Hotspot 方法调用之StubGenerator 源码解析

原创 展开

孙大圣666 发布于2019-11-08 15:24:46 阅读数 27 收藏

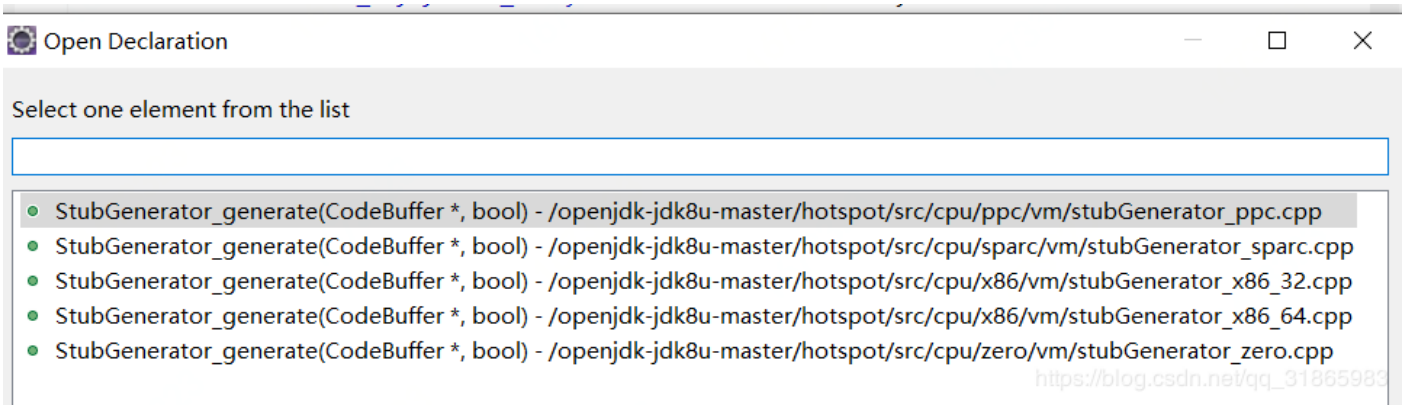
目录

上篇《Hotspot 方法调用之StubRoutines 源码解析》中讲解了StubRoutines初始化相关的类，这篇文章就顺着其中StubGenerator_generate方法的实现来深入了解下JVM解释器相关类的使用。

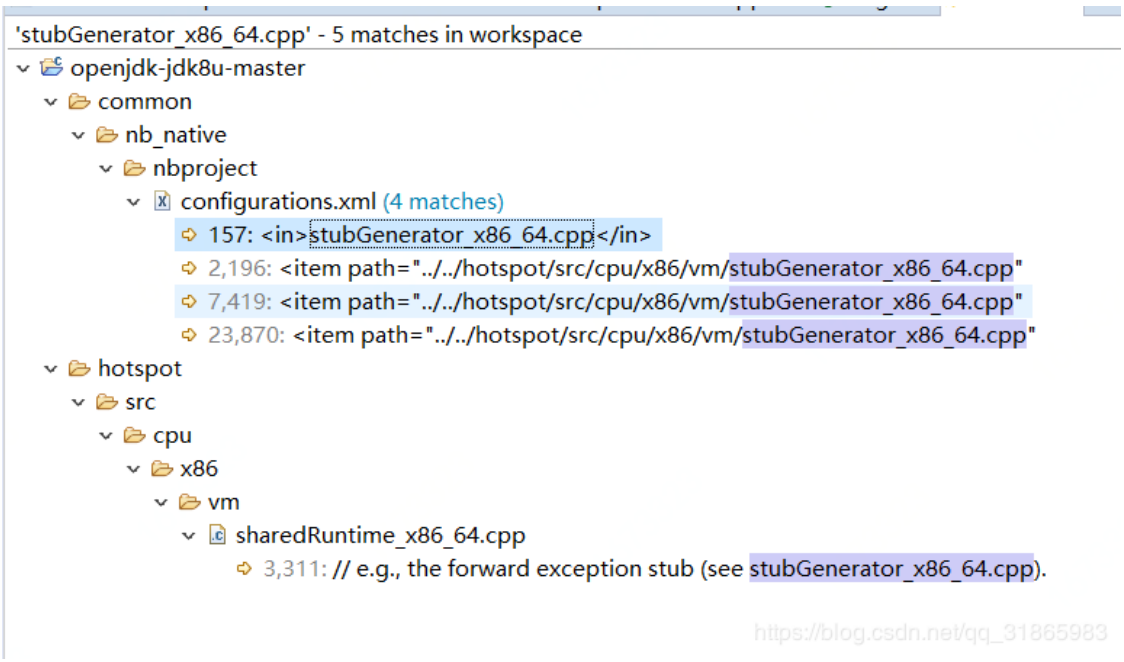
一、StubGenerator

1、StubGenerator_generate

查看StubGenerator_generate方法的实现时，eclipse弹框如下图，即存在不同CPU架构的实现，我们重点关注x86_64的实现，这是后端CentOS服务器的常用的CPU架构。



那么编译器编译的时候怎么知道使用哪个文件的实现了？可以全局搜索包含stubGenerator_x86_64.cpp的文件，结果如下图：



点开这个configurations.xml，如下图：

```
<item path="../../hotspot/src/cpu/x86/vm/stubGenerator_x86_64.cpp"
      ex="false"
      tool="1"
      flavor2="0">
</item>
```

其外层的xml的配置如下图：

```
<conf name="Linux_64" type="0">
  <toolsSet>
  <codeAssistance>
    <envVariables>
    <transientMacros>
  </codeAssistance>
  <makefileType>
  <item path="../../build/linux-x86_64-normal-server-release/hotspot/linux_amd64_compiler2/generated/adfiles/ad_x86_64.cpp"
        ex="false"
        tool="1"
        flavor2="0">
  </item>
```

即 configurations.xml指定了编译Linux-64版本时应该包含哪些文件。

StubGenerator_generate的源码实现如下，下面的几节会逐一讲解相关类的用途。

```
 public:
  StubGenerator(CodeBuffer* code, bool all) : StubCodeGenerator(code) {
    if (all) {
      generate_all();
    } else {
      generate_initial();
    }
  }
}; // end class declaration

void StubGenerator_generate(CodeBuffer* code, bool all) {
  StubGenerator g(code, all);
}
```

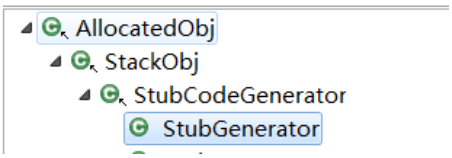

generate_initial和generate_all两个方法都是给StubRoutines中的static public的函数调用地址赋值，即生成stub，如下图：

```
// Initialization
void generate_initial() {
  // Generates all stubs and initializes the entry points

  // This platform-specific settings are needed by generate_call_stub()
  create_control_words();

  // entry points that exist in all platforms Note: This is code
  // that could be shared among different platforms - however the
  // benefit seems to be smaller than the disadvantage of having a
  // much more complicated generator structure. See also comment in
  // stubRoutines.hpp.

  StubRoutines::_forward_exception_entry = generate_forward_exception();

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

  // is referenced by megamorphic call
  StubRoutines::_catch_exception_entry = generate_catch_exception();

  // atomic calls
  StubRoutines::_atomic_xchg_entry          = generate_atomic_xchg();
  StubRoutines::_atomic_xchg_ptr_entry      = generate_atomic_xchg_ptr();
  StubRoutines::_atomic_cmpxchg_entry       = generate_atomic_cmpxchg();
  StubRoutines::_atomic_cmpxchg_long_entry = generate_atomic_cmpxchg_long();
  StubRoutines::_atomic_add_entry           = generate_atomic_add();
  StubRoutines::_atomic_add_ptr_entry       = generate_atomic_add_ptr();
  StubRoutines::_fence_entry                = generate_orderaccess_fence();
```

