Kris Mok, Software Engineer, Taobao
@rednaxelafx
莫枢 /“撒迦”

Java Crash分析

现象与对策

阿里巴巴集团-技术共享平台-
核心系统研发-专用计算组
莫枢（撒迦）
v0.1 2012-04-24
v0.2 2012-05-10

关于我...

- 2009年毕业自南京大学软件学院
- 同年10月加入淘宝
- 目前在淘宝核心系统部参与JVM相关研发
- 编程语言的设计与实现爱好者

- 希望与各位同好多交流！☺
  - 博客：http://rednaxelafx.iteye.com/
  - 新浪微博：http://weibo.com/rednaxelafx
  - 高级语言虚拟机群组：http://hllvm.group.iteye.com/
  - JVM源码阅读活动微群：http://q.weibo.com/1823766

JVM crash了！

真的是JVM crash了么…？

讨论范围

- 表现为真的JVM crash的问题
  - 实际原因可能是：
    - JVM内部实现的bug
    - native library的bug
    - Java应用层代码自身的bug
    - native memory耗光了
    - 外部环境的影响
- 遇到crash时的一些简单可操作的处理办法
  - 篇幅所限，无法深入介绍；不能解决所有crash

非讨论范围

- Java进程被人为kill掉
  - 这是首先要检查的情况
  - 在多用户共用的开发环境里特别容易出问题
    - 开发A要部署新代码，重启了服务器
    - 开发B正在用服务器做测试，发现进程突然消失了
    - 请协调好资源的使用

非讨论范围

- Java应用失去响应
  - 但并未crash，进程还在
  - 有可能是
    - Java代码陷入死循环
      - 可参考例子 https://gist.github.com/1081908
    - 连续的GC
      - 可观察GC日志或`jstat -gccause`
    - 日志把磁盘打满了
      - 可用 `df` 观察磁盘剩余空间
    - *…等等*
  - 这次不详细展开讨论

非讨论范围

- Java层面的异常
  - Java级别的OutOfMemoryError
  - ClassNotFoundException
  - NoClassDefFoundError
  - StackOverflowError
  - NullPointerException
  - *…其它异常*
    - 可捕获（已捕获），并且进程还存在

JVM Crash定义

- Java进程意外消失
- 通常伴随着
  - crash log：hs_err_pid<*pid*>.log（重要）
  - core dump: core.<*pid*>
  - 或系统日志
    - 主要是 /var/log/messages 里的异常信息

crash了

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00002add2265d648, pid=2188, tid=1078282560
#
# JRE version: 6.0_30-b12
# Java VM: OpenJDK (Taobao) 64-Bit Server VM (20.0-b12-internal-fastdebug
mixed mode linux-amd64 compressed oops)
# Problematic frame:
# V  [libjvm.so+0xd05648]  Unsafe_SetNativeInt+0xb8
#
# An error report file with more information is saved as:
# /home/sajia/hs_err_pid2188.log
#
# If you would like to submit a bug report, please visit:
#   http://java.sun.com/webapps/bugreport/crash.jsp
#
Current thread is 1078282560
Dumping core ...
Aborted (core dumped)
```

有打出core dump，
位置通常在“当前工作目录”

crash了但没core dump

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00002b4fb5a7a648, pid=20190, tid=1084868928
#
# JRE version: 6.0_30-b12
# Java VM: OpenJDK (Taobao) 64-Bit Server VM (20.0-b12-internal-fastdebug
mixed mode linux-amd64 compressed oops)
# Problematic frame:
# V  [libjvm.so+0xd05648]  Unsafe_SetNativeInt+0xb8
#
# An error report file with more information is saved as:
# /home/sajia/hs_err_pid20190.log
#
# If you would like to submit a bug r
#   http://java.sun.com/webapps/bugr
#
Current thread is 1084868928
Dumping core ...
Aborted
```

没打出core dump。
如需要core dump，请确认：
1、ulimit -c unlimited
2、该Java进程对“当前工作目录”有写权限
3、磁盘剩余空间充足

被OOM Killer干掉了


```
$ sudo cat /var/log/messages | grep -i "killed process"
Aug 30 15:51:54 testmachine kernel: : Out of memory:
Killed process 15605 (java).
```

这种情况下既不会有crash log也不会有core dump

事前准备

Preparation

事前准备

- 若要让JVM在检测到crash时执行外部命令
  - 可使用 -XX:OnError='...'
  - 其中可使用 %p 来指定pid
  - 例如：
    - -XX:OnError='pstack %p > ~/pstack%p.log'
  - 这样指定的命令在执行时要从Java进程fork出去，如果内存已经严重不足则无法成功fork
    - 如发现指定的命令根本没执行，请尝试用后面提到的-XX:+ShowMessageBoxOnError让Java进程在退出前先暂停，然后手动从外部执行需要的命令

事前准备

- 若要让JVM在检测到crash后不立即退出，而是提示打开调试器
  - 可使用 `-XX:+ShowMessageBoxOnError`
  - 主要用于crash时的现场调试
  - 通常不需要启用
  - 遇到很诡异的bug时可以靠该参数获取更直接的现场信息

