第3篇-CallStub新栈帧的创建

Original 鸠摩 深入剖析Java虚拟机HotSpot 2021-12-05 17:10

收录于合集

#java 9 #运行时 9 #hotspot 10 #main() 3 #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...

84篇原创内容

公众号

在第2篇中我们介绍了在call_helper()函数中通过函数指针的方式调用了一个函数，如下：

```
StubRoutines::call_stub()(
 (address)&link,
 result_val_address,
 result_type,
 method(),
 entry_point,
 args->parameters(),
 args->size_of_parameters(),
 CHECK
);
```

其中调用StubRoutines::call_stub()函数会返回一个函数指针，查清楚这个函数指针指向的函数的实现是我们这一篇的重点。 调用的call_stub()函数的实现如下：

```
static CallStub call_stub() {
 return CAST_TO_FN_PTR(
      CallStub,
     _call_stub_entry
 );
}
```

call_stub()函数返回一个函数指针，指向依赖于操作系统和cpu架构的特定的方法，原因很简单，要执行native代码，得看看是什么cpu架构以便确定寄存器，看看什么os以便确定ABI。

其中CAST_TO_FN_PTR是宏，具体定义如下：

```
#define
 CAST_TO_FN_PTR(func_type, value)
    ((func_type)(castable_address(value)))
```

对call_stub()函数进行宏替换和展开后会变为如下的形式：

```
static CallStub call_stub(){
    return (CallStub)(
      castable_address(_call_stub_entry)
    );
}
```

CallStub的定义如下：

```
typedef void (*CallStub)(
// 连接器
 address link,
 // 函数返回值地址
 intptr_t* result,
 //函数返回类型
 BasicType result_type,
 // JVM内部所表示的Java方法对象
 Method* method,
 // JVM调用Java方法的例程入口。
 // JVM内部的每一段例程都是在JVM启动过
 // 程中预先生成好的一段机器指令。要调用Java方法，
 // 必须经过本例程，即需要先执行这段机器指令，
 // 然后才能跳转到，Java方法字节码所对应的机器
 // 指令去执行
 address entry_point,
 intptr_t* parameters,
 int    size_of_parameters,
 TRAPS
);
```

如上定义了一种函数指针类型，指向的函数声明了8个形式参数。

在call_stub()函数中调用的castable_address()函数定义在globalDefinitions.hpp文件中，具体实现如下：

```
inline address_word castable_address(address x)  {
    return address_word(x) ;
}
```

address_word是一定自定义的类型，在globalDefinitions.hpp文件中的定义如下：

```
typedef    uintptr_t      address_word;
```

其中uintptr_t也是一种自定义的类型，在Linux内核的操作系统下使用globalDefinitions_gcc.hpp文件中的定义，具体定义如下：

```
typedef  unsigned int  uintptr_t;
```

这样call_stub()函数其实等同于如下的实现形式：

```
static CallStub call_stub(){
    return (CallStub)(
        unsigned int(_call_stub_entry) );
}
```

将_call_stub_entry强制转换为unsigned int类型，然后以强制转换为CallStub类型。CallStub是一个函数指针，所以_call_stub_entry应该也是一个函数指针，而不应该是一个普通的无符号整数。

在call_stub()函数中，_call_stub_entry的定义如下：

```
address StubRoutines::_call_stub_entry = NULL;
```

_call_stub_entry 的初始化在在/src/cpu/x86/vm/stubGenerator_x86_64.cpp 文件下的generate_initial()函数，调用链如下：

```
JavaMain()
InitializeJVM()
JNI_CreateJavaVM()
Threads::create_vm()
init_globals()
stubRoutines_init1()
StubRoutines::initialize1()
StubGenerator_generate()
StubGenerator::StubGenerator()
StubGenerator::generate_initial()
```

