

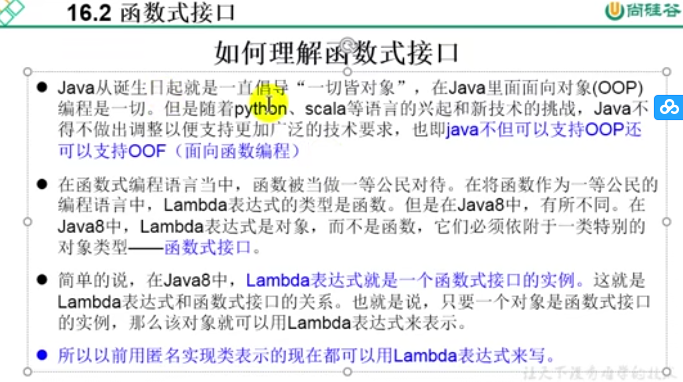
# 一 Lambda表达式



## 1 lambda表达式书写实例

**package** com.zy.study.java.java8newfeatures.lambdausedemo;  
  
**import** java.util.Comparator;  
**import** java.util.function.Consumer;  
  
***/\*\*  
 \* Lambda 用法  
 \* 1.举例： (o1, o2) -> Integer.compare(o1, o2);  
 \* 2.格式：  
 \* -> ：lambda操作符 或 箭头操作符  
 \* -> 左边 ：lambda形参列表，其实就是接口的匿名实现类中被覆写方法的形参列表  
 \* -> 右边 ：lambda体，其实就是接口匿名实现类中被覆写方法的方法体  
 \* 3.Lambda表达式的使用：（分六种情况）  
 \* 1）无参，无返回值  
 \* 2) lambda需要一个参数，无返回值  
 \* 3) 数据类型可以省略，因为可以由编译器推断得出，称为"类型推断"  
 \* 4) lambda需要两个或者两个以上参数，多条执行语句，并且可以有返回值。  
 \* 5) lambda需要两个或者两个以上参数，多条执行语句，并且可以有返回值  
 \* 6) 当lambda体只有一条语句时，return与大括号若有，都可以省略  
 \*  
 \* 总结：  
 \* -> 左边 ： lambda形参列表的参数类型可以省略（类型推断）；  
 \* 若lambda的形参列表只有一个参数，其一对()也可以省略；  
 \* 若lambda的形参列表没有参数，或者多个参数，其()就不要省略  
 \*  
 \* -> 右边 ：lambda体应该使用一对{}包裹；  
 \* 若lambda体只有一条执行语句（可能是return语句），则可以省略{}以及return关键字  
 \*  
 \* 4.Java中lambda表达式的本质： 作为函数式接口的一个实例  
 \*  
 \* 5.函数式接口：若一个接口只声明了一个抽象方法，则此接口就称为函数式接口  
 \*  
 \*/*public class** LambdaTest2 **{  
 *//语法格式1：无参，无返回值* public static void** test1()**{** Runnable r1 = **new Runnable**() **{** @Override  
 **public void** run() **{** System.***out***.println(**"匿名接口方式"**);  
 **}  
 }**;  
  
 r1.run();  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***//()->System.out.println("lambda表达式") 作为 Runnable 接口的一个实例*** Runnable r2 = ()->System.***out***.println(**"lambda表达式"**);  
 r2.run();  
 **}  
  
 *//语法格式2：lambda需要一个参数，无返回值* public static void** test2()**{** Consumer<String> consumer = **new Consumer**<String>() **{** @Override  
 **public void** accept(String s) **{** System.***out***.println(s);  
 **}  
 }**;  
 consumer.accept(**"一个参数，无返回值"**);  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 Consumer<String> consumer1 = s -> System.***out***.println(s);  
 Consumer<String> consumer2 = (String s) -> **{** System.***out***.println(s);  
 **}**;  
 consumer1.accept(**"lambda写法:省略类型"**);  
 consumer2.accept(**"lambda写法：不省略类型"**);  
 **}  
  
 *//语法格式3： 数据类型可以省略，因为可以由编译器推断得出，称为"类型推断"  
 //语法格式4： lambda只需要一个参数时，参数的小括号都可以省略* public static void** test3()**{** Consumer<String> consumer2 = (String s) -> **{** System.***out***.println(s);  
 **}**;  
 consumer2.accept(**"lambda写法：不省略类型"**);  
  
 Consumer<String> consumer1 = s -> System.***out***.println(s);  
 consumer1.accept(**"lambda写法:省略类型"**);  
 **}  
  
 *//语法格式5： lambda需要两个或者两个以上参数，多条执行语句，并且可以有返回值。* public static void** test4()**{** Comparator<Integer> comparator1 = **new Comparator**<Integer>() **{** @Override  
 **public int** compare(Integer o1, Integer o2) **{** System.***out***.println(**"入参1: "** + o1);  
 System.***out***.println(**"入参2: "** + o2);  
 **return** Integer.*compare*(o1, o2);  
 **}  
 }**;  
 **int** res1 = comparator1.compare(1,2);  
 System.***out***.println(res1);  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
  
