https://hllvm-group.iteye.com/group/topic/40491

HotSpot 解释器是怎样执行bytecode 的

cheney_love 2014-07-06

一段简单的代码：

Java代码

```
1. public class Tiger {
2.     public static void main(String args[]){
3.         Tiger tiger = new Tiger();
4.     }
5. }
```

main 方法对应的bytecode

Java代码

```
1.  省略...
2.  Code:
3.    stack=2, locals=2, args_size=1
4.       0: new           #1                  // class Tiger
5.       3: dup
6.       4: invokespecial #16                 // Method "<init>":()V
7.       7: astore_1
8.       8: return
9.    LineNumberTable:
10.     line 5: 0
11.     line 6: 8
12.   LocalVariableTable:
13. 省略...
```

在圈子里面看了几篇R大回复的帖子：

java_main的汇编入口在哪里

Java 字节码如何执行的

对方法的执行过程有了一定的了解。

在不使用-Xcomp的情况下，虚拟机是使用默认的解释器进行代码的执行工作。

方法的调用方式：

Java代码

```
1. StubRoutines::call_stub()(
2.         (address)&link,
3.         // (intptr_t*)&(result->_value), // see NOTE above (compiler problem)
4.         result_val_address,          // see NOTE above (compiler problem)
5.         result_type,
6.         method(),
7.         entry_point,
8.         args->parameters(),
9.         args->size_of_parameters(),
10.        CHECK
11.      );
```

call_stub() 是在jvm初始化的时候在StubGenerator::generate_call_stub()中生成的。

entry_point ： 指向的是解释器的方法入口处理函数。（这个入口函数是哪个函数？）

method() ： 是要执行的方法。

方法对应的 bytecode 应该是存在methodoop的_code 字段里面。

我现在弄不明白的地方是。

1.在执行call_stub() 后，bytecode 是怎么一个一个被解释然后执行的呢，对应的代码在哪里呢。比如这个new 指令，是在哪里被解析出来然后执行的呢，在TemplateTable 有_new() 这个方法，它是在什么地方被调用的呢？

2.有帖子说是interp_masm_sparc.cpp的dispatch_next，但是我debug的时候，这个方法是在call_stub（） 之前就执行完了。

nijiaben 2014-07-06
每条字节码都对应一段汇编代码，可参考TemplateInterpreterGenerator::generate_and_dispatch方法，顾名思义就是生成当前指令的汇编代码以及跳转，下一个字节码指针(byecode pointer)存在r13寄存器里，当要这条指令本身就要跳转（这个在指令实现里（一般在templateTable_x86_32/64.cpp里实现）就做了），或者执行完这条指令自动切换到下一条(就在上面这个方法里做，根据指令的大小，自动递增r13里的值，能在下一次环节自动跳转过去执行

至于怎么找到这条字节码的汇编代码段，就依靠Interpreter的一个映射表Interpreter::_active_table了，在上面的跳转代码里其实会用到它，这个表可能是_normal_table也可能是safept_table，取决于是否要进入安全点(比如要进行stw性质的gc的时候)，对于_normal_table的设置在TemplateInterpreterGenerator::set_entry_points这个函数里

cheney_love 2014-07-06
nijiaben 写道
每条字节码都对应一段汇编代码，可参考TemplateInterpreterGenerator::generate_and_dispatch方法，顾名思义就是生成当前指令的汇编代码以及跳转，下一个字节码指针(byecode pointer)存在r13寄存器里，当要这条指令本身就要跳转（这个在指令实现里（一般在templateTable_x86_32/64.cpp里实现）就做了），或者执行完这条指令自动切换到下一条(就在上面这个方法里做，根据指令的大小，自动递增r13里的值，能在下一次环节自动跳转过去执行. 至于怎么找到这条字节码的汇编代码段，就依靠Interpreter的一个映射表Interpreter::_active_table了，在上面的跳转代码里其实会用到它，这个表可能是_normal_table也可能是safept_table，取决于是否要进入安全点(比如要进行stw性质的gc的时候)，对于_normal_table的设置在TemplateInterpreterGenerator::set_entry_points这个函数里

