支持智能合约的公有链系统以太坊简介

目录

1.1 以太坊概述

1. 以太坊是一个可编程的区块链平台，是在比特币的基础上发展的新一代公共区块链平台。

2. 与比特币相比除了技术上的差异，更重要的是以太坊侧重于可编程的智能合同即分布式应用程序。

3. 以太币仅仅是驱动智能合约执行所需燃料的加密令牌。

1.2 以太坊设计原则

简单性

协议尽可能简单。

普遍性

没有“特征”。

模块化

各个部分尽可能模块化和可分离。

敏捷性

- 更改容易。
- 允许通过硬分叉进行重大的修改。

不歧视不审查

- 协议不会限制特定类别的使用。
- 监管机制不会反对具体的不可取的应用。
- 能运行无限循环脚本。

1.3 以太坊虚拟机

智能合约在不同的操作系统、硬件平台都应具有确定性和一致性的结果。

可以执行任意算法复杂性的代码，是以太坊的核心。

开发人员可以使用基于现有熟悉的编程创建在EVM上运行的应用程序。

每个运行EVM的节点都是共识协议的一部分，独立验证交易序列、运行交易触发的代码。

1.4 以太坊系统特点

01

可以部署基于分布式的大规模并行化计算。

02

为DApp提供了极端水平的容错能力，确保零停机时间。

03

数据不可更改和可追溯。

因此对于一个去中心化的应用程序(DApp)在设计时尽可能仅仅把业务逻辑和关键的共识状态放在区块链中。

1.5 以太坊应用领域

1. 对等实体之间的直接交互或者跨网络的组织之间组织协调。

2. 基于点对点的应用领域或复杂金融合约自动化运行领域。例如：复杂的金融交流或交换；信任、安全和持久性都很重要的环境--例如资产登记、投票、治理和物联网。

1.6 以太坊历史

最早2013年底，Vitalik Buterin在一份白皮书中描述了以太坊的想法，并于2014年1月在美国佛罗里达州迈阿密举行的北美比特币会议上正式宣布了以太坊。

2014年4月发布《以太坊黄皮书》作为以太坊虚机的技术规范。

2014年6月的瑞士楚格成立了以太坊基金会，并宣布了以太令牌的预售计划。

1.7 以太坊升级

以太坊的完整发展路线分为四个阶段：Frontier（前沿）、Homestead（家园）、Metropolis（大都会）和 Serenity（宁静）。

2019年1月16日在区块高度7080000上进行君士坦丁堡硬分叉（也就是硬分叉）。

Serenity（宁静）2020 到来（计划），以太坊将全部转为 POS。实际上是2022开始

1.8 硬分叉

1. 第一次分叉调整了未来挖矿的难度，确保未来的用户会有转换至权益证明的动机。
2. 第二次分叉发布了第一个稳定版本，称作“家园”（Homestead）。
3. 第三次分叉：DAO和区块链分叉。
4. 第四次分叉：减重和防DDoS。
5. 当前第五个分叉正在开发中。

软分叉向前兼容，硬分叉不向前兼容
旧节点是否接受新区块，是软硬分叉的本质区别，软分叉接受，硬分叉不接受，若拒绝升级，则产生两条链。

1.9 发展与挑战

1. 2018年9月，比特币核心开发者Jeremy Rubin在美国科技媒体TechCrunch上发表文章《ETH的崩溃无法避免》，称就算以太坊网络继续存续，ETH的价值也会必然归零。

2. 2018年12月10日，Vitalik在推特上宣称，未来采用基于权益证明（PoS）的分片技术的区块链“效率将提高数千倍”。

3. 2019年，以太坊项目进行君士坦丁堡硬分叉，这是一个刺激以太坊网络改变其核心共识机制算法的代码，这一段代码引导之后以太坊便会面临所谓的“冰河时代”，在该网络上的创建新区块的难度将会不断提升，最终减慢到完全停止。

02 以太坊框架

2.1 核心概念--以太坊账户

账户是以太坊中的核心角色，以太坊中的交易就是账户之间的价值和信息的转换。

账户（20字节的地址）构成部分			
随机数（用作计数器）	余额	合约代码（如果有）	存储

外部账户（EOAs）

由公钥-私钥对控制，没有相关代码。

合约账户

由交易类型、消息类型进行创建，由代码控制

以太坊中所有的交易发起者只能是外部账户，但合约账户中智能合约执行过程中可以创建新的交易。

2.2 核心概念--以太币

1. 以太币（Ether）是以太坊中货币的名称，它被用来支付以太坊虚拟机（EVM）计算的费用。

2. 以太币不是被直接消耗掉而是间接的通过购买gas（瓦斯）来实现的。

3. 其最小的面额也就是基础单位被称为：Wei，通常ether被认为是以太币的单位（1以太币等于10的18次方个Wei，1e18 Wei）。

4. 通常获取以太币有三种方式：成为以太坊矿工；通过第三方担保和持有以太币的人进行交易；使用用户友好的Mist中的以太坊GUI钱包，用API购买以太币。

2.3 核心概念--交易

以太坊中“交易”是指存储从外部账户发出的消息的签名数据包，数据包中包含从一个外部账户发往一个账户的消息。

交易构成部分
消息接收者
签名后的发送者
Value字段（单位：Wei）
STARTGAS值（设置最大的瓦斯使用量）
GASPRICE值（用户接受的最大瓦斯单价）
可选字段

2.4 核心概念--消息

消息可以理解为一种特殊的交易，是合约代码执行过程中对其他合约的功能调用（通过操作码“CALL”和“DELEGATECALL”）。

消息构成部分
消息发送者
消息接收者
Value字段（单位：Wei）
STARTGAS值（设置最大的瓦斯使用量）
可选字段

消息与交易都会导致接收者的合约账户执行合约代码，区别在于交易是外部账户创建的，而消息则由合约也只能通过合约创建。

2.5 核心概念--GAS

1. 在以太坊中用瓦斯（GAS）作为合约代码执行成本基本单位。瓦斯由外部账户通过以太币从执行代码的矿工那里购买。

2. 矿工可以决定瓦斯价格，即设置最低价格的单价，低于该价格的交易会被矿工拒绝。

3. 以太坊为每一个交易和合约的计算步骤设置费用，外部账号可以为交易设置以太币值、STARTGAS和GASPRICE等瓦斯限制，矿工也有权选择验证这个交易以获取费用或者拒绝。

4. 在发送一个交易时根据预估交易费用设置相应的以太币是有必要的，其计算方法如下：
预估交易费用 = 瓦斯使用量 * 瓦斯价格
瓦斯使用量 = 交易瓦斯使用量 + 消息瓦斯使用量
瓦斯价格 = 默认值 = 0.05e18 wei

