第8篇-dispatch_next()函数分派字节码

Original 鸠摩 深入剖析Java虚拟机HotSpot 2021-12-10 16:01

在generate_normal_entry()函数中会调用generate_fixed_frame()函数为Java方法的执行生成对应的栈帧，接下来还会调用dispatch_next()函数执行Java方法的字节码。generate_normal_entry()函数调用的dispatch_next()函数之前一些寄存器中保存的值如下：

```
rbx: Method*
ecx: invocation counter
r13: bcp(byte code pointer)
rdx: ConstantPool* 常量池的地址
r14: 本地变量表第1个参数的地址
```

dispatch_next()函数的实现如下：

```
// 从generate_fixed_frame()函数生成
// Java方法调用栈帧的时候，
// 如果当前是第一次调用，那么r13指向
// 的是字节码的首地址，
// 即第一个字节码，此时的step参数为0
void InterpreterMacroAssembler::dispatch_next(
    TosState state, int step) {
  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，这个值是Opcode（范围1~202），
  // 存储到rbxstep的值由字节码指令
  // 和它的操作数共同决定
  // 自增r13供下一次字节码分派使用
  increment(r13, step);

  // 返回当前栈顶状态的所有字节码入口点
```

```
  dispatch_base(state, Interpreter::dispatch_table(state));
}
```

r13指向字节码的首地址，当第1次调用时，参数step的值为0，那么load_unsigned_byte()函数从r13指向的内存中取一个字节的值，取出来的是字节码指令的操作码。增加r13的步长，这样下次执行时就会取出来下一个字节码指令的操作码。

调用的dispatch_table()函数的实现如下：

```
static address*
  dispatch_table(TosState state)  {
   return _active_table.table_for(state);
}
```

在_active_table中获取对应栈顶缓存状态的入口地址，_active_table变量定义在TemplateInterpreter类中，如下：

```
staticDispatchTable  _active_table;
```

DispatchTable类及table_for()等函数的定义如下：

```
DispatchTable TemplateInterpreter::_active_table;

class DispatchTable VALUE_OBJ_CLASS_SPEC {
 public:
  enum {
    length = 1 << BitsPerByte
  }; // BitsPerByte的值为8

 private:
  // number_of_states=9,length=256
  // _table是字节码分发表
  address _table[number_of_states][length];

 public:
  // ...
  address* table_for(TosState state){
    return _table[state];
  }

  address* table_for(){
    return table_for((TosState)0);
  }
```

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

address为u_char*类型的别名。_table是一个二维数组的表，维度为栈顶状态（共有9种）和字节码（最多有256个），存储的是每个栈顶状态对应的字节码的入口点。这里由于还没有介绍栈顶缓存，所以理解起来并不容易，不过后面会详细介绍栈顶缓存和字节码分发表的相关内容，等介绍完了再看这部分逻辑就比较容易理解了。

InterpreterMacroAssembler::dispatch_next()函数中调用的dispatch_base()函数的实现如下：

```
void InterpreterMacroAssembler::dispatch_base(
  TosState  state, // 表示栈顶缓存状态
  address* table,
  bool verifyoop
) {

  // ...

  // 获取当前栈顶状态字节码转发表
  // 的地址，保存到rscratch1
  lea(rscratch1,
     ExternalAddress((address)table));
  // 跳转到字节码对应的入口执行机器码指令
  // address = rscratch1 + rbx * 8
  jmp(Address(rscratch1,
     rbx, Address::times_8));
}
```

比如取一个字节大小的指令（如iconst_0、aload_0等都是一个字节大小的指令），那么InterpreterMacroAssembler::dispatch_next()函数生成的汇编代码如下：

