http://hllvm.group.iteye.com/group/topic/37707

[讨论] java_main的汇编入口在哪里

gaolingep 2013-05-03
你好, 问一下java_main的汇编入口在哪里　　gaolingep　　20 小时前
在你的一个视频里面, 我看到你用ollydbg调试找到了main的入口.
例如有一个.java文件

```
1.  public static void main(String[] args) {
2.     System.out.println("111111111111");
3.  }
```

其汇编代码可能是
X86 asm代码

```
1.  [Constants]
2.    # {method} 'main' '([Ljava/lang/String;)V'
3.    # parm0:    ecx       = '[Ljava/lang/String;'
4.    #           [sp+0x20]  (sp of caller)
5.    0x0283d230: mov    %eax,-0x3000(%esp)
6.    0x0283d237: push   %ebp
7.    0x0283d238: sub    $0x18,%esp
8.  ......
9.    0x0283d24e: cmp    (%ecx),%eax        ; implicit exception: dispatches to 0x0283d2d9
10.   0x0283d250: mov    $0x486b080,%edx    ;*invokevirtual println
```

这样的,
我想知道, **上面这段汇编代码是落地到C++的instanceKlass吗?而最终的汇编入口在哪里？**
谢谢回复.

gaolingep 2013-05-03
Re: 你好, 问一下java_main的汇编入口在哪里RednaxelaFX　　19 小时前
RednaxelaFX 写道
您好,
请问您想知道的“最终的汇编入口”是什么？有啥别的对应物么？

您贴的那段代码会是JIT编译后的Java的main()的情况. 不过通常HotSpot VM会从解释器进入Java的main(), 所以通常不会是在这里进入. 进入的地方会是一个由generate_normal_entry()生成出来的代码.

您引用的那段汇编对应的机器码存在CodeCache里. main()方法对应的methodOop对象里由_from_compiled_entry会指向它的起始地址. main()方法所在的类对应的instanceKlass里有_methods数组, 里面引用着main()方法的methodOop对象.

gaolingep 2013-05-03
Re: Re: 你好, 问一下java_main的汇编入口在哪里　　gaolingep　　5 小时前
感谢回信！
按照你的思路, 在调试时加上 -Xcomp -jar test.jar,
第一步在
java.c的
int JNICALL
JavaMain(void * _args)中

```
1.  mainID = (*env)->GetStaticMethodID(env, mainClass, "main",
2.                                     "([Ljava/lang/String;)V");
```

其中mainClass=我的入口java类:com.test,

第二步

```
1.  (*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);
```

第三步
jni_invoke_static中

```
1.  methodHandle method(THREAD, JNIHandles::resolve_jmethod_id(method_id));
```

把上面这个mainID, 转成了methodHandle , 就是methodoop的包装

第四步
在监视器中, 把查看(methodOopDesc *)*((*((Handle*)(&(method))))._handle)
_from_compiled_entry=0x077ac1b4
然后在反汇编中查看这个地址, 但是看来看去, 都不像是main的反汇编
X86 asm代码

```
077AC1B4 00 00                    add         byte ptr [eax],al
077AC1B6 00 00                    add         byte ptr [eax],al
077AC1B8 00 00                    add         byte ptr [eax],al
077AC1BA 00 00                    add         byte ptr [eax],al
077AC1BC 00 00                    add         byte ptr [eax],al
077AC1BE 00 00                    add         byte ptr [eax],al
077AC1C0 C0 24 7D 07 C8 C1 7A 07 shl         byte ptr [edi*2+7AC1C807h],7
077AC1C8 00 00                    add         byte ptr [eax],al
077AC1CA 00 00                    add         byte ptr [eax],al
077AC1CC 00 00                    add         byte ptr [eax],al
077AC1CE 00 00                    add         byte ptr [eax],al
077AC1D0 05 00 00 00 D8           add         eax,0D8000000h
077AC1D5 C1 7A 07 F0              sar         dword ptr [edx+7],0F0h
077AC1D9 C1 7A 07 D8              sar         dword ptr [edx+7],0D8h
077AC1DD 1A 7A 07                 sbb         bh,byte ptr [edx+7]
077AC1E0 30 F8                    xor         al,bh
077AC1E2 79 07          jns       FingerprintMethodsClosure::`RTTI Base Class Array'+13h (077AC1EBh)
```

我也查看了_from_interpreted_entry=0x0736f000
--- e:\workspace\jvm\openjdk\hotspot\src\share\vm\memory\allocation.cpp --------
X86 asm代码

```
0736F000 55                   push        ebp
0736F001 8B EC                mov         ebp,esp
0736F003 51                   push        ecx
0736F004 89 4D FC             mov         dword ptr [this],ecx
0736F007 8B 45 FC             mov         eax,dword ptr [this]
0736F00A 50                   push        eax
0736F00B 68 24 F4 69 07       push        769F424h
0736F010 8B 4D 08             mov         ecx,dword ptr [st]
0736F013 51                   push        ecx
0736F014 E8 E7 9A 1C 00       call        outputStream::print (07538B00h)
0736F019 83 C4 0C             add         esp,0Ch
0736F01C 8B E5                mov         esp,ebp
0736F01E 5D                   pop         ebp
0736F01F C2 04 00             ret         4
```

这个也不是, (因为我在这下了断点, 程序就没跑到这里来)

java代码很简单
Java代码

```
public static void main(String[] args) {
  System.out.println("111111111111");
  System.out.println("222222222222");
}
```

盼回信, 为感.

gaolingep 2013-05-03
Re: Re: Re: 你好, 问一下java_main的汇编入口在哪里 RednaxelaFX 3 小时前
RednaxelaFX 写道
回答问题前我想先建议将这个讨论放到HLLVM群组去: http://hllvm.group.iteye.com. 不介意的话请把这串内容整理起来去开个帖来讨论, 方便大家共同进步 ^_^

有几点:
1、开了-Xcomp之后解释器并不是就不存在了. 虽然UseInterpreter会被设置为false, 但实际上解释器的部分代码还是在干活的. 可以参考之前一帖: http://rednaxelafx.iteye.com/blog/1038324

2、通过JNI的invocation API从C/C++调用Java方法的入口总是从解释器一侧进入的, 所以会从method->from_interpreted_entry()进入.
这个逻辑在JavaCalls::call_helper()里, address entry_point = method->from_interpreted_entry();. 如果目标方法当前没有JIT编译好的版本, 那from_interpreted_entry就指向解释器的方法入口, 也就是之前说的InterpreterGenerator::generate_normal_entry()所生成的代码；如果目标方法已经被JIT编译好, 并且编译好的代码已经完成安装, 那么from_compiled_entry就会指向编译好的方法的入口, 而from_interpreted_entry会指向一个i2c stub, 用于将参数从解释器格式转换为JIT编译后的代码所接受的格式, 也就是解释器与编译后代码的calling convention之间的适配器.

