前言

zhisheng

在 Flink 应用程序中，无论你的应用程序是批程序，还是流程序，都是上图这种模型，有数据源（source），有数据下游（sink），我们写的应用程序多是对数据源过来的数据做一系列操作，总结如下。

1、**Source**: 数据源，Flink 在流处理和批处理上的 source 大概有 4 类：基于本地集合的 source、基于文件的 source、基于网络套接字的 source、自定义的 source。自定义的 source 常见的有 Apache kafka、Amazon Kinesis Streams、RabbitMQ、Twitter Streaming API、Apache NiFi 等，当然你也可以定义自己的 source。

2、**Transformation**: 数据转换的各种操作，有 Map / FlatMap / Filter / KeyBy / Reduce / Fold / Aggregations / Window / WindowAll / Union / Window join / Split / Select / Project 等，操作很多，可以将数据转换计算成你想要的数据。

3、**Sink**: 接收器，Sink 是指 Flink 将转换计算后的数据发送的地点 ，你可能需要存储下来。Flink 常见的 Sink 大概有如下几类：写入文件、打印出来、写入 Socket 、自定义的 Sink 。自定义的 sink 常见的有 Apache kafka、RabbitMQ、MySQL、ElasticSearch、Apache Cassandra、Hadoop FileSystem 等，同理你也可以定义自己的 Sink。

那么本文将给大家介绍的就是 Flink 中的批和流程序常用的算子（Operator）。

DataStream Operator

我们先来看看流程序中常用的算子。

Map

Map 算子的输入流是 DataStream，经过 Map 算子后返回的数据格式是 SingleOutputStreamOperator 类型，获取一个元素并生成一个元素，举个例子：

```
SingleOutputStreamOperator<Employee> map = employeeStream.map(new
MapFunction<Employee, Employee>() {
    @Override
    public Employee map(Employee employee) throws Exception {
        employee.salary = employee.salary + 5000;
        return employee;
    }
});
map.print();
```

新的一年给每个员工的工资加 5000。

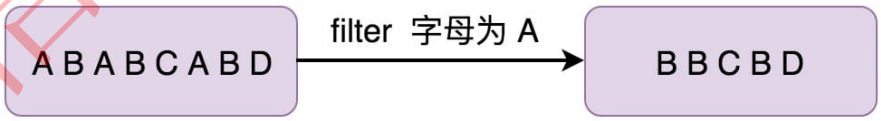
FlatMap

FlatMap 算子的输入流是 DataStream，经过 FlatMap 算子后返回的数据格式是 SingleOutputStreamOperator 类型，获取一个元素并生成零个、一个或多个元素，举个例子：

```
SingleOutputStreamOperator<Employee> flatMap = employeeStream.flatMap(new
FlatMapFunction<Employee, Employee>() {
    @Override
    public void flatMap(Employee employee, Collector<Employee> out) throws
Exception {
        if (employee.salary >= 40000) {
            out.collect(employee);
        }
    }
});
flatMap.print();
```

