# Jdk8

## Lambda

### 基本结构

（param1,param2,param3）-> {

//执行体

}

如果匿名内部类函数没有值，则括号不能省略

() -> {

//执行体

}

### 函数式接口定义：

1. 如果一个接口只有一个抽象方法，那么该接口就是一个函数式接口
2. 如果我们在某个接口上声明了FunctionalInterface注解，那么编译器就会按照函数式接口的定义来要求该接口
3. 如果某个接口只有一个抽象方法，但我们并没有给该接口声明FunctionalInterface注解，那么编译器依旧会将该接口，看作是函数式接口

### 函数式接口的实现方式：

\* <p>Note that instances of functional interfaces can be created with

\* lambda expressions, method references, or constructor references.

1. lambda expressions
2. method references
3. constructor references

### Consumer

函数式接口

没有返回值

意在消费掉

在Java中，lambda表达式是对象。在其他语言是函数。

lambda表达式的实例对象类型是特别的对象类型-函数式接口Functional interface

### Function

由一种类型转变为另外一种类型！

R apply(T t)

andThen(Function f)，变换后，追加结果

compose(Function f)，在数据开始变化之前，先进行转换处理

String compute = demo03.compute(12345, i -> i + "end");

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

//结果是：12345compose22compose11endandThen11andThen22

//执行顺序是：compose倒序(最后一个compose最先执行)，然后apply，最后依次执行andThen

//compute是自定义的方法，和API无关

**public** String compute(Integer str, Function<String, String> function) {

String apply = function.andThen(i -> i + "andThen11").andThen(i -> i + "andThen22").compose(i -> String.*valueOf*(i)+"compose11").compose(i -> String.*valueOf*(i)+"compose22").apply(str);

**return** apply;

}

#### compose方法的本质

compose存在3个对象

1. before匿名对象
2. Func的实现对象
3. Func经过compose的组合对象！

public interface Func<T, R> {

R apply(T t);

default <V> Func<V,R> compose(Func<V,T> before){

return new Func<V,R>(){// 1.首先创建一个独立的第三方组合对象！方法实现是该对象传入一个入参，经过两次转换后，返回出参！

@Override

public R apply(V v) {

T t = before.apply(v);// 2.接收到v后，先经过before转换

R r = Func.this.apply(t);// 3.拿到before转换后的值t，再进行Func的apply转换（t到r的转换）

return r;

}

};

}

}

#### 简化后

? super V 因为独立对象传入的V是入参，经过第一个before处理时，before不能接受V的子类！！必须保证处理V的父类！比如外部给我一个parent，before不能转换成处理child！

? extends T 转换的结果，却可以是子类，因为是出参

public interface Func<T, R> {

R apply(T t);

default <V> Func<V, R> compose(Func<? super V, ? extends T> before) {

return v -> apply(before.apply(v));

}

}

### BiFunction

R apply(T t, U u)

两个输入，1个输出

### filter

过滤流数据

当返回值为true时，留下。false时，抛弃数据

**public** **static** List<Person> getPersonByAge(**int** age, List<Person> list) {

**return** list.stream().filter(p -> p.getAge() > age).collect(Collectors.*toList*());

}//首先把list变成流，过滤后，最后通过collect转换成list

### Predicate

And 取交集

Or 取并集

public <T> void handleList2(List<T> list, Predicate<T> predicate, Predicate<T> predicate2, Predicate<T> predicate3) {

for (T t : list) {

if (predicate.or(predicate3).and(predicate2).test(t)) {//这里从左往右匹配规则，先取并集，再取交集

System.*out*.println(t);

}

}

}

negate 取反

public <T> void handleList2(List<T> list, Predicate<T> predicate, Predicate<T> predicate2, Predicate<T> predicate3) {

for (T t : list) {

if (predicate.or(predicate3).negate().and(predicate2).test(t)) {//negate只对左边的结果部分取反，不影响右边

System.*out*.println(t);

}

}

}

isEqual 2个对象是否是相同

通过泛型直接比较的是equal方法

System.*out*.println(Predicate.*isEqual*("haha").test("haha"));

#### isEqual本质

public interface PredicateTest<T> {

boolean test(T t);

static <T> PredicateTest<T> isEqual(Object target) {

return new PredicateTest<T>() {// 1.创建独立的第三方对象

@Override

public boolean test(T t) {

if (target == null) {

return Objects.*isNull*(t);// 如果target为null，则直接看第三方接受的对象是否为null，为null则返回true。因为null == null

}

return target.equals(t);// 如果target不为null，则target和传入的对象比较，是否相同

}

};

}

}

#### 简化后

通过方法引用简化代码

public interface PredicateTest<T> {

boolean test(T t);

static <T> PredicateTest<T> isEqual(Object target) {

return (target == null)

? Objects::*isNull*

: target::equals;

