链滴

java【Stream】（高效处理数据）

作者：haxLook

原文链接：https://ld246.com/article/1619518760641

来源网站：链滴

流的基本概念

流可以基本定义为一个数据流，顾名思义就是流水线->入口【数据输入】->中间【数据处理】->出【返回数据结果】

外部迭代、内部迭代

外部迭代表示自己编写过程，内部迭代表示只关注结果，不关注过程

```
@Test
    public void test01() {

        //外部迭代求和->求和

        int[] arrays = {1,2,3,4,56,34};

        int sum = 0;
        for (int i : arrays) {

            sum+= i;
        }
        System.out.println("外部迭代和"+sum);

        //内部迭代求和
        int sum2 = IntStream.of(arrays).sum();
        System.out.println("内部迭代"+sum2);
    }
```

返回结果为【内部迭代和100 ， 外部迭代和100】

中间操作，终止操作

```
@Test
    public void test02() {
        //中间操作
        //外部迭代求和
        /**
         * 我们可以在这里加一个filter【中间操作，返回的是流】
         * 加一个sum【求和，终止操作，返回一般为一个对象类型、基本类型】
         */
        int[] arrays = {1,2,3,4,56,34};

        int sum2 = IntStream.of(arrays).filter(hax->hax<20).sum();

        System.out.println("外部迭代和"+sum2);

    }
```

返回结果为【外部迭代和10】

惰性求值

无终止操作的流不会执行，节省资源

```
@Test
public void test03() {
    //中间操作
    //外部迭代求和
    /**
     * 1.我们可以在这里加一个filter【中间操作，返回的是流】
     *              加一个sum【求和，终止操作，返回一般为一个对象类型、基本类型】
     */
    int[] arrays = {1,2,3,4,56,34};

    int sum2 = IntStream.of(arrays).filter(this::filter).sum();

    System.out.println("外部迭代和"+sum2);

    //2.下面开始惰性求值【无终止操作】=>程序执行会发现先下面的中间操作并没有执行
    IntStream.of(arrays).filter(this::filter);

}

private boolean filter(int num) {
    System.out.println("开始惰性求值");
    return num < 20;
}
```

返回结果

开始惰性求值 开始惰性求值 开始惰性求值 开始惰性求值 开始惰性求值 开始惰性求值 外部迭代和10

流的创建

	相关方法
集合	Collection.stream/parallelStream
数组	Arrays.stream
数字Stream	IntStream/LongStream. range/rangeClosed
	Random.ints/longs/doubles
自己创建	Stream. generate/iterate

```
@Test
public void test01() {
    List<String> arrList =new ArrayList<String>();

    //从集合创建
    //串行流
    arrList.stream();
```

```
        //并行流
        arrList.parallelStream();

        //数组创建
        Arrays.stream(new int[] {1,2,3});
        //数字流
        IntStream.of(new int[] {1,2,3});

        //使用random创建一个无限流
        new Random().ints().limit(10);
        Random random = new Random();
        //使用自己创建流 generate(参数为一个提供者函数接口)
        Stream.generate(random::nextInt).limit(10);

    }
```

流的中间操作

流的中间操作分为了无状态和有状态

无状态，就是不用管前面执行的操作【一般参数只有一个】

有状态，需要根据前面执行的结果在进行下一步的操作【一般参数有二个】

	相关方法
无状态操作	map/ mapToXxx
	flatMap/ flatMapToXxx
	filter
	peek
	unordered
有状态操作	distinct
	sorted
	limit / skip

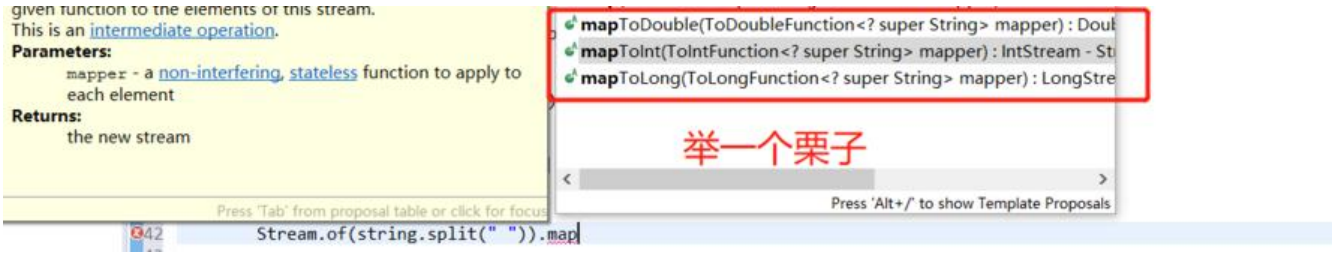
```
@Test
    public void test() {
        /**
         * 中间操作
         */
        String string = "I am form china";
        //将每个单词的长度输出
        Stream.of(string.split(" ")).map(hax->hax.length()).forEach(System.out::println);

        //将每一个单词的字节输出
        //flatMap适合 A-里面包含一个b(b是一个集合)
```

```
        /**
         * 需要注意下面的基本流不是 Stream的子类，需要使用boxed进行装箱操作
         *     IntStream
           lonIntStream
         */

        Stream.of(string.split(" ")).flatMap(hax->hax.chars().boxed()).forEach(System.out::println);

        //peek 用于debug操作 ， foreach是终止操作

        Stream.of(string.split(" ")).peek(System.out::println).forEach(System.out::println);;

        //limit的使用无限吃醋
        new Random().ints().limit(100).forEach(System.out::println);
    }
```

