# Lambda实现原理

## 实例解析

先从一个例子开始

***public class LambdaTest {***

***public static void print(String name, Print print){***

***print.print(name);***

***}***

***public static void main(String [] args) {***

***String name = "Chen Longfei";***

***String prefix = "hello, ";***

***print(name, (t) -> System.out.println(t));***

***//与上一行不同的是，Lambda表达式的函数体中引用了外部变量‘prefix’***

***print(name, (t) -> System.out.println(prefix + t));***

***}***

***}***

***@FunctionalInterface***

***interface Print {***

***void print(String name);***

***}***

例子很简单，定义了一个函数式接口***Print*** ，main方法中有两处代码以Lambda表达式的方式实现了print接口，分别打印出不带前缀与带前缀的名字。

运行程序，打印结果如下：

***Chen Longfei***

***hello, Chen Longfei***

而(t) -> System.out.println(t)与(t) -> System.out.println(prefix + t))之类的Lambda表达式到底是怎样被编译与调用的呢？

我们知道，编译器编译Java代码时经常在背地里“搞鬼”比如类的全限定名的补全，泛型的类型推断等，编译器耍的这些小聪明可以帮助我们写出更优雅、简洁、高效的代码。鉴于编译器的一贯作风，我们有理由怀疑，新颖而另类的Lambda表达式在编译时很可能会被改造过了。

下面通过javap反编译class文件一探究竟。

Javap是jdk自带的一个字节码查看工具及反编译工具：

***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***

javap -p Print.class

结果如下：

interface test.Print {

public abstract void print(java.lang.String);

}

javap -p LambdaTest.class

结果如下：

Compiled from "LambdaTest.java"

public class test.LambdaTest {

public test.LambdaTest();

public static void print(java.lang.String, test.Print);

public static void main(java.lang.String[]);

private static void lambda$main$1(java.lang.String);

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

}

可见，编译器对Print接口的改造比较小，只是为print方法添加了public abstract关键字，而对LambdaTest的变化就比较大了，添加了两个静态方法：

* private static void lambda$main$1(java.lang.String);
* private static void lambda$main$0(java.lang.String, java.lang.String);

对比原生的java代码，很容易做出推测，这两个静态方法与两处Lambda表达式相关：

print(name, (t) -> System.out.println(t));

print(name, (t) -> System.out.println(prefix + t));

到底有什么关联呢？使用javap -p -v -c LambdaTest.class查看更加详细的反编译结果：

public class test.LambdaTest

minor version: 0

major version: 52

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

#1 = Methodref #15.#30 // java/lang/Object."<init>":()V

#2 = InterfaceMethodref #31.#32 // test/Print.print:(Ljava/lang/String;)V

#3 = String #33 // Chen Longfei

#4 = String #34 // hello,

#5 = InvokeDynamic #0:#39 // #0:print:(Ljava/lang/String;)Ltest/Print;

#6 = Methodref #14.#40 // test/LambdaTest.print:(Ljava/lang/String;Ltest/Print;)V

#7 = InvokeDynamic #1:#42 // #1:print:()Ltest/Print;

#8 = Fieldref #43.#44 // java/lang/System.out:Ljava/io/PrintStream;

#9 = Methodref #45.#46 // java/io/PrintStream.println:(Ljava/lang/String;)V

#10 = Class #47 // java/lang/StringBuilder

#11 = Methodref #10.#30 // java/lang/StringBuilder."<init>":()V

#12 = Methodref #10.#48 // java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder

;

#13 = Methodref #10.#49 // java/lang/StringBuilder.toString:()Ljava/lang/String;

