深入JVM - 实例详解invoke相关操作码

Java虚拟机规范中有一个章节专门列出了操作码助记符, 对应的链接为: Java Virtual Machine Specification: Chapter 7. Opcode Mnemonics by Opcode

其中, 方法调用相关的操作码为:

十进制	十六进制	助记符	说明
182	(0xb6)	invokevirtual	调用类的实例方法;
183	(0xb7)	invokespecial	调用特殊实例方法; 如构造函数、超类方法,以及private
184	(0xb8)	invokestatic	调用静态方法
185	(0xb9)	invokeinterface	调用接口方法
186	(0xba)	invokedynamic	动态方法调用

下面我们通过实际的例子, 进行详细介绍。

请看代码:

```
package com.cncounter.opcode;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

/**
 * 演示invoke操作码
 */
public class DemoInvokeOpcode {

  public static void testMethodInvoke() {
      // 183; invokespecial
      HashMap<String, String> hashMap = new HashMap<String, String>(100);
      // 182; invokevirtual
      hashMap.put("name", "tiemao");
      // 赋值给Map接口引用
      Map<String, String> map = hashMap;
      // 185; invokeinterface
      map.putIfAbsent("url", "https://renfufei.blog.csdn.net");
      // 使用lambda
```

```
        List<String> upperKeys = map.keySet().stream()
                // 186; invokedynamic
                .map(i -> i.toUpperCase())
                .collect(Collectors.toList());
        // 184; invokestatic
        String str = String.valueOf(upperKeys);
        // 183; invokespecial
        System.out.println(str);
    }

    public static void main(String[] args) {
        // 184; invokestatic
        testMethodInvoke();
    }
}
```

执行main方法之后的输出内容为:

```
[NAME, URL]
```

我们可以使用以下命令进行编译和反编译:

```
# 查看JDK工具的帮助信息
javac -help
javap -help

# 带调试信息编译
javac -g DemoInvokeOpcode.java
# 反编译
javap -v DemoInvokeOpcode.class

# 因为带了package, 所以执行时需要注意路径:
cd ../../..
java com.cncounter.opcode.DemoInvokeOpcode
```

javac编译之后, 可以看到只生成了一个文件 `DemoInvokeOpcode.class`。 这也是 lambda 与内部类不同的地方。

反编译工具 javap 输出的字节码信息很多, 节选出我们最关心的testMethodInvoke方法部分:

```
public static void testMethodInvoke();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=4, args_size=0
       0: new           #2                  // class java/util/HashMap
       3: dup
       4: bipush        100
```

```
       6: invokespecial #3                  // Method java/util/HashMap."
<init>":(I)V
       9: astore_0
      10: aload_0
      11: ldc           #4                  // String name
      13: ldc           #5                  // String tiemao
      15: invokevirtual #6                  // Method java/util/HashMap.put:
(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
      18: pop
      19: aload_0
      20: astore_1
      21: aload_1
      22: ldc           #7                  // String url
      24: ldc           #8                  // String
https://renfufei.blog.csdn.net
      26: invokeinterface #9, 3             // InterfaceMethod
java/util/Map.putIfAbsent:
(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
      31: pop
      32: aload_1
      33: invokeinterface #10, 1            // InterfaceMethod
java/util/Map.keySet:()Ljava/util/Set;
      38: invokeinterface #11, 1            // InterfaceMethod
java/util/Set.stream:()Ljava/util/stream/Stream;
      43: invokedynamic #12, 0              // InvokeDynamic #0:apply:
()Ljava/util/function/Function;
      48: invokeinterface #13, 2            // InterfaceMethod
java/util/stream/Stream.map:
(Ljava/util/function/Function;)Ljava/util/stream/Stream;
      53: invokestatic  #14                 // Method
java/util/stream/Collectors.toList:()Ljava/util/stream/Collector;
      56: invokeinterface #15, 2            // InterfaceMethod
java/util/stream/Stream.collect:
(Ljava/util/stream/Collector;)Ljava/lang/Object;
      61: checkcast     #16                 // class java/util/List
      64: astore_2
      65: aload_2
      66: invokestatic  #17                 // Method java/lang/String.valueOf:
(Ljava/lang/Object;)Ljava/lang/String;
      69: astore_3
      70: getstatic     #18                 // Field
java/lang/System.out:Ljava/io/PrintStream;
      73: aload_3
      74: invokevirtual #19                 // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
      77: return
    LineNumberTable:
      line 15: 0
      line 17: 10
```

```
      line 19: 19
      line 21: 21
      line 23: 32
      line 25: 48
      line 26: 53
      line 28: 65
      line 30: 70
      line 31: 77
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
         10      68     0 hashMap   Ljava/util/HashMap;
         21      57     1   map   Ljava/util/Map;
         65      13     2 upperKeys   Ljava/util/List;
         70       8     3   str   Ljava/lang/String;
    LocalVariableTypeTable:
      Start  Length  Slot  Name   Signature
         10      68     0 hashMap
Ljava/util/HashMap<Ljava/lang/String;Ljava/lang/String;>;
         21      57     1   map
Ljava/util/Map<Ljava/lang/String;Ljava/lang/String;>;
         65      13     2 upperKeys   Ljava/util/List<Ljava/lang/String;>;
```

简单解释如下:

- 调用某个类的静态方法, 使用的是 invokestatic 指令。
- 当通过接口引用来调用方法时, 会直接编译为 invokeinterface 指令。
- 调用构造函数会编译为 invokespecial 指令, 当然还包括调用 private 方法, 以及可见的超类方法。
- 如果变量引用的类型是具体类, 则编译器会使用 invokevirtual 来调用 public, protected和包可见级别的方法。
- JDK7新增加了一个 `invokedynamic` 指令, 用来支持"动态类型语言"（Dynamically TypedLanguage, 从JDK8开始引入的lambda表达式, 在使用时会编译为这个指令。

更多文章请参考GitHub上的文章翻译项目: https://github.com/cncounter/translation

同时也请各位大佬点赞Star支持!

原文链接: 2020年文章: 41.深入JVM - 实例详解invoke相关操作码