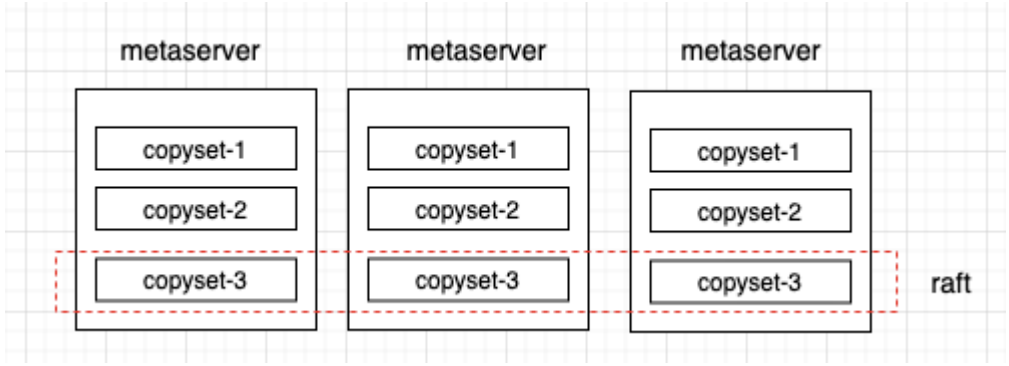
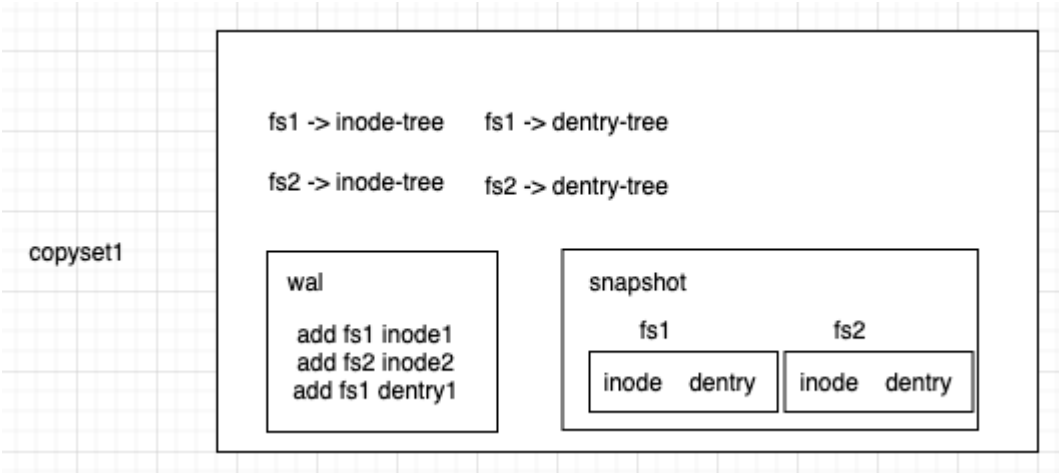
元数据持久化

