http://rednaxelafx.iteye.com/blog/1847971

借HSDB来探索HotSpot VM的运行时数据

(未经许可请勿转载。希望转载请与我联系。)

(如果打开此页面时浏览器有点卡住的话请耐心等待片刻。大概是ItEye的代码高亮太耗时了...)

几天前在HLLVM群组有人问了个小问题, 说

```
1.  public class Test {
2.      static Test2 t1 = new Test2();
3.          Test2 t2 = new Test2();
4.      public void fn() {
5.          Test2 t3 = new Test2();
6.      }
7.  }
8.
9.  class Test2 {
10.
11. }
```

这个程序的t1、t2、t3三个变量本身(而不是这三个变量所指向的对象)到底在哪里。

TL;DR版回答是：

- t1在存Java静态变量的地方, 概念上在JVM的方法区(method area)里
- t2在Java堆里, 作为Test的一个实例的字段存在
- t3在Java线程的调用栈里, 作为Test.fn()的一个局部变量存在

不过就这么简单的回答大家都会, 满足不了对JVM的实现感兴趣的同学们的好奇心。说到底, 这"方法区"到底是啥? Java堆在哪里?Java线程的调用栈又是啥样的?

那就让我们跑点例子, 借助调试器来看看在一个实际运行中的JVM里是啥状况。

(下文中代码也传了一份到https://gist.github.com/rednaxelafx/5392451)

==========

写个启动类来跑上面问题中的代码：

```
1.  public class Main {
2.      public static void main(String[] args) {
3.          Test test = new Test();
4.          test.fn();
5.      }
6.  }
```

(编译这个Main.java和上面的Test.java时最好加上-g参数生成LocalVariableTable等调试信息, 以便后面某些情况下可以用到)

接下来如无特别说明本文将使用Windows 7 64-bit, Oracle JDK 1.7.0_09 Server VM, Serial GC的环境中运行所有例子。

之前在GreenTeaJUG在杭州的活动演示Serviceability Agent的时候也讲到过这是个非常便于探索HotSpot VM内部实现的API, 而HSDB则是在SA基础上包装起来的一个调试器。这次我们就用HSDB来做实验。

SA的一个限制是它只实现了调试snapshot的功能：要么要让被调试的目标进程完全暂停, 要么就调试core dump。所以我们在用HSDB做实验前, 得先让我们的Java程序运行到我们关注的点上才行。

理想情况下我们会希望让这Java程序停在Test.java的第6行, 也就是Test.fn()中t3局部变量已经进入作用域, 而该方法又尚未返回的地方。怎样才能停在这里呢?

其实用个Java层的调试器即可。大家平时可能习惯了在Eclipse、IntelliJ IDEA、NetBeans等Java IDE里使用Java层调试器, 但为了减少对外部工具的依赖, 本文将使用Oracle JDK自带的jdb工具来完成此任务。

jdb跟上面列举的IDE里包含的调试器底下依赖着同一套调试API, 也就是Java Platform Debugger Architecture (JPDA)。功能也类似, 只是界面是命令行的, 表明上看起来不太一样而已。

为了方便后续步骤, 启动jdb的时候可以设定让目标Java程序使用serial GC和10MB的Java heap。
启动jdb之后可以用stop in命令在指定的Java方法入口处设置断点, 然后用run命令指定主类名称来启动Java程序, 等跑到断点看看位置是否已经到满足需求, 还没到的话可以用step、next之类的命令来向前进。
对jdb命令不熟悉的同学可以在启动jdb之后使用help命令来查看命令列表和说明。

具体步骤如下：

Command prompt代码

```
D:\test>jdb -XX:+UseSerialGC -Xmx10m
Initializing jdb ...
> stop in Test.fn
Deferring breakpoint Test.fn.
It will be set after the class is loaded.
> run Main
run  Main
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint Test.fn

Breakpoint hit: "thread=main", Test.fn(), line=5 bci=0
5          Test2 t3 = new Test2();

main[1] next

Step completed: > "thread=main", Test.fn(), line=6 bci=8
6       }

main[1]
```

按照上述步骤执行完最后一个next命令之后, 我们就来到了最初想要的Test.java的第6行, 也就是Test.fn()返回前的位置。

接下来把这个jdb窗口放一边, 另开一个命令行窗口用jps命令看看我们要调试的Java进程的pid是多少：

Command prompt代码

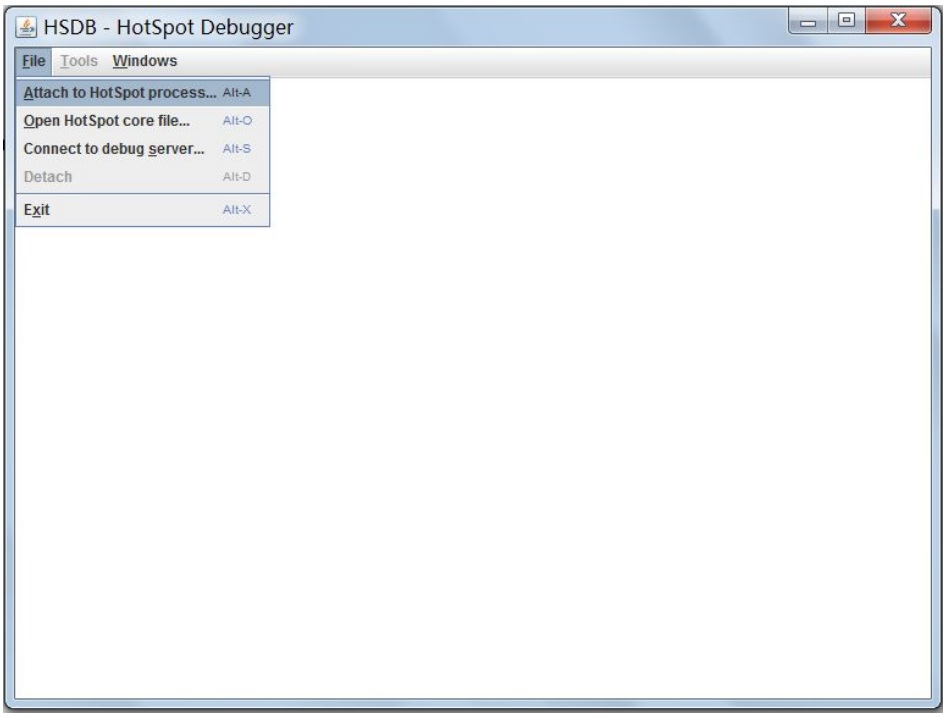
```
D:\test>jps
4328 Main
9064 Jps
7716 TTY
```

可以看到是4328。把这个pid记下来待会儿用。

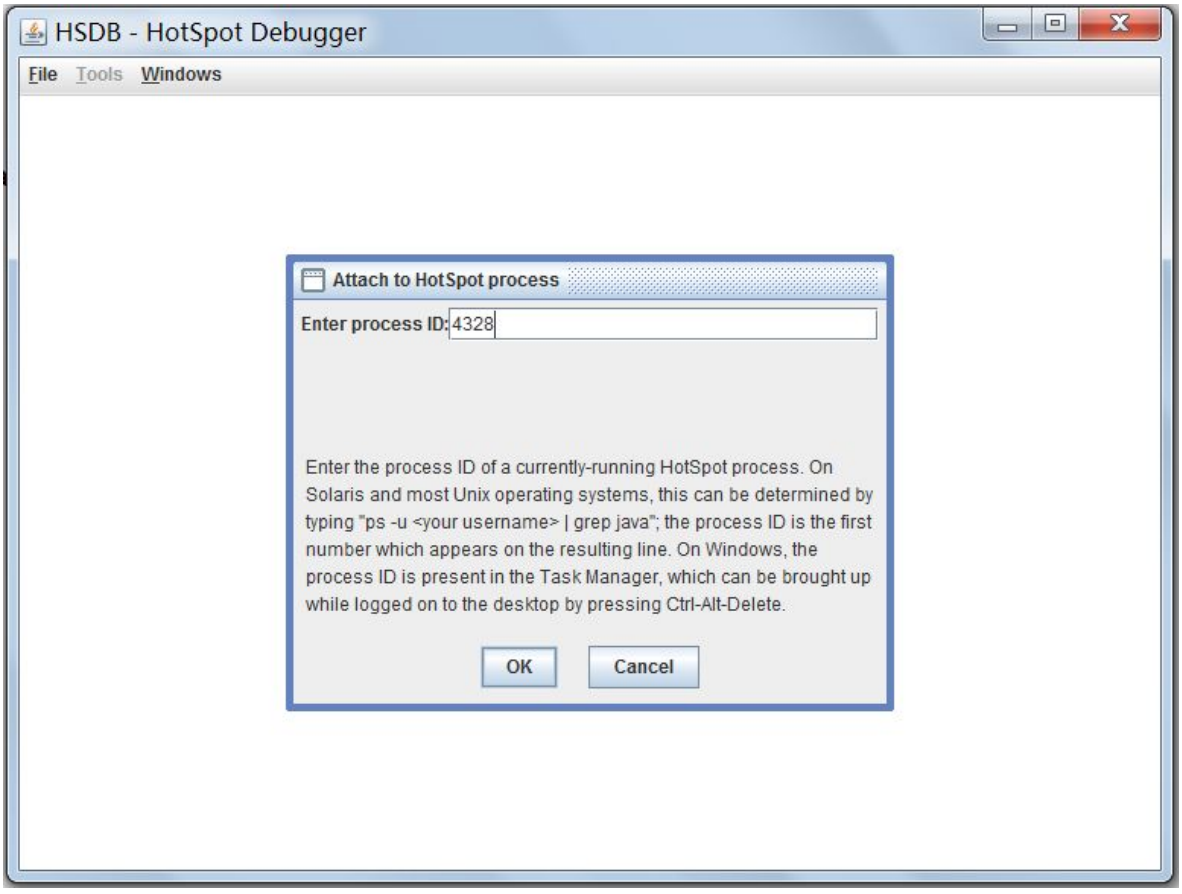
然后**启动HSDB**：

```
D:\test>java -cp .;%JAVA_HOME%/lib/sa-jdi.jar sun.jvm.hotspot.HSDB
```

