leveldb 源码解析

2022-11-29 于长沙市拙鱼书房。

本文是我个人阅读 leveldb 源码时所作的笔记形式的解析，主要包括 leveldb 的启动、handle 读写请求、compaction 和 recovery 四个部分。由于我阅读源码的目的是以 leveldb 为蓝本，实现一个精简版的 lsm-tree based key-value database，因此本文缺乏 leveldb 的很多实现细节和重要模块的解析。

关于图解和其它本文中没有的解析，参考：

leveldb-handbook - leveldb-handbook 文档

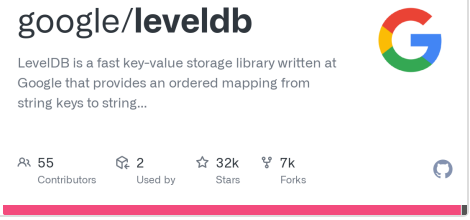
https://leveldb-handbook.readthedocs.io/zh/latest/index.html

leveldb 官方给出的一些解析：

leveldb/impl.md at main · google/leveldb

The implementation of leveldb is similar in spirit to the representation of a single Bigtable tablet (section 5.3). However the organization of the files that make up the representation is

https://github.com/google/leveldb/blob/main/doc/impl.md



其它可以阅读的解析：

透过leveldb感悟存储工程实践

leveldb是非常经典的key-value存储系统，在各种C++最佳实践推荐名单都排在前面。 通过leveldb，我们不仅能看到Google style的编程范式，也能获取到分布式存储系统许多经典问题的解决思路。 花点

知 https://zhuanlan.zhihu.com/p/516566364

`include/db.h` 文件中定义了一个抽象类 `DB`，其中定义了操作数据库的各种接口，包括 `Put`，`Delete`，`Get` 等，还提供了一个 static 函数 `Open` 用于创建内存中的 DB 实例。该抽

象类的实现 `DBImpl` 位于 `db/db_impl.h` 文件中 。

`db/db_test.c` 文件中定义了一系列测试函数，以其中的 `PutDeleteGet` 测试为例，分析 leveldb 的 put, delete 和 get。测试框架使用的是 google test fixture，继承自 `testing::Test` 的 `DBTest` 类的 ctor 会在 setup 阶段（每个测试执行之前）被调用，dtor 则会在 teardown 阶段（每个测试执行之后）被调用。 `DBTest` 的 ctor 会调用 `DB::Open` 函数创建一个 DB 实例。

启动

假设不需要进行 recover，即是首次创建这个数据库，基于此来观察 `Open` 这个函数的行为。首先 new 一个 `DBImpl` 实例，再创建内存中的 memtable 实例。由于每一个 memtable 都有一个 backed log file，因此需要首先创建一个 log file。为了区分不同的文件，db 内部通过 version set 维护了一个全局递增的 file num，每次创建一个新文件时，都会使用当前的 file num，在创建后再 increment file num。与文件有关的操作均涉及到操作系统，因此与运行的环境 env 有关，为此 leveldb 将所有与文件有关的操作都封装到了 `Env` 这个类中。对于 log file 的写入通过 `log::Writer` 完成。

manifest 是数据库的描述文件，描述了当前数据库的状态，包括内存和 disk 中的状态。内存中的状态包括 memtable 和 immutable memtable（以后简称 imm），disk 中的状态包括 sstables 和 log，每次修改这些状态后都需要更新 manifest。由于 `Open` 函数创建了新的 log file 和 memtable，因此需要更新 manifest 文件。leveldb 设计了 version, version edit 和 version set。version set 是 versions 的集合，其实现为一个循环双向链表，其内部有一个 `current_` 成员指向当前的 version。每个 version 内部保存了一个 mapping，这个 mapping 的 key 是 level num，value则是给定 level 中的所有 sstable 文件的 metadata。metadata 中保存了这个 sstable 文件的 file num（由于每个 sstable 文件的前缀相同，因此根据作为后缀的 unique file num 即可定位对应的 sstable 文件），以及存在于其中的 keys 的 key range。对于 version 的修改是增量式的，即将 version edit 应用到当前的 version 后，得到一个新的 version，这个新的 version 被添加到 version set 中，并成为当前的 `current_`。对于 manifest 文件的更新也是增量式的，即每次将 version edit 以 encoded record 的形式 append 到 manifest 文件的最后。

