Java Lambda表达式 实现原理分析

原创 衣舞晨风 于 2018-04-06 07:39:10 发布 阅读量2.4w 收藏 133 点赞数 59 版权

分类专栏： Java Java 进阶 文章标签： java lambda 表达式 实现原理 分析

本文分析基于JDK 9

一、目标

本文主要解决两个问题：

1、 函数式接口 到底是什么?

2、 Lambda 表达式是怎么实现的?

先介绍一个jdk的bin目录下的一个字节码查看工具及反编译 工具：javap

```
C:\Users\Code\Java\study>javap --help
用法: javap <options> <classes>
其中, 可能的选项包括:
  -help  --help  -?        输出此用法消息
  -version                 版本信息
  -v  -verbose             输出附加信息
  -l                       输出行号和本地变量表
  -public                  仅显示公共类和成员
  -protected               显示受保护的/公共类和成员
  -package                 显示程序包/受保护的/公共类
                           和成员 (默认)
  -p  -private             显示所有类和成员
  -c                       对代码进行反汇编
  -s                       输出内部类型签名
  -sysinfo                 显示正在处理的类的
                           系统信息 (路径, 大小, 日期, MD5 散列)
  -constants               显示最终常量
  -classpath <path>        指定查找用户类文件的位置
  -cp <path>               指定查找用户类文件的位置
  -bootclasspath <path>    覆盖引导类文件的位置
```


二、函数式接口

```
@FunctionalInterface
interface IFunctionTest<T> {
    public void print(T x);
}
```

通过javap 反编译IFunctionTest.class 可以看到如下信息

```
$C:\Users\Code\Java\study>javap -p IFunctionTest.class
Compiled from "FunctionTest.java"
interface IFunctionTest<T> {
  public abstract void print(T);
}
```

可以看到函数式接口编译完之后依然是一个接口，这个接口具有唯一的一个抽像方法。

为什么说需要是唯一一个抽象方法?

```
@FunctionalInterface
interface IFunctionTest<T> {
    public void print(T x);
    public void print22(T x,int rr);
}
```

```
C:\Users\Code\Java\study>javac FunctionTest.java
FunctionTest.java:3: 错误: 意外的 @FunctionalInterface 注释
@FunctionalInterface
^
  IFunctionTest 不是函数接口
    在 接口 IFunctionTest 中找到多个非覆盖抽象方法
1 个错误
```


虽然不能在函数式接口中定义多个方法，但可以定义默认方法、静态方法、定义java.lang.Object里的public方法：

```
@FunctionalInterface
interface Print<T> {
    public void print(T x);
    default void doSomeMoreWork1(){
        // Method body
    }
    static void printHello(){
        System.out.println("Hello");
    }
    @Override
    boolean equals(Object obj);
}
```

反编译文件内容如下：

```
$C:\Users\Code\Java\study>javap -p IFunctionTest.class
Compiled from "FunctionTest.java"
interface IFunctionTest<T> {
  public abstract void print(T);
  public void doSomeMoreWork1();
  public static void printHello();
  public abstract boolean equals(java.lang.Object);
}
```

三、Lambda

3.1 示例代码

```
public class LambdaTest {
    public static void printString(String s, Print<String> print) {
        print.print(s);
    }
    public static void main(String[] args) {
        printString("test", (x) -> System.out.println(x));
    }
}

@FunctionalInterface
interface Print<T> {
    public void print(T x);
}
```

通过javac编译LambdaTest.java文件，会生成LambdaTest.class、Print.class两个class文件

```
javac LambdaTest.java
```

```
C:\Users\Code\Java\study>java -version
java version "9.0.4"
Java(TM) SE Runtime Environment (build 9.0.4+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.4+11, mixed mode)

C:\Users\Code\Java\study>javac LambdaTest.java

C:\Users\Code\Java\study>dir
 驱动器 C 中的卷没有标签。
 卷的序列号是 000E-3B9B

 C:\Users\Code\Java\study 的目录

2018/04/05  14:29    <DIR>          .
2018/04/05  14:29    <DIR>          ..
2018/04/05  14:30             1,289 LambdaTest.class
2018/04/05  13:52               353 LambdaTest.java
2018/04/05  14:30               311 Print.class
               3 个文件          1,953 字节
               2 个目录 41,034,010,624 可用字节
```


