

08 | JVM是怎么实现invokedynamic的？（上）

2018-08-08 郑雨迪



08 | JVM是怎么实现invokedynamic的？（上）

朗读人：郑雨迪 12'40" | 5.81M

前不久，“虚拟机”赛马俱乐部来了个年轻人，标榜自己是动态语言，是先进分子。

这一天，先进分子牵着一头鹿进来，说要参加赛马。咱部里的老学究 Java 就不同意了呀，鹿又不是马，哪能参加赛马。

当然了，这种墨守成规的调用方式，自然是先进分子所不齿的。现在年轻人里流行的是鸭子类型（duck typing）[1]，只要是跑起来像只马的，它就是一只马，也就能够参加赛马比赛。

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

1

写留言

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

class Cobra {
    public void race() {
        System.out.println("How do you turn this on?");
    }
}
```

(如何用同一种方式调用他们的赛跑方法?)

说到这里，如果我们将赛跑定义为对赛跑方法（对应上述代码中的 race()）的调用的话，那么这个故事的关键，就在于能不能在马场中调用非马类型的赛跑方法。

为了解答这个问题，我们先来回顾一下 Java 里的方法调用。在 Java 中，方法调用会被编译为 invokestatic, invokespecial, invokevirtual 以及 invokeinterface 四种指令。这些指令与包含目标方法类名、方法名以及方法描述符的符号引用捆绑。在实际运行之前，Java 虚拟机将根据这个符号引用链接到具体的目标方法。

可以看到，在这四种调用指令中，Java 虚拟机明确要求方法调用需要提供目标方法的类名。在这种体系下，我们有两个解决方案。一是调用其中一种类型的赛跑方法，比如说马类的赛跑方法。对于非马的类型，则给它套一层马甲，当成马来赛跑。

另外一种解决方式，是通过反射机制，来查找并且调用各个类型中的赛跑方法，以此模拟真正的赛跑。

显然，比起直接调用，这两种方法都相当复杂，执行效率也可想而知。为了解决这个问题，Java 7 引入了一条新的指令 invokedynamic。该指令的调用机制抽象出调用点这一个概念，并允许应用程序将调用点链接至任意符合条件的方法上。

```
public static void startRace(java.lang.Object)
    0: aload_0                // 加载一个任意对象
    1: invokedynamic race      // 调用赛跑方法
```

写留言

(理想的调用方式)

作为 invokedynamic 的准备工作，Java 7 引入了更加底层、更加灵活的方法抽象：方法句柄 (MethodHandle)。

方法句柄的概念

方法句柄是一个强类型的，能够被直接执行的引用 [2]。该引用可以指向常规的静态方法或者实例方法，也可以指向构造器或者字段。当指向字段时，方法句柄实则指向包含字段访问字节码的虚构方法，语义上等价于目标字段的 getter 或者 setter 方法。

这里需要注意的是，它并不会直接指向目标字段所在类中的 getter/setter，毕竟你无法保证已有的 getter/setter 方法就是在访问目标字段。

方法句柄的类型 (MethodType) 是由所指向方法的参数类型以及返回类型组成的。它是用来确认方法句柄是否适配的唯一关键。当使用方法句柄时，我们其实并不关心方法句柄所指向方法的类名或者方法名。

打个比方，如果兔子的“赛跑”方法和“睡觉”方法的参数类型以及返回类型一致，那么对于兔子递过来的一个方法句柄，我们并不知道会是哪一个方法。

方法句柄的创建是通过 MethodHandles.Lookup 类来完成的。它提供了多个 API，既可以使用反射 API 中的 Method 来查找，也可以根据类、方法名以及方法句柄类型来查找。

当使用后者这种查找方式时，用户需要区分具体的调用类型，比如说对于用 invokestatic 调用的静态方法，我们需要使用 Lookup.findStatic 方法；对于用 invokevirtual 调用的实例方法，以及用 invokeinterface 调用的接口方法，我们需要使用 findVirtual 方法；对于用 invokespecial 调用的实例方法，我们则需要使用 findSpecial 方法。

调用方法句柄，和原本对应的调用指令是一致的。也就是说，对于原本用 invokevirtual 调用的方法句柄，它也会采用动态绑定；而对于原本用 invokespecial 调用的方法句柄，它会采用静态绑定。

```
class Foo {  
    private static void bar(Object o) {  
  
        ..  
    }  
  
    public static Lookup lookup() {
```

```
}

// 获取方法句柄的不同方式

MethodHandles.Lookup l = Foo.lookup(); // 具备 Foo 类的访问权限

Method m = Foo.class.getDeclaredMethod("bar", Object.class);

MethodHandle mh0 = l.unreflect(m);

