https://www.jianshu.com/p/8d28a91c9763
Stack & Frame In Hotspot (Base OpenJDK 8)

feihui

在开始今天的讲述之前,我们可以先来思考或者说回顾下何为栈(为什么是栈?其实你了解方法调用和栈的结构就会发现,两者在逻辑上有非常切合的操作,不过在这儿并不讨论为什么使用栈,更多详情可以自行搜索)?它的作用是什么?它的一般结构有是怎样的呢?

我们来看下 wiki 上一段关于 call stack 的描述:

In computer science, a call stack is a stack data structure that stores information about the active subroutines of a computer program. This kind of stack is also known as an execution stack, program stack, control stack, run-time stack, or machine stack, and is often shortened to just "the stack". Although maintenance of the call stack is important for the proper functioning of most software, the details are normally hidden and automatic in high-level programming languages. Many computer instruction sets provide special instructions for manipulating stacks.

A call stack is used for several related purposes, but the main reason for having one is to keep track of the point to which each active subroutine should return control when it finishes executing. An active subroutine is one that has been called but is yet to complete execution after which control should be handed back to the point of call. Such activations of subroutines may be nested to any level (recursive as a special case), hence the stack structure. If, for example, a subroutine DrawSquare calls a subroutine DrawLine from four different places, DrawLine must know where to return when its execution completes. To accomplish this, the address following the instruction that jumps to DrawLine, the return address, is pushed onto the call stack with each call.
[image]

从上面的描述以及示图我们可以看到 -- call stack 的最主要的功能是记录返回地址,还可以用来存储变量和传递参数,stack point 用于指向栈顶,而 frame point 则用于界定与当前方法相关的帧.

在 JVM 中又是怎么来表示 call stack 的呢?根据 JVM 规范,有两个种 stack -- Java Virtual Machine Stack 和 Native Method Stack,今天我就来探讨下 JVM 虚拟机栈在 Hotspot 的实现.JVM 对 Frame 的定义:

1. Local Variables
2. Operand Stacks
3. Dynamic Linking
4. Normal Method Invocation Completion
5. Abrupt Method Invocation Completion

那现在我们就来追寻下源码(我们都知道,C++ 程序有 main 方法作为程序入):

```
int main(int argc, char **argv) {
JLI_Launch(margc, margv,
                   sizeof(const_jargs) / sizeof(char *), const_jargs,
                   sizeof(const_appclasspath) / sizeof(char *), const_appclasspath,
                   FULL_VERSION, DOT_VERSION,
                   (const_progname != NULL) ? const_progname : *margv,
                   (const_launcher != NULL) ? const_launcher : *margv,
                   (const_jargs != NULL) ? JNI_TRUE : JNI_FALSE,
                   const_cpwildcard, const_javaw, const_ergo_class);
}

int ContinueInNewThread(InvocationFunctions* ifn, jlong threadStackSize,  int argc, char **argv, int mode, char *what, int ret) {
  ContinueInNewThread0(JavaMain, threadStackSize, (void*)&args);
}

int JNICALL JavaMain(void * _args) {
  InitializeJVM(&vm, &env, &ifn);
  LoadMainClass(env, mode, what);
  mainID = (*env)->GetStaticMethodID(env, mainClass, "main", "([Ljava/lang/String;)V");
  (*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);
}

static jboolean InitializeJVM(JavaVM **pvm, JNIEnv **penv, InvocationFunctions *ifn) {
  ifn->CreateJavaVM(pvm, (void **)penv, &args);
}

void (JNICALL *CallStaticVoidMethod) (JNIEnv *env, jclass cls, jmethodID methodID, ...);

JNI_ENTRY(void, jni_CallStaticVoidMethod(JNIEnv *env, jclass cls, jmethodID methodID, ...))
  ······ //
  jni_invoke_static(env, &jvalue, NULL, JNI_STATIC, methodID, &ap, CHECK);
  ······ //
JNI_END

static void jni_invoke_static(JNIEnv *env, JavaValue* result, jobject receiver, JNICallType call_type, jmethodID method_id, JNI_ArgumentPusher *args, TRAPS) {
  ······ //
  JavaCalls::call(result, method, &java_args, CHECK);
  ······ //
}

void JavaCalls::call(JavaValue* result, methodHandle method, JavaCallArguments* args, TRAPS) {
  ······ //
  os::os_exception_wrapper(call_helper, result, &method, args, THREAD);
}

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
      );
      ...... //
}
```

