Go-ethereum 씹어먹기 #1

(consensus / type / event / miner)

송무복(Jake)
jake.s@onther.io
18/09/14

Onther Inc.

내용

Onther Inc.

- 패키지별 역할과 사용된 go 문법 및 패턴에 관한 설명

1. 코어, 합의알고리즘 - consensus/ethash
2. 공통 - core/types
3. 코어, 채널 - event
4. 코어, 마이닝 - miner

Onther Inc.

1. 코어, 합의알고리즘
consensus/ethash

1. consensus/consensus.go

Onther Inc.

- ChainReader *1)Interface정의
  - header, uncle 정보를 읽을 수 있는 함수 정의
- Engine Interface정의
  - Verify부터 Seal까지의 함수 정의
- interface에 정의된 함수를 구현하고 struct에 정의해 유연하게 사용

```
// blockchain during header and/or uncle verification.
type ChainReader interface {
    // Config retrieves the blockchain's chain configuration.
    Config() *params.ChainConfig

    // CurrentHeader retrieves the current header from the local chain.
    CurrentHeader() *types.Header

    // GetHeader retrieves a block header from the database by hash and number.
    GetHeader(hash common.Hash, number uint64) *types.Header

    // GetHeaderByNumber retrieves a block header from the database by number.
    GetHeaderByNumber(number uint64) *types.Header

    // GetHeaderByHash retrieves a block header from the database by its hash.
    GetHeaderByHash(hash common.Hash) *types.Header

    // GetBlock retrieves a block from the database by hash and number.
    GetBlock(hash common.Hash, number uint64) *types.Block
}
```

2. ethash/consensus.go

Onther Inc.

- Engine Interface에 정의된 함수 구현
- worker : 마이닝을 하는 독립된 machine
  - channel : go routine 간에 주고 받을 수 있는 pipe (e.g. 하나의 go routine에서 value를 받아 다른 go routine에 보낼 수 있다)

```
// Author implements consensus.Engine, returning the header's coinbase as the
// proof-of-work verified author of the block.
func (ethash *Ethash) Author(header *types.Header) (common.Address, error) {
  return header.Coinbase, nil
}

// VerifyHeader checks whether a header conforms to the consensus rules of the
// stock Ethereum ethash engine.
func (ethash *Ethash) VerifyHeader(chain consensus.ChainReader, header *types.Header, seal bool) error {
  // If we're running a full engine faking, accept any input as valid
  if ethash.config.PowMode == ModeFullFake {
    return nil
  }
  // Short circuit if the header is known, or it's parent not
  number := header.Number.Uint64()
  if chain.GetHeader(header.Hash(), number) != nil {
    return nil
  }
  parent := chain.GetHeader(header.ParentHash, number-1)
  if parent == nil {
    return consensus.ErrUnknownAncestor
  }
  // Sanity checks passed, do a proper verification
  return ethash.verifyHeader(chain, header, parent, false, seal)
}
```

3. ethash/sealer.go

Onther Inc.

- ethash algorism을 이용해 nonce(mix digest)를 구하고 mining하는 함수 구현
- Seal : mining thread 생성
  - select : concurrency 환경에서 go routine이 상호 작용할 수 있게 해준다(let go routine wait on)

```
// mine is the actual proof-of-work miner that searches for a nonce starting from
// seed that results in correct final block difficulty.
func (ethash *Ethash) mine(block *types.Block, id int, seed uint64, abort chan struct{}, found chan *types.Block) {
  // Extract some data from the header
  var (
    header  = block.Header()
    hash    = header.HashNoNonce().Bytes()
    target  = new(big.Int).Div(maxUint256, header.Difficulty)
    number  = header.Number.Uint64()
    dataset = ethash.dataset(number)
  )
  // Start generating random nonces until we abort or find a good one
  var (
    attempts = int64(0)
    nonce    = seed
  )
  logger := log.New("miner", id)
  logger.Trace("Started ethash search for new nonces", "seed", seed)
search:
  for {
    select {
    case <-abort:
      // Mining terminated, update stats and abort
      logger.Trace("Ethash nonce search aborted", "attempts", nonce-seed)
      ethash.hashrate.Mark(attempts)
      break search

    default:
      // We don't have to update hash rate on every nonce, so update after after 2^X nonces
      attempts++
      if (attempts % (1 << 15)) == 0 {
        ethash.hashrate.Mark(attempts)
        attempts = 0
      }
      // Compute the PoW value of this nonce
      digest, result := hashimotoFull(dataset.dataset, hash, nonce)
      if new(big.Int).SetBytes(result).Cmp(target) <= 0 {
        // Correct nonce found, create a new header with it
        header = types.CopyHeader(header)
        header.Nonce = types.EncodeNonce(nonce)
        header.MixDigest = common.BytesToHash(digest)
```

4. ethash/algorithm.go

Onther Inc.

