# 背景

rocksdb将并发的多个写memtable请求，打包成一个write group，每个write group具有唯一的leader和多个follower（注意，这里的leader/follower，是指多个写memtable的并发线程之间的角色，跟raft协议中的leader/follower完全是不同且无关的两个概念）。

由leader负责将write group中的数据先写入WAL，然后再由所有leader和follower一起，并发地将各自数据写入memtable。即，整个过程分为两个步骤：i. leader写WAL；ii. 所有成员并发写memtable。

而不同的write group之间，不妨记为wg1和wg2，仅需要保证他们之间第一步写WAL是严格保持串行的。当wg1写完WAL（即完成了它的第一步）后，wg2就可以开始执行了，无需等待wg2完成并发写memtable（即第二步）。也即，当wg1还在并发写memtable时，wg2已经可以开始写WAL了。相当于两个“工序”，形成一道流水线。

**这里再做一个概念上的澄清：write group（组提交）和 pipeline write（流水线写）**

1、write group（组提交）是指将并发的写请求打包成一个write group的过程。仅由leader来处理整个write group的写WAL逻辑，从而保证写WAL是串行执行的。

2、pipeline write（流水线写）则是指，不同的write group之间，以流水线的形式来执行各自的写WAL和并发写memtable两个步骤。

**rocksdb DBOoptions中的相关参数：**

enable\_pipelined\_write – 若为true，则开启pipeline write（流水线写）；否则，wg2需要等待wg1完成写memtable之后，才能开始执行。默认打开。

allow\_concurrent\_memtable\_write – 若为true，所有成员是否并发写memtable；否则，则由leader将整个write group中的数据串行写入memtable。默认打开。由于memtbale默认使用skip list实现，所以即使并发乱序写入，仍能保持有序。

如下图所示：

a. enable\_pipelined\_write = true, allow\_concurrent\_memtable\_write = true.

网站

中度可信度描述已自动生成

b. enable\_pipelined\_write = true, allow\_concurrent\_memtable\_write = false.

图形用户界面, 网站

描述已自动生成

c. enable\_pipelined\_write = false, allow\_concurrent\_memtable\_write = true.

图形用户界面

描述已自动生成

d. enable\_pipelined\_write = false, allow\_concurrent\_memtable\_write = false.