从 main 方式开始,我们可以最终追寻到 StubRoutines 的 call_stub,查看这个方法的定义:

```
  static CallStub call_stub() { return CAST_TO_FN_PTR(CallStub, _call_stub_entry); }
```

返回值是一个函数指针:

```

  // Calls to Java
  typedef void (*CallStub)(
    address   link,
    intptr_t* result,
    BasicType result_type,
    Method* method,
    address   entry_point,
    intptr_t* parameters,
    int       size_of_parameters,
    TRAPS
  );
```

宏定义里面的逻辑也仅仅是简单的类型转换:

```

#define CAST_TO_FN_PTR(func_type, value) ((func_type)(castable_address(value)))
```

那这个函数的执行逻辑是什么呢?使用 Netbeans Show Call Graph 查看 _call_stub_entry,可以很好追查到:

[image.png]

```

  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();
    __ subptr(rsp, -rsp_after_call_off * wordSize);

    // save register parameters
#ifndef _WIN64
    __ movptr(parameters,   c_rarg5); // parameters
    __ movptr(entry_point,  c_rarg4); // entry_point
#endif

    __ movptr(method,       c_rarg3); // method
    __ movl(result_type,  c_rarg2);   // result type
    __ movptr(result,       c_rarg1); // result
    __ movptr(call_wrapper, c_rarg0); // call wrapper

    // save regs belonging to calling function
    __ movptr(rbx_save, rbx);
    __ movptr(r12_save, r12);
    __ movptr(r13_save, r13);
    __ movptr(r14_save, r14);
    __ movptr(r15_save, r15);
#ifdef _WIN64
    for (int i = 6; i <= 15; i++) {
      __ movdqu(xmm_save(i), as_XMMRegister(i));
    }

```

```
    const Address rdi_save(rbp, rdi_off * wordSize);
    const Address rsi_save(rbp, rsi_off * wordSize);

    __ movptr(rsi_save, rsi);
    __ movptr(rdi_save, rdi);
#else
    const Address mxcsr_save(rbp, mxcsr_off * wordSize);
    {
      Label skip_ldmx;
      __ stmxcsr(mxcsr_save);
      __ movl(rax, mxcsr_save);
      __ andl(rax, MXCSR_MASK);    // Only check control and mask bits
      ExternalAddress mxcsr_std(StubRoutines::addr_mxcsr_std());
      __ cmp32(rax, mxcsr_std);
      __ jcc(Assembler::equal, skip_ldmx);
      __ ldmxcsr(mxcsr_std);
      __ bind(skip_ldmx);
    }
#endif

    // Load up thread register
    __ movptr(r15_thread, thread);
    __ reinit_heapbase();

#ifdef ASSERT
    // make sure we have no pending exceptions
    {
      Label L;
      __ cmpptr(Address(r15_thread, Thread::pending_exception_offset()), (int32_t)NULL_WORD);
      __ jcc(Assembler::equal, L);
      __ stop("StubRoutines::call_stub: entered with pending exception");
      __ bind(L);
    }
#endif

    // pass parameters if any
    BLOCK_COMMENT("pass parameters if any");
    Label parameters_done;
    __ movl(c_rarg3, parameter_size);
    __ testl(c_rarg3, c_rarg3);
    __ jcc(Assembler::zero, parameters_done);

    Label loop;
    __ movptr(c_rarg2, parameters);       // parameter pointer
    __ movl(c_rarg1, c_rarg3);            // parameter counter is in c_rarg1
    __ BIND(loop);
    __ movptr(rax, Address(c_rarg2, 0));// get parameter
    __ addptr(c_rarg2, wordSize);       // advance to next parameter
    __ decrementl(c_rarg1);             // decrement counter
    __ push(rax);                       // pass parameter
    __ jcc(Assembler::notZero, loop);

    // call Java function
    __ BIND(parameters_done);
    __ movptr(rbx, method);             // get Method*
    __ movptr(c_rarg1, entry_point);    // get entry_point
    __ mov(r13, rsp);                   // set sender sp
    BLOCK_COMMENT("call Java function");
    __ call(c_rarg1);

    BLOCK_COMMENT("call_stub_return_address:");
    return_address = __ pc();

    // store result depending on type (everything that is not
    // T_OBJECT, T_LONG, T_FLOAT or T_DOUBLE is treated as T_INT)
    __ movptr(c_rarg0, result);
    Label is_long, is_float, is_double, exit;
    __ movl(c_rarg1, result_type);
    __ cmpl(c_rarg1, T_OBJECT);
    __ jcc(Assembler::equal, is_long);
    __ cmpl(c_rarg1, T_LONG);
    __ jcc(Assembler::equal, is_long);
    __ cmpl(c_rarg1, T_FLOAT);
    __ jcc(Assembler::equal, is_float);
    __ cmpl(c_rarg1, T_DOUBLE);
    __ jcc(Assembler::equal, is_double);

    // handle T_INT case
    __ movl(Address(c_rarg0, 0), rax);

    __ BIND(exit);

    // pop parameters
    __ lea(rsp, rsp_after_call);

#ifdef ASSERT
```

```
    // verify that threads correspond
    {
      Label L, S;
      __ cmpptr(r15_thread, thread);
      __ jcc(Assembler::notEqual, S);
      __ get_thread(rbx);
      __ cmpptr(r15_thread, rbx);
      __ jcc(Assembler::equal, L);
      __ bind(S);
      __ jcc(Assembler::equal, L);
      __ stop("StubRoutines::call_stub: threads must correspond");
      __ bind(L);
    }
#endif

    // restore regs belonging to calling function
#ifdef _WIN64
    for (int i = 15; i >= 6; i--) {
      __ movdqu(as_XMMRegister(i), xmm_save(i));
    }
#endif
    __ movptr(r15, r15_save);
    __ movptr(r14, r14_save);
    __ movptr(r13, r13_save);
    __ movptr(r12, r12_save);
    __ movptr(rbx, rbx_save);

#ifdef _WIN64
    __ movptr(rdi, rdi_save);
    __ movptr(rsi, rsi_save);
#else
    __ ldmxcsr(mxcsr_save);
#endif

    // restore rsp
    __ addptr(rsp, -rsp_after_call_off * wordSize);

    // return
    __ pop(rbp);
    __ ret(0);

    // 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;
  }
```

