http://hllvm.group.iteye.com/group/topic/35798

新生代回收调试的一些心得

jianglei3000 2013-03-03

一、废话

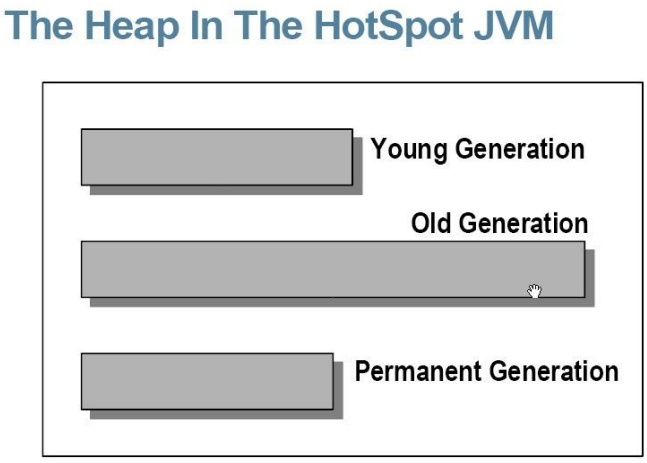
应大神RednaxelaFX要求写下自己调试JVM GC的一些心得体会。
我工作本身不是JVM/GC, 但在08年开始要处理外场一些java/jvm crash, 先拿着不完全一致的开源代码对照反汇编逆向分析, 开始对jvm有些兴趣, 也开始自己动手编译调试分析jvm的解释执行/gc/热点编译, 虽然远远谈不上熟悉, 更没有吃透, 但也学习到很多东西, 似乎比学习windows内核还多些收获。
三年前曾经有个初步心得总结, 但环境都变化了, 这次重新整理一把, 温故而知新。
本来打算元宵前就开始弄, 结果忙于windows上一个开发32位到64位移植, 耽误了不少时间, 所以赶得比较匆忙, 而且水平有限, 有些也不细致, 有些可能不正确, 大家尽量指点, 我也好有进步。
这里先把新生代回收的调试写下, 老生代用的cms, 其实cms更有趣, 相对要复杂得多, 在入门的时候看了Poonam写的一篇博客《Understanding CMS GC Logs》受益匪浅, 这个比较耗费时间, 如果有空打算下个月中旬前完成。

二、环境

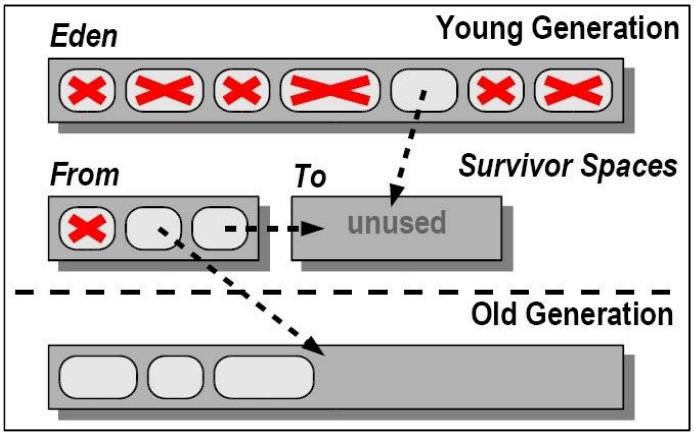
编译调试jvm版本是sun的jdk6 hotspot 6u20版本
我在windows下只编译了jvm, 其实在solaris下也如此, 如果要编译整个jdk有些组件找不到。
jdk采用的是java version "1.6.0_11", 两个版本不一致, 对调试gc关系不大哈。
jvm在windows xpsp3下用vc2003编译, 虽然我在solaris下用sun的编译器和mdb调试（gcc似乎编译不过）, 不过, 感觉还是在windows下调试更方便些。

三、几个术语和概念

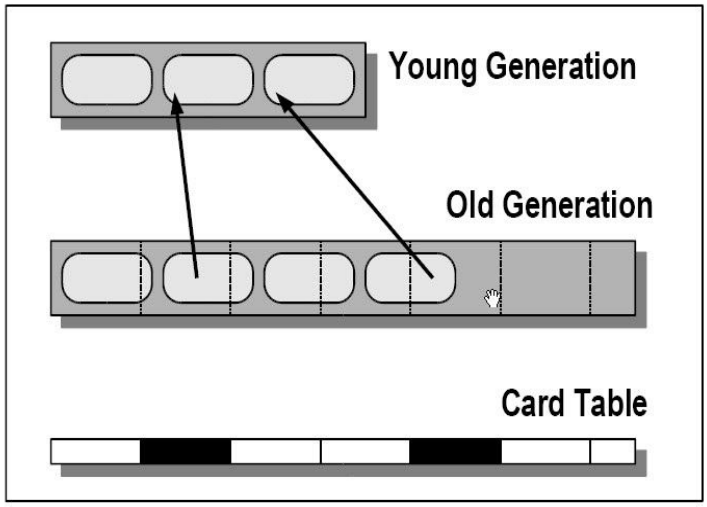
堆分为新生代、老生代和perm代。



新生代又由eden和from/to区三个部分组成。



注意, 还有个东东虽然不是java堆内存里面, 但辅助内存分配回收, 也很重要, 它就是card table, 后续会说到。



注意, 以上几个图都是以前从sun的公开文档下载来的, 后续图都是原创。

四、准备工作

1、运行参数如下

./java -server -Xmx600m -Xms600m -XX:PermSize=64M -XX:MaxPermSize=128M -XX:NewSize=128m
-XX:MaxNewSize=128m -XX:SurvivorRatio=8 -XX:-UseTLAB -XX:+UseSerialGC
-XX:CMSInitiatingOccupancyFraction=50 -XX:+PrintGCDetails -Xloggc:gc.log test.MyHelloM

