About Hotspot JVM interpreter and tos

Aug 12, 2019

OpenJDK source was based on: Jdk11u Changeset of 51,892: This is e86c5c20e188.

What is OpenJDK interpreter?

- Java byte code interpreter
- (Java) method is first executed by interpreter
- Execute the native code generated by JIT compilation (when conditions are met)

Figure 9 in Understanding the basic structure of the Hotspot VM is summarized here.

As an example of a simple Java bytecode, let's take a method that adds ints.

```
class Example1 {
    public static int add() {
        int x = 1;
        int y = 2;
        return x + y;
    }
}
```

This is compiled and the part corresponding to the add method is extracted from the class file as follows. Since the JVM is a stack machine, each Java byte code exchanges input and output via the stack.

```
$ javac Example1.java
$ javap -c -v Example1.class
  ...
  public static int add();
    descriptor: ()I
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=0
         0: iconst_1 // (int) 1 をスタックに push
         1: istore_0 // スタックから pop してローカル変数 0 番へ
         2: iconst_2 // (int) 2 を push
         3: istore_1 // pop してローカル変数 1 番へ
         4: iload_0  // ローカル変数 0 番 を push
         5: iload_1  // ローカル変数 1 番を push
         6: iadd     // スタックから val2, val1 と pop して (val1 + val2) を push
         7: ireturn  // スタック先頭の値を返す
      LineNumberTable:
        line 10: 0
        line 11: 2
        line 12: 4
  ...
```

Two types of Java byte code interpreter

As described in OpenJDK RuntimeOverview # Interpreter , there are two types of implementations.

- Cpp Interpreter
- Template Interpreter

Cpp Interpreter

CppInterpreter is one of the interpreter implementations in OpenJDK. Mentioned in the overview above

> a classic switch-statement loop

It can be said that the implementation adopts.

As the source, there is `src/hotspot/share/interpreter/bytecodeInterpreter.cpp`a main loop part.

```
// CppInterpreter 内で BytecodeInterpreter::run を使用
//
// 一部省略 & 単純化
void
BytecodeInterpreter::run(interpreterState istate) {
    ...
    register address pc = istate->bcp(); // bcp = byte code pointer
    ...
    while (1) {
        opcode = *pc;

        switch (opcode)
        {
        ...
        CASE(_i2f):       /* convert top of stack int to float */
            SET_STACK_FLOAT(VMint2Float(STACK_INT(-1)), -1);
            UPDATE_PC_AND_CONTINUE(1);
        ...
        CASE(_ireturn):
            {
            // Allow a safepoint before returning to frame manager.
            SAFEPOINT;

            goto handle_return;
            }
        }
        ...
    }
    ...
    handle_return: {
        ...
    }
}
```

Cpp Interpreter is fairly intuitive but not very good, so it will not be used unless you enable the flag in your development build. Normally, the Template Interpreter described below is used as the implementation of the interpreter.

Template Interpreter

- Generate dedicated native code for each Java byte code (immediately after starting the JVM)
- It feels like jumping between generated native code

To see the native code that is generated `-XX:+PrintInterpreter`, start the JVM with a flag provided for development (see previous article on how to build).

```
$ java -XX:+PrintInterpreter Hello | tee interpreter.log
```

The assembly generated for interpreter is output to interpreter.log.

For example, below is the native code generated for Java byte code called arraylength (for x86_64). arraylength is an instruction that "pops a reference to an array object from the operand stack and pushes its length".

```
arraylength  190 arraylength  [0x00007f6d3903c8a0, 0x00007f6d3903c8c0]  32 bytes

  # (必要なら) operand の取得
  0x00007f6d3903c8a0: pop    %rax
```

```
# メインの処理
0x00007f6d3903c8a1: mov    0xc(%rax),%eax       # 0xc(%rax) に配列オブジェクトならば長さが格納されている

# 次の Java byte code を読む準備
0x00007f6d3903c8a4: movzbl 0x1(%r13),%ebx       # ebx レジスタに次の Java byte code を保存
0x00007f6d3903c8a9: inc    %r13                 # bcp (byte code pointer) の increment
0x00007f6d3903c8ac: movabs $0x7f6d4ed7bd60,%r10 # r10 レジスタに dispatch table のベースアドレスを保存
0x00007f6d3903c8b6: jmpq   *(%r10,%rbx,8)       # 次の Java byte code 用のコードへ jmp
0x00007f6d3903c8ba: nop
0x00007f6d3903c8bb: nop
0x00007f6d3903c8bc: nop
0x00007f6d3903c8bd: nop
0x00007f6d3903c8be: nop
0x00007f6d3903c8bf: nop
```

First, the reference to the array object is set in the rax register by the pop instruction. This is the start address of the array object on the heap in the case of OpenJDK. If necessary in the comment, we say that this instruction may not be performed in consideration of tos described later (because it may start with the next mov instruction).