将工资大于 40000 的找出来。

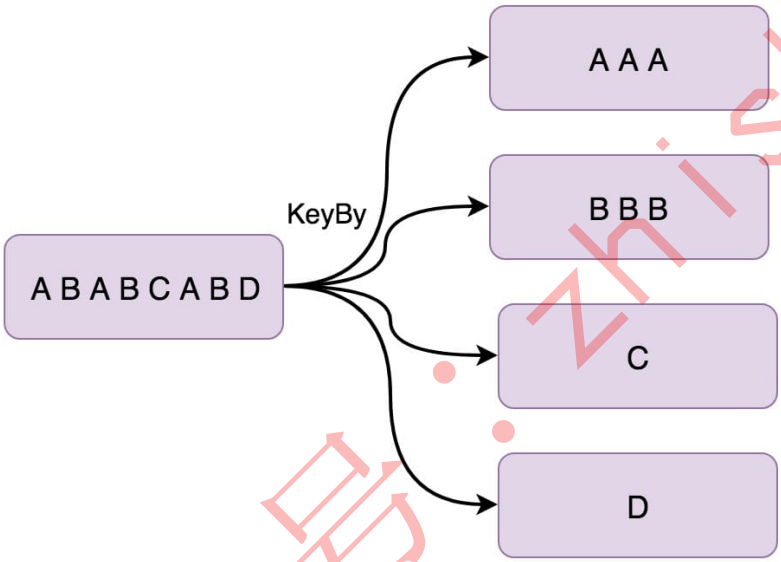
Filter



对每个元素都进行判断，返回为 true 的元素，如果为 false 则丢弃数据，上面找出工资大于 40000 的员工其实也可以用 Filter 来做：

```
SingleOutputStreamOperator<Employee> filter = employeeStream.filter(new
FilterFunction<Employee>() {
    @Override
    public boolean filter(Employee employee) throws Exception {
        if (employee.salary >= 40000) {
            return true;
        }
        return false;
    }
});
filter.print();
```

KeyBy



KeyBy 在逻辑上是基于 key 对流进行分区，相同的 Key 会被分到一个分区（这里分区指的就是下游算子多个并行节点的其中一个）。在内部，它使用 hash 函数对流进行分区。它返回 KeyedDataStream 数据流。举个例子：

```
KeyedStream<ProductEvent, Integer> keyBy = productStream.keyBy(new
KeySelector<ProductEvent, Integer>() {
    @Override
    public Integer getKey(ProductEvent product) throws Exception {
        return product.shopId;
    }
});
keyBy.print();
```

根据商品的店铺 id 来进行分区。

Reduce

Reduce 返回单个的结果值，并且 reduce 操作每处理一个元素总是创建一个新值。常用的方法有 average、sum、min、max、count，使用 Reduce 方法都可实现。

```
SingleOutputStreamOperator<Employee> reduce = employeeStream.keyBy(new
KeySelector<Employee, Integer>() {
    @Override
    public Integer getKey(Employee employee) throws Exception {
        return employee.shopId;
    }
}).reduce(new ReduceFunction<Employee>() {
    @Override
    public Employee reduce(Employee employee1, Employee employee2) throws
Exception {
        employee1.salary = (employee1.salary + employee2.salary) / 2;
        return employee1;
    }
});
reduce.print();
```

上面先将数据流进行 keyby 操作，因为执行 Reduce 操作只能是 KeyedStream，然后将员工的工资做了一个求平均值的操作。

Aggregations

DataStream API 支持各种聚合，例如 min、max、sum 等。 这些函数可以应用于 KeyedStream 以获得 Aggregations 聚合。

```
KeyedStream.sum(0)
KeyedStream.sum("key")
KeyedStream.min(0)
KeyedStream.min("key")
KeyedStream.max(0)
KeyedStream.max("key")
KeyedStream.minBy(0)
KeyedStream.minBy("key")
KeyedStream.maxBy(0)
KeyedStream.maxBy("key")
```

max 和 maxBy 之间的区别在于 max 返回流中的最大值，但 maxBy 返回具有最大值的键， min 和 minBy 同理。

Window

Window 函数允许按时间或其他条件对现有 KeyedStream 进行分组。 以下是以 10 秒的时间窗口聚合：

```
inputStream.keyBy(0).window(Time.seconds(10));
```

