Lamda表达式

# Lamda表达式简介

1. 认识一下Lamda表达式的作用；
2. 掌握Lamda表达式的基础语法。

## 具体内容

Lamda是一种基于函数的编程语言，对于这种类型的语言典型的代表就是Haskell。但是从Java最早开始一直提倡的是面向对象编程，也就是说一切的操作必须有类，所有的功能都要定义在类之中。

很多习惯于函数编程的开发者就觉得Java不好用了，于是快在20年之后，Java终于推出了Lamda表达式。

但是需要大家明确一点，并不是说有了Lamda表达式之后才开始的函数编程。最早在Java之中也支持这类编程，那么它的实现模式就是匿名内部类（以接口为主）。

**范例：回顾匿名内部类**

|  |
| --- |
| **interface** Message {  **void** print(String str); } **public class** TestDemo {  **public static void** main(String[] args) {  Message message = **new** Message() {  @Override  **public void** print(String str) {  System.***out***.println(str);  }  };  message.print(**"Hello World !"**);  } } |

实际上写了好久才发现真正需要的只是一个输出语句，但是由于Java之中类的强制的限制，所以很多人就觉得代码过于麻烦了。

**范例：利用Lamda表达式实现**

|  |
| --- |
| **interface** Message {  **void** print(String str); } **public class** TestDemo {  **public static void** main(String[] args) {  Message message = (s) -> System.***out***.println(s);  message.print(**"Hello World !"**);  } } |

首先先不看语法，至少通过本程序可以发现，如果使用Lamda表达式，这个语句少了，而且没有了严格的类结构的控制，感觉确实变了。

那么整个实现的Lamda表达式的语句：

|  |
| --- |
| (s) -> System.***out***.println(s); |

实际上这个语法结构并不难理解，它的理解过程：

|  |
| --- |
| (参数，名称可以随意起)->方法体; |

组成要求如下：

* (参数)：与Message接口定义的print()方法的参数类型一致，但是此处不需要进行声明，如果现在依然对完美追求比较高的用户，那么也可以进行声明。

|  |
| --- |
| Message message = (String s) -> System.***out***.println(s); |

* ->：是一个固定的语法，表示将参数指向方法体；
* 方法体：就是在进行最早匿名内部类实现方法的时候所编写的方法体代码。

在使用Lamda表达式的时候有一个最为重要的定义要求：接口里面的方法只能够存在一个。

## 总结

1. Lamda表达式最重要的目的是解决匿名内部类的问题；
2. 如果想要使用Lamda表达式，那么必须以接口为主，而且接口里面只能够定义一个抽象方法。

# 使用Lamda表达式

1. Lamda表达式语法；
2. 函数接口的注解定义；
3. 在接口里面定义default方法。

## 具体内容

在之前已经清楚了Lamda表达式的基本形式；

|  |
| --- |
| () -> 方法体 |

而在整个Java之中，对于Lamda表达式一共定义有三类语法：

* (params) -> 单行语句;
* (params) -> 表达式;
* (params) -> {多行语句};

**范例：单行语句**

|  |
| --- |
| **interface** Message {  **void** print(); } **public class** TestDemo {  **public static void** main(String[] args) {  Message message = () -> System.***out***.println(**"Hello World !"**);  message.print();  } } |

下面定义一个可以用于加法计算的数学接口，里面可以接收两个数字，而后进行加法计算。

如果在编写Lamda表达式的过程之中，需要返回值时，不需要编写return。

**范例：编写表达式**

|  |
| --- |
| **interface** MyMath {  **int** add(**int** x, **int** y); } **public class** TestDemo {  **public static void** main(String[] args) {  MyMath myMath = (x, y) -> x + y ;  System.***out***.println(myMath.add(10, 20));  } } |

如果说现在要在程序里编写多行代码，那么需要使用“{}”定义了。

**范例：多行语句**

|  |
| --- |
| **interface** MyMath {  **int** add(**int** x, **int** y); } **public class** TestDemo {  **public static void** main(String[] args) {  MyMath myMath = (x, y) -> {  **int** sum = x + y;  **return** sum;  };  System.***out***.println(myMath.add(10, 20));  } } |

