**函数式编程**

# 知识回顾

## IDEA的项目结构

|  |  |
| --- | --- |
| 目录结构 | 说明 |
| out | 字节码文件，编译以后的class文件 |
| src | 我们写的源代码 |

## IDEA常用快捷键

|  |  |
| --- | --- |
| 快捷键 | 功能 |
| Ctrl+Y | 删除一行 |
| Ctrl+D | 复制一行 |
| Ctrl+Alt+L | 格式化代码 |
| Alt+Enter | 代码建议，修复代码 |
| Ctrl+/ | 单行注释 |
| Ctrl+Shift+/ | 多行注释 |
| Ctrl+Space | 自动完成 |
| Alt+Ins | 产生getter/setter方法 |
| Ctrl + J | 使用代码模块 |

## java.lang.Comparable接口

|  |  |
| --- | --- |
| Comparable接口中的方法 | 说明 |
| int compareTo(T o) | 比较当前对象和传入的o对象的大小  正整数：表示当前元素大于另一个元素  0：相等  负整数：当前元素小于另一元素 |

## java.util.Comparator接口

|  |  |
| --- | --- |
| Comparator接口中的方法 | 说明 |
| int compare(T o1, T o2) | 比较o1和o2大小 正整数：表示o1大于o2  0：相等  负整数：o1小于o2 |

## Lambda格式

|  |
| --- |
| Lambda表达式的标准格式为 |
| (参数列表)->{方法体} |

|  |
| --- |
| Lambda表达式的三个组成部分： |
| () 方法参数 |
| -> 固定写法 |
| {} 方法体 |

## Lambda的使用前提

接口中只有一个抽象方法

## Lambda省略格式

1. 参数的类型可以省略
2. 如果方法体只有一条语句，可以省略return，{}，分号
3. 如果参数只有一个，可以省略小括号

## 遍历的语法

|  |  |
| --- | --- |
| java.lang.Iterable接口中的方法 | 说明 |
| forEach(Consumer c) | 遍历集合 |

|  |  |
| --- | --- |
| java.util.Map接口中的方法 | 说明 |
| forEach(BiCosumer b) | 遍历集合 |

* + 教学目标

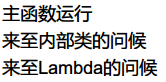
1. 能够理解Lambda延迟执行的特点
2. 能够使用Lambda作为方法的参数
3. 能够使用Lambda作为方法的返回值
4. 能够定义函数式接口
5. 能够使用@FunctionalInterface注解
6. 能够使用Supplier函数式接口
7. 能够使用Consumer函数式接口
8. 能够使用Function函数式接口
9. 能够使用Predicate函数式接口

# Lambda的延迟执行

使用Lambda表达式的主要原因是：将代码的执行延迟到一个合适的时间点。所有的Lambda表达式都是延迟执行的。因为匿名内部类的方法都是要等到调用的时候才会执行。

## 延迟执行的基本案例：

* 案例效果：



* 案例步骤：

1. 创建一个Person接口，有一个sayHi()的无参无返回值的方法。
2. 在main函数中直接实例化Person为p1对象，并且实现sayHi()方法，此时方法并没有执行，因为方法不调用不执行。
3. 在后面输出一句话：主函数执行
4. 再调用sayHi()方法，才发现sayHi()方法的输出。
5. 同理使用Lambda也是一样的效果。

* 案例代码：

package com.itheima;

interface Person {

void sayHi();

}

/\*\*

\* Lambda的延迟执行

\*/

public class Demo01Lazy {

public static void main(String[] args) {

Person p1 = new Person() {

@Override

public void sayHi() {

System.out.println("来至内部类的问候");

}

};

System.out.println("主函数的问候");

//调用内部类中的方法

p1.sayHi();

//创建一个lambda

Person p2 = () -> System.out.println("来至Lambda的问候");

p2.sayHi();

}

}

## 具体案例的演示

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，减少无用代码的执行，提升性能。下面案例实现了日志记录的功能。

* 案例效果：



* 案例步骤：

1. 有一个静态方法: void log(int level, String message)，当它为日志级别为1级的时候，打印出日志的信息。
2. 在main函数中创建一个时间格式化的类，格式是：yyyy-MM-dd HH:mm:ss.SSS
3. 在main函数中调用2次log()方法，1级调用1次，2级调用1次。
4. 第1次日志信息内容是："Rose: " + sdf.format(new Date()) + " 进行了转账操作"
5. 第2次日志信息内容是："Jack: " + sdf.format(new Date()) + " 进行了取钱操作"

* 案例代码：

package com.itheima;

import java.text.SimpleDateFormat;

import java.util.Date;

//模拟日志记录功能

public class Demo2Log {

//只输出1级的信息

public static void log(int level, String message) {

if (level == 1) {

System.out.println(message);

}

}

public static void main(String[] args) {

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");

String sdate = dateFormat.format(new Date());

//调用方法

log(1, "Rose：" + sdate + "进行了转账操作");

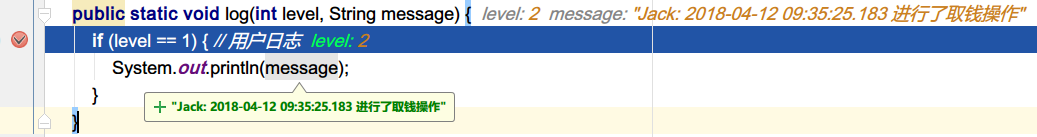
log(2, "Jack：" + sdate + "进行了取钱的操作");