事前准备

- 为便于事后分析，最好能留下core dump
  - 注意 `ulimit -c unlimited`
  - 尽量在程序的当前工作目录留出足够磁盘空间
  - 或配置core dump文件的路径到合适的位置
    - `kernel.core_uses_pid`
    - `kernel.core_pattern`
    - 可在`/etc/sysctl.conf`文件中配置
    - 请参考文档，例子1，例子2

保护现场

Keep the Crash Site Intact

保护现场

- 前面提到的crash相关日志、core dump都尽量保持在原位置，或至少在事发机器上
  - 如果磁盘空间不是特别紧张，别把它们删了
- 如果有core dump，最好保持程序运行时所使用的可执行文件在原路径上，如java、libjvm.so、libc.so等
  - 为便于事后分析，千万别急于在卸载掉相关程序（如JDK），或在原路径覆盖安装新版本

急救措施

First Aid

1. 观察crash log

- 最简单的观察方式：
  - 看文件开头的错误概述
    - OutOfMemoryError？SIGSEGV？EXCEPTION_ACCESS_VIOLATION？
    - 如果是访问异常，异常的地址是？是不是空指针？
  - 看出错的线程名及其调用栈信息
    - 是JVM内部的线程还是应用的Java线程？
    - 执行到什么地方出的错？
  - 根据上面两种信息总结出关键字
  - 用Google搜一下bugs.sun.com或其它地方有没有类似的crash报告，有的话找出其说明
  - 很可能是已知的VM bug或第三方库的bug

1. 观察crash log

- 例：https://gist.github.com/2584513#file_java_error_28522.short.log

1. 观察crash log

```
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00002aaf0d11c031, pid=28522, tid=1104951616
```

错误类型：
SIGSEGV

1. 观察crash log

```
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00002aaf0d11c031, pid=28522, tid=1104951616
```

出错时的程序指针的值：
（该线程执行到哪里了？）
0x00002aaf0d11c031

1. 观察crash log

```
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00002aaf0d11c031, pid=28522, tid=1104951616
```

出错的进程的ID：
28522

1. 观察crash log

```
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00002aaf0d11c031, pid=28522, tid=1104951616
```

这个可以忽略

1. 观察crash log

```
---------------  T H R E A D  ---------------

Current thread (0x0000000052109000):  JavaThread "CompilerThread1"
daemon [_thread_in_native, id=28538,
stack(0x0000000041cc3000,0x0000000041dc4000)]
```

出错的线程在HotSpot VM里的C++层Thread对象的地址：0x0000000052109000

1. 观察crash log

```
---------------  T H R E A D  ---------------

Current thread (0x0000000052109000):  JavaThread "CompilerThread1"
daemon [_thread_in_native, id=28538,
stack(0x0000000041cc3000,0x0000000041dc4000
```

出错的线程的类型：
JavaThread

1. 观察crash log

```
---------------  T H R E A D  ---------------

Current thread (0x0000000052109000):  JavaThread "CompilerThread1"
daemon [_thread_in_native, id=28538,
stack(0x0000000041cc3000,0x0000000041dc4000)]
```

出错的线程的名字（如果有）：
CompilerThread1

1. 观察crash log

```
---------------  T H R E A D  ---------------

Current thread (0x0000000052109000):  JavaThread "CompilerThread1"
daemon [_thread_in_native, id=28538,
stack(0x0000000041cc0000,0x0000000041dc4000)]
```

出错的线程的状态：
_thread_in_native

1. 观察crash log

```
---------------  T H R E A D  ---------------

Current thread (0x0000000052109000):  JavaThread "CompilerThread1"
daemon [_thread_in_native, id=28538,
stack(0x0000000041cc3000,0x0000000041dc4000)]
```

出错的线程的ID：
28538

1. 观察crash log

```
---------------  T H R E A D  ---------------

Current thread (0x0000000052109000):  JavaThread "CompilerThread1"
daemon
[_thread_in_native, id=28538, stack(0x0000000041cc3000,0x0000000041d
c4000)]
```

出错的线程的调用栈
在内存中的地址范围：
(0x0000000041cc3000,
0x0000000041dc4000)

1. 观察crash log

```
siginfo:si_signo=SIGSEGV:
si_errno=0, si_code=1
(SEGV_MAPERR), si_addr=0x0000000000000008
```

错误类型：
SIGSEGV

1. 观察crash log

```
siginfo:si_signo=SIGSEGV: si_errno=0,
si_code=1 (SEGV_MAPERR),
si_addr=0x0000000000000008
```

错误子类型：
SEGV_MAPERR

1. 观察crash log

```
siginfo:si_signo=SIGSEGV: si_errno=0,
si_code=1 (SEGV_MAPERR),
si_addr=0x0000000000000008
```

