Geth Package

Jason Hwang(황재승)
jason.h@onther.io

Index

1. core/state
2. trie
3. internal/ethapi
4. core/rawdb
5. ethdb

Onther Inc.

1. State package

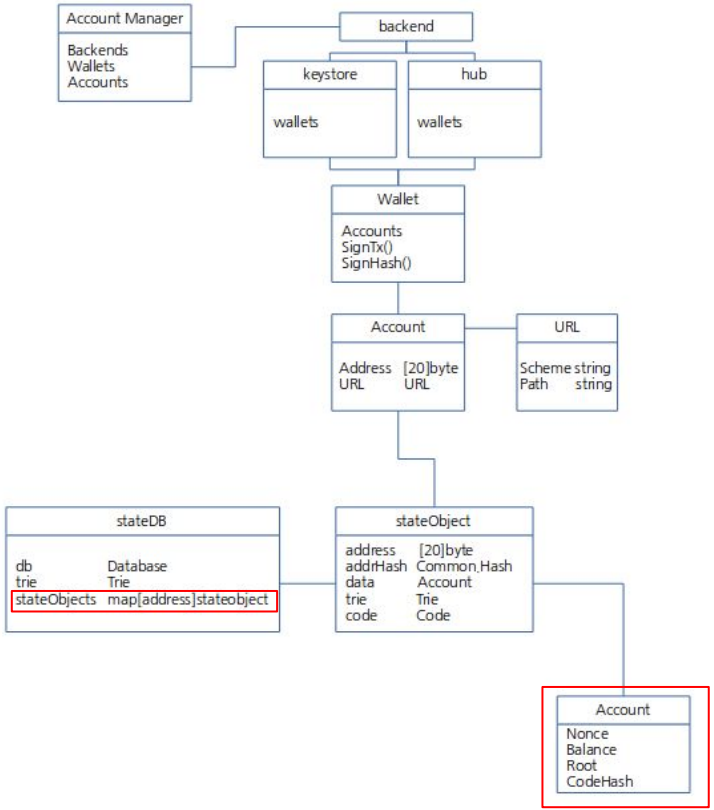
1. core/state

Onther Inc.

state

- 이더리움 블록체인상의 모든 계정 정보를 포함하는 스냅샷
- 상태 처리 - 트랜잭션 적용을 통해 어카운트를 변화시키고, 다른 상태를 생성하는 것
- 상태에 대한 관리/처리를 담당하는 것이 state 패키지
- 어카운트의 변화 상태를 state 패키지에서는 StateObject로 표현

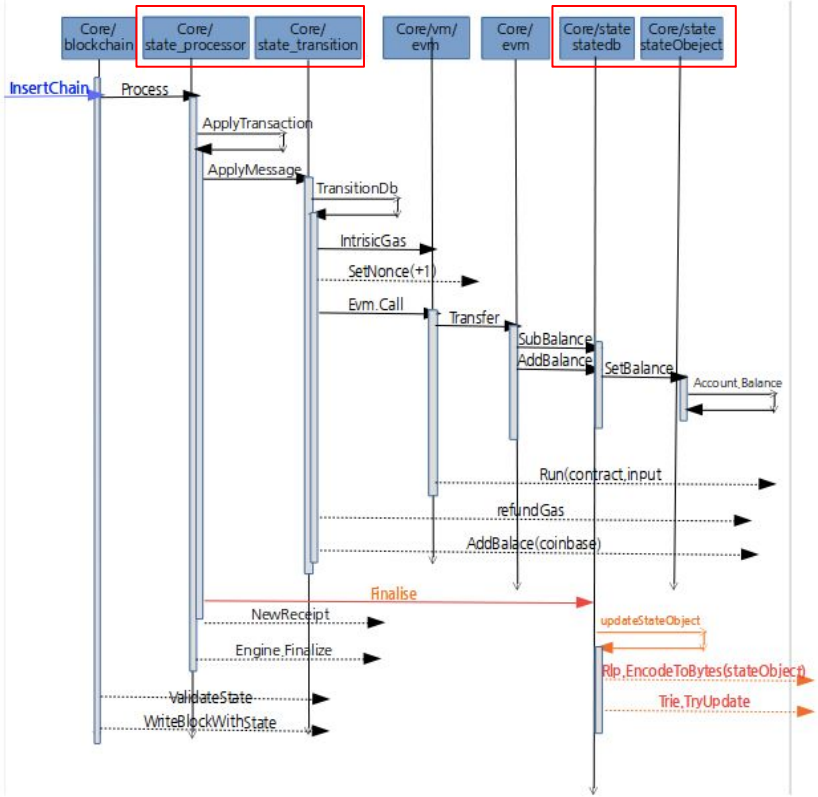
1. core/state



- 계정마다 stateObject를 갖고 있고 각 상태는 rlp 인코딩되어 stateDB에 저장
- 관리는 trie형태로
- stateDB에 stateObject를 mapping
- Account 패키지의 account 구조체는 주소와 물리적 위치만 나타냄
- state패키지의 account 구조체는 계정의 상태를 나타내는 논스, 밸런스, 루트해시, 코드해시를 갖고 있음
- state 패키지의 account 구조체는 trie형태로 저장

1. core/state

Onther Inc.



1. core/state - state_processor & transition

Onther Inc.

state_processer	상태 처리자는 기본 처리자로서, 한 지점에서 다른 지점으로의 state의 변환을 관리한다
state_transition	상태 전환은 현재 월드 상태에 대해 하나의 트렌젝션이 적용되었을 경우 발생하며 상태 전환 모델은 새로운 상태 루트를 만들기 위한 다음과 같은 일을 한다 1. 논스 핸들링 2. pre gas pay 3. receipt대한 새로운 state object 4. value 전송 만약 컨트랙트의 생성이라면 4-a) 트렌잭션을 실행하고 4-b) 제대로 실행되었을 경우 새로운 스테이트에 대한 코드로서 결과를 사용한다 스크립트 섹션을 실행하고 새로운 상태 루트를 유도한다

1. core/state - StateProcessor 구조체

Onther Inc.

```
type StateProcessor struct {
    config *params.ChainConfig // Chain configuration options
    bc     *BlockChain         // Canonical block chain
    engine consensus.Engine    // Consensus engine used for block rewards
}
```

```
func NewStateProcessor(config *params.ChainConfig, bc *BlockChain, engine consensus.Engine) *StateProcessor {
    return &StateProcessor{
        config: config,
        bc:     bc,
        engine: engine,
    }
}
```

StateProcessor 구조체, 체인 설정 옵션, 블록체인 구조체, 합의 알고리즘

newStateProcessor - 블록체인이 초기화 될 때 구성될 때 호출

1.1. core/state - Process

Onther Inc.

```go
func (p *StateProcessor) Process(block *types.Block, statedb *state.StateDB, cfg vm.Config) (types.Receipts, []*types.Log, uint64, error) {
    var (
        receipts types.Receipts
        usedGas  = new(uint64)
        header   = block.Header()
        allLogs  []*types.Log
        gp       = new(GasPool).AddGas(block.GasLimit())
    )
    // Mutate the the block and state according to any hard-fork specs
    if p.config.DAOForkSupport && p.config.DAOForkBlock != nil && p.config.DAOForkBlock.Cmp(block.Number()) == 0 {
        misc.ApplyDAOHardFork(statedb)
    }
    // Iterate over and process the individual transactions
    for i, tx := range block.Transactions() {
        statedb.Prepare(tx.Hash(), block.Hash(), i)
        receipt, _, err := ApplyTransaction(p.config, p.bc, author: nil, gp, statedb, header, tx, usedGas, cfg)
        if err != nil {
            return nil, nil, 0, err
        }
        receipts = append(receipts, receipt)
        allLogs = append(allLogs, receipt.Logs...)
    }
    // Finalize the block, applying any consensus engine specific extras (e.g. block rewards)
    p.engine.Finalize(p.bc, header, statedb, block.Transactions(), block.Uncles(), receipts)

    return receipts, allLogs, *usedGas, nil
}
```

statedb 준비

트랜잭션 결과를 적용

1.1 core/state - ApplyTransaction

Onther Inc.

```go
func ApplyTransaction(config *params.ChainConfig, bc ChainContext, author *common.Address, gp *GasPool,
    msg, err := tx.AsMessage(types.MakeSigner(config, header.Number))
    if err != nil {
        return nil, 0, err
    }
    // Create a new context to be used in the EVM environment
    context := NewEVMContext(msg, header, bc, author)
    // Create a new environment which holds all relevant information
    // about the transaction and calling mechanisms.
    vmenv := vm.NewEVM(context, statedb, config, cfg)
    // Apply the transaction to the current state (included in the env)
    _, gas, failed, err := ApplyMessage(vmenv, msg, gp)
    if err != nil {
        return nil, 0, err
    }
    // Update the state with pending changes
    var root []byte
    if config.IsByzantium(header.Number) {
        statedb.Finalise(true)
    } else {
        root = statedb.IntermediateRoot(config.IsEIP158(header.Number)).Bytes()
    }
    *usedGas += gas

    // Create a new receipt for the transaction, storing the intermediate root and gas used by the tx
    // based on the eip phase, we're passing wether the root touch-delete accounts.
    receipt := types.NewReceipt(root, failed, *usedGas)
    receipt.TxHash = tx.Hash()
    receipt.GasUsed = gas
    // if the transaction created a contract, store the creation address in the receipt.
    if msg.To() == nil {
        receipt.ContractAddress = crypto.CreateAddress(vmenv.Context.Origin, tx.Nonce())
    }
    // Set the receipt logs and create a bloom for filtering
    receipt.Logs = statedb.GetLogs(tx.Hash())
    receipt.Bloom = types.CreateBloom(types.Receipts{receipt})

    return receipt, gas, err
}
```

- ApplyTransaction 함수는 주어진 스테이트 DB에 트렌젝션을 적용하고, 인자로 전달된 파라미터를 EVM환경에서 사용하기 위한 함수
- 트렌젝션의 영수증과 가스사용량과 에러상태를 반환하며 트렌젝션이 실패할경우 블록이 검증되지 않았음을 지시한다.

VM 생성, applyMessage 함수 실행, 스테이트 디비 파이널라이즈, 가스비 반영, receipt 생성 하는 순서로 진행

1.1 core/state - ApplyTransaction

Onther Inc.

```
func ApplyTransaction(config *params.ChainConfig, bc ChainContext, author *common.Address, gp *GasPool,
    msg, err := tx.AsMessage(types.MakeSigner(config, header.Number))
    if err != nil {
        return nil, 0, err
    }
    // Create a new context to be used in the EVM environment
    context := NewEVMContext(msg, header, bc, author)
```

```
func NewEVMContext(msg Message, header *types.Header, chain ChainContext, author *common.Address) vm.Context {
    // If we don't have an explicit author (i.e. not mining), extract from the header
    var beneficiary common.Address
    if author == nil {
        beneficiary, _ = chain.Engine().Author(header) // Ignore error, we're past header validation
    } else {
        beneficiary = *author
    }
    return vm.Context{
        CanTransfer: CanTransfer,
        Transfer:    Transfer,
        GetHash:     GetHashFn(header, chain),
        Origin:      msg.From(),
        Coinbase:    beneficiary,
        BlockNumber: new(big.Int).Set(header.Number),
        Time:        new(big.Int).Set(header.Time),
        Difficulty:  new(big.Int).Set(header.Difficulty),
        GasLimit:    header.GasLimit,
        GasPrice:    new(big.Int).Set(msg.GasPrice()),
    }
}
```

```
    receipt.TxHash = tx.Hash()
    receipt.GasUsed = gas
    // if the transaction created a contract, store the creation address in the receipt.
    if msg.To() == nil {
        receipt.ContractAddress = crypto.CreateAddress(vmenv.Context.Origin, tx.Nonce())
    }
    // Set the receipt logs and create a bloom for filtering
    receipt.Logs = statedb.GetLogs(tx.Hash())
    receipt.Bloom = types.CreateBloom(types.Receipts{receipt})

    return receipt, gas, err
}
```

- ApplyTransaction 함수는 주어진 스테이트 DB에 트렌젝션을 적용하고, 인자로 전달된 파라미터를 EVM환경에서 사용하기 위한 함수
- 트렌젝션의 영수증과 가스사용량과 에러상태를 반환하며 트렌젝션이 실패할경우 블록이 검증되지 않았음을 지시한다.