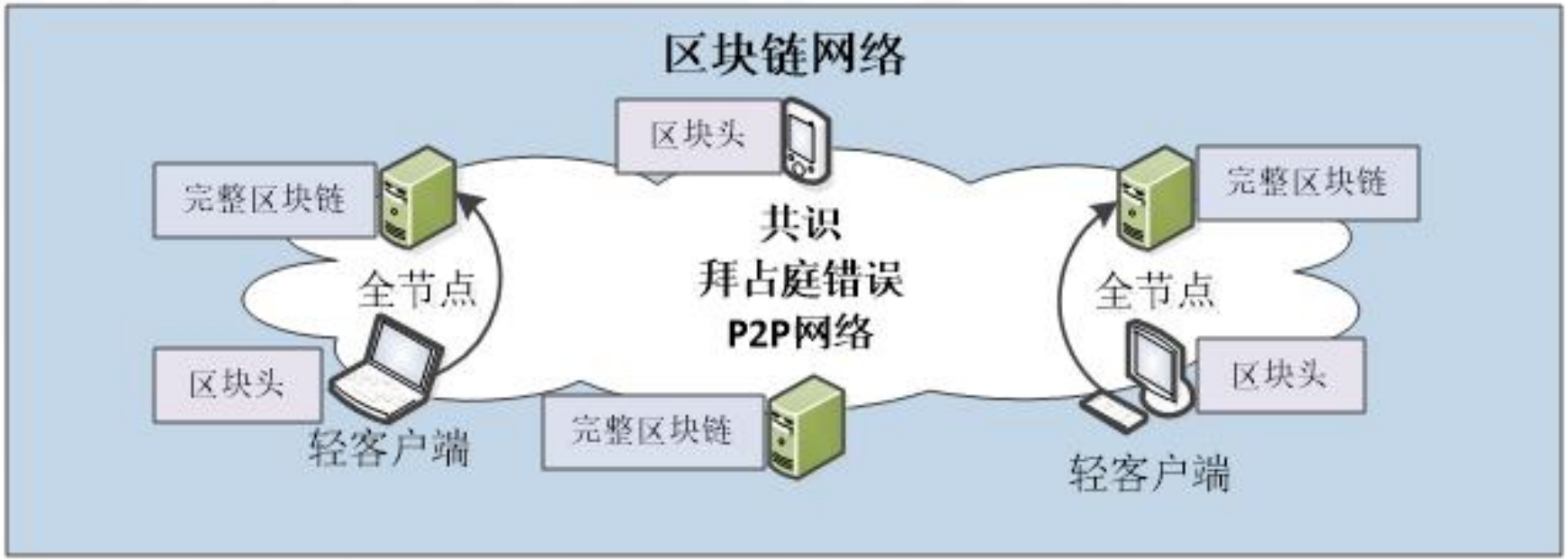
2.6 以太坊运行网络

轻节点

仅保存区块头部信息。

全节点

保存完整数据，独立验证数据，具有高性能硬件环境。



2.7 轻客户端

01

轻客户端使用全节点作为中介，不需要直接与区块链交互。

02

轻客户端只需要下载最新的区块头，速度快效率高。

03

轻客户端能判断哪些全节点是恶意全节点。

04

轻客户端在分片技术中能够发挥重要的作用。

05

轻客户端可以运行在移动设备上。

2.7 轻客户端

用户查询特定账户的余额

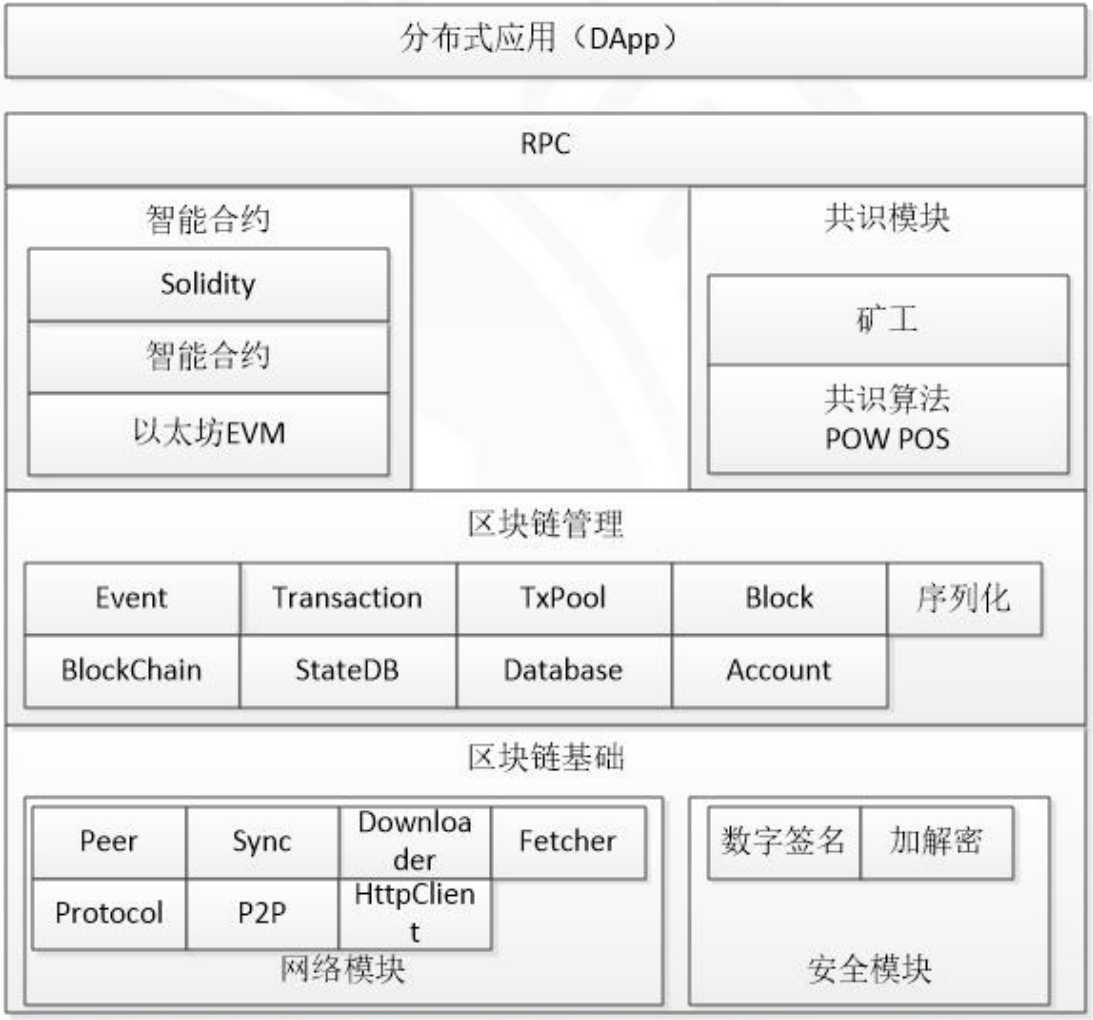
01 用户连接到轻客户端，并发送查询特定账户余额的请求。

02 轻客户端调用连接的全节点转发账户查询请求。

03 验证全节点给出的答案是否与它们所拥有的“签名”相匹配。

05 轻客户端返回结果给用户。

2.8 全节点运行网络


分布式应用（DApp）
RPC
智能合约
Solidity
智能合约
以太坊EVM
共识模块
矿工
共识算法
POW POS
区块链管理
Event
Transaction
TxPool
Block
序列化
BlockChain
StateDB
Database
Account
区块链基础
Peer
Sync
Downloader
Fetcher
Protocol
P2P
HttpClient
网络模块
数字签名
加解密
安全模块

2.9 DAPP

1. 以太坊的最上层是DApp，是在以太坊平台的基础上由许多开发团队所开发的各类分布式应用项目，如和以太坊基金会是独立的Augur, Digix, Maker。DApp和以太坊的关系与操作系统和应用软件的关系相似。

2. 以太坊平台向应用提供智能合约、共识、区块链、安全等基础设施，而应用则丰富并扩展了以太坊的使用范围并深入不同的行业。

2.10 以太坊客户端

客户端	语言	开发团队	发布版本
