第5篇-调用Java方法后弹出栈帧及处理返回结果

Original 鸠摩 深入剖析Java虚拟机HotSpot 2021-12-07 16:21

收录于合集

#java 9 #运行时 9 #hotspot 10 #虚拟机 10

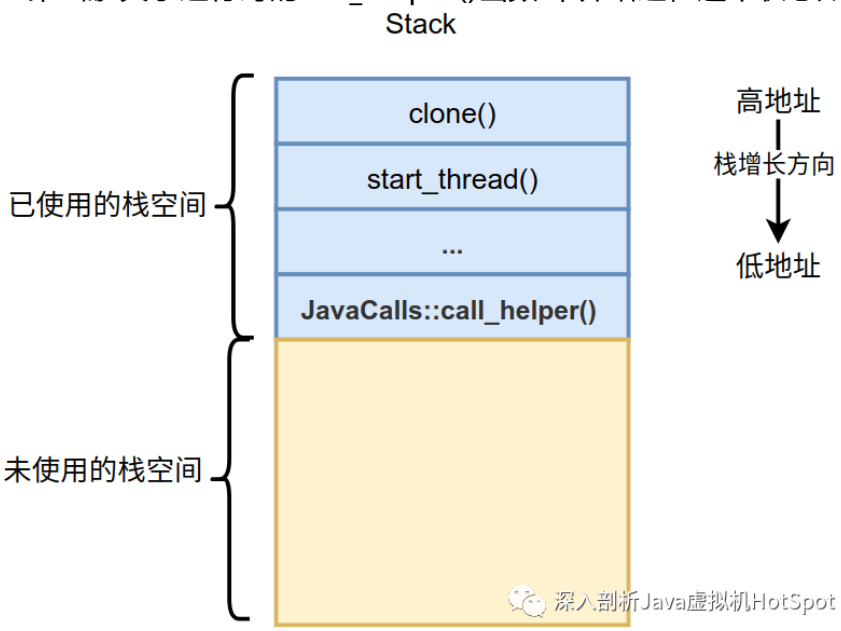
深入剖析Java虚拟机HotSpot

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

在前一篇 第4篇-JVM终于开始调用Java主类的main()方法啦 介绍了通过callq调用entry point，不过我们并没有看完generate_call_stub()函数的实现。接下来在generate_call_stub()函数中会处理调用Java方法后的返回值，同时还需要执行退栈操作，也就是将栈恢复到调用Java方法之前的状态。调用之前是什么状态呢？在 第2篇-关于运行时的call_helper()函数 中介绍过，这个状态如下图所示。



generate_call_stub()函数接下来的代码实现如下：

```
// 保存方法调用结果依赖于结果类型，只要
// 不是T_OBJECT, T_LONG, T_FLOAT or T_DOUBLE,
// 都当做T_INT处理将result地址的值拷贝到c_rarg0中,
// 也就是将方法调用的结果保存在rdi寄存器中,
// 注意result为函数返回值的地址
__ movptr(c_rarg0, result);
```

```
Label is_long, is_float, is_double, exit;

// 将result_type地址的值拷贝到c_rarg1中,
// 也就是将方法调用的结果返回的类型保存在esi寄存器中
__ movl(c_rarg1, result_type);

// 根据结果类型的不同跳转到不同的处理分支
__ cmpl(c_rarg1, T_OBJECT);
__ jcc(Assembler::equal, is_long);
__ cmpl(c_rarg1, T_LONG);
__ jcc(Assembler::equal, is_long);
__ cmpl(c_rarg1, T_FLOAT);
__ jcc(Assembler::equal, is_float);
__ cmpl(c_rarg1, T_DOUBLE);
__ jcc(Assembler::equal, is_double);

// 当逻辑执行到这里时，处理的就是T_INT类型,
// 将rax中的值写入c_rarg0保存的地址指向的内存中
// 调用函数后如果返回值是int类型，则根据调用约定
// 会存储在eax中
__ movl(Address(c_rarg0, 0), rax);

__ BIND(exit);

// 将rsp_after_call中保存的有效地址拷贝到rsp中,
// 即将rsp往高地址方向移动了,
// 原来的方法调用实参argument 1、...、argument n,
// 相当于从栈中弹出，所以下面语句执行的是退栈操作
// lea指令将地址加载到寄存器中
__ lea(rsp, rsp_after_call);
```

