leveldb

Table of Contents

http://code.google.com/p/leveldb/

1 Introduction

相关资源：

- 使用文档. http://leveldb.googlecode.com/svn/trunk/doc/index.html
- 设计说明. http://leveldb.googlecode.com/svn/trunk/doc/impl.html
- leveldb和baidu内部kv系统对比. http://hi.baidu.com/little_fxxker/blog/item/1915f300f7548a046b60fb08.html
- http://blog.csdn.net/anderscloud/article/details/7182165

leveldb是一个kv存储系统，其中kv都是二进制。用户接口非常简单就是Put(k,v),Get(k),Delete(k).但是还有以下特性

- k有序存储.因为k是二进制没有解释的所以用户需要提供比较函数
- 支持遍历包括前向和反向
- 支持atomic write
- 支持filter policy(bloomfilter)
- 数据支持自动压缩(使用snappy压缩算法.关于snappy分析可以看这里)
- 底层提供了抽象接口，允许用户定制

当然也存在一定的限制

- 不是SQL数据库，没有数据关系模型
- 一个table只允许一个process访问
- 单机系统没有client-server.

目录层次划分是这样的(意图是我猜想的)

- db // 和db逻辑相关的内容
- helpers // 里面有一个内存db接口
- include // Interface
- port // 操作系统相关的移植接口
- table // 表存储结构
- util // 公用部分.

leveldb还是比较麻烦的.开始阅读的时候(像我)很多策略细节就可以直接忽略.比如什么时候触发compaction的,以及挑选什么层次进行compaction的输出, 选择那些文件进行compaction等.阅读的时候需要了解每个类到底是用来做什么的.个人觉得里面最迷惑的东西就是Version/VersionEdit/VersionSet是用来做什么的. 所谓Version就是做一个compaction时候产生的一个对象.VersionSet是当前DB里面所有的Version.VersionEdit是针对Version的修改.包括添加和删除哪些文件等. 每次compaction时候会产生version表示这个哪些文件是需要的.在回收文件的时候会查看每一个version持有的文件,这样就可以确定哪些文件是不需要的了. 每次进行compaction都会产生这么一个version对象.将对version进行的操作称为version_edit.同时会将这个version_edit写入manifest文件里面去. 这样在恢复DB的时候，首先可以从manifest里面读取到挂掉之前的version是怎么样的.然后通过读取剩余的version_edit得到挂掉之前的version. 同时会读取log文件将挂掉之前操作的kv恢复.

#note: 最近看到一篇文章比较leveldb和mysql存储引擎性能(可能是innodb).里面提到了连续插入性能的抖动很大.这可能 和底层为了达到读取高效率不断地进行compaction有关的.关于compaction挑选以及触发这个策略的话以后可以好好研究一下.

#note: compaction策略没有仔细分析，但是这个部分是精髓。如何控制compaction策略来针对应用达到最好的读写平衡。另外对于Recovery部分没有仔细看代码， 但是我觉得这个部分倒不是很大的问题，可能学到的东西不多但是需要非常仔细地阅读才行。

2 WriteBatch

leveldb使用WriteBatch来达到atomic write操作.WriteBatch过程非常简单，就是将atomic write的内容全部写到一个内存buffer上，然后提交这个WriteBatch. 至于具体的分析可以查看"Code Analysis/Batch/WriteBatch"这节的分析。使用WriteBatch一方面可以做到原子操作，另外一方面可以提高吞吐。

3 BloomFilter

相关资源：

- Bloom Filter. http://en.wikipedia.org/wiki/Bloom_filter
- LevelDB Bloom Filter实现. http://duanple.blog.163.com/blog/static/7097176720123227403134/

bloom filter原理非常简单，似乎没有必要详细分析。关于代码部分的话可以看Code Analysis/Util/BloomFilter. 至于filter在磁盘上面是如何存储的可以参看下面一节Storage/DiskTable分析。

meta block存放了bloom filter信息，这样可以减少磁盘读取。关于Table内部支持bloom filter在table/filter_block.h有实现。 分别是FilterBlockBuilder和FilterBlockReader.

3.1 Format

leveldb是这么分配filter block的.以base(2KB)计算.如果block offset在[base*i,base*(i+1)-1]之间的话，那么就在filter i上面。存储格式是这样的。

```
[filter 0]
[filter 1]
[filter 2]
...
[filter N-1]
[offset of filter 0]                 : 4 bytes
[offset of filter 1]                 : 4 bytes
[offset of filter 2]                 : 4 bytes
...
[offset of filter N-1]               : 4 bytes
[offset of beginning of offset array] : 4 bytes
lg(base)                             : 1 byte
```

那么这个就是一个filter block的格式。filter block存放在meta block里面。在meta index block内部会记录key,filter block handle.其中key就是这个filter的名字,handle就是这个filter block offset.看看下面代码会更容易理解。

3.2 FilterBlockBuilder

对于Table在初始化之前会调用StartBlock.并且在每次进行Flush Data Block时候也会根据Data Block offset调用。

```
void FilterBlockBuilder::StartBlock(uint64_t block_offset) {
  uint64_t filter_index = (block_offset / kFilterBase);
  assert(filter_index >= filter_offsets_.size());
  while (filter_index > filter_offsets_.size()) {
    GenerateFilter();
  }
}
```

可以看到两个data block offset跨越超过base的话那么会产生几个empty filter.但是默认实现的话empty filter不占用太多空间。

然后每次Table在AddKey时候也会调用FilterBlock::AddKey

```
void FilterBlockBuilder::AddKey(const Slice& key) {
  Slice k = key;
  start_.push_back(keys_.size());
  keys_.append(k.data(), k.size());
}
```

注意这里keys_是一个string.start_记录每个新增key的偏移。AddKey是将这段时间内添加的Key全部缓存下来。

然后每次Flush的时候都会产生filter.

```
void FilterBlockBuilder::GenerateFilter() {
  const size_t num_keys = start_.size();
  if (num_keys == 0) {
    // Fast path if there are no keys for this filter
    filter_offsets_.push_back(result_.size());
    return;
  }

  // Make list of keys from flattened key structure
  start_.push_back(keys_.size());  // Simplify length computation
  tmp_keys_.resize(num_keys);
  for (size_t i = 0; i < num_keys; i++) {
    const char* base = keys_.data() + start_[i];
    size_t length = start_[i+1] - start_[i];
    tmp_keys_[i] = Slice(base, length);
  }

  // Generate filter for current set of keys and append to result_.
  filter_offsets_.push_back(result_.size()); // 记录每个filter的偏移.
  policy_->CreateFilter(&tmp_keys_[0], num_keys, &result_);

  tmp_keys_.clear();
  keys_.clear();
  start_.clear();
}
```

最后filter block需要刷新出去调用Flush方法。

```
Slice FilterBlockBuilder::Finish() {
  if (!start_.empty()) {
    GenerateFilter();
  }

  // Append array of per-filter offsets
  const uint32_t array_offset = result_.size();
  for (size_t i = 0; i < filter_offsets_.size(); i++) {
    PutFixed32(&result_, filter_offsets_[i]); // 这里使用Fixed32表示也非常好理解
    // 这样才能快速地映射到对应的filter上面。
  }

  PutFixed32(&result_, array_offset); // 这个array offset表示filter offset的起始地址
  result_.push_back(kFilterBaseLg);  // Save encoding parameter in result
  return Slice(result_); // 这个slice就是最终需要write的数据.
}
```

3.3 FilterBlockReader

了解上面的filter block的存储格式之后Reader就非常简单。构造函数首先计算出各个参数。simple huh?

```
FilterBlockReader::FilterBlockReader(const FilterPolicy* policy,
                                     const Slice& contents)
    : policy_(policy),
      data_(NULL),
      offset_(NULL),
      num_(0),
      base_lg_(0) {
  size_t n = contents.size();
  if (n < 5) return;  // 1 byte for base_lg_ and 4 for start of offset array
  base_lg_ = contents[n-1];
  uint32_t last_word = DecodeFixed32(contents.data() + n - 5);
  if (last_word > n - 5) return;
  data_ = contents.data();
  offset_ = data_ + last_word;
  num_ = (n - 5 - last_word) / 4;
}
```

阅读完成后面的Storage一节之后就会发现query key的话首先是在data index block找到这个key所在的

data block offset的。 所以这里filter就是判断某个offset的data block是否含所有key.

```
bool FilterBlockReader::KeyMayMatch(uint64_t block_offset, const Slice& key) {
  uint64_t index = block_offset >> base_lg_;
  if (index < num_) {
    uint32_t start = DecodeFixed32(offset_ + index*4); // filter起始地址
    uint32_t limit = DecodeFixed32(offset_ + index*4 + 4); // filter终止地址
    if (start <= limit && limit <= (offset_ - data_)) {
      Slice filter = Slice(data_ + start, limit - start);
      return policy_->KeyMayMatch(key, filter); // filter判断是否存在key.
    } else if (start == limit) {
      // Empty filters do not match any keys
      return false;
    }
  }
  return true;  // Errors are treated as potential matches
}
```

4 Storage

相关资源：

- Table Format. http://leveldb.googlecode.com/svn/trunk/doc/table_format.txt sst table存储格式。
- Log Format. http://leveldb.googlecode.com/svn/trunk/doc/log_format.txt block存储格式。
- LevelDB SSTable格式详解. http://wenku.baidu.com/view/19f83f51be23482fb4da4c29.html

4.1 MemTable

memtable在leveldb内部实现就是一个skiplist.所有的update都不是in-place的，对于memtable里面的修改来说的话使用的也是使用添加的方式完成的。 对于每个操作都会分配一个sequence number.所以底层也没有办法直接覆盖。对于sequence number很明显就是需要实现snapshot.底层的话leveldb 持有两个memtable,一个memtable用于接收当前的操作是mutable的，一个memtable是immutable专门用于dump to disk的，内部实现类似于双buffer机制。

4.2 RedoLog

我们首先阅读Log Format文档看看log存储格式(leveldb采用redo-log来记日志)。每个block都划分成为32KB，里面可能会存在很多条记录， 对于跨块的记录来说的里面存在type字段用来标记这个块是否已经结束。另外值得注意的就是每个记录之前带上了32bit的checksum.对于每条记录多4字节还是很大开销的， 但是其实这也反应了leveldb的定位，就是针对fault-tolerant的分布式系统设计。这些分布式系统架在commodity PC上面，磁盘可能很容易出现问题。 在文档最后作者也给给出了这种block存储方式(recordio)的利弊。

```
Some benefits over the recordio format:

(1) We do not need any heuristics for resyncing - just go to next
block boundary and scan.  If there is a corruption, skip to the next
block.  As a side-benefit, we do not get confused when part of the
contents of one log file are embedded as a record inside another log
file.

(2) Splitting at approximate boundaries (e.g., for mapreduce) is
simple: find the next block boundary and skip records until we
hit a FULL or FIRST record.

(3) We do not need extra buffering for large records.

Some downsides compared to recordio format:

(1) No packing of tiny records.  This could be fixed by adding a new
record type, so it is a shortcoming of the current implementation,
not necessarily the format.

(2) No compression.  Again, this could be fixed by adding new record types.
```

pros有：

- 如果磁盘数据发生损坏的话，那么对于数据定位的话非常简单。如果这个block出现问题的话那么直接跳过这个block即可。
- 程序处理方面可以很容易地找到边界。
- 对于单条大数据处理的话我们不需要分配很大的内存来做buffer.

cons有：

- 没有针对小record进行优化，比如如果record足够小的话完全可以在length部分使用1个字节。
- 没有进行压缩。对于压缩率完全取决于实现。对于小数据来说的话压缩比可能不高，对于大数据来说比如超过32KB的话，

取决于是按照32KB单个block压缩呢(压缩率可能不高),还是先针对整体压缩(压缩率可能比较耗，但是却需要很大的buffer).

4.3 DiskTable

然后可以看看Table Format文档关于table存储格式。table存储格式里面主要包括几个部分：

- data block
- meta block
- meta index block
- data index block
- footer

footer部分是放在最末尾的，里面包含了data index block以及meta index block的偏移信息，读取table时候从末尾读取。

首先我们看看data block是如何组织的。对于DiskTable(TableBuilder)就是不断地Add(Key,Value).当缓存的数据达到一定大小之后， 就会调用Flush这样就形成了一个Block.对于一个Block内部而言的话，有个很重要的概念就是restart point.所谓restart point就是为了解决 前缀压缩的问题的，所谓的restart point就是基准key。假设我们顺序加入abcd,abce,abcf.我们以abcd为restart point的话，那么abce可以存储为 (3,e),abcf存储为(3,f).对于restart point采用全量存储，而对于之后的部分采用增量存储。一个restart block可能存在多个restart point, 将这些restart point在整个table offset记录下来，然后放在data block最后面。每个data block尾部还有一个type和CRC32.其中type可以选择是否 需要针对这个data block进行snappy压缩，而CRC32是针对这个data block的校验。

data index block组织形式和data block非常类似，只不过有两个不同。1)data index block从不刷新直到Table构造完成之后才会刷新，所以 对于一个table而言的话只有一个data index block.2)data index block添加的key/value是在data block形成的时候添加的，添加key非常取巧 ，是上一个data block和这个data block的一个key seperator.比如上一个data block的max key是abcd,而这个data block的min key是ad.那么这个 seperator可以设置成为ac.seperator的生成可以参考Comparator.使用尽量短的seperator可以减小磁盘开销并且提高效率。而对于添加的value就是 这个data block的offset.同样在data index block也会存在restart point.

然后看看进行一个key的query是如何进行的。首先读取出data index block(这个部分可以常驻内存)，得到里面的restart point部分。针对restart point 进行二分。因为restart point指向的key都是全量的key.如果确定在某两个restart point之间之后，就可以遍历这个restart point之间范围分析seperator. 得到想要查找的seperator之后对应的value就是某个data block offset.读取这个data block和之前的方法一样就可以查找key了。对于遍历来说，过程是一样的。

这里我们稍微分析一下这样的工作方式的优缺点。对于写或者是merge来说的话，效率相当的高，所有写都是顺序写并且还可以进行压缩。影响写效率的话一个重要参数就是flush block的参数。 但是对于读来说的话，个人觉得过程有点麻烦，但是可以实现得高效率。对于flush block调节会影响到data index block和data block占用内存大小。如果flush block过大的话， 那么会造成data index block耗费内存小，但是每次读取出一个data block内存很大。如果flush block过小的话，那么data index block耗费内存很

大，但是每次读取data block内存很小。 而restart point数量会影响过多的话，那么可能会占用稍微大一些的内存空间，但是会使得查找过程更快(遍历数更少).

5 Compaction

对于Compaction触发的策略牵扯到了算法问题，自己表示没有仔细看这个策略(其实当时看了但是完全没有理解).这里谈谈compaction如何删除文件的问题。 在leveldb里面每次做一个compaction都会产生一个version对象添加到versionset里面，version里面包含了这个version管理了哪些文件。 每次进行读取都会从某个version读取，然后针对这个version做一个引用计数。然后每次需要删除一些不必要的文件时候就会遍历versionset了解哪些文件 还需要，然后对比文件系统目录下面的文件就知道哪些文件不再需要，即可删除。

#note: 这里稍微总结一下 http://leveldb.googlecode.com/svn/trunk/doc/impl.html 提到的compaction策略。可能阅读完了这些策略之后反过头来看看 代码可能会更好，只是记得当时阅读compaction策略太痛苦了所以直接忽略了。

每个level都有一定的大小限制，并且每个level里面的文件的key都是不会overlap的(L0除外).触发条件很多，文档上描述是某个level超过一定限制。 但是之前阅读代码发现其实并不是这样的，可以参看函数VersionSet::PickCompaction.可以看到有两个触发条件size_compaction和seek_compaction. 所谓的size_compaction就是说某个level超过一定大小，而seek_compaction指某个文件被seek次数超过一定次数之后会触发(关于这个值的更新可以查看VersionSet::Builder::Apply, 在一个文件初始创建的时候就已经设置好了allowed_seeks次数).

前面是触发条件，后面来说说compaction策略.文档上描述非常简单但是事实不是这样。如果需要compact某个level的话，如果level>0的话那么对于这个level 只会选出一个file来和level+1中存在overlap的文件进行合并然后生成一个新的文件。如果level==0的话那么对于这个level可能选择多个文件出来和level+1中overlap 文件合并。对于选取level中文件来说的话是采用rotate keyspace的方式来挑选的。在生成新文件的时候，通常会有两个情况拆分出一个新文件。1) 文件过大 2)文件和level+2中超过10个存在overlap. 2)情况非常好理解，因为如果产生一个大文件和level+2 overlap文件数量过多的话，那么进行level+1的compaction 时间就会非常长并且随机读非常严重。

http://leveldb.googlecode.com/svn/trunk/doc/impl.html 文档Timing这节个人感觉非常有价值。作者估算了一下compaction对于整个系统带宽带来的影响。 按照2MB一个sst文件在level(>0)上面的compaction来计算的话，一次compaction需要read 26MB和write 26MB~=50MB.假设磁盘带宽100MB/s我们通过后台线程限制速度的话， 那么做compaction需要耗费5s时间。假设用户写速度也在10MS/s的话，那么会生成50MB数据相当于25个sst level0文件。这样对读来说会造成很大影响。 作者给出的建议包括：

```
Solution 1: To reduce this problem, we might want to increase the log switching threshold when the number
Though the downside is that the larger this threshold, the more memory we will need to hold the correspond

Solution 2: We might want to decrease write rate artificially when the number of level-0 files goes up.

Solution 3: We work on reducing the cost of very wide merges. Perhaps most of the level-0 files will have
in the cache and we will only need to worry about the O(N) complexity in the merging iterator.
```

其中第二点感觉非常好就是认为控制写入速度当level0文件过多的时候。在db_impl.cc DBImpl::MakeRoomForWrite这个应该是在memtable缺少空间的时候的函数.

```
  allow_delay &&
  versions_->NumLevelFiles(0) >= config::kL0_SlowdownWritesTrigger) {
// We are getting close to hitting a hard limit on the number of
// L0 files.  Rather than delaying a single write by several
// seconds when we hit the hard limit, start delaying each
// individual write by 1ms to reduce latency variance.  Also,
// this delay hands over some CPU to the compaction thread in
// case it is sharing the same core as the writer.
mutex_.Unlock();
env_->SleepForMicroseconds(1000);
allow_delay = false;  // Do not delay a single write more than once
mutex_.Lock();
```

6 Recovery

#note: 这里稍微总结一下 http://leveldb.googlecode.com/svn/trunk/doc/impl.html 提到的关于recovery的部分。幸运的是在阅读这个文档的时候 也让我重新仔细地思考了一下这个recovery过程可能会如何进行的。

我们主要关注三个数据的恢复：

- 用户的data(log)
- leveldb所管理的文件(MANIFEST)
- 内部生成的sequence number(MANIFEST)

对于用户的data而言可以通过记录log来完成。注意这个log里面都是db的insert/delete等操作。值得注意的是，每次生成新的memtable也会生成新的log文件。 这点是非常必要的，因为这样才可以将需要恢复哪些log对应起来。并且log里面每条日志都带上了sequence number,所以log里面的sequence number也有助于 sequence number恢复。

记录leveldb所管理的文件非常简单。我们观察管理文件变化只会发生在compaction的时候，在当前version下面删除一部分文件生成一部分文件。我们将 这些变化称为VersionEdit.每次compaction完成之后的话我们将这个version edit记录在MANIFEST内部，同时生成一个Version。version edit是增量,version是全量。 (至于如何记录这个没有仔细看.但是看代码里面似乎有全量也有增量的记录).如果创建一个新的MANIFEST文件的话，会将MANIFEST文件名称记录在CURRENT内部。 这样启动之后就知道读取哪个MANIFEST文件了。当然记录在MANIFEST内部的不仅仅是文件的变化，还有生成这个Version时候对应的log以及sequence number.

这样我们的recovery过程就非常简单了。读取CURRENT文件知道读取哪个MANIFEST文件。从MANIFEST文件里面构造Version并且回放VersionEdit. 根据当前的状态知道需要读取哪些log.然后回放log更新sequence number等状态。

7 Snapshot

Snapshot集合在leveldb里面组织成为一个链表，oldest的节点必然最小的snapshot。对于每一个snapshot配备一个sequence number, 所以很明显oldest的节点的sequence number应该是最小的。每次进行compaction的时候会判断当前最小的sequence number 是多少然后将一些不必要的节点删除。另外在查询key的时候也会结合这个snapshot sequence number结合成为一个复合key进行查询。

8 Cache

对于leveldb来说的话存在两个cache系统，一个是TableCache，一个是BlockCache.其中TableCache是用来缓存文件描述符的， 而BlockCache是用来做data block的缓存的(Table::BlockeReader).对于leveldb只有一个cache实现在Code Analysis/Cache里面做了详细分析。

我们这里最感兴趣的东西，应该就是每个cache的kv分别是什么。对于TableCahce的k是file_number,v是Table的Iterator (Table::NewIterator).对于leveldb来说的话文件的file_number都是自增的所以使用file_number没有任何问题。对于BlockCache 来说的话k是(cache_id,offset),v是Block的内存。(#todo: 对于这个cache_id现在还不是非常理解，但是个人觉得 这个cache_id可以==file_number.使用cache_id就是每次Open的时候这个cache_id都会改变)

和BlockCache是针对disk block来进行cache的，另外一种cache方案就是Record Cache.相对Block Cache,Record Cache无疑更能够 提高使用效率包括内存大小以及Cache命中率。但是大家拒绝在内部使用RecordCache的原因非常简答，就是这个在应用层完成似乎更好， 应用层可以更好地进行Cache。在应用层完成同时会引入一个问题就是Cache一致性，但是其实维持这个一致性并不是一件很复杂的事情， Cache主要用来解决读取问题，做写穿透并且让Cache失效即可。leveldb维护BlockCache一致性并

不麻烦，因为leveldb的update并不是in-place的。

#note: 不过后来仔细想了一下觉得Record Cache还是在应用层做比较好，可以控制缓存策略比如大小失效时间。对于底层库还是在做BlockCache会比较好一些.

9 Option

在options.h里面有一些leveldb可选的选项。

- comparator.用户可以指定比较器
- create_if_missing.如果数据库不存在就创建
- error_if_exists.如果数据库存在就报错
- paranoid_checks.尽可能多地进行错误检查
- env.用户可以模拟db环境
- info_log.leveldb本身logger.
- write_buffer_size.memtable大小
- max_open_files.最大打开fd数量
- block_cache.Table读取data block的cache.
- block_size.Table里面Block大小
- block_restart_interval.在一个Block里面每隔多少个key创建一个restart point.
- compression.DataBlock是否需要压缩
- filter_policy.过滤策略默认就是bloom filter.
- verify_checksums.读取block时候是否校验checksum
- fill_cache.读取block是否会Cache.通常scan时候不要做cache
- sync.leveldb内部发起write的话是否会调用fsync.

10 ReadingCode

10.1 Interface

我们首先看看leveldb给我们暴露的头文件有哪些.稍微简单地看看接口提供了哪些功能.头文件目录是include/leveldb

- cache.h // kv内存cache接口
- c.h // leveldb C接口
- comparator.h // Slice的比较接口
- db.h // db对象接口
- env.h // 和环境相关的接口
- filter_policy.h // 过滤策略相关的接口
- iterator.h // 遍历接口
- options.h // db操作的选项对象
- slice.h // db操作的key对象(Slice)
- status.h // db操作返回状态的对象
- table_builder.h // 构建table
- table.h // immutable persistent sortedtable.
- write_batch.h // 批量(原子)写入对象

leveldb里面对象的实现方式，很多都是采用抽象类然后提供工厂模式来创建的，这样的话允许用户更换具体实现。

10.1.1 cache.h

Cache主要是用来作为kv查询cache部分.Cache接口非常简单，其中Handle是了为了管理cache item.注释

写得非常好

```
class Cache {
 public:
  Cache() { }

  // Destroys all existing entries by calling the "deleter"
  // function that was passed to the constructor.
  virtual ~Cache();

  // Opaque handle to an entry stored in the cache.
  struct Handle { };

  // Insert a mapping from key->value into the cache and assign it
  // the specified charge against the total cache capacity.
  //
  // Returns a handle that corresponds to the mapping.  The caller
  // must call this->Release(handle) when the returned mapping is no
  // longer needed.
  //
  // When the inserted entry is no longer needed, the key and
  // value will be passed to "deleter".
  virtual Handle* Insert(const Slice& key, void* value, size_t charge,
                         void (*deleter)(const Slice& key, void* value)) = 0;

  // If the cache has no mapping for "key", returns NULL.
  //
  // Else return a handle that corresponds to the mapping.  The caller
  // must call this->Release(handle) when the returned mapping is no
  // longer needed.
  virtual Handle* Lookup(const Slice& key) = 0;

  // Release a mapping returned by a previous Lookup().
  // REQUIRES: handle must not have been released yet.
  // REQUIRES: handle must have been returned by a method on *this.
  virtual void Release(Handle* handle) = 0;

  // Return the value encapsulated in a handle returned by a
  // successful Lookup().
  // REQUIRES: handle must not have been released yet.
  // REQUIRES: handle must have been returned by a method on *this.
  virtual void* Value(Handle* handle) = 0;

  // If the cache contains entry for key, erase it.  Note that the
  // underlying entry will be kept around until all existing handles
  // to it have been released.
  virtual void Erase(const Slice& key) = 0;

  // Return a new numeric id.  May be used by multiple clients who are
  // sharing the same cache to partition the key space.  Typically the
  // client will allocate a new id at startup and prepend the id to
  // its cache keys.
  virtual uint64_t NewId() = 0;

 private:
  void LRU_Remove(Handle* e);
  void LRU_Append(Handle* e);
  void Unref(Handle* e); // 可以看到Handle可能实际上底层有引用计数存在.

  struct Rep;
  Rep* rep_; // opaque实现指针的存在.
};
```

一般底层Handle有引用计数，然后调用Release的话会atomic dec.如果==0的话那么会调用Insert时候的deleter接口进行释放。 这里稍微总结一下Cache提供的接口

- Insert // 插入kv返回Handle
- Lookup // 查询k返回Handle
- Value // 传入handle得到对应的value
- Erase // 删除kv
- NewId // 产生一个新id.

实现是ShardedLRUCache.这个后面会具体分析.

```
extern Cache* NewLRUCache(size_t capacity);
Cache* NewLRUCache(size_t capacity) {
  return new ShardedLRUCache(capacity);
}
```

10.1.2 comparator.h

comparator提供了slice对象的比较方法接口.但是还有两个接口值得提一下

```
// Advanced functions: these are used to reduce the space requirements
// for internal data structures like index blocks.

// If *start < limit, changes *start to a short string in [start,limit).
// Simple comparator implementations may return with *start unchanged,
// i.e., an implementation of this method that does nothing is correct.
virtual void FindShortestSeparator(
    std::string* start,
    const Slice& limit) const = 0;

// Changes *key to a short string >= *key.
// Simple comparator implementations may return with *key unchanged,
// i.e., an implementation of this method that does nothing is correct.
virtual void FindShortSuccessor(std::string* key) const = 0;
```

关于这两个接口注释都解释得十分清楚了。

实现是ByteWiseComparatorImpl.

```
// Intentionally not destroyed to prevent destructor racing
// with background threads.
static const Comparator* bytewise = new BytewiseComparatorImpl;

const Comparator* BytewiseComparator() {
  return bytewise;
}
```

10.1.3 db.h

db.h下面有几个对象

- Snapshot(接口)
- Range
- DB(接口)

Snapshot结构非常简单，只是提供了一些虚析构函数接口。实现是SnapshotImpl.

Range的话就是两个Slice表示范围，表示[start,limit) #+BEGGIN_SRC C++ // A range of keys struct Range { Slice start; / *Included in the range Slice limit;* / Not included in the range

Range() { } Range(const Slice& s, const Slice& l) : start(s), limit(l) { } }; #+END_SRC

DB是我们发起操作的对象。然后我们看看最关键的DB提供了哪些接口.注释写得清晰真的没有必要解释了:).

```
// A DB is a persistent ordered map from keys to values.
// A DB is safe for concurrent access from multiple threads without
// any external synchronization.
class DB {
 public:
  // Open the database with the specified "name".
  // Stores a pointer to a heap-allocated database in *dbptr and returns
  // OK on success.
  // Stores NULL in *dbptr and returns a non-OK status on error.
  // Caller should delete *dbptr when it is no longer needed.
  static Status Open(const Options& options,
                     const std::string& name,
                     DB** dbptr);

  DB() { }
  virtual ~DB();

  // Set the database entry for "key" to "value".  Returns OK on success,
  // and a non-OK status on error.
  // Note: consider setting options.sync = true.
  virtual Status Put(const WriteOptions& options,
                     const Slice& key,
                     const Slice& value) = 0;

  // Remove the database entry (if any) for "key".  Returns OK on
  // success, and a non-OK status on error.  It is not an error if "key"
  // did not exist in the database.
  // Note: consider setting options.sync = true.
  virtual Status Delete(const WriteOptions& options, const Slice& key) = 0;

  // Apply the specified updates to the database.
  // Returns OK on success, non-OK on failure.
  // Note: consider setting options.sync = true.
  virtual Status Write(const WriteOptions& options, WriteBatch* updates) = 0;

  // If the database contains an entry for "key" store the
  // corresponding value in *value and return OK.
  //
  // If there is no entry for "key" leave *value unchanged and return
  // a status for which Status::IsNotFound() returns true.
  //
  // May return some other Status on an error.
  virtual Status Get(const ReadOptions& options,
                     const Slice& key, std::string* value) = 0;

  // Return a heap-allocated iterator over the contents of the database.
  // The result of NewIterator() is initially invalid (caller must
  // call one of the Seek methods on the iterator before using it).
  //
  // Caller should delete the iterator when it is no longer needed.
  // The returned iterator should be deleted before this db is deleted.
  virtual Iterator* NewIterator(const ReadOptions& options) = 0;

  // Return a handle to the current DB state.  Iterators created with
  // this handle will all observe a stable snapshot of the current DB
  // state.  The caller must call ReleaseSnapshot(result) when the
  // snapshot is no longer needed.
  virtual const Snapshot* GetSnapshot() = 0;

  // Release a previously acquired snapshot.  The caller must not
  // use "snapshot" after this call.
  virtual void ReleaseSnapshot(const Snapshot* snapshot) = 0;

  // DB implementations can export properties about their state
  // via this method.  If "property" is a valid property understood by this
  // DB implementation, fills "*value" with its current value and returns
  // true.  Otherwise returns false.
  //
  //
  // Valid property names include:
  //
  //  "leveldb.num-files-at-level<N>" - return the number of files at level <N>,
  //     where <N> is an ASCII representation of a level number (e.g. "0").
  //  "leveldb.stats" - returns a multi-line string that describes statistics
  //     about the internal operation of the DB.
  //  "leveldb.sstables" - returns a multi-line string that describes all
  //     of the sstables that make up the db contents.
  virtual bool GetProperty(const Slice& property, std::string* value) = 0;

  // For each i in [0,n-1], store in "sizes[i]", the approximate
  // file system space used by keys in "[range[i].start .. range[i].limit)".
  //
  // Note that the returned sizes measure file system space usage, so
  // if the user data compresses by a factor of ten, the returned
  // sizes will be one-tenth the size of the corresponding user data size.
  //
  // The results may not include the sizes of recently written data.
  virtual void GetApproximateSizes(const Range* range, int n,
                                   uint64_t* sizes) = 0;

  // Compact the underlying storage for the key range [*begin,*end].
  // In particular, deleted and overwritten versions are discarded,
  // and the data is rearranged to reduce the cost of operations
  // needed to access the data.  This operation should typically only
  // be invoked by users who understand the underlying implementation.
  //
  // begin==NULL is treated as a key before all keys in the database.
  // end==NULL is treated as a key after all keys in the database.
  // Therefore the following call will compact the entire database:
  //    db->CompactRange(NULL, NULL);
  virtual void CompactRange(const Slice* begin, const Slice* end) = 0;
};
```

实现是DBImpl.这里稍微总结一下DB提供的接口

- Open // 创建DB
- Put //

- Delete //
- Write // batch(atomic)写入.
- Get //
- NewInterator // 创建迭代器
- GetSnapshot // 创建snapshot
- ReleaseSnapShot // 销毁snapshot
- GetProperty // 获取属性
- GetApproximateSizes // 根据range这个部分内容占用磁盘大小
- CompactRange // 压缩range这个部分内容

此外还提供了两个函数用于删除DB以及修复DB.这个会单独讨论.

```
// Destroy the contents of the specified database.
// Be very careful using this method.
Status DestroyDB(const std::string& name, const Options& options);

// If a DB cannot be opened, you may attempt to call this method to
// resurrect as much of the contents of the database as possible.
// Some data may be lost, so be careful when calling this function
// on a database that contains important information.
Status RepairDB(const std::string& dbname, const Options& options);
```

10.1.4 env.h

env.h里面抽象了环境，这样用户可以很方便低进行定制.可以看到leveldb大量的设计依赖于虚函数。 对于这种存储系统用虚函数带来的开销应该是可以接受的。但是对于压缩或者是传输协议的话那么虚函数开销就不可以忽略了。 提供提供了默认环境实现在util/env_posix.cc里面。这个会在后面详细分析。所谓环境包括下面几个对象.

1. Env

```
class Env {
 public:
  Env() { }
  virtual ~Env();

  // Return a default environment suitable for the current operating
  // system.  Sophisticated users may wish to provide their own Env
  // implementation instead of relying on this default environment.
  //
  // The result of Default() belongs to leveldb and must never be deleted.
  static Env* Default();

  // Create a brand new sequentially-readable file with the specified name.
  // On success, stores a pointer to the new file in *result and returns OK.
  // On failure stores NULL in *result and returns non-OK.  If the file does
  // not exist, returns a non-OK status.
  //
  // The returned file will only be accessed by one thread at a time.
  virtual Status NewSequentialFile(const std::string& fname,
                                   SequentialFile** result) = 0;

  // Create a brand new random access read-only file with the
  // specified name.  On success, stores a pointer to the new file in
  // *result and returns OK.  On failure stores NULL in *result and
  // returns non-OK.  If the file does not exist, returns a non-OK
  // status.
  //
  // The returned file may be concurrently accessed by multiple threads.
  virtual Status NewRandomAccessFile(const std::string& fname,
                                     RandomAccessFile** result) = 0;

  // Create an object that writes to a new file with the specified
  // name.  Deletes any existing file with the same name and creates a
  // new file.  On success, stores a pointer to the new file in
  // *result and returns OK.  On failure stores NULL in *result and
  // returns non-OK.
  //
  // The returned file will only be accessed by one thread at a time.
  virtual Status NewWritableFile(const std::string& fname,
                                 WritableFile** result) = 0;

  // Returns true iff the named file exists.
  virtual bool FileExists(const std::string& fname) = 0;

  // Store in *result the names of the children of the specified directory.
  // The names are relative to "dir".
  // Original contents of *results are dropped.
  virtual Status GetChildren(const std::string& dir,
                             std::vector<std::string>* result) = 0;

  // Delete the named file.
  virtual Status DeleteFile(const std::string& fname) = 0;

  // Create the specified directory.
  virtual Status CreateDir(const std::string& dirname) = 0;

  // Delete the specified directory.
  virtual Status DeleteDir(const std::string& dirname) = 0;

  // Store the size of fname in *file_size.
  virtual Status GetFileSize(const std::string& fname, uint64_t* file_size) = 0;

  // Rename file src to target.
  virtual Status RenameFile(const std::string& src,
                            const std::string& target) = 0;

  // Lock the specified file.  Used to prevent concurrent access to
  // the same db by multiple processes.  On failure, stores NULL in
  // *lock and returns non-OK.
  //
  // On success, stores a pointer to the object that represents the
  // acquired lock in *lock and returns OK.  The caller should call
  // UnlockFile(*lock) to release the lock.  If the process exits,
  // the lock will be automatically released.
  //
  // If somebody else already holds the lock, finishes immediately
  // with a failure.  I.e., this call does not wait for existing locks
  // to go away.
  //
  // May create the named file if it does not already exist.
  virtual Status LockFile(const std::string& fname, FileLock** lock) = 0;

  // Release the lock acquired by a previous successful call to LockFile.
  // REQUIRES: lock was returned by a successful LockFile() call
  // REQUIRES: lock has not already been unlocked.
  virtual Status UnlockFile(FileLock* lock) = 0;

  // Arrange to run "(*function)(arg)" once in a background thread.
  //
  // "function" may run in an unspecified thread.  Multiple functions
  // added to the same Env may run concurrently in different threads.
  // I.e., the caller may not assume that background work items are
  // serialized.
  virtual void Schedule(
      void (*function)(void* arg),
      void* arg) = 0;

  // Start a new thread, invoking "function(arg)" within the new thread.
  // When "function(arg)" returns, the thread will be destroyed.
  virtual void StartThread(void (*function)(void* arg), void* arg) = 0;

  // *path is set to a temporary directory that can be used for testing. It may
  // or many not have just been created. The directory may or may not differ
  // between runs of the same process, but subsequent calls will return the
  // same directory.
  virtual Status GetTestDirectory(std::string* path) = 0;

  // Create and return a log file for storing informational messages.
  virtual Status NewLogger(const std::string& fname, Logger** result) = 0;

  // Returns the number of micro-seconds since some fixed point in time. Only
  // useful for computing deltas of time.
  virtual uint64_t NowMicros() = 0;

  // Sleep/delay the thread for the perscribed number of micro-seconds.
  virtual void SleepForMicroseconds(int micros) = 0;
};
```

稍微总结一下这些接口.对于这些接口创建的抽象对象，在后面也会详细解释。实现是PosixEnv.

- Default // 获得默认的环境
- NewSequentialFile // 创建顺序文件
- NewRandomAccessFile // 创建随机文件
- NewWritableFile // 创建可写文件
- FileExists // 文件是否存在

- GetChildren // 目录下面的文件
- DeleteFile // 删除文件
- CreateDir // 创建目录
- DeleteDir // 删除目录
- GetFileSize // filesize.
- RenameFile // rename.
- LockFile // 锁住文件
- UnlockFile // 解锁文件
- StartThread // 创建线程
- GetTestDirectory // 测试目录
- NewLogger // 创建logger打印到对应文件
- NowMicros // 当前us.
- SleepForMicroseconds // sleep us

环境还提供了一个EnvWrapper.这个类就是得到一个Env*对象然后重新转发出去。

2. FileLock

FileLock接口非常简单，可以说就没有接口.唯一要做的事情就是和Env里面的LockFile与UnlockFile配合。实现是PosixFileLock.

```
// Identifies a locked file.
class FileLock {
 public:
  FileLock() { }
  virtual ~FileLock();
 private:
  // No copying allowed
  FileLock(const FileLock&);
  void operator=(const FileLock&);
};
```

3. Logger

Logger接口也非常简单，就是Logv.允许打印变长参数.实现是PosixLogger.

```
// An interface for writing log messages.
class Logger {
 public:
  Logger() { }
  virtual ~Logger();

  // Write an entry to the log file with the specified format.
  virtual void Logv(const char* format, va_list ap) = 0;

 private:
  // No copying allowed
  Logger(const Logger&);
  void operator=(const Logger&);
};
```

为了编写方便提供了这么一个宏

```
// Log the specified data to *info_log if info_log is non-NULL.
extern void Log(Logger* info_log, const char* format, ...)
#   if defined(__GNUC__) || defined(__clang__)
    __attribute__((__format__ (__printf__, 2, 3)))
#   endif
    ;
```

4. RandomAccessFile

RandomAccessFile所提供的语义就是能够随机从offset读取n个字节，存放在scratch里面。 然后将result里面的内容指向scratch.随机读取并且要求能够多线程安全。实现是PosixRandomAccessFile.

```
// A file abstraction for randomly reading the contents of a file.
class RandomAccessFile {
 public:
  RandomAccessFile() { }
  virtual ~RandomAccessFile();

  // Read up to "n" bytes from the file starting at "offset".
  // "scratch[0..n-1]" may be written by this routine.  Sets "*result"
  // to the data that was read (including if fewer than "n" bytes were
  // successfully read).  May set "*result" to point at data in
  // "scratch[0..n-1]", so "scratch[0..n-1]" must be live when
  // "*result" is used.  If an error was encountered, returns a non-OK
  // status.
  //
  // Safe for concurrent use by multiple threads.
  virtual Status Read(uint64_t offset, size_t n, Slice* result,
                      char* scratch) const = 0;
};
```

5. SequentialFile

   SequentialFile提供的语义包括顺序读取以及Skip字节。这些都是外部来保证同步的。实现是PosixSequentialFile.

```
// A file abstraction for reading sequentially through a file
class SequentialFile {
 public:
  SequentialFile() { }
  virtual ~SequentialFile();

  // Read up to "n" bytes from the file.  "scratch[0..n-1]" may be
  // written by this routine.  Sets "*result" to the data that was
  // read (including if fewer than "n" bytes were successfully read).
  // May set "*result" to point at data in "scratch[0..n-1]", so
  // "scratch[0..n-1]" must be live when "*result" is used.
  // If an error was encountered, returns a non-OK status.
  //
  // REQUIRES: External synchronization
  virtual Status Read(size_t n, Slice* result, char* scratch) = 0;

  // Skip "n" bytes from the file. This is guaranteed to be no
  // slower that reading the same data, but may be faster.
  //
  // If end of file is reached, skipping will stop at the end of the
  // file, and Skip will return OK.
  //
  // REQUIRES: External synchronization
  virtual Status Skip(uint64_t n) = 0;
};
```

6. WritableFile

   WritableFile语义就是允许Append,Close,Flush,Sync.这里Flush的语义应该是将内部缓存数据完全写入，而Sync表示让磁盘进行同步。因为可能外部会调用小对象的写入，所以这里需要进行缓存。实现是PosixMmapFile.

```
// A file abstraction for sequential writing.  The implementation
// must provide buffering since callers may append small fragments
// at a time to the file.
class WritableFile {
 public:
  WritableFile() { }
  virtual ~WritableFile();

  virtual Status Append(const Slice& data) = 0;
  virtual Status Close() = 0;
  virtual Status Flush() = 0;
  virtual Status Sync() = 0;

 private:
  // No copying allowed
  WritableFile(const WritableFile&);
  void operator=(const WritableFile&);
};
```

10.1.5 filter_policy.h

通过阅读注释还是可以很容易地了解到filter_policy提供的语义的。另外还可以看到leveldb本身提供了bloomfilter的实现。#todo: 这里对于bits_per_key含义不是很了解。

#note: 所谓的bits_per_key就是希望为每个key分配多少个bits来进行检测。但是这个并不等于检测bits. 在leveldb这个bloomfilter里面的话这个数值叫做probe.

```
class FilterPolicy {
 public:
  virtual ~FilterPolicy();

  // 用来做兼容判断。如果新的filter policy发生改变的话，那么这个名字也必须变化。
  // Return the name of this policy.  Note that if the filter encoding
  // changes in an incompatible way, the name returned by this method
  // must be changed.  Otherwise, old incompatible filters may be
  // passed to methods of this type.
  virtual const char* Name() const = 0;

  // keys都是排好序的，将这些keys加入filter.相当于告知这些keys已经存在。
  // keys[0,n-1] contains a list of keys (potentially with duplicates)
  // that are ordered according to the user supplied comparator.
  // Append a filter that summarizes keys[0,n-1] to *dst.
  //
  // Warning: do not change the initial contents of *dst.  Instead,
  // append the newly constructed filter to *dst.
  virtual void CreateFilter(const Slice* keys, int n, std::string* dst)
      const = 0;

  // 判断key是否在filter里面。这个filter是在CreateFilter里面的dst包装出来的。
  // "filter" contains the data appended by a preceding call to
  // CreateFilter() on this class.  This method must return true if
  // the key was in the list of keys passed to CreateFilter().
  // This method may return true or false if the key was not on the
  // list, but it should aim to return false with a high probability.
  virtual bool KeyMayMatch(const Slice& key, const Slice& filter) const = 0;
};

// Return a new filter policy that uses a bloom filter with approximately
// the specified number of bits per key.  A good value for bits_per_key
// is 10, which yields a filter with ~ 1% false positive rate.
//
// Callers must delete the result after any database that is using the
// result has been closed.
//
// Note: if you are using a custom comparator that ignores some parts
// of the keys being compared, you must not use NewBloomFilterPolicy()
// and must provide your own FilterPolicy that also ignores the
// corresponding parts of the keys.  For example, if the comparator
// ignores trailing spaces, it would be incorrect to use a
// FilterPolicy (like NewBloomFilterPolicy) that does not ignore
// trailing spaces in keys.
extern const FilterPolicy* NewBloomFilterPolicy(int bits_per_key);
```

