第7篇-为Java方法创建栈帧

Original 鸠摩 深入剖析Java虚拟机HotSpot 2021-12-09 22:41

收录于合集

#java 9 #运行时 9 #hotspot 10 #虚拟机 10

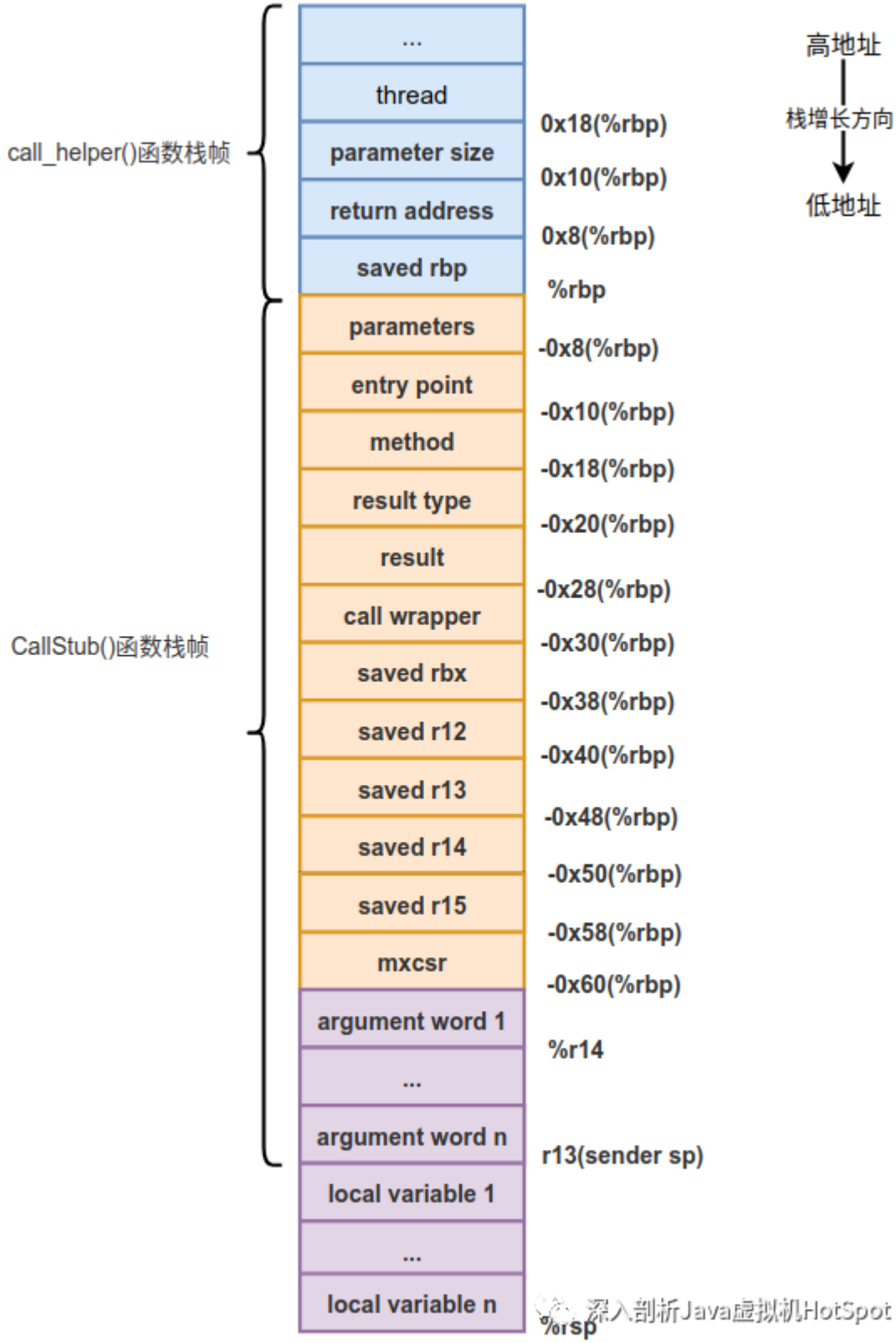
深入剖析Java虚拟机HotSpot

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6篇-Java方法新栈帧的创建 介绍过局部变量表的创建，创建完成后的栈帧状态如下图所示。



各个寄存器的状态如下所示。

```
// %rax寄存器中存储的是返回地址
rax: return address
// 要执行的Java方法的指针
rbx: Method*
```

```
// 本地变量表指针
r14: pointer to locals
// 调用者的栈顶
r13: sender sp
```