一直在强调，如果要想实现Lamda表达式，那么必须有一个前提，这个前提就是接口里面的抽象方法只能够存在一个，所以为了严格这样的语法要求。可以使用函数式的接口定义。

**范例：定义函数接口**

|  |
| --- |
| @FunctionalInterface **interface** MyMath {  **int** add(**int** x, **int** y); } **public class** TestDemo {  **public static void** main(String[] args) {  MyMath myMath = (x, y) -> {  **int** sum = x + y;  **return** sum;  };  System.***out***.println(myMath.add(10, 20));  } } |

如果此时在接口里面定义了更多的方法的时候，那么就会直接在接口上出现错误的信息提示。

但是从JDK1.8开始，接口的定义形式又发生了改变。在JDK1.8之前，所有的接口定义的方法其权限都属于public，可是这个严格的权限要求被“半打破”。接口允许用户进行动态的扩充了。

一个已经实现好的接口，现在又突然要增加新的方法，但是这个增加的方法不希望影响到子类。这种情况下就可以使用default方法。

**范例：观察default方法**

|  |
| --- |
| @FunctionalInterface **interface** Message {  String getInfo();  **default void** print() { *// default方法，此方法不要求子类强制实现* System.***out***.println(**"Hello World!"**);  } } **public class** TestDemo {  **public static void** main(String[] args) {  Message message = () -> **"Hello World !"**;  System.***out***.println(message.getInfo());  message.print();  } } |

对于此代码，实际上已经基本上算是颠覆了传统的开发思路。

所以从JDK1.8开发的过程就出现了变化，可以在接口里面定义default方法，但是这个方法并不需要子类强制性进行实现。同时既然可以有default方法，还可以再进一步的过分一点，在接口里面也可以利用static定义方法。

**范例：在接口里面定义static方法**

|  |
| --- |
| @FunctionalInterface **interface** Message {  String getInfo();  **default void** print() { *// default方法，此方法不要求子类强制实现* System.***out***.println(**"Hello World!"**);  }  **static void** fun() {  System.***out***.println(**"Hello World!"**);  } } **public class** TestDemo {  **public static void** main(String[] args) {  Message message = () -> **"Hello World !"**;  System.***out***.println(message.getInfo());  message.print();  Message.*fun*();  } } |

等于现在所有的已经学习过的关于接口的概念基本上都被推翻了一半，但是这一半的推翻使用的是兼容式的扩充方式，所有用户基本上是感觉不到这样的特征的，除非使用到这类方式。

## 总结

1. Lamda表达式可以有单行语句直接编写，如果是多行语句就使用“{}”定义，如果只是单行返回数据，那么连return也可以省略；
2. 在Lamda表达式代码，它的功能扩充直接影响到接口的功能扩充，可以使用default定义普通方法，以及使用static来定义由接口直接调用的方法，它在整个过程之中并没有对子类进行任何的强制性要求；
3. 所有的Lamda接口都使用函数式接口注解进行声明。

# 方法引用

方法引用是Java 8之后增加的一个新功能，可以理解为复制版的Lamda表达式。

1. 类之中静态方法的引用；
2. 类之中普通方法的引用；
3. 类之中构造方法的引用；
4. 特定类型的任意方法的引用。

## 具体内容

所谓的方法引用实际上指的是将一个特定类的方法功能映射过来，而后通过接口中的方法，利用Lamda表达式实现方法体的定义，当然这种定义的形式一共分为四种：

* 类中静态方法的引用： 类名称 :: 静态方法名称；
* 类中普通方法的引用： 实例化对象名称 :: 普通方法；
* 类之中构造方法的引用： 类名称 :: new
* 特定类型的任意方法的引用： 类名称 :: 方法名称。

下面先来完成一个最简单方法引用操作。

**范例：观察方法引用的形式 —— 引用构造方法**

|  |
| --- |
| @FunctionalInterface **interface** Message {  **void** getInfo(); } **class** Book {  **public** Book() {  System.***out***.println(**"This is Book: 'JAVA 8'"**);  } } **public class** TestDemo {  **public static void** main(String[] args) {  Message msg = Book :: **new** ;  System.***out***.println(msg.getClass());  msg.getInfo(); *// 这个时候调用的是构造方法的方法体* } } |

