第10篇-初始化模板表

Original 鸠摩 深入剖析Java虚拟机HotSpot 2021-12-12 17:08

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

在第9篇我们介绍了字节码指令并且将字节码指令相关的信息都存储到了相关数组中，只需要通过Opcode就可从相关数组中获取对应的信息。

在init_globals()函数中调用bytecodes_init()函数初始化好字节码指令后会调用interpreter_init()函数初始化解释器。函数最终会调用到TemplateInterpreter::initialize()函数。这个函数的实现如下：

源代码位置：/src/share/vm/interpreter/templateInterpreter.cpp

```
void TemplateInterpreter::initialize() {
  if (_code != NULL)
        return;

  // 抽象解释器AbstractInterpreter的初始化，
  // AbstractInterpreter是基于汇编模型的解释器的共同基类，
  // 定义了解释器和解释器生成器的抽象接口
  AbstractInterpreter::initialize();

  // 模板表TemplateTable的初始化，模板表TemplateTable保存了各个字节码的模板
  TemplateTable::initialize();

  // generate interpreter
  {
     ResourceMark rm;
     int code_size = InterpreterCodeSize;
     // CodeCache的Stub队列StubQueue的初始化
     _code = new StubQueue(new InterpreterCodeletInterface, code_size, NULL,"Interpreter");
     // 实例化模板解释器生成器对象TemplateInterpreterGenerator
```

```
    InterpreterGenerator g(_code);
  }

  // 初始化字节分发表
  _active_table = _normal_table;
}
```

这个初始化函数中涉及到的初始化逻辑比较多，而且比较复杂。我们将初始化分为4部分：

1. 抽象解释器AbstractInterpreter的初始化，AbstractInterpreter是基于汇编模型的解释器的共同基类，定义了解释器和解释器生成器的抽象接口；
2. 模板表TemplateTable的初始化，模板表TemplateTable保存了各个字节码的模板（目标代码生成函数和参数）；
3. CodeCache的Stub队列StubQueue的初始化；
4. 解释器生成器InterpreterGenerator的初始化。

