https://cloud.tencent.com/developer/article/1395348
[inside hotspot] java方法调用的StubCode
众所周知jvm有invokestatic,invokedynamic,invokestatic,invokespecial,invokevirtual几条方法调用指令,每个负责调用不同的方法,
而这些方法调用落实到hotspot上都位于hotspot\src\share\vm\runtime\javaCalls.hpp的JavaCalls.

1. JavaCalls

```
class JavaCalls: AllStatic {
  static void call_helper(JavaValue* result, const methodHandle& method, JavaCallArguments* args, TRAPS);
 public:
  // call_special
  // ------------
  // The receiver must be first oop in argument list
  static void call_special(JavaValue* result, KlassHandle klass, Symbol* name, Symbol* signature, JavaCallArguments* args, TRAPS);
  static void call_special(JavaValue* result, Handle receiver, KlassHandle klass, Symbol* name, Symbol* signature, TRAPS); // No args
  static void call_special(JavaValue* result, Handle receiver, KlassHandle klass, Symbol* name, Symbol* signature, Handle arg1, TRAPS);
  static void call_special(JavaValue* result, Handle receiver, KlassHandle klass, Symbol* name, Symbol* signature, Handle arg1, Handle arg2,
  TRAPS);

  // virtual call
  // ------------
  // The receiver must be first oop in argument list
  static void call_virtual(JavaValue* result, KlassHandle spec_klass, Symbol* name, Symbol* signature, JavaCallArguments* args, TRAPS);
  static void call_virtual(JavaValue* result, Handle receiver, KlassHandle spec_klass, Symbol* name, Symbol* signature, TRAPS); // No args
  static void call_virtual(JavaValue* result, Handle receiver, KlassHandle spec_klass, Symbol* name, Symbol* signature, Handle arg1, TRAPS);
  static void call_virtual(JavaValue* result, Handle receiver, KlassHandle spec_klass, Symbol* name, Symbol* signature, Handle arg1, Handle
  arg2, TRAPS);

  // Static call
  // -----------
  static void call_static(JavaValue* result, KlassHandle klass, Symbol* name, Symbol* signature, JavaCallArguments* args, TRAPS);
  static void call_static(JavaValue* result, KlassHandle klass, Symbol* name, Symbol* signature, TRAPS);
  static void call_static(JavaValue* result, KlassHandle klass, Symbol* name, Symbol* signature, Handle arg1, TRAPS);
  static void call_static(JavaValue* result, KlassHandle klass, Symbol* name, Symbol* signature, Handle arg1, Handle arg2, TRAPS);
  static void call_static(JavaValue* result, KlassHandle klass, Symbol* name, Symbol* signature, Handle arg1, Handle arg2, Handle arg3,
  TRAPS);

  // Low-level interface
  static void call(JavaValue* result, const methodHandle& method, JavaCallArguments* args, TRAPS);
};
```

上面的方法是自解释的,对应各自的invoke*指令,这些call_static,call_virtual内部调用了call()函数:

```
void JavaCalls::call(JavaValue* result, const methodHandle& method, JavaCallArguments* args, TRAPS) {
  assert(THREAD->is_Java_thread(), "only JavaThreads can make JavaCalls");
  os::os_exception_wrapper(call_helper, result, method, args, THREAD);
}
```

call()只是简单检查了一下线程信息,以及根据平台比如windows会使用结构化异常(SEH)包裹call_helper,最终执行方法调用的还是call_helper.

```
void JavaCalls::call_helper(JavaValue* result, const methodHandle& method, JavaCallArguments* args, TRAPS) {
  ...
  // 如果当前方法为空,则直接返回
  if (method->is_empty_method()) {
    assert(result->get_type() == T_VOID, "an empty method must return a void value");
    return;
  }
  ...
