# 一、学习目标

1. 能够使用@FunctionalInterface注解

2. 能够自定义无参无返回函数式接口

3. 能够自定义有参有返回函数式接口

4. 能够理解**Lambda延迟执行**的特点

5. 能够使用Lambda作为方法的参数

6. 能够使用Lambda作为方法的返回值

7. 能够使用**Supplier**函数式接口

8. 能够使用**Consumer**函数式接口

9. 能够使用**Function**函数式接口

10. 能够使用**Predicate**函数式接口

# 第一章 函数式接口

## 1.1 Lambda

一般都叫Lambda表达式（λ expression）是一个[匿名函数](https://baike.baidu.com/item/%E5%8C%BF%E5%90%8D%E5%87%BD%E6%95%B0/4337265)，Lambda表达式基于数学中的[λ演算](https://baike.baidu.com/item/%CE%BB%E6%BC%94%E7%AE%97)得名，直接对应于其中的lambda抽象（lambda abstraction），是一个匿名函数，即**没有函数名的函数**。Lambda表达式可以表示[闭包](https://baike.baidu.com/item/%E9%97%AD%E5%8C%85/10908873)（注意和数学传统意义上的不同）。

## 1.2 编程范式

常见的编程范式有**命令式编程（Imperative programming）**,**函数式编程**，**逻辑式编程**。常见的面向对象是一种命令式编程。

**命令式编程：**是面向计算机的抽象，有**变量**（对应着存储单元），**赋值语句**（获取，存储），**表达式**（内存引用和算术运算），**控制语句**（跳转、循环等），一句话：命令式编程是一个**冯诺依曼机**的**指令序列**。

**函数式编程：**是面向数学的抽象，将计算描述为一种**表达式求值**，一句话：函数式程序就是一个**表达式**。

## 1.3 函数式接口

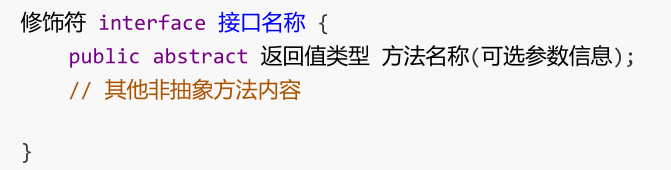
有且仅有一个抽象方法的接口。这个接口就是函数，就是一个表达式，所以可以使用Lambda表达式。

即适用于函数式编程场景的接口。而Java中的函数式编程体现就是Lambda，所以函数式接口就是可以适用于Lambda使用的接口。只有确保接口中有且仅有一个抽象方法，Java中的Lambda才能顺利地进行推导。

**“语法糖”**是指使用更加方便，但是原理不变的代码语法。例如在遍历集合时使用的for-each语法，其实底层的实现原理仍然是迭代器，这便是“语法糖”。从应用层面来讲，Java中的Lambda可以被当做是匿名内部类的“语法糖”，但是二者在原理上是不同的。

## 1.4 格式

只要确保接口中有且仅有一个**抽象**方法即可：可以包含其他方法（默认，静态，私有）



由于接口当中抽象方法的 public abstract 是可以省略的，所以定义一个函数式接口很简单：

public interface MyfactionaInterface {  
 void myMethod();  
}

## 1.5 @FunctionalInterface注解

与 @Override 注解的作用类似，Java 8中专门为函数式接口引入了一个新的注解： @FunctionalInterface 。该注解可用于一个接口的定义上：

@FunctionalInterface  
public interface MyfactionaInterface {  
 void myMethod();  
}

@FunctionalInterface作用：

是函数式接口：编译成功；

非函数式接口：编译失败；

## 1.6 自定义函数式接口

*/\*\*  
 \* 函数是接口的使用：  
 \* 1.作为参数  
 \* 2.作为返回值  
 \*  
 \** ***@author*** *: lipu  
 \** ***@since*** *: 2020-08-04 20:20  
 \*/*public class Demo {  
  
 *//定义一个方法，参数使用函数是接口MyfunctionaInterface* public static void show(MyfunctionaInterface myInter){  
 myInter.myMethod();  
 }  
  
 public static void main(String[] args) {  
 *//传递其实现类  
 show*(new MyfunctionaInterfaceImpl());  
  
 *//直接传递匿名内部类  
 show*(new MyfunctionaInterface() {  
 @Override  
 public void myMethod() {  
 System.*out*.println("我是MyfunctionaInterface的匿名内部实现类");  
 }  
 });  
  
 *//使用Lambda表达式  
 show*(()->{  
 System.*out*.println("使用完整的Lambda表达式重写接口的方法");  
 });  
 *show*(()-> System.*out*.println("使用简化的Lambda表达式重写接口的方法"));  
 }  
}

# 第二章 函数式编程

## 2.1 Lambda的延迟特性

有些场景的代码执行后，结果不一定会被使用，从而造成性能浪费。而Lambda表达式是延迟执行的，这正好可以作为解决方案，提升性能。

### 性能浪费的日志案例

注:日志可以帮助我们快速的定位问题，记录程序运行过程中的情况，以便项目的监控和优化。

一种典型的场景就是对参数进行有条件使用，例如对日志消息进行拼接后，在满足条件的情况下进行打印输出：

public class LoggerDemo01 {  
  
 public static void showLog(int level, String message){  
 *//对日志级别进行判断，如果是1级别，name输出日志信息* if (level == 1) {  
 System.*out*.println(message);  
 }  
 }  
  
 public static void main(String[] args) {  
 *//定义三个日志信息* String msg1 = "Hello";  
 String msg2 = "World";  
 String msg3 = "Java";  
  
 *showLog*(1,msg1+msg2+msg3);  
 }  
}

这段代码存在问题：无论级别是否满足要求，作为 log 方法的第二个参数，三个字符串一定会首先被拼接并传入方法内，然后才会进行级别判断。如果级别不符合要求，那么字符串的拼接操作就白做了，存在性能浪费。

**备注：**SLF4J是应用非常广泛的日志框架，它在记录日志时为了解决这种性能浪费的问题，并不推荐首先进行字符串的拼接，而是将字符串的若干部分作为可变参数传入方法中，仅在日志级别满足要求的情况下才会进行字符串拼接。例如： LOGGER.debug("变量{}的取值为{}。", "os", "macOS") ，其中的大括号 {} 为占位符。如果满足日志级别要求，则会将“os”和“macOS”两个字符串依次拼接到大括号的位置；否则不会进行字符串拼接。这也是一种可行解决方案，但Lambda可以做到更好。