#14 = Class #50 // test/LambdaTest

#15 = Class #51 // java/lang/Object

#16 = Utf8 <init>

#17 = Utf8 ()V

#18 = Utf8 Code

#19 = Utf8 LineNumberTable

#20 = Utf8 print

#21 = Utf8 (Ljava/lang/String;Ltest/Print;)V

#22 = Utf8 main

#23 = Utf8 ([Ljava/lang/String;)V

#24 = Utf8 lambda$main$1

#25 = Utf8 (Ljava/lang/String;)V

#26 = Utf8 lambda$main$0

#27 = Utf8 (Ljava/lang/String;Ljava/lang/String;)V

#28 = Utf8 SourceFile

#29 = Utf8 LambdaTest.java

#30 = NameAndType #16:#17 // "<init>":()V

#31 = Class #52 // test/Print

#32 = NameAndType #20:#25 // print:(Ljava/lang/String;)V

#33 = Utf8 Chen Longfei

#34 = Utf8 hello,

#35 = Utf8 BootstrapMethods

#36 = MethodHandle #6:#53 // invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/inv

oke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/M

ethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;

#37 = MethodType #25 // (Ljava/lang/String;)V

#38 = MethodHandle #6:#54 // invokestatic test/LambdaTest.lambda$main$0:(Ljava/lang/String;Ljava/lang/St

ring;)V

#39 = NameAndType #20:#55 // print:(Ljava/lang/String;)Ltest/Print;

#40 = NameAndType #20:#21 // print:(Ljava/lang/String;Ltest/Print;)V

#41 = MethodHandle #6:#56 // invokestatic test/LambdaTest.lambda$main$1:(Ljava/lang/String;)V

#42 = NameAndType #20:#57 // print:()Ltest/Print;

#43 = Class #58 // java/lang/System

#44 = NameAndType #59:#60 // out:Ljava/io/PrintStream;

#45 = Class #61 // java/io/PrintStream

#46 = NameAndType #62:#25 // println:(Ljava/lang/String;)V

#47 = Utf8 java/lang/StringBuilder

#48 = NameAndType #63:#64 // append:(Ljava/lang/String;)Ljava/lang/StringBuilder;

#49 = NameAndType #65:#66 // toString:()Ljava/lang/String;

#50 = Utf8 test/LambdaTest

#51 = Utf8 java/lang/Object

#52 = Utf8 test/Print

#53 = Methodref #67.#68 // java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHan

dles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;L

java/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;

#54 = Methodref #14.#69 // test/LambdaTest.lambda$main$0:(Ljava/lang/String;Ljava/lang/String;)V

#55 = Utf8 (Ljava/lang/String;)Ltest/Print;

#56 = Methodref #14.#70 // test/LambdaTest.lambda$main$1:(Ljava/lang/String;)V

#57 = Utf8 ()Ltest/Print;

#58 = Utf8 java/lang/System

#59 = Utf8 out

#60 = Utf8 Ljava/io/PrintStream;

#61 = Utf8 java/io/PrintStream

#62 = Utf8 println

#63 = Utf8 append

#64 = Utf8 (Ljava/lang/String;)Ljava/lang/StringBuilder;

#65 = Utf8 toString

#66 = Utf8 ()Ljava/lang/String;

#67 = Class #71 // java/lang/invoke/LambdaMetafactory

#68 = NameAndType #72:#76 // metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava

/lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/

lang/invoke/CallSite;

#69 = NameAndType #26:#27 // lambda$main$0:(Ljava/lang/String;Ljava/lang/String;)V

#70 = NameAndType #24:#25 // lambda$main$1:(Ljava/lang/String;)V

#71 = Utf8 java/lang/invoke/LambdaMetafactory

#72 = Utf8 metafactory

#73 = Class #78 // java/lang/invoke/MethodHandles$Lookup

#74 = Utf8 Lookup

#75 = Utf8 InnerClasses

#76 = Utf8 (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/

lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;

#77 = Class #79 // java/lang/invoke/MethodHandles

#78 = Utf8 java/lang/invoke/MethodHandles$Lookup

#79 = Utf8 java/lang/invoke/MethodHandles

{

public test.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 6: 0

public static void print(java.lang.String, test.Print);

descriptor: (Ljava/lang/String;Ltest/Print;)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 test/Print.print:(Ljava/lang/String;)V

7: return

LineNumberTable:

line 9: 0

line 10: 7

public static void main(java.lang.String[]);