谢谢您的回复，请问一下，generate_and_dispatch是被谁调用的呢，好像在call_stub 之前就执行完了。call_stub 和 generate_and_dispatch 是怎么串起来的呢。
谢谢
jiaben 2014-07-06
> cheney_love 写道
> 谢谢您的回复，请问一下，generate_and_dispatch是被谁调用的呢，好像在call_stub 之前就执行完了。call_stub 和 generate_and_dispatch 是怎么串起来的呢。谢谢

generate_and_dispatch在启动过程中就会执行，可理解为为每种字节码指令分别创建一个方法，在运行期根据对应的指令在执行到对应的方法里。

call_stub是在真正的方法调用过程中，传到call_stub中的entry_point对于正常的方法调用而言，一般都是zerolocals或者zerolocals_synchronized对应的entry入口，具体是哪个是根据你的方法类型而言的，比如是否方法加了同步，是否是native方法等，entry创建过程请看TemplateInterpreterGenerator::generate_all方法，每种方法类型的entry生成在AbstractInterpreterGenerator::generate_method_entry这个方法里，其实也就是生成一段汇编代码，将这段汇编的开始地址设置为对应的entry入口，找到入口地址之后就开始执行一系列汇编指令，最终会看到调用到InterpreterMacroAssembler::dispatch_next分发到方法里的第一条字节码指令，而执行这条指令的入口依赖于Interpreter::_active_table，在指令执行过程中依次寻找下一条指令进行分发处理.

ZHH2009 2014-07-09
这类问题确实有好几贴了，
要完全说清解释器是怎样执行bytecode的，我预想可能需要超长的篇幅，
不过最好还是先自己把代码跑起来，然后慢慢debug去分析代码。
理解这问题，需要先有很好的汇编语言基础，
然后抓住HotSpot中的一些实现关键点:
1.call_stub和method_entry_point(zerolocals、zerolocals_synchronized等)都是用汇编实现的；
2.每一个bytecode也都是由一段汇编代码来实现；
1和2中的汇编代码都是在初始化HotSpot阶段就生成的，你可以简单认为它们就是一些内置的方法，
call_stub调用method_entry_point，然后在method_entry_point里调用new，new再调用dup（用跳转更合适）……
这时可以动用GDB或VS从call_stub开始调试每条汇编,call_stub如何衔接到method_entry_point再到new,这里就是考验你的汇编能力的时候了，
有些寄存器是关键点: esp, ebp, ebx, esi……
好比ebx里放的是main(String args[])这个java方法在HotSpot中的method指针,通过ebx就能知道method的其他字段(例如ConstMethod),而通过esi能遍历ConstMethod中的bytecode……
不同的HotSpot版本的实现会存在一些差异，上面只是提了其中的一些关键点。

ZHH2009 2014-07-13
我的OpenJDK-Research上面有一些相关的研究笔记可能会对你有一些帮助
call_stub、method_entry_point_zerolocals都有
https://github.com/codefollower/OpenJDK-Research/tree/master/hotspot/my-docs/interpreter
以后会慢慢补充。

cheney_love 2014-07-15
谢谢您的研究笔记，内容很丰富，已经fork，慢慢学习，我现在还有个疑问，就是解析器在解析new 指令的时候，class 文件的解析是在哪里完成的呢。是在new 指令对应的那段汇编里面吗。现在正在看《深入理解计算机系统》恶补一下汇编。谢谢
ZHH2009 2014-07-15
在研究HotSpot的代码时，可以按顺序围绕下面５个大问题去探索：
1. HotSpot内部的各个模块是如何初始化的？
2. 类的装载、链接、初始化在HotSpot中是如何完成的？
3. HotSpot如何执行java方法中的字节码?
4. HotSpot如何实现GC?
5. HotSpot如何实现JIT编译器?

你问的其实是第３个问题，前面说了要回答好这个问题是需要很多篇幅的，回答它之前必须讲清楚前两个问题，比如在第一个问题中不讲清楚解释器模块、stub模块如何初始化你就不懂call_stub和字节码对应的的汇编是如何生成的；
同样需要在第二个问题中理清类的装载、链接，你才知道class 文件的解析是如何做的，才知道原始的字节码放在哪里？
所以，真想要研究HotSpot，就得循序渐进一步步来，回答好前面３个问题就足够写一本600页以上的书了。
class 文件的解析在src\share\vm\classfile\classFileParser.cpp实现，
入口是: ClassFileParser::parseClassFile

