# 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，InvokeDynamic的操作码是186（0xBA），格式是：

***invokedynamic indexbyte1 indexbyte2 0 0***

InvokeDynamic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InvokeDynamic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。

InvokeDynamic 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InvokeDynamic 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

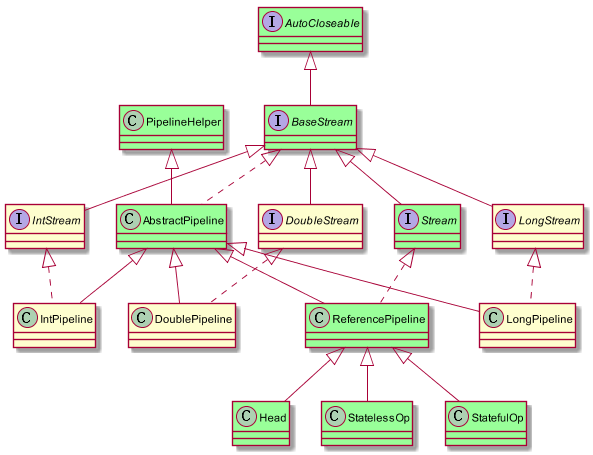
## 操作分类

|  |  |  |
| --- | --- | --- |
| Stream操作分类 | | |
| 中间操作  (Intermediate operations) | 无状态  (Stateless) | unordered() filter() map() mapToInt() mapToLong() mapToDouble() flatMap() flatMapToInt() flatMapToLong() flatMapToDouble() peek() |
| 有状态  (Stateful) | distinct() sorted() sorted() limit() skip() |
| 结束操作  (Terminal operations) | 非短路操作 | forEach() forEachOrdered() toArray() reduce() collect() max() min() count() |
| 短路操作  (short-circuiting) | anyMatch() allMatch() noneMatch() findFirst() findAny() |

Stream中的操作可以分为两大类：**中间操作**（Intermediate operations）与**结束操作**（Terminal operations），中间操作只是对操作进行了记录，只有结束操作才会触发实际的计算（即**惰性求值**），这也是Stream在迭代大集合时高效的原因之一。中间操作又可以分为**无状态**（Stateless）操作与**有状态**（Stateful）操作，前者是指元素的处理不受之前元素的影响；后者是指该操作只有拿到所有元素之后才能继续下去。结束操作又可以分为**短路**（short-circuiting）与非短路操作，这个应该很好理解，前者是指遇到某些符合条件的元素就可以得到最终结果；而后者是指必须处理所有元素才能得到最终结果。

之所以要进行如此精细的划分，是因为底层对每一种情况的处理方式不同。

## 包结构概览



* BaseStream定义了流的迭代、并行、串行等基本特性；
* Stream中定义了map、filter、flatmap等用户关注的常用操作；
* PipelineHelper用于执行管道流中的操作以及捕获输出类型、并行度等信息
* Head、StatelessOp、StatefulOp为ReferencePipeline中的内部子类，用于描述流的操作阶段。

## 源码分析

来看一个例子：

List<String> list = Arrays.asList("China", "America", "Russia", "Britain");

List<String> result = list.stream()

.filter(e -> e.length() >= 4)

.map(e -> e.charAt(0))

.map(e -> String.valueOf(e))

.collect(Collectors.toList());

上面List首先生成了一个stream，然后经过filter、map、map三次无状态的中间操作，最后由最终操作collect收尾。

下面通过源码来一次庖丁解牛，看看一步步到底是怎么实现的。

生成流的操作是通过调用StreamSupport类下面的方法实现的：

### Stream()

public static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel) {

Objects.requireNonNull(spliterator);

return new ReferencePipeline.Head<>(spliterator,

StreamOpFlag.fromCharacteristics(spliterator),

parallel);

}

方法很简单，直接new了一个ReferencePipeline.Head对象并返回。Head是ReferencePipeline的子类，而ReferencePipeline是Stream的子类。也就是说，返回了一个由Head实现的Stream。

追溯源码可以发现，Head最终通过调用父类ReferencePipeline的构造方法完成实例化：

public static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel) {

Objects.requireNonNull(spliterator);

//返回了一个由Head实现的Stream，三个参数分别代表流的数据源、特性组合、是否并行

return new ReferencePipeline.Head<>(spliterator,

StreamOpFlag.fromCharacteristics(spliterator),

parallel);

