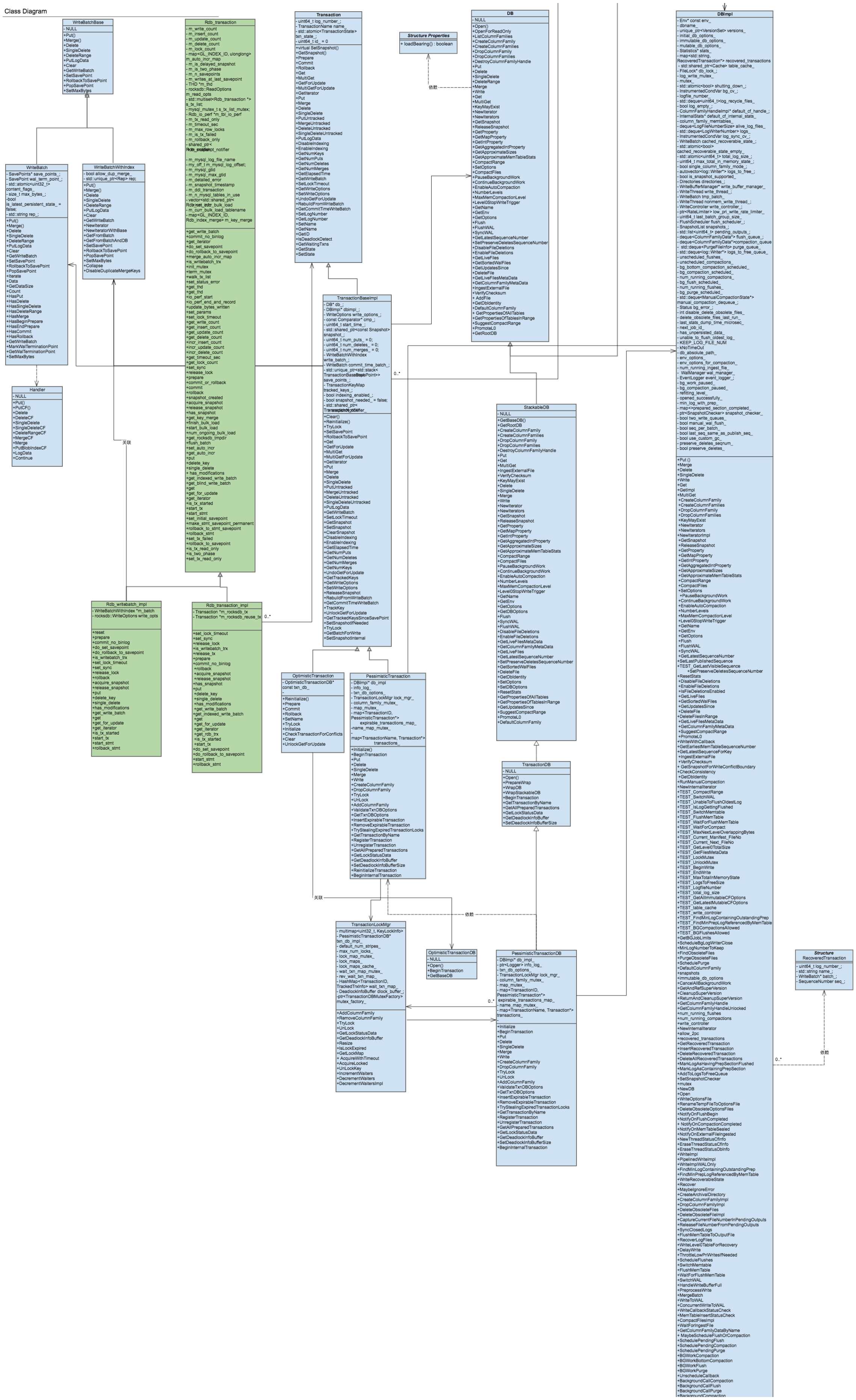
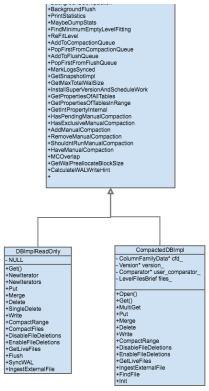
MyRocks事务模块调研

