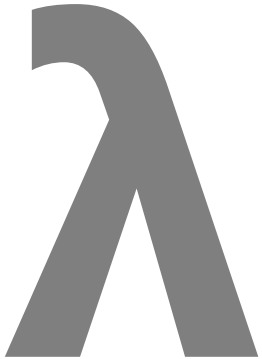
Java 8 函数式编程

Functional Programming

于文琦

JD.京东.COM

Lambda



- 希腊字母表中第十一位
- 物理上的波长符号
- 线性代数中的特征值

About Functional Programming

什么是函数式编程？

是一种编程范式，是如何编程的方法论（Methodology）。

什么是方法论？

方法论是哲学术语，简单说**就是用什么样的方式来处理问题。**

百度百科：方法论是普遍适用于各门具体社会科学并起到指导作用的范畴、原则、理论、方法和手段的总和。

函数式编程的知识结构（代码即数据的编程风格）

Stream②

lambda表达式①

Collectors收集器④

自定义类库③

数据并行化⑤

Lambda表达式

```
Runnable noArguments=()-> System.out.println("Hello World.");
noArguments.run();
```

```
Runnable multiStatement=()-> {
    System.out.println("Hello World.");
    try {
        Integer.parseInt("AAA");
    }catch(NumberFormatException e){}
    Scanner scanner=new Scanner(System.in);
    System.out.println(scanner.next());
};
multiStatement.run();
```

```
Comparable comparable=object-> 1;//编译器可根据上下文推断出object的类型
```

```
Comparator comparator=(Object x,Object y)->{
    if(x.hashCode()>y.hashCode()){//伪代码
        return 1;
    }
    return 0;
};
```

值传递和函数接口

```
String name="AAA";
Runnable value=()-> System.out.println("Hello World."+name);
value.run();
```

name就是一个既成事实的final（effectively final）变量（java8）
匿名内部类要引用外部变量，外部变量必须是final的
java8放开了这个限制，可以不用final修饰，但不能多次赋值。

*函数接口：是只有一个抽象方法的接口

java.util.function
Predicate(test)、Consumer(accept)、Function(apply)、Supplier(get)、
UnaryOperator 继承Function
BinaryOperator 继承BiFunction

```
@FunctionalInterface
public interface Runnable {
```

*未标记FunctionalInterface的接口也可以用于lambda表达式

JDK8之前已有的函数式接口

java.lang.Runnable
java.util.concurrent.Callable
java.util.Comparator
java.lang.reflect.InvocationHandler
java.awt.event.ActionListener
…

有的接口并没有标注@FunctionalInterface，但依然不改变它是函数接口的本质。

FI注解会检查是否只有一个待实现的方法（静态方法，默认方法除外）-java8特性

Stream

Stream：是用函数式编程方式在集合类上进行复杂操作的工具。

```
List<String> list= new ArrayList();
list.add(“A”);
list.add(“B”);
cnt=list.stream().filter(str->str.equals(“A”)).count(); //Predicate
for(String s:list){
    if(s.equals("A")){
      cnt++;
    }
}
```

惰性求值方法（filter）vs及早求值方法（count）

```
List<String>
bList=Stream.of("A","B").map(String::toLowerCase).collect(Collectors.toList());//Function
System.out.println(bList.equals(Arrays.asList("a","b")));
```

区别：Functional Interface vs @FunctionalInterface vs Function

Stream

Stream：是用函数式编程方式在集合类上进行复杂操作的工具。

```
Arrays.asList(3,1,2).stream().min(Comparator.comparing(i->i)).get();//Optional

Arrays.asList(obj1,obj2).stream().max(Comparator.comparing(obj->obj.hashCode())).get();
```

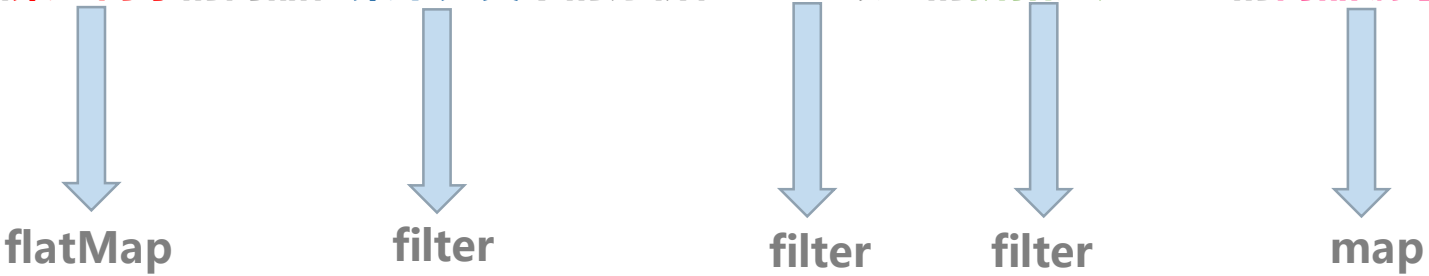
```
BinaryOperator<Integer> add=(x,y)->x+y;
int count=Stream.of(1,2,3).reduce(add).get();
```

```
BinaryOperator<Integer> add=(x,y)->x+y;
int count=Stream.of(1,2,3).reduce(0,add);
```

