

**【函数式接口】**

**内容**

自定义函数式接口

函数式编程

常用函数式接口

**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Supplier函数式接口

能够使用Consumer函数式接口

能够使用Function 函数式接口

能够使用Predicate 函数式接口

**第一章 函数式接口**

**1.1 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。

**1.2 格式**

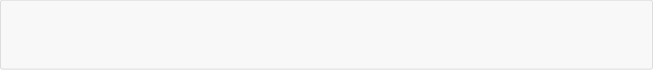
只要确保接口中有且仅有一个抽象方法即可：

 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);

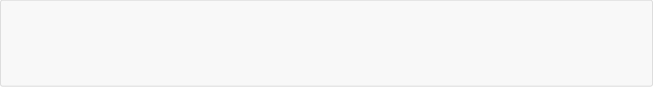
    // 其他非抽象方法内容

}



 public interface MyFunctionalInterface  {        void myMethod();

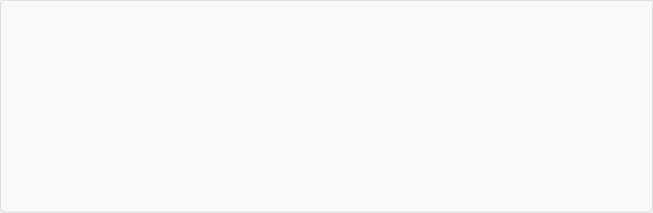
}



 @FunctionalInterface

public interface  MyFunctionalInterface {     void myMethod();

}



 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！"));     }

}

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：

**1.3 @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：

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。

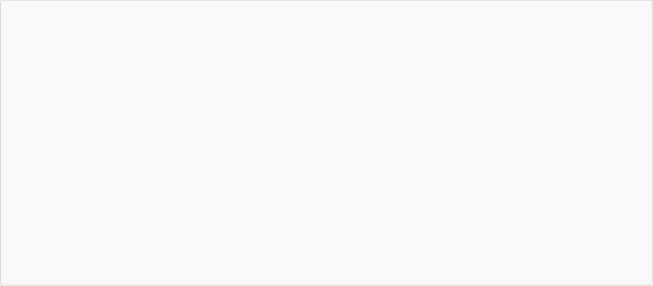
**1.4 自定义函数式接口**

对于刚刚定义好的 MyFunctionalInterface 函数式接口，典型使用场景就是作为方法的参数：

**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。

**2.1 Lambda的延迟执行**



 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";

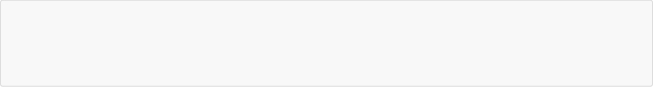
        String msgB = "World";

        String msgC = "Java";

        log(1, msgA + msgB + msgC);

    }

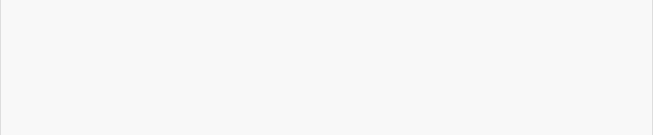
}



 @FunctionalInterface

public interface  MessageBuilder {      String buildMessage();

}



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：

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：

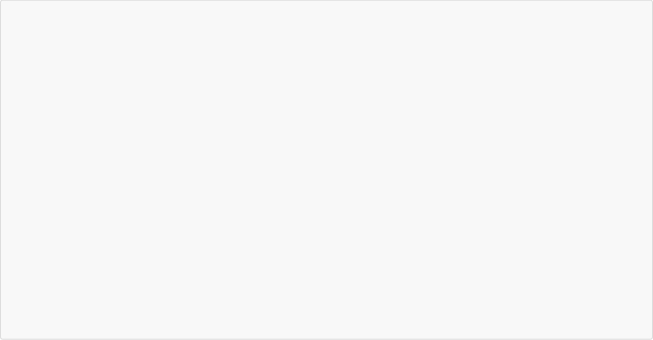
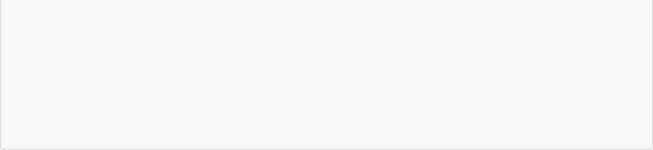
然后对 log 方法进行改造：

 public class Demo02LoggerLambda {

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

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

    }



 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;

