JVM方法执行的来龙去脉
半栈工程师

趁着春节放假,借着《揭秘Java虚拟机》,好好看了下Hotspot源码,对JVM执行Java方法的过程有了更深入的了解. 大过年的,不发红包,发篇文章吧.

一:CallStub例程
普通的Java类被编译成字节码后,对Java方法的调用都会转换为invoke指令,而Java第一个方法是由谁调用的呢? Java main()方法的执行其实是通过JVM自己调用的.
不过对于JVM来说,无论是如何执行Java方法,都是通过JavaCalls模块来实现的.

JavaCalls这个名字取得很形象,一看就知道是用来调用Java方法的.
JavaCalls中有很多用来调用Java方法的函数,如call_virtual(),call_special(),call_static等,用来调用不同类型的Java方法.
不过这些函数最终都是调用的call()方法:

```
void JavaCalls::call(JavaValue* result,methodHandle method,JavaCallArguments* args,TRAPS) {
  ......
  os::os_exception_wrapper(call_helper,result,&method,args,THREAD);
}
void os::os_exception_wrapper(java_call_t f,JavaValue* value,methodHandle* method,JavaCallArguments* args,Thread* thread) {
  f(value,method,args,thread);
}
```
f其实就是call()方法中传入的call_help,这里相当于调用了call_help(value,method,args,thread),因为call_help其实就是个函数指针,同样定义在JavaCalls中.
```
void JavaCalls::call_helper(JavaValue* result,methodHandle* m,JavaCallArguments* args,TRAPS) {
    ......
      StubRoutines::call_stub()(
        (address)&link,
        // (intptr_t*)&(result->_value),// see NOTE above (compiler problem)
        result_val_address,         // see NOTE above (compiler problem)
        result_type,
        method(),
        entry_point,
        args->parameters(),
        args->size_of_parameters(),
        CHECK
      );
    ......
}
```
可见call_help中最终是通过 StubRoutines::call_stub() 的返回值来调用java方法的;
由此可知,call_stub()返回的肯定也是个函数指针之类的.
我们来看看call_stub()返回的具体是啥.

```
/openjdk/hotspot/src/share/vm/runtime/stubRoutines.hpp
static CallStub call_stub() { return CAST_TO_FN_PTR(CallStub, _call_stub_entry); }
```
call_stub()返回了_call_stub_entry例程的地址.
例程是啥,我开始时也觉得很难理解,而且“例程”这个名字也取得很奇怪.
其实例程可以理解为用汇编写好的一个方法,和内联汇编差不多,被加载到内存中后,我们就可以直接通过它的首地址来调用执行它.
很多读者可能也觉得很奇怪,为什么要用汇编呢? 是因为汇编快吗? 那C语言写的方法最后不也会被编译成汇编吗,有什么区别呢?
首先,就是因为汇编快,“快”其实不太准确,C语言虽然也会被编译成汇编,最后编译成二进制指令.
但是编译器生成的C语言指令会很长,有很多冗余的指令,而为了实现同样一个功能,程序员自己写的汇编会比较精简,指令少,优化多,自然也就更“快”了.
那么 _call_stub_entry 这个例程是何时生成的呢? 答案就在 generate_call_stub() 中,这个方法有点长,大家有点耐心.

