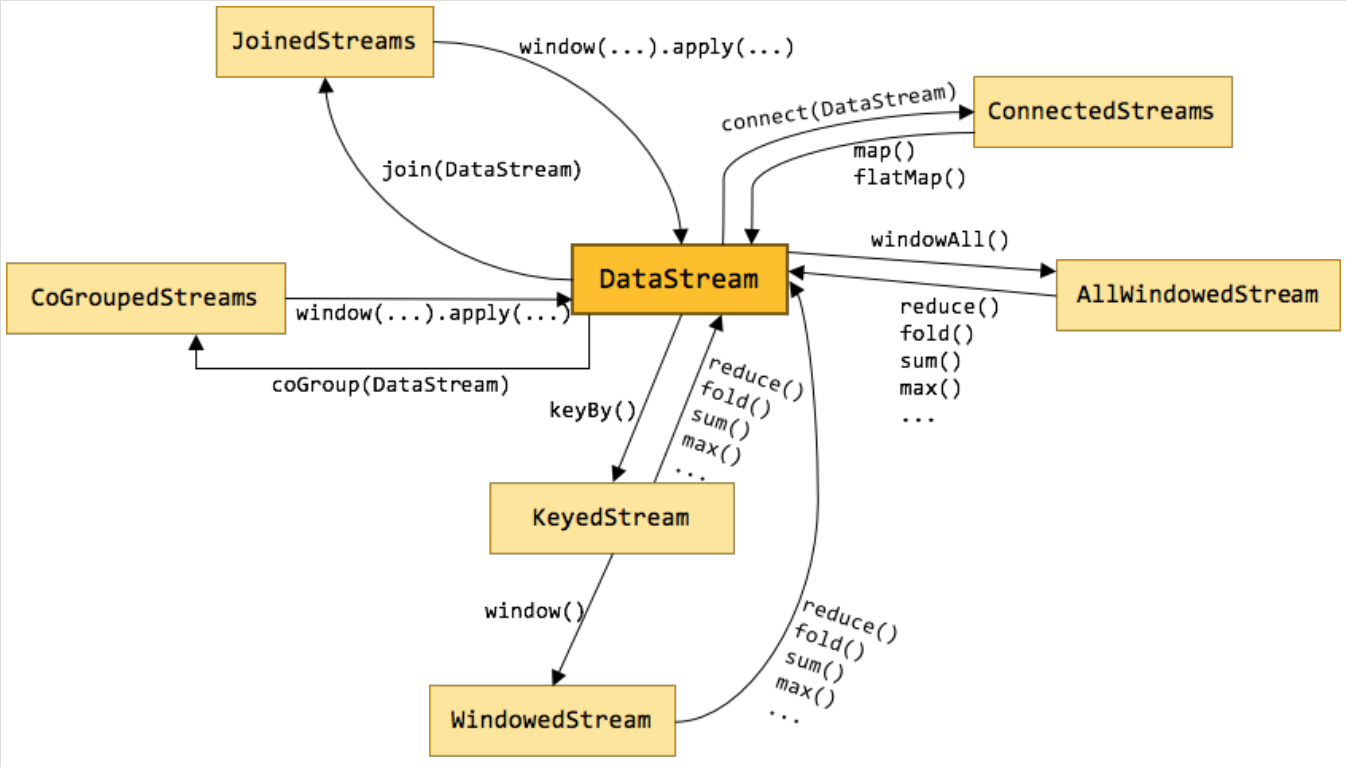
Flink 原理与实现：数据流上的类型和操作

Flink 为流处理和批处理分别提供了 DataStream API 和 DataSet API。正是这种高层的抽象和 flunent API 极大地便利了用户编写大数据应用。不过很多初学者在看到官方 Streaming 文档中那一大坨的转换时，常常会蒙了圈，文档中那些只言片语也很难讲清它们之间的关系。所以本文将介绍几种关键的数据流类型，它们之间是如何通过转换关联起来的。下图展示了 Flink 中目前支持的主要几种流的类型，以及它们之间的转换关系。