3.2 对于lambda实现的猜测

那么编译器对Lambda 都做了什么？反编译一下代码如下：

```
C:\Users\Code\Java\study>javap -p LambdaTest.class
Compiled from "LambdaTest.java"
public class LambdaTest {
  public LambdaTest();
  public static void printString(java.lang.String, Print<java.lang.String>);
  public static void main(java.lang.String[]);
  private static void lambda$main$0(java.lang.String);
}
```

由上面的代码可以看出编译器会根据Lambda表达式生成一个私有的静态函数：

```
private static void lambda$main$0(java.lang.String);
```

为了验证上面的转化是否正确? 我们在代码中定义一个lambda$main$0这个的函数，最终代码如下所示：

```
public class LambdaTest {
    public static void printString(String s, Print<String> print) {
        print.print(s);
    }
    public static void main(String[] args) {
        printString("test", (x) -> System.out.println(x));
    }
    private static void lambda$main$0(String s) {
    }
}

@FunctionalInterface
interface Print<T> {
    public void print(T x);
}
```

上面的代码在编译时会报错，错误信息如下：

```
C:\Users\Code\Java\study>javac LambdaTest.java
LambdaTest.java:8: 错误: 符号lambda$main$0(String)与LambdaTest中的 compiler-synt
hesized 符号冲突
    private static void lambda$main$0(String s) {
                        ^
LambdaTest.java:1: 错误: 符号lambda$main$0(String)与LambdaTest中的 compiler-synt
hesized 符号冲突
public class LambdaTest {
^
2 个错误
```

有了上面的内容，可以知道的是Lambda表达式在Java 9中首先会生成一个私有的静态函数，这个私有的静态函数干的就是Lambda表达式里面的内容，那么又是如何调用的生成的私有静态函数（lambda$main$0(String s)）呢?

