

09 | JVM是怎么实现invokedynamic的？（下）

2018-08-10 郑雨迪



09 | JVM是怎么实现invokedynamic的？（下）

朗读人：郑雨迪 11'38" | 5.34M

上回讲到，为了让所有的动物都能参加赛马，Java 7 引入了 `invokedynamic` 机制，允许调用任意类的“赛跑”方法。不过，我们并没有讲解 `invokedynamic`，而是深入地探讨了它所依赖的方法句柄。

今天，我便来正式地介绍 `invokedynamic` 指令，讲讲它是如何生成调用点，并且允许应用程序自己决定链接至哪一个方法中的。

`invokedynamic` 指令

`invokedynamic` 是 Java 7 引入的一条新指令，用以支持动态语言的方法调用。具体来说，它将调用点（`CallSite`）抽象成一个 Java 类，并且将原本由 Java 虚拟机控制的方法调用以及方法链接暴露给了应用程序。在运行过程中，每一条 `invokedynamic` 指令将捆绑一个调用点，并且会调用该调用点所链接的方法句柄。

在第一次执行 `invokedynamic` 指令时，Java 虚拟机会调用该指令所对应的启动方法（`Bootstrap Method`），来生成前面提到的调用点，并且将之绑定至该 `invokedynamic` 指令中。在之后的运行过程中，Java 虚拟机则会直接调用绑定的调用点所链接的方法句柄。

在字节码中，启动方法是用方法句柄来指定的。这个方法句柄指向一个返回类型为调用点的静态方法。该方法必须接收三个固定的参数，分别为一个 Lookup 类实例，一个用来指代目标方法名字的字符串，以及该调用点能够链接的方法句柄的类型。

除了这三个必需参数之外，启动方法还可以接收若干个其他的参数，用来辅助生成调用点，或者定位所要链接的目标方法。

```
import java.lang.invoke.*;

class Horse {
    public void race() {
        System.out.println("Horse.race()");
    }
}

class Deer {
    public void race() {
        System.out.println("Deer.race()");
    }
}

// javac Circuit.java
// java Circuit
public class Circuit {

    public static void startRace(Object obj) {
        // aload obj
        // invokedynamic race()
    }

    public static void main(String[] args) {
        startRace(new Horse());
        // startRace(new Deer());
    }

    public static CallSite bootstrap(MethodHandles.Lookup l, String name, MethodType callSiteTy
        MethodHandle mh = l.findVirtual(Horse.class, name, MethodType.methodType(void.class));
```

```
        return new ConstantCallSite(mh.asType(callSiteType));
    }
}
```

我在文稿中贴了一段代码，其中便包含一个启动方法。它将接收前面提到的三个固定参数，并且返回一个链接至 `Horse.race` 方法的 `ConstantCallSite`。

这里的 `ConstantCallSite` 是一种不可以更改链接对象的调用点。除此之外，Java 核心类库还提供多种可以更改链接对象的调用点，比如 `MutableCallSite` 和 `VolatileCallSite`。

