# 背景

Log Structured Merge Tree，简称LSM-Tree。2006年，Google发表了 BigTable的论文。这篇论文提到BigTable单机上所使用的数据结构就是LSM-Tree。  
 很多存储产品使用LSM-Tree作为数据结构，比如 Apache HBase，Apache Cassandra，MongoDB的Wired Tiger存储引擎，LevelDB存储引擎，RocksDB存储引擎等。  
 简单地说，**LSM-Tree的设计目标是提供比传统的B-Tree/B+Tree更好的写性能。LSM-Tree通过将磁盘的随机写转化为顺序写来提高写性能 ，而付出的代价就是牺牲部分读性能、写放大（B-Tree/B+Tree 同样有写放大的问题）**。

## 如何优化写性能

如果我们对写性能特别敏感，我们最好怎么做？

Append only：所有写操作都是将数据添加到文件末尾。这样顺序写的性能是最好的，大约等于磁盘的理论速度（无论是SSD还是HDD，顺序写性能都要明显由于随机写性能）。

注：GFS就是采用append only。

但是append only的方式会带来一些问题：

**1、不支持有序遍历。**

**2、需要垃圾回收（清理过期数据）。**

所以，纯粹的append only方式只能适用于一些简单的场景：

1、存储系统的 WAL。

2、能知道明确的offset的查询，比如 Bitcask。

注：HDFS，Kafka都是利用append only提高写效率的。

## 如何优化读性能

如果我们对读性能特别敏感，一般我们有两种方式：

有序存储，比如B-Tree/B+Tree。但是B-Tree/B+Tree会导致随机写。

哈希存储：不支持有序遍历，适用范围有限。

## 读写性能的权衡

如何获得接近append only的写性能，而又能拥有不错的读性能呢？以LevelDB/RocksDB为代表的LSM-Tree存储引擎给出了一个参考答案。  
 LevelDB 的写操作（Put/Delete/Write）主要由两步组成：

1、写日志（WAL，顺序写）。

2、**写MemTable（内存中的SkipList）**。

所以，正常情况下，LevelDB的写速度非常快。  
 内存中的 MemTable写满后，会转换为Immutable MemTable，然后**被后台线程compact成按key有序存储的SSTable（顺序写）**。  
 SSTable按照数据从新到旧被组织成多个层次（上层新下层旧），点查询（Get）的时候从上往下一层层查找，所以LevelDB的读操作可能会有多次磁盘IO（**LevelDB通过table cache、block cache和bloom filter等优化措施来减少读操作的I/O次数**）。  
 后台线程的定期compaction负责回收过期数据和维护每一层数据的有序性。在数据局部有序的基础上，LevelDB实现了数据的（全局）有序遍历。

# 概述

LevelDB是Google的Jeff Dean和Sanjay Ghemawat设计开发的key-value存储引擎。LevelDB底层存储利用了LSM tree的思想，RocksDB是Facebook基于LevelDB开发的存储引擎，针对LevelDB做了很多优化，但是大部分模块的实现机制是一样的。

**LevelDB是一个持久化存储的KV系统，和Redis这种内存型的KV系统不同，LevelDB不会像Redis一样狂吃内存，而是将大部分数据存储到磁盘上。**LevleDB在存储数据时，是根据记录的key值有序存储的，就是说**相邻的key值在存储文件中是依次顺序存储的**，而应用可以自定义key大小比较函数，LevleDB会按照用户定义的比较函数依序存储这些记录。

像大多数KV系统一样，LevelDB的操作接口简单，基本操作包括写记录、读记录以及删除记录。另外，LevelDB支持数据快照（snapshot）功能，使得读取操作不受写操作影响，可以在读操作过程中始终看到一致的数据。除此之外，LevelDB还支持数据压缩等操作，这对于减小存储空间以及增快IO效率都有直接的帮助。

参考：

<https://zhuanlan.zhihu.com/p/203578801>

<https://www.lmlphp.com/user/8275/article/item/377231/>

## 特点

作为一个函数库，LevelDB有以下特点：

键和值都是任意的字节数组；

数据以键的顺序存储；

可以使用定制的比较器定义排序的方式；

基本操作是Put(key,value)、 Get(key)、Delete(key)；

一个原子的Batch中可以做多个改变；

支持Snapshot，用户可以建立一个一致性的视图；

可以正向和反向迭代；

支持snappy压缩数据；

操作系统相关操作被抽象，跨平台时可以定制这些接口；

同一时间只能由一个进程访问数据库，但是支持多线程访问。

## 功能特性

## 对比

RocksDB与LevelDB对比：

1. 增加了column family，这样有利于多个不相关的数据集存储在同一个db中，因为不同column family的数据是存储在不同的sst和memtable中，所以一定程度上起到了隔离的作用。
2. 采用了多线程同时进行compaction的方法，优化了compact的速度。
3. 增加了merge operator，优化了modify的效率。
4. 将flush和compaction分开不同的线程池，能有效的加快flush，防止stall。
5. 增加了对write ahead log(WAL)的特殊管理机制，这样就能方便管理WAL文件，因为WAL是binlog文件。

# 编译安装

要使用LevelDB，必须先要编译安装函数库。LevelDB使用了cmake，需要安装cmake，可以根据需要，安装snappy、crc32c和tcmalloc。通过执行以下几个步骤，就可以安装函数库了：

mkdir -p build && cd build

cmake -DCMAKE\_BUILD\_TYPE=Release .. && cmake --build .

make install

# 源码

根目录下面有几个主要的目录：

include: 函数库的头文件

port: 可移植性相关的功能

util: 项目用到的一些功能函数

table: SSTable的实现

db: 数据库实现，版本管理，Compaction，WAL和MemTable实现

LevelDB作为函数库，对外提供的接口文件及功能如下：

cache.h: 缓存接口，提供了默认的LRU缓存，也可以自己实现缓存

comparator.h: 定以数据库比较器的接口，用来比较键，可以使用默认的基于字节的比较，可以定义自己的比较器

dumpfile.h: 以可读文本形式导出一个文件，调试使用

export.h: 可移植性相关

iterator.h: 迭代器接口

slice.h: 实现一个字符串，存储指针和长度，指向字符串

table\_builder.h: 构造一个SSTable

write\_batch.h: 实现批量写入的接口

c.h: 实现C语言相关的接口

db.h: 操作数据库的主要接口

env.h: 定义操作系统相关的功能，如读写文件之类的

filter\_policy.h: 定义布隆过滤器接口

options.h: 配置选项

status.h: 定义数据库操作的返回状态

table.h: SSTable相关的接口

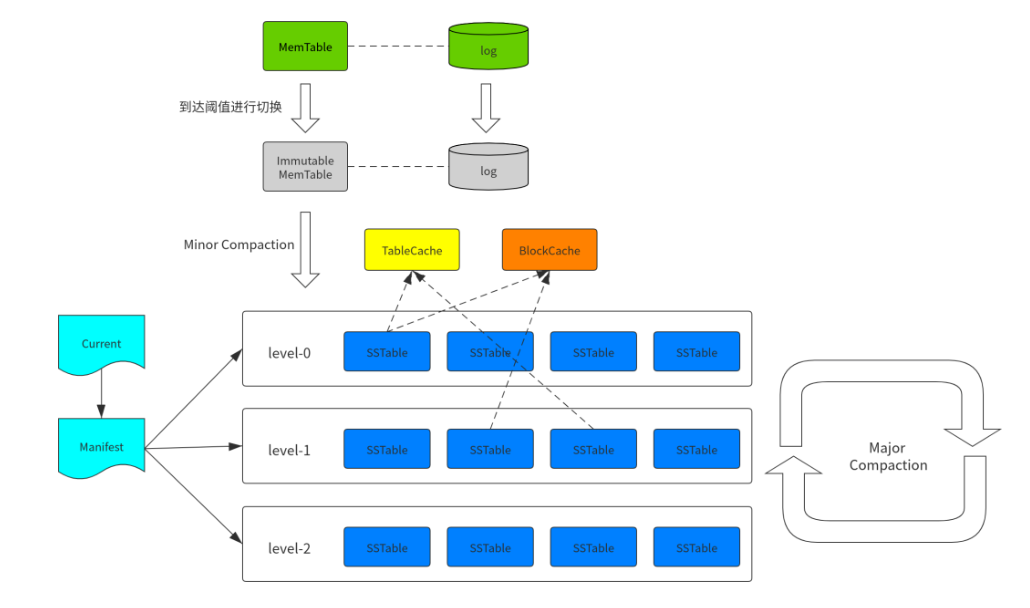
Comparator 定义排序的规则；

Status定义函数执行的结果信息；

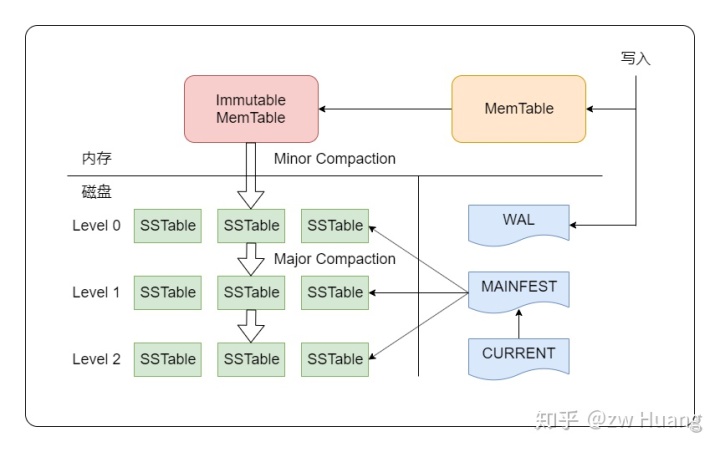
Env封装系统相关的调用，比如文件操作，线程操作；

Options指定数据库选项。

# 架构



LevelDB按照存储来分，可以分为三个组件：MemTable，SSTable，WAL。



MANIFEST是用来管理SSTable的，保存了SSTable的元信息。

# 原理

## arena内存管理

数据库内存分配非常重要，尤其是插入一个键值对时，需要分配内存给这个键值对。如果直接使用malloc/free或者new/delete碰到很小的键值对时，每个调用平均的开销比较大，而且会产生很多内存碎片。

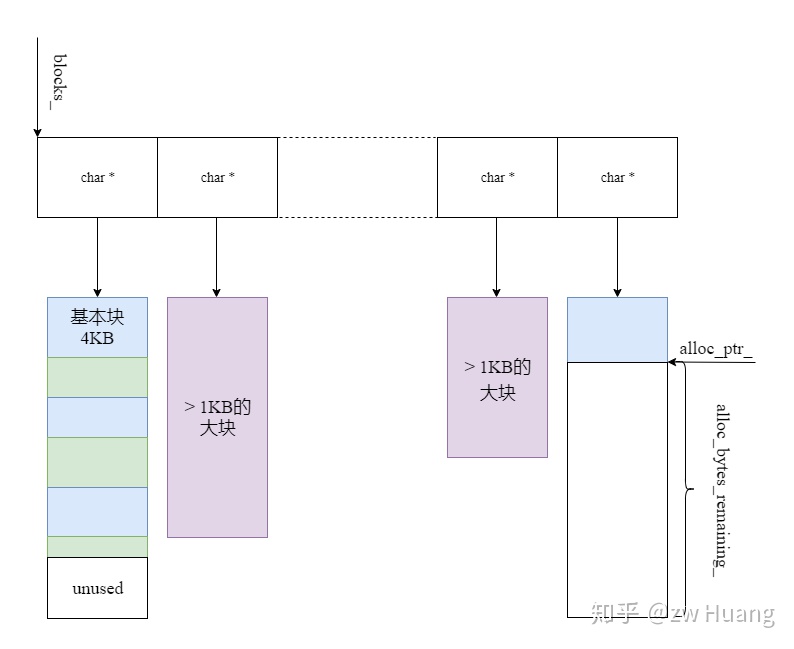
LevelDB只有一处使用了自己的内存管理，就是MemTable，MemTable使用一个Skiplist存储最新插入的键值对。LevelDB为每个MemTable都绑定了一个Arena来管理内存，其它地方则直接使用malloc/free，因为这些地方都使用了比较大块的内存或者新建销毁不频繁。

### 设计思想

内存分配经常采用的一种方式，就是首先使用new预分配一块比较大的内存，需要使用小块内存时，从这块大内存里面继续分配，这时候分配可能只是移动指针或者更新变量的事情了，非常高效。

arena内存管理就使用了这种思想，解决了小块内存频繁调用new的开销和内存碎片的问题，但是却可能浪费一些内存。

arena的内存分配如下图所示：



使用一个char \*的vector保存每个块；

当需要分配一块内存时，查看alloc\_bytes\_remaining\_(就是当前块还有多少内存未分配)是否大于等于所需内存；

如果大于等于，直接分配，这时候只需要移动指针即可；

如果小于，要分两种情况，看所需要分配的内存是否大于1KB；

如果大于1KB，直接分配相应大小的块，并且插入到vector中;

如果小于等于1KB，则分配一个4KB的块，插入到vector中，从4KB的块上分配相应的内存；

上一个块里没有分配的内存就浪费了。

arena使用一个Arena类来定义，将数据存储在std::vector<char\*> blocks\_变量里，依次存储，每个数组项存储一个内存块，使用alloc\_ptr\_和alloc\_bytes\_remaining\_来跟踪当前块分配的状态。

// util/arena.h util/arena.cc

static const int kBlockSize = 4096;

// 首先是Arena的定义

class Arena {

private:

...

char\* alloc\_ptr\_; // 指向当前块第一个free的字节

size\_t alloc\_bytes\_remaining\_; // 当前块还有多少字节free的内存

std::vector<char\*> blocks\_; // 用new分配的内存块的数组

...

