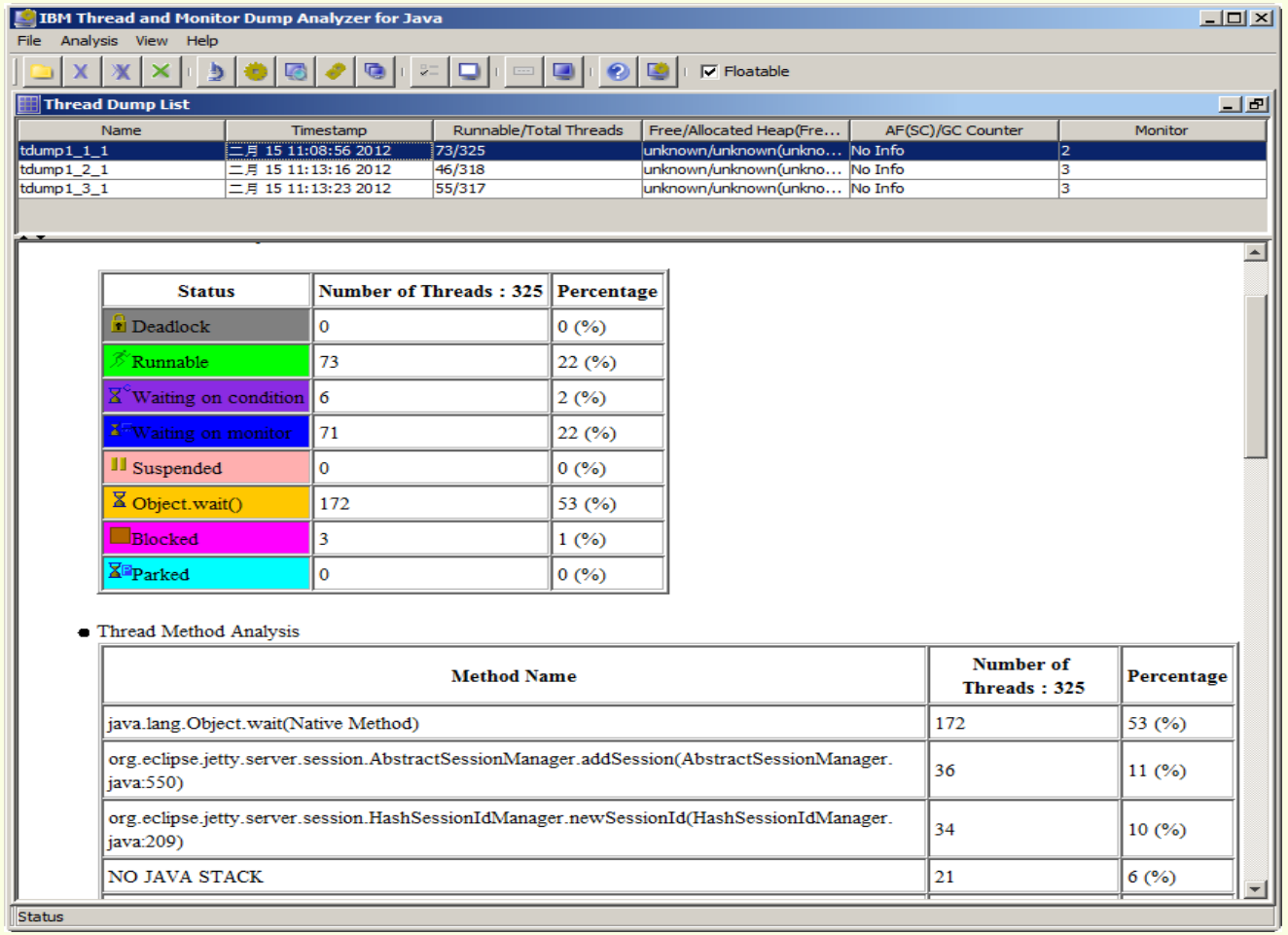
Thread Dump Analyzer(TDA)



Thread Dump Analyzer(TDA)

- 推荐首选使用。
- 可直接加载控制台输出，分析出多个线程Dump。
- 方便的查看各线程的运行状态，调用栈等信息。
- 查看异常的监视器及相关线程。
- 提供过滤器定位指定的线程。

IBM Thread and Monitor Dump Analyzer(TMDA)



IBM Thread and Monitor Dump Analyzer (TMDA)

- 能够分析虚拟机线程的统计信息。
- 能够方便查看监视器信息及相关依赖的线程。
- 能够比较多个线程Dump的线程信息，及监视器信息。

线程Dump的分析模式

- 分析的原则
- 可疑线程，三大入手点：
  - 进入区阻塞；
  - 持续运行的IO；
  - 非线程调度的休眠。
- 死锁问题分析的“三板斧”
- GC的干扰、Dump的比较