}

}

### Supplier

没有参数，返回一个值

### Optional

处理空值

Optional<String> optional = Optional.*of*("abc");//赋值，不能为空

Optional<String> optional = Optional.*ofNullable*("abc");//赋值，有可能为空

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

optional.ifPresent(System.*out*::println);//确保有值，才发送数据处理

## 方法引用

当Lambda的body体中，只有那么一行代码，而且这一行代码，恰好完全匹配某个函数方法的格式，则可以用方法引用的方式，来完全代替Lambda的写法！

### 类名：：静态方法名

本质还是对象引用

定义静态方法，方法参数要求必须和lambda表达式参数完全匹配

### 对象名：：实例方法名

本质还是对象引用

定义实例方法，方法参数要求必须和lambda表达式参数完全匹配

### 类名：：实例方法名

定义实例方法，第一个方法参数，作为调用者使用，剩下的参数，才和方法参数匹配

**public** **class** Student {

**private** String name;

**public** Student(String name) {

**super**();

**this**.name = name;

}

**public** **int** compareByName(Student s) {

**return** **this**.getName().compareTo(s.getName());

}

}

Student s1 = **new** Student("a");

Student s2 = **new** Student("b");

Student s3 = **new** Student("c");

List<Student> lists = Arrays.*asList*(s2, s1, s3);

lists.sort(Student::compareByName); 2个参数都进行了调用！！

关键点在于，sort要求传入2个参数。但是当实例方法，少于2个参数的时候，则lambda表达式取第一个参数为调用者，第2个，甚至第3个。。。作为方法参数传入

lists.sort((student1, student2) -> {

**return** student1.compareByName(student2);

});

可以看见，其实要求传入2个参数，2个参数都进行了调用！！

### 类名：：new

构造方法引用

通过构造方法，来进行函数式匹配规则

比如，构造方法，满足条件

1. 无参传入，返回1个值
2. 有1个参数传入，返回1个值
3. 有多个参数传入，返回1个值

**public** **class** Demo01 {

**public** String getString(Supplier<String> s) {

**return** s.get() + "haha";

}

**public** String getString2(String str,Function<String, String> fun) {

**return** fun.apply(str);

}

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

Demo01 demo01 = **new** Demo01();

System.***out***.println(demo01.getString(String::**new**));//String类，恰好有无参传入，返回1个值的构造方法

System.***out***.println(demo01.getString2("hehe",String::**new**));//String类，恰好有传入1个参数，返回一个值的构造方法

}

}

//String类，恰好有无参传入，返回1个值的构造方法

**public** String() {

**this**.value = "".value;

}

//String类，恰好有传入1个参数，返回一个值的构造方法

**public** String(String original) {

**this**.value = original.value;

**this**.hash = original.hash;

}

## Default

**public** **interface** IMyDefInterface {

**default** **void** test() {//默认方法，必须通过实现类，对象来调用。已经有默认实现了，不需要单独实现。也可以用

System.***out***.println("MyDefInterface -- > default test()");

}

static int getId(){//静态方法，可以直接通过接口名来调用。

return 0;

}

}

**public** **class** Demo02 {

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

IMyDefInterface mInterface = **new** IMyDefInterface() {

@Override

**public** **void** test() {

// **TODO** Auto-generated method stub

MyDefInterface.**super**.test();//接口名.super.方法名

}

};

mInterface.test();

IMyDefInterface.getId();//接口中的静态方法，可以直接通过接口名调用

}

}

当Demo02，实现了2个接口。并且2个接口都有重名的default方法，则必须重写方法，指明怎么调用

default实现类的优先级要高于default接口实现！！

因为实现类重写了接口实现

## Stream

流操作由3部分操作

1. 源
2. 零个或多个中间操作
3. 终止操作

每一个中间操作返回的stream对象。都是一个新的对象，一个新的stream接口的实现

流操作分类

1. 惰性求值

相当于RxJava中的cold订阅。只有订阅者订阅后，才会开始发送数据！

1. 及早求值

相当于RxJava中的消费者，一个数据流只有1个消费者

### Stream本质

public class Demo {

public static void main(String[] args) {

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

List<Integer> collect = list.parallelStream().collect(

() -> {

List<Integer> supplierTemp = new ArrayList<>();

return supplierTemp;

},

(accumulatorTemp, item) -> {

accumulatorTemp.add(item);

},

(finalList, tempList) -> {

finalList.addAll(tempList);

});

}

}

<R> R collect(Supplier<R> supplier,

BiConsumer<R, ? super T> accumulator,

BiConsumer<R, R> combiner);

#### supplier

给最后的collect容器，提供collect对象

如果是串行流的话，只会提供一个对象

如果是并行流的话，会提供多个对象！

#### accumulator

拿到collect容器对象，来做具体添加item的动作！

如果是串行流的话，会拿着唯一collect对象，来做添加动作，然后直接返回该collect

如果是并行流，会有多个collect对象，来做添加动作

#### combiner

如果是串行流的话，该方法不执行。因为只有一个collect，直接把唯一的collect返回即可

如果是并行流的话，该方法执行，需要合并多个collect成一个，最后返回给外面！

### 流陷阱

1. 切记一个Stream流，一旦被终止后，就自动关闭！！如果要复用Stream，只有创建多个Stream,去复用数据，不要复用Stream
2. Stream流一旦被中间操作符操作后，也是不能复用！！
3. Stream默认是串行的，所以发射数据是一个一个发送。发送了一个数据，该数据直到短路后，这时，才再发送另一个数据。循环遍历只有一次！！