  //根据情况决定是否编译该方法,JIT和-Xcomp都有可能触发它
  CompilationPolicy::compile_if_required(method, CHECK);

  // 解释器入口点
  address entry_point = method->from_interpreted_entry();
  if (JvmtiExport::can_post_interpreter_events() && thread->is_interp_only_mode()) {
    entry_point = method->interpreter_entry();
  }

  // 确定返回值类型
  BasicType result_type = runtime_type_from(result);
  bool oop_result_flag = (result->get_type() == T_OBJECT || result->get_type() == T_ARRAY);

  // 返回值地址
  intptr_t* result_val_address = (intptr_t*)(result->get_value_addr());

  // 确定receiver,如果是static函数就没有receiver
  Handle receiver = (!method->is_static()) ? args->receiver() : Handle();

  if (!thread->stack_guards_enabled()) {
    thread->reguard_stack();
  }

  // 确认当前sp是否到达ShadowPages,即是否会触发栈溢出错误
  address sp = os::current_stack_pointer();
  if (!os::stack_shadow_pages_available(THREAD, method, sp)) {
    // Throw stack overflow exception with preinitialized exception.
    Exceptions::throw_stack_overflow_exception(THREAD, __FILE__, __LINE__, method);
    return;
  } else {
    // Touch pages checked if the OS needs them to be touched to be mapped.
```

```
      os::map_stack_shadow_pages(sp);
    }

    // 执行调用
    { JavaCallWrapper link(method, receiver, result, CHECK);
      { HandleMark hm(thread);  // HandleMark used by HandleMarkCleaner

        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
        );

        result = link.result();  // circumvent MS C++ 5.0 compiler bug (result is clobbered across call)
        // Preserve oop return value across possible gc points
        if (oop_result_flag) {
          thread->set_vm_result((oop) result->get_jobject());
        }
      }
    }

    // 设置返回值
    if (oop_result_flag) {
      result->set_jobject((jobject)thread->vm_result());
      thread->set_vm_result(NULL);
    }
  }
```

call_helper又可以分为两步,第一步判断一下方法是否为空,是否可以JIT编译,是否还有栈空间可以等.
第二步StubRoutines::call_stub()实际调用os+cpu限定的方法.
这个StubRoutines::call_stub()返回的是一个函数指针,指向的是平台特定的方法,所以这段代码:

```
        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
        );
```

call_stub()返回一个函数指针,指向依赖于操作系统和cpu架构的特定的方法.
原因很简单,要执行native代码,得看看是什么cpu架构以便确定寄存器,看看什么os以便确定ABI.
然后传递8个参数到这个方法里面并执行这个方法.
那么这个方法是什么呢？进入stubRoutines.cpp便知是StubRoutines::_call_stub_entry.

2. windows+x86_64的stubGenerator
以x64为例,hotspot\src\cpu\x86\vm\stubGenerator_x86_64.cpp的generate_call_stub()会负责初始化StubRoutines::_call_stub_entry函数.
使用参数命令 -XX:+UnlockDiagnosticVMOptions -XX:+PrintStubCode 可以输出generate_call_stub方法生成的汇编,对照着看非常舒服:

```
 address generate_call_stub(address& return_address) {
    assert((int)frame::entry_frame_after_call_words == -(int)rsp_after_call_off + 1 &&
           (int)frame::entry_frame_call_wrapper_offset == (int)call_wrapper_off,
           "adjust this code");
    StubCodeMark mark(this, "StubRoutines", "call_stub");
    address start = __ pc();

    // same as in generate_catch_exception()!
    const Address rsp_after_call(rbp, rsp_after_call_off * wordSize);

    const Address call_wrapper  (rbp, call_wrapper_off   * wordSize);
    const Address result        (rbp, result_off         * wordSize);
    const Address result_type   (rbp, result_type_off    * wordSize);
    const Address method        (rbp, method_off         * wordSize);
    const Address entry_point   (rbp, entry_point_off    * wordSize);
    const Address parameters    (rbp, parameters_off     * wordSize);
    const Address parameter_size(rbp, parameter_size_off * wordSize);

    // same as in generate_catch_exception()!
