https://liuxiaofei.com.cn/blog/call_stub-jvm-java%E8%B0%83%E7%94%A8%E7%9A%84%E5%85%A5%E5%8F%A3/
call_stub-JVM Java调用的入口

Content:
调用堆栈:
下面通过代码注释的方式来介绍call_helper这个方法
call_stub方法的创建
call_stub代码生成

调用堆栈

JavaCalls::call_helper() at javaCalls.cpp:393 0x7ffff70f3600
os::os_exception_wrapper() at os_linux.cpp:5,104 0x7ffff7372b40
JavaCalls::call() at javaCalls.cpp:307 0x7ffff70f2fe6
InstanceKlass::call_class_initializer_impl() at instanceKlass.cpp:1,193 0x7ffff7088edf
InstanceKlass::call_class_initializer() at instanceKlass.cpp:1,161 0x7ffff7088c6d
InstanceKlass::initialize_impl() at instanceKlass.cpp:897 0x7ffff7087a40
InstanceKlass::initialize() at instanceKlass.cpp:557 0x7ffff7085f3e
InstanceKlass::initialize_impl() at instanceKlass.cpp:840 0x7ffff70874d7
InstanceKlass::initialize() at instanceKlass.cpp:557 0x7ffff7085f3e
initialize_class() at thread.cpp:983 0x7ffff74c5244
Threads::create_vm() at thread.cpp:3,497 0x7ffff74cc7df
JNI_CreateJavaVM() at jni.cpp:5,166 0x7ffff7134f13

下面通过代码注释的方式来介绍call_helper这个方法

```
void JavaCalls::call_helper(JavaValue* result, methodHandle* m, JavaCallArguments* args, TRAPS) {
  //JVM通过OOP-Klass的方式来对Java的类实例-类进行描述。Handle是对OOP和Klass的间接引用。
  //Handle类里重载了操作符()和->,通过method()或method->可以获取到methodOop。
  methodHandle method = *m;
  JavaThread* thread = (JavaThread*)THREAD;
  assert(thread->is_Java_thread(), "must be called by a java thread");
  assert(method.not_null(), "must have a method to call");
  assert(!SafepointSynchronize::is_at_safepoint(), "call to Java code during VM operation");
  assert(!thread->handle_area()->no_handle_mark_active(), "cannot call out to Java here");

  CHECK_UNHANDLED_OOPS_ONLY(thread->clear_unhandled_oops();)

  // Verify the arguments

  if (CheckJNICalls)  {
    args->verify(method, result->get_type(), thread);
  }
  else debug_only(args->verify(method, result->get_type(), thread));

  // Ignore call if method is empty
  if (method->is_empty_method()) {
    assert(result->get_type() == T_VOID, "an empty method must return a void value");
    return;
  }

#ifdef ASSERT
  { InstanceKlass* holder = method->method_holder();
    // A klass might not be initialized since JavaCall's might be used during the executing of
    // the . For example, a Thread.start might start executing on an object that is
    // not fully initialized! (bad Java programming style)
    assert(holder->is_linked(), "rewritting must have taken place");
  }
#endif

  assert(!thread->is_Compiler_thread(), "cannot compile from the compiler");
  if (CompilationPolicy::must_be_compiled(method)) {
    CompileBroker::compile_method(method, InvocationEntryBci,
                                  CompilationPolicy::policy()->initial_compile_level(),
                                  methodHandle(), 0, "must_be_compiled", CHECK);
  }

  //这儿获取method的entry_point,entry_point就是为Java方法调用准备栈桢,并把代码调用指针指向method的第一个字节码的内存地址
  //另一篇文章会详细说明entry_point方法的创建。
  //entry_point相当于是method的封装,不同的method类型有不同的entry_point。
  // Since the call stub sets up like the interpreter we call the from_interpreted_entry
  // so we can go compiled via a i2c. Otherwise initial entry method will always
  // run interpreted.