NewEVMContext에서 받아온 값을 context에 저장

체인 컨텍스트는 트렌젝션 프로세싱에 사용될 헤더와 합의파라미터들을 현재의 블록체인으로부터 반환한다. NewEVMContext함수는 EVM에서 사용할 새로운 컨텍스트를 생성한다

1.1 core/state - ApplyTransaction

Onther Inc.

```go
func ApplyTransaction(config *params.ChainConfig, bc ChainContext, author *common.Address, gp *GasPool,
    msg, err := tx.AsMessage(types.MakeSigner(config, header.Number))
    if err != nil {
        return nil, 0, err
    }
    // Create a new context to be used in the EVM environment
    context := NewEVMContext(msg, header, bc, author)
    // Create a new environment which holds all relevant information
    // about the transaction and calling mechanisms.
    vmenv := vm.NewEVM(context, statedb, config, cfg)
func NewEVM(ctx Context, statedb StateDB, chainConfig *params.ChainConfig, vmConfig Config) *EVM {
    evm := &EVM{
        Context:     ctx,
        StateDB:     statedb,
        vmConfig:    vmConfig,
        chainConfig: chainConfig,
        chainRules:  chainConfig.Rules(ctx.BlockNumber),
    }

    evm.interpreter = NewInterpreter(evm, vmConfig)
    return evm
}

    // Create a new receipt for the transaction, storing the intermediate root and gas used by the tx
    // based on the eip phase, we're passing wether the root touch-delete accounts.
    receipt := types.NewReceipt(root, failed, *usedGas)
    receipt.TxHash = tx.Hash()
    receipt.GasUsed = gas
    // if the transaction created a contract, store the creation address in the receipt.
    if msg.To() == nil {
        receipt.ContractAddress = crypto.CreateAddress(vmenv.Context.Origin, tx.Nonce())
    }
    // Set the receipt logs and create a bloom for filtering
    receipt.Logs = statedb.GetLogs(tx.Hash())
    receipt.Bloom = types.CreateBloom(types.Receipts{receipt})

    return receipt, gas, err
}
```

- ApplyTransaction 함수는 주어진 스테이트 DB에 트렌젝션을 적용하고, 인자로 전달된 파라미터를 EVM환경에서 사용하기 위한 함수
- 트렌젝션의 영수증과 가스사용량과 에러상태를 반환하며 트렌젝션이 실패할경우 블록이 검증되지 않았음을 지시한다.

NewEVMContext에서 받아온 값을 context를 NewEVM의 파라미터로 전달

NewEVM은 새 EVM을 반환. 반환 된 EVM은 스레드로부터 안전하지 않으므로 한번만 사용해야 함.

1.1 core/state - ApplyTransaction

Onther Inc.

```
func ApplyTransaction(config *params.ChainConfig, bc ChainContext, author *common.Address, gp *GasPool,
    msg, err := tx.AsMessage(types.MakeSigner(config, header.Number))
    if err != nil {
        return nil, 0, err
    }
    // Create a new context to be used in the EVM environment
    context := NewEVMContext(msg, header, bc, author)
    // Create a new environment which holds all relevant information
    // about the transaction and calling mechanisms.
    vmenv := vm.NewEVM(context, statedb, config, cfg)
    // Apply the transaction to the current state (included in the env)
    _, gas, failed, err := ApplyMessage(vmenv, msg, gp)
func ApplyMessage(evm *vm.EVM, msg Message, gp *GasPool) ([]byte, uint64, bool, error) {
    return NewStateTransition(evm, msg, gp).TransitionDb()
}
    if config.IsByzantium(header.Number) {
        statedb.Finalise(true)
    } else {
        root = statedb.IntermediateRoot(config.IsEIP158(header.Number)).Bytes()
    }
    *usedGas += gas

    // Create a new receipt for the transaction, storing the intermediate root and gas used by the tx
    // based on the eip phase, we're passing wether the root touch-delete accounts.
    receipt := types.NewReceipt(root, failed, *usedGas)
    receipt.TxHash = tx.Hash()
    receipt.GasUsed = gas
    // if the transaction created a contract, store the creation address in the receipt.
    if msg.To() == nil {
        receipt.ContractAddress = crypto.CreateAddress(vmenv.Context.Origin, tx.Nonce())
    }
    // Set the receipt logs and create a bloom for filtering
    receipt.Logs = statedb.GetLogs(tx.Hash())
    receipt.Bloom = types.CreateBloom(types.Receipts{receipt})

    return receipt, gas, err
}
```

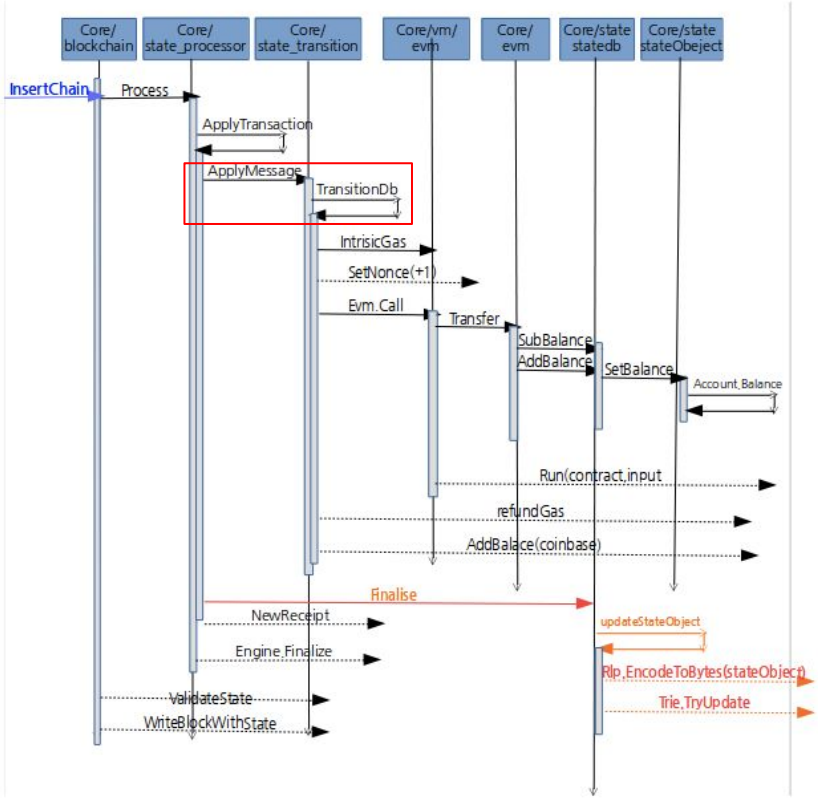
- ApplyTransaction 함수는 주어진 스테이트 DB에 트렌젝션을 적용하고, 인자로 전달된 파라미터를 EVM환경에서 사용하기 위한 함수
- 트렌젝션의 영수증과 가스사용량과 에러상태를 반환하며 트렌젝션이 실패할경우 블록이 검증되지 않았음을 지시한다.

NewEVMContext에서 받아온 값을 context를 NewEVM의 파라미터로 전달

이 함수는 주어진 환경에서 주어진 메시지를 적용함으로서 새로운 스테이트를 계산한다
이 함수는 EVM 실행에서 반환된 바이트 수와 가스 소모와 실패했을 경우 에러를 리턴한다.

1. core/state

Onther Inc.



TransitionDb (1/2)

Onther Inc.

```
func (st *StateTransition) TransitionDb() (ret []byte, usedGas uint64, failed bool, err error) {
    var (
        evm = st.evm
        // vm errors do not effect consensus and are therefor
        // not assigned to err, except for insufficient balance
        // error.
        vmerr error
    )

    msg := st.msg
    sender := vm.AccountRef(msg.From())
    homestead := st.evm.ChainConfig().IsHomestead(st.evm.BlockNumber)
    contractCreation := msg.To() == nil

    // get delegatee
    delegatee, _ := stamina.GetDelegatee(evm, msg.From())
    log.Info("GetDelegatee", "from", msg.From(), "delegatee", delegatee)
```

vm/evm에서 evm 호출

TransitionDb (1/2)

Onther Inc.

```go
func (st *StateTransition) TransitionDb() (ret []byte, usedGas uint64, failed bool, err error) {
    var (
        evm = st.evm
        // vm errors do not effect consensus and are therefor
        // not assigned to err, except for insuffici
        // error.
        vmerr error
    )

    msg := st.msg
    sender := vm.AccountRef(msg.From())
    homestead := st.evm.ChainConfig().IsHomestead(st
    contractCreation := msg.To() == nil

    // get delegatee
    delegatee, _ := stamina.GetDelegatee(evm, msg.Fr
    log.Info("GetDelegatee", "from", msg.From(), "de
```

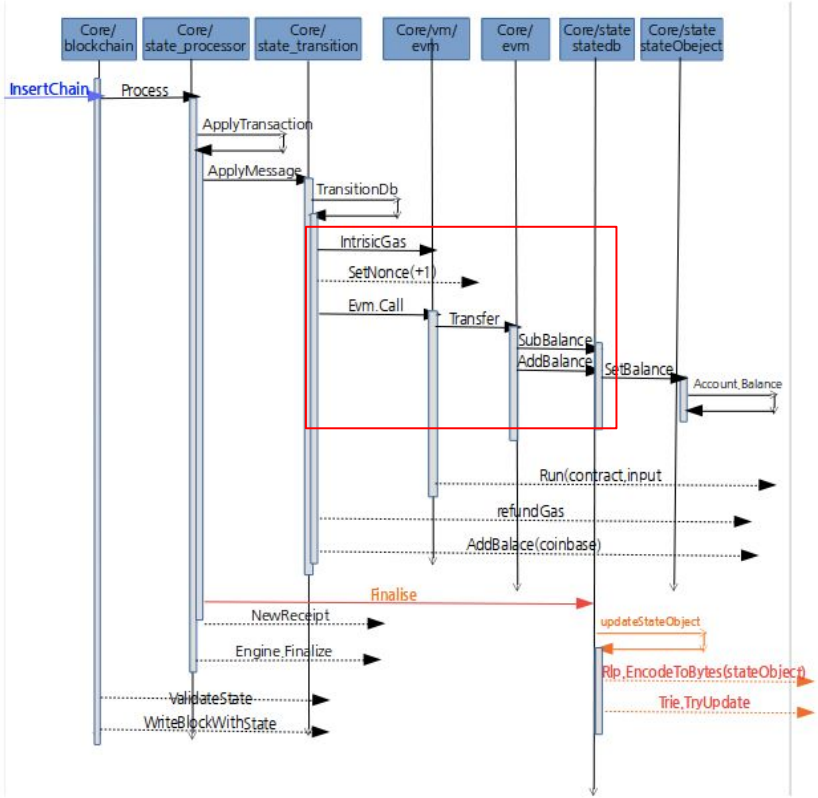
```go
type Message interface {
    From() common.Address
    //FromFrontier() (common.Address, error)
    To() *common.Address

    GasPrice() *big.Int
    Gas() uint64
    Value() *big.Int

    Nonce() uint64
    CheckNonce() bool
    Data() []byte
}
```

vm/evm에서 evm 호출

msg.To() 가 nil이면 contractCreation

1. core/state

Onther Inc.



