git clone <https://github.com/facebook/rocksdb.git>

git checkout v8.10.2

注释WARNING\_FLAGS += -Werror

动态库

make shared\_lib

make install-shared PREFIX=/usr/local/rocksdb

静态库

make static\_lib

make install-static PREFIX=/usr/local/rocksdb

将rocksdb作为子项目编译：

添加编译目标

add\_custom\_target(build\_rocksdb

COMMAD make -j4 shared\_lib -C ${ROCKSDB\_SOURCE\_DIR}

COMMENT “Building RocksDB”

}

添加依赖

ADD\_DEPENDENCIES(main\_target build\_rocksdb)

编译单元测试

如db\_iter\_test.cc：make db\_iter\_test -j4

编译ldb

进入rocksdb目录

make ldb //默认debug版本

DEBUG\_LEVEL=0 make ldb //release版本

data/LOG：rocksdb运行日志

先打印配置（Options）

DB Stats：db状态

Compaction Stats：LSM树状态

data/OPTIONS-xxxxxx：最近打开的一次配置

WAL

Write Ahead Log，数据写入前先写入到log中，提供数据持久性和故障恢复能力

持久性：系统崩溃或断电时，未落盘数据不会丢失

快速恢复：系统重启时，通过重放WAL中操作，将系统还原到最后一次正常关闭的状态

性能优化：相比于落盘，写入log通常更快，因为落盘可以延迟执行，写入日志文件可以立即完成

./ldb dump\_wal --walfile=../data/000003.log --header --print\_value

编译失败：make clean后再次编译，编译时间超级长

从左到右：序列号，操作个数，字节数，物理偏移，PUT(0)表示写入，0代表列族id为0（默认列族）

Sequence,Count,ByteSize,Physical Offset,Key(s) : value

1409182,3,79,0,DELETE(0) : 0x6B65793234343731393331 PUT(0) : 0x6B65793533353939363434 : 0x76616C75653533353939363434 PUT(0) : 0x6B65793232363230363439 : 0x76616C75653232363230363439

1409185,2,66,86,PUT(0) : 0x6B65793732313734373132 : 0x76616C75653732313734373132 PUT(0) : 0x6B65793432383836373732 : 0x76616C75653432383836373732

1409187,1,25,159,DELETE(0) : 0x6B65793130393032313930

1409188,2,66,191,PUT(0) : 0x6B65793638353537303639 : 0x76616C75653638353537303639 PUT(0) : 0x6B65793137363332353834 : 0x76616C75653137363332353834

1409190,1,25,264,DELETE(0) : 0x6B65793631303430343132

MANIFEST：描述所有列族中LSM树结构信息的文件，包括每层sst文件数量，每层文件概要信息，用于重建树

./ldb manifest\_dump --path="../data/MANIFEST-000005"

默认列族0

Log number：日志文件编号

Comparator：比较器，用于比较key之间大小

BytewiseComparator：根据字符大小比较key

--------------- Column family "default" (ID 0) --------------

log number: 8

comparator: leveldb.BytewiseComparator

--- level 0 --- version# 1 ---

9:19946862[1 .. 1409181]['key10000069' seq:1231705, type:1 .. 'key99999983' seq:1116682, type:1]

// 9对应sst编号，大小为19946862字节，最小seq为0，最大seq为1409181

// type为写入类型，参考枚举类型ValueType

--- level 1 --- version# 1 ---

--- level 2 --- version# 1 ---

...省略

--- level 63 --- version# 1 ---

next\_file\_number 11 last\_sequence 1409181 prev\_log\_number 0 max\_column\_family 0 min\_log\_number\_to\_keep 8

Next\_file\_number：下一个sst可用编号

Last\_sequence：上次写操作序列号

Prev\_log\_number：之前的log（WAL）文件编号

Max\_column\_family：最大的列族编号

Min\_log\_number\_to\_keep：2PC模式下使用，恢复中忽略小于等于该值的日志

Memtable：

写入过程：先写WAL，再写memtable，memtable满或满足一定条件之后变为immemtable待刷新（flush），生成一个新的memtable

数据写入memtable视为写入成功，memtable存放在内存中，同时服务于读和写，新的写入总是插入到memtable，一个memtable被写满或满足一定条件后，会变成不可修改的memtable（immemtable），并被一个新的memtable替换，一个后台线程会将immemtable落盘（flush）到一个sst文件，之后该immemtable会被销毁。

class MemTable {

...

  KeyComparator comparator\_; // 用于比较key的大小

...

  std::unique\_ptr<MemTableRep> table\_; // 指向skiplist

  std::unique\_ptr<MemTableRep> range\_del\_table\_; // 指向特殊的skiplist，用于kTypeRangeDeletion类型，如连续删除key1-key5时，仅需指定一次key(1-5)，节省资源

  std::atomic\_bool is\_range\_del\_table\_empty\_;

  // Total data size of all data inserted

  std::atomic<uint64\_t> data\_size\_; // 总数据大小

  std::atomic<uint64\_t> num\_entries\_; // 元素个数

  std::atomic<uint64\_t> num\_deletes\_; // 删除个数

  std::atomic<uint64\_t> num\_range\_deletes\_; // 批量删除个数

  // Dynamically changeable memtable option

  std::atomic<size\_t> write\_buffer\_size\_; // memtable支持的最大大小

  // These are used to manage memtable flushes to storage

  bool flush\_in\_progress\_;  // started the flush，是否在flush中

  bool flush\_completed\_;    // finished the flush，flush是否完成

  uint64\_t file\_number\_;    // filled up after flush is complete

  // The updates to be applied to the transaction log when this

  // memtable is flushed to storage.

  VersionEdit edit\_; // 从属的版本

  std::unique\_ptr<DynamicBloom> bloom\_filter\_; // 布隆过滤器，快速判断key是否在memtable中

}

Memtable插入一条kv的数据格式：

|-internal\_key\_size-|---key---|--seq--|--type--|--value\_size--|---value---|

Internal\_key\_size：varint类型，包括key、seq、type所占的字节数

Key：字符串，写入的key

Seq：序列号，占7个字节

Type：操作类型，占1个字节

Value\_size：var\_int类型，表示value长度

Value：字符串，写入的value

Skiplist（跳表）

Memtable最常用的实现基于skiplist，在多数情况下读、写、随机访问及序列化扫描性能较好，且支持并发写入。

struct Node{

std::atomic<Node\*> next\_[1];

}

生成节点时临时存储节点高度，插入到skiplist后不需要高度

最高节点开始，循环：同高度有下一节点且小于等于key则跳转该节点，否则高度-1

Status MemTable::Add(SequenceNumber s, ValueType type,

                     const Slice& key, /\* user key \*/

                     const Slice& value,

                     const ProtectionInfoKVOS64\* kv\_prot\_info,

                     bool allow\_concurrent,

                     MemTablePostProcessInfo\* post\_process\_info, void\*\* hint) {

  // kv编码

// 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()]

  //  checksum     : char[moptions\_.protection\_bytes\_per\_key]

  uint32\_t key\_size = static\_cast<uint32\_t>(key.size());

  uint32\_t val\_size = static\_cast<uint32\_t>(value.size());

  uint32\_t internal\_key\_size = key\_size + 8; // 对应seq7+type1字节

  const uint32\_t encoded\_len = VarintLength(internal\_key\_size) +

                               internal\_key\_size + VarintLength(val\_size) +

                               val\_size + moptions\_.protection\_bytes\_per\_key;

  char\* buf = nullptr;

  std::unique\_ptr<MemTableRep>& table = // 申请skiplist

      type == kTypeRangeDeletion ? range\_del\_table\_ : table\_;

  KeyHandle handle = table->Allocate(encoded\_len, &buf);

  char\* p = EncodeVarint32(buf, internal\_key\_size);

  memcpy(p, key.data(), key\_size);

  Slice key\_slice(p, key\_size);

  p += key\_size;

  uint64\_t packed = PackSequenceAndType(s, type); // seq和type打包为8字节

  EncodeFixed64(p, packed);

  p += 8;

  p = EncodeVarint32(p, val\_size);

  memcpy(p, value.data(), val\_size);

  assert((unsigned)(p + val\_size - buf + moptions\_.protection\_bytes\_per\_key) ==

         (unsigned)encoded\_len);

  UpdateEntryChecksum(kv\_prot\_info, key, value, type, s,

                      buf + encoded\_len - moptions\_.protection\_bytes\_per\_key);

  Slice encoded(buf, encoded\_len - moptions\_.protection\_bytes\_per\_key);

if (kv\_prot\_info != nullptr) {

    TEST\_SYNC\_POINT\_CALLBACK("MemTable::Add:Encoded", &encoded);

    Status status = VerifyEncodedEntry(encoded, \*kv\_prot\_info);

    if (!status.ok()) {

      return status;

    }

  }

  Slice key\_without\_ts = StripTimestampFromUserKey(key, ts\_sz\_);

// allow\_concurrent默认为false

  if (!allow\_concurrent) {

// Extract prefix for insert with hint.

    if (insert\_with\_hint\_prefix\_extractor\_ != nullptr &&

        insert\_with\_hint\_prefix\_extractor\_->InDomain(key\_slice)) {

      Slice prefix = insert\_with\_hint\_prefix\_extractor\_->Transform(key\_slice);

// 带hint插入，通过map记录一些前缀插入skiplist的位置，从而快速插入同样的前缀

// 默认不启用

      bool res = table->InsertKeyWithHint(handle, &insert\_hints\_[prefix]);

      if (UNLIKELY(!res)) {

        return Status::TryAgain("key+seq exists");

      }

} else {

// 插入到skiplist

      bool res = table->InsertKey(handle);

      if (UNLIKELY(!res)) {

        return Status::TryAgain("key+seq exists");

      }

}

// 更新统计信息

// this is a bit ugly, but is the way to avoid locked instructions

    // when incrementing an atomic

    num\_entries\_.store(num\_entries\_.load(std::memory\_order\_relaxed) + 1,

                       std::memory\_order\_relaxed);

    data\_size\_.store(data\_size\_.load(std::memory\_order\_relaxed) + encoded\_len,

                     std::memory\_order\_relaxed);

    if (type == kTypeDeletion || type == kTypeSingleDeletion ||

        type == kTypeDeletionWithTimestamp) {

      num\_deletes\_.store(num\_deletes\_.load(std::memory\_order\_relaxed) + 1,

                         std::memory\_order\_relaxed);

    } else if (type == kTypeRangeDeletion) {

      uint64\_t val = num\_range\_deletes\_.load(std::memory\_order\_relaxed) + 1;

      num\_range\_deletes\_.store(val, std::memory\_order\_relaxed);

    }

// 更新布隆过滤器

    if (bloom\_filter\_ && prefix\_extractor\_ &&

        prefix\_extractor\_->InDomain(key\_without\_ts)) {

      bloom\_filter\_->Add(prefix\_extractor\_->Transform(key\_without\_ts));

    }

    if (bloom\_filter\_ && moptions\_.memtable\_whole\_key\_filtering) {

      bloom\_filter\_->Add(key\_without\_ts);

}

    // The first sequence number inserted into the memtable

    assert(first\_seqno\_ == 0 || s >= first\_seqno\_);

    if (first\_seqno\_ == 0) {

      first\_seqno\_.store(s, std::memory\_order\_relaxed);

      if (earliest\_seqno\_ == kMaxSequenceNumber) {

        earliest\_seqno\_.store(GetFirstSequenceNumber(),

                              std::memory\_order\_relaxed);

      }

      assert(first\_seqno\_.load() >= earliest\_seqno\_.load());

    }

    assert(post\_process\_info == nullptr);

    // TODO(yuzhangyu): support updating newest UDT for when `allow\_concurrent`

    // is true.

    MaybeUpdateNewestUDT(key\_slice);

    UpdateFlushState();

  } else {

// 并发插入

// ...

  }

// 处理范围删除操作

// ...

  UpdateOldestKeyTime();

  TEST\_SYNC\_POINT\_CALLBACK("MemTable::Add:BeforeReturn:Encoded", &encoded);

  return Status::OK();

}

链表申请内存：

char\* InlineSkipList<Comparator>::AllocateKey(size\_t key\_size) {

// 随机一个节点里的高度

  return const\_cast<char\*>(AllocateNode(key\_size, RandomHeight())->Key());

}

申请节点空间：

InlineSkipList<Comparator>::AllocateNode(size\_t key\_size, int height) {

// 高度-1个Node指针，每个指针指向该高度的下一个节点(next\_[-n]~next\_[-1])

  auto prefix = sizeof(std::atomic<Node\*>) \* (height - 1);

// 最终通过AllocateAligned申请内存，申请height个节点+key的空间

  char\* raw = allocator\_->AllocateAligned(prefix + sizeof(Node) + key\_size);

// x为高度0的位置，数据从低地址到高地址存放，所以放在raw+prefix

  Node\* x = reinterpret\_cast<Node\*>(raw + prefix);

  // next\_[0]用于暂时存储节点高度（指定next\_[-n]时使用）

// 插入完成后就知道下一个节点地址了，会存储到next\_[0]，不需要继续存节点高度

  x->StashHeight(height);

  return x; // 返回最高地址，用于[-i]的取用（充分利用cpu的prefetch）

}

相较leveldb的优化：

低地址存放高节点，高地址存放低节点，充分利用CPU的prefetch

Key存放完整数据，而不是指针

SST（Sorted String Table）

通过flush和compaction生成，SST内Key有序，Key和Value都是任意长度的字符串。L0的各个SST的Key范围可以重叠，L1及以上的SST之间是严格有序的，下一个SST的最小Key必定大于上一个SST的最大Key。

Flush操作不会做合并，所以L0的SST之间不会保持有序。

SST结构：

Footer->metaindex block->meta block\*k(包括index block：存储data block句柄）

Footer存储index block位置、metaindex block位置

Index block存储所有data block的位置

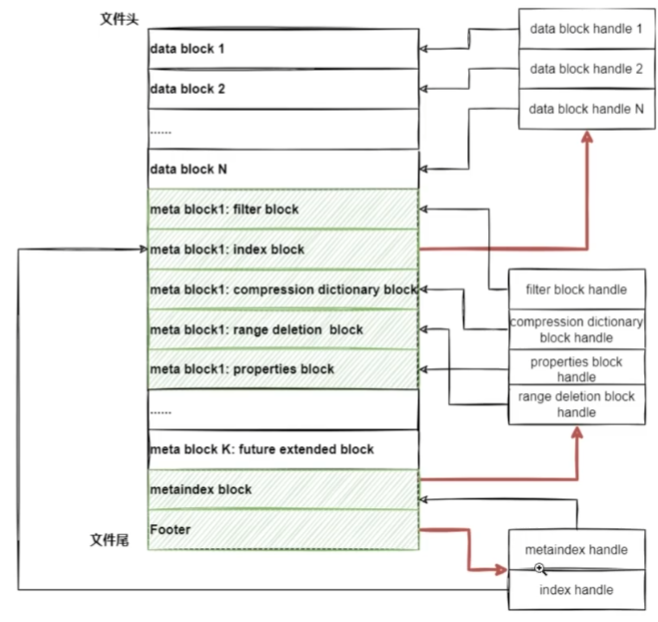
Metaindex block存储除index block外所有meta block的位置

Filter block：过滤器元数据块，快速确定key是否在sst中（一定不存在，或低误判率可能存在）

Compression dictionary block：压缩字典元数据块

Properties block：属性元数据块

Range deletion block：范围删除元数据块，可以快速确定key是否被删除



./sst\_dump --file=../data/000012.sst --command=raw

000012\_dump.txt：

Footer Details:

--------------------------------------

metaindex handle: DB928B0123

index handle: D99E8A01F86C

table\_magic\_number: 9863518390377041911 //魔术值，固定

format version: 5 // 版本

Metaindex Details:

--------------------------------------

Properties block handle: D68B8B018007

Table Properties:

--------------------------------------

# data blocks: 891 // datablock个数

# entries: 153074 // 写入次数，包括PUT，删除和范围删除

# deletions: 51033 // 删除次数，包括范围删除

# merge operands: 0 // merge个数

# range deletions: 0 // 范围删除次数

raw key size: 2891335 // Key总大小

raw average key size: 18.888479 // 平均Key大小

raw value size: 1315125 // value总大小

raw average value size: 8.591433 // 平均Value大小

data block size: 2264921 // data block大小

index block size (user-key? 1, delta-value? 1): 17531 // index block大小

filter block size: 0 // filter block大小

# entries for filter: 0 // filter接收到的请求个数？

(estimated) table size: 2282452 // 预估table大小

filter policy name: N/A // filter名称

prefix extractor name: nullptr // 前缀提取器名称

column family ID: 0 // 列族ID

column family name: default // 列族名称

comparator name: leveldb.BytewiseComparator // 比较器名称

user defined timestamps persisted: true // 用户是否定义时间戳保持不变

merge operator name: nullptr // 合并操作符名称

property collectors names: [] // 属性收集器名称

SST file compression algo: Snappy // SST压缩方法为Snappy

SST file compression options: window\_bits=-14; level=32767; strategy=0; max\_dict\_bytes=0; zstd\_max\_train\_bytes=0; enabled=0; max\_dict\_buffer\_bytes=0; use\_zstd\_dict\_trainer=1; // SST压缩配置

creation time: 1729415545 // 最先写入memtable的时间

time stamp of earliest key: 0 // 最早写入的key的时间戳

file creation time: 0 // sst文件创建的时间

slow compression estimated data size: 0

fast compression estimated data size: 0

DB identity: c50e5b37-63eb-4bd3-981a-c90e97dc79fd

DB session identity: QUDZI0KVIC4XED7IZDQL

DB host id: embedded415-System-Product-Name

original file number: 12

unique ID: 382E354BFA955664-8374E402ADC38AA9

Sequence number to time mapping:

Index Details: // 索引data block

--------------------------------------

Block key hex dump: Data block handle

Block key ascii

HEX 6B65793130313136: 00E813 offset 0 size 2536

ASCII k e y 1 0 1 1 6

------

HEX 6B65793130323035: ED13E013 offset 2541 size 2528

ASCII k e y 1 0 2 0 5

------

Range deletions: // 范围删除数据（样例）

--------------------------------------

HEX 6B65793130323035: ED13E013 offset 2541 size 2528

ASCII k e y 1 0 1 1 6 : k e y 1 0 2 0 5

------

Data Block # 1 @ 00E813 // data block存储数据

--------------------------------------

HEX 6B65793130303030313630: 76616C75653130303030313630

ASCII k e y 1 0 0 0 0 1 6 0 : v a l u e 1 0 0 0 0 1 6 0

------

HEX 6B65793130303031343038: 76616C75653130303031343038

ASCII k e y 1 0 0 0 1 4 0 8 : v a l u e 1 0 0 0 1 4 0 8

------

Data Block Summary: // data block 概要（在最后）

--------------------------------------

# data blocks: 891

min data block size: 2082

max data block size: 2622

avg data block size: 2536.998878

列族

将数据库的数据进行逻辑划分，弥补了rocksdb单个进程只能操作一个数据库的问题，每个列族有自己的LSM结构，共享WAL，不共享memtable、immemtable、SST。默认只有一个列族，名称default。类似数据库“表”的概念。

配置分为ColumnFamilyOptions和DBOptions，前者是单个列族的配置，后者是整个DB的配置

通过DBImpl获取ColumnFamilySet，存储了列族对应数据（使用ColumnFamilyHandleImpl句柄也能获取数据）

存储Version，管理版本的所有sst文件

存储SuperVersion，存储Verion、Memtable和MemtableListVersion，多管理了内存中的数据

class DBImpl : public DB {

  // 根据列族ID获取ColumnFamilyData

  ColumnFamilyHandle\* GetColumnFamilyHandle(uint32\_t column\_family\_id);

  // 可以用于获取ColumnFamilySet

  std::unique\_ptr<ColumnFamilyMemTablesImpl> column\_family\_memtables\_;

  std::unique\_ptr<VersionSet> versions\_;

}

class ColumnFamilyMemTablesImpl : public ColumnFamilyMemTables {

  ColumnFamilySet\* column\_family\_set\_;

}

class VersionSet {

  std::unique\_ptr<ColumnFamilySet> column\_family\_set\_;

}

class ColumnFamilySet {

  // 列族ID到列族数据的映射

  UnorderedMap<uint32\_t, ColumnFamilyData\*> column\_family\_data\_;

}

class ColumnFamilyData {

  MemTable\* mem\_; // 当前的Memtable

  MemTableList imm\_; // 管理immemtable

  SuperVersion\* super\_version\_; // super\_version，维护该列族最新版本的memtable、immemtable、sst

}

class MemTableList {

  MemTableListVersion\* current\_; // 管理多个immemtable

}

class MemTableListVersion {

  std::list<MemTable\*> memlist\_; // 等待flush的immemtable

}

Version管理

管理某一个时刻的db状态，任何读写都是对一个version的操作

当compaction结束或immemtable被flush到磁盘时，会创建一个新的version。在任何时刻rocksdb中只会有一个current version，Get查询操作或迭代器都会使用current version。没有被任何Get或迭代器使用的“过时”version会被清除，没有被任何version使用的SST文件则会被删除。

// Version保存着LSM结构的SST文件信息，一个Version属于一个列族

// 它会和该列族的其他version组成一个链表

class Version {

  ColumnFamilyData\* cfd\_; // version所属的columnFamilyData

  VersionStorageInfo storage\_info\_; // 对应的所有SST文件及层级信息

  VersionSet\* vset\_; // 所属的versionSet

  Version\* next\_; // 链表的下一个指针，链表为头部插入（除头部的dummy\_version）

  Version\* prev\_; // 链表的上一个指针

}

// 一个db一个versionSet，包含db的所有列族

// 新生成SST文件时通过该类的LogAndApply接口写入新的version

class VersionSet {

// 保存所有的列族集合

  std::unique\_ptr<ColumnFamilySet> column\_family\_set\_;

// 维护version number，作为下一个version的version\_number参数

  uint64\_t current\_version\_number\_;

}

// compaction时生成的version会先生成一个versionEdit

// 通过LogAndApply接口放入到ManifestWriter类型中

// 最终通过ProcessManifestWrites函数new出新的version

// versionEdit提供addFile和deleteFile接口，用于处理sst文件的新增和删除

class VersionEdit {

}

// 保存当前最新的version和关联的memtable、immemtable

// SuperVersion和Version的区别是Version只管理盘上不变的SST文件

// SuperVersion管理SST和memtable、immemtable

struct SuperVersion {

  ColumnFamilyData\* cfd; // 所属ColomnFamily

  MemTable\* mem; // 持有的MemTable

  MemTableListVersion\* imm; // 当前列族所有的immemtable

  Version\* current; // 当前最新version指针

  uint64\_t version\_number; // 当前SuperVersion的版本号

  static int dummy; // kSVInUse会指向dummy

  static void\* const kSVInUse; // 表示当前SV正在使用

  static void\* const kSVObsolete; // 表示当前SV已经释放

初始状态：

Version 1 = {f1, f2, f3} (current version)