这两者的区别就好比正常字段和 `volatile` 字段之间的区别。此外，应用程序还可以自定义调用点类，来满足特定的重链接需求。

由于 Java 暂不支持直接生成 `invokedynamic` 指令 [1]，所以接下来我会借助之前介绍过的字节码工具 ASM 来实现这一目的。

```
import java.io.IOException;
import java.lang.invoke.*;
import java.nio.file.*;

import org.objectweb.asm.*;

// javac -cp /path/to/asm-all-6.0_BETA.jar:. ASMHelper.java
// java -cp /path/to/asm-all-6.0_BETA.jar:. ASMHelper
// java Circuit
public class ASMHelper implements Opcodes {

    private static class MyMethodVisitor extends MethodVisitor {

        private static final String BOOTSTRAP_CLASS_NAME = Circuit.class.getName().replace('.', ' ');
        private static final String BOOTSTRAP_METHOD_NAME = "bootstrap";
        private static final String BOOTSTRAP_METHOD_DESC = MethodType
            .methodType(CallSite.class, MethodHandles.Lookup.class, String.class, MethodType.class)
            .toMethodDescriptorString();

        private static final String TARGET_METHOD_NAME = "race";
        private static final String TARGET_METHOD_DESC = "(Ljava/lang/Object;)V";
```

```
public final MethodVisitor mv;

public MyMethodVisitor(int api, MethodVisitor mv) {
    super(api);
    this.mv = mv;
}

@Override
public void visitCode() {
    mv.visitCode();
    mv.visitVarInsn(ALOAD, 0);
    Handle h = new Handle(H_INVOKESTATIC, BOOTSTRAP_CLASS_NAME, BOOTSTRAP_METHOD_NAME, BOOT
    mv.visitInvokeDynamicInsn(TARGET_METHOD_NAME, TARGET_METHOD_DESC, h);
    mv.visitInsn(RETURN);
    mv.visitMaxs(1, 1);
    mv.visitEnd();
}
}

public static void main(String[] args) throws IOException {
    ClassReader cr = new ClassReader("Circuit");
    ClassWriter cw = new ClassWriter(cr, ClassWriter.COMPUTE_FRAMES);
    ClassVisitor cv = new ClassVisitor(ASM6, cw) {
        @Override
        public MethodVisitor visitMethod(int access, String name, String descriptor, String sig
            String[] exceptions) {
            MethodVisitor visitor = super.visitMethod(access, name, descriptor, signature, except
            if ("startRace".equals(name)) {
                return new MyMethodVisitor(ASM6, visitor);
            }
            return visitor;
        }
    };
    cr.accept(cv, ClassReader.SKIP_FRAMES);

    Files.write(Paths.get("Circuit.class"), cw.toByteArray());
}
```

```
}  
}
```

你无需理解上面这段代码的具体含义，只须了解它会更改同一目录下 `Circuit` 类的 `startRace(Object)` 方法，使之包含 `invokedynamic` 指令，执行所谓的赛跑方法。

```
public static void startRace(java.lang.Object);  
    0: aload_0  
    1: invokedynamic #80, 0 // race:(Ljava/lang/Object;)V  
    6: return
```

如果你足够细心的话，你会发现该指令所调用的赛跑方法的描述符，和 `Horse.race` 方法或者 `Deer.race` 方法的描述符并不一致。这是因为 `invokedynamic` 指令最终调用的是方法句柄，而方法句柄会将调用者当成第一个参数。因此，刚刚提到的那两个方法恰恰符合这个描述符所对应的方法句柄类型。

到目前为止，我们已经可以通过 `invokedynamic` 调用 `Horse.race` 方法了。为了支持调用任意类的 `race` 方法，我实现了一个简单的单态内联缓存。如果调用者的类型命中缓存中的类型，便直接调用缓存中的方法句柄，否则便更新缓存。

```
// 需要更改 ASMHelper.MyMethodVisitor 中的 BOOTSTRAP_CLASS_NAME  
import java.lang.invoke.*;  
  
public class MonomorphicInlineCache {  
  
    private final MethodHandles.Lookup lookup;  
    private final String name;  
  
    public MonomorphicInlineCache(MethodHandles.Lookup lookup, String name) {  
        this.lookup = lookup;  
        this.name = name;  
    }  
  
    private Class<?> cachedClass = null;  
    private MethodHandle mh = null;
```

```
public void invoke(Object receiver) throws Throwable {
    if (cachedClass != receiver.getClass()) {
        cachedClass = receiver.getClass();
        mh = lookup.findVirtual(cachedClass, name, MethodType.methodType(void.class));
    }
    mh.invoke(receiver);
}

public static CallSite bootstrap(MethodHandles.Lookup l, String name, MethodType callSiteTy
    MonomorphicInlineCache ic = new MonomorphicInlineCache(l, name);
    MethodHandle mh = l.findVirtual(MonomorphicInlineCache.class, "invoke", MethodType.method
}
}
```

可以看到，尽管 `invokedynamic` 指令调用的是所谓的 `race` 方法，但是实际上我返回了一个链接至名为 `"invoke"` 的方法的调用点。由于调用点仅要求方法句柄的类型能够匹配，因此这个链接是合法的。

不过，这正是 `invokedynamic` 的目的，也就是将调用点与目标方法的链接交由应用程序来做，并且依赖于应用程序对目标方法进行验证。所以，如果应用程序将赛跑方法链接至兔子的睡觉方法，那也只能怪应用程序自己了。

Java 8 的 Lambda 表达式

在 Java 8 中，Lambda 表达式也是借助 `invokedynamic` 来实现的。

具体来说，Java 编译器利用 `invokedynamic` 指令来生成实现了函数式接口的适配器。这里的函数式接口指的是仅包括一个非 `default` 接口方法的接口，一般通过 `@FunctionalInterface` 注解。不过就算是没有使用该注解，Java 编译器也会将符合条件的接口辨认为函数式接口。

```
int x = ..
IntStream.of(1, 2, 3).map(i -> i * 2).map(i -> i * x);
```