发现这个时候使用Lamda表达式定义的任何的抽象操作，都可以引用于之结构完全无关的操作，因为要的只是方法体，调用接口的抽象方法时，实际上执行的是引用的方法体内容。

下面依次来看各种的方法引用形式。

**范例：引用静态方法**

在String类里面定义了许多的静态方法：public static String valueOf(int i)，引用此方法。

此时定义的接口就需要设置输入值以及返回值，那么为了让接口更加适合于各种方法，使用泛型定义。

|  |
| --- |
| */\*\*  \* 函数式接口  \** ***@param <P>*** *方法的参数类型  \** ***@param <R>*** *方法的返回值类型  \*/* @FunctionalInterface **interface** Message<P, R> {  R getInfo(P p); } **public class** TestDemo {  **public static void** main(String[] args) {  Message<Integer, String> msg = String :: *valueOf* ;  System.***out***.println(msg.getInfo(10000)); *// 此时返回的是一个字符串* } } |

以上是通过类来引用的静态方法，而实际上用户也可以通过一个类的实例化对象来引用一个普通方法。

**范例：引用普通方法**

在String类里面有一个判断是否以指定的字符串开头：public Boolean startsWith(String prefix)

此方法接收的参数类型为String，而返回值类型为boolean，所以在引用的时候就需要注意泛型类型。

|  |
| --- |
| */\*\*  \* 函数式接口  \** ***@param <P>*** *方法的参数类型  \** ***@param <R>*** *方法的返回值类型  \*/* @FunctionalInterface **interface** Message<P, R> {  R getInfo(P p); } **public class** TestDemo {  **public static void** main(String[] args) {  String str = **"KuhnWei"**;  Message<String, Boolean> msg = str :: startsWith ;  System.***out***.println(msg.getInfo(**"K"**));   } } |

在进行方法引用的时候还可以针对于特定的类型实现方法引用。

范例：特定类型的方法引用

在String类里面有一个不区分内容大小写判断的方法：

public int compareToIgnoreCase(String str)

本次将在Compartor接口上引用这个方法。

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  String [] data = **new** String [] {**"hello"**, **"world"** , **"lamda"**, **"java"**, **"Kuhn"**};  Comparator<String> cmp = String :: compareToIgnoreCase;  Arrays.*sort*(data, cmp);  System.***out***.println(Arrays.*toString*(data));  } } |

那么还有一种就是构造方法的引用了，但是本次引用的构造方法将使用有参构造。

**范例：构造方法的引用**

|  |
| --- |
| **class** Book {  **private** String **title**;  **private double price**;  **public** Book(String title, **double** price) {  **this**.**title** = title;  **this**.**price** = price;  }  @Override  **public** String toString() {  **return "书名："** + **this**.**title** + **"，价格："** + **this**.**price**;  } } @FunctionalInterface **interface** Creator<R **extends** Book> {  R create(String title, **double** price); } **public class** TestDemo {  **public static void** main(String[] args) {  Creator<Book> creator = Book :: **new**;  System.***out***.println(creator.create(**"Java思想编程"**, 98.88));  } } |

等于将构造方法设置了引用，而后利用引用的方法创建了类的对象。

## 总结

对于整个方法的引用，建议掌握静态方法（类：：static方法）、普通方法（实例化对象：：方法）、构造方法（类：：new）的三种引用操作。例如，日后如果见到：“System.out::println”，针对的是System.out.println()方法的引用。

# 内建函数式接口

1. Java提供的核心的内建函数式接口；
2. 集合的新型输出操作。

## 具体内容

从JDK 1.8开始为了方便用户专门提供了一个新的包：java.util.function，在这个包里面针对于用户有可能出现的函数式接口做了一个公共定义。在java.util.function包之中最为核心的是只有四个接口：

* 功能型接口：Function<T, R>
* 消费型接口：Consumer<T>
* 供给型接口：Supplier<T>
* 断言型接口：Predicate<T>

### 功能型接口：Function

观察Function接口的定义

|  |
| --- |
| @FunctionalInterface **public interface** Function<T, R> {  R apply(T t); *// 接收数据而后返回处理的结果* } |

