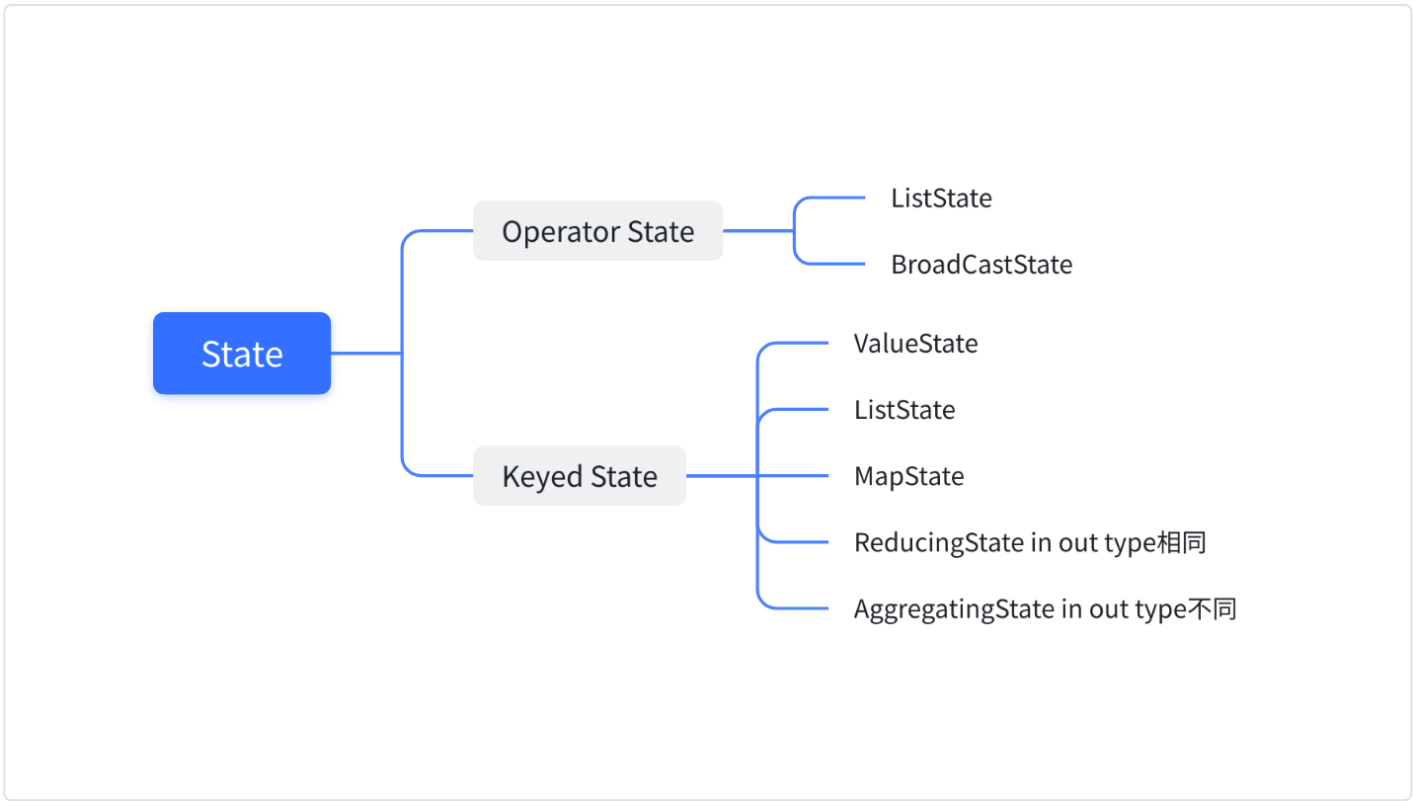
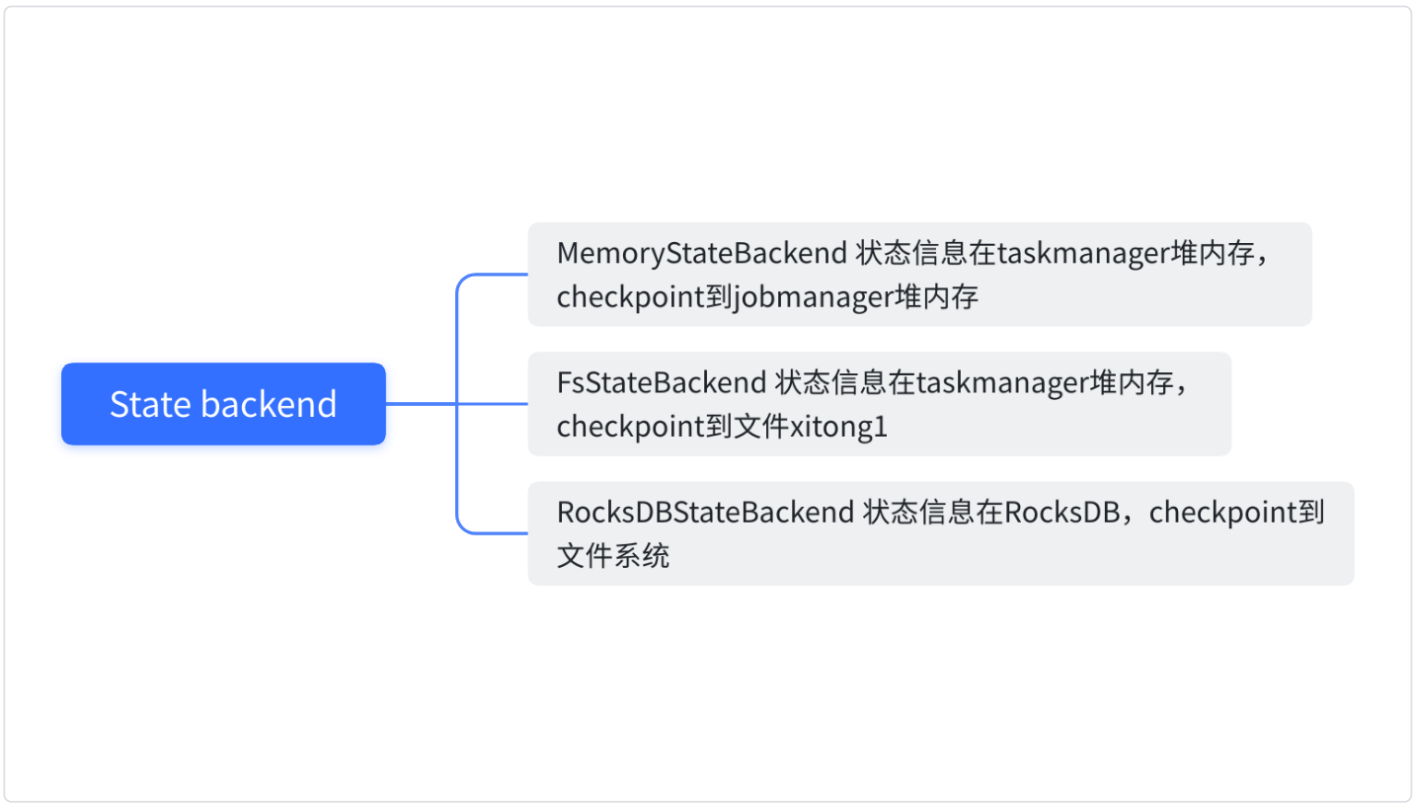
王卉自用Flink理论八股

