# JDK8

目录

[JDK8](#_Toc13499)

[第一章 简介](#_Toc25084)

[1.1 Java8因何而生](#_Toc14987)

[1.2 什么是函数式编程](#_Toc29207)

[1.3 Java8的编程风格](#_Toc8507)

[1.3 函数式接口认识](#_Toc1573)

[第一章 Lambda表达式](#_Toc25943)

[2.1 第一个Lambda表达式](#_Toc12058)

[2.2 lambda表达式的作用域](#_Toc10297)

[2.3 使用Lambda实现Runnable、Comparator](#_Toc21592)

[2.4类型推断](#_Toc31098)

[2.5 方法引用](#_Toc32201)

[2.6 常用函数接口](#_Toc1323)

[2.61 java.util.function.Predicate<T>接口](#_Toc9156)

[2.62 java.util.function.Function<T, R> 接口](#_Toc5295)

[2.63 java.util.function.Supplier<T>接口](#_Toc23290)

[2.64 java.util.function.Consumer<T>接口](#_Toc32609)

[第三章 Stream流](#_Toc9480)

[3.1 如何生成Stream](#_Toc20845)

[3.2 Stream操作](#_Toc12898)

[3.21 filter](#_Toc25227)

[3.22 map](#_Toc2214)

[3.23 flatMap](#_Toc9086)

[3.24 sorted](#_Toc22698)

[3.25 peek](#_Toc24700)

[3.26 limit](#_Toc27061)

[3.27 skip](#_Toc26605)

[3.28 distinct](#_Toc27589)

[3.29 foreach](#_Toc16429)

[3.210 count](#_Toc20695)

[3.211 sum](#_Toc11302)

[3.212 max](#_Toc3458)

[3.213 min](#_Toc32261)

[3.214 reduce](#_Toc25451)

[3.215 collect](#_Toc28761)

[3.216 findFirst](#_Toc11357)

[3.217 allMatch](#_Toc16727)

[3.218 anyMatch](#_Toc4868)

[3.219 noneMatch](#_Toc26911)

[3.3 数据并行化](#_Toc20123)

# 第一章 简介

Java 8 (又称为 jdk 1.8) 是 Java语言开发的一个主要版本。 Oracle 公司于 2014 年 3 月 18 日发布 Java 8 ，它支持函数式编程，新的 JavaScript 引擎，新的日期 API，新的Stream API 等。在开始探索Java8的核心特性如lambda表达式之前，首先我们要知道它因何而生。

## 1.1 Java8因何而生

1996年1月，Java1.0发布，此后计算机编程领域发生了翻天覆地的变化。商业发展需要更复杂的应用，大多数程序都跑在更强大的装备[多核CPU](https://www.baidu.com/s?wd=%E5%A4%9A%E6%A0%B8CPU&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)的机器上。带有高效运行期编译器的Java虚拟机（JVM）的出现，使得程序员将精力更多放在编写干净、易于维护的代码上，而不是思考如何将每一个[CPU时钟](https://www.baidu.com/s?wd=CPU%E6%97%B6%E9%92%9F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)、每一字节内存物尽其用。  
 1、[多核CPU](https://www.baidu.com/s?wd=%E5%A4%9A%E6%A0%B8CPU&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)的兴起成了不容回避的事实，涉及锁的算法不但容易出错，而且消耗时间。

1. 开发类库的程序员使用Java时，发现抽象的级别还不够。面对大型数据集合，Java还欠缺高效的并行操作。开发者可以使用Java8编写复杂的及和处理算法，只需要简单修改一个方法，就能让代码在多核CPU上高效运行。为了编写并行处理这些大数据的类库，需要在语言层面上修改现有的Java：增加[lambda表达式](https://www.baidu.com/s?wd=lambda%E8%A1%A8%E8%BE%BE%E5%BC%8F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)。  
    3、使用函数式编程的思想，面向对象编程是对数据进行抽象，而[函数式编程](https://www.baidu.com/s?wd=%E5%87%BD%E6%95%B0%E5%BC%8F%E7%BC%96%E7%A8%8B&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)是对行为进行抽象。现实世界中，数据和行为并存，程序也是如此，因此这两种编程方式我们都得学。  
   通过函数式编程，程序员能编写出更容易阅读的代码——这种代码更多地表达了[业务逻辑](https://www.baidu.com/s?wd=%E4%B8%9A%E5%8A%A1%E9%80%BB%E8%BE%91&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)，而不是从机制上如何实现。易读的代码也易于维护、更可靠、更不容易出错。  
    4、在写[回调函数](https://www.baidu.com/s?wd=%E5%9B%9E%E8%B0%83%E5%87%BD%E6%95%B0&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)和事件处理器时，程序员不必再纠缠于匿名内部类的冗繁和可读性，[函数式编程](https://www.baidu.com/s?wd=%E5%87%BD%E6%95%B0%E5%BC%8F%E7%BC%96%E7%A8%8B&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)让事件处理系统变得更加简单。能将函数方便地传递也让编写惰性代码变得容易，只有在真正需要的时候，才初始化变量的值。  
    5、Java 8还让集合类可以拥有一些额外的方法：default(默认)方法。程序员在维护自己的类库时，可以使用这些方法。

## 1.2 什么是函数式编程

每个人对函数式编程的理解不尽相同，但其核心是：在思考问题时，使用不可变值和函数，函数对一个值进行处理，映射成另一个值。

函数式编程，顾名思义，这种编程是以函数思维做为核心，在这种思维的角度去思考问题。这种编程最重要的基础是λ演算，接受函数当作输入和输出。

面向对象编程，这种编程是把问题看作由对象的属性与对象所进行的行为组成。基于对象的概念，以类作为对象的模板，把类和继承作为构造机制，以对象为中心，来思考并解决问题。

两边编程思想在不同的方面各有优劣，比如在面对对象的思想中，对于“对象”的定义，试图将所有的事情归纳到这个概念中，如“万物皆对象”的理念，有些情况下，有些东西使用函数而不是对象去表示，会更加简洁。所以可以搭配函数式编程，使用lambda表达式。

## 1.3 Java8的编程风格

//创建两个集合

List<String> names1 = Arrays.*asList*("yewenjie","luoji","zhangbeihai","shiqiang");

List<String> names2 = Arrays.*asList*("yewenjie","luoji","zhangbeihai","shiqiang");

names2.sort((t1,t2)->t1.compareTo(t2));

//使用java7的方式进行排序

names1.sort(**new** Comparator<String>() {

**public** **int** compare(String o1, String o2) {

**return** o1.compareTo(o2);

}

});

names2.forEach((t)->System.***out***.println(t));

//使用java7的方式进行遍历

**for**(String name : names1){

System.***out***.println(name);

}

System.***out***.println("======");

//使用java8的方式进行排序

//t1,t2的类型由List<String>推断得出

//lambda表达式/函数式接口

names2.sort((t1,t2)->t1.compareTo(t2));

//使用java8的方式进行遍历

//遍历方式1：lambda表达式/函数式接口

names2.forEach((t)->System.***out***.println(t));

//遍历方式2：方法引用的知识

//names2.forEach(System.***out***::println);

执行之后的结果：

luoji

shiqiang

yewenjie

zhangbeihai

======

luoji

shiqiang

yewenjie

zhangbeihai

### 1.3 函数式接口认识

函数式接口(Functional Interface)就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接口。

非抽象方法需要是默认/静态/Object的方法:

1）函数式接口里允许定义默认方法:

函数式接口里是可以包含默认方法，因为默认方法不是抽象方法，其有一个默认实现，所以是符合函数式接口的定义的

2）函数式接口里允许定义静态方法:

