JVM之模板解释器

汪先生
中华儿女多奇志，不爱无码爱代码

关注他

欢迎关注我的专栏：半栈工程师

闲来无事，编译调试了下OpenJDK9，仔细研究了下HotSpot中的模板解释器。

一：何为模版解释器

C和C++之类的语言，会在编译期就直接编译成平台相关的机器指令，对于不同平台，可执行文件类型也不一样，如Linux为ELF，Windows为PE，而MacOS为Mach-O。而写Java的应该都清楚，java之所以跨平台性比较强，是因为Java在编译期没有被直接编译成机器指令，而是被编译成一种中间语言：字节码。

2016年我读完周志明的《深入理解Java虚拟机》后，不觉过瘾，便紧接着看完了张秀宏老师的《自己动手写Java虚拟机》，书中关于如何实现一个小型JVM做了详细讲解，其中一部分就是讲如何执行Class文件中方法体的字节码。

《自己动手写Java虚拟机》中对于字节码的执行其实就是简单的翻译，比如要实现iload指令（将指定的 int 型局部变量推送至栈顶），其实就用GO（这本书用GO来实现JVM的）来实现其对应的功能：

```
func _iload(frame *rtda.Frame, index uint) {
        val := frame.LocalVars().GetInt(index)
        frame.OperandStack().PushInt(val)
}
```

当执行方法中的iload指令时，就直接调用该_iload（ ）方法即可。

这种解释器简单明了，而且容易理解，要是让我们来实现虚拟机，估计想到的也是这种方法（虽然我没有那个能力）。早期的HotSpot就是通过上面这种方法来解释执行字节码指令的，这种解释器有个通用的称呼：字节码解释器。目前HotSpot中还保留着字节码解释器，只不过没有使用了。

字节码解释器的优点上面已经说过了，但是缺点也很明显：慢。每个字节码指令都要通过翻译执行，虽然在用C++写成的JVM中，类似上面_iload()这样的方法，最后也会被编译成机器指令，但

是编译器生成的机器指令很冗余，而CPU本身就是不断取指执行，指令越多，耗时也就越长。对于JVM的解释器来说，其实也就是不断取指执行，如果每个字节码指令的执行时间都很慢，那么整体效率必然很差。

早期的字节码解释器既然已经不能适应时代的发展，那么JVM的工程师想出了什么优化呢？上面提到字节码解释器慢是因为编译器生成的机器指令不够理想，那么我们直接跳过编译器，自己动手写汇编代码不就行了。没错，现在的HotSpot就是这样干的，这种解释器便称为模板解释器。

模板解释器相对于为每一个指令都写了一段实现对应功能的汇编代码，在JVM初始化时，汇编器会将汇编代码翻译成机器指令加载到内存中，比如执行iload指令时，直接执行对应的汇编代码即可。如何执行汇编代码？直接跳往汇编代码生成的机器指令在内存中的地址即可。HotSpot中很多地方都是利用手动汇编代码来优化效率的，在我的文章《JVM方法执行的来龙去脉》中也提到，方法的调用也是通过手动汇编代码来执行的。

二：模板解释器的创建

1：模板的初始化及机器指令的生成

我们平时说的iload指令等，其实都只是字节码指令的助记符，帮助我们理解，真正的自己码指令其实就是一个数字，比如iload是21，虚拟机执行21这个指令时，就是执行iload。字节码指令定义在bytecodes.hpp中：

```
class Bytecodes: AllStatic {
 public:
  enum Code {
    _illegal              =  -1,

    // Java bytecodes
    _nop                  =   0, // 0x00
    _aconst_null          =   1, // 0x01
    _iconst_m1            =   2, // 0x02
    _iconst_0             =   3, // 0x03
    _iconst_1             =   4, // 0x04
    _iconst_2             =   5, // 0x05
    _iconst_3             =   6, // 0x06
    _iconst_4             =   7, // 0x07
    _iconst_5             =   8, // 0x08
    _lconst_0             =   9, // 0x09
    ......
```

```
  }
}
```

JVM初始化时会为每个自己码指令都创建一个模板，每个模板都关联其对应的汇编代码生成函数：

```
void TemplateTable::initialize() {
  ......
  def(Bytecodes::_nop                , ____|____|____|____, vtos, vtos, nop
  def(Bytecodes::_aconst_null        , ____|____|____|____, vtos, atos, aconst_null
  def(Bytecodes::_iconst_m1          , ____|____|____|____, vtos, itos, iconst
  ......
  def(Bytecodes::_iload              , ubcp|____|clvm|____, vtos, itos, iload
  ......
}
```

def（）函数其实就是用来创建模板的：

```
void TemplateTable::def(Bytecodes::Code code, int flags, TosState in, TosState out, vo
  ......
  Template* t = is_wide ? template_for_wide(code) : template_for(code);
  // setup entry
  t->initialize(flags, in, out, gen, arg);
}
```

在调用def（）时，我们传入了一系列参数，其中倒数第二个参数为一个函数指针，其实这个函数指针指向的就是字节码指令对应的汇编代码生成函数。我们还是拿iload指令说事吧，在创建iload指令模板时，传入的函数指针为iload：

```
void TemplateTable::iload() {
  ......
  //获取局部变量slot号，放入rbx中
  locals_index(rbx);
  //将slot对应的局部变量移动至rax中
  __ movl(rax, iaddress(rbx));
}
```

iload（）函数会生成iload指令对应的机器指令。

在定义完成所有字节码对应的模板后，JVM会遍历所有字节码，为每个字节码生成对应的机器指令入口：

```
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);
    }
  }
}
```

set_entry_points(code)最终会调用TemplateInterpreterGenerator::generate_and_dispatch（）来生成机器指令：

```
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);
  }
}
```

