

30 | 流计算与消息（二）：在流计算中使用Kafka链接计算任务

李玥 · 消息队列高手课



你好，我是李玥。

上节课我们一起实现了一个流计算的例子，并通过这个例子学习了流计算的实现原理。我们知道，流计算框架本身是个分布式系统，一般由多个节点组成一个集群。我们的计算任务在计算集群中运行的时候，会被拆分成多个子任务，这些子任务也是分布在集群的多个计算节点上的。

大部分流计算平台都会采用存储计算分离的设计，将计算任务的状态保存在 HDFS 等分布式存储系统中。每个子任务将状态分离出去之后，就变成了无状态的节点，如果某一个计算节点发生宕机，使用集群中任意一个节点都可以替代故障节点。

但是，对流计算来说，这里面还有一个问题没解决，就是在集群中流动的数据并没有被持久化，所以它们就有可能由于节点故障而丢失，怎么解决这个问题呢？办法也比较简单粗暴，就是直接重启整个计算任务，并且从数据源头向前回溯一些数据。计算任务重启之后，会重新分配计算节点，顺便就完成了故障迁移。

回溯数据源，可以保证数据不丢失，这和消息队列中，通过重发未成功的消息来保证数据不丢的方法是类似的。所以，它们面临同样的问题：可能会出现重复的消息。消息队列可以通过在消费端做幂等来克服这个问题，但是对于流计算任务来说，这个问题就很棘手了。

对于接收计算结果的下游系统，它可能会收到重复的计算结果，这还不是最糟糕的。像一些统计类的计算任务，就会有比较大的影响，比如上节课中统计访问次数的例子，本来这个 IP 地址在统计周期内被访问了 5 次，产生了 5 条访问日志，正确的结果应该是 5 次。如果日志被重复统计，那结果就会多于 5 次，重复的数据导致统计结果出现了错误。怎么解决这个问题呢？

我们之前提到过，Kafka 支持 Exactly Once 语义，它的这个特性就是为了解决这个问题而生的。这节课，我们就来通过一个例子学习一下，如何使用 Kafka 配合 Flink，解决数据重复的问题，实现端到端的 Exactly Once 语义。

Flink 是如何保证 Exactly Once 语义的？

我们所说的端到端 Exactly Once，这里面的“端到端”指的是，数据从 Kafka 的 A 主题消费，发送给 Flink 的计算集群进行计算，计算结果再发给 Kafka 的 B 主题。在这整个过程中，无论是 Kafka 集群的节点还是 Flink 集群的节点发生故障，都不会影响计算结果，每条消息只会被计算一次，不能多也不能少。

在理解端到端 Exactly Once 的实现原理之前，我们需要先了解一下，Flink 集群本身是如何保证 Exactly Once 语义的。为什么 Flink 也需要保证 Exactly Once 呢？Flink 集群本身也是一个分布式系统，它首先需要保证数据在 Flink 集群内部只被计算一次，只有在这个基础上，才谈得到端到端的 Exactly Once。

Flink 通过 CheckPoint 机制来定期保存计算任务的快照，这个快照中主要包含两个重要的数据：

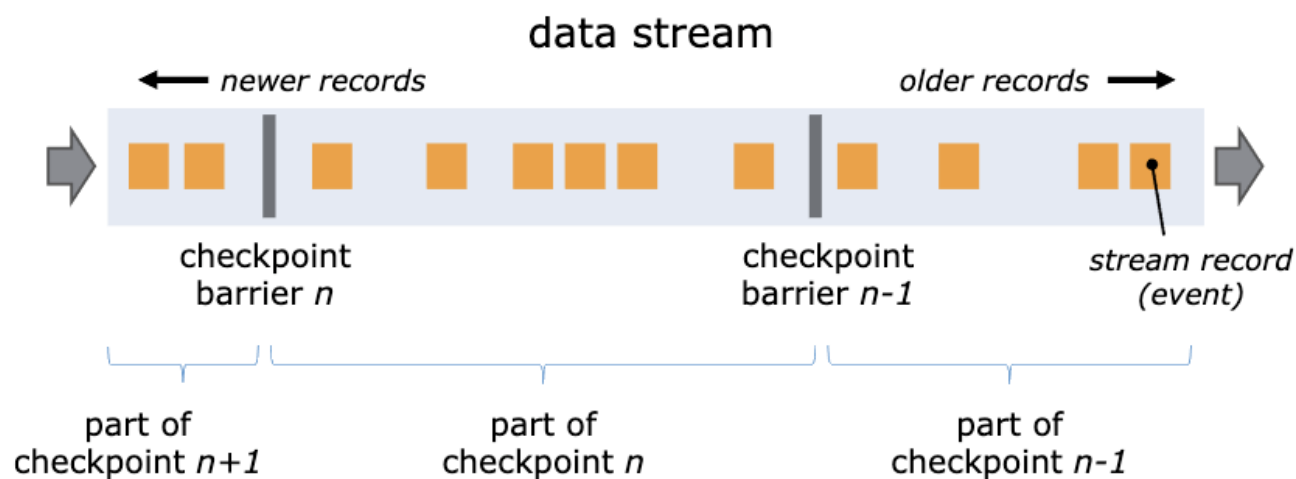
1. 整个计算任务的状态。这个状态主要是计算任务中，每个子任务在计算过程中需要保存的临时状态数据。比如，上节课例子中汇总了一半的数据。

