18 | Apache Samza：最简洁的开源流计算框架

今天，我们来看第三种开源流计算框架 Apache Samza。我们同样是从**系统架构、流的描述、流的处理、流的状态、消息处理可靠性**这五个方面来进行分析和讲解。

Apache Samza

Apache Samza (后简称 Samza)最初是由 LinkedIn 开源的一款分布式流计算框架，之后贡献给 Apache 并最终孵化成一个顶级项目。在众多的流计算框架中，Samza 都算得上是一个非常独特的分布式流计算框架。

这是因为，相比其他流计算框架的复杂实现，Samza 的设计和实现可以说是简单到了极致。Samza 与我们在模块二中实现的流计算框架有相似的设计观念，认为流计算就是从 Kafka 等消息中间件中取出消息，然后对消息进行处理，最后将处理结果重新发回消息中间件的过程。

Samza 将流数据的管理委托给 Kafka 等消息中间件，再将资源管理、任务调度和分布式执行等功能，借助于诸如 YARN 这样的分布式资源管理系统完成。其自身的主要逻辑，则是专注于对流计算过程的抽象，以及对用户编程接口的实现。因此，Samza 实现的流计算框架非常简洁，其早期版本的核心代码甚至不超过一万行。

上面就是 Samza 整体的设计思路了，下面我再针对各个部分进行详细讲解。

系统架构

我们先来看 Samza 的系统架构。以运行在 YARN 上的 Samza 为例，它就是一个典型的 YARN 应用。下面的图 1 是 Samza 的系统架构。

