新特性

Lambda表达式

背景：根据员工的 工资/年龄 来选择输出员工信息 //创建一个员工类

```
public class Employee {
private String name;
private int age;
private double sarlary;

public Employee() {

        super();
}

public Employee(String name, int age, double sarlary) {
        super();
        this.name = name;
        this.age = age;
        this.sarlary = sarlary;
}
// 这里省略所有的get and set的封装 ~~~
}
//根据需求写业务代码
public class Lambda01 {
        //创建一个列表，往里边添加内容
  List<Employee> emplpoyees = Arrays.asList(
                        new Employee("张三",23,888.88),
                        new Employee("张四",33,988.88),
                        new Employee("张吴",43,9988.88),
                        new Employee("张六",53,8998.88)
                        );
        //根据年龄判断：传统过滤方式
        public List<Employee> filterEmpByAge(List<Employee> list){
                List<Employee> emps = new ArrayList<>();
                for (Employee employee : list) {
                        if(employee.getAge()>=35) {条件是过滤年龄大于35的
                                emps.add(employee);
                        }
                }
                return emps;
        }

        //测试一个员工的年龄大于35岁的方法
        @Test
        public void test1() {
                List<Employee> filterEmp = filterEmpByAge(emplpoyees);
                for (Employee employee : filterEmp) {
                        System.out.println(employee);
                }
        }
        /**
         * test1()测试输入结果：
       Employee [name=张吴, age=43, sarlary=9988.88]
       Employee [name=张六, age=53, sarlary=8998.88]
         */
```

优点：代码直观、简单书写

缺点：如果过滤条件改变，比如年龄小于30的或者年龄大于45的，则需要添加两个新的过滤方法

```
//优化一：策略设计模式
  //第一步：创建一个接口：公共    公共的策略接口
  public interface MyPredicate<T> {
     public boolean test(T t);
     }
  //第二步：创建一个根据年龄判断员工的实现类：                    一个过滤条件就新建一个接口的实现类，
过滤条件：根据年龄      public class FiltyerByAge implements MyPredicate<Employee>{  不同复用
        @Override
        public boolean test(Employee t) {
                // TODO Auto-generated method stub
                return t.getAge()>=35;
        }
        }

  //第三步：编写策略方法
        public List<Employee> filterEmpee(List<Employee> list, MyPredicate<Employee> mp){    公共的策略方法
                List<Employee> emps = new ArrayList<>();
                for (Employee employee : list) {
                        if(mp.test(employee)) emps.add(employee);
                }
                return emps;
        }

        //第四步：测试优化一>>>>>>年龄判断
                @Test
                public void test2() {                    测试方法传递给策略方法我们接口的实现类
                List<Employee> filterEmp = filterEmpee(emplpoyees,new FiltyerByAge());
                for (Employee employee : filterEmp) {
                        System.out.println("————————————————优化一根据年龄
————————————————————: "+employee);
                }
        }

  //如果根据工资判断需要重写一个工资判断的实现类，如果还有其他要求仍同样需要重创建一个实
现类（显然不够合理）！！   所以需要完善，下边为优化二方式---------->"匿名内部类实现"

  //第五步：根据工资判断创建工资类
  public class FilterBySalary implements MyPredicate<Employee>{
        @Override
        public boolean test(Employee t) {
                return t.getSarlary()<=1000;
                }
过滤条件：根据工资      }

  //第六步：测试优化一>>>>>>工资判断
        @Test
  public void test3() {
     List<Employee> filterEmp = filterEmpee(emplpoyees,new FilterBySalary());

     for (Employee employee : filterEmp) {
        System.out.println("————————————————优化一根据工资
————————————————————: "+employee);
```

优点：公用了策略方法和策略接口，利用接口和多态提高了代码复用率
缺点：一个条件就要新建一个接口实现类，实现类过多，不合理

匿名内部类只是策略模式的简化版本，策略模式是自己新建接口的实现类，匿名内部类是jdk动态代理帮你创建接口的实现类，本质上还是一样的

```
/优化二：匿名内部类
        @Test
        public void test4() {
                List<Employee> filterEmpAge = filterEmpee(emplpoyees,new MyPredicate<Employee>() {
                        @Override
                        public boolean test(Employee t) {
                                return t.getAge()<=40;//使用了匿名内部类之后根据年龄/工资，只需要修改这里即可（不用再创建太多相关的实现类）
                        }
                });
                for (Employee employee : filterEmpAge) {
                        System.out.println("————————————————————优化二》》匿名内部类————————————————————："+employee);
                }
        }

 // 感觉不错吧，接下来看看lambda表达式写法----------优化方式三：lambda表达式
  优点：共用了接口和策略方法，匿名内部类进一步提高了代码复用率
  缺点：一个条件就要jdk基于匿名内部类帮你代理新建一个接口实现类，导致运行时动态加载的类过多
      //优化三：lambda表达式

        @Test
        public void test5() {//整个逻辑写下来，代码非常简洁（这就是lambda魅力所在）
                List<Employee> filterEmpee = filterEmpee(emplpoyees,(e) -> e.getSarlary() >=5000);
                filterEmpee.forEach(System.out::println);
        }

  //下面还有最终优化版本：---------------优化方式四：Stream API

                @Test
                public void test6() {//这里什么方法都没有调用，单单只用了stream就对员工进行了判断！
                        emplpoyees.stream().filter((e)->e.getSarlary()
        >=5000).forEach(System.out::println);
                }
}
```

本质上还是基于策略模式的匿名内部类，lambad删除了匿名内部类中不必要语句

基本语法

- 操作符：->
- 左侧：参数列表
- 右侧：执行代码块 / Lambda 体

写死小括号，拷贝右箭头，落地大括号
上联：左右遇一括号省
下联：左侧推断类型省
横批：能省就省

语法格式：
无参数，无返回值：() -> sout

```
public class Test02 {
   int num = 10; //jdk 1.7以前 必须final修饰，匿名内部类访问成员变量必须final，1.8编译器会自动加final
   @Test
   public void test01(){
      //匿名内部类
      new Runnable() {
         @Override
         public void run() {
//在局部类中引用同级局部变量
//只读
            System.out.println("Hello World" + num);
         }
      };
   }
   @Test
   public void test02()
{//lambda语法糖
      Runnable runnable = () -> {
         System.out.println("Hello Lambda");
      };
   }
}
```

第一种类型：lambda表达式，方法无参无返回值

```
@Test
  public void test03(){
     Consumer<String> consumer = (a) -> System.out.println(a);
     consumer.accept("我觉得还行！ ");
  }
```

第二种类型：方法一个参数，无返回值

```
  @Test
  public void test04(){
     Comparator<Integer> comparator = (a, b) -> {
        System.out.println("比较接口");
        return Integer.compare(a, b);
     };
  }
```

第三种类型：方法多个参数，有返回值

有返回值，使用大括号包含多条语句{}

```
@Test
  public void test04(){

    Comparator<Integer> comparator = (a, b) ->
                                                    Integer.compare(a, b);
  }
```

第四种类型：方法多个参数，有返回值

函数式接口基本概念

函数式接口：
接口中只有一个抽象方法的接口 @FunctionalIterface，注解会自动对接口进行检查是否是函数式接口
测试：
定义一个函数式接口

```
@FunctionalInterface
  public interface MyFun {   函数式接口定义
    Integer count(Integer a, Integer b);
  }

@Test
  public void test05(){
    MyFun myFun1 = (a, b) -> a + b;
    MyFun myFun2 = (a, b) -> a - b;
    MyFun myFun3 = (a, b) -> a * b;
    MyFun myFun4 = (a, b) -> a / b;
  }

  public Integer operation(Integer a, Integer b, MyFun myFun){
    return myFun.count(a, b);
  }
  @Test
  public void test06(){
    Integer result = operation(1, 2, (x, y) -> x + y);
    System.out.println(result);
  }
```

