再了解了Java 8 Lambda的一些基本概念和应用后， 我们会有这样的一个问题: Lambda表达式被编译成了什么？。 这是一个有趣的问题，涉及到JDK的具体的实现。 本文将介绍OpenJDK对Lambda表达式的转换细节， 读者可以了解Java 8 Lambda表达式背景知识。

**Lambda表达式的转换策略**

[Brian Goetz](https://www.google.com.hk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CB0QFjAA&url=%68%74%74%70%73%3a%2f%2f%74%77%69%74%74%65%72%2e%63%6f%6d%2f%42%72%69%61%6e%47%6f%65%74%7a&ei=9v9aVJCEGcn28QXsrIHwBQ&usg=AFQjCNHu9y69SvQVsjuMqyBv3PMELARZ5Q&bvm=bv.78677474,d.dGc)是Oracle的Java语言架构师， JSR 335(Lambda Expression)规范的lead, 写了几篇Lambda设计方面的文章， 其中之一就是[Translation of Lambda Expressions](http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html)。 这篇文章介绍了Java 8 Lambda设计时的考虑以及实现方法。

他提到， Lambda表达式可以通过内部类， method handle, dynamic proxy等方式实现， 但是这些方法各有优劣。 真正要实现Lambda表达式， 必须兼顾两个目标： 一是不引入特定策略，以期为将来的优化提供最大的灵活性， 二是保持类文件格式的稳定。 通过Java 7中引入的invokedynamic （[JSR 292](https://jcp.org/en/jsr/detail?id=292)）, 可以很好的兼顾这两个目标。

invokedynamic 在缺乏静态类型信息的情况下可以支持有效的灵活的方法调用。主要是为了日益增长的运行在JVM上的动态类型语言， 如Groovy, JRuby。

invokedynamic将Lambda表达式的转换策略推迟到运行时， 这也意味着我们现在编译的代码在将来的转换策略改变的情况下也能正常运行。

编译器在编译的时候， 会将Lambda表达式的表达式体 (lambda body)脱糖(desugar) 成一个方法，此方法的参数列表和返回类型和lambda表达式一致， 如果有捕获参数， 脱糖的方法的参数可能会更多一些， 并会产生一个invokedynamic调用， 调用一个call site。 这个call site被调用时会返回lambda表达式的目标类型(functional interface)的一个实现类。 这个call site称为这个lambda表达式的lambda factory。 lambda factory的[Bootstrap](http://www.codeceo.com/article/bootstrap-web-css-framework.html)方法是一个标准方法， 叫做lambda metafactory。

编译器在转换lambda表达式时， 可以推断出表达式的参数类型，返回类型以及异常， 称之为natural signature， 我们将目标类型的方法签名称之为lambda descriptor, lambda factory的返回对象实现了函数式接口， 并且关联的表达式的代码逻辑， 称之为lambda object。

**转换举例**

以上的解释有点晦涩， 简单来说

* 编译时
  + Lambda 表达式会生成一个方法， 方法实现了表达式的代码逻辑
  + 生成invokedynamic指令， 调用bootstrap方法， 由java.lang.invoke.LambdaMetafactory.metafactory方法实现
* 运行时
  + invokedynamic指令调用metafactory方法。 它会返回一个CallSite, 此CallSite返回目标类型的一个匿名实现类， 此类关联编译时产生的方法
  + lambda表达式调用时会调用匿名实现类关联的方法。

最简单的一个lambda表达式的例子：

**public** **class** **Lambda1** {

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

**Consumer**<**String**> c = s -> **System**.out.println(s);

c.accept("hello lambda");

}

}

使用javap查看生成的字节码 javap -c -p -v com/colobu/lambda/chapter5/Lambda1.class:

[root@colobu bin]**# javap -c -p -v com/colobu/lambda/chapter5/Lambda1.class**

Classfile /mnt/eclipse/Lambda/bin/com/colobu/lambda/chapter5/Lambda1.class

Last modified Nov 6, 2014; size 1401 bytes

MD5 checksum fe2b2d3f039a9ba4209c488a8c4b4ea8

Compiled from "Lambda1.java"

public class com.colobu.lambda.chapter5.Lambda1

SourceFile: "Lambda1.java"

BootstrapMethods:

0: **#57 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:

**#58 (Ljava/lang/Object;)V**

**#61 invokestatic com/colobu/lambda/chapter5/Lambda1.lambda$0:(Ljava/lang/String;)V**

**#62 (Ljava/lang/String;)V**

InnerClasses:

public **static** final **#68= #64 of #66; //Lookup=class java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles**

minor version: 0

major version: 52

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

**#1 = Class #2 // com/colobu/lambda/chapter5/Lambda1**

**#2 = Utf8 com/colobu/lambda/chapter5/Lambda1**

**#3 = Class #4 // java/lang/Object**

**#4 = Utf8 java/lang/Object**

**#5 = Utf8 <init>**

**#6 = Utf8 ()V**

**#7 = Utf8 Code**

**#8 = Methodref #3.#9 // java/lang/Object."<init>":()V**

**#9 = NameAndType #5:#6 // "<init>":()V**

**#10 = Utf8 LineNumberTable**

**#11 = Utf8 LocalVariableTable**

**#12 = Utf8 this**

**#13 = Utf8 Lcom/colobu/lambda/chapter5/Lambda1;**

**#14 = Utf8 main**

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

**#16 = NameAndType #17:#18 // accept:()Ljava/util/function/Consumer;**

**#17 = Utf8 accept**

**#18 = Utf8 ()Ljava/util/function/Consumer;**

**#19 = InvokeDynamic #0:#16 // #0:accept:()Ljava/util/function/Consumer;**

**#20 = String #21 // hello lambda**

**#21 = Utf8 hello lambda**

**#22 = InterfaceMethodref #23.#25 // java/util/function/Consumer.accept:(Ljava/lang/Object;)V**

**#23 = Class #24 // java/util/function/Consumer**

**#24 = Utf8 java/util/function/Consumer**

**#25 = NameAndType #17:#26 // accept:(Ljava/lang/Object;)V**

**#26 = Utf8 (Ljava/lang/Object;)V**

**#27 = Utf8 args**

**#28 = Utf8 [Ljava/lang/String;**

**#29 = Utf8 c**

**#30 = Utf8 Ljava/util/function/Consumer;**

**#31 = Utf8 LocalVariableTypeTable**

**#32 = Utf8 Ljava/util/function/Consumer<Ljava/lang/String;>;**

**#33 = Utf8 lambda$0**

**#34 = Utf8 (Ljava/lang/String;)V**

**#35 = Fieldref #36.#38 // java/lang/System.out:Ljava/io/PrintStream;**

**#36 = Class #37 // java/lang/System**

**#37 = Utf8 java/lang/System**

**#38 = NameAndType #39:#40 // out:Ljava/io/PrintStream;**

**#39 = Utf8 out**

**#40 = Utf8 Ljava/io/PrintStream;**

**#41 = Methodref #42.#44 // java/io/PrintStream.println:(Ljava/lang/String;)V**

**#42 = Class #43 // java/io/PrintStream**

**#43 = Utf8 java/io/PrintStream**

**#44 = NameAndType #45:#34 // println:(Ljava/lang/String;)V**

**#45 = Utf8 println**

**#46 = Utf8 s**

**#47 = Utf8 Ljava/lang/String;**

**#48 = Utf8 SourceFile**

**#49 = Utf8 Lambda1.java**

**#50 = Utf8 BootstrapMethods**

**#51 = Methodref #52.#54 // 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;**

**#52 = Class #53 // java/lang/invoke/LambdaMetafactory**

**#53 = Utf8 java/lang/invoke/LambdaMetafactory**

**#54 = NameAndType #55:#56 // 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;**

**#55 = Utf8 metafactory**

**#56 = 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;**

**#57 = MethodHandle #6:#51 // 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;**

**#58 = MethodType #26 // (Ljava/lang/Object;)V**

**#59 = Methodref #1.#60 // com/colobu/lambda/chapter5/Lambda1.lambda$0:(Ljava/lang/String;)V**

**#60 = NameAndType #33:#34 // lambda$0:(Ljava/lang/String;)V**

**#61 = MethodHandle #6:#59 // invokestatic com/colobu/lambda/chapter5/Lambda1.lambda$0:(Ljava/lang/String;)V**

**#62 = MethodType #34 // (Ljava/lang/String;)V**

**#63 = Utf8 InnerClasses**

**#64 = Class #65 // java/lang/invoke/MethodHandles$Lookup**

**#65 = Utf8 java/lang/invoke/MethodHandles$Lookup**

**#66 = Class #67 // java/lang/invoke/MethodHandles**

**#67 = Utf8 java/lang/invoke/MethodHandles**

**#68 = Utf8 Lookup**

{

public com.colobu.lambda.chapter5.Lambda1();

flags: ACC\_PUBLIC

Code:

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

0: aload\_0

1: invokespecial **#8 // Method java/lang/Object."<init>":()V**

4: **return**

LineNumberTable:

line 7: 0

LocalVariableTable:

Start Length Slot Name Signature

0 5 0 **this** Lcom/colobu/lambda/chapter5/Lambda1;

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

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

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

0: invokedynamic **#19, 0 // InvokeDynamic #0:accept:()Ljava/util/function/Consumer;**