其中参数说明：
-XX:-UseTLAB是为了更好调试new如何分配的；
-XX:+UseSerialGC为了调试最简单的回收, 其实一般来说新生代不够大, 用ParNew足够了,而它除了多线程外, 原理和DefNew一样, 而后者显然更好调试；
-XX:+PrintGCDetails 这个为了更好看回收日志；
新生代和老生代以及perm的最大最小都一致, 这个不仅更好调试, 而且即使在实际应用中, 如果内存足够, 也推荐这样, 免得反复回收频繁。

2、进程地址空间

注意, 在我的环境中, 在进程的地址空间中, eden/from/to/tenured/perm的实际地址如下gclog最后部分

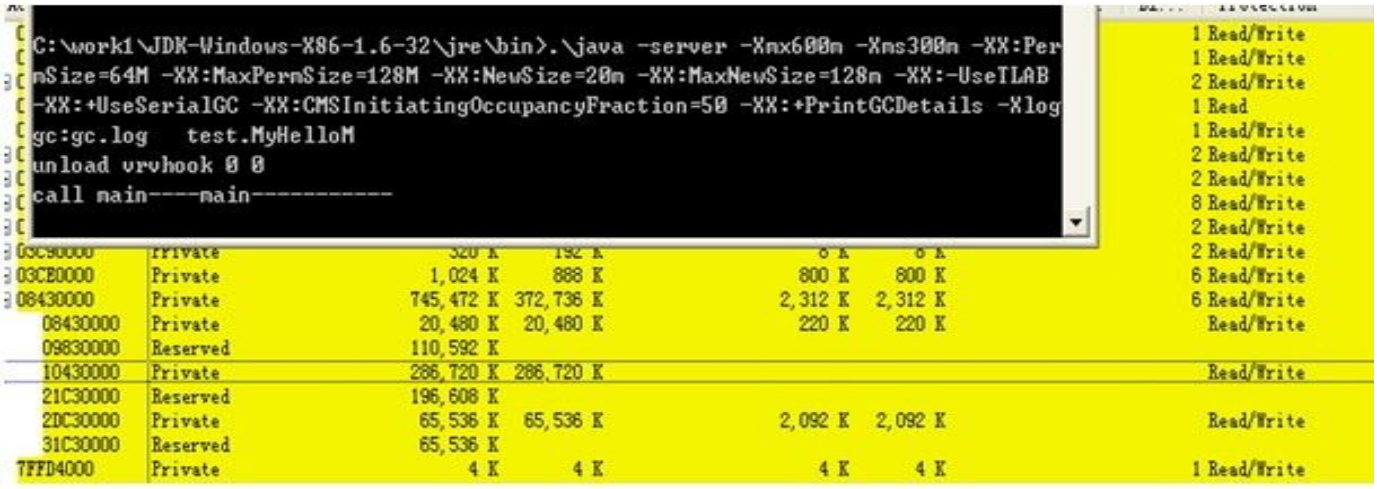
```
 def new generation   total 118016K, used 218K [0x08430000, 0x10430000, 0x10430000)
  eden space 104960K,   0% used [0x08430000, 0x08466b68, 0x0eab0000)
  from space 13056K,   0% used [0x0eab0000, 0x0eab0000, 0x0f770000)
  to   space 13056K,   0% used [0x0f770000, 0x0f770000, 0x10430000)
 tenured generation   total 483328K, used 0K [0x10430000, 0x2dc30000, 0x2dc30000)
   the space 483328K,   0% used [0x10430000, 0x10430000, 0x10430200, 0x2dc30000)
 compacting perm gen  total 65536K, used 2112K [0x2dc30000, 0x31c30000, 0x35c30000)
   the space 65536K,   3% used [0x2dc30000, 0x2de40218, 0x2de40400, 0x31c30000)
```