策略模式中的公共策略方法。（接口+多态）

lambda本质上就是匿名内部类，函数式接口MyFun的匿名内部类实现

Collection.sort定制排序，比较两个Employee（先按照年龄比，年龄相同按照姓名比）

定义实体类

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Employee {
  private Integer id;
  private String name;
  private Integer age;
  private Double salary;
}
```

定义 List 传入数据

```
List<Employee> emps = Arrays.asList(
      new Employee(101, "Z3", 19, 9999.99),
      new Employee(102, "L4", 20, 7777.77),
      new Employee(103, "W5", 35, 6666.66),
      new Employee(104, "Tom", 44, 1111.11),
      new Employee(105, "Jerry", 60, 4444.44)
  );
```

```
@Test
public void test01(){
    Collections.sort(emps, (e1, e2) -> { void sort(List<T> list, Comparator<? super T> c)
        if (e1.getAge() == e2.getAge()){
            return e1.getName().compareTo(e2.getName());
        } else {
            return Integer.compare(e1.getAge(), e2.getAge());
        }
    });
    for (Employee emp : emps) {
        System.out.println(emp);
    }
}
```

Comparator是函数式接口

```
@FunctionalInterface
public interface Comparator<T> {
多个抽象方法
}
```

内置四大函数接口（程序员就不需要自己自定义函数式接口了）

函数式接口	参数类型	返回类型	用途
Consumer 消费型接口	T	void	对类型为T的对象应用操作：void accept(T t)
Supplier 提供型接口	无	T	返回类型为T的对象：T get()
Function<T, R> 函数型接口	T	R	对类型为T的对象应用操作，并返回结果为R类型的对象：R apply(T t)
Predicate 断言型接口	T	boolean	确定类型为T的对象是否满足某约束，并返回boolean值：boolean test(T t)

消费型接口：有参数、无返回值

```
@Test
public void test01(){
    //Consumer
    Consumer<Integer> consumer = (x) -> System.out.println("消费型接口" + x);
    //test
    consumer.accept(100);
}
```

提供型接口：无参数，有返回值

```
@Test
public void test02(){
    List<Integer> list = new ArrayList<>();
    List<Integer> integers = Arrays.asList(1,2,3);
    list.addAll(integers);
    //Supplier<T>
    Supplier<Integer> supplier = () -> (int)(Math.random() * 10);
    list.add(supplier.get());
    System.out.println(supplier);
    for (Integer integer : list) {
        System.out.println(integer);
    }
}
```

函数型接口：有参数、有返回值

```
@Test
public void test03(){
    //Function<T, R>
    String oldStr = "abc123456xyz";
    Function<String, String> function = (s) -> s.substring(1, s.length()-1);
    //test
    System.out.println(function.apply(oldStr));
}
```

断言型接口：有参数，布尔类型返回值

```
@Test
public void test04(){
    //Predicate<T>
    Integer age = 35;
    Predicate<Integer> predicate = (i) -> i >= 35;
    if (predicate.test(age)){
        System.out.println("你该退休了");
    } else {
        System.out.println("我觉得还OK啦");
    }
}
```

Lambda 表达式 参数的数据类型可以省略不写 Jvm可以自动进行 “类型推断”

```
public class Demo {
public static void testLamba(){
    int a=10;
    int b=20;
    MyFun sum=(e1,e2)->{
      return (e1+e2);
    };
    test(a,b,sum);
  }

  public static void testInnerClass(){
  int a=10;
   int b=20;
  test(a, b, new MyFun() {
      @Override
      public Integer count(Integer a, Integer b) {

        return  a*b;
      }
    });
  }
  public static void test(int a,int b,MyFun myFun){
    System.out.println(myFun.count(a,b));
  }
@FunctionalInterface
interface MyFun {
  Integer count(Integer a, Integer b);
}
```

Lambda表达式的字节码

```
0 bipush 10
2 istore_0
3 bipush 20
5 istore_1
6 invokedynamic #2 <count, BootstrapMethods #0>
11 astore_2
12 iload_0
13 iload_1
14 aload_2
15 invokestatic #3 <se基础/demo/新特性/Demo.test>
18 return
```

10放入操作数栈

操作数栈中10放入方法局部变量表的0位置

20放入操作数栈

操作数栈中10放入方法局部变量表的1位置

匿名内部类的字节码

匿名内部类可以看之前的文章

```
0 bipush 10
2 istore_0
3 bipush 20
5 istore_1
6 iload_0
7 iload_1
8 new #4 <se基础/demo/新特性/Demo$1>
11 dup
12 invokespecial #5 <se基础/demo/新特性/Demo$1.<init>>
15 invokestatic #3 <se基础/demo/新特性/Demo.test>
18 return
```

匿名内部类会生成$1这种字节码class文件。

Lambda 表达式

想要更好的理解Android对Java 8的支持过程，Lambda表达式这一代表性的”语法糖“是一个非常不错的切入点。所以，我们首先需要搞清楚Lambda表达式到底是什么？其底层的实现原理又是什么？

Lambda表达式是Java支持函数式编程的基础，也可以称之为闭包。简单来说，就是在Java语法层面允许将函数当作方法的参数，函数可以当做对象。任一Lambda表达式都有且只有一个函数式接口与之对应，从这个角度来看，也可以说是该函数式接口的实例化。

Lambda表达式

通用格式：

```
语法格式：
(parameters) -> expression  或者  (parameters) -> { statements; }

格式理解：
( 对应函数式接口的参数列表 ) -> { 对应函数接口的实现方法 }
```

简单范例：

```
package com.j8sample;

public class J8Sample {

    public static void main(String arg[]) {

        Runnable runnable = () -> System.out.println("xixi"); // Lambda表达式1
        new Thread(runnable).start();

        new Thread(() -> {
            System.out.println("haha"); // Lambda表达式2
        }).start();
    }
}
```

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

说明：

- Lambda表达式中 () 对应的是函数式接口-run方法的参数列表。
- Lambda表达式中 System.out.println（“xixi”）/ System.out.println（“haha”），在运行时会是具体的run方法实现。

Lambda表达式原理

针对实例中的代码，我们来看下编译之后的字节码：