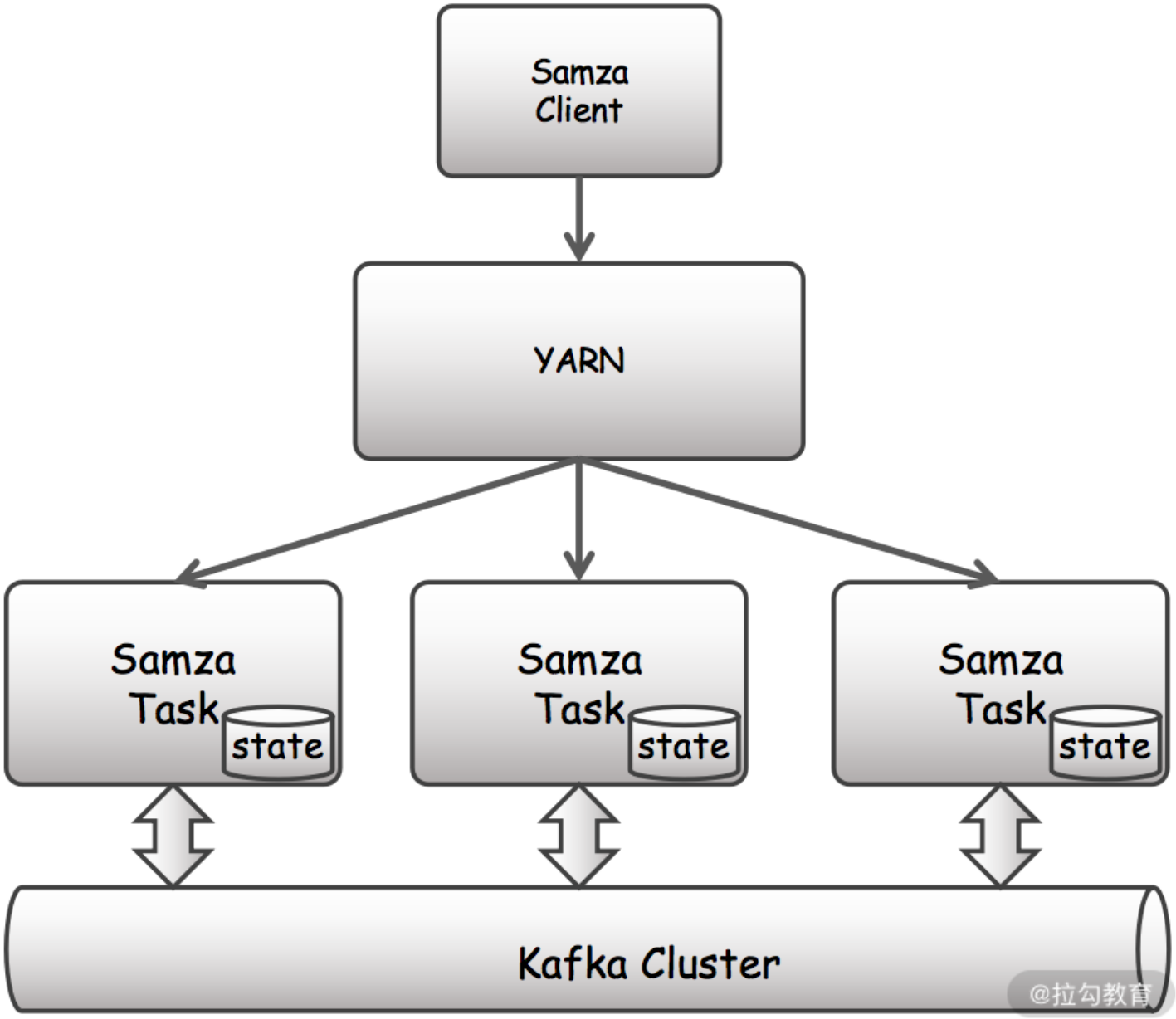

图 1 Samza 系统架构图

在上面的图 1 中，Samza 的 YARN 客户端向 YARN 提交 Samza 作业，并从 YARN 集群中申请资源（主要是 CPU 和内存）用于执行 Samza 应用中包含的作业（Job）。Samza 作业在运行时，表现为多个副本的任务。Samza 任务正是流计算应用的处理逻辑所在，它们从 Kafka 中读取消息，然后进行处理，并最终将处理结果重新发回 Kafka。可以看到，Samza 任务在计算过程中，还会在本地节点上使用到状态存储。

流的描述

接下来，我们再来看看在 Samza 中是如何描述一个流计算过程的。

首先是 Stream（流）。流是 Samza 处理的对象，它由一连串的消息组成。每个流都可以有任意多的消费者，从流中读取消息并不会删除这个消息。我们可以选择性地将消息与一个关键字关联，用于流的分区。Samza 使用插件系统实现了不同的流。比如，在 Kafka 中，流对应一个 topic 里的消息。在数据库中，流对应了一个表的更新操作。在 Hadoop 中，流则对应了目录下文件的追写和换行操作。

然后是 Job（作业）。一个 Samza 作业代表了一段对输入流进行转化并将结果写入输出流的程序。考虑到运行时的并行和水平扩展问题，Samza又将流和作业都进行了切分，将流切分为一个或多个分区，并相应地将作业切分成一个或多个任务。

