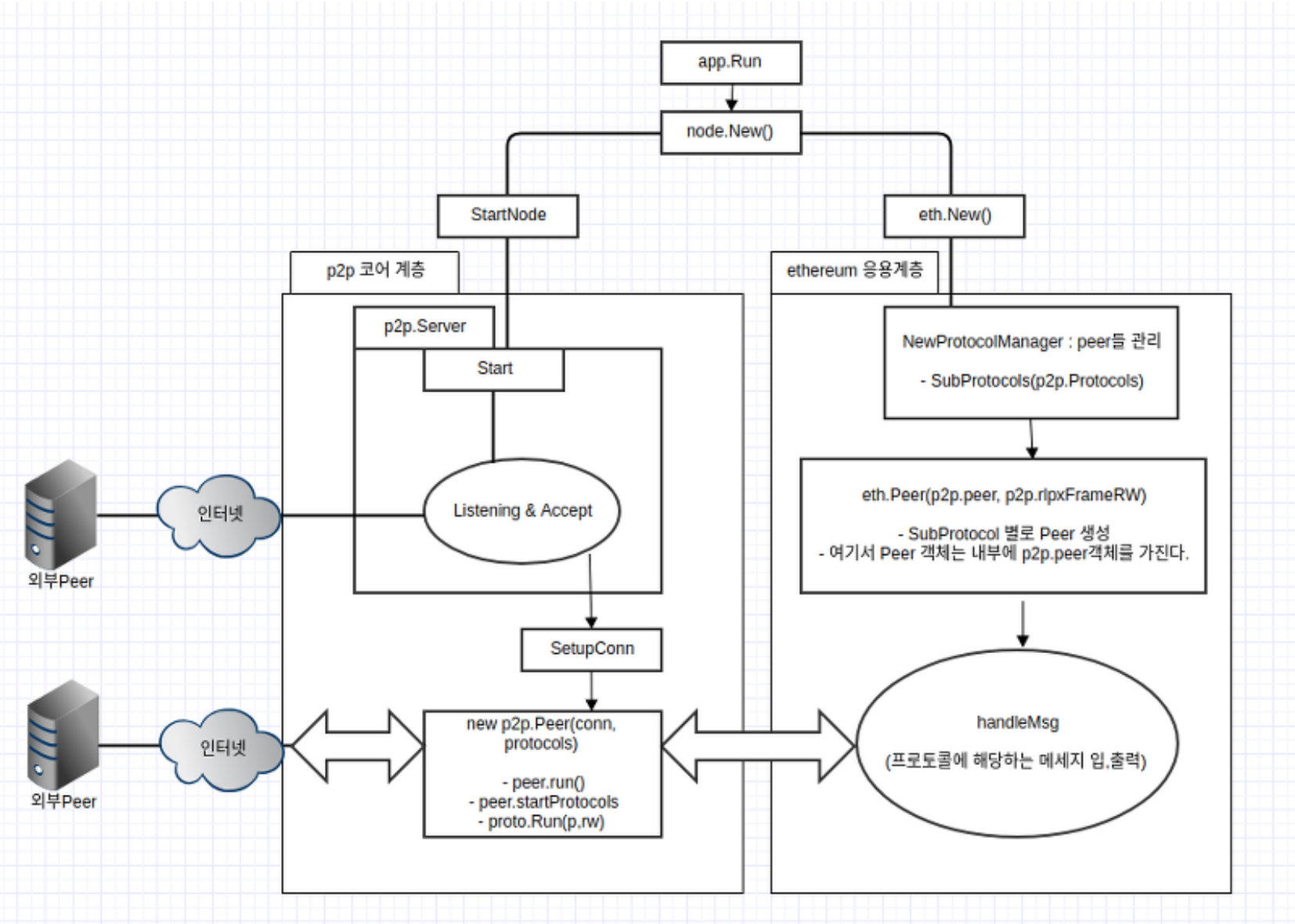
Ethereum Note

Pilkyu Jung



이미지 출처 : https://hamait.tistory.com/971?category=276132

1.8 release

geth

cmd\geth\main.go

어플리케이션을 실행하는 부분

(app.Run)
geth
cmd\geth\main.go

```go
func main() {
    if err := app.Run(os.Args); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}

// geth is the main entry point into the system if no special subcommand is ran.
// It creates a default node based on the command line arguments and runs it in
// blocking mode, waiting for it to be shut down.
func geth(ctx *cli.Context) error {
    log.Debug("[jpk] ")
    log.Debug("[jpk] ")
    log.Debug("[jpk] func geth(ctx *cli.Context) error {")
    if args := ctx.Args(); len(args) > 0 {
        return fmt.Errorf("invalid command: %q", args[0])
    }
    node := makeFullNode(ctx)
    startNode(ctx, node)
    node.Wait()
    return nil
}
```

geth

makeFullNode

cmd\geth\config.go

새로운 노드를 생성하는 부분

geth
1. makeFullNode
cmd\geth\config.go

```
func makeFullNode(ctx *cli.Context) *node.Node {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func makeFullNode(ctx *cli.Context) *node.Node {")
    stack, cfg := makeConfigNode(ctx)
    if ctx.GlobalIsSet(utils.ConstantinopleOverrideFlag.Name) {
        cfg.Eth.ConstantinopleOverride = new(big.Int).SetUint64(ctx.GlobalUint64
        (utils.ConstantinopleOverrideFlag.Name))
    }
    log.Info("[jpk] makeFullNode => ")
    utils.RegisterEthService(stack, &cfg.Eth)

    if ctx.GlobalBool(utils.DashboardEnabledFlag.Name) {
        utils.RegisterDashboardService(stack, &cfg.Dashboard, gitCommit)
    }
    // Whisper must be explicitly enabled by specifying at least 1 whisper flag or in dev mode
    shhEnabled := enableWhisper(ctx)
    shhAutoEnabled := !ctx.GlobalIsSet(utils.WhisperEnabledFlag.Name) && ctx.GlobalIsSet
    (utils.DeveloperFlag.Name)
    if shhEnabled || shhAutoEnabled {
        if ctx.GlobalIsSet(utils.WhisperMaxMessageSizeFlag.Name) {
            cfg.Shh.MaxMessageSize = uint32(ctx.Int(utils.WhisperMaxMessageSizeFlag.Name))
        }
        if ctx.GlobalIsSet(utils.WhisperMinPOWFlag.Name) {
```

새로 생성되는 노드의 config를 세팅함

makeFullNode

makeFullNode
2. makeConfigNode
cmd\geth\config.go

```
func makeConfigNode(ctx *cli.Context) (*node.Node, gethConfig) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func makeConfigNode(ctx *cli.Context) (*node.Node, gethConfig) {")
    // Load defaults.
    cfg := gethConfig{
        Eth:       eth.DefaultConfig,
        Shh:       whisper.DefaultConfig,
        Node:      defaultNodeConfig(),
        Dashboard: dashboard.DefaultConfig,
    }

    // Load config file.
    if file := ctx.GlobalString(configFileFlag.Name); file != "" {
        if err := loadConfig(file, &cfg); err != nil {
            utils.Fatalf("%v", err)
        }
    }

    // Apply flags.
    utils.SetNodeConfig(ctx, &cfg.Node)
    log.Info("[jpk] makeConfigNode => ")
    stack, err := node.NewNode(&cfg.Node)
    if err != nil {
        utils.Fatalf("Failed to create the protocol stack: %v", err)
    }
    utils.SetEthConfig(ctx, stack, &cfg.Eth)
```

makeFullNode

makeConfigNode

2-1. NewNode

node\node.go

```go
// New creates a new P2P node, ready for protocol registration.
func NewNode(conf *Config) (*Node, error) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func NewNode(conf *Config) (*Node, error) {")
    // Copy config and resolve the datadir so future changes to the current
    // working directory don't affect the node.
    confCopy := *conf
    conf = &confCopy
    if conf.DataDir != "" {
        absdatadir, err := filepath.Abs(conf.DataDir)
        if err != nil {
            return nil, err
        }
        conf.DataDir = absdatadir
    }
    // Ensure that the instance name doesn't cause weird conflicts with
    // other files in the data directory.
    if strings.ContainsAny(conf.Name, `/\`) {
        return nil, errors.New(`Config.Name must not contain '/' or '\'`)
    }
    if conf.Name == datadirDefaultKeyStore {
        return nil, errors.New(`Config.Name cannot be "` + datadirDefaultKeyStore + `"`)
    }
    if strings.HasSuffix(conf.Name, ".ipc") {
        return nil, errors.New(`Config.Name cannot end in ".ipc"`)
    }
    // Ensure that the AccountManager method works before the node has started.
    // We rely on this in cmd/geth.
    am, ephemeralKeystore, err := makeAccountManager(conf)
```