至此，成功地在内存中创建了一个 `DBImpl` 实例。

写请求

leveldb 会在内存中 buffer 一些 writes（put 或 deletes），在达到某些条件后，以 batch 的方式原子性地写入到 db 中。为了简化分析，假设每个 write batch 中只有一个 write。对于这个 write，其首先需要被写入到 log 中，再写入到 memtable 中。根据 leveldb 定义的 log format，这个 write 会被 encode 为一个 record，通过 log writer 写入到对应的 log file。leveldb 提供了一个 sync flag，如果其为 true，则 cache 在内存中的 log file 会在每次写入后 flush 到 disk。

当 log record 写入成功后，调用 `WriteBatchInternal::InsertInto` 将这个 write 写入到 memtable 中。这个过程经过了多层封装，简而言之，其会根据 write type，调用 put 或 delete 操作对应的函数。它们的区别在于，put 会将 user 传入的 value 作为写入的 value，而 delete 会将一个空的 slice 作为写入的 value。它们均调用 `Memtable::Add` 函数。

每一个 write 被表示为一个 key-value entry，由 key length, key data, type, value length, value data 五个部分连接组成。其中 key 的组成为 user key, sequence num 和 type。由于 leveldb 底层为 lsm tree，每个 key-value 被 append 到 db 中，而无需检测 duplicates。因此为了区分同一个 key 对应的不同数据项的新旧，给每一个 key 绑定一个全局递增的 sequence num。需要指明 length，是因为 leveldb 支持变长的 key 和 value。 `Memtable::Add` 函数在内存中构建了这样的一个 entry，再将其添加到 memtable 中。memtable 的底层数据结构是一个 skiplist。leveldb 设计了一个 `Arena` 类以更好地控制 skiplist 中内存的分配与释放。

至此，成功地在 log 和 memtable 中添加了一组 key-value entry。这些 buffer 在内存中的 writes 会在之后的某个时刻，由后台线程执行 compaction，此时才写入到磁盘中。

读请求

leveldb 使用 MVCC 进行并发控制。在每一次 get 调用时，将最新的 sequence num 作为 snapshot num，即此次读取的数据库的 snapshot 由所有 sequence num ≤ snapshot num 的 key-value entries 组成。

首先调用 `Memtable::Get` 搜索内存中的 memtable。leveldb 为 skiplist 设计了一个 iterator，对于 memtable 的搜索通过这个 iterator 进行。如果根据 comparator 判断找到了一个 entry，其 user key 与此次 get 给定的 user key 相等（iterator 内部的逻辑保证所找到的 entry 对应的 sequence num ≤ snapshot num），再检查该 entry 中所存的 type。如果是 put，则说明找到了。如果是 delete，则说明对于当前的 snapshot 而言，该 key 已删除，即不存在。

如果 memtable 中没找到，则去 imm 中找。imm 的类型也是 `Memtable`，只不过其为一个 frozen memtable。每次进行写入操作之前，会调用 `DBImpl::MakeRoomForWrite` 检查是否需要创建一个新的 memtable。如果当前的 memtable 的 approximate size 超过了预设的阈值（approximate size 由 `Arena::MemoryUsage` 函数给出，其返回当前 arena 已经 allocated 的内存）则会创建一个新的 memtable，所有的写请求被重定向到这个新的 memtable，而旧的 memtable 不再接受写请求，只接受读请求，成为一个 frozen （或称 immutable）memtable。imm 会在之后的某一时刻通过 compaction 被 flush 到 disk 中，转化为一个新的 sstable 文件。由于 compaction 是由后台线程进行的，因此在创建一个新的 memtable 时，可能尚存在一个 imm，此时需要等待后台线程完成 compaction。如果不使用 imm，而是在每次 memtable 容量满了之后立即进行 flush，则在等待 flush 完成的期间，所有写请求会被 block。

如果 imm 中也没有找到，则需要调用 `Version::Get` 搜索 disk 中的 sstables。version 中维护了 sstables 的所有 metadata，其中保存了每一层内 sstables 的 key range。由于 compaction 是 top-down，因此 level num 越小，数据越新，故搜索首先从 level 0 开始。由于 level 0 中 sstables 的来源是 memtable，在 flush 到 disk 中时，不会进行 key range 的合并，因此 level 0 中的不同 sstables 可能存在 overlapping 的 key range，故需要特别处理 level 0。

