# day12【函数式接口、方法引用】

## 主要内容

常用函数式接口方法引用

## 教学目标

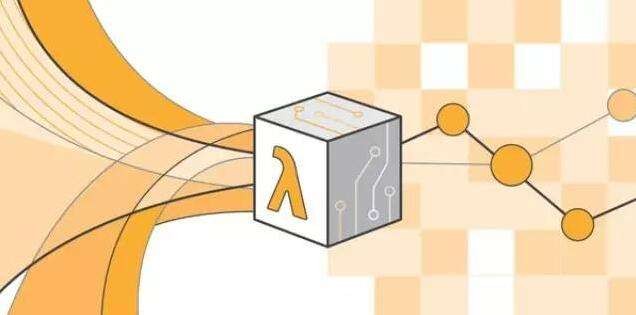
能够使用@FunctionalInterface注解能够自定义无参无返回函数式接口能够自定义有参有返回函数式接口能够理解Lambda延迟执行的特点能够使用Lambda作为方法的参数能够使用Lambda作为方法的返回值能够使用输出语句的方法引用



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能够使用Supplier函数式接口能够使用Consumer函数式接口



# 第一章 函数式接口



## 概念

函数式接口在Java中是指：**有且仅有一个抽象方法的接口**。

函数式接口，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。

备注：“**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。

## 格式

只要确保接口中有且仅有一个抽象方法即可：

修饰符 interface 接口名称 {

public abstract 返回值类型 方法名称(可选参数信息);

// 其他非抽象方法内容

}

由于接口当中抽象方法的 是可以省略的，所以定义一个函数式接口很简单：

public abstract

public interface MyFunctionalInterface { void myMethod();

}

## @FunctionalInterface注解



与 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： 。该注

@Override

@FunctionalInterface

解可用于一个接口的定义上：

@FunctionalInterface

public interface MyFunctionalInterface { void myMethod();

}

一旦使用该注解来定义接口，编译器将会强制检查该接口是否确实有且仅有一个抽象方法，否则将会报错。需要**注 意**的是，即使不使用该注解，只要满足函数式接口的定义，这仍然是一个函数式接口，使用起来都一样。

## 自定义函数式接口

对于刚刚定义好的 函数式接口，典型使用场景就是作为方法的参数：

MyFunctionalInterface

public class Demo09FunctionalInterface {

// 使用自定义的函数式接口作为方法参数

private static void doSomething(MyFunctionalInterface inter) { inter.myMethod(); // 调用自定义的函数式接口方法

}

public static void main(String[] args) {

// 调用使用函数式接口的方法

doSomething(() ‐> System.out.println("Lambda执行啦！"));

}

}

## 练习：自定义函数式接口（无参无返回）

### 题目

eat

请定义一个函数式接口通过Lambda来使用它。

Eatable

### 解答

函数式接口的定义：

，内含抽象

方法，没有参数或返回值。使用该接口作为方法的参数，并进而

@FunctionalInterface  
public interface Eatable {  
 void eat();  
}

应用场景代码：

public class EatableClass {  
 private static void doeat(Eatable eatable){  
 eatable.eat();  
 }  
 public static void main(String[] args) {  
 *doeat*(() -> System.*out*.println("lambda执行了"));  
 }  
}

## 练习：自定义函数式接口（有参有返回）

### 题目

请定义一个函数式接口

Sumable

sum

，内含抽象

方法，可以将两个int数字相加返回int结果。使用该接口作为方法

的参数，并进而通过Lambda来使用它。

### 解答

函数式接口的定义：

@FunctionalInterface  
public interface Sumable {  
 int sum(int i,int j);  
}

应用场景代码：

public class SumableClass {  
  
 private static void sumCla(int i,int j,Sumable sumable){  
 System.*out*.println(sumable.sum(i,j ));  
 }  
 public static void main(String[] args) {  
 *sumCla*(1,1 ,(x,y)->x+y );  
  