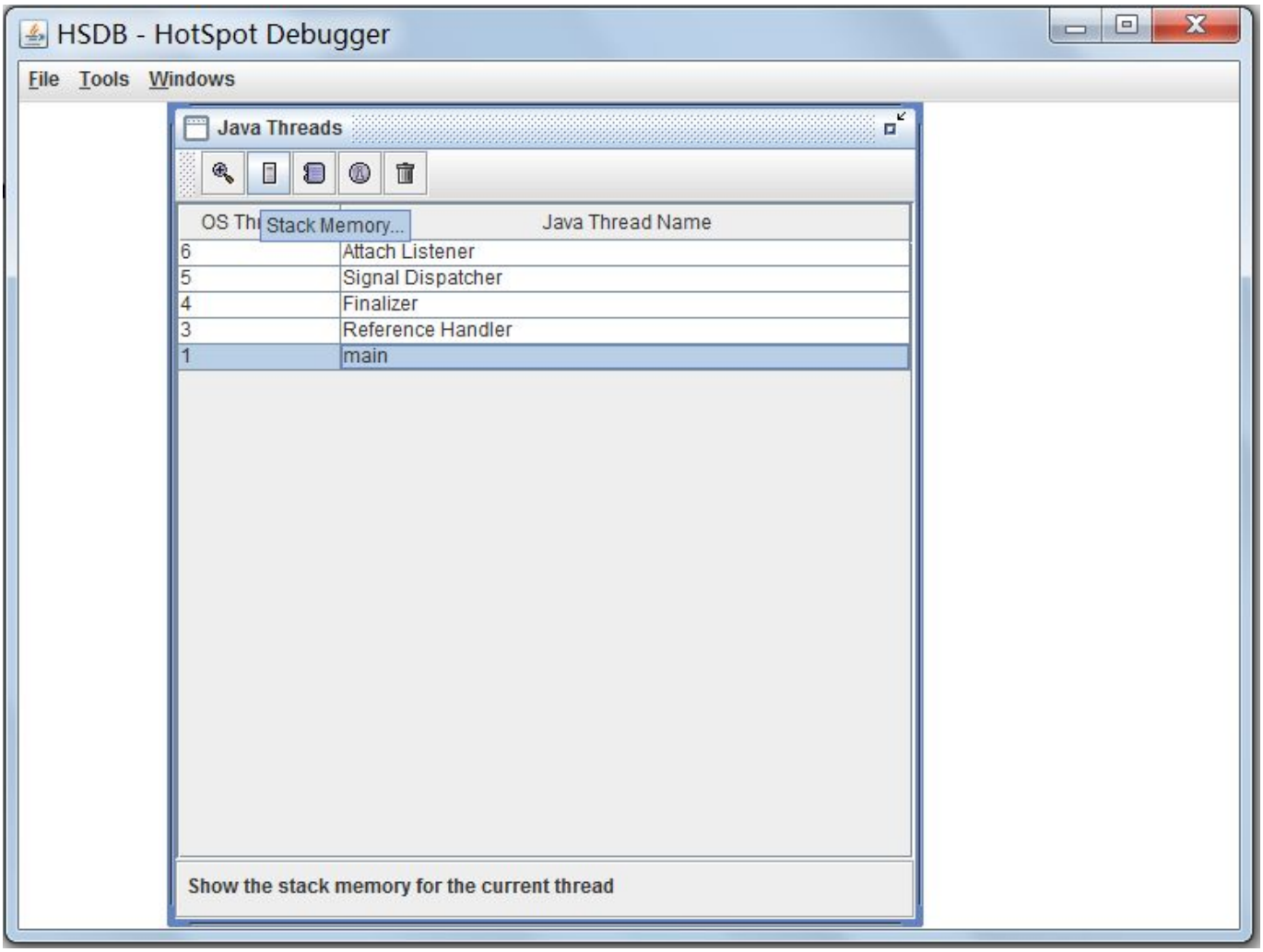
(要留意Linux和Solaris在Oracle/Sun JDK6就可以使用HSDB了, 但Windows上要到Oracle JDK7才可以用HSDB)
启动HSDB之后, 把它连接到目标进程上。从菜单里选择File -> Attach to HotSpot process：



在弹出的对话框里输入刚才记下的pid然后按OK：

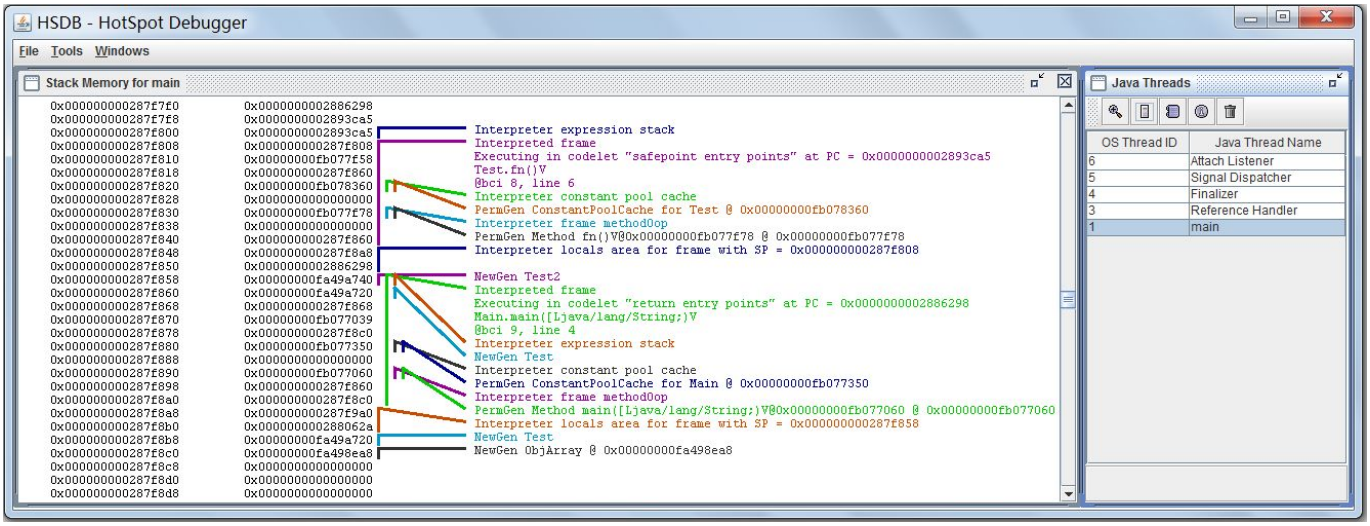


这会儿就连接到目标进程了：



刚开始打开的窗口是Java Threads, 里面有个线程列表。双击代表线程的行会打开一个Oop Inspector窗口显示HotSpot VM里记录线程的一些基本信息的C++对象的内容。

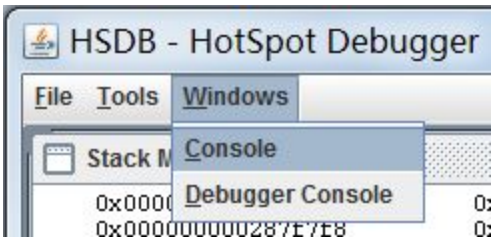
不过这里我们更可能会关心的是线程栈的内存数据。先选择main线程, 然后点击Java Threads窗口里的工具栏按钮从左数第2个可以打开Stack Memory窗口来显示main线程的栈：



Stack Memory窗口的内容有三栏：
左起第1栏是内存地址, 请让我提醒一下本文里提到“内存地址”的地方都是指虚拟内存意义上的地址, **不是**“物理内存地址”, 请不要弄混了这俩概念；
第2栏是该地址上存的数据, 以字宽为单位, 本文例子中我是在Windows 7 64-bit上跑64位的JDK7的HotSpot VM, 字宽是64位(8字节)；
第3栏是对数据的注释, 竖线表示范围, 横线或斜线连接范围与注释文字。

现在看不懂这个窗口里的数据没关系, 先放一边, 后面再回过头来看。

现在让我们打开HSDB里的控制台, 以便用命令来了解更多信息。
在菜单里选择Windows -> Console：



然后会得到一个空白的Command Line窗口。在里面敲一下回车就会出现hsdb>提示符。
(用过CLHSDB的同学可能会发现这就是把CLHSDB嵌入在了HSDB的图形界面里)

不知道有什么命令可用的同学可以先用help命令看看命令列表。

可以用universe命令来查看GC堆的地址范围和使用情况：

Hsdb代码

```
1.  hsdb> universe
2.  Heap Parameters:
3.  Gen 0:   eden [0x00000000fa400000,0x00000000fa4aad68,0x00000000fa6b0000) space capacity = 2818048,
    24.831088753633722 used
4.   from [0x00000000fa6b0000,0x00000000fa6b0000,0x00000000fa700000) space capacity = 327680, 0.0 used
5.   to   [0x00000000fa700000,0x00000000fa700000,0x00000000fa750000) space capacity = 327680, 0.0
    usedInvocations: 0
6.
7.  Gen 1:   old  [0x00000000fa750000,0x00000000fa750000,0x00000000fae00000) space capacity = 7012352, 0.0
    usedInvocations: 0
8.
9.   perm [0x00000000fae00000,0x00000000fb078898,0x00000000fc2c0000) space capacity = 21757952,
    11.90770160721009 usedInvocations: 0
```