当前磁盘上存在文件：f1, f2, f3

创建迭代器：

Iterator \*iter = db\_->NewIterator(); 最新的version被iter引用

随着数据写入，flush新生成了一个SST文件f4，并生成新version：

Version 2 = {f1, f2, f3, f4} (current version)

当前磁盘上存在文件：f1, f2, f3, f4

后台发生一次compaction，将f2-4压缩为f5，并产生新version：

Version 3 = {f1, f5} (current version)

此时各version结构：

Version 3 = {f1, f5} (current version)

Version 2 = {f1, f2, f3, f4}

Version 1 = {f1, f2, f3}

当前磁盘存在文件：f1, f2, f3, f4, f5

Version 2因为flush结束，没有被引用被删除，同时f4没有被引用也被删除：

Version 3 = {f1, f5} (current version)

Version 1 = {f1, f2, f3}

当前磁盘存在文件：f1, f2, f3, f5

此时如果迭代器iter被删除，则version 1被删除：

Version 3 = {f1, f5} (current version)

当前磁盘存在文件：f1, f5

**布隆过滤器（bloom filter）**

用于判断一个元素肯定不在一个集合中，或者可能在一个集合中。用来判断一个key是否在memtable或SST文件内，提高读效率。一个布隆过滤器返回的结果：

某个元素可能在集合中

某个元素一定不在集合中

长度m位的位数组和k个哈希函数，将key通过k个哈希函数计算出在位数组的k个位置，并将对应下标位置1。读一个key会先看k个位置是否都为1，如果部分位不为1则一定不在集合中，如果都为1则可能在集合中（不同key用不同哈希函数可能得到相同下标）

可以选择用多少位key进行哈希，从而进行优化。

MemTable::MemTable(超级超级多的参数) {

  // use bloom\_filter\_ for both whole key and prefix bloom filter

  if ((prefix\_extractor\_ || moptions\_.memtable\_whole\_key\_filtering) &&

      moptions\_.memtable\_prefix\_bloom\_bits > 0) {

    bloom\_filter\_.reset(

        new DynamicBloom(&arena\_, moptions\_.memtable\_prefix\_bloom\_bits,

                         6 /\* hard coded 6 probes \*/,

                         moptions\_.memtable\_huge\_page\_size, ioptions.logger));

  }

}

inline bool DynamicBloom::MayContain(const Slice& key) const {

  return (MayContainHash(BloomHash(key)));

}

inline bool DynamicBloom::MayContainHash(uint32\_t h32) const {

  size\_t a = FastRange32(h32, kLen);

  PREFETCH(data\_ + a, 0, 3);

  return DoubleProbe(h32, a);

}

// 添加一个key到bloom过滤器内

inline void DynamicBloom::AddHash(uint32\_t hash) {

  AddHash(hash, [](std::atomic<uint64\_t>\* ptr, uint64\_t mask) {

    ptr->store(ptr->load(std::memory\_order\_relaxed) | mask,

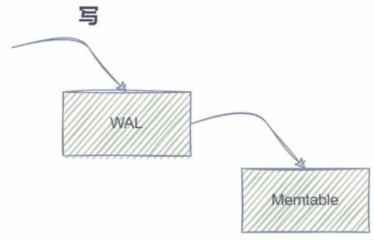
               std::memory\_order\_relaxed);

  });

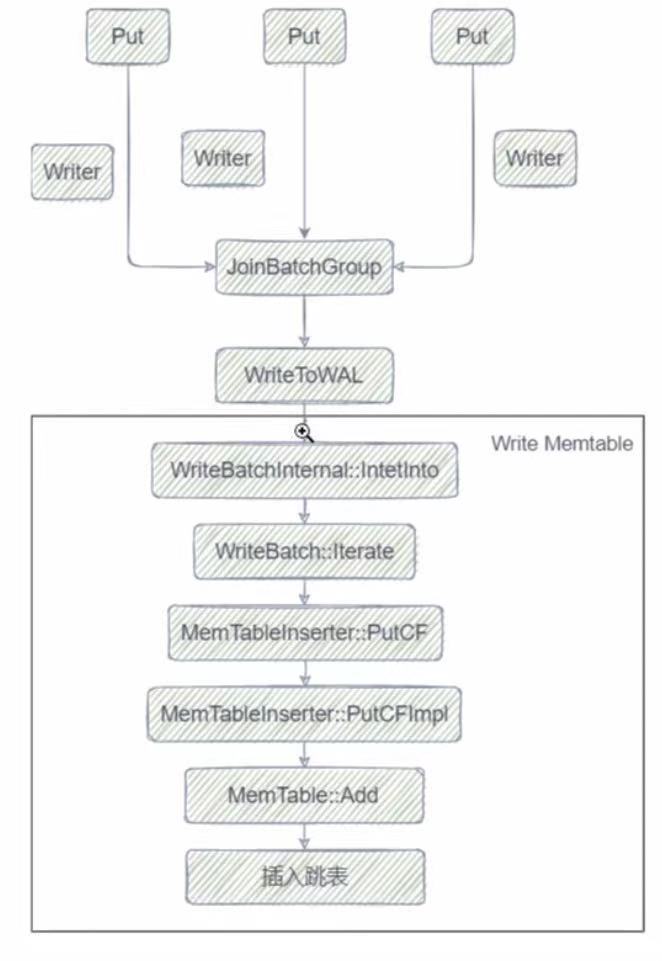
}

写流程

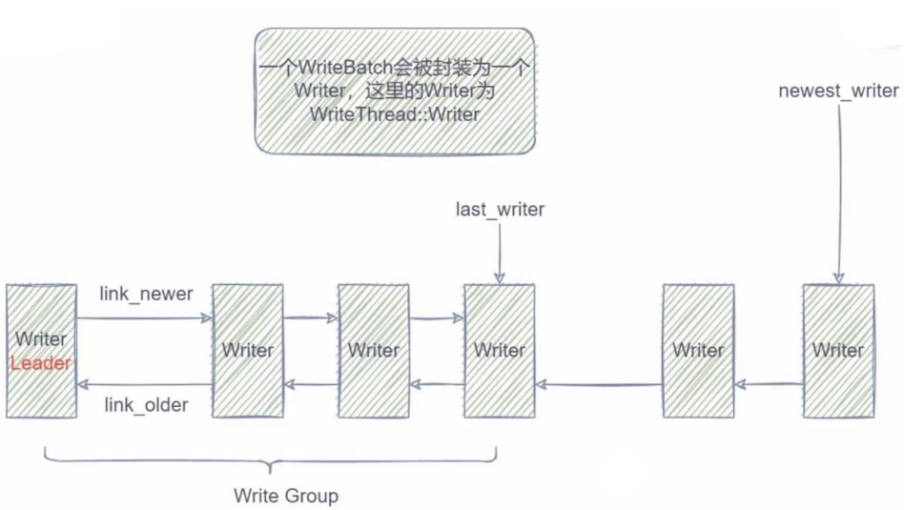
先写WAL，再写Memtable



Rocksdb的写是分批次（batch）写，将多个写入放入一个batch，从batch中选出一个leader，其他则为follower，由leader负责这一batch的写入，并在写完后唤醒所有follower。



一个WriteBatch被封装为一个WriteThread::Write，Write Group由双向链表组成（Leader和Follower），Write Group或Writer之间为单向链表，link\_older由链表后方往leader方向，link\_newer由leader指向链表后方



当前Group写入后，将后面的Writer组建Group，并参与写入。

写流程代码：

// 四个接口，分为带时间戳ts和不带的、直接用默认列族和指定列族的

// 常用的是默认列族的

virtual Status Put(const WriteOptions& options,

                   ColumnFamilyHandle\* column\_family, const Slice& key,

                   const Slice& value) = 0;

virtual Status Put(const WriteOptions& options,

                   ColumnFamilyHandle\* column\_family, const Slice& key,

                   const Slice& ts, const Slice& value) = 0;

virtual Status Put(const WriteOptions& options, const Slice& key,

                   const Slice& value) {

  return Put(options, DefaultColumnFamily(), key, value);

}

virtual Status Put(const WriteOptions& options, const Slice& key,

                   const Slice& ts, const Slice& value) {

  return Put(options, DefaultColumnFamily(), key, ts, value);

}

// 不带时间戳的

Status DB::Put(const WriteOptions& opt, ColumnFamilyHandle\* column\_family,

               const Slice& key, const Slice& value) {

  // 直接将key value封装到一个batch内参与写入

  // header占8字节, count占4字节, type占1字节

  // 额外11字节用于key长度和value长度

  WriteBatch batch(key.size() + value.size() + 24, 0 /\* 最大字节数 \*/,

                   opt.protection\_bytes\_per\_key, 0 /\* 默认时间戳大小 \*/);

  Status s = batch.Put(column\_family, key, value);

  if (!s.ok()) {

    return s;

  }

  return Write(opt, &batch);

}

// 带时间戳的

Status DB::Put(const WriteOptions& opt, ColumnFamilyHandle\* column\_family,

               const Slice& key, const Slice& ts, const Slice& value) {

  ColumnFamilyHandle\* default\_cf = DefaultColumnFamily();

  assert(default\_cf);

  const Comparator\* const default\_cf\_ucmp = default\_cf->GetComparator();

  assert(default\_cf\_ucmp);

  WriteBatch batch(0 /\* reserved\_bytes \*/, 0 /\* max\_bytes \*/,

                   opt.protection\_bytes\_per\_key,

                   default\_cf\_ucmp->timestamp\_size());

  Status s = batch.Put(column\_family, key, ts, value);

  if (!s.ok()) {

    return s;

  }

  return Write(opt, &batch);

}

Status DBImpl::Write(const WriteOptions& write\_options, WriteBatch\* my\_batch) {

  Status s;

  if (write\_options.protection\_bytes\_per\_key > 0) {

    s = WriteBatchInternal::UpdateProtectionInfo(

        my\_batch, write\_options.protection\_bytes\_per\_key);

  }

  if (s.ok()) {

    s = WriteImpl(write\_options, my\_batch, /\*callback=\*/nullptr,

                  /\*log\_used=\*/nullptr);

  }

  return s;

}

Status DBImpl::WriteImpl(一大堆参数) {

  ...

  WriteThread::Writer w(write\_options, my\_batch, callback, log\_ref,

                        disable\_memtable, batch\_cnt, pre\_release\_callback,

                        post\_memtable\_callback);

  StopWatch write\_sw(immutable\_db\_options\_.clock, stats\_, DB\_WRITE);

// 该函数称为leader的w返回，否则在内部阻塞等待唤醒

  write\_thread\_.JoinBatchGroup(&w);

  if (w.state == WriteThread::STATE\_PARALLEL\_MEMTABLE\_WRITER) {

// 并发写Memtable情况，默认没启用 ...

  }

  if (w.state == WriteThread::STATE\_COMPLETED) {

// follower被唤醒后进入这里返回写操作完成

    if (log\_used != nullptr) {

      \*log\_used = w.log\_used;

    }

    if (seq\_used != nullptr) {

      \*seq\_used = w.sequence;

    }

    return w.FinalStatus();

  }

  // 只有leader会执行这里

  assert(w.state == WriteThread::STATE\_GROUP\_LEADER);

  Status status;

// leader在执行写时可能将writers\_中的未唤醒任务一并在本batch中执行

  WriteContext write\_context;

  LogContext log\_context(write\_options.sync);

  WriteThread::WriteGroup write\_group;

  bool in\_parallel\_group = false;

  uint64\_t last\_sequence = kMaxSequenceNumber;

  assert(!two\_write\_queues\_ || !disable\_memtable);

  {

    // 写入前的准备工作

    // WAL文件是否超过max\_total\_wal\_size，是则切换WAL文件

// memtable占用内存超过阈值则需要flush

    // 预处理写入计时器

PERF\_TIMER\_STOP(write\_pre\_and\_post\_process\_time);

    status = PreprocessWrite(write\_options, &log\_context, &write\_context);

    if (!two\_write\_queues\_) {

      // 获取最新version的last sequence

      last\_sequence = versions\_->LastSequence();

}

    PERF\_TIMER\_START(write\_pre\_and\_post\_process\_time);

  }

  // 完成group成员确定：group大小、writer个数和连接、last\_writer\_位置

  TEST\_SYNC\_POINT("DBImpl::WriteImpl:BeforeLeaderEnters");

  last\_batch\_group\_size\_ =

      write\_thread\_.EnterAsBatchGroupLeader(&w, &write\_group);

  IOStatus io\_s;

  Status pre\_release\_cb\_status;

  if (status.ok()) {

    // 统计写入的概要信息，决定memtable是否可以并发写入

    bool parallel = immutable\_db\_options\_.allow\_concurrent\_memtable\_write &&

                    write\_group.size > 1;

    size\_t total\_count = 0;

    size\_t valid\_batches = 0;

    size\_t total\_byte\_size = 0;

    size\_t pre\_release\_callback\_cnt = 0;

    for (auto\* writer : write\_group) {

      assert(writer);

      if (writer->CheckCallback(this)) {

        valid\_batches += writer->batch\_cnt;

        if (writer->ShouldWriteToMemtable()) {

          total\_count += WriteBatchInternal::Count(writer->batch);

          total\_byte\_size = WriteBatchInternal::AppendedByteSize(

              total\_byte\_size, WriteBatchInternal::ByteSize(writer->batch));

          parallel = parallel && !writer->batch->HasMerge();

        }

        if (writer->pre\_release\_callback) {

          pre\_release\_callback\_cnt++;

        }

      }

    }

    // 写操作追踪，确保操作顺序

    if (tracer\_) {

      InstrumentedMutexLock lock(&trace\_mutex\_);

      if (tracer\_ && tracer\_->IsWriteOrderPreserved()) {

        for (auto\* writer : write\_group) {

          if (writer->CallbackFailed()) {

            continue;

          }

          tracer\_->Write(writer->batch).PermitUncheckedError();

        }

      }

    }

    // 计算本次写入消耗的seq

size\_t seq\_inc = seq\_per\_batch\_ ? valid\_batches : total\_count;

...

    if (!two\_write\_queues\_) {

      if (status.ok() && !write\_options.disableWAL) {

        assert(log\_context.log\_file\_number\_size);

        LogFileNumberSize& log\_file\_number\_size =

            \*(log\_context.log\_file\_number\_size);

        PERF\_TIMER\_GUARD(write\_wal\_time);

// 写WAL

        io\_s =

            WriteToWAL(write\_group, log\_context.writer, log\_used,

                       log\_context.need\_log\_sync, log\_context.need\_log\_dir\_sync,

                       last\_sequence + 1, log\_file\_number\_size);

      }

    } else {

      ...

    }

    status = io\_s;

    assert(last\_sequence != kMaxSequenceNumber);

    const SequenceNumber current\_sequence = last\_sequence + 1;

    last\_sequence += seq\_inc;

    // PreReleaseCallback is called after WAL write and before memtable write

    if (status.ok()) {

      SequenceNumber next\_sequence = current\_sequence;

      size\_t index = 0;

      // Note: the logic for advancing seq here must be consistent with the

      // logic in WriteBatchInternal::InsertInto(write\_group...) as well as

      // with WriteBatchInternal::InsertInto(write\_batch...) that is called on

      // the merged batch during recovery from the WAL.

      for (auto\* writer : write\_group) {

        if (writer->CallbackFailed()) {

          continue;

        }

// 为每个操作赋值sequence

        writer->sequence = next\_sequence;

        ...

      }

}

    if (status.ok()) {

      PERF\_TIMER\_GUARD(write\_memtable\_time);

      if (!parallel) {

        // 写memtable

        w.status = WriteBatchInternal::InsertInto(

            write\_group, current\_sequence, column\_family\_memtables\_.get(),

            &flush\_scheduler\_, &trim\_history\_scheduler\_,

            write\_options.ignore\_missing\_column\_families,

            0 /\*recovery\_log\_number\*/, this, parallel, seq\_per\_batch\_,

            batch\_per\_txn\_);

      } else {

        ...

      }

      if (seq\_used != nullptr) {

        \*seq\_used = w.sequence;

      }

    }

  }

  PERF\_TIMER\_START(write\_pre\_and\_post\_process\_time);

...

// 是否需要下刷WAL文件，默认False

  if (log\_context.need\_log\_sync) {

    ...

  }

  bool should\_exit\_batch\_group = true;

  ...

  if (should\_exit\_batch\_group) {

    if (status.ok()) {

      for (auto\* tmp\_w : write\_group) {

        assert(tmp\_w);

        if (tmp\_w->post\_memtable\_callback) {

          Status tmp\_s =

              (\*tmp\_w->post\_memtable\_callback)(last\_sequence, disable\_memtable);

          assert(tmp\_s.ok());

        }

      }

      // 更新version的last\_sequence

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

    }

MemTableInsertStatusCheck(w.status);

// 结束当前group的收尾工作，包括唤醒所有follower

    write\_thread\_.ExitAsBatchGroupLeader(write\_group, status);

  }

  if (status.ok()) {

    status = w.FinalStatus();

  }

  return status;

}

void WriteThread::JoinBatchGroup(Writer\* w) {

  assert(w->batch != nullptr);

// 将新的writer放到链表尾部（newest\_writer\_后面）

  bool linked\_as\_leader = LinkOne(w, &newest\_writer\_);

// 如果link后是leader，则将该w设置为LEADER状态

  if (linked\_as\_leader) {

    SetState(w, STATE\_GROUP\_LEADER);

  }

// 如果不是leader，就阻塞等待，直到w成为leader或被唤醒

  if (!linked\_as\_leader) {

    AwaitState(w,

               STATE\_GROUP\_LEADER | STATE\_MEMTABLE\_WRITER\_LEADER |

                   STATE\_PARALLEL\_MEMTABLE\_WRITER | STATE\_COMPLETED,

               &jbg\_ctx);

  }

}

bool WriteThread::LinkOne(Writer\* w, std::atomic<Writer\*>\* newest\_writer) {

  Writer\* writers = newest\_writer->load(std::memory\_order\_relaxed);

  while (true) {

    // 出现write\_stall就会阻塞等待唤醒

    if (writers == &write\_stall\_dummy\_) {

      ...

}

// 将新的写入w添加到newest\_writer后面

w->link\_older = writers;

// 如果writers等于newest\_writer，九江w赋值给newest\_writer

if (newest\_writer->compare\_exchange\_weak(writers, w)) {

// writers为空则w为第一个writer，成为leader

      return (writers == nullptr);

    }

  }

}

size\_t WriteThread::EnterAsBatchGroupLeader(Writer\* leader,

                                            WriteGroup\* write\_group) {

// 调整max\_size，避免全是小kv出现等待过久达到max\_size的情况

  size\_t size = WriteBatchInternal::ByteSize(leader->batch);

  size\_t max\_size = max\_write\_batch\_group\_size\_bytes;

  const uint64\_t min\_batch\_size\_bytes = max\_write\_batch\_group\_size\_bytes / 8;

  if (size <= min\_batch\_size\_bytes) {

    max\_size = size + min\_batch\_size\_bytes;

  }

// 创建write\_group，头尾指针都指向leader，group大小为1

  leader->write\_group = write\_group;

  write\_group->leader = leader;

  write\_group->last\_writer = leader;

  write\_group->size = 1;

// 从最新的writer开始（newest\_writer），连接1组group的writer

  Writer\* newest\_writer = newest\_writer\_.load(std::memory\_order\_acquire);

  CreateMissingNewerLinks(newest\_writer);

  Writer\* w = leader;

// 只要当前w不是newest\_writer就会一直循环

  while (w != newest\_writer) {

w = w->link\_newer;

// 很多if-break，判断follower和leader之间：

// 需要同步的writer不能放在不支持同步leader的batch中

// 是否允许延迟标志必须相同

// 是否禁用WAL标志必须相同

// protection\_bytes\_per\_key必须相同

// 速率限制器优先级必须相同

// writer的batch必须不为空

// writer的回调(callback)不为空时，leader必须允许批处理

// 当前写操作大小(size + batch\_size)需要<=max\_size

...

// 更新group大小和last\_writer位置

    w->write\_group = write\_group;

    size += batch\_size;

    write\_group->last\_writer = w;

    write\_group->size++;

  }

  return size;

}

void WriteThread::CreateMissingNewerLinks(Writer\* head) {

// 从后向前，将所有writer用link\_older连起来

  while (true) {

    Writer\* next = head->link\_older;

    if (next == nullptr || next->link\_newer != nullptr) {

      assert(next == nullptr || next->link\_newer == head);

      break;

    }

    next->link\_newer = head;

    head = next;

  }

}

void WriteThread::ExitAsBatchGroupLeader(WriteGroup& write\_group,

                                         Status& status) {

  Writer\* leader = write\_group.leader;

  Writer\* last\_writer = write\_group.last\_writer;

...

  if (enable\_pipelined\_write\_) {

    ...

  } else {

    Writer\* head = newest\_writer\_.load(std::memory\_order\_acquire);

    if (head != last\_writer ||

        !newest\_writer\_.compare\_exchange\_strong(head, nullptr)) {

      // 当前group写入期间可能有新的writer，将其双向连接

      CreateMissingNewerLinks(head);

      // 新的group的leader与当前group的last\_writer断开older关系

      last\_writer->link\_newer->link\_older = nullptr;

      // 设置当前group的last\_writer的下一个writer为下一个group的leader

      SetState(last\_writer->link\_newer, STATE\_GROUP\_LEADER);

    }

    // 遍历当前group，设置状态为STATE\_COMPLETED，唤醒其他writer

    while (last\_writer != leader) {

      last\_writer->status = status;

      auto next = last\_writer->link\_older;

      SetState(last\_writer, STATE\_COMPLETED);

      last\_writer = next;

    }

  }

}

读流程

按顺序读memtable、immemtable、sst，任何一个地方读到就返回

读sst文件时，L0层需要遍历所有sst文件，L1开始就采用二分方法读

// 与Put相似，Get也根据是否指定列族、是否传入时间戳分为4个API

// 不指定列族的会用默认列族调用指定列族的API

virtual inline Status Get(const ReadOptions& options,

                            ColumnFamilyHandle\* column\_family, const Slice& key,

                            std::string\* value) {

    PinnableSlice pinnable\_val(value);

    auto s = Get(options, column\_family, key, &pinnable\_val);

    if (s.ok() && pinnable\_val.IsPinned()) {

      value->assign(pinnable\_val.data(), pinnable\_val.size());

    }  // else value is already assigned

    return s;

  }

  virtual Status Get(const ReadOptions& options, const Slice& key,

                     std::string\* value) {

    return Get(options, DefaultColumnFamily(), key, value);

  }

virtual inline Status Get(const ReadOptions& options,

                            ColumnFamilyHandle\* column\_family, const Slice& key,

                            std::string\* value, std::string\* timestamp) {

    PinnableSlice pinnable\_val(value);

    auto s = Get(options, column\_family, key, &pinnable\_val, timestamp);

    if (s.ok() && pinnable\_val.IsPinned()) {

      value->assign(pinnable\_val.data(), pinnable\_val.size());

    }  // else value is already assigned

    return s;

  }

  virtual Status Get(const ReadOptions& options, const Slice& key,

                     std::string\* value, std::string\* timestamp) {

    return Get(options, DefaultColumnFamily(), key, value, timestamp);

  }

// 不带时间戳的会调用这个，再去调用带时间戳的DBImpl::Get

Status DBImpl::Get(const ReadOptions& read\_options,

                   ColumnFamilyHandle\* column\_family, const Slice& key,

                   PinnableSlice\* value) {

  return Get(read\_options, column\_family, key, value, /\*timestamp=\*/nullptr);

}

// 带时间戳，最后一层Get

Status DBImpl::Get(const ReadOptions& \_read\_options,

                   ColumnFamilyHandle\* column\_family, const Slice& key,

                   PinnableSlice\* value, std::string\* timestamp) {

  value->Reset();

...