原则

- 结合代码阅读的推理。需要线程Dump和源码的相互推导和印证。
- 造成Bug的根源往往不会在调用栈上直接体现，一定格外注意线程当前调用之前的所有调用。

入手点之一：进入区等待

- 线程状态BLOCKED，线程动作wait on monitor entry，调用修饰waiting to lock总是一起出现。
- 表示在代码级别已经存在冲突的调用。必然有问题的代码，需要尽可能减少其发生。

```
"d&a-3588" daemon waiting for monitor entry [0x000000006e5d5000]
   java.lang.Thread.State: BLOCKED (on object monitor)
    at com.jiuqi.dna.bap.authority.service.UserService$LoginHandler.handle()
    - waiting to lock <0x0000000602f38e90> (a java.lang.Object)
    at com.jiuqi.dna.bap.authority.service.UserService$LoginHandler.handle()
```

典型的同步块阻塞

- 线程Dump示例：一个线程锁住某对象，大量其他线程在该对象上等待。

```
"blocker" runnable
java.lang.Thread.State: RUNNABLE
    at com.jiuqi.hcl.javadump.Blocker$1.run(Blocker.java:23)
    - locked <0x00000000eb8eff68> (a java.lang.Object)

"blockee-11" waiting for monitor entry
java.lang.Thread.State: BLOCKED (on object monitor)
    at com.jiuqi.hcl.javadump.Blocker$2.run(Blocker.java:41)
    - waiting to lock <0x00000000eb8eff68> (a java.lang.Object)

"blockee-86" waiting for monitor entry
java.lang.Thread.State: BLOCKED (on object monitor)
    at com.jiuqi.hcl.javadump.Blocker$2.run(Blocker.java:41)
    - waiting to lock <0x00000000eb8eff68> (a java.lang.Object)
```

典型的同步块阻塞

使用工具（TDA）查看：

入手点之二：持续运行的IO

- IO操作是可以以RUNNABLE状态达成阻塞。例如：数据库死锁、网络读写。
- 格外注意对IO线程的真实状态的分析。
- 一般来说，被捕捉到RUNNABLE的IO调用，都是有问题的。

典型的数据库死锁

```
"d&a-614" daemon prio=6 tid=0x0000000022f1f000 nid=0x37c8 runnable
[0x0000000027cbd000]
   java.lang.Thread.State: RUNNABLE
        at java.net.SocketInputStream.socketRead0(Native Method)
        at java.net.SocketInputStream.read(Unknown Source)
        at oracle.net.ns.Packet.receive(Packet.java:240)
        at oracle.net.ns.DataPacket.receive(DataPacket.java:92)
        at oracle.net.ns.NetInputStream.getNextPacket(NetInputStream.java:172)
        at oracle.net.ns.NetInputStream.read(NetInputStream.java:117)
        at oracle.jdbc.driver.T4CMAREngine.unmarshalUB1(T4CMAREngine.java:1034)
        at oracle.jdbc.driver.T4C8Oall.receive(T4C8Oall.java:588)
```

- 线程状态为RUNNABLE。
- 调用栈在SocketInputStream或SocketImpl上，socketRead0等方法。
- 调用栈包含了jdbc相关的包。

典型的数据库死锁

- 可能是正在执行；可能是数据库死锁。
- 制作多个线程Dump比较。
- 使用数据库提供的死锁分析手段：v$locked_object，db2pd。

入手点之三：非线程调度的等待区等待

- 主要针对DNA平台而言：动作为in Object.wait()的线程，正常应该在线程池中休眠。

入手点之三：分线程调度的休眠

- 正常的线程池等待

```
"d&a-131" in Object.wait()
  java.lang.Thread.State: TIMED_WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    at com.jiuqi.dna.core.impl.WorkingManager.getWorkToDo(WorkingManager.java:322)
    - locked <0x0000000313f656f8> (a com.jiuqi.dna.core.impl.WorkingThread)
    at com.jiuqi.dna.core.impl.WorkingThread.run(WorkingThread.java:40)
```

- 可疑的线程等待

```
"d&a-121" in Object.wait()
  java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Object.java:485)
    at com.jiuqi.dna.core.impl.AcquirableAccessor.exclusive()
    - locked <0x00000003011678d8> (a com.jiuqi.dna.core.impl.CacheGroup)
    at com.jiuqi.dna.core.impl.Transaction.lock()
```

入手点总结

动作	建议
wait on monitor entry	被阻塞的，肯定有问题
runnable	注意IO线程
in Object.wait()	注意非线程池等待

死锁问题分析的“三板斧”

