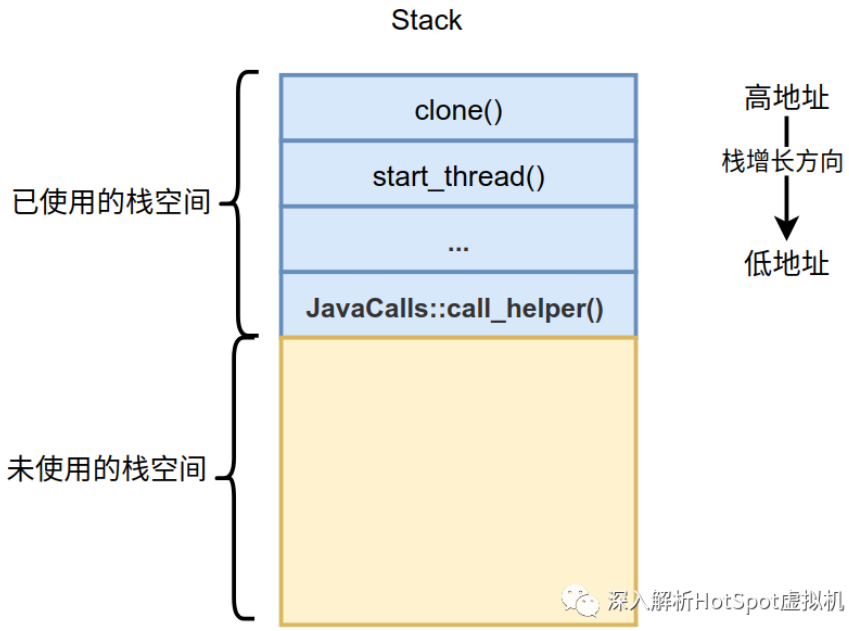
第2篇-关于运行时的call_helper()函数

Original 鸠摩 深入剖析Java虚拟机HotSpot 2021-12-04 16:31

在前一篇中介绍了call_static()、call_virtual()等函数的作用，这些函数会调用JavaCalls::call()函数。我们看Java类中main()方法的调用，调用栈如下：

```
clone() at clone.S
start_thread() at pthread_create.c
JavaMain() at java.c
jni_CallStaticVoidMethod() at jni.cpp
jni_invoke_static() at jni.cpp
JavaCalls::call() at javaCalls.cpp
os::os_exception_wrapper() at os_linux.cpp
JavaCalls::call_helper() at javaCalls.cpp
```

这是Linux上的调用栈，通过JavaCalls::call_helper()函数来执行main()方法。栈的起始函数为clone()，这个函数会为每个进程（Linux进程对应着Java线程）创建单独的栈空间，这个栈空间如下图所示。



在Linux操作系统上，栈的地址向低地址延伸，所以未使用的栈空间在已使用的栈空间之下。图中的每个蓝色小格表示对应方法的栈帧，而栈就是由一个一个的栈帧组成。native方法的栈帧、Java解释栈帧

和Java编译栈帧都会在黄色区域中分配，所以说他们寄生在宿主栈中，这些不同的栈帧都紧密的挨在一起，所以并不会产生什么空间碎片这类的问题，而且这样的布局非常有利于进行栈的遍历。上面给出的调用栈就是通过遍历一个一个栈帧得到的，遍历过程也是栈展开的过程。后续对于异常的处理、运行jstack打印线程堆栈、GC查找根引用等都会对栈进行展开操作，所以栈展开是后面必须要介绍的。

下面我们继续看JavaCalls::call_helper()函数，这个函数中有个非常重要的调用，如下：

```
{
    JavaCallWrapper link(method, receiver, result, CHECK);
    {
      HandleMark hm(thread);
      StubRoutines::call_stub()(
         (address)&link,
         result_val_address,
         result_type,
         method(),
         entry_point,
         args->parameters(),
         args->size_of_parameters(),
         CHECK
      );


      result = link.result();
      // Preserve oop return value across possible gc points
      if (oop_result_flag) {
        thread->set_vm_result((oop) result->get_jobject());
      }
    }
}
```