图片包含 图形用户界面

描述已自动生成

# 原理/RocksDB pipeline write

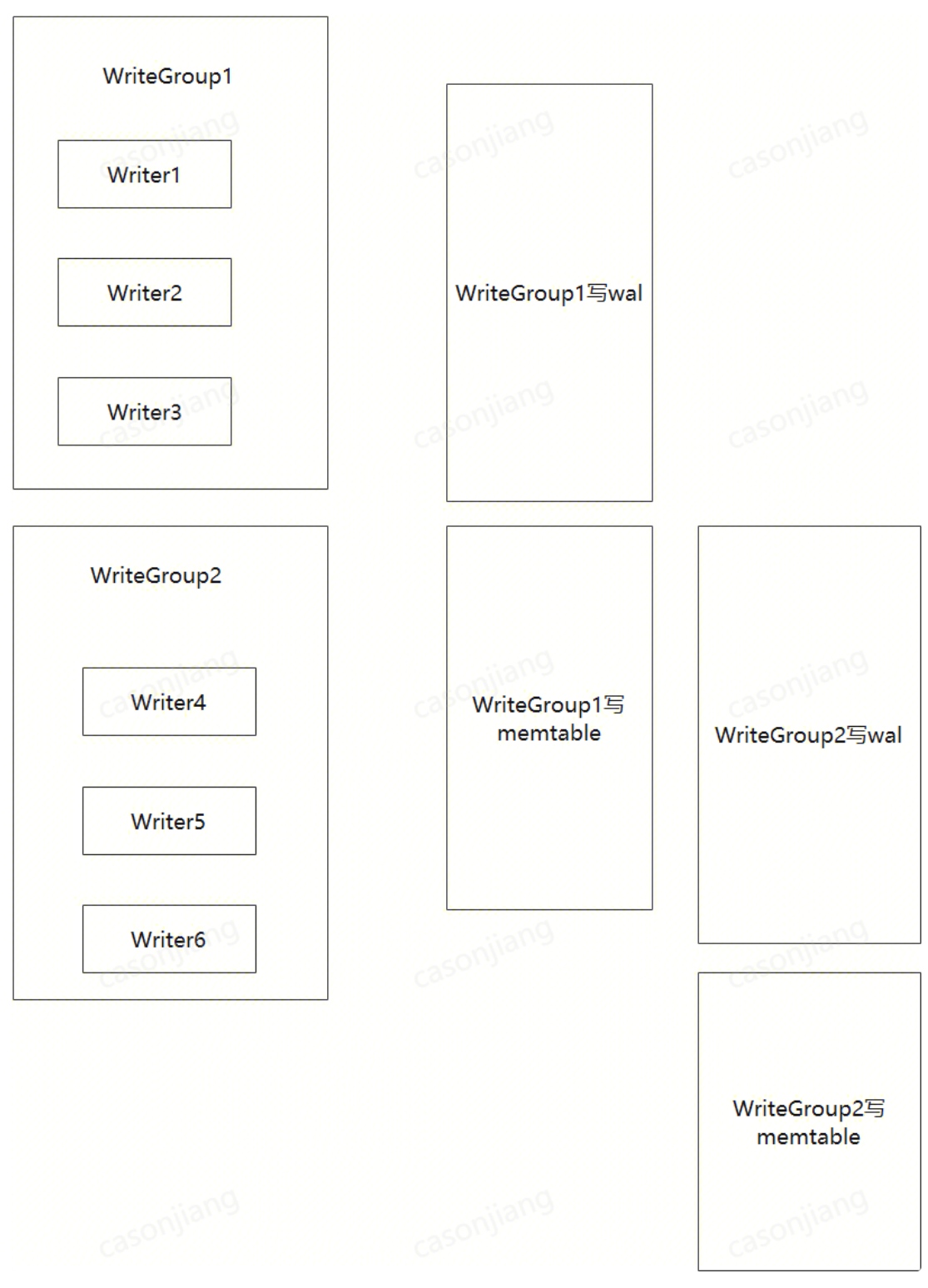
pipeline write是rocksdb的一个特性，可以通过 Options.enable\_pipelined\_write=true 打开该特性。根据rocksdb的官方文档，该配置与memtable concurrent wrtier一起使用，可以提升20%的性能。

pipeline write就是把并发的写请求，通过打包，pipeline执行，提升了部分性能。

pipeline write 从功能上描述，它把多个并发的 write请求，把它们的wal日志打包成一个大的WriteBatch写入，写入成功后，通过处于同一个WriteGroup的线程开始把自己的WriteBatch写入metmable。

pipeline可以这样理解，上一个BatchGroup写完wal，下一个WriteGroup就也可以开始写wal，这个操作是可以与上一个Group写memtable并发的。多线程把写入请求打包成WriteGroup，而WriteGroup的两个操作写wal与写memtable是可以pipeline进行的。

下图所示：WriteGroup1 的写memable 与WriteGroup2的写wal日志，本身是可以并发的。



流水线会负责把多个写请求打包成一个WriteGroup，把这些写请求的wal日志打包成连续的wal日志一起写下去，然后负责通过其它线程，把各自的数据插入到metmable当中。

流水线本身也实现了switch metmable与 metmable insert两种请求的barrier

本身也兼顾了stalling write机制。

## 关键对象

WriteBatch 每一个写请求（commit, prepare等一些需要写wal或者写memtable的请求），都会带自己的WriteBatch，它里面记录了事务写入的key value数据。