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

list.stream().map(s -> {

String result = s.substring(0, 1).toUpperCase() + s.substring(1);

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

return result;

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

结果居然是：

test

Hello

test

World

test

Yang

可以发现，第一个数据必须完整的走完整个通道后，再发送第二个数据！而且保证了数据的先后顺序

1. 小心无限流

注意操作符顺序，避免无限陷阱

IntStream.*iterate*(0, i -> (i + 1) % 2)

.distinct()//先去重复，结果只有0和1

.limit(6)//这样的话，永远都取不到6个，所以程序会无限运行

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

IntStream.*iterate*(0, i -> (i + 1) % 2)

.limit(6)//先取6个值，然后抛弃以后的值

.distinct()//再针对这6个值去重复

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

1. 并行流，不保证到达顺序，但是可以大大加快速度，使用parallel

### 创建方式

**public** **class** StreamClass {

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

Stream<String> streamOf = Stream.*of*("hello", "world");//可变参数

String[] strs = { "a", "b", "c" };

Stream<String> streamOfArray1 = Stream.*of*(strs);//可变参数

Stream<String> streamOfArray2 = Arrays.*stream*(strs);

List<String> lists = Arrays.*asList*("a", "b", "c");

Stream<String> streamOfList = lists.stream();//通过List直接得到

}

}

### 流转集合

String[] strs = {"hello", "world", "abc","hello"};

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

Stream<String> stream1 = Stream.*of*(strs);

Stream<String> stream2 = Stream.*of*(strs);//流被消费一次就自动关闭

List<String> list = stream.collect(Collectors.*toList*());//转arrayList

List<String> list = stream.collect(Collectors.*toCollection*(LinkedList::new));//转LinkedList

Set<String> list = stream1.collect(Collectors.*toCollection*(TreeSet::new));//排序去重

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

String collect = stream.collect(Collectors.*joining*()).toUpperCase();

System.*out*.println(collect);

### 流接口方法

#### peek

流中的数据，操作一下，然后再返回该数据，则用peek

流中的数据，操作一下后，返回另一个数据，则用map

Stream<T> peek(Consumer<? super T> action);

*Stream.of("one", "two", "three", "four")*

*\* .filter(e -> e.length() > 3)*

*\* .peek(e -> System.out.println("Filtered value: " + e))*

*\* .map(String::toUpperCase)*

*\* .peek(e -> System.out.println("Mapped value: " + e))*

*\* .collect(Collectors.toList());*

#### reduce

终止操作符！

和iterate很像。不过reduce只是操作流中的元素变换，不生成新的元素。

而iterate是直接无限生成新的元素！！

##### T reduce(T identity, BinaryOperator<T> accumulator);

从设定的值，开始执行变换！

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

// Optional<Integer> reduce = list.stream().reduce(Integer::sum);

int reduce = list.stream().reduce(5,Integer::*sum*);

System.*out*.println(reduce);

累计变换，从设置的5开始

结果5+1+2+3=11.

##### Optional<T> reduce(BinaryOperator<T> accumulator);

从流元素里面，取第一个值，开始变换！

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

Optional<Integer> reduce = list.stream().reduce(Integer::*sum*);

reduce.ifPresent(System.*out*::println);

累计变换，从流中的第一个元素1开始

结果为1+2+3=6

##### <U> U reduce(U identity,

##### BiFunction<U, ? super T, U> accumulator,

##### BinaryOperator<U> combiner);

和第一个方法一样，不过多了合并器，当并发操作的时候，并且没有设置CONCURRENT，合并器才管用

#### generate

generate ：参数是supplier类型，提供一个对象

Stream<String> stringStream = Stream.*generate*(UUID.*randomUUID*()::toString);//生成uuid

stringStream.findFirst().ifPresent(System.*out*::println);//如果存在，则打印出来

#### iterate

iterate：返回的是一个有序的，串行的，无限的流

Stream.*iterate*(1, integer -> integer \* integer + 1).limit(5).forEach(System.*out*::println);//取5个值，第一个值发送seed，第二个值发送应用函数后的值

结果是：

1

2

5

26

677

可以看见第一次直接发送seed值，后续值，通过无限循环应用Function得到

limit是一个会短路的，中间操作符

#### summaryStatistics

summaryStatistics：在intStream里面，可以拿到统计相关数据

IntSummaryStatistics intSummaryStatistics = Stream.*iterate*(1, integer -> integer + 2).limit(6)

.filter(integer -> integer > 2)

.mapToInt(integer -> integer \* 2)

.skip(2)

.limit(2)

.summaryStatistics();

System.*out*.println(intSummaryStatistics.getMax());//获取数据流最大值

System.*out*.println(intSummaryStatistics.getMin());//获取数据流最小值

System.*out*.println(intSummaryStatistics.getAverage());//获取数据流平均值

System.*out*.println(intSummaryStatistics.getCount());//获取数据流个数

System.*out*.println(intSummaryStatistics.getSum());//获取数据流求和

### 内部迭代

student.stream().filter(student -> student.getAge()>20 && student. getAddress().equals(“beijing”)).sorted(……).forEach(student -> System.out.println(student.getName()));

