Etcd关键内容总结

数据模型

etcd 被设计为能够可靠地存储不频繁更新的数据，并提供可靠的事件监听查询功能。etcd 提供对键值对的先前版本的访问支持，以便实现低成本的快照和监听历史事件（即“时间旅行查询”）。持久化、多版本、并发控制的数据模型非常适合这些使用场景。

etcd 将数据存储在一个多版本的持久化键值存储中。当某个键值对的值被新数据覆盖时，持久化键值存储会保留该键值对的旧版本。这种键值存储实际上是不可变的；其操作不会就地更新数据结构，而是始终生成一个新的更新后的结构。修改后，所有过去的键版本仍然可以被访问和监听。为了防止数据存储随着时间推移无限增长以及维护过多的旧版本，存储可以通过压缩操作去除最旧的被覆盖数据版本。

逻辑视图

存储的逻辑视图是一个扁平的二进制键空间。该键空间在字节字符串键上具有按字典顺序排序的索引，因此范围查询的成本很低。

键空间维护多个版本（修订版）。当存储被创建时，初始修订版为 1。每个原子性修改操作（例如，事务操作可能包含多个子操作）都会在键空间上生成一个新的修订版。所有由先前修订版持有的数据保持不变。旧版本的键仍然可以通过之前的修订版访问。同样，修订版也进行了索引化处理；通过监听器对修订版进行范围遍历是高效的。如果存储为了节省空间而被压缩，那么在压缩修订版之前的修订版将会被移除。在整个集群生命周期内，修订版是单调递增的。

一个键的生命周期跨越了一代，从创建到删除。每个键可能有一代或多代。创建一个键会增加该键的版本号，如果该键在当前修订版中不存在，则版本号从 1 开始。删除一个键会生成一个键的墓碑（tombstone），通过将版本号重置为 0 来结束该键的当前一代。每次修改一个键都会增加其版本号，因此版本号在一个键的生命周期内是单调递增的。一旦发生压缩操作，任何在压缩修订版之前结束的一代都将被移除，并且除了最新值外，在压缩修订版之前设置的所有值也将被移除。

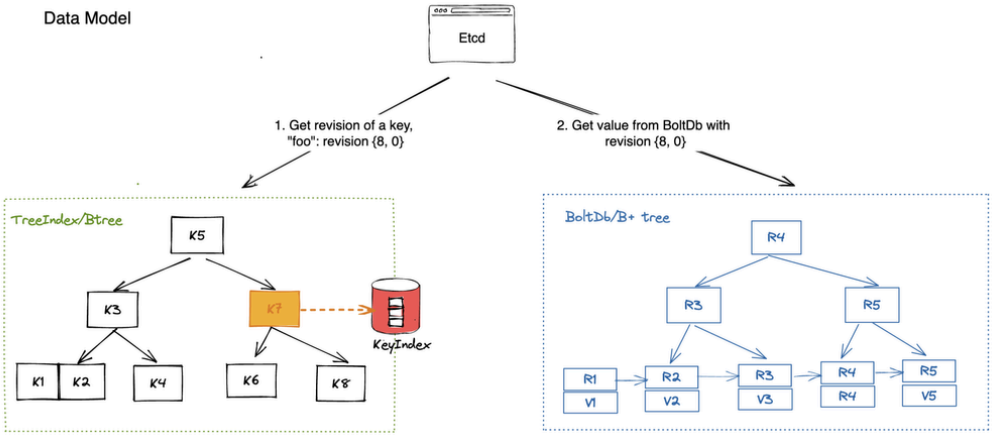
物理视图

etcd 将物理数据以键值对的形式存储在持久化的 B+ 树中。存储状态的每个修订版仅包含相对于其前一个修订版的增量变化，以提高效率。单个修订版可能对应树中的多个键。

键值对的键是一个 3 元组 `(major, sub, type)`。`major` 表示持有该键的存储修订版。`sub` 用于区分同一修订版内的不同键。`type` 是一个可选的后缀，用于标识特殊值（例如，如果值包含墓碑，则类型为 `t`）。键值对的值包含了相对于前一个修订版的修改内容，因此表示一个修订版增量。B+ 树按照键的字典序排列。基于修订版增量的范围查找速度很快，这使得能够快速找到从一个特定修订版到另一个修订版之间的修改内容。压缩操作会移除过时的键值对。

此外，etcd 还维护了一个二级内存中的 B 树索引，以加速对用户暴露的键的范围查询。B 树索引中的键是存储暴露给用户的键，而值是指向持久化 B+ 树中修改内容的指针。压缩操作会移除无效的指针。