makeFullNode

NewNode

2-2. makeAccountManager

node\config.go

```go
func makeAccountManager(conf *Config) (*accounts.Manager, string, error) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func makeAccountManager(conf *Config) (*accounts.Manager, string, error) {")
    scryptN, scryptP, keydir, err := conf.AccountConfig()
    var ephemeral string
    if keydir == "" {
        // There is no datadir.
        keydir, err = ioutil.TempDir("", "go-ethereum-keystore")
        ephemeral = keydir
    }

    if err != nil {
        return nil, "", err
    }
    if err := os.MkdirAll(keydir, 0700); err != nil {
        return nil, "", err
    }
    // Assemble the account manager and supported backends
    backends := []accounts.Backend{
        keystore.NewKeyStore(keydir, scryptN, scryptP),
    }
    if !conf.NoUSB {
        // Start a USB hub for Ledger hardware wallets
        if ledgerhub, err := usbwallet.NewLedgerHub(); err != nil {
            log.Warn(fmt.Sprintf("Failed to start Ledger hub, disabling: %v", err))
        } else {
            backends = append(backends, ledgerhub)
        }
        // Start a USB hub for Trezor hardware wallets
```

makeFullNode
RegisterEthService
cmd\utils\flags.go

Ethereum Service를 생성하고 등록함

makeFullNode

1. RegisterEthService

cmd\utils\flags.go

```go
// RegisterEthService adds an Ethereum client to the stack.
func RegisterEthService(stack *node.Node, cfg *eth.Config) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func RegisterEthService(stack *node.Node, cfg *eth.Config) {")
    var err error
    if cfg.SyncMode == downloader.LightSync {
        log.Info("[jpk] RegisterEthService => if cfg.SyncMode == downloader.LightSync {")
        err = stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
            return les.New(ctx, cfg)
        })
    } else {
        log.Info("[jpk] RegisterEthService => } else {")
        err = stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
            fullNode, err := eth.NewEthereumObj(ctx, cfg)
            if fullNode != nil && cfg.LightServ > 0 {
                ls, _ := les.NewLesServer(fullNode, cfg)
                fullNode.AddLesServer(ls)
            }
            return fullNode, err
        })
    }
    if err != nil {
        Fatalf("Failed to register the Ethereum service: %v", err)
    }
}
```

RegisterEthService

RegisterEthService

2. NewEthereumObj

eth\backend.go

```go
// New creates a new Ethereum object (including the
// initialisation of the common Ethereum object)
func NewEthereumObj(ctx *node.ServiceContext, config *Config) (*Ethereum, error) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func NewEthereumObj(ctx *node.ServiceContext, config *Config) (*Ethereum, error)
     {")
    // Ensure configuration values are compatible and sane
    if config.SyncMode == downloader.LightSync {
        return nil, errors.New("can't run eth.Ethereum in light sync mode, use les.LightEthereum")
    }
    if !config.SyncMode.IsValid() {
        return nil, fmt.Errorf("invalid sync mode %d", config.SyncMode)
    }
    if config.MinerGasPrice == nil || config.MinerGasPrice.Cmp(common.Big0) <= 0 {
        log.Warn("Sanitizing invalid miner gas price", "provided", config.MinerGasPrice, "updated",
        DefaultConfig.MinerGasPrice)
        config.MinerGasPrice = new(big.Int).Set(DefaultConfig.MinerGasPrice)
    }
    // Assemble the Ethereum object
    chainDb, err := CreateDB(ctx, config, "chaindata")
    if err != nil {
        return nil, err
    }
    chainConfig, genesisHash, genesisErr := core.SetupGenesisBlockWithOverride(chainDb,
```

RegisterEthService

NewEthereumObj

2-1. NewTxPool

core\tx_pool.go

```go
// NewTxPool creates a new transaction pool to gather, sort and filter inbound
// transactions from the network.
func NewTxPool(config TxPoolConfig, chainconfig *params.ChainConfig, chain blockChain) *TxPool {
    log.Debug("[jpk] ")
    log.Debug("[jpk] ")
    log.Debug("[jpk] func NewTxPool(config TxPoolConfig, chainconfig *params.ChainConfig, chain
    blockChain) *TxPool {")
    // Sanitize the input to ensure no vulnerable gas prices are set
    config = (&config).sanitize()

    // Create the transaction pool with its initial settings
    pool := &TxPool{
        config:      config,
        chainconfig: chainconfig,
        chain:       chain,
        signer:      types.NewEIP155Signer(chainconfig.ChainID),
        pending:     make(map[common.Address]*txList),
        queue:       make(map[common.Address]*txList),
        beats:       make(map[common.Address]time.Time),
        all:         newTxLookup(),
        chainHeadCh: make(chan ChainHeadEvent, chainHeadChanSize),
        gasPrice:    new(big.Int).SetUint64(config.PriceLimit),
    }
    pool.locals = newAccountSet(pool.signer)
    for _, addr := range config.Locals {
        log.Info("Setting new local account", "address", addr)
        pool.locals.add(addr)
    }
    pool.priced = newTxPricedList(pool.all)
    pool.reset2Pool(nil, chain.CurrentBlock().Header())

    // If local transactions and journaling is enabled, load from disk
    if !config.NoLocals && config.Journal != "" {
```

RegisterEthService

NewEthereumObj

2-2. NewProtocolManager

eth\handler.go

```go
// NewProtocolManager returns a new Ethereum sub protocol manager. The Ethereum sub protocol
manages peers capable
// with the Ethereum network.
func NewProtocolManager(config *params.ChainConfig, mode downloader.SyncMode, networkID uint64, mux
*event.TypeMux, txpool txPool, engine consensus.Engine, blockchain *core.BlockChain, chaindb
ethdb.Database, whitelist map[uint64]common.Hash) (*ProtocolManager, error) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func NewProtocolManager(config *params.ChainConfig, mode downloader.SyncMode,
    networkID uint64, mux *event.TypeMux, txpool txPool, engine consensus.Engine, blockchain
    *core.BlockChain, chaindb ethdb.Database, whitelist map[uint64]common.Hash) (*ProtocolManager,
    error) {")
    // Create the protocol manager with the base fields
    manager := &ProtocolManager{
        networkID:   networkID,
        eventMux:    mux,
        txpool:      txpool,
        blockchain:  blockchain,
        chainconfig: config,
        peers:       newPeerSet(),
        whitelist:   whitelist,
        newPeerCh:   make(chan *peer),
        noMorePeers: make(chan struct{}),
        txsyncCh:    make(chan *txsync),
        quitSync:    make(chan struct{}),
    }
    // Figure out whether to allow fast sync or not
    if mode == downloader.FastSync && blockchain.CurrentBlock().NumberU64() > 0 {
        log.Warn("Blockchain not empty, fast sync disabled")
        mode = downloader.FullSync
    }
```

RegisterEthService

NewEthereumObj

2-3. NewMiner

miner\miner.go

```
func NewMiner(eth Backend, config *params.ChainConfig, mux *event.TypeMux, engine consensus.Engine,
recommit time.Duration, gasFloor, gasCeil uint64, isLocalBlock func(block *types.Block) bool)
*Miner {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func NewMiner(eth Backend, config *params.ChainConfig, mux *event.TypeMux,
    engine consensus.Engine, recommit time.Duration, gasFloor, gasCeil uint64, isLocalBlock func
    (block *types.Block) bool) *Miner {")
    miner := &Miner{
        eth:      eth,
        mux:      mux,
        engine:   engine,
        exitCh:   make(chan struct{}),
        worker:   newWorker(config, engine, eth, mux, recommit, gasFloor, gasCeil, isLocalBlock),
        canStart: 1,
    }
    go miner.update()

    return miner
}
```

