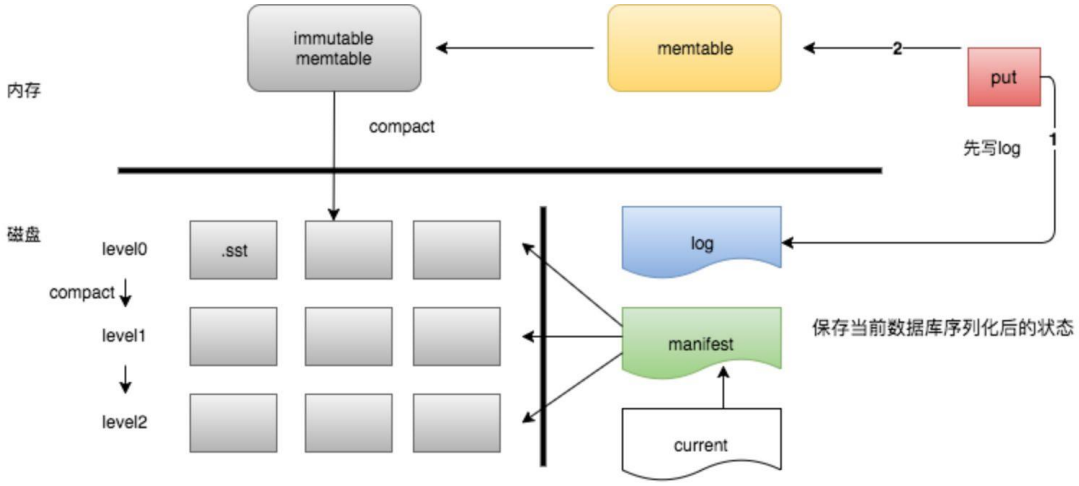


浅析开源项目之 LevelDB

前言

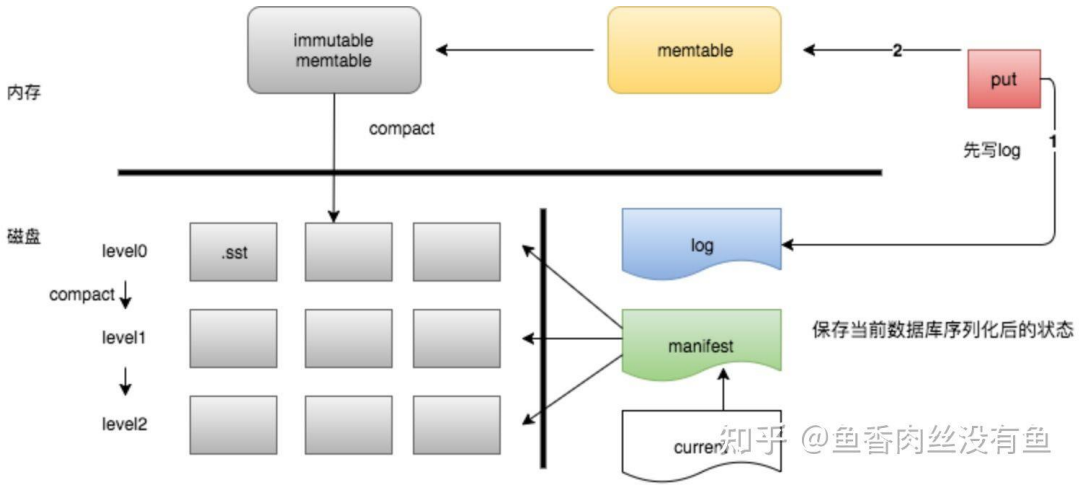
LevelDB 是一个单机持久化的 KV 存储引擎，通常用在分布式 KV、分布式数据库等领域，本文简要介绍 LevelDB 的整体架构以及涉及的重要模块，使读者对 LevelDB 有一个大致清晰的认识。

目录

- 架构设计
- 重要类图
- 数据结构
  - MemTable
  - WAL
  - SSTable
  - Manifest
- IO 流程
  - Open 流程
  - Put 流程
  - Get 流程
- Compaction
  - Minor Compaction
    - 触发时机
    - 执行过程
    - 输出
  - Major Compaction

        - 触发时机
        - 执行过程
        - Pick SSTable
        - 输出
    - Manual Compaction
        - 触发时机
        - 执行过程
        - 输出
- 版本控制

架构设计



LevelDB 整体由以下 6 个模块构成：

MemTable：KV 数据在内存的存储格式，由 SkipList 组织，整体有序。

Immutable MemTable：MemTable 达到一定阈值后变为不可写的 MemTable，等待被 Flush 到磁盘上。

Log：有点类似于文件系统的 Journal，用来保证 Crash 不丢数据、支持批量写的原子操作、转换随机写为顺序写。

SSTable：KV 数据在磁盘的存储格式，文件里面的 key 整体有序，一旦生成便是只读的，L0 可能会有重叠，其他层 sstable 之间都是有序的。

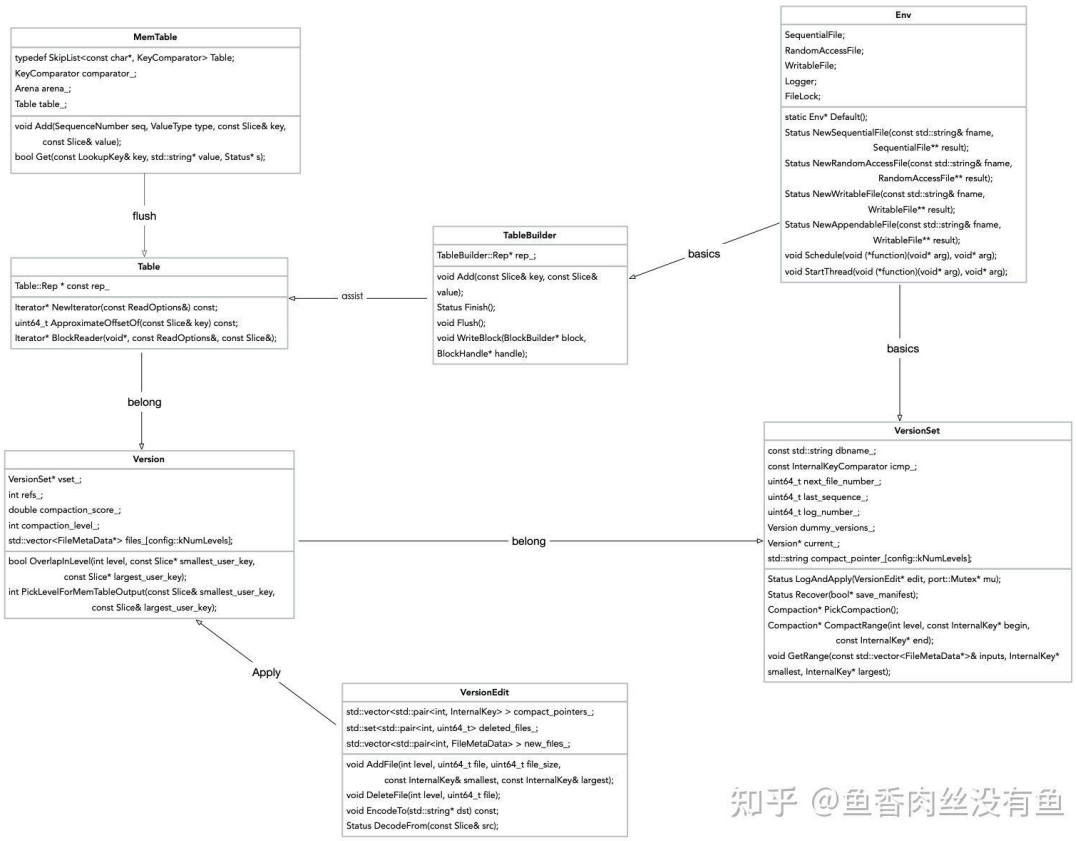
Manifest：增量的保存 DB 的状态信息，使得重启或者故障后可以恢复到退出前的状态。