}

AbstractPipeline(Spliterator<?> source,

int sourceFlags, boolean parallel) {

this.previousStage = null; //上一个stage指向null

this.sourceSpliterator = source;

this.sourceStage = this; //源头stage指向自己

this.sourceOrOpFlags = sourceFlags & StreamOpFlag.STREAM\_MASK;

// The following is an optimization of:

// StreamOpFlag.combineOpFlags(sourceOrOpFlags, StreamOpFlag.INITIAL\_OPS\_VALUE);

this.combinedFlags = (~(sourceOrOpFlags << 1)) & StreamOpFlag.INITIAL\_OPS\_VALUE;

this.depth = 0;

this.parallel = parallel;

}

AbstractPipeline类中定义了三个称为“stage”内部变量：

/\*\*

\* Backlink to the head of the pipeline chain (self if this is the source

\* stage).

\*/

@SuppressWarnings("rawtypes")

private final AbstractPipeline sourceStage;

/\*\*

\* The "upstream" pipeline, or null if this is the source stage.

\*/

@SuppressWarnings("rawtypes")

private final AbstractPipeline previousStage;

/\*\*

\* The next stage in the pipeline, or null if this is the last stage.

\* Effectively final at the point of linking to the next pipeline.

\*/

@SuppressWarnings("rawtypes")

private AbstractPipeline nextStage;

当前节点同时持有前一个节点与后一个节点的指针，并且保留了头结点的引用，这不是典型的双端链表吗？

基于此，分析上面的构造函数：

* 前一个节点为空
* 头结点指向自己
* 后一个节点暂时未指定

很显然，构造出的是一个双端列表的头结点

综上所述，stream函数返回了一个由ReferencePipeline类实现的管道流，且该管道流为一个双端链表的头结点

### filter()

再来看第二步，filter操作，具体实现在ReferencePipeline的如下方法：

public final Stream<P\_OUT> filter(Predicate<? super P\_OUT> predicate) {

//入参不能为空

Objects.requireNonNull(predicate);

//构建了一个StatelessOp对象，即无状态的中间操作

return new StatelessOp<P\_OUT, P\_OUT>(this, StreamShape.REFERENCE,

StreamOpFlag.NOT\_SIZED) {

//覆写了父类的一个方法opWrapSink

@Override

Sink<P\_OUT> opWrapSink(int flags, Sink<P\_OUT> sink) {

return new Sink.ChainedReference<P\_OUT, P\_OUT>(sink) {

@Override

public void begin(long size) {

downstream.begin(-1);

}

@Override

public void accept(P\_OUT u) {

if (predicate.test(u))

downstream.accept(u);

}

};

}

};

}

StatelessOp与Head一样，也是ReferencePipeline的内部子类，同样通过调用父类ReferencePipeline的构造方法完成实例化，注意第一个参数，传入的是this，就是将上一步创建的Head对象传入，作为StatelessOp的previousStage

AbstractPipeline(AbstractPipeline<?, E\_IN, ?> previousStage, int opFlags) {

if (previousStage.linkedOrConsumed)

throw new IllegalStateException(MSG\_STREAM\_LINKED);

previousStage.linkedOrConsumed = true;

previousStage.nextStage = this; //前一个stage指向自己

this.previousStage = previousStage; //自己指向前一个stage

this.sourceOrOpFlags = opFlags & StreamOpFlag.OP\_MASK;

this.combinedFlags = StreamOpFlag.combineOpFlags(opFlags, previousStage.combinedFlags);

this.sourceStage = previousStage.sourceStage; //也保留了头结点的引用

if (opIsStateful())

sourceStage.sourceAnyStateful = true;

this.depth = previousStage.depth + 1;

}

filter操作成为了双端链表的第二环。

值得注意的是，构造StatelessOp时，覆写了父类的一个方法opWrapSink，返回了一个Sink对象，作用暂时未知，猜测后面的操作应该会用到

### map()

再来看接下来的map操作：

@Override

@SuppressWarnings("unchecked")

public final <R> Stream<R> map(Function<? super P\_OUT, ? extends R> mapper) {

Objects.requireNonNull(mapper);

return new StatelessOp<P\_OUT, R>(this, StreamShape.REFERENCE,

StreamOpFlag.NOT\_SORTED | StreamOpFlag.NOT\_DISTINCT) {

@Override

Sink<P\_OUT> opWrapSink(int flags, Sink<R> sink) {

return new Sink.ChainedReference<P\_OUT, R>(sink) {

@Override

public void accept(P\_OUT u) {

downstream.accept(mapper.apply(u));