 }  
}

# 第二章 函数式编程



在兼顾面向对象特性的基础上，Java语言通过Lambda表达式与方法引用等，为开发者打开了函数式编程的大门。 下面我们做一个初探。

## Lambda的延迟执行



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

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 Demo01Logger {

private static void log(int level, String msg) { if (level == 1) {

System.out.println(msg);

}

}

public static void main(String[] args) { String msgA = "Hello";

String msgB = "World"; String msgC = "Java";

log(1, msgA + msgB + msgC);

}

}

这段代码存在问题：无论级别是否满足要求，作为 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

log

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 buildMessage();

}

然后对 方法进行改造：

log

public class Demo02LoggerLambda {

private static void log(int level, MessageBuilder builder) { if (level == 1) {

System.out.println(builder.buildMessage());

}

}

public static void main(String[] args) { String msgA = "Hello";

String msgB = "World"; String msgC = "Java";

log(1, () ‐> msgA + msgB + msgC );

}

}

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 Demo03LoggerDelay {

private static void log(int level, MessageBuilder builder) { if (level == 1) {

System.out.println(builder.buildMessage());

}

}

public static void main(String[] args) { String msgA = "Hello";

String msgB = "World"; String msgC = "Java";

log(2, () ‐> { System.out.println("Lambda执行！"); return msgA + msgB + msgC;

});

}

}

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。

## 使用Lambda作为参数和返回值

如果抛开实现原理不说，Java中的Lambda表达式可以被当作是匿名内部类的替代品。如果方法的参数是一个函数 式接口类型，那么就可以使用Lambda表达式进行替代。使用Lambda表达式作为方法参数，其实就是使用函数式 接口作为方法参数。

例如 接口就是一个函数式接口，假设有一个 startThread 方法使用该接口作为参数，那么就

java.lang.Runnable

Thread

可以使用Lambda进行传参。这种情况其实和 类的构造方法参数为 Runnable 没有本质区别。

public class Demo04Runnable {

private static void startThread(Runnable task) { new Thread(task).start();

}

public static void main(String[] args) {

startThread(() ‐> System.out.println("线程任务执行！"));

}

}

类似地，如果一个方法的返回值类型是一个函数式接口，那么就可以直接返回一个Lambda表达式。当需要通过一

java.util.Comparator

个方法来获取一个 接口类型的对象作为排序器时：

import java.util.Arrays; import java.util.Comparator;

public class Demo06Comparator {

private static Comparator<String> newComparator() { return (a, b) ‐> b.length() ‐ a.length();

}

public static void main(String[] args) { String[] array = { "abc", "ab", "abcd" }; System.out.println(Arrays.toString(array)); Arrays.sort(array, newComparator()); System.out.println(Arrays.toString(array));

}

}

其中直接return一个Lambda表达式即可。

## 练习：自定义Lambda参数和返回值

### 题目

get

Object

请自定义一个函数式接口

MySupplier

接口分别作为方法的参数和返回值。

### 解答

，含有无参数的抽象方法 得到

类型的返回值。并使用该函数式

01使用自定义Lambda作为方法的参数

函数式接口

@FunctionalInterface  
public interface MySupplier {  
 Object get();  
}

使用该接口作为方法的参数，并且在传递参数时将实际参数写成Lambda：

public class Demo2\_3 {  
 private static void canshu(MySupplier mySupplier){  
 System.*out*.println(mySupplier.get());  
 }  
  
 public static void main(String[] args) {  
 *canshu*(()-> "hello");  
 }  
}

使用该接口作为方法的返回值;

public static void main(String[] args) {  
// canshu(()-> "hello");  
 System.*out*.println(*fanhui*().get());  
 }  
  
 private static MySupplier fanhui(){  
 return ()->"hello";  
 }

# 第三章 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？

## 冗余的Lambda场景

来看一个简单的函数式接口以应用Lambda表达式：

@FunctionalInterface

public interface Printable { void print(String str);