调用StubRoutines::call_stub()函数返回一个函数指针，然后通过函数指针来调用函数指针指向的函数。通过函数指针调用和通过函数名调用的方式一样，这里我们需要清楚的是，调用的目标函数仍然是C/C++函数，所以由C/C++函数调用另外一个C/C++函数时，要遵守调用约定。这个调用约定会规定怎么给被调用函数（Callee）传递参数，以及被调用函数的返回值将存储在什么地方。

下面我们就来简单说说Linux X86架构下的C/C++函数调用约定，在这个约定下，以下寄存器用于传递参数：

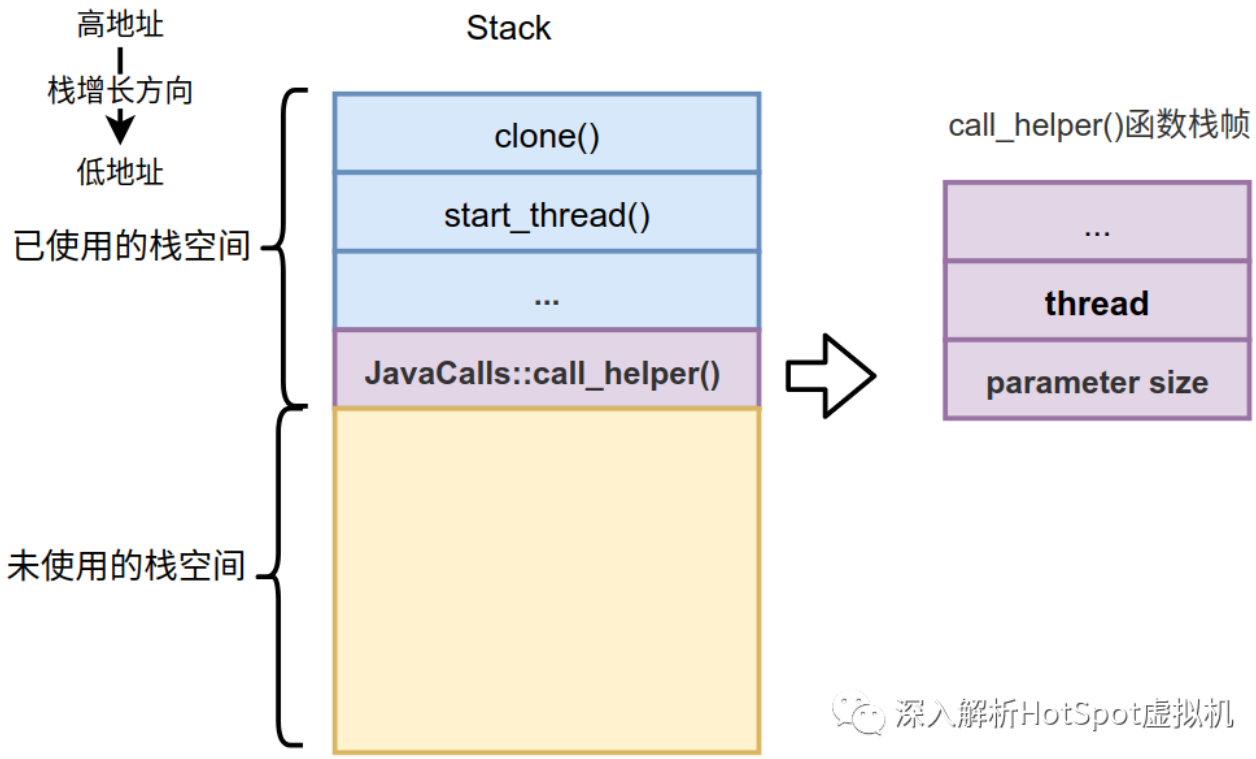
- 第1个参数：rdi　c_rarg0
- 第2个参数：rsi　c_rarg1
- 第3个参数：rdx　c_rarg2
- 第4个参数：rcx　c_rarg3
- 第5个参数：r8　c_rarg4

