**方法引用和Stream流**

# 知识点回顾

## 函数式接口

* 什么是函数式接口？

接口中只有一个抽象方法的接口

* 函数式接口使用哪个注解？

@FunctionalInterface

## 常用函数式接口

|  |  |  |
| --- | --- | --- |
| 接口 | 含义 | 抽象方法 |
| Supplier<T> | 提供者 | T get() |
| Consumer<T> | 消费者 | void accept(T t) |
| Predicate<T> | 谓语 | boolean test(T t) |
| Function<T,R> | 函数 | R apply(T t) |

# 教学目标

1. 能够通过4种方式使用方法引用
2. 能够理解流与集合相比的优点
3. 能够通过集合或数组获取流
4. 能够掌握常用的流操作
5. 能够使用流进行并行化操作
6. 能够将流中的内容收集到集合中
7. 能够将流中的内容收集到数组中

# 方法引用

在使用Lambda表达式的时候，我们实际上传递进去的代码就是一种解决方案：拿什么参数做什么操作。那么考虑一种情况：如果我们在Lambda中所指定的操作方案，已经有地方存在相同方案，那是否还有必要再写重复的功能代码?

## 什么是方法引用

方法引用是Lambda表达式的另一种表现形式，功能与Lambda相同，也可以理解为对匿名内部类的实现。在Java 8中，我们会使用Lambda表达式创建匿名方法。但是有时候，Lambda表达式可能仅仅调用一个已存在的方法，而不做任何其它事。对于这种情况，通过一个方法名字来引用这个已存在的方法会更加清晰。

方法引用是一种更加紧凑，易读的Lambda表达式。方法引用的操作符是双冒号"::"。

## 冗余的Lambda场景

* 需求说明：

1. 创建一个函数式接口Calculate，包含抽象方法int calc(int m,int n)，用于实现对两个数的计算。
2. 创建主类，使用匿名内部类实现Calculate接口，并且实现计算的功能。
3. 调用calc()方法，传入参数得到计算结果。
4. 使用Lambda实例化Calculate对象，并且自己写代码实现计算功能。
5. 调用calc()方法，传入参数得到计算结果。

* 案例效果：



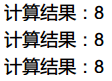
## 问题分析

假设在Demo01Method这个类中已经有了计算两个数的静态方法的实现，我们可以在Lambda表达式中直接调用这个方法，而不需要自己在Lambda表达式中去实现这个功能。

* 代码步骤：

1. 在主类中创建一个静态方法int sum(int a, int b)，实现两个数的相加。
2. 使用Lambda调用当前类的静态方法

* 运行效果：



## 用方法引用改进代码

能否省去Lambda的语法格式（尽管它已经相当简洁）呢？只要“引用”过去就好了：

* 案例效果：



* 案例代码：

package com.itheima;

//1) 创建一个函数式接口Calculate，包含抽象方法int calc(int m,int n)，用于实现对两个数的计算

interface Calculate {

//用于实现对两个数的计算

int calc(int m,int n);

}

public class Demo01Method {

//已经存在一个静态的方法，用来实现两个数的和

public static int sum(int a, int b) {

return a + b;

}

public static void main(String[] args) {

//使用匿名内部类实现Calculate接口

Calculate c1 = new Calculate() {

@Override

public int calc(int m, int n) {

//return m + n;

return Demo01Method.sum(m,n);

}

};

//调用方法

System.out.println(c1.calc(3,5));

//使用Lambda实例化Calculate对象

Calculate c2 = (m, n) -> m + n;

System.out.println(c2.calc(3,5));

//直接调用已经存在的方法

Calculate c3 = (m,n) -> Demo01Method.sum(m,n);

System.out.println(c3.calc(3,5));

//使用方法引用

Calculate c4 = Demo01Method::sum;

System.out.println(c4.calc(3,5));

}

}

* + 请注意其中的双冒号::写法，这被称为“**方法引用**”，而双冒号是一种新的语法。

## 方法引用符

双冒号::为引用运算符，而它所在的表达式被称为**方法引用**。如果Lambda要表达的函数方案已经存在于某个方法的实现中，那么则可以通过双冒号来引用该方法作为Lambda的替代者。

* + 要注意，这里的方法引用功能与Lambda是一样的，代替了Lambda表达式，也代替了以前的匿名内部类。可以理解为这个方法引用创建了一个匿名内部类，并且实现了接口中的方法，在实现的方法体中引用了另一个已经存在的方法来实现功能。

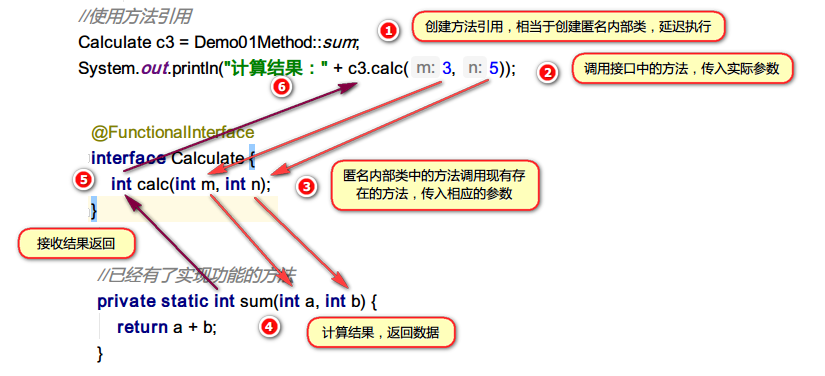
### 语义分析

对比下面两种写法，完全等效：

|  |  |
| --- | --- |
| 写法 | 格式 |
| Lambda表达式写法 | (m,n) -> Demo01Method.sum(m,n) |
| 方法引用写法 | Demo01Method::sum |

* 第一种语义是指：拿到参数之后经Lambda之手，继而传递给sum()方法去处理。
* 第二种语义是指：直接让Demo01Method类来引用sum()方法来取代Lambda。两种写法的执行效果完全一样，而第二种方法引用的写法更加简洁。