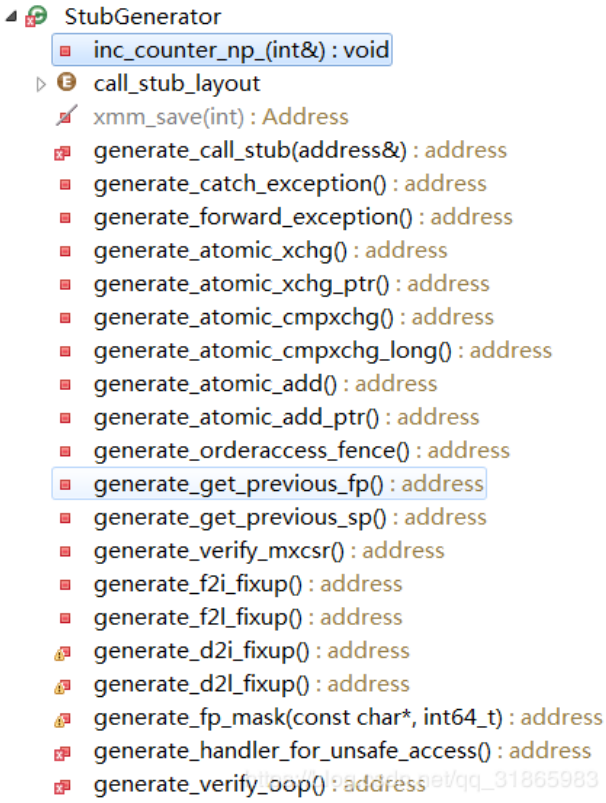
```
void generate_all() {
  // Generates all stubs and initializes the entry points

  // These entry points require SharedInfo::stack0 to be set up in
  // non-core builds and need to be relocatable, so they each
  // fabricate a RuntimeStub internally.
  StubRoutines::_throw_AbstractMethodError_entry =
    generate_throw_exception("AbstractMethodError throw_exception",
                             CAST_FROM_FN_PTR(address,
                                              SharedRuntime::
                                              throw_AbstractMethodError));

  StubRoutines::_throw_IncompatibleClassChangeError_entry =
    generate_throw_exception("IncompatibleClassChangeError throw_exception",
                             CAST_FROM_FN_PTR(address,
                                              SharedRuntime::
                                              throw_IncompatibleClassChangeError));

  StubRoutines::_throw_NullPointerException_at_call_entry =
    generate_throw_exception("NullPointerException at call throw_exception",
                             CAST_FROM_FN_PTR(address,
                                              SharedRuntime::
                                              throw_NullPointerException_at_call));
```

2、定义

StubGenerator顾名思义就是用来生成Stub的，这里的Stub实际是一段可执行的汇编代码，具体来说就是生成StubRoutines中定义的多个public static的函数调用点，调用方可以将其作为一个经过优化后的函数直接使用。其类继承关系如下：



StubGenerator继承自StubCodeGenerator，但是没有添加任何新的属性，其对外的public方法只有一个构造方法，参考上节的StubGenerator_generate方法的源码。其他的都是用来生成单个Stub的私有方法，这些私有方法通过generate_initial和generate_all调用，如下图：



3、generate_atomic_xchg和generate_disjoint_byte_copy

generate_atomic_xchg生成的函数相当于jint atomic::xchg(jint exchange_value, volatile jint* dest)，即原子的将exchange_value设置到dest指向的内存中，如果设置成功则返回原来的值。generate_disjoint_byte_copy用于来复制int或者byte数组。以这两个为例说明StubGenerator生成stub的实现。

generate_atomic_xchg的源码说明如下：

```
// Support for jint atomic::xchg(jint exchange_value, volatile jint* dest)
//
// Arguments :
//    c_rarg0: exchange_value
//    c_rarg1 : dest
//
// Result:
//    *dest <- ex, return (orig *dest)
address generate_atomic_xchg() {
  //通过StubCodeMark的构造和析构函数来执行必要的资源初始化和销毁
  StubCodeMark mark(this, "StubRoutines", "atomic_xchg");
  //__是_masm->的别名，masm是StubCodeGenerator中的属性，MacroAssembler*，表示汇编代码的生成器
  //pc()是MacroAssembler的方法，返回MacroAssembler绑定的CodeSection的end地址
  address start = __ pc();
  //movl()也是MacroAssembler的方法，对应movl汇编指令，其入参是两个Register，即寄存器
  //rax和c_rarg0分别对应rax寄存器和通常用于保存第一个参数的rdi寄存器
  //这里是将rdi寄存器中的数据拷贝到rax中
  __ movl(rax, c_rarg0); // Copy to eax we need a return value anyhow
  //将rax中的值同c_rarg1寄存器即rsi寄存器中保存的地址上的值原子的交换，如果交换成功，rax中的值就是原来dest的值
  //可以先读取dest的值，然后再执行原子交换，如果交换的结果是之前读取的dest的值说明当前线程加锁成功
  __ xchgl(rax, Address(c_rarg1, 0)); // automatic LOCK
  //ret表示调用结束，返回
  __ ret(0);

  return start;
}
```

```
#define __ _masm->
```