- 在最可能死锁的时间点上制作Dump。
- 找出引起大量线程阻塞的线程。

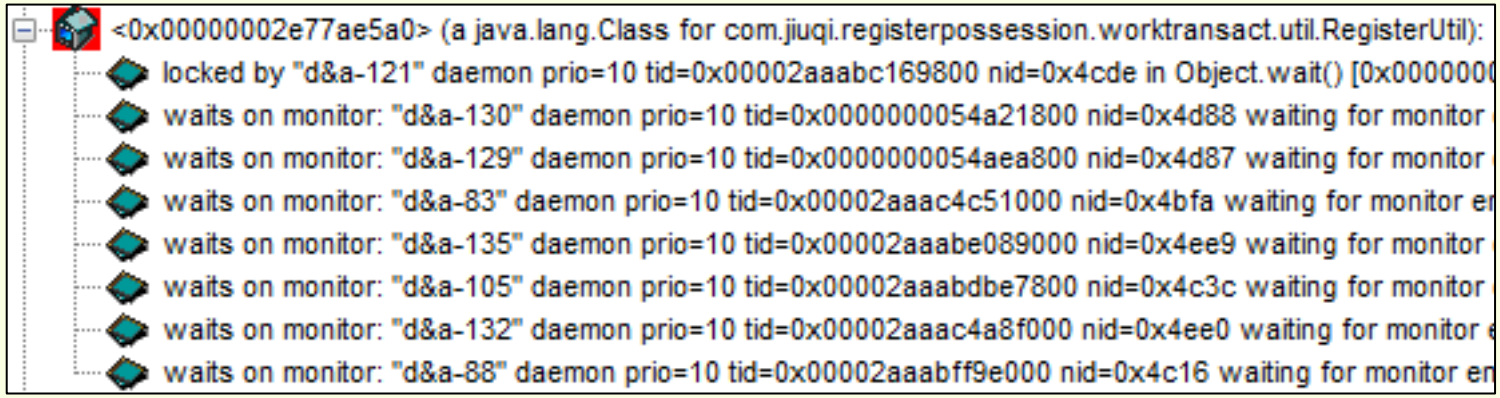
  线程X锁住了对象A，大量其他线程在对象A上阻塞。
- 分析导致该线程阻塞的原因。

  导致线程X阻塞的原因，数据库事务？内存事务？其他状态？
- 阅读代码，遍历其它等待或阻塞的线程的调用线索。

  其他线程在当前调用前，已经完成的所有的调用，是否可能造成线程X的阻塞？

案例一

- 应用系统界面按钮点击后无响应，制作线程Dump。
- 查看：发现存在对象锁的等待，如下图。



- 分析：d&a-121调用了某静态同步方法A，其他线程在类锁上等待。

案例一

- 分析：d&a-121的堆栈，如下：

```
"d&a-121" daemon in Object.wait()
  java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Object.java:485)
    at com.jiuqi.dna.core.impl.AcquirableAccessor.exclusive()
    - locked <0x00000003011678d8> (a com.jiuqi.dna.core.impl.CacheGroup)
    at com.jiuqi.dna.core.impl.Transaction.lock()
```

- 分析：d&a-121在等待其他线程提交缓存事务。
- 分析：如果其他阻塞或等待的线程，存在未提交的且冲突的缓存事务。
- 阅读代码，遍历其它的阻塞或等待的线程，发现类似的缓存操作。

案例一推演

- 线程X已经锁住对象A，大量其他线程在对象A上阻塞。
- 线程X在数据库IO上“阻塞”。
- 分析：其他等待或阻塞线程，若在当前调用前，存在冲突的未提交的数据库操作，则可能产生死锁。
- 任务：阅读代码，找出这样的线程。

案例二

- 登录界面点击登录按钮后无响应，制作线程Dump。
- 查看：发现大量同步块的阻塞，如下图：

案例二

- 查看：d&a-3433的调用栈。

```
java.lang.Thread.State: WAITING
......
at com.jiuqi.dna.core.impl.DataSourceImpl.alloc()
- locked <0x0000000600f3abf0> (a com.jiuqi.dna.core.impl.DataSourceImpl)
......
at com.jiuqi.dna.bap.authority.service.UserService$LoginHandler.handle()
- locked <0x0000000602f38e90> (a java.lang.Object)
```

- 分析：进入的同步方法LoginHandler，显然是一个登录功能的处理器，应此导致其他登录操作阻塞，界面无响应。
- 分析：线程在一个数据库连接池分配连接的方法上等待。

案例二

- 猜测：连接池可能已经达到最大连接分配数。
- 查看：线程Dump中的其他线程，发现大量线程——约200个——在数据库IO上阻塞，调用栈包括：

```
com.jiuqi.dna.bap.log.service.LogInfoService$AddLogInfo
Handler.handle
```

- 分析：什么阻塞了日志的写入？

一些技巧

- 大胆猜测，小心证明。
- 大量的代码阅读。
- 注意调用栈与众不同的线程。

GC对线程状态的干扰

虚拟机执行Full GC时，会阻塞所有的用户线程。因此，即时获取到同步锁的线程也有可能被阻塞。

```
"d&a-635" waiting for monitor entry
  java.lang.Thread.State: BLOCKED (on object monitor)
    at com.jiuqi.dna.core.impl.CacheHolder.isVisibleIn(CacheHolder.java:165)
    - locked <0x0000000097ba9aa8> (a com.jiuqi.dna.core.impl.CacheHolder)
    at com.jiuqi.dna.core.impl.CacheGroup$Index.internalFindHolder(CacheGroup.java:1977)
    at com.jiuqi.dna.core.impl.CacheGroup$Index.findHolder(CacheGroup.java:1838)
    at com.jiuqi.dna.core.impl.ContextImpl.internalFind(ContextImpl.java:1339)
```