前言

根据之前讨论的结果，元数据节点的架构如下图所示，这里涉及到两部分需要持久化/编码的内容：

- Raft Log：记录 operator log
- Raft Snapshot：将内存中的数据结构以特定格式 dump 到文件进行持久化



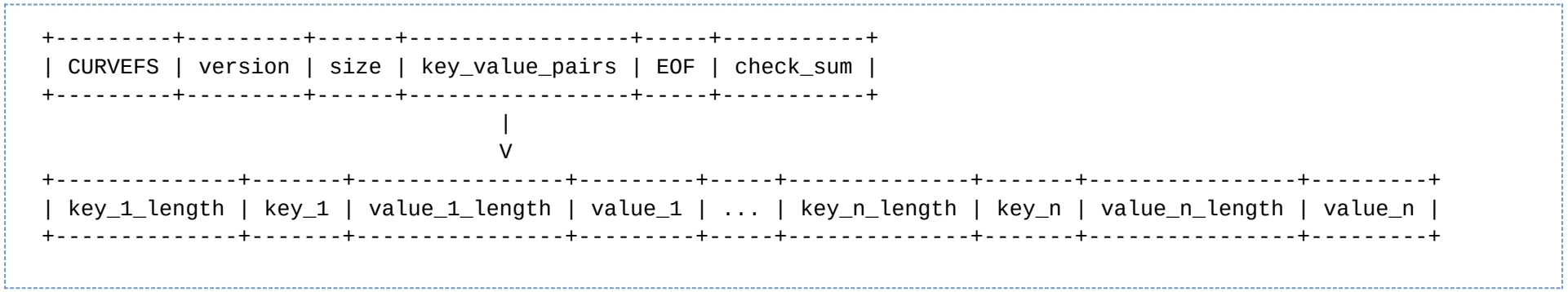


Raft Log

```
+------+------------+-----+----------------+---------+----------+
| type | key_length | key | [value_length] | [value] | checksum |
+------+------------+-----+----------------+---------+----------+
```

字段	字节数	说明
type	1	操作类型，共有以下 2 类： • SET (0X01)：ADD 和 UPDATE 都可以转换成 SET 操作 • DEL (0X02)：当为 DEL 操作时，value_length 和 value 则为空
key_length	4	key 长度
key	$key_length	编码后的 key
[value_length]	4	value 长度
[value]	$value_length	编码后的 value
checksum	8	前面 5 部分的校验和

Raft Snapshot



持久化文件

字段	字节数	说明
CURVEFS	7	magic number（常量字符 "CURVEFS"），用于标识该文件为 curvefs 元数据持久化文件
version	4	文件版本号（当文件格式变化时，可以 100% 向后兼容加载旧版持久化文件）
size	8	键值对数量
key_value_pairs	/	键值对（当 size 为 0 时，该字段为空）
EOF	1	特殊标记常量 (0XFF)，表示内容已结束
check_sum	8	保存校验和（根据前 5 个部分内容计算得出）

key_value_pairs

字段	字节数	说明
key_length	4	key 的长度
key	$key_length	保存编码后的 key
value_length	4	value 长度
value	$value_length	保存编码后的 value

其他说明

- 持久化文件中涉及到的数字均以小端序存储
- 利用 fork 子进程（COW）的方式解决在持久化的过程中，读写冲突的问题以及性能问题

实现

1、inode、entry 的编码

- 给 inode、dentry 增加编码函数　　// 这里要尽可能减少 key/value 编码后的字节数，这样同样的内存可以存入较多的 key/value 对
- 序列化目前主要考虑以下 2 种，一种是参考 chubaofs 顺序编码，一种是利用 protobuf 直接序列化

顺序编码：

```
// Dentry wraps necessary properties of the `dentry` information in file system.
// Marshal exporterKey:
//  +--------+----------+--------+
//  | item   | ParentId | Name |
//  +--------+----------+--------+
//  | bytes |    8     | rest |
//  +--------+----------+--------+
// Marshal value:
//  +--------+--------+--------+
//  | item   | Inode | Type |
//  +--------+--------+--------+
//  | bytes |   8   |   4  |
//  +--------+--------+--------+
// Marshal entity:
//  +--------+----------+----------------+-----------------+----------------+
//  | item   | KeyLength | MarshaledKey | ValLength | MarshaledVal |
//  +--------+----------+----------------+-----------------+----------------+
//  | bytes |     4     |   KeyLength  |     4     |   ValLength  |
//  +--------+----------+----------------+-----------------+----------------+
type Dentry struct {
        ParentId uint64 // FileID value of the parent inode.
        Name     string // Name of the current dentry.
        Inode    uint64 // FileID value of the current inode.
        Type     uint32
}
```

```
// Inode wraps necessary properties of `Inode` information in the file system.
// Marshal exporterKey:
//  +--------+--------+
//  | item   | Inode |
//  +--------+--------+
//  | bytes |   8   |
//  +--------+--------+
// Marshal value:
//  +--------+------+-----+------+------+------+------+-----+----------------------+
//  | item   | Type | Size | Gen | CT | AT | MT | ExtLen | MarshaledExtents |
//  +--------+------+-----+------+------+------+------+-----+----------------------+
//  | bytes |  4   |  8   |  8  |  8  |  8  |  8  |  4   |       ExtLen        |
//  +--------+------+-----+------+------+------+------+-----+----------------------+
// Marshal entity:
//  +--------+----------+----------------+-----------------+----------------+
//  | item   | KeyLength | MarshaledKey | ValLength | MarshaledVal |
//  +--------+----------+----------------+-----------------+----------------+
//  | bytes |     4     |   KeyLength  |     4     |   ValLength  |
//  +--------+----------+----------------+-----------------+----------------+
type Inode struct {
        sync.RWMutex
        Inode      uint64 // Inode ID
        Type       uint32
        Uid        uint32
        Gid        uint32
        Size       uint64
        Generation uint64
        CreateTime int64
        AccessTime int64
        ModifyTime int64
        LinkTarget []byte // SymLink target name
        NLink      uint32 // NodeLink counts
        Flag       int32
        Reserved   uint64 // reserved space
        //Extents    *ExtentsTree
        Extents *SortedExtents
}
```

