分布式流处理新贵Kafka Stream

2017-08-18 Spark技术日报

> *本文转发自技术世界，原文链接*
> *http://www.jasongj.com/kafka/kafka_stream/*

本文介绍了Kafka Stream的背景，如Kafka Stream是什么，什么是流式计算，以及为什么要有Kafka Stream。接着介绍了Kafka Stream的整体架构，并行模型，状态存储，以及主要的两种数据集KStream和KTable。并且分析了Kafka Stream如何解决流式系统中的关键问题，如时间定义，窗口操作，Join操作，聚合操作，以及如何处理乱序和提供容错能力。最后结合示例讲解了如何使用Kafka Stream。

Kafka Stream背景

Kafka Stream是什么

Kafka Stream是Apache Kafka从0.10版本引入的一个新Feature。它是提供了对存储于Kafka内的数据进行流式处理和分析的功能。

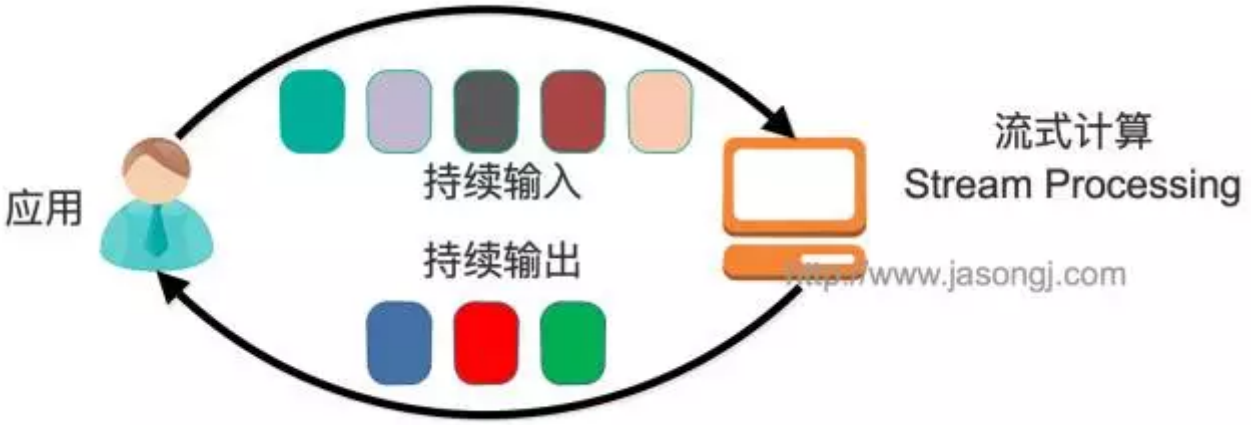
Kafka Stream的特点如下：

- Kafka Stream提供了一个非常简单而轻量的Library，它可以非常方便地嵌入任意Java应用中，也可以任意方式打包和部署
- 除了Kafka外，无任何外部依赖
- 充分利用Kafka分区机制实现水平扩展和顺序性保证
- 通过可容错的state store实现高效的状态操作（如windowed join和aggregation）

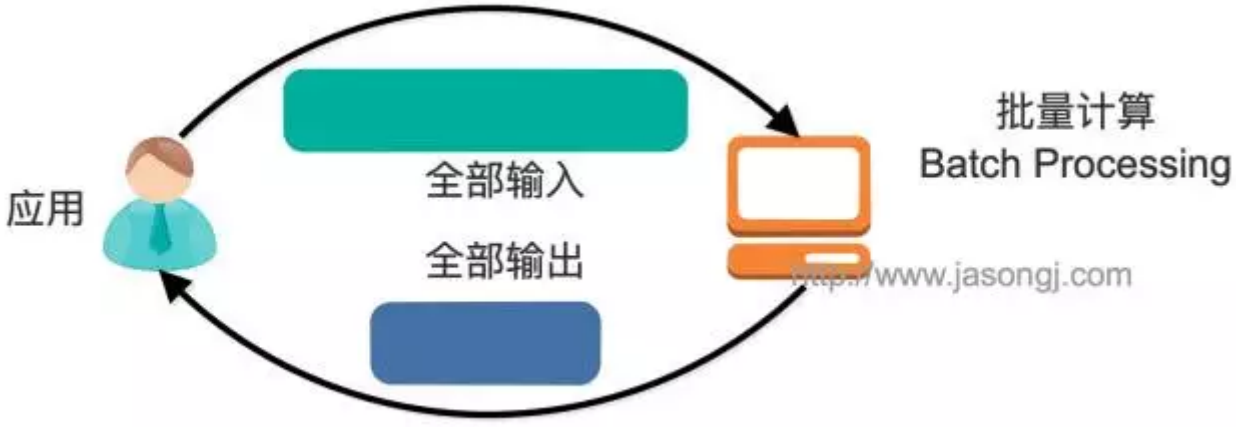
- 支持正好一次处理语义
- 提供记录级的处理能力，从而实现毫秒级的低延迟
- 支持基于事件时间的窗口操作，并且可处理晚到的数据（late arrival of records）
- 同时提供底层的处理原语Processor（类似于Storm的spout和bolt），以及高层抽象的DSL（类似于Spark的map/group/reduce）

什么是流式计算

一般流式计算会与批量计算相比较。在流式计算模型中，输入是持续的，可以认为在时间上是无界的，也就意味着，永远拿不到全量数据去做计算。同时，计算结果是持续输出的，也即计算结果在时间上也是无界的。流式计算一般对实时性要求较高，同时一般是先定义目标计算，然后数据到来之后将计算逻辑应用于数据。同时为了提高计算效率，往往尽可能采用增量计算代替全量计算。



