Filecoin-Sector State Management Logic

FileCoin (https://learnblockchain.cn/tags/FileCoin)

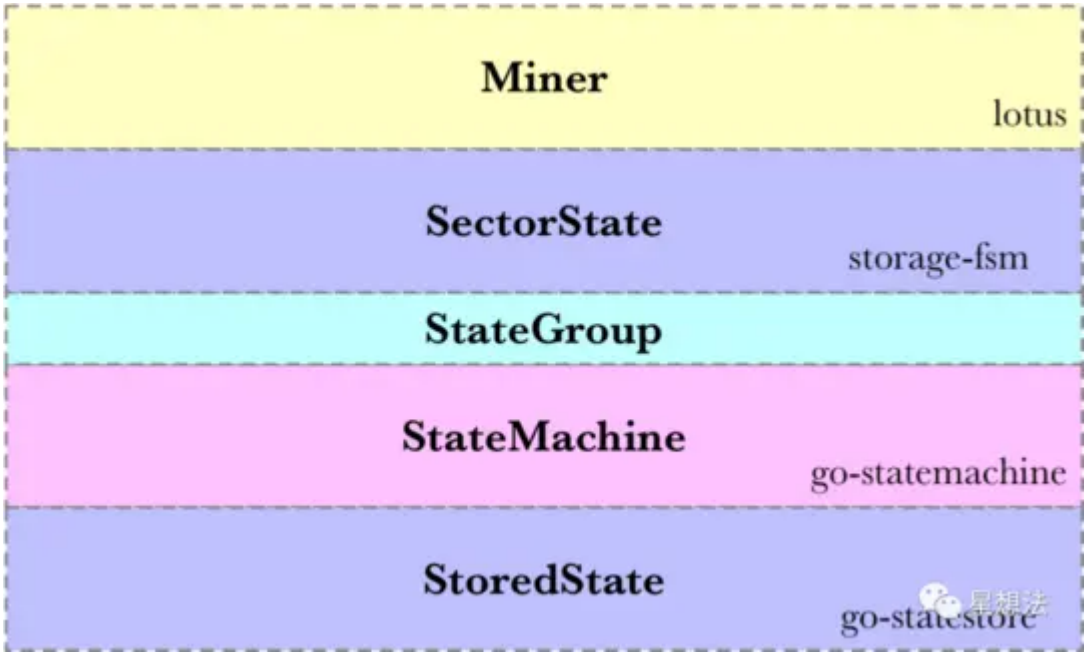
> The state management of a sector is based on a state machine. The realization of the general state machine is realized by go-statemachine. State storage is realized through go-statestore. On the basis of these modules, storage-fsm implements the state definition and state processing functions of the Sector.

It’s been a long time since I looked at the code of the go language. Recently I have time to change my mind and look at the state management of Sector, which is implemented in pure go. Look at the code design of a large project, it is of great benefit to the design and development of the code and the project. The latest code of the go language part of Lotus adopts a modular design. Lotus's code is also evolving step by step. The first version of the code is also all logic coupled together, and then it is gradually modularized.

Lotus Miner "packages" the data that users need to store into sectors, and processes the sectors. This article will talk about the state management of Sector.

01 Module frame

The modules related to Sector status management are as follows:



The state management of a sector is based on a state machine. Lotus implements a general state machine (statemachine), in the go-statemachine package. On top of StateMachine, StateGroup is further abstracted to manage multiple state machines. StateMachine only realizes the transition of state, and the specific state is stored through go-statestore. On the StateMachine, the rules for state transitions and the processing functions corresponding to the state are defined. These are in the specific business. SectorState is the specific business of Lotus to manage Sector. On the basis of these low-level modules, Lotus related code calls are relatively simple. Let's start with the underlying module of state management, StateMachine:

02 StateMachine

StateMachine is defined in the go-statemachine package, machine.go:

```
type StateMachine struct {
planner  Planner
eventsIn chan Event

name      interface{}
st        *statestore.StoredState
stateType reflect.Type

stageDone chan struct{}
closing   chan struct{}
closed    chan struct{}

busy int32
}

```

Among them, planner is the state transformation function of the abstracted state machine. Planner receives the Event and determines the next step in combination with the current status of the user.