};

// 分配内存的函数

inline char\* Arena::Allocate(size\_t bytes) {

// 如果当前块剩余的内存足够，更新free指针，返回内存指针

if (bytes <= alloc\_bytes\_remaining\_) {

char\* result = alloc\_ptr\_;

alloc\_ptr\_ += bytes;

alloc\_bytes\_remaining\_ -= bytes;

return result;

}

// 否则退化的方式分配

return AllocateFallback(bytes);

}

char\* Arena::AllocateFallback(size\_t bytes) {

// 如果分配的内存大于1KB，直接分配一个块

if (bytes > kBlockSize / 4) {

char\* result = AllocateNewBlock(bytes);

return result;

}

// 如果内存小于等于1KB，分配一个4KB的块，更新指针

alloc\_ptr\_ = AllocateNewBlock(kBlockSize);

alloc\_bytes\_remaining\_ = kBlockSize;

char\* result = alloc\_ptr\_;

alloc\_ptr\_ += bytes;

alloc\_bytes\_remaining\_ -= bytes;

return result;

}

这样的思路对于大块的内存会直接调用new分配，对于小块的内存会在大块内存的基础上分配。如果需要分配的内存刚好大于当前块剩余的大小，那么当前块剩余的内存空间就浪费了。这里采用简单化的处理，牺牲了内存使用率。大块的内存直接分配一个块，而不是分配一个4KB的块，可以减少内存的浪费。

对于释放内存，arena不支持单独释放某个块，而是只能销毁整个arena。这是和arena的使用场景有关的，arena存储的是内存中的键值对，对于LevelDB来说，只有插入操作，没有实际的删除操作，所以不需要释放一块内存。而当一个arena里的数据dump到SSTable后，只需要释放arena里所有的内存。

### 参考源码

util/arena.h util/arena.cc

### 总结

LevelDB的内存分配策略非常简单，这和使用场景有关的。如果Kv 数量很多，而且比较小的情况下，采用这种分配方式会非常高效，内存的浪费也可以控制在相对理想的水平，这些浪费的内存在MemTable写满后，就会释放了。

## Cache缓存

大多数磁盘数据库都提供了缓存，因为磁盘和内存的访问速度差了好几个数量级。如果整个数据库的工作集小于内存，那么热数据基本都可以缓存到内存里，这时候数据库表现得就像一个内存数据库，读写效率很高。

最完美的缓存就是将最近将要使用的数据缓存在内存里。然而，未来的访问数据是比较难估算的，一般会采取一些预读的方案将数据预先读取到内存中。而缓存的策略一般都是LRU，也就是根据过去的访问来决定缓存。遵循这样的原则：最近被访问过的数据未来有很大概率再次被访问。

LevelDB提供了一个Cache接口，用户可以实现自己的缓存方式。默认提供了一个LRU Cache，缓存最近使用的数据。

LevelDB的缓存使用在两个地方：

缓存SSTable里的Data Block，也就是缓存数据，数据的缓存不是以Kv为单位的，而是以Data Block为最小单位进行缓存，默认情况下会开启一个8MB的LRU Cache来缓存Data Block。考虑到一次扫描可能将所有的内存缓存都刷出去了，LevelDB支持在扫描时，不缓存数据；

缓存SSTable在内存中的数据结构Table，一个表在使用前需要先被Open，被Open时会将SSTable的元数据，比如Index Block和布隆过滤器，读取到内存中。缓存Table时是以个数计算的，缓存的个数是max\_open\_files - kNumNonTableCacheFiles，kNumNonTableCacheFiles表示给非SSTable预留的文件描述符数量，为10。

### 缓存的实现

#### 缓存接口

缓存有一个接口Cache，每个缓存需要实现这个接口，主要操作包括Insert、Lookup和Erase。

// include/leveldb/cache.h

class LEVELDB\_EXPORT Cache {

...

struct Handle {};

// 插入一个缓存项

virtual Handle\* Insert(const Slice& key, void\* value, size\_t charge,

void (\*deleter)(const Slice& key, void\* value)) = 0;

// 查询一个缓存项

virtual Handle\* Lookup(const Slice& key) = 0;

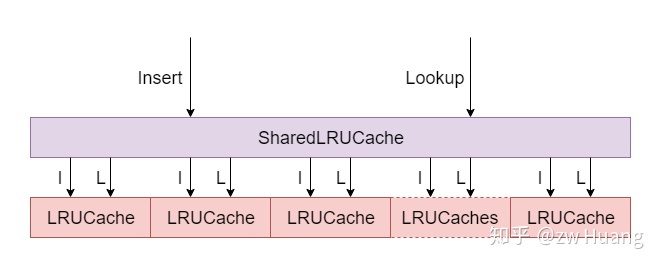
// 擦除一个缓存项

virtual void Erase(const Slice& key) = 0;

...

}

#### 分段锁缓存



LevelDB默认的LRU缓存采用了类似于分段锁的设计方式：

首先实现了一个LRUCache类，这个类实现了一个可以指定容量的LRU缓存，当达到容量后，会将旧的数据从缓存移除；

为了实现线程安全，LRUCache在做一些操作时，会进行加锁，但是加锁操作会降低并发度，针对这个问题，LevelDB对外提供的实际是一个ShardedLRUCache缓存；

ShardedLRUCache包含一个LRUCache缓存数组，大小是16，根据缓存键的Hash值的高4位进行哈希，将缓存项分布到不同的LRUCache里，这样当并发操作时，很有可能缓存项不在同一个LRUCache里，不会冲突，大大提高了并发度；

ShardedLRUCache的实现只是简单的将对缓存的操作代理到相应的LRUCache里。

以下是Insert操作的实现，根据hash值计算出对应的LRUCache，然后代理到对应的LRUCache。

// util/cache.cc

Handle\* ShardedLRUCache::Insert(const Slice& key, void\* value, size\_t charge,

void (\*deleter)(const Slice& key, void\* value)) override {

const uint32\_t hash = HashSlice(key); // 计算哈希值

return shard\_[Shard(hash)].Insert(key, hash, value, charge, deleter);

}

所以接下来重点讨论LRUCache的实现。

### LRUCache实现

// util/cache.cc

class LRUCache {

size\_t capacity\_; // 缓存容量

mutable port::Mutex mutex\_; // 包含缓存的锁

size\_t usage\_ GUARDED\_BY(mutex\_); // 当前使用了多少容量

LRUHandle lru\_ GUARDED\_BY(mutex\_); // 缓存项链表

LRUHandle in\_use\_ GUARDED\_BY(mutex\_); // 当前正在被使用的缓存项链表

HandleTable table\_ GUARDED\_BY(mutex\_); // 缓存的哈希表，快速查找缓存项

}

LRUCache的实现有以下特点：

每一个缓存项都保存在一个LRUHandler里；

每一个LRUHandler首先被保存在一个哈希表table\_里面，支持根据键快速的查找;

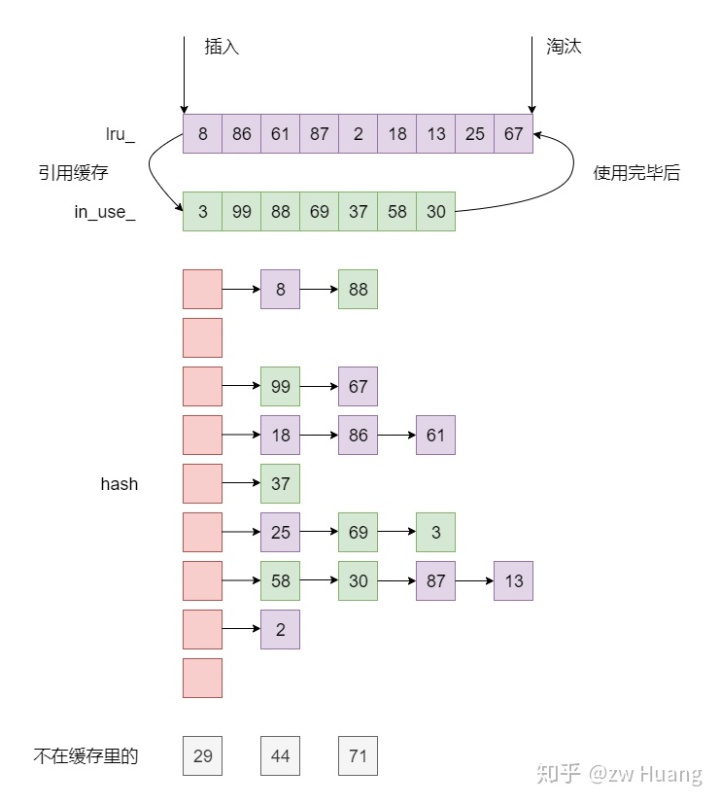
LRUCache里面有两个双向链表lru\_和in\_use\_，每一个LRUHandler可以在两个链表中的一个里，但是不会同时在两个里，也有可能有些LRUHandler被淘汰出缓存了，不在任何链表上；

in\_use\_保存当前正在被引用的LRUHandler，这个链表主要是为了检查；

lru\_保存没有被使用的LRUHandler，按照访问顺序来保存，lru\_.next保存最旧的，lru\_.prev保存最新的，需要淘汰缓存时，会从lru\_里的next开始淘汰；

当一个LRUHandler被使用时，会从lru\_移动到in\_use\_，使用完成后，会从in\_use\_重新移动到lru\_里；

每个LRUCache都有一个容量capacity\_，表示这个缓存的大小，每次插入一个项时都会指定这个缓存项的大小，更新usage\_字段，当usage\_超过capacity\_时，就淘汰最旧的缓存项，直到低于capacity\_。



以下是LRUHandler的定义：

// util/cache.cc

struct LRUHandle {

void\* value; // 值

void (\*deleter)(const Slice&, void\* value); // 数据项被移出缓存时的回调函数

LRUHandle\* next\_hash; // 哈希表的链接

LRUHandle\* next; // 两个双向链表的链接

LRUHandle\* prev;

size\_t charge; // 缓存项的大小

size\_t key\_length; // 键的长度

bool in\_cache; // 当前项是否在缓存中

uint32\_t refs; // 当前项的引用计数

uint32\_t hash; // 哈希值

char key\_data[1]; // 键值

Slice key() const {

return Slice(key\_data, key\_length);

}

};

LRUCache通过引用计数来管理LRUHandler。

// util/cache.cc

void LRUCache::Ref(LRUHandle\* e) {

if (e->refs == 1 && e->in\_cache) { // 如果当前在lru\_里，移动到in\_use\_里

LRU\_Remove(e); // 先从链表中移除

LRU\_Append(&in\_use\_, e); // 插入到in\_use\_

}

e->refs++;

}

void LRUCache::Unref(LRUHandle\* e) {

e->refs--;

if (e->refs == 0) { // 销毁缓存项

(\*e->deleter)(e->key(), e->value);

free(e);

} else if (e->in\_cache && e->refs == 1) {

// 重新移动到lru\_里

LRU\_Remove(e);

LRU\_Append(&lru\_, e);

}

}

通过引用计数，LRUCache有以下特点：

当一个LRUHandler被加入到缓存里面，并且没有被使用时，计数为1；

如果客户端需要访问一个缓存，就会找到这个LRUHandler，调用Ref，将计数加1，并且当此时缓存在lru\_里，就移动到in\_use里；

当客户端使用完一个缓存时，调用Unref里，将计数减1，当计数为0时，调用回调函数销毁缓存，当计数为1时，移动到in\_use里面；这样可以自动控制缓存的销毁，当一个LRUHandler被移出缓存时，如果还有其他的引用，也不会被销毁。

所以查找一个缓存就非常简单了:

// util/cache.cc

Cache::Handle\* LRUCache::Lookup(const Slice& key, uint32\_t hash) {

MutexLock l(&mutex\_); // 加锁操作，使用分段缓存减少锁等待

LRUHandle\* e = table\_.Lookup(key, hash);

if (e != nullptr) {

Ref(e);

}

return reinterpret\_cast<Cache::Handle\*>(e);

}

void LRUCache::Release(Cache::Handle\* handle) {

MutexLock l(&mutex\_);

Unref(reinterpret\_cast<LRUHandle\*>(handle));

}

通过哈希表查找对应的LRUHandler；

如果找到了，调用Ref，返回缓存项；

使用完缓存项后，调用Release释放缓存。

插入缓存需要将缓存项插入到哈希表以及链表中，并且更新容量，如果缓存容量过多，需要淘汰旧缓存。插入一个缓存项的步骤如下：

生成一个LRUHandler保存缓存的内容，计数为1；

再将计数加1，表示当前缓存项被当前客户端引用，插入到in\_use\_链表中；

插入时会指定插入项的大小更新usage\_字段；

插入到哈希表中；

如果有相同值旧的缓存项，释放旧项；

判断容量是否超标，如果超标，释放最旧的缓存项，直到容量不超标为止。

### 缓存使用

LevelDB里SSTable在内存中是以Table结构存在的，要使用一个SSTable，必须先进行Open操作，会将Index Block和Filter Data都读取到内存里，保存在Table里，但是Data Block依然保存在磁盘上。需要读取数据时，可以将数据放到缓存中，下次再次访问数据时，就可以从缓存里读取。所以缓存有两方面：

每个Table结构都要占据一定的内存，被打开的Table放在一个缓存中，缓存一定数量的Table，当数量太多时，有一些Table需要被驱逐出内存，这样当需要再次访问这些Table时需要再次被打开；

每个Table的Data Block可以被缓存，这样再次访问相同的数据时，不需要读磁盘。

### Table缓存

SSTable的文件名类似于000005.ldb，前缀部分就是一个file\_number，Table就是用这个file\_number作为键来缓存的。

Table的缓存存储在TableCache类里面。

// db/table\_cache.cc

Status TableCache::FindTable(uint64\_t file\_number, uint64\_t file\_size,

Cache::Handle\*\* handle) {

Status s;

char buf[sizeof(file\_number)];

EncodeFixed64(buf, file\_number);

Slice key(buf, sizeof(buf)); // key为file\_number

\*handle = cache\_->Lookup(key); // cache\_是LRUCache的实例

if (\*handle == nullptr) { // 如果缓存没命中，则打开新的Table

...

s = Table::Open(options\_, file, file\_size, &table);

TableAndFile\* tf = new TableAndFile;

tf->file = file;

tf->table = table;

// 插入一个缓存项，大小为1

\*handle = cache\_->Insert(key, tf, 1, &DeleteEntry);

}

return s;

}

查询一个Table时步骤如下：

先从缓存里面找，键是file\_number，如果找到了，就可以直接返回Table；

如果没有找到，需要Open这个SSTable，然后插入到缓存里面；

缓存的capacity\_大小为支持打开的Table的个数，而每一个缓存项大小为1，这样当缓存的Table个数大于容量时，就会将最旧的Table淘汰。

### Data Block

缓存每个Table打开的时候，都会指定一个cache\_id，这是一个单调递增的整数，每个Table都有一个唯一的cache\_id。在每一个SSTable里面，每一个Data Block都有一个固定的文件偏移offset。所以每一个Data Block都可以由cache\_id和offset来唯一标识，也就是根据这两个值生成一个键，来插入和查找缓存。

// table/table.cc

// 根据一个Index读取一个Data Block

Iterator\* Table::BlockReader(void\* arg, const ReadOptions& options,

const Slice& index\_value) {

Table\* table = reinterpret\_cast<Table\*>(arg);

Cache\* block\_cache = table->rep\_->options.block\_cache;

Block\* block = nullptr;

Cache::Handle\* cache\_handle = nullptr;

BlockHandle handle; // 保存索引项

Slice input = index\_value;

Status s = handle.DecodeFrom(&input);

if (s.ok()) {

BlockContents contents;

// 使用缓存，则先读缓存

if (block\_cache != nullptr) {

// 构造缓存键，使用cache\_id和offset

char cache\_key\_buffer[16];

EncodeFixed64(cache\_key\_buffer, table->rep\_->cache\_id);

EncodeFixed64(cache\_key\_buffer + 8, handle.offset());

Slice key(cache\_key\_buffer, sizeof(cache\_key\_buffer));

// 查找缓存是否存在

cache\_handle = block\_cache->Lookup(key);

// 存在则直接获取到block

if (cache\_handle != nullptr) {

block = reinterpret\_cast<Block\*>(block\_cache->Value(cache\_handle));

} else {

// 否则从文件里读取Data Block

s = ReadBlock(table->rep\_->file, options, handle, &contents);

if (s.ok()) {

block = new Block(contents);

if (contents.cachable && options.fill\_cache) {

// 插入缓存

cache\_handle = block\_cache->Insert(key, block, block->size(),

&DeleteCachedBlock);

}

}

}

} else {

// 不使用缓存，直接读取数据

s = ReadBlock(table->rep\_->file, options, handle, &contents);

if (s.ok()) {

block = new Block(contents);

}

}

}

...

