Apache Flink 漫谈系列 - Fault Tolerance

什么是Fault Tolerance

容错(Fault Tolerance) 是指容忍故障，在故障发生时能够自动检测出来并使系统能够自动回复正常运行。当出现某些指定的网络故障、硬件故障、软件错误时，系统仍能执行规定的一组程序，或者说程序不会因系统中的故障而中止，并且执行结果也不包含系统中故障所引起的差错。

传统数据库Fault Tolerance

我们在 后续会在 《流表对偶(duality)性篇》 中会介绍过mysql的主备复制机制，其中binlog是一个append only的日志文件，Mysql的主备复制是高可用的主要方式，binlog是主备复制的核心手段（当然mysql高可用细节也很复杂和多种不同的优化点，如 纯异步复制优化为半同步和同步复制以保证异步复制binlog导致的master和slave的同步时候网络坏掉，导致主备不一致问题等)。Mysql主备复制，是容错机制的一部分，在容错机制之中也包括事物控制，在传统数据库中事物可以设置不同的级别，以保证数据不同的质量，级别由低到高 如下：

- Read uncommitted - 读未提交，就是一个事务可以读取另一个未提交事务的数据。那么这种事物控制成本最低，但是会导致另一个事物读都时候脏数据，那么怎么解决读脏数据呢？利用Read committed 级别...
- Read committed - 读提交，就是一个事务要等另一个事务提交后才能读取数据。这种级别可以解决读脏数据的问题，那么这种级别有什么问题呢？这个级别还有一个不能重复读的问题，即：开启一个读事物时候T1，先读取字段F1值是V1，这时候另一个事物T2可以UPDATA这个字段值V2，导致T1再次读取字段值时候获得V2了，同一个事物中的两次读取不一致了。那么如何解决不可重复读的问题呢？利用Repeatable read 级别...
- Repeatable read - 重复读，就是在开始读取数据（事务开启）时，不再允许修改操作。重复读模式要有事物顺序的等待，需要一定的成本达到高质量的数据信息，那么重复读还会有什么问题吗？是的，重复读级别还有一个问题就是 幻读，幻读产生的原因是INSERT，那么幻读怎么解决呢？利用Serializable级别...
- Serializable - 序列化 是最高的事务隔离级别，在该级别下，事务串行化顺序执行，可以避免脏读、不可重复读与幻读。但是这种事务隔离级别效率低下，比较耗数据库性能，一般不使用。

主备复制，事物控制都是传统数据库容错的机制。

流计算Fault Tolerance的挑战

流计算Fault Tolerance的一个很大的挑战是低延迟，很多Flink任务都是7 x 24小时不间

断，端到端的秒级延迟，要想在遇上网络闪断，机器坏掉等非预期的问题时候快速恢复正常，并且不影响计算结果正确性是一件极其困难的事情。同时除了流计算的低延时要求，还有计算模式上面的挑战，在Flink中支持exactly-once和at-least-once两种计算模式，如何做到在failover时候不重复计算精准的做到exactly-once也是流计算Fault Tolerance要重点解决的问题。

Flink Fault Tolerance 机制

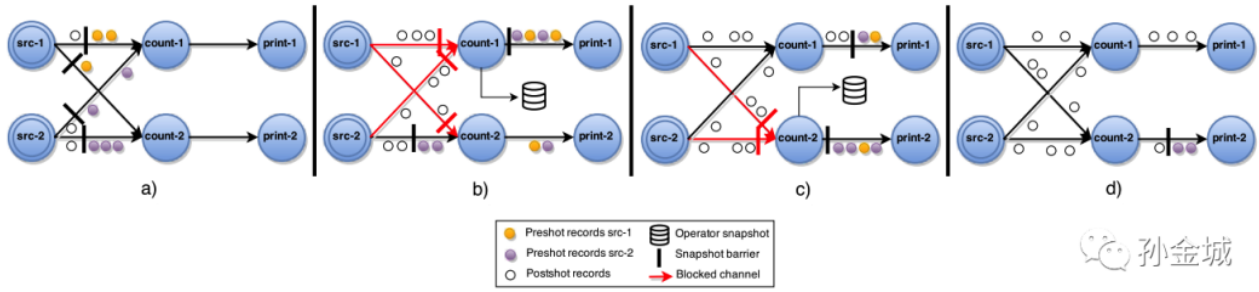
Flink Fault Tolerance机制上面理论基础与flink一致都是持续创建分布式数据流及其状态的快照。这些快照在系统遇到故障时，作为一个回退点。Blink中创建快照的机制叫做Checkpointing，Checkpointing的理论基础 Stephan 在 Lightweight Asynchronous Snapshots for Distributed Dataflows 进行了细节描述，该机制源于由K. MANI CHANDY 和 LESLIE LAMPORT 发表的 Determining-Global-States-of-a-Distributed-System Paper，该Paper描述了在分布式系统如何解决全局状态一致性问题。

