# 1. Java 8的函数式编程

先以几个简单的例子体验Lambda和Stream API。

准备工作：先新建一个Student类以便案例使用。Student类代码如下（就是一个实体类，有姓名、年龄、性别等属性）：

|  |
| --- |
| **package** java8.demo;  **public class** Student {  **private** String **name**;  **private int age**;  **private** String **gender**;   **public** Student() {  **super**();  }   **public** Student(String name, **int** age, String gender) {  **this**.**name** = name;  **this**.**age** = age;  **this**.**gender** = gender;  }  // getter和setter方法，省略...  @Override  **public** String toString() {  **return "Student{"** +  **"name='"** + **name** + **'\''** +  **", age="** + **age** +  **", gender='"** + **gender** + **'\''** +  **'}'**;  } } |

（1）例1：用TreeSet存储学生集合，利用Comparator实现按年龄从大到小排序学生。

一般，我们使用如下代码：

|  |
| --- |
| **package** java8.demo; **import** org.junit.Test; **public class** Main {  *// 准备的学生集合数据* **private** List<Student> **studentList** = Arrays.*asList*(  **new** Student(**"张三"**, 10, **"男"**),  **new** Student(**"李四"**, 9, **"女"**),  **new** Student(**"王五"**, 11, **"女"**));   @Test  **public void** test() {  *// 比较器。使用匿名内部类（也可自行新建一个类）。* Comparator<Student> comp = **new** Comparator<Student>() {  @Override  **public int** compare(Student o1, Student o2) {  **return** o1.getAge() - o2.getAge();  }  };  *// 使用比较器创建TreeSet，并加入学生集合* TreeSet<Student> treeSet = **new** TreeSet<>(comp);  treeSet.addAll(**studentList**);  *// 遍历输出* **for** (Student student : treeSet) {  System.***out***.println(student);  }  } } |

但是我们发现，实际在Comparator匿名内部类中，真正有效的只有一句话，就是“o1.getAge() - o2.getAge()”，别的只能算是辅助语句。

若我们采用Lambda方式，则代码就变得非常简洁：

*说明，为了节约篇幅和展示重点代码，下面只展示测试方法的代码。方法中能直接使用上面的学生集合数据，studentList成员变量。*

|  |
| --- |
| @Test **public void** testLambda() {  *// 使用lambda代替匿名内部类的写法* Comparator<Student> comp = (stu1, stu2) -> stu1.getAge() - stu2.getAge();  *// 下面还是将学生数据存入treeSet中* TreeSet<Student> treeSet = **new** TreeSet<>(comp);  treeSet.addAll(**studentList**);  *// 遍历时，我们可使用Java 8新提供的API，简洁* treeSet.forEach(System.***out***::println); } |

程序运行，会正确显示和之前一样的结果。我们发现，代码变得简洁了许多，仅用4行代码即完成功能。

（2）例2：在上例数据（即学生集合）的基础上完成：筛选出studentList中年龄大于10岁的学生。

一般方法：

|  |
| --- |
| @Test **public void** testFind() {  *// 待保存符合条件的学生集合* List<Student> newList = **new** ArrayList<>();  *// 遍历、判断再添加到新集合* **for** (Student student : **studentList**) {  **if** (student.getAge() > 10) {  newList.add(student);  }  }  *// 输出新集合* **for** (Student student : newList) {  System.***out***.println(student);  } } |

上述的核心代码仅仅是判断年龄。如果以后再有额外的需求，比如得到所有女同学的集合，那么我们还需要遍历进行性别的判断，如果将过滤同学提取为方法，则需要写多个方法，如过滤年龄的方法，过滤性别的方法。

那有何优雅的解决办法？我们可使用“策略设计模式”，即让结果根据我们提供的策略走，提供什么样的判断策略，就会得到什么样的结果。

首先需要写一个通用的接口，比如叫MyPredicate接口，代码如下：

|  |
| --- |
| **package** java8.demo;  **public interface** MyPredicate<T> {  **boolean** test(T t); } |

然后，在代码中提供“selectList”方法用于筛选，方法的参数是要筛选的集合和MyPredicate实现类对象，该实现类表示会接受哪些元素。代码如下所示：

|  |
| --- |
| *// 通用的筛选方法。* **public** <T> List<T> selectList(List<T> list, MyPredicate<T> mp) {  List<T> newList = **new** ArrayList<>();  **for** (T t : list) {  **if** (mp.test(t)) {  newList.add(t);  }  }  **return** newList; } @Test **public void** testFind() {  *// 编写predicate。* MyPredicate<Student> stuPredicate = **new** MyPredicate<Student>() {  @Override  **public boolean** test(Student student) {  **return** student.getAge() > 10;  }  };  *// 利用predicate直接得到结果。* List<Student> students = selectList(**studentList**, stuPredicate);  students.forEach(System.***out***::println); } |

