https://liuxiaofei.com.cn/blog/entry_point-jvm-java%e6%a0%88%e6%a1%a2%e7%9a%84%e5%88%9b%e5%bb%ba/
entry_point-JVM Java栈桢的创建

Content:
调用堆栈
entry_point的生成
固定桢生成
转发表与栈顶缓存
转发表入口设置

调用堆栈

```
InterpreterGenerator::generate_normal_entry() at templateInterpreter_x86_64.cpp:1,409 0x7ffff74a829d
AbstractInterpreterGenerator::generate_method_entry() at templateInterpreter_x86_64.cpp:1,660 0x7ffff74a8f81
TemplateInterpreterGenerator::generate_all() at templateInterpreter.cpp:369 0x7ffff749f303
InterpreterGenerator::InterpreterGenerator() at templateInterpreter_x86_64.cpp:2,051 0x7ffff74aa85f
TemplateInterpreter::initialize() at templateInterpreter.cpp:52 0x7ffff749dc67
interpreter_init() at interpreter.cpp:118 0x7ffff70df96e
init_globals() at init.cpp:107 0x7ffff7080d21
Threads::create_vm() at thread.cpp:3,424 0x7ffff74cc509
JNI_CreateJavaVM() at jni.cpp:5,166 0x7ffff7134f13
```

entry_point的生成

```
/hotspot/src/cpu/x86/vm/templateInterpreter_x86_64.cpp:1410
//
// Generic interpreted method entry to (asm) interpreter
//
address InterpreterGenerator::generate_normal_entry(bool synchronized) {
  // determine code generation flags
  bool inc_counter  = UseCompiler || CountCompiledCalls;

  // ebx: Method*
  // r13: sender sp (ebx 和 r13 的值在call_stub里面保存的)
  address entry_point = __ pc();//entry_point 函数的代码入口地址

  const Address constMethod(rbx, Method::const_offset());//得到constMethod的地址,rbx中是method的地址
  const Address access_flags(rbx, Method::access_flags_offset());
  const Address size_of_parameters(rdx,
                                   ConstMethod::size_of_parameters_offset());//得到parameter的大小和local变量的大小
  const Address size_of_locals(rdx, ConstMethod::size_of_locals_offset());

  //上面的地址只是构造函数,并没计算结果
  // get parameter size (always needed)
  __ movptr(rdx, constMethod);//计算constMethod的地址,并保存在rdx里面
  __ load_unsigned_short(rcx, size_of_parameters);//得到parameter大小,保存在rcx里面

  //rbx:保存基址;rcx:保存循环变量;rdx:保存目标地址;rax:保存返回地址(下面用到)
  // rbx: Method*
  // rcx: size of parameters
  // r13: sender_sp (could differ from sp+wordSize if we were called via c2i ) 即调用者的栈顶地址

  __ load_unsigned_short(rdx, size_of_locals); // get size of locals in words
  __ subl(rdx, rcx); // rdx = no. of additional locals 局部变量区保存传入的参数和被调用函数的局部变量

  // 所以参数在call_stub的栈桢里,被调用函数的局部变量在entry_point的栈桢里,即局部变量区在两个栈桢中重叠了

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

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

  //返回地址是在call_stub中保存的,如果不弹出堆栈到rax,那么局部变量区就如下面的样子:
  // [parameter 1]
  // [parameter 2]
  // ......
  // [parameter n]
  // [return address]
  // [local 1]
  // [local 2]
  // ......
  // [local n]
  // 显然中间有个return address很碍眼,不好计算地址,所以暂时把它挪出去。

  // get return address
  __ pop(rax);

  // compute beginning of parameters (r14)计算第一个参数的地址:当前栈顶地址 + 变量大小 * 8 - 一个字大小。
  // 这儿注意,因为地址保存在低地址上,而堆栈是向低地址扩展的,所以只需加n-1个变量大小就可以得到第一个参数的地址。
  __ lea(r14, Address(rsp, rcx, Address::times_8, -wordSize));

  // 把函数的局部变量全置0
  // 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);
    }

    // 生成固定桢,下面接着说
    // 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();

  // 调用函数的第一个字节码,当前栈顶缓存为vtos,即没有值。
  // 每一个字节码根据不同的栈顶缓存都会有不同的入口地址。
  // 什么是栈顶缓存呢? 就是栈顶的值在寄存器上面,是为了加速下一个指令的运行,比如省掉数据的传送。
  // 以istore字节码为例:
  // 如果栈顶缓存为vtos,则istore字节码会先pop被保存操作数到寄存器,然后调用mov被保存的操作数到堆栈。
  // 如果栈顶缓存为itos,则说明被保存的操作数已经在寄存器,则直接调用mov被保存的操作数到堆栈。
  // 下面接着说怎么执行字节码。
  __ 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;
}
```

