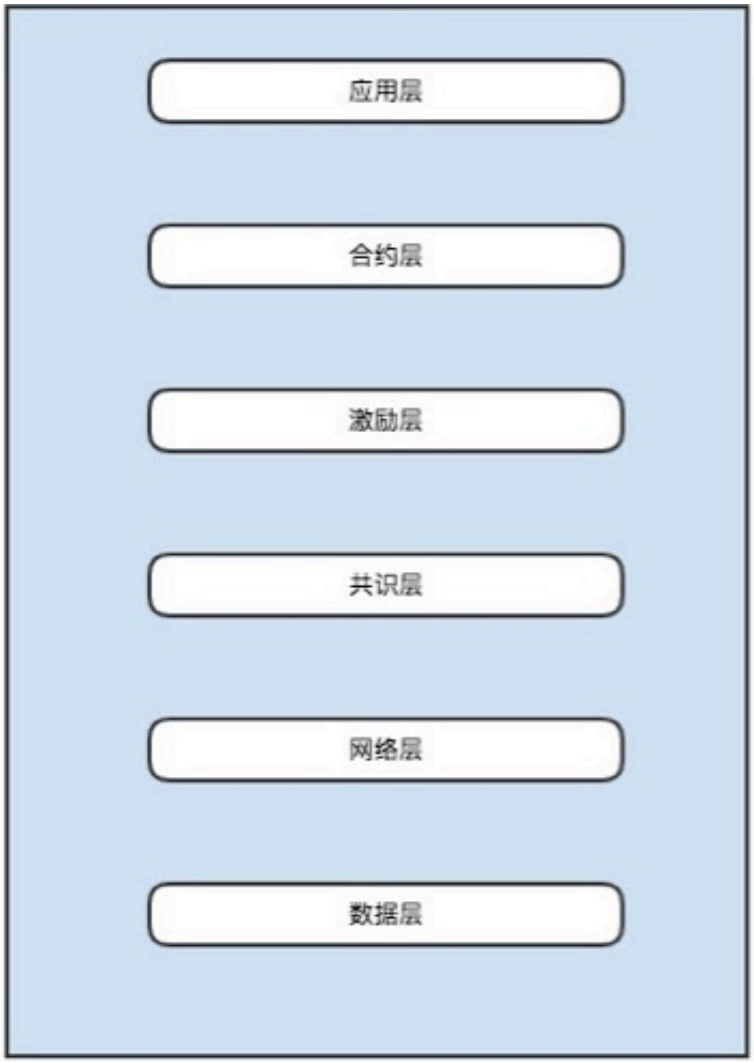
以太坊项目架构模型

一、ETH架构结构

学习以太坊的第一步，最重要的就是首先知道以太坊有什么功能和以太坊的基础架构，宏观上先清楚以太坊是什么，然后在学习各个模块的作用和模块之间的关系，最后在深入浩瀚的源码中，这种思路对学习以太坊源码很有帮助。下面这些以太坊架构图，很简单但是也很经典，所以首先我们先用这些图介绍以太坊的架构特征。

二、从逻辑上分析以太坊

首先看一张以太坊逻辑图



- 数据层：本质是一个数据库，存储区块链中的所有信息内容，以太坊的存储是由leveldb数据库保存的,leveldb的数据是以键值对的形势存储。所有与交易，操作相关的数据，都保存在区块中，并且每个区块链接起来，则构成更大的区块体系，称为区块链。所有交易或操作，存在各个个体账户的状态(state)中，账户的呈现形式是为stateObject有statedb统一管理。

- 网络层：以太坊底层分布式网络即P2P网络，使用了经典的Kademlia网络，可以称为kad，kad是一种能在本身节点边查找临近节点的算法，最终通过UDP实现网络广播，为了能够在全互联网实现通讯，以太坊采用NAT穿透技术实现路由器的穿透，确保全世界以太坊全节点信息保持一致。

- 共识层：共识层就是规定通过何种方式实现交易记录的过程。比如比特币用的是工作量证明机制，用PoW标示，该共识算法完全取决于电脑的性能越好，越容易获取到货币奖励。以太坊也是采用PoW算法，但是PoW存在很多大弊端，比如非常费电浪费资源，所以以太坊有从PoW转向PoS算法的趋势。我自己理解这种方式是必然趋势，但是这样会破坏以太坊现有架构，所以难度非凡。