批量处理模型中，一般先有全量数据集，然后定义计算逻辑，并将计算应用于全量数据。特点是全量计算，并且计算结果一次性全量输出。

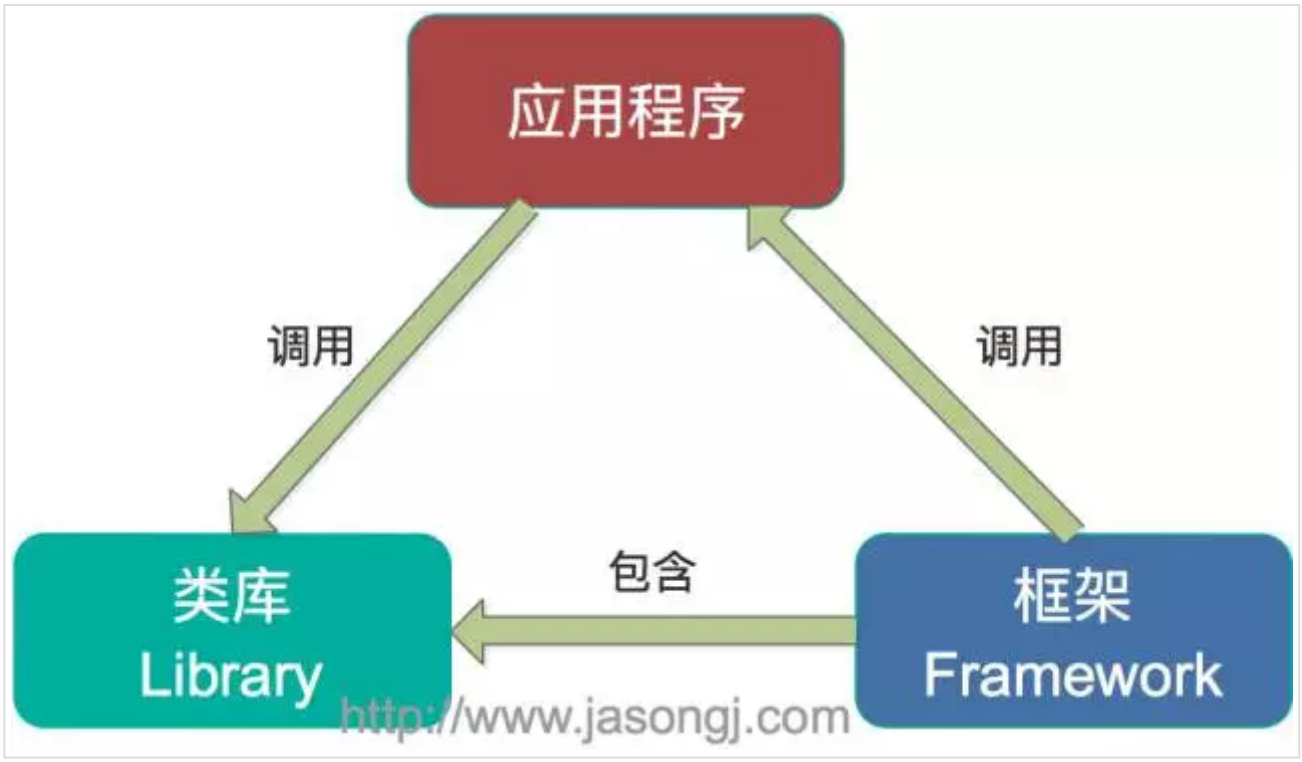


为什么要有Kafka Stream

当前已经有非常多的流式处理系统，最知名且应用最多的开源流式处理系统有Spark Streaming和Apache Storm。Apache Storm发展多年，应用广泛，提供记录级别的处理能力，当前也支持SQL on Stream。而Spark Streaming基于Apache Spark，可以非常方便与图计算，SQL处理等集成，功能强大，对于熟悉其它Spark应用开发的用户而言使用门槛低。另外，目前主流的Hadoop发行版，如MapR，Cloudera和Hortonworks，都集成了Apache Storm和Apache Spark，使得部署更容易。

既然Apache Spark与Apache Storm拥用如此多的优势，那为何还需要Kafka Stream呢？笔者认为主要有如下原因。

第一，Spark和Storm都是流式处理**框架**，而Kafka Stream提供的是一个基于Kafka的流式处理**类库**。框架要求开发者按照特定的方式去开发逻辑部分，供框架调用。开发者很难了解框架的具体运行方式，从而使得调试成本高，并且使用受限。而Kafka Stream作为流式处理**类库**，直接提供具体的类给开发者调用，整个应用的运行方式主要由开发者控制，方便使用和调试。



第二，虽然Cloudera与Hortonworks方便了Storm和Spark的部署，但是这些框架的部署仍然相对复杂。而Kafka Stream作为类库，可以非常方便的嵌入应用程序中，它对应用的打包和部署基本没有任何要求。更为重要的是，Kafka Stream充分利用了Kafka的分区机制和Consumer的Rebalance机制，使得Kafka Stream可以非常方便的水平扩展，并且各个实例可以使用不同的部署方式。具体来说，每个运行Kafka Stream的应用程序实例都包含了Kafka Consumer实例，多个同一应用的实例之间并行处理数据集。而不同实例之间的部署方式并不要求一致，比如部分实例可以运行在Web容器中，部分实例可运行在Docker或Kubernetes中。

第三，就流式处理系统而言，基本都支持Kafka作为数据源。例如Storm具有专门的kafka-spout，而Spark也提供专门的spark-streaming-kafka模块。事实上，Kafka基本上是主流的流式处理系统的标准数据源。换言之，

大部分流式系统中都已部署了Kafka，此时使用Kafka Stream的成本非常低。

第四，使用Storm或Spark Streaming时，需要为框架本身的进程预留资源，如Storm的supervisor和Spark on YARN的node manager。即使对于应用实例而言，框架本身也会占用部分资源，如Spark Streaming需要为shuffle和storage预留内存。