比如下面的图 2 描述了一个实现 join 操作的作业。在该图中，输入流划分为了两个分区，对应地就启动了两个任务来执行 join 操作。

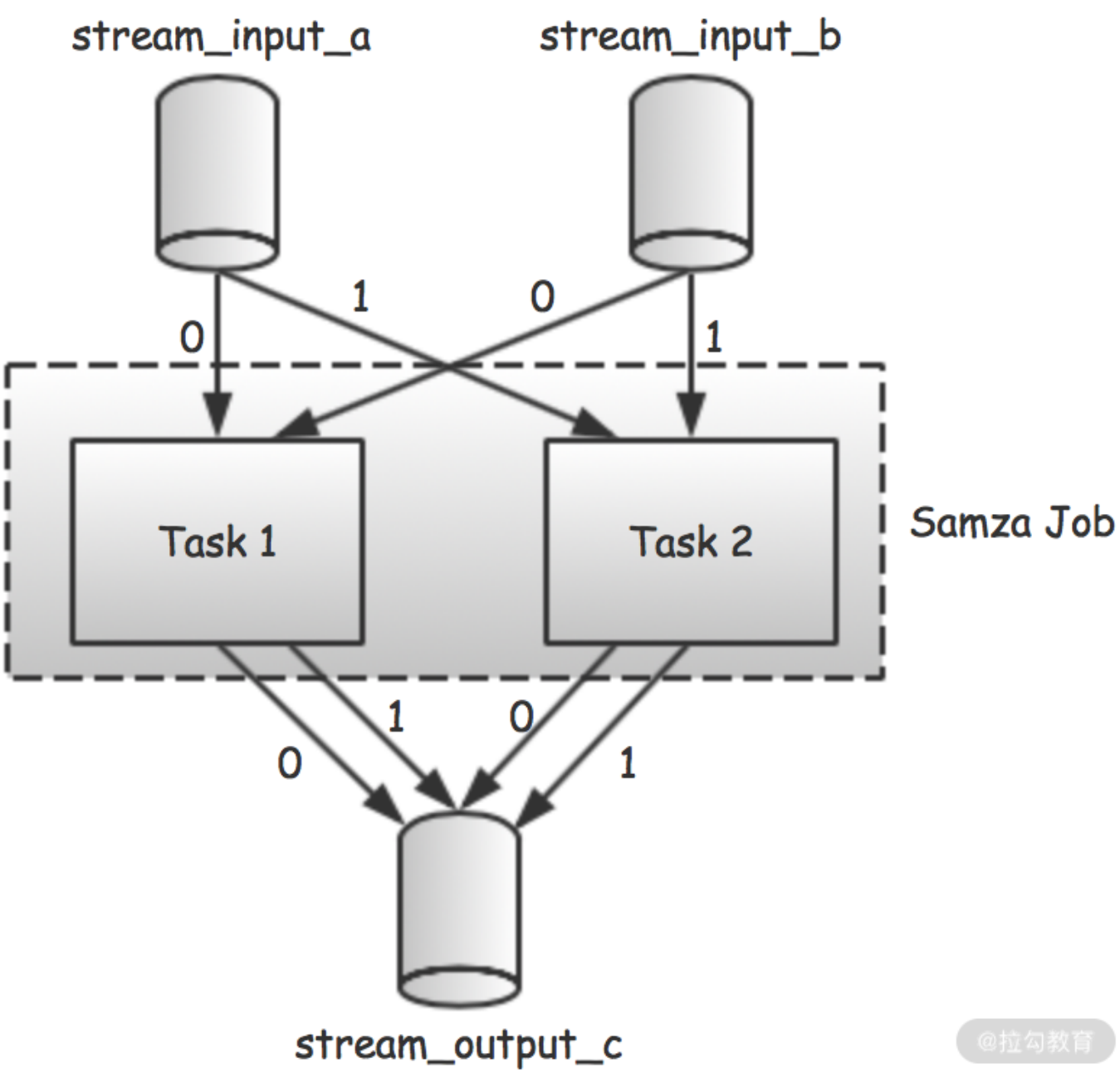

图 2 一个描述 join 操作的 Samza 作业

再然后是 Partition（分区）。Samza 的流和分区很明显是继承自 Kafka 的概念。当然 Sazma 也对这两个概念进行了抽象和泛化。Samza 的流被切分成了一个或多个分区，每个分区都是一个有序的消息序列。

接下来是 Task（任务）。Samza 作业又被切分为一个或多个任务。任务是作业并行化执行的单元，就像分区是流的并行化单元一样。每个任务负责处理流的一个或多个分区。通过 YARN 等资源调度器，任务被分布到 YARN 集群中的多个节点上运行，并且所有的任务彼此之间都是完全独立运行的。如果某个任务在运行时发生故障退出了，它会被 YARN 在另外的地方重启，并继续处理与之前相同的分区。

再接下来是Dataflow Graphs（数据流图）。将多个作业组合起来可以创建一个数据流图。数据流图描述了 Samza 流计算应用构成的整个系统的拓扑结构，它的边代表了数据流向，而节点代表了执行流转化操作的作业。

与 Storm 中 Topology 不同的是，数据流图中包含的各个作业并不要求一定是在同一个 Samza 应用程序中，数据流图可以是由多个不同的 Samza 应用程序共同构成，并且不同的 Samza 应用程序之间不会彼此相互影响。

在后面我们还会介绍 StreamApplication，需要注意 StreamApplication 和数据流图的不同之处。图 3 就展示了同样的数据流图，使用不同 StreamApplication 组合实现的例子。