错误地址（重要）：
0x0000000000000008
这是对空指针解引用的表现

1. 观察crash log

```
Instructions: (pc=0x00002aaf0d11c031)
0x00002aaf0d11c021:   8b 97 e0 09 00 00 44 89 c7 48 8b 0c fa 49 89 d0
0x00002aaf0d11c031:   4c 8b 49 08 49 83 39 00 75 2b 8b 73 08 66 90 8b
```

crash地点附近的代码：
（十六进制表示的机器码）
pc表示当前指令指针的值

淘宝JDK6u30开始直接在crash log
里显示这部分代码的反汇编

1. 观察crash log

```
Stack: [0x0000000041cc3000,0x0000000041dc4000],  sp=0x0000000041dbd480,  free
space=1001k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [libjvm.so+0x593031]
V  [libjvm.so+0x592cf2]
V  [libjvm.so+0x590001]
V  [libjvm.so+0x2d5bc2]
V  [libjvm.so+0x2d2cbc]
V  [libjvm.so+0x25b0de]
V  [libjvm.so+0x2dc468]
V  [libjvm.so+0x2dbd6e]
V  [libjvm.so+0x72d159]
V  [libjvm.so+0x7268c1]
V  [libjvm.so+0x623e1f]
```

native级别的调用栈记录
老JDK6上libjvm.so里的符号不会显示，
JDK6u25或更高版本则会显示

1. 观察crash log

```
// First letter indicates type of the frame:
//    J: Java frame (compiled)
//    j: Java frame (interpreted)
//    V: VM frame (C/C++)
//    v: Other frames running VM generated code (e.g.
stubs, adapters, etc.)
//    C: C/C++ frame
```

1. 观察crash log

```
Stack: [0x0000000041cc3000,0x0000000041dc4000],  sp=0x0000000041dbd480,  free
space=1001k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [libjvm.so+0x593031]
V  [libjvm.so+0x592cf2]
V  [libjvm.so+0x590001]
V  [libjvm.so+0x2d5bc2]
V  [libjvm.so+0x2d2cbc]
V  [libjvm.so+0x25b0de]
V  [libjvm.so+0x2dc468]
V  [libjvm.so+0x2dbd6e]
V  [libjvm.so+0x72d159]
V  [libjvm.so+0x7268c1]
V  [libjvm.so+0x623e1f]
```

有些crash log里，
后面还会有Java级别的栈帧

1. 观察crash log

```
Current CompileTask:
C2:2554  !
com.taobao.rate.ratesearch.RateSearchAdaptor.adaptRateDoTo
SchemaDO(Lcom/taobao/ratecenter/domain/dataobject/FeedRate
DO;ZLjava/util/Set;)Lcom/taobao/rate/ratesearch/dataobject
/RateSearchSchemaDO; (504 bytes)
```

（编译器线程）
当前正在编译的Java方法的签名

1. 观察crash log

```
Current CompileTask:
C2:2554  !
com.taobao.rate.ratesearch.RateSearchAdaptor.adaptRateDoTo
SchemaDO(Lcom/taobao/ratecenter/domain/dataobject/FeedRate
DO;ZLjava/util/Set;)Lcom/taobao/rate/ratesearch/dataobject
/RateSearchSchemaDO; (504 bytes)
```

编译器类型：
C1是Client Compiler
C2是Server Compiler

1. 观察crash log

```
Current CompileTask:
C2:2554  !
com.taobao.rate.ratesearch.RateSearchAdaptor.adaptRateDoTo
SchemaDO(Lcom/taobao/ratecenter/domain/dataobject/FeedRate
DO;ZLjava/util/Set;)Lcom/taobao/rate/ratesearch/dataobject
/RateSearchSchemaDO; (504 bytes)
```

编译任务的编号：
第2554个编译任务

1. 观察crash log

```
Current CompileTask:
C2:2554  !
com.taobao.rate.ratesearch.RateSearchAdaptor.adaptRateDoTo
SchemaDO(Lc  /taobao/ratecenter/domain/dataobject/FeedRate
DO;ZLjava/ut  Set;)Lcom/taobao/rate/ratesearch/dataobject
/RateSearchSc  O; (504 bytes)
```

该被编译的方法里有异常处理代码

1. 观察crash log

```
Current CompileTask:
C2:2554  !
com.taobao.rate.ratesearch.RateSearchAdaptor.adaptRateDoTo
SchemaDO(Lcom/taobao/ratecenter/domain/dataobject/FeedRate
DO;ZLjava/util/Set;)Lcom/taobao/rate/ratesearch/dataobject
/RateSearchSchemaDO; (504 bytes)
```

该被编译的方法的字节码大小：
504字节

1. 观察crash log

```
---------------  P R O C E S S  ---------------

Java Threads: ( => current thread )
  ...
  0x000000005210b800 JavaThread "Low Memory Detector" daemon [_thread_blocked, id=28539, stack(0x0000000041dc4000,0x0000000041ec5000)]
=>0x0000000052109000 JavaThread "CompilerThread1" daemon [_thread_in_native, id=28538, stack(0x0000000041cc3000,0x0000000041dc4000)]
  0x0000000052104000 JavaThread "CompilerThread0" daemon [_thread_blocked, id=28537, stack(0x0000000041bc2000,0x0000000041cc3000)]
  0x00[illegible]0000052102000 JavaThread "Signal Dispatcher" daemon [_thread_blocked, id=28536, stack(0x0000000040bec000,0x0000000040ced000)]
```