  ReadOptions read\_options(\_read\_options);

  if (read\_options.io\_activity == Env::IOActivity::kUnknown) {

    read\_options.io\_activity = Env::IOActivity::kGet;

  }

  Status s = GetImpl(read\_options, column\_family, key, value, timestamp);

  return s;

}

// 第一层GetImpl，接收5个参数

Status DBImpl::GetImpl(const ReadOptions& read\_options,

                       ColumnFamilyHandle\* column\_family, const Slice& key,

                       PinnableSlice\* value, std::string\* timestamp) {

  GetImplOptions get\_impl\_options;

  get\_impl\_options.column\_family = column\_family;

  get\_impl\_options.value = value;

  get\_impl\_options.timestamp = timestamp;

  Status s = GetImpl(read\_options, key, get\_impl\_options);

  return s;

}

Status DBImpl::GetImpl(const ReadOptions& read\_options, const Slice& key,

                       GetImplOptions& get\_impl\_options) {

  // 性能统计 ...

  StopWatch sw(immutable\_db\_options\_.clock, stats\_, DB\_GET);

  PERF\_TIMER\_GUARD(get\_snapshot\_time);

  auto cfh = static\_cast\_with\_check<ColumnFamilyHandleImpl>(

      get\_impl\_options.column\_family);

  auto cfd = cfh->cfd();

...

  // 获取最新SuperVersion并+1计数

  SuperVersion\* sv = GetAndRefSuperVersion(cfd);

...

// Get默认不带snapshot，通过version\_获取最新的seq给snapshot去读

  SequenceNumber snapshot;

  if (read\_options.snapshot != nullptr) {

    if (get\_impl\_options.callback) {

      snapshot = get\_impl\_options.callback->max\_visible\_seq();

    } else {

      snapshot =

          reinterpret\_cast<const SnapshotImpl\*>(read\_options.snapshot)->number\_;

    }

  } else {

    snapshot = GetLastPublishedSequence();

if (get\_impl\_options.callback) {

...

    }

  }

  // 时间戳相关 ...

  // Prepare to store a list of merge operations if merge occurs.

  MergeContext merge\_context;

  SequenceNumber max\_covering\_tombstone\_seq = 0;

  Status s;

// 要查找的key

  LookupKey lkey(key, snapshot, read\_options.timestamp);

  PERF\_TIMER\_STOP(get\_snapshot\_time);

  bool skip\_memtable = (read\_options.read\_tier == kPersistedTier &&

                        has\_unpersisted\_data\_.load(std::memory\_order\_relaxed));

  bool done = false;

  std::string\* timestamp =

      ucmp->timestamp\_size() > 0 ? get\_impl\_options.timestamp : nullptr;

// read流程：memtable -> immemtable -> 缓存和sst

  if (!skip\_memtable) {

if (get\_impl\_options.get\_value) {

// 1.读memtable

      if (sv->mem->Get(

              lkey,

              get\_impl\_options.value ? get\_impl\_options.value->GetSelf()

                                     : nullptr,

              get\_impl\_options.columns, timestamp, &s, &merge\_context,

              &max\_covering\_tombstone\_seq, read\_options,

              false /\* immutable\_memtable \*/, get\_impl\_options.callback,

              get\_impl\_options.is\_blob\_index)) {

        done = true;

        if (get\_impl\_options.value) {

          get\_impl\_options.value->PinSelf();

        }

        RecordTick(stats\_, MEMTABLE\_HIT);

// 2.读immemtable

      } else if ((s.ok() || s.IsMergeInProgress()) &&

                 sv->imm->Get(lkey,

                              get\_impl\_options.value

                                  ? get\_impl\_options.value->GetSelf()

                                  : nullptr,

                              get\_impl\_options.columns, timestamp, &s,

                              &merge\_context, &max\_covering\_tombstone\_seq,

                              read\_options, get\_impl\_options.callback,

                              get\_impl\_options.is\_blob\_index)) {

        done = true;

        if (get\_impl\_options.value) {

          get\_impl\_options.value->PinSelf();

        }

        RecordTick(stats\_, MEMTABLE\_HIT);

      }

    } else {

      // merge操作将结果暂时缓存在merge\_context中

      if (sv->mem->Get(lkey, /\*value=\*/nullptr, /\*columns=\*/nullptr,

                       /\*timestamp=\*/nullptr, &s, &merge\_context,

                       &max\_covering\_tombstone\_seq, read\_options,

                       false /\* immutable\_memtable \*/, nullptr, nullptr,

                       false)) {

        done = true;

        RecordTick(stats\_, MEMTABLE\_HIT);

      } else if ((s.ok() || s.IsMergeInProgress()) &&

                 sv->imm->GetMergeOperands(lkey, &s, &merge\_context,

                                           &max\_covering\_tombstone\_seq,

                                           read\_options)) {

        done = true;

        RecordTick(stats\_, MEMTABLE\_HIT);

      }

    }

    if (!done && !s.ok() && !s.IsMergeInProgress()) {

      ReturnAndCleanupSuperVersion(cfd, sv);

      return s;

    }

  }

  PinnedIteratorsManager pinned\_iters\_mgr;

// 读sst

  if (!done) {

    PERF\_TIMER\_GUARD(get\_from\_output\_files\_time);

    sv->current->Get(

        read\_options, lkey, get\_impl\_options.value, get\_impl\_options.columns,

        timestamp, &s, &merge\_context, &max\_covering\_tombstone\_seq,

        &pinned\_iters\_mgr,

        get\_impl\_options.get\_value ? get\_impl\_options.value\_found : nullptr,

        nullptr, nullptr,

        get\_impl\_options.get\_value ? get\_impl\_options.callback : nullptr,

        get\_impl\_options.get\_value ? get\_impl\_options.is\_blob\_index : nullptr,

        get\_impl\_options.get\_value);

    RecordTick(stats\_, MEMTABLE\_MISS);

  }

// 读取到之后的操作，主要处理Merge操作的读取

// 引用管理和清理

// 记录性能和统计信息

  return s;

}

bool MemTable::Get(一大堆参数) {

  ...

  bool found\_final\_value = false;

  bool merge\_in\_progress = s->IsMergeInProgress();

  bool may\_contain = true;

  Slice user\_key\_without\_ts = StripTimestampFromUserKey(key.user\_key(), ts\_sz\_);

  bool bloom\_checked = false;

  if (bloom\_filter\_) {

    // 通过布隆过滤器过滤一次

    if (moptions\_.memtable\_whole\_key\_filtering) {

      may\_contain = bloom\_filter\_->MayContain(user\_key\_without\_ts);

      bloom\_checked = true;

    } else {

      assert(prefix\_extractor\_);

      if (prefix\_extractor\_->InDomain(user\_key\_without\_ts)) {

        may\_contain = bloom\_filter\_->MayContain(

            prefix\_extractor\_->Transform(user\_key\_without\_ts));

        bloom\_checked = true;

      }

    }

  }

  if (bloom\_filter\_ && !may\_contain) {

    PERF\_COUNTER\_ADD(bloom\_memtable\_miss\_count, 1);

    \*seq = kMaxSequenceNumber;

  } else {

    if (bloom\_checked) {

      PERF\_COUNTER\_ADD(bloom\_memtable\_hit\_count, 1);

}

// 通过GetFromTable读取

    GetFromTable(key, \*max\_covering\_tombstone\_seq, do\_merge, callback,

                 is\_blob\_index, value, columns, timestamp, s, merge\_context,

                 seq, &found\_final\_value, &merge\_in\_progress);

  }

  if (!found\_final\_value && merge\_in\_progress && !s->IsCorruption()) {

    \*s = Status::MergeInProgress();

  }

  PERF\_COUNTER\_ADD(get\_from\_memtable\_count, 1);

  return found\_final\_value;

}

void MemTable::GetFromTable(一大堆参数) {

// 进行一些变量赋值后，用MemTableRep去读

  table\_->Get(key, &saver, SaveValue);

  \*seq = saver.seq;

}

void MemTableRep::Get(const LookupKey& k, void\* callback\_args,

                      bool (\*callback\_func)(void\* arg, const char\* entry)) {

// 获取一个迭代器，通过这个迭代器去skiplist查询对应的key

  auto iter = GetDynamicPrefixIterator();

// 如果能在skiplist中查询到对应的key，就进入回调callback\_func函数

// 这里的回调为SaveValue函数，从iter->key()中提取key和value

  for (iter->Seek(k.internal\_key(), k.memtable\_key().data());

       iter->Valid() && callback\_func(callback\_args, iter->key());

       iter->Next()) {

  }

}

// 2.读immemtable

bool Get(一大堆参数) {

  SequenceNumber seq;

  return Get(key, value, columns, timestamp, s, merge\_context,

             max\_covering\_tombstone\_seq, &seq, read\_opts, callback,

             is\_blob\_index);

}

bool MemTableListVersion::Get(一大堆参数) {

  return GetFromList(&memlist\_, key, value, columns, timestamp, s,

                     merge\_context, max\_covering\_tombstone\_seq, seq, read\_opts,

                     callback, is\_blob\_index);

}

bool MemTableListVersion::GetFromList(一大堆参数) {

  \*seq = kMaxSequenceNumber;

// 遍历immemtable list，然后读取数据，流程同memtable

  for (auto& memtable : \*list) {

    assert(memtable->IsFragmentedRangeTombstonesConstructed());

SequenceNumber current\_seq = kMaxSequenceNumber;

    bool done =

        memtable->Get(key, value, columns, timestamp, s, merge\_context,

                      max\_covering\_tombstone\_seq, &current\_seq, read\_opts,

                      true /\* immutable\_memtable \*/, callback, is\_blob\_index);

    if (\*seq == kMaxSequenceNumber) {

      \*seq = current\_seq;

    }

    if (done) {

      assert(\*seq != kMaxSequenceNumber || s->IsNotFound());

      return true;

    }

    if (!s->ok() && !s->IsMergeInProgress() && !s->IsNotFound()) {

      return false;

    }

  }

  return false;

}

// 3.读sst

void Version::Get(一大堆参数) {

  ...

// 创建一个查询上下文

  GetContext get\_context(

      user\_comparator(), merge\_operator\_, info\_log\_, db\_statistics\_,

      status->ok() ? GetContext::kNotFound : GetContext::kMerge, user\_key,

      do\_merge ? value : nullptr, do\_merge ? columns : nullptr,

      do\_merge ? timestamp : nullptr, value\_found, merge\_context, do\_merge,

      max\_covering\_tombstone\_seq, clock\_, seq,

      merge\_operator\_ ? pinned\_iters\_mgr : nullptr, callback, is\_blob\_to\_use,

      tracing\_get\_id, &blob\_fetcher);

...

// 创建FilePicker，来选取SST进行查询

// 该version的所有SST都存放在storage\_info\_.files\_内

  FilePicker fp(user\_key, ikey, &storage\_info\_.level\_files\_brief\_,

                storage\_info\_.num\_non\_empty\_levels\_,

                &storage\_info\_.file\_indexer\_, user\_comparator(),

                internal\_comparator());

// GetNextFile以二分方式获取下一个文件（L0全部参与）

  FdWithKeyRange\* f = fp.GetNextFile();

// 遍历查询

  while (f != nullptr) {

...

// 从缓存和SST文件内，sst文件的打开和关闭封装在table\_cache\_

    \*status = table\_cache\_->Get(

        read\_options, \*internal\_comparator(), \*f->file\_metadata, ikey,

        &get\_context, mutable\_cf\_options\_.block\_protection\_bytes\_per\_key,

        mutable\_cf\_options\_.prefix\_extractor,

        cfd\_->internal\_stats()->GetFileReadHist(fp.GetHitFileLevel()),

        IsFilterSkipped(static\_cast<int>(fp.GetHitFileLevel()),

                        fp.IsHitFileLastInLevel()),

        fp.GetHitFileLevel(), max\_file\_size\_for\_l0\_meta\_pin\_);

...

    f = fp.GetNextFile();

  }

...

}

Flush

将内存中的immemtable list数据合并，生成SST文件落盘

PreprocessWrite：

检查WAL文件大小并可能切换WAL

检查内存（写缓冲区）超限并可能刷新写缓冲区检查是否需要Flush操作并可能调度Flush

三者会生成或调度Flush请求，存在flush\_queue\_中

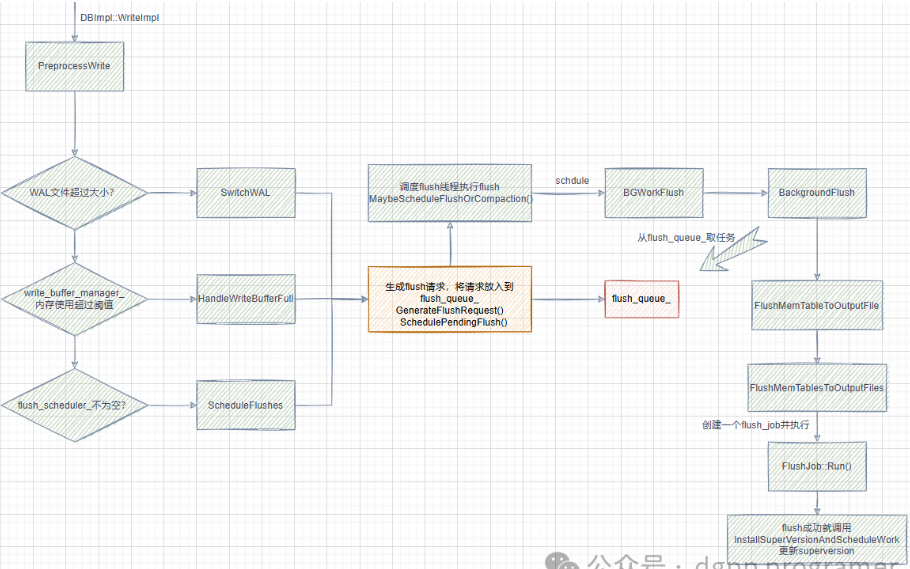
由MaybeScheduleFlushOrCompaction()进行调度

Flush请求调度到BGWorkFlush->后面如图

Write\_buffer\_size：单个memtable大小

Db\_write\_buffer\_size：DB内所有memtable的总大小（包括不同列族）

Max\_write\_buffer\_number：某个列族最大memtable数量，超过就会触发write stall



Status DBImpl::PreprocessWrite(const WriteOptions& write\_options,

                               LogContext\* log\_context,

                               WriteContext\* write\_context) {

// 如果不是单列族并且当前WAL超过大小上限则切换WAL

  if (UNLIKELY(status.ok() && total\_log\_size\_ > GetMaxTotalWalSize())) {

    InstrumentedMutexLock l(&mutex\_);

    const ColumnFamilySet\* const column\_families =

        versions\_->GetColumnFamilySet();

    size\_t num\_cfs = column\_families->NumberOfColumnFamilies();

if (num\_cfs > 1) {

// 等待其他写入线程写入完成才切换WAL（开启pipeline写才会等待）

// 不开启pipeline是单线程，可以直接写

      WaitForPendingWrites();

      status = SwitchWAL(write\_context);

    }

  }

// write\_buffer\_manager\_是否启用由db\_write\_buffer\_size决定，默认为0，不启用

// write\_buffer\_manager\_根据当前内存使用情况决定是否flush

  if (UNLIKELY(status.ok() && write\_buffer\_manager\_->ShouldFlush())) {

    // 默认不启用

    InstrumentedMutexLock l(&mutex\_);

    WaitForPendingWrites();

    status = HandleWriteBufferManagerFlush(write\_context);

  }

// trim\_history\_scheduler\_队列里存放着已经flush但还没有删除的immemtable

  if (UNLIKELY(status.ok() && !trim\_history\_scheduler\_.Empty())) {

InstrumentedMutexLock l(&mutex\_);

// 删除所有immemtable

    status = TrimMemtableHistory(write\_context);

  }

  if (UNLIKELY(status.ok() && !flush\_scheduler\_.Empty())) {

    InstrumentedMutexLock l(&mutex\_);

WaitForPendingWrites();

// 调度进行flush

    status = ScheduleFlushes(write\_context);

  }

  if (UNLIKELY(status.ok() && (write\_controller\_.IsStopped() ||

                               write\_controller\_.NeedsDelay()))) {

// 用上一轮时间预估本轮需要延迟多久写入

    InstrumentedMutexLock l(&mutex\_);

    status = DelayWrite(last\_batch\_group\_size\_, write\_thread\_, write\_options);

    PERF\_TIMER\_START(write\_pre\_and\_post\_process\_time);

  }

  // 内存超限则触发Write Stall

  if (UNLIKELY(status.ok() && write\_buffer\_manager\_->ShouldStall())) {

    default\_cf\_internal\_stats\_->AddDBStats(

        InternalStats::kIntStatsWriteBufferManagerLimitStopsCounts, 1,

        true /\* concurrent \*/);

    if (write\_options.no\_slowdown) {

      status = Status::Incomplete("Write stall");

    } else {

      InstrumentedMutexLock l(&mutex\_);

      WriteBufferManagerStallWrites();

    }

  }

  InstrumentedMutexLock l(&log\_write\_mutex\_);

  if (status.ok() && log\_context->need\_log\_sync) {

    // 如果开启sync，操作系统可能仅下刷到page cache，需要下刷到磁盘

    while (logs\_.front().IsSyncing()) {

      log\_sync\_cv\_.Wait();

    }

    for (auto& log : logs\_) {

      log.PrepareForSync();

    }

  } else {

    log\_context->need\_log\_sync = false;

  }

  log\_context->writer = logs\_.back().writer;

  log\_context->need\_log\_dir\_sync =

      log\_context->need\_log\_dir\_sync && !log\_dir\_synced\_;

  log\_context->log\_file\_number\_size = std::addressof(alive\_log\_files\_.back());

  return status;

}

Status DBImpl::SwitchWAL(WriteContext\* write\_context) {

  ...

// 选择需要切换memtable的列族

  autovector<ColumnFamilyData\*> cfds;

  if (immutable\_db\_options\_.atomic\_flush) {

// 将所有有immemtable或memtable有数据的列族放到cfds中

// 所有列族共享WAL，所以切换前要先flush其中的数据

    SelectColumnFamiliesForAtomicFlush(&cfds);

  } else {

    for (auto cfd : \*versions\_->GetColumnFamilySet()) {

      if (cfd->IsDropped()) {

        continue;

      }

// 列族下最早的一个WAL文件编号小于等于当前最早存活的WAL文件编号时都需要flush

      if (cfd->OldestLogToKeep() <= oldest\_alive\_log) {

        cfds.push\_back(cfd);

      }

}

// 可能需要带上统计信息列族一起flush

    MaybeFlushStatsCF(&cfds);

  }

  WriteThread::Writer nonmem\_w;

  if (two\_write\_queues\_) {

    nonmem\_write\_thread\_.EnterUnbatched(&nonmem\_w, &mutex\_);

  }

  for (const auto cfd : cfds) {

// 对每个列族切换memtable

    cfd->Ref();

    status = SwitchMemtable(cfd, write\_context);

    cfd->UnrefAndTryDelete();

    if (!status.ok()) {

      break;

    }

  }

  if (two\_write\_queues\_) {

    nonmem\_write\_thread\_.ExitUnbatched(&nonmem\_w);

  }

  if (status.ok()) {

    if (immutable\_db\_options\_.atomic\_flush) {

      AssignAtomicFlushSeq(cfds);

    }

for (auto cfd : cfds) {

// 将flush\_requested\_置为true，表示该列族imm list已经请求了执行flush

      cfd->imm()->FlushRequested();

      if (!immutable\_db\_options\_.atomic\_flush) {

// 生成flush\_req\_并放入flush\_queue\_队列

        FlushRequest flush\_req;

        GenerateFlushRequest({cfd}, FlushReason::kWalFull, &flush\_req);

        SchedulePendingFlush(flush\_req);

      }

    }

    if (immutable\_db\_options\_.atomic\_flush) {

      FlushRequest flush\_req;

      GenerateFlushRequest(cfds, FlushReason::kWalFull, &flush\_req);

      SchedulePendingFlush(flush\_req);

}

// 尝试启动flush或compaction任务

    MaybeScheduleFlushOrCompaction();

  }

  return status;

}

Status DBImpl::HandleWriteBufferManagerFlush(WriteContext\* write\_context) {

  ...

  autovector<ColumnFamilyData\*> cfds;

  if (immutable\_db\_options\_.atomic\_flush) {

// 将所有有immemtable或memtable有数据的列族放到cfds中

    SelectColumnFamiliesForAtomicFlush(&cfds);

  } else {

    ColumnFamilyData\* cfd\_picked = nullptr;

SequenceNumber seq\_num\_for\_cf\_picked = kMaxSequenceNumber;

// 选出所有列族最小的一个seq，内存满了只需要下刷一个列族即可

    for (auto cfd : \*versions\_->GetColumnFamilySet()) {

      if (cfd->IsDropped()) {

        continue;

      }

// 需要memtable不为空，imm没有正在flush或使用

      if (!cfd->mem()->IsEmpty() && !cfd->imm()->IsFlushPendingOrRunning()) {

        uint64\_t seq = cfd->mem()->GetCreationSeq();

        if (cfd\_picked == nullptr || seq < seq\_num\_for\_cf\_picked) {

          cfd\_picked = cfd;

          seq\_num\_for\_cf\_picked = seq;

        }

      }

    }

    if (cfd\_picked != nullptr) {

      cfds.push\_back(cfd\_picked);

    }

MaybeFlushStatsCF(&cfds); // 可能需要带上统计信息列族一起flush

  }

  // 后面的逻辑和SwitchWAL一样 ...