 ***//lambda 表达式写法*** Comparator<Integer> comparator2 = (o1, o2) -> **{** System.***out***.println(**"lambda 表达式写法 入参1: "** + o1);  
 System.***out***.println(**"lambda 表达式写法 入参2: "** + o2);  
 **return** Integer.*compare*(o1, o2);  
 **}**;  
 **int** res2 = comparator2.compare(3,2);  
 System.***out***.println(res2);  
 **}  
  
 *//语法格式6： 当lambda体只有一条语句时，return与大括号若有，都可以省略。* public static void** test5()**{** Comparator<Integer> comparator1 = (o1, o2) -> **{  
 return** Integer.*compare*(o1, o2);  
 **}**;  
 **int** res1 = comparator1.compare(3,2);  
 System.***out***.println(res1);  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
  
 ***//编译器能够推断出唯一结果的都可以省略*** Comparator<Integer> comparator2 = (o1, o2) -> Integer.*compare*(o1, o2);  
 **int** res2 = comparator2.compare(3,2);  
 System.***out***.println(res2);  
 **}  
  
  
 public static void** main(String[] args) **{** *test5*();  
 **}  
  
}**

## 二 函数式（functional）接口



## 1 函数式接口简单示例

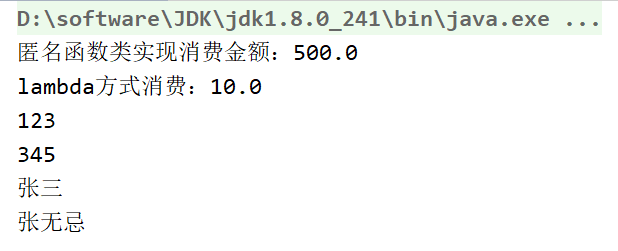
**package** com.zy.study.java.java8newfeatures.funinterdace;  
***/\*\*  
 \* @FunctionalInterface 显示的指明这是一个函数式接口  
 \* 也可用来检验一个接口是否是函数时接口  
 \* 只有一个抽象方法  
 \* 只有函数式接口才能用lambda表达式创建实例  
 \*  
 \* 在java.util.function包下定定义了JAVA8的丰富的函数式接口  
 \*/***@FunctionalInterface  
**public interface** MyInterface **{  
 void** method1();  
**}**

## 2 Java中内置四大核心函数式接口

|  |  |  |  |
| --- | --- | --- | --- |
| 函数式接口 | 参数类型 | 返回值类型 | 用途 |
| Consumer<T>  消费型接口，包含方法：  void accept(T t); | T | void | 对类型为T的对象应用操作 |
| Supplier<T>  供给型接口包含方法：  T get(); | / | T | 返回类型为T的对象 |
| Function<T, R>  函数型接口，包含方法：  R apply(T t); | T | R | 对类型为T的对象应用操作，并返回结果，结果是R类型的对象。 |
| Predicate<T>  断定型接口，包含方法：  boolean test(T t); | T | boolean | 确定类型为T的对象是否满足某约束，并返回Boolean值 |

## 3 内置函数式接口使用实例

**package** com.zy.study.java.java8newfeatures.funinterdace;  
  
**import** java.util.ArrayList;  
**import** java.util.Arrays;  
**import** java.util.List;  
**import** java.util.function.Consumer;  
**import** java.util.function.Predicate;  
  
***/\*\*  
 \* 函数式接口：若一个接口只声明了一个抽象方法，则此接口就称为函数式接口  
 \*  
 \* Java内置4大核心函数是接口  
 \* 消费型接口 Consumer<T> 包含函数 void accept(T t);  
 \* 供给型接口 Supplier<T> 包含函数 T get();  
 \* 函数型接口 Function<T, R> 包含函数 R apply(T t);  
 \* 断定型接口 Predicate<T> 包含函数 boolean test(T t);  
 \*/*public class** FunctionalInterfacetest **{  
 public static void** main(String[] args) **{  
 */\*  
 \* 消费示例  
 \* \*/  
 //调用方式1*** *consumerMoney*(500, **new Consumer**<Double>() **{** @Override  
 **public void** accept(Double aDouble) **{** System.***out***.println(**"匿名函数类实现消费金额："** + aDouble);  
 **}  
 }**);  
  
 ***//调用方式2*** *consumerMoney*(10, m1 -> System.***out***.println(**"lambda方式消费："** + m1));  
  
 ***/\*  
 \* 字符串过滤实例  
 \* \*/  
 //过滤字符串，返回长度大小于4的字符串集合*** List<String> list1 = *filter*(Arrays.*asList*(**"123"**, **"4567"**, **"345"**), **new Predicate**<String>() **{** @Override  
 **public boolean** test(String s) **{  
 if** (s.length()<4)**{  
 return true**;  
 **}  
 return false**;  
 **}  
 }**);  
 list1.forEach(System.***out***::println);  
  
 ***//过滤字符串，返回含有张字的字符串集合*** List<String> list2 = *filter*(Arrays.*asList*(**"张三"**, **"张无忌"**, **"王小二"**), s -> s.contains(**"张"**));  
 list2.forEach(System.***out***::println);  
  
 **}  
  
 *//该方法是用来消费money，具体消费的规则由调用者提供* public static void** consumerMoney(**double** money, Consumer<Double> consumer)**{** consumer.accept(money);  
 **}  
  
