09 | 流数据操作：最基本的流计算功能

在前面的两个模块中，我们讨论的主要是构成流计算系统的基础框架。我们有了这个框架，接下来就应该用它解决实际的实时计算问题。而解决实际问题的过程，落到实处就是实现某种具体算法的过程。

所以在第三模块，我将依次讲解实时流计算系统中的**几类算法问题**。在以后的流计算应用开发过程中，你所面对的计算问题，都将八九不离十地归于这几类问题中的一种或多种。因此，对这几类问题进行分析归纳，总结出特定的算法模式，这是非常有意义的。

那么今天，我们就先来看**第一类算法问题**，即流数据操作的问题。

什么是流数据操作

流数据操作应该说是流计算系统与生俱来的能力，它是针对数据流的“转化”或“转移”处理。流数据操作的内容主要包括四类。

- 一是**流数据的清洗、规整和结构化**。比如提取感兴趣字段、统一数据格式、过滤不合条件事件。
- 二是**流数据的关联及合并**。比如在广告转化率分析中，将“点击”事件流和“安装”事件流关联起来。
- 三是**流数据的分发和并行处理**。比如将一个包含了来自不同设备事件的数据流，按照设备id分发到不同的流中进行处理。
- 四是**流数据的转移和存储**。比如将数据从 Kafka 转移到数据库里。

虽然不同系统实现以上四类流数据操作的具体方法不尽相同，但经过多年的实践和经验积累，业界针对流数据操作的目标和手段都有了一定的共识，并已逐步形成一套通用的 API 集合，几乎所有的流计算平台都会提供这些 API 的实现。比如：

- 针对**流数据的清洗、规整和结构化**，抽象出 filter、map、flatMap、reduce 等方法；
- 针对**流数据的关联及合并**，抽象出 join、union 等方法；
- 针对**流数据的分发和并行处理**，抽象出 keyBy 或 groupBy 等方法；
- 针对**流数据的转移和存储**，则抽象出 foreach 等方法。

这些 API 的功能各不相同，但它们在一起共同构成了一个灵活操作流数据的方法集合。

所以接下来，我们就选出几个最重要，且能够覆盖日常大多数使用场景的 API ，来对流数据操作这类算法问题，进行详细讲解。

过滤 filter

首先是过滤 filter 。顾名思义，“过滤”就是在数据流上筛选出符合条件的数据。这个方法通常用于剔除流数据中你不想要的数据，比如不合预期的事件类型、不完整的数据记录等。或者，你也可以用这个方法来对流数据进行采样，比如只保留 1/10 的流数据，从而减少需要处理的数据量。

下面举一个具体的例子来讲下如何使用 filter 方法。比如，我们现在需要监控仓库的环境温度，在火灾发生前提前预警以避免火灾，那么我们就可以采用过滤功能，从来自于传感器的环境温度事件流中，过滤出温度高于 100 摄氏度的事件。

这里我们使用 Flink 来实现。如果你暂时还不熟悉 Flink 的话也没有关系，这里的代码很简单，只需要先了解下这些 API 的使用形式即可。另外，本课程后面还有专门的课时讲解 Flink 。

```
DataStream<JSONObject> highTemperatureStream = temperatureStream.filter(x -> x.getDouble("tempera
```