}

}

* 案例分析：

1. 在log方法的内部设置断点，证明两次都会对字符串进行计算。



1. 存在问题：字符串拼接完成之后，在调用log方法后可能不会被使用，那么此时之前的拼接工作就白做了，这样就会浪费性能。

## Lambda的更优写法

* 案例效果



* 案例步骤

1. 创建一个接口BuilderMessage，接口中有一个抽象方法String buildeMessage()，返回字符串。
2. 创建类，创建静态方法用于输出日志信息：  
   static void log(int level, BuilderMessage bulider)  
   在方法内部判断如果是1级信息，打印日志信息，并调用接口中的buildeMessage()方法拼接字符串。
3. 在main函数中创建日期格式化类，调用2次log()方法：1级调用1次，2级调用1次。log方法的第2个参数使用Lambda表达式，方法体中使用return返回拼接的字符串。
4. 第1次信息内容是："Rose: " + sdf.format(new Date()) + " 进行了转账操作"
5. 第2次信息内容是："Jack: " + sdf.format(new Date()) + " 进行了取钱操作"

## 证明Lambda的延迟执行

* 案例效果：



* 案例分析：

在Lambda表达式方法体中返回字符串之前，先输出当前的级别，发现只输出了级别1，级别2并没有输出。即当级别为2时，没有对字符串进行拼接操作了，这样就避免了拼接字符串带来的额外开销。

* 案例代码：

package com.itheima;

import java.text.SimpleDateFormat;

import java.util.Date;

interface BuilderMessage {

//拼接字符串

String buildMessage();

}

public class Demo3Log {

//在方法内部判断如果是1级信息，打印日志信息，并调用接口中的buildeMessage()方法拼接字符串。

static void log(int level, BuilderMessage builderMessage) {

if (level == 1) {

//通过调用方法得到要打印的字符串

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

}

}

public static void main(String[] args) {

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String sdate = dateFormat.format(new Date());

/\*log(1, new BuilderMessage() {

@Override

public String buildMessage() {

return "Rose: " + sdate + "转账的操作";

}

});\*/

log(1, ()-> {

System.out.println("1级信息：");

return "Rose: " + sdate + "转账的操作";

});

log(2, ()-> {

System.out.println("2级信息：");

return "Jack: " + sdate + "取钱的操作";

});

}

}

## 代码分析：

只有在需要的时候才能运行代码，这是使用lambda表达式的一种情况。

void log(int level, BuilderMessage bulider) 这段代码实际的运行效果分如下步骤：

1. 通过BuilderMessage接口接受lambda表达式。
2. 检查它是否应该被调用，即level是否等于1。
3. 在需要的时候调用它，bulider.buildeMessage()，其实是调用了这个方法才会进行字符串的拼接操作。

# 函数式接口

## 接口默认方法的回顾

### JDK8中为什么有接口默认方法

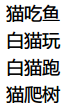
假如很久以前创建了一个接口，并且已经被大量的类实现。如果需要再扩充这个接口的功能加新的方法，就会导致所有已经实现的子类需要重写这个方法。如果在接口中使用默认方法就不会有这个问题。所以从JDK8开始新加了接口默认方法，便于接口的扩展。

### 接口中默认方法的规则

1. 默认方法使用default关键字，一个接口中可以有多个默认方法。
2. 接口中既可以定义抽象方法，又可以定义默认方法，默认方法不是抽象方法。
3. 子类实现接口的时候，可以直接调用接口中的默认方法，即继承了接口中的默认方法。
4. 接口中同时还可以定义静态方法，静态方法通过接口名调用。

### 默认方法的案例

* 案例效果：



* 案例步骤：

1. 创建一个Cat接口，包含play()抽象方法，eat()静态方法，run()默认方法，climb()默认方法
2. 写一个子类WhiteCat白猫实现了猫接口
3. 实现play()抽象方法，重写run()的默认方法
4. 创建类Demo04Interface，在main函数中创建WhiteCat对象
   1. 调用接口中静态方法eat()
   2. 调用白猫子类中实现的方法play()
   3. 调用白猫子类中重写的方法run()
   4. 调用接口中默认方法climb()

* 案例代码：

package com.itheima;

/\*

3.1.3 默认方法的案例

\*/

interface Cat {

//包含play()抽象方法

void play();

// eat()静态方法

static void eat() {

System.out.println("猫吃鱼");

}

// run()默认方法

default void run() {

System.out.println("猫跑");

}

// climb()默认方法

default void climb() {

System.out.println("猫爬树");

}

}

class WhiteCat implements Cat {

@Override

public void play() {

System.out.println("白猫白玩");

}

//重写跑的方法

@Override

public void run() {

System.out.println("白猫跑");

}

}

public class Demo4Interface {

public static void main(String[] args) {

WhiteCat whiteCat = new WhiteCat();

whiteCat.play(); //重写的抽象方法

whiteCat.run(); //重写的默认方法

whiteCat.climb(); //接口中的默认方法

Cat.eat(); //静态方法

}

}

## 函数式接口的概念

在Java中，当接口中只有一个抽象方法时，该接口就称为函数式接口。

函数式接口，即适用于函数式编程场景的接口。而Java中的函数式编程就是Lambda，即函数式接口就是可以适用于Lambda使用的接口。接口中只有一个抽象方法时，Java中的Lambda才能顺利地进行类型的推导。

函数式接口都是先使用Lambda接口实现方法体，再调用接口中的方法。

## 函数式接口的格式

@FunctionalInterface注解：这个注解是JDK1.8新增的特性，用来修饰接口的定义，表明该接口是一个函数式接口。格式如下：

|  |
| --- |
| @FunctionalInterface  interface 接口名 {  接口返回值 方法名(参数列表);  } |

之前学习的Runnable、Comparator都属于函数式接口。

* 提示：在idea中按两次Shift，弹出搜索类的输入框



查看Runable接口的源代码如下：

@FunctionalInterface

public interface Runnable {

/\*\*

\*

\*/

public abstract void run();

}

## 自定义函数式接口

### 无参无返回值

* 案例步骤

1. 定义一个函数式接口Eatable，加上注解：@FunctionalInterface
2. 接口中包含抽象eat()方法，没有参数或返回值。
3. 在main()函数中使用Lambda创建Eatable对象，输出一句话：吃东西。
4. 调用对象的eat()方法

