

Go-ethereum 씹어먹기

세미나 #1

박정원(Aiden)

aiden.p@onther.io

18/09/14



<외부자료 활용 시 출처 명시 부탁드립니다.>

오늘 다룰 내용

func geth() 의 주요 로직인 **makeFullNode()** 와 **startNode()** 의 실행로직 구체적으로 살펴보기

이 과정에서 cmd 패키지, node 패키지의 대부분을 자연스럽게 다루게 된다.

역시 *depth*를 너무 파고들기보다는 적정한 수준에서 코드의 전체 로직을 이해할 수 있게 정리하였음

Reference

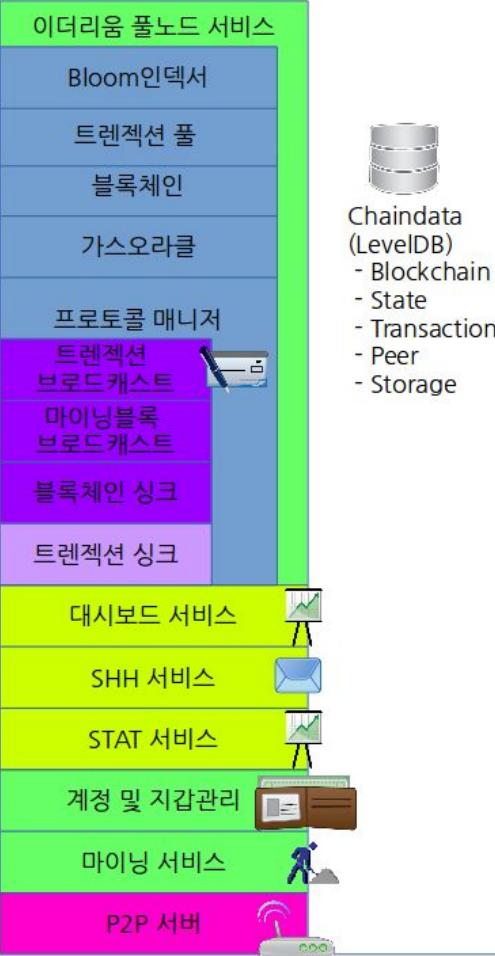
<https://steemit.com/@sigmoid>

<https://github.com/NAKsir-melody/go-ethereum>

<http://www.notforme.kr/block-chain/geth-code-reading>

자료를 정리하는데 **sigmoid** 님과 **woojin.joe**님의 글들이 큰 도움이 되었습니다. 감사합니다.

GETH 노드



우선 **geth**는 이더리움 서비스 및 그 외 통계적 서비스, 그리고 지갑(어카운트 매니저), 마이닝, p2p 노드 역할을 수행합니다. 그래서 **geth**는 지갑이자, 노드입니다.

이더리움은 **geth** 노드에서 동작하는 이더리움 프로토콜과 이더리움 서비스들로 모임입니다. 말이 좀 이상한데 서비스란 하나의 기능을 담당하는 것이고, 프로토콜이란 서비스간의 연계작이라고 생각하시면, 조금 머릿속이 편안해 질 것입니다.

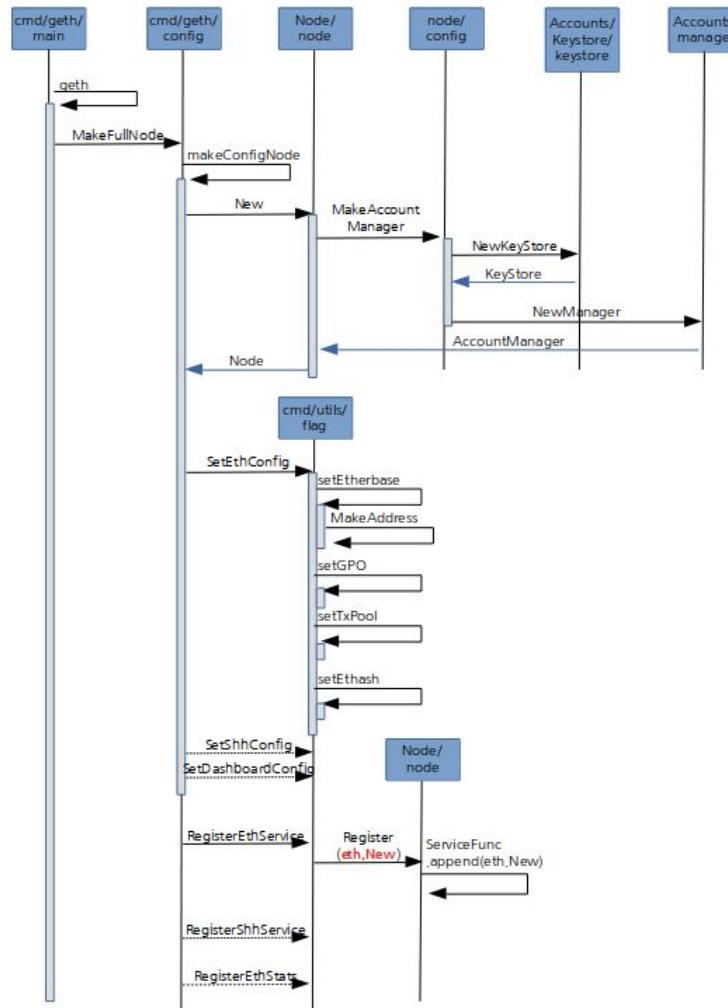
먼저 서비스들을 살펴보면, 블룸 인덱서는 엄청나게 많은 트렌젝션을 빠른시간안에 조회하기 위해 사용됩니다

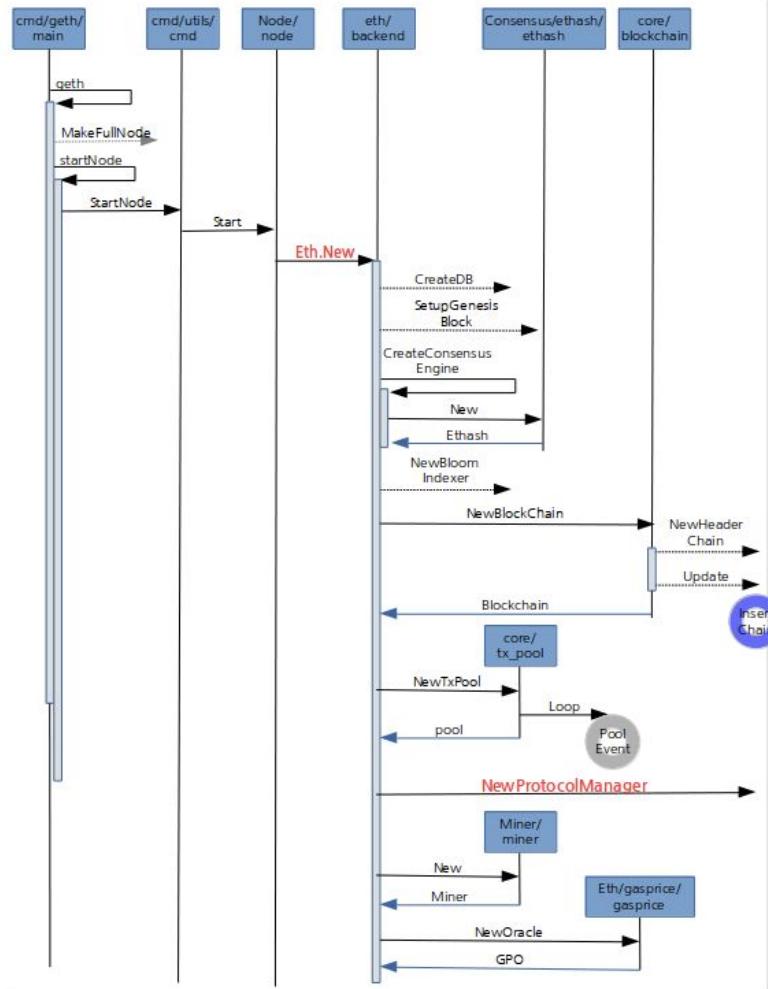
트렌젝션 풀 과 블록체인은 각각 채굴되지 않은 블록에 포함될 후보 트렌젝션들을 담아두는 역할과 블록체인 관련 합의나 데이터 쓰기등의 역할을 합니다.

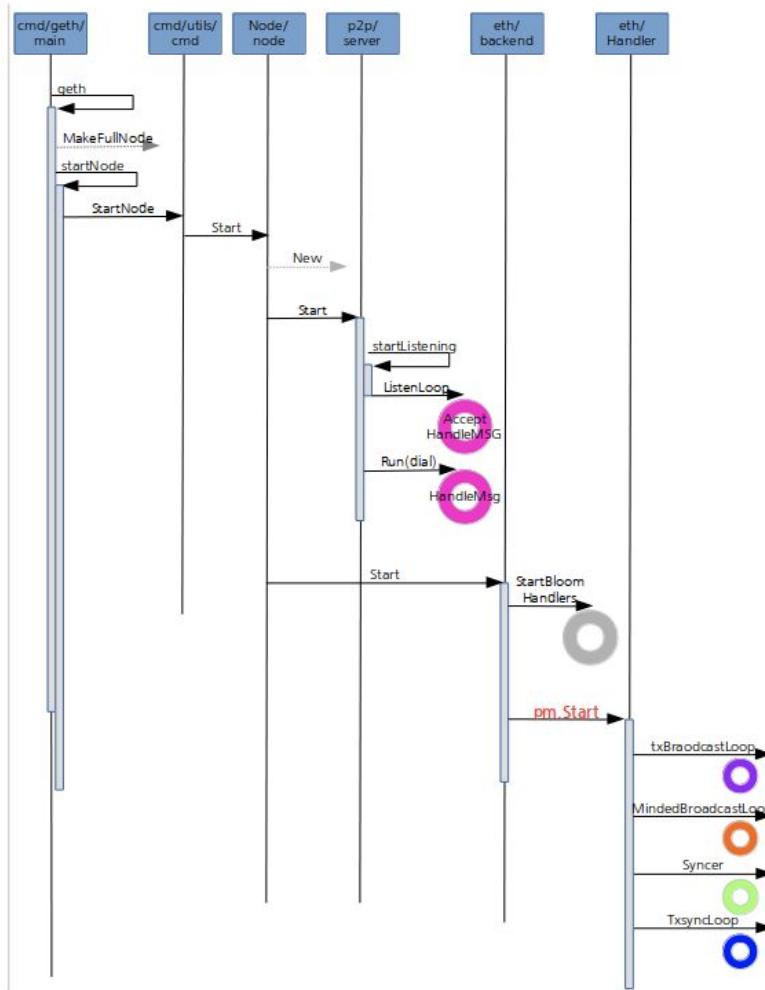
해당내용들은 DB에 저장됩니다. 가스오라클은 가스비 정책을 정하기위한 서비스입니다. 최근 네트워크의 가스비 추세를 잘 기억해두었다가 반영한다던지...

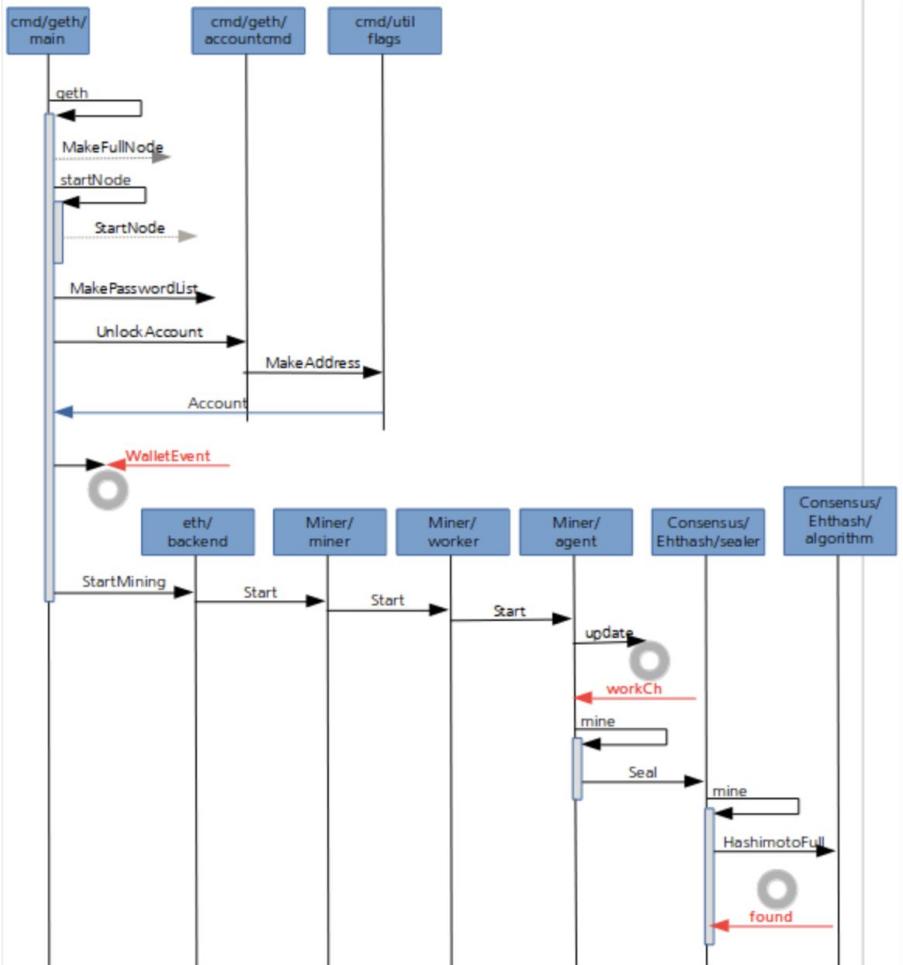
이더리움 프로토콜을 담당하는 프로토콜 매니저는 지갑서비스의 생산물(트렌젝션)을 트렌젝션 풀 서비스의 입력으로 전달하는 부분을 담당하거나, 마이닝 서비스의 생산물을 p2p서비스의 인풋으로 전달하여 최종적으로 블록체인 서비스(합의/db쓰기 등등)의 행동결과를 유발하는 일을 합니다.

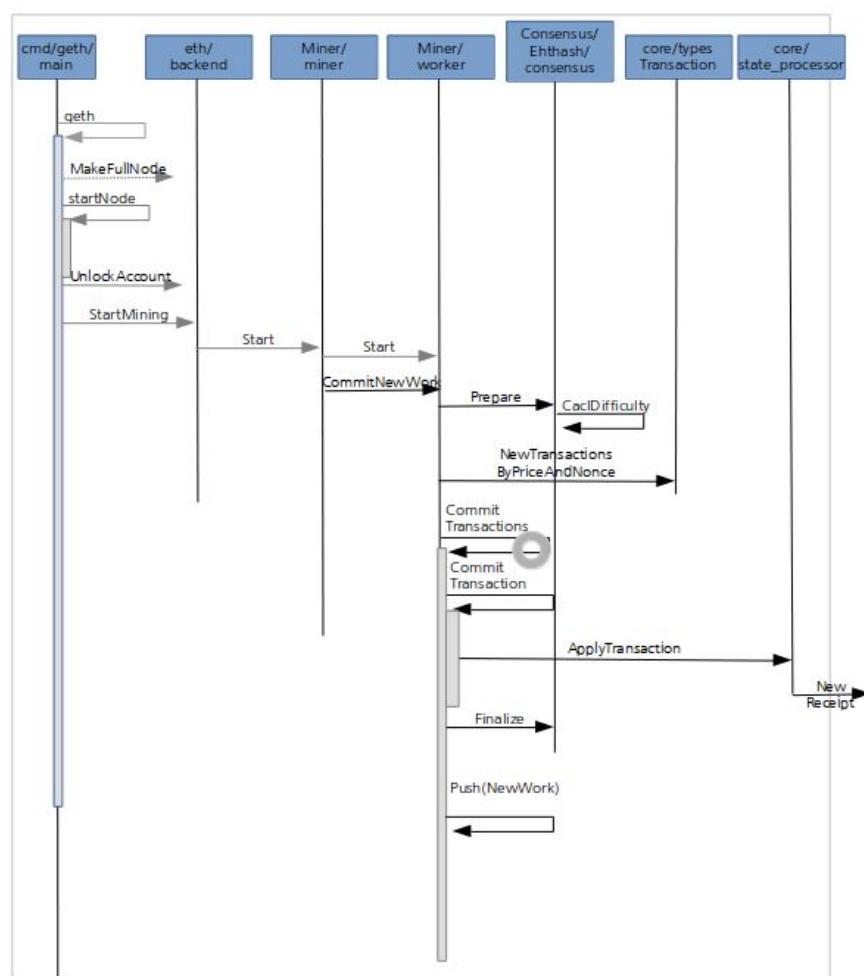
빨간색으로 표시된 부분을 기억하는 것이 매우매우 중요!!
 계속 저 말의 의미와 왼쪽의 그림을 머릿속으로 그리면서 가면 이해에 많은 도움된다.