线程列表中=>表明当前线程

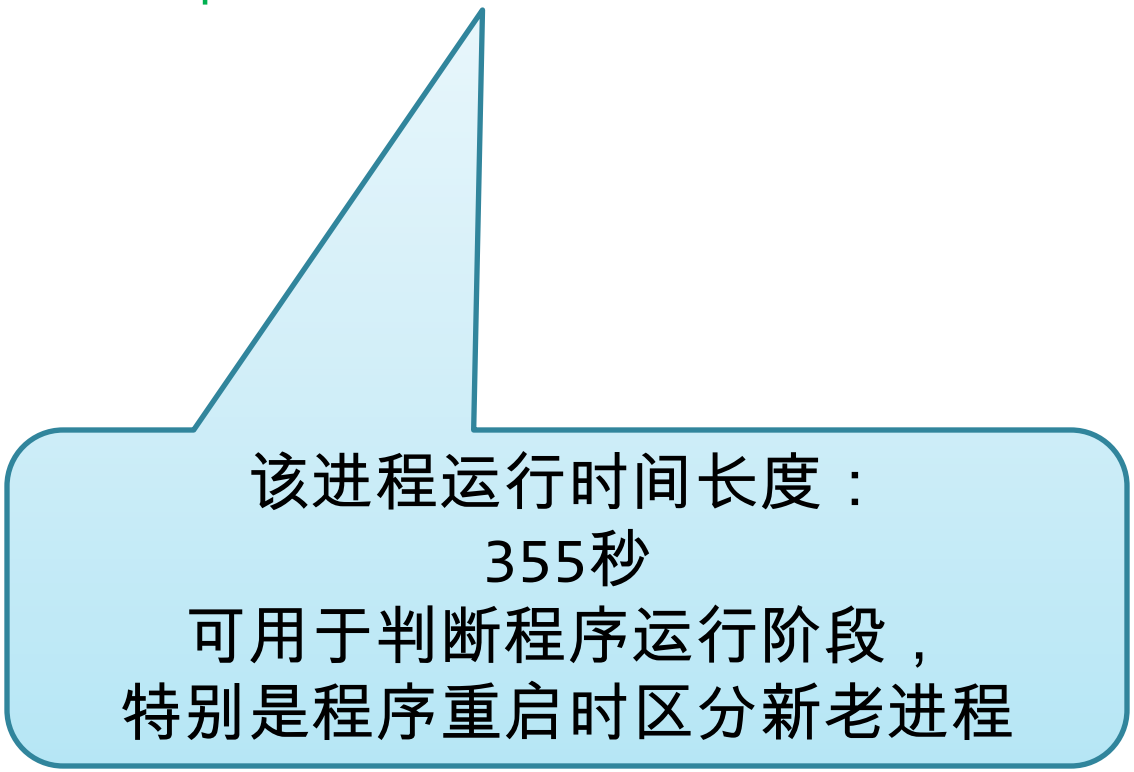
1. 观察crash log

```
time: Mon Apr  9 21:44:28 2012
elapsed time: 355 seconds
```

打出crash log的时刻：
2012-04-09 21:44:28

1. 观察crash log

```
time: Mon Apr  9 21:44:28 2012
elapsed time: 355 seconds
```



1. 观察crash log

- （续上例）
- 只看crash log如何知道crash地点附近的代码是怎样的？
- 用udis库里带的udcli工具来反汇编
  - `echo <code> | udcli -intel -x -[32|64] -o <addr>`
- 使用例：
  https://gist.github.com/2584513#file_udcli_example.log

- （如有core dump则直接用gdb的disassemble命令更方便）

1. 观察crash log

- （续上例）
- 看不到调用栈里libjvm.so的函数的符号信息怎么办？
  - `objdump -C -d --start-address=<addr> libjvm.so | egrep '>:$' -m 1`
- 一个简单的shell脚本来获取crash log里libjvm.so的函数的符号信息
  - https://gist.github.com/2584416
- 把符号信息放回到crash log里
  - https://gist.github.com/2584513#file_java_error_28522.short.extra.log

1. 观察crash log

- （续上例）
- 用关键字搜索
  - SIGSEGV PhaseIdealLoop::build_loop_late_post
- 搜到若干个bug ID，其中包括7068051
  - 但仔细观察“Release Fixed”与“Related Bugs”可知该bug从JDK7/HS21才开始存在，也只在JDK7与JDK8发布了修正。而本案例在JDK6u23上。
- 实际上更可能是7070134
  - 可惜该bug详情不公开
  - 从邮件列表的讨论可以一窥端倪
  - 从JDK的release note可知它在JDK6u29已修复