3、同样在JavaCalls::call_helper()里, 在获取entry_point之前会先看看目标方法是不是一定要被编译的, 如果是的话就先调用JIT编译器去编译它. 启用-Xcomp的时候CompilationPolicy::must_be_compiled(method)对所有方法都返回true, 保证所有方法即便从JNI那边调用过来也会先被编译然后再执行. 要在这里的CompileBroker::compile_method()执行完之后, main()方法的from_compiled_entry才会

正确指向JIT编译后的代码入口. 如果只是在jni_invoke_static()的入口的地方设断点来看, 那个时候main()都还没被编译, from_compiled_entry会指向一个c2i stub, 跟前面说的i2c stub正好相反.

gaolingep 2013-05-03
下午照着你的思路, 努力寻找java_main(String[] args) 汇编的入口, JavaCalls::call_helper()里跟踪并且看到执行了

```
1. CompileBroker::compile_method(method, InvocationEntryBci,
2.                               CompLevel_initial_compile,
3.                               methodHandle(), 0, "must_be_compiled", CHECK);
```

然后执行了

```
1. address entry_point = method->from_interpreted_entry();
```

然后执行

```
1. JavaCallWrapper link(method, receiver, result, CHECK);
```

【NO1】这个link没有过多注释, 看代码似乎就是把class字节码编译成asm bytecode, 不知道我理解得对不对
【NO2】然后就执行 StubRoutines::call_stub()(这个代码, 就跳转到前面提到的"不知道干嘛的"asm, 还是没捉住入口.
【NO3】我认为入口应该形如

```
0x0283d250: mov   "11111111",%edx
0x0283d25b: call   0x027fd3c0         ; printf
0x0283d250: mov   "22222222",%edx
0x0283d25b: call   0x027fd3c0         ; printf
```

这样的代码, 但是眼睛看疼了也没找到
【NO4】上一次回复中, 你提到“如果目标方法当前没有JIT编译好的版本”、“如果目标方法已经被JIT编译好, 并且编译好的代码已经完成安装”
这些我不知道是怎么判断的, 代码在哪里
但是,
“先看看目标方法是不是一定要被编译的”这个我知道
JavaCalls::call_helper的
if (CompilationPolicy::must_be_compiled(method)) {

【NO5】"如果只是在jni_invoke_static()的入口的地方设断点来看, 那个时候main()都还没被编译"
这句话不理解, 我似乎看到希望, 但是调来调去又破灭了

RednaxelaFX 2013-05-04
感谢楼主把讨论串从ItEye的站内信迁移到HLLVM群组的论坛来

我觉得有必要理清比较抽象的工作流程再去回答您问的问题,

HotSpot VM是解释执行与编译混合模式的执行引擎. 为了实现这点, Java方法的元数据对象methodOop里记录这两个入口地址, 一个是from_interpreted_entry(), 用于由解释模式的代码跳入该methodOop所代表的Java方法；另一个是from_compiled_entry(), 用于由JIT编译后的Java方法跳入该方法.

几个相关的字段是这样的:

```
1.  class methodOopDesc : public oopDesc {
2.    // Entry point for calling both from and to the interpreter.
3.    address _i2i_entry;         // All-args-on-stack calling convention
4.    // Adapter blob (i2c/c2i) for this methodOop. Set once when method is linked.
5.    AdapterHandlerEntry* _adapter;
6.    // Entry point for calling from compiled code, to compiled code if it exists
7.    // or else the interpreter.
8.    volatile address _from_compiled_entry;       // Cache of: _code ? _code->entry_point() : _adapter->c2i_entry()
9.    // The entry point for calling both from and to compiled code is
10.   // "_code->entry_point()".  Because of tiered compilation and de-opt, this
11.   // field can come and go.  It can transition from NULL to not-null at any
12.   // time (whenever a compile completes).  It can transition from not-null to
13.   // NULL only at safepoints (because of a de-opt).
14.   nmethod* volatile _code;                  // Points to the corresponding piece of native code
15.   volatile address          _from_interpreted_entry; // Cache of _code ? _adapter->i2c_entry() : _i2i_entry
16. };
```

这些字段都是什么时候初始化的呢？

在类加载的“初始化”阶段, 可以有这样的调用路径:

```
instanceKlass::initialize()
-> instanceKlass::initialize_impl()
  -> instanceKlass::link_class()
          -> instanceKlass::link_class_impl()
```

-> instanceKlass::rewrite_class()
-> Rewriter::rewrite()
-> Rewriter::Rewriter()
-> methodOopDesc::link_method()

