Java8 全新Stream 机制详解

课程：

1. stream概要与产生背景
2. lambada 表达示详解
3. Stream执行机制解密

一、stream概要与使用场景

---

stream 概要

首先要澄清的是 java8 中的stream 与InputStream和OutputStream是完全不同的概念，stream 是用于对集合迭代器的增强，使之完成 能够完成更高效的聚合操作（**过滤**、**排序**、**统计分组**）或者大批量数据操作。此外与stream 与lambda 表达示结合后编码效率与大大提高，并且可读性更强。

☐ 示例展示：

```
// 获取所有红色苹果的总重量
appleStore.stream().filter(a -> "red".equals(a.getColor()))
.mapToInt(w -> w.getWeight()).sum()
```

```
// 基于颜色统计平均重量
appleStore.stream().collect(Collectors.groupingBy(a -> a.getColor(),
        Collectors.averagingInt(a -> a.getWeight()))).forEach((k, v) -> {
    System.out.println(k + ":" + v);
});
```

stream 产生背景

‘获取所有红色苹果的总重量’，如果用SQL其实非常好实现，为什么不在直接关系数据库来实现呢?

```
//获取所有红色苹果的总重量
select sum(a.weight) from apple as a  where a.color='red';
// 基于颜色分组统计重量
select a.color,sum(a.weight) from apple as a group by color;
```

遍历在传统的javaEE 项目中数据源比较单而且集中，像这类的需求都我们可能通过关系数据库中进行获取计算。但现在的互联网项目数据源成多样化有：关系数据库、NoSQL、Redis、mongodb、ElasticSearch、Cloud Server 等。这时就需我们从各数据源中汇聚数据并进行统计。这在Stream出现之前只能过遍历实现 非常繁琐。

场景一：跨库join的问题
查询一个店铺的订单信息，需要用到订单表与会员表 在传统数据库单一例中 可以通过jon 关联轻松实现，但在分布场景中 这两张表分别存储在于 交易库 和会员库 两个实例中，join不能用。只能在服务端实现其流程如下：

1. 查询订单表数据
2. 找出订单中所有会员的ID
3. 根据会员ID查询会员表信息
4. 将订单数据与会员数据进行合并

这用传统迭代方法非常繁琐，而这正是stream 所擅长的。示例代码如下：

```
// 获取所有会员ID 并去重
List<Integer> ids = orders.stream().map(o ->
o.getMemberId()).distinct().collect(Collectors.toList());
//  合并会员信息 至订单信息
orders.stream().forEach(o -> {
    Member member = members.stream().filter(m -> m.getId() ==
o.getMemberId()).findAny().get();
    o.setMemberName(member.getName());
});

```

场景二：N+1 问题

二、lambda 表达示详解

Lambada 简介：

Lambda 表达式，也可称为闭包，它是推动 Java 8 发布的最重要新特性。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）。使用 Lambda 表达式可以使代码变的更加简洁紧凑

示例：
匿名类写法

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("hello  鲁班");
    }
}).start();
```

Lambada写法