2. 数据源的位置信息。这个信息记录了在数据源的这个流中已经计算了哪些数据。如果数据源是 Kafka 的主题，这个位置信息就是 Kafka 主题中的消费位置。

有了 CheckPoint，当计算任务失败重启的时候，可以从最近的一个 CheckPoint 恢复计算任务。具体的做法是，每个子任务先从 CheckPoint 中读取并恢复自己的状态，然后整个计算任务从 CheckPoint 中记录的数据源位置开始消费数据，只要这个恢复位置和 CheckPoint 中每个子任务的状态是完全对应的，或者说，每个子任务的状态恰好是：“刚刚处理完恢复位置之前的那条数据，还没有开始处理恢复位置对应的这条数据”，这个时刻保存的状态，就可以做到严丝合缝地恢复计算任务，每一条数据既不会丢失也不会重复。

因为每个子任务分布在不同的节点上，并且数据是一直在子任务中流动的，所以确保 CheckPoint 中记录的恢复位置和每个子任务的状态完全对应并不是一件容易的事儿，Flink 是怎么实现的呢？

Flink 通过在数据流中插入一个 Barrier（屏障）来确保 CheckPoint 中的位置和状态完全对应。下面这张图来自 [Flink 官网的说明文档](#)。



你可以把 Barrier 理解为一条特殊的数据。Barrier 由 Flink 生成，并在数据进入计算集群时被插入到数据流中。这样，无限的数据流就被很多的 Barrier 分隔成很多段。Barrier 在流经每个计算节点的时候，就会触发这个节点在 CheckPoint 中保存本节点的状态，如果这个节点是数据源节点，还会保存数据源的位置。

当一个 Barrier 流过所有计算节点，流出计算集群后，一个 CheckPoint 也就保存完成了。由于每个节点都是在 Barrier 流过的时候保存的状态，这时的状态恰好就是 Barrier 所在位置（也就是 CheckPoint 数据源位置）对应的状态，这样就完美解决了状态与恢复位置对应的问题。

Flink 通过 CheckPoint 机制实现了集群内计算任务的 Exactly Once 语义，但是仍然实现不了在输入和输出两端数据不丢不重。比如，Flink 在把一条计算结果发给 Kafka 并收到来自 Kafka 的“发送成功”响应之后，才会继续处理下一条数据。如果这个时候重启计算任务，Flink 集群内的数据都可以完美地恢复到上一个 CheckPoint，但是已经发给 Kafka 的消息却没办法撤回，还是会出现数据重复的问题。

所以，我们需要配合 Kafka 的 Exactly Once 机制，才能实现端到端的 Exactly Once。

Kafka 如何配合 Flink 实现端到端 Exactly Once?

Kafka 的 Exactly Once 语义是通过它的事务和生产幂等两个特性来共同实现的。其中 Kafka 事务的实现原理，我们在《[025 | RocketMQ 与 Kafka 中如何实现事务？](#)》这节课中讲过。它可以保证一个事务内的所有消息，要么都成功投递，要么都不投递。

生产幂等这个特性可以保证，在生产者给 Kafka Broker 发送消息这个过程中，消息不会重复发送。这个实现原理和我们在《[005 | 如何确保消息不会丢失？](#)》这节课中介绍的“检测消息丢失”的方法是类似的，都是通过连续递增的序号进行检测。Kafka 的生产者给每个消息增加都附加一个连续递增的序号，Broker 端会检测这个序号的连续性，如果序号重复了，Broker 会拒绝这个重复消息。