函数式接口里是可以包含静态方法，因为静态方法不能是抽象方法，是一个已经实现了的方法，所以是符合函数式接口的定义的

3）函数式接口里允许定义 java.lang.Object 里的 public方法，且只能定义为抽象的,而且不必必须实现，因为可以从Object继承，函数式接口里是可以包含Object里的public方法，这些方法对于函数式接口来说，不被当成是抽象方法（虽然它们是抽象方法）；因为任何一个函数式接口的实现，默认都继承了 Object 类，包含了来自 java.lang.Object 里对这些抽象方法的实现；

Java 8为函数式接口引入了一个新注解@FunctionalInterface，主要用于编译级错误检查，加上该注解，当你写的接口不符合函数式接口定义的时候，编译器会报错。

提醒：加不加 @FunctionalInterface 对于接口是不是函数式接口没有影响，该注解只是提醒编译器去检查该接口是否仅包含一个抽象方法。

自定义函数式接口：

@FunctionalInterface

**public** **interface** Test2FunctionInterface {

**void** method1();

//void method2();//只能有一个抽象方法

/\*public void method3(){//不能有被实现的方法

System.out.println("method3");

}\*/

//可以有默认被实现的方法，可以有多个

**default** **void** method4(){

System.***out***.println("method4");

}

**default** **void** method5(){

System.***out***.println("method5");

}

//可以有静态方法

**static** **public** **void** method6(){

System.***out***.println("method6");

}

**static** **public** **void** method7(){

System.***out***.println("method7");

}

}

JDK 1.8 之前已有的函数接口如：

java.lang.Runnable

**public** **abstract** **void** run();

java.util.Comparator

**int** compare(T o1, T o2);

java.awt.event.ActionListener

**public** **void** actionPerformed(ActionEvent e);

......

JDK 1.8之后新增的函数接口：

java.util.function包中包含了很多接口，用来支持 Java的函数式编程，后解

# Lambda表达式

Java8最大的变化是引入了Lambda表达式——一种紧凑的，传递“**行为**”的方式。lambda 允许把函数作为一个方法的参数（函数作为参数传递进方法中）。

## 2.1 第一个Lambda表达式

1）语法：

(parameters)->expression

(parameters)->{statements;}

2）特征:

可选类型声明：不需要声明参数类型，编译器可以统一识别参数值。

可选的参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号。

可选的大括号：如果主体包含了一个语句，就不需要使用大括号。

可选的返回关键字：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指定明表达式返回了一个数值。

1. Lambda表达式实例：

//不接受参数，返回值为字符串

()-> "hello"

()->{return "hello"}

//接受一个int类型参数x，返回x+1

(int x)->{return x+1;}

(x)->{return x+1;}

x->{return x+1;}

x->x+1

//接受一个String类型参数，只做打印，不返回任何值

(String x)->System.out.println(x)

//接受两参/多参，返回两数的和

(int x,int y)-> {return x+y;}

(x,y)->x+y

给定接口：

@FunctionalInterface

**public** **interface** Action1 {

**void** run();

}

@FunctionalInterface

**public** **interface** Action2<T> {

T run();

}

@FunctionalInterface

**interface** SayService{

**void** sayMessage(String Message);

}

@FunctionalInterface

**interface** MathOperation{

**int** operate(**int** x,**int** y);

}

@Test

**public** **void** method1\_jdk7(){

//1.测试lambda表达式的基本写法

//如果是java7面对函数式接口，需要使用匿名内部类的写法

Action1 a1 = **new** Action1() {

@Override

**public** **void** run() {

System.***out***.println("JDK7: Action1");

}

};

a1.run();

SayService s1 = **new** SayService() {

@Override

**public** **void** sayMessage(String Message) {

System.***out***.println("JDK7: 匿名内部类的写法:"+Message);

}

};

s1.sayMessage("hello");

MathOperation m1Add = **new** MathOperation() {

@Override

**public** **int** operate(**int** x, **int** y) {

**return** x+y;

}

};

MathOperation m2Minus = **new** MathOperation() {

@Override

**public** **int** operate(**int** x, **int** y) {

**return** x-y;

}

};

Test2LambdaDemo1 l1 = **new** Test2LambdaDemo1();

System.***out***.println(l1.mathTest(10, 20, m1Add));

System.***out***.println(l1.mathTest(10, 20, m2Minus));

}

运行结果：

JDK7: Action1

JDK7: 匿名内部类的写法:hello

30

-10

使用JDK8的写法：

Action1 a1 = ()->{System.***out***.println("lambda表达式实现：不接收参数，没有返回值");};

Action2 a2 = ()->"lambda表达式实现：不接收参数，返回此字符串";

a1.run();

System.***out***.println(a2.run());;

//类型声明

SayService sl1 = (String message)->{System.***out***.println("lambda1:"+message);};

//不用类型声明

SayService sl2 = (message)->{System.***out***.println("lambda2:"+message);};

//用括号

SayService sl3 = message->{

**int** a = 20;

System.***out***.println(a+"lambda3:"+message);

};

//不用括号

SayService sl4 = message->System.***out***.println("lambda4:"+message);

sl1.sayMessage("类型声明");

sl2.sayMessage("不用类型声明");

sl3.sayMessage("用括号");

sl4.sayMessage("不用括号");

//使用类型，大括号中的返回语句

MathOperation mlAdd = (**int** a,**int** b)->{

a+=b;

**return** a;

};

//不使用类型，大括号中的返回语句

MathOperation mlMinus = (a,b)->{**return** a-b;};

//不使用类型，没有大括号，没有返回语句

MathOperation mlMult = (a,b)->a\*b;

//调用方式1

System.***out***.println(mlAdd.operate(10, 20));

System.***out***.println(mlMinus.operate(10, 20));

System.***out***.println(mlMult.operate(10, 20));

//调用方式2

Test2LambdaDemo1 l1 = **new** Test2LambdaDemo1();

System.***out***.println(l1.mathTest(10, 20, mlAdd));

System.***out***.println(l1.mathTest(10, 20, mlMinus));

System.***out***.println(l1.mathTest(10, 20, mlMult));

运行结果：

lambda表达式实现：不接收参数，没有返回值

lambda表达式实现：不接收参数，返回此字符串

lambda1:类型声明

lambda2:不用类型声明

20lambda3:用括号

lambda4:不用括号

30

-10

200

30

-10

200

Lambda 表达式主要用来定义行内执行的方法类型接口，

例如，一个简单方法接口。在上面例子中，我们使用各种类型的Lambda表达式来定义Action1,Action2,sayMessage,MathOperation接口的方法。

Lambda 表达式免去了使用匿名方法的麻烦，并且给予Java简单但是强大的函数化的编程能力。

## 2.2 lambda表达式的作用域

lambda表达式的作用域，类似于匿名内部类，可以访问外部变量，局部变量必须是final，这就是说不能在 lambda 内部修改定义在域外的局部变量，否则会编译错误。

注：1） lambda 表达式的局部变量可以不用声明为 final，但是必须不可被后面的代码修改（即隐性的具有 final 的语义）

2）在 Lambda 表达式当中不允许声明一个与局部变量同名的参数或者局部变量，可以与实例/静态变量相同

**public** **class** Test3LambdaDemo2 {

**public** **static** String *name1* = "tom";

**public** String name2 = "tom2";

**public** **void** test1(){

*name1* = "n1";

SayService sl1 = m->System.***out***.println(m+*name1*);;

sl1.sayMessage("hah");

name2 = "n2";

SayService sl2 = m->System.***out***.println(m+name2);;

sl2.sayMessage("heih");

String name3 = "tom3";

// name3 = "n3?";//不能修改值

// String m = "xix";//Lambda 表达式当中不允许声明一个与局部变量同名的参数

// String m2 = "xix";//Lambda 表达式当中不允许声明一个与局部变量同名的局部变量

SayService sl3 = m->{

// name3 = "n3";//一样不能修改

String m2 = "ss";

System.***out***.println(m+name3);