跟之前字节码同样的套路,上面展示的是生成 call stub 汇编代码的代码,那么问题来了 -- call stub 作用是什么?我们思考这么一个问题,Hotspot 是由 C++ 编写而成,而根据 JVM 规范有自己的栈结构,那么其实这个 call stub 的目的就是 Call Java From C,我们来看下 call stub 需要处理的栈结构:

```
  // Linux Arguments:
  //    c_rarg0:   call wrapper address                   address
  //    c_rarg1:   result                                 address
  //    c_rarg2:   result type                            BasicType
  //    c_rarg3:   method                                 Method*
  //    c_rarg4:   (interpreter) entry point              address
  //    c_rarg5:   parameters                             intptr_t*
  //    16(rbp):   parameter size (in words)              int
  //    24(rbp):   thread                                 Thread*
  //
  //     [ return_from_Java     ] <--- rsp
  //     [ argument word n      ]
  //      ...
  // -12 [ argument word 1      ]
  // -11 [ saved r15            ] <--- rsp_after_call
  // -10 [ saved r14            ]
  //  -9 [ saved r13            ]
  //  -8 [ saved r12            ]
  //  -7 [ saved rbx            ]
  //  -6 [ call wrapper         ]
  //  -5 [ result               ]
  //  -4 [ result type          ]
  //  -3 [ method               ]
  //  -2 [ entry point          ]
  //  -1 [ parameters           ]
  //   0 [ saved rbp            ] <--- rbp
  //   1 [ return address       ]
```

```
  //   2 [ parameter size      ]
  //   3 [ thread              ]
  //
  // Windows Arguments:
  //    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     ] <--- rsp
  //     [ argument word n      ]
  //      ...
  // -28 [ argument word 1      ]
  // -27 [ saved xmm15          ] <--- rsp_after_call
  //     [ saved xmm7-xmm14     ]
  //  -9 [ saved xmm6           ] (each xmm register takes 2 slots)
  //  -7 [ saved r15            ]
  //  -6 [ saved r14            ]
  //  -5 [ saved r13            ]
  //  -4 [ saved r12            ]
  //  -3 [ saved rdi            ]
  //  -2 [ saved rsi            ]
  //  -1 [ saved rbx            ]
  //   0 [ saved rbp            ] <--- rbp
  //   1 [ return address       ]
  //   2 [ call wrapper         ]
  //   3 [ result               ]
  //   4 [ result type          ]
  //   5 [ method               ]
  //   6 [ entry point          ]
  //   7 [ parameters           ]
  //   8 [ parameter size       ]
  //   9 [ thread               ]
  //
  //    Windows reserves the callers stack space for arguments 1-4.
  //    We spill c_rarg0-c_rarg3 to this space.
```