}

当要获取一个Data Block时：

从这个Data Block的索引项出发，根据索引得到offset，然后根据Table得到cache\_id，这样就得到了缓存键；

在缓存里读取Data Block，如果存在就可以返回；

否则从文件里读取Data Block，这里根据选项fill\_cache，可以决定是否插入到缓存。

### 参考源码

include/leveldb/cache.h: 定义Cache接口

util/cache.cc: 实现LRU缓存

table/table.cc: 读取Data Block时使用缓存

db/table\_cache.cc：实现一个Table结构的缓存

### 小结

以上便是LevelDB里面缓存的实现，对于磁盘型的数据库，缓存是非常重要的，如果内存足够大，大到足以容纳所有数据，那么数据库的读效率就像内存数据库一样。

除了数据部分，索引和元数据LevelDB一般是缓存在内存里面的，基于SSTable的结构和存储，这些数据都不会改变，只读不写。只有Compaction时，才会变化，但是是生成新文件，而不是写旧数据，所以也不会有缓存更新过期的问题。

## 编码

编码说的是内存里的整数和字符串是怎么存储到磁盘上的。对于整数，主要是有Big Endian 和Little Endian的区分，还有变长整数和定长整数的区别。

对于LevelDB，有以下几点：

整数分为32位整数和64位整数；

整数分为定长整数和变长整数；

整数采用Little Endian的方式存储；

字符串采用长度前缀编码的方式存储，所以字符串里面可以出现任何字符，包括\0。

### 存储方式

#### 定长整数

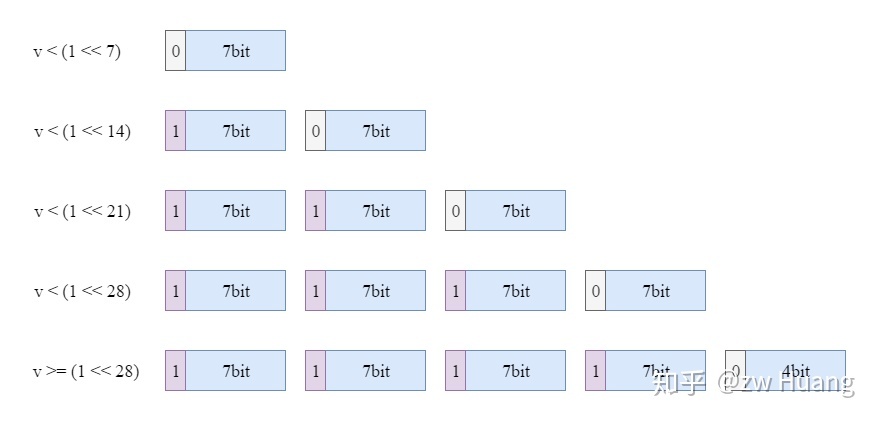
定长整数的存储非常简单，比如一个32位整数，把低8位的字节编码到第一个字节位置，低9-16位的字节放在第二个字节位置，以此类推。

#### 变长整数

一些小的整数，比如可能频繁使用100以内的整数（比如字符串的长度编码)，如果使用定长编码，至少需要4个字节，容易造成空间的浪费，所以LevelDB里面有一种变长整数的编码方式。对于32位整数使用1-5个字节进行编码，而对于64位整数使用1-10个字节进行编码。

这种编码方式的原理就是只使用一个字节的低7位存储数据，而高位用来做标识，高位为1的时候表示需要继续读取下一个字节，高位为0的时候表示当前字节已是最后一个字节。存储也是采用Little Endian的方式，拿出第一个字节的低7位作为数字的低7位，如果最高位为1，则拿出第二个字节的低7位作为数字的8-14位，以此类推。

以32位整数为例，如图:



如果编码值 v < 1 << 7，只需要7位即可编码，可使用0 + v的方式；

如果编码值 1 << 7 <= v < 1 << 14，需要两个字节编码，第一个字节使用1 + v的低7位，表示需要查看下一个字节，下一个字节使用0 + v的高7位，表示不需要查看下一个字节；

以此类推。

// util/coding.cc

char\* EncodeVarint32(char\* dst, uint32\_t v) {

uint8\_t\* ptr = reinterpret\_cast<uint8\_t\*>(dst);

// 设置字节最高位的掩码

static const int B = 128;

// 只需要1字节

if (v < (1 << 7)) {

\*(ptr++) = v;

} else if (v < (1 << 14)) {

// 低7位放在低字节，最高位置1

\*(ptr++) = v | B;

// 7-14位放在第二个字节，最高位置0

\*(ptr++) = v >> 7;

} else if (v < (1 << 21)) {

\*(ptr++) = v | B;

\*(ptr++) = (v >> 7) | B;

\*(ptr++) = v >> 14;

} else if (v < (1 << 28)) {

...

} else {

...

}

return reinterpret\_cast<char\*>(ptr);

}

#### 字符串

字符串的编码使用了前面32位变长整数来编码字符串长度，编码长度后跟字符串的实际值。因为采用长度前缀编码，所以不需要根据字符串里的\0来判断字符串的结束，字符串里面可以是任何值。

// util/coding.cc

void PutLengthPrefixedSlice(std::string\* dst, const Slice& value) {

PutVarint32(dst, value.size()); // 首先以变长的方式编码长度

dst->append(value.data(), value.size()); // 添加内容到长度后面

}

​

void PutVarint32(std::string\* dst, uint32\_t v) {

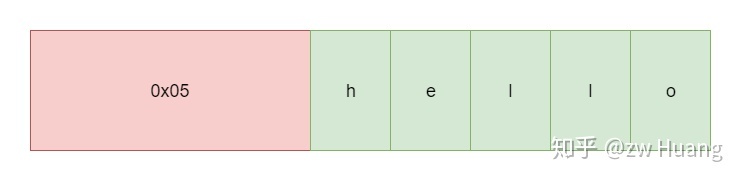
char buf[5];

char\* ptr = EncodeVarint32(buf, v);

dst->append(buf, ptr - buf);

}

假如需要编码字符串hello，因为长度是5个字节，所以长度的编码就是0x05，后面跟字符串实际值，所以最终的编码就是 05 48 45 4C 4C 4F，需要6个字节。



这种编码有以下好处：

字符串里面可以包含任何字符；

字符串的长度可以预先知道，读取文件的时候更加方便；

采用变长编码字符串长度，对于大多数的小字符串只需要1个字节。

### 参考源码

util/coding.h util/coding.cc

### 小结

LevelDB里面整数和字符串编码的方式很常见，我们经常可以在其它地方看到，这种方式简单高效紧凑。

## 字符串Slice

LevelDB里面主要用到的字符串类型不是C语言的char []，也不是C++的string对象，而是自己封装的一个简单的数据结构Slice。

Slice是一个简单的对象，只包含了指向字符串的指针和字符串的长度。为什么使用Slice而不使用char []或者string呢？有以下原因：

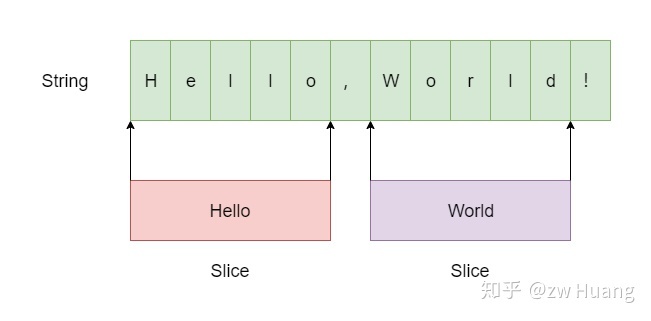
相对于拷贝string，拷贝Slice会轻量级很多，只需要拷贝指针和长度；

char []无法保存二进制字节，而Slice则没有这个问题；

Slice的底层可以是char，也可以是string；

多个Slice可以指向同一个字符串。

### 原理



Slice有一个字段char \* data\_保存字符串的指针，另一个字段size\_t size\_表示字符串的长度，也就是Slice指向另外一个字符串。

// include/leveldb/slice.h

class LEVELDB\_EXPORT Slice {

...

private:

const char\* data\_;

size\_t size\_;

}

使用Slice需要特别小心，使用者需要确保底层引用的字符串没有被销毁

鉴于以上特点，Slice对字符串的修改非常简单，比如要取一个子串，只需要改变data\_和size\_的值即可，不需要进行子串的拷贝，开销非常低，非常高效。真正做到了底层字符串不变，但是上层却可以高效地随意改变。

而Slice和string以及char []之间的转换也非常简单。

// include/leveldb/slice.h

class LEVELDB\_EXPORT Slice {

// 转换为字符串

std::string ToString() const { return std::string(data\_, size\_); }

// 去掉前缀

void remove\_prefix(size\_t n) {

data\_ += n;

size\_ -= n;

}

}

### 参考源码

include/leveldb/slice.h

### 小结

LevelDB的代码里面大量使用了Slice来代替字符串进行参数传递和返回，好处显而易见，避免了大量的字符串拷贝，提高了效率。

注：这个跟MySQL封装的String类似，也是为了效率考虑。

## 版本管理

在说版本管理实现前，有两个问题得先问问自己：

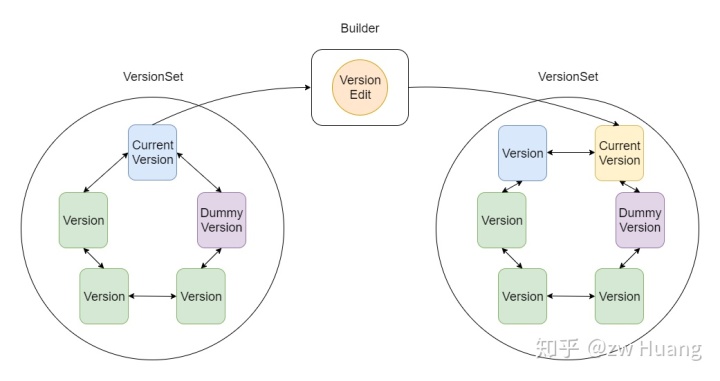
什么是版本？

为什么需要版本管理？

版本其实就是LevelDB数据库的元数据，之前提到过，在MemTable读取不到键时，需要去SSTable读取。SSTable的文件有哪些，每一个文件分别在哪一个Level上面，每个文件里面包含的键的最小值最大值是什么。需要知道这些信息，才可以快速的从SSTable里读取出相应的键的值。而版本就保存了这些信息，让我们通过版本读取SSTable。

那么为什么要版本管理呢？ 在数据不断写入后，MemTable写满了，这时候就会转换为Level 0的一个SSTable，或者Level n的一个文件和Level n + 1的多个文件进行Compaction，会转换成Level n + 1的多个文件。这会使SSTable文件数量改变，文件内容改变，也就是版本信息改变了，所以需要管理版本。

那么有一个版本就够了吗？在文件数量和内容改变时，修改当前的版本就行了。但是并不是，假设使用了整个数据库的迭代器，LevelDB在迭代数据库时，是提供了一个一致性的视图，也就是只能看到迭代前的写入，而迭代后的写入是看不到，在实现的时候，迭代器会引用这个版本，以及里面的SSTable，直到迭代结束。如果在迭代过程中，有文件删除了，那么就无法引用到这个文件了，就会出错。所以需要多个版本，有一个版本是当前版本，当新生成版本后，旧的版本如果有被其它线程引用，也需要保留，直到不被引用后，才会被删除，所以一个时刻可能有多个版本存在。



先看看LevelDB的版本管理架构图，可以看到主要用到了4个类:

Version标识了一个版本，主要信息是这个版本包含的SSTable信息；

VersionSet是一个版本集，里面保存了Version的一个双链表，其中有一个Version是当前版本，因为只有一个实例，还保存了其它的一些全局的元数据，Dummy Version是链表头；

VersionEdit保存了要对Version做的修改，在一个Version上应用一个VersionEdit，可以生成一个新的Version；

Builder是一个帮助类，帮助Version上应用VersionEdit，生成新版本。

版本管理的基本工作流程如下：

VersionSet里保存着当前版本，以及被引用的历史版本；

当有Compaction发生时，会将更改的内容写入到一个VersionEdit中；

利用Builder将VersionEdit应用到当前版本上面生成一个新的版本；

将新版本链接到VersionSet的双链表上面；将新的版本设置为当前版本；

将旧的当前版本Unref，就是引用计数减1。

版本控制中使用了引用计数来管理历史版本：

假设一个没有被访问的数据库，这时候只有一个版本，也就是当前版本v1，它的引用计数是1；

假设有一个迭代器开始访问数据库，这时候它会对当前版本v1 Ref，引用计数变成了2；

其它线程不断写入，导致了Compaction的生成，这时候生成了一个新的版本v2成为了当前版本，引用计数为1，然后对v1 Unref，这时候v1的引用计数为1；

因为v1的引用计数为1，所以不会被删除，迭代器线程还可以继续访问v1；

迭代器结束后，对v1进行Unref，这时候v1的引用计数变为0，就会从链表上删除了，这时候又只剩下v2版本了；

版本里面的SSTable也是采用引用计数来管理的，每个版本引用一个SSTable会Ref，删除版本时会Unref，如果一个SSTable的计数为0，那么这个SSTable就可以被删除了。

接下来讨论版本控制，以及这几个类的实现。

### Version

Version标识了一个版本，读取数据库时都需要使用Version里面的版本信息，主要保存的是SSTable的文件信息：

// db/version\_edit.h

struct FileMetaData {

FileMetaData() : refs(0), allowed\_seeks(1 << 30), file\_size(0) {}

int refs;

int allowed\_seeks; // Compaction时会介绍这个字段

uint64\_t number;

uint64\_t file\_size; // 文件大小

InternalKey smallest; // 文件里最小的internal key

InternalKey largest; // 文件里最大的internal key

};

class Version {

VersionSet\* vset\_; // 版本集的引用

Version\* next\_; // 版本在版本集里的next\_指针

Version\* prev\_; // 版本在版本集里的prev\_指针

int refs\_; // 引用计数

// SSTable的信息，每一项代表相应Level的SSTable信息

// 除了Level 0外，每个Level里的文件都是按照最小键的顺序排列的，并且没有重叠

// 通过这个数据项，搜索SSTable时，就可以从Level 0开始搜索

std::vector<FileMetaData\*> files\_[config::kNumLevels];