}

Status DBImpl::ScheduleFlushes(WriteContext\* context) {

  autovector<ColumnFamilyData\*> cfds;

  if (immutable\_db\_options\_.atomic\_flush) {

// 将所有有immemtable或memtable有数据的列族放到cfds中

    SelectColumnFamiliesForAtomicFlush(&cfds);

    for (auto cfd : cfds) {

      cfd->Ref();

    }

    flush\_scheduler\_.Clear();

  } else {

ColumnFamilyData\* tmp\_cfd;

// 调用TakeNextColumnFamily不停的从flush\_scheduler\_里的head\_链表取列族

    while ((tmp\_cfd = flush\_scheduler\_.TakeNextColumnFamily()) != nullptr) {

      cfds.push\_back(tmp\_cfd);

    }

    MaybeFlushStatsCF(&cfds); // 可能需要带上统计信息列族一起flush

  }

  Status status;

  WriteThread::Writer nonmem\_w;

  if (two\_write\_queues\_) {

// 默认不启用

    nonmem\_write\_thread\_.EnterUnbatched(&nonmem\_w, &mutex\_);

  }

  TEST\_SYNC\_POINT\_CALLBACK("DBImpl::ScheduleFlushes:PreSwitchMemtable",

                           nullptr);

  for (auto& cfd : cfds) {

// 需要memtable不为空，避免重入，比如某个cf切换后再没写过数据

    if (status.ok() && !cfd->mem()->IsEmpty()) {

      status = SwitchMemtable(cfd, context);

    }

    if (cfd->UnrefAndTryDelete()) {

      cfd = nullptr;

    }

  }

  if (two\_write\_queues\_) {

    nonmem\_write\_thread\_.ExitUnbatched(&nonmem\_w);

  }

  // 后面的逻辑和SwitchWAL一样 ...

}

// StatsCF来专门存储统计信息，有需求时启用

void DBImpl::MaybeFlushStatsCF(autovector<ColumnFamilyData\*>\* cfds) {

  assert(cfds != nullptr);

// 如果传入的cfds不为空，并且启用了StatsCF，则尝试将该列族一起flush

  if (!cfds->empty() && immutable\_db\_options\_.persist\_stats\_to\_disk) {

// 统计信息的列族名称为kPersistentStatsColumnFamilyName

    ColumnFamilyData\* cfd\_stats =

        versions\_->GetColumnFamilySet()->GetColumnFamily(

            kPersistentStatsColumnFamilyName);

    if (cfd\_stats != nullptr && !cfd\_stats->mem()->IsEmpty()) {

      for (ColumnFamilyData\* cfd : \*cfds) {

        if (cfd == cfd\_stats) {

          // StatsCF已经在cfds里了

          return;

        }

      }

      // 如果StatsCF不在cfds中，则确认StatsCF的WAL日志编号是否为所有列族最小

      bool force\_flush\_stats\_cf = true;

      for (auto\* loop\_cfd : \*versions\_->GetColumnFamilySet()) {

        if (loop\_cfd == cfd\_stats) {

// 无需与自己比较

          continue;

        }

        if (loop\_cfd->GetLogNumber() <= cfd\_stats->GetLogNumber()) {

// 当前StatsCF的WAL编号不是最小的，无需强制flush

          force\_flush\_stats\_cf = false;

        }

      }

      if (force\_flush\_stats\_cf) {

// 当前StatsCF的WAL编号是最小的，需要强制flush

        cfds->push\_back(cfd\_stats);

        ROCKS\_LOG\_INFO(immutable\_db\_options\_.info\_log,

                       "Force flushing stats CF with automated flush "

                       "to avoid holding old logs");

      }

    }

  }

}

// 切换memtable

Status DBImpl::SwitchMemtable(ColumnFamilyData\* cfd, WriteContext\* context) {

  ...

  // 将cached\_recoverable\_state\_写入到memtable

  Status s = WriteRecoverableState();

  if (!s.ok()) {

    return s;

  }

...

  uint64\_t recycle\_log\_number = 0;

// 如果需要创建新的WAL文件，尝试从log\_recycle\_files\_中获取可以复用的WAL文件

  if (creating\_new\_log && immutable\_db\_options\_.recycle\_log\_file\_num &&

      !log\_recycle\_files\_.empty()) {

    recycle\_log\_number = log\_recycle\_files\_.front();

  }

  uint64\_t new\_log\_number =

      creating\_new\_log ? versions\_->NewFileNumber() : logfile\_number\_;

  const MutableCFOptions mutable\_cf\_options = \*cfd->GetLatestMutableCFOptions();

  // 收集一些信息到MemTableInfo对象中 ...

  // immemtable个数

  int num\_imm\_unflushed = cfd->imm()->NumNotFlushed();

// 预先计算出要创建WAL文件大小（如果需要创建新WAL文件的话）

  const auto preallocate\_block\_size =

      GetWalPreallocateBlockSize(mutable\_cf\_options.write\_buffer\_size);

  mutex\_.Unlock();

  if (creating\_new\_log) {

    // 创建WAL文件

    io\_s = CreateWAL(write\_options, new\_log\_number, recycle\_log\_number,

                     preallocate\_block\_size, &new\_log);

    if (s.ok()) {

      s = io\_s;

    }

  }

  if (s.ok()) {

// 创建新的memtable

    SequenceNumber seq = versions\_->LastSequence();

    new\_mem = cfd->ConstructNewMemtable(mutable\_cf\_options, seq);

    context->superversion\_context.NewSuperVersion();

  }

  cfd->mem()->ConstructFragmentedRangeTombstones();

  mutex\_.Lock();

  if (recycle\_log\_number != 0) {

    assert(log\_recycle\_files\_.front() == recycle\_log\_number);

    log\_recycle\_files\_.pop\_front();

  }

  if (s.ok() && creating\_new\_log) {

    InstrumentedMutexLock l(&log\_write\_mutex\_);

    assert(new\_log != nullptr);

    if (!logs\_.empty()) {

      // Alway flush the buffer of the last log before switching to a new one

      log::Writer\* cur\_log\_writer = logs\_.back().writer;

      if (error\_handler\_.IsRecoveryInProgress()) {

        // WAL文件下刷到操作系统或磁盘（WAL文件有一个缓冲区，控制每次写WAL文件是否要flush）

        cur\_log\_writer->file()->reset\_seen\_error();

      }

      io\_s = cur\_log\_writer->WriteBuffer(write\_options);

      if (s.ok()) {

        s = io\_s;

      }

      if (!s.ok()) {

        ROCKS\_LOG\_WARN(immutable\_db\_options\_.info\_log,

                       "[%s] Failed to switch from #%" PRIu64 " to #%" PRIu64

                       "  WAL file\n",

                       cfd->GetName().c\_str(), cur\_log\_writer->get\_log\_number(),

                       new\_log\_number);

      }

    }

    if (s.ok()) {

      logfile\_number\_ = new\_log\_number;

      log\_empty\_ = true;

      log\_dir\_synced\_ = false;

      logs\_.emplace\_back(logfile\_number\_, new\_log);

      alive\_log\_files\_.push\_back(LogFileNumberSize(logfile\_number\_));

    }

  }

// 创建失败的操作

  if (!s.ok()) {

    // how do we fail if we're not creating new log?

    assert(creating\_new\_log);

    delete new\_mem;

    delete new\_log;

    context->superversion\_context.new\_superversion.reset();

    // We may have lost data from the WritableFileBuffer in-memory buffer for

    // the current log, so treat it as a fatal error and set bg\_error

    if (!io\_s.ok()) {

      error\_handler\_.SetBGError(io\_s, BackgroundErrorReason::kMemTable);

    } else {

      error\_handler\_.SetBGError(s, BackgroundErrorReason::kMemTable);

    }

    // Read back bg\_error in order to get the right severity

    s = error\_handler\_.GetBGError();

    return s;

  }

  cfd->mem()->SetNextLogNumber(logfile\_number\_);

  // 将memtable添加到immemtable

  cfd->imm()->Add(cfd->mem(), &context->memtables\_to\_free\_);

  new\_mem->Ref();

// 设置刚创建的memtable为列族使用的memtable

  cfd->SetMemtable(new\_mem);

// 更新superversion，内部会尝试触发flush和compaction

  InstallSuperVersionAndScheduleWork(cfd, &context->superversion\_context,

                                     mutable\_cf\_options);

...

  return s;

}

// 遍历cfds，生成一批flush\_req\_

void DBImpl::GenerateFlushRequest(const autovector<ColumnFamilyData\*>& cfds,

                                  FlushReason flush\_reason, FlushRequest\* req) {

  assert(req != nullptr);

  req->flush\_reason = flush\_reason;

  req->cfd\_to\_max\_mem\_id\_to\_persist.reserve(cfds.size());

  for (const auto cfd : cfds) {

    if (nullptr == cfd) {

      // cfd may be null, see DBImpl::ScheduleFlushes

      continue;

    }

    uint64\_t max\_memtable\_id = cfd->imm()->GetLatestMemTableID(

        immutable\_db\_options\_.atomic\_flush /\* for\_atomic\_flush \*/);

    req->cfd\_to\_max\_mem\_id\_to\_persist.emplace(cfd, max\_memtable\_id);

  }

}

// 将需要flush的任务放入到flush\_queue\_队列

void DBImpl::SchedulePendingFlush(const FlushRequest& flush\_req) {

  mutex\_.AssertHeld();

  if (reject\_new\_background\_jobs\_) {

    return;

  }

  if (flush\_req.cfd\_to\_max\_mem\_id\_to\_persist.empty()) {

    return;

  }

  if (!immutable\_db\_options\_.atomic\_flush) {

    // For the non-atomic flush case, we never schedule multiple column

    // families in the same flush request.

    assert(flush\_req.cfd\_to\_max\_mem\_id\_to\_persist.size() == 1);

    ColumnFamilyData\* cfd =

        flush\_req.cfd\_to\_max\_mem\_id\_to\_persist.begin()->first;

    assert(cfd);

    if (!cfd->queued\_for\_flush() && cfd->imm()->IsFlushPending()) {

      cfd->Ref();

      cfd->set\_queued\_for\_flush(true);

      ++unscheduled\_flushes\_;

      flush\_queue\_.push\_back(flush\_req);

    }

  } else {

    for (auto& iter : flush\_req.cfd\_to\_max\_mem\_id\_to\_persist) {

      ColumnFamilyData\* cfd = iter.first;

      cfd->Ref();

    }

    ++unscheduled\_flushes\_;

    flush\_queue\_.push\_back(flush\_req);

  }

}

void DBImpl::MaybeScheduleFlushOrCompaction() {

  ...

  if (bg\_work\_paused\_ > 0) {

    // 后台的flush和compaction可以暂停

    return;

  }

// 还有两个导致直接return;的判断 ...

  auto bg\_job\_limits = GetBGJobLimits();

  bool is\_flush\_pool\_empty =

      env\_->GetBackgroundThreads(Env::Priority::HIGH) == 0;

// 条件：1.HIGH线程池不为空（优先用HIGH线程执行flush），2.未执行的flush任务大于0，

// 3.后台正常执行的flush任务小于后台限制执行的flush任务数

  while (!is\_flush\_pool\_empty && unscheduled\_flushes\_ > 0 &&

         bg\_flush\_scheduled\_ < bg\_job\_limits.max\_flushes) {

    bg\_flush\_scheduled\_++;

    FlushThreadArg\* fta = new FlushThreadArg;

    fta->db\_ = this;

fta->thread\_pri\_ = Env::Priority::HIGH;

// 使用HIGH线程执行Flush

    env\_->Schedule(&DBImpl::BGWorkFlush, fta, Env::Priority::HIGH, this,

                   &DBImpl::UnscheduleFlushCallback);

    --unscheduled\_flushes\_;

  }

  // 如果HIGH线程池为空，并且有待flush任务、后台flush和compaction任务总数小于上限

// 则可以用LOW线程执行flush，保证优先执行flush，确保写性能

  if (is\_flush\_pool\_empty) {

    while (unscheduled\_flushes\_ > 0 &&

           bg\_flush\_scheduled\_ + bg\_compaction\_scheduled\_ <

               bg\_job\_limits.max\_flushes) {

      bg\_flush\_scheduled\_++;

      FlushThreadArg\* fta = new FlushThreadArg;

      fta->db\_ = this;

      fta->thread\_pri\_ = Env::Priority::LOW;

      env\_->Schedule(&DBImpl::BGWorkFlush, fta, Env::Priority::LOW, this,

                     &DBImpl::UnscheduleFlushCallback);

      --unscheduled\_flushes\_;

    }

  }

// 如果后台有单独停止的compaction任务，则返回而不停止flush任务

  if (bg\_compaction\_paused\_ > 0) {

    // we paused the background compaction

    return;

  } else if (error\_handler\_.IsBGWorkStopped()) {

    // Compaction is not part of the recovery sequence from a hard error. We

    // might get here because recovery might do a flush and install a new

    // super version, which will try to schedule pending compactions. Bail

    // out here and let the higher level recovery handle compactions

    return;

  }

  if (HasExclusiveManualCompaction()) {

    // manual compaction可以独占任务，其他compaction无法执行

    TEST\_SYNC\_POINT("DBImpl::MaybeScheduleFlushOrCompaction:Conflict");

    return;

  }

// 如果有可以调度的compaction，并且总任务数小于上限，则用LOW线程执行compaction

  while (bg\_compaction\_scheduled\_ + bg\_bottom\_compaction\_scheduled\_ <

             bg\_job\_limits.max\_compactions &&

         unscheduled\_compactions\_ > 0) {

    CompactionArg\* ca = new CompactionArg;

    ca->db = this;

    ca->compaction\_pri\_ = Env::Priority::LOW;

    ca->prepicked\_compaction = nullptr;

    bg\_compaction\_scheduled\_++;

    unscheduled\_compactions\_--;

    env\_->Schedule(&DBImpl::BGWorkCompaction, ca, Env::Priority::LOW, this,

                   &DBImpl::UnscheduleCompactionCallback);

  }

}

Flush主流程：

void DBImpl::BGWorkFlush(void\* arg) {

  FlushThreadArg fta = \*(static\_cast<FlushThreadArg\*>(arg));

  delete static\_cast<FlushThreadArg\*>(arg);

  IOSTATS\_SET\_THREAD\_POOL\_ID(fta.thread\_pri\_);

  TEST\_SYNC\_POINT("DBImpl::BGWorkFlush");

// 进入BackgroundCallFlush

  static\_cast\_with\_check<DBImpl>(fta.db\_)->BackgroundCallFlush(fta.thread\_pri\_);

  TEST\_SYNC\_POINT("DBImpl::BGWorkFlush:done");

}

void DBImpl::BackgroundCallFlush(Env::Priority thread\_pri) {

  bool made\_progress = false;

  JobContext job\_context(next\_job\_id\_.fetch\_add(1), true);

  LogBuffer log\_buffer(InfoLogLevel::INFO\_LEVEL,

                       immutable\_db\_options\_.info\_log.get());

  {

    InstrumentedMutexLock l(&mutex\_);

    assert(bg\_flush\_scheduled\_);

num\_running\_flushes\_++;

// 记录flush的文件number，完成后从pending\_outputs\_里删除

    std::unique\_ptr<std::list<uint64\_t>::iterator>

        pending\_outputs\_inserted\_elem(new std::list<uint64\_t>::iterator(

            CaptureCurrentFileNumberInPendingOutputs()));

    FlushReason reason;

bool flush\_rescheduled\_to\_retain\_udt = false;

// 执行Flush

    Status s =

        BackgroundFlush(&made\_progress, &job\_context, &log\_buffer, &reason,

                        &flush\_rescheduled\_to\_retain\_udt, thread\_pri);

    if (s.IsTryAgain() && flush\_rescheduled\_to\_retain\_udt) {

      bg\_cv\_.SignalAll();  // In case a waiter can proceed despite the error

      mutex\_.Unlock();

      TEST\_SYNC\_POINT\_CALLBACK("DBImpl::AfterRetainUDTReschedule:cb", nullptr);

      immutable\_db\_options\_.clock->SleepForMicroseconds(

          100000);  // prevent hot loop

      mutex\_.Lock();

    } else if (!s.ok() && !s.IsShutdownInProgress() &&

               !s.IsColumnFamilyDropped() &&

               reason != FlushReason::kErrorRecovery) {

// Wait a little bit before retrying background flush in

      // case this is an environmental problem and we do not want to

      // chew up resources for failed flushes for the duration of

      // the problem.

// Flush失败

      uint64\_t error\_cnt =

          default\_cf\_internal\_stats\_->BumpAndGetBackgroundErrorCount();

      bg\_cv\_.SignalAll();  // In case a waiter can proceed despite the error

      mutex\_.Unlock();

      ROCKS\_LOG\_ERROR(immutable\_db\_options\_.info\_log,

                      "[JOB %d] Waiting after background flush error: %s"

                      "Accumulated background error counts: %" PRIu64,

                      job\_context.job\_id, s.ToString().c\_str(), error\_cnt);

      log\_buffer.FlushBufferToLog();

      LogFlush(immutable\_db\_options\_.info\_log);

      immutable\_db\_options\_.clock->SleepForMicroseconds(1000000);

      mutex\_.Lock();

}

TEST\_SYNC\_POINT("DBImpl::BackgroundCallFlush:FlushFinish:0");

// 从pending\_outputs\_中删除记录的number

    ReleaseFileNumberFromPendingOutputs(pending\_outputs\_inserted\_elem);

    if (!flush\_rescheduled\_to\_retain\_udt) {

      // 如果flush失败，需要获取所有临时文件用于删除

      FindObsoleteFiles(&job\_context, !s.ok() && !s.IsShutdownInProgress() &&

                                          !s.IsColumnFamilyDropped());

    }

    // delete unnecessary files if any, this is done outside the mutex

    if (job\_context.HaveSomethingToClean() ||

        job\_context.HaveSomethingToDelete() || !log\_buffer.IsEmpty()) {

      mutex\_.Unlock();

    // 刷日志

      log\_buffer.FlushBufferToLog();

      if (job\_context.HaveSomethingToDelete()) {

// 删除临时文件

        PurgeObsoleteFiles(job\_context);

      }

      job\_context.Clean();

      mutex\_.Lock();

    }

TEST\_SYNC\_POINT("DBImpl::BackgroundCallFlush:ContextCleanedUp");

    assert(num\_running\_flushes\_ > 0);

    num\_running\_flushes\_--;

    bg\_flush\_scheduled\_--;

    // 再次尝试调度，确认是否有其他任务

    MaybeScheduleFlushOrCompaction();

atomic\_flush\_install\_cv\_.SignalAll();

// 尝试唤醒

    bg\_cv\_.SignalAll();

  }

}

Status DBImpl::BackgroundFlush(bool\* made\_progress, JobContext\* job\_context,

                               LogBuffer\* log\_buffer, FlushReason\* reason,

                               bool\* flush\_rescheduled\_to\_retain\_udt,

                               Env::Priority thread\_pri) {

  ...

// 取出flush\_queue\_中的所有flush请求，放入到bg\_flush\_args

  while (!flush\_queue\_.empty()) {

    // This cfd is already referenced

    FlushRequest flush\_req = PopFirstFromFlushQueue();

    FlushReason flush\_reason = flush\_req.flush\_reason;

    // 各种导致不能进行刷新的原因，return ...

    superversion\_contexts.clear();

    superversion\_contexts.reserve(

        flush\_req.cfd\_to\_max\_mem\_id\_to\_persist.size());

    for (const auto& [cfd, max\_memtable\_id] :

         flush\_req.cfd\_to\_max\_mem\_id\_to\_persist) {

// 如果列族启用MemPurge（内存清楚），则标记为已请求Flush

      if (cfd->GetMempurgeUsed()) {

        cfd->imm()->FlushRequested();

      }

// 如果列族被删除，或列族的immemtable不在flush pending队列中，则不能flush该列族

      if (cfd->IsDropped() || !cfd->imm()->IsFlushPending()) {

        column\_families\_not\_to\_flush.push\_back(cfd);

        continue;

      }

      superversion\_contexts.emplace\_back(SuperVersionContext(true));

      bg\_flush\_args.emplace\_back(cfd, max\_memtable\_id,

                                 &(superversion\_contexts.back()), flush\_reason);

}

    if (!bg\_flush\_args.empty() || !column\_families\_not\_to\_flush.empty()) {

      TEST\_SYNC\_POINT\_CALLBACK("DBImpl::BackgroundFlush:CheckFlushRequest:cb",

                               const\_cast<int\*>(&flush\_req.reschedule\_count));

      break;

    }

  }

  if (!bg\_flush\_args.empty()) {

// 是否超过flush个数限制

    auto bg\_job\_limits = GetBGJobLimits();

    for (const auto& arg : bg\_flush\_args) {

      ColumnFamilyData\* cfd = arg.cfd\_;

      ROCKS\_LOG\_BUFFER(

          log\_buffer,

          "Calling FlushMemTableToOutputFile with column "

          "family [%s], flush slots available %d, compaction slots available "

          "%d, "

          "flush slots scheduled %d, compaction slots scheduled %d",

          cfd->GetName().c\_str(), bg\_job\_limits.max\_flushes,

          bg\_job\_limits.max\_compactions, bg\_flush\_scheduled\_,

          bg\_compaction\_scheduled\_);

}

// 将memtable刷到SST文件

    status = FlushMemTablesToOutputFiles(bg\_flush\_args, made\_progress,

                                         job\_context, log\_buffer, thread\_pri);

...

