第6篇-Java方法新栈帧的创建

Original 鸠摩 深入剖析Java虚拟机HotSpot 2021-12-08 15:35

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2篇-关于运行时的call_helper()函数 介绍JavaCalls::call_helper()函数的实现时提到过如下一句代码：

```
address entry_point = method->from_interpreted_entry();
```

这个参数会做为实参传递给StubRoutines::call_stub()函数指针指向的“函数”，然后在 第4篇-JVM终于开始调用Java主类的main()方法啦 介绍到通过callq指令调用entry_point，那么这个entry_point到底是什么呢？这一篇我们将详细介绍。

首先看from_interpreted_entry()函数实现，如下：

```
volatile address from_interpreted_entry()const{
 return (address)OrderAccess::
    load_ptr_acquire(&_from_interpreted_entry);
}
```

_from_interpreted_entry只是Method类中定义的一个属性，如上方法直接返回了这个属性的值。那么这个属性是何时赋值的？其实是在方法连接（也就是在类的生命周期中的类连接阶段会进行方法连接）时会设置。方法连接时会调用如下方法：

```
// Called when the method_holder is getting
// linked. Setup entrypoints so the method
// is ready to be called from interpreter,
// compiler, and vtables.
void Method::link_method(
  methodHandle h_method, TRAPS) {
  // ...
  address entry = Interpreter::entry_for_method(h_method);
```

```
  // Sets both _i2i_entry and _from_interpreted_entry
  set_interpreter_entry(entry);

  // ...
}
```

首先调用Interpreter::entry_for_method()函数根据特定方法类型获取到方法的入口，得到入口entry后会调用set_interpreter_entry()函数将值保存到对应属性上。set_interpreter_entry()函数的实现非常简单，如下：

```
void set_interpreter_entry(address entry) {
  _i2i_entry = entry;
  _from_interpreted_entry = entry;
}
```

可以看到为_from_interpreted_entry属性设置了entry值。

下面看一下entry_for_method()函数的实现，如下：

```
static address entry_for_method(methodHandle m)  {
 return entry_for_kind(method_kind(m));
}
```

首先通过method_kind()函数拿到方法对应的类型，然后调用entry_for_kind()函数根据方法类型获取方法对应的入口entry_point。调用的entry_for_kind()函数的实现如下：

```
static address entry_for_kind(MethodKind k){
  return _entry_table[k];
}
```

这里直接返回了_entry_table数组中对应方法类型的entry_point地址。

这里涉及到Java方法的类型MethodKind，由于要通过entry_point进入Java世界，执行Java方法相关的逻辑，所以entry_point中一定会为对应的Java方法建立新的栈帧，但是不同方法的栈帧其实是有差别的，如Java普通方法、Java同步方法、有native关键字的Java方法等，所以就把所有的方法进行了归类，不同类型获取到不同的entry_point入口。到底有哪些类型，我们可以看一下MethodKind这个枚举类中定义出的枚举常量：

```
enum MethodKind {
    zerolocals, // 普通的方法
    // 普通的同步方法
    zerolocals_synchronized,
    native, // native方法
     // native同步方法
    native_synchronized,
    ...
}
```

当然还有其它一些类型，不过最主要的就是如上枚举类中定义出的4种类型方法。

为了能尽快找到某个Java方法对应的entry_point入口，把这种对应关系保存到了_entry_table中，所以entry_for_kind()函数才能快速的获取到方法对应的entry_point入口。 给数组中元素赋值专门有个方法：

```
void AbstractInterpreter::set_entry_for_kind(
  AbstractInterpreter::MethodKind kind,
  address entry) {
  _entry_table[kind] = entry;
}
```

那么何时会调用set_entry_for_kind()函数呢，答案就在TemplateInterpreterGenerator::generate_all()函数中，generate_all()函数会调用generate_method_entry()函数生成每种Java方法的entry_point，每生成一个对应方法类型的entry_point就保存到_entry_table中。