流关注的是对数据的计算；

### 外部迭代

集合关注的是数据与数据存储本身；

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

for(int i = 0 ; i< student.size(); i++){

Student s = student.get(i);

if(s.getAge()>20 && s.getAddress().equals(“beijing”)){

list.add(s);

}

}

Collections.sort(list,Compartor…..);

for(Student temp : list){

System.out.println(temp.getName());

}

### 流的并行和串行

#### 短路运算

只要找到合适的，后面的数据就不执行了

List<String> list = Arrays.*asList*("hello123", "world", "hello world");

list.stream().mapToInt(i -> {

int length = i.length();

System.*out*.println(i);//重点观察这里的打印结果

return length;

})

.filter(i -> i == 5)

.findFirst().ifPresent(System.*out*::println);

对于串行流来说。findFirst的短路操作，这里会打印 hello123 和 world。

当发现world符合条件后，后面就短路了。

对于并行流来说。过程是未知的，不一定所有的单词都会被打印出来。

### 流分组与分区

#### flatmap

当需要交叉2个流发送数据的时候，使用flatmap非常合适！

List<String> list = Arrays.*asList*("Hi", "Hello", "你好");

List<String> names = Arrays.*asList*("张三", "王五");

list.stream().flatMap(s -> names.stream().map(s2 -> s + s2))

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

map：输出单个值

flatmap：输出单个Stream

#### group by分组

巨方便的分组操作！！

##### Collectors.groupingBy(Fun……)

Fun函数是指的根据什么来分组，这个Fun返回的值就是map的key

Person yang = new Person("yang", 100, 22);

Person sun = new Person("sun", 66, 30);

Person wang = new Person("wang", 990, 40);

Person yang1 = new Person("yang", 80, 22);

List<Person> list = Arrays.*asList*(yang, sun, wang, yang1);

Map<String, List<Person>> collect = list.stream().collect(Collectors.*groupingBy*(Person::getName));//根据名字分组！

collect.forEach((s, people) -> System.*out*.println(s + " == " + people));

结果：

wang == [Person{name='wang', score=990, age=40}]

yang == [Person{name='yang', score=100, age=22}, Person{name='yang', score=80, age=22}]

sun == [Person{name='sun', score=66, age=30}]

##### Collectors.groupingBy(Fun……，Collectors……)

第二个参数：Collectors的返回值，决定了最终map的Values

Map<String, Long> collect = list.stream()

.collect(Collectors.*groupingBy*(Person::getName,Collectors.*counting*()));

wang ==1

yang == 2

sun == 1

Values为数据流的double类型平均值

Map<String, Double> collect = list.stream().collect(Collectors.*groupingBy*(Person::getName, Collectors.*averagingDouble*(Person::getScore)));

wang == 99.0

yang == 90.0

sun == 66.0

##### Collectors.groupingBy(Fun……，Supplier map…,Collectors……)

Fun的转换结果，决定了key的值

Supplier的结果，决定了收集器的map类型，默认是hashmap，如需要排序使用treeMap

Collectors的转换结果，决定了value的值，这里使用了mapping，values结果从List<Person>，转换成了Set<String>

Map<City, Set<String>> namesByCity = people.stream().collect(

*groupingBy*(Person::getCity, TreeMap::new, *mapping*(Person::getLastName, *toSet*())));

##### Collectors.groupingByConcurrent(Fun……，Supplier map…,Collectors……)

groupingByConcurrent也有3个重载方法，和groupingBy一模一样

区别是：ByConcurrent对并行流有更好的性能支持！！但是，对顺序不做要求的话！！！

源码显示该集合特征为：Characteristics.CONCURRENT和Characteristics.UNORDERED

底层是concurrentHashMap

#### partition by分区

分区是分组的特殊情况

分区的结果只有两种情况：key只能是true和false

把数据流满足某个特定条件的分为2组，符合的一组，不符合的一组

Map<Boolean, List<Person>> collect = list.stream().collect(Collectors.*partitioningBy*(person -> person.getScore() > 90));

结果：

false == [Person{name='sun', score=66, age=30}, Person{name='yang', score=80, age=22}]

true == [Person{name='yang', score=100, age=22}, Person{name='wang', score=99, age=40}]

和group by的重载方法一样，第一个参数是map的key，第二个参数是map的value

#### 多级分组

Person yang = new Person("yang", 100, 22);

Person sun = new Person("sun", 66, 30);

Person wang = new Person("wang", 99, 40);

Person yang1 = new Person("yang", 80, 22);

Person yang2 = new Person("yang", 20, 88);

按照名字分组，并且只保留组里面分数最低的

Map<String, Person> collect = list.stream().collect(*groupingBy*(Person::getName,

*collectingAndThen*(*minBy*(Comparator.*comparingInt*(Person::getScore)), Optional::get)));

System.*out*.println(collect);

结果：

{ wang=Person{name='wang', score=99, age=40},

yang=Person{name='yang', score=20, age=88},

sun=Person{name='sun', score=66, age=30}

}

### 比较器

Comparator

#### thenComparing(Coomparator ……)

可以多次比较串联比较

比如先按照成绩降序，再根据名字首字母降序

特别注意：只有当第一次比较结果相同，相同的部分，才拿给第二次结果比较！！，如果第一次比较结果全部都不一样，则第二次结果直接不比较！

List<String> list = Arrays.*asList*("nihao","hello","world","welcome");