// flush的原因，用于打日志

    \*reason = bg\_flush\_args[0].flush\_reason\_;

    for (auto& arg : bg\_flush\_args) {

      ColumnFamilyData\* cfd = arg.cfd\_;

      if (cfd->UnrefAndTryDelete()) {

        arg.cfd\_ = nullptr;

      }

    }

  }

  for (auto cfd : column\_families\_not\_to\_flush) {

    cfd->UnrefAndTryDelete();

  }

  return status;

}

Status DBImpl::FlushMemTablesToOutputFiles(

    const autovector<BGFlushArg>& bg\_flush\_args, bool\* made\_progress,

    JobContext\* job\_context, LogBuffer\* log\_buffer, Env::Priority thread\_pri) {

  if (immutable\_db\_options\_.atomic\_flush) {

// 原子下刷memtable，默认不启用

    return AtomicFlushMemTablesToOutputFiles(

        bg\_flush\_args, made\_progress, job\_context, log\_buffer, thread\_pri);

  }

  assert(bg\_flush\_args.size() == 1);

  std::vector<SequenceNumber> snapshot\_seqs;

  SequenceNumber earliest\_write\_conflict\_snapshot;

  SnapshotChecker\* snapshot\_checker;

  GetSnapshotContext(job\_context, &snapshot\_seqs,

                     &earliest\_write\_conflict\_snapshot, &snapshot\_checker);

// 新版本似乎将遍历bg\_flush\_args改成了指定头指针的方式

  const auto& bg\_flush\_arg = bg\_flush\_args[0];

// BGFlushArg结构体一共四个成员变量，包括下面三个和max\_memtable\_id

  ColumnFamilyData\* cfd = bg\_flush\_arg.cfd\_;

  MutableCFOptions mutable\_cf\_options\_copy = \*cfd->GetLatestMutableCFOptions();

  SuperVersionContext\* superversion\_context =

      bg\_flush\_arg.superversion\_context\_;

  FlushReason flush\_reason = bg\_flush\_arg.flush\_reason\_;

  Status s = FlushMemTableToOutputFile(

      cfd, mutable\_cf\_options\_copy, made\_progress, job\_context, flush\_reason,

      superversion\_context, snapshot\_seqs, earliest\_write\_conflict\_snapshot,

      snapshot\_checker, log\_buffer, thread\_pri);

  return s;

}

// 将所有immemtable刷到一个SST文件内

Status DBImpl::FlushMemTableToOutputFile(

    ColumnFamilyData\* cfd, const MutableCFOptions& mutable\_cf\_options,

    bool\* made\_progress, JobContext\* job\_context, FlushReason flush\_reason,

    SuperVersionContext\* superversion\_context,

    std::vector<SequenceNumber>& snapshot\_seqs,

    SequenceNumber earliest\_write\_conflict\_snapshot,

    SnapshotChecker\* snapshot\_checker, LogBuffer\* log\_buffer,

    Env::Priority thread\_pri) {

  ...

  // 如果有多个列族，则确保除最新WAL文件之外的WAL文件同步

// 否则如果崩溃发生在flush之后、WAL落盘之前，该sst文件可能包含某些CF的更新，

// 但其他CF的更新还没被记录，导致数据不一致

  const bool needs\_to\_sync\_closed\_wals =

      logfile\_number\_ > 0 &&

      versions\_->GetColumnFamilySet()->NumberOfColumnFamilies() > 1;

  // 有多个列族需要处理时，那么记录当前最新memtable的id，落盘不能处理比它大的memtable

// 因为SyncCloseLogs()可能释放db互斥锁，而后可能产生新的memtable切换

  uint64\_t max\_memtable\_id =

      needs\_to\_sync\_closed\_wals

          ? cfd->imm()->GetLatestMemTableID(false /\* for\_atomic\_flush \*/)

          : std::numeric\_limits<uint64\_t>::max();

  // 处理单个列族时，flush会在PickMemTable()中刷新所有memtable

  // 不会调用SyncCloseLogs()，但仍然可能在NotifyOnFlushBegin()中释放和获取互斥锁

  // 互斥锁释放过程中，应用可能创建新的快照（记录数据库所有可见状态）

// 为避免这样导致数据不一致，要保证NotifyOnFlushBegin()在memtable选择后执行

// 生成一个flush任务

  FlushJob flush\_job(

      dbname\_, cfd, immutable\_db\_options\_, mutable\_cf\_options, max\_memtable\_id,

      file\_options\_for\_compaction\_, versions\_.get(), &mutex\_, &shutting\_down\_,

      snapshot\_seqs, earliest\_write\_conflict\_snapshot, snapshot\_checker,

      job\_context, flush\_reason, log\_buffer, directories\_.GetDbDir(),

      GetDataDir(cfd, 0U),

      GetCompressionFlush(\*cfd->ioptions(), mutable\_cf\_options), stats\_,

      &event\_logger\_, mutable\_cf\_options.report\_bg\_io\_stats,

      true /\* sync\_output\_directory \*/, true /\* write\_manifest \*/, thread\_pri,

      io\_tracer\_, seqno\_to\_time\_mapping\_, db\_id\_, db\_session\_id\_,

      cfd->GetFullHistoryTsLow(), &blob\_callback\_);

  FileMetaData file\_meta;

  Status s;

  bool need\_cancel = false;

  IOStatus log\_io\_s = IOStatus::OK();

  if (needs\_to\_sync\_closed\_wals) {

    // 有多个列族需要处理时，先进行同步（除最新WAL文件都落盘）

    VersionEdit synced\_wals;

    bool error\_recovery\_in\_prog = error\_handler\_.IsRecoveryInProgress();

mutex\_.Unlock();

// 在此期间可能产生新的memtable

// 对WAL文件落盘

    log\_io\_s = SyncClosedLogs(write\_options, job\_context, &synced\_wals,

                              error\_recovery\_in\_prog);

    mutex\_.Lock();

    if (log\_io\_s.ok() && synced\_wals.IsWalAddition()) {

      log\_io\_s = status\_to\_io\_status(

          ApplyWALToManifest(read\_options, write\_options, &synced\_wals));

      TEST\_SYNC\_POINT\_CALLBACK("DBImpl::FlushMemTableToOutputFile:CommitWal:1",

                               nullptr);

}

    if (!log\_io\_s.ok() && !log\_io\_s.IsShutdownInProgress() &&

        !log\_io\_s.IsColumnFamilyDropped()) {

      error\_handler\_.SetBGError(log\_io\_s, BackgroundErrorReason::kFlush);

    }

  } else {

    TEST\_SYNC\_POINT("DBImpl::SyncClosedLogs:Skip");

  }

  s = log\_io\_s;

  // If the log sync failed, we do not need to pick memtable. Otherwise,

  // num\_flush\_not\_started\_ needs to be rollback.

  TEST\_SYNC\_POINT("DBImpl::FlushMemTableToOutputFile:BeforePickMemtables");

  // Exit a flush due to bg error should not set bg error again.

  bool skip\_set\_bg\_error = false;

  if (s.ok() && !error\_handler\_.GetBGError().ok() &&

      error\_handler\_.IsBGWorkStopped() &&

      flush\_reason != FlushReason::kErrorRecovery &&

      flush\_reason != FlushReason::kErrorRecoveryRetryFlush) {

    // Error recovery in progress, should not pick memtable which excludes

    // them from being picked up by recovery flush.

    // This ensures that when bg error is set, no new flush can pick

    // memtables.

    skip\_set\_bg\_error = true;

    s = error\_handler\_.GetBGError();

    assert(!s.ok());

    ROCKS\_LOG\_BUFFER(log\_buffer,

                     "[JOB %d] Skip flush due to background error %s",

                     job\_context->job\_id, s.ToString().c\_str());

  }

  if (s.ok()) {

// 选取memtable，生成的sst文件大小和memtable个数有关

    flush\_job.PickMemTable();

    need\_cancel = true;

  }

  TEST\_SYNC\_POINT\_CALLBACK(

      "DBImpl::FlushMemTableToOutputFile:AfterPickMemtables", &flush\_job);

  // may temporarily unlock and lock the mutex.

  NotifyOnFlushBegin(cfd, &file\_meta, mutable\_cf\_options, job\_context->job\_id,

                     flush\_reason);

  bool switched\_to\_mempurge = false;

  // 执行flush

  if (s.ok()) {

    s = flush\_job.Run(&logs\_with\_prep\_tracker\_, &file\_meta,

                      &switched\_to\_mempurge, &skip\_set\_bg\_error,

                      &error\_handler\_);

    need\_cancel = false;

  }

  if (!s.ok() && need\_cancel) {

    flush\_job.Cancel();

  }

  if (s.ok()) {

// 固化（更新）superversion

    InstallSuperVersionAndScheduleWork(cfd, superversion\_context,

                                       mutable\_cf\_options);

    ...

  }

  if (!s.ok() && !s.IsShutdownInProgress() && !s.IsColumnFamilyDropped() &&

      !skip\_set\_bg\_error) {

    ...

  }

  // 成功flush并且没有发生内存清除，则安装新的sst文件路径

  if (s.ok() && (!switched\_to\_mempurge)) {

    ...

  }

  TEST\_SYNC\_POINT("DBImpl::FlushMemTableToOutputFile:Finish");

  return s;

}

Status FlushJob::Run(LogsWithPrepTracker\* prep\_tracker, FileMetaData\* file\_meta,

                     bool\* switched\_to\_mempurge, bool\* skipped\_since\_bg\_error,

                     ErrorHandler\* error\_handler) {

  ...

// 发生内存清除

  if ((mempurge\_threshold > 0.0) &&

      (flush\_reason\_ == FlushReason::kWriteBufferFull) && (!mems\_.empty()) &&

      MemPurgeDecider(mempurge\_threshold) && !(db\_options\_.atomic\_flush)) {

...

  }

  Status s;

  if (mempurge\_s.ok()) {

    base\_->Unref();

    s = Status::OK();

  } else {

// This will release and re-acquire the mutex.

// 无需内存清除时，写Level0

    s = WriteLevel0Table();

  }

...

  if (!s.ok()) {

// 失败回滚，让memtable参与下次flush

    cfd\_->imm()->RollbackMemtableFlush(

        mems\_, /\*rollback\_succeeding\_memtables=\*/!db\_options\_.atomic\_flush);

  } else if (write\_manifest\_) {

    // 更新MANIFEST（版本文件）

  }

  if (s.ok() && file\_meta != nullptr) {

    \*file\_meta = meta\_;

  }

  RecordFlushIOStats();

...

  return s;

}

Status FlushJob::WriteLevel0Table() {

  AutoThreadOperationStageUpdater stage\_updater(

      ThreadStatus::STAGE\_FLUSH\_WRITE\_L0);

  db\_mutex\_->AssertHeld();

...

  {

...

// 遍历memtable，放入到memtables，收集del\_range并更新统计信息

    for (MemTable\* m : mems\_) {

      ROCKS\_LOG\_INFO(

          db\_options\_.info\_log,

          "[%s] [JOB %d] Flushing memtable with next log file: %" PRIu64 "\n",

          cfd\_->GetName().c\_str(), job\_context\_->job\_id, m->GetNextLogNumber());

// 用迭代器访问memtable

      memtables.push\_back(m->NewIterator(ro, &arena));

      auto\* range\_del\_iter = m->NewRangeTombstoneIterator(

          ro, kMaxSequenceNumber, true /\* immutable\_memtable \*/);

      if (range\_del\_iter != nullptr) {

        range\_del\_iters.emplace\_back(range\_del\_iter);

      }

      total\_num\_entries += m->num\_entries();

      total\_num\_deletes += m->num\_deletes();

      total\_data\_size += m->get\_data\_size();

      total\_memory\_usage += m->ApproximateMemoryUsage();

      total\_num\_range\_deletes += m->num\_range\_deletes();

    }

    ...

{

// 生成一个大的迭代器，用于遍历所有memtable的kv

      ScopedArenaIterator iter(

          NewMergingIterator(&cfd\_->internal\_comparator(), memtables.data(),

                             static\_cast<int>(memtables.size()), &arena));

      ...

// 遍历Key Value后生成sst文件

      s = BuildTable(

          dbname\_, versions\_, db\_options\_, tboptions, file\_options\_,

          cfd\_->table\_cache(), iter.get(), std::move(range\_del\_iters), &meta\_,

          &blob\_file\_additions, existing\_snapshots\_,

          earliest\_write\_conflict\_snapshot\_, job\_snapshot\_seq,

          snapshot\_checker\_, mutable\_cf\_options\_.paranoid\_file\_checks,

          cfd\_->internal\_stats(), &io\_s, io\_tracer\_,

          BlobFileCreationReason::kFlush, seqno\_to\_time\_mapping\_, event\_logger\_,

          job\_context\_->job\_id, &table\_properties\_, write\_hint,

          full\_history\_ts\_low, blob\_callback\_, base\_, &num\_input\_entries,

          &memtable\_payload\_bytes, &memtable\_garbage\_bytes);

      ...

      LogFlush(db\_options\_.info\_log);

    }

    ...

  }

  ...

  if (s.ok() && has\_output) {

// 将生成的文件添加到version中，一次flush只会生成一个sst文件

    edit\_->AddFile(0 /\* level \*/, meta\_.fd.GetNumber(), meta\_.fd.GetPathId(),

                   meta\_.fd.GetFileSize(), meta\_.smallest, meta\_.largest,

                   meta\_.fd.smallest\_seqno, meta\_.fd.largest\_seqno,

                   meta\_.marked\_for\_compaction, meta\_.temperature,

                   meta\_.oldest\_blob\_file\_number, meta\_.oldest\_ancester\_time,

                   meta\_.file\_creation\_time, meta\_.epoch\_number,

                   meta\_.file\_checksum, meta\_.file\_checksum\_func\_name,

                   meta\_.unique\_id, meta\_.compensated\_range\_deletion\_size,

                   meta\_.tail\_size, meta\_.user\_defined\_timestamps\_persisted);

    edit\_->SetBlobFileAdditions(std::move(blob\_file\_additions));

  }

  ...

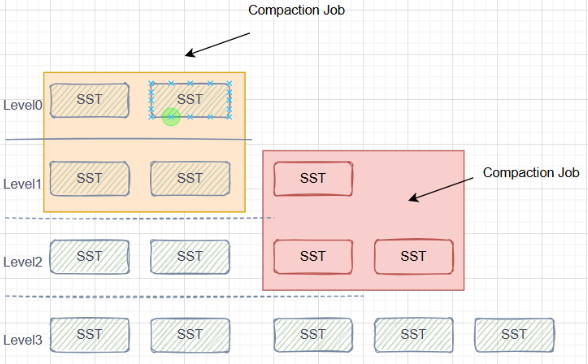
  return s;

}

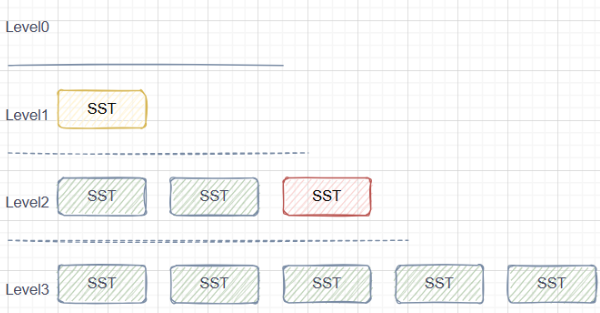
Compaction

默认指major compaction。Rocksdb采用追加写方式，对一个key执行插入/删除操作后，该key之前所有的插入操作都没有作用了；对一个key执行Merge更新（增量写入）时，读取时需要读完所有增量才可以返回，同时多个key-增量对也占用了额外空间。这些情况都造成了写放大/读放大，去除这些重复的key使用的就是compaction操作，从而降低rocksdb的写放大/读放大。

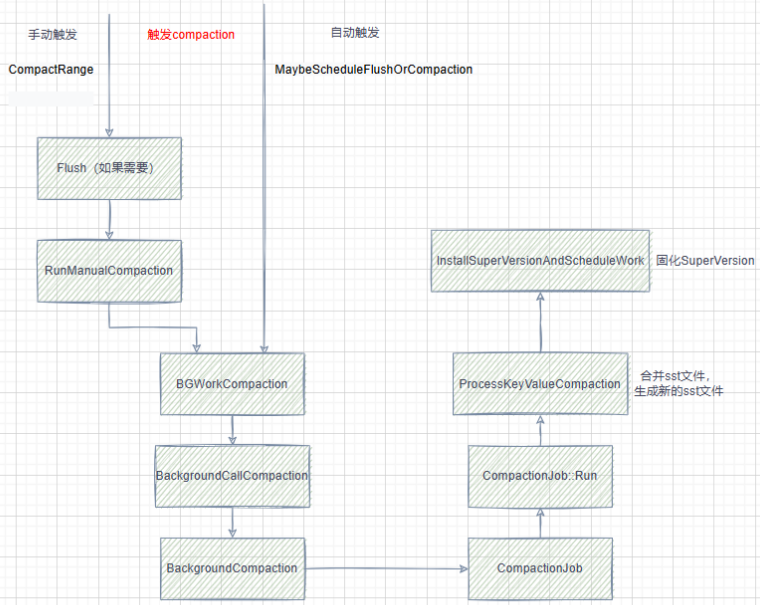
默认compaction策略：Leveled Compaction



compaction结束后：



触发接口，手动触发：CompactRange，自动触发：MaybeScheduleFlushOrCompaction



Compaction设计数据合并、sst文件生成和删除、版本管理、sst文件选举方式、层级大小计算等

手动触发流程

接口CompactRange，通过该接口手动指定某个范围的key执行compaction，或整个db全量执行compaction。

Status DBImpl::CompactRange(const CompactRangeOptions& options,

                            ColumnFamilyHandle\* column\_family,

                            const Slice\* begin\_without\_ts,

                            const Slice\* end\_without\_ts) {

  ...

// 根据时间戳大小来确认传参

  const Comparator\* const ucmp = column\_family->GetComparator();

  size\_t ts\_sz = ucmp->timestamp\_size();

  if (ts\_sz == 0) {

    return CompactRangeInternal(options, column\_family, begin\_without\_ts,

                                end\_without\_ts, "" /\*trim\_ts\*/);

  }

  std::string begin\_str, end\_str;

  auto [begin, end] =

      MaybeAddTimestampsToRange(begin\_without\_ts, end\_without\_ts, ts\_sz,

                                &begin\_str, &end\_str, false /\*exclusive\_end\*/);

  return CompactRangeInternal(

      options, column\_family, begin.has\_value() ? &begin.value() : nullptr,

      end.has\_value() ? &end.value() : nullptr, "" /\*trim\_ts\*/);

}

Status DBImpl::CompactRangeInternal(const CompactRangeOptions& options,

                                    ColumnFamilyHandle\* column\_family,

                                    const Slice\* begin, const Slice\* end,

                                    const std::string& trim\_ts) {

// 获取cfd（在对应列族进行compaction）

  auto cfh = static\_cast\_with\_check<ColumnFamilyHandleImpl>(column\_family);

  auto cfd = cfh->cfd();

// rocksdb可以为不同层级指定不同存储介质

// 列族每一种介质的路径和大小都存在cf\_paths内

  if (options.target\_path\_id >= cfd->ioptions()->cf\_paths.size()) {

    return Status::InvalidArgument("Invalid target path ID");

  }

  bool flush\_needed = true;

...

  Status s;

// 全量compaction的begin和end都是nullptr

// 只要不是全量，就判断memtable中是否包含key范围

  if (begin != nullptr && end != nullptr) {

    UserKeyRange range(\*begin, \*end);

    SuperVersion\* super\_version = cfd->GetReferencedSuperVersion(this);

    s = cfd->RangesOverlapWithMemtables(

        {range}, super\_version, immutable\_db\_options\_.allow\_data\_in\_errors,

        &flush\_needed);

    CleanupSuperVersion(super\_version);

  }

  if (s.ok() && flush\_needed) {

// 需要flush就执行flush

    ...

    if (immutable\_db\_options\_.atomic\_flush) {

      s = AtomicFlushMemTables(fo, FlushReason::kManualCompaction);

    } else {

      s = FlushMemTable(cfd, fo, FlushReason::kManualCompaction);

}

...

  }

  constexpr int kInvalidLevel = -1;

  int final\_output\_level = kInvalidLevel;

// 是否要独占db的compaction，手动的默认独占

  bool exclusive = options.exclusive\_manual\_compaction;

  if (cfd->ioptions()->compaction\_style == kCompactionStyleUniversal &&

      cfd->NumberLevels() > 1) {

...

// kCompactionStyleUniversal类型的compaction

// 传入的input\_level为kCompactAllLevels

    s = RunManualCompaction(cfd, ColumnFamilyData::kCompactAllLevels,

                            final\_output\_level, options, begin, end, exclusive,

                            false /\* disable\_trivial\_move \*/,

                            std::numeric\_limits<uint64\_t>::max(), trim\_ts);

  } else {

    int first\_overlapped\_level = kInvalidLevel;

    {

      ...

      bool overlap;

      for (int level = 0;

           level < current\_version->storage\_info()->num\_non\_empty\_levels();

           level++) {

        overlap = true;

        bool check\_overlap\_within\_file = false;

        if (begin != nullptr && end != nullptr) {

          // Typically checking overlap within files in this case

          check\_overlap\_within\_file = true;

          if (partitioner) {

            // 这里官方有关于partitioner的一大堆注释

            if (!partitioner->CanDoTrivialMove(\*begin, \*end)) {

              check\_overlap\_within\_file = false;

            }

          }

        }

        ...

        if (!check\_overlap\_within\_file) {

          overlap = current\_version->storage\_info()->OverlapInLevel(level,

                                                                    begin, end);

        }

        if (overlap) {

// 获取与compaction范围有重叠的sst文件的最大层数

          first\_overlapped\_level = level;

          break;

        }

      }

      CleanupSuperVersion(super\_version);

    }

    if (s.ok() && first\_overlapped\_level != kInvalidLevel) {

      if (cfd->ioptions()->compaction\_style == kCompactionStyleUniversal ||

          cfd->ioptions()->compaction\_style == kCompactionStyleFIFO) {

        ...