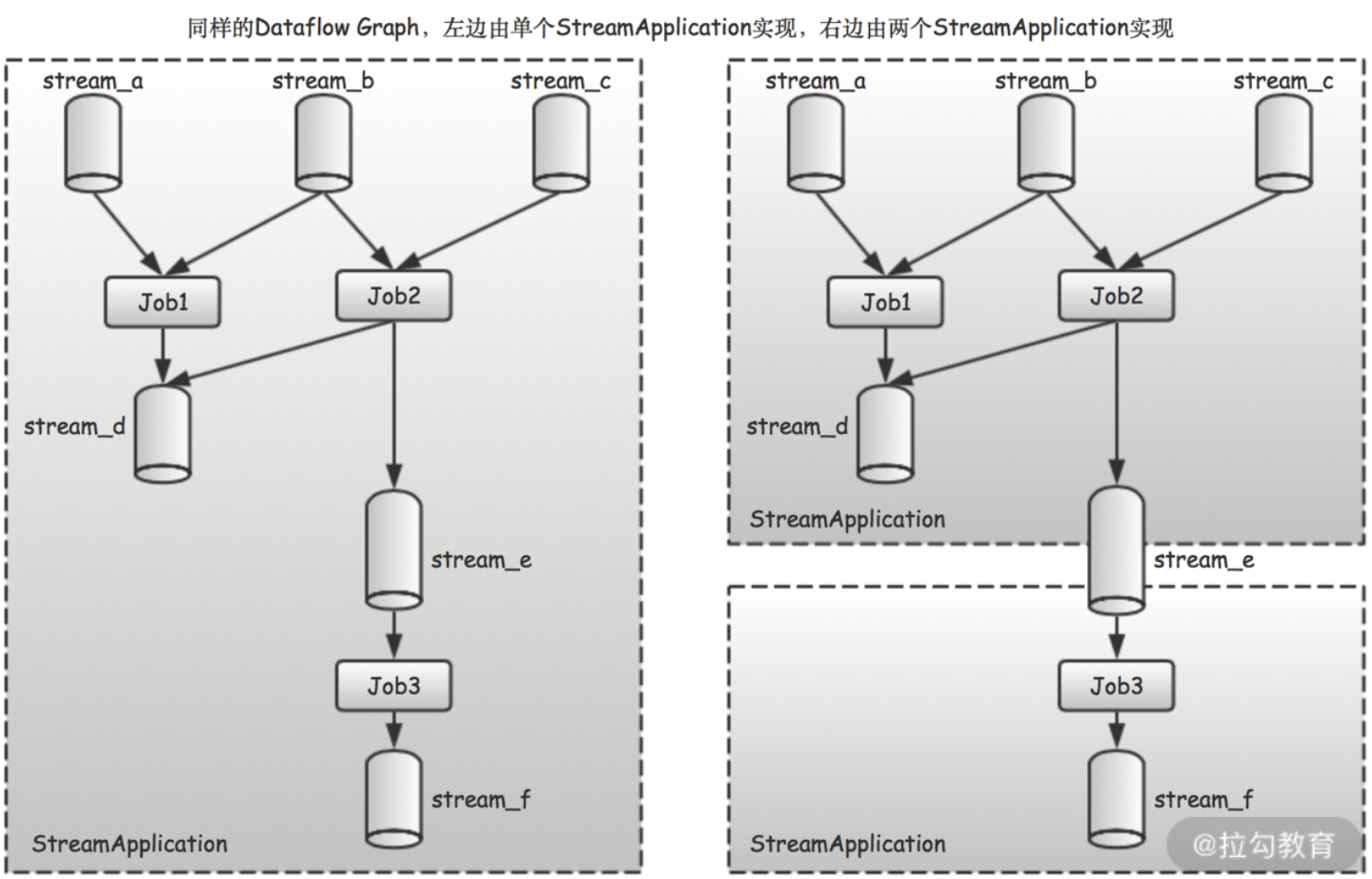

图 3 同样的数据流图可以使用多种方法实现

再接下来是 Containers（容器）。前面所讲的分区和任务都是逻辑上的并行单元，它们不是对计算资源的真实划分。那什么才是对计算资源的真实划分呢？这就是容器。容器是物理上的并行单元，每一个容器都代表着一定配额的计算资源。每个容器可以运行一个或多个任务。任务的数量由输入流的分区数确定，而容器的数量则可以由用户在运行时任意指定。