* 案例效果



* 案例代码

package com.itheima;

@FunctionalInterface

interface Eatable {

void eat();

}

public class Demo05Interface {

public static void main(String[] args) {

//内部类

Eatable eatable = new Eatable() {

@Override

public void eat() {

System.out.println("吃饭");

}

};

eatable.eat();

//lambda

Eatable eatable1 = () -> System.out.println("吃菜");

eatable1.eat();

}

}

### 有参有返回值

* 案例步骤

1. 定义一个函数式接口Sumable，加上注解：@FunctionalInterface
2. 接口中包含抽象int sum(int a, int b)方法，将两个int数字相加返回int结果。
3. 在类中通过Lambda实例化Sumable接口，并且调用sum()方法
4. 输出计算结果的值

* 案例效果



* 案例代码

package com.itheima;

@FunctionalInterface

interface Sumable {

int sum(int a, int b);

}

public class Demo06Interface {

public static void main(String[] args) {

//lambda，创建内部类，实现了sum()方法

Sumable sumable = (a, b) -> a + b;

//调用方法

System.out.println(sumable.sum(4, 5));

}

}

## 关于@FunctionalInterface注解的说明：

1. 该注解用在只有一个抽象方法的接口上，接口中可以存在其它的非抽象方法。如果该接口有超过一个抽象方法，则编译失败。
2. 接口中的静态方法和默认方法都不是抽象方法。
3. 接口中定义的与Object类同名的抽象方法不算抽象方法，因为此接口的实现类一定是Object的子类，会自动实现这个接口中定义的抽象方法。
4. 如果接口中只有一个抽象方法，不管是否使用@FunctionalInterface修饰，该接口都是函数式接口。加上注解的目的是让编译器进行语法检查，类似于以前学的@Override注解。
5. 接口加这个注解还可以限制后期对此接口修改时再加上新的抽象方法，从而保证这个函数式接口只有一个抽象方法。

### 面试题

* 下面定义的接口是否是函数式接口，编译能不能通过？为什么？

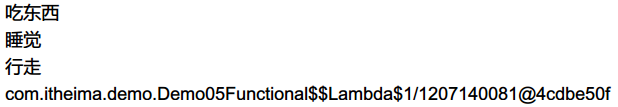
@FunctionalInterface  
**public interface** Animal {  
 **void** eat();  
  
 **public** String toString();  
  
 **default void** sleep() {  
 System.***out***.println(**"睡觉"**);  
 }  
  
 **static void** move() {  
 System.***out***.println(**"行走"**);  
 }  
}

### 测试函数式接口

* 案例需求：

1. 使用Lambda表达式实现上面的eat()方法，输出：吃饭。
2. 调用方法eat()
3. 调用接口中的默认方法sleep();
4. 使用类名调用静态方法move();
5. 输出toString()方法，即输出匿名内部类对象的地址。

* 案例效果：



* 案例代码：

package com.itheima;

@FunctionalInterface

interface Animal {

void eat();

public String toString();

default void sleep() {

System.out.println("睡觉");

}

static void move() {

System.out.println("行走");

}

}

//测试一下函数式接口

public class Demo07Functional {

public static void main(String[] args) {

Animal animal = () -> System.out.println("吃面包");

animal.eat(); //实现 的方法

animal.sleep(); //默认方法

Animal.move(); //静态方法

System.out.println(animal.toString()); //输出的是匿名内部类的地址

}

}

# 常用函数式接口

JDK8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。

## JDK 8中重要的函数接口

|  |  |
| --- | --- |
| 接口 | 含义 |
| Supplier<T> | 提供者，用于提供某个值。只出不进 |
| Consumer<T> | 消费者，使用提供的参数，没有返回值。只进不出 |
| Predicate<T> | 谓语：我(不)是程序猿。一个判断的接口，对提供的参数进行判断，返回boolean值 |
| Function<T,R> | 函数，是数学中的函数，提供一个参数T，返回一个计算后的结果R。 |
| BinaryOperator<T> | 二元运算符，提供2个操作数，得到1个计算结果 |
| UnaryOperator<T> | 一元运算符，只提供1个操作数，得到1个计算结果. |

## Supplier接口

### 概述

Supplier接口代表一个结果的提供者。

Supplier接口是用来生成数据的，数据的类型通过泛型参数给定。使用get()方法获得返回值，不要求每次调用时都返回新的或不同的结果。接口的源码如下：

package java.util.function;

@FunctionalInterface

public interface Supplier<T> {

T get();

}

### 抽象方法get：

|  |  |
| --- | --- |
| Supplier接口中的方法 | 说明 |
| T get() | 获得一个结果，没有参数 |

### 示例：直接使用Supplier对象

* 案例需求

get()方法的基本使用，在Supplier接口中get方法返回一个字符串。

* 案例步骤：

1. 使用Lambda创建Supplier对象，泛型类型为String，方法体返回一个字符串，return可以省略。
2. 调用Supplier的get()方法得到字符串，并打印输出

* 案例效果



* 案例代码

package com.itheima;

import org.junit.Test;

import java.util.function.Supplier;

//get()方法的使用

public class Demo08Supplier {

@Test

public void testGet() {

//创建Supplier对象

Supplier<String> supplier = () -> "Hello World";

//调用方法

System.out.println(supplier.get());

}

}

* 代码分析

1. () -> "Hello World" 相当于实现了匿名的内部类，重写了String get()
2. 方法体中return "Hello World"
3. supplier.get()，调用了重写的方法，并输出。