- 유효한 nonce를 구하는 ethash알고리즘을 정의
- 상수 정의
- epochLength

```
const (
  datasetInitBytes   = 1 << 30 // Bytes in dataset at genesis
  datasetGrowthBytes = 1 << 23 // Dataset growth per epoch
  cacheInitBytes     = 1 << 24 // Bytes in cache at genesis
  cacheGrowthBytes   = 1 << 17 // Cache growth per epoch
  epochLength        = 30000   // Blocks per epoch
  mixBytes           = 128     // Width of mix
  hashBytes          = 64      // Hash length in bytes
  hashWords          = 16      // Number of 32 bit ints in a hash
  datasetParents     = 256     // Number of parents of each dataset element
  cacheRounds        = 3       // Number of rounds in cache production
  loopAccesses       = 64      // Number of accesses in hashimoto loop
)
```

4. ethash/algorithm.go

Onther Inc.

- ethash 알고리즘 정의 함수 구현

```go
// value for a particular header hash and nonce.
func hashimoto(hash []byte, nonce uint64, size uint64, lookup func(index uint32) []uint32) ([]byte, []byte) {
  // Calculate the number of theoretical rows (we use one buffer nonetheless)
  rows := uint32(size / mixBytes)

  // Combine header+nonce into a 64 byte seed
  seed := make([]byte, 40)
  copy(seed, hash)
  binary.LittleEndian.PutUint64(seed[32:], nonce)

  seed = crypto.Keccak512(seed)
  seedHead := binary.LittleEndian.Uint32(seed)

  // Start the mix with replicated seed
  mix := make([]uint32, mixBytes/4)
  for i := 0; i < len(mix); i++ {
    mix[i] = binary.LittleEndian.Uint32(seed[i%16*4:])
  }
  // Mix in random dataset nodes
  temp := make([]uint32, len(mix))

  for i := 0; i < loopAccesses; i++ {
    parent := fnv(uint32(i)^seedHead, mix[i%len(mix)]) % rows
    for j := uint32(0); j < mixBytes/hashBytes; j++ {
      copy(temp[j*hashWords:], lookup(2*parent+j))
    }
    fnvHash(mix, temp)
  }
  // Compress mix
  for i := 0; i < len(mix); i += 4 {
    mix[i/4] = fnv(fnv(fnv(mix[i], mix[i+1]), mix[i+2]), mix[i+3])
  }
  mix = mix[:len(mix)/4]

  digest := make([]byte, common.HashLength)
  for i, val := range mix {
    binary.LittleEndian.PutUint32(digest[i*4:], val)
  }
  return digest, crypto.Keccak256(append(seed, digest...))
}
```

Onther Inc.

2. 공통

core / types

1. core/types/block.go

Onther Inc.

- Block header 정의
  - Marshal : go에서 쓰이는 데이터를 전송가능한 json 형식의 byte로 encoding
  - UnMashal : json형식의 byte 데이터를 go에서 쓸 수 있는 데이터로 decoding

```
// Header represents a block header in the Ethereum blockchain.
type Header struct {
    ParentHash  common.Hash    `json:"parentHash"       gencodec:"required"`
    UncleHash   common.Hash    `json:"sha3Uncles"       gencodec:"required"`
    Coinbase    common.Address `json:"miner"            gencodec:"required"`
    Root        common.Hash    `json:"stateRoot"        gencodec:"required"`
    TxHash      common.Hash    `json:"transactionsRoot" gencodec:"required"`
    ReceiptHash common.Hash    `json:"receiptsRoot"     gencodec:"required"`
    Bloom       Bloom           `json:"logsBloom"        gencodec:"required"`
    Difficulty  *big.Int       `json:"difficulty"       gencodec:"required"`
    Number      *big.Int       `json:"number"           gencodec:"required"`
    GasLimit    uint64          `json:"gasLimit"         gencodec:"required"`
    GasUsed     uint64          `json:"gasUsed"          gencodec:"required"`
    Time        *big.Int       `json:"timestamp"        gencodec:"required"`
    Extra       []byte          `json:"extraData"        gencodec:"required"`
    MixDigest   common.Hash    `json:"mixHash"          gencodec:"required"`
    Nonce       BlockNonce      `json:"nonce"            gencodec:"required"`
}
```

2. core/types/block.go

Onther Inc.