 *//该方法用来批量过滤字符串，然后把符合条件的字符串返回，字符串的过滤条件由调用者提供* public static** List<String> filter(List<String> list, Predicate<String> predicate)**{** ArrayList<String> result = **new** ArrayList<>(32);  
 **for** (String s: list) **{  
 if** (predicate.test(s))**{** result.add(s);  
 **}  
 }  
 return** result;  
 **}  
}**



## 4 其他内置函数式接口



# 三 方法引用与构造器引用

## 1 方法引用介绍



## 2 方法引用的三种使用情况

**package** com.zy.study.java.java8newfeatures.methodreference;  
  
**import** com.zy.study.java.java8newfeatures.methodreference.entity.User;  
  
**import** java.io.PrintStream;  
**import** java.util.Comparator;  
**import** java.util.function.BiPredicate;  
**import** java.util.function.Consumer;  
**import** java.util.function.Function;  
**import** java.util.function.Supplier;  
  
***/\*\*  
 \* 1. 使用情景：当要传递给lambda体的操作，已经有实现的方法，可以使用方法引用  
 \*  
 \* 2. 方法引用本质上就是lambda表达式，而lambda表达式作为函数式接口的实例，所以  
 \* 方法引用也是函数式接口的实例  
 \* 3. 使用格式： 类/对象(方法的调用者)::方法名  
 \* 4. 具体分为如下三种情况：  
 \* 情况1 对象::非静态方法  
 \* 情况2 类::静态方法  
 \* 情况3 类::非静态方法  
 \* 5.方法引用使用的要求： 传递给函数式接口中函方法的操作方法的 形参列表和返回值类型得和  
 \* 函数式接口中的方法的形参列表和返回值相同（主要针对情况1，2）  
 \*/*public class** MethodReferenceTest **{  
 */\*  
 \* 情况1：  
 \* Consumer中的void accept(T t)  
 \* PrintStream中的void println(T t)  
 \* 两个方法的返回值和形参相同，传递给accept的打印操作的实现方法是println  
 \* \*/* public static void** test1()**{** Consumer<String> consumer = str -> System.***out***.println(str);  
 consumer.accept(**"lambda表达式"**);  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 PrintStream pt = System.***out***;  
 ***//给Consumer的accept方法中传递的是PrintStream对象的println方法*** Consumer<String> consumer1 = pt::println;  
 ***//Consumer<String> consumer1 =System.out::println;*** consumer1.accept(**"方法引用"**);  
 **}  
  
 */\*  
 \* Supplier中的T get()和User中的getName 的方法样子类似，无参，有返回值  
 \* 传递给T get() 的user对象的获取名字的方法已经有实现了  
 \* \*/* public static void** test2()**{** User user = **new** User();  
 user.setName(**"Supplier lambda实现"**);  
 Supplier<String> stringSupplier = () -> user.getName();  
 System.***out***.println(stringSupplier.get());  
  
 user.setName(**"Supplier 方法引用用实现"**);  
 ***//给Supplier的get方法中 传递的是User对象的getName方法*** Supplier<String> stringSupplier2 = user::getName;  
 System.***out***.println(stringSupplier.get());  
  
 **}  
  
 */\*  
 \* 情况2： 类::静态方法  
 \* Comparator中的int compare(T o1, T o2);  
 \* Integer中的int compare(int x, int y)  
 \* \*/* public static void** test3()**{** System.***out***.println(**"情况2： 类::静态方法"**);  
 Comparator<Integer> comparator = (t1,t2)->Integer.*compare*(t1,t2);  
 System.***out***.println(comparator.compare(1,2));  
  
 Comparator<Integer> comparator1 = Integer::*compare*;  
 System.***out***.println(comparator1.compare(2,1));  
 **}  
  
 */\*  
 \* Function 中的R apply(T t);  
 \* Math 中的public static long round(double a)  
 \* \*/* public static void** test4()**{** Function<Double,Long> function = t1-> Math.*round*(t1);  
 System.***out***.println(**"function: "** +function.apply(3.12));  
  
 Function<Double,Long> function2 = Math::*round*;  
 System.***out***.println(**"function2: "** +function.apply(53.1));  
 **}  
  
 */\*  
 \* 情况3： 类::实例方法(有难度)  
 \* Comparator 中的 int compare(T o1, T o2);  
 \* String中的o1.compareTo(o2)  
 \* \*/* public static void** test5()**{  
 *//第一个参数作为方法的调用者，第二个参数作为方法从入参  
 //写方法引用的时候可以用类名*** Comparator<String> comparator = (s1,s2) -> s1.compareTo(s2);  
 System.***out***.println(**"情况3: "** + comparator.compare(**"qwe"**,**"wqe"**));  
  
 Comparator<String> comparator2 = String::compareTo;  
 System.***out***.println(**"情况3:方法引用: "** + comparator2.compare(**"qwe"**,**"wqe"**));  
 **}  
  