1. makeFullNode()

cmd/geth/main.go | func main()



```
247 ►  func main() {
248     if err := app.Run(os.Args); err != nil {
249         fmt.Fprintln(os.Stderr, err)
250         os.Exit(code: 1)
251     }
252 }
```

main()함수가 하는 역할은 app.Run()함수를 실행하여 app.Action에 해당하는 func geth()를 실행하는 것

cmd/geth/main.go | func main()



```
// NewApp creates an app with sane defaults.  
func NewApp(gitCommit, usage string) *cli.App {  
    app := cli.NewApp()  
    app.Name = filepath.Base(os.Args[0])  
    app.Author = ""  
    //app.Authors = nil  
    app.Email = ""  
    app.Version = params.VersionWithMeta  
    if len(gitCommit) >= 8 {  
        app.Version += "-" + gitCommit[:8]  
    }  
    app.Usage = usage  
    return app  
}
```

결국 app은 *cli.App 타입의 변수
오른쪽과 같은 여러 필드를 갖고 있음
중요한건 Commands와 Flags 필드
예를 들어, 아래처럼 version을 커맨드로 주면
해당 로직이 실행되는 것임

```
versionCommand = cli.Command{  
    Action:    utils.MigrateFlags(version),  
    Name:     "version",  
    Usage:    "Print version numbers",  
    ArgsUsage: "",  
    Category:  "MISCELLANEOUS COMMANDS",  
    Description: `
```

```
// App is the main structure of a cli application. It is recommended  
// an app be created with the cli.NewApp() function  
type App struct {  
    // The name of the program. Defaults to path.Base(os.Args[0])  
    Name string  
    // Full name of command for help, defaults to Name  
    HelpName string  
    // Description of the program.  
    Usage string  
    // Text to override the USAGE section of help  
    UsageText string  
    // Description of the program argument format.  
    ArgsUsage string  
    // Version of the program  
    Version string  
    // Description of the program  
    Description string  
    // List of commands to execute  
    Commands []Command  
    // List of flags to parse  
    Flags []Flag  
    // Boolean to enable bash completion commands  
    ...
```

cmd/geth/main.go | func geth()



Onther Inc.

```
257     func geth(ctx *cli.Context) error {
258         if args := ctx.Args(); len(args) > 0 {
259             return fmt.Errorf("invalid command: %q", args[0])
260         }
261         node := makeFullNode(ctx)
262         startNode(ctx, node)
263         node.Wait()
264         return nil
265     }
```

func geth() 함수만 제대로 봐도 geth 다 본 것 !
하지만.... depth의 압박이....

계속 언급했듯이 func geth()의 로직은 크게
`makeFullNode()` 와
`startNode()`로 나뉜다.

먼저 `makeFullNode()`부터 살펴보자

```
type Context struct {
    App          *App
    Command      Command
    shellComplete bool
    flagSet       *flag.FlagSet
    setFlags     map[string]bool
    parentContext *Context
}
```

geth함수의 파라미터 ctx는 `*cli.Context`타입이다. 내부 로직을 자세히 살펴보지 않아도 터미널에 입력한 명령을 파싱한 결과와 사전에 등록된 Command와 Flag정보등이 포함되어 있다는 것을 추측할 수 있다.

cmd/geth/config.go | func makeFullNode()



```
153     func makeFullNode(ctx *cli.Context) *node.Node {
154         stack, cfg := makeConfigNode(ctx)
155
156         utils.RegisterEthService(stack, &cfg.Eth)
157
158         if ctx.GlobalBool(utils.DashboardEnabledFlag.Name) {
159             utils.RegisterDashboardService(stack, &cfg.Dashboard, gitCommit)
160         }
161         // Whisper must be explicitly enabled by specifying at least 1 whisper flag or in dev mode
162         shhEnabled := enableWhisper(ctx)
163         shhAutoEnabled := !ctx.GlobalIsSet(utils.WhisperEnabledFlag.Name) && ctx.GlobalIsSet(utils.DeveloperFlag.Name)
164         if shhEnabled || shhAutoEnabled {
165             if ctx.GlobalIsSet(utils.WhisperMaxMessageSizeFlag.Name) {
166                 cfg.Shh.MaxMessageSize = uint32(ctx.Int(utils.WhisperMaxMessageSizeFlag.Name))
167             }
168             if ctx.GlobalIsSet(utils.WhisperMinPOWFlag.Name) {
169                 cfg.Shh.MinimumAcceptedPOW = ctx.Float64(utils.WhisperMinPOWFlag.Name)
170             }
171             utils.RegisterShhService(stack, &cfg.Shh)
172         }
173         // Add the Ethereum Stats daemon if requested.
174         if cfg.Ethstats.URL != "" {
175             utils.RegisterEthStatsService(stack, cfg.Ethstats.URL)
176         }
177     }
178     return stack
179 }
```

본격적으로 함수 내부를 살펴보자. 로직이 여러가지 있지만 빨간색으로 표시한 `makeConfigNode()`와 `RegisterEthService()`를 제외한 나머지는 크게 중요하지 않다 (Dashboard, whisper, Stats 관련 설정이다)

내부로 깊이 파고들기 전에 먼저 중요한 사실 하나를 꼭 기억하고 가야한다.

이 함수의 실행결과 리턴하는 변수는 `*node.Node` 즉 `Node` 타입이다.

안에서 뭘 지하고 봄든지 무조건 리턴값은 `Node`타입의 변수 `stack`이라는 것이다.

잠깐 `Node`타입이 도대체 뭔지 살짝 보고가자

```

39 // Node is a container on which services can be registered.
40 type Node struct {
41     eventmux *event.TypeMux // Event multiplexer used between the services of a stack
42     config   *Config
43     accman   *accounts.Manager
44
45     ephemeralKeystore string          // if non-empty, the key directory that will be removed by Stop
46     instanceDirLock    flock.Releaser // prevents concurrent use of instance directory
47
48     serverConfig p2p.Config
49     server       *p2p.Server // Currently running P2P networking layer
50
51     serviceFuncs []ServiceConstructor // Service constructors (in dependency order)
52     services      map[reflect.Type]Service // Currently running services
53
54     rpcAPIs        []rpc.API // List of APIs currently provided by the node
55     inprocHandler  *rpc.Server // In-process RPC request handler to process the API requests
56
57     ipcEndpoint string      // IPC endpoint to listen at (empty = IPC disabled)
58     ipcListener  net.Listener // IPC RPC listener socket to serve API requests
59     ipcHandler   *rpc.Server // IPC RPC request handler to process the API requests
60
61     httpEndpoint string      // HTTP endpoint (interface + port) to listen at (empty = HTTP disabled)
62     httpWhitelist []string    // HTTP RPC modules to allow through this endpoint
63     httpListener  net.Listener // HTTP RPC listener socket to serve API requests
64     httpHandler   *rpc.Server // HTTP RPC request handler to process the API requests
65
66     wsEndpoint string      // Websocket endpoint (interface + port) to listen at (empty = websocket disabled)
67     wsListener   net.Listener // Websocket RPC listener socket to serve API requests
68     wsHandler    *rpc.Server // Websocket RPC request handler to process the API requests
69
70     stop chan struct{} // Channel to wait for termination notifications
71     lock sync.RWMutex
72
73     log log.Logger
74 }

```

이 노드 구조체에 사실상 `geth`의 모든 로직이 포함되어 있다고 해도 과언이 아니다.

그 외중에 `serviceFuncs` 와 `services` 이 특히 중요한데, 이 필드가 바로 첫번째 슬라이드에서 강조했던 서비스들에 대한 필드이기 때문이다.

1.1 makeConfigNode()



Onther Inc.

1. 네트워크에 참여할 노드의 생성과 초기화
2. 기본 설정 및 터미널 명령과 함께 주어진 인자(*cli.Context)
에 따라 노드 설정 적용 및 기타 서비스 등록

파라미터로 *cli.Context타입을 받고 있고, 리턴값으로 Node, gethConfig타입을 리턴한다.

gethConfig의 두 가지 중요한 필드는 Eth와 Node다.
Eth는 이더리움 프로토콜과 관련된 설정들이고, Node는 주로
p2p통신과 관련된 설정들을 포함하고 있다.

먼저 geth와 관련된 Config들을 DefaultConfig으로 설정한다.

utils.SetNodeConfig() 와 node.New()

그 후 Node 설정을 하고 난 뒤에 stack변수에 Node를
초기화하여 할당한다.

utils.SetEthConfig()

Eth관련 설정을 진행한다.

다음 슬라이드에서는 위 짚은 글씨체의 함수를 조져보자.

```
type gethConfig struct {
    Eth      eth.Config
    Shh      whisper.Config
    Node    node.Config
    Ethstats ethstatsConfig
    Dashboard dashboard.Config
}
```

cmd/geth/config.go | func makeConfigNode()

```

110 func makeConfigNode(ctx *cli.Context) (*node.Node, gethConfig) {
111     // Load defaults.
112     cfg := gethConfig{
113         Eth:      eth.DefaultConfig,
114         Shh:      whisper.DefaultConfig,
115         Node:    defaultNodeConfig(),
116         Dashboard: dashboard.DefaultConfig,
117     }
118
119     // Load config file.
120     if file := ctx.GlobalString(configFileFlag.Name); file != "" {
121         if err := loadConfig(file, &cfg); err != nil {
122             utils.Fatalf(format: "%v", err)
123         }
124     }
125
126     // Apply flags.
127     utils.SetNodeConfig(ctx, &cfg.Node)
128     stack, err := node.New(&cfg.Node)
129     if err != nil {
130         utils.Fatalf(format: "Failed to create the protocol stack: %v", err)
131     }
132     utils.SetEthConfig(ctx, stack, &cfg.Eth)
133     if ctx.GlobalIsSet(utils.EthStatsURLFlag.Name) {
134         cfg.Ethstats.URL = ctx.GlobalString(utils.EthStatsURLFlag.Name)
135     }
136
137     utils.SetShhConfig(ctx, stack, &cfg.Shh)
138     utils.SetDashboardConfig(ctx, &cfg.Dashboard)
139
140     return stack, cfg
141 }
```

cmd/utils/flags.go | func setNodeConfig()



Onther Inc.

```
935 // SetNodeConfig applies node-related command line flags to the config.
936 func SetNodeConfig(ctx *cli.Context, cfg *node.Config) {
937     SetP2PConfig(ctx, &cfg.P2P)
938     setIPC(ctx, cfg)
939     setHTTP(ctx, cfg)
940     setWS(ctx, cfg)
941     setNodeUserIdent(ctx, cfg)
942
943     switch {
944         case ctx.GlobalIsSet(DataDirFlag.Name):
945             cfg.DataDir = ctx.GlobalString(DataDirFlag.Name)
946         case ctx.GlobalBool(DeveloperFlag.Name):
947             cfg.DataDir = "" // unless explicitly requested, use memory databases
948         case ctx.GlobalBool(TestnetFlag.Name):
949             cfg.DataDir = filepath.Join(node.DefaultDataDir(), "testnet")
950         case ctx.GlobalBool(RinkebyFlag.Name):
951             cfg.DataDir = filepath.Join(node.DefaultDataDir(), "rinkeby")
952     }
953
954     if ctx.GlobalIsSet(KeyStoreDirFlag.Name) {
955         cfg.KeyStoreDir = ctx.GlobalString(KeyStoreDirFlag.Name)
956     }
957     if ctx.GlobalIsSet(LightKDFFlag.Name) {
958         cfg.UseLightweightKDF = ctx.GlobalBool(LightKDFFlag.Name)
959     }
960     if ctx.GlobalIsSet(NoUSBFlag.Name) {
961         cfg.NoUSB = ctx.GlobalBool(NoUSBFlag.Name)
962     }
963 }
```

역시 보면 알겠지만 **p2p** 통신관련한 설정들을 잡아준다.
그리고 **flags**의 유무에 따라 세팅을 잡아주는 것을 볼 수 있다.

ctx.Global~ 같은 메소드는 모두 리턴값이 bool타입이기 때문에 결과에 따라 설정을 잡아줄지 말지를 결정할 수 있다.



node/node.go | func New() // node.New()

```
76 // New creates a new P2P node, ready for protocol registration.
77 func New(conf *Config) (*Node, error) {
78     // Copy config and resolve the datadir so future changes to the current
79     // working directory don't affect the node.
80     confCopy := *conf
81     conf = &confCopy
82     if conf.DataDir != "" {
83         absdatadir, err := filepath.Abs(conf.DataDir)
84         if err != nil {
85             return nil, err
86         }
87         conf.DataDir = absdatadir
88     }
89     // Ensure that the instance name doesn't cause weird conflicts with
90     // other files in the data directory.
91     if strings.ContainsAny(conf.Name, chars:`/\` ) {
92         return nil, errors.New(text: `Config.Name must not contain '/' or '\'`)
93     }
94     if conf.Name == datadirDefaultKeyStore {
95         return nil, errors.New(text: `Config.Name cannot be `` + datadirDefaultKeyStore + ```)
96     }
97     if strings.HasSuffix(conf.Name, suffix: ".ipc") {
98         return nil, errors.New(text: `Config.Name cannot end in ".ipc"`)
99     }
100    // Ensure that the AccountManager method works before the node has started.
101    // We rely on this in cmd/geth.
102    am, ephemeralKeystore, err := makeAccountManager(conf)
103    if err != nil {
104        return nil, err
105    }
106    if conf.Logger == nil {
107        conf.Logger = log.New()
108    }
109    // Note: any interaction with Config that would create/touch files
110    // in the data directory or instance directory is delayed until Start.
111    return &Node{
112        accman:           am,
113        ephemeralKeystore: ephemeralKeystore,
114        config:            conf,
115        serviceFuncs:      []ServiceConstructor{},
116        ipcEndpoint:       conf.IPCEndpoint(),
117        httpEndpoint:      conf.HTTPEndpoint(),
118        wsEndpoint:        conf.WSEndpoint(),
119        eventmux:          new(event.TypeMux),
120        log:               conf.Logger,
121    }, nil
122 }
```

이 함수는 인자로 &cfg.Node를 받아서 &Node를 리턴한다.

즉 노드와 관련된 설정들을 바탕으로 Node를 생성하여 리턴하는 함수

먼저 config과 관련해서 추후 변경되는 것들이 현재의 설정에 영향을 미치지 않게끔 설정을 해주고, 노드가 다른 파일들과 충돌이 나지 않게끔 체크를 해준다.

그리고 makeAccountManager() 함수를 호출하여 어카운트 매니저를 가져온다.

가져온 manager는 Node 구조체의 필드인 accman에 할당된다.

node/config.go | func makeAccountManager()



Onther Inc.

makeAccountManager()가 하는 역할은

KeyStore 를 생성하고,
어카운트 manager 생성 후 리턴하는 것이다.

```
408     func makeAccountManager(conf *Config) (*accounts.Manager, string, error) {
409         scryptN, scryptP, keydir, err := conf.AccountConfig()
410         var ephemeral string
411         if keydir == "" {
412             // There is no datadir.
413             keydir, err = ioutil.TempDir( dir: "", prefix: "go-ethereum-keystore")
414             ephemeral = keydir
415         }
416
417         if err != nil {
418             return nil, "", err
419         }
420         if err := os.MkdirAll(keydir, perm: 0700); err != nil {
421             return nil, "", err
422         }
423         // Assemble the account manager and supported backends
424         backends := []accounts.Backend{
425             keystore.NewKeyStore(keydir, scryptN, scryptP),
426         }
427         if !conf.NoUSB {
428             // Start a USB hub for Ledger hardware wallets
429             if ledgerhub, err := usbwallet.NewLedgerHub(); err != nil {
430                 log.Warn(fmt.Sprintf( format: "Failed to start Ledger hub, disabling: %v", err))
431             } else {
432                 backends = append(backends, ledgerhub)
433             }
434             // Start a USB hub for Trezor hardware wallets
435             if trezorhub, err := usbwallet.NewTrezorHub(); err != nil {
436                 log.Warn(fmt.Sprintf( format: "Failed to start Trezor hub, disabling: %v", err))
437             } else {
438                 backends = append(backends, trezorhub)
439             }
440         }
441         return accounts.NewManager(backends...), ephemeral, nil
442     }
```

keystore/keystore.go | func NewKeyStore() 외 1개



Onther Inc.

```
78 // NewKeyStore creates a keystore for the given directory.
79 func NewKeyStore(keydir string, scryptN, scryptP int) *KeyStore {
80     keydir, _ = filepath.Abs(keydir)
81     ks := &KeyStore{storage: &keyStorePassphrase{keysDirPath: keydir, scryptN: scryptN, scryptP: scryptP}}
82     ks.init(keydir)
83     return ks
84 }
```

```
41 // NewManager creates a generic account manager to sign transaction via various
42 // supported backends.
43 func NewManager(backends ...Backend) *Manager {
44     // Retrieve the initial list of wallets from the backends and sort by URL
45     var wallets []Wallet
46     for _, backend := range backends {
47         wallets = merge(wallets, backend.Wallets()...)
48     }
49     // Subscribe to wallet notifications from all backends
50     updates := make(chan WalletEvent, 4*len(backends))
51
52     subs := make([]event.Subscription, len(backends))
53     for i, backend := range backends {
54         subs[i] = backend.Subscribe(updates)
55     }
56
57     // Assemble the account manager and return
58     am := &Manager{
59         backends: make(map[reflect.Type][]Backend),
60         updaters: subs,
61         updates: updates,
62         wallets: wallets,
63         quit:    make(chan chan error),
64     }
65     for _, backend := range backends {
66         kind := reflect.TypeOf(backend)
67         am.backends[kind] = append(am.backends[kind], backend)
68     }
69     go am.update()
70
71 }
```

NewKeyStore()는 keystore 디렉토리에
어카운트 키파일을 만드는 것 같다.

