解读Flink中轻量级的异步快照机制--论文

本文根据论文Lightweight Asynchronous Snapshots for Distributed Dataflows
，通过这种轻量级的异步快照算法，解释Flink如何实现一致性快照以及恢复时如何实现exactly once的处理。

1、简介

对于分布式流处理系统而言，高吞吐、低延迟往往是最主要的需求。与此同时，容错在分布式系统中也很重要，对于正确性要求较高的场景，exactly once的实现往往也非常重要。

实时计算领域，往往低延迟、正确性是用户最关心的两个方面。

对于正确性，容错机制是必不可少的。当前分布式系统中保证exactly-once的处理，主要是通过对有状态的operator就行全局的、异性的快照完成的。但是这种方法通常有2个缺点：

```
1、为了获得全局的一致性状态，需要停止流处理程序，直到快照的完成，这显然对性能有着很大的影响
2、快照的内容包含传输过程中所有的内容，这导致快照的大小过大。
```

因此，一种新型的分布式快照的算法是即提供轻量级的快照，同时让快照发生时对系统的影响降到最低。这种算法不会停止流处理，因此是异步的，而且对于整个无环图的拓扑结构，只对有状态的operator进行快照，因此快照的大小也会占用很小的空间。

这里所说的新型的快照算法，既适用于有向无环图，也适用于有向有环图。本文重点关注在有向无环图中的应用，即在Apache Flink中的异步barrier快照（Asynchronous Barrier Snapshotting (ABS)）。

2、Apache Flink系统

简单来说，Flink就是一个分布式、有状态的批和流统一的流处理框架。每一个Flink的job都被编译为一个有向无环图（DAG）。在这个stream图中，每个点代表一个task，每个边代表task之间的数据传输。因此，每个operator既有输入也有输出（对于source而言，只有输出；对于sink而言，只有输入）。

2.1 流处理模型

Flink中的流处理被抽象为DataStream，DataStream可以由source产生，也可以由其他DataStream转化而来。每个DataStream上的操作可以包含filter、map、reduce等，同时，每个operator又可以并行执行。

我们这里以world count的例子来说明流处理中的快照如何运行。

先看这个例子：

```
val env : StreamExecutionEnvironment = ...
env.setParallelism(2)

val wordStream = env.readTextFile(path)
val countStream = wordStream.groupBy(_).count
countStream.print
```

Example 1: Incremental Word Count

注意：这个例子中groupby实际应该为keyBy，count应为sum。

这个job在client提交时，内部生成一张有向无环图（Execution Graph），如下：

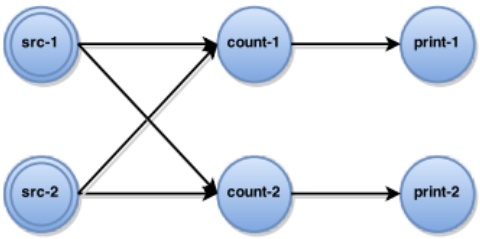

Figure 1: The execution graph for incremental word count

2.2 分布式数据流的执行

对于这个DAG图，我们对operator以及operator之间的数据传输做如下定义：

```
G = (T,E)
```

其中，G代表这个execution graph，定点T代表每个task（operator），边E代表2个task之间的数据通道（数据集合）。task又可以被细分为没有输入的source和没有输出的sink。

M代表每个task并行处理时的数据的集合。每个task t ∈ T由以下几部分组成：

```
1、输入、输出集合：It,Ot ⊆ E
2、这个operator中的状态st
3、用户自定义函数(UDF) ft
```

对于流入这个operator的每一条数据r ∈ M，通过UDF，产生一个新的状态值st'，同时产生一个新的输出的集合D ⊆ M。

比如这个例子中的count这个operator，它的输入集合It包含2个channel；每个key上的状态st记录了此key的count值，每来一个新的记录，这个st就会变为st'；st'是根据(UDF) ft的计算而来；最终产生一个output集合集合D。

3、Asynchronous Barrier Snapshotting（ABS）

3.1、问题定义

为了获得一致性的结果，分布式处理系统需要对task的失败要有弹性，即失败时可以恢复到一致性的结果。这种方法就是周期性的获得整个execution graph的全局快照，此快照要抓取所有必要的信息以备失败时恢复。所以，快照本身就是一个 execution graph G = (T,E)的子集t $G_* = (T_*,E_*)$。