最后是 StreamApplication（流应用程序）。这是 Samza 上层 API 中用于描述 Samza 流计算应用的概念。一个 StreamApplication 对应着一个 Samza 应用程序。如果我们将整个流计算系统各个子系统的实现都放在一个 StreamApplication 里，那么这个 StreamApplication 实际上就是数据流图的实现。如果我们将整个流计算系统各个子系统的实现放在多个 StreamApplication 里，那么所有这些 StreamApplication 才共同构成完整的数据流图。换言之，StreamApplication 代表的是部分 DAG，而 Dataflow Graphs 代表的是完整 DAG。

以上介绍了 Samza 中的核心概念。这里还是需要强调下，Samza 中关于 job 和 task 的定义与 Hadoop MapReduce 框架中关于 job 和 task 的定义完全不同。

在 MapReduce 里，一个 MapReduce 程序就是一个 job，而一个 job 里面可以有一个或多个 task。这些 task 由 job 解析而来，它们又分为 map task 和 reduce task，执行着不同的任务。

但是在 Samza 里，task 相当于是 job 的多个运行时副本，所有 task 都执行着完全相同的程序逻辑，它们仅仅是输入输出的流分区不同而已。

所以从这种意义上来讲，Samza 的 job 和 task 之间的关系，就相当于程序和进程之间的关系。一个程序可以起多个进程，所有这些进程都执行着相同的程序代码。

流的处理

接下来我们再来看 Samza 中的流是怎么被处理的。与 Storm 的发展非常相似，Samza 用于构建流计算应用的编程接口，也经历了从底层 API 到上层 API 演进的过程，这其实本身也代表了流计算领域，和 Samza 框架自身的发展历史。如果你想更加清楚地理解框架背后的工作原理，可以去详细研究底层 API。但是上层 API 更加“现代”，而且更加有助于我们理解流计算这种编程模式。所以我们在本节直接使用 Samza 的上层 API 来讲解 Samza 流的执行。

我们同样从**流的输入、流的处理、流的输出和反向压力四个方面**来讨论 Samza 中流的执行过程。

首先是**流的输入**。Samza 使用各种描述符来定义 Samza 应用的各个组成部分。以 Kafka 为例，Samza 提供了 KafkaSystemDescriptor 用于描述管理数据流的 Kafka 集群。对每一个 Kafka 输入流，我们创建一个 KafkaInputDescriptor 用于描述该输入的信息，然后通过 Samza 流应用描述符 StreamApplicationDescriptor 的 getInputStream 方法，创建出消息流 MessageStream。

下面就是一段 Samza 流数据输入的示例代码（本课时完整代码请参考这里）。

```
// 根据 Kafka 集群配置，创建 KafkaSystemDescriptor
KafkaSystemDescriptor kafkaSystemDescriptor = new KafkaSystemDescriptor(KAFKA_SYSTEM_NAME)
    .withConsumerZkConnect(KAFKA_CONSUMER_ZK_CONNECT)
    .withProducerBootstrapServers(KAFKA_PRODUCER_BOOTSTRAP_SERVERS)
    .withDefaultStreamConfigs(KAFKA_DEFAULT_STREAM_CONFIGS);
// 对每一个输入流/输出流，创建一个KafkaInput/Output descriptor
KafkaInputDescriptor<KV<String, String>> inputDescriptor =
    kafkaSystemDescriptor.getInputDescriptor(INPUT_STREAM_ID,
        KVSerde.of(new StringSerde(), new StringSerde()));
// 获取一个MessageStream，可以在其上链式添加各种操作
MessageStream<KV<String, String>> lines = streamApplicationDescriptor.getInputStream(inputDescrip
```