NewMiner
newWorker
miner\worker.go

worker 객체 생성
Loop 실행

NewMiner
1. newWorker
miner\worker.go

```
func newWorker(config *params.ChainConfig, engine consensus.Engine, eth Backend, mux *event.TypeMux,
 recommit time.Duration, gasFloor, gasCeil uint64, isLocalBlock func(*types.Block) bool) *worker {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func newWorker(config *params.ChainConfig, engine consensus.Engine, eth Backend,
     mux *event.TypeMux, recommit time.Duration, gasFloor, gasCeil uint64, isLocalBlock func
    (*types.Block) bool) *worker {")
    worker := &worker{
        config:             config,
        engine:             engine,
        eth:                eth,
        mux:                mux,
        chain:              eth.BlockChain(),
        gasFloor:           gasFloor,
        gasCeil:            gasCeil,
        isLocalBlock:       isLocalBlock,
        localUncles:        make(map[common.Hash]*types.Block),
        remoteUncles:       make(map[common.Hash]*types.Block),
        unconfirmed:        newUnconfirmedBlocks(eth.BlockChain(), miningLogAtDepth),
        pendingTasks:       make(map[common.Hash]*task),
        txsCh:              make(chan core.NewTxsEvent, txChanSize),
        chainHeadCh:        make(chan core.ChainHeadEvent, chainHeadChanSize),
        chainSideCh:        make(chan core.ChainSideEvent, chainSideChanSize),
        newWorkCh:          make(chan *newWorkReq),
        taskCh:             make(chan *task),
```

```
    go worker.mainLoop()
    go worker.newWorkLoop(recommit)
    go worker.resultLoop()
    go worker.taskLoop()
```

newWorker

newWorker
1-1. mainLoop
miner\worker.go

```go
// mainLoop is a standalone goroutine to regenerate the sealing task based on the received event.
func (w *worker) mainLoop() {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (w *worker) mainLoop() {")
    defer w.txsSub.Unsubscribe()
    defer w.chainHeadSub.Unsubscribe()
    defer w.chainSideSub.Unsubscribe()

    for {
        select {
        case req := <-w.newWorkCh:
            log.Info("[jpk] mainLoop => case req := <-w.newWorkCh:")
            w.commitNewWork(req.interrupt, req.noempty, req.timestamp)

        case ev := <-w.chainSideCh:
            // Short circuit for duplicate side blocks
            if _, exist := w.localUncles[ev.Block.Hash()]; exist {
                continue
            }
            if _, exist := w.remoteUncles[ev.Block.Hash()]; exist {
                continue
            }
```

newWorker

newWorker

1-2. newWorkLoop

miner\worker.go

```go
// newWorkLoop is a standalone goroutine to submit new mining work upon received events.
func (w *worker) newWorkLoop(recommit time.Duration) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (w *worker) newWorkLoop(recommit time.Duration) {")
    var (
        interrupt   *int32
        minRecommit = recommit // minimal resubmit interval specified by user.
        timestamp   int64      // timestamp for each round of mining.
    )

    timer := time.NewTimer(0)
    <-timer.C // discard the initial tick

    // commit aborts in-flight transaction execution with given signal and resubmits a new one.
    commit := func(noempty bool, s int32) {
        log.Info("[jpk] newWorkLoop => commit := func(noempty bool, s int32) {")
        if interrupt != nil {
            atomic.StoreInt32(interrupt, s)
        }
        interrupt = new(int32)
        log.Info("[jpk] newWorkLoop => ")
        w.newWorkCh <- &newWorkReq{interrupt: interrupt, noempty: noempty, timestamp: timestamp}
```

newWorker

newWorker
1-3. resultLoop
miner\worker.go

```go
// resultLoop is a standalone goroutine to handle sealing result submitting
// and flush relative data to the database.
func (w *worker) resultLoop() {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (w *worker) resultLoop() {")
    for {
        select {
        case block := <-w.resultCh:
            // Short circuit when receiving empty result.
            if block == nil {
                continue
            }
            // Short circuit when receiving duplicate result caused by resubmitting.
            if w.chain.HasBlock(block.Hash(), block.NumberU64()) {
                continue
            }
            var (
                sealhash = w.engine.SealHash(block.Header())
                hash     = block.Hash()
            )
            w.pendingMu.RLock()
            task, exist := w.pendingTasks[sealhash]
```

newWorker

newWorker
1-4. taskLoop
miner\worker.go

```go
// taskLoop is a standalone goroutine to fetch sealing task from the generator and
// push them to consensus engine.
func (w *worker) taskLoop() {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (w *worker) taskLoop() {")
    var (
        stopCh chan struct{}
        prev   common.Hash
    )

    // interrupt aborts the in-flight sealing task.
    interrupt := func() {
        if stopCh != nil {
            close(stopCh)
            stopCh = nil
        }
    }
    for {
        select {
        case task := <-w.taskCh:
            if w.newTaskHook != nil {
                w.newTaskHook(task)
```

geth

startNode

cmd\geth\main.go

geth
1. startNode
cmd\geth\main.go

```go
// startNode boots up the system node and all registered protocols, after which
// it unlocks any requested accounts, and starts the RPC/IPC interfaces and the
// miner.
func startNode(ctx *cli.Context, stack *node.Node) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func startNode(ctx *cli.Context, stack *node.Node) {")
    debug.Memsize.Add("node", stack)

    // Start up the node itself
    utils.StartNode(stack)

    // Unlock any account specifically requested
    ks := stack.AccountManager().Backends(keystore.KeyStoreType)[0].(*keystore.KeyStore)

    passwords := utils.MakePasswordList(ctx)
    unlocks := strings.Split(ctx.GlobalString(utils.UnlockedAccountFlag.Name), ",")
    for i, account := range unlocks {
        if trimmed := strings.TrimSpace(account); trimmed != "" {
            unlockAccount(ctx, ks, trimmed, i, passwords)
        }
    }
    // Register wallet event handlers to open and auto-derive wallets
    events := make(chan accounts.WalletEvent, 16)
```

startNode

startNode

2. StartNode

cmd\utils\cmd.go

```go
func StartNode(stack *node.Node) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func StartNode(stack *node.Node) {")
    if err := stack.StartNode(); err != nil {
        Fatalf("Error starting protocol stack: %v", err)
    }
    go func() {
        sigc := make(chan os.Signal, 1)
        signal.Notify(sigc, syscall.SIGINT, syscall.SIGTERM)
        defer signal.Stop(sigc)
        <-sigc
        log.Info("Got interrupt, shutting down...")
        go stack.Stop()
        for i := 10; i > 0; i-- {
            <-sigc
            if i > 1 {
                log.Warn("Already shutting down, interrupt more to panic.", "times", i-1)
            }
        }
        debug.Exit() // ensure trace and CPU profile data is flushed.
        debug.LoudPanic("boom")
    }()
}
```

startNode

StartNode

3. StartNode

node\node.go

```go
// Start create a live P2P node and starts running it.
func (n *Node) StartNode() error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (n *Node) Start() error {")
    n.lock.Lock()
    defer n.lock.Unlock()

    // Short circuit if the node's already running
    if n.server != nil {
        return ErrNodeRunning
    }
    if err := n.openDataDir(); err != nil {
        return err
    }

    // Initialize the p2p server. This creates the node key and
    // discovery databases.
    n.serverConfig = n.config.P2P
    n.serverConfig.PrivateKey = n.config.NodeKey()
    n.serverConfig.Name = n.config.NodeName()
    n.serverConfig.Logger = n.log
    if n.serverConfig.StaticNodes == nil {
        n.serverConfig.StaticNodes = n.config.StaticNodes()
    }
    if n.serverConfig.TrustedNodes == nil {
        n.serverConfig.TrustedNodes = n.config.TrustedNodes()
    }
    if n.serverConfig.NodeDatabase == "" {
        n.serverConfig.NodeDatabase = n.config.NodeDB()
    }
    running := &p2p.Server{Config: n.serverConfig}
```

startNode

StartNode

3. StartNode

node\node.go

```
    log.Info("[jpk] StartNode => ")
    if err := running.StartServer(); err != nil {
        return convertFileLockError(err)
    }
    // Start each of the services
    started := []reflect.Type{}
    for kind, service := range services {
        // Start the next service, stopping all previous upon failure
        log.Info("[jpk] StartNode =>")
        if err := service.Start(running); err != nil {
            for _, kind := range started {
                services[kind].Stop()
            }
            running.Stop()

            return err
        }
        // Mark the service started for potential cleanup
        started = append(started, kind)
    }
    // Lastly start the configured RPC interfaces
    log.Info("[jpk] StartNode =>")
    if err := n.startRPC(services); err != nil {
        for _, service := range services {
            service.Stop()
        }
        running.Stop()
        return err
    }
    // Finish initializing the startup
```

StartNode
StartServer
p2p\server.go

StartNode
1. StartServer
p2p\server.go

```go
// Start starts running the server.
// Servers can not be re-used after stopping.
func (srv *Server) StartServer() (err error) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (srv *Server) Start() (err error) {")
    srv.lock.Lock()
    defer srv.lock.Unlock()
    if srv.running {
        return errors.New("server already running")
    }
    srv.running = true
    srv.log = srv.Config.Logger
    if srv.log == nil {
        srv.log = log.New()
    }
    if srv.NoDial && srv.ListenAddr == "" {
        srv.log.Warn("P2P server will be useless, neither dialing nor listening")
    }

    // static fields
    if srv.PrivateKey == nil {
        return errors.New("Server.PrivateKey must be set to a non-nil key")
    }
    if srv.newTransport == nil {
        srv.newTransport = newRLPX
    }
    if srv.Dialer == nil {
        srv.Dialer = TCPDialer{&net.Dialer{Timeout: defaultDialTimeout}}
    }
    srv.quit = make(chan struct{})
    srv.addpeer = make(chan *conn)
```