对于一个快照G*而言，我们从最终性与可行性两个角度来阐述如何保障结果的正确性。

3.2、无环图的ABS

我们的方法是周期性的在source端注入特殊的barrier标记，此标记会跟随整个DAG最终流到sink端。

我们以下图来解释周期性的barrier如何起作用：

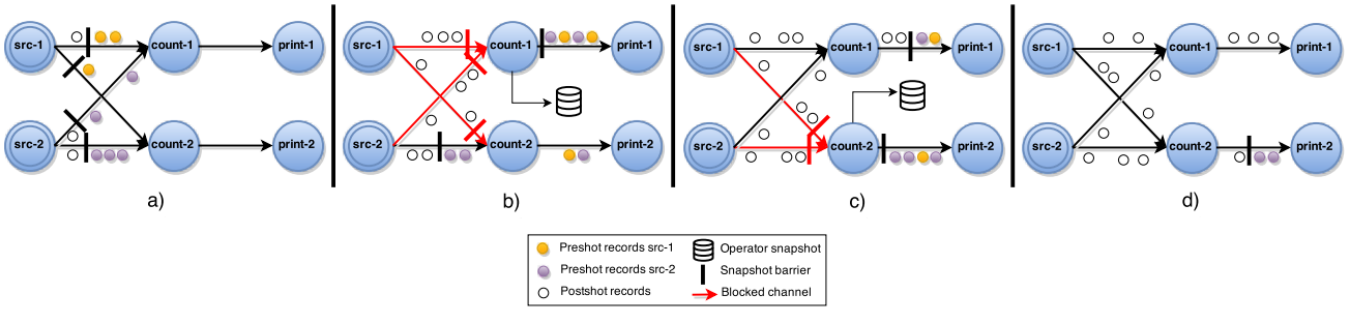

Figure 2: Asynchronous barrier snapshots for acyclic graphs


```
1、中央协调器（JobManager中）周期性的在source端注入barrier（黑色实线）。

2、当source端收到barrier后，立刻做一个快照，即记住当前的offset信息，然后将此barrier广

3、当中间的task收到其中一个输入端的barrier后，立刻阻塞这个channel；这个channel中被阻
```

```
6
7  4、当一个task接到它所有的input端的barrier后，立刻做一个快照，即记录当前这个operator中
8
9  5、最后，这个operator解除input channel的阻塞，继续后续的计算。直到最后的sink完成，才
```

我们将这个过程用下列的伪码表示：

Algorithm 1 Asynchronous Barrier Snapshotting for Acyclic Execution Graphs

```
1:  upon event ⟨Init | input_channels, output_channels, fun, init_state⟩ do
2:      state := init_state; blocked_inputs := ∅;
3:      inputs := input_channels;
4:      outputs := output_channels; udf := fun;
5:
6:  upon event ⟨receive | input, ⟨barrier⟩⟩ do
7:      if input ≠ Nil then
8:          blocked_inputs := blocked_inputs ∪ {input};
9:          trigger ⟨block | input⟩;
10:     if blocked_inputs = inputs then
11:         blocked_inputs := ∅;
12:         broadcast ⟨send | outputs, ⟨barrier⟩⟩;
13:         trigger ⟨snapshot | state⟩;
14:         for each inputs as input
15:             trigger ⟨unblock | input ⟩;
16:
17:
18: upon event ⟨receive | input, msg⟩ do
19:     {state′, out_records} := udf(msg, state);
20:     state := state′;
21:     for each out_records as {output, out_record}
22:         trigger ⟨send | output, out_record⟩;
23:
24:
```


我们再次简述一下这个过程：

```
1  检查点开始时，初始化输入集合，输出集合为空集，状态有初始值。当task接收到一个barrier时，
2  当task收到一个非barrier的数据时，根据udf更新状态的值，并输出到每一个output channel。
```

最终性如何保障：有向无环图中，barrier最终会按顺序流入到sink中；
可行性如何保障：根据FIFO的先进先出原则，barrier之前的记录能反映出每个operator的历史信

息。

4、失败恢复

```
(1) 从state Backend中拿到最后一份成功的快照；
(2) 还原备份记录；
(3) 从源端开始重新消费数据
```

为了达到exactly once语义的处理，我们从源端记录的offset开始重新消费数据，根据DAG图，流到下游的operator；先拿到快照中此operator的状态值，在此值基础上重新应用UDF进行计算。

5、性能影响

下图是一个拓扑结构：

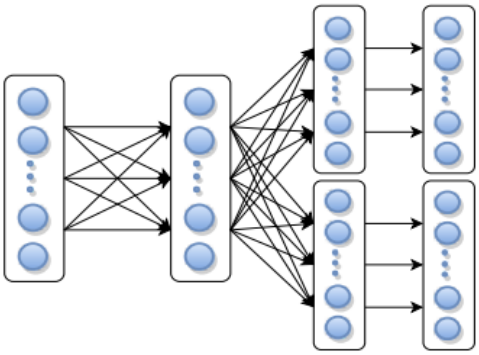

Figure 5: Execution topology used for evaluation

根据此DAG执行图，测试了ABS算法与同步快照算法对系统的影响：

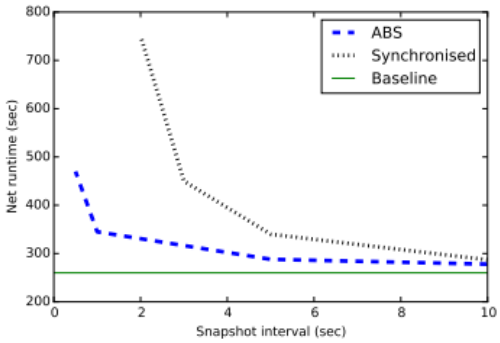

Figure 6: Runtime impact comparison on varying snapshot intervals

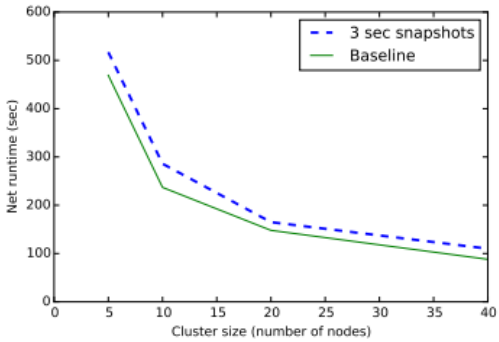

Figure 7: Runtime with ABS on different cluster sizes

6、总结

ABS快照的算法，从source端就开始做快照，到有状态的operator，最后是sink operator。这些operator中的UDF中的状态都被检查点所包含。

总结起来，ABS依赖能够重发的数据源以及有状态的operator实现。