go-ethereum	Go	以太坊基金会	go-ethereum-v1.4.18
Parity	Rust	Ethcore	Parity-v1.4.0
cpp-ethereum	C++	以太坊基金会	cpp-ethereum-v1.3.0
pyethapp	Python	以太坊基金会	pyethapp-v1.5.0
ethereumjs-lib	Javascript	以太坊基金会	ethereumjs-lib-v3.0.0
Ethereum(J)	Java	<ether.camp>	ethereumJ-v1.3.1
ruby-ethereum	Ruby	Jan Xie	ruby-ethereum-v0.9.6
ethereumH	Haskell	BlockApps	no Homestead release yet

2.11 以太坊API

开发库	语言	项目主页
JavaScript连接(web3.js)	JavaScript	https://github.com/ethereum/web3.js
Java连接(web3j)	Java	https://github.com/web3j/web3j
.Net连接(Nethereum)	C# .NET	https://github.com/Nethereum/Nethereum
Ruby连接(ethereum-ruby)	Ruby	https://github.com/DigixGlobal/ethereum-ruby

03 智能合约

3.1 智能合约简介

1. 以太坊中的智能合约是具有特定地址（合约账户）的代码（合约功能）和数据（合约状态）的集合。

2. 以太坊中的智能合约通常使用Solidity等高级语言编写，编译成基于EVM的字节代码后上传到合约账户。

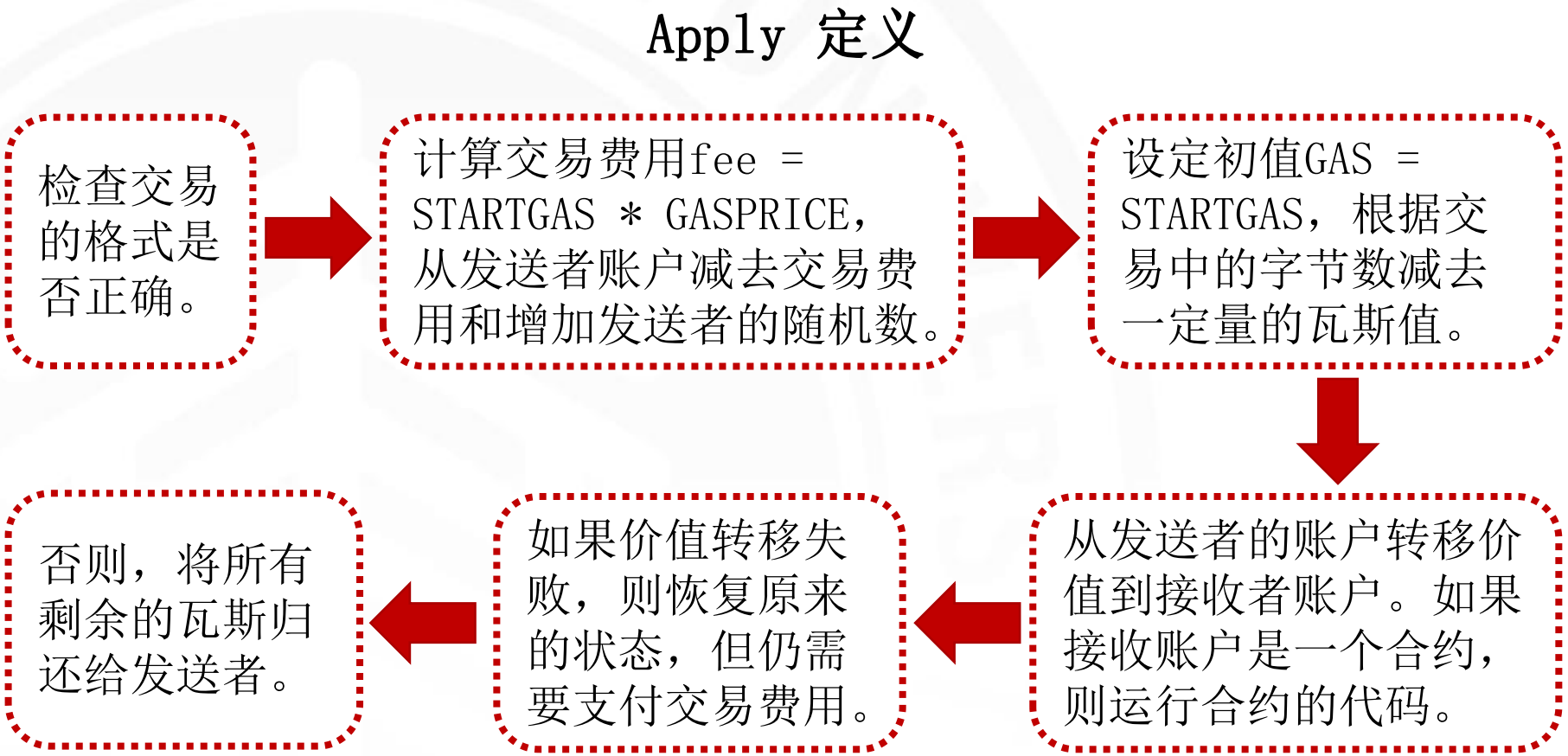
3. 合约的代码以基于EVM的二进制字节码存储在账户中，当被交易或消息触发时在EVM中运行。