在Flink中以checkpointing的机制进行容错，checkpointing会产生类似binlog一样的可以用来恢复的任务状态数据。Flink中也有类似于数据库事物控制（4个级别）一样的数据计算语义控制，在Flink中有另种语义模式设置，花费的成本有低到高，如下：

- at-least-once
- exactly-once

检查点-Checkpointing

上面我们说Checkpointing是Flink中Fault Tolerance的核心机制，我们以Checkpointing的方式创建包含timer，connector，window，user-defined state 等stateful Operator的快照。在Determining-Global-States-of-a-Distributed-System的全局状态一致性算法中重点描述了全局状态的对齐问题，在Lightweight Asynchronous Snapshots for Distributed Dataflows中核心描述了对齐的方式，在flink中采用以在流信息中插入barrier的方式完成DAG中异步快照。如下图(from Lightweight Asynchronous Snapshots for Distributed Dataflows)描述了Asynchronous barrier snapshots for acyclic graphs。也是Blink中采用的方式。

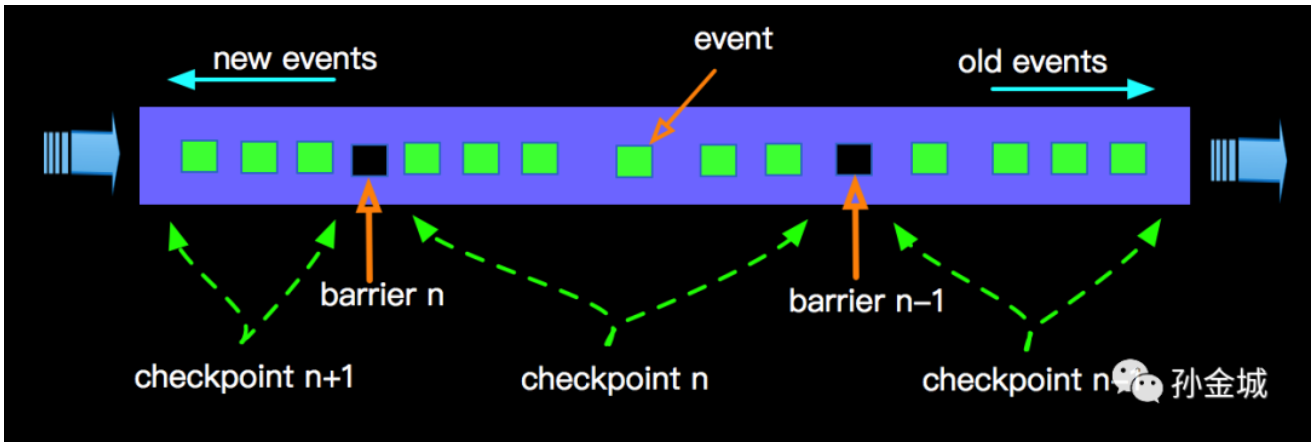


上图描述的是一个面描述 增量计算 word count的Job，上图核心说明了如下几点：

- barrier 由source节点发出；
- barrier会将流上event切分到不同的checkpoint中；
- 汇聚到当前节点的多流的barrier要对齐；