State





Checkpoint 容错 and 一致性

Spark streaming的checkpoint提供不了精确一次处理， Flink可以， 如果Spark一定要实现的话靠方案

Chandy-Lamport

假设kafka分区2个，发送自然数数据源， 做sum odd/even

JobManager发起Checkpoint，插入barrier到source中流入到stream中，barrier第一次checkpoint放1，第二次放2，第三次放3，过一个operator后上报checkpoint（上报state状态到statebackend），然后再往下游发送数据，同时把barrier广播到下游operator，如果下游operator乱序，先缓存数据，等到barrier对齐后，重复

barrier发送完再去处理缓存数据

最后sink上报checkpoint，结束

Redis/Hbase保证Exactly once 幂等

多次写和一次写效果一样

Kafka保证Exactly once 两阶段提

事务的两个，要么一起成功，要么一起失败

Kafka sink，第一阶段预提交结果，数据提交到kafka，但是用户看不到，再进行checkpoint，如果checkpoint失败了，那么就相当于没有提交，checkpoint成功，预提交变成提交

一个很小的可能性，假设最终提交commit失败了，任务重启，第一个步骤就是继续原来的操作(Checkpoint的snapshot包含了整个application的状态，包括外部系统的pre-commited的external state)，第一件事就是进行commit，所以也没有问题