我们再来关注下其中一个细节:

```
    // call Java function
    __ BIND(parameters_done);
    __ movptr(rbx, method);             // get Method*
    __ movptr(c_rarg1, entry_point);    // get entry_point
    __ mov(r13, rsp);                   // set sender sp
    BLOCK_COMMENT("call Java function");
    __ call(c_rarg1);
```

这其中的 entry point 又是什么呢?同样我们可以用 Show Call Graph 追查到:

```
address InterpreterGenerator::generate_normal_entry(bool synchronized) {
  // determine code generation flags
  bool inc_counter  = UseCompiler || CountCompiledCalls;

  // ebx: Method*
  // r13: sender sp
  address entry_point = __ pc();

  const Address constMethod(rbx, Method::const_offset());
  const Address access_flags(rbx, Method::access_flags_offset());
  const Address size_of_parameters(rdx,
                                   ConstMethod::size_of_parameters_offset());
  const Address size_of_locals(rdx, ConstMethod::size_of_locals_offset());


  // get parameter size (always needed)
  __ movptr(rdx, constMethod);
  __ load_unsigned_short(rcx, size_of_parameters);

  // rbx: Method*
  // rcx: size of parameters
  // r13: sender_sp (could differ from sp+wordSize if we were called via c2i )

  __ load_unsigned_short(rdx, size_of_locals); // get size of locals in words
  __ subl(rdx, rcx); // rdx = no. of additional locals

  // YYY
//   __ incrementl(rdx);
//   __ andl(rdx, -2);

  // see if we've got enough room on the stack for locals plus overhead.
  generate_stack_overflow_check();

  // get return address
  __ pop(rax);
```

```

    // compute beginning of parameters (r14)
    __ lea(r14, Address(rsp, rcx, Address::times_8, -wordSize));

    // rdx - # of additional locals
    // allocate space for locals
    // explicitly initialize locals
    {
      Label exit, loop;
      __ testl(rdx, rdx);
      __ jcc(Assembler::lessEqual, exit); // do nothing if rdx <= 0
      __ bind(loop);
      __ push((int) NULL_WORD); // initialize local variables
      __ decrementl(rdx); // until everything initialized
      __ jcc(Assembler::greater, loop);
      __ bind(exit);
    }

    // initialize fixed part of activation frame
    generate_fixed_frame(false);

    // make sure method is not native & not abstract
  #ifdef ASSERT
    __ movl(rax, access_flags);
    {
      Label L;
      __ testl(rax, JVM_ACC_NATIVE);
      __ jcc(Assembler::zero, L);
      __ stop("tried to execute native method as non-native");
      __ bind(L);
    }
    {
      Label L;
      __ testl(rax, JVM_ACC_ABSTRACT);
      __ jcc(Assembler::zero, L);
      __ stop("tried to execute abstract method in interpreter");
      __ bind(L);
    }
  #endif

    // Since at this point in the method invocation the exception
    // handler would try to exit the monitor of synchronized methods
    // which hasn't been entered yet, we set the thread local variable
    // _do_not_unlock_if_synchronized to true. The remove_activation
    // will check this flag.

    const Address do_not_unlock_if_synchronized(r15_thread,
          in_bytes(JavaThread::do_not_unlock_if_synchronized_offset()));
    __ movbool(do_not_unlock_if_synchronized, true);

    __ profile_parameters_type(rax, rcx, rdx);
    // increment invocation count & check for overflow
    Label invocation_counter_overflow;
    Label profile_method;
    Label profile_method_continue;
    if (inc_counter) {
      generate_counter_incr(&invocation_counter_overflow,
                            &profile_method,
                            &profile_method_continue);
      if (ProfileInterpreter) {
        __ bind(profile_method_continue);
      }
    }

    Label continue_after_compile;
    __ bind(continue_after_compile);

    // check for synchronized interpreted methods
    bang_stack_shadow_pages(false);

    // reset the _do_not_unlock_if_synchronized flag
    __ movbool(do_not_unlock_if_synchronized, false);

    // check for synchronized methods
    // Must happen AFTER invocation_counter check and stack overflow check,
    // so method is not locked if overflows.
    if (synchronized) {
      // Allocate monitor and lock method
      lock_method();
    } else {
      // no synchronization necessary
  #ifdef ASSERT
      {
        Label L;
        __ movl(rax, access_flags);
        __ testl(rax, JVM_ACC_SYNCHRONIZED);
```

```
      __ jcc(Assembler::zero, L);
      __ stop("method needs synchronization");
      __ bind(L);
    }
#endif
  }

  // start execution
#ifdef ASSERT
  {
    Label L;
     const Address monitor_block_top (rbp,
                 frame::interpreter_frame_monitor_block_top_offset * wordSize);
    __ movptr(rax, monitor_block_top);
    __ cmpptr(rax, rsp);
    __ jcc(Assembler::equal, L);
    __ stop("broken stack frame setup in interpreter");
    __ bind(L);
  }
#endif

  // jvmti support
  __ notify_method_entry();

  __ dispatch_next(vtos);

  // invocation counter overflow
  if (inc_counter) {
    if (ProfileInterpreter) {
      // We have decided to profile this method in the interpreter
      __ bind(profile_method);
      __ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::profile_method));
      __ set_method_data_pointer_for_bcp();
      __ get_method(rbx);
      __ jmp(profile_method_continue);
    }
    // Handle overflow of counter and compile method
    __ bind(invocation_counter_overflow);
    generate_counter_overflow(&continue_after_compile);
  }

  return entry_point;
}
```