};

sl3.sayMessage("hah");

}

}

运行结果：

hahn1

heihn2

hahtom3

### 2.3 使用Lambda实现Runnable、Comparator

@Test

**public** **void** method1(){

//java7实现Runnable接口，匿名内部类

Runnable r1 = **new** Runnable() {

@Override

**public** **void** run() {

System.***out***.println("hello1:");

}

};

**new** Thread(r1).start();

//lambda实现

Runnable r2 = ()->System.***out***.println("world2:");

**new** Thread(r2).start();

}

运行结果：

hello1:

world2:

@Test

**public** **void** method2(){

List<String> names = Arrays.*asList*("yewenjie","liucixin","yuntianming","luoji","chenxin");

//java7实现

Collections.*sort*(names, **new** Comparator<String>() {

@Override

**public** **int** compare(String o1, String o2) {

**return** o1.compareTo(o2);

}

});

**for**(String name : names){

System.***out***.println(name);

}

//java8

Collections.*sort*(names,(s1,s2)->s1.compareTo(s2));

names.forEach(x->System.***out***.println(x));

System.***out***.println("===============");

Collections.*sort*(names, (s1,s2)->s1.length()-s2.length());

names.forEach(System.***out***::println);

}

运行结果：

chenxin

liucixin

luoji

yewenjie

yuntianming

chenxin

liucixin

luoji

yewenjie

yuntianming

===============

luoji

chenxin

liucixin

yewenjie

Yuntianming

### 2.4类型推断

某些情况下，用户需要手动指明类型，有时候省略类型信息可以减少干扰，更易弄清状况。Lambda表达式中的类型推断，实际上是Java7就引入的目标类型推断的扩展。

如java7中：

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

如果将构造函数直接传一个方法，也可以根据方法签名来推断类型：

**public** **static** **void** method(Map<String, String> map){

}

**public** **static** **void** main(String[] args) {

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

Test5TypeDemo.*method*(**new** HashMap<>());

}

在java8中上例代码可以通过编译，java7中不能。

### 2.5 方法引用

方法引用使得开发者可以直接引用现存的方法、Java类的构造方法或者实例对象。方法引用和Lambda表达式配合使用，使得java类的构造方法看起来紧凑而简洁，没有很多复杂的模板代码。每次写出形如：

X->foo(x)的lambda表达式时，和直接调用方法foo是一样的，方法引用只是基于这样的事实，提供了一种简短的语法而已，其他类似于x->x.foo(),也可以使用方法引用 (X x,V v,R r)-> x.foo(v,r)。

方法引用通过方法的名字来指向一个方法。

方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

方法引用使用一对冒号 ::

1）构造器引用：Class::new

具体使用哪种构造器是看真正执行时传多少个值，而且需要与构造器的参数列表数量匹配，从而对应执行匹配的构造器。

2）静态方法引用：ClassName::staticMethod

Lambda体中调用方法的参数列表与返回值类型要与函数式接口中抽象方法的函数列表与返回值类型保持一致！

3）对象方法引用：objectName::instanceMethod

Lambda体中调用方法的参数列表与返回值类型要与函数式接口中抽象方法的函数列表与返回值类型保持一致！

4）实例方法引用：ClassName::instanceMethod

第2、3种方式类似，等同于把lambda表达式的参数直接当成instanceMethod|staticMethod的参数来调用。

比如System.out::println等同于x->System.out.println(x)；

最后一种方式，等同于把lambda表达式的第一个参数当成instanceMethod的目标对象，其他剩余参数当成该方法的参数。

比如String::toLowerCase等同于String x->x.toLowerCase()。

可以这么理解，前两种是将传入对象当参数执行方法，后一种是调用传入对象的方法。

注意:

1. Lambda体中调用方法的参数列表与返回值类型，要与函数式接口中抽象方法的函数列表和返回值类型保存一致

2.若Lambda参数列表中的第一个参数是实例方法的调用者，而第二个参数是实例方法的参数时，可以使用ClassName::method

3. 需要调用的构造器方法与函数式接口中抽象方法的参数列表保持一致。

测试代码：

自定义函数接口：

@FunctionalInterface

**interface** Supplier<T>{

T get();

}

@FunctionalInterface

**interface** Supplier2<T,R,V>{

T get(R r,V v);

}

**public** **class** Test6LambdaMethodConDemo1 {

**private** String name;

**private** **int** age;

**public** Test6LambdaMethodConDemo1(){

System.***out***.println("无参执行");

}

**public** Test6LambdaMethodConDemo1(String name, **int** age) {

**this**.name = name;

**this**.age = age;

System.***out***.println("有参执行:"+name+","+age);

}

//测试构造器引用1

**public** **static** Test6LambdaMethodConDemo1 create(Supplier<Test6LambdaMethodConDemo1> s){

**return** s.get();//使用的哪种构造器看这里，lambda表达式执行的时候的传参是无参

}

//测试构造器引用2

**public** **static** Test6LambdaMethodConDemo1 create2(Supplier2<Test6LambdaMethodConDemo1,String,Integer> s,String name,**int** age){

**return** s.get(name,age);//使用的哪种构造器看这里，lambda表达式执行的时候的传参是两参

}

//测试静态方法引用

**public** **static** **void** method3(Test6LambdaMethodConDemo1 t5){

System.***out***.println("method3:"+t5);

}

//测试有参实例方法引用

**public** **void** method4(Test6LambdaMethodConDemo1 t5){

System.***out***.println("method4:"+t5);

}

//测试无参实例方法引用

**public** **void** method5(){

System.***out***.println("method5:"+**this**);

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** **static** **void** main(String[] args) {

//构造器引用

Test6LambdaMethodConDemo1 t1 = Test6LambdaMethodConDemo1.*create*(Test6LambdaMethodConDemo1::**new**);

Test6LambdaMethodConDemo1 t2 = Test6LambdaMethodConDemo1.*create2*(Test6LambdaMethodConDemo1::**new**,"zs",18);

List<Test6LambdaMethodConDemo1> list = Arrays.*asList*(t1);

//静态方法引用

list.forEach(Test6LambdaMethodConDemo1::*method3*);

//实例对象方法的实现

list.forEach(t1::method4);

//不需要创建对象，也能调用的，相当于将参数作为调用者

list.forEach(Test6LambdaMethodConDemo1::method5);

list.forEach(System.***out***::println);

}

}

运行结果：

无参执行

有参执行:zs,18

method3:com.briup.test3MethodDemo.Test6LambdaMethodConDemo1@6ce253f1

method4:com.briup.test3MethodDemo.Test6LambdaMethodConDemo1@6ce253f1

method5:com.briup.test3MethodDemo.Test6LambdaMethodConDemo1@6ce253f1

com.briup.test3MethodDemo.Test6LambdaMethodConDemo1@6ce253f1

### 2.6 常用函数接口

函数接口是只有一个抽象方法的接口，用作Lambda表达式的类型，我们可以将lambda表达式当作任意只包含一个抽象方法的接口类型。

使用Java编程，总会遇到很多的函数接口，单Java开发工具包（JDK）提供的一组核心函数接口会频繁出现，如以下一些常用的函数接口：

|  |  |  |
| --- | --- | --- |
| 接口 | 参数 | 返回类型 |
| Predicate<T> | T | boolean |
| Consumer<T> | T | void |
| Function<T,R> | T | R |
| Supplier<T> | None | T |
| BinaryOperator<T> | <T,T> | T |

### 2.61 java.util.function.Predicate<T>接口

测试代码：

**public** **class** Test6Predicate {

**public** **static** **void** filter(List<String> list, Predicate<String> condition) {

**for**(String name: list) {

**if**(condition.test(name)) {

System.***out***.println(name + " ");

}

}

}

}

在测试方法中：

List<String> list = Arrays.*asList*("yewenjie","liucixin","yuntianming","luoji","chenxin");