Transitiondb(2/2)

Onther Inc.

```
if delegatee != common.HexToAddress("0x00") {
    if err = st.preDelegateeCheck(delegatee); err != nil {
        return
    }

    // Pay intrinsic gas
    gas, err := IntrinsicGas(st.data, contractCreation, homestead)
    if err != nil {
        return  ret:
    }
    if err = st.use
        return  ret:
    }

    if contractCrea
        ret, _, st.
    } else {
        // Increment
        st.state.Se
        ret, st.gas
    }
    if vmerr != nil
        log.Debug("
        // The only
        // sufficie
        // balance
        if vmerr ==
            return
        }
    }
    st.refundDelega
    // TODO: gas fe
    st.state.AddBal
                                                                    , st.gasPrice))

    return ret, st.
}
```

```
func IntrinsicGas(data []byte, contractCreation, homestead bool) (uint64, error) {
    // Set the starting gas for the raw transaction
    var gas uint64
    if contractCreation && homestead {
        gas = params.TxGasContractCreation
    } else {
        gas = params.TxGas
    }
    // Bump the required gas by the amount of transactional data
    if len(data) > 0 {
        // Zero and non-zero bytes are priced differently
        var nz uint64
        for _, byt := range data {
            if byt != 0 {
                nz++
            }
        }
        // Make sure we don't exceed uint64 for all data combinations
        if (math.MaxUint64-gas)/params.TxDataNonZeroGas < nz {
            return 0, vm.ErrOutOfGas
        }
        gas += nz * params.TxDataNonZeroGas

        z := uint64(len(data)) - nz
        if (math.MaxUint64-gas)/params.TxDataZeroGas < z {
            return 0, vm.ErrOutOfGas
        }
        gas += z * params.TxDataZeroGas
    }
    return gas, nil
```

이 함수는 현재 메시지를 적용하여 스테이트를 변화하고 사용된 가스와 함께 결과를 리턴한다

IntrinsicGas를 계산하는 과정

Transitiondb(2/2)

Onther Inc.

```
if delegatee != common.HexToAddress("0x00") {
    if err = st.preDelegateeCheck(delegatee); err != nil {
        return
    }

    // Pay intrinsic gas
    gas, err := IntrinsicGas(st.data, contractCreation, homestead)
    if err != nil {
        return ret: nil, usedGas: 0, failed: false, err
    }
    if err = st.useGas(gas); err != nil {
        return ret: nil, usedGas: 0, failed: false, err
    }

    if contractCreation {
        ret, _, st.gas, vmerr = evm.Create(sender, st.data, st.gas, st.value)
    } else {
        // Increment the nonce for the next transaction
        st.state.SetNonce(msg.From(), st.state.GetNonce(sender.Address())+1)
        ret, st.gas, vmerr = evm.Call(sender, st.to(), st.data, st.gas, st.value)
    }
    if vmerr != nil {
        log.Debug("VM returned with error", "err", vmerr)
        // The only possible consensus-error would be if there wasn't
        // sufficient balance to make the transfer happen. The first
        // balance transfer may never fail.
        if vmerr == vm.ErrInsufficientBalance {
            return ret: nil, usedGas: 0, failed: false, vmerr
        }
    }
    st.refundDelegateeGas(delegatee)
    // TODO: gas fee to miner
    st.state.AddBalance(st.evm.Coinbase, new(big.Int).Mul(new(big.Int).SetUint64(st.gasUsed()), st.gasPrice))

    return ret, st.gasUsed(), vmerr != nil, err
```

이 함수는 현재 메시지를 적용하여 스테이트를 변화하고 사용된 가스와 함께 결과를 리턴한다

Transitiondb(2/2)

Onther Inc.

```
if delegatee != common.HexToAddress("0x00") {
    if err = st.preDelegateeCheck(delegatee); err != nil {
        return
    }

    // Pay intrinsic gas
    gas, err := IntrinsicGas(st.data, contractCreation, homestead)
    if err != nil {
        return ret: nil, usedGas: 0, failed: false, err
    }
    if err = st.useGas(gas); err != nil {
        return ret: nil, usedGas: 0, failed: false, err
    }

    if contractCreation {
        ret, _, st.gas, vmerr = evm.Create(sender, st.data, st.gas, st.value)
    } else {
        // Increment the nonce for the next transaction
        st.state.SetNonce(msg.From(), st.state.GetNonce(sender.Address())+1)
        ret, st.gas, vmerr = evm.Call(sender, st.to(), st.data, st.gas, st.value)
    }
    if vmerr != nil {
        log.Debug("VM returned with error", "err", vmerr)
        // The only possible consensus-error would be if there wasn't
        // sufficient balance to make the transfer happen. The first
        // balance transfer may never fail.
        if vmerr == vm.ErrInsufficientBalance {
            return ret: nil, usedGas: 0, failed: false, vmerr
        }
    }
    st.refundDelegateeGas(delegatee)
    // TODO: gas fee to miner
    st.state.AddBalance(st.evm.Coinbase, new(big.Int).Mul(new(big.Int).SetUint64(st.gasUsed()), st.gasPrice))

    return ret, st.gasUsed(), vmerr != nil, err
```

이 함수는 현재 메시지를 적용하여 스테이트를 변화하고 사용된 가스와 함께 결과를 리턴한다

논스 세팅과 evm.call 호출

evm.Call

Onther Inc.

```
func (evm *EVM) Call(caller ContractRef, addr common.Address, input []byte, gas uint64, value *big.Int) (ret [
    if evm.vmConfig.NoRecursion && evm.depth > 0 {
        return ret: nil, gas, err: nil
    }

    // Fail if we're trying to execute above the call depth limit
    if evm.depth > int(params.CallCreateDepth) {
        return ret: nil, gas, ErrDepth
    }
    // Fail if we're trying to transfer more than the available balance
    if !evm.Context.CanTransfer(evm.StateDB, caller.Address(), value) {
        return ret: nil, gas, ErrInsufficientBalance
    }

    var (
        to       = AccountRef(addr)
        snapshot = evm.StateDB.Snapshot()
    )
    if !evm.StateDB.Exist(addr) {
        precompiles := PrecompiledContractsHomestead
        if evm.ChainConfig().IsByzantium(evm.BlockNumber) {
            precompiles = PrecompiledContractsByzantium
        }
        if precompiles[addr] == nil && evm.ChainConfig().IsEIP158(evm.BlockNumber) && value.Sign() == 0 {
            // Calling a non existing account, don't do antything, but ping the tracer
            if evm.vmConfig.Debug && evm.depth == 0 {
                evm.vmConfig.Tracer.CaptureStart(caller.Address(), addr, call: false, input, gas, value)
                evm.vmConfig.Tracer.CaptureEnd(ret, gasUsed: 0, t: 0, err: nil)
            }
            return ret: nil, gas, err: nil
        }
        evm.StateDB.CreateAccount(addr)
    }
    evm.Transfer(evm.StateDB, caller.Address(), to.Address(), value)
```

```
func Transfer(db vm.StateDB, sender, recipient common.Address, amount *big.Int) {
    db.SubBalance(sender, amount)
    db.AddBalance(recipient, amount)
}
```

evm.call에서 StateDB를 호출하는 과정

core/state/statedb.go - Add, Sub, SetBalance

Onther Inc.

```go
func (self *StateDB) AddBalance(addr common.Address, amount *big.Int) {
    stateObject := self.GetOrNewStateObject(addr)
    if stateObject != nil {
        stateObject.AddBalance(amount)
    }
}

// SubBalance subtracts amount from the account associated with addr.
func (self *StateDB) SubBalance(addr common.Address, amount *big.Int) {
    stateObject := self.GetOrNewStateObject(addr)
    if stateObject != nil {
        stateObject.SubBalance(amount)
    }
}

func (self *StateDB) SetBalance(addr common.Address, amount *big.Int) {
    stateObject := self.GetOrNewStateObject(addr)
```

```go
func (self *StateDB) GetOrNewStateObject(addr common.Address) *stateObject {
    stateObject := self.getStateObject(addr)
    if stateObject == nil || stateObject.deleted {
        stateObject, _ = self.createObject(addr)
    }
    return stateObject
}
```

Balance와 관련된 함수들을 호출하면 stateObject와 관련된 함수를 호출

GetOrNewStateObject 함수는 state를 가져오거나 state가 없으면 새 state를 생성하는 역할을 담당

func getStateObject()

Onther Inc.

```go
func (self *StateDB) getStateObject(addr common.Address) (stateObject *stateObject) {
    // Prefer 'live' objects.
    if obj := self.stateObjects[addr]; obj != nil {
        if obj.deleted {
            return stateObject: nil
        }
        return obj
    }

    // Load the object from the database.
    enc, err := self.trie.TryGet(addr[:])
    if len(enc) == 0 {
        self.setError(err)
        return stateObject: nil
    }
    var data Account
    if err := rlp.DecodeBytes(enc, &data); err != nil {
        log.Error("Failed to decode state object", "addr", addr, "err", err)
        return stateObject: nil
    }
    // Insert into the live set.
    obj := newObject(self, addr, data)
    self.setStateObject(obj)
    return obj
}
```

Transitiondb(2/2)

