# 以太坊p2p网络应用部分分享

## 0、引导：

这部分内容是为了与p2p部分不大，只是一个额外了解部分

分析以太坊源码，总要有一个引子，从引子一直往下学习，那么最终就会了解全套以太坊如何运作。

下面先看一下一个以太坊节点启动部分：

目录：cmd/geth/main.go

```
// geth is the main entry point into the system if no special subcommand is ran.
// It creates a default node based on the command line arguments and runs it in
// blocking mode, waiting for it to be shut down.
func geth(ctx *cli.Context) error {
    node := makeFullNode(ctx) //根据context创建一个完整的节点
    startNode(ctx, node) //启动本节点
    node.Wait()
    return nil
}
```

目录：cmd/geth/config.go

```
func makeFullNode(ctx *cli.Context) *node.Node {
    stack, cfg := makeConfigNode(ctx) //创建一个加了config的节点，默认配置在defauts.go
    utils.RegisterEthService(stack, &cfg.Eth) //给节点注册以太坊服务，实现在flags.go
    .......
    if shhEnabled || shhAutoEnabled {
        .........
        utils.RegisterShhService(stack, &cfg.Shh) //给节点注册shh服务
    }
    // Add the Ethereum Stats daemon if requested.
    if cfg.Ethstats.URL != "" {
        utils.RegisterEthStatsService(stack, cfg.Ethstats.URL)
    }

    // Add the release oracle service so it boots along with node.
    if err := stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
        .........
    }
    return stack
}
```

从上面可以看出 ，这一步将各种服务注册到Node上，下一步主要看注册以太坊的服务

```
// RegisterEthService adds an Ethereum client to the stack.
func RegisterEthService(stack *node.Node, cfg *eth.Config) {
        .......
        //这里调用了node来注册服务，将一个"构造函数"传递给node进行注册
        err = stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
            fullNode, err := eth.New(ctx, cfg) //此处创建一个以太坊服务节点
            if fullNode != nil && cfg.LightServ > 0 {
                ls, _ := les.NewLesServer(fullNode, cfg)
                fullNode.AddLesServer(ls)
            }
```

```
            return fullNode, err
        })
        .......
}
// 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
    }
    //在node的服务函数变量里面填加函数
    n.serviceFuncs = append(n.serviceFuncs, constructor)
    return nil
}
```

上面主要做的工作是创建一个Node实例，将各项配置同步到Node属性上，紧接着将要在Node上运行的服务注册到Node，下一步就是启动Node

```
// startNode boots up the system node and all registered protocols, after which
// it unlocks any requested accounts, and starts the RPC/IPC interfaces and the
// miner.
func startNode(ctx *cli.Context, stack *node.Node) {
    // Start up the node itself
    utils.StartNode(stack)
    ........
}

func StartNode(stack *node.Node) {
    if err := stack.Start(); err != nil {
        Fatalf("Error starting protocol stack: %v", err)
    }
}
--------------------------------------------------------------------------------------------
// 下面就与我们第一次分享的内容连接上，但是再往下分析的路线不同于第一次，提升到p2p应用层
// Start create a live P2P node and starts running it.
func (n *Node) Start() error {
    ............
    running := &p2p.Server{Config: n.serverConfig}
    log.Info("Starting peer-to-peer node", "instance", n.serverConfig.Name)

    // Otherwise copy and specialize the P2P configuration
    //此处就是我们搭载在node上的所有服务
    services := make(map[reflect.Type]Service)
    for _, constructor := range n.serviceFuncs {
        //下面对各个服务进初步打理，构造context  去重等等。
        // 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
        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
    //这里将各个服务的协议添加到p2p服务模块的协议里面
    for _, service := range services {
        running.Protocols = append(running.Protocols, service.Protocols()...)
    }
    //启动p2p模块，这个第一次分享讲过
    if err := running.Start(); err != nil {
        if errno, ok := err.(syscall.Errno); ok && datadirInUseErrnos[uint(errno)] {
            return ErrDatadirUsed
        }
        return err
    }
    // Start each of the services
    started := []reflect.Type{}
    //遍历所有服务，启动所有服务，并把p2p模块作为各个node上服务的基础模块
    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)
    }
    ..................
}
```