Kafka 的这两个机制，配合 Flink 就可以来实现端到端的 Exactly Once 了。简单地说就是，每个 Flink 的 CheckPoint 对应一个 Kafka 事务。Flink 在创建一个 CheckPoint 的时候，同时开启一个 Kafka 的事务，完成 CheckPoint 同时提交 Kafka 的事务。当计算任务重启的时候，在 Flink 中计算任务会恢复到上一个 CheckPoint，这个 CheckPoint 正好对应 Kafka 上一个成功提交的事务。未完成的 CheckPoint 和未提交的事务中的消息都会被丢弃，这样就实现了端到端的 Exactly Once。

但是，怎样才能保证“完成 CheckPoint 同时提交 Kafka 的事务”呢？或者说，如何来保证“完成 CheckPoint”和“提交 Kafka 事务”这两个操作，要么都成功，要么都失败呢？这不就是一个典型的分布式事务问题嘛！

所以，Flink 基于两阶段提交这个常用的分布式事务算法，实现了一分布式事务的控制器来解决这个问题。如果你对具体的实现原理感兴趣，可以看一下 Flink 官网文档中的 [这篇文章](#)。

Exactly Once 版本的 Web 请求的统计

下面进入实战环节，我们来把上节课的“统计 Web 请求的次数”的 Flink Job 改造一下，让这个 Job 具备 Exactly Once 特性。这个实时统计任务接收 NGINX 的 access.log，每 5 秒钟按照 IP 地址统计 Web 请求的次数。假设我们已经有一个实时发送 access.log 的日志服务来发送日志，日志的内容只包含访问时间和 IP 地址，这个日志服务就是我们流计算任务的数据源。

改造之后，我们需要把数据的来源替换成 Kafka 的 ip_count_source 主题，计算结果也要保存到 Kafka 的主题 ip_count_sink 中。

整个系统的数据流向就变成下图这样：



日志服务将日志数据发送到 Kafka 的主题 ip_count_source，计算任务消费这个主题的数据作为数据源，计算结果会被写入到 Kafka 的主题 ip_count_sink 中。

Flink 提供了 Kafka Connector 模块，可以作为数据源从 Kafka 中消费数据，也可以作为 Kafka 的 Producer，将计算结果发送给 Kafka，并且，这个 Kafka Connector 已经实现了 Exactly Once 语义，我们在使用的时候只要做适当的配置就可以了。


这次我们用 Java 语言来实现这个任务，改造后的计算任务代码如下：

```
1 public class ExactlyOnceIpCount {
2     public static void main(String[] args) throws Exception {
3
4         // 设置输入和输出
5         FlinkKafkaConsumer011<IpAndCount> sourceConsumer = setupSource();
6         FlinkKafkaProducer011<String> sinkProducer = setupSink();
7
8         // 设置运行时环境
9         final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExec
10 env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime); // 按照EventTime
11 env.enableCheckpointing(5000); // 每5秒保存一次Checkpoint
12 // 设置Checkpoint
13 CheckpointConfig config = env.getCheckpointConfig();
14 config.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE); // 设置CheckpointingMode
15 config.enableExternalizedCheckpoints(
16     CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
17 config.setPreferCheckpointForRecovery(true); // 启动时从Checkpoint恢复任务
18
19 // 设置Checkpoint的StateBackend, 在这里Checkpoint保存在本地临时目录中。
20 // 只适合单节点做实验, 在生产环境应该使用分布式文件系统, 例如HDFS。
21 File tmpDirFile = new File(System.getProperty("java.io.tmpdir"));
22 env.setStateBackend((StateBackend) new FsStateBackend(tmpDirFile.toURI()));
23 // 设置故障恢复策略: 任务失败的时候自动每隔10秒重启, 一共尝试重启3次
24 env.setRestartStrategy(RestartStrategies.fixedDelayRestart(
25     3, // number of restart attempts
26     10000 // delay
27 ));
28
29 // 定义输入: 从Kafka中获取数据
30 DataStream<IpAndCount> input = env
31     .addSource(sourceConsumer);
32
33 // 计算: 每5秒钟按照ip对count求和
34 DataStream<IpAndCount> output =
35     input
36     .keyBy(IpAndCount::getIp) // 按照ip地址统计
37     .window(TumblingEventTimeWindows.of(Time.seconds(5))) // 每5秒钟统计
38     .allowedLateness(Time.seconds(5))
39     .sum("count"); // 对count字段求和
40
41 // 输出到kafka topic
42 output.map(IpAndCount::toString).addSink(sinkProducer);
43
44 // execute program
45 env.execute("Exactly-once IpCount");
46 }
47 }
```