- Block 정의

```go
// Body is a simple (mutable, non-safe) data container for storing and moving
// a block's data contents (transactions and uncles) together.
type Body struct {
  Transactions []*Transaction
  Uncles       []*Header
}

// Block represents an entire block in the Ethereum blockchain.
type Block struct {
  header       *Header
  uncles       []*Header
  transactions Transactions

  // caches
  hash atomic.Value
  size atomic.Value

  // Td is used by package core to store the total difficulty
  // of the chain up to and including the block.
  td *big.Int

  // These fields are used by package eth to track
  // inter-peer block relay.
  ReceivedAt   time.Time
  ReceivedFrom interface{}
}
```

3. core/types/block.go

Onther Inc.

- 블록 생성 함수
- & : pointer(memory address를 가리킴)
- * : pointer가 가리키는 memory address에 저장되어 있는 value
- &로 접근한다는 것은 Block struct field 값을 수정한다는 것

```
// and receipts.
func NewBlock(header *Header, txs []*Transaction, uncles []*Header, receipts []*Receipt) *Block {
  b := &Block{header: CopyHeader(header), td: new(big.Int)}

  // TODO: panic if len(txs) != len(receipts)
  if len(txs) == 0 {
    b.header.TxHash = EmptyRootHash
  } else {
    b.header.TxHash = DeriveSha(Transactions(txs))
    b.transactions = make(Transactions, len(txs))
    copy(b.transactions, txs)
  }

  if len(receipts) == 0 {
    b.header.ReceiptHash = EmptyRootHash
  } else {
    b.header.ReceiptHash = DeriveSha(Receipts(receipts))
    b.header.Bloom = CreateBloom(receipts)
  }

  if len(uncles) == 0 {
    b.header.UncleHash = EmptyUncleHash
  } else {
    b.header.UncleHash = CalcUncleHash(uncles)
    b.uncles = make([]*Header, len(uncles))
    for i := range uncles {
      b.uncles[i] = CopyHeader(uncles[i])
    }
  }
```

4. core/types/transaction.go

Onther Inc.

- 거래생성 함수 구현
- 컨트랙트 생성 함수 구현

```go
func NewTransaction(nonce uint64, to common.Address, amount *big.Int, gasLimit uint64, gasPrice *big.Int, data []byte) *Transaction {
  return newTransaction(nonce, &to, amount, gasLimit, gasPrice, data)
}

func NewContractCreation(nonce uint64, amount *big.Int, gasLimit uint64, gasPrice *big.Int, data []byte) *Transaction {
  return newTransaction(nonce, nil, amount, gasLimit, gasPrice, data)
}

func newTransaction(nonce uint64, to *common.Address, amount *big.Int, gasLimit uint64, gasPrice *big.Int, data []byte) *Transaction {
  if len(data) > 0 {
    data = common.CopyBytes(data)
  }
  d := txdata{
    AccountNonce: nonce,
    Recipient:    to,
    Payload:      data,
    Amount:       new(big.Int),
    GasLimit:     gasLimit,
    Price:        new(big.Int),
    V:            new(big.Int),
    R:            new(big.Int),
    S:            new(big.Int),
  }
  if amount != nil {
    d.Amount.Set(amount)
  }
  if gasPrice != nil {
    d.Price.Set(gasPrice)
  }

  return &Transaction{data: d}
}
```

5. core/types/receipt.go

Onther Inc.

- receipt 생성
- receipt rlp encoding / decoding
  - rlp : 통신할 때나 DB에 저장할 사용하는 serialization 방법

```go
// NewReceipt creates a barebone transaction receipt, copying the init fields.
func NewReceipt(root []byte, failed bool, cumulativeGasUsed uint64) *Receipt {
  r := &Receipt{PostState: common.CopyBytes(root), CumulativeGasUsed: cumulativeGasUsed}
  if failed {
    r.Status = ReceiptStatusFailed
  } else {
    r.Status = ReceiptStatusSuccessful
  }
  return r
}

// EncodeRLP implements rlp.Encoder, and flattens the consensus fields of a receipt
// into an RLP stream. If no post state is present, byzantium fork is assumed.
func (r *Receipt) EncodeRLP(w io.Writer) error {
  return rlp.Encode(w, &receiptRLP{r.statusEncoding(), r.CumulativeGasUsed, r.Bloom, r.Logs})
}

// DecodeRLP implements rlp.Decoder, and loads the consensus fields of a receipt
// from an RLP stream.
func (r *Receipt) DecodeRLP(s *rlp.Stream) error {
  var dec receiptRLP
  if err := s.Decode(&dec); err != nil {
    return err
  }
  if err := r.setStatus(dec.PostStateOrStatus); err != nil {
    return err
  }
  r.CumulativeGasUsed, r.Bloom, r.Logs = dec.CumulativeGasUsed, dec.Bloom, dec.Logs
  return nil
}
```

6. core/types/log.go

Onther Inc.