10.1.6 iterator.h

遍历器接口非常简单，支持前向和反向遍历。还支持seek到某一个key.支持注册cleanup函数.实现是DBIter.

```
class Iterator {
 public:
  Iterator();
  virtual ~Iterator();

  // An iterator is either positioned at a key/value pair, or
  // not valid.  This method returns true iff the iterator is valid.
  virtual bool Valid() const = 0;

  // Position at the first key in the source.  The iterator is Valid()
  // after this call iff the source is not empty.
  virtual void SeekToFirst() = 0;

  // Position at the last key in the source.  The iterator is
  // Valid() after this call iff the source is not empty.
  virtual void SeekToLast() = 0;

  // Position at the first key in the source that at or past target
  // The iterator is Valid() after this call iff the source contains
  // an entry that comes at or past target.
  virtual void Seek(const Slice& target) = 0;

  // Moves to the next entry in the source.  After this call, Valid() is
  // true iff the iterator was not positioned at the last entry in the source.
  // REQUIRES: Valid()
  virtual void Next() = 0;

  // Moves to the previous entry in the source.  After this call, Valid() is
  // true iff the iterator was not positioned at the first entry in source.
  // REQUIRES: Valid()
  virtual void Prev() = 0;

  // Return the key for the current entry.  The underlying storage for
  // the returned slice is valid only until the next modification of
  // the iterator.
  // REQUIRES: Valid()
  virtual Slice key() const = 0;

  // Return the value for the current entry.  The underlying storage for
  // the returned slice is valid only until the next modification of
  // the iterator.
  // REQUIRES: !AtEnd() && !AtStart()
  virtual Slice value() const = 0;

  // If an error has occurred, return it.  Else return an ok status.
  virtual Status status() const = 0;

  // Clients are allowed to register function/arg1/arg2 triples that
  // will be invoked when this iterator is destroyed.
  //
  // Note that unlike all of the preceding methods, this method is
  // not abstract and therefore clients should not override it.
  typedef void (*CleanupFunction)(void* arg1, void* arg2);
  void RegisterCleanup(CleanupFunction function, void* arg1, void* arg2);

 private:
  struct Cleanup {
    CleanupFunction function;
    void* arg1;
    void* arg2;
    Cleanup* next;
  };
  Cleanup cleanup_;

  // No copying allowed
  Iterator(const Iterator&);
  void operator=(const Iterator&);
};
```

这个里面部分实现在table/iterator.cc里面有.都非常简单.创建好cleanup对象然后组织称为链表,在析构函数时候调用.

```
Iterator::Iterator() {
  cleanup_.function = NULL;
  cleanup_.next = NULL;
}

Iterator::~Iterator() {
  if (cleanup_.function != NULL) {
    (*cleanup_.function)(cleanup_.arg1, cleanup_.arg2);
    for (Cleanup* c = cleanup_.next; c != NULL; ) {
      (*c->function)(c->arg1, c->arg2);
      Cleanup* next = c->next;
      delete c;
      c = next;
    }
  }
}

void Iterator::RegisterCleanup(CleanupFunction func, void* arg1, void* arg2) {
  assert(func != NULL);
  Cleanup* c;
  if (cleanup_.function == NULL) {
    c = &cleanup_;
  } else {
    c = new Cleanup;
    c->next = cleanup_.next;
    cleanup_.next = c;
  }
  c->function = func;
  c->arg1 = arg1;
  c->arg2 = arg2;
}
```

10.1.7 options.h

对于options来说存在3种

- Options // 控制DB行为.
- ReadOptions // 控制读取行为
- WriteOptions // 控制写入行为

1. Options

   Options控制DB行为，在DB::Open时候就传入.我们需要针对这些字段仔细看看.

```
// DB contents are stored in a set of blocks, each of which holds a
// sequence of key,value pairs.  Each block may be compressed before
// being stored in a file.  The following enum describes which
// compression method (if any) is used to compress a block.
enum CompressionType {
  // NOTE: do not change the values of existing entries, as these are
  // part of the persistent format on disk.
  kNoCompression     = 0x0,
  kSnappyCompression = 0x1
};

// Options to control the behavior of a database (passed to DB::Open)
struct Options {
  // -------------------
  // Parameters that affect behavior

  // Comparator used to define the order of keys in the table.
  // Default: a comparator that uses lexicographic byte-wise ordering
  //
  // REQUIRES: The client must ensure that the comparator supplied
  // here has the same name and orders keys *exactly* the same as the
  // comparator provided to previous open calls on the same DB.
  const Comparator* comparator; // 如何进行slice compare

  // If true, the database will be created if it is missing.
  // Default: false
  bool create_if_missing; // database不存在是否需要创建.

  // If true, an error is raised if the database already exists.
  // Default: false
  bool error_if_exists; // 如果数据库存在是否error

  // If true, the implementation will do aggressive checking of the
  // data it is processing and will stop early if it detects any
  // errors.  This may have unforeseen ramifications: for example, a
  // corruption of one DB entry may cause a large number of entries to
  // become unreadable or for the entire DB to become unopenable.
  // Default: false
  bool paranoid_checks; // 会更多地检查数据正确性.这个在日志回放的时候有效.
  // MaybeIgnoreError里面使用.

  // Use the specified object to interact with the environment,
  // e.g. to read/write files, schedule background work, etc.
  // Default: Env::Default()
  Env* env; // 操作环境.

  // Any internal progress/error information generated by the db will
  // be written to info_log if it is non-NULL, or to a file stored
  // in the same directory as the DB contents if info_log is NULL.
  // Default: NULL
  Logger* info_log; // Logger.

  // -------------------
  // Parameters that affect performance

  // Amount of data to build up in memory (backed by an unsorted log
  // on disk) before converting to a sorted on-disk file.
  //
  // Larger values increase performance, especially during bulk loads.
  // Up to two write buffers may be held in memory at the same time,
  // so you may wish to adjust this parameter to control memory usage.
  // Also, a larger write buffer will result in a longer recovery time
  // the next time the database is opened.
  //
  // Default: 4MB
  size_t write_buffer_size; // memtable超过这些内存大小的话就会写table.

  // Number of open files that can be used by the DB.  You may need to
  // increase this if your database has a large working set (budget
  // one open file per 2MB of working set).
  //
  // Default: 1000
  int max_open_files; // 最大文件句柄数.

  // Control over blocks (user data is stored in a set of blocks, and
  // a block is the unit of reading from disk).

  // If non-NULL, use the specified cache for blocks.
  // If NULL, leveldb will automatically create and use an 8MB internal cache.
  // Default: NULL
  Cache* block_cache; // 内部cahce.

  // Approximate size of user data packed per block.  Note that the
  // block size specified here corresponds to uncompressed data.  The
  // actual size of the unit read from disk may be smaller if
  // compression is enabled.  This parameter can be changed dynamically.
  //
  // Default: 4K
  size_t block_size; // SSTable里面的Block大小.具体可以看BlockBuilder.

  // Number of keys between restart points for delta encoding of keys.
  // This parameter can be changed dynamically.  Most clients should
  // leave this parameter alone.
  //
  // Default: 16
  int block_restart_interval; // block里面的restart间隔.具体可以看BlockBuilder.

  // Compress blocks using the specified compression algorithm.  This
  // parameter can be changed dynamically.
  //
  // Default: kSnappyCompression, which gives lightweight but fast
  // compression.
  //
  // Typical speeds of kSnappyCompression on an Intel(R) Core(TM)2 2.4GHz:
  //    ~200-500MB/s compression
  //    ~400-800MB/s decompression
  // Note that these speeds are significantly faster than most
  // persistent storage speeds, and therefore it is typically never
  // worth switching to kNoCompression.  Even if the input data is
  // incompressible, the kSnappyCompression implementation will
  // efficiently detect that and will switch to uncompressed mode.
  CompressionType compression; // 压缩类型.不压缩和Snappy压缩.

  // Create an Options object with default values for all fields.
  Options();
};
```

2. ReadOptions

和Options一样里面也都是控制字段.

```
// Options that control read operations
struct ReadOptions {
  // If true, all data read from underlying storage will be
  // verified against corresponding checksums.
  // Default: false
  bool verify_checksums; // 读取时候进行checksum校验

  // Should the data read for this iteration be cached in memory?
  // Callers may wish to set this field to false for bulk scans.
  // Default: true
  bool fill_cache; // 是否需要从cache里面查找.如果是bulk scans的话那么设置false.

  // If "snapshot" is non-NULL, read as of the supplied snapshot
  // (which must belong to the DB that is being read and which must
  // not have been released).  If "snapshot" is NULL, use an impliicit
  // snapshot of the state at the beginning of this read operation.
  // Default: NULL
  const Snapshot* snapshot; // 如果存在Snapshot的话那么就在snapshot上面读取.

  ReadOptions()
      : verify_checksums(false),
        fill_cache(true),
        snapshot(NULL) {
  }
};
```

3. WriteOptions

和Options一样里面也都是控制字段.

```
// Options that control write operations
struct WriteOptions {
  // If true, the write will be flushed from the operating system
  // buffer cache (by calling WritableFile::Sync()) before the write
  // is considered complete.  If this flag is true, writes will be
  // slower.
  //
  // If this flag is false, and the machine crashes, some recent
  // writes may be lost.  Note that if it is just the process that
  // crashes (i.e., the machine does not reboot), no writes will be
  // lost even if sync==false.
  //
  // In other words, a DB write with sync==false has similar
  // crash semantics as the "write()" system call.  A DB write
  // with sync==true has similar crash semantics to a "write()"
  // system call followed by "fsync()".
  //
  // Default: false
  bool sync; // 是否每次写都需要fsync.

  WriteOptions()
      : sync(false) {
  }
};
```

10.1.8 slice.h

leveldb里面的Slice对象是用来作为key使用的。内部实现非常简单，仅仅是存储二进制的指针和大小。我们可能需要稍微注意一下slice对象的compare方法.首先按照最小长度比较，如果相等然后按照比较长度。

```
inline int Slice::compare(const Slice& b) const {
  const int min_len = (size_ < b.size_) ? size_ : b.size_;
  int r = memcmp(data_, b.data_, min_len);
  if (r == 0) {
    if (size_ < b.size_) r = -1;
    else if (size_ > b.size_) r = +1;
  }
  return r;
}
```

10.1.9 status.h

status就是一个非常简单的封装.内部持有一个char* status_;从注释里面可以看出如何安排的。 实现在util/status.cc里面，没有必要仔细阅读。不过我到觉得直接返回一个int就算了。 没有必要开辟char[].不过如果这个部分没有性能问题也就无所谓了。

```
class Status {
 public:
  // Create a success status.
  Status() : state_(NULL) { }
  ~Status() { delete[] state_; }

  // Copy the specified status.
  Status(const Status& s);
  void operator=(const Status& s);

  // Return a success status.
  static Status OK() { return Status(); }

  // Return error status of an appropriate type.
  static Status NotFound(const Slice& msg, const Slice& msg2 = Slice()) {
    return Status(kNotFound, msg, msg2);
  }
  static Status Corruption(const Slice& msg, const Slice& msg2 = Slice()) {
    return Status(kCorruption, msg, msg2);
  }
  static Status NotSupported(const Slice& msg, const Slice& msg2 = Slice()) {
    return Status(kNotSupported, msg, msg2);
  }
  static Status InvalidArgument(const Slice& msg, const Slice& msg2 = Slice()) {
    return Status(kInvalidArgument, msg, msg2);
  }
  static Status IOError(const Slice& msg, const Slice& msg2 = Slice()) {
    return Status(kIOError, msg, msg2);
  }

  // Returns true iff the status indicates success.
  bool ok() const { return (state_ == NULL); }

  // Returns true iff the status indicates a NotFound error.
  bool IsNotFound() const { return code() == kNotFound; }

  // Return a string representation of this status suitable for printing.
  // Returns the string "OK" for success.
  std::string ToString() const;

 private:
  // OK status has a NULL state_.  Otherwise, state_ is a new[] array
  // of the following form:
  //    state_[0..3] == length of message
  //    state_[4]    == code
  //    state_[5..]  == message
  const char* state_;

  enum Code {
    kOk = 0,
    kNotFound = 1,
    kCorruption = 2,
    kNotSupported = 3,
    kInvalidArgument = 4,
    kIOError = 5
  };

  Code code() const {
    return (state_ == NULL) ? kOk : static_cast<Code>(state_[4]);
  }

  Status(Code code, const Slice& msg, const Slice& msg2);
  static const char* CopyState(const char* s);
};

inline Status::Status(const Status& s) {
  state_ = (s.state_ == NULL) ? NULL : CopyState(s.state_);
}
inline void Status::operator=(const Status& s) {
  // The following condition catches both aliasing (when this == &s),
  // and the common case where both s and *this are ok.
  if (state_ != s.state_) {
    delete[] state_;
    state_ = (s.state_ == NULL) ? NULL : CopyState(s.state_);
  }
}
```

10.1.10 table_builder.h

TableBuilder用来帮助构建Table.而Table本身只是用来进行查询遍历等操作.用户在完成之后需要Finish或者是Abandon. 具体实现会在后面分析.

```
class TableBuilder {
 public:
  // Create a builder that will store the contents of the table it is
  // building in *file.  Does not close the file.  It is up to the
  // caller to close the file after calling Finish().
  TableBuilder(const Options& options, WritableFile* file); // 传入options和可写文件(追加写模式)

  // REQUIRES: Either Finish() or Abandon() has been called.
  ~TableBuilder();

  // Change the options used by this builder.  Note: only some of the
  // option fields can be changed after construction.  If a field is
  // not allowed to change dynamically and its value in the structure
  // passed to the constructor is different from its value in the
  // structure passed to this method, this method will return an error
  // without changing any fields.
  Status ChangeOptions(const Options& options); // 一旦构建好之后只允许修改部分字段.

  // Add key,value to the table being constructed.
  // REQUIRES: key is after any previously added key according to comparator.
  // REQUIRES: Finish(), Abandon() have not been called
  void Add(const Slice& key, const Slice& value); // 添加kv

  // Advanced operation: flush any buffered key/value pairs to file.
  // Can be used to ensure that two adjacent entries never live in
  // the same data block.  Most clients should not need to use this method.
  // REQUIRES: Finish(), Abandon() have not been called
  void Flush(); // 将buffered的kv刷新到文件

  // Return non-ok iff some error has been detected.
  Status status() const;
  // Finish building the table.  Stops using the file passed to the
  // constructor after this function returns.
  // REQUIRES: Finish(), Abandon() have not been called
  Status Finish(); // 构建完成.

  // Indicate that the contents of this builder should be abandoned.  Stops
  // using the file passed to the constructor after this function returns.
  // If the caller is not going to call Finish(), it must call Abandon()
  // before destroying this builder.
  // REQUIRES: Finish(), Abandon() have not been called
  void Abandon(); // 如果放弃构建的话

  // Number of calls to Add() so far.
  uint64_t NumEntries() const; // 添加了多少次

  // Size of the file generated so far.  If invoked after a successful
  // Finish() call, returns the size of the final generated file.
  uint64_t FileSize() const; // 当前已经写入多少文件了.

 private:
  bool ok() const { return status().ok(); }
  void WriteBlock(BlockBuilder* block, BlockHandle* handle);

  struct Rep;
  Rep* rep_;

  // No copying allowed
  TableBuilder(const TableBuilder&);
  void operator=(const TableBuilder&);
};
```

下面是一些比较重要的接口.

- ChangeOptions // 修改选项
- Add // 添加kv.k必须是有序
- Flush // 刷新
- Finish // 结束
- Abandon // 放弃

10.1.11 table.h

Table就是持久化并且不可变的sortedtable.下面来看看接口.具体实现会在后面分析.

```
// A Table is a sorted map from strings to strings.  Tables are
// immutable and persistent.  A Table may be safely accessed from
// multiple threads without external synchronization.
class Table {
 public:
  // Attempt to open the table that is stored in bytes [0..file_size)
  // of "file", and read the metadata entries necessary to allow
  // retrieving data from the table.
  //
  // If successful, returns ok and sets "*table" to the newly opened
  // table.  The client should delete "*table" when no longer needed.
  // If there was an error while initializing the table, sets "*table"
  // to NULL and returns a non-ok status.  Does not take ownership of
  // "*source", but the client must ensure that "source" remains live
  // for the duration of the returned table's lifetime.
  //
  // *file must remain live while this Table is in use.
  static Status Open(const Options& options,
                     RandomAccessFile* file,
                     uint64_t file_size,
                     Table** table);

  ~Table();

  // Returns a new iterator over the table contents.
  // The result of NewIterator() is initially invalid (caller must
  // call one of the Seek methods on the iterator before using it).
  Iterator* NewIterator(const ReadOptions&) const;

  // Given a key, return an approximate byte offset in the file where
  // the data for that key begins (or would begin if the key were
  // present in the file).  The returned value is in terms of file
  // bytes, and so includes effects like compression of the underlying data.
  // E.g., the approximate offset of the last key in the table will
  // be close to the file length.
  uint64_t ApproximateOffsetOf(const Slice& key) const;

 private:
  struct Rep;
  Rep* rep_;

  explicit Table(Rep* rep) { rep_ = rep; }
  static Iterator* BlockReader(void*, const ReadOptions&, const Slice&);

  // No copying allowed
  Table(const Table&);
  void operator=(const Table&);
};
```

主要提供的接口如下

- Open. // 这个接口可以看出访问的是随机文件.
- NewIterator // 创建一个迭代器.
- ApproximateOffsetOf // 可以通过key查找到大致位置然后后续可以发起读操作.

10.1.12 write_batch.h

WriteBatch用来持有批量写入的内容.注意底层实现有Handler是需要具体实现的。关于这个具体实现部分会在后面分析.

```
class WriteBatch {
 public:
  WriteBatch();
  ~WriteBatch();

  // Store the mapping "key->value" in the database.
  void Put(const Slice& key, const Slice& value);

  // If the database contains a mapping for "key", erase it.  Else do nothing.
  void Delete(const Slice& key);

  // Clear all updates buffered in this batch.
  void Clear();

  // Support for iterating over the contents of a batch.
  class Handler {
   public:
    virtual ~Handler();
    virtual void Put(const Slice& key, const Slice& value) = 0;
    virtual void Delete(const Slice& key) = 0;
  };
  Status Iterate(Handler* handler) const;
  // 阅读后面会发现，这个意思是遍历所有的WriteBatch里面的item
  // 然后操作handler.handler可能是一个memtable.这样可以将里面所有的内容
  // 全部存放到memtable里面去.

 private:
  friend class WriteBatchInternal; // 底层实现是这个.

  std::string rep_;  // See comment in write_batch.cc for the format of rep_ // 将所有的操作请求做成二进制存放在rep_

  // Intentionally copyable
};
```

10.2 Implementation

实现部分我按照功能划分了几个部分.

- DB
- Posix
- Cache
- Batch
- Log
- Table
- Util

10.3 DB

10.3.1 FileName

db/filename.cc 这里面都是关于文件名称操作的方法.文件包括

- db/CURRENT // 当前文件
- db/LOCK // DB锁文件
- db/LOG // info log.日志文件.
- db/LOG.old // info log.日志文件.
- db/MANIFEST-[0-9]+ // 描述文件
- db/[0-9]+.log // db日志文件
- db/[0-9]+.sst // dbtable文件
- db/[0-9]+.dbtmp // db临时文件

这里[0-9]+都表示一个sequence number.这里还有另外一个不是操作FileName的函数.SetCurrentFile

```
Status SetCurrentFile(Env* env, const std::string& dbname,
                      uint64_t descriptor_number) {
  // Remove leading "dbname/" and add newline to manifest file name
  std::string manifest = DescriptorFileName(dbname, descriptor_number);
  Slice contents = manifest;
  assert(contents.starts_with(dbname + "/"));
  contents.remove_prefix(dbname.size() + 1);
  std::string tmp = TempFileName(dbname, descriptor_number);
  Status s = WriteStringToFile(env, contents.ToString() + "\n", tmp);
  if (s.ok()) {
    s = env->RenameFile(tmp, CurrentFileName(dbname));
  }
  if (!s.ok()) {
    env->DeleteFile(tmp);
  }
  return s;
}
```

就是将MANIFEST-%(descriptor_number)llu+"\n"写入到.dbtmp下面去然后rename成为CURRENT文件.

10.3.2 Config

db/dbformat.h config下面是一些静态常数.这里可以仔细看看.

```
// Grouping of constants.  We may want to make some of these
// parameters set via options.
namespace config {
// leveldb最大level多少.
static const int kNumLevels = 7;

// Level-0 compaction is started when we hit this many files.
// level0文件超过多少个触发compaction.
static const int kL0_CompactionTrigger = 4;

// 下面两个可以在MakeRoomForWrite里面看到.
// Soft limit on number of level-0 files.  We slow down writes at this point.
// 如果level0文件超过这么多的话那么可能会放缓memtable写为level0的速度.比如delay 1s啥的.
static const int kL0_SlowdownWritesTrigger = 8;

// Maximum number of level-0 files.  We stop writes at this point.
// 如果level0文件超过这么多的话那不会memtable写为level0.
static const int kL0_StopWritesTrigger = 12;

// 对于memtable进行compaction的话选择的最高level.
// 这个可以在PickLevelForMemtableOutput里面可以看到.
// Maximum level to which a new compacted memtable is pushed if it
// does not create overlap.  We try to push to level 2 to avoid the
// relatively expensive level 0=>1 compactions and to avoid some
// expensive manifest file operations.  We do not push all the way to
// the largest level since that can generate a lot of wasted disk
// space if the same key space is being repeatedly overwritten.
static const int kMaxMemCompactLevel = 2;
}  // namespace config
```

10.3.3 DBImpl

db/db_impl.cc DBImpl这个结构体挺大的。我们先过一个这个结构然后仔细看看每个字段是什么

```
class DBImpl : public DB {
 private:
  // Constant after construction
  Env* const env_; // 环境.
  const InternalKeyComparator internal_comparator_; // 内部比较器.
  const Options options_;  // options_.comparator == &internal_comparator_ // 选项.
  bool owns_info_log_; // 是否自己分配的log.
  bool owns_cache_; // 是否自己分配的cache.
  const std::string dbname_; // db名称.

  // table_cache_ provides its own synchronization
  TableCache* table_cache_; // TableCache.

  // Lock over the persistent DB state.  Non-NULL iff successfully acquired.
  FileLock* db_lock_; // db FileLock. // 对于外部进程标记DB对象互斥锁.

  // State below is protected by mutex_
  port::Mutex mutex_; // 整个DB对象互斥锁.
  port::AtomicPointer shutting_down_; // 标记这个DB对象正在退出。后台线程或看到之后不应该进行任何操作。
  port::CondVar bg_cv_;          // Signalled when background work finishes
  MemTable* mem_; // 正在操作的memtable.
  MemTable* imm_;                // Memtable being compacted.正在被compacted的memtable
  port::AtomicPointer has_imm_;  // So bg thread can detect non-NULL imm_
  WritableFile* logfile_;
  uint64_t logfile_number_;
  log::Writer* log_;
  LoggerId* logger_;            // NULL, or the id of the current logging thread
  port::CondVar logger_cv_;     // For threads waiting to log.和mutex_关联.
  SnapshotList snapshots_; // 当前所有的snapshot

  // Set of table files to protect from deletion because they are
  // part of ongoing compactions.
  std::set<uint64_t> pending_outputs_; // 正在输出或者是进行compaction的file.
  // 保存这个信息的话这样在delete文件的放置不被删除.

  // Has a background compaction been scheduled or is running?
  bool bg_compaction_scheduled_; // 后台线程是否在运行.还是可以退出.

  // Information for a manual compaction
  struct ManualCompaction {
    int level;
    bool done;
    const InternalKey* begin;   // NULL means beginning of key range
    const InternalKey* end;     // NULL means end of key range
    InternalKey tmp_storage;    // Used to keep track of compaction progress
  };
  ManualCompaction* manual_compaction_;

  VersionSet* versions_; // 版本集合.

  // Have we encountered a background error in paranoid mode?
  Status bg_error_; // 后台线程运行状态.

  // Per level compaction stats.  stats_[level] stores the stats for
  // compactions that produced data for the specified "level".
  struct CompactionStats { // 压缩状态信息.
    int64_t micros;
    int64_t bytes_read;
    int64_t bytes_written;

    CompactionStats() : micros(0), bytes_read(0), bytes_written(0) { }

    void Add(const CompactionStats& c) {
      this->micros += c.micros;
      this->bytes_read += c.bytes_read;
      this->bytes_written += c.bytes_written;
    }
  };
  CompactionStats stats_[config::kNumLevels]; // 各个级别的压缩状态信息.
};
```

1. Open

   在Interface部分的db.h里面可以知道构造这个DB对象是通过DB::Open来构造的。

```
Status DB::Open(const Options& options, const std::string& dbname,
                DB** dbptr) {
  *dbptr = NULL;

  DBImpl* impl = new DBImpl(options, dbname); // 创建DBImpl实例.
  impl->mutex_.Lock(); // 似乎没有太大必要.这里应该没有竞争.
  VersionEdit edit;
  // 回复自上次依赖的edit所有内容.然后在后面直接log and apply这个edit对象.
  Status s = impl->Recover(&edit); // Handles create_if_missing, error_if_exists
  // 下面每个步骤大致上都很清楚
  // 1.创建新的log文件
  // 2.回放原来log信息
  // 3.删除不必要的文件
  // 4.进行compaction.
  if (s.ok()) {
    uint64_t new_log_number = impl->versions_->NewFileNumber();
    WritableFile* lfile;
    s = options.env->NewWritableFile(LogFileName(dbname, new_log_number),
                                     &lfile);
    if (s.ok()) {
      edit.SetLogNumber(new_log_number);
      impl->logfile_ = lfile;
      impl->logfile_number_ = new_log_number;
      impl->log_ = new log::Writer(lfile);
      s = impl->versions_->LogAndApply(&edit, &impl->mutex_);
    }
    if (s.ok()) {
      impl->DeleteObsoleteFiles();
      impl->MaybeScheduleCompaction();
    }
  }
  // 将产生的DB对象返回.
  impl->mutex_.Unlock();
  if (s.ok()) {
    *dbptr = impl;
  } else {
    delete impl;
  }
  return s;
}
```

2. DBImpl

先看看构造函数

```
DBImpl::DBImpl(const Options& options, const std::string& dbname)
    : env_(options.env),
      internal_comparator_(options.comparator),
      // 注意这里的顺序.必须先调用SanitizeOptions.
      options_(SanitizeOptions(dbname, &internal_comparator_, options)),
      // 判断是否自己持有info log和block cache.
      owns_info_log_(options_.info_log != options.info_log),
      owns_cache_(options_.block_cache != options.block_cache),
      dbname_(dbname),
      db_lock_(NULL),
      shutting_down_(NULL),
      bg_cv_(&mutex_),
      // 创建新的MemTable对象.
      mem_(new MemTable(internal_comparator_)),
      imm_(NULL),
      // 这里log都是指db的log而不是程序log.
      logfile_(NULL),
      logfile_number_(0),
      log_(NULL),
      logger_(NULL),
      logger_cv_(&mutex_),
      bg_compaction_scheduled_(false),
      manual_compaction_(NULL) {
  mem_->Ref();
  has_imm_.Release_Store(NULL);

  // Reserve ten files or so for other uses and give the rest to TableCache.
  const int table_cache_size = options.max_open_files - 10;
  // 对于文件描述符限制可以通过TableCache来完成.不错:).
  table_cache_ = new TableCache(dbname_, &options_, table_cache_size);
  // 创建VersionSet.
  versions_ = new VersionSet(dbname_, &options_, table_cache_,
                             &internal_comparator_);
}
```

然后看看析构函数

```
DBImpl::~DBImpl() {
  // Wait for background work to finish
  mutex_.Lock();
  // 设置shuting down标记.后台线程等待标记退出.
  // 等待bg_compaction_scheduled_置位，这样bg线程就退出了。
  shutting_down_.Release_Store(this);  // Any non-NULL value is ok
  while (bg_compaction_scheduled_) {
    bg_cv_.Wait();
  }
  mutex_.Unlock();

  // 释放db锁文件.
  if (db_lock_ != NULL) {
    env_->UnlockFile(db_lock_);
  }

  // 删除VersionSet.
  delete versions_;
  // 释放引用计数
  if (mem_ != NULL) mem_->Unref();
  if (imm_ != NULL) imm_->Unref();
  // 删除可能产生的log,logfile以及table_cache.
  delete log_;
  delete logfile_;
  delete table_cache_;

  // 释放程序log以及cache.
  if (owns_info_log_) {
    delete options_.info_log;
  }
  if (owns_cache_) {
    delete options_.block_cache;
  }
}
```

3. NewDB

但从名字上我们不知道里面做了哪些事情，所以还是来看看代码. 通过代码阅读会发现，完成的事情大致就是建立一个Manifest文件，然后将这个版本的Manifest文件 的文件名作为内容写入CURRENT文件。Manifest文件里面包含的就是VersionEdit信息。VersionEdit 可以认为就是这个数据库的元信息。

```
Status DBImpl::NewDB() {
  VersionEdit new_db;
  // 创建新的VersionEdit.设置好comparator的名字以及必要字段
  // 关于VersionEdit的信息会在后面仔细分析.
  new_db.SetComparatorName(user_comparator()->Name());
  new_db.SetLogNumber(0); // 从log number从0开始分配.
  new_db.SetNextFile(2);  // 下一个number从2开始分配
  new_db.SetLastSequence(0);

  // 创建新的Manifest文件.然后作为writable文件打开.
  // 1分配给manifest文件.
  const std::string manifest = DescriptorFileName(dbname_, 1);
  WritableFile* file;
  Status s = env_->NewWritableFile(manifest, &file);
  if (!s.ok()) {
    return s;
  }
  // 文件作为log格式打开.将VersionEdit序列化之后作为log写入.
  {
    log::Writer log(file);
    std::string record;
    new_db.EncodeTo(&record);
    s = log.AddRecord(record);
    if (s.ok()) {
      s = file->Close();
    }
  }
  delete file;
  // 然后将CURRENT里面的文件设置为版本1的manifest文件.
  if (s.ok()) {
    // Make "CURRENT" file that points to the new manifest file.
    s = SetCurrentFile(env_, dbname_, 1);
  } else {
    env_->DeleteFile(manifest);
  }
  return s;
}
```

4. Recover

恢复状态并且将恢复对于version日志操作.

```
Status DBImpl::Recover(VersionEdit* edit) {
  mutex_.AssertHeld();

  // Ignore error from CreateDir since the creation of the DB is
  // committed only when the descriptor is created, and this directory
  // may already exist from a previous failed creation attempt.
  env_->CreateDir(dbname_);
  assert(db_lock_ == NULL);
  Status s = env_->LockFile(LockFileName(dbname_), &db_lock_);
  if (!s.ok()) {
    return s;
  }

  if (!env_->FileExists(CurrentFileName(dbname_))) {
    if (options_.create_if_missing) {
      s = NewDB(); // 创建DB.
      if (!s.ok()) {
        return s;
      }
    } else {
      return Status::InvalidArgument(
          dbname_, "does not exist (create_if_missing is false)");
    }
  } else {
    if (options_.error_if_exists) {
      return Status::InvalidArgument(
          dbname_, "exists (error_if_exists is true)");
    }
  }

  s = versions_->Recover(); // 对于VersionSet首先进行恢复.
  // 恢复之后,根据里面的内容然后读取里面对应的version edit内容.
  // 不过从versionset的Recover方法来看的话里面所有的内容已经全部恢复了.
  // 后面edit的内容是因为在恢复log里面的内容造成的.然后将这个部分造成的edit
  // 之后调用LogAndApply.
  if (s.ok()) {
    SequenceNumber max_sequence(0);

    // Recover from all newer log files than the ones named in the
    // descriptor (new log files may have been added by the previous
    // incarnation without registering them in the descriptor).
    //
    // Note that PrevLogNumber() is no longer used, but we pay
    // attention to it in case we are recovering a database
    // produced by an older version of leveldb.
    // 从注释这里看以看出prev_log已经不适用了.
    const uint64_t min_log = versions_->LogNumber();
    const uint64_t prev_log = versions_->PrevLogNumber();
    std::vector<std::string> filenames;
    s = env_->GetChildren(dbname_, &filenames);
    if (!s.ok()) {
      return s;
    }
    uint64_t number;
    FileType type;
    std::vector<uint64_t> logs;
    // 分析logs文件然后判断哪些log文件是后来追加我们需要重放的.
    for (size_t i = 0; i < filenames.size(); i++) {
      if (ParseFileName(filenames[i], &number, &type)
          && type == kLogFile
          && ((number >= min_log) || (number == prev_log))) {
        logs.push_back(number);
      }
    }

    // Recover in the order in which the logs were generated
    // 按照顺序进行重放.
    std::sort(logs.begin(), logs.end());
    for (size_t i = 0; i < logs.size(); i++) {
      // 恢复某个log文件.并且将里面的操作修改填写到edit
      // 同时知道当前最大的sequence是多少.
      s = RecoverLogFile(logs[i], edit, &max_sequence);

      // The previous incarnation may not have written any MANIFEST
      // records after allocating this log number.  So we manually
      // update the file number allocation counter in VersionSet.
      // 标记file number已经被使用了.
      versions_->MarkFileNumberUsed(logs[i]);
    }

    if (s.ok()) {
      // 更新一下sequence number.
      if (versions_->LastSequence() < max_sequence) {
        versions_->SetLastSequence(max_sequence);
      }
    }
  }

  return s;
}
```

5. RecoverLogFile

从单个log恢复写入的内容.并且根据log里面内容了解到对于version修改以及max_sequence.

```
Status DBImpl::RecoverLogFile(uint64_t log_number,
                              VersionEdit* edit,
                              SequenceNumber* max_sequence) {
  struct LogReporter : public log::Reader::Reporter {
    Env* env;
    Logger* info_log;
    const char* fname;
    Status* status;  // NULL if options_.paranoid_checks==false
    virtual void Corruption(size_t bytes, const Status& s) {
      Log(info_log, "%s%s: dropping %d bytes; %s",
          (this->status == NULL ? "(ignoring error) " : ""),
          fname, static_cast<int>(bytes), s.ToString().c_str());
      if (this->status != NULL && this->status->ok()) *this->status = s;
    }
  };

  mutex_.AssertHeld();

  // 打开日志文件
  // Open the log file
  std::string fname = LogFileName(dbname_, log_number);
  SequentialFile* file;
  Status status = env_->NewSequentialFile(fname, &file);
  if (!status.ok()) {
    MaybeIgnoreError(&status);
    return status;
  }

  // 构造reporter对象.
  // Create the log reader.
  LogReporter reporter;
  reporter.env = env_;
  reporter.info_log = options_.info_log;
  reporter.fname = fname.c_str();
  reporter.status = (options_.paranoid_checks ? &status : NULL);
  // We intentially make log::Reader do checksumming even if
  // paranoid_checks==false so that corruptions cause entire commits
  // to be skipped instead of propagating bad information (like overly
  // large sequence numbers).
  // 读取日志文件.做校验并且从0偏移开始读取.
  log::Reader reader(file, &reporter, true/*checksum*/,
                     0/*initial_offset*/);
  Log(options_.info_log, "Recovering log #%llu",
      (unsigned long long) log_number);

  // Read all the records and add to a memtable
  std::string scratch;
  Slice record;
  WriteBatch batch;
  MemTable* mem = NULL;
  // 不断地读取
  while (reader.ReadRecord(&record, &scratch) &&
         status.ok()) {
    if (record.size() < 12) {
      reporter.Corruption(
          record.size(), Status::Corruption("log record too small"));
      continue;
    }
    // log里面都是write batch的内容
    // 所以可以直接设置.
    WriteBatchInternal::SetContents(&batch, record);

    if (mem == NULL) {
      // 创建mem table.
      mem = new MemTable(internal_comparator_);
      mem->Ref();
    }
    status = WriteBatchInternal::InsertInto(&batch, mem);
    // 对于日志恢复的话我们也可以选择性地进行status判断检查.
    MaybeIgnoreError(&status);
    if (!status.ok()) {
      break;
    }
    // 更新sequence.
    const SequenceNumber last_seq =
        WriteBatchInternal::Sequence(&batch) +
        WriteBatchInternal::Count(&batch) - 1;
    if (last_seq > *max_sequence) {
      *max_sequence = last_seq;
    }

    // 如果占用内存大于这个大小的话那么就会写level0.
    if (mem->ApproximateMemoryUsage() > options_.write_buffer_size) {
      status = WriteLevel0Table(mem, edit, NULL);
      if (!status.ok()) {
        // Reflect errors immediately so that conditions like full
        // file-systems cause the DB::Open() to fail.
        break;
      }
      mem->Unref();
      mem = NULL;
    }
  }

  // 剩下的内存文件也会做table.
  if (status.ok() && mem != NULL) {
    status = WriteLevel0Table(mem, edit, NULL);
    // Reflect errors immediately so that conditions like full
    // file-systems cause the DB::Open() to fail.
  }

  if (mem != NULL) mem->Unref();
  delete file;
  return status;
}
```

6. MaybeIgnoreError

   这个是在日志恢复部分是否进行错误恢复.

```
void DBImpl::MaybeIgnoreError(Status* s) const {
  if (s->ok() || options_.paranoid_checks) {
    // No change needed
  } else {
    Log(options_.info_log, "Ignoring error %s", s->ToString().c_str());
    *s = Status::OK();
  }
}
```

7. WriteLevel0Table

   将memtable写到level0.不过现在就实现来看的话不一定是写到level0.对于产生或者是删除的文件等对于version的操作都会反映到edit里面

```
Status DBImpl::WriteLevel0Table(MemTable* mem, VersionEdit* edit,
                                Version* base) {
  mutex_.AssertHeld();
  // 会针对这个操作进行计时.
  const uint64_t start_micros = env_->NowMicros();
  FileMetaData meta;
  // 产生新的file number.
  meta.number = versions_->NewFileNumber();
  //
  pending_outputs_.insert(meta.number);
  Iterator* iter = mem->NewIterator();
  Log(options_.info_log, "Level-0 table #%llu: started",
      (unsigned long long) meta.number);

  Status s;
  {
    mutex_.Unlock();
    // 注意写磁盘的时候没有必要加锁.
    s = BuildTable(dbname_, env_, options_, table_cache_, iter, &meta);
    mutex_.Lock();
  }

  Log(options_.info_log, "Level-0 table #%llu: %lld bytes %s",
      (unsigned long long) meta.number,
      (unsigned long long) meta.file_size,
      s.ToString().c_str());
  delete iter;
  pending_outputs_.erase(meta.number);
  // Note that if file_size is zero, the file has been deleted and
  // should not be added to the manifest.
  int level = 0;
  if (s.ok() && meta.file_size > 0) {
    const Slice min_user_key = meta.smallest.user_key();
    const Slice max_user_key = meta.largest.user_key();
    if (base != NULL) {
      // 如果存在base version的话
      // 那么会根据base version以及range来选择新的level进行序列化.
      level = base->PickLevelForMemTableOutput(min_user_key, max_user_key);
    }
    edit->AddFile(level, meta.number, meta.file_size,
                  meta.smallest, meta.largest);
  }

  CompactionStats stats;
  stats.micros = env_->NowMicros() - start_micros;
  stats.bytes_written = meta.file_size;
  // 修改这个level的compaction数据.
  stats_[level].Add(stats);
  return s;
}
```

8. Put

```
Status DBImpl::Put(const WriteOptions& o, const Slice& key, const Slice& val) {
  return DB::Put(o, key, val);
}
Status DB::Put(const WriteOptions& opt, const Slice& key, const Slice& value) {
  WriteBatch batch;
  batch.Put(key, value); // 将内容写到WriteBatch里面.然后通过
  return Write(opt, &batch); // Write写入到db内部.
}
```

9. Delete

```
Status DBImpl::Delete(const WriteOptions& options, const Slice& key) {
  return DB::Delete(options, key);
}
Status DB::Delete(const WriteOptions& opt, const Slice& key) {
  WriteBatch batch;
  batch.Delete(key); // 将删除内容写到WriteBatch里面，然后通过
  return Write(opt, &batch); // Write写入到db内部.
}
```

10. LoggingResponsibility

对于Log信息的打印的话确保每次只有一个实例在操作.这里logger_cv和mutex_关联起来的.可以看到 在调用AcquireLoggingResponsibility的地方之前都会加锁

```
// There is at most one thread that is the current logger.  This call
// waits until preceding logger(s) have finished and becomes the
// current logger.
void DBImpl::AcquireLoggingResponsibility(LoggerId* self) {
  while (logger_ != NULL) {
    logger_cv_.Wait();
  }
  logger_ = self;
}
```

而ReleaseLoggingResponsibility是释放logger的所有权.注意这里我们不会去主动操作解锁而是外部解锁。

```
void DBImpl::ReleaseLoggingResponsibility(LoggerId* self) {
  assert(logger_ == self);
  logger_ = NULL;
  logger_cv_.SignalAll();
}
```

11. MakeRoomForWrite

仅仅从函数名称上面开不出到底是开辟什么空间.看看实现吧.代码里面可以分析出这个部分是存在compaction的. 我们这里稍微总结一下逻辑

- 如果允许delay的话并且level0文件数目超过slowdown的阈值的话，那么就会先尝试delay 1s.下次不会进行delay
- 如果不是force的话并且memtable空间允许的话那么直接返回
- 剩下的逻辑就是force出一个memtable了.那么这个时候必须进行compaction to level0.
- 先检查是否正在被memtable compaction.如果正在的话那么等待
- 然后查看level0文件数目是否过多.如果过多的话那么也等待
- 最后创建新的memtable以及logfile.将原来的memtable保存起来准备后台compaction
- 发起compaction.并且force=false.