Onther Inc.

```
type StateDB interface {
    CreateAccount(common.Address)

    SubBalance(common.Address, *big.Int)
    AddBalance(common.Address, *big.Int)
    GetBalance(common.Address) *big.Int

    GetNonce(common.Address) uint64
    SetNonce(common.Address, uint64)

    GetCodeHash(co
    GetCode(common
    SetCode(common
    GetCodeSize(co

    AddRefund(uint
    GetRefund() ui

    GetState(commo
    SetState(commo

    Suicide(common
    HasSuicided(co

    // Exist repor
    // Notably thi
    Exist(common.A
    // Empty retur
    // is defined
    Empty(common.A

    RevertToSnapsh
    Snapshot() int

    AddLog(*types.
    AddPreimage(co

    ForEachStorage
}
```

```
func (self *StateDB) AddBalance(addr common.Address, amount *big.Int) {
    stateObject := self.GetOrNewStateObject(addr)
    if stateObject != nil {
        stateObject.AddBalance(amount)
    }
}

// SubBalance subtracts amount from the account associated with addr.
func (self *StateDB) SubBalance(addr common.Address, amount *big.Int) {
    stateObject := self.GetOrNewStateObject(addr)
    if stateObject != nil {
        stateObject.SubBalance(amount)
    }
}

func (self *StateDB) SetBalance(addr common.Address, amount *big.Int) {
    stateObject := self.GetOrNewStateObject(addr)
    if stateObject != nil {
        stateObject.SetBalance(amount)
    }
}

func (self *StateDB) SetNonce(addr common.Address, nonce uint64) {
    stateObject := self.GetOrNewStateObject(addr)
    if stateObject != nil {
        stateObject.SetNonce(nonce)
    }
}
```

core/vm/interface.go에 statedb 인터페이스

core/state/statedb에 인터페이스에 대한 함수들 구현

1.1 core/state - ApplyTransaction

Onther Inc.

```go
func ApplyTransaction(config *params.ChainConfig, bc ChainContext, author *common.Address, gp *GasPool,
    msg, err := tx.AsMessage(types.MakeSigner(config, header.Number))
    if err != nil {
        return nil, 0, err
    }
    // Create a new context to be used in the EVM environment
    context := NewEVMContext(msg, header, bc, author)
    // Create a new environment which holds all relevant information
    // about the transaction and calling mechanisms.
    vmenv := vm.NewEVM(context, statedb, config, cfg)
    // Apply the transaction to the current state (included in the env)
    _, gas, failed, err := ApplyMessage(vmenv, msg, gp)
```

```go
func ApplyMessage(evm *vm.EVM, msg Message, gp *GasPool) ([]byte, uint64, bool, error) {
    return NewStateTransition(evm, msg, gp).TransitionDb()
}
```

```go
func NewStateTransition(evm *vm.EVM, msg Message, gp *GasPool) *StateTransition {
    return &StateTransition{
        gp:       gp,
        evm:      evm,
        msg:      msg,
        gasPrice: msg.GasPrice(),
        value:    msg.Value(),
        data:     msg.Data(),
        state:    evm.StateDB,
    }
}
```

```go
        receipt.ContractAddress = crypto.CreateAddress(vmenv.Context.Origin, tx.Nonce())
    }
    // Set the receipt logs and create a bloom for filtering
    receipt.Logs = statedb.GetLogs(tx.Hash())
    receipt.Bloom = types.CreateBloom(types.Receipts{receipt})

    return receipt, gas, err
}
```

- ApplyTransaction 함수는 주어진 스테이트 DB에 트렌젝션을 적용하고, 인자로 전달된 파라미터를 EVM환경에서 사용하기 위한 함수
- 트렌젝션의 영수증과 가스사용량과 에러상태를 반환하며 트렌젝션이 실패할경우 블록이 검증되지 않았음을 지시한다.

NewEVMContext에서 받아온 값을 context를 NewEVM의 파라미터로 전달

ApplyMessage 함수는 주어진 환경에서 주어진 메시지를 적용함으로서 새로운 스테이트를 계산한다
ApplyMessage 함수는 EVM 실행에서 반환된 바이트 수와 가스 소모와 실패했을 경우 에러를 리턴한다.

NewStateTransition 함수는 새로운 상태 오브젝트를 생성하고 반환

Transitiondb(2/2)

Onther Inc.

```
if delegatee != common.HexToAddress("0x00") {
    if err = st.preDelegateeCheck(delegatee); err != nil {
        return
    }

    // Pay intrinsic gas
    gas, err := IntrinsicGas(st.data, contractCreation, homestead)
    if err != nil {
        return ret: nil, usedGas: 0, failed: false, err
    }
    if err = st.useGas(gas); err != nil {
        return ret: nil, usedGas: 0, failed: false, err
    }

    if contractCreation {
        ret, _, st.gas, vmerr = evm.Create(sender, st.data, st.gas, st.value)
    } else {
        // Increment the nonce for the next transaction
        st.state.SetNonce(msg.From(), st.state.GetNonce(sender.Address())+1)
        ret, st.gas, vmerr = evm.Call(sender, st.to(), st.data, st.gas, st.value)
    }
    if vmerr != nil {
        log.Debug("VM returned with error", "err", vmerr)
        // The only possible consensus-error would be if there wasn't
        // sufficient balance to make the transfer happen. The first
        // balance transfer may never fail.
        if vmerr == vm.ErrInsufficientBalance {
            return ret: nil, usedGas: 0, failed: false, vmerr
        }
    }
    st.refundDelegateeGas(delegatee)
    // TODO: gas fee to miner
    st.state.AddBalance(st.evm.Coinbase, new(big.Int).Mul(new(big.Int).SetUint64(st.gasUsed()), st.gasPrice))

    return ret, st.gasUsed(), vmerr != nil, err
```

이 함수는 현재 메시지를 적용하여 스테이트를 변화하고 사용된 가스와 함께 결과를 리턴한다

가스비를 채굴자에게 지급

1.1 core/state - ApplyTransaction

Onther Inc.

```
func ApplyTransaction(config *params.ChainConfig, bc ChainContext, author *common.Address, gp *GasPool,
    msg, err := tx.AsMessage(types.MakeSigner(config, header.Number))
    if err != nil {
        return nil, 0, err
    }
    // Create a new context to be used in the EVM environment
    context := NewEVMContext(msg, header, bc, author)
    // Create a new environment which holds all relevant information
    // about the transaction and calling mechanisms.
    vmenv := vm.NewEVM(context, statedb, config, cfg)
    // Apply the transaction to the current state (included in the env)
    _, gas, failed, err := ApplyMessage(vmenv, msg, gp)
    if err != nil {
        return nil, 0, err
    }
    // Update the state with pending changes
    var root []byte
    if config.IsByzantium(header.Number) {
        statedb.Finalise(true)
    } else {
        root = statedb.IntermediateRoot(config.IsEIP158(header.Number)).Bytes()
    }
    *usedGas += gas
```

```
func (s *StateDB) IntermediateRoot(deleteEmptyObjects bool) common.Hash {
    s.Finalise(deleteEmptyObjects)
    return s.trie.Hash()
}
```

```
    // if the transaction created a contract, store the creation address in the receipt.
    if msg.To() == nil {
        receipt.ContractAddress = crypto.CreateAddress(vmenv.Context.Origin, tx.Nonce())
    }
    // Set the receipt logs and create a bloom for filtering
    receipt.Logs = statedb.GetLogs(tx.Hash())
    receipt.Bloom = types.CreateBloom(types.Receipts{receipt})

    return receipt, gas, err
}
```

- ApplyTransaction 함수는 주어진 스테이트 DB에 트렌젝션을 적용하고, 인자로 전달된 파라미터를 EVM환경에서 사용하기 위한 함수
- 트렌젝션의 영수증과 가스사용량과 에러상태를 반환하며 트렌젝션이 실패할경우 블록이 검증되지 않았음을 지시한다.

비잔티움인지 확인하고 비잔티움이면 상태 디비를 finalize,
그렇지 않으면 statedb.IntermediateRoot 함수를 호출,
statedb.IntermediateRoot는 stateDB의 trie 해시, 즉 루트를 반환

이 과정이 끝나면 ApplyMessage에서 리턴 받은 가스를 usedGas에 추가

StateDB.Finalise

Onther Inc.

```go
func (s *StateDB) Finalise(deleteEmptyObjects bool) {
    for addr := range s.journal.dirties {
        stateObject, exist := s.stateObjects[addr]
        if !exist {
            // ripeMD
            // That
            // touch
            // it wi
            // it ma
            // Thus,
            continue
        }

        if stateObje
            s.delete
        } else {
            stateObj
            s.update
        }
        s.stateObject
    }
    // Invalidate jo
    s.clearJournalAn
}
```

```go
type StateDB struct {
    db   Database
    trie Trie

    // This map holds 'live' objects, which will get modified while processing a state transition.
    stateObjects      map[common.Address]*stateObject
    stateObjectsDirty map[common.Address]struct{}

    // DB error.
    // State objects are used by the consensus core a
    // unable to deal with database-level errors. Any
    // during a database read is memoized here and wi
    // by StateDB.Commit.
    dbErr error

    // The refund counter, also used by state transit
    refund uint64

    thash, bhash common.Hash
    txIndex      int
    logs         map[common.Hash][]*types.Log
    logSize      uint

    preimages map[common.Hash][]byte

    // Journal of state modifications. This is the ba
    // Snapshot and RevertToSnapshot.
    journal        *journal
    validRevisions []revision
    nextRevisionId int

    lock sync.Mutex
}
```

```go
type stateObject struct {
    address  common.Address
    addrHash common.Hash // hash of ethereum address of the account
    data     Account
    db       *StateDB

    // DB error.
    // State objects are used by the consensus core and VM which are
    // unable to deal with database-level errors. Any error that occurs
    // during a database read is memoized here and will eventually be returned
    // by StateDB.Commit.
    dbErr error

    // Write caches.
    trie Trie // storage trie, which becomes non-nil on first access
    code Code // contract bytecode, which gets set when code is loaded

    cachedStorage Storage // Storage entry cache to avoid duplicate reads
    dirtyStorage  Storage // Storage entries that need to be flushed to disk

    // Cache flags.
    // When an object is marked suicided it will be delete from the trie
    // during the "update" phase of the state transition.
    dirtyCode bool // true if the code was updated
    suicided  bool
    deleted   bool
}
```

StateDB.Finalise

Onther Inc.

```go
func (s *StateDB) Finalise(deleteEmptyObjects bool) {
    for addr := range s.journal.dirties {
        stateObject, exist := s.stateObjects[addr]
        if !exist {
            // ripeMD is 'touched' at block 1714175, in tx 0x1237f737031e40bcde4a8b7e717b2d15e3ecadfe49bb1bbc71ee9deb09c6fcf2
            // That tx goes out of gas, and although the notion of 'touched' does not exist there, the
            // touch-event will still be recorded in the journal. Since ripeMD is a special snowflake,
            // it will persist in the journal even though the journal is reverted. In this special circumstance,
            // it may exist in `s.journal.dirties` but not in `s.stateObjects`.
            // Thus, we can safely ignore it here
            continue
        }

        if stateObject.suicided || (deleteEmpt
            s.deleteStateObject(stateObject)
        } else {
            stateObject.updateRoot(s.db)
            s.updateStateObject(stateObject)
        }
        s.stateObjectsDirty[addr] = struct{}{}
    }
    // Invalidate journal because reverting ac
    s.clearJournalAndRefund()
}
```

```go
func (self *stateObject) updateRoot(db Database) {
    self.updateTrie(db)
    self.data.Root = self.trie.Hash()
}
```

```go
func (self *StateDB) updateStateObject(stateObject *stateObject) {
    addr := stateObject.Address()
    data, err := rlp.EncodeToBytes(stateObject)
    if err != nil {
        panic(fmt.Errorf( format: "can't encode object at %x: %v", addr[:], err))
    }
    self.setError(self.trie.TryUpdate(addr[:], data))
}
```

stateObject를 rlp 인코딩한 데이터를 trie에 업데이트

1.1 core/state - ApplyTransaction

Onther Inc.

```go
func ApplyTransaction(config *params.ChainConfig, bc ChainContext, author *common.Address, gp *GasPool,
    msg, err := tx.AsMessage(types.MakeSigner(config, header.Number))
    if err != nil {
        return nil, 0, err
    }
    // Create a new context to be used in the EVM environment
    context := NewEVMContext(msg, header, bc, author)
    // Create a new environment which holds all relevant information
    // about the transaction and calling mechanisms.
    vmenv := vm.NewEVM(context, statedb, config, cfg)
    // Apply the transaction to the current state (included in the env)
    _, gas, failed, err := ApplyMessage(vmenv, msg, gp)
    if err != nil {
        return nil, 0, err
    }
    // Update the state with pending changes
    var root []byte
    if config.IsByzantium(header.Number) {
        statedb.Finalise(true)
    } else {
        root = statedb.IntermediateRoot(config.IsEIP158(header.Number)).Bytes()
    }
    *usedGas += gas

    // Create a new receipt for the transaction, storing the intermediate root and gas used by the tx
    // based on the eip phase, we're passing wether the root touch-delete accounts.
    receipt := types.NewReceipt(root, failed, *usedGas)
    receipt.TxHash = tx.Hash()
    receipt.GasUsed = gas
    // if the transaction created a contract, store the creation address in the receipt.
    if msg.To() == nil {
        receipt.ContractAddress = crypto.CreateAddress(vmenv.Context.Origin, tx.Nonce())
    }
    // Set the receipt logs and create a bloom for filtering
    receipt.Logs = statedb.GetLogs(tx.Hash())
    receipt.Bloom = types.CreateBloom(types.Receipts{receipt})

    return receipt, gas, err
}
```

- ApplyTransaction 함수는 주어진 스테이트 DB에 트렌젝션을 적용하고, 인자로 전달된 파라미터를 EVM환경에서 사용하기 위한 함수
- 트렌젝션의 영수증과 가스사용량과 에러상태를 반환하며 트렌젝션이 실패할경우 블록이 검증되지 않았음을 지시한다.