```
javac J8Sample.java  ->  J8Sample.class

javap -c -p J8Sample.class
```

```
1  Compiled from "J8Sample.java"
2  public class com.j8sample.J8Sample {
3    public com.j8sample.J8Sample();
4      Code:
5         0: aload_0
6         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
7         4: return
8
9    public static void main(java.lang.String[]);
10     Code:
11        0: invokedynamic #2,  0              // InvokeDynamic #0:run:()Ljava/lang/Runnable;
12        5: astore_1
13        6: new           #3                  // class java/lang/Thread
14        9: dup
15       10: aload_1
16       11: invokespecial #4                  // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
17       14: invokevirtual #5                  // Method java/lang/Thread.start:()V
18       17: new           #3                  // class java/lang/Thread
19       20: dup
20       21: invokedynamic #6,  0              // InvokeDynamic #1:run:()Ljava/lang/Runnable;
21       26: invokespecial #4                  // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
22       29: invokevirtual #5                  // Method java/lang/Thread.start:()V
23       32: return
24
25   private static void lambda$main$1();
26     Code:
27        0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
28        3: ldc           #8                  // String haha
29        5: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
30        8: return
31
32   private static void lambda$main$0();
33     Code:
34        0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
35        3: ldc           #10                 // String xixi
36        5: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
37        8: return
38 }
```

从字节码中我们可以看到：

- 实例中 Lambda表达式1变成了字节码代码块中 Line 11的 0: invokedynamic #2, 0 // InvokeDynamic #0:run:()Ljava/lang/Runnable。
- 实例中 Lambda表达式2变成了字节码代码块中 Line 20的 21: invokedynamic #6, 0 // InvokeDynamic #1:run:()Ljava/lang/Runnable。

可见，Lambda表达式在虚拟机层面上，是通过一种名为invokedynamic字节码指令来实现的。那么invokedynamic又是何方神圣呢?

invokedynamic 指令解读

invokedynamic指令是Java 7中新增的字节码调用指令，作为Java支持动态类型语言的改进之一，跟invokevirtual、invokestatic、invokeinterface、invokespecial四大指令一起构成了虚拟机层面各种Java方法的分配调用指令集。区别在于：

- 后四种指令，在编译期间生成的class文件中，通过常量池(Constant Pool)的MethodRef常量已经固定了目标方法的符号信息（方法所属者及其类型，方法名字、参数顺序和类型、返回值）。虚拟机使用符号信息能直接解释出具体的方法，直接调用。

- 而invokedynamic指令在编译期间生成的class文件中，对应常量池(Constant Pool)的Invokedynamic_Info常量存储的符号信息中并没有方法所属者及其类型，替代的是BootstapMethod信息。在运行时，通过引导方法BootstrapMethod机制动态确定方法的所属者和类型。这一特点也非常契合动态类型语言只有在运行期间才能确定类型的特征。编译期无法判断出需要执行的具体方法，因为可能方法是虚拟机运行过程动态生成的class对象，然后调用该对象的方法

那么，invokedynamic如何通过引导方法找到所属者及其类型？我们依然结合前面的J8Sample实例： 官方字节码地址：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5.invokedynamic

```
javap -v J8Sample.class
```

```
Constant pool:  (class 常量池)
   #1 = Methodref          #12.#30        // java/lang/Object."<init>":()V
   #2 = InvokeDynamic      #0:#35         // #0:run:()Ljava/lang/Runnable;
   #3 = Class              #36            // java/lang/Thread
   #4 = Methodref          #3.#37         // java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
   #5 = Methodref          #3.#38         // java/lang/Thread.start:()V
   #6 = InvokeDynamic      #1:#35         // #1:run:()Ljava/lang/Runnable;
   #7 = Fieldref           #40.#41        // java/lang/System.out:Ljava/io/PrintStream;
   #8 = String             #42            // haha
   #9 = Methodref          #43.#44        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #10 = String             #45            // xixi
  #11 = Class              #46            // com/j8sample/J8Sample
  #12 = Class              #47            // java/lang/Object
  #13 = Utf8               <init>
  ..........

SourceFile: "J8Sample.java" (BootstrapMethod 引导方法)
InnerClasses:
     public static final #68= #67 of #71;
    //Lookup=class java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles
BootstrapMethods:
  0: #32 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$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:
        #33 ()V
        #34 invokestatic com/j8sample/J8Sample.lambda$main$0:()V
        #33 ()V
        ...........
```

结合J8Sample.class字节码，并对invokedynamic指令调用过程进行跟踪分析。总结如下：

依据上图invokedynamic调用步骤，我们一步一步做一个分析讲解。

步骤1 选取J8Sample.java源码中Lambda表达式1：

`Runnable runnable = () -> System.out.println("xixi"); // lambda表达式1`

步骤2 通过 `javac J8Sample.java` 编译得到 `J8Sample.class` 之后，

`Lambda表达式1变成：0: invokedynamic #2, 0 // InvokeDynamic #0:run:()Ljava/lang/Runnable;`

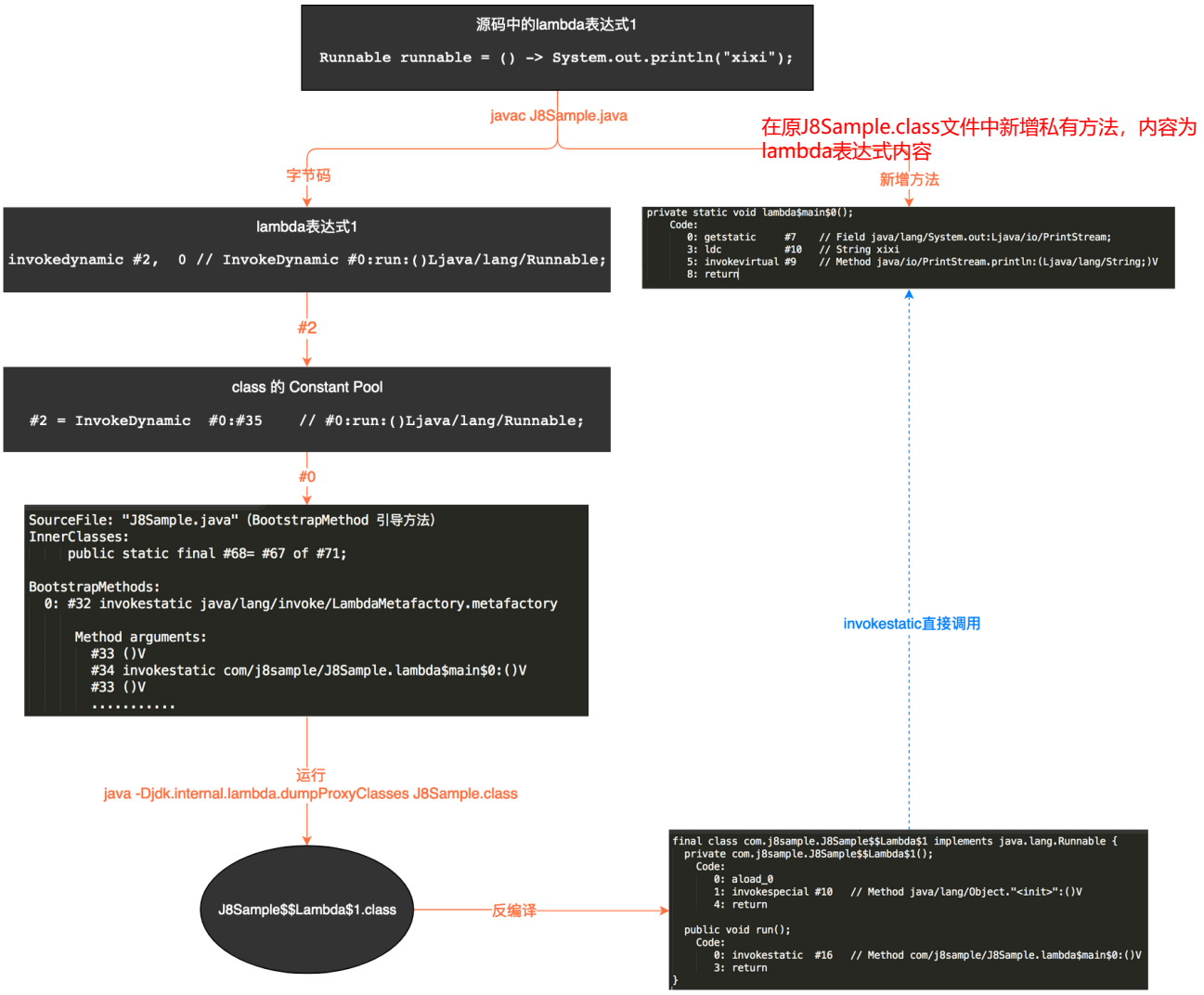
对应在 `J8Sample.class` 中发现了新增的私有静态方法：

步骤3 针对表达式1的字节码分析 #2 对应的是class文件中的常量池：

`#2 = InvokeDynamic #0:#35 // #0:run:()Ljava/lang/Runnable;`

注意，这里InvokeDynamic不是指令，代表的是 `Constant_InvokeDynamic_Info` 结构。

步骤4 结构后面紧跟的 #0 标识的是class文件中的BootstrapMethod区域中引导方法的索引：