Current：记录当前最新的 Manifest 文件。

相关概念

- **Env**：leveldb 将操作系统相关的操作(文件、线程、时间)抽象成 Env，用户可以实现自己的 Env(BlueRocksEnv)，灵活性比较高。
- **SequenceNnumber**：leveldb 的每次更新都会携带一个版本，由递增的 DB 唯一的 SequenceNnumber 标识。
- **Snapshot**：本质上一个 SequenceNnumber，用来给整个 DB 打快照，使用双向循环链表保存多个 snapshot。
- **Comparator**：key 排序的比较方法，默认按照字节比较，用户可传入自己的比较方法。
- **Slice**：存放 KV 数据的容器，类似于 Cellar 的 data_entry 和 ceph 的 bufferlist。
- **FileMetaData**：sstable 的元信息，包括文件大小、smallest_key，largest_key 等。
- **WAL Block**：WAL 文件和 Manifest 文件使用的 block 组织形式。
- **Block**：sstable 的数据组织粒度，多个 kv 会聚合成 block 一次写入，读取时也是按照 block 粒度读取。
- **BlockHandle**：block 的元信息，记录位于 sstable 的 offset/size。
- **BlockBuilder**：负责生成 block。
- **TableBuilder**：负责生成 sstable。

重要类图



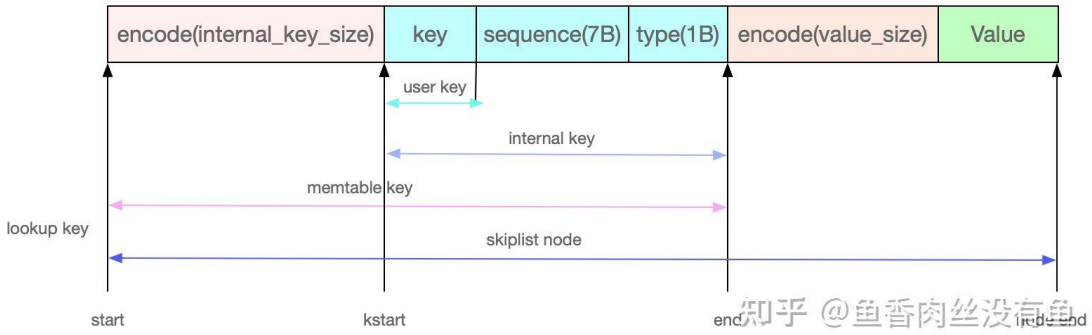
数据结构

MemTable

MemTable 以及 Immutable MemTable 是 KV 数据在内存中的存储格式，底层数据结构都是 SkipList，插入查找的时间复杂度都是 Olog(n)。

MemTable 的大小通过参数 write_buffer_size 控制，默认 4MB，最多 5MB dump(最大 batch size 为 1MB)成 SSTable。

当一个 MemTable 大小达到阈值后，将会变成 Immutable MemTable，同时生成一个新的 MemTable 来支持新的写入，Compaction 线程将 Immutable MemTable Flush 到 L0 上。所以在 LevelDB 中，同时最多只会存在两个 MemTable，一个可写的，一个只读的。



SkipList Node：

由于 SkipList 是链表形式的，所以我们需要把 KV 数据的映射形式转换成该形式。如上图所示，[start, node_end]区间就代表一个 SkipList Node。

Lookup Key：

Loopup Key 是 LevelDB 里面经常使用的查找 Key，包含 user_key、internal_key、memtable_key。

1. UserKey：最直接简单的 client 传入的 key。
2. InternalKey：UserKey+SequenceNumber(7B)+ValueType(1B)，SSTable 用的 key。
3. MemTableKey：InternalKeyLength+InternalKey，MemTable 用的 key。

Key 比较：

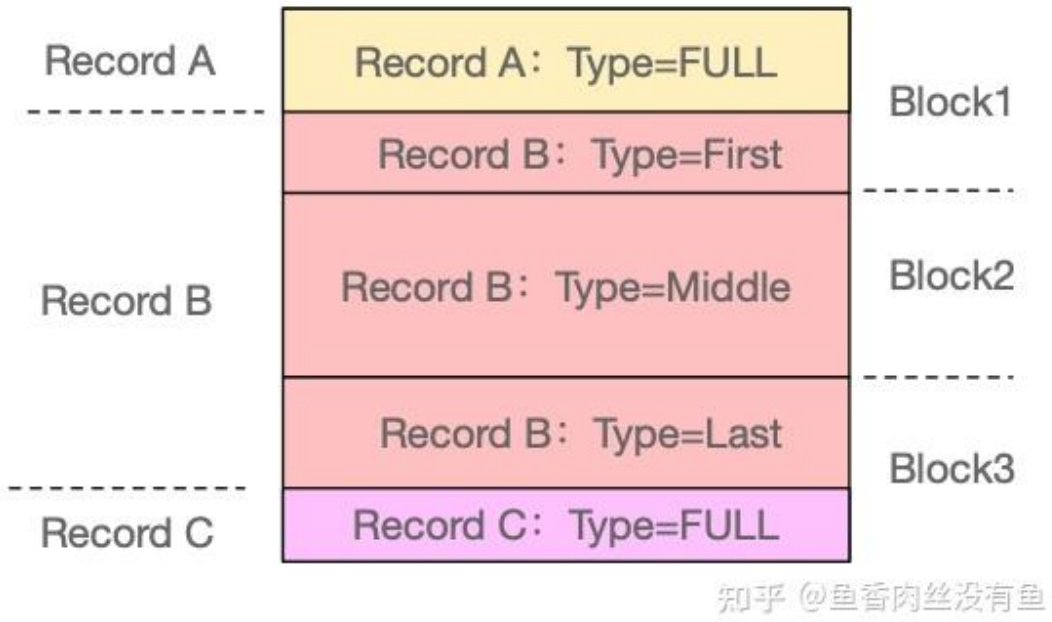
1. 从 MemTableKey 中解析出 InternalKey。
2. 先比较 UserKey，akey < bkey 则返回-1。若 UserKey 相等则比较 SequenceNumber。
3. 解析出逆序排序的 SequenceNumber，越大数据越新即：akey_seq_num > bkey_seq_num 则返回-1，更快的找到数据。
4. 递增排序示例： keya10--->keya8--->keya6--->keyb5--->keyb3--->keyc1。

WAL

WAL 即 Log，每次数据都会先顺序写到 Log 中，然后再写入 MemTable，可以起到转换随机写为顺序写以及保证 Crash 不丢数据的作用。

一个完整的 Log 由多个固定大小的 block 组成，block 大小默认 32KB；block 由一个或者多个 record 组成。

WAL Format：



Record Format:

Format:	checksum + length + type	data
Comment:	header: 4 + 2 + 1 = 7B	key-value data

- checksum：计算 type 和 data 的 crc。
- length：data 的长度，2Byte 可表示 64KB，而 block 为 32KB，刚好够用。
- type：一个 record 可以在一个或者跨越多个 block，类型有 5 种：FULL、First、Middle、Last、Zero(预分配连续的磁盘空间用)。
- data：用户的 kv 数据。

