Solana代码解析

1. 术语

- lamport：1 lamport = 10^{-9} SOL
- shred：A fraction of a block; the smallest unit sent between validators。为block的一部分，validators之间传输的最小单元。
- ledger entry：账本上的entry要么为tick要么为transactions entry。其中tick用于：estimates wallclock duration；而transactions entry 是指：A set of transactions that may be executed in parallel。
- slot：The period of time for which each leader ingests transactions and produces a block.【A leader produces at most one block per slot. 即每个slot最多只有一个区块。Solana中，设置1个slot对应64个ticks。】
  Collectively, slots create a logical clock. Slots are ordered sequentially and non-overlapping, comprising roughly equal real-world time as per PoH.
  ・・・
  /// Slot is a unit of time given to a leader for encoding,
  /// is some some number of Ticks long.
  pub type Slot = u64;
  ・・・
- skipped slot：A past slot that did not produce a block, because the leader was offline or the fork containing the slot was abandoned for a better alternative by cluster consensus. A skipped slot will not appear as an ancestor for blocks at subsequent slots, nor increment the block height, nor expire the oldest recent_blockhash.
  Whether a slot has been skipped can only be determined when it becomes older than the latest rooted (thus not-skipped) slot.
- transaction：One or more instructions signed by the client using one or more keypairs and executed atomically with only two possible outcomes: success or failure.
  Solana的交易中会封装recent_hash，可通过`getRecentBlockhash` rpc 接口获得：

```
$ curl https://api.testnet.solana.com -X POST -H "Content-Type: application/json" -d '
  {"jsonrpc":"2.0","id":1, "method":"getRecentBlockhash"}
'
{"jsonrpc":"2.0","result":{"context":{"slot":91170278},"value":{"blockhash":"78WXFoVnDdBcqFVQp3peAKzma3igxBXhpat1CGpRUPX4","feeCalculator":{"lamportsPerSignature":5000}}},"id":1}
```

- transaction id：The first signature in a transaction, which can be used to uniquely identify the transaction across the complete ledger.
- transaction confirmations：The number of confirmed blocks since the transaction was accepted onto the ledger. A transaction is finalized when its block becomes a root.
- transactions entry：A set of transactions that may be executed in parallel.
- epoch：目前Solana一个epoch对应43.2万个slot。【若一个slot对应400ms，则一个epoch对应48小时。】
- blockhash：A preimage resistant hash of the ledger at a given block height. Taken from the last entry id in the slot。Slot中的最后一个entry id作为block hash。即：【粒度为：一个slot内若没有交易，则对应为64个tick；若有交易，一个slot对应有多个entry，每个entry对应多笔交易，每个entry会切分为多个shreds便于传输，会对每个shred进行签名`sign_shred()`。】

```
let new_extra_entry = Entry::new(&prev_entry_hash, 1, vec![extra_tx]);
// 实际即为 `hashv(prev_entry_hash, hash_transactions(transactions))`
```