其中的StubGenerator类定义在src/cpu/x86/vm目录下的stubGenerator_x86_64.cpp文件中，这个文件中的generate_initial()方法会初始化call_stub_entry变量，如下：

```
StubRoutines::_call_stub_entry = generate_call_stub(
 StubRoutines::_call_stub_return_address
);
```

现在我们终于找到了函数指针指向的函数的实现逻辑，这个逻辑是通过调用generate_call_stub()函数来实现的。

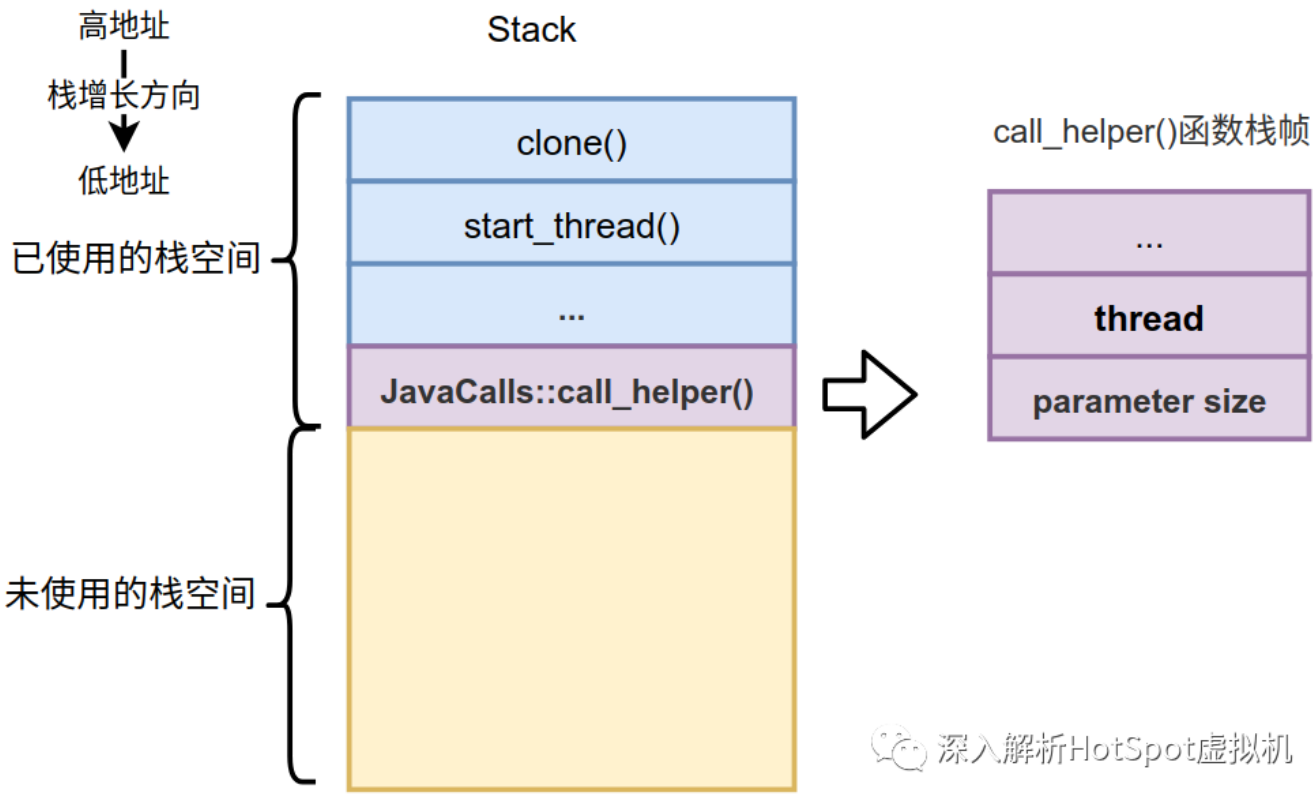
不过经过查看后我们发现这个函数指针指向的并不是一个C++函数，而是一个机器指令片段，我们可以将其看为C++函数经过C++编译器编译后生成的指令片段即可。在generate_call_stub()函数中有如下调用语句：

```
__ enter();
__ subptr(rsp, -rsp_after_call_off * wordSize);
```

