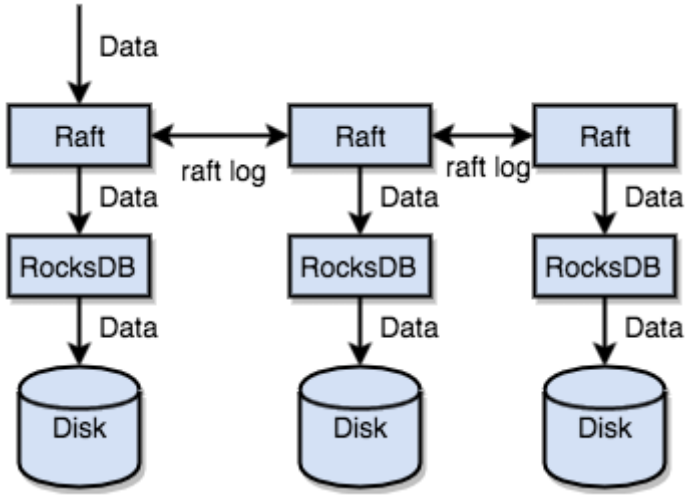
零声教育出品 Mark 老师 QQ : 2548898954

Raft 共识算法

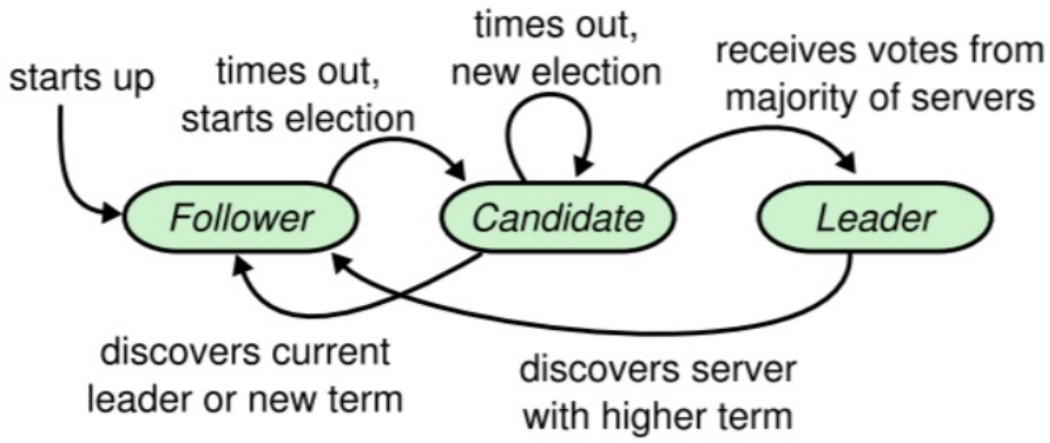
> 分布式一致性算法中比较有代表性的有 Raft 和 Paxos；其中 ZooKeeper 基于 Paxos（zab paxos 的变种）；Raft 共识算法提供了几个重要的功能：
>
> - Leader（主副本） 选举
> - 成员变更（如添加副本、删除副本、转移 Leader 等操作）
> - 日志复制
>
> TiKV 利用 Raft 来做数据复制，每个数据变更都会落地为一条 Raft 日志，通过 Raft 的日志复制功能，将数据安全可靠地同步到复制组的每一个节点中。不过在实际写入中，根据 Raft 的协议，只需要同步复制到大多数节点，即可安全地认为数据写入成功。



Region

> TiKV 可以看做是一个巨大的有序的 KV Map，那么为了实现存储的水平扩展，数据将被分散在多台机器上。
>
> 对于一个 KV 系统，将数据分散在多台机器上有两种比较典型的方案：
>
> - Hash：按照 Key 做 Hash，根据 Hash 值选择对应的存储节点。
> - Range：按照 Key 分 Range，某一段连续的 Key 都保存在一个存储节点上。
>
> TiKV 选择了第二种方式，将整个 Key-Value 空间分成很多段，每一段是一系列连续的 Key，将每一段叫做一个 Region，并且会尽量保持每个 Region 中保存的数据不超过一定的大小，目前在 TiKV 中默认是 96MB。每一个 Region 都可以用 [StartKey，EndKey) 这样一个左闭右开区间来描述。
>
> 将数据划分成 Region 后，TiKV 将会做两件重要的事情：
>
> - 以 Region 为单位，将数据分散在集群中所有的节点上，并且尽量保证每个节点上服务的 Region 数量差不多。这是 PD Server 的主要工作；
> - 以 Region 为单位做 Raft 的复制和成员管理。复制过程中 Replica 之间是通过 Raft 来保持数据的一致，一个 Region 的多个 Replica 会保存在不同的节点上，构成一个 Raft Group。其中一个 Replica 会作为这个 Group 的 Leader，其他的 Replica 作为 Follower。默认情况下，所有的读和写都是通过 Leader 进行，读操作在 Leader 上即可完成，而写操作再由 Leader 复制给 Follower。