Map vs Reduce ?
Map:将一组值转换为另一组值
Reduce：从一组值中生成一个值

For example

假设要找出某些商家的商品在某个分类下的原价在1500以上的折扣超过75%的商品编号




```
List<Integer> wareIdList=venderList.stream()
        .flatMap(vender -> vender.getWareList())
        .filter(ware->ware.getCategory()==1)
        .filter(ware -> ware.getPrice() > 1500)
        .filter(ware -> ware.getDiscount() > 75)
        .map(Ware::getWareId)//方法引用 method reference
        .collect(toList());
```

自定义类库

自定义可以使用lambda表达式的方法

```
public void debug(Supplier<String> message){
    debug(message.get());
}

public void debug(String message){
    System.out.println(message);
}
```

JAVA8 new feature： Collection stream，Iterable forEach

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

接口的默认方法的继承、静态方法

类胜于接口
子类胜于父类

```
接口继承，Walk extends Move
接口实现类，WalkImpl
接口实现类的继承 WalkImpl extends MoveImpl
多重实现类 RunImpl implements A ,B
```

Java 8 new feature：**Stream.of, Optional.of**

```
Optional<Integer> optional=Optional.empty();
if(optional.isPresent()){
    System.out.println(optional.get());
}
System.out.println(optional.orElse(1));
System.out.println(optional.orElseGet(()->2));
```

用Optional类型代替null，鼓励程序员检查变量是否为空。

Collectors

import static java.util.stream.Collectors.toList;//静态引入

```
List<Integer> wareIdList=venderList.stream()
        .flatMap(vender -> vender.getWareList())
        .filter(ware->ware.getCategory()==1)
        .filter(ware -> ware.getPrice() > 1500)
        .filter(ware -> ware.getDiscount() > 75)
        .map(Ware::getWareId)//方法引用 method reference
        .collect(toList());

toCollection, toSet, toMap, toConcurrentMap, maxBy, averagingInt,
partitionBy, groupingBy, joining,
```

stream has already been operated upon or closed

partitionBy vs groupingBy
组合收集groupingBy($lambda,mapping)

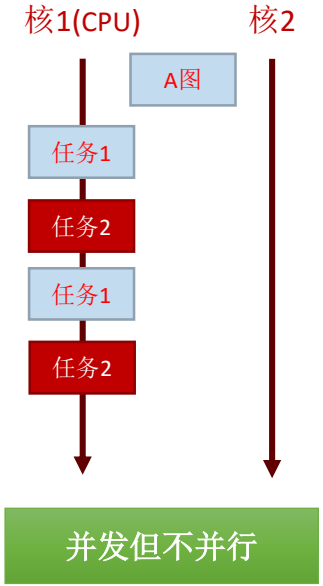
自定义Collector

```
public class StringCollector implements Collector<String,StringCombiner,String> {
    private String append;
    private static final Set<Characteristics> characteristics = Collections.emptySet();
    public StringCollector(String append){
        this.append=append;
    }
    @Override
    public Supplier supplier() {
        return ()->new StringCombiner(append);
    }
    @Override
    public BiConsumer<StringCombiner,String> accumulator() {
        return StringCombiner::accumulate;
    }
    @Override
    public BinaryOperator<StringCombiner> combiner() {
        return StringCombiner::merge;
    }
    @Override
    public Function<StringCombiner,String> finisher() {
        return StringCombiner::toString;
    }
    @Override
    public Set<Characteristics> characteristics() {
        return characteristics;
    }
}
```

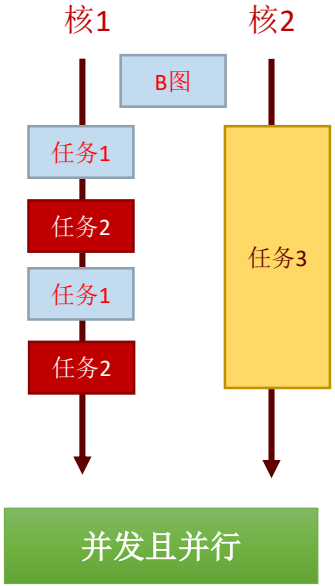
```
skuList
.stream()
.map(SKU::getName)
.collect(
new StringCollector("^")
);
```

Since1.8
Map.computeIfAbsent
Get? Put?

并行与并发



并发但不并行



并发且并行

任务1与任务2是并发处理

任务1与任务3是并行处理

任务2与任务3是并行处理

并发：多个任务共享时间段。
并行：多个任务同一时间发生。

数据并行化：是指将数据分成快，为每块数据分配单独的处理单元。

并行化操作

Stream.of(1,2,3).parallel().collect(...);

List.parallelStream().collect(...);

性能好：ArrayList，数组，IntStream.range
性能一般：HashSet，TreeSet
性能差：LinkedList，Stream.iterate

无状态性能好：map，filter，flatMap
有状态性能差：limit，sorted，distinct

```
int size=1000000;
Seller[] sellers=new Seller[size];
Arrays.setAll(sellers,
        i -> new Seller(i,new BigDecimal(i+ Math.random()),1+i));
double ret=Arrays.asList(sellers)
        .stream().mapToDouble(
                seller -> seller.getPrice().doubleValue() *
seller.getRate()).sum();
```

Sequential vs Parallel
10000 条：0.475 vs 0.558
100000 条：1.132 vs 1.040
1000000条：9.770 vs 6.858

```
int size=1000000;
Seller[] sellers=new Seller[size];
Arrays.parallelSetAll(sellers,
        i->new Seller(i,new BigDecimal(i+ Math.random()),1+i));
double ret=Arrays.asList(sellers)
        .parallelStream().mapToDouble(
                seller -> seller.getPrice().doubleValue() *
seller.getRate()).sum();
```

THANK YOU