大致整理下各个区的开始地址：

eden	from	to	tenured	perm
0x08430000	0x0eab0000	0x0f770000	0x10430000	0x2dc30000

以后调试的时候可以很容易在内存中看到相关对象在那个区。

ps: 也可以把把各个代的最小和最大设置成不一样, 然后用vmmap看很容易看到, 如下图



3、java代码

Java代码

```
1.  package test;
2.  import java.util.Vector;
3.
4.  public class MyHelloM {
5.     public static Vector ve ;
6.     private int _count;
7.     private String _name;
8.     MyHelloM(int count){
9.        _count = count;
10.
11.    }
12.    public void setName(String name){
13.       _name = name;
14.    }
15.    static {
16.
17.    }
18.
19.    public static void main(
20.          String[] arg) {
21.
22.       int i  =0;
23.       while(true){
24.          try {
25.             if (i == 0){
26.                System.gc();
27.                ve = new Vector(10);
28.                Thread.sleep(1* 1 * 1000);
29.                MyHelloM my1 = new MyHelloM(i);
30.                String testString1 = new String("testo");
31.                my1.setName(testString1);
32.                ve.add(my1);
33.             }
34.             if (i == 1){
35.                MyHelloM my2 = new MyHelloM(i);
36.                String testString2 = new String("test1");
```

```
37.                 my2.setName(testString2);
38.                 ve.add(my2);
39.             }
40.
41.             if (i == 2){
42.                 System.out.println("before sleep i = 2\n");
43.                 Thread.sleep(1* 30 * 1000);
44.                 Byte []bb = new Byte[100 * 1024 * 1024];
45.                 ve.add(bb);
46.                 System.out.println("after sleep i = 2\n");
47.             }
48.             if (i ==  3){
49.                 Thread.sleep(1* 1 * 1000);
50.                 ve.remove(2);
51.                 ve.remove(1);
52.                 System.out.println("after sleep i = 3\n");
53.             }
54.             if (i > 3 ){
55.                 System.out.println("before sleep i = " + i + " ...\n");
56.                 if (i == 6) Thread.sleep(1* 40 * 1000);
57.                 Byte []bb = new Byte[10 * 1024 * 1024];
58.                 Thread.sleep(1* 20 * 1000);
59.                 System.out.println("after sleep i = " + i + " ...\n");
60.             }
61.             i++;
62.
63.         } catch (InterruptedException e) {
64.             // TODO Auto-generated catch block
65.             e.printStackTrace();
66.         }
67.     }
68.  }
69. }
```

代码说明：
Vector ve是static变量；
循环第4次的时候, i=3, 会把my2从ve中remove, 但my1仍然在ve中；
循环第7次的时候, i=6, new Byte[10 * 1024 * 1024]触发新生代回收；
加上Thread.sleep, 是为了方便sleep的时候打jmap, 看到各个对象的引用；
如果不打算用mat/jmap来看, 非要自己调试各个对象怎么new出来的, 跟踪每个bytecode的解释执行太累, 注意每次new不一定都会到InterpreterRuntime::_new这里, 所以在JVM_Sleep完毕的JVM_END处下断点很容易跟踪到下一个bytecode的解释执行那里, 也可以在new/newarry的解释执行开始那里设置断点, 不过每次跟踪还是有些累。
再次啰嗦下, 为什么不在bytecode new那里设置断点呢？
比如在下面这里bb这个的解释那里设置断点
```
00982884  mov      edx,dword ptr [esi+1] //__ get_unsigned_2_byte_index_at_bcp(rdx, 1)
00982887  bswap    edx
00982889  shr      edx,10h
```
是因为中间可能会有很多其它不关心的对象产生, 所以还是建议在JVM_Sleep完毕的JVM_END处下断点。

4、修改下jvm的代码

为了代码容易测试, 在ClassFileParser.cpp
ClassFileParser::parseClassFile函数开头地方加上如下代码

```
 //added by jl
 char *tmp = name->as_utf8();
 if ( strstr(tmp, "test/MyHello") > 0)  {
    printf("test--parseClassFile class %s\n", tmp);
        //_has_print_constpool = true;
 }
 //added end
```

在JavaCalls.cpp
JavaCalls::call_helper加上如下代码

```
 char * tmp = method->name()->as_C_string();
 if (strstr(tmp, "main") > 0) //main或者其它想要调试的函数名字都可以
 {
       printf("call myfunction----%s----------\n", tmp);
 }
```

这两处代码加上就是为了方便容易设断点：
parseClassFile那里修改后, 可以方便下断点看到自己的class的constpool和各个东东（比如cp/klassOop/methods）以及放到perm的位置；
JavaCalls::call_helper那里修改后, 可以方便调试自己的函数的bytecode的执行。
注意, 这些代码修改并不影响任何逻辑, 只是方便调试。

5、其它工具

除了vc2003和jdk自代工具外（jmap/jstack/jconsole这些）, 需要vmmap/MemoryAnalyzer, 后者简称mat, 特别重要, 完全可以免除很多费神的手工操作, 比如跟踪每个对象的new/newarray, 很烦, 我在sleep某个时刻, 自己jmap打出来, 然后用mat一清二楚。
准备工作差不多了, 可以开始动手了。

jianglei3000 2013-03-03

五、调试

1、回收前的对象

当i=6的时候, if (i == 6) Thread.sleep(1* 40 * 1000)
这个时候执行jmap, 然后sleep后继续执行下一条
Byte []bb = new Byte[10 * 1024 * 1024];
将触发gc, 还没有触发的时候, 这个时候mat看到如下