NewManager()함수는 생성된 키스토어를
바탕으로 *Manager타입을 리턴함

accounts/manager.go | type Manager struct



Onther Inc.

```
27 // Manager is an overarching account manager that can communicate with various
28 // backends for signing transactions.
29 type Manager struct {
30     backends map[reflect.Type][]Backend // Index of backends currently registered
31     updaters []event.Subscription      // Wallet update subscriptions for all backends
32     updates chan WalletEvent          // Subscription sink for backend wallet changes
33     wallets []Wallet                  // Cache of all wallets from all registered backends
34
35     feed event.Feed // Wallet feed notifying of arrivals/departures
36
37     quit chan chan error
38     lock sync.RWMutex
39 }
```

Manager 타입은 요렇게 생겼다. 참고용

cmd/utils/flags.go | func setEthConfig()

```

1076 // SetEthConfig applies eth-related command line flags to the config.
1077 func SetEthConfig(ctx *cli.Context, stack *node.Node, cfg *eth.Config) {
1078     // Avoid conflicting network flags
1079     checkExclusive(ctx, DeveloperFlag, TestnetFlag, RinkebyFlag)
1080     checkExclusive(ctx, LightServFlag, SyncModeFlag, "light")
1081
1082     ks := stack.AccountManager().Backends(keystore.KeyStoreType)[0].(*keystore.KeyStore)
1083     setEtherbase(ctx, ks, cfg)
1084     setGPO(ctx, &cfg.GPO)
1085     setTxPool(ctx, &cfg.TxPool)
1086     setEthash(ctx, cfg)
1087
1088     if ctx.GlobalIsSet(SyncModeFlag.Name) {
1089         cfg.SyncMode = *GlobalTextMarshaler(ctx, SyncModeFlag.Name).(*downloader.SyncMode)
1090     }
1091     if ctx.GlobalIsSet(LightServFlag.Name) {
1092         cfg.LightServ = ctx.GlobalInt(LightServFlag.Name)
1093     }
1094     if ctx.GlobalIsSet(LightPeersFlag.Name) {
1095         cfg.LightPeers = ctx.GlobalInt(LightPeersFlag.Name)
1096     }
1097     if ctx.GlobalIsSet(NetworkIdFlag.Name) {
1098         cfg.NetworkId = ctx.GlobalUint64(NetworkIdFlag.Name)
1099     }
1100
1101     if ctx.GlobalIsSet(CacheFlag.Name) || ctx.GlobalIsSet(CacheDatabaseFlag.Name) {
1102         cfg.DatabaseCache = ctx.GlobalInt(CacheFlag.Name) * ctx.GlobalInt(CacheDatabaseFlag.Name) / 100
1103     }
1104     cfg.DatabaseHandles = makeDatabaseHandles()

```

매우 길다... 쭉 이어져 있다.
 앞 로직 보면 TxPool 잡아주고, Ethash 설정해주고
 하는 모습을 볼 수 있다.
 즉 커맨드 라인 플래그를 바탕으로 이더리움
 프로토콜 관련한 설정을 잡아주는 함수이다.

여기까지 보면 makeConfigNode()의 큰 흐름을 다
 본 것이다. 결국 커맨드라인에서 입력받은 것을
 바탕으로 Node와 관련된 여러 설정들을
 잡아준다고 보면 된다.

1.2 RegisterEthService()

cmd/geth/config.go | func makeFullNode()



```
153 func makeFullNode(ctx *cli.Context) *node.Node {
154     stack, cfg := makeConfigNode(ctx)
155
156     utils.RegisterEthService(stack, &cfg.Eth) // Line 156 highlighted with a red box
157
158     if ctx.GlobalBool(utils.DashboardEnabledFlag.Name) {
159         utils.RegisterDashboardService(stack, &cfg.Dashboard, gitCommit)
160     }
161     // Whisper must be explicitly enabled by specifying at least 1 whisper flag or in dev mode
162     shhEnabled := enableWhisper(ctx)
163     shhAutoEnabled := !ctx.GlobalIsSet(utils.WhisperEnabledFlag.Name) && ctx.GlobalIsSet(utils.DeveloperFlag.Name)
164     if shhEnabled || shhAutoEnabled {
165         if ctx.GlobalIsSet(utils.WhisperMaxMessageSizeFlag.Name) {
166             cfg.Shh.MaxMessageSize = uint32(ctx.Int(utils.WhisperMaxMessageSizeFlag.Name))
167         }
168         if ctx.GlobalIsSet(utils.WhisperMinPOWFlag.Name) {
169             cfg.Shh.MinimumAcceptedPOW = ctx.Float64(utils.WhisperMinPOWFlag.Name)
170         }
171         utils.RegisterShhService(stack, &cfg.Shh)
172     }
173
174     // Add the Ethereum Stats daemon if requested.
175     if cfg.Ethstats.URL != "" {
176         utils.RegisterEthStatsService(stack, cfg.Ethstats.URL)
177     }
178     return stack
179 }
```

makeConfigNode()에서 생성한 노드와 config.Eth를 파라미터로 하여 RegisterEthService()를 실행한다.
이름에서 알 수 있듯이 이는 이더리움 프로토콜과 관련된 여러 서비스들을 등록하는 역할을 한다.

cmd/utils/flags.go | func RegisterEthService()



Onther Inc.

```
1195     // RegisterEthService adds an Ethereum client to the stack.
1196     func RegisterEthService(stack *node.Node, cfg *eth.Config) {
1197         var err error
1198         if cfg.SyncMode == downloader.LightSync {
1199             err = stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
1200                 return les.New(ctx, cfg)
1201             })
1202         } else {
1203             err = stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
1204                 fullNode, err := eth.New(ctx, cfg)
1205                 if fullNode != nil && cfg.LightServ > 0 {
1206                     ls, _ := les.NewLesServer(fullNode, cfg)
1207                     fullNode.AddLesServer(ls)
1208                 }
1209                 return fullNode, err
1210             })
1211         }
1212         if err != nil {
1213             Fatalf(format: "Failed to register the Ethereum service: %v", err)
1214         }
1215     }
```

이 함수는 인자로 받은 cfg의 SyncMode에 따라 등록하는 생성자 함수가 다르다.
생성자 부분을 요약하면 함수의 코드 구조는 다음 슬라이드와 같이 매우 단순하다.

cmd/utils/flags.go | func RegisterEthService()



```
// RegisterEthService adds an Ethereum client to the stack.
func RegisterEthService(stack *node.Node, cfg *eth.Config) {
    var err error
    if cfg.SyncMode == downloader.LightSync {
        err = stack.Register(/* LightSync 일때 생성자 함수 */)
    } else {
        err = stack.Register(/* LightSync가 아닐때 생성자 함수 */)
    }
    if err != nil {
        Fatalf("Failed to register the Ethereum service: %v", err)
    }
}
```

기본 설정은 FastSync로 잡혀있다.
따라서 아래 else 문을 타게 된다.

```
24 const (
25     FullSync SyncMode = iota // Synchronise the entire blockchain history from full blocks
26     FastSync           // Quickly download the headers, full sync only at the chain head
27     LightSync          // Download only the headers and terminate afterwards
28 )
```

```
const (
    FullSync = 0
    FastSync = 1
    LightSync= 2
)
```

```
37 var DefaultConfig = Config{
38     SyncMode: downloader.FastSync,
```

cmd/utils/flags.go | func RegisterEthService()



```
1195 // RegisterEthService adds an Ethereum client to the stack.
1196 func RegisterEthService(stack *node.Node, cfg *eth.Config) {
1197     var err error
1198     if cfg.SyncMode == downloader.LightSync {
1199         err = stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
1200             return les.New(ctx, cfg)
1201         })
1202     } else {
1203         err = stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
1204             fullNode, err := eth.New(ctx, cfg)
1205             if fullNode != nil && cfg.LightServ > 0 {
1206                 ls, _ := les.NewLesServer(fullNode, cfg)
1207                 fullNode.AddLesServer(ls)
1208             }
1209             return fullNode, err
1210         })
1211     }
1212     if err != nil {
1213         Fatalf(format: "Failed to register the Ethereum service: %v", err)
1214     }
1215 }
```

추후에 이야기할 serviceFuncs에 등록되어 호출되는 생성자 함수는 바로 Register() 내부의 익명함수이다.

이 익명함수는 eth 패키지의 함수 eth.New로 이더리움 객체를 생성하고, 이를 fullNode라는 변수로 담는다. 하지만 지금 당장은 실행되지는 않고 일종의 콜백처럼 함수가 등록되었다가 추후에 실행되게 된다.

패키지 les는 Light Ethereum Subprotocol 구현체로 현재 개발중인가봐요. 뭔지 정확히는 모르겠..

<https://github.com/ethereum/wiki/wiki/Light-client-protocol>

node/node.go | func Register()



```
124     // Register injects a new service into the node's stack. The service created by
125     // the passed constructor must be unique in its type with regard to sibling ones.
126     func (n *Node) Register(constructor ServiceConstructor) error {
127         n.lock.Lock()
128         defer n.lock.Unlock()
129
130         if n.server != nil {
131             return ErrNodeRunning
132         }
133         n.serviceFuncs = append(n.serviceFuncs, constructor)
134         return nil
135     }
```

Register 메소드는 Node.serviceFuncs 슬라이스에 constructor 함수를 append하는 역할을 한다.
여기서 append되는 constructor가 바로 앞에서 살펴본 익명함수이다.

이후에 여기서 append된 함수들이 startNode() 실행중에 호출된다.

cmd/geth/config.go | func makeFullNode()



```
153     func makeFullNode(ctx *cli.Context) *node.Node {
154         stack, cfg := makeConfigNode(ctx)
155
156         utils.RegisterEthService(stack, &cfg.Eth)
157
158         if ctx.GlobalBool(utils.DashboardEnabledFlag.Name) {
159             utils.RegisterDashboardService(stack, &cfg.Dashboard, gitCommit)
160         }
161         // Whisper must be explicitly enabled by specifying at least 1 whisper flag or in dev mode
162         shhEnabled := enableWhisper(ctx)
163         shhAutoEnabled := !ctx.GlobalIsSet(utils.WhisperEnabledFlag.Name) && ctx.GlobalIsSet(utils.DeveloperFlag.Name)
164         if shhEnabled || shhAutoEnabled {
165             if ctx.GlobalIsSet(utils.WhisperMaxMessageSizeFlag.Name) {
166                 cfg.Shh.MaxMessageSize = uint32(ctx.Int(utils.WhisperMaxMessageSizeFlag.Name))
167             }
168             if ctx.GlobalIsSet(utils.WhisperMinPOWFlag.Name) {
169                 cfg.Shh.MinimumAcceptedPOW = ctx.Float64(utils.WhisperMinPOWFlag.Name)
170             }
171             utils.RegisterShhService(stack, &cfg.Shh)
172         }
173
174         // Add the Ethereum Stats daemon if requested.
175         if cfg.Ethstats.URL != "" {
176             utils.RegisterEthStatsService(stack, cfg.Ethstats.URL)
177         }
178     return stack
179 }
```

다시 makeFullNode로 돌아와서 보면 Register~ 라고 붙은 함수가 연달아서 호출되는 걸 알 수 있다.

위 빨간색으로 표시한 함수는 딱히 중요하지 않은 함수라고 앞에서 언급했지만 결국 RegisterEthService()와 로직이 매우 유사하다는 것을 알기 위해 간단하게만 보고 갑시다.

cmd/utils/flags.go |

```

1217 // RegisterDashboardService adds a dashboard to the stack.
1218 func RegisterDashboardService(stack *node.Node, cfg *dashboard.Config, commit string) {
1219     stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
1220         return dashboard.New(cfg, commit, ctx.ResolvePath(path: "logs")), nil
1221     })
1222 }
1223
1224 // RegisterShhService configures Whisper and adds it to the given node.
1225 func RegisterShhService(stack *node.Node, cfg *whisper.Config) {
1226     if err := stack.Register(func(n *node.ServiceContext) (node.Service, error) {
1227         return whisper.New(cfg), nil
1228     }); err != nil {
1229         Fatalf(format: "Failed to register the Whisper service: %v", err)
1230     }
1231 }
1232
1233 // RegisterEthStatsService configures the Ethereum Stats daemon and adds it to
1234 // the given node.
1235 func RegisterEthStatsService(stack *node.Node, url string) {
1236     if err := stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
1237         // Retrieve both eth and les services
1238         var ethServ *eth.Ethereum
1239         ctx.Service(&ethServ)
1240
1241         var lesServ *les.LightEthereum
1242         ctx.Service(&lesServ)
1243
1244         return ethstats.New(url, ethServ, lesServ)
1245     }); err != nil {
1246         Fatalf(format: "Failed to register the Ethereum Stats service: %v", err)
1247     }
1248 }
```

보면 알겠지만 앞에서 살펴본 것과 같이 모두 `Register(익명함수)`의 형태로 `stack.serviceFuncs`에 등록하는 것을 알 수 있음

어쨌든 중요한 것은 이처럼 `Register()`메소드를 통해 인자로 주어진 익명함수들이 `serviceFuncs`에 등록되어 추후에 호출된다는 것!

여기까지 보면 `makeFullNode()`의 큰 흐름은 다 본 것!!!



Onther Inc.

node/node.go | func Start()



```
199     // Start each of the services
200     started := []reflect.Type{}
201     for kind, service := range services {
202         // Start the next service, stopping all previous upon failure
203         if err := service.Start(running); err != nil {
204             for _, kind := range started {
205                 services[kind].Stop()
206             }
207             running.Stop()
208         }
209         return err
210     }
211     // Mark the service started for potential cleanup
212     started = append(started, kind)
213 }
```

node.go 의 func Start() 내부에서 eth/backend.go func Start()를 호출
이는 추후에 startNode() 함수로직 설명하면서 자세히 살펴보게될 내용임

eth/backend.go | func Start()

```

385     // Start implements node.Service, starting all internal goroutines needed by the
386     // Ethereum protocol implementation.
387     func (s *Ethereum) Start(srver *p2p.Server) error {
388         // Start the bloom bits servicing goroutines
389         s.startBloomHandlers()
390
391         // Start the RPC service
392         s.netRPCService = ethapi.NewPublicNetAPI(srver, s.NetVersion())
393
394         // Figure out a max peers count based on the server limits
395         maxPeers := srver.MaxPeers
396         if s.config.LightServ > 0 {
397             if s.config.LightPeers >= srver.MaxPeers {
398                 return fmt.Errorf(fmt.Sprintf("invalid peer config: light peer count (%d) >= total peer count (%d)", s.config.LightPeers, srver.MaxPeers))
399             }
400             maxPeers -= s.config.LightPeers
401         }
402         // Start the networking layer and the light server if requested
403         s.protocolManager.Start(maxPeers)
404         if s.lesServer != nil {
405             s.lesServer.Start(srver)
406         }
407     }
408 }
```

앞서 살펴본 New 함수로 생성/초기화된 fullNode 객체는 다음의 Start 함수와 함께 시작되는데요. 코드를 보면 시작의 의미가 결국 네트워크 관련된 코드를 활성화시키는 부분이란 걸 알 수 있습니다.

주석이 이미 역할별로 끊어서 잘 설명해주고 있어서 부연할 것이 없는 간단한 로직이네요. 여기서 실행하는 모든 서비스들이 별도의 독립적인 고루틴으로 실행된다고 주석에서 알려줍니다. 마지막으로 종료함수를 봅시다.