      } else {

        assert(cfd->ioptions()->compaction\_style == kCompactionStyleLevel);

        uint64\_t next\_file\_number = versions\_->current\_next\_file\_number();

        int level = first\_overlapped\_level;

        final\_output\_level = level;

        int output\_level = 0, base\_level = 0;

        for (;;) {

          // Always allow L0 -> L1 compaction

          if (level > 0) {

            if (cfd->ioptions()->level\_compaction\_dynamic\_level\_bytes) {

              assert(final\_output\_level < cfd->ioptions()->num\_levels);

              if (final\_output\_level + 1 == cfd->ioptions()->num\_levels) {

                break;

              }

            } else {

              InstrumentedMutexLock l(&mutex\_);

              if (final\_output\_level + 1 >=

                  cfd->current()->storage\_info()->num\_non\_empty\_levels()) {

                break;

              }

            }

          }

          output\_level = level + 1;

// 是否让数据快速进入下层，层之间介质不同一般不启用（上层通常快）

          if (cfd->ioptions()->level\_compaction\_dynamic\_level\_bytes &&

              level == 0) {

            output\_level = ColumnFamilyData::kCompactToBaseLevel;

          }

          // 全量Compaction

          s = RunManualCompaction(

              cfd, level, output\_level, options, begin, end, exclusive,

              !trim\_ts.empty() /\* disallow\_trivial\_move \*/,

              std::numeric\_limits<uint64\_t>::max() /\* max\_file\_num\_to\_ignore \*/,

              trim\_ts,

              output\_level == ColumnFamilyData::kCompactToBaseLevel

                  ? &base\_level

                  : nullptr);

          if (!s.ok()) {

            break;

          }

          if (output\_level == ColumnFamilyData::kCompactToBaseLevel) {

            assert(base\_level > 0);

            level = base\_level;

          } else {

            ++level;

          }

          final\_output\_level = level;

        }

        if (s.ok()) {

          assert(final\_output\_level > 0);

          // bottommost level intra-level compaction

          if ((options.bottommost\_level\_compaction ==

                   BottommostLevelCompaction::kIfHaveCompactionFilter &&

               (cfd->ioptions()->compaction\_filter != nullptr ||

                cfd->ioptions()->compaction\_filter\_factory != nullptr)) ||

              options.bottommost\_level\_compaction ==

                  BottommostLevelCompaction::kForceOptimized ||

              options.bottommost\_level\_compaction ==

                  BottommostLevelCompaction::kForce) {

            // Use `next\_file\_number` as `max\_file\_num\_to\_ignore` to avoid

            // rewriting newly compacted files when it is kForceOptimized

            // or kIfHaveCompactionFilter with compaction filter set.

            s = RunManualCompaction(

                cfd, final\_output\_level, final\_output\_level, options, begin,

                end, exclusive, true /\* disallow\_trivial\_move \*/,

                next\_file\_number /\* max\_file\_num\_to\_ignore \*/, trim\_ts);

          }

        }

      }

    }

  }

  if (!s.ok() || final\_output\_level == kInvalidLevel) {

    LogFlush(immutable\_db\_options\_.info\_log);

    return s;

  }

  if (options.change\_level) {

    DisableManualCompaction();

    s = PauseBackgroundWork();

if (s.ok()) {

// 如果需要将某个层级数据迁移到另外一个层级，则需要暂停集群刷新和压缩

// 然后使用ReFitLevel函数迁移

      s = ReFitLevel(cfd, final\_output\_level, options.target\_level);

      // ContinueBackgroundWork always return Status::OK().

      Status temp\_s = ContinueBackgroundWork();

      assert(temp\_s.ok());

    }

    EnableManualCompaction();

    TEST\_SYNC\_POINT(

        "DBImpl::CompactRange:PostRefitLevel:ManualCompactionEnabled");

  }

// 下刷日志

  LogFlush(immutable\_db\_options\_.info\_log);

  {

    InstrumentedMutexLock l(&mutex\_);

    // an automatic compaction that has been scheduled might have been

// preempted by the manual compactions. Need to schedule it back.

// 尝试调度自动compaction

    MaybeScheduleFlushOrCompaction();

  }

  return s;

}

// 一个层级到下一个层级的compaction

Status DBImpl::RunManualCompaction(

    ColumnFamilyData\* cfd, int input\_level, int output\_level,

    const CompactRangeOptions& compact\_range\_options, const Slice\* begin,

    const Slice\* end, bool exclusive, bool disallow\_trivial\_move,

    uint64\_t max\_file\_num\_to\_ignore, const std::string& trim\_ts,

    int\* final\_output\_level) {

// 初始化手动compaction信息 ...

  // 添加手动任务，暂时禁用自动compaction，独占db

  AddManualCompaction(&manual);

  TEST\_SYNC\_POINT\_CALLBACK("DBImpl::RunManualCompaction:NotScheduled", &mutex\_);

  if (exclusive) {

    // 检查自动执行的任务数

    while (bg\_bottom\_compaction\_scheduled\_ > 0 ||

           bg\_compaction\_scheduled\_ > 0) {

      if (manual\_compaction\_paused\_ > 0 || manual.canceled == true) {

        // Pretend the error came from compaction so the below cleanup/error

        // handling code can process it.

        manual.done = true;

        manual.status =

            Status::Incomplete(Status::SubCode::kManualCompactionPaused);

        break;

      }

      TEST\_SYNC\_POINT("DBImpl::RunManualCompaction:WaitScheduled");

      ROCKS\_LOG\_INFO(

          immutable\_db\_options\_.info\_log,

          "[%s] Manual compaction waiting for all other scheduled background "

          "compactions to finish",

          cfd->GetName().c\_str());

// 等待后台comapciton执行完毕

      bg\_cv\_.Wait();

    }

  }

  LogBuffer log\_buffer(InfoLogLevel::INFO\_LEVEL,

                       immutable\_db\_options\_.info\_log.get());

  ROCKS\_LOG\_BUFFER(&log\_buffer, "[%s] Manual compaction starting",

                   cfd->GetName().c\_str());

  // 出现错误时，也会将manual.done设置为true

  while (!manual.done) {

    manual\_conflict = false;

Compaction\* compaction = nullptr;

// 如果不应该执行手动compaction（shouldnt），或任务被执行，或任务被调度，或没有任务

    if (ShouldntRunManualCompaction(&manual) || (manual.in\_progress == true) ||

        scheduled ||

        (((manual.manual\_end = &manual.tmp\_storage1) != nullptr) &&

         ((compaction = manual.cfd->CompactRange(

               \*manual.cfd->GetLatestMutableCFOptions(), mutable\_db\_options\_,

               manual.input\_level, manual.output\_level, compact\_range\_options,

               manual.begin, manual.end, &manual.manual\_end, &manual\_conflict,

               max\_file\_num\_to\_ignore, trim\_ts)) == nullptr &&

          manual\_conflict))) {

      if (!scheduled) {

        // 出现冲突compaction

        if (manual\_compaction\_paused\_ > 0 || manual.canceled == true) {

          // 如果有挂起的手动compaction或当前manual被取消，则返回未完成

          manual.done = true;

          manual.status =

              Status::Incomplete(Status::SubCode::kManualCompactionPaused);

        }

      }

      if (!manual.done) {

// 没有冲突并且未完成则等待被唤醒

        bg\_cv\_.Wait();

      }

// 如果有挂起的手动任务，当前任务被调度且没有被取消调度

      if (manual\_compaction\_paused\_ > 0 && scheduled && !unscheduled) {

        // 取消调度所有手动compaction

        auto unscheduled\_task\_num = env\_->UnSchedule(

            GetTaskTag(TaskType::kManualCompaction), thread\_pool\_priority);

        if (unscheduled\_task\_num > 0) {

          // it may unschedule other manual compactions, notify others.

          bg\_cv\_.SignalAll();

        }

        unscheduled = true;

        TEST\_SYNC\_POINT("DBImpl::RunManualCompaction:Unscheduled");

      }

// 未完成状态，等待重新调度

      if (scheduled && manual.incomplete == true) {

        assert(!manual.in\_progress);

        scheduled = false;

        manual.incomplete = false;

      }

} else if (!scheduled) {

// 直到执行manual.cfd->CompactRange后compaction才不为nullptr

// 否则一直进入if尝试唤醒

      if (compaction == nullptr) {

        manual.done = true;

        if (final\_output\_level) {

          // No compaction needed or there is a conflicting compaction.

          // Still set `final\_output\_level` to the level where we would

          // have compacted to.

          \*final\_output\_level = output\_level;

          if (output\_level == ColumnFamilyData::kCompactToBaseLevel) {

            \*final\_output\_level = cfd->current()->storage\_info()->base\_level();

          }

        }

        bg\_cv\_.SignalAll();

        continue;

      }

// CompactRange已经完成，并且选取了compactin

      ca = new CompactionArg;

      ca->db = this;

      ca->prepicked\_compaction = new PrepickedCompaction;

      ca->prepicked\_compaction->manual\_compaction\_state = &manual;

      ca->prepicked\_compaction->compaction = compaction;

// 请求token（限流机制）

      if (!RequestCompactionToken(

              cfd, true, &ca->prepicked\_compaction->task\_token, &log\_buffer)) {

        // Don't throttle manual compaction, only count outstanding tasks.

        assert(false);

      }

      manual.incomplete = false;

      if (compaction->bottommost\_level() &&

          env\_->GetBackgroundThreads(Env::Priority::BOTTOM) > 0) {

        bg\_bottom\_compaction\_scheduled\_++;

        ca->compaction\_pri\_ = Env::Priority::BOTTOM;

// 调度执行compaction

        env\_->Schedule(&DBImpl::BGWorkBottomCompaction, ca,

                       Env::Priority::BOTTOM,

                       GetTaskTag(TaskType::kManualCompaction),

                       &DBImpl::UnscheduleCompactionCallback);

        thread\_pool\_priority = Env::Priority::BOTTOM;

      } else {

        bg\_compaction\_scheduled\_++;

        ca->compaction\_pri\_ = Env::Priority::LOW;

        env\_->Schedule(&DBImpl::BGWorkCompaction, ca, Env::Priority::LOW,

                       GetTaskTag(TaskType::kManualCompaction),

                       &DBImpl::UnscheduleCompactionCallback);

        thread\_pool\_priority = Env::Priority::LOW;

      }

// 调度之后，scheduled为true，下次while循环会进入if继续阻塞

      scheduled = true;

      TEST\_SYNC\_POINT("DBImpl::RunManualCompaction:Scheduled");

      if (final\_output\_level) {

        \*final\_output\_level = compaction->output\_level();

      }

    }

  }

  log\_buffer.FlushBufferToLog();

  assert(!manual.in\_progress);

  assert(HasPendingManualCompaction());

// 从manual\_compaction\_dequeue\_队列中移除已经执行完成的任务

  RemoveManualCompaction(&manual);

  if (manual.status.IsIncomplete() &&

      manual.status.subcode() == Status::SubCode::kManualCompactionPaused) {

    MaybeScheduleFlushOrCompaction();

  }

// 尝试唤醒

  bg\_cv\_.SignalAll();

  return manual.status;

}

// 返回true代表不能执行compaction

bool DBImpl::ShouldntRunManualCompaction(ManualCompactionState\* m) {

  if (m->exclusive) {

// 启用独占db compaction情况下，后台有其他compaction任务也不能执行

    return (bg\_bottom\_compaction\_scheduled\_ > 0 ||

            bg\_compaction\_scheduled\_ > 0);

  }

  std::deque<ManualCompactionState\*>::iterator it =

      manual\_compaction\_dequeue\_.begin();

  bool seen = false;

  while (it != manual\_compaction\_dequeue\_.end()) {

    if (m == (\*it)) {

      ++it;

      seen = true;

      continue;

} else if (MCOverlap(m, (\*it)) && (!seen && !(\*it)->in\_progress)) {

// 如果多个手动compaction存在key重叠，则也不能执行

      return true;

    }

    ++it;

  }

  return false;

}

Compaction\* ColumnFamilyData::CompactRange(

    const MutableCFOptions& mutable\_cf\_options,

    const MutableDBOptions& mutable\_db\_options, int input\_level,

    int output\_level, const CompactRangeOptions& compact\_range\_options,

    const InternalKey\* begin, const InternalKey\* end,

    InternalKey\*\* compaction\_end, bool\* conflict,

uint64\_t max\_file\_num\_to\_ignore, const std::string& trim\_ts) {

// 通过compaction\_picker\_创建一个compaction

  auto\* result = compaction\_picker\_->CompactRange(

      GetName(), mutable\_cf\_options, mutable\_db\_options,

      current\_->storage\_info(), input\_level, output\_level,

      compact\_range\_options, begin, end, compaction\_end, conflict,

      max\_file\_num\_to\_ignore, trim\_ts);

  if (result != nullptr) {

    result->FinalizeInputInfo(current\_);

  }

  TEST\_SYNC\_POINT("ColumnFamilyData::CompactRange:Return");

  return result;

}

Compaction\* CompactionPicker::CompactRange(

    const std::string& cf\_name, const MutableCFOptions& mutable\_cf\_options,

    const MutableDBOptions& mutable\_db\_options, VersionStorageInfo\* vstorage,

    int input\_level, int output\_level,

    const CompactRangeOptions& compact\_range\_options, const InternalKey\* begin,

    const InternalKey\* end, InternalKey\*\* compaction\_end, bool\* manual\_conflict,

    uint64\_t max\_file\_num\_to\_ignore, const std::string& trim\_ts) {

  // CompactionPickerFIFO has its own implementation of compact range

  assert(ioptions\_.compaction\_style != kCompactionStyleFIFO);

// 在kCompactionStyleUniversal模式下进入该if

  if (input\_level == ColumnFamilyData::kCompactAllLevels) {

......

     Compaction\* c = new Compaction(......);

    RegisterCompaction(c);

    vstorage->ComputeCompactionScore(ioptions\_, mutable\_cf\_options);

    return c;

  }

  CompactionInputFiles inputs; // 选取输入文件

  inputs.level = input\_level; // 输入层级

  bool covering\_the\_whole\_range = true;

  // All files are 'overlapping' in universal style compaction.

  // We have to compact the entire range in one shot.

  if (ioptions\_.compaction\_style == kCompactionStyleUniversal) {

    begin = nullptr;

    end = nullptr;

  }

// 获取input\_level中所有与begin和end有重叠的sst文件，并翻入inputs.files

// L0采用遍历查找，L1及以上二分查找

  vstorage->GetOverlappingInputs(input\_level, begin, end, &inputs.files);

  if (inputs.empty()) {

    return nullptr;

  }

// 检测L0是否冲突，因为L0同时只能参与一个compaction

  if ((input\_level == 0) && (!level0\_compactions\_in\_progress\_.empty())) {

    // Only one level 0 compaction allowed

    TEST\_SYNC\_POINT("CompactionPicker::CompactRange:Conflict");

    \*manual\_conflict = true;

    return nullptr;

  }

  // 限流，单次compaction字节数限制max\_compaction\_bytes

  if (input\_level > 0) {

    const uint64\_t limit = mutable\_cf\_options.max\_compaction\_bytes;

    uint64\_t input\_level\_total = 0;

    int hint\_index = -1;

    InternalKey\* smallest = nullptr;

    InternalKey\* largest = nullptr;

    for (size\_t i = 0; i + 1 < inputs.size(); ++i) {

      if (!smallest) {

        smallest = &inputs[i]->smallest;

      }

      largest = &inputs[i]->largest;

      uint64\_t input\_file\_size = inputs[i]->fd.GetFileSize();

      uint64\_t output\_level\_total = 0;

      if (output\_level < vstorage->num\_non\_empty\_levels()) {

        std::vector<FileMetaData\*> files;

        vstorage->GetOverlappingInputsRangeBinarySearch(

            output\_level, smallest, largest, &files, hint\_index, &hint\_index);

        for (const auto& file : files) {

          output\_level\_total += file->fd.GetFileSize();

        }

      }

      input\_level\_total += input\_file\_size;

      if (input\_level\_total + output\_level\_total >= limit) {

        covering\_the\_whole\_range = false;

        // 如果超过limit，则对文件进行resize

        inputs.files.resize(i + 1);

        break;

      }

    }

  }

...

  InternalKey key\_storage;

  InternalKey\* next\_smallest = &key\_storage;

// 根据inputs中的文件扩展出当前level的一个清晰边界

// 根据这些sst文件的最小key和最大key进行扩展，和前面传入的begin和end有区别

  if (ExpandInputsToCleanCut(cf\_name, vstorage, &inputs, &next\_smallest) ==

      false) {

    // manual compaction is now multi-threaded, so it can

    // happen that ExpandWhileOverlapping fails

    // we handle it higher in RunManualCompaction

    \*manual\_conflict = true;

    return nullptr;

  }

  if (covering\_the\_whole\_range || !next\_smallest) {

    \*compaction\_end = nullptr;

  } else {

    \*\*compaction\_end = \*next\_smallest;

  }

  CompactionInputFiles output\_level\_inputs;

  if (output\_level == ColumnFamilyData::kCompactToBaseLevel) {

    assert(input\_level == 0);

    output\_level = vstorage->base\_level();

    assert(output\_level > 0);

  }

  output\_level\_inputs.level = output\_level;

  if (input\_level != output\_level) {

int parent\_index = -1;

// 在output\_level查找重叠的sst文件，内部逻辑和查找input\_level相似

    if (!SetupOtherInputs(cf\_name, vstorage, &inputs, &output\_level\_inputs,

                          &parent\_index, -1)) {

      // manual compaction is now multi-threaded, so it can

      // happen that SetupOtherInputs fails

      // we handle it higher in RunManualCompaction

      \*manual\_conflict = true;

      return nullptr;

    }

  }

// 将input\_level和output\_level的文件都放入compaction\_inputs

  std::vector<CompactionInputFiles> compaction\_inputs({inputs});

  if (!output\_level\_inputs.empty()) {

    compaction\_inputs.push\_back(output\_level\_inputs);

  }

  for (size\_t i = 0; i < compaction\_inputs.size(); i++) {

// 判断每个文件是否正在参与compaction

    if (AreFilesInCompaction(compaction\_inputs[i].files)) {

      \*manual\_conflict = true;

      return nullptr;

    }

  }

// 对于2个没有开启独占db的compaction，判断当前compaction的最小key和最大key

// 是否和compactions\_in\_progress\_中相同output level的key有范围重叠

// 是的话要暂停当前compaction

  if (FilesRangeOverlapWithCompaction(

          compaction\_inputs, output\_level,

          Compaction::EvaluatePenultimateLevel(vstorage, ioptions\_, input\_level,

                                               output\_level))) {

    // This compaction output could potentially conflict with the output

    // of a currently running compaction, we cannot run it.

    \*manual\_conflict = true;

    return nullptr;

  }

  std::vector<FileMetaData\*> grandparents;

// 获取input和output中所有的sst文件的最小key和最大key

// 在output level + 1级别查找重叠的sst，将其放入到grandparents

  GetGrandparents(vstorage, inputs, output\_level\_inputs, &grandparents);

// 已经计算完一个compaction需要的数据，创建一个compaction

  Compaction\* compaction = new Compaction(

      vstorage, ioptions\_, mutable\_cf\_options, mutable\_db\_options,

      std::move(compaction\_inputs), output\_level,

      MaxFileSizeForLevel(mutable\_cf\_options, output\_level,

                          ioptions\_.compaction\_style, vstorage->base\_level(),

                          ioptions\_.level\_compaction\_dynamic\_level\_bytes),

      mutable\_cf\_options.max\_compaction\_bytes,

      compact\_range\_options.target\_path\_id,

      GetCompressionType(vstorage, mutable\_cf\_options, output\_level,

                         vstorage->base\_level()),

      GetCompressionOptions(mutable\_cf\_options, vstorage, output\_level),

      Temperature::kUnknown, compact\_range\_options.max\_subcompactions,

      std::move(grandparents), /\* is manual \*/ true, trim\_ts, /\* score \*/ -1,

      /\* deletion\_compaction \*/ false, /\* l0\_files\_might\_overlap \*/ true,

      CompactionReason::kUnknown,

      compact\_range\_options.blob\_garbage\_collection\_policy,

      compact\_range\_options.blob\_garbage\_collection\_age\_cutoff);

  TEST\_SYNC\_POINT\_CALLBACK("CompactionPicker::CompactRange:Return", compaction);

// 注册到compactions\_in\_progress\_

  RegisterCompaction(compaction);

  // Creating a compaction influences the compaction score because the score

  // takes running compactions into account (by skipping files that are already

  // being compacted). Since we just changed compaction score, we recalculate it

  // here

// 重新计算每一层的score

  vstorage->ComputeCompactionScore(ioptions\_, mutable\_cf\_options);

  return compaction;

}

自动触发compaction

rocksdb内部多处会调用MaybeScheduleFlushOrCompaction，以自动触发Compaction

void DBImpl::MaybeScheduleFlushOrCompaction() {

  ...

// 如果有可以调度的compaction，并且总任务数小于上限，则用LOW线程执行compaction

  while (bg\_compaction\_scheduled\_ + bg\_bottom\_compaction\_scheduled\_ <

             bg\_job\_limits.max\_compactions &&

         unscheduled\_compactions\_ > 0) {

    CompactionArg\* ca = new CompactionArg;

    ca->db = this;

    ca->compaction\_pri\_ = Env::Priority::LOW;

    ca->prepicked\_compaction = nullptr;

    bg\_compaction\_scheduled\_++;

    unscheduled\_compactions\_--;

    env\_->Schedule(&DBImpl::BGWorkCompaction, ca, Env::Priority::LOW, this,

                   &DBImpl::UnscheduleCompactionCallback);

  }

}

rocksdb内部根据各个列族每一层的score值来决定是否需要执行compaction和执行的顺序（score大于1代表需要执行compaction，score越大越先执行）

每个列族的sst呈现LSM结构，每层都会有一个大小。每层容纳数据和score的计算方式：

max\_bytes\_for\_level\_base：L0和L1层能承载的字节数

max\_bytes\_for\_level\_multiplier：倍数

level\_compaction\_dynamic\_level\_bytes：开启该开关后，数据优先落入到最下层，空间放大最小（同一存储介质下才会打开）

max\_bytes\_for\_level\_multiplier\_additional：单独对某一层新增倍数

if (level\_compaction\_dynamic\_level\_bytes):

L(n+1) = (Ln \* max\_bytes\_for\_level\_multiplier);

else

L(n+1) = (Ln \* max\_bytes\_for\_level\_multiplier) \*

max\_bytes\_for\_level\_multiplier\_additional[n];

例：关闭优先落入下层、单独倍数时，

max\_bytes\_for\_level\_base=1024\*1024\*1024;

max\_bytes\_for\_level\_multiplier=10

L0=1G, L1=1G, L2=10G, L3=100G, L4=1000G...

score的计算方式为：

Score = 当前层级的数据大小 / 当前层级理论可以承载的数据大小

L0的score计算比较特殊，新增了sst文件个数计算score，然后取两种方式的最大值：

level0\_file\_num\_compaction\_trigger：触发L0进行compaction的配置阈值

score = 当前L0未参与compaction的sst文件个数 / level0\_file\_num\_compaction\_trigger

score = max(文件个数计算的score, L0数据量计算的score)

Score计算代码：

void VersionStorageInfo::ComputeCompactionScore(

    const ImmutableOptions& immutable\_options,

    const MutableCFOptions& mutable\_cf\_options) {

  double total\_downcompact\_bytes = 0.0;

  // 目前数据量计算阈值仍为1.0，但优先级由更多因素决定

  // 现在当比值为1.0以上时会乘以10，用来根据其他因素调整优先级

  const double kScoreScale = 10.0;

  int max\_output\_level = MaxOutputLevel(immutable\_options.allow\_ingest\_behind);

  for (int level = 0; level <= MaxInputLevel(); level++) {

    double score;

    if (level == 0) {

      // 目前L0用文件数score代替了数据量score

      int num\_sorted\_runs = 0;

      uint64\_t total\_size = 0;

      for (auto\* f : files\_[level]) {

// 统计文件总大小

        total\_downcompact\_bytes += static\_cast<double>(f->fd.GetFileSize());

        if (!f->being\_compacted) {

// compensated\_file\_size是一个估算的值，把无效key也算进去了

          total\_size += f->compensated\_file\_size;

          num\_sorted\_runs++;

        }

      }

      if (compaction\_style\_ == kCompactionStyleUniversal) {

        ...