在上面代码中，lambda 表达式“x->x.getDouble("temperature")>100”即过滤火灾高温事件的条件。

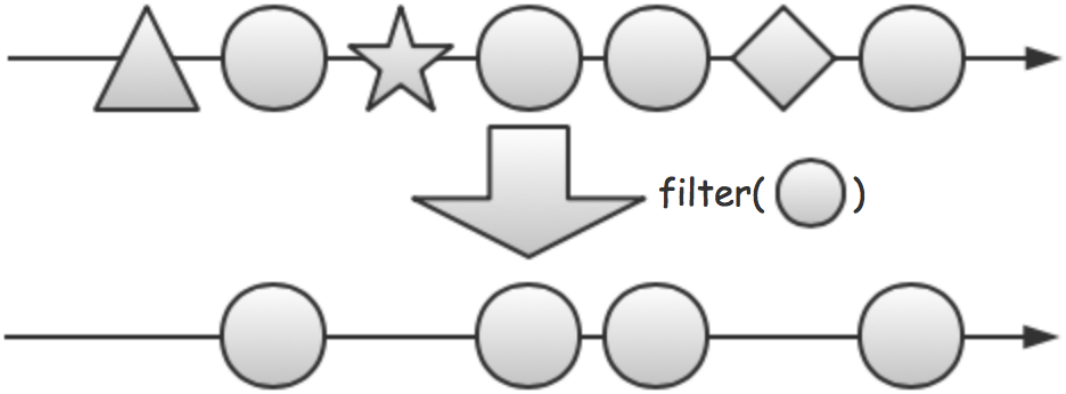


图 1 过滤操作

就像图 1 展示的一样，过滤操作的作用，是将一个具有多种形状的数据流，转化为只含圆形的数据流。当然，你在实际开发中，可以将“形状”替换为任何东西。比如，上面监控仓库环境温度的例子，“圆形”就对应着“高温事件”。

映射 map

“映射”用于将数据流中的每条数据转化为新的数据。它最大的价值在于对流数据进行信息增强，也就是将额外的信息附加到数据流中的数据上。比如，你只对哪些字段感兴趣、需要将数据转化为哪种格式、给数据添加一个新的字段等，这些“信息”在原来的流数据里是没有的，你可以通过 map 方法将这些信息附加到流数据上。

下面同样以仓库环境温度监控为例，来讲解 map 的使用方法。不过，这次我们不是将高温事件过滤出来，而是采用数据工程师在做特征工程时常用的一种操作，也就是“二值化”。

我们在原始环境温度事件中，添加一个新的布尔（boolean）类型字段，用于表示该事件是否是高温事件。同样，使用 Flink 实现如下：

```
DataStream<JSONObject> enhancedTemperatureStream = temperatureStream.map(x -> {
    x.put("isHighTemperature", x.getDouble("temperature") > 100);
    return x;
});
```

上面示意代码的 lambda 表达式中，通过原始事件的 temperature 字段判断是否为高温事件，然后将结果附加到事件上，最后返回附加了高温信息的事件。

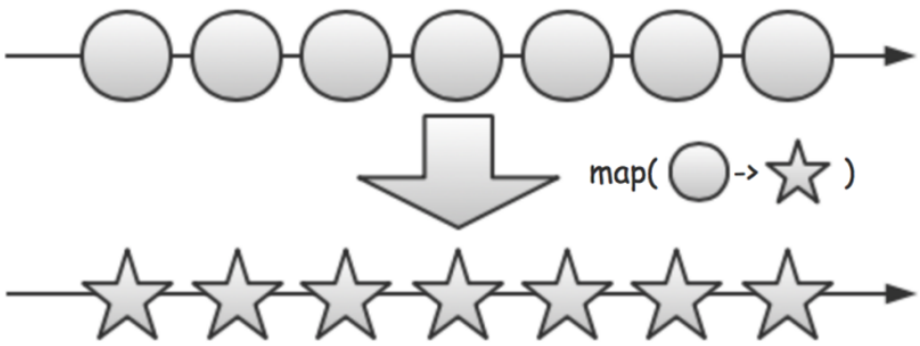


图 2 映射操作

上图 2 展示了映射操作的作用，它将一个由圆形组成的数据流，转化为了五角星形状的数据流。同样在实际开发中，我们可以将“形状”具象为任何东西。

展开映射 flatMap

“展开映射”用于将数据流中的每条数据转化为 N 条新数据。相比 map 而言， flatMap 是个更加灵活的方法，因为 map 只能 1 对 1 地对数据流元素进行转化，而 flatMap 能 1 对 N 地对数据流元素进行转化。

flatMap 最大的作用体现在“flat”上，也就是“展开摊平”。它最典型的使用场景就是，比如原本数据流中的数据有一个字段是数组，现在你需要将这个数组里的每个元素拆解开，然后分成一条条单独的数据，并形成一个新的数据流。