}

};

}

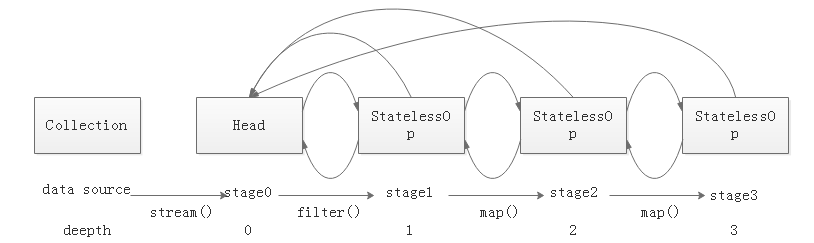
};

}

与filter类似，构造了一个StatelessOp对象，追加到双端列表中的末尾。

不同的地方在于opWrapSink方法的实现，继续猜测，通过覆写opWrapSink，应该可以影响管道流的流程，实现定制化的操作

调用一系列操作后会形成如下所示的双链表结构：



### collect()

最后来看collect操作，不同于filter与map，collect为结束操作，肯定有特殊之处

@Override

@SuppressWarnings("unchecked")

public final <R, A> R collect(Collector<? super P\_OUT, A, R> collector) {

A container;

if (isParallel()

&& (collector.characteristics().contains(Collector.Characteristics.CONCURRENT))

&& (!isOrdered() || collector.characteristics().contains(Collector.Characteristics.UNORDERED))) {

container = collector.supplier().get();

BiConsumer<A, ? super P\_OUT> accumulator = collector.accumulator();

forEach(u -> accumulator.accept(container, u));

}

else { //串行模式

container = evaluate(ReduceOps.makeRef(collector)); //evaluate触发

}

return collector.characteristics().contains(Collector.Characteristics.IDENTITY\_FINISH)

? (R) container

: collector.finisher().apply(container);

}

ReduceOps.makeRef(collector) 会构造一个TerminalOp对象，传入evaluate方法，追溯源码，发现最终是调用copyInto方法来启动流水线：

@Override

final <P\_IN> void copyInto(Sink<P\_IN> wrappedSink, Spliterator<P\_IN> spliterator) {

Objects.requireNonNull(wrappedSink);

if (!StreamOpFlag.SHORT\_CIRCUIT.isKnown(getStreamAndOpFlags())) { //无短路操作

wrappedSink.begin(spliterator.getExactSizeIfKnown());//通知开始遍历

spliterator.forEachRemaining(wrappedSink); //依次处理每个元素

wrappedSink.end();//通知结束遍历

}

else { //有短路操作

copyIntoWithCancel(wrappedSink, spliterator);

}

}

该方法从数据源Spliterator获取的元素，推入Sink进行处理，如果有短路操作，在每个元素处理后会通过Sink.cancellationRequested()判断是否立即返回。

前面的filter、map操作只是做了一系列的准备工作，并没有执行，真正的迭代是由结束操作collect来触发的。

### Sink

Stream中使用Stage的概念来描述一个完整的操作，将具有先后顺序的各个Stage连到一起，就构成了整个流水线。

很多Stream操作会需要一个回调函数（Lambda表达式），因此一个完整的操作是<数据来源，操作，回调函数>构成的三元组。

stage只是解决了操作记录的问题，要想让流水线起到应有的作用我们需要一种将所有操作叠加到一起的方案。你可能会觉得这很简单，只需要从流水线的head开始依次执行每一步的操作（包括回调函数）就行了。这听起来似乎是可行的，但是你忽略了前面的Stage并不知道后面Stage到底执行了哪种操作，以及回调函数是哪种形式。换句话说，只有当前Stage本身才知道该如何执行自己包含的动作。这就需要有某种协议来协调相邻Stage之间的调用关系。

继续从源码找答案。

filter、map源码中，都覆写了一个名为opWrapSink的方法，该方法会返回一个Sink对象，而collect正是通过Sink来处理流中的数据。种种迹象表明，这个名为Sink的类在流的处理流程当中扮演了极其重要的角色。

interface Sink<T> extends Consumer<T> {

//开始遍历元素之前调用该方法，通知Sink做好准备，size代表要处理的元素总数，如果传入-1代表总数未知或者无限

default void begin(long size) {}

//所有元素遍历完成之后调用，通知Sink没有更多的元素了。

default void end() {}

//如果返回true，代表这个Sink不再接收任何数据

default boolean cancellationRequested() {

return false;

}

//还有一个继承自Consumer的方法，用于接收管道流中的数据

//void accept(T t);

...