//全都显示

*filter*(list, name->**true**);

//全都不显示

*filter*(list, name->**false**);

//开头是y的语言

*filter*(list,name->name.startsWith("y"));

//5结尾的

*filter*(list,name->name.endsWith("5"));

//显示名字长度大于4

*filter*(list,name->name.length()>4);

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

//名字以y开头并且长度大于4的

Predicate<String> c1 = name->name.startsWith("y");

Predicate<String> c2 = name->name.length()>4;

*filter*(list,c1.and(c2));

//名字不是以y开头

Predicate<String> c3 = (name)->name.startsWith("y");

*filter*(list,c3.negate());

//名字以y开头或者长度小于4的

Predicate<String> c4 = name->name.startsWith("y");

Predicate<String> c5 = name->name.length()<4;

*filter*(list,c4.or(c5));

//也可以直接使用Predicate接口中的静态方法

//名字为Java的

*filter*(list,Predicate.*isEqual*("luoji"));

//判断俩个字符串是否相等

**boolean** test = Predicate.*isEqual*("hello").test("world");

System.***out***.println(test);

### 2.62 java.util.function.Function<T, R> 接口

Function接口有一个参数并且返回一个结果,并附带了一些可以和其他函数组合的默认方法

compose方法表示在某个方法之前执行

andThen方法表示在某个方法之后执行

注意:compose和andThen方法调用之后都会把对象自己本身返回,这可以方便链式编程

测试代码：

**public** **class** Test7Function {

//静态内部类

**private** **static** **class** Student{

**private** String name;

**public** Student(String name){

**this**.name = name;

}

**public** String getName() {

**return** name;

}

}

**public** **static** **void** main(String[] args) {

/\*用户注册输入一个名字tom\*/

String name = "tom";

/\*使用用户的输入的名字创建一个对象\*/

Function<String, Student> f1 =(s)->**new** Student(s);

//注意上面的代码也可以写出这样,引用类中的构造器

//Function<String, Student> f1 =Student::new;

Student stu1 = f1.apply(name);

System.***out***.println(stu1.getName());

/\*需求改变,使用name创建Student对象之前需要给name加一个前缀\*/

Function<String,String> before = (s)->"briup\_"+s;

//表示f1调用之前先执行before对象的方法,把before对象的方法返回结果作为f1对象方法的参数

Student stu2 = f1.compose(before).apply(name);

System.***out***.println(stu2.getName());

/\*获得创建好的对象中的名字的长度\*/

Function<Student,Integer> after = (stu)->stu.getName().length();

//before先调用方法,结果作为参数传给f1来调用方法,结果再作为参数传给after,结果就是我们接收的数据

**int** len = f1.compose(before).andThen(after).apply(name);

System.***out***.println(len);

}

}

运行结果：

tom

briup\_tom

9

### 2.63 java.util.function.Supplier<T>接口

Supplier接口返回一个任意范型的值,和Function接口不同的是该接口没有任何参数

@FunctionalInterface

**public** **interface** Supplier<T> {

T get();

}

测试代码：

@Test

**public** **void** test1(){

//生成一个八位的随机字符串

Supplier<String> f = ()->{

String base = "abcdefghijklmnopqrstuvwxyz0123456789";

Random random = **new** Random();

StringBuffer sb = **new** StringBuffer();

**for** (**int** i = 0; i < 8; i++) {

//生成[0,base.length)之间的随机数

**int** number = random.nextInt(base.length());

sb.append(base.charAt(number));

}

**return** sb.toString();

};

System.***out***.println(f.get());

}

运行结果：

53mwlfp0

**public** **void** test2(Supplier<List<String>> su){

System.***out***.println(su.get());

}

**public** **static** **void** main(String[] args) {

Test8Supplier t = **new** Test8Supplier();

t.test2(ArrayList::**new**);

}

运行结果：

[]

### 2.64 java.util.function.Consumer<T>接口

Consumer接口接收一个任意泛型的值,和Function接口不同的是该接口没有任何值，也称为消费函数。

@FunctionalInterface

**public** **interface** Consumer<T> {

**void** accept(T t);

**default** Consumer<T> andThen(Consumer<? **super** T> after) {

Objects.*requireNonNull*(after);

**return** (T t) -> { accept(t); after.accept(t); };

}

}

测试代码：

**public** **class** Test9Consumer {

//静态内部类

**private** **static** **class** Student{

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

**public** **static** **void** main(String[] args) {

Student s = **new** Student();

s.setName("tom");

Consumer<Student> first = stu->{

System.***out***.println("我要第一个执行,把name值给改了");

stu.setName("zhangsan");

};

Consumer<Student> then = stu->{

System.***out***.println("我紧跟着执行,输出的name值为:"+stu.getName());

};

first.andThen(then).accept(s);

}

}

运行结果：

我要第一个执行,把name值给改了

我紧跟着执行,输出的name值为:zhangsan

注:可以根据具体的功能需求,来自定义出自己的函数式接口

2.7 java.util.Optional<T>类

Optional 不是接口而是一个类,这是个用来防止NullPointerException异常的辅助类型

Optional 被定义为一个简单的容器,其值可能是null或者不是null。在Java8之前一般某个函数应该返回非空对象但是偶尔却可能返回了null,而在Java8中,不推荐你返回null而是返回Optional。

这是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true,调用get()方法会返回该对象。

测试代码：

\*of方法 为非null的值创建一个Optional对象\*/

//需要注意的是,创建对象时传入的参数不能为null。

//如果传入参数为null,则抛出NullPointerException 。

Optional<String> op1 = Optional.*of*("hello");

/\*ofNullable方法 为指定的值创建一个Optional,如果指定的值为null,则返回一个空的Optional。\*/

//ofNullable与of方法相似,唯一的区别是可以接受参数为null的情况

Optional<String> op2 = Optional.*ofNullable*(**null**);

/\*isPresent方法 如果值存在返回true,否则返回false。\*/

/\*get方法 如果Optional有值则将其返回,否则抛出NoSuchElementException。\*/

**if**(op1.isPresent()){

System.***out***.println(op1.get());//hello

}

**if**(op2.isPresent()){

System.***out***.println(op2.get());//不执行

}

/\*ifPresent方法 如果Optional实例有值则为其调用consumer接口中的方法,否则不做处理\*/

//consumer接口中的方法只有参数没有返回值

//public void accept(T t);

op1.ifPresent(str->System.***out***.println(str));//hello

op2.ifPresent(str->System.***out***.println(str));//这个不执行 因为op2里面的值是null

/\*orElse方法 如果有值则将其返回,否则返回指定的其它值。\*/

System.***out***.println(op1.orElse("如果op1中的值为null则返回这句话"));//hello

System.***out***.println(op2.orElse("如果op2中的值为null则返回这句话"));//如果op2中的值为null则返回这句话

/\*orElseGet方法 orElseGet与orElse方法类似,区别在于得到的默认值的方式不同。orElse方法将传入的字符串作为默认值,orElseGet方法可以接受Supplier接口的实现用来生成默认值。\*/

//Supplier接口中的方法没有参数但是有返回值

//public T get();

System.***out***.println(op1.orElseGet(()->"自己定义的返回值"));//hello

System.***out***.println(op2.orElseGet(()->"自己定义的返回值"));//自己定义的返回值

/\*orElseThrow方法 如果有值则将其返回,否则抛出supplier接口创建的异常。\*/

//在orElseThrow中我们可以传入一个lambda表达式或方法,如果值不存在来抛出异常。

//orElseThrow方法的声明如下 所有只能返回一个Throwable类型对象

//public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X

**try** {

System.***out***.println(op1.orElseThrow(Exception::**new**));hello

//System.out.println(op2.orElseThrow(NullPointerException::new));这个会抛出异常

} **catch** (Exception e) {

e.printStackTrace();