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

MethodHandle mh1 = l.findStatic(Foo.class, "bar", t);
```

方法句柄同样也有权限问题。但它与反射 API 不同，其权限检查是在句柄的创建阶段完成的。在实际调用过程中，Java 虚拟机并不会检查方法句柄的权限。如果该句柄被多次调用的话，那么与反射调用相比，它将省下重复权限检查的开销。

需要注意的是，方法句柄的访问权限不取决于方法句柄的创建位置，而是取决于 Lookup 对象的创建位置。

举个例子，对于一个私有字段，如果 Lookup 对象是在私有字段所在类中获取的，那么这个 Lookup 对象便拥有对该私有字段的访问权限，即使是在所在类的外边，也能够通过该 Lookup 对象创建该私有字段的 getter 或者 setter。

由于方法句柄没有运行时权限检查，因此，应用程序需要负责方法句柄的管理。一旦它发布了某些指向私有方法的方法句柄，那么这些私有方法便被暴露出去了。

方法句柄的操作

方法句柄的调用可分为两种，一是需要严格匹配参数类型的 `invokeExact`。它有多严格呢？假设一个方法句柄将接收一个 `Object` 类型的参数，如果你直接传入 `String` 作为实际参数，那么方法句柄的调用会在运行时抛出方法类型不匹配的异常。正确的调用方式是将该 `String` 显式转化为 `Object` 类型。

在普通 Java 方法调用中，我们只有在选择重载方法时，才会用到这种显式转化。这是因为经过显式转化后，参数的声明类型发生了改变，因此有可能匹配到不同的方法描述符，从而选取不同的目标方法。调用方法句柄也是利用同样的原理，并且涉及了一个签名多态性（signature polymorphism）的概念。（在这里我们暂且认为签名等同于方法描述符。）

```
public final native @PolymorphicSignature Object invokeExact(Object... args) throws Throwable
```

方法句柄 API 有一个特殊的注解类 `@PolymorphicSignature`。在碰到被它注解的方法调用时，Java 编译器会根据所传入参数的声明类型来生成方法描述符，而不是采用目标方法所声明的描述符。

在刚才的例子中，当传入的参数是 `String` 时，对应的方法描述符包含 `String` 类；而当我们转化为 `Object` 时，对应的方法描述符则包含 `Object` 类。

```
public void test(MethodHandle mh, String s) throws Throwable {
    mh.invokeExact(s);
    mh.invokeExact((Object) s);
}

// 对应的 Java 字节码
public void test(MethodHandle, String) throws java.lang.Throwable;
Code:
    0: aload_1
    1: aload_2

    2: invokevirtual MethodHandle.invokeExact:(Ljava/lang/String;)V
    5: aload_1
    6: aload_2

    7: invokevirtual MethodHandle.invokeExact:(Ljava/lang/Object;)V
   10: return
```

`invokeExact` 会确认该 `invokevirtual` 指令对应的方法描述符，和该方法句柄的类型是否严格匹配。在不匹配的情况下，便会在运行时抛出异常。

如果你需要自动适配参数类型，那么你可以选取方法句柄的第二种调用方式 `invoke`。它同样是一个签名多态性的方法。`invoke` 会调用 `MethodHandle.asType` 方法，生成一个适配器方法句柄，对传入的参数进行适配，再调用原方法句柄。调用原方法句柄的返回值同样也会先进行适配，然后再返回给调用者。

方法句柄还支持增删改参数的操作，这些操作都是通过生成另一个方法句柄来实现的。这其中，改操作就是刚刚介绍的 `MethodHandle.asType` 方法。删操作指的是将传入的部分参数就地抛弃，再调用另一个方法句柄。它对应的 API 是 `MethodHandles.dropArguments` 方法。

增操作则非常有意思。它会往传入的参数中插入额外的参数，再调用另一个方法句柄，它对应的

增操作还可以用来实现方法的柯里化 [3]。举个例子，有一个指向 $f(x, y)$ 的方法句柄，我们可以通过将 x 绑定为 4，生成另一个方法句柄 $g(y) = f(4, y)$ 。在执行过程中，每当调用 $g(y)$ 的方法句柄，它会在参数列表最前面插入一个 4，再调用指向 $f(x, y)$ 的方法句柄。

方法句柄的实现

下面我们来看看 HotSpot 虚拟机中方法句柄调用的具体实现。（由于篇幅原因，这里只讨论 `DirectMethodHandle`。）

前面提到，调用方法句柄所使用的 `invokeExact` 或者 `invoke` 方法具备签名多态性的特性。它们会根据具体的传入参数来生成方法描述符。那么，拥有这个描述符的方法实际存在吗？对 `invokeExact` 或者 `invoke` 的调用具体会进入哪个方法呢？

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

public class Foo {
    public static void bar(Object o) {
        new Exception().printStackTrace();
    }

    public static void main(String[] args) throws Throwable {
        MethodHandles.Lookup l = MethodHandles.lookup();
        MethodType t = MethodType.methodType(void.class, Object.class);
        MethodHandle mh = l.findStatic(Foo.class, "bar", t);
        mh.invokeExact(new Object());
    }
}
```