### 示例：创建员工对象

* 案例需求：

下面的例子演示了如何通过调用一个静态方法，生成一个员工对象返回。使用构造方法做为Supplier参数的引用。

* 案例步骤：

1. 在主类内部创建一个私有的静态Employee对象，重写toString()方法，返回一个字符串："我是员工"。
2. 在main函数中创建一个Supplier对象，泛型类型是Employee。使用Lambda传入Supplier对象，方法体实例化员工对象，省略return方法。
3. 使用supplier对象的get()方法得到员工对象
4. 打印输出员工对象
   * 因为Employee对象是私有的，外部类无法直接实例化员工对象。调用Supplier的get()方法来生成员工对象，这样做的目的是可以控制员工对象的生成方式，类似于工厂模式。

* 案例效果



* 案例代码

package com.itheima;

import java.util.function.Supplier;

public class Demo07Supplier {

//命名内部类

private static class Employee {

@Override

public String toString() {

return "我是员工";

}

}

//只能通过我提供的方法得到员工对象

public static void main(String[] args) {

//可以控制对象的生成

Supplier<Employee> supplier = () -> new Employee();

//得到一个员工

System.out.println(supplier.get());

}

}

### 示例： 将Supplier做为方法的参数

* 需求说明

求数组中的最大值，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。

* 需求分析

1. 定义整型数组int[] arr = {12,68,10,2,99,313,46};
2. 创建静态方法getMax()：返回int类型，将Supplier做为参数，泛型类型为Integer，方法体调用get()方法返回值。
3. 在main函数中调用getMax()方法，使用Lambda传入Supplier对象，并且实现查找最大值的功能。
4. 先使用匿名内部类实现Supplier接口，找出最大值输出。
5. 再使用Lambda表达式实现Supplier接口，找出最大值。

* 案例效果



* 代码实现

package com.itheima;

import java.util.function.Supplier;

public class Demo09SupplierMax {

public static void main(String[] args) {

int[] arr = {12,68,10,2,99,313,46};

//调用的时候实现功能

//内部类写法

int max = getMax(new Supplier<Integer>() {

@Override

public Integer get() {

int m = arr[0];

for (int i = 1; i < arr.length ; i++) {

if (m < arr[i]) m = arr[i];

}

return m;

}

});

System.out.println("最大值：" + max);

//lambda写法

int max2 = getMax(() -> {

int m = arr[0];

for (int i = 1; i < arr.length ; i++) {

if (m < arr[i]) m = arr[i];

}

return m;

});

System.out.println(max2);

}

/\*\*

\* 创建静态方法getMax()：返回int类型，将Supplier做为参数，泛型类型为Integer，方法体调用get()方法返回值。

\*/

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

return supplier.get(); //返回一个最大值

}

}

## Consumer接口

### 概述

Consumer接口代表接受单一的输入变量而且没有返回值的一类操作。它的作用和Supplier相反，是消费一个数据的，消费的数据类型需要通过泛型指定。它的源代码如下：

package java.util.function;

import java.util.Objects;

@FunctionalInterface

public interface Consumer<T> {

//接受t对象，无返回值

void accept(T t);

}

### 抽象方法：accept

|  |  |
| --- | --- |
| Consumer接口中的方法 | 说明 |
| void accept(T t) | 接受一个参数，对参数进行处理，没有返回值。 |

### 示例：直接使用Consumer对象

* 实现步骤：

1. 使用Lambda创建Consumer对象，直接打印传入的字符串数据。
2. 调用Consumer的accept()方法，在accept()方法中传入一个字符串数据。

* 执行效果：



* 代码实现：

package com.itheima;

import java.util.function.Consumer;

public class Demo10Consumer {

public static void main(String[] args) {

//创建Consumer对象: 重写 void accept(T t) 方法

Consumer<String> consumer = s -> System.out.println(s);

//传递实参

consumer.accept("Hello World");

}

}

* 代码分析：

1. s -> System.out.println(s) 重写accept(String s)方法
2. 方法体：打印参数字符串
3. accept()调用重写的方法

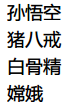
### 示例：Consumer做为参数

List和Set集合中遍历的forEach方法它的参数就是Consumer，请看下面的代码：

* 案例需求：

1. 创建一个数组，使用Arrays.asList("孙悟空", "猪八戒", "白骨精", "嫦娥") 转成List对象。
2. 使用forEach方法打印每一个元素，forEach中使用Lamba表达式传入的字符串

* 案例效果：



* 案例代码：

@Test

//参数就是Consumer

public void testForEach() {

List<String> names = Arrays.asList("孙悟空", "猪八戒", "白骨精", "嫦娥");

//使用lambd告诉forEach的消费行为: void accept(T t)

names.forEach(s -> System.out.println(s));

}

* 分析forEach()方法的源代码

default void forEach(Consumer<? super T> action) {

Objects.requireNonNull(action);

for (T t : this) {

action.accept(t);

}

}

这是定义在java.lang.Iterable接口中的默认方法，参数就是Consumer对象，方法体内对当前集合使用for遍历，this就是集合对象。每次对一个元素调用accept()方法。而我们外部调用的代码中对accept()方法进行了实现，输出了每个元素。

|  |  |
| --- | --- |
| java.utils.Objects类中的方法 | 说明 |
| public static <T> T requireNonNull(T obj) | 静态方法，JDK7中新增的方法，判断传入的对象是否为NULL，如果是NULL则抛出异常，不为NULL则返回对象本身。常用于方法或构造方法中传入对象参数的校验。 |

### Consumer接口中的默认方法：andThen()

在Consumer接口中有一个默认方法andThen()。它的参数和返回值都是Consumer类型，它实现了一种效果：消费一个数据的时候，首先做一个操作，然后再做另一个操作，两个操作依次执行，实现一种组合操作。方法内部其实是先调用自己的accept()方法，再调用参数传递进来的accept()方法。下面是JDK的源代码：