}

/\*map方法 如果有值,则调用mapper的函数处理并得到返回值\*/

//返回值并且依然Optional包裹起来,其泛型和你返回值的类型一致

//public<U> Optional<U> map(Function<? super T, ? extends U> mapper)

Optional<Integer> map1 = op1.map(str->1);

System.***out***.println(map1.orElse(0));//1

Optional<Double> map2 = op2.map(str->1.2);

System.***out***.println(map2.orElse(0D));//0.0

/\*flatMap方法 如果有值,则调用mapper的函数返回Optional类型返回值,否则返回空Optional。\*/

//flatMap与map方法类似,区别在于flatMap中的mapper返回值必须是Optional。

//调用结束时,flatMap不会对结果用Optional封装

//需要我们自己把返回值封装为Optional

//public<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper);

Optional<String> flatMap = op1.flatMap(str->Optional.*of*(str+"\_briup"));

System.***out***.println(flatMap.get());//hello\_briup

//编译出错,因为lambda表示的函数的返回类型不对

//op1.flatMap(str->"");

/\*filter方法 如果有值并且满足断言条件返回包含该值的Optional,否则返回空Optional。\*/

//public Optional<T> filter(Predicate<? super T> predicate);

op1 = op1.filter(str->str.length()<10);

System.***out***.println(op1.orElse("值为null"));//hello

op1 = op1.filter(str->str.length()>10);

System.***out***.println(op1.orElse("值为null"));//值为null

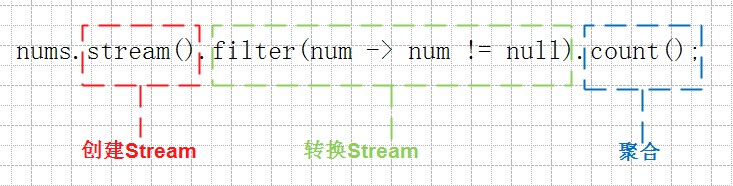
# 第三章 Stream流

Java 8 API添加了一个新的抽象称为流Stream，可以让你以一种声明的方式处理数据。

Stream API可以极大提高Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。Java8中的Stream是对集合(Collection)对象功能的增强,它专注于对集合对象进行各种非常便利、高效的聚合操作,或者大批量数据操作。Stream的API结合Lambda表达式,极大的提高编程效率和程序可读性。同时它提供串行和并行两种模式进行操作

这种风格将要处理的元素集合看作一种流， 流在管道中传输， 并且可以在管道的节点上进行处理， 比如筛选， 排序，聚合等。

Stream操作分为中间操作或者最终操作两种,最终操作返回一特定类型的计算结果,而中间操作返回Stream本身,这样你就可以将多个操作依次串起来(链式编程)。元素流在管道中经过中间操作（intermediate operation）的处理，最后由最终操作(terminal operation)得到前面处理的结果。



## 3.1 如何生成Stream

1）通过Stream接口的静态工厂方法（注意：Java8里接口可以带静态方法）；

1.1 of方法：有两个overload方法，一个接受变长参数，一个接口单一值,都是返回Stream对象

Stream<String> stream11 = Stream.*of*("hello");

stream11.forEach(System.***out***::println);

Stream<Object> stream12 = Stream.*of*("hello","world",123);

stream12.forEach(System.***out***::println);

1.2 generator方法：生成一个无限长度的Stream，其元素的生成是通过给定的Supplier（这个接口可以看成一个对象的工厂，每次调用返回一个给定类型的对象）

Stream<Double> stream21 = Stream.*generate*(**new** Supplier<Double>() {

@Override

**public** Double get() {

**return** Math.*random*();

}

});

stream21.limit(10).forEach(System.***out***::println);

Stream<Double> stream22 = Stream.*generate*(()->Math.*random*());

stream22.limit(10).forEach(System.***out***::println);

Stream<Double> stream23 = Stream.*generate*(Math::*random*);

stream23.limit(10).forEach(System.***out***::println);

三条语句的作用都是一样的，只是使用了lambda表达式和方法引用的语法来简化代码。每条语句其实都是生成一个无限长度的Stream，其中值是随机的。这个无限长度Stream是懒加载，一般这种无限长度的Stream都会配合Stream的limit()方法来用。

1.3 iterate方法

也是生成无限长度的Stream，和generator不同的是，其元素的生成是重复对给定的种子值(seed)调用用户指定函数来生成的。其中包含的元素可以认为是：seed，f(seed),f(f(seed))无限循环

Stream<Integer> stream31 = Stream.*iterate*(1, i->i+1);

stream31.limit(10).forEach(System.***out***::println);

System.***out***.println("\*\*\*");

对于基本数值类型,目前有三种对应的Stream包装类型:IntStream、LongStream、DoubleStream。

当然也可以用泛型:Stream<Integer>、Stream<Long> 、Stream<Double>,但是自动拆箱装箱会很耗时,所以特别为这三种基本数值型提供了对应的Stream。

Java8中还没有提供其它基本类型数值的Stream

IntStream is = Stream.*iterate*(1, i->i+1).limit(10).mapToInt(x->x);

IntSummaryStatistics iss = is.summaryStatistics();

System.***out***.println(iss.getMax());

System.***out***.println(iss.getMin());

System.***out***.println(iss.getSum());

System.***out***.println(iss.getCount());

System.***out***.println(iss.getAverage());

1. 通过Collection接口的默认方法–stream()

查看Java doc就可以发现Collection接口有一个stream方法，把一个Collection对象转换成Stream所以其所有子类都都可以获取对应的Stream对象。

List<String> list1 = Arrays.*asList*("hello","world","hah");

Stream<String> stream41 = list1.stream();

Stream<Integer> smap = stream41.map(x->x.length());

smap.forEach(System.***out***::println);

1. 通过数组创建Stream

@Test

**public** **void** method1\_Array(){

//3.通过数组创建stream

String[] s1 = "hello".split("");

Stream<String> stream42 = Arrays.*stream*(s1);

stream42.forEach(System.***out***::println);

}

1. 数值Stream的构建:

IntStream stream1 = IntStream.*of*(**new** **int**[]{1, 2, 3});

stream1.forEach(System.***out***::println);

IntStream stream2 = IntStream.*range*(1, 3);//[1,3)，不包含结束

stream2.forEach(System.***out***::println);

IntStream stream3 = IntStream.*rangeClosed*(1, 3);//[1,3]，包含结束

stream3.forEach(System.***out***::println);

1. IO构建

文件内容test.txt:

first

second

third

BufferedReader br =

**new** BufferedReader(**new** FileReader("src/com/briup/test4StreamDemo/test.txt"));

Stream<String> lines = br.lines();

lines.forEach(System.***out***::println);

## 3.2 Stream操作

当把一个数据结构包装成Stream后,就要开始对里面的元素进行各类操作了。常见的操作可以归类如下。

Intermediate:中间操作

map (mapToInt, flatMap 等)、filter、distinct、sorted、peek、limit、skip、parallel、sequential、unordered

Terminal: 最终操作

forEach、forEachOrdered、toArray、reduce、collect、min、max、count、anyMatch、allMatch、noneMatch、findFirst、findAny、iterator

Short-circuiting: 短路操作

anyMatch、allMatch、noneMatch、findFirst、findAny、limit

像filter这样只描述stream，最终不产结果值的方法叫做惰性求值（中间操作）方法；而像count这样最终会从stream产生值的方法叫做及早求值（最终操作）方法

测试操作的特性：看方法的返回值，如果返回值是Stream，就是惰性求值，如果返回值是另一个值或者为空，就是及早求值

中间操作会返回一个新的steam——执行一个中间操作（例如filter）并不会执行实际的过滤操作，而是创建一个新的steam，并将原steam中符合条件的元素放入新创建的steam。