一、MyRocks事务类图

- 其中蓝色的类为rocksdb
- 绿色的类为myrocks

Class Diagram





二、事务类图关系简述

2.1 DB相关

DB−−>TransactionDB→ PessimisticTransactionDB

OptimisticTransactionDB

是主要实现LSM_Tree的主要逻辑（通过DBImpl），同时也是transaction的"容器"，一个transactionDB可以包括多个transaction。

DB→DBImpl

DBImpl是最主要的实现类，实现了最主要的LSM_tree的逻辑。另外奔溃恢复的相关逻辑，也在该类中实现。

基本用法：

```
Options options;
TransactionDBOptions txn_db_options;
options.create_if_missing = true;
TransactionDB* txn_db;

// DB(Pessimistic)
Status s = TransactionDB::Open(options, txn_db_options, kDBPath, &txn_db);
assert(s.ok());

//
Transaction* txn = txn_db->BeginTransaction(write_options);
assert(txn);

// txnkey
s = txn->Get(read_options, "abc", &value);
assert(s.IsNotFound());

// txnkey
s = txn->Put("abc", "def");
assert(s.ok());

// TransactionDB::Getkey
s = txn_db->Get(read_options, "abc", &value);

// TrasactionDB::Putkey
// "abc"
// "abc"
// PutCommit()
s = txn_db->Put(write_options, "xyz", "zzz");

s = txn->Commit();
assert(s.ok());
//
delete txn;
delete txn_db;
```

2.2 transaction相关

Transaction−−>TransactionBaseImpl−−>PessimisticTransaction

|

−−−−−−−−−−−−−−−→OptimisticTransaction

Transaction是实现事务的主要模块。通过Snapshot, lock, write_batch等模块共同完成事务的实现。

事务分为悲观事务和乐观事务，悲观事务在写之前都要获取锁，而乐观事务不会获取锁，只是在提交的时候进行冲突检查，如果写冲突则回滚。

PessimisticTransaction

PessimisticTransactionDB通过TransactionLockMgr进行行锁管理。事务中的每次写入操作之前都需要TryLock进Key锁的独占及冲突检测，以Put为例:

```
Status TransactionBaseImpl::Put(ColumnFamilyHandle* column_family,
                                const Slice& key, const Slice& value) {
  // TryLock
  Status s =
      TryLock(column_family, key, false /* read_only */, true /* exclusive
*/);

  if (s.ok()) {
    s = GetBatchForWrite()->Put(column_family, key, value);
    if (s.ok()) {
      num_puts_++;
    }
  }

  return s;
}
```

可以看到Put接口定义在TransactionBase中，无论Pessimistic还是Optimistic的Put都是这段逻辑，二者的区别是在对TryLock的重载。先看Pessimistic的，TransactionBaseImpl::TryLock通过TransactionBaseImpl::TryLock -> PessimisticTransaction::TryLock -> PessimisticTransactionDB::TryLock -> TransactionLockMgr::TryLock一路调用到TransactionLockMgr的TryLock，在里面完成对key加锁，加锁成功便实现了对key的独占，此时直到事务commit之前，其他事务是无法修改这个key的。