1. 观察crash log

- （续上例）
- 如在老版本JDK上遇到crash，且能稳定重现
- 然后能找到已知的JDK bug，并且确认官方已在新版JDK提供修复
- 那...赶紧找个测试环境升级到新JDK再跑跑回归测试吧☺

- 广告：该案例在淘宝JDK6u32上测试运行无问题

1. 观察crash log

- （续上例）
- 如果碰到编译器问题但没找到已知的JDK bug怎么办？
- 可针对部分Java方法禁用JIT编译
  - `-XX:CompileCommand='exclude,class/name,methodName'`
  - 参考例子

1. 观察crash log

- 如果有已知bug，但暂时无法升级JDK
  - 请参考bug report中给出的workaround

Work Around	Use this flag as workaround: -XX:-UseLoopPredicate

2. 用jstack/gdb看栈的状况

- 为定位问题原因，可先查看线程栈的状况
- jstack与gdb皆可从core dump获取栈信息
- （如果有core dump但无crash log，看栈的状况是第一着手点）

2. 用jstack/gdb看栈的状况

- `jstack <java_executable_path> <core_dump_path>`
- `gdb <java_executable_path> <core_dump_path>`

2. 用jstack/gdb看栈的状况

- jstack可看到Java与native栈帧的符号信息
- gdb可看到更准确的native栈帧的符号信息
  - gdb的backtrace（简写bt）命令查看当前线程栈
  - gdb的info threads命令查看线程列表
  - gdb的thread <n>切换线程

2. 用jstack/gdb看栈的状况

- 例：https://gist.github.com/2574440
- 可见jstack的输出中有一个线程在深度递归，且该线程最顶上的方法是native方法
- 用gdb打开core dump确定出问题的线程正是在上面看到的在深度递归的线程
- 此可推断该crash虽然表现为native crash，但根本问题在Java代码中（递归条件不完善导致无限递归）

2. 用jstack/gdb看栈的状况

- （续上例）
- 如果既没crash log又没core dump怎么办？
- `$ sudo cat /var/log/messages | grep -i java`
- `May  2 14:08:47 testmachine kernel: : java[2089]: segfault at 0000000041a2efa8 rip 00002aaab81e2507 rsp 0000000041a2f018 error 6`

- 通过上面的系统日志也可以看到是爆栈了
  - 出错地址与栈顶指针（rsp）很接近
- 还是需要core dump来定位根本问题
  - `ulimit -c unlimited`

3. 连续在不固定位置遇到native OutOfMemoryError

- 老版本JDK在JVM需要分配native memory无法得到满足时
  - 报告java.lang.OutOfMemoryError
  - 并打出crash log，然后退出
- 例：
  https://gist.github.com/2575866#file_command_prompt

3. 连续在不固定位置遇到native OutOfMemoryError


```
# A fatal error has been detected by the Java Runtime Environment:
#
# java.lang.OutOfMemoryError: requested 4092 bytes for char in
/BUILD_AREA/jdk6_24/hotspot/src/share/vm/utilities/stack.inline.hpp. Out of
swap space?
#
#  Internal Error (allocation.inline.hpp:39), pid=15553, tid=1325374352
#  Error: char in
/BUILD_AREA/jdk6_24/hotspot/src/share/vm/utilities/stack.inline.hpp
```

3. 连续在不固定位置遇到native OutOfMemoryError

- 新版本JDK（JDK6u25或更高版本）的提示信息更为友善
- 例：
  https://gist.github.com/2575866#file_command_prompt2

3. 连续在不固定位置遇到native OutOfMemoryError


```
# There is insufficient memory for the Java Runtime Environment to continue.
# Native memory allocation (malloc) failed to allocate 4092 bytes for char
in /BUILD_AREA/jdk6_32/hotspot/src/share/vm/utilities/stack.inline.hpp
# Possible reasons:
#   The system is out of physical RAM or swap space
#   In 32 bit mode, the process size limit was hit
# Possible solutions:
#   Reduce memory load on the system
#   Increase physical memory or swap space
#   Check if swap backing store is full
#   Use 64 bit Java on a 64 bit OS
#   Decrease Java heap size (-Xmx/-Xms)
#   Decrease number of Java threads
#   Decrease Java thread stack sizes (-Xss)
#   Set larger code cache with -XX:ReservedCodeCacheSize=
# This output file may be truncated or incomplete.
#
#  Out of Memory Error (allocation.inline.hpp:44), pid=16388,
tid=1325312912
```

3. 连续在不固定位置遇到native OutOfMemoryError

- 如果native memory吃紧，则JVM可能在一些需要多用native memory的地方遇到问题
  - 动态编译过程中需要使用临时内存
  - GC过程中需要使用GC堆之外额外的临时内存
- 写得不好的native库也可能出问题
  - 不检查malloc()结果是否为NULL就直接使用的
- 实际表现为不固定位置的native OutOfMemoryError