## 方法引用的过程



### 推导与省略

如果使用Lambda，那么根据“**可推导就是可省略**”的原则，无需指定参数类型和返回值——它们都将被自动推导。而如果使用方法引用，也是同样可以根据具体传入的参数类型和参数个数进行推导。

函数式接口是Lambda的基础，而方法引用是可以代替Lambda，让Lambda更加简化，但在功能上是一样的。

## 方法引用的原则：

1. 如果Lambda表达式的方法体中只有一句话，而这句话就是调用另一个方法，可以使用方法引用代替。
2. 被引用的方法与函数式接口中的抽象方法：参数类型相同，参数个数相同，返回值类型相同，与方法名无关。

# 四种方法引用类型

所有的方法引用都会有一个函数式接口

|  |  |  |
| --- | --- | --- |
| 方法引用类型 | 语法 | |
| 静态方法引用 | 类名::静态方法名 | |
| 对象方法引用 | 实例上的对象方法引用 | 对象名::成员方法名 |
| 类型上的对象方法引用 | 类名::成员方法名 |
| 类构造器引用 | 类名::new 创建一个对象new Xxx() | |
| 数组构造器引用 | 类名[]::new 创建数组 | |

## 通过类名称引用静态方法

|  |
| --- |
| 类名称引用静态方法的语法 |
| 类名::静态方法 |

由于在java.lang.Math类中已经存在了静态方法abs()，用于求一个数的绝对值。所以当我们需要通过Lambda来调用该方法时，有三种写法。

* 案例说明：

1. 有一个函数式接口Calcable，包含抽象方法int calc(int num)
2. 在Lambda中调用Math.abs()方法实现求绝对值
3. 直接通过Math类方法引用实现求绝对值

* 实现步骤：

1. 创建函数式接口Calcable，包含抽象方法int calc(int num)，用于计算传入整数，返回计算结果。
2. 创建主类，创建主函数，使用Lambda表达式创建Calcable对象，计算传入整数的绝对值。
3. 调用calc()方法传入-10，输出计算结果
4. 使用类方法引用创建Calcable对象，直接引用类方法Math::abs方法
5. 调用calc()方法传入-10，输出计算结果

* 运行效果：



* 实现代码：

package com.itheima;

interface Calcable {

//计算绝对值

int calc(int num);

}

public class Demo2StaticMethod {

public static void main(String[] args) {

//1. 使用内部类实现

Calcable c1 = new Calcable() {

@Override

public int calc(int num) {

return Math.abs(num); //调用一个已经存在的方法

}

};

System.out.println(c1.calc(-10));

//2. Lambda

Calcable c2 = num -> Math.abs(num);

System.out.println(c2.calc(-10));

//3. 方法引用

Calcable c3 = Math::abs;

System.out.println(c3.calc(-10));

}

}

* 方法引用分析



在上面的案例中接口中的int calc(int num)与被引用的int Math.abs(int num)，具有相同的行为，参数类型和返回值类型相同。在这个例子中，下面两种写法是等效的：

|  |  |
| --- | --- |
| 下面两种写法是等效的 |  |
| Lambda表达式 | m -> Math.abs(m) |
| 方法引用 | Math::abs |

## 通过对象引用成员方法

对象方法引用类型：实例上的对象方法引用、类型上的对象方法引用

### 实例上的对象方法引用

|  |
| --- |
| 实例上的对象方法引用语法 |
| 对象名::成员方法 |

提问：System.out是一个对象还是一个类？

答： out其实是一个对象，它是PrintStream类的一个对象

public static final PrintStream out = null;

* 案例需求：

使用Consumer接口，调用void accept(字符串)，将提供的字符串直接打印出来。

* 案例效果：



* 案例步骤：

1. 创建主类和主函数
2. 创建Consumer对象，这是一个函数式接口，有一个抽象方法void accept(T t)
3. 使用Lambda表达式，实现方法体，在方法体中调用System.out.println()方法打印字符串。
4. 调用accept()方法提供要打印的字符串
5. 创建Consumer对象，使用对象方法引用，引用out对象的println方法
6. 调用accept()方法提供要打印的字符串

* 案例代码：

package com.itheima;

import java.util.function.Consumer;

public class Demo02ObjectMethod {

public static void main(String[] args) {

//使用内部类

Consumer<String> c1 = new Consumer<String>() {

@Override

public void accept(String s) {

System.out.println(s); //调用另一个方法实现功能

}

};

c1.accept("Hello Java");

//使用Lambda

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

c2.accept("Hello World");

//方法引用

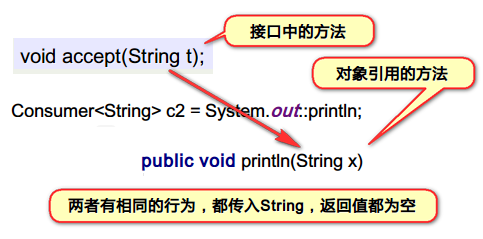
Consumer<String> c3 = System.out::println;

c3.accept("Hello Consumer");

}

}

* 方法引用分析：



在这个例子中，下面两种写法是等效的：

|  |  |
| --- | --- |
| 下面两种写法是等效的 |  |
| Lambda表达式 | s -> System.out.println(s) |
| 方法引用 | System.out::println |

### 类型上的对象方法引用

|  |
| --- |
| 类型上的对象方法引用语法 |
| 类名::成员方法 |

如果要引用一个类中的对象方法，每次都需要先实例化得到对象，再通过"对象::方法名"去调用也是比较繁琐的。能不能直接通过类名去调用一个对象方法呢？在方法引用中是可以的。