}

可以看到Version是比较简单的，最重要的信息就是这个版本包含的SSTable的信息。通过这些信息，版本提供了接口Status Version::Get(const ReadOptions&, const LookupKey& key, std::string\* val, GetStats\* stats)，也就是上一篇介绍的，可以在这些SSTable里读取出相应的键的值。

### VersionSet

VersionSet听名字就是一个Version的集合，实际上也就是这样，里面包含了一个当前存活的Version的双链表。不过除了Version以外，VersionSet还保存了一些全局的只有一份的元数据。VersionSet只有一个实例。

先来看看VersionSet的字段：

// db/version\_set.h

class VersionSet {

Env\* const env\_; // 封装部分操作系统调用，包括文件、线程操作等

const std::string dbname\_; // 数据库名称，Open时传入

const Options\* const options\_; // 数据库选项，Open时传入

TableCache\* const table\_cache\_; // 打开的SSTable的缓存，Open时创建

const InternalKeyComparator icmp\_; // 根据User Key生成的Internal Key的Comparator

uint64\_t next\_file\_number\_; // ldb、log和MANIFEST生成新文件时都有一个序号单调递增

uint64\_t manifest\_file\_number\_; // 当前的MANIFEST的编号

uint64\_t last\_sequence\_; // 上一个使用的SequenceNumber

uint64\_t log\_number\_; // 当前的日志的编号

WritableFile\* descriptor\_file\_; // MANIFEST打开的文件描述符

log::Writer\* descriptor\_log\_; // MANIFEST实际存储的格式是WAL日志的格式，所以这里用来写入数据

Version dummy\_versions\_; // Version链表的头结点

Version\* current\_; // 当前的Version

// 这是用来记录Compact的进度，Compact总是从某一Level的最小的键开始到某个键结束，

// 下次再从下一个键开始，所以这个就是下一次这个Level从哪个键开始Compact

std::string compact\_pointer\_[config::kNumLevels];

}

可以看到，VersionSet除了保存了Version的双链表以外，还保存了其它的一些元数据。有些元数据，比如当前的版本，当数据库关闭后，再次打开的时候还是需要的，这些信息就持久化到MANIFEST文件中。

### VersionEdit

VersionEdit记录了一次版本变更有哪些改变，先看看有哪些字段：

// db/version\_edit.h

class VersionEdit {

typedef std::set<std::pair<int, uint64\_t>> DeletedFileSet;

std::string comparator\_; // 比较器的名称，持久化后，下次打开时需要对比一致

uint64\_t log\_number\_; // 日志文件的编号

uint64\_t next\_file\_number\_; // ldb、log和MANIFEST下一个文件的编号

SequenceNumber last\_sequence\_; // 上一个使用的SequenceNumber

bool has\_comparator\_; // 记录上面4个字段是否存在，存在才会持久化的MANIFEST中

bool has\_log\_number\_;

bool has\_next\_file\_number\_;

bool has\_last\_sequence\_

// 和VersionSet里面的compact\_pointers\_相同

std::vector<std::pair<int, InternalKey>> compact\_pointers\_;

// 有哪些文件被删除，就是Version里哪些SSTable被删除

DeletedFileSet deleted\_files\_;

// 有哪些文件被增加，pair的第一个参数是Level，第二个参数是文件的元信息

std::vector<std::pair<int, FileMetaData>> new\_files\_;

};

VersionEdit不仅仅包含了增加了哪些文件，减少了哪些文件，这是Version的内容变更，还有其它的一些VersionSet里的信息，这些信息主要是为了持久化到MANIFEST。

VersionEdit主要有两个作用一个就是应用到版本上作版本变迁，这个就是Builder做的事情，这主要发生在内存的数据结构中。另外一个作用就是持久化到MANIFEST记录版本变迁的内容，这里会记录更多的内容，包括last\_sequence\_、next\_file\_number等，由VersionEdit::EncodeTo完成。

先看看VersionEdit的内容是如何持久化到MANIFEST里的。

首先定义了一些常量标志：

// db/version\_edit.cc

enum Tag {

kComparator = 1, // 记录Comparator的名字

kLogNumber = 2, // 记录当前时刻的log\_number

kNextFileNumber = 3, // 记录当前时刻的next\_file\_number\_

kLastSequence = 4, // 记录当前时刻的last\_sequence

kCompactPointer = 5, // 记录compact\_pointer

kDeletedFile = 6, // 记录删除的文件信息

kNewFile = 7 // 记录新增的文件信息

};

主要有两个地方需要持久化：

初始写入，需要写入当前数据状态的信息；

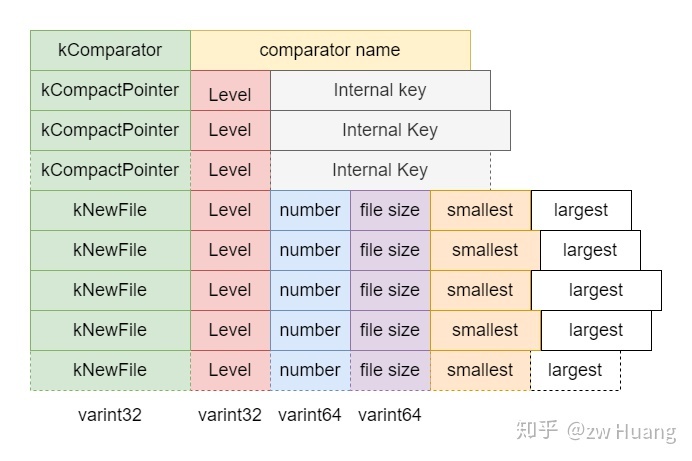
版本变迁写入，写入的是每一次版本变化的写入。

初始写入，由VersionSet::WriteSnapshot完成，当打开一个已存在的数据库，读取完现有的MANIFEST后，如果要新建一个MANIFEST替换现有的MANIFEST，就要先调用这个函数，将现有的数据库状态写入，主要包括三部分内容：

比较器的名称;

compact\_pointer\_数组，可能有多条记录，有Level和对应的Internal Key，表示Size Compaction的进程；

当前存在的SSTable文件，包含元数据，包括在哪一Level，文件编号，文件大小，里面的最小键和最大键。



版本变迁写入，就是当需要生成一个新版本时，都会向MANIFEST写入一条新纪录，记录了新版本相对旧版本有哪些变化，和旧版本合并后可以得出当前的状态，主要包括以下几部分：

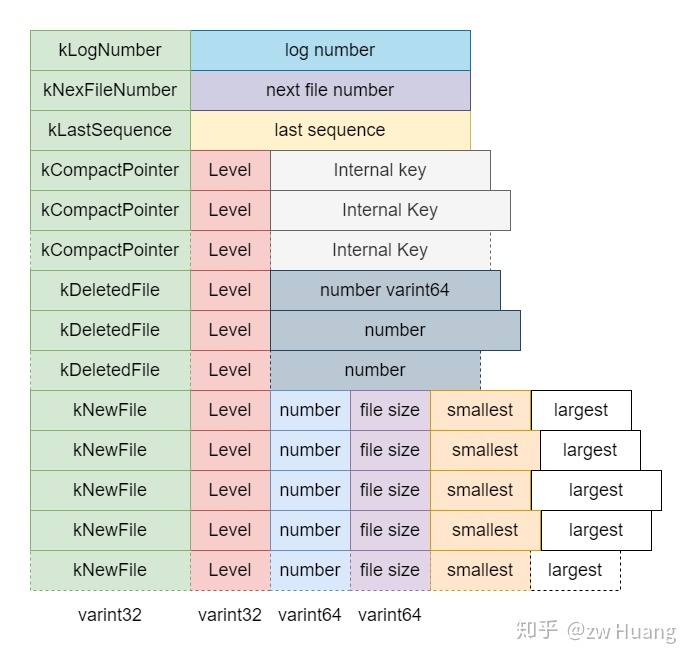
写入时刻的日志编号；

写入时刻下个可用文件的编号；

写入时刻上一个用掉的SequenceNumber；compact\_pointer\_数组，可能有多条记录，有Level和对应的Internal Key；

记录删除了哪些文件，只记录了Level和对应的file number；

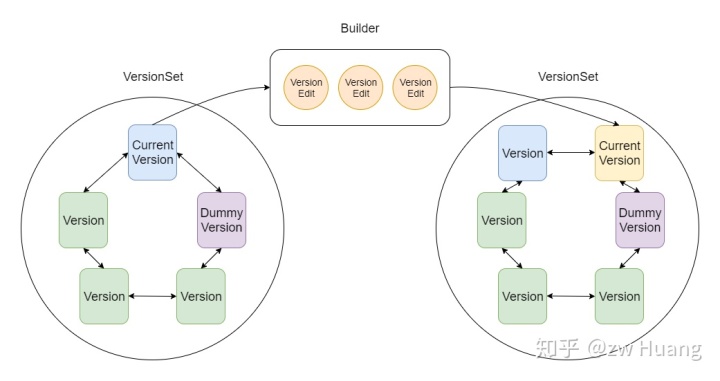
当前存在的SSTable文件，包含元数据，包括在哪一Level，文件编号，文件大小，里面的最小键和最大键。



以上就是VersionEdit的持久化，记录MANIFEST相关的内容，接下里介绍VersionEdit如何应用到当前版本生成新版本。

### Builder

我们可能会问为什么还需要Builder，当有VersionEdit改变版本时，直接引用到当前的Version生成一个新的Version即可。这其实是为了效率，才引入了Builder。正常情况下，一次Compaction或者MemTable写入后，会产生一个VersionEdit，将这个VersionEdit应用到当前Version上生成一个新的Version，这没有什么问题。不过在版本变更时，也会将VersionEdit的内容写入MANIFEST中。当重新打开一个数据库时，需要读取MANIFEST重新构造版本信息，这个版本信息由初始的Version和多个VersionEdit生成，如果直接用VersionEdit应用会生成多个版本，降低了效率。所以使用了Builder，将多个VersionEdit的内容累积到Builder上，然后一次性应用到当前Version即可生成新的Version。



生成新Version的过程大体如下：

VersionSet::Builder builder(&vset, curr\_version); // 创建一个builder

builder.Apply(version\_edit1); // 应用VersionEdit，可应用多个

builder.Apply(version\_edit2);

Version new\_version;

builder.SaveTo(&new\_version); // 将之前的Version和VersionEdit生成一个新的Version

vset.AppendVersion(new\_version); // 将新Version添加到VersionSet

先来看看Builder有哪些字段：

// db/version\_set.cc

class VersionSet::Builder {

typedef std::set<FileMetaData\*, BySmallestKey> FileSet;

// 表示某一Level，删除的文件和增加的文件

struct LevelState {

std::set<uint64\_t> deleted\_files;

FileSet\* added\_files;

};

VersionSet\* vset\_; // 对应的VersionSet

Version\* base\_; // 变化前的Version，均作为参数传入

// 表示每一Level删除的文件和增加的文件，VersionEdit Apply时都累积到这里

LevelState levels\_[config::kNumLevels];

}

可以看到Builder最重要的信息，就是VersionEdit里删除的文件和添加的文件的累积。

再来看看Apply做了哪些工作：

// db/version\_set.cc

void Builder::Apply(VersionEdit\* edit) {

// 首先更新comact\_pointer\_，这个直接在VersionSet里更新，因为这个的信息只有一份

// 新版本肯定是覆盖旧版本的，所以直接更新即可

for (size\_t i = 0; i < edit->compact\_pointers\_.size(); i++) {

const int level = edit->compact\_pointers\_[i].first;

vset\_->compact\_pointer\_[level] = edit->compact\_pointers\_[i].second.Encode().ToString();

}

// 把VersionEdit里删除的文件插入到levels\_相应Level里面去

for (const auto& deleted\_file\_set\_kvp : edit->deleted\_files\_) {

const int level = deleted\_file\_set\_kvp.first;

const uint64\_t number = deleted\_file\_set\_kvp.second;

levels\_[level].deleted\_files.insert(number);

}

// 把VersionEdit里添加的文件插入到levels\_相应的Level里去

for (size\_t i = 0; i < edit->new\_files\_.size(); i++) {

const int level = edit->new\_files\_[i].first;

// 因为是新文件，构造一个FileMetaData

FileMetaData\* f = new FileMetaData(edit->new\_files\_[i].second);

f->refs = 1;

levels\_[level].deleted\_files.erase(f->number);

levels\_[level].added\_files->insert(f);

}

}

Apply非常简单，更新VersionSet的comact\_pointer\_以及Builder的levels\_。最后再来看看SaveTo如何生成一个新版本：// db/version\_set.cc

// 尝试将一个文件插入到新版本

void Builder::MaybeAddFile(Version\* v, int level, FileMetaData\* f) {

if (levels\_[level].deleted\_files.count(f->number) > 0) {

// 文件已删除

} else {

// 插入到Version的files里面

std::vector<FileMetaData\*>\* files = &v->files\_[level];

f->refs++;

files->push\_back(f);

}

}

void Builder::SaveTo(Version\* v) {

BySmallestKey cmp;

cmp.internal\_comparator = &vset\_->icmp\_;

for (int level = 0; level < config::kNumLevels; level++) {

// 拿出原本Version里的文件，以及Builder里累积的，添加的文件

const std::vector<FileMetaData\*>& base\_files = base\_->files\_[level];

std::vector<FileMetaData\*>::const\_iterator base\_iter = base\_files.begin();

std::vector<FileMetaData\*>::const\_iterator base\_end = base\_files.end();

const FileSet\* added\_files = levels\_[level].added\_files;

v->files\_[level].reserve(base\_files.size() + added\_files->size());

// 按顺序进行合并

for (const auto& added\_file : \*added\_files) {

// 找到base里面比added\_file小的文件，添加到新的Version里

// 采用MaybeAddFile，让被删除的文件无法添加

for (std::vector<FileMetaData\*>::const\_iterator bpos =

std::upper\_bound(base\_iter, base\_end, added\_file, cmp);

base\_iter != bpos; ++base\_iter) {

MaybeAddFile(v, level, \*base\_iter);

}

MaybeAddFile(v, level, added\_file);

}

// 添加剩下的文件

for (; base\_iter != base\_end; ++base\_iter) {

MaybeAddFile(v, level, \*base\_iter);

}

}

}

可以看到Builer的作用就是将多个VersionEdit里添加的文件和删除的文件信息合并到原本的Version里生成新的Version，而对于旧的那些文件，在Version被删除时，对应的FileMetaData的引用计数会变为0，也会被自动删除。最后只剩下了新的Version和对应的SSTable文件。