- barrier对齐之后会进行Checkpointing，生成snapshot；
- 完成snapshot之后向下游发出barrier，继续直到Sink节点；

这样在整个流上面以barrier方式进行Checkpointing，随着时间的推移，整个流的计算过程中按时间顺序不断的进行Checkpointing，如下图：



生成的snapshot会存储到StateBackend中，相关State的介绍可以查阅《Apache Flink漫谈系列 - State》。这样在进行failover时候，从最后一次成功的checkpoint进行恢复；

Checkpointing的控制

上面我们了解到整个流上面我们会随着时间推移不断的做Checkpointing，不断的产生snapshot存储到Statebackend中，那么多久进行一次Checkpointing？对产生的snapshot如何持久化的呢？带着这些疑问，我们看看Flink对于Checkpointing如何控制的？有哪些可配置的参数:

- checkpointMode - 检查点模式 AT_LEAST_ONCE 或 EXACTLY_ONCE
- checkpointInterval - 检查点时间间隔，单位是毫秒
- checkpointTimeout - 检查点超时时间， 单位毫秒

如何做到exactly-once

上面内容我们了解了Flink中exactly-once和at-least-once只是在进行checkpointing时候的配置模式，两种模式下进行checkpointing的原理是一致的，那么在实现上有什么本质区别呢?

语义

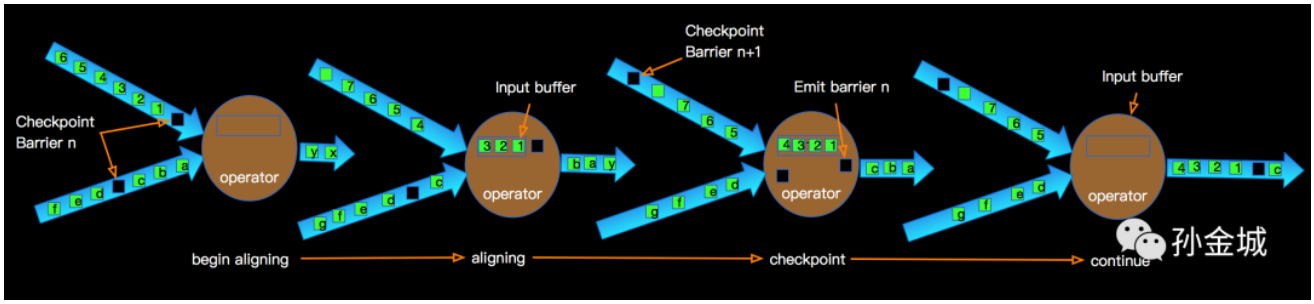
从语义上面exactly-once 比 at-least-once对数据处理的要求很严格，更精准，那么更高的要求就意味着更高的代价，这里的代价就是 延迟。

- at-least-once - 语义是指流上所有数据至少被处理过一次（不要丢数据）
- exactly-once - 语义是指流上所有数据必须被处理且只能处理一次（不丢数据，且不能重复）

实现

那在实现上面Flink中at-least-once 和 exactly-once有什么区别呢？区别体现在多路输入的时候（比如 Join），当所有输入的barrier没有完全到来的时候，早到来的event在exactly-once的情况会进行缓存（不进行处理），而at-least-once的模式下即使所有输入的barrier没有完全到来的时候，早到来的event也会进行处理。也就是说对于at-least-once模式下，对于下游节点而言，本来数据属于checkpoint n的数据在checkpoint n-1里面也可能处理过了。