cheney_love 2014-07-17
1.今天经过不断调试，基本弄明白了new 指令中load class文件的地方了。
templateTable_x86_64.cpp 中的TemplateTable::_new()方法。
创建对象分两种方式：快速分配和慢速分配。
快速分配是针对class 文件已经被解析过的情况，这种情况下，直接从常量池取，然后分配对象空间就可以了。该过程直接通过生成的汇编就完成了，所以调试不到。如果class 文件是第一次加载，那么就直接进入慢速分配。对应的代码：
```
 // slow case
 __ bind(slow_case);
 __ get_constant_pool(c_rarg1);
 __ get_unsigned_2_byte_index_at_bcp(c_rarg2, 1);
 call_VM(rax, CAST_FROM_FN_PTR(address, InterpreterRuntime::_new), c_rarg1, c_rarg2);
 __ verify_oop(rax);
```
首先解析class 文件，然后调用InterpreterRuntime::_new() 对对象进行初始化,这个过程可以调试到。
2. TemplateTable::newarray() 没有快速分配和慢速分配之说，所以每次都调用InterpreterRuntime::newarray() 进行创建，这个过程也可以调试到。
再次谢谢大家的指点。

ZHH2009 2014-07-17
能进到InterpreterRuntime::_new() 里就已经是慢速分配的场景了，并且只会为对象分配内存空间，
但是执行构造函数的java代码是由接下来的那条invokespecial触发的。
汇编代码也可以调的，把汇编代码调顺了会知道更多细节，
比如参数是怎么传给构造函数的，new如何转到invokespecial，
invokespecial如何得到InterpreterRuntime::_new()里生成的oop，
对象的字段如何得到值？
InterpreterRuntime类中那些代码基本上是由汇编代码触发的，
要是把里面的C++代码用汇编来实现会导致汇编代码很长很难维护，所以不太实际。
看懂汇编了，才是真懂了，否则只算是了解了中间的某个过程。
小施_重名后缀 2014-07-21
我觉得楼主的这个问题应该是分为3个部分.一个是 bytecode怎么变为汇编并执行.
然后就是 _new 和 invokespecial 的执行情况.
第一个问题. 就是虚拟机在启动的时候,会给每个bytecode准备好该字节码的汇编代码.

首先是在, TemplateTable::initialize() 里,给每个字节码准备他的生成函数.
类似这样
Java代码

```
def(Bytecodes::_iload_0    , ____|____|____|____, vtos, itos, iload    ,  0       );

def(Bytecodes::_iconst_2   , ____|____|____|____, vtos, itos, iconst   ,  2       );


def(Bytecodes::_imul       , ____|____|____|____, itos, itos, iop2     , mul      );
```

然后再 TemplateInterpreterGenerator::generate_all() -> TemplateInterpreterGenerator::set_entry_points_for_all_bytes()->
set_entry_points()->
set_short_entry_points()-> //这里可能也是wide
generate_and_dispatch()
中,会给每个字节码生成汇编代码.
其中 t->generate(_masm); 就是执行前面的def里指定的函数啦.
__ dispatch_epilog(tos_out, step); 这个就去是跳到下一个bytecode对应的汇编代码
举个例子
Java代码

```
static int zoo(int i)
{
    i*=2;
    return  foo(i);
}
```

i*=2 部分; 的字节码就是
Java代码

```
iload_0
iconst_2
imul
istore_0
```

从iconst_2的汇编代码是这样的:(1,不从iload_0开始是因为这里还有些细节,不过不影响解释原理.2 不同的tos,这里的代码会有点不同) 我这里是 itos. 注意,r13寄存器,存放的就是当前字节码的地址.

Java代码

```
(gdb) x /10i $rip
=> 0x7fffed02588f:   push   %rax
   0x7fffed025890:  mov    $0x2,%eax  //eax = 2;
   0x7fffed025895:  movzbl 0x1(%r13),%ebx // ebx = r13 [1]
   0x7fffed02589a:  inc    %r13 // ++ r13;
   0x7fffed02589d:  movabs $0x7ffff7018ea0,%r10
   0x7fffed0258a7:  jmpq   *(%r10,%rbx,8)
```

进入这段代码的时候
(gdb) p $ebx
$17 = 5 //iconst_2 =5
这条mov$0x2,%eax 可以看桌是 iconst_2的本体. 后面的就是到下条字节码的过程.
执行到 0x7fffed02589a 的时候, 下一条字节码就已经到ebx了

(gdb) p $ebx
$18 = 104 // _imul = 104

其实就是跳到 0x7ffff7018ea0 + rbx * 8 , rbx = r13[1] 就是下一条字节码的值, 8 是因为我用的是64位.

0x7ffff7018ea0 魔数是怎么来的呢? 其实他就是当前tos下各个字节码数组地址了,他是一个数组,数组的每个成员都指向该字节码的汇编代码的入口.