    const Address thread        (rbp, thread_off         * wordSize);

    const Address r15_save(rbp, r15_off * wordSize);
    const Address r14_save(rbp, r14_off * wordSize);
    const Address r13_save(rbp, r13_off * wordSize);
    const Address r12_save(rbp, r12_off * wordSize);
    const Address rbx_save(rbp, rbx_off * wordSize);

    // stub code
    __ enter();
```

```
170       __ subptr(rsp, -rsp_after_call_off * wordSize);
171   StubRoutines::call_stub [0x0000026b0a5d09d7, 0x0000026b0a5d0b44[ (365 bytes)
172     0x0000026b0a5d09d7: push   %rbp
173     0x0000026b0a5d09d8: mov    %rsp,%rbp
174     0x0000026b0a5d09db: sub    $0x1d8,%rsp
175       // save register parameters
176   #ifndef _WIN64
177       __ movptr(parameters,   c_rarg5); // parameters
178       __ movptr(entry_point,  c_rarg4); // entry_point
179   #endif
180
181       __ movptr(method,       c_rarg3); // method
182       __ movl(result_type,  c_rarg2);   // result type
183       __ movptr(result,       c_rarg1); // result
184       __ movptr(call_wrapper, c_rarg0); // call wrapper
185     // r9方法,r8d返回值类型,rdx,返回值,rcx即JavaCallsWrapper
186     0x0000026b0a5d09e2: mov    %r9,0x28(%rbp)
187     0x0000026b0a5d09e6: mov    %r8d,0x20(%rbp)
188     0x0000026b0a5d09ea: mov    %rdx,0x18(%rbp)
189     0x0000026b0a5d09ee: mov    %rcx,0x10(%rbp)
190       // save regs belonging to calling function
191       __ movptr(rbx_save, rbx);
192       __ movptr(r12_save, r12);
193       __ movptr(r13_save, r13);
194       __ movptr(r14_save, r14);
195       __ movptr(r15_save, r15);
196       if (UseAVX > 2) {
197         __ movl(rbx, 0xffff);
198         __ kmovwl(k1, rbx);
199       }
200   #ifdef _WIN64
201       int last_reg = 15;
202       if (UseAVX > 2) {
203         last_reg = 31;
204       }
205       if (VM_Version::supports_evex()) {
206         for (int i = xmm_save_first; i <= last_reg; i++) {
207           __ vextractf32x4(xmm_save(i), as_XMMRegister(i), 0);
208         }
209       } else {
210         for (int i = xmm_save_first; i <= last_reg; i++) {
211           __ movdqu(xmm_save(i), as_XMMRegister(i));
212         }
213       }
214     // caller-save 寄存器
215     0x0000026b0a5d09f2: mov    %rbx,-0x8(%rbp)
216     0x0000026b0a5d09f6: mov    %r12,-0x20(%rbp)
217     0x0000026b0a5d09fa: mov    %r13,-0x28(%rbp)
218     0x0000026b0a5d09fe: mov    %r14,-0x30(%rbp)
219     0x0000026b0a5d0a02: mov    %r15,-0x38(%rbp)
220
221     0x0000026b0a5d0a06: vmovdqu %xmm6,-0x48(%rbp)
222     0x0000026b0a5d0a0b: vmovdqu %xmm7,-0x58(%rbp)
223     0x0000026b0a5d0a10: vmovdqu %xmm8,-0x68(%rbp)
224     0x0000026b0a5d0a15: vmovdqu %xmm9,-0x78(%rbp)
225     0x0000026b0a5d0a1a: vmovdqu %xmm10,-0x88(%rbp)
226     0x0000026b0a5d0a22: vmovdqu %xmm11,-0x98(%rbp)
227     0x0000026b0a5d0a2a: vmovdqu %xmm12,-0xa8(%rbp)
228     0x0000026b0a5d0a32: vmovdqu %xmm13,-0xb8(%rbp)
229     0x0000026b0a5d0a3a: vmovdqu %xmm14,-0xc8(%rbp)
230     0x0000026b0a5d0a42: vmovdqu %xmm15,-0xd8(%rbp)
231       const Address rdi_save(rbp, rdi_off * wordSize);