其中抽象解释器初始化时会涉及到一些计数，这些计数主要与编译执行有关，所以这里暂不过多介绍，到后面介绍编译执行时再介绍。

下面我们分别介绍如上3个部分的初始化过程，这一篇只介绍模板表的初始化过程。

函数TemplateTable::initialize()的实现如下：

源代码位置：/src/share/vm/interpreter/templateInterpreter.cpp

```
void TemplateTable::initialize() {
  if (_is_initialized) return;



  _bs = Universe::heap()->barrier_set();

  // For better readability
  const char _ = ' ';
  const int ____ = 0;
  const int ubcp = 1 << Template::uses_bcp_bit;
  const int disp = 1 << Template::does_dispatch_bit;
  const int clvm = 1 << Template::calls_vm_bit;
  const int iswd = 1 << Template::wide_bit;
  // interpr. templates
  // Java spec bytecodes ubcp|disp|clvm|iswd in out generator argument
  def(Bytecodes::_nop , ____|____|____|____, vtos, vtos, nop , _ );
  def(Bytecodes::_aconst_null , ____|____|____|____, vtos, atos, aconst_null , _ );
  def(Bytecodes::_iconst_m1 , ____|____|____|____, vtos, itos, iconst , -1           );
```

```
def(Bytecodes::_iconst_0 , ____|____|____|____, vtos, itos, iconst , 0           );
// ...
def(Bytecodes::_tableswitch , ubcp|disp|____|____, itos, vtos, tableswitch , _ );
def(Bytecodes::_lookupswitch , ubcp|disp|____|____, itos, itos, lookupswitch , _ );
def(Bytecodes::_ireturn , ____|disp|clvm|____, itos, itos, _return , itos );
def(Bytecodes::_lreturn , ____|disp|clvm|____, ltos, ltos, _return , ltos );
def(Bytecodes::_freturn , ____|disp|clvm|____, ftos, ftos, _return , ftos );
def(Bytecodes::_dreturn , ____|disp|clvm|____, dtos, dtos, _return , dtos );
def(Bytecodes::_areturn , ____|disp|clvm|____, atos, atos, _return , atos );
def(Bytecodes::_return , ____|disp|clvm|____, vtos, vtos, _return , vtos );
def(Bytecodes::_getstatic , ubcp|____|clvm|____, vtos, vtos, getstatic , f1_byte );
def(Bytecodes::_putstatic , ubcp|____|clvm|____, vtos, vtos, putstatic , f2_byte );
def(Bytecodes::_getfield , ubcp|____|clvm|____, vtos, vtos, getfield , f1_byte );
def(Bytecodes::_putfield , ubcp|____|clvm|____, vtos, vtos, putfield , f2_byte );
def(Bytecodes::_invokevirtual , ubcp|disp|clvm|____, vtos, vtos, invokevirtual , f2_byte );
def(Bytecodes::_invokespecial , ubcp|disp|clvm|____, vtos, vtos, invokespecial , f1_byte );
def(Bytecodes::_invokestatic , ubcp|disp|clvm|____, vtos, vtos, invokestatic , f1_byte );
def(Bytecodes::_invokeinterface , ubcp|disp|clvm|____, vtos, vtos, invokeinterface , f1_byte );
def(Bytecodes::_invokedynamic , ubcp|disp|clvm|____, vtos, vtos, invokedynamic , f1_byte );
def(Bytecodes::_new , ubcp|____|clvm|____, vtos, atos, _new , _ );
def(Bytecodes::_newarray , ubcp|____|clvm|____, itos, atos, newarray , _ );
def(Bytecodes::_anewarray , ubcp|____|clvm|____, itos, atos, anewarray , _ );
def(Bytecodes::_arraylength , ____|____|____|____, atos, itos, arraylength , _ );
def(Bytecodes::_athrow , ____|disp|____|____, atos, vtos, athrow , _ );
def(Bytecodes::_checkcast , ubcp|____|clvm|____, atos, atos, checkcast , _ );
def(Bytecodes::_instanceof , ubcp|____|clvm|____, atos, itos, instanceof          , _ );
def(Bytecodes::_monitorenter , ____|disp|clvm|____, atos, vtos, monitorenter , _ );
def(Bytecodes::_monitorexit , ____|____|clvm|____, atos, vtos, monitorexit , _ );
def(Bytecodes::_wide , ubcp|disp|____|____, vtos, vtos, wide , _ );
def(Bytecodes::_multianewarray , ubcp|____|clvm|____, vtos, atos, multianewarray , _ );
def(Bytecodes::_ifnull , ubcp|____|clvm|____, atos, vtos, if_nullcmp , equal );
def(Bytecodes::_ifnonnull , ubcp|____|clvm|____, atos, vtos, if_nullcmp , not_equal );
def(Bytecodes::_goto_w , ubcp|____|clvm|____, vtos, vtos, goto_w , _ );
def(Bytecodes::_jsr_w , ubcp|____|____|____, vtos, vtos, jsr_w , _ );

// wide Java spec bytecodes
def(Bytecodes::_iload , ubcp|____|____|iswd, vtos, itos, wide_iload , _ );
def(Bytecodes::_lload , ubcp|____|____|iswd, vtos, ltos, wide_lload , _ );
// ...

// JVM bytecodes
// ...
```

```
  def(Bytecodes::_shouldnotreachhere , ____|____|____|____, vtos, vtos, shouldnotreachhere , _ );
}
```

TemplateTable的初始化调用def()将所有字节码的目标代码生成函数和参数保存在_template_table或_template_table_wide（wide指令）模板数组中。除了虚拟机规范本身定义的字节码指令外，HotSpot虚拟机也定义了一些字节码指令，这些指令为了辅助虚拟机进行更好的功能实现，例如Bytecodes::_return_register_finalizer等在之前已经介绍过，可以更好的实现finalizer类型对象的注册功能。

我们只给出部分字节码指令的模板定义，调用def()函数对每个字节码指令的模板进行定义，传递的参数是我们关注的重点：

（1）指出为哪个字节码指令定义模板

（2）ubcp|disp|clvm|iswd，这是一个组合数字，具体的数字与Template中定义的枚举类紧密相关，枚举类中定义的常量如下：

```
enum Flags {
 // set if template needs the bcp pointing to bytecode
 uses_bcp_bit,
 // set if template dispatches on its own 就其本身而言；靠自己
 does_dispatch_bit,
 // set if template calls the vm
 calls_vm_bit,
 // set if template belongs to a wide instruction
 wide_bit
};
```