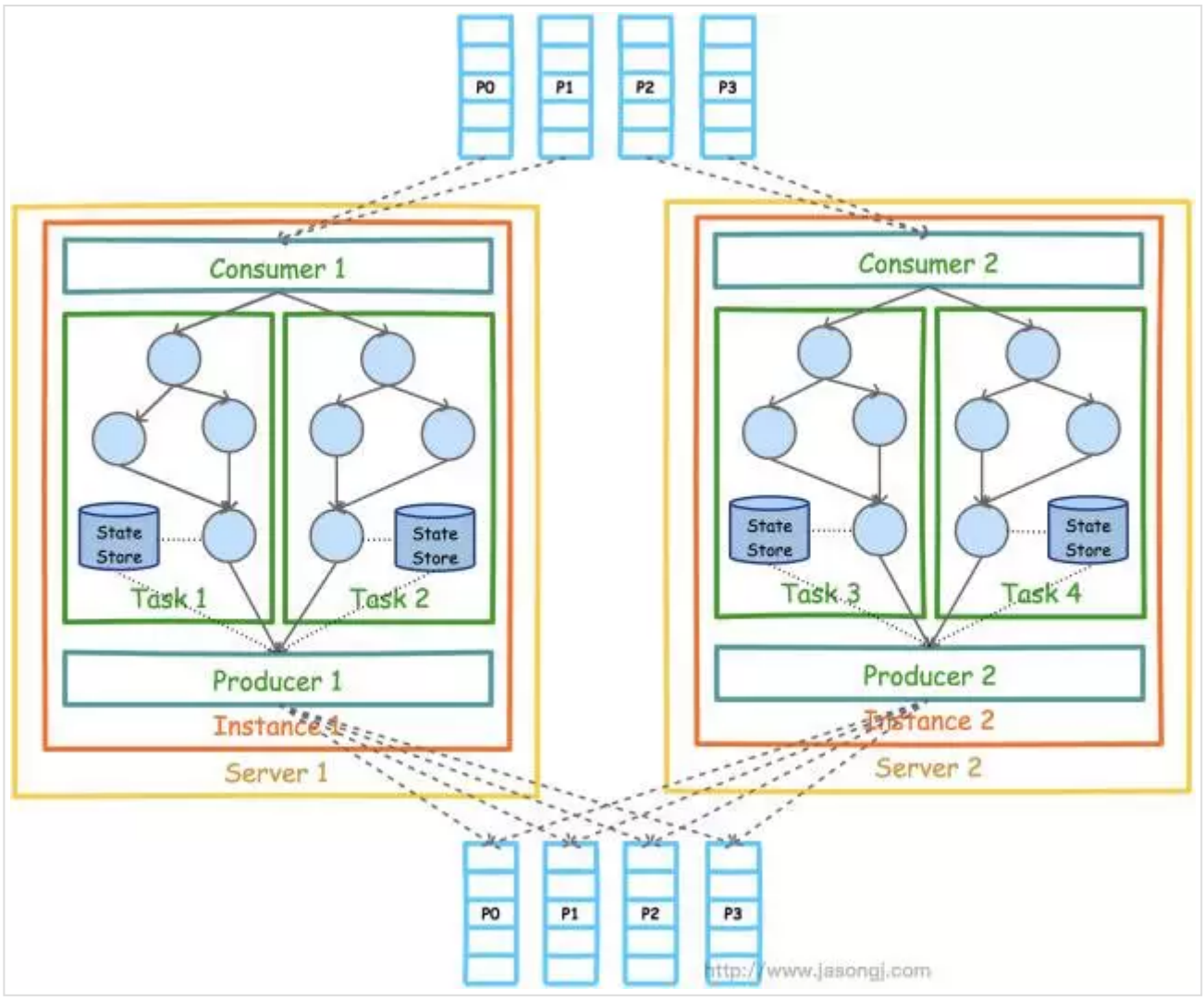
第五，由于Kafka本身提供数据持久化，因此Kafka Stream提供滚动部署和滚动升级以及重新计算的能力。

第六，由于Kafka Consumer Rebalance机制，Kafka Stream可以在线动态调整并行度。

Kafka Stream架构

Kafka Stream整体架构

Kafka Stream的整体架构图如下所示。



目前（Kafka 0.11.0.0）Kafka Stream的数据源只能如上图所示是Kafka。但是处理结果并不一定要如上图所示输出到Kafka。实际上KStream和Ktable的实例化都需要指定Topic。

```
KStream<String, String> stream = builder.stream("words-stream");

KTable<String, String> table = builder.table("words-table", "words-store");
```

另外，上图中的Consumer和Producer并不需要开发者在应用中显示实例化，而是由Kafka Stream根据参数隐式实例化和管理，从而降低了使用门槛。开发者只需要专注于开发核心业务逻辑，也即上图中Task内的部分。

Processor Topology

基于Kafka Stream的流式应用的业务逻辑全部通过一个被称为Processor Topology的地方执行。它与Storm的Topology和Spark的DAG类似，都定义了数据在各个处理单元（在Kafka Stream中被称作Processor）间的流动方式，或者说定义了数据的处理逻辑。

下面是一个Processor的示例，它实现了Word Count功能，并且每秒输出一次结果。

```java
public class WordCountProcessor implements Processor<String, String> {

  private ProcessorContext context;
  private KeyValueStore<String, Integer> kvStore;

  @SuppressWarnings("unchecked")
  @Override
  public void init(ProcessorContext context) {
    this.context = context;
    this.context.schedule(1000);
    this.kvStore = (KeyValueStore<String, Integer>) context.getStateStore("Counts");
  }

  @Override
  public void process(String key, String value) {
    Stream.of(value.toLowerCase().split(" ")).forEach((String word) -> {
      Optional<Integer> counts = Optional.ofNullable(kvStore.get(word));
      int count = counts.map(wordcount -> wordcount + 1).orElse(1);
      kvStore.put(word, count);
    });
  }

  @Override
  public void punctuate(long timestamp) {
    KeyValueIterator<String, Integer> iterator = this.kvStore.all();
    iterator.forEachRemaining(entry -> {
      context.forward(entry.key, entry.value);
      this.kvStore.delete(entry.key);
    });
    context.commit();
  }

  @Override
  public void close() {
    this.kvStore.close();
  }

}
```

