第9篇-字节码指令的定义

Original 鸠摩 深入剖析Java虚拟机HotSpot 2021-12-11 15:39

收录于合集

#java 9 #运行时 9 #hotspot 10 #虚拟机 10

深入剖析Java虚拟机HotSpot

对HotSpot VM进行深度源码剖析，如果要系统的学习相关内容，推荐作者的《深入剖析Ja...

85篇原创内容

公众号

之前的文章介绍了解释执行下的Java栈帧创建以及字节码分派逻辑，但是始终没有讲到虚拟机到底是怎么执行Java方法中的字节码的，在介绍字节码的执行之前，需要先知道字节码指令的定义。在Bytecodes::initialize()函数中会定义字节码指令的一些属性。这个函数的调用链如下：

```
Bytecodes::initialize()
bytecodes_init()
init_globals()
```

在Bytecodes::initialize()函数中有类似这样的定义：

```
// bytecode bytecode name format wide f. result tp stk traps
def(_nop , "nop"                  , "b"    , NULL    , T_VOID , 0, false);
def(_aconst_null , "aconst_null"         , "b"    , NULL    , T_OBJECT , 1, false);
def(_iconst_m1 , "iconst_m1"           , "b"    , NULL    , T_INT , 1, false);
def(_iconst_0 , "iconst_0"            , "b"    , NULL    , T_INT , 1, false);
def(_iconst_1 , "iconst_1"            , "b"    , NULL    , T_INT , 1, false);
// ...
```

现在Java虚拟机规范定义的202个字节码指令都会向上图那样，调用def()函数进行定义，我们需要重点关注调用def()函数时传递的参数bytecode name、format等。下面一个一个解释，如下：

- bytecode name就是字节码名称；
- wide表示字节码前面是否可以加wild，如果可以，则值为"wbii"；
- result tp表示指令执行后的结果类型，如为T_ILLEGAL时，表示只参考当前字节码无法决定执行结果的类型，如_invokevirtual方法调用指令，结果类型应该为方法返回类型，但是此时只参数这个调用方法的字节码指令是无法决定的；
- stk表示对表达式栈深度的影响，如_nop指令不执行任何操作，所以对表达式栈的深度无影响，stk的值为0；当用_iconst_0向栈中压入0时，栈的深度增加1，所以stk的值为1。当为_lconst_0时，栈的深度会增加2；当为_lstore_0时，栈的深度会减少2；
- traps表示can_trap，这个比较重要，在后面会详细介绍；

- format，这个属性能表达2个意思，首先能表达字节码的格式，另外还能表示字节码的长度。

下面我们需要重点介绍一下format这个参数。format表示字节码的格式，当字符串中有一个字符时就是一个字节长度的字节码，当为2个字符时就是2个字节长度的字节码...，如_iconst_0就是一个字节宽度的字节码，_istore的format为"bi"，所以是2个字节宽度。format还可能为空字符串，当为空字符串时，表示当前的字节码不是Java虚拟机规范中定义的字节码，如为了提高解释执行效率的_fast_agetfield、_fast_bgetfield等字节码，这些字节码是虚拟机内部定义的。还能表达字节码的格式，其中的字符串中各个字符的含义如下：

- b：表示字节码指令是非可变长度的，所以对于tableswitch、lookupswitch这种可变长度的指令来说，format字符串中不会含有b字符；
- c：操作数为有符号的常量，如bipush指令将byte带符号扩展为一个int类型的值，然后将这个值入栈到操作数栈中；
- i：操作数为无符号的本地变量表索引值，如iload指令从局部变量表加载一个int类型的值到操作数栈中；
- j：操作数为常量池缓存的索引，注意常量池缓存索引不同与常量池索引，关于常量池索引索引，在《深入剖析Java虚拟机：源码剖析与实例详解》基础卷中详细介绍过，这里不再介绍；
- k：操作数为无符号的常量池索引，如ldc指令将从运行时常量池中提取数据并压入操作数栈，所以格式为"bk"；
- o：操作数为分支偏移，如ifeq表示整数与零比较，如果整数为0，则比较结果为真，将操作数看为分支偏移量进行跳转，所以格式为”boo“；
- _：可直接忽略
- w：可用来扩展局部变量表索引的字节码，这些字节码有iload、fload等，所以wild的值为"wbii"；

调用的def()函数的实现如下：

```
void Bytecodes::def(
Code code,
const char* name,
const char* format,
const char* wide_format,
BasicType result_type,
int           depth,
bool          can_trap,
Code java_code
) {
  int len = (format != NULL ? (int) strlen(format) : 0);
  int wlen = (wide_format != NULL ? (int) strlen(wide_format) : 0);
```

```
  _name [code] = name;
  _result_type [code] = result_type;
  _depth [code] = depth;
  _lengths [code] = (wlen << 4) | (len & 0xF); // 0xF的二进制值为1111
  _java_code [code] = java_code;


  int bc_flags = 0;
  if (can_trap){
    // ldc、ldc_w、ldc2_w、_aload_0、iaload、iastore、idiv、ldiv、ireturn等
    // 字节码指令都会含有_bc_can_trap
    bc_flags |= _bc_can_trap;
  }
  if (java_code != code){
    bc_flags |= _bc_can_rewrite; // 虚拟机内部定义的指令都会有_bc_can_rewrite
  }

  // 在这里对_flags赋值操作
  _flags[(u1)code+0*(1<<BitsPerByte)] = compute_flags(format, bc_flags);
  _flags[(u1)code+1*(1<<BitsPerByte)] = compute_flags(wide_format, bc_flags);
}
```

