[inside hotspot] 汇编模板解释器(Template Interpreter)和字节码执行

1.模板解释器

hotspot解释器模块(`hotspot\src\share\vm\interpreter`)有两个实现：基于C++的解释器和基于汇编的模板解释器。hotspot默认使用比较快的模板解释器。
其中

- C++解释器 = `bytecodeInterpreter*` + `cppInterpreter*`
- 模板解释器 = `templateTable*` + `templateInterpreter*`

它们前者负责字节码的解释，后者负责解释器的运行时，共同完成解释功能。这里我们只关注模板解释器。

模板解释器又分为三个组成部分：

- `templateInterpreterGenerator` 解释器生成器
- `templateTable` 字节码实现
- `templateInterpreter` 解释器
  可能看起来很奇怪，为什么有一个解释器生成器和字节码实现。进入解释器实现：

```
class TemplateInterpreter: public AbstractInterpreter {
  friend class VMStructs;
  friend class InterpreterMacroAssembler;
  friend class TemplateInterpreterGenerator;
  friend class TemplateTable;
  friend class CodeCacheExtensions;
  // friend class Interpreter;
 public:

  enum MoreConstants {
    number_of_return_entries  = number_of_states,               // number of return entry points
    number_of_deopt_entries   = number_of_states,               // number of deoptimization entry points
    number_of_return_addrs    = number_of_states                // number of return addresses
  };

 protected:

  static address    _throw_ArrayIndexOutOfBoundsException_entry;
  static address    _throw_ArrayStoreException_entry;
  static address    _throw_ArithmeticException_entry;
  static address    _throw_ClassCastException_entry;
  static address    _throw_NullPointerException_entry;
  static address    _throw_exception_entry;

  static address    _throw_StackOverflowError_entry;

  static address    _remove_activation_entry;                   // continuation address if an exception is not handled by cur
#ifdef HOTSWAP
  static address    _remove_activation_preserving_args_entry;   // continuation address when current frame is being popped
#endif // HOTSWAP

#ifndef PRODUCT
  static EntryPoint _trace_code;
#endif // !PRODUCT
  static EntryPoint _return_entry[number_of_return_entries];    // entry points to return to from a call
  static EntryPoint _earlyret_entry;                            // entry point to return early from a call
  static EntryPoint _deopt_entry[number_of_deopt_entries];      // entry points to return to from a deoptimization
  static EntryPoint _continuation_entry;
  static EntryPoint _safept_entry;

  static address _invoke_return_entry[number_of_return_addrs];           // for invokestatic, invokespecial, invokevirtual re
  static address _invokeinterface_return_entry[number_of_return_addrs];  // for invokeinterface return entries
  static address _invokedynamic_return_entry[number_of_return_addrs];    // for invokedynamic return entries
```

```
  static DispatchTable _active_table;                           // the active    dispatch table (used by the interpreter for
  static DispatchTable _normal_table;                           // the normal    dispatch table (used to set the active table
  static DispatchTable _safept_table;                           // the safepoint dispatch table (used to set the active table
  static address       _wentry_point[DispatchTable::length];    // wide instructions only (vtos tosca always)


 public:
  ...
  static int InterpreterCodeSize;
};
```

里面很多 `address` 变量，`EntryPoint` 是一个address数组， `DispatchTable` 也是。
模板解释器就是由一系列例程(routine)组成的，即 `address` 变量，它们每个都表示一个例程的入口地址，比如异常处理例程，invoke指令例程，用于gc的safepoint例程...
举个形象的例子，我们都知道字节码文件长这样：

```
public void f();                                                                                        0: aload_0
1: invokespecial #5                  // Method A.f:()V
4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
7: ldc           #6                  // String ff
9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
12: return
```

如果要让我们写解释器，可能基本上就是一个循环里面switch，根据不同opcode派发到不同例程，例程的代码都是一样的模板代码，对aload_0的处理永远是取局部变量槽0的数据放到栈顶，那么完全可以在switch派发字节码前准备好这些模板代码，`templateInterpreterGenerator` 就是做的这件事，它的 `generate_all()` 函数初始化了所有的例程：

```
void TemplateInterpreterGenerator::generate_all() {
  // 设置slow_signature_handler例程
  { CodeletMark cm(_masm, "slow signature handler");
    AbstractInterpreter::_slow_signature_handler = generate_slow_signature_handler();
  }
  // 设置error_exit例程
  { CodeletMark cm(_masm, "error exits");
    _unimplemented_bytecode    = generate_error_exit("unimplemented bytecode");
    _illegal_bytecode_sequence = generate_error_exit("illegal bytecode sequence - method not verified");
  }
  ......
}
```