下面详细解释这几个参数，如下：

- uses_bcp_bit，标志需要使用字节码指针（byte code pointer，数值为字节码基址+字节码偏移量）。表示生成的模板代码中是否需要使用指向字节码指令的指针，其实也就是说是否需要读取字节码指令的操作数，所以含有操作数的指令大部分都需要bcp，但是有一些是不需要的，如monitorenter与monitorexit等，这些的操作数都在表达式栈中，表达式栈顶就是其操作数，并不需要从Class文件中读取，所以不需要bcp；
- does_dispatch_bit，标志表示自己是否含有控制流转发逻辑，如tableswitch、lookupswitch、invokevirtual、ireturn等字节码指令，本身就需要进行控制流转发；
- calls_vm_bit，标志是否需要调用JVM函数，在调用TemplateTable::call_VM()函数时都会判断是否有这个标志，通常方法调用JVM函数时都会通过调用TemplateTable::call_VM()函数来间接完成调用。JVM函数就是用C++写的函数；
- wide_bit，标志是否是wide指令（使用附加字节扩展全局变量索引）。

（3）_tos_in与_tos_out：表示模板执行前与模板执行后的TosState（操作数栈栈顶元素的数据类型，TopOfStack，用来检查模板所声明的输出输入类型是否和该函数一致，以确保栈顶元素被正确使用）。

_tos_in与_tos_out的值必须是枚举类中定义的常量，如下：

```
enum TosState { // describes the tos cache contents
  btos = 0, // byte, bool tos cached
  ctos = 1, // char tos cached
  stos = 2, // short tos cached
  itos = 3, // int tos cached
  ltos = 4, // long tos cached
  ftos = 5, // float tos cached
  dtos = 6, // double tos cached
  atos = 7, // object cached
  vtos = 8, // tos not cached
  number_of_states,
  ilgl // illegal state: should not occur
};
```

如iload指令，执行之前栈顶状态为vtos，表示并不会使用栈顶的数据，所以如果程序为了提高执行效率将上一次执行的结果缓存到了寄存器中，那么此时就应该在执行iload指令之前将这个寄存器的值压入栈顶。iload指令执行之后的栈顶状态为itos，因为iload是向操作数栈中压入一个整数，所以此时的栈顶状态为int类型，那么这个值可以缓存到寄存器中，假设下一个指令为ireturn，那么栈顶之前与之后的状态分别为itos和itos，那么可直接将缓存在寄存器中的int类型返回即可，不需要做任何和操作数栈相关的操作。

（4）_gen与_arg：_gen表示模板生成器（函数指针），这个函数会为对应的字节码生成对应的执行逻辑；_arg表示为模板生成器传递的参数。调用函数指针会为每个字节码指令按其语义针对不同的平台上生成不同的机器指令，这里我们只讨论x86架构下64位的机器指令实现，由于机器指令很难读懂，所以我们后续只阅读由机器指令反编译的汇编指令。

下面看一下TemplateTable::initialize()函数中调用的Template::def()函数，如下：

```
void TemplateTable::def(
  Bytecodes::Code code, // 字节码指令
  int flags, // 标志位
  TosState in, // 模板执行前TosState
  TosState out, // 模板执行后TosState
  void (*gen)(int arg), // 模板生成器，是模板的核心组件
  int arg
) {
  // 表示是否需要bcp指针
  const int ubcp = 1 << Template::uses_bcp_bit;
  // 表示是否在模板范围内进行转发
  const int disp = 1 << Template::does_dispatch_bit;
  // 表示是否需要调用JVM函数
```

```
  const int clvm = 1 << Template::calls_vm_bit;
  // 表示是否为wide指令
  const int iswd = 1 << Template::wide_bit;

  // 如果是允许在字节码指令前加wide字节码指令的一些指令，那么
  // 会使用_template_table_wild模板数组进行字节码转发，否则
  // 使用_template_table模板数组进行转发
  bool is_wide = (flags & iswd) != 0;
  Template* t = is_wide ? template_for_wide(code) : template_for(code);

  // 调用模板表t的initialize()方法初始化模板表
  t->initialize(flags, in, out, gen, arg);
}
```