根据给定的 user key，以及 metadata 中的 key range，收集所有与 user key overlap 的 sstables 的 metadata。根据 metadata 中保存的 file num 进行排序，使得搜索从最新（即 file num 最大）的 sstable 开始。leveldb 在内存中维护了一个 table cache，它是关于 LRU cache 的 wrapper，提供了包括 `Get` 在内的一些接口，用于访问和操作 LRU cache。LRU cache 的 key 是 file num，value 的类型是 `TableAndFile`，该类型包含一个 `RandomAccessFile`，指向在内存中打开的 sstable 文件，还包含一个 `Table` 。一个 `Table` 是 `Rep` 的 wrapper， `Rep` 是 table 的 representation，维护一个 sstable 在内存中的相关信息。对于外界而言， `Table` 就是一个 sorted map，即 sorted key-value entries 的集合。

table cache 用于缓存 `TableAndFile`，但不缓存 blocks。对于 blocks 的缓存，由 block cache 完成。其也为一个 backed LRU cache 的 wrapper。

leveldb 首先以 file num 为 key 去 table cache 中 lookup，如果发现对应的 sstable 并没有被 cache 在内存的某个 table 中，则会打开 disk 中对应的 sstable，并在内存中创建一个 table 实例，用于维护和访问这个 sstable。同时，会在 table cache 中插入对应的 cache entry。

一个 sstable 文件依次包含 data blocks, filter block, meta index block, index block 以及 footer。除了 data blocks 部分包含多个 blocks，其他仅分别占用单个 block。

`Table::Open` 函数会先将 footer 读入到内存，一个 footer 的长度为固定的两个 block size，同时 sstable 是 immutable 的，在 memtable flush 到 disk 后便不会再改变，因此根据 sstable 的文件大小，从尾端往前推两个 block，即可读取到 footer。footer 的作用类似 header，保存了 sstable 中其他部分的位置和大小信息。leveldb 使用 footer 而不是 header 的原因，是因为 data block 从 sstable 的头部开始存储。根据 footer 中保存的信息，再读取 index block 和 meta index block，前者保存 data blocks 的索引信息，后者指明 filter block 在 sstable 中的 offset，用于读取 filter block。每一个 block 在 disk 中就是一连串的 bytes，在内存中，有一个对应的 `Block` 类，其包含一个 `data_` 成员，指向 cache 在 block cache 中的 block 的起始地址，此外该类还会维护一些其它信息。它们的关系类似 buffer pool 中的 frame 和 page。

在内存中成功读取到某个 file num 对应的 table 实例后，则会调用 `Table::InternalGet` 在该 table 中搜索给定的 key。leveldb 根据之前读取到的 index block，创建一个 iterator。index block 中包含一系列的 index entries，每个 entry 对应一个 data block，其组成为 max key, data block offset, data block size。max key 表示该 data block 中包含的最大的 key，其它二者则用于在 sstable 中定位这个 data block。iterator 利用 max key，判断给定的 key 所在的 data block。如果找到了这样的一个 data block，则将该 data block 读取到内存中，再创建一个关于 data block 的 iterator，利用它在 data block 内查找给定的 key。为了加速在 data block 内的查找，leveldb 为每个 data block 中所存储的 key-value entries 构建了对应的 bloom filter，这些 bloom filters 存储在 filter block 中。简化而言，一个 filter block 由一系列的 filter entry 组成，第 i 个 filter entry 存储了第 i 个 data block 对应的 bloom filter 序列化后的数据。

leveldb 设计的 data block 包含 data, compression type, crc checksum 三个部分。其中 data 部分又包含 key-value entries, restart points, restart point length 三个部分。在 data block 内查找给定的 key 的过程比较复杂，可以认为其根据 key 的有序存储特性，使用二分查找快速定位 key 可能存在的位置，再进一步判断 key 是否存在。（当然，实际上并不是这样）

对于 level 0 的查找，需要如上所述找到与给定的 key overlap 的所有 sstables 中。而对于 level 1 以及之后的 levels，由于 sstables 的 key range 不会 overlap，则可以利用 version 中保存的 file metadata 中的关于每个 sstable 所存储的最大的 key，使用二分查找快速定位到 key 可能所在的 sstable，再根据上述流程在 sstable 内查找。

