1. 函数式接口
2. 概念

有且仅有一个抽象方法的接口。该接口可以用具体的实现类，匿名内部类，Lambda表达式实现。最方便简介的是Lambda表达式实现，注意Lambda表达式都是用于重写接口的抽象方法的。

1. 函数式接口作方法的参数

函数式接口可以作为方法的参数，用匿名内部类实现或者Lambda表达式实现。

package 函数式接口;  
  
public class RunnableDemo {  
 public static void main(String[] args) {  
// startThread(new Runnable() {  
// @Override  
// public void run() {  
// System.out.println(Thread.currentThread().getName() + "线程启动");  
// }  
// });  
  
 //Lambda  
 *startThread*( () -> System.*out*.println(Thread.*currentThread*().getName() + "线程启动"));  
  
 }  
  
 private static void startThread(Runnable r){  
 new Thread(r).start();  
 }  
  
}

1. 函数式接口作方法的返回值

package 函数式接口;  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
  
public class ComparatorDemo {  
 public static void main(String[] args) {  
 ArrayList<String> array = new ArrayList<String>();  
  
 array.add("CC");  
 array.add("D");  
 array.add("B");  
 array.add("A");  
  
 System.*out*.println("排序前：" + array);  
  
 Collections.*sort*(array, *getComparator*());  
  
 System.*out*.println("排序后：" + array);  
 }  
  
 private static Comparator<String> getComparator(){  
 //匿名内部类  
// return new Comparator<String>() {  
// @Override  
// public int compare(String s1, String s2) {  
// return s1.length() - s2.length();  
// }  
// };  
  
 //Lambda  
 return (s1, s2) -> s1.length() - s2.length();  
 }  
}

1. 常用的函数式接口
2. Supplier接口
3. 概述

生产型接口，产生对应的类型数据。接口中有一个抽象方法get()用于产生数据。

1. 方法

方法名 说明

T get() 按照某种实现逻辑(由Lambda表达式实现)返回一个数据

1. 代码实现

package 函数式接口;  
  
import java.util.function.Supplier;  
  
public class SupplierDemo {  
 public static void main(String[] args) {  
 //Lambda重写该抽象方法  
 Integer i = *getInteget*(()->30);  
 System.*out*.println(i);  
  
 String s = *getString*( ()->"xx");  
 System.*out*.println(s);  
  
 }  
  
 private static Integer getInteget(Supplier<Integer> sup){  
 //调用Supllier接口中的唯一抽象方法  
 return sup.get();  
 }  
  
 private static String getString(Supplier<String> sup){  
 return sup.get();  
 }  
  
  
}

1. Consumer接口
2. 概述

消费型接口，对给定的数据进行操作。不同于Supplier，Consumer用于进行数据的操作。

1. 方法

方法名 说明

void accept(T t) 对给定的参数执行此操作

default Consumer andThen(Consumer

after)

返回一个组合的Consumer，依次执行此操作，然后执行

after操作

1. 代码实现

package 函数式接口;  
  
import java.util.function.Consumer;  
  
public class ConsumerDemo {  
 public static void main(String[] args) {  
 *operatorString*("xx", s -> System.*out*.println(s), s -> System.*out*.println("dd"));  
 }  
  
 //连续处理两次一个String  
 private static void operatorString(String name, Consumer<String> con1, Consumer<String> con2){  
 con1.andThen(con2).accept(name);  
 }  
  
 //处理一次  
 private static void operatorString(String name, Consumer<String> con){  
 con.accept(name);  
 }  
}

1. Predicate接口
2. 概述

布尔逻辑判断，与或非，对于参数是否满足指定的条件进行判断，通常用于筛选数据，筛选出满足条件的数据。

1. 方法

test()是抽象方法，用于对条件进行判断。

方法名 说明

boolean test(T t)

对给定的参数进行判断(判断逻辑由Lambda表达式实现)，返回

一个布尔值

default Predicate negate() 返回一个逻辑的否定，对应逻辑非

default Predicate and(Predicate

other)

返回一个组合判断，对应短路与

default Predicate or(Predicate

other)

返回一个组合判断，对应短路或

3．代码实现

package 函数式接口;  
  
import java.util.function.Predicate;  
  
public class PredicateDemo {  
 public static void main(String[] args) {  
 boolean b1 = *checkString*("hello", s -> s.length() > 8);  
 System.*out*.println(b1);  
  
 boolean b2 = *checkString2*("hello", s -> s.length() > 8, s -> s.length() < 2);  
 System.*out*.println(b2);  
 }  
  