}

collect操作在调用copyInto方法时，传入了一个名为wrappedSink的参数，就是一个Sink对象，由AbstractPipeline.wrapSin方法构造：

@Override

@SuppressWarnings("unchecked")

final <P\_IN> Sink<P\_IN> wrapSink(Sink<E\_OUT> sink) {

Objects.requireNonNull(sink);

for ( @SuppressWarnings("rawtypes") AbstractPipeline p=AbstractPipeline.this; p.depth > 0; p=p.previousStage) {

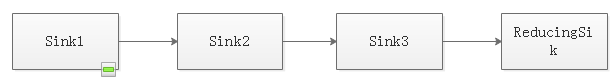
//自本身stage开始，不断调用前一个stage的opWrapSink，直到头节点

sink = p.opWrapSink(p.previousStage.combinedFlags, sink);

}

return (Sink<P\_IN>) sink;

}

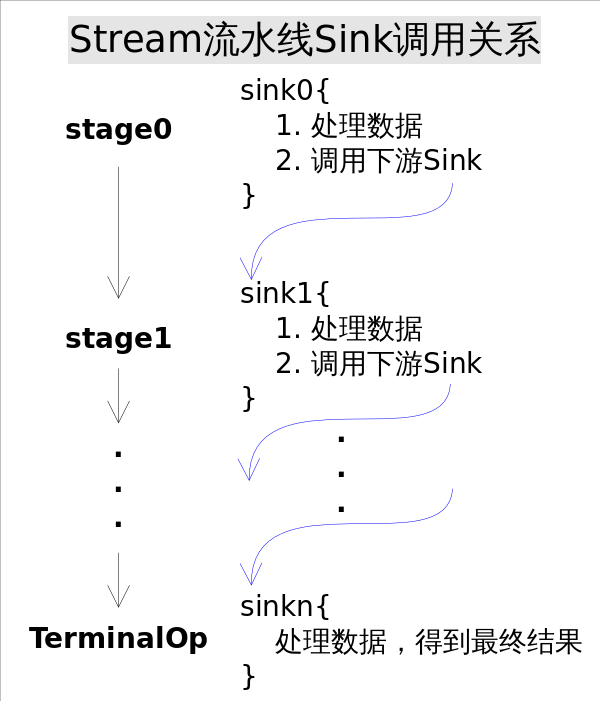


opWrapSink()方法的作用是将当前操作与下游Sink结合成新Sink，试想，只要从流水线的最后一个Stage开始，不断调用上一个Stage的opWrapSink()方法直到头节点，就可以得到一个代表了流水线上所有操作的Sink。

而这个opWrapSink方法不就是前面filter、map源码中一直很神秘的未知操作吗？

至此，任督二脉打通，豁然开朗！

有了上面的协议，相邻Stage之间调用就很方便了，每个Stage都会将自己的操作封装到一个Sink里，前一个Stage只需调用后一个Stage的accept()方法即可，并不需要知道其内部是如何处理的。当然对于有状态的操作，Sink的begin()和end()方法也是必须实现的。比如Stream.sorted()是一个有状态的中间操作，其对应的Sink.begin()方法可能会创建一个盛放结果的容器，而accept()方法负责将元素添加到该容器，最后end()负责对容器进行排序。对于短路操作，Sink.cancellationRequested()也是必须实现的，比如Stream.findFirst()是短路操作，只要找到一个元素，cancellationRequested()就应该返回true，以便调用者尽快结束查找。Sink的四个接口方法常常相互协作，共同完成计算任务。实际上Stream API内部实现的的本质，就是如何重载Sink的这四个接口方法。



有了Sink对操作的包装，Stage之间的调用问题就解决了，执行时只需要从流水线的head开始对数据源依次调用每个Stage对应的Sink.{begin(), accept(), cancellationRequested(), end()}方法就可以了。一种可能的Sink.accept()方法流程是这样的：

void accept(U u){

1. 使用当前Sink包装的回调函数处理u

2. 将处理结果传递给流水线下游的Sink

}

Sink接口的其他几个方法也是按照这种[处理->转发]的模型实现。下面我们结合具体例子看看Stream的中间操作是如何将自身的操作包装成Sink以及Sink是如何将处理结果转发给下一个Sink的。先看Stream.map()方法：