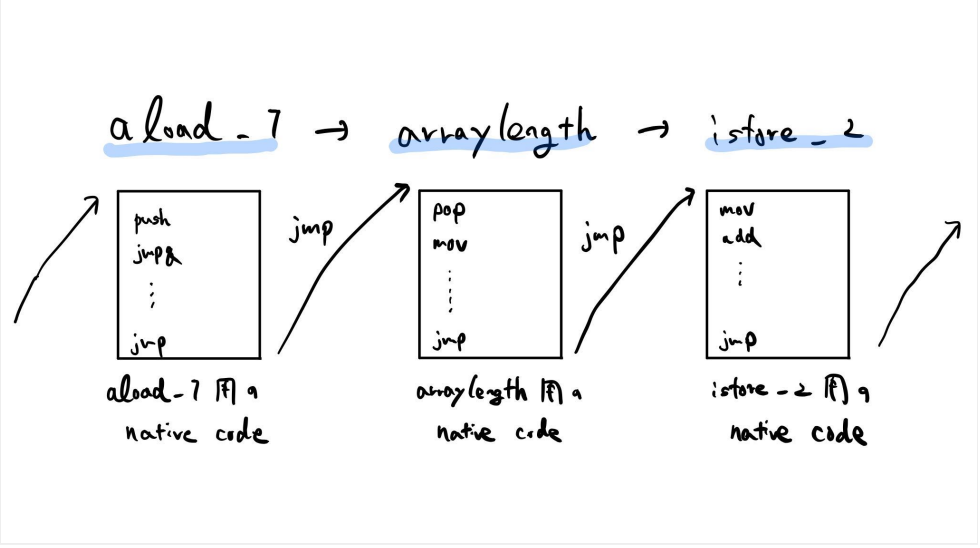
The following `mov 0xc(%rax),%eax`is. As described in the comment, here is the only processing unique to arraylength. The length information is pulled from the reference to the array object set in the rax register and set in the eax register. In the case of an array object (depending on the execution environment and options passed to the JVM at startup), the length information is to be entered in 4 bytes from the first 12 bytes. The fact that the information obtained here is set in the eax register instead of being pushed onto the stack is also explained in the section on tos described later.

After the mov instruction, prepare to process the next Java byte code.

- bcp increment
  - bcp is like a program counter for Java byte code
- Jmp to code for next Java byte code
  - You can fly without branching with the information of the number (+ α) assigned to each Java byte code

And so on. The processing around this is not limited to arraylength but is a regular pattern to be performed for each Java byte code.

This is how the processing performed by the Template Interpreter looks. For example, to process Java byte code of aload_1, arraylength, istore_2, first execute native code of alad_1, jmp to native code for arraylength, and then execute to native code for istore_2, jmp….



Processing by Template Interpreter

The content so far is in the source of OpenJDK

- Template
- TemplateTable
- DispatchTable

The perimeter is the applicable location.

Template

in `src/hotspot/share/interpreter/templateTable.hpp`

- native code generator for interpreter

```
class Template {
  ...

  typedef void (*generator)(int arg);

  int       _flags;        // describes interpreter template properties (bcp unknown)
  TosState  _tos_in;      // tos cache state before template execution
  TosState  _tos_out;     // tos cache state after  template execution
  generator _gen;         // template code generator
  int       _arg;         // argument for template code generator

  ...
}
```

TemplateTable

in `src/hotspot/share/interpreter/templateTable.hpp`

- A compilation of Template
- What native code to generate for each Java byte code can be tracked from the initialize method of TemplateTable
  - This method is called when the JVM starts

```
void TemplateTable::initialize() {
  ...

  //                                 interpr. templates
  // Java spec bytecodes             ubcp|disp|clvm|iswd  in    out   generator             argument

    def(Bytecodes::_arraylength    , ____|____|____|____, atos, itos, arraylength         ,  _           );

  ...
}
```

Each generator `src/hotspot/cpu/<arch>/templateTable_<arch>.cpp`presides in a naming source that is CPU dependent . For example, for Intel x86 `src/hotspot/cpu/x86/templateTable_x86.cpp`.