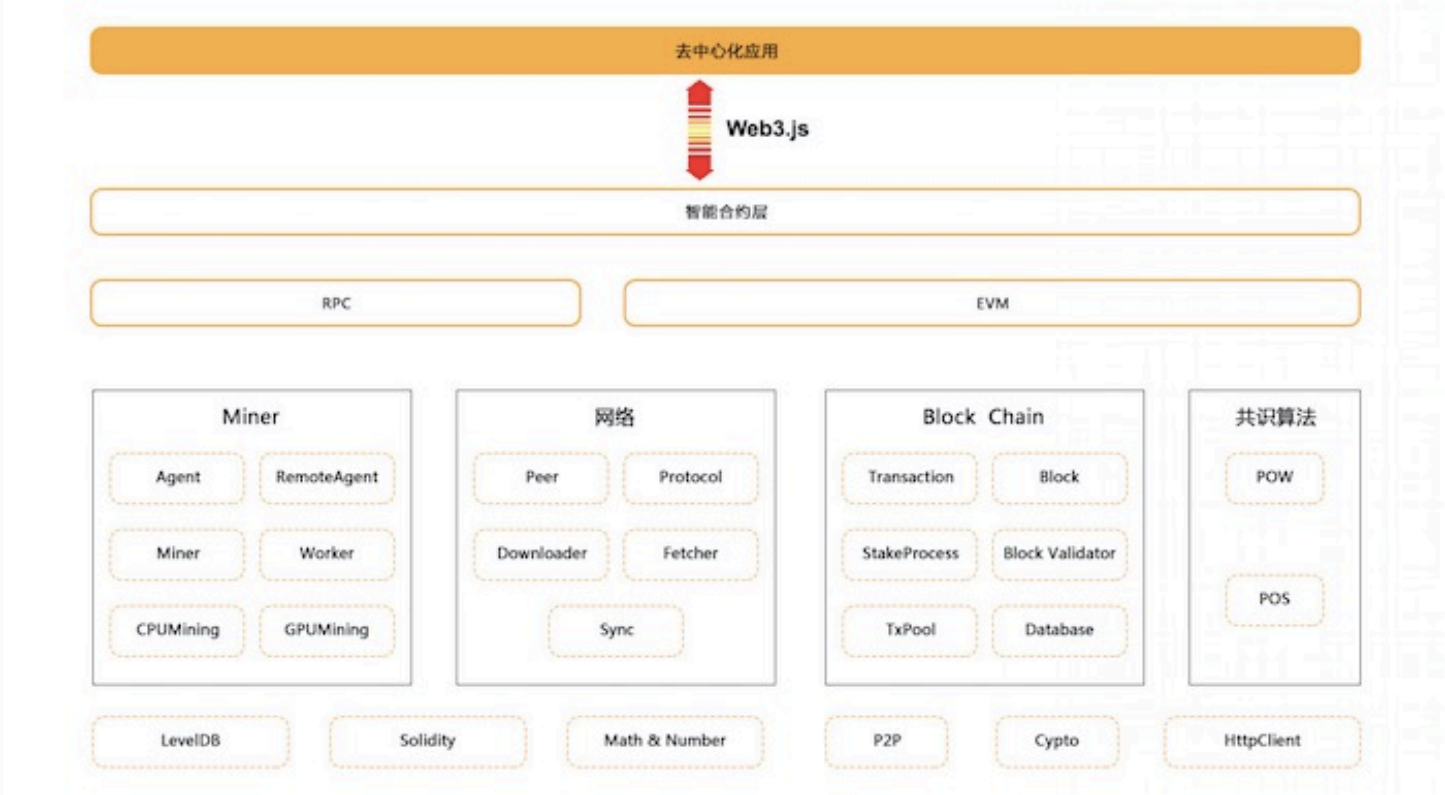
- 激励层：以太坊采用PoW方式实现挖矿机制。

- 合约层：最早的公链是没有这一层的，所以最初的公链只能进行交易，而无法用于其他的领域或是进行其他的逻辑处理。由以太坊的出现，合约层显体现出重大作用，通过合约层开发者可以开发任意多的DAPP，实现完整的区块链生态系统，以太坊中这部分包括了坊虚拟机EVM和智能合约两部分。