StartNode
1. StartServer
p2p\server.go

```
        srv.Dialer = TCPDialer{&net.Dialer{Timeout: defaultDialTimeout}}
    }
    srv.quit = make(chan struct{})
    srv.addpeer = make(chan *conn)
    srv.delpeer = make(chan peerDrop)
    srv.posthandshake = make(chan *conn)
    srv.addstatic = make(chan *enode.Node)
    srv.removestatic = make(chan *enode.Node)
    srv.addtrusted = make(chan *enode.Node)
    srv.removetrusted = make(chan *enode.Node)
    srv.peerOp = make(chan peerOpFunc)
    srv.peerOpDone = make(chan struct{})

    if err := srv.setupLocalNode(); err != nil {
        return err
    }
    if srv.ListenAddr != "" {
        if err := srv.setupListening(); err != nil {
            return err
        }
    }
    if err := srv.setupDiscovery(); err != nil {
        return err
    }

    dynPeers := srv.maxDialedConns()
    dialer := newDialState(srv.localnode.ID(), srv.StaticNodes, srv.BootstrapNodes, srv.ntab,
    dynPeers, srv.NetRestrict)
    srv.loopWG.Add(1)
    go srv.runServer(dialer)
    return nil
}
```

StartServer

StartServer

1-1. setupLocalNode

p2p\server.go

```go
func (srv *Server) setupLocalNode() error {
    // Create the devp2p handshake.
    pubkey := crypto.FromECDSAPub(&srv.PrivateKey.PublicKey)
    srv.ourHandshake = &protoHandshake{Version: baseProtocolVersion, Name: srv.Name, ID: pubkey[1:]}
    for _, p := range srv.Protocols {
        srv.ourHandshake.Caps = append(srv.ourHandshake.Caps, p.cap())
    }
    sort.Sort(capsByNameAndVersion(srv.ourHandshake.Caps))

    // Create the local node.
    db, err := enode.OpenDB(srv.Config.NodeDatabase)
    if err != nil {
        return err
    }
    srv.nodedb = db
    srv.localnode = enode.NewLocalNode(db, srv.PrivateKey)
    srv.localnode.SetFallbackIP(net.IP{127, 0, 0, 1})
    srv.localnode.Set(capsByNameAndVersion(srv.ourHandshake.Caps))
    // TODO: check conflicts
    for _, p := range srv.Protocols {
        for _, e := range p.Attributes {
            srv.localnode.Set(e)
        }
    }
    switch srv.NAT.(type) {
    case nil:
```

StartServer

StartServer
1-2. setupListening
p2p\server.go

```go
func (srv *Server) setupListening() error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (srv *Server) setupListening() error {")
    // Launch the TCP listener.
    listener, err := net.Listen("tcp", srv.ListenAddr)
    if err != nil {
        return err
    }
    laddr := listener.Addr().(*net.TCPAddr)
    srv.ListenAddr = laddr.String()
    srv.listener = listener
    srv.localnode.Set(enr.TCP(laddr.Port))

    srv.loopWG.Add(1)
    go srv.listenLoop()

    // Map the TCP listening port if NAT is configured.
    if !laddr.IP.IsLoopback() && srv.NAT != nil {
        srv.loopWG.Add(1)
        go func() {
            nat.Map(srv.NAT, srv.quit, "tcp", laddr.Port, laddr.Port, "ethereum p2p")
            srv.loopWG.Done()
        }()
    }
    return nil
}
```

StartServer

setupListening

1-2-1. listenLoop

p2p\server.go

```go
// listenLoop runs in its own goroutine and accepts
// inbound connections.
func (srv *Server) listenLoop() {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (srv *Server) listenLoop() {")
    defer srv.loopWG.Done()
    srv.log.Debug("TCP listener up", "addr", srv.listener.Addr())

    tokens := defaultMaxPendingPeers
    if srv.MaxPendingPeers > 0 {
        tokens = srv.MaxPendingPeers
    }
    slots := make(chan struct{}, tokens)
    for i := 0; i < tokens; i++ {
        slots <- struct{}{}
    }

    for {
        // Wait for a handshake slot before accepting.
        <-slots

        var (
```

StartServer

listenLoop

1-2-1-1. SetupConn

p2p\server.go

```go
// SetupConn runs the handshakes and attempts to add the connection
// as a peer. It returns when the connection has been added as a peer
// or the handshakes have failed.
func (srv *Server) SetupConn(fd net.Conn, flags connFlag, dialDest *enode.Node) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] SetupConn => ", "fd.RemoteAddr()", fd.RemoteAddr())
    if dialDest != nil {
        log.Info("[jpk] SetupConn => ", "dialDest.ID()", dialDest.ID())
    }
    log.Info("[jpk] func (srv *Server) SetupConn(fd net.Conn, flags connFlag, dialDest *enode.Node)
    error {")
    c := &conn{fd: fd, transport: srv.newTransport(fd), flags: flags, cont: make(chan error)}
    err := srv.setupConn(c, flags, dialDest)
    if err != nil {
        c.close(err)
        srv.log.Trace("Setting up connection failed", "addr", fd.RemoteAddr(), "err", err)
    }
    return err
}
```

StartServer

SetupConn

1-2-1-2. setupConn

p2p\server.go

```go
func (srv *Server) setupConn(c *conn, flags connFlag, dialDest *enode.Node) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    if dialDest != nil {
        log.Info("[jpk] setupConn => ", "dialDest.ID", dialDest.ID())
    }
    log.Info("[jpk] setupConn => ", "c.fd.RemoteAddr()", c.fd.RemoteAddr())
    log.Info("[jpk] func (srv *Server) setupConn(c *conn, flags connFlag, dialDest *enode.Node)
    error {")
    // Prevent leftover pending conns from entering the handshake.
    srv.lock.Lock()
    running := srv.running
    srv.lock.Unlock()
    if !running {
        return errServerStopped
    }
    // If dialing, figure out the remote public key.
    var dialPubkey *ecdsa.PublicKey
    if dialDest != nil {
        log.Info("[jpk] setupConn => if dialDest != nil {")
        dialPubkey = new(ecdsa.PublicKey)
        if dialPubkey != nil {
            log.Info("[jpk] setupConn => ", "dialPubkey", dialPubkey)
        }
```

StartServer

StartServer

1-3. setupDiscovery

p2p\server.go

```go
func (srv *Server) setupDiscovery() error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (srv *Server) setupDiscovery() error {")
    if srv.NoDiscovery && !srv.DiscoveryV5 {
        return nil
    }

    addr, err := net.ResolveUDPAddr("udp", srv.ListenAddr)
    if err != nil {
        return err
    }
    conn, err := net.ListenUDP("udp", addr)
    if err != nil {
        return err
    }
    realaddr := conn.LocalAddr().(*net.UDPAddr)
    srv.log.Debug("UDP listener up", "addr", realaddr)
    if srv.NAT != nil {
        if !realaddr.IP.IsLoopback() {
            go nat.Map(srv.NAT, srv.quit, "udp", realaddr.Port, realaddr.Port, "ethereum discovery")
        }
    }
    srv.localnode.SetFallbackUDP(realaddr.Port)

    // Discovery V4
    var unhandled chan discover.ReadPacket
    var sconn *sharedUDPConn
    if !srv.NoDiscovery {
        if srv.DiscoveryV5 {
            unhandled = make(chan discover.ReadPacket, 100)
            sconn = &sharedUDPConn{conn, unhandled}
```

StartServer

setupDiscovery

1-3-1. ListenUDP

p2p\discover\udp.go

```go
// ListenUDP returns a new table that listens for UDP packets on laddr.
func ListenUDP(c conn, ln *enode.LocalNode, cfg Config) (*Table, error) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func ListenUDP(c conn, ln *enode.LocalNode, cfg Config) (*Table, error) {")
    tab, _, err := newUDP(c, ln, cfg)
    if err != nil {
        return nil, err
    }
    return tab, nil
}
```

StartServer

ListenUDP

1-3-1-1. newUDP

p2p\discover\udp.go

```go
func newUDP(c conn, ln *enode.LocalNode, cfg Config) (*Table, *udp, error) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func newUDP(c conn, ln *enode.LocalNode, cfg Config) (*Table, *udp, error) {")
    udp := &udp{
        conn:            c,
        priv:            cfg.PrivateKey,
        netrestrict:     cfg.NetRestrict,
        localNode:       ln,
        db:              ln.Database(),
        closing:         make(chan struct{}),
        gotreply:        make(chan reply),
        addReplyMatcher: make(chan *replyMatcher),
    }
    tab, err := newTable(udp, ln.Database(), cfg.Bootnodes)
    if err != nil {
        return nil, nil, err
    }
    udp.tab = tab

    udp.wg.Add(2)
    go udp.loop()
    go udp.readLoop(cfg.Unhandled)
    return udp.tab, udp, nil
}
```