### 版本变迁怎么做

介绍了前面的基本数据结构，来看看版本变迁到底是怎么做的，这是通过函数VersionSet::LogAndApply来完成的。

// db/version\_set.cc

// 输入参数edit表示的是改变的内容，比如一次Compaction可以得到edit

Status VersionSet::LogAndApply(VersionEdit\* edit, port::Mutex\* mu) {

// 设定当前的log number

if (edit->has\_log\_number\_) {

} else {

edit->SetLogNumber(log\_number\_);

}

// 设定当前的next\_file\_number和last\_sequence，这些都会被持久化到MANIFEST

edit->SetNextFile(next\_file\_number\_);

edit->SetLastSequence(last\_sequence\_);

// 创建一个新版本，新版本是current\_和edit的结合

Version\* v = new Version(this);

{

Builder builder(this, current\_);

builder.Apply(edit);

builder.SaveTo(v);

}

Finalize(v);

std::string new\_manifest\_file;

Status s;

// 这里只有Open数据库的时候才会走到，如果需要保存新的MANIFEST，此时这个变量为null

// 会创建一个新的MANIFEST，然后将当前的状态写入

if (descriptor\_log\_ == nullptr) {

new\_manifest\_file = DescriptorFileName(dbname\_, manifest\_file\_number\_);

edit->SetNextFile(next\_file\_number\_);

s = env\_->NewWritableFile(new\_manifest\_file, &descriptor\_file\_);

if (s.ok()) {

descriptor\_log\_ = new log::Writer(descriptor\_file\_);

s = WriteSnapshot(descriptor\_log\_);

}

}

{

// 写文件时释放锁

mu->Unlock();

std::string record;

edit->EncodeTo(&record);

s = descriptor\_log\_->AddRecord(record);

...

mu->Lock();

}

// 安装新版本，会把v放到VersionSet的链表中，然后将当前Version指向v

AppendVersion(v);

...

return s;

}

版本变迁基本的流程如下：

首先有了一个准备好的VersionEdit实例，包含了版本变迁的内容，然后设置其它的一些字段；

然后生成一个新的Version，使用Builder将当前的版本和VersionEdit应用生成一个新的Version；

如果当前MANIFEST的描述符不存在，新生成一个MANIFEST文件，将当前状态写入；

将VersionEdit的内容写入到MANIFEST文件；

将新生成的Version加入到VersionSet，替换当前的Version。

### 参考源码

version\_edit.h

version\_edit.cc: 实现VersionEdit

version\_set.h

version\_set.cc：实现Version、VersionSet和Builder

### 小结

以上便是版本管理的实现了，可以看到版本管理最主要管理的是SSTable的删除和添加，生成新的Version。其它的一些全局元数据的管理往往是直接更新VersionSet对应的字段的。

## Comparator

LevelDB是一个SortedMap，既然是一个SortedMap，键与键的顺序肯定是要有一个比较的规则。

有很多的比较规则：

基于二进制字节序的比较；

基于某个字符集的比较；

基于某个字符集不区分大小写的比较；

...

LevelDB没有规定比较的规则，只是定义了一个Comparator接口，用户可以提供自己的规则实现这个接口。

// include/leveldb/comparator.h

class LEVELDB\_EXPORT Comparator {

public:

virtual ~Comparator();

virtual int Compare(const Slice& a, const Slice& b) const = 0;

virtual const char\* Name() const = 0;

virtual void FindShortestSeparator(std::string\* start, const Slice& limit) const = 0;

virtual void FindShortSuccessor(std::string\* key) const = 0;

};

Compare函数实现了比较的规则，根据大小返回-1，0，1；

Name定义了比较器的名称，这个为了兼容用的，比如用某一个比较器创建了一个数据库，但是如果用一个不兼容的比较器打开这个数据库，可能就会出错，而根据Name来检查这个不匹配；

FindShortestSeparator将start更改为一个位于[start, limit)里的最短的字符串。这主要是为了优化SSTable里的Index Block里的索引项的长度，使得索引更短。因为每一个Data Block对应的索引项大于等于这个Data Block的最后一个项，而小于下一个Data Block的第一个项，通过这个函数可以减小索引项的长度；

FindShortSuccessor将key更改为大于key的最短的key，这也是为了减小索引项的长度，不过这是优化一个SSTable里最后一个索引项的。

LevelDB定义了一个默认的比较器BytewiseComparatorImpl，实现了基于二进制字节的比较，这个比较器的Name是leveldb.BytewiseComparator。

**参考源码：**

leveldb/include/comparator.h：Comparator接口定义

util/cmparator.cc：默认BytewiseComparatorImpl实现

## Status

Status定义很多操作的返回码，很多操作需要通过返回的status来判断下一步的行为。

Status由Status类来定义:

// include/leveldb/status.h

class LEVELDB\_EXPORT Status {

public:

static Status OK() { return Status(); }

static Status NotFound(const Slice& msg, const Slice& msg2 = Slice()) {

return Status(kNotFound, msg, msg2);

}

...

bool ok() const { return (state\_ == nullptr); }

bool IsNotFound() const { return code() == kNotFound; }

...

// 定义了错误码的种类

enum Code {

kOk = 0,

kNotFound = 1,

kCorruption = 2,

kNotSupported = 3,

kInvalidArgument = 4,

kIOError = 5

};

Code code() const {

// 提取错误码，在第4个字节

return (state\_ == nullptr) ? kOk : static\_cast<Code>(state\_[4]);

}

// state\_[0..3] 消息长度

// state\_[4] 错误码

// state\_[5..] 消息

const char\* state\_;

}

Status的状态都由一个私有变量char \* state\_定义：

如果状态为OK，则此变量为null；

否则，状态由一个状态码code和一个message组成；

state\_[0..3]是message的长度；

state\_[4]第4个字节是code的值，预先定义了一些状态，由enum Code定义；

state\_[5..]就是message的值了，从构造函数看出，可以同时提供两个msg，这两个msg会通过:分隔；

另外为每个enum Code都定义了两个快捷函数，一个是生成一个这个错误的status，一个是判断是否是某个错误。

**参考源码：**

leveldb/include/status.h util.status.cc: Status定义

## Env

LevelDB是一个数据库函数库，数据库总是需要操作文件和线程，这就需要做很多系统调用。各个操作系统的系统调用方式不一样，为了跨平台支持，LevelDB对这些系统调用做了一层封装，提供了统一的接口来操作，并且提供了Posix和Windows两种实现，如果需要实现其他的系统，只需要根据系统实现相应的Env即可。

Posix通过文件env\_posix.cc和posix\_logger.h两个文件来实现；

Wondows通过文件env\_windows.cc和windows\_logger.h两个文件来实现；

通过cmake来选择相应的实现。

具体的实现比较简单。

**参考源码：**

include/leveldb/env.c：env相关的接口定义

util/env\_posix.cc：Posix系统相关的封装，包括文件操作，文件锁，后台线程创建

util/posix\_logger.h：Posix写日志

util/env\_windows.cc util/windows\_logger.h: Windows相关的实现

## Options

Options定义了操作数据库的选项，定义了3个struct来操作：

Options定义打开数据库的选项；

ReadOptions定义读操作相关的选项；

WriteOptions定义写操作相关的选项。

这些选项就是简单的属性。

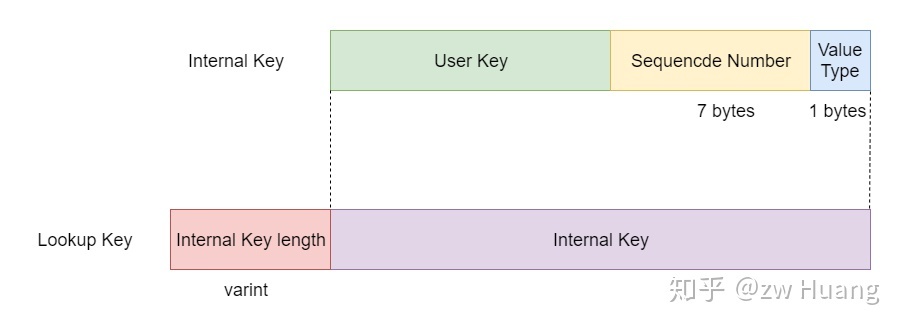
**参考源码：**

leveldb/include/options.h：定义这三个Struct

util/options.cc：定义Options的默认构造函数

## Key

LevelDB里面用到了三种键，分别是User Key，Internal Key和Lookup Key，这三种Key是包含的关系。



### User Key

第一种是User Key，这种是最简单的情况，也就是读写键值对时提供的键，只是一个简单的字符串，一般用Slice来表示。

比如调用db->Put(key, value)插入一个Kv，这个键就是一个User Key。

说简单点，应用程序和数据库之间的交互都是使用User Key来进行。

### Internal Key

第二种是Internal Key，是SSTable里实际存储的键值，也就是这个持久化有序的Map的键，定义如下：

typedef uint64\_t SequenceNumber;

static const SequenceNumber kMaxSequenceNumber = ((0x1ull << 56) - 1);

enum ValueType { kTypeDeletion = 0x0, kTypeValue = 0x1 };

struct ParsedInternalKey {

Slice user\_key;

SequenceNumber sequence;

ValueType type;

}

可以看到Internal Key在User Key的后面增加了一个64位的整数，并且将这个整数分为两部分，低位的一个字节是一个ValueType，高位的7个字节是一个SequenceNumber。

ValueType是为了区分一个键是插入还是删除，删除其实也是一条数据的插入，但是是一条特殊的插入，通过在User Key后面附上kTypeDeletion来说明要删除这个键，而kTypeValue说明是插入这个键。

SequenceNumber是一个版本号，是全局的，每次有一个键写入时，都会加一，每一个Internal Key里面都包含了不同的SequenceNumber。SequenceNumber是单调递增的，SequenceNumber越大，表示这键越新，如果User Key相同，就会覆盖旧的键。所以就算User Key相同，对应的Internal Key也是不同的，Internal Key是全局唯一的。当我们更新一个User Key多次时，数据库里面可能保存了多个User Key，但是它们所在的Internal Key是不同的，并且SequenceNumber可以决定写入的顺序。

LevelDB中每次写操作(put/delete)都有一个版本，由sequence number来标识，整个DB有一个全局值保存当前使用的Sequence Number，key的排序以及snapshot都要依赖它。

当用户写入时，将User Key封装成Internal Key，保留版本信息，存储到SSTable里，当需要读取时，将User Key从Internal Key里提取出来，所有User Key相同的Internal Key里面SequenceNumber最大的Internal Key就是当前的键，它对应的值就是当前值。

另外Internal Key的比较方式和User Key是不一样的，Options提供的是User Key的比较方式，而LevelDB内部会生成一个根据这个User Key的比较方式得到的Internal Key的比较方式。

int InternalKeyComparator::Compare(const Slice& akey, const Slice& bkey) const {

// Order by:

// user key

// sequence number 倒序

// type 倒序，实际不需要，因为sequence number唯一的

int r = user\_comparator\_->Compare(ExtractUserKey(akey), ExtractUserKey(bkey));

if (r == 0) {

const uint64\_t anum = DecodeFixed64(akey.data() + akey.size() - 8);

const uint64\_t bnum = DecodeFixed64(bkey.data() + bkey.size() - 8);

if (anum > bnum) {

r = -1;

} else if (anum < bnum) {

r = +1;

}

}

return r;

}

可以看到比较方式被改造了，当Internal Key里面包含的User Key不同时，直接用User Key的比较方式即可。否则，根据SequenceNumber比较，按SequenceNumber倒序排序。这样的好处就是，在全局有序的Map里，根据User Key排序，User Key相同的会排在一起，SequenceNumber大的Internal Key排在前面。当Seek一个User Key时，会定位到第一个符合条件的Internal Key，也就是具有最大的SequenceNumber的Internal Key。

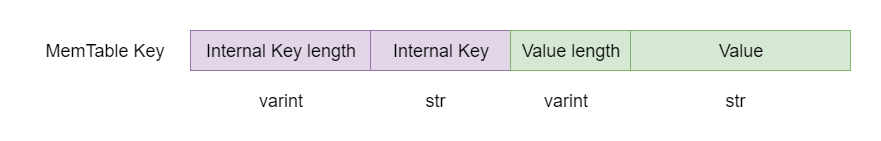
除了比较器，布隆过滤器也会被改造，以适应User Key到Internal Key的转换。

Internal Key里的SequenceNumber主要是为了支持Snapshot的功能。当生成一个DB Iterator或者给一个Option显示设置一个Snapshot，只会读取那个时刻的数据。实现方式就是当读取的时候，如果Snapshot对应的SequenceNumber小于Internal Key的SequenceNumber，那么这个键就是不可见的，找到可见的SequenceNumber最大的Internal Key就是需要读取的键的值。

### Lookup Key

Lookup Key其实就是简单的在Internal Key前面加上键的长度，使用varint32编码，主要用在MemTable的查找上。

为什么需要Lookup Key呢？



这主要还是要从MemTable的存储说起。MemTable的底层是一个Skiplist，而LevelDB的Skiplist只存储了一个键，而没有值。而LevelDB在存储Kv时，是将键和值编码在一起存储的，使用的就是字符串的长度前缀编码。所以在MemTable里查找Key时，提供的Lookup Key就是编码值的一个前缀，刚好可以定位MemTable里相应的键。

### 参考源码

db/dbformat.h

db/dbformat.cc: 定义了Internal Key、Lookup Key、InternalKeyComparator和InternalFilterPolicy

### 小结

以上就是LevelDB的三种键，是包含的关系：

User Key是用户提供的键，是我们看到的键；

Internal Key是实际存储的键，支持版本号和Tag的功能；

Lookup Key则是为了查找MemTable而构造的键。

## Iterator

Level DB中，实现了各种Iterator，Iterator有以下功能：

按顺序对所有的元素进行迭代；

处理某一范围内的元素，正序或者逆序；

定位到某一个特定的元素进行处理。

LevelDB在以下部分使用了迭代器：

对MemTable进行迭代；

对于SSTable的Block进行迭代；

对整个SSTable进行迭代；

Level File Num Iterator，因为LevelDB的SSTable是分层的，这个Iterator对某一个版本的某一层的SSTable的文件信息进行迭代；

Concatenating Iterator，Level > 0的SSTable是有序的，Concatenating Iterator可以对这些有序的SSTable同时迭代；

对MemTable、Immutable MemTable和所有的SSTable同时迭代；DB Iterator对整个数据库进行迭代。

以上迭代器的实现，有些是从零开始实现的，有些是组合其它迭代器实现的。为了组合其它迭代器，LevelDB实现了两种迭代器的组合方式：

Two Level Iterator，可以组合两个迭代器A和B，其中A里面的每个元素可以产生一个迭代器B，可以迭代A，取出一个元素产生迭代器B，然后迭代B，然后再产生A的下一个元素，再产生一个迭代器B，如此往复；

Merger Iterator可以组合多个迭代器，同时对多个迭代器进行迭代，就好像对这些迭代器做了一次归并排序，产生结果。

### Iterator接口

// include/leveldb/iterator.h

class LEVELDB\_EXPORT Iterator {

public:

...