__ Is used to generate native code in what is called InterpreterMacroAssembler.

```
void TemplateTable::arraylength() {
  transition(atos, itos);
  __ null_check(rax, arrayOopDesc::length_offset_in_bytes());
  __ movl(rax, Address(rax, arrayOopDesc::length_offset_in_bytes()));
}
```

There is only "main processing" in the above assembly (`__movl`but that is). (transition is just an assetion, null_check does nothing by default)

DispatchTable

in `src/hotspot/share/interpreter/templateInterpreter.hpp`

- Manages the entry point (address) of generated code for each Java byte code interpreter
  - 256 (<type of Java byte code) address of x tos (Top Of Stack) 10 `_table`with the
- `set_entry` Set the entry point with
- `table_for` To get base address of table for each tos

```
class DispatchTable {
 public:
  enum { length = 1 << BitsPerByte }; // an entry point for each byte value
 public:
  address _table[number_of_states][length]; // dispatch tables, indexed by tosca and bytecode

  ...

  void       set_entry(int i, EntryPoint& entry);     // set entry point for a given bytecode i
  address*   table_for(TosState state)          { return _table[state]; }
}
```

Initialization of this table is `TemplateInterpreterGenerator::set_entry_points_for_all_bytes`done in.

About tos (Top Of Stack)

- What to expect as the type of the top value of the stack before and after each Java byte code processing
  - Since JVM is a stack machine, it can be said to be the input and output type of each instruction
  - Here, tos on the input side is called in tos and output side is called out tos
- If Java byte code is decided, in tos and out tos are decided
  - For example, if arraylength, in tos = atos, out tos = itos
- 10 types of tos in total
  - btos (byte, bool)
  - ztos (byte, bool)
  - ctos (char)
  - stos (short)
  - itos (int)
  - ltos (long)
  - ftos (float)
  - dtos (double)
  - atos (object)
  - vtos (void)

tos is a concept that does not exist in the JVM spec and is unique to OpenJDK. OpenJDK improves interpreter performance by considering tos.

Given that the JVM behaves like a stack machine, it is tempting to simply pass the input and output of each Java bytecode using the native stack. The performance of interpreter can be improved because it doesn't need to go. However, it is not always possible to exchange data using registers. To determine the condition, compare tos between consecutive Java byte codes.

The idea is

- For the time being, put the calculated value in the register and jmp to the next Java byte code processing
  - At this time, the jmp destination changes depending on the judgment of tos
- When processing the next Java byte code
  - If you can successfully exchange with the register, use it as it is
  - If not, put the passed value on the stack or get the value from the stack

It is like that.

For example, for the combination aload_1, arraylength,

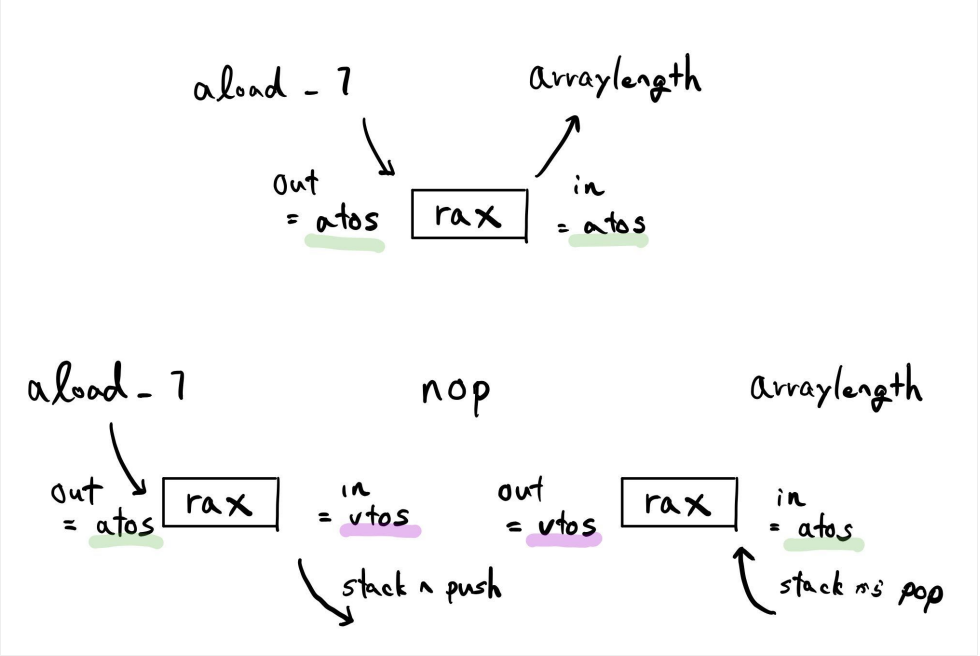
- aload_1's in tos is vtos, out tos is atos
- arraylength in tos is atos, out tos is itos