至此，关于给定的 key 的查找过程结束。

Compaction

关于 compaction，需要思考如下几个问题：

- 何时做 compaction？
- 根据预期的 workload 和给定的 performance 指标，考虑 read, write, space amplification，应该选择什么 compaction strategy？leveled, size-tiered, hybrid, etc.
- 选择哪个或哪些 sstables 做 compaction。具体而言，当某一个 level 中 sstables 的数量或者总大小超过了某个阈值，应该选择该 level 中哪个 sstable 与下一层的 sstables 进行 merge。

关于不同的 compaction strategy，参考：

leveldb 在多个不同情况下会调用 `DBImpl::MaybeScheduleCompaction`，该函数会做一些判断，例如当前是否正在做 compaction，如果该函数判断当前可以做 compaction，则会向后台线程推入一个 compaction work，后台线程会异步地调用 `DBImpl::BackgroundCompaction` 函数执行 compaction。

在执行 compaction work 时，如果当前有一个 imm，则会调用 `DBImpl::CompactMemtable` 优先将 imm flush 到 disk。一方面，该函数会调用 `DBImpl::WriteLevel0Table` 执行 sstable 文件的创建和 imm 的销毁。另一方面，该函数会更新 version。sstable 文件的创建由 `BuildTable` 函数完成。该函数会创建一个 `TableBuilder` 实例，再从由 imm 创建的 iterator 中，依次读取内存中的 key-value entries，调用 `TableBuilder::Add` 函数写入这些 entries。

table builder 内部维护了一个 `TableBuilder::Rep`，用于在内存中维护一个 sstable 相关的信息。对于每一个 key-value entry，`TableBuilder::Add` 函数会向内存中的 data block 中写入一条 key-value entry（根据 data block 中指定的 key-value entry 格式），如果指定了 filter，例如 bloom filter，则还会向内存中的 bloom filter 中写入该 key-value entry 对应的 filter 数据。当写入的 key-value entries 超过了一个 data block 可以容纳的量时，向内存中的 index block 写入一条 index entry，同时将该 data block flush 到磁盘上。除了

向内存中的 sstable 写入数据，`BuildTable` 函数还会向内存中的 table cache 写入数据，例如 file metadata 中所需的 key range。

当所有的 key-value entries 以 data block 的形式写入到 disk 中的 sstable 之后，调用 `TableBuilder::Finish` 函数依次将 filter block, meta index block, index block 和 footer 写入到磁盘中。

至此，磁盘中的 sstable 写入完毕。之后，需要将该 sstable 添加到逻辑上的 lsm tree 中。理论上，一个从 memtable 转变过来的 sstable 应该待在 level 0，但是 leveldb 在此处做了一个改动。它会检查这个新的 sstable 是否与 level 0 层已有的 sstables 在 key range 上有 overlap，如果没有则将这个新的 sstable 推入到 level 1 层。之后，如果这个 sstable 与下一层的 sstables 不存在 overlap，则继续往下推。

这样做的理由是：如果当前层达到了 sstables 的文件数或者文件总大小限制，则会引发当前层与下一层的 compaction。如果 compaction 之后，下一层也达到了限制，则会继续 compaction，即 cascading compaction。对于当前层的 compaction，会有一些来自当前层的 input files，即参与 compaction 的 sstables。根据这些 input files，选择下一层与它们存在 overlap 的 sstables 添加到 input files 中。如果我们在添加一个 sstable 到 lsm tree 中时，不做上述的下沉操作，则在 cascading compaction 时，从上往下，可能会卷入很多 input files，导致参与 compaction 的 sstables 过多，阻塞对于这些 sstables 的读请求。

实际上，做这样的下沉操作，违反了 lsm tree 从上往下的数据新旧关系。由于读请求会依次访问 memtable, imm 和 lsm tree，因此下沉操作本身也会影响读请求的效率。但是 leveldb 可能考虑到 cascading compaction 对读请求的影响更大，因此选择在每次添加一个新的 sstable 到 lsm tree 时，均尝试做这样的下沉操作。

以上为 `DBImpl::BackgroundCompaction` 函数检查到存在一个 imm 时所做的操作。如果此时没有 imm，则该函数会调用 `VersionSet::PickCompaction` 函数，选择一个 level 以及该 level 中的某些 sstables 作为 compaction 的初始 input files。影响选择的指标有两个，一个是 compaction score，另一个是 sstable 的无效 seek 次数。