一个Java类里的所有方法都会在此时link上. 这里就会初始化Java方法的解释模式和编译模式入口:

```
1.  // Called when the method_holder is getting linked. Setup entrypoints so the method
2.  // is ready to be called from interpreter, compiler, and vtables.
3.  void methodOopDesc::link_method(methodHandle h_method, TRAPS) {
4.    // If the code cache is full, we may reenter this function for the
5.    // leftover methods that weren't linked.
6.    if (_i2i_entry != NULL) return;
7.
8.    assert(_adapter == NULL, "init'd to NULL" );
9.    assert( _code == NULL, "nothing compiled yet" );
10.
11.   // Setup interpreter entrypoint
12.   assert(this == h_method(), "wrong h_method()" );
13.   address entry = Interpreter::entry_for_method(h_method);
14.   assert(entry != NULL, "interpreter entry must be non-null");
15.   // Sets both _i2i_entry and _from_interpreted_entry
16.   set_interpreter_entry(entry);
17.   if (is_native() && !is_method_handle_intrinsic()) {
18.     set_native_function(
19.       SharedRuntime::native_method_throw_unsatisfied_link_error_entry(),
20.       !native_bind_event_is_interesting);
21.   }
22.
23.   // Setup compiler entrypoint.  This is made eagerly, so we do not need
24.   // special handling of vtables.  An alternative is to make adapters more
25.   // lazily by calling make_adapter() from from_compiled_entry() for the
26.   // normal calls.  For vtable calls life gets more complicated.  When a
27.   // call-site goes mega-morphic we need adapters in all methods which can be
28.   // called from the vtable.  We need adapters on such methods that get loaded
29.   // later.  Ditto for mega-morphic itable calls.  If this proves to be a
30.   // problem we'll make these lazily later.
31.   (void) make_adapters(h_method, CHECK);
32.
33.   // ONLY USE the h_method now as make_adapter may have blocked
34.
35. }
```

_i2i_entry 与 _from_interpreted_entry:
从methodOopDesc::link_method()调用methodOopDesc:: set_interpreter_entry()

```
1.  void set_interpreter_entry(address entry)     { _i2i_entry = entry;  _from_interpreted_entry = entry; }
```

_adapter 与 _from_compiled_entry:
从methodOopDesc::link_method()调用methodOopDesc::make_adapters()

```
1.  address methodOopDesc::make_adapters(methodHandle mh, TRAPS) {
2.    // Adapters for compiled code are made eagerly here.  They are fairly
3.    // small (generally < 100 bytes) and quick to make (and cached and shared)
4.    // so making them eagerly shouldn't be too expensive.
5.    AdapterHandlerEntry* adapter = AdapterHandlerLibrary::get_adapter(mh);
6.    if (adapter == NULL ) {
7.      THROW_MSG_NULL(vmSymbols::java_lang_VirtualMachineError(), "out of space in CodeCache for adapters");
8.    }
9.
10.   mh->set_adapter_entry(adapter);
11.   mh->_from_compiled_entry = adapter->get_c2i_entry();
12.   return adapter->get_c2i_entry();
13. }
```

这个初始化反映了HotSpot VM的混合模式执行引擎默认以解释模式启动.
可以看到:
_i2i_entry 指向该方法的解释器入口. 这个值设定好就不会变了.
_from_interpreted_entry 初始的值与 _i2i_entry 一样. 但后面当该Java方法被JIT编译并“安装”之后, _from_interpreted_entry 就会被设置为指向 i2c adapter stub. 而如果因为某些原因需要抛弃掉之前已经编译并安装好的机器码, 则 _from_interpreted_entry 会被恢复为 _i2i_entry.

_adapter 指向该Java方法的签名(signature)所对应的 i2c2i adapter stub. 其实是一个 i2c stub 和一个 c2i stub 粘在一起这样的对象, 可以看到用的时候都是从 _adapter 取 get_i2c_entry() 或 get_c2i_entry(). 这些adapter stub用于在HotSpot VM里的解释模式与编译模式的代码之间适配其calling convention. HotSpot VM里的解释模式calling convention用栈来传递参数, 而编译模式的calling convention更多采用寄存器来传递参数, 两者不兼容, 因而从解释模式的代码调用已经被编译的方法, 或者反之, 都需要在调用时进行适配.
_from_compiled_entry 初始值指向c2i adapter stub. 原因上面已经说了, 因为一开始该方法尚未被JIT编译, 需要在解释模式执行, 那么从已经JIT编译好的Java方法调用过来的话就需要进行calling convention的转换, 把参数挪到正确的位置上. 当该方法被JIT编译并"安装"完之后, _from_compiled_entry 就会指向编译出来的机器码的入口, 具体说时指向verified entry point. 如果要抛弃之前编译好的机器码, 那么 _from_compiled_entry 会恢复为指向 c2i stub.
_code 这个字段指向含有JIT编译后的机器码. 初始值为NULL, 意味着该方法尚未被JIT编译(或者说至少尚未被"标准编译"；OSR编译的入口不在这里). 当一个方法被JIT编译并"安装"后, _code 就会指向编译生成的 nmethod . 而要抛弃编译好的代码时 _code 会恢复为 NULL.

JIT编译的产物包装在nmethod对象里. 编译完成后"安装"的逻辑在:

```
1.  // Install compiled code.  Instantly it can execute.
2.  void methodOopDesc::set_code(methodHandle mh, nmethod *code) {
3.    assert( code, "use clear_code to remove code" );
4.    assert( mh->check_code(), "" );
5.
6.    guarantee(mh->adapter() != NULL, "Adapter blob must already exist!");
7.
8.    // These writes must happen in this order, because the interpreter will
9.    // directly jump to from_interpreted_entry which jumps to an i2c adapter
10.   // which jumps to _from_compiled_entry.
11.   mh->_code = code;           // Assign before allowing compiled code to exec
12.
13.   int comp_level = code->comp_level();
14.   // In theory there could be a race here. In practice it is unlikely
15.   // and not worth worrying about.
16.   if (comp_level > mh->highest_comp_level()) {
17.     mh->set_highest_comp_level(comp_level);
18.   }
19.
20.   OrderAccess::storestore();
21.
22.   mh->_from_compiled_entry = code->verified_entry_point();
23.   OrderAccess::storestore();
24.   // Instantly compiled code can execute.
25.   if (!mh->is_method_handle_intrinsic())
26.     mh->_from_interpreted_entry = mh->get_i2c_entry();
27. }
```

所谓"安装"其实就是把nmethod与其对应的methodOop关联起来, 把各入口都设置上.