        });

    }

}



例如 java.lang.Runnable 接口就是一个函数式接口，假设有一个 使用该接口作为参数，那么就 可以使用Lambda进行传参。这种情况其实和 Thread 类的构造方法 没有本质区别。

    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：

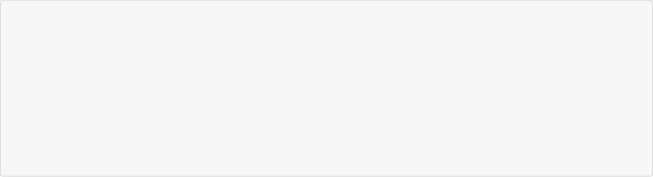
从结果中可以看出，在不符合级别要求的情况下，Lambda将不会执行。从而达到节省性能的效果。

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**

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。

|  |  |  |
| --- | --- | --- |
| startThread | | 方法 |
| 参数为 | Runnable | |



 public class Demo04Runnable  {

    private static void startThread(Runnable task) {

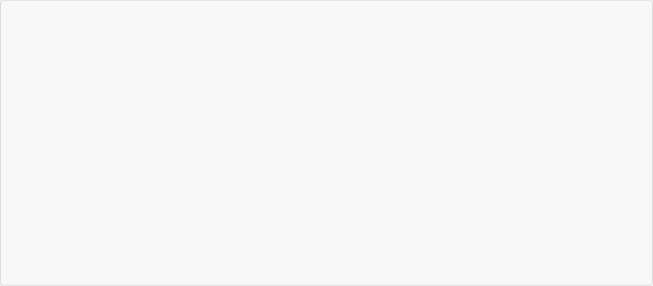
        new Thread(task).start();

    }

    public static void main(String[] args) {

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

}



 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));     }

}



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 个方法来获取一个 java.util.Comparator 接口类型的对象作为排序器时,就可以调该方法获取。

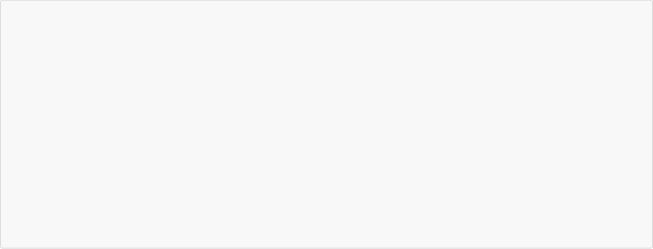
其中直接return一个Lambda表达式即可。

**第三章 常用函数式接口**

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。 下面是最简单的几个接口及使用示例。

**3.1 Supplier接口**

java.util.function.Supplier<T> 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。



 import java.util.function .Supplier;

public class Demo08Supplier  {

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

    }

    public static void main(String[] args) {

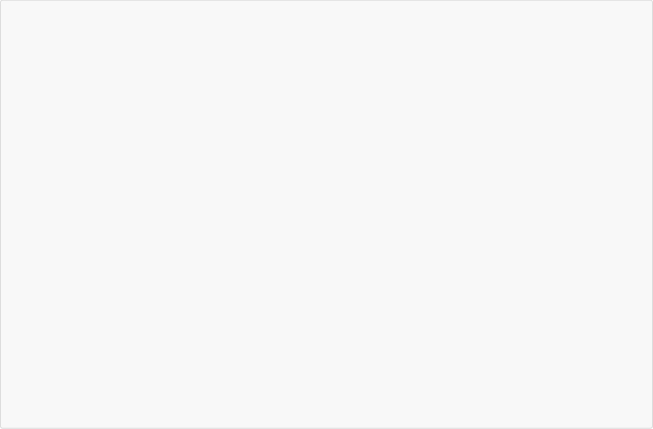
        String msgA = "Hello";

