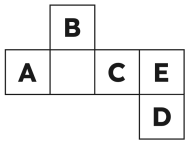
Understanding Ethereum:
Go-ethereum Source Code Analysis

理解以太坊: Go-Ethereum 源码剖析

Siyuan Han @ABCDE Capital

May 2023



Special thanks for the outstanding ABCDE Team.

目录

第 1 章

前言 (Preface)

1.1 写作背景

1.1.1 时代的弄潮儿: Blockchain

Blockchain 最早作为支撑 Bitcoin 结算的分布式账本技术，由中本聪在**比特币白皮书**提出，至今已经过了十几年。在这十几年中，随着加密货币价格的飞涨，社区的参与人数不断的增加，大量的来自不同背景专业人士围绕 Blockchain 这一话题源源不断的注入新的想法。随着新的思想持续的涌入，区块链技术的边界不断的拓展，逐渐成为了融合了数据库，分布式系统，密码学，点对点网络，编译原理，静态软件分析，众包，经济学，货币金融学，社会学等多个学科知识的一个全新技术领域。在源源不断的新技术和新思想的催化下，Blockchain 也逐渐从小众的去中心化技术社区逐渐走向了主流社会的舞台，至今仍是当下**最热度最高**，**技术迭代最快**，**最能引起社会讨论**的技术话题之一。在 Blockchain 原生的 decentralized 的思想的影响下，市面上绝大多数的 Blockchain 系统都是以开源的形式持续在 Github 上开发和维护。这就为我们提供了一种的极好的学习 Blockchain 技术的方式: 结合文档，基于源代码，理解和学习系统的设计思想和实现原理。

1.1.2 为什么要研究以太坊 & Go-Ethereum 的原理

从基础建设的角度看，随着网络基础建设的不断完善，网络带宽增加和通信延迟下降的趋势将会持续。伴随着存储技术和分布式算法的不断发展，未来软件系统的运

行效率将会持续的提高，并且会逐渐逼近硬件设备的性能极限。这些基础建设的发展为构建大规模去中心化应用带来更大的确定性。在未来的五到十年内，云端服务/去中心化系统的性能以及覆盖场景一定还会有很大的提升。未来的技术世界大概率会是两极分化的。一极是以大型科技公司（i.e, Google，MS，Oracle，Snowflake，and Alibaba）为代表的中心化服务商。另一极就是以 Blockchain 技术作为核心的去中心化的世界。在这个世界中，Ethereum 及其生态系统是当之无愧的领头羊。Ethereum 作为通用型 Public Chain 中的翘楚构建了稳定强大的生态系统。Ethereum 及其生态吸引到了一大批世界上最优秀的工程师和研究人员不断的将新思想，新理念，新技术引入到 Ethereum 及其生态中，并且持续的引领整个 Blockchain 生态系统发展。从计算机技术的发展史上看，优秀的平台化的开源项目，例如 Linux，Android，都拥有持久强大的生命力。由于这类项目的代码经过时间的验证，拥有良好的生态循环，最终会有源源不断的开发人员基于这些项目的代码库来开发全新的应用。Go-Ethereum 作为 Ethereum 的优秀稳健的开源实现，目前已经被广泛的订制，被应用在了多种私有/联盟/Layer-2 的场景中 (e.g., Quorum, Binance Smart Chain, Scroll, Arbitrum , Optimism)。不管是哪种场景，Ethereum 的相关代码总是位于系统的核心中的核心位置。因此，作为开发人员/研究人员理解 Ethereum 的设计和实现是至关重要的。

1.1.3　本书的写作目的

一个热门的技术是否热门的标志之一是: 是否有不同视角的作者，在不同的技术发展阶段记录下来的文档资料。目前，对于学习者，不管是探究以加密货币导向（Crypto-based）的 Bitcoin, 还是了解致力于实现通用 Web3 框架（General-Purpose）的 Ethereum，社区中有丰厚的从基础概念的角度的出发的技术文档来讲述它们的基础概念和设计的思想。比如，技术社区有非常多的资料来讲述什么是梅克尔树 (Merkle Hash Tree)，什么是梅克尔帕特里夏树 (Merkle Patricia Trie)，什么是有向无环图 (Directed acyclic Graph); BFT (Byzantine Fault Tolerance) 和 PoW (Proof-Of-Work) 共识算法算法的区别; 以及介绍 Blockchain 系统为什么可以抵抗双花攻击 (Double-Spending)，或者为什么 Ethereum 会遇到 DAO Attack (Decentralized autonomous organization) 等具体问题。

但是，现有的资料往往对工程实现的细节介绍的不够清晰。对于研究人员和开发人员来说，只了解关键组件的实现细节，或者高度抽象的系统工作流，并不代表着搞清楚 Blockchain 的**工作原理**。反而很容易在一些关键细节上一头雾水，似懂非懂。

比如，当我们谈到 Ethereum 中 Transaction 的生命周期时，翻阅文档时经常会看到类似的说法，“Miner 节点批量地从自己维护的 Transaction pool 中选择一些 Transaction 并打包成一个新的 Block 中”。那么究竟 Miner 是怎么从网络中获取到 Transaction？又是基于什么样的策略从 Transaction pool 中选取**多少** Transaction？最终又按照什么样的 Order 把 Transaction 打包进区块中的呢？打包成功的 Block 是怎么交互/传播给其他节点呢？我搜索了大量的文档，发现鲜有文章详细的解释了上述的问题。因此，社区需要一些文章从整体的系统工作流的角度出发，以**细粒度**的视角对区块链系统中的具体的实现细节进行解析。与数据库系统 (Database Management System) 相似，Blockchain 系统同样是一个包含网络层，业务逻辑层，任务解析层，存储层的复杂数据管理系统。对它研究同样需要从系统的实现细节出发，从宏观到微观的了解每个执行逻辑的工作流，才能彻底理解和掌握这门技术的秘密。

本系列文章将会从 Ethereum 执行层中具体业务的工作的视角出发，在源码的层面，细粒度地解析以太坊系统中各个模块的实现的细节，以及背后的蕴含的技术和设计思想。Go-ethereum 是以太坊协议的 Go 语言实现版本，目前由以太坊基金会维护。目前除了 Go-ethereum 之外，Ethereum 还有 C++, Python，Java, Rust 等基于其他语言实现的版本。由于 Go-ethereum 的代码库在持续的更新，源码分析的文档很难持续追踪最新的代码库。因此，本系列文档目前选择基于 Go-ethereum version 1.10.*(post-merge) 版本首先进行编写。相比于其他的由社区维护的版本，Go-ethereum 的用户数量最多，开发人员最多，版本更新最频繁，issues 的发现和处理都较快。其他语言的 Ethereum 实现版本因为用户与开发人员的数量相对较少，更新频率相对较低，隐藏问题出现的可能性更高。同时 Go 语言语法简单，容易阅读。对于没有 Go 语言开发经验的读者，仍然可以快速的理解代码逻辑。因此我们选择从 Go-ethereum 代码库作为我们的主要学习资料。

在合并 (Merge) 之后，以太坊信标链和原有的主链进行了合并。原有的主链节点 (Go-ethereum 节点) 进行了功能缩减，放弃了共识相关的功能，仅作为执行层继续在以太坊的生态中发挥至关重要的作用。例如，交易的执行，状态的维护，数据的存储等基本功能还是由执行层进行维护。因此，作为开发和研究人员，了解 Go-ethereum 代码库仍然是十分有意义的。

1.1.4 我们为什么要阅读区块链系统的源代码？

1. 关于以太坊细节实现的文档资料相对较少。由于 Ethereum 进行了多次设计上的更新，一些源代码解析的文章中采用的代码已经经历了多次的修改，导致文章

丧失了部分的时效性。同时，不少文章在分析细节的时候，浅尝辄止，对一些关键问题没有解析到位。比如，很多的科普文章都提到，在打包新的 *Block* 的时候，*miner* 负责把 *a batch of transactions* 从 *transaction pool* 中打包到新的 *block* 中。那么我们希望读者思考如下的几个问题：

- Miner 是从什么方式获取到待打包的 Transactions?
- Miner 是基于什么样策略从 Transaction Pool 中选择 Transaction 呢？
- 被选择的 Transactions 又是以怎样的顺序 (Order) 被打包到区块中的呢？
- 在执行 Transaction 的 EVM 是怎么计算 gas used，从而限定 Block 中 Transaction 的数量?
- 剩余的 gas 又是怎么返还给 Transaction Proposer 的呢？
- EVM 是怎么解释 Contract Code 的 Message Call 并执行的呢？
- 在执行 Transaction 时，是什么模块，怎样去修改 Contract 中持久化变量？
- Smart Contract 中的持久化变量是以什么样的形式存储？又是存储在什么地方？
- 当新的 Block 更新到 Blockchain 中时，World State 又是在什么时机，以什么方式更新的呢？
- 哪些数据常驻内存，哪些数据需要保存在 Disk 中呢？

2. 目前的 Blockchain 系统并没有像数据库系统 (DBMS) 那样统一的形成系统性的方法论。在 Ethereum 中每个不同的模块中都集成了大量的细节。从源码的角度出发可以了解到很多容易被忽视的细节。简单的说，一个完整的区块链系统至少包含以下的模块:

   - 密码学模块: 加解密，签名，安全 hash，Mining
   - 网络模块: P2P 节点通信
   - 分布式共识模块: PoW, BFT，PoA
   - 智能合约解释器模块: Solidity 编译语言，EVM 解释器
   - 数据存储模块: 状态数据库，Caching，数据存储，Index，LevelDB
   - Log 日志模块
   - etc.

而在具体实现中，由于设计理念，以及 go 语言的特性 (没有继承派生关系)，Go-ethereum 中的模块之间相互调用关系相对复杂。因此，只有通过阅读源码的方式才

能更好理解不同模块之间的调用关系，以及业务流程中的关键细节。

1.2 如何衡量对一个系统的理解程度？

阅读源代码是一种漫长的修行。为了方便自检修行的结果，我们将对一个系统的理解程度划分为下面四个等级。

- Level 4: 掌握（Mastering）
  - 在完全理解的基础上，可以设计并编写一个全新的系统
  - 根据实际需求，重写系统模块
  - 可以使用另一种编程语言重新复现本系统
- Level 3: 完全理解（Complete Understanding）
  - 在理解的基础上，完全掌握系统各个模块实现的细节
  - 能快速的从系统功能模块定位到其对应的代码库的位置
  - 可以将系统定制化到不同的应用场景
  - 能对系统中的各个模块做出优化
- Level 2: 理解（Understanding）
  - 熟练使用系统的常用 API
  - 了解系统各个模块的调用关系
  - 了解部分核心模块的设计细节
  - 能对系统的部分模块进行简单修改/重构
- Level 1: 了解（Brief understanding）
  - 了解系统设计的主要目标
  - 了解系统的应用场景
  - 了解系统的主要功能
  - 可以使用系统的部分的 API

我们希望读者在阅读完本系列之后，对以太坊的理解能够达到 Level 2 - Level 3 的水平。

第 2 章

万物的起点: Geth Start

本章概要:

1. go-ethereum 代码库的主要目录结构。
2. geth 客户端/节点是如何启动的。
3. 如何修改/添加 geth 对外的 APIs。

2.1 什么是 geth？

geth 是以太坊基金会基于 Go 语言开发以太坊的官方客户端，它实现了 Ethereum 协议 (黄皮书) 中所有需要的实现的功能模块。我们可以通过启动 geth 来运行一个 Ethereum 的节点。在以太坊 Merge 之后，geth 作为节点的执行层继续在以太坊生态中发挥重要的作用。go-ethereum 是包含了 geth 客户端代码和以及编译 geth 所需要的其他代码在内的一个完整的代码库。在本系列中我们会通过深入 go-ethereum 代码库，从 High-level 的 API 接口出发，沿着 Ethereum 主 Workflow，逐一的理解 Ethereum 具体实现的细节。

为了方便区分，在接下来的文章中，我们用 geth 来表示 go-ethereum 客户端程序，用 GETH 来表示 go-ethereum 的代码库。

总结的来说:

1. 基于 go-ethereum 代码库中的代码，我们可以编译出 geth 客户端程序。
2. 通过运行 geth 客户端程序我们可以启动一个 Ethereum 的节点。

2.1.1 go-ethereum Codebase 结构

为了更好的从整体工作流的角度来理解 Ethereum，根据主要的业务功能，我们可以把 `go-ethereum` 划分成如下几个模块。

- Geth Client 模块
- Core 数据结构模块
- State Management 模块
  - StateDB 模块
  - Trie 数据结构模块
  - State Optimization (Pruning)
- Mining 模块
- EVM 模块
- P2P 网络模块
  - 节点数据同步
    - 交易数据
    - 区块数据
    - 区块链数据
- Storage 模块
  - 抽象数据库层
  - LevelDB 调用
- ...

目前，go-ethereum 代码库中的主要目录结构如下所示:

```
cmd/  以太坊基金会官方开发的一些  Command-line  程序。该目录下的每个子目录都是一个单独运行的 CLI 程序。
   |— clef/ 以太坊官方推出的账户管理程序.
   |— geth/ 以太坊官方的节点客户端。
core/   以太坊核心模块，包括核心数据结构，statedb，EVM 等核心数据结构以及算法实现
   |— rawdb/ db 相关函数的高层封装(在 ethdb 和更底层的 leveldb 之上的封装)
      ├—accessors_state.go 从 Disk Level 读取/写入与 State 相关的数据结构。
   |— state/
```

├── statedb.go　StateDB 是管理以太坊 World State 最核心的代码，用于管理链上所有的 State 相关操作。

├── state_object.go state_object 是以太坊账户(包括 EOA & Contract)在 StateDB 具体的实现。

|── txpool　　Transaction Pool 相关的代码。

|── txpool.go　Transaction Pool 的具体实现。

|── types/　以太坊中最核心的数据结构

|── block.go　以太坊 Block 的的数据结构定义与相关函数实现

|── bloom9.go　以太坊使用的一个 Bloom Filter 的实现

|── transaction.go 以太坊 Transaction 的数据结构定义与相关函数实现。

|── transaction_signing.go 用于对 Transaction 进行签名的函数的实现。

|── receipt.go　以太坊交易收据的实现，用于记录以太坊 Transaction 执行的结果

|── vm/　　以太坊的核心中核心 EVM 相关的一些的数据结构的定义。

|── evm.go　　EVM 数据结构和方法的定义

|── instructions.go　EVM 指令的具体的定义，核心中的核心中的核心文件。

|── logger.go　用于追踪 EVM 执行交易过程的日志接口的定义。具体的实现在eth/tracers/logger/logger.go 文件中。

|── opcode.go　EVM 指令和数值的对应关系。

|── genesis.go　　创世区块相关的函数。每个 geth 客户端/以太坊节点初始化的都需要调用这个模块。

|── state_processor.go EVM 执行交易的核心代码模块。

console/

|── bridge.go

|── console.go　Geth Web3 控制台的入口

eth/　　Ethereum 节点/后端/客户端具体功能定义和实现。例如节点的启动关闭，P2P 网络中交易和区块的同步。

ethdb/　Ethereum 本地存储的相关实现，包括 leveldb 的调用

|── leveldb/　Go-Ethereum使用的与 Bitcoin Core version一样的Leveldb作为本机存储用的数据库

internal/ 一些内部使用的工具库的集合，比如在测试用例中模拟 cmd 的工具。在构建 Ethereum 生态相关的工具时值得注意这个文件夹。

miner/

|── miner.go　矿工模块的实现。

|── worker.go　Block generation 的实现，包括打包 transaction，计算合法的 Block

p2p/　　Ethereum 的P2P模块

|── params　　Ethereum 的一些参数的配置，例如: bootnode 的 enode 地址

|── bootnodes.go　bootnode 的 enode 地址 like: aws 的一些节点，azure 的一些节点

，Ethereum Foundation 的节点和 Rinkeby 测试网的节点
rlp/ RLP的 Encode与 Decode的相关
rpc/ Ethereum RPC客户端的实现
les/ Ethereum light client 轻节点的实现
trie/ Ethereum 中至关重要的数据结构 Merkle Patrica Trie(MPT) 的实现
|— committer.go Trie 向 Memory Database 提交数据的工具函数。
|— database.go Memory Database，是 Trie 数据和 Disk Database 提交的中间层。同时还实现了 Trie 剪枝的功能。**非常重要**
|— node.go MPT中的节点的定义以及相关的函数。
|— secure_trie.go 基于 Trie 的封装的结构。与 trie 中的函数功能相同，不过secure_trie中的 key 是经过hashKey()函数hash过的，无法通过路径获得原始的 key值
|— stack_trie.go Block 中使用的 Transaction/Receipt Trie 的实现
|— trie.go MPT 具体功能的函数实现。