这段代码和上节课中原始版本的代码整体架构是差不多的，同样是：定义数据源、定义计算逻辑和定义输入这三大步骤。下面主要来说不同之处，这些不同的地方也就是如何配置 Exactly Once 特性的关键点。

首先，我们需要开启并配置好 CheckPoint。在这段代码中，我们开启了 CheckPoint，设置每 5 秒钟创建一个 CheckPoint。然后，还需要定义保存 CheckPoint 的 StateBackend，也就是告诉 Flink 把 CheckPoint 保存在哪儿。在生产环境中，CheckPoint 应该保存到 HDFS 这样的分布式文件系统中。我们这个例子中，为了方便运行调试，直接把 CheckPoint 保存到本地的临时目录中。之后，我们还需要将 Job 配置成自动重启，这样当节点发生故障时，Flink 会自动重启 Job 并从最近一次 CheckPoint 开始恢复。

我们在定义输出创建 FlinkKafkaProducer 的时候，需要指定 Exactly Once 语义，这样 Flink 才会开启 Kafka 的事务，代码如下：

 复制代码

```
1 private static FlinkKafkaProducer011<String> setupSink() {
2     // 设置Kafka Producer属性
3     Properties producerProperties = new Properties();
4     producerProperties.put("bootstrap.servers", "localhost:9092");
5     // 事务超时时间设置为1分钟
6     producerProperties.put("transaction.timeout.ms", "60000");
7
8     // 创建 FlinkKafkaProducer, 指定语义为EXACTLY_ONCE
9     return new FlinkKafkaProducer011<>(
10         "ip_count_sink",
11         new KeyedSerializationSchemaWrapper<>(new SimpleStringSchema()),
12         producerProperties,
13         FlinkKafkaProducer011.Semantic.EXACTLY_ONCE);
14 }
```

最后一点需要注意的，在从 Kafka 主题 ip_count_sink 中消费计算结果的时候，需要配置 Consumer 属性：isolation.level=read_committed，也就是只消费已提交事务的消息。因为默认情况下，Kafka 的 Consumer 是可以消费到未提交事务的消息的。

这个例子的完整代码我放到了 GitHub 上，编译和运行这个例子的方法我也写在了项目的 README 中，你可以点击 [🔗 这里](#) 查看。

小结

端到端 Exactly Once 语义，可以保证在分布式系统中，每条数据不多不少只被处理一次。在流计算中，因为数据重复会导致计算结果错误，所以 Exactly Once 在流计算场景中尤其重要。Kafka 和 Flink 都提供了保证 Exactly Once 的特性，配合使用可以实现端到端的 Exactly Once 语义。

在 Flink 中，如果节点出现故障，可以自动重启计算任务，重新分配计算节点来保证系统的可用性。配合 CheckPoint 机制，可以保证重启后任务的状态恢复到最后一次 CheckPoint，然后从 CheckPoint 中记录的恢复位置继续读取数据进行计算。Flink 通过一个巧妙的 Barrier 使 CheckPoint 中恢复位置和各节点状态完全对应。

Kafka 的 Exactly Once 语义是通过它的事务和生产幂等两个特性来共同实现的。在配合 Flink 的时候，每个 Flink 的 CheckPoint 对应一个 Kafka 事务，只要保证 CheckPoint 和 Kafka 事务同步提交就可以实现端到端的 Exactly Once，Flink 通过“二阶段提交”这个分布式事务的经典算法来保证 CheckPoint 和 Kafka 事务状态的一致性。

可以看到，Flink 配合 Kafka 来实现端到端的 Exactly Once 语义，整个实现过程比较复杂，但是，这个复杂的大问题是由一个一个小问题组成的，每个小问题的原理都是很简单的。比如：Kafka 如何实现的生产幂等？Flink 如何通过存储计算分离解决子任务状态恢复的？很多这些小问题和我们课程中遇到的类似问题是差不多的，那你就可以用到我们学习过的解决方法。