静态方法引用和类型上的对象方法引用拥有一样的语法，编译器会根据实际情况做出决定。

### 示例1

* 案例需求

使用Function函数式接口，String apply(String s)方法传入一个字符串，得到它的大写字符串返回。

* 案例效果



* 案例步骤

1. 创建主类和主函数
2. 创建Function<String,String>接口，先使用匿名内部类实现方法。
3. 使用Lambda实例化接口，传入字符串，调用字符串的toUpperCase()方法得到返回值。
4. 输出function对象的apply()方法的结果
5. 创建Function<String,String>接口，使用String::toUpperCase，引用对象方法，传入字符串，调用字符串的toUpperCase方法。这里的实例调用对象是s。
6. 输出function对象的apply()方法的结果

* 案例代码

package com.itheima;

import java.util.function.Function;

/\*

使用Function函数式接口，String apply(String s)方法传入一个字符串，得到它的大写字符串返回。

\*/

public class Demo03Method {

public static void main(String[] args) {

//匿名内部类

Function<String, String> f1 = new Function<String, String>() {

@Override

public String apply(String s) {

return s.toUpperCase(); //方法引用

}

};

System.out.println(f1.apply("Hello Java"));

//使用Lambda

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

System.out.println(f2.apply("Hello World"));

//方法引用

Function<String, String> f3 = String::toUpperCase;

System.out.println(f3.apply("NewBoy"));

}

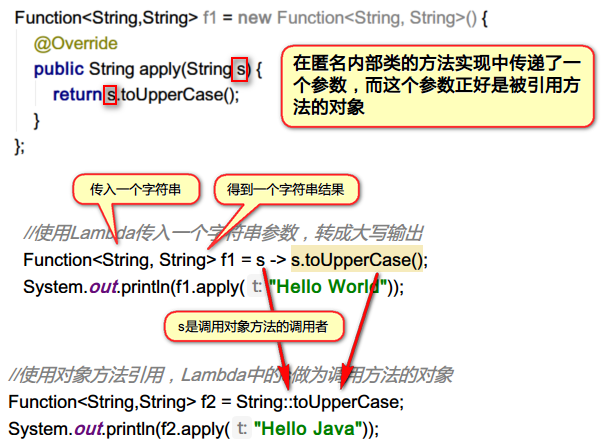
}

* + **当使用 类名::对象方法名 来引用方法时，Lambda表达式所接收的第1个参数用来调用对象方法，如果Lambda表达式接收多个参数，后面剩余的参数作为对象方法的参数传进去。**

|  |  |
| --- | --- |
| 下面两种写法是等效的 |  |
| Lambda表达式 | s -> s.toUpperCase() |
| 方法引用 | String::toUpperCase |

* 方法引用分析

Lambda表达式只传入了1个参数，而这个参数就是调用方法的对象，则可以使用类型上的方法引用。

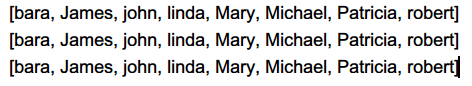


### 示例2

* 案例需求

有字符串数组的一组名字：{"bara", "James", "Mary", "john", "Patricia", "robert", "Michael", "linda"}。将这组名字忽略大小写对字符串数组进行排序，并且输出排序以后的数组。分别使用匿名内部类，Lambda表达式和类型上的对象方法引用来实现。

* 案例效果



* 实现步骤

|  |  |
| --- | --- |
| Arrays中排序的方法 | 说明 |
| sort(T[] a, Comparator c) | 使用指定的比较器进行排序  参数1：要排序的数组  参数2：指定比较器，重写比较器的方法，返回负整数、0 或正整数 |

|  |  |
| --- | --- |
| String中的比较方法 | 说明 |
| int compareToIgnoreCase(String other) | 根据指定 String 大于、等于还是小于此 String（不考虑大小写），分别返回一个负整数、0 或一个正整数。 |

1. 创建主类和主函数，直接调用Arrays.sort()方法进行排序。
2. 使用匿名内部类实现比较器，重写int compare(String s1, String s2)方法，方法内部调用s1.compareToIgnoreCase(s2)这个方法，返回方法比较的整数结果。
3. 使用Lambda实现比较器的类，方法有两个参数s1,s2，调用s1的字符串忽略大小写的方法返回一个整数。
4. 使用String::compareToIgnoreCase，实现比较器对象，调用对象方法。其中Lambda中的s1为方法的调用对象，s2为方法的参数。

* 案例代码

package com.itheima;

import java.util.Arrays;

import java.util.Comparator;

public class Demo04MethodRef {

public static void main(String[] args) {

String[] names = {"bara", "James", "Mary", "john", "Patricia", "robert", "Michael", "linda"};

//匿名内部类

/\* Arrays.sort(names, new Comparator<String>() {

@Override

public int compare(String o1, String o2) {

//忽略大小写比较

return o1.compareToIgnoreCase(o2); //直接调用另一个方法

}

});\*/

//使用lambda

//Arrays.sort(names, (o1,o2) -> o1.compareToIgnoreCase(o2));

//使用方法引用

Arrays.sort(names, String::compareToIgnoreCase);

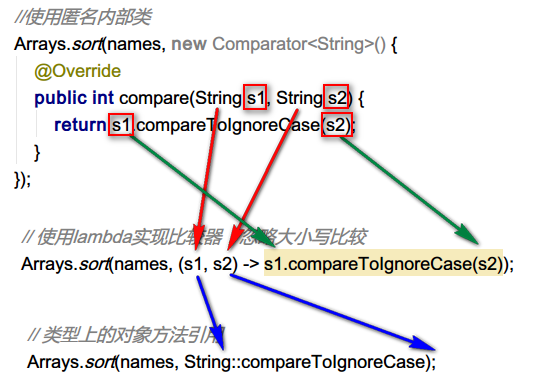
System.out.println(Arrays.toString(names));