这里用的是HotSpot VM的serial GC。GC堆由young gen = DefNewGeneration(包括eden和两个survivor space)、old gen = TenuredGeneration和perm gen = PermGen构成。
其中young gen和old gen构成了这种配置下HotSpot VM里的Java堆(Java heap), 而perm gen不属于Java heap的一部分, 它存储的主要是元数据或者叫反射信息, 主要用于实现JVM规范里的“方法区”概念。

在我们的Java代码里, 执行到Test.fn()末尾为止应该创建了3个Test2的实例。它们必然在GC堆里, 但都在哪里呢?用scanoops命令来看：

Hsdb代码

```
1.  hsdb> scanoops 0x00000000fa400000 0x00000000fc2c0000 Test2
2.  0x00000000fa49a710 Test2
3.  0x00000000fa49a730 Test2
4.  0x00000000fa49a740 Test2
```

scanoops接受两个必选参数和一个可选参数：必选参数是要扫描的地址范围, 一个是起始地址一个是结束地址；可选参数用于指定要扫描什么类型的对象实例。实际扫描的时候会扫出指定的类型及其派生类的实例。

这里可以看到确实扫出了3个Test2的实例。内容有两列：左边是对象的起始地址, 右边是对象的实际类型。
从它们所在的地址, 对照前面universe命令看到的GC堆的地址范围, 可以知道它们都在eden里。
通过whatis命令可以进一步知道它们都在eden之中分配给main线程的thread-local allocation buffer (TLAB)中：

Hsdb代码

```
1.  hsdb> whatis 0x00000000fa49a710
2.  Address 0x00000000fa49a710: In thread-local allocation buffer for thread "main" (1)
    [0x00000000fa48f490,0x00000000fa49a750,0x00000000fa49d118)
3.
4.  hsdb> whatis 0x00000000fa49a730
5.  Address 0x00000000fa49a730: In thread-local allocation buffer for thread "main" (1)
    [0x00000000fa48f490,0x00000000fa49a750,0x00000000fa49d118)
6.
7.  hsdb> whatis 0x00000000fa49a740
```

```
8.  Address 0x00000000fa49a740: In thread-local allocation buffer for thread "main" (1)
    [0x00000000fa48f490,0x00000000fa49a750,0x00000000fa49d118)
9.
10. hsdb>
```

还可以用inspect命令来查看对象的内容：

Hsdb代码

```
1. hsdb> inspect 0x00000000fa49a710
2. instance of Oop for Test2 @ 0x00000000fa49a710 @ 0x00000000fa49a710 (size = 16)
3. _mark: 1
```

可见一个Test2的实例要16字节。因为Test2类没有任何Java层的实例字段, 这里就没有任何Java实例字段可显示。不过本来这里还应该显示一行：

Hsdb代码

```
1. _metadata._compressed_klass: InstanceKlass for Test2 @ 0x00000000fb078608
```

不幸因为这个版本的HotSpot VM里带的SA有bug所以没显示出来。此bug在新版里已修。

还想看到更裸的数据的同学可以用mem命令来看实际内存里的数据长啥样：

Hsdb代码

```
1. hsdb> mem 0x00000000fa49a710 2
2. 0x00000000fa49a710: 0x0000000000000001
3. 0x00000000fa49a718: 0x00000000fb078608
```

mem命令接受的两个参数都必选, 一个是起始地址, 另一个是以字宽为单位的“长度”。我们知道一个Test2实例有16字节, 所以给定长度为2来看。

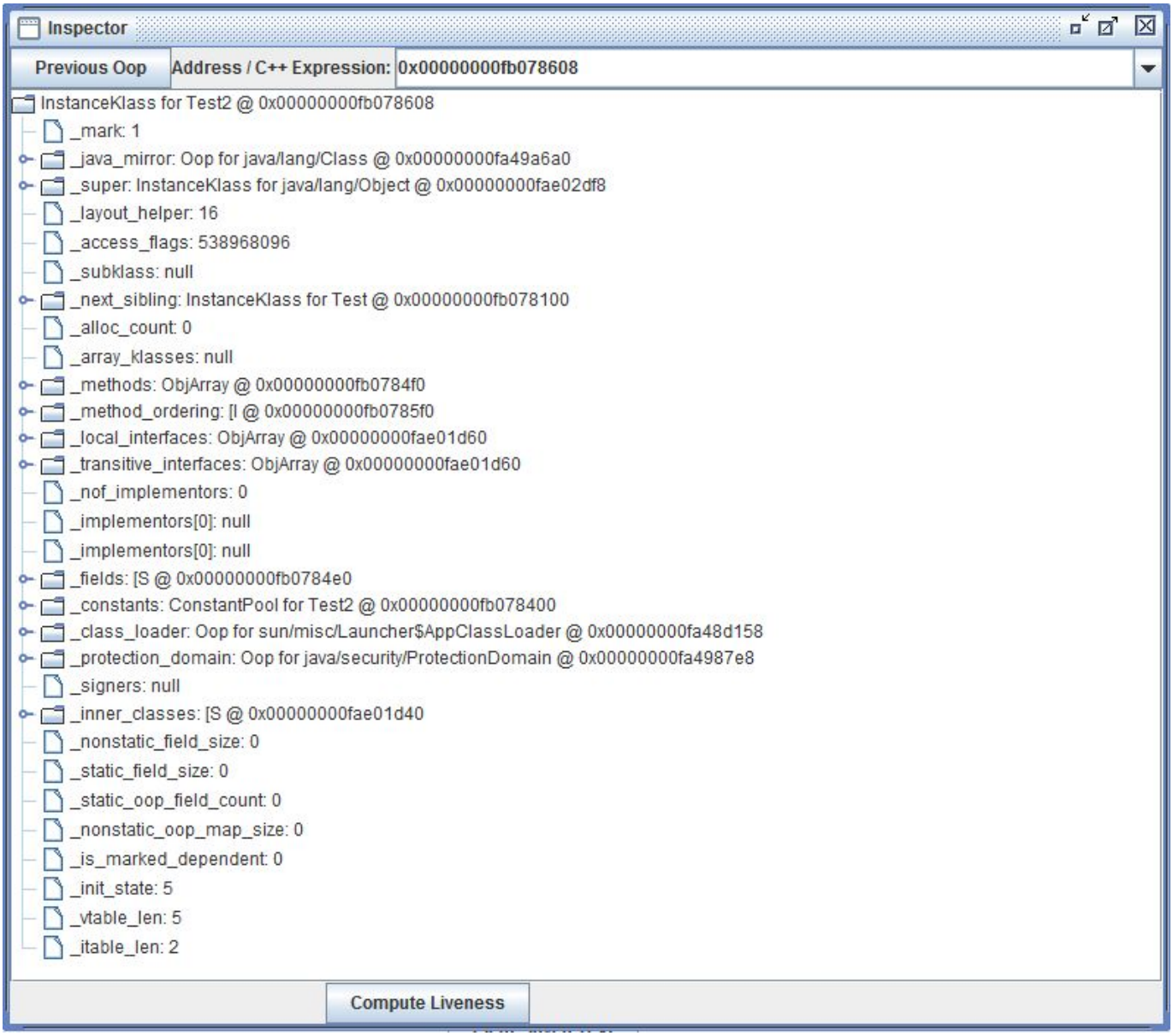
上面的数字都是啥来的呢?

Memory代码

```
1. 0x00000000fa49a710:  _mark:                  0x0000000000000001
2. 0x00000000fa49a718:  _metadata._compressed_klass:  0xfb078608
3. 0x00000000fa49a71c:  (padding):              0x00000000
```

一个Test2的实例包含2个给VM用的隐含字段作为对象头, 和0个Java字段。
对象头的第一个字段是mark word, 记录该对象的GC状态、同步状态、identity hash code之类的多种信息。
对象头的第二个字段是个类型信息指针, klass pointer。这里因为默认开启了压缩指针, 所以本来应该是64位的指针存在了32位字段里。
最后还有4个字节是为了满足对齐需求而做的填充(padding)。
以前在另一帖里也介绍过这部分内容, 可以参考：借助HotSpot SA来一窥PermGen上的对象

顺带发张Inspector的截图来展示HotSpot VM里描述Test2类的VM对象长啥样吧。
在菜单里选Tools -> Inspector, 在地址里输入前面看到的klass地址：



InstanceKlass存着Java类型的名字、继承关系、实现接口关系, 字段信息, 方法信息, 运行时常量池的指针, 还有内嵌的虚方法表(vtable)、接口方法表(itable)和记录对象里什么位置上有GC会关心的指针(oop map)等等。

留意到这个InstanceKlass是给VM内部用的, 并不直接暴露给Java层；InstanceKlass不是java.lang.Class的实例。
在HotSpot VM里, java.lang.Class的实例被称为“Java mirror”, 意思是它是VM内部用的klass对象的“镜像”, 把klass对象包装了一层来暴露给Java层使用。
在InstanceKlass里有个_java_mirror字段引用着它对应的Java mirror, 而mirror里也有个隐藏字段指向其对应的InstanceKlass。
所以当我们写obj.getClass(), 在HotSpot VM里实际上经过了两层间接引用才能找到最终的Class对象：