232       const Address rsi_save(rbp, rsi_off * wordSize);
233       __ movptr(rsi_save, rsi);
234       __ movptr(rdi_save, rdi);
235     // rsi rdi
236     0x0000026b0a5d0a4a: mov    %rsi,-0x10(%rbp)
237     0x0000026b0a5d0a4e: mov    %rdi,-0x18(%rbp)
238       // Load up thread register
239       __ movptr(r15_thread, thread);
240       __ reinit_heapbase();
241     // 线程寄存器
242     0x0000026b0a5d0a52: mov    0x48(%rbp),%r15
243     0x0000026b0a5d0a56: movabs $0x7ffe4c5b2be8,%r10
244     0x0000026b0a5d0a60: mov    (%r10),%r12
245       // pass parameters if any
246       BLOCK_COMMENT("pass parameters if any");
247       Label parameters_done;
248       __ movl(c_rarg3, parameter_size);
249       __ testl(c_rarg3, c_rarg3);
250       __ jcc(Assembler::zero, parameters_done);
251
252       Label loop;
253       __ movptr(c_rarg2, parameters);       // parameter pointer
254       __ movl(c_rarg1, c_rarg3);            // parameter counter is in c_rarg1
255       __ BIND(loop);
```

```
256       __ movptr(rax, Address(c_rarg2, 0));// get parameter
257       __ addptr(c_rarg2, wordSize);       // advance to next parameter
258       __ decrementl(c_rarg1);             // decrement counter
259       __ push(rax);                       // pass parameter
260       __ jcc(Assembler::notZero, loop);
261     // 这里是个循环,用于传递参数,相当于
262     // while(r9d){
263     //    rax = *arg
264     //    push_arg(rax)
265     //    arg++;  // ptr++
266     //    r9d--;  // counter--
267     // }
268     0x0000026b0a5d0a63: mov    0x40(%rbp),%r9d
269     0x0000026b0a5d0a67: test   %r9d,%r9d
270     0x0000026b0a5d0a6a: je     0x0000026b0a5d0a83
271     0x0000026b0a5d0a70: mov    0x38(%rbp),%r8
272     0x0000026b0a5d0a74: mov    %r9d,%edx
273     0x0000026b0a5d0a77: mov    (%r8),%rax
274     0x0000026b0a5d0a7a: add    $0x8,%r8
275     0x0000026b0a5d0a7e: dec    %edx
276     0x0000026b0a5d0a80: push   %rax
277     0x0000026b0a5d0a81: jne    0x0000026b0a5d0a77
278       // call Java function
279       __ BIND(parameters_done);
280       __ movptr(rbx, method);             // get Method*
281       __ movptr(c_rarg1, entry_point);    // get entry_point
282       __ mov(r13, rsp);                   // set sender sp
283       BLOCK_COMMENT("call Java function");
284       __ call(c_rarg1);
285     // [!!]调用java方法
286     0x0000026b0a5d0a83: mov    0x28(%rbp),%rbx
287     0x0000026b0a5d0a87: mov    0x30(%rbp),%rdx
288     0x0000026b0a5d0a8b: mov    %rsp,%r13
289     0x0000026b0a5d0a8e: callq  *%rdx
290       BLOCK_COMMENT("call_stub_return_address:");
291       return_address = __ pc();
292
293       // store result depending on type (everything that is not
294       // T_OBJECT, T_LONG, T_FLOAT or T_DOUBLE is treated as T_INT)
295       __ movptr(c_rarg0, result);
296       Label is_long, is_float, is_double, exit;
297       __ movl(c_rarg1, result_type);
298       __ cmpl(c_rarg1, T_OBJECT);
299       __ jcc(Assembler::equal, is_long);