GC对线程状态的干扰

- 在查看线程Dump时，首先查看内存使用情况。

```
Heap
 PSYoungGen      total 241600K, used 125092K [0x00000007f0000000, 0x0
  eden space 221696K, 55% used [0x00000007f0000000,0x00000007f77e1300
  from space 19904K, 11% used [0x00000007fd880000,0x00000007fdac8048,
  to   space 19328K, 0% used [0x00000007fed20000,0x00000007fed20000,0
 PSOldGen        total 11739776K, used 10380329K [0x0000000500000000,
  object space 11739776K, 88% used [0x0000000500000000,0x000000077990
 PSPermGen       total 167936K, used 167600K [0x00000004f0000000, 0x0
  object space 167936K, 99% used [0x00000004f0000000,0x00000004fa3ac2
```

- 使用虚拟机参数“-verbose:gc”在控制台输出GC信息。

```
195.016: [GC 1780056K->1644630K(1795904K), 0.0211303 secs]
195.338: [GC 1781846K->1648995K(1795712K), 0.0220126 secs]
195.361: [Full GC 1648995K->1584815K(2386496K), 3.1349677 secs]
198.635: [GC 1721903K->1591316K(2330624K), 0.0055168 secs]
198.914: [GC 1728404K->1594348K(2369856K), 0.0082131 secs]
199.340: [GC 1707372K->1599398K(2314624K), 0.0094291 secs]
```

可以比较多个线程Dump

- 可以制作多份线程Dump，使用TMDA的比较功能，分析系统的连续状态。

d&a-3484	java.lang.Obje...	com.jiuqi.dna.c...	java.lang.Obje...
d&a-3528	java.lang.Obje...	java.lang.Obje...	java.lang.Obje...
d&a-3538	org.eclipse.jett...	com.jiuqi.dna.c...	java.lang.Obje...
d&a-3604	java.lang.Obje...	java.lang.Obje...	java.lang.Obje...
d&a-3605	java.lang.Obje...	com.jiuqi.dna.c...	java.lang.Obje...
d&a-3641	org.eclipse.jett...	com.jiuqi.dna.u...	java.lang.Class...
d&a-3649	java.lang.Obje...	java.lang.Obje...	java.lang.Obje...
d&a-3677	org.eclipse.jett...	sun.misc.Unsaf...	java.util.Hash...
d&a-3681	org.eclipse.jett...	sun.misc.Unsaf...	java.lang.Obje...
d&a-3682	java.lang.Obje...		
d&a-3685	java.lang.Obje...		
d&a-3686	org.eclipse.jett...	com.jiuqi.dna.c...	java.lang.Obje...
d&a-3687	org.eclipse.jett...		
d&a-3688	java.lang.Obje...	com.jiuqi.dna.c...	java.lang.Obje...
d&a-3689 => d&a-...	java.lang.Obje...	java.lang.Obje...	java.lang.Obje...

