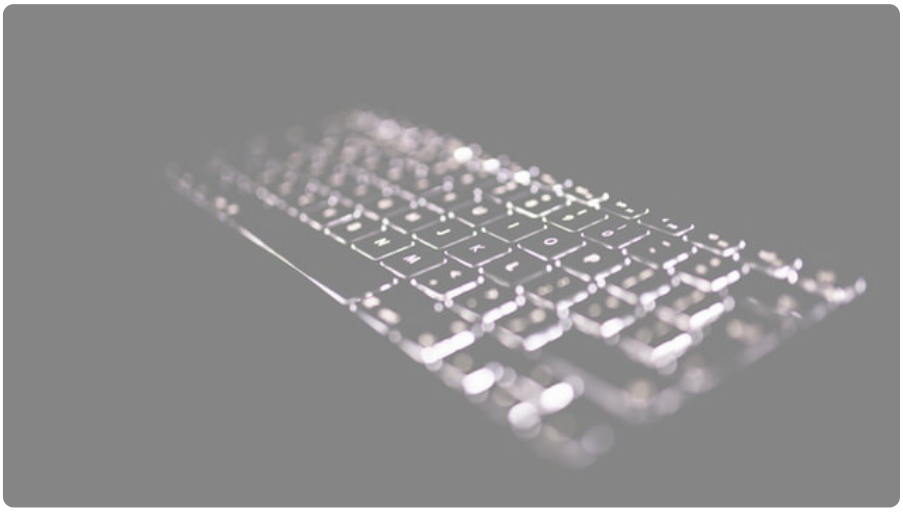
目录

41 突破难点：如何看 Lambda 源码

更新时间：2019-11-25 10:00:14



耐心和恒心总会得到报酬的。

——爱因斯坦

引导语

大家都知道 Java8 中新增了 Lambda 表达式，使用 Lambda 表达式可以对代码进行大量的优化，用几行代码就可以做很多事情，本章以 Lambda 为例，第一小节说明一下其底层的执行原理，第二小节说明一下 Lambda 流在工作中常用的姿势。

1 Demo

首先我们来看一个 Lambda 表达式的 Demo，如下图：

Lambda.java

```
package demo.eight;

/**
 * Lambda
 *author  wenhe
 *date 2019/10/12
 */
public class Lambda {

    public static void simple() {
        Runnable runnable = () -> System.out.println("lambda is run");
        runnable.run();
    }

    public static void main(String[] args) throws Exception {
        simple();
    }

}
```

目录

如果我们修改成匿名内部类的写法，就很清楚，大家都能看懂，如下图：

```
/**
 * Lambda
 *author  wenhe
 *date 2019/10/12
 */
public class Lambda {

  public static void simple() {
//    Runnable runnable = () -> System.out.println("lambda is run");
    Runnable runnable = new Runnable() {
      @Override
      public void run() {
        System.out.println("lambda is run");
      }
    };
    runnable.run();
  }

  public static void main(String[] args) throws Exception {
    simple();
  }

}
```

那是不是说 () -> System.out.println (“ lambda is run “) 这种形式的代码，其实就是建立了内部类呢？其实这就是最简单 Lambda 表达式，我们是无法通过 IDEA 看到源码和其底层结构的，下面我们就来介绍几种可看到其底层实现的方式。

2 异常判断法

我们可以在代码执行中主动抛出异常，打印出堆栈，堆栈会说明其运行轨迹，一般这种方法简单高效，基本上可以看到很多情况下的隐藏代码，我们来试一下，如下图：

目录

```
*date 2019/10/12
*/
public class Lambda {

  public static void simple() {
    Runnable runnable = () -> {
      System.out.println("lambda is run");
      throw new RuntimeException("exception");
    };
    runnable.run();
  }
  public static void main(String[] args) throws Exception {
    simple();
  }
}
```

```
Run: Lambda (1)
/Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/
lambda is run
Exception in thread "main" java.lang.RuntimeException: exception
    at demo.eight.Lambda.lambda$simple$0(Lambda.java:13)
    at demo.eight.Lambda$$Lambda$1/1023892928.run(Unknown Source)
    at demo.eight.Lambda.simple(Lambda.java:15)
    at demo.eight.Lambda.main(Lambda.java:18)

Process finished with exit code 1
```

从异常的堆栈中，我们可以看到 JVM 自动给当前类建立了内部类（错误堆栈中出现多次的 $ 表示有内部类），内部类的代码在执行过程中，抛出了异常，但这里显示的代码是 Unknown Source，所以我们也无法 debug 进去，一般情况下，异常都能暴露出代码执行的路径，我们可以打好断点后再次运行，但对于 Lambda 表达式而言，通过异常判断法我们只清楚有内部类，但无法看到内部类中的源码。

3 javap 命令法

javap 是 Java 自带的可以查看 class 字节码文件的工具，安装过 Java 基础环境的电脑都可以直接执行 javap 命令，如下图：

```
luanqiudeMacBook-Pro:~ luanqiu$ javap
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
```

