
目錄

前言	1.1
第1章 去中心化世界	1.2
去中心化	1.2.1
引言	1.2.1.1
分布式系统与去中心化	1.2.1.2
Paxos算法	1.2.1.3
云共识与发展	1.2.1.4
区块链	1.2.2
引言	1.2.2.1
什么是区块链	1.2.2.2
结构	1.2.2.3
分类	1.2.2.4
共识算法	1.2.3
引言	1.2.3.1
共识	1.2.3.2
智能合约	1.2.4
引言	1.2.4.1
比特币	1.2.5
引言	1.2.5.1
第2章 基于Go实现区块链	1.3
基本原型	1.3.1
引言	1.3.1.1
区块(Block)	1.3.1.2
区块链(Blockchain)	1.3.1.3
总结	1.3.1.4
工作量证明	1.3.2
引言	1.3.2.1
PoW	1.3.2.2
哈希	1.3.2.3
Hashcash算法	1.3.2.4

实现	1.3.2.5
总结	1.3.2.6
持久化和CLI	1.3.3
引言	1.3.3.1
数据库选型	1.3.3.2
BoltDB	1.3.3.3
数据库结构	1.3.3.4
序列化	1.3.3.5
持久化	1.3.3.6
查看blockchain内容	1.3.3.7
命令行接口(CLI)	1.3.3.8
总结	1.3.3.9
交易	1.3.4
引言	1.3.4.1
There is no spoon	1.3.4.2
比特币交易	1.3.4.3
交易输出(TXO)	1.3.4.4
交易输入(TXI)	1.3.4.5
先有蛋后有鸡	1.3.4.6
交易(Transaction)	1.3.4.7
工作量证明	1.3.4.8
未消费TXO(UTXO)	1.3.4.9
货币流通	1.3.4.10
总结	1.3.4.11
地址	1.3.5
引言	1.3.5.1
比特币地址	1.3.5.2
公钥加密	1.3.5.3
数字签名	1.3.5.4
椭圆曲线加密算法	1.3.5.5
Base58编码算法	1.3.5.6
地址	1.3.5.7
签名	1.3.5.8
总结	1.3.5.9

再论交易	1.3.6
引言	1.3.6.1
奖励	1.3.6.2
UTXO 集合	1.3.6.3
默克尔树(Merkle Tree)	1.3.6.4
P2PKH	1.3.6.5
总结	1.3.6.6
网络	1.3.7
引言	1.3.7.1
Blockchain 网络	1.3.7.2
节点角色	1.3.7.3
精简	1.3.7.4
实现	1.3.7.5
演练	1.3.7.6
总结	1.3.7.7
第3章 "集市"以太坊	1.4
以太坊概述	1.4.1
基于以太坊建立私有链	1.4.2
软/硬件环境	1.4.2.1
安装Geth	1.4.2.2
创建私有链	1.4.2.3
钱包应用：Mist	1.4.2.4
多节点互联	1.4.2.5
智能合约语言Solidity	1.4.3
构建自己的DApp	1.4.4
第4章 "大教堂"Hyperledger	1.5
Hyperledger概述	1.5.1
Hyperledger架构	1.5.2
构建联盟链应用	1.5.3
第5章 ZBChain平台	1.6
ZBChain白皮书	1.6.1
ZBChain架构	1.6.2
ZBChain应用	1.6.3

附录	1.7
公钥与私钥	1.7.1

Dummies for Blockchain

本书分为5章：

第1章主要介绍比特币、区块链、智能合约、共识算法等概念

第2章主要介绍如何使用Go语言实现一个简化的区块链，该章内容翻译自Ivan Kuznetsov的系列文章《Building Blockchain in Go》，原文地址：<https://jeiwan.cc>

第3章主要介绍以太坊平台，涉及概念、智能合约、DApp开发。

第4章主要介绍如何Hyperledger平台，涉及概念、架构、联盟链开发。

第5章主要介绍自研的ZBChain平台。

待续

互联网的发展从上个世纪70年代至今，已经经历了一个很长的发展过程。随着万物互联互通的理念越来越深入人们的思想，应用上的创新周期越来越短。从21世纪初的P2P、分布式系统的应用，云计算到大数据普及，机器学习到集体智慧，现实生活已经被“云”化。区块链是这发展过程中的一颗新星，作为一个“去中心化”系统，在这之上建立的共识机制能够达到“云”共识的目的。