注意rax中保存的返回地址，因为在generate_call_stub()函数中通过__ call(c_rarg1) 语句调用了由generate_normal_entry()函数生成的entry_point，所以当entry_point执行完成后，还会返回到generate_call_stub()函数中继续执行__ call(c_rarg1) 语句下面的代码，也就是

第5篇-调用Java方法后弹出栈帧及处理返回结果 涉及到的那些代码。

调用的generate_fixed_frame()函数的实现如下：

```
void TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) {

  // 把返回地址紧接着局部变量区保存
  __ push(rax);
  // 为Java方法创建栈帧
  __ enter();
  // 保存调用者的栈顶地址
  __ push(r13);
   // 暂时将last_sp属性的值设置为NULL_WORD
  __ push((int)NULL_WORD);
  // 获取ConstMethod*并保存到r13中
  __ movptr(r13, Address(rbx, Method::const_offset()));
  // 保存Java方法字节码的地址到r13中
  __ lea(r13, Address(r13, ConstMethod::codes_offset()));
  // 保存Method*到堆栈上
  __ push(rbx);

  // ProfileInterpreter属性的默认值为true,
  // 表示需要对解释执行的方法进行相关信息的统计
  if (ProfileInterpreter) {
    Label method_data_continue;
    // MethodData结构基础是ProfileData,
    // 记录函数运行状态下的数据
    // MethodData里面分为3个部分,
    // 一个是函数类型等运行相关统计数据,
    // 一个是参数类型运行相关统计数据,
    // 还有一个是extra扩展区保存着
    // deoptimization的相关信息
    // 获取Method中的_method_data属性的值并保存到rdx中
    __ movptr(rdx, Address(rbx,
           in_bytes(Method::method_data_offset())));
```

```
    __ testptr(rdx, rdx);
    __ jcc(Assembler::zero, method_data_continue);
    // 执行到这里，说明_method_data已经进行了初始化，
    // 通过MethodData来获取_data属性的值并存储到rdx中
    __ addptr(rdx, in_bytes(MethodData::data_offset()));
    __ bind(method_data_continue);
    __ push(rdx);

  } else {
    __ push(0);
  }


  // 获取ConstMethod*存储到rdx
  __ movptr(rdx, Address(rbx,
        Method::const_offset()));
  // 获取ConstantPool*存储到rdx
  __ movptr(rdx, Address(rdx,
         ConstMethod::constants_offset()));
 // 获取ConstantPoolCache*并存储到rdx
  __ movptr(rdx, Address(rdx,
         ConstantPool::cache_offset_in_bytes()));
  // 保存ConstantPoolCache*到堆栈上
  __ push(rdx);
  // 保存第1个参数的地址到堆栈上
  __ push(r14);


  if (native_call) {
   // native方法调用时，不需要保存Java
   // 方法的字节码地址，因为没有字节码
    __ push(0);
  } else {
   // 保存Java方法字节码地址到堆栈上，
   // 注意上面对r13寄存器的值进行了更改
    __ push(r13);
  }


  // 预先保留一个slot，后面有大用处
  __ push(0);
```