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=2, locals=3, args\_size=1

0: ldc #3 // String Chen Longfei

2: astore\_1

3: ldc #4 // String hello,

5: astore\_2

6: aload\_1

7: aload\_2

8: invokedynamic #5, 0 // InvokeDynamic #0:print:(Ljava/lang/String;)Ltest/Print;

13: invokestatic #6 // Method print:(Ljava/lang/String;Ltest/Print;)V

16: aload\_1

17: invokedynamic #7, 0 // InvokeDynamic #1:print:()Ltest/Print;

22: invokestatic #6 // Method print:(Ljava/lang/String;Ltest/Print;)V

25: return

LineNumberTable:

line 13: 0

line 14: 3

line 16: 6

line 18: 16

line 19: 25

private static void lambda$main$1(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 #8 // Field java/lang/System.out:Ljava/io/PrintStream;

3: aload\_0

4: invokevirtual #9 // Method java/io/PrintStream.println:(Ljava/lang/String;)V

7: return

LineNumberTable:

line 18: 0

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

descriptor: (Ljava/lang/String;Ljava/lang/String;)V

flags: ACC\_PRIVATE, ACC\_STATIC, ACC\_SYNTHETIC

Code:

stack=3, locals=2, args\_size=2

0: getstatic #8 // Field java/lang/System.out:Ljava/io/PrintStream;

3: new #10 // class java/lang/StringBuilder

6: dup

7: invokespecial #11 // Method java/lang/StringBuilder."<init>":()V

10: aload\_0

11: invokevirtual #12 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;

14: aload\_1

15: invokevirtual #12 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;

18: invokevirtual #13 // Method java/lang/StringBuilder.toString:()Ljava/lang/String;

21: invokevirtual #9 // Method java/io/PrintStream.println:(Ljava/lang/String;)V

24: return

LineNumberTable:

line 16: 0

}

SourceFile: "LambdaTest.java"

InnerClasses:

public static final #74= #73 of #77; //Lookup=class java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles

BootstrapMethods:

0: #36 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(

Ljava/lang/invoke/MethodHandles$Lookup;

Ljava/lang/String;

Ljava/lang/invoke/MethodType;

Ljava/lang/invoke/MethodType;

Ljava/lang/invoke/MethodHandle;

Ljava/lang/invoke/MethodType;)

Ljava/lang/invoke/CallSite;

Method arguments:

#37 (Ljava/lang/String;)V

#38 invokestatic test/LambdaTest.lambda$main$0:(Ljava/lang/String;Ljava/lang/String;)V

#37 (Ljava/lang/String;)V

1: #36 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(

Ljava/lang/invoke/MethodHandles$Lookup;

Ljava/lang/String;

Ljava/lang/invoke/MethodType;

Ljava/lang/invoke/MethodType;

Ljava/lang/invoke/MethodHandle;

Ljava/lang/invoke/MethodType;)

Ljava/lang/invoke/CallSite;

Method arguments:

#37 (Ljava/lang/String;)V

#41 invokestatic test/LambdaTest.lambda$main$1:(Ljava/lang/String;)V

#37 (Ljava/lang/String;)V

这个 class 文件展示了三个主要部分：

* 常量池
* 构造方法和 main、print、lambda$main$0、lambda$main$1方法
* lambda表达式生成的内部类。

重点看下main方法的实现：

public static void main(java.lang.String[]);

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=2, locals=3, args\_size=1

// 将字符串常量"Chen Longfei"从常量池压栈到操作数栈

0: ldc #3 // String Chen Longfei

// 将栈顶引用型数值存入第二个本地变，即 String name = "Chen Longfei"

2: astore\_1

// 将字符串常量"hello,"从常量池压栈到操作数栈

3: ldc #4 // String hello,

// 将栈顶引用型数值存入第三个本地变量， 即 String prefix = "hello, "

5: astore\_2

//将第二个引用类型本地变量推送至栈顶，即 name

6: aload\_1

//将第三个引用类型本地变量推送至栈顶，即 prefix

7: aload\_2

//通过invokedynamic指令创建Print接口的实匿名内部类，实现 (t) -> System.out.println(prefix + t)

