004 以太坊源码解析 - RPC 通信

上一节提到的 node.Start 会启动 RPC服务。

```
// go-ethereum/node/node.go
// Start create a live P2P node and starts running it.
func (n *Node) Start() error {
...
    if err := n.startRPC(services); err != nil {
        for _, service := range services {
            service.Stop()
        }
        running.Stop()
        return err
    }
...
}
```

startRPC

首先会调用每个Service的APIs()函数，把所有RPC API收集到一个数组中：

```
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()...)
    }
}
```

apis 主要包括两部分：

- node.apis()

```
func (n *Node) apis() []rpc.API {
    return []rpc.API{
        {
            Namespace: "admin",
            Version:   "1.0",
            Service:   NewPrivateAdminAPI(n),
		}, {
			Namespace: "admin",
			Version:   "1.0",
			Service:   NewPublicAdminAPI(n),
			Public:    true,
		}, {
			Namespace: "debug",
			Version:   "1.0",
			Service:   debug.Handler,
		}, {
			Namespace: "debug",
			Version:   "1.0",
			Service:   NewPublicDebugAPI(n),
			Public:    true,
		}, {
			Namespace: "web3",
			Version:   "1.0",
			Service:   NewPublicWeb3API(n),
			Public:    true,
		},
	}
}
```

- service.APIs() Ethereum 服务实现的APIs()接口。

```
// 代码位于 go-ethereum/eth/backend.go

// APIs return the collection of RPC services the ethereum package offers.
// NOTE, some of these services probably need to be moved to somewhere else.
func (s *Ethereum) APIs() []rpc.API {
	apis := ethapi.GetAPIs(s.APIBackend)

	// Append any APIs exposed explicitly by the consensus engine
	apis = append(apis, s.engine.APIs(s.BlockChain())...)

	// Append all the local APIs and return
	return append(apis, []rpc.API{
		{
			Namespace: "eth",
			Version:   "1.0",
			Service:   NewPublicEthereumAPI(s),
			Public:    true,
		}, {
			Namespace: "eth",
			Version:   "1.0",
			Service:   NewPublicMinerAPI(s),
			Public:    true,
        }, {
            Namespace: "eth",
            Version:   "1.0",
            Service:   downloader.NewPublicDownloaderAPI(s.protocolManager.download
er, s.eventMux),
            Public:    true,
        }, {
            Namespace: "miner",
            Version:   "1.0",
            Service:   NewPrivateMinerAPI(s),
            Public:    false,
        }, {
            Namespace: "eth",
            Version:   "1.0",
            Service:   filters.NewPublicFilterAPI(s.APIBackend, false),
            Public:    true,
        }, {
            Namespace: "admin",
            Version:   "1.0",
            Service:   NewPrivateAdminAPI(s),
        }, {
            Namespace: "debug",
            Version:   "1.0",
            Service:   NewPublicDebugAPI(s),
            Public:    true,
        }, {
            Namespace: "debug",
            Version:   "1.0",
            Service:   NewPrivateDebugAPI(s.chainConfig, s),
        }, {
            Namespace: "net",
            Version:   "1.0",
            Service:   s.netRPCService,
            Public:    true,
        },
    }...)
}
```

获得所有RPC API的集合后，就开始启动RPC server

startRPC方法收集Node里面所有service 的 APIs。然后分别启动了 `InProc` `IPC` `Http` `Ws` 。

- InProc：直接生成RPCService实例，挂在Node上面可以直接调用
- IPC：进程间调用，通过Unix Domain Socket（datadir/geth.ipc）
- HTTP：通过HTTP协议调用

- WS：通过WebSocket调用

RPC server

RPC server创建流程大概是：写符合规范的RPC server接口-->new server（实现serverHttp()方法）-->RPC 注册server-->RPC HandleHTTP()-->net.Listen 端口和地址-->http.server(listen)

这里以HTTP为例进行分析：

```go
// 代码 node/node.go

// startHTTP initializes and starts the HTTP RPC endpoint.
func (n *Node) startHTTP(endpoint string, apis []rpc.API, modules []string, cors []
string, vhosts []string) error {
    // Short circuit if the HTTP endpoint isn't being exposed
    if endpoint == "" {
        return nil
    }
    listener, handler, err := rpc.StartHTTPEndpoint(endpoint, apis, modules, cors,
vhosts)
    if err != nil {
        return err
    }
    n.log.Info("HTTP endpoint opened", "url", fmt.Sprintf("http://%s", endpoint), "
cors", strings.Join(cors, ","), "vhosts", strings.Join(vhosts, ","))
    // All listeners booted successfully
    n.httpEndpoint = endpoint
    n.httpListener = listener
    n.httpHandler = handler

    return nil
}
```