下面举一个 flatMap 在社交活动分析中使用的例子。现在有一组代表用户信息的数据流，其中每条数据记录了用户（用 user 字段表示）及其好友列表（用 friends 数组字段表示）的信息。现在我们要分析每个用户与他的每一个好友之间的亲密程度，以判断他们之间是否是“塑料兄弟”或者“塑料姐妹”。

所以我们先要将用户和它的好友列表一一展开，展开后的每条数据代表了用户和他的其中一个好友之间的关系。下面是采用 Flink 实现的例子。

```
DataStream<String> relationStream = socialWebStream.flatMap(new FlatMapFunction<JSONObject, Strin
    @Override
    public void flatMap(JSONObject value, Collector<String> out) throws Exception {
        List<String> collect = value.getJSONArray("friends").stream()
                .map(y -> String.format("%s->%s", value.getString("user"), y))
                .collect(Collectors.toList());
        collect.forEach(out::collect);
    }
});
```

上面代码的 flatMap 方法中，我们使用 Java 8 的流式 API，将用户的好友列表 friends 展开，与用户形成一对对的好友关系记录（用"%s->%s"格式表示），最终由 out::collect 收集起来，写入输出数据流中。

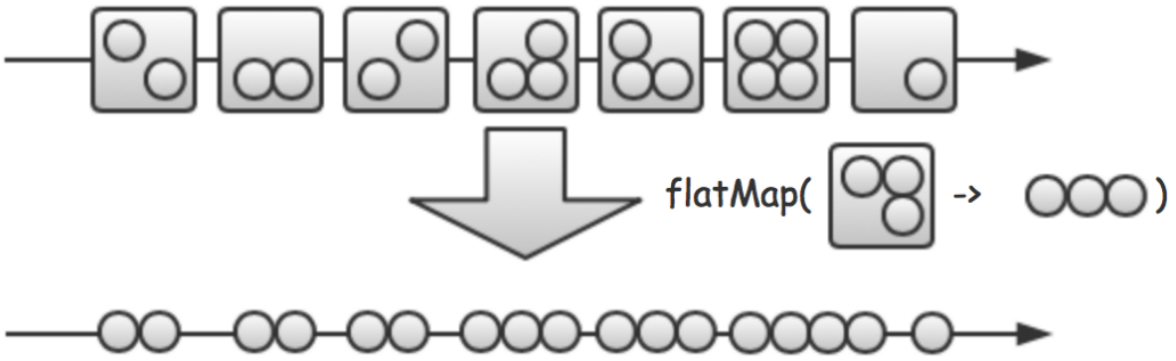


图 3 展开映射操作

图 3 展示了展开映射操作的作用，它将一个由包含小圆形在体内的正方形组成的数据流，展开转化为由小圆形组成的数据流。

在实际开发过程中，我们还经常使用 flatMap 实现 Map/Reduce 或 Fork/Join 计算模式中的 Map 或 Frok 操作。并且更有甚者，由于 flatMap 的输出元素个数能够为 0，所以我们有时候连 Reduce 或 Join 操作也可以使用 flatMap 操作实现。比如，在后面第 20 课时讲解用 Flink 实现风控系统时，你就会看到具体如何用 flatMap 实现针对流式处理的 Map/Reduce 计算模式。这里我们暂时就不展开了。

聚合 reduce

“聚合”用于将数据流中的数据按照指定方法进行聚合。它最典型的业务场景是，比如计算一段时间窗口内的订单数量、交易总额、人均消费额等。

由于流数据具有时间序列的特征，所以聚合操作不能像诸如 Hadoop 等批处理计算框架那样作用在整个数据集上。换言之，**流数据的聚合操作必然是指定了窗口，或者说这样做才有更加实际的意义。这些窗口可以基于时间、事件或会话（session）等。**

同样以社交活动分析为例，这次我们需要每秒钟统计一次 10 秒内用户活跃事件数。使用 Flink 实现如下。

```
DataStream<Tuple2<String, Integer>> countStream = socialWebStream
    .map(x -> Tuple2.of("count", 1))
    .returns(Types.TUPLE(Types.STRING, Types.INT))
    .timeWindowAll(Time.seconds(10), Time.seconds(1))
    .reduce((count1, count2) -> Tuple2.of("count", count1.f1 + count2.f1));
```