2.2 Geth 节点是如何启动的

2.2.1 前奏: Geth Console

当我们想要部署一个 Ethereum 节点的时候，最直接的方式就是下载官方提供的发行版的 geth 客户端程序。geth 是一个基于 CLI 的应用，启动 geth 和调用 geth 的功能性 API 需要使用对应的指令来操作。geth 提供了一个相对友好的 console 来方便用户调用各种指令。当我第一次阅读 Ethereum 的文档的时候，我曾经有过这样的疑问，为什么 geth 是由 Go 语言编写的，但是在官方文档中的 Web3 的 API 却是基于 Javascript 的调用？

这是因为 geth 内置了一个 Javascript 的解释器: *Goja* (interpreter)，来作为用户与 geth 交互的 CLI Console。我们可以在 console/console.go 中找到它的定义。

```
// Console is a JavaScript interpreted runtime environment. It is a fully fledged
// JavaScript console attached to a running node via an external or in-process RPC
// client.
type Console struct {
 client   *rpc.Client         // RPC client to execute Ethereum requests through
 jsre     *jsre.JSRE          // JavaScript runtime environment running the interpreter
 prompt   string              // Input prompt prefix string
 prompter prompt.UserPrompter // Input prompter to allow interactive user feedback
 histPath string              // Absolute path to the console scrollback history
 history  []string            // Scroll history maintained by the console
```

```
 printer  io.Writer          // Output writer to serialize any display strings to
}
```

2.2.2　geth 节点的启动流程

了解 Ethereum，我们首先要了解 Ethereum 客户端 Geth 是怎么运行的。geth 程序的启动点位于 `cmd/geth/main.go/main()` 函数处，如下所示。

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

我们可以看到 `main()` 函数非常的简短，其主要功能就是启动一个解析 command line 命令的工具: `gopkg.in/urfave/cli.v1`。继续深入，我们会发现在 cli app 初始化的时候会调用 `app.Action = geth` ，来调用 `geth()` 函数。而 `geth()` 函数就是用于启动 Ethereum 节点的顶层函数，其代码如下所示。

```
func geth(ctx *cli.Context) error {
 if args := ctx.Args(); len(args) > 0 {
  return fmt.Errorf("invalid command: %q", args[0])
 }

 prepare(ctx)
 stack, backend := makeFullNode(ctx)
 defer stack.Close()

 startNode(ctx, stack, backend, false)
 stack.Wait()
 return nil
}
```

在 `geth()` 函数中，有三个比较重要的函数调用，分别是：`prepare()`，`makeFullNode()`，以及 `startNode()`。

`prepare()` 函数的实现就在当前的 `main.go` 文件中。它主要用于设置一些节点初始化需要的配置。比如，我们在节点启动时看到的这句话: *Starting Geth on Ethereum mainnet...* 就是在 `prepare()` 函数中被打印出来的。

`makeFullNode()` 函数的实现位于 `cmd/geth/config.go` 文件中。它会将 Geth 启动时的命令的上下文加载到配置中，并生成 `stack` 和 `backend` 这两个实例。其中 `stack`

是一个 Node 类型的实例，它是通过 `makeFullNode()` 函数调用 `makeConfigNode()` 函数来初始化的。`Node` 是 geth 生命周期中最顶级的实例，它负责管理节点中的 P2P Server, Http Server, Database 等业务非直接相关的高级抽象。关于 Node 类型的定义位于 `node/node.go` 文件中。

这里的 `backend` 是一个 `ethapi.Backend` 类型的接口，提供了获取以太坊执行层运行时，所需要的基本函数功能。它的定义位于 `internal/ethapi/backend.go` 中。由于这个接口中函数较多，我们选取了其中的部分关键函数方便大家理解这个接口所提供的基本功能，如下所示。

```
type Backend interface {
    SyncProgress() ethereum.SyncProgress
    SuggestGasTipCap(ctx context.Context) (*big.Int, error)
    ChainDb() ethdb.Database
    AccountManager() *accounts.Manager
    ExtRPCEnabled() bool
    RPCGasCap() uint64 // global gas cap for eth_call over rpc: DoS protection
    RPCEVMTimeout() time.Duration // global timeout for eth_call over rpc: DoS protectio
    RPCTxFeeCap() float64 // global tx fee cap for all transaction related APIs
    UnprotectedAllowed() bool // allows only for EIP155 transactions.
    SetHead(number uint64)
    HeaderByNumber(ctx context.Context, number rpc.BlockNumber) (*types.Header, error)
    HeaderByHash(ctx context.Context, hash common.Hash) (*types.Header, error)
    HeaderByNumberOrHash(ctx context.Context, blockNrOrHash rpc.BlockNumberOrHash) (*type
    CurrentHeader() *types.Header
    CurrentBlock() *types.Header
    BlockByNumber(ctx context.Context, number rpc.BlockNumber) (*types.Block, error)
    BlockByHash(ctx context.Context, hash common.Hash) (*types.Block, error)
    BlockByNumberOrHash(ctx context.Context, blockNrOrHash rpc.BlockNumberOrHash) (*types
    StateAndHeaderByNumber(ctx context.Context, number rpc.BlockNumber) (*state.StateDB,
    StateAndHeaderByNumberOrHash(ctx context.Context, blockNrOrHash rpc.BlockNumberOrHash
    PendingBlockAndReceipts() (*types.Block, types.Receipts)
    GetReceipts(ctx context.Context, hash common.Hash) (types.Receipts, error)
    GetTd(ctx context.Context, hash common.Hash) *big.Int
    GetEVM(ctx context.Context, msg *core.Message, state *state.StateDB, header *types.He
    SubscribeChainEvent(ch chan<- core.ChainEvent) event.Subscription
    SubscribeChainHeadEvent(ch chan<- core.ChainHeadEvent) event.Subscription
    SubscribeChainSideEvent(ch chan<- core.ChainSideEvent) event.Subscription
    SendTx(ctx context.Context, signedTx *types.Transaction) error
    GetTransaction(ctx context.Context, txHash common.Hash) (*types.Transaction, common.H
    GetPoolTransactions() (types.Transactions, error)
    GetPoolTransaction(txHash common.Hash) *types.Transaction
    GetPoolNonce(ctx context.Context, addr common.Address) (uint64, error)
    Stats() (pending int, queued int)
```

```
35      TxPoolContent() (map[common.Address]types.Transactions, map[common.Address]types.Tran
36      TxPoolContentFrom(addr common.Address) (types.Transactions, types.Transactions)
37      SubscribeNewTxsEvent(chan<- core.NewTxsEvent) event.Subscription
38      ChainConfig() *params.ChainConfig
39      Engine() consensus.Engine
40      GetBody(ctx context.Context, hash common.Hash, number rpc.BlockNumber) (*types.Body,
41      GetLogs(ctx context.Context, blockHash common.Hash, number uint64) ([][]*types.Log, e
42      SubscribeRemovedLogsEvent(ch chan<- core.RemovedLogsEvent) event.Subscription
43      SubscribeLogsEvent(ch chan<- []*types.Log) event.Subscription
44      SubscribePendingLogsEvent(ch chan<- []*types.Log) event.Subscription
45      BloomStatus() (uint64, uint64)
46      ServiceFilter(ctx context.Context, session *bloombits.MatcherSession)
47  }
```

我们可以发现 `ethapi.Backend` 接口主要对外提供了: 1. General Ethereum APIs, 这些 General APIs 对外提供了查询区块链节点管理对象的接口，例如 `ChainDb()` 返回当前节点的 DB 实例, `AccountManager()`; 2. Blockchain 相关的 APIs, 例如链上数据的查询 (Block & Transaction), `CurrentHeader()`, `BlockByNumber()`, `GetTransaction()`; 3. Transaction Pool 相关的 APIs, 例如发送交易到本节点的 Transaction Pool, 以及查询交易池中的 Transactions, `GetPoolTransaction`。

目前 Geth 代码库中，有两个 `ethapi.Backend` 接口的实现，分别是: 1. 位于 `eth\api_backend` 中的 `EthAPIBackend`; 2. 位于 `les\api_backend` 的 `LesApiBackend`; 顾名思义，`EthAPIBackend` 提供了针对全节点的 Backend API 服务, 而 `LesApiBackend` 提供了轻节点的 Backend API 服务。总结的来说，如果读者想定制一些新的 RPC API，可以在 `ethapi.Backend` 接口中定义函数，并给 `EthAPIBackend` 添加具体的实现。

读者可能会发现，`ethapi.Backend` 接口所提供的函数功能，主要读写本地的维护的数据结构 (i.e. Transaction Pool, Blockchain) 的为主。那么作为一个有网络连接的 Backend, 以太坊的 Backend 或者说 Node 是怎么管理以太坊执行层节点的网络连接，共识等功能模块的呢？

我们深入 `makeFullNode()` 函数可以发现，生成 `ethapi.Backend` 接口的语句 `backend, eth := utils.RegisterEthService(stack, &cfg.Eth)`, 还返回了另一个 `Ethereum` 类型的实例 `eth`。这个 `Ethereum` 类型才是以太坊节点数结构中核心中的核心，它实现了以太坊全节点所需要的所有的 Service。它负责提供更为具体的以太坊的功能性 Service, 负责与以太坊业务直接相关的抽象，比如维护 Blockchain 的更新，共识算法，从 P2P 网络中同步区块，同步 P2P 节点远端的交易并放到交易池中，等业务功能。我们会在后续详细讲解 `Ethereum` 类型具体提供的服务。

`Ethereum` 实例根据上下文的配置信息在调用 `utils.RegisterEthService()` 函数生

成。在 `utils.RegisterEthService()` 函数中，首先会根据当前的 config 来判断需要生成的 Ethereum backend 的类型，是 light node backend 还是 full node backend。我们可以在 `eth/backend/new()` 函数和 `les/client.go/new()` 中找到这两种 Ethereum backend 的实例是如何初始化的。Ethereum backend 的实例定义了一些更底层的配置，比如 chainid，链使用的共识算法的类型等。这两种后端服务的一个典型的区别是 light node backend 不能启动 Mining 服务。在 `utils.RegisterEthService()` 函数的最后，调用了 `Nodes.RegisterAPIs()` 函数，将刚刚生成的 backend 实例注册到 `stack` 实例中。

总结的说，`api_backend` 主要是用于对外提供查询，或者与后端功能性生命周期无关的函数，`Ethereum` 这类的节点层的后端，主要用于管理/控制节点后端的生命周期。

最后一个关键函数，`startNode()` 的作用是正式的启动一个以太坊执行层的节点。它通过调用 `utils.StartNode()` 函数来触发 `Node.Start()` 函数来启动 `Stack` 实例 (Node)。在 `Node.Start()` 函数中，会遍历 `Node.lifecycles` 中注册的后端实例，并启动它们。此外，在 `startNode()` 函数中，还是调用了 `unlockAccounts()` 函数，并将解锁的钱包注册到 `stack` 中，以及通过 `stack.Attach()` 函数创建了与 local Geth 交互的 RPClient 模块。

在 `geth()` 函数的最后，函数通过执行 `stack.Wait()`，使得主线程进入了阻塞状态，其他的功能模块的服务被分散到其他的子协程中进行维护。

2.2.3 Node

正如我们前面提到的，Node 类型在 geth 的生命周期性中属于顶级实例，它负责作为与外部通信的高级抽象模块的管理员，比如管理 rpc server，http server，Web Socket，以及 P2P Server 外部接口。同时，Node 中维护了节点运行所需要的后端的实例和服务 (`lifecycles []Lifecycle`)，例如我们上面提到的负责具体 Service 的 `Ethereum` 类型。

```
// Node is a container on which services can be registered.
type Node struct {
 eventmux      *event.TypeMux
 config        *Config
 accman        *accounts.Manager
 log           log.Logger
 keyDir        string            // key store directory
 keyDirTemp    bool              // If true, key directory will be removed by Stop
```

```
 dirLock       fileutil.Releaser // prevents concurrent use of instance directory
 stop          chan struct{}     // Channel to wait for termination notifications
 server        *p2p.Server       // Currently running P2P networking layer
 startStopLock sync.Mutex        // Start/Stop are protected by an additional lock
 state         int               // Tracks state of node lifecycle
 lock          sync.Mutex
 lifecycles    []Lifecycle // All registered backends, services, and auxiliary services t
 rpcAPIs       []rpc.API   // List of APIs currently provided by the node
 http          *httpServer //
 ws            *httpServer //
 httpAuth      *httpServer //
 wsAuth        *httpServer //
 ipc           *ipcServer  // Stores information about the ipc http server
 inprocHandler *rpc.Server // In-process RPC request handler to process the API requests

 databases map[*closeTrackingDB]struct{} // All open databases
}
```

2.2.3.1 Node 的关闭

在前面我们提到，整个程序的主线程因为调用了 `stack.Wait()` 而进入了阻塞状态。我们可以看到 Node 结构中声明了一个叫做 `stop` 的 channel。由于这个 Channel 一直没有被赋值，所以整个 geth 的主进程才进入了阻塞状态，持续并发的执行其他的业务协程。

```
// Wait blocks until the node is closed.
func (n *Node) Wait() {
 <-n.stop
}
```

当 `n.stop` 这个 Channel 被赋予值的时候，`geth` 主函数就会停止当前的阻塞状态，并开始执行相应的一系列的资源释放的操作。这个地方的写法还是非常有意思的，值得我们参考。我们为读者编写了一个简单的示例: 如何使用 Channel 来管理 Go 程序的生命周期。

值得注意的是，在目前的 go-ethereum 的 codebase 中，并没有直接通过给 `stop` 这个 channel 赋值方式来结束主进程的阻塞状态，而是使用一种更简洁粗暴的方式: 调用 `close()` 函数直接关闭 Channel。我们可以在 `node.doClose()` 找到相关的实现。`close()` 是 go 语言的原生函数，用于关闭 Channel 时使用。

```
// doClose releases resources acquired by New(), collecting errors.
func (n *Node) doClose(errs []error) error {
```

```
    // Close databases. This needs the lock because it needs to
    // synchronize with OpenDatabase*.
    n.lock.Lock()
    n.state = closedState
    errs = append(errs, n.closeDatabases()...)
    n.lock.Unlock()

    if err := n.accman.Close(); err != nil {
     errs = append(errs, err)
    }
    if n.keyDirTemp {
     if err := os.RemoveAll(n.keyDir); err != nil {
      errs = append(errs, err)
     }
    }

    // Release instance directory lock.
    n.closeDataDir()

    // Unblock n.Wait.
    close(n.stop)

    // Report any errors that might have occurred.
    switch len(errs) {
    case 0:
     return nil
    case 1:
     return errs[0]
    default:
     return fmt.Errorf("%v", errs)
    }
}
```

2.2.4 Ethereum Backend

我们可以在 `eth/backend.go` 中找到 `Ethereum` 这个结构体的定义。这个结构体包含的成员变量以及接收的方法实现了一个 Ethereum full node 所需要的全部功能和数据结构。我们可以在下面的代码定义中看到，Ethereum 结构体中包含 `TxPool`，`Blockchain`，`consensus.Engine`，`miner` 等最核心的几个数据结构作为成员变量，我们会在后面的章节中详细的讲述这些核心数据结构的主要功能，以及它们的实现的方法。

```
type Ethereum struct {
    config *ethconfig.Config
    txPool     *txpool.TxPool
    blockchain *core.BlockChain
   handler            *handler     // handler 是 P2P 网络数据同步的核心实例，我们会在后续的
    ethDialCandidates  enode.Iterator
    snapDialCandidates enode.Iterator
    merger             *consensus.Merger
    chainDb ethdb.Database // Block chain database
    eventMux       *event.TypeMux
    engine         consensus.Engine
    accountManager *accounts.Manager
    bloomRequests     chan chan *bloombits.Retrieval // Channel receiving bloom data retr
    bloomIndexer      *core.ChainIndexer             // Bloom indexer operating during bl
    closeBloomHandler chan struct{}
    APIBackend *EthAPIBackend
    miner     *miner.Miner
    gasPrice  *big.Int
    etherbase common.Address
    networkID     uint64
    netRPCService *ethapi.NetAPI
    p2pServer *p2p.Server
    lock sync.RWMutex // Protects the variadic fields (e.g. gas price and etherbase)
    shutdownTracker *shutdowncheck.ShutdownTracker // Tracks if and when the node has shu
}
```

