https://www.jianshu.com/p/38286d9859b4

Interpreter In Hotspot (Base OpenJDK 8)

feihui

在阅读这片文章之前，请各位看官可以先思考一个问题，字节码在 Hotspot 中是编译执行还是解释执行？又或者是？

上面这个问题的答案可以通过在命令行执行 java -version 命令得到，得到如下输出：

```
openjdk version "1.8.0_222"
OpenJDK Runtime Environment (AdoptOpenJDK)(build 1.8.0_222-b10)
OpenJDK 64-Bit Server VM (AdoptOpenJDK)(build 25.222-b10, mixed mode)
```

关注最后一行最后的 mixed mode -- 解释执行和编译执行相结合的混合模式 (你可以通过关闭 JIT 来达到完全解释执行的效果，你也可以通过开启 AOT 来达到完全编译执行的效果)。

今天就先来讲下 Hotspot 与解释执行相关的解释器。

在 OpenJDK 源码(base 8 version 实现)中，可以搜索到两个 Interpreter：

BytecodeInterpreter.cpp

TemplateInterpreter.cpp

BytecodeInterperter：作为最早的 Interpreter，实现方式非常简单明了，但其执行效率也同样非常"简(bu)单(ren)明(zhi)了(shi)”，以 new 为例看如下代码：

```
    switch (opcode) {
      ......
      CASE(_new): {
        u2 index = Bytes::get_Java_u2(pc+1);
        ConstantPool* constants = istate->method()->constants();
        if (!constants->tag_at(index).is_unresolved_klass()) {
          // Make sure klass is initialized and doesn't have a finalizer
          Klass* entry = constants->slot_at(index).get_klass();
          assert(entry->is_klass(), "Should be resolved klass");
          Klass* k_entry = (Klass*) entry;
          assert(k_entry->oop_is_instance(), "Should be InstanceKlass");
          InstanceKlass* ik = (InstanceKlass*) k_entry;
          if ( ik->is_initialized() && ik->can_be_fastpath_allocated() ) {
            size_t obj_size = ik->size_helper();
            oop result = NULL;
            // If the TLAB isn't pre-zeroed then we'll have to do it
            bool need_zero = !ZeroTLAB;
            if (UseTLAB) {
              result = (oop) THREAD->tlab().allocate(obj_size);
            }
            if (result == NULL) {
              need_zero = true;
              // Try allocate in shared eden
        retry:
              HeapWord* compare_to = *Universe::heap()->top_addr();
              HeapWord* new_top = compare_to + obj_size;
              if (new_top <= *Universe::heap()->end_addr()) {
                if (Atomic::cmpxchg_ptr(new_top, Universe::heap()->top_addr(), compare_to) != compare_to) {
                  goto retry;
                }
                result = (oop) compare_to;
              }
            }
            if (result != NULL) {
              // Initialize object (if nonzero size and need) and then the header
              if (need_zero ) {
                HeapWord* to_zero = (HeapWord*) result + sizeof(oopDesc) / oopSize;
                obj_size -= sizeof(oopDesc) / oopSize;
                if (obj_size > 0 ) {
                  memset(to_zero, 0, obj_size * HeapWordSize);
                }
              }
              if (UseBiasedLocking) {
                result->set_mark(ik->prototype_header());
              } else {
                result->set_mark(markOopDesc::prototype());
              }
              result->set_klass_gap(0);
              result->set_klass(k_entry);
              SET_STACK_OBJECT(result, 0);
              UPDATE_PC_AND_TOS_AND_CONTINUE(3, 1);
            }
          }
        }
        // Slow case allocation
        CALL_VM(InterpreterRuntime::_new(THREAD, METHOD->constants(), index),
                handle_exception);
        SET_STACK_OBJECT(THREAD->vm_result(), 0);
        THREAD->set_vm_result(NULL);
        UPDATE_PC_AND_TOS_AND_CONTINUE(3, 1);
      }
    ......
```

```
    }
```