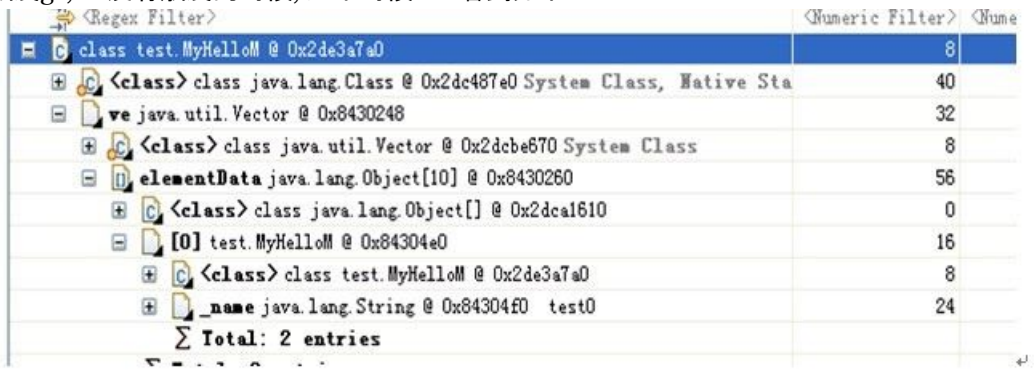
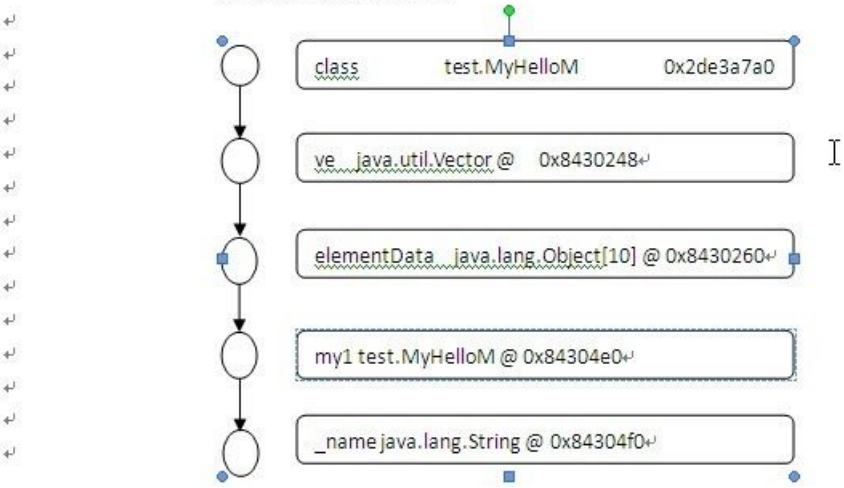
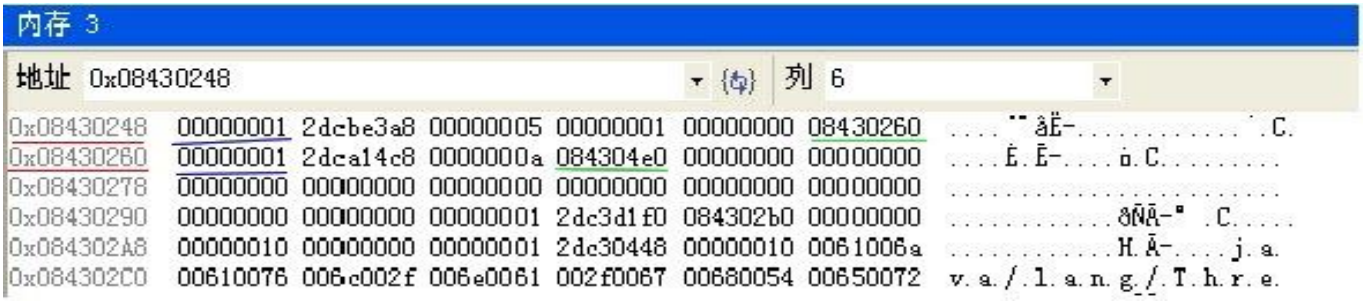


图 5 mat 看到的对象引用



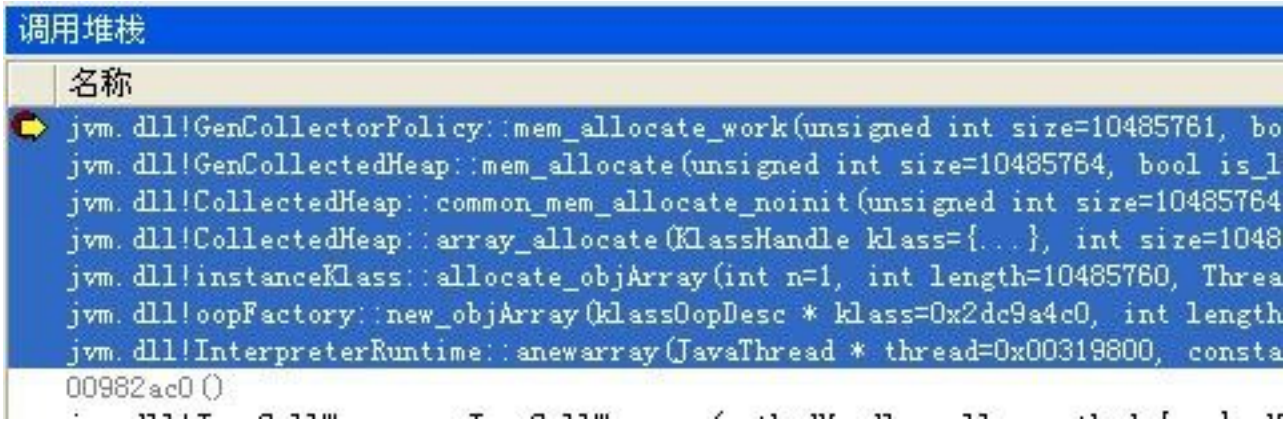
这个时候, vc看到ve的地址, 内存如下



在图中,
红色线那里是对象地址；
蓝色线那里是生命值, 每次新生代copy到survivor后, 它就会计算一次, 如果达到一定条件, 比如超过MaxTenuringThreshold, 就提升到老生代；
绿色线那里是引用, 如果都回收后, 那里需要更新的。