最终操作（例如forEach或者sum），会遍历steam并得出结果或者附带结果；在执行晚期操作之后，steam处理线已经处理完毕，就不能使用了。在几乎所有情况下，晚期操作都是立刻对steam进行遍历。

steam的另一个价值是创造性地支持并行处理（parallel processing）

使用这些操作的理想方式就是形成一个惰性求值的链，最后用一个及早求值的操作返回想要的结果。

### 3.21 filter

filter对于Stream中包含的元素使用给定的过滤函数进行过滤操作，新生成的Stream只包含符合条件的元素。

通过一个predicate接口来过滤并只保留符合条件的元素,该操作属于中间操作,所以我们可以在过滤后的结果来应用其他Stream操作(比如forEach)。forEach需要一个函数来对过滤后的元素依次执行。forEach是一个最终操作,所以我们不能在forEach之后来执行其他Stream操作

Stream<T> filter(Predicate<? **super** T> predicate);

测试代码：

@Test

**public** **void** method2\_filter(){

List<String> list = Arrays.*asList*("hello","world","yewenjian","luoji","huangtian");

Stream<String> stream2 = list.stream().filter(

(element)->{System.***out***.println(element+"..");**return** element.length()>5;}

);

System.***out***.println(stream2);//filter只是中间操作，不是真正执行，没有打印语句

stream2.forEach(System.***out***::println);//最终操作，真正执行

//原集合不改变

**for** (String string : list) {

System.***out***.println("\*\*"+string);

}

stream2.forEach(System.***out***::println);//第二次打印报错 java.lang.IllegalStateException: stream has already been operated upon or closed

}

### 3.22 map

map:对于Stream中包含的元素使用给定的转换函数进行转换操作，新生成的Stream只包含转换生成的元素。这个方法有三个对于原始类型的变种方法，分别是：mapToInt，mapToLong和mapToDouble。这三个方法也比较好理解，比如mapToInt就是把原始Stream转换成一个新的Stream，这个新生成的Stream中的元素都是int类型。之所以会有这样三个变种方法，可以免除自动装箱/拆箱的额外消耗。

@Test

**public** **void** method\_map(){

List<String> list = Arrays.*asList*("hello","world","yewenjian","luoji","huangtian");

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

(string-> string.toUpperCase()

));

Stream<String> map2 = list.stream().map(String::toUpperCase);//传入的String参数当做实例方法的调用者,返回一个结果

map.forEach(System.***out***::println);

map2.forEach(System.***out***::println);

}

Filter+map:

@Test

**public** **void** method\_filterMap(){

// 多个惰性求值操作顺序执行，不多次迭代

Stream.*of*("2abc","1abc","abc1").filter(

element->{System.***out***.println(element+"-filter-"); **return** Character.*isDigit*(element.charAt(0));}

).map(element->{System.***out***.println(element+"-map-"); **return** element.length();}).limit(1).forEach(System.***out***::println);

//多次操作多个stream对象

Stream<String> s1 = Stream.*of*("2abc","1abc","abc1").filter(

element->{System.***out***.println(element+"-filter2-"); **return** Character.*isDigit*(element.charAt(0));}

);

Stream<Integer> s2 = s1.map(element->{System.***out***.println(element+"-map2-"); **return** element.length();});

System.***out***.println(s2.findFirst());

}

运行结果：

2abc-filter-

2abc-map-

4

2abc-filter2-

2abc-map2-

Optional[4]

### 3.23 sorted

排序是一个中间操作,返回的是排序好后的Stream。如果你不指定一个自定义的Comparator则会使用默认排序。

对 Stream 的排序通过 sorted 进行,它比数组的排序更强之处在于你可以首先对 Stream 进行各类操作: map、filter、limit、skip 甚至 distinct 来减少元素数量后,再排序,这能帮助程序明显缩短执行时间。

测试代码：

@Test

**public** **void** method\_sorted(){

List<String> list1 = Arrays.*asList*("hello","hello2","yewenjian","luoji","huangtian");

//默认排序

list1.stream().map(String::length).sorted().forEach(System.***out***::println);

//倒序

list1.stream().map(String::length).sorted((x,y)->y-x).forEach(System.***out***::println);

//通过Comparator的静态方法返回比较器 list1.stream().filter(x->x.length()>5).sorted(Comparator.*comparing*(String::length)).forEach(System.***out***::println);

}

3.25 peek

生成一个包含原Stream的所有元素的新Stream，同时会提供一个消费函数（Consumer实例），新Stream每个元素被消费的时候都会执行给定的消费函数；

**public** **void** method\_peek(){

List<String> list1 = Arrays.*asList*("hello","world","yewenjian","luoji","huangtian");

//处理之前进行一步查看原内容的操作，不影响之后的操作

list1.stream().map(String::toUpperCase).peek(System.***out***::println).map(element->element.length()).forEach(System.***out***::println);;

}

运行结果：

HELLO

5

WORLD

5

YEWENJIAN

9

LUOJI

5

HUANGTIAN

9

### 3.26 limit

对一个Stream进行截断操作，获取其前N个元素，如果原Stream中包含的元素个数小于N，那就获取其所有的元素。

@Test

**public** **void** method\_peek(){

List<String> list1 = Arrays.*asList*("hello","world","yewenjian","luoji","huangtian");

list1.stream().map(String::toUpperCase).limit(2).forEach(System.***out***::println);

}

运行结果：

HELLO

WORLD

### 3.27 skip

返回一个丢弃原Stream的前N个元素后剩下元素组成的新Stream，如果原Stream中包含的元素个数小于N，那么返回空Stream；

@Test

**public** **void** method\_skip(){

List<String> list1 = Arrays.*asList*("hello","world","yewenjian","luoji","huangtian");

list1.stream().map(String::toUpperCase).skip(2).forEach(System.***out***::println);

}

运行结果：

YEWENJIAN

LUOJI

HUANGTIAN

### 3.28 distinct

对于Stream中包含的元素进行去重操作（去重逻辑依赖元素的equals方法），新生成的Stream中没有重复的元素；

@Test

**public** **void** method\_distinct(){

//distinct: 对于Stream中包含的元素进行去重操作（去重逻辑依赖元素的equals方法），新生成的Stream中没有重复的元素；

List<String> list1 = Arrays.*asList*("hello","hello","yewenjian","luoji","huangtian");

list1.stream().distinct().forEach(System.***out***::println);

}

运行结果：

hello

yewenjian

luoji

huangtian

### 3.29 foreach

forEach方法前面已经用了好多次，其用于遍历Stream中的所元素，避免了使用for循环，让代码更简洁，逻辑更清晰

@Test

**public** **void** method\_foreach(){

List<String> list1 = Arrays.*asList*("hello","hello","yewenjian","luoji","huangtian");

list1.stream().filter(element->element.length()>5).forEach(element->System.***out***.println("长度大于5的字符串："+element));

}

运行结果：

长度大于2的字符串：yewenjian

长度大于2的字符串：huangtian

### 3.210 count

返回Stream中元素的个数

@Test

**public** **void** method\_count(){

List<String> list1 = Arrays.*asList*("hello","hello","yewenjian","luoji","huangtian");

**long** count = list1.stream().filter(element->element.length()>5).count();

System.***out***.println("长度大于5的字符串数量："+count);

}

运行结果：

长度大于5的字符串数量：2

### 3.211 sum

返回Stream中元素（数值）的和

@Test

**public** **void** method\_sum(){

List<String> list1 = Arrays.*asList*("hello","hello","yewenjian","luoji","huangtian");

**long** count = list1.stream().map(element->element.length()).mapToInt(x->x).sum();

System.***out***.println("长度大于5的字符串数量："+count);

}

运行结果：

长度大于5的字符串数量：33

### 3.212 max

返回Stream中符合比较条件的最大的值

@Test

**public** **void** method\_max(){

List<String> list1 = Arrays.*asList*("hello","hello","yewenjian","luoji","huangtian");