锁是加成功了，但这也只能说明从此刻起到事务结束前这个key不会再被外部修改，但如果事务在最开始执行SetSnapshot设置了快照，如果在打快照和Put之间的过程中外部对相同key进行了修改（并commit），此时已经打破了snapshot的保证，所以事务之后的Put也不能成功，这个冲突检测也是在PessimisticTransaction::TryLock中做的，如下：

```
Status PessimisticTransaction::TryLock(ColumnFamilyHandle* column_family,
                                       const Slice& key, bool read_only,
                                       bool exclusive, bool skip_validate)
{
  ...
  //
  if (!previously_locked || lock_upgrade) {
    s = txn_db_impl_->TryLock(this, cfh_id, key_str, exclusive);
  }

  SetSnapshotIfNeeded();

  ...

    // snapshotsequence1keyDB
    // sequence2sequence2 > sequence1snapshot
    // key
    s = ValidateSnapshot(column_family, key, &tracked_at_seq);

      if (!s.ok()) {
        //
        // Failed to validate key
        if (!previously_locked) {
          // Unlock key we just locked
          if (lock_upgrade) {
            s = txn_db_impl_->TryLock(this, cfh_id, key_str,
                                      false /* exclusive */);
            assert(s.ok());
          } else {
            txn_db_impl_->UnLock(this, cfh_id, key.ToString());
          }
        }
      }

  if (s.ok()) {
    // key
    // We must track all the locked keys so that we can unlock them later.
If
    // the key is already locked, this func will update some stats on the
    // tracked key. It could also update the tracked_at_seq if it is lower
than
    // the existing trackey seq.
    TrackKey(cfh_id, key_str, tracked_at_seq, read_only, exclusive);
  }
}
```

OptimisticTransaction

OptimisticTransactionDB不使用锁进行key的独占，只在commit是进行冲突检测。所以OptimisticTransaction::TryLock如下：

```
Status OptimisticTransaction::TryLock(ColumnFamilyHandle* column_family,
                                      const Slice& key, bool read_only,
                                      bool exclusive, bool untracked) {
  if (untracked) {
    return Status::OK();
  }
  uint32_t cfh_id = GetColumnFamilyID(column_family);

  SetSnapshotIfNeeded();
  // snapshotkeyseq
  // snapshotsequencekeyseq
  SequenceNumber seq;
  if (snapshot_) {
    seq = snapshot_->GetSequenceNumber();
  } else {
    seq = db_->GetLatestSequenceNumber();
  }

  std::string key_str = key.ToString();
  // keyseqcommitseq
  // keysequence
  TrackKey(cfh_id, key_str, seq, read_only, exclusive);

  // Always return OK. Confilct checking will happen at commit time.
  return Status::OK();
}
```

这里TryLock实际上就是给key标记一个sequence并记录，用作commit时的冲突检测，commit实现如下：

```
Status OptimisticTransaction::Commit() {
  // Set up callback which will call CheckTransactionForConflicts() to
  // check whether this transaction is safe to be committed.
  OptimisticTransactionCallback callback(this);

  DBImpl* db_impl = static_cast_with_check<DBImpl, DB>(db_->GetRootDB());
  // WriteWithCallbackDB
  Status s = db_impl->WriteWithCallback(
      write_options_, GetWriteBatch()->GetWriteBatch(), &callback);

  if (s.ok()) {
    Clear();
  }

  return s;
}
```

2.3 writebatch

事务会将所有的写操作追加进同一个WriteBatch，直到Commit时才向DB原子写入。

WriteBatchWithIndex在WriteBatch之外，额外搞一个Skiplist来记录每一个操作在WriteBatch中的offset等信息。在事务没有commit之前，数据还不在Memtable中，而是存在WriteBatch里，如果有需要，这时候可以通过WriteBatchWithIndex来拿到自己刚刚写入的但还没有提交的数据。

事务的SetSavePoint和RollbackToSavePoint也是通过WriteBatch来实现的，SetSavePoint记录当前WriteBatch的大小及统计信息，若干操作之后，若想回滚，则只需要将WriteBatch truncate到之前记录的大小并恢复统计信息即可。

write_batch保证原子性的基础。

2.4 Rdb_transaction

myrocks 支持两种事务 一种是Rdb_writebatch_impl 另一种 是Rdb_transaction_impl。

前一种将一次batch 作为一次事务提交，而后一种 是 rocksdb 内置的事务了。

而myrocks对事务的选择，可以从源码里看出 如果跳过 tx_api 并且是 在主从同步 使用 Rdb_writebatch_impl ，或者 在 master 跳过 tx_api 时，没有主从同步 ，也是用Rdb_writebatch_impl