2、触发回收

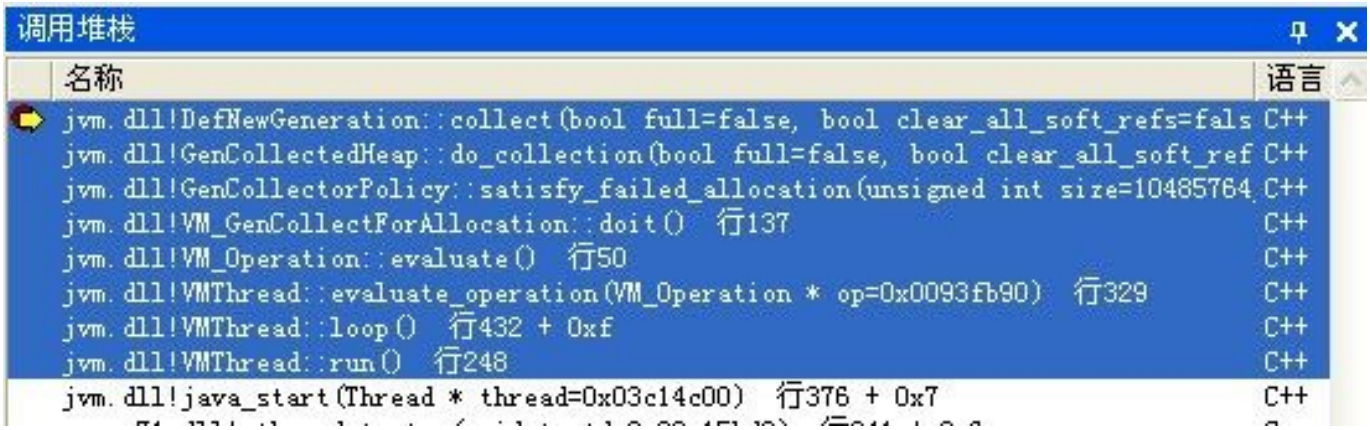
当i=6的时候, 再次new Byte[10 * 1024 * 1024], eden空间不足了, 将触发新生代gc回收。
堆栈如下



在mem_allocate_work中, 发现新生代不足, 创建一个VM_GenCollectForAllocation op, 然后VMThread::execute(&op), 即把这个任务放到VMThread里面, 后者会调用gc来处理这次任务。

3、开始回收

VMThread处理回收, 由于指定参数-XX:+UseSerialGC, 所以会调用函数DefNewGeneration::collect, 堆栈如下。



在这个函数中, 本java代码着重关注两个地方

```
 gch->gen_process_strong_roots(_level,
                               true,  // Process younger gens, if any,
                                      // as strong roots.
                               true,  // activate StrongRootsScope
                               false, // not collecting perm generation.
```

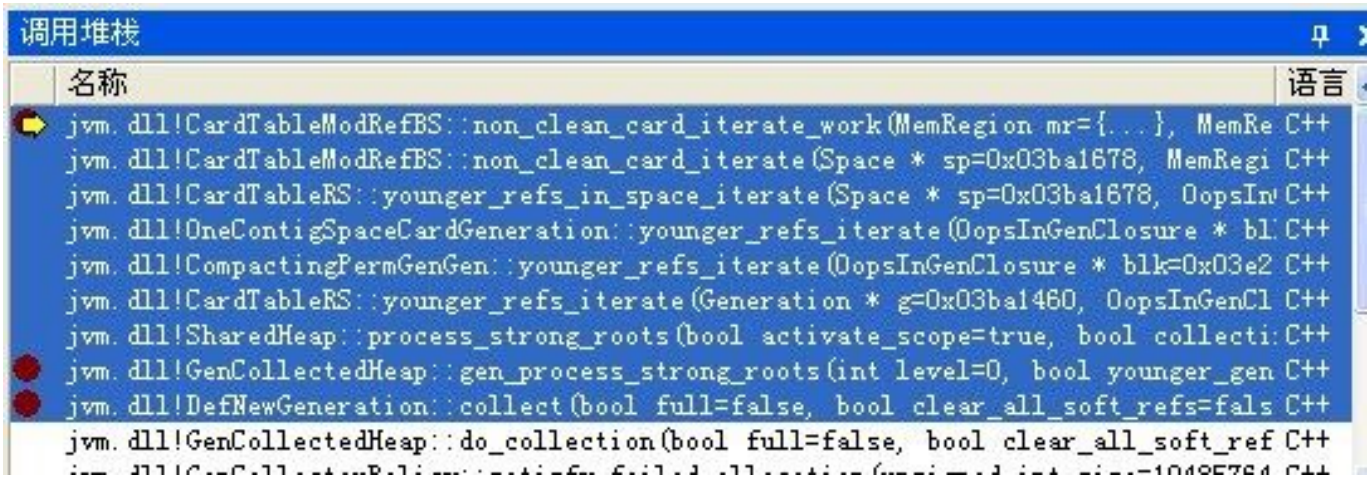
```
                SharedHeap::SO_AllClasses,
                &fsc_with_no_gc_barrier,
                true,   // walk *all* scavengable nmethods
                &fsc_with_gc_barrier);

 // "evacuate followers".
 evacuate_followers.do_void();
```