### 体验Lambda的更优写法

使用Lambda必然需要一个函数式接口：

@FunctionalInterface  
public interface MessageBuilder {  
  
 String builderMessage();  
}

然后对 log 方法进行改造：

public class LoggerLambdaDemo02 {  
  
 public static void showLog(int level, MessageBuilder message){  
 *//对日志级别进行判断，如果是1级别，name输出日志信息* if (level == 1) {  
 System.*out*.println(message.builderMessage());  
 }  
 }  
  
 public static void main(String[] args) {  
 *//定义三个日志信息* String msg1 = "Hello ";  
 String msg2 = "World ";  
 String msg3 = "Java";  
  
 *showLog*(1,()->msg1+msg2+msg3);

这样一来，只有当级别满足要求的时候，才会进行三个字符串的拼接；否则三个字符串将不会进行拼接。

### 证明Lambda的延迟

下面的代码可以通过结果进行验证：

public class LoggerDelayDemo03 {  
  
 public static void showLog(int level, MessageBuilder message){  
 *//对日志级别进行判断，如果是1级别，name输出日志信息* if (level == 1) {  
 System.*out*.println(message.builderMessage());  
 }  
 }  
  
 public static void main(String[] args) {  
 *//定义三个日志信息* String msg1 = "Hello ";  
 String msg2 = "World ";  
 String msg3 = "Java";  
  
 *showLog*(2,()->{  
 System.*out*.println("Lambda执行");return msg1+msg2+msg3;});  
 }  
}

从结果中可以看出，在不符合级别要求的情况下，Lambda将不会执行。从而达到节省性能的效果。

扩展：实际上使用内部类也可以达到同样的效果，只是将代码操作延迟到了另外一个对象当中通过调用方法来完成。而是否调用其所在方法是在条件判断之后才执行的。

## 2.2使用Lambda作为参数和返回值

参数

public class Demo01Runnable {  
  
 private static void startThread(Runnable task){  
 new Thread(task).start();  
 }  
  
 public static void main(String[] args) {  
 for (int i = 0; i < 10; i++) {  
 *startThread*(() -> System.*out*.println(Thread.*currentThread*().getName() + "线程执行"));  
 }  
 }  
}

返回值

public class Demo02Comparator {  
  
 private static Comparator<String> newComparator(){  
 return (a,b)->b.length() - a.length();*//谁在前，为true则在前面* }  
  
  
 public static void main(String[] args) {  
 String[] arr = {"abc","ab","abcd","a"};  
 System.*out*.println(Arrays.*toString*(arr));  
 Arrays.*sort*(arr,*newComparator*());  
 System.*out*.println(Arrays.*toString*(arr));  
 }  
}

# 第三章 常用的函数式接口

JDK提供了大量常用的函数式接口以丰富Lambda的典型使用场景，它们主要在 java.util.function 包中被提供。

## 3.1 Supplier接口

java.util.function.Supplier<T> 接口（生产型接口）仅包含一个无参的方法： T get() 。用来获取一个泛型参数指定类型的对象数据。由于这是一个函数式接口，这也就意味着对应的Lambda表达式需要“对外提供”一个符合泛型类型的对象数据。

public class Demo01Supplier {  
  
 private static String getString(Supplier<String> supplier){  
 return supplier.get();  
 }  
  
 public static void main(String[] args) {  
 String msg1 = "Hello ";  
 String msg2 = "World";  
  
 System.*out*.println(*getString*(()->msg1+msg2));  
 }  
}

求数组元素最大值

public class Demo02Test {  
  
 private static int getMax(Supplier<Integer> supplier){  
 return supplier.get();  
 }  
  
 public static void main(String[] args) {  
 int[] arr = {2,234,45,6,7,88,954,34,5,20};  
  
 int maxNum = *getMax*(() -> {  
 int max = arr[0];  
 for (int i : arr) {  
 if (i > max) {  
 max = i;  
 }  
 }  
 return max;  
 });  
 System.*out*.println("最大值为："+maxNum);  
 }  
}

## 3.2 Consumer接口

java.util.function.Consumer<T> 接口（消费型接口）则正好与Supplier接口相反，它不是生产一个数据，而是消费一个数据，其数据类型由泛型决定。

### 抽象方法：accept

Consumer 接口中包含抽象方法 void accept(T t) ，意为消费一个指定泛型的数据。基本使用如：

public class Demo01Consumer {  
  
 private static void consumerString(String name,Consumer<String> consumer){  
 consumer.accept(name);  
 }  
  
 public static void main(String[] args) {  
 *consumerString*("Hello",s-> System.*out*.println(s.toUpperCase()));  
 }  
}

当然，更好的写法是使用方法引用。

### 默认方法：andThen

如果一个方法的参数和返回值全都是 Consumer 类型，那么就可以实现效果：消费数据的时候，首先做一个操作，然后再做一个操作，实现组合。而这个方法就是 Consumer 接口中的default方法 andThen 。下面是JDK的源代码：

default Consumer<T> andThen(Consumer<? super T> after) {  
 Objects.*requireNonNull*(after);  
 return (T t) -> { accept(t); after.accept(t); };  
}

备注： java.util.Objects 的 requireNonNull 静态方法将会在参数为null时主动抛出NullPointerException 异常。这省去了重复编写if语句和抛出空指针异常的麻烦。

要想实现组合，需要两个或多个Lambda表达式即可，而 andThen 的语义正是“一步接一步”操作。例如两个步骤组合的情况：

*/\*\*  
 \* 谁写前面谁先消费  
 \*  
 \** ***@author*** *: lipu  
 \** ***@since*** *: 2020-08-04 22:30  
 \*/*public class Demo02AndThen {  
  
 public static void method(String s, Consumer<String> consumer1, Consumer<String> consumer2){  
*// consumer1.accept(s);  
// consumer2.accept(s);* consumer1.andThen(consumer2).accept(s);  
 }  
  
 public static void main(String[] args) {  
 *method*("Hello",(t)->{  
 System.*out*.println(t.toUpperCase());  
 },(t)-> System.*out*.println(t.toLowerCase()));  
 }  
  