上述就是“策略设计模式”，提取的方法灵活性提高了，但代码不直观了，并且并不是很简洁。这时我们可再用Lambda表达式简化代码：

|  |
| --- |
| @Test **public void** testFind() {  *// 用lambda编写predicate。* MyPredicate<Student> stuPredicate = (student) -> student.getAge() > 10;  *// 得到结果。* List<Student> students = selectList(**studentList**, stuPredicate);  students.forEach(System.***out***::println); } |

实际上，利用Java 8的Stream API，可使代码更加简洁！我们无需编写上述的MyPredicate接口和selectList方法，下面直接在测试方法中使用Stream API和lambda一步到位达到效果：

|  |
| --- |
| @Test **public void** testFind() {  **studentList**.stream()  .filter((stu) -> stu.getAge() > 10)  .forEach(System.***out***::println); } |

只要上述一条语句，实现了需求，无需额外代码。我们还能限制只要女同学中的前1个：

|  |
| --- |
| @Test **public void** testFind() {  **studentList**.stream()  .filter((stu) -> stu.getGender().equals(**"女"**))  .limit(1)  .forEach(System.***out***::println); } |

甚至能只显示集合中的学生姓名：

|  |
| --- |
| @Test **public void** testFind() {  **studentList**.stream()  .map(Student::getName)  .forEach(System.***out***::println); } |

So，Let’s Study。

# 2. Lambda表达式

## 2.1 什么是Lambda表达式

Lambda 是一个匿名函数，可以把 Lambda表达式理解为是一段可以传递的代码（将代码像数据一样进行传递）。使用Lambda可以写出更简洁、更灵活的代码。

Lambda 使用“->”作为操作符，该操作符被称为 Lambda 操作符或箭头操作符。它将 Lambda分为两个部分：

左侧：指定Lambda表达式需要的参数；

右侧：指定Lambda 体，即Lambda表达式要执行的功能。

## 2.2 使用Lambda表达式

Lambda表达式用来简化匿名内部类的书写，如之前的简化Comparator接口。因此Lambda 表达式需要“函数式接口”的支持。

函数式接口是指只有一个抽象方法的接口，比如下面示例中要使用的Runnable接口等。我们可以在函数式接口上使用“@FunctionalInterface”注解，该注解表明该接口是一个函数式接口，并且注解会检查该接口是否符合函数式接口。

由于函数式接口中抽象方法参数和返回值不同，Lambda表达式有以下几种格式：

（1）无参数，无返回值的Lambda。格式是：() -> {要执行的Lambda体}。

这里以Runnable接口为例：

|  |
| --- |
| @Test **public void** test() {  Runnable r = () -> {  System.***out***.println(**"Running"**);  System.***out***.println(**"Finish"**);  };  r.run(); } |

如果Lambda体中只有一条语句，那么Lambda体的大括号是可以省略的，即：

|  |
| --- |
| Runnable r = () -> System.***out***.println(**"Finish"**); |

（2）有参数，有（或者无）返回值。格式是：(参数列表) -> {Lambda体}。

参数列表中，无需指定参数的类型，JVM会根据上下文推断，这就是“类型推断”。若有返回值，则使用return语句即可。

|  |
| --- |
| @Test **public void** test() {  Comparator<Integer> comparator = (x, y) -> {  System.***out***.println(**"comp"**);  **return** x - y;  };  Integer i = comparator.compare(20, 10);  System.***out***.println(i); *// 输出10* } |

如果Lambda体只有一条语句且是return语句，则Lambda体的大括号和return都可省略不写（要省都省。当然，若Lambda体本就无返回值且只有一条语句，还是直接省略大括号）：

|  |
| --- |
| Comparator<Integer> comparator = (x, y) -> x - y; |

另外，如果参数只有一个，则参数列表的小括号可以省略，如（这里使用一下下面要讲的Consumer）：

|  |
| --- |
| @Test **public void** test() {  Consumer consumer = (x) -> System.***out***.println(x);  consumer.accept(**"city"**); } |

由上我们看到，Lambda表达式左侧就是接口方法的参数，右侧就是接口方法具体需要执行的操作。Lambda体中可以对方法传入的值进行操作和返回（如果方法有参数和返回值的话）。

## 2.3 自定义函数式接口

实际上我们在1.2节已经自定义了一个函数式接口MyPredicate，现在，可以为它加上@FunctionalInterface注解了。该函数式接口的作用是“判定”。