Rdb_writebatch_impl 这种事务 缺少很多机制，没有Rdb_transaction_impl 健全，比如锁，batch 就没有。

```
static Rdb_transaction *get_or_create_tx(THD *const thd) {
  Rdb_transaction *&tx = get_tx_from_thd(thd);
  // TODO: this is called too many times.. O(#rows)
  if (tx == nullptr) {
    if ((rpl_skip_tx_api && thd->rli_slave) ||
        (THDVAR(thd, master_skip_tx_api) && !thd->rli_slave))
    {
      tx = new Rdb_writebatch_impl(thd);
    }
    else
    {
      tx = new Rdb_transaction_impl(thd);
    }
    tx->set_params(THDVAR(thd, lock_wait_timeout), THDVAR(thd,
max_row_locks));
    tx->start_tx();
  } else {
    tx->set_params(THDVAR(thd, lock_wait_timeout), THDVAR(thd,
max_row_locks));
    if (!tx->is_tx_started()) {
      tx->start_tx();
    }
  }

  return tx;
}
```

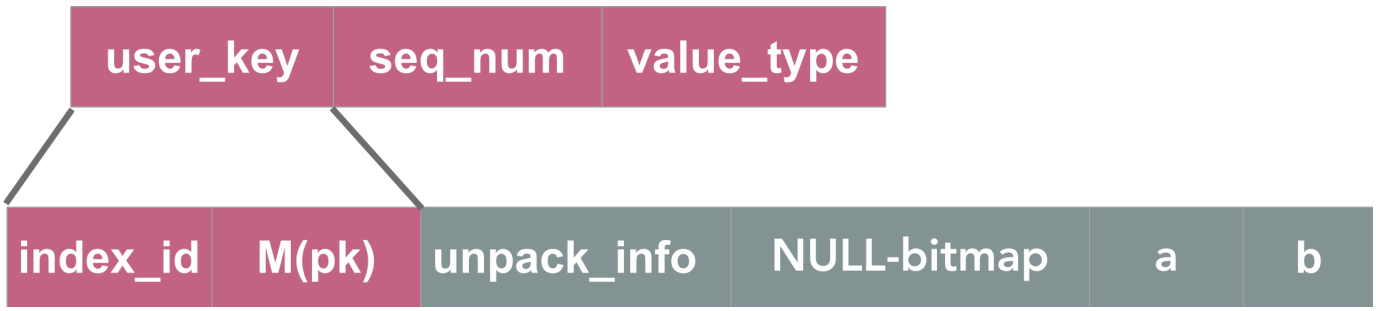
三、事务实现

3.1 sequence number

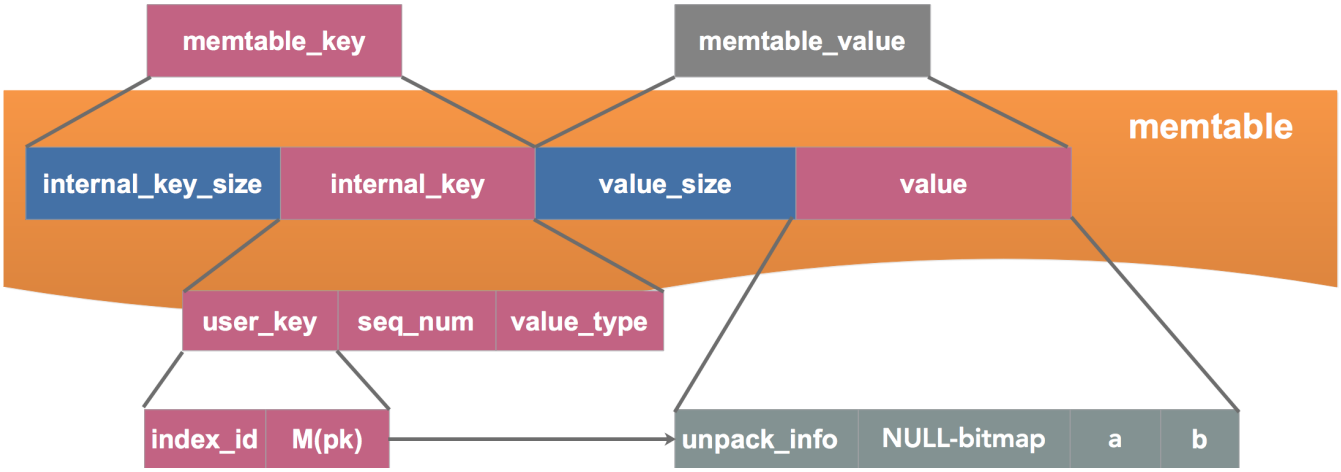
谈到rocksdb事务，就必须提及rocksdb中的sequence number机制。rocksdb中的每一条记录都有一个sequence number, 这个sequence number存储在记录的key中。

```
InternalKey: | User key (string) | sequence number (7 bytes) | value type
(1 byte) |
```

1. user_key：key
2. seq_num：全局递增 sequence，用于 MVCC
3. value_type：put、merge、delete



- 将 internal key 封装为 memtable key，memtable 中的最终数据格式



对于同样的User key记录，在rocksdb中可能存在多条，但他们的sequence number不同。
sequence number是实现事务处理的关键，同时也是MVCC的基础。

3.2 snapshot

snapshot是rocksdb的快照信息，snapshot实际就是对应一个sequence number。 简单的讲，假设snapshot的sequence number为Sa, 那么对于此snapshot来说，只能看到sequence number<=sa的记录，sequence number>sa的记录是不可见的。

- snapshot 结构
  snapshot 主要包含sequence number和snapshot创建时间,sequence number 取自当前的sequence number。

```
class SnapshotImpl : public Snapshot {
  SequenceNumber number_;  // sequence number