在generate_and_dispatch（）中，会调用模版的generate（）方法，因为模板初始化时记录了对应的机器指令生成函数的指针，存在_gen中，所以这里直接调用_gen（）即可生成机器指令，对于iload来说，就相当于调用了TemplateTable::iload（）：

```
void Template::generate(InterpreterMacroAssembler* masm) {
  // parameter passing
```

```
  TemplateTable::_desc = this;
  TemplateTable::_masm = masm;
  // code generation
  _gen(_arg);
  masm->flush();
}
```

2：字节码派发表的创建

机器指令生成完成后，事情还没结束，因为我们需要记录机器指令的入口地址。在set_entry_points（ ）末尾，会创建一个EntryPoint记录生成的机器指令的入口，并将EntryPoint以字节码为索引，存储到Interpreter::_normal_table表中。注：因为字节码指令本身就是从0开始递增的：_nop = 0, _aconst_null = 1 ，.....。所以这里可以直接根据字节码指令作为索引。

```
// set entry points
  EntryPoint entry(bep, zep, cep, sep, aep, iep, lep, fep, dep, vep);
  Interpreter::_normal_table.set_entry(code, entry);
  Interpreter::_wentry_point[code] = wep;
```

其中Entrypoint定义如下：

```
EntryPoint::EntryPoint(address bentry, address zentry, address centry, address sentry,
  assert(number_of_states == 10, "check the code below");
  _entry[btos] = bentry;
  _entry[ztos] = zentry;
  _entry[ctos] = centry;
  _entry[stos] = sentry;
  _entry[atos] = aentry;
  _entry[itos] = ientry;
  _entry[ltos] = lentry;
  _entry[ftos] = fentry;
  _entry[dtos] = dentry;
  _entry[vtos] = ventry;
}
```

这里大家会看到很多btos、ztos之类的，这是TosState，即TopOfStackState，其实这描述的是当前栈顶数据的类型，栈顶数据类型不同时，会进入不同的entry。这部分用到的是栈顶缓存技

术，可参考《栈顶缓存（Top-of-Stack Cashing）技术》，大家只要记住，EntryPoint是用来记录机器指令入口地址即可。

3：取指执行过程

大家有没有想过，CPU是如何不断的执行指令的？难道有个统一的管理者，不断的取出下一条指令执行？其实代码段被加载到内存后，会放到连续的一块内存区域，每条指令都是线性排在一起的。CPU利用CS:IP寄存器来记录当前指令地址，因为指令都是连续排在一起的，所以当执行完一条指令后，直接根据当前指令长度进行偏移，就可以拿到下一条指令地址，送入IP寄存器，从而实现连续不断的取指。

HotSpot借用了这一思想，在每个字节码指令对应生成的机器指令末尾，会插入一段跳转下一条指令的逻辑。这样当前字节码在完成自己的功能后，就会自动取出方法体中排在它后面的下一条指令开始执行。

我们回到上面字节码机器指令生成的函数generate_and_dispatch（）中：

```
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);
  }
}
```

t->generate(_masm)后并没有立马撤走，而是会进行dispatch操作，调用 __ dispatch_epilog(tos_out, step)来进行下一条指令的执行，dispatch_epilog（）里面调用的是dispatch_next（）方法：

```
void InterpreterMacroAssembler::dispatch_next(TosState state, int step) {
  load_unsigned_byte(rbx, Address(_bcp_register, step));
```

```
  // advance _bcp_register
  increment(_bcp_register, step);
  dispatch_base(state, Interpreter::dispatch_table(state));
}
```

load_unsigned_byte（ ）会根据当前指令地址偏移，获取下条指令地址，并通过地址获得指令,放入rbx寄存器。何为bcp？即：bytecode pointer，所以_bcp_register存储的是当前字节码指令地址，

取指完成后，调用increment(_bcp_register, step)来更新rsi寄存器，使其指向下一条指令地址。

dispatch_base(state, Interpreter::dispatch_table(state))开始进行下一条指令的执行，Interpreter::dispatch_table(state)返回了之前生成的字节码派发表。

```
void InterpreterMacroAssembler::dispatch_base(TosState state,
                                              address* table,
                                              bool verifyoop) {
  ......
  lea(rscratch1, ExternalAddress((address)table));
  jmp(Address(rscratch1, rbx, Address::times_8));
}
```

lea(rscratch1, ExternalAddress((address)table)) 将存储指令对应的机器指令地址的DispatchTable内存地址放到rscratch1中。

jmp(Address(rscratch1, rbx, Address::times_8))：因为DispatchTable中索引直接为字节码指令，从0开始，而rbx现在存的就是下一条指令，所以可以通过（DispatchTable首地址 + rbx * 每个地址所占字节）来索引。然后直接利用jmp指令跳往字节码对应机器指令的地址。

三：总结

到这里模板解释器的大致逻辑就讲完了，主要分为以下几部分：

为每个字节码创建模板；
利用模板为每个字节码生成对应的机器指令；
将每个字节码生成的机器指令地址存储在派发表中；
在每个字节码生成的机器指令末尾，插入自动跳转下条指令逻辑。

HotSpot真是座宝库，学习HotSpot不仅仅是为了打开虚拟机这个黑匣子，更重要的是学习它的思想，从而将这种思想能为我们所用！

参考：《揭秘Java虚拟机：JVM设计原理与实现》