经常会把Lambda作为参数传递，以灵活实现需要的功能。比如在1.2节中，selectList方法会根据传递进来的Lambda判定符合条件的对象，从而返回正确的结果。

我们可以定义多个函数式接口来实现不同的功能，实际这些接口很简单，只不过是参数和返回值不同。再举例：我们想用Lambda实现：将传入的字符串变为大写的：

（1）编写MyFunction函数式接口，接受一个待处理的字符串，并且有返回值以便返回结果：

|  |
| --- |
| **package** java8.demo; @FunctionalInterface **public interface** MyFunction {  String process(String str); } |

（2）编写“处理”方法并调用。下面的processStr方法就接收了字符串和MyFunction参数。

|  |
| --- |
| **package** java8.demo; **import** org.junit.Test; **public class** Main {  *// 处理字符串。接收待处理字符串和MyFunction* **public** String processStr(String str, MyFunction mf) {  **return** mf.process(str);  }  *// 测试方法* @Test  **public void** test() {  String upper = processStr(**"hello"**, (str) -> str.toUpperCase());  System.***out***.println(upper); *// 输出HELLO* } } |

实际利用processStr()方法不仅能将字符串转为大写，这是传入的Lambda表达式决定的。并且，我们需要的函数式接口其实很有限，就是特定的参数和返回值。实际上Java 8内置了许多常用的函数式接口，因此我们无需自己定义。下面讲。

## 2.4 Java内置的函数式接口

Java 8内置四大核心函数式接口，分别是：

（1）Consumer<T>：消费型接口。该接口中的方法是“void accept(T t)”，接收一个对象且无返回值，适合对数据进行操作。例子：

|  |
| --- |
| **package** java8.demo; **import** org.junit.Test;  **import** java.util.function.Consumer;  **public class** Main {  *// “消费"的方法，处理数据* **public void** consumer(Double money, Consumer<Double> consumer) {  consumer.accept(money);  }  *// 测试方法* @Test  **public void** test() {  consumer(500D, (x) -> System.***out***.println(**"月消费"** + x));  *// 会输出“月消费500.0”* } } |

注意的是，Lambda中和匿名内部类中一样，同样不能修改外部的局部变量：

|  |
| --- |
| @Test **public void** test() {  **int** variable = 10;  *// 下面修改variable是错误的* consumer(500D, (x) -> System.***out***.println(**"月消费"** + variable++)); } |

（2）Supplier<T>：供给型接口。该接口中的方法是“T get()”，无参数并能返回一个对象，适合获取数据。例如：

|  |
| --- |
| **package** java8.demo; **import** org.junit.Test;  **public class** Main {  *// “获取"n个数的方法* **public** List<Integer> getRandom(**int** n, Supplier<Integer> sp) {  List<Integer> integerList = **new** ArrayList<>();  **for** (**int** i = 0; i < n; i++) {  integerList.add(sp.get());  }  **return** integerList;  }  *// 测试方法* @Test  **public void** test() {  *// 获取0-100随机数* List<Integer> list = getRandom(10, () -> (**int**)(Math.*random*() \* 100));  System.***out***.println(list);  } } |

（3）Function<T, R>：函数型接口。该接口中的方法是“R apply(T t)”，接受T类型参数，可返回R类型。适合根据某数据获得结果集等情况。例如：

|  |
| --- |
| **package** java8.demo; **import** org.junit.Test; **import** java.util.function.Function;  **public class** Main {  *// 根据字符串获得数组的方法* **public** String[] getArrayString(String str, Function<String, String[]> func) {  **return** func.apply(str);  }  *// 测试方法* @Test  **public void** test() {  *// 分割字符串* String[] strArr = getArrayString(**"hello-world-java"**,  (str) -> str.split(**"-"**)  );  *// 输出strArr* **for** (String str : strArr) {  System.***out***.println(str);  }  } } |

（4）Predicate<T>：断言型接口。该接口中的方法是“boolean test(T t)”，接受T类型参数，返回布尔类型。适合选择条件的情况。例如：

|  |
| --- |
| **package** java8.demo; **import** org.junit.Test;  **public class** Main {  *// 根据条件得到合适的字符串集合* **public** List<String> conditionArrayString(String[] strArr, Predicate<String> func) {  List<String> stringList = **new** ArrayList<>();  **for** (String str : strArr) {  **if** (func.test(str)) {  stringList.add(str);  }  }  **return** stringList;  }  *// 测试方法* @Test  **public void** test() {  String[] strArr = {**"hello"**, **"abc"**, **"yellow"**};  *// 获得字符串长度大于3的字符串集合* List<String> resultList = conditionArrayString(strArr,  (str) -> str.length() > 3);  *// 输出strArr* **for** (String str : resultList) {  System.***out***.println(str);  }  } } |