}

在 接口当中唯一的抽象方法 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

Printable

print

来使用它的代码很简单：

public class Demo01PrintSimple {

private static void printString(Printable data) { data.print("Hello, World!");

}

public static void main(String[] args) { printString(s ‐> System.out.println(s));

}

}

其中 方法只管调用

printString

Printable

print

main

接口的

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 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**。

## 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

println(String)

对象中的用呢？

println(String)

方法。既然Lambda希望做的事情就是调用

方法，那何必自己手动调

## 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：

public class Demo02PrintRef {

private static void printString(Printable data) { data.print("Hello, World!");

}

public static void main(String[] args) { printString(System.out::println);

}

}

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。



::

## 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 方法恰好就是我们所需要的。那么对于

println(String)

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 中的 方法来取代Lambda。两种写法的执行效果完全一

System.out

println

样，而第二种方法引用的写法复用了已有方案，更加简洁。

### 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 方法的不同重载形式，将函数式接口改为int类型的参数：

println

@FunctionalInterface

public interface PrintableInteger { void print(int str);

}

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：

public class Demo03PrintOverload {

private static void printInteger(PrintableInteger data) { data.print(1024);

}

public static void main(String[] args) { printInteger(System.out::println);

}

}

这次方法引用将会自动匹配到 的重载形式。

println(int)

## 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：

public class MethodRefObject {

public void printUpperCase(String str) { System.out.println(str.toUpperCase());

}

}

函数式接口仍然定义为：

@FunctionalInterface

public interface Printable { void print(String str);

}

那么当需要使用这个

printUpperCase

Printable

MethodRefObject

成员方法来替代

接口的Lambda的时候，已经具有了

类的对象实例，则可以通过对象名引用成员方法，代码为：

public class Demo04MethodRef {

private static void printString(Printable lambda) { lambda.print("Hello");

}

public static void main(String[] args) { MethodRefObject obj = new MethodRefObject(); printString(obj::printUpperCase);

}

}

## 练习：对象名引用成员方法

### 题目

假设有一个助理类

Assistant

，其中含有成员方法

如下：

public class Assistant {

public void dealFile(String file) { System.out.println("帮忙处理文件：" + file);

}

}

请自定义一个函数式接口

WorkHelper

，其中的抽象方法

的预期行为与

方法一致，并定义一个方法

使用该函数式接口作为参数。通过方法引用的形式，将助理对象中的

help

dealFile

help

### 解答

函数式接口可以定义为：

方法作为Lambda的实现。

@FunctionalInterface  
public interface WorkHelper {  
 void help(String file);  
}

通过对象名引用成员方法的使用场景代码为：

public class Demo3\_6 {  
 private static void helpString(WorkHelper helper){  
 helper.help("workHelp");  
 }  
  
 public static void main(String[] args) {  
 Assistant a = new Assistant();  
 *helpString*(a::deelFile);  
 }  
}

## 通过类名称引用静态方法

由于在

java.lang.Math

abs

类中已经存在了静态方法

，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：

dealFile

@FunctionalInterface public interface Calcable {

int calc(int num);

}

第一种写法是使用Lambda表达式：

public class Demo05Lambda {

private static void method(int num, Calcable lambda) { System.out.println(lambda.calc(num));

}

public static void main(String[] args) { method(‐10, n ‐> Math.abs(n));

}

}

但是使用方法引用的更好写法是：

public class Demo06MethodRef {

private static void method(int num, Calcable lambda) { System.out.println(lambda.calc(num));

}

public static void main(String[] args) { method(‐10, Math::abs);

}

}

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

## 练习：类名称引用静态方法

### 题目

isBlank

假设有一个

StringUtils

字符串工具类，其中含有静态方法

如下：

public final class StringUtils {

public static boolean isBlank(String str) { return str == null || "".equals(str.trim());