可以看到创建一个对象对应如此多的代码(还不包括其中的宏展开以及方法调用)，这效率也能想象得到，之所以一直保留在源码中，主要用于方便理解每个字节码的逻辑。

TemplateInterpreter：当前默认的 Interpreter，每一个字节码会有一段对应的精简汇编代码，同样以 new 为例看如下代码：

```
void TemplateTable::initialize() {
  ......
  // Java spec bytecodes                ubcp|disp|clvm|iswd  in    out   generator             argument
  def(Bytecodes::_new                 , ubcp|____|clvm|____, vtos, atos, _new                ,  _           );
  ......
}

void TemplateTable::_new() {
  transition(vtos, atos);
  __ get_unsigned_2_byte_index_at_bcp(rdx, 1);
  Label slow_case;
  Label done;
  Label initialize_header;
  Label initialize_object; // including clearing the fields
  Label allocate_shared;

  __ get_cpool_and_tags(rsi, rax);
  // Make sure the class we're about to instantiate has been resolved.
  // This is done before loading InstanceKlass to be consistent with the order
  // how Constant Pool is updated (see ConstantPool::klass_at_put)
  const int tags_offset = Array<u1>::base_offset_in_bytes();
  __ cmpb(Address(rax, rdx, Address::times_1, tags_offset),
          JVM_CONSTANT_Class);
  __ jcc(Assembler::notEqual, slow_case);

  // get InstanceKlass
  __ movptr(rsi, Address(rsi, rdx,
            Address::times_8, sizeof(ConstantPool)));

  // make sure klass is initialized & doesn't have finalizer
  // make sure klass is fully initialized
  __ cmpb(Address(rsi,
                  InstanceKlass::init_state_offset()),
          InstanceKlass::fully_initialized);
  __ jcc(Assembler::notEqual, slow_case);

  // get instance_size in InstanceKlass (scaled to a count of bytes)
  __ movl(rdx,
          Address(rsi,
                  Klass::layout_helper_offset()));
  // test to see if it has a finalizer or is malformed in some way
  __ testl(rdx, Klass::_lh_instance_slow_path_bit);
  __ jcc(Assembler::notZero, slow_case);

  // Allocate the instance
  // 1) Try to allocate in the TLAB
  // 2) if fail and the object is large allocate in the shared Eden
  // 3) if the above fails (or is not applicable), go to a slow case
  // (creates a new TLAB, etc.)

  const bool allow_shared_alloc =
    Universe::heap()->supports_inline_contig_alloc() && !CMSIncrementalMode;

  if (UseTLAB) {
    __ movptr(rax, Address(r15_thread, in_bytes(JavaThread::tlab_top_offset())));
    __ lea(rbx, Address(rax, rdx, Address::times_1));
    __ cmpptr(rbx, Address(r15_thread, in_bytes(JavaThread::tlab_end_offset())));
    __ jcc(Assembler::above, allow_shared_alloc ? allocate_shared : slow_case);
    __ movptr(Address(r15_thread, in_bytes(JavaThread::tlab_top_offset())), rbx);
    if (ZeroTLAB) {
      // the fields have been already cleared
      __ jmp(initialize_header);
    } else {
      // initialize both the header and fields
      __ jmp(initialize_object);
    }
  }

  // Allocation in the shared Eden, if allowed.
  //
  // rdx: instance size in bytes
  if (allow_shared_alloc) {
    __ bind(allocate_shared);

    ExternalAddress top((address)Universe::heap()->top_addr());
    ExternalAddress end((address)Universe::heap()->end_addr());

    const Register RtopAddr = rscratch1;
    const Register RendAddr = rscratch2;