// Stream.map()，调用该方法将产生一个新的Stream

public final <R> Stream<R> map(Function<? super P\_OUT, ? extends R> mapper) {

...

return new StatelessOp<P\_OUT, R>(this, StreamShape.REFERENCE,

StreamOpFlag.NOT\_SORTED | StreamOpFlag.NOT\_DISTINCT) {

@Override /\*opWripSink()方法返回由回调函数包装而成Sink\*/

Sink<P\_OUT> opWrapSink(int flags, Sink<R> downstream) {

return new Sink.ChainedReference<P\_OUT, R>(downstream) {

@Override

public void accept(P\_OUT u) {

R r = mapper.apply(u);// 1. 使用当前Sink包装的回调函数mapper处理u

downstream.accept(r);// 2. 将处理结果传递给流水线下游的Sink

}

};

}

};

}

上述代码看似复杂，其实逻辑很简单，就是将回调函数mapper包装到一个Sink当中。由于Stream.map()是一个无状态的中间操作，所以map()方法返回了一个StatelessOp内部类对象（一个新的Stream），调用这个新Stream的opWripSink()方法将得到一个包装了当前回调函数的Sink。

再来看一个复杂一点的例子。Stream.sorted()方法将对Stream中的元素进行排序，显然这是一个有状态的中间操作，因为读取所有元素之前是没法得到最终顺序的。抛开模板代码直接进入问题本质，sorted()方法是如何将操作封装成Sink的呢？sorted()一种可能封装的Sink代码如下：

// Stream.sort()方法用到的Sink实现

class RefSortingSink<T> extends AbstractRefSortingSink<T> {

private ArrayList<T> list;// 存放用于排序的元素

RefSortingSink(Sink<? super T> downstream, Comparator<? super T> comparator) {

super(downstream, comparator);

}

@Override

public void begin(long size) {

...

// 创建一个存放排序元素的列表

list = (size >= 0) ? new ArrayList<T>((int) size) : new ArrayList<T>();

}

@Override

public void end() {

list.sort(comparator);// 只有元素全部接收之后才能开始排序

downstream.begin(list.size());

if (!cancellationWasRequested) {// 下游Sink不包含短路操作

list.forEach(downstream::accept);// 2. 将处理结果传递给流水线下游的Sink

}

else {// 下游Sink包含短路操作

for (T t : list) {// 每次都调用cancellationRequested()询问是否可以结束处理。

if (downstream.cancellationRequested()) break;

downstream.accept(t);// 2. 将处理结果传递给流水线下游的Sink

}

}

downstream.end();

list = null;

}

@Override

public void accept(T t) {

list.add(t);// 1. 使用当前Sink包装动作处理t，只是简单的将元素添加到中间列表当中

}

}

上述代码完美的展现了Sink的四个接口方法是如何协同工作的：

1. 首先beging()方法告诉Sink参与排序的元素个数，方便确定中间结果容器的的大小；
2. 之后通过accept()方法将元素添加到中间结果当中，最终执行时调用者会不断调用该方法，直到遍历所有元素；
3. 最后end()方法告诉Sink所有元素遍历完毕，启动排序步骤，排序完成后将结果传递给下游的Sink；
4. 如果下游的Sink是短路操作，将结果传递给下游时不断询问下游cancellationRequested()是否可以结束处理。

## 结果收集

最后一个问题是流水线上所有操作都执行后，用户所需要的结果（如果有）在哪里？首先要说明的是不是所有的Stream结束操作都需要返回结果，有些操作只是为了使用其副作用(Side-effects)，比如使用Stream.forEach()方法将结果打印出来就是常见的使用副作用的场景（事实上，除了打印之外其他场景都应避免使用副作用），对于真正需要返回结果的结束操作结果存在哪里呢？

特别说明：副作用不应该被滥用，也许你会觉得在Stream.forEach()里进行元素收集是个不错的选择，就像下面代码中那样，但遗憾的是这样使用的正确性和效率都无法保证，因为Stream可能会并行执行。大多数使用副作用的地方都可以使用归约操作更安全和有效的完成。

// 错误的收集方式

ArrayList<String> results = new ArrayList<>();

stream.filter(s -> pattern.matcher(s).matches())

.forEach(s -> results.add(s)); // Unnecessary use of side-effects!