}

**格式化打印信息**

下面的字符串数组当中存有多条信息，请按照格式“ 姓名：XX。性别：XX。 ”的格式将信息打印出来。要求将打印姓名的动作作为第一个 Consumer 接口的Lambda实例，将打印性别的动作作为第二个 Consumer 接口的Lambda实例，将两个 Consumer 接口按照顺序“拼接”到一起。

public class DemoTest {  
  
 public static void main(String[] args) {  
 String[] arr = {"迪丽热巴,女","马尔扎哈,男","古力娜扎,女"};  
  
 *printInfo*(arr,t->{  
 System.*out*.print("姓名："+t.split(",")[0]);  
 },t->{  
 System.*out*.println("，性别："+t.split(",")[1]+"。");  
 });  
 }  
  
 private static void printInfo(String[] arr,Consumer<String> consumer1,Consumer<String> consumer2){  
 for (String s : arr) {  
 consumer1.andThen(consumer2).accept(s);  
 }  
 }  
}

## 3.3 Predicate接口

有时候我们需要对某种类型的数据进行**判断**，从而得到一个boolean值结果。这时可以使用

### 抽象方法：test

Predicate 接口中包含一个抽象方法： boolean test(T t) 。用于条件判断的场景：

public class Demo01Predicate {  
  
 private static void method(String name,Predicate<String> predicate){  
 boolean b = predicate.test(name);  
 System.*out*.println("字符串的是否超过10："+b);  
 }  
  
 public static void main(String[] args) {  
 *method*("aaswedsdw2sss",t->t.length()>10);  
 }  
}

条件判断的标准是传入的Lambda表达式逻辑，只要字符串长度大于10则认为很长。

### 默认方法：and

既然是条件判断，就会存在与、或、非三种常见的逻辑关系。其中将两个 Predicate 条件使用“与”逻辑连接起来实现“并且”的效果时，可以使用default方法 and 。其JDK源码为：

default Predicate<T> and(Predicate<? super T> other) {  
 Objects.*requireNonNull*(other);  
 return (t) -> test(t) && other.test(t);  
}

如果要判断一个字符串既要包含大写“H”，又要包含大写“W”，那么：

public class Demo02PredicateAnd {  
  
 private static void method(String name, Predicate<String> predicate1, Predicate<String> predicate2){  
 boolean b = predicate1.and(predicate2).test(name);  
 System.*out*.println("字符串是否同时包含H和W："+b);  
 }  
  
 public static void main(String[] args) {  
 *method*("HelloWorld",t->t.contains("H"),t->t.contains("W"));  
 }  
}

### 默认方法：or

与 and 的“与”类似，默认方法 or 实现逻辑关系中的“或”。JDK源码为

default Predicate<T> or(Predicate<? super T> other) {  
 Objects.*requireNonNull*(other);  
 return (t) -> test(t) || other.test(t);  
}

如果希望实现逻辑“字符串包含大写H或者包含大写W”，那么代码只需要将“and”修改为“or”名称即可，其他都不变：

public class Demo03PredicateOr {  
  
 private static void method(String name, Predicate<String> predicate1, Predicate<String> predicate2){  
 boolean b = predicate1.or(predicate2).test(name);  
 System.*out*.println("字符串是否同时包含H和W："+b);  
 }  
  
 public static void main(String[] args) {  
 *method*("HelloWorld",t->t.contains("H"),t->t.contains("M"));  
 }  
}

### 默认方法：negate

“与”、“或”已经了解了，剩下的“非”（取反）也会简单。默认方法 negate 的JDK源代码为：

default Predicate<T> negate() {  
 return (t) -> !test(t);  
}

从实现中很容易看出，它是执行了test方法之后，对结果boolean值进行“!”取反而已。一定要在 test 方法调用之前调用 negate 方法，正如 and 和 or 方法一样：

public class Demo04PredicateNegate {  
  
 private static void method(String name, Predicate<String> predicate1, Predicate<String> predicate2){  
 boolean b = predicate1.negate().test(name);  
 System.*out*.println("字符串是否不包含H："+b);  
 }  
  
 public static void main(String[] args) {  
 *method*("HelloWorld",t->t.contains("H"),t->t.contains("M"));  
 }  
}

### 集合信息筛选

数组当中有多条“姓名+性别”的信息如下，请通过 Predicate 接口的拼装将符合要求的字符串筛选到集合ArrayList 中，需要同时满足两个条件：

1. 必须为女生；

2. 姓名为4个字。

public class DemoTest {  
  
 public static void main(String[] args) {  
 String[] arr = {"迪丽热巴,女","马尔扎哈,男","古力娜扎,女"};  
  
 List<String> list = *filter*(arr, t -> "女".equals(t.split(",")[1]),  
 t -> t.split(",")[0].length() == 4);  
 System.*out*.println(list);  
 }  
  
  
 private static List<String> filter(String[] arr, Predicate<String> predicate1,Predicate<String> predicate2){  
 ArrayList<String> list = new ArrayList<>();  
 for (String info : arr) {  
 if (predicate1.and(predicate2).test(info)) {  
 list.add(info);  
 }  
 }  
 return list;  
 }  
}

## 3.3 Function接口

java.util.function.Function<T,R> 接口（转换型）用来根据一个类型的数据得到另一个类型的数据，前者称为前置条件，后者称为后置条件。

### 抽象方法：apply

Function 接口中最主要的抽象方法为： R apply(T t) ，根据类型T的参数获取类型R的结果。

使用的场景例如：将 String 类型转换为 Integer 类型。

public class Demo01Function {  
 private static void method(String name,Function<String,Integer> function){  
 Integer num = function.apply(name);  
 System.*out*.println(num+10);  
 }  
  
 public static void main(String[] args) {  
 *method*("10",t->Integer.*valueOf*(t));  
 }  
}

当然，最好是通过方法引用的写法。

### 默认方法：andThen

Function 接口中有一个默认的 andThen 方法，用来进行组合操作。JDK源代码如：

default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {  
 Objects.*requireNonNull*(after);  
 return (T t) -> after.apply(apply(t));  
}

该方法同样用于“先做什么，再做什么”的场景，和 Consumer 中的 andThen 差不多：

public class Demo02FunctionAndThe {  
  
 private static void method(String name, Function<String,Integer> function1,Function<Integer,Integer> function2){  
 Integer num = function1.andThen(function2).apply(name);  
 System.*out*.println(num+10);  
 }  
  
 public static void main(String[] args) {  
 *method*("10",t->Integer.*valueOf*(t),t->t-2);  
 }  
}