      }

      if (compaction\_style\_ == kCompactionStyleFIFO) {

        ...

      } else {

// score=L0未参与compaction的文件总个数 / 配置的文件数阈值

        score = static\_cast<double>(num\_sorted\_runs) /

                mutable\_cf\_options.level0\_file\_num\_compaction\_trigger;

        if (compaction\_style\_ == kCompactionStyleLevel && num\_levels() > 1) {

          // 层次化压缩，可能进行L0->L0的压缩，L0文件过大会导致迁移到LBase开支过大

// L0score过高会让L0->LBase始终优先LBase->LBase+1，导致LBase堆积数据

// L0score过低会在L0积累过多文件

          if (immutable\_options.level\_compaction\_dynamic\_level\_bytes) {

// 启用动态层级大小判断（数据直接落到最底层）

            if (total\_size >= mutable\_cf\_options.max\_bytes\_for\_level\_base) {

              // 数据量超过阈值时，保证score超过阈值

              score = std::max(score, 1.01);

            }

            if (total\_size > level\_max\_bytes\_[base\_level\_]) {

              // 当数据量超过基础层时，更新score为L0数据量/LBase数据量（取最大）

// 确保L0压缩优先级更高

              uint64\_t base\_level\_size = 0;

              for (auto f : files\_[base\_level\_]) {

                base\_level\_size += f->compensated\_file\_size;

              }

              score = std::max(score, static\_cast<double>(total\_size) /

                                          static\_cast<double>(std::max(

                                              base\_level\_size,

                                              level\_max\_bytes\_[base\_level\_])));

            }

            if (score > 1.0) {

// score超过1.0，则默认放大10倍

              score \*= kScoreScale;

            }

          } else {

// 未启用动态层级大小调整，则对比数据量score和文件数score

            score = std::max(score,

                             static\_cast<double>(total\_size) /

                                 mutable\_cf\_options.max\_bytes\_for\_level\_base);

          }

        }

      }

    } else {  // level > 0

      // Compute the ratio of current size to size limit.

      uint64\_t level\_bytes\_no\_compacting = 0;

      uint64\_t level\_total\_bytes = 0;

// 遍历所有文件，计算数据量

      for (auto f : files\_[level]) {

        level\_total\_bytes += f->fd.GetFileSize();

        if (!f->being\_compacted) {

          level\_bytes\_no\_compacting += f->compensated\_file\_size;

        }

      }

      if (!immutable\_options.level\_compaction\_dynamic\_level\_bytes) {

// 没启用动态层级大小调整，则直接计算数据量score

        score = static\_cast<double>(level\_bytes\_no\_compacting) /

                MaxBytesForLevel(level);

      } else {

// 启用了，就判断数据量是否超过当前层级上限

        if (level\_bytes\_no\_compacting < MaxBytesForLevel(level)) {

// 没超过当前层级上限，就正常计算

          score = static\_cast<double>(level\_bytes\_no\_compacting) /

                  MaxBytesForLevel(level);

        } else {

          // 即将接收大量数据，降低优先级

          score = static\_cast<double>(level\_bytes\_no\_compacting) /

                  (MaxBytesForLevel(level) + total\_downcompact\_bytes) \*

                  kScoreScale;

        }

        // 非必要层，通过调整，使其略低于L0的压缩需求

        if (level\_bytes\_no\_compacting > 0 &&

            level <= lowest\_unnecessary\_level\_) {

          score = std::max(

              score, kScoreScale \*

                         (1.001 + 0.001 \* (lowest\_unnecessary\_level\_ - level)));

        }

      }

      if (level <= lowest\_unnecessary\_level\_) {

// 小于或低于不必要的最低层级，则即将被压缩到下一层，用在动态层级大小调整，降低优先级

        total\_downcompact\_bytes += level\_total\_bytes;

      } else if (level\_total\_bytes > MaxBytesForLevel(level)) {

        total\_downcompact\_bytes +=

            static\_cast<double>(level\_total\_bytes - MaxBytesForLevel(level));

      }

}

// 将每一层的score存放在compaction\_score\_

    compaction\_level\_[level] = level;

    compaction\_score\_[level] = score;

  }

  // 将所有score从大到小排序

  for (int i = 0; i < num\_levels() - 2; i++) {

    for (int j = i + 1; j < num\_levels() - 1; j++) {

      if (compaction\_score\_[i] < compaction\_score\_[j]) {

        double score = compaction\_score\_[i];

        int level = compaction\_level\_[i];

        compaction\_score\_[i] = compaction\_score\_[j];

        compaction\_level\_[i] = compaction\_level\_[j];

        compaction\_score\_[j] = score;

        compaction\_level\_[j] = level;

      }

    }

  }

// 对于一些特殊条件需要compaction的sst文件，将其放入files\_marked\_for\_compaction\_

  ComputeFilesMarkedForCompaction(max\_output\_level);

// 对于最底层需要进行压缩的sst文件，将其放入bottommost\_files\_marked\_for\_compaction\_

  ComputeBottommostFilesMarkedForCompaction(

      immutable\_options.allow\_ingest\_behind);

// 生命周期过期的sst文件，将其放入到expired\_ttl\_files\_

  ComputeExpiredTtlFiles(immutable\_options, mutable\_cf\_options.ttl);

// 放入files\_marked\_for\_periodic\_compaction\_

  ComputeFilesMarkedForPeriodicCompaction(

      immutable\_options, mutable\_cf\_options.periodic\_compaction\_seconds,

      max\_output\_level);

// 标记强制垃圾回收的，放进files\_marked\_for\_forced\_blob\_gc\_

  ComputeFilesMarkedForForcedBlobGC(

      mutable\_c f\_options.blob\_garbage\_collection\_age\_cutoff,

      mutable\_cf\_options.blob\_garbage\_collection\_force\_threshold,

      mutable\_cf\_options.enable\_blob\_garbage\_collection);

// 预估需要参与compaction的数据大小

  EstimateCompactionBytesNeeded(mutable\_cf\_options);

}

待compaction文件排序

// 每次compaction都需要选取sst文件，这些需要compaction的sst文件会

// 按照compaction\_pri排序，然后放入到files\_by\_compaction\_pri\_内

// PickFileToCompact时会从中获取sst文件

void VersionStorageInfo::UpdateFilesByCompactionPri(

    const ImmutableOptions& ioptions, const MutableCFOptions& options) {

  if (compaction\_style\_ == kCompactionStyleNone ||

      compaction\_style\_ == kCompactionStyleFIFO ||

      compaction\_style\_ == kCompactionStyleUniversal) {

    // 这些类型不需要Update

    return;

  }

  // No need to sort the highest level because it is never compacted.

  for (int level = 0; level < num\_levels() - 1; level++) {

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

    auto& files\_by\_compaction\_pri = files\_by\_compaction\_pri\_[level];

    assert(files\_by\_compaction\_pri.size() == 0);

    // populate a temp vector for sorting based on size

    std::vector<Fsize> temp(files.size());

    for (size\_t i = 0; i < files.size(); i++) {

      temp[i].index = i;

      temp[i].file = files[i];

}

    // sort the top number\_of\_files\_to\_sort\_ based on file size

    size\_t num = VersionStorageInfo::kNumberFilesToSort;

    if (num > temp.size()) {

      num = temp.size();

}

// 按照compaction\_pri排序sst文件

switch (ioptions.compaction\_pri) {

// delete key优先，无效key越多越往前

      case kByCompensatedSize:

        std::partial\_sort(temp.begin(), temp.begin() + num, temp.end(),

                          CompareCompensatedSizeDescending);

        break;

// largest\_seqno优先

      case kOldestLargestSeqFirst:

        std::sort(temp.begin(), temp.end(),

                  [](const Fsize& f1, const Fsize& f2) -> bool {

                    return f1.file->fd.largest\_seqno <

                           f2.file->fd.largest\_seqno;

                  });

        break;

// smallest\_seqno优先

      case kOldestSmallestSeqFirst:

        std::sort(temp.begin(), temp.end(),

                  [](const Fsize& f1, const Fsize& f2) -> bool {

                    return f1.file->fd.smallest\_seqno <

                           f2.file->fd.smallest\_seqno;

                  });

        break;

// 与下一层sst文件重叠最小的优先（减少写放大？），默认策略

      case kMinOverlappingRatio:

        SortFileByOverlappingRatio(\*internal\_comparator\_, files\_[level],

                                   files\_[level + 1], ioptions.clock, level,

                                   num\_non\_empty\_levels\_, options.ttl, &temp);

        break;

// 轮询排序，在多个文件之间均匀分配读写操作

      case kRoundRobin:

        SortFileByRoundRobin(\*internal\_comparator\_, &compact\_cursor\_,

                             level0\_non\_overlapping\_, level, &temp);

        break;

      default:

        assert(false);

    }

assert(temp.size() == files.size());

    // initialize files\_by\_compaction\_pri\_

    for (size\_t i = 0; i < temp.size(); i++) {

      files\_by\_compaction\_pri.push\_back(static\_cast<int>(temp[i].index));

    }

    next\_file\_to\_compact\_by\_size\_[level] = 0;

    assert(files\_[level].size() == files\_by\_compaction\_pri\_[level].size());

  }

}

Compaction执行流程，自动和手动的compaction都会进入DBImpl::BGWorkCompaction入口，进行compaction流程。

void DBImpl::BGWorkCompaction(void\* arg) {

  CompactionArg ca = \*(static\_cast<CompactionArg\*>(arg));

// 调用BGWorkCompation前会new一个CompactionArg，由本函数释放内存

  delete static\_cast<CompactionArg\*>(arg);

  IOSTATS\_SET\_THREAD\_POOL\_ID(Env::Priority::LOW);

  TEST\_SYNC\_POINT("DBImpl::BGWorkCompaction");

// 手动触发的compaction，prepicked\_compaction已经实例化，自动触发的还没实例化

  auto prepicked\_compaction =

      static\_cast<PrepickedCompaction\*>(ca.prepicked\_compaction);

  static\_cast\_with\_check<DBImpl>(ca.db)->BackgroundCallCompaction(

      prepicked\_compaction, Env::Priority::LOW);

  delete prepicked\_compaction;

}

void DBImpl::BackgroundCallCompaction(PrepickedCompaction\* prepicked\_compaction,

                                      Env::Priority bg\_thread\_pri) {

  bool made\_progress = false;

  JobContext job\_context(next\_job\_id\_.fetch\_add(1), true);

  TEST\_SYNC\_POINT("BackgroundCallCompaction:0");

  LogBuffer log\_buffer(InfoLogLevel::INFO\_LEVEL,

                       immutable\_db\_options\_.info\_log.get());

  {

    InstrumentedMutexLock l(&mutex\_);

num\_running\_compactions\_++;

// 记录参与compaction的最后一个文件号，完成后移除

    std::unique\_ptr<std::list<uint64\_t>::iterator>

        pending\_outputs\_inserted\_elem(new std::list<uint64\_t>::iterator(

            CaptureCurrentFileNumberInPendingOutputs()));

    assert((bg\_thread\_pri == Env::Priority::BOTTOM &&

            bg\_bottom\_compaction\_scheduled\_) ||

           (bg\_thread\_pri == Env::Priority::LOW && bg\_compaction\_scheduled\_));

// 进入后台compaction

    Status s = BackgroundCompaction(&made\_progress, &job\_context, &log\_buffer,

                                    prepicked\_compaction, bg\_thread\_pri);

    TEST\_SYNC\_POINT("BackgroundCallCompaction:1");

// 失败处理 ...

// 移除刚记录的文件号

ReleaseFileNumberFromPendingOutputs(pending\_outputs\_inserted\_elem);

    // 如果compaction失败，扫描出需要删除的文件

    FindObsoleteFiles(&job\_context, !s.ok() && !s.IsShutdownInProgress() &&

                                        !s.IsManualCompactionPaused() &&

                                        !s.IsColumnFamilyDropped() &&

                                        !s.IsBusy());

    if (job\_context.HaveSomethingToClean() ||

        job\_context.HaveSomethingToDelete() || !log\_buffer.IsEmpty()) {

      mutex\_.Unlock();

      log\_buffer.FlushBufferToLog();

      if (job\_context.HaveSomethingToDelete()) {

// 执行删除操作

        PurgeObsoleteFiles(job\_context);

        TEST\_SYNC\_POINT("DBImpl::BackgroundCallCompaction:PurgedObsoleteFiles");

      }

      job\_context.Clean();

      mutex\_.Lock();

}

...

    // See if there's more work to be done

MaybeScheduleFlushOrCompaction();

...

  }

}

// 总体是在组装compaction任务

Status DBImpl::BackgroundCompaction(bool\* made\_progress,

                                    JobContext\* job\_context,

                                    LogBuffer\* log\_buffer,

                                    PrepickedCompaction\* prepicked\_compaction,

                                    Env::Priority thread\_pri) {

// 根据prepicked\_compaction判断是否为手动触发

  ManualCompactionState\* manual\_compaction =

      prepicked\_compaction == nullptr

          ? nullptr

          : prepicked\_compaction->manual\_compaction\_state;

  \*made\_progress = false;

  mutex\_.AssertHeld();

  TEST\_SYNC\_POINT("DBImpl::BackgroundCompaction:Start");

  const ReadOptions read\_options(Env::IOActivity::kCompaction);

  const WriteOptions write\_options(Env::IOActivity::kCompaction);

  bool is\_manual = (manual\_compaction != nullptr);

  std::unique\_ptr<Compaction> c;

  if (prepicked\_compaction != nullptr &&

      prepicked\_compaction->compaction != nullptr) {

// 手动触发，则直接将组装好的compaction交给c

    c.reset(prepicked\_compaction->compaction);

  }

  bool is\_prepicked = is\_manual || c;

  // (manual\_compaction->in\_progress == false);

// 是否禁用move机制（从一个level直接将sst移动到下一个level，不产生新sst）

  bool trivial\_move\_disallowed =

      is\_manual && manual\_compaction->disallow\_trivial\_move;

  CompactionJobStats compaction\_job\_stats;

  Status status;

...

  if (is\_manual) {

    // another thread cannot pick up the same work

    manual\_compaction->in\_progress = true;

  }

  std::unique\_ptr<TaskLimiterToken> task\_token;

  bool sfm\_reserved\_compact\_space = false;

  if (is\_manual) {

    ManualCompactionState\* m = manual\_compaction;

    if (!c) {

      m->done = true;

      m->manual\_end = nullptr;

...

    } else {

      // 检查有没有空间进行compaction

      bool enough\_room = EnoughRoomForCompaction(

          m->cfd, \*(c->inputs()), &sfm\_reserved\_compact\_space, log\_buffer);

      if (!enough\_room) {

        // Then don't do the compaction

        c->ReleaseCompactionFiles(status);

        c.reset();

        status = Status::CompactionTooLarge();

      } else {

        ...打日志

      }

    }

  } else if (!is\_prepicked && !compaction\_queue\_.empty()) {

// 自动的

    if (HasExclusiveManualCompaction()) {

      // 自动compaction，但有手动compaction开启exclusive

// 因此自动的就暂停，稍后执行

      unscheduled\_compactions\_++;

      return Status::OK();

}

// 从compaction\_queue\_队列取一个需要compaction的列族

    auto cfd = PickCompactionFromQueue(&task\_token, log\_buffer);

    if (cfd == nullptr) {

      ++unscheduled\_compactions\_;

      return Status::Busy();

}

    // 通过引用计算判断这个列族是否还在使用，没有使用就不进行compaction

    if (cfd->UnrefAndTryDelete()) {

      return Status::OK();

    }

// 获取当前列族最新的配置，位于mutable的配置都是可以在线修改的

// 所以每次都去获取该列族最新的配置

    auto\* mutable\_cf\_options = cfd->GetLatestMutableCFOptions();

    if (!mutable\_cf\_options->disable\_auto\_compactions && !cfd->IsDropped()) {

      // 从列族生成一个compaction然后交给c

      c.reset(cfd->PickCompaction(\*mutable\_cf\_options, mutable\_db\_options\_,

                                  log\_buffer));

      if (c != nullptr) {

        bool enough\_room = EnoughRoomForCompaction(

            cfd, \*(c->inputs()), &sfm\_reserved\_compact\_space, log\_buffer);

// 检查空间是否充足

        if (!enough\_room) {

          // Then don't do the compaction

          ...

// 将cfd放回队列

          AddToCompactionQueue(cfd);

          ++unscheduled\_compactions\_;

          c.reset();

          status = Status::CompactionTooLarge();

        } else {

          // update statistics

          size\_t num\_files = 0;

          for (auto& each\_level : \*c->inputs()) {

            num\_files += each\_level.files.size();

          }

          RecordInHistogram(stats\_, NUM\_FILES\_IN\_SINGLE\_COMPACTION, num\_files);

          // 从选取的列族中，判断是否还有新的需要执行compaction

// 如果需要，就尝试触发一个新的调度来执行

          if (cfd->NeedsCompaction()) {

            // Yes, we need more compactions!

            AddToCompactionQueue(cfd);

            ++unscheduled\_compactions\_;

            MaybeScheduleFlushOrCompaction();

          }

        }

      }

    }

  }

  IOStatus io\_s;

  bool compaction\_released = false;

  if (!c) {

    // Nothing to do

    ROCKS\_LOG\_BUFFER(log\_buffer, "Compaction nothing to do");

  } else if (c->deletion\_compaction()) {

    // 如果compaction仅仅是删除文件，就可以直接删除（FIFO模式下会使用，Level模式没用）

...

// 从version中删除sst文件

    for (const auto& f : \*c->inputs(0)) {

      c->edit()->DeleteFile(c->level(), f->fd.GetNumber());

}

// 落盘version到manifest文件

    status = versions\_->LogAndApply(

        c->column\_family\_data(), \*c->mutable\_cf\_options(), read\_options,

        write\_options, c->edit(), &mutex\_, directories\_.GetDbDir(),

        /\*new\_descriptor\_log=\*/false, /\*column\_family\_options=\*/nullptr,

        [&c, &compaction\_released](const Status& s) {

          c->ReleaseCompactionFiles(s);

          compaction\_released = true;

        });

io\_s = versions\_->io\_status();

// 更新Superversion

    InstallSuperVersionAndScheduleWork(c->column\_family\_data(),

                                       &job\_context->superversion\_contexts[0],

                                       \*c->mutable\_cf\_options());

    ...

  } else if (!trivial\_move\_disallowed && c->IsTrivialMove()) {

    // 如果input的文件和下一层没有重叠，则可以直接将sst move到下一层

    ThreadStatusUtil::SetColumnFamily(c->column\_family\_data());

ThreadStatusUtil::SetThreadOperation(ThreadStatus::OP\_COMPACTION);

compaction\_job\_stats.num\_input\_files = c->num\_input\_files(0);

    NotifyOnCompactionBegin(c->column\_family\_data(), c.get(), status,

                            compaction\_job\_stats, job\_context->job\_id);

    // Move files to next level

    int32\_t moved\_files = 0;

    int64\_t moved\_bytes = 0;

    for (unsigned int l = 0; l < c->num\_input\_levels(); l++) {

      if (c->level(l) == c->output\_level()) {

        continue;

      }

      for (size\_t i = 0; i < c->num\_input\_files(l); i++) {

        FileMetaData\* f = c->input(l, i);

        c->edit()->DeleteFile(c->level(l), f->fd.GetNumber());

        c->edit()->AddFile(一大堆参数);

        ++moved\_files;

        moved\_bytes += f->fd.GetFileSize();

      }

}

// 如果是因为当前层超过最大大小，则进行处理

    if (c->compaction\_reason() == CompactionReason::kLevelMaxLevelSize &&

        c->immutable\_options()->compaction\_pri == kRoundRobin) {

      ...

}

// 更新version

    status = versions\_->LogAndApply(

        c->column\_family\_data(), \*c->mutable\_cf\_options(), read\_options,

        write\_options, c->edit(), &mutex\_, directories\_.GetDbDir(),

        /\*new\_descriptor\_log=\*/false, /\*column\_family\_options=\*/nullptr,

        [&c, &compaction\_released](const Status& s) {

          c->ReleaseCompactionFiles(s);

          compaction\_released = true;

        });

    io\_s = versions\_->io\_status();

    // Use latest MutableCFOptions

    InstallSuperVersionAndScheduleWork(c->column\_family\_data(),

                                       &job\_context->superversion\_contexts[0],

                                       \*c->mutable\_cf\_options());

    ...

  } else if (!is\_prepicked && c->output\_level() > 0 &&

             c->output\_level() ==

                 c->column\_family\_data()

                     ->current()

                     ->storage\_info()

                     ->MaxOutputLevel(

                         immutable\_db\_options\_.allow\_ingest\_behind) &&

             env\_->GetBackgroundThreads(Env::Priority::BOTTOM) > 0) {

    // 这里是最底层bottom compaction（level compaction一般不需要）

    ...

    env\_->Schedule(&DBImpl::BGWorkBottomCompaction, ca, Env::Priority::BOTTOM,

                   this, &DBImpl::UnscheduleCompactionCallback);

  } else {

// compaction进入最频繁的地方，在这里创建compaction\_job

...