我以exactly-once为例说明exactly-once模式相对于at-least-once模式为啥会有更高的延时？如下图：



上图示意了某个节点进行Checkpointing的过程：

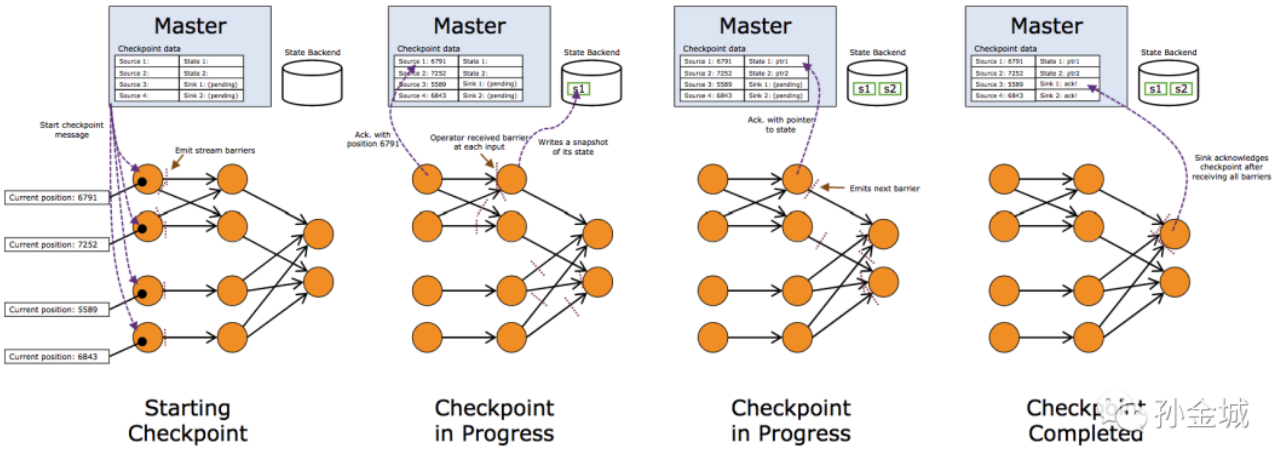
- 当Operator接收到某个上游发下来的barrier时候开始进行barrier的对齐阶段；
- 在进行对齐期间barrier之后到来的input的数据会被缓存到buffer中；
- 当Operator接收到上游所有barrier的时候，当前Operator会进行checkpointing，生成snapshot并持久化；
- 当checkpointing完成时候将barrier广播给下游operator；

当多路输入的barrier没有没有对齐时候，barrier先到的输入数据会缓存在buffer中，不进行处理，这样对于下游而言buffer的数据越多就有更大的延迟。这个延时带来的好处就是相邻checkpointing所记录的数据（计算结果或event)没有重复。相对at-least-once模式数据不会被buffer，减少延时的利好是以容忍数据重复计算为代价的。

完整Flink任务Checkpointing过程

在 《Apache Flink 漫谈系列 - State》中我们有过对Flink存储到State中的内容做过介绍，比如在connector会利用OperatorState记录读取位置offset，那么一个完整的Blink任务的执行图是一个DAG，上面我们描述了DAG中一个节点的过程，那么整体来看Checkpointing的过程是怎样的呢？在产生checkpoint并分布式持久到HDFS的过程是怎样的呢？