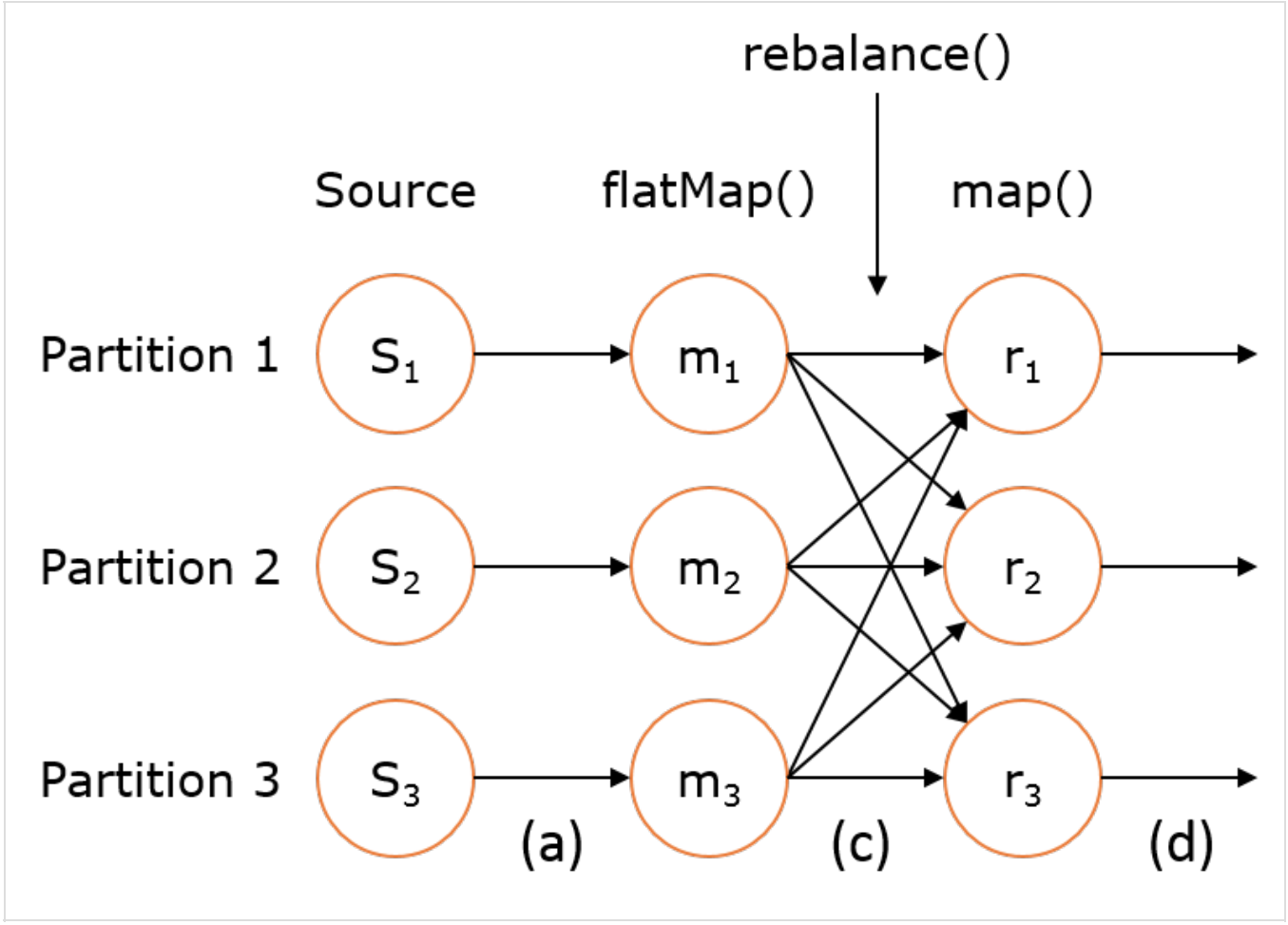
DataStream

`DataStream` 是 Flink 流处理 API 中最核心的数据结构。它代表了一个运行在多个分区上的并行流。一个 `DataStream` 可以从 `StreamExecutionEnvironment` 通过`env.addSource(SourceFunction)` 获得。

DataStream 上的转换操作都是逐条的，比如 `map()`，`flatMap()`，`filter()`。DataStream 也可以执行 `rebalance`（再平衡，用来减轻数据倾斜）和 `broadcaseted`（广播）等分区转换。

```
val stream: DataStream[MyType] = env.addSource(new FlinkKafkaConsumer08[String](...))
val str1: DataStream[(String, MyType)] = stream.flatMap { ... }
val str2: DataStream[(String, MyType)] = stream.rebalance()
val str3: DataStream[AnotherType] = stream.map { ... }
```