这里我们要关注result和result_type，result在调用call_helper()函数时就会传递，也就是会指示call_helper()函数将调用Java方法后的返回值存储在哪里。对于类型为JavaValue的result来说，其实在调用之前就已经设置了返回类型，所以如上的result_type变量只需要从JavaValue中获取结果类型即可。例如，调用Java主类的main()方法时，在jni_CallStaticVoidMethod()函数和jni_invoke_static()函数中会设置返回类型为T_VOID，也就是main()方法返回void。

生成的汇编代码如下：

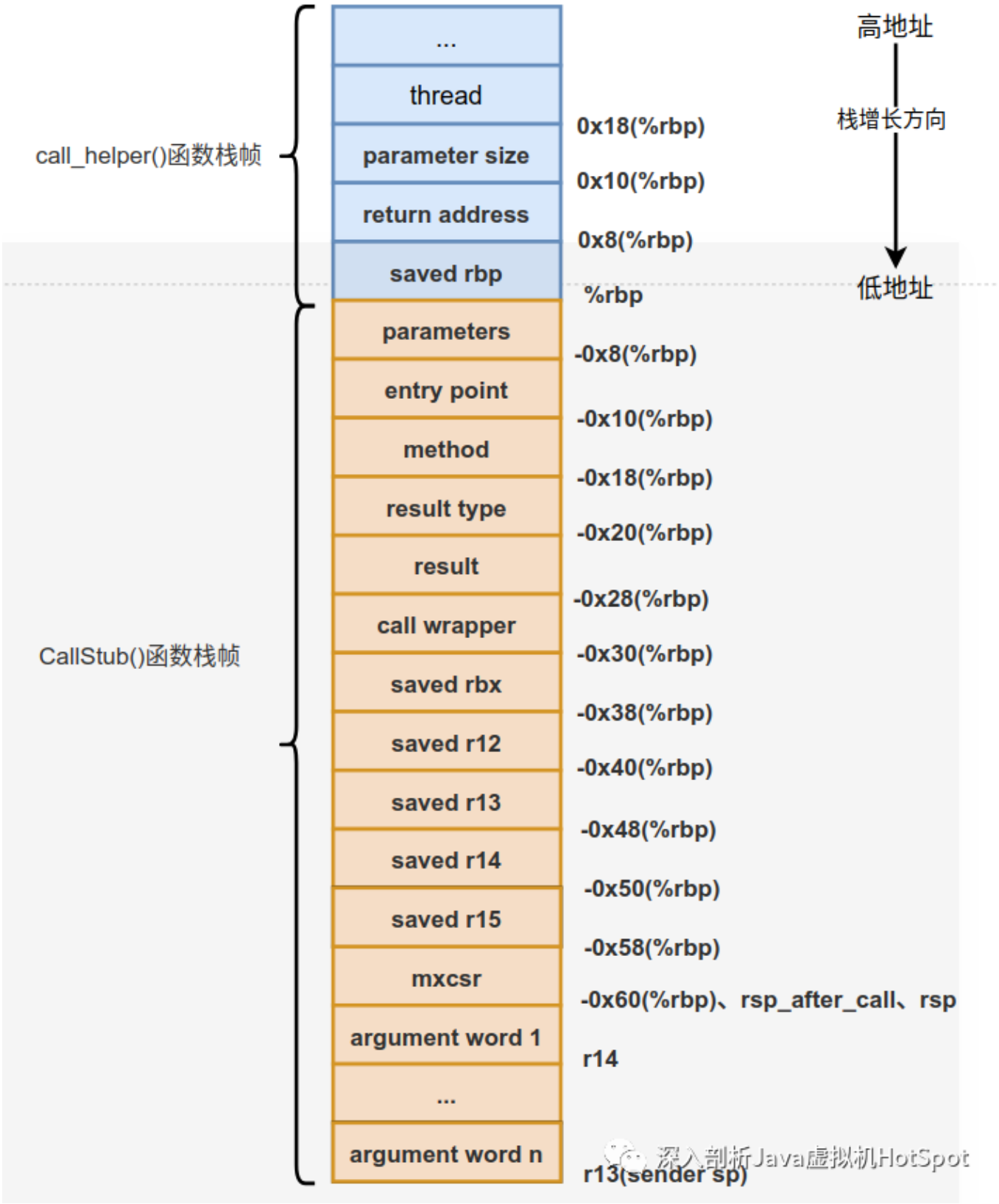
```
// 栈中的-0x28位置保存result
mov -0x28(%rbp),%rdi
// 栈中的-0x20位置保存result type
```

```
mov -0x20(%rbp),%esi
// 是否为T_OBJECT类型
cmp $0xc,%esi
je 0x00007fdf450007f6
// 是否为T_LONG类型
cmp $0xb,%esi
je 0x00007fdf450007f6
// 是否为T_FLOAT类型
cmp $0x6,%esi
je 0x00007fdf450007fb
// 是否为T_DOUBLE类型
cmp $0x7,%esi
je 0x00007fdf45000801
// 如果是T_INT类型，直接将返
// 回结果%eax写到栈中-0x28(%rbp)的位置
mov %eax,(%rdi)

// -- exit --

// 将rsp_after_call的有效地址拷到rsp中
lea -0x60(%rbp),%rsp
```