스테이트의 변경 내역을 receipt에 반영

1.1. core/state - Process

Onther Inc.

```go
func (p *StateProcessor) Process(block *types.Block, statedb *state.StateDB, cfg vm.Config) (types.Receipts, []*types.Log, uint64, error) {
    var (
        receipts types.Receipts
        usedGas  = new(uint64)
        header   = block.Header()
        allLogs  []*types.Log
        gp       = new(GasPool).AddGas(block.GasLimit())
    )
    // Mutate the the block and state according to any hard-fork specs
    if p.config.DAOForkSupport && p.config.DAOForkBlock != nil && p.config.DAOForkBlock.Cmp(block.Number()) == 0 {
        misc.ApplyDAOHardFork(statedb)
    }
    // Iterate over and process the individual transactions
    for i, tx := range block.Transactions() {
        statedb.Prepare(tx.Hash(), block.Hash(), i)
        receipt, _, err := ApplyTransaction(p.config, p.bc, author: nil, gp, statedb, header, tx, usedGas, cfg)
        if err != nil {
            return nil, nil, 0, err
        }
        receipts = append(receipts, receipt)
        allLogs = append(allLogs, receipt.Logs...)
    }
    // Finalize the block, applying any consensus engine specific extras (e.g. block rewards)
    p.engine.Finalize(p.bc, header, statedb, block.Transactions(), block.Uncles(), receipts)

    return receipts, allLogs, *usedGas, nil
}
```

트랜잭션 결과들은 적용

1.1. core/state - Process

Onther Inc.

```go
func (p *StateProcessor) Process(block *types.Block, statedb *state.StateDB, cfg vm.Config) (types.Receipts, []*types.Log, uint64, error) {
    var (
        receipts types.Receipts
        usedGas  = new(uint64)
        header   = block.Header()
        allLogs  []*types.Log
        gp       = new(GasPool).AddGas(block.GasLimit())
    )
    // Mutate the the block and state according to any hard-fork specs
    if p.config.DAOForkSupport && p.config.DAOForkBlock != nil && p.config.DAOForkBlock.Cmp(block.Number()) == 0 {
        misc.ApplyDAOHardFork(statedb)
    }
    // Iterate over and process the individual transactions
    for i, tx := range block.Transactions() {
        statedb.Prepare(tx.Hash(), block.Hash(), i)
        receipt, _, err := ApplyTransaction(p.config, p.bc,  author: nil, gp, statedb, header, tx, usedGas, cfg)
        if err != nil {
            return nil, nil, 0, err
        }
        receipts = append(receipts, receipt)
        allLogs = append(allLogs, receipt.Logs...)
    }
    // Finalize the block, applying any consensus engine specific extras (e.g. block rewards)
    p.engine.Finalize(p.bc, header, statedb, block.Transactions(), block.Uncles(), receipts)

    return receipts, allLogs, *usedGas, nil
}
```

트랜잭션 결과들은 적용

Onther Inc.

2. Trie

2. trie

Onther Inc.

```
func (self *stateObject) updateTrie(db Database) Trie {
    tr := self.getTrie(db)
    for key, value := range self.dirtyStorage {
        delete(self.dirtyStorage, key)
        if (value == common.Hash{}) {
            self.setError(tr.TryDelete(key[:]))
            continue
        }
        // Encoding []byte cannot fail, ok to ignore the error.
        v, _ := rlp.EncodeToBytes(bytes.TrimLeft(value[:], cutset: "\x00"))
        self.setError(tr.TryUpdate(key[:], v))
    }
    return tr
}
```

스테이트가 파이널라이즈 되는 과정에서 호출되는 updateTrie
데이터베이스의 변경 내역을 업데이트하고 trie를 리턴한다

2. trie

Onther Inc.

```
func (self *stateObject) updateTrie(
    tr := self.getTrie(db)
    for key, value := range self.dir
        delete(self.dirtyStorage, ke
        if (value == common.Hash{})
            self.setError(tr.TryDele
            continue
        }
        // Encoding []byte cannot fa
        v, _ := rlp.EncodeToBytes(by
        self.setError(tr.TryUpdate(k
    }
    return tr
}
```

```
type stateObject struct {
    address  common.Address
    addrHash common.Hash // hash of ethereum address of the account
    data     Account
    db       *StateDB

    // DB error.
    // State objects are used by the consensus core and VM which are
    // unable to deal with database-level errors. Any error that occurs
    // during a database read is memoized here and will eventually be returned
    // by StateDB.Commit.
    dbErr error

    // Write caches.
    trie Trie // storage trie, which becomes non-nil on first access
    code Code // contract bytecode, which gets set when code is loaded

    cachedStorage Storage // Storage entry cache to avoid duplicate reads
    dirtyStorage  Storage // Storage entries that need to be flushed to disk

    // Cache flags.
    // When an object is marked suicided it will be delete from the trie
    // during the "update" phase of the state transition.
    dirtyCode bool // true if the code was updated
    suicided  bool
    deleted   bool
}
```

2. trie

Onther Inc.

```
func (self *stateObject) updateTrie(db Database) Trie {
    tr := self.getTrie(db)
    func (c *stateObject) getTrie(db Database) Trie {
        if c.trie == nil {
            var err error
            c.trie, err = db.OpenStorageTrie(c.addrHash, c.data.Root)
            if err != nil {
                c.trie, _ = db.OpenStorageTrie(c.addrHash, common.Hash{})
                c.setError(fmt.Errorf( format: "can't create storage trie: %v", err))
            }
        }
        return c.trie
    }
}
```

trie를 업데이트하기 위해 일단 getTrie 함수를 이용해 trie 호출

stateObject의 trie를 호출해서 c.trie가 nil이면 OpenStorageTrie를 호출, OpenStorageTrie는 계정의 storage trie를 오픈함

2. trie

Onther Inc.

```
func (self *stateObject) updateTrie(db Database) Trie {
    tr := self.getTrie(db)
    for key, value := range self.dirtyStorage {
        delete(self.dirtyStorage, key)
        if (value == common.Hash{}) {
            self.setError(tr.TryDelete(key[:]))
            continue
        }
        // Encoding []byte cannot fail, ok to ignore the error.
        v, _ := rlp.EncodeToBytes(bytes.TrimLeft(value[:], cutset: "\x00"))
        self.setError(tr.TryUpdate(key[:], v))
    }
    return tr
}
```

2. trie

Onther Inc.

```
func (t *Trie) TryUpdate(key, value []byte) error {
    k := keybytesToHex(key)
    if len(value) != 0 {
        _, n, err := t.insert(t.root, prefix: nil, k, valueNode(value))
        if err != nil {
            return err
        }
        t.root = n
    } else {
        _, n, err := t.delete(t.root, prefix: nil, k)
        if err != nil {
            return err
        }
        t.root = n
    }
    return nil
}
```

2. trie

Onther Inc.

```
func (t *Trie) TryUpdate(key, value []byte) error {
    k := keybytesToHex(key)
    if len(va
        _, n,
        if er
            r
        }
        t.roo
    } else {
        _, n,
        if er
            r
        }
        t.roo
    }
    return ni
}
```

```
func keybytesToHex(str []byte) []byte {
    l := len(str)*2 + 1
    var nibbles = make([]byte, l)
    for i, b := range str {
        nibbles[i*2] = b / 16
        nibbles[i*2+1] = b % 16
    }
    nibbles[l-1] = 16
    return nibbles
}
```

2. trie

Onther Inc.

```
func (t *Trie) TryUpdate(key, value []byte) error {
    k := keybytesToHex(key)
    if len(value) != 0 {
        _, n, err := t.insert(t.root, prefix: nil, k, valueNode(value))
    } e
        t.root = n
    }
    return nil
}
```

```
func (t *Trie) insert(n node, prefix, key []byte, value node) (bool, node, error) {
    if len(key) == 0 {
        if v, ok := n.(valueNode); ok {
            return !bytes.Equal(v, value.(valueNode)), value, nil
        }
        return true, value, nil
    }
    switch n := n.(type) {
```

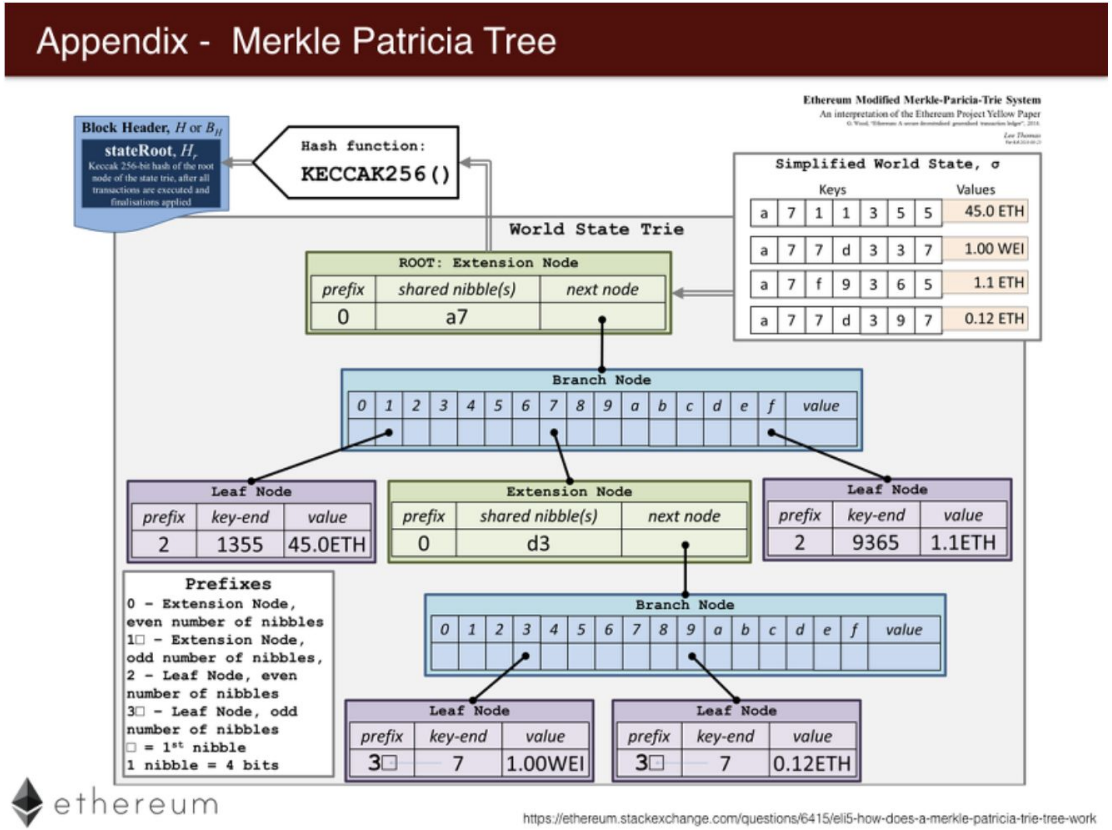
insert 함수는 노드의 유형을 확인해서 그 유형에 맞게 노드를 삽입시킴, 노드는 세가지 종류가 있음