Java代码

1. obj->_klass->_java_mirror

在Oracle JDK7之前, Oracle/Sun JDK的HotSpot VM把Java类的静态变量存在InstanceKlass结构的末尾；从Oracle JDK7开始, 为了配合PermGen移除的工作, Java类的静态变量被挪到Java mirror(Class对象)的末尾了。
还有就是, 在JDK7之前Java mirror存放在PermGen里, 而从JDK7开始Java mirror默认也跟普通Java对象一样先从eden开始分配而不放在PermGen里。到JDK8则进一步彻底移除了PermGen, 把诸如klass之类的元数据都挪到GC堆之外管理, 而Java mirror的处理则跟JDK7一样。

==========

前面对HSDB的操作和HotSpot VM里的一些内部数据结构有了一定的了解, 现在让我们回到主题：找指针！

HotSpot VM内部使用直接指针来实现Java引用。在64位环境中有可能启用“压缩指针”的功能把64位指针压缩到只用32位来存。压缩指针与非压缩指针直接有非常简单的1对1对应关系，前者可以看作后者的特例。

于是我们要找t1、t2、t3这三个变量，等同于找出存有指向上述3个Test2实例的地址的存储位置。

不嫌麻烦的话手工扫描内存去找也能找到，不过幸好HSDB内建了revptrs命令，可以找出“反向指针”——如果a变量引用着b对象，那么从b对象出发去找a变量就是找一个“反向指针”。

先拿第一个Test2的实例试试看：

Hsdb代码

```
1. hsdb> revptrs 0x00000000fa49a710
2. Computing reverse pointers...
3. Done.
4. null
5. Oop for java/lang/Class @ 0x00000000fa499b00
```

还真的找到了一个包含指向Test2实例的指针，在一个java.lang.Class的实例里。
用whatis命令来看看这个Class对象在哪里：

Hsdb代码

```
1. hsdb> whatis 0x00000000fa499b00
2. Address 0x00000000fa499b00: In thread-local allocation buffer for thread "main" (1)
   [0x00000000fa48f490,0x00000000fa49a750,0x00000000fa49d118)
3.
```

可以看到这个Class对象也在eden里，具体来说在main线程的TLAB里。

这个Class对象是如何引用到Test2的实例的呢?再用inspect命令：

Hsdb代码

```
1. hsdb> inspect 0x00000000fa499b00
2. instance of Oop for java/lang/Class @ 0x00000000fa499b00 @ 0x00000000fa499b00 (size = 120)
3. <<Reverse pointers>>:
4. t1: Oop for Test2 @ 0x00000000fa49a710 Oop for Test2 @ 0x00000000fa49a710
```

可以看到，这个Class对象里存着Test类的静态变量t1，指向着第一个Test2实例。

成功找到t1了！这个有点特别，本来JVM规范里也没明确规定静态变量要存在哪里，通常认为它应该在概念中的“方法区”里；但现在在JDK7的HotSpot VM里它实质上也被放在Java heap里了。可以把这种特例看作是HotSpot VM把方法区的一部分数据也放在Java heap里了。
前面也已经提过，在JDK7之前的Oracle/Sun JDK里的HotSpot VM把静态变量存在InstanceKlass末尾，存在PermGen里。那个时候的PermGen更接近于完整的方法区一些。
关于PermGen移除计划的一些零星笔记可以参考我以前一老帖。

再接再厉，用revptrs看看第二个Test2实例有谁引用：

Hsdb代码

```
1. hsdb> revptrs 0x00000000fa49a730
2. Oop for Test @ 0x00000000fa49a720
```

找到了一个Test实例。同样用whatis来看看它在哪儿：

Hsdb代码

```
1. hsdb> whatis 0x00000000fa49a720
2. Address 0x00000000fa49a720: In thread-local allocation buffer for thread "main" (1)
   [0x00000000fa48f490,0x00000000fa49a750,0x00000000fa49d118)
3.
```

果然也在main线程的TLAB里。
然后看这个Test实例的内容：

Hsdb代码

```
1. hsdb> inspect 0x00000000fa49a720
2. instance of Oop for Test @ 0x00000000fa49a720 @ 0x00000000fa49a720 (size = 16)
3. <<Reverse pointers>>:
```

4. _mark: 1
5. t2: Oop for Test2 @ 0x00000000fa49a730 Oop for Test2 @ 0x00000000fa49a730