现在就以太坊服务启动进行分析，引入到p2p应用层的讲解

```
// Start implements node.Service, starting all internal goroutines needed by the
// Ethereum protocol implementation.
func (s *Ethereum) Start(srvr *p2p.Server) error {
    s.netRPCService = ethapi.NewPublicNetAPI(srvr, s.NetVersion())

    s.protocolManager.Start() //这一步开启p2p
    if s.lesServer != nil {
        s.lesServer.Start(srvr)
    }
    return nil
}
```

---

额外讲解下p2p底层模块的peer如何和protocolManager里面的上层peer连接

回想第一次分享，一个peer查找到后经过各种握手最终被添加到p2p.server的peer列表的代码：

```
    case c := <-srv.addpeer:
            // At this point the connection is past the protocol handshake.
            // Its capabilities are known and the remote identity is verified.
            err := srv.protoHandshakeChecks(peers, c)
            if err == nil {
                // The handshakes are done and it passed all checks.
                p := newPeer(c, srv.Protocols)
                name := truncateName(c.name)
                log.Debug("Adding p2p peer", "id", c.id, "name", name, "addr", c.fd.RemoteAddr(),
"peers", len(peers)+1)
                peers[c.id] = p
                go srv.runPeer(p)
            }
            ........
```

下面看ser.runPeer()

```
// runPeer runs in its own goroutine for each peer.
// it waits until the Peer logic returns and removes
// the peer.
func (srv *Server) runPeer(p *Peer) {
    if srv.newPeerHook != nil {
        srv.newPeerHook(p)
    }
    remoteRequested, err := p.run() //这一句是重点
    // Note: run waits for existing peers to be sent on srv.delpeer
    // before returning, so this send should not select on srv.quit.
    srv.delpeer <- peerDrop{p, err, remoteRequested}
}
```