generate_disjoint_byte_copy的源码说明如下：

```
// 本方法的参数：
//   aligned - true => 如果为true，则表示输入输出都已经按照8字节对齐了
//   name    - stub的名字
//   entry   - 就是address*,无特别意义
//
// 调用此stub的参数:
//   c_rarg0   - 原数组的地址
//   c_rarg1   - 目标数组的地址
//   c_rarg2   - 元素个数
//
// If 'from' and/or 'to' are aligned on 4-, 2-, or 1-byte boundaries,
// we let the hardware handle it.  The one to eight bytes within words,
// dwords or qwords that span cache line boundaries will still be loaded
// and stored atomically.
//
// Side Effects:
//   disjoint_byte_copy_entry is set to the no-overlap entry point
//   used by generate_conjoint_byte_copy().
//
address generate_disjoint_byte_copy(bool aligned, address* entry, const char *name) {
  //让Assamber关联的CodeSection按指定的内存大小对齐
  __ align(CodeEntryAlignment);
  //通过StubCodeMark的构造和析构函数执行初始化和善后处理工作
  StubCodeMark mark(this, "StubRoutines", name);
  //获取Assamber关联的CodeSection的end属性，即汇编代码写入的起始地址
  address start = __ pc();

  //Label表示跳转指令用到的标签
  Label L_copy_bytes, L_copy_8_bytes, L_copy_4_bytes, L_copy_2_bytes;
  Label L_copy_byte, L_exit;
  //Register表示寄存器
  const Register from        = rdi;  // 原数组地址
  const Register to          = rsi;  // 目标数组地址
  const Register count       = rdx;  // 元素个数
  const Register byte_count  = rcx;
  const Register qword_count = count;
  const Register end_from    = from; // source array end address
  const Register end_to      = to;   // destination array end address
  // End pointers are inclusive, and if count is not zero they point
  // to the last unit copied:  end_to[0] := end_from[0]

  __ enter(); // 往下移动rbp，开启一个新的栈帧
  assert_clean_int(c_rarg2, rax);    // 验证rdx寄存器中保存的数值是一个32位的int

  if (entry != NULL) {
    //将end属性赋值给entry
    *entry = __ pc();
     // 添加注释
    BLOCK_COMMENT("Entry:");
  }
  //window下将r9和r10寄存器中的数据拷贝到rdi和rsi寄存器中，从而与linux等保持一致
  setup_arg_regs(); // from => rdi, to => rsi, count => rdx
                    // r9 and r10 may be used to save non-volatile registers

  // 'from', 'to' and 'count' are now valid
```

```
  //将count对应的rdx寄存器的值拷贝到byte_count对应的寄存器rcx中
  __ movptr(byte_count, count);
  //shr是逻辑右移指令，即将count的值右移3位
  __ shrptr(count, 3); // count => qword_count

  // Copy from low to high addresses.  Use 'to' as scratch.
  //lea指令用来加载有效地址到指定的寄存器
  __ lea(end_from, Address(from, qword_count, Address::times_8, -8));
  __ lea(end_to,   Address(to,   qword_count, Address::times_8, -8));
  //neg指令用来对操作数取补，qword_count变成count的负数
  __ negptr(qword_count); // make the count negative
  //jmp指令用来跳转到指定的地址
  __ jmp(L_copy_bytes);

  // Copy trailing qwords
__ BIND(L_copy_8_bytes);
  __ movq(rax, Address(end_from, qword_count, Address::times_8, 8));
  __ movq(Address(end_to, qword_count, Address::times_8, 8), rax);
  __ increment(qword_count);
  __ jcc(Assembler::notZero, L_copy_8_bytes);

  // Check for and copy trailing dword
__ BIND(L_copy_4_bytes);
  __ testl(byte_count, 4);
  __ jccb(Assembler::zero, L_copy_2_bytes);
  __ movl(rax, Address(end_from, 8));
  __ movl(Address(end_to, 8), rax);

  __ addptr(end_from, 4);
  __ addptr(end_to, 4);

  // Check for and copy trailing word
__ BIND(L_copy_2_bytes);
  __ testl(byte_count, 2);
  __ jccb(Assembler::zero, L_copy_byte);
  __ movw(rax, Address(end_from, 8));
  __ movw(Address(end_to, 8), rax);

  __ addptr(end_from, 2);
  __ addptr(end_to, 2);

  // Check for and copy trailing byte
__ BIND(L_copy_byte);
  __ testl(byte_count, 1);
  //检查byte_count的值是否是零，如果是则跳转到L_exit
  __ jccb(Assembler::zero, L_exit);
  //将rax的值拷贝到end_from往后的8字节上
  __ movb(rax, Address(end_from, 8));
  //将end_to往后8个字节的地址拷贝到rax中
  __ movb(Address(end_to, 8), rax);

__ BIND(L_exit);
  //恢复原来的寄存器的值
  restore_arg_regs();
  inc_counter_np(SharedRuntime::_jbyte_array_copy_ctr); // Update counter after rscratch1 is free
  __ xorptr(rax, rax); // return 0
  __ leave(); // required for proper stackwalking of RuntimeStub frame
  __ ret(0);

  //复制qword_count字节的数据
  copy_bytes_forward(end_from, end_to, qword_count, rax, L_copy_bytes, L_copy_8_bytes);
```

```
    __ jmp(L_copy_4_bytes);
    return start;
  }
```

4、generate_call_stub

_call_stub_entry就是执行Java方法调用的Stub，如下图：

```
// Calls to Java
typedef void (*CallStub)(
  address   link,
  intptr_t* result,
  BasicType result_type,
  Method* method,
  address   entry_point,
  intptr_t* parameters,
  int       size_of_parameters,
  TRAPS
);

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

_call_stub_entry的初始化是在generate_initial方法中完成的，如下图：

```
// Initialization
void generate_initial() {
  // Generates all stubs and initializes the entry points

  // This platform-specific settings are needed by generate_call_stub()
  create_control_words();

  // entry points that exist in all platforms Note: This is code
  // that could be shared among different platforms - however the
  // benefit seems to be smaller than the disadvantage of having a
  // much more complicated generator structure. See also comment in
  // stubRoutines.hpp.

  StubRoutines::_forward_exception_entry = generate_forward_exception();