}

}

* 代码分析

接口中的int compare(String s1,String s2)方法，在实现的过程中引用了s1.compareToIgnoreCase(s2)这个方法。传入的2个参数中，第1个参数是调用方法的对象，第2个参数是调用方法的参数。可以使用类型上的方法引用。



|  |  |
| --- | --- |
| 下面两种写法是等效的 |  |
| Lambda表达式 | (o1,o2)->o1.compareToIgnoreCase(o2) |
| 方法引用 | String::compareToIgnoreCase |

## 类的构造器引用

|  |
| --- |
| 类的构造器引用语法： |
| 类名::new |

由于构造器的名称与类名完全一样，但一个类可以有多个构造方法，参数不同。请看下面的案例：

* 案例效果：



* 案例步骤：

1. 创建汽车类，有一个String属性品牌
2. 创建有参的构造方法和无参的构造方法
3. 重写toString()方法，返回band+"汽车"
4. 创建函数式接口Factory，包含抽象方法Car makeCar(String name)，用于创建汽车对象。
5. 创建主类和主函数
6. 使用lambda表达式直接调用有参的构造方法实例化汽车。
7. 调用makeCar()方法，输出汽车对象。
8. 使用构造器引用，调用有参的构造方法，因为接口中的makeCar方法是有参数的
9. 调用makeCar()方法，输出汽车对象。

* 案例代码：

package com.itheima;

class Car {

private String brand; //牌

public Car() {

brand = "未知";

}

public Car(String brand) {

this.brand = brand;

}

@Override

public String toString() {

return brand + "牌的汽车";

}

}

//4) 创建函数式接口Factory，包含抽象方法Car makeCar(String name)，用于创建汽车对象。

interface Factory {

//创建一个Car对象

Car makeCar(String name); //方法的行为与构造方法的行为相同

}

public class Demo05New {

public static void main(String[] args) {

//使用内部类

Factory f1 = new Factory() {

@Override

public Car makeCar(String name) {

return new Car(name); //引用构造方法

}

};

System.out.println(f1.makeCar("BMW"));

//使用lambda

Factory f2 = name -> new Car(name);

System.out.println(f2.makeCar("BenZ"));

//使用类构造器引用

Factory f3 = Car::new;

System.out.println(f3.makeCar("Audi"));

}

}

* 代码分析：



|  |  |
| --- | --- |
| 下面两种写法是等效的 |  |
| Lambda表达式 | name -> new Car(name) |
| 方法引用 | Car::new |

## 数组的构造器引用

|  |
| --- |
| 数组构造器语法： |
| 类名[]::new |

数组也是Object的子类对象，所以同样具有构造器，只是语法稍有不同。如果对应到Lambda的使用场景中时，需要一个创建数组的函数式接口。

* 案例需求：

分别使用Lambda表达式和数组构造器引用创建2个长度各为5的整数数组。

* 案例效果：



* 案例步骤：

1. 创建一个用于创建数组的接口ArrayBuilder，包含一个抽象方法int[] buildArray(int length), 提供数组的长度，返回一个创建好的数组。
2. 创建主类主函数
3. 使用Lambda表达式创建上面的接口对象，调用方法创建一个长度为5的数组，并且输出数组。
4. 使用数组构造器创建上面的接口对象，调用方法创建一个长度为5的数组，并且输出数组。

* 案例代码：

package com.itheima;

import java.util.Arrays;

//接口ArrayBuilder，包含一个抽象方法int[] buildArray(int length), 提供数组的长度

interface ArrayBuilder {

int[] buildArray(int length);

}

//数组构造器的引用

public class Demo06Array {

public static void main(String[] args) {

//1. 内部类

ArrayBuilder a1 = new ArrayBuilder() {

@Override

public int[] buildArray(int length) {

return new int[length]; //方法引用

}

};

System.out.println(Arrays.toString(a1.buildArray(5)));

//2. 使用lambda

ArrayBuilder a2 = length -> new int[length];

System.out.println(Arrays.toString(a2.buildArray(6)));

//3.使用方法引用

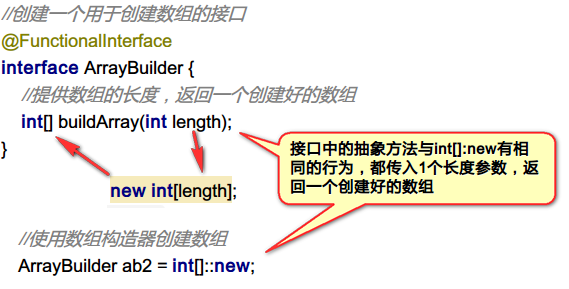
ArrayBuilder a3 = int[]::new;

System.out.println(Arrays.toString(a3.buildArray(7)));

}

}

* 代码分析：



在这个例子中，下面两种写法是等效的：

|  |  |
| --- | --- |
| 下面两种写法是等效的 |  |
| Lambda表达式 | length -> new int[length] |
| 方法引用 | int[]::new |

## 方法引用小结

### 方法引用语法

|  |  |  |
| --- | --- | --- |
| 方法引用类型 | 语法 | |
| 静态方法引用 | **类名::静态方法名** | |
| 对象方法引用 | **实例上的对象方法引用** | **对象名::成员方法** |
|  | **类型上的对象方法引用** | **类名::成员方法** |
| 类构造器引用 | **类名::new** | |
| 数组构造器引用 | **类名[]::new** | |

### 方法引用小结

|  |  |  |  |
| --- | --- | --- | --- |
| 接口中的抽象方法 | 引用的方法 | Lambda表达式 | 方法引用 |
| int calc(int num) | **int Math.abs(int x)** | **x -> Math.abs(x)** | **Math::abs** |
| void accept(String s) | **void println(String s)** | **s -> System.out.println(s)** | **System.out::println** |
| String apply(String s) | **s.toUpperCase()** | **s -> s.toUpperCase()** | **String::toUpperCase** |
| Car makeCar(String name) | **new Car(String name)** | **name -> new Car(name)** | **Car::new** |
| int[] buildArray(int length) | **new int[length]** | **length -> new int[length]** | **int[]::new** |