```
private static void lambda$main$0();
  Code:
     0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: ldc           #10                 // String xixi
     5: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
     8: return
```

```
BootstrapMethods:
  0: #32 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;
                                                                     Ljava/lang/String;
                                                                     Ljava/lang/invoke/MethodType;
                                                                     Ljava/lang/invoke/MethodType;
                                                                     Ljava/lang/invoke/MethodHandle;
                                                                     Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
     Method arguments:
       #33 ()V
       #34 invokestatic com/j8sample/J8Sample.lambda$main$0:()V
       #33 ()V
```

步骤5 引导方法中的 java/lang/invoke/LambdaMetafactory.metafactory才是invokedynamic指令的关键：

该方法会在运行时，在内存中动态生成一个实现Lambda表达式对应函数式接口的实例类型，并在接口的实现方法中调用步骤2中新增的静态私有方法。

步骤6 使用 `java -Djdk.internal.lambda.dumpProxyClasses J8Sample.class` 运行一下，可以内存中动态生成的类型输出到本地：

步骤7 通过 `javap -p -c J8Sample\$\$Lambda\$1.class` 反编译一下，可以看到生成类的实现：

```
public static CallSite metafactory(MethodHandles.Lookup caller,  caller: "se基础.demo.新特性.J8sample"
                                   String invokedName,  invokedName: "run"
                                   MethodType invokedType,  invokedType: "()Runnable"
                                   MethodType samMethodType,  samMethodType: "()void"
                                   MethodHandle implMethod,  implMethod: "MethodHandle()void"
                                   MethodType instantiatedMethodType)  instantiatedMethodType: "()void"
        throws LambdaConversionException {
    AbstractValidatingLambdaMetafactory mf;  mf (slot_6): InnerClassLambdaMetafactory@709
    mf = new InnerClassLambdaMetafactory(caller, invokedType,  caller: "se基础.demo.新特性.J8sample"  invokedType: "()Runnable"
                                         invokedName, samMethodType,  invokedName: "run"  samMethodType: "()void"
                                         implMethod, instantiatedMethodType,  implMethod: "MethodHandle()void"  instantiatedMethodType: "()void"
                                         isSerializable: false, EMPTY_CLASS_ARRAY, EMPTY_MT_ARRAY);
    mf.validateMetafactoryArgs();
    return mf.buildCallSite();  mf (slot_6): InnerClassLambdaMetafactory@709
```

```
public InnerClassLambdaMetafactory(MethodHandles.Lookup caller,  caller: "se基础.demo.新特性.J8sample"
                                   MethodType invokedType,  invokedType: "()Runnable"
                                   String samMethodName,  samMethodName: "run"
                                   MethodType samMethodType,  samMethodType: "()void"
                                   MethodHandle implMethod,  implMethod: "MethodHandle()void"
                                   MethodType instantiatedMethodType,  instantiatedMethodType: "()void"
                                   boolean isSerializable,  isSerializable: false
                                   Class<?>[] markerInterfaces,  markerInterfaces: Class[0]@710
                                   MethodType[] additionalBridges)  additionalBridges: MethodType[0]@711
        throws LambdaConversionException {
    super(caller, invokedType, samMethodName, samMethodType,  caller: "se基础.demo.新特性.J8sample"  samMethodName: "run"  samMethodType: "()void"
          implMethod, instantiatedMethodType,  implMethod: "MethodHandle()void"  instantiatedMethodType: "()void"
          isSerializable, markerInterfaces, additionalBridges);  isSerializable: false  markerInterfaces: Class[0]@710  additionalBridges: MethodType[0]@71
    implMethodClassName = implDefiningClass.getName().replace( oldChar: '.', newChar: '/');  implMethodClassName: "se基础/demo/新特性/J8sample"
    implMethodName = implInfo.getName();  implMethodName: "lambda$main$0"
    implMethodDesc = implMethodType.toMethodDescriptorString();  implMethodDesc: "()V"
    implMethodReturnClass = (implKind == MethodHandleInfo.REF_newInvokeSpecial)  implMethodReturnClass: "void"
            ? implDefiningClass
            : implMethodType.returnType();
    constructorType = invokedType.changeReturnType(Void.TYPE);  constructorType: "()void"
    lambdaClassName = targetClass.getName().replace( oldChar: '.', newChar: '/') + "$$Lambda$" + counter.incrementAndGet();  lambdaClassName: "se基础/demo/新特性
    cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);  cw: ClassWriter@903
    int parameterCount = invokedType.parameterCount();  parameterCount (slot_10): 0
    if (parameterCount > 0) {
        argNames = new String[parameterCount];
        argDescs = new String[parameterCount];
        for (int i = 0; i < parameterCount; i++) {  parameterCount (slot_10): 0
            argNames[i] = "arg$" + (i + 1);
            argDescs[i] = BytecodeDescriptor.unparse(invokedType.parameterType(i));  invokedType: "()Runnable"
```

import jdk.internal.org.objectweb.asm.ClassWriter;
cw = new ClassWriter(ClassWriter.COMPUTE_MAXS); 这里是用ASM字节码框架，类加载前生成一个class对象，但是并不以.class文件形式保存在磁盘中，对比匿名内部类会磁盘中生成$1.class这种形式的字节码文件

```
final class com.j8sample.J8Sample$$Lambda$1 implements java.lang.Runnable {
  private com.j8sample.J8Sample$$Lambda$1();
    Code:
       0: aload_0
       1: invokespecial #10                 // Method java/lang/Object."<init>":()V
       4: return

  public void run();
    Code:
       0: invokestatic  #16                 // Method com/j8sample/J8Sample.lambda$main$0:()V
       3: return
}
```

在run方法中使用了invokestatic指令，直接调用了 `J8Sample.lambda$main$0` 这个在编译期间生成的静态私有方法。

至此，上面7个步骤就是Lambda表达式在Java的底层的实现原理。Android 针对这些实现会怎么处理呢?

Lambda底层原理总结:
1、lambda表达式会在原来的class文件中生成私有静态方法，方法内容就是lambda表达式内容。
2、执行lambda表达式会执行invokedynamic字节码指令，会调用引导方法BootstrptMethod执行LambdaMetaFactore.metafactore方法（Lambda元工厂），底层利用ASM字节码框架，在类加载前动态生成一个实现了lambda接口的class对象，基于匿名内部类的方式，但是并不以.class文件形式存放在磁盘中。
3、该class对象会执行invokestatic方法调用1过程中生成的静态方法。

invokedynamic原理分析参考

```
//lambda使用匿名内部类作为接口的实现类。
public class Test {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        int a=10;
        int b=20;
        MyFun fun=(a1,b2)->{
            return a+a1+b2;
        };
        System.out.println(fun.count(a,b));

    }
}
interface MyFun {
    int count(int a, int b);
}
```

使用java -Djdk.internal.lambda.dumpProxyClasses Test得到的Test.java运行过程中生成的匿名内部类，因此具备内部类的特性，当lambda中使用到外部的对象或者数据时候，必须是final类型的。