Raft 算法详解



> [运行机理]
>
> 关于 Raft 算法几个问题：
>
> - 为什么是大多数？
>
> 只能选举出一个 leader；
> - Raft 有哪些流程？
>
> 主要包含 leader 选举以及日志复制；
> - Raft 选举过程中有哪些对象？它们之间状态如何转换？
>
> 主要对象为：follower、candidate、leader；
>
> follower 当选举超时时间过期，则自动成为 candidate；
>
> candidate 会主动给自己投票；转而向其他节点发送拉票广播；当自身选票超过半数以上将成为 leader；
>
> leader 会定时向其他节点同步自身的变化信息；如果 leader 发现有比自身更高的选举任期，则自己马上下台成为 follower，并接收新的 leader 的数据变更同步；
>
> 每个人只有一张选票，只有成为 candidate 才能给自己投票；如果自己是 follower 并且收到其他 candidate 的拉票，那么会给第一个给自己拉票的候选者投票，此时会重置自身的选举超时；
> - Raft 怎么成为候选者？
>
> 每个节点通过各自随机选举超时，先到达选举超时的节点先成为候选者；
> - 选举超时时间过期会做哪些操作？
>
> 1. 将选举任期加一；
> 2. 重新随机一个选举超时（避免出现有相同的选举超时节点，从而避免平票的情况产生）；
> 3. 成为 candidate；
> 4. 如果之前是平票，在 candicate 状态下选举超时，则此时还会将上轮选举的选票重置；
> - follower 接收到其他节点发送的信息，会怎么处理？
>
> 如果收到 candidate 的拉票，那么 follower 会向第一次接收到的拉票所属 candidate 发送投票，并将自己的任期设置为该 candidate 的任期；同时会重置自身选举超时和心跳超时；
>
> 思考：
>
> - 详述日志复制流程？
> - Raft 在脑裂下如何工作的？

- 修复脑裂后，Raft 如何恢复一致性?
- Raft 下读写是如何工作的?

分布式事务

TiDB 中分布式事务采用的是 Percolator 的模型；Percolator 是 Google 在 OSDI 2010 的一篇 论文 中提出的在一个分布式 KV 系统上构建分布式事务的模型，其本质上还是一个标准的 2PC（2 Phase Commit），2PC 是一个经典的分布式事务的算法。但是 2PC 一般来说最大的问题是事务管理器（Transaction Manager）。在分布式的场景下，有可能会出现第一阶段后某个参与者与协调者的连接中断，此时这个参与者并不清楚这个事务到底最终是提交了还是被回滚了，因为理论上来说，协调者在第一阶段结束后，如果确认收到所有参与者都已经将数据落盘，那么即可标注这个事务提交成功。然后进入第二阶段，但是第二阶段如果某参与者没有收到 COMMIT 消息，那么在这个参与者复活以后，它需要到一个地方去确认本地这个事务后来到底有没有成功被提交，此时就需要事务管理器的介入。这个事务管理器在整个系统中是个单点，即使参与者，协调者都可以扩展，但是事务管理器需要原子的维护事务的提交和回滚状态。

Percolator 模型的写事务流程：

每当事务开始，协调者（在 TiDB 内部的 tikv-client 充当这个角色）会从 PD leader 上获取一个 timestamp，然后使用这个 ts 作为标记这个事务的唯一 id。标准的 Percolator 模型采用的是乐观事务模型，在提交之前，会收集所有参与修改的行（key-value pairs），从里面随机选一行，作为这个事务的 Primary row，剩下的行自动作为 secondary rows，这里注意，primary 是随机的，具体是哪行完全不重要，primary 的唯一意义就是负责标记这个事务的完成状态。在选出 Primary row 后， 开始走正常的两阶段提交，第一阶段是上锁+写入新的版本，所谓的上锁，其实就是写一个 lock key。