这些是flink内部控制的

如果是spark streaming得自己写代码

flink中两阶段提交是为了保证端到端的Exactly Once，主要依托checkpoint机制来实现，先看一下

checkpoint的整体流程，

1.JobManager会周期性的发送执行checkpoint命令(start checkpoint)；

2.当source端收到执行指令后会产生一条barrier消息插入到input消息队列中，当处理到barrier时

会执行本地checkpoint, 并且会将barrier发送到下一个节点，当checkpoint完成之后会发送一条ack信

息给JobManager;

3. 当所有节点都完成checkpoint之后，JobManager会收到来自所有节点的ack信息，那么就表示一次

完整的checkpoint的完成；

4. JobManager会给所有节点发送一条callback信息，表示通知checkpoint完成消息。接下来就可以

提交事务了

对比flink整个checkpoint机制调用流程可以发现与2PC非常相似，JobManager相当于协调者，flink提

供了CheckpointedFunction与CheckpointListener这样两个接口，CheckpointedFunction中有

snapshotState方法，每次checkpoint触发执行方法，通常会将缓存数据放入状态中，可以理解为是一个

hook，这个方法里面可以实现预提交，CheckpointListener中有notifyCheckpointComplete方法，

checkpoint完成之后的通知方法，这里可以做一些额外的操作，比如真正提交kafka的事务；在2PC中提到

如果对应流程2预提交失败，那么本 checkpoint就被取消不会执行，不会影响数据一致性.如果流程4失

败，那么重启的时候会再次执行commit

Checkpoint问题定位

Decline or Expire

Decline checkpoint 10423 by task 0b60f08bf8984085b59f8d9bc74ce2e1 of job 85d268e6fbc19411185f7e4868a44178. 其中 10423 是 checkpointID，0b60f08bf8984085b59f8d9bc74ce2e1 是 execution id，85d268e6fbc19411185f7e4868a44178 是 job id，我们可以在 jobmanager.log 中查找 execution id，找到被调度到哪个 taskmanager 上

从上面的日志我们知道该 execution 被调度到 hostnameABCDE 的 container_e24_1566836790522_8088_04_013155_1 slot 上，接下来我们就可以到 container container_e24_1566836790522_8088_04_013155 的 taskmanager.log 中查找Checkpoint 失败的具体原因了。

Expire

如果 Checkpoint 做的非常慢，超过了 timeout 还没有完成，则整个 Checkpoint 也会失败。当一个 Checkpoint 由于超时而失败是，会在 jobmanager.log 中看到如下的日志：

Checkpoint 1 of job 85d268e6fbc19411185f7e4868a44178 expired before completing.

Received late message **for** now expired checkpoint attempt 1 from 0b60f08bf8984085b59f8d9bc74ce2e1 of job 85d268e6fbc19411185f7e4868a44178.

找到对应的 taskmanager.log 查看具体信息。

Checkpoint 慢

Checkpoint 慢的情况如下：比如 Checkpoint interval 1 分钟，超时 10 分钟，Checkpoint 经常需要做 9 分钟（我们希望 1 分钟左右就能够做完），而且我们预期 state size 不是非常大。对于 Checkpoint 慢的情况，我们可以按照下面的顺序逐一检查。

- Source Trigger Checkpoint 慢
- 使用增量 Checkpoint RocksDB
- 作业存在反压或者数据倾斜
- Barrier 对齐慢
- 主线程太忙，导致没机会做 snapshot
- 同步阶段做的慢
- 异步阶段做的慢

反压问题排查

反压（backpressure）是实时计算应用开发中，特别是流式计算中，十分常见的问题。反压意味着数据管道中某个节点成为瓶颈，处理速率跟不上上游发送数据的速率，而需要对上游进行限速。由于实时计算应用通常使用消息队列来进行生产端和消费端的解耦，消费端数据源是 pull-based 的，所以反压通常是从某个节点传导至数据源并降低数据源（比如 Kafka consumer）的摄入速率。

要解决反压首先要做的是定位到造成反压的节点，这主要有两种办法：

1. 通过 Flink Web UI 自带的反压监控面板 SubTask
2. 通过 Flink Task Metrics

处理反压状态：

1. 该节点的发送速度跟不上产生速度，一条输入多条输出的，比如flatmap
2. 下游节点接收的慢

反压的根源节点并不一定会在反压面板体现出高反压，因为反压面板监控的是发送端，如果某个节点是性能瓶颈并不会导致它本身出现高反压，而是导致它的上游出现高反压。总体来看，如果我们找到第一个出现反压的节点，那么反压根源要么是就这个节点，要么是它紧接着的下游节点。