- 컨트랙트 이벤트 정의

```go
// Log represents a contract log event. These events are generated by the LOG opcode and
// stored/indexed by the node.
type Log struct {
  // Consensus fields:
  // address of the contract that generated the event
  Address common.Address `json:"address" gencodec:"required"`
  // list of topics provided by the contract.
  Topics []common.Hash `json:"topics" gencodec:"required"`
  // supplied by the contract, usually ABI-encoded
  Data []byte `json:"data" gencodec:"required"`

  // Derived fields. These fields are filled in by the node
  // but not secured by consensus.
  // block in which the transaction was included
  BlockNumber uint64 `json:"blockNumber"`
  // hash of the transaction
  TxHash common.Hash `json:"transactionHash" gencodec:"required"`
  // index of the transaction in the block
  TxIndex uint `json:"transactionIndex" gencodec:"required"`
  // hash of the block in which the transaction was included
  BlockHash common.Hash `json:"blockHash"`
  // index of the log in the receipt
  Index uint `json:"logIndex" gencodec:"required"`

  // The Removed field is true if this log was reverted due to a chain reorganisation.
  // You must pay attention to this field if you receive logs through a filter query.
  Removed bool `json:"removed"`
}
```

Onther Inc.

3. 코어, 채널

event

1. event/event.go

Onther Inc.

- multi threads 상황에서 mutex 를 관리하는 함수 구현
- event : mutex를 쓸 어떠한 이벤트 (e.g. mining)
- Subscribe : mutex를 쓸 이벤트를 등 록하여 mutex 사용
  - mutex : 여러 go routine이 하나의 st -ruct에 접근할 시 field 값이 올바르기 사용 및 수정될 수 있도록 입출력을

```go
// Subscribe creates a subscription for events of the given types. The
// subscription's channel is closed when it is unsubscribed
// or the mux is closed.
func (mux *TypeMux) Subscribe(types ...interface{}) *TypeMuxSubscription {
  sub := newsub(mux)
  mux.mutex.Lock()
  defer mux.mutex.Unlock()
  if mux.stopped {
    // set the status to closed so that calling Unsubscribe after this
    // call will short circuit.
    sub.closed = true
    close(sub.postC)
  } else {
    if mux.subm == nil {
      mux.subm = make(map[reflect.Type][]*TypeMuxSubscription)
    }
    for _, t := range types {
      rtyp := reflect.TypeOf(t)
      oldsubs := mux.subm[rtyp]
      if find(oldsubs, sub) != -1 {
        panic(fmt.Sprintf("event: duplicate type %s in Subscribe", rtyp))
      }
      subs := make([]*TypeMuxSubscription, len(oldsubs)+1)
      copy(subs, oldsubs)
      subs[len(oldsubs)] = sub
      mux.subm[rtyp] = subs
    }
  }
  return sub
}
```

2. event/subscribtion.go

Onther Inc.

- Subscription 함수 구현
- defer : 제일 나중에 실행
- close : channel에 더 이상 보낼
  value가 없음을 명시적으로 선
  더 이상 value를 보낼 수 없다

```go
// called any number of times.
type Subscription interface {
  Err() <-chan error // returns the error channel
  Unsubscribe()      // cancels sending of events, closing the error channel
}

// NewSubscription runs a producer function as a subscription in a new goroutine. The
// channel given to the producer is closed when Unsubscribe is called. If fn returns an
// error, it is sent on the subscription's error channel.
func NewSubscription(producer func(<-chan struct{}) error) Subscription {
  s := &funcSub{unsub: make(chan struct{}), err: make(chan error, 1)}
  go func() {
    defer close(s.err)
    err := producer(s.unsub)
    s.mu.Lock()
    defer s.mu.Unlock()
    if !s.unsubscribed {
      if err != nil {
        s.err <- err
      }
      s.unsubscribed = true
    }
  }()
  return s
}
```

3. event/feed.go

Onther Inc.

- 일대다 상황일 때 한번에 value를 여러 채널에 전달하는 역할

```go
// Feed implements one-to-many subscriptions where the carrier of events is a channel.
// Values sent to a Feed are delivered to all subscribed channels simultaneously.
//
// Feeds can only be used with a single type. The type is determined by the first Send or
// Subscribe operation. Subsequent calls to these methods panic if the type does not
// match.
//
// The zero value is ready to use.
type Feed struct {
  once      sync.Once        // ensures that init only runs once
  sendLock  chan struct{}    // sendLock has a one-element buffer and is empty when held.It protects sendCases.
  removeSub chan interface{} // interrupts Send
  sendCases caseList         // the active set of select cases used by Send

  // The inbox holds newly subscribed channels until they are added to sendCases.
  mu     sync.Mutex
  inbox  caseList
  etype  reflect.Type
  closed bool
}
```

4. event/filter/filter.go

Onther Inc.