3. 连续在不固定位置遇到native OutOfMemoryError

- 例：https://gist.github.com/2575866
- 演示场景：
  - 将native memory用到接近极限，然后触发GC，让GC因无法分配到临时内存而报native OOME

3. 连续在不固定位置遇到native OutOfMemoryError

- （续上例）
- 实验方法：
  - 在64位Linux上运行32位JDK
  - 分别使用JDK6u24与JDK6u32进行两次实验
  - 使用较大的GC堆
    - `-Xmx2g -XX:MaxPermSize=512m`
  - 创建82个、每个占用16MB native memory的DirectByteBuffer，保证它们存活
  - 此时32位寻址空间已经快被耗尽
  - 然后触发GC；JVM尝试为GC分配辅助数据结构的内存空间时失败，报native OOME

3. 连续在不固定位置遇到native OutOfMemoryError

用户态寻址空间的默认大小

	32位程序	64位程序
32位Linux	3GB	
64位Linux	4GB	很大…
32位Windows	2GB	

无论“物理内存”有多大，
进程在用户态可寻址的空间就这么大，
如果native memory的需求超过了寻址空间的限制就会分配不出内存

3. 连续在不固定位置遇到native OutOfMemoryError

用户态寻址空间的默认大小

	32位程序	64位程序
32位Linux	3GB	
64位Linux	4GB	很大…
32位Windows	2GB	

如果对native memory需求大，
但受到了32位寻址空间的限制，
请升级到64位系统+64位JDK

3. 连续在不固定位置遇到native OutOfMemoryError

- （续上例）
- 可借助ps或top等工具来判断是否遇到了地址空间的瓶颈
- 参考例子中的做法
  - https://gist.github.com/2575866#file_command_prompt3
  - 先靠-XX:+ShowMessageBoxOnError让JVM在临退出前先暂停下来
  - 然后执行`ps u -p <pid>`，观察结果中的VSZ（默认单位是KB）
  - 例中两次实验VSZ都已非常接近4GB，也就是32位进程在64位Linux上用户态地址空间的极限

3. 连续在不固定位置遇到native OutOfMemoryError

- （续上例）
- 如在64位JDK上遇到native OOME
- 通常不是寻址空间不足，而是整个虚拟内存的资源不足了
  - 物理内存+交换空间

3. 连续在不固定位置遇到native OutOfMemoryError

- 已知native memory使用过多，如何排查？
- 从简单的情况开始
  - 先检查是不是NIO direct memory造成的问题
  - 然后看是否遇到别的已知容易导致native memory泄漏的情况
  - 还不行...试试挂上perftools或valgrind之类
- 请参考毕玄的经验帖

3. 连续在不固定位置遇到native OutOfMemoryError

- NIO direct memory
  - 一部分在GC堆内
    - DirectByteBuffer自身是普通Java对象
    - 大小固定
    - 32位HotSpot VM上：56字节
    - 64位HotSpot VM上（开压缩指针）：64字节
    - 64位HotSpot VM上（不开压缩指针）：80字节
  - 一部分在GC堆外的native memory中
    - malloc()得到，大小由用户指定（外加一个内存页）
    - DirectByteBuffer.address字段记录着地址

3. 连续在不固定位置遇到native OutOfMemoryError


```
hsdb> inspect 0xe366e730
instance of Oop for java/nio/DirectByteBuffer @ 0xe366e730 (size = 56)
_mark: 5
_metadata._klass: InstanceKlass for java/nio/DirectByteBuffer @
0xba1a9a40
mark: -1
position: 0
limit: 128
capacity: 128
address: 3092664320
hb: null
offset: 0
isReadOnly: false
bigEndian: true
nativeByteOrder: false
fd: null
att: null
cleaner: Oop for sun/misc/Cleaner @ 0xe366e7a0
```

32位HotSpot VM

3. 连续在不固定位置遇到native OutOfMemoryError


```
hsdb> inspect 0x00000000ee9a34f8
instance of Oop for java/nio/DirectByteBuffer @ 0x00000000ee9a34f8
(size = 64)
_mark: 5
_metadata._compressed_klass: InstanceKlass for
java/nio/DirectByteBuffer @ 0x00000000c443ee38
mark: -1
position: 0
limit: 128
capacity: 128
address: 46912657712128
hb: null
offset: 0
isReadOnly: false
bigEndian: true
nativeByteOrder: false
fd: null
att: null
cleaner: Oop for sun/misc/Cleaner @ 0x00000000ee9a3600
```

64位HotSpot VM
（开压缩指针）

3. 连续在不固定位置遇到native OutOfMemoryError


```
hsdb> inspect 0x00002aaad7f14650
instance of Oop for java/nio/DirectByteBuffer @ 0x00002aaad7f14650
(size = 80)
_mark: 1
_metadata._klass: InstanceKlass for java/nio/DirectByteBuffer @
0x00002aaaae6388a8
mark: -1
position: 0
limit: 128
capacity: 128
address: 46913730187408
hb: null
offset: 0
isReadOnly: false
bigEndian: true
nativeByteOrder: false
fd: null
att: null
cleaner: Oop for sun/misc/Cleaner @ 0x00002aaad7f147d0
```