和查阅反射调用的方式一样，我们可以通过新建异常实例来查看栈轨迹。打印出来的栈轨迹如下所示：

```
$ java Foo
java.lang.Exception
    at Foo.bar(Foo.java:5)
    at Foo.main(Foo.java:12)
```

先别高兴太早。我刚刚提到过，`invokeExact` 会对参数的类型进行校验，并在不匹配的情况下抛出异常。如果它直接调用了方法句柄所指向的方法，那么这部分参数类型校验的逻辑将无处安放。因此，唯一的可能便是 Java 虚拟机隐藏了部分栈信息。

当我们启用了 `-XX:+ShowHiddenFrames` 这个参数来打印被 Java 虚拟机隐藏了的栈信息时，你会发现 `main` 方法和目标方法中间隔着两个貌似是生成的方法。

```
$ java -XX:+UnlockDiagnosticVMOptions -XX:+ShowHiddenFrames Foo
java.lang.Exception
    at Foo.bar(Foo.java:5)
    at java.base/java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(DirectMethodHan
    at java.base/java.lang.invoke.LambdaForm$MH000/766572210.invokeExact_MT000_LLL_V(Lam
    at Foo.main(Foo.java:12)
```

实际上，Java 虚拟机会对 `invokeExact` 调用做特殊处理，调用至一个共享的、与方法句柄类型相关的特殊适配器中。这个适配器是一个 `LambdaForm`，我们可以通过添加虚拟机参数将之导出成 class 文件 (`-Djava.lang.invoke.MethodHandle.DUMP_CLASS_FILES=true`)。

```
final class java.lang.invoke.LambdaForm$MH000 { static void invokeExact_MT000_LLLL_V(jeava.la
Code:
    : aload_0
1 : checkcast      #14                //Mclass java/lang/invoke/ethodHandle
    : dup
5 : astore_0
    : aload_32      : checkcast      #16                //Mclass java/lang/invoke/etho
10: invokestatic   I#22                // Method java/lang/invoke/nvokers.checkExactTyp
13: aload_0
14: invokestatic   #26      I          // Method java/lang/invoke/nvokers.checkCustomiz
17: aload_0
18: aload_1
19: ainvokevirtudl #30                2    // Methodijava/lang/nvokev/ethodHandle.invokeBas
23 return
```

可以看到，在这个适配器中，它会调用 `Invokers.checkExactType` 方法来检查参数类型，然后调用 `Invokers.checkCustomized` 方法。后者会在方法句柄的执行次数超过一个阈值时进行优化（对应参数 `-Djava.lang.invoke.MethodHandle.CUSTOMIZE_THRESHOLD`，默认值为 127）。最后，它会调用方法句柄的 `invokeBasic` 方法。

Java 虚拟机同样会对 `invokeBasic` 调用做特殊处理，这会将调用至方法句柄本身所持有的适配器中。这个适配器同样是一个 `LambdaForm`，你可以通过反射机制将其打印出来。

```
// 该方法句柄持有的 LambdaForm 实例的 toString() 结果
DMH.invokeStatic_LL_V=Lambda(a0:L,a1:L)=>{
    t2:L=DirectMethodHandle.internalMemberName(a0:L);
    t3:V=MethodHandle.linkToStatic(a1:L,t2:L);void}
```

这个适配器将获取方法句柄中的 `MemberName` 类型的字段，并且以它为参数调用 `linkToStatic` 方法。估计你已经猜到了，Java 虚拟机也会对 `linkToStatic` 调用做特殊处理，它将根据传入的 `MemberName` 参数所存储的方法地址或者方法表索引，直接跳转至目标方法。

```
final class MemberName implements Member, Cloneable {
    ...
    //@Injected JVM_Method* vmtarget;
    //@Injected int          vmindex;
    ...
}
```