请注意，Function的前置条件泛型和后置条件泛型可以相同。

### 自定义函数模型拼接

题目：请使用 Function 进行函数模型的拼接，按照顺序需要执行的多个函数操作为：

String str = "赵丽颖,20";

1. 将字符串截取数字年龄部分，得到字符串；

2. 将上一步的字符串转换成为int类型的数字；

3. 将上一步的int数字累加100，得到结果int数字。

public class Demo03Test {  
  
 public static void main(String[] args) {  
 String str = "赵丽颖,20";  
 int ageNum = *getAgeNum*(str, t -> Integer.*valueOf*(t.split(",")[1]), t -> t + 100);  
 System.*out*.println(ageNum);  
 }  
  
 private static int getAgeNum(String str, Function<String,Integer> function1,Function<Integer , Integer> function2){  
 return function1.andThen(function2).apply(str);  
 }  
}

# 二、主要内容

Stream流

方法引用

# 教学目标

能够理解**流与集合**相比的优点

能够理解**流的延迟执行特点**

能够通过**集合、映射或数组获取流**

能够掌握常用的**流操作**

能够使用输出语句的**方法引用3**

能够通过**4种方式使用方法引用**

能够使用**类和数组的构造器引用8**

# 第一章 Stream流

说到Stream便容易想到I/O Stream，而实际上，谁规定“流”就一定是“IO流”呢？在Java 8中，得益于Lambda所带来的函数式编程，引入了一个全新的Stream概念，用于解决已有集合类库既有的弊端。

## 1.1引言

**传统集合的多步遍历代码**

几乎所有的集合（如 Collection 接口或 Map 接口等）都支持直接或间接的遍历操作。而当我们需要对集合中的元素进行操作的时候，除了必需的添加、删除、获取外，最典型的就是集合遍历。例如：

public class Demo01List {  
 public static void main(String[] args) {  
 ArrayList<String> list = new ArrayList<>();  
 list.add("张无忌");  
 list.add("周芷若");  
 list.add("赵敏");  
 list.add("张强");  
 list.add("张三丰");  
  
 *//对list集合遍历，以张开头的元素存到一个集合，三个字的存到一个集合* ArrayList<String> A = new ArrayList<>();  
 for (String s : list) {  
 if (s.startsWith("张")) {  
 A.add(s);  
 }  
 }  
  
 ArrayList<String> B = new ArrayList<>();  
 for (String s : list) {  
 if (s.length() ==3) {  
 B.add(s);  
 }  
 }  
  
 System.*out*.println(A);  
 System.*out*.println(B);  
  
 }  
}

### 循环遍历的弊端

这段代码中含有三个循环，每一个作用不同：

1. 首先筛选所有姓张的人；

2. 然后筛选名字有三个字的人；

3. 最后进行对结果进行打印输出。

每当我们需要对集合中的元素进行操作的时候，总是需要进行循环、循环、再循环。这是理所当然的么？不是。循环是做事情的方式，而不是目的。另一方面，使用线性循环就意味着只能遍历一次。如果希望再次遍历，只能再使用另一个循环从头开始。

那，Lambda的衍生物Stream能给我们带来怎样更加优雅的写法呢？

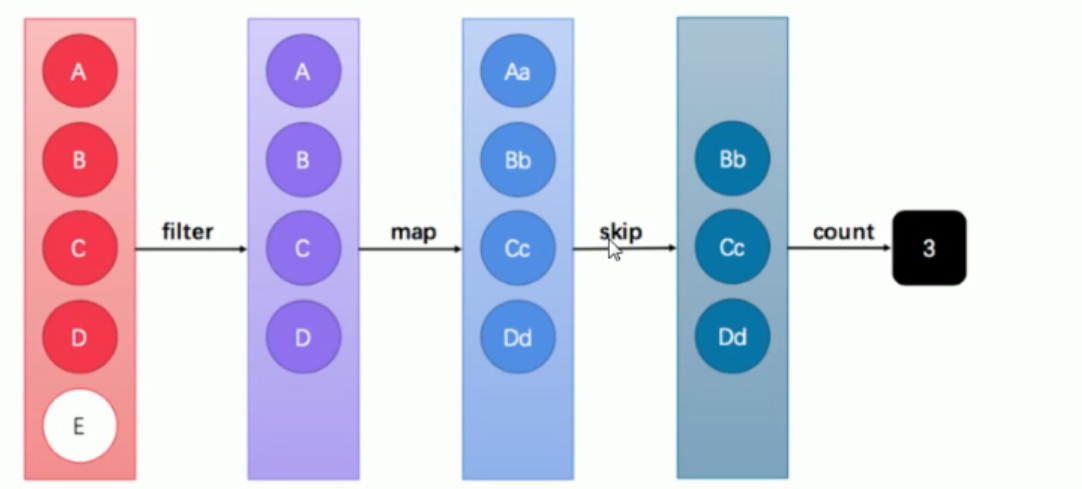
### Stream的更优写法

public class Demo02StreamFilter {  
 public static void main(String[] args) {  
 ArrayList<String> list = new ArrayList<>();  
 list.add("张无忌");  
 list.add("周芷若");  
 list.add("赵敏");  
 list.add("张强");  
 list.add("张三丰");  
  
 ArrayList<String> A = new ArrayList<>();  
 ArrayList<String> B = new ArrayList<>();  
 list.stream().filter(s->s.startsWith("张")).forEach(a->A.add(a));  
 list.stream().filter(s->s.length()==3).forEach(a->B.add(a));  
 System.*out*.println(A);  
 System.*out*.println(B);  
 }  
}

## 1.2 流式思想概述

整体来看，流式思想类似于工厂车间的“生产流水线”。

当需要对多个元素进行操作（特别是多步操作）的时候，考虑到性能及便利性，我们应该首先拼好一个“模型”步骤方案，然后再按照方案去执行它。



这张图中展示了过滤、映射、跳过、计数等多步操作，这是一种集合元素的处理方案，而方案就是一种“函数模型”。图中的每一个方框都是一个“流”，调用指定的方法，可以从一个流模型转换为另一个流模型。而最右侧的数字3是最终结果。

这里的 filter 、 map 、 skip 都是在对函数模型进行操作，集合元素并没有真正被处理。只有当终结方法 count执行的时候，整个模型才会按照指定策略执行操作。而这得益于Lambda的延迟执行特性。