- httpEndpoint：这是一个字符串，表示IP和端口号，默认是localhost:8545
- httpListener：这是一个接口，调用net.Listen()时返回，包含了Accept()/Close()/Addr()这3个函数，可以用来接受和关闭连接
- httpHandler：是一个 RPC server，结构如下：

```go
// 代码 rpc/types.go
// Server represents a RPC server
type Server struct {
    services serviceRegistry
    run      int32
    codecsMu sync.Mutex
    codecs   *set.Set
}
```

StartHTTPEndpoint

```
// 代码 rpc/endpoints.go

// StartHTTPEndpoint starts the HTTP RPC endpoint, configured with cors/vhosts/modu
les
func StartHTTPEndpoint(endpoint string, apis []API, modules []string, cors []string
, vhosts []string) (net.Listener, *Server, error) {
    // Generate the whitelist based on the allowed modules
    whitelist := make(map[string]bool)
    for _, module := range modules {
        whitelist[module] = true
    }
    // Register all the APIs exposed by the services
    handler := NewServer()
    for _, api := range apis {
        if whitelist[api.Namespace] || (len(whitelist) == 0 && api.Public) {
            if err := handler.RegisterName(api.Namespace, api.Service); err != nil
{
                return nil, nil, err
            }
            log.Debug("HTTP registered", "namespace", 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 nil, nil, err
    }
    go NewHTTPServer(cors, vhosts, handler).Serve(listener)
    return listener, handler, err
}
```

- 过滤白名单的接口，白名单在defaultConfig里面配置。
- 将api的namespace和service传入RegisterName()

```go
// 代码 rpc/server.go

func (s *Server) RegisterName(name string, rcvr interface{}) error {
    if s.services == nil {
        s.services = make(serviceRegistry)
    }

    svc := new(service)
    svc.typ = reflect.TypeOf(rcvr)
    rcvrVal := reflect.ValueOf(rcvr)

    if name == "" {
        return fmt.Errorf("no service name for type %s", svc.typ.String())
    }
    if !isExported(reflect.Indirect(rcvrVal).Type().Name()) {
        return fmt.Errorf("%s is not exported", reflect.Indirect(rcvrVal).Type().Na
me())
    }

    methods, subscriptions := suitableCallbacks(rcvrVal, svc.typ)

    // already a previous service register under given sname, merge methods/subscri
ptions
    if regsvc, present := s.services[name]; present {
        if len(methods) == 0 && len(subscriptions) == 0 {
            return fmt.Errorf("Service %T doesn't have any suitable methods/subscri
ptions to expose", rcvr)
        }
        for _, m := range methods {
            regsvc.callbacks[formatName(m.method.Name)] = m
        }
        for _, s := range subscriptions {
            regsvc.subscriptions[formatName(s.method.Name)] = s
        }
        return nil
    }

    svc.name = name
    svc.callbacks, svc.subscriptions = methods, subscriptions

    if len(svc.callbacks) == 0 && len(svc.subscriptions) == 0 {
        return fmt.Errorf("Service %T doesn't have any suitable methods/subscriptio
ns to expose", rcvr)
    }

    s.services[svc.name] = svc
    return nil
}
```

用go的反射方法获取到service的类型和所持有的值。
suitableCallbacks方法获取Service所有的的方法和符合订阅标准的方法。
将Service的所有方法放入map s.services, service的name作为map key。

- 侦听TCP端口，获得listener接口实例。然后创建了一个http.Server实例，并启动一个goroutine调用它的Serve()方法。

```
// 代码 rpc/http.go

func NewHTTPServer(cors []string, vhosts []string, srv *Server) *http.Server {
    // Wrap the CORS-handler within a host-handler
    handler := newCorsHandler(srv, cors)
    handler = newVHostHandler(vhosts, handler)
    return &http.Server{Handler: handler}
}
```