那么前面那个适配器中的优化又是怎么回事？实际上，方法句柄一开始持有的适配器是共享的。当它被多次调用之后，`Invokers.checkCustomized` 方法会为该方法句柄生成一个特有的适配器。这个特有的适配器会将方法句柄作为常量，直接获取其 `MemberName` 类型的字段，并继续后面的 `linkToStatic` 调用。

```
final class java.lang.invoke.LambdaForm$DMH000 {
    static void invokeStatic000_LL_V(java.lang.Object, java.lang.Object);
    Code:
        0: ldc          #14          // String CONSTANT_PLACEHOLDER_1 <<Foo.bar(Object
        2: checkcast     #16          // class java/lang/invoke/MethodHandle
        5: astore_0      // 上面的优化代码覆盖了传入的方法句柄
        6: aload_0       // 从这开始跟初始版本一致
```



```
10: astore_2
11: aload_1
12: aload_2
13: checkcast    #24          // class java/lang/invoke/MemberName
16: invokestatic  #28          // Method java/lang/invoke/MethodHandle.linkToSta
19: return
```

可以看到，方法句柄的调用和反射调用一样，都是间接调用。因此，它也会面临无法内联的问题。不过，与反射调用不同的是，方法句柄的内联瓶颈在于即时编译器能否将该方法句柄识别为常量。具体内容我会在下一篇中进行详细的解释。

总结与实践

今天我介绍了 invokedynamic 底层机制的基石：方法句柄。

方法句柄是一个强类型的、能够被直接执行的引用。它仅关心所指向方法的参数类型以及返回类型，而不关心方法所在的类以及方法名。方法句柄的权限检查发生在创建过程中，相较于反射调用节省了调用时反复权限检查的开销。

方法句柄可以通过 invokeExact 以及 invoke 来调用。其中，invokeExact 要求传入的参数和所指向方法的描述符严格匹配。方法句柄还支持增删改参数的操作，这些操作是通过生成另一个充当适配器的方法句柄来实现的。

方法句柄的调用和反射调用一样，都是间接调用，同样会面临无法内联的问题。

今天的实践环节，我们来测量一下方法句柄的性能。你可以尝试通过重构代码，将方法句柄变成常量，来提升方法句柄调用的性能。

```
public class Foo {
    public void bar(Object o) {
    }

    public static void main(String[] args) throws Throwable {
        MethodHandles.Lookup l = MethodHandles.lookup();
        MethodType t = MethodType.methodType(void.class, Object.class);
        MethodHandle mh = l.findVirtual(Foo.class, "bar", t);
    }
}
```

```
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(new Foo(), new Object());  
}  
}  
}
```

[1] https://en.wikipedia.org/wiki/Duck_typing

[2]

<https://docs.oracle.com/javase/10/docs/api/java/lang/invoke/MethodHandle.html>

[3]

<https://en.wikipedia.org/wiki/Currying>



版权归极客邦科技所有，未经许可不得转载

精选留言



秋天

这个东西的应用场景是什么？讲的挺深，联系不起来知识

2020-08-12

👍 5

写留言



那么方法句柄是否可以取代反射了呢？

2018-08-08



雨亦奇

👍 2

个人觉得还是按老师的课程安排来走吧。跳来跳去的讲可能会零散不系统。上面两位的说法我不赞同。

2018-08-09



Geek_488a8e

👍 1

这篇读的好吃力，我的一个建议，先抛出一个使用方法句柄的代码例子，然后再剖析代码在虚拟机的实际过程。自顶向下的讲，由浅入深，这篇直接自底向上了，咬咬牙读到最后，才发现这是类似反射的模拟方法调用的方法

2018-08-30

作者回复

多谢建议！

2018-08-30



Leon Wong

👍 0

这篇文章我是花比较长时间去读，有个地方想了很久，文档里也没明示，我自己尝试写代码操作方法句柄参数，为什么MethodType.methodType()里指定的首参类型(ptype0)，居然是方法调用实例的类型，而不是方法的参数类型，这个为什么是这么设计呢？有什么渊源？想了解一下

2018-08-30



FortyTo

👍 0

老师，反射和动态绑定有什么不同，感觉实现的功能都一样。

2018-08-19



ext4

👍 0

雨迪，我看了一下MethodHandle的增操作，即你所提到的bindTo这个API，它貌似只能用于为virtual method绑定第一个参数（即caller的this*指针），并不能普适地为方法绑定一个任意参数（例如把参数列表(int, int)里的第一个参数绑定为常数4）。那么你例子中所提到的更为一般性的柯里化又是怎么实现的呢？

2018-08-09

作者回复

多谢指出：)

bindTo确实限制了只能使用引用类型，而且正如你所说普遍是用来绑定this的。但是由于方法句柄不区分调用者和参数，所以还是可以滥用的。

你可以试试用Integer，然后使用静态方法，或者在使用virtual方法时将bindTo返回的方法句

写留言

BoundMethodHande里有各种非公有的bindArgument*方法，感兴趣可以了解一下

2018-08-10



齐帆

老师能讲下JVM对线程的实现方式吗？

2018-08-08

👍 0

写留言