其他操作的栗子

mapToInt

```
String string = "I am form china";

Stream.of(string.split(" ")).mapToInt(mapper>mapper.length())
                        .forEach(System.out::println);
```




```
@Test
    public void test02() {

        /**
         * S unordered(); 产生一个与当前流中元素相同的无序流, 当流本身就是无序或流已经无序, 会
回流本身
         */
        for (int i = 0; i < 100; i++) {
            Stream.of(string.split(" ")).unordered().forEach(System.out::println);
        }
    }
```

unordered返回一个无需流；输出的结果大家会发现却一直是按照顺序的，这是因为对于 unordered() 作不会执行任何操作来显示地对流进行排序。它的作用是消除了流必须保持有序的约束，从而允许后操作使用不必考虑排序的优化。

一般这种操作是用在并行流中， 对于并行流，放宽排序约束有时可以实现更高效的执行 ， 在流有序时 但用户不特别关心该顺序的情况下，使用 unordered 明确地对流进行去除有序约束可以改善某些有态或终端操作的并行性能。

有状态操作中distinct去重，sorted排序

终止操作

	相关方法
非短路操作	forEach/ forEachOrdered
	collect/ toArray
	reduce
	min/ max/ count
短路操作	findFirst/ findAny
	allMatch/ anyMatch/ noneMatch

终止操作中分为了短路操作和非短路操作， 我们看下面举例的非短路操作中的forEachOrdered用于行流中的排序操作。

短路操作：需要等待全部操作结束才可以返回

非短路操作：不需要等待所有的结果操作就可返回

```
@Test
    public void test03() {
        /**
         *终止操作中分为了短路操作和非短路操作
         *foreach不在说了
         *主要看下
         *forEachOrdered
         *
         *带上Ordered一般都是和并行流有关系
         */
        Stream.of(string.split(" ")).parallel().forEach(System.out::print);
        //输出结果 formchinaaml
        Stream.of(string.split(" ")).parallel().forEachOrdered(System.out::print);
        //输出结果 Iamformchina
    }
```

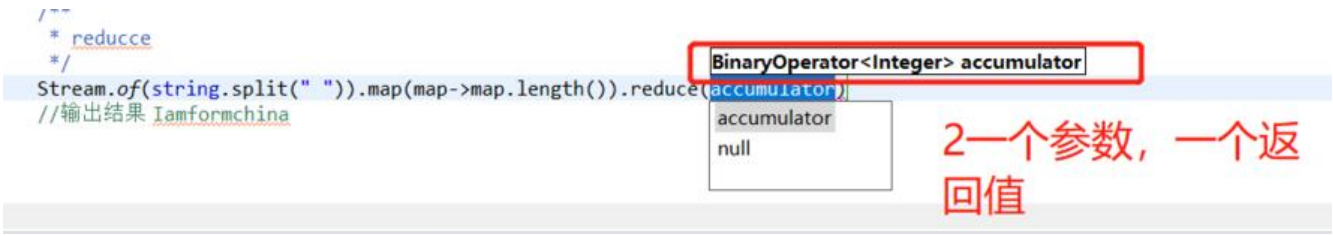
collect

收集器，类似于将数据汇总之类的

```
/**
* collect收集器
*/
List<Integer> collect = Stream.of(string.split(" ")).map(map->map.length())
                                        .collect(Collectors.toList());
System.out.println(collect)
```

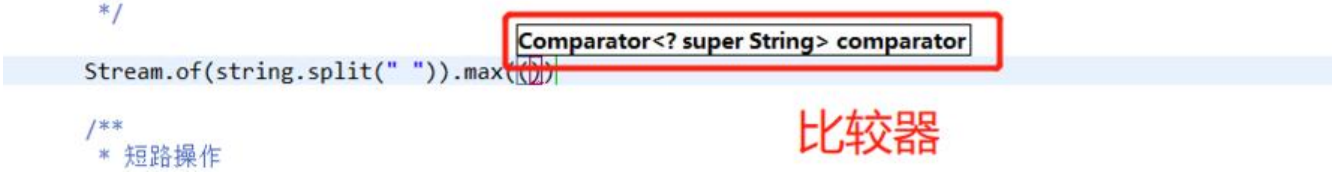
reducce

减少的意思，就是相当于将多个数据合并成一个的意思




```
/**
* reducce求和
*/
      Optional<Integer> reduce = Stream.of(string.split(" ")).map(map->map.length()).reduce(
s1,s2) ->s1+s2);

//返回了一个Optional 也是jdk1.8以后提出来的 一个流

System.out.println(reduce.isPresent()?reduce.get():null);

/**
* reducce求和第二种给与默认值
*/
      Optional<Integer> reduce2 = Stream.of(string.split(" ")).map(map->map.length()).reduce
(s1,s2) ->s1+s2);

//返回了一个Optional 也是jdk1.8以后提出来的 一个流

System.out.println(reduce.isPresent()?reduce.get():null);
```

max




```
/**
* max
*/
      Optional<String> optional = Stream.of(string.split(" ")).max((s1,s2)->s1.length()- s2.leng
h());

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

举一个短路操作的

```
/**
* 短路操作
*/
int asInt = new Random().ints().findFirst().getAsInt();

System.out.println(asInt);
```

Stream的基本操作就差不多，主要是要弄懂原理为主，在工作中多实操，正所谓熟能生巧