而抛弃已编译好的机器码时与"安装"相反的"卸载"逻辑在:

```
1.  // Revert to using the interpreter and clear out the nmethod
2.  void methodOopDesc::clear_code() {
3.
4.    // this may be NULL if c2i adapters have not been made yet
5.    // Only should happen at allocate time.
6.    if (_adapter == NULL) {
7.      _from_compiled_entry    = NULL;
8.    } else {
9.      _from_compiled_entry    = _adapter->get_c2i_entry();
10.   }
11.   OrderAccess::storestore();
12.   _from_interpreted_entry = _i2i_entry;
13.   OrderAccess::storestore();
14.   _code = NULL;
15. }
```

在HotSpot VM的实现里, 除了CompileTheWorld(CTW)这个用于测试动态编译器的特殊模式之外, 一个类被加载进来之后, 里面的方法最早最早也要到它即将第一次被执行的时候才有可能被JIT编译器所编译. 在那之前, 非abstract非native的Java方法的"代码"部分都只是好好的以Java字节码的形式存在, 没有对应的机器码(上面所说的 _code 字段为NULL).

启用-Xcomp模式的时候, UseInterpreter被设置为false, 于是所有Java方法都会被认为是"首次执行前就必须被编译"的:

```
1.  // Returns true if m must be compiled before executing it
```

```
2.  // This is intended to force compiles for methods (usually for
3.  // debugging) that would otherwise be interpreted for some reason.
4.  bool CompilationPolicy::must_be_compiled(methodHandle m, int comp_level) {
5.    if (m->has_compiled_code()) return false;       // already compiled
6.    if (!can_be_compiled(m, comp_level)) return false;
7.
8.    return !UseInterpreter ||                                 // must compile all methods
9.          (UseCompiler && AlwaysCompileLoopMethods && m->has_loops() &&
    CompileBroker::should_compile_new_jobs()); // eagerly compile loop methods
10. }
```

有两个地方会受-Xcomp的影响.
一是当解释器对某个方法做resolution的时候(通常发生在该方法即将被调用时):

```
1.  void CallInfo::set_common(KlassHandle resolved_klass, KlassHandle selected_klass, methodHandle resolved_method,
    methodHandle selected_method, int vtable_index, TRAPS) {
2.    // ...
3.
4.    if (CompilationPolicy::must_be_compiled(selected_method)) {
5.      // This path is unusual, mostly used by the '-Xcomp' stress test mode.
6.
7.      // ...
8.      CompileBroker::compile_method(selected_method, InvocationEntryBci,
9.                                    CompilationPolicy::policy()->initial_compile_level(),
10.                                   methodHandle(), 0, "must_be_compiled", CHECK);
11.   }
12. }
```

另一个是外部代码通过JNI的invocation API来调用Java方法时:

```
1.  void JavaCalls::call_helper(JavaValue* result, methodHandle* m, JavaCallArguments* args, TRAPS) {
2.    methodHandle method = *m;
3.    // ...
4.    if (CompilationPolicy::must_be_compiled(method)) {
5.      CompileBroker::compile_method(method, InvocationEntryBci,
6.                                    CompilationPolicy::policy()->initial_compile_level(),
7.                                    methodHandle(), 0, "must_be_compiled", CHECK);
8.    }
9.
10.   // ...
11. }
```

这样就保证某个Java方法在首次被调用的那个瞬间会先被JIT编译, 然后再跳进目标方法去执行.

JIT编译的产物时method对象, 里面包含一些描述信息(元数据), 以及编译生成的机器码本身. 每个method有两个实际入口, 一个是unverified entry point(UEP), 用于实现虚方法分派的monomorphic inline cache；另一个是verified entry point(VEP), 是方法的真正入口. 只有需要虚方法分派的方法才会有独立的UEP；对静态方法、私有成员方法之类的Java方法, UEP与VEP实际上在同一个位置. 关于UEP与VEP的更详细介绍, 请参考HotSpot的wiki: https://wiki.openjdk.java.net/display/HotSpot/VirtualCalls (不过这个wiki页面当前把UEP与VEP弄反了, 我稍后会去修正)

==========

之前也提到JNI invocation API的实现总是从解释器的一侧进入. 原因是解释器的calling convention比较简单, 传参数都在栈上, 这样从native code把参数转到解释器所要求的位置上就比较简单, 逻辑不太受参数个数的影响. 反正通过methodOop的from_interpreted_entry 也能正确的进入Java方法已被编译的版本, 这样实现就比较方便一些.

==========

具体到Java层的main()方法在-Xcomp模式下的执行. 假设我们的Main-Class名为JavaMainClass, 下面为了区分java launcher里C的main()与Java层程序里的main(), 把后者写作JavaMainClass.main().

从刚进入C的main()开始:

primordial thread:
main()
-> //... 做一些参数检查
-> //... 开启新线程作为main线程, 让它从JavaMain()开始执行；该线程等待main线程执行结束

main thread:

JavaMain()
-> //... 找到指定的JVM
-> //... 加载并初始化JVM
-> //... 根据Main-Class指定的类名加载JavaMainClass
-> //... 在JavaMainClass类里找到名为"main", 签名为"([Ljava/lang/String;)V", 修饰符是public的静态方法
-> (*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs); // 通过JNI调用JavaMainClass.main()
// 以上步骤都还在java launcher的控制下；当控制权转移到JavaMainClass.main()之后就没java launcher什么事了, 等JavaMainClass.main()返回之后java launcher才接手过来清理和关闭JVM.
// 下面就都在HotSpot VM里了.
  -> jni_CallStaticVoidMethod()   // HotSpot VM里对JNI的CallStaticVoidMethod的实现. 留意要传给Java方法的参数以C的可变长度参数(...)传入, 这个函数将其收集打包为JNI_ArgumentPusherVaArg对象
        -> jni_invoke_static()   // 这里进一步将要传给Java的参数转换为JavaCallArguments对象传下去
        -> JavaCalls::call()   // 真正底层实现的开始. 这个方法只是层皮, 把JavaCalls::call_helper()用os::os_exception_wrapper()包装起来, 目的是设置HotSpot VM的C++层面的异常处理
        -> JavaCalls::call_helper()
        -> //... 检查目标方法是否为空方法, 是的话直接返回
        -> //... 检查目标方法是否“首次执行前就必须被编译”, 是的话调用JIT编译器去编译目标方法
        -> //... 获取目标方法的解释模式入口from_interpreted_entry, 下面将其称为entry_point
        -> //... 确保Java栈溢出检查机制正确启动
        -> //... 创建一个JavaCallWrapper, 用于管理JNIHandleBlock的分配与释放, 以及在调用Java方法前后保存和恢复Java的frame pointer/stack pointer
        -> StubRoutines::call_stub()(...) //... StubRoutines::call_stub()返回一个指向call stub的函数指针, 紧接着调用这个call stub, 传入前面获取的entry_point和要传给Java方法的参数等信息
        // call stub是在VM初始化时生成的. 对应的代码在StubGenerator::generate_call_stub(). 它的功能可以参考代码前面的注释.
        -> //... 把相关寄存器的状态调整到解释器所需的状态
        -> //... 把要传给Java方法的参数从JavaCallArguments对象解包展开到解释模式calling convention所要求的位置
        -> //... 跳转到前面传入的entry_point, 也就是目标方法的from_interpreted_entry
              -> //... 在-Xcomp模式下, 实际跳入的是i2c adapter stub, 将解释模式calling convention传入的参数挪到编译模式calling convention所要求的位置
              -> //... 跳转到目标方法被JIT编译后的代码里, 也就是跳到 nmethod 的 VEP 所指向的位置
              -> //... 正式开始执行目标方法被JIT编译好的代码 <- 这里大概就是楼主想要的“main()方法的真正入口”

==========

回到楼主的问题:

gaolingep 写道
【NO1】这个link没有过多注释, 看代码似乎就是把class字节码编译成asm bytecode, 不知道我理解得对不对

不对. JavaCallWrapper做的事情非常简单, 注释里也说清楚了:

1. // A JavaCallWrapper is constructed before each JavaCall and destructed after the call.
2. // Its purpose is to allocate/deallocate a new handle block and to save/restore the last
3. // Java fp/sp. A pointer to the JavaCallWrapper is stored on the stack.

把Java字节码编译为机器码的工作之前就在调用CompileBroker::compile_method()的时候做完了.
另外没有“asm bytecode”这种东西.机器码就是机器码,一般不将其称为bytecode(能只用一个字节放opcode的机器这年头也不多了…)

gaolingep 写道
【NO2】然后就执行 StubRoutines::call_stub()(这个代码, 就跳转到前面提到的"不知道干嘛的"asm, 还是没捉住入口.

“不知道干嘛”做的事情主要就是把要传给Java的参数从传入的JavaCallArguments对象里解开, 放到解释器预期的位置上.
解开完了之后会用一个call指令(在x86/x64上)跳转到之前传入的entry_point, 这样就进入目标方法了.

gaolingep 写道
【NO3】我认为入口应该形如

```
  0x0283d250: mov   "11111111",%edx
  0x0283d25b: call   0x027fd3c0         ; printf
  0x0283d250: mov   "22222222",%edx
  0x0283d25b: call   0x027fd3c0         ; printf
```

这样的代码, 但是眼睛看疼了也没找到

其实这事情非常简单. **您想看到JavaMainClass.main()在JIT编译后生成的机器码在哪里, 内容是怎样的, 只要打开-XX:+PrintAssembly来启动即可. 启用该参数需要hsdis插件, 请在HLLVM群组里搜一下, 已经有很多人问过了.**

假设要运行的代码如下:
Java代码

1.

在我的Mac OS X上用Oracle JDK 1.7.0_05来运行
Command prompt代码

```
1. java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -Xcomp JavaMainClass
```

得到的输出可以看到
X64 asm代码

```
1. Decoding compiled method 0x00000000010a54cb10:
2. Code:
3. [Entry Point]
4. [Verified Entry Point]
5. [Constants]
6.   # {method} 'main' '([Ljava/lang/String;)V' in 'JavaMainClass'
7.   # parm0:    rsi:rsi   = '[Ljava/lang/String;'
8.   #           [sp+0x20]  (sp of caller)
9.   0x00000000010a54cc40: mov    %eax,-0x14000(%rsp)
10.  0x00000000010a54cc47: push   %rbp
11.  0x00000000010a54cc48: sub    $0x10,%rsp        ;*synchronization entry
12.                                ; - JavaMainClass::main@-1 (line 3)
13.  0x00000000010a54cc4c: mov    $0x11,%esi
14.  0x00000000010a54cc51: xchg   %ax,%ax
15.  0x00000000010a54cc53: callq  0x00000000010a50c020  ; OopMap{off=24}
16.                                ;*getstatic out
17.                                ; - JavaMainClass::main@0 (line 3)
18.                                ;   {runtime_call}
19.  0x00000000010a54cc58: callq  0x0000000109caf432  ;*getstatic out
20.                                ; - JavaMainClass::main@0 (line 3)
21.                                ;   {runtime_call}
22.  0x00000000010a54cc5d: hlt
23.  0x00000000010a54cc5e: hlt
24.  0x00000000010a54cc5f: hlt
25. [Exception Handler]
26. [Stub Code]
27.  0x00000000010a54cc60: jmpq   0x00000000010a5308a0  ;   {no_reloc}
28. [Deopt Handler Code]
29.  0x00000000010a54cc65: callq  0x00000000010a54cc6a
30.  0x00000000010a54cc6a: subq   $0x5,(%rsp)
31.  0x00000000010a54cc6f: jmpq   0x00000000010a50bc00  ;   {runtime_call}
32.  0x00000000010a54cc74: hlt
33.  0x00000000010a54cc75: hlt
34.  0x00000000010a54cc76: hlt
35.  0x00000000010a54cc77: hlt
```

这**个0x00000000010a54cc40**地址就是-Xcomp模式下JavaMainClass.main()货真价实的编译后入口.
在调试的时候, 您可以等-XX:+PrintAssembly功能打印出JavaMainClass.main()的内容之后, 再去看methodOop里各入口的状态, 留意_from_compiled_entry的值是否指向这里的起始地址. 如果不是的话说明您查看methodOop内容的方式不太对…

但编译出来的代码看起来怪怪的对吧, System.out.println()跑哪儿去了？
这就请参考我之前写的一帖: http://rednaxelafx.iteye.com/blog/1038324, 说的就是这个问题.