从上述代码中可见

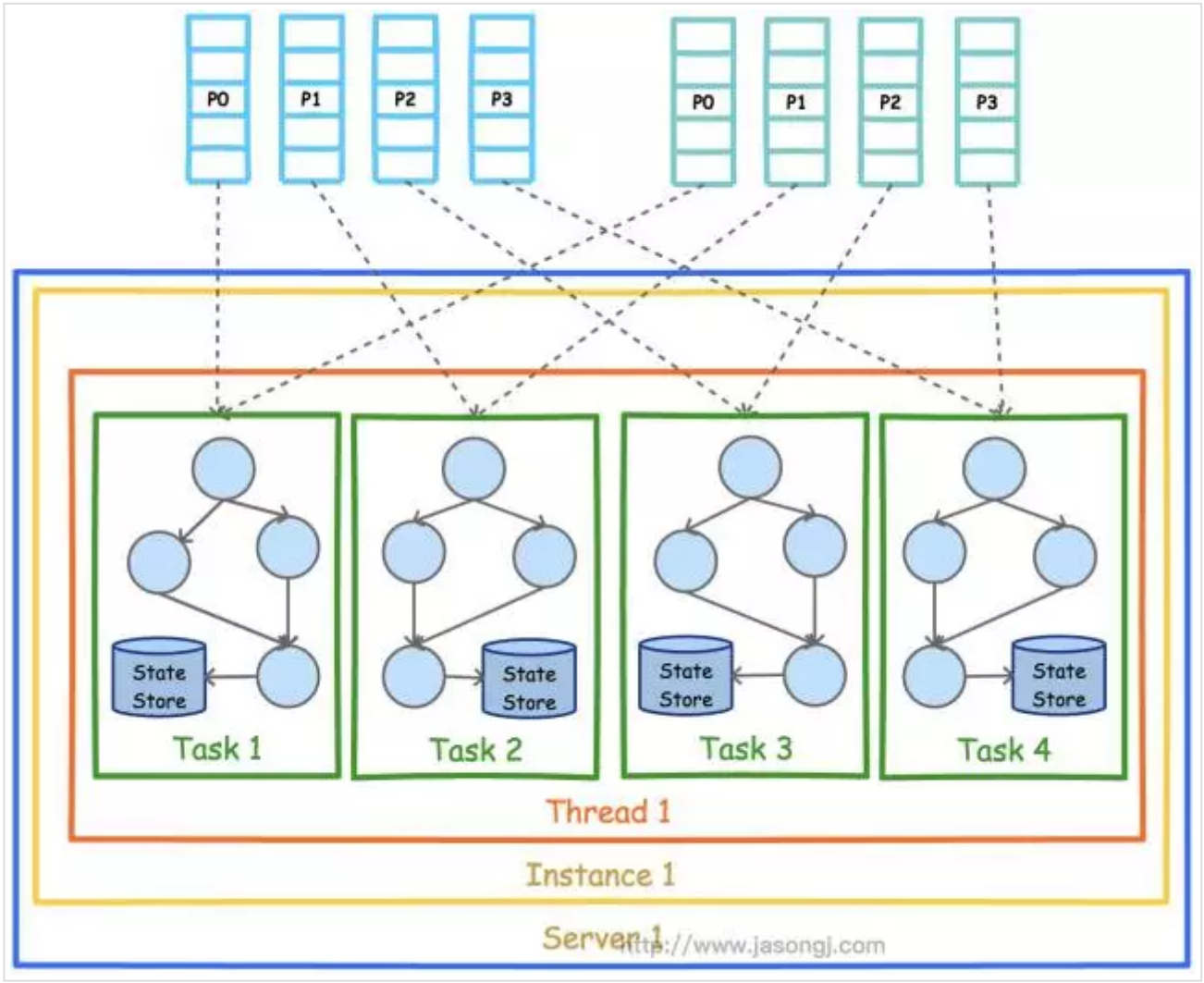
- process定义了对每条记录的处理逻辑，也印证了Kafka可具有记录级的数据处理能力。
- context.scheduler定义了punctuate被执行的周期，从而提供了实现窗口操作的能力。
- context.getStateStore提供的状态存储为有状态计算（如窗口，聚合）提供了可能。

Kafka Stream并行模型

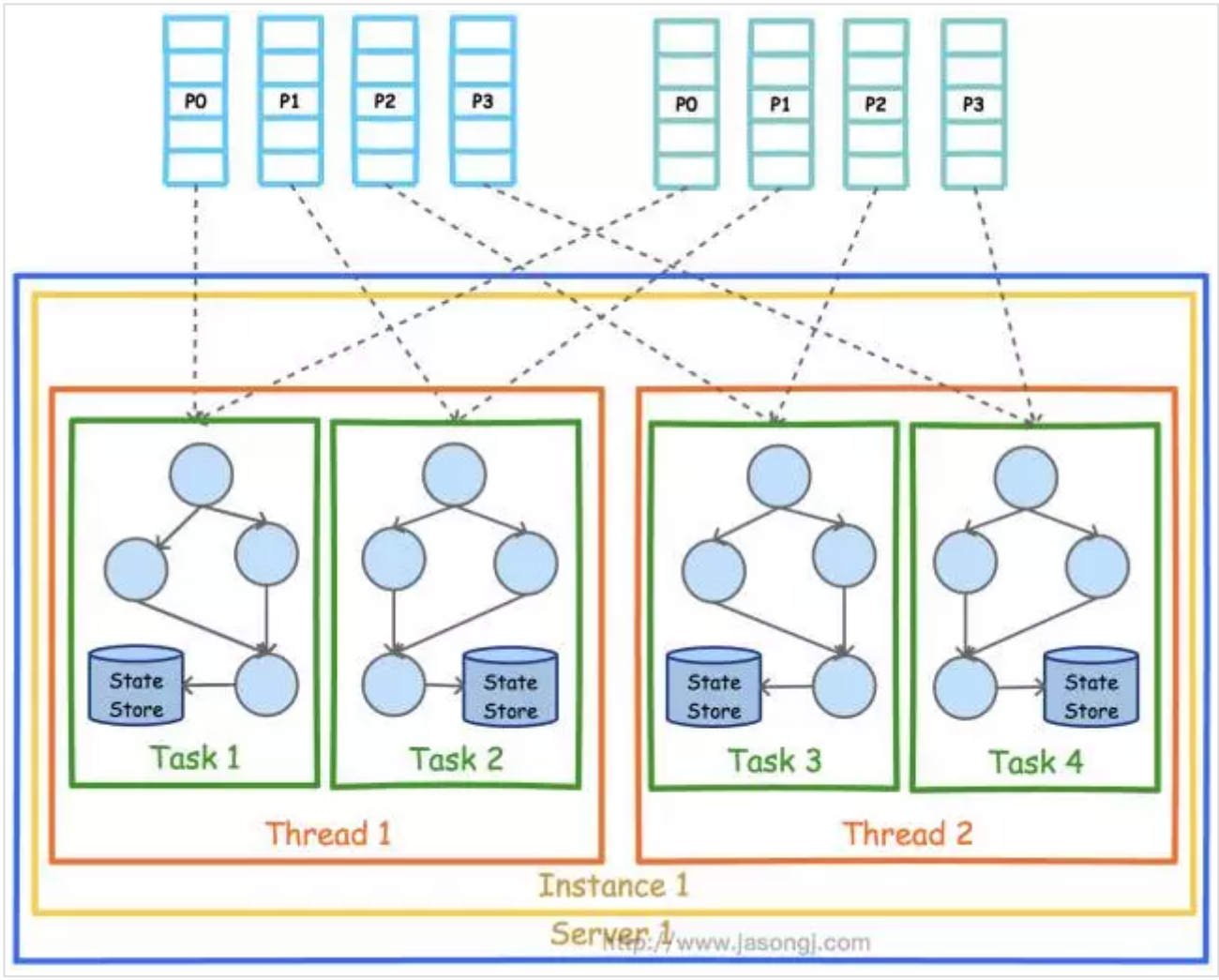
Kafka Stream的并行模型中，最小粒度为Task，而每个Task包含一个特定子Topology的所有Processor。因此每个Task所执行的代码完全一样，唯一的不同在于所处理的数据集互补。这一点跟Storm的Topology完全不一样。Storm的Topology的每一个Task只包含一个Spout或Bolt的实例。因此Storm的一个Topology内的不同Task之间需要通过网络通信传递数据，而Kafka Stream的Task包含了完整的子Topology，所以Task之间不需要传递数据，也就不需要网络通信。这一点降低了系统复杂度，也提高了处理效率。