节点启动和停止 Mining 的就是通过调用 `Ethereum.StartMining()` 和 `Ethereum.StopMining()` 实现的。设置 Mining 的收益账户是通过调用 `Ethereum.SetEtherbase()` 实现的。

```
// StartMining starts the miner with the given number of CPU threads. If mining
// is already running, this method adjust the number of threads allowed to use
// and updates the minimum price required by the transaction pool.
func (s *Ethereum) StartMining(threads int) error {
   ...
 // If the miner was not running, initialize it
 if !s.IsMining() {
      ...
      // Start Mining
  go s.miner.Start(eb)
 }
 return nil
}
```

这里我们额外关注一下 `handler` 这个成员变量。`handler` 的定义在 `eth/handler.go` 中。

我们从从宏观角度来看，一个节点的主工作流需要: 1. 从网络中获取/同步 Transaction 和 Block 的数据 2. 将网络中获取到 Block 添加到 Blockchain 中。而 handler 就负责提供中同步区块和交易数据的功能，例如，downloader.Downloader 负责从网络中同步 Block ，fetcher.TxFetcher 负责从网络中同步交易。关于这些方法的具体实现，我们会在后续章节：数据同步中详细介绍。

```
type handler struct {
 networkID  uint64
 forkFilter forkid.Filter // Fork ID filter, constant across the lifetime of the node

 snapSync  uint32 // Flag whether snap sync is enabled (gets disabled if we already have
 acceptTxs uint32 // Flag whether we're considered synchronised (enables transaction proc

 checkpointNumber uint64      // Block number for the sync progress validator to cross re
 checkpointHash   common.Hash // Block hash for the sync progress validator to cross refe

 database ethdb.Database
 txpool   txPool
 chain    *core.BlockChain
 maxPeers int

 downloader   *downloader.Downloader
 blockFetcher *fetcher.BlockFetcher
 txFetcher    *fetcher.TxFetcher
 peers        *peerSet
 merger       *consensus.Merger

 eventMux      *event.TypeMux
 txsCh         chan core.NewTxsEvent
 txsSub        event.Subscription
 minedBlockSub *event.TypeMuxSubscription

 peerRequiredBlocks map[uint64]common.Hash

 // channels for fetcher, syncer, txsyncLoop
 quitSync chan struct{}

 chainSync *chainSyncer
 wg        sync.WaitGroup
 peerWG    sync.WaitGroup
}
```

到此，我们就介绍了 geth 及其所需要的基本模块是如何启动的和关闭的。我们在接下来将视角转入到各个模块中，从更细粒度的角度深入探索 Ethereum 的具体实

现。

2.2.5 Related Terms

- Geth
- go-ethereum (geth)

2.2.6 Appendix

这里补充一个 Go 语言的语法知识: **类型断言**。在 `Ethereum.StartMining()` 函数中，出现了 `if c, ok := s.engine.(*clique.Clique); ok` 的写法。这中写法是 Golang 中的语法糖，称为类型断言。具体的语法是 `value, ok := element.(T)`，它的含义是如果 `element` 是 `T` 类型的话，那么 ok 等于 `True`, `value` 等于 `element` 的值。在 `if c, ok := s.engine.(*clique.Clique); ok` 语句中，就是在判断 `s.engine` 的是否为 `*clique.Clique` 类型。

```
var cli *clique.Clique
if c, ok := s.engine.(*clique.Clique); ok {
 cli = c
} else if cl, ok := s.engine.(*beacon.Beacon); ok {
 if c, ok := cl.InnerEngine().(*clique.Clique); ok {
  cli = c
 }
}
```

2.3 总结

在本章节中，我们简述了 Go-Ethereum 中启动和关闭 Node 工作流。感兴趣的读者可以继续阅读源代码来深入理解这个模块。

第 3 章

账户和合约 (Account and Contract)

我们常常听到这样一个说法，“Ethereum 和 Bitcoin 最大的不同之一是二者使用链上数据模型不同。其中，Bitcoin 是基于 UTXO 模型的 Blockchain/Ledger 系统，Ethereum 是基于 Account/State 模型的系统”。那么，Account/State 模型相比于 UTXO 究竟不同在何处呢？在本文，我们就来探索一下以太坊中的基本数据结构之一的 `Account`。

简单的来说，Ethereum 的运行是一种基于交易的状态机模型 (Transaction-based State Machine)。整个系统由若干的账户组成 (Account)，类似于银行账户。状态 (State) 反应了某一账户 (Account) 在某一时刻下的值 (value)。在以太坊中，State 对应的基本数据结构，称为 StateObject。当 StateObject 的值发生了变化时，我们称为状态转移。在 Ethereum 的运行模型中，StateObject 所包含的数据会因为 Transaction 的执行引发数据更新/删除/创建，引发状态转移，我们说：StateObject 的状态从当前的 State 转移到另一个 State。

在 Ethereum 中，承载 StateObject 的具体实例就是 Ethereum 中的 Account 。通常，我们提到的 State 具体指的就是 Account 在某个时刻下所包含的数据的值。

- Account –> StateObject
- State –> The value/data of the Account

总的来说，Account (账户) 是参与链上交易 (Transaction) 的基本角色，是 Ethereum 状态机模型中的基本单位，承担了链上交易的发起者以及接收者的角色。

目前，在以太坊中，有两种类型的 Account，分别是外部账户 (EOA) 以及合约账户 (Contract)。

外部账户 (EOA) 是由用户直接控制的账户，负责签名并发起交易 (Transaction)。用户通过控制 Account 的私钥来保证对账户数据的控制权。

合约账户 (Contract)，简称为合约，是由外部账户通过 Transaction 创建的。合约账户保存了**不可篡改的图灵完备的代码段**，以及一些**持久化的数据变量**。这些代码使用专用的图灵完备的编程语言编写 (Solidity)，并通常提供一些对外部访问 API 接口函数。这些 API 接口函数可以通过构造 Transaction，或者通过本地/第三方提供的节点 RPC 服务来调用。这种模式构成了目前的 DApp 生态的基础。

通常，合约中的函数用于计算以及查询或修改合约中的持久化数据。我们经常看到这样的描述”**一旦被记录到区块链上数据不可被修改**，或者**不可篡改的智能合约**”。现在我们知道这种笼统的描述其实是不准确。针对一个链上的智能合约，不可修改/篡改的部分是合约中的代码段，或说合约中的函数逻辑/代码逻辑是不可以被修改/篡改的。而合约中的**持久化的数据变量**是可以通过调用代码段中的函数进行数据操作的 (CURD)。具体的操作方式取决于合约函数中的代码逻辑。

根据合约中函数是否会修改合约中持久化的变量，合约中的函数可以分为两种: 只读函数和写函数。如果用户**只**希望查询某些合约中的持久化数据，而不对数据进行修改的话，那么用户只需要调用相关的只读函数。调用只读函数不需要通过构造一个 Transaction 来查询数据。用户可以通过直接调用本地节点或者第三方节点提供的 RPC 接口来直接调用对应的合约中的只读函数。如果用户需要对合约中的数据进行更新，那么他就要构造一个 Transaction 来调用合约中相对应的写函数。注意，每个 Transaction 每次调用一个合约中的一个写函数。因为，如果想在链上实现复杂的逻辑，需要将写函数接口化，在其中调用更多的逻辑。

对于如何编写合约，以及 Ethereum 的执行层如何解析 Transaction 并调用对应的合约中函数的，我们会在后面的文章中详细的进行解析。

3.1 StateObject, EOA 和 Contract

3.1.1 概述

在实际代码中，这两种 Account 都是由 `stateObject` 这一数据结构定义的。`stateObject` 的相关代码位于 *core/state/state_object.go* 文件中，隶属于 *package*

state。我们摘录了 stateObject 的结构代码，如下所示。通过下面的代码，我们可以观察到，stateObject 是由小写字母开头。根据 go 语言的特性，我们可以知道这个结构主要用于 package 内部数据操作，并不对外暴露。

```
type stateObject struct {
  address  common.Address
  addrHash common.Hash // hash of ethereum address of the account
  data     types.StateAccount
  db       *StateDB
  dbErr error

  // Write caches.
  trie Trie // storage trie, which becomes non-nil on first access
  code Code // contract bytecode, which gets set when code is loaded

  // 这里的 Storage 是一个 map[common.Hash]common.Hash
  originStorage  Storage // Storage cache of original entries to dedup rewrites, reset
  pendingStorage Storage // Storage entries that need to be flushed to disk, at the end
  dirtyStorage   Storage // Storage entries that have been modified in the current tran
  fakeStorage    Storage // Fake storage which constructed by caller for debugging purp

  // Cache flags.
  // When an object is marked suicided it will be delete from the trie
  // during the "update" phase of the state transition.
  dirtyCode bool // true if the code was updated
  suicided  bool
  deleted   bool
}
```

3.1.2 Address

在 stateObject 这一结构体中，开头的两个成员变量为 address 以及 address 的哈希值 addrHash。address 是 common.Address 类型，addrHash 是 common.Hash 类型，它们分别对应了一个 **20 字节**长的 byte 类型数组和一个 32 字节长的 byte 类型数组。关于这两种数据类型的定义如下所示。

```
// Lengths of hashes and addresses in bytes.
const (
 // HashLength is the expected length of the hash
 HashLength = 32
 // AddressLength is the expected length of the address
 AddressLength = 20
)
```

```
// Address represents the 20 byte address of an Ethereum account.
type Address [AddressLength]byte
// Hash represents the 32 byte Keccak256 hash of arbitrary data.
type Hash [HashLength]byte
```

在 Ethereum 中，每个 Account 都拥有独一无二的地址。Address 作为每个 Account 的身份信息，类似于现实生活中的身份证，它与用户信息时刻绑定而且不能被修改。

3.1.3 data and StateAccount

继续向下探索，我们会遇到成员变量 `data`，它是一个 `types.StateAccount` 类型的变量。在上面的分析中我们提到，`stateObject` 这种类型只对 Package State 这个内部使用。所以相应的，Package State 也为外部 Package API 提供了与 Account 相关的数据类型 `State Account`。在上面的代码中我们就可以看到，`State Account` 对应了 `State Object` 中 `data Account` 成员变量。State Account 的具体数据结构的被定义在 `core/types/state_account.go` 文件中 (在之前的版本中 Account 的代码位于 core/account.go)，其定义如下所示。

```
// Account is the Ethereum consensus representation of accounts.
// These objects are stored in the main account trie.
type StateAccount struct {
  Nonce    uint64
  Balance  *big.Int
  Root     common.Hash // merkle root of the storage trie
  CodeHash []byte
}
```

其中的包含四个变量为:

- Nonce 表示该账户发送的交易序号，随着账户发送的交易数量的增加而单调增加。每次发送一个交易，Nonce 的值就会加 1。
- Balance 表示该账户的余额。这里的余额指的是链上的 Native Token Ether (以太)。
- Root 表示当前账户的下 Storage 层的 Merkle Patricia Trie 的 Root。这里的存储层是为了管理合约中持久化变量准备的。对于 EOA 账户这个部分为空值。
- CodeHash 是该账户的 Contract 代码的哈希值。同样的，这个变量是用于保存合约账户中的代码的 hash ，EOA 账户这个部分为空值。

3.1.4 db

上述的几个成员变量基本覆盖了 Account 主生命周期相关的全部成员变量。那么我们继续向下看，会遇到 `db` 和 `dbErr` 这两个成员变量。db 这个变量保存了一个 `StateDB` 类型的指针。这是为了方便调用 `StateDB` 相关的 API 对 Account 所对应的 `stateObject` 进行操作。StateDB 本质上是用于管理 `stateObject` 信息的而抽象出来的内存数据库。所有的 Account 数据的更新，检索都会使用 StateDB 提供的 API。关于 StateDB 的具体实现，功能，以及如何与更底层物理存储层 (leveldb) 进行结合的，我们会在之后的文章中进行详细描述。

3.1.5 Cache

对于剩下的成员变量，它们的主要用于内存缓存。trie 用于保存和管理合约账户中的持久化变量存储的数据，code 用于缓存合约中的代码段到内存中，它是一个 byte 类型的数组。剩下的四个 Storage 字段主要在执行 Transaction 的时候缓存合约修改的持久化数据，比如 dirtyStorage 就用于缓存在 Block 被 Finalize 之前，Transaction 所修改的合约中的持久化存储数据。对于外部账户，由于没有代码字段，所以对应 stateObject 对象中的 code 字段，以及四个 Storage 类型的字段对应的变量的值都为空 (originStorage, pendingStorage, dirtyStorage, fakeStorage)。

从调用关系上看，这四个缓存变量的修改顺序是: originStorage –> dirtyStorage–> pendingStorage。关于合于中的 Storage 层的详细信息，我们会在后面部分进行详细的描述。

3.2 深入 Account (EOA)

3.2.1 谁掌握了你的账户

我们经常会在各种科技网站/自媒体上看到这样的说法，“用户在区块链系统中保存的 Cryptocurrency/Token，除了用户自己，不存在第三方可以不经过用户的允许转走你的财富”。这个说法基本是正确的。目前，用户账户里的由链级别定义的 Crypto/Token，或者称为原生货币 (Native Token)，比如 Ether，Bitcoin，BNB(Only in BSC)，是没办法被第三方在不被批准的情况下转走的。这是因为链级别上的所有数据的修改都要执行由用户私钥 (Private Key) 签名的 Transaction。因此，只要用户保

管好自己账户的私钥 (Private Key)，保证其没有被第三方知晓，就没有人可以转走你链上的财富。

我们说上述说法是基本正确，而不是完全正确。原因有两个。首先，用户的链上数据安全是基于当前 `Go-ethereum` 中使用的密码学工具足够保证：不存在第三方可以在**有限的时间**内在**不知道用户私钥的前提**下获取到用户的私钥信息来伪造签名交易。这个安全保证前提是当今 Ethereum 使用的密码学工具的强度足够大，没有计算机可以在有限的时间内 hack 出用户的私钥信息。在量子计算机出现之前，目前 Ethereum 和其他 Blockchain 使用的密码学工具的强度都是足够安全的。这也是为什么很多新的区块链项目在研究抗量子计算机密码体系的原因。第二点原因是，当今很多的所谓的 Crypto/Token 并不是链级别的代币，而是保存在合约中持久化变量中的数据，比如 ERC-20 Token 和 NFT 对应的 ERC-721 的 Token。由于这部分的 Token 都是基于合约代码生成和维护的，所以这部分 Token 的安全依赖于合约本身的安全。如果合约本身的代码是有问题的，存在后门或者漏洞，比如存在给第三方任意提取其他账户下 Token 的漏洞。那么即使用户的私钥信息没有泄漏，合约中的 Token 仍然可以被第三方获取到。由于合约的代码段在链上是不可修改的，因此合约代码的安全性是极其重要的。目前有很多研究人员，技术团队在进行合约审计方面的工作，来保证上传的合约代码是安全的。随着 Layer-2 技术和一些跨链技术的发展，用户持有的 `Token` ，在很多情况下并不是我们上面提到的由私钥来保证安全的 Naive Token，而是 ERC-20 Token。这种 Token 只是合约中的简单数值记录。这种类型的资产的安全性是远低于低于 layer-1 上的 Native Token 的。用户在持有这类资产的时候需要小心。这里我们推荐阅读 Jay Freeman 所分析的关于一个热门 Layer-2 系统 Optimism 上的由于非 Naive Token 造成的任意提取漏洞。

3.2.2 Account Generation

首先，EOA 账户的创建分为本地创建和链上注册两个部分。当我们使用诸如 Metamask 等钱包工具创建账户的时候，在区块链上并没有同步注册账户信息。链上账户的创建和管理都是通过 `StateDB` 模块来操作的，因此我们将 `geth` 中账户管理部分的代码整合到 `StateDB` 模块章节来一起讲述。而合约账户，或者说智能合约的创建是需要通过 EOA 账户构造特定的交易生成的。关于这部分的细节我们也放在之后的章节中进行解析。