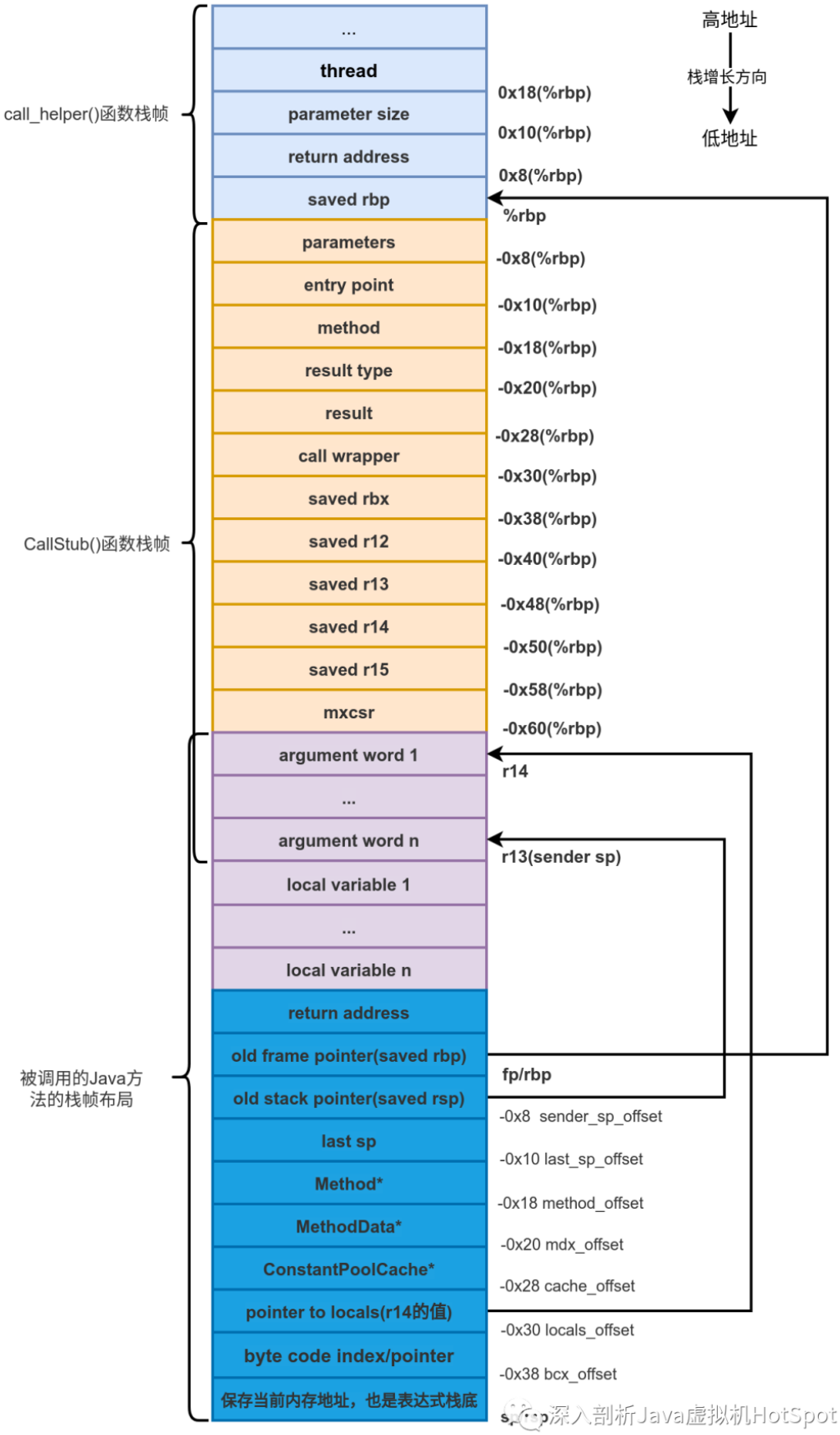
```
  // 将栈底地址保存到这个slot上
  __ movptr(Address(rsp, 0), rsp);

}
```

对于普通的Java方法来说，生成的汇编代码如下：

```
push %rax
push %rbp
mov %rsp,%rbp
push %r13
pushq $0x0
mov 0x10(%rbx),%r13
// lea指令获取内存地址本身
lea 0x30(%r13),%r13
push %rbx
mov 0x18(%rbx),%rdx
test   %rdx,%rdx
je 0x00007fffed01b27d
add $0x90,%rdx
push %rdx
mov 0x10(%rbx),%rdx
mov 0x8(%rdx),%rdx
mov 0x18(%rdx),%rdx
push %rdx
push %r14
push %r13
pushq $0x0
mov %rsp,(%rsp)
```

汇编比较简单，这里不再多说。执行完如上的汇编后生成的栈帧状态如下图所示。



调用完generate_fixed_frame()函数后一些寄存器中保存的值如下：

```
rbx: Method*
ecx: invocation counter
r13: bcp(byte code pointer)
rdx: ConstantPool* 常量池的地址
r14: 本地变量表第1个参数的地址
```

执行完generate_fixed_frame()函数后会继续返回执行InterpreterGenerator::generate_normal_entry()函数，如果是为同步方法生成机器码，那么还需要调用lock_method()函数，这个函数会改变当前栈帧的状态，添加同步所需要的一些信息，在后面介绍锁的实现时会详细介绍。

InterpreterGenerator::generate_normal_entry()函数最终会返回生成机器码的入口执行地址，然后通过变量_entry_table数组来保存，这样就可以使用方法类型做为数组下标获取对应的方法入口了。

收录于合集 #java 9

上一篇
第6篇-Java方法新栈帧的创建

下一篇
第8篇-dispatch_next()函数分派字节码

People who liked this content also liked