上述 DataStream 上的转换在运行时会转换成如下的执行图：



如上图的执行图所示，DataStream 各个算子会并行运行，算子之间是数据流分区。如 Source 的第一个并行实例（S1）和 flatMap() 的第一个并行实例（m1）之间就是一个数据流分区。而在 flatMap() 和 map() 之间由于加了 rebalance()，它们之间的数据流分区就有3个子分区（m1的数据流向3个map()实例）。这与 Apache Kafka 是很类似的，把流想象成 Kafka Topic，而一个流分区就表示一个 Topic Partition，流的目标并行算子实例就是 Kafka Consumers。

KeyedStream

`KeyedStream`用来表示根据指定的key进行分组的数据流。一个`KeyedStream`可以通过调用`DataStream.keyBy()`来获得。而在`KeyedStream`上进行任何transformation都将转变回`DataStream`。在实现中，`KeyedStream`是把key的信息写入到了transformation中。每条记录只能访问所属key的状态，其上的聚合函数可以方便地操作和保存对应key的状态。

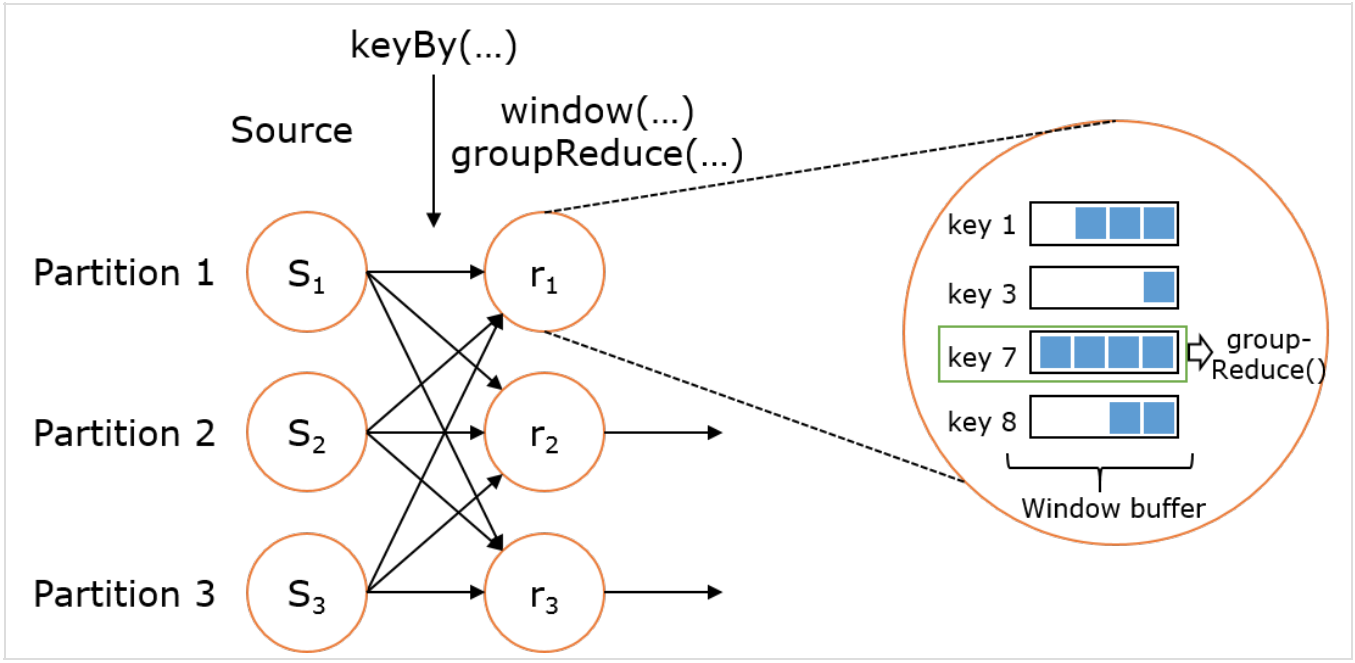
WindowedStream & AllWindowedStream

`WindowedStream`代表了根据key分组，并且基于`WindowAssigner`切分窗口的数据流。所以`WindowedStream`都是从`KeyedStream`衍生而来的。而在`WindowedStream`上进行任何transformation也都将转变回`DataStream`。

```
val stream: DataStream[MyType] = ...
val windowed: WindowedDataStream[MyType] = stream
        .keyBy("userId")
        .window(TumblingEventTimeWindows.of(Time.seconds(5))) // Last 5 seconds of data
```

```
val result: DataStream[ResultType] = windowed.reduce(myReducer)
```