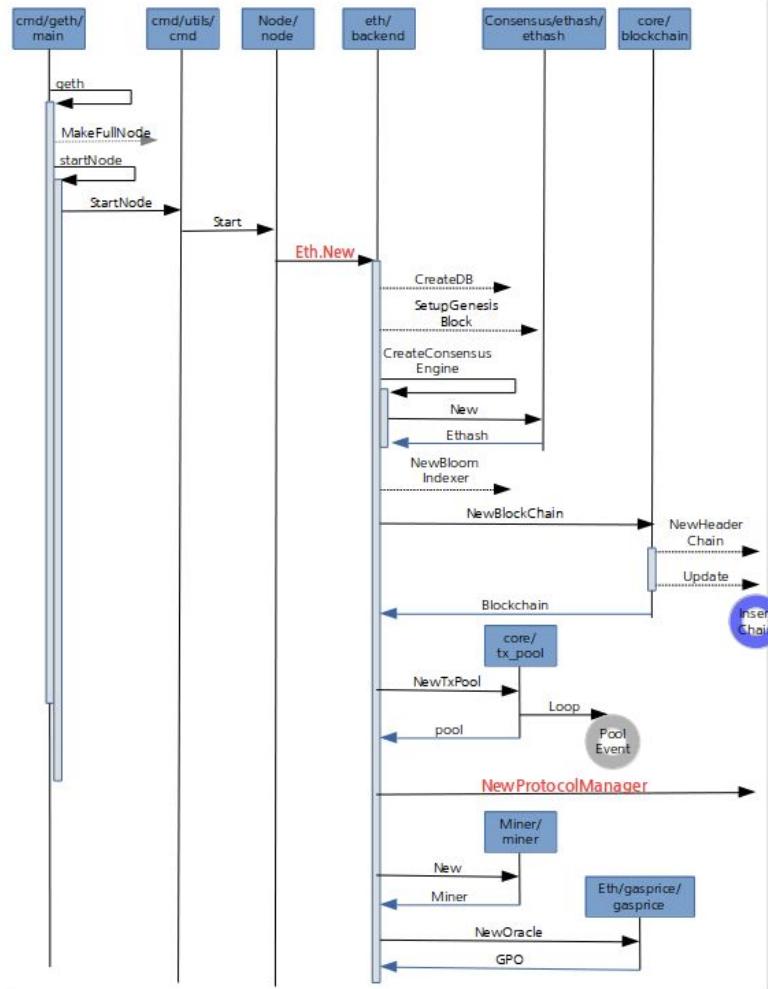
eth/backend.go | func Stop()

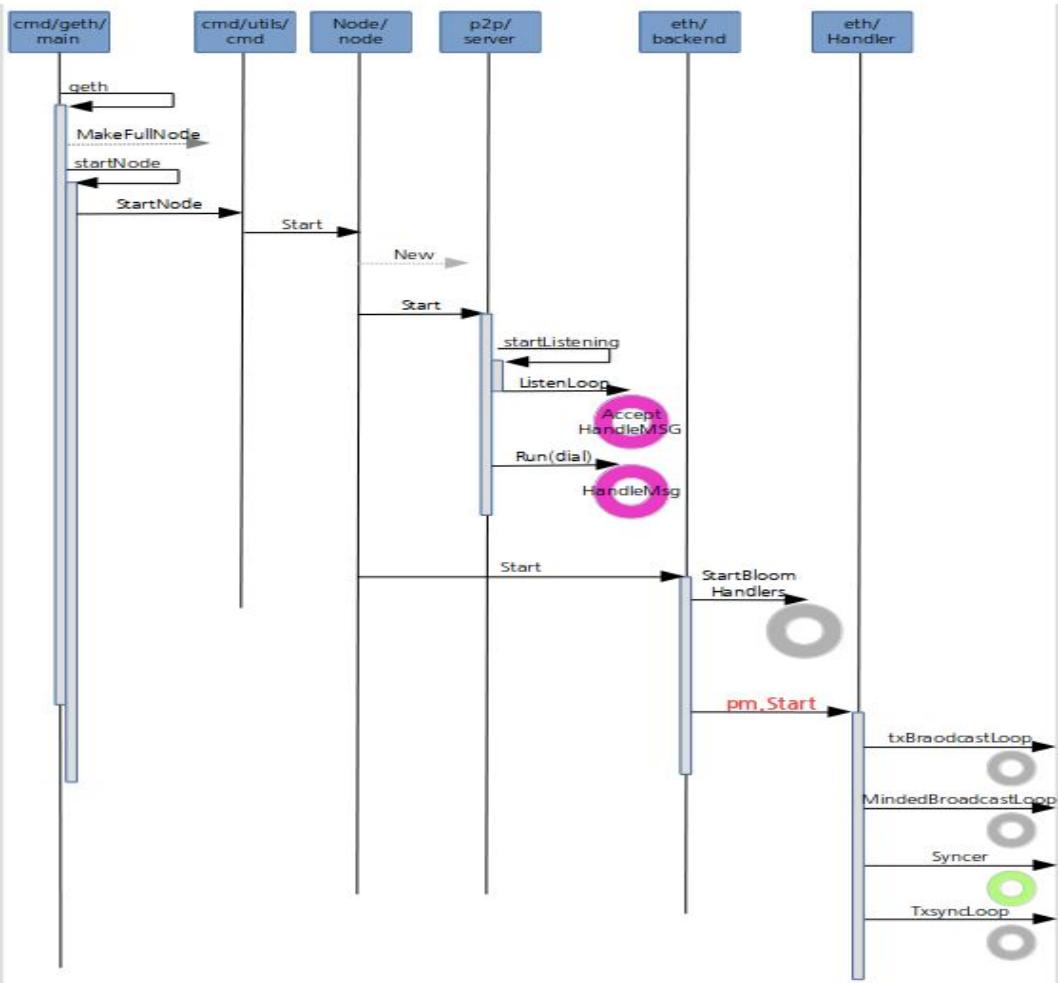


```
410     // Stop implements node.Service, terminating all internal goroutines used by the
411     // Ethereum protocol.
412     func (s *Ethereum) Stop() error {
413         s.bloomIndexer.Close()
414         s.blockchain.Stop()
415         s.engine.Close()
416         s.protocolManager.Stop()
417         if s.lesServer != nil {
418             s.lesServer.Stop()
419         }
420         s.txPool.Stop()
421         s.miner.Stop()
422         s.eventMux.Stop()
423
424         s.chainDb.Close()
425         close(s.shutdownChan)
426         return nil
427     }
```

종료함수는 간단합니다. "조립은 분해의 역순"처럼 각각 독립적으로 돌고있을 고루틴을 하나씩 종료하는군요.
여기서 종료하는 로직들 하나하나가 결국 독립적인 모듈이라 파트2에서 살펴볼 주제가 됩니다.

2. startNode()





cmd/geth/main.go | func geth()

```

255     // geth is the main entry point into the system if no special subcommand is ran.
256     // It creates a default node based on the command line arguments and runs it in
257     // blocking mode, waiting for it to be shut down.
258     func geth(ctx *cli.Context) error {
259         if args := ctx.Args(); len(args) > 0 {
260             return fmt.Errorf( format: "invalid command: %q", args[0])
261         }
262         node := makeFullNode(ctx)
263         startNode(ctx, node)
264         node.Wait()
265         return nil
266     }

268     // startNode boots up the system node and all registered protocols, after which
269     // it unlocks any requested accounts, and starts the RPC/IPC interfaces and the
270     // miner.
271     func startNode(ctx *cli.Context, stack *node.Node) {
272         debug.Memsize.Add( name: "node", stack)
273
274         // Start up the node itself
275         utils.StartNode(stack)

```

makeFullNode()에서 설정을 모두 마친 후 func geth() 바로 아래에 선언되어 있는 startNode()함수를 호출하여 노드를 실행하게 된다.

startNode() 함수는 크게 세가지 주요 로직으로 요약할 수 있다.

1. 노드 자체를 시작하는 로직
2. 어카운트를 언락하는 로직
3. 마이닝을 시작하는 로직

그럼 먼저 1. 노드 자체를 시작하는 로직부터 살펴보자.

startNode() 는 utils.StartNode() 를 실행하여 makeFullNode()에서 설정을 마친 노드를 실행하게 된다.

2.1 Start Node itself

cmd/utils/cmd.go | func StartNode()

```

66     func StartNode(stack *node.Node) {
67         if err := stack.Start(); err != nil {
68            .Fatalf( format: "Error starting protocol stack: %v", err)
69         }
70         go func() {
71             sigc := make(chan os.Signal, 1)
72             signal.Notify(sigc, syscall.SIGINT, syscall.SIGTERM)
73             defer signal.Stop(sigc)
74             <-sigc
75             log.Info( msg: "Got interrupt, shutting down...")
76             go stack.Stop()
77             for i := 10; i > 0; i-- {
78                 <-sigc
79                 if i > 1 {
80                     log.Warn( msg: "Already shutting down, interrupt more to panic.", ctx: "times", i-1)
81                 }
82             }
83             debug.Exit() // ensure trace and CPU profile data is flushed.
84             debug.LoudPanic( x: "boom")
85         }()
86     }

```

사실 이 함수는 실제 노드를 실행하는 로직은 없다. stack의 Start()메소드에 노드의 구체적인 실행로직이 있다.

아래 부분에 stack.Stop 도 있는데, 이는 노드의 종료를 위한 메소드로, sigc채널에서 OS시그널을 대기하다 메세지가 들어오면 호출된다.

node/node.go | func Start() // stack.Start()



```
137 // Start create a live P2P node and starts running it.
138 func (n *Node) Start() error {
139     n.lock.Lock()
140     defer n.lock.Unlock()
141
142     // Short circuit if the node's already running
143     if n.server != nil {
144         return ErrNodeRunning
145     }
146     if err := n.openDataDir(); err != nil {
147         return err
148 }
```

이제 본격적으로 start() 함수를 살펴보자. 로직이 매우 길기 때문에 하나씩 차근차근 살펴보도록 하자.

먼저, (mutex) Lock 을 잡아주고, Unlock()을 defer로 함수의 종료시에 Lock을 해제한다.

n.server 가 이미 존재한다면 노드가 작동중임을 의미하기 때문에 에러를 리턴하고 실행을 종료한다.

노드에서 사용할 데이터 보관을 위하여 디렉토리를 만든다.

node/node.go | func Start() // stack.Start()

```
150 // Initialize the p2p server. This creates the node key and
151 // discovery databases.
152 n.serverConfig = n.config.P2P
153 n.serverConfig.PrivateKey = n.config.NodeKey()
154 n.serverConfig.Name = n.config.NodeName()
155 n.serverConfig.Logger = n.log
156 if n.serverConfig.StaticNodes == nil {
157     n.serverConfig.StaticNodes = n.config.StaticNodes()
158 }
159 if n.serverConfig.TrustedNodes == nil {
160     n.serverConfig.TrustedNodes = n.config.TrustedNodes()
161 }
162 if n.serverConfig.NodeDatabase == "" {
163     n.serverConfig.NodeDatabase = n.config.NodeDB()
164 }
165 running := &p2p.Server{Config: n.serverConfig}
166 n.log.Info( msg: "Starting peer-to-peer node", ctx: "instance", n.serverConfig.Name)
```

```
37 // DefaultConfig contains reasonable default settings.
38 var DefaultConfig = Config{
39     DataDir:          DefaultDataDir(),
40     HTTPPort:         DefaultHTTPPort,
41     HTTPModules:      []string{"net", "web3"},
42     HTTPVirtualHosts: []string{"localhost"},
43     HTTPTimeouts:    rpc.DefaultHTTPTimeouts,
44     WSPort:          DefaultWSPort,
45     WSMODULES:        []string{"net", "web3"},
46     P2P: p2p.Config{
47         ListenAddr: "":30303",
48         MaxPeers: 25,
49         NAT: nat.Any(),
50     },
51 }
```

152 : 이어서 네트워크에 참여할 실제 서버 노드 설정인 n.config.P2P를 n.serverConfig 필드에 할당한다.

n.config.P2P는 makeConfigNode() 실행중에 Node에 관한 설정을 할 때 잡아줬음. Node설정은 defaultNodeConfig()의 리턴값으로 세팅되기 때문에 오른쪽의 P2P필드로 값이 초기값이 설정될 것임.

이것 말고도 n.config.P2P필드의 value 타입인 p2p.Config에는 여러가지 필드가 있다. 그와 관련된 설정들을 아래에서 추가적으로 잡아주게 되는 것임
(자세한 필드는 p2p/server.go에 있다)

153 : n.serverConfig.PrivateKey 는 *ecdsa.PrivateKey에 관한 것. NodeKey()함수를 호출하면 PrivateKey를 리턴함

154 : n.serverConfig.Name 은 devp2p node identifier를 리턴한다고 함

156~164 : StaticNode, TrustedNode는 정확히 뭘 의미하는지는 잘 모르겠음. NodeDB()는 노드 데이터베이스가 없으면 다시 잡아주는 것 같다.

위 설정을 바탕으로 **p2p.Server**타입의 구조체 변수 **running**을 생성한 후 INFO 레벨로 p2p 노드를 시작한다고 로그를 찍습니다.

node/node.go | type Node struct



```
// Node is a container on which services can be registered.
type Node struct {
    eventmux *event.TypeMux // Event multiplexer used between the services of a stack
    config   *Config
    accman   *accounts.Manager

    ephemeralKeystore string           // if non-empty, the key directory that will be removed by Stop
    instanceDirLock   flock.Releaser // prevents concurrent use of instance directory

    serverConfig p2p.Config
    server      *p2p.Server // Currently running P2P networking layer

    serviceFuncs []ServiceConstructor // Service constructors (in dependency order)
    services     map[reflect.Type]Service // Currently running services

    rpcAPIs       []rpc.API    // List of APIs currently provided by the node
    inprocHandler *rpc.Server // In-process RPC request handler to process the API requests

    ipcEndpoint  string      // IPC endpoint to listen at (empty = IPC disabled)
    ipcListener  net.Listener // IPC RPC listener socket to serve API requests
    ipcHandler   *rpc.Server // IPC RPC request handler to process the API requests

    httpEndpoint string      // HTTP endpoint (interface + port) to listen at (empty = HTTP disabled)
    httpWhitelist []string    // HTTP RPC modules to allow through this endpoint
    httpListener  net.Listener // HTTP RPC listener socket to server API requests
    httpHandler   *rpc.Server // HTTP RPC request handler to process the API requests

    wsEndpoint   string      // Websocket endpoint (interface + port) to listen at (empty = websocket disabled)
    wsListener   net.Listener // Websocket RPC listener socket to server API requests
    wsHandler    *rpc.Server // Websocket RPC request handler to process the API requests

    stop chan struct{} // Channel to wait for termination notifications
    lock sync.RWMutex

    log log.Logger
}
```

node/node.go | func Start() // stack.Start()

```

168     // Otherwise copy and specialize the P2P configuration
169     services := make(map[reflect.Type]Service)
170     for _, constructor := range n.serviceFuncs {
171         // Create a new context for the particular service
172         ctx := &ServiceContext{
173             config:          n.config,
174             services:        make(map[reflect.Type]Service),
175             EventMux:        n.eventmux,
176             AccountManager: n.accman,
177         }
178         for kind, s := range services { // copy needed for threaded access
179             ctx.services[kind] = s
180         }
181         // Construct and save the service
182         service, err := constructor(ctx)
183         if err != nil {
184             return err
185         }
186         kind := reflect.TypeOf(service)
187         if _, exists := services[kind]; exists {
188             return &DuplicateServiceError{Kind: kind}
189         }
190         services[kind] = service
191     }

```

이어서 Service를 값타입으로 하고 reflect.Type을 키타입으로 하는 map을 services 변수에 초기화한다.

for 문에서 순회할 대상은 n.serviceFuncs이고 순회 결과는 _, constructor 2개가 있다.

즉, key는 버리고 value부분인 Service타입 (함수이다) 만 constructor에 할당한다.



node/node.go | func Start() // stack.Start()

```

168     // Otherwise copy and specialize the P2P configuration
169     services := make(map[reflect.Type]Service)
170     for _, constructor := range n.serviceFuncs {
171         // Create a new context for the particular service
172         ctx := &ServiceContext{
173             config:          n.config,
174             services:        make(map[reflect.Type]Service),
175             EventMux:        n.eventmux,
176             AccountManager: n.accman,
177         }
178         for kind, s := range services { // copy needed for threaded access
179             ctx.services[kind] = s
180         }
181         // Construct and save the service
182         service, err := constructor(ctx)
183         if err != nil {
184             return err
185         }
186         kind := reflect.TypeOf(service)
187         if _, exists := services[kind]; exists {
188             return &DuplicateServiceError{Kind: kind}
189         }
190         services[kind] = service
191     }

```

```

func RegisterEthService(stack *node.Node, cfg *eth.Config) {
func RegisterDashboardService(stack *node.Node, cfg *dashboard.Config, commit string)
func RegisterShhService(stack *node.Node, cfg *whisper.Config)
func RegisterEthStatsService(stack *node.Node, url string)

```

이 for문에서 n.serviceFuncs 슬라이스에 있는 아이템, 즉 각 service function들을 constructor에 할당하면서 반복문을 하나씩 돌리게 된다.