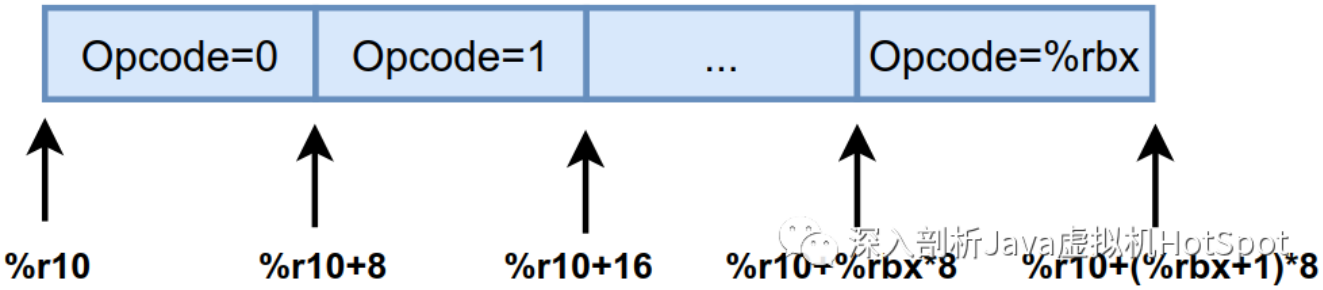
```
// 在generate_fixed_frame()函数中
// 已经让%r13存储了bcp
// %ebx中存储的是字节码的Opcode，也就是操作码
movzbl 0x0(%r13),%ebx

// $0x7ffff73ba4a0这个地址指向的
// 是对应state状态下的一维数组，长度为256
movabs $0x7ffff73ba4a0,%r10

// 注意%r10中存储的是常量，根据计算公式
// %r10+%rbx*8来获取指向存储入口地址的地址，
// 通过*(%r10+%rbx*8)获取到入口地址，
```

```
// 然后跳转到入口地址执行
jmpq *(%r10,%rbx,8)
```

%r10指向的是对应栈顶缓存状态state下的一维数组，长度为256，其中存储的值为opcode，如下图所示。

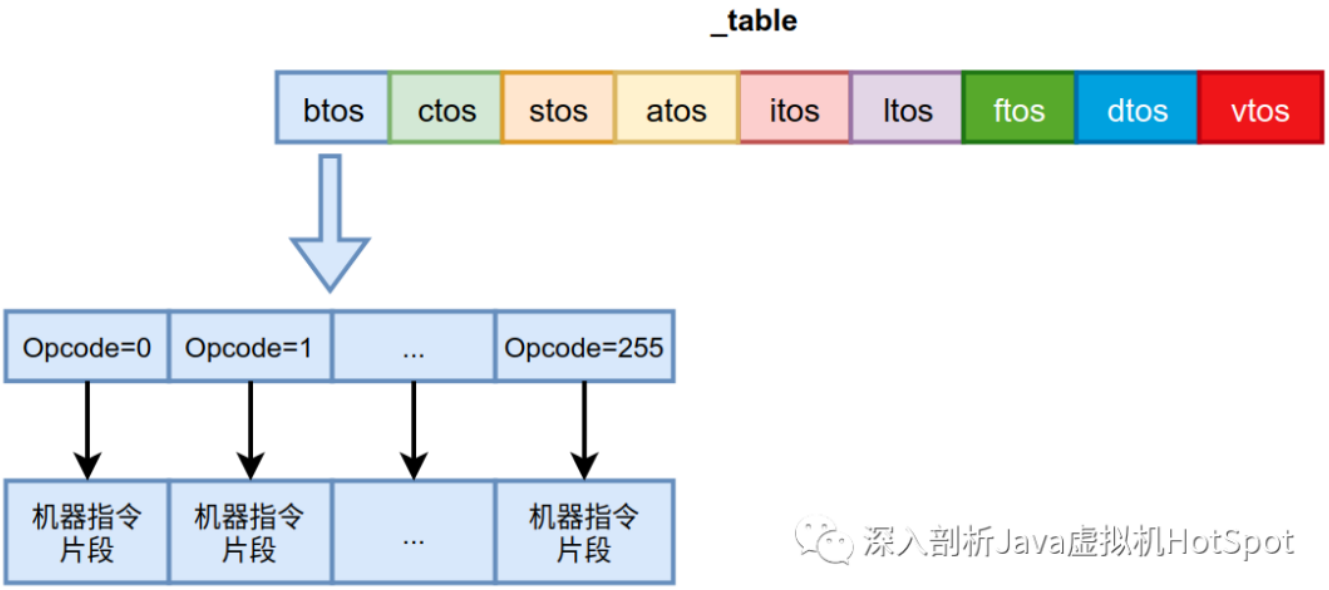


下面的函数显示了对每个字节码的每个栈顶状态都设置入口地址。

```
void DispatchTable::set_entry(
  int i, EntryPoint& entry) {
  _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);
}
```

其中的参数i就是opcode，各个字节码及对应的opcode可参考https://docs.oracle.com/javase/specs/jvms/se8/html/index.html。

所以_table表如下图所示。



_table的一维为栈顶缓存状态，二维为Opcode，通过这2个维度能够找到一段机器指令，这就是根据当前的栈顶缓存状态定位到的字节码需要执行的机器指令片段。

调用dispatch_next()函数执行Java方法的字节码，其实就是根据字节码找到对应的机器指令片段的入口地址来执行，这段机器码就是根据对应的字节码语义翻译过来的，这些都会在后面详细介绍。