举个例子，上面这段代码会对 `IntStream` 中的元素进行两次映射。我们知道，映射方法 `map` 所接收的参数是 `IntUnaryOperator`（这是一个函数式接口）。也就是说，在运行过程中我们需要将 `i->i2` 和 `i->ix` 这两个 Lambda 表达式转化成 `IntUnaryOperator` 的实例。这个转化过程便是由 `invokedynamic` 来实现的。

在编译过程中，Java 编译器会对 Lambda 表达式进行解语法糖（desugar），生成一个方法来保存 Lambda 表达式的内容。该方法的参数列表不仅包含原本 Lambda 表达式的参数，还包含它所捕获的变量。（注：方法引用，如 `Horse::race`，则不会生成额外的方法。）

在上面那个例子中，第一个 Lambda 表达式没有捕获其他变量，而第二个 Lambda 表达式（也就是 `i->i*x`）则会捕获局部变量 `x`。这两个 Lambda 表达式对应的方法如下所示。可以看到，所捕获的变量同样也会作为参数传入生成的方法之中。

```
// i -> i * 2
private static int lambda$0(int);

Code:
    0: iload_0
    1: iconst_2
    2: imul
    3: ireturn

// i -> i * x
private static int lambda$1(int, int);

Code:
    0: iload_1
    1: iload_0
    2: imul
    3: ireturn
```

第一次执行 `invokedynamic` 指令时，它所对应的启动方法会通过 ASM 来生成一个适配器类。这个适配器类实现了对应的函数式接口，在我们的例子中，也就是 `IntUnaryOperator`。启动方法的返回值是一个 `ConstantCallSite`，其链接对象为一个返回适配器类实例的方法句柄。

根据 Lambda 表达式是否捕获其他变量，启动方法生成的适配器类以及所链接的方法句柄皆不同。

如果该 Lambda 表达式没有捕获其他变量，那么可以认为它是上下文无关的。因此，启动方法将新建一个适配器类的实例，并且生成一个特殊的方法句柄，始终返回该实例。

如果该 Lambda 表达式捕获了其他变量，那么每次执行该 `invokedynamic` 指令，我们都要更新这些捕获了的变量，以防止它们发生了变化。

另外，为了保证 Lambda 表达式的线程安全，我们无法共享同一个适配器类的实例。因此，在每次执行 `invokedynamic` 指令时，所调用的方法句柄都需要新建一个适配器类实例。

在这种情况下，启动方法生成的适配器类将包含一个额外的静态方法，来构造适配器类的实例。该方法将接收这些捕获的参数，并且将它们保存为适配器类实例的实例字段。

你可以通过虚拟机参数 `-Djdk.internal.lambda.dumpProxyClasses=/DUMP/PATH` 导出这些具体的适配器类。这里我导出了上面这个例子中两个 Lambda 表达式对应的适配器类。

```
// i->i*2 对应的适配器类
final class LambdaTest$$Lambda$1 implements IntUnaryOperator {
    private LambdaTest$$Lambda$1();

    Code:
        0: aload_0
        1: invokespecial java/lang/Object."<init>":()V
        4: return

    public int applyAsInt(int);

    Code:
        0: iload_1
        1: invokestatic LambdaTest.lambda$0:(I)I
        4: ireturn
}

// i->i*x 对应的适配器类
final class LambdaTest$$Lambda$2 implements IntUnaryOperator {
    private final int arg$1;

    private LambdaTest$$Lambda$2(int);

    Code:
        0: aload_0
        1: invokespecial java/lang/Object."<init>":()V
        4: aload_0
        5: iload_1
        6: putfield arg$1:I
        9: return

    private static java.util.function.IntUnaryOperator get$Lambda(int);

    Code:
        0: new LambdaTest$$Lambda$2
```



```
3: dup
4: iload_0
5: invokespecial "<init>":(I)V
8: areturn

public int applyAsInt(int);
Code:
0: aload_0
1: getfield arg$1:I
4: iload_1
5: invokestatic LambdaTest.lambda$1:(II)I
8: ireturn
}
```

可以看到，捕获了局部变量的 Lambda 表达式多出了一个 `get$Lambda` 的方法。启动方法便会所返回的调用点链接至指向该方法的方法句柄。也就是说，每次执行 `invokedynamic` 指令时，都会调用至这个方法中，并构造一个新的适配器类实例。