可以看到这个Test实例里有个成员字段t2, 指向了第二个Test2实例。

于是t2也找到了！在Java堆里, 作为Test的实例的成员字段存在。

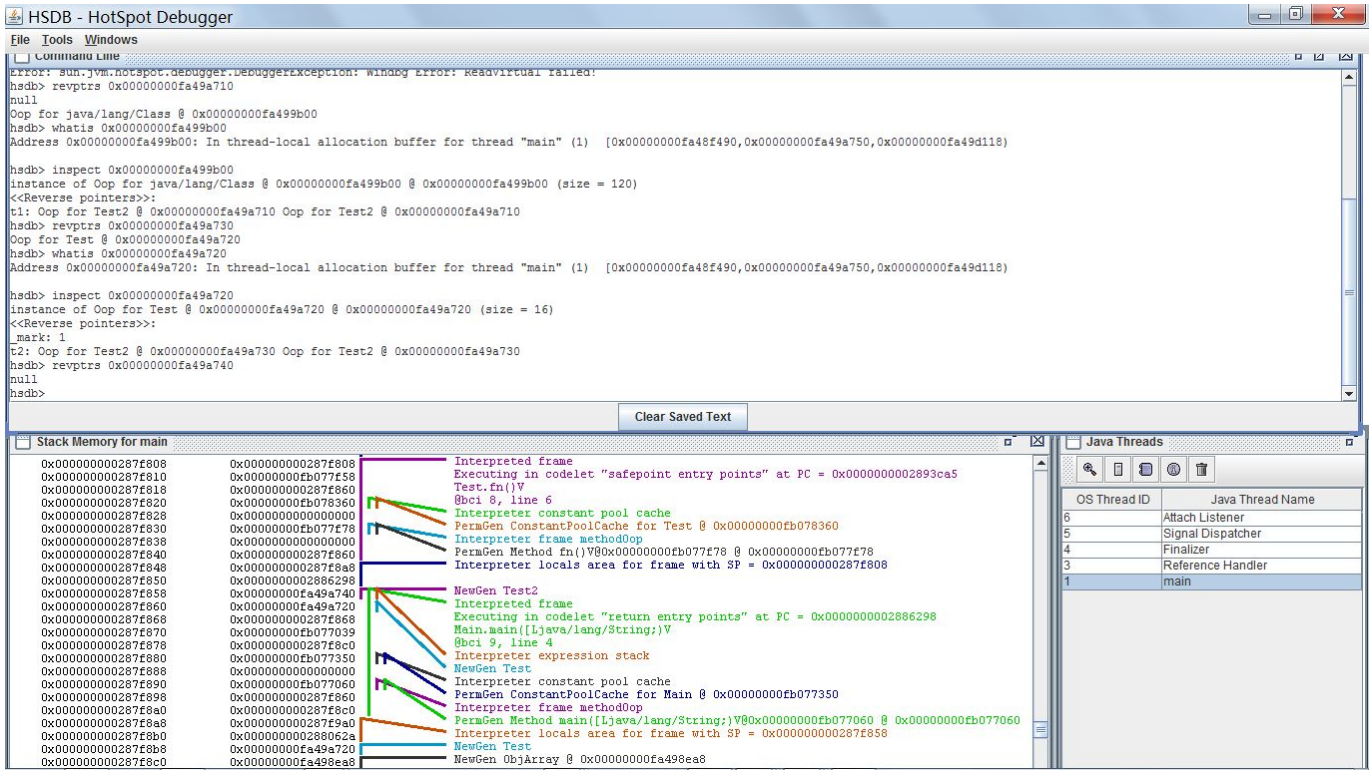
那么赶紧试试用revptrs命令看第三个Test2实例：

Hsdb代码

1. hsdb> revptrs 0x00000000fa49a740
2. null

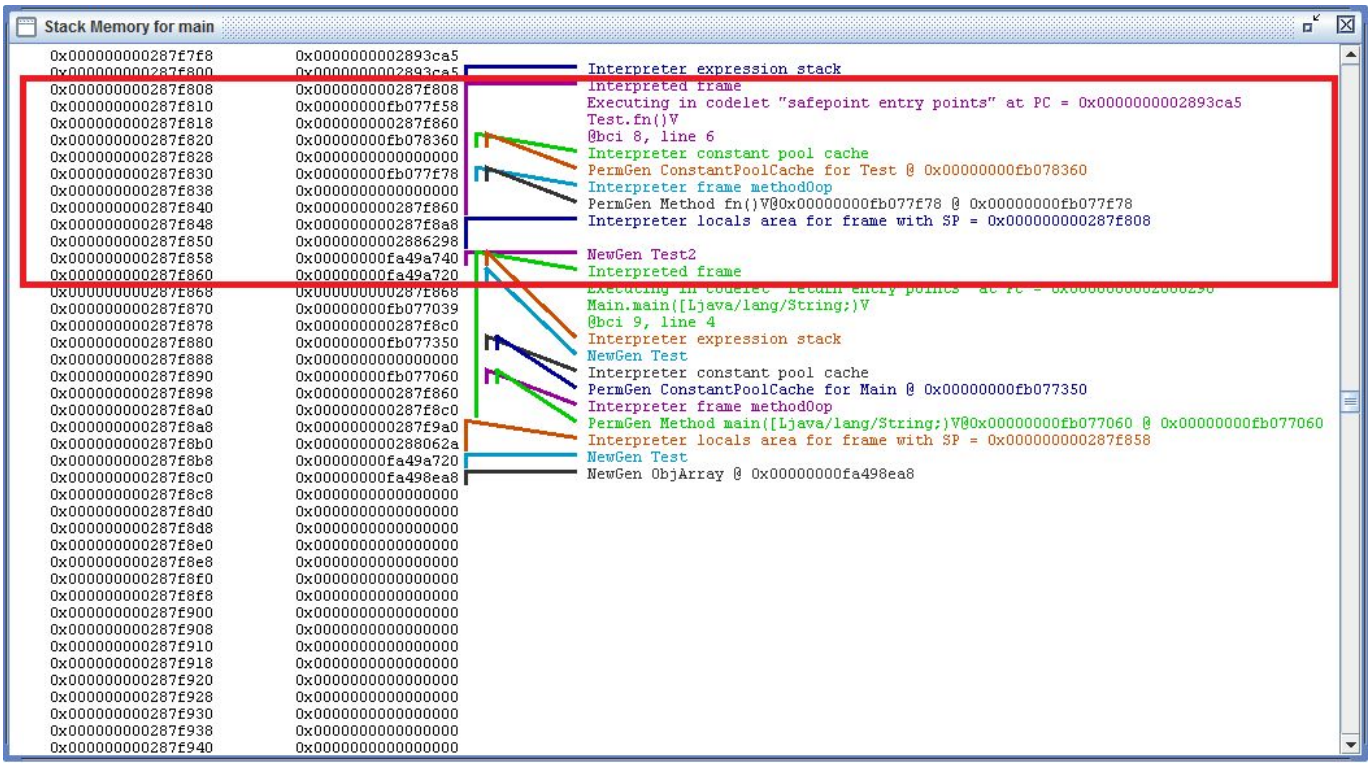
啥?没找到?！SA这也太弱小了吧。明明就在那里...
回头我会做个补丁让新版HotSpot VM的SA能处理这种情况。

这个时候的HSDB界面全貌：



0x00000000fa49a740看起来有没有点眼熟?
回到前面打开的Stack Memory窗口看, 仔细看会发现那个窗口里正好就有0x00000000fa49a740这数字, 位于0x000000000287f858地址上。

实际情况是, 下面这张图里红色框住的部分就是main线程上Test.fn()的调用对应的栈帧：



如果图里看得不清楚的话, 我再用文字重新写一遍(两道横线之间的是Test.fn()的栈帧内容, 前后的则是别的东西) :

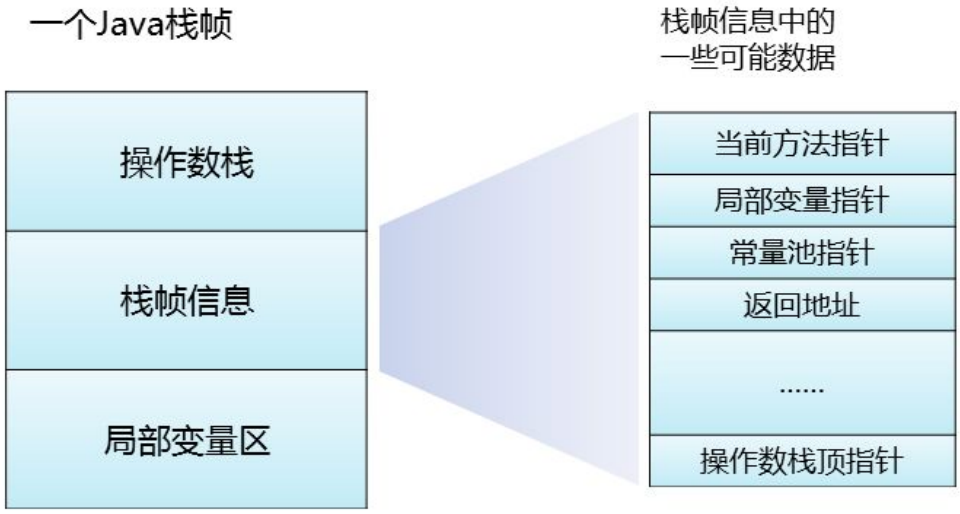
Memory代码

1. 0x000000000287f7f0: 0x0000000002886298
2. 0x000000000287f7f8: 0x0000000002893ca5
3. 0x000000000287f800: 0x0000000002893ca5
4. ---
5. Stack frame for Test.fn() @bci=8, line=6, pc=0x0000000002893ca5, methodOop=0x00000000fb077f78 (Interpreted frame)
6. 0x000000000287f808: 0x000000000287f808 expression stack bottom <- rsp
7. 0x000000000287f810: 0x00000000fb077f58 bytecode pointer = 0x00000000fb077f50 (base) + 8 (bytecode index) in PermGen
8. 0x000000000287f818: 0x000000000287f860 pointer to locals
9. 0x000000000287f820: 0x00000000fb078360 constant pool cache = ConstantPoolCache for Test in PermGen
10. 0x000000000287f828: 0x0000000000000000 method data oop = null
11. 0x000000000287f830: 0x00000000fb077f78 method oop = Method for Test.fn()V in PermGen
12. 0x000000000287f838: 0x0000000000000000 last Java stack pointer (not set)
13. 0x000000000287f840: 0x000000000287f860 old stack pointer (saved rsp)
14. 0x000000000287f848: 0x000000000287f8a8 old frame pointer (saved rbp) <- rbp
15. 0x000000000287f850: 0x0000000002886298 return address = in interpreter codelet "return entry points" [0x00000000028858b8, 0x00000000028876c0) 7688 bytes
16. 0x000000000287f858: 0x00000000fa49a740 local[1] "t3" = Oop for Test2 in NewGen
17. 0x000000000287f860: 0x00000000fa49a720 local[0] "this" = Oop for Test in NewGen
18. ---
19. 0x000000000287f868: 0x000000000287f868
20. 0x000000000287f870: 0x00000000fb077039
21. 0x000000000287f878: 0x000000000287f8c0
22. 0x000000000287f880: 0x00000000fb077350
23. 0x000000000287f888: 0x0000000000000000
24. 0x000000000287f890: 0x00000000fb077060
25. 0x000000000287f898: 0x000000000287f860
26. 0x000000000287f8a0: 0x000000000287f8c0
27. 0x000000000287f8a8: 0x000000000287f9a0
28. 0x000000000287f8b0: 0x000000000288062a
29. 0x000000000287f8b8: 0x00000000fa49a720
30. 0x000000000287f8c0: 0x00000000fa498ea8

31. 0x000000000287f8c8: 0x0000000000000000
32. 0x000000000287f8d0: 0x0000000000000000
33. 0x000000000287f8d8: 0x0000000000000000

回顾JVM规范里所描述的Java栈帧结构, 包括：
[操作数栈 (operand stack)]
[栈帧信息 (dynamic linking)]
[局部变量区 (local variables)]

上张我以前做的投影稿里的图：

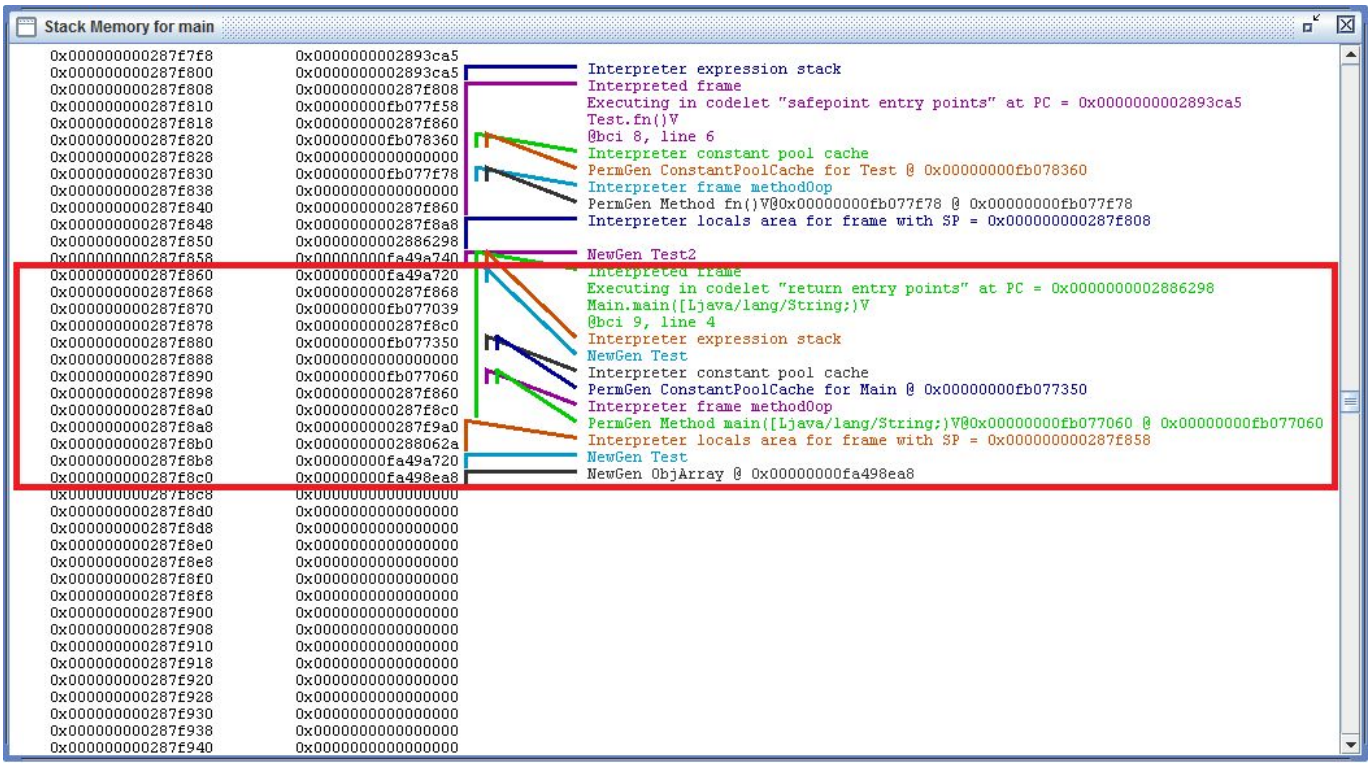


再跟HotSpot VM的解释器所使用的栈帧布局对比看看, 是不是正好能对应上?局部变量区(locals)有了, VM所需的栈帧信息也有了；执行到这个位置operand stack正好是空的所以看不到它。
(HotSpot VM里把operand stack叫做expression stack。这是因为operand stack通常只在表达式求值过程中才有内容)