StartServer

loop

1-3-1-1-1. loop

p2p\discover\udp.go

```go
// loop runs in its own goroutine. it keeps track of
// the refresh timer and the pending reply queue.
func (t *udp) loop() {
    defer t.wg.Done()

    var (
        plist        = list.New()
        timeout      = time.NewTimer(0)
        nextTimeout  *replyMatcher // head of plist when timeout was last reset
        contTimeouts = 0           // number of continuous timeouts to do NTP checks
        ntpWarnTime  = time.Unix(0, 0)
    )
    <-timeout.C // ignore first timeout
    defer timeout.Stop()

    resetTimeout := func() {
        if plist.Front() == nil || nextTimeout == plist.Front().Value {
            return
        }
        // Start the timer so it fires when the next pending reply has expired.
        now := time.Now()
        for el := plist.Front(); el != nil; el = el.Next() {
            nextTimeout = el.Value.(*replyMatcher)
```

StartServer

readLoop

1-3-1-1-2. readLoop

p2p\discover\udp.go

```go
// readLoop runs in its own goroutine. it handles incoming UDP packets.
func (t *udp) readLoop(unhandled chan<- ReadPacket) {
    defer t.wg.Done()
    if unhandled != nil {
        defer close(unhandled)
    }

    // Discovery packets are defined to be no larger than 1280 bytes.
    // Packets larger than this size will be cut at the end and treated
    // as invalid because their hash won't match.
    buf := make([]byte, 1280)
    for {
        nbytes, from, err := t.conn.ReadFromUDP(buf)
        if netutil.IsTemporaryError(err) {
            // Ignore temporary read errors.
            log.Debug("Temporary UDP read error", "err", err)
            continue
        } else if err != nil {
            // Shut down the loop for permament errors.
            log.Debug("UDP read error", "err", err)
            return
        }
        if t.handlePacket(from, buf[:nbytes]) != nil && unhandled != nil {
            select {
            case unhandled <- ReadPacket{buf[:nbytes], from}:
            default:
            }
        }
    }
}
```

StartNode
Start(Ethereum)
eth\backend.go

StartNode
1. Start(Ethereum)
eth\backend.go

```go
// Start implements node.Service, starting all internal goroutines needed by the
// Ethereum protocol implementation.
func (s *Ethereum) Start(srvr *p2p.Server) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (s *Ethereum) Start(srvr *p2p.Server) error {")
    // Start the bloom bits servicing goroutines
    s.startBloomHandlers(params.BloomBitsBlocks)

    // Start the RPC service
    s.netRPCService = ethapi.NewPublicNetAPI(srvr, s.NetVersion())

    // Figure out a max peers count based on the server limits
    maxPeers := srvr.MaxPeers
    if s.config.LightServ > 0 {
        if s.config.LightPeers >= srvr.MaxPeers {
            return fmt.Errorf("invalid peer config: light peer count (%d) >= total peer count (%d)",
             s.config.LightPeers, srvr.MaxPeers)
        }
        maxPeers -= s.config.LightPeers
    }
    // Start the networking layer and the light server if requested
    s.protocolManager.ProtocolManagerStart(maxPeers)
    if s.lesServer != nil {
        s.lesServer.Start(srvr)
    }
    return nil
}
```

Start(Ethereum)

Start(Ethereum)
2. startBloomHandlers
eth\backend.go

```go
// startBloomHandlers starts a batch of goroutines to accept bloom bit database
// retrievals from possibly a range of filters and serving the data to satisfy.
func (eth *Ethereum) startBloomHandlers(sectionSize uint64) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (eth *Ethereum) startBloomHandlers(sectionSize uint64) {")
    for i := 0; i < bloomServiceThreads; i++ {
        go func() {
            for {
                select {
                case <-eth.shutdownChan:
                    return

                case request := <-eth.bloomRequests:
                    task := <-request
                    task.Bitsets = make([][]byte, len(task.Sections))
                    for i, section := range task.Sections {
                        head := rawdb.ReadCanonicalHash(eth.chainDb, (section+1)*sectionSize-1)
                        if compVector, err := rawdb.ReadBloomBits(eth.chainDb, task.Bit, section,
                        head); err == nil {
                            if blob, err := bitutil.DecompressBytes(compVector, int(sectionSize/8));
                             err == nil {
                                task.Bitsets[i] = blob
                            } else {
                                task.Error = err
                            }
                        } else {
                            task.Error = err
                        }
                    }
                    request <- task
```

Start(Ethereum)

Start(Ethereum)
3. ProtocolManagerStart
eth\backend.go

```go
func (pm *ProtocolManager) ProtocolManagerStart(maxPeers int) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (pm *ProtocolManager) ProtocolManagerStart(maxPeers int) {")
    pm.maxPeers = maxPeers

    // broadcast transactions
    pm.txsCh = make(chan core.NewTxsEvent, txChanSize)
    pm.txsSub = pm.txpool.SubscribeNewTxsEvent(pm.txsCh)
    log.Info("[jpk] ProtocolManagerStart => ")
    go pm.txBroadcastLoop()

    // broadcast mined blocks
    pm.minedBlockSub = pm.eventMux.Subscribe(core.NewMinedBlockEvent{})
    log.Info("[jpk] ProtocolManagerStart => ")
    go pm.minedBroadcastLoop()

    // start sync handlers
    log.Info("[jpk] ProtocolManagerStart => ")
    go pm.syncer()
    log.Info("[jpk] ProtocolManagerStart => ")
    go pm.txsyncLoop()
}
```

Start(Ethereum)

ProtocolManagerStart

3-1. txBroadcastLoop

eth\backend.go

```go
func (pm *ProtocolManager) txBroadcastLoop() {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (pm *ProtocolManager) txBroadcastLoop() {")
    for {
        select {
        case event := <-pm.txsCh:
            log.Info("[jpk] txBroadcastLoop => case event := <-pm.txsCh:")
            pm.BroadcastTxs(event.Txs)

        // Err() channel will be closed when unsubscribing.
        case <-pm.txsSub.Err():
            log.Info("[jpk] txBroadcastLoop => case <-pm.txsSub.Err():")
            return
        }
    }
}
```

Start(Ethereum)

ProtocolManagerStart

3-2. minedBroadcastLoop

eth\backend.go

```go
// Mined broadcast loop
func (pm *ProtocolManager) minedBroadcastLoop() {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (pm *ProtocolManager) minedBroadcastLoop() {")
    // automatically stops if unsubscribe
    for obj := range pm.minedBlockSub.Chan() {
        if ev, ok := obj.Data.(core.NewMinedBlockEvent); ok {
            pm.BroadcastBlock(ev.Block, true)  // First propagate block to peers
            pm.BroadcastBlock(ev.Block, false) // Only then announce to the rest
        }
    }
}
```

Start(Ethereum)

ProtocolManagerStart

3-3. syncer

eth\backend.go

피어연결시 혹은 (10초)주기적으로 BestPeer와 sync를 맞춤

```go
// syncer is responsible for periodically synchronising with the network, both
// downloading hashes and blocks as well as handling the announcement handler.
func (pm *ProtocolManager) syncer() {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (pm *ProtocolManager) syncer() {")
    // Start and ensure cleanup of sync mechanisms
    pm.fetcher.FetcherStart()
    defer pm.fetcher.Stop()
    defer pm.downloader.Terminate()

    // Wait for different events to fire synchronisation operations
    forceSync := time.NewTicker(forceSyncCycle)
    defer forceSync.Stop()

    for {
        select {
        case <-pm.newPeerCh:
            log.Info("[jpk] syncer => case <-pm.newPeerCh:")
            // Make sure we have peers to select from, then sync
            if pm.peers.Len() < minDesiredPeerCount {
                break
            }
            go pm.synchronise(pm.peers.BestPeer())

        case <-forceSync.C:
            log.Info("[jpk] syncer => case <-forceSync.C:")
            // Force a sync even if not enough peers are present
            go pm.synchronise(pm.peers.BestPeer())

        case <-pm.noMorePeers:
```

Start(Ethereum)

syncer

3-3-1. synchronise

eth\sync.go

```go
// synchronise tries to sync up our local block chain with a remote peer.
func (pm *ProtocolManager) synchronise(peer *peer) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    if peer != nil {
        log.Info("[jpk] synchronise => ", "peer.id", peer.id)
    } else {
        log.Info("[jpk] synchronise => peer is nil")
    }
    log.Info("[jpk] func (pm *ProtocolManager) synchronise(peer *peer) {")
    // Short circuit if no peers are available
    if peer == nil {
        log.Info("[jpk] synchronise => if peer == nil {")
        return
    }
    // Make sure the peer's TD is higher than our own
    currentBlock := pm.blockchain.CurrentBlock()
    td := pm.blockchain.GetTd(currentBlock.Hash(), currentBlock.NumberU64())

    pHead, pTd := peer.Head()
    log.Info("[jpk] synchronise => ")
    if pTd.Cmp(td) <= 0 {
        log.Info("[jpk] synchronise => if pTd.Cmp(td) <= 0 {")
```

Start(Ethereum)