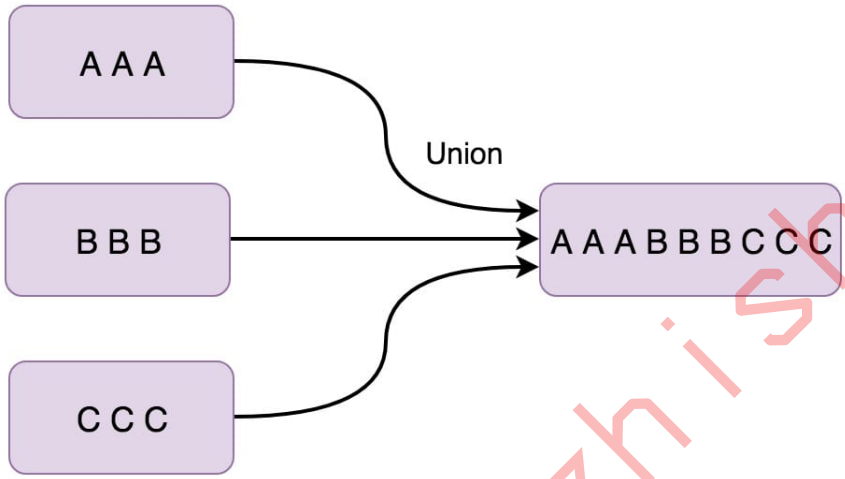
有时候因为业务需求场景要求：聚合一分钟、一小时的数据做统计报表使用。

WindowAll

WindowAll 将元素按照某种特性聚集在一起，该函数不支持并行操作，默认的并行度就是 1，所以如果使用这个算子的话需要注意一下性能问题，以下是使用例子：

```
inputStream.keyBy(0).windowAll(Time.seconds(10));
```

Union



Union 函数将两个或多个数据流结合在一起。 这样后面在使用的时候就只需使用一个数据流就行了。如果我们将一个流与自身组合，那么组合后的数据流会有两份同样的数据。

```
inputStream.union(inputStream1, inputStream2, ...);
```

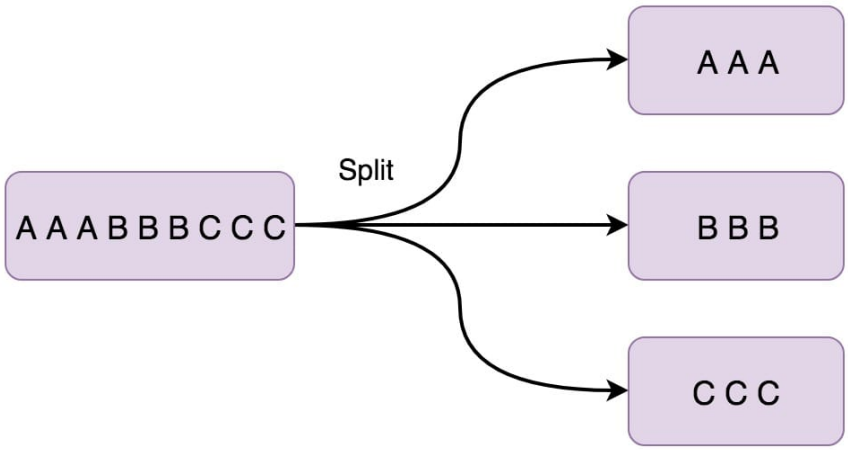
Window Join

我们可以通过一些 key 将同一个 window 的两个数据流 join 起来。

```
inputStream.join(inputStream1)
    .where(0).equalTo(1)
    .window(Time.seconds(5))
    .apply (new JoinFunction () {...});
```

以上示例是在 5 秒的窗口中连接两个流，其中第一个流的第一个属性的连接条件等于另一个流的第二个属性。

Split

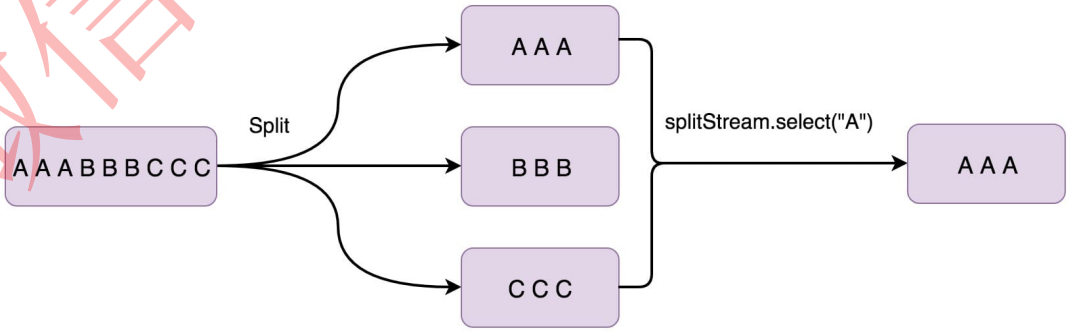


此功能根据条件将流拆分为两个或多个流。 当你获得混合流然后你可能希望单独处理每个数据流时，可以使用此方法。

```
SplitStream<Integer> split = inputStream.split(new OutputSelector<Integer>
() {
    @Override
    public Iterable<String> select(Integer value) {
        List<String> output = new ArrayList<String>();
        if (value % 2 == 0) {
            output.add("even");
        } else {
            output.add("odd");
        }
        return output;
    }
});
```