另外，既然已经涉及到机器码了，单独的 `templateInterpreterGenerator` 显然是不能完成这件事的，它还需要配合

`hotspot\src\cpu\x86\vm\templateInterpreterGenerator_x86.cpp` && `hotspot\src\cpu\x86\vm\templateInterpreterGenerator_x86_64.cpp` 一起做事(我的机器是x86+windows)。

使用 `-XX:+UnlockDiagnosticVMOptions -XX:+PrintInterpreter -XX:+LogCompilation -XX:LogFile=file.log` 保存结果到文件，可以查看生成的这些例程。
随便举个例子，模板解释器特殊处理java.lang.Math里的很多数学函数，使用它们不需要建立通常意义的java栈帧，且使用sse指令可以得到极大的性能提升：

```
// hotspot\src\cpu\x86\vm\templateInterpreterGenerator_x86_64.cpp
address TemplateInterpreterGenerator::generate_math_entry(AbstractInterpreter::MethodKind kind) {
  // rbx,: Method*
  // rcx: scratrch
  // r13: sender sp
  if (!InlineIntrinsics) return NULL; // Generate a vanilla entry
  address entry_point = __ pc();
```

```
if (kind == Interpreter::java_lang_math_fmaD) {
  if (!UseFMA) {
    return NULL; // Generate a vanilla entry
  }
  __ movdbl(xmm0, Address(rsp, wordSize));
  __ movdbl(xmm1, Address(rsp, 3 * wordSize));
  __ movdbl(xmm2, Address(rsp, 5 * wordSize));

  __ fmad(xmm0, xmm1, xmm2, xmm0);
} else if (kind == Interpreter::java_lang_math_fmaF) {
  if (!UseFMA) {
    return NULL; // Generate a vanilla entry
  }
  __ movflt(xmm0, Address(rsp, wordSize));
  __ movflt(xmm1, Address(rsp, 2 * wordSize));
  __ movflt(xmm2, Address(rsp, 3 * wordSize));
  __ fmaf(xmm0, xmm1, xmm2, xmm0);
} else if (kind == Interpreter::java_lang_math_sqrt) {
  __ sqrtsd(xmm0, Address(rsp, wordSize));
} else if (kind == Interpreter::java_lang_math_exp) {
  __ movdbl(xmm0, Address(rsp, wordSize));
  if (StubRoutines::dexp() != NULL) {
    __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, StubRoutines::dexp())));
  } else {
    __ call_VM_leaf0(CAST_FROM_FN_PTR(address, SharedRuntime::dexp));
  }
} else if (kind == Interpreter::java_lang_math_log) {
  __ movdbl(xmm0, Address(rsp, wordSize));
  if (StubRoutines::dlog() != NULL) {
    __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, StubRoutines::dlog())));
  } else {
    __ call_VM_leaf0(CAST_FROM_FN_PTR(address, SharedRuntime::dlog));
  }
} else if (kind == Interpreter::java_lang_math_log10) {
  __ movdbl(xmm0, Address(rsp, wordSize));
  if (StubRoutines::dlog10() != NULL) {
    __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, StubRoutines::dlog10())));
  } else {
    __ call_VM_leaf0(CAST_FROM_FN_PTR(address, SharedRuntime::dlog10));
  }
} else if (kind == Interpreter::java_lang_math_sin) {
  __ movdbl(xmm0, Address(rsp, wordSize));
  if (StubRoutines::dsin() != NULL) {
    __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, StubRoutines::dsin())));
  } else {
    __ call_VM_leaf0(CAST_FROM_FN_PTR(address, SharedRuntime::dsin));
  }
} else if (kind == Interpreter::java_lang_math_cos) {
  __ movdbl(xmm0, Address(rsp, wordSize));
  if (StubRoutines::dcos() != NULL) {
    __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, StubRoutines::dcos())));
  } else {
    __ call_VM_leaf0(CAST_FROM_FN_PTR(address, SharedRuntime::dcos));
  }
} else if (kind == Interpreter::java_lang_math_pow) {
  __ movdbl(xmm1, Address(rsp, wordSize));
  __ movdbl(xmm0, Address(rsp, 3 * wordSize));
  if (StubRoutines::dpow() != NULL) {
    __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, StubRoutines::dpow())));
  } else {
    __ call_VM_leaf0(CAST_FROM_FN_PTR(address, SharedRuntime::dpow));
  }
} else if (kind == Interpreter::java_lang_math_tan) {
  __ movdbl(xmm0, Address(rsp, wordSize));
  if (StubRoutines::dtan() != NULL) {
    __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, StubRoutines::dtan())));
```

```
      __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, StubRoutines::dtan())));
    } else {
      __ call_VM_leaf0(CAST_FROM_FN_PTR(address, SharedRuntime::dtan));
    }
  } else {
    __ fld_d(Address(rsp, wordSize));
    switch (kind) {
    case Interpreter::java_lang_math_abs:

      __ fabs();
      break;
    default:
      ShouldNotReachHere();
    }

    __ subptr(rsp, 2*wordSize);
    // Round to 64bit precision
    __ fstp_d(Address(rsp, 0));
    __ movdbl(xmm0, Address(rsp, 0));
    __ addptr(rsp, 2*wordSize);
  }

  __ pop(rax);
  __ mov(rsp, r13);
  __ jmp(rax);

  return entry_point;
}
```