从以上例子我们看出：函数式接口用于定义Lambda表达式的格式、我们写的“getArrayString”等方法定义通用的“数据传入传出格式”、而调用这些方法时用户写的Lambda表达式才是真正具体的指令。

除了上述内置的四大核心函数式接口，Java还提供了以下函数式接口以便使用：

|  |  |
| --- | --- |
| 函数式接口 | 提供的方法 |
| BiFunction<T, U, R> | R apply(T t, U u) |
| UnaryOperator<T> | T apply(T t) |
| BinaryOperator<T> | T apply(T t1, T t2) |
| BiConsumer<T, U> | void accept(T t, U u) |
| ToIntFunction<T>  ToLongFunction<T>  ToDoubleFunction<T> | int applyAsInt(T value)  long applyAsLong(T value)  double applyAsDouble(T value) |
| IntFunction<R>  LongFunction<R>  DoubleFunction<R> | R apply(int value)  R apply(long value)  R apply(double value) |

## 2.5 方法引用和构造器引用

（1）方法引用

若Lambda体中的功能，已经有方法提供了实现，则整个Lambda表达式可使用方法引用。方法引用使用“::”操作符将所引用方法与类或者对象分隔开来。可以将方法引用理解为Lambda表达式的另外一种形式。

方法引用有如下三种形式：

1. 对象::实例方法名

2. 类名::静态方法名

3. 类名::实例方法名

要注意，方法引用所引用的方法的参数列表与返回值类型，需要与函数式接口中抽象方法的参数列表和返回值类型一致。

另外，只有当Lambda的参数列表的第一个参数，是实例方法的调用者，且第二个参数(或无参)是实例方法的参数时，才能使用“类名::实例方法名”的形式。

举例说明：

|  |
| --- |
| **package** java8.demo; **import** org.junit.Test;  **public class** Main {  *// 1. 对象::实例方法名形式* @Test  **public void** test1() {  *// 没用方法引用前，是这样的形式：* Consumer<String> consumer = (x) -> System.***out***.println(x);  consumer.accept(**"hello"**);  *// 由于Consumer中抽象方法原型和println(xxx)原型一致，因此可进行方法引用：* Consumer<String> consumerNew = System.***out***::println;  consumerNew.accept(**"world"**);  *// 同样，我们还可利用Supplier和Student* Supplier<String> func = **new** Student()::getName;  System.***out***.println(func.get()); *// 输出null，因为无赋值* }  *// 2. 类::静态方法名形式* @Test  **public void** test2() {  *// 没用方法引用前：* Comparator<Integer> comp = (x, y) -> Integer.*compare*(x, y);  *// 使用方法引用后：* Comparator<Integer> compNew = Integer::*compare*;  *// 因为Integer.compare方法和Comparator的抽象方法原型是一样的。  // 再比如计算两个数的最大值* BiFunction<Integer, Integer, Integer> max = Math::*max*;  System.***out***.println(max.apply(32, 16)); *// 输出32* }  *// 3. 类::实例方法名形式* @Test  **public void** test3() {  *// 需求：判断两字符串是否相等。可用：* BiPredicate<String, String> bp = (x, y) -> x.equals(y);  *// 由于上述Lambda符合：x是调用方，且y是实例方法中的参数。因此可写为：* BiPredicate<String, String> bpNew = String::equals;  System.***out***.println(bpNew.test(**"hello"**, **"hello"**)); *// 输出true* } } |

（2）构造器引用

若Lambda体中是用构造方法新建对象，那么就可用构造器引用，格式是“类名::new”。要求是：类中要有一个构造器的参数列表与函数式接口中抽象方法的参数列表一致。

例子：

|  |
| --- |
| **package** java8.demo; **import** org.junit.Test;  **import** java.util.function.\*;  **public class** Main {  @Test  **public void** test() {  *// 原本* Supplier<Student> sup = () -> **new** Student();  *// 可改为* Supplier<Student> supNew = Student::**new**;  } } |

（3）数组引用

若Lambda体中是用新建数组对象，那么就能使用数组引用，格式是“数组类型[]::new”。

例子：

|  |
| --- |
| **package** java8.demo; **import** org.junit.Test;  **import** java.util.function.\*;  **public class** Main {  @Test  **public void** test() {  *// 需求：根据传入的参数创建数组。原本方式* Function<Integer, String[]> sup = (args) -> **new** String[args];  String[] strArr = sup.apply(10);  *// 现在* Function<Integer, String[]> supNew = String[]::**new**;  String[] strArrNew = supNew.apply(10);  System.***out***.println(strArrNew.**length**); *// 输出为10* } } |