总体而言，etcd 首先从 B 树中获取修订版信息，然后使用该修订版作为键从 B+ 树中检索值（如下所示）。



raft算法

日志复制

一旦领导者被选举出来，它便开始处理客户端请求。每个客户端请求包含一条需要由复制状态机执行的命令。领导者将该命令作为新的条目追加到其日志中，然后并行向其他服务器发送 **AppendEntries RPC** 请求以复制该条目。当该条目被安全地复制后（如下文所述），领导者将其应用到自己的状态机，并将执行结果返回给客户端。如果追随者发生崩溃、运行缓慢，或者网络数据包丢失，领导者会无限重试 **AppendEntries RPC** 请求（即使它已经向客户端返回了响应），直到所有追随者最终存储了所有的日志条目。

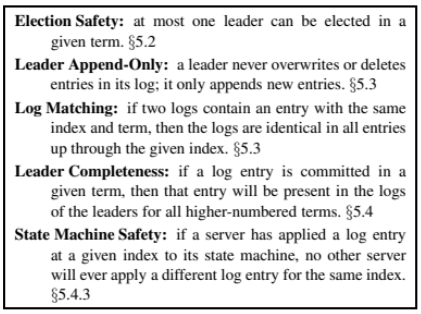


图3

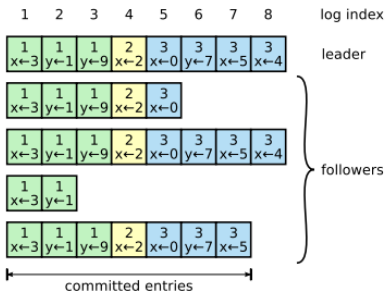


图6

日志的组织方式如图 6 所示。每个日志条目存储了一个状态机命令以及该条目被领导者接收时对应的任期编号。日志条目中的任期编号用于检测日志之间的一致性差异，并确保实现图 3 中的一些特性。此外，每个日志条目还包含一个整数索引，用于标识其在日志中的位置。

领导者决定何时可以安全地将日志条目应用到状态机；这样的条目被称为已提交（committed）。Raft 保证已提交的条目是持久化的，并且最终会被所有可用的状态机执行。当创建该条目的领导者将其复制到大多数服务器上时，日志条目即被视为已提交（例如，图 6 中的条目 7）。这同时也会提交领导者日志中所有之前的条目，包括由先前领导者创建的条目。领导者会跟踪它所知道的最高的已提交索引，并在未来的**AppendEntries RPC** 请求中（包括心跳消息）包含该索引，以便其他服务器最终也能得知。一旦追随者得知某个日志条目已被提交，它就会按照日志顺序将该条目应用到其本地状态机。

Raft 的日志机制，以在不同服务器的日志之间保持高度的一致性。这不仅简化了系统的行为并使其更加可预测，而且是确保安全性的重要组成部分。Raft 维护了以下特性，这些特性共同构成了图 3 中的 **日志匹配属性（Log Matching Property）**：

- 如果两个不同日志中的条目具有相同的索引和任期号，那么它们存储的是相同的命令。
- 如果两个不同日志中的条目具有相同的索引和任期号，那么这两个日志在所有之前的条目中都是完全相同的。

第一条性质源于以下事实：在一个特定的任期内，领导者在给定的日志索引位置最多创建一个条目，并且日志条目在其日志中的位置永远不会改变。第二条性质由 **AppendEntries** 操作执行的一个简单一致性检查来保证。当发送 **AppendEntries RPC** 时，领导者会包含其日志中紧邻新条目前的那个条目的索引和任期号。如果追随者在其日志中找不到具有相同索引和任期号的条目，则它会拒绝这些新条目。这种一致性检查充当了一个归纳步骤：日志的初始空状态满足 **日志匹配属性（Log Matching Property）**，而一致性检查在扩展日志时保留了该属性。因此，每当 **AppendEntries** 成功返回时，领导者就知道追随者的日志在新条目范围内与其自身的日志完全一致。

在正常运行期间，领导者和追随者的日志保持一致，因此 **AppendEntries** 的一致性检查永远不会失败。然而，领导者的崩溃可能会导致日志不一致（旧的领导者可能尚未将其日志中的所有条目完全复制到追随者）。这些不一致性可能会在一个系列的领导者和追随者崩溃过程中累积。图 7 展示了追随者的日志可能与新领导者日志不同的几种方式。追随者的日志可能缺少领导者日志中存在的条目，或者包含领导者日志中不存在的额外条目，或者两者兼有。日志中的缺失条目和多余条目可能跨越多个任期。