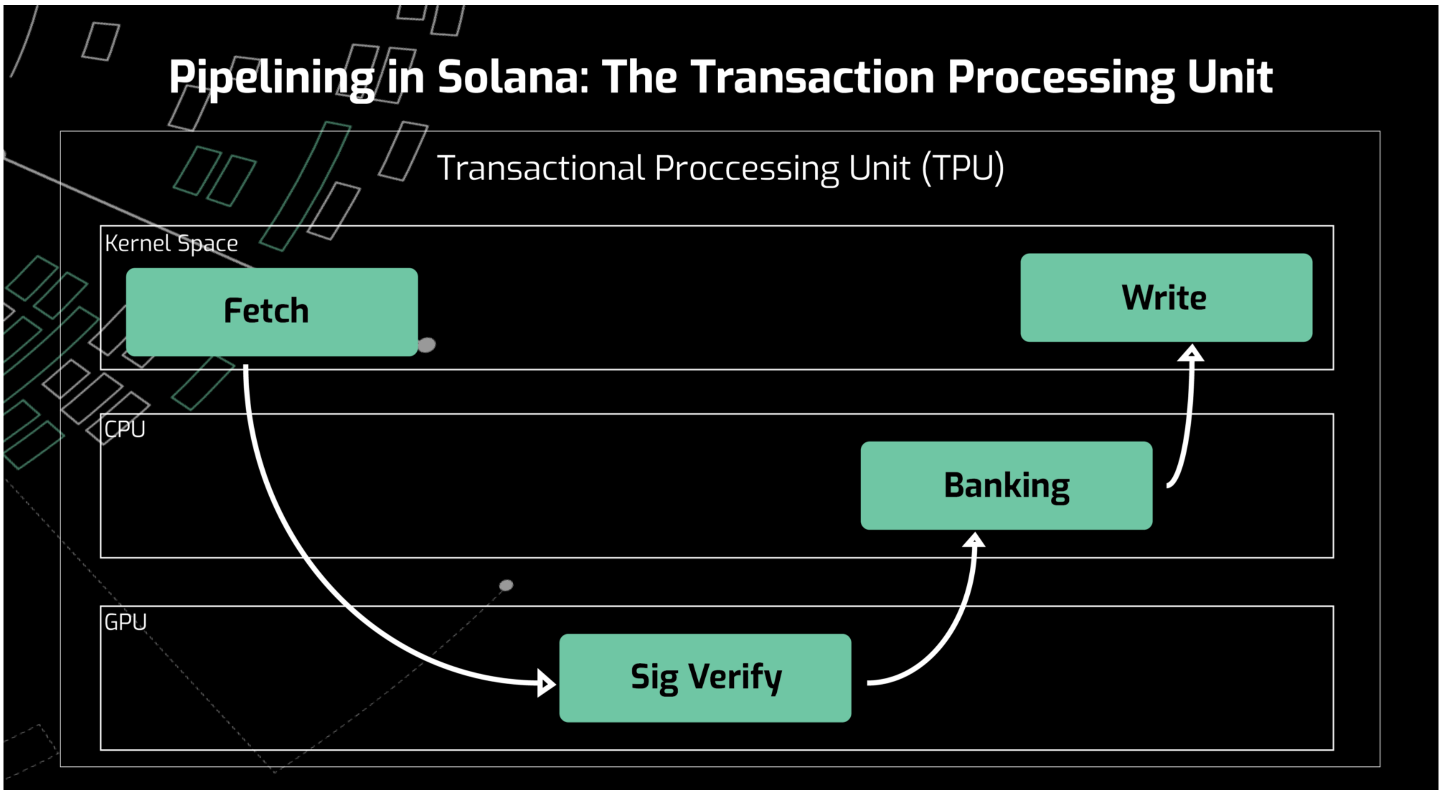
```
/// Tell the bank which Entry IDs exist on the ledger. This function
/// assumes subsequent calls correspond to later entries, and will boot
/// the oldest ones once its internal cache is full. Once boot, the
/// bank will reject transactions using that `hash`.
pub fn register_tick(&self, hash: &Hash) {
    assert!(
        !self.freeze_started(),
        "register_tick() working on a bank that is already frozen or is undergoing freezing!"
    );

    inc_new_counter_debug!("bank-register_tick-registered", 1);
    let mut w_blockhash_queue = self.blockhash_queue.write().unwrap();
    if self.is_block_boundary(self.tick_height.load(Relaxed) + 1) {
        w_blockhash_queue.register_hash(hash, &self.fee_calculator);
        self.update_recent_blockhashes_locked(&w_blockhash_queue);
    }
    // ReplayStage will start computing the accounts delta hash when it
    // detects the tick height has reached the boundary, so the system
    // needs to guarantee all account updates for the slot have been
    // committed before this tick height is incremented (like the blockhash
    // sysvar above)
    self.tick_height.fetch_add(1, Relaxed);
}

pub fn is_complete(&self) -> bool {
    self.tick_height() == self.max_tick_height()
}

pub fn is_block_boundary(&self, tick_height: u64) -> bool {
    tick_height % self.ticks_per_slot == 0
}
```

Validator节点可同时运行2个pipelined processes，TPU对应为leader mode，TVU对应为validator mode。在TPU和TVU这2种情况下，硬件的pipeline顺序是一样的：the network input -> the GPU cards -> the CPU cores -> writes to disk -> the network output。但是，TPU主要用来创建ledger entries，而TVU用于validate ledger entries。
根据 `validator/src/main.rs` 和 `core/src/validator.rs`：启动Validator时，会同时配置TPU和TVU资源。


Pipelining in Solana: The Transaction Processing Unit
Transactional Proccessing Unit (TPU)
Kernel Space
Fetch
Write
CPU
Banking
GPU
Sig Verify

Fetch stage：将(<=)1024笔交易打包为packet，发送给Sigverify stage。

Sigverify stage：分4个线程接收fetch stage发来的packet，调用`ed25519_verify`（借助GPU/CPU）进行批量验签，对验签不通过和签名不合格的交易，将其中的meta.discard标记为true。将所有交易发给Banking stage。

Banking stage：分3个线程（verified_receiver）接收sigverify stage发来的packet，有1个线程（verified_vote_receiver）用于接收投票信息。timeout时长为（10ms或100ms）来接收packet。
支持最多buffer 50万笔交易（若超过，则会pop_front，然后再push_back。），每次batch处理128笔交易。

Broadcast stage：会通过entry_receiver接收Banking stage中process_and_record_transactions_locked()通过PohRecorder.record() 发送的已成功执行的交易，在processe_receive_results()将entries转换为shreds（会sign_shred()签名，为可传输block的最小单位），会按质押权重通过broadcast_shreds() 将shreds发送给相应的nodes，同时将shreds通过record()存入rocksdb数据库中。