WriteThread::Writer： 代码一个写请求，在并发纬度，也代表当前写请求所在的线程。多个Writer会形成一个双向链表。

WriteThread 每一个DB只有一个object，写了多线程（多个写请求）并发过程中如何实现pipeline写入的。

WriteThread ::WriteGroup 多个顺序的 WriteThread::Writer 的打包，顺序按照 WriteThread::Writer 调用 WriteThread::JoinBatchGroup 的先后顺序（严格意义上应该是指 WriteThread::Writer 链表的顺序）

## 流水线工作的样例

下图展示了一个打包wal的样例， Writer1 把Writer1 Writer2 Writer3 打包成一个WriteGroup。

1. 初使状态，当前的DB还没有任务一个写请求进来。

② 第一个写请求(Writer1)进来，因为 newest\_writer\_ 为空，Writer1（它所在的线程）被委派成了 STATE\_GROUP\_LEADER 角色，负责把多个writer的wal日志进行打包，并写入。 而 newest\_writer\_ 指向了 Writer1

③ 第二个写请求进来(Writer2)，直接通过link\_older指针挂到Writer1的后面，因为当前已经有STATE\_GROUP\_LEADER ，所以它不会成STATE\_GROUP\_LEADER ,等待被委派其它角色

④ 第三个写请求进来(Writer3)，直接通过link\_older指针挂到Writer1的后面，因为当前已经有STATE\_GROUP\_LEADER ，所以它不会成STATE\_GROUP\_LEADER ,等待被委派其它角色

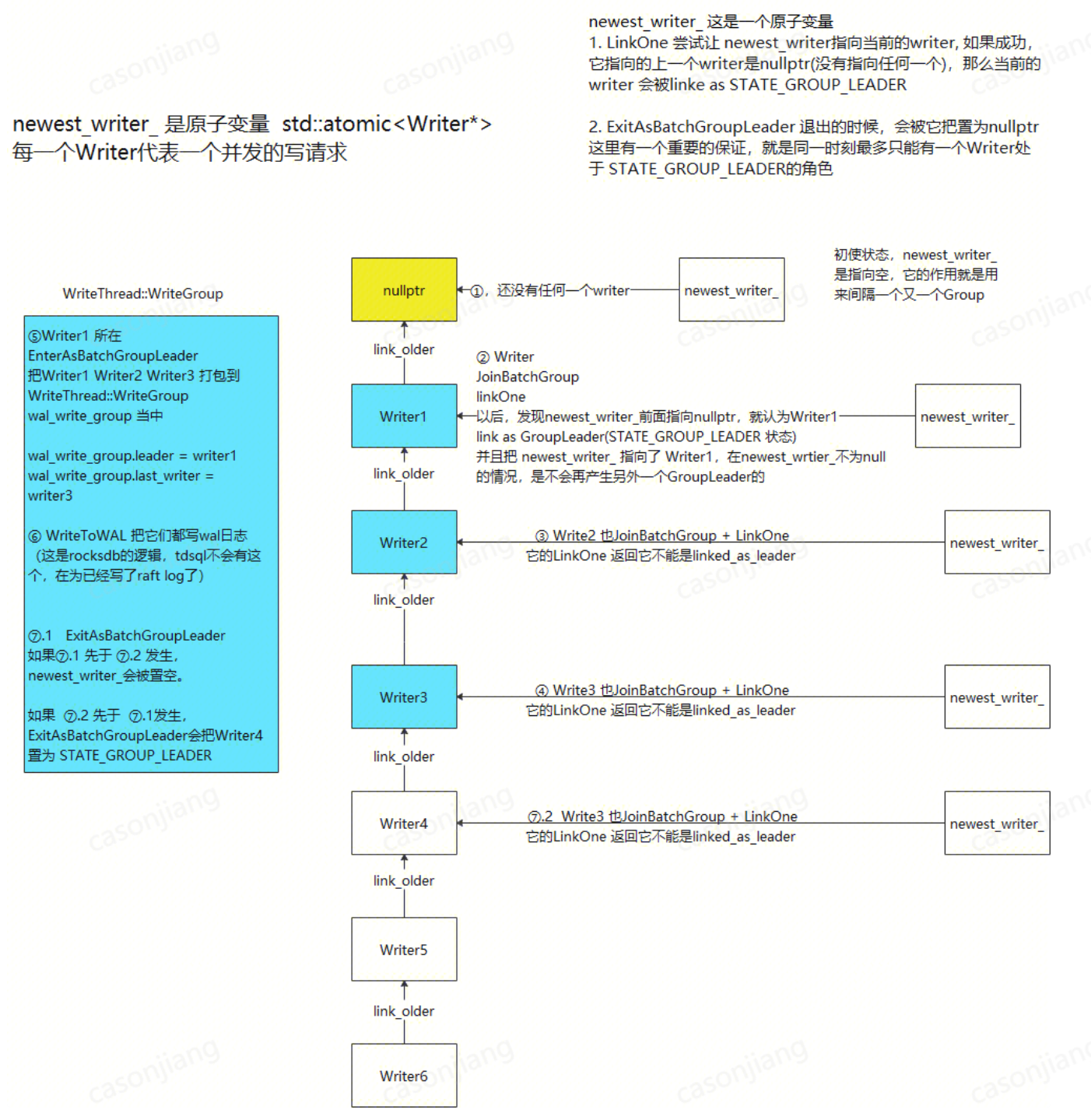
⑤ Wrtier1 所在EnterAsBatchGroupLeader，把Writer1 Writer2 Write3 要写入的数据打包成一个WriteGroup，叫做wal\_write\_group, WriteGroup的leader 指向 Writer1, WriteGroup的last\_writer\_ 指向Writer3