我们关注 `java.lang.math.Pow()` 方法，加上 `-XX:+PrintInterpreter` 查看生成的例程：

```
else if (kind == Interpreter::java_lang_math_pow) {
    __ movdbl(xmm1, Address(rsp, wordSize));
    __ movdbl(xmm0, Address(rsp, 3 * wordSize));
    if (StubRoutines::dpow() != NULL) {
      __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, StubRoutines::dpow())));
    } else {
      __ call_VM_leaf0(CAST_FROM_FN_PTR(address, SharedRuntime::dpow));
    }
  }



----------------------------------------------------------------------
method entry point (kind = java_lang_math_pow)  [0x000001bcb62feaa0, 0x000001bcb62feac0]  32 bytes

  0x000001bcb62feaa0: vmovsd 0x8(%rsp),%xmm1
  0x000001bcb62feaa6: vmovsd 0x18(%rsp),%xmm0
  0x000001bcb62feaac: callq  0x000001bcb62f19d0
  0x000001bcb62feab1: pop    %rax
  0x000001bcb62feab2: mov    %r13,%rsp
  0x000001bcb62feab5: jmpq   *%rax
  0x000001bcb62feab7: nop
  0x000001bcb62feab8: add    %al,(%rax)
  0x000001bcb62feaba: add    %al,(%rax)
  0x000001bcb62feabc: add    %al,(%rax)
  0x000001bcb62feabe: add    %al,(%rax)
```

`callq` 会调用 `hotspot\src\cpu\x86\vm\stubGenerator_x86_64.cpp` 的 `address generate_libmPow()` ，感兴趣的可以去看一下，这里就不展开了。

2.字节码的解释执行

现在我们知道了模板解释器其实是由一堆例程构成的，但是，字节码的例程的呢？看看上面 `TemplateInterpreter` 的类定义，有个 `static DispatchTable _active_table;` ,它就是我们要找的东西了。具体来说 `templateInterpreterGenerator` 会调用 `TemplateInterpreterGenerator::set_entry_points()` 为每个字节码设置例程，该例程通过 `templateTable::template_for()` 获得。同样，这些代码需要关心cpu架构，所以自己每个字节码的例程也是由 `hotspot\src\cpu\x86\vm\templateTable_x86.cpp + templateTable` 共同完成的。

字节码太多了，这里也随便举个例子，考虑istore，它负责将栈顶数据出栈并存放到当前方法的局部变量表，实现如下：

```
void TemplateTable::istore() {
  transition(itos, vtos);
  locals_index(rbx);
  __ movl(iaddress(rbx), rax);
}
```