下面我们简单分析一下，如何在本地创建一个 EOA 账户的。

总的来说，创建新账户的依赖的入口函数 `NewAccount` 位于 `accounts/keystore/keystore.go`

文件中。函数有一个 string 类型的 passphrase 参数。注意，这个参数仅用于加密本地保存私钥的 Keystore 文件，与生成账户的私钥，地址的生成都无关。

```
// passphrase 参数用于本地加密
func (ks *KeyStore) NewAccount(passphrase string) (accounts.Account, error) {
//生成 account 的函数
 _, account, err := storeNewKey(ks.storage, crand.Reader, passphrase)
 if err != nil {
  return accounts.Account{}, err
 }
 // Add the account to the cache immediately rather
 // than waiting for file system notifications to pick it up.
 ks.cache.add(account)
 ks.refreshWallets()
 return account, nil
}
```

上述代码段中，最核心的调用是 `storeNewKey` 函数。在 `storeNewKey` 函数中，首先就调用了 `newKey` 函数，该函数的主要功能就是生成一个账户需要的私钥和公钥对。而 `newKey` 函数的核心是调用了生成椭圆曲线加密对相关的函数 `ecdsa.GenerateKey`。

```
func newKey(rand io.Reader) (*Key, error) {
 privateKeyECDSA, err := ecdsa.GenerateKey(crypto.S256(), rand)
 if err != nil {
  return nil, err
 }
 return newKeyFromECDSA(privateKeyECDSA), nil
}
```

这部分的代码位于 `crypto/ecdsa.go` 文件中。由于这一部分涉及到了大量的椭圆曲线加密的知识，与以太坊的主要业务关系不大，因此对于这部分的内容，我们仅简述主要流程。对于密码学原理的部分在此我们不进行详细说明，感兴趣的读者可以自行搜索。值得注意的时候，在整个流程中，首先生成的是账户的私钥，而账户对应的地址，是基于该私钥在椭圆曲线上对用的公钥值经过哈希计算得到的。下面我们简述一下，如何从账户私钥计算出账户地址的。

- 首先，我们在创建一个新的 EOA 账户的时候，首先会通过 `GenerateKey` 函数随机的得到一串私钥，它是一个 32bytes 长的变量，表现为 64 位 16 进制数。这个私钥就是平时需要用户激活钱包时，发送交易时必要的门禁卡，一旦这个私钥暴露了，钱包也将不再安全。
  - 64 个 16 进制位，256bit，32 字节 `var AlicePrivateKey = "289c2857d4598e37fb9647507e47a309d6133539bf21a8b9cb6df88fd5232032"`

- 在得到私钥后，我们使用私钥来计算公钥和地址地址。基于上述私钥，我们使用 ECDSA 算法，选择 spec256k1 曲线进行计算。通过将私钥带入到所选择的椭圆曲线中，计算出点的坐标即是公钥。以太坊和比特币使用了同样的 spec256k1 曲线，在实际的代码中，我们也可以看到在 go-Ethereum 直接调用了比特币的 secp256k1 的 C 语言代码。`ecdsaSK, err := crypto.ToECDSA(privateKey)`
- 对私钥进行椭圆加密之后，我们可以得到一个 64bytes 的数，它是由两个 32bytes 的数构成，这两个数代表了 spec256k1 曲线上某个点的 XY 坐标值。`ecdsaPK := ecdsaSK.PublicKey`
- 最终账户的地址，是基于上述公钥 (ecdsaSK.PublicKey) 进行 **Keccak-256 算法**计算之后得到的哈希值的后 20 个字节，用 0x 开头表示 (Keccak-256 是 SHA-3（Secure Hash Algorithm 3）标准下的一种哈希算法)。`addr := crypto.PubkeyToAddress(ecdsaSK.PublicKey)`

3.2.2.1 Signature & Verification

这里我们简述一下，怎么利用 ECDSA 来进行数字签名和校验的。

- Hash（m,R）$X + R = S$ P
- P 是椭圆曲线函数的基点 (base point) 可以理解为一个 P 是一个在曲线 C 上的一个 order 为 n 的加法循环群的生成元，n 是一个大质数。
- R = r * P (r 是个随机数，并不告知 verifier)
- 以太坊签名校验的核心思想是: 首先基于上面得到的 ECDSA 下的私钥 ecdsaSK 对数据 msg 进行签名 (sign) 得到 msgSig. `sig, err := crypto.Sign(msg[:], ecdsaSK)` `msgSig := decodeHex(hex.EncodeToString(sig))`
- 然后基于 msg 和 msgSig 可以反推出来签名的公钥（用于生成账户地址的公钥 ecdsaPK）。`recoveredPub, err := crypto.Ecrecover(msg[:],msgSig)`
- 通过反推出来的公钥可以得到发送者的地址，并与当前交易的发送者在 ECDSA 下的 pk 进行对比。`crypto.VerifySignature(testPk, msg[:], msgSig[:len(msgSig)-1])`
- 这套体系的安全性保证在于，即使知道了公钥 ecdsaPk/ecdsaSK.PublicKey 也难以推测出 ecdsaSK 以及生成他的 privateKey。

3.2.2.2 ECDSA & spec256k1 曲线

最后，我们来简述一下 ECDSA 的原理。感兴趣的读者可以以此为切入点自行搜索。

- spec256k1 的解析函数 y^2 = x^3 +7
- 在椭圆曲线上的有一类特殊的计算称为 Elliptic curve point multiplication, 它的计算规则如下
  - Point addition P + Q = R
  - Point doubling P + P = 2P
- 在 ECC 中的 + 号不是四则运算中的加法，而是定义椭圆曲线 C 上的新的二元运算 (Point Multiplication)。他代表了过两点 P 和 Q 的直线与椭圆曲线 C 的交点 R'关于 X 轴对称的点 R。因为 C 是关于 X 轴对称的所以关于 X 对称的点也都在椭圆曲线上。
- Based Point P 是在椭圆曲线上的群的生成元
- x 次 computation on Based Point 得到 X 点，x 为私钥，X 为公钥。x 由 Account Private Key 得出。

3.3 合约和合约存储 (Storage)

- 这部分的示例代码位于: [example/signature] 中。

3.3.1 Contract Storage (合约存储)

在文章的开头我们提到，在外部账户对应的 stateObject 结构体的实例中，有四个 Storage 类型的变量是空值。那显然的，这四个变量是为 Contract 类型的账户准备的。

在 `state_object.go` 文件的开头部分 (41 行左右)，我们可以找到 `Storage` 类型的定义。具体如下所示。

```
1 type Storage map[common.Hash]common.Hash
```

我们可以看到，`Storage` 是一个 key 和 value 都是 `common.Hash` 类型的 map 结构。`common.Hash` 类型本质上是一个长度为 32bytes 的 `byte` 类型数组。`common.Hash`

类型在 go-ethereum 的代码库中被大量使用，通常用于表示 32 字节长度的数据，比如 Keccak256 函数的哈希值。在之后的旅程中，我们也会经常看到它的身影，它的定义在 common.type.go 文件中。

```
// HashLength is the expected length of the hash
HashLength = 32
// Hash represents the 32 byte Keccak256 hash of arbitrary data.
type Hash [HashLength]byte
```

从数据层面讲，外部账户 (EOA) 与合约账户 (Contract) 不同的点在于: 外部账户并没有维护自己的代码 (codeHash) 以及额外的 Storage 层。相比与外部账户，合约账户额外保存了一个存储层 (Storage) 用于存储合约代码中持久化的变量的数据。在上文中我们提到，StateObject 结构体中的声明的四个 Storage 类型的变量，就是作为 Contract Storage 层的内存缓存。

在 Ethereum 中，每个合约都维护了自己的独立的存储空间，用于保存合约中的持久化变量，我们称为 Storage 层。Storage 层的基本组成单元称为槽 (Slot)。若干个 Slot 按照栈 (Stack) 的方式顺序集合在一起就构造成了 Storage 层。每个 Slot 的大小是 256 bits，也就是最多保存 32 bytes 的数据。作为基本的存储单元，Slot 管理的方式与内存或者 HDD 中的基本单元的管理方式类似，通过地址索引的方式被上层函数访问。Slot 的地址索引的长度同样是 32 bytes(256 bits)，寻址空间从 0x0000000000000000000000000000000000000000000000000000000000000000 到 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF。因此，每个 Contract 的 Storage 层最多可以保存 $2^{256} - 1$ 个 Slot。也就说在理论状态下，一个 Contract 可以最多保存 $(2^{256} - 1)$ bytes 的数据，这是个相当大的数字。

为了更好的管理 Storage 层的 Slot 数据，Contract 同样使用 MPT 作为索引来管理 Storage 层的 Slot。这里值得注意的是，合约 Storage 层的数据并不会跟随交易一起被打包进入 Block 中。正如我们之前提到的，管理合约账户中 Storage 层 Storage Trie 的根数据被保存在 StateAccount 结构体中的 Root 变量中 (它是一个 32bytes 长的 byte 数组)。当某个 Contract 的 Storage 层的数据发生变化时，会向上传导，并更新 World State Root 的值，从而影响到 Chain 链上数据。目前，Storage 层的数据读取和修改是在执行相关 Transaction 的时候，通过调用 EVM 中的两个专用的指令 *OpSload* 和 *OpSstore* 来执行的。关于这两个指令的具体实现原理，我们会在后续的 EVM 章节进行详细的解读。

我们知道目前 Ethereum 中的大部分合约都通过 Solidity 语言编写。Solidity

做为强类型的图灵完备的语言，支持多种类型的变量。总的来说，根据变量的长度性质，Ethereum 中的持久化的变量可以分为定长的变量和不定长度的变量两种。定长的变量有常见的单变量类型，比如 uint256。不定长的变量包括了由若干单变量组成的 Array，以及 KV 形式的 Map 类型。

根据上面的介绍，我们了解到对 Contract Storage 层的访问是通过 Slot 的地址来进行的。请读者先思考下面的几个问题:

- **如何给定一个包含若干持久化存储变量的 Solidity 的合约，EVM 是怎么给其包含的变量分配存储空间的呢？**
- 怎么保证 Contract Storage 的一致性读写的？(怎么保证每个合约的验证者和执行者都能获取到相同的数据？)

我们将通过下面的一些实例来展示，在 Ethereum 中，Contract 是如何保存持久化变量的，以及保证所有的参与者都能一致性读写的 Contract 中的数据的。

3.3.2 Contract Storage Example One

我们使用一个简单的合约来展示 Contract Storage 层的逻辑，合约代码如下所示。在本例中，我们使用了一个叫做 `Storage` 合约，其中定义了了三个持久化 uint256 类型的变量分别是 number, number1, 以及 number2。同时，我们定义一个 `stores` 函数给这个三个变量进行赋值。合约代码如下所示。

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 */
contract Storage {

    uint256 number;
    uint256 number1;
    uint256 number2;

    function stores(uint256 num) public {
        number = num;
        number1 = num + 1;
        number2 = num + 2;
```

```
    }

    function get_number() public view returns (uint256){
        return number;
    }

    function get_number1() public view returns (uint256){
        return number1;
    }

    function get_number2() public view returns (uint256){
        return number2;
    }
}
```

我们使用Remix来在本地部署这个合约，并构造一个调用 `stores(1)` 函数的交易，同时使用 Remix debugger 来观察 Storage 层的变化。在交易执行之后，合约中三个变量的值将被分别赋给 1，2，3。此时，我们观察 Storage 层会发现，存储层增加了三个 Storage Object 。这三个 Storage Object 对应了三个 Slot。所以在本例中，合约增加了三个 Slots 来存储数据。我们可以发现每个 Storage Object 由三个字段组成，分别是一个 32 bytes 的 key 字段和 32 bytes 的 value 字段，以及外层的一个 32 bytes 的字段。这三个字段在下面的例子中都表现为 64 位的 16 进制数 (32 Bytes)。

下面我们来逐个解释一下这个三个值的实际意义。首先我们观察内部的 Key-Value 对，可以发现下面三个 Storage Object 中 key 的值其实是从 0 开始的递增整数，分别是 0，1，2。它代表了当前 Slot 的地址索引值，或者说该 Slot 在 Storage 层对应的绝对位置 (Position)。比如，key 的值为 0 时，它代表整个 Storage 层中的第 1 个 Slot，或者说在 1 号位置的 Slot，当 key 等于 1 时代表 Storage 层中的第 2 个 Slot，以此类推。每个 Storage Object 中的 value 变量，存储了合约中三个变量的值 (1,2,3)。而 Storage Object 外层的值则等于 Storage Object 的 key 的值的 sha3 的哈希值。比如，下面例子中的第一个 Storage Object 的外层索引值 `0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563` 是等于 `keccak256(0)` 计算出的值，它代表了第一个 Slot position 的 Sha3 的哈希，而 `0xb10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6` 等于 `keccak(1)` 计算出的值。我们在示例代码中展示了如何计算的过程。

```
{
 "0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563": {
  "key": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "value": "0x0000000000000000000000000000000000000000000000000000000000000001"
```

```
 },
 "0xb10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6": {
  "key": "0x0000000000000000000000000000000000000000000000000000000000000001",
  "value": "0x0000000000000000000000000000000000000000000000000000000000000002"
 },
 "0x405787fa12a823e0f2b7631cc41b3ba8828b3321ca811111fa75cd3aa3bb5ace": {
  "key": "0x0000000000000000000000000000000000000000000000000000000000000002",
  "value": "0x0000000000000000000000000000000000000000000000000000000000000003"
 }
}
```

到这里读者可能已经发现了，在这个 Storage Object 中，外层的索引值其实与 Key 值的关系是一一对应的。换句话说，这两个键值本质上都是关于 Slot 位置的唯一索引。这里我们简单讲述一下这两个值在使用上的区别。Key 值代表了 Slot 在 Storage 层的 Position，这个值会在实际的代码中作为 `stateObject.go/getState()` 以及 `setState()` 函数的参数，用于定位 Slot。如果我们继续深入上面的两个函数，我们就会发现，当内存中不存在该 Slot 的缓存时，`geth` 就会尝试从更底层的数据库中来获取这个 Slot 的值。而正如我们之前提到的，Storage 层使用了一个 MPT 的结构作为访问底层数据的索引结构。为了 MPT 树的平衡，这里在具体实现的时候使用了一个 `Secure Trie` 的特殊结构。与常规的 MPT 不同的是，`Secure Trie` 中的节点的 Key 值都是需要 Hash 的。因此当我们使用 `Secure Trie` 来查询/修改需要的数据时，需要使用哈希之后的值作为索引键，具体就是上述例子中的外层 hash 值。具体的关于 Secure Trie 的描述可以参考Trie这一章节。总结下来，在上层函数 (stateObject) 调用中使用的键值是 Slot 的 Position，在下层的函数 (Trie) 调用中使用的键值是 Slot 的 Position 的哈希值。

```
func (t *SecureTrie) TryGet(key []byte) ([]byte, error) {
  // Secure Trie 中查询的例子
  // 这里的 key 还是 Slot 的 Position
  // 但是在更下层的 Call 更下层的函数的时候使用了这个 Key 的 hash 值作为查询使用的键值。
  return t.trie.TryGet(t.hashKey(key))
}
```

3.3.3 Account Storage Example Two

下面我们来看另外的一个例子。在这个例子中，我们调整了一下合约中变量的声明顺序，从 (number，number1，number2) 调整为 (number 2, number 1, number)。合约代码如下所示。

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 */
contract Storage {

    uint256 number2;
    uint256 number1;
    uint256 number;

    function stores(uint256 num) public {
        number = num;
        number1 = num + 1;
        number2 = num + 2;
    }

    function get_number() public view returns (uint256){
        return number;
    }

    function get_number1() public view returns (uint256){
        return number1;
    }

    function get_number2() public view returns (uint256){
        return number2;
    }
}
```

同样我们还是构造交易来调用合约中的 `stores` 函数。此时我们可以在 Storage 层观察到不一样的结果。我们发现 number2 这个变量的值被存储在了第一个 Slot 中（Key:“0x0000000000000000000000000000000000000000000000000000000000000000”），而 number 这个变量的值被存储在了第三个 Slot 中（Key:“0x0000000000000000000000000000000000000000000000000000000000000002”）。

```
{
  "0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563": {
    "key": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "value": "0x0000000000000000000000000000000000000000000000000000000000000003"
```

```
    },
  "0xb10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6": {
    "key": "0x0000000000000000000000000000000000000000000000000000000000000001",
    "value": "0x0000000000000000000000000000000000000000000000000000000000000002"
  },
  "0x405787fa12a823e0f2b7631cc41b3ba8828b3321ca811111fa75cd3aa3bb5ace": {
    "key": "0x0000000000000000000000000000000000000000000000000000000000000002",
    "value": "0x0000000000000000000000000000000000000000000000000000000000000001"
  }
}
```