```

```
    __ lea(RtopAddr, top);
    __ lea(RendAddr, end);
    __ movptr(rax, Address(RtopAddr, 0));

    // For retries rax gets set by cmpxchgq
    Label retry;
    __ bind(retry);
    __ lea(rbx, Address(rax, rdx, Address::times_1));
    __ cmpptr(rbx, Address(RendAddr, 0));
    __ jcc(Assembler::above, slow_case);

    // Compare rax with the top addr, and if still equal, store the new
    // top addr in rbx at the address of the top addr pointer. Sets ZF if was
    // equal, and clears it otherwise. Use lock prefix for atomicity on MPs.
    //
    // rax: object begin
    // rbx: object end
    // rdx: instance size in bytes
    if (os::is_MP()) {
      __ lock();
    }
    __ cmpxchgptr(rbx, Address(RtopAddr, 0));

    // if someone beat us on the allocation, try again, otherwise continue
    __ jcc(Assembler::notEqual, retry);

    __ incr_allocated_bytes(r15_thread, rdx, 0);
  }

  if (UseTLAB || Universe::heap()->supports_inline_contig_alloc()) {
    // The object is initialized before the header.  If the object size is
    // zero, go directly to the header initialization.
    __ bind(initialize_object);
    __ decrementl(rdx, sizeof(oopDesc));
    __ jcc(Assembler::zero, initialize_header);

    // Initialize object fields
    __ xorl(rcx, rcx); // use zero reg to clear memory (shorter code)
    __ shrl(rdx, LogBytesPerLong);  // divide by oopSize to simplify the loop
    {
      Label loop;
      __ bind(loop);
      __ movq(Address(rax, rdx, Address::times_8,
                      sizeof(oopDesc) - oopSize),
              rcx);
      __ decrementl(rdx);
      __ jcc(Assembler::notZero, loop);
    }

    // initialize object header only.
    __ bind(initialize_header);
    if (UseBiasedLocking) {
      __ movptr(rscratch1, Address(rsi, Klass::prototype_header_offset()));
      __ movptr(Address(rax, oopDesc::mark_offset_in_bytes()), rscratch1);
    } else {
      __ movptr(Address(rax, oopDesc::mark_offset_in_bytes()),
               (intptr_t) markOopDesc::prototype()); // header (address 0x1)
    }
    __ xorl(rcx, rcx); // use zero reg to clear memory (shorter code)
    __ store_klass_gap(rax, rcx);  // zero klass gap for compressed oops
    __ store_klass(rax, rsi);      // store klass last

    {
      SkipIfEqual skip(_masm, &DTraceAllocProbes, false);
      // Trigger dtrace event for fastpath
      __ push(atos); // save the return value
      __ call_VM_leaf(
           CAST_FROM_FN_PTR(address, SharedRuntime::dtrace_object_alloc), rax);
      __ pop(atos); // restore the return value

    }
    __ jmp(done);
  }


  // slow case
  __ bind(slow_case);
  __ get_constant_pool(c_rarg1);
  __ get_unsigned_2_byte_index_at_bcp(c_rarg2, 1);
  call_VM(rax, CAST_FROM_FN_PTR(address, InterpreterRuntime::_new), c_rarg1, c_rarg2);
  __ verify_oop(rax);

  // continue
  __ bind(done);
}
```