模板表由模板表数组与一组生成器组成：

模板数组有_template_table与_template_table_wild，数组的下标为字节码的Opcode，值为Template。定义如下：

```
Template TemplateTable::_template_table[Bytecodes::number_of_codes];
Template TemplateTable::_template_table_wide[Bytecodes::number_of_codes];
```

模板数组的值为Template，这个Template类中定义了保存标志位flags的_flags属性，保存栈顶缓存状态in和out的_tos_in和_tos_out，还有保存生成器gen及参数arg的_gen与_arg，所以调用t->initialize()后其实是初始化Template中的变量。initialize()函数的实现如下：

```
void Template::initialize(
 int flags,
 TosState tos_in,
 TosState tos_out,
 generator gen,
 int arg
) {
  _flags   = flags;
  _tos_in  = tos_in;
  _tos_out = tos_out;
  _gen     = gen;
  _arg     = arg;
}
```

不过这里并不会调用gen函数生成对应的汇编代码，只是将传递给def()函数的各种信息保存到Template实例中，在TemplateTable::def()函数中，通过template_for()或template_for_wild()函数获取到数组中对应的Template实例后，就会调用Template::initialize()函数将信息保存到对应的

Template实例中，这样就可以根据字节码索引从数组中获取对应的Template实例，进而获取字节码指令模板的相关信息。

虽然这里并不会调用gen来生成字节码指令对应的机器指令，但是我们可以提前看一下gen这个指针函数是怎么为某个字节码指令生成对应的机器指令的。

看一下TemplateTable::initialize()函数中对def()函数的调用，以_iinc（将局部变量表中对应的slot位的值增加1）为例，调用如下：

```
def(
 Bytecodes::_iinc, // 字节码指令
 ubcp|____|clvm|____, // 标志
 vtos, // 模板执行前的TosState
 vtos, // 模板执行后的TosState
 iinc , // 模板生成器，是一个iinc()函数的指针
 _ // 不需要模板生成器参数
);
```

设置标志位uses_bcp_bit和calls_vm_bit，表示iinc指令的生成器需要使用bcp指针函数at_bcp()，且需要调用JVM函数，下面给出了生成器的定义：

```
源代码位置：/hotspot/src/cpu/x86/vm/templateTable_x86_64.cpp


void TemplateTable::iinc() {
  transition(vtos, vtos);
  __ load_signed_byte(rdx, at_bcp(2)); // get constant
  locals_index(rbx);
  __ addl(iaddress(rbx), rdx);
}
```

由于iinc指令只涉及到对局部变量表的操作，并不会影响操作数栈，也不需要使用操作数栈顶的值，所以栈顶之前与之后的状态为vtos与vtos，调用transition()函数只是验证栈顶缓存的状态是否正确。

iinc指令的字节码格式如下：

```
iinc
index // 局部变量表索引值
const // 将局部变量表索引值对应的slot值加const
```

操作码iinc占用一个字节，而index与const分别占用一个字节。使用at_bcp()函数获取iinc指令的操作数，2表示偏移2字节，所以会将const取出来存储到rdx中。调用locals_index()函数取出index，locals_index()就是JVM函数。最终生成的汇编如下：

```
// %r13存储的是指向字节码的指针，偏移
// 2字节后取出const存储到%edx
movsbl 0x2(%r13),%edx
```

```
// 取出index存储到%ebx
movzbl 0x1(%r13),%ebx
neg    %rbx
// %r14指向本地变量表的首地址，将%edx加到
// %r14+%rbx*8指向的内存所存储的值上
// 之所以要对%rbx执行neg进行符号反转，
// 是因为在Linux内核的操作系统上，
// 栈是向低地址方向生长的
add    %edx,(%r14,%rbx,8)
```

注释解释的已经非常清楚了，这里不再过多介绍。

收录于合集 #java 9

上一篇 · 第9篇-字节码指令的定义