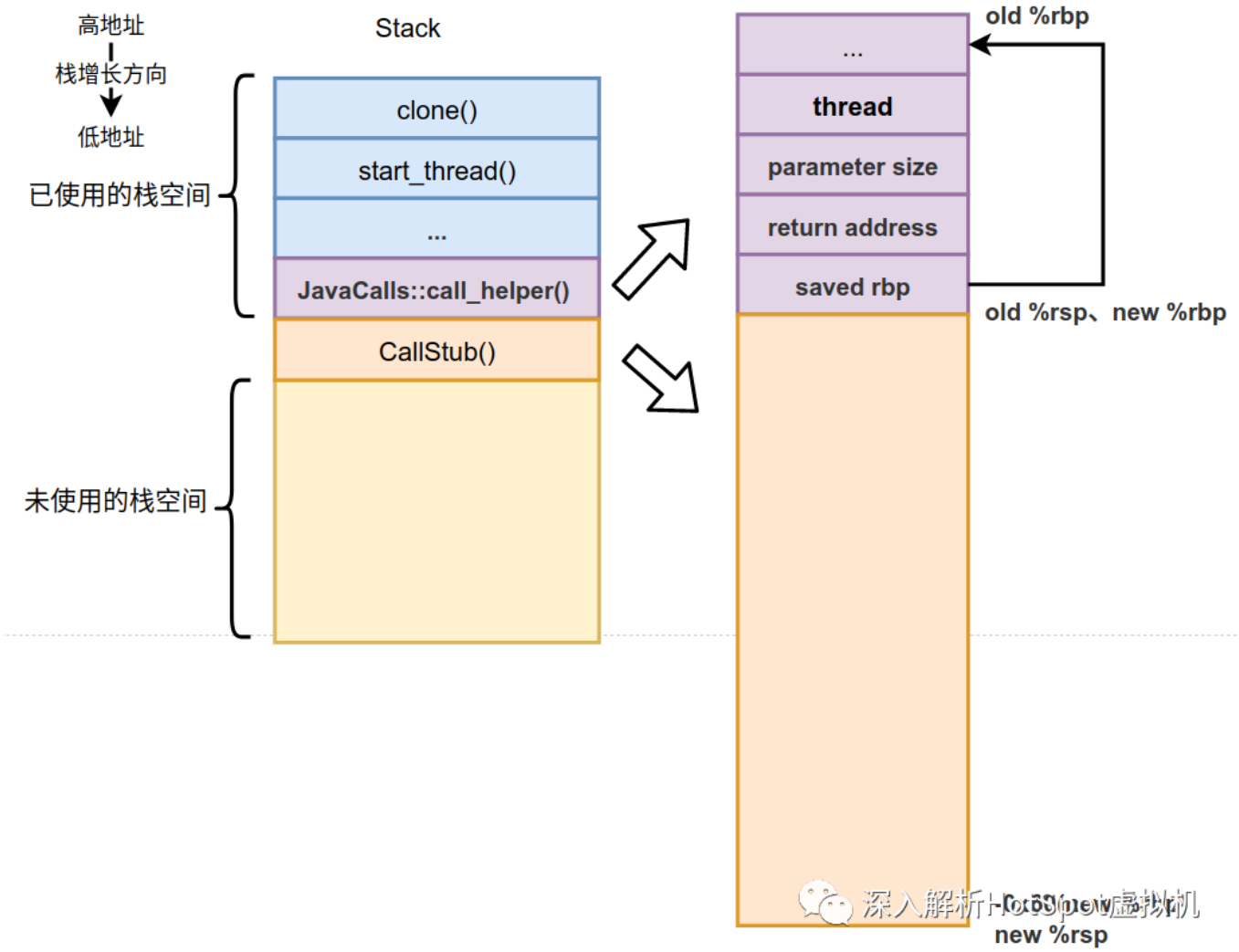
这两段代码直接生成机器指令，不过为了查看机器指令，我们借助了HSDB工具将其反编译为可读性更强的汇编指令。如下：