}

}

请自定义一个函数式接口 ，其中的抽象方法 checkBlank 的预期行为与 isBlank 一致，并定义一个

StringChecker

方法使用该函数式接口作为参数。通过方法引用的形式，将 StringUtils 工具类中的 isBlank 方法作为Lambda的实现。

### 解答

函数式接口的定义可以为：

@FunctionalInterface  
public interface StringChecker {  
 boolean chekBlank(String str);  
}

使用方法引用:

public class Demo3\_8 {  
  
 private static void isBlank2(String str, StringChecker checker){  
 System.*out*.println(checker.chekBlank(str));  
 }  
  
 public static void main(String[] args) {  
 *isBlank2*("1",StringUtiles::*isBlank* );  
 }  
}

## 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：

@FunctionalInterface

public interface Greetable { void greet();

}

然后是父类 的内容：

Human

public class Human {

public void sayHello() { System.out.println("Hello!");

}

}

最后是子类 的内容，其中使用了Lambda的写法：

Man

public class Man extends Human { @Override

public void sayHello() {

method(() ‐> super.sayHello());

}

private void method(Greetable lambda) { lambda.greet(); System.out.println("I'm a man!");

}

}

但是如果使用方法引用来调用父类中的 方法会更好，例如另一个子类 ：

sayHello

Woman

public class Woman extends Human { @Override

public void sayHello() { method(super::sayHello);

}

private void method(Greetable lambda) { lambda.greet(); System.out.println("I'm a 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

## 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：

@FunctionalInterface public interface Richable {

void buy();

}

下面是一个丈夫 类：

Husband

public class Husband {

private void marry(Richable lambda) { lambda.buy();

}

public void beHappy() {

marry(() ‐> System.out.println("买套房子"));

}

}

开心方法

beHappy

marry

调用了结婚方法

，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：

public class Husband { private void buyHouse() {

System.out.println("买套房子");

}

private void marry(Richable lambda) { lambda.buy();

}

public void beHappy() {

marry(() ‐> this.buyHouse());

}

}

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：

public class Husband { private void buyHouse() {

System.out.println("买套房子");

}

private void marry(Richable lambda) { lambda.buy();

}

public void beHappy() { marry(this::buyHouse);

}

}

在这个例子中，下面两种写法是等效的：

Lambda表达式： () -> this.buyHouse()

方法引用： this::buyHouse

## 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 的格式表示。首先是一个简单

类名称::new

Person

的 类：

public class Person { private String name;

public Person(String name) { this.name = name;

}

public String getName() { return name;

}

}

然后是用来创建 对象的函数式接口：

Person

public interface PersonBuilder { Person buildPerson(String name);

}

要使用这个函数式接口，可以通过Lambda表达式：

public class Demo09Lambda {

public static void printName(String name, PersonBuilder builder) { System.out.println(builder.buildPerson(name).getName());

}

public static void main(String[] args) { printName("赵丽颖", name ‐> new Person(name));

}

}

但是通过构造器引用，有更好的写法：

public class Demo10ConstructorRef {

public static void printName(String name, PersonBuilder builder) { System.out.println(builder.buildPerson(name).getName());

}

public static void main(String[] args) { printName("赵丽颖", Person::new);

}

}

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

## 数组的构造器引用

数组也是 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，

Object

需要一个函数式接口：

@FunctionalInterface

public interface ArrayBuilder { int[] buildArray(int length);

}

在应用该接口的时候，可以通过Lambda表达式：

public class Demo11ArrayInitRef {

private static int[] initArray(int length, ArrayBuilder builder) { return builder.buildArray(length);

}

public static void main(String[] args) {

int[] array = initArray(10, length ‐> new int[length]);

}

}

但是更好的写法是使用数组的构造器引用：

public class Demo12ArrayInitRef {

private static int[] initArray(int length, ArrayBuilder builder) { return builder.buildArray(length);

}

public static void main(String[] args) { int[] array = initArray(10, int[]::new);