您原本所想像的
X86 asm代码

```
1.  0x0283d250: mov    "11111111",%edx
2.  0x0283d25b: call   0x027fd3c0        ; printf
3.  0x0283d250: mov    "22222222",%edx
4.  0x0283d25b: call   0x027fd3c0        ; printf
```

即便在没碰到上面那帖说的问题时也跟HotSpot VM实际会编译出来的代码不太一样. println()方法会被内联到JavaMainClass.main()里, 于是**最终生成的JavaMainClass.main()的机器码会比您想像的要更复杂一些**.

就算通过配置文件来禁用方法内联, 编译出来的机器码也会时这个样子的:
X64 asm代码

```
1. Decoding compiled method 0x0000000109b5e410:
2. Code:
3. [Entry Point]
4. [Verified Entry Point]
5. [Constants]
```

```
6.  # {method} 'main_' '([Ljava/lang/String;)V' in 'JavaMainClass'
7.  # parm0:    rsi:rsi   = '[Ljava/lang/String;'
8.  #          [sp+0x20]  (sp of caller)
9.  0x0000000109b5e560: mov    %eax,-0x14000(%rsp)
10. 0x0000000109b5e567: push   %rbp
11. 0x0000000109b5e568: sub    $0x10,%rsp        ;*synchronization entry
12.                                ; - JavaMainClass::main_@-1 (line 10)
13. 0x0000000109b5e56c: movabs $0x7e4cb0d08,%rbp  ;   {oop(a 'java/lang/Class' = 'java/lang/System')}
14. 0x0000000109b5e576: mov    0x74(%rbp),%r11d   ;*getstatic out
15.                                ; - JavaMainClass::main_@0 (line 10)
16. 0x0000000109b5e57a: test   %r11d,%r11d
17. 0x0000000109b5e57d: je     0x0000000109b5e5bc  ;*invokevirtual println
18.                                ; - JavaMainClass::main_@5 (line 10)
19. 0x0000000109b5e57f: lea    (%r12,%r11,8),%rsi  ;*getstatic out
20.                                ; - JavaMainClass::main_@0 (line 10)
21. 0x0000000109b5e583: movabs $0x7e4dc7118,%rdx  ;   {oop("111111111111")}
22. 0x0000000109b5e58d: xchg   %ax,%ax
23. 0x0000000109b5e58f: callq  0x00000000109a5dc60  ; OopMap{rbp=Oop off=52}
24.                                ;*invokevirtual println
25.                                ; - JavaMainClass::main_@5 (line 10)
26.                                ;   {optimized virtual_call}
27. 0x0000000109b5e594: mov    0x74(%rbp),%r10d   ;*getstatic out
28.                                ; - JavaMainClass::main_@8 (line 11)
29. 0x0000000109b5e598: test   %r10d,%r10d
30. 0x0000000109b5e59b: je     0x0000000109b5e5cd  ;*invokevirtual println
31.                                ; - JavaMainClass::main_@13 (line 11)
32. 0x0000000109b5e59d: lea    (%r12,%r10,8),%rsi  ;*getstatic out
33.                                ; - JavaMainClass::main_@8 (line 11)
34. 0x0000000109b5e5a1: movabs $0x7e4dc7440,%rdx  ;   {oop("222222222222")}
35. 0x0000000109b5e5ab: callq  0x00000000109a5dc60  ; OopMap{off=80}
36.                                ;*invokevirtual println
37.                                ; - JavaMainClass::main_@13 (line 11)
38.                                ;   {optimized virtual_call}
39. 0x0000000109b5e5b0: add    $0x10,%rsp
40. 0x0000000109b5e5b4: pop    %rbp
41. 0x0000000109b5e5b5: test   %eax,-0xe9d5bb(%rip)        # 0x00000000108cc1000
42.                                ;   {poll_return}
43. 0x0000000109b5e5bb: retq
```

还是比您想像的要复杂那么一点.

gaolingep 写道
【NO4】上一次回复中, 你提到“如果目标方法当前没有JIT编译好的版本”、“如果目标方法已经被JIT编译好, 并且编译好的代码已经完成安装”
这些我不知道是怎么判断的, 代码在哪里

HotSpot VM判断目标方法有没有JIT编译好的逻辑很简单, 就是看methodOop里的 _code 字段是否非NULL. 如果不是NULL说明已经有JIT编译好的代码而且安装好了, 如果是NULL说明还没JIT编译好或者是还没安装好.

只考虑方法调用的时候发生的事, 其实不需要显式判断目标方法(被调用方)有没有被编译及安装, 只要知道自己(调用方)是解释模式还是编译模式的, 据此选择 _from_interpreted_entry 或 _from_compiled_entry 进入就好了, 这两个入口本身就暗含着目标方法是解释模式还是编译模式的正确处理逻辑.

gaolingep 写道
【NO5】"如果只是在jni_invoke_static()的入口的地方设断点来看, 那个时候main()都还没被编译"

从前面讲的整体工作流程可以看出, jni_invoke_static()在一开始的地方把传入的methodID解析为具体的methodOop, 然后用一个methodHandle包住它. 但在这个时候JavaMainClass.main()方法尚未被JIT编译, 它对应的methodOop里的各入口都还在解释模式的初始状态, 如果在这个地方下断点去观察那个methodOop的内容, 恐怕与您预期的内容会不一样.
如果要观察JavaMainClass.main()已经被JIT编译好的时候其methodOop的状态, 一个可下断点的地方是在JavaCalls::call_helper()里从CompileBroker::compile_method()返回之后.
(注意这里的讨论全部都基于-Xcomp模式这个前提条件. 不然的话JavaMainClass.main()就应该正常的从解释器进入)

gaolingep 2013-05-06
感谢RednaxelaFX 的解惑, 让我对方法从字节码到汇编码的过程有了更进一步的理解, 方法的入口我也找到了, 谢谢.