4. 合约账户之间可以相互传递消息并实现图灵完备运算。

3.2 状态转换

以太坊中所有的账户状态构成世界状态，外部账户触发的每个交易都会导致世界状态的变化。这一过程可以抽象为以太坊的状态转换函数：APPLY(S,TX) -> S'

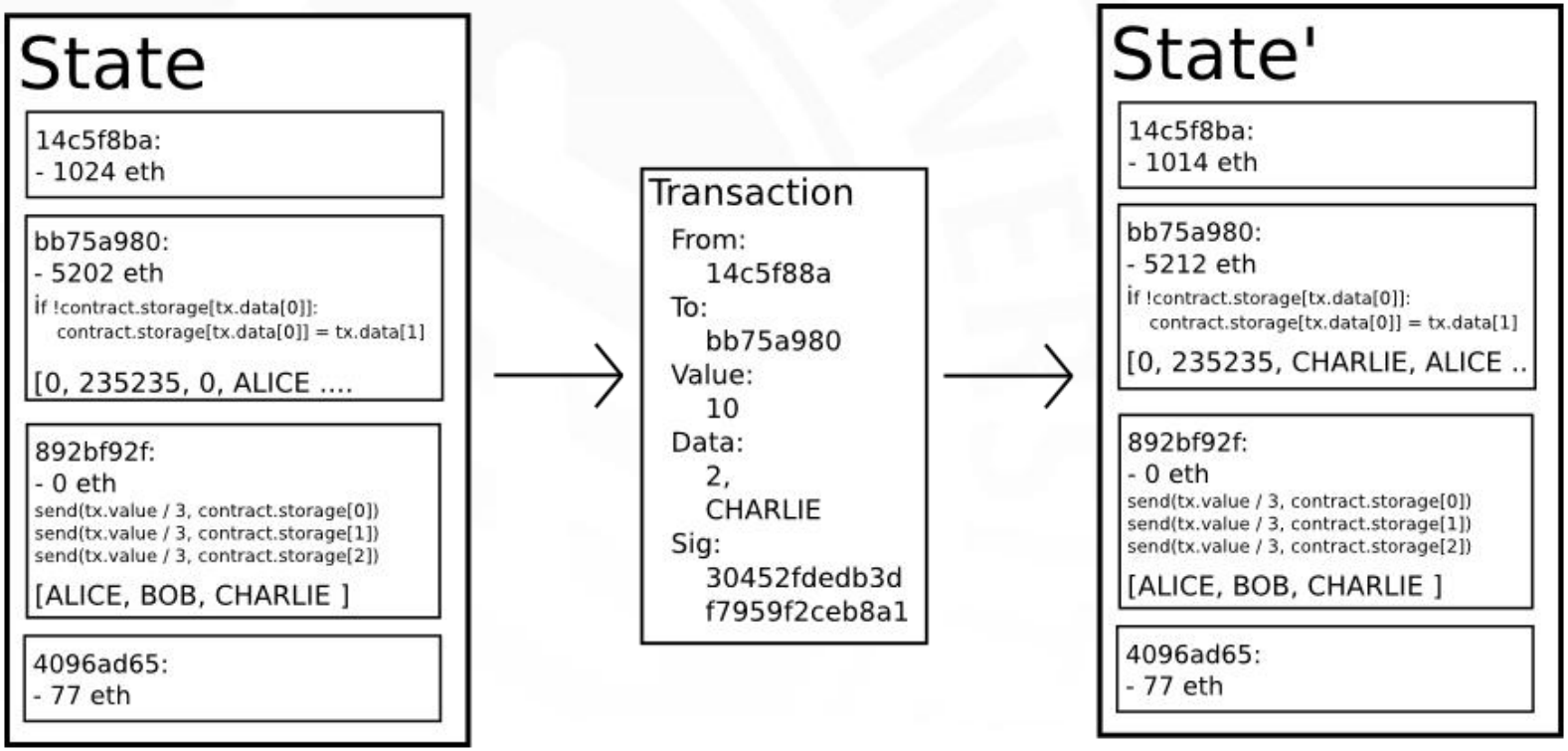
3.2 状态转换


Apply 定义
检查交易的格式是否正确。
计算交易费用fee = STARTGAS * GASPRICE，从发送者账户减去交易费用和增加发送者的随机数。
设定初值GAS = STARTGAS，根据交易中的字节数减去一定量的瓦斯值。
从发送者的账户转移价值到接收者账户。如果接收账户是一个合约，则运行合约的代码。
如果价值转移失败，则恢复原来的状态，但仍需要支付交易费用。
否则，将所有剩余的瓦斯归还给发送者。

3.2 状态转换

例：合约代码如下

```
if not self.storage[calldataload(0)]:
    self.storage[calldataload(0)] = calldataload(32)
```



状态处理流程

– 交易中STARTGAS为2000瓦斯，GASPRICE为0.001以太币

- 检查交易是否有效、格式是否正确
- 检查交易发送者至少有2000*0.001=2个以太币。如果有，从发送者账户中减去2个以太币。***- 先按预估瓦斯费用扣款，再返款***
- 初始设定gas=2000,假设交易长为170字节，每字节的瓦斯使用量是5，减去850，所以还剩1150。
- 从发送者账户减去10个以太币，为合约账户增加10个以太币。***- 转账金额***
- 运行代码。在这个合约中，运行代码很简单：它检查合约存储器索引为2处是否已使用，注意到它未被使用，然后将其值置为CHARLIE。假设这消耗了187单位的瓦斯，于是剩余的瓦斯为1150 - 187 = 963。***- 智能合约运行瓦斯使用量***
- 向发送者的账户增加963*0.001=0.963个以太币，返回最终状态。 ***- 返款金额***

智能合约举例

- 1. 编写智能合约，将其发布到区块链网络中，每个智能合约对应一个合约账户，其他用户通过合约账户调用智能合约。
- 2. 若有用户想要创建一个会议，发起一项交易：sender是用户自己账户地址，receiver是智能合约的合约账户地址，调用函数是合约里面定义的Conference（）函数，用户对交易签名。
- 3. 区块链中的节点收到这笔调用智能合约的交易之后，验证交易，验证通过后调用执行该合约的Conference（）函数。
- 4. 当用户想要修改会议上限人数和给用户退款的话，分别发起交易，具体步骤与第二步相同，但是需要在交易中指定相应函数的参数，同样，用户需要对交易签名。