在上面的代码中，我们创建了一个从 Kafka 读取消息的输入流 lines。Samza 用键值对表示消息，其中键代表的是消息的主键，通常带有业务含义，比如用户 id、事件类型、产品编号等。而值则代表的是消息的具体内容。在很多场景下，带有业务含义的键都非常有用，比如实现类似于 Flink 中 KeyedStream 的功能。

然后是**流的处理**。Samza 对流的处理，是通过建立在 MessageStream 上的各种算子（Operator）完成。MessageStream 上定义的算子，主要包括两类，即流数据处理类算子和流数据管理类算子。流数据处理类算包括 map、flatMap、asyncFlatMap、filter 等。流数据管理类算子则包括 partitionBy、merge、broadcast、join 和 window 等。

下面就是对 MessageStream 进行处理的示例代码。

```
lines
   .map(kv -> kv.value)
   .flatMap(s -> Arrays.asList(s.split("\\W+")))
   .window(Windows.keyedSessionWindow(
       w -> w, Duration.ofSeconds(5), () -> 0, (m, prevCount) -> prevCount + 1,
       new StringSerde(), new IntegerSerde()), "count")
   .map(windowPane ->
       KV.of(windowPane.getKey().getKey(),
           windowPane.getKey().getKey() + ": " + windowPane.getMessage().toString()))
   .sendTo(counts);
```

在上面的代码中，先将 Kafka 中读出的键值对消息流 lines 转化为由其值组成的消息流，再用 flatMap 将每行的文本字符串转化为单词流。然后用 Windows.keyedSessionWindow 定义了一个以 5 秒钟为窗口进行聚合的窗口操作，这样原来的单词流会转化为以 <word, count> 为键值对的数据流。之后，再用 map 将数据流转化为其输出的格式，并最终发送到 Kafka。至此就完成了单词计数的功能。

接下来是**流的输出**。与输入流对应，Samza 提供了 KafkaOutputDescriptor 用于描述将消息发送到 Kafka 的输出流。通过 Samza 流应用描述符 StreamApplicationDescriptor 的 getOutputStream 方法，就可以创建消息输出流 OutputStream。下面是示例代码。

```
KafkaOutputDescriptor<KV<String, String>> outputDescriptor =
    kafkaSystemDescriptor.getOutputDescriptor(OUTPUT_STREAM_ID,
        KVSerde.of(new StringSerde(), new StringSerde()));
OutputStream<KV<String, String>> counts = streamApplicationDescriptor.getOutputStream(outputDescr
```

上面的代码中，我们定义了将消息发送到 Kafka 的输出流 counts。

继而，将以上输入、处理和输出三个部分的代码片段整合起来，就能实现一个具备单词计数功能的 Samza 流计算应用了。

最后是**反向压力**。Samza 不支持反向压力，但它却有另外的方法避免 OOM，这就是 Kafka 的消息缓冲功能！由于 Samza 是直接借用 Kafka 来保存处理过程中的流数据，所以即便没有反向压力，Samza 也不会存在内存不足的问题。但我们明白，就算躲得过初一，也躲不过十五，再大的磁盘时间长了也会被不断积压的消息占满。

所以在使用 Samza 时，我们还是需要对 Kafka 中消息的消费情况和积压情况进行监控。当发现消息积压时，我们应立即采取措施来处理消息积压的问题。比如，可以给下游任务分配更多的计算资源。

流的状态

接下来，我们再来看 Samza 中流的状态问题。Samza 支持无状态和有状态的处理。无状态的处理是指在处理过程中，不涉及任何状态处理相关的操作，比如 map、filter 等操作。而有状态的处理则是消息处理过程中，需要保存一些与消息有关的状态，比如计算网站每五分钟的 UV（Unique Visitor）等。Samza 提供了错误容忍的、可扩展的状态存储。

在**流数据状态**方面，由于 Samza 使用了 Kafka 来管理其处理各个环节的数据流，所以可以说 Samza 的大部分流数据状态，都直接保存在了 Kafka 中。Kafka 帮助 Samza 完成了对消息的可靠性存储、对流的分区、对消息顺序的保证等功能。

除了保存在 Kafka 中的消息外，在 Samza 的任务节点进行诸如 window、join 等操作时，也会依赖于在缓冲区中，暂存一段时间窗口内的消息，所以我们将这类 API 也归于流数据状态管理。