上面就是将偶数数据流放在 even 中，将奇数数据流放在 odd 中。

Select



上面用 Split 算子将数据流拆分成两个数据流（奇数、偶数），接下来你可能想从拆分流中选择特定流，那么就得搭配使用 Select 算子（一般这两者都是搭配在一起使用的），

```
SplitStream<Integer> split;
DataStream<Integer> even = split.select("even");
DataStream<Integer> odd = split.select("odd");
DataStream<Integer> all = split.select("even","odd");
```

我们就介绍这么些常用的算子了，当然肯定也会有遗漏，具体还得查看官网 https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/stream/operators/ 的介绍。

DataSet Operator

上面介绍了 DataStream 的常用算子，其实上面也有一些算子也是同样适合于 DataSet 的，比如 Map、FlatMap、Filter 等（相同的我就不再重复了）；也有一些算子是 DataSet API 独有的，比如 DataStream 中分区使用的是 KeyBy，但是 DataSet 中使用的是 GroupBy。

First-n

```
DataSet<Tuple2<String, Integer>> in =
// 返回 DataSet 中前 5 的元素
DataSet<Tuple2<String, Integer>> out1 = in.first(5);

// 返回分组后每个组的前 2 元素
DataSet<Tuple2<String, Integer>> out2 = in.groupBy(0)
                                          .first(2);

// 返回分组后每个组的前 3 元素（按照上升排序）
DataSet<Tuple2<String, Integer>> out3 = in.groupBy(0)
                                          .sortGroup(1, Order.ASCENDING)
                                          .first(3);
```

还有一些，感兴趣的可以查看官网 https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/batch/dataset_transformations.html。

流批统一

一般公司里的业务场景需求肯定不止是只有批计算，也不只是有流计算的。一般这两种需求是都存在的。比如每天凌晨 00:00 去算昨天一天商品的售卖情况，然后出报表给运营或者老板去分析；另外的就是处理实时的数据。

但是这样就会有一个问题，需要让开发掌握两套 API。有些数据工程师的开发能力可能并不高，他们会更擅长写一些 SQL 去分析，所以要是掌握两套 API的话，对他们来说成本可能会很大。要是 Flink 能够提供一种高级的 API，上层做好完全封装，让开发无感知底层到底运行的是 DataSet 还是 DataStream API，这样不管是开发还是数据工程师只需要学习一套高级的 API 就行。

Flink 社区包括阿里巴巴实时计算团队也在大力推广这块，那就是我们的 Flink Table/SQL API，在写这篇文章的时候，Blink 的代码还未合进去 Flink 中，据说要到 Flink 1.9 版本才能够正式合进去，期待阿里实时计算团队为社区带来的巨大贡献。

对于开发人员来说，流批统一的引擎（Table & SQL API）在执行之前会根据运行的环境翻译成 DataSet 或者 DataStream API。因为这两种 API 底层的实现有很大的区别，所以在统一流和批的过程中遇到了不少挑战。

- 理论基础：动态表
- 架构改进（统一的 Operator 框架、统一的查询处理）
- 优化器的统一
- 基础数据结构的统一
- 物理实现的共享

关于 Table 和 SQL API，我们会在进阶篇中讲解，敬请期待！

总结

本文讲解了 Flink 中 DataStream 和 DataSet API 中常使用的算子（Operator），其实我们在 Flink 这层做数据计算主要依靠的就是这些，然后再加上一些自己的业务场景需求，所以建议大家对这些常见的算子还是多熟悉一下，知道它的使用场景，然后自己根据一些简单的需求利用这些算子写一些简单的 demo，做到学以致用。另外就是讲解了下社区的发展之路——流批统一，阿里实时计算团队在这块发力很足，包括阿里内部大量的计算 Job 如今也都是使用 Table 和 SQL API，他们现在很少使用 DataStream API 了，所以未来我们也需要去深入了解下 Table 和 SQL API。