上述 WindowedStream 的样例代码在运行时会转换成如下的执行图：



Flink 的窗口实现中会将到达的数据缓存在对应的窗口buffer中（一个数据可能会对应多个窗口）。当到达窗口发送的条件时（由Trigger控制），Flink 会对整个窗口中的数据进行处理。Flink 在聚合类窗口有一定的优化，即不会保存窗口中的所有值，而是每到一个元素执行一次聚合函数，最终只保存一份数据即可。

在key分组的流上进行窗口切分是比较常用的场景，也能够很好地并行化（不同的key上的窗口聚合可以分配到不同的task去处理）。不过有时候我们也需要在普通流上进行窗口的操作，这就是 `AllWindowedStream`。`AllWindowedStream` 是直接在 `DataStream` 上进行 `windowAll(...)` 操作。AllWindowedStream 的实现是基于 WindowedStream 的（Flink 1.1.x 开始）。Flink 不推荐使用 `AllWindowedStream`，因为在普通流上进行窗口操作，就势必需要将所有分区的流都汇集到单个的Task中，而这个单个的Task很显然就会成为整个Job的瓶颈。

JoinedStreams & CoGroupedStreams

双流 Join 也是一个非常常见的应用场景。深入源码你可以发现，JoinedStreams 和 CoGroupedStreams 的代码实现有80%是一模一样的，JoinedStreams 在底层又调用了 CoGroupedStreams 来实现 Join 功能。除了名字不一样，一开始很难将它们区分开来，而且为什么要提供两个功能类似的接口呢？？

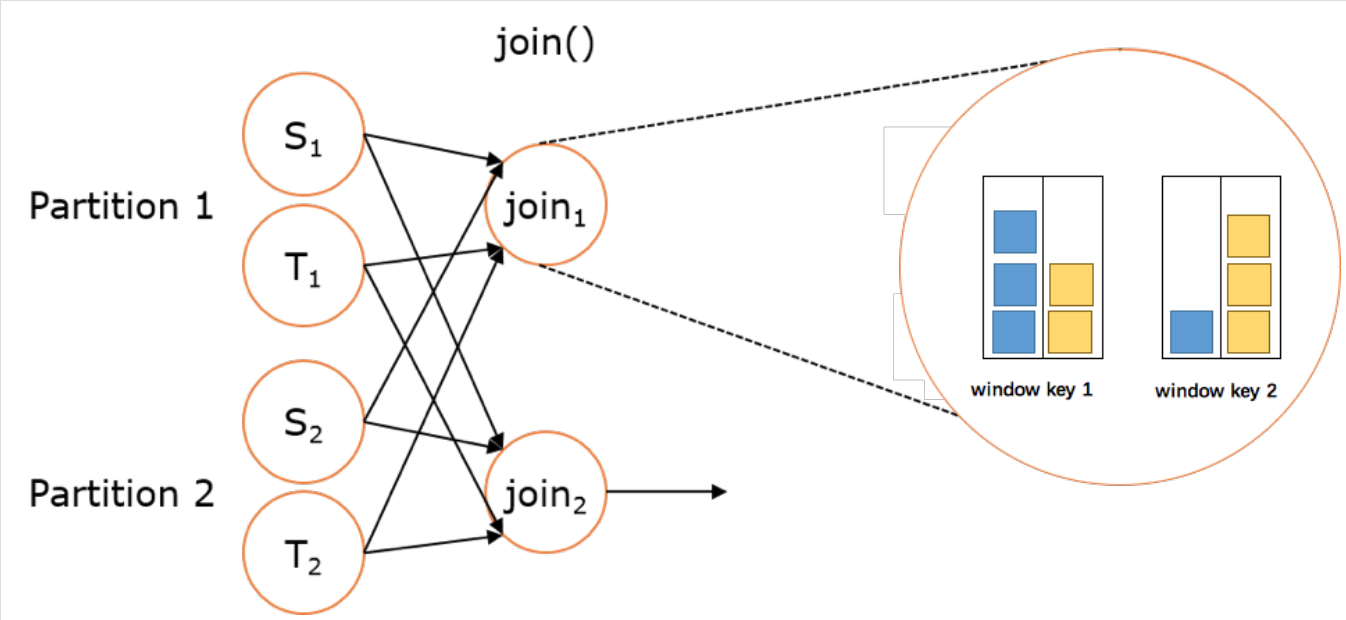
实际上这两者还是很点区别的。首先 co-group 侧重的是group，是对同一个key上的两组集合进行操作，而 join 侧重的是pair，是对同一个key上的每对元素进行操作。co-group 比 join 更通用一些，因为 join 只是 co-group 的一个特例，所以 join 是可以基于 co-group 来实现的（当然有优化的空间）。而在 co-group 之外又提供了 join 接口是因为用户更熟悉 join（源于数据库吧），而且能够跟 DataSet API 保持一致，降低用户的学习成本。