# Stream流概述

## 引言

说到Stream便容易想到I/O Stream，而实际上，谁规定“流”就一定是“IO流”呢?在Java 8中，得益于Lambda所带来的函数式编程，引入了一个**全新的Stream**概念，**用于解决已有集合类库既有的弊端。**

## Stream流引入案例

* 案例需求：
* 有如下集合

List<String> list = new ArrayList<>();   
list.add("张无忌");  
list.add("周芷若");  
list.add("赵敏");   
list.add("张强");   
list.add("张三丰");

* 按要求执行以下三个操作
  1. 首先筛选所有姓张的人。
  2. 然后筛选名字有三个字的人。
  3. 最后进行对结果进行打印输出。

### JDK1.8之前的实现方式

* 实现步骤：

1. 创建原始的集合添加上面的数据
2. 创建一个新的集合用于存储姓张的姓名，判断字符串是否以"张"开头，如果符合要求添加到集合中。
3. 再创建一个集合，遍历上面的集合，判断每个元素的长度，如果是3个就添加到集合中。
4. 循环输出最后过滤的结果

* 实例代码：

package com.itheima;

import java.util.ArrayList;

import java.util.List;

public class Demo07Stream {

public static void main(String[] args) {

//创建一个集合

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

list.add("张无忌");

list.add("周芷若");

list.add("赵敏");

list.add("张强");

list.add("张三丰");

// 1) 首先筛选所有姓张的人。

//创建一个新的集合

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

for (String s : list) {

if (s.startsWith("张")) {

names.add(s);

}

}

System.out.println(names);

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

// 2) 然后筛选名字有三个字的人。

for (String name : names) {

if (name.length() == 3) {

two.add(name);

}

}

// 3) 最后进行对结果进行打印输出。

System.out.println(two);

}

}

每当我们需要对集合中的元素进行操作的时候，总是需要进行循环、循环、再循环。这是理所当然的么?不是。循环是做事情的方式，而不是目的。另一方面，使用线性循环就意味着只能遍历一次。如果希望再次遍历，只能再使用另一个循环从头开始。**Lambda的衍生物Stream能给我们带来怎样更加优雅的写法呢?**

### Stream流的初体验

上面的案例如果使用Stream流来实现代码如何写呢？请看下面的案例。

* 案例效果：



* 案例步骤：

1. 将数组转成一个List，"张无忌","周芷若","赵敏","张强","张三丰"
2. 将list转成stream流，调用两次filter()方法，filter()方法的参数是Predicate，使用Lambda传递参数。
3. 首先筛选所有姓张的人，然后筛选名字有三个字的人，最后使用forEach进行对结果进行打印输出

* 案例代码：

package com.itheima;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Stream;

public class Demo08Stream {

public static void main(String[] args) {

//创建一个集合

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

list.add("张无忌");

list.add("周芷若");

list.add("赵敏");

list.add("张强");

list.add("张三丰");

// 1) 首先筛选所有姓张的人。

// 2) 然后筛选名字有三个字的人。

// 3) 最后进行对结果进行打印输出,支持链式写法

list.stream().filter(s -> s.startsWith("张")).filter(s -> s.length() == 3).forEach(System.out::println);

}

}

直接阅读代码的字面意思即可完美展示无关逻辑方式的语义：**获取流、过滤姓张、过滤长度为3、逐一打印。**

## 获取流的方式

java.util.stream.Stream<T>是Java 8新加入的最常用的流接口。这不是一个函数式接口。

获取一个流非常简单，有以下几种常用的方式：

|  |  |
| --- | --- |
| 获取流的方式 | 说明 |
| collection对象.stream() | 从一个集合对象中得到一个流对象 |
| Stream.of() | of()是一个静态方法，将一个数组转成一个流对象 |

### 根据Collection获取流

首先，java.util.Collection接口中加入了default方法stream()用来获取流，所以其所有子接口和实现类均可获取流。只要是Collection单列集合，都可以直接调用stream()方法获得流对象。

* 案例说明：

创建一个List和Set集合，添加几个元素，调用它们的stream()方法得到它们的流对象。并且调用forEach()方法输出每个元素。

* 案例代码：

@Test

public void testGetFromCollection() {

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

list.add("aaa");

list.add("bbb");

list.add("ccc");

//转成流

Stream<String> stream = list.stream();

stream.forEach(System.out::println);

Set<String> set = new HashSet<>();

set.add("牛魔王");

set.add("红孩儿");

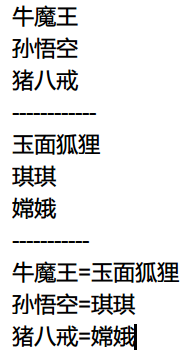
set.add("铁扇公主");

set.stream().forEach(System.out::println);

}

### 根据Map获取流

java.util.Map接口不是Collection的子接口，且其K-V数据结构不符合流元素的单一特征，所以获取对应的流需要分key(Set)、value(Collection)或entry(Set)等情况，双列集合不能直接获得流对象，需要先转换为Set集合。



* 案例说明：

1. 创建Map对象，向map中添加几个元素
2. 通过keySet()得到键的集合，再调用stream()得到键的流，输出所有的键。
3. 通过values()得到值的集合，再调用stream()得到值的流，输出所有的值。
4. 通过entrySet()得到键值的集合，再调用stream()得到流，输出所有的键和值。

@Test

public void testGetStreamFromMap() {

HashMap<String, String> map = new HashMap<>();

map.put("牛魔王","玉面狐狸");

map.put("孙悟空","琪琪");

map.put("猪八戒","嫦娥");

Set<String> set = map.keySet();

set.stream().forEach(System.out::println);

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

Collection<String> values = map.values();

values.stream().forEach(System.out::println);

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

Set<Map.Entry<String, String>> entries = map.entrySet();

entries.stream().forEach(System.out::println);