下面详细介绍一下generate_all()函数的实现逻辑，在HotSpot启动时就会调用这个函数生成各种Java方法的entry_point。调用栈如下：

```
start_thread()
JavaMain()
InitializeJVM()
JNI_CreateJavaVM()
Threads::create_vm()
init_globals()
interpreter_init()
TemplateInterpreter::initialize()
InterpreterGenerator::InterpreterGenerator()
TemplateInterpreterGenerator::generate_all()
```

调用的generate_all()函数将生成一系列HotSpot运行过程中所执行的一些公共代码的入口和所有字节码的InterpreterCodelet，一些非常重要的入口实现逻辑会在后面详细介绍，这里只看普通的、没有native关键字修饰的Java方法生成入口的逻辑。generate_all()函数中有如下实现：

```
#define method_entry(kind) \
{ \
CodeletMark cm(_masm, "method entry point (kind = " #kind ")"); \
Interpreter::_entry_table[Interpreter::kind]= generate_method_entry(Interpreter::kind); \
}

method_entry(zerolocals)
```

其中method_entry是一个宏，扩展后如上的method_entry(zerolocals)语句变为如下的形式：

```
Interpreter::_entry_table[Interpreter::zerolocals]
   = generate_method_entry(Interpreter::zerolocals);
```

_entry_table变量定义在AbstractInterpreter类中，如下：

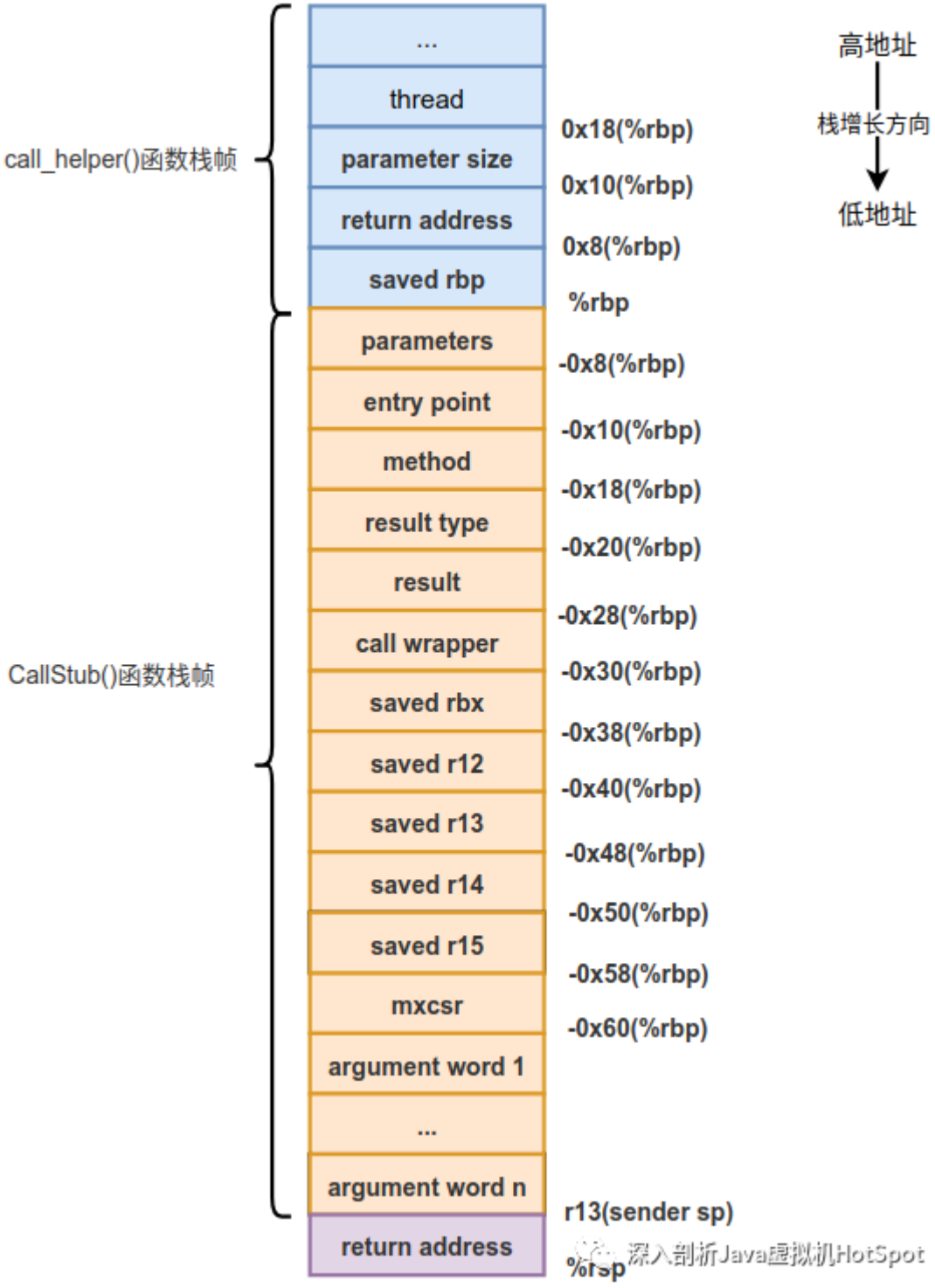
```
static address _entry_table[number_of_method_entries];
```