我当前是 itos, 也就是3.
(gdb) p &(TemplateInterpreter::_active_table._table[3])
$40 = (u_char *(*)[256]) 0x7ffff7018ea0

0x7fffed0258a7: jmpq   *(%r10,%rbx,8) 执行这一条以后,他就跳到 imul对应的汇编代码了.

(gdb) x /10i $pc
=> 0x7fffed0292c7: mov   (%rsp),%edx
  0x7fffed0292ca: add $0x8,%rsp
  0x7fffed0292ce: imul   %edx,%eax
  0x7fffed0292d1: movzbl 0x1(%r13),%ebx
  0x7fffed0292d6: inc   %r13
  0x7fffed0292d9: movabs $0x7ffff7018ea0,%r10
  0x7fffed0292e3: jmpq   *(%r10,%rbx,8)

现在是itos,其中栈顶元素是i的值, 大家应该知道, rsp就是当前的栈. 执行过程.

mov   (%rsp),%edx // edx = i;
0x7fffed0292ca: add $0x8,%rsp // 栈的地址加,是弹出成员

之前iconst_2的时候, 赋值 eax =2.

0x7fffed0292ce: imul   %edx,%eax 这里就是做乘法了.结果在eax里.

然后准备跳到下一句.临走前看一眼 ebx,下一个bytecode.

(gdb) p $ebx
$21 = 59   59就是 _istore_0 啦. 跳过去以后

(gdb) x /10i $pc
=> 0x7fffed027c07: mov   %eax,(%r14)
  0x7fffed027c0a: movzbl 0x1(%r13),%ebx
  0x7fffed027c0f: inc   %r13
  0x7fffed027c12: movabs $0x7ffff701b6a0,%r10
  0x7fffed027c1c: jmpq   *(%r10,%rbx,8)

说白了,就是在TemplateInterpreter::_active_table._table[tos] 里跳来跳去.
要吃饭了. _new 和 invokespecia 的执行过程我又空补上.

前面已经提到字节码的汇编生成函数是在 TemplateTable::initialize() 定义的, 查看实现直接来这里看就可以了.
Java代码

```
1.  def(Bytecodes::_new                , ubcp|____|clvm|____, vtos, atos, _new              ,  _          );
```

看 void TemplateTable::_new()这个函数.
一开始 是
Java代码

```
1.  __ get_unsigned_2_byte_index_at_bcp(rdx, 1); //就是取index, 也就是
2.   //new          #1 的那个 1
3.
4.   __ get_cpool_and_tags(rsi, rax);
5.  //获取constant poll 和 class 的tags.
```

现在
rdx = class_index,
rsi = cpoll

rax=tags
还有之前提到过的 r13 = 当前bytecode地址,好像叫bcp?

Java代码

```
__ movptr(rsi, Address(rsi, rdx,
            Address::times_8, sizeof(constantPoolOopDesc)));
//获取 instanceKlass 对象, rsi = rsi + rdx* 8 + sizeof(constantPoolOopDesc)


 __ cmpl(Address(rsi,
                instanceKlass::init_state_offset_in_bytes() +
                sizeof(oopDesc)),
          instanceKlass::fully_initialized);
 __ jcc(Assembler::notEqual, slow_case); // 判断instanceKlass  是否完全初始化,没有就到慢分配.
```

Java代码

```
__ movl(rdx,
         Address(rsi,
                 Klass::layout_helper_offset_in_bytes() + sizeof(oopDesc)));
```

rdx = 对象长度
然后尝试去 tlb分配,如果启用的话.

查看tlb的空间够不够.
Java代码

```
__ movptr(rax, Address(r15_thread, in_bytes(JavaThread::tlab_top_offset())));
 __ lea(rbx, Address(rax, rdx, Address::times_1));
 __ cmpptr(rbx, Address(r15_thread, in_bytes(JavaThread::tlab_end_offset())));
 __ jcc(Assembler::above, allow_shared_alloc ? allocate_shared : slow_case);
```

伪代码就是.
Java代码

```
rax = tlb_top.
rbx = tlb_top + rdx. //rdx 是对象长度

if( rbx > tlb_end)
  goto 慢分配或 shared eden
```

shared eden里的分配

Java代码

```
__ movptr(rax, Address(RtopAddr, 0)); //rax = top.


__ bind(retry);
__ lea(rbx, Address(rax, rdx, Address::times_1));//rbx = top+ 对象长度
__ cmpptr(rbx, Address(RendAddr, 0));
__ jcc(Assembler::above, slow_case); // 如果 top+ 对象长度 超过 end了, 就是没空间了,跳到慢分配.


if (os::is_MP()) {
  __ lock();
}
__ cmpxchgptr(rbx, Address(RtopAddr, 0));//cas的方式来设置值,被其他人修改top,说明在被其他线程在我们分配时分配了个对象,就要重试.

// if someone beat us on the allocation, try again, otherwise continue
```

```
16.  __ jcc(Assembler::notEqual, retry);
```