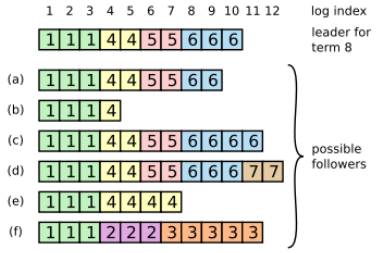


图7

在 Raft 中，领导者通过强制追随者的日志与其自身的日志保持一致来处理不一致性。这意味着追随者日志中与领导者日志冲突的条目将被领导者日志中的条目覆盖。当结合一个额外的限制条件时，这种方法是安全的。

为了让追随者的日志与其自身保持一致，领导者必须找到两个日志最后达成一致的条目位置，删除追随者日志中该位置之后的所有条目，并将领导者日志中该位置之后的所有条目发送给追随者。所有这些操作都是在 **AppendEntries RPC** 执行的一致性检查过程中完成的。领导者为每个追随者维护一个 `nextIndex`，表示即将发送给该追随者的下一个日志条目的索引。当一个新的领导者当选时，它会将所有 `nextIndex` 值初始化为自身日志最后一个条目索引的下一个位置（如图 7 中的索引 11）。如果追随者的日志与领导者的日志不一致，那么在下一次 **AppendEntries RPC** 中，一致性检查将会失败。在遭到拒绝后，领导者会递减 `nextIndex` 并重新尝试 **AppendEntries RPC**。最终，`nextIndex` 将回退到一个位置，在这个位置上领导者和追随者的日志是一致的。当这种情况发生时，**AppendEntries** 将成功执行，这会移除追随者日志中所有冲突的条目，并追加领导者日志中的条目（如果有）。一旦 **AppendEntries** 成功，追随者的日志就与领导者的日志保持了一致，并且在整个任期内都将维持这种一致性。

如果需要，可以对协议进行优化以减少被拒绝的 **AppendEntries RPC** 的次数。例如，在拒绝一个 **AppendEntries** 请求时，追随者可以包含冲突条目的任期号以及该任期中它存储的第一个条目的索引。通过这些信息，领导者可以递减 `nextIndex`，从而跳过该任期中所有冲突的条目；这样，每个存在冲突条目的任期只需要一次 **AppendEntries RPC**，而不是每个条目都需要一次 RPC。在实际应用中，这种优化不是必要的，因为故障发生的频率很低，并且不太可能出现大量不一致的条目。

通过这种机制，领导者在当选时无需采取任何特殊措施来恢复日志的一致性。它只需开始正常的操作，日志会在 **AppendEntries** 一致性检查失败时自动收敛。领导者从不会覆盖或删除其自身日志中的条目（如图 3 中的 **Leader Append-Only Property** 所述）。

这种日志复制机制展现了第 2 节中描述的理想的共识特性：只要大多数服务器处于运行状态，Raft 就能够接受、复制和应用新的日志条目；在正常情况下，一个新的条目可以通过向集群中的大多数节点发送一轮 RPC 来完成复制；并且单个缓慢的追随者不会影响整体性能。

持久化存储文件

bbolt b+tree: member/snap/db

该文件包含主 etcd 内容，应用到了 Raft 日志的特定位置（参见 `consistent_index`）。

物理组织结构

更好的 Bolt 存储在物理上以 B+ 树的形式组织。B 树的物理页**永远不会被原地修改**。相反，内容会被复制到一个新页（从空闲页列表中分配），并且一旦没有可能访问该页的打开事务，旧页将立即被添加回空闲页列表。通过这一过程，打开的只读（RO）事务能够看到存储的一致历史状态。读写（RW）事务是独占的，并会阻塞所有其他读写事务。

大值会被存储在多个连续的页上。页回收的过程结合对分配不同大小连续页区域的需求，可能会导致 Bolt 存储的碎片化逐渐增加。

Bolt 文件不会自行缩小。只有在碎片整理过程中，文件才会被重写为一个新的文件，该文件在其末尾具有一些空闲页缓冲区，并且文件大小被截断以减少占用空间。

逻辑组织结构

Bolt 存储被划分为多个 **bucket**（桶）。在每个桶中，以字典序存储键值对（键为字节数组 `byte[]`，值也为字节数组 `byte[]`）。以下列表展示了 etcd（截至 3.5 版本）使用的桶及其包含的键：