# 3. 强大的Stream API

## 3.1 什么是Stream

Stream是Java 8中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行复杂的查找、过滤和映射数据等操作。

使用Stream API 对集合数据进行操作，就类似于使用SQL查询数据库数据。也可以使用Stream API来并行执行操作。简而言之，Stream API 提供了一种高效且易于使用的处理数据的方式。

流(Stream)即是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。“集合讲的是数据，流讲的是计算！”通过流进行一系列的操作后，会得到新的流，新的流就包含了新的结果集，而原本流中的数据是不变的。

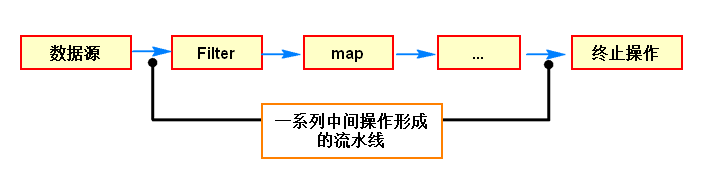
对Stream的操作有以下三个步骤：

（1）创建Stream：从一个数据源，获取一个流。

（2）中间操作：一个中间操作链，对数据源的数据进行处理。

（3）终止操作：一个终止操作，执行中间操作链，并产生结果。

图解：



## 3.2 创建Stream

有四种获取Stream流的方式：

（1）获取集合的Stream。Java 8的Collection接口中扩展了两个实例方法用于获取流，分别是stream()和parallelStream()。stream()方法返回一个串行流，而parallelStream()返回一个并行流。例：

|  |
| --- |
| @Test **public void** test() {  List<String> list = **new** ArrayList<>();  *// 获得List的串行流* Stream<String> stream = list.stream();  *// 获得list的并行流* Stream<String> parallelStream = list.parallelStream(); } |

（2）获取数组的Stream。Java 8中，使用Arrays的静态方法stream()获取数组流。Arrays.stream()有多种重载形式，可以获得不同数据类型数据的stream。例：

|  |
| --- |
| @Test **public void** test() {  Integer[] intArr = **new** Integer[3];  Stream<Integer> stream = Arrays.*stream*(intArr); } |

（3）使用静态方法Stream.of(T... values)，通过显式的值创建stream流。例：

|  |
| --- |
| @Test **public void** test() {  Stream<String> stream = Stream.*of*(**"hello"**, **"world"**); } |

（4）使用静态方法 Stream.iterate()和Stream.generate()创建无限流。

“迭代”iterate()的原型：

public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)，参数1是种子seed（即产生数据的初始值），参数2是函数式接口实现，指明如何迭代产生数据。

“生成”generate()的原型：

public static<T> Stream<T> generate(Supplier<T> s)，参数中传递函数式接口Supplier的实现，表明如何生成数据。

下面是使用上述两个方法产生无限流的例子：

|  |
| --- |
| **package** java8.demo; **import** org.junit.Test; **import** java.util.stream.Stream;  **public class** Main {  *// 创建无限流  // 1. 采用“迭代”方式* @Test  **public void** test1() {  Stream<Integer> streamIterate = Stream.*iterate*(0, (seed) -> seed + 2);  *// 为了知道流中是何数据，可用forEach方法“终止”流操作，并查看* streamIterate.forEach(System.***out***::println);  *// 以上会连续不断地从0开始输出所有偶数。我们可以用limit限制输出的个数（注意先注释掉上面的语句）：* streamIterate.limit(20).forEach(System.***out***::println);  }  *// 2. 采用“生成”方式* @Test  **public void** test2() {  Stream<Double> streamGenerate = Stream.*generate*(Math::*random*);  streamGenerate.limit(10).forEach(System.***out***::println);  *// 上述执行输出了10个随机小数。* } } |

## 3.3 Stream的中间操作

### 3.3.1 筛选与切片

（1）filter(Predicate p)方法：接收Lambda，从流中筛选某些元素。例：

|  |
| --- |
| **package** java8.demo; **public class** Main {  **private** List<Student> **studentList** = Arrays.*asList*(  **new** Student(**"张三"**, 12, **"男"**),  **new** Student(**"李四"**, 11, **"女"**),  **new** Student(**"王五"**, 9, **"女"**)  );  @Test  **public void** test() {  *// 筛选出大于10岁的同学  // filter操作后会返回新的流。* Stream<Student> stream = **studentList**.stream()  .filter((stu) -> stu.getAge() > 10);  *// 用终止操作forEach输出新的流。流中数据就是筛选后的数据* stream.forEach(System.***out***::println);  } } |