}

### 根据数组获取流

如果使用的不是集合或映射而是数组，由于数组对象不可能添加默认方法，所以Stream接口中提供了静态方法of，使用很简单：

|  |  |
| --- | --- |
| Stream中的静态方法 | 说明 |
| static <T> Stream<T> of(T... values) | 通过指定的值返回一个有顺序的流  参数：指定流中的所有元素  返回：一个新的流 |

* 示例：通过Stream类的静态方法of得到一组字符串的流对象

//通过数组得到一个流

@Test

public void testGetStreamFromArray() {

Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);

stream.forEach(System.out::println);

}

# java.util.stream.Stream常用方法

流操作的方法很丰富，这里介绍一些常用的方法，这些方法可以被分成两种：终结方法和非终结方法

## 非终结方法与终结方法

凡是返回值仍然为Stream接口的为非终结方法(函数拼接方法)，它们支持链式调用；而返回值不再为Stream接口的为终结方法，不再支持链式调用。如下表所示：

|  |  |  |
| --- | --- | --- |
| 方法名 | 是否终结方法 | 方法作用 |
| forEach | 是 | 用于遍历流中的元素，参数是：Consumer |
| count | 是 | 统计流中元素的个数 |
| filter | 否 | 使用指定的条件Predicate对流中元素过滤 |
| limit | 否 | 取得流中前面n个元素，生成一个新的流 |
| skip | 否 | 跳过n个元素，将后面的所有元素生成一个新的流 |
| map | 否 | 通过Function，将整个流中所有的元素映射成另一种元素 |
| concat | 否 | 将两个或多个流拼接成一个新的流 |

## Stream流中逐一处理的方法：forEach

### 方法声明

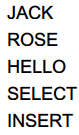
|  |  |
| --- | --- |
| 方法声明如下 | 说明 |
| void forEach(Consumer action) | 对此流的每个元素执行一个操作，这是一个终结方法。  对于并行流，此操作并不保证流的操作顺序，因为这样做将牺牲并行性的好处。  参数：对元素执行的操作 void accept(T t) |

### 方法演示

* 案例说明：

使用of方法将一组字符串转成流，调用forEach方法将每个元素转成大写，输出每个元素。

* 执行效果：



* 执行代码：

//forEach方法

//将每个元素转成大写，输出每个元素

@Test

public void testForEach() {

Stream<String> names = Stream.of("Jack","Rose","Make","Marry","William");

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

}

## 过滤：filter

### 方法声明

|  |  |
| --- | --- |
| 方法声明如下 | 说明 |
| Stream<T> filter(Predicate<T> p) | 返回一个与给定判断条件匹配的元素组成的流。  参数：谓词对象，指定判断条件，应用于每个元素以确定是否应该包含它。  返回：一个新的流 |

### Predicate接口

此前我们已经学习过java.util.function.Predicate函数式接口，其中唯一的抽象方法为：boolean test(T t);

该方法将会产生一个boolean值结果，代表指定的条件是否满足。如果结果为true，那么Stream流的filter方法将会留用元素；如果结果为false，那么filter方法将会舍弃元素。

### 基本使用

* 案例说明：

1. 通过Stream类的of静态方法获得流对象："Jack","Rose","Hello","Select","Insert"
2. 得到所有长度是4个的元素生成一个新的流
3. 使用forEach()输出打印流对象

* 执行效果：



* 案例代码：

//filter方法

//2) 得到所有长度是4个的元素生成一个新的流

@Test

public void testFilter() {

Stream<String> names = Stream.of("Jack","Rose","Make","Marry","William");

names.filter(t -> t.length() == 4).forEach(System.out::println);

}

## 统计个数：count

### 方法声明如下

|  |  |
| --- | --- |
| 方法声明如下 | 说明 |
| long count() | 返回此流中元素的计数。这是一个终结方法。  返回：此流中元素的个数。 |

### 基本使用

* 案例说明：

统计一个流中元素的个数

* 案例代码：

@Test

public void testCount() {

//统计元素的个数

Stream<String> names = Stream.of("Jack","Rose","Make","Marry","William");

System.out.println(names.count());

}

## 获取前几个：limit

### 方法声明如下

|  |  |
| --- | --- |
| 方法声明如下 | 说明 |
| Stream<T> limit(long maxSize) | 获得流中前maxSize个元素，将元素添加到另一个流中返回 如果maxSize大于等于当前流的元素个数，则所有元素都会获取到 如果maxSize 等于0，则会获得一个空流。 |

### 基本使用

* 案例说明：

得到流中前3和元素输出，得到流中前10个元素输出，得到流中前0个元素输出

* 案例代码：

@Test

public void testLimit() {

Stream<String> names = Stream.of("Jack","Rose","Make","Marry","William");

//返回一个流对象

names.limit(3).forEach(System.out::println);

//流只能操作一次，java.lang.IllegalStateException: stream has already been operated upon or closed

names.limit(0).forEach(System.out::println);

}

## 跳过前几个：skip

### 方法声明如下

|  |  |
| --- | --- |
| 方法声明如下 | 说明 |
| Stream<T> skip(long n) | 跳过前面n元素，将后面元素添加到另一个流中 |

### 基本使用

* 案例说明：

跳过前面3个元素得到一个新的流，并且使用forEach输出。

* 案例代码：

//跳过前面n元素，将后面元素添加到另一个流中

@Test

public void testSkip() {

Stream<String> names = Stream.of("Jack","Rose","Make","Marry","William");

//跳过3个元素

names.skip(3).forEach(System.out::println);

}

## 映射：map

### 方法说明

|  |  |
| --- | --- |
| 方法声明如下 | 说明 |
| <R> Stream<R> map(Function mapper) | 使用map可以遍历集合中的每个元素，并对其进行操作。将一种类型映射成另一种类型，得到一个新的流。 |

