leveldb 简介

Leveldb是一个google实现的非常高效的kv数据库，能够支持billion级别的数据量了。

特点

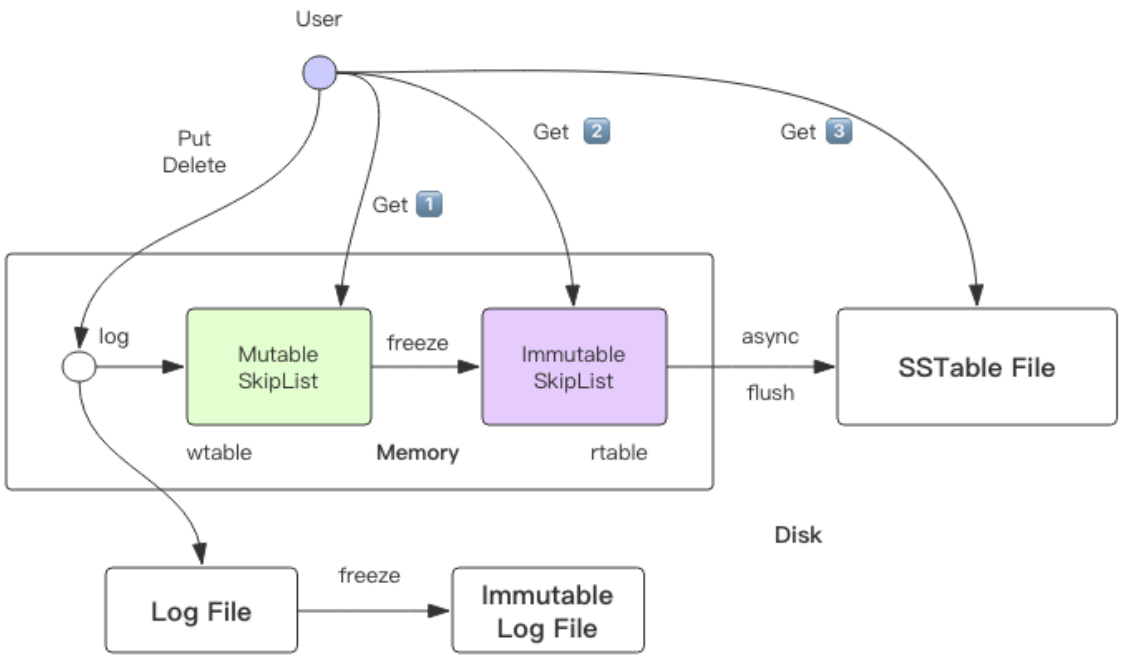
- key、value支持任意的byte类型数组，不单单支持字符串。
- LevelDb是一个持久化存储的KV系统，将大部分数据存储到磁盘上。
- 按照记录key值顺序存储数据，并且LevleDb支持按照用户定义的比较函数进行排序。
- 操作接口简单，基本操作包括写记录，读记录以及删除记录，也支持针对多条操作的原子批量操作。
- 支持数据快照（snapshot）功能，使得读取操作不受写操作影响，可以在读操作过程中始终看到一致的数据。
- 支持数据压缩(snappy压缩)操作，有效减小存储空间、并增快IO效率。

总体来说，LevelDb的写操作要大大快于读操作，而顺序读写操作则大大快于随机读写操作。

限制

- LevelDB 只是一个 C/C++ 编程语言的库, 使用者应该封装自己的网络服务器。 所以无法像一般意义的存储服务器(如 MySQL)那样, 用客户端来连接它。
- 非关系型数据模型（NoSQL），不支持sql语句，也不支持索引。
- 一次只允许一个进程访问一个特定的数据库。

leveldb 的架构



User 代表客户端，提供增删改查功能

```
  virtual Status Put(const WriteOptions&, const Slice& key, const Slice& value);
  virtual Status Delete(const WriteOptions&, const Slice& key);
  virtual Status Write(const WriteOptions& options, WriteBatch* updates);
  virtual Status Get(const ReadOptions& options, const Slice& key, std::string*
value);
```