300       __ cmpl(c_rarg1, T_LONG);
301       __ jcc(Assembler::equal, is_long);
302       __ cmpl(c_rarg1, T_FLOAT);
303       __ jcc(Assembler::equal, is_float);
304       __ cmpl(c_rarg1, T_DOUBLE);
305       __ jcc(Assembler::equal, is_double);
306
307       // handle T_INT case
308       __ movl(Address(c_rarg0, 0), rax);
309
310       __ BIND(exit);
311
312       // pop parameters
313       __ lea(rsp, rsp_after_call);
314     // 储存java方法返回值并弹出参数,这里弹出操作即移动一下rsp指针
315     0x0000026b0a5d0a90: mov    0x18(%rbp),%rcx
316     0x0000026b0a5d0a94: mov    0x20(%rbp),%edx
317     0x0000026b0a5d0a97: cmp    $0xc,%edx
318     0x0000026b0a5d0a9a: je     0x0000026b0a5d0b30
319     0x0000026b0a5d0aa0: cmp    $0xb,%edx
320     0x0000026b0a5d0aa3: je     0x0000026b0a5d0b30
321     0x0000026b0a5d0aa9: cmp    $0x6,%edx
322     0x0000026b0a5d0aac: je     0x0000026b0a5d0b35
323     0x0000026b0a5d0ab2: cmp    $0x7,%edx
324     0x0000026b0a5d0ab5: je     0x0000026b0a5d0b3b
325     0x0000026b0a5d0abb: mov    %eax,(%rcx)
326     0x0000026b0a5d0abd: lea    -0x1d8(%rbp),%rsp
327       // restore regs belonging to calling function
328   #ifdef _WIN64
329       // emit the restores for xmm regs
330       if (VM_Version::supports_evex()) {
331         for (int i = xmm_save_first; i <= last_reg; i++) {
332           __ vinsertf32x4(as_XMMRegister(i), as_XMMRegister(i), xmm_save(i), 0);
333         }
334       } else {
335         for (int i = xmm_save_first; i <= last_reg; i++) {
336           __ movdqu(as_XMMRegister(i), xmm_save(i));
337         }
338       }
339   #endif
340       __ movptr(r15, r15_save);
341       __ movptr(r14, r14_save);
```

```
      __ movptr(r13, r13_save);
      __ movptr(r12, r12_save);
      __ movptr(rbx, rbx_save);

      __ movptr(rdi, rdi_save);
      __ movptr(rsi, rsi_save);
    // 恢复之前保存的caller-save寄存器
    0x0000026b0a5d0ac4: vmovdqu -0x48(%rbp),%xmm6
    0x0000026b0a5d0ac9: vmovdqu -0x58(%rbp),%xmm7
    0x0000026b0a5d0ace: vmovdqu -0x68(%rbp),%xmm8
    0x0000026b0a5d0ad3: vmovdqu -0x78(%rbp),%xmm9
    0x0000026b0a5d0ad8: vmovdqu -0x88(%rbp),%xmm10
    0x0000026b0a5d0ae0: vmovdqu -0x98(%rbp),%xmm11
    0x0000026b0a5d0ae8: vmovdqu -0xa8(%rbp),%xmm12
    0x0000026b0a5d0af0: vmovdqu -0xb8(%rbp),%xmm13
    0x0000026b0a5d0af8: vmovdqu -0xc8(%rbp),%xmm14
    0x0000026b0a5d0b00: vmovdqu -0xd8(%rbp),%xmm15
    0x0000026b0a5d0b08: mov    -0x38(%rbp),%r15
    0x0000026b0a5d0b0c: mov    -0x30(%rbp),%r14
    0x0000026b0a5d0b10: mov    -0x28(%rbp),%r13
    0x0000026b0a5d0b14: mov    -0x20(%rbp),%r12
    0x0000026b0a5d0b18: mov    -0x8(%rbp),%rbx
    0x0000026b0a5d0b1c: mov    -0x18(%rbp),%rdi
    0x0000026b0a5d0b20: mov    -0x10(%rbp),%rsi
      // restore rsp
      __ addptr(rsp, -rsp_after_call_off * wordSize);

      // return
      __ pop(rbp);
      __ ret(0);
    // 结束__call_stub_entry这个函数
    0x0000026b0a5d0b24: add    $0x1d8,%rsp
    0x0000026b0a5d0b2b: vzeroupper
    0x0000026b0a5d0b2e: pop    %rbp
    0x0000026b0a5d0b2f: retq
  下面这段代码逻辑上属于之前的存储java方法的返回值.