**备注：**“Stream流”其实是一个集合元素的函数模型，它并不是集合，也不是数据结构，其本身并不存储任何元素（或其地址值）。

stream（流）是一个来自数据源的元素队列

元素是特定类型的对象，形成一个队列。 Java中的Stream并不会存储元素，而是按需计算。

**数据源** 流的来源。 可以是集合，数组 等。

和以前的Collection操作不同， Stream操作还有两个基础的特征：

**Pipelining:** 中间操作都会返回流对象本身。 这样多个操作可以串联成一个管道， 如同流式风格（fluentstyle）。 这样做可以对操作进行优化， 比如延迟执行(laziness)和短路( short-circuiting)。

**内部迭代：** 以前对集合遍历都是通过Iterator或者增强for的方式, 显式的在集合外部进行迭代， 这叫做外部迭代。 Stream提供了内部迭代的方式，流可以直接调用遍历方法。

当使用一个流的时候，通常包括三个基本步骤：获取一个数据源（source）→ 数据转换→执行操作获取想要的结果，每次转换原有 Stream 对象不改变，返回一个新的 Stream 对象（可以有多次转换），这就允许对其操作可以像链条一样排列，变成一个管道。

## 1.3获取流

java.util.stream.Stream<T> 是Java 8新加入的最常用的流接口。（这并不是一个函数式接口。）

获取一个流非常简单，有以下几种常用的方式：

1.所有的 Collection 集合（单列集合）都可以通过 stream 默认方法获取流；

2.Stream 接口的静态方法 of 可以获取数组对应的流。（Map集合）

### 根据Collection获取流

首先， java.util.Collection 接口中加入了default方法 stream 用来获取流，所以其所有实现类均可获取流。

### 根据Map获取流

### 根据数组获取流

public class Demo03Stream {  
 public static void main(String[] args) {  
 ArrayList<String> list = new ArrayList<>();  
 Stream<String> stream = list.stream();  
  
 HashMap<String, String> map = new HashMap<>();  
 int[] a = {1,2,3,4};  
 Stream<HashMap<String, String>> map1 = Stream.*of*(map);  
 Stream<int[]> a1 = Stream.*of*(a);  
 }  
}

## 1.4常用方法

流模型的操作很丰富，这里介绍一些常用的API。这些方法可以被分成两种：

**延迟方法：**返回值类型仍然是 Stream 接口自身类型的方法，因此支持链式调用。（除了终结方法外，其余方法均为延迟方法。）

**终结方法：**返回值类型不再是 Stream 接口自身类型的方法，因此不再支持类似 StringBuilder 那样的链式调用。本小节中，终结方法包括 count 和 forEach 方法。

备注：本小节之外的更多方法，请自行参考API文档。

### 1.4.1 逐一处理 forEach

### 1.4.2 过滤 filter

public class Demo04Stream\_forEach {  
 public static void main(String[] args) {  
 Stream<String> stream = Stream.*of*("张三", "李四", "王五", "赵六", "田七");  
*// stream.forEach(name-> System.out.println(name));* stream.filter(name->name.contains("张")).forEach(name-> System.*out*.println(name));  
 }  
}

### .4.3 映射 map

如果需要将流中的元素映射到另一个流中，可以使用 map 方法。方法签名：

### 1.4.4 统计个数 count

正如旧集合 Collection 当中的 size 方法一样，流提供 count 方法来数一数其中的元素个数：

public class Demo05Stream\_Map {  
  
 public static void main(String[] args) {  
 Stream<String> stream = Stream.*of*("10", "12","2");  
 Stream<Integer> stream1 = stream.map(s -> Integer.*parseInt*(s));  
 System.*out*.println(stream1.count());  
 }  
}

### 1.4.5 取前几个 limit

public class Demo05Stream\_Map {  
  
 public static void main(String[] args) {  
 Stream<String> stream = Stream.*of*("10", "12","2");  
 stream.map(s -> Integer.*parseInt*(s)).limit(2).forEach(s-> System.*out*.println(s));  
 }  
}

### 1.4.6 跳过前几个 skip

public class Demo05Stream\_Map {  
  
 public static void main(String[] args) {  
 Stream<String> stream = Stream.*of*("10", "12","2");  
 stream.map(s -> Integer.*parseInt*(s)).skip(2).forEach(s-> System.*out*.println(s));  
 }  
}

### 1.4.7 组合 concat

public class Demo06Stream\_concat {  
 public static void main(String[] args) {  
 Stream<String> a = Stream.*of*("pupu");  
 Stream<String> b = Stream.*of*("琼");  
 Stream.*concat*(a, b).forEach(c-> System.*out*.println(c));  
 }  
}

### 题目一

现在有两个 ArrayList 集合存储队伍当中的多个成员姓名，要求使用传统的for循环（或增强for循环）依次进行以下若干操作步骤：

1. 第一个队伍只要名字为3个字的成员姓名；存储到一个新集合中。

2. 第一个队伍筛选之后只要前3个人；存储到一个新集合中。

3. 第二个队伍只要姓张的成员姓名；存储到一个新集合中。

4. 第二个队伍筛选之后不要前2个人；存储到一个新集合中。

5. 将两个队伍合并为一个队伍；存储到一个新集合中。

6. 根据姓名创建 Person 对象；存储到一个新集合中。

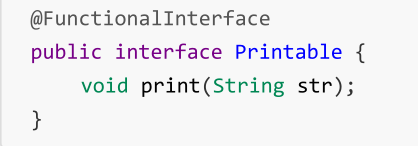
7. 打印整个队伍的Person对象信息。

public class Test1 {  
  
 public static void main(String[] args) {  
 *//第一支队伍* ArrayList<String> one = new ArrayList<>();  
 one.add("迪丽热巴");  
 one.add("宋远桥");  
 one.add("苏星河");  
 one.add("石破天");  
 one.add("石中玉");  
 one.add("老子");  
 one.add("庄子");  
 one.add("洪七公");  
 *//第二支队伍* ArrayList<String> two = new ArrayList<>();  
 two.add("古力娜扎");  
 two.add("张无忌");  
 two.add("赵丽颖");  
 two.add("张三丰");  
 two.add("尼古拉斯赵四");  
 two.add("张天爱");  
 two.add("张二狗");  
  