合情合理的实现

等等，当使用 `-XX:+PrintInterpreter` 查看istore的合情合理的例程时却得到了一大堆汇编：

```
----------------------------------------------------------------------
istore  54 istore  [0x00000192d1972ba0, 0x00000192d1972c00]  96 bytes

  0x00000192d1972ba0: mov    (%rsp),%eax
  0x00000192d1972ba3: add    $0x8,%rsp
  0x00000192d1972ba7: movzbl 0x1(%r13),%ebx
  0x00000192d1972bac: neg    %rbx
  0x00000192d1972baf: mov    %eax,(%r14,%rbx,8)
  0x00000192d1972bb3: movzbl 0x2(%r13),%ebx
  0x00000192d1972bb8: add    $0x2,%r13
  0x00000192d1972bbc: movabs $0x7fffd56e0fa0,%r10
  0x00000192d1972bc6: jmpq   *(%r10,%rbx,8)
  0x00000192d1972bca: mov    (%rsp),%eax
  0x00000192d1972bcd: add    $0x8,%rsp
  0x00000192d1972bd1: movzwl 0x2(%r13),%ebx
  0x00000192d1972bd6: bswap  %ebx
  0x00000192d1972bd8: shr    $0x10,%ebx
  0x00000192d1972bdb: neg    %rbx
  0x00000192d1972bde: mov    %eax,(%r14,%rbx,8)
  0x00000192d1972be2: movzbl 0x4(%r13),%ebx
  0x00000192d1972be7: add    $0x4,%r13
  0x00000192d1972beb: movabs $0x7fffd56e0fa0,%r10
  0x00000192d1972bf5: jmpq   *(%r10,%rbx,8)
  0x00000192d1972bf9: nopl   0x0(%rax)
```

虽然勉强能看出 `mov    %eax,(%r14,%rbx,8)` 对应 `__ movl(iaddress(n), rax);` ,但是多出来的代码怎么回事。
要回答这个问题，需要点其他知识。

之前提到

> `templateInterpreterGenerator` 会调用 `TemplateInterpreterGenerator::set_entry_points()` 为每个字节码设置例程

可以从 `set_entry_points` 出发看看它为istore做了什么特殊的事情：

```
...
  // 指令是否存在
  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);
  }
  // 指令是否可以扩宽，即wide
  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);
    }
  ...
  }
```

中间有一句话：

```
Template* t = TemplateTable::template_for(code);
```

从模板表中的查找 `Bytecodes::Code` 常量得到的是一个 `Template` ， `Template` 描述了一个指定的字节码对应的代码的一些属性

> // A Template describes the properties of a code template for a given bytecode
> // and provides a generator to generate the code template.

```
// hotspot\src\share\vm\utilities\globalDefinitions.hpp
// TosState用来描述一个字节码或者方法执行前后的状态。
enum TosState {         // describes the tos cache contents
  btos = 0,             // byte, bool tos cached
  ztos = 1,             // byte, bool tos cached
  ctos = 2,             // char tos cached
  stos = 3,             // short tos cached
  itos = 4,             // int tos cached
  ltos = 5,             // long tos cached
  ftos = 6,             // float tos cached
  dtos = 7,             // double tos cached
  atos = 8,             // object cached
  vtos = 9,             // tos not cached
  number_of_states,
  ilgl                  // illegal state: should not occur
};



// hotspot\src\share\vm\interpreter\templateTable.hpp
class Template VALUE_OBJ_CLASS_SPEC {
 private:
  enum Flags {
    uses_bcp_bit,                                // 是否需要字节码指针(bcp)？
    does_dispatch_bit,                           // 是否需要dispatch？
    calls_vm_bit,                                // 是否调用了虚拟机方法？
    wide_bit                                     // 能否扩宽，即加wide
  };

  typedef void (*generator)(int arg);           // 字节码代码生成器，其实是一个函数指针

  int       _flags;                              // 就是↑描述的flag
  TosState  _tos_in;                             // 执行字节码前的栈顶缓存状态
  TosState  _tos_out;                            // 执行字节码的栈顶缓存状态
  generator _gen;                                // 字节码代码生成器
  int       _arg;                                // 字节码代码生成器参数
```

然后找到istore对应的模板定义：

```
  //hotspot\src\share\vm\interpreter\templateTable.cpp
void TemplateTable::initialize() {
  ...
  //                                    interpr. templates
  // Java spec bytecodes                ubcp|disp|clvm|iswd  in    out   generator             argument
```