아래 그림에 있는 Register~~ 함수 내부에서 Register() 함수를 실행하게 되는데 이 함수에 전달되는 인자가 모두 익명 함수 형태이다. 따라서 해당 익명 함수가 모두 반복문에 의해 각각 constructor에 할당되는 것이다.

내부에 포함된 또 다른 반복문을 돌리면서 services 매핑의 key를 kind에 value를 s에 할당한 후, 위에서 생성한 ctx 변수의 services 필드의 map[reflect.Type]Service에 각각 key value로 할당을 해준다.
(제일 처음 실행할 때는 ctx.services 필드가 빈 매핑이겠지만 반복문을 하나씩 돌릴 때마다 등록된 서비스가 차곡차곡 쌓일 것이다)

여기서 할당된 ctx가 각 serviceFuncs 들을 실행할 때 쓰이는 context가 된다. (잘 보면 services를 제외한 모든 필드를 앞에서 설정했던 값들을 불러오는 것을 알 수 있다.)

이후 불러온 serviceFuncs 즉, constructor에 ctx를 인자로 전달하여 실행한 결과를 service에 할당한다.



Onther Inc.

```
111 return &Node{  
112     accman:           am,  
113     ephemeralKeystore: ephemeralKeystore,  
114     config:            conf.  
115     serviceFuncs:      []ServiceConstructor{},  
116     ipcEndpoint:       conf.IPCEndpoint(),  
117     httpEndpoint:      conf.HTTPEndpoint(),  
118     wsEndpoint:        conf.WSEndpoint(),  
119     eventmux:          new(event.TypeMux),  
120     log:               conf.Logger,  
121 }, nil  
122 }
```

func New()

```
70 // ServiceConstructor is the function signature of the constructors needed to be  
71 // registered for service instantiation.  
72 type ServiceConstructor func(ctx *ServiceContext) (Service, error)
```

```
124 // Register injects a new service into the node's stack. The service created by  
125 // the passed constructor must be unique in its type with regard to sibling ones.  
126 func (n *Node) Register(constructor ServiceConstructor) error {  
127     n.lock.Lock()  
128     defer n.lock.Unlock()  
129  
130     if n.server != nil {  
131         return ErrNodeRunning  
132     }  
133     n.serviceFuncs = append(n.serviceFuncs, constructor)  
134     return nil  
135 }
```

이 슬라이드는 굳이
설명할 필요 없음

우선 최초 Node 생성시에는 다음과 같이 빈 인터페이스 배열이 할당되어 있음

[]ServiceConstructor{}

이후 실제 배열에 값을 넣는 곳은 Node의 Register 메서드에서 일어납니다. 바로 위 그림에서 n.serviceFuncs에 append 함수로 constructor를 포함시키는 부분이 Register 메서드의 코드입니다.

Start 메서드와 동일하게 락부터 셋팅 하고 서버가 실행 중인 것을 확인 후 인자로 주어진 constructor를 Node의 serviceFuncs의 포함시키고 있습니다. append함수는 golang의 slice를 위한 함수입니다. 동적인 배열 slice에 요소를 추가할 때 append 함수를 씁니다.

node/node.go | func Start() // stack.Start()

```

168     // Otherwise copy and specialize the P2P configuration
169     services := make(map[reflect.Type]Service)
170     for _, constructor := range n.serviceFuncs {
171         // Create a new context for the particular service
172         ctx := &ServiceContext{
173             config:          n.config,
174             services:        make(map[reflect.Type]Service),
175             EventMux:        n.eventmux,
176             AccountManager: n.acccman,
177         }
178         for kind, s := range services { // copy needed for threaded access
179             ctx.services[kind] = s
180         }
181         // Construct and save the service
182         service, err := constructor(ctx)
183         if err != nil {
184             return err
185         }
186         kind := reflect.TypeOf(service)
187         if _, exists := services[kind]; exists {
188             return &DuplicateServiceError{Kind: kind}
189         }
190         services[kind] = service
191     }

```

남은 코드는 실제 serviceFuncs의 포함된 constructor를 호출하여 서비스 변수를 생성 및 초기화하고 이를 실행하는 것과 연관된 코드입니다. 먼저 이미 살펴봤던 첫번째 for문의 다음 코드는 생성자를 호출하고 이를 services 맵에 담아서 서비스 변수를 관리하는 로직입니다.

constructor(ctx)를 호출하여 등록된 n.serviceFuncs들을 실행하고 리턴값을 service에 할당

services맵에 만약 해당 service가 등록되어 있다면 에러를 리턴!

services맵에 kind(여기서는 service type이 되겠죠?)를 key로하여 service를 할당

이렇게 반복문을 계속 돌리면서 services맵에 serviceFunc들을 construct(초기화)하고 save(저장)한다.

node/node.go | func Start() // stack.Start()

```

192     // Gather the protocols and start the freshly assembled P2P server
193     for _, service := range services {
194         running.Protocols = append(running.Protocols, service.Protocols()...)
195     }
196     if err := running.Start(); err != nil {
197         return convertFileLockError(err)
198     }
199     // Start each of the services
200     started := []reflect.Type{}
201     for kind, service := range services {
202         // Start the next service, stopping all previous upon failure
203         if err := service.Start(running); err != nil {
204             for _, kind := range started {
205                 services[kind].Stop()
206             }
207             running.Stop()
208
209             return err
210         }
211         // Mark the service started for potential cleanup
212         started = append(started, kind)
213     }

```

다음 for 문에서는 service마다 갖고 있는 Protocols()을 running변수 즉, p2p.Server구조체의 Protocols필드(정확히는 p2p.Server.Config.Protocols)에 포함시킵니다.

이어서 running.Start()로 p2p server를 실행한다.

드디어 위에서 생성한 각 서비스를 순차적으로 실행하고, 혹시 하나라도 시작하지 못하면 모두 종료시키는 코드가 나옵니다.

마지막으로 실행이 문제 없이 되었다면 started에 서비스가 실행중임을 마킹하게 됩니다.

p2p/server.go | func Start() // running.Start()



```
419     // Start starts running the server.
420     // Servers can not be re-used after stopping.
421     func (srv *Server) Start() (err error) {
422         srv.lock.Lock()
423         defer srv.lock.Unlock()
424         if srv.running {
425             return errors.New(text: "server already running")
426         }
427         srv.running = true
428         srv.log = srv.Config.Logger
429         if srv.log == nil {
430             srv.log = log.New()
431         }
432         srv.log.Info(msg: "Starting P2P networking")
433
434         // static fields
435         if srv.PrivateKey == nil {
436             return fmt.Errorf(format: "Server.PrivateKey must be set to a non-nil key")
437         }
438         if srv.newTransport == nil {
439             srv.newTransport = newRLPX
440         }
441         if srv.Dialer == nil {
442             srv.Dialer = TCPDialer{ Dialer: &net.Dialer{Timeout: defaultDialTimeout}}
443     }
```

running.Start() 부분을 자세히 살펴봅시다.

P2P 서비스를 시작합니다. 리스닝 루프에서는 피어의 접속을 수락하기 위한 Accept 함수가, RunLoop에서는 피어에 접속하기 위한 dial 함수가 동작합니다. 시간적 순서에 따라 누군가는 dial을 누군가는 accept를 하게 되고 어디서 많이 들어본 discovery v5 protocol이라던지, Devp2p의 handshake라던지 이런 동작을 통해 연결이 확정되면 각 피어의 해당 루프에서는 handleMsg 함수가 호출되게 됩니다. (저 아래 그림에서 insertChain 함수를 호출하도록 만드는 메시지가 여기서 처리됩니다. 핑크색으로 연결해두었습니다)

cmd/eth/backend.go | type Ethereum struct



```
61 // Ethereum implements the Ethereum full node service.
62 type Ethereum struct {
63     config        *Config
64     chainConfig   *params.ChainConfig
65
66     // Channel for shutting down the service
67     shutdownChan chan bool // Channel for shutting down the Ethereum
68
69     // Handlers
70     txPool         *core.TxPool
71     blockchain    *core.BlockChain
72     protocolManager *ProtocolManager
73     lesServer      LesServer
74
75     // DB interfaces
76     chainDb ethdb.Database // Block chain database
77
78     eventMux      *event.TypeMux
79     engine        consensus.Engine
80     accountManager *accounts.Manager
81
82     bloomRequests chan chan *bloombits.Retrieval // Channel receiving bloom data retrieval requests
83     bloomIndexer  *core.ChainIndexer           // Bloom indexer operating during block imports
84
85     APIBackend *EthAPIBackend
86
87     miner        *miner.Miner
88     gasPrice     *big.Int
89     etherbase    common.Address
90
91     networkID    uint64
92     netRPCService *ethapi.PublicNetAPI
93
94     lock sync.RWMutex // Protects the variadic fields (e.g. gas price and etherbase)
95 }
```

다음으로 service.Start()를 자세히 살펴보기 전에,

결국 service에 할당되는 변수는 fullNode 즉, type Ethereum struct이다.

앞에서 service.Start(running)은 결국 running 즉 p2p.server를 인자로 하여 Ethereum 구조체의 Start 메소드를 실행하는 것이다.

eth/backend.go | func Start() // service.Start()



```
385 // Start implements node.Service, starting all internal goroutines needed by the
386 // Ethereum protocol implementation.
387 func (s *Ethereum) Start(srver *p2p.Server) error {
388     // Start the bloom bits servicing goroutines
389     s.startBloomHandlers()
390
391     // Start the RPC service
392     s.netRPCService = ethapi.NewPublicNetAPI(srver, s.NetVersion())
393
394     // Figure out a max peers count based on the server limits
395     maxPeers := srver.MaxPeers
396     if s.config.LightServ > 0 {
397         if s.config.LightPeers >= srver.MaxPeers {
398             return fmt.Errorf("invalid peer config: light peer count (%d) >= total peer count (%d)", s.config.LightPeers, srver.MaxPeers)
399         }
400         maxPeers -= s.config.LightPeers
401     }
402     // Start the networking layer and the light server if requested
403     s.protocolManager.Start(maxPeers)
404     if s.lesServer != nil {
405         s.lesServer.Start(srver)
406     }
407     return nil
408 }
```

앞장에서 알 수 있듯이 service.Start()의 호출로 실행되는 함수는 Ethereum 구조체의 메소드 Start()다.

이 메소드에서 중요한 로직은 **protocolManager**를 실행한다는 것이다. maxPeers를 인자로 받아 protocolManager의 Start()메소드를 실행하게 된다.

eth/handler.go | func Start() // s.protocolManager.Start()



Onther Inc.

```
203 func (pm *ProtocolManager) Start(maxPeers int) {
204     pm.maxPeers = maxPeers
205
206     // broadcast transactions
207     pm.txsCh = make(chan core.NewTxsEvent, txChanSize)
208     pm.txsSub = pm.txpool.SubscribeNewTxsEvent(pm.txsCh)
209     go pm.txBroadcastLoop()
210
211     // broadcast mined blocks
212     pm.minedBlockSub = pm.eventMux.Subscribe(core.NewMinedBlockEvent{})
213     go pm.minedBroadcastLoop()
214
215     // start sync handlers
216     go pm.syncer()
217     go pm.txsyncLoop()
218 }
```

프로토콜 매니저가 시작되면 총 4개의 루프가 돌게 된다.

txBroadcastLoop() : 트랜잭션을 공유하는 루프

minedBroadcastLoop() : 블록이 마이닝 되었을 때, 해당 블록을 피어들에게 전파하는 역할을 담당

syncer() : p2p레이어로부터 전달된 블록을 노드가 관리하는 체인에 추가하는 로직

txsyncLoop() : TxsyncCh 채널에서 이벤트가 발생했을 때 트랜잭션을 동기화하는 역할(노드가 처음 연결되었을 때 딱 한번, 대기중인 트랜잭션을 상대 노드와 동기화하는 용도로 사용된다. 그 이후에는 **txBroadcastLoop()**를 통해 동기화가 된다.)

// 자세한 로직은 다음에 살펴보는걸로 !

node/node.go | func Start() // stack.Start()



```
214     // Lastly start the configured RPC interfaces
215     if err := n.startRPC(services); err != nil {
216         for _, service := range services {
217             service.Stop()
218         }
219         running.Stop()
220         return err
221     }
222     // Finish initializing the startup
223     n.services = services
224     n.server = running
225     n.stop = make(chan struct{})
226
227     return nil
228 }
```

마지막으로 노드에 접속할 엔드포인트로 RPC 관련 기능을 실행합니다. startRPC를 들여다 보면 HTTP, WebSocket등 geth가 지원하는 여러 엔드포인트용 서비스를 실행합니다.

Start 메서드의 마지막입니다. 성공적으로 실행한 services와 running을 Node의 각각 services와 server 필드에 저장합니다.

2.1.1 Stop Node

cmd/utils/cmd.go | func StartNode()

```
66 func StartNode(stack *node.Node) {
67     if err := stack.Start(); err != nil {
68         Fatalf(format: "Error starting protocol stack: %v", err)
69     }
70     go func() {
71         sigc := make(chan os.Signal, 1)
72         signal.Notify(sigc, syscall.SIGINT, syscall.SIGTERM)
73         defer signal.Stop(sigc)
74         <-sigc
75         log.Info(msg: "Got interrupt, shutting down...")
76         go stack.Stop()
77         for i := 10; i > 0; i-- {
78             <-sigc
79             if i > 1 {
80                 log.Warn(msg: "Already shutting down, interrupt more to panic.", ctx: "times", i-1)
81             }
82         }
83         debug.Exit() // ensure trace and CPU profile data is flushed.
84         debug.LoudPanic(x: "boom")
85     }()
86 }
```

go func() {...}()와 같이 익명함수로 바로 고루틴을 실행시킨다. 먼저 Signal 타입의 채널 sigc를 생성한다. os.Signal을 본 적은 없지만 느낌상 golang에서 지원하는 운영체제의 시그널을 핸들링하는 타입으로 보입니다. 마찬가지로 이어서 signal 패키지의 Notify라는 함수에 sigc 채널변수와 상수처럼 보이는 SIGINT, SIGTERM을 전달하고 있습니다. 운영체제로부터 SIGINT나 SIGTERM 시그널을 받으면 sigc 채널에 무언가 알려달라고 등록하는 것 같군요. 모두 golang의 빌트인 패키지입니다. 내용을 찾아보면 실제로도 그려합니다.

이어서 defer키워드가 나오네요. 이미 함께 공부했듯이 signal.Stop(sigc)로작을 이 익명함수 실행의 마지막으로 미뤄뒀습니다.

이후 로직은 지난시간에 살펴본 채널의 문법을 사용하고 있습니다. 터미널에서 SIGINT, SIGTERM 시그널을 기다리다가 시그널 받으면 stack.Stop 호출합니다. 바로 앞서 살펴본 Stop 메서드를 실행하라는 의미겠지요? 그리고 아래 추가로 for문이 나옵니다만 이 로직은 강제로 서비스를 중지하는 기능으로 보입니다.

이렇게 함수가 종료되면 defer로 미뤄뒀던 signal.Stop(sigc)이 실행될 겁니다. 프로세스를 종료할 것이기 때문에 Notify로 등록한 SIGINT, SIGTERM신호를 더이상 기다릴 필요가 없다는 의미의 호출이란 것을 알 수 있습니다.

node/node.go | func Stop()



```
399     // Stop terminates a running node along with all it's services. In the node was
400     // not started, an error is returned.
401     func (n *Node) Stop() error {
402         n.lock.Lock()
403         defer n.lock.Unlock()
404
405         // Short circuit if the node's not running
406         if n.server == nil {
407             return ErrNodeStopped
408         }
409
410         // Terminate the API, services and the p2p server.
411         n.stopWS()
412         n.stopHTTP()
413         n.stopIPC()
414         n.rpcAPIs = nil
415         failure := &StopError{
416             Services: make(map[reflect.Type]error),
417         }
418         for kind, service := range n.services {
419             if err := service.Stop(); err != nil {
420                 failure.Services[kind] = err
421             }
422         }
423         n.server.Stop()
424         n.services = nil
425         n.server = nil
```

조립은 분해의 역순! 서비스의 종료도 시작한 것을 순차적으로 종료시키면 됩니다. Stop 메서드의 코드는 상대적으로 간단합니다. 이전에 살펴본 Node의 Start, Register와 동일하게 락을 잡고 서버의 실행 여부 확인 코드는 동일합니다.