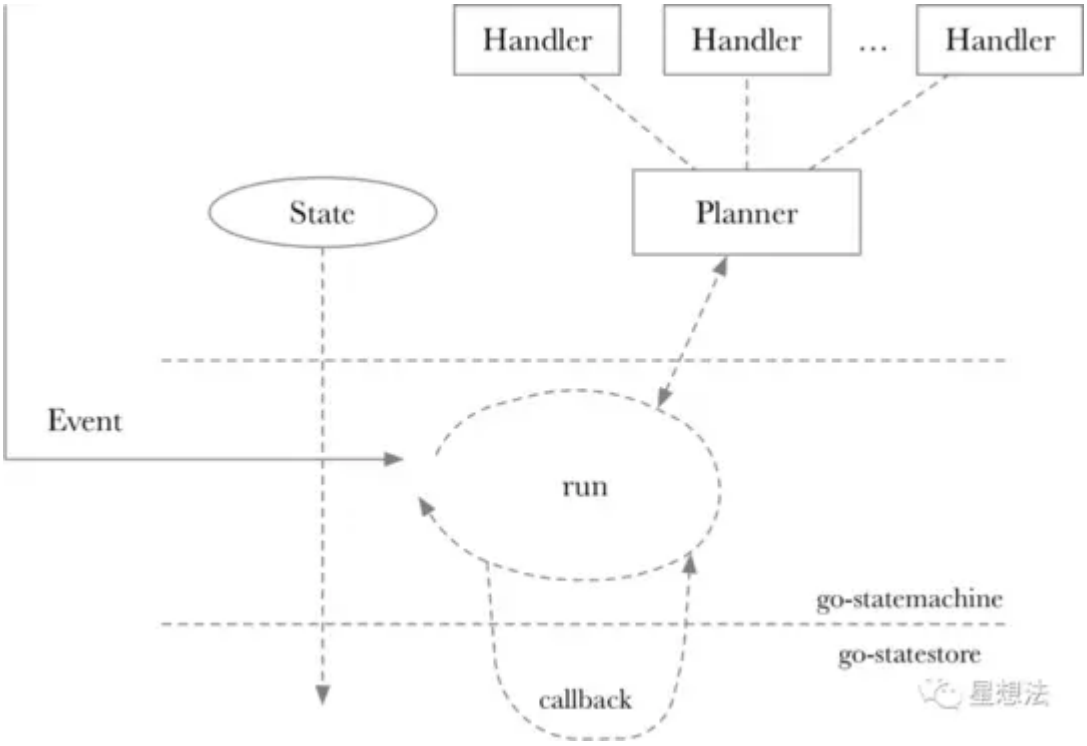
```
type Planner func(events []Event, user interface{}) (interface{}, uint64, error)
```

st is the stored state. stateType is the type of storage state.

The core of StateMachine is the run function, which is divided into three parts: receiving Event, state processing, and the next call. State processing is the key:

```
err := fsm.mutateUser(func(user interface{}) (err error) {
 nextStep, processed, err = fsm.planner(pendingEvents, user)
 ustate = user
 if xerrors.Is(err, ErrTerminated) {
 terminated = true
 return nil
 }
 return err
})
```

MutateUser is to view the current stored state (StoredState), execute the planner function, and store the state processed by the planner. The planner function returns to the next step of processing, the number of events that have been processed, and there may be errors. The run function will start another go routine to execute nextStep.



The specific implementation of mutateUser is to use the reflection of go to achieve abstraction and modularity. Friends who are interested in related logic can check it out for themselves.

03 StateGroup

Often the business needs a lot of StateMachine. For example, Lotus Miner stores multiple Sectors, and each Sector maintains an independent state by a StateMachine. StateGroup is to implement multiple "same" StateMachines, defined in group.go.

```
type StateGroup struct {
sts       *statestore.StateStore
hnd       StateHandler
stateType reflect.Type

closing      chan struct{}
initNotifier sync.Once

lk  sync.Mutex
sms map[datastore.Key]*StateMachine
}
```

Among them, sms is the state array of StateMachine. StateHandler is the state processing function interface of StateMachine:

```
type StateHandler interface {
Plan(events []Event, user interface{}) (interface{}, uint64, error)
}
```

That is to say, on the StateMachine state machine, it is necessary to implement the StateHandler interface (Plan function, which realizes the state transition).

04 StoredState

StoredState is an abstract Key-Value storage. Relatively simple, the Get/Put interface is very easy to understand.