```
// $FF: synthetic class，匿名内部类MyFun子类对象
final class Test$$Lambda$1 implements MyFun {
   private final int arg$1;
   private Test$$Lambda$1(int var1) {
      this.arg$1 = var1;
   }
   private static MyFun get$Lambda(int var0) {
      return new Test$$Lambda$1(var0);
   }
   @Hidden
   public int count(int var1, int var2) {
      return Test.lambda$main$0(this.arg$1, var1, var2);//调用Test.java中为lambda生成的私有静态方法
   }
}
```

方法引用

定义：
方法引用是用来直接访问类或者实例的已经存在的方法或者构造方法。方法引用提供了一种引用而不执行方法的方式，它需要由兼容的函数式接口构成的目标类型上下文。计算时，方法引用会创建函数式接口的一个实例。
当Lambda表达式中只是执行一个方法调用时，不用Lambda表达式，直接通过方法引用的形式可读性更高一些。方法引用是一种更简洁易懂的Lambda表达式。
注意方法引用是一个Lambda表达式，其中方法引用的操作符是双冒号"::"。
==========

简单地说，就是一个Lambda表达式。在Java 8中，我们会使用Lambda表达式创建匿名方法，但是有时候，我们的Lambda表达式可能仅仅调用一个已存在的方法，而不做任何其它事，对于这种情况，通过一个方法名字来引用这个已存在的方***更加清晰，Java 8的方法引用允许我们这样做。方法引用是一个更加紧凑，易读的Lambda表达式，注意方法引用是一个Lambda表达式，其中方法引用的操作符是双冒号"::"

对象 :: 实例方法
类 :: 静态方法
类 :: 实例方法

```
@Test
public void test01(){
   PrintStream ps = System.out;
   Consumer<String> con1 = (s) -> ps.println(s);
   con1.accept("aaa");

   Consumer<String> con2 = ps::println;
   con2.accept("bbb");
}
```

注意：Lambda 表达实体中调用方法的参数列表、返回类型必须和函数式接口中抽象方法保持一致

```
con2方法引用对应的字节码
 0 getstatic #3 <java/lang/System.out>
 3 astore_0
 4 aload_0
 5 dup
 6 invokevirtual #4 <java/lang/Object.getClass>
 9 pop
10 invokedynamic #5 <accept, BootstrapMethods #0>
15 astore_1
16 aload_1
17 ldc #6 <bbb>
19 invokeinterface #7 <java/util/function/Consumer.accept> count 2
24 return
```

源代码：使用了对象：：实例方法

```
public class MethodReferenct2 {
   public void sayHello(Consumer<String> f){
      f.accept("aa");
   }
   public static void main(String argv[]){
      MethodReferenct2 main=new MethodReferenct2();
      main.sayHello(System.out::print);
   }
}
```

```
 0 new #4 <se基础/demo/新特性/MethodReferenct2>
 3 dup
 4 invokespecial #5 <se基础/demo/新特性/MethodReferenct2.<init>>
 7 astore_1
 8 aload_1
 9 getstatic #6 <java/lang/System.out>
12 dup
13 invokevirtual #7 <java/lang/Object.getClass>
16 pop
17 invokedynamic #8 <accept, BootstrapMethods #0>
22 invokevirtual #9 <se基础/demo/新特性/MethodReferenct2.sayHello>
25 return
```

MethodRfefenct2.class运行时产生的class文件，没有保存在磁盘中
java -Djdk.internal.lambda.dumpProxyClasses MethodReferenct2

```
final class MethodReferenct2$$Lambda$1 implements Consumer {
    private final PrintStream arg$1;
```

接口的子类对象内部arg$1引用了（方法引用的对象PrinStream）

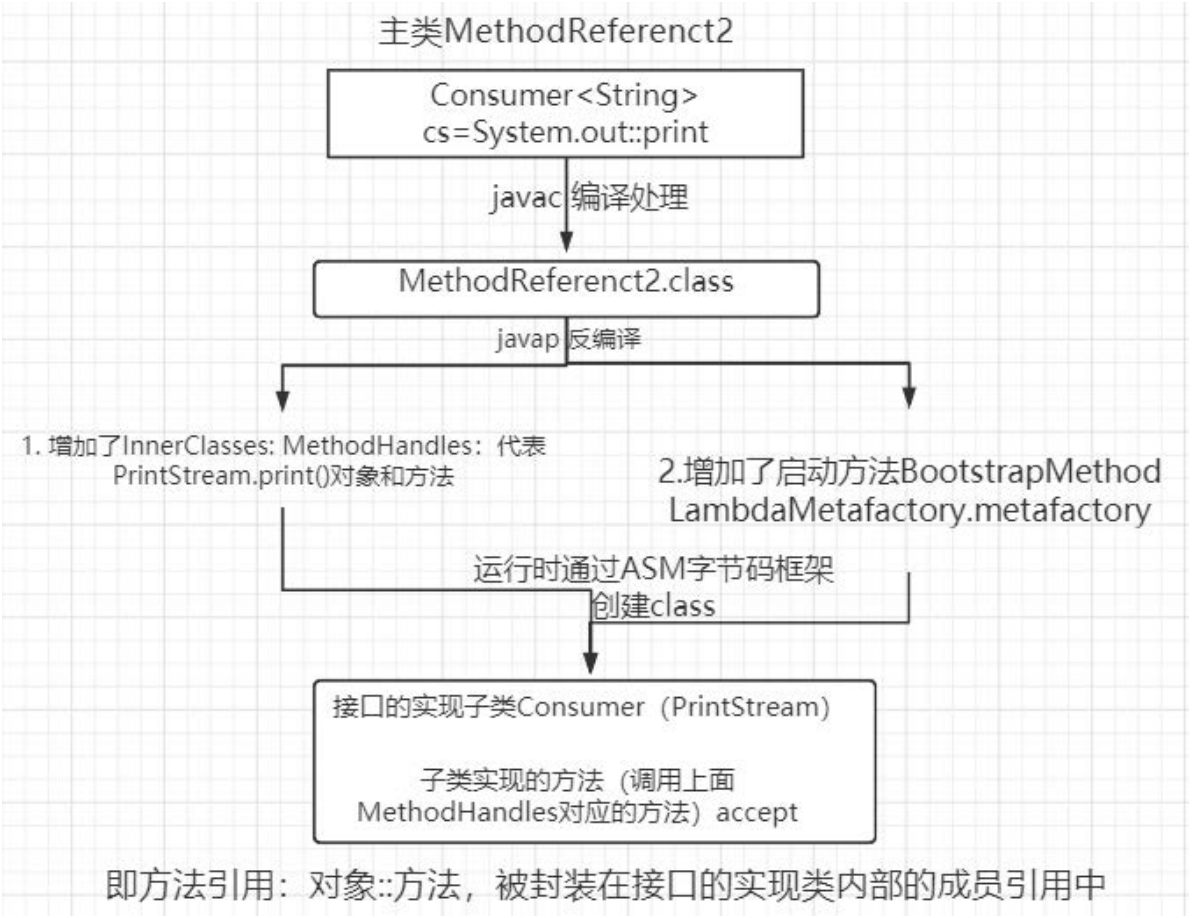
```
  private MethodReferenct2$$Lambda$1(PrintStream var1) {
     this.arg$1 = var1;
  }

  private static Consumer get$Lambda(PrintStream var0) {
     return new MethodReferenct2$$Lambda$1(var0);    }

  @Hidden
  public void accept(Object var1) {
     this.arg$1.print((String)var1);
  }