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

这3条汇编是非常典型的开辟新栈帧的指令。之前我们介绍过在通过函数指针进行调用之前的栈状态，如下：



那么经过运行如上3条汇编后这个栈状态就变为了如下的状态：



我们需要关注的就是old %rbp和old %rsp在没有运行开辟新栈帧（CallStub()栈帧）时的指向，以及开辟新栈帧（CallStub()栈帧）时的new %rbp和new %rsp的指向。另外还要注意saved rbp保存的就是old %rbp，这个值对于栈展开非常重要，因为能通过它不断向上遍历，最终能找到所有的栈帧。

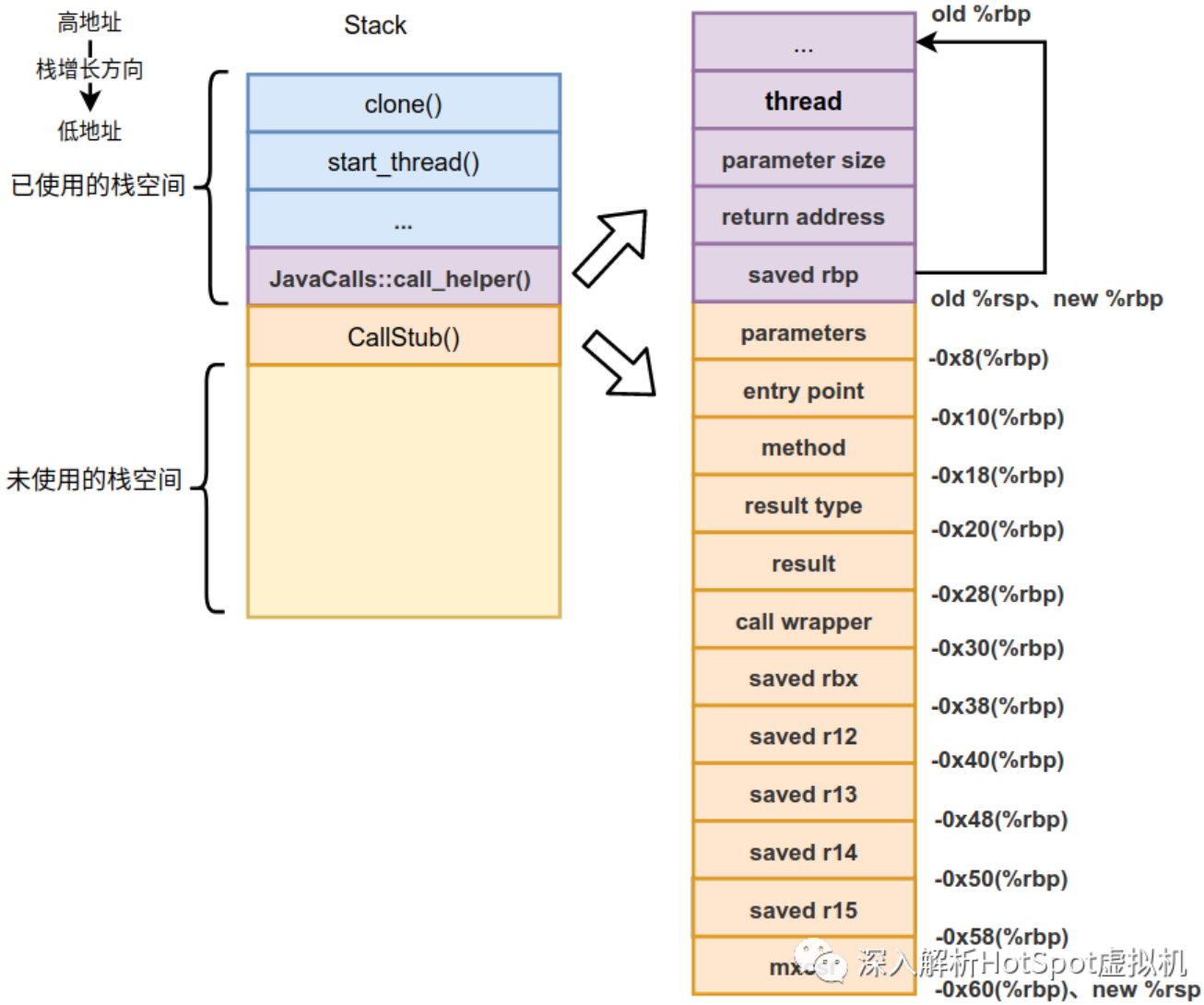
下面接着看generate_call_stub()函数的实现，如下：

```
address generate_call_stub(address& return_address) {
    ...
    address start = __ pc();


    const Address rsp_after_call(rbp, rsp_after_call_off * wordSize);

    const Address call_wrapper  (rbp, call_wrapper_off * wordSize);
    const Address result        (rbp, result_off * wordSize);
    const Address result_type   (rbp, result_type_off * wordSize);
    const Address method        (rbp, method_off * wordSize);
    const Address entry_point   (rbp, entry_point_off * wordSize);
    const Address parameters    (rbp, parameters_off * wordSize);
    const Address parameter_size(rbp, parameter_size_off * wordSize);
```

```
const Address thread        (rbp, thread_off * wordSize);

const Address r15_save(rbp, r15_off * wordSize);
const Address r14_save(rbp, r14_off * wordSize);
const Address r13_save(rbp, r13_off * wordSize);
