Java 虚拟机的指令集从 1.0 开始到 JDK7 之间的十余年间没有新增任何指令。但基于 JVM 的语言倒是百花齐放，但是 JVM 有诸多的限制，大部分情况下这些非 Java 的语言需要许多 trick 才能很好的工作。随着 JDK7 的发布，字节码指令集新增了一个重量级指令 invokedynamic。这个指令为为多语言在 JVM 上百花齐放提供了坚实的技术支撑。因为 invokedynamic 概念不是那么好理解，这篇文章专门讲这个指令，希望能帮助你掌握。

JDK7 中虽然在指令集中新增了这个指令，但是 javac 并不会生成 invokedynamic，直到 JDK8 lambda 表达式的出现，在 Java 中才第一次用上了这个指令。

动态语言：变量无类型，变量值才有类型

对于 JVM 而言都是强类型语言，它会在编译时检查传入参数的类型和返回值的类型，比如下面这段代码

```
obj.println("hello world");
```

Java 语言中情况下，对应字节码如下

```
Constant pool:
#1 = Methodref          #6.#15         //
java/lang/Object."<init>":()V
#4 = Methodref          #19.#20        //
java/io/PrintStream.println:(Ljava/lang/String;)V

0: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
3: ldc           #3                  // String
Hello World
5: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
```

可以看到 println 要求如下

- 要调用的对象类：java.io.PrintStream
- 方法名必须是 println
- 方法参数必须是 (String)
- 函数返回值必须是 void

如果当前类找不到符合条件的函数，会在其父类中继续查找，如果 obj 所属的类与 PrintStream 没有继承关系，就算 obj 所属的类有符合条件的函数，上述调用也不会成功（类型检查不会通过）
但是相同的代码在 Groovy 或者 JavaScript 等语言中就不一样了，无论 obj 是何种类型，只有所属类包含函数名为 println，函数参数为(String)的函数，那么上述调用就会成功。这也是我们下面要讲到的「鸭子类型」

鸭子类型（Duck Typing）

鸭子类型概念的名字来源于由 James Whitcomb Riley 提出的鸭子测试，可以这样表述：

> 当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子

在鸭子类型中，关注点在于对象的行为，能做什么；而不是关注对象所属的类型，不关注对象的继承关系

以下面这段 Groovy 脚本为例

```
class Duck {
    void fly() { println "duck flying" }
}


class Airplane {
    void fly() { println "airplane flying" }
}

class Whale {
    void swim() { println "Whale swim" }
}

def liftOff(entity) { entity.fly() }

liftOff(new Duck())
liftOff(new Airplane())
liftOff(new Whale())

输出:
duck flying
airplane flying
groovy.lang.MissingMethodException: No signature
of method: Whale.fly() is applicable for argument
types: () values: []
```

我们可以看到 liftOff 函数，调用了一个传入对象的 fly 方法，但是它并不知道这个对象的类型，也不知道这个对象是否包含了 fly 方法。

开始讲解 invokedynamic 之前需要先介绍一个核心的概念方法句柄（MethodHandle）。

MethodHandle 是什么

MethodHandle 又被称为方法句柄或方法指针，
是java.lang.invoke 包中的 一个类，它的出现使得 Java 可以像其它语言一样把函数当做参数进行传递。MethodHandle 类似于反射中的 Method 类，但它比 Method 类要更加灵活和轻量级。
下面以一个实际的例子来看 MethodHandle 的用法

```
public class Foo {
    public void print(String s) {
        System.out.println("hello, " + s);
    }
    public static void main(String[] args) throws
Throwable {
        Foo foo = new Foo();

        MethodType methodType =
MethodType.methodType(void.class, String.class);
        MethodHandle methodHandle =
MethodHandles.lookup().findVirtual(Foo.class,
"print", methodType);
        methodHandle.invokeExact(foo, "world");
    }
}

运行输出
hello, world
```

使用 MethodHandle 的方法的步骤是:

- 创建 MethodType 对象。MethodType 用来表示方法签名，每个 MethodHandle 都有一个 MethodType 实例，用来指定方法的返回值类型和各个参数类型

- 调用 MethodHandles.lookup 静态方法返回 MethodHandles.Lookup对象，这个对象是查找的上下文，根据方法的不同类型通过 findStatic、findSpecial、findVirtual 等方法查找方法签名为 MethodType 的方法句柄
- 拿到方法句柄以后就可以执行了。通过传入目标方法的参数，使用invoke或者invokeExact就可以进行方法的调用

什么是 invokedynamic

援引 JRuby 作者给 invokedynamic 下一个定义：

> invokedynamic is a user-definable bytecode，You decide how the JVM implements it。

回顾上一节的内容，

```
invokestatic:
System.currentTimeMillis()
Math.abs(-100)
---
invokevirtual:
"hello, world".toUpperCase()
---
invokespecial:
new ArrayList()
---
invokeinterface:
myRunnable.run()
```