이어서 서비스를 순차적으로 종료하고 중간에 종료에 실패한 서비스가 있다면 따로 담아두는 코드가 나옵니다.



Onther Inc.

node/node.go | func Stop()

```
427 // Release instance directory lock.  
428 if n.instanceDirLock != nil {  
429     if err := n.instanceDirLock.Release(); err != nil {  
430         n.log.Error( msg: "Can't release datadir lock", ctx: "err", err)  
431     }  
432     n.instanceDirLock = nil  
433 }  
434  
435 // unblock n.Wait  
436 close(n.stop)  
437  
438 // Remove the keystore if it was created ephemerally.  
439 var keystoreErr error  
440 if n.ephemeralKeystore != "" {  
441     keystoreErr = os.RemoveAll(n.ephemeralKeystore)  
442 }  
443  
444 if len(failure.Services) > 0 {  
445     return failure  
446 }  
447 if keystoreErr != nil {  
448     return keystoreErr  
449 }  
450  
451 }
```

node.Wait() 과 연결

이후 geth가 사용한 디렉토리에 대한 락을 해제합니다.

지난 글에서 node.Wait의 코드를 읽으면서 잠시 언급했었던 stop 채널을 닫는 close 함수를 호출합니다.

이후에는 임시로 등록한 계정 파일이 있으면 지우고 위에서 종료하다 실패한 서비스가 있으면 반환하는 등의 부수적인 처리로 함수의 실행을 마무리합니다.

2.2 Account Unlock

cmd/geth/main.go | func startNode()

```

267     // startNode boots up the system node and all registered protocols, after which
268     // it unlocks any requested accounts, and starts the RPC/IPC interfaces and the
269     // miner.
270     func startNode(ctx *cli.Context, stack *node.Node) {
271         debug.Memsize.Add( name: "node", stack)
272
273         // Start up the node itself
274         utils.StartNode(stack)
275
276         // Unlock any account specifically requested
277         ks := stack.AccountManager().Backends(keystore.KeyStoreType)[0].(*keystore.KeyStore)
278
279         passwords := utils.MakePasswordList(ctx)
280         unlocks := strings.Split(ctx.GlobalString(utils.UnlockedAccountFlag.Name), sep: ",")
281         for i, account := range unlocks {
282             if trimmed := strings.TrimSpace(account); trimmed != "" {
283                 unlockAccount(ctx, ks, trimmed, i, passwords)
284             }
285         }
286         // Register wallet event handlers to open and auto-derive wallets
287         events := make(chan accounts.WalletEvent, 16)
288         stack.AccountManager().Subscribe(events)

```

277 : 먼저 이전에 생성했던 키스토어를 가져와 ks 변수에 할당한다.

279: 패스워드 파일을 읽어서 각 패스워드를 line별로 나누어 모두 passwords에 할당한다.

280 : unlock할 어카운트를 unlocks에 ,(콤마)를 기준으로 잘라서 슬라이스의 아이템으로 넣는다. 즉 unlocks는 언락할 어카운트들의 슬라이스다.

281 ~ 285 : 그리고 for range 문을 돌려서 각 어카운트를 trim (white space를 잘라낸다)한 후, unlockAccount에 인자로 전달하여 어카운트들을 언락하게 된다.

287~288 : 그리고 이벤트 채널을 만들어 어카운트 매니저가 구독하게 한다.

cmd/geth/main.go | func startNode()



```
290 go func() {
291     // Create a chain state reader for self-derivation
292     rpcClient, err := stack.Attach()
293     if err != nil {
294         utils.Fatalf(format: "Failed to attach to self: %v", err)
295     }
296     stateReader := ethclient.NewClient(rpcClient)
297
298     // Open any wallets already attached
299     for _, wallet := range stack.AccountManager().Wallets() {
300         if err := wallet.Open( passphrase: ""); err != nil {
301             log.Warn( msg: "Failed to open wallet", ctx: "url", wallet.URL(), "err", err)
302         }
303     }
304
305     // Listen for wallet event till termination
306     for event := range events {
307         switch event.Kind {
308             case accounts.WalletArrived:
309                 if err := event.Wallet.Open( passphrase: ""); err != nil {
310                     log.Warn( msg: "New wallet appeared, failed to open", ctx: "url", event.Wallet.URL(), "err", err)
311                 }
312             case accounts.WalletOpened:
313                 status, _ := event.Wallet.Status()
314                 log.Info( msg: "New wallet appeared", ctx: "url", event.Wallet.URL(), "status", status)
315
316                 derivationPath := accounts.DefaultBaseDerivationPath
317                 if event.Wallet.URL().Scheme == "ledger" {
318                     derivationPath = accounts.DefaultLedgerBaseDerivationPath
319                 }
320                 event.Wallet.SelfDerive(derivationPath, stateReader)
321
322             case accounts.WalletDropped:
323                 log.Info( msg: "Old wallet dropped", ctx: "url", event.Wallet.URL())
324                 event.Wallet.Close()
325         }
326     }()
}
```

고루틴을 돌리면서 하는 일은 크게 세가지다.

1. 체인정보를 읽어들이는 `rpcClient` 생성
2. 등록된 지갑 `open`
3. 지갑 이벤트 수신 대기

cmd/geth/main.go | func startNode()

```

327     // Start auxiliary services if enabled
328     if ctx.GlobalBool(utils.MiningEnabledFlag.Name) || ctx.GlobalBool(utils.DeveloperFlag.Name) {
329         // Mining only makes sense if a full Ethereum node is running
330         if ctx.GlobalString(utils.SyncModeFlag.Name) == "light" {
331             utils.Fatalf( format: "Light clients do not support mining")
332         }
333         var ethereum *eth.Ethereum
334         if err := stack.Service(&ethereum); err != nil {
335             utils.Fatalf( format: "Ethereum service not running: %v", err)
336         }
337         // Use a reduced number of threads if requested
338         threads := ctx.GlobalInt(utils.MinerLegacyThreadsFlag.Name)
339         if ctx.GlobalIsSet(utils.MinerThreadsFlag.Name) {
340             threads = ctx.GlobalInt(utils.MinerThreadsFlag.Name)
341         }
342         if threads > 0 {
343             type threaded interface {
344                 SetThreads(threads int)
345             }
346             if th, ok := ethereum.Engine().(threaded); ok {
347                 th.SetThreads(threads)
348             }
349         }
350         // Set the gas price to the limits from the CLI and start mining
351         gasprice := utils.GlobalBig(ctx, utils.MinerLegacyGasPriceFlag.Name)
352         if ctx.IsSet(utils.MinerGasPriceFlag.Name) {
353             gasprice = utils.GlobalBig(ctx, utils.MinerGasPriceFlag.Name)
354         }
355         ethereum.TxPool().SetGasPrice(gasprice)
356         if err := ethereum.StartMining( local: true); err != nil {
357             utils.Fatalf( format: "Failed to start mining: %v", err)
358         }
359     }
360 }
```

328~360 : 마이닝 플래그를 주거나 dev모드시? 마이닝을 실행하는 로직이다.

330~336 : 라클은 마이닝 못하니까 예러 처리하고, Service메소드는 현재 등록되어 동작하는 이더리움 서비스를 리턴한다. 즉 현재 동작하는 서비스를 검색하는 것임

338~349 : 마이닝 관련해서 쓰레드 설정 잡아주는 로직이고

351~355 : 만약에 마이닝할 트랜잭션에 대해 설정할 미니멈 가스프라이스 기준이 있다면 세팅을 해준다.

356 : 드디어 StartMining()메소드를 호출하여 마이닝을 시작한다.

2.3 startMining()

-> go-ethereum 씹어먹기 #2에서 자세히 다룹니다

eth/backend.go | func StartMining()

```

336     func (s *Ethereum) StartMining(local bool) error {
337         eb, err := s.Etherbase()
338         if err != nil {
339             log.Error( msg: "Cannot start mining without etherbase", ctx: "err", err)
340             return fmt.Errorf( format: "etherbase missing: %v", err)
341         }
342         if clique, ok := s.engine.(*clique.Clique); ok {
343             wallet, err := s.accountManager.Find(accounts.Account{Address: eb})
344             if wallet == nil || err != nil {
345                 log.Error( msg: "Etherbase account unavailable locally", ctx: "err", err)
346                 return fmt.Errorf( format: "signer missing: %v", err)
347             }
348             clique.Authorize(eb, wallet.SignHash)
349         }
350         if local {
351             // If local (CPU) mining is started, we can disable the transaction rejection
352             // mechanism introduced to speed sync times. CPU mining on mainnet is ludicrous
353             // so none will ever hit this path, whereas marking sync done on CPU mining
354             // will ensure that private networks work in single miner mode too.
355             atomic.StoreUint32(&s.protocolManager.acceptTxs, val: 1)
356         }
357         go s.miner.Start(eb)
358     }
359 }
```

337~341 : 이더베이스 설정 잡아주고

342~349 : 만약 PoA라면 관련 설정 잡아주는 것 같음

350~356 : 정확히 뭔지 모르겠음

357 : 이더베이스를 인자로 전달하여 miner.Start() 메소드 실행

miner/miner.go | func Start()



```
107 func (self *Miner) Start(coinbase common.Address) {
108     atomic.StoreInt32(&self.shouldStart, val: 1)
109     self.SetEtherbase(coinbase)
110
111     if atomic.LoadInt32(&self.canStart) == 0 {
112         log.Info( msg: "Network syncing, will start miner afterwards")
113         return
114     }
115     self.worker.start()
116 }
```

```
41 // Miner creates blocks and searches for proof-of-work values.
42 type Miner struct {
43     mux    *event.TypeMux
44     worker *worker
45     coinbase common.Address
46     eth     Backend
47     engine  consensus.Engine
48     exitCh chan struct{}
49
50     canStart int32 // can start indicates whether we can start the mining operation
51     shouldStart int32 // should start indicates whether we should start after sync
52 }
```

108 : sync 이후에 마이닝 시작했다는 표시로 1을 저장한다

109 : 이더베이스 관련 설정

111~114 : 만약 canStart 가 0이라면 아직 네트워크 동기화가 진행중임을 의미하므로 로그를 찍고 실행을 종료한다.

115 : Miner.worker의 start()메소드를 실행한다.

miner/worker.go | func start()

```
// start sets the running status as 1 and triggers new work submitting.
97 // worker is the main object which takes care of submitting new work to consensus engine
98 // and gathering the sealing result.
99 type worker struct {
100     config *params.ChainConfig
101     engine consensus.Engine
102     eth Backend
103     chain *core.BlockChain
104
105     // Subscriptions
106     mux      *event.TypeMux
107     txsCh   chan core.NewTxsEvent
108     txsSub  event.Subscription
109     chainHeadCh chan core.ChainHeadEvent
110     chainHeadSub event.Subscription
111     chainSideCh chan core.ChainSideEvent
112     chainSideSub event.Subscription
113
114     // Channels
115     newWorkCh chan *newWorkReq
116     taskCh    chan *task
117     resultCh  chan *task
118     startCh   chan struct{}
119     exitCh    chan struct{}
120
121     current    *environment           // An environment for current running cycle.
122     possibleUncles map[common.Hash]*types.Block // A set of side blocks as the possible uncle blocks.
123     unconfirmed  *unconfirmedBlocks    // A set of locally mined blocks pending canonicalness confirmations.
124
125     mu sync.RWMutex // The lock used to protect the coinbase and extra fields
126     coinbase common.Address
127     extra []byte
128
129     snapshotMu sync.RWMutex // The lock used to protect the block snapshot and state snapshot
130     snapshotBlock *types.Block
131     snapshotState *state.StateDB
132
133     // Test hooks
134     newTaskHook func(*task)           // Method to call upon receiving a new sealing task
135     skipSealHook func(*task) bool     // Method to decide whether skipping the sealing.
136     fullTaskHook func()              // Method to call before pushing the full sealing task
137
138
139
140 }
```

108 : sync 이후에 마이닝 시작했다는 표시로 1을 저장한다

109 : 이더베이스 관련 설정

111~114 : 만약 canStart 가 0이라면 아직 네트워크동기화가 진행중임을 의미하므로 로그를 찍고 실행을 종료한다.

115 : Miner.worker의 start()메소드를 실행한다.

3. node.Wait()

cmd/geth/main.go | func geth()

```
254     // geth is the main entry point into the system if no special subcommand is ran.  
255     // It creates a default node based on the command line arguments and runs it in  
256     // blocking mode, waiting for it to be shut down.  
257     func geth(ctx *cli.Context) error {  
258         if args := ctx.Args(); len(args) > 0 {  
259             return fmt.Errorf(format: "invalid command: %q", args[0])  
260         }  
261         node := makeFullNode(ctx)  
262         startNode(ctx, node)  
263         node.Wait()  
264         return nil  
265     }
```

이제 마지막으로 `node.Wait()`을 살펴봅시다.

node/node.go | func Wait()

```
453 // Wait blocks the thread until the node is stopped. If the node is not running
454 // at the time of invocation, the method immediately returns.
455 func (n *Node) Wait() {
456     n.lock.RLock()
457     if n.server == nil {
458         n.lock.RUnlock()
459         return
460     }
461     stop := n.stop
462     n.lock.RUnlock()
463
464     <-stop
465 }
```

```
stop chan struct{} // Channel to wait for termination notifications
```

이 함수는 이름에서 드러나듯 전 단계인 `startNode(ctx, node)`에서 실행한 로직들이 모두 종료되기를 기다리는 함수입니다. 다음 글에서 다루겠지만 `startNode` 함수에서는 golang의 채널을 사용하여 스레드로 여러 로직을 실행합니다. 이러한 로직들이 정상적으로 종료되기를 기다리면서 `geth` 프로세스가 죽지 않도록 메인 프로세스를 대기하도록 하는 것이 이 함수가 하는 전부입니다.

`n.stop`은 `Node`의 `chan struct{}`타입의 필드이다. `stop` 채널에서 메세지를 전달받으면 `Wait()`함수는 실행이 종료되게 되는데, 앞서 **Node의 Stop메소드에 close(n.stop)**

즉, `stop` 채널을 닫는 로직이 있기 때문에, 이로 인해 `Wait()`함수가 종료되면서 자연스럽게 `geth`도 종료된다.

serviceFuncs 자세히 살펴보기

각 서비스들은 추후에 더욱 자세하게
다를 예정

cmd/geth/config.go | func makeFullNode()



```
153     func makeFullNode(ctx *cli.Context) *node.Node {
154         stack, cfg := makeConfigNode(ctx)
155
156         utils.RegisterEthService(stack, &cfg.Eth)
157
158         if ctx.GlobalBool(utils.DashboardEnabledFlag.Name) {
159             utils.RegisterDashboardService(stack, &cfg.Dashboard, gitCommit)
160         }
161         // Whisper must be explicitly enabled by specifying at least 1 whisper flag or in dev mode
162         shhEnabled := enableWhisper(ctx)
163         shhAutoEnabled := !ctx.GlobalIsSet(utils.WhisperEnabledFlag.Name) && ctx.GlobalIsSet(utils.DeveloperFlag.Name)
164         if shhEnabled || shhAutoEnabled {
165             if ctx.GlobalIsSet(utils.WhisperMaxMessageSizeFlag.Name) {
166                 cfg.Shh.MaxMessageSize = uint32(ctx.Int(utils.WhisperMaxMessageSizeFlag.Name))
167             }
168             if ctx.GlobalIsSet(utils.WhisperMinPOWFlag.Name) {
169                 cfg.Shh.MinimumAcceptedPOW = ctx.Float64(utils.WhisperMinPOWFlag.Name)
170             }
171             utils.RegisterShhService(stack, &cfg.Shh)
172         }
173
174         // Add the Ethereum Stats daemon if requested.
175         if cfg.Ethstats.URL != "" {
176             utils.RegisterEthStatsService(stack, cfg.Ethstats.URL)
177         }
178         return stack
179     }
```

serviceFuncs은 다양하게 있지만, 계속 강조했듯이 RegisterEthService만 알아도 큰 무리는 없다.