分析堆DUMP

SINCE 1997

分析堆Dump

1. 堆Dump的内容
2. 使用场景
3. 相关的Java机制
4. 常用工具
5. 一个案例

堆Dump的内容

- 系统信息、虚拟机属性。
- 完整的线程Dump。
- 所有类和对象的状态。

堆Dump分析的使用场景

- 在出现内存不足、GC异常时，怀疑代码存在内存泄漏。制作堆Dump，找出生命周期错误关联的对象以及相关代码。
- 某些复杂场景下，需要查看对象的状态。

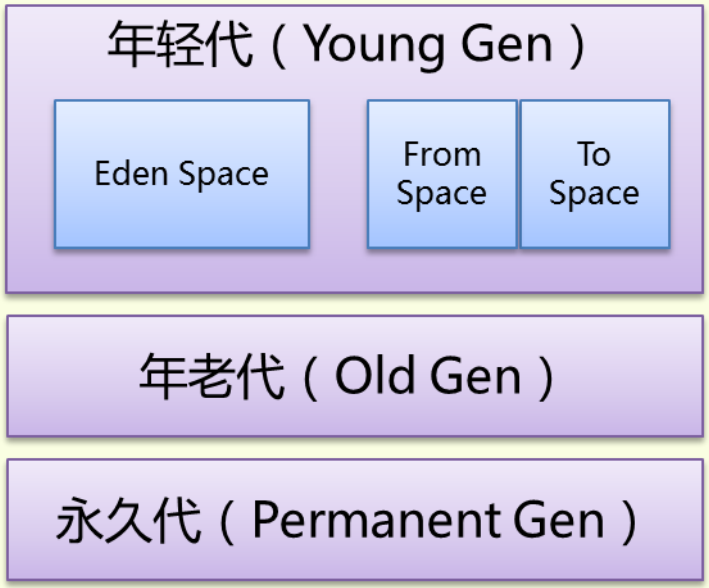
堆Dump相关的Java机制

- Java虚拟机的内存模型
- 垃圾回收（GC）的介绍
- 垃圾回收根、对象的保留大小
- 内存不足的错误

Java虚拟机的内存模型

- 三块：年轻代、年老代、永久代。
- 新创建的对象放在年轻代。
- 年轻代垃圾回收后，残余的对象会被移动到年老代。
- 永久代存放类信息等，一般不会被回收。

Java虚拟机的内存模型



典型分配方案：

- 永久代：默认64M，大型应用相应增加到256M。
- 年轻代1/4。
- 剩余为年老代。