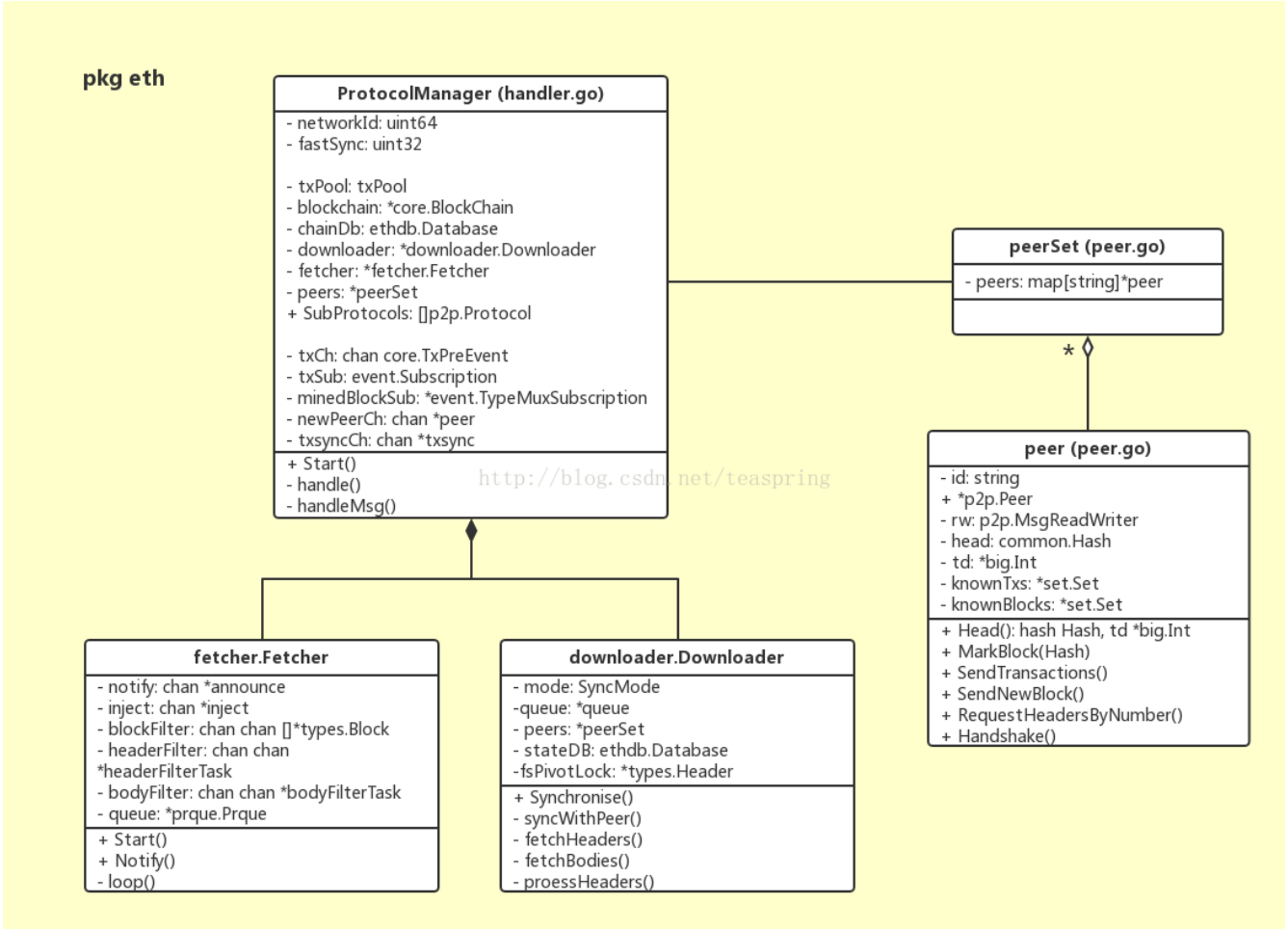
再看p.run()

```
func (p *Peer) run() (remoteRequested bool, err error) {
    .......
    p.startProtocols(writeStart, writeErr)//此句为重点，省略其他部分
    ........
}

func (p *Peer) startProtocols(writeStart <-chan struct{}, writeErr chan<- error) {
            ..............
            err := proto.Run(p, proto)//在此就是调用protocolManager中定义的Run函数将底层peer传给上层
peer做组合
            ..............
}
```

由此我们进入第一节的分享

## 1、以太坊p2p通信的管理模块ProtocolManager

以太坊中，管理个体间p2p通信的顶层结构体叫eth.ProtocolManager，它也是eth.Ethereum的核心成员变量之一。先来看一下它的主要UML关系：