⑥ Writer1 所在的线程，把wal\_write\_group写入wal 日志（rocksdb的流程，tdsql 3.0 不需要）

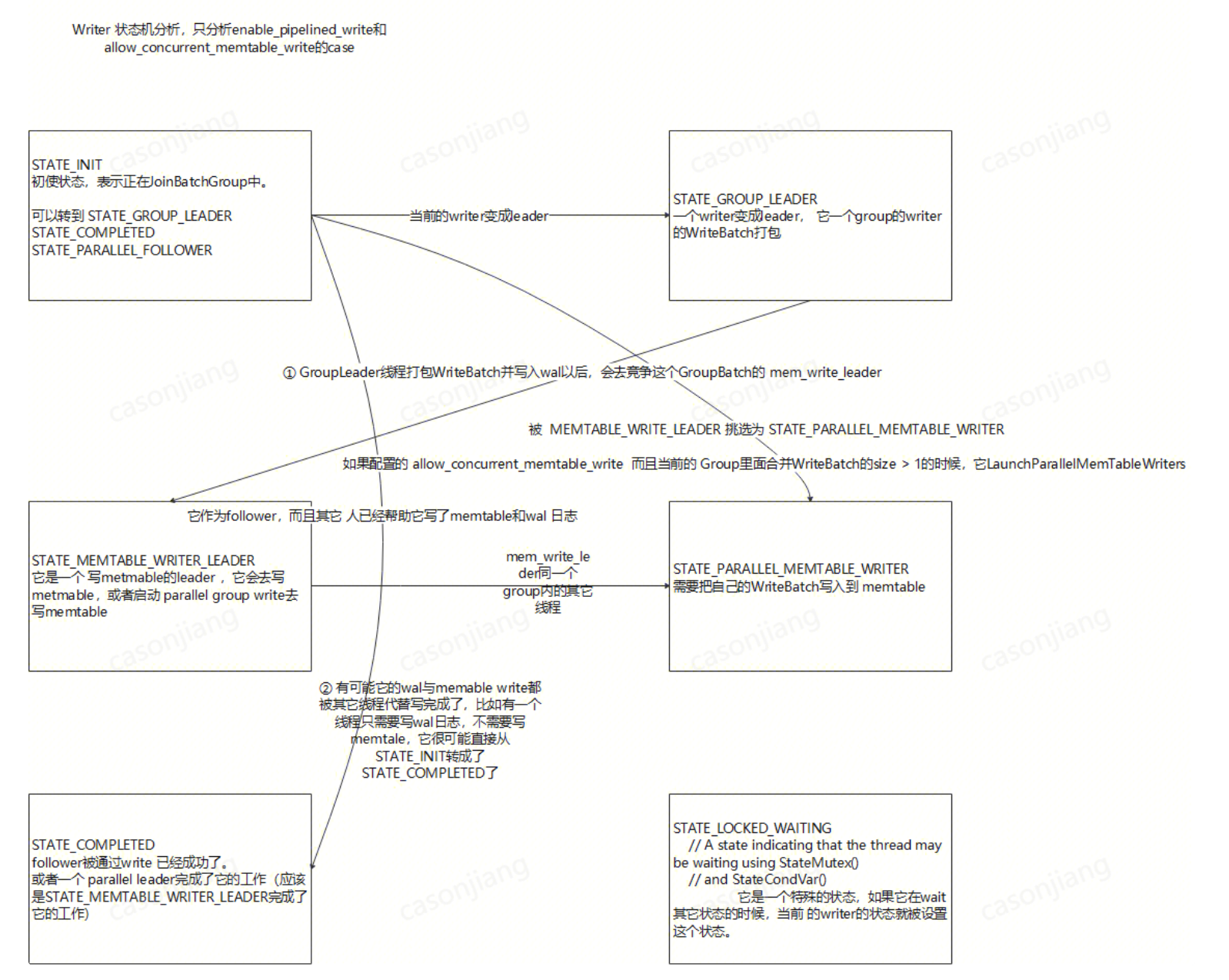
⑦Writer1 ExitAsBatchGroupLeader

case1: 如果这个过程中，没有新来的write请求，也就是说 ⑦.1 先于 ⑦.2 发生， 那么 newest\_writer\_ 会被置为nullptr，Writer4 在linkOne的时候，发现newest\_writer\_ 为nullptr，也会像Writer1 一样，被委派成 STATE\_GROUP\_LEADER 角色

case2: 如果 ⑦.2 先于 ⑦.1发生，ExitAsBatchGroupLeader会把Writer4 置为 STATE\_GROUP\_LEADER



## Writer角色的状态机转换



这里简单的介绍一个状态机转换：

1、当一个writer调用 JoinBatchGroup的时候， 如果当前WriteThread没有WalGroupLeader，那么writer就会被委派为WalGroupLeader的角色，对应的状态机是STATE\_GROUP\_LEADER，它负责按照writer链表的顺序，把挂在它后面的writer打印成一个WriterGroup，并把它们的wal日志写入到磁盘

2、当一个Writer调用JoinBatchGroup的时候，如果当前已经WriteThread 已经有了WalGroupLeader，它就进入到 STATE\_LOCKED\_WAITING状态，等待被设置成其它状态。