按照性能角度出发的话，这种逻辑应该非常make sense.

```
// REQUIRES: mutex_ is held
// REQUIRES: this thread is the current logger
Status DBImpl::MakeRoomForWrite(bool force) {
  mutex_.AssertHeld();
  assert(logger_ != NULL);
  bool allow_delay = !force; // 如果设置force的话
  // 如果空间不够的话那么就会发起compaction操作
  // 然后等待compaction操作完成看空间是否足够.
  Status s;
  while (true) {
    // 如果后台存在错误的话那么直接返回.
    if (!bg_error_.ok()) {
      // yield previous error
      s = bg_error_;
      break;
    } else if (
        allow_delay &&
        versions_->NumLevelFiles(0) >= config::kL0_SlowdownWritesTrigger) {
      // We are getting close to hitting a hard limit on the number of
      // L0 files.  Rather than delaying a single write by several
      // seconds when we hit the hard limit, start delaying each
      // individual write by 1ms to reduce latency variance.  Also,
      // this delay hands over some CPU to the compaction thread in
      // case it is sharing the same core as the writer.
      mutex_.Unlock();
      env_->SleepForMicroseconds(1000); // 延迟1s来看level0文件个数.
      allow_delay = false;  // Do not delay a single write more than once
      mutex_.Lock();
    } else if (!force &&
               (mem_->ApproximateMemoryUsage() <= options_.write_buffer_size)) {
      // 如果memtable允许写入的话那么没有任何问题.
      // There is room in current memtable
      break;
    } else if (imm_ != NULL) {
      // We have filled up the current memtable, but the previous
      // one is still being compacted, so we wait.
      // imm_应该是表示正在进行compact的memtable.
      // 这里我担心condition是边缘触发而不是水平触发的话那么signal就会丢失.
      bg_cv_.Wait();
    } else if (versions_->NumLevelFiles(0) >= config::kL0_StopWritesTrigger) {
      // There are too many level-0 files.
      // level0文件个数过多的话那么等待compaction的完成.
      // note:这个部分非常重要，这样可以限制写入速度保证系统balance.
      Log(options_.info_log, "waiting...\n");
      bg_cv_.Wait();
    } else {
      // Attempt to switch to a new memtable and trigger compaction of old
      // 试图创建一个新的memtable用来写.创建memtable的话同时也创建一个log文件.
      assert(versions_->PrevLogNumber() == 0);
      uint64_t new_log_number = versions_->NewFileNumber();
      WritableFile* lfile = NULL;
      s = env_->NewWritableFile(LogFileName(dbname_, new_log_number), &lfile);
      if (!s.ok()) {
        break;
      }
      // 销毁原来的log文件b并且创建新的memtable.
      // 新memtable存放在imm里面.后台应该是将imm进行compaction或者是level disk files之间进行compaction.
      delete log_;
      delete logfile_;
      logfile_ = lfile;
      logfile_number_ = new_log_number;
      log_ = new log::Writer(lfile);
      imm_ = mem_; // 将原来的mem_存放起来.
      has_imm_.Release_Store(imm_); // imm存在标记.
      mem_ = new MemTable(internal_comparator_); // 新创建memtable.
      mem_->Ref();
      force = false;   // Do not force another compaction if have room
      MaybeScheduleCompaction(); // 尝试进行compaction.这个函数我们后面分析.
    }
  }
  return s;
}
```

12. MaybeScheduleCompaction

发起compaction调度.

```
void DBImpl::MaybeScheduleCompaction() {
  mutex_.AssertHeld();
  if (bg_compaction_scheduled_) { // 如果已经被调度的话.那么没有必要产能重新发起.
    // Already scheduled
  } else if (shutting_down_.Acquire_Load()) { // 如果已经关闭的话那么忽略.
    // DB is being deleted; no more background compactions
  } else if (imm_ == NULL && // 如果memtable没有
             manual_compaction_ == NULL && // 如果没有人工compaction
             !versions_->NeedsCompaction()) { // 如果leveldb本身也没有必要认为compaction.
    // No work to be done
  } else {
    // compaction之前标记
    bg_compaction_scheduled_ = true;
    // 并且这个时候发起compaction操作.这个后面会仔细分析BGWork这个过程.
    // BGWork是pthread_create封装的接口，里面仅仅是调用了BackgroundCall这个函数.
    env_->Schedule(&DBImpl::BGWork, this);
  }
}
```

13. BackgroundCall

```
void DBImpl::BackgroundCall() {
  MutexLock l(&mutex_);
  assert(bg_compaction_scheduled_);
  if (!shutting_down_.Acquire_Load()) { // 如果没有关闭的话.
    BackgroundCompaction(); // 那么会调用这个函数发起compaction.
  }
  bg_compaction_scheduled_ = false;

  // Previous compaction may have produced too many files in a level,
  // so reschedule another compaction if needed.
  MaybeScheduleCompaction(); // 这里可能会重新发起compaction.
  bg_cv_.SignalAll();
}
```

14. BackgroundCompaction

这个部分是真正进行compaction的部分.我们仔细分析其中的行为.

```
void DBImpl::BackgroundCompaction() {
  mutex_.AssertHeld();

  // 从代码实现上来看的话，如果真实地进行compaction的话
  // 对于version造成的修改都会记录为edit,然后调用VersionSet::LogAndApply保存起来.
  if (imm_ != NULL) {
    CompactMemTable(); // 如果imm!=NULL的话那么首先对imm进行compaction.
    return;
  }

  // 注意这里imm_==NULL.
  Compaction* c;
  bool is_manual = (manual_compaction_ != NULL);
  InternalKey manual_end;
  if (is_manual) {
    // 如果是manual compaction的话
    // 那么compaction里面需要提供level以及range.
    // 但是我猜想仅仅是将这个compaction提供一个包装信息出来
    // 具体操作延迟到后面进行.
    ManualCompaction* m = manual_compaction_;
    c = versions_->CompactRange(m->level, m->begin, m->end);
    m->done = (c == NULL);
    if (c != NULL) {
      manual_end = c->input(0, c->num_input_files(0) - 1)->largest;
    }
    Log(options_.info_log,
        "Manual compaction at level-%d from %s .. %s; will stop at %s\n",
        m->level,
        (m->begin ? m->begin->DebugString().c_str() : "(begin)"),
        (m->end ? m->end->DebugString().c_str() : "(end)"),
        (m->done ? "(end)" : manual_end.DebugString().c_str()));
  } else { // 如果不是manual compaction那么挑选一个出来.
    c = versions_->PickCompaction();
  }
  Status status;
  if (c == NULL) {
    // Nothing to do
  } else if (!is_manual && c->IsTrivialMove()) {
    // Move file to next level
    assert(c->num_input_files(0) == 1);
    FileMetaData* f = c->input(0, 0);
    c->edit()->DeleteFile(c->level(), f->number);
    c->edit()->AddFile(c->level() + 1, f->number, f->file_size,
                       f->smallest, f->largest);
    status = versions_->LogAndApply(c->edit(), &mutex_);
    VersionSet::LevelSummaryStorage tmp;
    Log(options_.info_log, "Moved #%lld to level-%d %lld bytes %s: %s\n",
        static_cast<unsigned long long>(f->number),
        c->level() + 1,
        static_cast<unsigned long long>(f->file_size),
        status.ToString().c_str(),
        versions_->LevelSummary(&tmp));
  } else { // 将compaction信息包装称为state进行操作.
    CompactionState* compact = new CompactionState(c);
    status = DoCompactionWork(compact);
    CleanupCompaction(compact);
  }
  delete c;

  if (status.ok()) {
    // Done
  } else if (shutting_down_.Acquire_Load()) {
    // Ignore compaction errors found during shutting down
  } else {
    Log(options_.info_log,
        "Compaction error: %s", status.ToString().c_str());
    if (options_.paranoid_checks && bg_error_.ok()) {
      bg_error_ = status;
    }
  }

  if (is_manual) { // 如果是manual_compaction的话那么我们
    // 有方法可以知道这次manual compaction实际操作范围有多少.
    // 实际范围就是range所对应文件的largest key.:).这个之前可以看到.
    ManualCompaction* m = manual_compaction_;
    if (!m->done) {
      // We only compacted part of the requested range.  Update *m
      // to the range that is left to be compacted.
      m->tmp_storage = manual_end;
      m->begin = &m->tmp_storage;
    }
    manual_compaction_ = NULL;
  }
}
```

15. CompactMemTable

对memtable进行compaction.注意这里针对的对象是imm_.mem对象是write操作的对象.

```
Status DBImpl::CompactMemTable() {
  mutex_.AssertHeld();
  assert(imm_ != NULL);

  // Save the contents of the memtable as a new Table
  VersionEdit edit;
  Version* base = versions_->current();
  base->Ref();
  Status s = WriteLevel0Table(imm_, &edit, base);
  base->Unref();

  if (s.ok() && shutting_down_.Acquire_Load()) {
    s = Status::IOError("Deleting DB during memtable compaction");
  }

  // Replace immutable memtable with the generated Table
  if (s.ok()) {
    // 一旦这个memtable进行compaction完成之后
    // 那么就可以认为这个log file number不需要了.
    // 将这个信息记录在version edit里面.
    edit.SetPrevLogNumber(0);
    edit.SetLogNumber(logfile_number_);  // Earlier logs no longer needed
    s = versions_->LogAndApply(&edit, &mutex_);
  }

  // 然后释放imm这个内存并且删除多余文件.
  if (s.ok()) {
    // Commit to the new state
    imm_->Unref();
    imm_ = NULL;
    has_imm_.Release_Store(NULL);
    DeleteObsoleteFiles();
  }

  return s;
}
```

16. CompactRange

    这个语义应该是针对某个range来进行compaction的.

    - 首先查看和这些range存在overlap的最大level是多少
    - 首先对memtable进行compaction(不管是否存在overlap)(TEST_CompactMemTable)
    - 然后遍历这些level,分别对每层进行compact range.(TEST_CompactRange)

```
void DBImpl::CompactRange(const Slice* begin, const Slice* end) {
  int max_level_with_files = 1;
  {
    MutexLock l(&mutex_);
    Version* base = versions_->current();
    for (int level = 1; level < config::kNumLevels; level++) {
      if (base->OverlapInLevel(level, begin, end)) {
        max_level_with_files = level;
      }
    }
  }
  TEST_CompactMemTable(); // TODO(sanjay): Skip if memtable does not overlap
  for (int level = 0; level < max_level_with_files; level++) {
    TEST_CompactRange(level, begin, end);
  }
}
```

17. TEST_CompactMemTable

```
Status DBImpl::TEST_CompactMemTable() {
  MutexLock l(&mutex_);
  LoggerId self;
  AcquireLoggingResponsibility(&self);
  // 这里应该是要求imm==NULL也就是说没有memtable在进行compaction的.
  // 那么这样的行为非常明显就是希望mem重新开辟
  // 将原来的mem进行compaction
  Status s = MakeRoomForWrite(true /* force compaction */);
  ReleaseLoggingResponsibility(&self);
  if (s.ok()) {
    // Wait until the compaction completes
    // 并且等待memtable compaction结束.
    while (imm_ != NULL && bg_error_.ok()) {
      bg_cv_.Wait();
    }
    if (imm_ != NULL) {
      s = bg_error_;
    }
  }
  return s;
}
```

18. TEST_CompactRange

    针对某个Level来进行range compaction.

```
void DBImpl::TEST_CompactRange(int level, const Slice* begin,const Slice* end) {
  assert(level >= 0);
  assert(level + 1 < config::kNumLevels);

  InternalKey begin_storage, end_storage;

  ManualCompaction manual;
  manual.level = level; // level
  manual.done = false; // 是否已经完成.
  if (begin == NULL) { // 选择manual compaction range.
    manual.begin = NULL;
  } else {
    begin_storage = InternalKey(*begin, kMaxSequenceNumber, kValueTypeForSeek);
    manual.begin = &begin_storage;
  }
  if (end == NULL) {
    manual.end = NULL;
  } else {
    end_storage = InternalKey(*end, 0, static_cast<ValueType>(0));
    manual.end = &end_storage;
  }

  MutexLock l(&mutex_);
  while (!manual.done) { // 如果没有完成的话
    // 等待上一次完成
    while (manual_compaction_ != NULL) {
      bg_cv_.Wait();
    }
    // 等待本次完成
    // 还是调用maybe schedule compaction
    // 按照代码来看的话走的分支主要是关注VersionSet::CompactRange这个部分.
    manual_compaction_ = &manual;
    MaybeScheduleCompaction();
    while (manual_compaction_ == &manual) {
      bg_cv_.Wait();
    }
  }
}
```

19. DoCompactionWork

完成的工作是实质性地Compaction.通过读取提交的CompactionState来完成.过程比较长可以逐段逐段地阅读代码.

```
Status DBImpl::DoCompactionWork(CompactionState* compact) {
  const uint64_t start_micros = env_->NowMicros();
  int64_t imm_micros = 0;  // Micros spent doing imm_ compactions

  Log(options_.info_log,  "Compacting %d@%d + %d@%d files",
      compact->compaction->num_input_files(0),
      compact->compaction->level(),
      compact->compaction->num_input_files(1),
      compact->compaction->level() + 1);

  assert(versions_->NumLevelFiles(compact->compaction->level()) > 0);
  assert(compact->builder == NULL);
  assert(compact->outfile == NULL);
  // 首先根据当前所有提交的snapshot知道当前最小的sequence number是多少.
  if (snapshots_.empty()) {
    compact->smallest_snapshot = versions_->LastSequence();
  } else {
    compact->smallest_snapshot = snapshots_.oldest()->number_;
  }

  // Release mutex while we're actually doing the compaction work
  mutex_.Unlock();

  // 针对这个compaction创建出iterator.我们在后面看看这个iterator是如何建立的.
  Iterator* input = versions_->MakeInputIterator(compact->compaction);
  input->SeekToFirst();
  Status status;
  ParsedInternalKey ikey;
  std::string current_user_key;
  bool has_current_user_key = false;
  SequenceNumber last_sequence_for_key = kMaxSequenceNumber;
  for (; input->Valid() && !shutting_down_.Acquire_Load(); ) {
    // Prioritize immutable compaction work
    // 做compaction之前先将immutable table compaction.
    if (has_imm_.NoBarrier_Load() != NULL) {
      const uint64_t imm_start = env_->NowMicros();
      mutex_.Lock();
      if (imm_ != NULL) {
        CompactMemTable();
        bg_cv_.SignalAll();  // Wakeup MakeRoomForWrite() if necessary
      }
      mutex_.Unlock();
      imm_micros += (env_->NowMicros() - imm_start);
    }

    // 判断在这个key是否应该独立产生一个文件.
    Slice key = input->key();
    if (compact->compaction->ShouldStopBefore(key) &&
        compact->builder != NULL) {
      status = FinishCompactionOutputFile(compact, input);
      if (!status.ok()) {
        break;
      }
    }

    // Handle key/value, add to state, etc.
    // drop表示这个key是否应该直接丢弃.
    // 这个地方始终感觉有点问题.为什么计算last seuqnce number呢?.
    // 不过大致思想是了解的.
    bool drop = false;
    if (!ParseInternalKey(key, &ikey)) {
      // Do not hide error keys
      current_user_key.clear();
      has_current_user_key = false;
      last_sequence_for_key = kMaxSequenceNumber;
    } else {
      if (!has_current_user_key ||
          user_comparator()->Compare(ikey.user_key,
                                     Slice(current_user_key)) != 0) {
        // First occurrence of this user key
        current_user_key.assign(ikey.user_key.data(), ikey.user_key.size());
        has_current_user_key = true;
        last_sequence_for_key = kMaxSequenceNumber;
      }
      // 这段逻辑判断是否需要丢弃.
      if (last_sequence_for_key <= compact->smallest_snapshot) {
        // Hidden by an newer entry for same user key
        drop = true;    // (A)
      } else if (ikey.type == kTypeDeletion && // 如果是删除并且这个key < snapshot.
                 ikey.sequence <= compact->smallest_snapshot &&
                 compact->compaction->IsBaseLevelForKey(ikey.user_key)) {
        // For this user key:
        // (1) there is no data in higher levels
        // (2) data in lower levels will have larger sequence numbers
        // (3) data in layers that are being compacted here and have
        //     smaller sequence numbers will be dropped in the next
        //     few iterations of this loop (by rule (A) above).
        // Therefore this deletion marker is obsolete and can be dropped.
        drop = true;
      }

      last_sequence_for_key = ikey.sequence;
    }
#if 0
    Log(options_.info_log,
        "  Compact: %s, seq %d, type: %d %d, drop: %d, is_base: %d, "
        "%d smallest_snapshot: %d",
        ikey.user_key.ToString().c_str(),
        (int)ikey.sequence, ikey.type, kTypeValue, drop,
        compact->compaction->IsBaseLevelForKey(ikey.user_key),
        (int)last_sequence_for_key, (int)compact->smallest_snapshot);
#endif

    // 不管前面细节如何到这个步骤表明这个kv没有被drop掉.
    if (!drop) {
      // Open output file if necessary
      if (compact->builder == NULL) { // 根据compaction状态产生新输出文件.
        status = OpenCompactionOutputFile(compact);
        if (!status.ok()) {
          break;
        }
      }
      // ==0的时候记录最小key.
      if (compact->builder->NumEntries() == 0) {
        compact->current_output()->smallest.DecodeFrom(key);
      }
      // 之后每次更新最大key
      compact->current_output()->largest.DecodeFrom(key);
      // 记录这个kv.
      compact->builder->Add(key, input->value());

      // Close output file if it is big enough
      // 如果文件超过大小的话那么需要切换文件.
      if (compact->builder->FileSize() >=
          compact->compaction->MaxOutputFileSize()) {
        status = FinishCompactionOutputFile(compact, input);
        if (!status.ok()) {
          break;
        }
      }
    }

    input->Next();
  }

  if (status.ok() && shutting_down_.Acquire_Load()) {
    status = Status::IOError("Deleting DB during compaction");
  }
  // 最后可能需要关闭compaction的文件.
  if (status.ok() && compact->builder != NULL) {
    status = FinishCompactionOutputFile(compact, input);
  }
  if (status.ok()) {
    status = input->status();
  }
  delete input; // 删除这个iterator.
  input = NULL;

  CompactionStats stats;
  // compaction部分的时间还需要出去内存memtable compaction是时间.
  stats.micros = env_->NowMicros() - start_micros - imm_micros;
  // 统计这次操作读取的大小.
  for (int which = 0; which < 2; which++) {
    for (int i = 0; i < compact->compaction->num_input_files(which); i++) {
      stats.bytes_read += compact->compaction->input(which, i)->file_size;
    }
  }
  // 统计这次操作写磁盘大小.
  for (size_t i = 0; i < compact->outputs.size(); i++) {
    stats.bytes_written += compact->outputs[i].file_size;
  }

  mutex_.Lock();
  // 将这个状态合并上来.
  stats_[compact->compaction->level() + 1].Add(stats);

  if (status.ok()) {
    status = InstallCompactionResults(compact);
  }
  VersionSet::LevelSummaryStorage tmp;
  Log(options_.info_log,
      "compacted to: %s", versions_->LevelSummary(&tmp));
  return status;
}
```

20. CleanupCompaction

完成compaction之后的工作.

```
void DBImpl::CleanupCompaction(CompactionState* compact) {
  mutex_.AssertHeld();
  if (compact->builder != NULL) { // 如果这个builder没有销毁的话那么认为
    // 中途是shutdown了.那么我们这里直接abandon掉.
    // May happen if we get a shutdown call in the middle of compaction
    compact->builder->Abandon();
    delete compact->builder;
  } else {
    assert(compact->outfile == NULL);
  }
  delete compact->outfile;
  // compaction过程中的话新输出的output文件应该都加入了pengding_outputs
  // 防止这个文件部分被删除.完成compaction之后的话可以移除了
  // 考虑这里外部应该有标记通知这些文件还是需要的.
  // 所以针对这些文件可能已经创建了另外一个version对象存放在version_set内部.
  // 但是啥时候释放version这个对象呢???.
  for (size_t i = 0; i < compact->outputs.size(); i++) {
    const CompactionState::Output& out = compact->outputs[i];
    pending_outputs_.erase(out.number);
  }
  delete compact;
}
```

21. InstallCompactionResults

这个函数调用过程是在DoCompactionWork里面.大致工作就是将这次compaction工作内容作为日志保存起来.

```
Status DBImpl::InstallCompactionResults(CompactionState* compact) {
  mutex_.AssertHeld();
  Log(options_.info_log,  "Compacted %d@%d + %d@%d files => %lld bytes",
      compact->compaction->num_input_files(0),
      compact->compaction->level(),
      compact->compaction->num_input_files(1),
      compact->compaction->level() + 1,
      static_cast<long long>(compact->total_bytes));

  // Add compaction outputs
  compact->compaction->AddInputDeletions(compact->compaction->edit()); // 将compaction这次操作删除和增加文件加入version_
  const int level = compact->compaction->level();
  for (size_t i = 0; i < compact->outputs.size(); i++) {
    const CompactionState::Output& out = compact->outputs[i];
    // 将本地操作放在version_edit里面.
    compact->compaction->edit()->AddFile(
        level + 1,
        out.number, out.file_size, out.smallest, out.largest);
    pending_outputs_.erase(out.number); // 从pending output里面删除.
  }
  compact->outputs.clear();

  Status s = versions_->LogAndApply(compact->compaction->edit(), &mutex_); // 通知version set本次修改内容.
  // 并且将这次compaction的内容作为version写入log里面去.
  if (s.ok()) {
    compact->compaction->ReleaseInputs();
    DeleteObsoleteFiles();
  } else {
    // Discard any files we may have created during this failed compaction
    for (size_t i = 0; i < compact->outputs.size(); i++) {
      env_->DeleteFile(TableFileName(dbname_, compact->outputs[i].number));
    }
  }
  return s;
}
```

22. OpenCompactionOutputFile

    打开用于compaction输出的文件.倒不是非常麻烦.

```
Status DBImpl::OpenCompactionOutputFile(CompactionState* compact) {
  assert(compact != NULL);
  assert(compact->builder == NULL);
  uint64_t file_number;
  {
    mutex_.Lock();
    file_number = versions_->NewFileNumber();
    pending_outputs_.insert(file_number); // 放在pending output里面.
    CompactionState::Output out;
    out.number = file_number;
    out.smallest.Clear();
    out.largest.Clear();
    compact->outputs.push_back(out);
    mutex_.Unlock();
  }

  // Make the output file
  // 创建输出文件以及对应的table builder.
  std::string fname = TableFileName(dbname_, file_number);
  Status s = env_->NewWritableFile(fname, &compact->outfile);
  if (s.ok()) {
    compact->builder = new TableBuilder(options_, compact->outfile);
  }
  return s;
}
```

23. FinishCompactionOutputFile

    对于输出了文件之后我们需要finish的操作.倒不是非常麻烦.

```
Status DBImpl::FinishCompactionOutputFile(CompactionState* compact,
                                          Iterator* input) {
  assert(compact != NULL);
  assert(compact->outfile != NULL);
  assert(compact->builder != NULL);

  const uint64_t output_number = compact->current_output()->number;
  assert(output_number != 0);

  // Check for iterator errors
  Status s = input->status();
  const uint64_t current_entries = compact->builder->NumEntries();
  if (s.ok()) {
    s = compact->builder->Finish(); // 通知对应的builder对象finish.
  } else {
    compact->builder->Abandon();
  }
  const uint64_t current_bytes = compact->builder->FileSize();
  compact->current_output()->file_size = current_bytes;
  compact->total_bytes += current_bytes;
  delete compact->builder; // 释放原来builder.
  compact->builder = NULL;

  // Finish and check for file errors
  if (s.ok()) {
    s = compact->outfile->Sync();
  }
  if (s.ok()) {
    s = compact->outfile->Close();
  }
  delete compact->outfile; // 释放原来的file对象.
  compact->outfile = NULL;

  // 验证这个文件是否OK.
  if (s.ok() && current_entries > 0) {
    // Verify that the table is usable
    Iterator* iter = table_cache_->NewIterator(ReadOptions(),
                                               output_number,
                                               current_bytes);
    s = iter->status();
    delete iter;
    if (s.ok()) {
      Log(options_.info_log,
          "Generated table #%llu: %lld keys, %lld bytes",
          (unsigned long long) output_number,
          (unsigned long long) current_entries,
          (unsigned long long) current_bytes);
    }
  }
  return s;
}
```

24. Write

这个是Put/Delete底层的最终实现.仔细看看吧

```
Status DBImpl::Write(const WriteOptions& options, WriteBatch* updates) {
  Status status;
  MutexLock l(&mutex_);
  LoggerId self;
  AcquireLoggingResponsibility(&self);
  status = MakeRoomForWrite(false);  // May temporarily release lock and wait
  uint64_t last_sequence = versions_->LastSequence(); // 分配sequence.
  if (status.ok()) {
    WriteBatchInternal::SetSequence(updates, last_sequence + 1); // WriteBatch的sequence从+1开始.
    last_sequence += WriteBatchInternal::Count(updates); // 更新上次sequence.

    // Add to log and apply to memtable.  We can release the lock during
    // this phase since the "logger_" flag protects against concurrent
    // loggers and concurrent writes into mem_.
    {
      assert(logger_ == &self);
      mutex_.Unlock();
      // 这个部分是肯定需要写入log的.
      // 注意这里释放锁是没有问题，因为这个地方logger可以作为锁存在.
      // 这里解开mutex似乎可以使得Get进行.
      // 我觉得第一遍看leveldb代码的时候可以撇开并发这个部分.
      status = log_->AddRecord(WriteBatchInternal::Contents(updates)); // 将updates的内容作为contents得到写入log.
      if (status.ok() && options.sync) {
        status = logfile_->Sync();
      }
      if (status.ok()) {
        // 然后将updates写入memtable.
        status = WriteBatchInternal::InsertInto(updates, mem_);
      }
      mutex_.Lock();
      assert(logger_ == &self);
    }

    // 写入成功之后的话那么将这个sequence重新写回.
    versions_->SetLastSequence(last_sequence); // 设置新的sequence.
  }
  ReleaseLoggingResponsibility(&self);
  return status;
}
```

25. SanitizeOptions

这个函数应该就是规范一下options这个结构.

```
// Fix user-supplied options to be reasonable
template <class T,class V>
static void ClipToRange(T* ptr, V minvalue, V maxvalue) {
  if (static_cast<V>(*ptr) > maxvalue) *ptr = maxvalue;
  if (static_cast<V>(*ptr) < minvalue) *ptr = minvalue;
}
Options SanitizeOptions(const std::string& dbname,
                        const InternalKeyComparator* icmp,
                        const Options& src) {
  Options result = src;
  result.comparator = icmp;
  // 规范取值范围.
  ClipToRange(&result.max_open_files,           20,     50000); // 20 , 50000
  ClipToRange(&result.write_buffer_size,        64<<10, 1<<30); // 64K , 1G
  ClipToRange(&result.block_size,               1<<10,  4<<20); // 1K, 4M.
  // 如果用户没有填写logger对象的话
  // 那么产生一个logger对象.
  if (result.info_log == NULL) {
    // Open a log file in the same directory as the db
    src.env->CreateDir(dbname);  // In case it does not exist
    src.env->RenameFile(InfoLogFileName(dbname), OldInfoLogFileName(dbname));
    Status s = src.env->NewLogger(InfoLogFileName(dbname), &result.info_log);
    if (!s.ok()) {
      // No place suitable for logging
      result.info_log = NULL;
    }
  }
  // 对于block cache也是.开辟的block cache大小8MB.
  if (result.block_cache == NULL) {
    result.block_cache = NewLRUCache(8 << 20);
  }
  return result;
}
```

26. GetProperty

关于db的属性信息.

```
bool DBImpl::GetProperty(const Slice& property, std::string* value) {
  value->clear();

  MutexLock l(&mutex_);
  Slice in = property;
  // 属性的key必须是以leveldb.开头的.
  Slice prefix("leveldb.");
  if (!in.starts_with(prefix)) return false;
  in.remove_prefix(prefix.size());

  if (in.starts_with("num-files-at-level")) { // 每个level的文件数目.
    in.remove_prefix(strlen("num-files-at-level"));
    uint64_t level;
    bool ok = ConsumeDecimalNumber(&in, &level) && in.empty();
    if (!ok || level >= config::kNumLevels) {
      return false;
    } else {
      char buf[100];
      snprintf(buf, sizeof(buf), "%d",
               versions_->NumLevelFiles(static_cast<int>(level)));
      *value = buf;
      return true;
    }
  } else if (in == "stats") { // 整个db的状态.
    char buf[200];
    snprintf(buf, sizeof(buf),
             "                               Compactions\n"
             "Level  Files Size(MB) Time(sec) Read(MB) Write(MB)\n"
             "--------------------------------------------------\n"
             );
    value->append(buf);
    for (int level = 0; level < config::kNumLevels; level++) {
      int files = versions_->NumLevelFiles(level);
      if (stats_[level].micros > 0 || files > 0) {
        snprintf(
            buf, sizeof(buf),
            "%3d %8d %8.0f %9.0f %8.0f %9.0f\n",
            level,
            files,
            versions_->NumLevelBytes(level) / 1048576.0,
            stats_[level].micros / 1e6, // 在这个level上面操作的时间.
            stats_[level].bytes_read / 1048576.0, // 在这个level上面读取的字节数目.
            stats_[level].bytes_written / 1048576.0); // 在这个level上面写入的字节数目.
        value->append(buf);
      }
    }
    return true;
  } else if (in == "sstables") { // 当前sstables的状态.
    // 这个是Version打印出的字符串.
    *value = versions_->current()->DebugString();
    return true;
  }

  return false;
}
```

27. GetApproximateSizes

得到某个Range占用的大小.底层依赖于VersionSet

```
void DBImpl::GetApproximateSizes(
    const Range* range, int n,
    uint64_t* sizes) {
  // TODO(opt): better implementation
  Version* v;
  {
    MutexLock l(&mutex_); // 注意这里是全局加锁的.
    versions_->current()->Ref(); // 对当前version加上引用计数.
    v = versions_->current();
  }

  for (int i = 0; i < n; i++) {
    // Convert user_key into a corresponding internal key.
    // 这里sequence number是否正确呢?
    // 不过大致上占用的空间差不多就是这么大.
    InternalKey k1(range[i].start, kMaxSequenceNumber, kValueTypeForSeek);
    InternalKey k2(range[i].limit, kMaxSequenceNumber, kValueTypeForSeek);
    // 具体实现可以查看VersionSet方法.根据某个key判断这个key在文件中的偏移.
    // 可能会有一部分偏差吧但是大致正确.
    uint64_t start = versions_->ApproximateOffsetOf(v, k1);
    uint64_t limit = versions_->ApproximateOffsetOf(v, k2);
    sizes[i] = (limit >= start ? limit - start : 0);
  }

  {
    MutexLock l(&mutex_);
    v->Unref();
  }
}
```

28. GetSnapshot

```
const Snapshot* DBImpl::GetSnapshot() {
  MutexLock l(&mutex_);
  // 返回最新的sequence number创建一个Snapshot实例.
  // 并且添加到snapshot list里面.
  return snapshots_.New(versions_->LastSequence());
}
```

29. ReleaseSnapshot

```
void DBImpl::ReleaseSnapshot(const Snapshot* s) {
  MutexLock l(&mutex_);
  // 从snapshot list里面删除.
  snapshots_.Delete(reinterpret_cast<const SnapshotImpl*>(s));
}
```

30. DeleteObsoleteFiles

根据当前所有version所持有的file来决定哪些文件是不再需要的.version里面会持有file meta信息.

```
void DBImpl::DeleteObsoleteFiles() {
  // Make a set of all of the live files
  std::set<uint64_t> live = pending_outputs_;
  versions_->AddLiveFiles(&live); // 持有pending outputs里面正在输出的文件
  // 并且将verisons里面所有version持有的文件得到.

  std::vector<std::string> filenames;
  // 遍历当前目录下面的文件.
  // 注意这里文件数目不会过多.
  // 因为通常来说每个level文件都会有一个下限大小数目
  // 而每个level的bytes有一个限制.对于最高层的level的话那么文件已经非常大了:)
  env_->GetChildren(dbname_, &filenames); // Ignoring errors on purpose
  uint64_t number;
  FileType type;
  for (size_t i = 0; i < filenames.size(); i++) {
    if (ParseFileName(filenames[i], &number, &type)) {
      bool keep = true;
      switch (type) {
        case kLogFile:
          // 对于log文件的number是按照顺序分配的.
          keep = ((number >= versions_->LogNumber()) || //
                  (number == versions_->PrevLogNumber())); // prev log number已经没有必要了.
          break;
        case kDescriptorFile:
          // Keep my manifest file, and any newer incarnations'
          // (in case there is a race that allows other incarnations)
          keep = (number >= versions_->ManifestFileNumber());
          break;
        case kTableFile:
          keep = (live.find(number) != live.end()); // 判断这个文件是否需要删除.
          break;
        case kTempFile:
          // Any temp files that are currently being written to must
          // be recorded in pending_outputs_, which is inserted into "live"
          keep = (live.find(number) != live.end()); // 判断文件是否需要删除.
          break;
        case kCurrentFile:
        case kDBLockFile:
        case kInfoLogFile:
          keep = true; // 对于其他文件的话直接keep住就好了.
          break;
      }

      if (!keep) {
        if (type == kTableFile) { // 如果是table文件的话还需要从cache里面去掉.
          table_cache_->Evict(number);
        }
        Log(options_.info_log, "Delete type=%d #%lld\n",
            int(type),
            static_cast<unsigned long long>(number));
        env_->DeleteFile(dbname_ + "/" + filenames[i]); // 然后删除文件.
      }
    }
  }
}
```

31. Get

Get的过程非常简单.因为实际Get的过程已经托管为Version::Get这个方法了.这里面我们只需要 构造出正确的seuqnce number,和user key拼接成为internal key即可.

```
Status DBImpl::Get(const ReadOptions& options,
                   const Slice& key,
                   std::string* value) {
  Status s;
  MutexLock l(&mutex_);
  SequenceNumber snapshot;
  if (options.snapshot != NULL) { // 判断当前读取是否需要snapshot.
    snapshot = reinterpret_cast<const SnapshotImpl*>(options.snapshot)->number_;
  } else {
    snapshot = versions_->LastSequence();
  }

  MemTable* mem = mem_;
  MemTable* imm = imm_;
  Version* current = versions_->current();
  mem->Ref();
  if (imm != NULL) imm->Ref();
  current->Ref();

  // 这个查询是否会造成compaction触发.
  bool have_stat_update = false;
  Version::GetStats stats;

  // Unlock while reading from files and memtables
  {
    mutex_.Unlock();
    // 以这个key去进行查询.
    // 1.首先查询memtable 2.然后查询正在dump的memtable 3.查询磁盘.
    // First look in the memtable, then in the immutable memtable (if any).
    LookupKey lkey(key, snapshot);
    if (mem->Get(lkey, value, &s)) {
      // Done
    } else if (imm != NULL && imm->Get(lkey, value, &s)) {
      // Done
    } else {
      s = current->Get(options, lkey, value, &stats);
      have_stat_update = true; // 查询磁盘的话会进行标记.
    }
    mutex_.Lock();
  }

  // 得到的内容会反馈到当前的version里面然后尝试触发compaction.
  if (have_stat_update && current->UpdateStats(stats)) {
    MaybeScheduleCompaction();
  }
  mem->Unref();
  if (imm != NULL) imm->Unref();
  current->Unref();
  return s;
}
```

32. NewInternalIterator

开辟内部的迭代器.但是功能上来说基本上以及完成了db::iterator所需要完成的工作.但是需要 注意这里面merge iterator接口是针对internal key的，所以外部的话还是需要保证user key 并且配合sequence number的行为.另外还需要考虑存在deletion情况.

```
struct IterState {
  port::Mutex* mu;
  Version* version;
  MemTable* mem;
  MemTable* imm;
};

static void CleanupIteratorState(void* arg1, void* arg2) {
  IterState* state = reinterpret_cast<IterState*>(arg1);
  state->mu->Lock();
  state->mem->Unref();
  if (state->imm != NULL) state->imm->Unref();
  state->version->Unref();
  state->mu->Unlock();
  delete state;
}

Iterator* DBImpl::NewInternalIterator(const ReadOptions& options,
                                      SequenceNumber* latest_snapshot) {
  IterState* cleanup = new IterState;
  mutex_.Lock();
  *latest_snapshot = versions_->LastSequence();

  // 将可能存在的iterator放在一个list里面做成一个merge list内容.
  // Collect together all needed child iterators
  std::vector<Iterator*> list;
  // 首先memtable下面需要遍历.
  list.push_back(mem_->NewIterator());
  mem_->Ref();
  // 其次imm memtable需要遍历
  if (imm_ != NULL) {
    list.push_back(imm_->NewIterator());
    imm_->Ref();
  }
  // 对于version来说可能存在很多文件需要遍历.
  versions_->current()->AddIterators(options, &list);
  // 将这些内容构造称为一个merge iterator.
  // 注意这里的内容都加了引用计数.
  Iterator* internal_iter =
      NewMergingIterator(&internal_comparator_, &list[0], list.size());
  versions_->current()->Ref();

  // 然后将这些释放内容存放在internal iter销毁回调里面.
  cleanup->mu = &mutex_;
  cleanup->mem = mem_;
  cleanup->imm = imm_;
  cleanup->version = versions_->current();
  internal_iter->RegisterCleanup(CleanupIteratorState, cleanup, NULL);

  mutex_.Unlock();
  return internal_iter;
}
```

33. NewInterator

实现在DBIter里面.工厂方式进行创建.

```
Iterator* DBImpl::NewIterator(const ReadOptions& options) {
  // 了解当前最近的snapshot sequence number.
  SequenceNumber latest_snapshot;
  Iterator* internal_iter = NewInternalIterator(options, &latest_snapshot);
  // 调用NewDBIterator来进行创建.具体可以查看DBIter实现.
  return NewDBIterator(
      &dbname_, env_, user_comparator(), internal_iter,
      (options.snapshot != NULL
       ? reinterpret_cast<const SnapshotImpl*>(options.snapshot)->number_
       : latest_snapshot));
}
```

10.3.4 DBIter

db/db_iter.cc DBIter通过工厂方法创建.然后就DBIter结构以及里面的接口看看. 关于这个iterator的具体细节的话可以不用了解非常清楚，稍微了解工作原理即可。 实际上里面FindNextUserEntry和FindPrevUserEntry没有仔细阅读.:(.

1. NewDBIterator

```
Iterator* NewDBIterator(
    const std::string* dbname,
    Env* env,
    const Comparator* user_key_comparator,
    Iterator* internal_iter,
    const SequenceNumber& sequence) {
  return new DBIter(dbname, env, user_key_comparator, internal_iter, sequence);
}
```

2. DBIter

```
// Memtables and sstables that make the DB representation contain
// (userkey,seq,type) => uservalue entries.  DBIter
// combines multiple entries for the same userkey found in the DB
// representation into a single entry while accounting for sequence
// numbers, deletion markers, overwrites, etc.
class DBIter: public Iterator {
 public:
  // Which direction is the iterator currently moving?
  // (1) When moving forward, the internal iterator is positioned at
  //     the exact entry that yields this->key(), this->value()
  // (2) When moving backwards, the internal iterator is positioned
  //     just before all entries whose user key == this->key().
  enum Direction {
    kForward,
    kReverse
  };

  DBIter(const std::string* dbname, Env* env,
         const Comparator* cmp, Iterator* iter, SequenceNumber s)
      : dbname_(dbname),
        env_(env),
        user_comparator_(cmp),
        iter_(iter),
        sequence_(s),
        direction_(kForward), // 默认是向前查找.
        valid_(false) { // 当前没有任何kv.
  }
  virtual ~DBIter() {
    delete iter_;
  }
 private:
  const std::string* const dbname_;
  Env* const env_;
  const Comparator* const user_comparator_;
  Iterator* const iter_;
  SequenceNumber const sequence_;

  Status status_;
  // 如果是kReverse方向的话,那么从这里返回
  // 否则直接调用iter返回.
  std::string saved_key_;     // == current key when direction_==kReverse
  std::string saved_value_;   // == current raw value when direction_==kReverse
  Direction direction_;
  bool valid_;
};
```

3. ClearSavedValue

   清除saved_value这个字段.好像有点技巧.可以看看代码啥的测试一下.

```
inline void ClearSavedValue() {
  if (saved_value_.capacity() > 1048576) { // >1M的话那么直接swap.
    std::string empty;
    swap(empty, saved_value_);
  } else { // 否则使用clear.
    saved_value_.clear();
  }
}
```

4. ParseKey

工作非常简单就是从iter得到对应的key.

```
inline bool DBIter::ParseKey(ParsedInternalKey* ikey) {
  if (!ParseInternalKey(iter_->key(), ikey)) {
    status_ = Status::Corruption("corrupted internal key in DBIter");
    return false;
  } else {
    return true;
  }
}
```

5. SaveKey

```
inline void SaveKey(const Slice& k, std::string* dst) {
  dst->assign(k.data(), k.size());
}
```

6. Seek

```
void DBIter::Seek(const Slice& target) {
  direction_ = kForward;
  ClearSavedValue(); // 将saved_value_清空.
  saved_key_.clear(); // 将saved_key_清空.
  AppendInternalKey( // 重新设置查询key.保存到saved_key_.
      &saved_key_, ParsedInternalKey(target, sequence_, kValueTypeForSeek));
  iter_->Seek(saved_key_);
  if (iter_->Valid()) {
    FindNextUserEntry(false, &saved_key_ /* temporary storage */);
  } else {
    valid_ = false;
  }
}
```

7. Next

```
void DBIter::Next() {
  assert(valid_);

  if (direction_ == kReverse) {  // Switch directions?
    direction_ = kForward;
    // iter_ is pointing just before the entries for this->key(),
    // so advance into the range of entries for this->key() and then
    // use the normal skipping code below.
    if (!iter_->Valid()) {
      iter_->SeekToFirst();
    } else {
      iter_->Next();
    }
    if (!iter_->Valid()) {
      valid_ = false;
      saved_key_.clear();
      return;
    }
  }

  // Temporarily use saved_key_ as storage for key to skip.
  std::string* skip = &saved_key_;
  SaveKey(ExtractUserKey(iter_->key()), skip);
  FindNextUserEntry(true, skip);
}
```

8. Prev

```
void DBIter::Prev() {
  assert(valid_);

  if (direction_ == kForward) {  // Switch directions?
    // iter_ is pointing at the current entry.  Scan backwards until
    // the key changes so we can use the normal reverse scanning code.
    // 首先向前一直找到略过当前saved_key的内容.
    assert(iter_->Valid());  // Otherwise valid_ would have been false
    SaveKey(ExtractUserKey(iter_->key()), &saved_key_);
    while (true) {
      iter_->Prev();
      if (!iter_->Valid()) {
        valid_ = false;
        saved_key_.clear();
        ClearSavedValue();
        return;
      }
      if (user_comparator_->Compare(ExtractUserKey(iter_->key()),
                                    saved_key_) < 0) {
        break;
      }
    }
    direction_ = kReverse;
  }

  FindPrevUserEntry();
}
```

9. FindNextUserEntry

我们需要考虑到在DBIter里面处理的是mergeiterator.多路的话可能会存在很多.

:现在才发现原来在find的时候并没有将sequence number放在里面而仅仅是比较user key 的内容，然后将所有的user key内容放在一起然后来处理sequence.不过对于memtable等 在插入的时候还是按照user key+sequence来进行存放.按照这个理解的话很多事情就比较好懂了.