  随便举个例子 0x0000026b0a5d0b30 这个地址正是之前存放java方法的代码段je     0x0000026b0a5d0b30所跳之处.
  只是放到了最后而已(不过我也不知道为什么要放到这后面)

      // handle return types different from T_INT
      __ BIND(is_long);
      __ movq(Address(c_rarg0, 0), rax);
      __ jmp(exit);

      __ BIND(is_float);
      __ movflt(Address(c_rarg0, 0), xmm0);
      __ jmp(exit);

      __ BIND(is_double);
      __ movdbl(Address(c_rarg0, 0), xmm0);
      __ jmp(exit);

      return start;
    }
    0x0000026b0a5d0b30: mov    %rax,(%rcx)
    0x0000026b0a5d0b33: jmp    0x0000026b0a5d0abd
    0x0000026b0a5d0b35: vmovss %xmm0,(%rcx)
    0x0000026b0a5d0b39: jmp    0x0000026b0a5d0abd
    0x0000026b0a5d0b3b: vmovsd %xmm0,(%rcx)
    0x0000026b0a5d0b3f: jmpq   0x0000026b0a5d0abd
  对照汇编看非常清晰,不过也可以看到它建立了栈帧结构,但它还是没有执行java代码,而是使用callq *rdx进行的,
   这也是为什么它叫做stub的原因.
   另外上面的栈帧里面内容比较多,[rsp+xx]存放什么内容啊这些比较难记,已经归纳好的结构可以参见代码注释:

    // Windowsx86_64平台
    //
    //    注意c_rarg\d 表示寄存器,method/result表示内存地址[rbp+\d]
    //
    //    c_rarg0:   call wrapper address                   address
    //    c_rarg1:   result                                 address
    //    c_rarg2:   result type                            BasicType
    //    c_rarg3:   method                                 Method*
    //    48(rbp): (interpreter) entry point              address
    //    56(rbp): parameters                             intptr_t*
    //    64(rbp): parameter size (in words)              int
    //    72(rbp): thread                                 Thread*
    //
    //     [ return_from_Java     ] <--- 这里执行callq调用java方法.压入返回地址,跳转到java方法,也就是说↑上面的部分就是java方法使用的栈帧了
    //     [ argument word n      ] <--- 循环传递的java方法实参
    //      ...
    // -60 [ argument word 1      ]
    // -59 [ saved xmm31          ] <--- rsp after_call
    //     [ saved xmm16-xmm30    ]
    // -27 [ saved xmm15          ]
    //     [ saved xmm7-xmm14     ]
    //  -9 [ saved xmm6           ]
```

```
428     //  -7 [ saved r15            ]
429     //  -6 [ saved r14            ]
430     //  -5 [ saved r13            ]
431     //  -4 [ saved r12            ]
432     //  -3 [ saved rdi            ]
433     //  -2 [ saved rsi            ]
434     //  -1 [ saved rbx            ]
435     //   0 [ saved rbp            ] <--- rbp
436     //   1 [ return address       ] <--- last rbp
437     //   2 [ call wrapper         ] <--- arg0
438     //   3 [ result               ] <--- arg1
439     //   4 [ result type          ] <--- arg2
440     //   5 [ method               ] <--- arg3
441     //   6 [ entry point          ] <--- arg4
442     //   7 [ parameters           ] <--- arg5
443     //   8 [ parameter size       ] <--- arg6
444     //   9 [ thread               ] <--- arg7
445   这8个arg正是之前传递给函数指针指向的函数的实参：
446
447         StubRoutines::call_stub()(
448            (address)&link,
449            // (intptr_t*)&(result->_value), // see NOTE above (compiler problem)
450            result_val_address,          // see NOTE above (compiler problem)
451            result_type,
452            method(),
453            entry_point,
454            args->parameters(),
455            args->size_of_parameters(),
456            CHECK
457          );
```