 Stream<String> streamone = one.stream().filter(name -> name.length() == 3).limit(3);  
  
 Stream<String> streamtwo = two.stream().filter(name -> name.startsWith("张")).skip(2);  
  
 Stream<String> concat = Stream.*concat*(streamone, streamtwo);  
  
 concat.map(Person::new).forEach(System.*out*::println);  
 }  
}

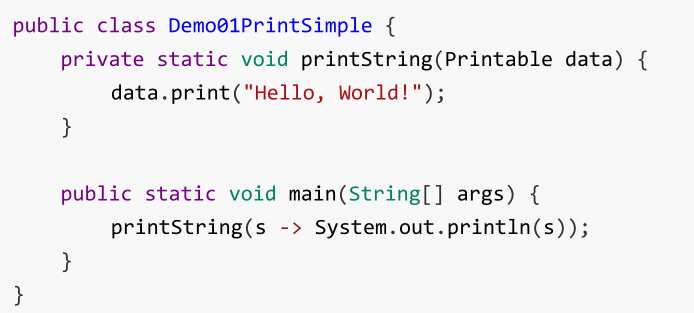
# 第二章 方法引用

在使用Lambda表达式的时候，我们实际上传递进去的代码就是一种解决方案：拿什么参数做什么操作。那么考虑一种情况：如果我们在Lambda中所指定的操作方案，已经有地方存在相同方案，那是否还有必要再写重复逻辑？

## 2.1 冗余的Lambda场景



在 Printable 接口当中唯一的抽象方法 print 接收一个字符串参数，目的就是为了打印显示它。那么通过Lambda来使用它的代码很简单：



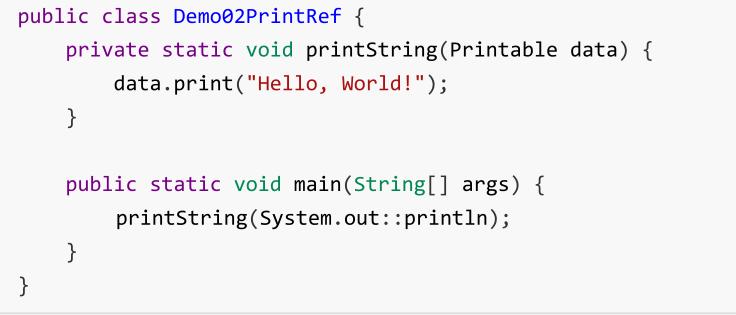
其中 printString 方法只管调用 Printable 接口的 print 方法，而并不管 print 方法的具体实现逻辑会将字符串打印到什么地方去。而 main 方法通过Lambda表达式指定了函数式接口 Printable 的具体操作方案为：**拿到String（类型可推导，所以可省略）数据后，在控制台中输出它**。

## 2.2 问题分析

这段代码的问题在于，对字符串进行控制台打印输出的操作方案，明明已经有了现成的实现，那就是 System.out对象中的 println(String) 方法。既然Lambda希望做的事情就是调用 println(String) 方法，那何必自己手动调用呢？

## 2.3 用方法引用改进代码

能否省去Lambda的语法格式（尽管它已经相当简洁）呢？只要“引用”过去就好了：



请注意其中的双冒号 :: 写法，这被称为“方法引用”，而双冒号是一种新的语法。

## 2.4 方法引用符

双冒号 :: 为引用运算符，而它所在的表达式被称为方法引用。如果Lambda要表达的函数方案已经存在于某个方法的实现中，那么则可以通过双冒号来引用该方法作为Lambda的替代者。

### 语义分析

例如上例中， System.out 对象中有一个重载的 println(String) 方法恰好就是我们所需要的。那么对于printString 方法的函数式接口参数，对比下面两种写法，完全等效：

Lambda表达式写法： s -> System.out.println(s);

方法引用写法： System.out::println

第一种语义是指：拿到参数之后经Lambda之手，继而传递给 System.out.println 方法去处理。

第二种等效写法的语义是指：直接让 System.out 中的 println 方法来取代Lambda。两种写法的执行效果完全一样，而第二种方法引用的写法复用了已有方案，更加简洁。

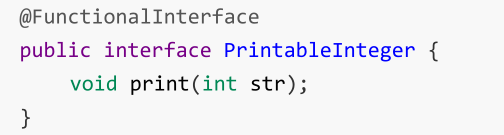
注:Lambda 中 传递的参数 一定是方法引用中 的那个方法可以接收的类型,否则会抛出异常

### 推导与省略

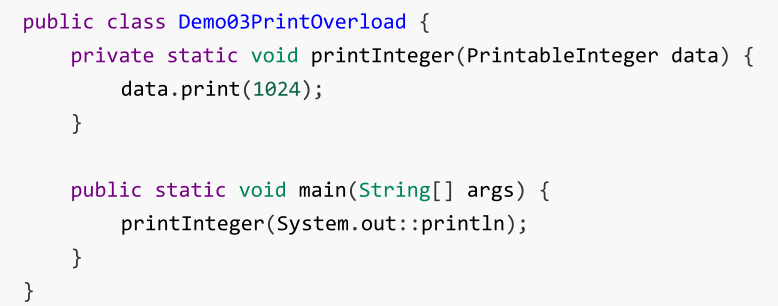
如果使用Lambda，那么根据“可推导就是可省略”的原则，无需指定参数类型，也无需指定的重载形式——它们都将被自动推导。而如果使用方法引用，也是同样可以根据上下文进行推导。