// 当前迭代器是否有效

virtual bool Valid() const = 0;

// 定位到某个键，当键不存在时，定位到比这个键大的第一个键

virtual void Seek(const Slice& target) = 0;

// 定位到下一项

virtual void Next() = 0;

// 定位到前一项

virtual void Prev() = 0;

// 返回当前键

virtual Slice key() const = 0;

// 返回当前值

virtual Slice value() const = 0;

// 返回当前迭代器状态

virtual Status status() const = 0;

// 迭代器被销毁时的回调函数，注册后可在销毁时调用

using CleanupFunction = void (\*)(void\* arg1, void\* arg2);

void RegisterCleanup(CleanupFunction function, void\* arg1, void\* arg2);

...

};

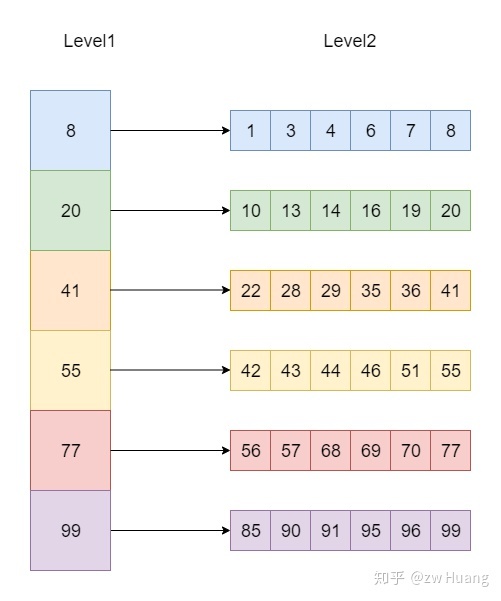
LevelDB的Iterator接口如上，里面有很多函数，但是最重要的还是Seek、Prev和Next这三个函数。

首先先来看看两个组合迭代器的实现方式，它们是很多迭代器的基础。

### Two Level Iterator

Two Level Iterator其实就是使用两个Iterator，第一个Iterator是第二个Iterator的索引。先在第一层的Iterator做迭代，每次拿出一个元素后，根据这个元素调用回调函数，生成第二层的一个Iterator，然后第二层的Iterator迭代完成后，再在第一层取下一个元素。

如图，左边的为第一层的Iterator，从上到下迭代，取出一个元素，根据这个元素找到第二层，也生成一个Iterator，然后迭代第二层的数据，完成这个Iterator的迭代后，再取出第一层的下一个元素继续。

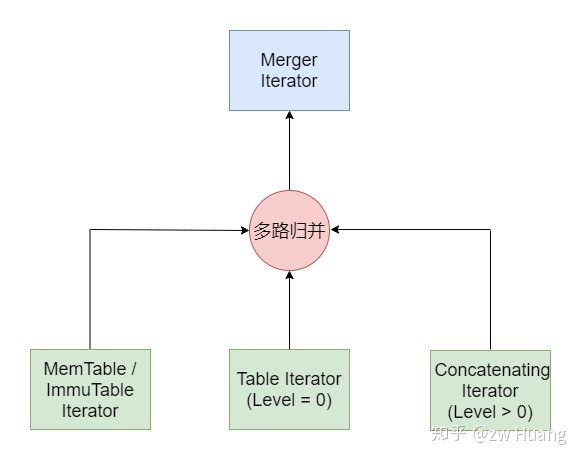


对使用Two Level Iterator有两个要求: 第一层的Iterator的元素是有序排序的； 根据第一层的Iterator生成的第二层的Iterator也是全局有序的，也就是第一层第n个元素生成的第二层Iterator的最大元素小于第一层第n + 1个元素生成的第二层Iterator的最小元素，并且第二层的每个Iterator内部也是有序的。

Two Level Iterator使用就像一个二级索引，对第一级生成Iterator，然后Iterator里每个元素指向第二级里的数据，也可以生成一个Iterator。

### Merger Iterator

Merger Iterator可以用来组合多个Iterator，只需要保证每一个Iterator内部是有序，而不需要每一个Iterator都是全局有序的。Merger Iterator会对所有Iterator进行迭代，就好像归并排序一样。找到每个Iterator头部最小的元素，next时，将最小的元素所在的Iterator next，然后再次找到最小的元素。



以下是生成一个Merger Iterator的方式：

// table/merger.cc

Iterator\* NewMergingIterator(const Comparator\* comparator, Iterator\*\* children,

int n) {

if (n == 0) {

return NewEmptyIterator();

} else if (n == 1) {

return children[0];

} else {

return new MergingIterator(comparator, children, n);

}

}

可以看出，当输入超过1个Iterator时，会生成一个Merger Iterator。输入是一个Iterator数组，只需要实现Iterator接口，不用管具体的实现是什么，输出也是一个实现了Iterator接口的Merger Iterator。

先来看看Next的实现：

// table/merger.cc

void MergingIterator::Next() override {

// 改变方向

if (direction\_ != kForward) {

for (int i = 0; i < n\_; i++) {

IteratorWrapper\* child = &children\_[i];

if (child != current\_) {

child->Seek(key());

if (child->Valid() &&

comparator\_->Compare(key(), child->key()) == 0) {

child->Next();

}

}

}

direction\_ = kForward;

}

current\_->Next();

FindSmallest();

}

属性current\_保存了当前迭代的键所在的子Iterator；

如果之前就是Next，这次再做一次Next很简单，只需要把当前的Iterator取一次Next，然后在所有的子Iterator的当前元素里面找到最小的那个元素，就是迭代的下一个元素，所在的迭代器更新为当前Iterator；

如果direction\_ != kForward，说明变换了方向，之前是Prev，这时候需要将所有的子Iterator Seek到当前元素，使得除了current\_的所有子Iteartor都指向大于当前键的下一个元素。

Prev的实现和Next类似。

Seek操作只需要对所有的子Iterator做一次Seek操作，找到最小的那个元素，就是当前Seek游标所在。

### MemTable Iterator

MemTable的Iterator其实就是对Skiplist进行操作。

对于Seek操作，其实只需要调用Skiplist的查找操作即可；

对于Next操作，因为Skiplist的最低层是一个单链表，所以只需要取这个链表的Next即可定位到下一个元素；

对于Prev操作，稍微复杂一点，需要用查找函数找到当前元素的前一个元素。

### Block Iterator

Block Iterator是用来迭代Block的，Block的数据部分是按照键的顺序排列的，所以Next的实现非常简单，只需要解析下一个Kv即可。

因为每个Kv的长度是不同的，没法直接定位到具体的某一个Kv，但是对于Seek操作，可以使用restart point来进行快速定位。之前说过restart point指向了某一个键，是一个稀疏索引。可以先对restart point进行二分搜索，找到restart point 对应的键小于等于target最大的那个restart point，如果键存在，则必在这个restart point开始的16个键中，再从这个位置开始搜索，就可以找到对应的键。

Prev的实现和Seek类似，只需要找到当前元素的前一个元素所在的restart point，然后搜索即可。

### SSTable Iterator

SSTable Iterator其实是一个Two Level Iterator。

SSTable是由Data Block和Index Block组成的，而且满足Two Level Iterator的两个条件，可以将对SSTable的迭代使用一个Two Level Iterator来实现。使用Index Block的Iterator成为第一层的Iterator，里面每取一个元素，都可以找到对应的Data Block，生成第二层的Iterator。

先来看一个SSTable的Iterator是怎样生成的：

// table/table.cc

Iterator\* Table::NewIterator(const ReadOptions& options) const {

return NewTwoLevelIterator(

rep\_->index\_block->NewIterator(rep\_->options.comparator),

&Table::BlockReader, const\_cast<Table\*>(this), options);

}

第一个参数是第一层的Iterator，第二个参数是一个回调函数，这个回调函数以第一个参数迭代返回的元素，以及第三个和第四个参数为参数进行调用，以此来获取索引值对应的第二层的Data Block的Iterator。

### Level File Num Iterator

LevelDB的SSTable是分层保存在一个数组里面，每一层一个数组，保存的是文件的File Number。

Level File Num Iterator其实非常简单，就是对每一层的文件信息数组，生成一个迭代器，将一个内存里的vector变成一个迭代器。

### Concatenating Iterator

除了Table的迭代，Two Level Iterator还用在另外一个地方：

// db/version\_set.cc

Iterator\* Version::NewConcatenatingIterator(const ReadOptions& options,

int level) const {

return NewTwoLevelIterator(

new LevelFileNumIterator(vset\_->icmp\_, &files\_[level]), &GetFileIterator,

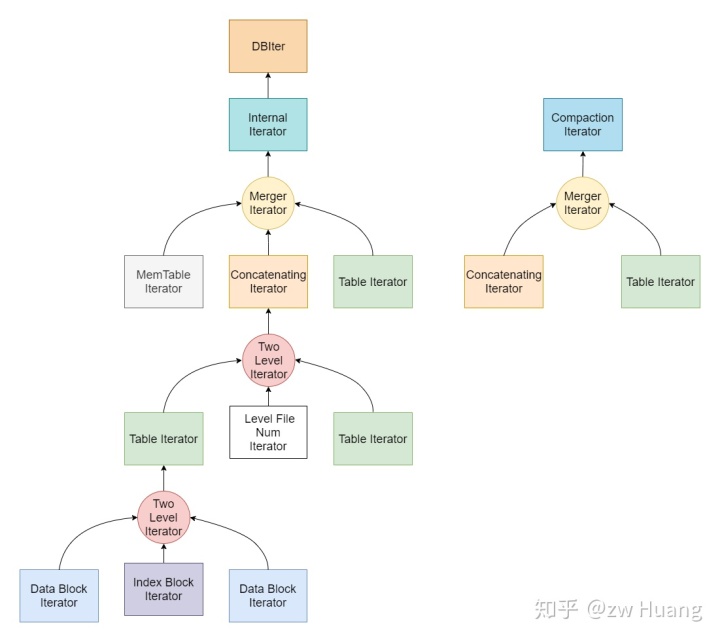
vset\_->table\_cache\_, options);

}

这个迭代器可以对某一层（除了level 0）的SSTable进行迭代。第一层是一个Level File Num Iterator，可以返回这一层的SSTable文件信息，而第二层则是SSTable Iterator，使用GetFileIterator可以获取第二层的SSTable Iterator。这样就可以对这一层的所有SSTable里的键从小到大迭代。

### Iteator整体结构概览

经过以上的讨论，可以把LevelDB里的Itertaor的层次结构，用一张图来表示。



从上图可以看到：

Two Level Iterator可以将Index Block Iterator和Data Block Iterator组合成一个Table Iterator；

Two Level Iterator可以将Level File Num Iterator和Table Iterator组合成一个Concatenating Iterator，这个Iterator迭代某一Level（Level > 0）的所有SSTable；

Merger Iteator可以组成一个Internal Iterator，对整个数据库里的数据进行迭代，包括MemTable Iterator、Table Iterator（Level 0）和Concatenating Iterator（Level > 0)，注意这个Iterator是不会区分版本的，所有版本的数据都能看到；

Internal Iterator做一定的处理后可以生成DBIter迭代整个数据库；

Merger Iterator也可以组成一个Compaction Iterator，包括Table Iterator（Level 0）和Concatenating Iterator（Level > 0)，这主要是Compaction时使用。

以上Internal Iterator是会看到所有数据的，比如一个键覆盖了之前的值，这两个键值对都会看到。所以LevelDB里面还有一个DBIter，封装了对Internal Iterator的访问，根据当前的SequenceNumber，只会看到可见的那个键。DBIter会对Internal Iterator作封装，查找一个Key时会找到可见的SequenceNumber最大的那个Key，Next的话，会跳过所有User Key相同的Internal Key直到一个User Key不同的并且可见的Internal Key，这样迭代整个数据库不会出现重复的User Key。

### 参考源码

include/leveldb/iterator.h：定义Iterator的接口

table/iterator.cc: 实现CleanUp的功能以及Empty Iterator

table/block.h table/block.cc: 实现了Block Iterator

db/skiplist.h: 实现了skiplist的Iterator

table/two\_level\_iterator.h table/two\_level\_iterator.cc： 实现了Two Level Iterator

table/merger.h table/merger.cc: 实现了Merger Iterator

table/iterator\_wrapper.cc: 实现一个Iterator包装器，缓存valid()和key()的结果，避免虚函数调用和更好的cache locality

db/db\_iter.h db/db\_iter.h：实现对整个数据库的迭代器

### 小结

可以看到LevelDB里面大量使用了Iterator，而Iterator的统一都是使用了接口。Iterator是分层次的，多种Iterator可以进行组合，但是使用的时候的方式又是一样的，让代码更为简洁。

## 数据库文件

LevelDB在代码层面并没有区分数据库层和存储引擎层，不过这里把相关功能拆分为两个层面，我们先讲述存储引擎层。

存储引擎的作用是存储数据以及检索数据，核心就是数据是用怎么存储的。

LevelDB的存储引擎主要分为三个部件：SSTable，就是Sorted String Table，是一个持久化的、有序的SortedMap，存储在磁盘上；

WAL，Write Ahead Log，数据库里面经常用的技术，要写数据时，不直接写数据文件，而是先写一条日志，这样可以把对磁盘的随机写转换成顺序写；

MemTable，保存了最近写入的键值对，数据写入WAL后，会同时写入MemTable，这样便于查询。

SSTable是数据最终落盘的地方，而WAL保存了最近写入的数据，持久化到磁盘上，MemTable则是WAL里数据的内存表示，因为日志的格式不便于查询，在内存中才便于快速查询。

### 概述

使用Python的LevelDB客户端plyvel来操作LevelDB，可以使用pip来安装

创建并且打开一个数据库，插入100000个键值对。

>>> import plyvel

>>> db = plyvel.DB('/tmp/testdb/', create\_if\_missing=True)

>>> for j in range(100000):

... db.put(bytes(j), 'hello:%d' % j)

...