  int64_t unix_time_;      // snapshot
  ......
};
```

- snapshot 管理
  snapshot由全局双向链表管理，根据sequence number排序。snapshot的创建和删除都需要维护双向链表。
- snapshot与compact
  rocksdb的compact操作与snapshot有紧密联系。以我们熟悉的innodb为例，rocksdb的compact类似于innodb的purge操作， 而snapshot类似于InnoDB的read view。 innodb做purge操作时会根据已有的read view来判断哪些undo log可以purge，而rocksdb的compact操作会根据已有snapshot信息即全局双向链表来判断哪些记录在compace时可以清理。
  判断的大体原则是，从全局双向链表取出最小的snapshot sequence number Sn。 如果已删除的老记录sequence number <=Sn, 那么这些老记录在compact时可以清理掉。

3.3 MVCC

有了snapshot，MVCC实现起来就很顺利了。记录的sequence number天然的提供了记录的多版本信息。 每次查询用户记录时，并不需要加锁。而是根据当前的sequence number Sn创建一个snapshot, 查询过程中只取小于或等于Sn的最大sequence number的记录。查询结束时释放snapshot。

关键代码段

```
DBIter::FindNextUserEntryInternal

 if (ikey.sequence <= sequence_) {
   if (skipping &&
      user_comparator_->Compare(ikey.user_key, saved_key_.GetKey()) <= 0) {
     num_skipped++;  // skip this entry
     PERF_COUNTER_ADD(internal_key_skipped_count, 1);
   } else {
     switch (ikey.type) {
       case kTypeDeletion:
       case kTypeSingleDeletion:
         // Arrange to skip all upcoming entries for this key since
         // they are hidden by this deletion.
         saved_key_.SetKey(
             ikey.user_key,
             !iter_->IsKeyPinned() || !pin_thru_lifetime_ /* copy */);
         skipping = true;
         num_skipped = 0;
         PERF_COUNTER_ADD(internal_delete_skipped_count, 1);
         break;
       case kTypeValue:
         valid_ = true;
         saved_key_.SetKey(
             ikey.user_key,
             !iter_->IsKeyPinned() || !pin_thru_lifetime_ /* copy */);
         return;
       case kTypeMerge:

       ......
```