从Test.fn()的栈帧中我们可以看到t3变量就在locals[1]的位置上。t3变量也找到了！大功告成！

栈帧信息里具体都是些啥, 以后有机会再展开讲吧。

都看到这里了, 干脆把main方法的栈帧也如法炮制分析一下。先上图：



然后再用文字写一次：

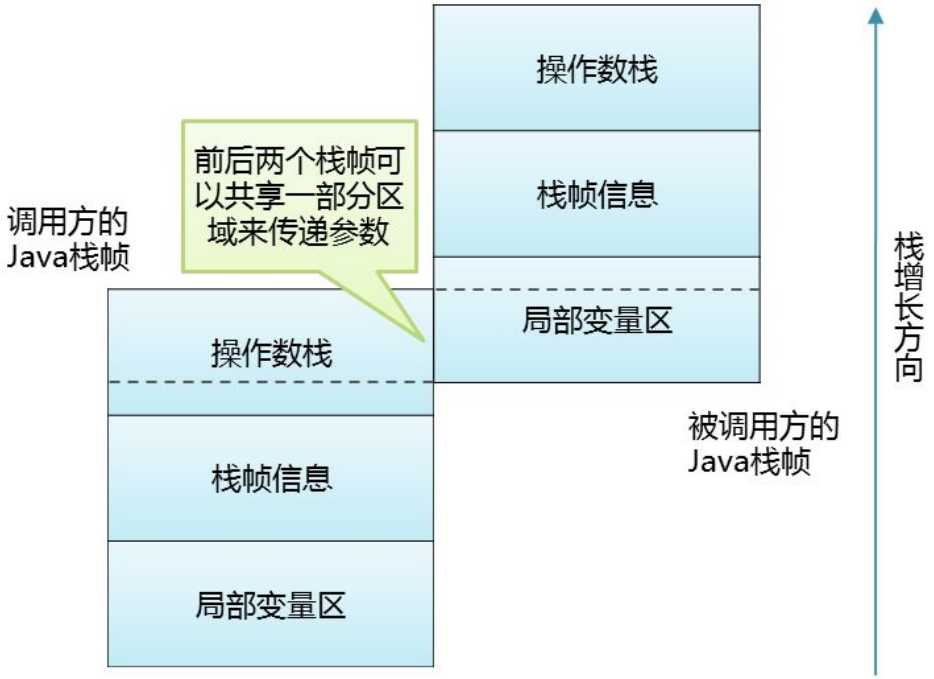
Memory代码

1. 0x000000000287f7f0: 0x0000000002886298
2. 0x000000000287f7f8: 0x0000000002893ca5
3. 0x000000000287f800: 0x0000000002893ca5
4. 0x000000000287f808: 0x000000000287f808
5. 0x000000000287f810: 0x00000000fb077f58
6. 0x000000000287f818: 0x000000000287f860
7. 0x000000000287f820: 0x00000000fb078360
8. 0x000000000287f828: 0x0000000000000000
9. 0x000000000287f830: 0x00000000fb077f78
10. 0x000000000287f838: 0x0000000000000000
11. 0x000000000287f840: 0x000000000287f860
12. 0x000000000287f848: 0x000000000287f8a8
13. 0x000000000287f850: 0x0000000002886298
14. 0x000000000287f858: 0x00000000fa49a740
15. ---
16. Stack frame for Main.main(java.lang.String[]) @bci=9, line=4, pc=0x0000000002886298, methodOop=0x00000000fb077060 (Interpreted frame)
17. 0x000000000287f860: 0x00000000fa49a720 expression stack[0] = Oop for Test in NewGen
18. 0x000000000287f868: 0x000000000287f868 expression stack bottom
19. 0x000000000287f870: 0x00000000fb077039 bytecode pointer = 0x00000000fb077030 (base) + 9 (bytecode index) in PermGen
20. 0x000000000287f878: 0x000000000287f8c0 pointer to locals
21. 0x000000000287f880: 0x00000000fb077350 constant pool cache = ConstantPoolCache for Main in PermGen
22. 0x000000000287f888: 0x0000000000000000 method data oop = null
23. 0x000000000287f890: 0x00000000fb077060 method oop = Method for Main.main([Ljava/lang/String;)V in PermGen
24. 0x000000000287f898: 0x000000000287f860 last Java stack pointer
25. 0x000000000287f8a0: 0x000000000287f8c0 old stack pointer
26. 0x000000000287f8a8: 0x000000000287f9a0 old frame pointer
27. 0x000000000287f8b0: 0x000000000288062a return address = in StubRoutines
28. 0x000000000287f8b8: 0x00000000fa49a720 local[1] "test" = Oop for Test in NewGen
29. 0x000000000287f8c0: 0x00000000fa498ea8 local[0] "args" = Oop for java.lang.String[] in NewGen
30. ---
31. 0x000000000287f8c8: 0x0000000000000000
32. 0x000000000287f8d0: 0x0000000000000000
33. 0x000000000287f8d8: 0x0000000000000000

main的栈帧的operand stack就不是空的了，有一个元素，用来传递参数给其调用的Test.fn()方法(作为“this”)。

仔细的同学可能发现了, 0x00000000287f860这个地址前面不是说是调用Test.fn()产生的栈帧么?怎么这里又变成调用main()方法的栈帧的一部分了呢?

其实栈帧直接可以有重叠：(再上一张以前做的投影稿里的图)



这样可以减少传递参数所需的数据拷贝, 也节省了空间。

回到HSDB, 我们换个方式来把t3变量找出来。这里就需要编译Test.java时给的-g参数所生成的LocalVariableTable的信息了：

Hsdb代码

```
1.  hsdb> jseval "ts = jvm.threads"
2.  [Thread (address=0x00000000fa48fb38, name=Service Thread), Thread (address=0x00000000fa48fa18, name=C2
    CompilerThread1), Thread (address=0x00000000fa48f8f8, name=C2 CompilerThread0), Thread
    (address=0x00000000fa49d178, name=JDWP Command Reader), Thread (address=0x00000000fa48f820,
    name=JDWP Event Helper Thread), Thread (address=0x00000000fa48f6d8, name=JDWP Transport Listener:
    dt_shmem), Thread (address=0x00000000fa48dc88, name=Attach Listener), Thread
    (address=0x00000000fa48db68, name=Signal Dispatcher), Thread (address=0x00000000fa405828,
    name=Finalizer), Thread (address=0x00000000fa4053a0, name=Reference Handler), Thread
    (address=0x00000000fa404860, name=main)]
3.  hsdb> jseval "t = ts[ts.length - 1]"
4.  Thread (address=0x00000000fa404860, name=main)
5.  hsdb> jseval "fs = t.frames"
6.  [Frame (method=Test.fn(), bci=8, line=6), Frame (method=Main.main(java.lang.String[]), bci=9, line=4)]
7.  hsdb> jseval "f0 = fs[0]"
8.  Frame (method=Test.fn(), bci=8, line=6)
9.  hsdb> jseval "f1 = fs[1]"
10. Frame (method=Main.main(java.lang.String[]), bci=9, line=4)
11. hsdb> jseval "f0.locals"
12. {t3=Object 0x00000000fa49a740}
13. hsdb>
```

==========

上面讲栈帧布局的时候出现了“bytecode pointer”字眼。既然之前被不少好奇的同学问过“JVM里字节码存在哪里”, 这里就一并回答掉好了。

强调一点：“字节码”只是元数据的一部分。它只负责描述运行逻辑, 而其它信息像是类型名、成员的个数、类型、名字等等都**不是字节码**。在Class文件里是如此, 到运行时在JVM里仍然是如此。

HotSpot VM里有一套对象专门用来存放元数据, 它们包括：

- Klass系对象。元数据的最主要入口。用于描述类型的总体信息

- ConstantPool/ConstantPoolCache对象。每个InstanceKlass关联着一个ConstantPool, 作为该类型的运行时常量池。这个常量池的结构跟Class文件里的常量池基本上是对应的。可以参考我以前的一个回帖。ConstantPoolCache主要用于存储某些字节码指令所需的解析(resolve)好的常量项, 例如给[get|put]static、[get|put]field、invoke[static|special|virtual|interface|dynamic]等指令对应的常量池项用。
- Method对象, 用来描述Java方法的总体信息, 像是方法入口地址、调用/循环计数器等等
- ConstMethod对象, 记录着Java方法的不变的描述信息, 包括方法名、方法的访问修饰符、**字节码**、行号表、局部变量表等等。注意了, 字节码就嵌在这ConstMethod对象里面。
- Symbol对象, 对应Class文件常量池里的JVM_CONSTANT_Utf8类型的常量。有一个VM全局的SymbolTable管理着所有Symbol。Symbol由所有Java类所共享。
- MethodData对象, 记录着Java方法执行时的profile信息, 例如某方法里的某个字节码之类是否从来没遇到过null, 某个条件跳转是否总是走同一个分支, 等等。这些信息在解释器(多层编译模式下也在低层的编译生成的代码里)收集, 然后供给HotSpot Server Compiler用于做激进优化。

在PermGen移除前, 上述元数据对象都在PermGen里, 直接被GC管理着。
JDK8彻底移除PermGen后, 这些对象被挪到GC堆外的一块叫做Metaspace的空间里做特殊管理, 仍然间接的受GC管理。

介绍了背景, 让我们回到HSDB里。前面不是说“bytecode pointer (bcp)”嘛, 从背景介绍可以知道字节码存在ConstMethod对象里, 那就让我们用Test.fn()栈帧里存的bcp来验证一下是否真的如此。
还是用whatis命令：

Hsdb代码

```
1. hsdb> whatis 0x00000000fb077f58
2. Address 0x00000000fb077f58: In perm generation   perm [0x00000000fae00000,0x00000000fb078898,0x00000000fc2c0000) space capacity = 21757952, 11.90770160721009 used
```

这地址确实在PermGen里了。那么inspect一下看看?