- shortNode는 키/값쌍 하나를 가진 형태
- full은 자식노드를 17개까지 가지는 형태
- value node는 데이터 그자체만을 가진 노드

shortNode는 트라이에 노드가 딱 하나만 있을때 사용하고, 이후부터는 fullNode를 사용, value node는 full node의 맨끝에 leaf로서 붙는 값 자체이기 때문에 사실은 2가지 타입이라고 봐도 무방

2. trie

Onther Inc.



https://ethereum.stackexchange.com/questions/6415/eli5-how-does-a-merkle-patricia-trie-tree-work

Extension node는 short node

Branch Node 는 Fullnode,

Leaf node는 value node

ethereum

2. trie

Onther Inc.

```
switch n := n.(type) {
case *shortNode:
    matchlen := prefixLen(key, n.Key)
    // If the whole key matches, keep this short node as is
    // and only update the value.
    if matchlen == len(n.Key) {
        dirty, nn, err := t.insert(n.Val, append(prefix, key[:matchlen]...), key[matchlen:], value)
        if !dirty || err != nil {
            return false, n, err
        }
        return true, &shortNode{ Key: n.Key,  Val: nn,  flags: t.newFlag()}, nil
    }
    // Otherwise branch out at the index where they differ.
    branch := &fullNode{flags: t.newFlag()}
    var err error
    _, branch.Children[n.Key[matchlen]], err = t.insert( n: nil, append(prefix, n.Key[:matchlen+1]...), n.Key[matchlen+1:], n.Val)
    if err != nil {
        return false, nil, err
    }
    _, branch.Children[key[matchlen]], err = t.insert( n: nil, append(prefix, key[:matchlen+1]...), key[matchlen+1:], value)
    if err != nil {
        return false, nil, err
    }
    // Replace this shortNode with the branch if it occurs at index 0.
    if matchlen == 0 {
        return true, branch, nil
    }
    // Otherwise, replace it with a short node leading up to the branch.
    return true, &shortNode{ Key: key[:matchlen],  Val: branch,  flags: t.newFlag()}, nil
```

short node일 경우

2. trie

Onther Inc.

```
switch n := n.(type) {
case *shortNode:
    matchlen := prefixLen(key, n.Key)
    // If the
    // and or
    if match
        dirty
        if !d
        }
        retu
    }
    // Other
    branch :=
    var err
    _, branch
    if err !=
        retu
    }
    _, branch
    if err !=
        retu
    }
    // Replac
    if matchlen == 0 {
        return true, branch, nil
    }
    // Otherwise, replace it with a short node leading up to the branch.
    return true, &shortNode{ Key: key[:matchlen], Val: branch, flags: t.newFlag()}, nil
```

```
// prefixLen returns the length of the common prefix of a and b.
func prefixLen(a, b []byte) int {
    var i, length = 0, len(a)
    if len(b) < length {
        length = len(b)
    }
    for ; i < length; i++ {
        if a[i] != b[i] {
            break
        }
    }
    return i
}
```

```
len+1:], n.Val)
], value)
```

prefixLen을 통해 prefix의 길이 확인 (ex. abf81e4da와 abf71ac90이 있다고 한다면 prefixLen의 길이는 3)

2. trie

Onther Inc.

```
switch n := n.(type) {
case *shortNode:
    matchlen := prefixLen(key, n.Key)
    // If the whole key matches, keep this short node as is
    // and only update the value.
    if matchlen == len(n.Key) {
        dirty, nn, err := t.insert(n.Val, append(prefix, key[:matchlen]...), key[matchlen:], value)
        if !dirty || err != nil {
            return false, n, err
        }
        return true, &shortNode{ Key: n.Key,  Val: nn,  flags: t.newFlag()}, nil
    }
    // Otherwise branch out at the index where they differ.
    branch := &fullNode{flags: t.newFlag()}
    var err error
    _, branch.Children[n.Key[matchlen]], err = t.insert( n: nil, append(prefix, n.Key[:matchlen+1]...), n.Key[matchlen+1:], n.Val)
    if err != nil {
        return false, nil, err
    }
    _, branch.Children[key[matchlen]], err = t.insert( n: nil, append(prefix, key[:matchlen+1]...), key[matchlen+1:], value)
    if err != nil {
        return false, nil, err
    }
    // Replace this shortNode with the branch if it occurs at index 0.
    if matchlen == 0 {
        return true, branch, nil
    }
    // Otherwise, replace it with a short node leading up to the branch.
    return true, &shortNode{ Key: key[:matchlen],  Val: branch,  flags: t.newFlag()}, nil
```

새로 입력하는 key와 원래 있던 키의 prefix 길이와 기존 prefix의 길이가 같으면 다음 값들의 입력 진행

2. trie

Onther Inc.

```
switch n := n.(type) {
case *shortNode:
    matchlen := prefixLen(key, n.Key)
    // If the whole key matches, keep this short node as is
    // and only update the value.
    if matchlen == len(n.Key) {
        dirty, nn, err := t.insert(n.Val, append(prefix, key[:matchlen]...), key[matchlen:], value)
        if !dirty || err != nil {
            return false, n, err
        }
        return true, &shortNode{ Key: n.Key,  Val: nn,  flags: t.newFlag()}, nil
    }
    // Otherwise branch out at the index where they differ.
    branch := &fullNode{flags: t.newFlag()}
    var err error
    _, branch.Children[n.Key[matchlen]], err = t.insert( n: nil, append(prefix, n.Key[:matchlen+1]...), n.Key[matchlen+1:], n.Val)
    if err != nil {
        return false, nil, err
    }
    _, branch.Children[key[matchlen]], err = t.insert( n: nil, append(prefix, key[:matchlen+1]...), key[matchlen+1:], value)
    if err != nil {
        return false, nil, err
    }
    // Replace this shortNode with the branch if it occurs at index 0.
    if matchlen == 0 {
        return true, branch, nil
    }
    // Otherwise, replace it with a short node leading up to the branch.
    return true, &shortNode{ Key: key[:matchlen],  Val: branch,  flags: t.newFlag()}, nil
```

기존에 trie에 들어 있는 값들을 조정하는 단계

2. trie

Onther Inc.

```
switch n := n.(type) {
case *shortNode:
    matchlen := prefixLen(key, n.Key)
    // If the whole key matches, keep this short node as is
    // and only update the value.
    if matchlen == len(n.Key) {
        dirty, nn, err := t.insert(n.Val, append(prefix, key[:matchlen]...), key[matchlen:], value)
        if !dirty || err != nil {
            return false, n, err
        }
        return true, &shortNode{ Key: n.Key,  Val: nn,  flags: t.newFlag()}, nil
    }
    // Otherwise branch out at the index where they differ.
    branch := &fullNode{flags: t.newFlag()}
    var err error
    _, branch.Children[n.Key[matchlen]], err = t.insert( n: nil, append(prefix, n.Key[:matchlen+1]...), n.Key[matchlen+1:], n.Val)
    if err != nil {
        return false, nil, err
    }
    _, branch.Children[key[matchlen]], err = t.insert( n: nil, append(prefix, key[:matchlen+1]...), key[matchlen+1:], value)
    if err != nil {
        return false, nil, err
    }
    // Replace this shortNode with the branch if it occurs at index 0.
    if matchlen == 0 {
        return true, branch, nil
    }
    // Otherwise, replace it with a short node leading up to the branch.
    return true, &shortNode{ Key: key[:matchlen],  Val: branch,  flags: t.newFlag()}, nil
```

새로 trie에 입력하는 값들의 입력을 진행

Onther Inc.

3. internal/ethapi

3. internal/ethapi - backend.go

Onther Inc.

```go
type Backend interface {
    // General Ethereum API
    Downloader() *downloader.Downloader
    ProtocolVersion() int
    SuggestPrice(ctx context.Context) (*big.Int, error)
    ChainDb() ethdb.Database
    EventMux() *event.TypeMux
    AccountManager() *accounts.Manager

    // BlockChain API
    SetHead(number uint64)
    HeaderByNumber(ctx context.Context, blockNr rpc.BlockNumber) (*types.Header, error)
    BlockByNumber(ctx context.Context, blockNr rpc.BlockNumber) (*types.Block, error)
    StateAndHeaderByNumber(ctx context.Context, blockNr rpc.BlockNumber) (*state.StateDB, *types.Header, error)
    GetBlock(ctx context.Context, blockHash common.Hash) (*types.Block, error)
    GetReceipts(ctx context.Context, blockHash common.Hash) (types.Receipts, error)
    GetTd(blockHash common.Hash) *big.Int
    GetEVM(ctx context.Context, msg core.Message, state *state.StateDB, header *types.Header, vmCfg vm.Config) (*vm.EVM, func() error, error)
    SubscribeChainEvent(ch chan<- core.ChainEvent) event.Subscription
    SubscribeChainHeadEvent(ch chan<- core.ChainHeadEvent) event.Subscription
    SubscribeChainSideEvent(ch chan<- core.ChainSideEvent) event.Subscription

    // TxPool API
    SendTx(ctx context.Context, signedTx *types.Transaction) error
    GetPoolTransactions() (types.Transactions, error)
    GetPoolTransaction(txHash common.Hash) *types.Transaction
    GetPoolNonce(ctx context.Context, addr common.Address) (uint64, error)
    Stats() (pending int, queued int)
    TxPoolContent() (map[common.Address]types.Transactions, map[common.Address]types.Transactions)
    SubscribeNewTxsEvent(chan<- core.NewTxsEvent) event.Subscription

    ChainConfig() *params.ChainConfig
    CurrentBlock() *types.Block
}
```

3. internal/ethapi

Onther Inc.

```
type PrivateAccountAPI struct {
    am        *accounts.Manager
    nonceLock *AddrLocker
    b         Backend
}
```

PrivateAccountAPI는이 노드에서 관리하는 계정에 액세스하는 API를 제공.