Optional<String> max = list1.stream().filter(element->element.length()>=5).max(Comparator.*comparing*(str->str.length()));

System.***out***.println("长度最长的字符串："+max.get());

}

运行结果：

长度最长的字符串：yewenjian

### 3.213 min

返回Stream中符合比较条件的最小的值

@Test

**public** **void** method\_min(){

List<String> list1 = Arrays.*asList*("hello","hello","yewenjian","luoji","huangtian");

Optional<String> max = list1.stream().filter(element->element.length()>=5).min(Comparator.*comparing*(str->str.length()));

System.***out***.println("长度最短的字符串："+max.get());

}

运行结果：

长度最短的字符串：hello

### 3.214 reduce

可以实现从一组值中生成一个值，上述的sum,count,min,max方法因为常用被纳入标准库中，实际上都是reduce操作

@Test

**public** **void** method\_reduce(){

/\*\*

\* 第一种：Optional<T> reduce(BinaryOperator<T> accumulator);

\* BinaryOperator接口，可以看到reduce方法接受一个函数，这个函数有两个参数，

\* 第一个参数是上次函数执行的返回值（也称为中间结果），第二个参数是stream中的元素，

\* 这个函数把这两个值相加，得到的和会被赋值给下次执行这个函数的第一个参数。

\* 要注意的是：第一次执行的时候第一个参数的值是Stream的第一个元素，第二个参数是Stream的第二个元素。

\* 这个方法返回值类型是Optional，

\*

\* \*/

List<Integer> list = Arrays.*asList*(1,2,3,4,5);

Optional<Integer> sum1 = list.stream().reduce((sum,item)->{

System.***out***.println("sum:"+sum);

System.***out***.println("item:"+item);

sum+=item;

System.***out***.println("sum+:"+sum);

**return** sum;

});

System.***out***.println(sum1);

运行结果：

sum:1

item:2

sum+:3

sum:3

item:3

sum+:6

sum:6

item:4

sum+:10

sum:10

item:5

sum+:15

Optional[15]

/\*\*

\* 第二种： T reduce(T identity, BinaryOperator<T> accumulator);

\* 这个定义上上面已经介绍过的基本一致，都会接受一个BinaryOperator函数接口

\* 不同的是：它允许用户提供一个循环计算的初始值，如果Stream为空，就直接返回该值。

\* 而且这个方法不会返回Optional，因为其不会出现null值

\*

\* \*/

Integer sum2 = list.stream().reduce(0,(sum,item)->{

System.***out***.println("sum:"+sum);

System.***out***.println("item:"+item);

sum+=item;

System.***out***.println("sum+:"+sum);

**return** sum;

});

System.***out***.println(sum2);

}

运行结果：

sum:0

item:1

sum+:1

sum:1

item:2

sum+:3

sum:3

item:3

sum+:6

sum:6

item:4

sum+:10

sum:10

item:5

sum+:15

15

### 3.215 collect

Collection, Collections, collect, Collector, Collectors

Collection是Java集合的祖先接口。

Collections是java.util包下的一个工具类，内涵各种处理集合的静态方法。

java.util.stream.Stream#collect(java.util.stream.Collector<? super T,A,R>)是Stream的一个函数，负责收集流。

java.util.stream.Collector 是一个收集函数的接口, 声明了一个收集器的功能。

[java](eclipse-javadoc:%E2%98%82=NewFeature/I:\/WinJDK8\/JDK\/jre\/lib\/rt.jar%3Cjava).[util](eclipse-javadoc:%E2%98%82=NewFeature/I:\/WinJDK8\/JDK\/jre\/lib\/rt.jar%3Cjava.util).[stream](eclipse-javadoc:%E2%98%82=NewFeature/I:\/WinJDK8\/JDK\/jre\/lib\/rt.jar%3Cjava.util.stream).Collectors则是一个收集器的工具类，内置了一系列收集器实现。

收集器的作用

你可以把Java8的流看做花哨又懒惰的数据集迭代器。他们支持两种类型的操作：中间操作(e.g. filter, map)和

终端操作(如count, findFirst, forEach, reduce). 中间操作可以连接起来，将一个流转换为另一个流。

这些操作不会消耗流，其目的是建立一个流水线。与此相反，终端操作会消耗类，产生一个最终结果。

collect就是一个归约操作，就像reduce一样可以接受各种做法作为参数，将流中的元素累积成一个汇总结果。

具体的做法是通过定义新的Collector接口来定义的。

@Test

**public** **void** method\_collect(){

//转换成其他集合

List<String> list1 = Arrays.*asList*("hello","hello2","yewenjian","luoji","huangtian");

List<Integer> collect11 = list1.stream().map(String::length).collect(Collectors.*toList*());

LinkedList<Integer> collect12 = list1.stream().map(String::length).collect(Collectors.*toCollection*(LinkedList::**new**));

Set<Integer> collect21 = list1.stream().map(String::length).collect(Collectors.*toSet*());

TreeSet<Integer> collect22 = list1.stream().map(String::length).collect(Collectors.*toCollection*(TreeSet::**new**));

//key重复将抛异常

Map<String, Integer> collect31 = list1.stream().collect(Collectors.*toMap*(x->x, x->x.length()));

//[5, 6, 9, 5, 9]

System.***out***.println(collect11);

//[5, 6, 9, 5, 9]

System.***out***.println(collect12);

//[5, 6, 9]

System.***out***.println(collect21);

//[5, 6, 9]

System.***out***.println(collect22);

//{huangtian=9, hello2=6, hello=5, luoji=5, yewenjian=9}

System.***out***.println(collect31);

//转换成值，返回长度最小的字符串数量

Optional<Integer> res1 = list1.stream().map(String::length).collect(Collectors.*minBy*((o1,o2)->o1-o2));

//转换成值，返回长度最小的字符串

//指明要比较的对象执行什么方法，取其结果值进行compareTo比较

Optional<String> res2 = list1.stream().collect(Collectors.*minBy*(Comparator.*comparing*(String::length)));

//Optional[5]

System.***out***.println(res1);

//Optional[hello]

System.***out***.println(res2);

//集合中所有字符串长度和

Integer res5 = list1.stream().collect(Collectors.*summingInt*(x->x.length()));

//集合中所有字符串数量

Long res6 = list1.stream().collect(Collectors.*counting*());

//集合中所有字符串平均值

Double res7 = list1.stream().collect(Collectors.*averagingDouble*(x->x.length()));

System.***out***.println(res5);//34

System.***out***.println(res6);//5

System.***out***.println(res7);//6.8

//整合总结信息

IntSummaryStatistics res8 = list1.stream().map(String::length).collect(Collectors.*summarizingInt*(t->t));

System.***out***.println(res8.getMax());//9

System.***out***.println(res8.getMin());//5

System.***out***.println(res8.getCount());//5

System.***out***.println(res8.getSum());//34

System.***out***.println(res8.getAverage());//6.8

//数据分块,使用true/false作为key进行分组

Map<Boolean, List<String>> part1 = list1.stream().collect(Collectors.*partitioningBy*(element->element.length()>5));

List<String> list2 = Arrays.*asList*("hello","hello","yewenjian","luoji","huangtian");

//数据分组，使用任意值作为key进行分组,value为List集合，集合中存放的是原集合中的值

Map<String, List<String>> part2 = list2.stream().collect(Collectors.*groupingBy*(element->element));

Map<Character, List<String>> part3 = list2.stream().collect(Collectors.*groupingBy*(element->element.charAt(1)));

//数据分组，使用任意值作为key进行分组(使用原集合中字符串的首字母),修改value(value为List集合，集合中存放的是原集合中的字符串的大写)