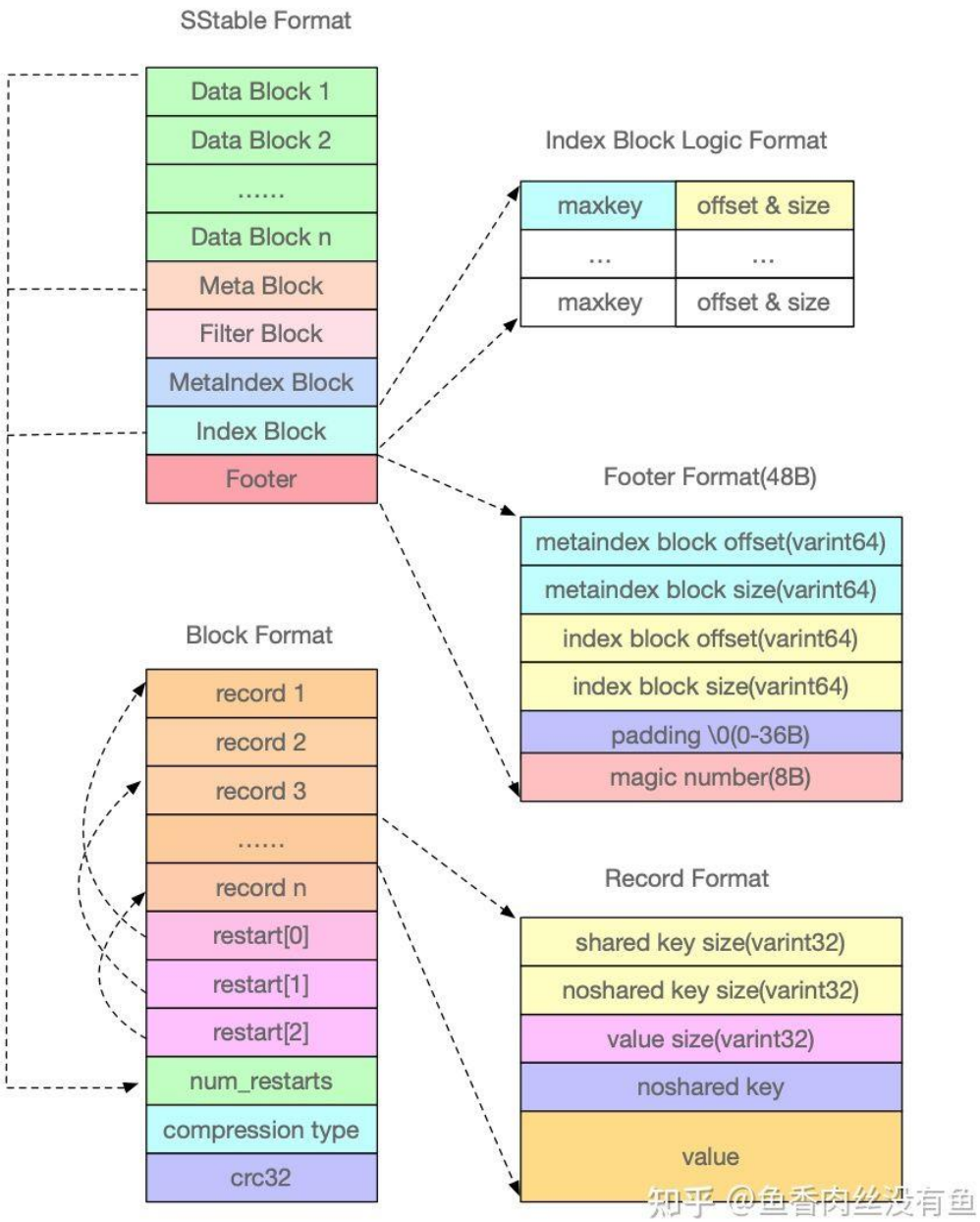
读写概要：

1. 如果 block 剩余空间小于 kHeaderSize(7B)，则填充 0。

2. 如果 block 剩余空间等于 kHeaderSize(7B)，则仅仅写入 header，不写入数据。
3. header 使用小端序存储。
4. PosixWritableFile 类有一个 64KB 的 WriteBuffer，先写入该 buf，写满则 Flush，然后剩余的字节数如果大于 buf 总大小直接全部写磁盘，否则写 buf。
5. 每写一个 record 或者 block 写满就要 Flush 数据，注意 Flush 的语义仅仅是调用系统调用写入数据到 PageCache。
6. 读取 record 时如果发现 crc 不一致，则 report Corruption 错误。
7. 只有在 DB 启动 Replay 时才会读取 Log 且每次读取 kBlockSize(32KB)的数据。

SSTable

SSTable 整体的格式如下图所示：



- DataBlock：存储实际的 kv data、type、crc。
- MetaBlock：暂时没有使用，不过可将 Filter Block 当成一种特殊的 MetaBlock。
- MetaIndexBlock：保存 MetaBlock 的索引信息，目前仅有一行 KV 数据，记录了 FilterBlock 的 name 以及 offset/size。
- IndexBlock：保存每个 DataBlock 的 LastKey 和在 SST 文件中的 offset/size。
- Footer：文件末尾固定长度的数据，保存 MetaIndexBlock、IndexBlock 的索引信息。

SSTable 中的 BlockSize 大小默认为 4K，MetaIndex、DataBlock、IndexBlock 都是使用同样的 BlockBuilder 来构建 Block，区别是里面的 KV 数据不同。

DataBlock 中的 KV 是有序存储的，相邻的 key 之间很有可能重复，因此采用前缀压缩来存储 key，后一个 key 只存储与前一个 key 不同的部分。

然后重启点指出的位置就表示该 key 不按前缀压缩，而是完整存储该 key。对于 MetaBlock 和 IndexBlock 来说由于相邻 key 差距比较大，所以不开启前缀压缩，即 block_restart_interval 为 1。

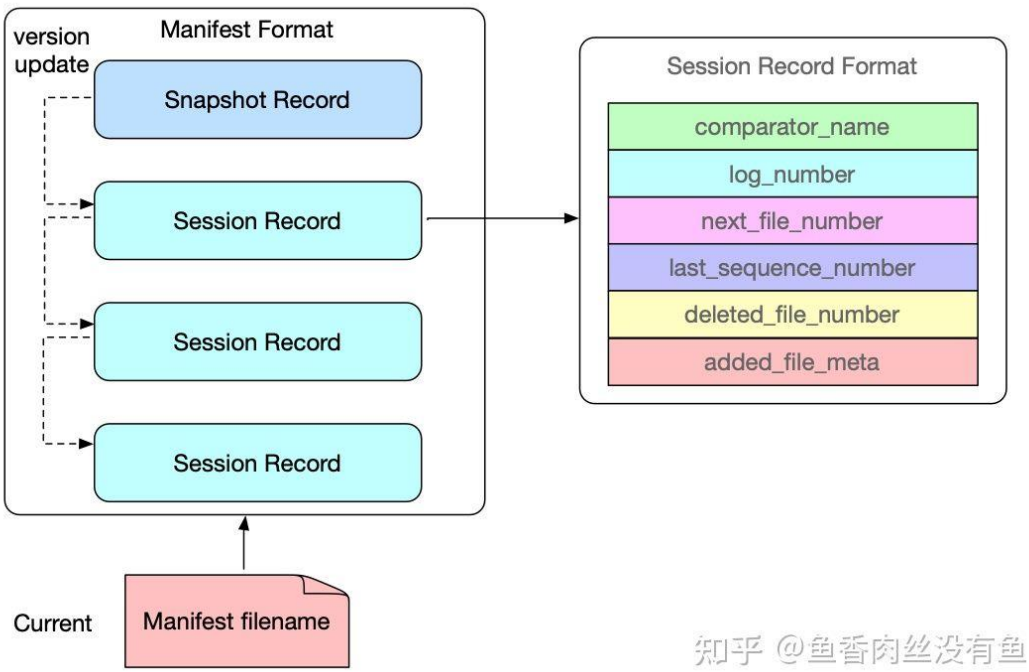
TableCache：缓存 SST 文件的元信息，包含文件 fd、metaindex_block、index_block、filter_block 等，默认缓存 1000 个 SSTable 文件的元信息，可通过 options.max_open_files 指定。

