开始学习 Lambda 表达式时以为它不过是匿名内部类的一个语法糖，其实里面大有玄机。这篇文章我们来介绍一下 Lambda 表达式与字节码的关系。

0x01 测试匿名内部类的实现

```
Test.java
public static void main(String[] args) {
     Runnable r1 = new Runnable() {
        @Override
        public void run() {
            System.out.println("hello, inner
class");
        }
    };
    r1.run();
}
```

使用 javac 进行编译会生成两个 class 文件`Test.class`和`Test$1.class`

main 函数简化过的字节码如下：

```
public static void main(java.lang.String[]);
Code:
  stack=2, locals=2, args_size=1
     0: new           #2                  // class Test$1
     3: dup
     4: invokespecial #3                  // Method Test$1."<init>":()V
     7: astore_1
     8: aload_1
     9: invokeinterface #4,  1            // InterfaceMethod java/lang/Runnable.run:()V
    14: return
```

- 第 0 ~ 7 行：新建Test$1实例对象
- 第 8 ~ 9 行：执行 Test$1 对象的 run 方法

整个过程的伪代码就是

```
class Test$1 implements Runnable {
    public Test$1(Test test) {
    }

    @Override
    public void run() {
        System.out.println("hello, inner class");
    }
}
public class Test {
    public static void main(String[] args) {
        Runnable r1 = new Test$1(this);
        r1.run();
    }
}
```

因此可以得出结论：匿名内部类是在编译期间生成新的 class 文件来实现的。

0x02 测试 lambda 表达式

还是上面的代码，修改为 lambda 的方式

```
public static void main(String[] args) {
    Runnable r = ()->{
        System.out.println("hello, lambda");
    };
    r.run();
}
```

继续使用 javac 编译，发现这次只生成了 Test.class 一个类文件，并没有生成匿名内部类，使用`javap -p -s -c -v -l Test`查看对应字节码如下

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=1, locals=2, args_size=1
     0: invokedynamic #2,  0              //
InvokeDynamic #0:run:()Ljava/lang/Runnable;
     5: astore_1
     6: aload_1
     7: invokeinterface #3,  1            //
InterfaceMethod java/lang/Runnable.run:()V
    12: return

private static void lambda$main$0();
Code:
     0: getstatic     #4                  //
Field java/lang/System.out:Ljava/io/PrintStream;
     3: ldc           #5                  //
String hello, lambda
     5: invokevirtual #6                  //
Method java/io/PrintStream.println:
(Ljava/lang/String;)V
     8: return
```

出乎意料的出现了一个名为 lambda$main$0 的静态方法，这段字节码比较简单，人肉翻译一下就是

```
private static void lambda$main$0() {
    System.out.println("hello, lambda");
}
```

这里 main 函数中出现了上一节最后介绍的神奇指令 invokedynamic。第 0 行中 #2 表示常量池中#2，它又指向了 #0:#23

```
Constant pool:
   #1 = Methodref          #8.#18         //
java/lang/Object."<init>":()V
   #2 = InvokeDynamic      #0:#23         //
#0:run:()Ljava/lang/Runnable;
   ...
   #23 = NameAndType        #35:#36        //
run:()Ljava/lang/Runnable;

BootstrapMethods:
  0: #20 invokestatic
java/lang/invoke/LambdaMetafactory.metafactory:
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lan
g/String;Ljava/lang/invoke/MethodType;Ljava/lang/
invoke/MethodType;Ljava/lang/invoke/MethodHandle;
Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/C
allSite;
    Method arguments:
      #21 ()V
      #22 invokestatic Test.lambda$main$0:()V
      #21 ()V
```

其中#0是一个特殊的查找，对应 BootstrapMethods 中的 0 行，可以看到这是一个对静态方法 LambdaMetafactory.metafactory() 的调用，它的返回值是 java.lang.invoke.CallSite 对象，这个对象代表了真正执行的目标方法调用。

核心的 metafactory 函数定义如下

```
public static CallSite metafactory(
    MethodHandles.Lookup caller,
    String invokedName,
    MethodType invokedType,
    MethodType samMethodType,
    MethodHandle implMethod,
    MethodType instantiatedMethodType
)
```

各个参数的含义如下

- caller：JVM 提供的查找上下文
- invokedName：表示调用函数名，在本例中 invokedName 为 "run"
- samMethodType：函数式接口定义的方法签名（参数类型和返回值类型），本例中为 run 方法的签名 "()void"
- implMethod：编译时生成的 lambda 表达式对应的静态方法 `invokestatic Test.lambda$main$0`
- instantiatedMethodType：一般和 samMethodType 是一样或是它的一个特例，在本例中是 "()void"

metafactory 方法的内部细节是整个 lambda 表达式最复杂的地方。它的源码如下

跟进 InnerClassLambdaMetafactory 类，看到它在默默生成新的内部类，类名的规则是`ClassName$Lambda$1`，其中 ClassName 是 lambda 所在的类名，后面的数字按生成的顺序依次递增。

同样可以使用打印的方式看一下具体生成的类名