```
new Thread(() -> System.out.println("hello  鲁班")).start();
```

在上述例子中编译器会将“System.out.println(”hello 鲁班”)”编译成Runnable.run 的执行指令。可代码中我们并没有指明Run方法，这是因为 run 方法是Runnable接口的唯一方法，也就是说如果Runable有多个方法是不能使用Lambada表达示的，这种支持Lambada的接口统称函数式接口。

函数式接口：

必须是 函数式接口 才可以使用lambada 表达示 ， 函数式接口笼统的讲就是只有一个抽像方法接口就是函数式接口，其详细特征如下：

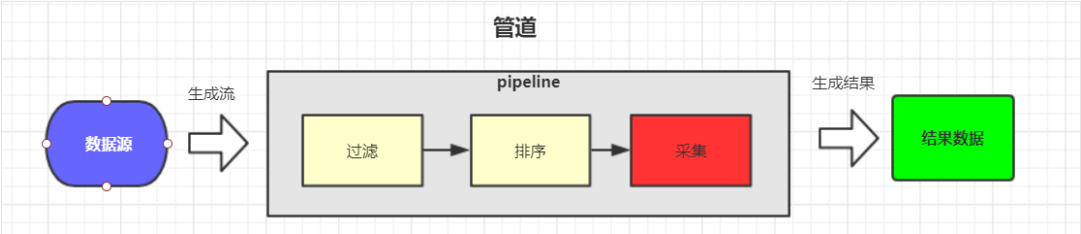
- 接口中标注了 @FunctionalInterface 注解
- 接口中只有一个抽像方法 会被编译器自动认识成函数式接口
- 接口中有一个抽像方法，同时包含了Object类的其它抽像方法也会被识别成抽像接口

lambada表达式三种编写方式：

1. expression/ɪk'spreʃn/：单条语句表达式
2. statement：语句块
3. reference：方法引用

三、Stream执行机制解密

流的执行过程图



流的操作特性

1. stream不存储数据
2. stream不改变源数据
3. stream 不可重复使用

流的操作类型

stream 所有操作组合在一起即变成了管道，管道中有以下两种操作：

- **中间操作**(intermediate /ˌɪntəˈmiːdɪət/): 调用中间操作方法会返回一个新的流。通过连续执行多个操作倒便就组成了Stream中的执行管道（pipeline）。需要注意的是这些管道被添加后并不会真正执行，只有等到调用终值操作之后才会执行。
- **终值操作**(terminal /ˈtɜːmɪn(ə)l/)： 在调用该方法后，将执行之前所有的中间操作，获返回结果结束对流的使用

流的执行顺序说明：其每个元素挨着作为参数去调用中间操作及终值操作，而不是遍历完一个方法，在遍历下一个方法。

☐ 演示说明该特性：

流的并形操作

调用Stream.parallel() 方法可以将流基于多个线程并行执行

☐ 演示说明该特性：

流的生成

- Collection#stream

- Arrays#stream
- Stream#Stream
- Stream#generate

Stream 中的常用API及场景

方法	描述	操作类型
filter	接收一个Boolean表达示来过滤元素	中间操作
map	将流中元素 1:1 映谢成另外一个元素	中间操作
mapToInt	将流中元素映谢成int，mapToLong、mapToDouble操作类似目的减少 装箱拆箱带来的损耗	中间操作
flatMap	如map时返回的是一个List，将会进一步拆分。详见flatMap示例	中间操作
forEach	遍历流中所有元素	终值操作
sorted	排序	中间操作
peek	遍历流中所有元素 ，如forEach不同在于不会结束流	中间操作
toArray	将流中元素转换成一个数组返回	终值操作
reduce	归约合并操作	中间操作
collect	采集数据，返回一个新的结果 参数说明： Supplier<R>： 采集需要返回的结果 BiConsumer<R, ? super T>：传递结果与元素进行合并。 BiConsumer<R, R>：在并发执行的时候 结果合并操作。详见 collec示例	终值操作
distinct	基于equal 表达示去重	中间操作
max	通过比较函数 返回最大值	终值操作
anyMatch	流中是否有任一元素满足表达示	终值操作
allMatch	流中所有元素满足表达示返回true	终值操作
noneMatch	与allMatch 相反，都不满足的情况下返回 true	终值操作
findFirst	找出流中第一个元素	终值操作
of	生成流	生成流操作
iterate	基于迭代生成流	生成流操作
generate	基于迭代生成流，与iterate 不同的是不 后一元素的生成，不依懒前一元素	生成流操作
concat	合并两个相同类型的类	生成流操作

Stream 示例：

```
 @Test
    public void filterTest() {
        appleStore.stream().filter(a ->
a.getColor().equals("red")).forEach(a -> {
            System.out.println(a.getColor());
```

```
        });
    }
    @Test
    public void mapTest() {
        appleStore.stream().map(a ->
a.getOrigin()).forEach(System.out::println);
    }
    @Test
    public void flatMapTest() throws IOException {
        Stream<String> lines = Files.lines(new File("G:\\git\\tuling-
java8\\src\\main\\java\\com\\tuling\\java8\\stream\\bean\\Order.java").toP
ath());
        lines.flatMap(a -> Arrays.stream(a.split("
"))).forEach(System.out::println);
    }
    @Test
    public void sortedTest() {
        appleStore.stream().sorted((a, b) -> a.getWeight() -
b.getWeight())
                .map(a -> a.getWeight()).forEach(System.out::println);
    }
    @Test
    public void peekTest() {
        appleStore.stream().peek(a -> {
            System.out.println(a.getId());
        }).map(a -> a.getOrigin())
                .peek(System.out::println).forEach(a -> {
        });
    }
    @Test
    public void reduceTest() {
        // 找出最重的那个苹果
        appleStore.stream().reduce((a, b) -> a.getWeight() > b.getWeight()
? a : b)
                .ifPresent(a -> {
                    System.out.println(a.getWeight());
                });
    }
    @Test
    public void collectTest() {
        // 将结果转换成id作为key map<Integer,Apple>
        HashMap<Integer, Apple> map =
appleStore.stream().collect(HashMap::new, (m, a) -> m.put(a.getId(), a),
(m1, m2) -> m1.putAll(m2));
        map.forEach((k, v) -> {
            System.out.println(k);
            System.out.println(v);
        });
        // Map<String,List<Apple>>
```

```
        //  基于颜色分组， 并获取其平均重量
    }
```

Collectors 中的常用API及场景

方法	描述	
toList	转换成list	
toMap	转换成map	
groupingBy	统计分组	
averagingInt	求平均值	
summingInt	求总值	
maxBy	获取最大值	

Collectors 使用例子:

```
 // 获得所有颜色苹果的平均重量
    @Test
public void groupByTest() {
        Collector<Apple, ?, Map<String, Double>> groupCollect =
                Collectors.groupingBy((Apple a) -> a.getColor(),
Collectors.averagingInt((Apple a) -> a.getWeight()));
        appleStore.stream().collect(groupCollect).forEach((k, v) -> {
            System.out.println(k + ":" + v);
        });
    }
```

流的关闭机制

一般情况使用完流之后不需要调用close 方法进行关闭，除非是使用channel FileInputStream 这类的操作需要关闭,可调用 java.util.stream.BaseStream#onClose() 添加关闭监听.