1. **bucket: "meta"**
   - 存储与集群元数据相关的信息。
   - 示例键：

    - `"cluster.id"`：集群的唯一标识符。
    - `"member.<id>"`：成员节点的元数据。
2. **bucket: "lease"**
    - 存储与租约（lease）相关的数据。
    - 示例键：
        - 租约的 ID 和其对应的 TTL（Time-To-Live）信息。
3. **bucket: "key"**
    - 存储用户写入的键值对数据。
    - 这是 etcd 的核心存储区域，包含了通过 API 写入的所有键值数据。
4. **bucket: "index/lease"`
    - 存储键与租约之间的索引关系。
    - 用于快速查找哪些键关联到特定租约。
5. **bucket: "wal"**
    - 存储 Write-Ahead Log（WAL，预写日志）的元数据。
    - 用于崩溃恢复和数据持久化。
6. **bucket: "alarm"**
    - 存储集群告警信息。
    - 例如，当成员节点不可用时生成的告警。
7. **bucket: "snapshot"**
    - 存储快照相关信息。
    - 用于保存集群状态的快照，便于后续恢复或备份。

这些桶和键的设计使得 etcd 能够高效地组织和管理其内部数据结构，同时支持高并发读写操作和分布式一致性要求。

WAL: Write ahead log

预写日志（Write-Ahead Log，WAL）是 Raft 持久化存储的一部分，用于存储提案（proposals）。首先，领导者会将其提案存储到自身的日志中，然后（并发地）通过 Raft 协议将该提案复制到追随者节点。每个追随者在向领导者确认复制完成之前，都会先将提案持久化到其本地的 WAL 中。

这种机制确保了即使在系统崩溃的情况下，提案数据也不会丢失，并且可以支持 Raft 协议所需的持久化和一致性要求。

etcd 中使用的 WAL（预写日志）与经典的 Raft 模型有两点不同：

1. **持久化内容扩展**：
   etcd 的 WAL 日志不仅持久化索引条目，还持久化 Raft 快照（轻量级快照）和硬状态（hard-state）。因此，成员的整个 Raft 状态可以通过 WAL 日志单独恢复。
2. **仅追加模式**：
   WAL 日志是只追加的。条目不会在原地被覆盖，而是通过在文件中追加新的条目（即使具有相同的索引）来取代之前的条目。这种方式确保了日志的历史记录可以被保留，并且简化了日志的管理和恢复过程。

文件名

WAL 日志文件的命名遵循以下模式：

深色版本
"%016x-%016x.wal", seq, index
例如：./member/wal/0000000000000010-00000000000bf1e6.wal

因此，文件名包含十六进制编码的以下信息：

WAL 日志文件的序列号：

表示该 WAL 文件的顺序编号。
文件中第一个条目或快照的索引：

特别地，第一个文件 0000000000000000-0000000000000000.wal 包含初始的快照记录，其索引为 0。

物理内容

WAL 日志文件包含一系列的“帧”（Frames）。每一帧包含以下内容：

1. **长度字段**：
   使用小端序（LittleEndian）编码的 64 位无符号整数，表示序列化后的 `walpb.Record` 数据的长度。
2. **填充字节（Padding）**：
   一些值为 0 的字节，用于确保整个帧的大小是对齐的（即帧的大小是 8 的倍数）。
3. **序列化的 `walpb.Record` 数据**：
   - **类型（type）**：一个整数编码的枚举值，用于指导如何解释下面的 `data` 字段。
   - **数据（data）**：根据类型的不同，通常是序列化后的 Protocol Buffer 数据。
   - **CRC 校验值（crc）**：从该 WAL 日志创建以来，此副本上所有日志记录中 `data` 字段的组合计算出的 CRC-32 校验值（不包括 `type` 字段）。需要注意的是，CRC 校验值考虑了所有的记录（即使这些记录尚未通过 Raft 协议提交）。