上面的代码片段中，socialWebStream 是用户活跃事件流，我们使用 timeWindowAll 指定每隔 1 秒钟，对 10 秒钟窗口内的数据进行一次计算。而 reduce 方法的输入是一个用于求和的 lambda 表达式。在实际执行时，这个求和 lambda 表达式会依次将每条数据与前一次计算的结果相加，最终完成对窗口内全部流数据的求和计算。

如果将求和操作换成其他“二合一”的计算，则可以实现相应功能的聚合运算。由于使用了窗口，所以聚合后流的输出不再是像 map 运算那样逐元素地输出，而是每隔一段时间才会输出窗口内的聚合运算结果。

比如前面的示例代码中，就是每隔 1 秒钟输出 10 秒钟窗口内的聚合计算结果。

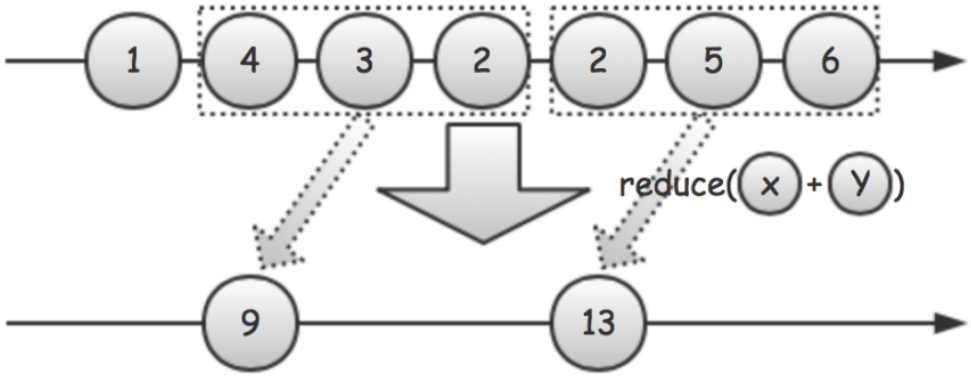


图 4 聚合操作

图 4 展示了聚合操作的作用，它将一个由带有数值的圆形组成数据流，以 3 个元素为窗口，进行求和聚合运算，并输出为新的数据流。在实际开发过程中，我们可选择不同的窗口实现、不同的窗口长度、不同的聚合内容、不同的聚合方法，从而在流数据上实现各种各样的聚合操作。

关联 join

“关联”用于将两个数据流中满足特定条件的数据对组合起来，再按指定规则形成新数据，最后将新数据添加到输出数据流。

在关系型数据库中，关联操作是非常常用的操作手段，这是由关系型数据库的设计理念，也就是数据库的三种设计范式所决定的。而在流数据领域，由于数据来源的多样性和在时序上的差异性，数据流之间的关联也成为一种非常自然的需求。

但相比关系型数据库表间 join 操作，流数据的关联在语义和实现上都更加复杂些。由于流的无限性，只有在类似于“一对一”等非常受限的使用场景下，不限时间窗口的关联设计和实现才有意义。大多数使用场景下，我们需要引入“窗口”来对关联的流数据进行时间同步，即只对两个流中处于指定时间窗口内的数据进行关联操作。

即使引入了窗口，流数据的关联依旧复杂。当窗口时间很长，窗口内的数据量很大（需要将部分数据存入磁盘），而关联的条件又比较宽泛（比如关联条件不是等于而是大于）时，那么流之间的关联计算将非常慢（不是相对于关系型数据库慢，而是相对于实时计算的要求慢），基本上你也别指望能够非常快速地获得两个流关联的结果了。

同样以社交网络分析为例子，这次我们需要将两个不同来源的事件流，按照用户 id 将它们关联起来，汇总为一条包含用户完整信息的数据流。以下就是用 Flink 实现这个功能的示意代码。

```
DataStream<JSONObject> joinStream = socialWebStream.join(socialWebStream2)
    .where(x1 -> x1.getString("user"))
    .equalTo(x2 -> x2.getString("user"))
    .window(TumblingEventTimeWindows.of(Time.seconds(10), Time.seconds(1)))
    .apply((x1, x2) -> {
        JSONObject res = new JSONObject();
        res.putAll(x1);
        res.putAll(x2);
        return res;
    });
```