}
```

接口的实现类的构造器中初始化方法引用的对象

接口的实现方法中,调用方法引用的对象和方法

这是一个实现Consumer类的子类对象，由BootstrapMethod引导方法运行时创建的class对象。在MethodReferenct2$$Lambda$1.accept()方法中直接调用了PrintStream对象的print()方法，这与lambda表达式生成的的内部类不同。

原理总结：方法引用（对象::实例方法）会通过主类的启动方法BootstraptMethod的LambdaMetafactory.metafactory元工厂，创建实现接口（Consumer）的子类，并且把实例对象(PrintStream)传递给子类构造器，并初始化子类。子类调用accept方法时候，其实内部调用的是对象(PrintStream)的实例方法（print)。不像原本的Lambda语法，生成private static私有方法，方法内容包含了Lambda表达式的内容。



类::静态方法

```
@Test
public void test02(){
    Comparator<Integer> com1 = (x, y) -> Integer.compare(x, y);
    System.out.println(com1.compare(1, 2));
    Comparator<Integer> com2 = Integer::compare;
    System.out.println(com2.compare(2, 1));
}
```

类::实例方法

```
@Test
public void test03(){
    BiPredicate<String, String> bp1 = (x, y) -> x.equals(y);
    System.out.println(bp1.test("a","b"));
    BiPredicate<String, String> bp2 = String::equals;
    System.out.println(bp2.test("c","c"));
}
```

›

Java 8中处理集合的优雅姿势——Stream (https://www.hollischuang.com/archives/3333)

Stream产生背景：像操作数据库一样操作集合

在Java中，集合和数组是我们经常会用到的数据结构，需要经常对他们做增、删、改、查、聚合、统计、过滤等操作。相比之下，关系型数据库中也同样有这些操作，但是在Java 8之前，集合和数组的处理并不是很便捷。

不过，这一问题在Java 8中得到了改善，Java 8 API添加了一个新的抽象称为流Stream，可以让你以一种声明的方式处理数据。本文就来介绍下如何使用Stream。特别说明一下，关于Stream的性能及原理不是本文的重点，如果大家感兴趣后面会出文章单独介绍。

Stream介绍

Stream 使用一种类似用 SQL 语句从数据库查询数据的直观方式来提供一种对 Java 集合运算和表达的高阶抽象。

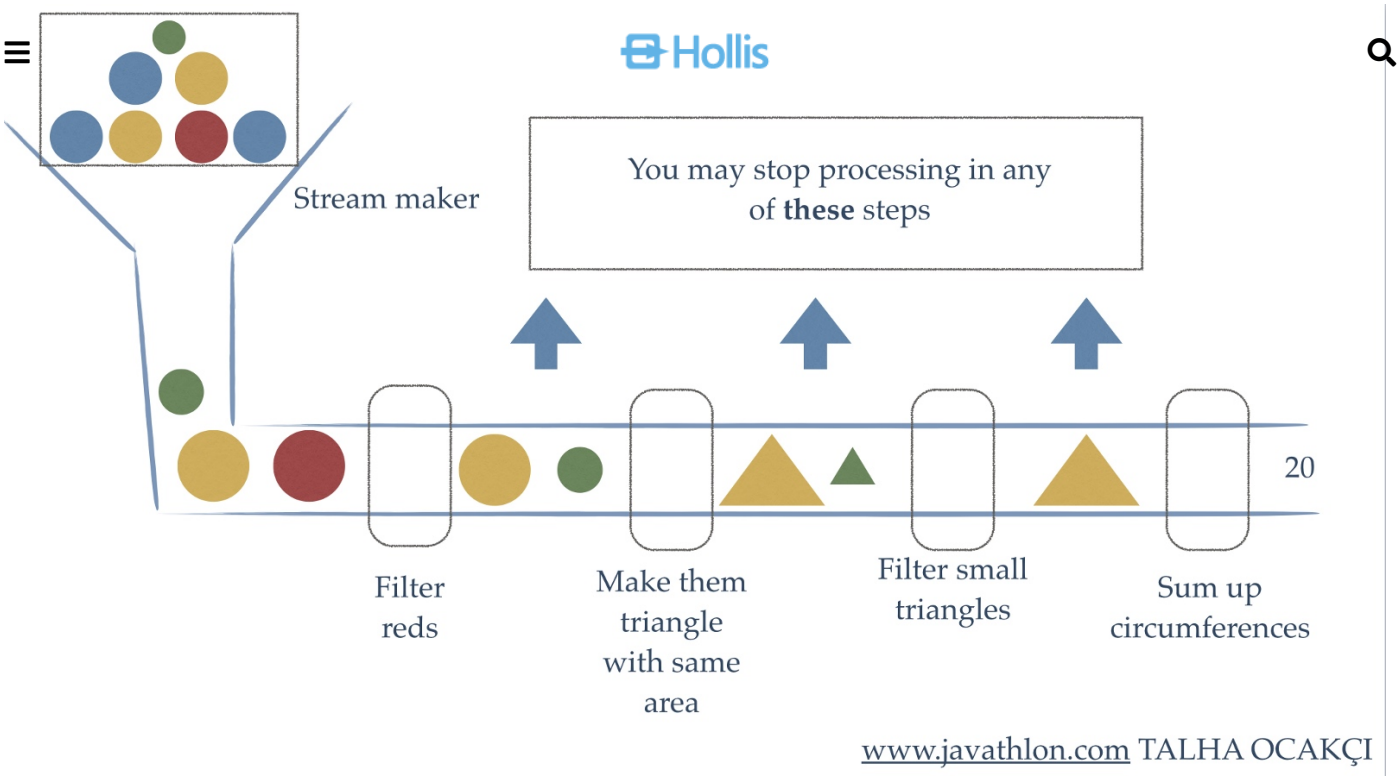
Stream API可以极大提高Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

这种风格将要处理的元素集合看作一种流，流在管道中传输，并且可以在管道的节点上进行处理，比如筛选，排序，聚合等。

Stream有以下特性及优点：

- 无存储。Stream不是一种数据结构，它只是某种数据源的一个视图，数据源可以是一个数组，Java容器或I/O channel等。
- 为函数式编程而生。对Stream的任何修改都不会修改背后的数据源，比如对Stream执行过滤操作并不会删除被过滤的元素，而是会产生一个不包含被过滤元素的新Stream。
- 惰式执行。Stream上的操作并不会立即执行，只有等到用户真正需要结果的时候才会执行。
- 可消费性。Stream只能被“消费”一次，一旦遍历过就会失效，就像容器的迭代器那样，想要再次遍历必须重新生成。

我们举一个例子，来看一下到底Stream可以做什么事情：



上面的例子中，获取一些带颜色塑料球作为数据源，首先过滤掉红色的、把它们融化成随机的三角形。再过滤器并删除小的三角形。最后计算出剩余图形的周长。

如上图，对于流的处理，主要有三种关键性操作：分别是流的创建、中间操作（intermediate operation）以及最终操作(terminal operation)。

Stream的创建

在Java 8中，可以有多种方法来创建流。

1、通过已有的集合来创建流

在Java 8中，除了增加了很多Stream相关的类以外，还对集合类自身做了增强，在其中增加了stream方法，可以将一个集合类转换成流。