>>>

查看一下，生成了哪些文件：/tmp $ cd /tmp/testdb

/tmp/testdb $ ls -lh

total 1.7M

-rw-r--r-- 1 hzw hzw 595K Oct 22 19:58 000004.log

-rw-r--r-- 1 hzw hzw 721K Oct 22 19:58 000005.ldb

-rw-r--r-- 1 hzw hzw 16 Oct 22 19:58 CURRENT

-rw-r--r-- 1 hzw hzw 0 Oct 22 19:58 LOCK

-rw-r--r-- 1 hzw hzw 266 Oct 22 19:58 LOG

-rw-r--r-- 1 hzw hzw 96 Oct 22 19:58 MANIFEST-000002

发现生成了6个文件：

000004.log是WAL文件，写入的键值对，都会先写到这个日志文件中；

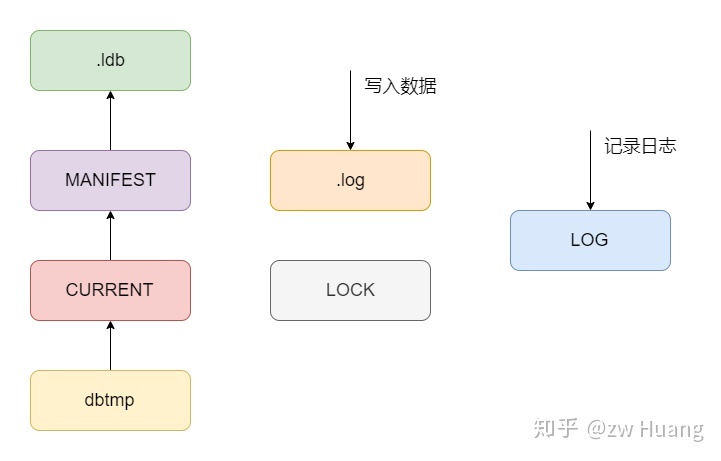
000005.ldb是SSTable文件，存储了持久化到磁盘的键值对；

LOCK文件是锁文件，一个LevelDB数据库同时只允许被一个进程操作，一个进程打开一个数据库时，会对这个文件加锁，防止其它进程并发打开这个数据库；

LOG是通用日志文件，在里面打印一些系统运行的信息；

MANIFEST-000002是资源文件，记录了版本信息，LevelDB有一系列的ldb文件，各个文件在不同的Level，而资源文件记录了当前各个文件在哪一层，下一个待分配的文件编号是什么等信息；

CURRENT，在进行一次Compaction后，生成新的版本信息，会将变化写入到MANIFEST文件中，如果MANIFEST太大，下次打开时会重写MANIFEST文件，会新增一个MANIFEST，而CURRENT则保存了当前使用的MANIFEST文件，是为了安全地重写MANIFEST文件。以上文件有带编号和不带编号的，带编号的文件会同时存在多个，并且使用一个共用的单调递增的数字来赋予编号，而不带编号的文件只有一个。



以下是文件类型的定义，发现还多了一个dbtmp文件，这个文件的作用是更换CURRENT时，不是直接覆盖CURRENT文件，而是先生成一个临时文件，然后将临时文件重命名为CURRENT文件，防止更新的时候出错。

// db/filename.h

enum FileType {

kLogFile, // .log

kDBLockFile, // LOCK

kTableFile, // .ldb

kDescriptorFile, // MANIFEST

kCurrentFile, // CURRENT

kTempFile, // dbtmp

kInfoLogFile // LOG

};

### 参考源码

db/filename.h

db/filename.cc：存储文件类型、生成和解析

### 小结

LevelDB里存储数据主要有磁盘上的WAL和SSTable以及内存中的MemTable；

Compaction后，会生成新的版本，版本信息使用MANIFEST文件来保存，而为了安全的管理MANIFEST，使用了CURRENT文件，指向当前的版本信息。

## Memtable

DB数据在内存中的存储方式，写操作会先写入memtable，memtable有最大限制(write\_buffer\_size)。**LevelDB/RocksDB的memtable的默认实现是skiplist**。**当memtable的size达到阈值，会变成只读的memtable(immutable memtable)。后台compaction线程负责把immutable memtable dump成sstable文件**。

RocksDB增加了column family的概念，不同的column family不共享memtable，其他memtable机制与LevelDB一样。

### MemTable

MemTable是LevelDB的存储组件之一，是一个内存数据结构，它只需要满足一个SortedMap的特性即可：

必须是一个Map，也就是可以快速根据键查询到值，Map的实现多用哈希或者红黑树；

Map中键是排序的，也就是可以对Map进行范围查询，或者按键的顺序迭代，这时候哈希就无法满足条件了。

MemTable：内存数据结构，具体实现是SkipList。接受用户的读写请求，新的数据会先在这里写入。

MemTable是一个内存数据结构，简单说它就是一个SortedMap：

MemTable是一个Map，提供了Get接口，也就是可以快速地根据键查找到值，也可以使用Put接口插入KV；

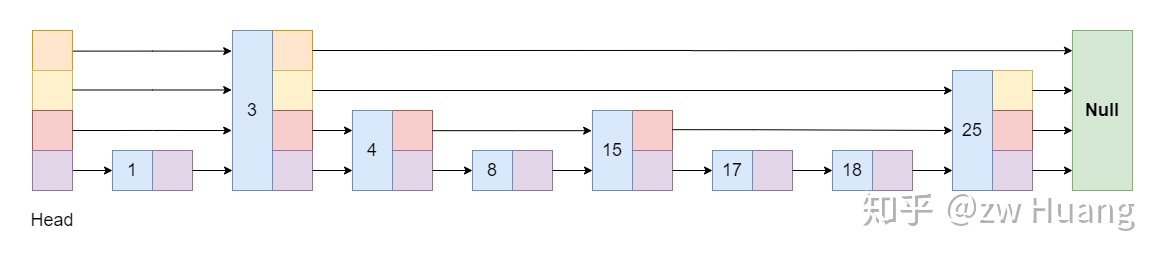
MemTable是Sorted，也就是里面存储的键都是有序的，可以按照键的顺序迭代Map，或者做范围查询。

有了MemTable提供的Get和Put，就有了一个简单的内存KV数据库，可以实现KV的查询，插入和范围查询。

因为MemTable是内存数据结构，不需要磁盘IO，所以读写的速度非常快，所以就达到了场景需求：高效的写性能。然而这会有个问题，当数据库实例崩溃、宕机或者停机维护的时候，存储的数据就会丢失，这时候需要持久化数据。

#### Skiplist

SortedMap的实现，一般会采用红黑树，不过LevelDB采用的是Skiplist。Skiplist是一种概率性的数据结构，支持SortedMap的所有功能，性能和红黑树相当。



Skiplist每一个节点有一个随机产生的层数，从1层到N层。每一层的节点组成一个链表。第1层的链表包括所有的节点，并且是按照顺序排列的，从Head开始遍历第一层就可以遍历所有数据。上面的层的链表是稀疏的，层越高越稀疏。查找时从从Head节点的第N层开始逐渐下降，直到找到对应的节点。

假设要查找节点17:

Head节点最高层开始，下一个节点是3，小于17，所以定位到3这个节点;

3节点的最高层的下一个节点是Null，所以在3这个节点下降一层;

3节点的次高层的下一个节点是25，大于17，所以目标节点肯定在3和25节点之间，3这个节点再下降一层;

3节点的第二层的下一个节点是4，小于17，就定位到节点4;

节点4的最高层的下一个节点是15，小于17，就定位到节点15；

节点15的下一层节点是25，大于17，所以在节点15再下降一层;

节点15的下一个节点是17，就找到了目标节点。

Skiplist的原理大家可以查询相关的文件，不过多介绍。

采用Skiplist的最大理由还是Skiplist的实现比红黑树简单太多了。

LevelDB里面使用SkipList类表示一个Skiplist，它主要的接口是：

void SkipList::Iterator::Seek(const Key& target);

void SkipList::Insert(const Key& key);

这个接口就是查找一个键，插入一个键，并没有提供删除接口，这是因为LevelDB里面没有实际的删除，删除也只是简单的插入一条记录，说明某个键被删除了。

#### 内存分配

SkipList插入一个键时，需要分配内存给一个节点，malloc是一个比较耗时的系统调用，尤其对于小内存分配来说，而且会产生很多内存碎片。

LevelDB里面的SkipList和一般的Skiplist使用有很大的不同，只会分配内存，而不需要回收小部分内存。SkipList的内存是等待MemTable写满后，转换为Immutable MemTable，写入SSTable，写入完成后，一起被销毁。也就是内存的回收是针对整个Skiplist的。

针对这些特点，LevelDB为MemTable定制了Arena内存管理，前面有介绍过，这种内存分配有以下特点：

每次申请一个较大的内存，供小内存分配使用，这时候只需要改变几个指针即可；

对于较大的内存分配，直接调用系统调用进行分配；

不会回收部分内存，只会整块回收;

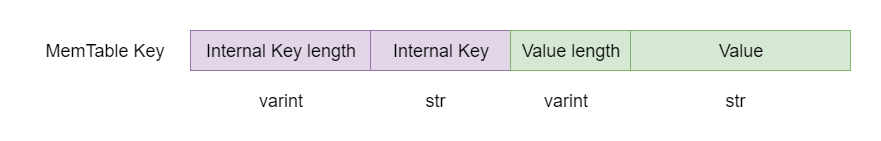
可能会有一些内存的浪费。

#### 构造MemTable

MemTable的实现是比较简单的，对SkipList的实现做了一层封装，将对键的查找和插入的请求，代理到相应的SkipList，调用相应的接口。

在实现中有一点值得注意，LevelDB存储的是Kv，而SkipList存储的是键，所以在MemTable里需要做一个转换。

对于插入操作，MemTable会把键和值编码在一个字符串里面，如下图所示，先是键，再是值，使用了字符串的长度前缀编码，然后将这个字符串插入到SkipList里。



而对于查找操作，只会按照前缀编码键，构造上面图的前半部分，而SkipList::Iterator::Seek的实现，会将迭代器定位到大于等于查找键的第一个键的位置，读出这个键，然后比对里面的键和查找的键是否相同，相同的话，才会读取对应的值，否则就是键不存在。

#### 参考源码

db/skiplist.h: 跳跃表实现

db/memtable.h db/memtable.cc: MemTable的实现

#### 小结

MemTable的实现很简单，主要就是对Skiplist的了解，相比红黑树来说，Skiplist实现的代码非常好，也很容易理解。

### Immutable MemTable

Immutable MemTable：当MemTable的大小达到设定的阈值后，会被转换成 Immutable MemTable，只接受读操作，不再接受写操作，然后由后台线程flush到磁盘上—这个过程称为minor compaction。

## WAL

Log：数据写入MemTable之前会先写日志，用于防止宕机导致MemTable的数据丢失。一个日志文件对应到一个MemTable。

为了不丢失数据，需要持久化的功能。把数据持久化到磁盘上，最常用的技术就是日志技术，也就是当修改数据时，先把对数据的修改写到磁盘上，然后在MemTable里做修改。因为日志记录了每个操作，只要对日志进行重放便可以恢复MemTable，这样就做到了数据库实例崩溃、宕机或者停机维护的时候数据不丢失。

这种日志技术在数据库里面很常用（Redis里的AOF，Innodb里的Redo Log都是这样的技术），一般称为WAL （Write Ahead Log），正如名字一样，就是在写入前先写入日志的意思。

因为日志的写入都是Append的，也就是顺序写，所以写磁盘也是顺序写，虽然磁盘的随机写效率比较低，但是顺序写效率还是很高的，所以就算加入了日志，写效率还是很高，依然可以满足写多的场景。

另外可以通过控制日志写同步的策略在性能和可靠性之间做折中：

1、每次写入都做一次sync，可靠性最高，不会丢数据，但是性能最低；

2、每次写入，但是不sync，数据库崩溃不会丢数据，但是机器崩溃会丢数据，性能高；

3、每次写入，不sync，但是每1s做一次sync，数据库崩溃不会丢数据，但是机器崩溃丢最多1s的数据，性能较高。

有了MemTable和WAL，就有了一个功能比较完备的数据库了，类似于一个简化版的Redis。然而这又引入了新的问题：

1、如果日志量很大，每次重启数据库时，恢复的时间非常长；

2、内存的容量是有限的，所以当数据量太大，MemTable的大小超过内存容量时，就没法写入数据了。

这时候不仅仅需要将日志写入到磁盘，也需要定时将MemTable的镜像写入磁盘，并且清空MemTable。

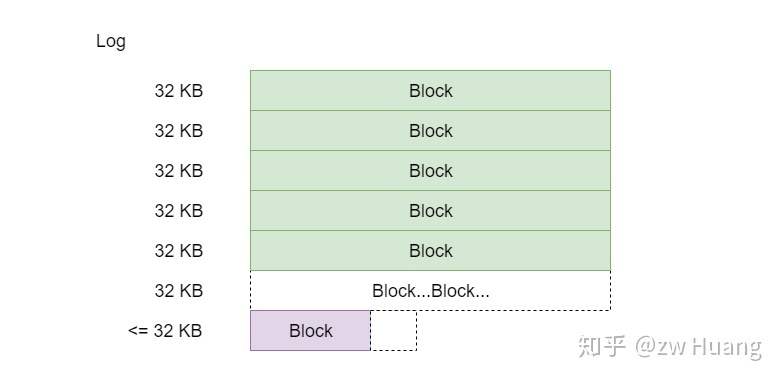
说明：Redis是完全基于内存的KV数据库，LevelDB是基于内存和磁盘的KV数据库，比Redis多了一个SSTable，即从内存刷新到磁盘的动作。

### 概述

LevelDB写入一个Kv时，都会先向日志里写入一条记录，这种日志一般称为WAL，也就是Write Ahead Log，类似于MySQL的Redo Log。这种日志最大的作用就是将对磁盘的随机写转换成了顺序写。当故障宕机时，可以通过WAL进行故障恢复。控制每次WAL写入磁盘的方式，可以控制最多可能丢失的数据量。

WAL里的内容实际就是内存里MemTable内容的持久化，当一个MemTable写满后，开启一个新的MemTable时，也同时会开启一个新的WAL，当MemTable被Dump到磁盘后，相应的WAL可以被删除。

WAL的格式很简单，由一系列32KB的Block组成，当然最后一个块可能是不满的，正在写入中。



### Log Block结构

Log Block的结构如下：Struct LogBlock {

LogRecord[] records; // 一个Block包括一系列的LogRecord

byte[] padding; // 通过padding正好到32KB

}

每个Log Block由一系列的Log Record组成，每个Log Block大小为32KB，一个Log Record至少有7个字节，所以当距离32KB小于7个字节时，需要用padding补齐到32KB，padding都是0x00，再开始下一个Log Block。

Log Record的结构如下:

struct LogRecord {

uint32 checksum; // 下面三个字段的checksum

unint16 length; // 数据的长度

byte type; // Log Record的类型

byte data[length]; // 实际的数据

}

// db/log\_format.h

enum RecordType {

// Zero is reserved for preallocated files

kZeroType = 0,

kFullType = 1,

// For fragments

kFirstType = 2,

kMiddleType = 3,

kLastType = 4

};

每个Log Record都由7个字节的固定部分开头，4个字节是后面所有部分的checksum，2个字节是数据的长度，一个字节是这个Log Record的类型。数据是二进制的，需要自己解释，没有任何格式的要求。

对于type有几种情况，主要是为了解决写入数据时，Log Block里的空间不足以容纳数据的情况。一条记录可能很大，无法在一个Data Block里容纳，这时就需要拆分这个记录，例如：