垃圾回收（Garbage Collection）

- YoungGen GC，新生代回收，Minor GC

  针对年轻代的回收。毫秒级别。

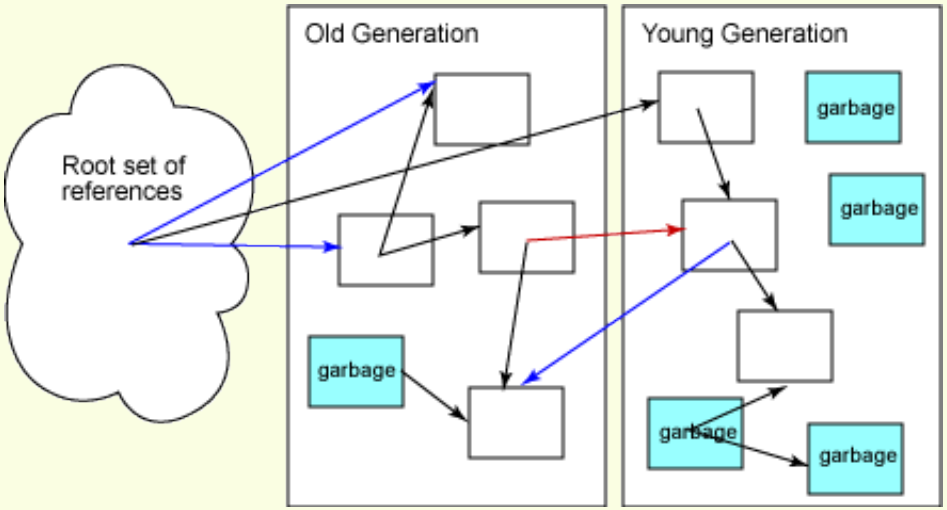
- Full GC，全回收，Major GC

  针对全虚拟机内存——包括年轻代、年老代、永久代——的回收。

  秒级别。会暂停虚拟机的线程。

垃圾回收根 Garbage Collection Roots

- 虚拟机如何决定哪些对象需要被回收？
- 从垃圾回收根开始，不再被引用到的对象即为垃圾对象。
- GC Root一般为线程、会话等对象。



对象的保留大小

- 从对象上，能递归引用到的对象的合计大小。
- 非准确数，循环引用等造成。

内存不足的错误

- OutOfMemoryError

  年老代内存不足。

- OutOfMemoryError：PermGen Space

  永久带内存不足。

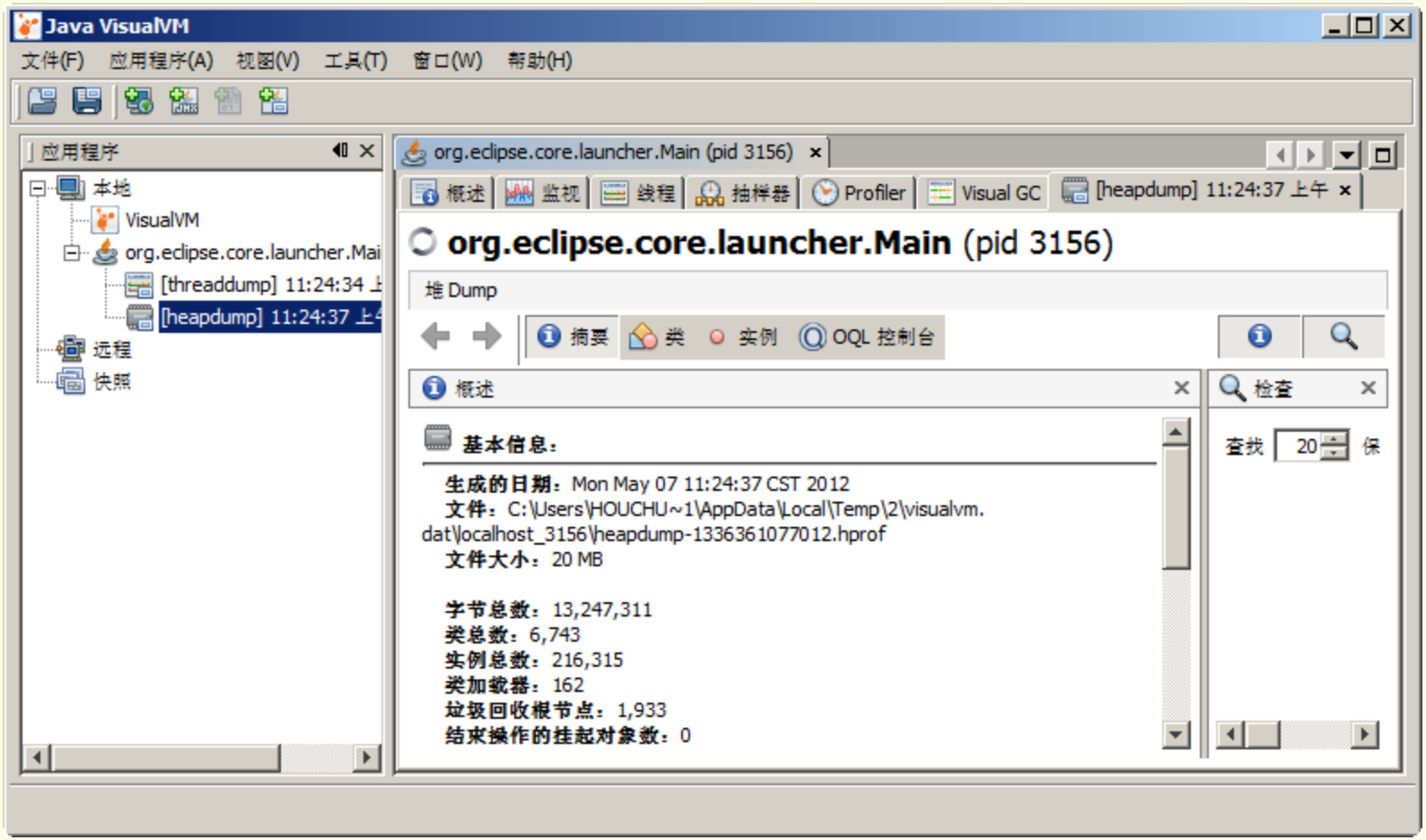
- OutOfMemoryError：GC overhead limit exceed