**范例：实现功能性接口的使用**

* 本次引用Integer类的parseInt()方法，这个方法要求接收String型数据，而后返回int型数据

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  Function<String, Integer> fun = Integer :: *parseInt*;  **int** num = fun.apply(**"100"**);  System.***out***.println(num \* 2);  } } |

也就是说这种既能够接收输入数据，又能够返回结果的方法，都使用Function接口定义。

### 消费型接口：Consumer

观察Consumer接口的定义

|  |
| --- |
| @FunctionalInterface **public interface** Consumer<T> {  **void** accept(T t); *// 只是接收数据，并没有返回值存在* } |

**范例：使用消费型接口，引用System.out.println()方法，只接收数据但是没有返回值。**

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  Consumer<String> con = System.***out*** :: println;  con.accept(**"Hello world !"**);  } } |

### 供给型接口：Supplier

观察Supplier接口定义

|  |
| --- |
| @FunctionalInterface **public interface** Supplierr<T> {  T get(); } |

本接口的方法没有参数，但是却可以返回数据。

**范例：设置供给型接口方法引用，本次引用System.currentTimeMillis**

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  Supplier<Long> sup = System :: *currentTimeMillis*;  sup.get();  } } |

### 断言型接口：Predicate

观察接口定义：

|  |
| --- |
| @FunctionalInterface **public interface** Predicate<T> {  **boolean** test(T t); } |

现在是一个判断操作，例如正则验证。

**范例：引用String类中的matches()方法**

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  String str = **"100"**;  Predicate<String> pre = str :: matches;  System.***out***.println(pre.test(**"\\d+"**));  } } |

以上是四个核心接口，实际上这四个接口会了，那么整个java.util.function包之中的接口就明白怎么使了。

例如，随便找一个BiFunction接口，此接口定义如下：

|  |
| --- |
| @FunctionalInterface **public interface** BiFunction<T,U,R> {  R apply(T t, U u); } |

虽然与Function接口不同，但是这里面可以设置两个参数。

**范例：利用BiFunction接口引用一个方法：引用方法String类的**

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  String str = **"Hello"**;  BiFunction<String, String, String> bf = str :: replaceAll;  System.***out***.println(bf.apply(**"e"**, **"\_"**));  } } |

整个包之中的接口的功能都是类似的，实际上四个核心的接口会了，所有的也就都会了。

之所以系统会提供内建的函数式的接口，那么就会在大量的系统类库之中使用它。

在Collection接口里面新定义了一个forEach()方法：

default void forEach(Consumer<? Super T> action)

此方法时一个default方法可以直接利用接口对象调用，同时这个方法里面接收有一个消费型接口。

**范例：List遍历输出**

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"android"**);  list.add(**"ios"**);  list.add(**"python"**);  list.add(**"node.js"**);  list.forEach(s -> System.***out***.println(s));  } } |

在以后学习系统类的时候会大量使用到在java.util.function包之中定义的函数式接口，所以说掌握这四个接口就掌握了整个包的使用。

## 总结

由于内建的函数式的接口提供，所以让开发标准进行了统一，也就是说日后的用户不再需要自己去开发函数式的接口，而直接使用已经提供好的接口进行函数式编程（Lamda编程）。

# 数据流

数据流的基本使用。一定是要结合集合来进行操作。

## 具体内容

类集自从JDK 1.2开始引入之后，一开始给予我们的特征是动态对象数组，后来有了泛型（JDK 1.5），让类集操作更加的安全，再后来到JDK 1.8的时代，类集又变为了重要的批量数据处理的容器，而这个容器的操作实现就是利用数据流完成的。

所以在JDK 1.8版本之后专门提供了一个数据流的工具包：java.util.stream。而在整个包里面最需要关注的是Stream，此接口是BaseStream接口的子接口。

在BaseStream接口里面会包含有：DoubleStream、IntStream,、LongStream、**Stream<T>**。

如果要想关注Stream接口使用，那么首先需要观察Collection接口。在Collection接口里定义了如下一个方法可以取得Stream接口的实例化对象：default Stream<E> stream()。