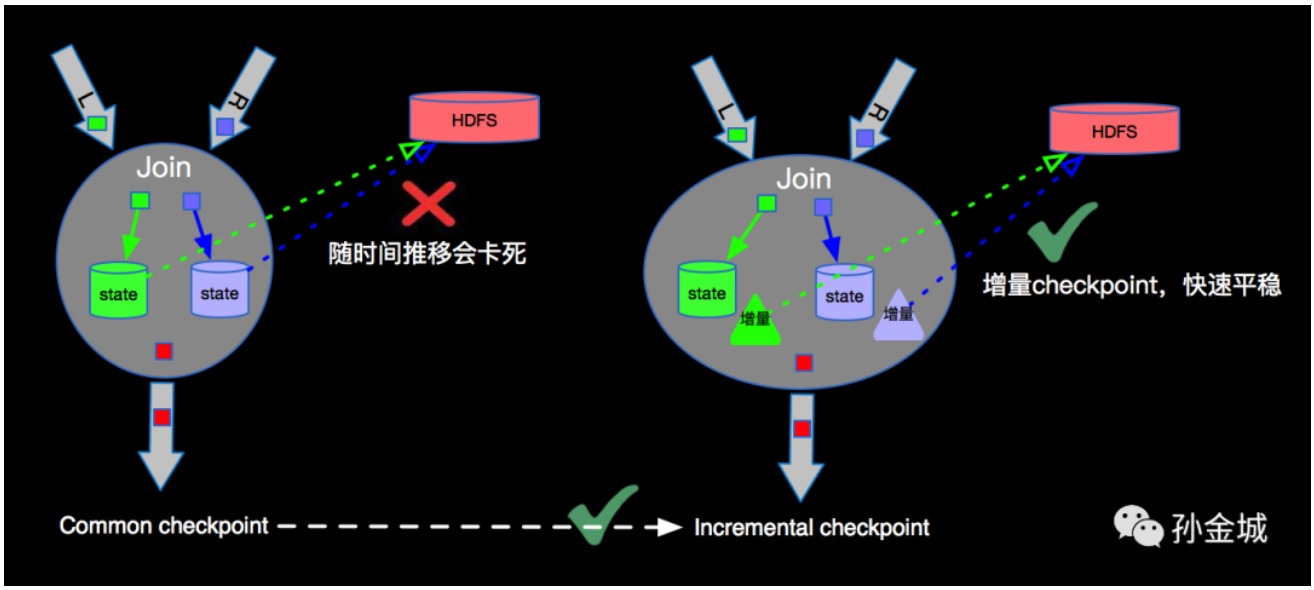
整体checkpoint流程



上图我们看到一个完整的Flink任务进行Checkpointing的过程，JM触发Soruce发射barriers,当某个Operator接收到上游发下来的barrier，开始进行barrier的处理，整体根据DAG自上而下的逐个节点进行Checkpointing，并持久化到Statebackend。一直到DAG的sink节点。

Incremental checkpoint

对于一个流计算的任务，数据会源源不断的流入，比如要进行双流JOIN，由于两边的流event的到来有先后顺序问题，我们必须将left和right的数据都会在state中进行存储，Left event流入会在Right的State进行join数据，Right event流入会在LState中join数据，如下图左右两边的数据都会持久化到State中：



由于流上数据源源不断，随着时间的增加，每次checkpoint产生的snapshot的文件（RocksDB的sst文件）会变的非常庞大，增加网络IO，拉长checkpoint时间，最终导致无法完成checkpoint，进而失去failover的能力。为了解决checkpoint不断变大的问题，Flink内部实现了Incremental checkpoint，这种增量进行checkpoint的机制，会大大减少checkpoint时间，并且如果业务数据稳定的情况下每次checkpoint的时间是相对稳定的，根据不同的业务需求设定checkpoint的interval，稳定快速的进行checkpointing，保障Flink任务在遇到故障时候可以顺利的进行failover。Incremental checkpoint的优化对于Flink成百上千的任务节点带来的利好不言而喻。

端到端exactly-once

根据上面的介绍我们知道Flink内部支持exactly-once，要想达到端到端（Soruce到Sink）的exactly-once，需要Flink外部Soruce和Sink的支持，比如Source要支持精准的offset，Sink要支持两阶段提交，也就是继承TwoPhaseCommitSinkFunction。

Unaligned Checkpoints

在刚刚发布的Flink1.11及以后版本中，为了解决在特殊情况下由于barrier对齐导致的Checkpoint时间过长，甚至Checkpoint失败问题，Flink提供了Unaligned checkpoins的机制。也就是可以将原来需要buffer的数据也存储到checkpoint的state文件中。具体使用和实现的细节我将在《Apache Flink 知其然，知其所以然》的 **知其然** 和 **知其所以然** 两个部分进行分别介绍。

小结

本篇和大家介绍了Flink的容错（Fault Tolerance）机制，本篇内容结合《Apache Flink漫谈系列 - State》一起查阅相信大家会对State和Fault Tolerance会有更好的理解，同时还可以观看《Apache Flink 知其然，知其所以然》课程中有关容错方面的内容。

欢迎扫码 孙金城 订阅号
更多Flink视频 👆