上面的代码片段中， socialWebStream 和 socialWebStream2 分别是两个来源的用户事件流，我们使用 where 和 equalTo 指定了关联的条件，即按照 user 字段相等的条件进行关联。然后使用 window 指定每隔 1 秒钟，对 10 秒钟窗口内的数据进行关联计算。最后是 apply 方法，指定了合并计算的方法。

流的关联是一个我们经常想用，但又容易让人头疼的操作。因为稍不注意，关联操作的性能就会惨不忍睹。关联操作需要保存大量的状态，尤其是窗口越长，需要保存的数据越多。因此当使用流数据的关联功能时，应尽可能让窗口较短。

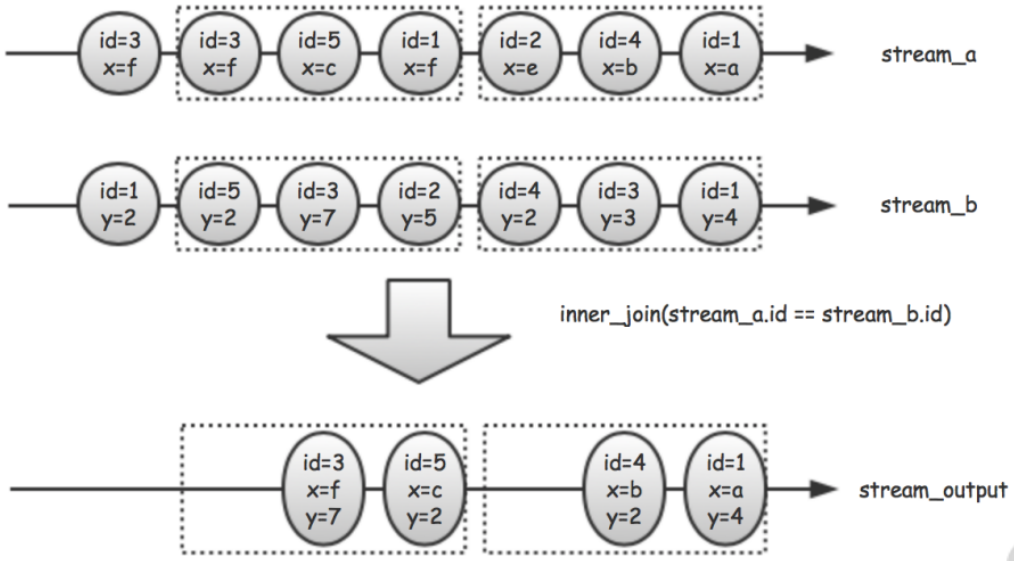


图 5 关联操作

图 5 展示了采用内联接（inner join）的关联操作，它将两个各带 id 和部分字段的数据流，分成相同的时间窗口后，按照 id 相等进行内联接关联，最后输出两个流内联接后的数据流。

分组 key by

接下来我们再来看下流计算中非常重要的分组 key by 操作。如果说各种流计算框架最终能够实现分布式计算，实现高并发和高吞吐，那么最大的功臣莫过于“分组”操作的实现。

“分组”操作是实现并行流计算的最主要手段，它将流划分为不相交的分区流，之后分组键相同的消息会被划分到相同的分区流中。并且，各个分区流在逻辑上互不干扰，具有各自独立的运行时上下文。这就带来两个非常大的好处。

其一，流分组后，能够被分配到不同的计算节点上执行，从而实现了 CPU、内存、磁盘等资源的分布式使用和扩展。

其二，分区流具有独立的运行时上下文，就像线程局部量一样，对于涉及运行时状态的流计算任务来说，极大地简化了安全处理并发问题的难度。

以电商场景为例，假设我们要在“双十一抢购”那天，实时统计各个商品的销量以展现在监控大屏上。使用 Flink 实现如下。

```
DataStream<Tuple2<String, Integer>> keyedStream = transactionStream
    .map(x -> Tuple2.of(x.getString("product"), x.getInteger("number")))
    .returns(Types.TUPLE(Types.STRING, Types.INT))
    .keyBy(0)
    .window(TumblingEventTimeWindows.of(Time.seconds(10)))
    .sum(1);
```