同样也是生成 Entry Point 汇编代码的代码(这儿只展示普通 Interpreter 模式下的 Entry Point),我们来看下这其中涉及的内存布局:

```
// 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
```

我们可以看到不同于 Call Stub,Entry Point 主要是用于维护虚拟机栈帧结构.

在这儿我们关注一个细节 -- 栈帧重叠,在传统的 Call Stack 中,函数的入参会被保存两份 -- 一份在调用方中作为局部变量,另一份在被调用方中作为方法入参,中间被 return address 隔开(就如文章开头图示),这种方式无疑是增加了不必要的复制逻辑和内存消耗.而在 Hotspot 虚拟机栈中,调用方操作数栈中元素同时作为被调用方局部变量表元素(return address 被挪到另外地方).无论是哪种方式,目的都是为了保证方法外入参和方法内变量处于连续内存区域,不然访问数据的时候还得额外计算索引,相对与只执行一次 return 得不偿失.

我们再来关注另外一个细节 -- Entry Point 中的 dispatch_next(vtos),这是实现 JVM 语言程序连续执行的关键,其中 increment(r13, step) 指向下一条要执行的字节码,dispatch_base(state, Interpreter::dispatch_table(state)) 则是跳转都当前字节码对应的汇编代码,而每个字节码对应的汇编代码都有由 dispatch_epilog(tos_out, step) 生成的同样的计数和跳转指令.

```
void InterpreterMacroAssembler::dispatch_next(TosState state, int step) {
  // load next bytecode (load before advancing r13 to prevent AGI)
  load_unsigned_byte(rbx, Address(r13, step));
  // advance r13
  increment(r13, step);
  dispatch_base(state, Interpreter::dispatch_table(state));
}

void InterpreterMacroAssembler::dispatch_base(TosState state,
                                              address* table,
                                              bool verifyoop) {
  verify_FPU(1, state);
  if (VerifyActivationFrameSize) {
    Label L;
    mov(rcx, rbp);
    subptr(rcx, rsp);
    int32_t min_frame_size =
      (frame::link_offset - frame::interpreter_frame_initial_sp_offset) *
      wordSize;
    cmpptr(rcx, (int32_t)min_frame_size);
    jcc(Assembler::greaterEqual, L);
    stop("broken stack frame");
    bind(L);
  }
  if (verifyoop) {
    verify_oop(rax, state);
  }
  lea(rscratch1, ExternalAddress((address)table));
  jmp(Address(rscratch1, rbx, Address::times_8));
}

void TemplateInterpreterGenerator::generate_and_dispatch(Template* t, TosState tos_out) {
  ······ //
  // generate template
  t->generate(_masm);
  // advance
  if (t->does_dispatch()) {
#ifdef ASSERT
    // make sure execution doesn't go beyond this point if code is broken
    __ should_not_reach_here();
#endif // ASSERT
  } else {
    // dispatch to next bytecode
    __ dispatch_epilog(tos_out, step);
  }
}

void InterpreterMacroAssembler::dispatch_epilog(TosState state, int step) {
  dispatch_next(state, step);
}
```