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

下面我们就来深入分析下generate_call_stub方法的实现源码，其说明如下：

```
address generate_call_stub(address& return_address) {
   //rsp_after_call_off和call_wrapper_off都是定义的枚举，表示对应项相对于rbp的偏移字节数
   //比如call_wrapper_off就是JavaCallWrapper实例相对于rbp的偏移字节数
   //这里是校验栈帧的属性和当前系统的属性是否一致
    assert((int)frame::entry_frame_after_call_words == -(int)rsp_after_call_off + 1 &&
           (int)frame::entry_frame_call_wrapper_offset == (int)call_wrapper_off,
           "adjust this code");
    StubCodeMark mark(this, "StubRoutines", "call_stub");
    //获取写入汇编代码的内存地址
    address start = __ pc();

    //根据各项的偏移量计算各项的存储位置
    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); //entry_point就是解释器的调用入口
const Address parameters    (rbp, parameters_off     * wordSize);
const Address parameter_size(rbp, parameter_size_off * wordSize);
    // same as in generate_catch_exception()!
    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);

    // stub code
    //enterf方法是保存rbp寄存器到栈中，然后把rsp中的值拷贝到rbp中
    __ enter();
    //sub指令是减去特定值，这里是将rsp往低地址方向移动指定的偏移量，至此一个新的栈帧展开了
    __ subptr(rsp, -rsp_after_call_off * wordSize);

    // 即不是WIN64系统
#ifndef _WIN64
    //将c_rarg5即r9寄存器的值拷贝到parameters地址上
    __ movptr(parameters,   c_rarg5); // parameters
    //将c_rarg4即r8寄存器的值拷贝到entry_point地址上
    __ movptr(entry_point,  c_rarg4); // entry_point
#endif
    //同上，c_rarg3对应rcx寄存器
    __ movptr(method,       c_rarg3); // method
    //c_rarg2对应rdx寄存器
    __ movl(result_type,  c_rarg2);   // result type
    //c_rarg1对应rsi寄存器
    __ movptr(result,       c_rarg1); // result
    //c_rarg0对应rdi寄存器
    __ movptr(call_wrapper, c_rarg0); // call wrapper

    // 将下列寄存器的值复制到对应的地址上
    __ movptr(rbx_save, rbx);
    __ movptr(r12_save, r12);
    __ movptr(r13_save, r13);
    __ movptr(r14_save, r14);
    __ movptr(r15_save, r15);
#ifdef _WIN64
    for (int i = 6; i <= 15; i++) {
      __ movdqu(xmm_save(i), as_XMMRegister(i));
    }

    const Address rdi_save(rbp, rdi_off * wordSize);
    const Address rsi_save(rbp, rsi_off * wordSize);

    __ movptr(rsi_save, rsi);
    __ movptr(rdi_save, rdi);
#else
    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);
    }
#endif

    // Load up thread register
    //将r15_thread即r15寄存器中的数值拷贝到thread处的栈中
    __ movptr(r15_thread, thread);
    //heapbase的重新初始化话
    __ reinit_heapbase();


    // pass parameters if any
    BLOCK_COMMENT("pass parameters if any");
    Label parameters_done;
    //parameter_size拷贝到c_rarg3即rcx寄存器中
    __ movl(c_rarg3, parameter_size);
    //校验c_rarg3的数值是否合法
    __ testl(c_rarg3, c_rarg3);
    //如果不合法则跳转到parameters_done分支上
    __ jcc(Assembler::zero, parameters_done);

    Label loop;‘=
      //将地址parameters包含的数据即参数对象的指针拷贝到c_rarg2寄存器中
    __ movptr(c_rarg2, parameters);       // parameter pointer
    //将c_rarg3中值拷贝到c_rarg1中，即将参数个数复制到c_rarg1中
    __ movl(c_rarg1, c_rarg3);            // parameter counter is in c_rarg1
    //打标
    __ BIND(loop);
    //将c_rarg2指向的内存中包含的地址复制到rax中
    __ movptr(rax, Address(c_rarg2, 0));// get parameter
    //c_rarg2中的参数对象的指针加上指针宽度8字节，即指向下一个参数
    __ addptr(c_rarg2, wordSize);       // advance to next parameter
    //将c_rarg1中的值减一
    __ decrementl(c_rarg1);             // decrement counter
    //传递方法调用参数
    __ push(rax);                       // pass parameter
    //如果参数个数大于0则跳转到loop继续
    __ jcc(Assembler::notZero, loop);

    // 打标
    __ BIND(parameters_done);
    //将method地址包含的数据接Method*拷贝到rbx中
    __ movptr(rbx, method);             // get Method*
    //将解释器的入口地址拷贝到c_rarg1寄存器中
    __ movptr(c_rarg1, entry_point);    // get entry_point
    //将rsp寄存器的数据拷贝到r13寄存器中
    __ mov(r13, rsp);                   // set sender sp

    BLOCK_COMMENT("call Java function");
    //调用解释器的解释函数，从而调用Java方法
    __ call(c_rarg1);

    BLOCK_COMMENT("call_stub_return_address:");
    //获取此时的汇编代码写入位置，
    return_address = __ pc();

    //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INT处理
    //将result地址的值拷贝到c_rarg0中
    __ movptr(c_rarg0, result);
```

```
    Label is_long, is_float, is_double, exit;          //将result_type地址的值拷贝到c_rarg1
    __ 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);

    // 处理结果类型是int的情形，将rax中的值写入c_rarg0对应地址的内存中
    __ movl(Address(c_rarg0, 0), rax);
    //打标
    __ BIND(exit);

    // rsp_after_call的有效地址拷贝到rsp中，即将rsp往高地址方向移动了，原来的方法调用参数相当于pop掉了
    __ lea(rsp, rsp_after_call);


    // restore regs belonging to calling function
#ifdef _WIN64
    for (int i = 15; i >= 6; i--) {
      __ movdqu(as_XMMRegister(i), xmm_save(i));
    }
#endif
    //恢复其他寄存器的值，即恢复方法调用前的现场
    __ movptr(r15, r15_save);
    __ movptr(r14, r14_save);
    __ movptr(r13, r13_save);
    __ movptr(r12, r12_save);
    __ movptr(rbx, rbx_save);

#ifdef _WIN64
    __ movptr(rdi, rdi_save);
    __ movptr(rsi, rsi_save);
#else
    __ ldmxcsr(mxcsr_save);
#endif

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

    // 恢复rbp
    __ pop(rbp);
    //退出
    __ ret(0);

    // handle return types different from T_INT
    __ BIND(is_long);
    //调用不同的指令将rax中的值写入c_rarg0的地址对应的内存中
    __ 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);

    return start;
  }
```

执行Java方法调用的思路整体上是先将用来传递参数的所有寄存器的值拷贝到栈帧中，再从栈帧中将必要的方法调用参数，参数个数，Method*等复制到寄存器中，然后执行方法调用，调用结束再从栈帧中将参数寄存器中的值恢复至执行方法调用前的状态，在恢复rsp，rbp等就可以正常退出调用了。这里涉及了诸多底层调用栈帧的演变，寄存器和汇编指令的使用相关的知识，可以参考《C与汇编语言》。

5、栈帧的演变

上述代码中涉及的栈帧的演变可以参考StubGenerator的注释，如下图：

```
// Call stubs are used to call Java from C
//
// Linux Arguments:
//    c_rarg0:   call wrapper address                   address
//    c_rarg1:   result                                 address
//    c_rarg2:   result type                            BasicType
//    c_rarg3:   method                                 Method*
//    c_rarg4:   (interpreter) entry point              address
//    c_rarg5:   parameters                             intptr_t*
//    16(rbp): parameter size (in words)              int
//    24(rbp): thread                                 Thread*
//
//     [ return_from_Java     ] <--- rsp
//     [ argument word n      ]
//      ...
// -12 [ argument word 1      ]
// -11 [ saved r15            ] <--- rsp_after_call
// -10 [ saved r14            ]
//  -9 [ saved r13            ]
//  -8 [ saved r12            ]
//  -7 [ saved rbx            ]
//  -6 [ call wrapper         ]
//  -5 [ result               ]
//  -4 [ result type          ]
//  -3 [ method               ]
//  -2 [ entry point          ]
//  -1 [ parameters           ]
//   0 [ saved rbp            ] <--- rbp
//   1 [ return address       ]
//   2 [ parameter size       ]
//   3 [ thread               ]
//
```


上图的rbp的位置就是一个新的栈帧的起始地址，是执行StubRoutines::call_stub()方法前的一个栈帧的rsp指针，开始调用call_stub方法，call_stub的前6个参数会依次拷贝c_rarg0到c_rarg5 6个寄存器中，对应于下列一段代码，如下：

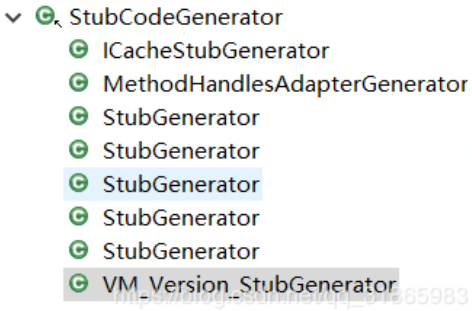
```
#ifndef _WIN64
    __ movptr(parameters,   c_rarg5); // parameters
    __ movptr(entry_point,  c_rarg4); // entry_point
#endif

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

因为x86的CPU的寄存器最多只能传递6个参数， call_stub()的第七个参数parameter_size和第8个参数thread就通过上一栈帧来传递了，因此注释中parameter size相对于的rbp的偏移是2个字宽，thread的偏移是3个字宽。参数准备好后就会将当前rbp的地址保存，即上面的saved rbp，然后将rsp寄存器的值复制到rbp中，即rbp指向了原来的rsp的位置，rsp再往上即低地址方向移动若干个字段，至此一个新的栈帧形成了，然后就开始依次将c_rarg0到c_rarg5 6个寄存器，rbx，r12-r15寄存器的数据拷贝到栈帧中，即对应注释中rbp到rsp_after_call的一段，接

着就是将方法调用的参数依次从JavaCallArguments中取出，放到占中，即对应注释中rsp_after_call到rsp的一段，参数复制完毕后就是执行Java方法调用，又是开启一个新的栈帧了。

二、StubCodeGenerator

StubCodeGenerator是所有的Stub generators的基类，主要提供一些公用的工具性方法，其子类如下图：



StubGenerator有四个是因为不同的CPU架构的实现不同，后面的文章中会逐一介绍相关子类。

StubCodeGenerator的定义位于hotspot src/share/vm/runtime/stubCodeGenerator.hpp中，其定义了如下四个私有属性：

- _masm：MacroAssembler*，用来生成汇编代码
- _first_stub：StubCodeDesc*，第一个生成的stub
- _last_stub：StubCodeDesc*，最后一个生成的stub
- _print_code：bool，是否打印汇编代码

StubCodeGenerator定义的方法不多，关键是其构建和析构方法，源码说明如下：