"但编译出来的代码看起来怪怪的对吧, System.out.println()跑哪儿去了？

这就请参考我之前写的一帖: http://rednaxelafx.iteye.com/blog/1038324, 说的就是这个问题. "
没发现编译后的代码哪里怪怪的,
Asm代码

```
028EC060 89 84 24 00 80 FF FF mov         dword ptr [esp-8000h],eax
028EC067 55                   push        ebp
028EC068 83 EC 18             sub         esp,18h
028EC06B 90                   nop
028EC06C 90                   nop
028EC06D 90                   nop
028EC06E 90                   nop
028EC06F 90                   nop
028EC070 E9 70 00 00 00       jmp         028EC0E5
028EC075 90                   nop
028EC076 90                   nop
028EC077 90                   nop
028EC078 E9 93 00 00 00       jmp         028EC110
028EC07D 90                   nop
028EC07E 3B 01                cmp         eax,dword ptr [ecx]
028EC080 BA A0 10 C8 07       mov         edx,7C810A0h
028EC085 90                   nop
028EC086 B8 FF FF FF FF       mov         eax,0FFFFFFFFh
028EC08B E8 40 F4 FB FF       call        028AB4D0
```

这是内存中的代码, 和-XX:+PrintAssembly 中的代码对应上了, call 028AB4D0, 调用的就是println虚函数.

今天晚上做几个实验再验证一下, 谢谢.

RednaxelaFX 2013-05-06
gaolingep 写道
没发现编译后的代码哪里怪怪的

我用的是HotSpot Server VM. 如果您用的是Client VM的话那生成的代码会有点不一样. 后者生成的代码比前者保守一些, 对未加载的类更宽容些(会生成代码resolve了符号引用后patch代码到正确的调用地址)

gaolingep 2013-05-06
_from_compiled_entry=0x025dc060中, 看到asm代码如下

[图片居然上传不上来, 画了一个viso图,
instanceOopDesc继承于
oopDesc(_mark:markOopDesc* ,_metadata:<_metadata:union>,_bs:BarrierSet *)
其中
_metadata:union(_klass:klassOopDesc*, _compressed_klass;unsigned int)
]

Asm代码

```
0x025dc075: nop
0x025dc076: nop
0x025dc077: nop
0x025dc078: jmp    0x025dc110        ; implicit exception: dispatches to 0x025dc0ef
0x025dc07d: nop                      ;*getstatic out
                                     ; - com.gaoling.T2::main@0 (line 10)
0x025dc07e: cmp    (%ecx),%eax       ; implicit exception: dispatches to 0x025dc11a
0x025dc080: mov    $0xf2310a0,%edx   ;*invokevirtual println
                                     ; - com.gaoling.T2::main@5 (line 10)
                                     ;   {oop("111111111111")}
0x025dc085: nop
0x025dc086: mov    $0xffffffff,%eax  ;   {oop(NULL)}
```

关注 0x025dc080: mov $0xf2310a0,%edx的内容

Binary代码

```
0x0F2310A0  01 00 00 00 58 77 18 1f b8 10 23 0f 00 00 00 00 0c 00 00 00 00 00 00 00 01 00 00  ....Xw..?.#.....................
0x0F2310BB  00 a0 03 18 1f 0c 00 00 00 31 00 31 00 31 00 31 00 31 00 31 00 31 00 31 00 31 00  .?.......1.1.1.1.1.1.1.1.1.
0x0F2310D6  31 00 31 00 31 00 00 00 00 00 01 00 00 00 58 77 18 1f f8 10 23 0f 00 00 00 00 0c  1.1.1.........Xw..?.#......
0x0F2310F1  00 00 00 00 00 00 00 01 00 00 00 a0 03 18 1f 0c 00 00 00 32 00 32 00 32 00 32 00  ...........?.......2.2.2.2.
0x0F23110C  32 00 32 00 32 00 32 00 32 00 32 00 32 00 32 00 00 00 00 00 be ba ad ba be ba ad  2.2.2.2.2.2.2.2.2.....???????
```

拆分一下
_mark:0x00000001

_klass:0x1f187758
_compressed_klass;0f2310b8 指向后面真正的object
_bs:00 00 00 00: 干嘛用的？
0c 00 00 00 00 00 00 00: 无意义

01 00 00 00 a0 03 18 1f
才是指向了的真正string对象, 是从上面的_compressed_klass指向过来的
其中
_mark:0x00000001
_klass:0x1f1803a0

0c 00 00 00貌似是这个'111111111111'的长度
31 00 31 00 31 00 31 00 31 00 31 00 31 00 31 00 31 00 .?.....1.1.1.1.1.1.1.1.1.
0x0F2310D6 31 00 31 00 31 00 才是真正的值

问题:
【NO1】 _bs:00 00 00 00: 干嘛用的？
这个 0c 00 00 00 00 00 00 00: 无意义, 真的无意义吗？
【NO2】为什么会有这种指向的指向
【NO3】在读源码的过程中, 发现
jstring mainClassName;

typedef jobject jstring;
typedef _jobject *jobject;
class _jobject {};

貌似mainClassName是一个空类的实例, 但实际上mainClassName是一个oopdesc, jvm为什么要这样做
【NO4】后面准备开始分析_klass的内容, 有没有指导性的建议

RednaxelaFX 2013-05-06
(楼主以后发帖请用BBCode编辑器, 这边的富文本编辑器感觉很不好用...)
(另外, 图片可以上传到自己在ItEye的相册里然后链过来. ItEye的群组的论坛不让上传图片实在奇怪)

楼主先把你用的java -version发一下我再回答前俩问题. 不过可以一说的是instanceOopDesc的结构跟你想像的或看到的不一样. 它的头部跟oopDesc一模一样, 开头是mark word, 然后是metadata指针. 这个metadata指针是个union(不是struct, 注意！), 里面到底是_klass还是_compressed_klass取决于你在用什么版本的VM, 什么模式. 所以请你先发一下java -version. 看样子你用的是32-bit HotSpot Client VM.