BlockCache：缓存未压缩的 data block 数据，默认 8MB，16 个 shard，可通过 options.block_cache 指定大小，读取的时候也可以指定是否放到 cache 中。

Manifest

Manifest 文件以增量的方式持久化版本信息，DB 中可能包含多个 Manifest 文件，需要 Current 文件来指向最新的 Manifest。

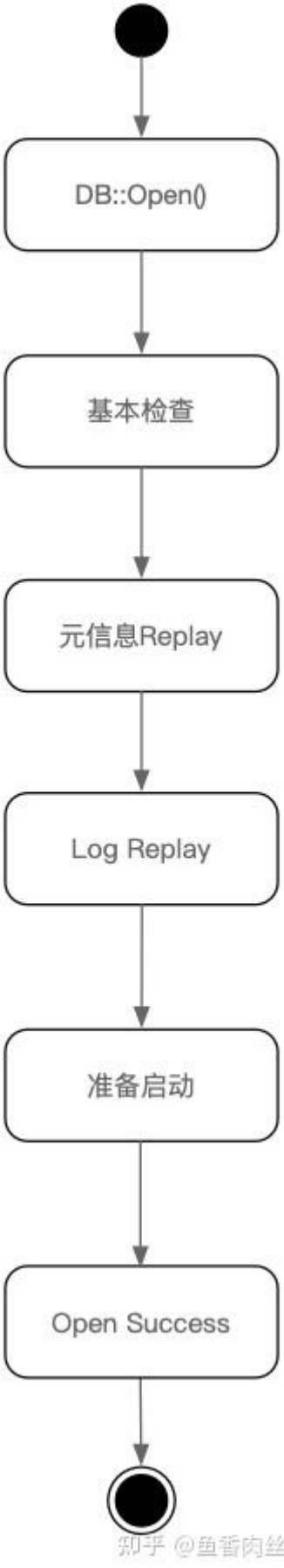
Manifest 包含了多条 Record，第一条是 Snapshot Record，记录了 DB 初始的状态；之后的每条 Record 记录了从上一个版本到当前版本的变化，具体格式如下图：



每次做完 Minor Compaction、Major Compaction 或者重启 Replay 日志生成新的 Level0 文件，都会触发版本变更。

IO 流程

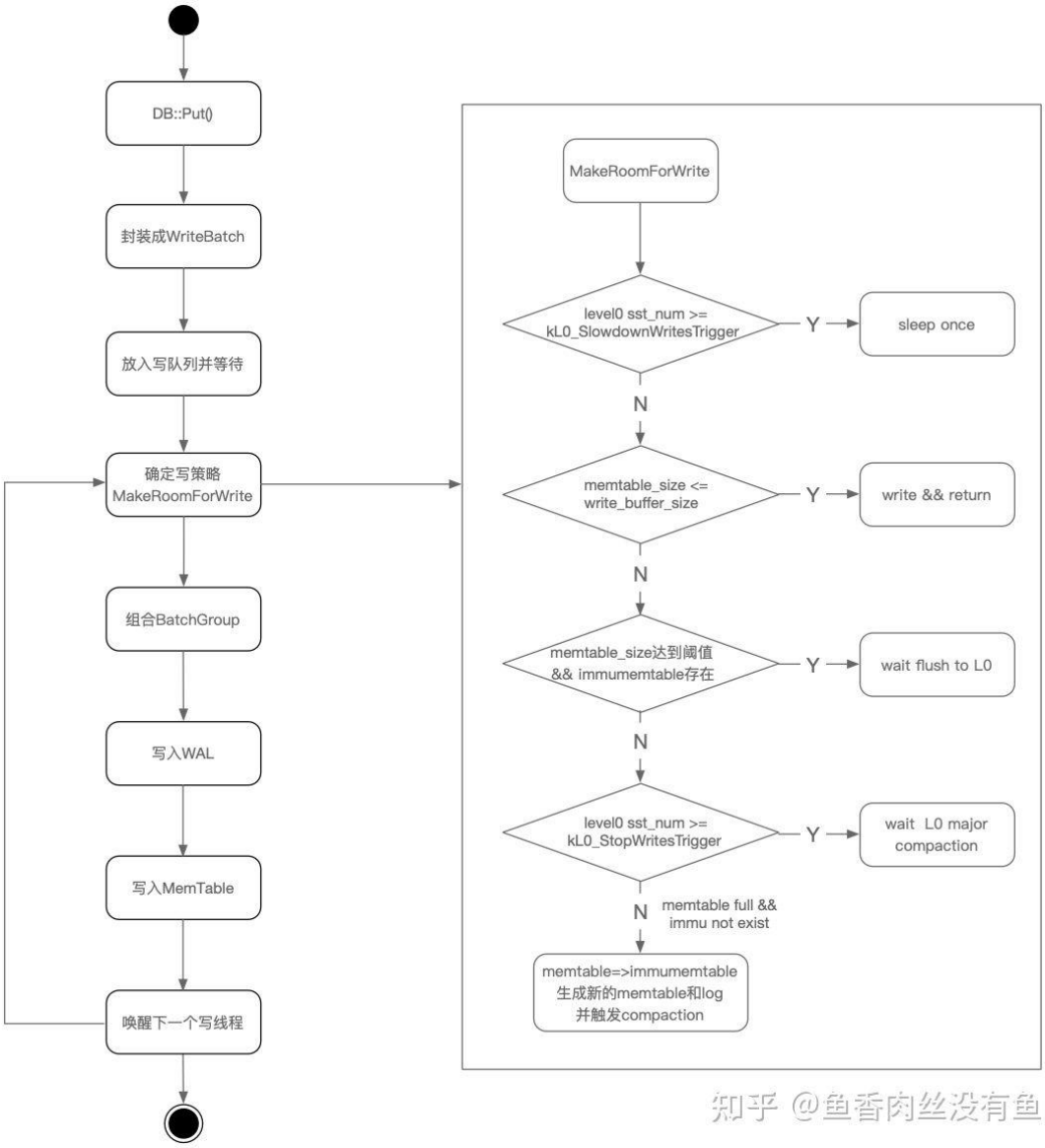
Open 流程



1. DB::Open()
2. 基本检查
    1. 对 DB Lock 文件加锁确保一份数据只能启动一个 DB 实例。
    2. 根据 option 传入的 create_if_missing/error_if_exists 参数做不同的处理。
3. 元信息 Replay
    1. 从 Current 文件获取当前的 Manifest 文件。
    2. 从 Manifest 文件依次读取并解析每个 record Apply 到 builder。
    3. 创建当前唯一的 Version 并调用 builder 的 SaveTo 方法保存信息到当前 Version。
    4. 调用 Finalize()计算下次 compaction 要处理的 level。
    5. 判断是否需要 Reuse Manifest，减少 DB 的启动时间。
    6. 检查从 Manifest 解析的最终状态的基本信息是否完整并应用到当前 DB 状态。
    7. DB 已恢复到上次退出的状态。
4. Log Replay
    1. 遍历 DB 中的 Log 文件，根据 LogNumber 找到需要 replay 的 Log。
    2. 遍历 Log 中的 record 重建 MemTable，并且 MemTable 达到阈值时就 dump 成 SSTable。
    3. 将最后的 MemTable dump 成 SSTable。
    4. 根据 Log 的 FileNumber 和遍历 record 的 SequenceNumber 修正从 Manifest 获取的值。
5. 准备启动
    1. 生成新的 Log 文件并更新 DB 元信息，调用 LogAndApply 方法。
    2. 删除无用文件，尝试触发 compaction。
6. Open Success。