其中的_name、_result_type等都是在Bytecodes类中定义的静态数组，其下标为Opcode值，而存储的值就是name、result_type等。这些变量的定义如下：

```
const char* Bytecodes::_name [Bytecodes::number_of_codes];
BasicType Bytecodes::_result_type [Bytecodes::number_of_codes];
s_char Bytecodes::_depth [Bytecodes::number_of_codes];
u_char Bytecodes::_lengths [Bytecodes::number_of_codes];
Bytecodes::Code Bytecodes::_java_code [Bytecodes::number_of_codes];
u_short Bytecodes::_flags [(1<<BitsPerByte)*2];
```

Bytecodes::number_of_codes的值为234，足够存储所有的字节码指令了（包含虚拟机内部扩展的指令）。

回看Bytecodes::def()函数，通过调用compute_flags()函数根据传入的wide_format和format来计算字节码的一些属性，然后存储到高8位和低8位中。调用的compute_flags()函数的实现如下：

```
int Bytecodes::compute_flags(const char* format, int more_flags) {
  if (format == NULL) {
    return 0; // not even more_flags
  }

  int flags = more_flags;
```

```
const char* fp = format;
switch (*fp) {
case '\0':
  flags |= _fmt_not_simple; // but variable
  break;
case 'b':
  flags |= _fmt_not_variable; // but simple
  ++fp; // skip 'b'
  break;
case 'w':
  flags |= _fmt_not_variable | _fmt_not_simple;
  ++fp; // skip 'w'
  guarantee(*fp == 'b', "wide format must start with 'wb'");
  ++fp; // skip 'b'
  break;
}

int has_nbo = 0, has_jbo = 0, has_size = 0;
for (;;) {
  int this_flag = 0;
  char fc = *fp++;
  switch (fc) {
  case '\0': // end of string
    assert(flags == (jchar)flags, "change _format_flags");
    return flags;

  case '_': continue; // ignore these

  case 'j': this_flag = _fmt_has_j; has_jbo = 1; break;
  case 'k': this_flag = _fmt_has_k; has_jbo = 1; break;
  case 'i': this_flag = _fmt_has_i; has_jbo = 1; break;
  case 'c': this_flag = _fmt_has_c; has_jbo = 1; break;
  case 'o': this_flag = _fmt_has_o; has_jbo = 1; break;

  case 'J': this_flag = _fmt_has_j; has_nbo = 1; break;
  ...
  default: guarantee(false, "bad char in format");
  }// 结束switch

  flags |= this_flag;

  guarantee(!(has_jbo && has_nbo), "mixed byte orders in format");
```