当前Log Block里的空间足以容纳写入的数据，type为kFullType，表示当前Log Record里包含所有的数据；

当前的Log Block里的空间不足以容纳写入的数据时，将写入的数据拆分，用前面部分将当前Log Block填满，这时候type就是KFirstType，表示当前的Log Record是数据的第一个部分；

接下来开始一个新的Log Block，如果这个Log Block依然不能容纳所有的数据，这时候type就是kMiddleType，表示这个Log Record保存了中间部分的数据，后面还有数据；

当剩余的数据可以容纳到新的Log Block时，这时候type就是kLastType，表示这个记录的数据结束了，可以和前面的数据组合起来；

kZeroType是为了兼容mmap相关的代码，这种方式会先将数据分配好，置0，所以当读取日志的文件读取这些0时，就可以跳过这些数据，我们不会写入这种类型的日志记录。

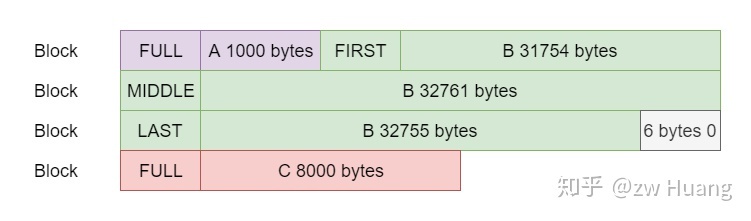
### Log Block举例

举个例子，假设从头开始，写入以下3个记录：

A: length 1000

B: length 97270

C: length 8000



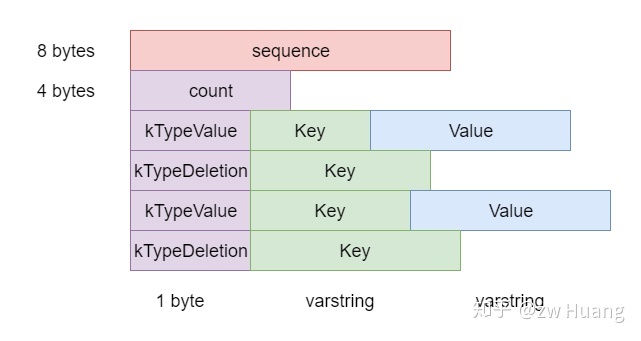
A记录会被写入到第一个Log Block，type为kFullType；

B记录会首先将第一个Log Block填满，type是KFirstType，然后第二个Log Block也会被填满，type是kMiddleType，最后使用第三个Log Block的前面部分，type是kLastType，表示B记录的数据结束了；

记录C发现第三个Log Block只有6个字节了，所以置零，将数据放到第四个Log Block，type是kFullType。

### Log的内容

Log Record里的内容是自解释的。WAL里保存的是对键值对的操作，包括键值对的插入和删除。一个Log Record可能包括多个键值对的操作。



内容先是SequenceNumber，然后count表示有多少个操作记录，接着就是这count个record了。record分为两种，分别是插入和删除，插入需要指定键和值，删除只需要指定键。

Log的内容其实和WriteBatch的内容一样，这种结构可以实现一次性操作多个键值对。

### 参考源码

db/log\_format.h: 定义了RecordType和一些常量

db/log\_writer.h db/log\_writer.cc: 主要实现Writer::AddRecord，写一条记录到日志中

db/log\_reader.h db/log\_reader.cc: 主要实现Reader::ReadRecord，读取一条日志记录

### 小结

日志的实现非常简单，用32KB大小的块对日志进行分割，每个Log Record都有校验，对大的Log Record也不需要缓存。如果一个记录太大也不需要一次性全部读出，这种简单的实现对于LevelDB的场景完全够用。

实际的数据就是二进制数据，是由程序自己解释的，WAL没有规定需要什么格式。

除了WAL用这种日志格式以外，MANIFEST文件的记录也使用了这种格式。

## SSTable

SSTable：Sorted String Table。分为level-0到level-n多层，每一层包含多个SSTable，文件内数据有序。除了level-0之外，每一层内部的SSTable的key范围都不相交。

DB数据持久化文件，内部key是有序的，文件内部前面是数据，后面是索引元数据。

sstable文件之间逻辑上是分层的，**LevelDB最大支持7层**。

将MemTable的镜像写入到磁盘有一个要求：需要具有快速地在磁盘上查询一个键的功能。

最先想到的就是B+ Tree，这是一种古老的磁盘数据结构，现在很多数据库依然在采用，具有很好的读取性能。B+ Tree其实是一种多级索引，设计成这样是为了支持快速读取，同时也支持更新，但是B+ Tree更新的开销会比较大。如果将一个MemTable写入磁盘的方式是将数据原地更新磁盘上已存在的镜像，那么其实就是使用了类似于B+ Tree，这样会有很大的更新开销。所以LevelDB不更新磁盘镜像，每次都将MemTable写入一个新镜像。

对于不更新的磁盘数据结构，并不需要使用B+ Tree。LevelDB使用了SSTable（Sorted String Table），意思是数据按照键的顺序存储在磁盘上。键是有序的，查找键可以采用二分搜索。但是磁盘的二分搜索会读取多个磁盘块。这时候只需要给每个磁盘块一个索引，告诉这个磁盘块里面存储键的范围，那么在查找时，可以先通过对索引进行二分搜索找到键所在的磁盘块，只需要读取这个磁盘块，便可以找到这个键。索引是一个稀疏索引，比较小，可以放在内存中缓存。

定时将MemTable的镜像写入磁盘中的一个SSTable，数据就会分为两个部分：一部分是内存中的MemTable里的数据，一部分是磁盘中SSTable的数据。当写入数据时，还是写入MemTable和日志，而后台线程定时将MemTable的镜像写入磁盘的SSTable中，如果很好的控制后台线程的写入频率，或者SSTable和日志不在同一个磁盘，那么写入依然是比较符合顺序写的，可以有很高的写性能。写入SSTable后，之前的日志就可以被淘汰了，因为之前的数据已经持久化到磁盘上了。这就同时解决了上面的两个问题，恢复时间和数据库容量。

使用定期写入MemTable镜像的方式，解决了恢复和数据库容量的问题，并且数据库具有持久化的功能，在满足这些条件的情况下，数据库依然有很高的写性能，符合LSM Tree的写多的场景。

但是，这又引入了一个新的问题：读变慢了。以前读的时候，只需要读取MemTable，现在还需要读取SSTable。随着SSTable不断地写入，SSTable会越来越多，当查找一个键时，可能需要读取多个SSTable，这就涉及了多次随机读，读取效率会很低。

如果把多个SSTable合并成一个大的SSTable，那么查找时，就不需要读取多个小的SSTable，而只需要读取一个大的SSTable，磁盘IO就会减少。

注：SSTable的合并本质上是去除无用的数据，加速读取的效率。

### Compaction

DB有一个后台线程负责将memtable持久化成sstable，以及均衡整个DB各个level层的sstable。

**compact分为minor compaction和major compaction。memtable持久化成sstable称为minor compaction，level(n)和level(n+1)之间某些sstable的merge称为major compaction**。

将多个小SSTable合并成一个大SSTable，可以解决查找的效率问题。但是，如果不断地有新的小SSTable进来，这些小SSTable都需要和这个大的SSTable进行合并，不管多大的SSTable来合并，都需要读取所有的磁盘数据，并且写入所有的磁盘数据。

大的SSTable是按键顺序存储的，可以将大SSTable进行分割，分割成多个小SSTable，每个SSTable都包含一段键的范围，而每个SSTable键的范围是不重复的，并且是按顺序排列的。这时候物理上存在多个SSTable文件，但是逻辑上依然是有一个大的SSTable。在查找的时候只需要多做一步，根据每个小的SSTable包含键的范围，可以做一个二分搜索，就可以找到实际的键在哪个小SSTable文件里。这样依然只需要读取一个SSTable，但是却使用了多个小文件代替一个大文件。这样的好处就是当有新SSTable需要合并到这个逻辑上的大SSTable时，只需要找到和新SSTable的键范围有重合的小SSTable的物理文件进行合并，这样就可以降低合并需要读取的数据量。

这样磁盘文件实际上分为两部分：一部分是MemTable直接写入的SSTable，另一部分是逻辑上的大SSTable。这实际上就是LevelDB里面的Level 0和Level 1。Level 0文件的键范围可能有重叠，而Level 1不会有重叠。而读取的时候，要读取Level 0和Level 1的数据，如果限制Level 0的文件数量，磁盘的读IO依然可以控制在一个常数范围内。

随着数据越写越多，Level 1越来越大，此时就算Level 0的SSTable很小，依然可能会和Level 1很多的SSTable文件重合，那么读写量依然很大。

这时候还需要改进，需要控制Level 1里所有文件的大小，那么多出的文件该怎么办呢？可以将它们再推到更高的一层Level 2中。Level 2里总文件大小设置为Level 1的10倍，在生成Level 1的SSTable时，控制大小使得最多与10个Level 2的SSTable重叠。如果需要将Level 1的一个SSTable合并到Level 2，需要读取的SSTable的数据量依然是一个常数级，是可控的。如果Level 2满了，可以将SSTable继续推向Level 3。因为每一Level的大小都是指数增长的，所以不需要几层，就可以放大量数据了。

这就是LevelDB名字的由来了，SSTable是分层存储的，Level 0的SSTable之间是重叠的，而Level 0以上的SSTable是有序不重叠的。SSTable在各Level之间移动的过程叫做Compaction，意思是让文件变得更加紧凑，易于查询。Level 0的SSTable会Compaction到Level 1，而Level 1的SSTable会Compaction到Level 2，随着Level的增大，每一层的文件总大小会以10倍增大，这样不但可以有大量的存储空间，而且每一次Compaction涉及的SSTable的数量都是可控的。Compaction实际上就是对输入的多个SSTable进行多路归并的过程。

随着Level的增多，读取更复杂了。要先读取MemTable，再读取Level 0的文件，Level 0可能有多个文件的键范围包括这个查找键，还需要读取Level 0以上的文件，每一Level最多有一个文件的键范围包括查找键。不过Level的数量有限，Level 0的文件数量也有限，所以需要读取的SSTable的数量依然是常数级，配合缓存、布隆过滤器等优化技术，可以提高读的性能。这是在读取性能和后台操作性能之间的折中，为了让写操作成为顺序写，而做的牺牲。

磁盘上有多个SSTable，需要知道每个SSTable属于哪一层，每一个SSTable的键范围，这些信息都存储在内存中。但是如果数据库重启了，就丢失了这些元信息，所以需要将它们持久化到磁盘。

### Manifest

Manifest：Manifest文件中记录SSTable在不同level的信息，包括每一层由哪些SSTable，每个SSTable的文件大小、最大key、最小key等信息。

MANIFEST是用来管理SSTable的，保存了SSTable的元信息。

对于当前数据有哪些SSTable，这些SSTable属于哪一层，每一个SSTable的键范围和文件大小等信息，需要持久化到磁盘上，下一次打开数据库的时候，就可以从磁盘上读取到这些元数据，恢复内存里的数据结构，这个持久化数据就存储在MANIFEST文件中。

随着Compaction的进行，元数据会改变，所以每次还需要将改变的元数据写到MANIFEST中。恢复元数据时，使用初始的元数据和各个改变恢复出最终的元数据。但是如果改变太多，MANIFEST太大，恢复就会太耗时，这时可以将当前的元数据写入到有一个新的MANIFEST中，而舍弃旧的MANIFEST。而CURRENT文件则存储了当前使用的MANIFEST文件是哪一个，写完MANIFEST后，需要将CURRENT指向新的MANIFEST。

### Current

Current：重启时，LevelDB会重新生成Manifest，所以Manifest文件可能同时存在多个，Current记录的是当前使用的Manifest文件名。

### TableCache

TableCache：TableCache用于缓存SSTable的文件描述符、索引和filter。

### BlockCache

BlockCache：SSTable的数据是被组织成一个个block。BlockCache用于缓存这些block（解压后）的数据。

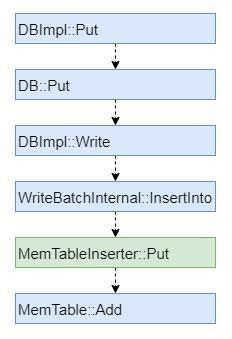
### 版本管理

将每次compact后的最新数据状态定义为一个version，也就是当前DB的元信息以及每层level的sstable的集合。**跟version有关的一个数据结构是VersionEdit，记录了一次version的变化，包括删除了哪些sstable，新增了哪些sstable**。old version + versionedit= new version。整个DB存在的所有version被VersionSet数据结构保存，这个数据结构包含：全局sequencenumber、filenumber、tablecache、每个level中下一次compact要选取的start\_key。

## 操作接口

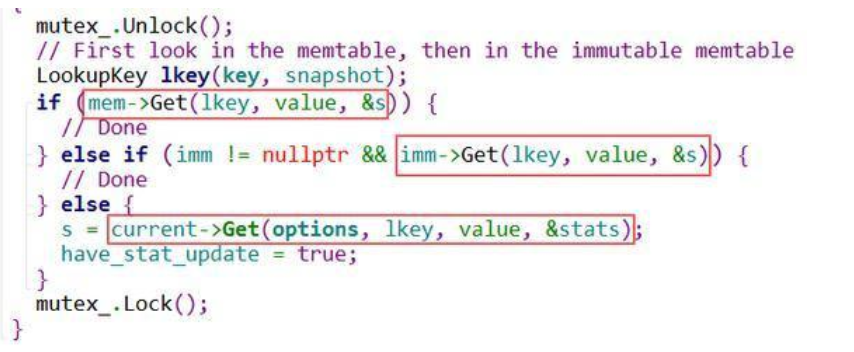
### 写流程

LevelDB的写操作包括设置key-value和删除key两种。需要指出的是这两种情况在LevelDB的处理上是一致的，删除操作其实是向LevelDB插入一条标识为删除的数据。下面我们先看一下LevelDB插入值的整体流程，具体如图所示。



### 读流程

读流程要比写流程简单一些，核心代码逻辑如图所示。



首先，生成内部查询所用的Key，该Key是由用户请求的UserKey拼接上Sequence生成的。其中Sequence可以用户提供或使用当前最新的Sequence，LevelDB可以保证仅查询在这个Sequence之前的写入。然后，用生成的Key，依次尝试从 Memtable，Immtable以及SST文件中读取，直到找到。

从SST文件中查找需要依次尝试在每一层中读取，得益于Manifest中记录的每个文件的key区间，我们可以很方便的知道某个key是否在文件中。Level0的文件由于直接由Immutable Dump 产生，不可避免的会相互重叠，所以需要对每个文件依次查找。对于其他层次，由于归并过程保证了其互相不重叠且有序，二分查找的方式提供了更好的查询效率。

可以看出同一个Key出现在上层的操作会屏蔽下层的。也因此删除Key时只需要在Memtable压入一条标记为删除的条目即可。被其屏蔽的所有条目会在之后的归并过程中清除。

# 应用场景

Level适用于写多读少的场景。