如果某个Stream的输入Topic有多个(比如2个Topic，1个Partition数为4，另一个Partition数为3)，则总的Task数等于Partition数最多的那个Topic的Partition数（max(4,3)=4）。这是因为Kafka Stream使用了Consumer的Rebalance机制，每个Partition对应一个Task。

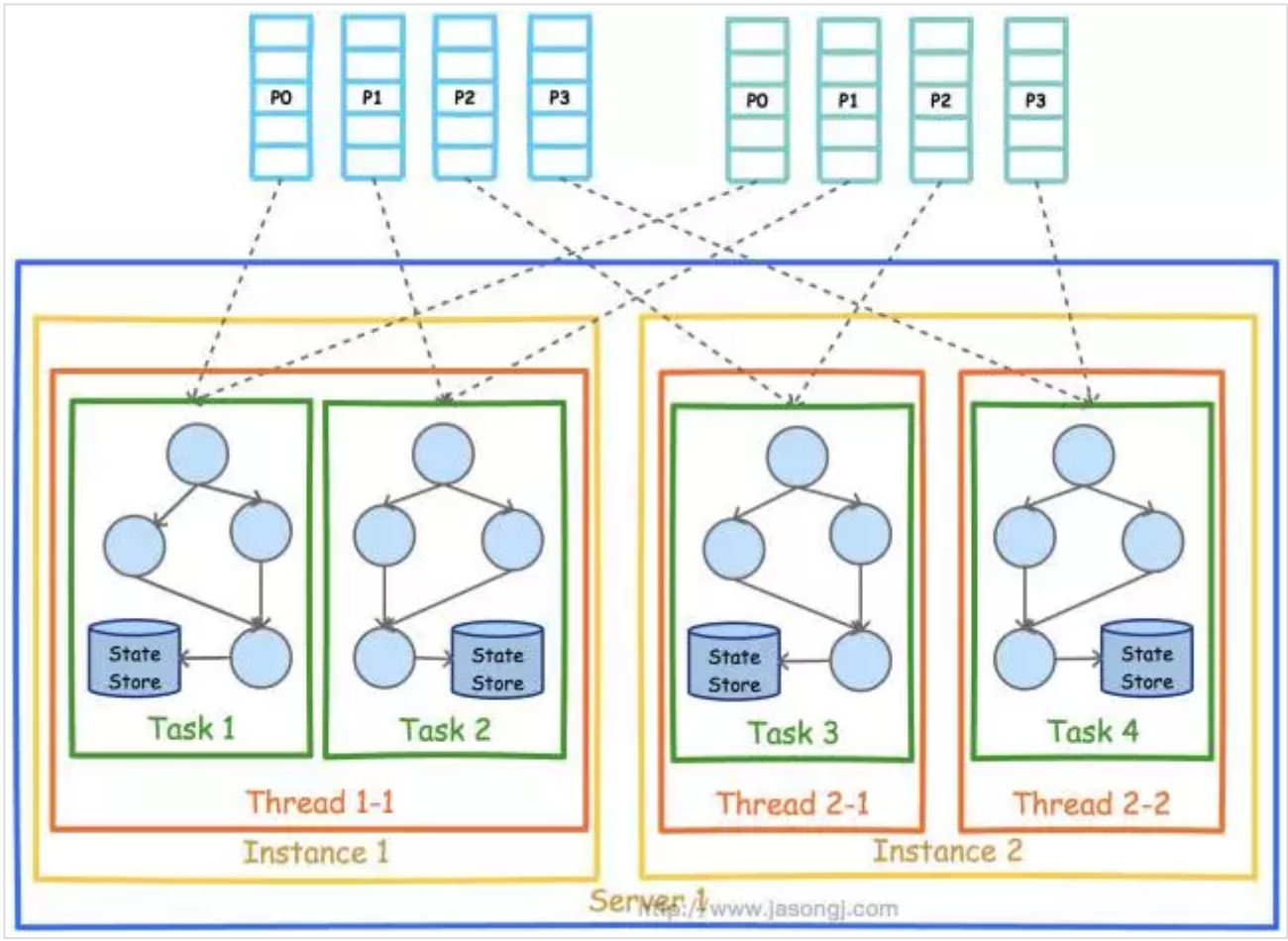
下图展示了在一个进程（Instance）中以2个Topic（Partition数均为4）为数据源的Kafka Stream应用的并行模型。从图中可以看到，由于Kafka Stream应用的默认线程数为1，所以4个Task全部在一个线程中运行。