其中Put/Delete 都会转换成Write 操作. 即delete 操作会通过 `type` 增加一个新的Node, 并不实际删除。

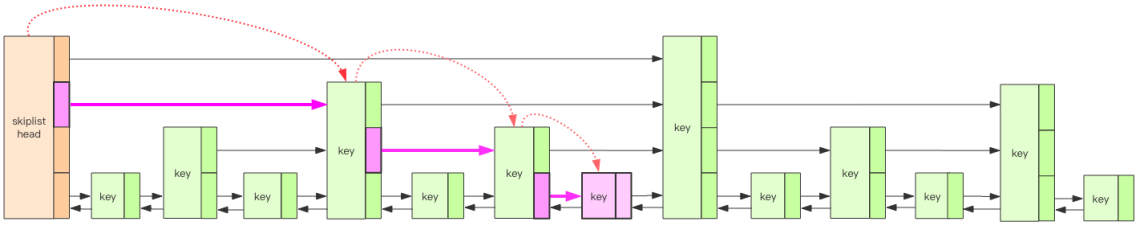
Memory

内存中存在两个MemTable(SkipList)

```
MemTable* mem_; // wtable
MemTable* imm_ GUARDED_BY(mutex_); // rtable
```

- wtable 要支持多线程读写，所以访问它是需要加锁控制。
- rtable 是只读的, 会异步把信息dump到磁盘。

SkipList 结构



SkipList 为Set集合，只有Key。 这里存在多种key 简单介绍以下。

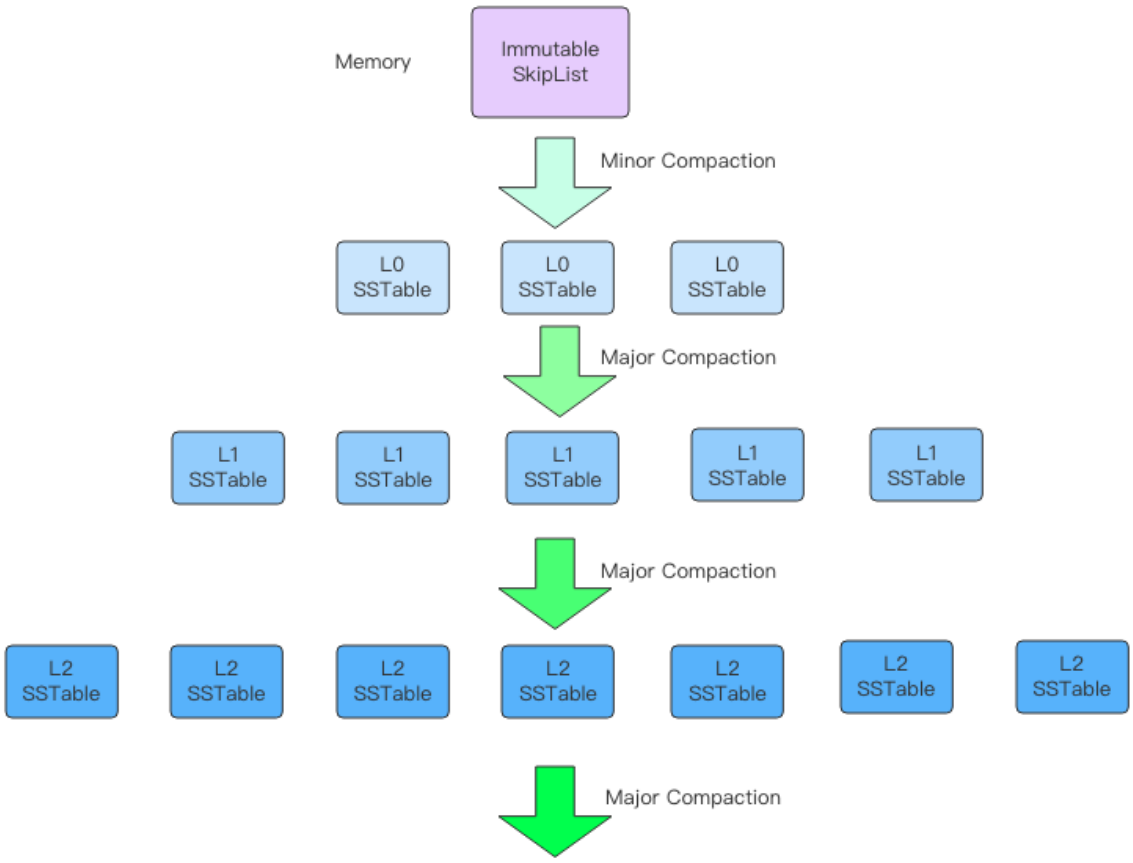
```
key(skiplist_key) = internal_key_size + internal_key + value_size + value
internal_key = user_key + sequence + type
```