Map<Character, List<String>> part4 = list2.stream().collect(Collectors.*groupingBy*(element->element.charAt(1),Collectors.*mapping*(String::toUpperCase,Collectors.*toList*())));

//数据分组，使用任意值作为key进行分组(使用原集合中字符串的首字母),修改value(value为List集合，集合中存放的是原集合中的字符串的首字母)

Map<Character, List<Character>> part5 = list2.stream().collect(Collectors.*groupingBy*(element->element.charAt(1),Collectors.*mapping*(element->element.charAt(0),Collectors.*toList*())));

//{false=[hello, luoji], true=[hello2, yewenjian, huangtian]}

System.***out***.println(part1);

//{huangtian=[huangtian], hello=[hello, hello], luoji=[luoji], yewenjian=[yewenjian]}

System.***out***.println(part2);

//{u=[luoji, huangtian], e=[hello, hello, yewenjian]}

System.***out***.println(part3);

//{u=[LUOJI, HUANGTIAN], e=[HELLO, HELLO, YEWENJIAN]}

System.***out***.println(part4);

//{u=[l, h], e=[h, h, y]}

System.***out***.println(part5);

//字符串:原集合中字符串拼接

String str1 = list1.stream().collect(Collectors.*joining*());

//字符串:原集合中字符串使用,拼接

String str2 = list1.stream().collect(Collectors.*joining*(","));

//字符串:原集合中字符串使用,做间隔，[为前缀，]为后缀拼接

String str3 = list1.stream().collect(Collectors.*joining*(",","[","]"));

//hellohello2yewenjianluojihuangtian

System.***out***.println(str1);

//hello,hello2,yewenjian,luoji,huangtian

System.***out***.println(str2);

//[hello,hello2,yewenjian,luoji,huangtian]

System.***out***.println(str3);

}

### 3.216 findFirst

总是返回 Stream 的第一个元素,或者空,返回值类型:Optional。

如果集中什么都没有,那么list.stream().findFirst()返回一个Optional对象,但是里面封装的是一个null。

注:只是返回并没有移除

测试代码:

@Test

**public** **void** method\_findFirst(){

List<String> list1 = Arrays.*asList*("hello","hello2","yewenjian","luoji","huangtian");

Optional<String> findFirst = list1.stream().findFirst();

System.***out***.println(findFirst);//Optional[hello]

}

### 3.217 allMatch

所有元素匹配成功才返回true 否则返回false

@Test

**public** **void** method\_allMatch(){

List<String> list1 = Arrays.*asList*("hello","hello2","yewenjian","luoji","huangtian");

**boolean** allMatch = list1.stream().allMatch((x->x.length()>5));

System.***out***.println(allMatch);//false

}

短路操作：只执行一次就判断为false

### 3.218 anyMatch

任意一个匹配成功就返回true 否则返回false

@Test

**public** **void** method\_anyMatch(){

List<String> list1 = Arrays.*asList*("hello","hello2","yewenjian","luoji","huangtian");

**boolean** allMatch = list1.stream().anyMatch(x->{System.***out***.println(x); **return** x.length()>5;});

System.***out***.println(allMatch);//true

}

### 3.219 noneMatch

没有一个匹配的就返回true 否则返回false

@Test

**public** **void** method\_noneMatch(){

List<String> list1 = Arrays.*asList*("hello","hello2","yewenjian","luoji","huangtian");

**boolean** allMatch = list1.stream().noneMatch(x->{System.***out***.println(x); **return** x.length()>5;});

System.***out***.println(allMatch);//false

}

## 3.3 数据并行化

并行与并发：

并发:指应用能够交替执行不同的任务,其实并发有点类似于多线程的原理,多线程并非是同时执行多个任务,如果你开两个线程执行,就是在你几乎不可能察觉到的速度不断去切换这两个任务,已达到"同时执行效果",其实并不是的,只是计算机的速度太快,我们无法察觉到而已。

就类似于你,吃一口饭喝一口水,以正常速度来看,完全能够看的出来,当你把这个过程以n倍速度执行时..可以想象一下.

并行:指应用能够同时执行不同的任务,例:吃饭的时候可以边吃饭边打电话,这两件事情可以同时执行

两者区别:一个是交替执行,一个是同时执行。

简单理解：

你吃饭吃到一半，电话来了，你一直到吃完了以后才去接，这就说明你不支持并发也不支持并行。

你吃饭吃到一半，电话来了，你停了下来接了电话，接完后继续吃饭，这说明你支持并发。 （不一定是同时的）

你吃饭吃到一半，电话来了，你一边打电话一边吃饭，这说明你支持并行。

并发的关键是你有处理多个任务的能力，不一定要同时。 并行的关键是你有同时处理多个任务的能力。

并行化是指为了缩短任务执行时间，将一个任务分解成几部分，然后并行执行，这和顺序执行的任务量是一样的，区别就像用更多的马来拉车，花费的事件自然减少了。当需要在大量数据上执行相同的操作时，数据并行化很管用，她将问题分解为可在多块数据上求解的形式，然后对每块数据执行运算，最后将各数据块上得到的结果汇总，从而获得最终答案。

为什么要使用并行化：

现代主流的芯片厂商相对于提高cpu时钟频率来说，更多转向了多核处理器，这种变化影响到了软件设计，我们不能再依赖提升CPU的时钟频率来提高先有代码的计算能力，需要利用现代CPU的架构，而这唯一的办法就是编写并行化的代码，这类问题的求解时间取决于它可以被分解成几部分。

并行化操作流只需改变一个方法调用，如果已经有一个Stream对象，调用它的parallel方法就能让其拥有并行操作的能力，如果想从一个集合类创建一个流，调用parallelStream就能立即获得一个拥有并行能力的流

测试代码：

数据量不大的情况下，串行话代码速度和并行化代码速度相当，甚至更低

@Test

**public** **void** method\_parallel1(){

//10个不同的字符串

List<String> list1 = Arrays.*asList*("hello","world","yewenjian","luoji","huangtian","shiqiang","weide","yuntianming","chenxin","wangmiao");

//1纳秒\*10^9=1秒

**long** t0 = System.*nanoTime*();

//串行stream

// long count = list1.stream().map(String::toUpperCase).count();//76718209

//并行stream

**long** count = list1.stream().parallel().map(String::toUpperCase).count();//89539183

**long** t1 = System.*nanoTime*();

**long** time = t1 - t0;

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

System.***out***.println(time);

}

数据量增加到100万，并行化代码执行速度大约是串行话代码的2倍

@Test

**public** **void** method\_seri(){

//生成100万个不同的字符串放到集合中

**int** max = 1000000;

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

**for** (**int** i = 0; i < max; i++) {

UUID uuid = UUID.*randomUUID*();

list.add(uuid.toString());

}

//1纳秒\*10^9=1秒

**long** t0 = System.*nanoTime*();

//串行stream

// long count = list.stream().sorted().count();//1006592459

//并行stream

**long** count = list.parallelStream().sorted().count();//679122774

**long** t1 = System.*nanoTime*();

**long** time = t1 - t0;

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

System.***out***.println(time);

}

影响并行处理流是否比串行流快的一些因素：

1. 数据大小，只有数据足够大，每个数据处理管道花费的时间足够多时，并行化处理才有意义。
2. 源数据结构，通常是集合，数据源分割相对容易，这里的开销影响在管道中并行处理数据时能带来的性能提升
3. 装箱，处理基本类型比处理装箱类型要快
4. 核的数量。极端情况下只有一个核，因此完全没必要并行化，拥有的核越多，获得潜在性能提升的幅度就越大，在实践中，核的数量不单指机器上有多少核，更是指运行时你的机器能使用多少核。也就是说同时运行的其他进程也会影响性能。
5. 单元处理开销，比如数据大小，在并行执行花费的时间与分解合并操作开销之间的对比，花在每个元素身上的时间越长，并行操作带来的性能提升越明显。