3.3 反编译代码详解

查看更加详细的反编译结果：

```
$C:\Users\Code\Java\study> javap -p -v -c LambdaTest.class
Classfile /C:/Users/Code/Java/study/LambdaTest.class
  Last modified 2018-4-5; size 1184 bytes
  MD5 checksum b144b5a936a04a7c975eae93c7370174
  Compiled from "LambdaTest.java"
public class LambdaTest
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #9.#24         // java/lang/Object."<init>":()V
   #2 = InterfaceMethodref #25.#26        // Print.print:(Ljava/lang/Object;)V
   #3 = String             #27            // test
   #4 = InvokeDynamic      #0:#33         // #0:print:()LPrint;
   #5 = Methodref          #8.#34         // LambdaTest.printString:(Ljava/lang/String;LPrint;)V
   #6 = Fieldref           #35.#36        // java/lang/System.out:Ljava/io/PrintStream;
   #7 = Methodref          #37.#38        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #8 = Class              #39            // LambdaTest
   #9 = Class              #40            // java/lang/Object
  #10 = Utf8               <init>
  #11 = Utf8               ()V
  #12 = Utf8               Code
  #13 = Utf8               LineNumberTable
  #14 = Utf8               printString
  #15 = Utf8               (Ljava/lang/String;LPrint;)V
  #16 = Utf8               Signature
  #17 = Utf8               (Ljava/lang/String;LPrint<Ljava/lang/String;>;)V
  #18 = Utf8               main
  #19 = Utf8               ([Ljava/lang/String;)V
  #20 = Utf8               lambda$main$0
  #21 = Utf8               (Ljava/lang/String;)V
  #22 = Utf8               SourceFile
  #23 = Utf8               LambdaTest.java
  #24 = NameAndType        #10:#11        // "<init>":()V
  #25 = Class              #41            // Print
  #26 = NameAndType        #42:#43        // print:(Ljava/lang/Object;)V
  #27 = Utf8               test
  #28 = Utf8               BootstrapMethods
  #29 = MethodHandle       #6:#44         // invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/
  #30 = MethodType         #43            //  (Ljava/lang/Object;)V
  #31 = MethodHandle       #6:#45         // invokestatic LambdaTest.lambda$main$0:(Ljava/lang/String;)V
  #32 = MethodType         #21            //  (Ljava/lang/String;)V
  #33 = NameAndType        #42:#46        // print:()LPrint;
  #34 = NameAndType        #14:#15        // printString:(Ljava/lang/String;LPrint;)V
  #35 = Class              #47            // java/lang/System
  #36 = NameAndType        #48:#49        // out:Ljava/io/PrintStream;
  #37 = Class              #50            // java/io/PrintStream
  #38 = NameAndType        #51:#21        // println:(Ljava/lang/String;)V
  #39 = Utf8               LambdaTest
  #40 = Utf8               java/lang/Object
  #41 = Utf8               Print
  #42 = Utf8               print
  #43 = Utf8               (Ljava/lang/Object;)V
  #44 = Methodref          #52.#53        // java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles
  #45 = Methodref          #8.#54         // LambdaTest.lambda$main$0:(Ljava/lang/String;)V
  #46 = Utf8               ()LPrint;
  #47 = Utf8               java/lang/System
  #48 = Utf8               out
  #49 = Utf8               Ljava/io/PrintStream;
  #50 = Utf8               java/io/PrintStream
  #51 = Utf8               println
```

```
  #52 = Class              #55            // java/lang/invoke/LambdaMetafactory
  #53 = NameAndType        #56:#60        // metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lan
  #54 = NameAndType        #20:#21        // lambda$main$0:(Ljava/lang/String;)V

  #55 = Utf8               java/lang/invoke/LambdaMetafactory
  #56 = Utf8               metafactory
  #57 = Class              #62            // java/lang/invoke/MethodHandles$Lookup
  #58 = Utf8               Lookup
  #59 = Utf8               InnerClasses
  #60 = Utf8               (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang
  #61 = Class              #63            // java/lang/invoke/MethodHandles
  #62 = Utf8               java/lang/invoke/MethodHandles$Lookup
  #63 = Utf8               java/lang/invoke/MethodHandles
{
  public LambdaTest();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 1: 0

  public static void printString(java.lang.String, Print<java.lang.String>);
    descriptor: (Ljava/lang/String;LPrint;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=2
         0: aload_1
         1: aload_0
         2: invokeinterface #2,  2            // InterfaceMethod Print.print:(Ljava/lang/Object;)V
         7: return
      LineNumberTable:
        line 3: 0
        line 4: 7
    Signature: #17                          // (Ljava/lang/String;LPrint<Ljava/lang/String;>;)V

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: ldc           #3                  // String test
         2: invokedynamic #4,  0              // InvokeDynamic #0:print:()LPrint;
         7: invokestatic  #5                  // Method printString:(Ljava/lang/String;LPrint;)V
        10: return
      LineNumberTable:
        line 6: 0
        line 7: 10

  private static void lambda$main$0(java.lang.String);
    descriptor: (Ljava/lang/String;)V
    flags: ACC_PRIVATE, ACC_STATIC, ACC_SYNTHETIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #6                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: aload_0
         4: invokevirtual #7                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
         7: return
      LineNumberTable:
        line 6: 0
}
SourceFile: "LambdaTest.java"
InnerClasses:
     public static final #58= #57 of #61; //Lookup=class java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/Met
BootstrapMethods:
```

```
131     0: #29 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/Stri
132         #30 (Ljava/lang/Object;)V
133         #31 invokestatic LambdaTest.lambda$main$0:(Ljava/lang/String;)V
            #32 (Ljava/lang/String;)V
```

这个 class 文件展示了三个主要部分：常量池、构造器方法和 printString、main、lambdamain$main$0方法还有lambda表达式生成的内部类。

3.3.1 动态链接

每个栈帧都有一个运行时常量池的引用。这个引用指向栈帧当前运行方法所在类的常量池。通过这个引用支持动态链接（dynamic linking）。

C/C++ 代码一般被编译成对象文件，然后多个对象文件被链接到一起产生可执行文件或者 dll。在链接阶段，每个对象文件的符号引用被替换成了最终执行文件的相对偏移内存地址。在 Java中，链接阶段是运行时动态完成的。

当 Java 类文件编译时，所有变量和方法的引用都被当做符号引用存储在这个类的常量池中。符号引用是一个逻辑引用，实际上并不指向物理内存地址。JVM 可以选择符号引用解析的时机，一种是当类文件加载并校验通过后，这种解析方式被称为饥饿方式。另外一种是符号引用在第一次使用的时候被解析，这种解析方式称为惰性方式。无论如何 ， JVM 必须要在第一次使用符号引用时完成解析并抛出可能发生的解析错误。绑定是将对象域、方法、类的符号引用替换为直接引用的过程。绑定只会发生一次。一旦绑定，符号引用会被完全替换。如果一个类的符号引用还没有被解析，那么就会载入这个类。每个直接引用都被存储为相对于存储结构（与运行时变量或方法的位置相关联的）偏移量。