```
StubCodeGenerator::StubCodeGenerator(CodeBuffer* code, bool print_code) {
   //构造一个新的MacroAssembler实例
  _masm = new MacroAssembler(code);
  _first_stub = _last_stub = NULL;
  _print_code = print_code;
}

StubCodeGenerator::~StubCodeGenerator() {
  if (PrintStubCode || _print_code) {
    CodeBuffer* cbuf = _masm->code();
    //CodeBuffer的inst section的start是用CodeBlob的content_start初始化的，这里是反向查找
    CodeBlob*   blob = CodeCache::find_blob_unsafe(cbuf->insts()->start());
    if (blob != NULL) {
      //将cbuf的_code_strings设置到blob中
      blob->set_strings(cbuf->strings());
    }
    bool saw_first = false;
    StubCodeDesc* toprint[1000];
    int toprint_len = 0;
    //遍历所有的StubCodeDesc，将其放到toprint数组中
    for (StubCodeDesc* cdesc = _last_stub; cdesc != NULL; cdesc = cdesc->_next) {
      toprint[toprint_len++] = cdesc;
      if (cdesc == _first_stub) { saw_first = true; break; }
    }
    assert(saw_first, "must get both first & last");
    //将其按照StubCodeDesc的index属性排序
    qsort(toprint, toprint_len, sizeof(toprint[0]), compare_cdesc);
    //遍历排序后的StubCodeDesc
    for (int i = 0; i < toprint_len; i++) {
      StubCodeDesc* cdesc = toprint[i];
```

```
      //打印原始的汇编代码
      cdesc->print();
      //写入换行符，tty是outputStream*
      tty->cr();
      //打印反汇编代码，即数字形式的汇编指令转换成对应的速记符，并加上java程序的相关信息
      Disassembler::decode(cdesc->begin(), cdesc->end());
      tty->cr();
    }
  }
}
```

三、StubCodeDesc

StubCodeDesc用来描述一段生成的Stub，StubCodeDesc保存的信息通常用于调试和打印日志。目前所有的StubCodeDesc都是链式保存的，如果查找比较慢就可能会改变。StubCodeDesc同样定义在stubCodeGenerator.hpp中，其包含的protected属性如下：

- _list：static StubCodeDesc* volatile，StubCodeDesc实例链表
- _count：static int，链表的长度
- _next：StubCodeDesc*，当前StubCodeDesc实例的下一个StubCodeDesc实例
- _group：char*，stub所属的组
- _name：char*，stub的名称
- _index：int，当前StubCodeDesc实例的序号
- _begin：address，stub code的起始地址
- _end：address，stub code的结束地址，即对应内存区域的下一个字节的地址

除构造方法外，StubCodeDesc定义的方法主要有两种：

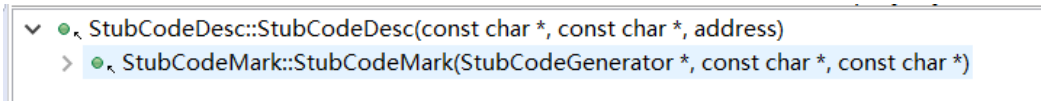
- 属性操作相关的实例方法，如group，index，set_begin，size_in_bytes等
- 在StubCodeDesc链表中搜索的静态方法，如desc_for，desc_for_index，name_for等

构造方法和desc_for等静态搜索方法源码说明如下：

```
StubCodeDesc(const char* group, const char* name, address begin) {
    assert(name != NULL, "no name specified");
    //_list相当于链表头的StubCodeDesc指针，每创建一个新的StubCodeDesc实例则插入到链表的头部
    //将原来的头部实例作为当前实例的的_next
    _next           = (StubCodeDesc*)OrderAccess::load_ptr_acquire(&_list);
    _group          = group;
    _name           = name;
    _index          = ++_count; // (never zero)
    _begin          = begin;
    _end            = NULL;
    //将当前实例作为新的链表头部实例指针
    OrderAccess::release_store_ptr(&_list, this);
  };

  inline void*    OrderAccess::load_ptr_acquire(volatile void*       p) { return *(void* volatile *)p; }

  inline void     OrderAccess::release_store_ptr(volatile void*     p, void*    v) { *(void* volatile *)p = v; }


StubCodeDesc* StubCodeDesc::desc_for(address pc) {
  StubCodeDesc* p = (StubCodeDesc*)OrderAccess::load_ptr_acquire(&_list);
  //遍历所有的StubCodeDesc实例，直到找到一个包含目标地址的实例
  while (p != NULL && !p->contains(pc)) p = p->_next;
  // p == NULL || p->contains(pc)
  return p;
```

```
}

  bool    contains(address pc) const         { return _begin <= pc && pc < _end; }

StubCodeDesc* StubCodeDesc::desc_for_index(int index) {
  StubCodeDesc* p = (StubCodeDesc*)OrderAccess::load_ptr_acquire(&_list);
  while (p != NULL && p->index() != index) p = p->_next;
  return p;
}

//获取包含指定地址的StubCodeDesc的name属性
const char* StubCodeDesc::name_for(address pc) {
  StubCodeDesc* p = desc_for(pc);
  return p == NULL ? NULL : p->name();
}
```

StubCodeDesc的构造方法的调用方只有一个StubCodeMark，如下图：



四、StubCodeMark

StubCodeMark是一个工具类，用于将一个生成的stub同其名称关联起来，StubCodeMark会给当前stub创建一个新的StubCodeDesc实例，并将其注册到全局的StubCodeDesc链表中，stub可以通过地址查找到对应的StubCodeDesc实例。StubCodeMark的定义同样在stubCodeGenerator.hpp中，其定义的属性有两个，如下：

- _cgen：StubCodeGenerator*，生成当前stub的StubCodeGenerator实例
- _cdesc：StubCodeDesc*，描述当前stub的StubCodeDesc实例

StubCodeMark只有构造和析构函数，没有多余的方法，其源码说明如下：

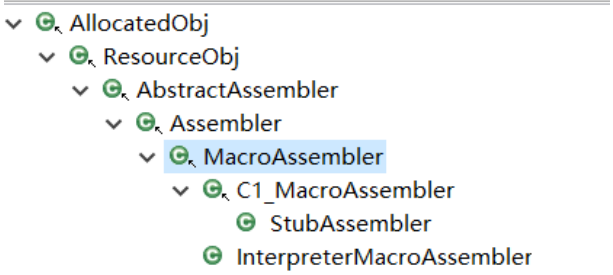
```
StubCodeMark::StubCodeMark(StubCodeGenerator* cgen, const char* group, const char* name) {
  _cgen  = cgen;
  //_cgen->assembler()->pc()返回的是StubCodeDesc的start属性，即stub code的起始地址
  _cdesc = new StubCodeDesc(group, name, _cgen->assembler()->pc());
  _cgen->stub_prolog(_cdesc);
  //重置stub code的起始地址，避免stub_prolog中改变了起始地址
  _cdesc->set_begin(_cgen->assembler()->pc());
}