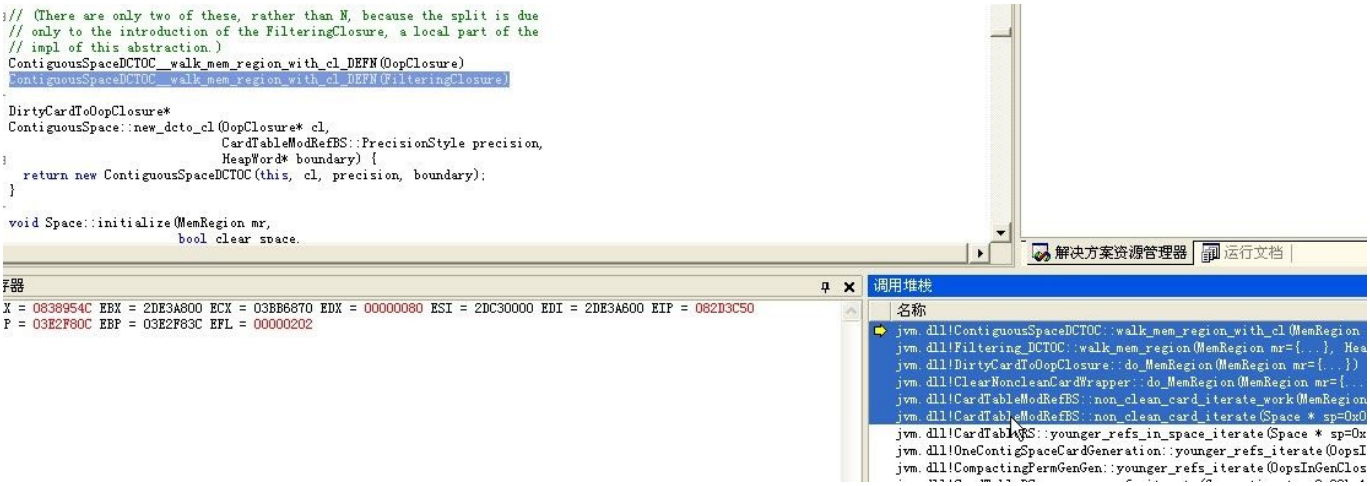
第一处, 处理gen_process_strong_roots将找到并处理Vector ve,
第二处 evacuate_followers.do_void将处理后续引用。

4、gen_process_strong_roots

从图5图6来看, 即使是新生代回收, 也必须先找到test.MyHelloM, 而不是先直接在新生代去找ve, 而前者是在perm那里, 所以必须从这里开始找, 而perm那么大, 是一个一个枚举么, 当然不是, 这个需要借助图3那里看到的Card Table。



这下理解Card Table的作用了吧。
在函数CardTableModRefBS::non_clean_card_iterate_work中, 找到相关bottom和top（其中bottom就是在walk_mem_region得到bottom_obj）,调用到walk_mem_region_with_cl



jianglei3000 2013-03-03
来到一个宏
ContiguousSpaceDCTOC__walk_mem_region_with_cl_DEFN(FilteringClosure)
在代码space.cpp中能够看到这个宏, 为了更详细的分析, 我这里把它和汇编代码列出来

Cpp代码

```
1.  // We must replicate this so that the static type of "FilteringClosure"
2.  // (see above) is apparent at the oop_iterate calls.
3.  #define ContiguousSpaceDCTOC__walk_mem_region_with_cl_DEFN(ClosureType) \
4.  void ContiguousSpaceDCTOC::walk_mem_region_with_cl(MemRegion mr,       \
5.                                                    HeapWord* bottom,   \
```

```
6.                                      HeapWord* top,      \
7.                                      ClosureType* cl) {  \
8.    bottom += oop(bottom)->oop_iterate(cl, mr);                \
9.    if (bottom < top) {                                        \
10.     HeapWord* next_obj = bottom + oop(bottom)->size();       \
11.     while (next_obj < top) {                                 \
12.       /* Bottom lies entirely below top, so we can call the */  \
13.       /* non-memRegion version of oop_iterate below. */         \
14.       oop(bottom)->oop_iterate(cl);                          \
15.       bottom = next_obj;                                     \
16.       next_obj = bottom + oop(bottom)->size();               \
17.     }                                                        \
18.     /* Last object. */                                       \
19.     oop(bottom)->oop_iterate(cl, mr);                        \
20.   }                                                          \
21. }
22.
23. // (There are only two of these, rather than N, because the split is due
24. // only to the introduction of the FilteringClosure, a local part of the
25. // impl of this abstraction.)
26. ContiguousSpaceDCTOC__walk_mem_region_with_cl_DEFN(OopClosure)
27. ContiguousSpaceDCTOC__walk_mem_region_with_cl_DEFN(FilteringClosure)
```