3、当一个writer把打包好的WriteGroup的wal写入成功后，会调用WriteThread::ExitAsBatchGroupLeader， 让出来当前 WalGroupLeader的角色，并且尝试把当前writer切换成 MemWriteLeader (STATE\_MEMTABLE\_WRITER\_LEADER)，正常情况下都能切换成功，也可能失败，因为可能有其它操作抢占了当前的 MemWriteLeader(比如 WriteThread::WaitForMemTableWriters 会设置memtable write的barrier，它可能会抢了MemWriteLeader)，如果抢占失败就 调用 AwaitState 进入 wait状态。

4、当一个writer处于 MemWriteLeader 角色，它会把跟他处于同一个WriteGroup的其它writer唤醒，被唤醒状态的writer处于 STATE\_PARALLEL\_MEMTABLE\_WRITER 状态，这些writer会把各自的WriteBatch写入metmable，写入成功以后调用 CompleteParallelMemTableWriter， 表明当前的writer已经成功的完成了请求。 但是处于 STATE\_PARALLEL\_MEMTABLE\_WRITER 状态的writer要等同一个WriteGroup的所有writer都成功了，才会把状态切换成 STATE\_COMPLETED

## 关键代码分析

Class WriteThread

// Points to the newest pending writer. Only leader can remove

// elements, adding can be done lock-free by anybody.

std::atomic<Writer\*> newest\_writer\_;

WriteThread::newest\_writer\_ 指向了writer 链表中最后一个 Writer .