number_of_method_entries表示方法类型的总数，使用方法类型做为数组下标就可以获取对应的方法入口。调用generate_method_entry()函数为各种类型的方法生成对应的方法入口。generate_method_entry()函数的实现如下：

```
address AbstractInterpreterGenerator::generate_method_entry(
  AbstractInterpreter::MethodKind kind) {
  bool synchronized = false;
  address entry_point = NULL;
  InterpreterGenerator* ig_this = (InterpreterGenerator*)this;

  // 根据方法类型kind生成不同的入口
  switch (kind) {
  // 表示普通方法类型
  case Interpreter::zerolocals :
      break;
  // 表示普通的、同步方法类型
  case Interpreter::zerolocals_synchronized:
      synchronized = true;
      break;
  // ...
  }

  if (entry_point) {
     return entry_point;
  }

  return
    ig_this->generate_normal_entry(synchronized);
}
```

zerolocals表示正常的Java方法调用，包括Java程序的main()方法，对于zerolocals来说，会调用ig_this->generate_normal_entry()函数生成入口。generate_normal_entry()函数会为执行的方法生成堆栈，而堆栈由局部变量表（用来存储传入的参数和被调用方法的局部变量）、Java方法栈帧数据和操作数栈这三大部分组成，所以entry_point例程（其实就是一段机器指令片段，英文名为stub）会创建这3部分来辅助Java方法的执行。

我们还是回到开篇介绍的知识点，通过callq指令调用entry_point例程。此时的栈帧状态在 第4篇-JVM终于开始调用Java主类的main()方法啦 中介绍过，为了大家阅读的方便，这里再次给出：



注意，在执行callq指令时，会将函数的返回地址存储到栈顶，所以上图中会压入return address一项。

CallStub()函数在通过callq指令调用generate_normal_entry()函数生成的entry_point时，有几个寄存器中存储着重要的值，如下：

```
rbx -> Method*
r13 -> sender sp
rsi -> entry point
```