上面展示的是生成汇编代码的代码，下面则展示的 Hotspot 初始化后生成的汇编代码：

```
new  187 new  [0x000000011828f8c0, 0x000000011828fac0]  512 bytes

  0x000000011828f8c0: push   %rax
  0x000000011828f8c1: jmpq   0x000000011828f8f0
  0x000000011828f8c6: sub    $0x8,%rsp
  0x000000011828f8ca: vmovss %xmm0,(%rsp)
  0x000000011828f8cf: jmpq   0x000000011828f8f0
  0x000000011828f8d4: sub    $0x10,%rsp
  0x000000011828f8d8: vmovsd %xmm0,(%rsp)
  0x000000011828f8dd: jmpq   0x000000011828f8f0
  0x000000011828f8e2: sub    $0x10,%rsp
  0x000000011828f8e6: mov    %rax,(%rsp)
  0x000000011828f8ea: jmpq   0x000000011828f8f0
  0x000000011828f8ef: push   %rax
  0x000000011828f8f0: movzwl 0x1(%r13),%edx
  0x000000011828f8f5: bswap  %edx
  0x000000011828f8f7: shr    $0x10,%edx
  0x000000011828f8fa: mov    -0x18(%rbp),%rsi
  0x000000011828f8fe: mov    0x8(%rsi),%rsi
  0x000000011828f902: mov    0x8(%rsi),%rsi
  0x000000011828f906: mov    0x8(%rsi),%rax
  0x000000011828f90a: cmpb   $0x7,0x4(%rax,%rdx,1)
  0x000000011828f90f: jne    0x000000011828f9e4
  0x000000011828f915: mov    0x50(%rsi,%rdx,8),%rsi
  0x000000011828f91a: cmpb   $0x4,0x162(%rsi)
  0x000000011828f921: jne    0x000000011828f9e4
  0x000000011828f927: mov    0x8(%rsi),%edx
  0x000000011828f92a: test   $0x1,%edx
  0x000000011828f930: jne    0x000000011828f9e4
  0x000000011828f936: mov    0x60(%r15),%rax
  0x000000011828f93a: lea    (%rax,%rdx,1),%rbx
  0x000000011828f93e: cmp    0x70(%r15),%rbx
  0x000000011828f942: ja     0x000000011828f951
  0x000000011828f948: mov    %rbx,0x60(%r15)
  0x000000011828f94c: jmpq   0x000000011828f983
  0x000000011828f951: movabs $0x7fa4fd6009c8,%r10
  0x000000011828f95b: movabs $0x7fa4fd6009a0,%r11
  0x000000011828f965: mov    (%r10),%rax
  0x000000011828f968: lea    (%rax,%rdx,1),%rbx
  0x000000011828f96c: cmp    (%r11),%rbx
  0x000000011828f96f: ja     0x000000011828f9e4
  0x000000011828f975: lock cmpxchg %rbx,(%r10)
  0x000000011828f97a: jne    0x000000011828f968
  0x000000011828f97c: add    %rdx,0xb8(%r15)
  0x000000011828f983: sub    $0x10,%edx
  0x000000011828f986: je     0x000000011828f99a
  0x000000011828f98c: xor    %ecx,%ecx
  0x000000011828f98e: shr    $0x3,%edx
  0x000000011828f991: mov    %rcx,0x8(%rax,%rdx,8)
  0x000000011828f996: dec    %edx
  0x000000011828f998: jne    0x000000011828f991
  0x000000011828f99a: mov    0xa8(%rsi),%r10
  0x000000011828f9a1: mov    %r10,(%rax)
  0x000000011828f9a4: xor    %ecx,%ecx
  0x000000011828f9a6: mov    %rsi,0x8(%rax)
  0x000000011828f9aa: cmpb   $0x0,-0x87f88f6(%rip)        # 0x000000010fa970bb
  0x000000011828f9b1: je     0x000000011828f9df
  0x000000011828f9b7: push   %rax
  0x000000011828f9b8: mov    %rax,%rdi
  0x000000011828f9bb: test   $0xf,%esp
  0x000000011828f9c1: je     0x000000011828f9d9
  0x000000011828f9c7: sub    $0x8,%rsp
  0x000000011828f9cb: callq  0x000000010f6c06c4
  0x000000011828f9d0: add    $0x8,%rsp
  0x000000011828f9d4: jmpq   0x000000011828f9de
  0x000000011828f9d9: callq  0x000000010f6c06c4
  0x000000011828f9de: pop    %rax
  0x000000011828f9df: jmpq   0x000000011828faa8
  0x000000011828f9e4: mov    -0x18(%rbp),%rsi
  0x000000011828f9e8: mov    0x8(%rsi),%rsi
  0x000000011828f9ec: mov    0x8(%rsi),%rsi
  0x000000011828f9f0: movzwl 0x1(%r13),%edx
  0x000000011828f9f5: bswap  %edx
  0x000000011828f9f7: shr    $0x10,%edx
  0x000000011828f9fa: callq  0x000000011828fa04
  0x000000011828f9ff: jmpq   0x000000011828faa8
  0x000000011828fa04: lea    0x8(%rsp),%rax
  0x000000011828fa09: mov    %r13,-0x38(%rbp)
  0x000000011828fa0d: mov    %r15,%rdi
  0x000000011828fa10: mov    %rbp,0x1d0(%r15)
  0x000000011828fa17: mov    %rax,0x1c0(%r15)
  0x000000011828fa1e: test   $0xf,%esp
  0x000000011828fa24: je     0x000000011828fa3c
  0x000000011828fa2a: sub    $0x8,%rsp
  0x000000011828fa2e: callq  0x000000010f4def58
```

```
  0x000000011828fa33: add    $0x8,%rsp
  0x000000011828fa37: jmpq   0x000000011828fa41
  0x000000011828fa3c: callq  0x000000010f4def58
  0x000000011828fa41: movabs $0x0,%r10
  0x000000011828fa4b: mov    %r10,0x1c0(%r15)
  0x000000011828fa52: movabs $0x0,%r10
  0x000000011828fa5c: mov    %r10,0x1d0(%r15)
  0x000000011828fa63: movabs $0x0,%r10
  0x000000011828fa6d: mov    %r10,0x1c8(%r15)
  0x000000011828fa74: cmpq   $0x0,0x8(%r15)
  0x000000011828fa7c: je     0x000000011828fa87
  0x000000011828fa82: jmpq   0x000000011826a420
  0x000000011828fa87: mov    0x220(%r15),%rax
  0x000000011828fa8e: movabs $0x0,%r10
  0x000000011828fa98: mov    %r10,0x220(%r15)
  0x000000011828fa9f: mov    -0x38(%rbp),%r13
  0x000000011828faa3: mov    -0x30(%rbp),%r14
  0x000000011828faa7: retq
  0x000000011828faa8: movzbl 0x3(%r13),%ebx
  0x000000011828faad: add    $0x3,%r13
  0x000000011828fab1: movabs $0x10faab9e0,%r10
  0x000000011828fabb: jmpq   *(%r10,%rbx,8)
  0x000000011828fabf: nop
```