逻辑内容

在逻辑层面上，预写日志（Write-Ahead Log, WAL）文件包含以下内容：

1. **Raftpb.Entry**：
   由 Raft 领导者复制的最近提案。其中一些提案被认为是“已提交”（committed），而其他提案可能会被逻辑上覆盖。
2. **Raftpb.HardState(term, commit, vote)**：
   定期（非常频繁）记录的日志条目中“已提交”索引的信息（即已复制到大多数服务器的日志条目，保证不会被更改或覆盖，并可以应用于后端存储）。此外，它还包含一个“任期号”（term），用于指示是否有选举相关的变更，以及“投票”（vote），表示当前副本在当前任期内投票支持的成员。
3. **walpb.Snapshot(term, index)**：
   定期记录的 Raft 状态快照（不包含数据库内容，仅包含快照日志索引和 Raft 任期号）。
4. **V2 存储内容**：
   存储在单独的 `*.store` 文件中。
5. **V3 存储内容**：
   维护在 bbolt 文件中，一旦条目被应用到该文件中，bbolt 文件就成为一个隐式的快照。
6. **CRC-32 校验值记录**：
   每个文件开头的 CRC-32 校验值记录，用于恢复对文件其余部分的校验。
7. **etcdserverpb.Metadata(node_id, cluster_id)**：
   用于标识该日志所属的集群和副本。

每个 WAL 日志文件的结构（按顺序）：

1. **CRC-32 帧**：
   包含来自所有先前文件的累积 CRC 校验值（第一个文件为 0）。
2. **元数据帧**：
   包含集群 ID 和副本 ID。
3. **初始 WAL 文件特有内容**：
   - **空快照帧（Index: 0, Term: 0）**：此帧的目的是确保所有条目都被一个快照“前置”。
4. **非初始（第二个及后续）WAL 文件内容**：
   - **HardState 帧**。
   - 条目、硬状态和快照记录的混合。

关于 WAL 日志的特性：

1. **索引重复条目**：
   WAL 日志中可能存在多个相同索引的条目。这种情况可能发生在 Raft 论文中图 7 描述的情形中。由于 etcd 的 WAL 日志是只追加的，因此通过追加具有相同索引的新条目来覆盖旧条目。
2. **读取 WAL 日志时的行为**：
   在读取 WAL 日志时，逻辑会用较新的条目覆盖较旧的条目。因此，只有满足 `entry.index <= HardState.commit` 的条目的最后一个版本可以被视为最终版本。索引大于 `HardState.commit` 的条目可能会发生变化。
3. **任期号（terms）**：
   WAL 日志中的任期号应为单调递增。
4. **索引号（indexes）**：
   - 索引号从某个快照开始。
   - 在同一任期号内，索引号应在快照之后依次递增。
   - 如果任期号发生变化，索引号可能会降低，但必须高于最新的 `HardState.commit`。
   - 任何索引号大于等于 `HardState.commit` 的新快照都可能开启一个新的索引序列。

Snapshots of (Store V2): member/snap/{term}-{index}.snap

文件名

文件名在此处生成（格式为 `"%(016x-%016x.snap"`），并使用两个十六进制编码的组成部分：

1. **term**：
   快照生成时的 Raft 任期号（term）。任期号表示两次选举之间的周期。
2. **index**：
   快照生成时最后一个已应用提案的索引号（index）。

创建

`.snap` 文件是由 `Snapshotter.SaveSnap` 方法创建的。

这些文件的生成由以下两种触发机制控制：

1. **基于提案数量的触发**：
   每当应用大约 `--snapshotCount`（默认值为 100,000）个提案时，会创建一个新的快照文件。这是一个近似值，因为我们可能会以批处理的方式接收提案，并且仅在批处理结束时才会考虑创建快照。此外，快照创建过程是异步调度的。`--snapshotCount` 这个标志名称有些误导性，因为它实际上决定了最后一个快照索引与最后一个已应用提案索引之间的差值。
2. **Raft 请求恢复的触发**：
   当 Raft 要求副本从快照中恢复时，副本在通过网络接收到快照消息（`msgSnap`）时，还会将其作为轻量级检查点记录到 WAL 日志中。这确保了在 WAL 日志的尾部始终存在一个有效的快照，后面跟随的是日志条目。这种方式抑制了 WAL 日志中可能出现的连续性缺失问题。

目前，`.snap` 文件大致上与 WAL 日志中的快照条目以 1:1 的方式关联。随着 v2 存储的逐步淘汰（在 3.5.x 版本中为可选，在 3.6.x 版本中为强制），我们预计这些 `.snap` 文件将完全停止写入。

文件内容

该文件包含序列化的 `snapdb.snapshot` 协议缓冲区数据（结构为：`uint32 crc, bytes data`），其中 `data` 字段中存储了 `Raftpb.Snapshot` 数据：

`Raftpb.Snapshot` 的结构：

- `bytes data`：实际的快照数据。
- `SnapshotMetadata`：包含以下元数据：
  - `index`：快照对应的日志索引。
  - `term`：快照对应的任期号。
  - `conf`：集群配置信息。