- `user_key` 和 `value` 即为客户端提交的KV。
- `sequence` 为 序列号(占7B)
- `type` 标记是 Put 还是 Delete 操作，只有两个取值，0 表示 Delete，1 表示 Put, (占1B)
- 所以在leveldb 中 `sequence<<8|type` 使用 `uint64` 表示
- internal_key_size 和 value_size 都要采用 varint 整数编码。(为什么采用变长编码？ 直接使用 `uint64` 有啥不好的呢？)

LOG

write数据时会先写入log，再写入wtable.
简单来说LevelDB 遇到突发停机事故，没有持久化的 wtable 和 rtable 数据就会丢失。可以通过LOG 进行数据库恢复.

DISK(LSM-Tree)



- SSTable 表示 Sorted String Table，文件里所有的 Key(internel_key) 都会有序的.
- Minor Compaction 表示从rtable 中dump 数据
- Major Compaction 表示从 n 层 sst 文件下沉到 n+1 层 sst 文件.
- L0 之间的文件中的key 不是全局排序，即整体无序， 当个文件中有序
- L1-L6 每层的SSTable(多个) 是整体有序的。不同层间是无序的。
- Major Compaction 采用多路归并。

Write/Get 流程

write 过程

1. wtable size超过阈值则wtable 转化成 rtable,调度器异步的进行步骤3。进入步骤2
2. 先写Log，再把数据更新到wtable. 写流程完毕
3. 调度器负责异步compaction(以下三个步骤没有顺序依赖关系)
    1. 把rtable 刷到磁盘, 然后释放相应的Log文件的空间。即 Minor Compaction
    2. L0 SST文件个数超过阈值，进行Major Compaction。 (归并排序)
    3. L1-L6 同理。

```
// 步骤1
Status status = MakeRoomForWrite(updates == nullptr);
// Put(增/改)、Delete 均为转换为Write 操作
// 步骤2
status = log_->AddRecord(WriteBatchInternal::Contents(updates))
status = WriteBatchInternal::InsertInto(updates, mem_);
// 在MakeRoomForWrite函数中，假设wtable size 超过阈值， 则进行一下操作。
imm_ = mem_; // wtable 转成rtable
mem_ = new MemTable(internal_comparator_);
// 步骤3， 异步进行 compaction
MaybeScheduleCompaction 会调用到BackgroundCompaction
// 如果rtabel 有数据，则进入步骤3.1. 经过一系列的调用，会进入到BuildTable.
```

```
// meta 记录最大值最小值
// TODO: 假设操作为， PUT(KEY, V1), PUT(KEY, V2), DELETE(KEY)
// 那么在保存到磁盘时，可以忽略这个KEY的。
// 暂时没有看到相关逻辑
// iter是rtable 的迭代器
meta->smallest.DecodeFrom(iter->key());
for (; iter->Valid(); iter->Next()) {
  Slice key = iter->key();
  meta->largest.DecodeFrom(key);
  builder->Add(key, iter->value());
}
// 检查是否需要进行步骤3.2， 3.3
// 再次进入BackgroundCompaction(MaybeScheduleCompaction)
// 经过一系列调用到函数DoCompactionWork
// TODO: Compaction需要详细了解，后续展开
// TODO: SSTable 的写过程。 后续会展开。
```