固定桢生成

/hotspot/src/cpu/x86/vm/templateInterpreter_x86_64.cpp:565

```
// Generate a fixed interpreter frame. This is identical setup for
// interpreted methods and for native methods hence the shared code.
//
// Args:
//      rax: return address
//      rbx: Method*
//      r14: pointer to locals
//      r13: sender sp <---------
//      rdx: cp cache
void TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) {
  // initialize fixed part of activation frame
  __ push(rax);        // save return address 把返回地址紧接着局部变量区保存
  __ enter();          // save old & set new rbp 进入固定桢
  __ push(r13);        // set sender sp 保存调用者的地址,即call_stub调用entry_point的地址
  __ push((int)NULL_WORD); // leave last_sp as null
  __ movptr(r13, Address(rbx, Method::const_offset()));      // get ConstMethod*
  __ lea(r13, Address(r13, ConstMethod::codes_offset())); // get codebase 保存字节码的地址到r13
  __ push(rbx);        // save Method* 保存method的地址到堆栈上
  if (ProfileInterpreter) {
    Label method_data_continue;
    __ movptr(rdx, Address(rbx, in_bytes(Method::method_data_offset())));
    __ testptr(rdx, rdx);
    __ jcc(Assembler::zero, method_data_continue);
    __ addptr(rdx, in_bytes(MethodData::data_offset()));
    __ bind(method_data_continue);
    __ push(rdx);      // set the mdp (method data pointer)
  } else {
    __ push(0);
  }

  __ movptr(rdx, Address(rbx, Method::const_offset()));
  __ movptr(rdx, Address(rdx, ConstMethod::constants_offset()));
  __ movptr(rdx, Address(rdx, ConstantPool::cache_offset_in_bytes()));
  __ push(rdx); // set constant pool cache 保存常量池的地址到堆栈上
  __ push(r14); // set locals pointer 保存第一个参数的地址到堆栈上
  if (native_call) {
    __ push(0); // no bcp
```

```
  } else {
    __ push(r13); // set bcp   保存字节码池地址到堆栈上
  }
  __ push(0); // reserve word for pointer to expression stack bottom
  __ movptr(Address(rsp, 0), rsp); // set expression stack bottom //在rsp的地址保存rsp的值
}
```