总感觉这个地方可能存在问题，没有考虑到delete k然后add k的情况.不过撇开这个问题就好理解多了， 我们这里得到skip之后的话就可以在next的时候需要越过skip这个key.因为上次已经得到这个key的内容了.

```
void DBIter::FindNextUserEntry(bool skipping, std::string* skip) {
  // Loop until we hit an acceptable entry to yield
  assert(iter_->Valid());
  assert(direction_ == kForward);
  do {
    ParsedInternalKey ikey;
    if (ParseKey(&ikey) && ikey.sequence <= sequence_) { // 首先需要满足sequence.
      switch (ikey.type) {
        case kTypeDeletion: // 如果是deletion的话,那么保存这个skip的内容.
          // Arrange to skip all upcoming entries for this key since
          // they are hidden by this deletion.
          SaveKey(ikey.user_key, skip);
          skipping = true;
          break;
        case kTypeValue:
          if (skipping &&
              user_comparator_->Compare(ikey.user_key, *skip) <= 0) {
            // Entry hidden
          } else {
            valid_ = true;
            saved_key_.clear();
            return;
          }
          break;
      }
    }
    iter_->Next();
  } while (iter_->Valid());
  saved_key_.clear();
  valid_ = false;
}
```

10. FindPrevUserEntry

```
void DBIter::FindPrevUserEntry() {
  assert(direction_ == kReverse);

  ValueType value_type = kTypeDeletion;
  if (iter_->Valid()) {
    do {
      ParsedInternalKey ikey;
      if (ParseKey(&ikey) && ikey.sequence <= sequence_) {
        if ((value_type != kTypeDeletion) &&
            user_comparator_->Compare(ikey.user_key, saved_key_) < 0) {
          // We encountered a non-deleted value in entries for previous keys,
          break;
        }
        value_type = ikey.type;
        if (value_type == kTypeDeletion) {
          saved_key_.clear();
          ClearSavedValue();
        } else {
          Slice raw_value = iter_->value();
          if (saved_value_.capacity() > raw_value.size() + 1048576) {
```

```
          std::string empty;
          swap(empty, saved_value_);
        }
        SaveKey(ExtractUserKey(iter_->key()), &saved_key_);
        saved_value_.assign(raw_value.data(), raw_value.size());
      }
    }
    iter_->Prev();
  } while (iter_->Valid());
}

if (value_type == kTypeDeletion) {
  // End
  valid_ = false;
  saved_key_.clear();
  ClearSavedValue();
  direction_ = kForward;
} else {
  valid_ = true;
}
}
```

11. SeekToFirst

```
void DBIter::SeekToFirst() {
  direction_ = kForward;
  ClearSavedValue();
  iter_->SeekToFirst();
  if (iter_->Valid()) {
    FindNextUserEntry(false, &saved_key_ /* temporary storage */);
  } else {
    valid_ = false;
  }
}
```

12. SeekToLast

```
void DBIter::SeekToLast() {
  direction_ = kReverse;
  ClearSavedValue();
  iter_->SeekToLast();
  FindPrevUserEntry();
}
```

10.3.5 LookupKey

db/dbformat.h LookupKey是为了方便对不同的结构进行查询的key结构.抽象出来的话会使得操作更加方便. 我们只需要传入我们的user_key之后的话，那么就可以构造出对应查询结构的key.首先我们看看结构

```
// A helper class useful for DBImpl::Get()
class LookupKey {
 public:
  // Initialize *this for looking up user_key at a snapshot with
  // the specified sequence number.
  LookupKey(const Slice& user_key, SequenceNumber sequence);

  ~LookupKey();

  // Return a key suitable for lookup in a MemTable.
  // 返回能够为memtable查询的key.
  Slice memtable_key() const { return Slice(start_, end_ - start_); }

  // Return an internal key (suitable for passing to an internal iterator)
  // 返回internal key.这个应该是作为sstable查询的key.
  // 后面我们看看InternalKey的结构.
  Slice internal_key() const { return Slice(kstart_, end_ - kstart_); }

  // 返回user_key本身.
  // Return the user key
  Slice user_key() const { return Slice(kstart_, end_ - kstart_ - 8); }

 private:
  // 这里面给出了传入user_key之后构造出key的格式.
  // We construct a char array of the form:
  //    klength  varint32               <-- start_
  //    userkey  char[klength]          <-- kstart_
  //    tag      uint64
  //                                    <-- end_
  // The array is a suitable MemTable key.
  // The suffix starting with "userkey" can be used as an InternalKey.
  const char* start_;
  const char* kstart_;
  const char* end_;
  char space_[200];      // Avoid allocation for short keys
};
```

结构非常好理解，在看看构造函数和析构函数即可

```
inline LookupKey::~LookupKey() {
  // 这里space_是为了对于short keys不进行分配.
  if (start_ != space_) delete[] start_;
}

LookupKey::LookupKey(const Slice& user_key, SequenceNumber s) {
  size_t usize = user_key.size();
  // klength占用5个字节
  // tag占用8个字节.
  size_t needed = usize + 13;  // A conservative estimate
  char* dst;
  if (needed <= sizeof(space_)) {
    dst = space_;
  } else {
    dst = new char[needed];
  }
  start_ = dst;
  dst = EncodeVarint32(dst, usize + 8);
  kstart_ = dst;
  memcpy(dst, user_key.data(), usize);
  dst += usize;
  EncodeFixed64(dst, PackSequenceAndType(s, kValueTypeForSeek));
  dst += 8;
  end_ = dst;
}
```

10.3.6 ValueType

db/dbformat.h ValueType是存在于internal key内部的key类型.有普通类型和删除类型.

```
// Value types encoded as the last component of internal keys.
// DO NOT CHANGE THESE ENUM VALUES: they are embedded in the on-disk
// data structures.
enum ValueType {
  kTypeDeletion = 0x0,
  kTypeValue = 0x1
};
// kValueTypeForSeek defines the ValueType that should be passed when
// constructing a ParsedInternalKey object for seeking to a particular
// sequence number (since we sort sequence numbers in decreasing order
// and the value type is embedded as the low 8 bits in the sequence
// number in internal keys, we need to use the highest-numbered
// ValueType, not the lowest).
static const ValueType kValueTypeForSeek = kTypeValue;
```

10.3.7 SequenceNumber

db/dbfotmat.h SequenceNumber也存在于internal key内部，表示这个key的序号。现在就我自己的理解， 能想到这个序号的用户就是用来完成snapshot.

```
typedef uint64_t SequenceNumber;

// We leave eight bits empty at the bottom so a type and sequence#
// can be packed together into 64-bits.
static const SequenceNumber kMaxSequenceNumber =
    ((0x1ull << 56) - 1);
```

10.3.8 InternalKey

db/dbformat.h InternalKey的实现一份还存在于MemTable里面.因为从WriteBatch是首先写入MemTable的. 这个部分MemTable并没有复用而是重新实现.InternalKey应该也存在于SSTable里面.内部非常简单就是std::string 存储打包之后的格式。里面的方法比较多但是都相对非常简单。

```
// Modules in this directory should keep internal keys wrapped inside
// the following class instead of plain strings so that we do not
// incorrectly use string comparisons instead of an InternalKeyComparator.
class InternalKey {
 private:
  std::string rep_;
 public:
  InternalKey() { }   // Leave rep_ as empty to indicate it is invalid
  InternalKey(const Slice& user_key, SequenceNumber s, ValueType t) {
    AppendInternalKey(&rep_, ParsedInternalKey(user_key, s, t));
  }

  void DecodeFrom(const Slice& s) { rep_.assign(s.data(), s.size()); }
  Slice Encode() const {
    assert(!rep_.empty());
    return rep_;
  }

  Slice user_key() const { return ExtractUserKey(rep_); }

  void SetFrom(const ParsedInternalKey& p) {
    rep_.clear();
    AppendInternalKey(&rep_, p);
  }

  void Clear() { rep_.clear(); }

  std::string DebugString() const;
};
```

1. ExtracrUserKey

   之前我们知道user key是如何分布的了.后面8个字节有附加信息.

```
inline Slice ExtractUserKey(const Slice& internal_key) {
  assert(internal_key.size() >= 8);
  return Slice(internal_key.data(), internal_key.size() - 8);
}
```

2. ExtractValueType

```
inline ValueType ExtractValueType(const Slice& internal_key) {
  assert(internal_key.size() >= 8);
  const size_t n = internal_key.size();
  uint64_t num = DecodeFixed64(internal_key.data() + n - 8);
  unsigned char c = num & 0xff;
  return static_cast<ValueType>(c);
}
```

3. ParsedInternalKey

   ParsedInternalKey是从InternalKey解析之后的表示.非常简单.

```
struct ParsedInternalKey {
  Slice user_key;
  SequenceNumber sequence;
  ValueType type;

  ParsedInternalKey() { }  // Intentionally left uninitialized (for speed)
  ParsedInternalKey(const Slice& u, const SequenceNumber& seq, ValueType t)
      : user_key(u), sequence(seq), type(t) { }
  std::string DebugString() const;
};
```

4. InternalKeyEncodingLength

   如果ParsedInternalKey打包称为InternalKey的长度.

```
// Return the length of the encoding of "key".
inline size_t InternalKeyEncodingLength(const ParsedInternalKey& key) {
  return key.user_key.size() + 8;
}
```

5. AppendInternalKey

   将ParsedInternalKey直接序列化到二进制格式.

```
void AppendInternalKey(std::string* result, const ParsedInternalKey& key) {
  result->append(key.user_key.data(), key.user_key.size());
  PutFixed64(result, PackSequenceAndType(key.sequence, key.type));
}
```

6. PackSequenceAndType

   sequence number占据高56bits,type占据低8bits

```
static uint64_t PackSequenceAndType(uint64_t seq, ValueType t) {
  assert(seq <= kMaxSequenceNumber);
  assert(t <= kValueTypeForSeek);
  return (seq << 8) | t;
}
```

7. ParseInternalKey

   根据InternalKey解析出ParsedInternalKey

```
inline bool ParseInternalKey(const Slice& internal_key,
                             ParsedInternalKey* result) {
  const size_t n = internal_key.size();
  if (n < 8) return false;
  uint64_t num = DecodeFixed64(internal_key.data() + n - 8);
  unsigned char c = num & 0xff;
  result->sequence = num >> 8;
  result->type = static_cast<ValueType>(c);
  result->user_key = Slice(internal_key.data(), n - 8);
  return (c <= static_cast<unsigned char>(kTypeValue));
}
```

10.3.9 InternalKeyComparator

db/dbformat.cc 我们在MemTable::KeyComparator::operator()里面看到了调用InternalKeyComparator的Compare方法. Compare接收两个Slice对象。对象是这样encode的,key_size + key_data + (seq << 8) | type(8 bytes). 其中key_size包括了后面附加的8个字节.我们来看看InternalKeyComparator是怎么实现的.底层的Comparator是用来直接比较UserKey的，没有考虑sequence number.

```
// A comparator for internal keys that uses a specified comparator for
// the user key portion and breaks ties by decreasing sequence number.
class InternalKeyComparator : public Comparator {
 private:
  const Comparator* user_comparator_;
 public:
  explicit InternalKeyComparator(const Comparator* c) : user_comparator_(c) { }
  virtual const char* Name() const; // "leveldb.InternalKeyComparator"
  virtual int Compare(const Slice& a, const Slice& b) const;
  virtual void FindShortestSeparator(
      std::string* start,
      const Slice& limit) const;
  virtual void FindShortSuccessor(std::string* key) const;

  const Comparator* user_comparator() const { return user_comparator_; }

  int Compare(const InternalKey& a, const InternalKey& b) const;
};
```

1. Compare

```
int InternalKeyComparator::Compare(const Slice& akey, const Slice& bkey) const {
  // Order by:
  //    increasing user key (according to user-supplied comparator)
  //    decreasing sequence number
  //    decreasing type (though sequence# should be enough to disambiguate)
  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) { // 按照sequence number比较.
     // 之前我们在MemTableInserter里面可以看到sequence number是不断增加的.
      r = -1;
    } else if (anum < bnum) {
      r = +1;
    }
  }
  return r;
}
```

2. FindShortestSeparator

```
void InternalKeyComparator::FindShortestSeparator(
      std::string* start,
      const Slice& limit) const {
  // Attempt to shorten the user portion of the key
  Slice user_start = ExtractUserKey(*start);
  Slice user_limit = ExtractUserKey(limit);
  std::string tmp(user_start.data(), user_start.size());
  // 首先使用user_comparator找到更短的
  user_comparator_->FindShortestSeparator(&tmp, user_limit);
  if (tmp.size() < user_start.size() && // 然后比较是否真的更短.
      user_comparator_->Compare(user_start, tmp) < 0) {
    // 如果更短的话，那么会将后面的8个字节补齐
    // 这里的8个字节使用kMaxSequenceNumber以及kValueTypeForSeek
    // 其中kMaxSequenceNumber == (1 << 56)-1
    // 而kValueTypeForSeek == KValueType.(就是普通的kv类型)
    // User key has become shorter physically, but larger logically.
    // Tack on the earliest possible number to the shortened user key.
    PutFixed64(&tmp, PackSequenceAndType(kMaxSequenceNumber,kValueTypeForSeek));
    assert(this->Compare(*start, tmp) < 0);
    assert(this->Compare(tmp, limit) < 0);
    start->swap(tmp);
  }
}
```

3. FindShortSuccessor

```
void InternalKeyComparator::FindShortSuccessor(std::string* key) const {
  Slice user_key = ExtractUserKey(*key);
  std::string tmp(user_key.data(), user_key.size());
  user_comparator_->FindShortSuccessor(&tmp); // 首先通过user_comparator_找到下一个
  if (tmp.size() < user_key.size() &&
      user_comparator_->Compare(user_key, tmp) < 0) {
    // 如果确实更短的话那么会加上特定的8字节附加信息.
    // 关于附加信息在上面那个函数已经解释过了.
    // User key has become shorter physically, but larger logically.
    // Tack on the earliest possible number to the shortened user key.
    PutFixed64(&tmp, PackSequenceAndType(kMaxSequenceNumber,kValueTypeForSeek));
    assert(this->Compare(*key, tmp) < 0);
    key->swap(tmp);
  }
}
```

10.3.10 FileMetaData

db/version_edit.h 对于一个sstable的元信息表示.

```
struct FileMetaData {
  int refs;
  // 首先会设置一个初值
  int allowed_seeks;          // Seeks allowed until compaction
  uint64_t number; // file_number.比如1.sst.这个结合BuildTable和TableCache可以理解意思.
  uint64_t file_size;         // File size in bytes
  InternalKey smallest;       // Smallest internal key served by table
  InternalKey largest;        // Largest internal key served by table

  FileMetaData() : refs(0), allowed_seeks(1 << 30), file_size(0) { }
};
```

10.3.11 Version

db/version_set.h 对于Version可以理解为每个Snapshot对应的内容。因为每个snapshot可能需要持有不同的文件， 这样Version里面可以带上所需要管理的文件。如果释放Snapshot的话并且也可以释放Version的话，那么就可以认为这些文件 没有被任何Snapshot所引用就可以被回收。首先看看Version结构题里面的字段。

```
class Version {
 public:
  // private:
 public:
  friend class Compaction;
  friend class VersionSet;

  class LevelFileNumIterator;
  Iterator* NewConcatenatingIterator(const ReadOptions&, int level) const;

  VersionSet* vset_;            // VersionSet to which this Version belongs
  Version* next_;               // Next version in linked list
  Version* prev_;               // Previous version in linked list
  int refs_;                    // Number of live refs to this version

  // List of files per level
  // 每个level的files_都是经过排序的。
  // 对于level0可能存在overlap,对于level-x不存在overlap.
  std::vector<FileMetaData*> files_[config::kNumLevels]; // 这个version在各个level保持的文件.

  // Next file to compact based on seek stats.
  // 根据seek的统计下次需要进行compact的文件.
  FileMetaData* file_to_compact_;  // 下次进行compact文件
  int file_to_compact_level_; // 这个文件所在的level.

  // Level that should be compacted next and its compaction score.
  // Score < 1 means compaction is not strictly needed.  These fields
  // are initialized by Finalize().
  double compaction_score_;
  int compaction_level_;

  // 构造函数非常简单.
  explicit Version(VersionSet* vset)
      : vset_(vset), next_(this), prev_(this), refs_(0),
        file_to_compact_(NULL),
        file_to_compact_level_(-1),
        compaction_score_(-1),
        compaction_level_(-1) {
  }

  ~Version();
};
```

1. AddIterators

   将所持有的所有的level文件打开并且返回iterator.从注释上来看的话得到这些iterators的话是为了进行merge.

```
void Version::AddIterators(const ReadOptions& options,
                           std::vector<Iterator*>* iters) {
  // Merge all level zero files together since they may overlap
  // 打开所有的level0文件.
  for (size_t i = 0; i < files_[0].size(); i++) {
    iters->push_back(
        vset_->table_cache_->NewIterator(
            options, files_[0][i]->number, files_[0][i]->file_size));
  }

  // 打开其他level的文件.关于这个concatenating后面会仔细分析.
  // For levels > 0, we can use a concatenating iterator that sequentially
  // walks through the non-overlapping files in the level, opening them
  // lazily.
  for (int level = 1; level < config::kNumLevels; level++) {
    if (!files_[level].empty()) {
      iters->push_back(NewConcatenatingIterator(options, level));
    }
  }
}
```

2. NewConcatenatingIterator

```
Iterator* Version::NewConcatenatingIterator(const ReadOptions& options,
                                            int level) const {
  return NewTwoLevelIterator(
      // 注意这里的level>1.每个文件之间是没有overlap的.
      // 并且这里我们也可以知道files_对于meta之间都是进行了排序的。
      new LevelFileNumIterator(vset_->icmp_, &files_[level]), // 一级遍历器采用LevelFileNumberIterator.
      // 映射到二级遍历器的话使用GetFileIterator来获得.
      &GetFileIterator, vset_->table_cache_, options);
}
```

3. LevelFileNumIterator

结构应该非常简单，我们大致看看即可。主要是关心一下二级映射函数GetFileIterator.对于 key的话采用这个filemetadata里面的最大的key,而value采用filenumber+filesize表示.

```
class Version::LevelFileNumIterator : public Iterator {
 public:
  LevelFileNumIterator(const InternalKeyComparator& icmp,
                       const std::vector<FileMetaData*>* flist)
      : icmp_(icmp),
        flist_(flist),
        index_(flist->size()) {        // Marks as invalid
  }
  virtual bool Valid() const {
    return index_ < flist_->size();
  }
  virtual void Seek(const Slice& target) {
    index_ = FindFile(icmp_, *flist_, target); // 找到这个target所存在的最小的index.
    // 这个后面会具体分析.之前看到了这层level都是disjoint的
    // 所以在FindFile里面可以根据smallest也可以根据largest来进行查找.
  }
  virtual void SeekToFirst() { index_ = 0; }
  virtual void SeekToLast() {
    index_ = flist_->empty() ? 0 : flist_->size() - 1;
  }
  virtual void Next() {
    assert(Valid());
    index_++;
  }
  virtual void Prev() {
    assert(Valid());
    if (index_ == 0) {
      index_ = flist_->size();  // Marks as invalid
    } else {
      index_--;
    }
  }
  Slice key() const {
    assert(Valid());
    return (*flist_)[index_]->largest.Encode();
  }
  Slice value() const {
    assert(Valid()); // file_number + file_size.
    // 关于如何使用这个value.应该是根据这个value对应到这个具体文件的iterator.
    EncodeFixed64(value_buf_, (*flist_)[index_]->number);
    EncodeFixed64(value_buf_+8, (*flist_)[index_]->file_size);
    return Slice(value_buf_, sizeof(value_buf_));
  }
  virtual Status status() const { return Status::OK(); }
 private:
  const InternalKeyComparator icmp_;
  const std::vector<FileMetaData*>* const flist_;
  uint32_t index_;

  // Backing store for value().  Holds the file number and size.
  mutable char value_buf_[16];
};
```

4. GetFileIterator

根据上面的分析file_value就是file_number + file_size.这里我们可以知道file_number应该是全局唯一的， 而不是在level上面唯一的。关于TableCache后面会分析。

```
static Iterator* GetFileIterator(void* arg,
                                 const ReadOptions& options,
                                 const Slice& file_value) {
  TableCache* cache = reinterpret_cast<TableCache*>(arg);
  if (file_value.size() != 16) {
    return NewErrorIterator(
        Status::Corruption("FileReader invoked with unexpected value"));
  } else {
    return cache->NewIterator(options,
                              DecodeFixed64(file_value.data()),
                              DecodeFixed64(file_value.data() + 8));
  }
}
```

5. FindFile

   db/version_set.h 语义直接阅读注释即可.注意这里files都是排好序并且是不重叠的。按照二分法搜索largest key即可.

```
// Return the smallest index i such that files[i]->largest >= key.
// Return files.size() if there is no such file.
// REQUIRES: "files" contains a sorted list of non-overlapping files.
int FindFile(const InternalKeyComparator& icmp,
             const std::vector<FileMetaData*>& files,
             const Slice& key) {
  uint32_t left = 0;
  uint32_t right = files.size();
  while (left < right) {
    uint32_t mid = (left + right) / 2;
    const FileMetaData* f = files[mid];
    if (icmp.InternalKeyComparator::Compare(f->largest.Encode(), key) < 0) {
      // Key at "mid.largest" is < "target".  Therefore all
      // files at or before "mid" are uninteresting.
      left = mid + 1;
    } else {
      // Key at "mid.largest" is >= "target".  Therefore all files
      // after "mid" are uninteresting.
      right = mid;
    }
  }
  return right;
}
```

6. SomeFileOverlapsRange

   db/version_set.h 判断是否有文件和某个范围[smallest,largest]相交.注意这里这里files也是按照key排序的.

```
// Returns true iff some file in "files" overlaps the user key range
// [*smallest,*largest].
// smallest==NULL represents a key smaller than all keys in the DB.
// largest==NULL represents a key largest than all keys in the DB.
// REQUIRES: If disjoint_sorted_files, files[] contains disjoint ranges
//           in sorted order.
bool SomeFileOverlapsRange(
    const InternalKeyComparator& icmp,
    bool disjoint_sorted_files, // 表示files
    const std::vector<FileMetaData*>& files,
    const Slice* smallest_user_key,
    const Slice* largest_user_key) {
  const Comparator* ucmp = icmp.user_comparator();
  if (!disjoint_sorted_files) {
    // 如果文件之间可能存在overlap的话.那么必须顺序.
    // 那么需要遍历每个文件.判断这个文件是否和[small,large]相交.
    // 如果有相交，那么直接返回.:).
    // Need to check against all files
    for (int i = 0; i < files.size(); i++) {
      const FileMetaData* f = files[i];
      // AfterFile和BeforeFile稍后分析.
      if (AfterFile(ucmp, smallest_user_key, f) ||
          BeforeFile(ucmp, largest_user_key, f)) {
        // No overlap
      } else {
        return true;  // Overlap
      }
    }
    return false;
  }

  // 如果文件本身之间没有overlap的话.
  // 那么我们只需要首先按照二分方法找到相交文件index.
  // 然后针对这个index分析即可.
  // Binary search over file list
  uint32_t index = 0;
  if (smallest_user_key != NULL) {
    // Find the earliest possible internal key for smallest_user_key
    InternalKey small(*smallest_user_key, kMaxSequenceNumber,kValueTypeForSeek);
    index = FindFile(icmp, files, small.Encode());
  }

  if (index >= files.size()) {
    // beginning of range is after all files, so no overlap.
    return false;
  }

  return !BeforeFile(ucmp, largest_user_key, files[index]);
}
```

7. AfterFile

   判断user_key是否在这个file之后.

```
static bool AfterFile(const Comparator* ucmp,
                      const Slice* user_key, const FileMetaData* f) {
  // NULL user_key occurs before all keys and is therefore never after *f
  return (user_key != NULL &&
          ucmp->Compare(*user_key, f->largest.user_key()) > 0);
}
```

8. BeforeFile

   判断user_key是否在这个file之前.

```
static bool BeforeFile(const Comparator* ucmp,
                       const Slice* user_key, const FileMetaData* f) {
  // NULL user_key occurs after all keys and is therefore never before *f
  return (user_key != NULL &&
          ucmp->Compare(*user_key, f->smallest.user_key()) < 0);
}
```

9. GetStats

   这个结构反应的是查询到的一些附加结果可以用来指导compaction.

```
struct GetStats {
  FileMetaData* seek_file; // 被seek到的文件
  int seek_file_level; // 以及这个文件所处level.
};
```

10. NewestFirst

    按照file_number进行排序.逆序.越迟生成的file那么是最新的。

```
static bool NewestFirst(FileMetaData* a, FileMetaData* b) {
  return a->number > b->number;
}
```

11. Get

```
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 = NULL;
  stats->seek_file_level = -1;
  FileMetaData* last_file_read = NULL;
  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 an 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.
      // 因为level0存在重叠，所以需要都进行搜索.
      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;
      // 然后按照进行排序.最新生成的file放在最前面.
      std::sort(tmp.begin(), tmp.end(), NewestFirst);
      files = &tmp[0];
      num_files = tmp.size();
    } else {
      // 对于其他level的话越低level越新.并且level内部没有overlap
      // 那么可以通过简单的二分法来判断哪个文件.只会存在一个文件.
      // Binary search to find earliest index whose largest key >= ikey.
      uint32_t index = FindFile(vset_->icmp_, files_[level], ikey);
      if (index >= num_files) {
        files = NULL;
        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 = NULL;
          num_files = 0;
        } else {
          files = &tmp2;
          num_files = 1;
        }
      }
    }

    for (uint32_t i = 0; i < num_files; ++i) {
      if (last_file_read != NULL && stats->seek_file == NULL) {
        // We have had more than one seek for this read.  Charge the 1st file.
        // stats这里只保留第一次读取的文件.
        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;

      // 通过iterator的seek方式来进行查找.
      Iterator* iter = vset_->table_cache_->NewIterator(
          options,
          f->number,
          f->file_size);
      iter->Seek(ikey);
      // seek只是一个大概位置这里需要精确比较返回值.后面我们仔细分析GetValue.
      const bool done = GetValue(ucmp, iter, user_key, value, &s);
      if (!iter->status().ok()) {
        s = iter->status();
        delete iter;
        return s;
      } else {
        delete iter;
        if (done) {
          return s;
        }
      }
    }
  }

  return Status::NotFound(Slice());  // Use an empty error message for speed
}
```

12. GetValue

GetValue根据某个iter和key判断这个key是否为想查询的key.

```
// If "*iter" points at a value or deletion for user_key, store
// either the value, or a NotFound error and return true.
// Else return false.
static bool GetValue(const Comparator* cmp,
                     Iterator* iter, const Slice& user_key,
                     std::string* value,
                     Status* s) {
  if (!iter->Valid()) {
    return false;
  }
  ParsedInternalKey parsed_key;
  // 对于iterator里面是internal key.解析出parsed internal key出来.
  if (!ParseInternalKey(iter->key(), &parsed_key)) {
    *s = Status::Corruption("corrupted key for ", user_key);
    return true;
  }
  // 比较user key内容是否一致.这里没有考虑sequence number.
  if (cmp->Compare(parsed_key.user_key, user_key) != 0) {
    return false;
  }
  // 然后判断这个key是否标记删除.
  switch (parsed_key.type) {
    case kTypeDeletion:
      *s = Status::NotFound(Slice());  // Use an empty error message for speed
      break;
    case kTypeValue: {
      Slice v = iter->value();
      value->assign(v.data(), v.size());
      break;
    }
  }
  return true;
}
```

13. UpdateStats

    根据stats来更新version内部状态.设置将要compaction文件以及对应的level是什么.

```
bool Version::UpdateStats(const GetStats& stats) {
  FileMetaData* f = stats.seek_file; // 如果stats里面标记了数据的话.
  if (f != NULL) {
    f->allowed_seeks--; //
    if (f->allowed_seeks <= 0 && file_to_compact_ == NULL) {
      file_to_compact_ = f;
      file_to_compact_level_ = stats.seek_file_level;
      return true;
    }
  }
  return false;
}
```

14. GetOverlappingInputs

    得到某个level下面所有和[begin,end]有overlap的文件.语义不知道是否理解正确.因为代码里面有个地方没有太看懂.

```
// Store in "*inputs" all files in "level" that overlap [begin,end]
void Version::GetOverlappingInputs(
    int level,
    const InternalKey* begin,
    const InternalKey* end,
    std::vector<FileMetaData*>* inputs) {
  inputs->clear();
  Slice user_begin, user_end;
  if (begin != NULL) {
    user_begin = begin->user_key();
  }
  if (end != NULL) {
    user_end = end->user_key();
  }
  const Comparator* user_cmp = vset_->icmp_.user_comparator();
  for (size_t i = 0; i < files_[level].size(); ) {
    FileMetaData* f = files_[level][i++];
    const Slice file_start = f->smallest.user_key();
    const Slice file_limit = f->largest.user_key();
    if (begin != NULL && user_cmp->Compare(file_limit, user_begin) < 0) {
      // "f" is completely before specified range; skip it
    } else if (end != NULL && user_cmp->Compare(file_start, user_end) > 0) {
      // "f" is completely after specified range; skip it
    } else {
      inputs->push_back(f);
      if (level == 0) {
        // todo:实现上可能存在问题吧.看上去对于level0的话似乎在不断地过大范围.
        // Level-0 files may overlap each other.  So check if the newly
        // added file has expanded the range.  If so, restart search.
        if (begin != NULL && user_cmp->Compare(file_start, user_begin) < 0) {
          user_begin = file_start;
          inputs->clear();
          i = 0;
        } else if (end != NULL && user_cmp->Compare(file_limit, user_end) > 0) {
          user_end = file_limit;
          inputs->clear();
          i = 0;
        }
      }
    }
  }
}
```

15. OverlapInLevel

    判断某个level里面是否有文件和[small,large]这个范围内容的key重叠

```
bool Version::OverlapInLevel(int level,
                             const Slice* smallest_user_key,
                             const Slice* largest_user_key) {
  // (level>0)的话表示每个文件没有重叠的部分.
  return SomeFileOverlapsRange(vset_->icmp_, (level > 0), files_[level],
                               smallest_user_key, largest_user_key);
}
```

16. PickLevelForMemTableOutput

    根据[small,large]这个范畴选择某个level来作为memtable的output.这个range应该就是memtable的range. 这个策略比较精巧。这个策略在常数部分定义注释里面给出来了。

```
// db/dbformat.h

// 如果level0过低的话那么会造成过多compaction
// 但是level0多高的话那么会浪费磁盘空间因为key的重复.
// Maximum level to which a new compacted memtable is pushed if it
// does not create overlap.  We try to push to level 2 to avoid the
// relatively expensive level 0=>1 compactions and to avoid some
// expensive manifest file operations.  We do not push all the way to
// the largest level since that can generate a lot of wasted disk
// space if the same key space is being repeatedly overwritten.
static const int kMaxMemCompactLevel = 2;

// db/version_set.cc
static const int kTargetFileSize = 2 * 1048576;

// Maximum bytes of overlaps in grandparent (i.e., level+2) before we
// stop building a single file in a level->level+1 compaction.
static const int64_t kMaxGrandParentOverlapBytes = 10 * kTargetFileSize;
```

#=BEGIN_SRC C++ int Version::PickLevelForMemTableOutput(const Slice& smallest_user_key, const Slice& largest_user_key) { int level = 0; // 首先判断和level0是否有overlap.如果有overlapd的话那么选择level0. if (!OverlapInLevel(0, &smallest_user_key, &largest_user_key)) { / *Push to next level if there is no overlap in next level,* / and the #bytes overlapping in the level after that are limited. InternalKey start(smallest_user_key, kMaxSequenceNumber, kValueTypeForSeek); InternalKey limit(largest_user_key, 0, static_cast<ValueType>(0)); std::vector<FileMetaData*> overlaps; while (level < config::kMaxMemCompactLevel) { / 判断和其他*level*是否有*overlap*.如果存在*overlap*的那么也选择. / >0的level是不允许overlap的. if (OverlapInLevel(level + 1, &smallest_user_key, &largest_user_key)) { break; } / 如果没有*overlap*的话那么判断和*grandparent level*重叠文件.判断重叠文件大小. / 这个策略没有太明白. / 关于这个策略估计可以从*Compaction::IsTrivialMove*的注释里面看到.这里应该是担心*grand parent*本身就存在很多*overlap* / 如果放在level+1做compaction的话，会造成grand parent这个部分合并时间过长. / 从值上来看意思应该是存在*overlap*的文件数目不应该超过*10*个*(*假设每个*overlap*文件都是*MaxSizeForLevel*的*). /* Avoid a move if there is lots of overlapping grandparent data. / *Otherwise, the move could create a parent file that will require* / a very expensive merge later on. GetOverlappingInputs(level + 2, &start, &limit, &overlaps); const int64_t sum = TotalFileSize(overlaps); if (sum > kMaxGrandParentOverlapBytes) { break; } level++; } } return level; } #+END_SRC

17. DebugString

    DebugString作为Version的调试信息打印出来.我们可以稍微得到一点启发吧.尤其如果是自己调试的话.

```
std::string Version::DebugString() const {
  std::string r;
  for (int level = 0; level < config::kNumLevels; level++) {
    // E.g.,
    //   --- level 1 ---
    //   17:123['a' .. 'd']
    //   20:43['e' .. 'g']
    r.append("--- level ");
    AppendNumberTo(&r, level);
    r.append(" ---\n");
    const std::vector<FileMetaData*>& files = files_[level];
    for (size_t i = 0; i < files.size(); i++) {
      r.push_back(' ');
      AppendNumberTo(&r, files[i]->number); // file_number
      r.push_back(':');
      AppendNumberTo(&r, files[i]->file_size); // file大小.
      r.append("[");
      r.append(files[i]->smallest.DebugString()); // 最小和最大的key的打印.
      r.append(" .. ");
      r.append(files[i]->largest.DebugString());
      r.append("]\n");
    }
  }
  return r;
}
```

10.3.12 VersionSet

db/version_set.h VersionSet用来管理不同的Version并且应该维护了一些全局信息.还是首先看看结构

```
class VersionSet {
 public:
  VersionSet(const std::string& dbname,
             const Options* options,
             TableCache* table_cache,
             const InternalKeyComparator*);
  ~VersionSet();
 private:
  Env* const env_;
  const std::string dbname_;
  const Options* const options_;
  TableCache* const table_cache_;
  const InternalKeyComparator icmp_;
  uint64_t next_file_number_; // 下一个分配的file number
  uint64_t manifest_file_number_; // manifest file number.
  uint64_t last_sequence_; // 下次操作的sequence.
  // 从LogNumber和PrevLogNumber注释可以看出来
  // 分别表示当前使用的log number以及之前使用的log number(当前正在被压缩).
  uint64_t log_number_; //
  uint64_t prev_log_number_;  // 0 or backing store for memtable being compacted

  // Opened lazily
  WritableFile* descriptor_file_; // manifest文件
  log::Writer* descriptor_log_; // manifest以log形式打开.
  Version dummy_versions_;  // Head of circular doubly-linked list of versions. // 为了组织称为链表.
  Version* current_;        // == dummy_versions_.prev_ // 当前version

  // Per-level key at which the next compaction at that level should start.
  // Either an empty string, or a valid InternalKey.
  // 对于每层的话都会设置一个key.只有>这层设置的key才允许进行compaction.
  std::string compact_pointer_[config::kNumLevels]; // 每个level进行compaction使用的internal key
  // todo:似乎每个level进行compaction都配上了一个internal key
  // DONE:记录每层的最大key.
};
```

1. VersionSet

   首先看看构造函数和析构函数.

```
VersionSet::VersionSet(const std::string& dbname,
                       const Options* options,
                       TableCache* table_cache,
                       const InternalKeyComparator* cmp)
    : env_(options->env),
      dbname_(dbname),
      options_(options),
      table_cache_(table_cache),
      icmp_(*cmp),
      next_file_number_(2),
      // 下面这些字段都是通过Recover恢复的.
      manifest_file_number_(0),  // Filled by Recover()
      last_sequence_(0),
      log_number_(0),
      prev_log_number_(0),
      descriptor_file_(NULL),
      descriptor_log_(NULL),
      dummy_versions_(this),
      current_(NULL) {
  AppendVersion(new Version(this)); // 添加一个当前version.
}
VersionSet::~VersionSet() {
  current_->Unref();
  // 析构时候必须确保里面没有任何版本.
  assert(dummy_versions_.next_ == &dummy_versions_);  // List must be empty
  delete descriptor_log_;
  delete descriptor_file_;
}
```

2. NeedsCompaction

   当前versionset是否需要触发compaction操作.

```
// Returns true iff some level needs a compaction.
bool NeedsCompaction() const {
  Version* v = current_;
  return (v->compaction_score_ >= 1) || (v->file_to_compact_ != NULL);
}
```

3. AppendVersion

   添加version.非常简单修改引用计数挂载链表上.将version挂在version_set下面的话可以使得version_set了解到哪些文件依然是被正在使用的.

```
void VersionSet::AppendVersion(Version* v) {
  // Make "v" current
  assert(v->refs_ == 0);
  assert(v != current_);
  if (current_ != NULL) {
    current_->Unref();
  }
  current_ = v;
  v->Ref();

  // Append to linked list
  v->prev_ = dummy_versions_.prev_;
  v->next_ = &dummy_versions_;
  v->prev_->next_ = v;
  v->next_->prev_ = v;
}
```

4. NumLevelFiles

   得到某个level的文件数目.以current这个Version来计算的

```
int VersionSet::NumLevelFiles(int level) const {
  assert(level >= 0);
  assert(level < config::kNumLevels);
  return current_->files_[level].size();
}
```

5. NumLevelBytes

   某个level的文件大小.以current这个Version来计算的.

```
int64_t VersionSet::NumLevelBytes(int level) const {
  assert(level >= 0);
  assert(level < config::kNumLevels);
  return TotalFileSize(current_->files_[level]);
}
```

6. TotalFileSize

   根据file meta data得到所有文件大小.非常简单因为里面有file_size.

```
static int64_t TotalFileSize(const std::vector<FileMetaData*>& files) {
  int64_t sum = 0;
  for (size_t i = 0; i < files.size(); i++) {
    sum += files[i]->file_size;
  }
  return sum;
}
```

7. MarkFileNumberUsed

   标记当前使用了file_number进度.number表示已经使用的进度,所以下次分配必须+1.

```
void VersionSet::MarkFileNumberUsed(uint64_t number) {
  if (next_file_number_ <= number) {
    next_file_number_ = number + 1;
  }
}
```

8. ApproximateOffsetOf

   判断这个key在数据库内部大致偏移是多少.从这个实现里面我们可以看到.对于level-0的文件并不是排序的 对于level>0的文件都是按照range进行排序的。并且这个排序是按照smallest来进行排序的。

```
uint64_t VersionSet::ApproximateOffsetOf(Version* v, const InternalKey& ikey) {
  uint64_t result = 0;
  for (int level = 0; level < config::kNumLevels; level++) {
    const std::vector<FileMetaData*>& files = v->files_[level];
    for (size_t i = 0; i < files.size(); i++) {
      if (icmp_.Compare(files[i]->largest, ikey) <= 0) { // 如果比largest key大的话那么这个
        // 那么偏移需要算上整个文件.但是这里还包括了index block,meta block以及footer大小.
        // 所以只能够说是大致大小.
        // Entire file is before "ikey", so just add the file size
        result += files[i]->file_size;
      } else if (icmp_.Compare(files[i]->smallest, ikey) > 0) { // 如果smallest比当前key大的话那么取消.
        // Entire file is after "ikey", so ignore
        if (level > 0) { // 如果是>0的level那么可以直接取消.因为这个部分的file都是按照smallest排序的.
          // Files other than level 0 are sorted by meta->smallest, so
          // no further files in this level will contain data for
          // "ikey".
          break;
        }
      } else { // 如果存在交集的话那么那么可以构造出Table对象找到这个key大致偏移.
        // "ikey" falls in the range for this table.  Add the
        // approximate offset of "ikey" within the table.
        Table* tableptr;
        Iterator* iter = table_cache_->NewIterator(
            ReadOptions(), files[i]->number, files[i]->file_size, &tableptr);
        if (tableptr != NULL) {
          result += tableptr->ApproximateOffsetOf(ikey.Encode());
        }
        delete iter;
      }
    }
  }
  return result;
}
```

9. AddLiveFiles

   将version set里面所有version所持有的文件收集起来.非常简单:).

```
void VersionSet::AddLiveFiles(std::set<uint64_t>* live) {
  for (Version* v = dummy_versions_.next_;
       v != &dummy_versions_;
       v = v->next_) {
    for (int level = 0; level < config::kNumLevels; level++) {
      const std::vector<FileMetaData*>& files = v->files_[level];
      for (size_t i = 0; i < files.size(); i++) {
        live->insert(files[i]->number);
      }
    }
  }
}
```

10. LevelSummary

每层level的文件个数分别是多少.

```
const char* VersionSet::LevelSummary(LevelSummaryStorage* scratch) const {
  // Update code if kNumLevels changes
  assert(config::kNumLevels == 7);
  snprintf(scratch->buffer, sizeof(scratch->buffer),
           "files[ %d %d %d %d %d %d %d ]",
           int(current_->files_[0].size()),
           int(current_->files_[1].size()),
           int(current_->files_[2].size()),
           int(current_->files_[3].size()),
           int(current_->files_[4].size()),
           int(current_->files_[5].size()),
           int(current_->files_[6].size()));
  return scratch->buffer;
}
```

11. WriteSnapshot

将当前current version写入到磁盘记为log.方式是将内容copy到version edit对象里面去然后持久化. 过程还是非常简单吧:).

```
Status VersionSet::WriteSnapshot(log::Writer* log) {
  // TODO: Break up into multiple records to reduce memory usage on recovery?

  // Save metadata
  VersionEdit edit;
  edit.SetComparatorName(icmp_.user_comparator()->Name());

  // Save compaction pointers
  for (int level = 0; level < config::kNumLevels; level++) {
    if (!compact_pointer_[level].empty()) {
      InternalKey key;
      key.DecodeFrom(compact_pointer_[level]);
      edit.SetCompactPointer(level, key);
    }
  }

  // Save files
  for (int level = 0; level < config::kNumLevels; level++) {
    const std::vector<FileMetaData*>& files = current_->files_[level];
    for (size_t i = 0; i < files.size(); i++) {
      const FileMetaData* f = files[i];
      edit.AddFile(level, f->number, f->file_size, f->smallest, f->largest);
    }
  }

  std::string record;
  edit.EncodeTo(&record);
  return log->AddRecord(record);
}
```

12. Finalize

所谓的Finalize的含义应该是，如果我们不希望修改version这个结构之后我们应该做的事情。可能叫做Finish会更好。 相当于针对这个version而言的话，最高一级的compaction level是什么，分数是多少.

```
void VersionSet::Finalize(Version* v) {
  // Precomputed best level for next compaction
  int best_level = -1;
  double best_score = -1;

  for (int level = 0; level < config::kNumLevels-1; level++) {
    double score;
    if (level == 0) {
      // We treat level-0 specially by bounding the number of files
      // instead of number of bytes for two reasons:
      //
      // (1) With larger write-buffer sizes, it is nice not to do too
      // many level-0 compactions.
      //
      // (2) 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, or very high compression ratios, or lots of
      // overwrites/deletions).
      score = v->files_[level].size() /
```

```
        static_cast<double>(config::kL0_CompactionTrigger);
    } else {
      // Compute the ratio of current size to size limit.
      const uint64_t level_bytes = TotalFileSize(v->files_[level]);
      score = static_cast<double>(level_bytes) / MaxBytesForLevel(level);
    }

    if (score > best_score) {
      best_level = level;
      best_score = score;
    }
  }

  v->compaction_level_ = best_level;
  v->compaction_score_ = best_score;
}
```

13. Recover

从db_impl.cc里面的DB::Open可以看到,这里的Recover并没有将剩余的version_edit回放完成. 回放这个部分在LogAndApply里面完成.所以Recover可能只是恢复到以前某个状态.将当前的 CURRENT里面的manifest文件回放之后就得到最新的内容.然后根据里面的log信息回放后面的内容.

```
Status VersionSet::Recover() {
  struct LogReporter : public log::Reader::Reporter {
    Status* status;
    virtual void Corruption(size_t bytes, const Status& s) {
      if (this->status->ok()) *this->status = s;
    }
  };

  // 读取当前current这个内容,得到最新的manifest文件.
  // Read "CURRENT" file, which contains a pointer to the current manifest file
  std::string current;
  Status s = ReadFileToString(env_, CurrentFileName(dbname_), &current);
  if (!s.ok()) {
    return s;
  }
  if (current.empty() || current[current.size()-1] != '\n') {
    return Status::Corruption("CURRENT file does not end with newline");
  }
  current.resize(current.size() - 1);

  std::string dscname = dbname_ + "/" + current;
  SequentialFile* file;
  s = env_->NewSequentialFile(dscname, &file);
  if (!s.ok()) {
    return s;
  }

  bool have_log_number = false;
  bool have_prev_log_number = false;
  bool have_next_file = false;
  bool have_last_sequence = false;
  uint64_t next_file = 0;
  uint64_t last_sequence = 0;
  uint64_t log_number = 0;
  uint64_t prev_log_number = 0;
  // 相当于从empty status来构建current version.
  Builder builder(this, current_);

  {
    LogReporter reporter;
    reporter.status = &s;
    log::Reader reader(file, &reporter, true/*checksum*/, 0/*initial_offset*/);
    Slice record;
    std::string scratch;
    // 按照每条log内容读取.里面的内容都是version edit.
    while (reader.ReadRecord(&record, &scratch) && s.ok()) {
      VersionEdit edit;
      s = edit.DecodeFrom(record);
      if (s.ok()) {
        if (edit.has_comparator_ &&
            edit.comparator_ != icmp_.user_comparator()->Name()) {
          s = Status::InvalidArgument(
              edit.comparator_ + "does not match existing comparator ",
              icmp_.user_comparator()->Name());
        }
      }

      // 直接apply上去.
      if (s.ok()) {
        builder.Apply(&edit);
      }

      if (edit.has_log_number_) {
        log_number = edit.log_number_;
        have_log_number = true;
      }

      if (edit.has_prev_log_number_) {
        prev_log_number = edit.prev_log_number_;
        have_prev_log_number = true;
      }

      if (edit.has_next_file_number_) {
        next_file = edit.next_file_number_;
        have_next_file = true;
      }

      if (edit.has_last_sequence_) {
        last_sequence = edit.last_sequence_;
        have_last_sequence = true;
      }
    }
  }
  delete file;
  file = NULL;

  if (s.ok()) {
    if (!have_next_file) {
      s = Status::Corruption("no meta-nextfile entry in descriptor");
    } else if (!have_log_number) {
      s = Status::Corruption("no meta-lognumber entry in descriptor");
    } else if (!have_last_sequence) {
      s = Status::Corruption("no last-sequence-number entry in descriptor");
    }

    if (!have_prev_log_number) {
      prev_log_number = 0;
    }

    MarkFileNumberUsed(prev_log_number);
    MarkFileNumberUsed(log_number);
  }

  if (s.ok()) {
    Version* v = new Version(this);
    builder.SaveTo(v);
    // Install recovered version
    Finalize(v);
    // 增加这么一个version作为当前current version.
    AppendVersion(v);
    manifest_file_number_ = next_file;
    next_file_number_ = next_file + 1;
    last_sequence_ = last_sequence;
    log_number_ = log_number;
    prev_log_number_ = prev_log_number;
  }

  return s;
}
```