Get 过程

1. 遍历wtable，如果找到则返回，否则进入下一步
2. 遍历rtabel，如果找到则返回，否则进入下一步
3. 遍历磁盘，按照顺序遍历L0 到L7

```
// 根据user_key 和 sequence num 生成 lookup_key
LookupKey lkey(key, snapshot);
// 1. 遍历wtable
if (mem->Get(lkey, value, &s)) {
  // Done
}
// 2. 遍历rtable
else if (imm != nullptr && imm->Get(lkey, value, &s)) {
  // Done
}
// 3. 遍历磁盘
else {
  s = current->Get(options, lkey, value, &stats);
  have_stat_update = true;
```

1, 2 都是遍历SkipList的过程。
对过程3 进行展开

```
Status Version::Get(const ReadOptions& options, const LookupKey& k,
                    std::string* value, GetStats* stats) {
  Slice ikey = k.internal_key();
  Slice user_key = k.user_key();
  const Comparator* ucmp = vset_->icmp_.user_comparator();
  Status s;

  stats->seek_file = nullptr;
  stats->seek_file_level = -1;
  FileMetaData* last_file_read = nullptr;
  int last_file_read_level = -1;

  // We can search level-by-level since entries never hop across
  // levels.  Therefore we are guaranteed that if we find data
  // in a smaller level, later levels are irrelevant.
```

```
  std::vector<FileMetaData*> tmp;
  FileMetaData* tmp2;
  for (int level = 0; level < config::kNumLevels; level++) {
    size_t num_files = files_[level].size();
    if (num_files == 0) continue;

    // Get the list of files to search in this level
    FileMetaData* const* files = &files_[level][0];
    if (level == 0) {
      // Level-0 files may overlap each other.  Find all files that
      // overlap user_key and process them in order from newest to oldest.
      tmp.reserve(num_files);
      for (uint32_t i = 0; i < num_files; i++) {
        FileMetaData* f = files[i];
        if (ucmp->Compare(user_key, f->smallest.user_key()) >= 0 &&
            ucmp->Compare(user_key, f->largest.user_key()) <= 0) {
          tmp.push_back(f);
        }
      }
      if (tmp.empty()) continue;

      std::sort(tmp.begin(), tmp.end(), NewestFirst);
      files = &tmp[0];
      num_files = tmp.size();
    } else {
      // 二分查找
      // 找到第一个sstable，它的最大值大于ikey
      // 每层的sstables 是全局有序的
      // sstable 记录了internal_key 的最大值和最小值
      uint32_t index = FindFile(vset_->icmp_, files_[level], ikey);
      if (index >= num_files) {
        files = nullptr;
        num_files = 0;
      } else {
        tmp2 = files[index];
        if (ucmp->Compare(user_key, tmp2->smallest.user_key()) < 0) {
          // All of "tmp2" is past any data for user_key
          files = nullptr;
          num_files = 0;
        } else {
          files = &tmp2;
          num_files = 1;
        }
      }
    }

    for (uint32_t i = 0; i < num_files; ++i) {
      if (last_file_read != nullptr && stats->seek_file == nullptr) {
        // We have had more than one seek for this read.  Charge the 1st file.
        stats->seek_file = last_file_read;
        stats->seek_file_level = last_file_read_level;
      }

      FileMetaData* f = files[i];
      last_file_read = f;
      last_file_read_level = level;

      Saver saver;
```

```
      saver.state = kNotFound;
      saver.ucmp = ucmp;
      saver.user_key = user_key;
      saver.value = value;
      // TODO 详细过程分析， SSTable 的读过程，后续会展开
      s = vset_->table_cache_->Get(options, f->number, f->file_size, ikey,
                                   &saver, SaveValue);
      if (!s.ok()) {
        return s;
      }
      switch (saver.state) {
        case kNotFound:
          break;  // Keep searching in other files
        case kFound:
          return s;
        case kDeleted:
          s = Status::NotFound(Slice());  // Use empty error message for speed
          return s;
        case kCorrupt:
          s = Status::Corruption("corrupted key for ", user_key);
          return s;
      }
    }
  }
  return Status::NotFound(Slice());
}
```

参考文献

1. https://leveldb-handbook.readthedocs.io/zh/latest/basic.html
2. https://zhuanlan.zhihu.com/p/54510835
3. https://github.com/balloonwj/CppGuide/tree/master/articles/leveldb%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90
4. https://zhuanlan.zhihu.com/p/80684560