利用 protobuf（SerializeToString）进行序列化

```
// curvefs/proto/metaserver.proto
message DentryKey {  //
    required uint32 fsId = 1;
    required uint64 parentInodeId = 2;
    required string name = 3;
}

message Dentry {
 required uint32 fsId = 1;
    required uint64 inodeId = 2;
    required uint64 parentInodeId = 3;
    required string name = 4;
}

message InodeKey {  //
    required uint32 fsId = 1;
    required uint64 inodeId = 2;
}

message Inode {
 required uint64 inodeId = 1;
    required uint32 fsId = 2;
    required uint64 length = 3;
    required uint32 ctime = 4;
    required uint32 mtime = 5;
    required uint32 atime = 6;
    required uint32 uid = 7;
    required uint32 gid = 8;
    required uint32 mode = 9;
    required sint32 nlink = 10;
    required FsFileType type = 11;
    optional string symlink = 12;   // TYPE_SYM_LINK only
    optional VolumeExtentList volumeExtentList = 13;  // TYPE_FILE only
}
```

测试对比：

10 万条随机生成 inode	耗时 (MS)	内存 (KB)
顺序编码	13	5079
protobuf 序列化	81	4996

从对比结果来看，10 万条 inode 耗时相差不大（CPU 并不是瓶颈），内存 protobuf 消耗却更少，推介使用 protobuf 进行序列化

2、KVStore

将当前实现中的 MemoryDentryStorage 和 MemoryInodeStorage 抽象成一个 KVStore，对外提供 SET/GET/DEL 等接口，inode/dentry 均编码后以 key-value 的形式存入 KVStore

当前实现可先只实现 KVStore（提供方便 API），Raft 等可以后续接入（目前实现中持久化可以在 KVStore 退出时触发持久化，或定时持久化）

```
class KVStore : public braft::StateMachine {
 public:
 enum class OP_TYPE {
     OP_TYPE_SET,
  OP_TYPE_DEL,
 };
 public:
    bool Init();
    int Set(const std::string& key, const std::string& value);

    int Get(const std::string& key, std::string* value);

 int Delete(const std::string& key);

    int Save();  //

    int Load();  //

 public:
    // on_apply
    // on_snapshot_save
    // on_snapshot_load
    // ...

 private:
    std::string EncodeJournal(OpType opType, const std::string& key, const std::string& value="");
 bool WriteJournal(const std::string& message);
 private:
    map<std::string, std::string> Hash;  //  B+
    std::string filePtah;  // WAL  dump  (WAL : curvefs.waldump : curvefs.dump)
};
```

Q&A

单靠 redis 的 AOF 机制能否保证数据不丢失?