下面大家会看到很多类似汇编指令的代码,其实这些不是指令,而是一个个用来生成汇编指令的方法.JVM是通过MacroAssembler来生成指令的.
我会将具体的执行过程通过注释的方式插入到代码中

```
/openjdk/hotspot/src/cpu/x86/vm/stubGenerator_x86_32.cpp
address generate_call_stub(address& return_address) {
    StubCodeMark mark(this,"StubRoutines","call_stub");
    //汇编器会将生成的例程在内存中线性排列. 所以取当前汇编器生成的上个例程最后一行汇编指令的地址,用来作为即将生成的新例程的首地址
    address start = __ pc();

    // stub code parameters / addresses
    assert(frame::entry_frame_call_wrapper_offset == 2,"adjust this code");
    bool  sse_save = false;
    const Address rsp_after_call(rbp,-4 * wordSize); // same as in generate_catch_exception()!
    const int     locals_count_in_bytes  (4*wordSize);

    //定义一些变量,用于保存一些调用方的信息,这四个参数放在被调用者堆栈中,即call_stub例程堆栈中.
    //所以相对于call_stub例程的栈基址(rbp)为负数(栈是向下增长).后面会用到这四个变量.
    const Address mxcsr_save    (rbp,-4 * wordSize);
    const Address saved_rbx     (rbp,-3 * wordSize);
    const Address saved_rsi     (rbp,-2 * wordSize);
    const Address saved_rdi     (rbp,-1 * wordSize);
    //传参,放在调用方堆栈中,所以相对call_stub例程的栈基址为正数,可以理解为调用方在调用call_stub例程之前,会将传参都放在自己的堆栈中.
    //这样call_stub例程中就可以直接基于栈基址进行偏移取用了.
    const Address result        (rbp, 3 * wordSize);
    const Address result_type   (rbp, 4 * wordSize);
    const Address method        (rbp, 5 * wordSize);
    const Address entry_point   (rbp, 6 * wordSize);
    const Address parameters    (rbp, 7 * wordSize);
    const Address parameter_size(rbp, 8 * wordSize);
    const Address thread        (rbp, 9 * wordSize); // same as in generate_catch_exception()!
    sse_save =  UseSSE > 0;

    //enter()对应的方法如下,用来保存调用方栈基址,并将call_stub栈基址更新为当前栈顶地址.
    //c语言编译器其实在调用方法前都会插入这件事,这里JVM相对于借用了这种思想.
 ---------------------------------------------
|        void MacroAssembler::enter() {       |
```

```
87  |            push(rbp);                       |
88  |            mov(rbp,rsp);                    |
89  |       }                                     |
90   ---------------------------------------------
91      __ enter();
92
93      //接下来计算并分配call_stub堆栈所需栈大小.
94      //先将参数数量放入rcx寄存器.
95      __ movptr(rcx,parameter_size);              // parameter counter
96      //shl用于左移,这里将rcx中的值左移了Interpreter::logStackElementSize位;
97      //在64位平台,logStackElementSize=3;在32位平台,logStackElementSize=2;
98      //所以在64位平台上,rcx = rcx * 8,即每个参数占用8字节;32位平台rcx = rcx *4 ,即每个参数占4个字节.
99      __ shlptr(rcx,Interpreter::logStackElementSize); // convert parameter count to bytes
100
101     // locals_count_in_bytes 在上面有定义:const int locals_count_in_bytes (4*wordSize);这四个字节其实就是上面用来保存调用方信息所占空间.
102     __ addptr(rcx,locals_count_in_bytes);       // reserve space for register saves
103
104     //rcx现在保存了计算好的所需栈空间,将保存栈顶地址的寄存器rsp减去rcx,即向下扩展栈.
105     __ subptr(rsp,rcx);
106
107     //引用《揭秘Java虚拟机》:为了加速内存寻址和回收,物理机器在分配堆栈空间时都会进行内存对齐,JVM也借用了这个思想.
108     //JVM中是按照两个字节,即16位进行对齐的:const int StackAlignmentInBytes = (2*wordSize);
109     __ andptr(rsp,-(StackAlignmentInBytes));    // Align stack
110
111     //将调用方的一些信息,保存到栈中分配的地址处,最后会再次还原到寄存器中
112     __ movtr(saved_rdi,rdi);
113     __ movptr(saved_rsi,rsi);
114     __ movptr(saved_rbx,rbx);
115     // save and initialize %mxcsr
116     if (sse_save) {
117       Label skip_ldmx;
118       __ stmxcsr(mxcsr_save);
119       __ movl(rax,mxcsr_save);
120       __ andl(rax,MXCSR_MASK);    // Only check control and mask bits
121       ExternalAddress mxcsr_std(StubRoutines::addr_mxcsr_std());
122       __ cmp32(rax,mxcsr_std);
123       __ jcc(Assembler::equal,skip_ldmx);
124       __ ldmxcsr(mxcsr_std);
125       __ bind(skip_ldmx);
126     }
127
128     // make sure the control word is correct.
129     __ fldcw(ExternalAddress(StubRoutines::addr_fpu_cntrl_wrd_std()));
130
131 #ifdef ASSERT
132     // make sure we have no pending exceptions
133     { Label L;
134       __ movptr(rcx,thread);
135       __ cmpptr(Address(rcx,Thread::pending_exception_offset()),(int32_t)NULL_WORD);
136       __ jcc(Assembler::equal,L);
137       __ stop("StubRoutines::call_stub: entered with pending exception");
138       __ bind(L);
139     }
140 #endif
141
142     //接下来就要进行参数压栈了;
143     Label parameters_done;
144     //检查参数数量是否为0,为0则直接跳到标号parameters_done处.
145     __ movl(rcx,parameter_size);  // parameter counter
146     __ testl(rcx,rcx);
147     __ jcc(Assembler::zero,parameters_done);
148
149     Label loop
150     //将参数首地址放到寄存器rdx中,并将rbx置0;
151     __ movptr(rdx,parameters);          // parameter pointer
152     __ xorptr(rbx,rbx);
153
154     //标号loop处
155     __ BIND(loop);
156
157     //此处开始循环;从最后一个参数倒序往前进行参数压栈,初始时,rcx=parameter_size;
158     //要注意,这里的参数是指java方法所需的参数,而不是call_stub例程所需参数!
159     //将(rdx + rcx * stackElementScale()- wordSize )移到 rax 中,(rdx + rcx * stackElementScale()- wordSize )指向了要压栈的参数.
160     __ movptr(rax,Address(rdx,rcx,Interpreter::stackElementScale(),-wordSize));
161     //再从rax中转移到(rsp + rbx * stackElementScale()) 处,expr_offset_in_bytes(0) = 0;
162     //这里是基于栈顶地址进行偏移寻址的,最后一个参数会被压到栈顶处,第一个参数会被压到rsp + (parameter_size-1)* stackElementScale()处.
163     __ movptr(Address(rsp,rbx,Interpreter::stackElementScale(),Interpreter::expr_offset_in_bytes(0)),rax); // store parameter
164     //更新rbx
165     __ increment(rbx);
166     //自减rcx,当rcx不为0时,继续跳往loop处循环执行.
167     __ decrement(rcx);
168     __ jcc(Assembler::notZero,loop);
169
170     //标号parameters_done处
171     __ BIND(parameters_done);
172
```

```
    //接下来要开始调用Java方法了.
    //将调用java方法的entry_point例程所需的一些参数保存到寄存器中
     __ movptr(rbx,method);           // get Method*
     __ movptr(rax,entry_point);      // get entry_point
     __ mov(rsi,rsp);                 // set sender sp
     //跳往entry_point例程执行
     __ call(rax);
     ......
  }
```