cmd/utils/flags.go | func RegisterEthService()

```

1195     // RegisterEthService adds an Ethereum client to the stack.
1196     func RegisterEthService(stack *node.Node, cfg *eth.Config) {
1197         var err error
1198         if cfg.SyncMode == downloader.LightSync {
1199             err = stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
1200                 return les.New(ctx, cfg)
1201             })
1202         } else {
1203             err = stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
1204                 fullNode, err := eth.New(ctx, cfg)
1205                 if fullNode != nil && cfg.LightServ > 0 {
1206                     ls, _ := les.NewLesServer(fullNode, cfg)
1207                     fullNode.AddLesServer(ls)
1208                 }
1209                 return fullNode, err
1210             })
1211         }
1212         if err != nil {
1213             Fatalf(format: "Failed to register the Ethereum service: %v", err)
1214         }
1215     }

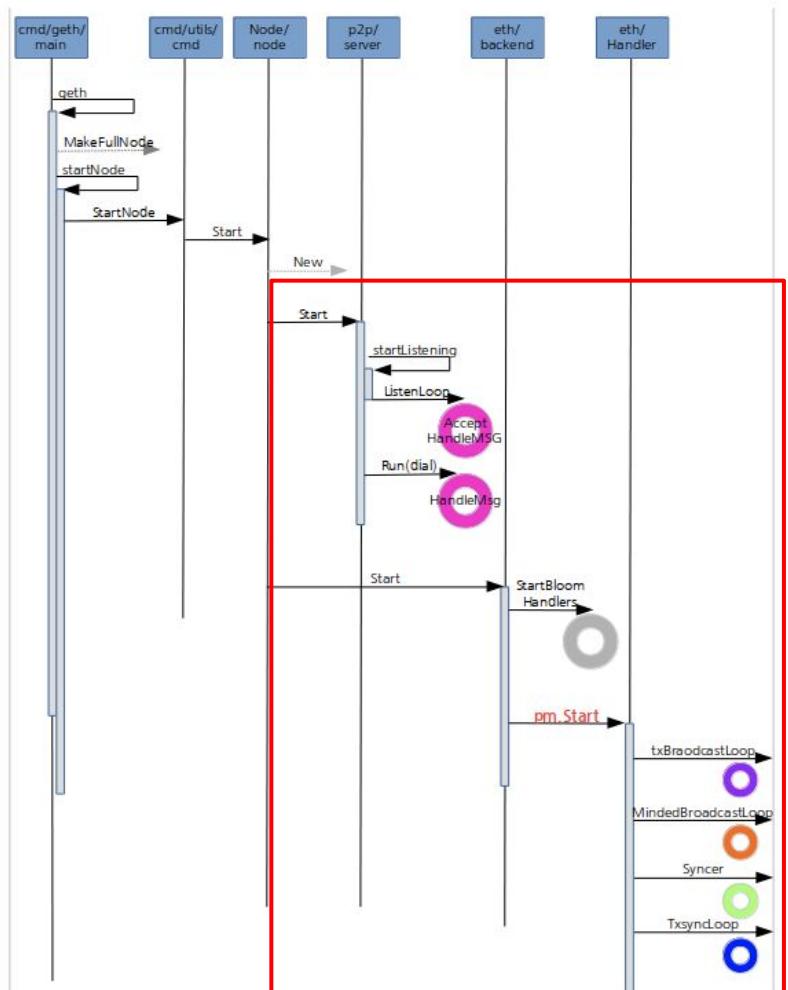
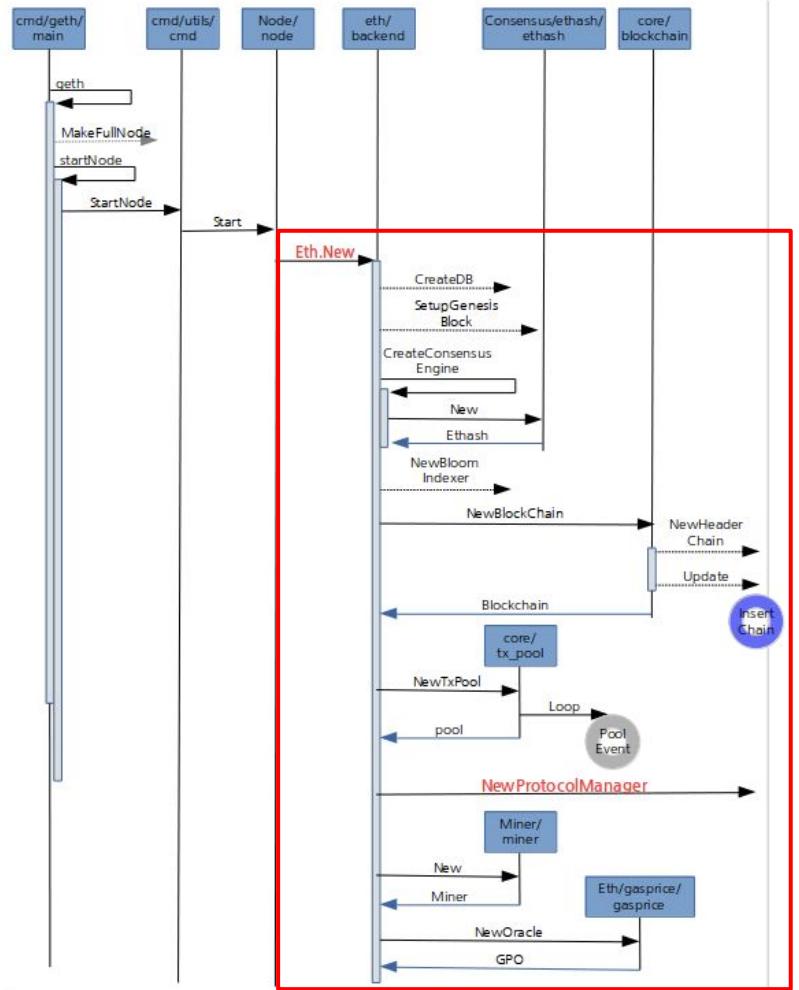
```

결국 serviceFuncs으로 등록되는 함수는 Register()의 내부 인자로 주어진 익명함수임

그러므로 이 함수를 까보면 됩니당

중요한건 eth.New(ctx, cfg)함수의 실행결과 fullNode를 리턴한다는 것인데, 그렇다면 eth.New()를 살펴보면 되겠죠

eth.New()



eth/backend.go | func New()



```
102     // New creates a new Ethereum object (including the
103     // initialisation of the common Ethereum object)
104     func New(ctx *node.ServiceContext, config *Config) (*Ethereum, error) {
105
106         // Ethereum implements the Ethereum full node service.
107         type Ethereum struct {
108             config      *Config
109             chainConfig *params.ChainConfig
110
111             // Channel for shutting down the service
112             shutdownChan chan bool // Channel for shutting down the Ethereum
113
114             // Handlers
115             txPool          *core.TxPool
116             blockchain     *core.BlockChain
117             protocolManager *ProtocolManager
118             lesServer       LesServer
119
120             // DB interfaces
121             chainDb ethdb.Database // Block chain database
122
123             eventMux        *event.TypeMux
124             engine          consensus.Engine
125             accountManager *accounts.Manager
126
127             bloomRequests chan chan *bloombits.Retrieval // Channel receiving bloom data retrieval requests
128             bloomIndexer   *core.ChainIndexer           // Bloom indexer operating during block imports
129
130             APIBackend *EthAPIBackend
131
132             miner          *miner.Miner
133             gasPrice       *big.Int
134             etherbase      common.Address
135
136             networkID      uint64
137             netRPCService  *ethapi.PublicNetAPI
138
139             lock sync.RWMutex // Protects the variadic fields (e.g. gas price and etherbase)
140         }
141     }
```

func New()는 이더리움 프로토콜에 대한 서비스들을 모두 초기 설정을 한 후에 이를 바탕으로 Ethereum구조체인 fullNode를 리턴한다. 그럼 함수 내부에서 어떤 설정을 하는지를 살펴보자.

eth/backend.go | func New()



```
102 // New creates a new Ethereum object (including the
103 // initialisation of the common Ethereum object)
104 func New(ctx *node.ServiceContext, config *Config) (*Ethereum, error) {
105     if config.SyncMode == downloader.LightSync {
106         return nil, errors.New(text: "can't run eth.Ethereum in light sync mode, use les.LightEthereum")
107     }
108     if !config.SyncMode.IsValid() {
109         return nil, fmt.Errorf(format: "invalid sync mode %d", config.SyncMode)
110     }
111     chainDb, err := CreateDB(ctx, config, name: "chaindata")
112     if err != nil {
113         return nil, err
114     }
115     chainConfig, genesisHash, genesisErr := core.SetupGenesisBlock(chainDb, config.Genesis)
116     if _, ok := genesisErr.(*params.ConfigCompatError); genesisErr != nil && !ok {
117         return nil, genesisErr
118     }
119     log.Info(msg: "Initialised chain configuration", ctx: "config", chainConfig)
```

105~110 : 시작은 언제나 이 함수를 실행하기 적절한지 검증하는 로직. 문제가 있으면 에러 객체와 함께 함수를 종료합니다.

111 : 제일 먼저 체인DB를 생성하는 로직이 나옵니다.

115 : 생성한 chainDb와 기본 Genesis 블록 설정을 가지고 core.SetupGenesisBlock 함수를 호출합니다.

geth는 체인DB로 구글이 오픈소스화한 레벨 db를 씁니다. 정확히는 레벨db는 C++로 구현된 것이고 여기서는 레벨db의 Go 구현체를 따로 씁니다.

<https://github.com/syndtr/goleveldb>

119 : 여기까지가 기본 설정이므로 다음과 같이 로그상에 체인 설정이 끝났다고 남깁니다.

eth/backend.go | func CreateDB()



Onther Inc.

```
200     // CreateDB creates the chain database.
201     func CreateDB(ctx *node.ServiceContext, config *Config, name string) (ethdb.Database, error) {
202         db, err := ctx.OpenDatabase(name, config.DatabaseCache, config.DatabaseHandles)
203         if err != nil {
204             return nil, err
205         }
206         if db, ok := db.(*ethdb.LDBDatabase); ok {
207             db.Meter( prefix: "eth/db/chaindata/" )
208         }
209         return db, nil
210     }
```

이 함수는 주어진 이름으로 Level DB를 생성하거나 이미 존재한다면 접속한다.
만약 노드가 수명이 짧은 노드라면, 메모리 DB를 리턴한다.

node/service.go | func OpenDatabase()



```
39 // OpenDatabase opens an existing database with the given name (or creates one
40 // if no previous can be found) from within the node's data directory. If the
41 // node is an ephemeral one, a memory database is returned.
42 func (ctx *ServiceContext) OpenDatabase(name string, cache int, handles int) (ethdb.Database, error) {
43     if ctx.config.DataDir == "" {
44         return ethdb.NewMemDatabase(), nil
45     }
46     db, err := ethdb.NewLDBDatabase(ctx.config.ResolvePath(name), cache, handles)
47     if err != nil {
48         return nil, err
49     }
50     return db, nil
51 }
```

이 함수는 주어진 이름으로 Level DB를 생성하거나 이미 존재한다면 접속한다.
만약 노드가 수명이 짧은 노드라면, 메모리 DB를 리턴한다.

core/genesis.go | func SetupGenesisBlock()



```
164 // DB가 비었을때 사용할 제네시스 블록.
165 // 기본적으로 비어있으므로 이더리움 메인넷 블록이 사용될것이다
166 func SetupGenesisBlock(db ethdb.Database, genesis *Genesis) (*params.ChainConfig, common.Hash, error) {
167     //제네시스 블록이 넘어왔을 경우
168     if genesis != nil && genesis.Config == nil {
169         return params.AllEthashProtocolChanges, common.Hash{}, errGenesisNoConfig
170     }
171
172     // Just commit the new block if there is no stored genesis block.
173     // core/rawdb/accessors_chain.go 참조
174     // db에서 제네시스 블록에 기록되어 있을 캐노니컬 해시를 읽는다.
175     // 블록의 해시가 없다면 혹은 비었다면 커먼해시를 리턴
176     stored := rawdb.ReadCanonicalHash(db, 0)
177     if (stored == common.Hash{}) {
178         //전달된 제네시스가 없다면 메인넷 블럭을 사용한다.
179         if genesis == nil {
180             log.Info("Writing default main-net genesis block")
181             genesis = DefaultGenesisBlock()
182         } else {
183             log.Info("Writing custom genesis block")
184         }
185         // 새블록을 commit한다(제네시스 블록)
186         block, err := genesis.Commit(db)
187         return genesis.Config, block.Hash(), err
188     }
189
190     // Check whether the genesis block is already written.
191     // 제네시스가 이미 있다면 해당블록의 해시를 체크한다
192     if genesis != nil {
193         hash := genesis.ToBlock(nil).Hash()
194         if hash != stored {
195             return genesis.Config, hash, &GenesisMismatchError{stored, hash}
196         }
197     }
```

core/genesis.go | func SetupGenesisBlock()



```
199 // Get the existing chain configuration.
200 // net의 config가 test net인지 main net인지 체크함.
201 newcfg := genesis.configOrDefault(stored)
202 storedcfg := rawdb.ReadChainConfig(db, stored)
203 if storedcfg == nil {
204     log.Warn("Found genesis block without chain config")
205     rawdb.WriteChainConfig(db, stored, newcfg)
206     return newcfg, stored, nil
207 }
208 // Special case: don't change the existing config of a non-mainnet chain if no new
209 // config is supplied. These chains would get AllProtocolChanges (and a compat error)
210 // if we just continued here.
211 // 새로운 config가 제공되지 않을 경우 메인넷 이외의 채널의 설정을 바꾸지 말것.
212 if genesis == nil && stored != params.MainnetGenesisHash {
213     return storedcfg, stored, nil
214 }
215
216 // Check config compatibility and write the config. Compatibility errors
217 // are returned to the caller unless we're already at block zero.
218 // 설정의 호환성을 체크하고 설정을 쓴다.
219 // 우리가 블록제로에 있지 않다면 호출자에게 호환에러가 반환될 것이다
220 height := rawdb.ReadHeaderNumber(db, rawdb.ReadHeadHeaderHash(db))
221 if height == nil {
222     return newcfg, stored, fmt.Errorf("missing block number for head header hash")
223 }
224 compatErr := storedcfg.CheckCompatible(newcfg, *height)
225 if compatErr != nil && *height != 0 && compatErr.RewindTo != 0 {
226     return newcfg, stored, compatErr
227 }
228 rawdb.WriteChainConfig(db, stored, newcfg)
229 return newcfg, stored, nil
230 }
```

eth/backend.go | func New()



```
121     eth := &Ethereum{  
122         config:           config,  
123         chainDb:          chainDb,  
124         chainConfig:       chainConfig,  
125         eventMux:         ctx.EventMux,  
126         accountManager:   ctx.AccountManager,  
127         engine:           CreateConsensusEngine(ctx, chainConfig, &config.Ethash, config.MinerNotify, chainDb),  
128         shutdownChan:     make(chan bool),  
129         networkID:        config.NetworkId,  
130         gasPrice:         config.GasPrice,  
131         etherbase:        config.Etherbase,  
132         bloomRequests:   make(chan chan *bloombits.Retrieval),  
133         bloomIndexer:     NewBloomIndexer(chainDb, params.BloomBitsBlocks, bloomConfirms),  
134     }  
135  
136     log.Info( msg: "Initialising Ethereum protocol",  ctx: "versions", ProtocolVersions, "network", config.NetworkId)
```

이어서 앞서 살펴본 Ethereum 타입의 객체를 생성합니다.