3.4 隔离级别

隔离级别也是通过snapshot来实现的。在innodb中，隔离级别为read-committed时，事务中每的个stmt都会建立一个read view, 隔离级别为repeatable-read时,只在事务开启时建立一次read view。 rocksdb同innodb类似，隔离级别为read-committed时，事务中每的个stmt都会建立一个snapshot, 隔离级别为repeatable-read时,只在事务开启时第一个stmt建立一次snapshot。

关键代码片段

```
rocksdb_commit:

  if (my_core::thd_tx_isolation(thd) <= ISO_READ_COMMITTED)
  {
    // For READ_COMMITTED, we release any existing snapshot so that we will
    // see any changes that occurred since the last statement
    tx->release_snapshot();
  }
```

- 隔离级别实现差异
  在read committed隔离级别下，如果一个大事务要更新1000w行，当它更新了前900w行时，
  同时另一个事务已经更新了后100w行，那么myrocks会重新获取快照，再次尝试更新，这样 更新的是新提交的数据，也符合read committed逻辑。具体的讨论可以参考最近的issue#340. 而之前的处理方式是直接报死锁错误。

```
rocksdb::Status ha_rocksdb::get_for_update(
    Rdb_transaction*              tx,
    rocksdb::ColumnFamilyHandle* column_family,
    const rocksdb::Slice&         key,
    std::string*                  value) const
{
  rocksdb::Status s= tx->get_for_update(column_family, key, value);

  // If we have a lock conflict and we are running in READ COMMITTTED mode
  // release and reacquire the snapshot and then retry the
get_for_update().
  if (s.IsBusy() && my_core::thd_tx_isolation(ha_thd()) ==
ISO_READ_COMMITTED)
  {
    tx->release_snapshot();
    tx->acquire_snapshot(false);

    s= tx->get_for_update(column_family, key, value);
  }

  return s;
}
```

innodb不会出现上述情况，当第一个大事更新是会持有b树的index lock, 第二个事务会一直等待index lock直至第一个事务提交完成。

3.5 锁

myrocks目前只支持一种锁类型：排他锁（X锁），并且所有的锁信息都保存在内存中。

- 锁结构
  每个锁实际上存储的哪条记录被哪个事务锁住。

```

```

```
struct LockInfo {
  TransactionID txn_id;

  // Transaction locks are not valid after this time in us
  uint64_t expiration_time;
  ......
  }
```

每个锁实际是key和LockInfo的映射。 锁信息都保存在map中

```
struct LockMapStripe {
  std::unordered_map<std::string, LockInfo> keys;
  ......
}
```

为了减少全局锁信息访问的冲突， rocksdb将锁信息进行按key hash分区，

```
struct LockMap {
    std::vector<LockMapStripe*> lock_map_stripes_;
}
```

column family LockMap

```
using LockMaps = std::unordered_map<uint32_t, std::shared_ptr<LockMap>>;
LockMaps lock_maps_;
```

锁相关参数：
max_num_locks：事务锁个数限制
expiration：事务过期时间

通过设置以上两个参数，来控制事务锁占用过多的内存。

- 死锁检测

rocksdb内部实现了简单的死锁检测机制，每次加锁发生等待时都会向下面的map中插入一条等待信息，表示一个事务id等待另一个事务id。 同时会检查wait_txn_map_是否存在等待环路，存在环路则发生死锁。

```
std::unordered_map<TransactionID, TransactionID> wait_txn_map_;
```