- 应用层：以太坊最上层，也是显示层。如以太坊使用的是truffle和web3-js技术。可以是移动端或者web端，目前在应用层开发的产品层次不穷，比如以太坊加密猫、fomo3D都是风靡一时的应用层产品。基于此层开发DAPP，发行代币等都是在这个层面，通过DAPP提供的Web应用和智能合约交互。

三、从结构上分析以太坊

我们看一下以太坊层次图



以太坊最上层的是DApp。它通过Web3.js和智能合约层进行交互。所有的智能合约都运行在EVM（以太坊虚拟机）上，并会用到RPC的调用。在EVM和RPC下面是以太坊的四大核心内容，包括：blockChain, 共识算法，挖矿以及网络层。除了DApp外，其他的所有部分都在以太坊的客户端里，目前最流行的以太坊客户端就是Geth（Go-Ethereum）

四、从体系上分析以太坊

- 1 区块链范式
  - 1.1 基于交易的状态机系统
    - 1.1.1 创世区块起源（genesis）
    - 1.1.2 状态改变到最终状态
  - 1.2 账户状态
    - 1.2.1 账户余额
    - 1.2.2 名誉度
    - 1.2.3 信誉度
    - 1.2.4 显示世界附属
    - 1.2.5 扫绘信息
  - 1.3 交易是链的两个状态桥梁
  - 1.4 一个有效的状态是通过交易进行的
  - 1.5 区块链中交易信息，通过Hash算法连接
  - 1.6 挖矿
    - 1.6.1 付出一些列工作量还赢得交易记账权
    - 1.6.2 Proof Of Work

五、以太坊简易框架

如果阅读当前版本的以太坊难度大，那么我们不读一下github上的以太坊最初版本，通过阅读这个代码可以加深理解以太坊的模块组成，揣测设计的想法和思路。去掉单元测试文件，整个项目最重要的部分其实只有以下部分

```
big.go
vm.go
parsing.go
transaction.go
block.go
block_manager.go
ethereum.go
serialization.go
8个文件。这8个文件都比较小，功能比较简单，也很好理解。
```

数学封装

big.go 封装了大整数的指数运输。

虚拟机

vm.go 主要作用为通过操作码编译只能合约，定义了虚拟机操作码, 操作码类型, 虚拟机结构和虚拟机的实现。
虚拟机内部定义的指令码有：

```
    oSTOP       int = 0x00
    oADD        int = 0x10
    oSUB        int = 0x11
    oMUL        int = 0x12
```

```
    oDIV       int = 0x13
    oSDIV      int = 0x14
    oMOD       int = 0x15
    oSMOD      int = 0x16
    oEXP       int = 0x17
    oNEG       int = 0x18
    oLT        int = 0x20
    oLE        int = 0x21
    oGT        int = 0x22
    oGE        int = 0x23
    oEQ        int = 0x24
    oNOT       int = 0x25
    oSHA256    int = 0x30
    oRIPEMD160 int = 0x31
    oECMUL     int = 0x32
    oECADD     int = 0x33
    oSIGN      int = 0x34
    oRECOVER   int = 0x35
    oCOPY      int = 0x40
    oST        int = 0x41
    oLD        int = 0x42
    oSET       int = 0x43
    oJMP       int = 0x50
    oJMPI      int = 0x51
    oIND       int = 0x52
    oEXTRO     int = 0x60
    oBALANCE   int = 0x61
    oMKTX      int = 0x70
    oDATA      int = 0x80
    oDATAN     int = 0x81
    oMYADDRESS int = 0x90
    oSUICIDE   int = 0xff
```

总共36个指令码，我们通过阅读36个指令码可能会发现指令码定义编号的值不是连续的，通过通读代码分析得知，指令码值的高位是记录指令类型。以太坊vm的实现是基于栈的完成的，实现相对比较简单。以上指定的大部分指令码的功能没有实现，只实现了以下指令码的功能 。

```
    oSTOP
    oADD
    oSUB
    oMUL
    oDIV
    oSET
    oLD
    oLT
    oJMP
```

```
    oJMPI
```