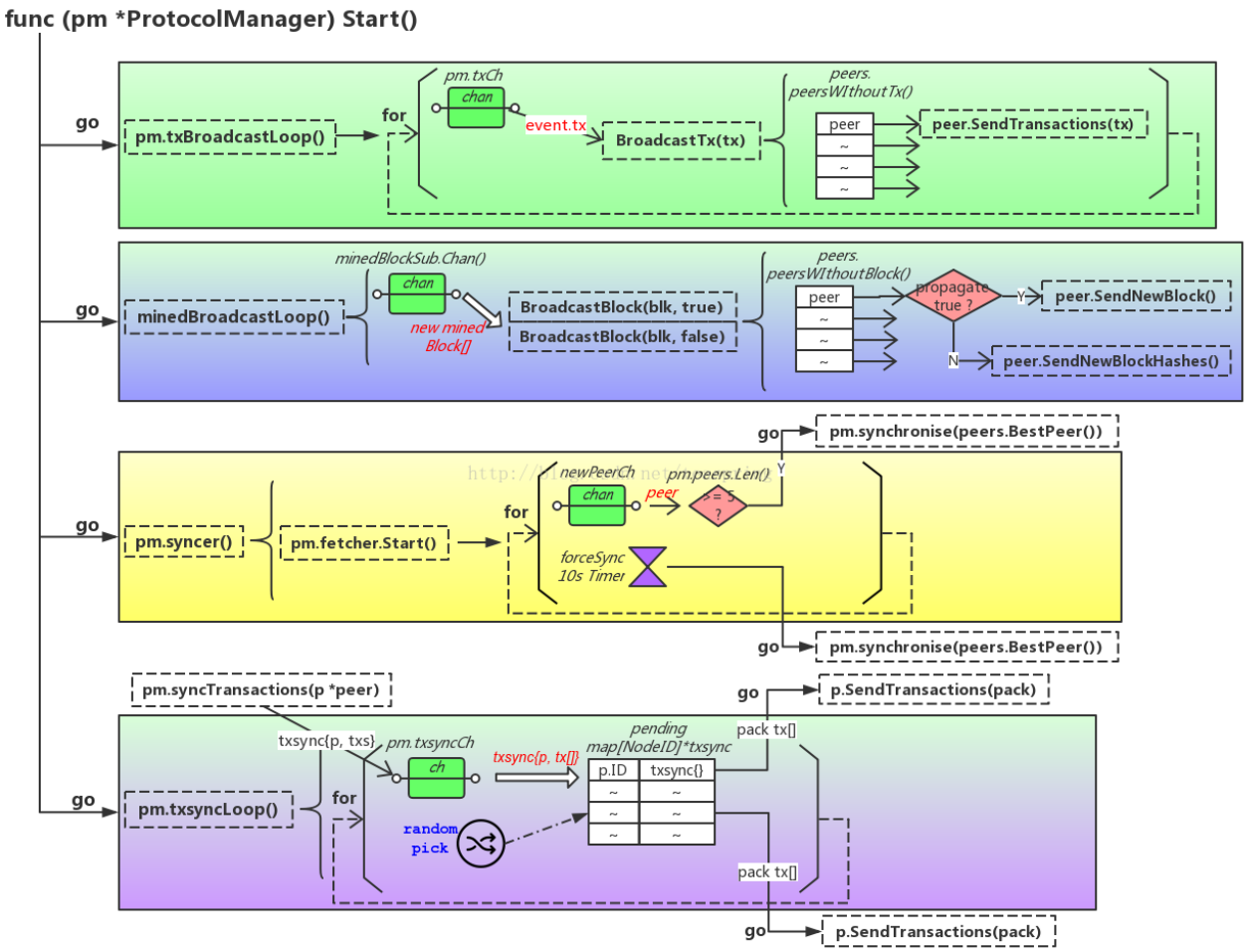



ProtocolManager主要成员包括：

- **peertSet{}**类型成员用来缓存相邻个体列表，**peer{}**表示网络中的一个远端个体。
- 通过各种**通道(chan)**和**事件订阅(subscription)**的方式，接收和发送包括交易和区块在内的数据更新。当然在应用中，订阅也往往利用通道来实现事件通知。
- **Fetcher**类型成员累积所有其他个体发送来的有关新数据的宣布消息，并在自身对照后，安排相应的获取请求。
- **Downloader**类型成员负责所有向相邻个体主动发起的同步流程。to fetch hashes and blocks from remote peers