为了让大家看清楚，我贴一下在调用Java方法之前的栈帧状态，如下：



由图可看到-0x60(%rbp)地址指向的位置，恰好不包括调用Java方法时压入的实际参数argument word 1 ... argument word n。所以现在rbp和rsp就是图中指向的位置了。

接下来恢复之前保存的caller-save寄存器，这也是调用约定的一部分，如下：

```
__ movptr(r15, r15_save);
__ movptr(r14, r14_save);
__ movptr(r13, r13_save);
__ movptr(r12, r12_save);
__ movptr(rbx, rbx_save);
__ ldmxcsr(mxcsr_save);
```

生成的汇编代码如下：

```
mov      -0x58(%rbp),%r15
mov      -0x50(%rbp),%r14
mov      -0x48(%rbp),%r13
mov      -0x40(%rbp),%r12
mov      -0x38(%rbp),%rbx
ldmxcsr  -0x60(%rbp)
```

在弹出了为调用Java方法保存的实际参数及恢复caller-save寄存器后，继续执行退栈操作，实现如下：

```
// restore rsp
__ addptr(rsp, -rsp_after_call_off * wordSize);


// return
__ pop(rbp);
__ ret(0);
```

生成的汇编代码如下：

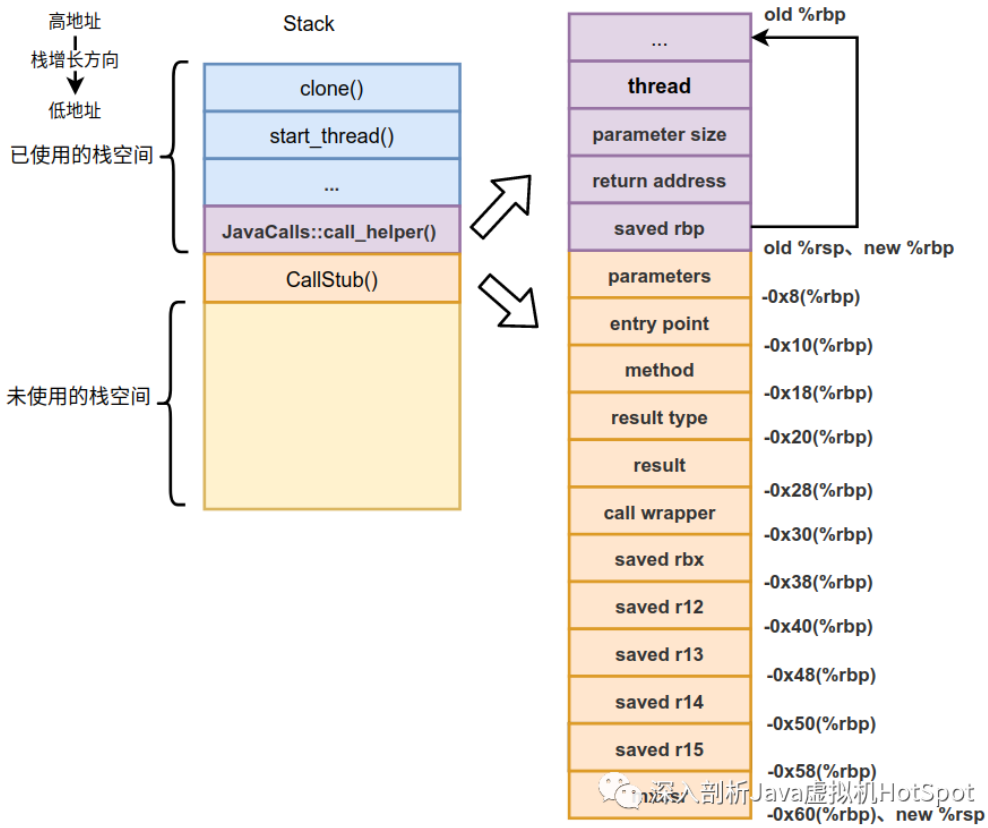
```
// %rsp加上0x60，也就是执行退栈操作，
// 也就相当于弹出了callee_save寄存器
// 和压栈的那6个参数
add    $0x60,%rsp
pop %rbp
// 方法返回，指令中的q表示64位操作数，
// 就是指的栈中存储的return address是64位的
retq
```