说明：下述示例将直接使用studentList集合，不再展示代码全貌。

在Stream中，可有多个中间操作，这些中间操作形成一个流水线（如筛选、映射等）。如果没有Stream的终止操作，这些中间操作是不会执行的，中间操作是在终止操作时一次性全部处理，这称为“惰性求值”。

为了证实上述说明，现在我们编写代码：

|  |
| --- |
| @Test **public void** test() {  Stream<Student> stream = **studentList**.stream()  .filter((stu) -> {  System.***out***.println(**"进行筛选..."**);  **return** stu.getAge() > 10;  });  *// stream.forEach(System.out::println);* } |

上述代码在Lambda中写了一条输出语句，但是执行程序并无任何输出，说明中间操作没有执行。这是因为上述代码中，stream无终止操作，我们注释掉了forEach终止操作。如果取消注释forEach语句，那么就会正常输出。

（2）limit(long maxSize)：截断流，使流中元素不超过给定数量。例：

|  |
| --- |
| @Test **public void** test() {  **studentList**.stream()  .limit(2)  .forEach(System.***out***::println); } |

上述会输出前两个学生数据。可使用多个中间操作。执行limit时是“短路”操作，即一旦满足了数量，就不再进行中间操作了，以提高效率。

|  |
| --- |
| @Test **public void** test() {  **studentList**.stream()  .filter((stu) -> stu.getAge() > 10)  .limit(2)  .forEach(System.***out***::println); } |

（3）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，则返回一个空流。与 limit(n) 互补。这里不做示例了。

（4）distinct()：筛选，通过流所生成元素的hashCode()和equals()去除重复元素，也就是说，如果你想去除重复的对象元素，则需要写好hashCode()和equals()方法。

### 3.3.2 映射

（1）map()：接收Lambda参数，将元素转换成其他形式或提取信息。该方法接收的Lambda函数会被应用到每个元素上，并将其映射成一个新的元素，最终返回一个新流（新流的泛型类型就是Lambda返回类型）。例：

|  |
| --- |
| @Test **public void** test() {  List<String> strList = Arrays.*asList*(**"hello"**, **"world"**);  *// 1. 利用map将strList中值转换成大写输出。因为map会作用于每个元素* strList.stream()  .map(String::toUpperCase)  .forEach(System.***out***::println);  *// 2. 利用map提取指定信息* **studentList**.stream()  .map(Student::getName)  .forEach(System.***out***::println); *// 这里只输出姓名信息* } |

（2）flatMap()：接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。比如我们在本主类Main中定义了一个静态方法：

|  |
| --- |
| *// 将字符串转成字符数组并返回流* **public static** Stream<Character> filterCharacter(String str) {  List<Character> list = **new** ArrayList<>();  **for**(Character ch : str.toCharArray()) {  list.add(ch);  }  **return** list.stream(); } |

这时，若想一字一字地输出“h e l l o w o r l d”，则需要在forEach中再执行forEach，因为map返回的值是Stream<Stream<Character>>：

|  |
| --- |
| @Test **public void** test() {  List<String> strList = Arrays.*asList*(**"hello"**, **"world"**);  Stream<Stream<Character>> stream = strList.stream().map(Main::*filterCharacter*);  stream.forEach((sm) -> {  sm.forEach(System.***out***::println);  }); } |

如果用flatMap，则直接会将各个Stream<Character>返回成一个Stream<Character>流，这样就比较方便了，代码为：

|  |
| --- |
| @Test **public void** test() {  List<String> strList = Arrays.*asList*(**"hello"**, **"world"**);  Stream<Character> stream = strList.stream().flatMap(Main::*filterCharacter*);  stream.forEach(System.***out***::println); } |

### 3.3.3 排序

使用sorted()方法进行排序。sort有两个重载，分别用于自然排序和比较器排序。

（1）sorted()：自然排序（Comparable）。返回排序后的新流。例：

|  |
| --- |
| @Test **public void** test() {  List<String> strList = Arrays.*asList*(**"cc"**, **"aa"**, **"bb"**);  strList.stream()  .sorted()  .forEach(System.***out***::println);  *// 输出结果是aa bb cc* } |

（2）sorted(Comparator<? super T> comparator)：按照传入的比较器排序，返回排序后的新流。例：

|  |
| --- |
| @Test **public void** test() {  **studentList**.stream()  .sorted((stu1, stu2) -> stu1.getAge() - stu2.getAge())  .forEach(System.***out***::println);  *// 结果是按照年龄排序的* } |

## 3.4 Stream的终止操作

终止操作会从流中生成结果。其结果可以是任何不是流的值，如List、boolean、void等。

### 3.4.1 查找与匹配

（1）boolean allMatch(Predicate<T> p)：检查所有元素是否都匹配断言predicate。如果都满足则返回true，否则返回false。例：

|  |
| --- |
| @Test **public void** test() {  **boolean** flag = **studentList**.stream()  .allMatch((stu) -> stu.getAge() > 10);  System.***out***.println(flag);  // 不是所有同学年龄都大于10，因此输出false } |

