node在go ethereum中代表了一个节点。 可能是全节点，可能是轻量级节点。 node可以理解为一个进程，以太坊由运行在世界各地的很多中类型的node组成。

一个典型的node就是一个p2p的节点。 运行了p2p网络协议，同时根据节点类型不同，运行了不同的业务层协议(以区别网络层协议。 参考p2p peer中的Protocol接口)。

node的结构。

```
// 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 req
uests

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

节点的初始化, 节点的初始化并不依赖其他的外部组件， 只依赖一个Config对象。

```
// New creates a new P2P node, ready for protocol registration.
func New(conf *Config) (*Node, error) {
    // Copy config and resolve the datadir so future changes to the current
    // working directory don't affect the node.
    confCopy := *conf
    conf = &confCopy
    if conf.DataDir != "" {  //转化为绝对路径。
        absdatadir, err := filepath.Abs(conf.DataDir)
        if err != nil {
            return nil, err
        }
        conf.DataDir = absdatadir
    }
    // Ensure that the instance name doesn't cause weird conflicts with
    // other files in the data directory.
    if strings.ContainsAny(conf.Name, `/\`) {
        return nil, errors.New(`Config.Name must not contain '/' or '\'`)
    }
    if conf.Name == datadirDefaultKeyStore {
        return nil, errors.New(`Config.Name cannot be "` + datadirDefaultKeyStore +
 `"`)
    }
    if strings.HasSuffix(conf.Name, ".ipc") {
        return nil, errors.New(`Config.Name cannot end in ".ipc"`)
    }
    // Ensure that the AccountManager method works before the node has started.
    // We rely on this in cmd/geth.
    am, ephemeralKeystore, err := makeAccountManager(conf)
    if err != nil {
        return nil, err
    }
    // Note: any interaction with Config that would create/touch files
    // in the data directory or instance directory is delayed until Start.
    return &Node{
        accman:            am,
        ephemeralKeystore: ephemeralKeystore,
        config:            conf,
        serviceFuncs:      []ServiceConstructor{},
        ipcEndpoint:       conf.IPCEndpoint(),
        httpEndpoint:      conf.HTTPEndpoint(),
        wsEndpoint:        conf.WSEndpoint(),
        eventmux:          new(event.TypeMux),
    }, nil
}
```

node 服务和协议的注册

因为node并没有负责具体的业务逻辑。所以具体的业务逻辑是通过注册的方式来注册到node里面来的。
其他模块通过Register方法来注册了一个 服务构造函数。 使用这个服务构造函数可以生成服务。

```
// Register injects a new service into the node's stack. The service created by
// the passed constructor must be unique in its type with regard to sibling ones.
func (n *Node) Register(constructor ServiceConstructor) error {
    n.lock.Lock()
    defer n.lock.Unlock()

    if n.server != nil {
        return ErrNodeRunning
    }
    n.serviceFuncs = append(n.serviceFuncs, constructor)
    return nil
}
```

服务是什么

```
type ServiceConstructor func(ctx *ServiceContext) (Service, error)
// Service is an individual protocol that can be registered into a node.
//
// Notes:
//
// • Service life-cycle management is delegated to the node. The service is allowed
 to
// initialize itself upon creation, but no goroutines should be spun up outside of
the
// Start method.
//
// • Restart logic is not required as the node will create a fresh instance
// every time a service is started.

// 服务的生命周期管理已经代理给node管理。该服务允许在创建时自动初始化，但是在Start方法之外
不应该启动goroutines。
```

```
// 重新启动逻辑不是必需的，因为节点将在每次启动服务时创建一个新的实例。
type Service interface {
    // Protocols retrieves the P2P protocols the service wishes to start.
    // 服务希望提供的p2p协议
    Protocols() []p2p.Protocol

    // APIs retrieves the list of RPC descriptors the service provides
    // 服务希望提供的RPC方法的描述
    APIs() []rpc.API

    // Start is called after all services have been constructed and the networking
    // layer was also initialized to spawn any goroutines required by the service.
    // 所有服务已经构建完成后，调用开始，并且网络层也被初始化以产生服务所需的任何goroutine
。
    Start(server *p2p.Server) error

    // Stop terminates all goroutines belonging to the service, blocking until they
    // are all terminated.