 */\*  
 \* BiPredicate 中的 boolean test(T t, U u);  
 \* String中的 public boolean equals(Object anObject)  
 \* \*/* public static void** test6()**{** BiPredicate<String,String> biPredicate = (s1,s2) ->s1.equals(s2);  
 System.***out***.println(**"情况3 biPredicate: "** + biPredicate.test(**"qwe"**,**"wqe"**));  
  
 BiPredicate<String,String> biPredicate2 = String::equals;  
 System.***out***.println(**"情况3 biPredicate: "** + biPredicate2.test(**"qwe"**,**"wqe"**));  
 **}  
  
 */\*  
 \* BiPredicate 中的 boolean test(T t, U u);  
 \* User 中的 String getName()  
 \* \*/* public static void** test7()**{** User user = **new** User();  
 user.setName(**"王八蛋"**);  
 Function<User, String> function = u -> u.getName();  
 System.***out***.println(**"情况3 function: "** + function.apply(user));  
  
 Function<User, String> function2 = User::getName;  
 System.***out***.println(**"情况3 function2: "** + function2.apply(user));  
 **}  
  
 public static void** main(String[] args) **{** *test1*();  
 *test2*();  
 *test3*();  
 *test4*();  
 *test5*();  
 *test6*();  
 *test7*();  
 **}  
}**

lambda表达式

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

方法引用

Supplier lambda实现

Supplier 方法引用用实现

情况2： 类::静态方法

-1

1

function: 3

function2: 53

情况3: -6

情况3:方法引用: -6

情况3 biPredicate: false

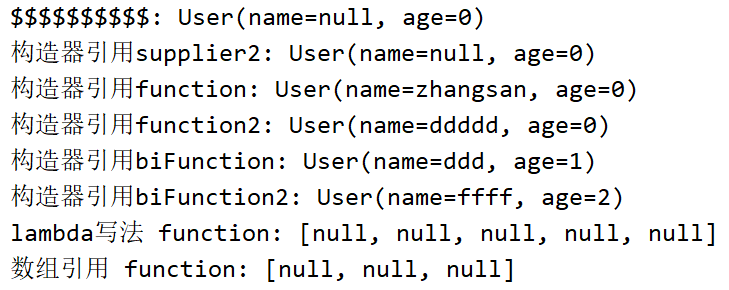
情况3 biPredicate: false

情况3 function: 王八蛋

情况3 function: 王八蛋

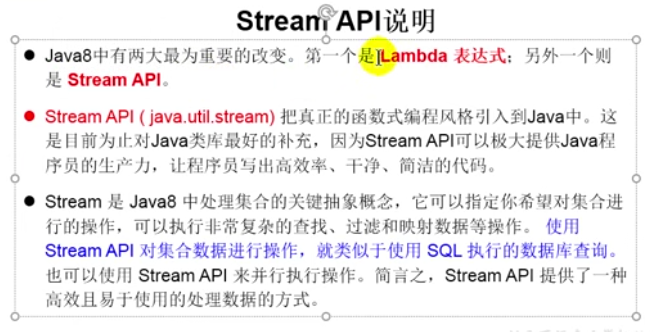
## 3 构造器引用与数组引用

**package** com.zy.study.java.java8newfeatures.methodreference;  
  
  
**import** com.zy.study.java.java8newfeatures.methodreference.entity.User;  
  
**import** java.util.Arrays;  
**import** java.util.function.BiFunction;  
**import** java.util.function.Function;  
**import** java.util.function.Supplier;  
  
***/\*\*  
 \* 一. 构造器引用  
 \* 和方法引用类似，函数式接口的抽象方法的形参列表和构造器的形参列表一致  
 \* 抽象方法的返回值即为构造器所属类的类型。  
 \*  
 \* 二. 数组引用  
 \* 可以把数组看作是一个特殊的类，则写法与构造器引用一致  
 \*/*public class** ConstructorRefTest **{  
 */\*  
 \* 构造器引用  
 \* Supplier中的T get()  
 \* User的空参构造器  
 \* \*/* public static void** test()**{** Supplier<User> supplier1 = **new Supplier**<User>() **{** @Override  
 **public** User get() **{  
 return new** User();  
 **}  
 }** ;  
 ***//###############################################*** Supplier<User> supplier = () -> **new** User();  
 System.***out***.println(**"$$$$$$$$$$: "** + supplier.get());  
 ***//###############################################  
 //构造器引用*** Supplier<User> supplier2 = User::**new**;  
 System.***out***.println(**"构造器引用supplier2: "** + supplier2.get());  
 **}  
  
 */\*  
 \* Function中的 R apply(T t);  
 \* \*/* public static void** test2()**{** Function<String,User> function = name -> **new** User(name);  
 System.***out***.println(**"构造器引用function: "** + function.apply(**"zhangsan"**));  
  
 Function<String,User> function2 = User::**new**;  
 System.***out***.println(**"构造器引用function2: "** + function2.apply(**"ddddd"**));  
 **}  
  
 */\*  
 \* BiFunction 中的 R apply(T t, U u);  
 \* \*/* public static void** test3()**{** BiFunction<Integer, String, User> biFunction = (i, s) ->**new** User(i,s);  
 System.***out***.println(**"构造器引用biFunction: "** + biFunction.apply(1,**"ddd"**));  
  