//默认的组合方法，参数和返回值都是Consumer类型，先调用自己的accept()方法，再调用参数的accept()方法

**default** Consumer<T> andThen(Consumer<? **super** T> after) {  
 Objects.requireNonNull(after); //判断after是否为null

//先调用自己的accept()方法，再调用参数的accept()方法  
 **return** (T t) -> { accept(t); after.accept(t); };   
}

要想实现组合，需要两个或多个Lambda表达式，而andThen的语义正是执行“一步接一步”操作。

### andThen方法演示示例

* 案例需求：

将字符串Hello首先打印大写的HELLO，然后打印小写的hello

* 案例效果：



* 实现步骤：

1. 创建Consumer对象c1，使用Lambda打印s对象的大写
2. 创建Consumer对象c2，使用Lambda打印s对象的小写
3. c1调用andThen(c2)方法，再调用accept("字符串")，完成依次的操作。

* 案例代码：

@Test

//andThen方法的使用

public void testAndThen() {

//void accept(T t) ，行为是打印大写

Consumer<String> c1 = s -> System.out.println(s.toUpperCase());

//第二个消费

Consumer<String> c2 = s -> System.out.println(s.toLowerCase());

//连续消费, 传递消费数据

c1.andThen(c2).accept("Hello Consumer");

//消费顺序不同

c2.andThen(c1).accept("Hello Boy");

}

* 代码分析

1. s -> System.out.println(s.toUpperCase()) 方法体实现了accept(String s)
2. c1就是Consumer对象，andThen()返回Consumer，可以调用accept()
3. andThen先调用自己的c1.accept()再调用c2的accept()方法

### 示例：使用Consumer做为参数

* 需求说明

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“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接口按照顺序组合操作。

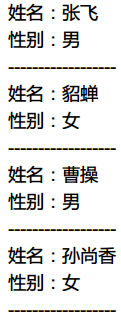
以下数组共5个元素，每个元素包含2项信息用逗号分隔：

String[] arr = { "张飞,男", "貂蝉,女", "曹操,男","孙尚香,女","小乔,女" };

* 实现步骤

1. 创建静态方法printInfo()，有3个参数，第1个是需要打印的字符串数组，第2个是Consumer<String>用于打印姓名name，第3个是Consumer<String>用于打印性别gender。
2. 在printInfo方法中遍历数组中每个元素，再调用name.andThen(gender).accept(单个元素)
3. 每调用一次andThen()方法，在下面输出一行横线
4. 在main函数中创建上面要遍历的数组
5. 调用printInfo方法，传入3个参数，第1个参数是数组，第2个参数使用Lambda打印姓名，参数s表示数组中的每个元素。第3个参数使用Lambda打印性别。

* 案例效果



* 代码实现

package com.itheima;

import java.util.function.Consumer;

public class Demo11Consumer {

public static void main(String[] args) {

String[] arr = { "张飞,男", "貂蝉,女", "曹操,男","孙尚香,女","小乔,女" };

//创建Consumer void accept(T t)

Consumer<String> name = s -> {

String n = s.split(",")[0];

System.out.println("姓名：" + n);

};

//创建打印性别

Consumer<String> gender = s -> System.out.println("性别：" + s.split(",")[1]);

//调用方法

printInfo(arr, name, gender);

}

/\*

创建静态方法printInfo()，有3个参数，

第1个是需要打印的字符串数组，

第2个是Consumer<String>用于打印姓名name，

第3个是Consumer<String>用于打印性别gender

\*/

public static void printInfo(String[] info, Consumer<String> name, Consumer<String> gender) {

for (String s : info) {

name.andThen(gender).accept(s);

System.out.println("-----------------------");

}

}

}

## Predicate接口

### 概述

Predicate中文意思为谓语，"我是一个程序员"，"是"或"不是"就是谓语。

它代表只有一个变量的函数，返回boolean类型。有时候我们需要进行某种判断，从而得到一个boolean值的结果。可以使用java.util.function.Predicate<T>接口。

以下是Predicate的源码：

@FunctionalInterface

public interface Predicate<T> {

boolean test(T t); //抽象方法，对t进行测试，返回boolean类型

}

### 抽象方法：test()

|  |  |
| --- | --- |
| Predicate接口中的方法 | 说明 |
| boolean test(T t) | 对t进行测试，返回boolean类型 |

### 示例：test()方法演示

* 案例效果：



* 案例需求：

判断test("字符串")方法给定的参数长度是否大于5

* 案例步骤：

1. 创建一个Predicate谓语对象，使用Lambda实现boolean test(T t)方法
2. 方法体的参数是s，返回字符串的长度大于5，省略return关键字。
3. 两次调用test()方法看运行结果，第1次使用字符串Hello，第2次使用字符串Predicate

* 案例代码：

@Test

public void testStrLength() {

//1) 创建一个Predicate谓语对象，使用Lambda实现boolean test(T t)方法

Predicate<String> predicate = s -> s.length() > 5;

// 两次调用test()方法看运行结果，第1次使用字符串Hello，第2次使用字符串Predicate

System.out.println("Hello的长度是否大于5:" + predicate.test("Hello"));

System.out.println("Predicate的长度是否大于5:" + predicate.test("Predicate"));

}

* 代码分析

1. s -> s.length() > 5 相当于实现了boolean test(T t)方法
2. predicate.test("Hello")调用了重写的test()方法并且传递了参数"Hello"

### 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。这个默认方法接收一个Predicate参数，返回一个Predicate参数。

其JDK源码为：

/\*\*

\* 组合方法，将当前的谓语与另一个谓语进行短路的与操作，返回一个谓语对象

\*/

default Predicate<T> and(Predicate<? super T> other) {

Objects.requireNonNull(other); //判断other是否为空

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

}

### and方法演示示例：

* 案例效果：



* 案例需求：

判断一个字符串是否包含指定的字符串：既包含大写“H”，又要包含大写“W”