这个多出来的新建实例会对程序性能造成影响吗？

Lambda 以及方法句柄的性能分析

我再次请出测试反射调用性能开销的那段代码，并将其改造成使用 Lambda 表达式的 v6 版本。

```
// v6 版本
import java.util.function.IntConsumer;

public class Test {
    public static void target(int i) { }

    public static void main(String[] args) throws Exception {
        long current = System.currentTimeMillis();
        for (int i = 1; i <= 2_000_000_000; i++) {
            if (i % 100_000_000 == 0) {
                long temp = System.currentTimeMillis();
                System.out.println(temp - current);
                current = temp;
            }
        }
    }
}
```

```
    }

    ((IntConsumer) j -> Test.target(j)).accept(128);
    // ((IntConsumer) Test::target).accept(128);
  }
}
}
```

测量结果显示，它与直接调用的性能并无太大的区别。也就是说，即时编译器能够将转换 Lambda 表达式所使用的 `invokedynamic`，以及对 `IntConsumer.accept` 方法的调用统统内联进来，最终优化为空操作。

这个其实不难理解：Lambda 表达式所使用的 `invokedynamic` 将绑定一个 `ConstantCallSite`，其链接的目标方法无法改变。因此，即时编译器会将该目标方法直接内联进来。对于这类没有捕获变量的 Lambda 表达式而言，目标方法只完成了一个动作，便是加载缓存的适配器类常量。

另一方面，对 `IntConsumer.accept` 方法的调用实则是对适配器类的 `accept` 方法的调用。

如果你查看了 `accept` 方法对应的字节码的话，你会发现它仅包含一个方法调用，调用至 Java 编译器在解 Lambda 语法糖时生成的方法。

该方法的内容便是 Lambda 表达式的内容，也就是直接调用目标方法 `Test.target`。将这几个方法调用内联进来之后，原本对 `accept` 方法的调用则会被优化为空操作。

下面我将之前的代码更改为带捕获变量的 v7 版本。理论上，每次调用 `invokedynamic` 指令，Java 虚拟机都会新建一个适配器类的实例。然而，实际运行结果还是与直接调用的性能一致。

```
// v7 版本
import java.util.function.IntConsumer;

public class Test {
    public static void target(int i) { }

    public static void main(String[] args) throws Exception {
        int x = 2;

        long current = System.currentTimeMillis();
        for (int i = 1; i <= 2_000_000_000; i++) {
            if (i % 100_000_000 == 0) {
```

```
        long temp = System.currentTimeMillis();

        System.out.println(temp - current);

        current = temp;
    }

    ((IntConsumer) j -> Test.target(x + j)).accept(128);
}
}
}
```

显然，即时编译器的逃逸分析又将该新建实例给优化掉了。我们可以通过虚拟机参数 `-XX:-DoEscapeAnalysis` 来关闭逃逸分析。果然，这时候测得的值约为直接调用的 2.5 倍。

尽管逃逸分析能够去除这些额外新建实例开销，但是它也不是时时奏效。它需要同时满足两件事：`invokedynamic` 指令所执行的方法句柄能够内联，和接下来的对 `accept` 方法的调用也能内联。

只有这样，逃逸分析才能判定该适配器实例不逃逸。否则，我们会在运行过程中不停地生成适配器类实例。所以，我们应当尽量使用非捕获的 Lambda 表达式。

总结与实践

今天我介绍了 `invokedynamic` 指令以及 Lambda 表达式的实现。

`invokedynamic` 指令抽象出调用点的概念，并且将调用该调用点所链接的方法句柄。在第一次执行 `invokedynamic` 指令时，Java 虚拟机将执行它所对应的启动方法，生成并且绑定一个调用点。之后如果再次执行该指令，Java 虚拟机则直接调用已经绑定了的调用点所链接的方法。

Lambda 表达式到函数式接口的转换是通过 `invokedynamic` 指令来实现的。该 `invokedynamic` 指令对应的启动方法将通过 ASM 生成一个适配器类。

对于没有捕获其他变量的 Lambda 表达式，该 `invokedynamic` 指令始终返回同一个适配器类的实例。对于捕获了其他变量的 Lambda 表达式，每次执行 `invokedynamic` 指令将新建一个适配器类实例。

不管是捕获型的还是未捕获型的 Lambda 表达式，它们的性能上限皆可以达到直接调用的性能。其中，捕获型 Lambda 表达式借助了即时编译器中的逃逸分析，来避免实际的新建适配器类实例的操作。