在上面的代码中， transactionStream 代表了交易数据流，在取出了分别代表商品和销量的 product 和 number 字段后，我们使用 keyBy 方法根据商品对数据流进行分组，然后每 10 秒统计一次 10 秒内的各商品销售总量。

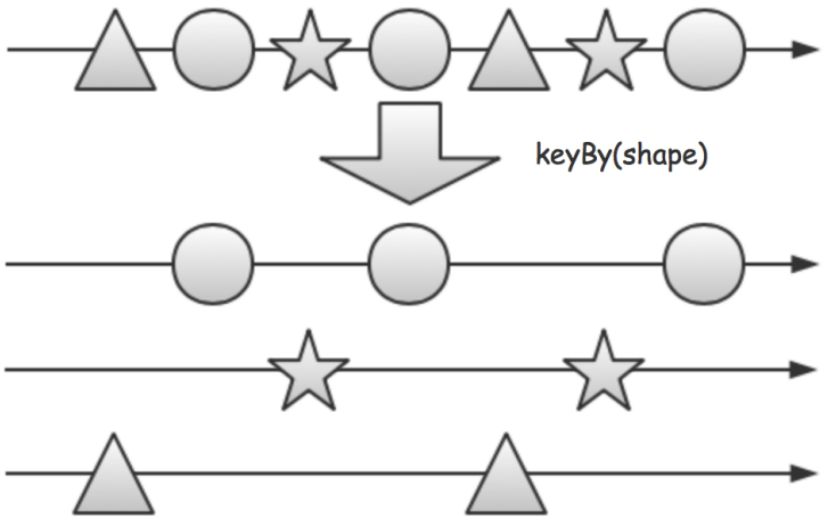


图 6 分组操作

图 6 展示了数据流的分组操作。通过分组操作，将原本包含多种形状的数据流，划分为了多个包含单一形状的数据流。当然，这里的“多个”是指逻辑上的多个，它们在物理上可以是多个流，也可以是一个流，这就与具体的并行度设置有关了。

遍历 foreach

最后，我们来看下流数据的归宿，即遍历 foreach 操作。“遍历”是对数据流的每个元素执行指定方法的过程。遍历与映射非常相似但又非常不同。

说相似是因为 foreach 和 map 都是将一个表达式作用在数据流上，只不过 foreach 使用是“方法”（没有返回值的函数），而 map 使用的是“函数”。

说不同是因为 foreach 和 map 语义大不相同。从 API 语义上来讲，map 作用是对数据流进行转换，但 foreach 并非对数据流进行转换，而是“消费”掉数据流。也就是说，数据流在经过 foreach 后也就终结了。所以我们通常使用 foreach 操作对数据流进行各种 IO 操作，比如写入文件、存入数据库、打印到显示器等。

下面的 Flink 示例代码以及图 7 均展示了将数据流打印到显示屏的功能。

```
transactionStream.addSink(new PrintSinkFunction<>()).name("Print to Std. Out")
```

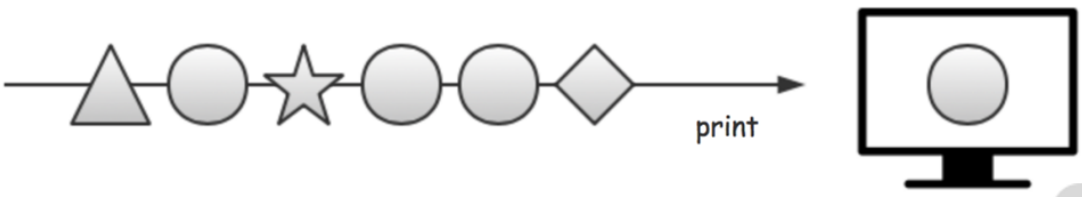


图 7 遍历操作

到此为止，我们讨论了过滤 filter、映射 map、展开映射 flatMap、聚合 reduce、关联 join、分组key by、遍历 foreach 这 7 个通用的流数据操作 API。这 7 个 API 是最基础的流式编程接口，几乎所有的开源流计算框架都提供了这些 API 的实现，而其他功能更丰富的 API 也会构建在这些方法基础之上。