Comparator<String> stringComparator = Comparator.*comparingInt*(String::length).thenComparing(String::compareTo);

//先根据字符串长度比较，再根据字母升序排序

list.stream().sorted(stringComparator).forEach(System.*out*::println);

结果：

hello

nihao

world

welcome

#### thenComparing (Fun……,Coomparator ……)

比较的2个值，先进行Fun转换，再比较转换后的值

list.sort(Comparator.*comparing*(String::toLowerCase, Comparator.*comparing*(String::length)));

Collections.*sort*(list,Comparator.*comparingInt*(String::length).reversed()

.thenComparing(String::toLowerCase,Comparator.*reverseOrder*()));

### collect

#### 总结

1. collect是一个收集器
2. Collector作为collect方法的参数
3. Collector是一个接口，是一个可变的汇聚操作，将输入元素累积到一个可变的结果容器中；它会在所有元素都处理完毕后，将累积的结果转换为一个最终的表示（这是一个可选操作）；它支持串行与并行
4. Collectors本身提供了关于Collector的常见汇聚实现，实际上是一个工厂
5. 为了确保串行与并行操作结果的等价性，Collector函数需要满足两个条件，一个是identity同一性，一个associativity是结合性。
6. a == combiner.apply(a,supplier.get())
7. 函数式编程最大的特点：表示做什么，而不是如何做

#### enum Characteristics

Characteristics给Collector，赋予独特的特征！！

##### CONCURRENT

多个结果容器，并行同时操作一个结果容器！！accumulator收集器

list.stream()。如果不加CONCURRENT，表示的是，list所在的一个线程，操作一个accumulator，所以accumulator内容会越来越多。supplier只会调用一次。

list.parallelStream()。如果不加CONCURRENT，表示的是，多个子线程操作的是多个结果容器。然后多个结果容器，通过combiner被调用，合并。另supplier会调用多次。

list.parallelStream()。如果加CONCURRENT，表示的是，多个子线程操作的是同一个结果容器。所以combiner不被调用，不需要通过其合并，直接返回结果容器即可。另supplier会调用多次。

切记：如果当前是CONCURRENT+parallelStream的时候，切记不要在accumulator方法里面又做add，又做遍历！会出错！。正确的做法是：只是单纯add即可

如果当前是parallelStream，去掉CONCURRENT的时候，多个线程，在accumulator方法里面就有多个set，所以又做add，又做遍历！互不影响！

##### UNORDERED

两种情况，一种是不考虑顺序，那么a,b,c 和 b,c,a是一个相同的集合

一种是考虑顺序，那么a,b,c 和 b,c,a 是两个不同的集合

如果数据源定义为无序的，则设置UNORDERED，例如set

##### IDENTITY\_FINISH

设置了后，finisher函数无效，因为A和R类型一样，提高效率。

不设置，finisher函数才有效！！

#### 自定义collector

Collector泛型接口<T,A,R>

T：输入流的数据类型

A：中间合并搜集的集合类型

R：最终返回的集合类型

combiner：

reduce 是针对不可变容器，并行流

collect 是针对可变容器

import static java.util.stream.Collector.Characteristics.\*;

public class MySetCollector<T> implements Collector<T, Set<T>, Set<T>> {

//提供一个收集器

@Override

public Supplier<Set<T>> supplier() {

System.*out*.println("MySetCollector Supplier");

return HashSet::new;

}

//把流数据添加给集合

@Override

public BiConsumer<Set<T>, T> accumulator() {

System.*out*.println("MySetCollector accumulator");

return Set::add;

}

//当多并发时，把多个集合合并到一个最终集合

@Override

public BinaryOperator<Set<T>> combiner() {

System.*out*.println("MySetCollector combiner");

return (set1, set2) -> {

set1.addAll(set2);

return set1;

};

}

//当结果不一致的时候，调用，统一最终集合类型

@Override

public Function<Set<T>, Set<T>> finisher() {

System.*out*.println("MySetCollector finisher");

// return set -> (Set<T>) set;

return Function.*identity*();

}

//设置合并特征规则

@Override

public Set<Characteristics> characteristics() {

System.*out*.println("MySetCollector characteristics");

return Set.*of*(*IDENTITY\_FINISH*,*UNORDERED*);

}

}

#### 混合转换

数据流通过Set搜集，最后转换成map输出

只有*IDENTITY\_FINISH*不设置的时候，finisher方法内部才有效！！

import static java.util.stream.Collector.Characteristics.\*;

public class MySetCollector<T> implements Collector<T, Set<T>, Map<T, T>> {

//提供一个收集器

@Override

public Supplier<Set<T>> supplier() {

System.*out*.println("MySetCollector Supplier");

return HashSet::new;

}

//把流数据添加给集合

@Override

public BiConsumer<Set<T>, T> accumulator() {

System.*out*.println("MySetCollector accumulator");

// return (map,str) -> map.add(str,str);

return Set::add;