该接口需要一个Function函数式接口参数，可以将当前流中的T类型数据转换为另一种R类型的流。

### 复习Function接口

此前我们已经学习过java.util.function.Function函数式接口，其中唯一的抽象方法为：R apply(T t);这可以将一种T类型转换成为R类型，而这种转换的动作，就称为“映射”。

### 基本使用

* 案例说明：

1. 有一组字符串类型的数字流
2. 使用map方法，将字符串类型全部转成整数，可以使用静态方法引用，映射成一个新的流
3. 使用map方法，将整数类型每个元素加1，映射成一个新的流
4. 使用forEach输出新流中的每个元素。  
   注：流只能消费一次，即只能使用forEach一次，第二次使用会抛出异常。  
   java.lang.IllegalStateException: stream has already been operated upon or closed

* 案例代码：

@Test

public void testMap() {

Stream<String> nums = Stream.of("111","222","25","9");

//将字符串类型全部转成整数, R apply(T t)

//将整数类型每个元素加1

nums.map(s -> Integer.parseInt(s)).map(i -> i+1).forEach(System.out::println);

}

## 组合：concat

如果有两个流，希望合并成为一个流，那么可以使用Stream接口的静态方法concat：

### 方法声明如下

|  |  |
| --- | --- |
| 方法声明如下 | 说明 |
| public static <T> Stream<T> concat(Stream a, Stream b) | 静态方法，通过类名调用，将两个流合并成一个流  返回一个新的流 |

### 基本使用

* 案例说明

1. 从两个字符串数组创建两个流
2. 使用concat方法对两个流进行拼接
3. 输出拼接后的流

* 案例效果



* 案例代码：

public void testConcat() {

//两个流必须是同一种类型

//Stream<String> nums = Stream.of("111","222","25","9");

Stream<String> names = Stream.of("Jack","Rose","Make","Marry","William");

Stream<Integer> nums = Stream.of(1,2,3,4);

//结果是两个流的共同父类

Stream<Object> all = Stream.concat(nums, names);

all.forEach(System.out::println);

}

## 流的综合案例

* 案例说明

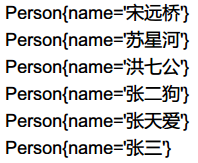
现在有两个ArrayList集合存储队伍当中的多个成员姓名，两个队伍（集合）的代码如下：

List<String> one = new ArrayList<>();  
 one.add("迪丽热巴");  
 one.add("宋远桥");  
 one.add("苏星河");  
 one.add("老子");  
 one.add("庄子");  
 one.add("孙子");  
 one.add("洪七公");  
  
 List<String> two = new ArrayList<>();  
 two.add("古力娜扎");  
 two.add("张无忌");  
 two.add("张三丰");  
 two.add("赵丽颖");  
 two.add("张二狗");  
 two.add("张天爱");  
 two.add("张三");

* 要求如下：

1. 使用传统的for循环（或增强for循环）依次进行以下若干操作步骤
2. 使用Stream方式依次进行以下若干操作步骤
3. 第一个队伍只要名字为3个字的成员姓名；
4. 第一个队伍筛选之后只要前3个人；
5. 第二个队伍只要姓张的成员姓名；
6. 第二个队伍筛选之后不要前2个人；
7. 将两个队伍合并为一个队伍；
8. 根据姓名创建Person对象；
9. 打印整个队伍的Person对象信息。

* 执行效果：



* 实现代码

package com.itheima;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Stream;

class Person {

private String name;

public Person(String name) {

this.name = name;

}

@Override

public String toString() {

return "人：" + name;

}

}

/\*\*

\* 综合案例

\*/

public class Demo10 {

public static void main(String[] args) {

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

one.add("迪丽热巴");

one.add("宋远桥");

one.add("苏星河");

one.add("老子");

one.add("庄子");

one.add("孙子");

one.add("洪七公");

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

two.add("古力娜扎");

two.add("张无忌");

two.add("张三丰");

two.add("赵丽颖");

two.add("张二狗");

two.add("张天爱");

two.add("张三");

// 1) 第一个队伍只要名字为3个字的成员姓名；

// 2) 第一个队伍筛选之后只要前3个人；

Stream<String> s1 = one.stream().filter(s -> s.length() == 3).limit(2);

//s1.forEach(System.out::println);

// 3) 第二个队伍只要姓张的成员姓名；

// 4) 第二个队伍筛选之后不要前2个人；

Stream<String> s2 = two.stream().filter(s -> s.startsWith("张")).skip(2);

//s2.forEach(System.out::println);

// 5) 将两个队伍合并为一个队伍；

Stream<String> all = Stream.concat(s1, s2);

// all.forEach(System.out::println);

// 6) 根据姓名创建Person对象；

// 7) 打印整个队伍的Person对象信息。

all.map(Person::new).forEach(System.out::println);

}

}

# 收集Stream结果

集合和流虽然具有一些相似性，但有不同的目标。集合主要是对其中的元素进行管理和访问add,remove。而流不提供直接访问或操纵其元素的功能，流的出现只是对集合中的元素进行声明性地描述或对元素进行计算等操作。而且流只能进行一次操作，即调用非终结方法或终结方法操作一次。所以对流操作完成之后，有时需要将其结果进行收集，获取对应的集合、数组等。

集合 🡪 流 🡪 集合

## 收集到集合中

Stream流提供collect方法，其参数需要指定一个收集器对象java.util.stream.Collector。java.util.stream.Collectors类中提供一些方法，可以作为Collector接口的实例：

|  |  |
| --- | --- |
| Stream流中的方法 | 说明 |
| collect(Collector collector) | 使用指定的收集器收集元素 |

|  |  |
| --- | --- |
| Collectors类提供一些方法 | 说明 |
| public static Collector toList() | 得到一个List集合的收集器 |
| public static Collector toSet() | 得到一个Set集合的收集器 |