流数据操作 API 总结

最后，为了更加清晰地理解流数据操作，我这里用一个表格对今天讲到的各个 API 做了一个比较和总结。

流数据操作类别	API 名称	功能	意义	适用场景
过滤	filter	从数据流中筛选出符合指定条件的数据	剔除无用的数据，降低处理的数据量	1.剔除数据流中无用数据、错误数据等 2.对流数据进行采样
映射	map	将数据流中的每条数据转化为新的数据	对流数据进行信息增强	给数据流中的数据删除旧字段、添加新字段、进行格式转化等
展开映射	flatMap	将数据流中的每条数据转化为 N 条新数据。	可以将数据流中的一条数据转化为与之相关的多条数据，也可以将多条相关的数据转化为一条数据	1.将流数据的数组字段展开 2.实现流式处理的 Map/Reduce 计算模式
聚合	reduce	将数据流中的数据按照指定方法进行聚合	流数据的聚合操作必须指定窗口，这些窗口可以基于时间、事件或会话	针对流数据特定窗口内的数据做聚合计算，比如 count、sum、avg 等
关联	join	将两个数据流中满足特定条件的数据对组合起来，按指定规则形成新数据	用一条流对另外一条流进行信息增强	根据相同的业务主键，将两个各自包含一部分信息的数据流合并成一个数据流
分组	keyBy	将数据流划分为不相交的分区流，之后分组键相同的数据会被划分到相同的分区流中，并且各个分区流在逻辑上互不干扰，具有各自独立的运行时上下文。	1.实现分布式计算，从而实现高并发和高吞吐 2.实现对流根据业务主键的逻辑划分	根据业务主键，将数据划分到不同的逻辑流以及不同的物理节点上执行，从而实现业务数据的并行计算
遍历	foreach	对数据流的每条数据执行指定的方法	“消费”掉数据流中的数据，是流的“归宿”	对数据流进行各种 IO 操作，比如写入文件、存入数据库、打印到显示器等