64位HotSpot VM
（不开压缩指针）

3. 连续在不固定位置遇到native OutOfMemoryError

- NIO direct memory释放native memory
  - 通过sun.misc.Cleaner
    - 是PhantomReference
    - 释放DirectByteBuffer关联的native memory
  - 需要借助GC来触发Cleaner的清理动作
  - 详细介绍请参考这帖：
    - http://hllvm.group.iteye.com/group/topic/27945

3. 连续在不固定位置遇到native OutOfMemoryError

- NIO direct memory释放native memory
  - 如果GC不频繁，则已死的DirectByteBuffer所关联的native memory可能得不到及时释放
  - 从而导致“内存泄漏”的表象
  - 可能进而引发JVM crash

3. 连续在不固定位置遇到native OutOfMemoryError

- 判断是否有已死DirectByteBuffer堆积？
- 对运行中的Java进程使用工具：
  - `jmap -histo <pid> | grep DirectByteBuffer$Deallocator`
    - 可获得通过Java的ByteBuffer.allocateDirect()创建的DirectByteBuffer对象的个数
  - DirectMemorySize
    - 可获得当前NIO direct memory所申请的native memory的大小（reserved size）
    - 可选的-e -v参数可提供更详细信息
      - 但加这俩参数会大幅降低该工具的运行速度，请谨慎使用

3. 连续在不固定位置遇到native OutOfMemoryError

- 操作步骤
  - jmap -histo <pid> | grep DirectByteBuffer$Deallocator
  - java -cp .:$JAVA_HOME/lib/sa-jdi.jar DirectMemorySize <pid>
  - jmap -histo:live <pid> 2>&1 1>/dev/null
  - jmap -histo:live <pid> | grep DirectByteBuffer$Deallocator
  - java -cp .:$JAVA_HOME/lib/sa-jdi.jar DirectMemorySize <pid>

3. 连续在不固定位置遇到native OutOfMemoryError

- 操作步骤
  - jmap -histo <pid> | grep DirectByteBuffer$Deallocator
  - java -cp .:$JAVA_HOME/lib/sa-jdi.jar DirectMemorySize <pid>
  - jmap -histo:live <pid> 2>
  - jmap -histo:live <pid> | DirectByteBuffer$Deallocat
  - java -cp .:$JAVA_HOME/lib DirectMemorySize <pid>

先看当前在GC堆内的DirectByteBuffer对象的个数（不论死活）

3. 连续在不固定位置遇到native OutOfMemoryError

- 操作步骤
  - jmap -histo <pid> | grep DirectByteBuffer$Deallocator
  - java -cp .:$JAVA_HOME/lib/sa-jdi.jar DirectMemorySize <pid>
  - jmap -histo:live <pid> 2>
  - jmap -histo:live <pid> |
    DirectByteBuffer$Deallocat
  - java -cp .:$JAVA_HOME/lib
    DirectMemorySize <pid>

看当前NIO direct memory所占用的native memory大小（地址空间意义上）

3. 连续在不固定位置遇到native OutOfMemoryError

- 操作步骤
  - jmap -histo <pid> | grep DirectByteBuffer$Deallocat
  - java -cp .:$JAVA_HOME/lib DirectMemorySize <pid>
  - jmap -histo:live <pid> 2>&1 1>/dev/null
  - jmap -histo:live <pid> | grep DirectByteBuffer$Deallocator
  - java -cp .:$JAVA_HOME/lib/sa-jdi.jar DirectMemorySize <pid>

从外部强行触发一次full GC
（不需关心它的输出）

3. 连续在不固定位置遇到native OutOfMemoryError

- 操作步骤
  - jmap -histo <pid> | grep DirectByteBuffer$Deallocat
  - java -cp .:$JAVA_HOME/lib DirectMemorySize <pid>
  - jmap -histo:live <pid> 2>&1>/dev/null
  - jmap -histo:live <pid> | grep DirectByteBuffer$Deallocator
  - java -cp .:$JAVA_HOME/lib/sa-jdi.jar DirectMemorySize <pid>

再触发一次full GC，并观察GC后还存活的DirectByteBuffer对象的个数

3. 连续在不固定位置遇到native OutOfMemoryError

- 操作步骤
  - jmap -histo <pid> | grep DirectByteBuffer$Deallocat
  - java -cp .:$JAVA_HOME/lib DirectMemorySize <pid>
  - jmap -histo:live <pid> 2>&1 > /dev/null
  - jmap -histo:live <pid> | grep DirectByteBuffer$Deallocator
  - java -cp .:$JAVA_HOME/lib/sa-jdi.jar DirectMemorySize <pid>