智能合约举例

- 5.区块链节点收到交易之后，验证交易，调用智能合约内相应的函数。
- 6.参会者想要参加会议时，发起交易，sender是自己的账户地址，receiver是合约地址，在交易中指明买票的函数buyTicket(),之后对交易签名。
- 7.区块链节点收到买票交易之后，验证交易的签名，验证通过之后调用合约账户里的buyTicket（）函数，记录买票信息，参会者的买票的钱会保存在合约账户里。
- 8.当最后用户（会议组织者）结束会议时，用户发起结束交易，调用合约内的destroy（）函数，用户对交易签名。区块链节点收到结束会议交易并验证通过之后，调用合约内的destroy（）函数，合约内的钱会转到组织者的账户里。合约必须实现destroy函数，否则合约账户里面的钱没有办法取出。

智能合约举例

```
<code style="font-family:Menlo, Courier, monospace, monospace, sans-serif;font-siz
  address public organizer;
  mapping (address => uint) public registrantsPaid;
  uint public numRegistrants;
  uint public quota;

  event Deposit(address _from, uint _amount);  // so you can log these events
  event Refund(address _to, uint _amount);

  function Conference() { // Constructor
    organizer = msg.sender;
    quota = 500;
    numRegistrants = 0;
  }
  function buyTicket() public returns (bool success) {
    if (numRegistrants >= quota) { return false; }
    registrantsPaid[msg.sender] = msg.value;
    numRegistrants++;
    Deposit(msg.sender, msg.value);
    return true;
  }
  function changeQuota(uint newquota) public {
    if (msg.sender != organizer) { return; }
    quota = newquota;
  }
  function refundTicket(address recipient, uint amount) public {
    if (msg.sender != organizer) { return; }
    if (registrantsPaid[recipient] == amount) {
      address myAddress = this;
      if (myAddress.balance >= amount) {
        recipient.send(amount);
        registrantsPaid[recipient] = 0;
        numRegistrants--;
        Refund(recipient, amount);
      }
    }
  }
  function destroy() { // so funds not locked in contract forever
    if (msg.sender == organizer) {
      suicide(organizer); // send funds to organizer
    }
  }
}</code>
```

04 挖矿

4.1 以太坊挖矿简介

1. 以太坊与比特币一样，使用奖励-驱动的安全模型，矿工生产出可以被其它矿工验证通过的合法区块，以太坊给挖出该区块的矿工一定金额的以太币进行奖励。

2. 在以太坊的里程碑Serenity版本前采用工作量证明（PoW），之后的版本中将会被股权证明（PoS）逐步取代。

   PoS（Proof of Stake）这种机制与PoW机制的最大不同在于，只有持有数字货币的人才能进行挖矿，而且不需要大量的算力就可以挖到货币，避免了比特币网络中出现的“算力集中”趋势，回归到区块链“去中心化”的本质要求。