}

//当多并发时，把多个集合合并到一个最终集合

@Override

public BinaryOperator<Set<T>> combiner() {

System.*out*.println("MySetCollector combiner");

return (set1, set2) -> {

set1.addAll(set2);

return set1;

};

}

//最终返回的类型，接口当中的R决定其参数类型，不能设置*IDENTITY\_FINISH*

@Override

public Function<Set<T>, Map<T, T>> finisher() {

System.*out*.println("MySetCollector finisher");

// return set -> (Set<T>) set;

// return Function.identity();

return set -> set.stream().collect(Collectors.*toMap*(s -> s, s -> s));

}

//设置合并特征规则

@Override

public Set<Characteristics> characteristics() {

System.*out*.println("MySetCollector characteristics");

return Set.*of*(*UNORDERED*);

}

}

### Collectors

静态工厂类，实现一共分为两种：

1. 通过CollectorImpl来实现
2. 通过reducing来实现。reducing方法本来又是通过CollectorImpl来实现

#### mapping

把数据对象，分组后，更加扁平化，不聚合对象，直接聚合具体字段！

Person p1 = new Person("CQ", "yang");

Person p2 = new Person("CQ", "sun");

Person p3 = new Person("BJ", "wang");

Person p4 = new Person("BJ", "zhao");

List<Person> personList = Arrays.*asList*(p1, p2, p3, p4);

Map<String, List<Person>> collect = personList.stream().collect(Collectors.*groupingBy*(Person::getCity, Collectors.*toList*()));

System.*out*.println(collect);

结果是聚合对象！

{BJ=[Person{city='BJ', lastName='wang'}, Person{city='BJ', lastName='zhao'}], CQ=[Person{city='CQ', lastName='yang'}, Person{city='CQ', lastName='sun'}]}

Person p1 = new Person("CQ", "yang");

Person p2 = new Person("CQ", "sun");

Person p3 = new Person("BJ", "wang");

Person p4 = new Person("BJ", "zhao");

List<Person> personList = Arrays.*asList*(p1, p2, p3, p4);

Map<String, Set<String>> collect = personList.stream().collect(Collectors.*groupingBy*(Person::getCity, Collectors.*mapping*(Person::getLastName, Collectors.*toSet*())));//把对象Person，再做一次map转换，转换成String字段

// Map<String, List<Person>> collect = personList.stream().collect(Collectors.groupingBy(Person::getCity, Collectors.toList()));

System.*out*.println(collect);

结果是聚合字段：

{BJ=[wang, zhao], CQ=[yang, sun]}

#### collectingAndThen

collectingAndThen(Collector<T,A,R> downstream , Function<R,RR> finisher)

*List<String> list = people.stream().collect(*

*\* collectingAndThen(toList(),*

*\* Collections::unmodifiableList));*

源码显示是：把collections：：unmodifiableList这个函数当成finisher，做最后变换！

## AutoClose

### try-with-resource的用法

小括号：声明。

大括号：使用方法

小括号的类，必须实现AutoCloseable接口。不然无法编译

public class AutoCloseableTest implements AutoCloseable {

public void doSomething() {

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

}

@Override

public void close() throws Exception {

System.*out*.println("close invoked");

}

public static void main(String[] args) {

try (AutoCloseableTest autoCloseableTest = new AutoCloseableTest()) {

autoCloseableTest.doSomething();

} catch (Exception e) {

e.printStackTrace();

}

}

}

结果：

doSomething

close invoked

### onClose

通过try-with-resource的用法

要触发onClose方法，必须使用try-with-resource

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6);

try (Stream<Integer> integerStream = list.stream().onClose(() -> System.*out*.println("close")).onClose(() -> System.*out*.println("close again"))) {

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

结果：

1

2

3

4

5

6

close

close again

#### 细节

1. 前面一个onClose抛出了异常，但是后一个onClose依然可以执行
2. 如果catch了异常，则走catch代码块
3. 多个onClose抛多个异常，只有第一个异常可以完整显示，后面的异常会显示不完整被Suppressed住
4. 如果后面的异常对象和第一个异常对象一样。则都不会被压制，毕竟自己对象不能被自己对象压制
5. 如果后面的异常，虽然名字一样，但是对象不一样。则会被压制

try (Stream<Integer> integerStream = list.stream().onClose(() -> {throw new IllegalArgumentException("abc");}).onClose(() -> System.*out*.println("close again"))) {

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

}

结果1：

1

2

3

4

5

6

close again

Exception in thread "main" java.lang.IllegalArgumentException: abc

at com.mantoupackage.jdk8.stream.StreamTest15.lambda$main$0(StreamTest15.java:14)

at java.base/java.util.stream.Streams$1.run(Streams.java:842)

at java.base/java.util.stream.AbstractPipeline.close(AbstractPipeline.java:323)

at com.mantoupackage.jdk8.stream.StreamTest15.main(StreamTest15.java:16)

try (Stream<Integer> integerStream = list.stream().onClose(() -> {throw new IllegalArgumentException("abc");}).onClose(() -> System.*out*.println("close again"))) {

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

}catch (IllegalArgumentException e){

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

}

结果2：

1

2

3

4

5

6

close again

catch

try (Stream<Integer> integerStream = list.stream().onClose(() -> {throw new IllegalArgumentException("abc");}).onClose(() -> {throw new RuntimeException("run");})) {