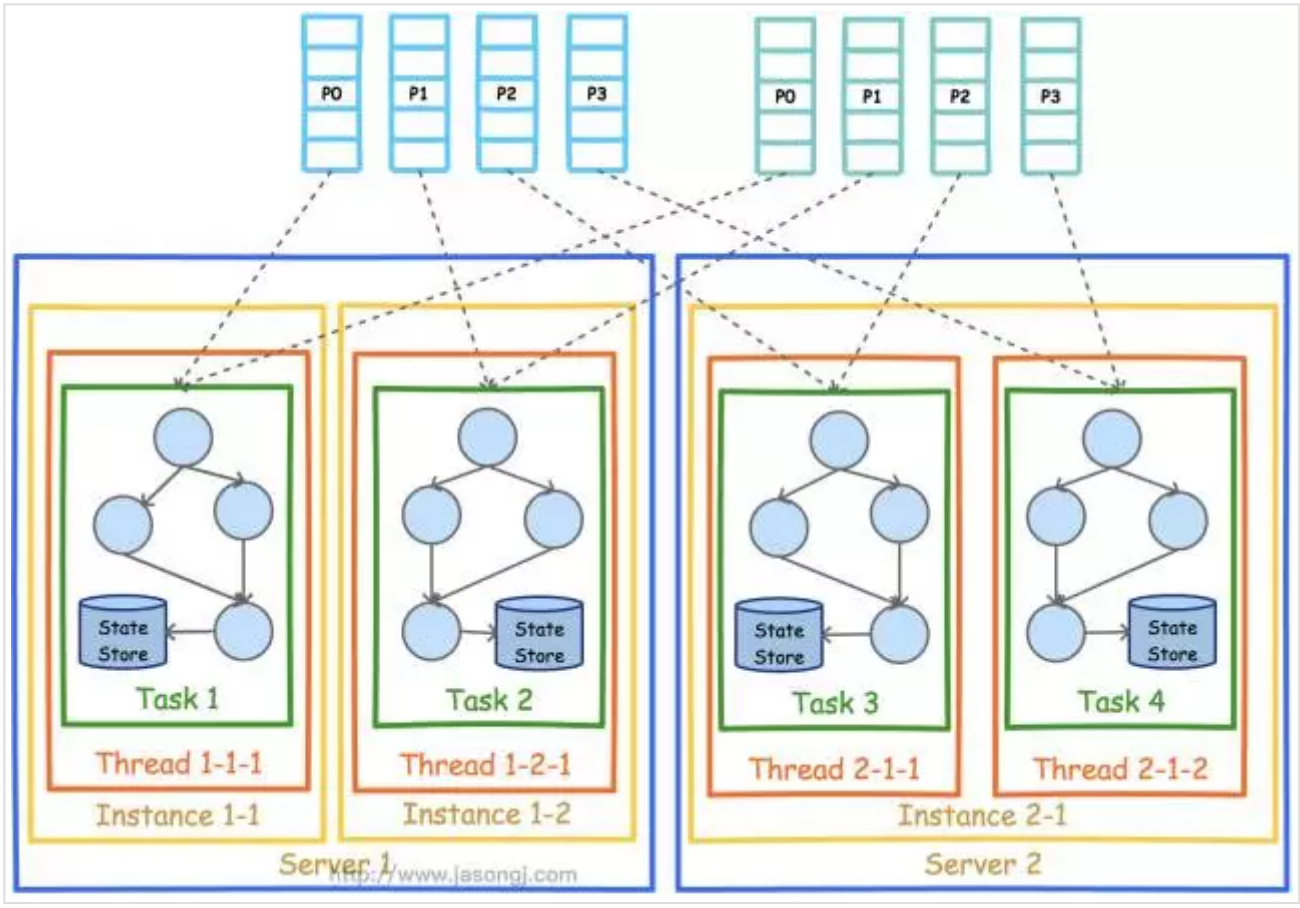
为了充分利用多线程的优势，可以设置Kafka Stream的线程数。下图展示了线程数为2时的并行模型。



前文有提到，Kafka Stream可被嵌入任意Java应用（理论上基于JVM的应用都可以）中，下图展示了在同一台机器的不同进程中同时启动同一Kafka Stream应用时的并行模型。注意，这里要保证两个进程的StreamsConfig.APPLICATION_ID_CONFIG完全一样。因为Kafka Stream将APPLICATION_ID_CONFI作为隐式启动的Consumer的Group ID。只有保证APPLICATION_ID_CONFI相同，才能保证这两个进程的Consumer属于同一个Group，从而可以通过Consumer Rebalance机制拿到互补的数据集。



既然实现了多进程部署，可以以同样的方式实现多机器部署。该部署方式也要求所有进程的APPLICATION_ID_CONFIG完全一样。从图上也可以看到，每个实例中的线程数并不要求一样。但是无论如何部署，Task总数总会保证一致。



注意：Kafka Stream的并行模型，非常依赖于《Kafka设计解析（一）- Kafka背景及架构介绍》一文中介绍的Kafka分区机制和《Kafka设计解析（四）- Kafka Consumer设计解析》中介绍的Consumer的Rebalance机制。强烈建议不太熟悉这两种机制的朋友，先行阅读这两篇文章。

这里对比一下Kafka Stream的Processor Topology与Storm的Topology。

- Storm的Topology由Spout和Bolt组成，Spout提供数据源，而Bolt提供计算和数据导出。Kafka Stream的Processor Topology完全由Processor组成，因为它的数据固定由Kafka的Topic提供。
- Storm的不同Bolt运行在不同的Executor中，很可能位于不同的机器，需要通过网络通信传输数据。而Kafka Stream的Processor Topology的不同Processor完全运行于同一个Task中，也就完全处于同一个线程，无需网络通信。
- Storm的Topology可以同时包含Shuffle部分和非Shuffle部分，并且往往一个Topology就是一个完整的应用。而Kafka Stream的一个物理Topology只包含非Shuffle部分，而Shuffle部分需要通过`through`操作显示完成，该操作将一个大的Topology分成了2个子Topology。
- Storm的Topology内，不同Bolt/Spout的并行度可以不一样，而Kafka Stream的子Topology内，所有Processor的并行度完全一样。
- Storm的一个Task只包含一个Spout或者Bolt的实例，而Kafka Stream的一个Task包含了一个子Topology的所有Processor。

KTable vs. KStream

KTable和KStream是Kafka Stream中非常重要的两个概念，它们是Kafka实现各种语义的基础。因此这里有必要分析下二者的区别。

KStream是一个数据流，可以认为所有记录都通过Insert only的方式插入进这个数据流里。而KTable代表一个完整的数据集，可以理解为数据库中的表。由于每条记录都是Key-Value对，这里可以将Key理解为数据库中的Primary Key，而Value可以理解为一行记录。可以认为KTable中的数据都是通过Update only的方式进入的。也就意味着，如果KTable对应的Topic中新进入的数据的Key已经存在，那么从KTable只会取出同一Key对应的最后一条数据，相当于新的数据更新了旧的数据。

以下图为例，假设有一个KStream和KTable，基于同一个Topic创建，并且该Topic中包含如下图所示5条数据。此时遍历KStream将得到与Topic内数据完全一样的所有5条数据，且顺序不变。而此时遍历KTable时，因为这5条记录中有3个不同的Key，所以将得到3条记录，每个Key对应最新的值，并且这三条数据之间的顺序与原来在Topic中的顺序保持一致。这一点与Kafka的日志compact相同。

此时如果对该KStream和KTable分别基于key做Group，对Value进行Sum，得到的结果将会不同。对KStream的计算结果是<Jack，4>，<Lily，7>，<Mike，4>。而对Ktable的计算结果是<Mike，4>，<Jack，3>，<Lily，5>。

State store

流式处理中，部分操作是无状态的，例如过滤操作（Kafka Stream DSL中用filer方法实现）。而部分操作是有状态的，需要记录中间状态，如Window操作和聚合计算。State store被用来存储中间状态。它可以是一个持久化的Key-Value存储，也可以是内存中的HashMap，或者是数据库。Kafka提供了基于Topic的状态存储。

Topic中存储的数据记录本身是Key-Value形式的，同时Kafka的log compaction机制可对历史数据做compact操作，保留每个Key对应的最后一个Value，从而在保证Key不丢失的前提下，减少总数据量，从而提高查询效率。