## Start()：全面启动p2p通信

由Start()启动的四个函数在业务逻辑上各有侧重，下图是关于它们所在流程的简单示意图：




以上这四段相对独立的业务流程的逻辑分别是：

- **广播新出现的交易对象。**txBroadcastLoop()会在txCh通道的收端持续等待，一旦接收到有关新交易的事件，会立即调用BroadcastTx()函数广播给那些尚无该交易对象的相邻个体。
- **广播新挖掘出的区块。**minedBroadcastLoop()持续等待本个体的新挖掘出区块事件，然后立即广播给需要的相邻个体。当不再订阅新挖掘区块事件时，这个函数才会结束等待并返回。很有意思的是,在收到新挖掘出区块事件后，minedBroadcastLoop()会**连续调用两次BroadcastBlock()**，两次调用仅仅一个bool型参数@propagate不一样，当该参数为true时，会将整个新区块依次发给相邻区块中的一小部分；而当其为false时，仅仅将新区块的Hash值和Number发送给所有相邻列表。
- **定时与相邻个体进行区块全链的强制同步。**syncer()首先启动fetcher成员，然后进入一个无限循环，每次循环中都会向相邻peer列表中**“最优”的那个peer作一次区块全链同步。**发起上述同步的理由分两种：如果有新登记(加入)的相邻个体，则在整个peer列表数目大于5时，发起之；如果没有新peer到达，则以10s为间隔定时的发起之。这里所谓"最优"指的是peer中所维护区块链的TotalDifficulty(td)最高，由于Td是全链中从创世块到最新头块的Difficulty值总和，所以**Td值最高就意味着它的区块链是最新的**，跟这样的peer作区块全链同步，显然改动量是最小的，此即"最优"。
- **将新出现的交易对象均匀的同步给相邻个体。**txsyncLoop()主体也是一个无限循环，它的逻辑稍微复杂一些：首先有一个数据类型txsync{p, txs},包含peer和tx列表；通道txsyncCh用来接收txsync{}对象；txsyncLoop()每次循环时，如果从通道txsyncCh中收到新数据，则将它存入一个本地map[]结构，k为peer.ID，v为txsync{}，并将这组tx对象发送给这个peer；每次向peer发送tx对象的上限数目100*1024，如果txsync{}对象中有剩余tx，则该txsync{}对象继续存入map[]并更新tx数目；如果本次循环没有新到达txsync{},则从map[]结构中**随机**找出一个txsync对象，将其中的tx组发送给相应的peer，重复以上循环。

## handle()：交给其他peer的回调函数

对于peer间通信而言，除了己方需要主动向对方peer发起通信(比如Start()中启动的四个独立流程)之外，还需要一种**由对方peer主动调用的数据传输**，这种传输不仅仅是由对方peer发给己方，更多的用法是对方peer主动调用一个函数让己方发给它们某些特定数据。这种通信方式，在代码实现上适合用回调(callback)来实现。

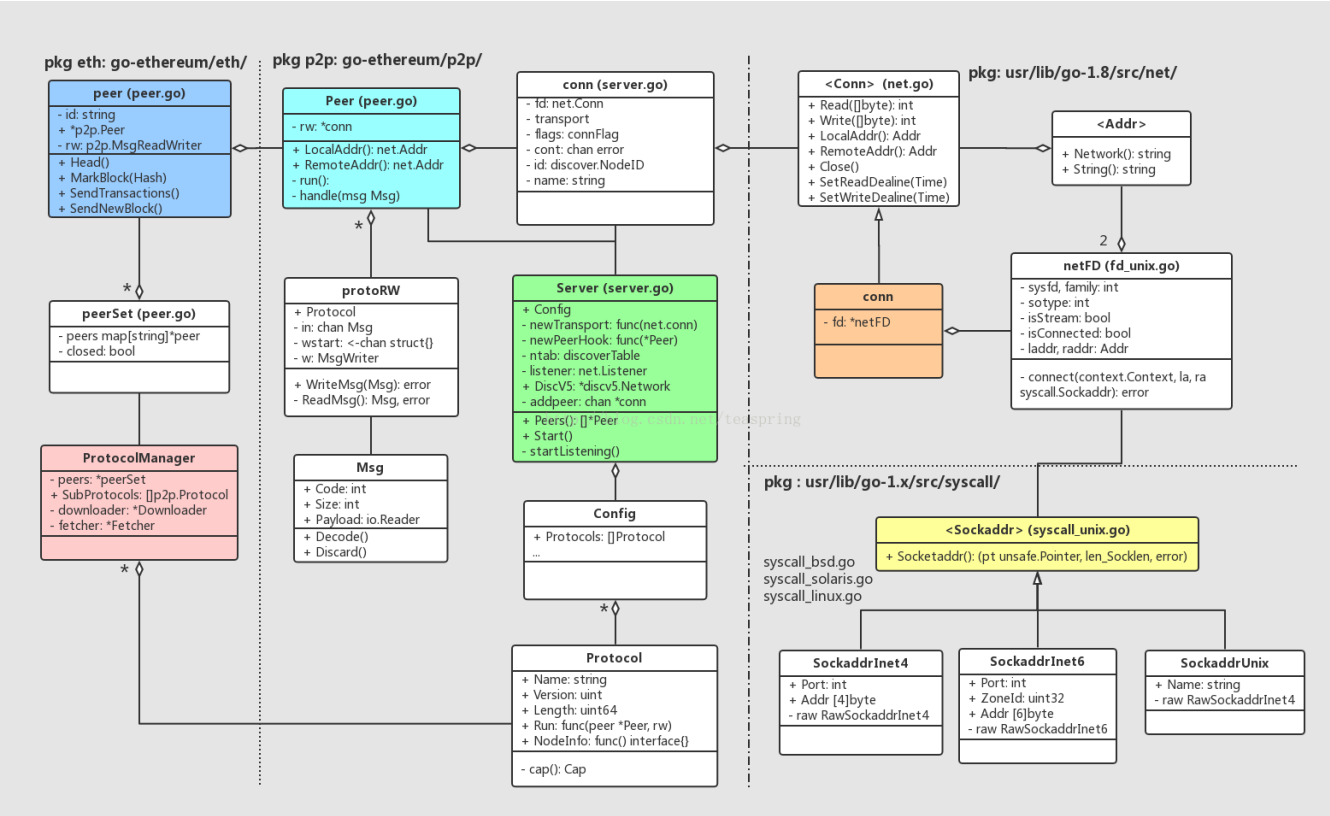
ProtocolManager.handle()就是这样一个函数，它会在ProtocolManager对象创建时，以回调函数的方式“埋入”每个p2p.Protocol对象中(实现了Protocol.Run()方法)。之后每当有新peer要与己方建立通信时，如果对方能够支持该Protocol，那么双方就可以顺利的建立并开始通信。以下是handle()的基本代码：

```
// /eth/handler.go
func (pm *ProtocolManager) handle(p *peer) error {
    td, head, genesis := pm.blockchain.Status()
    p.Handshake(pm.networkId, td, head, genesis)

    if rw, ok := p.rw.(*meteredMsgReadWriter); ok {
        rm.Init(p.version)
    }

    pm.peers.Register(p)
    defer pm.removePeer(p.id)

    pm.downloader.RegisterPeer(p.id, p.version, p)

    pm.syncTransactions(p)
    ...
    for {
        if err := pm.handleMsg(p); err != nil {
            return err
        }
    }
}
```