쭉 살펴보면 어떤 설정을 해주는지 알 수 있다. CreateConsensusEngine()은 ethash관련 설정

여기까지 진행되면 프로토콜 버전 및 네트워크 id를 한번 로그로 남깁니다.

eth/backend.go | func CreateConsensusEngine()



Onther Inc.

```
212     // CreateConsensusEngine creates the required type of consensus engine instance for an Ethereum service
213     func CreateConsensusEngine(ctx *node.ServiceContext, chainConfig *params.ChainConfig, config *ethash.Config, notify []string, db ethdb.Database)
214         consensus.Engine {
215             // If proof-of-authority is requested, set it up
216             if chainConfig.Clique != nil {
217                 return clique.New(chainConfig.Clique, db)
218             }
219             // Otherwise assume proof-of-work
220             switch config.PowMode {
221                 case ethash.ModeFake:
222                     log.Warn( msg: "Ethash used in fake mode")
223                     return ethash.NewFaker()
224                 case ethash.ModeTest:
225                     log.Warn( msg: "Ethash used in test mode")
226                     return ethash.NewTester( notify: nil)
227                 case ethash.ModeShared:
228                     log.Warn( msg: "Ethash used in shared mode")
229                     return ethash.NewShared()
230             default:
231                 engine := ethash.New(ethash.Config{
232                     CacheDir:        ctx.ResolvePath(config.CacheDir),
233                     CachesInMem:    config.CachesInMem,
234                     CachesOnDisk:   config.CachesOnDisk,
235                     DatasetDir:     config.DatasetDir,
236                     DatasetsInMem: config.DatasetsInMem,
237                     DatasetsOnDisk: config.DatasetsOnDisk,
238                 }, notify)
239                 engine.SetThreads( threads: -1) // Disable CPU mining
240                 return engine
241             }
242         }
```

이 함수는 이더리움 서비스를 위한 합의 엔진 타입을 생성한다. 엔진으로 ethash를 사용한다
(잘 보면 PoA로 할지 PoW할지에 따라 로직이 바뀌는걸 알 수 있다)

느낌만 보시면 될 듯

eth/backend.go | func New()



```
138     if !config.SkipBcVersionCheck {
139         bcVersion := rawdb.ReadDatabaseVersion(chainDb)
140         if bcVersion != core.BlockChainVersion && bcVersion != 0 {
141             return nil, fmt.Errorf("Blockchain DB version mismatch (%d / %d). Run geth upgradedb.\n", bcVersion,
142                                         core.BlockChainVersion)
143         }
144         rawdb.WriteDatabaseVersion(chainDb, core.BlockChainVersion)
145     }
146     var (
147         vmConfig      = vm.Config{EnablePreimageRecording: config.EnablePreimageRecording}
148         cacheConfig = &core.CacheConfig{Disabled: config.NoPruning, TrieNodeLimit: config.TrieCache, TrieTimeLimit:
149                                     config.TrieTimeout}
150     )
151     eth.blockchain, err = core.NewBlockChain(chainDb, cacheConfig, eth.chainConfig, eth.engine, vmConfig)
152     if err != nil {
153         return nil, err
154     }
155     // Rewind the chain in case of an incompatible config upgrade.
156     if compat, ok := genesisErr.(*params.ConfigCompatError); ok {
157         log.Warn(msg: "Rewinding chain to upgrade configuration", ctx: "err", compat)
158         eth.blockchain.SetHead(compat.RewindTo)
159         rawdb.WriteChainConfig(chainDb, genesisHash, chainConfig)
160     }
161     eth.bloomIndexer.Start(eth.blockchain)
```

138 : 버전 확인 : 현재 실행하는 버전이 유효한 블록체인 버전인지 체크하고 있습니다.

149 : 블록체인 생성 : 이 함수는 DB의 정보를 이용해서 완전히 초기화된 블록체인을 리턴한다. 이더리움의 기본 검증자와 처리자를 초기화한다. DB로부터 마지막으로 알려진 state를 읽어온다. 메인 계정 trie를 오픈하고 stateDB를 생성한다. 현재 블록과 현재 블록헤더를 설정하고 total difficulty를 계산한다. 5초마다 퓨처블록들을 체인에 추가하는 루틴 실행

154 : 다음은 메인 로직은 아닌듯 보이고 호환성 관련된 처리를 위해 체인을 조정하는 코드가 따로옵니다.

159 : 블룸필터 인덱서 시작 : 한 줄이지만 블룸필터 인덱서를 시작하는 코드가 이어집니다.

core/blockchain.go | func NewBlockChain()



```
134 // NewBlockChain returns a fully initialised block chain using information
135 // available in the database. It initialises the default Ethereum Validator and
136 // Processor.
137 func NewBlockChain(db ethdb.Database, cacheConfig *CacheConfig, chainConfig *params.ChainConfig, engine consensus.Engine, vmConfig vm.Config) (*BlockChain, error) {
138     if cacheConfig == nil {
139         cacheConfig = &CacheConfig{
140             TrieNodeLimit: 256 * 1024 * 1024,
141             TrieTimeLimit: 5 * time.Minute,
142         }
143     }
144     bodyCache, _ := lru.New(bodyCacheLimit)
145     bodyRLPCache, _ := lru.New(bodyCacheLimit)
146     blockCache, _ := lru.New(blockCacheLimit)
147     futureBlocks, _ := lru.New(maxFutureBlocks)
148     badBlocks, _ := lru.New(badBlockLimit)
149
150     bc := &BlockChain{
151         chainConfig: chainConfig,
152         cacheConfig: cacheConfig,
153         db: db,
154         triegc: prque.New(),
155         stateCache: state.NewDatabase(db),
156         quit: make(chan struct{}),
157         bodyCache: bodyCache,
158         bodyRLPCache: bodyRLPCache,
159         blockCache: blockCache,
160         futureBlocks: futureBlocks,
161         engine: engine,
162         vmConfig: vmConfig,
163         badBlocks: badBlocks,
164     }
165     bc.SetValidator(NewBlockValidator(chainConfig, bc, engine))
166     bc.SetProcessor(NewStateProcessor(chainConfig, bc, engine))
```

주석 : 이 함수는 DB의 정보를 이용해서 완전히 초기화된 블록체인을 리턴한다. 이더리움의 기본 검증자와 처리자를 초기화한다

검증자와 스테이트 처리자 설정

블록 검증자는 블록의 바디와 state에 대한 검증을 한다.

상태 처리자는 기본 처리자로서, 한 지점에서 다른 지점으로의 state의 변환을 관리한다

core/blockchain.go | type BlockChain struct



Onther Inc.

```
89 // type blockChain struct {
90     chainConfig *params.ChainConfig // Chain & network configuration
91     cacheConfig *CacheConfig       // Cache configuration for pruning
92
93     db      ethdb.Database // Low level persistent database to store final content in
94     triegc *prque.Prque   // Priority queue mapping block numbers to tries to gc
95     gcproc time.Duration // Accumulates canonical block processing for trie dumping
96
97     hc      *HeaderChain
98     rmLogsFeed event.Feed
99     chainFeed event.Feed
100    chainSideFeed event.Feed
101    chainHeadFeed event.Feed
102    logsFeed event.Feed
103    scope    event.SubscriptionScope
104    genesisBlock *types.Block
105
106    mu      sync.RWMutex // global mutex for locking chain operations
107    chainmu sync.RWMutex // blockchain insertion lock
108    procmu sync.RWMutex // block processor lock
109
110    checkpoint int        // checkpoint counts towards the new checkpoint
111    currentBlock atomic.Value // Current head of the block chain
112    currentFastBlock atomic.Value // Current head of the fast-sync chain (may be above the block chain!)
113
114    stateCache state.Database // State database to reuse between imports (contains state cache)
115    bodyCache  *lru.Cache    // Cache for the most recent block bodies
116    bodyRLPCache *lru.Cache  // Cache for the most recent block bodies in RLP encoded format
117    blockCache *lru.Cache    // Cache for the most recent entire blocks
118    futureBlocks *lru.Cache // future blocks are blocks added for later processing
119
120    quit    chan struct{} // blockchain quit channel
121    running int32         // running must be called atomically
122    // procInterrupt must be atomically called
123    procInterrupt int32    // interrupt signaler for block processing
124    wg      sync.WaitGroup // chain processing wait group for shutting down
125
126    engine  consensus.Engine
127    processor Processor // block processor interface
128    validator Validator // block and state validator interface
129    vmConfig vm.Config
130
131    badBlocks *lru.Cache // Bad block cache
132 }
```

앞의 **setValidator**, **setProcessor** 메소드의 실행 결과
bc.processor, **bd.validator** 필드에 각각 프로세서와
밸리데이터가 할당이 된다.

core/types.go | type Validator



```
25 // Validator is an interface which defines the standard for block validation. It
26 // is only responsible for validating block contents, as the header validation is
27 // done by the specific consensus engines.
28 /**
29 ① type Validator interface {
30     // ValidateBody validates the given block's content.
31 ①     ValidateBody(block *types.Block) error
32
33     // ValidateState validates the given statedb and optionally the receipts and
34     // gas used.
35 ①     ValidateState(block, parent *types.Block, state *state.StateDB, receipts types.Receipts, usedGas uint64) error
36 }
```

Validator 타입이 하는 역할은 block의 contents나 statedb같은 것들만을 검증한다. (블록 헤더 부분은 컨센서스 엔진 부분에서 검증이 이루어진다고 함)

ValidateBody validates the given block's uncles and verifies the the block header's transaction and uncle roots. The headers are assumed to be already validated at this point.

ValidateState validates the various changes that happen after a state transition, such as amount of used gas, the receipt roots and the state root itself.

ValidateState returns a database batch if the validation was a success otherwise nil and an error is returned.

core/types.go | type Processor



Onther Inc.

```
38 // Processor is an interface for processing blocks using a given initial state.
39 //
40 // Process takes the block to be processed and the statedb upon which the
41 // initial state is based. It should return the receipts generated, amount
42 // of gas used in the process and return an error if any of the internal rules
43 // failed.
44 type Processor interface {
45     Process(block *types.Block, statedb *state.StateDB, cfg vm.Config) (types.Receipts, []*types.Log, uint64, error)
46 }
47
48 func (p *StateProcessor) Process(block *types.Block, statedb *state.StateDB, cfg vm.Config) (types.Receipts, []*types.Log, uint64, error) {
49     var (
50         receipts types.Receipts
51         usedGas = new(uint64)
52         header = block.Header()
53         allLogs []*types.Log
54         gp = new(GasPool).AddGas(block.GasLimit())
55     )
56
57     // Mutate the block and state according to any hard-fork specs
58     if p.config.DAOForkSupport && p.config.DAOForkBlock != nil && p.config.DAOForkBlock.Cmp(block.Number()) == 0 {
59         misc.ApplyDAOHardFork(statedb)
60     }
61
62     // Iterate over and process the individual transactions
63     for i, tx := range block.Transactions() {
64         statedb.Prepare(tx.Hash(), block.Hash(), i)
65         receipt, _, err := ApplyTransaction(p.config, p.bc, author: nil, gp, statedb, header, tx, usedGas, cfg)
66         if err != nil {
67             return nil, nil, 0, err
68         }
69         receipts = append(receipts, receipt)
70         allLogs = append(allLogs, receipt.Logs...)
71     }
72
73     // Finalize the block, applying any consensus engine specific extras (e.g. block rewards)
74     p.engine.Finalize(p.bc, header, statedb, block.Transactions(), block.Uncles(), receipts)
75
76     return receipts, allLogs, *usedGas, nil
77 }
```

프로세서는 주어진 상태에서 블록을 처리하여 state를 변경하는 친구이다.

빨간색 표시된 곳을 보면 트랜잭션들을 처리하는 로직이 있다. ApplyTransaction이 보일 것이다.

프로세스를 하고 나면 리시트, 사용된 가스 등이 리턴된다.

core/blockchain.go | func NewBlockChain()

```

168     var err error
169     bc.hc, err = NewHeaderChain(db, chainConfig, engine, bc.getProcInterrupt)
170     if err != nil {
171         return nil, err
172     }
173     bc.genesisBlock = bc.GetBlockByNumber( number: 0 )
174     if bc.genesisBlock == nil {
175         return nil, ErrNoGenesis
176     }
177     if err := bc.loadLastState(); err != nil {
178         return nil, err
179     }
180     // Check the current state of the block hashes and make sure that we do not have any of the bad blocks in our chain
181     for hash := range BadHashes {
182         if header := bc.GetHeaderByHash(hash); header != nil {
183             // get the canonical block corresponding to the offending header's number
184             headerByNumber := bc.GetHeaderByNumber(header.Number.Uint64())
185             // make sure the headerByNumber (if present) is in our current canonical chain
186             if headerByNumber != nil && headerByNumber.Hash() == header.Hash() {
187                 log.Error( msg: "Found bad hash, rewinding chain", ctx: "number", header.Number, "hash", header.ParentHash)
188                 bc.SetHead(header.Number.Uint64() - 1)
189                 log.Error( msg: "Chain rewind was successful, resuming normal operation")
190             }
191         }
192     }
193     // Take ownership of this particular state
194     go bc.update()
195     return bc, nil
196 }
```

169 : 다시 NewBlockChain() 함수로 돌아오면, 먼저 새로운 헤더체인 구조체를 생성하여 bc.hc에 할당한다.

173 : bc.GetBlockByNumber 메소드를 실행하여 블록넘버를 기준으로 0번째 블록, 즉 제네시스 블록을 가져와서 bc.genesisBlock에 할당한다.

177 : 그리고 bc.loadLastState() 메소드를 실행한다. 이 메소드는 DB로부터 마지막으로 알려진 state를 읽어오고, 메인 계정 trie를 오픈하고 stateDB를 생성한다. 또한 현재 블록과 블록헤더를 설정하고 total difficulty를 계산한다.

181 : 블록 해시들의 상태를 체크하고 배드블럭들이 체인에 없는지 체크한다.

194 : 마지막으로 bc.update() 메소드를 실행하여 5초마다 퓨처블록들을 체인에 추가하는 루틴 실행

eth/backend.go | func New()

```
161     if config.TxPool.Journal != "" {
162         config.TxPool.Journal = ctx.ResolvePath(config.TxPool.Journal)
163     }
164     eth.txPool = core.NewTxPool(config.TxPool, eth.chainConfig, eth.blockchain)
165
166     if eth.protocolManager, err = NewProtocolManager(eth.chainConfig, config.SyncMode, config.NetworkId, eth.eventMux,
167         eth.txPool, eth.engine, eth.blockchain, chainDB); err != nil {
168         return nil, err
169     }
170
171     eth.miner = miner.New(eth, eth.chainConfig, eth.EventMux(), eth.engine)
172     eth.miner.SetExtra(makeExtraData(config.ExtraData))
173
174     eth.APIBackend = &EthAPIBackend{eth: eth, gpo: nil}
175     gpoParams := config.GPO
176     if gpoParams.Default == nil {
177         gpoParams.Default = config.GasPrice
178     }
179     eth.APIBackend.gpo = gasprice.NewOracle(eth.APIBackend, gpoParams)
180
181     return eth, nil
182 }
```

다시 func New() 함수를 보자

트랜잭션 Pool 생성 : 블록체인에 담을 트랜잭션 Pool을 생성하고 체인 헤드 이벤트를 구독한다.

프로토콜 매니저 : 이더리움 서브프로토콜은 이더리움 네트워크에서 동작 가능한 피어들을 관리한다. 해쉬나 블록을 원격피어로 부터 가져오는

다운로더를 만든다. Qos 튜너는 산발적으로 피어들의 지연속도를 모아 예측시간을 업데이트 한다. statefetcher는 피어 일동의 active state 동기화 및 요청 수락을 관리한다. 해쉬 어나운스먼트를 베이스로 블록을 검색하는 블록패쳐를 만든다

マイ닝 설정 : 다운로드 이벤트를 트래킹(한 번), 체인이 sync 되었는지 실패했는지 확인한다.

가스 오라클 설정 : 가스 오라클 관련 코드

여기까지 설정을 마치고 eth 즉, *Ethereum 타입의 구조체를 fullNode 변수에 리턴한다.

eth/handler.go | func NewProtocolManager()



Onther Inc.

```
99 // NewProtocolManager returns a new Ethereum sub protocol manager. The Ethereum sub protocol manages peers capable
100 // with the Ethereum network.
101 func NewProtocolManager(config *params.ChainConfig, mode downloader.SyncMode, networkID uint64, mux *event.TypeMux, txpool txPool,
102 engine consensus.Engine, blockchain *core.BlockChain, chaindb ethdb.Database) (*ProtocolManager, error) {
103     // Create the protocol manager with the base fields
104     manager := &ProtocolManager{
105         networkID:   networkID,
106         eventMux:    mux,
107         txpool:      txpool,
108         blockchain: blockchain,
109         chainconfig: config,
110         peers:       newPeerSet(),
111         newPeerCh:   make(chan *peer),
112         noMorePeers: make(chan struct{}),
113         txsyncCh:   make(chan *txsync),
114         quitSync:   make(chan struct{}),
115     }
116     return manager, nil
117 }
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203     func (pm *ProtocolManager) Start(maxPeers int) {
204         pm.maxPeers = maxPeers
205
206         // broadcast transactions
207         pm.txsCh = make(chan core.NewTxsEvent, txChanSize)
208         pm.txsSub = pm.txpool.SubscribeNewTxsEvent(pm.txsCh)
209         go pm.txBroadcastLoop()
210
211         // broadcast mined blocks
212         pm.minedBlockSub = pm.eventMux.Subscribe(core.NewMinedBlockEvent{})
213         go pm.minedBroadcastLoop()
214
215         // start sync handlers
216         go pm.syncer()
217         go pm.txsyncLoop()
218     }
```

프로토콜 매니저는 중요한 친구임. 이 친구가 사실상 블록 동기화나 트랜잭션 브로드캐스팅 등과 같은 p2p 프로토콜의 실질적인 부분들을 담당하고 있음
(오른쪽에 있는) Protocolmanager의 메소드인 Start()가 바로 앞에서 살펴봤듯이 그런 역할을 수행하는 친구!