更多的 Entry Point 如下:

```
void TemplateInterpreterGenerator::generate_all() {
  AbstractInterpreterGenerator::generate_all();

  { CodeletMark cm(_masm, "error exits");
    _unimplemented_bytecode    = generate_error_exit("unimplemented bytecode");
    _illegal_bytecode_sequence = generate_error_exit("illegal bytecode sequence - method not verified");
  }

#ifndef PRODUCT
  if (TraceBytecodes) {
    CodeletMark cm(_masm, "bytecode tracing support");
    Interpreter::_trace_code =
      EntryPoint(
        generate_trace_code(btos),
```

```
        generate_trace_code(ctos),
        generate_trace_code(stos),
        generate_trace_code(atos),
        generate_trace_code(itos),
        generate_trace_code(ltos),
        generate_trace_code(ftos),
        generate_trace_code(dtos),
        generate_trace_code(vtos)
      );
  }
#endif // !PRODUCT

  { CodeletMark cm(_masm, "return entry points");
    const int index_size = sizeof(u2);
    for (int i = 0; i < Interpreter::number_of_return_entries; i++) {
      Interpreter::_return_entry[i] =
        EntryPoint(
          generate_return_entry_for(itos, i, index_size),
          generate_return_entry_for(itos, i, index_size),
          generate_return_entry_for(itos, i, index_size),
          generate_return_entry_for(atos, i, index_size),
          generate_return_entry_for(itos, i, index_size),
          generate_return_entry_for(ltos, i, index_size),
          generate_return_entry_for(ftos, i, index_size),
          generate_return_entry_for(dtos, i, index_size),
          generate_return_entry_for(vtos, i, index_size)
        );
    }
  }

  { CodeletMark cm(_masm, "invoke return entry points");
    const TosState states[] = {itos, itos, itos, itos, ltos, ftos, dtos, atos, vtos};
    const int invoke_length = Bytecodes::length_for(Bytecodes::_invokestatic);
    const int invokeinterface_length = Bytecodes::length_for(Bytecodes::_invokeinterface);
    const int invokedynamic_length = Bytecodes::length_for(Bytecodes::_invokedynamic);

    for (int i = 0; i < Interpreter::number_of_return_addrs; i++) {
      TosState state = states[i];
      Interpreter::_invoke_return_entry[i] = generate_return_entry_for(state, invoke_length, sizeof(u2));
      Interpreter::_invokeinterface_return_entry[i] = generate_return_entry_for(state, invokeinterface_length, sizeof(u2));
      Interpreter::_invokedynamic_return_entry[i] = generate_return_entry_for(state, invokedynamic_length, sizeof(u4));
    }
  }

  { CodeletMark cm(_masm, "earlyret entry points");
    Interpreter::_earlyret_entry =
      EntryPoint(
        generate_earlyret_entry_for(btos),
        generate_earlyret_entry_for(ctos),
        generate_earlyret_entry_for(stos),
        generate_earlyret_entry_for(atos),
        generate_earlyret_entry_for(itos),
        generate_earlyret_entry_for(ltos),
        generate_earlyret_entry_for(ftos),
        generate_earlyret_entry_for(dtos),
        generate_earlyret_entry_for(vtos)
      );
  }

  { CodeletMark cm(_masm, "deoptimization entry points");
    for (int i = 0; i < Interpreter::number_of_deopt_entries; i++) {
      Interpreter::_deopt_entry[i] =
        EntryPoint(
          generate_deopt_entry_for(itos, i),
          generate_deopt_entry_for(itos, i),
          generate_deopt_entry_for(itos, i),
          generate_deopt_entry_for(atos, i),
          generate_deopt_entry_for(itos, i),
          generate_deopt_entry_for(ltos, i),
          generate_deopt_entry_for(ftos, i),
          generate_deopt_entry_for(dtos, i),
          generate_deopt_entry_for(vtos, i)
        );
    }
  }

  { CodeletMark cm(_masm, "result handlers for native calls");
    // The various result converter stublets.
    int is_generated[Interpreter::number_of_result_handlers];
    memset(is_generated, 0, sizeof(is_generated));

    for (int i = 0; i < Interpreter::number_of_result_handlers; i++) {
      BasicType type = types[i];
      if (!is_generated[Interpreter::BasicType_as_index(type)]++) {
        Interpreter::_native_abi_to_tosca[Interpreter::BasicType_as_index(type)] = generate_result_handler_for(type);
      }
```

```
      }
    }

    { CodeletMark cm(_masm, "continuation entry points");
      Interpreter::_continuation_entry =
        EntryPoint(
          generate_continuation_for(btos),
          generate_continuation_for(ctos),
          generate_continuation_for(stos),
          generate_continuation_for(atos),
          generate_continuation_for(itos),
          generate_continuation_for(ltos),
          generate_continuation_for(ftos),
          generate_continuation_for(dtos),
          generate_continuation_for(vtos)
        );
    }

    { CodeletMark cm(_masm, "safepoint entry points");
      Interpreter::_safept_entry =
        EntryPoint(
          generate_safept_entry_for(btos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
          generate_safept_entry_for(ctos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
          generate_safept_entry_for(stos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
          generate_safept_entry_for(atos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
          generate_safept_entry_for(itos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
          generate_safept_entry_for(ltos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
          generate_safept_entry_for(ftos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
          generate_safept_entry_for(dtos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
          generate_safept_entry_for(vtos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint))
        );
    }

    { CodeletMark cm(_masm, "exception handling");
      // (Note: this is not safepoint safe because thread may return to compiled code)
      generate_throw_exception();
    }

    { CodeletMark cm(_masm, "throw exception entrypoints");
      Interpreter::_throw_ArrayIndexOutOfBoundsException_entry = generate_ArrayIndexOutOfBounds_handler("java/lang/ArrayIndexOutOfBoundsException");
      Interpreter::_throw_ArrayStoreException_entry            = generate_klass_exception_handler("java/lang/ArrayStoreException" );
      Interpreter::_throw_ArithmeticException_entry            = generate_exception_handler("java/lang/ArithmeticException"           , "/ by zero");
      Interpreter::_throw_ClassCastException_entry             = generate_ClassCastException_handler();
      Interpreter::_throw_NullPointerException_entry           = generate_exception_handler("java/lang/NullPointerException"          , NULL );
      Interpreter::_throw_StackOverflowError_entry             = generate_StackOverflowError_handler();
    }

  #define method_entry(kind)                                                                          \
    { CodeletMark cm(_masm, "method entry point (kind = " #kind ")");                    \
      Interpreter::_entry_table[Interpreter::kind] = generate_method_entry(Interpreter::kind);  \
    }

    // all non-native method kinds
    method_entry(zerolocals)
    method_entry(zerolocals_synchronized)
    method_entry(empty)
    method_entry(accessor)
    method_entry(abstract)
    method_entry(java_lang_math_sin  )
    method_entry(java_lang_math_cos  )
    method_entry(java_lang_math_tan  )
    method_entry(java_lang_math_abs  )
    method_entry(java_lang_math_sqrt )
    method_entry(java_lang_math_log  )
    method_entry(java_lang_math_log10)
    method_entry(java_lang_math_exp  )
    method_entry(java_lang_math_pow  )
    method_entry(java_lang_ref_reference_get)

    if (UseCRC32Intrinsics) {
      method_entry(java_util_zip_CRC32_update)
      method_entry(java_util_zip_CRC32_updateBytes)
      method_entry(java_util_zip_CRC32_updateByteBuffer)
    }

    initialize_method_handle_entries();

    // all native method kinds (must be one contiguous block)
    Interpreter::_native_entry_begin = Interpreter::code()->code_end();
    method_entry(native)
    method_entry(native_synchronized)
    Interpreter::_native_entry_end = Interpreter::code()->code_end();
```

```

#undef method_entry

  // Bytecodes
  set_entry_points_for_all_bytes();
  set_safepoints_for_all_bytes();
}
```