* 案例步骤：

1. 创建2个需要判断的字符串：s1="Hello world"和s2="Hello World"
2. 使用Lambda表达式，创建两个Predicate对象
3. 判断字符串s是否包含H
4. 判断字符串s是否包含W
5. 调用and方法和test方法，分别输出s1和s2的结果

* 案例代码：

@Test

public void testAnd() {

String s1 = "Hello world";

String s2 = "Hello World";

//创建Predicate, boolean test(T t)方法

Predicate<String> p1 = s -> s.contains("H");

Predicate<String> p2 = s -> s.contains("W");

//先判断p1再p2

System.out.println(p1.and(p2).test(s1));

System.out.println(p1.and(p2).test(s2));

}

* 代码分析：

1. s -> s.contains("H") 生成一个内部类，重写了test()
2. s -> s.contains("W") 生成一个内部类，重写test()方法
3. p1.and(p2).test(s1) ，使用第1个条件 ，再使用第2个条件，提供s1做为参数

### 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**”操作。这个默认方法接收一个Predicate参数，返回一个Predicate参数。JDK源码为：

/\*\*

\* 组合方法，将当前的谓语与另一个谓语进行短路的或操作，返回一个谓语对象

\*/

default Predicate<T> or(Predicate<? super T> other) {

Objects.requireNonNull(other);

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

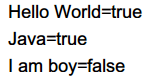
}

### or 方法演示示例：

* 案例需求：

判断指定的字符串的长度大于10或者小于5

* 案例效果：



* 案例步骤：

1. 创建三个字符串s1,s2,s3内容，分别是："Hello World" "Java" "I am boy"
2. 使用Lambda创建2个Predicate接口对象，第1个判断长度是否大于10，每2个判断长度是否小于5
3. 调用or和test方法输出s1，s2，s3每个字符串的测试结果

* 案例代码：

@Test

public void testOr() {

String s1 = "Hello World";

String s2 = "Java";

String s3= "I am boy";

//创建2个Predicate boolean test(T t)方法

Predicate<String> p1 = s -> s.length() > 10;

Predicate<String> p2 = s -> s.length() < 5;

System.out.println(p1.or(p2).test(s1)); //true

System.out.println(p1.or(p2).test(s2)); //true

System.out.println(p1.or(p2).test(s3)); //false

}

### 默认方法negate()

“与”、“或”已经了解了，剩下的“非”（取反）也会简单。方法没有参数，返回值为Predicate。默认方法negate的JDK源代码为：

/\*\*

\* 对当前的谓语进行逻辑非操作，返回一个谓语对象

\*/

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。要在test方法调用之前调用negate方法，正如and和or方法一样。

* 案例需求：

判断年龄是否小于18岁，将判断的结果取反。

* 案例效果



* 案例步骤

1. 创建2个整数类型的年龄，一个25，一个15岁。
2. 使用Lambda创建1个Predicate，判断年龄小于18岁。
3. 使用nagate()取反以后再调用test()方法，输出两个年龄的结果

* 案例代码

@Test

public void testNegate() {

int a1 = 25;

int a2 = 15;

//判断 条件是否大于18, boolean test(T t)方法

Predicate<Integer> predicate = i -> i > 18;

System.out.println(predicate.test(a1)); //true

System.out.println(predicate.negate().test(a1)); //false

System.out.println(predicate.negate().test(a2)); //true

}

* 代码分析：

1. i -> i > 18 相当于实现test()方法，返回一个Predicate的对象
2. predicate.negate().test(a1) 调用默认方法negate() 返回Predicate对象，再调用test()方法。

### 静态方法isEqual ()

Predicate中唯一的静态方法，方法的参数是两个Object类型，返回一个Predicate类型。

作用：根据Objects.equals(Object, Object)方法比较两个参数是否相等，一个对象通过isEqual()传入，另一个对象通过test()传入。

|  |  |
| --- | --- |
| java.util.Objects类中的方法 | 说明 |
| public static boolean equals(Object a, Object b) | 作用：用于比较两个对象是否相等  参数：a和b是要比较的两个对象  返回：如果两个对象相等，则返回true，否则返回false |

JDK源代码为：

/\*\*

\* 静态方法，判断test(object)方法传入的对象是否与参数targetRef对象相等

\*/

static <T> Predicate<T> isEqual(Object targetRef) {

return (null == targetRef) //传入的对象如果为空

? Objects::isNull //静态方法引用，下次课学习，调用Objects.isNull()方法，当前对象为null则返回true

: object -> targetRef.equals(object); //否则调用targetRef的equals()方法判断是否相等

}

* 案例需求：

比较两个字符串是否相等

* 案例效果：



* 案例步骤：

1. 通过静态方法isEqual("newboy")，直接返回Predicate对象
2. 调用Predicate中的test()方法传入另两个字符串分别比较

* 案例代码：

@Test

public void testEquals() {

//得到一个Preidate对象

Predicate<String> p1 = Predicate.isEqual("newboy");

//通过test传另一个对象

System.out.println(p1.test("newboy"));

System.out.println(p1.test("NewBoy"));

}

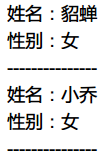
### Predicate的应用示例

* 需求说明

集合当中有多条“姓名+性别”的信息如下："张飞,男", "貂蝉,女", "曹操,男","孙尚香,女","小乔,女"

请通过Predicate接口的and组合方法，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) 姓名为两个字

* 案例效果：



* 开发步骤：

1. 创建第1个Predicate判断条件：使用逗号分隔的第0个元素姓名长度是2
2. 创建第2个Predicate判断条件：使用逗号分隔的第1个元素性别等于女
3. 创建一个新的List集合，用于存储过滤以后符合条件的字符串
4. 使用List中的forEach(Lambda)遍历上面的原始List集合，使用Predicate中的and和test方法判断每个元素
5. 两个条件都为真才添加到新的List集合中
6. 创建第1个Consumer接口，输出使用逗号分隔的第0个元素姓名
7. 创建第2个Consumer接口，输出使用逗号分隔的第1个元素性别
8. 使用List中的forEach(Lambda)遍历，输出过滤后的新的集合
9. 使用Consumer接口中的andThen和accept方法，输出每一个元素