//stub_prolog是一个虚方法，默认是空实现
void StubCodeGenerator::stub_prolog(StubCodeDesc* cdesc) {
  // default implementation - do nothing
}

StubCodeMark::~StubCodeMark() {
  //flush方法将生成的汇编代码写入到CodeBuffer中
  _cgen->assembler()->flush();
  //设置end属性
  _cdesc->set_end(_cgen->assembler()->pc());
  //校验当前StubCodeDesc处于链表头部，即在StubCodeMark构造完成到析构前没有创建一个新的StubCodeDesc实例
  assert(StubCodeDesc::_list == _cdesc, "expected order on list");
  _cgen->stub_epilog(_cdesc);
  //将生成的stub注册到操作系统中，相当于操作系统加载了某个函数的实现到当前进程的代码区
  Forte::register_stub(_cdesc->name(), _cdesc->begin(), _cdesc->end());
  //发布Jvmti事件
  if (JvmtiExport::should_post_dynamic_code_generated()) {
```

```
    JvmtiExport::post_dynamic_code_generated(_cdesc->name(), _cdesc->begin(), _cdesc->end());
  }
}

//stub_epilog是一个虚方法，下列是默认实现
void StubCodeGenerator::stub_epilog(StubCodeDesc* cdesc) {
  //设置StubCodeGenerator的_last_stub属性
  if (_first_stub == NULL)  _first_stub = cdesc;
  _last_stub = cdesc;
}
```

五、MacroAssembler

1、定义

MacroAssembler扩展自Assembler，主要是添加了一些常用的汇编指令支持，用来生成汇编代码即Stub的。其类继承关系估下：



AbstractAssembler定义了生成汇编代码的抽象公共基础方法，如获取关联CodeBuffer的当前内存位置的pc()方法，将汇编指令全部刷新的CodeBuffer中的flush()方法，绑定跳转标签的bind方法等。其定义位于hotspot src/share/vm/asm/assembler.hpp中。

assembler.hpp中除定义AbstractAssembler外，还定义了用来表jmp跳转指令用到的标签的Lable类，调用bind方法后就会将当前Lable实例绑定到指令流中一个特定的位置，比如jmp指令接收Lable参数，就会跳转到对应的位置处开始执行，可用于实现循环或者条件判断等控制流操作。

Assembler的定义跟CPU架构有关，通过assembler.hpp中的宏包含特定CPU下的Assembler实现，如下图：

```
#ifdef TARGET_ARCH_x86
# include "assembler_x86.hpp"
#endif
#ifdef TARGET_ARCH_sparc
# include "assembler_sparc.hpp"
#endif
#ifdef TARGET_ARCH_zero
# include "assembler_zero.hpp"
#endif
#ifdef TARGET_ARCH_arm
# include "assembler_arm.hpp"
#endif
#ifdef TARGET_ARCH_ppc
# include "assembler_ppc.hpp"
#endif
```


Assembler类添加了特定于CPU架构的指令实现和指令操作相关的枚举，如下图：

```
enum Condition {
  zero          = 0x4,
  notZero       = 0x5,
  equal         = 0x4,
  notEqual      = 0x5,
  less          = 0xc,
  lessEqual     = 0xe,
  greater       = 0xf,
  greaterEqual  = 0xd,
  below         = 0x2,
  belowEqual    = 0x6,
  above         = 0x7,
  aboveEqual    = 0x3,
  overflow      = 0x0,
  noOverflow    = 0x1,
  carrySet      = 0x2,
  carryClear    = 0x3,
  negative      = 0x8,
  positive      = 0x9,
  parity        = 0xa,
  noParity      = 0xb
};
```

其中指令实现根据指令操作数类型和位数的差异，有多个不同的重载版本，如下：

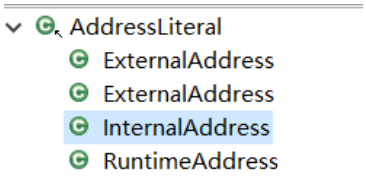
- membar(Membar_mask_bits) : void
- mfence() : void
- mov64(Register, int64_t) : void
- movb(Address, Register) : void
- movb(Address, int) : void
- movb(Register, Address) : void
- movdl(XMMRegister, Register) : void
- movdl(Register, XMMRegister) : void
- movdl(XMMRegister, Address) : void
- movdl(Address, XMMRegister) : void
- movdq(XMMRegister, Register) : void
- movdq(Register, XMMRegister) : void
- movdqa(XMMRegister, XMMRegister) : voi
- movdqa(XMMRegister, Address) : void
- movdqu(Address, XMMRegister) : void
- movdqu(XMMRegister, Address) : void
- movdqu(XMMRegister, XMMRegister) : voi
- vmovdqu(Address, XMMRegister) : void
- vmovdqu(XMMRegister, Address) : void
- vmovdqu(XMMRegister, XMMRegister) : vo
- movlhps(XMMRegister, XMMRegister) : voi
- movl(Register, int32_t) : void
- movl(Address, int32_t) : void
- movl(Register, Register) : void
- movl(Register, Address) : void

注意 Assembler类生成的汇编代码并没有经过指令优化的，使用什么指令最终得到的就是什么指令。

assembler_x86.hpp中除定义了Assembler类以外，还定义了表示内存地址的多个不同Address类，如下图：

- Address
- AddressLiteral
- RuntimeAddress
- ExternalAddress
- InternalAddress
- ArrayAddress

其中RuntimeAddress等是AddressLiteral的子类，如下图。AddressLiteral主要是为了应对部分特殊指令在32位和64位下处理不一样的情形，三个子类是特定于不同的地址类型的。



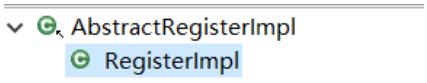
assembler_x86.hpp中还定义了特定于该CPU架构的寄存器枚举，如下图：

```
#ifdef _WIN64

REGISTER_DECLARATION(Register, c_rarg0, rcx);
REGISTER_DECLARATION(Register, c_rarg1, rdx);
REGISTER_DECLARATION(Register, c_rarg2, r8);
REGISTER_DECLARATION(Register, c_rarg3, r9);

REGISTER_DECLARATION(XMMRegister, c_farg0, xmm0);
REGISTER_DECLARATION(XMMRegister, c_farg1, xmm1);
REGISTER_DECLARATION(XMMRegister, c_farg2, xmm2);
REGISTER_DECLARATION(XMMRegister, c_farg3, xmm3);

#else

REGISTER_DECLARATION(Register, c_rarg0, rdi);
REGISTER_DECLARATION(Register, c_rarg1, rsi);
REGISTER_DECLARATION(Register, c_rarg2, rdx);
REGISTER_DECLARATION(Register, c_rarg3, rcx);
REGISTER_DECLARATION(Register, c_rarg4, r8);
REGISTER_DECLARATION(Register, c_rarg5, r9);