汇编代码如下

```
30 082D3C50  push        ebp
31 082D3C51  mov         ebp,esp
32 082D3C53  mov         edx,dword ptr [ebp+8]   //EDX = 2DE3A600 mr.start
33 082D3C56  push        ebx
34 082D3C57  mov         ebx,dword ptr [ebp+18h] //EBX = 03E2F828
35 082D3C5A  push        esi
36 082D3C5B  mov         esi,dword ptr [ebp+10h] //bottom ESI = 2DE3A5F0
37 082D3C5E  mov         eax,dword ptr [esi+4]   //EAX = 2DC30DA0
38 082D3C61  push        edi
39 082D3C62  sub         esp,8
40 082D3C65  lea         ecx,[eax+8]             //ECX = 2DC30DA8
41 082D3C68  mov         eax,esp
42 082D3C6A  mov         dword ptr [eax],edx
43 082D3C6C  mov         edx,dword ptr [ebp+0Ch]
44 082D3C6F  push        ebx
45 082D3C70  mov         dword ptr [eax+4],edx
46 082D3C73  mov         eax,dword ptr [ecx]
47 082D3C75  push        esi
48 082D3C76  call        dword ptr [eax+174h]  //> jvm.dll!Klass::oop_oop_iterate_nv_m(oopDesc * obj=0x2de3a5f0, FilterAndMark
49 082D3C7C  lea         edi,[esi+eax*4]       //得出当前oop obj ESI = 2DE3A5F0 EAX = 00000014  EDI = 2DE3A640
50 082D3C7F  cmp         edi,dword ptr [top]
51 082D3C82  jae         ContiguousSpaceDCTOC::walk_mem_region_with_cl+0C1h (82D3D11h)
52 082D3C88  mov         ecx,edi
53 082D3C8A  call        oopDesc::size (803D0A0h)
54 082D3C8F  lea         esi,[edi+eax*4]     //关键点！！！HeapWord* next_obj = bottom + oop(bottom)->size();
55                                           //EAX = 00000058 EDI = 2DE3A640  ESI = 2DE3A7A0 class test.MyHelloM @ 0x2de3a7a0
56 082D3C92  cmp         esi,dword ptr [top] //top 0x2de3a800
57 082D3C95  jae         ContiguousSpaceDCTOC::walk_mem_region_with_cl+0A1h (82D3CF1h)
58 082D3C97  mov         eax,dword ptr [edi+4] //oop(bottom)->oop_iterate(cl);   EAX = 2DC30DA0 EDI = 2DE3C410
59 082D3C9A  mov         edx,dword ptr [eax+8] //EDX = 08399FD0
60 082D3C9D  lea         ecx,[eax+8]           //ECX = 2DC30170
61 082D3CA0  push        ebx                   //EBX = 03E2FA28
62 082D3CA1  push        edi                   //EDI = 2DE3A640
63 082D3CA2  call        dword ptr [edx+124h]  //oop_oop_iterate_v(oop obj, OopClosure* blk) //+ obj 0x2de3c410 (_mark=0x00000
```

汇编有三个关键点
第一个, 082D3C76 call   dword ptr [eax+174h]
它得到当前oop obj
第二个, 082D3C8F lea   esi,[edi+eax*4]
//关键点！！！HeapWord* next_obj = bottom + oop(bottom)->size();
ESI = 2DE3A7A0
这个ESI 正好是test.MyHelloM的klass obj啊！

第三个, 082D3CA2 call   dword ptr [edx+124h]
会jmp到instanceKlassKlass::oop_oop_iterate, 调用iterate_static_fields,
0817FF8B lea   esi,[ecx+eax*4+128h]   //ESI = 2DE3A790
得到引用0x2DE3A790,

终于找到了Vector ve。其实迭代找到了test.MyHelloM的时候就会走过来。

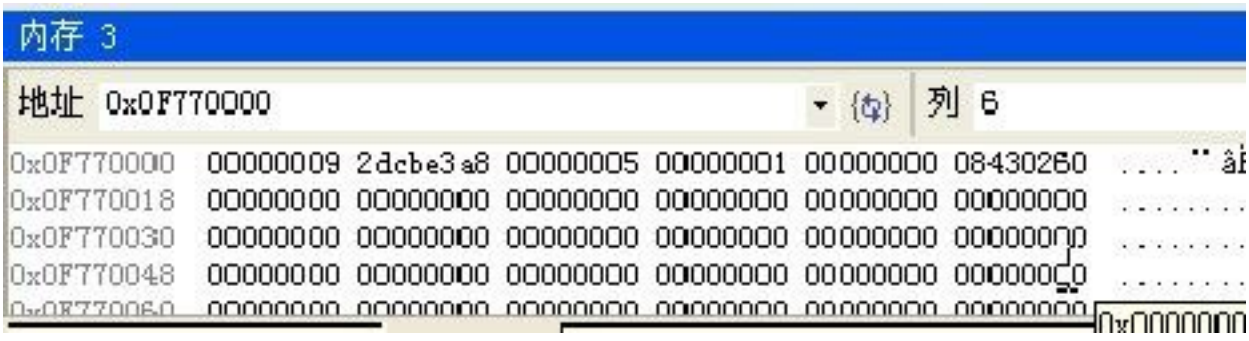
持续跟踪会走到FilteringClosure::do_oop_work(p), p=0x2DE3A790

Cpp代码

```
1.  template <class T> inline void FastScanClosure::do_oop_work(T* p) {
2.    T heap_oop = oopDesc::load_heap_oop(p); //heap_oop = EAX = 08430248
3.    // Should we copy the obj?
4.    if (!oopDesc::is_null(heap_oop)) {
5.      oop obj = oopDesc::decode_heap_oop_not_null(heap_oop);
6.      if ((HeapWord*)obj < _boundary) {
7.        assert(!_g->to()->is_in_reserved(obj), "Scanning field twice?");
8.        oop new_obj = obj->is_forwarded() ? obj->forwardee()
9.                                          : _g->copy_to_survivor_space(obj);
10.       oopDesc::encode_store_heap_oop_not_null(p, new_obj);
11.       if (_gc_barrier) {
12.         // Now call parent closure
13.         do_barrier(p);
14.       }
15.     }
16.   }
17. }
```

看到里面的copy_to_survivor_space了吧, 这个一目了然。
现在, to区就是survivor_space, 下一次新生代回收, from区就是survivor_space了。

这次实际效果是把ve这个obj copy到了to区, 这个时候to区什么都没有, 所以它刚好占第一个位置, 很方便以后调试它引用的Vector的Object数组。



它就是从0x8430248copy到0x0F770000, 注意第一个从1变成9了, 表示age增加了一。
它引用的0x8430260都没有变化, 还没有轮到它gc。
而Class test.MyHelloM对ve的引用已经变了
0x2DE3A790 0f770000
那么, 它们什么时候回收或者copy到to来呢？

jianglei3000 2013-03-03
5、回收子节点到叶子
回收下面的在evacuate_followers.do_void()里面, 里面涉及到多个宏, 其中最重要的是