// Points to the newest pending memtable writer. Used only when pipelined

// write is enabled.

std::atomic<Writer\*> newest\_memtable\_writer\_;

WriteThread::newest\_memtable\_writer\_ 代表了最新的MemWriteLeader

WriteThread::LinkOne : 把一个writer挂到 writer 链表后面， 返回true，表明抢到了WalWriteLeader或者MemWriteLeader，

WriteThread::LinkGroup ： WalWriteLeader 再进行一次打包（实际上打包的writer跟 写wal日志打包的writer的对象是一样的）

详细代码参见 write\_thread.cc ，细节太多，这里不详述。

## memtable writer 与 memtable switch的barrier

从LSM的算法来看, metmable switch是不允许与metmable insert并发的。这一章节分析pipeline 如何保证两者无法并发。

### DBImpl::SwitchMemtable

DBImpl::SwitchMemtable 它本身是在流水中被调用的（流水线的GroupLeader角色才会调用它）。下面是一个调用关系。处于这个状态，当前的WalWriteGroup 还没有确定，上一个GroupLeader一定已经退出，当前GroupLeader还没有选择出来MemWriteLeader。上一个MemWriteLeader可能还存在。

rocksdb::DBImpl::SwitchMemtable(ColumnFamilyData \*, rocksdb::DBImpl::WriteContext \*) : Status

rocksdb::DBImpl::HandleWriteBufferFull(rocksdb::DBImpl::WriteContext \*) : Status

rocksdb::DBImpl::PreprocessWrite(const WriteOptions &, bool \*, rocksdb::DBImpl::WriteContext \*) : Status

rocksdb::DBImpl::PipelinedWriteImpl(const WriteOptions &, WriteBatch \*, WriteCallback \*, uint64\_t \*, uint64\_t, bool, uint64\_t \*) : Status

rocksdb::DBImpl::TDPipelinedWriteImpl(const WriteOptions &, WriteBatch \*, WriteCallback \*) : Status

rocksdb::DBImpl::TDWriteImpl(const WriteOptions &, WriteBatch \*, WriteCallback \*) : Status

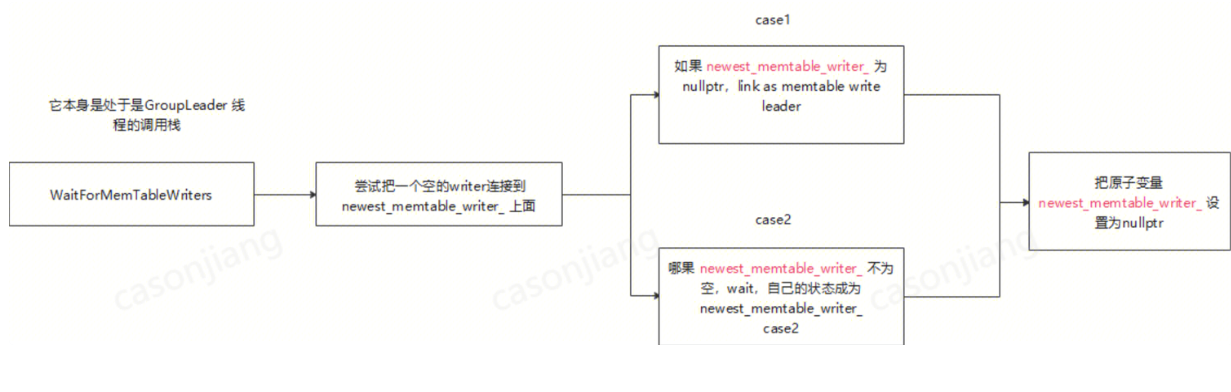
rocksdb::DBImpl::WriteImpl(const WriteOptions &, WriteBatch \*, WriteCallback \*, uint64\_t \*, uint64\_t, bool, uint64\_t \*, size\_t, PreReleaseCallback \*) : Status