4.2 以太坊共识机制

1. 以太坊的共识机制是基于选择最高难度的区块，其PoW算法称为Ethash（the Dagger-Hashimoto algorithm 的一个变种）。

2. Ethash计算时依赖一个根据随机数和块头数据所选择的固定资源的子集。这个固定资源是每30，000个块所生成的DAG。不同的30000个块产生的DAG是不同的，这个DAG大小量级为数个GB。

3. DAG计算完毕后，才能继续生产新的区块，但可以提前被计算出来。对于块的验证节点则不需要DAG资源，仅需要很低的CPU和内存资源。

4. 按照15秒一个块的难度，大约一个DAG数据会使用125个小时(大约5.2天)被称为一个纪元(epoch)。

4.2 以太坊共识机制

其他节点接收到矿工产生的区块后的验证过程

检查区块引用的上一个区块的有效性。

↓

检查区块的时间戳是否比引用的上一个区块大，而且小于15分钟。

↓

检查区块序号、难度值、 交易根，叔根和瓦斯限额是否有效。

↓

检查区块的工作量证明是否有效。

→

将S[0]赋值为上一个区块的STATE_ROOT。

↑

将TX赋值为区块的交易列表，进行状态转换S[i+1] = APPLY(S[i],TX[i])。

↑

用S[n]给S_FINAL赋值，向矿工支付区块奖励。

↑

检查S_FINAL是否与STATE_ROOT相同。

4.3 挖矿的分类

CPU挖矿

使用计算机的中央处理器（CPU）来开采以太币。这种方式目前已经不太适合了，因为GPU的挖矿效率已经比它高出两个数量级了。

GPU挖矿

GPU挖矿使用OpenCL进行实现，所以AMD的显卡会比同档次的NVIDIA显卡速度“更快”。每个GPU的RAM都需要达到1-2GB。

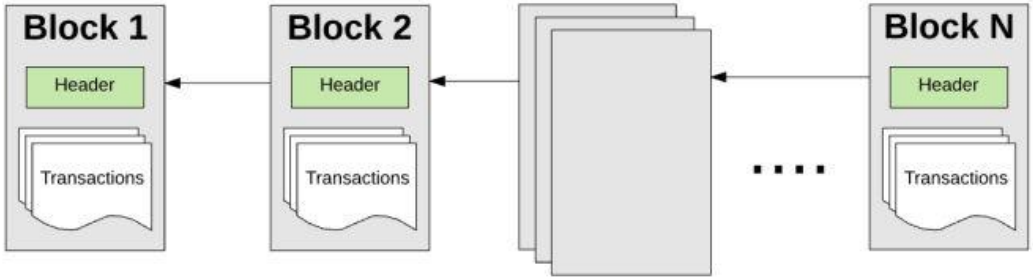
矿池挖矿

矿池是一个合作体，通过整合分散矿工的算力合作开采，来使每个矿工的回报趋于平滑。矿池的算力增长增加了51%攻击的风险。

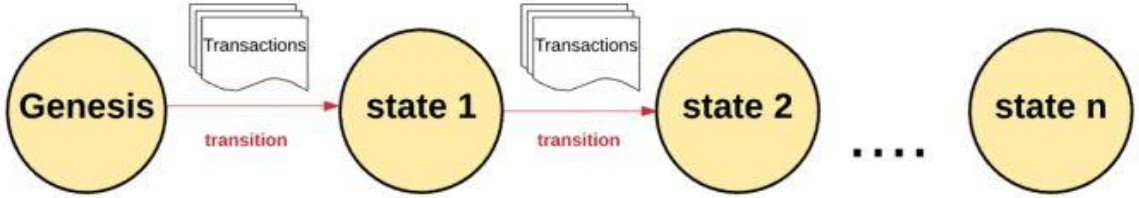
05 以太坊数据结构

5.1 区块链

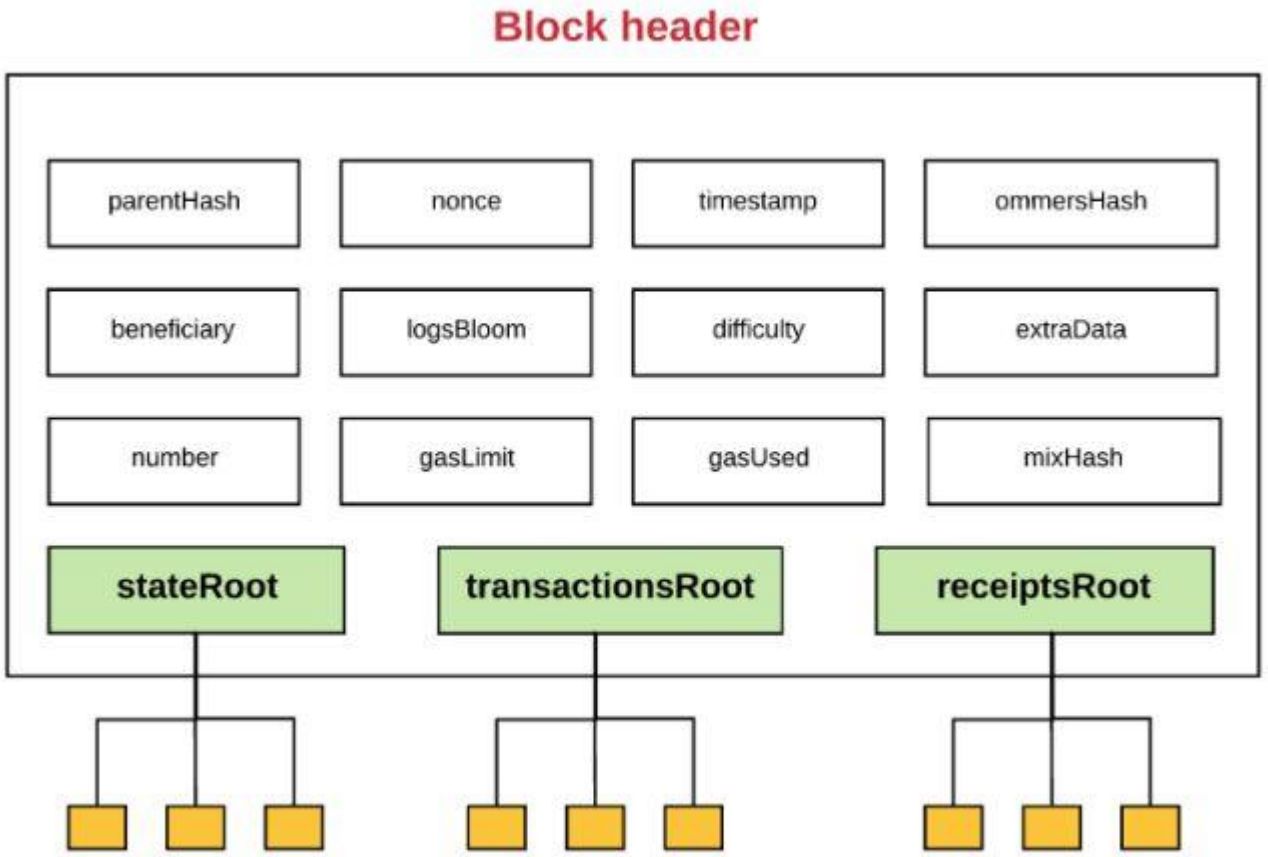
在单个节点中，数据是以区块链（BlockChain）的形式来存储的。区块链由一个个串在一起的块（Block）组成。以太坊大概每十几秒会生成一个新块，记录了这个段时间内的各种信息。



以太坊可以被描述为为一个交易驱动的状态机。



5.2 区块头


Block header
parentHash
nonce
timestamp
ommersHash
beneficiary
logsBloom
difficulty
extraData
number
gasLimit
gasUsed
mixHash
stateRoot
transactionsRoot
receiptsRoot

5.2 区块头

Number	区块的序号，也称为区块高度
ParentHash	指向父区块(parentBlock)的指针，通过CurrentBlock往前遍历形成完整规范链
UncleHash	区块体中的uncles属性的RLP哈希值
Coinbase	挖掘出这个区块的账户地址
Difficulty	挖掘出该区块的难度
Time	区块产生时的全局时间戳
GasLimit	区块内所有Gas消耗的理论上限
Nonce	一个64bit的哈希数
TxHash	交易列表所形成的MPT树的根节点的RLP哈希值
Root	StateDB中的“state Trie”的根节点的RLP哈希值
ReceiptHash	Block中的 "Receipt Trie”的根节点的RLP哈希值
Bloom	Bloom过滤器(Filter)
GasUsed	区块内所有交易执行时实际消耗的Gas总和