  垃圾回收时间占用系统运行时间的98%或以上。

常用工具

- Java VisualVM：JDK自带。
- Memory Analyzer（MAT）：Eclipse基金会开发
- IBM HeapAnalyzer

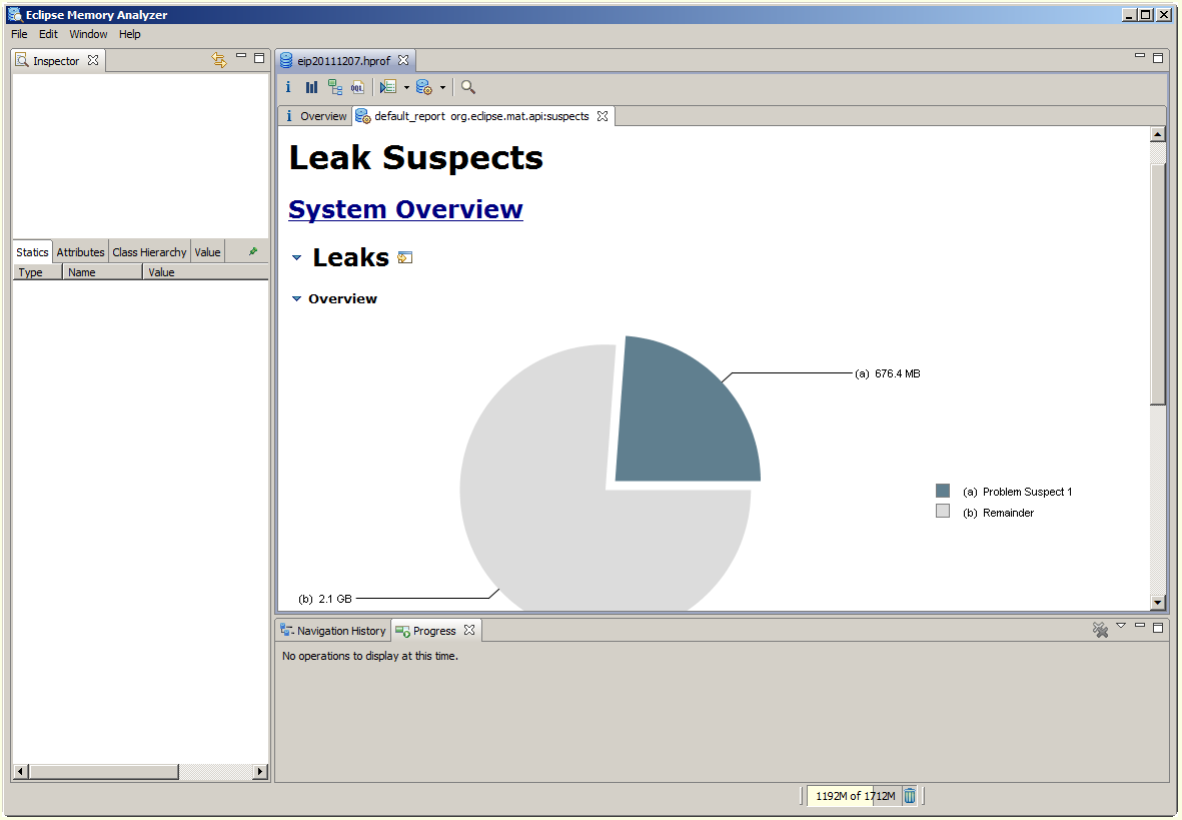
Java VisualVM



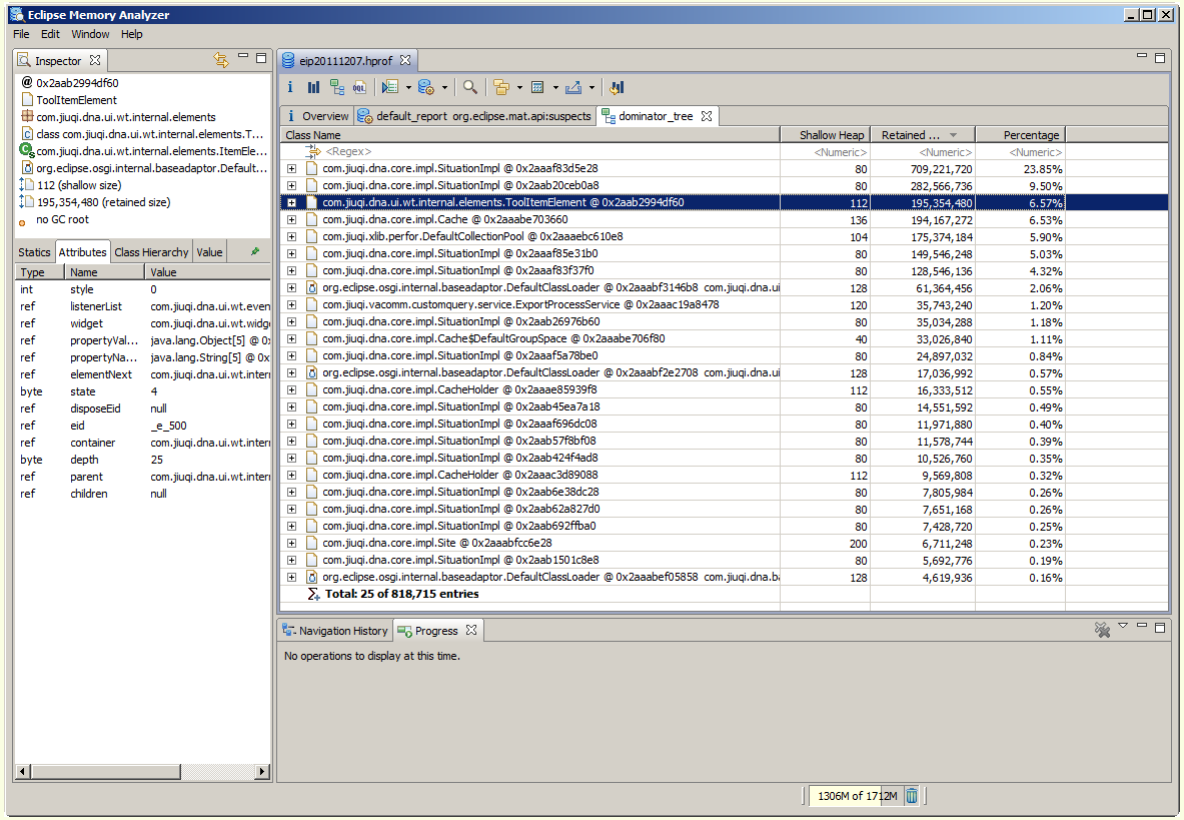
Java VisualVM

- 机器配置太好，并且只是简单查看Dump时使用。
- 上手简单，使用方便，没有太多的概念。
- 可以查看系统信息、虚拟机参数、类、对象状态。
- 支持OQL查询语言，及插件扩展：Visual GC、BTrace Workbench。
- 对物理内存要求极高；机器配置较低时，无法加载较大的堆Dump；对象保留大小的计算效率低。