handle()函数针对一个新peer做了如下几件事：

1. 握手，与对方peer沟通己方的区块链状态
2. 初始化一个读写通道，用以跟对方peer相互数据传输。
3. 注册对方peer，存入己方peer列表；只有handle()函数退出时，才会将这个peer移除出列表。
4. Downloader成员注册这个新peer；Downloader会自己维护一个相邻peer列表。
5. 调用syncTransactions()，用当前txpool中新累计的tx对象组装成一个txsync{}对象，推送到内部通道txsyncCh。还记得Start()启动的四个函数么? 其中第四项txsyncLoop()中用以等待txsync{}数据的通道txsyncCh，正是在这里被推入txsync{}的。
6. **在无限循环中启动handleMsg()**，当对方peer发出任何msg时，handleMsg()可以捕捉相应类型的消息并在己方进行处理。

## p2p通信协议族的结构设计




## 小结:

1. eth.ProtocolManager中，会对每一个远端peer发起**主动传输数据**的操作，这组操作按照数据类型区分，可分为**交易和区块**；而若以发送数据方式来区分，亦可分为**广播单项数据**，**和同步一组同类型数据**。这样两两配对，即可形成4组主动传输数据的操作。
2. ProtocolManager通过**在p2p.Protocol{}对象中埋入回调函数**，可以对远端peer的任何事件及状态更新作出响应。这些Protocol对象，会由p2p.Server传递给每一个新连接上的远端peer。
3. 以太坊目前实现的**p2p通信协议族**的结构类型中，**按照功能和作用，可分为三层**：顶层pkg eth中的类型直接服务于当前以太坊系统（Ethereum，ProtocolManager等模块），中间层pkg p2p是泛化结构类型，底层包括golang语言包自带的pkg net， syscall等，封装了网络层和传输层协议的系统实现。