对于 compaction score 指标，在每次 compaction 结束以后，leveldb 会调用 `Finalize` 函数计算最大的 compaction score。在计算过程中，每一层有一个 compaction score。对于 level 0，compaction score 与 sstables 的文件数量有关，文件数量越多，compaction score 越高。对于其它层，compaction score 与 sstables 的文件总大小有关，该层文件的总大小与该层预设的最大文件大小的比值越高，compaction score 越高。compaction score 最高的那一层，成为下一次 compaction 的起始层。

对于无效 seek 次数，leveldb 会在内存中为每个打开的 sstable 分别维护一个 `allowed_seek` 变量。当处理读请求时，对于某一层（除 level 0 层外），由于其为一个 sorted run，因此只需要根据 metadata 中所存的 sstable max key，使用二分查找即可快速定位到可能含有给定的 key 的 sstable，再使用一次 disk seek 即可开始读取该 sstable。由于 lsm tree 不同层之间存在 key range 的 overlap，因此如果当前层没有找到，还需要往下一层继续找。在这个查找路径中，存在很多无效 seek。由于 disk seek 耗时较大，当无效 seek 变多，leveldb 的读性能会下降。由于一个 sstable 文件在此过程中引发一次 disk seek，因此只要 sstable 文件的数量减小，则无效的 disk seek 次数便会减小，从而提高读性能。

compaction 是 leveldb 唯一可以减小 sstable 数量的方法。但是执行 compaction 时会阻塞对于参与此次 compaction 的 sstables 的读取，会降低读性能。我们可以对于每一次无效的 disk seek 均执行一次 compaction，以最大化地消除无效 disk seek 对读性能的影响，但同时也会由于频繁的 compaction 从而降低读性能。显然，我们需要权衡 compaction 与无效的 disk seek 对读性能的影响。

leveldb 选择了这样一种平衡策略：等待关于某个 sstable 的无效 seek 积累到一定次数再做 compaction。这个次数的值是这样的：在达到这个次数之前，无效 disk seek 所带来的读取时的总延时小于一次 compaction 的耗时，因此不做 compaction 性能更好。在达到这个次数之后，总延时高于一次 compaction 的耗时，此时应该选择做 compaction，消除后续可能的无效 seek 对读性能的影响。

那么如何确定这个次数呢？考虑极端情况，当前层的所有 sstables 均参与到 compaction 中。再考虑极端情况，这些 sstables 的 key range 与下一层所有 sstables 的 key range 存在 overlap。那么此次 compaction 需要读取当前层的所有数据以及下一层的所有数据。假设当前层的所有数据的总长度为 1 个 block，由于从 level 1 层开始，每一层的最大容量是上一层的 10 倍，那么下一层的所有数据的总长度为 10 个 blocks。由于可能没有边界对齐，长度为 N 个 blocks 的连续数据可能跨越 N + 1 个 blocks。因此在上述极端情况的 compaction 中，读取当前层一个 block 长度的数据，实际需要读取 2 个 blocks。而读取下一层 10 个 blocks 长度的数据，实际需要读取 11 个 blocks。因此总共需要读取 2 + 11 = 13 个 blocks。在写入时，最坏情况下总共需要写入 1 + 10 = 11 blocks 长度的数据，即虽然 key range overlap，但是没有 deleted keys，故 compaction 后所有数据均需要保留。假设写入时也没有进行边界对齐，故 11 个 blocks 长度的数据，实际需要对 12 个 blocks 进行写入。则极端情况下一次 compaction 需要 13 + 12 = 25 个 blocks 的读取。

将以上的 block 单位置换为 MB 单位，可以认为 1 MB 数据在 compaction 时，最多需要 25 MB 的 disk io。leveldb 假定 disk io 速度均为 100 MB/s，则读写 25 MB 数据耗时

250 ms。leveldb 还假定一次 disk seek 需要 10 ms，则得到这样一个关系：1 次 disk seek 的耗时等同于对 40 KB 的数据执行 compaction 的耗时。那么对于一个大小为 L KB 的 sstable 文件，其应该选择的最大可容忍的无效 seek 次数，应该为 L / 40。leveldb 认为 compaction 实际带来的性能降低应该更大，因此选择了一个更保守的数值，即 L / 16，如此可容忍的最大无效 seek 次数就会更多，compaction 也执行地更不频繁。同时，考虑到当 write_buffer_size 设置的过于小时，sstable 文件就会很小，因此可容忍的最大无效 seek 次数会很低，则 compaction 会执行地很频繁。为此，leveldb 规定最小的可容忍的无效 seek 次数为 100。