 BiFunction<Integer, String, User> biFunction2 = User::**new**;  
 System.***out***.println(**"构造器引用biFunction2: "** + biFunction2.apply(2,**"ffff"**));  
 **}  
  
 */\*  
 \* 数组引用  
 \*Function 中的 R apply(T t);  
 \* \*/* public static void** test4()**{  
 *//Function<Integer, String> 第一个范型表示定义的数组长度*** Function<Integer, String[]> function = length -> **new** String[length];  
 String[] arr1 = function.apply(5);  
 System.***out***.println(**"lambda写法 function: "** + Arrays.*toString*(arr1));  
  
 Function<Integer, String[]> function2 = String[] :: **new**;  
 String[] arr2 = function2.apply(3);  
 System.***out***.println(**"数组引用 function: "** + Arrays.*toString*(arr2));  
  
 **}  
  
 public static void** main(String[] args) **{** *test*();  
 *test2*();  
 *test3*();  
 *test4*();  
 **}  
}**

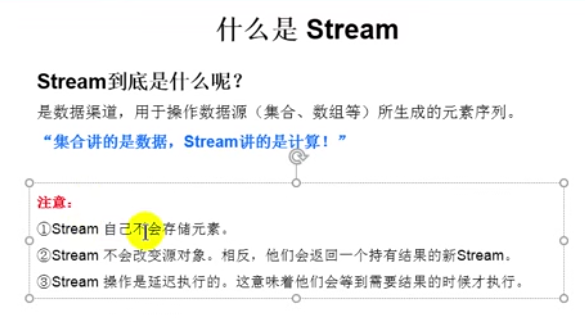


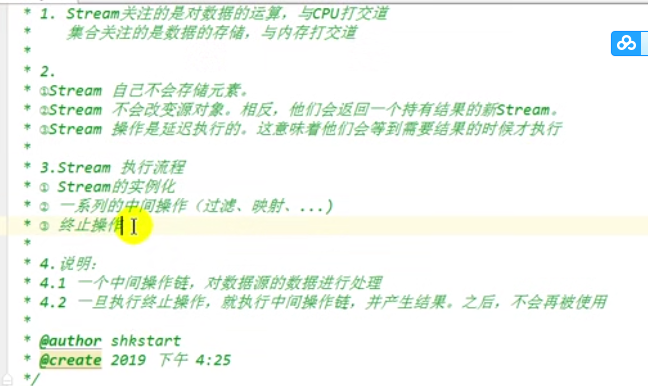
# 四 强大的Stream API

## 1 stream API说明







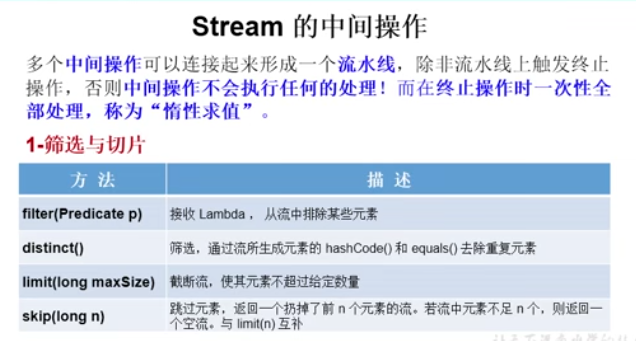




## 2 stream的创建

**package** com.zy.study.java.java8newfeatures.streamapi;  
  
**import** com.zy.study.java.java8newfeatures.methodreference.entity.User;  
  
**import** java.util.Arrays;  
**import** java.util.List;  
**import** java.util.stream.IntStream;  
**import** java.util.stream.Stream;  
  
***/\*\*  
 \* @autho 18392  
 \* @date 2020/2/12  
 \* <p>  
 \* 1. stream 关注的是对数据的运算，与CPU打交道  
 \* 集合关注的是数据存储，与内存打交道  
 \* <p>  
 \* 2.stream的特点  
 \* · stream自己不会存储元素  
 \* · stream不会改变源对象，相反，他会返回一个有特定结果的新stream  
 \* · stream操作是延迟执行的，这意味着它会等到需要结果的时候才执行  
 \* <p>  
 \* 3. stream的执行流程  
 \* · stream的实例化  
 \* · 一系列的中间操作（过滤，映射， 。。。）  
 \* · 终止操作  
 \* <p>  
 \* 4.说明  
 \* · 一个中间操作链，对数据源的数据进行处理  
 \* · 一旦执行终止操作，就执行中间操作链，并产生结果，之后不会再被使用  
 \* <p>  
 \* 5.创建stream  
 \* 5.1 方式一，通过集合创建  
 \*/*public class** StreamApiTest **{  
 *//创建stream方式1：通过集合创建 Collection<E> 接口中的方法* public static void** test() **{** List<User> userList = UserList.*getUserList*();  
 ***//default Stream<E> stream(); 返回一个顺序流*** Stream<User> userStream = userList.stream();  
 ***//default Stream<E> parallelStream(); 返回一个并行流*** Stream<User> parallelStream = userList.parallelStream();  
 **}  
  