ProtocolManagerStart

3-4. txsyncLoop

eth\backend.go

```go
// txsyncLoop takes care of the initial transaction sync for each new
// connection. When a new peer appears, we relay all currently pending
// transactions. In order to minimise egress bandwidth usage, we send
// the transactions in small packs to one peer at a time.
func (pm *ProtocolManager) txsyncLoop() {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (pm *ProtocolManager) txsyncLoop() {")
    var (
        pending = make(map[enode.ID]*txsync)
        sending = false               // whether a send is active
        pack    = new(txsync)         // the pack that is being sent
        done    = make(chan error, 1) // result of the send
    )

    // send starts a sending a pack of transactions from the sync.
    send := func(s *txsync) {
        log.Info("[jpk] txsyncLoop => send := func(s *txsync) {")
        // Fill pack with transactions up to the target size.
        size := common.StorageSize(0)
        pack.p = s.p
        pack.txs = pack.txs[:0]
        for i := 0; i < len(s.txs) && size < txsyncPackSize; i++ {
```

StartNode
startRPC
node\node.go

StartNode
1. startRPC
node\node.go

```go
// startRPC is a helper method to start all the various RPC endpoint during node
// startup. It's not meant to be called at any time afterwards as it makes certain
// assumptions about the state of the node.
func (n *Node) startRPC(services map[reflect.Type]Service) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (n *Node) startRPC(services map[reflect.Type]Service) error {")
    // Gather all the possible APIs to surface
    apis := n.apis()
    for _, service := range services {
        apis = append(apis, service.APIs()...)
    }
    // Start the various API endpoints, terminating all in case of errors
    if err := n.startInProc(apis); err != nil {
        return err
    }
    if err := n.startIPC(apis); err != nil {
        n.stopInProc()
        return err
    }
    if err := n.startHTTP(n.httpEndpoint, apis, n.config.HTTPModules, n.config.HTTPCors,
    n.config.HTTPVirtualHosts, n.config.HTTPTimeouts); err != nil {
        n.stopIPC()
```

StartServer
runServer
p2p\server.go

StartServer
1. runServer
p2p\server.go

```go
func (srv *Server) runServer(dialstate dialer) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (srv *Server) run(dialstate dialer) {")
    srv.log.Info("Started P2P networking", "self", srv.localnode.Node())
    defer srv.loopWG.Done()
    defer srv.nodedb.Close()

    var (
        peers        = make(map[enode.ID]*Peer)
        inboundCount = 0
        trusted      = make(map[enode.ID]bool, len(srv.TrustedNodes))
        taskdone     = make(chan task, maxActiveDialTasks)
        runningTasks []task
        queuedTasks  []task // tasks that can't run yet
    )
    // Put trusted nodes into a map to speed up checks.
    // Trusted peers are loaded on startup or added via AddTrustedPeer RPC.
    for _, n := range srv.TrustedNodes {
        trusted[n.ID()] = true
    }

    // removes t from runningTasks
    delTask := func(t task) {
        for i := range runningTasks {
            if runningTasks[i] == t {
                runningTasks = append(runningTasks[:i], runningTasks[i+1:]...)
```

StartServer
1. runServer
p2p\server.go

```
// starts until max number of active tasks is satisfied
startTasks := func(ts []task) (rest []task) {
    log.Info("[jpk] run => startTasks := func(ts []task) (rest []task) {")
    i := 0
    for ; len(runningTasks) < maxActiveDialTasks && i < len(ts); i++ {
        t := ts[i]
        srv.log.Trace("New dial task ", "task", t)
        go func() {
            log.Info("[jpk] run => go func() {")
            t.Do(srv)
            taskdone <- t
        }()
        runningTasks = append(runningTasks, t)
    }
    return ts[i:]
}
scheduleTasks := func() {
    // log.Info("[jpk] run => scheduleTasks := func() {")
    // Start from queue first.
    queuedTasks = append(queuedTasks[:0], startTasks(queuedTasks)...)
    // Query dialer for new tasks and start as many as possible now.
    if len(runningTasks) < maxActiveDialTasks {
        // log.Info("[jpk] run => scheduleTasks := func() {")
        nt := dialstate.newTasks(len(runningTasks)+len(queuedTasks), peers, time.Now())
        queuedTasks = append(queuedTasks, startTasks(nt)...)
```

runServer

runServer

1-1. Do

p2p\dial.go

```go
func (t *dialTask) Do(srv *Server) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (t *dialTask) Do(srv *Server) {")
    if t.dest.Incomplete() {
        if !t.resolve(srv) {
            return
        }
    }
    err := t.dial(srv, t.dest)
    if err != nil {
        log.Trace("Dial error", "task", t, "err", err)
        // Try resolving the ID of static nodes if dialing failed.
        if _, ok := err.(*dialError); ok && t.flags&staticDialedConn != 0 {
            if t.resolve(srv) {
                log.Info("[jpk] Do => if t.resolve(srv) {")
                t.dial(srv, t.dest)
            }
        }
    }
}
```

runServer

Do

1-2. dial

p2p\dial.go

```go
// dial performs the actual connection attempt.
func (t *dialTask) dial(srv *Server, dest *enode.Node) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] dial => ", "dest.ID()", dest.ID())
    log.Info("[jpk] dial => ", "dest.IP()", dest.IP(), "dest.TCP()", dest.TCP())
    log.Info("[jpk] func (t *dialTask) dial(srv *Server, dest *enode.Node) error {")
    fd, err := srv.Dialer.Dial(dest)
    if err != nil {
        return &dialError{err}
    }
    mfd := newMeteredConn(fd, false, dest.IP())
    log.Info("[jpk] dial => ")
    return srv.SetupConn(mfd, t.flags, dest)
}
```

runServer

dial

1-3. SetupConn

p2p\server.go

```go
// SetupConn runs the handshakes and attempts to add the connection
// as a peer. It returns when the connection has been added as a peer
// or the handshakes have failed.
func (srv *Server) SetupConn(fd net.Conn, flags connFlag, dialDest *enode.Node) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] SetupConn => ", "fd.RemoteAddr()", fd.RemoteAddr())
    if dialDest != nil {
        log.Info("[jpk] SetupConn => ", "dialDest.ID()", dialDest.ID())
    }
    log.Info("[jpk] func (srv *Server) SetupConn(fd net.Conn, flags connFlag, dialDest *enode.Node)
    error {")
    c := &conn{fd: fd, transport: srv.newTransport(fd), flags: flags, cont: make(chan error)}
    err := srv.setupConn(c, flags, dialDest)
    if err != nil {
        c.close(err)
        srv.log.Trace("Setting up connection failed", "addr", fd.RemoteAddr(), "err", err)
    }
    return err
}
```

runServer

SetupConn

1-4. setupConn

p2p\server.go

```
func (srv *Server) setupConn(c *conn, flags connFlag, dialDest *enode.Node) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    if dialDest != nil {
        log.Info("[jpk] setupConn => ", "dialDest.ID", dialDest.ID())
    }
    log.Info("[jpk] setupConn => ", "c.fd.RemoteAddr()", c.fd.RemoteAddr())
    log.Info("[jpk] func (srv *Server) setupConn(c *conn, flags connFlag, dialDest *enode.Node)
    error {")
    // Prevent leftover pending conns from entering the handshake.
    srv.lock.Lock()
    running := srv.running
    srv.lock.Unlock()
    if !running {
        return errServerStopped
    }
    // If dialing, figure out the remote public key.
    var dialPubkey *ecdsa.PublicKey
    if dialDest != nil {
        log.Info("[jpk] setupConn => if dialDest != nil {")
        dialPubkey = new(ecdsa.PublicKey)
        if dialPubkey != nil {
            log.Info("[jpk] setupConn => ", "dialPubkey", dialPubkey)
        }
```

runServer

SetupConn

1-4. setupConn

p2p\server.go

```
    err = srv.checkpoint(c, srv.posthandshake)
    if err != nil {
        clog.Trace("Rejected peer before protocol handshake", "err", err)
        return err
    }
    // Run the protocol handshake
    phs, err := c.doProtoHandshake(srv.ourHandshake)
    if err != nil {
        clog.Trace("Failed proto handshake", "err", err)
        return err
    }
    if id := c.node.ID(); !bytes.Equal(crypto.Keccak256(phs.ID), id[:]) {
        clog.Trace("Wrong devp2p handshake identity", "phsid", hex.EncodeToString(phs.ID))
        return DiscUnexpectedIdentity
    }

    if err := c.doSporkHandshake_test(srv.ourSporkHandshake); err != nil {
        clog.Trace("Failed proto handshake", "err", err)
        return err
    }

    // jpk
    c.caps, c.name, c.crp = phs.Caps, phs.Name, phs.Crp
    err = srv.checkpoint(c, srv.addpeer)
    if err != nil {
```

runServer

setupConn

1-5. checkpoint

p2p\server.go

```go
// checkpoint sends the conn to run, which performs the
// post-handshake checks for the stage (posthandshake, addpeer).
func (srv *Server) checkpoint(c *conn, stage chan<- *conn) error {
    select {
    case stage <- c:
        log.Info("[jpk] checkpoint => case stage <- c:")
    case <-srv.quit:
        log.Info("[jpk] checkpoint => case <-srv.quit:")
        return errServerStopped
    }
    select {
    case err := <-c.cont:
        log.Info("[jpk] checkpoint => case err := <-c.cont:")
        return err
    case <-srv.quit:
        log.Info("[jpk] checkpoint => case <-srv.quit:")
        return errServerStopped
    }
}
```