1、DBImpl::SwitchMemtable 通过调用 WaitForMemTableWriters，放入一个空的MemWriteLeader，这个空的MemWriteLeader实际上就是一个barrier，能够等上一次MemWriteLeader已经退出。

2、第2步保证了上一次的MemWriteLeader已经退出，而当前的调用栈又是处于GroupLeader还未启动下一个MemWriteLeader的时候，所以这时可以保证没有并发的操作在写metmable。

### WaitForMemTableWriters 函数分析



这里参考 LinkGroup的代码，大概可以这样理解：

1、newest\_memtable\_writer\_为nullptr，说明当前没有正在执行的metmable insert操作。

2、而 WaitForMemTableWriters 本身的调用栈就在pipeline的GroupLeader当中，阻塞了后来的write request进来。

那么 WaitForMemTableWriters 就是简单的抢一下 memtable\_writer\_leader ，就能够保证上一个WriteGroup的 metmable writer状态的线程都不再写 memtable了。

case1: 本来上一个WriteGroup的metmable write都已经退出了

case2: 可能还有上一个WriteGroup正在执行的metmable write 任务，那就等它执行完。

所以WaitForMemTableWriters 达到了他的效果，调用它之后，已经没有正在写metmable的任务正在执行

## stalling write

如果待compaction的数据太多的时候，会触发stalling write，表示compaction的速度已经跟不写入的速度，LSM树的形状处于一个不好的状态，读性能会严重下降。stalling write就是在待compaction的数据太多的时候，降低写入的速度。

* 在流水线的process\_write(调用栈在GroupLeader状态)，调用BeginWriteStall， 它会向Writer队列里面插入一个write\_stall\_dummy\_, 并且把writer队列还没有 进行Group 的wrtier遍历一下，看一看是不是有no\_slow\_down的writer，如果有，让他们先报错返回。
* 后续来的Writer在linkOne函数中，发现最后一个writer是write\_stall\_dummy\_， 就wait到一个 stall\_cv\_一个变量上去。
* process\_write 在BeginWriteStall后，sleep一段时间（这个时间是计算出来的，计算的方法参见WriteController::GetDelay）。然后调用EndWriteStall, 把这个write\_stall\_dummy\_,从writer queue中去掉，并且singal，通过第2步wait的writer

# 方案

## region cf 去除写WAL

system region cf 和 user region cf 所写入的数据，是通过raft协议来保证一致性的，因此它们无需再写rocksdb WAL。相关的修改如下：

对应rocksdb原生的WriteImpl和PipelinedWriteImpl，我们相应实现了TDWriteImpl和TDPipelinedWriteImpl，去掉了WriteImpl和PipelinedWriteImpl中的写WAL逻辑。

TDWriteImpl和TDPipelinedWriteImpl中，修改了seqno的获取的方式，由TDInsertInto函数从WriteBatch中获取，即由mc下发的commit\_ts，而不是rocksdb原生的本地自增值。

由于 system region cf 和 user region cf 与其他 cf 生成seqno的逻辑不同（其他cf仍然走rocksdb原生逻辑，需要写WAL），需要避免两者的写请求被分配到同一个write group中。因此增加了is\_tdstore\_region\_writer\_，将system region cf和user region cf的写请求单独形成write group。