 *//创建stream方式2： 通过数组创建* public static void** test2() **{  
 *//基础类型数据的数组创建流* int**[] arr = **new int**[]**{**1, 2, 3, 4**}**;  
 ***//调用Arrays类中的public static <T> Stream<T> stream(T[] array)：返回一个流*** IntStream intStream = Arrays.*stream*(arr);  
  
 ***//自定义的数据类型数组创建流*** User user1 = **new** User(12, **"qqq"**);  
 User user2 = **new** User(13, **"qwe"**);  
 User[] arr1 = **new** User[]**{**user1, user2**}**;  
 Stream<User> userStream = Arrays.*stream*(arr1);  
 **}  
  
 *//创建stream方式3： 通过数组创建Stream的of()方法* public static void** test3() **{** Stream<Integer> integerStream = Stream.*of*(1, 2, 3, 4);  
 **}  
  
 *//创建stream方式4： 创建无线流* public static void** test4() **{  
 *//迭代  
 //public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)  
 //便利前10个偶数*** Stream.*iterate*(0, t -> t + 2).limit(10).forEach(System.***out***::println);  
  
 ***//生成  
 //public static<T> Stream<T> generate(Supplier<T> s)*** Stream.*generate*(Math::*random*).limit(10).forEach(System.***out***::println);  
 **}  
  
 public static void** main(String[] args) **{** *test4*();  
 **}  
}**

## 3 stream的中间操作



### 3.1 筛选与切片

***//1-筛选与切片*public static void** test1() **{  
 *//构建原始数据*** List<User> userList = UserList.*getUserList*();  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*filter\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \*Stream<T> filter(Predicate<? super T> predicate)  
 \* 接收lambda，从流中排除某些元素  
 \* \*/*** Stream<User> userStream = userList.stream();  
 ***//从流中排除年龄小于14的人,及查询年龄大于14的人  
 //.forEach(System.out::println) 这是终止操作，会吧流关闭*** userStream.filter(user -> user.getAge() > 14).forEach(System.***out***::println);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*limit\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* Stream<T> limit(long maxSize);  
 \* 截断流，使其元素不超过给定数量  
 \* \*/*** userList.stream().limit(2).forEach(System.***out***::println);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*skip\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* Stream<T> skip(long n);  
 \* 跳过元素，返回一个扔掉了前n个元素的流，若流中元素不足n个，就返回一个空流  
 \* \*/*** userList.stream().skip(3).forEach(System.***out***::println);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*distinct\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* Stream<T> distinct();  
 \* 筛选，通过流所生成元素的hashCode()和eauqls()去除重复元素  
 \* 需要覆写对象的hashCode()和eauqls()方法  
 \* \*/*** userList.stream().distinct().forEach(System.***out***::println);  
**}**

### 3.2 映射

***//2-映射*public static void** test2() **{** System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*map\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* <R> Stream<R> map(Function<? super T, ? extends R> mapper);  
 \* 接收一个函数作为参数，将元素转换成其他形式或提取信息，  
 \* 该函数函数会被应用到每一个元素上，并将器其映射成一个新元素  
 \* \*/*** Arrays.*asList*(**"a"**, **"cv"**, **"f"**).stream().map(str -> str.toUpperCase())  
 .forEach(System.***out***::println);  
 ***//练习1：获取员工姓名大于3的员工的姓名  
 //从员工列表中获取姓名的stream*** UserList.*getUserList*().stream().map(user -> user.getName())  
 ***//过滤nameStream*** .filter(name -> name.length() > 3)  
 .forEach(System.***out***::println);  
 ***//练习2：*** System.***out***.println(**"\*\*练习2\*\*\*\*"**);  
 Stream<Stream<Character>> streamStream = Arrays.*asList*(**"aa"**, **"cc"**, **"ff"**).stream()  
 .map(StreamApiOmTest::*fromStringToString*);  
 streamStream.forEach(s -> **{** s.forEach(System.***out***::println);  
 **}**);  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*flatMap\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);  
 \* 接收一个函数作为参数，将流中的每一个值都换成一个流，  
 \* 然后把所有的流连成一个流  
 \* 两个list做运算：  
 \* map类似于list1.add(list2);  
 \* flaMap类似于list1.addAll(list2)；  
 \* \*/*** Stream<Character> characterStream = Arrays.*asList*(**"aa"**, **"cc"**, **"ff"**).stream()  
 .flatMap(StreamApiOmTest::*fromStringToString*);  
 characterStream.forEach(System.***out***::println);  
**}  
  
*//将字符串中的多个字符串构成的集合转换为对应的stream实例*public static** Stream<Character> fromStringToString(String string) **{** ArrayList<Character> result = **new** ArrayList<>();  
 **for** (Character c : string.toCharArray()) **{** result.add(c);  
 **}  
 return** result.stream();  
**}**