从上面的代码我们可以看出，相对于 BytecodeInterpreter 的实现，TemplateInterpreter 的实现精简了很多。

如何打印字节码对应的汇编代码？示例如下：

```
javac -g Solution.java
java -XX:-UseCompressedOops -XX:+UnlockDiagnosticVMOptions -XX:+PrintStubCode ->XX:+PrintInterpreter -XX:+PrintAssembly Solution
// Could not load hsdis-amd64.dylib; library not loadable; PrintAssembly is disabled
// mv hsdis-amd64.dylib > /Library/Java/JavaVirtualMachines/jdk1.8.0_221.jdk/Contents/Home/jre/lib
```

从 BytecodeInterpreter 到 TemplateInterpreter，性能有了很大的提升(当然提升的方式不仅于此)，可能你会觉得手工编写生成汇编的代码维护成本过高(实际上就相当于用汇编编写逻辑，想想都觉得阔怕，曾经也尝试过自动生成，但效果不尽人意)，但从这方面去想 -- 本身每个字节码的逻辑基本上是固定不变的，那么维护成本看起来也并不是不可取，况且伊始更关注的执行效率。

Hotspot 为了提升执行效率，除了 Interpreter 的演进，另一个重要的技术则是 JIT -- 通过 profiling 实现运行时优化，这种方法带来的优化使得 JVM 语言在一些场合下可以优于编译执行语言。除了上述两种主要优化，还有诸如 常量替换/循环展开/同步消除/栈上分配/方法内联 等一些列策略性优化。(或许你会问为什么没有提到垃圾收集，就个人而言， 我觉得垃圾收集的演进更多的匹配业务场景，例如在后台业务中 Parallel GC 会有更好的吞吐，而在交互业务中 CMS 会有更好的停顿，并不存在一种 GC 适用于所有业务场景)

我们再来简单聊聊 栈顶缓存 -- 将当前操作数据优先存放在栈顶缓存上(寄存器) 而非实际栈顶上 (内存)，我们先通过一个简单例子来了解下。

```
 int m = a + b
 // 未使用栈顶缓存，需要七次数据移动
 a local areas(memory) -> register
 a register -> stack top(memory)
 b local areas(memory) -> register
 b register -> stack top(memory)
 b stack top(memory) -> register
 a stack top(memory) -> register
 m register -> stack top(memory)

 // 使用栈顶缓存，只需要四次数据移动
 a local areas(memory) -> register
 a register -> stack top(memory)
 b local areas(memory) -> register
 a stack top(memory) -> register
```

从上面的例子我们可以看到，使用栈顶缓存能明显减少一些数据移动。我们以 iconst_0 为例，我们先看来下其模板定义

```
  def(Bytecodes::_iconst_0            , ____|____|____|____, vtos, itos, iconst              ,  0           );
```