死锁检测关键代码片段

```
TransactionLockMgr::IncrementWaiters:

        for (int i = 0; i < txn->GetDeadlockDetectDepth(); i++) {
          if (next == id) {
            DecrementWaitersImpl(txn, wait_id);
            return true;
          } else if (wait_txn_map_.count(next) == 0) {
            return false;
          } else {
            next = wait_txn_map_[next];
          }
        }
```

死锁检测相关参数
deadlock_detect：是否开启死锁检测
deadlock_detect_depth：死锁检查深度，默认50

- gap lock

innodb中是存在gap lock的，主要是为了实现repeatable read和唯一性检查的。 而在rocksdb中，不支持gap lock(rocksdb insert是也会多对唯一键加锁，以防止重复插入， 严格的来讲也算是gap lock)。

那么在rocksdb一些需要gap lock的地方，目前是报错和打印日志来处理的。

相关参数
gap_lock_write_log： 只打印日志，不返回错误
gap_lock_raise_error： 打印日志并且返回错误

- 锁示例

直接看例子

四、二级索引

- **主键索引记录**

```
key: index_id, M(pk)
value: unpack_info, NULL-bitmap,a,b
```

index_id	M(pk)	unpack_info	NULL-bitmap	a	b

1. index_id：索引 ID，全局唯一，4B
2. M(pk)：转化后的主键，转化后的 pk 可以直接 memcmp
3. unpack_info：pk 逆转化的信息
4. NULL-bitmap： 表示为 NULL 的字段
5. a/b：数据

综上：数据与主键索引保存在一起，MyRocks 的主键索引为聚簇索引

- **二级索引记录**

```
key: index_id,NULL-byte, M(b),M(pk)
value: unpack_info
```

index_id	NULL-byte	M(b)	M(pk)	unpack_info

1. index_id：二级索引 ID
2. NULL-byte：索引 b 是否为空
3. M(b)：转化后的二级索引
4. M(pk)：转化后的主键
5. unpack_info：逆转化信息

五、崩溃恢复

DBImpl类中实现

```
  // hollow transactions shell used for recovery.
  // these will then be passed to TransactionDB so that
  // locks can be reacquired before writing can resume.
  struct RecoveredTransaction {
    uint64_t log_number_;
    std::string name_;
    WriteBatch* batch_;
    // The seq number of the first key in the batch
    SequenceNumber seq_;
    explicit RecoveredTransaction(const uint64_t log, const std::string&
name,
                                  WriteBatch* batch, SequenceNumber seq)
        : log_number_(log), name_(name), batch_(batch), seq_(seq) {}

    ~RecoveredTransaction() { delete batch_; }
  };

  bool allow_2pc() const { return immutable_db_options_.allow_2pc; }

  std::unordered_map<std::string, RecoveredTransaction*>
  recovered_transactions() {
    return recovered_transactions_;
  }

  RecoveredTransaction* GetRecoveredTransaction(const std::string& name) {
    auto it = recovered_transactions_.find(name);
    if (it == recovered_transactions_.end()) {
      return nullptr;
    } else {
      return it->second;
    }
  }

  void InsertRecoveredTransaction(const uint64_t log, const std::string&
name,
                                  WriteBatch* batch, SequenceNumber seq) {
    recovered_transactions_[name] =
        new RecoveredTransaction(log, name, batch, seq);
    MarkLogAsContainingPrepSection(log);
  }
```

```
void DeleteRecoveredTransaction(const std::string& name) {
  auto it = recovered_transactions_.find(name);
  assert(it != recovered_transactions_.end());
  auto* trx = it->second;
  recovered_transactions_.erase(it);
  MarkLogAsHavingPrepSectionFlushed(trx->log_number_);
  delete trx;
}

void DeleteAllRecoveredTransactions() {
  for (auto it = recovered_transactions_.begin();
       it != recovered_transactions_.end(); it++) {
    delete it->second;
  }
  recovered_transactions_.clear();
}
```