Percolator 示例

比如一个事务操作 A、B、C，3 行。在数据库中的原始 Layout 如下：

```
Key          |           Value
-----------------------------------------
A_ver3       | Value of A at Versioin 3
A_ver2       | Value of A at Versioin 2
B_ver1       | Value of B at Versioin 1
C_ver3       | Value of C at Versioin 3
```

假设我们这个事务要 `Update (A, B, C, Version 4)`，第一阶段，我们选出的 Primary row 是 A，那么第一阶段后，数据库的 Layout 会变成：

```
Key       |         Value
-------------------------------------
A_Lock    | Primary lock at Version 4   <-------------------------+
A_ver4    | Value of A at Version 4                               |
A_ver3    | Value of A at Version 3                               |
A_ver2    | Value of A at Version 2                               |
B_Lock    | Secondary lock, primary lock is A at Version 4  ------+
B_ver4    | Value of B at Version 4                               |
B_ver1    | Value of B at Version 1                               |
C_Lock    | Secondary lock, primary lock is A at Version 4  ------+
C_ver4    | Value of C at Version 4
C_ver3    | Value of B at Version 3
```

> 上面这个只是一个释义图，实际在 TiKV 做了一些优化，但是原理上是相通的。上图中标红色的是在第一阶段中在数据库中新写入的数据，可以注意到，`A_Lock`、`B_Lock`、`C_Lock` 这几个就是所谓的锁，大家看到 B 和 C 的锁的内容其实就是存储了这个事务的 Primary lock 是谁。在 2PC 的第二阶段，标志事务是否提交成功的关键就是对 Primary lock 的处理，如果提交 Primary row 完成（写入新版本的提交记录+清除 Primary lock），那么表示这个事务完成，反之就是失败，对于 Secondary rows 的清理不需要关心，可以异步做（为什么不需要关心这个问题？）。利用了原子性；
>
> Percolator 是采用了一种化整为零的思路，将集中化的事务状态信息分散在每一行的数据中（每个事务的 Primary row 里），对于未决的情况，只需要通过 lock 的信息，顺藤摸瓜找到 Primary row 上就能确定这个事务的状态。

两阶段提交

> 二阶段提交是将事务的提交过程分成了两个阶段来进行处理；目的是使分布式系统架构下的所有节点在进行事务处理过程中能够保持原子性和一致性；二阶段提交能够非常方便地完成所有分布式事务参与者的协调，统一决定事务的提交或回滚；

- 阶段一：提交事务请求
    1. 事务询问

       向所有的参与者发送事务内容，询问是否可以执行事务提交操作，并开始等待各参与者的响应；
    2. 执行事务

       各参与者节点执行事务操作，并将 “undo” 和 “redo” 信息记录在事务日志中；
    3. 各参与者向协调者反馈事务询问的响应

       执行成功，返回 yes 响应；执行失败，返回 no 响应；
- 阶段二：执行事务提交（根据反馈决定是否进行事务提交）
    - 执行事务提交（全是 yes 响应）
        1. 发送提交请求

           协调者向所有参与者节点发出 Commit 请求；
        2. 事务提交

           参与者收到 Commit 请求后，会正式执行事务提交操作，并在提交操作之后释放在整个事务执行期间占用的资源；
        3. 反馈事务提交结果

           参与者完成事务后，向协调者发送 ack 消息；
        4. 完成事务

           协调者接收到所有参与者反馈的 ack 消息后，完成事务；
    - 中断事务（只要有一个 no 响应)
        1. 发送回滚请求

           协调者向所有参与者节点发出 Rollback 请求；
        2. 事务回滚

           参与者接收到 Rollback 请求后，会利用 “Undo” 信息执行事务回滚，并在回滚后释放整个事务执行期间占用的资源；
        3. 反馈事务回滚结果

           参与者在完成事务回滚之后，向协调者发送 ack 消息；

4. 中断事务

协调者接收到所有参与者反馈的 ack 消息后，完成事务中断；

> 总的来说，二阶段提交将一个事务的处理过程分为了投票和执行两个阶段；核心是对每个事务都采用先尝试后提交的处理方式。

隔离级别

> 最早的 ANSI SQL-92 提出了至今为止仍然是应用最广的隔离级别定义，读提交、可重复读、可序列化。但是 「A Critique of ANSI SQL Isolation Levels」这篇文章指出了 ANSI SQL-92 的缺陷，并对其做出了补充。「Generalized Isolation Level Definitions」这篇文章，指出了此前对隔离级别定义重度依赖数据库的实现，并且提出了与实现无关的隔离级别定义。
>
> 隔离级别定义的是数据库事务间的隔离程度；
>
> ANSI SQL-92 提出了最经典的隔离级别定义如下：