05 SectorState

storage-fsm implements the business logic related to the sector state. That is to say, the definition of the state, and the state transition functions are all implemented in this package. The information of the entire Sector is defined in storage-fsm/types.go:

```
type SectorInfo struct {
State        SectorState
SectorNumber abi.SectorNumber
Nonce        uint64
SectorType abi.RegisteredProof
// Packing
Pieces []Piece
// PreCommit1
TicketValue   abi.SealRandomness
TicketEpoch   abi.ChainEpoch
PreCommit1Out storage.PreCommit1Out
// PreCommit2
CommD *cid.Cid
CommR *cid.Cid
Proof []byte
PreCommitMessage *cid.Cid
// WaitSeed
SeedValue abi.InteractiveSealRandomness
SeedEpoch abi.ChainEpoch
// Committing
CommitMessage *cid.Cid
InvalidProofs uint64
// Faults
FaultReportMsg *cid.Cid
// Debug
LastErr string
Log []Log
}
```

SectorInfo includes Sector status, Precommit1/2 data, Committing data and so on. Among them, SectorState describes the specific state of the Sector. All the states of the sector are defined in the sector_state.go file:

```
const (
UndefinedSectorState SectorState = ""

// happy path
Empty          SectorState = "Empty"
Packing        SectorState = "Packing"       // sector not in sealStore, and not on chain
PreCommit1     SectorState = "PreCommit1"    // do PreCommit1
PreCommit2     SectorState = "PreCommit2"    // do PreCommit1
PreCommitting  SectorState = "PreCommitting" // on chain pre-commit
WaitSeed       SectorState = "WaitSeed"      // waiting for seed
Committing     SectorState = "Committing"
CommitWait     SectorState = "CommitWait" // waiting for message to land on chain
FinalizeSector SectorState = "FinalizeSector"
Proving        SectorState = "Proving"
// error modes
FailedUnrecoverable SectorState = "FailedUnrecoverable"
SealFailed          SectorState = "SealFailed"
PreCommitFailed     SectorState = "PreCommitFailed"
ComputeProofFailed  SectorState = "ComputeProofFailed"
CommitFailed        SectorState = "CommitFailed"
PackingFailed       SectorState = "PackingFailed"
Faulty              SectorState = "Faulty"        // sector is corrupted or gone for some reason
FaultReported       SectorState = "FaultReported" // sector has been declared as a fault on chai
FaultedFinal        SectorState = "FaultedFinal"  // fault declared on chain
)
```

Friends who are familiar with the SDR algorithm will find many familiar words: PreCommit1, PreCommit2, Commiting and so on. Combined with the state processing function, you can clearly understand the meaning of each state and the content that needs to be processed.

The Plan function of the Sealing structure in fsm.go is the processing function of the Sector state:

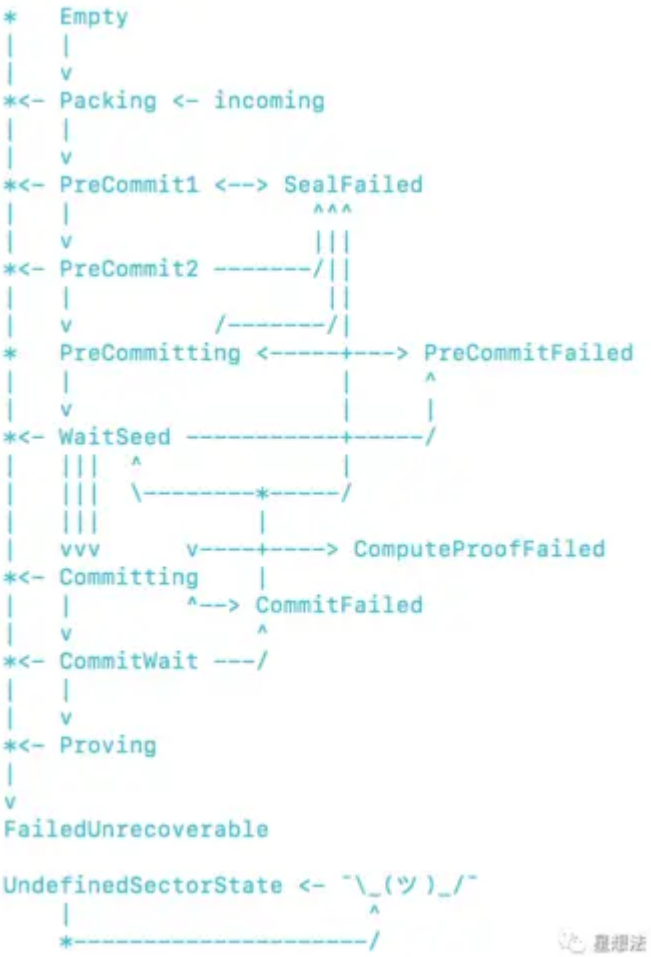
```
1    func (m *Sealing) Plan(events []statemachine.Event, user interface{}) (interface{}, uint64, erro
2    next, err := m.plan(events, user.(*SectorInfo))
3    if err != nil || next == nil {
4    return nil, uint64(len(events)), err
5    }
6
7    return func(ctx statemachine.Context, si SectorInfo) error {
8    err := next(ctx, si)
9    if err != nil {
10   log.Errorf("unhandled sector error (%d): %+v", si.SectorNumber, err)
11   return nil
12   }
13
14   return nil
15   }, uint64(len(events)), nil // TODO: This processed event count is not very correct
16   }
```