（2）boolean anyMatch(Predicate<T> p)：只要有任意一个元素满足断言，就会返回true。如果上述代码改用anyMatch，则返回值为true。

（3）boolean noneMatch(Predicate<T> p)：检查是否所有元素都不满足断言，如果是，则返回true。

（4）Optional<T> findFirst()：返回流中的第一个元素，返回的元素被封装在Optional对象中（后面我们将指导Optional可用来避免空指针异常）。当然，可以进行一些中间操作来获取第一个元素（或者直接没有中间操作）。例：

|  |
| --- |
| @Test **public void** test() {  Optional<Student> optStu = **studentList**.stream()  .filter((stu) -> stu.getAge() > 10)  .findFirst();  System.***out***.println(optStu.get()); *// 用get方法获得Optional中数据* } |

（5）Optional<T> findAny()：返回流中任意一个元素。下面的代码中使用并行流parallelStream来演示，多线程获取元素，哪个先拿到就先返回，避免每次演示的时候都直接返回第一个元素：

|  |
| --- |
| @Test **public void** test() {  Optional<Student> optStu = **studentList**.parallelStream()  .findAny();  System.***out***.println(optStu.get()); } |

（6）long count()：返回流中元素总数。例：

|  |
| --- |
| @Test **public void** test() {  **long** count = **studentList**.stream()  .count();  System.***out***.println(count); } |

（7）Optional<T> max(Comparator<> comp)和Optional<T> min(Comparator<T> comp)：分别返回流中的最大值和最小值。函数中需要传入比较的方法，例：

|  |
| --- |
| @Test **public void** test() {  Optional<Student> opt = **studentList**.stream()  .max((x, y) -> x.getAge() - y.getAge());  *// 会获得年龄最大的人。因为比较器就是根据年龄算出最大值* System.***out***.println(opt.get()); } |

（8）void forEach(Consumer<T> action)：此方法很常用，用于内部迭代集合。

注意：流一旦执行了终止操作，就不能再使用了。

### 3.4.2 归约与收集

（1）归约，使用reduce方法，reduce用于将流中的元素反复结合起来，最终得到一个值并返回。有下面两种重载：

T reduce(T identity, BinaryOperator<T> accumulator)：参数1相当于是初始结合的值，参数2是二元操作的函数式，用于指定两个元素的结合方式。

Optional<T> reduce(BinaryOperator<T> accumulator)：这是第二种重载，只有一个参数，返回值包含在Optional中，防止为空。

例1：计算1,2,3,4,5的总和。

|  |
| --- |
| @Test **public void** test() {  **int** sum = Stream.*of*(1, 2, 3, 4, 5)  .reduce(0, (x, y) -> x + y);  *// 上述也可写成reduce(Integer::sum)，因为Integer中静态方法sum也用于执行两数相加* System.***out***.println(sum); } |

参数1相当于初始值，执行流程如下：第一次，x是初始的0，y是流中的第一个元素1，他们执行相加算法得到1。第二次，x就为上次的总和1，而y是流中的第二个元素2，再执行相加得到3。第三次，x为上次的总和3...如此依次执行，最终得到1-5的总和为15。这就是reduce归约。

下面用一个参数的重载来计算学生年龄的总和。由于该重载无identity，返回值可能为空，因此返回值类型用Optional。

|  |
| --- |
| @Test **public void** test() {  Optional<Integer> sum = **studentList**.stream()  .map(Student::getAge)  .reduce(Integer::*sum*);  System.***out***.println(sum.get()); } |

map和reduce的连接通常称为 map-reduce 模式，因 Google 用它来进行网络搜索而出名。

（2）收集，使用collect方法。可用collect方法收集流中各种各样的信息。一般使用的collect重载为：

<R, A> R collect(Collector<? super T, A, R> collector)，接收一个Collector接口的实现，用于汇总收集。一般我们不需要写collector的实现，因为Java 8提供了Collectors类，其中提供了很多Collector的实现类，我们只要调用方法即可。Collectors类提供了大部分实用的方法，方便了开发者。下面举例说明：