14. LogAndApply

```
Status VersionSet::LogAndApply(VersionEdit* edit, port::Mutex* mu) {
  if (edit->has_log_number_) {
    assert(edit->log_number_ >= log_number_);
    assert(edit->log_number_ < next_file_number_);
  } else {
    edit->SetLogNumber(log_number_);
  }

  if (!edit->has_prev_log_number_) {
    edit->SetPrevLogNumber(prev_log_number_);
  }

  edit->SetNextFile(next_file_number_);
  edit->SetLastSequence(last_sequence_);

  // 这个过程应该是将edit的修改放在current_里面
  // 然后将edit作用在current上面，将current_保存到v里面
  // 然后对v进行finish.
  Version* v = new Version(this);
  {
    Builder builder(this, current_);
    builder.Apply(edit);
    builder.SaveTo(v);
  }
  Finalize(v);

  // Initialize new descriptor log file if necessary by creating
  // a temporary file that contains a snapshot of the current version.
  std::string new_manifest_file;
  Status s;
  if (descriptor_log_ == NULL) {
    // No reason to unlock *mu here since we only hit this path in the
    // first call to LogAndApply (when opening the database).
    assert(descriptor_file_ == NULL);
    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_);
      // 将current做snapshot保存在descriptor log里面.
      s = WriteSnapshot(descriptor_log_);
    }
  }

  // Unlock during expensive MANIFEST log write
  {
    mu->Unlock();

    // Write new record to MANIFEST log
    // 将这个修改也保存在log里面.
    // 现在比较担心最后的结构.似乎现在snapshot以及log内容都混在一起了.
    // 之前在descriptor_log里面填写了current version内容
    // 而这里还添加了edit增量内容
    // 这个格式现在看上去还好.到时候可能需要主要看看Recover内容.
    if (s.ok()) {
      std::string record;
      edit->EncodeTo(&record);
      s = descriptor_log_->AddRecord(record);
      if (s.ok()) {
        s = descriptor_file_->Sync(); // 注意这里我们需要做sync.
      }
    }

    // If we just created a new descriptor file, install it by writing a
    // new CURRENT file that points to it.
    if (s.ok() && !new_manifest_file.empty()) {
      s = SetCurrentFile(env_, dbname_, manifest_file_number_);
    }

    mu->Lock();
  }

  // Install the new version
  // 作为当前current version存在.
  if (s.ok()) {
    AppendVersion(v);
    log_number_ = edit->log_number_;
    prev_log_number_ = edit->prev_log_number_;
  } else {
    delete v;
    if (!new_manifest_file.empty()) {
      delete descriptor_log_;
      delete descriptor_file_;
      descriptor_log_ = NULL;
      descriptor_file_ = NULL;
      env_->DeleteFile(new_manifest_file);
    }
  }

  return s;
}
```

15. GetRange

得到input files里面的最大和最小key.

```
// Stores the minimal range that covers all entries in inputs in
// *smallest, *largest.
// REQUIRES: inputs is not empty
void VersionSet::GetRange(const std::vector<FileMetaData*>& inputs,
                          InternalKey* smallest,
                          InternalKey* largest) {
  assert(!inputs.empty());
  smallest->Clear();
  largest->Clear();
  for (size_t i = 0; i < inputs.size(); i++) {
    FileMetaData* f = inputs[i];
    if (i == 0) {
      *smallest = f->smallest;
      *largest = f->largest;
    } else {
      if (icmp_.Compare(f->smallest, *smallest) < 0) {
        *smallest = f->smallest;
      }
      if (icmp_.Compare(f->largest, *largest) > 0) {
        *largest = f->largest;
      }
    }
  }
}
```

16. PickCompaction

选择一个Compaction来进行操作.对于选择Compaction应该比较具有策略性的。然后丢给DB去执行这个Compaction操作. PickCompaction和CompactRange都是compaction行为.不过PickCompaction是自动触发的，而CompactRange 是用户自己手动触发的.

```
Compaction* VersionSet::PickCompaction() {
  Compaction* c;
  int level;

  // We prefer compactions triggered by too much data in a level over  // the compactions triggered by seeks.
  // 根据当前version的统计结果判断是否需要compaction.
  const bool size_compaction = (current_->compaction_score_ >= 1);
  const bool seek_compaction = (current_->file_to_compact_ != NULL);
  if (size_compaction) {
    level = current_->compaction_level_;
    assert(level >= 0);
    assert(level+1 < config::kNumLevels);
    c = new Compaction(level);

    // Pick the first file that comes after compact_pointer_[level]
    for (size_t i = 0; i < current_->files_[level].size(); i++) {
      FileMetaData* f = current_->files_[level][i];
      if (compact_pointer_[level].empty() ||
          icmp_.Compare(f->largest.Encode(), compact_pointer_[level]) > 0) {
        c->inputs_[0].push_back(f);
        break;
      }
    }
    if (c->inputs_[0].empty()) {
      // Wrap-around to the beginning of the key space
      c->inputs_[0].push_back(current_->files_[level][0]);
    }
  } else if (seek_compaction) {
    level = current_->file_to_compact_level_;
    c = new Compaction(level);
    c->inputs_[0].push_back(current_->file_to_compact_);
  } else {
    return NULL;
  }

  // 现在已经选择好了一个文件来进行compaction.
  // 对当前的version +ref count.
  c->input_version_ = current_;
  c->input_version_->Ref();

  // Files in level 0 may overlap each other, so pick up all overlapping ones
  if (level == 0) {
    InternalKey smallest, largest;
    // 注释写的非常清楚了.
    GetRange(c->inputs_[0], &smallest, &largest);
    // Note that the next call will discard the file we placed in
    // c->inputs_[0] earlier and replace it with an overlapping set
    // which will include the picked file.
    current_->GetOverlappingInputs(0, &smallest, &largest, &c->inputs_[0]);
    assert(!c->inputs_[0].empty());
  }

  // 选择了level的文件之后需要选择level+1的文件
  // 把其他的input files字段填上.
  SetupOtherInputs(c);

  return c;
}
```

17. CompactRange

DBImpl里面提供CompactRange打包成为一个Manual Compaction的请求提交到后端.然后判断 如果

为Manual Compaction的话那么实际调用的还是VersionSet::CompactRange这个方法产生一个Compaction对象真正进行执行.

```
Compaction* VersionSet::CompactRange(
    int level,
    const InternalKey* begin,
    const InternalKey* end) {
  std::vector<FileMetaData*> inputs;
  current_->GetOverlappingInputs(level, begin, end, &inputs);
  if (inputs.empty()) {
    return NULL;
  }

  // Avoid compacting too much in one shot in case the range is large.
  // 压缩范围的话可以提交多个input files作为底层level输入.
  // 但是通过控制大小来限制输入文件多少.
  const uint64_t limit = MaxFileSizeForLevel(level);
  uint64_t total = 0;
  for (int i = 0; i < inputs.size(); i++) {
    uint64_t s = inputs[i]->file_size;
    total += s;
    if (total >= limit) {
      inputs.resize(i + 1);
      break;
    }
  }

  Compaction* c = new Compaction(level);
  c->input_version_ = current_;
  c->input_version_->Ref();
  c->inputs_[0] = inputs;
  SetupOtherInputs(c);
  return c;
}
```

18. GetRange2

得到input1和input2两个文件compaction之后的最小和最大key.

```
// Stores the minimal range that covers all entries in inputs1 and inputs2
// in *smallest, *largest.
// REQUIRES: inputs is not empty
void VersionSet::GetRange2(const std::vector<FileMetaData*>& inputs1,
                           const std::vector<FileMetaData*>& inputs2,
                           InternalKey* smallest,
                           InternalKey* largest) {
  std::vector<FileMetaData*> all = inputs1;
  all.insert(all.end(), inputs2.begin(), inputs2.end());
  GetRange(all, smallest, largest);
}
```

19. SetupOtherInputs

将compaction的另外一层内容(input files)补齐.

```
void VersionSet::SetupOtherInputs(Compaction* c) {
  const int level = c->level();
  InternalKey smallest, largest;
  GetRange(c->inputs_[0], &smallest, &largest);

  // 判断level+1里面有哪些文件是存在重叠的.
  current_->GetOverlappingInputs(level+1, &smallest, &largest, &c->inputs_[1]);

  // Get entire range covered by compaction
  // 得到整个compaction所覆盖的范围
  InternalKey all_start, all_limit;
  GetRange2(c->inputs_[0], c->inputs_[1], &all_start, &all_limit);

  // See if we can grow the number of inputs in "level" without
  // changing the number of "level+1" files we pick up.
  // 观察是否可以扩大level input files数量但是不造成level+1 input files数量变化.
  if (!c->inputs_[1].empty()) {
    std::vector<FileMetaData*> expanded0;
    current_->GetOverlappingInputs(level, &all_start, &all_limit, &expanded0);
    if (expanded0.size() > c->inputs_[0].size()) {
      InternalKey new_start, new_limit;
      GetRange(expanded0, &new_start, &new_limit);
      std::vector<FileMetaData*> expanded1;
      current_->GetOverlappingInputs(level+1, &new_start, &new_limit,
                                     &expanded1);
      if (expanded1.size() == c->inputs_[1].size()) {
        Log(options_->info_log,
            "Expanding@%d %d+%d to %d+%d\n",
            level,
            int(c->inputs_[0].size()),
            int(c->inputs_[1].size()),
            int(expanded0.size()),
            int(expanded1.size()));
        smallest = new_start;
        largest = new_limit;
        c->inputs_[0] = expanded0;
        c->inputs_[1] = expanded1;
        GetRange2(c->inputs_[0], c->inputs_[1], &all_start, &all_limit);
      }
    }
  }

  // Compute the set of grandparent files that overlap this compaction
  // (parent == level+1; grandparent == level+2)
  if (level + 2 < config::kNumLevels) {
    // 假设这两个合并之后,和level+2会有哪些文件出现重叠.
    // 保存在grandparents里面.
    current_->GetOverlappingInputs(level + 2, &all_start, &all_limit,
                                   &c->grandparents_);
  }

  if (false) {
    Log(options_->info_log, "Compacting %d '%s' .. '%s'",
        level,
        smallest.DebugString().c_str(),
        largest.DebugString().c_str());
  }

  // Update the place where we will do the next compaction for this level.
  // We update this immediately instead of waiting for the VersionEdit
  // to be applied so that if the compaction fails, we will try a different
  // key range next time.
  // 保存到本层最大key.
  compact_pointer_[level] = largest.Encode().ToString();
  // 将level的largest保存到edit里面.这个可能对于恢复起来之后可能有用.
  c->edit_.SetCompactPointer(level, largest);
}
```

20. MakeInputIterator

针对Compaction而言我们可能需要读取多个文件设计的遍历器.底层应该是MergeIterator实现.

```
Iterator* VersionSet::MakeInputIterator(Compaction* c) {
  ReadOptions options;
  options.verify_checksums = options_->paranoid_checks;
  options.fill_cache = false;

  // Level-0 files have to be merged together.  For other levels,
  // we will make a concatenating iterator per level.
  // TODO(opt): use concatenating iterator for level-0 if there is no overlap
  // 对于>0的level来说的话可以使用LevelFileNumIterator来当做这一层的iterator.
  // 而对==0的level来说的话每一个文件作为一个iterator存在.
  const int space = (c->level() == 0 ? c->inputs_[0].size() + 1 : 2);
  Iterator** list = new Iterator*[space];
  int num = 0;
  for (int which = 0; which < 2; which++) {
    if (!c->inputs_[which].empty()) {
      if (c->level() + which == 0) {
        const std::vector<FileMetaData*>& files = c->inputs_[which];
        for (size_t i = 0; i < files.size(); i++) {
          list[num++] = table_cache_->NewIterator(
              options, files[i]->number, files[i]->file_size);
        }
      } else {
        // Create concatenating iterator for the files from this level
        list[num++] = NewTwoLevelIterator(
            new Version::LevelFileNumIterator(icmp_, &c->inputs_[which]),
            &GetFileIterator, table_cache_, options);
      }
    }
  }
  assert(num <= space);
  // 然后作为一个merge iterator存在.
  Iterator* result = NewMergingIterator(&icmp_, list, num);
  delete[] list;
  return result;
}
```

10.3.13 VersionSetBuilder

db/version_set.cc VersoionSetBuilder存在作用就是为了能够高效地做version edit对象的回放. 沃恩首先看看里面可能涉及到的结构.注意这里面我们没有修改base对应的version对象,只是在SaveTo 的时候可以将这些log全部作用到base这个对象,写到新的对象上面.

```
// A helper class so we can efficiently apply a whole sequence
// of edits to a particular state without creating intermediate
// Versions that contain full copies of the intermediate state.
class VersionSet::Builder {
 private:
  // Helper to sort by v->files_[file_number].smallest
  struct BySmallestKey {
    const InternalKeyComparator* internal_comparator;

    bool operator()(FileMetaData* f1, FileMetaData* f2) const {
      int r = internal_comparator->Compare(f1->smallest, f2->smallest);
      if (r != 0) {
        return (r < 0);
      } else {
        // Break ties by file number
        return (f1->number < f2->number);
      }
    }
  };

  typedef std::set<FileMetaData*, BySmallestKey> FileSet;
  struct LevelState { // 每个level需要增加和删除的文件.
    std::set<uint64_t> deleted_files;
    FileSet* added_files;
  };

  VersionSet* vset_;
  Version* base_; // 针对某一个version进行的edit操作.
  LevelState levels_[config::kNumLevels];
};
```

1. VersionSetBuilder

   首先看看构造和析构函数.不是非常麻烦.对于析构函数的话将所有的add files refs–.

```
// Initialize a builder with the files from *base and other info from *vset
Builder(VersionSet* vset, Version* base)
    : vset_(vset),
      base_(base) {
  base_->Ref();
  BySmallestKey cmp;
  cmp.internal_comparator = &vset_->icmp_;
  for (int level = 0; level < config::kNumLevels; level++) {
    levels_[level].added_files = new FileSet(cmp);
  }
}

~Builder() {
  // 将所有的add files全部refs--.
  // 注意在SaveTo的时候会将FileMetaData这部分的ref转义到SaveTo的Version上面.
  for (int level = 0; level < config::kNumLevels; level++) {
    const FileSet* added = levels_[level].added_files;
    std::vector<FileMetaData*> to_unref;
    to_unref.reserve(added->size());
    for (FileSet::const_iterator it = added->begin();
        it != added->end(); ++it) {
      to_unref.push_back(*it);
    }
    delete added;
    for (uint32_t i = 0; i < to_unref.size(); i++) {
      FileMetaData* f = to_unref[i];
      f->refs--;
      if (f->refs <= 0) {
        delete f;
      }
    }
  }
  base_->Unref();
}
```

2. Apply

   将edit这个log记录起来.可能会被调用多次.

```
// Apply all of the edits in *edit to the current state.
void Apply(VersionEdit* edit) {
  // Update compaction pointers
  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();
  }

  // Delete files
  const VersionEdit::DeletedFileSet& del = edit->deleted_files_;
  for (VersionEdit::DeletedFileSet::const_iterator iter = del.begin();
       iter != del.end();
       ++iter) {
    const int level = iter->first;
    const uint64_t number = iter->second;
    levels_[level].deleted_files.insert(number);
  }

  // Add new files
  for (size_t i = 0; i < edit->new_files_.size(); i++) {
    const int level = edit->new_files_[i].first;
    FileMetaData* f = new FileMetaData(edit->new_files_[i].second);
    f->refs = 1;

    // We arrange to automatically compact this file after
    // a certain number of seeks.  Let's assume:
    //   (1) One seek costs 10ms
    //   (2) Writing or reading 1MB costs 10ms (100MB/s)
    //   (3) A compaction of 1MB does 25MB of IO:
    //         1MB read from this level
    //         10-12MB read from next level (boundaries may be misaligned)
    //         10-12MB written to next level
    // This implies that 25 seeks cost the same as the compaction
    // of 1MB of data.  I.e., one seek costs approximately the
    // same as the compaction of 40KB of data.  We are a little
    // conservative and allow approximately one seek for every 16KB
    // of data before triggering a compaction.
    // 设置allow seek的限制,超过这么多的限制之后的话那么作为潜在的compaction文件.
    f->allowed_seeks = (f->file_size / 16384);
    if (f->allowed_seeks < 100) f->allowed_seeks = 100;

    levels_[level].deleted_files.erase(f->number);
    levels_[level].added_files->insert(f);
  }
}
```

3. MaybeAddFile

   将f添加到新增加到对应level的文件列表尾部.需要确保没有overlap并且没有存在于标记删除文件集合中.

```
void MaybeAddFile(Version* v, int level, FileMetaData* f) {
  if (levels_[level].deleted_files.count(f->number) > 0) {
    // File is deleted: do nothing
  } else {
    std::vector<FileMetaData*>* files = &v->files_[level];
    if (level > 0 && !files->empty()) {
      // Must not overlap
      assert(vset_->icmp_.Compare((*files)[files->size()-1]->largest,
                                  f->smallest) < 0);
    }
    f->refs++;
    files->push_back(f);
  }
}
```

4. SaveTo

   当前通过edit修改后的状态保存到新的version里面.我们来仔细看看这个过程.将之前edit回放的内容保存在v这个version里面来.典型使用方法非常简单.

   - 首先构造builder对象
   - Apply使用某个edit.
   - SaveTo生成某个version.

```
// Save the current state in *v.
void SaveTo(Version* v) {
  BySmallestKey cmp;
  cmp.internal_comparator = &vset_->icmp_;
  for (int level = 0; level < config::kNumLevels; level++) {
    // Merge the set of added files with the set of pre-existing files.
    // Drop any deleted files.  Store the result in *v.
    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 = levels_[level].added_files;
    v->files_[level].reserve(base_files.size() + added->size());
    // 这里必须按照最小的key来进行排序.这里是存在顺序问题的.
    // 但是对于level>0的层来说里面的文件都是不重复的.
    for (FileSet::const_iterator added_iter = added->begin();
         added_iter != added->end();
         ++added_iter) {
      // Add all smaller files listed in base_
      for (std::vector<FileMetaData*>::const_iterator bpos
               = std::upper_bound(base_iter, base_end, *added_iter, cmp);
           base_iter != bpos;
           ++base_iter) {
        MaybeAddFile(v, level, *base_iter);
      }

      MaybeAddFile(v, level, *added_iter);
    }

    // Add remaining base files
    for (; base_iter != base_end; ++base_iter) {
      MaybeAddFile(v, level, *base_iter);
    }
  }
}
```

10.3.14 Compaction

db/version_set.h Compaction包含的是进行compaction操作的信息.针对compaction对象通常是current version 所对应的文件。我们首先来看看Compaction结构.

```
// A Compaction encapsulates information about a compaction.
class Compaction {
 public:

  int level_; // 针对哪个level进行
  uint64_t max_output_file_size_; // compaction之后单个文件的最大大小.
  Version* input_version_; // 针对哪个version进行操作.
  VersionEdit edit_; // 应该是针对version操作之后的log信息.比如删除哪些文件添加了哪些文件等.

  // Each compaction reads inputs from "level_" and "level_+1"
  std::vector<FileMetaData*> inputs_[2];      // The two sets of inputs. 要进行compaction的文件有哪些.
  //1 应该就是parent level.

  // State used to check for number of of overlapping grandparent files
  // (parent == level_ + 1, grandparent == level_ + 2)
  std::vector<FileMetaData*> grandparents_;
  size_t grandparent_index_;  // Index in grandparent_starts_
  bool seen_key_;             // Some output key has been seen
  int64_t overlapped_bytes_;  // Bytes of overlap between current output
                              // and grandparent files

  // State for implementing IsBaseLevelForKey

  // level_ptrs_ holds indices into input_version_->levels_: our state
  // is that we are positioned at one of the file ranges for each
  // higher level than the ones involved in this compaction (i.e. for
  // all L >= level_ + 2).
  size_t level_ptrs_[config::kNumLevels];
};
```

1. Compaction

   构造函数过程非常简单.析构函数的话也只是将input_version_修改引用计数。

```
Compaction::Compaction(int level)
    : level_(level),
      max_output_file_size_(MaxFileSizeForLevel(level)), // 这个函数后面分析.
      input_version_(NULL),
      grandparent_index_(0),
      seen_key_(false),
      overlapped_bytes_(0) {
```

```
  for (int i = 0; i < config::kNumLevels; i++) {
    level_ptrs_[i] = 0;
  }
}
Compaction::~Compaction() {
  if (input_version_ != NULL) {
    input_version_->Unref(); // NOTICE:这个非常重要
    // 如果这个compaction完成的话,那么version就会销毁.
  }
}
```

2. MaxFileSizeForLevel

   每一层单个文件大小最大是多少.现在实现非常简单是一个固定值.

```
static const int kTargetFileSize = 2 * 1048576; // 2MB.
static uint64_t MaxFileSizeForLevel(int level) {
  return kTargetFileSize;  // We could vary per level to reduce number of files?
}
```

3. MaxBytesForLevel

   每一层文件所占用的字节数上限是多少.level0,1是10MB.以后每上一层*10.

```
static double MaxBytesForLevel(int level) {
  // Note: the result for level zero is not really used since we set
  // the level-0 compaction threshold based on number of files.
  double result = 10 * 1048576.0;  // Result for both level-0 and level-1
  while (level > 1) {
    result *= 10;
    level--;
  }
  return result;
}
```

4. IsTrivialMove

   这个compaction是否为简单的移动.需要满足3个条件

```
bool Compaction::IsTrivialMove() const {
  // Avoid a move if there is lots of overlapping grandparent data.
  // Otherwise, the move could create a parent file that will require
  // a very expensive merge later on.
  return (num_input_files(0) == 1 && // level 1个文件
          num_input_files(1) == 0 && // level+1 没有文件
          // grandparent level存在overlap的文件大小没有过大.
          TotalFileSize(grandparents_) <= kMaxGrandParentOverlapBytes);
}
```

5. AddInputDeletions

   通知这次操作做了文件的删除.将这些可以删除的文件填写到edit里面.做好compaction之后的话，那么inputs_里面的文件基本上就没有用途就可以被删除了。

```
void Compaction::AddInputDeletions(VersionEdit* edit) {
  for (int which = 0; which < 2; which++) {
    for (size_t i = 0; i < inputs_[which].size(); i++) {
      edit->DeleteFile(level_ + which, inputs_[which][i]->number);
    }
  }
}
```

6. IsBaseLevelForKey

   在db_impl.cc里面的DoCompactionWork里面调用了.具体策略实现没有太仔细分析.判断某个key在高层是否还存在.

```
bool Compaction::IsBaseLevelForKey(const Slice& user_key) {
  // Maybe use binary search to find right entry instead of linear search?
  const Comparator* user_cmp = input_version_->vset_->icmp_.user_comparator();
  for (int lvl = level_ + 2; lvl < config::kNumLevels; lvl++) {
    const std::vector<FileMetaData*>& files = input_version_->files_[lvl];
    for (; level_ptrs_[lvl] < files.size(); ) {
      FileMetaData* f = files[level_ptrs_[lvl]];
      if (user_cmp->Compare(user_key, f->largest.user_key()) <= 0) {
        // We've advanced far enough
        if (user_cmp->Compare(user_key, f->smallest.user_key()) >= 0) {
          // Key falls in this file's range, so definitely not base level
          return false;
        }
        break;
      }
      level_ptrs_[lvl]++;
    }
  }
  return true;
}
```

7. ShouldStopBefore

   在db_impl.cc里面的DoCompactionWork里面调用了.作用是判断这个key的话是否应该停止然后 重新开辟一个新的文件用于compaction的输出.这是一个策略,大致应该是希望这个key如果 和grandparent level重合不要过多.更具体的策略方面不太清楚.

```
bool Compaction::ShouldStopBefore(const Slice& internal_key) {
  // Scan to find earliest grandparent file that contains key.
  const InternalKeyComparator* icmp = &input_version_->vset_->icmp_;
  while (grandparent_index_ < grandparents_.size() &&
      icmp->Compare(internal_key,
                    grandparents_[grandparent_index_]->largest.Encode()) > 0) {
    if (seen_key_) {
      overlapped_bytes_ += grandparents_[grandparent_index_]->file_size;
    }
    grandparent_index_++;
  }
  seen_key_ = true;

  if (overlapped_bytes_ > kMaxGrandParentOverlapBytes) {
    // Too much overlap for current output; start new output
    overlapped_bytes_ = 0;
    return true;
  } else {
    return false;
  }
}
```

8. ReleaseInputs

```
void Compaction::ReleaseInputs() {
  if (input_version_ != NULL) {
    input_version_->Unref();
    input_version_ = NULL;
  }
}
```

10.3.15 CompactionState

db/db_impl.cc CompactionState里面记录的是在这次Compaction时候保存的状态.Compaction只是 保存了本次compaction操作所需要的信息，但是没有保存过程状态.

```
struct DBImpl::CompactionState {
  Compaction* const compaction; // 本次compaction所需要进行的操作.

  // Sequence numbers < smallest_snapshot are not significant since we
  // will never have to service a snapshot below smallest_snapshot.
  // Therefore if we have seen a sequence number S <= smallest_snapshot,
  // we can drop all entries for the same key with sequence numbers < S.
  SequenceNumber smallest_snapshot; // 当前有效的sequence number是多少.

  // Files produced by compaction
  struct Output {
    uint64_t number; // 输出文件number.
    uint64_t file_size; // 输出文件大小
    InternalKey smallest, largest; // 输出文件size.
  };
  std::vector<Output> outputs;

  // State kept for output being generated
  WritableFile* outfile; // 当前输出文件以及table_builder.
  TableBuilder* builder;

  uint64_t total_bytes;

  Output* current_output() { return &outputs[outputs.size()-1]; }

  explicit CompactionState(Compaction* c)
      : compaction(c),
        outfile(NULL),
        builder(NULL),
        total_bytes(0) {
  }
};
```

10.3.16 VersionEdit

db/version_edit.h 阅读完DBImpl::NewDB的话就会发现VersionEdit里面存储的是整个数据库的元信息. 元信息序列化之后作为log写入到Manifest文件里面去。CURRENT文件里面存放的就是当前Manifest文件. 我们首先看看VersionEdit的结构以及里面的字段。个人觉得VersionEdit本身是非常简单的，但是需要了解各个字段含义。

:阅读完Compaction之后，觉得这个结构更多的含义应该是针对Version进行Compaction这个过程，对于Version造成了哪些变动，比如增加或者是删除啦哪些文件，下一个file number以及sequence是什么。将这些 内容记录到log(manifest)里面这样启动的时候就可以进行恢复。

```
class VersionEdit {
  // private:
 public:
  friend class VersionSet;

  typedef std::set< std::pair<int, uint64_t> > DeletedFileSet;

  std::string comparator_; // 比较器的名字.这样可以防止我comparator的变动造成db的逻辑损坏.
  uint64_t log_number_; //
  uint64_t prev_log_number_;
  uint64_t next_file_number_;
  SequenceNumber last_sequence_; // 上次操作完成之后的序列号.
  bool has_comparator_;
  bool has_log_number_;
  bool has_prev_log_number_;
  bool has_next_file_number_;
  bool has_last_sequence_;

  std::vector< std::pair<int, InternalKey> > compact_pointers_;
  DeletedFileSet deleted_files_;
  std::vector< std::pair<int, FileMetaData> > new_files_;
};
```

1. SetCompactPoint

   ```
   void SetCompactPointer(int level, const InternalKey& key) {
     compact_pointers_.push_back(std::make_pair(level, key));
   }
   ```

2. AddFile

   相当于告诉db系统新增加了这么一个文件以及这个文件里面的信息是什么.这样可以方便db操作.

   ```
   void AddFile(int level, uint64_t file,
                uint64_t file_size,
                const InternalKey& smallest,
                const InternalKey& largest) {
     FileMetaData f;
     f.number = file;
     f.file_size = file_size;
     f.smallest = smallest;
     f.largest = largest;
     new_files_.push_back(std::make_pair(level, f));
   }
   ```

3. DeleteFile

   告诉db系统删除了这个文件.删除并没有操作new_files数组.而是放进了一个set.

```
// Delete the specified "file" from the specified "level".
void DeleteFile(int level, uint64_t file) {
  deleted_files_.insert(std::make_pair(level, file));
}
```

10.3.17 ByteWiseComparatorImpl

util/comparator.cc 对于Compare实现使用slice对象的compare实现。我们着重看看上面两个接口实现

```
virtual void FindShortestSeparator(
    std::string* start,
    const Slice& limit) const {
  // Find length of common prefix
  size_t min_length = std::min(start->size(), limit.size());
  size_t diff_index = 0;
  while ((diff_index < min_length) &&
         ((*start)[diff_index] == limit[diff_index])) {
    diff_index++;
  }

  // 实现上仅仅是对于第一个不同的字符尝试+1
  if (diff_index >= min_length) {
    // Do not shorten if one string is a prefix of the other
  } else {
    uint8_t diff_byte = static_cast<uint8_t>((*start)[diff_index]);
    if (diff_byte < static_cast<uint8_t>(0xff) &&
        diff_byte + 1 < static_cast<uint8_t>(limit[diff_index])) {
      (*start)[diff_index]++;  // +1
      start->resize(diff_index + 1); // 并且修改长度
      assert(Compare(*start, limit) < 0);
    }
  }
}

virtual void FindShortSuccessor(std::string* key) const {
  // Find first character that can be incremented
  size_t n = key->size();
  for (size_t i = 0; i < n; i++) {
    const uint8_t byte = (*key)[i];
    if (byte != static_cast<uint8_t>(0xff)) {
      (*key)[i] = byte + 1; // +1
      key->resize(i+1); // 修改长度
      return;
    }
  }
  // *key is a run of 0xffs.  Leave it alone.
}
```

还算是比较直白吧.

10.3.18 SnapshotImpl

db/snapshot.h Snapshot实现非常简单，就是一个双向链表的节点，然后挂在一个双向链表上面。 每一个Snapshot实现都附带一个seq number.对于Snapshot最重要的应该是在上面的操作吧. 我们可以猜想对于每次插入的key都会带上一个seq number.这样如果对snapshot操作的话读取的话， 那么只需要读取seq number以下的内容即可了。

```
// Snapshots are kept in a doubly-linked list in the DB.
// Each SnapshotImpl corresponds to a particular sequence number.
class SnapshotImpl : public Snapshot {
 public:
  SequenceNumber number_;  // const after creation

 private:
  friend class SnapshotList;

  // SnapshotImpl is kept in a doubly-linked circular list
  SnapshotImpl* prev_;
  SnapshotImpl* next_;

  SnapshotList* list_;                 // just for sanity checks
};

class SnapshotList {
 public:
  SnapshotList() {
    list_.prev_ = &list_;
    list_.next_ = &list_;
  }

  bool empty() const { return list_.next_ == &list_; }
  SnapshotImpl* oldest() const { assert(!empty()); return list_.next_; }
  SnapshotImpl* newest() const { assert(!empty()); return list_.prev_; }

  const SnapshotImpl* New(SequenceNumber seq) { // 分配一个snapshot实例
    SnapshotImpl* s = new SnapshotImpl;
    s->number_ = seq;
    s->list_ = this;
    s->next_ = &list_;
    s->prev_ = list_.prev_;
    s->prev_->next_ = s;
    s->next_->prev_ = s;
    return s;
  }

  void Delete(const SnapshotImpl* s) { // 将这个snapshot从链中删除
    assert(s->list_ == this);
    s->prev_->next_ = s->next_;
    s->next_->prev_ = s->prev_;
    delete s;
  }

 private:
  // Dummy head of doubly-linked list of snapshots
  SnapshotImpl list_;
};
```

这里SequenceNumber的实现是一个在db/dbformat.h里面定义了

```
typedef uint64_t SequenceNumber;
```

10.4 Posix

10.4.1 PosixEnv

util/env_posix.cc PosixEnv是Env接口的实现。PosixEnv的大部分实现都相当直接.不过这里还是把代码贴上了，主要是方便在一些细小的地方进行注记。

```
class PosixEnv : public Env {
 public:
  PosixEnv();
  virtual ~PosixEnv() {
    fprintf(stderr, "Destroying Env::Default()\n");
    exit(1);
  }

  virtual Status NewSequentialFile(const std::string& fname,
                                   SequentialFile** result) {
    FILE* f = fopen(fname.c_str(), "r");
    if (f == NULL) {
      *result = NULL;
      return IOError(fname, errno);
    } else {
      *result = new PosixSequentialFile(fname, f);
      return Status::OK();
    }
  }

  virtual Status NewRandomAccessFile(const std::string& fname,
                                     RandomAccessFile** result) {
    int fd = open(fname.c_str(), O_RDONLY);
    if (fd < 0) {
      *result = NULL;
      return IOError(fname, errno);
    }
    *result = new PosixRandomAccessFile(fname, fd);
    return Status::OK();
  }

  virtual Status NewWritableFile(const std::string& fname,
                                 WritableFile** result) {
```

```
    Status s;
    const int fd = open(fname.c_str(), O_CREAT | O_RDWR | O_TRUNC, 0644);
    if (fd < 0) {
      *result = NULL;
      s = IOError(fname, errno);
    } else {
      *result = new PosixMmapFile(fname, fd, page_size_);
    }
    return s;
  }

  virtual bool FileExists(const std::string& fname) {
    return access(fname.c_str(), F_OK) == 0;
  }

  virtual Status GetChildren(const std::string& dir,
                             std::vector<std::string>* result) {
    result->clear();
    DIR* d = opendir(dir.c_str());
    if (d == NULL) {
      return IOError(dir, errno);
    }
    struct dirent* entry;
    while ((entry = readdir(d)) != NULL) {
      result->push_back(entry->d_name);
    }
    closedir(d);
    return Status::OK();
  }

  virtual Status DeleteFile(const std::string& fname) {
    Status result;
    if (unlink(fname.c_str()) != 0) {
      result = IOError(fname, errno);
    }
    return result;
  };

  virtual Status CreateDir(const std::string& name) {
    Status result;
    if (mkdir(name.c_str(), 0755) != 0) {
      result = IOError(name, errno);
    }
    return result;
  };

  virtual Status DeleteDir(const std::string& name) {
    Status result;
    if (rmdir(name.c_str()) != 0) {
      result = IOError(name, errno);
    }
    return result;
  };

  virtual Status GetFileSize(const std::string& fname, uint64_t* size) {
    Status s;
    struct stat sbuf;
    if (stat(fname.c_str(), &sbuf) != 0) {
      *size = 0;
      s = IOError(fname, errno);
    } else {
      *size = sbuf.st_size;
    }
    return s;
  }

  virtual Status RenameFile(const std::string& src, const std::string& target) {
    Status result;
    if (rename(src.c_str(), target.c_str()) != 0) {
      result = IOError(src, errno);
    }
    return result;
  }

  virtual Status LockFile(const std::string& fname, FileLock** lock) {
    *lock = NULL;
    Status result;
    int fd = open(fname.c_str(), O_RDWR | O_CREAT, 0644); // 创建一个文件表示锁住.
    if (fd < 0) {
      result = IOError(fname, errno);
    } else if (LockOrUnlock(fd, true) == -1) { // 这个会在后面仔细看看.true表示lock.
      result = IOError("lock " + fname, errno);
      close(fd);
    } else {
      PosixFileLock* my_lock = new PosixFileLock; // 创建一个PosixFileLock对象
      my_lock->fd_ = fd; // 标记fd.
      *lock = my_lock;
    }
    return result;
  }

  virtual Status UnlockFile(FileLock* lock) {
    PosixFileLock* my_lock = reinterpret_cast<PosixFileLock*>(lock);
    Status result;
    if (LockOrUnlock(my_lock->fd_, false) == -1) { // false表示unlock
      result = IOError("unlock", errno);
    }
    close(my_lock->fd_); // 关闭对应的文件描述符.
    delete my_lock;
    return result;
  }

  virtual void Schedule(void (*function)(void*), void* arg);

  virtual void StartThread(void (*function)(void* arg), void* arg);

  virtual Status GetTestDirectory(std::string* result) {
    const char* env = getenv("TEST_TMPDIR");
    if (env && env[0] != '\0') {
      *result = env;
    } else {
      char buf[100];
      snprintf(buf, sizeof(buf), "/tmp/leveldbtest-%d", int(geteuid()));
      *result = buf;
    }
    // Directory may already exist
    CreateDir(*result);
    return Status::OK();
  }

  static uint64_t gettid() {
    pthread_t tid = pthread_self();
    uint64_t thread_id = 0;
    memcpy(&thread_id, &tid, std::min(sizeof(thread_id), sizeof(tid)));
    return thread_id;
  }

  virtual Status NewLogger(const std::string& fname, Logger** result) {
    FILE* f = fopen(fname.c_str(), "w");
    if (f == NULL) {
      *result = NULL;
      return IOError(fname, errno);
    } else {
      *result = new PosixLogger(f, &PosixEnv::gettid); // Logger.后面会仔细分析.
      return Status::OK();
    }
  }

  virtual uint64_t NowMicros() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return static_cast<uint64_t>(tv.tv_sec) * 1000000 + tv.tv_usec;
  }

  virtual void SleepForMicroseconds(int micros) {
    usleep(micros);
  }

 private:
  void PthreadCall(const char* label, int result) {
    if (result != 0) {
      fprintf(stderr, "pthread %s: %s\n", label, strerror(result));
      exit(1);
    }
  }

  // 猜想DB后面有一个线程完成一些事情通过queue来进行通知.
  // BGThread() is the body of the background thread
  void BGThread();
  static void* BGThreadWrapper(void* arg) {
    reinterpret_cast<PosixEnv*>(arg)->BGThread();
    return NULL;
  }

  size_t page_size_;
  pthread_mutex_t mu_; // queue的锁和条件变量
  pthread_cond_t bgsignal_;
  pthread_t bgthread_; // background thread identity.
  bool started_bgthread_; // bg thread是否启动

  // Entry per Schedule() call
  // background item还是比较通用的,function+arg.
  struct BGItem { void* arg; void (*function)(void*); };
  typedef std::deque<BGItem> BGQueue; // Background queue队列.
  BGQueue queue_;
};
```

1. LockOrUnlock

   将fd设置成写锁.lock表示是加锁还是解锁.这里flock不需要保留空间只是传引用即可,对于内部调用仅仅是修改了fd的一个标记吧.

```
static int LockOrUnlock(int fd, bool lock) {
  errno = 0;
  struct flock f;
  memset(&f, 0, sizeof(f));
  f.l_type = (lock ? F_WRLCK : F_UNLCK);
  f.l_whence = SEEK_SET;
  f.l_start = 0;
  f.l_len = 0;        // Lock/unlock entire file
  return fcntl(fd, F_SETLK, &f);
}
```

2. Schedule

   Schedule语义就是将一个function+arg丢到background里面运行.background线程是惰性初始化的.注意background只有一个执行线程，需要考虑工作是否会阻塞住。

```
void PosixEnv::Schedule(void (*function)(void*), void* arg) {
  PthreadCall("lock", pthread_mutex_lock(&mu_));

  // Start background thread if necessary
  if (!started_bgthread_) {
    started_bgthread_ = true;
    PthreadCall( // 创建线程.
        "create thread",
        pthread_create(&bgthread_, NULL,  &PosixEnv::BGThreadWrapper, this));
  }

  // If the queue is currently empty, the background thread may currently be
  // waiting.
  if (queue_.empty()) {
    PthreadCall("signal", pthread_cond_signal(&bgsignal_)); // 队列通知
  }

  // Add to priority queue
  queue_.push_back(BGItem()); // 在队列内部加入对象.
  queue_.back().function = function;
  queue_.back().arg = arg;

  PthreadCall("unlock", pthread_mutex_unlock(&mu_));
}
```

   其中BGThreadWrapper非常简单调用BGThread函数执行

```
  static void* BGThreadWrapper(void* arg) {
    reinterpret_cast<PosixEnv*>(arg)->BGThread();
    return NULL;
  }
void PosixEnv::BGThread() {
  while (true) {
    // Wait until there is an item that is ready to run
    PthreadCall("lock", pthread_mutex_lock(&mu_));
    while (queue_.empty()) {
      PthreadCall("wait", pthread_cond_wait(&bgsignal_, &mu_));
    }

    void (*function)(void*) = queue_.front().function; // 在线程内部执行函数.
    void* arg = queue_.front().arg;
    queue_.pop_front();

    PthreadCall("unlock", pthread_mutex_unlock(&mu_));
    (*function)(arg);
  }
}
```

3. StartThread

   启动一个线程.但是似乎有限制没有能力进行join.

```
void PosixEnv::StartThread(void (*function)(void* arg), void* arg) {
  pthread_t t;
  StartThreadState* state = new StartThreadState;
  state->user_function = function;
  state->arg = arg;
  PthreadCall("start thread",
              pthread_create(&t, NULL,  &StartThreadWrapper, state));
}
```

4. Default

   Default是为了获得Env的一个实例.为了防止多个线程产生多个Env实例.使用pthread_once来产生.

```
static pthread_once_t once = PTHREAD_ONCE_INIT;
static Env* default_env;
static void InitDefaultEnv() { default_env = new PosixEnv; }

Env* Env::Default() {
  pthread_once(&once, InitDefaultEnv);
  return default_env;
}
```

10.4.2 PosixFileLock

util/env_posix.cc PosixFilLock是FileLock的实现。基本上PosixFileLock没有任何内容，里面只需要维护fd即可。然后在LockOrUnlock里面操作fd即可以加锁解锁

```
class PosixFileLock : public FileLock {
 public:
  int fd_;
};
```

10.4.3 PosixLogger

util/posix_logger.h PosixLogger是Logger的实现。我们主要看看logger打印了哪些信息，并且稍微看看如果我们写logger的话大致应该需要考虑什么问题 PosixLogger持有了FILE*对象以及gettid的函数指针.只有一个方法Logv

```
virtual void Logv(const char* format, va_list ap) {
  const uint64_t thread_id = (*gettid_)();

  // We try twice: the first time with a fixed-size stack allocated buffer,
  // and the second time with a much larger dynamically allocated buffer.
  // 这个地方值得学习一下.第一次使用stack分配内存，第二次使用heap分配内存.
  char buffer[500];
  for (int iter = 0; iter < 2; iter++) {
    char* base;
    int bufsize;
    if (iter == 0) {
      bufsize = sizeof(buffer);
      base = buffer;
    } else {
      bufsize = 30000;
      base = new char[bufsize];
    }
    char* p = base;
    char* limit = base + bufsize;

    struct timeval now_tv;
    gettimeofday(&now_tv, NULL);
    const time_t seconds = now_tv.tv_sec;
    struct tm t;
    localtime_r(&seconds, &t); // 本地时间+microseconds 线程号 自定义格式.
    p += snprintf(p, limit - p,
                  "%04d/%02d/%02d-%02d:%02d:%02d.%06d %llx ",
                  t.tm_year + 1900,
                  t.tm_mon + 1,
                  t.tm_mday,
                  t.tm_hour,
                  t.tm_min,
                  t.tm_sec,
                  static_cast<int>(now_tv.tv_usec),
                   static_cast<long long unsigned int>(thread_id));

    // Print the message
    if (p < limit) {
      va_list backup_ap;
      va_copy(backup_ap, ap);
      p += vsnprintf(p, limit - p, format, backup_ap);
      va_end(backup_ap);
    }

    // Truncate to available space if necessary
    if (p >= limit) { // 如果超过的话并且是stack分配那么重新一轮.
      if (iter == 0) {
        continue;       // Try again with larger buffer
      } else {
        p = limit - 1; // 第二轮的话进行截断.
      }
    }

    // Add newline if necessary
    if (p == base || p[-1] != '\n') { // 确保存在换行.
      *p++ = '\n';
    }

    assert(p <= limit);
    fwrite(base, 1, p - base, file_); // 写入文件并且这里注意需要flush.
    fflush(file_);
    if (base != buffer) {
      delete[] base;
    }
    break;
  }
}
```

10.4.4 PosixRandomAccessFile

util/env_posix.cc PosixRandomAccessFile是RandomAccessFile的实现。对于随机读取的话使用pread应该就可以了.

```
virtual Status Read(uint64_t offset, size_t n, Slice* result,
                    char* scratch) const {
  Status s;
  ssize_t r = pread(fd_, scratch, n, static_cast<off_t>(offset));
  *result = Slice(scratch, (r < 0) ? 0 : r);
  if (r < 0) {
    // An error: return a non-ok status
    s = IOError(filename_, errno);
  }
  return s;
}
```

10.4.5 PosixSequentialFile

util/env_posix.cc PosixSequentialFile是SequentialFile的实现。内部实现的持有FILE*指针.猜想对于顺序访问而言的话，缓存还是会非常有效果的

```
virtual Status Read(size_t n, Slice* result, char* scratch) {
  Status s;
  size_t r = fread_unlocked(scratch, 1, n, file_);
  *result = Slice(scratch, r);
  if (r < n) {
    if (feof(file_)) {
      // We leave status as ok if we hit the end of the file
    } else {
      // A partial read with an error: return a non-ok status
      s = IOError(filename_, errno);
    }
  }
  return s;
}

virtual Status Skip(uint64_t n) {
  if (fseek(file_, n, SEEK_CUR)) { // 从当前跳过n个字节.
    return IOError(filename_, errno);
  }
  return Status::OK();
}
```

fread_unlocked表示内部不会存在多个线程来读取这个文件.可以在一定程度上提高性能.