8: invokedynamic #5, 0 // InvokeDynamic #0:print:(Ljava/lang/String;)Ltest/Print;

//调用静态方法print

13: invokestatic #6 // Method print:(Ljava/lang/String;Ltest/Print;)V

//将第二个引用类型本地变量推送至栈顶，即 name

16: aload\_1

//通过invokedynamic指令创建Print接口的匿名内部类，实现 (t) -> System.out.println(t)

17: invokedynamic #7, 0 // InvokeDynamic #1:print:()Ltest/Print;

//调用静态方法print

22: invokestatic #6 // Method print:(Ljava/lang/String;Ltest/Print;)V

25: return

……

两个匿名内部类是通过BootstrapMethods方法创建的：

//匿名内部类

InnerClasses:

public static final #74= #73 of #77; //Lookup=class java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles

BootstrapMethods:

//调用静态工厂LambdaMetafactory.metafactory创建匿名内部类1。实现了 (t) -> System.out.println(prefix + t)

0: #36 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(

Ljava/lang/invoke/MethodHandles$Lookup;

Ljava/lang/String;

Ljava/lang/invoke/MethodType;

Ljava/lang/invoke/MethodType;

Ljava/lang/invoke/MethodHandle;

Ljava/lang/invoke/MethodType;)

Ljava/lang/invoke/CallSite;

Method arguments:

#37 (Ljava/lang/String;)V

//该类会调用静态方法LambdaTest.lambda$main$0

#38 invokestatic test/LambdaTest.lambda$main$0:(Ljava/lang/String;Ljava/lang/String;)V

#37 (Ljava/lang/String;)V

//调用静态工厂LambdaMetafactory.metafactory创建匿名内部类2，实现了 (t) -> System.out.println(t)

1: #36 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(

Ljava/lang/invoke/MethodHandles$Lookup;

Ljava/lang/String;

Ljava/lang/invoke/MethodType;

Ljava/lang/invoke/MethodType;

Ljava/lang/invoke/MethodHandle;

Ljava/lang/invoke/MethodType;)

Ljava/lang/invoke/CallSite;

Method arguments:

#37 (Ljava/lang/String;)V

//该类会调用静态方法LambdaTest.lambda$main$1

#41 invokestatic test/LambdaTest.lambda$main$1:(Ljava/lang/String;)V

#37 (Ljava/lang/String;)V

可以在运行时加上-Djdk.internal.lambda.dumpProxyClasses=%PATH%，加上这个参数后，运行时，会将生成的内部类class输出到%PATH%路径下



Javap -p -c 反编译两个文件

//print(name, (t) -> System.out.println(t))的实例

final class test.LambdaTest$$Lambda$1 implements test.Print {

private test.LambdaTest$$Lambda$1(); //构造方法

Code:

0: aload\_0

1: invokespecial #10 // Method java/lang/Object."<init>":()V

4: return

//实现test.Print接口方法

public void print(java.lang.String);

Code:

0: aload\_1

//调用静态方法LambdaTest.lambda$1

1: invokestatic #18 // Method test/LambdaTest.lambda$1:(Ljava/lang/String;)V

4: return

}

//print(name, (t) -> System.out.println(prefix + t))的实例

final class test.LambdaTest$$Lambda$2 implements test.Print {

private final java.lang.String arg$1;

private test.LambdaTest$$Lambda$2(java.lang.String);

Code:

0: aload\_0

1: invokespecial #13 // Method java/lang/Object."<init>":()V

4: aload\_0

5: aload\_1

//final变量arg$1由构造方法传入

6: putfield #15 // Field arg$1:Ljava/lang/String;

9: return

//该方法返回一个 LambdaTest$$Lambda$2实例

private static test.Print get$Lambda(java.lang.String);

Code:

0: new #2 // class test/LambdaTest$$Lambda$2

3: dup

4: aload\_0

5: invokespecial #19 // Method "<init>":(Ljava/lang/String;)V

8: areturn

//实现test.Print接口方法

public void print(java.lang.String);