后面就是把对象的那段地址设成0,还有设置对象头

Java代码

```
1.
2.    __ xorl(rcx, rcx); // rcx = 0
3.      __ shrl(rdx, LogBytesPerLong);  // divide by oopSize to simplify the loop
4.      {
5.        Label loop;
6.        __ bind(loop);
7.        __ movq(Address(rax, rdx, Address::times_8,
8.                        sizeof(oopDesc) - oopSize),
9.                rcx);//
10.       __ decrementl(rdx);//
11.       __ jcc(Assembler::notZero, loop);  //用个循环把对象体设置成0.
12.     }
13.
14.  /后面的代码就是设置对象头
```

简单的说就是获取 cpool里的index,然后后去instancesKlass,获取对象长度,分配空间,对象体清零.

慢分配的方式, 调用的就是 InterpreterRuntime::_new 都是cpp代码,就不用多说了.
之后就是调用构造函数, 看字节码就知道其实就是调用个普通的invokespecial

直接看 TemplateTable::invokespecial

1. prepare_invoke() 函数.获取 index,根据index去const pool里获取 metod的信息,比如名称,签名什么的,保存当前的bytecode的指针位置, 然后解析,运行时链接. 总之获取一个完全准备好的 methodoop之后调用
2 InterpreterMacroAssembler::jump_from_interpreted
  2.1 在prepare_to_jump_from_interpreted 里. 把当前 r13 保存起来.

2.2 获取 method->interpreter_entry() 来跳 解释器方法或jit方法的入口.

在解释执行的时候.其实是调到 InterpreterGenerator::generate_normal_entry 所生成的汇编代码中的.

generate_normal_entry 里.先做检查准备, 例如 generate_stack_overflow_check 检查stack够不够

然后开始按照 _max_locals,和函数参数的个数_size_of_parameters,把本地变量入栈初始化为0.

接下来的generate_fixed_frame 来把栈帧的完整(不是平常说的操作数栈那个,是rsp,rbp这个和c++一样的东西,还有java调用约定的其他信息).另外还要从 constMethod 中把被调用函数的第一个字节码,设置到r13去.constMethod 其实就是从classFileparser解析出来的

最后调用 dispatch_next() 就是开始执行新方法的第一个bytecode对应的汇编代码了.
ZHH2009 2014-07-21

> 小施_重名后缀 写道
> 1. prepare_invoke() 函数.获取 index,根据index去const pool里获取 metod的信息,比如名称,签名什么的,保存当前的bytecode的指针位置, 然后解析,运行时链接. 总之获取一个完全准备好的 methodoop

请问这个index跟javap打印出来的那个invokespecial #XXX是否一样？如果不一样，为什么不一样？

ConstantPoolCache的内存布局又如何理解？在这里如何修改它的值？

细节就是魔鬼，理解一个问题，需要先理解更前面的问题，任何片断性的解释都是不够准确了。

光从HotSpot里的宏汇编是不足以理解问题的，debug起来，左边一个memory框，中间一个真实的汇编代码框，右边一个register框，把堆栈的变化情况一个个绘制下来，这样才有可能真正理解HotSpot里的细节。

HotSpot最好的地方不是在架构,架构设计得非常烂,而是体现在某一些细节之处,字节码的解释执行就是其中之一,尝试着写个java方法,然后完整debug一遍: call_stub -> method_entry_point -> invokeXXX -> put/getXXX -> return

完全理解了这样一条链的前前后后的所有细节,我之前回贴中提的三个大问题就已经理解70%了,HotSpot的解释器之所以还有实用性(有些vm根本就不提供解释器),在我看来确实是里面的汇编代码细节实现得挺精妙的,性能不算太差。

小施_重名后缀 2014-07-23
既然cpool cache的部分遗漏了,我就把他加上吧.