JoinedStreams 和 CoGroupedStreams 是基于 Window 上实现的，所以 CoGroupedStreams 最终又调用了 WindowedStream 来实现。

```
val firstInput: DataStream[MyType] = ...
val secondInput: DataStream[AnotherType] = ...
```

```
val result: DataStream[(MyType, AnotherType)] = firstInput.join(secondInput)
    .where("userId").equalTo("id")
    .window(TumblingEventTimeWindows.of(Time.seconds(3)))
    .apply (new JoinFunction () {...})
```

上述 JoinedStreams 的样例代码在运行时会转换成如下的执行图：



双流上的数据在同一个key的会被分别分配到同一个window窗口的左右两个篮子里，当window结束的时候，会对左右篮子进行笛卡尔积从而得到每一对pair，对每一对pair应用 JoinFunction。不过目前（Flink 1.1.x）JoinedStreams 只是简单地实现了流上的join操作而已，距离真正的生产使用还是有些距离。因为目前 join 窗口的双流数据都是被缓存在内存中的，也就是说如果某个key上的窗口数据太多就会导致 JVM OOM（然而数据倾斜是常态）。双流join的难点也正是在这里，这也是社区后面对 join 操作的优化方向，例如可以借鉴Flink在批处理join中的优化方案，也可以用ManagedMemory来管理窗口中的数据，并当数据超过阈值时能spill到硬盘。

ConnectedStreams

在 DataStream 上有一个 union 的转换 `dataStream.union(otherStream1, otherStream2, ...)`，用来合并多个流，新的流会包含所有流中的数据。union 有一个限制，就是所有合并的流的类型必须是一致的。`ConnectedStreams` 提供了和 union 类似的功能，用来连接**两个**流，但是与 union 转换有以下几个区别：

1. ConnectedStreams 只能连接两个流，而 union 可以连接多于两个流。
2. ConnectedStreams 连接的两个流类型可以不一致，而 union 连接的流的类型必须一致。
3. ConnectedStreams 会对两个流的数据应用不同的处理方法，并且双流之间可以共享状态。这在第一个流的输入会影响第二个流时, 会非常有用。

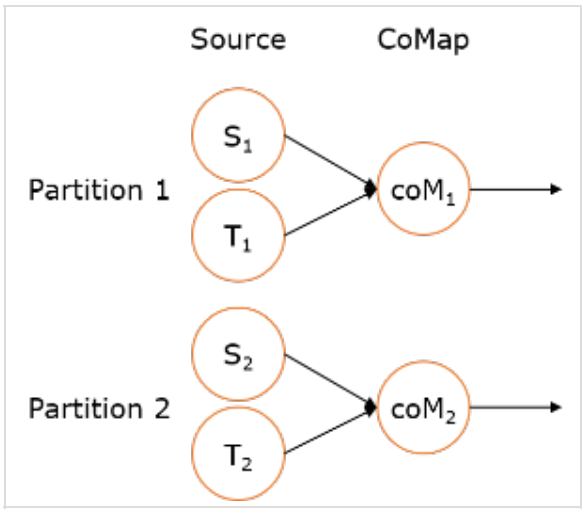
如下 ConnectedStreams 的样例，连接 `input` 和 `other` 流，并在`input`流上应用`map1`方法，在`other`上应用`map2`方法，双流可以共享状态（比如计数）。

```
val input: DataStream[MyType] = ...
val other: DataStream[AnotherType] = ...
```

```
val connected: ConnectedStreams[MyType, AnotherType] = input.connect(other)

val result: DataStream[ResultType] =
        connected.map(new CoMapFunction[MyType, AnotherType, ResultType]() {
            override def map1(value: MyType): ResultType = { ... }
            override def map2(value: AnotherType): ResultType = { ... }
        })
```

当并行度为2时，其执行图如下所示：



总结

本文介绍通过不同数据流类型的转换图来解释每一种数据流的含义、转换关系。后面的文章会深入讲解 Window 机制的实现，双流 Join 的实现等。