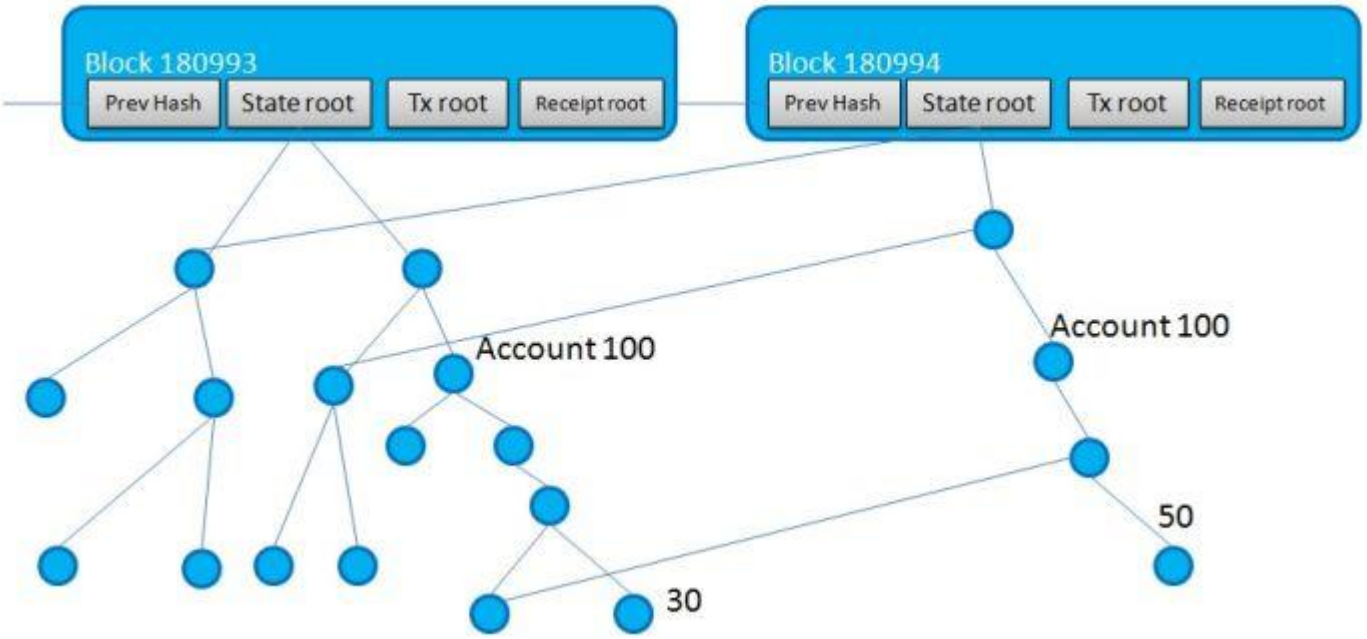
反映了区块在区块链中的位置。

与挖矿有关，UncleHash与Time影响后续难度，Difficulty反映了当前难度，GasLimit限制了区块大小，Nonce是挖矿的最终结果。

反映了区块体中交易的签名、交易执行有关的瓦斯费用、交易执行后的结果以及交易执行的日志等

5.3 区块体

状态树是Merkle Patricia Tree，多个块的MPT树共享了账户状态，子块状态树和父块状态树的差别在于它指向了在子区块中被改变了的账户。这样节省了总的存储空间，方便了块的回滚操作。



5.3 区块体

以太坊区块的Body中主要存储Header对应的具体数据，如交易列表及uncle区块，但并不保存世界状态、交易日志等数据。

1. Transactions：区块的交易列表，通过RLP序列化存储在物理数据库中。

2. Uncles：在以太坊挖矿模块创建新的区块时，如果由于节点接收到的新区块导致原有的规范链发生变更，使得原来处于规范链中的区块变为非规范链中的区块，当矿工接收到侧链消息（chainSideCh）后，会将这些区块作为UncleBlock打包到新的区块中。

5.3 物理存储

1. 区块链的相关数据以k-v序列对的形式保存在LevelDB数据库中。

2. 以太坊通过ethdb.LDBDatabase对其进行封装并通过代码（core/database_util.go）提供一系列的存取服务。

3. 区块链存储到物理数据库中的信息主要包括区块链的信息以及区块的相关信息。

5.3 物理存储

序号	key	value	说明
1	LastHeader	hash	规范链的最新区块头，HeaderChain中使用
2	LastBlock	hash	规范链的最新区块，BlockChain中使用
3	LastFast	hash	规范链的最新区块，BlockChain中使用
4	"h"+num+"n"	hash	规范链中相应高度所对应的区块
5	"H" + hash	num	区块高度
6	"b" + num + hash	body	区块的区块体
7	"r" + num + hash	receipts	区块的log信息
8	"l" + txHash	{hash,num,TxIndex}	交易对应的区块、高度及交易在区块中的索引
9	"h" + num + hash + "t"	td	区块的总难度

The DAO

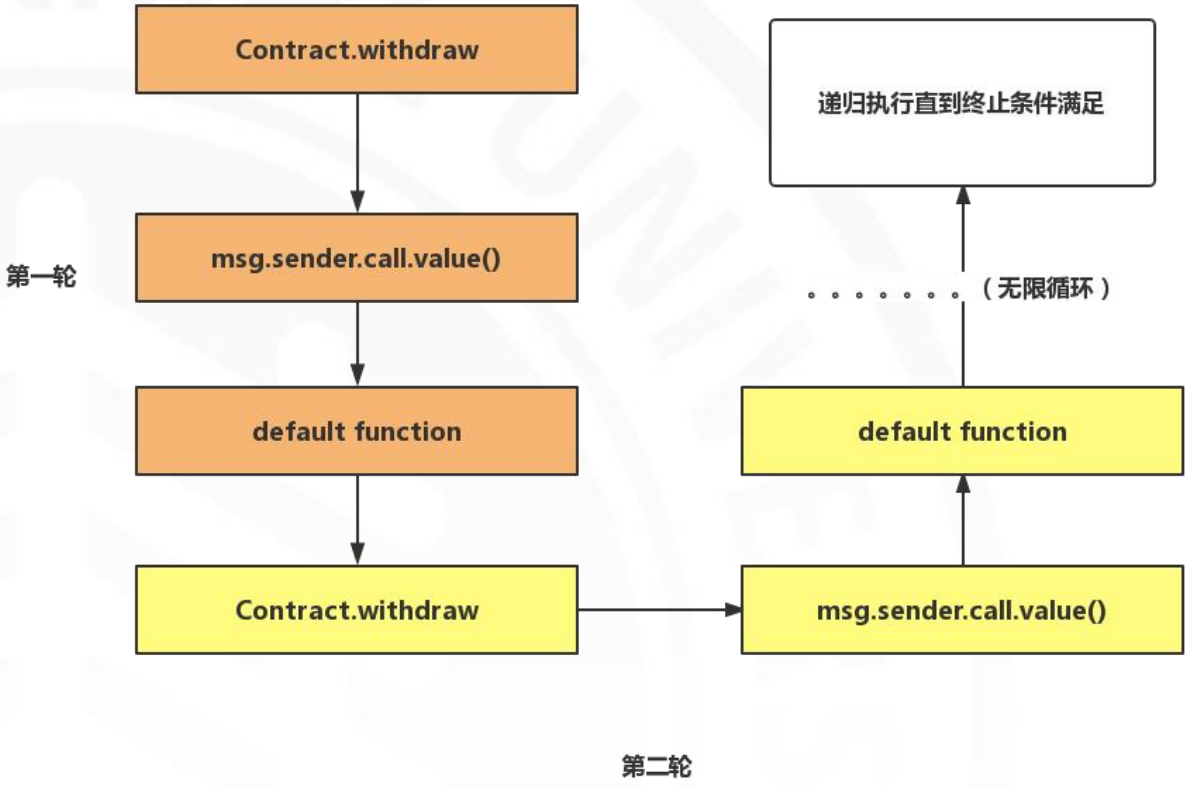
- The Dao 是一个去中心化的自治风险投资基金，通过发布的智能合约来募集资金，参与者可以通过投票的方式来投资以太坊上的应用，如果盈利，参与者就能获得回报。