5: astore\_1

6: aload\_1

7: ldc **#20 // String hello lambda**

9: invokeinterface **#22, 2 // InterfaceMethod java/util/function/Consumer.accept:(Ljava/lang/Object;)V**

14: **return**

LineNumberTable:

line 10: 0

line 11: 6

line 12: 14

LocalVariableTable:

Start Length Slot Name Signature

0 15 0 args [Ljava/lang/String;

6 9 1 c Ljava/util/function/Consumer;

LocalVariableTypeTable:

Start Length Slot Name Signature

6 9 1 c Ljava/util/function/www.baiyuewang.netConsumer<Ljava/lang/String;>;

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

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

Code:

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

0: getstatic **#35 // Field java/lang/System.out:Ljava/io/PrintStream;**

3: aload\_0

4: invokevirtual **#41 // Method java/io/PrintStream.println:(Ljava/lang/String;)V**

7: **return**

LineNumberTable:

line 10: 0

LocalVariableTable:

Start Length Slot Name Signature

0 8 0 s Ljava/lang/String;

}

可以看到， Lambda表达式体被生成一个称之为lambda$0的方法。 看字节码知道它调用System.out.println输出传入的参数。

原lambda表达式处产生了一条invokedynamic #19, 0。它会调用bootstrap方法。

0: #57 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:

#58 (Ljava/lang/Object;)V

#61 invokestatic com/colobu/lambda/chapter5/Lambda1.lambda$0:(Ljava/lang/String;)V

#62 (Ljava/lang/String;)V

如果Lambda表达式写成Consumer<String> c = (Consumer<String> & Serializable)s -> System.out.println(s);, 则BootstrapMethods的字节码为

**BootstrapMethods:**

0: #108 invokestatic java/lang/invoke/LambdaMetafactory.altMetafactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;[Ljava/lang/Object;)Ljava/lang/invoke/CallSite;

Method arguments:

#109 (Ljava/lang/Object;)V

#112 invokestatic com/colobu/lambda/chapter5/Lambda1.lambda$0:(Ljava/lang/String;)V

#113 (Ljava/lang/String;)V

#114 1

它调用的是LambdaMetafactory.altMetafactory,和上面的调用的方法不同。#114 1意味着要实现Serializable接口。

如果Lambda表达式写成``,则BootstrapMethods的字节码为

**BootstrapMethods:**

0: #57 invokestatic java/lang/invoke/LambdaMetafactory.altMetafactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;[Ljava/lang/Object;)Ljava/lang/invoke/CallSite;

Method arguments:

#58 (Ljava/lang/Object;)V

#61 invokestatic com/colobu/lambda/chapter5/Lambda1.lambda$0:(Ljava/lang/String;)V

#62 (Ljava/lang/String;)V

#63 2

#64 1

#65 com/colobu/lambda/chapter5/ABC

#63 2意味着要实现额外的接口。#64 1意味着要实现额外的接口的数量为1。

字节码的指令含义可以参考这篇文章：[Java bytecode instruction listings](http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings)。

可以看到， Lambda表达式具体的转换是通过java.lang.invoke.LambdaMetafactory.metafactory实现的， 静态参数依照lambda表达式和目标类型不同而不同。

**LambdaMetafactory.metafactory**

现在我们可以重点关注以下 LambdaMetafactory.metafactory的实现。

**public** **static** CallSite **metafactory**(MethodHandles.Lookup caller,

String invokedName,

MethodType invokedType,

MethodType samMethodType,

MethodHandle implMethod,

MethodType instantiatedMethodType)

**throws** LambdaConversionException {返回值类型

AbstractValidatingLambdaMetafactory mf;

mf = **new** InnerClassLambdaMetafactory(caller, invokedType,

invokedName, samMethodType,

implMethod, instantiatedMethodType,

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

mf.validateMetafactoryArgs();

**return** mf.buildCallSite();

}

实际是由InnerClassLambdaMetafactory的buildCallSite来生成。 生成之前会调用validateMetafactoryArgs方法校验目标类型(SAM)方法的参数/和产生的方法的参数/返回值类型是否一致。

metaFactory方法的参数：

* caller: 由JVM提供的lookup context
* invokedName: JVM提供的NameAndType
* invokedType: JVM提供的期望的CallSite类型
* samMethodType: 函数式接口定义的方法的签名
* implMethod: 编译时产生的那个实现方法
* instantiatedMethodType: 强制的方法签名和返回类型， 一般和samMethodType相同或者是它的一个特例