* 案例代码：

@Test

public void testPredicate() {

String names[] = {"张飞,男", "貂蝉,女", "曹操,男","孙尚香,女","小乔,女"};

// 1) 创建第1个Predicate判断条件：使用逗号分隔的第0个元素姓名长度是2

// boolean test(T t)

Predicate<String> p1 = s -> s.split(",")[0].length() == 2;

// 2) 创建第2个Predicate判断条件：使用逗号分隔的第1个元素性别等于女

Predicate<String> p2 = s -> s.split(",")[1].equals("女");

// 3) 创建一个新的List集合，用于存储过滤以后符合条件的字符串

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

for (int i = 0; i <names.length ; i++) {

//表示同时符合两个条件

// 4) 使用List中的forEach(Lambda)遍历上面的原始List集合，使用Predicate中的and和test方法判断每个元素

// 5) 两个条件都为真才添加到新的List集合中

if (p1.and(p2).test(names[i])) {

list.add(names[i]);

}

}

// 6) 创建第1个Consumer接口，输出使用逗号分隔的第0个元素姓名

Consumer<String> c1 = s -> System.out.println("姓名：" + s.split(",")[0]);

Consumer<String> c2 = s -> System.out.println("性别：" + s.split(",")[1]);

// 7) 创建第2个Consumer接口，输出使用逗号分隔的第1个元素性别

// void accept(T t)

// 8) 使用List中的forEach(Lambda)遍历，输出过滤后的新的集合

// 9) 使用Consumer接口中的andThen和accept方法，输出每一个元素

list.forEach(s -> c1.andThen(c2).accept(s));

}

## Function接口

### 概述

Function<T,R>接口：根据一个参数得到另一个参数值，前面称为计算的参数，后面称为计算的结果。有进有出，所以称为“函数Function”。类似于数学中的函数，通过一个变量求出另一个变量的值。

以下是它的Java源代码

import java.util.Objects;

/\*\*

\* 代表通过一个变量求出另一个变量的结果的函数

\* @param <T> 输入给函数的变量

\* @param <R> 函数输出的结果

\*/

@FunctionalInterface

public interface Function<T, R> {

/\*\*

\* 对给定的变量t进行计算，得到返回的结果R

\*/

R apply(T t);

}

### 抽象方法：apply()

|  |  |
| --- | --- |
| java.util.function.Function接口中的方法 | 说明 |
| R apply(T t); | 对给定的变量t进行计算，得到返回的结果R |

### apply方法演示示例：

* 案例需求

将Integer类型转换为String类型，并且输出转换以后字符串的长度。

* 案例效果



* 案例步骤

1. 创建一个Function对象，输入类型是整数，输出类型是字符串
2. Lambda表达式将一个整数i转成字符串
3. 调用apply(数字)方法得到转换后的字符串，再调用字符串的length()方法得到长度，打印输出。
4. 第1次转换99这个数字，第2次转换1000这个数字。

* 案例代码

@Test

public void testApply() {

//应用

// 1) 创建一个Function对象，输入类型是整数，输出类型是字符串

// 2) Lambda表达式将一个整数i转成字符串

// R apply (T t)

Function<Integer, String> function = integer -> Integer.toString(integer);

// 3) 调用apply(数字)方法得到转换后的字符串，再调用字符串的length()方法得到长度，打印输出。

System.out.println(function.apply(99).length());

// 4) 第1次转换99这个数字，第2次转换1000这个数字

System.out.println(function.apply(1000).length());

}

* 代码分析：

1. integer -> Integer.toString(integer) 重写方法 String apply(Integer i)
2. function.apply(99) 调用已经实现的方法

### 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。先计算当前函数，再计算传入的函数。两个函数依次执行。andThen方法的参数是Function对象，返回一个Function对象。JDK源代码如：

/\*\*

\* 默认组合方法，先计算当前函数，再计算传入的函数

\*/

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

Objects.requireNonNull(after);

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

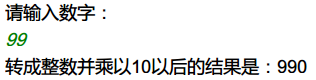
}

### andThen方法演示示例：

* 案例需求：

连续进行两个操作：第1个操作是将字符串转换成为int数字，第2个操作将转换好的数字乘以10。两个操作按照前后顺序组合到一起。

* 案例效果：



* 案例步骤：

1. 让用户从键盘输入1个数字，使用字符串接收。
2. 创建第1个Function函数将字符串转成整数
3. 创建第2个函数将整数乘以10返回
4. 调用andThen方法和apply，并且输出结果

* 案例代码：

@Test

public void testAndThen() {

// 1) 让用户从键盘输入1个数字，使用字符串接收。

// 2) 创建第1个Function函数将字符串转成整数

//R apply (T t)

Function<String, Integer> f1 = s -> Integer.parseInt(s);

// 3) 创建第2个函数将整数乘以10返回

Function<Integer, Integer> f2 = i -> i \* 10;

// 4) 调用andThen方法和apply，并且输出结果

System.out.println(f1.andThen(f2).apply("99"));

}

* 代码分析

1. s -> Integer.parseInt(s) 相当于重写Integer apply(String s)

### 默认方法：compose()

Function中有一个与andThen非常类似的compose方法。中文是"组成"的意思，方法参数是Function，返回值是Function，先运行参数的apply方法，再调用自己的apply方法。其JDK源代码为：

/\*\*

\* 默认组合方法，先计算传入的函数，再计算当前函数

\*/

default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {

Objects.requireNonNull(before);

return (V v) -> apply(before.apply(v));