这个例子可以说明，在 `go-ethereum` 中，变量对应的存储层的 Slot，是按照其在在合约中的声明顺序，从第一个 Slot（position：0）开始分配的。

3.3.4 Account Storage Example Three

我们再考虑另一种情况：声明的三个变量，但只对其中的两个变量进行赋值。具体的来说，我们按照 number，number1，和 number2 的顺序声明三个 `uint256` 变量。但是，在函数 `stores` 中只对 number1 和 number2 进行赋值操作。合约代码如下所示。

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 */
contract Storage {
    uint256 number;
    uint256 number1;
    uint256 number2;

    function stores(uint256 num) public {
        number1 = num + 1;
        number2 = num + 2;
    }

    function get_number() public view returns (uint256){
        return number;
    }

    function get_number1() public view returns (uint256){
```

```
        return number1;
    }

    function get_number2() public view returns (uint256){
        return number2;
    }
}
```

基于上述合约，我们构造交易并调用 `stores` 函数，输入参数 1，将 number1 和 number2 的值修改为 2，和 3。在交易执行完成后，我们可以观察到 Storage 层 Slot 的结果如下所示。

```
{
 "0xb10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6": {
  "key": "0x0000000000000000000000000000000000000000000000000000000000000001",
  "value": "0x0000000000000000000000000000000000000000000000000000000000000002"
 },
 "0x405787fa12a823e0f2b7631cc41b3ba8828b3321ca811111fa75cd3aa3bb5ace": {
  "key": "0x0000000000000000000000000000000000000000000000000000000000000002",
  "value": "0x0000000000000000000000000000000000000000000000000000000000000003"
 }
}
```

我们可以观察到，在 `stores` 函数执行后，只对 Storage 层中位置在 1 和 2 位置的两个 Slot 进行了赋值。值得注意的是，在本例中，对于 Slot 的赋值是从 1 号位置 Slot 的开始，而不是 0 号 Slot。这说明对于固定长度的变量，其值的所占用的 Slot 的位置在合约初始化开始的时候就已经分配的。即使变量只是被声明还没有真正的赋值，保存其值所需要的 Slot 也已经被 EVM 分配完毕。而不是在第一次进行变量赋值的时候，进行再对变量所需要的的 Slot 进行分配。

3.3.5　Account Storage Example Four

在 Solidity 中，有一类特殊的变量类型 **Address**，通常用于表示账户的地址信息。例如在 ERC-20 合约中，用户拥有的 token 信息是被存储在一个 (address->uint) 的 map 结构中。在这个 map 中，key 就是 Address 类型的，它表示了用户实际的 address 。目前 Address 的大小为 160bits(20bytes)，并不足以填满一整个 Slot。因此当 Address 作为 value 单独存储在的时候，它并不会排他的独占用一个 Slot。我们使用下面的例子来说明。

在下面的示例中，我们声明了三个变量，分别是 number(uint256)，addr(address)，以及 isTrue(bool)。我们知道，在以太坊中 Address 类型变量的长度是 20 bytes，

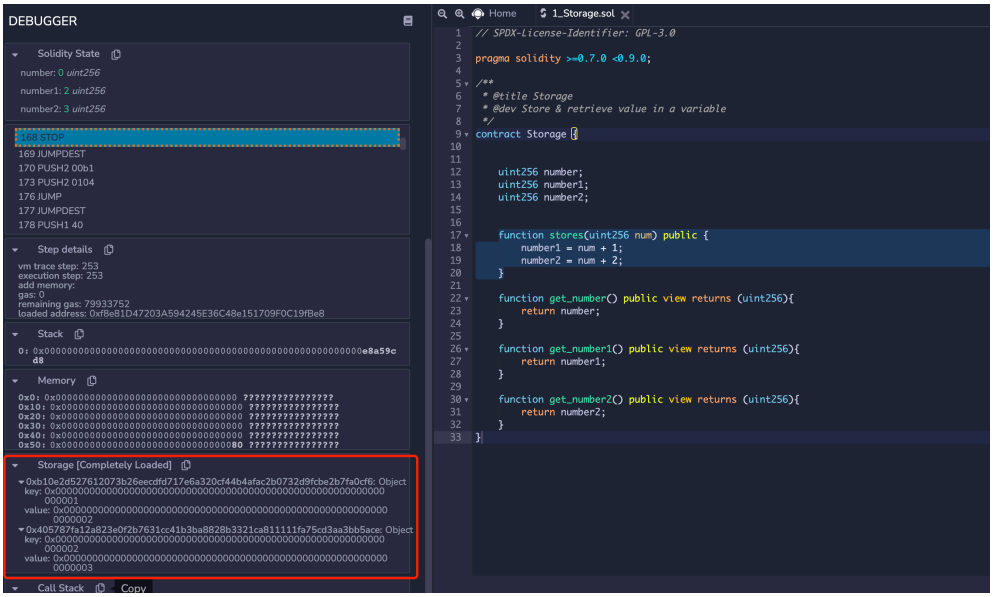


图 3.1: Remix Debugger

所以一个 Address 类型的变量是没办法填满整个的 Slot(32 bytes) 的。同时，布尔类型在以太坊中只需要一个 bit(0 or 1) 的空间. 因此，我们构造 transaction 并调用函数 `storeaddr()` 来给这三个变量赋值，函数的 input 参数是一个 uint25 6 的值，一个 address 类型的值，分别为 {1, `0xb6186d3a3D32232BB21E87A33a4E176853a49d12`}。

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 */
contract Storage {

    uint256 number;
    address addr;
    bool isTrue;

    function stores(uint256 num) public {
        // number1 = num + 1;
        // number2 = num + 2;
    }

```

```
    function storeaddr(uint256 num, address a) public {
        number = num;
        addr = a;
        isTure = true;
    }

    function get_number() public view returns (uint256){
        return number;
    }

}
```

交易的运行后 Storage 层的结果如下面的 Json 所示。我们可以观察到，在本例中 Contract 声明了三个变量，但是在 Storage 层只调用了两个 Slot。第一个 Slot 用于保存了 uint256 的值，而在第二个 Slot 中 (Key:`0x0000000000000000000000000000000000000000000000000000000000000001`) 保存了 `addr` 和 `isTrue` 的值。这里需要注意，虽然这种将两个小于 32 bytes 长的变量合并到一个 Slot 的做法节省了物理空间，但是也同样带来读写放大的问题。因为在 `Geth` 中，读操作最小的读的单位都是按照 `32bytes` 来进行的 (由于 `OpSload` 指令的实际调用)。在本例中，即使我们只需要读取 `isTrue` 或者 `addr` 这两个变量的值，在具体的函数调用中，我们仍然需要将对应的 Slot 先读取到内存中。同样的，如果我们想修改这两个变量的值，同样需要对整个的 Slot 进行重写。这无疑增加了额外的开销。所以在 Ethereum 使用 32 bytes 的变量，在某些情况下消耗的 Gas 反而比更小长度类型的变量要小 (例如 unit8)。这也是为什么 Ethereum 官方也建议使用长度为 32 bytes 变量的原因。

// Todo Gas cost? here or in EVM Section


```
{
 "0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563": {
  "key": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "value": "0x0000000000000000000000000000000000000000000000000000000000000001"
 },
 "0xb10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6": {
  "key": "0x0000000000000000000000000000000000000000000000000000000000000001",
  "value": "0x000000000000000000000001b6186d3a3d32232bb21e87a33a4e176853a49d12"
 }
}
```

3.3.6 Account Storage Example Five

对于变长数组和 Map 结构的变量存储分配则相对的复杂。虽然 Map 本身就是 key-value 的结构，但是在 Storage 层并不直接使用 map 中 key 的值或者 key 的值的 sha3 哈希值来作为 Storage 分配的 Slot 的索引值。目前，EVM 首先会使用 map 中元素的 key 的值和当前 Map 变量声明位置对应的 slot 的值进行拼接，再使用拼接后的值的 keccak256 哈希值作为 Slot 的位置索引 (Position)。我们在下面的例子中展示了 Ethereum 是如何处理 map 这种变长的数据结构的。在下面的合约中，我们声明了一个定长的 uint256 类型的对象 number，和一个 [string=>uint256] 类型的 Map 对象。

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 */
contract Storage {

    uint256 number;

    mapping(string => uint256) balances;

    function set_balance(uint256 num) public {
        number = num;
        balances["hsy"] = num;
        balances["lei"] = num + 1;
    }

    function get_number() public view returns (uint256){
        return number;
    }

}
```

我们构造一个交易来调用 set_balance 函数。在交易执行之后的 Storage 层的结果如下面的 Json 所示。我们发现，对于定长的变量 number 占据了第一个 Slot 的空间 (Position:0x0000000000000000000000000000000000000000000000000000000000000000)。但是对于 Map 类型变量 balances，它包含的两个数据并没有按照变量定义的物理顺序来定义 Slot。此外，我们观察到存储这两个值的 Slot 的 key，也并不是这两个字

在 mapping 中 key 的直接 hash。正如我们在上段中提到的那样，EVM 会使用 Map 中元素的 key 值与当前 Map 被分配的 Slot 的位置进行拼接，之后对拼接之后对值进行使用 `keccak256` 函数求得哈希值，来最终得到 map 中元素最终的存储位置。比如在本例中，按照变量定义的顺序，`balances` 这个 Map 变量会被分配到第二个 Slot，对应的 Slot Position 是 `1`。因此，`balances` 中的 kv 对分配到的 Slot 的位置就是，`keccak(key, 1)`，这里是一个特殊的拼接操作。

```
{
 "0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563": {
  "key": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "value": "0x0000000000000000000000000000000000000000000000000000000000000001"
 },
 "0xa601d8e9cd2719ca27765dc16042655548d1ac3600a53ffc06b4a06a12b7c65c": {
  "key": "0xbaded3bf529b04b554de2e4ee0f5702613335896b4041c50a5555b2d5e279f91",
  "value": "0x0000000000000000000000000000000000000000000000000000000000000001"
 },
 "0x53ac6681d92653b13055d2e265b672e2db2b2a19407afb633928597f144edbb0": {
  "key": "0x56a8a0d158d59e2fd9317c46c65b1e902ed92f726ecfe82c06c33c015e8e6682",
  "value": "0x0000000000000000000000000000000000000000000000000000000000000002"
 }
}
```

为了验证上面的说法，我们使用 go 语言编写了一段代码，来调用相关的库来验证一下上面的结论。对于 balances[“hsy”]，它被分配的 Slot 的位置可以由下面的代码求得。读者可以阅读/使用示例代码进行尝试。这里的 `k1` 是一个整形实数，代表了 Slot 的在 storage 层的位置 (Position)。

```
k1 := solsha3.SoliditySHA3([]byte("hsy"), solsha3.Uint256(big.NewInt(int64(1))))
fmt.Printf("Test the Solidity Map storage Key1:         0x%x\n", k1)
```

3.4　总结

在本章节中，我们简述了 Go-Ethereum 中 Account 这一关键数据结构的一些具体的实现的细节。了解更多的细节可以参照章节阅读相关部分的源代码。

第 4 章

状态管理 (State Management) i: StateDB

4.1 概述

在本章中，我们来简析一下 go-ethereum 状态管理模块 StateDB。

4.2 理解 StateDB 的结构

我们知道以太坊是是基于以账户为核心的状态机 (State Machine) 的模型。在账户的值发生变化的时候，我们说该账户从一个状态转换到了另一个状态。我们知道，在实际中，每个地址都对应了一个账户。随着以太坊用户和合约数量的增加，如何管理这些账户是客户端开发人员需要解决的首要问题。在 go-ethereum 中，StateDB 模块就是为管理账户状态设计的。它是直接提供了与 `StateObject` (账户和合约的抽象) 相关的 CURD 的接口给其他的模块，比如：

- (这个模块在 Merge 之后被废弃) Mining 模块: 执行新 Block 中的交易时调用 `StateDB` 来更新对应账户的值，并且形成新 world state。
- Block 同步模块，执行新 Block 中的交易时调用 `StateDB` 来更新对应账户的值，并且形成新 world state，同时用这个计算出来的 world state 与 Block Header 中提供的 state root 进行比较，来验证区块的合法性。

- 在 EVM 模块中，调用与合约存储有关的相关的两个 opcode, `sStore` 和 `sSload` 时会调用 `StateDB` 中的函数来查询和更新 Contract 中的持久化存储.
- ...

在实际中，所有的账户数据 (包括当前和历史的状态数据) 最终还是持久化在硬盘中。目前所有的状态数据都是通过 KV 的形式被持久化到了基于 LSM-Tree 的存储引擎中 (例如 go-leveldb)。显然，直接从这种 KV 存储引擎中读取和更新状态数据是不友好的。而 `StateDB` 就是为了操作这些数据而诞生的抽象层。`StateDB` 本质上是一个用于管理所有账户状态的位于内存中的抽象组件。从某种意义上说，我们可以把它理解成一个中间层的内存数据库。

`StateDB` 的定义位于 `core/state/statedb.go` 文件中，如下所示。

```
type StateDB struct {
    db         Database
    prefetcher *triePrefetcher
    trie       Trie
    hasher     crypto.KeccakState

    // originalRoot is the pre-state root, before any changes were made.
    // It will be updated when the Commit is called.
    originalRoot common.Hash

    snaps        *snapshot.Tree
    snap         snapshot.Snapshot
    snapAccounts map[common.Hash][]byte
    snapStorage  map[common.Hash]map[common.Hash][]byte

    // This map holds 'live' objects, which will get modified while processing a state tr
    stateObjects         map[common.Address]*stateObject
    stateObjectsPending  map[common.Address]struct{} // State objects finalized but not y
    stateObjectsDirty    map[common.Address]struct{} // State objects modified in the cur
    stateObjectsDestruct map[common.Address]struct{} // State objects destructed in the b

    // DB error.
    // State objects are used by the consensus core and VM which are
    // unable to deal with database-level errors. Any error that occurs
    // during a database read is memoized here and will eventually be returned
    // by StateDB.Commit.
    dbErr error

    // The refund counter, also used by state transitioning.
    refund uint64

```

```
    thash   common.Hash
    txIndex int
    logs    map[common.Hash][]*types.Log
    logSize uint

    preimages map[common.Hash][]byte

    // Per-transaction access list
    accessList *accessList

    // Transient storage
    transientStorage transientStorage

    // Journal of state modifications. This is the backbone of
    // Snapshot and RevertToSnapshot.
    journal        *journal
    validRevisions []revision
    nextRevisionId int

    // Measurements gathered during execution for debugging purposes
    AccountReads         time.Duration
    AccountHashes        time.Duration
    AccountUpdates       time.Duration
    AccountCommits       time.Duration
    StorageReads         time.Duration
    StorageHashes        time.Duration
    StorageUpdates       time.Duration
    StorageCommits       time.Duration
    SnapshotAccountReads time.Duration
    SnapshotStorageReads time.Duration
    SnapshotCommits      time.Duration
    TrieDBCommits        time.Duration

    AccountUpdated int
    StorageUpdated int
    AccountDeleted int
    StorageDeleted int
}
```

4.2.1 db

StateDB 结构中的第一个变量 db 是一个由 Database 类型定义的。这里的 Database 是一个抽象层的接口类型，它的定义如下所示。我们可以看到在 Database 接口中定义了一些操作更细粒度的数据管理模块的函数。例如 DiskDB() 函数会返回一个更底

层的 key-value disk database 的实例，`TrieDB()` 函数会返回一个指向更底层的 Trie Databse 的实例。这两个模块都是非常重要的管理链上数据的模块。由于这两个模块本身就涉及到了大量的细节，因此我们在此就不对两个模块进行细节分析。在后续的章节中，我们会单独的对这两个模块的实现进行解读。

```
type Database interface {
    // OpenTrie opens the main account trie.
    OpenTrie(root common.Hash) (Trie, error)

    // OpenStorageTrie opens the storage trie of an account.
    OpenStorageTrie(stateRoot common.Hash, addrHash, root common.Hash) (Trie, error)

    // CopyTrie returns an independent copy of the given trie.
    CopyTrie(Trie) Trie