2. TPU

为Validator作为leader角色运行的Transaction Processing Unit（TPU），用于create ledger entries。

- `fetch_stage`： batches input from a UDP socket and sends it to a channel.
- `sigverify_stage`： implements the signature verification stage of the TPU. It receives a list of lists of packets and outputs the same list, but tags each top-level list with a list of booleans, telling the next stage whether the signature in that packet is valid. It assumes each packet contains one transaction. All processing is done on the CPU by default and on a GPU if perf-libs are available.
- `banking_stage`： processes Transaction messages. It is intended to be used to construct a software pipeline. The stage uses all available CPU cores and can do its processing in parallel with signature verification on the GPU.
- `broadcast_state`： A stage to broadcast data from a leader node to validators.
- `core/src/consensus.rs`： 包含了Tower BFT的实现以及相关测试用例。
- `core/src/cluster_info_vote_listener.rs`： 定义了`SlotVoteTracker`来map投票情况。
- `core/src/cost_model.rs`： 用于评估交易开销，提供的维度有writable_accounts、account_access_cost和execution_cost。在banking_stage会调用`could_fit()`函数评估交易开销是否超过limit。

```
pub struct TransactionCost {
    pub writable_accounts: Vec<Pubkey>,
    pub account_access_cost: u64,
    pub execution_cost: u64,
}

/*
self.would_fit(
            &tx_cost.writable_accounts,
            &(tx_cost.account_access_cost + tx_cost.execution_cost),
        )
*/
fn would_fit(&self, keys: &[Pubkey], cost: &u64) -> Result<(), CostModelError> {
        // check against the total package cost
        if self.block_cost + cost > self.block_cost_limit {
            return Err(CostModelError::WouldExceedBlockMaxLimit);
        }

        // check if the transaction itself is more costly than the account_cost_limit
        if *cost > self.account_cost_limit {
            return Err(CostModelError::WouldExceedAccountMaxLimit);
        }

        // check each account against account_cost_limit,
        for account_key in keys.iter() {
            match self.cost_by_writable_accounts.get(account_key) {
                Some(chained_cost) => {
                    if chained_cost + cost > self.account_cost_limit {
                        return Err(CostModelError::WouldExceedAccountMaxLimit);
                    } else {
                        continue;
                    }
                }
                None => continue,
            }
        }

        Ok(())
    }
// Guestimated from mainnet-beta data, sigver averages 1us, average read 7us and average write 25us
const SIGVER_COST: u64 = 1;
const NON_SIGNED_READONLY_ACCOUNT_ACCESS_COST: u64 = 7;
const NON_SIGNED_WRITABLE_ACCOUNT_ACCESS_COST: u64 = 25;
const SIGNED_READONLY_ACCOUNT_ACCESS_COST: u64 =
    SIGVER_COST + NON_SIGNED_READONLY_ACCOUNT_ACCESS_COST;
const SIGNED_WRITABLE_ACCOUNT_ACCESS_COST: u64 =
    SIGVER_COST + NON_SIGNED_WRITABLE_ACCOUNT_ACCESS_COST;
```

```
pub struct SlotVoteTracker {
    // Maps pubkeys that have voted for this slot
    // to whether or not we've seen the vote on gossip.
    // True if seen on gossip, false if only seen in replay.
    voted: HashMap<Pubkey, bool>,
    optimistic_votes_tracker: HashMap<Hash, VoteStakeTracker>,
    voted_slot_updates: Option<Vec<Pubkey>>,
    gossip_only_stake: u64,
}
```

```
                             .------------.
                             | Upstream   |
                             | Validators |
                             `----+-------`
                                  |
                                  |
           .-----------------------------------.
           | Validator            |            |
           |                      v            |
           |  .-----------.   .------------.   |
.--------. |  | Fetch     |   | Repair     |   |
| Client +---->| Stage     |   | Stage      |   |
`--------` |  `---+-------`   `----+-------`   |
           |      |                |           |
           |      v                v           |
           |  .-----------.   .------------.   |
           |  | TPU       |<-->| Blockstore |   |
           |  |           |   |            |   |
           |  `-----------`   `----+-------`   |
           |                       |           |
           |                       v           |
           |                 .------------.    |
           |                 | Multicast  |    |
           |                 | Stage      |    |
           |                 `----+-------`    |
           |                      |            |
           `-----------------------------------`
                                  |
                                  v
                            .------------.
                            | Downstream |
                            | Validators |
                            `------------`


                            .------------.
                            | PoH        |
                            | Service    |
                            `-------+----`
                                ^   |
                                |   |
           .-----------------------------------.
           | TPU                |   |          |
           |                    |   v          |
.-------.  |  .-----------.   .---+--------.  |  .------------.
| Fetch +---->| SigVerify +--->| Banking    |<--->| Blockstore |
| Stage |  |  | Stage     |   | Stage      |  |  |            |
`-------`  |  `-----------`   `-----+------`  |  `------------`
           |                        |         |
           |                        |         |
           `-----------------------------------`
                                    |
                                    v
                            .------------.
                            | Banktree   |
                            |            |
                            `------------`
```

对应的TPU:

```
Self {
  fetch_stage,
  sigverify_stage,
  banking_stage,
  cluster_info_vote_listener,
  broadcast_stage,
}
```