        String msgB = "World";

        System.out.println (getString (() ‐> msgA + msgB));

    }

}



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 类。

**3.2 练习：求数组元素最大值**

**题目**

**解答**

 public class Demo02Test {

    //定一个方法,方法的参数传递Supplier,泛型使用Integer

    public static int getMax(Supplier <Integer> sup){

        return sup.get();

    }

    public static void main(String[] args) {

        int arr[] = {2,3,4,52,333,23};

        //调用getMax方法,参数传递Lambda

        int maxNum = getMax(()‐>{

           //计算数组的最大值

           int max = arr[0];

           for(int i :arr){

               if(i>max){

                   max = i;

               }

           }

           return max;

        });

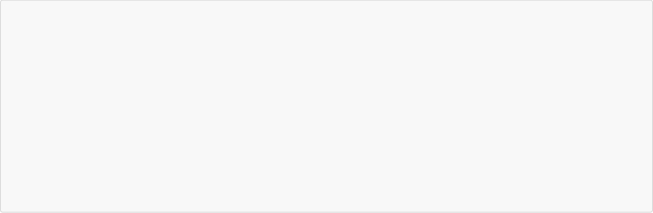
        System.out.println (maxNum);

    }

}

**3.3 Consumer接口**

|  |  |  |
| --- | --- | --- |
| 使用 | Supplier | 接口 |
| java.lang.Integer | | |



 import java.util.function .Consumer;

public class Demo09Consumer  {

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

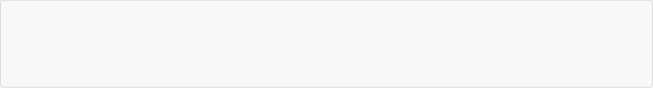
    }

    public static void main(String[] args) {

        consumeString(s ‐> System.out.println(s));

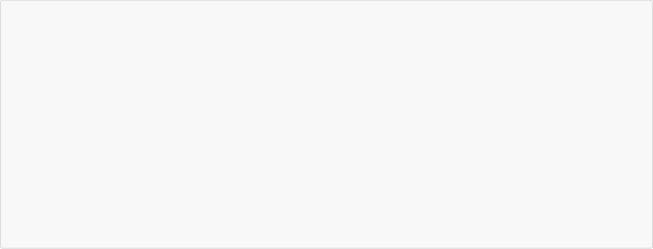
    }

}



 default  Consumer<T> andThen(Consumer<? super T> after) {     Objects.requireNonNull(after);

    return (T t) ‐> { accept(t); after.accept(t); }; }



 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()));

    }

}

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

java.util.function.Consumer<T> 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：

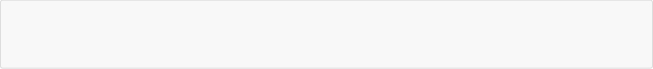
当然，更好的写法是使用方法引用。

**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：

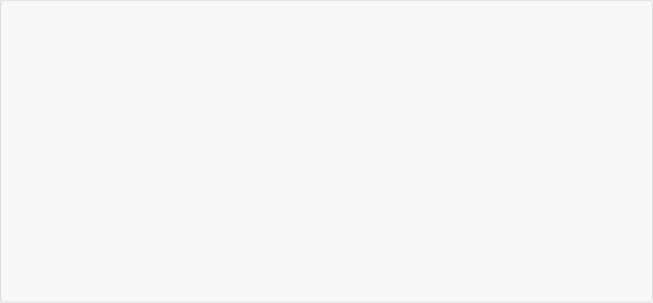
|  |  |  |
| --- | --- | --- |
| 备注： | java.util.Objects | |
| NullPointerException | | 异 |