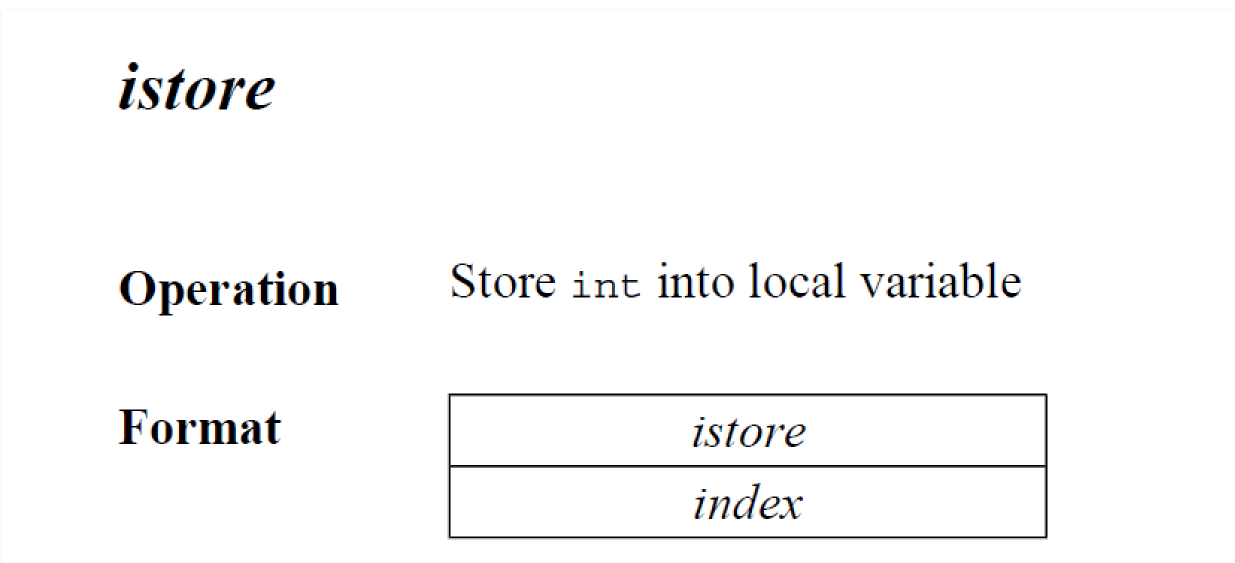
```
  // Java spec bytecodes               ubcp|disp|clvm|iswd  in    out   generator             argument
  def(Bytecodes::_istore              , ubcp|____|clvm|____, itos, vtos, istore              , _          );
  def(Bytecodes::_lstore              , ubcp|____|____|____, ltos, vtos, lstore              , _          );
  def(Bytecodes::_fstore              , ubcp|____|____|____, ftos, vtos, fstore              , _          );
  def(Bytecodes::_dstore              , ubcp|____|____|____, dtos, vtos, dstore              , _          );
  def(Bytecodes::_astore              , ubcp|____|clvm|____, vtos, vtos, astore              , _          );
 ...
  // wide Java spec bytecodes
  def(Bytecodes::_istore              , ubcp|____|____|iswd, vtos, vtos, wide_istore         , _          );
  def(Bytecodes::_lstore              , ubcp|____|____|iswd, vtos, vtos, wide_lstore         , _          );
  def(Bytecodes::_fstore              , ubcp|____|____|iswd, vtos, vtos, wide_fstore         , _          );
  def(Bytecodes::_dstore              , ubcp|____|____|iswd, vtos, vtos, wide_dstore         , _          );
  def(Bytecodes::_astore              , ubcp|____|____|iswd, vtos, vtos, wide_astore         , _          );
  def(Bytecodes::_iinc                , ubcp|____|____|iswd, vtos, vtos, wide_iinc           , _          );
  def(Bytecodes::_ret                 , ubcp|disp|____|iswd, vtos, vtos, wide_ret            , _          );
  def(Bytecodes::_breakpoint          , ubcp|disp|clvm|____, vtos, vtos, _breakpoint         , _          );

  ...
}
```

这里定义的意思就是， `istore` 使用无参数的生成器istore函数生成例程，这个生成器正是之前提到的那个很短的汇编代码：

```
void TemplateTable::istore() {
  transition(itos, vtos);
  locals_index(rbx);
  __ movl(iaddress(rbx), rax);
}
```