命令选项中，我们主要是用-v -verbose 这个命令，可以完整输出字节码文件的内容。

接下来我们使用 javap 命令查看下 Lambda.class 文件，在讲解的过程中，我们会带上一些关于 class 文件的知识。

目录

所有的参考资料来自 Java 虚拟机规范，不再一一引用说明）：

汇编指令中我们很容易找到 Constant pool 打头的一长串类型，我们叫做常量池，官方英文叫做 Run-Time Constant Pool，我们简单理解成一个装满常量的 table ， table 中包含编译时明确的数字和文字，类、方法和字段的类型信息等等。table 中的每个元素叫做 cp*info*，*cp*info 由唯一标识 (tag) + 名称组成，目前 tag 的类型一共有：

Table 4.4-A. Constant pool tags

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1
CONSTANT_MethodHandle	15
CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

贴出我们解析出来的部分图：

1. 图中 Constant pool 字样代表当前信息是常量池；

2. 每行都是一个 cp_info ，第一列的 #1 代表是在常量池下标为 1 的位置；

目录

法的描述信息的，比如说方法的名称，入参类型，出参数类型等，具体的含义在 Java 虚拟机规范中都可以查询到，Methodref 的截图如下：

4. 每行的第三列，如果是具体的值的话，直接显示具体的值，如果是复杂的值的话，会显示 cp_info 的引用，比如说图中标红 2 处，引用两个 13 和 14 位置的 cp_info，13 表示方法名字是 init，14 表示方法无返回值，结合起来表示方法的名称和返回类型，就是一个无参构造器；
5. 每行的第四列，就是具体的值了。

对于比较重要的 cp_info 类型我们说明下其含义：

1. InvokeDynamic 表示动态的调用方法，后面我们会详细说明；
2. Fieldref 表示字段的描述信息，如字段的名称、类型；
3. NameAndType 是对字段和方法类型的描述；
4. MethodHandle 方法句柄，动态调用方法的统称，在编译时我们不知道具体是那个方法，但运行时肯定会知道调用的是那个方法；
5. MethodType 动态方法类型，只有在动态运行时才会知道其方法类型是什么。

我们从上上图中标红的 3 处，发现 Ljava/lang/invoke/MethodHandles$Lookup，java/lang/invoke/LambdaMetafactory.metafactory 类似这样的代码，MethodHandles 和 LambdaMetafactory 都是 java.lang.invoke 包下面的重要方法，invoke 包主要实现了动态语言的功能，我们知道 java 语言属于静态编译语言，在编译的时候，类、方法、字段等等的类型都已经确定了，而 invoke 实现的是一种动态语言，也就是说编译的时候并不知道类、方法、字段是什么类型，只有到运行的时候才知道。

比如这行代码：Runnable runnable = () -> System.out.println(“lambda is run”); 在编译器编译的时候 () 这个括号编译器并不知道是干什么的，只有在运行的时候，才会知道原来这代表着的是 Runnable.run() 方法。invoke 包里面很多类，都是为了代表这些 () 的，我们称作为方法句柄（MethodHandler），在编译的时候，编译器只知道这里是个方法句柄，并不知道实际上执行什么方法，只有在执行的时候才知道，那么问题来了，JVM 执行的时候，是如何知道 () 这个方法句柄，实际上是执行 Runnable.run() 方法的呢？

目录

```
#72 = Class              #74            // java/lang/invoke/MethodHandles
#73 = Utf8               java/lang/invoke/MethodHandles$Lookup
#74 = Utf8               java/lang/invoke/MethodHandles
{
public demo.eight.Lambda();
  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 8: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   Ldemo/eight/Lambda;

public static void simple();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=0
       0: invokedynamic #2,  0              // InvokeDynamic #0:run:()Ljava/lang/Runnable;
       5: astore_0
       6: aload_0
       7: invokeinterface #3,  1            // InterfaceMethod java/lang/Runnable.run:()V
      12: return
    LineNumberTable:
      line 11: 0
      line 15: 6
      line 16: 12
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          6       7     0 runnable   Ljava/lang/Runnable;
```