在 leveldb 中，尽管一次 compaction 可能引发 cascading compaction，但是不同层的 compaction 实际上中间可能存在 gap，即做完 level i 层 和 level i+1 层的 compaction 后，即使发现 level i+1 和 level i+2 层还需要做 compaction，但是由于那是另一个不同的 compaction work，因此选择不立即做。 一个 `Compaction` 类的实例即对应两层之间的 compaction。

每一次 compaction，设当前层为 L0 层，下一层为 L1 层。compaction score 指标对应 size compaction，无效 seek 次数指标对应 seek compaction。对于 size compaction，我们选择 compaction score 最大的那一层作为 L0 层。leveldb 为每一层维护了一个 compaction pointer，其为一个 key，这个 key 是上一次参与该层 compaction 的所有 key 的最大 key。对于此次 compaction，我们按照 max key 升序的顺序便利当前层的所有 sstables，找到第一个 max key 比这个 key 大的 sstable 作为 L0 层初始 input file。如果没有找到，则选择该层的第一个 sstable。对于 seek compaction，直接选择无效 seek 次数降为 0 的那个 sstable 作为初始 input file，相应地，L0 层即该 sstable 所在的 level。当 L0 为 level 0 层时，由于 level 0 层的 sstables 之间存在 overlap key range，因此会选择与初始 input file 有 key range overlap 的所有 sstables 作为 L0 层的 input files。

之后，调用 `VersionSet::SetupOtherInputs` 添加其它的 input files。该函数首先调用 `AddBoundaryInputs` 向右扩展 L0 层 input files。这个扩展的意思是：假设当前 L0 层的 input files 的最右端文件（即 max key 最大的那个文件）的最后（即最大）那个 key-value entry 的 user key，还存在于 L0 层其它文件（设为 F）的起始 key-value entry 中。即虽然这两个 entry 的 user key 相等，但是 sequence number 不相等。那么文件 F 也需要被添加到 L0 层 input files 中。如果文件 F 的最大 key 也有类似的情况，那么继续向右扩展。这样做的原因是，假设最右端的那个 key-value entry 实际上更新，而我们只把原有的 input files compact 到下一层或更底层，那么比它更旧的那个存在于文件 F 中的 key-value entry 会被之后的 get 操作读取到，从而返回错误的数据。因为 lsm-tree 的数据从上往下按新旧排列，在某一层找到数据之后，便不会再往更低层去找。

在 L0 层扩展完毕之后，根据 L0 层的 key range，找到 L1 层存在 overlap key range 的所有 sstables，将它们作为 L1 层 input files。之后同样利用 `AddBoundaryInputs` 函数向右扩展 L1 层的 input files。

至此，从上往下根据 overlap key range 的流程结束。但是可能还存在从下往上 key range overlap 的情况，因此还需要根据 L1 层 input files 的 key range，反向地在 L0 层找 overlap key range 的 sstables，将它们添加到 L0 层的 input files 中。最后，再向右扩展 L0 层的 input files。

当所有的 input files 添加完毕之后，调用 `DBImpl::DoCompactionWork` 执行真正的 compaction。这里的过程有些复杂，简化而言，leveldb 会利用所有的 input files，构建一个 iterator，key-value entries 按照 key 升序排列，key 相同、sequence num 不同的 entries 被聚簇在一起，按照 sequence num 降序排列，即更新的 entries 排在前面。在读取的过程中，记录上一次读取到的 entry 的 key。如果此次读取的 entry 的 key 与上一次想等，则说明这个 entry 更旧，因此可以直接跳过该 entry。如果一个 entry 没有被跳过，则会调用 `TableBuilder::Add` 函数将其添加到内存中的 sstable 中。相关流程之前已详细描述过。当所有的 entries 都添加完毕后，将 compact 之后的 sstable 写入到磁盘。最后将其插入到对应的 level，更新 version，并做一些清理工作，例如销毁不再使用的 sstables。