下面就是分析generate_normal_entry()函数的实现逻辑了，这是调用Java方法的最重要的部分。函数的重要实现逻辑如下：

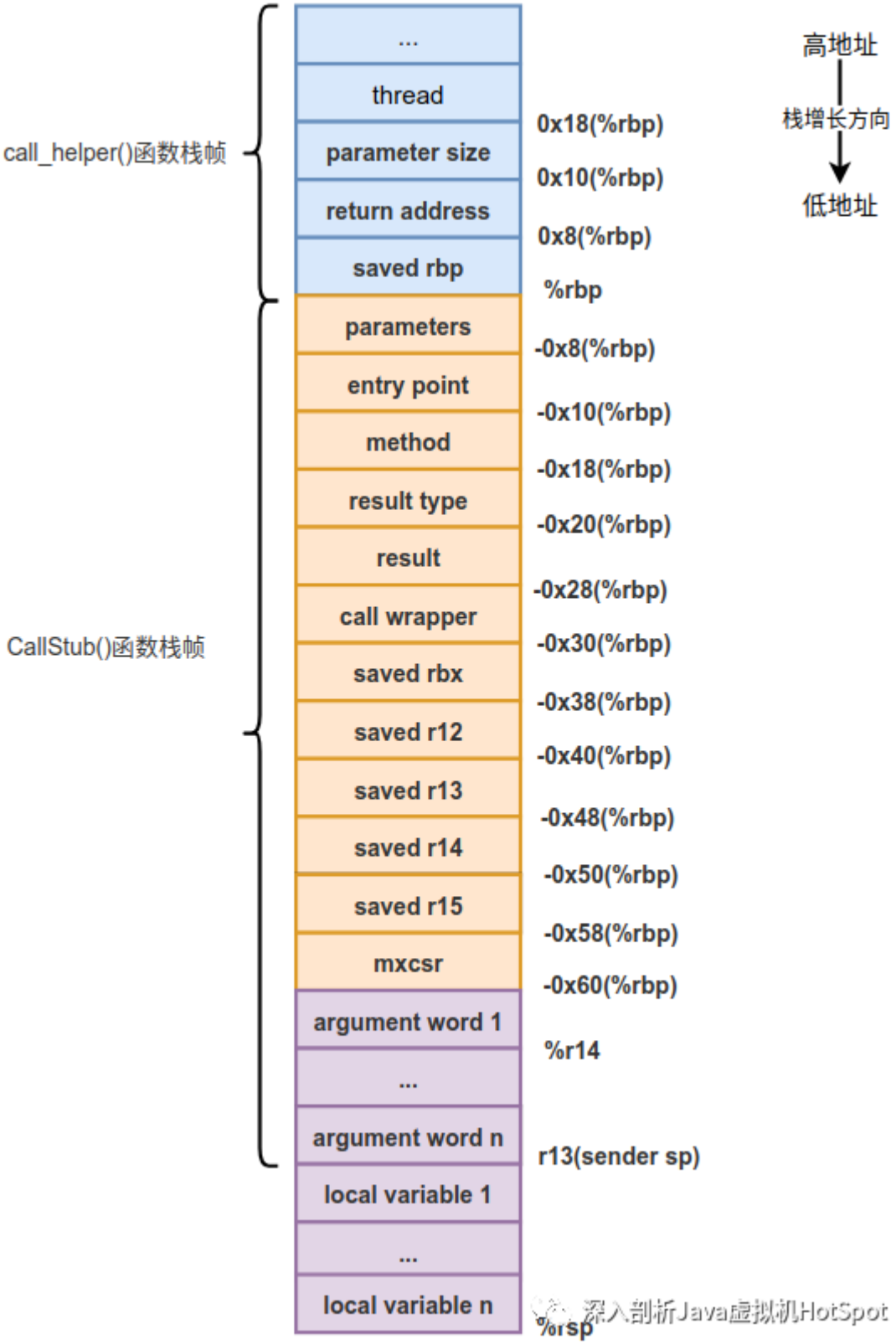
```
address InterpreterGenerator::generate_normal_entry(
    bool synchronized) {

  // ...

  // entry_point函数的代码入口地址
  address entry_point = __ pc();

  // 当前rbx中存储的是指向Method的指针，
  // 通过Method*找到ConstMethod*
  const Address constMethod(rbx,Method::const_offset());

  // 通过Method*找到AccessFlags
  const Address access_flags(rbx,
        Method::access_flags_offset());

  // 通过ConstMethod*得到parameter的大小
  const Address size_of_parameters(
        rdx,ConstMethod::size_of_parameters_offset());

  // 通过ConstMethod*得到local变量的大小
  const Address size_of_locals(rdx,
         ConstMethod::size_of_locals_offset());

  // 上面已经说明了获取各种方法元数据的计算方式，
  // 但并没有执行计算，下面会生成对应的汇编来执行计算
  // 计算ConstMethod*，保存在rdx里面
  __ movptr(rdx, constMethod);

  // 计算parameter大小，保存在rcx里面
  __ load_unsigned_short(rcx, size_of_parameters);

  // rbx：保存基址；rcx：保存循环变量；
  // rdx：保存目标地址；rax：保存返回地址（下面用到）
  // 此时的各个寄存器中的值如下：
  // rbx: Method*
```

```
// rcx: size of parameters
// r13:
// sender_sp (could differ from sp+wordSize
// if we were called via c2i ) 即调用者的栈顶地址
// 计算local变量的大小，保存到rdx
__ load_unsigned_short(rdx, size_of_locals);

// 由于局部变量表用来存储传入的参数和被调
// 用方法的局部变量，所以rdx减去
// rcx后就是被调用方法的局部变量可使用的大小
__ subl(rdx, rcx);

// ...

// 返回地址是在CallStub中保存的，如果不弹
// 出堆栈到rax，中间会有个
// return address使的局部变量表不是连续的，
// 这会导致其中的局部变量计算方式不一致，所以
// 暂时将返回地址存储到rax中
__ pop(rax);

// 计算第1个参数的地址：
// 当前栈顶地址 + 变量大小 * 8 - 一个字大小
// 注意，因为地址保存在低地址上，而堆栈是向低
// 地址扩展的，所以只需加n-1个
// 变量大小就可以得到第1个参数的地址
__ lea(r14, Address(rsp,
       rcx, Address::times_8, -wordSize));

// 把函数的局部变量设置为0,也就是做初始化，
// 防止之前遗留下的值影响
// rdx：被调用方法的局部变量可使用的大小
{
  Label exit, loop;
  __ testl(rdx, rdx);
  // 如果rdx<=0，不做任何操作
  __ jcc(Assembler::lessEqual, exit);
  __ bind(loop);
  // 初始化局部变量
  __ push((int) NULL_WORD);
  __ decrementl(rdx);
  __ jcc(Assembler::greater, loop);
```

```
    __ bind(exit);
  }

  // 生成固定桢
  generate_fixed_frame(false);

  // ... 省略统计及栈溢出等逻辑，后面会详细介绍
  // 如果是同步方法时，还需要执行lock_method()
  // 函数，所以会影响到栈帧布局
  if (synchronized) {
    lock_method();
  }

  // 跳转到目标Java方法的第一条字节码指令，
  // 并执行其对应的机器指令
   __ dispatch_next(vtos);

  // ... 省略统计相关逻辑，后面会详细介绍
  return entry_point;
}
```