**范例：操作Stream**

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"android"**);  list.add(**"ios"**);  Stream<String> stream = list.stream();  System.***out***.println(stream.count()); *// 取得数据流的长度* } } |

通过本程序应该清楚一点：Collection集合里面已经支持了Stream接口对象的取得。这一点可以为随后要讲解的集合的数据分析带来帮助。

除了Collection集合可以为Stream接口实例化之外，那么实际上也可以利用Stream接口里面提供的方法完成；

* 取得Stream集合：static <T> Stream<T> of(T… values);
* 支持forEach输出：void forEach(Consumer<? Super T> action)。

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  Stream<String> stream = Stream.*of*(**"java"**, **"c++"**, **"python"**);  stream.forEach(System.***out***::println);  } } |

现在可以利用Collection接口实例化Stream接口，也可以利用Stream接口里面的of()方法，利用static方法取得Stream接口对象，但是折腾半天，感觉它就是一个集合个数取得以及结合的输出。

Java8之后最大的特征是支持了数据的分析操作，所以有了Stream接口对象最大的好处是在于可以进行集合的处理。在函数式接口里面有一个Predicate，这个接口可以负责断言操作。

范例：新的判断模式

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"ios"**);  list.add(**"android"**);  *filter*(list, str -> str.contains(**"a"**));  }  **public static void** filter(List<String> temp, Predicate<String> pre) {  temp.forEach((s) -> {  **if** (pre.test(s)) {  System.***out***.println(**"data = "** + s);  }  });  } } |

感觉上和输出没有任何区别，但是这个时候是属于Java 8的输出模式。

但是现在有人觉得，这种代码看起来很乱，相当于结合了两个函数式接口。

范例：执行集合数据过滤

* 集合过滤操作：public Stream<T> filter(Predicate<? super T> predicate);

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"ios"**);  list.add(**"android"**);  list.stream().filter(x -> x.contains(**"a"**)).forEach(System.***out***::println);  } } |

此时虽然简化了过滤操作，但是感觉到，这种操作还是完全可以利用Iteraotr输出实现。还是觉得没用。

范例：取得过滤后的子集合

在Stream接口里面有一个收集器：<R,A> R collect(Collector<? super T,A,R> collector);

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"ios"**);  list.add(**"android"**);  List<String> subList = list.stream().filter(x -> x.contains(**"a"**)).collect(Collectors.*toList*());  subList.forEach(System.***out***::println);  } } |

除了可以进行数据的判断之外，那么也可以进行数据的逐个处理。

范例：将包含的字符串数据进行小写变大写

* 如果要想针对于每个数据处理：

public <R> Stream<R> map(Function<? Super T, ? extends R> mapper);

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"java"**);  list.add(**"ios"**);  list.add(**"ios"**);  list.add(**"android"**);  list.add(**"android"**);  List<String> subList = list.stream().map((x) -> x.toUpperCase()).collect(Collectors.*toList*());  subList.forEach(System.***out***::println);  } } |

也就是说这个时候可以发现，map()方法可以将每一条数据分别进行处理，而后里面会包含原始的内容。

范例：消除重复数据

* 在Stream接口里面提供了重复数据清除的方法：public Stream<T> distinct()。

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"java"**);  list.add(**"ios"**);  list.add(**"ios"**);  list.add(**"android"**);  list.add(**"android"**);  List<String> subList = list.stream().map((x) -> x.toUpperCase()).distinct()  .collect(Collectors.*toList*());  subList.forEach(System.***out***::println);  } } |

这个时候已经可以成功的消除掉了重复的数据内容。

在Stream接口里面也提供了一些数据的处理方法：

* 判断集合中的全部数据：

public boolean all allMatch(Predicate<? super T> predicate);

* 判断集合中的任意一个数据：

public boolean anyMatch(Predicate<? super T> predicate);

不匹配：public boolean noneMatch(Predicate<? super T> predicate);

范例：集合的全部信息查询

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"ios"**);  list.add(**"android"**);  list.add(**"node.js"**);  list.add(**"python"**);  **if** (list.stream().allMatch((s) -> s.contains(**"a"**))) {  System.***out***.println(**"集合中的全部内容都包含有字母a！"**);  }  } } |