Code:

0: aload\_0

1: getfield #15 // Field arg$1:Ljava/lang/String;

4: aload\_1

//调用静态方法LambdaTest.lambda$0

5: invokestatic #27 // Method test/LambdaTest.lambda$0:(Ljava/lang/String;Ljava/lang/String;)V

8: return

}

对比两个实例，可以发现，由于表达式print(name, (t) -> System.out.println(prefix + t))引用了局部变量prefix，LambdaTest$$Lambda$2

类多了一个final参数：

private final java.lang.String arg$1

该参数由构造方法传入，用来存储main方法中的局部变量prefix：

String prefix = "hello, ";

由于外部类的main方法与匿名内部类LambdaTest$$Lambda$2引用了同一份变量，该变量虽然在代码层面独立存储于两个类当中，但是在逻辑上具有一致性，所以匿名内部类中加上了final关键字，而外部类中虽然没有为prefix显式地添加final，但是在被Lambda表达式引用后，该变量就自动隐含了final语意（再次更改会报错）。

## InvokeDynamic

通过上面的例子可以发现，Lambda表达式由虚拟机指令InvokeDynamic实现方法调用。

### 方法调用

方法调用不等同于方法执行，方法调用阶段的唯一任务就是确定被调用方法的版本（即确定具体调用那一个方法），不涉及方法内部具体运行。

java虚拟机中提供了5条方法调用的字节码指令：

* invokestatic:调用静态方法
* invokespecial:调用实例构造器<init>方法、私有方法、父类方法
* invokevirtual:调用虚方法。
* invokeinterface:调用接口方法，在运行时再确定一个实现该接口的对象
* invokedynamic:运行时动态解析出调用的方法，然后去执行该方法。

InvokeDynamic是 java 7 引入的一条新的虚拟机指令，这是自 1.0 以来第一次引入新的虚拟机指令。到了 java 8 这条指令才第一次在 java 应用，用在 Lambda 表达式中。InvokeDynamic与其他invoke指令不同的是它允许由应用级的代码来决定方法解析。

### 指令规范

根据JVM规范的规定，indy的操作码是186（0xBA），格式是：

***invokedynamic indexbyte1 indexbyte2 0 0***

InDy指令有四个操作数，前两个操作数构成一个索引[ (indexbyte1 << 8) | indexbyte2 ]，指向类的常量池，后两个操作数保留，必须是0。

查看上例中LambdaTest类的反编译结果，第一处Lambda表达式

print(name, (t) -> System.out.println(t));

对应的指令为：

17: invokedynamic #7, 0 // InvokeDynamic #1:print:()Ltest/Print;

常量池中#7对应的常量为：

#7 = InvokeDynamic #1:#42 // #1:print:()Ltest/Print;

其类型为CONSTANT\_InvokeDynamic\_info，CONSTANT\_InvokeDynamic\_info结构是Java7新引入class文件的，其用途就是给indy指令指定启动方法（bootstrap method）、调用点call site（）等信息, 实际上是个 MethodHandle（方法句柄）对象。

#1代表BootstrapMethods表中的索引，即

BootstrapMethods:

//第一个

0: #36 ……

//第二个

1: #36 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(

Ljava/lang/invoke/MethodHandles$Lookup;

Ljava/lang/String;

Ljava/lang/invoke/MethodType;

Ljava/lang/invoke/MethodType;

Ljava/lang/invoke/MethodHandle;

Ljava/lang/invoke/MethodType;)

Ljava/lang/invoke/CallSite;

Method arguments:

#37 (Ljava/lang/String;)V

#41 invokestatic test/LambdaTest.lambda$main$1:(Ljava/lang/String;)V

#37 (Ljava/lang/String;)V

也就是说，最终调用的是java.lang.invoke.LambdaMetafactory类的静态方法metafactory()。

### 执行过程

为了更深入的了解InvokeDynamic，先来看几个术语：

* dynamic call site

程序中出现lambda的地方都被称作dynamic call site，CallSite 就是一个 MethodHandle（方法句柄）的 holder。方法句柄指向一个调用点真正执行的方法。