隔离级别	脏读	不可重复读	幻读
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

> ANSI SQL-92 所给出的隔离级别的定义被广泛使用，但也造成了今天隔离级别指代混乱的现象，其原因在于这一套定义是不够严谨的；「A Critique of ANSI SQL Isolation Levels」这篇文章指出了 ANSI SQL-92 所遗漏的一些问题，同时针对 ANSI SQL-92 的隔离级别在数据库实现之下提出了更高的要求，最后，这篇文章给出了 **Snapshot Isolation 的隔离级别**。

Isolation level	**P0** Dirty Write	**P1** Dirty Read	**P4C** Cursor Lost Update	**P4** Lost Update	**P2** Fuzzy Read	**P3** Phantom	**A5A** Read Skew	**A5B** Write Skew
READ UNCOMMITTED == Degree 1	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
READ COMMITTED == Degree 2	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL SERIALIZABLE == Degree 3 == Repeatable Read Date, IBM, Tandem, ...	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

Snapshot Isolation

> 1995年Hal Berenson等人在《A critique of ANSI SQL Isolation levels》中提出了 Snapshot Isolation 的概念。

- 事务的读操作从 Committed 快照中读取数据，快照时间可以是事务的第一次读操作之前的任意时间，记为StartTimestamp；
- 事务准备提交时，获取一个 CommitTimestamp，它需要比现存的 StartTimestamp 和 CommitTimestamp 都大；

- 事务提交时进行冲突检查，如果没有其他事务在 [StartTS, CommitTS] 区间内提交了与自己的 WriteSet 有交集的数据，则本事务可以提交；这里阻止了 Lost Update 异常；
- SI 允许事务用很旧的 StartTS 来执行，从而不被任何的写操作阻塞，或者读一个历史数据；当然，如果用一个很旧的 CommitTS 提交，大概率是会 Abort 的；

Write Skew（写偏序）

Txn1	Txn2
`r(x, 10)`	
	`r(y, 20)`
`w(y, 10)`	
	`w(x, 20)`
`commit`	`commit`
`r(x, 20)`	
`r(y, 10)`	

> Write Skew 是两个事务在写操作上发生的异常，上图表示了 Write Skew 现象，即 T1 尝试把 x 的值赋给 y，T2 尝试把 y 的值赋给 x，如果这两个事务 Serializable 的执行，那么在结束之后 x 和 y 应该拥有一样的值，但是在 Write Skew 中，并发操作使得他们的值互换了。

MVCC

> 设想这样的场景：两个客户端同时去修改一个 Key 的 Value，如果没有数据的多版本控制，就需要对数据上锁，在分布式场景下，可能会带来性能以及死锁问题。TiKV 的 MVCC 实现是通过在 Key 后面添加版本号来实现。

```
Key1_Version3 -> Value
Key1_Version2 -> Value
Key1_Version1 -> Value
……
Key2_Version4 -> Value
Key2_Version3 -> Value
Key2_Version2 -> Value
Key2_Version1 -> Value
……
KeyN_Version2 -> Value
KeyN_Version1 -> Value
```

> 注意，对于同一个 Key 的多个版本，版本号较大的会被放在前面，版本号小的会被放在后面，这样当用户通过一个 `Key + Version` 来获取 Value 的时候，可以通过 Key 和 Version 构造出 MVCC 的 Key，也就是 Key_Version。然后可以直接通过 RocksDB 的 `SeekPrefix(Key_Version)` API，定位到第一个大于等于这个 Key_Version 的位置。