`ubcp` 表示使用字节码指针，所谓字节码指针指的是该字节码的操作数是否存在于字节码里面,一图胜千言：



istore的index紧跟在istore(0x36)后面，所以istore需要移动字节码指针以获取index。

`istore` 还规定执行前栈顶缓存int值(itos)，执行后不缓存(vtos)，且istore还有一个wide版本，这个版本使用两个字节的index。

有了这些信息，可以试着解释多出的汇编是怎么回事了。 `set_entry_points()` 为istore和wide版本的istore生成代码，
我们选择普通版本的istore解释，wide版本的依样画葫芦即可。它又进一步调用了 `set_short_entry_points()` ：

```
void TemplateInterpreterGenerator::set_entry_points(Bytecodes::Code code) {
 ...
  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);
    }
  ...
  }

  void TemplateInterpreterGenerator::set_short_entry_points(Template* t, address& bep, address& cep, address& sep, address& aep
    assert(t->is_valid(), "template must exist");
    switch (t->tos_in()) {
      case btos:
      case ztos:
      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;
      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;
    }
  }
```

`set_short_entry_points` 会根据该指令执行前是否需要栈顶缓存pop数据，istore使用了itos缓存，所以需要pop：

```
  // hotspot\src\cpu\x86\vm\interp_masm_x86.cpps
  void InterpreterMacroAssembler::pop_i(Register r) {
    // XXX can't use pop currently, upper half non clean
    movl(r, Address(rsp, 0));
    addptr(rsp, wordSize);
  }
```

稍微需要注意的是这里说的pop是一个弹出的概念，实际生成的代码是mov，试着解释那一大堆汇编：
mov指令

```
  ----------------------------------------------------------------------
  istore  54 istore  [0x00000192d1972ba0, 0x00000192d1972c00]  96 bytes
    ;获取栈顶int缓存
    0x00000192d1972ba0: mov    (%rsp),%eax
    0x00000192d1972ba3: add    $0x8,%rsp

    0x00000192d1972ba7: movzbl 0x1(%r13),%ebx
    0x00000192d1972bac: neg    %rbx
    0x00000192d1972baf: mov    %eax,(%r14,%rbx,8)
    0x00000192d1972bb3: movzbl 0x2(%r13),%ebx
    0x00000192d1972bb8: add    $0x2,%r13
    0x00000192d1972bbc: movabs $0x7fffd56e0fa0,%r10
    0x00000192d1972bc6: jmpq   *(%r10,%rbx,8)
    0x00000192d1972bca: mov    (%rsp),%eax
    0x00000192d1972bcd: add    $0x8,%rsp
    0x00000192d1972bd1: movzwl 0x2(%r13),%ebx
    0x00000192d1972bd6: bswap  %ebx
    0x00000192d1972bd8: shr    $0x10,%ebx
```

```
  0x00000192d1972bdb: neg    %rbx
  0x00000192d1972bde: mov    %eax,(%r14,%rbx,8)
  0x00000192d1972be2: movzbl 0x4(%r13),%ebx
  0x00000192d1972be7: add    $0x4,%r13
  0x00000192d1972beb: movabs $0x7fffd56e0fa0,%r10
  0x00000192d1972bf5: jmpq   *(%r10,%rbx,8)
  0x00000192d1972bf9: nopl   0x0(%rax)
```

接着 `generate_and_dispatch()` 又分为执行前(`dispatch_prolog`)+执行字节码(`t->generate()`)+执行后三部分(`dispatch_epilog`):

```
void TemplateInterpreterGenerator::generate_and_dispatch(Template* t, TosState tos_out) {
  ...
  int step = 0;
  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);
  // 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);
  }
}
```