    // ContractCode retrieves a particular contract's code.
    ContractCode(addrHash, codeHash common.Hash) ([]byte, error)

    // ContractCodeSize retrieves a particular contracts code's size.
    ContractCodeSize(addrHash, codeHash common.Hash) (int, error)

    // DiskDB returns the underlying key-value disk database.
    DiskDB() ethdb.KeyValueStore

    // TrieDB retrieves the low level trie database used for data storage.
    TrieDB() *trie.Database
}
```

4.2.2 Trie

这里的 `trie` 变量同样的是由一个 `Trie` 类型的接口定义的。通过这个 `Trie` 类型的接口，上层其他模块就可以通过 `StateDB.tire` 来具体的对 `trie` 的数据进行操作。

```
type Trie interface {
    // GetKey returns the sha3 preimage of a hashed key that was previously used
    // to store a value.
    //
    // TODO(fjl): remove this when StateTrie is removed
    GetKey([]byte) []byte

    // TryGet returns the value for key stored in the trie. The value bytes must
    // not be modified by the caller. If a node was not found in the database, a
    // trie.MissingNodeError is returned.
```

```
    TryGet(key []byte) ([]byte, error)

    // TryGetAccount abstracts an account read from the trie. It retrieves the
    // account blob from the trie with provided account address and decodes it
    // with associated decoding algorithm. If the specified account is not in
    // the trie, nil will be returned. If the trie is corrupted(e.g. some nodes
    // are missing or the account blob is incorrect for decoding), an error will
    // be returned.
    TryGetAccount(address common.Address) (*types.StateAccount, error)

    // TryUpdate associates key with value in the trie. If value has length zero, any
    // existing value is deleted from the trie. The value bytes must not be modified
    // by the caller while they are stored in the trie. If a node was not found in the
    // database, a trie.MissingNodeError is returned.
    TryUpdate(key, value []byte) error

    // TryUpdateAccount abstracts an account write to the trie. It encodes the
    // provided account object with associated algorithm and then updates it
    // in the trie with provided address.
    TryUpdateAccount(address common.Address, account *types.StateAccount) error

    // TryDelete removes any existing value for key from the trie. If a node was not
    // found in the database, a trie.MissingNodeError is returned.
    TryDelete(key []byte) error

    // TryDeleteAccount abstracts an account deletion from the trie.
    TryDeleteAccount(address common.Address) error

    // Hash returns the root hash of the trie. It does not write to the database and
    // can be used even if the trie doesn't have one.
    Hash() common.Hash

    // Commit collects all dirty nodes in the trie and replace them with the
    // corresponding node hash. All collected nodes(including dirty leaves if
    // collectLeaf is true) will be encapsulated into a nodeset for return.
    // The returned nodeset can be nil if the trie is clean(nothing to commit).
    // Once the trie is committed, it's not usable anymore. A new trie must
    // be created with new root and updated trie database for following usage
    Commit(collectLeaf bool) (common.Hash, *trie.NodeSet)

    // NodeIterator returns an iterator that returns nodes of the trie. Iteration
    // starts at the key after the given start key.
    NodeIterator(startKey []byte) trie.NodeIterator

    // Prove constructs a Merkle proof for key. The result contains all encoded nodes
    // on the path to the value at key. The value itself is also included in the last
```

```
57      // node and can be retrieved by verifying the proof.
58      //
59      // If the trie does not contain a value for key, the returned proof contains all
60      // nodes of the longest existing prefix of the key (at least the root), ending
61      // with the node that proves the absence of the key.
62      Prove(key []byte, fromLevel uint, proofDb ethdb.KeyValueWriter) error
63  }
```

4.3　StateDB 的持久化

当新的 Block 被添加到 Blockchain 时，State 的数据并不一会立即被写入到 Disk Database 中。在 `writeBlockWithState` 函数中，函数会判断 `gc` 条件，只有满足一定的条件，才会在此刻调用 `TrieDB` 中的 `Cap` 或者 `Commit` 函数将数据写入 Disk Database 中。

具体 World State 的更新顺序是:

```
StateDB --> Memory_Trie_Database --> LevelDB
```

StateDB 调用 `Commit` 的时候并没有同时触发 `TrieDB` 的 `Commit()`。在 `Block` 被插入到 Blockchain 的这个 Workflow 中，`stateDB` 的 `commit` 首先在 `writeBlockWithState()` 函数中被调用了。之后 `writeBlockWithState()` 函数会判断 `GC` 的状态来决定在本次调用中，是否需要向 `Disk Database` 写入数据。

第 5 章

状态管理 (State Management) ii: State Trie and Storage Trie

写在前面: 在最新的 geth 代码库中，SecureTrie 已经被重命名为了 StateTrie，相关的代码功能也进行了些许调整。因此，为了避免歧义，我们在这里提醒读者 **StateTrie 就是之前的 SecureTrie**。读者在阅读其他的文档时，如果遇到了 SecureTrie, 可以将其理解为 StateTrie。

5.1 理解 Trie 结构

Trie 结构是 Ethereum 中用于管理数据的基本数据结构，它被广泛的运用在 Ethereum 里的多个模块中，包括管理全局的 State Trie，管理 Contract 中持久化存储的 Storage Trie，以及每个 Block 中的与交易相关的 Transaction Trie 和 Receipt Trie。

在以太坊的体系中，广义上的 Trie 的指的是 Merkel Patricia Trie (MPT) 这种数据结构。在实际的实现中，根据业务功能的不同，在 go-ethereum 中一共实现了三种不同的 MPT 的 instance，分别是，Trie，State Trie(Secure Trie) 以及 Stack Trie。由于已经有大量的资料来介绍 MPT 的具体数据结构，在本文中我们就不对 MPT 具体的结构进行解析，感兴趣的读者可以自行搜索相应的资料。

从调用关系上看 `Trie` 是最底层的核心结构，它用于之间负责 StateObject 数据的保存，以及提供相应的 CURD 函数。它的定义在 `trie/trie.go` 文件中。

State Trie 结构本质上是对 Trie 的一层封装。它具体的 CURD 操作的实现都是通过 `Trie` 中定义的函数来执行的。它的定义在 `trie/secure_trie.go` 文件中。目前 StateDB 中的使用的 Trie 是经过封装之后的 State Trie。这个 Trie 也就是我们常说的 State Trie，它是唯一的一个全局 Trie 结构。与 Trie 不同的是，Secure Trie 要求新加入的 Key-Value pair 中的 Key 的数据都是 Sha 函数哈希过的。这是为了方式恶意的构造 Key 来增加 MPT 的高度。

```
type StateTrie struct {
 trie             Trie
 preimages        *preimageStore
 hashKeyBuf       [common.HashLength]byte
 secKeyCache      map[string][]byte
 secKeyCacheOwner *StateTrie // Pointer to self, replace the key cache on mismatch
}
```

不管是 Secure Trie 还是 Trie，他们的创建的前提是: 更下层的 db 的实例已经创建成功了，否则就会报错。值得注意的是一个关键函数 Prove 的实现，并不在这两个 Trie 的定义文件中，而是位于 `trie/proof.go` 文件中。

5.2　Trie 的使用

5.2.1　Read Operation

具体的读取 Trie 上节点的数据是通过 `tryGet()` 函数来实现的。

```
func (t *Trie) tryGet(origNode node, key []byte, pos int) (value []byte, newnode node, di
    switch n := (origNode).(type) {
    case nil:
        return nil, nil, false, nil
    case valueNode:
        return n, n, false, nil
    case *shortNode:
        if len(key)-pos < len(n.Key) || !bytes.Equal(n.Key, key[pos:pos+len(n.Key)]) {
            // key not found in trie
            return nil, n, false, nil
        }
        value, newnode, didResolve, err = t.tryGet(n.Val, key, pos+len(n.Key))
```

```
        if err == nil && didResolve {
            n = n.copy()
            n.Val = newnode
        }
        return value, n, didResolve, err
    case *fullNode:
        value, newnode, didResolve, err = t.tryGet(n.Children[key[pos]], key, pos+1)
        if err == nil && didResolve {
            n = n.copy()
            n.Children[key[pos]] = newnode
        }
        return value, n, didResolve, err
    case hashNode:
        child, err := t.resolveAndTrack(n, key[:pos])
        if err != nil {
            return nil, n, true, err
        }
        value, newnode, _, err := t.tryGet(child, key, pos)
        return value, newnode, true, err
    default:
        panic(fmt.Sprintf("%T: invalid node: %v", origNode, origNode))
    }
}
```

5.2.2 Insert

在 Trie 上插入新节点是通过 `insert()` 函数来实现的。

```
func (t *Trie) insert(n node, prefix, key []byte, value node) (bool, node, error) {
 fmt.Println("Out n:", &n)
 if len(key) == 0 {
  if v, ok := n.(valueNode); ok {
   return !bytes.Equal(v, value.(valueNode)), value, nil
  }
  return true, value, nil
 }
 switch n := n.(type) {
 case *shortNode:
  matchlen := prefixLen(key, n.Key)
  // If the whole key matches, keep this short node as is
  if matchlen == len(n.Key) {
   dirty, nn, err := t.insert(n.Val, append(prefix, key[:matchlen]...), key[matchlen:], v
   if !dirty || err != nil {
    return false, n, err
   }
```

```
    return true, &shortNode{n.Key, nn, t.newFlag()}, nil
   }
   // Otherwise branch out at the index where they differ.
   branch := &fullNode{flags: t.newFlag()}
   var err error
   _, branch.Children[n.Key[matchlen]], err = t.insert(nil, append(prefix, n.Key[:matchlen
   if err != nil {
    return false, nil, err
   }
   _, branch.Children[key[matchlen]], err = t.insert(nil, append(prefix, key[:matchlen+1],
   if err != nil {
    return false, nil, err
   }
   // Replace this shortNode with the branch if it occurs at index 0.
   if matchlen == 0 {
    return true, branch, nil
   }
   // Otherwise, replace it with a short node leading up to the branch.
   return true, &shortNode{key[:matchlen], branch, t.newFlag()}, nil

  case *fullNode:
   dirty, nn, err := t.insert(n.Children[key[0]], append(prefix, key[0]), key[1:], value)
   if !dirty || err != nil {
    return false, n, err
   }
   n = n.copy()
   n.flags = t.newFlag()
   n.Children[key[0]] = nn
   return true, n, nil

  case nil:
   return true, &shortNode{key, value, t.newFlag()}, nil

  case hashNode:
   // We've hit a part of the trie that isn't loaded yet. Load
   // the node and insert into it. This leaves all child nodes on
   // the path to the value in the trie.
   rn, err := t.resolveHash(n, prefix)
   if err != nil {
    return false, nil, err
   }
   dirty, nn, err := t.insert(rn, prefix, key, value)
   if !dirty || err != nil {
    return false, rn, err
   }
   return true, nn, nil
```

```
64
65  default:
66   panic(fmt.Sprintf("%T: invalid node: %v", n, n))
67  }
68 }
```

这里有一个关于 go 语言的知识。我们可以观察到 insert 函数的第一个参数是一个变量名为 n 的 node 类型的变量。有趣的是，在 switch 语句中我们看到了一个这样的写法.

```
1 switch n := n.(type)
```

显然语句两端的 n 的含义并不相同。这种写法在 go 中是合法的。

5.2.3 Finalize And Commit and Commit to Disk

在更底层的 leveldb 中，KV 保存的是 Trie 中的节点，<hash, node.rlprawdata>。在 Geth 中，Trie 并不是实时更新的，而是依赖于 Committer 和 Database 两个额外的辅助组件。

```
Trie.Commit --> Committer.Commit --> trie/Database.insert
```

事实上，由于缓存机制，Trie 的 Commit 并不会真的对 Disk Database 的值进行修改。Trie 真正更新到 Disk Database 的，是依赖于 `trie/Database.Commit` 函数的调用。我们可以在诸多函数中找到这个函数的调用比如。

```
1  func GenerateChain(config *params.ChainConfig, parent *types.Block, engine consensus.Engi
2    ...
3    // Write state changes to db
4    root, err := statedb.Commit(config.IsEIP158(b.header.Number))
5    if err != nil {
6     panic(fmt.Sprintf("state write error: %v", err))
7    }
8    if err := statedb.Database().TrieDB().Commit(root, false, nil); err != nil {
9     panic(fmt.Sprintf("trie write error: %v", err))
10   }
11   ...
12 }
```

5.2.3.1 State Trie 的更新是什么时候发生的？

State Trie 的更新，通常是指的是基于 State Trie 中节点值的变化从而重新计算 State Trie 的 Root 的 Hash 值的过程。目前这一过程是通过调用 StateDB 中的 `IntermediateRoot` 函数来完成的。

我们从三个粒度层面来看待 State Trie 更新的问题。

- Block 层。在一个新的 Block Insert 到 Blockchain 的过程中，State Trie 可能会发生多次的更新。比如，在每次 Transaction 被执行之后，`IntermediateRoot` 函数都会被调用。同时，更新后的 State Trie 的 Root 值，会被写入到 Transaction 对应的 Receipt 中。请注意，在调用 `IntermediateRoot` 函数时，更新后的值在此时并没有被立刻写入到 Disk Database 中。此时的 State Trie Root 只是基于内存中的数据计算出来的。真正的 Trie 数据写盘，需要等到 `trieDB.Commit` 函数的执行。
- Transaction 层。如上面提到的，在每次 Transaction 执行完成后，系统都会调用一次 StateDB 的 `IntermediateRoot` 函数，来更新 State Trie。并且会将更新后的 Trie 的 Root Hash 写入到该 Transaction 对应的 Receipt 中。这里提一下关于 `IntermediateRoot` 函数细节。在 IntermediateRoot'函数调用时，会首先更新被修改的 Contract 的 Storage Trie 的 Root。
- Instruction 层。执行 Contract 的 Instruction，并不会直接的引发 State Trie 的更新。比如，我们知道，EVM 指令 `OpSstore` 会修改 Contract 中的持久化存储。这个指令调用了 StateDB 中的 `SetState` 函数，并最终调用了对应的 StateObject 中的 `setState` 函数。StateObject 中的 `setState` 函数并没有直接对 Contract 的 Storage Trie 进行更新，而是将修改的存储对象保存在了 StateObject 中的 *dirtyStorage* 中 (*dirtyStorage* 是用于缓存 Storage Slot 数据的 Key-Value Map). Storage Trie 的更新是由更上层的函数调用所触发的，比如 `IntermediateRoot` 函数，以及 `StateDB.Commit` 函数。

5.3 StackTrie

`StackTrie` 虽然也是 `MPT` 结构，但是相比于作为索引结构来管理数据，`StackTrie` 更直接的一个用法是给一组数据生成证明。例如，在 `Block` 中的 `Transaction Hash` 以及 `Receipt Hash` 都是基于 `StackTrie` 生成的。这里我们使用一个更直观的例子。

这个部分的代码位于 core/block_validator.go 中。在 block_validator 中定义了一系列验证用的函数, 比如 ValidateBody() 和 ValidateState() 函数。我们选取了这两个函数的其中一部分，如下所示。为了验证 Block 的合法性，ValidateBody() 和 ValidateState() 函数分别在本地基于 Block 中提供的数据来构造 Transaction 和 Receipt 的哈希来与 Header 中的 TxHash 与 ReceiptHash。我们可以发现，函数 types.DeriveSha 需要一个 TrieHasher 类型的参数。但是在具体调用的时候，却传入了了一个 trie.NewStackTrie 类型的变量。这是因为 StackTrie 实现了 TrieHasher 接口所需要的三个函数，所以这种调用是合法的。我们可以在 core/types/hashing.go 中找到 TrieHasher 的定义。这里 DeriveSha 不断的向 StackTrie 中添加数据，并最终返回 StackTrie 的 Root 哈希值用作数据证明。

```
func (v *BlockValidator) ValidateBody(block *types.Block) error {
 ...
 if hash := types.DeriveSha(block.Transactions(), trie.NewStackTrie(nil)); hash != header
  return fmt.Errorf("transaction root hash mismatch: have %x, want %x", hash, header.TxHa
 }
 ...
}
```