3.3.2 常量池

JVM 维护了一个按类型区分的常量池，一个类似于符号表的运行时数据结构。尽管它包含更多数据。Java 字节码需要数据。这个数据经常因为太大不能直接存储在字节码中，取而代之的是存储在常量池中，字节码包含这个常量池的引用。

常量池中可以存储多种类型的数据：

- 数字型
- 字符串型
- 类引用型
- 域引用型
- 方法引用

3.3.3 方法

每一个方法包含四个区域：

- 签名和访问标签
- 字节码
- LineNumberTable：为调试器提供源码中的每一行对应的字节码信息。
- LocalVariableTable：列出了所有栈帧中的局部变量。

操作码	作用
aload0	这个操作码是aload格式操作码中的一个。它们用来把对象引用加载到操作码栈。 表示正在被访问的局部变量数组的位置，但只能是0、1、2、3 中的一个。还有一些其它类似的操作码用来载入非对象引用的数据，如iload, lload, float 和 dload。其中 i 表示 int，l 表示 long，f 表示 float，d 表示 double。局部变量数组位置大于 3 的局部变量可以用 iload, lload, float, dload 和 aload 载入。这些操作码都只需要一个操作数，即数组中的位置
ldc	这个操作码用来将常量从运行时常量池压栈到操作数栈
getstatic	这个操作码用来把一个静态变量从运行时常量池的静态变量列表中压栈到操作数栈
return	这个操作码属于ireturn、lreturn、freturn、dreturn、areturn 和 return 操作码组。每个操作码返回一种类型的返回值，其中 i 表示 int，l 表示 long，f 表示 float，d 表示 double，a 表示 对象引用。没有前缀类型字母的 return 表示返回 void

函数调用操作码	作用
invokestatic	调用类方法（静态绑定，速度快）
invokevirtual	指令调用一个对象的实例方法（动态绑定）

函数调用操作码	作用
invokespecial	指令调用实例初始化方法、私有方法、父类方法。（静态绑定，速度快）
invokeinterface	调用引用类型为interface的实例方法（动态绑定）
invokedynamic	JDK 7引入的，主要是为了支持动态语言的方法调用

3.3.4 代码分析

注意反编译后main方法部分：

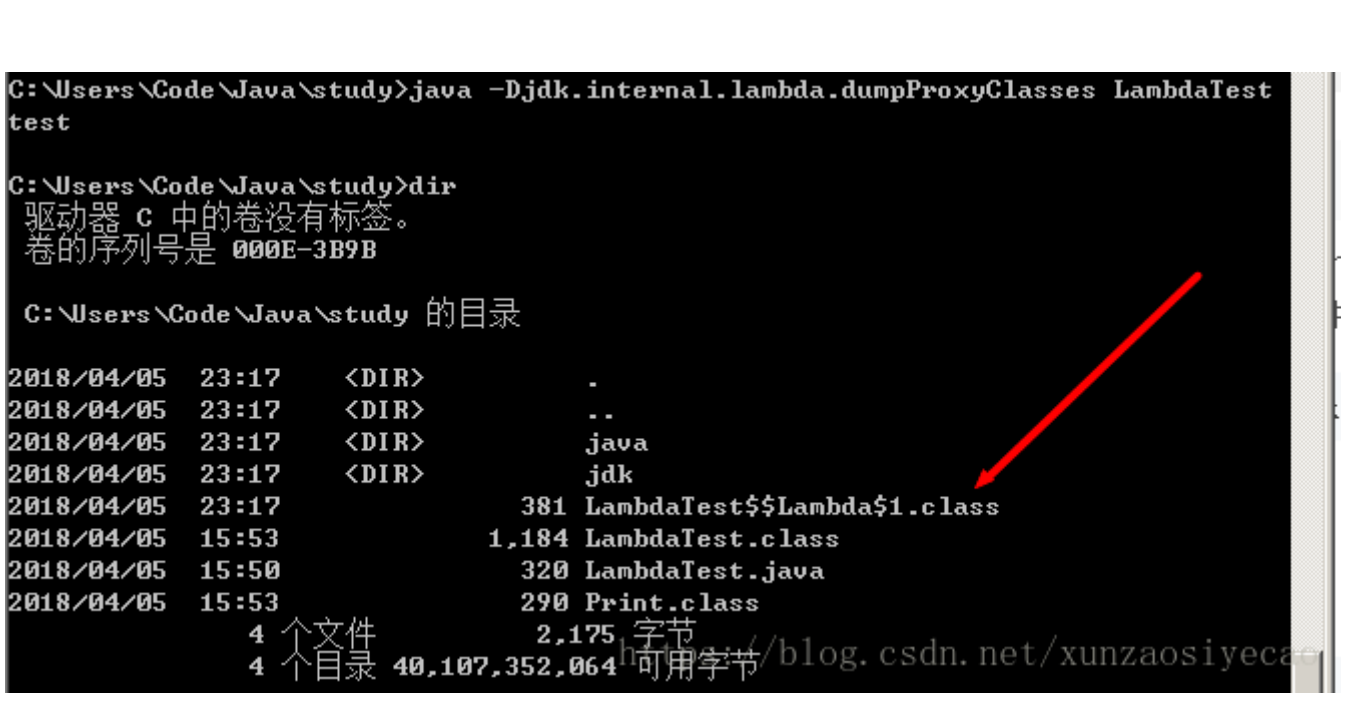
```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
       // ldc 这个操作码用来将常量从运行时常量池压栈到操作数栈
       0: ldc           #3                  // String test
       // 注意下面两句：通过实例调用 print
       2: invokedynamic #4,  0              // InvokeDynamic #0:print:()LPrint;
       //调用静态方法 printString
       7: invokestatic  #5                  // Method printString:(Ljava/lang/String;LPrint;)V
      10: return
```