계정들을 생성하고 락, 언락하는 메소드를 제공

일부 방법은 암호를 허용하기 때문에 기본적으로 private으로 간주됨

3. internal/ethapi

Onther Inc.

```
// PublicEthereumAPI provides an API to access Ethereum related information.
// It offers only methods that operate on public data that is freely available to anyone.
type PublicEthereumAPI struct {
    b Backend
}

// PublicTxPoolAPI offers and API for the transaction pool. It only operates on data that is non confidential.
type PublicTxPoolAPI struct {
    b Backend
}

// PublicBlockChainAPI provides an API to access the Ethereum blockchain.
// It offers only methods that operate on public data that is freely available to anyone.
type PublicBlockChainAPI struct {
    b Backend
}
```

위 구조체들은 Backend 인터페이스를 호출해 사용

3. internal/ethapi

Onther Inc.

```
// RPCTransaction represents a transaction that will serialize to the RPC representation of a transaction
type RPCTransaction struct {
    BlockHash        common.Hash     `json:"blockHash"`
    BlockNumber      *hexutil.Big    `json:"blockNumber"`
    From             common.Address  `json:"from"`
    Gas              hexutil.Uint64  `json:"gas"`
    GasPrice         *hexutil.Big    `json:"gasPrice"`
    Hash             common.Hash     `json:"hash"`
    Input            hexutil.Bytes   `json:"input"`
    Nonce            hexutil.Uint64  `json:"nonce"`
    To               *common.Address `json:"to"`
    TransactionIndex hexutil.Uint    `json:"transactionIndex"`
    Value            *hexutil.Big    `json:"value"`
    V                *hexutil.Big    `json:"v"`
    R                *hexutil.Big    `json:"r"`
    S                *hexutil.Big    `json:"s"`
}
```

3. internal/ethapi

Onther Inc.

```
// PublicTransactionPoolAPI exposes methods for the RPC interface
type PublicTransactionPoolAPI struct {
    b         Backend
    nonceLock *AddrLocker
}
```

3. internal/ethapi

Onther Inc.

```go
// SendTxArgs represents the arguments to sumbit a new transaction into the transaction pool.
type SendTxArgs struct {
    From     common.Address  `json:"from"`
    To       *common.Address `json:"to"`
    Gas      *hexutil.Uint64 `json:"gas"`
    GasPrice *hexutil.Big    `json:"gasPrice"`
    Value    *hexutil.Big    `json:"value"`
    Nonce    *hexutil.Uint64 `json:"nonce"`
    // We accept "data" and "input" for backwards-compatibility reasons. "input" is the
    // newer name and should be preferred by clients.
    Data  *hexutil.Bytes `json:"data"`
    Input *hexutil.Bytes `json:"input"`
}
```

위 구조체들은 Backend 인터페이스를 호출해 사용

3. internal/ethapi

Onther Inc.

```go
// SignTransactionResult represents a RLP encoded signed transaction.
type SignTransactionResult struct {
    Raw hexutil.Bytes      `json:"raw"`
    Tx  *types.Transaction `json:"tx"`
}
```

Onther Inc.

4. core/rawdb

4. core/rawdb

Onther Inc.

Index

Variables
func DeleteBlock(db DatabaseDeleter, hash common.Hash, number uint64)
func DeleteBody(db DatabaseDeleter, hash common.Hash, number uint64)
func DeleteCanonicalHash(db DatabaseDeleter, number uint64)
func DeleteHeader(db DatabaseDeleter, hash common.Hash, number uint64)
func DeleteReceipts(db DatabaseDeleter, hash common.Hash, number uint64)
func DeleteTd(db DatabaseDeleter, hash common.Hash, number uint64)
func DeleteTxLookupEntry(db DatabaseDeleter, hash common.Hash)
func FindCommonAncestor(db DatabaseReader, a, b *types.Header) *types.Header
func HasBody(db DatabaseReader, hash common.Hash, number uint64) bool
func HasHeader(db DatabaseReader, hash common.Hash, number uint64) bool
func ReadBlock(db DatabaseReader, hash common.Hash, number uint64) *types.Block
func ReadBloomBits(db DatabaseReader, bit uint, section uint64, head common.Hash) ([]byte, error)
func ReadBody(db DatabaseReader, hash common.Hash, number uint64) *types.Body
func ReadBodyRLP(db DatabaseReader, hash common.Hash, number uint64) rlp.RawValue
func ReadCanonicalHash(db DatabaseReader, number uint64) common.Hash
func ReadChainConfig(db DatabaseReader, hash common.Hash) *params.ChainConfig
func ReadDatabaseVersion(db DatabaseReader) int
func ReadFastTrieProgress(db DatabaseReader) uint64
func ReadHeadBlockHash(db DatabaseReader) common.Hash
func ReadHeadFastBlockHash(db DatabaseReader) common.Hash
func ReadHeadHeaderHash(db DatabaseReader) common.Hash
func ReadHeader(db DatabaseReader, hash common.Hash, number uint64) *types.Header
func ReadHeaderNumber(db DatabaseReader, hash common.Hash) *uint64
func ReadHeaderRLP(db DatabaseReader, hash common.Hash, number uint64) rlp.RawValue

4. core/rawdb

Onther Inc.

```
func (bc *BlockChain) insert(block *types.Block) {
    // If the block is on a side chain or an unknown one, force other heads onto it too
    updateHeads := rawdb.ReadCanonicalHash(bc.db, block.NumberU64()) != block.Hash()

    // Add the block to the canonical chain number scheme and mark as the head
    rawdb.WriteCanonicalHash(bc.db, block.Hash(), block.NumberU64())
    rawdb.WriteHeadBlockHash(bc.db, block.Hash())

    bc.currentBlock.Store(block)

    // If the block is better than our head or is on a different chain, force update heads
    if updateHeads {
        bc.hc.SetCurrentHeader(block.Header())
        rawdb.WriteHeadFastBlockHash(bc.db, block.Hash())

        bc.currentFastBlock.Store(block)
    }
}
```

core/blockchain.go의 insert 함수를 엔트리포인트로 삼아 rawdb를 살펴볼 예정

4. core/rawdb

Onther Inc.

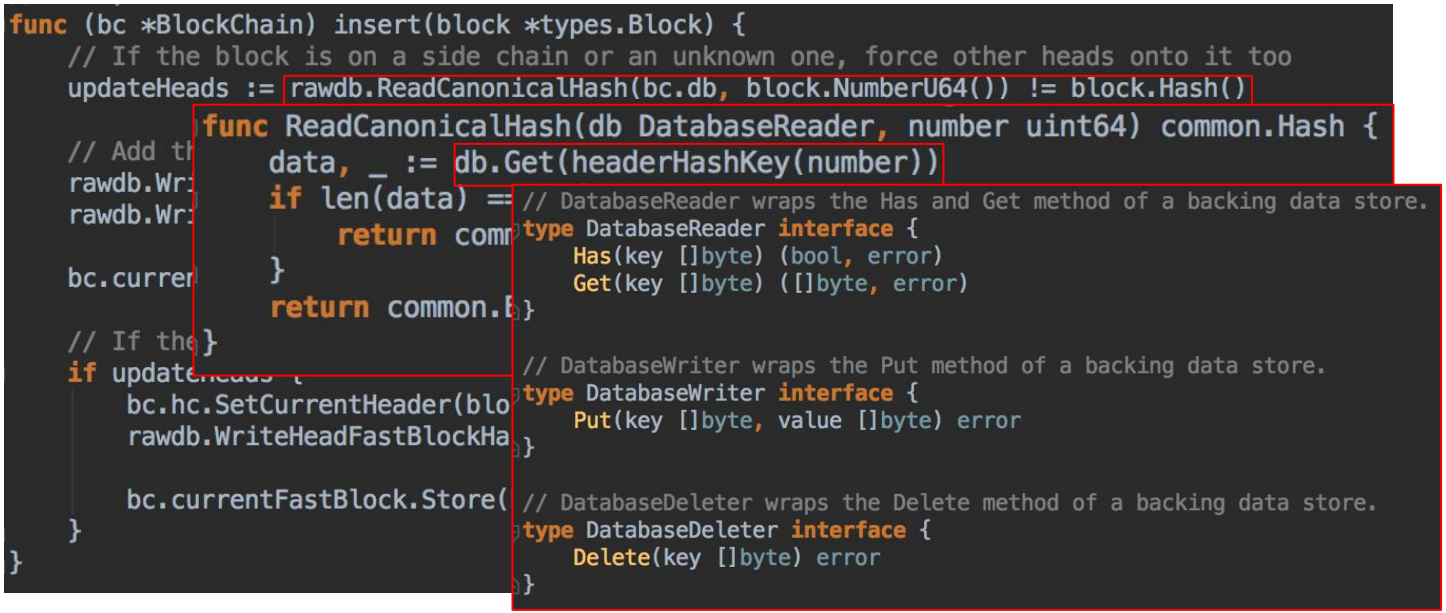
```
func (bc *BlockChain) insert(block *types.Block) {
    // If the block is on a side chain or an unknown one, force other heads onto it too
    updateHeads := rawdb.ReadCanonicalHash(bc.db, block.NumberU64()) != block.Hash()

    // Add th
    rawdb.Wri
    rawdb.Wri

    bc.curren

    // If the
    if update
        bc.hc.SetCurrentHeader(block.Header())
        rawdb.WriteHeadFastBlockHash(bc.db, block.Hash())

        bc.currentFastBlock.Store(block)
    }
}
```

```
func ReadCanonicalHash(db DatabaseReader, number uint64) common.Hash {
    data, _ := db.Get(headerHashKey(number))
    if len(data) == 0 {
        return common.Hash{}
    }
    return common.BytesToHash(data)
}
```

ReadCanonicalHash는 표준 블록 번호에 할당 된 해시를 검색

4. core/rawdb

Onther Inc.



```go
func (bc *BlockChain) insert(block *types.Block) {
    // If the block is on a side chain or an unknown one, force other heads onto it too
    updateHeads := rawdb.ReadCanonicalHash(bc.db, block.NumberU64()) != block.Hash()

    // Add th
    rawdb.Wr
    rawdb.Wr

    bc.curren

    // If the
    if update
        bc.hc.SetCurrentHeader(blo
        rawdb.WriteHeadFastBlockHa

        bc.currentFastBlock.Store(
    }
}
```

```go
func ReadCanonicalHash(db DatabaseReader, number uint64) common.Hash {
    data, _ := db.Get(headerHashKey(number))
    if len(data) ==
        return comm
    }
    return common.B
}
```

```go
// DatabaseReader wraps the Has and Get method of a backing data store.
type DatabaseReader interface {
    Has(key []byte) (bool, error)
    Get(key []byte) ([]byte, error)
}

// DatabaseWriter wraps the Put method of a backing data store.
type DatabaseWriter interface {
    Put(key []byte, value []byte) error
}

// DatabaseDeleter wraps the Delete method of a backing data store.
type DatabaseDeleter interface {
    Delete(key []byte) error
}
```

데이터베이스에 읽고 쓰고 지우는 인터페이스들은 rawdb/interface.go에 정의되어 있음

4. core/rawdb

Onther Inc.

```
func (bc *BlockChain) insert(block *types.Block) {
    // If the block is on a side chain or an unknown one, force other heads onto it too
    updateHeads := rawdb.ReadCanonicalHash(bc.db, block.NumberU64()) != block.Hash()

    // Add the block to the canonical chain number scheme and mark as the head
    rawdb.WriteCanonicalHash(bc.db, block.Hash(), block.NumberU64())
    rawdb
    bc.cu
    // If
    if up
        b
        rawdb.WriteHeadFastBlockHash(bc.db, block.Hash())

        bc.currentFastBlock.Store(block)
    }
}
```

```
// WriteCanonicalHash stores the hash assigned to a canonical block number.
func WriteCanonicalHash(db DatabaseWriter, hash common.Hash, number uint64) {
    if err := db.Put(headerHashKey(number), hash.Bytes()); err != nil {
        log.Crit("Failed to store number to hash mapping", "err", err)
    }
}
```