  address entry_point = method->from_interpreted_entry();
  if (JvmtiExport::can_post_interpreter_events() && thread->is_interp_only_mode()) {
    entry_point = method->interpreter_entry();
  }

  // Figure out if the result value is an oop or not (Note: This is a different value
  // than result_type. result_type will be T_INT of oops. (it is about size)
  BasicType result_type = runtime_type_from(result);
  bool oop_result_flag = (result->get_type() == T_OBJECT || result->get_type() == T_ARRAY);

  // NOTE: if we move the computation of the result_val_address inside
  // the call to call_stub, the optimizer produces wrong code.
  intptr_t* result_val_address = (intptr_t*)(result->get_value_addr());
```

```
87
88    // Find receiver
89    Handle receiver = (!method->is_static()) ? args->receiver() : Handle();
90
91    // When we reenter Java, we need to reenable the yellow zone which
92    // might already be disabled when we are in VM.
93    if (thread->stack_yellow_zone_disabled()) {
94      thread->reguard_stack();
95    }
96
97    // Check that there are shadow pages available before changing thread state
98    // to Java
99    if (!os::stack_shadow_pages_available(THREAD, method)) {
100     // Throw stack overflow exception with preinitialized exception.
101     Exceptions::throw_stack_overflow_exception(THREAD, __FILE__, __LINE__, method);
102     return;
103   } else {
104     // Touch pages checked if the OS needs them to be touched to be mapped.
105     os::bang_stack_shadow_pages();
106   }
107
108   // do call
109   { JavaCallWrapper link(method, receiver, result, CHECK);
110     { HandleMark hm(thread);  // HandleMark used by HandleMarkCleaner
111       //通过 call_stub->entry_point->method 的调用链,完成Java方法的调用
112       StubRoutines::call_stub()(
113         (address)&link,//call_stub调用完后,返回值通过link指针带回来
114         // (intptr_t*)&(result->_value), // see NOTE above (compiler problem)
115         result_val_address,          // see NOTE above (compiler problem)返回值地址
116         result_type,//返回类型
117         method(),//Java的方法实例
118         entry_point,//Wrap在method上,不同方法类型的调用入口
119         args->parameters(),//method的参数
120         args->size_of_parameters(),//参数大小
121         CHECK
122       );
123
124       result = link.result();  // circumvent MS C++ 5.0 compiler bug (result is clobbered across call)
125       // Preserve oop return value across possible gc points
126       if (oop_result_flag) {
127         thread->set_vm_result((oop) result->get_jobject());
128       }
129     }
130   } // Exit JavaCallWrapper (can block - potential return oop must be preserved)
131
132   // Check if a thread stop or suspend should be executed
133   // The following assert was not realistic.  Thread.stop can set that bit at any moment.
134   //assert(!thread->has_special_runtime_exit_condition(), "no async. exceptions should be installed");
135
136   // Restore possible oop return
137   if (oop_result_flag) {
138     result->set_jobject((jobject)thread->vm_result());
139     thread->set_vm_result(NULL);
140   }
141 }
142
143 call_stub方法的创建
144 -------------------
145   /hotspot/src/share/vm/runtime/stubRoutines.hpp:264
146   // Calls to Java
147   typedef void (*CallStub)(
148     address   link,
149     intptr_t* result,
150     BasicType result_type,
151     Method* method,
152     address   entry_point,
153     intptr_t* parameters,
154     int       size_of_parameters,
155     TRAPS
156   );
157
158   /hotspot/src/share/vm/runtime/stubRoutines.hpp:275
159   static CallStub call_stub() { return CAST_TO_FN_PTR(CallStub, _call_stub_entry); }
160
161 call_stub其实是由一个函数指针_call_stub_entry转换而来的,即_call_stub_entry指针指向了内存中的一个函数.
162 它是怎么形成的呢？它是在JVM启动的时候通过如下生成的.