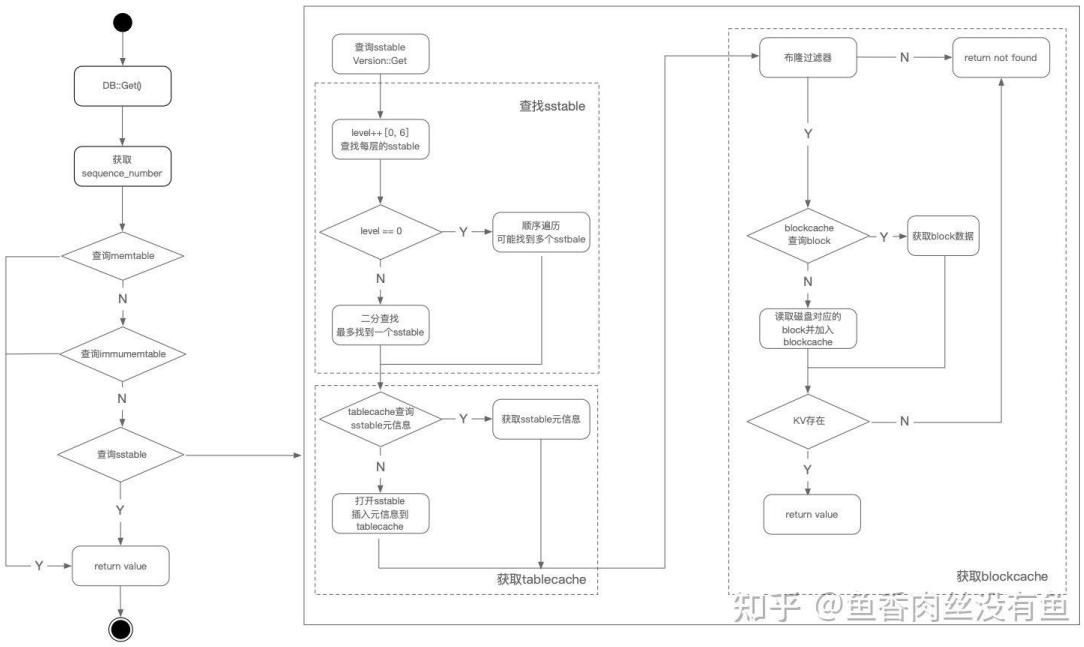
Put 流程

1. 先写 Log，再写 MemTable，通过 WriteOptions 的 sync 参数控制是否 Log 落盘。
2. 无论是单个写入还是批量写入都会封装成 WriteBatch 接口，写入 string 类型的变量里面。
3. Put/Delete 的每个 key 都会有一个唯一的全局递增的 SequenceNumber(7 Bytes)。
4. 组合 BatchGroup 时阈值为 1MB，如果 size 小于 128K，则阈值为 size+128K，减少小写延迟。



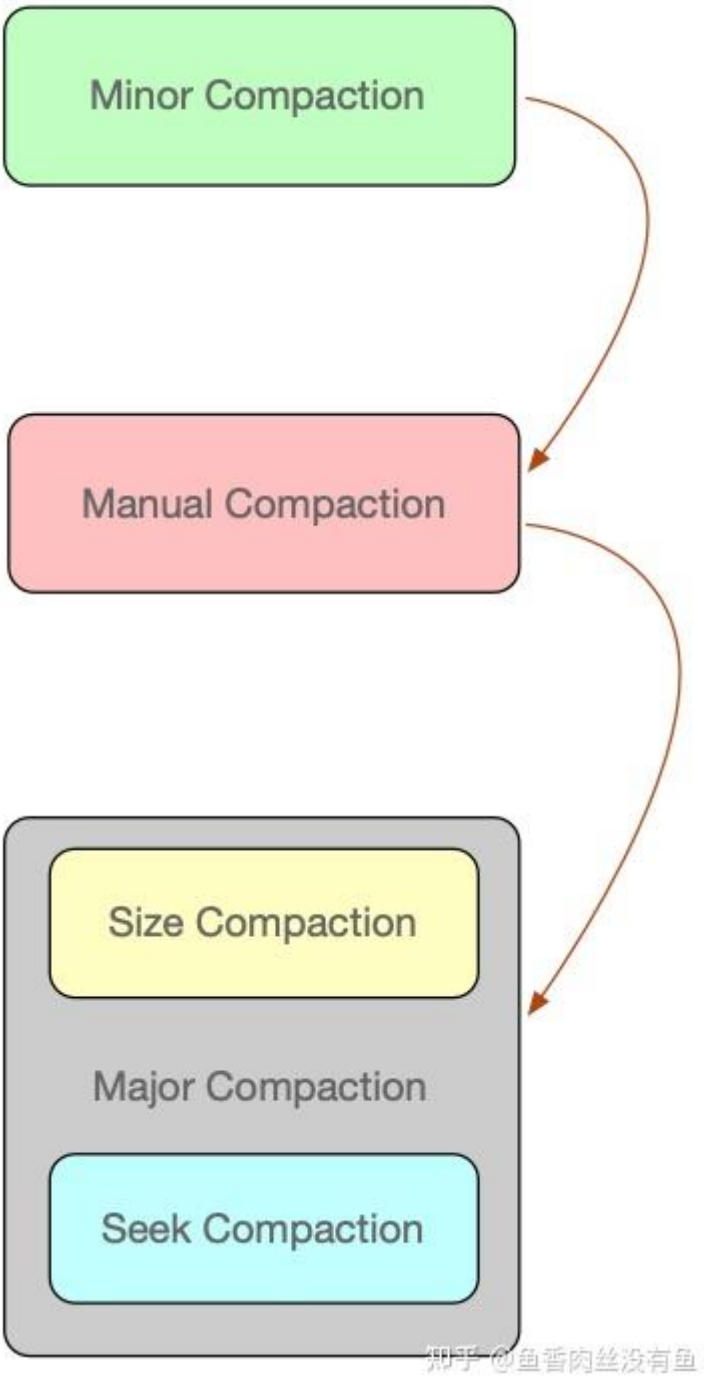
Get 流程

1. 先从 VersionSet 中获取当前的 SequnceNumber(如果指定 snapshot 则使用 snapshot 的 sn)。
2. 读取 MemTable 再读取 ImmutableMemTable，如果没有找到，则从磁盘上依次读取 Level0~LevelN。
3. 查询 Level 时先查找符合的 sstable，再获取 sstable 的 tablecache、最后获取 blockcache。
4. 由于 Level0 之间的 SST 文件可能会有 Key 重叠，Level1~N 之间的 SST 文件不会有 Key 重叠，所以查找 sstable 时 L0 需要遍历，其他 Level 二分查找。



Compaction

LevelDB 的写入和删除都是追加写 WAL，所以需要 Compaction 来删除那些重复的、过期的、待删除的 KV 数据，同时也可以加速读的作用，其类型和优先级如下图所示：



compaction 类型：

LevelDB 中有三类 Compaction：

1. minor compaction：immutable memtable 持久化为 sstable。
2. major compaction：sstable 之间的 compaction，多路归并排序。
3. manual compaction：外部调用 CompactRange 产生的 Compaction。