* bootstrap method

java里对所有Lambda的有统一的bootstrap method(LambdaMetafactory.metafactory)，bootstrap运行期动态生成了匿名类，将其与CallSite绑定，得到了一个获取匿名类实例的call site object

* call site object

call site object持有MethodHandle的引用作为它的target，它是bootstrap method方法成功调用后的结果，将会与 dynamic call site永久绑定。call site object的target会被JVM执行，就如同执行一条invokevirtual指令，其所需的参数也会被压入operand stack。最后会得一个实现了functional interface的对象。

inDy 首先需要生成一个 CallSite（调用点对象），CallSite 是由 bootstrap method 返回，也就是调(LambdaMetafactory.metafactory方法。

public static CallSite metafactory(MethodHandles.Lookup caller,

String invokedName,

MethodType invokedType,

MethodType samMethodType,

MethodHandle implMethod,

MethodType instantiatedMethodType)

throws LambdaConversionException {

AbstractValidatingLambdaMetafactory mf;

mf = new InnerClassLambdaMetafactory(caller, invokedType,

invokedName, samMethodType,

implMethod, instantiatedMethodType,

false, EMPTY\_CLASS\_ARRAY, EMPTY\_MT\_ARRAY);

mf.validateMetafactoryArgs();

return mf.buildCallSite();

}

前三个参数是固定的，由VM自动压栈：

* MethodHandles.Lookup caller代表Indy 指令所在的类的上下文（在上例中就是LambdaTest），可以通过 Lookup#lookupClass()获取这个类
* String invokedName表示要实现的方法名（在上例中就是Print接口的方法名”print”）
* MethodType invokedType call site object所持有的MethodHandle需要的参数和返回类型（signature）

接下来就是附加参数，这些参数是灵活的，由Bootstrap methods 表提供：

* MethodType samMethodType表示要实现functional interface里面抽象方法的类型
* MethodHandle implMethod表示编译器给生成的 desugar 方法，是一个 MethodHandle
* MethodType instantiatedMethodType即运行时的类型，因为方法定义可能是泛型，传入时可能是具体类型String之类的，要做类型校验强转等等

LambdaMetafactory.metafactory 方法会创建一个匿名类，这个类是通过 ASM 编织字节码在内存中生成的，然后直接通过 UNSAFE 直接加载而不会写到文件里。

### MethodHandle

要让invokedynamic正常运行，一个核心的概念就是**方法句柄**（method handle）。它代表了一个可以从invokedynamic调用点进行调用的方法。每个invokedynamic指令都会与一个特定的方法关联（也就是bootstrap method或BSM）。当编译器遇到invokedynamic指令的时候，BSM会被调用，会返回一个包含了方法句柄的对象，这个对象表明了调用点要实际执行哪个方法。

Java 7 API中加入了java.lang.invoke.MethodHandle（及其子类），通过它们来代表invokedynamic指向的方法。

一个Java方法可以视为由四个基本内容所构成：

* 名称
* 签名（包含返回类型）
* 定义它的类
* 实现方法的字节码

这意味着如果要引用某个方法，我们需要有一种有效的方式来表示**方法签名**（而不是反射中强制使用的令人讨厌的Class<?>[] hack方式）。

方法句柄首先需要的一个表达方法签名的方式，以便于查找。在Java 7引入的Method Handles API中，这个角色是由java.lang.invoke.MethodType类来完成的，它使用一个不可变的实例来代表签名。要获取MethodType，我们可以使用methodType()工厂方法。这是一个参数可变的方法，以class对象作为参数。