范例：实现部分判断

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  String str = **"a"**;  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"ios"**);  list.add(**"android"**);  list.add(**"node.js"**);  list.add(**"python"**);  **if** (list.stream().anyMatch((s) -> s.contains(str))) {  System.***out***.println(**"集合中的全部内容部分包含有字母a！"**);  }  } } |

在JDK 1.8之前，如果匿名内部类要想访问方法中的参数，则参数前必须加上final关键字，但是从JDK 1.8开始提供的新特性，匿名内部类访问方法参数的时候可以不加上final关键字了。

以上的判断都是全部都是单个条件，如果现在又多个条件呢？如果是判断，则一定使用断言式函数接口：Predicate，在这个接口里面提供有一些链接的操作方法：

* 与操作：default Predicate<T> and(Predicate<? super T> other);
* 或操作：default Predicate<T> or(Predicate<? super T> other);

范例：执行多个判断

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"ios"**);  list.add(**"android"**);  list.add(**"node.js"**);  list.add(**"python"**);  Predicate<String> condA = (str) -> str.contains(**"a"**);  Predicate<String> condB = (str) -> str.length() == 3;  list.stream().filter(condA.or(condB)).forEach(System.***out***::println);  } } |

如果有更多条件，那么就更多的连接or()或and()。

实际上数据流本身支持两类处理方式，一类是并行处理，另一类串行处理（默认），可以利用如下的方法改变处理：

* 设置为并行处理：S parallel();
* 设置为串行处理：S sequential()。

范例：采用并行处理

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"java"**);  list.add(**"ios"**);  list.add(**"android"**);  list.add(**"node.js"**);  list.add(**"python"**);  Predicate<String> condA = (str) -> str.contains(**"a"**);  Predicate<String> condB = (str) -> str.length() == 3;  list.stream().filter(condA.or(condB)).parallel().forEach(System.***out***::println);  } } |

幸运的是在Java之中，数据流的处理里面所有的并发操作完全不需要用户来关心，都可以自己处理。并且最有意思的是可以并行和串行互相切换。

通过以上的讲解，实际上就应该清楚一点：所有的操作像map()、filter()这样的操作都属于中间操作，而像collectors()、forEach()一定都属于结尾操作。

在BaseStream接口里面一共定义了四个子接口，以IntStream为例。

范例：使用IntStream接口

* 生成整型的数据流：static IntStream range(int startInclusive, int endExclusive).

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  IntStream stream = IntStream.*range*(0, 30);  stream.forEach(System.***out***::println);  } } |

Stream接口可以保存各种类型，而IntStream里面只能够保存int型数据。

从JDK 1.8开始一些类里面也增加了改变，例如：java.util.Random类，在这个类里面增加了新的操作方法：

* 返回一个整型的数据流：public IntStream ints();

范例：随机生成10个整型数据流

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  **new** Random().ints().limit(10).forEach(System.***out***::println);  } } |

随着大数据的发展，基本上Java的开发也向大数据靠拢，所有的概念也都是相通的。

## 总结

在JDK 1.8里面，所有的流操作依然还是建立在Lamda表达式基础上的，另外数据提供了批处理的能力，而不是像最早那样，需要利用Iterator接口固定的输出后再处理，可以直接针对于数据处理，或者是返回，或者是直接输出。

# MapReduce开发案例

通过Lamda中的Stream接口实现MapReduce工具，简单理解就时类似于SQL的分组统计工具，只不过MapReduce是一种可以针对于每个数据处理 + 集合的最终统计操作。

## 具体内容

集合不管怎样改变，那么有一点是不会改变 —— 它是一个动态对象数组，所以整个MapReduce操作都围绕着对象完成。

范例：定义一个购物车的类

|  |
| --- |
| **class** Car {  **private** String **pname**;  **private** Integer **amount**;  **private** Double **price**;  **public** Car() {  }  **public** Car(String pname, Integer amount, Double price) {  **this**.**pname** = pname;  **this**.**amount** = amount;  **this**.**price** = price;  }  **public** String getPname() {  **return pname**;  }  **public void** setPname(String pname) {  **this**.**pname** = pname;  }  **public** Integer getAmount() {  **return amount**;  }  **public void** setAmount(Integer amount) {  **this**.**amount** = amount;  }  **public** Double getPrice() {  **return price**;  }   **public void** setPrice(Double price) {  **this**.**price** = price;  } } |