WriteCanonicalHash는 표준 블록 번호에 할당 된 해시를 저장

4. core/rawdb

Onther Inc.

```
func (bc *BlockChain) insert(block *types.Block) {
    // If the block is on a side chain or an unknown one, force other heads onto it too
    updateHeads := rawdb.ReadCanonicalHash(bc.db, block.NumberU64()) != block.Hash()

    // Add the block to the canonical chain number scheme and mark as the head
    rawdb.WriteCanonicalHash(bc.db, block.Hash(), block.NumberU64())
    rawdb.WriteHeadBlockHash(bc.db, block.Hash())
```

```
// WriteHeadBlockHash stores the head block's hash.
func WriteHeadBlockHash(db DatabaseWriter, hash common.Hash) {
    if err := db.Put(headBlockKey, hash.Bytes()); err != nil {
        log.Crit("Failed to store last block's hash", "err", err)
    }
}
```

```
        bc.currentFastBlock.Store(block)
    }
}
```

WriteHeadBlockHash는 헤드 블록의 해시를 저장

4. core/rawdb

Onther Inc.

```
func (bc *BlockChain) insert(block *types.Block) {
    // If the block is on a side chain or an unknown one, force other heads onto it too
    updateHeads := rawdb.ReadCanonicalHash(bc.db, block.NumberU64()) != block.Hash()

    // Add the block to the canonical chain number scheme and mark as the head
    rawdb.WriteCanonicalHash(bc.db, block.Hash(), block.NumberU64())
    rawdb.Wri
    bc.curre
    // If th
    if updat
        bc.h
        rawdb.WriteHeadFastBlockHash(bc.db, block.Hash())

        bc.currentFastBlock.Store(block)
    }
}
```

```
// WriteHeadFastBlockHash stores the hash of the current fast-sync head block.
func WriteHeadFastBlockHash(db DatabaseWriter, hash common.Hash) {
    if err := db.Put(headFastBlockKey, hash.Bytes()); err != nil {
        log.Crit("Failed to store last fast block's hash", "err", err)
    }
}
```

WriteHeadBlockHash는 헤드 블록의 해시를 저장

4. core/rawdb - accesor_chain&indexes

Onther Inc.

```go
// ReadReceipt retrieves a specific transaction receipt from the database, along with
// its added positional metadata.
func ReadReceipt(db DatabaseReader, hash common.Hash) (*types.Receipt, common.Hash, uint64, uint64) {
    blockHash, blockNumber,
    if blockHash == (commor
        return nil, common.
    }
    receipts := ReadReceipt
    if len(receipts) <= int
        log.Error("Receipt
        return nil, common.
    }
    return receipts[receipt
}
```

```go
// ReadReceipts retrieves all the transaction receipts belonging to a block.
func ReadReceipts(db DatabaseReader, hash common.Hash, number uint64) types.Receipts {
    // Retrieve the flattened receipt slice
    data, _ := db.Get(blockReceiptsKey(number, hash))
    if len(data) == 0 {
        return nil
    }
    // Convert the revceipts from their storage form to their internal representation
    storageReceipts := []*types.ReceiptForStorage{}
    if err := rlp.DecodeBytes(data, &storageReceipts); err != nil {
        log.Error("Invalid receipt array RLP", "hash", hash, "err", err)
        return nil
    }
    receipts := make(types.Receipts, len(storageReceipts))
    for i, receipt := range storageReceipts {
        receipts[i] = (*types.Receipt)(receipt)
    }
    return receipts
}
```

Onther Inc.

5. ethdb

5. ethdb

Onther Inc.

Index

Constants
Variables
type Batch
- func NewTableBatch(db Database, prefix string) Batch

type Database
- func NewTable(db Database, prefix string) Database

type Deleter
type LDBDatabase
- func NewLDBDatabase(file string, cache int, handles int) (*LDBDatabase, error)
- func (db *LDBDatabase) Close()
- func (db *LDBDatabase) Delete(key []byte) error
- func (db *LDBDatabase) Get(key []byte) ([]byte, error)
- func (db *LDBDatabase) Has(key []byte) (bool, error)
- func (db *LDBDatabase) LDB() *leveldb.DB
- func (db *LDBDatabase) Meter(prefix string)
- func (db *LDBDatabase) NewBatch() Batch
- func (db *LDBDatabase) NewIterator() iterator.Iterator
- func (db *LDBDatabase) NewIteratorWithPrefix(prefix []byte) iterator.Iterator
- func (db *LDBDatabase) Path() string
- func (db *LDBDatabase) Put(key []byte, value []byte) error

type MemDatabase
- func NewMemDatabase() *MemDatabase
- func NewMemDatabaseWithCap(size int) *MemDatabase
- func (db *MemDatabase) Close()
- func (db *MemDatabase) Delete(key []byte) error
- func (db *MemDatabase) Get(key []byte) ([]byte, error)
- func (db *MemDatabase) Has(key []byte) (bool, error)
- func (db *MemDatabase) Keys() [][]byte
- func (db *MemDatabase) Len() int
- func (db *MemDatabase) NewBatch() Batch
- func (db *MemDatabase) Put(key []byte, value []byte) error

type Putter

5. ethdb

Onther Inc.

```go
const IdealBatchSize = 100 * 1024

// Putter wraps the database write operation supported by both batches and regular databases.
type Putter interface {
    Put(key []byte, value []byte) error
}

// Deleter wraps the database delete operation supported by both batches and regular databases.
type Deleter interface {
    Delete(key []byte) error
}

// Database wraps all database operations. All methods are safe for concurrent use.
type Database interface {
    Putter
    Deleter
    Get(key []byte) ([]byte, error)
    Has(key []byte) (bool, error)
    Close()
    NewBatch() Batch
}

// Batch is a write-only database that commits changes to its host database
// when Write is called. Batch cannot be used concurrently.
type Batch interface {
    Putter
    Deleter
    ValueSize() int // amount of data in the batch
    Write() error
    // Reset resets the batch for reuse
    Reset()
}
```

ethdb/interface.go

5. ethdb

Onther Inc.

```go
// Put puts the given key / value to the queue
func (db *LDBDatabase) Put(key []byte, value []byte) error {
    return db.db.Put(key, value, nil)
}

func (db *LDBDatabase) Has(key []byte) (bool, error) {
    return db.db.Has(key, nil)
}

// Get returns the given key if it's present.
func (db *LDBDatabase) Get(key []byte) ([]byte, error) {
    dat, err := db.db.Get(key, nil)
    if err != nil {
        return nil, err
    }
    return dat, nil
}

// Delete deletes the key from the queue and database
func (db *LDBDatabase) Delete(key []byte) error {
    return db.db.Delete(key, nil)
}
```

rawdb 인터페이스에서 정의된 함수들이 구현되어 있음

5. ethdb

Onther Inc.

```
type LDBDatabase struct {
    fn string      // filename for reporting
    db *leveldb.DB // LevelDB instance

    compTimeMeter    metrics.Meter // Meter for measuring the total time spent in database compaction
    compReadMeter    metrics.Meter // Meter for measuring the data read during compaction
    compWriteMeter   metrics.Meter // Meter for measuring the data written during compaction
    writeDelayNMeter metrics.Meter // Meter for measuring the write delay number due to database compaction
    writeDelayMeter  metrics.Meter // Meter for measuring the write delay duration due to database compaction
    diskReadMeter    metrics.Meter // Meter for measuring the effective amount of data read
    diskWriteMeter   metrics.Meter // Meter for measuring the effective amount of data written

    quitLock sync.Mutex      // Mutex protecting the quit channel access
    quitChan chan chan error // Quit channel to stop the metrics collection before closing the database

    log log.Logger // Contextual logger tracking the database path
}
```

데이터베이스 압축에 소요 된 총 시간을 측정하기위한 미터, 압축 도중 읽은 데이터를 측정하기위한 미터, 압축 도중 기록 된 데이터를 측정하기위한 미터, 데이터베이스 압축으로 인한 쓰기 지연 수를 측정하기위한 미터, 데이터베이스 압축으로 인한 쓰기 지연 기간 측정을위한 미터, 유효 데이터 량을 측정하기위한 미터, 기록 된 데이터의 유효량을 측정하기위한 미터

5. ethdb

Onther Inc.

```go
// Meter configures the database metrics collectors and
func (db *LDBDatabase) Meter(prefix string) {
    if metrics.Enabled {
        // Initialize all the metrics collector at the requested prefix
        db.compTimeMeter = metrics.NewRegisteredMeter(prefix+"compact/time", nil)
        db.compReadMeter = metrics.NewRegisteredMeter(prefix+"compact/input", nil)
        db.compWriteMeter = metrics.NewRegisteredMeter(prefix+"compact/output", nil)
        db.diskReadMeter = metrics.NewRegisteredMeter(prefix+"disk/read", nil)
        db.diskWriteMeter = metrics.NewRegisteredMeter(prefix+"disk/write", nil)
    }
    // Initialize write delay metrics no matter we are in metric mode or not.
    db.writeDelayMeter = metrics.NewRegisteredMeter(prefix+"compact/writedelay/duration", nil)
    db.writeDelayNMeter = metrics.NewRegisteredMeter(prefix+"compact/writedelay/counter", nil)

    // Create a quit channel for the periodic collector and run it
    db.quitLock.Lock()
    db.quitChan = make(chan chan error)
    db.quitLock.Unlock()

    go db.meter(3 * time.Second)
}
```

Meter는 메트릭 모드에 상관없이 데이터베이스 메트릭 컬렉터를 구성하고 요청 된 prefix 초기화시 쓰기 지연 메트릭에서 모든 메트릭 콜렉터를 초기화. 주기적 컬렉터에 대한 종료 채널을 생성하고 실행.

5. ethdb

Onther Inc.

```
// meter periodically retrieves internal leveldb counters and reports them to
// the metrics subsystem.
//
// This is how a stats table look like (currently):
//   Compactions
//    Level |   Tables   |    Size(MB)   |    Time(sec)  |    Read(MB)   |   Write(MB)
//   -------+------------+---------------+---------------+---------------+---------------
//      0   |          0 |       0.00000 |       1.27969 |       0.00000 |      12.31098
//      1   |         85 |     109.27913 |      28.09293 |     213.92493 |     214.26294
//      2   |        523 |    1000.37159 |       7.26059 |      66.86342 |      66.77884
//      3   |        570 |    1113.18458 |       0.00000 |       0.00000 |       0.00000
//
// This is how the write delay look like (currently):
// DelayN:5 Delay:406.604657ms Paused: false
//
// This is how the iostats look like (currently):
// Read(MB):3895.04860 Write(MB):3654.64712
func (db *LDBDatabase) meter(refresh time.Duration) {
    // Create the counters to store current and previous compaction values
    compactions := make([][]float64, 2)
    for i := 0; i < 2; i++ {
        compactions[i] = make([]float64, 3)
    }
    // Create storage for iostats.
    var iostats [2]float64

    // Create storage and warning log tracer for write delay.
    var (
        delaystats      [2]int64
        lastWritePaused time.Time
    )

    var (
        errc chan error
        merr error
    )
```

Reference

Onther Inc.

http://golang.site/go/article/18-Go-%EC%9D%B8%ED%84%B0%ED%8E%98%EC%9D%B4%EC%8A%A4

https://steemit.com/ethereum/@sigmoid/state

https://github.com/NAKsir-melody/go-ethereum/blob/master/