3. TVU

为Validator作为validator 角色运行的Transaction Verification Unit（TVU），用于validate ledger entries。

```
                                                  +------------+
                                                  |   Gossip   |
                                                  |   Service  |
                      +-------------+                     |           |
                      |Child        |    +------------+       +-+-----+----+
                      |Validators   |    |Neighborhood|        |     ^
                      |             |   |Validators  |        |     |
                      +-------+-----+    |           |        |     |
                           ^          +---------+--+     Peer|     |Votes
                           |                    ^        List|     |
                           +-----------------+  |           |     |
                         shreds(forward=true)|  |           |     |
                 +--------------------------------------------------------------------------------------+
                 |                           | |          |     |                          |
                 |  TVU                        | |   +--------+     |                             |
                 |                           | |   v            |                          |
+------------+           |  +-------+  +------------+    +-+--+---+---+      +---+--------------+      +---------+ |
|            |    Repair |  |       |  |            |    | Retransmit |      | Replay           |      | Storage | |
| Upstream   +------------->+       +->+ Shred      +--->+ Stage      +----->+ Stage            +----->+ Stage   | |
| Validators |     TVU   |  | Shred |  | Verify     |    |            |      | +--------------+ |      |         | |
|            +------------->+ Fetch |  | Leader Sig |    +------+-----+      | | PoH Verify   | |      +---------+ |
|            |           |  | Stage |  | Stage      |           ^           | | TX Sig Verify| |                  |
|            +------------->+       |  |            |           |           | |              | |                  |
|            |  TVU      |  |       |  +--+---------+           |           | +-+------------+ |                  |
+------------+  Forwards|  +-------+     ^                    |           |  |            |                  |
                 |           |                |           +------------------+                 |
                 |           |                |             |                          |
                 |           |                |             |                          |
                 |           |                |             |                          |
                 +--------------------------------------------------------------------------------------+
                             |                |             |
                             |                |Validator      v
                             |                |Stakes      +---+-----+
                             |                +------------+         |
                             +---------------------------------+ Bank    |
                                 Leader                   |         |
                                 Schedule                 +---------+
```

批量验签过程中做了判断，若有GPU则走GPU验签，否则走CPU验签：

```
pub fn verify_shreds_gpu(
    batches: &[Packets],
    slot_leaders: &HashMap<u64, [u8; 32]>,
    recycler_cache: &RecyclerCache,
) -> Vec<Vec<u8>> {
    let api = perf_libs::api();
    if api.is_none() {
        return verify_shreds_cpu(batches, slot_leaders);
    }
    let api = api.unwrap();

    let mut elems = Vec::new();
    let mut rvs = Vec::new();
    let count = batch_size(batches);
    let (pubkeys, pubkey_offsets, mut num_packets) =
        slot_key_data_for_gpu(0, batches, slot_leaders, recycler_cache);
    //HACK: Pubkeys vector is passed along as a `Packets` buffer to the GPU
    //TODO: GPU needs a more opaque interface, which can handle variable sized structures for data
    let pubkeys_len = num_packets * size_of::<Packet>();
    trace!("num_packets: {}", num_packets);
    trace!("pubkeys_len: {}", pubkeys_len);
    let (signature_offsets, msg_start_offsets, msg_sizes, v_sig_lens) =
        shred_gpu_offsets(pubkeys_len, batches, recycler_cache);
    let mut out = recycler_cache.buffer().allocate("out_buffer");
    out.set_pinnable();
    elems.push(perf_libs::Elems {
        #[allow(clippy::cast_ptr_alignment)]
        elems: pubkeys.as_ptr() as *const solana_sdk::packet::Packet,
        num: num_packets as u32,
    });

    for p in batches {
        elems.push(perf_libs::Elems {
            elems: p.packets.as_ptr(),
            num: p.packets.len() as u32,
        });
        let mut v = Vec::new();
        v.resize(p.packets.len(), 0);
        rvs.push(v);
        num_packets += p.packets.len();
    }
    out.resize(signature_offsets.len(), 0);

    trace!("Starting verify num packets: {}", num_packets);
    trace!("elem len: {}", elems.len() as u32);
    trace!("packet sizeof: {}", size_of::<Packet>() as u32);
    const USE_NON_DEFAULT_STREAM: u8 = 1;
    unsafe {
        let res = (api.ed25519_verify_many)(
            elems.as_ptr(),
            elems.len() as u32,
            size_of::<Packet>() as u32,
            num_packets as u32,
            signature_offsets.len() as u32,
            msg_sizes.as_ptr(),
            pubkey_offsets.as_ptr(),
            signature_offsets.as_ptr(),
            msg_start_offsets.as_ptr(),
            out.as_mut_ptr(),
            USE_NON_DEFAULT_STREAM,
        );
        if res != 0 {
            trace!("RETURN!!!: {}", res);
        }
    }
    trace!("done verify");
    trace!("out buf {:?}", out);

    sigverify::copy_return_values(&v_sig_lens, &out, &mut rvs);

    inc_new_counter_debug!("ed25519_shred_verify_gpu", count);
    rvs
}
```