// 正确的收集方式

List<String>results =

stream.filter(s -> pattern.matcher(s).matches())

.collect(Collectors.toList()); // No side-effects!

回到流水线执行结果的问题上来，需要返回结果的流水线结果存在哪里呢？这要分不同的情况讨论，下表给出了各种有返回结果的Stream结束操作。

|  |  |
| --- | --- |
| 返回类型 | 对应的结束操作 |
| boolean | anyMatch() allMatch() noneMatch() |
| Optional | findFirst() findAny() |
| 归约结果 | reduce() collect() |
| 数组 | toArray() |

* 对于表中返回boolean或者Optional的操作的操作，由于值返回一个值，只需要在对应的Sink中记录这个值，等到执行结束时返回就可以了。
* 对于归约操作，最终结果放在用户调用时指定的容器中（容器类型通过收集器指定）。collect(), reduce(), max(), min()都是归约操作，虽然max()和min()也是返回一个Optional，但事实上底层是通过调用reduce()方法实现的。
* 对于返回是数组的情况，在最终返回数组之前，结果其实是存储在一种叫做Node的数据结构中的。Node是一种多叉树结构，元素存储在树的叶子当中，并且一个叶子节点可以存放多个元素。这样做是为了并行执行方便。

## 并行流

如果将上面的例子改为如下形式，管道流将会以并行模式处理数据：

List<String> list = Arrays.asList("China", "America", "Russia", "Britain");

List<String> result = list.stream()

.parallel()

.filter(e -> e.length() >= 4)

.map(e -> e.charAt(0))

.map(e -> String.valueOf(e))

.collect(Collectors.toList());

parallel()方法的实现很简单，只是将源stage的并行标记只为true：

@Override

@SuppressWarnings("unchecked")

public final S parallel() {

sourceStage.parallel = true;

return (S) this;

}

在结束操作通过evaluate方法启动管道流时，会根据并行标记来判断：

final <R> R evaluate(TerminalOp<E\_OUT, R> terminalOp) {

assert getOutputShape() == terminalOp.inputShape();

if (linkedOrConsumed)

throw new IllegalStateException(MSG\_STREAM\_LINKED);

linkedOrConsumed = true;

return isParallel()

? terminalOp.evaluateParallel(this, sourceSpliterator(terminalOp.getOpFlags()))

: terminalOp.evaluateSequential(this, sourceSpliterator(terminalOp.getOpFlags()));

}

collect操作会通过ReduceTask来执行并发任务：

@Override

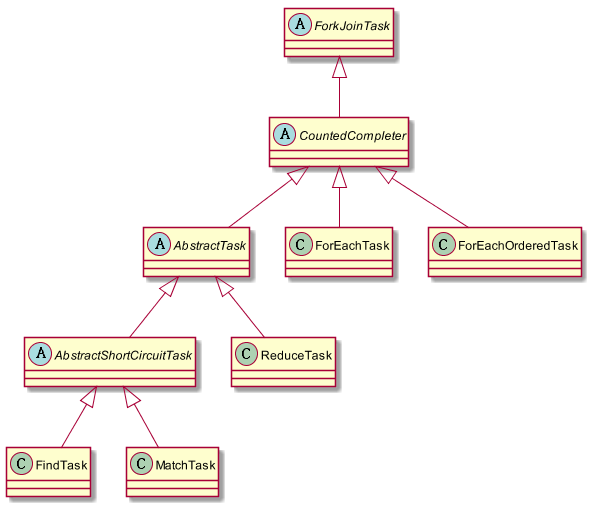
public <P\_IN> R evaluateParallel(PipelineHelper<T> helper,