上面的代码基本上是InnerClassLambdaMetafactory.buildCallSite的包装，下面看看这个方法的实现：

CallSite **buildCallSite**() **throws** LambdaConversionException {

**final** Class<?> innerClass = spinInnerClass();

**if** (invokedType.parameterCount() == 0) {

..... //调用构造函数初始化一个SAM的实例

**return** **new** ConstantCallSite(MethodHandles.www.baba120.comconstant(samBase, inst));

} **else** {

UNSAFE.ensureClassInitialized(innerClass);

**return** **new** ConstantCallSite(

MethodHandles.Lookup.IMPL\_LOOKUP

.findStatic(innerClass, NAME\_FACTORY, invokedType));

}

}

其中spinInnerClass调用asm框架动态的产生SAM的实现类， 这个实现类的的方法将会调用编译时产生的那个实现方法。  
你可以在编译的时候加上参数-Djdk.internal.lambda.dumpProxyClasses, 这样编译的时候会自动产生运行时spinInnerClass产生的类。

你可以访问OpenJDK的bug系统了解这个功能。 [JDK-8023524](https://bugs.openjdk.java.net/browse/JDK-8023524)

**重复的lambda表达式**

下面的代码中，在一个循环中重复生成调用lambda表达式，只会生成同一个lambda对象， 因为只有同一个invokedynamic指令。

**for** (int i = 0; i<100; i++){

**Consumer**<**String**> c = s -> **System**.out.println(s);

**System**.out.println(c.hashCode());

}

但是下面的代码会生成两个lambda对象, 因为它会生成两个invokedynamic指令。

**Consumer**<**String**> c = s -> **System**.out.println(s);

**System**.out.println(c.hashCode());

**Consumer**<**String**> c2 = s -> **System**.out.println(s);

**System**.out.println(c2.hashCode());

**生成的类名**

既然LambdaMetafactory会使用asm框架生成一个匿名类， 那么这个类的类名有什么规律的。

**Consumer**<**String**> c = s -> **System**.out.println(s);

**System**.out.println(c.getClass().getName());

**System**.out.println(c.getClass().getSimpleName());

**System**.out.println(c.getClass().getCanonicalName());

输出结果如下：

com.colobu.lambda.chapter5.Lambda3$$Lambda$1/640070680

Lambda3$$Lambda$1/640070680

com.colobu.lambda.chapter5.Lambda3$$Lambda$1/640070680

类名格式如 <包名>.<类名>$$Lambda$/.

number是由一个计数器生成counter.incrementAndGet()。

后缀/<NN>中的数字是一个hash值, 那就是类对象的hash值c.getClass().hashCode()。

在Klass::external\_name()中生成。

sprintf(hash\_buf, "/" UINTX\_FORMAT, (uintx)hash);

**直接调用生成的方法**

上面提到， Lambda表达式体会由编译器生成一个方法，名字格式如Lambda$XXX。

既然是类中的实实在在的方法，我们就可以直接调用。当然， 你在代码中直接写lambda$0()编译通不过， 因为Lambda表达式体还没有被抽取成方法。

但是在运行中我们可以通过反射的方式调用。 下面的例子使用发射和MethodHandle两种方式调用这个方法。

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

**Consumer**<**String**> c = s -> **System**.out.println(s);

**Method** m = **Lambda4**.**class**.getDeclaredMethod("lambda$0", **String**.**class**);

m.invoke(null, "hello reflect");

**MethodHandle** mh = **MethodHandles**.lookup().findStatic(**Lambda4**.**class**, "lambda$0", **MethodType**.methodType(void.**class**, **String**.**class**));

mh.invoke("hello MethodHandle");

}

**捕获的变量等价于’final’**

我们知道，在匿名类中调用外部的参数时，参数必须声明为final。

Lambda体内也可以引用上下文中的变量，变量可以不声明成final的，但是必须等价于final。

下面的例子中变量capturedV等价与final， 并没有在上下文中重新赋值。

**public** **class** **Lambda5** {

**String** greeting = "hello";

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

**Lambda5** capturedV = new **Lambda5**();

**Consumer**<**String**> c = s -> **System**.out.println(capturedV.greeting + " " + s);

c.accept("captured variable");

//capturedV = null; //Local variable capturedV defined in an enclosing scope must be final or effectively final

//capturedV.greeting = "hi";

}

}

如果反注释capturedV = null;编译出错，因为capturedV在上下文中被改变。

但是如果反注释capturedV.greeting = "hi"; 则没问题， 因为capturedV没有被重新赋值， 只是它指向的对象的属性有所变化。

**方法引用**

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

**Consumer**<**String**> c = **System**.out::println;

c.accept("hello");

}

这段代码不会产生一个类似”Lambda$0″新方法。 因为LambdaMetafactory会直接使用这个引用的方法。

**BootstrapMethods:**

0: #51 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:

#52 (Ljava/lang/Object;)V

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

#60 (Ljava/lang/String;)V

#59指示实现方法为System.out::println