不能，因为 AOF 与 SET/DEL 这些操作不是同步进行的，即使刷入文件配置项 appendfsync 开启最高级别的 always 选项，也有可能丢失一个事件循环的数据，实现如下：

```
// :
call(...)  //
    propagate(...)
        feedAppendOnlyFile(cmd, ...)
            server.aof_buf = sdscatlen(server.aof_buf, ...)  //  op log  buffer
            aofRewriteBufferAppend(...)

/*************************************************/
// :
serverCron()  //
    flushAppendOnlyFile()  //
```

- (1) 命令追加: 将写命令追加到 AOF 缓冲区 server.aof_buf (详见: aof.c/feedAppendOnlyFile)
- (2) 文件写入: 将 AOF 缓冲区的内容以 append 方式写入文件 (详见: aof.c/flushAppendOnlyFile)
- (3) 文件同步: 根据 appendfsync 配置选项决定文件同步频率, 该步骤与步骤 2 紧密关联 (详见: aof.c/flushAppendOnlyFile)

AOF 持久化三种文件同步策略对比

策略	说明	概括
always	• 缓冲区内容写入 AOF 文件后, 并随即同步 AOF 文件 • 如遇故障停机, 最多丢失一个事件循环中所产生的命令数据	效率最低, 但是最安全
everysec	• 缓冲区内容写入 AOF 文件后, 每隔一秒由单独线程同步 AOF 文件 • 如遇故障停机, 最多丢失一秒钟的命令数据	效率较高, 安全性一般
no	• 缓冲区内容写入 AOF 文件后, 何时同步由系统决定 • 如遇故障停, 丢失上次同步 AOF 文件之后的所有命令数据	效率最高, 但是安全性较低

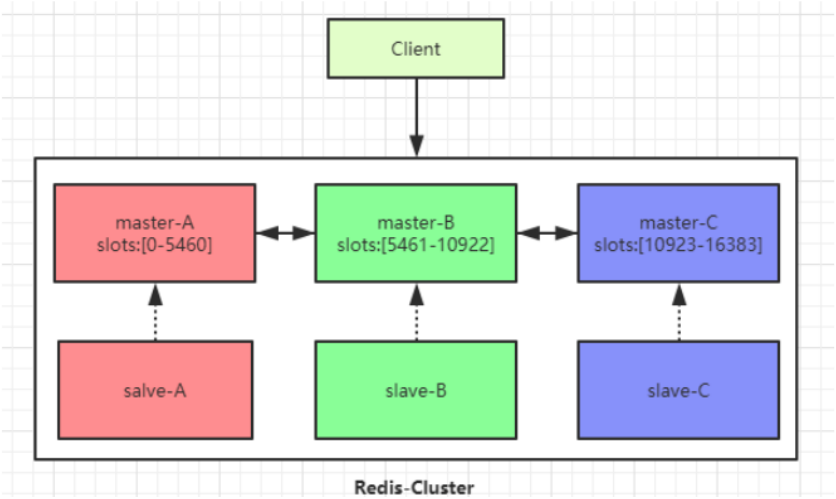
所以，AOF 不能保证数据 100% 不丢失（RDB 持久化更不能），结论就是单靠 redis 无法保证数据 100% 不丢失（这主要是 redis 基于性能考量，毕竟纯内存数据库，如果利用 WAL 每次写文件再 sync，那么性能就会下降很多）

所以，单靠 redis 的方案是不行了.

redis 的高可用、高可扩方案?

主要是 redis cluster + 主从复制（或者第三方 codis + 哨兵）

- redis cluster/codis 主要解决扩展性的问题，它会进行分片，每个 redis 实例保存分片的 key
- 主从复制主要解决高可用，一个分片实例挂 2 个从实例，当主节点挂掉时，cluster/哨兵会自动将从节点升为主节点



redis + muliraft 存在的问题?

- 每个 raft ，需要独立的 snapshot（目前 redis 做不到）探索其可行性？？
- rocksdb/leveldb + multiraft 可行，因为 leveldb 是可嵌入的，一个 raft 实例中可以绑定一个 leveldb 实例（leveldb 中的 wal 和 SST 文件都可以写到指定的目录）

redis 改造 vs 自己实现?