DAO attack

- 攻击的漏洞:

1. 递归调用splitDAO函数
2. DAO资产分离后避免从TheDAO资产池中销毁

```
1   function withdrawBalance() {
2       amountToWithdraw = userBalances[msg.sender];
3       if (!(msg.sender.call.value(amountToWithdraw)())) { throw; }
4       userBalances[msg.sender] = 0;
5   }
6   function () {
7
8       vulnerableContract v;
9       uint times;
10      if (times == 0 && attackModeIsOn) {
11        times = 1;
12        v.withdrawBalance();
13       }
14
15    else { times = 0; }
16
17    }
```


Contract.withdraw
msg.sender.call.value()
default function
Contract.withdraw
第一轮
递归执行直到终止条件满足
（无限循环）
default function
msg.sender.call.value()
第二轮

研究生作业

- 分析以太坊的挖矿过程，特别是全面迁移为POS后的共识算法
- 分析DAO攻击原理和具体攻击流程

DAO 攻击

- 攻击需要具备的条件：

每一个合约有且仅有一个没有名字的函数，fallback函数。如果其他函数都不能匹配给定的函数标识符，则执行fallback函数。

以太坊中触发 fallback 函数中存在能消耗尽可能多的gas的函数存在。
recipient.call.value()

DAO攻击源码

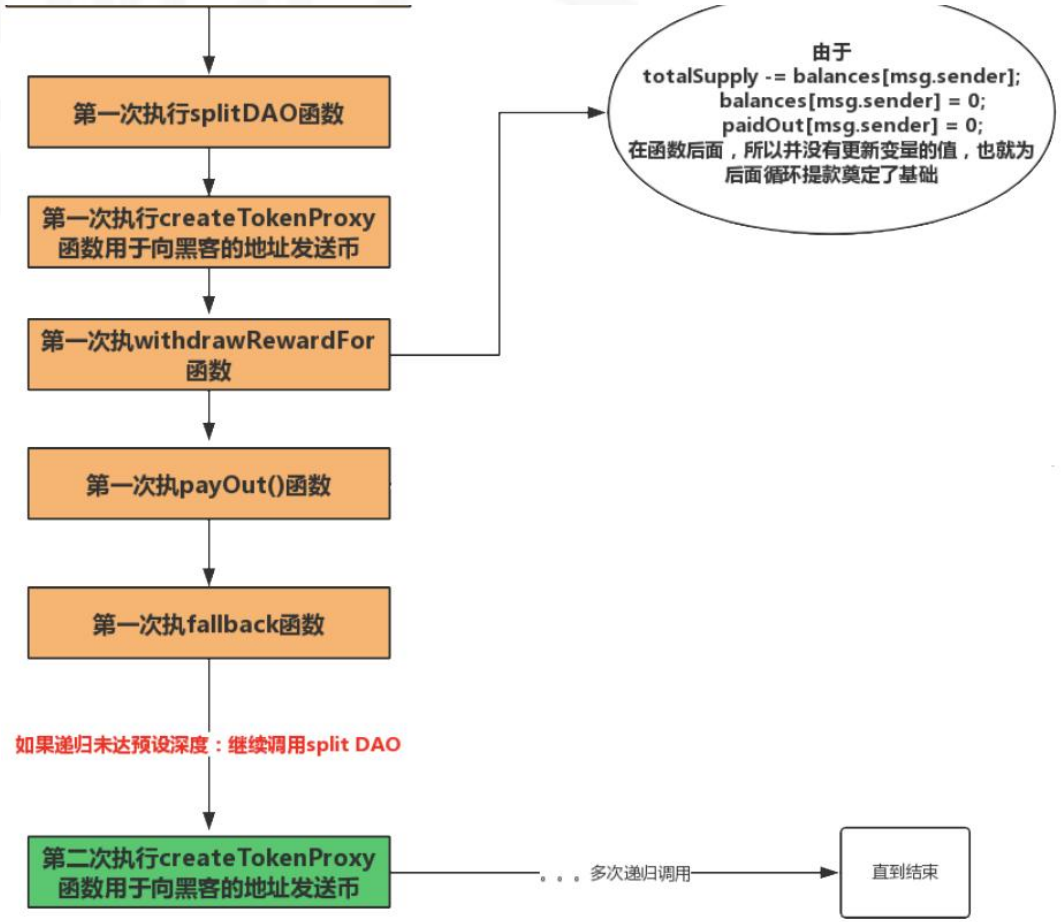
```
//用户选择分裂出去调用的函数
function splitDAO(uint _proposalID, address _newCurator) noEther onlyTokenholders returns (bool _success) {
    // ...
    //利用平衡数组计算应该转移多少代币 p是提案对象
    uint fundsToBeMoved = (balances[msg.sender] * p.splitData[0].splitBalance) / p.splitData[0].totalSupply;
    if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender) == false)

        throw;
    // ...
    // Burn DAO Tokens
    Transfer(msg.sender, 0, balances[msg.sender]);
    withdrawRewardFor(msg.sender);  // 转移对应的金额给用户
    // XXXXX Notice the preceding line is critically before the next few
    totalSupply -= balances[msg.sender];    // 相应变量更新
    balances[msg.sender] = 0;   // 余额置为0
    paidOut[msg.sender] = 0;
    return true;
}
```

DAO攻击源码

```
function withdrawRewardFor(address _account) noEther internal returns(bool _success) {
    if ((balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply < paidOut[_account])
        throw;
    uint reward = (balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply - paidOut[_account];
    if (!rewardAccount.payOut(_account, reward))    // XXXXX vulnerable
        throw;
    paidOut[_account] += reward;
    return true;
}
```

DAO攻击源码

```
function payOut(address _recipient, uint _amount) returns (bool) {
    if (msg.sender != owner || msg.value > 0 || (payOwnerOnly && _recipient != owner))
        throw;
    if (_recipient.call.value(_amount)()) { // XXXXX vulnerable
        PayOut(_recipient, _amount);
        return true;
    } else {
        return false;
    }
}
```


第一次执行splitDAO函数
第一次执行createTokenProxy函数用于向黑客的地址发送币
第一次执withdrawRewardFor函数
第一次执payOut()函数
第一次执fallback函数
如果递归未达预设深度：继续调用split DAO
第二次执行createTokenProxy函数用于向黑客的地址发送币
. . . 多次递归调用
直到结束
由于
totalSupply -= balances[msg.sender];
balances[msg.sender] = 0;
paidOut[msg.sender] = 0;
在函数后面，所以并没有更新变量的值，也就为后面循环提款奠定了基础

谢谢！