REGISTER_DECLARATION(XMMRegister, c_farg0, xmm0);
REGISTER_DECLARATION(XMMRegister, c_farg1, xmm1);
REGISTER_DECLARATION(XMMRegister, c_farg2, xmm2);
REGISTER_DECLARATION(XMMRegister, c_farg3, xmm3);
REGISTER_DECLARATION(XMMRegister, c_farg4, xmm4);
REGISTER_DECLARATION(XMMRegister, c_farg5, xmm5);
REGISTER_DECLARATION(XMMRegister, c_farg6, xmm6);
REGISTER_DECLARATION(XMMRegister, c_farg7, xmm7);

#endif // _WIN64
```


REGISTER_DECLARATION的宏定义如下：

```
#define REGISTER_DECLARATION(type, name, value) \
extern const type name;                         \
enum { name##_##type##EnumValue = value##_##type##EnumValue }
```

Register其实是RegisterImpl*的别名，RegisterImpl的类继承关系如下：



其中AbstractRegisterImpl的定义位于同目录下的register.hpp中，RegisterImpl的更CPU架构相关，在register.hpp中通过宏的方式引入特定CPU架构的实现版本，如下图：

```
#ifdef TARGET_ARCH_x86
# include "register_x86.hpp"
#endif
#ifdef TARGET_ARCH_sparc
# include "register_sparc.hpp"
#endif
#ifdef TARGET_ARCH_zero
# include "register_zero.hpp"
#endif
#ifdef TARGET_ARCH_arm
# include "register_arm.hpp"
#endif
#ifdef TARGET_ARCH_ppc
# include "register_ppc.hpp"
#endif
```


在register_x86.hpp中定义了x86下的寄存器的常量和枚举，如下图：

```
CONSTANT_REGISTER_DECLARATION(Register, noreg, (-1));


CONSTANT_REGISTER_DECLARATION(Register, rax,    (0));
CONSTANT_REGISTER_DECLARATION(Register, rcx,    (1));
CONSTANT_REGISTER_DECLARATION(Register, rdx,    (2));
CONSTANT_REGISTER_DECLARATION(Register, rbx,    (3));
CONSTANT_REGISTER_DECLARATION(Register, rsp,    (4));
CONSTANT_REGISTER_DECLARATION(Register, rbp,    (5));
CONSTANT_REGISTER_DECLARATION(Register, rsi,    (6));
CONSTANT_REGISTER_DECLARATION(Register, rdi,    (7));
#ifdef AMD64
CONSTANT_REGISTER_DECLARATION(Register, r8,     (8));
CONSTANT_REGISTER_DECLARATION(Register, r9,     (9));
CONSTANT_REGISTER_DECLARATION(Register, r10,   (10));
CONSTANT_REGISTER_DECLARATION(Register, r11,   (11));
CONSTANT_REGISTER_DECLARATION(Register, r12,   (12));
CONSTANT_REGISTER_DECLARATION(Register, r13,   (13));
CONSTANT_REGISTER_DECLARATION(Register, r14,   (14));
CONSTANT_REGISTER_DECLARATION(Register, r15,   (15));
#endif // AMD64
```


CONSTANT_REGISTER_DECLARATION定义如下：

```
#define CONSTANT_REGISTER_DECLARATION(type, name, value) \
extern const type name;                                  \
enum { name##_##type##EnumValue = (value) }
```

即REGISTER_DECLARATION和CONSTANT_REGISTER_DECLARATION两者关联起来，如c_rarg0对应rdi，其对应关系如下图：

```
|------------------------------------------------------|
| c_rarg0   c_rarg1  c_rarg2 c_rarg3 c_rarg4 c_rarg5   |
|------------------------------------------------------|
| rcx       rdx      r8      r9      rdi*    rsi*      | windows (* not a c_rarg)
| rdi       rsi      rdx     rcx     r8      r9        | solaris/linux
|------------------------------------------------------|
| j_rarg5   j_rarg0  j_rarg1 j_rarg2 j_rarg3 j_rarg4   |
|------------------------------------------------------|
```

MacroAssembler的实现也是跟CPU架构相关的，特定架构的实现通过同目录下的macroAssembler.hpp的宏定义引入，如下图：

```
#include "asm/assembler.hpp"

#ifdef TARGET_ARCH_x86
# include "macroAssembler_x86.hpp"
#endif
#ifdef TARGET_ARCH_sparc
# include "macroAssembler_sparc.hpp"
#endif
#ifdef TARGET_ARCH_zero
# include "assembler_zero.hpp"
#endif
#ifdef TARGET_ARCH_arm
# include "macroAssembler_arm.hpp"
#endif
#ifdef TARGET_ARCH_ppc
# include "macroAssembler_ppc.hpp"
#endif
```


MacroAssembler在Assembler的基础上增加了一些频繁使用的宏指令，即一组特定功能的汇编代码，具体实现上MacroAssembler是对Assembler的进一步封装，提供更高级更实用的方法，如将寄存器的值加上某个值的incrementq方法的实现，如下图：

```
void MacroAssembler::incrementq(Address dst, int value) {
  if (value == min_jint) { addq(dst, value); return; }
  if (value <  0) { decrementq(dst, -value); return; }
  if (value == 0) {                        ; return; }
  if (value == 1 && UseIncDec) { incq(dst) ; return; }
  /* else */      { addq(dst, value)       ; return; }
}
```

2、实现原理

以上节中generate_atomic_xchg方法用到的相关MacroAssembler方法的实现为例说明，如下：

```
address generate_atomic_xchg() {
    StubCodeMark mark(this, "StubRoutines", "atomic_xchg");
    address start = __ pc();

    __ movl(rax, c_rarg0); // Copy to eax we need a return value anyhow
    __ xchgl(rax, Address(c_rarg1, 0)); // automatic LOCK
    __ ret(0);

    return start;
  }

//StubGenerator创建时会创建一个MacroAssembler实例
MacroAssembler(CodeBuffer* code) : Assembler(code) {}

Assembler(CodeBuffer* code) : AbstractAssembler(code) {}

AbstractAssembler::AbstractAssembler(CodeBuffer* code) {
  if (code == NULL)  return;
  //获取CodeBuffer中的inst 指令section
  CodeSection* cs = code->insts();
  //将原有的mark清楚
  cs->clear_mark();   // new assembler kills old mark
  //如果inst section未初始化
  if (cs->start() == NULL)  {
    vm_exit_out_of_memory(0, OOM_MMAP_ERROR, err_msg("CodeCache: no room for %s",
                                  code->name()));
  }
  //设置对应属性
```

```
  _code_section = cs;
  _oop_recorder= code->oop_recorder();
  DEBUG_ONLY( _short_branch_delta = 0; )
}

void    clear_mark()              { _mark = NULL; }

//StubCodeMark会为准备生成的stub创建一个新的StubCodeDesc，此过程会调用pc方法获取MacroAssembler准备写入指令的地址
//即stub的起始地址
 address       pc()           const   { return code_section()->end();   }

 void Assembler::movl(Register dst, Register src) {
  int encode = prefix_and_encode(dst->encoding(), src->encoding());
  //0x8B表示汇编指令movl
  emit_int8((unsigned char)0x8B);
  //movl的操作数
  emit_int8((unsigned char)(0xC0 | encode));
}

int   encoding() const                       { assert(is_valid(), "invalid register"); return (intptr_t)this; }

void emit_int8(   int8_t  x) { code_section()->emit_int8(   x); }

//在end处写入特定值，然后移动end
void emit_int8 ( int8_t  x)  { *((int8_t*)  end()) = x; set_end(end() + sizeof(int8_t)); }

void Assembler::xchgl(Register dst, Address src) { // xchg
  InstructionMark im(this);
  prefix(src, dst);
  emit_int8((unsigned char)0x87);
  emit_operand(dst, src);
}

void Assembler::ret(int imm16) {
  if (imm16 == 0) {
    emit_int8((unsigned char)0xC3);
  } else {
    emit_int8((unsigned char)0xC2);
    emit_int16(imm16);
  }
}