函数式接口是Lambda的基础，而方法引用是Lambda的孪生兄弟。

下面这段代码将会调用 println 方法的不同重载形式，将函数式接口改为int类型的参数：



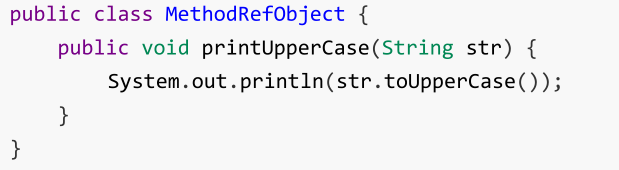
由于上下文变了之后可以自动推导出唯一对应的匹配重载，所以方法引用没有任何变化：



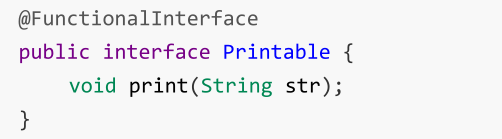
这次方法引用将会自动匹配到 println(int) 的重载形式。

## 2.5 通过对象名引用成员方法

这是最常见的一种用法，与上例相同。如果一个类中已经存在了一个成员方法：

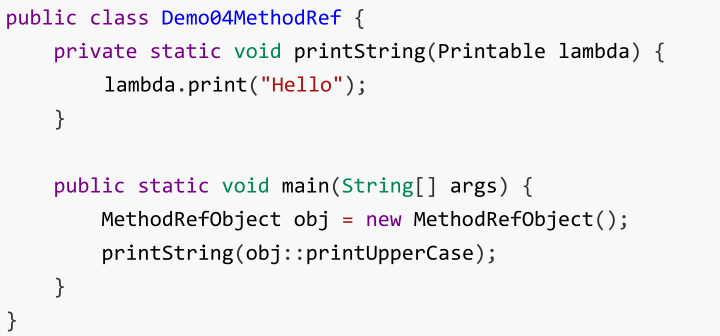


函数式接口仍然定义为：



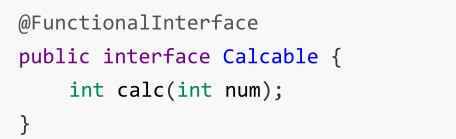
那么当需要使用这个 printUpperCase 成员方法来替代 Printable 接口的Lambda的时候，已经具有了

MethodRefObject 类的对象实例，则可以通过对象名引用成员方法，代码为：

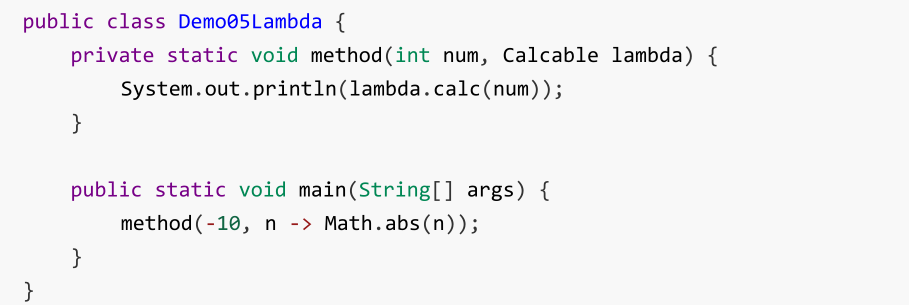


## 2.6 通过类名称引用静态方法

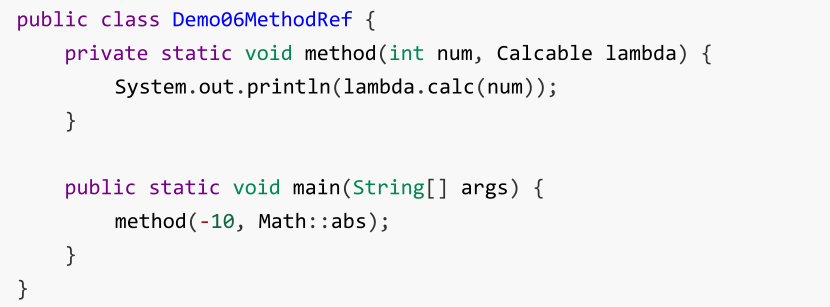
由于在 java.lang.Math 类中已经存在了静态方法 abs ，所以当我们需要通过Lambda来调用该方法时，有两种写法。首先是函数式接口：



第一种写法是使用Lambda表达式：



但是使用方法引用的更好写法是：



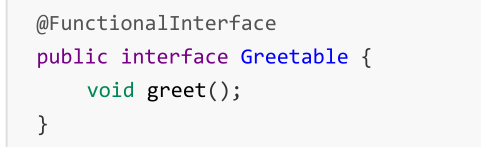
在这个例子中，下面两种写法是等效的：

Lambda表达式： n -> Math.abs(n)

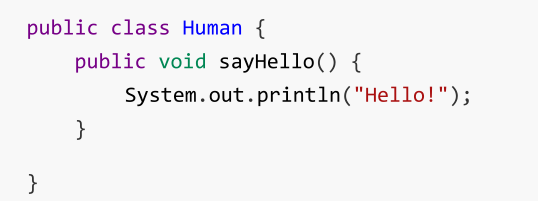
方法引用： Math::abs

## 2.7 通过super引用成员方法

如果存在继承关系，当Lambda中需要出现super调用时，也可以使用方法引用进行替代。首先是函数式接口：



然后是父类 Human 的内容：



最后是子类 Man 的内容，其中使用了Lambda的写法：



但是如果使用方法引用来调用父类中的 sayHello 方法会更好，例如另一个子类 Woman ：



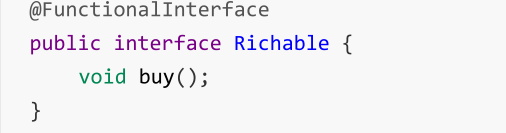
在这个例子中，下面两种写法是等效的：

Lambda表达式： () -> super.sayHello()