StartServer
2. runServer
p2p\server.go

연결이 성공해서 새로운 피어객체를 생성한다.

```
        case c := <-srv.addpeer:
            log.Info("[jpk] run => case c := <-srv.addpeer:")
            // At this point the connection is past the protocol handshake.
            // Its capabilities are known and the remote identity is verified.
            err := srv.protoHandshakeChecks(peers, inboundCount, c)
            if err == nil {
                log.Info("[jpk] run => if err == nil {")
                // The handshakes are done and it passed all checks.
                p := newPeer(c, srv.Protocols)
                // If message events are enabled, pass the peerFeed
                // to the peer
                if srv.EnableMsgEvents {
                    p.events = &srv.peerFeed
                }
                name := truncateName(c.name)
                srv.log.Debug("Adding p2p peer", "name", name, "addr", c.fd.RemoteAddr(), "peers",
                len(peers)+1)
                log.Info("[jpk] run => ")
                go srv.runPeer(p)
                peers[c.node.ID()] = p
                if p.Inbound() {
                    inboundCount++
                }
            }
            // The dialer logic relies on the assumption that
            // dial tasks complete after the peer has been added or
            // discarded. Unblock the task last.
            select {
            case c.cont <- err:
            case <-srv.quit:
                break running
```

runServer

runServer
2-1. newPeer
p2p\server.go

```go
func newPeer(conn *conn, protocols []Protocol) *Peer {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] newPeer => ", "conn.fd.RemoteAddr()", conn.fd.RemoteAddr())
    log.Info("[jpk] func newPeer(conn *conn, protocols []Protocol) *Peer {")
    protomap := matchProtocols(protocols, conn.caps, conn)
    p := &Peer{
        rw:       conn,
        running:  protomap,
        created:  mclock.Now(),
        disc:     make(chan DiscReason),
        protoErr: make(chan error, len(protomap)+1), // protocols + pingLoop
        closed:   make(chan struct{}),
        log:      log.New("id", conn.node.ID(), "conn", conn.flags),
    }
    return p
}
```

runServer
runPeer
p2p\server.go

runPeer
1. runPeer
p2p\server.go

```go
// runPeer runs in its own goroutine for each peer.
// it waits until the Peer logic returns and removes
// the peer.
func (srv *Server) runPeer(p *Peer) {
    if srv.newPeerHook != nil {
        srv.newPeerHook(p)
    }

    // broadcast peer add
    srv.peerFeed.Send(&PeerEvent{
        Type: PeerEventTypeAdd,
        Peer: p.ID(),
    })

    // run the protocol
    remoteRequested, err := p.run()

    // broadcast peer drop
    srv.peerFeed.Send(&PeerEvent{
        Type:  PeerEventTypeDrop,
        Peer:  p.ID(),
        Error: err.Error(),
    })

    // Note: run waits for existing peers to be sent on srv.delpeer
    // before returning, so this send should not select on srv.quit.
    // log.Info("[jpk] runPeer => ")
    srv.delpeer <- peerDrop{p, err, remoteRequested}
}
```

runPeer

runPeer

1-1. run

p2p\peer.go

```go
func (p *Peer) run() (remoteRequested bool, err error) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] run => ", "p.rw.fd.RemoteAddr()", p.rw.fd.RemoteAddr())
    log.Info("[jpk] func (p *Peer) run() (remoteRequested bool, err error) {")
    var (
        writeStart = make(chan struct{}, 1)
        writeErr   = make(chan error, 1)
        readErr    = make(chan error, 1)
        reason     DiscReason // sent to the peer
    )
    p.wg.Add(2)
    go p.readLoop(readErr)
    go p.pingLoop()

    // Start all protocol handlers.
    writeStart <- struct{}{}
    p.startProtocols(writeStart, writeErr)

    // Wait for an error or disconnect.
loop:
    for {
        select {
        case err = <-writeErr:
```

연결된 피어와의 채널을 통해서 Msg를 받는다

runPeer

run

1-1-1. readLoop

p2p\peer.go

```go
func (p *Peer) readLoop(errc chan<- error) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (p *Peer) readLoop(errc chan<- error) {")
    defer p.wg.Done()
    for {
        msg, err := p.rw.ReadMsg()
        if err != nil {
            errc <- err
            return
        }
        msg.ReceivedAt = time.Now()
        log.Info("[jpk] readLoop => ")
        if err = p.handle(msg); err != nil {
            errc <- err
            return
        }
    }
}
```

연결된 피어와의 채널을 통해 Ping을 (5초)주기적으로 보낸다

runPeer

run

1-1-2. pingLoop

p2p\peer.go

```
func (p *Peer) pingLoop() {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (p *Peer) pingLoop() {")
    ping := time.NewTimer(pingInterval)
    defer p.wg.Done()
    defer ping.Stop()
    for {
        select {
        case <-ping.C:
            log.Info("[jpk] pingLoop => case <-ping.C:", "p.ID()", p.ID(), "p.RemoteAddr().String()
            ", p.RemoteAddr().String())
            if err := SendItems(p.rw, pingMsg); err != nil {
                p.protoErr <- err
                return
            }
            ping.Reset(pingInterval)
        case <-p.closed:
            log.Info("[jpk] pingLoop => case <-p.closed:", "p.ID()", p.ID(), "p.RemoteAddr().String
            ()", p.RemoteAddr().String())
            return
        }
    }
}
```

runPeer

run

1-2. startProtocols

p2p\peer.go

```go
func (p *Peer) startProtocols(writeStart <-chan struct{}, writeErr chan<- error) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (p *Peer) startProtocols(writeStart <-chan struct{}, writeErr chan<- error)
     {")
    p.wg.Add(len(p.running))
    for _, proto := range p.running {
        // log.Info("[jpk] startProtocols => for _, proto := range p.running {", "index", index)
        proto := proto
        proto.closed = p.closed
        proto.wstart = writeStart
        proto.werr = writeErr
        var rw MsgReadWriter = proto
        if p.events != nil {
            rw = newMsgEventer(rw, p.events, p.ID(), proto.Name)
        }
        p.log.Trace(fmt.Sprintf("Starting protocol %s/%d", proto.Name, proto.Version))
        go func() {
            log.Info("[jpk] startProtocols => go func() {")
            err := proto.Run(p, rw)
            if err == nil {
                p.log.Trace(fmt.Sprintf("Protocol %s/%d returned", proto.Name, proto.Version))
                err = errProtocolReturned
            } else if err != io.EOF {
                p.log.Trace(fmt.Sprintf("Protocol %s/%d failed", proto.Name, proto.Version), "err",
                err)
            }
            p.protoErr <- err
            p.wg.Done()
```

실제 프로토콜 통신 담당

runPeer

startProtocols

1-2-1. run(in NewProtocolManager)

eth\handler.go

```
Run: func(p *p2p.Peer, rw p2p.MsgReadWriter) error {
    log.Info("[jpk] NewProtocolManager => func(p *p2p.Peer, rw p2p.MsgReadWriter) error
    {")
    peer := manager.newPeer(int(version), p, rw)
    select {
    case manager.newPeerCh <- peer:
        log.Info("[jpk] NewProtocolManager => case manager.newPeerCh <- peer:")
        manager.wg.Add(1)
        defer manager.wg.Done()
        return manager.handle(peer)
    case <-manager.quitSync:
        return p2p.DiscQuitting
    }
},
```

run(in NewProtocolManager)
handle
eth\handler.go

run(in NewProtocolManager)
1. handle
eth\handler.go

```
// handle is the callback invoked to manage the life cycle of an eth peer. When
// this function terminates, the peer is disconnected.
func (pm *ProtocolManager) handle(p *peer) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] handle => ", "p.id", p.id)
    log.Info("[jpk] func (pm *ProtocolManager) handle(p *peer) error {")
    // Ignore maxPeers if this is a trusted peer
    if pm.peers.Len() >= pm.maxPeers && !p.Peer.Info().Network.Trusted {
        log.Info("[jpk] handle => if pm.peers.Len() >= pm.maxPeers && !p.Peer.Info()
        .Network.Trusted {")
        return p2p.DiscTooManyPeers
    }
    log.Info("[jpk] handle => Ethereum peer connected", "p.id", p.id, "p.Name()", p.Name())
    p.Log().Debug("Ethereum peer connected", "name", p.Name())

    // Execute the Ethereum handshake
    var (
        genesis = pm.blockchain.Genesis()
        head    = pm.blockchain.CurrentHeader()
        hash    = head.Hash()
        number  = head.Number.Uint64()
        td      = pm.blockchain.GetTd(hash, number)
```