在上一篇的课后实践中，你应该测过这一段代码的性能开销了。我这边测得的结果约为直接调用的 3.5 倍。

```
// v8 版本

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;

public class Test {

    public static void target(int i) { }

    public static void main(String[] args) throws Exception {

        MethodHandles.Lookup l = MethodHandles.lookup();

        MethodType t = MethodType.methodType(void.class, int.class);

        MethodHandle mh = l.findStatic(Test.class, "target", t);

        long current = System.currentTimeMillis();
        for (int i = 1; i <= 2_000_000_000; i++) {
            if (i % 100_000_000 == 0) {
                long temp = System.currentTimeMillis();
                System.out.println(temp - current);
                current = temp;
            }

            mh.invokeExact(128);
        }
    }
}
```

实际上，它与使用 Lambda 表达式或者方法引用的差别在于，即时编译器无法将该方法句柄识别为常量，从而无法进行内联。那么如果将它变成常量行不行呢？

一种方法便是将其赋值给 final 的静态变量，如下面的 v9 版本所示：

```
// v9 版本

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;
```

```
public class Test {  
    public static void target(int i) { }  
  
    static final MethodHandle mh;  
    static {  
        try {  
            MethodHandles.Lookup l = MethodHandles.lookup();  
            MethodType t = MethodType.methodType(void.class, int.class);  
            mh = l.findStatic(Test.class, "target", t);  
        } catch (Throwable e) {  
            throw new RuntimeException(e);  
        }  
    }  
  
    public static void main(String[] args) throws Throwable {  
        long current = System.currentTimeMillis();  
        for (int i = 1; i <= 2_000_000_000; i++) {  
            if (i % 100_000_000 == 0) {  
                long temp = System.currentTimeMillis();  
                System.out.println(temp - current);  
                current = temp;  
            }  
  
            mh.invokeExact(128);  
        }  
    }  
}
```

这个版本测得的数据和直接调用的性能数据一致。也就是说，即时编译器能够将该方法句柄完全内联进来，成为空操作。

今天的实践环节，我们来继续探索方法句柄的性能。运行下面的 v10 版本以及 v11 版本，比较它们的性能并思考为什么。

```
// v10 版本  
import java.lang.invoke.*;
```

```
public class Test {

    public static void target(int i) {

    }

    public static class MyCallSite {

        public final MethodHandle mh;

        public MyCallSite() {
            mh = findTarget();
        }

        private static MethodHandle findTarget() {
            try {
                MethodHandles.Lookup l = MethodHandles.lookup();
                MethodType t = MethodType.methodType(void.class, int.class);
                return l.findStatic(Test.class, "target", t);
            } catch (Throwable e) {
                throw new RuntimeException(e);
            }
        }
    }

    private static final MyCallSite myCallSite = new MyCallSite();

    public static void main(String[] args) throws Throwable {
        long current = System.currentTimeMillis();
        for (int i = 1; i <= 2_000_000_000; i++) {
            if (i % 100_000_000 == 0) {
                long temp = System.currentTimeMillis();
                System.out.println(temp - current);
                current = temp;
            }

            myCallSite.mh.invokeExact(128);
        }
    }
}
```

```
}

// v11 版本
import java.lang.invoke.*;

public class Test {
    public static void target(int i) {
    }

    public static class MyCallSite extends ConstantCallSite {

        public MyCallSite() {
            super(findTarget());
        }

        private static MethodHandle findTarget() {
            try {
                MethodHandles.Lookup l = MethodHandles.lookup();
                MethodType t = MethodType.methodType(void.class, int.class);
                return l.findStatic(Test.class, "target", t);
            } catch (Throwable e) {
                throw new RuntimeException(e);
            }
        }
    }
}

public static final MyCallSite myCallSite = new MyCallSite();

public static void main(String[] args) throws Throwable {
    long current = System.currentTimeMillis();
    for (int i = 1; i <= 2_000_000_000; i++) {
        if (i % 100_000_000 == 0) {
            long temp = System.currentTimeMillis();
            System.out.println(temp - current);
            current = temp;
        }
    }
}
```

```
        myCallSite.getTarget().invokeExact(128);  
    }  
}  
}
```

感谢你的收听，我们下次再见。

[1] <http://openjdk.java.net/jeps/303>



版权归极客邦科技所有，未经许可不得转载

通过留言可与作者互动