Samza 的 MessageStream 类中，与流数据状态管理相关的 API 包括 window、join、partitionBy。其中 partitionBy 又比较特殊，它不像 window、join 那样主要使用内存或诸如 RocksDB 的本地数据库来存储状态，而是将消息流按照主键重新分区后，输出到以 Kafka 为载体的中间流。另外，Samza 还包括了将两个流合并的 merge 操作（其实是类似于 SQL 中的 UNION ALL 操作），这种对流的合并操作，实现起来相对简单，并不涉及状态操作。

在**流信息状态**方面，Samza 的流信息状态，可以通过对任务状态（task state）的管理完成。虽然 Samza 并不阻止我们，在 Samza 任务中使用远程数据库来进行状态管理，但它还是极力推荐我们使用本地数据库的方式存储状态。这样做是出于对性能优化、资源隔离、故障恢复和失败重处理等多方面因素的考虑。

Samza 提供了 KeyValueStore 接口用于状态的存储。在 KeyValueStore 接口背后，Samza 实现了基于 RocksDB 的本地状态存储。当 Samza 进行状态操作时，所有的操作都是直接访问本地 RocksDB 数据库，所以性能比跨网络远程访问的数据库高出很多，有时甚至能达到 2 到 3 个数量级的区别。

另外，为了保障任务在其他节点上重启时，访问的是相同状态数据，还会将每次写入 RocksDB 的操作，复制一份到 Kafka 作为变更日志。这样，当任务在其他节点上重启时，能够从 Kafka 中读取并重放变更日志，恢复出任务转移物理节点前 RocksDB 中的数据。

消息处理可靠性

最后，我们来看下 Samza 中消息处理可靠性的问题。

Samza 目前只提供了 at-least-once 级别的消息处理可靠性保证，但是有计划支持 exactly-once 级别的消息处理可靠性保证。所以到目前为止，如果我们需要实现消息的不重复处理，就应该尽量让状态的更新是幂等操作。

在缺乏像 Storm 消息追踪机制，或像 Spark 和 Flink 中用到的快照机制的情况下，Samza 要达到诸如计数、求和不能重复的要求还是比较困难的。而像计数、求和这样的聚合计算，在流计算系统中还是比较常见的需求。所以，目前 Samza 还不是非常适合于这种对计算准确度，要求非常严格的场景。

我认为，Samza 在以后实现 exactly-once 级别消息处理可靠性保证时，也会采取类似于 Flink 使用的方案，即实现状态的 checkpoint 机制，在此基础上实现分布式快照管理，最终就可以实现 exactly-once 级别的消息处理可靠性保证了。

小结

总的来说，相较其他开源流计算框架而言，Samza 的设计思路和具体实现都是相对简洁的，它反映出了最朴实的流计算系统构建思路。也就是，流计算无非是一个从消息中间件中，取出消息进行处理，然后再将处理结果写回消息中间件的过程。

不过，由于 Samza 缺少了 exactly-once 级别的消息处理可靠性支持，也缺少了 OOM 机制，并且还需要监控 Kafka 等消息中间件的积压情况，所以有时候使用起来并不是非常方便。这点需要你根据具体的业务场景，判断使用 Samza 是否合适。如果 Samza 不合适的话，你可以考虑我在下一个课时将会讲解的 Flink。

最后留一个小作业，今天的课程中我们提到 Samza “为了保障任务在其他节点上重启时，访问的是相同状态数据，还会将每次写入 RocksDB 的操作，复制一份到 Kafka 作为变更日志。这样，当任务在其他节点上重启时，能够从 Kafka 中读取并重放变更日志，恢复出任务转移物理节点前 RocksDB 中的数据”。

对于这种实现分布式状态存储的方案，你是否觉得存在什么问题呢？如果是你，你会选择怎样的方法来进行改进呢？是否可以用某种方案，来同时解决“分布式状态存储”和 “exactly-once 级别消息处理可靠性保证”这两个问题呢？

可以将你的想法或问题写在留言区。

下面是本课时的脑图，以帮助你理解。