3、这样保证通过JNI API传到VM之外的handle都是不透明的, 外部程序就不能随意使用对象里真实的成员, 而只能通过JNI提供的函数来做有限的、受VM监管的操作.

4、先用oopHierarchy.hpp开始.

gaolingep 2013-05-07
谢谢.

instanceOopDesc 以及oopDesc 的源码

```
1.  class instanceOopDesc : public oopDesc
```

Cpp代码

```
1.  class oopDesc {
2.    friend class VMStructs;
3.  private:
4.    volatile markOop  _mark;
5.    union _metadata {
6.      wideKlassOop    _klass;
7.      narrowOop       _compressed_klass;
8.    } _metadata;
9.
10.   // Fast access to barrier set.  Must be initialized.
11.   static BarrierSet* _bs;
12.
13. public:
```

```
14.  enum ConcSafeType {
15.    IsUnsafeConc = false,
16.    IsSafeConc   = true
17.  };
```

回答MR.R 关于jdk版本的问题
我在win7 x86下编译了jvm
E:\WorkSpace\JVM\openjdk\build\windows-i586\j2sdk-image\bin\java -version
openjdk version "1.7.0-internal"
OpenJDK Runtime Environment (build 1.7.0-internal-gaoling_2013_04_15_00_11-b00)
OpenJDK Client VM (build 21.0-b17, mixed mode)

在win7 x64下用vs2012打开hotspot做调试(x64+vs2012实在编译不过去)
E:\WorkSpace\JVM\openjdk\hotspot\build\vs-i486\compiler1\debug\hotspot.exe -version
Using java runtime at: E:\WorkSpace\JVM\openjdk\build\windows-i586\j2sdk-image\jre
openjdk version "1.7.0-internal"
OpenJDK Runtime Environment (build 1.7.0-internal-gaoling_2013_04_15_00_11-b00)
OpenJDK Client VM (build 21.0-b17-internal-debug, mixed mode)

RednaxelaFX 2013-05-09
OK, good. 这样就可以回答之前的问题了.
HotSpot VM里的对象布局会受平台和参数的影响, 必须要知道您实际用的是什么版本才可以解释您看到的内存里的东西是什么意思.

前面您贴的内存数据, 实际上是这样的:
Memory代码

```
1.  instanceOopDesc for String "111111111111" (size = 24)
2.  0x0F2310A0  01 00 00 00  ;; oopDesc::_mark          = 0x00000001
3.  0x0F2310A4  58 77 18 1f  ;; oopDesc::_metadata._klass = 0x1F187758 (-> klassOopDesc for java.lang.String)
4.  0x0F2310A8  b8 10 23 0f  ;; java.lang.String.value    = 0x0F2310B8 (-> typeArrayOopDesc for char[])
5.  0x0F2310AC  00 00 00 00  ;; java.lang.String.offset   = 0
6.  0x0F2310B0  0c 00 00 00  ;; java.lang.String.count    = 12
7.  0x0F2310B4  00 00 00 00  ;; java.lang.String.hash     = 0
8.
9.  typeArrayOopDesc for char[12] (size = 40)
10. 0x0F2310B8  01 00 00 00  ;; oopDesc::_mark          = 0x00000001
11. 0x0F2310BC  a0 03 18 1f  ;; oopDesc::_metadata._klass = 0x1F1803A0 (-> klassOopDesc for char[])
12. 0x0F2310C0  0c 00 00 00  ;; arrayOopDesc::_length     = 12
13. 0x0F2310C4  31 00 31 00  ;; [0] = '1', [1] = '1'
14. 0x0F2310C8  31 00 31 00  ;; [2] = '1', [3] = '1'
15. 0x0F2310CC  31 00 31 00  ;; [4] = '1', [5] = '1'
16. 0x0F2310D0  31 00 31 00  ;; [6] = '1', [7] = '1'
17. 0x0F2310D4  31 00 31 00  ;; [8] = '1', [9] = '1'
18. 0x0F2310D8  31 00 31 00  ;; [10] = '1', [11] = '1'
19. 0x0F2310DC  00 00 00 00  ;; (padding: 4 bytes)
```

Java对象布局的计算在ClassFileParser里, 其中只有对象头的部分是在C++里显式声明的(也就是oopDesc的部分), 而后面Java字段的布局都是HotSpot VM自己计算的, 没有在C++里直接声明. 这是实现VM的常见技巧.

关于32位HotSpot VM里java.lang.String的布局, 我正好以前在几个地方都提到过,
借助HotSpot SA来一窥PermGen上的对象
分享: Java 程序的编译, 加载 和 执行

BarrierSet* _bs是instanceOopDesc的静态变量, 不包含在其实例里.

---

huangriyan 2014-08-28
generate_call_stub、generate_normal_entry这种方法什么时候调用？他们 在 generate 的时候已经执行了 __ pc() 到 return address 这段代码了, 之后再call address , 岂不是执行了两次, 如 generate_normal_entry 内 , address entry_point = __ pc(); 到 return entry_point 已经在 generate_normal_entry 的时候执行一次了, 再在 call_stub 中 call 一次, 岂不是在执行了两次这段代码

cheney_love 2014-09-16
huangriyan 写道
generate_call_stub、generate_normal_entry这种方法什么时候调用？他们 在 generate 的时候已经执行了 __ pc() 到 return address 这段代码了, 之后再call address , 岂不是执行了两次, 如 generate_normal_entry 内 , address entry_point = __ pc(); 到 return entry_point 已经在 generate_normal_entry 的时候执行一次了, 再在 call_stub 中 call 一次, 岂不是在执行了两次这段代码

jvm 启动的时候调用