二:EntryPoint例程

上面最后会跳往 entry_point 例程执行.

现在有个新的问题,entry_point例程是个啥? 其实entry_point例程和call_stub例程一样,都是用汇编写的来执行java方法的工具.

我们回到JavaCalls::call_helper()中:

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

entry_point是从当前要执行的Java方法中获取的:

```
/openjdk/hotspot/src/share/vm/oops/method.hpp
volatile address from_interpreted_entry() const{
    return (address)OrderAccess::load_ptr_acquire(&_from_interpreted_entry);
}
```

那么 _from_interpreted_entry 是何时赋值的? method.hpp中有这样一个set方法:

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

我们来看看是何时调用了method的这个 set_interpreter_entry 方法:

```
// Called when the method_holder is getting linked. Setup entrypoints so the method
// is ready to be called from interpreter,compiler,and vtables.
void Method::link_method(methodHandle h_method,TRAPS) {
  ......
  address entry = Interpreter::entry_for_method(h_method);
  assert(entry != NULL,"interpreter entry must be non-null");
  // Sets both _i2i_entry and _from_interpreted_entry
  set_interpreter_entry(entry);
  ......
}
```

根据注释都可以得知,当方法链接时,会去设置方法的entry_point, entry_point是由Interpreter::entry_for_method(h_method)得到的:

```
static address entry_for_method(methodHandle m) { return entry_for_kind(method_kind(m)); }
```

首先通过method_kind()拿到方法类型,接着调用entry_for_kind():

```
static address entry_for_kind(MethodKind k){
    return _entry_table[k];
}
```

这里直接返回了_entry_table数组中对应方法类型索引的entry_point地址.