首先就是要确定 cpool cache在哪里, 我们看看栈帧. (frame_x86.hpp)

Java代码

```
1.  // Layout of asm interpreter frame:
2.  //    [expression stack      ] * <- sp
3.  //    [monitors              ]   \
4.  //     ...                                        | monitor block size
5.  //    [monitors              ]   /
6.  //    [monitor block size    ]
7.  //    [byte code index/pointr]                    = bcx()                bcx_offset
8.  //    [pointer to locals     ]                    = locals()             locals_offset
9.  //    [constant pool cache   ]                    = cache()              cache_offset
10. //    [methodData            ]                    = mdp()                mdx_offset
11. //    [methodOop             ]                    = method()             method_offset
12. //    [last sp               ]                    = last_sp()            last_sp_offset
13. //    [old stack pointer     ]                      (sender_sp)
    sender_sp_offset
14. //    [old frame pointer     ]   <- fp            = link()
15. //    [return pc             ]
16. //    [oop temp              ]                      (only for native calls)
17. //    [locals and parameters ]
18. //                                <- sender sp
```

所谓的fp就是 rbp, sp就是 rsp,在图上向上方向,栈增加,地址减小.

看这个函数 resolve_cache_and_index --> get_cache_and_index_at_bcp

首先要取得方法的index,紧跟在invoke后面. 也就是r13,

引用
load_unsigned_short(index, Address(r13, bcp_offset));//

即: index = *(r13+1);

之后就是获取cpool cache. 从栈帧可以看到他在rbp上面5个位置

Java代码

```
1.  movptr(cache, Address(rbp, frame::interpreter_frame_cache_offset * wordSize));
2.
3.  cache = *( rbp + 5*8);
4.
5.  //后面有代码.
6.
7.  shll(index, 2);  index = index *4 . //为啥要莫名的乘以4呢?
```

出来以后

movl(temp, Address(Rcache, index, Address::times_ptr, constantPoolCacheOopDesc::base_offset() + ConstantPoolCacheEntry::indices_offset()));

这个寻址稍微复杂一点.

直接写就是 temp = cache + index * 8 + sizeof(constantPoolCacheOopDesc) + (ConstantPoolCacheEntry*)0 -> _indices

(ConstantPoolCacheEntry*)0 -> _indices 就是 _indices 在 ConstantPoolCacheEntry 中的偏移量.

回想一下上面那个被莫名的乘以4. 整个表达式其实应该写成

cache + sizeof(constantPoolCacheOopDesc) + 原始的index * 32 + _indices 在 ConstantPoolCacheEntry 中的偏移量.

而 sizeof(ConstantPoolCacheEntry) 正好就是32.

如果对内存布局比较敏感,基本上就知道是怎么回事了.

cache + sizeof(constantPoolCacheOopDesc) 定位到 constantPoolCacheOopDesc的后面的地址.

原始的index * sizeof(ConstantPoolCacheEntry) 就是数组的成员的地址了. 再加上 _indices 的偏移量.即可获取 invoke的函数的对应的 _indices的地址 rbp + 5*8 指向的内存区域,应该是这样

[constantPoolCacheOopDesc]
[ConstantPoolCacheEntry][ConstantPoolCacheEntry][ConstantPoolCacheEntry][ConstantPoolCacheEntry]

constantPoolCacheOopDesc的 _length字段,就是说后面跟着多少个entry. 由此可知, 读出来原始的index,应该是从0开始的.而个数就和javap出来的最前面的methodref, fieldref的数量

如果改成c代码,大概就是
C代码

```
1.  constantPoolCacheOopDesc *cache = (constantPoolCacheOopDesc *)( *(long*)( rbp + 5*8));
2.
3.  ConstantPoolCacheEntry *entryArray =  (ConstantPoolCacheEntry *)((char*)cache +
    sizeof(constantPoolCacheOopDesc))
4.
5.  tmp = entryArray[raw_index]._indices
```

后面的代码 就查看标志位,看看是不是已经被解析过了.

引用

```
__ shrl(temp, shift_count);
        // have we resolved this bytecode?
        __ andl(temp, 0xFF);
        __ cmpl(temp, (int) bytecode());
        __ jcc(Assembler::equal, resolved);
```

另外_indices在没解析过的时候,和javap出来的#1 #2是相等的,是从1开始.

ConstantPoolCacheEntry 各个值得详细意义,可以直接查看cpCacheOop.hpp,那上面的注释也比较详细了.