转发表与栈顶缓存

从上面固定桢的生成代码中知道,第一次调用时,r13指向的是字节码池的首地址,即第一个字节码,而step为0。

```
/hotspot/src/cpu/x86/vm/interp_masm_x86_64.cpp:509
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));
  //在当前字节码的位置,指针向前移动step宽度,获取地址上的值,这个值即为字节码在转发表中的index,存储到 rbx.
  //step的值由字节码指令和操作数决定。
  //转发表中的 index 其实就是字节码(范围1~202),参考void DispatchTable::set_entry(int i, EntryPoint& entry)方法

  // advance r13
  increment(r13, step);//自增r13供下一次dispatch使用
  dispatch_base(state, Interpreter::dispatch_table(state));
  //Interpreter::dispatch_table(state) 返回当前栈顶状态的所有字节码入口点
}

//DispatchTable是一个二维数组的表,维度为栈顶状态和字节码,存储的是每个栈顶状态对应的字节码的入口点entry
/hotspot/src/share/vm/interpreter/templateInterpreter.hpp:158
static address*   dispatch_table(TosState state)              { return _active_table.table_for(state); }
/hotspot/src/share/vm/interpreter/templateInterpreter.cpp:195
DispatchTable TemplateInterpreter::_active_table;
/hotspot/src/share/vm/interpreter/templateInterpreter.hpp:62
class DispatchTable VALUE_OBJ_CLASS_SPEC {
 public:
  enum { length = 1 << BitsPerByte };                 // an entry point for each byte value (also for undefined bytecodes)

 private:
  address _table[number_of_states][length];           // dispatch tables, indexed by tosca and bytecode

 public:
  // Attributes
  EntryPoint entry(int i) const;                      // return entry point for a given bytecode i
  void       set_entry(int i, EntryPoint& entry);     // set    entry point for a given bytecode i
  address*   table_for(TosState state)          { return _table[state]; }
  address*   table_for()                        { return table_for((TosState)0); }
  int        distance_from(address *table)      { return table - table_for(); }
  int        distance_from(TosState state)      { return distance_from(table_for(state)); }

  // Comparison
  bool operator == (DispatchTable& y);                // for debugging only
};
//下面的方法显示了对每个字节码的每个栈顶状态都设置入口地址,在字节码编译完后调用。下面继续说。
/hotspot/src/share/vm/interpreter/templateInterpreter.cpp:145
void DispatchTable::set_entry(int i, EntryPoint& entry) {
  assert(0 <= i && i < length, "index out of bounds");
  assert(number_of_states == 9, "check the code below");
  _table[btos][i] = entry.entry(btos);
  _table[ctos][i] = entry.entry(ctos);
  _table[stos][i] = entry.entry(stos);
  _table[atos][i] = entry.entry(atos);
  _table[itos][i] = entry.entry(itos);
  _table[ltos][i] = entry.entry(ltos);
  _table[ftos][i] = entry.entry(ftos);
  _table[dtos][i] = entry.entry(dtos);
  _table[vtos][i] = entry.entry(vtos);
}

/hotspot/src/cpu/x86/vm/interp_masm_x86_64.cpp:473
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));//获取当前栈顶状态字节码转发表的地址,保存到rscratch1
```

```
  jmp(Address(rscratch1, rbx, Address::times_8)); //跳转到字节码对应的入口执行机器码指令。address = rscratch1 + rbx * 8
}
```