// 先创建一个compaction job

    CompactionJob compaction\_job(

        job\_context->job\_id, c.get(), immutable\_db\_options\_,

        mutable\_db\_options\_, file\_options\_for\_compaction\_, versions\_.get(),

        &shutting\_down\_, log\_buffer, directories\_.GetDbDir(),

        GetDataDir(c->column\_family\_data(), c->output\_path\_id()),

        GetDataDir(c->column\_family\_data(), 0), stats\_, &mutex\_,

        &error\_handler\_, snapshot\_seqs, earliest\_write\_conflict\_snapshot,

        snapshot\_checker, job\_context, table\_cache\_, &event\_logger\_,

        c->mutable\_cf\_options()->paranoid\_file\_checks,

        c->mutable\_cf\_options()->report\_bg\_io\_stats, dbname\_,

        &compaction\_job\_stats, thread\_pri, io\_tracer\_,

        is\_manual ? manual\_compaction->canceled

                  : kManualCompactionCanceledFalse\_,

        db\_id\_, db\_session\_id\_, c->column\_family\_data()->GetFullHistoryTsLow(),

        c->trim\_ts(), &blob\_callback\_, &bg\_compaction\_scheduled\_,

        &bg\_bottom\_compaction\_scheduled\_);

// 主要判断是否需要将compaction分为多个sub\_compaction来并发执行，默认关闭

compaction\_job.Prepare();

    NotifyOnCompactionBegin(c->column\_family\_data(), c.get(), status,

                            compaction\_job\_stats, job\_context->job\_id);

    mutex\_.Unlock();

    TEST\_SYNC\_POINT\_CALLBACK(

        "DBImpl::BackgroundCompaction:NonTrivial:BeforeRun", nullptr);

    // 正式执行compaction

    compaction\_job.Run().PermitUncheckedError();

    TEST\_SYNC\_POINT("DBImpl::BackgroundCompaction:NonTrivial:AfterRun");

mutex\_.Lock();

// 固化compaction的执行结果，更新统计信息

// 包括把input的sst文件从version删除，然后新生成sst文件加入到version

    status =

        compaction\_job.Install(\*c->mutable\_cf\_options(), &compaction\_released);

    io\_s = compaction\_job.io\_status();

if (status.ok()) {

// 固化superversion

      InstallSuperVersionAndScheduleWork(c->column\_family\_data(),

                                         &job\_context->superversion\_contexts[0],

                                         \*c->mutable\_cf\_options());

    }

    \*made\_progress = true;

    TEST\_SYNC\_POINT\_CALLBACK("DBImpl::BackgroundCompaction:AfterCompaction",

                             c->column\_family\_data());

  }

  if (status.ok() && !io\_s.ok()) {

    status = io\_s;

  } else {

    io\_s.PermitUncheckedError();

  }

  if (c != nullptr) {

if (!compaction\_released) {

// 释放当前

      c->ReleaseCompactionFiles(status);

    } else {

#ifndef NDEBUG

      ...

#endif

}

\*made\_progress = true;

    // Need to make sure SstFileManager does its bookkeeping

    auto sfm = static\_cast<SstFileManagerImpl\*>(

        immutable\_db\_options\_.sst\_file\_manager.get());

    if (sfm && sfm\_reserved\_compact\_space) {

      sfm->OnCompactionCompletion(c.get());

}

    NotifyOnCompactionCompleted(c->column\_family\_data(), c.get(), status,

                                compaction\_job\_stats, job\_context->job\_id);

  }

  ...

  if (is\_manual) {

    ...

  }

  TEST\_SYNC\_POINT("DBImpl::BackgroundCompaction:Finish");

  return status;

}

Compaction\* ColumnFamilyData::PickCompaction(

    const MutableCFOptions& mutable\_options,

const MutableDBOptions& mutable\_db\_options, LogBuffer\* log\_buffer) {

// 通过compaction\_picker\_生成一个compaction

  auto\* result = compaction\_picker\_->PickCompaction(

      GetName(), mutable\_options, mutable\_db\_options, current\_->storage\_info(),

      log\_buffer);

  if (result != nullptr) {

    result->FinalizeInputInfo(current\_);

  }

  return result;

}

Compaction\* LevelCompactionPicker::PickCompaction(

    const std::string& cf\_name, const MutableCFOptions& mutable\_cf\_options,

    const MutableDBOptions& mutable\_db\_options, VersionStorageInfo\* vstorage,

    LogBuffer\* log\_buffer) {

  LevelCompactionBuilder builder(cf\_name, vstorage, this, log\_buffer,

                                 mutable\_cf\_options, ioptions\_,

                                 mutable\_db\_options);

  return builder.PickCompaction();

}

}

Compaction\* LevelCompactionBuilder::PickCompaction() {

  // 根据compaction\_score\_和files\_by\_compaction\_pri\_选取sst文件

// 先选score最大的，然后在目标层级选一个sst文件，然后扩展边界（CleanCut）

  SetupInitialFiles();

  if (start\_level\_inputs\_.empty()) {

    return nullptr;

  }

  assert(start\_level\_ >= 0 && output\_level\_ >= 0);

  // 如果是L0，需要把L0的所有sst文件带上

  if (!SetupOtherL0FilesIfNeeded()) {

    return nullptr;

  }

  // 在output\_level选取和input重合的sst文件，需要一起compaction，同时扩展边界

  if (!SetupOtherInputsIfNeeded()) {

    return nullptr;

  }

  // 根据上面的一些结果生成compaction，同时注册到compactions\_in\_progress\_，更新score

  Compaction\* c = GetCompaction();

  TEST\_SYNC\_POINT\_CALLBACK("LevelCompactionPicker::PickCompaction:Return", c);

  return c;

}

// 选取一个compaction后，是否需要继续compaction

bool ColumnFamilyData::NeedsCompaction() const {

  return !mutable\_cf\_options\_.disable\_auto\_compactions &&

         compaction\_picker\_->NeedsCompaction(current\_->storage\_info());

}

bool LevelCompactionPicker::NeedsCompaction(

const VersionStorageInfo\* vstorage) const {‘’

// 是否有过期sst文件（过期sst文件放在expired\_ttl\_files\_）

  if (!vstorage->ExpiredTtlFiles().empty()) {

    return true;

  }

// 是否有定期执行compaction的文件

  if (!vstorage->FilesMarkedForPeriodicCompaction().empty()) {

    return true;

  }

// 是否有最底层sst被标记做compaction

  if (!vstorage->BottommostFilesMarkedForCompaction().empty()) {

    return true;

  }

// 是否有sst被标记做compaction

  if (!vstorage->FilesMarkedForCompaction().empty()) {

    return true;

  }

// 是否有sst被标记强制compaction

  if (!vstorage->FilesMarkedForForcedBlobGC().empty()) {

    return true;

  }

// 遍历每一层，是否还有score大于1的level

  for (int i = 0; i <= vstorage->MaxInputLevel(); i++) {

    if (vstorage->CompactionScore(i) >= 1) {

      return true;

    }

  }

  return false;

}

Status CompactionJob::Run() {

  AutoThreadOperationStageUpdater stage\_updater(

      ThreadStatus::STAGE\_COMPACTION\_RUN);

  TEST\_SYNC\_POINT("CompactionJob::Run():Start");

  log\_buffer\_->FlushBufferToLog();

  LogCompaction();

  const size\_t num\_threads = compact\_->sub\_compact\_states.size();

  assert(num\_threads > 0);

  const uint64\_t start\_micros = db\_options\_.clock->NowMicros();

  // 为前面分出的多个sub\_compaction依次分配一个线程执行ProcessKeyValueCompaction

  std::vector<port::Thread> thread\_pool;

  thread\_pool.reserve(num\_threads - 1);

  for (size\_t i = 1; i < compact\_->sub\_compact\_states.size(); i++) {

    thread\_pool.emplace\_back(&CompactionJob::ProcessKeyValueCompaction, this,

                             &compact\_->sub\_compact\_states[i]);

  }

  // 合并Key value，这里面是真正合并的过程

  ProcessKeyValueCompaction(&compact\_->sub\_compact\_states[0]);

  // 等待所有线程执行完成

  for (auto& thread : thread\_pool) {

    thread.join();

  }

// 更新统计信息和校验sst文件

  ...

  return status;

}

void CompactionJob::ProcessKeyValueCompaction(SubcompactionState\* sub\_compact) {

  assert(sub\_compact);

  assert(sub\_compact->compaction);

// 使用外部压缩算法

  if (db\_options\_.compaction\_service) {

    CompactionServiceJobStatus comp\_status =

        ProcessKeyValueCompactionWithCompactionService(sub\_compact);

    if (comp\_status == CompactionServiceJobStatus::kSuccess ||

        comp\_status == CompactionServiceJobStatus::kFailure) {

      return;

    }

    // fallback to local compaction

    assert(comp\_status == CompactionServiceJobStatus::kUseLocal);

  }

  uint64\_t prev\_cpu\_micros = db\_options\_.clock->CPUMicros();

  ColumnFamilyData\* cfd = sub\_compact->compaction->column\_family\_data();

  // 配置了compaction\_filter\_factory时生成compaction\_filter，默认没有

  const CompactionFilter\* compaction\_filter =

      cfd->ioptions()->compaction\_filter;

  std::unique\_ptr<CompactionFilter> compaction\_filter\_from\_factory = nullptr;

  if (compaction\_filter == nullptr) {

    compaction\_filter\_from\_factory =

        sub\_compact->compaction->CreateCompactionFilter();

    compaction\_filter = compaction\_filter\_from\_factory.get();

  }

  if (compaction\_filter != nullptr && !compaction\_filter->IgnoreSnapshots()) {

    sub\_compact->status = Status::NotSupported(

        "CompactionFilter::IgnoreSnapshots() = false is not supported "

        "anymore.");

    return;

  }

  NotifyOnSubcompactionBegin(sub\_compact);

// 连续删除会放在range\_del\_agg里

  auto range\_del\_agg = std::make\_unique<CompactionRangeDelAggregator>(

      &cfd->internal\_comparator(), existing\_snapshots\_, &full\_history\_ts\_low\_,

      &trim\_ts\_);

  ...

// 创建一个输入迭代器，这个迭代器能遍历所有input的sst里的key value

  std::unique\_ptr<InternalIterator> raw\_input(versions\_->MakeInputIterator(

      read\_options, sub\_compact->compaction, range\_del\_agg.get(),

      file\_options\_for\_read\_, start, end));

  InternalIterator\* input = raw\_input.get();

...

// 输入开始的位置

  if (start.has\_value()) {

    start\_ikey.SetInternalKey(\*start, kMaxSequenceNumber, kValueTypeForSeek);

    if (ts\_sz > 0) {

      start\_ikey.UpdateInternalKey(kMaxSequenceNumber, kValueTypeForSeek,

                                   &ts\_slice);

    }

    start\_slice = start\_ikey.GetInternalKey();

    start\_user\_key = start\_ikey.GetUserKey();

  }

// 输入结束的位置

  if (end.has\_value()) {

    end\_ikey.SetInternalKey(\*end, kMaxSequenceNumber, kValueTypeForSeek);

    if (ts\_sz > 0) {

      end\_ikey.UpdateInternalKey(kMaxSequenceNumber, kValueTypeForSeek,

                                 &ts\_slice);

    }

    end\_slice = end\_ikey.GetInternalKey();

    end\_user\_key = end\_ikey.GetUserKey();

  }

  std::unique\_ptr<InternalIterator> clip;

// 更新input迭代器的开始、结束位置

  if (start.has\_value() || end.has\_value()) {

    clip = std::make\_unique<ClippingIterator>(

        raw\_input.get(), start.has\_value() ? &start\_slice : nullptr,

        end.has\_value() ? &end\_slice : nullptr, &cfd->internal\_comparator());

    input = clip.get();

  }

...

// 用于支持自定义合并操作

  MergeHelper merge(

      env\_, cfd->user\_comparator(), cfd->ioptions()->merge\_operator.get(),

      compaction\_filter, db\_options\_.info\_log.get(),

      false /\* internal key corruption is expected \*/,

      existing\_snapshots\_.empty() ? 0 : existing\_snapshots\_.back(),

      snapshot\_checker\_, compact\_->compaction->level(), db\_options\_.stats);

...

// CompactionIterator封装了compaction时的处理逻辑，包括4个组件：

// InternalIterator\* input\_; 遍历输入的sst文件

// MergeHelper\* merge\_helper\_; 自定义的merge操作，默认不启用

// std::vector<SequenceNumber>\* snapshots\_; 记录db现有快照，不在里面的合并后可丢弃

// const CompactionFilter\* compaction\_filter\_; 自定义的compaction操作，默认不启用

// input\_负责输入的key value，其余三个负责合并的处理逻辑

  auto c\_iter = std::make\_unique<CompactionIterator>(

      input, cfd->user\_comparator(), &merge, versions\_->LastSequence(),

      &existing\_snapshots\_, earliest\_write\_conflict\_snapshot\_, job\_snapshot\_seq,

      snapshot\_checker\_, env\_, ShouldReportDetailedTime(env\_, stats\_),

      /\*expect\_valid\_internal\_key=\*/true, range\_del\_agg.get(),

      blob\_file\_builder.get(), db\_options\_.allow\_data\_in\_errors,

      db\_options\_.enforce\_single\_del\_contracts, manual\_compaction\_canceled\_,

      sub\_compact->compaction

          ->DoesInputReferenceBlobFiles() /\* must\_count\_input\_entries \*/,

      sub\_compact->compaction, compaction\_filter, shutting\_down\_,

      db\_options\_.info\_log, full\_history\_ts\_low, preserve\_time\_min\_seqno\_,

      preclude\_last\_level\_min\_seqno\_);

// 封装了一些合并逻辑，以在移动迭代器时就做一些合并操作

// 比如迭代器里面key的排序规则是：user\_key正序，seqnum降序、valueType降序

// 如果上一个key和当前key相同，valueType也相同，那么当前key的seq小于上个key

// 因为seq是递增的，所以这个是旧key，没有快照的话就可以直接丢弃

  c\_iter->SeekToFirst();

  // Assign range delete aggregator to the target output level, which makes sure

  // it only output to single level

  sub\_compact->AssignRangeDelAggregator(std::move(range\_del\_agg));

  const auto& c\_iter\_stats = c\_iter->iter\_stats();

  // define the open and close functions for the compaction files, which will be

  // used open/close output files when needed.

  const CompactionFileOpenFunc open\_file\_func =

      [this, sub\_compact](CompactionOutputs& outputs) {

        return this->OpenCompactionOutputFile(sub\_compact, outputs);

      };

  const CompactionFileCloseFunc close\_file\_func =

      [this, sub\_compact, start\_user\_key, end\_user\_key](

          CompactionOutputs& outputs, const Status& status,

          const Slice& next\_table\_min\_key) {

        return this->FinishCompactionOutputFile(

            status, sub\_compact, outputs, next\_table\_min\_key,

            sub\_compact->start.has\_value() ? &start\_user\_key : nullptr,

            sub\_compact->end.has\_value() ? &end\_user\_key : nullptr);

      };

  Status status;

  TEST\_SYNC\_POINT\_CALLBACK(

      "CompactionJob::ProcessKeyValueCompaction()::Processing",

      static\_cast<void\*>(const\_cast<Compaction\*>(sub\_compact->compaction)));

  uint64\_t last\_cpu\_micros = prev\_cpu\_micros;

// 开始遍历key，然后写入到输出sst文件

  while (status.ok() && !cfd->IsDropped() && c\_iter->Valid()) {

    // Invariant: c\_iter.status() is guaranteed to be OK if c\_iter->Valid()

    // returns true.

    assert(!end.has\_value() ||

           cfd->user\_comparator()->Compare(c\_iter->user\_key(), \*end) < 0);

    if (c\_iter\_stats.num\_input\_records % kRecordStatsEvery ==

        kRecordStatsEvery - 1) {

      RecordDroppedKeys(c\_iter\_stats, &sub\_compact->compaction\_job\_stats);

      c\_iter->ResetRecordCounts();

      RecordCompactionIOStats();

      uint64\_t cur\_cpu\_micros = db\_options\_.clock->CPUMicros();

      assert(cur\_cpu\_micros >= last\_cpu\_micros);

      RecordTick(stats\_, COMPACTION\_CPU\_TOTAL\_TIME,

                 cur\_cpu\_micros - last\_cpu\_micros);

      last\_cpu\_micros = cur\_cpu\_micros;

}

    // Add current compaction\_iterator key to target compaction output, if the

    // output file needs to be close or open, it will call the `open\_file\_func`

    // and `close\_file\_func`.

    // TODO: it would be better to have the compaction file open/close moved

    // into `CompactionOutputs` which has the output file information.

    status = sub\_compact->AddToOutput(\*c\_iter, open\_file\_func, close\_file\_func);

    if (!status.ok()) {

      break;

}

    c\_iter->Next();

    if (c\_iter->status().IsManualCompactionPaused()) {

      break;

    }

  }

  // 确认合并状态

  ...

  // 写入新的sst文件

  status = sub\_compact->CloseCompactionFiles(status, open\_file\_func,

                                             close\_file\_func);

  if (blob\_file\_builder) {

    if (status.ok()) {

      status = blob\_file\_builder->Finish();

    } else {

      blob\_file\_builder->Abandon(status);

    }

    blob\_file\_builder.reset();

    sub\_compact->Current().UpdateBlobStats();

  }

// 更新统计信息，重置等

}

InternalIterator\* VersionSet::MakeInputIterator(

    const ReadOptions& read\_options, const Compaction\* c,

    RangeDelAggregator\* range\_del\_agg,

    const FileOptions& file\_options\_compactions,

    const std::optional<const Slice>& start,

    const std::optional<const Slice>& end) {

  auto cfd = c->column\_family\_data();

  // 如果输入是L0，那么为每个sst创建一个迭代器，为L1创建一个迭代器

// 如果不是L0那就输入Level和输出Level各一个迭代器

  const size\_t space = (c->level() == 0 ? c->input\_levels(0)->num\_files +

                                              c->num\_input\_levels() - 1

                                        : c->num\_input\_levels());

  InternalIterator\*\* list = new InternalIterator\*[space];

  std::vector<

      std::pair<TruncatedRangeDelIterator\*, TruncatedRangeDelIterator\*\*\*>>

      range\_tombstones;

  size\_t num = 0;

// 让上面new的list数组中的每个迭代器指向具体的位置

  for (size\_t which = 0; which < c->num\_input\_levels(); which++) {

if (c->input\_levels(which)->num\_files != 0) {

// L0为每个sst生成一个迭代器

      if (c->level(which) == 0) {

        const LevelFilesBrief\* flevel = c->input\_levels(which);

        for (size\_t i = 0; i < flevel->num\_files; i++) {

          const FileMetaData& fmd = \*flevel->files[i].file\_metadata;

          if (start.has\_value() &&

              cfd->user\_comparator()->CompareWithoutTimestamp(

                  \*start, fmd.largest.user\_key()) > 0) {

            continue;

          }

          // We should be able to filter out the case where the end key

          // equals to the end boundary, since the end key is exclusive.

          // We try to be extra safe here.

          if (end.has\_value() &&

              cfd->user\_comparator()->CompareWithoutTimestamp(

                  \*end, fmd.smallest.user\_key()) < 0) {

            continue;

          }

          TruncatedRangeDelIterator\* range\_tombstone\_iter = nullptr;

          list[num++] = cfd->table\_cache()->NewIterator(

              read\_options, file\_options\_compactions,

              cfd->internal\_comparator(), fmd, range\_del\_agg,

              c->mutable\_cf\_options()->prefix\_extractor,

              /\*table\_reader\_ptr=\*/nullptr,

              /\*file\_read\_hist=\*/nullptr, TableReaderCaller::kCompaction,

              /\*arena=\*/nullptr,

              /\*skip\_filters=\*/false,

              /\*level=\*/static\_cast<int>(c->level(which)),

              MaxFileSizeForL0MetaPin(\*c->mutable\_cf\_options()),

              /\*smallest\_compaction\_key=\*/nullptr,

              /\*largest\_compaction\_key=\*/nullptr,

              /\*allow\_unprepared\_value=\*/false,

              c->mutable\_cf\_options()->block\_protection\_bytes\_per\_key,

              /\*range\_del\_read\_seqno=\*/nullptr,

              /\*range\_del\_iter=\*/&range\_tombstone\_iter);

          range\_tombstones.emplace\_back(range\_tombstone\_iter, nullptr);

        }

      } else {

// 非L0就为每层level生成一个迭代器

        // Create concatenating iterator for the files from this level

        TruncatedRangeDelIterator\*\*\* tombstone\_iter\_ptr = nullptr;

        list[num++] = new LevelIterator(

            cfd->table\_cache(), read\_options, file\_options\_compactions,

            cfd->internal\_comparator(), c->input\_levels(which),

            c->mutable\_cf\_options()->prefix\_extractor,

            /\*should\_sample=\*/false,

            /\*no per level latency histogram=\*/nullptr,

            TableReaderCaller::kCompaction, /\*skip\_filters=\*/false,

            /\*level=\*/static\_cast<int>(c->level(which)),

            c->mutable\_cf\_options()->block\_protection\_bytes\_per\_key,

            range\_del\_agg, c->boundaries(which), false, &tombstone\_iter\_ptr);

        range\_tombstones.emplace\_back(nullptr, tombstone\_iter\_ptr);

      }

    }

  }

  assert(num <= space);

// 根据迭代器集合list，生成MergeingIterator，从而遍历所有sst文件的kv

  InternalIterator\* result = NewCompactionMergingIterator(

      &c->column\_family\_data()->internal\_comparator(), list,

      static\_cast<int>(num), range\_tombstones);

  delete[] list;

  return result;

}

InternalIterator\* NewCompactionMergingIterator(

    const InternalKeyComparator\* comparator, InternalIterator\*\* children, int n,

    std::vector<std::pair<TruncatedRangeDelIterator\*,

                          TruncatedRangeDelIterator\*\*\*>>& range\_tombstone\_iters,

    Arena\* arena) {

  assert(n >= 0);

  if (n == 0) {

    return NewEmptyInternalIterator<Slice>(arena);

  } else {

    if (arena == nullptr) {

      return new CompactionMergingIterator(comparator, children, n,

                                           false /\* is\_arena\_mode \*/,

                                           range\_tombstone\_iters);

    } else {

      auto mem = arena->AllocateAligned(sizeof(CompactionMergingIterator));

      return new (mem) CompactionMergingIterator(comparator, children, n,

                                                 true /\* is\_arena\_mode \*/,

                                                 range\_tombstone\_iters);

    }

  }

}

// 管理多个子迭代器形成一个最小堆，Next调用堆顶迭代器的Next

// 然后和所有子迭代器当前key比较，最小的最为新的堆顶

CompactionMergingIterator(

      const InternalKeyComparator\* comparator, InternalIterator\*\* children,

      int n, bool is\_arena\_mode,

      std::vector<

          std::pair<TruncatedRangeDelIterator\*, TruncatedRangeDelIterator\*\*\*>>

          range\_tombstones)

      : is\_arena\_mode\_(is\_arena\_mode),

        comparator\_(comparator),

        current\_(nullptr),

        minHeap\_(CompactionHeapItemComparator(comparator\_)),

        pinned\_iters\_mgr\_(nullptr) {

    children\_.resize(n);

    for (int i = 0; i < n; i++) {

      children\_[i].level = i;

      children\_[i].iter.Set(children[i]);

      assert(children\_[i].type == HeapItem::ITERATOR);

    }

    assert(range\_tombstones.size() == static\_cast<size\_t>(n));

    for (auto& p : range\_tombstones) {

      range\_tombstone\_iters\_.push\_back(p.first);

    }

    pinned\_heap\_item\_.resize(n);

    for (int i = 0; i < n; ++i) {

      if (range\_tombstones[i].second) {

        // for LevelIterator

        \*range\_tombstones[i].second = &range\_tombstone\_iters\_[i];

      }

      pinned\_heap\_item\_[i].level = i;

      pinned\_heap\_item\_[i].type = HeapItem::DELETE\_RANGE\_START;

    }

  }