163
164 StubGenerator::generate_call_stub() at stubGenerator_x86_64.cpp:221 0x7ffff745af7e
165 StubGenerator::generate_initial() at stubGenerator_x86_64.cpp:3,809 0x7ffff74719e2
166 StubGenerator::StubGenerator() at stubGenerator_x86_64.cpp:3,912 0x7ffff7471e97
167 StubGenerator_generate() at stubGenerator_x86_64.cpp:3,918 0x7ffff745ad3b
168 StubRoutines::initialize1() at stubRoutines.cpp:163 0x7ffff7471f87
169 stubRoutines_init1() at stubRoutines.cpp:297 0x7ffff747295b
170 init_globals() at init.cpp:101 0x7ffff7080d06
171 Threads::create_vm() at thread.cpp:3,424 0x7ffff74cc509
172 JNI_CreateJavaVM() at jni.cpp:5,166 0x7ffff7134f13
```

```
173
174 call_stub代码生成
175 -----------------
176 同上,通过代码注释进行说明。
177
178   /hotspot/src/cpu/x86/vm/stubGenerator_x86_64.cpp:3991
179   StubRoutines::_call_stub_entry = generate_call_stub(StubRoutines::_call_stub_return_address);
180
181   //call_stub方法的生成逻辑是直接向内存空间写入汇编代码
182   /hotspot/src/cpu/x86/vm/stubGenerator_x86_64.cpp:217
183   address generate_call_stub(address& return_address) {
184     assert((int)frame::entry_frame_after_call_words == -(int)rsp_after_call_off + 1 &&
185            (int)frame::entry_frame_call_wrapper_offset == (int)call_wrapper_off,
186            "adjust this code");
187     StubCodeMark mark(this, "StubRoutines", "call_stub");
188     address start = __ pc();//获取当前代码段地址,从此地址开始写入stub方法
189
190   //以下是以Linux的call_stub桢为例(line 190-221)
191   /hotspot/src/cpu/x86/vm/stubGenerator_x86_64.cpp:95
192   // Call stubs are used to call Java from C
193   // Linux Arguments:
194   //    c_rarg0:   call wrapper address                 address
195   //    c_rarg1:   result                               address
196   //    c_rarg2:   result type                          BasicType
197   //    c_rarg3:   method                               Method*
198   //    c_rarg4:   (interpreter) entry point            address
199   //    c_rarg5:   parameters                           intptr_t*
200   //    16(rbp): parameter size (in words)              int
201   //    24(rbp): thread                                 Thread*
202   //
203   //     [ return_from_Java     ] <--- rsp                    |
204   //     [ argument word n      ]                             |
205   //      ...                                                 |
206   // -12 [ argument word 1      ]                             |
207   // -11 [ saved r15            ] <--- rsp_after_call         |
208   // -10 [ saved r14            ]                             |
209   //  -9 [ saved r13            ]                             |
210   //  -8 [ saved r12            ]                             |
211   //  -7 [ saved rbx            ]                             |
212   //  -6 [ call wrapper         ]                             |call_stub栈桢
213   //  -5 [ result               ]                             |
214   //  -4 [ result type          ]                             |
215   //  -3 [ method               ]                             |
216   //  -2 [ entry point          ]                             |
217   //  -1 [ parameters           ]                             |
218   //   0 [ saved rbp            ] <--- rbp                    |
219   //   1 [ return address       ]                             |
220   //   2 [ parameter size       ]                             |
221   //   3 [ thread               ]                             |
222
223     // same as in generate_catch_exception()!
224     const Address rsp_after_call(rbp, rsp_after_call_off * wordSize);
225     // 设置相应参数的地址,注意堆栈是向低地址方向增长的,地址存在地址低的一端。
226     const Address call_wrapper  (rbp, call_wrapper_off   * wordSize);
227     const Address result        (rbp, result_off         * wordSize);
228     const Address result_type   (rbp, result_type_off    * wordSize);
229     const Address method        (rbp, method_off         * wordSize);
230     const Address entry_point   (rbp, entry_point_off    * wordSize);
231     const Address parameters    (rbp, parameters_off     * wordSize);
232     const Address parameter_size(rbp, parameter_size_off * wordSize);
233
234     // same as in generate_catch_exception()!