我们再来看下 JVM 是如何执行方法的,对应的字节码为 invoke 系列字节码.

```

IRT_ENTRY(void, InterpreterRuntime::resolve_invoke(JavaThread* thread, Bytecodes::Code bytecode)) {
  // extract receiver from the outgoing argument list if necessary
  Handle receiver(thread, NULL);
  if (bytecode == Bytecodes::_invokevirtual || bytecode == Bytecodes::_invokeinterface) {
    ResourceMark rm(thread);
    methodHandle m (thread, method(thread));
    Bytecode_invoke call(m, bci(thread));
    Symbol* signature = call.signature();
    receiver = Handle(thread,
                  thread->last_frame().interpreter_callee_receiver(signature));
    assert(Universe::heap()->is_in_reserved_or_null(receiver()),
           "sanity check");
    assert(receiver.is_null() ||
           !Universe::heap()->is_in_reserved(receiver->klass()),
           "sanity check");
  }

  // resolve method
  CallInfo info;
  constantPoolHandle pool(thread, method(thread)->constants());

  {
    JvmtiHideSingleStepping jhss(thread);
    LinkResolver::resolve_invoke(info, receiver, pool,
                                 get_index_u2_cpcache(thread, bytecode), bytecode, CHECK);
    if (JvmtiExport::can_hotswap_or_post_breakpoint()) {
      int retry_count = 0;
      while (info.resolved_method()->is_old()) {
        // It is very unlikely that method is redefined more than 100 times
        // in the middle of resolve. If it is looping here more than 100 times
        // means then there could be a bug here.
        guarantee((retry_count++ < 100),
                  "Could not resolve to latest version of redefined method");
        // method is redefined in the middle of resolve so re-try.
        LinkResolver::resolve_invoke(info, receiver, pool,
                                     get_index_u2_cpcache(thread, bytecode), bytecode, CHECK);
      }
    }
  } // end JvmtiHideSingleStepping

  // check if link resolution caused cpCache to be updated
  if (already_resolved(thread)) return;

  if (bytecode == Bytecodes::_invokeinterface) {
    if (TraceItables && Verbose) {
      ResourceMark rm(thread);
      tty->print_cr("Resolving: klass: %s to method: %s", info.resolved_klass()->name()->as_C_string(),
      info.resolved_method()->name()->as_C_string());
    }
  }
#ifdef ASSERT
  if (bytecode == Bytecodes::_invokeinterface) {
    if (info.resolved_method()->method_holder() ==
                                            SystemDictionary::Object_klass()) {
      // NOTE: THIS IS A FIX FOR A CORNER CASE in the JVM spec
      // (see also CallInfo::set_interface for details)
      assert(info.call_kind() == CallInfo::vtable_call ||
             info.call_kind() == CallInfo::direct_call, "");
      methodHandle rm = info.resolved_method();
      assert(rm->is_final() || info.has_vtable_index(),
             "should have been set already");
    } else if (!info.resolved_method()->has_itable_index()) {
      // Resolved something like CharSequence.toString.  Use vtable not itable.
      assert(info.call_kind() != CallInfo::itable_call, "");
    } else {
      // Setup itable entry
      assert(info.call_kind() == CallInfo::itable_call, "");
      int index = info.resolved_method()->itable_index();
      assert(info.itable_index() == index, "");
    }
  } else {
    assert(info.call_kind() == CallInfo::direct_call ||
           info.call_kind() == CallInfo::vtable_call, "");
  }
#endif
  switch (info.call_kind()) {
  case CallInfo::direct_call:
```

```
      cache_entry(thread)->set_direct_call(
        bytecode,
        info.resolved_method());
      break;
    case CallInfo::vtable_call:
      cache_entry(thread)->set_vtable_call(
        bytecode,
        info.resolved_method(),
        info.vtable_index());
      break;
    case CallInfo::itable_call:
      cache_entry(thread)->set_itable_call(
        bytecode,
        info.resolved_method(),
        info.itable_index());
      break;
    default:  ShouldNotReachHere();
    }
  }
  IRT_END
```