解析的过程在, 入口是 InterpreterRuntime::resolve_invoke, cpp代码,细节就不用提了,随便看看应该就知道个大概了.

resolve完成以后,最后会调用

```
cache_entry(thread)->set_method(
        bytecode,
        info.resolved_method(),
        info.vtable_index())
```

给 ConstantPoolCacheEntry 设置值, 和之前略有不同的就是他通过JavaThread 来定位CacheEntry, 在JavaThread 的lastframe 和前面rbp的功能差不多,过程代码是 method(thread)->constants()->cache()->entry_at(i)

douyu 2014-07-23
楼上的兄弟，你真有把实际生成的汇编代码调试起来吗？

首先，你上面提的那个index = *(r13+1)根本就不是javap打印出来的那个invokespecial中的#XXX,
这个index是在Rewriter阶段重写过的,
ConstantPoolCacheEntry:_indices字段的格式是
Java代码

```
1. // bit number |31                0|
2. // bit length |-8--|-8--|---16----|
3. // --------------------------------
4. // _indices   [ b2 | b1 |  index  ]
```

里面最后16bit的index才是真的#XXX。

其次，上面连续的这5条汇编mov、shr、and、cmp、je也不是与标志位(_flags字段)相关的,
而是取出ConstantPoolCacheEntry::_indices字段中的b1部分，如果b1的值刚好等于invokespecial，就说明解析过了。

最后，为啥要莫名的乘以4呢? 也不是像你解释的那么复杂，甚至是错误的。

在我的32位系统上面实际的汇编是
Java代码

```
1. shl    $0x2,%edx
2. mov    0x8(%ecx,%edx,4),%ebx //这里的4代表ConstantPoolCacheEntry每个字段的字节数
3. shr    $0x10,%ebx
4. and    $0xff,%ebx
5.
6. cmp    $0xb7,%ebx
7. je     0x01cc5897
```

乘以4,是因为每个ConstantPoolCacheEntry刚好有4个字段，每个字段占用的字节数刚好又一样，都是4,
加8是因为ConstantPoolCache类的_length和_constant_pool占了8个字节,

所以对于第0个ConstantPoolCacheEntry::_indices字段的地址就是:
ConstantPoolCache的地址 + 8 + (0*4)*4,
其实就是: ConstantPoolCache的地址 + 8

第1个ConstantPoolCacheEntry::_indices字段的地址就是:
ConstantPoolCache的地址 + 8 + (1*4)*4
(这里的1*4就是上面的shl $0x2,%edx，因为多了前面的第0个ConstantPoolCacheEntry）

第2个ConstantPoolCacheEntry::_indices字段的地址就是:
ConstantPoolCache的地址 + 8 + (2*4)*4
(这里的2*4是因为多了前面的第0、1个ConstantPoolCacheEntry）

依此类推……

所以_indices字段的地址计算公式就是:

第i个_indices字段的地址 = ConstantPoolCache的地址 + 8 + (i * 4) * 4
(其中i>=0，第一个4代表ConstantPoolCacheEntry有4个字段，第二个4代表ConstantPoolCacheEntry每个字段都占用4字节)

invokespecial的汇编代码在后面还有
Java代码

```
1. mov    0xc(%ecx,%edx,4),%ebx //ConstantPoolCacheEntry::_f1字段(其实是method指针)
2. mov    0x14(%ecx,%edx,4),%edx //ConstantPoolCacheEntry::_flags字段
```

0xc是因为_f1字段在_indices字段后面，偏移多了4个字节，
0x14是因为_flags字段在_indices字段后面，偏移多了12个字节
douyu 2014-07-23
这是ConstantPoolCache的内存布局
可以在我的OpenJDK-Research上面找到:
https://github.com/codefollower/OpenJDK-Research/blob/master/hotspot/my-docs/oops/ConstantPoolCache.java

Java代码

```
/*
    偏移(10) 偏移(16)  字段                  类型
    ----     ------    --------        --------------------
    0        0         _length               int
    4        4         _constant_pool        ConstantPool *

    ConstantPoolCacheEntry (0)
    --------------------------
    8        8         _indices              intx //占4个字节
    12       C         _f1                   Metadata*
    16       10        _f2                   intx
    20       14        _flags                intx
    --------------------------

    ConstantPoolCacheEntry (1)
    --------------------------
    24       18        _indices              intx //占4个字节
    28       1C        _f1                   Metadata*
    32       20        _f2                   intx
    36       24        _flags                intx
    --------------------------

    ......

    ConstantPoolCacheEntry (n)
    --------------------------
    ......
    --------------------------
*/
```