The implementation of the Plan function is also relatively simple, calling plan to process the current state and returning to the function that needs to be processed next. State processing can be divided into two parts: 1/definition of state transition 2/state processing.

For the definition of the state transition of Sector, in fsmPlanners:

```
1    var fsmPlanners = map[SectorState]func(events []statemachine.Event, state *SectorInfo) error{
2    UndefinedSectorState: planOne(on(SectorStart{}, Packing)),
3    Packing:              planOne(on(SectorPacked{}, PreCommit1)),
4    PreCommit1: planOne(
5    on(SectorPreCommit1{}, PreCommit2),
6    on(SectorSealPreCommitFailed{}, SealFailed),
7    on(SectorPackingFailed{}, PackingFailed),
8    ),
9    PreCommit2: planOne(
10   on(SectorPreCommit2{}, PreCommitting),
11   on(SectorSealPreCommitFailed{}, SealFailed),
12   on(SectorPackingFailed{}, PackingFailed),
13   ),
14   ...
```

For example, in the state of PreCommit1, if the SectorPreCommit1 event is received, it will enter the state of PreCommit2. The state transition of all Sectors is as follows:



When the processing state is good, it enters the processing program of the corresponding state. Take the PreCommit1 state as an example, the corresponding processing function is handlePreCommit1.

```
1   func (m *Sealing) handlePreCommit1(ctx statemachine.Context, sector SectorInfo) error {
2    tok, epoch, err := m.api.ChainHead(ctx.Context())
3    ...
4    pc1o, err := m.sealer.SealPreCommit1(ctx.Context(), m.minerSector(sector.SectorNumber), ticketVa
5    if err != nil {
6    return ctx.Send(SectorSealPreCommitFailed{xerrors.Errorf("seal pre commit(1) failed: %w", err)})
7    }
8    ...
9    }
```

It is clear that in the processing function of PreCommit1, rust-fil-proofs will be called through SealPrecommit1 to realize the calculation of Precommit1. Finally, to summarize, the meaning of each state:

Empty -empty state

Packing -Packing state, multiple Pieces are filled into one Sector

PreCommit1 -PreCommit1 calculation

PreCommit2 -PreCommit2 calculation

PreCommitting -Submit the result of Precommit2 to the chain

WaitSeed -Waiting for the random seed (given a time of 10 blocks, the random number seed is not predictable in advance)

Committing -Calculate Commit1/Commit2 and submit the proof to the chain

CommitWait -Waiting for confirmation on the chain **FinalizeSector** -Sector status confirmed

to sum up:

The state management of a sector is based on a state machine. The realization of the general state machine is realized by go-statemachine. State storage is realized through go-statestore. On the basis of these modules, storage-fsm implements the state definition and state processing functions of the Sector.

There are many original high-quality articles in my official account **star idea** , and you are welcome to scan the code and pay attention.

This article participates in the DingChain community writing incentive plan (https://learnblockchain.cn/site/coins) , good articles are good for profit, and you are welcome to join as well.

Published on 2020-06-16 11:54 Reading (1667) Credits (111)
Category: FileCoin (https://learnblockchain.cn/categories/FileCoin)