235     const Address thread        (rbp, thread_off         * wordSize);
236
237     const Address r15_save(rbp, r15_off * wordSize);
238     const Address r14_save(rbp, r14_off * wordSize);
239     const Address r13_save(rbp, r13_off * wordSize);
240     const Address r12_save(rbp, r12_off * wordSize);
241     const Address rbx_save(rbp, rbx_off * wordSize);
242
243     // stub code
244     __ enter(); //函数的实现在/hotspot/src/cpu/x86/vm/macroAssembler_x86.cpp:2991 push(rbp);mov(rbp, rsp);
245                 //即保存rbp的值到堆栈供函数返回使用,把rsp的值保存到rbp,即栈顶地址成为新的基址。
246     __ subptr(rsp, -rsp_after_call_off * wordSize);//堆栈增长 rsp_after_call_off 个字空间,以供下面使用
247
248     // save register parameters 把参数压栈到上面设定的地址
249   #ifndef _WIN64
250     __ movptr(parameters,   c_rarg5); // parameters
251     __ movptr(entry_point,  c_rarg4); // entry_point
252   #endif
253
254     __ movptr(method,       c_rarg3); // method
255     __ movl(result_type,    c_rarg2); // result type
256     __ movptr(result,       c_rarg1); // result
257     __ movptr(call_wrapper, c_rarg0); // call wrapper
258
```

```
259       // save regs belonging to calling function保存寄存器里面的值,函数调用完后需要恢复,别人的数据不能动。
260       __ movptr(rbx_save, rbx);
261       __ movptr(r12_save, r12);
262       __ movptr(r13_save, r13);
263       __ movptr(r14_save, r14);
264       __ movptr(r15_save, r15);
265   #ifdef _WIN64
266       for (int i = 6; i <= 15; i++) {
267         __ movdqu(xmm_save(i), as_XMMRegister(i));
268       }
269
270       const Address rdi_save(rbp, rdi_off * wordSize);
271       const Address rsi_save(rbp, rsi_off * wordSize);
272
273       __ movptr(rsi_save, rsi);
274       __ movptr(rdi_save, rdi);
275   #else
276       const Address mxcsr_save(rbp, mxcsr_off * wordSize);
277       {
278         Label skip_ldmx;
279         __ stmxcsr(mxcsr_save);
280         __ movl(rax, mxcsr_save);
281         __ andl(rax, MXCSR_MASK);    // Only check control and mask bits
282         ExternalAddress mxcsr_std(StubRoutines::addr_mxcsr_std());
283         __ cmp32(rax, mxcsr_std);
284         __ jcc(Assembler::equal, skip_ldmx);
285         __ ldmxcsr(mxcsr_std);
286         __ bind(skip_ldmx);
287       }
288   #endif
289
290       // Load up thread register
291       __ movptr(r15_thread, thread);
292       __ reinit_heapbase();
293
294   #ifdef ASSERT
295       // make sure we have no pending exceptions
296       {
297         Label L;
298         __ cmpptr(Address(r15_thread, Thread::pending_exception_offset()), (int32_t)NULL_WORD);
299         __ jcc(Assembler::equal, L);
300         __ stop("StubRoutines::call_stub: entered with pending exception");
301         __ bind(L);
302       }
303   #endif
304
305       // pass parameters if any 把函数的参数依次传递到当前堆栈上
306       BLOCK_COMMENT("pass parameters if any");
307       Label parameters_done;//把参数大小复制到 c_rarg3,如果c_rarg3为0则退出参数传递
308       __ movl(c_rarg3, parameter_size);
309       __ testl(c_rarg3, c_rarg3);
310       __ jcc(Assembler::zero, parameters_done);
311
312       Label loop;
313       __ movptr(c_rarg2, parameters);       // parameter pointer
314       __ movl(c_rarg1, c_rarg3);            // parameter counter is in c_rarg1 把参数大小复制到c_rarg1
315       __ BIND(loop); //循环把参数先传递到rax,然后push到堆栈