小结

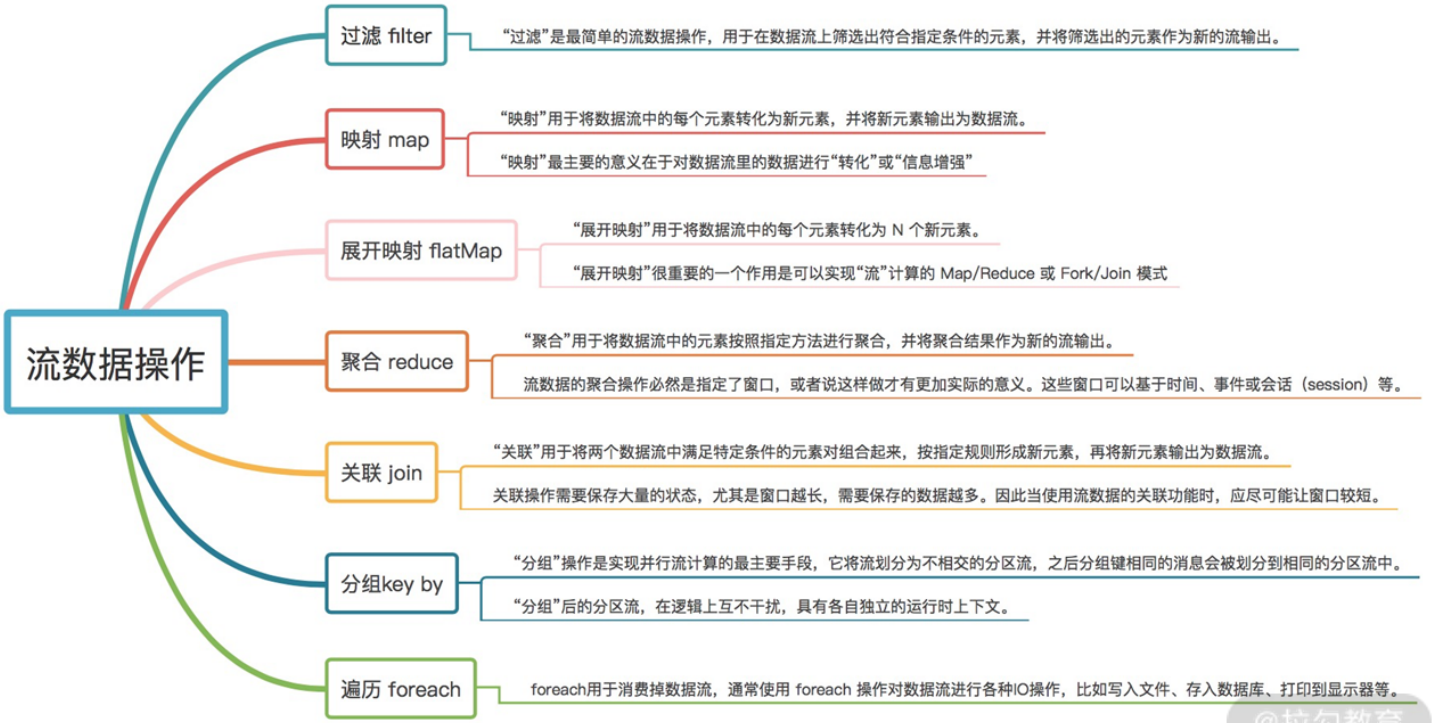
今天，我们讨论了使用流计算技术可以解决的第一类算法问题，即流数据操作。

应该说，今天讲解的流数据操作 API ，既是流计算系统的基本功能，也是实现更复杂算法和功能的基础。在日常开发中，我们经常会使用到流数据操作。比如，大数据领域有个专门的岗位就是“ETL 工程师”，对于“ETL 工程师”而言，他们不可避免地会用到今天所讨论的这些 API 。

目前，有一些开源流计算框架（比如 Flink），直接提供了更方便好用的 SQL 来实现流数据操作，这当然是非常好的新功能。但它们在经过 SQL 层的解析后，最终也会对应到今天所讨论的这些相对底层的 API 。所以，如果你以前没有接触过这类流式编程 API 的话，今天的内容就需要好好理解下了。因为这些 API 以后你会经常用到，而且需要灵活地运用。

最后留一个小问题，你知道在 Flink 中都有哪几种 join 操作，以及每一种 join 操作的设计意图是怎样的呢？可以将你的想法或问题发表在留言区，我看到后会进行解答，或者在后面的课程中进一步补充说明。

下面是本课时的内容脑图，以便于你理解。


流数据操作
过滤 filter
“过滤”是最简单的流数据操作，用于在数据流上筛选出符合指定条件的元素，并将筛选出的元素作为新的流输出。
映射 map
“映射”用于将数据流中的每个元素转化为新元素，并将新元素输出为数据流。
“映射”最主要的意义在于对数据流里的数据进行“转化”或“信息增强”
展开映射 flatMap
“展开映射”用于将数据流中的每个元素转化为 N 个新元素。
“展开映射”很重要的一个作用是可以实现“流”计算的 Map/Reduce 或 Fork/Join 模式
聚合 reduce
“聚合”用于将数据流中的元素按照指定方法进行聚合，并将聚合结果作为新的流输出。
流数据的聚合操作必然是指定了窗口，或者说这样做才有更加实际的意义。这些窗口可以基于时间、事件或会话（session）等。
关联 join
“关联”用于将两个数据流中满足特定条件的元素对组合起来，按指定规则形成新元素，再将新元素输出为数据流。
关联操作需要保存大量的状态，尤其是窗口越长，需要保存的数据越多。因此当使用流数据的关联功能时，应尽可能让窗口较短。
分组key by
“分组”操作是实现并行流计算的最主要手段，它将流划分为不相交的分区流，之后分组键相同的消息会被划分到相同的分区流中。
“分组”后的分区流，在逻辑上互不干扰，具有各自独立的运行时上下文。
遍历 foreach
foreach用于消费掉数据流，通常使用 foreach 操作对数据流进行各种IO操作，比如写入文件、存入数据库、打印到显示器等。