```
Runnable r = ()->{
    System.out.println("hello, lambda");
};

System.out.println(r.getClass().getName());

输出:
Test$Lambda$1/1108411398
```

其中斜杠/后面的数字 1108411398 是类对象的 hashcode 值。

InnerClassLambdaMetafactory 这个类的静态初始化方法块里有一个开关可以选择是否把生成的类 dump 到磁盘中。

```
// For dumping generated classes to disk, for
debugging purposes
private static final ProxyClassesDumper dumper;
static {
        final String key =
"jdk.internal.lambda.dumpProxyClasses";
        String path =
AccessController.doPrivileged(
                new GetPropertyAction(key), null,
                new PropertyPermission(key ,
"read"));
        dumper = (null == path) ? null :
ProxyClassesDumper.getInstance(path);
    }
```

使用java -Djdk.internal.lambda.dumpProxyClasses=. Test运行 Test 类会发现在运行期间生成了一个新的内部类：Test$Lambda$1.class。这个类正是由InnerClassLambdaMetafactory使用 ASM 字节码技术动态生成的，只是默认情况看不到而已。

这个类实现了 Runnable 接口，并在 run 方法里调用了 Test 类的静态方法lambda$main$0()。

把这个类的字节码人肉翻译过来是下面这样

```
final class Test$Lambda$1 implements Runnable {
    @Override
    public void run() {
        Test.lambda$main$0();
    }
}
```

整个过程就比较明朗了：

- lambda 表达式声明的地方会生成一个 invokedynamic 指令，同时编译器生成一个对应的引导方法（Bootstrap Method）
- 第一次执行 invokedynamic 指令时，会调用对应的引导方法（Bootstrap Method），该引导方法会调用 LambdaMetafactory.metafactory 方法动态生成内部类
- 引导方法会返回一个动态调用 CallSite 对象，这个 CallSite 会链接最终调用的实现了 Runnable 接口的内部类
- lambda 表达式中的内容会被编译成静态方法，前面动态生成的内部类会直接调用该静态方法
- 真正执行 lambda 调用的还是用 invokeinterface 指令

0x03 为什么 Java 8 的 Lambda 表达式要基于 invokedynamic

关于为什么用 invokedynamic 来实现 Lambda，Oracle 的开发者专门写了一篇文章
Translation of Lambda Expressions

(http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html)，介绍了 Java 8 Lambda 设计时的考虑以及实现方法。
文中提到 Lambda 表达式可以通过内部类、method handle、dynamic proxies 等方式实现，但是这些方法各有优劣。实现 Lambda 表达式需要达成两个目标：

- 为未来的优化提供最大的灵活性
- 保持类文件字节码格式的稳定

使用 invokedynamic 可以很好的兼顾这两个目标。

原文如下

> here are a number of ways we might represent a lambda expression in bytecode, such as inner classes, method handles, dynamic proxies, and others. Each of these approaches has pros and cons. In selecting a strategy, there are two competing goals: maximizing flexibility for future optimization by not committing to a specific strategy, vs providing stability in the classfile representation.

invokedynamic 与之前四个 invoke 指令最大的不同就在于它把方法分派的逻辑从虚拟机层面下放到程序语言。
lambda 表达式采用的方式是不在编译期间就生成匿名内部类，而是提供了一个稳定的字节码二进制表示规范，对用户而言看到的只有 invokedynamic 这样一个非常简单的指令。用 invokedynamic 来实现就是把翻译的逻辑隐藏在 JDK 的实现中，后续想替换实现方式

非常简单，只用修改 LambdaMetafactory.metafactory 里面的逻辑就可以了，这种方法把 lambda 翻译的策略由编译期间推迟到运行时。

0x04 小结

lambda 表达式与普通的匿名内部类的实现方式不一样，在编译阶段只是新增了一个 invokedynamic 指令，并没有在编译期间生成匿名内部类，lambda 表达式的内容会被编译成一个静态方法。在运行时 LambdaMetafactory.metafactory 这个工厂方法来动态生成一个内部类 class，该内部类会调用前面生成的静态方法。
lambda 表达式最终还是会生成一个内部类，只不过是不是在编译期间而是在运行时，未来的 JDK 会怎么实现 Lambda 表达式可能还会有变化。

0x05 思考题

下面的两段代码分别会生成多少个内部类？为什么？

代码片段 1

```
for (int i = 0; i < 10; i++) {
    Runnable r = () -> {
        System.out.println("hello, lambda");
    };
    r.run();
}
```

代码片段 2

```
Runnable r1 = () -> {
    System.out.println("hello, lambda");
};
r1.run();

Runnable r2 = () -> {
    System.out.println("hello, lambda");
};
r2.run();
```