1. 利用Collectors.toList()等方法将流收集成需要的集合：

|  |
| --- |
| @Test **public void** test() {  *// 例1：将学生的年龄收集成list集合* List<String> stuNameList = **studentList**.stream()  .map(Student::getName)  .collect(Collectors.*toList*());  System.***out***.println(stuNameList);  *// 还能转成Set集合* Set<String> stuNameSet = **studentList**.stream()  .map(Student::getName)  .collect(Collectors.*toSet*());  stuNameSet.forEach(System.***out***::println);  *// 若是一些“复杂”的集合，则用toCollection。比如LinkedHashSet* LinkedHashSet<String> stuLinkedHashSet = **studentList**.stream()  .map(Student::getName)  .collect(Collectors.*toCollection*(LinkedHashSet::**new**));  stuLinkedHashSet.forEach(System.***out***::println); } |

2. 使用Collectors.maxBy()等“聚合函数”统计信息：

|  |
| --- |
| @Test **public void** test() {  *// 1. 统计学生中的最大年龄(当然也可用之前的其他方式)* Optional<Integer> maxAge = *studentList*.stream()  .map(Student::getAge)  .collect(Collectors.*maxBy*(Integer::*compare*));  System.***out***.println(maxAge.get());  *// 2. 找出最小年龄的学生* Optional<Student> minAge = *studentList*.stream()  .collect(Collectors.*minBy*((x, y) -> x.getAge() - y.getAge()));  System.***out***.println(minAge.get());  *// 3. 统计有多少学生* **long** count = *studentList*.stream()  .collect(Collectors.*counting*()); *// 当然也可直接用stream.count()* System.***out***.println(count);  *// 4. 统计学生年龄总和* **int** sumAge = *studentList*.stream()  .collect(Collectors.*summingInt*(Student::getAge));  *// 上述中summingInt中本来就需要传递mapper，因此事先不用调用mapper方法单独拿到年龄  // 除了summingInt方法外，还有summingLong和summingDouble，根据数据类型需要来使用* System.***out***.println(sumAge);  *// 5. 计算学生的平均年龄* **int** aveAge = *studentList*.stream()  .collect(Collectors.*averagingInt*(Student::getAge));  System.***out***.println(aveAge); } |

上述使用了Collectors中提供的不同方法实现了类似于SQL中的“聚合函数”功能，这也是函数式编程的体现。

我们也可以直接用Collectors.summarizingInt()等方法，使collect方法返回IntSummaryStatistics对象，IntSummaryStatistics对象中一次性就提供了上述几种方法。例：

|  |
| --- |
| @Test **public void** test() {  IntSummaryStatistics iss = *studentList*.stream()  .collect(Collectors.*summarizingInt*(Student::getAge));  System.***out***.println(iss.getMax()); *// 最大* System.***out***.println(iss.getMin()); *// 最小* System.***out***.println(iss.getCount()); *// 个数* System.***out***.println(iss.getSum()); *// 总和* System.***out***.println(iss.getAverage()); *// 平均* } |

当然，还有summarizingFloat、summarizingLong方法，返回的就是对应类型的对象了，按需使用。

3. 使用“Collectors.groupingBy()”方法进行分组。例：

|  |
| --- |
| @Test **public void** test() {  Map<String, List<Student>> map = *studentList*.stream()  .collect(Collectors.*groupingBy*(Student::getGender));  System.***out***.println(map); } |

这样是通过性别进行分组，返回的结果是Map类型，其中键类型是“性别”的字符串类型，值是Student的List类型。输出结果是：

|  |
| --- |
| {女=[Student{name='李四', age=11, gender='女'}, Student{name='王五', age=9, gender='女'}], 男=[Student{name='张三', age=12, gender='男'}]} |

由于Collectors.groupingBy()还有重载为：

Collector<T, ?, Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier, Collector<? super T, A, D> downstream)

即参数2还能传递Collectors，因此可以进行多级分组。下面例子是二级分组（先按照性别分组，再按照年龄）：

|  |
| --- |
| @Test **public void** test() {  *// 返回的结果类型是Map的值又是一个Map。* Map<String, Map<Integer, List<Student>>> map = *studentList*.stream()  .collect(Collectors.*groupingBy*(Student::getGender, Collectors.*groupingBy*(Student::getAge)));  System.***out***.println(map); } |

输出结果为：

|  |
| --- |
| {女={李四=[Student{name='李四', age=11, gender='女'}], 王五=[Student{name='王五', age=9, gender='女'}]}, 男={张三=[Student{name='张三', age=12, gender='男'}]}} |

4. 利用Collectors.join()方法给连接字符串。例：

|  |
| --- |
| @Test **public void** test() {  *// 返回的结果类型是Map的值又是一个Map。* String str = *studentList*.stream()  .map(Student::getName)  .collect(Collectors.*joining*(**"，"**, **"["**, **"]"**));  *// joining的参数1是delemiter，参数2是prefix，参数3是suffix* System.***out***.println(str);  *// 输出结果是 [张三，李四，王五]* } |