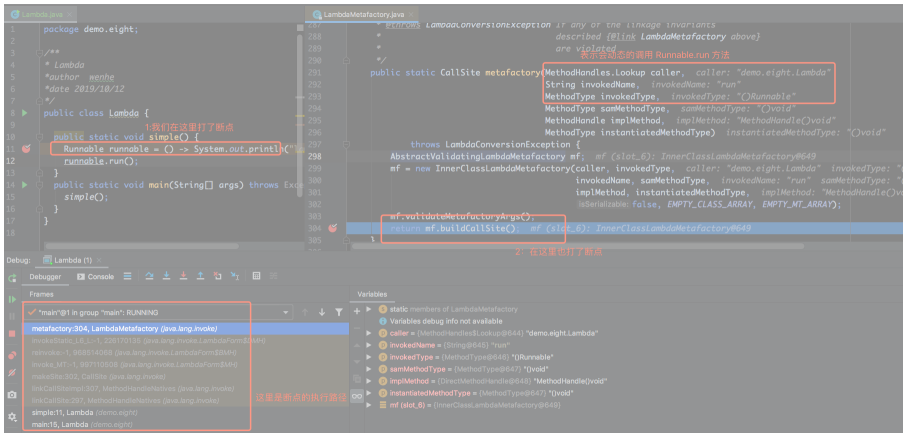
常量池下面就是方法区

动态执行 Runnable 的 run 方法

从上图中就可以看出 simple 方法中的 () -> System.out.println(“lambda is run”) 代码中的 ()，实际上就是 Runnable.run 方法。

我们追溯到 # 2 常量池，也就是上上图中标红 1 处，InvokeDynamic 表示这里是个动态调用，调用的是两个常量池的 cp_info，位置是 #0:#37 ，我们往下找 #37 代表着是 // run:()Ljava/lang/Runnable，这里表明了在 JVM 真正执行的时候，需要动态调用 Runnable.run() 方法，从汇编指令上我们可以看出 () 实际上就是 Runnable.run()，下面我们 debug 来证明一下。

我们在上上图中 3 处发现了 LambdaMetafactory.metafactory 的字样，通过查询官方文档，得知该方法正是执行时， 链接到真正代码的关键，于是我们在 metafactory 方法中打个断点 debug 一下，如下图：



metafactory 方法入参 caller 代表实际发生动态调用的位置，invokedName 表示调用方法名称，invokedType 表示调用的多个入参和出参，samMethodType 表示具体的实现者的参数，implMethod 表示实际上的实现者，instantiatedMethodType 等同于 implMethod。

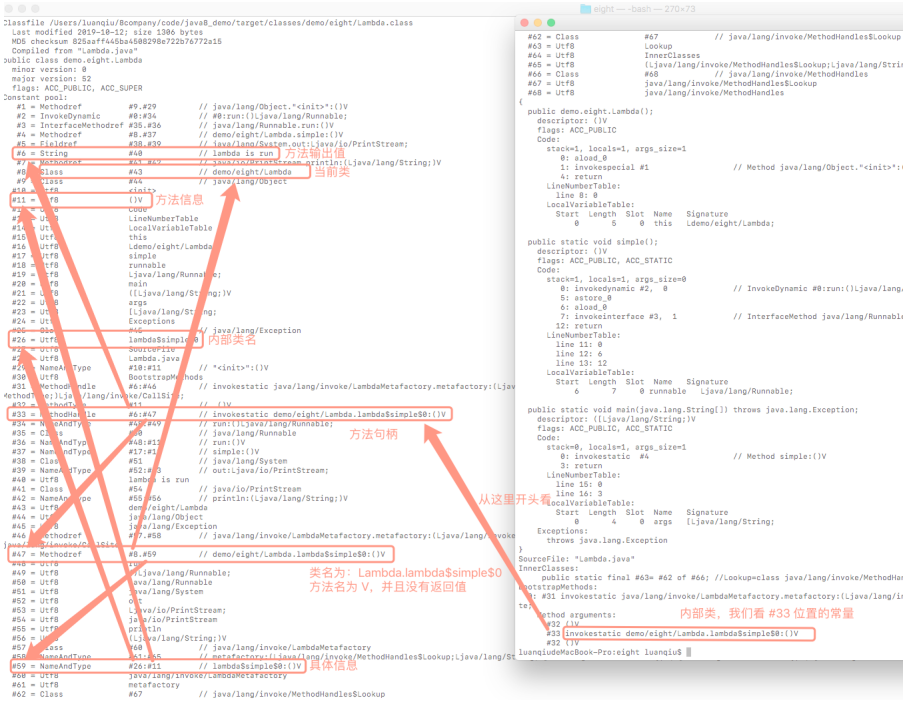
以上内容总结一下：

1：从汇编指令的 simple 方法中，我们可以看到会执行 Runnable.run 方法；

目录

所以可以把 Lambda 表达值的具体执行归功于 invokedynamic JVM 指令，正是因为这个指令，才可以做到虽然编译时不知道要干啥，但动态运行时却能找到具体要执行的代码。

接着我们看一下在汇编指令输出的最后，我们发现了异常判断法中发现的内部类，如下图：



上图中箭头很多，一层一层的表达清楚了当前内部类的所有信息。

4 总结

我们总结一下，Lambda 表达式执行主要是依靠 invokedynamic 的 JVM 指令来实现，咱们演示的类的全路径为：demo.eight.Lambda 感兴趣的同学可以自己尝试一下。

← 40 打动面试官：线程池流程编排中的运用实战

42 常用的 Lambda 表达式使用场景解析和应用 →

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论

目录