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：



 public static void main(String[] args) {

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



 import java.util.function .Consumer;

public class DemoConsumer {

    public static void main(String[] args) {

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

        printInfo(s ‐> System.out.print("姓名：" + s.split(",")[0]),

                  s ‐> System.out.println("。性别：" + s.split(",")[1] + "。"),                   array);

    }

    private static void printInfo(Consumer<String> one, Consumer<String> two, String[] array) {         for (String info : array) {

            one.andThen(two).accept(info); // 姓名：迪丽热巴。性别：女。

        }

    }

}



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 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 顺序“拼接”到一起。

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。

**3.4 练习：格式化打印信息**

**题目**

|  |  |  |
| --- | --- | --- |
| 为第一个 | Consumer | |
| Consumer | | 接口按照 |

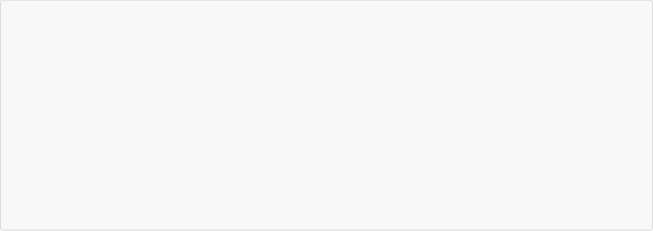
**解答**

**3.5 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 java.util.function.Predicate<T> 接口。

**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：



 import java.util.function .Predicate;

public class Demo15PredicateTest {

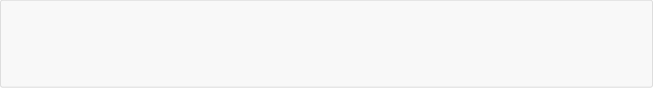
    private static void method(Predicate <String> predicate) {         boolean veryLong = predicate.test("HelloWorld");         System.out.println ("字符串很长吗：" + veryLong);     }

    public static void main(String[] args) {

        method(s ‐> s.length() > 5);

    }

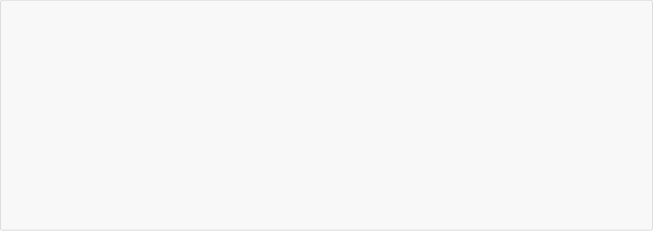
}



 default  Predicate<T> and(Predicate<? super T> other) {     Objects.requireNonNull(other);

    return (t) ‐> test(t) && other.test(t);

}



 import java.util.function .Predicate;

public class Demo16PredicateAnd {

    private static void method(Predicate <String> one, Predicate<String> two) {         boolean isValid  = one.and(two).test("Helloworld");

        System.out.println ("字符串符合要求吗：" + isValid);

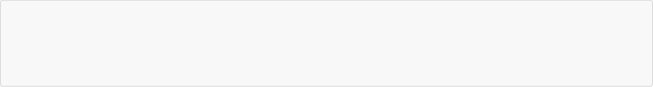
    }

    public static void main(String[] args) {

        method(s ‐> s.contains ("H"), s ‐> s.contains ("W"));

    }

}



 default  Predicate<T> or(Predicate<? super T> other) {     Objects.requireNonNull(other);

    return (t) ‐> test(t) || other.test(t);

}

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5则认为很长。

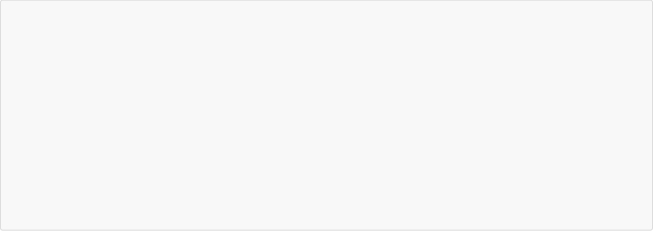
**默认方法：and**

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：

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：

**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：



 import java.util.function .Predicate;

public class Demo16PredicateAnd {

    private static void method(Predicate <String> one, Predicate<String> two) {         boolean isValid  = one.or(two).test("Helloworld" );