handle

handle

2. Handshake

eth\peer.go

StatusMsg를 보냄

```go
// Handshake executes the eth protocol handshake, negotiating version number,
// network IDs, difficulties, head and genesis blocks.
func (p *peer) Handshake(network uint64, td *big.Int, head common.Hash, genesis common.Hash) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (p *peer) Handshake(network uint64, td *big.Int, head common.Hash, genesis
    common.Hash) error {")
    // Send out own handshake in a new thread
    errc := make(chan error, 2)
    var status statusData // safe to read after two values have been received from errc

    go func() {
        log.Info("[jpk] Handshake => go func() {")
        errc <- p2p.Send(p.rw, StatusMsg, &statusData{
            ProtocolVersion: uint32(p.version),
            NetworkId:       network,
            TD:              td,
            CurrentBlock:    head,
            GenesisBlock:    genesis,
        })
    }()
    go func() {
        log.Info("[jpk] Handshake => go func() {")
        errc <- p.readStatus(network, &status, genesis)
    }()
    timeout := time.NewTimer(handshakeTimeout)
    defer timeout.Stop()
    for i := 0; i < 2; i++ {
        select {
        case err := <-errc:
```

handle

handle
3. Register
eth\peer.go

```go
// Register injects a new peer into the working set, or returns an error if the
// peer is already known. If a new peer it registered, its broadcast loop is also
// started.
func (ps *peerSet) Register(p *peer) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] Register => ", "p.id", p.id)
    log.Info("[jpk] func (ps *peerSet) Register(p *peer) error {")
    ps.lock.Lock()
    defer ps.lock.Unlock()

    if ps.closed {
        // log.Info("[jpk] Register => if ps.closed {")
        return errClosed
    }
    if _, ok := ps.peers[p.id]; ok {
        // log.Info("[jpk] Register => if _, ok := ps.peers[p.id]; ok {")
        return errAlreadyRegistered
    }
    // log.Info("[jpk] Register => ", "p.id", p.id)
    ps.peers[p.id] = p
    go p.broadcast()

    return nil
}
```

handle

Register

3-1. broadcast

eth\peer.go

노드에 tx, block이 생기면 broadcast함

```go
// broadcast is a write loop that multiplexes block propagations, announcements
// and transaction broadcasts into the remote peer. The goal is to have an async
// writer that does not lock up node internals.
func (p *peer) broadcast() {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] func (p *peer) broadcast() {")
    for {
        select {
        case txs := <-p.queuedTxs:
            if err := p.SendTransactions(txs); err != nil {
                return
            }
            p.Log().Trace("Broadcast transactions", "count", len(txs))

        case prop := <-p.queuedProps:
            if err := p.SendNewBlock(prop.block, prop.td); err != nil {
                return
            }
            p.Log().Trace("Propagated block", "number", prop.block.Number(), "hash", prop.block.Hash
            ().Hex(), "td", prop.td)

        case block := <-p.queuedAnns:
            if err := p.SendNewBlockHashes([]common.Hash{block.Hash()}, []uint64{block.NumberU64()})
            ; err != nil {
                return
            }
            p.Log().Trace("Announced block", "number", block.Number(), "hash", block.Hash().Hex())

        case <-p.term:
            return
        }
```

노드의 tx들을 바탕으로 새로이 연결된 피어와 tx싱크를 맞춤

handle

handle

4. syncTransactions

eth\peer.go

```go
// syncTransactions starts sending all currently pending transactions to the given peer.
func (pm *ProtocolManager) syncTransactions(p *peer) {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] syncTransactions => ", "p.id", p.id)
    log.Info("[jpk] func (pm *ProtocolManager) syncTransactions(p *peer) {")
    var txs types.Transactions
    pending, _ := pm.txpool.Pending()
    for _, batch := range pending {
        txs = append(txs, batch...)
    }
    if len(txs) == 0 {
        return
    }
    select {
    case pm.txsyncCh <- &txsync{p, txs}:
        log.Info("[jpk] syncTransactions => case pm.txsyncCh <- &txsync{p, txs}:")
    case <-pm.quitSync:
        log.Info("[jpk] syncTransactions => case <-pm.quitSync:")
    }
}
```

handle
handleMsg
eth\handler.go

Msg를 수신 후 실제 처리함

handle
1. handleMsg
eth\handler.go

```
// handleMsg is invoked whenever an inbound message is received from a remote
// peer. The remote connection is torn down upon returning any error.
func (pm *ProtocolManager) handleMsg(p *peer) error {
    log.Info("[jpk] ")
    log.Info("[jpk] ")
    log.Info("[jpk] handleMsg => ", "p.id", p.id)
    log.Info("[jpk] func (pm *ProtocolManager) handleMsg(p *peer) error {")
    // Read the next message from the remote peer, and ensure it's fully consumed
    msg, err := p.rw.ReadMsg()
    log.Info("[jpk] handleMsg => ")
    if err != nil {
        log.Info("[jpk] handleMsg => if err != nil {", "err", err)
        return err
    }
    if msg.Size > ProtocolMaxMsgSize {
        log.Info("[jpk] handleMsg => if msg.Size > ProtocolMaxMsgSize {")
        return errResp(ErrMsgTooLarge, "%v > %v", msg.Size, ProtocolMaxMsgSize)
    }
    defer msg.Discard()

    // Handle the message depending on its contents
    switch {
    case msg.Code == StatusMsg:
        log.Info("[jpk] handleMsg => case msg.Code == StatusMsg:")
```

Loop in worker

```
go worker.mainLoop()
go worker.newWorkLoop(recommit)
go worker.resultLoop()
go worker.taskLoop()
```

Loop in srv

```
go srv.listenLoop()
```

Loop in udp

```
go udp.loop()
go udp.readLoop(cfg.Unhandled)
```

Export PrivateKey From File

```
var keythereum = require("keythereum");
var datadir = "/home/pilkyu/.ethereum";
var address= process.argv[2];
const password = process.argv[3];

var keyObject = keythereum.importFromFile(address, datadir);
var privateKey = keythereum.recover(password, keyObject);
console.log(privateKey.toString('hex'));
~
"test.js" 8L, 305C written                                    1,1           All
pilkyu@pilkyu-ForBitcoind:~$ node test.js "0x847561a80a4e0694afeb6c28a2570a18eb47ba85" "1"
58a37c730829d8b87cfa9689c476b8e4be6131af7650a7f80cadade99ca745af
pilkyu@pilkyu-ForBitcoind:~$
```

Import Account by PrivateKey

```
> personal.importRawKey("58a37c730829d8b87cfa9689c476b8e4be6131af7650a7f80cadade99ca745af","1")
INFO [01-08|19:04:12.853] func (b *bridge) Send(call otto.FunctionCall) (response otto.Value) {
INFO [01-08|19:04:12.853] func (c *Client) Call(result interface{}, method string, args ...inter
face{}) error {
INFO [01-08|19:04:12.854] func (c *Client) CallContext(ctx context.Context, result interface{},
method string, args ...interface{}) error {
INFO [01-08|19:04:12.854] func (c *Client) newMessage(method string, paramsIn ...interface{}) (*
jsonrpcMessage, error) {
INFO [01-08|19:04:12.854] newMessage.....................................................|method metho
d=personal_importRawKey
INFO [01-08|19:04:12.854] func (c *Client) sendHTTP(ctx context.Context, op *requestOp, msg inte
rface{}) error {
INFO [01-08|19:04:14.363] func (op *requestOp) wait(ctx context.Context) (*jsonrpcMessage, error
) { op.ids="[[51 55 50]]"
INFO [01-08|19:04:14.363] CallContext##############################################|default:
"0x847561a80a4e0694afeb6c28a2570a18eb47ba85"
```

Docker

-removing image
docker rmi [options] <"image">

-removing container
docker rm [options] <"container name" or ID>

-show current images
docker images

-show containers
docker ps [options]

-show status of container
docker stats <"container name" or ID>

-show process in container
docker top <"container name" or ID>

-show port in container
docker port <"container name" or ID>

-show different
docker diff <"container name" or ID>

Docker

-creating container & run
docker run [options] <"image">

-starting container
docker start [options] <"container name" or ID>

-stoping container
docker stop [options] <"container name" or ID>

-rebooting container
docker restart [options] <"container name" or ID>

-pause container
docker pause <"container name" or ID>

-attaching container
docker attach <"container name" or ID>

Docker

-modify container's name
docker rename <Current> <After>

-copying to container or host
docker cp <"container name" or ID>:<path in container> <path in host>
docker cp <path in host> <"container name" or ID>:<path in container>

-making image
docker commit [options] <"container name" or ID>

-saving to tar file
docker export <"container name" or ID>