嵌套的 `data` 内容：

最终，嵌套的 `data` 字段中包含了 JSON 序列化的 v2 存储内容。具体包括以下信息：

1. **Term**：
   快照对应的 Raft 任期号。
2. **Index**：
   快照对应的日志索引。
3. **成员数据（Membership Data）**：
   - `/0/members/8e9e05c52164694d/attributes`：

     ```
     {"name":"default","clientURLs":["<http://localhost:2379>"]}
     ```

     表示成员的名称和客户端访问地址。
   - `/0/members/8e9e05c52164694d/RaftAttributes`：

     ```
     {"peerURLs":["<http://localhost:2380>"]}
     ```

     表示成员的 Raft 通信地址。
4. **存储版本（Storage Version）**：
   - `/0/version`：
     `3.5.0`
     表示存储的版本号。

snap与wal作用与区别

在 etcd 中，**snap（快照）** 和 **WAL（预写日志）** 是两种重要的持久化机制，它们各自有不同的作用和特点。以下是它们的作用和区别：

---

1. WAL（Write-Ahead Log）的作用

- **主要作用**：
  - 记录 Raft 日志条目（proposals），用于实现分布式一致性。
  - 提供崩溃恢复的能力：即使系统发生崩溃，WAL 可以帮助恢复到崩溃前的状态。
- **特点**：
  - **只追加（Append-Only）**：WAL 文件是只追加的，不会修改已有的日志条目，而是通过追加新的条目来覆盖旧的内容。
  - **记录详细信息**：WAL 文件包含所有 Raft 日志条目、硬状态（HardState）、以及周期性生成的轻量级快照元数据。
  - **文件分段**：当 WAL 文件大小超过一定阈值（默认 64MB）时，会创建新的 WAL 文件。
  - **实时性高**：WAL 文件记录了最新的日志条目，确保系统可以快速恢复到最新状态。
- **用途**：
  - 在系统启动或崩溃恢复时，WAL 文件用于重建内存中的 Raft 状态机。
  - 用于存储未提交的日志条目，直到这些条目被确认为已提交并应用到状态机。

---

2. Snap（快照）的作用

- **主要作用**：
  - 定期保存当前状态机的完整状态，减少 WAL 文件的增长。
  - 提供一种高效的方式将整个状态机同步到新加入的节点或替换损坏的节点。
- **特点**：
  - **压缩存储**：快照文件通常是对当前状态机的完整备份，经过压缩后存储。
  - **周期性生成**：快照根据 `--snapshotCount` 参数（默认 100,000 个已应用提案）定期生成。
  - **轻量级检查点**：快照会被记录到 WAL 日志中，确保 WAL 日志尾部始终有一个有效的快照。
  - **独立性**：快照文件与 WAL 文件相对独立，快照记录的是状态机的完整状态，而 WAL 记录的是增量变化。
- **用途**：
  - 在系统启动时，快照文件用于快速加载状态机的初始状态。
  - 当新节点加入集群时，快照可以作为初始同步的基础，避免从零开始复制所有的 WAL 日志。
  - 减少 WAL 文件的增长，降低磁盘空间占用。

---

3. snap 和 WAL 的区别

特性	WAL	Snap
主要功能	记录 Raft 日志条目，支持崩溃恢复和分布式一致性。	定期保存状态机的完整状态，减少 WAL 文件增长并加速新节点同步。
存储内容	增量的日志条目、硬状态（HardState）、轻量级快照元数据。	状态机的完整快照，包括键值对数据和集群元数据。
生成频率	每次有新的日志条目时都会追加到 WAL 文件中。	根据配置（如 `--snapshotCount`）定期生成。
文件大小	文件较小，但会随着日志条目的增加而增长。	文件较大，但生成频率较低，且能有效减少 WAL 文件的增长。
恢复方式	通过回放 WAL 文件中的日志条目来恢复状态机。	直接加载快照文件中的完整状态机，速度更快。

适用场景	适用于记录实时的增量变化，确保系统的崩溃恢复能力。	适用于快速加载状态机、新节点加入集群时的同步，以及减少磁盘占用。
依赖关系	快照文件依赖于 WAL 文件，快照记录会被写入 WAL 文件以保证连续性。	快照文件独立于 WAL 文件，但两者结合使用以优化性能和存储效率。