//StubCodeMark销毁时调用的flush方法，MacroAssembler使用AbstractAssembler的实现
void AbstractAssembler::flush() {
  //ICache提供了汇编指令缓存的相关功能，这里实际是刷新指令缓存
  ICache::invalidate_range(addr_at(0), offset());
}

address addr_at(int pos) const { return code_section()->start() + pos; }

int           offset()       const   { return code_section()->size();  }
```

六、ICache

1、ICache

ICache继承自AbstractICache，表示一个用来更新处理器的汇编指令缓存的接口，任何时候JVM修改了汇编指令，就必须刷新指令缓存。AbstractICache定义在hotspot src/share/vm/runtime/icache.hpp中，其定义的属性只有一个，如下图：

```
class AbstractICache : AllStatic {
 public:
  // The flush stub signature
  typedef int (*flush_icache_stub_t)(address addr, int lines, int magic);

 protected:
  // The flush stub function address
  static flush_icache_stub_t _flush_icache_stub;
```

_flush_icache_stub表示执行指令缓存刷新的stub。_flush_icache_stub的第一个参数addr表示指令缓存的起始地址，第二个参数lines表示刷新的缓存行的行数，第三个参数magic，用来将其作为调用结果，来表示stub正确的执行了。因为_flush_icache_stub的特殊性，所以他必须在其他的stub生成前先生成，而且第一次调用_flush_icache_stub也是刷新_flush_icache_stub自身，因为stub还不存在所以第一次调用时不会真实的刷新。

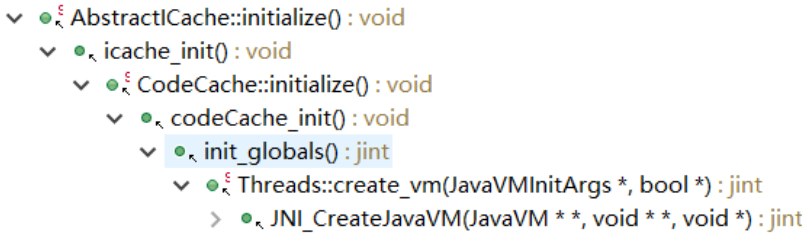
AbstractICache定义的方法只有三个，其中initialize用来初始化属性_flush_icache_stub，另外两个方法是不同场景下用来刷新指令缓存的，invalidate_word和invalidate_range。

其中initialize和invalidate_range的实现源码说明如下：

```
void AbstractICache::initialize() {
  //当此方法结束回调ResourceMark的析构函数时，通过析构函数最终调用_flush_icache_stub
  //从而实现处理器的指令缓存刷新，这是第一次使用
  ResourceMark rm;

  BufferBlob* b = BufferBlob::create("flush_icache_stub", ICache::stub_size);
  CodeBuffer c(b);

  ICacheStubGenerator g(&c);
  //通过ICacheStubGenerator生成_flush_icache_stub
  g.generate_icache_flush(&_flush_icache_stub);
}

void AbstractICache::invalidate_range(address start, int nbytes) {
  static bool firstTime = true;
  if (firstTime) {
    //第一次调用时，start就等于_flush_icache_stub
    guarantee(start == CAST_FROM_FN_PTR(address, _flush_icache_stub),
              "first flush should be for flush stub");
    firstTime = false;
    return;
  }
  if (nbytes == 0) {
    return;
  }
  //计算缓存行的位置
  const uint line_offset = mask_address_bits(start, ICache::line_size-1);
  if (line_offset != 0) {
    start -= line_offset;
    nbytes += line_offset;
  }
  call_flush_stub(start, round_to(nbytes, ICache::line_size) >>
                         ICache::log2_line_size);
}


void AbstractICache::call_flush_stub(address start, int lines) {
  //生成_flush_icache_stub时不能调用它自己
  static int magic = 0xbaadbabe;

  int auto_magic = magic; // Make a local copy to avoid race condition
```

```
  int r = (*_flush_icache_stub)(start, lines, auto_magic);

//如果返回值等于auto_magic，即第三个参数，说明flush stub未执行
guarantee(r == auto_magic, "flush stub routine did not execute");    ++magic;
}
```

其中initialize方法的调用链如下：



在stubRoutines_init1方法执行前调用，即在生成其他的stub前先生成该stub。

```
jint init_globals() {
  HandleMark hm;
  management_init();
  bytecodes_init();
  classLoader_init();
  codeCache_init();
  VM_Version_init();
  os_init_globals();
  stubRoutines_init1();
  jint status = universe_init();  // depe
                                  // stub
```

ICache的实现跟特定的CPU架构有关，通过icache.hpp中的宏定义引入，如下图：

```
#ifdef TARGET_ARCH_x86
# include "icache_x86.hpp"
#endif
#ifdef TARGET_ARCH_sparc
# include "icache_sparc.hpp"
#endif
#ifdef TARGET_ARCH_zero
# include "icache_zero.hpp"
#endif
#ifdef TARGET_ARCH_arm
# include "icache_arm.hpp"
#endif
#ifdef TARGET_ARCH_ppc
# include "icache_ppc.hpp"
#endif
```

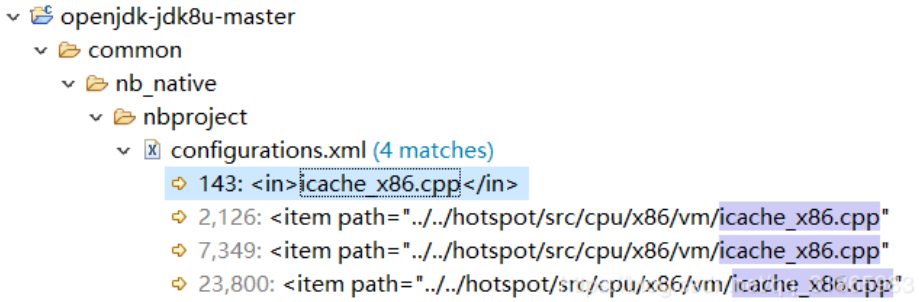
x86下的ICache未添加新的方法和属性，就重新定义了枚举，如下图：

```
class ICache : public AbstractICache {
 public:
#ifdef AMD64
  enum {
    stub_size      = 64, // Size of the icache flush stub in bytes
    line_size      = 64, // Icache line size in bytes
    log2_line_size = 6   // log2(line_size)
  };

  // Use default implementation
#else
  enum {
    stub_size      = 16,               // Size of the icache flush stub in bytes
    line_size      = BytesPerWord,     // conservative
    log2_line_size = LogBytesPerWord   // log2(line_size)
  };
#endif // AMD64
};
```


2、ICacheStubGenerator

ICacheStubGenerator继承自StubCodeGenerator，用来生成ICache中刷新指令缓存的stub，其定义在icache.hpp中，其定义的方法只有一个generate_icache_flush。

ICacheStubGenerator的实现跟特定CPU架构有关，通过configurations.xml指定，如下图：



icache_x86.cpp中的实现的核心就是mfence指令和clflush指令，参考：《clflush指令》和《SFENCE、LFENCE、MFENCE指令》。