Hsdb代码

```
1. hsdb> inspect 0x00000000fb077f58
2. Error: sun.jvm.hotspot.debugger.UnalignedAddressException: 100011
```

呃, 这样不行。inspect命令只能接受对象的起始地址, 但字节码是嵌在ConstMethod对象中间的。

那换条路子。栈帧里还有method oop, 指向该栈帧对应的Method对象。先从它入手：

Hsdb代码

```
1. hsdb> inspect 0x00000000fb077f78
2. instance of Method fn()V@0x00000000fb077f78 @ 0x00000000fb077f78 @ 0x00000000fb077f78 (size = 136)
3. _mark: 1
4. _constMethod: ConstMethod fn()V@0x00000000fb077f08 @ 0x00000000fb077f08 Oop @ 0x00000000fb077f08
5. _constants: ConstantPool for Test @ 0x00000000fb077c68 Oop @ 0x00000000fb077c68
6. _method_size: 17
7. _max_stack: 2
8. _max_locals: 2
9. _size_of_parameters: 1
10. _access_flags: 1
```

这样就找到了Test.fn()的Method对象, 看到里面的_constMethod字段所指向的ConstMethod对象：

Hsdb代码

```
1. hsdb> inspect 0x00000000fb077f08
2. instance of ConstMethod fn()V@0x00000000fb077f08 @ 0x00000000fb077f08 @ 0x00000000fb077f08 (size = 112)
3. _mark: 1
4. _method: Method fn()V@0x00000000fb077f78 @ 0x00000000fb077f78 Oop @ 0x00000000fb077f78
5. _exception_table: [I @ 0x00000000fae01d50 Oop for [I @ 0x00000000fae01d50
6. _constMethod_size: 14
7. _flags: 5
8. _code_size: 9
9. _name_index: 18
10. _signature_index: 12
11. _generic_signature_index: 0
12. _code_size: 9
```

这个ConstMethod对象从0x00000000fb077f08开始, 长度112字节, 也就是这个对象的范围是[0x00000000fb077f08, 0x00000000fb077f78)。bcp指向0x00000000fb077f58, 确实在这个ConstMethod范围内。

通过经验可以知道实际上这里字节码的起始地址是0x00000000fb077f50。经验就是：字节码是ConstMethod内嵌的第一个变长表, 紧帖在ConstMethod的最后一个显式C++字段后面。所以只要知道sizeof(constMethodOopDesc), 字节码就会从这个偏移量开始。
通过ConstMethod的_code_size字段可以知道该方法的字节码有9字节。找出来用mem命令看看内存里的数据：

Hsdb代码

```
1. hsdb> mem 0x00000000fb077f50 2
2. 0x00000000fb077f50: 0x4c0001b7590200ca
3. 0x00000000fb077f58: 0x0000000004105b1
```

这串数字是什么东西呢?展开来写清楚一点就是：

Memory代码

```
1. 0x00000000fb077f50:  bb 00 02  new <cp index #2> [Class Test2]
2. 0x00000000fb077f53:  59        dup
3. 0x00000000fb077f54:  b7 01 00  invokespecial <cp cache index #1> [Method Test2.<init>()V]
4. 0x00000000fb077f57:  4c        astore_1
5. 0x00000000fb077f58:  b1        return
```

眼尖的同学要吐槽了：在0x00000000fb077f50的字节不是0xca么, 怎么变成0xbb了?
其实0xca是JVM规范里有描述的一个可选字节码指令, breakpoint

Memory代码

```
1. 0x00000000fb077f50:  ca 00 02  breakpoint // 00 02 not used
```

还记得本文的实验一开始用了jdb在Test.fn()的入口设置了断点吗?这就是结果——入口处的字节码指令被改写为breakpoint了。当然, 原本的字节码指令也还在别的地方存着, 等断点解除之后这个位置就会被恢复成原本的0xbb指令。

把ConstMethod里存的字节码跟Class文件里存的比较一下看看。用javap工具来看Class文件的内容：

Javap代码

```
1. public void fn();
2.   Code:
3.     stack=2, locals=2, args_size=1
4.        0:  bb 00 02  new         #2             // class Test2
5.        3:  59        dup
6.        4:  b7 00 03  invokespecial #3           // Method Test2."<init>":()V
7.        7:  4c        astore_1
8.        8:  b1        return
```

几乎一模一样。唯一的不同也是个有趣的小细节：invokespecial的参数的常量池号码不一样了。HotSpot VM执行new指令的时候用的还是Class文件里的常量池号和字节序。而在执行invokespecial时, 光是ConstantPool里的的常量项不够地方放解析(resolve)出来的信息, 所以把这些信息放在ConstantPoolCache里, 然后也把invokespecial指令里的参数改写过来, 顺带变成了平台相关的字节序。

同样也看看Main.main()方法。内存内容：

Memory代码

```
1. hsdb> mem 0x00000000fb077030 2
2. 0x00000000fb077030: 0x4c0001b7590200bb
3. 0x00000000fb077038: 0x214103b10002b62b
```

展开来注解：

Memory代码

```
1. 0x00000000fb077030:  bb 00 02  new <cp index #2> [Class Test]
2. 0x00000000fb077033:  59        dup
3. 0x00000000fb077034:  b7 01 00  invokespecial <cp cache index #1> [Method Test.<init>()V]
4. 0x00000000fb077037:  4c        astore_1
5. 0x00000000fb077038:  2b        aload_1
6. 0x00000000fb077039:  b6 02 00  invokevirtual <cp cache index #2> [Method Test.fn()V]
```

```
7.  0x00000000fb07703c:  b1      return
```

对应的javap输出：

Javap代码

```
1.  public static void main(java.lang.String[]);
2.    Code:
3.      stack=2, locals=2, args_size=1
4.        0:  bb 00 02  new         #2              // class Test
5.        3:  59        dup
6.        4:  b7 00 03  invokespecial #3            // Method Test."<init>":()V
7.        7:  4c        astore_1
8.        8:  2b        aload_1
9.        9:  b6 00 04  invokevirtual #4            // Method Test.fn:()V
10.      12:  b1        return
```

好, 今天就写到这里吧～

豆列：从表到里学习JVM实现 | 记GreenTeaJUG第二次线下活动(杭州)

- 2013-04-16 09:08

评论

19 楼 jiewo 2014-08-11
被科普了。R打果真神人也！！！！

18 楼 RednaxelaFX 2014-07-31

> **playboy0651140 写道**
> 打扰下R大,我用whatis 命令
> whatis 0x00000007d6f00000
> 总是返回的是
> Address 0x00000007d6f00000: In unknown section of Java heap
> 不知道哪里操作有问题..用的jdk1.7的在suse和win 7下面都是这个问题

呵呵, 那是因为Serviceability Agent对GenCollectedHeap系的GC支持得比较好, 而对另外俩GC(Parallel GC和G1 GC)支持得没那么好。猜您是在用Parallel GC?其实稍微改造一下Serviceability Agent的Java部分就可以让whatis正确显示在哪里了。

17 楼 playboy0651140 2014-07-31
打扰下R大,我用whatis 命令
whatis 0x00000007d6f00000
总是返回的是
Address 0x00000007d6f00000: In unknown section of Java heap
不知道哪里操作有问题..用的jdk1.7的在suse和win 7下面都是这个问题

16 楼 minixalpha 2014-04-21
请问是否有什么方法可以得到运行时数据区中, 线程的程序计数器在某一时刻的值, 找了很久, 都没有找到有什么好办法, 先行谢过！

15 楼 fh63045 2014-01-21
终于找到这类文章了... 感谢大神

14 楼 0ooyooo 2013-10-23

> **0ooyooo 写道**
> > **0ooyooo 写道**
> > C:\Program Files\Java\jdk1.7.0_45\lib>java -cp ./sa-jdi.jar sun.jvm.hotspot.HSDB
> >
> > Exception in thread "Thread-1" java.lang.UnsatisfiedLinkError: Can't load librar
> > y: C:\Program Files\Java\jre7\bin\sawindbg.dll
> > at java.lang.ClassLoader.loadLibrary(Unknown Source)
> > at java.lang.Runtime.load0(Unknown Source)
> > at java.lang.System.load(Unknown Source)
> > at sun.jvm.hotspot.debugger.windbg.WindbgDebuggerLocal.<clinit>(WindbgDebuggerLocal.java:651)
> > at sun.jvm.hotspot.HotSpotAgent.setupDebuggerWin32(HotSpotAgent.java:521)
> > at sun.jvm.hotspot.HotSpotAgent.setupDebugger(HotSpotAgent.java:336)
> > at sun.jvm.hotspot.HotSpotAgent.go(HotSpotAgent.java:313)
> > at sun.jvm.hotspot.HotSpotAgent.attach(HotSpotAgent.java:157)
> > at sun.jvm.hotspot.HSDB.attach(HSDB.java:1168)
> > at sun.jvm.hotspot.HSDB.access$1700(HSDB.java:53)

at sun.jvm.hotspot.HSDB$25$1.run(HSDB.java:436)
at sun.jvm.hotspot.utilities.WorkerThread$MainLoop.run(WorkerThread.java:66)
at java.lang.Thread.run(Unknown Source)

我的jdk1.7.0_45里没有sawindbg.dll这个文件, 为什么呢?R大有遇到过这种情况吗?