你需要重点掌握的是，每一个小问题它面临的场景是什么样的，以及如何解决问题的方法。而不要拘泥于，Kafka 或者 Flink 的某个参数怎么配这些细节问题。这些问题可以等到你在生产中真正需要使用的时候，再去读文档，“现学现卖”都来得及。

思考题

我们的课程中反复强调过，在消息队列的消费端，一定要“先执行消费业务逻辑，再确认消费”，这样才能保证不丢数据。我们这节课中，并没有提到 FlinkKafkaConsumer 在从数据源主题 ip_count_sink 消费数据之后，如何来确认消费的。如果消费位置管理不好，一样会导致消息丢失或者重复，课后请你查看一下相关的文档和源代码，看一下 FlinkKafkaConsumer 是如何来确认消费的。欢迎在留言区与我分享讨论。

感谢阅读，如果你觉得这篇文章对你有一些启发，也欢迎把它分享给你的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (9)



D Fighting

2019-10-28

关于思考题，我在infoQ上找了一篇文章https://www.infoq.cn/article/58bzvlbT2fqyW*cXzGIG，不知道是不是这么实现的，请老师帮忙看下。

作者回复: 是这样的。



14



张天屹

2019-10-05

老师你好，能介绍下Kafka 配合 Flink，与Kafka Stream 的核心区别吗

作者回复: Kafka Stream目前来说，相关的生态还不够成熟，可以了解一下，但不建议在生产系统中使用。

它和flink最大的区别是，它是一个库，运行在你的应用程序进程内，而不是一个流计算框架。



3



jack

2019-10-09

老师，使用spark streaming 和kafka时，

1、spark官方文档说，如果保存到checkpoint和把offset 提交到kafka，必须保证输出是幂等的，光使用事务是不行的；

2、那么如果无法保证输出是幂等的，是否只能把offset 保存在第三方的数据库(比如redis)中，但是这样做是否是不可以设置checkpoints？否则spark依然会从checkpoint中读取，和从数据库中读取会造成冲突呢？

3、但不设置checkpoint，spark如何恢复现场呢？在提交命令时加入--supervise，好像yarn的模式不支持？即使使用supervise重启，没有checkpoint，也无法恢复现场吧？

作者回复: A1: 是这样的，所以Kafka的Exactly Once特性中是有事务和生产幂等（相当于流计算输出幂等）二个功能组成的。

A2: 这个方法不太可行，因为你很难做到完美的故障恢复。原因我在课程中也讲到了。

A3: 具体操作细节层面的问题，还是建议你以官方的文档为准。



👍 2



Geek_c24555

2020-06-13

老师好，请问下 rocketmq 可以配合 flink 实现exactly once 吗

作者回复: 据我了解，RocketMQ目前还没有支持。你可以跟踪一下这个Issue: <https://github.com/apache/rocketmq-externals/issues/500>



👍 2



i_chase

2022-03-28

kafka ==> flink ==> kafka虽然实现了exactly once,但是最终进入output kafka的数据不也需要消费出来的吗？

是不是因为这里从output topic消费只是打印一下消息，即使重复消费也没关系？



Geek_7825d4

2021-03-24

Flink 中有个 AsyncIO 算子，自身提供了 Exactly Once 语义，但是一致有个问题不太清楚，AsyncIO 为了提升性能，提供异步无序处理的方式，这种情况下，假设有一个 offset 为 1 的数据 阻塞在异步中，会不会有1一个 offset 为 2 的消息已经被处理完了，并继续向下游发

送。此时当 2 已经端到端完成时，Flink 是否会向 Kafka Source 提交 2 位置的偏移量呢，这样如何保证 Exactly Once 呢



Heaven

2021-02-20

看了上面同学的留下的文章,发现Flink是维护两个位置offset和committedOffset,内部在进行快照保存的时候,保存了offset作为快照内部位置,在快照完成之后,会变动维护的committedOffset属性值,将变更后的committedOffset提交到Kafka brokers或者ZK中



长脖子树

2020-08-26

正好最近在看 flink , 这部分端到端的 exactly once 实现讲得很清晰!



不惑ing

2019-10-06

第25章讲kafka exactly once需要从kafka topicA读取计算再保存到kafka topicB,但从这章讲的流程看,最后不需要保存到kafka topicB, 保存到其他hdfs里也可以,

所以最后一步保存位置有具体要求吗?

作者回复: 理论上是可以的, 但是实际上hdfs没有原生事务支持, 实现起来比较困难。

共 2 条评论 >