我们从上面的可以清晰看到(又再一次验证),方法的调用都是借助 vtable 和 itable 的方式实现,当解析完对应的符号链接将其转换成常量池缓存,而常量池缓存则保存着对应的 vtable index(针对继承) 或者 interface address + table index(针对实现).另外,我们查看过源码应该知道,被 native 标注的方法,无法直接查看其实现 (实现为 C++ 实现),那如何查看这些方法的实现呢?这些方法在 C++ 实现中的方法名一般为包名 + 类名 + 方法名(Hotspot 也是通过这种规则来查找 native 方法的).OpenJDK的话,Java 里声明为 native 的方法多数在 jdk/src/<platform>/native 里可以找到.其中<platform>可以是 share,也就是平台中立的代码; 也可以是某个具体平台.这个 native 目录里的结构跟 Java 源码结构一样是按包名来组织的,不难找.不过需要提醒的是这些 native 方法不是 “JVM” 的,是 “类库” 的,不在 JVM 里面.要判断是不是 JVM 的代码很简单:OpenJDK里不在 hotspot 目录里的代码都不是 JVM 的代码.有些类的方法需要 JVM 的特殊支持的,可能会在实现里调用 JVM_ 开头的函数.这些函数在 hotspot/src/share/vm/prims/jvm.cpp 里实现.有少量 native 方法确实是纯由 JVM 来实现的,例如 sun.misc.Unsafe 里的那些.那些是特例.

最后我们来看一张图(HSDB 生成):
[pasted-image.png]

根据注释,结合上面的介绍,我们很明白地知道每个内存地址上存储的数值代表的是什么.