至此，两层之间的 compaction 执行完毕。在 `DBImpl::DoCompactionWork` 函数的最后，会调用 `DBImpl::InstallCompactionResults` 函数，该函数除了做些 version 更新操作之外，还会调用 `VersionSet::LogAndReply` 函数。这个函数则会调用 `Finalize` 函数，计算 compaction score，为下一次 compaction 做准备。因此如果本次 compaction 引发了 cascading compaction，则在之后的某个时刻，leveldb 会按照上述流程继续执行 cascading compaction。当然，这个过程可能并不是连续的，因为 compaction 是由后台线程异步执行，且中间可能会有 imm 的 flush 或由于 compaction score 的改变转而从其它层开始执行 compaction。

为什么 level 0 用文件数量计算 score，而其它层用文件总大小计算 score 呢？

一方面，leveldb 为每一层都预设了一个不可 config 的最大容量，level 0 和 level 1 都是 10 MB，从 level 2 开始，每一层的最大容量是上一层的 10 倍。实际上，level 0 层的最大容量并没有使用，而是给 level 0 层加了一个最大文件数量的限制，其值为 4，且不可 config。另一方面，leveldb 提供了一个可 config 的 options，其中包含一个 write_buffer_size 变量，默认值是 4 MB。

当我们使用默认 4 MB 的 write_buffer_size 时，level 0 层的理论文件总大小为 4 * 4 MB = 16 MB。这使得可以在 level 0 层存储更多的数据（与使用最大容量 10 MB 相比），降低 level-0 compaction 的频率。这就对应了 leveldb 在 comment 中给出的第一条理由：With larger write-buffer sizes, it is nice not to do too many level-0 compactions.

leveldb 在选择 compaction 的 input files 时，会将当前层与某个 sstable 有重叠 key 的 sstables 添加到 input files 中。对于当前层的每个 input files，再去下一层选择有重叠 key 的 sstables。这个选择过程的耗时显然只与文件数量有关，而与文件大小无关。由于 level 0 层的 sstables 之间存在重叠 key，因此当 level 0 层的文件数量增多时，上述过程的耗时会增加。考虑极端情况，当 level 0 层的文件数量过多，由其引发的 cascading compaction 的耗时会显著增加。

为了减少这个耗时，最直接的做法就是限制 level 0 层的文件数量。如果使用最大容量去限制 level 0 层，当我们降低 write_buffer_size 时（因为其 configurable），则 level 0 层的文件数量会增加，导致性能降低。这对应了 leveldb 在 comment 中给出的第二条理由：The files in level-0 are merged on every read and therefore we wish to avoid too many files when the individual file size is small (perhaps because of a small write-buffer setting).

至于以上所述的选择过程的耗时，是否真的会因为 level 0 层的文件数量的增多，而显著增加，且这个增加的幅度会显著降低性能，不得而知。我感觉这只是 leveldb 和 rocksdb 的一个 heuristics。

最后的思考：为什么不能完全摒弃使用最大容量限制某个 level 的做法，转而为每一层设置一个最大文件数量的限制呢？我的猜想是：如果为每一层分别设置一个固定的最大文件数量、而不限制最大容量，那么当我们减小 write_buffer_size 时，每一层的最大容量会降低。对于数据量一定的 workload 而言，compaction 会更频繁地下沉，这应该会降低性能。从另一个角度来猜想，leveldb 应该是秉着参数最小影响的原则来设计限制策略的。如果限制最大文件数量，则修改 write_buffer_size 参数会影响到 lsm tree 每一层。而如果限制 level 1 层及以上层的最大容量，则只会影响到 level 0 层。这使得更多的代码空间是可控的。

Recovery

在启动数据库时，`DB::Open` 函数会调用 `DBImpl::Recover` 执行 recovery。由于一台机器上可以运行多个 leveldb，而每个 leveldb 根据名字进行标识，因此有可能多个线程同时尝试打开同一个数据库。故需要进行并发控制。一个普遍的做法是为每个 leveldb 设置一个 lock file，在启动这个 leveldb 之前，需要先拿到这个 lock file 的锁，在关闭 leveldb 时再