Handler参数，用到了装饰者模式，其实最终实现还是在rpc.Server中。Handler是一个接口，需要实现它的ServerHTTP()函数来处理网络数据，代码如下：

```
// 代码位于 rpc/http.go

// ServeHTTP serves JSON-RPC requests over HTTP.
func (srv *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // Permit dumb empty requests for remote health-checks (AWS)
    if r.Method == http.MethodGet && r.ContentLength == 0 && r.URL.RawQuery == "" {
        return
    }
    if code, err := validateRequest(r); err != nil {
        http.Error(w, err.Error(), code)
        return
    }
    // All checks passed, create a codec that reads direct from the request body
    // untilEOF and writes the response to w and order the server to process a
    // single request.
    ctx := context.Background()
    ctx = context.WithValue(ctx, "remote", r.RemoteAddr)
    ctx = context.WithValue(ctx, "scheme", r.Proto)
    ctx = context.WithValue(ctx, "local", r.Host)

    body := io.LimitReader(r.Body, maxRequestContentLength)
    codec := NewJSONCodec(&httpReadWriteNopCloser{body, w})
    defer codec.Close()
    w.Header().Set("content-type", contentType)
    srv.ServeSingleRequest(ctx, codec, OptionMethodInvocation)
}
```

可以看到，首先创建一个Reader用于读取原始数据，然后创建一个JSON的编解码器，最后调用ServeSingleRequest()函数，

```
func (s *Server) ServeSingleRequest(ctx context.Context, codec ServerCodec, options
 CodecOption) {
    s.serveRequest(ctx, codec, true, options)
}

func (s *Server) serveRequest(ctx context.Context, codec ServerCodec, singleShot bo
ol, options CodecOption) error {
    var pend sync.WaitGroup

    defer func() {
        if err := recover(); err != nil {
            const size = 64 << 10
            buf := make([]byte, size)
            buf = buf[:runtime.Stack(buf, false)]
            log.Error(string(buf))
        }
        s.codecsMu.Lock()
        s.codecs.Remove(codec)
        s.codecsMu.Unlock()
    }()

    //  ctx, cancel := context.WithCancel(context.Background())
    ctx, cancel := context.WithCancel(ctx)
    defer cancel()

    // if the codec supports notification include a notifier that callbacks can use
    // to send notification to clients. It is thight to the codec/connection. If th
e
    // connection is closed the notifier will stop and cancels all active subscript
ions.
    if options&OptionSubscriptions == OptionSubscriptions {
        ctx = context.WithValue(ctx, notifierKey{}, newNotifier(codec))
    }
    s.codecsMu.Lock()
    if atomic.LoadInt32(&s.run) != 1 { // server stopped
        s.codecsMu.Unlock()
        return &shutdownError{}
    }
    s.codecs.Add(codec)
    s.codecsMu.Unlock()
```

```go
// test if the server is ordered to stop
for atomic.LoadInt32(&s.run) == 1 {
    reqs, batch, err := s.readRequest(codec)
    if err != nil {
        // If a parsing error occurred, send an error
        if err.Error() != "EOF" {
            log.Debug(fmt.Sprintf("read error %v\n", err))
            codec.Write(codec.CreateErrorResponse(nil, err))
        }
        // Error or end of stream, wait for requests and tear down
        pend.Wait()
        return nil
    }

    // check if server is ordered to shutdown and return an error
    // telling the client that his request failed.
    if atomic.LoadInt32(&s.run) != 1 {
        err = &shutdownError{}
        if batch {
            resps := make([]interface{}, len(reqs))
            for i, r := range reqs {
                resps[i] = codec.CreateErrorResponse(&r.id, err)
            }
            codec.Write(resps)
        } else {
            codec.Write(codec.CreateErrorResponse(&reqs[0].id, err))
        }
        return nil
    }
    // If a single shot request is executing, run and return immediately
    if singleShot {
        if batch {
            s.execBatch(ctx, codec, reqs)
        } else {
            s.exec(ctx, codec, reqs[0])
        }
        return nil
    }
    // For multi-shot connections, start a goroutine to serve and loop back
    pend.Add(1)

    go func(reqs []*serverRequest, batch bool) {
        defer pend.Done()
        if batch {
            s.execBatch(ctx, codec, reqs)
        } else {
            s.exec(ctx, codec, reqs[0])
            }
        }(reqs, batch)
    }
    return nil
}
```

可以看到，就是一个循环，每次调用readRequest()解析请求数据，然后调用exec()或者execBatch()执行API调用。