- 第6个参数：r9　c_rarg5

在函数调用时，6个及小于6个用如下寄存器来传递，在HotSpot中通过更易理解的别名c_rarg*来使用对应的寄存器。如果参数超过六个，那么程序将会用调用栈来传递那些额外的参数。

数一下我们通过函数指针调用时传递了几个参数？8个，那么后面的2个就需要通过调用函数（Caller）的栈来传递，这两个参数就是args->size_of_parameters()和CHECK（这是个宏，扩展后就是传递线程对象）。

所以我们的调用栈在调用函数指针指向的函数时，变为了如下状态：



右边是具体的call_helper()栈帧中的内容，我们把thread和parameter size压入了调用栈中，其实在调目标函数的过程还会开辟新的栈帧并在parameter size后压入返回地址和调用栈的栈底，下一篇我们再详细介绍。先来介绍下JavaCalls::call_helper()函数的实现，我们分3部分依次介绍。

1、检查目标方法是否"首次执行前就必须被编译”，是的话调用JIT编译器去编译目标方法；

代码实现如下：

```
void JavaCalls::call_helper(
 JavaValue* result,
 methodHandle* m,
 JavaCallArguments* args,
 TRAPS
) {
  methodHandle method = *m;
  JavaThread* thread = (JavaThread*)THREAD;
  ...
```

```
  assert(!thread->is_Compiler_thread(),
             "cannot compile from the compiler");
  if (CompilationPolicy::must_be_compiled(method)) {
    CompileBroker::compile_method(
            method,
            InvocationEntryBci,
            CompilationPolicy::policy()->initial_compile_level(),
            methodHandle(),
            0,
            "must_be_compiled",
            CHECK);
  }
  ...
}
```

对于main()方法来说，如果配置了-Xint选项，则是以解释模式执行的，所以并不会走上面的compile_method()函数的逻辑。后续我们要研究编译执行时，可以强制要求进行编译执行，然后查看执行过程。

2、获取目标方法的解释模式入口from_interpreted_entry，也就是entry_point的值。获取的entry_point就是为Java方法调用准备栈桢，并把代码调用指针指向method的第一个字节码的内存地址。entry_point相当于是method的封装，不同的method类型有不同的entry_point。

接着看call_helper()函数的代码实现，如下：

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

调用method的from_interpreted_entry()函数获取Method实例中_from_interpreted_entry属性的值，这个值到底在哪里设置的呢？我们后面会详细介绍。

3、调用call_stub()函数，需要传递8个参数。这个代码在前面给出过，这里不再给出。下面我们详细介绍一下这几个参数，如下：

1. link 此变量的类型为JavaCallWrapper，这个变量对于栈展开过程非常重要，后面会详细介绍；
2. result_val_address 函数返回值地址；
3. result_type 函数返回类型；
4. method() 当前要执行的方法。通过此参数可以获取到Java方法所有的元数据信息，包括最重要的字节码信息，这样就可以根据字节码信息解释执行这个方法了；
5. entry_point HotSpot每次在调用Java函数时，必然会调用CallStub函数指针，这个函数指针的值取自_call_stub_entry，HotSpot通过_call_stub_entry指向被调用函数地址。在调用函数之前，必须要先经过entry_point，HotSpot实际是通过entry_point从method()对象上拿到Java方法对应的第1个字节码命令，这也是整个函数的调用入口；

6. args->parameters() 描述Java函数的入参信息；

7. args->size_of_parameters() 描述Java函数的入参数量；

8. CHECK 当前线程对象。

这里最重要的就是entry_point了，这也是下一篇要介绍的内容。