```
func (v *BlockValidator) ValidateState(block *types.Block, statedb *state.StateDB, receip
 ...
 // Tre receipt Trie's root (R = (Tr [[H1, R1], ... [Hn, Rn]]))
 receiptSha := types.DeriveSha(receipts, trie.NewStackTrie(nil))
 if receiptSha != header.ReceiptHash {
  return fmt.Errorf("invalid receipt root hash (remote: %x local: %x)", header.ReceiptHas
 }
 ...
}
```

同时，我们可以发现，在调用 DeriveSha 函数的时候，我们每次都会 new 一个新的 StackTrie 作为参数。这也反映出了，在这个部分 StackTrie 的主要作用就是生成验证用的 Proof，而不是像管理账户状态的 State Trie 一样唯一存在。

第 6 章

交易 (Transaction)

6.1 概述

前面的章节中，我们简述了一下 Account/Contract 的基本数据结构。在本章我们就来探索一下，Ethereum 中的一个基本数据结构 Transaction。在本文中，我们提到的交易指的是在 Ethereum Layer-1 层面上构造的交易。

首先，Transaction 是 Ethereum 执行数据操作的媒介，它主要起到下面的几个作用:

1. 在 Layer-1 网络上的 Account 之间进行 Native Token 的转账。
2. 创建新的 Contract。
3. 调用 Contract 中会修改目标 Contract 中持久化数据或者间接修改其他 Account/Contract 数据的函数。

这里我们对 Transaction 的功能性的细节再进行一些额外的补充。首先，Transaction 只能创建合约 (Contract) 账户，而不能用于创建外部账户 (EOA)。第二，如果调用 Contract 中的只读函数，是不需要构造 Transaction 的。相对的，所有参与 Account/Contract 数据修改的操作都需要通过 Transaction 来进行。第三，广义上的 Transaction 只能由外部账户 (EOA) 构建。Contract 是没有办法显式构造 Layer-1 层面的交易的。在某些合约函数的执行过程中，Contract 在可以通过构造 internal transaction 来与其他的合约进行交互，但是这种 Internal transaction 与我们提到的 Layer-1 层面的交易有所不同，我们会在之后的章节介绍。

6.2 LegacyTx & AccessListTX & DynamicFeeTx

下面我们根据源代码中的定义来了解一下 Transaction 的数据结构。Transaction 结构体的定义位于 *core/types/transaction.go* 中。Transaction 的结构体如下所示。

```
type Transaction struct {
 inner TxData    // Consensus contents of a transaction
 time  time.Time // Time first seen locally (spam avoidance)

 // caches
 hash atomic.Value
 size atomic.Value
 from atomic.Value
}
```

从代码定义中我们可以看到，`Transaction` 的结构体是非常简单的，它只包含了五个变量分别是, `TxData` 类型的 inner，`Time` 类型的 time，以及三个 `atomic.Value` 类型的 hash，size，以及 from。这里我们需要重点关注一下 `inner` 这个变量。目前与 Transaction 直接相关的数据都由这个变量来维护。

目前，`TxData` 类型是一个接口，它的定义如下面的代码所示。

```
type TxData interface {
 txType() byte // returns the type ID
 copy() TxData // creates a deep copy and initializes all fields

 chainID() *big.Int
 accessList() AccessList
 data() []byte
 gas() uint64
 gasPrice() *big.Int
 gasTipCap() *big.Int
 gasFeeCap() *big.Int
 value() *big.Int
 nonce() uint64
 to() *common.Address

 rawSignatureValues() (v, r, s *big.Int)
 setSignatureValues(chainID, v, r, s *big.Int)
}
```

这里注意，在目前版本的 geth 中 (1.10.*)，根据 [EIP-2718][EIP2718] 的设

计，原来的 TxData 现在被声明成了一个 interface，而不是定义了具体的结构。这样的设计好处在于，后续版本的更新中可以对 Transaction 类型进行更加灵活的修改。目前，在 Ethereum 中定义了三种类型的 Transaction 来实现 TxData 这个接口。按照时间上的定义顺序来说，这三种类型的 Transaction 分别是，LegacyT，AccessListTx，TxDynamicFeeTx。LegacyTx 顾名思义，是原始的 Ethereum 的 Transaction 设计，目前市面上大部分早年关于 Ethereum Transaction 结构的文档实际上都是在描述 LegacyTx 的结构。而 AccessListTX 是基于 EIP-2930(Berlin 分叉) 的 Transaction。DynamicFeeTx 是EIP-1559(伦敦分叉) 生效之后的默认的 Transaction。

(PS: 目前 Ethereum 的黄皮书只更新到了 Berlin 分叉的内容，还没有添加 London 分叉的更新, 2022.3.10)

6.2.1 LegacyTx

LegacyTx 是最原始的以太坊交易的定义。

```
type LegacyTx struct {
 Nonce    uint64          // nonce of sender account
 GasPrice *big.Int        // wei per gas
 Gas      uint64          // gas limit
 To       *common.Address `rlp:"nil"` // nil means contract creation
 Value    *big.Int        // wei amount
 Data     []byte          // contract invocation input data
 V, R, S  *big.Int        // signature values
}
```

6.2.2 AccessListTX

AccessListTx 在 LegacyTx 基础上多了 `ChainID` 和 `AccessList` 这两个变量。

```
type AccessListTx struct {
 ChainID    *big.Int        // destination chain ID
 Nonce      uint64          // nonce of sender account
 GasPrice   *big.Int        // wei per gas
 Gas        uint64          // gas limit
 To         *common.Address `rlp:"nil"` // nil means contract creation
 Value      *big.Int        // wei amount
 Data       []byte          // contract invocation input data
 AccessList AccessList      // EIP-2930 access list
 V, R, S    *big.Int        // signature values
}
```

6.2.3 DynamicFeeTx

如果我们观察 DynamicFeeTx 就会发现，DynamicFeeTx 的定义其实就是在 LegacyTx/AccessListTX 的定义的基础上额外的增加了 GasTipCap 与 GasFeeCap 这两个字段。

```
type DynamicFeeTx struct {
 ChainID    *big.Int
 Nonce      uint64
 GasTipCap  *big.Int // a.k.a. maxPriorityFeePerGas
 GasFeeCap  *big.Int // a.k.a. maxFeePerGas
 Gas        uint64
 To         *common.Address `rlp:"nil"` // nil means contract creation
 Value      *big.Int
 Data       []byte
 AccessList AccessList

 // Signature values
 V *big.Int `json:"v" gencodec:"required"`
 R *big.Int `json:"r" gencodec:"required"`
 S *big.Int `json:"s" gencodec:"required"`
}
```

6.3 Transaction 的执行

Transaction 的执行主要在发生在两个 Workflow 中:

1. Miner 在打包新的 Block 时。此时 Miner 会按 Block 中 Transaction 的打包顺序来执行其中的 Transaction。
2. 其他节点添加 Block 到 Blockchain 时。当节点从网络中监听并获取到新的 Block 时，它们会执行 Block 中的 Transaction，来更新本地的 State Trie 的 Root，并与 Block Header 中的 State Trie Root 进行比较，来验证 Block 的合法性。

一条 Transaction 执行，可能会涉及到多个 Account/Contract 的值的变化，最终造成一个或多个 Account 的 State 的发生转移。在 Byzantium 分叉之前的 Geth 版本中，在每个 Transaction 执行之后，都会计算一个当前的 State Trie Root，并写入到对应的 Transaction Receipt 中。这符合以太坊黄皮书中的原始设计。即交易是使

得 Ethereum 状态机发生状态状态转移的最细粒度单位。读者们可能已经来开产生疑惑了，“每个 Transaction 都会重算一个 State Trie Root” 的方式岂不是会带来大量的计算 (重算一次一个 MPT Path 上的所有 Node) 和读写开销 (新生成的 MPT Node 是很有可能最终被持久化到 LevelDB 中的)？结论是显然的。因此在 Byzantium 分叉之后，在一个 Block 的验证周期中只会计算一次的 State Root。我们仍然可以在 `state_processor.go` 找寻到早年代码的痕迹。最终，一个 Block 中所有 Transaction 执行的结果使得 World State 发生状态转移。下面我们就来根据 geth 代码库中的调用关系，从 Miner 的视角来探索一个 Transaction 的生命周期。

6.3.1　Transaction 修改 Contract 的持久化存储的

在 Ethereum 中，当 Miner 开始构造新的区块的时候，首先会启动 *miner/worker.go* 的 `mainLoop()` 函数。具体的函数如下所示。

```
func (w *worker) mainLoop() {
    ....
    // 设置接受该区块中挖矿奖励的账户地址
    coinbase := w.coinbase
    w.mu.RUnlock()

    txs := make(map[common.Address]types.Transactions)
    for _, tx := range ev.Txs {
        acc, _ := types.Sender(w.current.signer, tx)
        txs[acc] = append(txs[acc], tx)
    }
    // 这里看到，通过 NewTransactionsByPriceAndNonce 获取一部分的 Tx 并打包
    txset := types.NewTransactionsByPriceAndNonce(w.current.signer, txs, w.current.header
    tcount := w.current.tcount
    //提交打包任务
    w.commitTransactions(txset, coinbase, nil)
    ....
}
```

在 Mining 新区块前，Worker 首先需要决定，哪些 Transaction 会被打包到新的 Block 中。这里选取 Transaction 其实经历了两个步骤。首先，`txs` 变量保存了从 Transaction Pool 中拿去到的合法的，以及准备好被打包的交易。这里举一个例子，来说明什么是**准备好被打包的交易**，比如 Alice 先后发了新三个交易到网络中，对应的 Nonce 分别是 100 和 101，102。假如 Miner 只收到了 100 和 102 号交易。那么对于此刻的 Transaction Pool 来说 Nonce 100 的交易就是**准备好被打包的交易**，交易 Nonce 是 102 需要等待 Nonce 101 的交易被确认之后才能提交。

在 Worker 会从 Transaction Pool 中拿出若干的 transaction, 赋值给 *txs* 之后, 然后调用 `NewTransactionsByPriceAndNonce` 函数按照 Gas Price 和 Nonce 对 *txs* 进行排序，并将结果赋值给 *txset*。此外在 Worker 的实例中，还存在 `fillTransactions` 函数，为了未来定制化的给 Transaction 的执行顺序进行排序。

在拿到 *txset* 之后，mainLoop 函数会调用 `commitTransactions` 函数，正式进入 Mining 新区块的流程。`commitTransactions` 函数如下所示。

```
func (w *worker) commitTransactions(txs *types.TransactionsByPriceAndNonce, coinbase comm
    ....

    // 首先给 Block 设置最大可以使用的 Gas 的上限
    gasLimit := w.current.header.GasLimit
    if w.current.gasPool == nil {
    w.current.gasPool = new(core.GasPool).AddGas(gasLimit)
    // 函数的主体是一个 For 循环
    for{
    .....
        // params.TxGas 表示了 transaction 需要的最少的 Gas 的数量
        // w.current.gasPool.Gas() 可以获取当前 block 剩余可以用的 Gas 的 Quota，如果剩余的
        if w.current.gasPool.Gas() < params.TxGas {
            log.Trace("Not enough gas for further transactions", "have", w.current.gasPoo
    }
    ....
    tx := txs.Peek()
    if tx == nil {
        break
    }
    ....
    // 提交单条 Transaction 进行验证
    logs, err := w.commitTransaction(tx, coinbase)
    ....
    }
}
```

`commitTransactions` 函数的主体是一个 for 循环，每次获取结构体切片头部的 txs.Peek() 的 transaction，并作为参数调用函数 miner/worker.go 的 `commitTransaction()`。`commitTransaction()` 函数如下所示。

```
func (w *worker) commitTransaction(tx *types.Transaction, coinbase common.Address) ([]*ty
    // 在每次 commitTransaction 执行前都要记录当前 StateDB 的 Snapshot, 一旦交易执行失败则
    // TODO StateDB 如何进行快照 (Snapshot) 和回滚的
    snap := w.current.state.Snapshot()
    // 调用执行 Transaction 的函数
    receipt, err := core.ApplyTransaction(w.chainConfig, w.chain, &coinbase, w.current.ga
```

```
    ....
}
```

Blockchain 系统中的 Transaction 和 DBMS 中的 Transaction 一样，要么完成要么失败。所以在调用执行 Transaction 的函数前，首先记录了一下当前 world state 的 Snapshot，用于交易失败时回滚操作。之后调用 core/state_processor.go/ApplyTransaction() 函数。

```
func ApplyTransaction(config *params.ChainConfig, bc ChainContext, author *common.Address
    // 将 Transaction 转化为 Message 的形式
    msg, err := tx.AsMessage(types.MakeSigner(config, header.Number), header.BaseFee)
    if err != nil {
        return nil, err
    }
    // Create a new context to be used in the EVM environment
    blockContext := NewEVMBlockContext(header, bc, author)
    vmenv := vm.NewEVM(blockContext, vm.TxContext{}, statedb, config, cfg)
    // 调用执行 Contract 的函数
    return applyTransaction(msg, config, bc, author, gp, statedb, header.Number, header.H
}
```

在 ApplyTransaction() 函数中首先 Transaction 会被转换成 Message 的形式。在执行每一个 Transaction 的时候，都会生成一个新的 EVM 来执行。之后调用 core/state_processor.go/applyTransaction() 函数来执行 Message。

```
func applyTransaction(msg types.Message, config *params.ChainConfig, bc ChainContext, aut
    ....
    // Apply the transaction to the current state (included in the env).
    result, err := ApplyMessage(evm, msg, gp)
    ....

}
```

之后调用 core/state_transition.go/ApplyMessage() 函数。

```
func ApplyMessage(evm *vm.EVM, msg Message, gp *GasPool) (*ExecutionResult, error) {
    return NewStateTransition(evm, msg, gp).TransitionDb()
    }
```

之后调用 core/state_transition.go/TransitionDb() 函数。

```
func (st *StateTransition) TransitionDb() (*ExecutionResult, error) {
    ....
    ret, st.gas, vmerr = st.evm.Call(sender, st.to(), st.data, st.gas, st.value)
    ....
}
```

之后调用 core/vm/evm.go/Call() 函数。

```
func (evm *EVM) Call(caller ContractRef, addr common.Address, input []byte, gas uint64, v
    ....
    // Execute the contract
    ret, err = evm.interpreter.Run(contract, input, false)
    ....
}
```

之后调用 core/vm/interpreter.go/Run() 函数。

```
// Run loops and evaluates the contract's code with the given input data and returns
// the return byte-slice and an error if one occurred.
func (in *EVMInterpreter) Run(contract *Contract, input []byte, readOnly bool) (ret []byt
    ....
    cost = operation.constantGas // For tracing
    // UseGas 函数：当前剩余的 gas quota 减去 input 参数。
    // 剩余的 gas 小于 input 直接返回 false
    // 否则当前的 gas quota 减去 input 并返回 true
    if !contract.UseGas(operation.constantGas) {
        return nil, ErrOutOfGas
        }
    ....
    // execute the operation
    res, err = operation.execute(&pc, in, callContext)
    ....