- event filter 구현
- event filter : 특정 event 정해 놓고 event가 발생하면 필터링함

```
func (self *Filters) loop() {
out:
  for {
    select {
    case <-self.quit:
      break out
    case event := <-self.ch:
      for _, watcher := range self.watchers {
        if reflect.TypeOf(watcher) == reflect.TypeOf(event.filter) {
          if watcher.Compare(event.filter) {
            watcher.Trigger(event.data)
          }
        }
      }
    }
  }
}
```

Onther Inc.

4. 코어, 마이닝

miner

1. miner/miner.go

Onther Inc.

- Backend interface 정의
- miner struct 정의
- New : chain 설정, event, consensus engine을 가져와 miner 생성

```go
// Backend wraps all methods required for mining.
type Backend interface {
  AccountManager() *accounts.Manager
  BlockChain() *core.BlockChain
  TxPool() *core.TxPool
  ChainDb() ethdb.Database
}

// Miner creates blocks and searches for proof-of-work values.
type Miner struct {
  mux *event.TypeMux

  worker *worker

  coinbase common.Address
  mining   int32
  eth      Backend
  engine   consensus.Engine

  canStart    int32 // can start indicates whether we can start the mining operation
  shouldStart int32 // should start indicates whether we should start after sync
}

func New(eth Backend, config *params.ChainConfig, mux *event.TypeMux, engine consensus.Engine) *Miner {
  miner := &Miner{
    eth:      eth,
    mux:      mux,
    engine:   engine,
    worker:   newWorker(config, engine, common.Address{}, eth, mux),
    canStart: 1,
  }
  miner.Register(NewCpuAgent(eth.BlockChain(), engine))
  go miner.update()

  return miner
}
```

2. miner/worker.go

Onther Inc.

- worker 정의
- worker : mining 하는 독립된 machine

```
func newWorker(config *params.ChainConfig, engine consensus.Engine, coinbase common.Address, eth Backend, mux *event.TypeMux) *worker {
  worker := &worker{
    config:         config,
    engine:         engine,
    eth:            eth,
    mux:            mux,
    txCh:           make(chan core.TxPreEvent, txChanSize),
    chainHeadCh:    make(chan core.ChainHeadEvent, chainHeadChanSize),
    chainSideCh:    make(chan core.ChainSideEvent, chainSideChanSize),
    chainDb:        eth.ChainDb(),
    recv:           make(chan *Result, resultQueueSize),
    chain:          eth.BlockChain(),
    proc:           eth.BlockChain().Validator(),
    possibleUncles: make(map[common.Hash]*types.Block),
    coinbase:       coinbase,
    agents:         make(map[Agent]struct{}),
    unconfirmed:    newUnconfirmedBlocks(eth.BlockChain(), miningLogAtDepth),
  }
  // Subscribe TxPreEvent for tx pool
  worker.txSub = eth.TxPool().SubscribeTxPreEvent(worker.txCh)
  // Subscribe events for blockchain
  worker.chainHeadSub = eth.BlockChain().SubscribeChainHeadEvent(worker.chainHeadCh)
  worker.chainSideSub = eth.BlockChain().SubscribeChainSideEvent(worker.chainSideCh)
  go worker.update()

  go worker.wait()
  worker.commitNewWork()

  return worker
}
```

3. miner/agent.go

Onther Inc.

- agent : worker를 관리하는 상위 개념
- 여러개의 work을 채널로 관리

```
func NewCpuAgent(chain consensus.ChainReader, engine consensus.Engine) *CpuAgent {
  miner := &CpuAgent{
    chain:  chain,
    engine: engine,
    stop:   make(chan struct{}, 1),
    workCh: make(chan *Work, 1),
  }
  return miner
}

func (self *CpuAgent) Work() chan<- *Work            { return self.workCh }
func (self *CpuAgent) SetReturnCh(ch chan<- *Result) { self.returnCh = ch }

func (self *CpuAgent) Stop() {
  if !atomic.CompareAndSwapInt32(&self.isMining, 1, 0) {
    return // agent already stopped
  }
  self.stop <- struct{}{}
done:
  // Empty work channel
  for {
    select {
    case <-self.workCh:
    default:
      break done
    }
  }
}
```