}

}

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

备注：数组的构造器引用，可以和Java 8的Stream API结合，在一定程度上“解决”集合中型擦除问题。

toArray

# 第四章 常用函数式接口

方法的泛



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。

前文的 接口就是在模拟一个函数式接口： java.util.function.Supplier<T> 。其实还有很多，下面

MySupplier

是最简单的几个接口及使用示例。

## Supplier接口

接口仅包含一个无参的方法： 。用来获取一个泛型参数指定类型的对

java.util.function.Supplier<T>

T get()

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。

import java.util.function.Supplier;

public class Demo08Supplier {

private static String getString(Supplier<String> function) { return function.get();

}

public static void main(String[] args) { String msgA = "Hello";

String msgB = "World"; System.out.println(getString(() ‐> msgA + msgB));

}

}

备注：其实这个接口在前面的练习中已经模拟过了。

## 练习：求数组元素最大值

### 题目

使用 Supplier 接口作为方法参数类型，通过Lambda表达式求出int数组中的最大值。提示：接口的泛型请使用

java.lang.Integer 类。

### 解答

package lambdaTest;  
  
import java.util.function.Supplier;  
  
public class Demo4\_2 {  
 private static int getMax(Supplier<Integer> supplier){  
 return supplier.get();  
 }  
  
 public static void main(String[] args) {  
 int[]array={1,3,2,0};  
  
 System.*out*.println(*getMax*(()-> {  
 int max = array[0];  
 for (int i = 1; i < array.length; i++) {  
 if (array[i] > max) {  
 max = array[i];  
 }  
 }  
 return max;  
 }  
 ));  
 }  
}

* 1. **Consumer接口**

型参数决定。

### 抽象方法：accept

接口中包含抽象方法

Consumer

接口则正好相反，它不是生产一个数据，而是**消费**一个数据，其数据类型由泛

，意为消费一个指定泛型的数据。基本使用如：

java.util.function.Consumer<T>

void accept(T t)

import java.util.function.Consumer;

public class Demo09Consumer {

private static void consumeString(Consumer<String> function) { function.accept("Hello");

}

public static void main(String[] args) { consumeString(s ‐> System.out.println(s)); consumeString(System.out::println);

}

}

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：消费一个数据的时候，首先做一个操

andThen

作，然后再做一个操作，实现组合。而这个方法就是码：

Consumer

接口中的default方法

。下面是JDK的源代

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 静态方法将会在参数为null时主动抛出

requireNonNull

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，而和的情况：

andThen

的语义正是“一步接一步”操作。例如两个步骤组

import java.util.function.Consumer;

public class Demo10ConsumerAndThen {

private static void consumeString(Consumer<String> one, Consumer<String> two) { one.andThen(two).accept("Hello");

}

public static void main(String[] args) { consumeString(

s ‐> System.out.println(s.toUpperCase()), s ‐> System.out.println(s.toLowerCase()));

}

}

运行结果将会首先打印完全大写的HELLO，然后打印完全小写的hello。当然，通过链式写法可以实现更多步骤的 组合。

## 练习：格式化打印信息

### 题目

下面的字符串数组当中存有多条信息，请按照格式“ ”的格式将信息打印出来。要求将打印姓

姓名：XX。性别：XX。

Consumer

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例，将两个 Consumer 接口按照顺序“拼接”到一起。

接口的Lambda实

public static void main(String[] args) {

String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男" };

}

### 解答

package lambdaTest;  
  
import java.util.function.Consumer;  
  
public class Demo4\_4 {  
 private static void consumerString(String s,Consumer<String> one,Consumer<String> two){  
 one.andThen(two).accept(s);  
 }  
  
 public static void main(String[] args) {  
 String[] array={ "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男" };  
 for (String s : array) {  
 String[] split = s.split(",");  
 *consumerString*(s, s2->System.*out*.println(split[0]),  
 s3->System.*out*.println(split[1]) );  
 }  
  
 }  
}