10.4.6 PosixMmapFile

util/env_posix.cc PosixMmapFile是为了实现WritableFile的.内部使用mmap来实现的文件追加写和同步方式.还是比较精巧的，值得仔细看看

```
PosixMmapFile(const std::string& fname, int fd, size_t page_size)
    : filename_(fname),
      fd_(fd),
      page_size_(page_size),
      map_size_(Roundup(65536, page_size)), // 65536 roundup to pagesize作为一个mmap的大小.
      base_(NULL), // mmap出来的base,dst(当前可用地址),limit(结束地址)
      limit_(NULL),
      dst_(NULL),
      last_sync_(NULL), // 下载进行sync的起始地址
      file_offset_(0), //
      pending_sync_(false) { // 是否存在pending sync磁盘内容.否则需要调用fdatasync
  assert((page_size & (page_size - 1)) == 0);
}


~PosixMmapFile() {
  if (fd_ >= 0) {
    PosixMmapFile::Close();
  }
}

virtual Status Append(const Slice& data) { // 尝试增加数据.
  const char* src = data.data();
  size_t left = data.size();
  while (left > 0) { // 如果还有部分数据没有写入.
    assert(base_ <= dst_);
    assert(dst_ <= limit_);
    size_t avail = limit_ - dst_;
    if (avail == 0) {
      if (!UnmapCurrentRegion() || // unmap当前的区域并且分配新的区域.
          !MapNewRegion()) {
        return IOError(filename_, errno);
      }
    }

    size_t n = (left <= avail) ? left : avail;
    memcpy(dst_, src, n);
    dst_ += n;
    src += n;
    left -= n;
  }
  return Status::OK();
}

virtual Status Close() {
  Status s;
  size_t unused = limit_ - dst_;
  if (!UnmapCurrentRegion()) { // 将剩余的部分unmap掉
    s = IOError(filename_, errno);
  } else if (unused > 0) {
    // Trim the extra space at the end of the file
    if (ftruncate(fd_, file_offset_ - unused) < 0) { // 同时需要truncate多mmap的部分.
      s = IOError(filename_, errno);
    }
  }

  if (close(fd_) < 0) {
    if (s.ok()) {
      s = IOError(filename_, errno);
    }
  }

  fd_ = -1;
  base_ = NULL;
  limit_ = NULL;
  return s;
}

virtual Status Flush() {
  return Status::OK();
}
```

```
virtual Status Sync() {
  Status s;

  if (pending_sync_) { // 如果存在需要sync的部分.
    // Some unmapped data was not synced
    pending_sync_ = false;
    if (fdatasync(fd_) < 0) { // 将这个部分data sync.
      s = IOError(filename_, errno);
    }
  }

  if (dst_ > last_sync_) { // 将内存部分进行sync.
    // Find the beginnings of the pages that contain the first and last
    // bytes to be synced.
    size_t p1 = TruncateToPageBoundary(last_sync_ - base_); // 可能会多msync部分但是没有关系.
    size_t p2 = TruncateToPageBoundary(dst_ - base_ - 1);
    last_sync_ = dst_;
    if (msync(base_ + p1, p2 - p1 + page_size_, MS_SYNC) < 0) {
      s = IOError(filename_, errno);
    }
  }

  return s;
}
```

1. Roundup

   Roundup非常简单.使用时候就是希望mapsize作为pagesize的整数倍存在.

```
// Roundup x to a multiple of y
static size_t Roundup(size_t x, size_t y) {
  return ((x + y - 1) / y) * y;
}
```

2. TruncateToPageBoundar

   我们需要记住这个函数的语境，是将s的长度截断到pagesize整数倍，这样可以让msync刷新确保正确。

```
size_t TruncateToPageBoundary(size_t s) {
  s -= (s & (page_size_ - 1));
  assert((s % page_size_) == 0);
  return s;
}
```

3. UnmapCurrentRegion

   解除当前的内存映射。

```
bool UnmapCurrentRegion() {
  bool result = true;
  if (base_ != NULL) {
    if (last_sync_ < limit_) { // 如果上一次没内存映射完成的话.
      // Defer syncing this data until next Sync() call, if any
      pending_sync_ = true;
    }
    if (munmap(base_, limit_ - base_) != 0) { // 解除内存映射.
      result = false;
    }
    file_offset_ += limit_ - base_; // 当前fileoffset有多少.
    base_ = NULL;
    limit_ = NULL;
    last_sync_ = NULL;
    dst_ = NULL;

    // Increase the amount we map the next time, but capped at 1MB
    if (map_size_ < (1<<20)) { // 下次mapsize需要翻倍.
      map_size_ *= 2;
    }
  }
  return result;
}
```

4. MapNewRegion

   使用mmap分配一块新的内存然后用于后续的写文件

```
bool MapNewRegion() {
  assert(base_ == NULL);
  if (ftruncate(fd_, file_offset_ + map_size_) < 0) { // 因为这次我们打算只写file_offset+map_size这么大
    // 所以多余的部分我们可以截断.
    return false;
  }
  void* ptr = mmap(NULL, map_size_, PROT_READ | PROT_WRITE, MAP_SHARED,
                   fd_, file_offset_);
  if (ptr == MAP_FAILED) {
    return false;
  }
  base_ = reinterpret_cast<char*>(ptr);
  limit_ = base_ + map_size_;
  dst_ = base_;
  last_sync_ = base_;
  return true;
}
```

10.5 Cache

10.5.1 ShardedLRUCache

util/cache.cc 所谓Shard意思非常简单，就是将所有的请求进行load-balance.首先看看字段.里面使用到了LRUCache.LRUCache后面会讲到

```
static const int kNumShardBits = 4;
static const int kNumShards = 1 << kNumShardBits;

class ShardedLRUCache : public Cache {
 private:
  LRUCache shard_[kNumShards];
  port::Mutex id_mutex_;
  uint64_t last_id_;
};
```

可以看到有16个slot.对于load-balance策略也非常简单.取hash结果的高位.

```
static uint32_t Shard(uint32_t hash) {
  return hash >> (32 - kNumShardBits);
}
```

对于Shard而言的话，capacity也需要进行均分:)

```
explicit ShardedLRUCache(size_t capacity)
    : last_id_(0) {
  const size_t per_shard = (capacity + (kNumShards - 1)) / kNumShards;
  for (int s = 0; s < kNumShards; s++) {
    shard_[s].SetCapacity(per_shard);
  }
}
```

10.5.2 LRUCache

util/cache.cc LRUCache实现就是按照LRU来实现的.LRUCache每一个item是一个LRUHandle，而LRUHandle管理是放在HandleTable里面管理的。

```
// A single shard of sharded cache.
class LRUCache {
 public:
  LRUCache();
  ~LRUCache();

  // Separate from constructor so caller can easily make an array of LRUCache
  void SetCapacity(size_t capacity) { capacity_ = capacity; }

  // Like Cache methods, but with an extra "hash" parameter.
  Cache::Handle* Insert(const Slice& key, uint32_t hash,
                        void* value, size_t charge,
                        void (*deleter)(const Slice& key, void* value));
  Cache::Handle* Lookup(const Slice& key, uint32_t hash);
  void Release(Cache::Handle* handle);
  void Erase(const Slice& key, uint32_t hash);

 private:
  void LRU_Remove(LRUHandle* e);
  void LRU_Append(LRUHandle* e);
  void Unref(LRUHandle* e);

  // Initialized before use.
  size_t capacity_; // usage_的最大值.

  // mutex_ protects the following state.
  port::Mutex mutex_; // 对于多线程有效
  size_t usage_; // 当前使用大小.+=Insert参数的charge.
  uint64_t last_id_; // 在ShardedLRUCache里面使用

  // Dummy head of LRU list.
  // lru.prev is newest entry, lru.next is oldest entry.
  LRUHandle lru_; // LRU链.

  HandleTable table_; // LRUHandle的管理对象表(hashtable).
};
```

对于LRU_Remove/LRU_Append非常简单.这里稍微着重看看Insert/Loopup/Release/Erase的实现.

1. Insert

   Insert的工作就是创建一个LRUHandle对象加入hashtable以及LRU队列里面.注意这里的引用计数

```
Cache::Handle* LRUCache::Insert(
    const Slice& key, uint32_t hash, void* value, size_t charge,
    void (*deleter)(const Slice& key, void* value)) {
  MutexLock l(&mutex_);

  LRUHandle* e = reinterpret_cast<LRUHandle*>(
      malloc(sizeof(LRUHandle)-1 + key.size()));
  e->value = value; // value.
  e->deleter = deleter; // 删除函数
  e->charge = charge; // 占用大小
  e->key_length = key.size(); // key
  e->hash = hash; // hash
  e->refs = 2;  // One from LRUCache, one for the returned handle
  memcpy(e->key_data, key.data(), key.size());
  LRU_Append(e); // 加入LRU队列
  usage_ += charge; // 当前占用量

  LRUHandle* old = table_.Insert(e);
  if (old != NULL) { // 淘汰出的内容.
    LRU_Remove(old);
    Unref(old);
  }

  while (usage_ > capacity_ && lru_.next != &lru_) {
    LRUHandle* old = lru_.next;
    LRU_Remove(old);
    table_.Remove(old->key(), old->hash);
    Unref(old);
  }

  return reinterpret_cast<Cache::Handle*>(e);
}
```

2. Lookup

```
Cache::Handle* LRUCache::Lookup(const Slice& key, uint32_t hash) {
  MutexLock l(&mutex_);
  LRUHandle* e = table_.Lookup(key, hash); // 找到对象
  if (e != NULL) {
    e->refs++; // 需要修改引用计数.
    LRU_Remove(e); // 修改一下LRU状态.
    LRU_Append(e);
  }
  return reinterpret_cast<Cache::Handle*>(e);
}
```

3. Release

```
void LRUCache::Release(Cache::Handle* handle) {
  MutexLock l(&mutex_);
  Unref(reinterpret_cast<LRUHandle*>(handle)); // 减少引用计数
}
```

4. Erase

```
void LRUCache::Erase(const Slice& key, uint32_t hash) {
  MutexLock l(&mutex_);
  LRUHandle* e = table_.Remove(key, hash); // 从table里面remove出来
  if (e != NULL) {
    LRU_Remove(e); // 从LRU队列里面移除出来
    Unref(e); // --引用计数.
  }
}
```

10.5.3 LRUHandle

util/cache.cc LRUHandle是LRUCache的管理对象，我们主要就是为了看看里面的结构.LRUHandle一方面在LRU队列里面需要维护指针， 一方面在hashtable(HandleTable下面会讲到)需要维护指针.同时存在于LRU队列和HashTable里面.但是引用计数以LRU队列为准

```
// An entry is a variable length heap-allocated structure.  Entries
// are kept in a circular doubly linked list ordered by access time.
struct LRUHandle {
  void* value;
  void (*deleter)(const Slice&, void* value);
  LRUHandle* next_hash; // 在hash table的单向链表.
  LRUHandle* next; // LRU队列所需要的指针.
  LRUHandle* prev;
  size_t charge;      // TODO(opt): Only allow uint32_t? // 占用cache大小
  size_t key_length;
  uint32_t refs; // 引用计数.
  uint32_t hash;      // Hash of key(); used for fast sharding and comparisons // 可能需要进行rehash.
  char key_data[1];   // Beginning of key

  Slice key() const {
    // For cheaper lookups, we allow a temporary Handle object
    // to store a pointer to a key in "value".
    if (next == this) {
      return *(reinterpret_cast<Slice*>(value));
    } else {
      return Slice(key_data, key_length);
    }
  }
};
```

10.5.4 HandleTable

util/cache.cc HandleTable实现非常简单，就是一个可扩展的hashtable.没有必要值得仔细分析。HandleTable主要就是用来管理 LRUHandle所管理的cache对象的。触发resize的操作是element >= length.这里可以看看length的分配方式

```
uint32_t new_length = 4;
while (new_length < elems_) {
  new_length *= 2;
}
```

length肯定是2^n.计算slot的方式就是

```
LRUHandle** ptr = &list_[hash & (length_ - 1)];
```

10.6 Batch

10.6.1 WriteBatch

db/write_batch.cc db/write_batch_internal.h WriteBatch可以认为就是一个小型的数据记录仓库.

write_batch_internal.h里面是一些write_batch里面可能使用到的静态函数，将选择将声明放在这个头文件里面而不是放在include/leveldb/write_batch.h里面. 之前有一段注释表明WriteBatch内部二进制组织方式.

```
// WriteBatch::rep_ :=
//    sequence: fixed64
//    count: fixed32
//    data: record[count]
// record :=
//    kTypeValue varstring varstring         |
//    kTypeDeletion varstring
// varstring :=
//    len: varint32
//    data: uint8[len]
```

- sequence fixed64
- 记录长度 fixed32(因为这个写入是多次的,所以存放fixed比较容易修改)
- 每条记录有头部表示数据还是删除.对于数据就是kv,对于删除只是存放k
- kv的话都是varint32 + data

这里rep_的类型是std::stirng.

1. Count

```
int WriteBatchInternal::Count(const WriteBatch* b) {
  return DecodeFixed32(b->rep_.data() + 8);
}

void WriteBatchInternal::SetCount(WriteBatch* b, int n) {
  EncodeFixed32(&b->rep_[8], n);
}
```

2. Sequence

```
SequenceNumber WriteBatchInternal::Sequence(const WriteBatch* b) {
  return SequenceNumber(DecodeFixed64(b->rep_.data()));
}

void WriteBatchInternal::SetSequence(WriteBatch* b, SequenceNumber seq) {
  EncodeFixed64(&b->rep_[0], seq);
}
```

3. Contents

```
// 直接设置WriteBatch内容.不过我觉得应该没有太多用途吧.
void WriteBatchInternal::SetContents(WriteBatch* b, const Slice& contents) {
  assert(contents.size() >= 12);
  b->rep_.assign(contents.data(), contents.size());
}
```

4. Clear

```
void WriteBatch::Clear() {
  rep_.clear();
  rep_.resize(12); // 头部必然存放12个字节.
}
```

5. Put

```
void WriteBatch::Put(const Slice& key, const Slice& value) {
  WriteBatchInternal::SetCount(this, WriteBatchInternal::Count(this) + 1);
  rep_.push_back(static_cast<char>(kTypeValue)); // 仅仅是value类型操作
  PutLengthPrefixedSlice(&rep_, key); // 放入k,v.之前都存放了长度.
  PutLengthPrefixedSlice(&rep_, value);
}
```

6. Delete

```
void WriteBatch::Delete(const Slice& key) {
  WriteBatchInternal::SetCount(this, WriteBatchInternal::Count(this) + 1);
  rep_.push_back(static_cast<char>(kTypeDeletion)); // 仅仅是delete类型操作
  PutLengthPrefixedSlice(&rep_, key); // 仅仅是存放key.
}
```

7. InsertInto

```
// 将WriteBatch内容全部插入到MemTable里面去.调用Iterate接口.
// MemTableInserter会在后面讲解到.这个可以用于将WriteBatch提交.
Status WriteBatchInternal::InsertInto(const WriteBatch* b,
                                      MemTable* memtable) {
  MemTableInserter inserter;
  inserter.sequence_ = WriteBatchInternal::Sequence(b);
  inserter.mem_ = memtable;
  return b->Iterate(&inserter);
}
```

8. Iterate

   遍历内容存放到handler对象里面.要求handler有Put和Delete实现.这个实现在后面MemTableInserter有.

```
Status WriteBatch::Iterate(Handler* handler) const {
  Slice input(rep_);
  if (input.size() < 12) {
    return Status::Corruption("malformed WriteBatch (too small)");
  }

  // 忽略之前的12个字节.
  input.remove_prefix(12);
  Slice key, value;
  int found = 0; // found应该是用来进行校验的.
  while (!input.empty()) {
    found++;
    char tag = input[0];
    input.remove_prefix(1);
    switch (tag) {
      case kTypeValue: // 如果是kv数据的话.
        if (GetLengthPrefixedSlice(&input, &key) &&
            GetLengthPrefixedSlice(&input, &value)) {
          handler->Put(key, value); // 调用handler->Put接口.
        } else {
          return Status::Corruption("bad WriteBatch Put"); // 否则认为数据损坏.
        }
        break;
      case kTypeDeletion: // 如果是delete操作的话.
        if (GetLengthPrefixedSlice(&input, &key)) {
          handler->Delete(key); // 那么调用Delete接口
        } else {
          return Status::Corruption("bad WriteBatch Delete");
        }
        break;
      default: // 默认为数据损坏.
        return Status::Corruption("unknown WriteBatch tag");
    }
  }
  // 校验一下个数是否正确.
  if (found != WriteBatchInternal::Count(this)) {
    return Status::Corruption("WriteBatch has wrong count");
  } else {
    return Status::OK();
  }
}
```

10.6.2 MemTable

db/memtable.cc MemTable应该算是leveldb里面的table一种.只不过所有的内容都是存放在内存里面的。我们首先看看字段和数据结构.

(#note: 阅读MemTable必须和WriteBatch的数据结构分开，这里非常容易弄混。WriteBatch是一个连续字节流的表示，是将所有的请求 序列化称为连续的字节流，而MemTable是将每一对kv做成一个连续字节流放入SkipList里面去.如果不分清楚的话这里非常容易搞混).

1. MemTable

```
class MemTable {
 public:
  // MemTables are reference counted.  The initial reference count
  // is zero and the caller must call Ref() at least once.
  explicit MemTable(const InternalKeyComparator& comparator);

  // Increase reference count.
  void Ref() { ++refs_; } // 存在引用计数并且一开始==0.

  // Drop reference count.  Delete if no more references exist.
  void Unref() {
    --refs_;
    assert(refs_ >= 0);
    if (refs_ <= 0) {
      delete this;
    }
  }

  // Returns an estimate of the number of bytes of data in use by this
  // data structure.
  //
  // REQUIRES: external synchronization to prevent simultaneous
  // operations on the same MemTable.
  size_t ApproximateMemoryUsage(); // 调用arena的MemoryUsage.因为这里面所有的内存都是从arena里面分配的.

  // Return an iterator that yields the contents of the memtable.
  //
  // The caller must ensure that the underlying MemTable remains live
  // while the returned iterator is live.  The keys returned by this
  // iterator are internal keys encoded by AppendInternalKey in the
  // db/format.{h,cc} module.
  Iterator* NewIterator(); // 产生新的迭代器.接下来我们会看看迭代器是如何实现的。

  // Add an entry into memtable that maps key to value at the
  // specified sequence number and with the specified type.
  // Typically value will be empty if type==kTypeDeletion.
  void Add(SequenceNumber seq, ValueType type,
           const Slice& key,
           const Slice& value);

  // If memtable contains a value for key, store it in *value and return true.
  // If memtable contains a deletion for key, store a NotFound() error
  // in *status and return true.
  // Else, return false.
  bool Get(const LookupKey& key, std::string* value, Status* s);

 private:
  ~MemTable();  // Private since only Unref() should be used to delete it

  struct KeyComparator {
    const InternalKeyComparator comparator;
    explicit KeyComparator(const InternalKeyComparator& c) : comparator(c) { }
    int operator()(const char* a, const char* b) const;
  };
  friend class MemTableIterator;
  friend class MemTableBackwardIterator;

  typedef SkipList<const char*, KeyComparator> Table; // 底层Table结构是使用SkipList存储的.

  KeyComparator comparator_; // k比较器.
  int refs_; // ref.
  Arena arena_; // 内存分配
  Table table_; // table
};
```

2. Compare

这里KeyComparator的话因为考虑到key是length-prefixed的.底层使用InternalKeyComparator处理的就是 后面我们会看看InternalKeyComparator的实现.

```
int MemTable::KeyComparator::operator()(const char* aptr, const char* bptr)
    const {
  // Internal keys are encoded as length-prefixed strings.
  Slice a = GetLengthPrefixedSlice(aptr);
  Slice b = GetLengthPrefixedSlice(bptr);
  return comparator.Compare(a, b); // 这里调用底层的
  //InternalKeyComparator.注意这里的aptr,bptr包括了后面附加的8字节信息.
}
```

3. Add

```
void MemTable::Add(SequenceNumber s, ValueType type,
                   const Slice& key,
                   const Slice& value) {
  // Format of an entry is concatenation of:
  //  key_size     : varint32 of internal_key.size()
  //  key bytes    : char[internal_key.size()]
  //  value_size   : varint32 of value.size()
  //  value bytes  : char[value.size()]
  size_t key_size = key.size();
  size_t val_size = value.size();
  size_t internal_key_size = key_size + 8; // internal key size +8.
  const size_t encoded_len =
      // 注意这里的key size包括了8字节的附加信息.
      VarintLength(internal_key_size) + internal_key_size +
      VarintLength(val_size) + val_size;
  char* buf = arena_.Allocate(encoded_len);
  char* p = EncodeVarint32(buf, internal_key_size);
  memcpy(p, key.data(), key_size);
  p += key_size;
  EncodeFixed64(p, (s << 8) | type); // 这就是8个字节代表的含义.(seq << 8) | type.
  p += 8;
  p = EncodeVarint32(p, val_size);
  memcpy(p, value.data(), val_size);
  assert((p + val_size) - buf == encoded_len);
  table_.Insert(buf); // 然后将这个key插入.
}
```

4. Get

```
// 这里使用迭代器查找对应的内容.格式的话可以参考Add的里面的实现
// key_size + key_data + (seq << 8) | type(8 bytes) + val_size + val_data.
bool MemTable::Get(const LookupKey& key, std::string* value, Status* s) {
  Slice memkey = key.memtable_key();
  Table::Iterator iter(&table_);
  iter.Seek(memkey.data());
  if (iter.Valid()) {
    // entry format is:
    //    klength  varint32
    //    userkey  char[klength]
    //    tag      uint64
    //    vlength  varint32
    //    value    char[vlength]
    // Check that it belongs to same user key.  We do not check the
    // sequence number since the Seek() call above should have skipped
    // all entries with overly large sequence numbers.
    const char* entry = iter.key();
    uint32_t key_length;
    const char* key_ptr = GetVarint32Ptr(entry, entry+5, &key_length);
    // 注意这里使用底层的comparator来进行比较.
    if (comparator_.comparator.user_comparator()->Compare(
            Slice(key_ptr, key_length - 8),
            key.user_key()) == 0) {
      // Correct user key
      const uint64_t tag = DecodeFixed64(key_ptr + key_length - 8);
      switch (static_cast<ValueType>(tag & 0xff)) {
        case kTypeValue: {
          Slice v = GetLengthPrefixedSlice(key_ptr + key_length);
          value->assign(v.data(), v.size());
          return true;
        }
        case kTypeDeletion:
          *s = Status::NotFound(Slice());
          return true;
      }
    }
  }
  return false;
}
```

10.6.3 MemTableInserter

db/write_batch.cc MemTableInserter作为WriteBatch::Handler实现,基本上可以认为是操作MemTable的代理类. 具体代码非常简单，实现部分可能需要看MemTable.这个后面会讲解

```
class MemTableInserter : public WriteBatch::Handler {
 public:
  SequenceNumber sequence_;
  MemTable* mem_;
  virtual void Put(const Slice& key, const Slice& value) {
    mem_->Add(sequence_, kTypeValue, key, value);
    sequence_++; // 这里这里sequence_在++.虽然不太清楚这里sequence有什么用途.
  }
  virtual void Delete(const Slice& key) {
    mem_->Add(sequence_, kTypeDeletion, key, Slice());
    sequence_++;
  }
};
```

至于sequence的初始值，在WriteBatchInternal::InsertInto里面是WriteBatch本身的Sequence作为初始值的。

10.6.4 MemtableIterator

db/memtable.cc MemtableIterator实现非常简单，就是之前Table::Iterator的封装.后面我们会着重看看SkipList的实现.

```
typedef SkipList<const char*, KeyComparator> Table;
class MemTableIterator: public Iterator {
 public:
  explicit MemTableIterator(MemTable::Table* table) : iter_(table) { }

  virtual bool Valid() const { return iter_.Valid(); }
  virtual void Seek(const Slice& k) { iter_.Seek(EncodeKey(&tmp_, k)); }
  virtual void SeekToFirst() { iter_.SeekToFirst(); }
  virtual void SeekToLast() { iter_.SeekToLast(); }
  virtual void Next() { iter_.Next(); }
  virtual void Prev() { iter_.Prev(); }
  virtual Slice key() const { return GetLengthPrefixedSlice(iter_.key()); }
  virtual Slice value() const {
    Slice key_slice = GetLengthPrefixedSlice(iter_.key());
    return GetLengthPrefixedSlice(key_slice.data() + key_slice.size());
  }

  virtual Status status() const { return Status::OK(); }

 private:
  MemTable::Table::Iterator iter_;
  std::string tmp_;       // For passing to EncodeKey
};
```

10.7 Log

10.7.1 LogFormat

db/log_format.h 里面规定的是db log格式上面的一些常量和信息.内容非常少，但是可以给予我们一点启发

```
enum RecordType { // record类型.
  // Zero is reserved for preallocated files
  kZeroType = 0,

  kFullType = 1,

  // For fragments
  kFirstType = 2,
  kMiddleType = 3,
  kLastType = 4
};
static const int kMaxRecordType = kLastType;

static const int kBlockSize = 32768; // 一个block大小.

// Header is checksum (4 bytes), type (1 byte), length (2 bytes).
// 从EmitPhysicalRecord可以看到，实际安排是checksum,length,type.
static const int kHeaderSize = 4 + 1 + 2; // 一个block的header是怎么安排的.
```

下面LogWriter以及LogReader里面给出这个格式的实现，代码里面可以看出具体的实现。 但是幸运的是在level的文档里面也有这个format的说明。

```
The log file contents are a sequence of 32KB blocks.  The only
exception is that the tail of the file may contain a partial block.

Each block consists of a sequence of records:
   block := record* trailer?
   record :=
checksum: uint32// crc32c of type and data[]
length: uint16
type: uint8// One of FULL, FIRST, MIDDLE, LAST
data: uint8[length]

A record never starts within the last six bytes of a block (since it
won't fit).  Any leftover bytes here form the trailer, which must
consist entirely of zero bytes and must be skipped by readers.

Aside: if exactly seven bytes are left in the current block, and a new
non-zero length record is added, the writer must emit a FIRST record
(which contains zero bytes of user data) to fill up the trailing seven
bytes of the block and then emit all of the user data in subsequent
blocks.

More types may be added in the future.  Some Readers may skip record
types they do not understand, others may report that some data was
skipped.

FULL == 1
FIRST == 2
MIDDLE == 3
LAST == 4

The FULL record contains the contents of an entire user record.

FIRST, MIDDLE, LAST are types used for user records that have been
split into multiple fragments (typically because of block boundaries).
FIRST is the type of the first fragment of a user record, LAST is the
type of the last fragment of a user record, and MID is the type of all
interior fragments of a user record.

Example: consider a sequence of user records:
   A: length 1000
   B: length 97270
   C: length 8000
A will be stored as a FULL record in the first block.

B will be split into three fragments: first fragment occupies the rest
of the first block, second fragment occupies the entirety of the
second block, and the third fragment occupies a prefix of the third
block.  This will leave six bytes free in the third block, which will
be left empty as the trailer.

C will be stored as a FULL record in the fourth block.

===================

Some benefits over the recordio format:

(1) We do not need any heuristics for resyncing - just go to next
block boundary and scan.  If there is a corruption, skip to the next
block.  As a side-benefit, we do not get confused when part of the
contents of one log file are embedded as a record inside another log
file.

(2) Splitting at approximate boundaries (e.g., for mapreduce) is
simple: find the next block boundary and skip records until we
hit a FULL or FIRST record.

(3) We do not need extra buffering for large records.

Some downsides compared to recordio format:

(1) No packing of tiny records.  This could be fixed by adding a new
record type, so it is a shortcoming of the current implementation,
not necessarily the format.

(2) No compression.  Again, this could be fixed by adding new record types.
```

10.7.2 LogWriter

db/log_writer.h db log writer.这里我们需要结合前面的LogFormatter一起看看.

```
class Writer {
 public:
  // Create a writer that will append data to "*dest".
  // "*dest" must be initially empty.
  // "*dest" must remain live while this Writer is in use.
  explicit Writer(WritableFile* dest);
  ~Writer();

  Status AddRecord(const Slice& slice);

 private:
  WritableFile* dest_;
  int block_offset_;       // Current offset in block.在当前block的位置.

  // crc32c values for all supported record types.  These are
  // pre-computed to reduce the overhead of computing the crc of the
  // record type stored in the header.
  uint32_t type_crc_[kMaxRecordType + 1]; // 各种类型的crc32c.

  Status EmitPhysicalRecord(RecordType type, const char* ptr, size_t length);
};
```

1. Writer

```
Writer::Writer(WritableFile* dest)
    : dest_(dest),
      block_offset_(0) {
  for (int i = 0; i <= kMaxRecordType; i++) {
    char t = static_cast<char>(i); // 针对类型计算的CRC32C.
    type_crc_[i] = crc32c::Value(&t, 1);
  }
}
```

2. AddRecord

   我们这里看看每条Log到底是怎么写入的.

```
Status Writer::AddRecord(const Slice& slice) {
  const char* ptr = slice.data();
  size_t left = slice.size();

  // Fragment the record if necessary and emit it.  Note that if slice
  // is empty, we still want to iterate once to emit a single
  // zero-length record
  // 对于Slice剩余长度为0的话那么依然需要输出一条记录.
  Status s;
  bool begin = true; // 当前为头部.
  do {
    const int leftover = kBlockSize - block_offset_; // 当前block剩余多少内容.
    assert(leftover >= 0);
    if (leftover < kHeaderSize) { // 不足以存放下Header.
      // Switch to a new block
      if (leftover > 0) {
        // Fill the trailer (literal below relies on kHeaderSize being 7)
        assert(kHeaderSize == 7);
        dest_->Append(Slice("\x00\x00\x00\x00\x00\x00", leftover)); // 剩余的部分填充0x00.
      }
      block_offset_ = 0; // 现在blcok_offset==0重新开辟了一块内容.
    }

    // Invariant: we never leave < kHeaderSize bytes in a block.
    assert(kBlockSize - block_offset_ - kHeaderSize >= 0);

    // 当前剩余多少内容.
    const size_t avail = kBlockSize - block_offset_ - kHeaderSize;
    const size_t fragment_length = (left < avail) ? left : avail;

    RecordType type;
    const bool end = (left == fragment_length); // 这个是否为记录的尾部.
    if (begin && end) { // 如果完全记录那么FullType.
      type = kFullType;
    } else if (begin) { // 如果只是开头,FirstType.
      type = kFirstType;
    } else if (end) { // 如果不是开头但是结尾LastType.
      type = kLastType;
    } else {
      type = kMiddleType; // 否则MiddleType.
    }

    s = EmitPhysicalRecord(type, ptr, fragment_length); // 将这个部分输出.
    ptr += fragment_length;
    left -= fragment_length;
    begin = false;
  } while (s.ok() && left > 0);
  return s;
}
```

3. EmitPhysicalRecord

   我们在上面AddRecord里面看到了调用EmitPhysicalRecord.我们这里看看实现.参考在LogFormatter里面的注释.

```
Status Writer::EmitPhysicalRecord(RecordType t, const char* ptr, size_t n) {
  assert(n <= 0xffff);  // Must fit in two bytes
  assert(block_offset_ + kHeaderSize + n <= kBlockSize);

  // Format the header
  char buf[kHeaderSize];
  // 这里存在差别，最后一个字节表示type.
  buf[4] = static_cast<char>(n & 0xff); // 低位.
  buf[5] = static_cast<char>(n >> 8); // 高位.这样可以小端方式读取.
  buf[6] = static_cast<char>(t);

  // Compute the crc of the record type and the payload.
  uint32_t crc = crc32c::Extend(type_crc_[t], ptr, n); // 针对ptr进行CRC32加密.
  crc = crc32c::Mask(crc);                 // Adjust for storage
  EncodeFixed32(buf, crc); // 存放在header部分.

  // Write the header and the payload
  Status s = dest_->Append(Slice(buf, kHeaderSize)); // 首先写入Header部分.
  if (s.ok()) {
    s = dest_->Append(Slice(ptr, n)); // 然后追加数据部分.
    if (s.ok()) {
      s = dest_->Flush();
    }
  }
  block_offset_ += kHeaderSize + n;
  return s;
}
```

10.7.3 LogReader

db/log_reader.h db log reader.这里我们需要结合前面的LogReader以及LogFormatter一起看看.

```
class Reader {
 public:
  // Interface for reporting errors.
  // 汇报错误接口.
  class Reporter {
   public:
    virtual ~Reporter();

    // Some corruption was detected.  "size" is the approximate number
    // of bytes dropped due to the corruption.
    virtual void Corruption(size_t bytes, const Status& status) = 0;
  };

  // Create a reader that will return log records from "*file".
  // "*file" must remain live while this Reader is in use.
  //
  // If "reporter" is non-NULL, it is notified whenever some data is
  // dropped due to a detected corruption.  "*reporter" must remain
  // live while this Reader is in use.
  //
  // If "checksum" is true, verify checksums if available.
  //
  // The Reader will start reading at the first record located at physical
  // position >= initial_offset within the file.
  Reader(SequentialFile* file, Reporter* reporter, bool checksum,
         uint64_t initial_offset);

  ~Reader();

  // Read the next record into *record.  Returns true if read
  // successfully, false if we hit end of the input.  May use
  // "*scratch" as temporary storage.  The contents filled in *record
  // will only be valid until the next mutating operation on this
  // reader or the next mutation to *scratch.
  bool ReadRecord(Slice* record, std::string* scratch);

  // Returns the physical offset of the last record returned by ReadRecord.
  //
  // Undefined before the first call to ReadRecord.
  uint64_t LastRecordOffset();
 private:
  SequentialFile* const file_; // 顺序读取文件对象.
  Reporter* const reporter_; // 报告错误对象.
  bool const checksum_; // 检查CRC32C.
  char* const backing_store_; // block buffer.
  Slice buffer_; // record buffer.
  // 表示上次读取block是否到达尾部.
  bool eof_;   // Last Read() indicated EOF by returning < kBlockSize

  // Offset of the last record returned by ReadRecord.
  uint64_t last_record_offset_; // 上次ReadRecord完成之后的文件偏移.
  // Offset of the first location past the end of buffer_.
  uint64_t end_of_buffer_offset_; // 读取到buffer_之后顺序文件偏移.

  // Offset at which to start looking for the first record to return
  uint64_t const initial_offset_; // 初始化读取的offset.

  // Extend record types with the following special values
  enum {
    kEof = kMaxRecordType + 1,
    // Returned whenever we find an invalid physical record.
    // Currently there are three situations in which this happens:
    // * The record has an invalid CRC (ReadPhysicalRecord reports a drop)
    // * The record is a 0-length record (No drop is reported)
    // * The record is below constructor's initial_offset (No drop is reported)
    kBadRecord = kMaxRecordType + 2
  };

  // Skips all blocks that are completely before "initial_offset_".
  //
  // Returns true on success. Handles reporting.
  bool SkipToInitialBlock();

  // Return type, or one of the preceding special values
  unsigned int ReadPhysicalRecord(Slice* result);

  // Reports dropped bytes to the reporter.
  // buffer_ must be updated to remove the dropped bytes prior to invocation.
  void ReportCorruption(size_t bytes, const char* reason);
  void ReportDrop(size_t bytes, const Status& reason);
};
```

1. Reader

   构造函数非常简单，注意这里开辟的backing_store_，可以猜想到这个backing_store是读取block内容临时开辟. 对于这些字段的含义可能第一眼看上去不太明白，这个分析代码应该可以看到每个字段的含义。

```
Reader::Reader(SequentialFile* file, Reporter* reporter, bool checksum,
               uint64_t initial_offset)
    : file_(file),
      reporter_(reporter),
      checksum_(checksum),
      backing_store_(new char[kBlockSize]),
      buffer_(),
      eof_(false),
      last_record_offset_(0),
      end_of_buffer_offset_(0),
      initial_offset_(initial_offset) {
}
```

2. ReadPhysicalRecord

   从文件里面读取一个物理块大小出来kBlockSize.并且解析内容存放在bakcing_store_里面，同时返回type. 这里很多细节都没有仔细看,只是理解大致的意思.

```
unsigned int Reader::ReadPhysicalRecord(Slice* result) {
  while (true) {
    if (buffer_.size() < kHeaderSize) { // 如果之前读取的内容不够kHeaderSize大小的话.
      if (!eof_) { // 如果上次读取没有到末尾的话，那么认为上次读取无效，直接忽略.
        // Last read was a full read, so this is a trailer to skip
        buffer_.clear();
        // 忽略之后重新读取一块.
        Status status = file_->Read(kBlockSize, &buffer_, backing_store_);
        // 修正读取偏移.当前读取偏移.
        end_of_buffer_offset_ += buffer_.size();
        if (!status.ok()) { // 如果读取失败的话.
          buffer_.clear();
          ReportDrop(kBlockSize, status);
          eof_ = true;
          return kEof;
        } else if (buffer_.size() < kBlockSize) {
          // 成功但是已经达到结尾了.eof_=true
          // 重新判断读取的是否为正确的记录.
          // 如果没有正确读取的话(<kHeaderSize).会进入下面那个错误逻辑.
          // 否则进入正常逻辑.
          eof_ = true;
        }
        continue;
      } else if (buffer_.size() == 0) { // 这个是确实没有任何记录存在了.
        // End of file
        return kEof;
      } else { // 文件末尾读取失败(截断)
        size_t drop_size = buffer_.size();
        buffer_.clear();
        ReportCorruption(drop_size, "truncated record at end of file");
        return kEof;
      }
    }

    // 假设到这里的话我们都已经读取了一个完整的block了.我们首先解析头部.
    // Parse the header
    const char* header = buffer_.data();
    const uint32_t a = static_cast<uint32_t>(header[4]) & 0xff;
    const uint32_t b = static_cast<uint32_t>(header[5]) & 0xff;
    const unsigned int type = header[6]; // 根据类型和长度.
    const uint32_t length = a | (b << 8);
    if (kHeaderSize + length > buffer_.size()) {
      size_t drop_size = buffer_.size();
      buffer_.clear();
      ReportCorruption(drop_size, "bad record length");
      return kBadRecord;
    }

    if (type == kZeroType && length == 0) {
      // Skip zero length record without reporting any drops since
      // such records are produced by the mmap based writing code in
      // env_posix.cc that preallocates file regions.
      buffer_.clear();
      return kBadRecord;
    }

    // 检查crc校验.
    // Check crc
    if (checksum_) {
      uint32_t expected_crc = crc32c::Unmask(DecodeFixed32(header));
      uint32_t actual_crc = crc32c::Value(header + 6, 1 + length); // 做crc32c需要包含type字段.
      if (actual_crc != expected_crc) {
        // Drop the rest of the buffer since "length" itself may have
        // been corrupted and if we trust it, we could find some
        // fragment of a real log record that just happens to look
        // like a valid log record.
        size_t drop_size = buffer_.size();
        buffer_.clear();
        ReportCorruption(drop_size, "checksum mismatch");
        return kBadRecord;
      }
    }

    // 对buffer_这里做一个校验.
    buffer_.remove_prefix(kHeaderSize + length);

    // Skip physical record that started before initial_offset_
    if (end_of_buffer_offset_ - buffer_.size() - kHeaderSize - length <
        initial_offset_) {
      result->clear();
      return kBadRecord;
    }

    *result = Slice(header + kHeaderSize, length);
    return type;
  }
}
```

3. ReadRecord

ReadRecord读取的是一条逻辑记录.底层调用ReadPhysicalRecord将多条物理记录结合起来.实话说里面具体逻辑没有看懂，但是大致意思算是明白了。对于FullType来说的话record里面使用backing_store内存，而对于First/Middle/Last来说的话 里面使用的是scratch分配的内存.

```
bool Reader::ReadRecord(Slice* record, std::string* scratch) {
  // 如果还没有跳过initial offset的话
  if (last_record_offset_ < initial_offset_) {
    if (!SkipToInitialBlock()) { // 这个函数后面会分析.
      return false;
    }
  }

  scratch->clear(); // 将scratch以及record清空
  record->clear(); // 如果没有猜错的话，那么record里面管理的是scratch的内存.
  // 但是之后会发现其实record里面也可能含有backing_store的内容.
  bool in_fragmented_record = false;
  // Record offset of the logical record that we're reading
  // 0 is a dummy value to make compilers happy
  uint64_t prospective_record_offset = 0;

  Slice fragment;
  while (true) {
    uint64_t physical_record_offset = end_of_buffer_offset_ - buffer_.size();
    const unsigned int record_type = ReadPhysicalRecord(&fragment); // 读取一个block.
    switch (record_type) {
      case kFullType:
        if (in_fragmented_record) {
          // Handle bug in earlier versions of log::Writer where
          // it could emit an empty kFirstType record at the tail end
          // of a block followed by a kFullType or kFirstType record
          // at the beginning of the next block.
          if (scratch->empty()) {
            in_fragmented_record = false; // 忽略这种错误.可能是考虑兼容.
          } else { // 读取到FullType但是却有fragmented的表示.
            ReportCorruption(scratch->size(), "partial record without end(1)");
          }
        }
        prospective_record_offset = physical_record_offset;
        scratch->clear();
        *record = fragment; // 直接给record.这里record持有backing_store数据.
        last_record_offset_ = prospective_record_offset;
        return true;

      case kFirstType:
        if (in_fragmented_record) {
          // Handle bug in earlier versions of log::Writer where
          // it could emit an empty kFirstType record at the tail end
          // of a block followed by a kFullType or kFirstType record
          // at the beginning of the next block.
          if (scratch->empty()) {
            in_fragmented_record = false; // 忽略这种错误,可能是考虑兼容.
          } else {
            ReportCorruption(scratch->size(), "partial record without end(2)");
          }
        }
        prospective_record_offset = physical_record_offset;
        scratch->assign(fragment.data(), fragment.size());
        in_fragmented_record = true;
        break;

      case kMiddleType:
        if (!in_fragmented_record) {
          ReportCorruption(fragment.size(),
                           "missing start of fragmented record(1)");
        } else {
          scratch->append(fragment.data(), fragment.size());
        }
        break;

      case kLastType:
        if (!in_fragmented_record) {
          ReportCorruption(fragment.size(),
                           "missing start of fragmented record(2)");
        } else {
          scratch->append(fragment.data(), fragment.size());
          *record = Slice(*scratch);
          last_record_offset_ = prospective_record_offset;
          return true;
        }
        break;

      case kEof:
        if (in_fragmented_record) {
          ReportCorruption(scratch->size(), "partial record without end(3)");
          scratch->clear();
        }
        return false;

      case kBadRecord:
        if (in_fragmented_record) {
          ReportCorruption(scratch->size(), "error in middle of record");
          in_fragmented_record = false;
          scratch->clear();
        }
        break;

      default: {
        char buf[40];
        snprintf(buf, sizeof(buf), "unknown record type %u", record_type);
        ReportCorruption(
            (fragment.size() + (in_fragmented_record ? scratch->size() : 0)),
            buf);
        in_fragmented_record = false;
        scratch->clear();
        break;
      }
    }
  }
  return false;
}
```

4. SkipToInitialBlock

   跳过开头的部分.实际上只是跳过按照kBlockSize整数块部分。这个部分不需要用户主动调用.在ReadRecord里面调用.

```
bool Reader::SkipToInitialBlock() {
  size_t offset_in_block = initial_offset_ % kBlockSize;
  uint64_t block_start_location = initial_offset_ - offset_in_block;

  // Don't search a block if we'd be in the trailer
  if (offset_in_block > kBlockSize - 6) { // 这个不太明白是什么意思.为什么是6.
    offset_in_block = 0;
    block_start_location += kBlockSize;
  }

  end_of_buffer_offset_ = block_start_location;

  // Skip to start of first block that can contain the initial record
  if (block_start_location > 0) {
    Status skip_status = file_->Skip(block_start_location);
    if (!skip_status.ok()) {
      ReportDrop(block_start_location, skip_status); // 后面会分析.
      return false;
    }
  }

  return true;
}
```