    // Stop方法会停止这个服务拥有的所有goroutine。 需要阻塞到所有的goroutine都已经终止
    Stop() error
}
```

node的启动

node的启动过程会创建和运行一个p2p的节点。

```
// Start create a live P2P node and starts running it.
func (n *Node) Start() error {
    n.lock.Lock()
    defer n.lock.Unlock()

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

    // Initialize the p2p server. This creates the node key and
    // discovery databases.
    n.serverConfig = n.config.P2P
    n.serverConfig.PrivateKey = n.config.NodeKey()
    n.serverConfig.Name = n.config.NodeName()
    if n.serverConfig.StaticNodes == nil {
```

```go
        // 处理配置文件static-nodes.json
        n.serverConfig.StaticNodes = n.config.StaticNodes()
    }
    if n.serverConfig.TrustedNodes == nil {
        // 处理配置文件trusted-nodes.json
        n.serverConfig.TrustedNodes = n.config.TrustedNodes()
    }
    if n.serverConfig.NodeDatabase == "" {
        n.serverConfig.NodeDatabase = n.config.NodeDB()
    }
    //创建了p2p服务器
    running := &p2p.Server{Config: n.serverConfig}
    log.Info("Starting peer-to-peer node", "instance", n.serverConfig.Name)

    // Otherwise copy and specialize the P2P configuration
    services := make(map[reflect.Type]Service)
    for _, constructor := range n.serviceFuncs {
        // Create a new context for the particular service
        ctx := &ServiceContext{
            config:         n.config,
            services:       make(map[reflect.Type]Service),
            EventMux:       n.eventmux,
            AccountManager: n.accman,
        }
        for kind, s := range services { // copy needed for threaded access
            ctx.services[kind] = s
        }
        // Construct and save the service
        // 创建所有注册的服务。
        service, err := constructor(ctx)
        if err != nil {
            return err
        }
        kind := reflect.TypeOf(service)
        if _, exists := services[kind]; exists {
            return &DuplicateServiceError{Kind: kind}
        }
        services[kind] = service
    }
    // Gather the protocols and start the freshly assembled P2P server
    // 收集所有的p2p的protocols并插入p2p.Rrotocols
    for _, service := range services {
        running.Protocols = append(running.Protocols, service.Protocols()...)
    }
    // 启动了p2p服务器
    if err := running.Start(); err != nil {
        return convertFileLockError(err)
    }
    // Start each of the services
    // 启动每一个服务
    started := []reflect.Type{}
    for kind, service := range services {
        // Start the next service, stopping all previous upon failure
        if err := service.Start(running); err != nil {
            for _, kind := range started {
                services[kind].Stop()
            }
            running.Stop()

            return err
        }
        // Mark the service started for potential cleanup
        started = append(started, kind)
    }
    // Lastly start the configured RPC interfaces
    // 最后启动RPC服务
    if err := n.startRPC(services); err != nil {
        for _, service := range services {
            service.Stop()
        }
        running.Stop()
        return err
    }
    // Finish initializing the startup
    n.services = services
    n.server = running
    n.stop = make(chan struct{})

    return nil
}
```

startRPC,这个方法收集所有的apis。 并依次调用启动各个RPC服务器， 默认是启动InProc和IPC。 如果指定也可以配置是否启动HTTP和websocket。

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

startXXX 是具体的RPC的启动。 流程都是大同小异。 这里就只看startWS了

```
// startWS initializes and starts the websocket RPC endpoint.
func (n *Node) startWS(endpoint string, apis []rpc.API, modules []string, wsOrigins
 []string, exposeAll bool) error {
    // Short circuit if the WS endpoint isn't being exposed
    if endpoint == "" {
        return nil
    }
    // Generate the whitelist based on the allowed modules
    // 生成白名单
    whitelist := make(map[string]bool)
    for _, module := range modules {
        whitelist[module] = true
    }
    // Register all the APIs exposed by the services
    handler := rpc.NewServer()
    for _, api := range apis {
        if exposeAll || whitelist[api.Namespace] || (len(whitelist) == 0 && api.Pub
lic) {
        // 只有这集中情况下才会把这个api进行注册。
            if err := handler.RegisterName(api.Namespace, api.Service); err != nil
{
```

```
                return err
            }
            log.Debug(fmt.Sprintf("WebSocket registered %T under '%s'", api.Service
, api.Namespace))
        }
    }
    // All APIs registered, start the HTTP listener
    var (
        listener net.Listener
        err      error
    )
    if listener, err = net.Listen("tcp", endpoint); err != nil {
        return err
    }
    go rpc.NewWSServer(wsOrigins, handler).Serve(listener)
    log.Info(fmt.Sprintf("WebSocket endpoint opened: ws://%s", listener.Addr()))

    // All listeners booted successfully
    n.wsEndpoint = endpoint
    n.wsListener = listener
    n.wsHandler = handler

    return nil
}
```