const Address r12_save(rbp, r12_off * wordSize);
const Address rbx_save(rbp, rbx_off * wordSize);

// 开辟新的栈帧
__ enter();
__ subptr(rsp, -rsp_after_call_off * wordSize);

// save register parameters
__ movptr(parameters, c_rarg5); // parameters
__ movptr(entry_point, c_rarg4); // entry_point



__ movptr(method, c_rarg3); // method
__ movl(result_type, c_rarg2); // result type
__ movptr(result, c_rarg1); // result
__ movptr(call_wrapper, c_rarg0); // call wrapper

// save regs belonging to calling function
__ movptr(rbx_save, rbx);
__ movptr(r12_save, r12);
__ movptr(r13_save, r13);
__ movptr(r14_save, r14);
__ movptr(r15_save, r15);

const Address mxcsr_save(rbp, mxcsr_off * wordSize);
{
  Label skip_ldmx;
  __ stmxcsr(mxcsr_save);
  __ movl(rax, mxcsr_save);
  __ andl(rax, MXCSR_MASK); // Only check control and mask bits
  ExternalAddress mxcsr_std(StubRoutines::addr_mxcsr_std());
  __ cmp32(rax, mxcsr_std);
  __ jcc(Assembler::equal, skip_ldmx);
  __ ldmxcsr(mxcsr_std);
  __ bind(skip_ldmx);
}
```

```
    // ... 省略了接下来的操作
}
```

其中开辟新栈帧的逻辑我们已经介绍过，下面就是将call_helper()传递的6个在寄存器中的参数存储到CallStub()栈帧中了，除了存储这几个参数外，还需要存储其它寄存器中的值，因为函数接下来要做的操作是为Java方法准备参数并调用Java方法，我们并不知道Java方法会不会破坏这些寄存器中的值，所以要保存下来，等调用完成后进行恢复。加载完成这些参数后如下图所示。



下一篇我们继续介绍下generate_call_stub()函数中其余的实现。