第一个参数所使用的class对象，对应着签名的返回类型；剩余参数中所使用的class对象，对应着签名中方法参数的类型。例如：

***//toString()的签名***

***MethodType mtToString = MethodType.methodType(String.class);***

***// setter方法的签名***

***MethodType mtSetter = MethodType.methodType(void.class, Object.class);***

***// Comparator中compare()方法的签名***

***MethodType mtStringComparator = MethodType.methodType(int.class, String.class, String.class);***

现在我们就可以使用MethodType，再组合方法名称以及定义方法的类来查找方法句柄。要实现这一点，我们需要调用静态的MethodHandles.lookup()方法。这样的话，会给我们一个“查找上下文（lookup context）”，这个上下文基于当前正在执行的方法（也就是调用lookup()的方法）的访问权限。

查找上下文对象有一些以“find”开头的方法，例如，findVirtual()、findConstructor()、findStatic()等。这些方法将会返回实际的方法句柄，需要注意的是，只有在创建查找上下文的方法能够访问（调用）被请求方法的情况下，才会返回句柄。这与反射不同，我们没有办法绕过访问控制。换句话说，方法句柄中并没有与setAccessible()对应的方法。例如

***public MethodHandle getToStringMH() {***

***MethodHandle mh = null;***

***MethodType mt = MethodType.methodType(String.class);***

***MethodHandles.Lookup lk = MethodHandles.lookup();***

***try {***

***mh = lk.findVirtual(getClass(), "toString", mt);***

***} catch (NoSuchMethodException | IllegalAccessException mhx) {***

***throw (AssertionError)new AssertionError().initCause(mhx);***

***}***

***return mh;***

***}***

MethodHandle中有两个方法能够触发对方法句柄的调用，那就是invoke()和invokeExact()。这两个方法都是以接收者（receiver）和调用变量作为参数，所以它们的签名为：

public final Object invoke(Object... args) throws Throwable;

public final Object invokeExact(Object... args) throws Throwable;

两者的区别在于，invokeExact()在调用方法句柄时会试图严格地直接匹配所提供的变量。而invoke()与之不同，在需要的时候，invoke()能够稍微调整一下方法的变量。invoke()会执行一个asType()转换，它会根据如下的这组规则来进行变量的转换：

* 如果需要的话，原始类型会进行装箱操作
* 如果需要的话，装箱后的原始类型会进行拆箱操作
* 如果必要的话，原始类型会进行扩展
* void返回类型会转换为0（对于返回原始类型的情况），而对于预期得到引用类型的返回值的地方，将会转换为null
* null值会被视为正确的，不管静态类型是什么都可以进行传递

接下来，我们看一下考虑上述规则的简单调用样例：

***Object rcvr = "a";***

***try {***

***MethodType mt = MethodType.methodType(int.class);***

***MethodHandles.Lookup l = MethodHandles.lookup();***

***MethodHandle mh = l.findVirtual(rcvr.getClass(), "hashCode", mt);***

***int ret;***

***try {***

***ret = (int)mh.invoke(rcvr);***

***System.out.println(ret);***

***} catch (Throwable t) {***

***t.printStackTrace();***

***}***

***} catch (IllegalArgumentException | NoSuchMethodException | SecurityException e) {***

***e.printStackTrace();***

***} catch (IllegalAccessException x) {***

***x.printStackTrace();***

***}***

上面的代码调用了Object的hashcode()方法，看到这里，你肯定会说这不就是 Java 的反射吗？

确实，MethodHandle 和 Reflection 实现的功能有太多相似的地方，都是运行时解析方法调用，理解方法句柄的一种方式就是将其视为以安全、现代的方式来实现反射的核心功能，在这个过程会尽可能地保证类型的安全。

但是，究其本质，两者之间还是有区别的：

* MethodHandle 和 Reflection 都可以分派方法调用，但是 MethodHandle 比 Reflection 更强大，它是模拟字节码层次的方法分派。有兴趣的同学可以对比 MethodHandles.Lookup 提供的findStatic、findVirtual、findSpecial三个方法和 Reflection 的反射调用；
* MethodHandle 是结合 invokedynamic 指令一起为动态语言服务的，也就是说MethodHandle （更准确的来说是其设计理念）是服务于所有运行在JVM之上的语言，而 Relection 则只是适用 Java 语言本身。

# Stream实现原理