ALL_SINCE_SAVE_MARKS_CLOSURES(DefNew_SINCE_SAVE_MARKS_DEFN), 参考代码
defNewGeneration.cpp

Cpp代码

```
1.  ALL_SINCE_SAVE_MARKS_CLOSURES(DefNew_SINCE_SAVE_MARKS_DEFN)
2.
3.  0810465C call     ContiguousSpace::oop_since_save_marks_iterate_nv (82D4650h) //eden()->
4.  08104661 mov      ecx,dword ptr [edi+150h]
5.  08104667 push     esi
6.  08104668 call     ContiguousSpace::oop_since_save_marks_iterate_nv (82D4650h) //to()->
7.  0810466D mov      ecx,dword ptr [edi+14Ch]
8.  08104673 push     esi
9.  08104674 call     ContiguousSpace::oop_since_save_marks_iterate_nv (82D4650h) //from()->
```

由于刚才ve已经被copy到了to区, 所以它的子节点以及叶子都在这里被回收。

08104668 call ContiguousSpace::oop_since_save_marks_iterate_nv (82D4650h) //to()->

Cpp代码

```
1.  #define ContigSpace_OOP_SINCE_SAVE_MARKS_DEFN(OopClosureType, nv_suffix)  \
2.                                                  \
3.  void ContiguousSpace::                          \
4.  oop_since_save_marks_iterate##nv_suffix(OopClosureType* blk) {         \
5.    HeapWord* t;                                  \
6.    HeapWord* p = saved_mark_word();              \
7.    assert(p != NULL, "expected saved mark");     \
8.                                                  \
9.    const intx interval = PrefetchScanIntervalInBytes;          \
10.   do {                                          \
11.     t = top();                                  \
12.     while (p < t) {                             \
13.       Prefetch::write(p, interval);             \
14.       debug_only(HeapWord* prev = p);           \
15.       oop m = oop(p);                           \
16.       p += m->oop_iterate(blk);                 \
17.     }                                           \
18.   } while (t < top());                          \
19.                                                 \
20.   set_saved_mark_word(p);                       \
21. }
22.
23. ALL_SINCE_SAVE_MARKS_CLOSURES(ContigSpace_OOP_SINCE_SAVE_MARKS_DEFN)
24. 082D4650 push     ebp
25. 082D4651 mov      ebp,esp
26. 082D4653 push     ecx
27. 082D4654 push     ebx
28. 082D4655 push     esi
29. 082D4656 push     edi
30. 082D4657 mov      edi,ecx
31. 082D4659 mov      eax,dword ptr [edi]
32. 082D465B mov      dword ptr [ebp-4],edi
33. 082D465E call     dword ptr [eax+8]    //t = top() 返回eax
34. 082D4661 mov      ebx,dword ptr [blk]
35. 082D4664 mov      esi,eax          //ESI = 0F770000
36. 082D4666 jmp      ContiguousSpace::oop_since_save_marks_iterate_nv+20h (82D4670h)
37. 082D4668 mov      edi,dword ptr [ebp-4] //循环
38. 082D466B jmp      ContiguousSpace::oop_since_save_marks_iterate_nv+20h (82D4670h)
39. 082D466D lea      ecx,[ecx]
40. 082D4670 mov      edi,dword ptr [edi+34h] //EDI = 0F770018
41. 082D4673 cmp      esi,edi
42. 082D4675 jae      ContiguousSpace::oop_since_save_marks_iterate_nv+3Fh (82D468Fh)
```

```
43. 082D4677  mov       eax,dword ptr [esi+4]
44. 082D467A  mov       edx,dword ptr [eax+8]
45. 082D467D  lea       ecx,[eax+8]
46. 082D4680  push      ebx             //EBX = 03E2FC8C 下面函数参数 OopClosureType* closure
47. 082D4681  push      esi             //ESI = 0F770000 下面函数参数 oop obj
48. 082D4682  call      dword ptr [edx+128h] //instanceKlass::oop_oop_iterate##nv_suffix##_m
49. 082D4688  lea       esi,[esi+eax*4]
50. 082D468B  cmp       esi,edi
```

注意, 汇编代码里面的注释只是第一次的, 第一次从ve找到Object[10]这个数组对象, 并copy到to区call后, 回到这个循环, 从Object[10]第二次找到my1,以此类推, 这里都copy到了to区, 这里比较啰嗦, 但相对流程比较简单, 就不再赘述了。

jianglei3000 2013-03-03
6、最后
所以有引用关系的都被copy或者提升了, 那么没有引用的是怎么被回收的呢？
这个在新生代其实很简单, 就直接清零内存了。

fh63045 2013-05-03
佩服 ,高端 ,可有编译自己的JDK教程

RednaxelaFX 2013-05-03

> **fh63045 写道**
> 佩服 ,高端 ,可有编译自己的JDK教程

请从这帖开始参考：http://rednaxelafx.iteye.com/blog/1549577
另外为了编译顺利, 请尽量用最新的OpenJDK（例如现在请用OpenJDK 7u）

iminto 2013-05-28
很不错, 启发很大

RednaxelaFX 2014-02-11
在http://hllvm.group.iteye.com/group/topic/39376/回帖的时候想起这边已经有个这么好的帖了。得把这两帖关联起来～