其中 major compaction 有两类：

1. size compaction：根据 level 的大小来触发。
2. seek compaction：每个 sstable 都有一个 seek miss 阈值，超过了就会触发。

compaction 优先级：

LevelDB 在 MaybeScheduleCompaction/BackgroundCompaction 中完成对 compaction 优先级的调度。

具体优先级为：minor > manual > size > seek。

1. 如果 immutable memtable 不为空，则 dump 到 L0 的 sstable。
2. 如果 is_manual 为 true 即 manual compaction，则调用 CompactRange。
3. 最后调用 PickCompaction 函数，里面会优先进行 size compaction，再进行 seek compaction。

Minor Compaction

minor compaction 将 immutable memtable 持久化为 sstable，我们主要关注何时触发以及生成的 sstable 需要放在哪一层。

触发时机

Write(Put/Delete)、CompactRange、Recovery 以及 compaction 之后都会触发 minor compaction，最频繁触发的操作还是 Write 操作。

执行过程

当 immutable memtable 持久化为 sstable 的时候，大多数情况下都会放在 L0，然后并不是所有的情况都会放在 L0，具体放在哪一层由 PickLevelForMemTableOutput 函数计算。

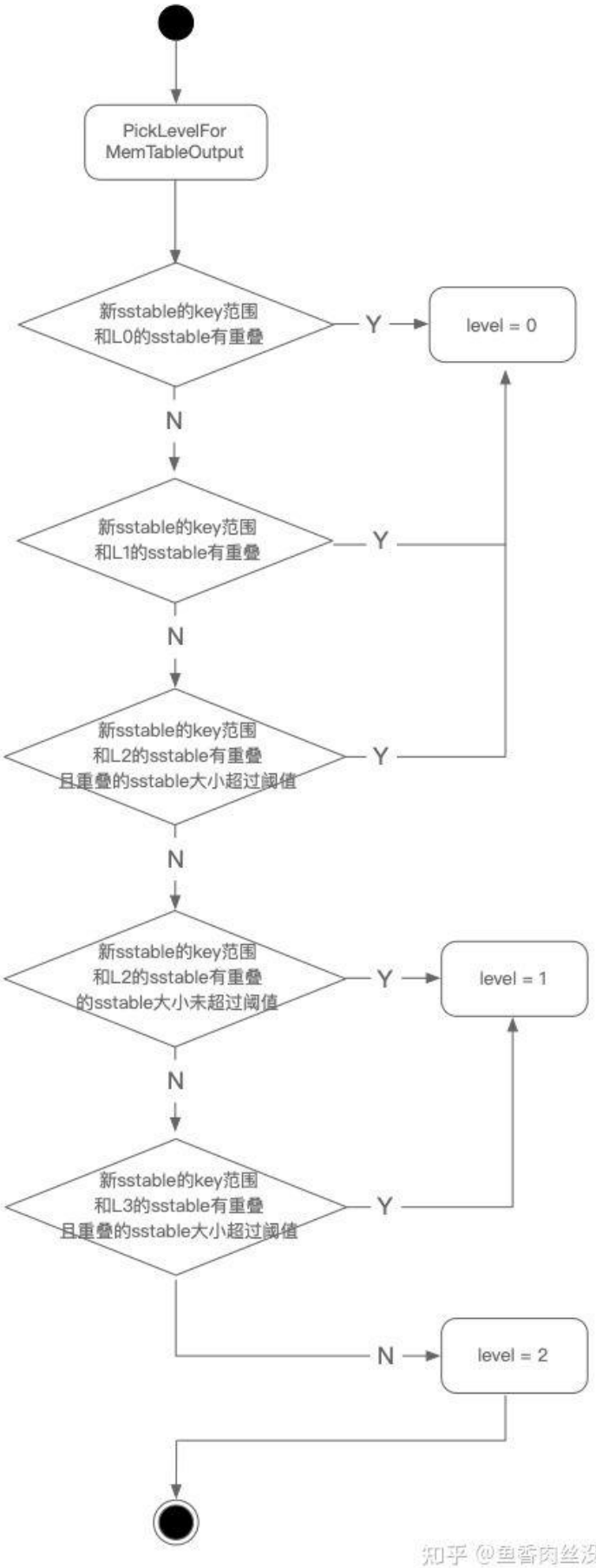
理论上应该需要将 dump 的 sstable 推至高 level，因为 L0 文件过多会导致**查找耗时增加**以及 **compaction 时内部 IO 消耗严重；**

但是又不能推至太高的 level，因为需要控制查找的次数，而且某些范围的 key 更新频繁时，往高 level compaction **内部 IO 消耗严重**，而且也不易 compaction 到高 level，导致**空间放大严重**。

所以 PickLevelForMemTableOutput 在选择输出到哪个 level 的时候，需要权衡查找效率、compaction IO 消耗以及空间放大，大体策略如下：

1. 最高可推至哪层由 kMaxMemCompactLevel 控制，默认最高 L2。
2. 如果 dump 成的 sstable 和 L0/L1 有重叠，则放到 L0。
3. 如果 dump 成的 sstable 和 L2 有重叠且重叠 sstable 总大小超过 10 * max_file_size，则放在 L0。
    1. 因为此时如果放在 L1 会造成 compaction IO 消耗比较大。
    2. 所以放在 L0，之后和 L1 的 sstable 进行 compaction，减小 sstable 的 key 范围，从而减小下次 compaction 涉及的 sstable 总大小。
4. 如果 dump 成的 sstable 和 L3 有重叠且重叠 sstable 总大小超过 10 * max_file_size，则放在 L1。

具体 PickLevelForMemTableOutput 的策略如下图：


PickLevelFor
MemTableOutput
新sstable的key范围
和L0的sstable有重叠
Y
level = 0
N
新sstable的key范围
和L1的sstable有重叠
Y
N
新sstable的key范围
和L2的sstable有重叠
且重叠的sstable大小超过阈值
Y
N
新sstable的key范围
和L2的sstable有重叠
的sstable大小未超过阈值
Y
level = 1
N
新sstable的key范围
和L3的sstable有重叠
且重叠的sstable大小超过阈值
Y
N
level = 2

输出

minor compaction 生成的 sstable 不受 max_file_size(default 2MB)的限制，通常为 write_buffer_size 的大小。

Major Compaction

major compaction 是 LevelDB compaction 中最复杂的部分，主要包含 size_compaction 和 seek_compaction，会进行重复数据、待删除的数据的清理，减少空间放大，提高读效率。

Seek Compaction

每个 sstable 都有一个 allowed_seek 的初始化阈值，表示允许 seek_miss 多少次；每当 get miss 的时候都会减 1，当减为 0 的时候标记为需要 compaction 的文件，参与 compaction，从而避免不必要的 seek miss 消耗 IO。但是引入了布隆过滤器之后，查找 miss 消耗的 IO 就会小很多，seek compaction 的作用也大大减小。

接下来主要介绍 Size Compaction

触发时机

1. DB Open 时会触发 compaction。
2. Write(Put、Delete)会检查是否需要触发 size compaction。
3. Get 时会检查是否需要触发 seek compaction。
4. compaction 之后会检查是否需要再次触发 compaction。

执行过程

1. 调用 versions_->PickCompaction()函数获取需要参加 compaction 的 sstable。
2. 如果不是 manual 且可以 TrivialMove，则直接将 sstable 逻辑上移动到下一层。
    1. 当且仅当 level_n 的 sstable 个数为 1，level_n+1 的 sstable 个数为 0，且该 sstable 与 level_n+2 层重叠的总大小不超过 10 * max_file_size。