这 4 条 invoke* 指令分配规则固化在了虚拟机中，invokedynamic 则把如何查找目标方法的决定权从虚拟机下放到了具体的用户代码中。

invokedynamic 的调用流程如下

- JVM 首次执行 invokedynamic 调用时会调用引导方法（Bootstrap Method）
- 引导方法返回 CallSite 对象，CallSite 内部根据方法签名进行目标方法查找。它的 getTarget 方法返回方法句柄（MethodHandle）对象。
- 在 CallSite 没有变化的情况下，MethodHandle 可以一直被调用，如果 CallSite 有变化的话重新查找即可。以`def add(a, b) { a + b }`为例，如果在代码中一开始使用两个 int 参数进行调用，那么极有可能后面很多次调用还会继续使用两个 int，这样就不用每次都重新选择目标方法。

它们之间的关系如下图所示：

Groovy 与 invokedynamic

以下面的 groovy 代码为例来讲解 invokedynamic 在 Groovy 语言上的应用。

```
Test.groovy
def add(a, b) {
    new Exception().printStackTrace()
    return a + b
}

add("hello", "world")
```

默认情况下 invokedynamic 在 Groovy 上并未启用。如果需要使用需要加上`--indy`选项，使用`groovy --indy Test.groovy`进行编译。生成的 Test.class 对应的字节码如下：

```
public java.lang.Object run();
descriptor: ()Ljava/lang/Object;
```

```
flags: ACC_PUBLIC
Code:
  stack=3, locals=1, args_size=1
     0: aload_0
     1: ldc           #44                 //
String hello
     3: ldc           #46                 //
String world
     5: invokedynamic #52,  0             //
InvokeDynamic #1:invoke:
(LTest;Ljava/lang/String;Ljava/lang/String;)Ljava
/lang/Object;
    10: areturn
------------------
Constant pool:
#1 = Utf8               Test
...省略掉部分字节码...
#52 = InvokeDynamic      #1:#51        //
#1:invoke:
(LTest;Ljava/lang/String;Ljava/lang/String;)Ljava
/lang/Object;
------------------
BootstrapMethods:
...省略掉部分字节码...
1: #34 invokestatic
org/codehaus/groovy/vmplugin/v7/IndyInterface.boo
tstrap:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lan
g/String;Ljava/lang/invoke/MethodType;Ljava/lang/
String;I)Ljava/lang/invoke/CallSite;
Method arguments:
  #48 add
  #49 2
```

可以看到 add("hello", "world") 调用被翻译为了 invokedynamic 指令，第一次参数是常量池中的#52，这个条目又指向了 BootstrapMethods 中的 #1，调用了静态方法 IndyInterface.bootstrap，返回值是一个 CallSite 对象，这个函数签名如下：

```
public static CallSite bootstrap(
    Lookup caller, // the caller
    String callType, // the type of the call
    MethodType type, // the MethodType
    String name, // the real method name
    int flags // call flags
    ) {
}
```

其中 callType 为调用类型，是枚举类 CALL_TYPES 的一种，这里为CALL_TYPES.METHOD("invoke")，name 为实际调用的函数名，这里为"add"。

这个函数内部调用了 realBootstrap 函数，这个函数返回了 CallSite 对象，这个 CallSite 的目标方法句柄（MethodHandle）真正调用了 selectMethod 方法，这个方法在运行期选择合适的方式进行调用。

selectMethod 方法内部则是通过 MethodHandle 的 invokeExact 方法执行了最终的方法调用。

简化上述过程为伪代码就是

```
public static void main(String[] args) throws
Throwable {
    MethodHandles.Lookup lookup =
MethodHandles.lookup();
    MethodType mt =
MethodType.methodType(Object.class,
            Object.class, Object.class);
    CallSite callSite =
            IndyInterface.bootstrap(lookup,
"invoke", mt, "add", 0);
    MethodHandle mh = callSite.getTarget();
    mh.invoke(obj, "hello", "world");
}
```

Groovy 采用 invokedynamic 指令有哪些好处?

- 标准化。使用 Bootstrap Method、CallSite、MethodHandle 机制使得动态调用的方式得到统一
- 保持了字节码层的统一和向后兼容。把动态方法的分派逻辑下放到语言实现层，未来版本可以很方便的进行优化、修改
- 高性能。接近原生 Java 调用的性能，也可以享受到 JIT 优化等

有人会有疑问，invokedynamic 只能在动态语言上吗？其实不是的，invokedynamic 是一种方法动态分派的方式，除了用于动态语言还有其他很多的用途，比如下一篇文章我们要讲的 Java 的 lambda 表达式。

小结

这篇文章主要介绍了 invokedynamic 指令的原理。
invokedynamic 其实是一种调用方法的新方式，它用来告诉 JVM 可以延迟确认最终要调用的哪个方法。一开始 invokedynamic 并不知

道要调用什么目标方法。第一次调用时引导方法（Bootstrap Method）会被调用，由这个引导方法决定哪个目标方法进行调用。