释放这个 lock file 的锁。这里涉及的 file locking/unlocking 通过操作系统提供的接口执行。

每一个 leveldb 在运行过程中的状态变化由 manifest 文件描述，该文件使用 append version edits 的方式增量式地记录数据库状态的变更。通过 playback 这些 version edits，可以得到数据库在 crash 之前已记录的持久化的状态（可能有一些状态尚未来得及持久化，或者已经持久化但是尚未来得及往 manifest 文件中写入对应的记录）。每次启动时会创建一个新的 manifest，其内容会继承之前的那个 manifest 文件（如果之前存在 manifest 文件的话）。由于可能存在多个 manifest 文件，因此 leveldb 还提供了一个 current 文件，里面存储着最新的 manifest 文件的 file num，基于此可定位最新的那个 manifest 文件。

数据库的状态包括内存中的 memtable, imm 的状态，和磁盘中的 log, sstables 的状态，以及一些其它信息。磁盘中的状态已经持久化，因此只需要在 version 中找到对应的 file nums。version 中存储了 `log_number_` 和 `prev_log_number_`，分别对应 memtable 的 log file 和 imm 的 log file，还存储了每个 level 内的 sstables 的 file nums。当我们 playback version edits，重构出 version 后，即可拿到这些信息，从而在磁盘中找到对应的文件。至此，数据库在磁盘中的状态恢复完毕。对于数据库在内存中的状态，即 memtable 和 imm，需要读取刚才找到的 memtable 和 imm 的 log files，通过 playback 其中存储的 log records，重构 memtable 和 imm。至此，数据库在内存中的状态恢复完毕。

从 current file 中拿到 manifest file num 后，`DBImpl::Recover` 调用 `VersionSet::Recover` 函数读取 manifest 文件。manifest 文件中的 version edits 以 log record 的形式存储，每个 log record 包含一个或多个 log blocks。每个 log block 的大小固定为 32 KB，以分块的方式存储能够提高读写的效率。一个 log block 由 header 和 data 两个部分组成，前者又包含 checksum, data length 以及 log block type。checksum 用于校验，检测数据是否在存储或读取过程中被污染。data length 指明了 data 部分所存的有效数据的长度。因为 log block 大小固定，而 data 大小不定，因此 data 部分的尾端可能会空出一些未使用的空间。

为了指明一个 log record 的起始 block，中间 block 和 结束 block，需要指明每个 log block 的 type，包括 `kFullType`, `kFirstType`, `kMiddleType` 和 `kLastType`，分别表明这个 log block 是一个 log record 的全部，首个，中间，最后那个 log block。对于 version edits，分界 log blocks 并没有作用，而对于之后提到的 write batch，分界 log blocks 是必须的。由于 leveldb 使用统一的 log record 形式存储 manifest 文件和 log file，因此对于 version edits 对应的 log record，也需要按照分界的方式读写。

每次以 log block 为单位，读取 manifest 文件。根据 log block type，界定出一个 log record。每个 log record 被反序列化为一个 version edit。play back 所有读取到的

version edits，即可重构出 version。从 version 中拿到 `log_number_` 和 `prev_log_number_` 后， `DBImpl::Recover` 调用 `DBImpl::RecoverLogFile` ，读取 log files 中的 log records。此时每一个 log record 对应一个 write batch。在之前关于写请求的解析中，我们忽略了 write batch。实际上，多个 writes 会被 buffer 在内存中，之后再以 batch 的形式原子性地写入 memtable，进而写入到磁盘中。这是为了 amortize disk io 所带来的开销。在写入 memtable 之前，将 write batch 以一个 log record 的形式写入到 log 中。

由于 leveldb 只提供了 write batch 的写入接口，没有提供单个 write 的写入接口，因此我们需要将一个 log record 作为一个整体，反序列化得到 write batch，再利用 write batch 的写入接口，play back 这些 writes。leveldb 按照从旧往新的顺序读取 log files，即首先读取 imm 的 log file，从其中提取出 log record，反序列化得到 write batch，再调用 `WriteBatchInternal::InsertInto` 插入到 memtable 中。注意，这里并没有直接插入到 imm 中，而是遵循了 imm 的 immutable 的特性，先插入到 memtable 中，再按照正常的流程判断 memtable 的容量是否满了，如果满了则创建一个 imm。imm 的 log file play back 完毕后，再读取 memtable 的 log file，重复以上的 play back 流程。至此，数据库在内存中的状态恢复完毕。

leveldb 在 crash recovery 过程中以及在 compaction 之后，可能会产生 obsolete files。leveldb 会在 recovery 和 compaction 成功之后，调用 `RemoveObsoleteFiles` 函数将这些过时的旧文件删除。