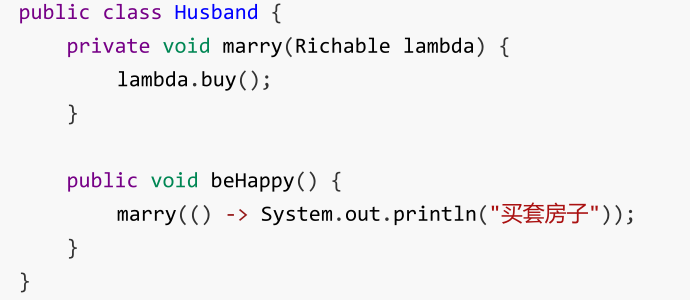
方法引用： super::sayHello

## 2.8 通过this引用成员方法

this代表当前对象，如果需要引用的方法就是当前类中的成员方法，那么可以使用“this::成员方法”的格式来使用方法引用。首先是简单的函数式接口：

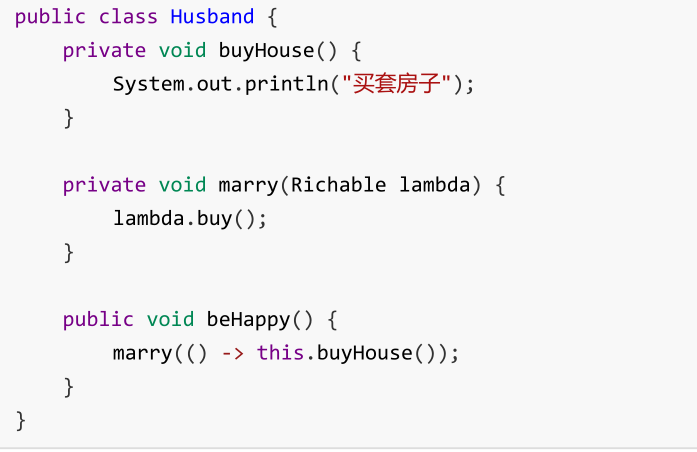


下面是一个丈夫 Husband 类：

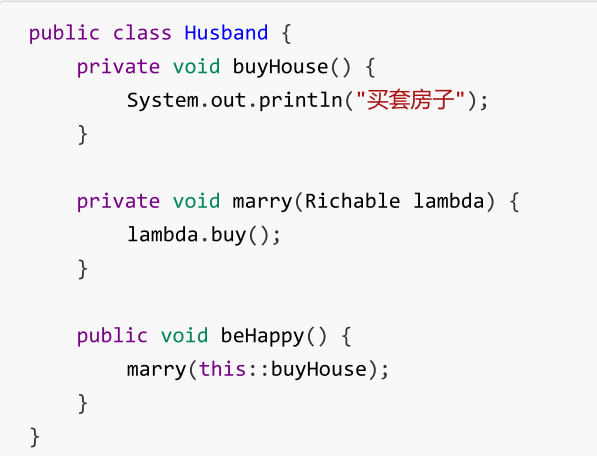


开心方法 beHappy 调用了结婚方法 marry ，后者的参数为函数式接口 Richable ，所以需要一个Lambda表达式。

但是如果这个Lambda表达式的内容已经在本类当中存在了，则可以对 Husband 丈夫类进行修改：

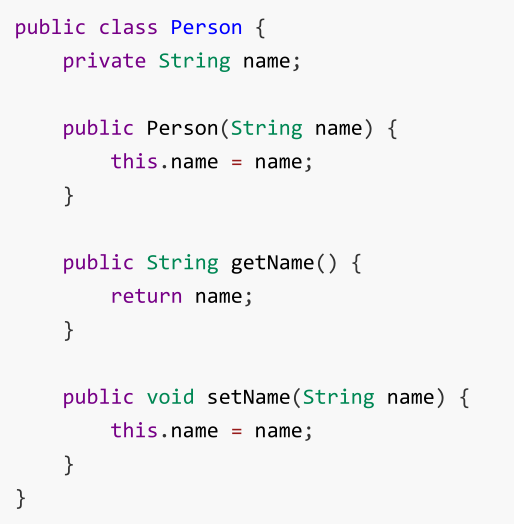


如果希望取消掉Lambda表达式，用方法引用进行替换，则更好的写法为：

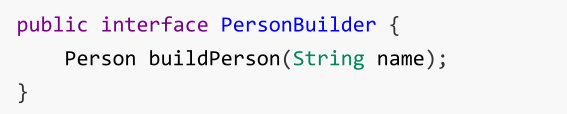


## 2.9 类的构造器引用

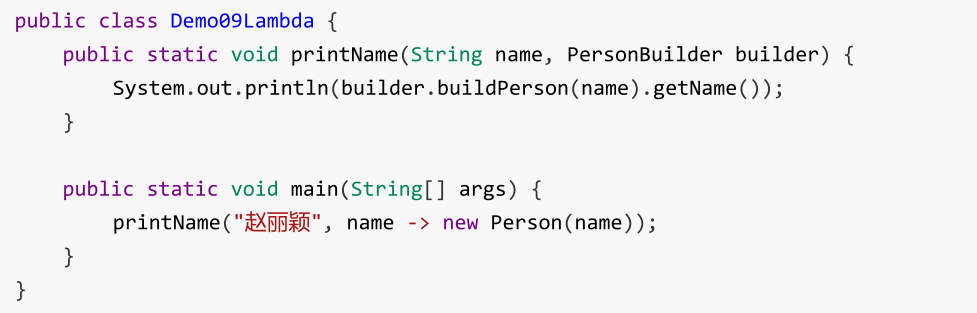
由于构造器的名称与类名完全一样，并不固定。所以构造器引用使用 类名称::new 的格式表示。首先是一个简单的 Person 类：



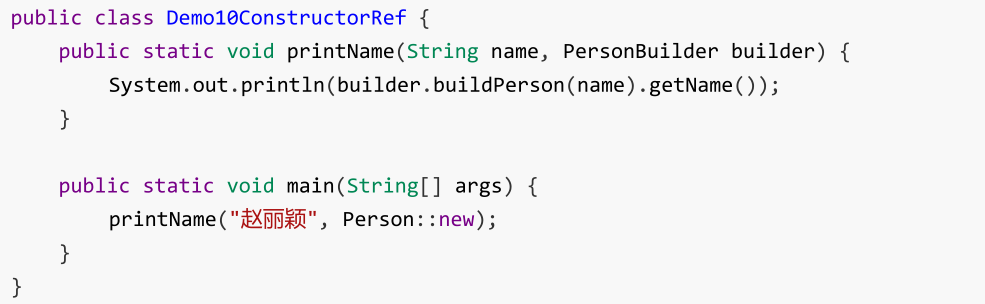
然后是用来创建 Person 对象的函数式接口：



要使用这个函数式接口，可以通过Lambda表达式：



但是通过构造器引用，有更好的写法：



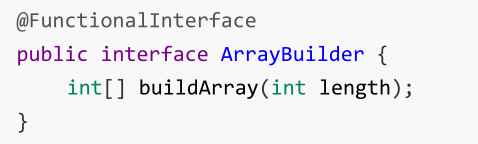
在这个例子中，下面两种写法是等效的：

Lambda表达式： name -> new Person(name)

方法引用： Person::new

## 2.10 数组的构造器引用

数组也是 Object 的子类对象，所以同样具有构造器，只是语法稍有不同。如果对应到Lambda的使用场景中时，需要一个函数式接口：



在应用该接口的时候，可以通过Lambda表达式：



但是更好的写法是使用数组的构造器引用：



在这个例子中，下面两种写法是等效的：

Lambda表达式： length -> new int[length]

方法引用： int[]::new