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

}

结果3：

1

2

3

4

5

6

Exception in thread "main" java.lang.IllegalArgumentException: abc

at com.mantoupackage.jdk8.stream.StreamTest15.lambda$main$0(StreamTest15.java:14)

at java.base/java.util.stream.Streams$1.run(Streams.java:842)

at java.base/java.util.stream.AbstractPipeline.close(AbstractPipeline.java:323)

at com.mantoupackage.jdk8.stream.StreamTest15.main(StreamTest15.java:16)

//第一个异常被完整展示

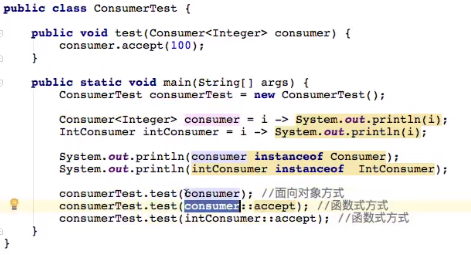
Suppressed: java.lang.RuntimeException: run

at com.mantoupackage.jdk8.stream.StreamTest15.lambda$main$1(StreamTest15.java:14)

at java.base/java.util.stream.Streams$1.run(Streams.java:846)

... 2 more //这里显示不完整，被抑制suppressed了

## Spliterator



面向对象的方式传递，consumer和intConsumer必须是子父类，发生关系，才能传递

而函数式方式传递数据，传递的是行为，通过行为推导，推测出，intConsumer也符合函数式编程风格规范！所以可以传递

## Lambda

public class LambdaTest01 {

Runnable r1 = () -> System.*out*.println(this);//打印出来是对象

Runnable r2 = new Runnable() {//打印出来是匿名内部类！生成了class文件

@Override

public void run() {

System.*out*.println(this);

}

};

public static void main(String[] args) {

LambdaTest01 lambdaTest01 = new LambdaTest01();

Thread t1 = new Thread(lambdaTest01.r1);

Thread t2 = new Thread(lambdaTest01.r2);

t1.start();

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

t2.start();