4. 交易处理

交易由一系列签名和消息组成：

```
/// An atomic transaction
#[frozen_abi(digest = "AAeVxvWiiotwxDLxKLxsfgkA6ndW74nVbaAEb6cwJYqR")]
#[derive(Debug, PartialEq, Default, Eq, Clone, Serialize, Deserialize, AbiExample)]
pub struct Transaction {
    /// A set of digital signatures of `account_keys`, `program_ids`, `recent_blockhash`, and `instructions`, signed by the first
    /// signatures.len() keys of account_keys
    /// NOTE: Serialization-related changes must be paired with the direct read at sigverify.
    #[serde(with = "short_vec")]
    pub signatures: Vec<Signature>,

    /// The message to sign.
    pub message: Message,
}
```

4.1 锁定交易相关账号

借助MutexGuard，锁定交易相关账号，防止多线程同时对同一账号状态进行修改：

```
// Once accounts are locked, other threads cannot encode transactions that will modify the
        // same account state
        let batch = bank.prepare_sanitized_batch(txs);

/// This function will prevent multiple threads from modifying the same account state at the
    /// same time
    #[must_use]
    #[allow(clippy::needless_collect)]
    pub fn lock_accounts<'a>(&self, txs: impl Iterator<Item = &'a Transaction>) -> Vec<Result<()>> {
        let keys: Vec<_> = txs
            .map(|tx| tx.message().get_account_keys_by_lock_type())
            .collect();
        let mut account_locks = &mut self.account_locks.lock().unwrap();
        keys.into_iter()
            .map(|(writable_keys, readonly_keys)| {
                self.lock_account(&mut account_locks, writable_keys, readonly_keys)
            })
            .collect()
    }
/// Once accounts are unlocked, new transactions that modify that state can enter the pipeline
    #[allow(clippy::needless_collect)]
    pub fn unlock_accounts<'a>(
        &self,
        txs: impl Iterator<Item = &'a Transaction>,
        results: &[Result<()>],
    )

pub fn prepare_sanitized_batch<'a, 'b>(
        &'a self,
        sanitized_txs: &'b [SanitizedTransaction],
    ) -> TransactionBatch<'a, 'b> {
        let lock_results = self
            .rc
            .accounts
            .lock_accounts(sanitized_txs.as_transactions_iter());
        TransactionBatch::new(lock_results, self, Cow::Borrowed(sanitized_txs))
    }
```

4.2 标记交易有效性

`vec![1u8, 0u8, 1u8]` 可有效标记交易的有效性，具体体现在discard字段，无效的交易标记为1。

```
pub fn convert_from_old_verified(mut with_vers: Vec<(Packets, Vec<u8>)>) -> Vec<Packets> {
        with_vers.iter_mut().for_each(|(b, v)| {
            b.packets
                .iter_mut()
                .zip(v)
                .for_each(|(p, f)| p.meta.discard = *f == 0)
        });
        with_vers.into_iter().map(|(b, _)| b).collect()
    }

fn test_banking_stage_entries_only() {
        ........

            // fund another account so we can send 2 good transactions in a single batch.
            let keypair = Keypair::new();
            let fund_tx =
                system_transaction::transfer(&mint_keypair, &keypair.pubkey(), 2, start_hash);
            bank.process_transaction(&fund_tx).unwrap();

            // good tx
            let to = solana_sdk::pubkey::new_rand();
            let tx = system_transaction::transfer(&mint_keypair, &to, 1, start_hash);

            // good tx, but no verify
            let to2 = solana_sdk::pubkey::new_rand();
            let tx_no_ver = system_transaction::transfer(&keypair, &to2, 2, start_hash);

            // bad tx, AccountNotFound
            let keypair = Keypair::new();
            let to3 = solana_sdk::pubkey::new_rand();
            let tx_anf = system_transaction::transfer(&keypair, &to3, 1, start_hash);

            // send 'em over
            let packets = to_packets_chunked(&[tx_no_ver, tx_anf, tx], 3);

            // glad they all fit
            assert_eq!(packets.len(), 1);

            let packets = packets
                .into_iter()
                .map(|packets| (packets, vec![1u8, 0u8, 1u8]))
                .collect();
           ........

            assert_eq!(bank.get_balance(&to), 1);
            assert_eq!(bank.get_balance(&to2), 2);

            drop(entry_receiver);
        }
        Blockstore::destroy(&ledger_path).unwrap();
    }
```