}
```

在更细粒度的层面，每个 opcode 循环调用 core/vm/jump_table.go 中的 execute 函数。这里值得一提的是，获取 Contract 中每条 Operate 的方式，是从 Contact 中的 code 数组中按照第 n 个拿取。

```
// GetOp returns the n'th element in the contract's byte array
func (c *Contract) GetOp(n uint64) OpCode {
 return OpCode(c.GetByte(n))
}

// GetByte returns the n'th byte in the contract's byte array
func (c *Contract) GetByte(n uint64) byte {
 if n < uint64(len(c.Code)) {
  return c.Code[n]
 }

 return 0
}
```

OPCODE 的具体实现代码位于 core/vm/instructor.go 文件中。比如，对 Contract 中持久化数据修改的 OPSSTORE 指令的实现位于 opStore() 函数中。而 opStore 的函数的具体操作又是调用了 StateDB 中的 SetState 函数，将 Go-ethereum 中的几个主要的模块串联了起来。

```
func opSstore(pc *uint64, interpreter *EVMInterpreter, scope *ScopeContext) ([]byte, erro
    loc := scope.Stack.pop()
    val := scope.Stack.pop()
    //根据指令跟地址来修改 StateDB 中某一存储位置的值。
    interpreter.evm.StateDB.SetState(scope.Contract.Address(),loc.Bytes32(), val.Bytes32(
    return nil, nil
}

//core/state/stateDB
func (s *StateDB) SetState(addr common.Address, key, value common.Hash) {
    stateObject := s.GetOrNewStateObject(addr)
    if stateObject != nil {
    stateObject.SetState(s.db, key, value)
    }
}

对于一条调用合约函数的交易，其中必然会存在修改 StateDB 的操作。通过上述的函数调用关系，我们就

![Transaction Execution Flow](../figs/02/tx_execu_flow.png)

![Transaction Execution stack Flow](../figs/04/tx_exec_calls.png)


## 验证节点是如何执行 Transaction 来更新 World State

而对于不参与 Mining 的节点，他们执行 Block 中 Transaction 的入口是在 core/blockchain.go 中的

```go
// Process processes the state changes according to the Ethereum rules by running
// the transaction messages using the statedb and applying any rewards to both
// the processor (coinbase) and any included uncles.
//
// Process returns the receipts and logs accumulated during the process and
// returns the amount of gas that was used in the process. If any of the
// transactions failed to execute due to insufficient gas it will return an error.
func (p *StateProcessor) Process(block *types.Block, statedb *state.StateDB, cfg vm.Confi
 var (
  receipts    types.Receipts
  usedGas     = new(uint64)
  header      = block.Header()
```
```

```
 blockHash   = block.Hash()
 blockNumber = block.Number()
 allLogs     []*types.Log
 gp          = new(GasPool).AddGas(block.GasLimit())
)
// Mutate the block and state according to any hard-fork specs
if p.config.DAOForkSupport && p.config.DAOForkBlock != nil && p.config.DAOForkBlock.Cmp(
 misc.ApplyDAOHardFork(statedb)
}
blockContext := NewEVMBlockContext(header, p.bc, nil)
vmenv := vm.NewEVM(blockContext, vm.TxContext{}, statedb, p.config, cfg)
// Iterate over and process the individual transactions
for i, tx := range block.Transactions() {
 msg, err := tx.AsMessage(types.MakeSigner(p.config, header.Number), header.BaseFee)
 if err != nil {
  return nil, nil, 0, fmt.Errorf("could not apply tx %d [%v]: %w", i, tx.Hash().Hex(), e
 }
 statedb.Prepare(tx.Hash(), i)
 //核心: 与 Mining 中 Commit Transaction 不同，Process 在外部循环 Block 中的 Transaction
 receipt, err := applyTransaction(msg, p.config, p.bc, nil, gp, statedb, blockNumber, bl
 if err != nil {
  return nil, nil, 0, fmt.Errorf("could not apply tx %d [%v]: %w", i, tx.Hash().Hex(), e
 }
 receipts = append(receipts, receipt)
 allLogs = append(allLogs, receipt.Logs...)
}
// Finalize the block, applying any consensus engine specific extras (e.g. block rewards
p.engine.Finalize(p.bc, header, statedb, block.Transactions(), block.Uncles())

return receipts, allLogs, *usedGas, nil
}
```

6.3.2 Background of State-based Blockchain

- State-based Blockchain 的数据主要由两部分的数据管理模块组成：World State 和 Blockchain。
- State Object 是系统中基于 K-V 结构的基础数据元素。在 Ethereum 中，State Object 是 Account。
- World State 表示了 System 中所有 State Object 的最新值的一个 Snapshot，。
- Blockchain 是以块为单位的数据结构，每个块中包含了若干 Transaction。Blockchain 可以被视为历史交易数据的组合。

- Transaction 是 Blockchain System 中与承载数据更新的载体。通过 Transaction，State Object 从当前状态切换到另一个状态。
- World State 的更新是以 Block 为单位的。

6.3.3 Read Transaction from Database

当我们想要通过 Transaction 的 Hash 查询一个 Transaction 具体的数据的时候，上层的 API 会调用 `eth/api_backend.go` 中的 `GetTransaction()` 函数，并最终调用了 `core/rawdb/accessors_indexes.go` 中的 `ReadTransaction()` 函数来查询。

```
func (b *EthAPIBackend) GetTransaction(ctx context.Context, txHash common.Hash) (*types.T
 tx, blockHash, blockNumber, index := rawdb.ReadTransaction(b.eth.ChainDb(), txHash)
 return tx, blockHash, blockNumber, index, nil
}
```

这里值得注意的是，在读取 Transaction 的时候，`ReadTransaction()` 函数首先获取了保存该 Transaction 的函数 Block body，并循环遍历该 Block Body 中获取到对应的 Transaction。这是因为，虽然 Transaction 是作为一个基本的数据结构 (Transaction Hash 可以保证 Transaction 的唯一性)，但是在写入数据库的时候就是被按照 Block Body 的形式被整个的打包写入到 Database 中的。具体的代码逻辑可以查看 `core/rawdb/accesssor_chain.go` 中的 `WriteBlock()` 和 `WriteBody()` 函数。

```
func ReadTransaction(db ethdb.Reader, hash common.Hash) (*types.Transaction, common.Hash,
 blockNumber := ReadTxLookupEntry(db, hash)
 if blockNumber == nil {
  return nil, common.Hash{}, 0, 0
 }
 blockHash := ReadCanonicalHash(db, *blockNumber)
 if blockHash == (common.Hash{}) {
  return nil, common.Hash{}, 0, 0
 }
 body := ReadBody(db, blockHash, *blockNumber)
 if body == nil {
  log.Error("Transaction referenced missing", "number", *blockNumber, "hash", blockHash)
  return nil, common.Hash{}, 0, 0
 }
 for txIndex, tx := range body.Transactions {
  if tx.Hash() == hash {
   return tx, blockHash, *blockNumber, uint64(txIndex)
  }
 }
 log.Error("Transaction not found", "number", *blockNumber, "hash", blockHash, "txhash",
```

```
21  return nil, common.Hash{}, 0, 0
22  }
```

第 7 章

区块和区块链 (Block & Blockchain)

7.1 Block

7.1.1 基础数据结构

```
type Block struct {
 header       *Header
 uncles       []*Header
 transactions Transactions
 hash atomic.Value
 size atomic.Value
 td *big.Int
 ReceivedAt   time.Time
 ReceivedFrom interface{}
}
```

```
type Header struct {
 ParentHash  common.Hash    `json:"parentHash"       gencodec:"required"`
 UncleHash   common.Hash    `json:"sha3Uncles"       gencodec:"required"`
 Coinbase    common.Address `json:"miner"            gencodec:"required"`
 Root        common.Hash    `json:"stateRoot"        gencodec:"required"`
 TxHash      common.Hash    `json:"transactionsRoot" gencodec:"required"`
 ReceiptHash common.Hash    `json:"receiptsRoot"     gencodec:"required"`
 Bloom       Bloom          `json:"logsBloom"        gencodec:"required"`
 Difficulty  *big.Int       `json:"difficulty"       gencodec:"required"`
```

```
 Number      *big.Int       `json:"number"           gencodec:"required"`
 GasLimit    uint64         `json:"gasLimit"         gencodec:"required"`
 GasUsed     uint64         `json:"gasUsed"          gencodec:"required"`
 Time        uint64         `json:"timestamp"        gencodec:"required"`
 Extra       []byte         `json:"extraData"        gencodec:"required"`
 MixDigest   common.Hash    `json:"mixHash"`
 Nonce       BlockNonce     `json:"nonce"`
 // BaseFee was added by EIP-1559 and is ignored in legacy headers.
 BaseFee *big.Int `json:"baseFeePerGas" rlp:"optional"`
}
```

7.2 Blockchain

7.2.1 基础数据结构

```
type BlockChain struct {
 chainConfig *params.ChainConfig // Chain & network configuration
 cacheConfig *CacheConfig        // Cache configuration for pruning

 db     ethdb.Database // Low level persistent database to store final content in
 snaps  *snapshot.Tree // Snapshot tree for fast trie leaf access
 triegc *prque.Prque   // Priority queue mapping block numbers to tries to gc
 gcproc time.Duration  // Accumulates canonical block processing for trie dumping

 // txLookupLimit is the maximum number of blocks from head whose tx indices
 // are reserved:
 //  * 0:   means no limit and regenerate any missing indexes
 //  * N:   means N block limit [HEAD-N+1, HEAD] and delete extra indexes
 //  * nil: disable tx reindexer/deleter, but still index new blocks
 txLookupLimit uint64

 hc            *HeaderChain
 rmLogsFeed    event.Feed
 chainFeed     event.Feed
 chainSideFeed event.Feed
 chainHeadFeed event.Feed
 logsFeed      event.Feed
 blockProcFeed event.Feed
 scope         event.SubscriptionScope
 genesisBlock  *types.Block

 // This mutex synchronizes chain write operations.
```

```
28  // Readers don't need to take it, they can just read the database.
29  chainmu *syncx.ClosableMutex
30
31  currentBlock     atomic.Value // Current head of the block chain
32  currentFastBlock atomic.Value // Current head of the fast-sync chain (may be above the b
33
34  stateCache    state.Database // State database to reuse between imports (contains state
35  bodyCache     *lru.Cache     // Cache for the most recent block bodies
36  bodyRLPCache  *lru.Cache     // Cache for the most recent block bodies in RLP encoded fo
37  receiptsCache *lru.Cache     // Cache for the most recent receipts per block
38  blockCache    *lru.Cache     // Cache for the most recent entire blocks
39  txLookupCache *lru.Cache     // Cache for the most recent transaction lookup data.
40  futureBlocks  *lru.Cache     // future blocks are blocks added for later processing
41
42  wg            sync.WaitGroup //
43  quit          chan struct{}  // shutdown signal, closed in Stop.
44  running       int32          // 0 if chain is running, 1 when stopped
45  procInterrupt int32          // interrupt signaler for block processing
46
47  engine     consensus.Engine
48  validator  Validator // Block and state validator interface
49  prefetcher Prefetcher
50  processor  Processor // Block transaction processor interface
51  forker     *ForkChoice
52  vmConfig   vm.Config
53 }
```

第 8 章

交易和区块的同步

在本章中，我们会像哲学家一样思考：数据结构/实例/对象/变量是从哪里来，又要到哪里去呢？

8.1 概述

在前面的章节中，我们已经讨论了在以太坊中 Transactions 是从 Transaction Pool 中，被 Validator/Miner 们验证打包，最终被保存在区块链中。那么，接下来的问题是，Transaction 是怎么被进入到 Transaction Pool 中的呢？基于同样的思考方式，那么一个刚刚在某个节点被打包好的 Block，它又将怎么传输到区块链网络中的其他节点那里，并最终实现 Blockchain 长度是一致的呢？在本章中，我们就来探索一下，节点是如何发送和接收 Transaction 和 Block 的。

8.2 geth 节点是如何同步交易的？

在前面的章节中，我们曾经提到，Geth 节点中最顶级的对象是 Node 类型，负责节点最高级别生命周期相关的操作，例如节点的启动以及关闭，节点数据库的打开和关闭，启动 RPC 监听。而更具体的管理业务生命周期 (lifecycle) 的函数，都是由后端 Service 实例 `Ethereum` 和 `LesEthereum` 来实现的。

定义在 `eth/backend.go` 中的 `Ethereum` 提供了一个全节点的所有的服务包括：TxPool 交易池，Miner 模块，共识模块，API 服务，以及解析从 P2P 网络中获取的数

据。LesEthereum 提供了轻节点对应的服务。由于轻节点所支持的功能相对较少，在这里我们不过多描述。Ethereum 结构体的定义如下所示。

```
type Ethereum struct {
    config *ethconfig.Config
    txPool     *txpool.TxPool
    blockchain *core.BlockChain
    handler            *handler // 我们关注的核心对象
    ethDialCandidates  enode.Iterator
    snapDialCandidates enode.Iterator
    merger             *consensus.Merger
    chainDb ethdb.Database // Block chain database
    eventMux       *event.TypeMux
    engine         consensus.Engine
    accountManager *accounts.Manager
    bloomRequests     chan chan *bloombits.Retrieval // Channel receiving bloom data retr
    bloomIndexer      *core.ChainIndexer             // Bloom indexer operating during bl
    closeBloomHandler chan struct{}
    APIBackend *EthAPIBackend
    miner     *miner.Miner
    gasPrice  *big.Int
    etherbase common.Address
    networkID     uint64
    netRPCService *ethapi.NetAPI
    p2pServer *p2p.Server
    lock sync.RWMutex // Protects the variadic fields (e.g. gas price and etherbase)
    shutdownTracker *shutdowncheck.ShutdownTracker // Tracks if and when the node has shu
}
```

这里值得提醒一下，在 Geth 代码中，不少地方都使用 backend 这个变量名，来指代 Ethereum。但是，也存在一些代码中使用 backend 来指代 ethapi.Backend 接口。在这里，我们可以做一下区分，Ethereum 负责维护节点后端的生命周期的函数，例如 Miner 的开启与关闭。而 ethapi.Backend 接口主要是提供对外的业务接口，例如查询区块和交易的状态。读者可以根据上下文来判断 backend 具体指代的对象。我们在 geth 是如何启动的一章中提到，Ethereum 是在 Geth 启动的实例化的。在实例化 Ethereum 的过程中，就会创建一个 APIBackend *EthAPIBackend 的成员变量，它就是 ethapi.Backend 接口类型的。

第 9 章

交易池的设计与实现

9.1 概述

交易可以分为 Local Transaction 和 Remote Transaction 两种。通过节点提供的 RPC 传入的交易，被划分为 Local Transaction，通过 P2P 网络传给节点的交易被划分为 Remote Transaction。

9.2 交易池的基本结构

Transaction Pool 主要有两个内存组件，`Pending` 和 `Queue` 组成。具体的定义如下所示。

```
type TxPool struct {
    config      Config
    chainconfig *params.ChainConfig
    chain       blockChain
    gasPrice    *big.Int
    txFeed      event.Feed
    scope       event.SubscriptionScope
    signer      types.Signer
    mu          sync.RWMutex

    istanbul bool // Fork indicator whether we are in the istanbul stage.
    eip2718  bool // Fork indicator whether we are using EIP-2718 type transactions.
    eip1559  bool // Fork indicator whether we are using EIP-1559 type transactions.
    shanghai bool // Fork indicator whether we are in the Shanghai stage.
```

```
15
16      currentState  *state.StateDB // Current state in the blockchain head
17      pendingNonces *noncer        // Pending state tracking virtual nonces
18      currentMaxGas uint64         // Current gas limit for transaction caps
19
20      locals  *accountSet // Set of local transaction to exempt from eviction rules
21      journal *journal    // Journal of local transaction to back up to disk
22
23      pending map[common.Address]*list     // All currently processable transactions
24      queue   map[common.Address]*list     // Queued but non-processable transactions
25      beats   map[common.Address]time.Time // Last heartbeat from each known account
26      all     *lookup                      // All transactions to allow lookups
27      priced  *pricedList                  // All transactions sorted by price
28
29      chainHeadCh     chan core.ChainHeadEvent
30      chainHeadSub    event.Subscription
31      reqResetCh      chan *txpoolResetRequest
32      reqPromoteCh    chan *accountSet
33      queueTxEventCh  chan *types.Transaction
34      reorgDoneCh     chan chan struct{}
35      reorgShutdownCh chan struct{}  // requests shutdown of scheduleReorgLoop
36      wg              sync.WaitGroup // tracks loop, scheduleReorgLoop
37      initDoneCh      chan struct{}  // is closed once the pool is initialized (for tests)
38
39      changesSinceReorg int // A counter for how many drops we've performed in-between reor
40  }
```

9.3　交易池的限制

交易池设置了一些的参数来限制单个交易的 Size ，以及整个 Transaction Pool 中所保存的交易的总数量。当交易池的中维护的交易超过某个阈值的时候，交易池会丢弃/驱逐 (Discard/Evict) 一部分的交易。这里注意，被清除的交易通常都是 Remote Transaction，而 Local Transaction 通常都会被保留下来。

负责判断哪些交易会被丢弃的函数是 `txPricedList.Discard()`。

Transaction Pool 引入了一个名为 `txSlotSize` 的 Metrics 作为计量一个交易在交易池中所占的空间。目前，`txSlotSize` 的值是 `32 * 1024`。每个交易至少占据一个 `txSlot`，最大能占用四个 `txSlotSize`，`txMaxSize = 4 * txSlotSize = 128 KB`。换句话说，如果一个交易的物理数据大小不足 32KB，那么它会占据一个 `txSlot`。同时，一个合法的交易最大是 128KB 的大小

按照默认的设置，交易池的最多保存 `4096+1024` 个交易，其中 Pending 区保存 4096 个 `txSlot` 规模的交易，Queue 区最多保存 1024 个 `txSlot` 规模的交易。