5. ReportDrop

```
void Reader::ReportDrop(size_t bytes, const Status& reason) {
  if (reporter_ != NULL &&
      end_of_buffer_offset_ - buffer_.size() - bytes >= initial_offset_) { // 对于后面这个条件没有看懂
    // 应该是判断如果还没有开始读的话那么不会报告错误.
    reporter_->Corruption(bytes, reason);
  }
}
```

6. ReportCorruption

   这个过程非常简单就是调用ReportDrop.然后使用Status::Corruption封装.

```
void Reader::ReportCorruption(size_t bytes, const char* reason) {
  ReportDrop(bytes, Status::Corruption(reason));
}
```

7. LastRecordOffset

   上次调用ReadRecord之后的文件偏移是多少

```
uint64_t Reader::LastRecordOffset() {
  return last_record_offset_;
}
```

10.8 Table

10.8.1 TableFormat

我们在分析后面的接口之前最好首先看看table文件的组织格式是怎么样的。这些table都是排好序的所以查询特别快。 对于这些table通常称为sstable(sorted string table).关于table格式在leveldb/doc/table_format.txt里面有说明. 本质上我们可认为sstable是一个二级索引文件.首先读取footer.得到metaindex block和index block位置.然后读取index block 或者是metaindex block(上面存在排序k进行二分查找)，然后读取data block(k排序然后进行二分查找).

```
File format
===========

  <beginning_of_file>
  [data block 1]
  [data block 2]
  ...
  [data block N]
  [meta block 1]
  ...
  [meta block K]
  [metaindex block]
  [index block]
  [Footer]        (fixed size; starts at file_size - sizeof(Footer))
  <end_of_file>

The file contains internal pointers.  Each such pointer is called
a BlockHandle and contains the following information:
offset:    varint64
size:    varint64

(1) The sequence of key/value pairs in the file are stored in sorted
order and partitioned into a sequence of data blocks.  These blocks
come one after another at the beginning of the file.  Each data block
is formatted according to the code in block_builder.cc, and then
optionally compressed.

(2) After the data blocks we store a bunch of meta blocks.  The
supported meta block types are described below.  More meta block types
may be added in the future.  Each meta block is again formatted using
block_builder.cc and then optionally compressed.

(3) A "metaindex" block.  It contains one entry for every other meta
block where the key is the name of the meta block and the value is a
BlockHandle pointing to that meta block.

(4) An "index" block.  This block contains one entry per data block,
where the key is a string >= last key in that data block and before
the first key in the successive data block.  The value is the
BlockHandle for the data block.

(6) At the very end of the file is a fixed length footer that contains
the BlockHandle of the metaindex and index blocks as well as a magic number.
       metaindex_handle:       char[p];    // Block handle for metaindex
index_handle:       char[q];    // Block handle for index
padding:       char[40-p-q]; // 0 bytes to make fixed length
       // (40==2*BlockHandle::kMaxEncodedLength)
magic:       fixed64;    // == 0xdb4775248b80fb57

"stats" Meta Block
------------------

This meta block contains a bunch of stats.  The key is the name
of the statistic.  The value contains the statistic.
TODO(postrelease): record following stats.
  data size
  index size
  key size (uncompressed)
  value size (uncompressed)
  number of entries
  number of data blocks
```

10.8.2 BuildTable

db/builder.h BuildTable过程非常简单就是构造一个sstable.

```
Status BuildTable(const std::string& dbname,
                  Env* env,
                  const Options& options,
                  TableCache* table_cache,
                    Iterator* iter,
                    FileMetaData* meta) {
  Status s;
  meta->file_size = 0;
  iter->SeekToFirst();

  std::string fname = TableFileName(dbname, meta->number); // 注意number这个字段含义.
  if (iter->Valid()) {
    WritableFile* file;
    s = env->NewWritableFile(fname, &file); // 创建新的文件.
    if (!s.ok()) {
      return s;
    }

    TableBuilder* builder = new TableBuilder(options, file);
    meta->smallest.DecodeFrom(iter->key()); // 遍历iter里面的对象.这里会记录
    // 最小和最大的key记录在meta里面.
    for (; iter->Valid(); iter->Next()) {
      Slice key = iter->key();
      meta->largest.DecodeFrom(key);
      builder->Add(key, iter->value());
    }

    // Finish and check for builder errors
    if (s.ok()) {
      s = builder->Finish(); // 写入index以及footer部分.
      if (s.ok()) {
        meta->file_size = builder->FileSize();
        assert(meta->file_size > 0);
      }
    } else {
      builder->Abandon();
    }
    delete builder;

    // Finish and check for file errors
    if (s.ok()) {
      s = file->Sync();
    }
    if (s.ok()) {
      s = file->Close();
    }
    delete file;
    file = NULL;

    // 会将这个sstable加入到TableCache里面.
    if (s.ok()) {
      // Verify that the table is usable
      Iterator* it = table_cache->NewIterator(ReadOptions(),
                                              meta->number,
                                              meta->file_size);
      s = it->status();
      delete it;
    }
  }

  // Check for input iterator errors
  if (!iter->status().ok()) {
    s = iter->status();
  }

  if (s.ok() && meta->file_size > 0) {
    // Keep it
  } else {
    env->DeleteFile(fname);
  }
  return s;
}
```

10.8.3 TableCache

db/table_cache.cc TableCache的工作非常简单，就是针对Table对象以及Table::Iterator对象进行cache. 这样底层的话可以防止过多的文件打开。了解这个功能之后代码就非常好阅读了。我们首先看看TableCache结构. 对于Cache的key使用uint64_t file_number来进行标记.

```
class TableCache {
 public:
  TableCache(const std::string& dbname, const Options* options, int entries);
  ~TableCache();

  // Return an iterator for the specified file number (the corresponding
  // file length must be exactly "file_size" bytes).  If "tableptr" is
  // non-NULL, also sets "*tableptr" to point to the Table object
  // underlying the returned iterator, or NULL if no Table object underlies
  // the returned iterator.  The returned "*tableptr" object is owned by
  // the cache and should not be deleted, and is valid for as long as the
  // returned iterator is live.
  Iterator* NewIterator(const ReadOptions& options,
                        uint64_t file_number,
                        uint64_t file_size,
                        Table** tableptr = NULL);

  // Evict any entry for the specified file number
  void Evict(uint64_t file_number);

 private:
  Env* const env_; // 底层env
  const std::string dbname_; // 打开db名称.
  const Options* options_; // 打开options.
  Cache* cache_; // Cache对象.
};
```

1. TableCache

```
TableCache::TableCache(const std::string& dbname,
                       const Options* options,
                       int entries)
    : env_(options->env),
      dbname_(dbname),
      options_(options),
      // entries表示Cache的capacity.
      cache_(NewLRUCache(entries)) {
}

TableCache::~TableCache() {
  delete cache_;
}
```

2. Evict

```
void TableCache::Evict(uint64_t file_number) {
  char buf[sizeof(file_number)];
  EncodeFixed64(buf, file_number);
  cache_->Erase(Slice(buf, sizeof(buf)));
}
```

3. NewIterator

```
struct TableAndFile {
  RandomAccessFile* file; // 将file和table绑定.
  Table* table;
};

static void DeleteEntry(const Slice& key, void* value) {
  TableAndFile* tf = reinterpret_cast<TableAndFile*>(value);
  delete tf->table;
  delete tf->file;
  delete tf;
}

static void UnrefEntry(void* arg1, void* arg2) {
  Cache* cache = reinterpret_cast<Cache*>(arg1);
  Cache::Handle* h = reinterpret_cast<Cache::Handle*>(arg2);
  cache->Release(h);
}

Iterator* TableCache::NewIterator(const ReadOptions& options,
                                  uint64_t file_number,
                                  uint64_t file_size,
                                  Table** tableptr) {
  if (tableptr != NULL) {
    *tableptr = NULL;
  }

  char buf[sizeof(file_number)];
  EncodeFixed64(buf, file_number); // 根据file_number作为key查询.
  Slice key(buf, sizeof(buf));
  Cache::Handle* handle = cache_->Lookup(key); // 首先在cache里面查找.
  if (handle == NULL) {
    std::string fname = TableFileName(dbname_, file_number);
    RandomAccessFile* file = NULL;
    Table* table = NULL;
    Status s = env_->NewRandomAccessFile(fname, &file);
    if (s.ok()) {
      s = Table::Open(*options_, file, file_size, &table);
    }

    if (!s.ok()) {
      assert(table == NULL);
      delete file;
      // We do not cache error results so that if the error is transient,
      // or somebody repairs the file, we recover automatically.
      return NewErrorIterator(s);
    }
    // 创建一个cache item
    // 注意这里charge=1那么capacity就表示个数.
    TableAndFile* tf = new TableAndFile;
    tf->file = file;
    tf->table = table;
    handle = cache_->Insert(key, tf, 1, &DeleteEntry);
  }

  Table* table = reinterpret_cast<TableAndFile*>(cache_->Value(handle))->table;
  Iterator* result = table->NewIterator(options);
  // 这里cleanup函数是减少引用计数.
  result->RegisterCleanup(&UnrefEntry, cache_, handle);
  if (tableptr != NULL) {
    *tableptr = table;
  }
  return result;
}
```

10.8.4 Table

table/table.cc 这个接口在include/leveldb/table.h里面已经给出了.我们这里就是看看具体实现.

1. Rep

   Rep是Table内部的数据结构.这样实现了之后那么头文件里面可以不包含任何实现了.

```
struct Table::Rep {
  ~Rep() {
    delete index_block;
  }

  Options options;
  Status status;
  RandomAccessFile* file;
  uint64_t cache_id; // todo:在BlockReader里面作为cache key的一部分存在.
  // 问题是这个cache_id主要是用来解决什么问题的呢?

  BlockHandle metaindex_handle;  // Handle to metaindex_block: saved from footer
  Block* index_block; // 保存data index的block.
};
```

2. Open

```
Status Table::Open(const Options& options,
                   RandomAccessFile* file,
                   uint64_t size,
                   Table** table) {
  *table = NULL;
  if (size < Footer::kEncodedLength) { // 对于文件大小肯定需要一个footer对象.
    return Status::InvalidArgument("file is too short to be an sstable");
  }

  char footer_space[Footer::kEncodedLength]; // footer空间.
  Slice footer_input; // 读取footer.
  Status s = file->Read(size - Footer::kEncodedLength, Footer::kEncodedLength,
                        &footer_input, footer_space);
  if (!s.ok()) return s;

  Footer footer;
  s = footer.DecodeFrom(&footer_input);
  if (!s.ok()) return s;

  // Read the index block
  Block* index_block = NULL;
  if (s.ok()) { // 读取index block.
    s = ReadBlock(file, ReadOptions(), footer.index_handle(), &index_block);
  }
  if (s.ok()) {
    // We've successfully read the footer and the index block: we're
    // ready to serve requests.
    Rep* rep = new Table::Rep; // new创建Rep对象然后构造Table对象.
    rep->options = options;
    rep->file = file;
    rep->metaindex_handle = footer.metaindex_handle();
    rep->index_block = index_block;
    rep->cache_id = (options.block_cache ? options.block_cache->NewId() : 0);
    *table = new Table(rep);
  } else {
    if (index_block) delete index_block;
  }

  return s;
}
```

3. ApproximateOffsetOf

   根据key找到kv在file的偏移.

```
uint64_t Table::ApproximateOffsetOf(const Slice& key) const {
  Iterator* index_iter =
      rep_->index_block->NewIterator(rep_->options.comparator);
  index_iter->Seek(key); // 使用iterator seek到kv的index的位置.
  uint64_t result;
  if (index_iter->Valid()) {
    BlockHandle handle;
    Slice input = index_iter->value();
    Status s = handle.DecodeFrom(&input); // 从index这个BlockHandle中知道数据的偏移.
    if (s.ok()) {
      result = handle.offset();
    } else {
      // Strange: we can't decode the block handle in the index block.
      // We'll just return the offset of the metaindex block, which is
      // close to the whole file size for this case.
      result = rep_->metaindex_handle.offset();
    }
  } else {
    // key is past the last key in the file.  Approximate the offset
    // by returning the offset of the metaindex block (which is
    // right near the end of the file).
    result = rep_->metaindex_handle.offset();
  }
  // 如果找不到这个key的话，那么返回metaindex block的偏移.
  // 这个是meta block的最后位置.
  delete index_iter;
  return result;
}
```

4. NewIterator

   NewIterator创建的是TwoLevelIterator.意思非常简单因为需要读取两次才能够读到内容.首先读取index block, 然后读取具体的data block.关于TwoLevelIterator的实现会在后面分析.建议首先分析TwoLevelIterator 然后分析BlockReader.

```
Iterator* Table::NewIterator(const ReadOptions& options) const {
  return NewTwoLevelIterator(
      rep_->index_block->NewIterator(rep_->options.comparator),
      &Table::BlockReader, const_cast<Table*>(this), options);
}
```

5. DeleteBlock

```
static void DeleteBlock(void* arg, void* ignored) {
  delete reinterpret_cast<Block*>(arg); // 作为Block直接delete.
}
```

6. DeleteCachedBlock

```
static void DeleteCachedBlock(const Slice& key, void* value) {
  Block* block = reinterpret_cast<Block*>(value); // 作为Block直接删除.
  delete block;
}
```

7. ReleaseBlock

```
static void ReleaseBlock(void* arg, void* h) {
  Cache* cache = reinterpret_cast<Cache*>(arg);
  Cache::Handle* handle = reinterpret_cast<Cache::Handle*>(h);
  cache->Release(handle);
}
```

8. BlockReader

   BlockReader任务就是通过读取index_value给定的位置然后读取对应的Block位置并且返回BlockIterator对象.

```
// Convert an index iterator value (i.e., an encoded BlockHandle)
// into an iterator over the contents of the corresponding 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 = NULL;
  Cache::Handle* cache_handle = NULL;

  BlockHandle handle;
  Slice input = index_value;
  Status s = handle.DecodeFrom(&input); // 先找到offset和size.
  // We intentionally allow extra stuff in index_value so that we
  // can add more features in the future.

  if (s.ok()) {
    if (block_cache != NULL) { // 如果存在Cache的话.
      char cache_key_buffer[16];
      EncodeFixed64(cache_key_buffer, table->rep_->cache_id); // 将handle.offset和cache_id做一个签名查询.
      EncodeFixed64(cache_key_buffer+8, handle.offset());
      Slice key(cache_key_buffer, sizeof(cache_key_buffer));
      cache_handle = block_cache->Lookup(key);
      if (cache_handle != NULL) {
        // 如果存在这个对象的话.
        block = reinterpret_cast<Block*>(block_cache->Value(cache_handle));
      } else {
        // 如果没有这个对象的话.那么直接读取对象.
        s = ReadBlock(table->rep_->file, options, handle, &block);
        if (s.ok() && options.fill_cache) { // 如果查到结果放到cache的话.
          cache_handle = block_cache->Insert( // 插入这个对象.
              // 这里从Cache中删除的回调就是直接delete.
              key, block, block->size(), &DeleteCachedBlock);
        }
      }
    } else {
      s = ReadBlock(table->rep_->file, options, handle, &block);
    }
  }

  Iterator* iter;
  if (block != NULL) {
    // 得到这个block之后的话那么创建这个Iterator.
    // 这个是一个user comparator.
    iter = block->NewIterator(table->rep_->options.comparator);
    if (cache_handle == NULL) { // 如果没有放在cache里面的话.
      // 那么当这个iterator失效的话那么直接将Block删除.
      iter->RegisterCleanup(&DeleteBlock, block, NULL);
    } else {
      // 否则的话那么需要使用引用计数.所以ReleaseBlock里面是采用Release方式.
      iter->RegisterCleanup(&ReleaseBlock, block_cache, cache_handle);
    }
  } else {
    iter = NewErrorIterator(s);
  }
  return iter;
}
```

10.8.5 TableBuilder

table/table_builde.cc TableBuilder接口在之前include/leveldb/table_builder.h里面已经提供了。 这里我们具体看看TableBuilder的接口.关于table格式的话可以参考TableFormat这节.注意这里TableBuilder只是将kv加入到了block并且写入了磁盘，但是对于index block并没有写入磁盘。

1. Rep

   Rep是TableBuilder里面具体涉及到的字段.我们来看看这个结构

```
struct TableBuilder::Rep {
  Options options; // 数据options.
  Options index_block_options; // index options.
  WritableFile* file; // sstable文件.
  uint64_t offset; // 当前向这个file写入了多少数据.
  Status status; // file操作返回的status.
  BlockBuilder data_block; // data block.
  BlockBuilder index_block; // index block.
  std::string last_key; // 上次插入的key.
  int64_t num_entries; // 已经插入了多少个kv.
  bool closed;          // Either Finish() or Abandon() has been called.

  // We do not emit the index entry for a block until we have seen the
  // first key for the next data block.  This allows us to use shorter
  // keys in the index block.  For example, consider a block boundary
  // between the keys "the quick brown fox" and "the who".  We can use
  // "the r" as the key for the index block entry since it is >= all
  // entries in the first block and < all entries in subsequent
  // blocks.
  //
  // Invariant: r->pending_index_entry is true only if data_block is empty.
  bool pending_index_entry; // 是否刚才调用了Finish.
  BlockHandle pending_handle;  // Handle to add to index block

  std::string compressed_output; // 作为compressed存放的内容.

  Rep(const Options& opt, WritableFile* f)
      : options(opt),
        index_block_options(opt),
        file(f),
        offset(0),
        data_block(&options),
        index_block(&index_block_options),
        num_entries(0),
        closed(false),
        pending_index_entry(false) {
    // 里面存放的key是全量.
    index_block_options.block_restart_interval = 1; // index block option restart为1.
  }
};
```

2. ChangeOptions

```
Status TableBuilder::ChangeOptions(const Options& options) {
  // Note: if more fields are added to Options, update
  // this function to catch changes that should not be allowed to
  // change in the middle of building a Table.
  if (options.comparator != rep_->options.comparator) {
    // 需要确保comparator对象没有发生改变.
    return Status::InvalidArgument("changing comparator while building table");
  }

  // Note that any live BlockBuilders point to rep_->options and therefore
  // will automatically pick up the updated options.
  rep_->options = options;
  rep_->index_block_options = options;
  rep_->index_block_options.block_restart_interval = 1
  return Status::OK();
}
```

3. Add

```
void TableBuilder::Add(const Slice& key, const Slice& value) {
  Rep* r = rep_;
  assert(!r->closed);
  if (!ok()) return;
  if (r->num_entries > 0) { // 确保按照顺序操作.
    assert(r->options.comparator->Compare(key, Slice(r->last_key)) > 0);
  }

  if (r->pending_index_entry) { // 如果这里新开辟一个block的话对于第一块没有.
    // 那么我们这里做一个index.
    // index key是按照last_ley和key之间的FindShortestSeparator得到的
    // 这样可以使用二分法来进行搜索.
    assert(r->data_block.empty());
    r->options.comparator->FindShortestSeparator(&r->last_key, key);
    std::string handle_encoding;
    // 阅读完Finish会发现这里handle_encoding实际上是就是last_key的位置.
    // 这里使用FindShortestSeparator更加节省空间作为index_block里面的内容.:).
    // 但这里也决定了index_block的key不能够作为data block里面准确的key.
    r->pending_handle.EncodeTo(&handle_encoding);
    r->index_block.Add(r->last_key, Slice(handle_encoding));
    r->pending_index_entry = false;
  }

  // 更新last_key并且插入data block.
  r->last_key.assign(key.data(), key.size());
  r->num_entries++;
  r->data_block.Add(key, value);

  // 如果当前的size超过block size的话那么就需要刷新.
  const size_t estimated_block_size = r->data_block.CurrentSizeEstimate();
  if (estimated_block_size >= r->options.block_size) {
    Flush();
  }
}
```

4. Flush

```
void TableBuilder::Flush() {
  Rep* r = rep_;
  assert(!r->closed);
  if (!ok()) return;
  if (r->data_block.empty()) return;
  assert(!r->pending_index_entry);
  // 将data block作为Block写入然后将这个handle放在pengding_handle里面.
  // 这个WriteBlock后面会仔细分析了.
  WriteBlock(&r->data_block, &r->pending_handle);
  if (ok()) {
    r->pending_index_entry = true;
    r->status = r->file->Flush();
  }
}
```

5. WriteBlock

写入block的内容并且将block所在的位置交给handle.

```
void TableBuilder::WriteBlock(BlockBuilder* block, BlockHandle* handle) {
  // File format contains a sequence of blocks where each block has:
  //    block_data: uint8[n]
  //    type: uint8
  //    crc: uint32
  assert(ok());
  Rep* r = rep_;
  Slice raw = block->Finish();

  Slice block_contents;
  CompressionType type = r->options.compression;
  // TODO(postrelease): Support more compression options: zlib?
  switch (type) {
    case kNoCompression:
      block_contents = raw;
      break;

    case kSnappyCompression: {
      std::string* compressed = &r->compressed_output;
      // 尝试使用snappy compress.如果压缩更大的话那么放弃.
      if (port::Snappy_Compress(raw.data(), raw.size(), compressed) &&
          compressed->size() < raw.size() - (raw.size() / 8u)) {
        block_contents = *compressed;
      } else {
        // Snappy not supported, or compressed less than 12.5%, so just
        // store uncompressed form
        block_contents = raw;
        type = kNoCompression;
      }
      break;
    }
  }
  // 设置handle为block的位置.
  handle->set_offset(r->offset);
  handle->set_size(block_contents.size());
  r->status = r->file->Append(block_contents);
  if (r->status.ok()) {
    // 写入type和crc33c:).
    char trailer[kBlockTrailerSize];
    trailer[0] = type;
    uint32_t crc = crc32c::Value(block_contents.data(), block_contents.size());
    crc = crc32c::Extend(crc, trailer, 1);  // Extend crc to cover block type
    EncodeFixed32(trailer+1, crc32c::Mask(crc));
    r->status = r->file->Append(Slice(trailer, kBlockTrailerSize));
    if (r->status.ok()) {
      r->offset += block_contents.size() + kBlockTrailerSize;
    }
  }
  r->compressed_output.clear();
  block->Reset();
}
```

6. Finish

   Finish写入的TableFormat表述的最后面的index部分以及footer.

```
Status TableBuilder::Finish() {
  Rep* r = rep_;
  Flush();
  assert(!r->closed);
  r->closed = true;
  BlockHandle metaindex_block_handle;
  BlockHandle index_block_handle;
  if (ok()) { // 实际上这个meta_index_block部分没有任何内容.
    BlockBuilder meta_index_block(&r->options);
    // TODO(postrelease): Add stats and other meta blocks
    WriteBlock(&meta_index_block, &metaindex_block_handle); // 写入之后然后得到handle.
  }
  if (ok()) {
    if (r->pending_index_entry) {
      r->options.comparator->FindShortSuccessor(&r->last_key);
      std::string handle_encoding;
      r->pending_handle.EncodeTo(&handle_encoding);
      r->index_block.Add(r->last_key, Slice(handle_encoding));
      r->pending_index_entry = false;
    }
    // 写入index block.
    WriteBlock(&r->index_block, &index_block_handle);
  }
  if (ok()) { // 最后将footer部分写入.
    Footer footer;
    footer.set_metaindex_handle(metaindex_block_handle);
    footer.set_index_handle(index_block_handle);
    std::string footer_encoding;
    footer.EncodeTo(&footer_encoding);
    r->status = r->file->Append(footer_encoding);
    if (r->status.ok()) {
      r->offset += footer_encoding.size();
    }
  }
  return r->status;
}
```

7. Abandon

```
// 操作非常简单就是放弃构建.
void TableBuilder::Abandon() {
  Rep* r = rep_;
  assert(!r->closed);
  r->closed = true;
}
```

10.8.6 BlockBuilder

table/block_builder.h 从提供的接口来看的话，BlockBuilder功能应该是将多个有序的kv写到一个连续内存块内部。 我们首先看看结构然后具体分析里面的方法.提供的Reset接口允许BlockBuilder重复使用. 底层针对key的prefix部分进行了压缩.

```
class BlockBuilder {
 public:
  explicit BlockBuilder(const Options* options);

  // Reset the contents as if the BlockBuilder was just constructed.
  void Reset();

  // REQUIRES: Finish() has not been callled since the last call to Reset().
  // REQUIRES: key is larger than any previously added key
  void Add(const Slice& key, const Slice& value); // 确保key的有序性.
  // Finish building the block and return a slice that refers to the
  // block contents.  The returned slice will remain valid for the
  // lifetime of this builder or until Reset() is called.
  Slice Finish(); // 完成之后返回写入的buffer.

  // Returns an estimate of the current (uncompressed) size of the block
  // we are building.
  size_t CurrentSizeEstimate() const; // 返回block buffer大小.

  // Return true iff no entries have been added since the last Reset()
  bool empty() const {
    return buffer_.empty();
  }

 private:
  const Options*        options_;
  std::string           buffer_;      // Destination buffer
  std::vector<uint32_t> restarts_;    // Restart points
  int                   counter_;     // Number of entries emitted since restart
  bool                  finished_;    // Has Finish() been called?
  std::string           last_key_;
};
```

1. PrefixCompressed

   我们首先看看BlockBuilder是如何针对key进行prefix-compressed的.对于每K个key的话会保存一个完整key,然后对于 剩余的K-1个key采用prefix-compressed的方式压缩。共享的部分长度叫做shared_bytes,非共享的部分叫做unshared_bytes. 对于保存这些完整的key的点，叫做restarts.这个非常好理解。然后从下面注释可以看到restarts的信息保存在最后.

```
// When we store a key, we drop the prefix shared with the previous
// string.  This helps reduce the space requirement significantly.
// Furthermore, once every K keys, we do not apply the prefix
// compression and store the entire key.  We call this a "restart
// point".  The tail end of the block stores the offsets of all of the
// restart points, and can be used to do a binary search when looking
// for a particular key.  Values are stored as-is (without compression)
// immediately following the corresponding key.
//
// An entry for a particular key-value pair has the form:
//     shared_bytes: varint32
//     unshared_bytes: varint32
//     value_length: varint32
//     key_delta: char[unshared_bytes]
//     value: char[value_length]
// shared_bytes == 0 for restart points.
//
// The trailer of the block has the form:
//     restarts: uint32[num_restarts]
//     num_restarts: uint32
// restarts[i] contains the offset within the block of the ith restart point.
```

2. BlockBuilder

   构造函数非常简单.这里options->block_restrat_interval可能就是之前说的参数K

```
BlockBuilder::BlockBuilder(const Options* options)
    : options_(options),
      restarts_(),
      counter_(0),
      finished_(false) {
  assert(options->block_restart_interval >= 1);
  restarts_.push_back(0);       // First restart point is at offset 0
}
```

   我们这里注意到0就是restarts一个点.

3. Reset

   Reset和构造函数非常简单.

```
void BlockBuilder::Reset() {
  buffer_.clear();
  restarts_.clear();
  restarts_.push_back(0);       // First restart point is at offset 0
  counter_ = 0;
  finished_ = false;
  last_key_.clear();
}
```

4. CurrentSizeEstimate

   按照PrefixCompressed里面的注释来看的话，长度应该就是按照下面的代码计算出来的

```
size_t BlockBuilder::CurrentSizeEstimate() const {
  return (buffer_.size() +                        // Raw data buffer
          restarts_.size() * sizeof(uint32_t) +   // Restart array
          sizeof(uint32_t));                      // Restart array length
}
```

5. Finish

   Finish的工作就是将restarts的信息全部写入到block buffer的结尾吧.同时将buffer_包装称为Slice返回.

```
Slice BlockBuilder::Finish() {
  // Append restart array
  for (size_t i = 0; i < restarts_.size(); i++) {
    PutFixed32(&buffer_, restarts_[i]);
  }
  PutFixed32(&buffer_, restarts_.size());
  finished_ = true;
  return Slice(buffer_);
}
```

6. Add

   我们这里仔细看看prefix-compressed是相对哪一个key来进行压缩的.阅读代码会发现是针对last_key来进行的.

```
void BlockBuilder::Add(const Slice& key, const Slice& value) {
  Slice last_key_piece(last_key_);
  assert(!finished_); // 首先我们不允许finished.
  // 因为counter_是自restart的话那么会reset0.所以应该是<=.
  assert(counter_ <= options_->block_restart_interval);
  // 如果有内容的话，那么这里需要验证key是否有序.
  assert(buffer_.empty() // No values yet?
         || options_->comparator->Compare(key, last_key_piece) > 0);
  size_t shared = 0;
  if (counter_ < options_->block_restart_interval) {
    // See how much sharing to do with previous string
    const size_t min_length = std::min(last_key_piece.size(), key.size());
    // 这里prefix compress针对last_key来做的.
    while ((shared < min_length) && (last_key_piece[shared] == key[shared])) {
      shared++;
    }
  } else {
    // Restart compression
    restarts_.push_back(buffer_.size()); // 注意这里restart存放的是字节偏移而不是counter.
    // 因为记录counter那么在read的时候没有办法还原，因为kv都是变长的.
    counter_ = 0;
    // 如果restart point的话，那么shared的部分应该是0.
  }
  const size_t non_shared = key.size() - shared;

  // Add "<shared><non_shared><value_size>" to buffer_
  PutVarint32(&buffer_, shared);
  PutVarint32(&buffer_, non_shared);
  PutVarint32(&buffer_, value.size());

  // Add string delta to buffer_ followed by value
  buffer_.append(key.data() + shared, non_shared);
  buffer_.append(value.data(), value.size());

  // Update state
  last_key_.resize(shared);
  last_key_.append(key.data() + shared, non_shared);
  assert(Slice(last_key_) == key);
  counter_++;
}
```

10.8.7 Block

table/block.h Block可以认为是BlockBuilder的Reader对象，解析BlockBuilder生成的Block.提供访问接口是遍历。 首先我们看看具体接口.可以看到解析的就是一个完整的Block对象.

```
class Block {
 public:
  // Initialize the block with the specified contents.
  // Takes ownership of data[] and will delete[] it when done.
  Block(const char* data, size_t size);

  ~Block();

  size_t size() const { return size_; }
  Iterator* NewIterator(const Comparator* comparator); // 访问接口方式是遍历.

 private:
  uint32_t NumRestarts() const;

  const char* data_;
  size_t size_;
  uint32_t restart_offset_;     // Offset in data_ of restart array
  // restart数组的偏移.每个restart都是4个字节.
  class Iter; // 内部实现.
};
```

1. NumRestarts

   从之前的BloclBuilder知道restart个数在最后面的4字节

```
inline uint32_t Block::NumRestarts() const {
  assert(size_ >= 2*sizeof(uint32_t));
  return DecodeFixed32(data_ + size_ - sizeof(uint32_t));
}
```

2. Block

   注意这里data已经由这个Block来托管了.在析构函数里面会释放data_

```
Block::Block(const char* data, size_t size)
    : data_(data),
      size_(size) {
  if (size_ < sizeof(uint32_t)) {
    size_ = 0;  // Error marker
  } else {
    restart_offset_ = size_ - (1 + NumRestarts()) * sizeof(uint32_t); // 注意最后面有一个uint32表示restart个数.
    if (restart_offset_ > size_ - sizeof(uint32_t)) { // 如果restart_offset_存在问题的话.
      // The size is too small for NumRestarts() and therefore
      // restart_offset_ wrapped around.
      size_ = 0;
    }
  }
}

Block::~Block() {
  delete[] data_;
}
```

3. NewIterator

   创建一个迭代器.采用了工厂方法创建了具体类.关于这个具体类在后面会被称为BlockIterator来进行分析.

```
Iterator* Block::NewIterator(const Comparator* cmp) {
  if (size_ < 2*sizeof(uint32_t)) {
    return NewErrorIterator(Status::Corruption("bad block contents"));
  }
  const uint32_t num_restarts = NumRestarts();
  if (num_restarts == 0) {
    return NewEmptyIterator();
  } else {
    return new Iter(cmp, data_, restart_offset_, num_restarts);  // 创建BlockIterator.
  }
}
```

10.8.8 BlockIterator

table/block.cc BlockIterator本身完成的功能很简单就是Block的遍历器，我们首先看看结构

```
class Block::Iter : public Iterator {
 private:
  const Comparator* const comparator_; // 比较对象.
  const char* const data_;      // underlying block contents
  uint32_t const restarts_;     // Offset of restart array (list of fixed32)
  uint32_t const num_restarts_; // Number of uint32_t entries in restart array

  // current_ is offset in data_ of current entry.  >= restarts_ if !Valid
  uint32_t current_; // 当前的data_位置.
  uint32_t restart_index_;  // Index of restart block in which current_ falls
  std::string key_; // 这里key需要单独保存,因为我们使用了prefix-compressed.
  Slice value_;
  Status status_;
}
```

1. BlockIterator

   构造函数非常简单

   ```
   Iter(const Comparator* comparator,
        const char* data,
        uint32_t restarts,
        uint32_t num_restarts)
       : comparator_(comparator),
         data_(data),
         restarts_(restarts),
         num_restarts_(num_restarts),
         current_(restarts_),
         restart_index_(num_restarts_) {
     assert(num_restarts_ > 0);
   }
   ```

2. NextEntryOffset

   得到下一个entry的偏移.直接使用value_的偏移和大小即可计算出来

   ```
   // Return the offset in data_ just past the end of the current entry.
   inline uint32_t NextEntryOffset() const {
     return (value_.data() + value_.size()) - data_;
   }
   ```

3. GetRestartPoint

   得到某个index的restart offset.这个直接访问最后的restart可以得到.

   ```
   uint32_t GetRestartPoint(uint32_t index) {
     assert(index < num_restarts_);
     return DecodeFixed32(data_ + restarts_ + index * sizeof(uint32_t));
   }
   ```

4. SeekToRestartPoint

   这个接口的语义是到某个restart point.我们需要调整restart_index_.这里value_为了可以调用NextEntryOffset.

   ```
   void SeekToRestartPoint(uint32_t index) {
     key_.clear();
     restart_index_ = index;
     // current_ will be fixed by ParseNextKey();

     // ParseNextKey() starts at the end of value_, so set value_ accordingly
     uint32_t offset = GetRestartPoint(index);
     value_ = Slice(data_ + offset, 0);
   }
   ```

5. Next

   Next底层调用了ParseNextKey.后面我们会仔细看看ParseNextKey.

   ```
   virtual void Next() {
   ```

```
  assert(Valid());
  ParseNextKey();
}
```

6. SeekToFirst

```
virtual void SeekToFirst() {
  SeekToRestartPoint(0); // 到restart0.
  ParseNextKey(); // 然后解析下一个元素即可.
}
```

7. SeekToLast

```
virtual void SeekToLast() {
  SeekToRestartPoint(num_restarts_ - 1); // 首先到最后一个restart.
  while (ParseNextKey() && NextEntryOffset() < restarts_) { // 一直解析到最后一个元素.
    // Keep skipping
  }
}
```

8. ParseNextKey

```
bool ParseNextKey() {
  current_ = NextEntryOffset(); // 下一个entry offset.
  const char* p = data_ + current_;
  const char* limit = data_ + restarts_;  // Restarts come right after data
  if (p >= limit) { // 如果到达结尾的话.
    // No more entries to return.  Mark as invalid.
    current_ = restarts_;
    restart_index_ = num_restarts_;
    return false;
  }

  // Decode next entry
  uint32_t shared, non_shared, value_length;
  p = DecodeEntry(p, limit, &shared, &non_shared, &value_length); // 从p解析key出来.
  // 同时应该取出了prefix部分.
  if (p == NULL || key_.size() < shared) {
    CorruptionError();
    return false;
  } else {
    key_.resize(shared);
    key_.append(p, non_shared); // 加入后面的部分
    value_ = Slice(p + non_shared, value_length);
    // 判断下一个restart point是否>=current_.
    while (restart_index_ + 1 < num_restarts_ &&
           GetRestartPoint(restart_index_ + 1) < current_) {
      ++restart_index_; // 否则需要进入下一个restart point.
    }
    return true;
  }
}
```

9. DecodeEntry

从头部decode出shared,non_shared以及value_length.这里面为了加快判断的话有一个技巧.返回的是下一个要读取的地址.

```
static inline const char* DecodeEntry(const char* p, const char* limit,
                                      uint32_t* shared,
                                      uint32_t* non_shared,
                                      uint32_t* value_length) {
  if (limit - p < 3) return NULL;
  *shared = reinterpret_cast<const unsigned char*>(p)[0];
  *non_shared = reinterpret_cast<const unsigned char*>(p)[1];
  *value_length = reinterpret_cast<const unsigned char*>(p)[2];
  // 至少存在3个字节.但是如果任意3个字节<128的话，表示每个部分都是1个字节.
  if ((*shared | *non_shared | *value_length) < 128) {
    // Fast path: all three values are encoded in one byte each
    p += 3;
  } else { // 如果没有这个fast path的话那么分别取出3个变长uint32.
    if ((p = GetVarint32Ptr(p, limit, shared)) == NULL) return NULL;
    if ((p = GetVarint32Ptr(p, limit, non_shared)) == NULL) return NULL;
    if ((p = GetVarint32Ptr(p, limit, value_length)) == NULL) return NULL;
  }

  if (static_cast<uint32_t>(limit - p) < (*non_shared + *value_length)) {
    return NULL;
  }
  return p;
}
```

10. Prev

Prev相对于Next有一点低效.首先遍历找到restart point,然后在这个restart range里面遍历.

```
virtual void Prev() {
  assert(Valid());

  // Scan backwards to a restart point before current_
  const uint32_t original = current_;
  while (GetRestartPoint(restart_index_) >= original) { // 首先找到restart range.
    if (restart_index_ == 0) {
      // No more entries
      current_ = restarts_;
      restart_index_ = num_restarts_;
      return;
    }
    restart_index_--;
  }

  SeekToRestartPoint(restart_index_); // 然后跳到这个restart range.
  do { // 在这个restart range里面遍历.
    // Loop until end of current entry hits the start of original entry
  } while (ParseNextKey() && NextEntryOffset() < original);
}
```

11. Seek

Seek非常简单，首先在restart point地方因为里面存放都是有序的完整的key.那么可以restart point这些地方进行二分查找.然后在restart range里面通过遍历查找.还算是比较高效吧。注意这里如果没有找到的话， 返回的是第一个>=target的对象.

```
virtual void Seek(const Slice& target) {
  // Binary search in restart array to find the first restart point
  // with a key >= target
  uint32_t left = 0;
  uint32_t right = num_restarts_ - 1;
  while (left < right) { // 在restart这些部分二分查找.
    uint32_t mid = (left + right + 1) / 2;
    uint32_t region_offset = GetRestartPoint(mid);
    uint32_t shared, non_shared, value_length;
    const char* key_ptr = DecodeEntry(data_ + region_offset,
                                      data_ + restarts_,
                                      &shared, &non_shared, &value_length);
    if (key_ptr == NULL || (shared != 0)) {
      CorruptionError();
      return;
    }
    Slice mid_key(key_ptr, non_shared);
    if (Compare(mid_key, target) < 0) {
      // Key at "mid" is smaller than "target".  Therefore all
      // blocks before "mid" are uninteresting.
      left = mid;
    } else {
      // Key at "mid" is >= "target".  Therefore all blocks at or
      // after "mid" are uninteresting.
      right = mid - 1;
    }
  }

  // Linear search (within restart block) for first key >= target
  SeekToRestartPoint(left); // 从这个部分开始遍历查找.
  while (true) {
    if (!ParseNextKey()) {
      return;
    }
    if (Compare(key_, target) >= 0) {
      return;
    }
  }
}
```

10.8.9 BlockHandle

table/format.h BlockHandle用于压缩和解压文件信息.包括两个字段offset和size.不是很麻烦

```
// BlockHandle is a pointer to the extent of a file that stores a data
// block or a meta block.
class BlockHandle {
 public:
  BlockHandle();

  // The offset of the block in the file.
  uint64_t offset() const { return offset_; }
  void set_offset(uint64_t offset) { offset_ = offset; }

  // The size of the stored block
  uint64_t size() const { return size_; }
  void set_size(uint64_t size) { size_ = size; }

  void EncodeTo(std::string* dst) const;
  Status DecodeFrom(Slice* input);

  // Maximum encoding length of a BlockHandle
  enum { kMaxEncodedLength = 10 + 10 }; // uint64最大压缩大小为10字节.

 private:
  uint64_t offset_;
  uint64_t size_;
};
inline BlockHandle::BlockHandle()
    : offset_(~static_cast<uint64_t>(0)), // 初始值非常大.
      size_(~static_cast<uint64_t>(0)) {
}
```

1. EncodeTo

```
void BlockHandle::EncodeTo(std::string* dst) const {
  // Sanity check that all fields have been set
  assert(offset_ != ~static_cast<uint64_t>(0));
  assert(size_ != ~static_cast<uint64_t>(0));
  PutVarint64(dst, offset_);
  PutVarint64(dst, size_);
}
```

2. DecodeFrom

```
Status BlockHandle::DecodeFrom(Slice* input) {
  if (GetVarint64(input, &offset_) &&
      GetVarint64(input, &size_)) {
    return Status::OK();
  } else {
    return Status::Corruption("bad block handle");
  }
}
```

10.8.10 Footer

db/format.h Footer里面打包称为定长信息.从注释上说保存在各个table文件末尾.现在还不知道具体有什么用途. (但是阅读完TableFormat这节之后应该知道这两个字段的含义).前面两个BlockHandle使用变长打包但是空出了MaxEncodedLength. 最后8个字节使用

```
// kTableMagicNumber was picked by running
//    echo http://code.google.com/p/leveldb/ | sha1sum
// and taking the leading 64 bits.
static const uint64_t kTableMagicNumber = 0xdb4775248b80fb57ull;
```

使用这个魔术数字比较有意思。我们来看看大致的结构.

```
// Footer encapsulates the fixed information stored at the tail
// end of every table file.
class Footer {
 public:
  Footer() { }

  // The block handle for the metaindex block of the table
  const BlockHandle& metaindex_handle() const { return metaindex_handle_; }
  void set_metaindex_handle(const BlockHandle& h) { metaindex_handle_ = h; }
  // The block handle for the index block of the table
  const BlockHandle& index_handle() const {
    return index_handle_;
  }
  void set_index_handle(const BlockHandle& h) {
    index_handle_ = h;
  }

  void EncodeTo(std::string* dst) const;
  Status DecodeFrom(Slice* input);

  // Encoded length of a Footer.  Note that the serialization of a
  // Footer will always occupy exactly this many bytes.  It consists
  // of two block handles and a magic number.
  enum {
    kEncodedLength = 2*BlockHandle::kMaxEncodedLength + 8 // 固定长度.
  };

 private:
  BlockHandle metaindex_handle_; // metaindex handle.
  BlockHandle index_handle_; // index handle.
};
```

1. EncodeTo

```
void Footer::EncodeTo(std::string* dst) const {
#ifndef NDEBUG
  const size_t original_size = dst->size();
#endif
  metaindex_handle_.EncodeTo(dst);
  index_handle_.EncodeTo(dst);
  dst->resize(2 * BlockHandle::kMaxEncodedLength);  // Padding
  PutFixed32(dst, static_cast<uint32_t>(kTableMagicNumber & 0xffffffffu));
  PutFixed32(dst, static_cast<uint32_t>(kTableMagicNumber >> 32));
  assert(dst->size() == original_size + kEncodedLength);
}
```

2. DecodeFrom

   这里input作为输入，头部是EncodeTo的内容后面可能带有其他数据。解析完成之后将剩余返回给input.

```
Status Footer::DecodeFrom(Slice* input) {
  const char* magic_ptr = input->data() + kEncodedLength - 8;
  const uint32_t magic_lo = DecodeFixed32(magic_ptr);
  const uint32_t magic_hi = DecodeFixed32(magic_ptr + 4);
  const uint64_t magic = ((static_cast<uint64_t>(magic_hi) << 32) |
                          (static_cast<uint64_t>(magic_lo)));
  if (magic != kTableMagicNumber) {
    return Status::InvalidArgument("not an sstable (bad magic number)");
  }

  Status result = metaindex_handle_.DecodeFrom(input);
  if (result.ok()) {
    result = index_handle_.DecodeFrom(input);
  }
  if (result.ok()) {
    // We skip over any leftover data (just padding for now) in "input"
    const char* end = magic_ptr + 8;
    *input = Slice(end, input->data() + input->size() - end);
  }
  return result;
}
```