### 3.3 排序

***//3-排序*public static void** test3() **{** System.***out***.println(**"\*\*\*\*\*sorted自然排序\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* Stream<T> sorted();自然排序  
 \* \*/*** Arrays.*asList*(1, 3, 5, 7, 4, 2).stream().sorted().forEach(System.***out***::println);  
 System.***out***.println(**"\*\*\*\*\*sorted 定制排序\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* Stream<T> sorted(Comparator<? super T> comparator);  
 \* 定制排序  
 \* \*/  
 //按年龄排序*** System.***out***.println(**"\*\*\*\*\*sorted 按年龄排序\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 UserList.*getUserList*().stream()  
 .sorted((u1, u2) -> Integer.*compare*(u1.getAge(), u2.getAge()))  
 .forEach(System.***out***::println);  
 ***//先按年龄排序，年龄相同，按工资排序*** System.***out***.println(**"\*\*\*\*\*sorted 先按年龄排序，年龄相同，按工资排序\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 UserList.*getUserList*().stream()  
 .sorted((u1, u2) -> **{  
 int** boo = Integer.*compare*(u1.getAge(), u2.getAge());  
 **if** (boo != 0) **{  
 return** boo;  
 **} else {  
 return** Double.*compare*(u1.getSalary(), u2.getSalary());  
 **}  
 }**)  
 .forEach(System.***out***::println);  
**}**

## 4 stream的终止操作

### 4.1 匹配与查找

***//1-匹配与查找*public static void** test1() **{  
 *//构建原始数据*** List<User> userList = UserList.*getUserList*();  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*allMatch\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \*boolean allMatch(Predicate<? super T> predicate);  
 \* 检查是否匹配所有元素  
 \* \*/  
 //是否所有员工的年龄都大于18岁* boolean** b = userList.stream().allMatch(user -> user.getAge() > 18);  
 System.***out***.println(b);  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*anyMatch\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \*boolean anyMatch(Predicate<? super T> predicate);  
 \* 检查只要有一个满足条件  
 \* \*/  
 //是否有员工的年龄大于30岁* boolean** b1 = userList.stream().anyMatch(user -> user.getAge() > 30);  
 System.***out***.println(b1);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*noneMatch\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \*boolean noneMatch(Predicate<? super T> predicate);  
 \* 检查是否没有匹配的元素  
 \* \*/  
 //是否存在员工姓雷* boolean** b2 = userList.stream().noneMatch(user -> user.getName().startsWith(**"雷"**));  
 System.***out***.println(b2);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*findFirst\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \*Optional<T> findFirst();  
 \* 查找第一个元素  
 \* \*/  
 //是否存在员工姓雷*** Optional<User> first = userList.stream().findFirst();  
 System.***out***.println(first);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*findAny\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \*Optional<T> findAny();  
 \* 查找任意一个元素  
 \* \*/  
 //是否存在员工姓雷*** Optional<User> any = userList.parallelStream().findAny();  
 System.***out***.println(any);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*count\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* long count();  
 \* 返回流中元素的总个数  
 \* \*/* long** count = userList.parallelStream().filter(user -> user.getSalary() > 100).count();  
 System.***out***.println(count);  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*max\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* Optional<T> max(Comparator<? super T> comparator);  
 \* 返回流中最大值  
 \* \*/  
 //返回最高的工作 流-》工资流-》取最大值*** Optional<Double> max = userList.stream().map(User::getSalary).max(Double::compareTo);  
 Optional<Double> max2 = userList.stream().map(user -> user.getSalary()).max(Double::compareTo);  
 System.***out***.println(max + **" "** + max2);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*min\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* Optional<T> min(Comparator<? super T> comparator);  
 \* 返回流中最小值  
 \* \*/  
 //返回最低工资的员工*** Optional<User> min = userList.stream().min((user1, user2) -> Double.*compare*(user1.getSalary(), user2.getSalary()));  
 Optional<User> min2 = userList.stream().min(Comparator.*comparingDouble*(User::getSalary));  
 System.***out***.println(min + **" "** + min2);  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*forEach\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* void forEach(Consumer<? super T> action);  
 \* 内部迭代  
 \* \*/*** userList.stream().forEach(System.***out***::println);  
  
 ***//使用集合的遍历操作*** userList.forEach(System.***out***::println);  
  
**}**

### 4.2 规约

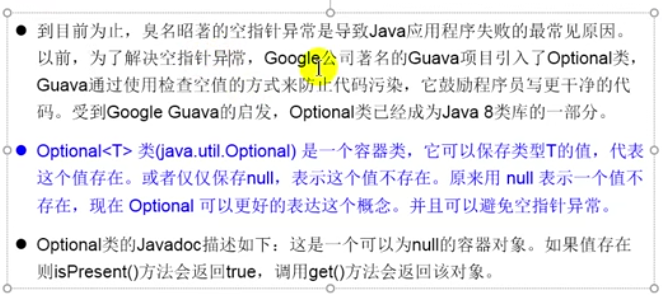
***//2-规约*public static void** test2() **{** System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*reduce\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* T reduce(T identity, BinaryOperator<T> accumulator);  
 \* 可以将流中元素反复结合起来，得到一个值,返回T  
 \* \*/  
 //计算1-10自然数的和*** Integer reduce = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).stream().reduce(0, Integer::*sum*);  
 System.***out***.println(reduce);  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*reduce2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* Optional<T> reduce(BinaryOperator<T> accumulator);  
 \* 可以将流中元素反复结合起来，等到一个值，返回Optional<T>  
 \* \*/  
 //计算公司所有员工工资的综合*** Optional<Double> reduce1 = UserList.*getUserList*().stream().map(User::getSalary).reduce(Double::*sum*);  
 ***//不知道Double中的sum方法，自己求和写法*** Optional<Double> reduce3 = UserList.*getUserList*().stream().map(User::getSalary).reduce((d1, d2) -> d1 + d2);  
 System.***out***.println(reduce1 + **" "** + reduce3);  
**}**