---

4. 总结

- **WAL** 是 etcd 的核心日志机制，负责记录所有的增量变化，确保系统的分布式一致性和崩溃恢复能力。
- **Snap** 是状态机的完整备份，主要用于减少 WAL 文件的增长、加速新节点同步以及提供高效的系统恢复手段。

两者相辅相成：WAL 提供了实时的日志记录和恢复能力，而快照则提供了高效的状态机备份和同步机制。

使用ETCD进行分布式协调

etcd 自带了一些分布式协调原语，例如事件监听（event watches）、租约（leases）、选举（elections）以及分布式共享锁（distributed shared locks）（需要注意的是，在分布式共享锁的情况下，用户需要了解其一些不明显的特性，具体细节将在下文描述）。这些原语由 etcd 开发者维护和支持。将这些原语交给外部库实现会推卸开发基础分布式软件的责任，实际上会使系统变得不完整。NewSQL 数据库通常期望这些分布式协调原语由第三方编写。类似地，ZooKeeper 以其独立的协调配方库而闻名。Consul 提供了原生的锁定 API，但甚至也承认它“不是万无一失的方法”。

理论上，可以在任何提供强一致性的存储系统之上构建这些原语。然而，这些算法往往非常微妙；很容易开发出看似可行的锁定算法，但在遇到“雷鸣效应”（thundering herd）和时间偏差（timing skew）时却突然失效。此外，etcd 支持的其他原语（例如事务内存）依赖于 etcd 的多版本并发控制（MVCC）数据模型；仅靠简单的强一致性是不够的。

在分布式协调方面，选择 etcd 可以帮助避免操作上的麻烦并节省工程努力。

关于锁和租约使用的注意事项

etcd 提供了基于租约机制及其在 etcd 中实现的锁 API。租约机制的基本思想是：服务器向请求的客户端授予一个令牌，这个令牌被称为租约（lease）。当服务器授予租约时，它会为该租约关联一个 TTL（生存时间）。当服务器检测到时间已经超过 TTL 时，它将撤销该租约。只要客户端持有的租约未被撤销，就可以声称它拥有与该租约相关联资源的访问权。在 etcd 的情况下，该资源是 etcd 键空间中的一个键。etcd 使用这种方案提供了锁 API。然而，这些锁 API 本身并不能作为互斥机制使用。这些 API 被称为“锁”是因为历史原因。不过，如下所述，这些锁 API 可以用作互斥机制的一种优化手段。

租约机制最重要的方面是，TTL（生存时间）被定义为一个物理时间间隔。服务器和客户端分别使用各自的时钟来测量时间的流逝。这种情况可能导致服务器已经撤销了租约，而客户端仍然声称它拥有该租约。

那么，租约机制是如何保证锁机制的互斥性的呢？实际上，租约机制本身并不能保证互斥性。拥有一个租约并不能保证其持有者对资源持有锁。

在使用 etcd 锁来控制对 etcd 自身键的互斥访问时，互斥性是基于版本号验证机制实现的（在其他系统如 Consul 中，这种机制有时被称为“比较并交换”）。在 etcd 的 RPC 操作（如 `Put` 或 `Txn`）中，我们可以为操作指定关于修订版本号和租约 ID 的条件。如果这些条件不满足，操作可能会失败。通过这种机制，etcd 为客户端提供了分布式锁功能。这意味着，当客户端的请求被 etcd 集群成功处理完成后，客户端就知道它已经获取了某个键的锁。

在分布式锁的相关文献中，可以找到类似的设计：

- 在 Chubby 的论文中，引入了“sequencer”（序列器）的概念。我们可以将其理解为与 etcd 中的修订版本号（revision number）和租约 ID（lease ID）的组合几乎相同。
- 在 Martin Kleppmann 撰写的《如何实现分布式锁》一文中，介绍了“fencing token”（围栏令牌）的概念。在 etcd 的情况下，作者们认为 fencing token 就是修订版本号。Flavio Junqueira 在《关于围栏和分布式锁的笔记》中讨论了在 ZooKeeper 的情况下应该如何实现 fencing token 的思想。

- 在《分布式系统中同步时钟的实际应用》一文中，我们发现 Thor 实现了一种基于版本号验证和租约的分布式锁机制。

如果 etcd 和其他系统已经提供了基于版本号验证的互斥机制，为什么还要提供租约（lease）呢？其实，租约提供了一种优化机制，可以减少被中止的请求数量。

需要注意的是，在 etcd 键的情况下，由于租约和版本号验证机制的存在，键可以被高效地锁定。如果用户需要保护与 etcd 无关的资源，那么这些资源必须提供类似于 etcd 键的版本号验证机制以及副本一致性。etcd 自身的锁功能无法用于保护外部资源。