这个函数的实现看起来比较多，但其实逻辑实现比较简单，就是根据被调用方法的实际情况创建出对应的局部变量表，然后就是2个非常重要的函数generate_fixed_frame()和dispatch_next()函数了，这2个函数我们后面再详细介绍。

在调用generate_fixed_frame()函数之前，栈的状态变为了下图所示的状态。



与前一个图对比一下，可以看到多了一些local variable 1 ... local variable n等slot，这些slot与argument word 1 ... argument word n共同构成了被调用的Java方法的局部变量表，也就是图中紫色的部分。其实local variable 1 ... local variable n等slot属于被调用的Java方法栈帧的一部分，而argument word 1 ... argument word n却属于CallStub()函数栈帧的一部分，这2部分共同构成局部变量表，专业术语叫栈帧重叠。

另外还能看出来，%r14指向了局部变量表的第1个参数，而CallStub()函数的return address被保存到了%rax中，另外%rbx中依然存储着Method*。这些寄存器中保存的值将在调用generate_fixed_frame()函数时用到，所以我们需要在这里强调一下。

收录于合集 #java 9

上一篇
第5篇-调用Java方法后弹出栈帧及处理返回结果

下一篇
第7篇-为Java方法创建栈帧

People who liked this content also liked

如何随心所欲调试HotSpot源代码?
深入剖析Java虚拟机HotSpot

猜：宅男看片神器
放毒