转发表入口设置

```
//JVM启动的时候会调用此方法,生成所有的entry_point,编译所有的字节码,并设置每个字节码在不同栈顶缓存状态下的入口
void TemplateInterpreterGenerator::generate_all() {
  AbstractInterpreterGenerator::generate_all();
......
#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)//普通的JAVA方法调用的entry_point在这儿生成
......

#undef method_entry

  // Bytecodes
  set_entry_points_for_all_bytes();//为每个字节码编译并设置在不同栈顶缓存状态下的入口
  set_safepoints_for_all_bytes();
}

void TemplateInterpreterGenerator::set_entry_points_for_all_bytes() {
  for (int i = 0; i < DispatchTable::length; i++) {
    Bytecodes::Code code = (Bytecodes::Code)i;
    if (Bytecodes::is_defined(code)) {
      set_entry_points(code);
    } else {
      set_unimplemented(i);
    }
  }
}

void TemplateInterpreterGenerator::set_entry_points(Bytecodes::Code code) {
  CodeletMark cm(_masm, Bytecodes::name(code), code);
  // initialize entry points
  assert(_unimplemented_bytecode    != NULL, "should have been generated before");
  assert(_illegal_bytecode_sequence != NULL, "should have been generated before");
  address bep = _illegal_bytecode_sequence;
  address cep = _illegal_bytecode_sequence;
  address sep = _illegal_bytecode_sequence;
  address aep = _illegal_bytecode_sequence;
  address iep = _illegal_bytecode_sequence;
  address lep = _illegal_bytecode_sequence;
  address fep = _illegal_bytecode_sequence;
  address dep = _illegal_bytecode_sequence;
  address vep = _unimplemented_bytecode;
  address wep = _unimplemented_bytecode;
  // code for short & wide version of bytecode
  if (Bytecodes::is_defined(code)) {
    Template* t = TemplateTable::template_for(code);
    assert(t->is_valid(), "just checking");
    set_short_entry_points(t, bep, cep, sep, aep, iep, lep, fep, dep, vep);
  }
  if (Bytecodes::wide_is_defined(code)) {
    Template* t = TemplateTable::template_for_wide(code);
    assert(t->is_valid(), "just checking");
    set_wide_entry_point(t, wep);
  }
  // set entry points
  EntryPoint entry(bep, cep, sep, aep, iep, lep, fep, dep, vep);
  Interpreter::_normal_table.set_entry(code, entry);//上面已经说了,给当前字节码code设置不同栈顶缓存(bep,cep,sep,aep,iep,lep,fep,dep,vep)下的入
  Interpreter::_wentry_point[code] = wep;
}

void TemplateInterpreterGenerator::set_short_entry_points(Template* t, address& bep, address& cep, address& sep,
                             address& aep, address& iep, address& lep, address& fep, address& dep, address& vep)
{
  assert(t->is_valid(), "template must exist");
  switch (t->tos_in()) {
    case btos:
    case ctos:
    case stos:
      ShouldNotReachHere();  // btos/ctos/stos should use itos.
      break;
    case atos: vep = __ pc(); __ pop(atos); aep = __ pc(); generate_and_dispatch(t); break;
    case itos: vep = __ pc(); __ pop(itos); iep = __ pc(); generate_and_dispatch(t); break;//以istore为例,此字节码的vep和iep地址在这儿获取
    case ltos: vep = __ pc(); __ pop(ltos); lep = __ pc(); generate_and_dispatch(t); break;
    case ftos: vep = __ pc(); __ pop(ftos); fep = __ pc(); generate_and_dispatch(t); break;
    case dtos: vep = __ pc(); __ pop(dtos); dep = __ pc(); generate_and_dispatch(t); break;
    case vtos: set_vtos_entry_points(t, bep, cep, sep, aep, iep, lep, fep, dep, vep);     break;
    default  : ShouldNotReachHere();                                                 break;
  }
```

```
}

void TemplateInterpreterGenerator::generate_and_dispatch(Template* t, TosState tos_out) {
  if (PrintBytecodeHistogram)                                    histogram_bytecode(t);
#ifndef PRODUCT
  // debugging code
  if (CountBytecodes || TraceBytecodes || StopInterpreterAt > 0) count_bytecode();
  if (PrintBytecodePairHistogram)                                histogram_bytecode_pair(t);
  if (TraceBytecodes)                                            trace_bytecode(t);
  if (StopInterpreterAt > 0)                                     stop_interpreter_at();
  __ verify_FPU(1, t->tos_in());
#endif // !PRODUCT
  int step;
  if (!t->does_dispatch()) {
    step = t->is_wide() ? Bytecodes::wide_length_for(t->bytecode()) : Bytecodes::length_for(t->bytecode());
    if (tos_out == ilgl) tos_out = t->tos_out();
    // compute bytecode size
    assert(step > 0, "just checkin'");
    // setup stuff for dispatching next bytecode
    if (ProfileInterpreter && VerifyDataPointer
        && MethodData::bytecode_has_profile(t->bytecode())) {
      __ verify_method_data_pointer();
    }
    __ dispatch_prolog(tos_out, step);
  }
  // generate template
  t->generate(_masm);//生成当前字节码的汇编模板
  // 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);//把指针指向下一个字节码,并跳转到当前字节码的代码位置执行机器码
  }
}
```