传输安全模型

etcd 支持自动 TLS，并通过客户端证书为客户端到服务器以及节点间（服务器到服务器/集群）通信提供身份验证功能。需要注意的是，为了减少用户在初次使用数据库时的复杂性，etcd 默认情况下并未启用基于 RBAC 的身份验证或传输层的身份验证功能。此外，更改这一默认设置将会对自 2013 年以来建立的项目造成破坏性影响。一个未启用安全功能的 etcd 集群可能会将其数据暴露给任何客户端。

要开始运行，首先需要有一个 CA 证书和一个成员的签名密钥对。建议为集群中的每个成员创建并签署一个新的密钥对。

为了方便起见，cfssl 工具提供了一个简单的证书生成接口。

维护

历史数据压缩：v3 API 键值数据库

自动压缩的历史记录：v3.3.3 及后续版本的行为

在 v3.3.3 中，通过以下参数配置自动压缩功能：

- `--auto-compaction-mode=revision`
  - `--auto-compaction-retention=1000`：每 5 分钟基于“最新修订版”减去 1000 的修订版进行压缩。例如，当最新修订版为 30000 时，压缩会作用于修订版 29000。

在此之前：

- `--auto-compaction-mode=periodic`
  - `--auto-compaction-retention=72h`：每 7.2 小时执行一次压缩，保留 72 小时的历史数据窗口。而现在，每 1 小时执行一次压缩，但仍保留 72 小时的历史数据窗口。
  - `--auto-compaction-retention=30m`：每 3 分钟执行一次压缩，保留 30 分钟的历史数据窗口。而现在，每 30 分钟执行一次压缩，但仍保留 30 分钟的历史数据窗口。

周期性压缩器的行为

当给定的压缩周期小于 1 小时时，周期性压缩器会记录每个压缩周期内的最新修订版；如果给定的压缩周期大于 1 小时，则每小时记录一次最新修订版（例如，当 `--auto-compaction-mode=periodic --auto-compaction-retention=24h` 时，每小时记录一次）。

对于每个压缩周期或每小时，压缩器会使用在压缩周期开始前获取的最后一个修订版来丢弃历史数据。压缩的时间窗口会随着每个给定的压缩周期或每小时向前移动。

示例

1. **每小时写入量为 100，**`--auto-compaction-mode=periodic --auto-compaction-retention=24h`：
   - 在 v3.2.x、v3.3.0、v3.3.1 和 v3.3.2 中，每 2.4 小时分别压缩修订版 2400、2640 和 2880。
   - 在 v3.3.3 或更高版本中，每 1 小时分别压缩修订版 2400、2500 和 2600。
2. **每分钟写入量约为 1000，**`--auto-compaction-mode=periodic --auto-compaction-retention=30m`：
   - 在 v3.3.0、v3.3.1 和 v3.3.2 中，每 3 分钟分别压缩修订版 30000、33000 和 36000。
   - 在 v3.3.3 或更高版本中，每 30 分钟分别压缩修订版 30000、60000 和 90000。

通过这些改进，etcd 在保持相同保留窗口的同时优化了压缩频率和性能。

运维

恢复损坏的成员

有三种方法可以恢复损坏的 etcd 成员：

1. **清除成员持久状态**
2. **替换成员**
3. **恢复整个集群**

在损坏的成员恢复后，可以移除 **CORRUPT ALARM**。

清除成员持久状态

可以通过以下步骤清除成员的状态：

1. 停止 etcd 实例。
2. 备份 etcd 数据目录。
3. 从 etcd 数据目录中移出 `snap` 子目录。
4. 使用 `--initial-cluster-state=existing` 和 `--initial-cluster` 中列出的集群成员启动 etcd。

   预期该 etcd 成员将从 leader 下载最新的快照。

替换成员

可以通过以下步骤替换成员：

1. 停止 etcd 实例。
2. 备份 etcd 数据目录。
3. 删除数据目录。
4. 运行 `etcdctl member remove` 将该成员从集群中移除。
5. 运行 `etcdctl member add` 将其重新添加回集群。
6. 使用 `--initial-cluster-state=existing` 和 `--initial-cluster` 中列出的集群成员启动 etcd。

恢复整个集群

可以通过从当前 leader 保存快照并将其恢复到所有成员来恢复整个集群。运行 `etcdctl snapshot save` 对 leader 进行快照操作，并按照恢复集群的流程进行操作。