列式存储 TiFlash

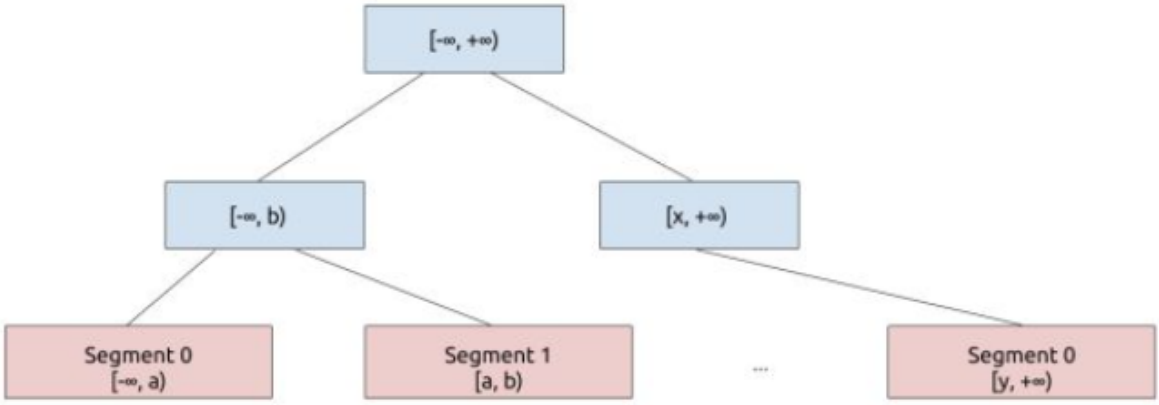
> TiDB 是一款分布式 HTAP 数据库，它目前有两种存储节点，分别是 TiKV 和 TiFlash。TiKV 采用了行式存储，更适合 TP 类型的业务；而 TiFlash 采用列式存储，擅长 AP 类型的业务。TiFlash 通过 Raft 协议从 TiKV 节点实时同步数据，拥有毫秒级别的延迟，以及非常优秀的数据分析性能。它支持实时同步 TiKV 的数据更新，以及支持在线 DDL。我们把 TiFlash 作为 Raft Learner 融合进 TiDB 的 raft 体系，将两种节点整合在一个数据库集群中，上层统一通过 TiDB 节点查询，使得 TiDB 成为一款真正的 HTAP 数据库。
>
> TiFlash 研发了新的列存引擎 Delta Tree。它可以在支持高 TPS 写入的同时，仍然能保持良好的读性能；

整体架构

> Delta Tree 的架构设计充分参考了 B+ Tree 和 LSM Tree 的设计思想。从整体上看，Delta Tree 将表数据按照主键进行 range 分区，切分后的数据块称为 Segment；然后 Segment 内部则采用了类似 LSM Tree 的分层结构。分区是为了减少每个区的数据量，降低复杂度。

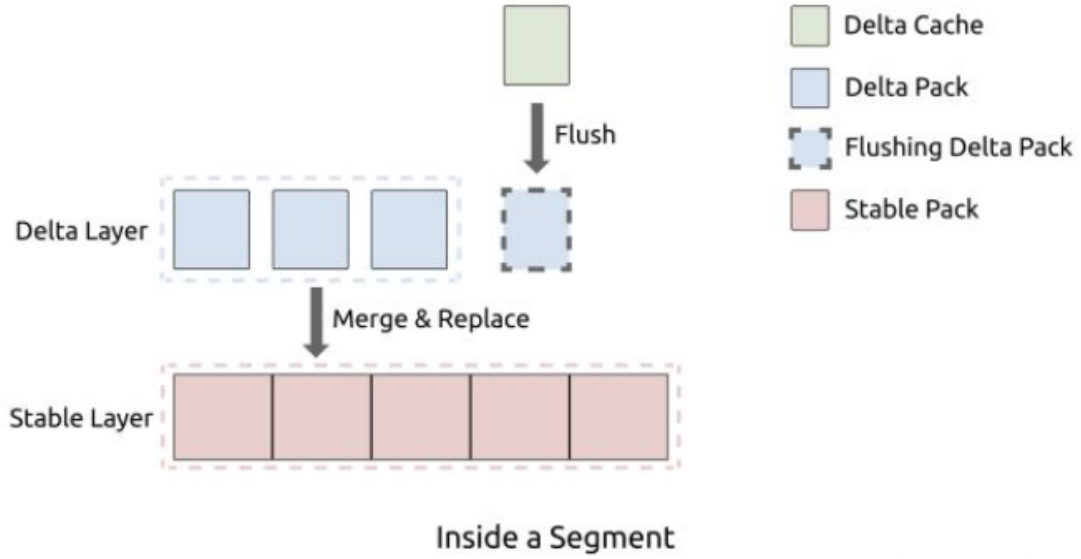
Segment

> Segment 的切分粒度通常在150 万行左右，远超传统 B+ Tree 的 Leaf Node 的大小。Segment 的数量在一台机器上通常在 10 万以内，所以我们可以把 Segment 的元信息完全放在内存，这简化了工程实现的复杂度。和 B+ Tree 的叶子节点一样，Segment 支持 Split、Merge。在初始状态，一张表只存在一个 range 为 [-∞, +∞) 的 Segment。