Task Metrics 是更好的反压监控手段 outPoolUsage, inPoolUsage

如果一个 Subtask 的发送端 Buffer 占用率很高，则表明它被下游反压限速了；如果一个 Subtask 的接受端 Buffer 占用很高，则表明它将反压传导至上游

outPoolUsage 和 inPoolUsage 同为低或同为高分别表明当前 Subtask 正常或处于被下游反压，这应该没有太多疑问。而比较有趣的是当 outPoolUsage 和 inPoolUsage 表现不同时，这可能是出于反压传导的中间状态或者表明该 Subtask就是反压的根源。

很多情况下的反压是由于数据倾斜造成的，这点我们可以通过 Web UI 各个 SubTask 的 Records Sent 和 Record Received 来确认，另外 Checkpoint detail 里不同 SubTask 的 State size 也是一个分析数据倾斜的有用指标。

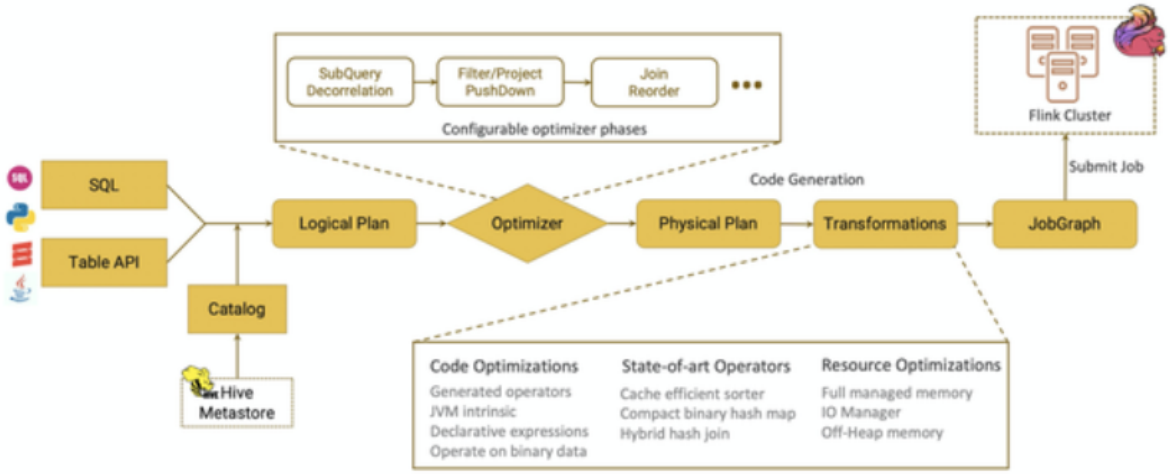
最常见的问题可能是用户代码的执行效率问题（频繁被阻塞或者性能问题）。最有用的办法就是对 TaskManager 进行 CPU profile，从中我们可以分析到Task Thread 是否跑满一个 CPU 核：如果是的话要分析 CPU 主要花费在哪些函数里面，比如我们生产环境中就偶尔遇到卡在 Regex 的用户函数（ReDoS）；如果不是的话要看 Task Thread 阻塞在哪里，可能是用户函数本身有些同步的调用，可能是checkpoint 或者 GC 等系统活动导致的暂时系统暂停。

另外 TaskManager 的内存以及 GC 问题也可能会导致反压，包括 TaskManager JVM 各区内存不合理导致的频繁 Full GC 甚至失联。推荐可以通过给TaskManager 启用 G1 垃圾回收器来优化 GC，并加上 -XX:+PrintGCDetails 来打印 GC 日志的方式来观察 GC 的问题。

Time

Event time, Ingestion time, Process time

SQL


SubQuery Decorrelation
Filter/Project PushDown
Join Reorder
Configurable optimizer phases
Flink Cluster
Submit Job
Code Generation
SQL
Table API
Logical Plan
Optimizer
Physical Plan
Transformations
JobGraph
Catalog
Hive Metastore
Code Optimizations
Generated operators
JVM intrinsic
Declarative expressions
Operate on binary data
State-of-art Operators
Cache efficient sorter
Compact binary hash map
Hybrid hash join
Resource Optimizations
Full managed memory
IO Manager
Off-Heap memory