3. 获取 smallest_snapshot 作为 sequence_number。如果有 snapshot 则使用所有 snapshot 中最小的 sequence_number，否则使用当前 version 的 sequence_number。
4. 生成 MergingIterator 对参与 compaction 的 sstable 进行多路归并排序。
5. 依次处理每对 KV，把有效的 KV 数据通过 TableBuilder 写入到 level+1 层的 sstable 中。
    1. 期间如果有 immu memtable，则优先执行 minor compaction。
    2. 重复的数据直接跳过，具体细节处理如下：
        1. 如果有 snapshot，则保留大于 smallest_snapshot 的所有的 record 以及一个小于 smallest_snapshot 的 record。
        2. 如果没有 snapshot，则仅保留 sequence_number 最大的 record。
    3. 有删除标记的数据则判断 level i+2 以上层有没有该数据，有则保留，否则丢弃。
6. InstallCompactionResults 将本次 compaction 产生的 VersionEdit 调用 LogAndApply 写入到 Manifest 文件中，期间会创建新的 Version 成为 Current Version。
7. CleanupCompaction 以及调用 DeleteObsoleteFiles 删除不属于任何 version 的 sstable 文件以及 WAL、Manifest 文件。

Pick SSTable

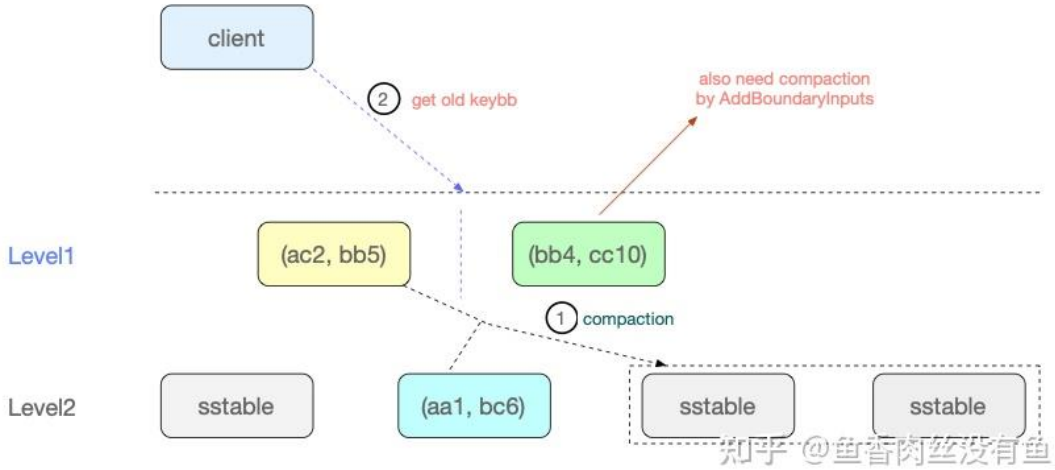
1、选取出最急需进行 compaction 的 level。

1. 为了选择出最紧急 compaction 的 level，每层都会有一个 score 值，在 VersionSet::Finalize 函数中计算。
2. 然后选取 score 值最大且大于等于 1 的 level 进行 major compaction。
3. level_0 score = L0_current_sstable_number / L0_compaction_trigger(4)
4. level_x score = Lx_total_file_size / max_bytes_for_Lx
5. 每层阈值：L0 = 10MB，L1 = 10MB，Ln = 10^n MB

2、选取 level_i 上的 sstable。

1. 每个 level 都有一个 string 类型的 compact_pointer 来判断需要从该 level 的那个位置开始 compaction。
2. 选择一个或者多个大于 compact_pointer 的 sstable 参与 compaction，大部分情况下是一个。
    1. 会先选择一个大于 compact_pointer 的 sstable 参与 compaction。

2. 在此修复了一个 snapshot compaction 下数据不一致的 BUG：_github.com/google/level_



1. BUG 产生：随着 compaction 的不断进行，在有 snapshot 的情况下，可能会导致每一层中有许多按照 sequence number 排序的 user_key 相同的 record，如果这些 record 比较多或者对应的 value 比较大，那么这些 record 就会被分散保存到相邻的 sstable，从而触发这个 BUG。
2. BUG 修复：会调用 AddBoundaryInputs 函数添加同层的有和当前选取的 sstable 的 largest_key 的 user_key 相等的其他 sstable 参与 compaction。

```
// AddBoundaryInputs 主要包含以下两步
InternalKey largest_key = FindLargestKey(compaction_files);
FileMetaData* smallest_boundary_file =
FindSmallestBoundaryFile(level_files);
// 选取重叠的 sstable 的判断条件
if (icmp.Compare(f->smallest, largest_key) > 0 &&
 user_cmp->Compare(f->smallest.user_key(),
largest_key.user_key()) == 0) {
}
```

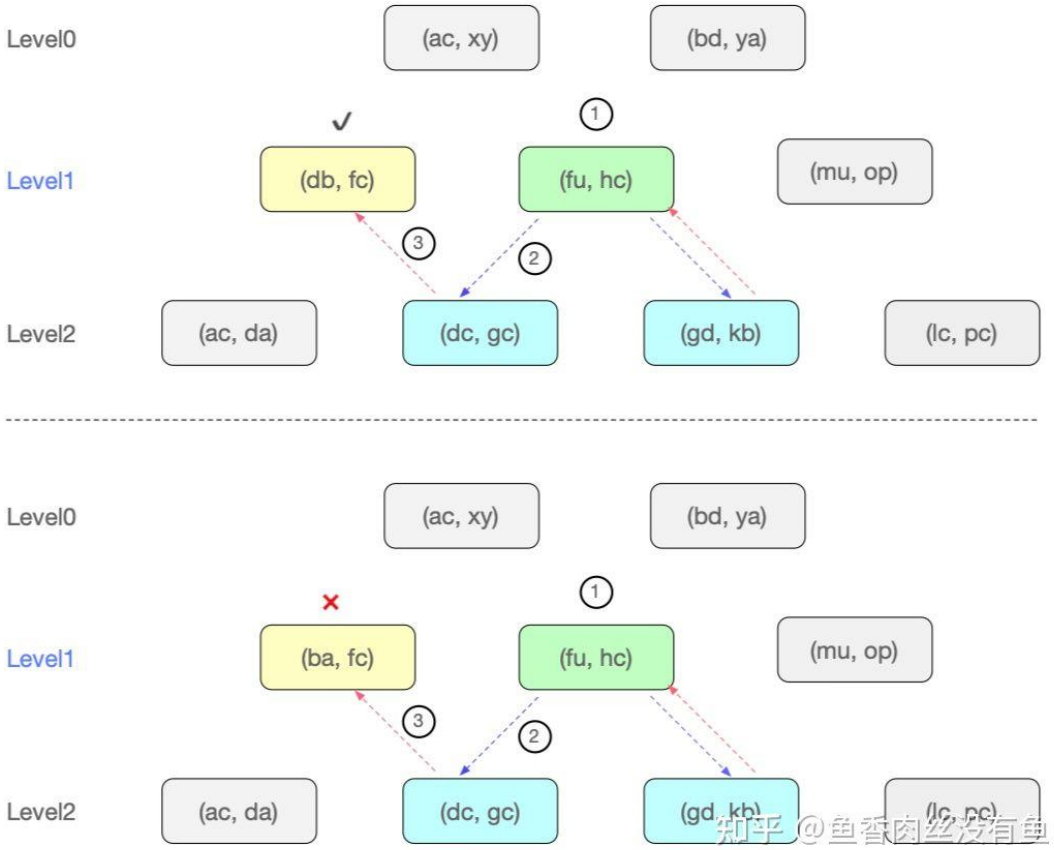
1. 如果是 level0，还需要选取所有与当前 sstable 重合的 sstable 参与 compaction，因为 L0 允许 sstable 之间重叠。