Memory Analyzer（MAT）



Memory Analyzer（MAT）



Memory Analyzer（MAT）

- 内存溢出分析时建议使用。
- 对硬件配置要求不算太高。
- 分析效率较为优秀，提供分析建议。
- 使用额外的文件存储分析结果。

工具比较

	JVVM	MAT
上手难度	低	较高
硬件配置要求	高	中等
分析效率	低	中等
分析建议	不支持	较好

堆Dump分析的硬件需求

- Java VisualVM，建议物理内存大于Dump文件的大小。
- MAT性能明显优于前者，并且提供分析建议。

比较堆Dump？

- 堆Dump中只有对象的内存地址，不存在对象ID号。工具中对象ID号只是一种优化的显示。而对象的内存地址会随着GC而改变。
- 不能依赖于工具提供的对象ID。

堆Dump中的线程Dump

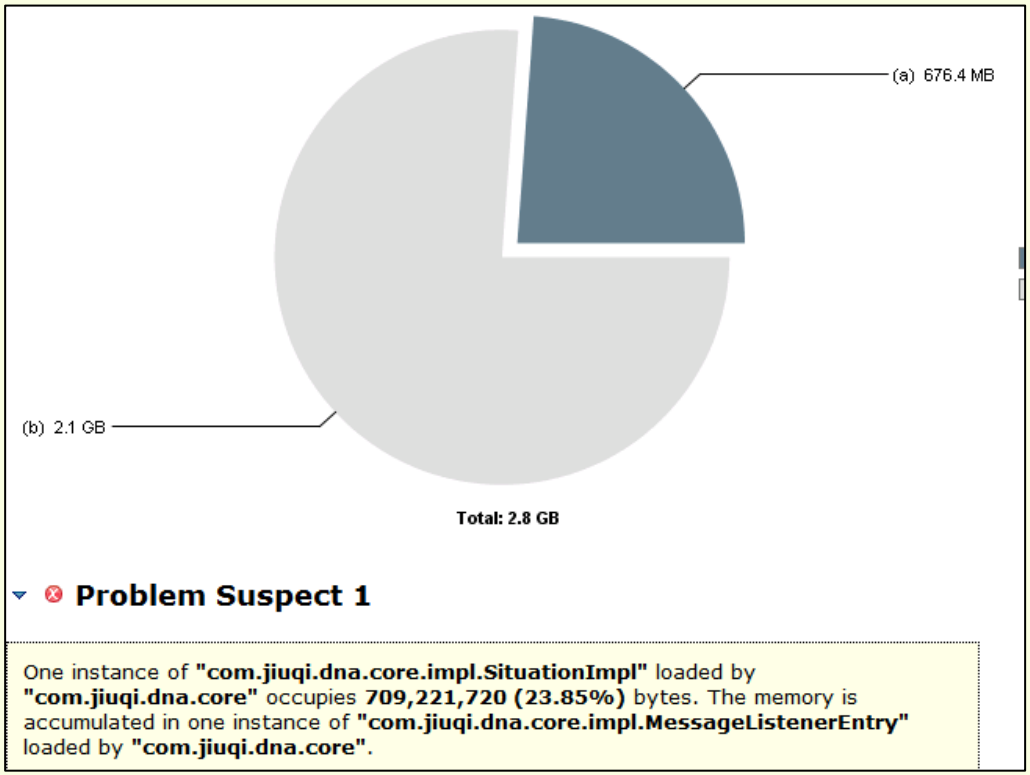
- 虽然堆Dump中包含了线程Dump，但两者的状态并不一定完全一致。
- 堆Dump的分析一般不会分析其中的线程Dump信息。

一个内存泄漏的案例

- 现状：EIP响应极慢，控制台出现GC Overhead Limit Exceeded的错误，于是制作堆Dump。

一个内存泄漏的案例

- 使用MAT分析堆Dump，生成的Leak Supspects如下：



一个内存泄漏的案例

- 使用DominatorTree查看怀疑泄漏的对象及引用。

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
com.jiuqi.dna.core.impl.SituationImpl @ 0x2aaaf83d5e28	80	709,221,720	23.85%
com.jiuqi.dna.core.impl.MessageListenerEntry @ 0x2aaaf83e5458	80	709,221,640	23.85%
com.jiuqi.dna.core.impl.MessageListenerEntry @ 0x2aaaf83e54c0	80	709,221,536	23.85%
com.jiuqi.dna.core.impl.MessageListenerEntry @ 0x2aaaf89e99b8	80	709,221,432	23.85%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	4,215,320	0.14%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	4,190,616	0.14%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	4,188,336	0.14%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	4,186,768	0.14%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	4,185,944	0.14%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	4,185,904	0.14%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	4,172,208	0.14%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	3,993,032	0.13%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	3,991,736	0.13%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	3,990,968	0.13%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	3,989,344	0.13%
com.jiuqi.dna.bap.billmgr.common.controls.UIBillMgrBillWindow @ 0x2a	144	3,988,904	0.13%

一个内存泄漏的案例

- 结合源代码，分析发现在Situation上注册了监听器，而监听器引用了一个大小4M的页面对象。
- 页面打开时注册监听器，在关闭页面时没有反注册。而Situation的生命周期相对较长。
- 累计操作后，造成较为严重的内存泄漏。

内存泄漏分析的总结

- 使用MAT分析堆Dump，查看LeakSuspect及DominatorTree。
- 阅读代码，确定对象引用的错误关联而导致的生命周期错误。

总结

SINCE 1997

总结

- JavaDump：Java进程的快照。
- JavaDump分析：针对多线程并发、内存泄漏。
- 制作JavaDump。
- 分析线程Dump：“三板斧”。
- 分析堆Dump：使用MAT。

相关资源一

随后邮件发布：

- 本PPT；相应的Word文档，可作为手册查看。
- VMWare虚拟机环境，练习命令行环境下的Dump制作。
- 实例代码，生成各种线程场景。
- 相关案例的线程Dump。
- 研究院联系人：芦星

相关资源二

Oracle官方网站。

MAT的官方Wiki和Blog。

BTrace：一个Java平台上非常强大的调试框架。

谢谢