最后观察两次full GC后NIO direct memory所占用的native memory的大小

3. 连续在不固定位置遇到native OutOfMemoryError

- 经过上述操作
  - 如发现一开始DirectByteBuffer对象个数较多，且NIO direct memory占用的内存也是大头
  - 且在两次full GC后，这两个数值都有显著降低
  - 那么可以断定native memory使用量大与NIO direct memory占用的内存未及时清理相关

3. 连续在不固定位置遇到native OutOfMemoryError

- 对策：
  - 检查是否使用了-XX:+DisableExplicitGC
    - 如有，去掉它
    - 如在使用CMS GC，可改为使用：
    - -XX:+ExplicitGCInvokesConcurrent

3. 连续在不固定位置遇到native OutOfMemoryError

- 对策：
  - 检查是否使用-XX:MaxDirectMemorySize
    - 如没有，请将其设置到一个合理的值
    - -Xmx + -XX:MaxPermSize +
    - -XX:ReservedCodeCacheSize +
    - (-Xss * 线程数) +
    - -XX:MaxDirectMemorySize
    - 的总和应保持在物理内存的大小以下
    - 并应留出缓冲的余度
  - 详细仍请参考前面提到的参数经验帖

4. 进一步分析

- 需要更多的底层知识
  - 这里不展开讲
  - 需要对HotSpot VM和JDK有深入的了解
  - 能够熟练使用gdb调试C或C++写的程序
    - 或其它native debugger
  - 能够借助HotSpot VM自带的CLHSDB工具来辅助调试

4. 进一步分析

- for 阿里系的同事
  - 遇到crash请先自救
  - 在前面介绍的方法都无法分析出crash的原因时
    - 确认已保存好现场
    - 欢迎向**核心系统研发-专用计算组**咨询☺

QUESTIONS?

Kris Mok, Software Engineer, Taobao
@rednaxelafx
莫枢 /“撒迦”

以下是未使用内容

hs_err_pid<*pid*>.log文件的解析

CRASH LOG的结构

instructions段

```
Instructions: (pc=0x00002b4fb5a7a648)
0x00002b4fb5a7a628:   48 8b 07 ff 50 20 66 90 48 8b 43 48 48 8b 40 08
0x00002b4fb5a7a638:   83 40 30 01 e8 bf 2f c4 ff c6 80 9c 02 00 00 01
0x00002b4fb5a7a648:   45 89 2e c6 80 9c 02 00 00 00 48 8b 5b 48 48 8b
0x00002b4fb5a7a658:   7b 10 4c 8b 63 08 48 83 7f 08 00 74 09 e8 66 4d
```

instructions段

```
Instructions: (pc=0x00002b4fb5a7a648)
0x00002b4fb5a7a628:   48 8b 07 ff 50 20 66 90 48 8b 43 48 48 8b 40 08
0x00002b4fb5a7a638:   83 40 30 01 e8 bf 2f c4 ff c6 80 9c 02 00 00 01
0x00002b4fb5a7a648:   45 89 2e c6 80 9c 02 00 00 00 48 8b 5b 48 48 8b
0x00002b4fb5a7a658:   7b 10 4c 8b 63 08 48 83 7f 08 00 74 09 e8 66 4d

[Disassembling for mach='i386:x86-64']
  0x00002b4fb5a7a628: mov    (%rdi),%rax
  0x00002b4fb5a7a62b: callq  *0x20(%rax)
  0x00002b4fb5a7a62e: xchg   %ax,%ax
  0x00002b4fb5a7a630: mov    0x48(%rbx),%rax
  0x00002b4fb5a7a634: mov    0x8(%rax),%rax
  0x00002b4fb5a7a638: addl   $0x1,0x30(%rax)
  0x00002b4fb5a7a63c: callq  0x00002b4fb56bd600
  0x00002b4fb5a7a641: movb   $0x1,0x29c(%rax)
  0x00002b4fb5a7a648: mov    %r13d,(%r14)
  0x00002b4fb5a7a64b: movb   $0x0,0x29c(%rax)
  0x00002b4fb5a7a652: mov    0x48(%rbx),%rbx
  0x00002b4fb5a7a656: mov    0x10(%rbx),%rdi
  0x00002b4fb5a7a65a: mov    0x8(%rbx),%r12
  0x00002b4fb5a7a65e: cmpq   $0x0,0x8(%rdi)
  0x00002b4fb5a7a663: je     0x00002b4fb5a7a66e
  0x00002b4fb5a7a665: .byte 0xe8
  0x00002b4fb5a7a666: data16
  0x00002b4fb5a7a667: rex.WRB
```

淘宝JDK6u30开始带有的附加信息：汇编指令

栈帧类型

```
// First letter indicates type of the frame:
//    J: Java frame (compiled)
//    j: Java frame (interpreted)
//    V: VM frame (C/C++)
//    v: Other frames running VM generated code (e.g. stubs, adapters, etc.)
//    C: C/C++ frame
```

TODO

- SIGSEGV
- SIGBUS
- SIGILL
- EXCEPTION_ACCESS_VIOLATION
- Internal Error

CLHSDB TODO

CLHSDB