3、选取 level i+1 上的 sstable。

根据 level i 上选取出的 sstable，确定其[smallest, largest]，然后选出 level i+1 上与其有重叠的所有 sstable。

4、是否需要扩展 level i 上的 sstable。

1. 在已经选取的 level i+1 的 sstable 数量不变的情况下，尽可能的增加 level i 中参与 compaction 的 sstable 数量。
2. 总的参与 compaction 的 sstable 的大小阈值为 25 * max_file_size。
3. 计算出 level i 和 level i+1 的[smallest, largest]，然后计算出和 level i 上有哪些 sstable 重叠，如果 level i 上新增的 sstable 不会与 level i+1 上的非 compaction 的 sstable 重叠，则加入此次 compaction。



输出

level i 和 level i+1 上的 sstable compaction 后生成的 sstable 放在 level i+1 上，同时生成新的 version，删除之前参与 compaction 的 sstable。

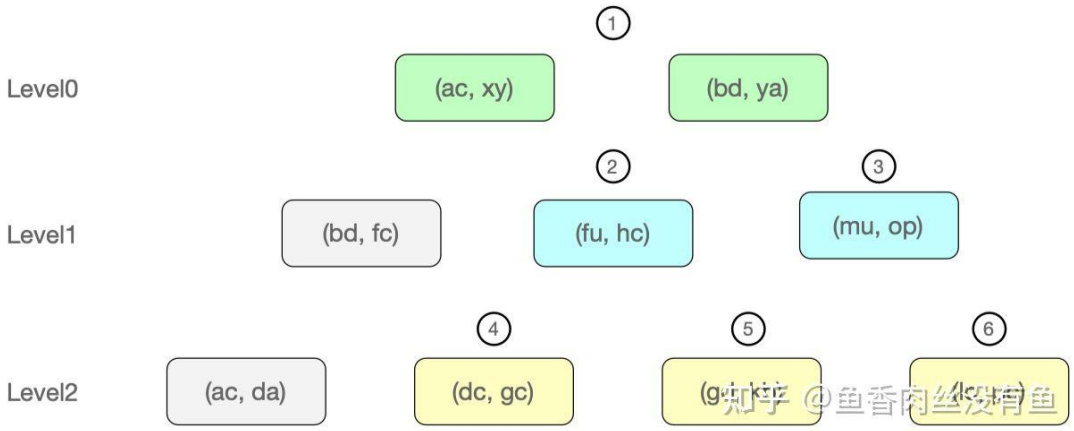
Manual Compaction

触发时机

外部调用 DBImpl::CompactRange(const Slice *begin, const Slice* end)时触发 manual compaction。

```
Slice begin("key_start");
Slice end("key_end");
db->CompactRange(&begin, &end);
// CompactRange 实现
TEST_CompactMemTable();
for (int level = 0; level < max_level_with_files; level++) {
  TEST_CompactRange(level, &begin, &end);
}
```

执行过程



manual compaction 会一个 level 一个 level 的 compact 所有与[begin, end]有重叠的 sstable，具体流程如下：

1. 写入一条 nullptr 的 batch 等待前面的 batch 结束从而尝试将 memtable 的数据 dump 成 sstable。
2. 从 level0 开始每层都依次执行 manual compaction。
    1. 在当前 level 选取与 range(begin, end)有重叠的所有 sstable，如果重叠的 sstable 过多则选择的 sstable 总大小不超过 MaxFileSizeForLevel(2MB)。
    2. 选取 level i+1 上的 sstable，流程和 Major Compaction - Pick SSTable - 选取 level i+1 上的 sstable 一致。

    3. 具体的 compaction 过程和 major compaction 一样，执行完后更新 manual compaction 的 begin 为 new_begin，循环执行 2.a，重新选择 sstable 做 compaction。
3. 循环执行下一层的 manual compaction，直到所有 level 执行完为止。

输出

同 major compaction，会导致产生很多 version 以及 sstable。

版本控制

LevelDB 使用 MVCC 来避免读写冲突，相比于锁机制，提升了读写效率。版本控制的主要作用为：

1. 记录 compaction 之后，DB 由哪些 SSTable 组成。
2. 记录哪些 SSTable 属于哪个 Version。

版本控制整体由以下几部分组成：

Version：管理当前 DB 元信息以及每一层的 SSTable 文件集合，引用计数为 0 时自动从 VersionSet 删除。

VersionEdit：记录 SSTable 文件的变化，新增的 SSTable 以及删除的 SSTable。

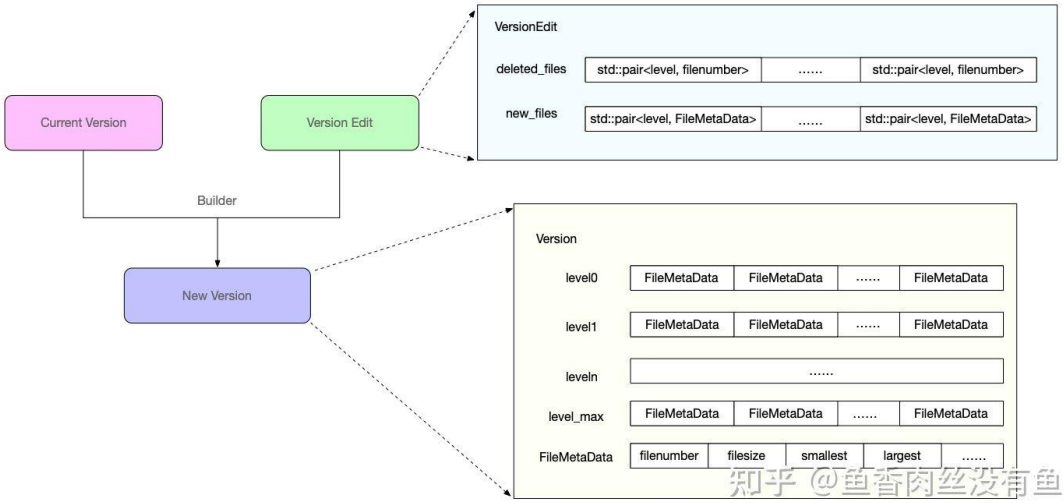
VersionSet：使用双向循环链表管理 DB 当前所有活跃的 Version，最后一个节点为 dummy_version。

Builder：将 VersionEdit 应用到 Version 的过程封装成 VersionSet::Builder。主要包含 Apply(+)、SaveTo(=)两个方法。

Manifest：持久化当前 DB 的状态以及每次的 VersionEdit。

Current：指向当前最新的 Manifest 文件。

Version + VersionEdit = NewVersion



VersionSet 应用 VersionEdit 生成新 Version 主要都在 **LogAndApply** 函数中，流程如下：

```
Version* v = new Version(this);
{
        Builder builder(this, current_);
        builder.Apply(edit);
        builder.SaveTo(v);
}
```

1. 生成新 Version 并应用 VersionEdit。
2. 调用 Finalize 计算下次 compaction 要处理的 Level。
3. 如果不复用老的 Manifest 文件则创建一个新的并且写入 Snapshot Record。
    1. 重启时在 **Recovery()/ReuseManifest()** 中会判断。
    2. 如果 Manifest 文件大于 TargetFileSize 则不复用老的 Manifest 文件，等到 LogAndApply 时创建新的 Manifest 文件。
    3. 如果 Manifest 文件小于 TargetFileSize 则复用老的 Manifest 文件，在 LogAndApply 中就可以跳过创建 Manifest 文件的步骤。
4. 将 VersionEdit 的内容作为 Session Record 写入 Manifest。
5. 如果新建 Manifest 文件则更新 Current 文件指向最新的 Manifest 文件。
6. 调用 AppendVersion 将最新的 Version 更新到 VersionSet。