在这个下面找到了, 打扰R大了, 我再看看。

C:\Program Files\Java\jdk1.7.0_45\jre\bin下面
13 楼 oooyooo 2013-10-23
oooyooo 写道
C:\Program Files\Java\jdk1.7.0_45\lib>java -cp ./sa-jdi.jar sun.jvm.hotspot.HSDB

Exception in thread "Thread-1" java.lang.UnsatisfiedLinkError: Can't load librar
y: C:\Program Files\Java\jre7\bin\sawindbg.dll
at java.lang.ClassLoader.loadLibrary(Unknown Source)
at java.lang.Runtime.load0(Unknown Source)
at java.lang.System.load(Unknown Source)
at sun.jvm.hotspot.debugger.windbg.WindbgDebuggerLocal.<clinit>(WindbgDebuggerLocal.java:651)
at sun.jvm.hotspot.HotSpotAgent.setupDebuggerWin32(HotSpotAgent.java:521)
at sun.jvm.hotspot.HotSpotAgent.setupDebugger(HotSpotAgent.java:336)
at sun.jvm.hotspot.HotSpotAgent.go(HotSpotAgent.java:313)
at sun.jvm.hotspot.HotSpotAgent.attach(HotSpotAgent.java:157)
at sun.jvm.hotspot.HSDB.attach(HSDB.java:1168)
at sun.jvm.hotspot.HSDB.access$1700(HSDB.java:53)
at sun.jvm.hotspot.HSDB$25$1.run(HSDB.java:436)
at sun.jvm.hotspot.utilities.WorkerThread$MainLoop.run(WorkerThread.java:66)
at java.lang.Thread.run(Unknown Source)

我的jdk1.7.0_45里没有sawindbg.dll这个文件, 为什么呢?R大有遇到过这种情况吗?

在这个下面找到了, 打扰R大了, 我再看看。
12 楼 oooyooo 2013-10-23
C:\Program Files\Java\jdk1.7.0_45\lib>java -cp ./sa-jdi.jar sun.jvm.hotspot.HSDB

Exception in thread "Thread-1" java.lang.UnsatisfiedLinkError: Can't load librar
y: C:\Program Files\Java\jre7\bin\sawindbg.dll
at java.lang.ClassLoader.loadLibrary(Unknown Source)
at java.lang.Runtime.load0(Unknown Source)
at java.lang.System.load(Unknown Source)
at sun.jvm.hotspot.debugger.windbg.WindbgDebuggerLocal.<clinit>(WindbgDebuggerLocal.java:651)
at sun.jvm.hotspot.HotSpotAgent.setupDebuggerWin32(HotSpotAgent.java:521)
at sun.jvm.hotspot.HotSpotAgent.setupDebugger(HotSpotAgent.java:336)
at sun.jvm.hotspot.HotSpotAgent.go(HotSpotAgent.java:313)
at sun.jvm.hotspot.HotSpotAgent.attach(HotSpotAgent.java:157)
at sun.jvm.hotspot.HSDB.attach(HSDB.java:1168)
at sun.jvm.hotspot.HSDB.access$1700(HSDB.java:53)
at sun.jvm.hotspot.HSDB$25$1.run(HSDB.java:436)
at sun.jvm.hotspot.utilities.WorkerThread$MainLoop.run(WorkerThread.java:66)
at java.lang.Thread.run(Unknown Source)

我的jdk1.7.0_45里没有sawindbg.dll这个文件, 为什么呢?R大有遇到过这种情况吗?
11 楼 finallygo 2013-09-30
补充下服务器jvm的版本信息:

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)
10 楼 finallygo 2013-09-30

你好,我在使用jdb进行远程调试的时候,没一会儿之后就会出现,远程连接已断开 的提示,这是我哪个地方设置的有问题么? 我服务器配置的参数是:

Java代码

```
1.  -agentlib:jdwp=transport=dt_socket,address=localhost:8765,server=y,suspend=n
```

9 楼 RednaxelaFX 2013-08-14

> **woosheep 写道**
> R大, 你的javap怎么做到输出字节码?
>
> 像这样0: bb 00 02 new #2
>
> 输出bb 00 02?

javap自己是没这个功能的, 至少JDK6带的javap没有。
可以参考javap在OpenJDK 6的实现：
http://hg.openjdk.java.net/jdk6/jdk6/langtools/file/81689043bd7f/src/share/classes/sun/tools/javap/JavapPrinter.javaprintInstr()
以及最新的OpenJDK 8的实现：
http://hg.openjdk.java.net/jdk8/jdk8/langtools/file/76cfe7c61f25/src/share/classes/com/sun/tools/javap/CodeWriter.javawriteInstr()

我当时发这帖的时候是人肉对着JVM规范和Class文件内容来查出这些字节码对应的字节的。Serviceability Agent里的jdis之类则可以输出字节出来。

8 楼 woosheep 2013-08-14

R大, 你的javap怎么做到输出字节码?

像这样0: bb 00 02 new #2

输出bb 00 02?

7 楼 RednaxelaFX 2013-04-16

> **General_PF 写道**
> > **RednaxelaFX 写道**
> > ConstantPool/ConstantPoolCache对象。每个InstanceKlass关联着一个ConstantPool, 作为该类型的运行时常量池。
>
> 这个具体指什么?我看到你的回帖里面贴的图, 既有 fieldref, 也有 methodref。为什么叫做 constantPool?

回答这个问题最简单精确的办法就是让你去读JVM规范：
首先看4.4 The Constant Pool：http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4
然后看2.5.5 Run-time Constant Pool：
http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.5.5

> **General_PF 写道**
> > **RednaxelaFX 写道**
> > ConstMethod对象, 记录着Java方法的不变的描述信息, 包括方法名、方法的访问修饰符、字节码、行号表、局部变量表等等
>
> 就是方法的定义是么?

确实是方法的定义。Class文件里记录的方法层面的信息, 大部分其实都在ConstMethod对象里。

> **General_PF 写道**
> 能详细说说 vtable 和 itable 么?

这个可以看HotSpotInternals wiki：
Virtual Calls
Interface Calls

6 楼 RednaxelaFX 2013-04-16

> **stefmoon 写道**
> 呵呵, 看了下自己的笔记, 发现jdis确实有这功能, 不怎么用就是记不住。。。话说print这个命令还挺智能, 属于见人说人话, 见鬼说鬼话的。。。

不知道是人是鬼就直接扔去disassemble了...orz
5 楼 stefmoon 2013-04-16
> **RednaxelaFX 写道**
> > **stefmoon 写道**
> > CLHSDB没有把二进制逆向生成字节码的功能, 只有反汇编的功能, 有点可惜。。。还有要是能根据ConstMethod对象直接得到字节码就好了~
>
> 其实有的, jdis命令。只不过你要传一个Method地址给它, 而不是直接给bytecode pointer。这也很合理, 因为正确的decode需要constant pool的信息, 随便反汇编一段字节码也没啥用。
>
> 另外使用print命令给一个Method地址进去也可以看到字节码。
>
> 如果你手上有ConstMethod了的话从它找回holder Method就好了。

呵呵, 看了下自己的笔记, 发现jdis确实有这功能, 不怎么用就是记不住。。。话说print这个命令还挺智能, 属于见人说人话, 见鬼说鬼话的。。。
4 楼 RednaxelaFX 2013-04-16
> **stefmoon 写道**
> CLHSDB没有把二进制逆向生成字节码的功能, 只有反汇编的功能, 有点可惜。。。还有要是能根据ConstMethod对象直接得到字节码就好了~

其实有的, jdis命令。只不过你要传一个Method地址给它, 而不是直接给bytecode pointer。这也很合理, 因为正确的decode需要constant pool的信息, 随便反汇编一段字节码也没啥用。

另外使用print命令给一个Method地址进去也可以看到字节码。

如果你手上有ConstMethod了的话从它找回holder Method就好了。
3 楼 feng_jyie 2013-04-16
很好, 不错的文章
2 楼 stefmoon 2013-04-16
CLHSDB没有把二进制逆向生成字节码的功能, 只有反汇编的功能, 有点可惜。。。还有要是能根据ConstMethod对象直接得到字节码就好了~
1 楼 General_PF 2013-04-16
ConstantPool/ConstantPoolCache对象。每个InstanceKlass关联着一个ConstantPool, 作为该类型的运行时常量池。

这个具体指什么?我看到你的回帖里面贴的图, 既有 fieldref, 也有 methodref。为什么叫做 constantPool?

ConstMethod对象, 记录着Java方法的不变的描述信息, 包括方法名、方法的访问修饰符、字节码、行号表、局部变量表等等

就是方法的定义是么?

能详细说说 vtable 和 itable 么?

谢谢拉。