构造KTable时，需要指定其state store name。默认情况下，该名字也即用于存储该KTable的状态的Topic的名字，遍历KTable的过程，实际就是遍历它对应的state store，或者说遍历Topic的所有key，并取每个Key最新值的过程。为了使得该过程更加高效，默认情况下会对该Topic进行compact操作。

另外，除了KTable，所有状态计算，都需要指定state store name，从而记录中间状态。

Kafka Stream如何解决流式系统中关键问题

时间

在流式数据处理中，时间是数据的一个非常重要的属性。从Kafka 0.10开始，每条记录除了Key和Value外，还增加了`timestamp`属性。目前Kafka Stream支持三种时间

- 事件发生时间。事件发生的时间，包含在数据记录中。发生时间由Producer在构造ProducerRecord时指定。并且需要Broker或者Topic将`message.timestamp.type`设置为`CreateTime`（默认值）才能生效。
- 消息接收时间，也即消息存入Broker的时间。当Broker或Topic将`message.timestamp.type`设置为`LogAppendTime`时生效。此时Broker会在接收到消息后，存入磁盘前，将其`timestamp`属性值设置为当前机器时间。一般消息接收时间比较接近于事件发生时间，部分场景下可代替事件发生时间。
- 消息处理时间，也即Kafka Stream处理消息时的时间。

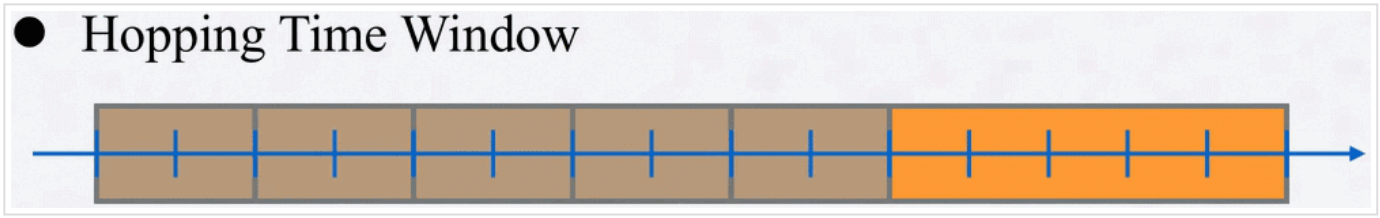
注：Kafka Stream允许通过实现`org.apache.kafka.streams.processor.TimestampExtractor`接口自定义记录时间。

窗口

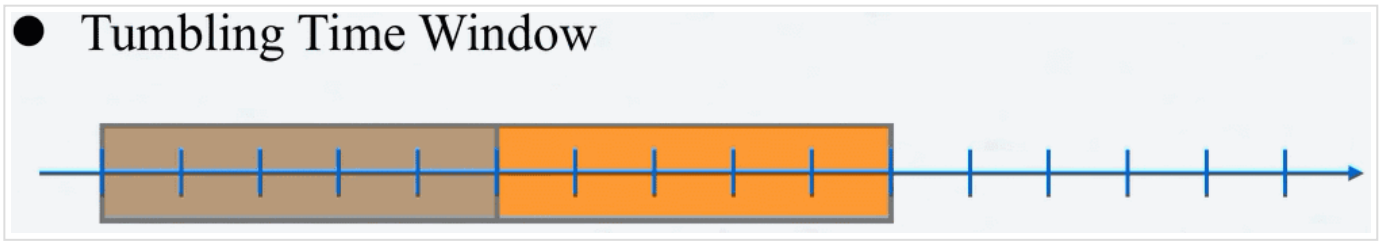
前文提到，流式数据是在时间上无界的数据。而聚合操作只能作用在特定的数据集，也即有界的数据集上。因此需要通过某种方式从无界的数据集上按特定的语义选取出有界的数据。窗口是一种非常常用的设定计算边界的方式。不同的流式处理系统支持的窗口类似，但不尽相同。

Kafka Stream支持的窗口如下。

- `Hopping Time Window` 该窗口定义如下图所示。它有两个属性，一个是Window size，一个是Advance interval。Window size指定了窗口的大小，也即每次计算的数据集的大小。而Advance interval定义输出的时间间隔。一个典型的应用场景是，每隔5秒钟输出一次过去1个小时内网站的PV或者UV。



- Tumbling Time Window该窗口定义如下图所示。可以认为它是Hopping Time Window的一种特例，也即Window size和Advance interval相等。它的特点是各个Window之间完全不相交。



- Sliding Window该窗口只用于2个KStream进行Join计算时。该窗口的大小定义了Join两侧KStream的数据记录被认为在同一个窗口的最大时间差。假设该窗口的大小为5秒，则参与Join的2个KStream中，记录时间差小于5的记录被认为在同一个窗口中，可以进行Join计算。

- Session Window该窗口用于对Key做Group后的聚合操作中。它需要对Key做分组，然后对组内的数据根据业务需求定义一个窗口的起始点和结束点。一个典型的案例是，希望通过Session Window计算某个用户访问网站的时间。对于一个特定的用户（用Key表示）而言，当发生登录操作时，该用户（Key）的窗口即开始，当发生退出操作或者超时时，该用户（Key）的窗口即结束。窗口结束时，可计算该用户的访问时间或者点击次数等。

Join

Kafka Stream由于包含KStream和Ktable两种数据集，因此提供如下Join计算

- KTable Join KTable 结果仍为KTable。任意一边有更新，结果KTable都会更新。
- KStream Join KStream 结果为KStream。必须带窗口操作，否则会造成Join操作一直不结束。
- KStream Join KTable / GlobakKTable 结果为KStream。只有当KStream中有新数据时，才会触发Join计算并输出结果。KStream无新数据时，KTable的更新并不会触发Join计算，也不会输出数据。并且该更新只对下次Join生效。一个典型的使用场景是，KStream中的订单信息与KTable中的用户信息做关联计算。

对于Join操作，如果要得到正确的计算结果，需要保证参与Join的KTable或KStream中Key相同的数据被分配到同一个Task。具体方法是

- 参与Join的KTable或KStream的Key类型相同（实际上，业务含意也应该相同）
- 参与Join的KTable或KStream对应的Topic的Partition数相同
- Partitioner策略的最终结果等效（实现不需要完全一样，只要效果一样即可），也即Key相同的情况下，被分配到ID相同的Partition内