316       __ movptr(rax, Address(c_rarg2, 0));// get parameter
317       __ addptr(c_rarg2, wordSize);       // advance to next parameter
318       __ decrementl(c_rarg1);             // decrement counter
319       __ push(rax);                       // pass parameter
320       __ jcc(Assembler::notZero, loop);
321
322       // call Java function
323       __ BIND(parameters_done);
324       __ movptr(rbx, method);             // get Method* 保存method到rbx,entry_point里面会从rbx中取
325       __ movptr(c_rarg1, entry_point);    // get entry_point 保存entry_point的地址到c_rarg1供调用
326       __ mov(r13, rsp);                   // set sender sp 保存当前rsp到 r13,entry_point里面会用到,rsp即为最后一个参数的地址
327       BLOCK_COMMENT("call Java function");
328       __ call(c_rarg1);
329
330       BLOCK_COMMENT("call_stub_return_address:");
331       return_address = __ pc();//返回值地址,返回值的下一个地址即为新栈桢entry_point的开始
332
333       // store result depending on type (everything that is not
334       // T_OBJECT, T_LONG, T_FLOAT or T_DOUBLE is treated as T_INT)
335       __ movptr(c_rarg0, result);
336       Label is_long, is_float, is_double, exit;
337       __ movl(c_rarg1, result_type);
338       __ cmpl(c_rarg1, T_OBJECT);
339       __ jcc(Assembler::equal, is_long);
340       __ cmpl(c_rarg1, T_LONG);
341       __ jcc(Assembler::equal, is_long);
342       __ cmpl(c_rarg1, T_FLOAT);
343       __ jcc(Assembler::equal, is_float);
344       __ cmpl(c_rarg1, T_DOUBLE);
```

```
345       __ jcc(Assembler::equal, is_double);
346
347       // handle T_INT case
348       __ movl(Address(c_rarg0, 0), rax);
349
350       __ BIND(exit);
351
352       // pop parameters
353       __ lea(rsp, rsp_after_call);
354
355   #ifdef ASSERT
356       // verify that threads correspond
357       {
358         Label L, S;
359         __ cmpptr(r15_thread, thread);
360         __ jcc(Assembler::notEqual, S);
361         __ get_thread(rbx);
362         __ cmpptr(r15_thread, rbx);
363         __ jcc(Assembler::equal, L);
364         __ bind(S);
365         __ jcc(Assembler::equal, L);
366         __ stop("StubRoutines::call_stub: threads must correspond");
367         __ bind(L);
368       }
369   #endif
370
371       // restore regs belonging to calling function
372   #ifdef _WIN64
373       for (int i = 15; i >= 6; i--) {
374         __ movdqu(as_XMMRegister(i), xmm_save(i));
375       }
376   #endif
377       __ movptr(r15, r15_save);
378       __ movptr(r14, r14_save);
379       __ movptr(r13, r13_save);
380       __ movptr(r12, r12_save);
381       __ movptr(rbx, rbx_save);
382
383   #ifdef _WIN64
384       __ movptr(rdi, rdi_save);
385       __ movptr(rsi, rsi_save);
386   #else
387       __ ldmxcsr(mxcsr_save);
388   #endif
389
390       // restore rsp
391       __ addptr(rsp, -rsp_after_call_off * wordSize);
392
393       // return
394       __ pop(rbp);
395       __ ret(0);
396
397       // handle return types different from T_INT
398       __ BIND(is_long);
399       __ movq(Address(c_rarg0, 0), rax);
400       __ jmp(exit);
401
402       __ BIND(is_float);
403       __ movflt(Address(c_rarg0, 0), xmm0);
404       __ jmp(exit);
405
406       __ BIND(is_double);
407       __ movdbl(Address(c_rarg0, 0), xmm0);
408       __ jmp(exit);
409
410       return start;
411     }
412   下一篇文章说说entry_point的创建。
```