`sdk/src/transaction.rs` 中定义了交易错误类型有：

```
pub enum TransactionError {
    /// An account is already being processed in another transaction in a way
    /// that does not support parallelism
    #[error("Account in use")]
    AccountInUse,

    /// A `Pubkey` appears twice in the transaction's `account_keys`.  Instructions can reference
    /// `Pubkey`s more than once but the message must contain a list with no duplicate keys
    #[error("Account loaded twice")]
    AccountLoadedTwice,
    /// Attempt to debit an account but found no record of a prior credit.
    #[error("Attempt to debit an account but found no record of a prior credit.")]
    AccountNotFound,

    /// Attempt to load a program that does not exist
    #[error("Attempt to load a program that does not exist")]
    ProgramAccountNotFound,

    /// The from `Pubkey` does not have sufficient balance to pay the fee to schedule the transaction
    #[error("Insufficient funds for fee")]
    InsufficientFundsForFee,

    /// This account may not be used to pay transaction fees
    #[error("This account may not be used to pay transaction fees")]
    InvalidAccountForFee,

    /// The bank has seen this transaction before. This can occur under normal operation
    /// when a UDP packet is duplicated, as a user error from a client not updating
    /// its `recent_blockhash`, or as a double-spend attack.
    #[error("This transaction has already been processed")]
    AlreadyProcessed,

    /// The bank has not seen the given `recent_blockhash` or the transaction is too old and
    /// the `recent_blockhash` has been discarded.
    #[error("Blockhash not found")]
    BlockhashNotFound,

    /// An error occurred while processing an instruction. The first element of the tuple
    /// indicates the instruction index in which the error occurred.
    #[error("Error processing Instruction {0}: {1}")]
    InstructionError(u8, InstructionError),

    /// Loader call chain is too deep
    #[error("Loader call chain is too deep")]
    CallChainTooDeep,

    /// Transaction requires a fee but has no signature present
    #[error("Transaction requires a fee but has no signature present")]
    MissingSignatureForFee,

    /// Transaction contains an invalid account reference
    #[error("Transaction contains an invalid account reference")]
    InvalidAccountIndex,

    /// Transaction did not pass signature verification
    #[error("Transaction did not pass signature verification")]
    SignatureFailure,

    /// This program may not be used for executing instructions
    #[error("This program may not be used for executing instructions")]
    InvalidProgramForExecution,

    /// Transaction failed to sanitize accounts offsets correctly
    /// implies that account locks are not taken for this TX, and should
    /// not be unlocked.
    #[error("Transaction failed to sanitize accounts offsets correctly")]
    SanitizeFailure,

    #[error("Transactions are currently disabled due to cluster maintenance")]
    ClusterMaintenance,

    /// Transaction processing left an account with an outstanding borrowed reference
    #[error("Transaction processing left an account with an outstanding borrowed reference")]
    AccountBorrowOutstanding,
    }
```