        System.out.println ("字符串符合要求吗：" + isValid);

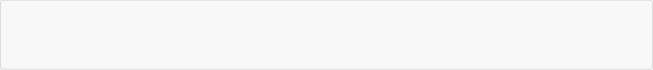
    }

    public static void main(String[] args) {

        method(s ‐> s.contains ("H"), s ‐> s.contains ("W"));

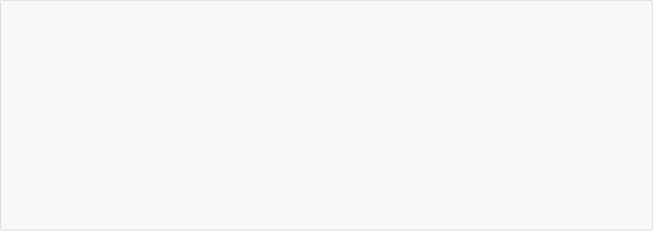
    }

}



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

}



 import java.util.function .Predicate;

public class Demo17PredicateNegate  {

    private static void method(Predicate <String> predicate) {         boolean veryLong = predicate.negate().test("HelloWorld");         System.out.println ("字符串很长吗：" + veryLong);     }

    public static void main(String[] args) {

        method(s ‐> s.length() < 5);

    }

}



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：

**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：

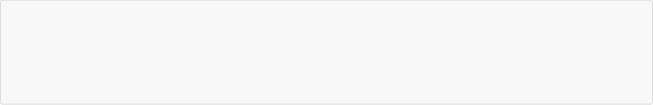
从实现中很容易看出，它是执行了test方法之后，对结果boolean 值进行“!”取反而已。一定要在 test 方法调用之前 调用 negate 方法，正如 and 和 or 方法一样：

**3.6 练习：集合信息筛选**

**题目**

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。

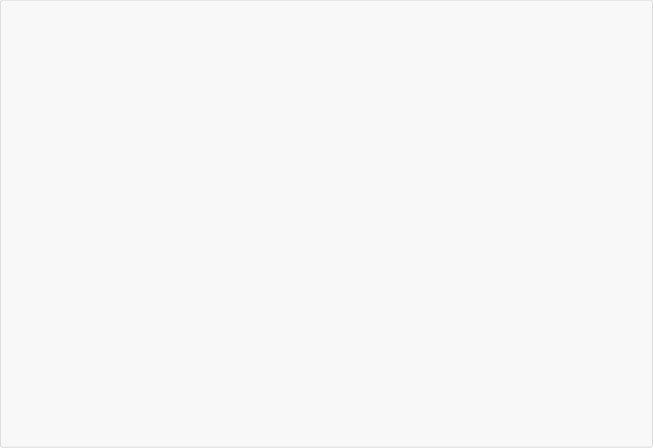


 public class DemoPredicate {

    public static void main(String[] args) {

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

}



**解答**

 import java.util.ArrayList ;

import java.util.List;

import java.util.function .Predicate;

public class DemoPredicate {

    public static void main(String[] args) {

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

        List<String> list = filter(array,

                                   s ‐> "女".equals(s.split(",")[1]),

                                   s ‐> s.split(",")[0].length() == 4);

        System.out.println (list);

    }

    private static List<String> filter(String[] array, Predicate<String> one,                                        Predicate <String> two) {

        List<String> list = new ArrayList<>();

        for (String info : array) {

            if (one.and(two).test(info)) {

                list.add(info);

            }

        }

        return list;