10.8.11 ReadBlock

table/format.cc ReadBlock应该是从随机文件里面读取Block出来.对于这个Block的位置的话由handle提供. 注意每个Block后面还有crc和type.这个可以参考TableBuilder实现。然后将读取的Block二进制构造Block对象返回。

```
// 1-byte type + 32-bit crc
// type表示使用什么压缩方式.
static const size_t kBlockTrailerSize = 5;
```

下面看看具体实现代码

```
Status ReadBlock(RandomAccessFile* file,
                 const ReadOptions& options,
                 const BlockHandle& handle,
                 Block** block) {
  *block = NULL;

  // Read the block contents as well as the type/crc footer.
  // See table_builder.cc for the code that built this structure.
  size_t n = static_cast<size_t>(handle.size());
  char* buf = new char[n + kBlockTrailerSize];
  Slice contents;
  Status s = file->Read(handle.offset(), n + kBlockTrailerSize, &contents, buf);
  if (!s.ok()) {
    delete[] buf;
    return s;
  }
  if (contents.size() != n + kBlockTrailerSize) {
    delete[] buf;
    return Status::Corruption("truncated block read");
  }

  // 校验CRC32C
  // Check the crc of the type and the block contents
  const char* data = contents.data();    // Pointer to where Read put the data
  if (options.verify_checksums) {
    const uint32_t crc = crc32c::Unmask(DecodeFixed32(data + n + 1));
    const uint32_t actual = crc32c::Value(data, n + 1);
    if (actual != crc) {
      delete[] buf;
      s = Status::Corruption("block checksum mismatch");
      return s;
    }
  }

  // 根据type判断使用什么压缩方式.
  switch (data[n]) {
    case kNoCompression:
      if (data != buf) {
        // File implementation gave us pointer to some other data.
        // Copy into buf[].
        memcpy(buf, data, n + kBlockTrailerSize);
      }

      // Ok
      break;
    case kSnappyCompression: {
      // ... // 这里我故意省去了.我们这里暂时不关系这个逻辑
      // 使用Snappy来解压缩.
      break;
    }
    default:
      delete[] buf;
      return Status::Corruption("bad block type");
  }

  // 返回Block对象.
  // Block对象可以通过迭代器来进行访问.
  *block = new Block(buf, n);  // Block takes ownership of buf[]
  return Status::OK();
}
```

10.8.12 IteratorWrapper

table/iterator_wrapper.h IteratorWrapper就是Iterator的装饰者。对于Valid,Key进行了缓存. 这个其实还是比较有必要的.因为有些iterator取key和valid代价比较大.不过似乎对于虚函数没有节省开销， 因为内部存放的还是Iterator对象而不是模板。阅读完了TwoLevelIterator就会发现，这个类存在原因， 主要是为了管理Iterator对象的管理。比如Set的时候会将原来的Iterator对象释放，析构函数会将持有的Iterator释放。

```
// A internal wrapper class with an interface similar to Iterator that
// caches the valid() and key() results for an underlying iterator.
// This can help avoid virtual function calls and also gives better
// cache locality.
```

10.8.13 EmptyIterator

EmptyIterator用来构造返回错误或者是空的Iterator.提供了两个函数

- NewEmptyIterator
- NewErrorIterator

来进行构造。具体实现放在了table/iterator.cc里面

```
class EmptyIterator : public Iterator {
 public:
  EmptyIterator(const Status& s) : status_(s) { }
  virtual bool Valid() const { return false; }
  virtual void Seek(const Slice& target) { }
  virtual void SeekToFirst() { }
  virtual void SeekToLast() { }
  virtual void Next() { assert(false); }
  virtual void Prev() { assert(false); }
  Slice key() const { assert(false); return Slice(); }
  Slice value() const { assert(false); return Slice(); }
  virtual Status status() const { return status_; }
 private:
  Status status_;
};
```

10.8.14 TwoLevelIterator

table/two_level_iterator.h TwoLevelIterator是一个二级Iterator配合sstable二级索引文件使用的。 通过工厂方法来进行构造所以接口非常简单。主要还是看TwoLevelIterator里面的实现。

```
// Return a new two level iterator.  A two-level iterator contains an
// index iterator whose values point to a sequence of blocks where
// each block is itself a sequence of key,value pairs.  The returned
// two-level iterator yields the concatenation of all key/value pairs
// in the sequence of blocks.  Takes ownership of "index_iter" and
// will delete it when no longer needed.
//
// Uses a supplied function to convert an index_iter value into
// an iterator over the contents of the corresponding block.
extern Iterator* NewTwoLevelIterator(
    Iterator* index_iter, // index block iterator.
    Iterator* (*block_function)(
        void* arg,
        const ReadOptions& options,
        const Slice& index_value),
    void* arg,
    const ReadOptions& options);
```

1. NewTwoLevelIterator

```
Iterator* NewTwoLevelIterator(
    Iterator* index_iter,
    BlockFunction block_function,
    void* arg,
    const ReadOptions& options) {
  // 非常简单就是创建对象.
  return new TwoLevelIterator(index_iter, block_function, arg, options);
}
```

然后我们看看TwoLevelIterator的结构以及构造函数实现。接下来我们看看几个主要的接口。

```
typedef Iterator* (*BlockFunction)(void*, const ReadOptions&, const Slice&);

class TwoLevelIterator: public Iterator {
 private:
  BlockFunction block_function_;
  void* arg_;
  const ReadOptions options_;
  Status status_;
  IteratorWrapper index_iter_;
  // 对于key,value这样的方法直接给data_iter_代理即可.
  IteratorWrapper data_iter_; // May be NULL
  // If data_iter_ is non-NULL, then "data_block_handle_" holds the
  // "index_value" passed to block_function_ to create the data_iter_.
  std::string data_block_handle_;
};

TwoLevelIterator::TwoLevelIterator(
    Iterator* index_iter,
    BlockFunction block_function,
    void* arg,
    const ReadOptions& options)
    : block_function_(block_function),
      arg_(arg),
      options_(options),
      index_iter_(index_iter),
      data_iter_(NULL) {
}
```

2. Seek

```
void TwoLevelIterator::Seek(const Slice& target) {
  index_iter_.Seek(target); // 首先通过index iter进行定位.
  // 注意这里可能没有找到。BlockIterator::Seek行为是找到>=target对象.
  InitDataBlock(); // 初始化data_iter.
  if (data_iter_.iter() != NULL) data_iter_.Seek(target); // 使用data_iter定位.
  SkipEmptyDataBlocksForward(); // 向前略过空白记录.
}
```

3. SeekToFirst

```
void TwoLevelIterator::SeekToFirst() {
  index_iter_.SeekToFirst(); // 首先index inter.
  InitDataBlock();
  if (data_iter_.iter() != NULL) data_iter_.SeekToFirst(); // 然后data iter.
  SkipEmptyDataBlocksForward(); // 向前跳过空白记录.
}
```

4. SeekToLast

```
void TwoLevelIterator::SeekToLast() {
  index_iter_.SeekToLast(); // 首先index iter.
  InitDataBlock();
  if (data_iter_.iter() != NULL) data_iter_.SeekToLast(); // 然后data iter.
  SkipEmptyDataBlocksBackward(); // 向后跳过空白记录.
}
```

5. Next

```
void TwoLevelIterator::Next() {
  assert(Valid());
  data_iter_.Next();
  SkipEmptyDataBlocksForward(); // 向前跳过空白记录.这个部分可能需要考虑data_iter_失效.
}
```

6. Prev

```
void TwoLevelIterator::Prev() {
  assert(Valid());
  data_iter_.Prev();
  SkipEmptyDataBlocksBackward(); // 向后跳过空白记录.这个部分可能需要考虑data_iter_失效.
}
```

7. InitDataBlock

```
void TwoLevelIterator::InitDataBlock() {
  if (!index_iter_.Valid()) {
    SetDataIterator(NULL); // 如果index_iter已经失效的话.
  } else {
    Slice handle = index_iter_.value();
    if (data_iter_.iter() != NULL && handle.compare(data_block_handle_) == 0) {
      // data_iter_ is already constructed with this iterator, so
      // no need to change anything
    } else { // 调用block_function来读取具体的data block.
      // 这里可以看到,block_function的用途就是读取handle
      // 其中handle里面是对应data block所在的位置。
      Iterator* iter = (*block_function_)(arg_, options_, handle);
      // 设置data block handle.:).
      data_block_handle_.assign(handle.data(), handle.size());
      SetDataIterator(iter);
    }
  }
}
```

8. SetDataIterator

```
void TwoLevelIterator::SetDataIterator(Iterator* data_iter) {
  if (data_iter_.iter() != NULL) SaveError(data_iter_.status());
  data_iter_.Set(data_iter); // 注意这里IteratorWrapper进行Set的话原来会释放.
  // 这也解释了为什么我们需要IteratorWrapper对象存在.
}
```

9. SkipEmptyDataBlocksForward

   对于空的DataBlock的话那么!data_iter_.Valid().对于空的DataBlock可以跳过IndexBlock快速跨越。

```
void TwoLevelIterator::SkipEmptyDataBlocksForward() {
  while (data_iter_.iter() == NULL || !data_iter_.Valid()) {
    // Move to next block
    if (!index_iter_.Valid()) {
      SetDataIterator(NULL);
      return;
    }
    index_iter_.Next();
    InitDataBlock();
    if (data_iter_.iter() != NULL) data_iter_.SeekToFirst();
  }
}
```

10. SkipEmptyDataBlocksBackward

```
void TwoLevelIterator::SkipEmptyDataBlocksBackward() {
  while (data_iter_.iter() == NULL || !data_iter_.Valid()) {
    // Move to next block
    if (!index_iter_.Valid()) {
      SetDataIterator(NULL);
      return;
    }
    index_iter_.Prev();
    InitDataBlock();
    if (data_iter_.iter() != NULL) data_iter_.SeekToLast();
  }
}
```

10.8.15 MergingIterator

table/merger.cc 可以认为这个是一个Iterator的多路归并实现。但是巧妙的是将多路Iterator 归并称为一个Iterator进行遍历。从注释上面可以看到对于key不会进行去重。依然是一个工厂方法。

```
// Return an iterator that provided the union of the data in
// children[0,n-1].  Takes ownership of the child iterators and
// will delete them when the result iterator is deleted.
//
// The result does no duplicate suppression.  I.e., if a particular
// key is present in K child iterators, it will be yielded K times.
//
// REQUIRES: n >= 0
extern Iterator* NewMergingIterator(
    const Comparator* comparator, Iterator** children, int n);
```

1. NewMergingIterator

```
// 注意这里我们需要托管list里面的Iterator对象.
Iterator* NewMergingIterator(const Comparator* cmp, Iterator** list, int n) {
  assert(n >= 0);
  if (n == 0) {
    return NewEmptyIterator(); // 对于n==0那么是empty iterator.
  } else if (n == 1) {
    return list[0]; // 如果n==1的话那么只需要返回第一个即可.
    // 这里并没有问题因为我们最后会直接delete这个对象的。
  } else {
    return new MergingIterator(cmp, list, n); // 否则就需要进行合并.
  }
}
```

2. MergingIterator

   首先看看结构以及构造和析构函数.

```
class MergingIterator : public Iterator {
 public:
  MergingIterator(const Comparator* comparator, Iterator** children, int n)
      : comparator_(comparator),
        children_(new IteratorWrapper[n]),
        n_(n),
        current_(NULL),
        direction_(kForward) {
    for (int i = 0; i < n; i++) {
      children_[i].Set(children[i]);
    }
  }

  virtual ~MergingIterator() {
    delete[] children_;
  }

 private:
  // We might want to use a heap in case there are lots of children.
  // For now we use a simple array since we expect a very small number
  // of children in leveldb.
  const Comparator* comparator_; // 比较器.
  IteratorWrapper* children_; // 多路Iterator.
  int n_; // 多少路.
  IteratorWrapper* current_; // 当前Iterator.

  // Which direction is the iterator moving?
  enum Direction {
    kForward,
    kReverse
  };
  Direction direction_; // 方向.
};
```

3. Key

```
virtual Slice key() const {
  assert(Valid());
  return current_->key();
}
```

4. Value

```
virtual Slice value() const {
  assert(Valid());
  return current_->value();
}
```

5. Seek

```
virtual void Seek(const Slice& target) {
  for (int i = 0; i < n_; i++) { // 每个children都seek到这个target.
    children_[i].Seek(target);
  }
  FindSmallest(); // 然后对比一下最小的对象.设置为current.
  direction_ = kForward; // 设置一下方向.
  // 对于Key,Value的话调用current这个Iterator.
}
```

6. SeekToFirst

```
virtual void SeekToFirst() {
  for (int i = 0; i < n_; i++) {
    children_[i].SeekToFirst();
  }
  FindSmallest();
  direction_ = kForward;
}
```

7. SeekToLast

```
virtual void SeekToLast() {
  for (int i = 0; i < n_; i++) {
    children_[i].SeekToLast();
  }
  FindLargest(); // 找到最大的作为current_.
  direction_ = kReverse;
}
```

8. Next

```
virtual void Next() {
  assert(Valid());

  // 如果不是正方向的话.不过觉得这个部分代码可能存在问题.
  // 不过也没有更好的解决办法了。我觉得这里直接assert就好了，不需要调整。
  // Ensure that all children are positioned after key().
  // If we are moving in the forward direction, it is already
  // true for all of the non-current_ children since current_ is
  // the smallest child and key() == current_->key().  Otherwise,
  // we explicitly position the non-current_ children.
  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();
}
```

9. Prev

```
virtual void Prev() {
  assert(Valid());

  // 如果不是反方向的话.不过觉得这个部分代码可能存在问题.
  // 不过也没有更好的解决办法了。我觉得这里直接assert就好了，根本不需要调整。
  // Ensure that all children are positioned before key().
  // If we are moving in the reverse direction, it is already
  // true for all of the non-current_ children since current_ is
  // the largest child and key() == current_->key().  Otherwise,
  // we explicitly position the non-current_ children.
  if (direction_ != kReverse) {
    for (int i = 0; i < n_; i++) {
      IteratorWrapper* child = &children_[i];
      if (child != current_) {
        child->Seek(key());
        if (child->Valid()) {
          // Child is at first entry >= key().  Step back one to be < key()
          child->Prev();
        } else {
          // Child has no entries >= key().  Position at last entry.
          child->SeekToLast();
        }
      }
    }
    direction_ = kReverse;
  }

  current_->Prev();
  FindLargest();
}
```

10. FindSmallest

从这些Iterator找到最小的Iterator作为current.算法naive.

```
void MergingIterator::FindSmallest() {
  IteratorWrapper* smallest = NULL;
  for (int i = 0; i < n_; i++) {
    IteratorWrapper* child = &children_[i];
    if (child->Valid()) {
      if (smallest == NULL) {
        smallest = child;
      } else if (comparator_->Compare(child->key(), smallest->key()) < 0) {
        smallest = child;
      }
    }
  }
  current_ = smallest;
}
```

11. FindLargest

    从这些Iterator里面找到最大的Iterator作为current.算法naive.

```
void MergingIterator::FindLargest() {
  IteratorWrapper* largest = NULL;
  for (int i = n_-1; i >= 0; i--) {
    IteratorWrapper* child = &children_[i];
    if (child->Valid()) {
      if (largest == NULL) {
        largest = child;
      } else if (comparator_->Compare(child->key(), largest->key()) > 0) {
        largest = child;
      }
    }
  }
  current_ = largest;
}
```

10.9 Util

10.9.1 Arena

util/arena.h[.cc] arena作为一个局部,对于逻辑来说存在生命周期的内存分配器.所有的内存都在这上面分配然后一次性释放.

```
class Arena {
 public:
  Arena();
  ~Arena();

  // Return a pointer to a newly allocated memory block of "bytes" bytes.
  char* Allocate(size_t bytes);

  // Allocate memory with the normal alignment guarantees provided by malloc
  char* AllocateAligned(size_t bytes);

  // Returns an estimate of the total memory usage of data allocated
  // by the arena (including space allocated but not yet used for user
  // allocations).
  size_t MemoryUsage() const { // 当前分配内存以及std::vector分配内存.
    return blocks_memory_ + blocks_.capacity() * sizeof(char*);
  }

 private:
  char* AllocateFallback(size_t bytes); // 如果当前的block不够分配的话，那么需要new一个新的block.
  char* AllocateNewBlock(size_t block_bytes); // new新的block逻辑.

  // Allocation state
  char* alloc_ptr_; // 当前允许分配的ptr
  size_t alloc_bytes_remaining_; // 当前还有连续内存地址还有多少没有分配掉

  // Array of new[] allocated memory blocks
  std::vector<char*> blocks_; // 现在分配了多少连续内存地址

  // Bytes of memory in blocks allocated so far
  size_t blocks_memory_; // 当前分配了多少内存.
};
```

阅读完了这个接口基本上就能够猜想到怎么实现了.都是基于sample方式的内存分配,每次分配固定大小的block 然后在上面不断地进行切分。但是这里必须确保block内存大小足够大，不然不能够正常分配内存. (#note: 不过阅读了后面实现，发现如果超过block的话按照本身大小分配，不会存在问题).

还是稍微看看实现吧

```
inline char* Arena::Allocate(size_t bytes) {
  // The semantics of what to return are a bit messy if we allow
  // 0-byte allocations, so we disallow them here (we don't need
  // them for our internal use).
  assert(bytes > 0);
  if (bytes <= alloc_bytes_remaining_) { // 查看当前是否可以分配.
    char* result = alloc_ptr_;
    alloc_ptr_ += bytes;
    alloc_bytes_remaining_ -= bytes;
    return result;
  }
  return AllocateFallback(bytes);
}

static const int kBlockSize = 4096; // block size是4K.

Arena::Arena() {
  blocks_memory_ = 0;
  alloc_ptr_ = NULL;  // First allocation will allocate a block
  alloc_bytes_remaining_ = 0;
}

Arena::~Arena() {
  for (size_t i = 0; i < blocks_.size(); i++) {
    delete[] blocks_[i];
  }
}

char* Arena::AllocateFallback(size_t bytes) {
  if (bytes > kBlockSize / 4) { // 如果分配超过1K.那么直接分配占用.
    // Object is more than a quarter of our block size.  Allocate it separately
    // to avoid wasting too much space in leftover bytes.
    char* result = AllocateNewBlock(bytes);
    return result;
  }

  // We waste the remaining space in the current block.
  alloc_ptr_ = AllocateNewBlock(kBlockSize); // 否则会分配出来然后在这上面进行切分.
  alloc_bytes_remaining_ = kBlockSize;

  char* result = alloc_ptr_;
  alloc_ptr_ += bytes;
  alloc_bytes_remaining_ -= bytes;
  return result;
}

char* Arena::AllocateAligned(size_t bytes) {
  const int align = sizeof(void*);    // We'll align to pointer size
  assert((align & (align-1)) == 0);   // Pointer size should be a power of 2
  size_t current_mod = reinterpret_cast<uintptr_t>(alloc_ptr_) & (align-1); // 需要考虑当前地址是否对其.
  size_t slop = (current_mod == 0 ? 0 : align - current_mod);
  size_t needed = bytes + slop; // 如果没有对齐我们可能需要做的偏移.
  char* result;
  if (needed <= alloc_bytes_remaining_) {
    result = alloc_ptr_ + slop;
    alloc_ptr_ += needed;
    alloc_bytes_remaining_ -= needed;
  } else {
    // AllocateFallback always returned aligned memory
    result = AllocateFallback(bytes);
  }
  assert((reinterpret_cast<uintptr_t>(result) & (align-1)) == 0);
  return result;
}

char* Arena::AllocateNewBlock(size_t block_bytes) {
  char* result = new char[block_bytes];
  blocks_memory_ += block_bytes;
  blocks_.push_back(result);
  return result;
}
```

10.9.2 Coding

util/coding.h[.cc] Coding主要是用来完成变长数据以及字符串还有Slice的序列化和反序列化的.主要方法包括下面这些

- PutFixed32 写入定长uint32

- PutFixed64 写入定长uint64
- PutVarint32 写入变长uint32
- PutVarint64 写入变长uint64
- PutLengthPrefixedSlice 后面解释.
- GetVarint32 读取变长uint32
- GetVarint64 读取变长uint64
- GetLengthPrefixedSlice 后面解释.
- GetVarint32Ptr(p,limit,v) 将[p,limit)部分的内存解析为变长uint32放到v里面,返回下一个字节
- GetVarint64Ptr(p,limit,v) 将[p,limit)部分的内存解析为变长uint64放到v里面,返回下一个字节
- VarintLength 变长uint32/uint64长度
- EncodeFixed32 PutFixed32 low-level
- EncodeFixed64 PutFixed64 low-level
- EncodeVarint32 PutVarint32 low-level
- EncodeVarint64 PutVarint64 low-level
- DecodeFixed32 读取定长uint32
- DecodeFixed64 读取定长uint64
- GetVarint32PtrFallback
- GetVarint32Ptr 从[p,limit)读取uint32并且返回下一个字节.

编码方式有点类似于google::protobuf里面的变长整数打包。我们这里着重看看下面两个函数

```
void PutLengthPrefixedSlice(std::string* dst, const Slice& value) {
  PutVarint32(dst, value.size());
  dst->append(value.data(), value.size());
}
bool GetLengthPrefixedSlice(Slice* input, Slice* result) {
  uint32_t len;
  if (GetVarint32(input, &len) &&
      input->size() >= len) {
    *result = Slice(input->data(), len);
    input->remove_prefix(len);
    return true;
  } else {
    return false;
  }
}
```

PutLengthPrefixedSlice对于slice的存储，首先是放入uint32长度然后放入内容. GetLengthPrefixedSlice从input首先取出长度，然后取出数据,并且将input跳过这些长度.

10.9.3 Histogram

todo:

10.9.4 SkipList

db/skiplist.h SkipList(跳表)之前在课本上看到过，但是当时觉得实在是没有太大的用途。现在仔细看看吧. 这里的SkipList附带了一个iterator.首先看看SkipList的结构.SkipList对于写的话是需要外部进行同步的， 对于读的话可以是多个读同时发起。

```
template<typename Key, class Comparator>
class SkipList {
 private:
  struct Node;

 public:
  // Create a new SkipList object that will use "cmp" for comparing keys,
  // and will allocate memory using "*arena".  Objects allocated in the arena
  // must remain allocated for the lifetime of the skiplist object.
  explicit SkipList(Comparator cmp, Arena* arena);

  // Insert key into the list.
  // REQUIRES: nothing that compares equal to key is currently in the list.
  void Insert(const Key& key);
  // Returns true iff an entry that compares equal to key is in the list.
  bool Contains(const Key& key) const;

  // Iteration over the contents of a skip list
  class Iterator {
   public:
    // Initialize an iterator over the specified list.
    // The returned iterator is not valid.
    explicit Iterator(const SkipList* list);

    // Returns true iff the iterator is positioned at a valid node.
    bool Valid() const;

    // Returns the key at the current position.
    // REQUIRES: Valid()
    const Key& key() const;

    // Advances to the next position.
    // REQUIRES: Valid()
    void Next();

    // Advances to the previous position.
    // REQUIRES: Valid()
    void Prev();

    // Advance to the first entry with a key >= target
    void Seek(const Key& target);

    // Position at the first entry in list.
    // Final state of iterator is Valid() iff list is not empty.
    void SeekToFirst();

    // Position at the last entry in list.
    // Final state of iterator is Valid() iff list is not empty.
    void SeekToLast();

   private:
    const SkipList* list_;
    Node* node_;
    // Intentionally copyable
  };

 private:
  enum { kMaxHeight = 12 }; // 跳表最大高度是12.

  // Immutable after construction
  Comparator const compare_;
  Arena* const arena_;    // Arena used for allocations of nodes

  Node* const head_;

  // Modified only by Insert().  Read racily by readers, but stale
  // values are ok.
  port::AtomicPointer max_height_;   // Height of the entire list

  inline int GetMaxHeight() const { // 当前最大的skiplist高度.
    return reinterpret_cast<intptr_t>(max_height_.NoBarrier_Load());
  }

  // Read/written only by Insert().
  Random rnd_;

  Node* NewNode(const Key& key, int height);
  int RandomHeight();
  bool Equal(const Key& a, const Key& b) const { return (compare_(a, b) == 0); }

  // Return true if key is greater than the data stored in "n"
  bool KeyIsAfterNode(const Key& key, Node* n) const;

  // Return the earliest node that comes at or after key.
  // Return NULL if there is no such node.
  //
  // If prev is non-NULL, fills prev[level] with pointer to previous
  // node at "level" for every level in [0..max_height_-1].
  Node* FindGreaterOrEqual(const Key& key, Node** prev) const;

  // Return the latest node with a key < key.
  // Return head_ if there is no such node.
  Node* FindLessThan(const Key& key) const;

  // Return the last node in the list.
  // Return head_ if list is empty.
  Node* FindLast() const;
};
```

1. RandomHeight

   随机产生一个SkipList的高度.从1开始每次以1/4的概率+1.

   - h=1 3/4
   - h=2 3/16
   - ...

   主要就是有这么一个大概的概率就是,随机初始化的高度不会很大.

```
template<typename Key, class Comparator>
int SkipList<Key,Comparator>::RandomHeight() {
  // Increase height with probability 1 in kBranching
  static const unsigned int kBranching = 4;
  int height = 1;
  while (height < kMaxHeight && ((rnd_.Next() % kBranching) == 0)) {
    height++;
  }
  assert(height > 0);
  assert(height <= kMaxHeight);
  return height;
}
```

2. KeyIsAfterNode

   可以认为是key的比较函数吧.拿key和当前某一个node进行比较，看看这个key是否应该存在这个node之后.

```
template<typename Key, class Comparator>
bool SkipList<Key,Comparator>::KeyIsAfterNode(const Key& key, Node* n) const {
  // NULL n is considered infinite
  return (n != NULL) && (compare_(n->key, key) < 0); // 内部的comparator.
}
```

3. FindGreaterOrEqual

   找到和key相当或者是>key的第一个node.对跳表的原理稍微熟悉一些大概就可以看懂代码了.

```
template<typename Key, class Comparator>
typename SkipList<Key,Comparator>::Node* SkipList<Key,Comparator>::FindGreaterOrEqual(const Key& key, Node** prev)
    const {
  Node* x = head_;
  int level = GetMaxHeight() - 1; // 按照最高层的跳表查找
  while (true) {
    Node* next = x->Next(level);
    if (KeyIsAfterNode(key, next)) { // 如果key在这个后面的话,那么继续按照这个高度.
      // Keep searching in this list
      x = next;
    } else {
      if (prev != NULL) prev[level] = x; // 记录这个prev.
      if (level == 0) { // 如果到了最下层的话.
        return next; // 注意这里的返回值不一定Equal(key)
      } else {
        // Switch to next list
        level--; // 否则level--.
      }
    }
  }
}
```

4. Insert

   插入某一个key.恩,逻辑还算是比较简单吧

```
template<typename Key, class Comparator>
void SkipList<Key,Comparator>::Insert(const Key& key) {
  // TODO(opt): We can use a barrier-free variant of FindGreaterOrEqual()
  // here since Insert() is externally synchronized.
  Node* prev[kMaxHeight];
  Node* x = FindGreaterOrEqual(key, prev); // 查找到key的previous的内容.

  // Our data structure does not allow duplicate insertion
  assert(x == NULL || !Equal(key, x->key));
  int height = RandomHeight();
  if (height > GetMaxHeight()) { // 随机产生高度.:我在想，是不是只有这个地方才会改变高度呢?.
    for (int i = GetMaxHeight(); i < height; i++) {
      prev[i] = head_; // 如果没有的话，那么初始化为head_.
    }
    //fprintf(stderr, "Change height from %d to %d\n", max_height_, height);

    // It is ok to mutate max_height_ without any synchronization
    // with concurrent readers.  A concurrent reader that observes
    // the new value of max_height_ will see either the old value of
    // new level pointers from head_ (NULL), or a new value set in
    // the loop below.  In the former case the reader will
    // immediately drop to the next level since NULL sorts after all
    // keys.  In the latter case the reader will use the new node.
    max_height_.NoBarrier_Store(reinterpret_cast<void*>(height));
  }

  x = NewNode(key, height); // 产生一个新的节点.
  for (int i = 0; i < height; i++) { // 然后插入维护好这个跳表结构.
    // NoBarrier_SetNext() suffices since we will add a barrier when
    // we publish a pointer to "x" in prev[i].
    x->NoBarrier_SetNext(i, prev[i]->NoBarrier_Next(i));
    prev[i]->SetNext(i, x);
  }
}
```

5. Contains

Contains用来判断跳表里面是否包含key

```
template<typename Key, class Comparator>
bool SkipList<Key,Comparator>::Contains(const Key& key) const {
  Node* x = FindGreaterOrEqual(key, NULL);
  if (x != NULL && Equal(key, x->key)) {
    return true; // 如果!=NULL并且Equal的话那么返回true.
  } else {
    return false;
  }
}
```

6. FindLessThan

查找最后面一个less than key的这个node.

```
template<typename Key, class Comparator>
typename SkipList<Key,Comparator>::Node*
SkipList<Key,Comparator>::FindLessThan(const Key& key) const {
  Node* x = head_;
  int level = GetMaxHeight() - 1;
  while (true) {
    assert(x == head_ || compare_(x->key, key) < 0);
    Node* next = x->Next(level);
    if (next == NULL || compare_(next->key, key) >= 0) {
      if (level == 0) {
        return x;
      } else {
        // Switch to next list
        level--;
      }
    } else {
      x = next;
    }
  }
}
```

7. FindLast

FindLast就是找到元素的最后一个节点.非常简单，首先在最高节点遍历然后不断地降低level遍历.

```
template<typename Key, class Comparator>
typename SkipList<Key,Comparator>::Node* SkipList<Key,Comparator>::FindLast()
    const {
  Node* x = head_;
  int level = GetMaxHeight() - 1;
  while (true) {
    Node* next = x->Next(level);
    if (next == NULL) {
      if (level == 0) {
        return x;
      } else {
        // Switch to next list
        level--;
      }
    } else {
      x = next;
    }
  }
}
```

8. Node

在看下面的函数之前我们先看看Node的实现.其实非常简单，而且似乎没有必要使用atomic的方式来操作。 因为这里面我们已经要求写需要外部加锁保证了.(#note: 后来想想还是有必要的,这里我们只是要求用户 在写的时候使用外部同步机保证，而读的时候没有。那么这样必须强制读取内存才可以保证正确,所以这里 需要Acquire_Load以及Release_Load).非常简单，就是持有key以及跳表指针.

```
template<typename Key, class Comparator>
struct SkipList<Key,Comparator>::Node {
  explicit Node(const Key& k) : key(k) { }

  Key const key;

  // Accessors/mutators for links.  Wrapped in methods so we can
  // add the appropriate barriers as necessary.
  Node* Next(int n) {
    assert(n >= 0);
    // Use an 'acquire load' so that we observe a fully initialized
    // version of the returned Node.
    return reinterpret_cast<Node*>(next_[n].Acquire_Load());
  }
  void SetNext(int n, Node* x) {
    assert(n >= 0);
    // Use a 'release store' so that anybody who reads through this
    // pointer observes a fully initialized version of the inserted node.
    next_[n].Release_Store(x);
  }

  // No-barrier variants that can be safely used in a few locations.
  Node* NoBarrier_Next(int n) {
    assert(n >= 0);
    return reinterpret_cast<Node*>(next_[n].NoBarrier_Load());
  }
  void NoBarrier_SetNext(int n, Node* x) {
    assert(n >= 0);
    next_[n].NoBarrier_Store(x);
  }

 private:
  // Array of length equal to the node height.  next_[0] is lowest level link.
  port::AtomicPointer next_[1]; // 至少有一个next指针.
};
```

9. NewNode

前面看完了Node的结构.然后Node分配就非常简单了.这里采用了inplacement new的方式来分配内存

```
template<typename Key, class Comparator>
typename SkipList<Key,Comparator>::Node*
SkipList<Key,Comparator>::NewNode(const Key& key, int height) {
  char* mem = arena_->AllocateAligned(
      sizeof(Node) + sizeof(port::AtomicPointer) * (height - 1));
  return new (mem) Node(key);
}
```

注意这里只需要分配height-1的高度即可，因为在Node里面已经存在了1个链表指针.

10. Iterator

了解了跳表的工作原理之后，对于其遍历器理解就非常简单了。代码也非常简单。但是阅读代码就会发现， 跳表的前向遍历功能不怎么样，因为没有维护prev指针。这里prev的方法是用过调用FindLessThan来实现的。

```
template<typename Key, class Comparator>
```

```
inline SkipList<Key,Comparator>::Iterator::Iterator(const SkipList* list) {
  list_ = list;
  node_ = NULL;
}

template<typename Key, class Comparator>
inline bool SkipList<Key,Comparator>::Iterator::Valid() const {
  return node_ != NULL;
}

template<typename Key, class Comparator>
inline const Key& SkipList<Key,Comparator>::Iterator::key() const {
  assert(Valid());
  return node_->key;
}

template<typename Key, class Comparator>
inline void SkipList<Key,Comparator>::Iterator::Next() {
  assert(Valid());
  node_ = node_->Next(0);
}

template<typename Key, class Comparator>
inline void SkipList<Key,Comparator>::Iterator::Prev() {
  // Instead of using explicit "prev" links, we just search for the
  // last node that falls before key.
  assert(Valid());
  node_ = list_->FindLessThan(node_->key); // 可能性能会存在问题.所以最好不要前向遍历.
  if (node_ == list_->head_) {
    node_ = NULL;
  }
}

template<typename Key, class Comparator>
inline void SkipList<Key,Comparator>::Iterator::Seek(const Key& target) {
  node_ = list_->FindGreaterOrEqual(target, NULL);
}

template<typename Key, class Comparator>
inline void SkipList<Key,Comparator>::Iterator::SeekToFirst() {
  node_ = list_->head_->Next(0);
}

template<typename Key, class Comparator>
inline void SkipList<Key,Comparator>::Iterator::SeekToLast() {
  node_ = list_->FindLast();
  if (node_ == list_->head_) {
    node_ = NULL;
  }
}
```

10.9.5 AtomicPointer

port/atomic_pointer.h 指针的存取都是原子操作.对于pointer来说，不管是32位还是64位的话都是原子操作的。但是这里必须考虑内存屏障。 对于leveldb还考虑到c++0x的原子操作库cstdatomic本身所提供的功能。我们都看看这个是如何实现的。

对于没有cstdatomic并且是x86CPU的话.

```
// AtomicPointer built using platform-specific MemoryBarrier()

// Gcc on x86
#elif defined(ARCH_CPU_X86_FAMILY) && defined(__GNUC__)
inline void MemoryBarrier() {
  // See http://gcc.gnu.org/ml/gcc/2003-04/msg01180.html for a discussion on
  // this idiom. Also see http://en.wikipedia.org/wiki/Memory_ordering.
  __asm__ __volatile__("" : : : "memory");
}

#if defined(LEVELDB_HAVE_MEMORY_BARRIER)
class AtomicPointer {
 private:
  void* rep_;
 public:
  AtomicPointer() { }
  explicit AtomicPointer(void* p) : rep_(p) {}
  inline void* NoBarrier_Load() const { return rep_; }
  inline void NoBarrier_Store(void* v) { rep_ = v; }
  inline void* Acquire_Load() const {
    void* result = rep_;
    MemoryBarrier();
    return result;
  }
  inline void Release_Store(void* v) {
    MemoryBarrier();
    rep_ = v;
  }
};
```

对于有cstdatomic库的话.

```
#elif defined(LEVELDB_CSTDATOMIC_PRESENT)
class AtomicPointer {
 private:
  std::atomic<void*> rep_;
 public:
  AtomicPointer() { }
  explicit AtomicPointer(void* v) : rep_(v) { }
  inline void* Acquire_Load() const {
    return rep_.load(std::memory_order_acquire); // memory order acquire
  }
  inline void Release_Store(void* v) {
    rep_.store(v, std::memory_order_release); // memory order release.
  }
  inline void* NoBarrier_Load() const {
    return rep_.load(std::memory_order_relaxed); // memory order relaxed.
  }
  inline void NoBarrier_Store(void* v) {
    rep_.store(v, std::memory_order_relaxed);
  }
};
```

10.9.6 CRC32C

util/crc32c.cc CRC32C算法实现.注意这里Extend的第一个参数是初始值.

```
// Return the crc32c of concat(A, data[0,n-1]) where init_crc is the
// crc32c of some string A.  Extend() is often used to maintain the
// crc32c of a stream of data.
extern uint32_t Extend(uint32_t init_crc, const char* data, size_t n);

// Return the crc32c of data[0,n-1]
inline uint32_t Value(const char* data, size_t n) {
  return Extend(0, data, n);
}
```

后面还有两个函数没有看懂存在的意义，但是本身算法并不麻烦.

```
static const uint32_t kMaskDelta = 0xa282ead8ul;

// Return a masked representation of crc.
//
// Motivation: it is problematic to compute the CRC of a string that
// contains embedded CRCs.  Therefore we recommend that CRCs stored
// somewhere (e.g., in files) should be masked before being stored.
inline uint32_t Mask(uint32_t crc) {
  // Rotate right by 15 bits and add a constant.
  return ((crc >> 15) | (crc << 17)) + kMaskDelta;
}

// Return the crc whose masked representation is masked_crc.
inline uint32_t Unmask(uint32_t masked_crc) {
  uint32_t rot = masked_crc - kMaskDelta;
  return ((rot >> 17) | (rot << 15));
}
```

10.9.7 Hash

hash.cc提供了Hash算法，看上去有点类似于murmurhash.

```
uint32_t Hash(const char* data, size_t n, uint32_t seed) {
  // Similar to murmur hash
  const uint32_t m = 0xc6a4a793;
  const uint32_t r = 24;
  const char* limit = data + n;
  uint32_t h = seed ^ (n * m);

  // Pick up four bytes at a time
  while (data + 4 <= limit) {
    uint32_t w = DecodeFixed32(data);
    data += 4;
    h += w;
    h *= m;
    h ^= (h >> 16);
  }

  // Pick up remaining bytes
  switch (limit - data) {
    case 3:
      h += data[2] << 16;
      // fall through
    case 2:
      h += data[1] << 8;
      // fall through
    case 1:
      h += data[0];
      h *= m;
      h ^= (h >> r);
      break;
  }
  return h;
}
```

之前一直纠结于这个seed应该如何来进行设置。其实现在自己也不知道:(. 在bloom.cc里面使用方式是：

```
static uint32_t BloomHash(const Slice& key) {
  return Hash(key.data(), key.size(), 0xbc9f1d34);
}
```

10.9.8 BloomFilterPolicy

第一次看接口有点混淆。为什么CreateFilter需要将生成的filter追加到dst里面呢?其实我猜想这个完全取决于应用， 可能应用在上层希望将filter的内存统一管理，然后一次释放。使用的时候可以选取某一个slice作为filter.

#note: 之前可能对于filter有点错误理解.觉得这个filter没有必要存放在磁盘上而应该全部在内存上面。但是leveldb会针对若干个data block生成一个filter存放在磁盘上面。如果是这样接口就非常好理解了，将所有的filter全部聚合在连续的内存上面写入table.

```
explicit BloomFilterPolicy(int bits_per_key)
    : bits_per_key_(bits_per_key) {
  // 计算为每个key生成多少个probe(检测点).
  // bits_per_key*ln(2)
  // We intentionally round down to reduce probing cost a little bit
  k_ = static_cast<size_t>(bits_per_key * 0.69);  // 0.69 =~ ln(2)
  if (k_ < 1) k_ = 1;
  if (k_ > 30) k_ = 30;
}

virtual void CreateFilter(const Slice* keys, int n, std::string* dst) const {
  // Compute bloom filter size (in both bits and bytes)
  size_t bits = n * bits_per_key_; // 分配多少个bits.

  // For small n, we can see a very high false positive rate.  Fix it
  // by enforcing a minimum bloom filter length.
  if (bits < 64) bits = 64;

  size_t bytes = (bits + 7) / 8; // 做一个round然后计算分配多少个bits.
  bits = bytes * 8;

  const size_t init_size = dst->size();
  dst->resize(init_size + bytes, 0); // dst前面的部分不同追加bytes字节.
  dst->push_back(static_cast<char>(k_));  // Remember # of probes in filter // 将这个probe个数记录在dst末尾.
  char* array = &(*dst)[init_size]; //
  for (size_t i = 0; i < n; i++) {
    // Use double-hashing to generate a sequence of hash values.
    // See analysis in [Kirsch,Mitzenmacher 2006].
    uint32_t h = BloomHash(keys[i]);
    const uint32_t delta = (h >> 17) | (h << 15);  // Rotate right 17 bits
    for (size_t j = 0; j < k_; j++) { // 每次做key个点的probe.
```

```
      const uint32_t bitpos = h % bits;
      array[bitpos/8] |= (1 << (bitpos % 8));
      h += delta;
    }
  }
}

virtual bool KeyMayMatch(const Slice& key, const Slice& bloom_filter) const {
  const size_t len = bloom_filter.size(); // 注意这个len包含了最后一个字节的probe.
  // 后面都非常好理解.
  if (len < 2) return false;

  const char* array = bloom_filter.data();
  const size_t bits = (len - 1) * 8;

  // Use the encoded k so that we can read filters generated by
  // bloom filters created using different parameters.
  const size_t k = array[len-1];
  if (k > 30) {
    // Reserved for potentially new encodings for short bloom filters.
    // Consider it a match.
    return true;
  }

  uint32_t h = BloomHash(key);
  const uint32_t delta = (h >> 17) | (h << 15);  // Rotate right 17 bits
  for (size_t j = 0; j < k; j++) {
    const uint32_t bitpos = h % bits;
    if ((array[bitpos/8] & (1 << (bitpos % 8))) == 0) return false;
    h += delta;
  }
  return true;
}
```

11 Discussion

11.1 leveldb通过iterator遍历，对于相同的key如何保证获取到最新的值（hpplinux)

Question

```
我在看LevelDB代码的时候遇到了一个问题，百思不得其解，也找不到可以探讨请教的人，所以冒昧的给您发了这封邮件，希望得到
我遇到的问题是这样的:
在
void Version::AddIterators(const ReadOptions& options,
                           std::vector<Iterator*>* iters) {
  // Merge all level zero files together since they may overlap
  for (size_t i = 0; i < files_[0].size(); i++) {
    iters->push_back(
        vset_->table_cache_->NewIterator(
            options, files_[0][i]->number, files_[0][i]->file_size));
  }

  // For levels > 0, we can use a concatenating iterator that sequentially
  // walks through the non-overlapping files in the level, opening them
  // lazily.
  for (int level = 1; level < config::kNumLevels; level++) {
    if (!files_[level].empty()) {
      iters->push_back(NewConcatenatingIterator(options, level));
    }
  }
}

中对于Level 0层是按照下标从0到N开始遍历的， 但是由于数据加入的时候老的文件在前，新的在后，所以这样的话在iters数组中
下标最小的不一定是最新的。
而在DBImpl::NewInternalIterator 中会把该函数的返回结果直接进行merging，而且原则是key相同的话选取丢弃后面出现的。

这样的策略的话会不会导致较老的数据被留下，较新的被删除 ?
```

Answer

是这样的，你可以看到AddIterators这个部分是被DBImpl::NewInternalIterator调用的，得到所有的iterators之后，构造一个MergingIterator对象。

```
// 对于version来说可能存在很多文件需要遍历.
versions_->current()->AddIterators(options, &list);
// 将这些内容构造称为一个merge iterator.
// 注意这里的内容都加了引用计数.
Iterator* internal_iter =
    NewMergingIterator(&internal_comparator_, &list[0], list.size());
```

注意它这里提供的comparator是一个internal_comparator. 这个comparator不仅仅比较user key, 还比较sequence number. 因为sequence number是顺序分配的，所以新的kv得到更大的sequence number. 代码在这里：

```
int InternalKeyComparator::Compare(const Slice& akey, const Slice& bkey) const {
  // Order by:
  //    increasing user key (according to user-supplied comparator)
  //    decreasing sequence number
  //    decreasing type (though sequence# should be enough to disambiguate)
  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) { // 按照sequence number比较.
     // 之前我们在MemTableInserter里面可以看到sequence number是不断增加的.
      r = -1;
    } else if (anum < bnum) {
      r = +1;
    }
  }
  return r;
}
```

然后这个就好解释问题了。首先每个iterator内部都是按照key做好排序的，多路iterator如果出现相同的key那么使用sequence number大的那个，这样就可以保证始终首先看到的是新值。

comments powered by Disqus