上面是 iconst_0 的模板定义，vtos 表示这个字节码执行之前栈顶缓存需为空，itos 表示这个字节码执行之后栈顶缓存为int，我们再来看下生成汇编码部分的代码

```
void TemplateInterpreterGenerator::set_short_entry_points(Template* t, address& bep, address& cep, address& sep, address& aep, address& iep, address& lep, address& fep, address& dep, address& vep) {
  assert(t->is_valid(), "template must exist");
  switch (t->tos_in()) {
    case btos:
    case ctos:
    case stos:
      ShouldNotReachHere();  // btos/ctos/stos should use itos.
      break;
    case atos: vep = __ pc(); __ pop(atos); aep = __ pc(); generate_and_dispatch(t); break;
    case itos: vep = __ pc(); __ pop(itos); iep = __ pc(); generate_and_dispatch(t); break;
    case ltos: vep = __ pc(); __ pop(ltos); lep = __ pc(); generate_and_dispatch(t); break;
    case ftos: vep = __ pc(); __ pop(ftos); fep = __ pc(); generate_and_dispatch(t); break;
    case dtos: vep = __ pc(); __ pop(dtos); dep = __ pc(); generate_and_dispatch(t); break;
    case vtos: set_vtos_entry_points(t, bep, cep, sep, aep, iep, lep, fep, dep, vep);     break;
    default  : ShouldNotReachHere();                                                 break;
  }
}

void TemplateInterpreterGenerator::set_vtos_entry_points(Template* t,
                                                         address& bep,
                                                         address& cep,
                                                         address& sep,
```

```
                                                          address& aep,
                                                          address& iep,
                                                          address& lep,
                                                          address& fep,
                                                          address& dep,
                                                          address& vep) {
  assert(t->is_valid() && t->tos_in() == vtos, "illegal template");
  Label L;
  aep = __ pc();  __ push_ptr();  __ jmp(L);
  fep = __ pc();  __ push_f();    __ jmp(L);
  dep = __ pc();  __ push_d();    __ jmp(L);
  lep = __ pc();  __ push_l();    __ jmp(L);
  bep = cep = sep =
  iep = __ pc();  __ push_i();
  vep = __ pc();
  __ bind(L);
  generate_and_dispatch(t);
}
```

上面可以看到，当需要栈顶缓存为空时，会首先将缓存数据放入实际栈顶，然后再执行字节码逻辑，如下图汇编代码所示：

```
0x00000001184968c0: push   %rax
0x00000001184968c1: jmpq   0x00000001184968f0
0x00000001184968c6: sub    $0x8,%rsp
0x00000001184968ca: vmovss %xmm0,(%rsp)
0x00000001184968cf: jmpq   0x00000001184968f0
0x00000001184968d4: sub    $0x10,%rsp
0x00000001184968d8: vmovsd %xmm0,(%rsp)
0x00000001184968dd: jmpq   0x00000001184968f0
0x00000001184968e2: sub    $0x10,%rsp
0x00000001184968e6: mov    %rax,(%rsp)
0x00000001184968ea: jmpq   0x00000001184968f0
0x00000001184968ef: push   %rax
0x00000001184968f0: xor    %eax,%eax
0x00000001184968f2: movzbl 0x1(%r13),%ebx
0x00000001184968f7: inc    %r13
0x00000001184968fa: movabs $0x10c309160,%r10
0x0000000118496904: jmpq   *(%r10,%rbx,8)
```

可以看到 0x00000001184968f0 为 iconst_0 汇编码起始地址，以上的为处理栈顶缓存和跳转逻辑。

最后再来简单聊下 Profiling，Hotspot JIT-compiler 之所以强大的一点，不仅仅因为它能将字节码编辑成机器码 (单纯地将字节码编译成机器码并不见得比编译执行语言高效)，更重要的结合 运行时 Profiling 数据，能将热点路径以最优性能执行 -- 例如：对于一段比较复杂的处理逻辑，有：输入 A 输出 B，而且大部分情况输入都为 A，那么可以直接改优化为比较输入是否为 A，若为 A 则直接输出 B 而不需要执行复杂逻辑，类似于哈夫曼编码。更多详情可阅读郑雨迪博士的专栏《深入拆解Java虚拟机》，专栏干货满满。