Since the out tos of the preceding Java byte code and the in tos of the subsequent Java byte code match, it is determined that it is okay to pass by register.

On the other hand, if it is an extreme case (aload_1, nop, arraylength)

- nop in tos is vtos, out tos is vtos

In this case, since the out tos of aload_1 and the in tos of nop do not match, passing in the register failed. For this reason, nop first puts the value on the register on the stack for subsequent processing, and then performs its own processing (although nothing is done because it is a nop). Again, tos does not match between nop and arraylength, and arraylength takes its value from the stack at the beginning and performs its own processing.



Changing the method of passing values by comparing tos

The following process generates the native code for this part.

```
void TemplateInterpreterGenerator::set_short_entry_points(Template* t, address& bep, address& cep, address& se
  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;
  }
}
```

Looking at this, we can eventually generalize to the following four patterns.

- Xtos to Xtos (Xtos is any other than vtos)
  - Processing proceeds using the value passed in the register. Most efficient
- vtos to Xtos
  - Failed in register passing (because tos is not prepared in the template before jmp)
  - There must be a value on the stack, so pop it and start processing.

- Xtos to vtos
  - Failed to pass in register (because tos is not used in jmp destination template)
  - Since it may be used in the processing of the subsequent Java byte code, the value passed in the register is put back on the stack before proceeding
- vtos to vtos
  - Do not pass by register
  - The actual source assumes that the pattern is the same as Xtos to vtos (just don't put it back on the stack)

Finally, check how the jmp destination address is managed in the DispatchTable using arraylength as an example. The following is the information confirmed from gdb, but the specific address value can change from run to run.

(If you see this information with gdb `TemplateInterpreter::_normal_table._table[8][190]`)

- `_table[atos][190]` Has 0x00007f6d3903c8a1
  - For example, the jmp destination from aload_1 in the order of aload_1, arraylength
  - Since the reference value of the array object should have been stored in the register in the previous aload_1 process, use it as it is
- `_table[vtos][190]` Has 0x00007f6d3903c8a0
  - For example, aload_1, nop, jmp destination from nop in array length
  - Nothing is output by the previous nop, but the reference value of the previous aload_1 should have been pushed on the stack, so start by popping it
- Otherwise jump to error handling routine (should)

```
arraylength  190 arraylength  [0x00007f6d3903c8a0, 0x00007f6d3903c8c0]  32 bytes
  0x00007f6d3903c8a0: pop    %rax
  0x00007f6d3903c8a1: mov    0xc(%rax),%eax       # 0xc(%rax) に配列オブジェクトならば長さが格納されている
  0x00007f6d3903c8a4: movzbl 0x1(%r13),%ebx       # ebx レジスタに次の Java byte code を保存
  0x00007f6d3903c8a9: inc    %r13                 # bcp (byte code pointer) の increment
  0x00007f6d3903c8ac: movabs $0x7f6d4ed7bd60,%r10 # r10 レジスタに dispatch table のベースアドレスを保存
  0x00007f6d3903c8b6: jmpq   *(%r10,%rbx,8)       # 次の Java byte code 用のコードへ jmp
  0x00007f6d3903c8ba: nop
  0x00007f6d3903c8bb: nop
  0x00007f6d3903c8bc: nop
  0x00007f6d3903c8bd: nop
  0x00007f6d3903c8be: nop
  0x00007f6d3903c8bf: nop
```