x86的字节码执行前不会做任何事，所以没有其他代码：

```
----------------------------------------------------------------------
istore  54 istore  [0x00000192d1972ba0, 0x00000192d1972c00]  96 bytes
  ;获取栈顶int缓存
  0x00000192d1972ba0: mov    (%rsp),%eax
  0x00000192d1972ba3: add    $0x8,%rsp
  ; 执行istore，即移动bcp指针获取index，放入局部变量槽
  0x00000192d1972ba7: movzbl 0x1(%r13),%ebx
  0x00000192d1972bac: neg    %rbx
  0x00000192d1972baf: mov    %eax,(%r14,%rbx,8)

  0x00000192d1972bb3: movzbl 0x2(%r13),%ebx
  0x00000192d1972bb8: add    $0x2,%r13
  0x00000192d1972bbc: movabs $0x7fffd56e0fa0,%r10
  0x00000192d1972bc6: jmpq   *(%r10,%rbx,8)
  0x00000192d1972bca: mov    (%rsp),%eax
  0x00000192d1972bcd: add    $0x8,%rsp
  0x00000192d1972bd1: movzwl 0x2(%r13),%ebx
  0x00000192d1972bd6: bswap  %ebx
  0x00000192d1972bd8: shr    $0x10,%ebx
  0x00000192d1972bdb: neg    %rbx
```

```
  0x00000192d1972bde: mov    %eax,(%r14,%rbx,8)
  0x00000192d1972be2: movzbl 0x4(%r13),%ebx
  0x00000192d1972be7: add    $0x4,%r13
  0x00000192d1972beb: movabs $0x7fffd56e0fa0,%r10
  0x00000192d1972bf5: jmpq   *(%r10,%rbx,8)
  0x00000192d1972bf9: nopl   0x0(%rax)
```

执行后调用的是 `dispatch_prolog`：

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

void InterpreterMacroAssembler::dispatch_next(TosState state, int step) {
  // load next bytecode (load before advancing _bcp_register to prevent AGI)
  load_unsigned_byte(rbx, Address(_bcp_register, step));
  // advance _bcp_register
  increment(_bcp_register, 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);
  }
#ifdef _LP64
  // 防止意外执行到死代码
  lea(rscratch1, ExternalAddress((address)table));
  jmp(Address(rscratch1, rbx, Address::times_8));
#else
  Address index(noreg, rbx, Address::times_ptr);
  ExternalAddress tbl((address)table);
  ArrayAddress dispatch(tbl, index);
  jump(dispatch);
#endif // _LP64
}



----------------------------------------------------------------------
istore  54 istore  [0x00000192d1972ba0, 0x00000192d1972c00]  96 bytes
  ; 获取栈顶int缓存
  0x00000192d1972ba0: mov    (%rsp),%eax
  0x00000192d1972ba3: add    $0x8,%rsp

  ; 执行istore，即移动bcp指针获取index，放入局部变量槽
  0x00000192d1972ba7: movzbl 0x1(%r13),%ebx
  0x00000192d1972bac: neg    %rbx
  0x00000192d1972baf: mov    %eax,(%r14,%rbx,8)
```

```
; 加载下一个字节码，istore后面一个字节是index，所以需要r13+2
0x00000192d1972bb3: movzbl 0x2(%r13),%ebx
0x00000192d1972bb8: add    $0x2,%r13

; 防止意外执行到死代码
0x00000192d1972bbc: movabs $0x7fffd56e0fa0,%r10
0x00000192d1972bc6: jmpq   *(%r10,%rbx,8)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; 之前提到istore有一个wide版本的也会一并生成，wide istore格式如下
; wide istore byte1, byte2 [四个字节]
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 获取栈顶缓存的int
0x00000192d1972bca: mov    (%rsp),%eax
0x00000192d1972bcd: add    $0x8,%rsp

; 获取两个字节的index
0x00000192d1972bd1: movzwl 0x2(%r13),%ebx         ; 除两个字节的index外0填充，比如当前index分别为2,2,扩展后ebx=0x00000202
0x00000192d1972bd6: bswap  %ebx                   ; 4个字节反序，ebx=0x02020000
0x00000192d1972bd8: shr    $0x10,%ebx             ; ebx=0x00000202
0x00000192d1972bdb: neg    %rbx                   ; 取负数
0x00000192d1972bde: mov    %eax,(%r14,%rbx,8)     ; r14-rbx*8,

; 加载下一个字节码，wide istore byte1,byte2 所以r13+4
0x00000192d1972be2: movzbl 0x4(%r13),%ebx
0x00000192d1972be7: add    $0x4,%r13

; 防止意外执行到死代码
0x00000192d1972beb: movabs $0x7fffd56e0fa0,%r10
0x00000192d1972bf5: jmpq   *(%r10,%rbx,8)
0x00000192d1972bf9: nopl   0x0(%rax)
```