Spliterator<P\_IN> spliterator) {

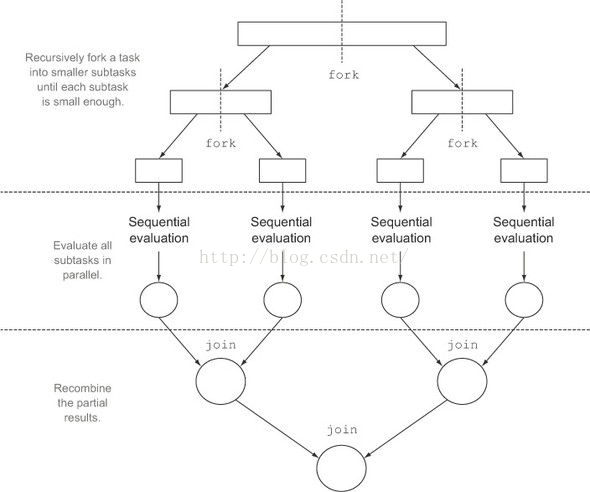
return new ReduceTask<>(this, helper, spliterator).invoke().get();

}

ReduceTask是ForkJoinTask的子类，其实Stream的并行处理都是基于Fork/Join框架的，相关类与接口的结构如下图所示：



fork/join框架是jdk1.7引入的，可以以递归方式将并行的任务拆分成更小的任务，然后将每个子任务的结果合并起来生成整体结果。它是ExecutorService接口的一个实现，它把子任务分配线程池（ForkJoinPool）中的工作线程。要把任务提交到这个线程池，必须创建RecursiveTask<R>的一个子类，如果任务不返回结果则是RecursiveAction的子类。



对于ReduceTask来说，任务分解的实现定义在其父类AbstractTask的compute()方法当中：

@Override

public void compute() {

Spliterator<P\_IN> rs = spliterator, ls; // right, left spliterators

long sizeEstimate = rs.estimateSize();

long sizeThreshold = getTargetSize(sizeEstimate);

boolean forkRight = false;

@SuppressWarnings("unchecked") K task = (K) this;

while (sizeEstimate > sizeThreshold && (ls = rs.trySplit()) != null) {

K leftChild, rightChild, taskToFork;

task.leftChild = leftChild = task.makeChild(ls);

task.rightChild = rightChild = task.makeChild(rs);

task.setPendingCount(1);

if (forkRight) {

forkRight = false;

rs = ls;

task = leftChild;

taskToFork = rightChild;

}

else {

forkRight = true;

task = rightChild;

taskToFork = leftChild;

}

taskToFork.fork();

sizeEstimate = rs.estimateSize();

}

task.setLocalResult(task.doLeaf());

task.tryComplete();

}

主要逻辑如下：

先调用当前splititerator 方法的estimateSize 方法，预估这个分片中的数据量，根据预估的数据量获取最小处理单元的阈值，即当数据量已经小于这个阈值的时候进行计算，否则进行fork 将任务划分成更小的数据块，进行求解。

值得注意的是，这里面有个很重要的参数，用来判断是否需要继续分割成更小的子任务，默认为*parallelism\*4，parallelism*是并发度的意思，默认值为***cpu 数 – 1***，可以通过java.util.concurrent.ForkJoinPool.common.parallelism设置，

如果当前分片大小仍然大于处理数据单元的阈值，且分片继续尝试切分成功，那么就继续切分，分别将左右分片的任务创建为新的Task，并且将当前的任务关联为两个新任务的父级任务（逻辑在makeChild 里面）

先后对左右子节点的任务进行fork，对另外的分区进行分解。同时设定pending 为1，这代表一个task 实际上只会有一个等待的子节点（被fork）。

当任务已经分解到足够小的时候退出循环，尝试进行结束。调用子类实现的doLeaf方法，完成最小计算单元的计算任务，并设置到当前任务的localResult中。

然后调用tryComplete 方法进行最终任务的扫尾工作，如果该任务pending 值不等于0，则原子的减1，如果已经等于0，说明任务都已经完成，则调用onCompletion 回调，如果该任务是叶子任务，则直接销毁中间数据结束；如果是中间节点会将左右子节点的结果进行合并

最后检查这个任务是否还有父级任务了，如果没有则将该任务置为正常结束，如果还有则尝试递归的去调用父级节点的onCompletion回调，逐级进行任务的合并。

public final void tryComplete() {

CountedCompleter<?> a = this, s = a;

for (int c;;) {

if ((c = a.pending) == 0) {

a.onCompletion(s);

if ((a = (s = a).completer) == null) {

s.quietlyComplete();

return;

}

}

else if (U.compareAndSwapInt(a, PENDING, c, c - 1))

return;

}

}

并行流的实现本质上就是在ForkJoin上进行了一层封装，将Stream 不断尝试分解成更小的split，然后使用fork/join 框架分而治之。