The assembly on the aload_1 side is

```
aload_1  43 aload_1  [0x00007fffe101f520, 0x00007fffe101f580]  96 bytes

  0x00007fffe101f520: push   %rax
  0x00007fffe101f521: jmpq   0x00007fffe101f55f
  0x00007fffe101f526: sub    $0x8,%rsp
  0x00007fffe101f52a: vmovss %xmm0,(%rsp)
  0x00007fffe101f52f: jmpq   0x00007fffe101f55f
  0x00007fffe101f534: sub    $0x10,%rsp
  0x00007fffe101f538: vmovsd %xmm0,(%rsp)
  0x00007fffe101f53d: jmpq   0x00007fffe101f55f
  0x00007fffe101f542: sub    $0x10,%rsp
  0x00007fffe101f546: mov    %rax,(%rsp)
  0x00007fffe101f54a: movabs $0x0,%r10
  0x00007fffe101f554: mov    %r10,0x8(%rsp)
  0x00007fffe101f559: jmpq   0x00007fffe101f55f
  0x00007fffe101f55e: push   %rax
  0x00007fffe101f55f: mov    -0x8(%r14),%rax
  0x00007fffe101f563: movzbl 0x1(%r13),%ebx
  0x00007fffe101f568: inc    %r13
  0x00007fffe101f56b: movabs $0x7ffff7b86d60,%r10 # r10 レジスタに dispatch table のベースアドレスを保存
  0x00007fffe101f575: jmpq   *(%r10,%rbx,8)
  0x00007fffe101f579: nop
  0x00007fffe101f57a: nop
  0x00007fffe101f57b: nop
```

```
  0x00007fffe101f57c: nop
  0x00007fffe101f57d: nop
  0x00007fffe101f57e: nop
  0x00007fffe101f57f: nop
```

It's like, but `table[atos]`I use atos for calculation of jmp destination . `jmp *(%r10,%rbx,8)`In `table[atos][190]`point jmp to the address of. In this way, DispatchTable provides a jmp destination for each tos and bytecode, so it can be processed without any branching at runtime.

In the case of nop, the same is `table[vtos]`used here, and the jmp destination is `table[vtos][190]`.

```
nop  0 nop  [0x00007fffe101c6a0, 0x00007fffe101c700]  96 bytes

  0x00007fffe101c6a0: push   %rax
  0x00007fffe101c6a1: jmpq   0x00007fffe101c6df
  0x00007fffe101c6a6: sub    $0x8,%rsp
  0x00007fffe101c6aa: vmovss %xmm0,(%rsp)
  0x00007fffe101c6af: jmpq   0x00007fffe101c6df
  0x00007fffe101c6b4: sub    $0x10,%rsp
  0x00007fffe101c6b8: vmovsd %xmm0,(%rsp)
  0x00007fffe101c6bd: jmpq   0x00007fffe101c6df
  0x00007fffe101c6c2: sub    $0x10,%rsp
  0x00007fffe101c6c6: mov    %rax,(%rsp)
  0x00007fffe101c6ca: movabs $0x0,%r10
  0x00007fffe101c6d4: mov    %r10,0x8(%rsp)
  0x00007fffe101c6d9: jmpq   0x00007fffe101c6df
  0x00007fffe101c6de: push   %rax
  0x00007fffe101c6df: movzbl 0x1(%r13),%ebx
  0x00007fffe101c6e4: inc    %r13
  0x00007fffe101c6e7: movabs $0x7ffff7b87560,%r10 # r10 レジスタに dispatch table のベースアドレスを保存
  0x00007fffe101c6f1: jmpq   *(%r10,%rbx,8)
  0x00007fffe101c6f5: nop
  0x00007fffe101c6f6: nop
  0x00007fffe101c6f7: nop
  0x00007fffe101c6f8: int3
  0x00007fffe101c6f9: int3
  0x00007fffe101c6fa: int3
  0x00007fffe101c6fb: int3
  0x00007fffe101c6fc: int3
  0x00007fffe101c6fd: int3
  0x00007fffe101c6fe: int3
  0x00007fffe101c6ff: int3
```