根据上一个回复中的公式算一下验证一下就懂了。
douyu 2014-07-23
还是那句话,要了解所有细节,不要只光看原始的宏汇编,把实际生成的汇编代码debug起来,实际生成的汇编代码有时比原始的宏汇编简单得多。
通过上面这个invokespecial对应的汇编代码的例子就能看出HotSpot的一些细节之美,做了相当多的优化,包括ConstantPoolCacheEntry中各类字段的使用.
当然，也有缺点：就是代码更难懂了。

比如：ClassFileParser::layout_fields就是个极端例子,

为了重排字段的布局搞了一个500多行的方法，繁琐之极。

小施_重名后缀 2014-07-24
我这个是用64位的代码，不同的原因就是因为 wordsize 不同啊,怎么会有问题.那几个字段都是8字节的,没什么问题.
那个4实际上是字段个数.当然这不重要, 而是这个shl的指令,和后面的wordsize乘起来等于sizeof(cpcache entry)
"标志位"这个词可能用的不太合适,因为有个叫flag的字段.不过我前面的
tmp = entryArray[raw_index]._indices 应该说的很明显了吧.后面的都是比较tmp的高位.
而且关于javap的#编号,我说的是和未解析过的_indices字段相等,没说是 R13+1后面的内容.
对于 r13+1,我说的是从0开始
invoke static 对应的汇编

```
(gdb) x /20i $rip
=> 0x7fffed03a5cf:         push   %rax
   0x7fffed03a5d0:         mov    %r13,-0x38(%rbp)
   0x7fffed03a5d4:         movzwl 0x1(%r13),%edx
   0x7fffed03a5d9:         mov    -0x28(%rbp),%rcx
   0x7fffed03a5dd:         shl    $0x2,%edx
   0x7fffed03a5e0:         mov    0x20(%rcx,%rdx,8),%ebx
   0x7fffed03a5e4:         shr    $0x10,%ebx
   0x7fffed03a5e7:         and    $0xff,%ebx
   0x7fffed03a5ed:         cmp    $0xb8,%ebx
   0x7fffed03a5f3:         je     0x7fffed03a84d
   0x7fffed03a5f9:         mov    $0xb8,%ebx
   0x7fffed03a5fe:         callq  0x7fffed03a608
   0x7fffed03a603:         jmpq   0x7fffed03a841
   0x7fffed03a608:         mov    %rbx,%rsi
   0x7fffed03a60b:         lea    0x8(%rsp),%rax
   0x7fffed03a610:         mov    %r13,-0x38(%rbp)
   0x7fffed03a614:         cmpq   $0x0,-0x10(%rbp)
   0x7fffed03a61c:         je     0x7fffed03a699
   0x7fffed03a622:         mov    %rsp,-0x28(%rsp)
   0x7fffed03a627:         sub    $0x80,%rsp
```

mov -0x28(%rbp),%rcx cpool cache在 rbp - 40
shl $0x2,%edx 乘以4
(gdb) p sizeof(constantPoolCacheOopDesc)
$10 = 32

_indices是第一个字段, offset是0 32+0还是32

mov 0x20(%rcx,%rdx,8),%ebx
cpcache + 乘过4的index值 * 8 + 32

(gdb) p sizeof(ConstantPoolCacheEntry)
$15 = 32
就是4乘以8

(gdb) p (constantPoolCacheOopDesc*)$rcx
$11 = (constantPoolCacheOopDesc *) 0xdba984f8
(gdb) p *$11
$12 = {<oopDesc> = {_mark = 0x1, _metadata = {_klass = 0xdb801900, _compressed_klass = 3682605312}, static _bs = 0x7ffff0030cc8}, _length = 6,
_constant_pool = 0xdba97bc0}

这内存结果解释应该没问题

引用

```
(gdb) p *(ConstantPoolCacheEntry*)($rcx + sizeof(constantPoolCacheOopDesc))@6
$13 = {{_indices = 1, _f1 = 0x0, _f2 = 0, _flags = 0}, {_indices = 2, _f1 = 0x0, _f2 = 0,
_flags = 0}, {_indices = 12058627, _f1 = 0xdba98040, _f2 = 0,
       _flags = 813694976}, {_indices = 4, _f1 = 0x0, _f2 = 0, _flags = 0}, {_indices = 5, _f1
= 0x0, _f2 = 0, _flags = 0}, {_indices = 6, _f1 = 0xf58fa990, _f2 = 112,
       _flags = 838860800}}
```

解释器这玩意,,跑一跑固然有不少帮助,但是看看宏汇编就能看个大概

对于这个主题,我懂得也不多,以后就不回复了.