sdk/src/banking.rs 中 load_and_execute_transactions() 函数会对净化交易进一步处理，在 banking_stage 中会调用该函数。 transactions_from_packets() 中借助 verify_precompiles() 对交易进行净化，排除 AccountNotFound 和 InvalidAccountIndex 的错误交易。 process_loop() -> 【process_buffered_packets() -> consume_buffered_packets() 和 process_packets() -> process_packets_transactions() 】 -> process_transactions() -> process_and_record_transactions() -> process_and_record_transactions_locked() -> load_and_execute_transactions()

```
/// This function will prevent multiple threads from modifying the same account state at the
/// same time
#[must_use]
#[allow(clippy::needless_collect)]
pub fn lock_accounts<'a>(&self, txs: impl Iterator<Item = &'a Transaction>) -> Vec<Result<()>> {
    let keys: Vec<_> = txs
        .map(|tx| tx.message().get_account_keys_by_lock_type())
        .collect();
    let mut account_locks = &mut self.account_locks.lock().unwrap();
    keys.into_iter()
        .map(|(writable_keys, readonly_keys)| {
            self.lock_account(&mut account_locks, writable_keys, readonly_keys)
        })
        .collect()
}
pub fn prepare_sanitized_batch<'a, 'b>(
    &'a self,
    sanitized_txs: &'b [SanitizedTransaction],
) -> TransactionBatch<'a, 'b> {
    let lock_results = self
        .rc
        .accounts
        .lock_accounts(sanitized_txs.as_transactions_iter());
    TransactionBatch::new(lock_results, self, Cow::Borrowed(sanitized_txs))
}
```

```
pub fn load_and_execute_transactions(
    &self,
    batch: &TransactionBatch,
    max_age: usize,
    enable_cpi_recording: bool,
    enable_log_recording: bool,
    timings: &mut ExecuteTimings,
) -> (
    Vec<TransactionLoadResult>,
    Vec<TransactionExecutionResult>,
    Vec<Option<InnerInstructionsList>>,
    Vec<Option<TransactionLogMessages>>,
    Vec<usize>,
    u64,
    u64,
) {
    let sanitized_txs = batch.sanitized_transactions();
    debug!("processing transactions: {}", sanitized_txs.len());
    inc_new_counter_info!("bank-process_transactions", sanitized_txs.len());
    let mut error_counters = ErrorCounters::default();

    let retryable_txs: Vec<_> = batch
        .lock_results()
        .iter()
        .enumerate()
        .filter_map(|(index, res)| match res {
            Err(TransactionError::AccountInUse) => {
                error_counters.account_in_use += 1;
                Some(index)
        }
        Err(_) => None,
        Ok(_) => None,
    })
    .collect();

let mut check_time = Measure::start("check_transactions");
let check_results = self.check_transactions(
    sanitized_txs,
    batch.lock_results(),
    max_age,
    &mut error_counters,
);
check_time.stop();

let mut load_time = Measure::start("accounts_load");
let mut loaded_txs = self.rc.accounts.load_accounts(
    &self.ancestors,
    sanitized_txs.as_transactions_iter(),
    check_results,
    &self.blockhash_queue.read().unwrap(),
    &mut error_counters,
    &self.rent_collector,
    &self.feature_set,
);
load_time.stop();

let mut execution_time = Measure::start("execution_time");
let mut signature_count: u64 = 0;
let mut inner_instructions: Vec<Option<InnerInstructionsList>> =
    Vec::with_capacity(sanitized_txs.len());
let mut transaction_log_messages: Vec<Option<Vec<String>>> =
    Vec::with_capacity(sanitized_txs.len());

let executed: Vec<TransactionExecutionResult> = loaded_txs
    .iter_mut()
    .zip(sanitized_txs.as_transactions_iter())
    .map(|(accs, tx)| match accs {
        (Err(e), _nonce_rollback) => {
            transaction_log_messages.push(None);
            inner_instructions.push(None);
            (Err(e.clone()), None)
        }
        (Ok(loaded_transaction), nonce_rollback) => {
            let feature_set = self.feature_set.clone();
            signature_count += u64::from(tx.message().header.num_required_signatures);

            let mut compute_budget = self.compute_budget.unwrap_or_else(ComputeBudget::new);

            let mut process_result = if feature_set.is_active(&tx_wide_compute_cap::id()) {
                compute_budget.process_transaction(tx)
            } else {
                Ok(())
            };

            if process_result.is_ok() {
                let executors =
                    self.get_executors(&tx.message, &loaded_transaction.loaders);

                let (account_refcells, loader_refcells) = Self::accounts_to_refcells(
                    &mut loaded_transaction.accounts,
                    &mut loaded_transaction.loaders,
                );

                let instruction_recorders = if enable_cpi_recording {
                    let ix_count = tx.message.instructions.len();
        let ix_count = tx.message.instructions.len();
        let mut recorders = Vec::with_capacity(ix_count);
        recorders.resize_with(ix_count, InstructionRecorder::default);
        Some(recorders)
    } else {
        None
    };

    let log_collector = if enable_log_recording {
        Some(Rc::new(LogCollector::default()))
    } else {
        None
    };

    let compute_meter = Rc::new(RefCell::new(TransactionComputeMeter::new(
        compute_budget.max_units,
    )));

    process_result = self.message_processor.process_message(
        tx.message(),
        &loader_refcells,
        &account_refcells,
        &self.rent_collector,
        log_collector.clone(),
        executors.clone(),
        instruction_recorders.as_deref(),
        feature_set,
        compute_budget,
        compute_meter,
        &mut timings.details,
        self.rc.accounts.clone(),
        &self.ancestors,
    );

    transaction_log_messages.push(Self::collect_log_messages(log_collector));
    inner_instructions.push(Self::compile_recorded_instructions(
        instruction_recorders,
        &tx.message,
    ));

    if let Err(e) = Self::refcells_to_accounts(
        &mut loaded_transaction.accounts,
        &mut loaded_transaction.loaders,
        account_refcells,
        loader_refcells,
    ) {
        warn!("Account lifetime mismanagement");
        process_result = Err(e);
    }

    if process_result.is_ok() {
        self.update_executors(executors);
    }
} else {
    transaction_log_messages.push(None);
    inner_instructions.push(None);
}

let nonce_rollback =
    if let Err(TransactionError::InstructionError(_, _)) = &process_result {
        error_counters.instruction_error += 1;
        nonce_rollback.clone()
    } else if process_result.is_err() {
        None
    } else {
        nonce_rollback.clone()
```

```
                    nonce_rollback.clone()
                };
            (process_result, nonce_rollback)
        }
    })
    .collect();

execution_time.stop();

debug!(
    "check: {}us load: {}us execute: {}us txs_len={}",
    check_time.as_us(),
    load_time.as_us(),
    execution_time.as_us(),
    sanitized_txs.len(),
);
timings.check_us = timings.check_us.saturating_add(check_time.as_us());
timings.load_us = timings.load_us.saturating_add(load_time.as_us());
timings.execute_us = timings.execute_us.saturating_add(execution_time.as_us());

let mut tx_count: u64 = 0;
let err_count = &mut error_counters.total;
let transaction_log_collector_config =
    self.transaction_log_collector_config.read().unwrap();

for (i, ((r, _nonce_rollback), tx)) in executed.iter().zip(sanitized_txs).enumerate() {
    if let Some(debug_keys) = &self.transaction_debug_keys {
        for key in &tx.message.account_keys {
            if debug_keys.contains(key) {
                info!("slot: {} result: {:?} tx: {:?}", self.slot, r, tx);
                break;
            }
        }
    }

    if Self::can_commit(r) // Skip log collection for unprocessed transactions
        && transaction_log_collector_config.filter != TransactionLogCollectorFilter::None
    {
        let mut transaction_log_collector = self.transaction_log_collector.write().unwrap();
        let transaction_log_index = transaction_log_collector.logs.len();

        let mut mentioned_address = false;
        if !transaction_log_collector_config
            .mentioned_addresses
            .is_empty()
        {
            for key in &tx.message.account_keys {
                if transaction_log_collector_config
                    .mentioned_addresses
                    .contains(key)
                {
                    transaction_log_collector
                        .mentioned_address_map
                        .entry(*key)
                        .or_default()
                        .push(transaction_log_index);
                    mentioned_address = true;
                }
            }
        }

        let is_vote = is_simple_vote_transaction(tx);
        let store = match transaction_log_collector_config.filter {
            TransactionLogCollectorFilter::All => !is_vote || mentioned_address,
            TransactionLogCollectorFilter::AllWithVotes => true,
            TransactionLogCollectorFilter::None => false
```

```
                    TransactionLogCollectorFilter::None => false,
                    TransactionLogCollectorFilter::OnlyMentionedAddresses => mentioned_address,
                };

                if store {
                    if let Some(log_messages) = transaction_log_messages.get(i).cloned().flatten() {
                        transaction_log_collector.logs.push(TransactionLogInfo {
                            signature: tx.signatures[0],
                            result: r.clone(),
                            is_vote,
                            log_messages,
                        });
                    }
                }
            }

            if r.is_ok() {
                tx_count += 1;
            } else {
                if *err_count == 0 {
                    debug!("tx error: {:?} {:?}", r, tx);
                }
                *err_count += 1;
            }
        }
        if *err_count > 0 {
            debug!(
                "{} errors of {} txs",
                *err_count,
                *err_count as u64 + tx_count
            );
        }
        Self::update_error_counters(&error_counters);
        (
            loaded_txs,
            executed,
            inner_instructions,
            transaction_log_messages,
            retryable_txs,
            tx_count,
            signature_count,
        )
    }
```