    }

}

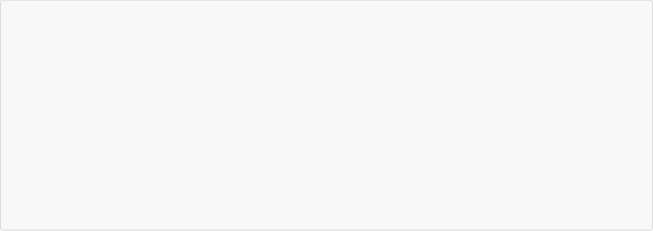
**3.7 Function接口**

java.util.function.Function<T,R> 接口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。



 import java.util.function .Function;

public class Demo11FunctionApply {

    private static void method(Function <String, Integer> function) {         int num = function.apply("10");

        System.out.println (num + 20);

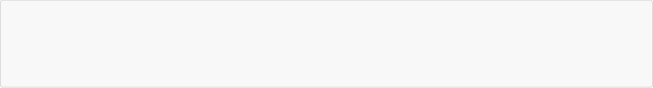
    }

    public static void main(String[] args) {

        method(s ‐> Integer.parseInt(s));

    }

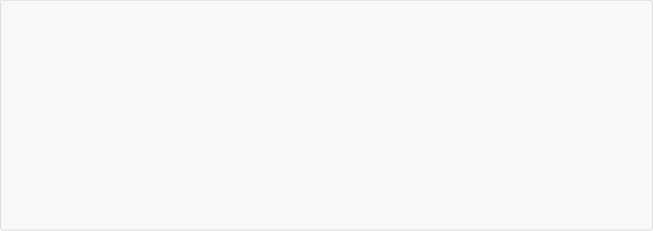
}



 default  <V> Function<T, V> andThen(Function<? super R,? extends  V> after) {     Objects.requireNonNull(after);

    return (T t) ‐> after.apply(apply(t));

}



 import java.util.function .Function;

public class Demo12FunctionAndThen  {

    private static void method(Function <String, Integer> one, Function <Integer, Integer > two) {         int num = one.andThen(two).apply("10");

        System.out.println (num + 20);

    }

    public static void main(String[] args) {

        method(str‐>Integer.parseInt(str)+10, i ‐> i \*= 10);

    }

}

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：

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：

第一个操作是将字符串解析成为int数字，第二个操作是乘以10。两个操作通过 andThen 按照前后顺序组合到了一 起。

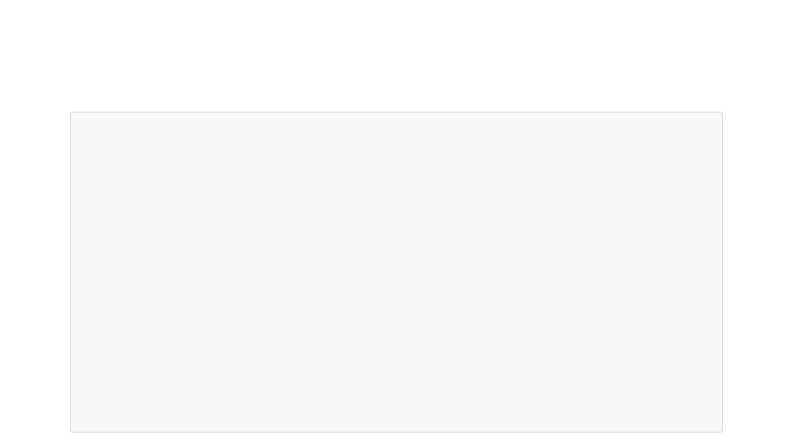
请注意，Function的前置条件泛型和后置条件泛型可以相同。

**3.8 练习：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";



 import java.util.function .Function;

public class DemoFunction {

    public static void main(String[] args) {

        String str = "赵丽颖,20";

        int age = getAgeNum(str, s ‐> s.split(",")[1],                             s ‐>Integer.parseInt (s),                             n ‐> n += 100);

        System.out.println (age);

    }

    private static int getAgeNum(String str, Function<String, String> one,                                  Function<String, Integer> two,                                  Function<Integer , Integer> three) {         return one.andThen (two).andThen(three).apply(str);

    }

}

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。 **解答**