### 收集到集合的示例

* 案例需求：

将一个字符串组成的流对象，调用collect方法，分别转成List和Set集合。

注意：流只能收集一次

* 案例代码：

@Test

public void testCollect() {

Stream<String> names = Stream.of("Jack","Rose","Make","Marry","William");

//转成List

/\* List<String> list = names.collect(Collectors.toList());

System.out.println(list);\*/

//转成Set

Set<String> set = names.collect(Collectors.toSet());

System.out.println(set);

}

## 收集到数组中

|  |  |
| --- | --- |
| Stream接口中定义的方法 | 说明 |
| Object[] toArray() | 返回一个包含流中元素的数组，返回值类型是Object[]的 |
| <A> A[] toArray(IntFunction<A[]> generator) | 返回包含此流元素的数组，使用提供的生成器函数分配返回的数组 |

### 收集到数组的示例

将一个流转成Object数组

### 解决泛型数组问题

有了Lambda和方法引用之后，可以使用toArray方法的另一种重载形式传递一个IntFunction<A[]>的函数，继而从外面指定泛型参数。这是IntFunction接口的源代码：

@FunctionalInterface  
**public interface** IntFunction<R> {  
 /\*\*  
 \* 使用给定的整数变量，通过函数计算出结果  
 \*/  
 R apply(**int** value);  
}

方法说明：

|  |  |
| --- | --- |
| IntFunction接口中的方法 | 说明 |
| R apply(int value) | 传入一个整数变量，计算出结果 |

* 案例说明：

1. 创建一个由字符串组成的流
2. 调用toArray()方法生成一个String[]数组。
3. 使用Lambda和数组方法引用的写法分别写出来
4. 输出转换后的数组

@Test

public void testToArray() {

Stream<String> names = Stream.of("Jack","Rose","Make","Marry","William");

//Object[] objects = names.toArray(); //直接得到数组

//R apply(int value), i是数组的长度

String[] strings = names.toArray(i -> new String[i]);

System.out.println(Arrays.toString(strings));

}

# 并行流

前面使用的都是串行流，即单线程执行的流，其实Stream API还提供了并行流。

串行流的操作是在单线程上执行的，而并行流的操作是在多线程上并行执行的。所以并行流的执行效率会更高，它是由多个线程同时操作一个流对象。

|  |  |
| --- | --- |
| java.util.Collection接口两个默认方法 | 说明 |
| Stream stream() | 得到一个串行流 |
| Stream parallelStream() | 得到一个并行流 |

这两个方法都是返回Stream对象，说明它们在功能的使用上是没差别的。唯一的差别就是流内部处理的时候一个是单线程处理和一个是多线程的处理。

## 串行流转换为并行流

|  |  |
| --- | --- |
| java.util.stream.BaseStream接口定义的方法 | 说明 |
| S parallel() | Stream的父接口中定义了一个parallel()方法，可以将一个串行流转成并行流，转换后与之前的流是同一个流。 |

只需要在流上调用一下无参数的parallel方法，那么当前流即可变身成为支持并行操作的流，返回值仍然为Stream类型。例如：

public static void main(String[] args) {

Stream<String> names = Stream.of("Jack","Rose","Make","Marry","William");

//将串行流转成并行流

Stream<String> parallel = names.parallel();

System.out.println(names == parallel);

parallel.forEach(System.out::println);

}

## 直接获取并行流

在通过集合获取流时，也可以直接调用parallelStream方法来直接获取支持并行操作的流。方法定义为：

|  |  |
| --- | --- |
| Collection接口中定义的方法 | 说明 |
| default Stream<E> parallelStream() | 获取支持并行操作的流。 |

* 案例说明：

创建一个ArrayList，添加几个字符串元素，通过list对象调用上面的方法得到一个并行流。输出并行流的内容。

* 应用代码：

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

list.add("张无忌");

list.add("周芷若");

list.add("赵敏");

list.add("张强");

Stream<String> stream = list.parallelStream();

stream.forEach(System.out::println);

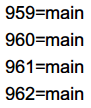
## 并行流的分析

* 案例说明：

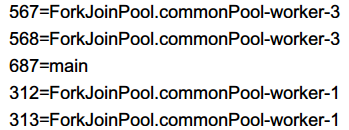
1. 创建一个测试类，声明一个全局ArrayList<Integer>
2. 创建init()方法，使用@Before注解，在每个测试方法运行前执行
3. 循环向集合中添加1000个数
4. 写测试testStream()串行流的方法，将上面的集合调用stream()方法转成串行流，再调用forEach方法输出每个元素的值和处理当前线程的名字。
5. 写测试testParallel()并行流的方法，将上面的集合调用parallelStream()方法转成并行流，再调用forEach方法输出每个元素的值和当前线程的名字。
6. 查看运行结果可以得知，并行流中同时有多个线程在处理，不是按顺序处理的，而串行流中只有一个线程在依次处理。

* 案例效果：

串行流



并行流



* 案例代码：

package com.itheima;

import org.junit.Before;

import org.junit.Test;

import java.util.ArrayList;

import java.util.stream.Stream;

public class Demo12Test {

ArrayList<Integer> nums = new ArrayList<>();

@Before

public void init() {

for (int i = 0; i <100 ; i++) {

nums.add(i);

}

}

@Test

public void testStream() {

Stream<Integer> stream = nums.stream();

stream.forEach(n -> {

System.out.println(n + "=" + Thread.currentThread().getName());

});

}

@Test

public void testParallelStream() {

Stream<Integer> stream = nums.parallelStream();

stream.forEach(n -> {

System.out.println(n + "=" + Thread.currentThread().getName());

});

}

}

### 使用并行流需要考虑几点：

1. 并行流只是做到不浪费CPU，假如本身电脑CPU的负载很大，用并行流并不能起到很好的效果。
2. 不要在多线程中使用并行流，原因同上类似，多线程都抢着使用CPU是不会有太多的提升效果的，反而还会加大线程切换开销
3. 并行流操作会带来不确定性，请确保流中每个元素是无状态且没有关联的