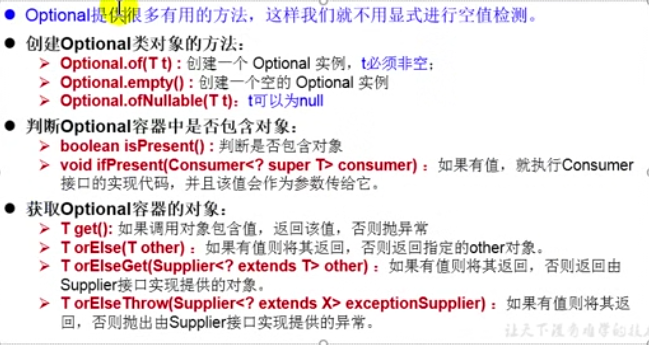
### 4.3 收集

***//3-收集*public static void** test3() **{** System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*collect\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* <R, A> R collect(Collector<? super T, A, R> collector);  
 \* 将流转换成其他形式，接收一个Collector接口实现，用于给stream中元素做汇总  
 \* Collector 接口中方法的实现决定了如何对流执行收集的操作（如收集到List，Set，Map等）  
 \* 另外，Collectors类中提供了很多静态方法，可以方便的创建场景的收集器实例。  
 \* \*/  
 //查找工资大于200的员工，将结果返回成一个List或者Set*** List<User> collect = UserList.*getUserList*().stream().filter(user -> user.getSalary() > 200).collect(Collectors.*toList*());  
 collect.forEach(System.***out***::println);  
  
 Set<User> collect1 = UserList.*getUserList*().stream().filter(user -> user.getSalary() > 200).collect(Collectors.*toSet*());  
 collect1.forEach(System.***out***::println);  
**}**

# 五 Optional类

## 1 optional简介





## 2 简单演示

**package** com.zy.study.java.java8newfeatures.optionaldemo;  
  
**import** java.util.Optional;  
  
***/\*\*  
 \* Optional类：为了在程序中避免出现空指针异常而创建的  
 \* 常用方法：  
 \* of(T value)  
 \* ofNullable(T value)  
 \*  
 \* @autho 18392  
 \* @date 2020/2/12  
 \*/*public class** OptionalTest **{  
  
 public static void** test1() **{** System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*Optional\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 ***/\*  
 \* public static <T> Optional<T> of(T value) 创建一个optional实例，t必须非空  
 \* public static<T> Optional<T> empty() 创建一个optional实例  
 \* public static <T> Optional<T> ofNullable(T value) 创建一个optional实例，t可以为null  
 \* \*/*** Optional<Girl> girl = Optional.*of*(**new** Girl());  
  
 Girl girl1 = **new** Girl();  
 Optional<Girl> optionalgirl = Optional.*ofNullable*(girl1);  
 System.***out***.println(optionalgirl);  
  
 ***//orElse :如果调用该方法的optional中封装的对象是非空的，则返回调用方内部封装的数据  
 //若调用方内部封装的对象为空，则返回orElse方法中的参数*** Girl girl2 = optionalgirl.orElse(**new** Girl(**"wwww"**));  
 System.***out***.println(girl2);  
  
 **}  
  
 *//#################之前写法，容易造成空指针###########* public static** String getGrilName(Boy boy) **{  
 return** boy.getGirl().getName();  
 **}  
  
 public static void** test2() **{** Boy boy = **new** Boy();  
 String grilName = *getGrilName*(boy);  
 System.***out***.println(grilName);  
 **}  
 *//###############################################  
  
 //优化后的写法* public static** String getGrilName1(Boy boy) **{  
 if** (boy != **null**) **{** Girl girl = boy.getGirl();  
 **if** (girl != **null**) **{  
 return** girl.getName();  
 **}  
 }  
 return null**;  
 **}  
  
 public static void** test3() **{** Boy boy = **new** Boy();  
 String grilName = *getGrilName1*(boy);  
 System.***out***.println(grilName);  
 **}  
 *//######################################  
  
 //使用optional操作* public static** String getGrilName2(Boy boy) **{** Optional<Boy> optionalBoy = Optional.*ofNullable*(boy);  
 ***//此时的newboy一定非空*** Boy newboy = optionalBoy.orElse(**new** Boy(**new** Girl(**"eeee"**)));  
 Girl girl = newboy.getGirl();  
 ***//先用optional将girl包起来*** Optional<Girl> optionalGirl1 = Optional.*ofNullable*(girl);  
 ***//此时的girl1一定非空  
 // 通过orElse 让girl1要么是girl的值，要么是orElse方法入参的值*** Girl girl1 = optionalGirl1.orElse(**new** Girl(**"fff"**));  
  
 ***//这样就避免了空指针* return** girl1.getName();  
 **}  
  
 public static void** main(String[] args) **{** *test3*();  
 **}  
}**