记得在之前 第3篇-CallStub新栈帧的创建时，通过如下的汇编完成了新栈帧的创建：

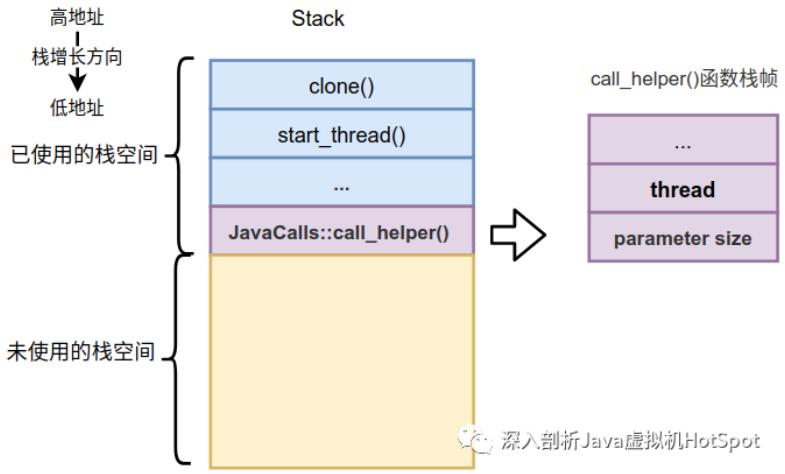
```
push   %rbp
mov %rsp,%rbp
sub $0x60,%rsp
```

现在要退出这个栈帧时要在%rsp指向的地址加上$0x60，同时恢复%rbp的指向。然后就是跳转到return address指向的指令继续执行了。

为了方便大家查看，我再次给出了之前使用到的图片，这个图是退栈之前的图片：



退栈之后如下图所示。



至于paramter size与thread则由JavaCalls::call_hlper()函数负责释放，这是C/C++调用约定的一部分。所以如果不看这2个参数，我们已经完全回到了本篇给出的第一张图表示的栈的样子。

上面这些图片大家应该不陌生才对，我们在一步步创建栈帧时都给出过，现在怎么创建的就会怎么退出。

之前介绍过，当Java方法返回int类型时（如果返回char、boolean、short等类型时统一转换为int类型），根据Java方法调用约定，这个返回的int值会存储到%rax中；如果返回对象，那么%rax中存储的就是这个对象的地址，那后面到底怎么区分是地址还是int值呢？答案是通过返回类型区分即可；如果返回非int，非对象类型的值呢？我们继续看generate_call_stub()函数的实现逻辑：

```
// handle return types different from T_INT

__ BIND(is_long);
__ movq(Address(c_rarg0, 0), rax);
__ jmp(exit);

__ BIND(is_float);
__ movflt(Address(c_rarg0, 0), xmm0);
__ jmp(exit);

__ BIND(is_double);
__ movdbl(Address(c_rarg0, 0), xmm0);
__ jmp(exit);
```

对应的汇编代码如下：

```
// -- is_long --
mov %rax,(%rdi)
jmp 0x00007fdf450007d4

// -- is_float --
vmovss %xmm0,(%rdi)
jmp 0x00007fdf450007d4

// -- is_double --
vmovsd %xmm0,(%rdi)
jmp 0x00007fdf450007d4
```

当返回long类型时也存储到%rax中，因为Java的long类型是64位，我们分析的代码也是x86下64位的实现，所以%rax寄存器也是64位，能够容纳64位数；当返回为float或double时，存储到%xmm0中。

统合这一篇和前几篇文章，我们应该学习到C/C++的调用约定以及Java方法在解释执行下的调用约定（包括如何传递参数，如何接收返回值等），如果大家不明白，多读几遍文章就会有一个清晰的认识。