结论：从目前元数据持久化的需要来看，更倾向于自己实现，理由如下：

1. redis 目前不支持单独持久化 redis 中的某个 DB（一个 redis 实例可包含多个 DB）或数据结构，这对于在要使用 multiraft 的场景下，每个 raft 实例需要独立的快照并不合适
    - 如果改造 redis，初步评估了下，其工作量要比自己实现持久化的逻辑要大一些，改造主要是为了让 redis 提供单独 dump/load 一个 DB 的功能：
        - 如果改造，dump/load 的逻辑都得动，而且会牵扯到一些其他逻辑（如主从复制，因为 redis 主从全量复制发送的就是一整个 RDB 文件，即使我们不需要这个功能，这部分代码也是有耦合的）

    - 如果自己实现，只是一个简单的 sava/load 逻辑，比较清晰
2. redis 中有许多数据结构可供使用，如（哈希、列表、set、sort_set），但对于目前的需求来说，我们内存数据结构使用的是 C++ STL 中的哈希表（unsorted_map），之后有可能根据需求换成 B+ 树或跳表，但是 redis 中的这些数据结构我们是不需要的
    - 另外，如果 C++ 中的哈希表在后期使用中发现性能不达标的话（特别是在 rehash 扩桶的时候），我们可以把 redis 中的哈希表借鉴过来用（redis 中的哈希实现很独立，单独的文件 t_hash.c，其性能表现也非常好）
    - redis 哈希表实现主要优点参考以下
3. 总的来说，我们只是参考 redis 持久化实现，而 redis 中的大头（各类数据结构、模块化、主从复制、集群等这些都是我们目前不需要的），因此去改造 redis 感觉不是很划算

redis 中哈希表实现的优点？

主要是当哈希表需要扩桶的时候，rehash 过程中 redis 采用了均摊/渐进式的思想，把 rehash 中的性能损耗均摊在每一次 SET/DEL 操作中（如 rehash 总耗时 1 秒，均摊给 100 个请求，那么每个请求只增加延时 10 毫秒），rehash 过程如下：

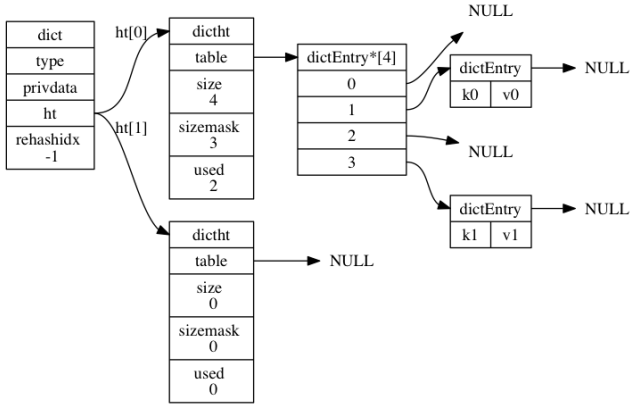


图 4-3　普通状态下的字典

哈希表渐进式 rehash 的详细步骤：

- (1) 为 ht[1] 分配空间，让字典同时持有 ht[0] 和 ht[1] 两个哈希表
- (2) 在字典中维持一个索引计数器变量 rehashidx，并将它的值设置为 0，表示 rehash 工作正式开始
- (3) 在 rehash 进行期间，每次对字典执行添加、删除、查找或者更新操作时，程序除了执行指定的操作以外，还会顺带将 ht[0] 哈希表在 rehashidx 索引上的所有键值对 rehash 到 ht[1]，当 rehash 工作完成之后，程序将 rehashidx 属性的值增一
- (4) 随着字典操作的不断执行，最终在某个时间点上，ht[0] 的所有键值对都会被 rehash 至 ht[1]，这时程序将 rehashidx 属性的值设为 -1，表示 rehash 操作已完成

哈希表渐进式 rehash 执行期间的哈希表操作：

- 因为在进行渐进式 rehash 的过程中，字典会同时使用 ht[0] 和 ht[1] 两个哈希表，所以在渐进式 rehash 进行期间，字典的删除 (delete)、查找(find)、更新(update) 等操作会在两个哈希表上进行：比如说，要在字典里面查找一个键的话，程序会先在 ht[0] 里面进行查找，如果没找到的话，就会继续到 ht[1] 里面进行查找，诸如此类
- 另外，在渐进式 rehash 执行期间，新添加到字典的键值对一律会被保存到 ht[1] 里面，而 ht[0] 则不再进行任何添加操作：这一措施保证了 ht[0] 包含的键值对数量会只减不增，并随着 rehash 操作的执行而最终变成空表

参考

- leveldb/boltdb/redis 持久化调研