5. PoH

其中一种PoH为：

```
pub struct TransactionRecorder {
    // shared by all users of PohRecorder
    pub record_sender: CrossbeamSender<Record>,
    pub is_exited: Arc<AtomicBool>,
}
// 有：
poh: &TransactionRecorder,
```

以及另一种PoH为：

```
poh: &Arc<Mutex<PohRecorder>>,

pub struct PohRecorder {
    pub poh: Arc<Mutex<Poh>>,
    tick_height: u64,
    clear_bank_signal: Option<SyncSender<bool>>,
    start_slot: Slot,              // parent slot
    start_tick_height: u64,        // first tick_height this recorder will observe
    tick_cache: Vec<(Entry, u64)>, // cache of entry and its tick_height
    working_bank: Option<WorkingBank>,
    sender: Sender<WorkingBankEntry>,
    leader_first_tick_height: Option<u64>,
    leader_last_tick_height: u64, // zero if none
    grace_ticks: u64,
    id: Pubkey,
    blockstore: Arc<Blockstore>,
    leader_schedule_cache: Arc<LeaderScheduleCache>,
    poh_config: Arc<PohConfig>,
    ticks_per_slot: u64,
    target_ns_per_tick: u64,
    record_lock_contention_us: u64,
    flush_cache_no_tick_us: u64,
    flush_cache_tick_us: u64,
    prepare_send_us: u64,
    send_us: u64,
    tick_lock_contention_us: u64,
    tick_overhead_us: u64,
    total_sleep_us: u64,
    record_us: u64,
    ticks_from_record: u64,
    last_metric: Instant,
    record_sender: CrossbeamSender<Record>,
    pub is_exited: Arc<AtomicBool>,
}

pub struct Poh {
    pub hash: Hash,
    num_hashes: u64,
    hashes_per_tick: u64,
    remaining_hashes: u64,
    ticks_per_slot: u64,
    tick_number: u64,
    slot_start_time: Instant,
}
```

6. concurrent database

Cloudbreak：Horizontally-Scaled Accounts Database
对应实现在：`runtime/src/accounts_db.rs`：

```
//! Persistent accounts are stored in below path location:
//!  <path>/<pid>/data/
//!
//! The persistent store would allow for this mode of operation:
//!  - Concurrent single thread append with many concurrent readers.
//!
//! The underlying memory is memory mapped to a file. The accounts would be
//! stored across multiple files and the mappings of file and offset of a
//! particular account would be stored in a shared index. This will allow for
//! concurrent commits without blocking reads, which will sequentially write
//! to memory, ssd or disk, and should be as fast as the hardware allow for.
//! The only required in memory data structure with a write lock is the index,
//! which should be fast to update.
//!
//! AppendVec's only store accounts for single slots.  To bootstrap the
//! index from a persistent store of AppendVec's, the entries include
//! a "write_version".  A single global atomic `AccountsDb::write_version`
//! tracks the number of commits to the entire data store. So the latest
//! commit for each slot entry would be indexed.
```

7. 投票

Solana中的 中有:

```
Vote Program#
Create and manage accounts that track validator voting state and rewards.

Program id: Vote111111111111111111111111111111111111111
Instructions: VoteInstruction
```

中描述了Vote账号的相关管理规则。

中指出:
Validator会从当前leader中接收到entries，并提交votes来确认那些entries是valid的。提交vote的过程中存在相关安全风险，因为伪造的投票违背了共识，将会slash掉该Validator的stake。