如果上述条件不满足，可通过调用如下方法使得它满足上述条件。

> KStream<K, V> through(Serde<K> keySerde, Serde<V> valSerde, StreamPartitioner<K, V> partitioner, String topic)

合与乱序处理

聚合操作可应用于KStream和KTable。当聚合发生在KStream上时必须指定窗口，从而限定计算的目标数据集。

需要说明的是，聚合操作的结果肯定是KTable。因为KTable是可更新的，可以在晚到的数据到来时（也即发生数据乱序时）更新结果KTable。

这里举例说明。假设对KStream以5秒为窗口大小，进行Tumbling Time Window上的Count操作。并且KStream先后出现时间为1秒, 3秒, 5秒的数据，此时5秒的窗口已达上限，Kafka Stream关闭该窗口，触发Count操作并将结果3输出到KTable中（假设该结果表示为<1-5,3>）。若1秒后，又收到了时间为2秒的记录，由于1-5秒的窗口已关闭，若直接抛弃该数据，则可认为之前的结果<1-5,3>不准确。而如果直接将完整的结果<1-5,4>输出到KStream中，则KStream中将会包含该窗口的2条记录，<1-5,3>, <1-5,4>，也会存在肮数据。因此Kafka Stream选择将聚合结果存于KTable中，此时新的结果<1-5,4>会替代旧的结果<1-5,3>。用户可得到完整的正确的结果。

这种方式保证了数据准确性，同时也提高了容错性。

但需要说明的是，Kafka Stream并不会对所有晚到的数据都重新计算并更新结果集，而是让用户设置一个`retention period`，将每个窗口的结果集在内存中保留一定时间，该窗口内的数据晚到时，直接合并计算，并更新结果KTable。超过`retention period`后，该窗口结果将从内存中删除，并且晚到的数据即使落入窗口，也会被直接丢弃。

容错

Kafka Stream从如下几个方面进行容错

- 高可用的Partition保证无数据丢失。每个Task计算一个Partition，而Kafka数据复制机制保证了Partition内数据的高可用性，故无数据丢失风险。同时由于数据是持久化的，即使任务失败，依然可以重新计算。
- 状态存储实现快速故障恢复和从故障点继续处理。对于Join和聚合及窗口等有状态计算，状态存储可保存中间状态。即使发生Failover或Consumer Rebalance，仍然可以通过状态存储恢复中间状态，从而可以继续从Failover或Consumer Rebalance前的点继续计算。
- KTable与`retention period`提供了对乱序数据的处理能力。

Kafka Stream应用示例

下面结合一个案例来讲解如何开发Kafka Stream应用。本例完整代码可从作者Github获取。

订单KStream（名为orderStream），底层Topic的Partition数为3，Key为用户名，Value包含用户名，商品名，订单时间，数量。用户KTable（名为userTable），底层Topic的Partition数为3，Key为用户名，Value包含性别，地址和年龄。商品KTable（名为itemTable），底层Topic的Partition数为6，Key为商品名，价格，种类和产地。现在希望计算每小时购买产地与自己所在地相同的用户总数。

首先由于希望使用订单时间，而它包含在orderStream的Value中，需要通过提供一个实现TimestampExtractor接口的类从orderStream对应的Topic中抽取出订单时间。

```
public class OrderTimestampExtractor implements TimestampExtractor {

  @Override
  public long extract(ConsumerRecord<Object, Object> record) {
    if(record instanceof Order) {
      return ((Order)record).getTS();
    } else {
      return 0;
    }
  }
}
```

接着通过将orderStream与userTable进行Join，来获取订单用户所在地。由于二者对应的Topic的Partition数相同，且Key都为用户名，再假设Producer往这两个Topic写数据时所用的Partitioner实现相同，则此时上文所述Join条件满足，可直接进行Join。

```
orderUserStream = orderStream
    .leftJoin(userTable,
         // 该lamda表达式定义了如何从orderStream与userTable生成结果集的Value
        (Order order, User user) -> OrderUser.fromOrderUser(order, user),
         // 结果集Key序列化方式
        Serdes.String(),
         // 结果集Value序列化方式
         SerdesFactory.serdFrom(Order.class))
    .filter((String userName, OrderUser orderUser) -> orderUser.userAddress != null)
```

从上述代码中，可以看到，Join时需要指定如何从参与Join双方的记录生成结果记录的Value。Key不需要指定，因为结果记录的Key与Join Key相同，故无须指定。Join结果存于名为orderUserStream的KStream中。

接下来需要将orderUserStream与itemTable进行Join，从而获取商品产地。此时orderUserStream的Key仍为用户名，而itemTable对应的Topic的Key为产品名，并且二者的Partition数不一样，因此无法直接Join。此时需要通过through方法，对其中一方或双方进行重新分区，使得二者满足Join条件。这一过程相当于Spark的Shuffle过程和Storm的FieldGrouping。

```
orderUserStrea
    .through(
        // Key的序列化方式
        Serdes.String(),
        // Value的序列化方式
        SerdesFactory.serdFrom(OrderUser.class),
        // 重新按照商品名进行分区，具体取商品名的哈希值，然后对分区数取模
        (String key, OrderUser orderUser, int numPartitions) -> (orderUser.getItemName().hashCode()
& 0x7FFFFFFF) % numPartitions,
        "orderuser-repartition-by-item")
    .leftJoin(itemTable, (OrderUser orderUser, Item item) -> OrderUserItem.fromOrderUser(orderUser,
item), Serdes.String(), SerdesFactory.serdFrom(OrderUser.class))
```

从上述代码可见，through时需要指定Key的序列化器，Value的序列化器，以及分区方式和结果集所在的Topic。这里要注意，该Topic（orderuser-repartition-by-item）的Partition数必须与itemTable对应Topic的Partition数相同，并且through使用的分区方法必须与iteamTable对应Topic的分区方式一样。经过这种through操作，orderUserStream与itemTable满足了Join条件，可直接进行Join。

总结

- Kafka Stream的并行模型完全基于Kafka的分区机制和Rebalance机制，实现了在线动态调整并行度
- 同一Task包含了一个子Topology的所有Processor，使得所有处理逻辑都在同一线程内完成，避免了不必的网络通信开销，从而提高了效率。
- through方法提供了类似Spark的Shuffle机制，为使用不同分区策略的数据提供了Join的可能
- log compact提高了基于Kafka的state store的加载效率
- state store为状态计算提供了可能
- 基于offset的计算进度管理以及基于state store的中间状态管理为发生Consumer rebalance或Failover时从断点处继续处理提供了可能，并为系统容错性提供了保障
- KTable的引入，使得聚合计算拥用了处理乱序问题的能力