给数组中元素赋值专门有个方法:

```
void AbstractInterpreter::set_entry_for_kind(AbstractInterpreter::MethodKind kind,address entry) {
  _entry_table[kind] = entry;
}
```

那么何时会调用 set_entry_for_kind()呢? 答案就在 TemplateInterpreterGenerator::generate_all()中.

generate_all()会调用generate_method_entry()去生成每种方法的entry_point,所有Java方法的执行,都会通过对应类型的entry_point例程来辅助.

```
/openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_64.cpp
// Entry points
//
// Here we generate the various kind of entries into the interpreter.
// The two main entry type are generic bytecode methods and native
// call method.  These both come in synchronized and non-synchronized
// versions but the frame layout they create is very similar. The
// other method entry types are really just special purpose entries
// that are really entry and interpretation all in one. These are for
// trivial methods like accessor, empty, or special math methods.
//
// When control flow reaches any of the entry types for the interpreter
// the following holds ->
//
// Arguments:
//
// rbx: Method*
//
// Stack layout immediately at entry
//
// [ return address     ] <--- rsp
// [ parameter n        ]
//   ...
// [ parameter 1        ]
// [ expression stack   ] (caller's java expression stack)

// Assuming that we don't go to one of the trivial specialized entries
// the stack will look like below when we are ready to execute the
// first bytecode (or call the native routine). The register usage
// will be as the template based interpreter expects (see
// interpreter_amd64.hpp).
//
```

```
  // local variables follow incoming parameters immediately; i.e.
  // the return address is moved to the end of the locals).
  //
  // [ monitor entry      ] <--- rsp
  //   ...
  // [ monitor entry      ]
  // [ expr. stack bottom ]
  // [ saved r13          ]
  // [ current r14        ]
  // [ Method*            ]
  // [ saved ebp          ] <--- rbp
  // [ return address     ]
  // [ local variable m   ]
  //   ...
  // [ local variable 1   ]
  // [ parameter n        ]
  //   ...
  // [ parameter 1        ] <--- r14

  address AbstractInterpreterGenerator::generate_method_entry(AbstractInterpreter::MethodKind kind)
  {
    // determine code generation flags
    bool synchronized = false;
    address entry_point = NULL;
    InterpreterGenerator* ig_this = (InterpreterGenerator*)this;

    switch (kind) {
    case Interpreter::zerolocals             :                                                  break;
    case Interpreter::zerolocals_synchronized: synchronized = true;                             break;
    case Interpreter::native                 : entry_point = ig_this->generate_native_entry(false); break;
    case Interpreter::native_synchronized    : entry_point = ig_this->generate_native_entry(true);  break;
    case Interpreter::empty                  : entry_point = ig_this->generate_empty_entry();       break;
    case Interpreter::accessor               : entry_point = ig_this->generate_accessor_entry();    break;
    case Interpreter::abstract               : entry_point = ig_this->generate_abstract_entry();    break;

    case Interpreter::java_lang_math_sin     : // fall thru
    case Interpreter::java_lang_math_cos     : // fall thru
    case Interpreter::java_lang_math_tan     : // fall thru
    case Interpreter::java_lang_math_abs     : // fall thru
    case Interpreter::java_lang_math_log     : // fall thru
    case Interpreter::java_lang_math_log10   : // fall thru
    case Interpreter::java_lang_math_sqrt    : // fall thru
    case Interpreter::java_lang_math_pow     : // fall thru
    case Interpreter::java_lang_math_exp     : entry_point = ig_this->generate_math_entry(kind);      break;
    case Interpreter::java_lang_ref_reference_get
                                             : entry_point = ig_this->generate_Reference_get_entry(); break;
    case Interpreter::java_util_zip_CRC32_update
                                             : entry_point = ig_this->generate_CRC32_update_entry();  break;
    case Interpreter::java_util_zip_CRC32_updateBytes
                                             : // fall thru
    case Interpreter::java_util_zip_CRC32_updateByteBuffer
                                             : entry_point = ig_this->generate_CRC32_updateBytes_entry(kind); break;
    default:
      fatal(err_msg("unexpected method kind: %d", kind));
      break;
    }

    if (entry_point) {
      return entry_point;
    }

    return ig_this->generate_normal_entry(synchronized);
  }
```

现在就豁然开朗了,调用Java方法时,首先通过method找到对应的entry_point例程,并传递给call_stub例程.
call_stub准备好堆栈后,就开始前往entry_point处,entry_point例程就会开始执行传递给它的Java方法了.
在研究JVM时,我们不要把Java方法当作方法,要把它当作一个对象来对待,下次有时间在好好研究下entry_point例程.