```
List<String> strings = Arrays.asList("Hollis", "HollisChuang", "hollis", "Hello", "HelloWorld", "Hollis");
Stream<String> stream = strings.stream();
```

以上，通过一个已有的List创建一个流。除此以外，还有一个parallelStream方法，可以为集合创建一个并行流。

这种通过集合创建出一个Stream的方式也是比较常用的一种方式。

2、通过Stream创建流

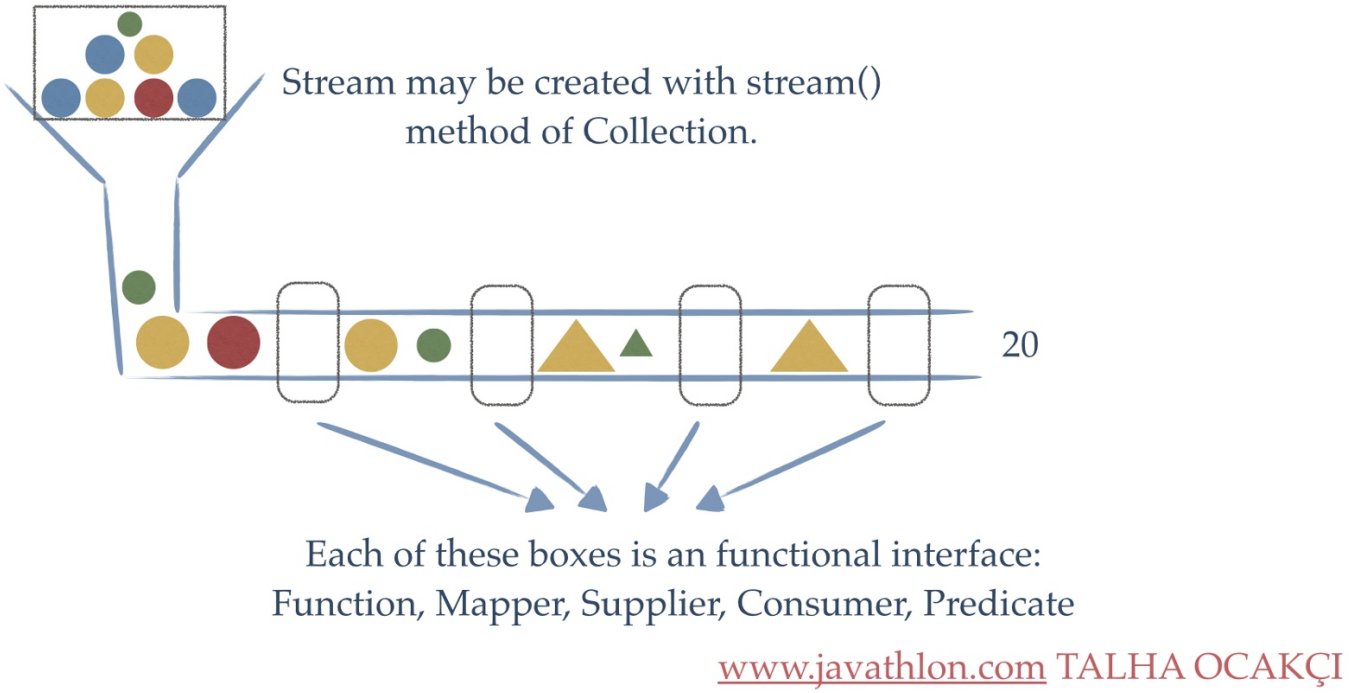
可以使用Stream类提供的方法，直接返回一个由指定元素组成的流。

```
Stream<String> stream = Stream.of("Hollis", "HollisChuang", "hollis", "Hello", "HelloWorld",
"Hollis");
```

如以上代码，直接通过of方法，创建并返回一个Stream。

Stream中间操作

Stream有很多中间操作，多个中间操作可以连接起来形成一个流水线，每一个中间操作就像流水线上的一个工人，每人工人都可以对流进行加工，加工后得到的结果还是一个流。



以下是常用的中间操作列表:

Stream Operation	Goal	Input
filter	Filter items according to a given predicate	Predicate
map	Processes items and transforms	Function
limit	Limit the results	int
sorted	Sort items inside stream	Comparator
distinct	Remove duplicate items according to equals method of the given type	

filter

filter 方法用于通过设置的条件过滤出元素。以下代码片段使用 filter 方法过滤掉空字符串：

```
List<String> strings = Arrays.asList("Hollis", "", "HollisChuang", "H", "hollis");
strings.stream().filter(string -> !string.isEmpty()).forEach(System.out::println);
//Hollis, , HollisChuang, H, hollis
```

map

map 方法用于映射每个元素到对应的结果，以下代码片段使用 map 输出了元素对应的平方数：

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
numbers.stream().map( i -> i*i).forEach(System.out::println);
//9,4,4,9,49,9,25
```

limit/skip

limit 返回 Stream 的前面 n 个元素；skip 则是扔掉前 n 个元素。以下代码片段使用 limit 方法保理4个元素：

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
numbers.stream().limit(4).forEach(System.out::println);
//3,2,2,3
```

sorted

sorted 方法用于对流进行排序。以下代码片段使用 sorted 方法进行排序：

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
numbers.stream().sorted().forEach(System.out::println);
//2,2,3,3,3,5,7
```

distinct

distinct主要用来去重，以下代码片段使用 distinct 对元素进行去重：

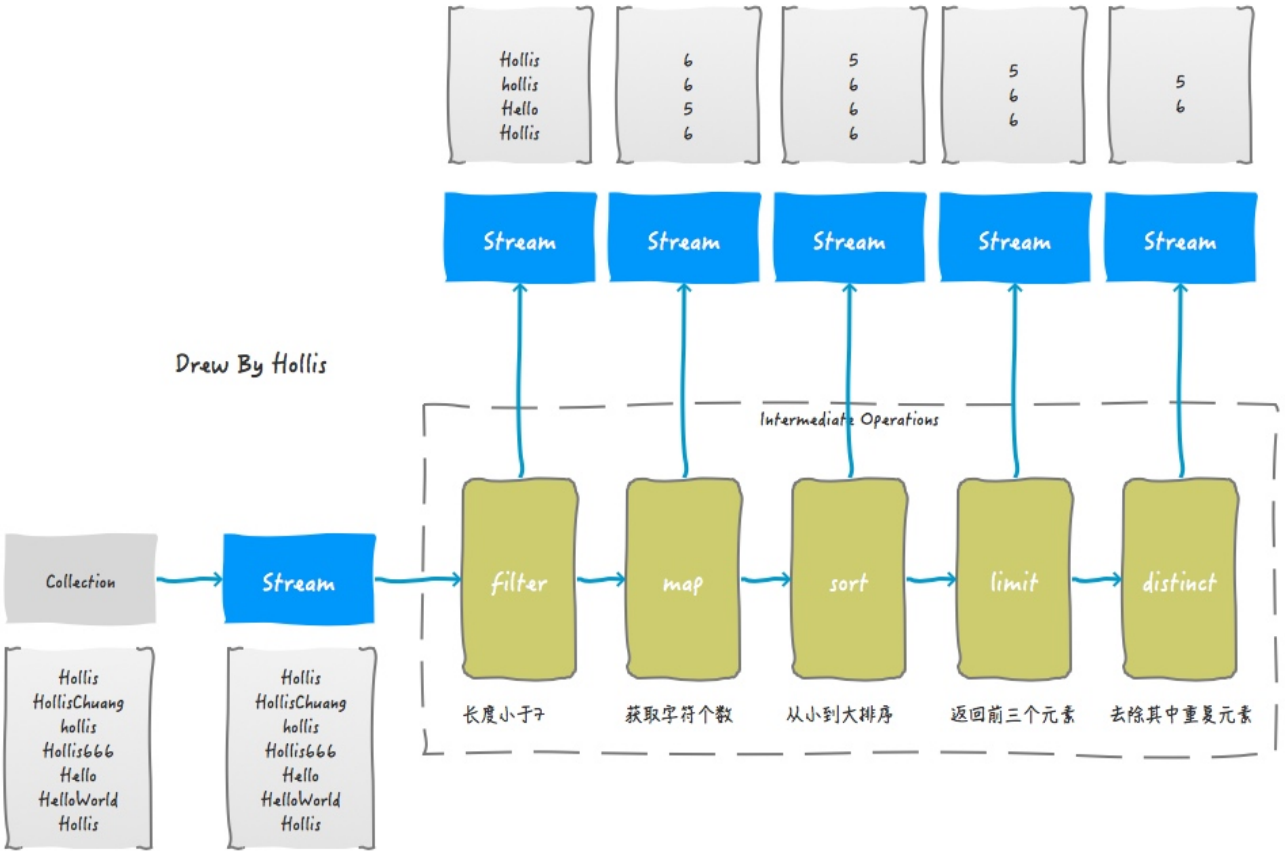
```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
numbers.stream().distinct().forEach(System.out::println);
//3,2,7,5
```