在集合里面会保存有多个Car类的对象。

范例：分别处理

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<Car> all = **new** ArrayList<Car>();  all.add(**new** Car(**"Java编程思想"**, 200, 79.8));  all.add(**new** Car(**"设计模式"**, 120, 109.8));  all.add(**new** Car(**"Oracle从入门到精通"**, 150, 99.8));  all.add(**new** Car(**"Java开发编程"**, 210, 78.8));  all.add(**new** Car(**"Spring实战讲解"**, 190, 88.8));  all.stream().map((myCar) -> {  System.***out***.print(**" 书名："** + myCar.getPname() + **", 购买总价："**);  **return** myCar.getAmount() \* myCar.getPrice();  }).forEachOrdered(System.***out***::println);*// 统计每本书的各自话费* } } |

现在可以发现map()方法的功能就是针对于集合的每个数据进行处理。

如果说使用map()方法实现了数据的重新组合，那么reduce()就是将我们集合中的所有数据变为一个结果，例如：类似SQL中的sum()、avg()、count()函数的功能。

* reduce()方法：public T reduce(T identity, BinaryOperator<T> accumulator).

范例：实现购买商品的总和操作

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<Car> all = **new** ArrayList<Car>();  all.add(**new** Car(**"Java编程思想"**, 200, 79.8));  all.add(**new** Car(**"设计模式"**, 120, 109.8));  all.add(**new** Car(**"Oracle从入门到精通"**, 150, 99.8));  all.add(**new** Car(**"Java开发编程"**, 210, 78.8));  all.add(**new** Car(**"Spring实战讲解"**, 190, 88.8));  **double** result = all.stream().map((myCar) -> { *// System.out.print(" 书名：" + myCar.getPname() + ", 购买总价：");* **return** myCar.getAmount() \* myCar.getPrice();  }).reduce((sum, carPrice) -> sum + carPrice).get();  System.***out***.println(**"购买总金额： "** + result);  } } |

如果要进行统计，可能会包含：总和、最大值、最小值、平均值、数量。

在Stream接口里面提供了相应的操作：

* 处理double数据：

DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);

* 处理int数据：IntStream mapToInt(ToIntFunction<? super T> mapper);
* 处理long数据：LongStream mapToLong(ToLongFunction<? super T> mapper);

在每个返回的接口里面都提供了如下的统计操作方法：

* dobule数据的统计：public DoubleSummaryStatistics summaryStatistics();
* int数据的统计：public IntSummaryStatistics summaryStatistics();
* long数据的统计：public LongSummaryStatistics summaryStatistics();

这些类里面都提供一系列的getXxx()方法用于统计相关信息。

范例：进行reduce的功能

|  |
| --- |
| **public class** TestDemo {  **public static void** main(String[] args) {  List<Car> all = **new** ArrayList<Car>();  all.add(**new** Car(**"Java编程思想"**, 200, 79.8));  all.add(**new** Car(**"设计模式"**, 120, 109.8));  all.add(**new** Car(**"Oracle从入门到精通"**, 150, 99.8));  all.add(**new** Car(**"Java开发编程"**, 210, 78.8));  all.add(**new** Car(**"Spring实战讲解"**, 190, 88.8));  DoubleSummaryStatistics result = all.stream().mapToDouble((myCar) -> {  **return** myCar.getAmount() \* myCar.getPrice();  }).summaryStatistics();  System.***out***.println(**"统计量： "** + result.getCount());  System.***out***.println(**"最大值： "** + result.getMax());  System.***out***.println(**"最小值： "** + result.getMin());  System.***out***.println(**"总 和： "** + result.getSum());  System.***out***.println(**"平均值： "** + result.getAverage());  } } |

整个过程就是在Mongodb里面用的MapReduce的分析方法。

## 总结

1. Stream接口提供的完整的数据流的批处理操作；
2. 前提：Lamda表达式上（函数式接口）。