}

}

结果：

~~~~~~~~~~~~~~~~~~~~~~

com.mantoupackage.jdk8.lambda.LambdaTest01@19011d69

[com.mantoupackage.jdk8.lambda.LambdaTest01$1@6a5ad9a2](mailto:com.mantoupackage.jdk8.lambda.LambdaTest01$1@6a5ad9a2)

## 时间与日期

JDK8中所有的时间与日期都是不可变类。确保线程安全！

原date，日历类。都是可变类，是线程不安全的

### jodaTime

jodaTime是不可变的对象

implementation("joda-time:joda-time:2.10.1")

public class JodaTest02 {

public static void main(String[] args) {

DateTime today = new DateTime();//今天时间

DateTime tomorrow = today.plusDays(1);//明天时间

System.*out*.println(today.toString("yyyy--MM--dd"));

System.*out*.println(tomorrow.toString("yyyy--MM--dd"));

DateTime d1 = today.withDayOfMonth(1);//保留当前年和月，日期改为1日

System.*out*.println(d1.toString("yyyy--MM--dd"));

LocalDate localDate = new LocalDate();//当前时区日期

System.*out*.println(localDate);

//输出上个月的最后一天

localDate = localDate.plusMonths(-1).dayOfMonth().withMaximumValue();

System.*out*.println(localDate);

//计算2年前，第3个月，最后1天的日期

LocalDate d2 = new LocalDate().minusYears(2)

.monthOfYear().setCopy(3)

.dayOfMonth().withMaximumValue();

System.*out*.println(d2);

}

}

#### UTC时间

public class JodaTest03 {

//服务端把客服端传的字符串时间，转换为date对象

//传入标准UTC时间字符串："2019-03-06T08:49:41.881Z"

//返回toString后的结果："Wed Mar 06 16:49:41 CST 2019"

public static Date convertUTC2Date(String utcDate){

try {

DateTime dateTime = DateTime.*parse*(utcDate, DateTimeFormat.*forPattern*("yyyy-MM-dd'T'HH:mm:ss.SSSZ"));

return dateTime.toDate();

}catch (Exception e){

return null;

}

}

//客户端把当前时间，转换为统一的UTC时间

//传入new Date()，也就是当前时间

//返回字符串："2019-03-06T08:49:41.881Z"

public static String convertDate2String(Date date){

DateTime dateTime = new DateTime(date, DateTimeZone.*UTC*);

return dateTime.toString();

}

//当前的时间，按照本地格式解析时间对象，转换成字符串

//传入new Date()，也就是当前时间，根据格式"yyyy--MM--dd HH-mm-ss"

//返回字符串："2019--03--06 17-33-01"

public static String convertDate2LocalByDateFormat(Date date,String format){

DateTime dateTime = new DateTime(date);

return dateTime.toString(format);

}

public static void main(String[] args) {

System.*out*.println(*convertDate2LocalByDateFormat*(new Date(),"yyyy--MM--dd HH-mm-ss"));

}

}

### TimeFromJDK8

可读性更好，月份返回1-12。原date返回0-11

LocalDateTime 不带时区显示的，本地区完整年月日时分秒 ： 2019-03-07T11:55:17.199

ZonedDateTime 带时区显示的，本地完整年月日时分秒：

2019-03-07T11:55:17.199+08:00[Asia/Chongqing]

YearMonth 年月：2019-03

MonthDay 月日： 03-06

LocalDate 年月日 ：2019-03-06

LocalTime 时分秒： 10:59:33.404

Clock 以毫秒为计数：1551928235297

Instant UTC时间：2019-03-07T05:47:33.274Z

Period 间隔：年份，月份，天份全部单独分开比较的间隔

ChronoUnit.DAYS.between(date1,date2) 间隔：2个日期的间隔天数

public class Jdk8Time {

public static void main(String[] args) {

//LocalDate关注的是年月日

LocalDate date = LocalDate.*now*();

System.*out*.println(date);//打印当前时间结果："2019-03-06"

LocalDate date1 = LocalDate.*of*(2017,1,31);

System.*out*.println(date1);//打印自定义的时间结果："2017-01-31"

LocalDate localDate = LocalDate.*of*(2010,3,25);

//MonthDay 只存储月份和天数。用于比较忽略年份的情况，非常方便！

//MonthDay 只关注月份和日！

MonthDay monthDay1 = MonthDay.*of*(localDate.getMonth(),localDate.getDayOfMonth());

MonthDay monthDay2 = MonthDay.*from*(LocalDate.*of*(2011,3,25));

if (monthDay1.equals(monthDay2)){

System.*out*.println("equals");//结果："equals"

}else {

System.*out*.println("not equals");

}

//LocalTime关注时分秒！

LocalTime time = LocalTime.*now*();

System.*out*.println(time);//结果:"10:59:33.404"

//plus往后计算，重载方法，有各种单位，天，月，年，周，时，分，秒

LocalDate localDate1 = LocalDate.*now*().plus(2, ChronoUnit.*WEEKS*);

System.*out*.println(localDate1);

//minus往前计算，重载方法，有各种单位，天，月，年，周，时，分，秒

LocalDate localDate2 = LocalDate.*now*().minus(2, ChronoUnit.*DAYS*);

System.*out*.println(localDate2);

//表示时间线上的某个时间点，或者叫某个时刻

Clock clock = Clock.*systemDefaultZone*();

System.*out*.println(clock.millis());//打印结果："1551928235297"

LocalDate localDate3 = LocalDate.*now*();

LocalDate localDate4 = LocalDate.*of*(2019,3,7);

System.*out*.println(localDate3.isBefore(localDate4));//是否在指定日期之前，如果相同为false

System.*out*.println(localDate3.isAfter(localDate4));//是否在指定日期之后，如果相同为false

System.*out*.println(localDate3.isEqual(localDate4));//是否和指定日期相同，必须相同才为true

Set<String> zoneIds = ZoneId.*getAvailableZoneIds*();//时区

zoneIds.stream().sorted().forEach(System.*out*::println);//世界上所有的时区全部打印出来

//LocalDateTime关注的是，完整的本地,本时区时间！

LocalDateTime localDate5 = LocalDateTime.*now*();

ZoneId zoneId = ZoneId.*of*("Asia/Chongqing");

System.*out*.println(localDate5);//打印结果："2019-03-07T11:55:17.199"

//ZonedDateTime关注的是带时区的完整时间！

ZonedDateTime zonedDateTime = ZonedDateTime.*of*(localDate5,zoneId);

System.*out*.println(zonedDateTime);//打印结果："2019-03-07T11:55:17.199+08:00[Asia/Chongqing]"

//YearMonth关注的是年份和月份

YearMonth yearMonth = YearMonth.*now*();

System.*out*.println(yearMonth);//打印结果："2019-03"

System.*out*.println(yearMonth.isLeapYear());//是否是闰年

System.*out*.println(yearMonth.lengthOfMonth());//当前月份有多少天

System.*out*.println(yearMonth.lengthOfYear());//当前年份有多少天

LocalDate localDate6 = LocalDate.*now*();//2019-3-7

LocalDate localDate7 = LocalDate.*of*(2019,4,7);

//period表示纯粹分离年，月，日的间隔。

Period period = Period.*between*(localDate6,localDate7);//前面日期比后面日期旧，得到结果是正数，反之是负数

System.*out*.println(period.getMonths());//纯粹只比较月份。3-7和3-8相差0月份，结果为：5

System.*out*.println(period.getDays());//纯粹只比较日期，3-7和3-8相差1天，结果为：1

System.*out*.println(period.getYears());//纯粹只比较年份。结果为：1

//计算两个日期之间相差的天数，也是含头不含尾

//含头不含尾的意思是：比如3-7日和3-10日。差3天。以7日开始算，还有3天就是10号。

//如果从7号开始算数字1。那么数字3，就是9日，所以看上去就是含7不含10

long between = ChronoUnit.*DAYS*.between(localDate6, localDate7);

System.*out*.println("两个日期相差的天数是：" + between);

//Instant表示当前的时刻，UTC时间

System.*out*.println(Instant.*now*()); //结果是："2019-03-07T05:47:33.274Z"

}

}