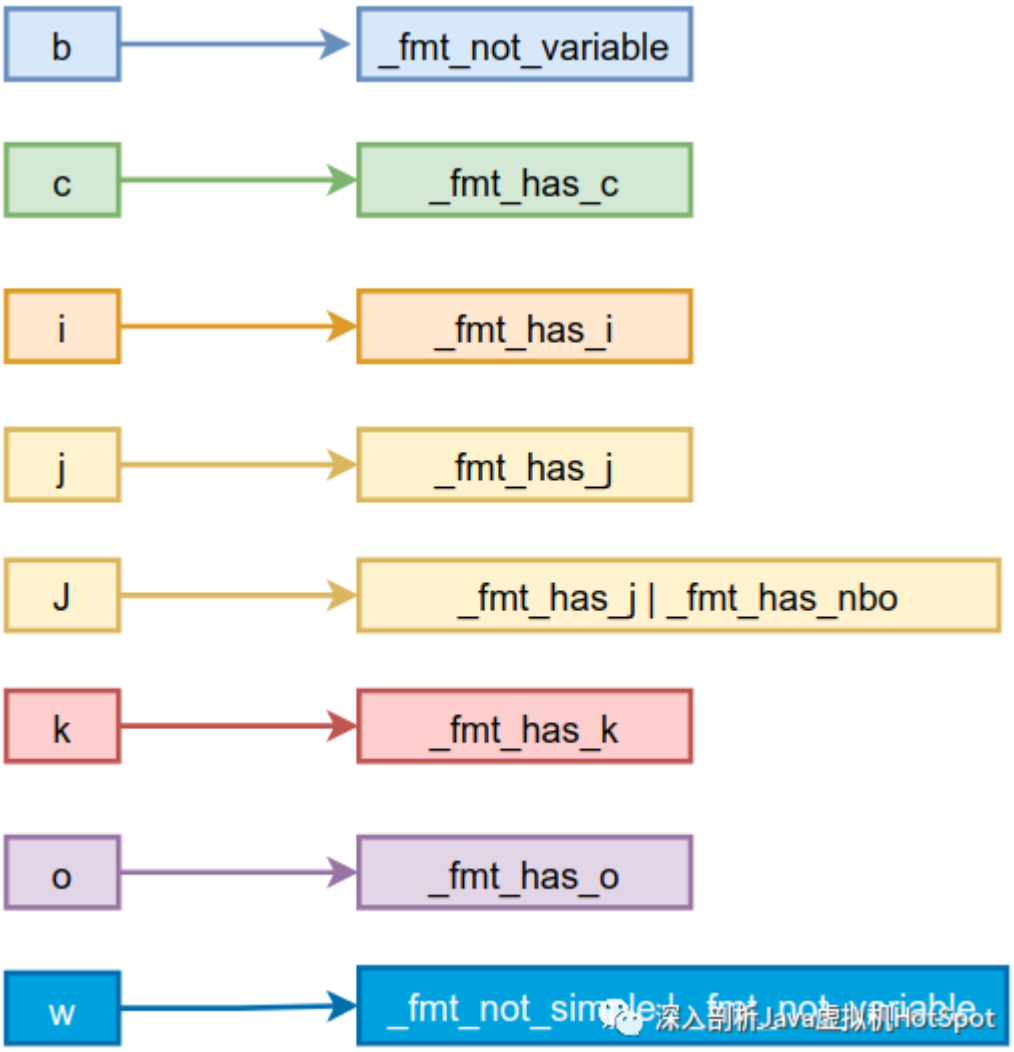
```
    if (has_nbo){
      flags |= _fmt_has_nbo;
    }

    int this_size = 1;
    if (*fp == fc) {
      // advance beyond run of the same characters
      this_size = 2;
      while (*++fp == fc){
        this_size++;
      }
      switch (this_size) {
      case 2: flags |= _fmt_has_u2; break; // 如sipush、ldc_w、ldc2_w、wide iload等
      case 4: flags |= _fmt_has_u4; break; // 如goto_w和invokedynamic指令
      default:
        guarantee(false, "bad rep count in format");
      }
    }

    has_size = this_size;
  }
}
```

函数要根据wide_format和format来计算flags的值，通过flags中的值能够表示字节码的b、c、i、j、k、o、w（在之前介绍format时介绍过）和字节码操作数的大小（操作数是2字节还是4字节）。以_fmt开头的一些变量在枚举类中已经定义，如下：

```
// Flag bits derived from format strings, can_trap, can_rewrite, etc.:
enum Flags {
// semantic flags:
_bc_can_trap = 1<<0, // bytecode execution can trap(卡住) or block
// 虚拟机内部定义的字节码指令都会含有这个标识
_bc_can_rewrite = 1<<1, // bytecode execution has an alternate(代替者) form

// format bits (determined only by the format string):
_fmt_has_c = 1<<2, // constant, such as sipush "bcc"
_fmt_has_j = 1<<3, // constant pool cache index, such as getfield "bjj"
_fmt_has_k = 1<<4, // constant pool index, such as ldc "bk"
_fmt_has_i = 1<<5, // local index, such as iload
_fmt_has_o = 1<<6, // offset, such as ifeq

_fmt_has_nbo = 1<<7, // contains native-order field(s)
```

```
_fmt_has_u2 = 1<<8, // contains double-byte field(s)
_fmt_has_u4 = 1<<9, // contains quad-byte field
_fmt_not_variable = 1<<10, // not of variable length (simple or wide) 不可变长度的指令
_fmt_not_simple = 1<<11, // either wide or variable length 或者是可加wild的字节码指令，或者是可变长度的指令
_all_fmt_bits = (_fmt_not_simple*2 - _fmt_has_c),

// ...
};
```

与format的对应关系如下：



这样通过组合就可表示出不同的值，枚举类中定义了常用的组合如下：

```
_fmt_b      = _fmt_not_variable,
_fmt_bc     = _fmt_b | _fmt_has_c,
_fmt_bi     = _fmt_b | _fmt_has_i,
_fmt_bkk    = _fmt_b | _fmt_has_k | _fmt_has_u2,
_fmt_bJJ    = _fmt_b | _fmt_has_j | _fmt_has_u2 | _fmt_has_nbo,
```

```
_fmt_bo2    = _fmt_b | _fmt_has_o | _fmt_has_u2,
_fmt_bo4    = _fmt_b | _fmt_has_o | _fmt_has_u4
```

例如字节码为bipush时，format就是"bc"，那么flags的值为_fmt_b | _fmt_has_c，ldc字节码的format为"bk"，则flags的值为_fmt_b | _fmt_has_k。

收录于合集 #java 9

上一篇
第8篇-dispatch_next()函数分派字节码

下一篇
第10篇-初始化模板表