接下来我们通过一个例子和一张图，来演示下，当一个Stream先后通过filter、map、sort、limit以及distinct处理后会发生什么。

代码如下：

```
List<String> strings = Arrays.asList("Hollis", "HollisChuang", "hollis", "Hello", "HelloWorld", "Hollis");
Stream s = strings.stream().filter(string -> string.length()<= 6).map(String::length).sorted().limit(3)
            .distinct();
```

过程及每一步得到的结果如下图：



Stream最终操作

Stream的中间操作得到的结果还是一个Stream，那么如何把一个Stream转换成我们需要的类型呢？比如计算出流中元素的个数、将流装换成集合等。这就需要最终操作（terminal operation）

最终操作会消耗流，产生一个最终结果。也就是说，在最终操作之后，不能再次使用流，也不能在使用任何中间操作，否则将抛出异常：

```
java.lang.IllegalStateException: stream has already been operated upon or closed
```

俗话说，“你永远不会两次踏入同一条河”也正是这个意思。

常用的最终操作如下图：

STREAM OPERATION	GOAL	INPUT
forEach	For every item, outputs something	Consumer
count	Counts current items	
collect	Reduces the stream into a desired collection	

forEach

Stream 提供了方法 ‘forEach’ 来迭代流中的每个数据。以下代码片段使用 forEach 输出了10个随机数：

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

count

count用来统计流中的元素个数。

```
List<String> strings = Arrays.asList("Hollis", "HollisChuang", "hollis","Hollis666", "Hello", "HelloWorld",
"Hollis");
System.out.println(strings.stream().count());
//7
```

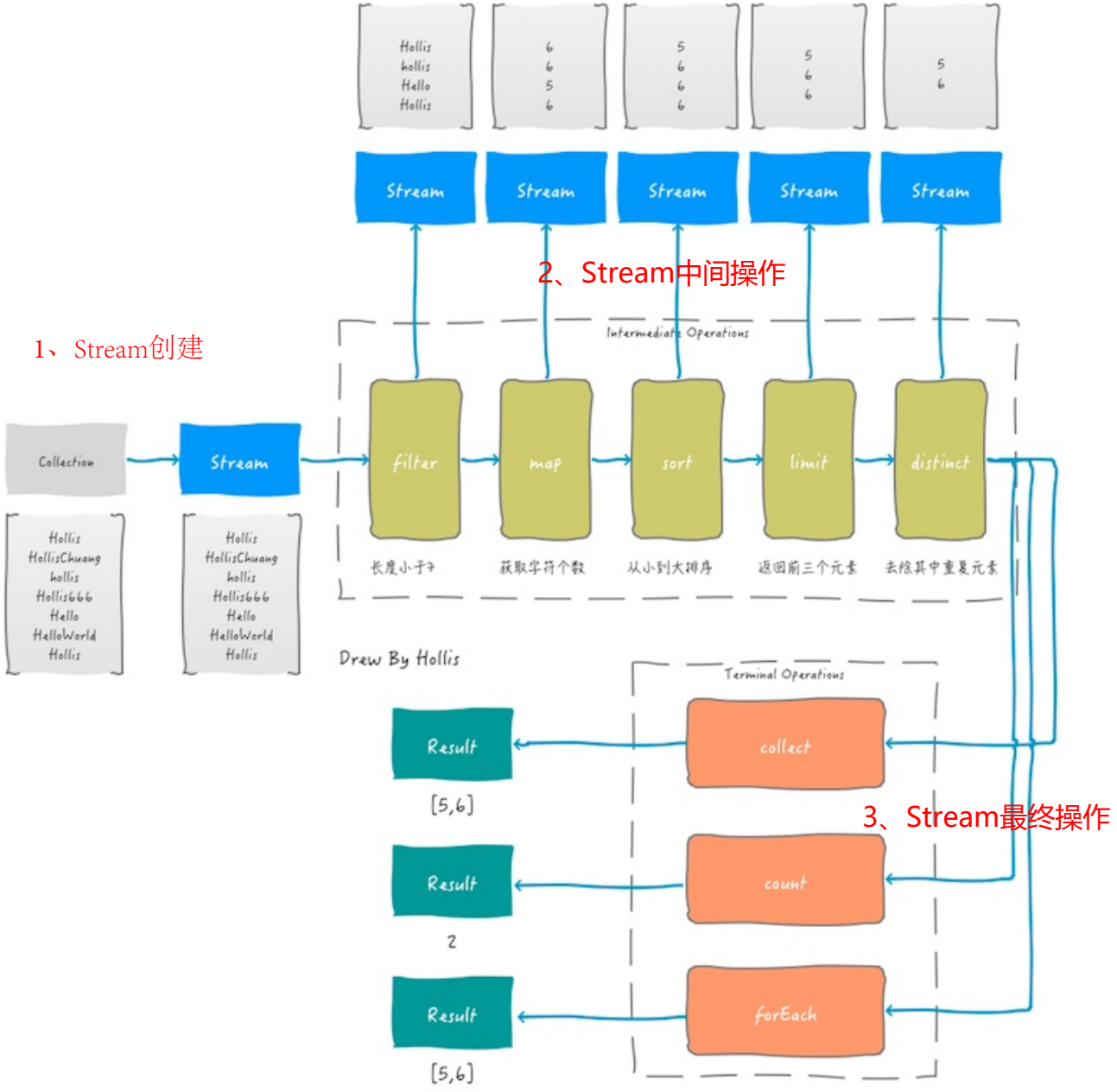
collect

collect就是一个归约操作，可以接受各种做法作为参数，将流中的元素累积成一个汇总结果：

```
List<String> strings = Arrays.asList("Hollis", "HollisChuang", "hollis","Hollis666", "Hello", "HelloWorld",
"Hollis");
strings  = strings.stream().filter(string -> string.startsWith("Hollis")).collect(Collectors.toList());
System.out.println(strings);
//Hollis, HollisChuang, Hollis666, Hollis
```

接下来，我们还是使用一张图，来演示下，前文的例子中，当一个Stream先后通过filter、map、sort、limit以及distinct处理后会，在分别使用不同的最终操作可以得到怎样的结果：

下图，展示了文中介绍的所有操作的位置、输入、输出以及使用一个案例展示了其结果。



总结

本文介绍了Java 8中的Stream 的用途，优点等。还接受了Stream的几种用法，分别是Stream创建、中间操作和最终操作。

Stream的创建有两种方式，分别是通过集合类的stream方法、通过Stream的of方法。

Stream的中间操作可以用来处理Stream，中间操作的输入和输出都是Stream，中间操作可以是过滤、转换、排序等。

Stream的最终操作可以将Stream转成其他形式，如计算出流中元素的个数、将流装换成集合、以及元素的遍历等。

(全文完)