## raft follower并发回放，去除组提交逻辑（不再形成write group）

raft follower并发回放的详细设计文档：raft log并行回放

简而言之，为了提升raft follower回放性能，我们对raft log进行并发乱序回放。而并发乱序回放的特点，导致无法直接使用TDWriteImpl/TDPipelinedWriteImpl来向memtable写入数据。以一个例子简单说明：

log1{raft\_log\_index=1, key='a', seqno=10} 和 log2{raft\_log\_index=2, key='a', seqno=20} 并发回放。由于乱序，log2先执行，写入了memtable，然后在log1执行时，在PreprocessWrite阶段发现memtable已经满了，触发switch memtable。从而导致log1与log2被分开到了两个memtable，而且log1反而处于更新的memtable，读取key='a'时反而读到了log1中的数据，正确性被打破了。

因此并发乱序回放必须保证，发生switch memtable时，已经发射的raft log必须都已经写入到了当前的memtable中。

也就是说，这个例子中，log1调用TDPipelinedWriteImpl，触发了switch memtable的话，必须要等到它自己写入memtable之后才可以switch，陷入了一种”我等我自己“的状态。

因此在并发回放设计方案中，为了打破这一情况，让follower并发回放请求不再走pipeline write（TDPipelinedWriteImpl）流程，不再形成write group，而是直接去并发写memtable。（注：skip list实现的memtable支持并发写入，且能保证有序性）

另外实现了一个 ParallelReplayControl 类，对已发射进入并发回放队列的raft log进行计数，当每次触发switch memtable时，都要先等待已发射进入并发回访队列的raft log执行完毕。

## raft leader并发apply

将raft follower回放日志改为并发乱序执行之后，raft leader apply日志成为了性能瓶颈，这是因为单 region的leader在raft log提交后，是串行apply的这些raft log。

因此对raft leader appy的场景，也引入并发乱序执行的逻辑。简而言之，不同region上的raft log可以并行apply，同一region上的不同事务raft log之间可以并行apply（有key冲突的事务之间的互斥机制是通过事务层的memory lock来保证的）。

相比于raft follower的并发回放，raft leader并发apply的改动要更加简单。主要需要考虑的点是：由于日志乱序执行，raft index更大的日志可能先于raft index小的日志执行完毕，因此当save snapshot时，需要有额外的机制来保证snapshot index之前的日志都已经执行完毕。

主要改动不涉及对rocksdb原生逻辑，并发apply仍然走pipeline write流程（TDWriteImpl/TDPipelinedWriteImpl）。

# 总结

tdstore对于rocksdb原生的写memtable逻辑，主要做了以下几点改动：

新实现了TDWriteImpl/TDPipelinedImpl。对比原生的WriteImpl/PipelinedWriteImpl，去除了write group leader写WAL的逻辑。

新实现了TDConcurrentWriteImpl，用于raft follower并发乱序回放日志。对比原生的WriteImpl/PipelinedWriteImpl，去除了写WAL、write group、pipeline write的逻辑，直接将write batch中的数据并发写入memtable。

新实现了TDInsertInto，用于TDWriteImpl/TDPipelinedImpl、TDConcurrentWriteImpl。与原生的InsertInto相比，使用write batch中的sequence来写memtable，即mc下发的事务commit\_ts，而非InsertInto使用传入的rocksdb本地自增sequence。

新实现了ParallelReplayControl类，负责raft follower并发乱序回放日志场景下，遇到switch memtable时，作为memtbale barrier，确保已经发射的日志回放完毕。

不同cf向memtable写入数据的总结：

data\_db - system region cf和user region cf：

raft leader apply 事务日志 – TDWriteImpl/TDPipelinedImpl

raft follower replay 事务日志 – TDConcurrentWriteImpl

data\_db - local\_metadata cf：

write fence、destroy replica job metainfo等 – 走rocksdb 原生的Put/Delete逻辑，即WriteImpl/PipelinedWriteImpl，需要写WAL

multi\_raft\_db、monitor\_info\_db：

走rocksdb 原生的Put/Delete逻辑，即WriteImpl/PipelinedWriteImpl，需要写WAL