智能合约编译 parsing.go 主要实现的是智能合约的编译和对编译后的代码进行处理，后续供虚拟机执行。交易transaction.go 定义了交易的结构，还有费用和收益变量。一笔交易包括发起者，接受者，交易的数量，交易的费用，编译后的脚本源码，运行需要内存，交易签名和地址。费用和收益变量只有初始化赋值，没有具体使用。脚本源码是智能合约的雏形，为了方便描述和理解还是称呼它为智能合约。此时的智能合约语言和x86 intel汇编类似，语法比较简单，一个操作指令加上操作数，操作数的个数最常见是0个，1个，2个和3个，设计者实现的时候，最多可以支持6个。操作指令和操作数之间用空格分开，操作数与操作数之间也用空格分开。定义的操作指令有以下这些：

```
    "STOP":       "0",
    "ADD":        "16",  // 0x10
    "SUB":        "17",  // 0x11
    "MUL":        "18",  // 0x12
    "DIV":        "19",  // 0x13
    "SDIV":       "20",  // 0x14
    "MOD":        "21",  // 0x15
    "SMOD":       "22",  // 0x16
    "EXP":        "23",  // 0x17
    "NEG":        "24",  // 0x18
    "LT":         "32",  // 0x20
    "LE":         "33",  // 0x21
    "GT":         "34",  // 0x22
    "GE":         "35",  // 0x23
    "EQ":         "36",  // 0x24
    "NOT":        "37",  // 0x25
    "SHA256":     "48",  // 0x30
    "RIPEMD160":  "49",  // 0x31
    "ECMUL":      "50",  // 0x32
    "ECADD":      "51",  // 0x33
    "SIGN":       "52",  // 0x34
    "RECOVER":    "53",  // 0x35
    "COPY":       "64",  // 0x40
    "ST":         "65",  // 0x41
    "LD":         "66",  // 0x42
    "SET":        "67",  // 0x43
    "JMP":        "80",  // 0x50
    "JMPI":       "81",  // 0x51
    "IND":        "82",  // 0x52
    "EXTRO":      "96",  // 0x60
    "BALANCE":    "97",  // 0x61
    "MKTX":       "112", // 0x70
    "DATA":       "128", // 0x80
    "DATAN":      "129", // 0x81
```

```
    "MYADDRESS":  "144", // 0x90
    "BLKHASH":    "145", // 0x91
    "COINBASE":   "146", // 0x92
    "SUICIDE":    "255", // 0xff
```

可以看出这是操作指令到虚拟机内部指令码的映射。编译规则很简单：

1. 操作指令根据映射表1，得到vm的内部指令码。
2. 每一个操作数（第i个操作数，i记作位置序数， 从1开始）分别乘以 256的i次方，
3. 将步骤2的乘积依次相加，最后加上步骤1得到的指令， 最终的和作为编译结果。

一个合法的智能合约源码片段可能是这样（记作 代码片段1）

```
    "SET 10 6",
    "LD 10 10",
```

按照编译规则，代码片段1最终的编译结果是这样的

```
    395843    // 67 + 10 * 256 + 6 * 256^2
    133698    // 66 + 10 * 256 + 10 * 256^2
```

vm运行时，根据编译规则的逆规则，解析出指令码和操作数，根据指令码的功能，进行下一步处理。运行需要内存没有使用，猜测是用作运行智能合约。签名字段没有使用，猜测是校验交易是否篡改过。 地址是对transaction结构序列化后的字节数组取sha256的前20位。

区块

block.go用来定义块结构，非常简单，仅包含一个transaction数组.

区块管理器

block_manager.go是定义块管理器，用来处理块，持有一个vm指针，依次执行块里面的每一个交易的智能合约。

以太坊入口

ethereum.go是demo程序入口， mock两笔交易，打印vm执行的日志，最后打印了其中一笔交易的序列化结果。

交易序列化

serialization.go 实现序列化功能，采用的是RLP编码，只能对字符串编码。编码规则是
1. 如果是字符串，编码结果是”\x00” 加上字符串的长度，再加上原字符串。计算字符串的长度有一个规则，确保编码无二义性，能正确解码。
2.如果是字符串数组，编码结果是”\x01”加上每一个字符串编码结果的长度和的编码，再加上每一个字符串的编码结果。有点绕口，这是个递归的过程。
3.如果是其他类型需要转换成字符串或者字符串数组。
RLP编码的规律是以数据类型开始，字符串是”\x00”, 字符串数组是”\x01, 然后是数据长度，最后是数据内容。
RLP编码和解码是互逆过程，实现比较简单，编码紧凑，传输效率较高，后续版本中，在网络传输和本地存储都有RLP编码。

总结

总体来说，这个版本的代码比较简单，是geth的初始设计和验证，不过对于初次学习geth源码，整体认识geth还是有一定的意义。