}

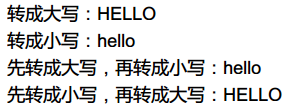
结合andThen方法的JDK源码实现进行对比，会发现compose方法的参数Lamda将会先执行。所以二者只是先后顺序的不同而已。

### compose方法的演示

* 案例需求：

创建两个函数对象：1个将字符串转成大写，1个将字符串转成小写。分别使用andThen和compose方法组合调用，查看不同的计算结果。

* 案例效果：



* 开发步骤：

1. 创建第1个Function，输入输出都是String类型，将字符串转成大写。
2. 创建第2个Function，输入输出都是String类型，将字符串转成小写。
3. 调用第1个函数的apply方法，并且输出值
4. 调用第2个函数的apply方法，并且输出值
5. 调用andThen方法和apply方法查看运行结果
6. 调用compose方法和apply方法查看运行结果

* 案例代码：

@Test

public void testCompose() {

String str = "Hello";

// 1) 创建第1个Function，输入输出都是String类型，将字符串转成大写。

Function<String, String> f1 = s -> s.toUpperCase();

// 2) 创建第2个Function，输入输出都是String类型，将字符串转成小写。

Function<String, String> f2 = s -> s.toLowerCase();

// 3) 调用第1个函数的apply方法，并且输出值

System.out.println(f1.apply(str)); //大写

// 4) 调用第2个函数的apply方法，并且输出值

System.out.println(f2.apply(str)); //小写

// 5) 调用andThen方法和apply方法查看运行结果

System.out.println(f1.andThen(f2).apply(str));

// 6) 调用compose方法和apply方法查看运行结果

System.out.println(f1.compose(f2).apply(str));

}

### Function的应用示例

* 需求说明

请使用Function进行函数拼接，按照顺序执行多个函数。操作依次为：

1. 将字符串"赵丽颖,20"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数字加100，得到结果int数字。

* 案例效果



* 开发步骤：

1. 创建第1个Function对象，将字符串20取出，返回一个字符串
2. 创建第2个Function对象，将字符串转成整数，返回整数
3. 创建第3个Function对象，将整数加100，返回计算结果
4. 调用andThen方法2次，apply方法应用字符串："赵丽颖,20"，输出结果

* 代码实现

@Test

public void testFunction() {

String str = "赵丽颖,20";

// 1) 将字符串"赵丽颖,20"截取数字年龄部分，得到字符串；

Function<String,String> f1 = s -> s.split(",")[1];

// 2) 将上一步的字符串转换成为int类型的数字；

Function<String, Integer> f2 = s -> Integer.parseInt(s);

// 3) 将上一步的int数字加100，得到结果int数字

Function<Integer, Integer> f3 = i -> i + 100;

System.out.println(f1.andThen(f2).andThen(f3).apply(str));

}

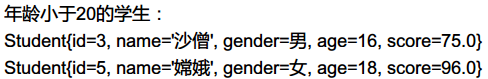
# 作业：函数式接口的案例

* 案例说明

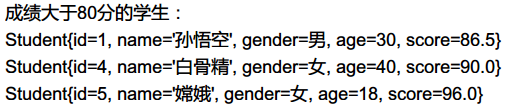
设计一个学生类，该类包含字段：编号int，姓名String，性别char，年龄int，成绩double。根据不同的条件对学生的信息进行过滤，返回过滤后的集合并且输出。

过滤条件如下：

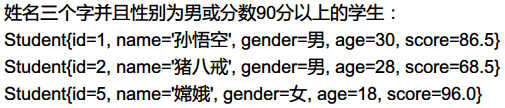
* 1. 年龄小于20岁的，返回一个集合。



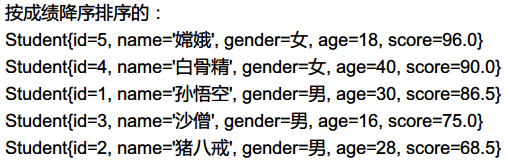
* 1. 成绩大于80分的，返回一个集合。



* 1. 姓名是三个字，性别为男，或分数90分以上的返回一个集合



* 1. 按成绩的降序排序输出原始集合中所有的元素



* 案例步骤

1. 创建学生Student类，添加属性：编号int，姓名String，性别char，年龄int，成绩double
2. 生成全参的构造方法和无参的构造方法，生成toString()方法，生成getter/setter方法
3. 创建过滤的静态方法filter，参数1：原始的学生列表，参数2：Predicate接口，返回值：过滤符合条件的学生集合。
4. 编写filter方法，首先创建空的List集合，用于存储过滤以后的结果。遍历传入的学生对象，对每个学生使用predicate的test()方法进行判断，符合条件的添加到集合中，最后返回添加好的集合对象。
5. 创建主类，创建5个学生放入集合对象。如下：

new Student(1,"孙悟空",'男',30,86.5),

new Student(2,"猪八戒",'男',28,68.5),

new Student(3,"沙僧",'男',16,75),

new Student(4,"白骨精",'女',40,90),

new Student(5,"嫦娥",'女',18,96)

1. 调用filter方法，使用Lambda创建Predicate，条件是年龄小于20岁。使用Lambda输出过滤后的集合。
2. 调用filter方法，使用Lambda创建Predicate，条件是成绩大于80分的。使用Lambda输出过滤后的集合。
3. 创建三个Predicate，1个是姓名长度为3，1个是性别等于男，1个是分数大于90。使用and和or方法组合生成Predicate对象。再调用filter方法，输出过滤后的集合。
4. 使用List接口中的默认sort()方法对学生集合排序，方法传入的参数是Comparator比较器，使用Lambda表达式创建比较器对象。方法体中调用Double中的compare(o1,o2) 方法，比较2个学生的成绩。因为是降序，所以学生2写在前面。
5. 使用forEach输出排序后的学生集合。