那么，既然是调用实例方法，那么实例在哪？

```
InnerClasses:
     public static final #58= #57 of #61; //Lookup=class java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/Met
BootstrapMethods:
  0: #29 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/Stri
  Method arguments:
      //对象类型终结符为 L 和 ;
      //Object V
      #30 (Ljava/lang/Object;)V
      #31 invokestatic LambdaTest.lambda$main$0:(Ljava/lang/String;)V
      #32 (Ljava/lang/String;)V
```

可以在运行时加上-Djdk.internal.lambda.dumpProxyClasses，加上这个参数后，运行时，会将生成的内部类class码输出到一个文件中

```
java -Djdk.internal.lambda.dumpProxyClasses LambdaTest
```



通过jad反编译LambdaTest$$Lambda$1.class文件，内容如下：

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
final class LambdaTest$$Lambda$1 implements Print {
```

```
    private LambdaTest$$Lambda$1() {
    }

    public void print(Object obj) {
        LambdaTest.lambda$main$0((String) obj);
    }
}
```

3.3.5 代码还原

至此，我们可以推断出最终执行代码应该是这样的：

```
public class LambdaTest {
    public static void PrintString(String s, Print<String> print) {
        print.print(s);
    }

    public static void main(String[] args) {
        PrintString("test", new LambdaTest$$Lambda$1());
    }

    private static void lambda$main$0(String x) {
        System.out.println(x);
    }

    static final class LambdaTest$$Lambda$1 implements Print {
        public void print(Object obj) {
            LambdaTest.lambda$main$0((String) obj);
        }
        private LambdaTest$$Lambda$1() {
        }
    }

}

@FunctionalInterface
interface Print<T> {
    public void print(T x);
}
```

这里总结稍微有点问题，原理是生成实现了函数式接口的一个内部类和一个lambda表达式里具体操作的方法，内部类重写的方法里调了这个生成的方法。因为本文用的是静态的main函数执行的，所以结论是生成一个静态方法。改成实例方法后，如果lambda表达式直接引用了外部类的静态成员变量或者方法，还是会生成静态方法，但如果直接引用的实例成员变量或者方法，则会生成实例方法。
这是涉及基础知识：静态方法不能引用实例方法！

四、小结

1. 在类编译时，会生成一个私有静态方法+一个内部类；
2. 在内部类中实现了函数式接口，在实现接口的方法中，会调用编译器生成的静态方法；
3. 在使用lambda表达式的地方，通过传递内部类实例，来调用函数式接口方法。

> 就是传递个函数指针，在Java中搞得这么复杂。。。。。。

本文参考：
https://www.cnblogs.com/WJ5888/p/4667086.html
https://www.jianshu.com/p/57bffc6e7acd
http://www.importnew.com/17770.html