 //判断非逻辑  
 private static boolean checkString(String s, Predicate<String> pre){  
 return pre.negate().test(s);  
 }  
  
 //判断或逻辑  
 private static boolean checkString2(String s, Predicate<String> pre1, Predicate<String> pre2){  
 return pre1.or(pre2).test(s);  
 }  
  
 //判断与逻辑  
 private static boolean checkString3(String s, Predicate<String> pre1, Predicate<String> pre2){  
 return pre1.and(pre2).test(s);  
 }  
  
  
  
}

1. Function接口
2. 概述

也是对参数进行处理操作的，可以对同一参数进行多次操作。

1. 方法

Lambda表达式只是在重写抽象方法apply()。

方法名 说明

R apply(T t) 将此函数应用于给定的参数

default Function

andThen(Function after)

返回一个组合函数，首先将该函数应用于输入，然后将after函

数应用于结果

1. 代码实现

public class FunctionDemo {

  public static void main(String[] args) {

//操作一

    convert("100",s -> Integer.parseInt(s));

//操作二

    convert(100,i -> String.valueOf(i + 566));

    //使用andThen的方式连续执行两个操作

    convert("100", s -> Integer.parseInt(s), i -> String.valueOf(i + 566));

 }

  //定义一个方法，把一个字符串转换int类型，在控制台输出

  private static void convert(String s, Function<String,Integer> fun) {

//    Integer i = fun.apply(s);

    int i = fun.apply(s);

    System.out.println(i);

 }

  //定义一个方法，把一个int类型的数据加上一个整数之后，转为字符串在控制台输出

  private static void convert(int i, Function<Integer,String> fun) {

    String s = fun.apply(i);

    System.out.println(s);

 }

  //定义一个方法，把一个字符串转换int类型，把int类型的数据加上一个整数之后，转为字符串在控制台

输出

  private static void convert(String s, Function<String,Integer> fun1,

Function<Integer,String> fun2) {

    String ss = fun1.andThen(fun2).apply(s);

    System.out.println(ss);

 }

}

1. Stream流

使用Stream流可以极大地简化代码，将解决问题而忽略复杂语法发挥到了极致，体现了函数式编程的真谛。

1. 生成Stream流
2. Collection集合

默认方法stream()生成stream流

1. Map集合

将Map转为Set，在生成Stream流

1. 数组

通过Stream接口的静态方法of(T... values)生成流

1. 代码实现

package 函数式接口;  
  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
import java.util.stream.Stream;  
  
public class 生成Stream流 {  
 public static void main(String[] args) {  
 List<String> list = new ArrayList<String>();  
 Stream<String> listStream = list.stream();  
  
 Map<String, Integer> map = new HashMap<String, Integer>();  
 Stream<String> keyStream = map.keySet().stream();  
 Stream<Integer> valueStream = map.values().stream();  
 Stream<Map.Entry<String, Integer>> entryStream = map.entrySet().stream();  
  
 String[] strArray = {"hello", "world"};  
 Stream<String> strArrayStream = Stream.*of*(strArray);  
  
 }  
}

1. Stream流的中间操作
2. 概述

中间操作，即在执行完这些操作后，Stream流依然可以执行其他操作，不会终止。

1. 常用方法

方法名 说明

Stream filter(Predicate predicate) 用于对流中的数据进行过滤

Stream limit(long maxSize) 返回此流中的元素组成的流，截取前指定参数个数的数据

Stream skip(long n)

跳过指定参数个数的数据，返回由该流的剩余元素组成的

流

static Stream concat(Stream a, Stream

b)

合并a和b两个流为一个流

Stream distinct()

返回由该流的不同元素（根据Object.equals(Object) ）组

成的流

Stream sorted() 返回由此流的元素组成的流，根据自然顺序排序

Stream sorted(Comparator

comparator)

返回由该流的元素组成的流，根据提供的Comparator进行

排序

Stream map(Function mapper) 返回由给定函数应用于此流的元素的结果组成的流

IntStream mapToInt(ToIntFunction

mapper)

返回一个IntStream其中包含将给定函数应用于此流的元素

的结果

1. Stream流的终结操作
2. 概述

终结操作即最后一步操作，之后Stream流就会终止。

1. 常用方法

方法名 说明

void forEach(Consumer action) 对此流的每个元素执行操作

long count() 返回此流中的元素数