Levels LSM-Tree

> 在 Segment 内部，通过类似 LSM Tree 的分层的方式组织数据。 因为 Segment 的数据量相对于其他 LSM Tree 实现要小的多，所以 Delta Tree 只需要固定的两层，即 Delta Layer 和 Stable Layer，分别对应 LSM Tree 的 L0 和 L1。我们知道对于 LSM Tree 来说，层数越少，写放大越小。默认配置下，Delta Tree 的理论写放大（不考虑压缩）约为 19 倍。因为列式存储连续存储相同类型的数据，天然对压缩算法更加友好，在生产环境下，Delta Tree 引擎常见的实际写放大低于 5 倍。



Inside a Segment

Pack

> Segment 内部的数据管理单位是 Pack，通常一个 Pack 包含 8K 行或者更多的数据。关系型数据库的 schema 由多个列定义组成，每个列定义包括 column name，column id，column type 和 default value 等。由于支持 DDL，比如加列、删列、改数据类型等操作，所以不同的 Pack schema 有可能是不一样的。Pack 的数据也由列数据（column data）组成，每个列数据其实就是一维数组。Pack 除了主键列 Primary Keys(PK) 以及 schema 包含的列之外，还额外包含 version 列和 del_mark 列。version 就是事务的 commit 时间戳，通过它来实现 MVCC。del_mark 是布尔类型，表明这一行是否已经被删除。
>
> 将 Segment 数据分割成 Pack 的作用是，可以以 Pack 为 IO 单位和索引过滤单位。在数据分析场景，从存储引擎获取数据的方式都是 Scan。为了提高 Scan 的性能，通常我们的 IO 块要比较大，所以每次读 IO 可以读一个或者多个 Pack 的数据。另外通常在分析场景，传统的行级别的精确索引通常用处不大，但是我们仍然可以实现一些简单的粗糙索引，比如 Min-Max 索引，这类索引的过滤单位也是 Pack。
>
> 在 TiDB 的架构中，TiKV 的数据是以 Region 为调度单位，Region 是数据以 range 切分出来的虚拟数据块。而 Delta Tree 的 Pack 内部的数据是以 `(PK, version)` 组合字段按升序排序的，与 TiKV 内的数据顺序一致。这样可以让 TiFlash 无缝接入 TiDB 集群，复用原来的 Region 调度机制。

Delta Layer

> Delta Layer 相当于 LSM Tree 的 L0，它可以认为是对 Segment 的增量更新，所以命名为 Delta。与 LSM Tree 的 MemTable 类似，最新的数据会首先被写入一个称为 Delta Cache 的数据结构，当写满之后会被刷入磁盘上的 Delta Layer。而当 Delta Layer 写满之后，会与 Stable Layer 做一次 Merge（这个动作称为 Delta Merge）得到新的 Stable Layer。

Stable Layer

> Stable Layer 相当于 LSM Tree 的 L1，是存放 Segment 的大部分数据的地方。它由一个不可修改的文件存储，称为 DTFile。一个 Segment 只有一个 DTFile。Stable Layer 同样由 Pack 组成，并且数据以 `(PK, version)` 组合字段按升序排序。不一样的是，Stable Layer 中的数据是全局有序，而 Delta Layer 则只保证 Pack 内有序。原因很简单，因为 Delta Layer 的数据是从 Delta Cache 写下去的，各个 Pack 之间会有重复数据；而 Stable Layer 的数据则经过了 Delta Merge 动作的整理，可以实现全局有序。

> 当 Segment 的总数据量超过配置的容量上限，就会以 Segment range 的中点为分裂点，进行 split，分裂成两个 Segment；如果相邻的两个 Segment 都很小，则可能会被 merge 在一起，变成一个 Segment。

Delta Cache

> 为了缓解高频写入的 IOPS 压力，我们在 Delta Tree 的 Delta Layer 中设计了内存 cache，称为 Delta Cache。更新会先写入 Delta Cache，写满之后才会被 flush 到磁盘。而批量写入是不需要写 Delta Cache 的，这种类型的数据会被直接写入磁盘。Delta Tree 并没有在写入数据的之前写 WAL；而是充分利用 TiDB 中使用的 Raft 协议；在 Raft 协议中，任何一个更新，会先写入 Raft log，等待大多数副本确认之后，才会将这个更新应用到存储引擎；在这里 TiFlash 直接利用了 Raft log 实现 WAL，即在 flush 之后才更新 raft log applied index。