本节作为开篇，会详细讲解区块链中的重要特性：去中心化，包含去中心化的来由来，分布式系统与中心化的关系，以及它体现的容错与共识特点，云共识与发展。

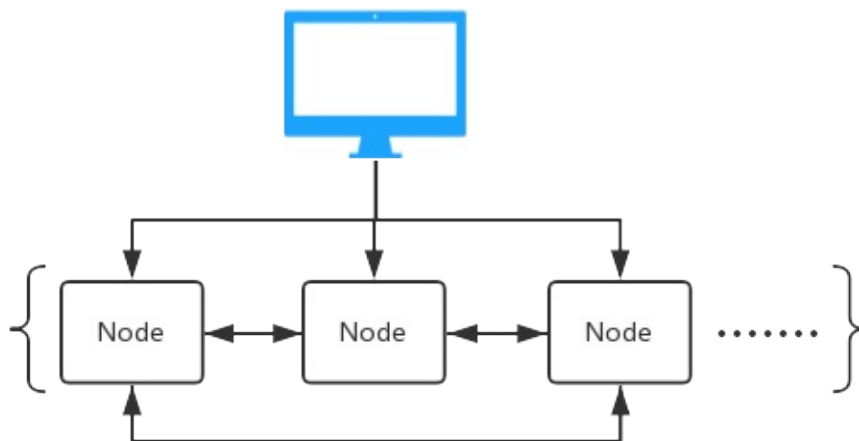
分布式系统与去中心化

分布式系统

分布式系统的起源，早在上个世纪80年代，就已经有计算机科学家开始围绕多台计算机协同工作的方式展开研究。什么是分布式系统呢？计算机发展和普及已经接近半个世纪，人们习惯了使用一台计算机操作系统后，理解多台计算机分散工作可能有些困难，从简单的概念上来说：

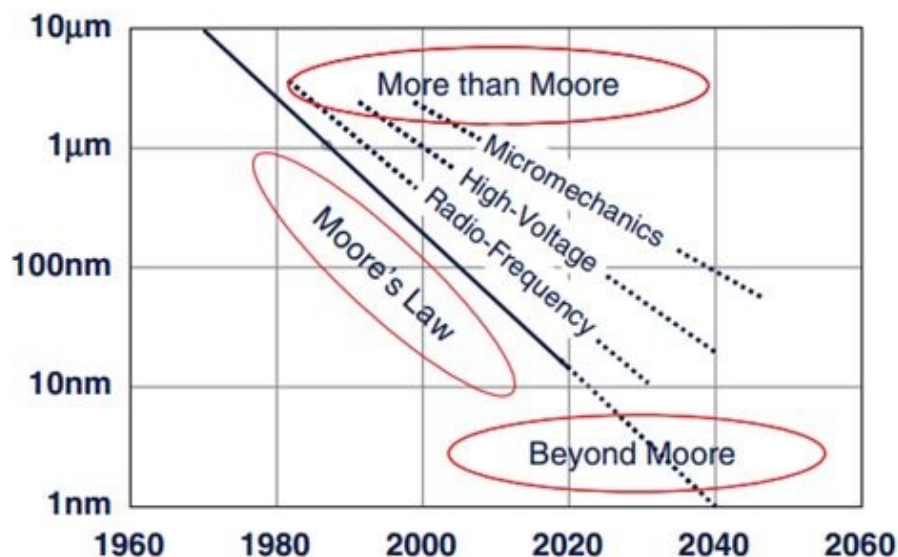
- 广义概念：一组计算机协同工作，对用户展现的是一个完整的系统。
- 狭义概念：一个由网络连接的分散的物理和逻辑资源的系统。

根据这两组概念不难看出，分布式系统的重要特点是多节点、多资源以及高性能。



如上图所示，在分布式系统下，**Node**作为一个系统的工作单元，它拥有完整的“最小系统”功能，每个节点之间都是相互连接、数据共享形成一个**Cluster**。因此对于用户来说，无论访问集群中哪个节点，都能够执行一套完整的系统命令；区别在于，一个集群的规模大小，以及对性能的影响。

实际上，摩尔定律的建立与失效、分布式系统的理论与应用，这两者之间的命运是息息相关的。从1965年，由Intel创始人之一的戈登·摩尔提出了著名的摩尔定律——**Moore's Law**，并且整个计算机行业始终以缩小晶体管体积作为计算机性能提升的手段。因为受到物理条件限制，这种方式的性能提升始终会有“终点”，而此时人类对于计算速度的要求可能远远不够。



2002年1月Moore在Intel开发者论坛上声明，定律预计在2017年终结，但实际上此时Intel仍在推进更小工艺技术，但成本急剧攀升。

摩尔定律延续的主要阻碍：

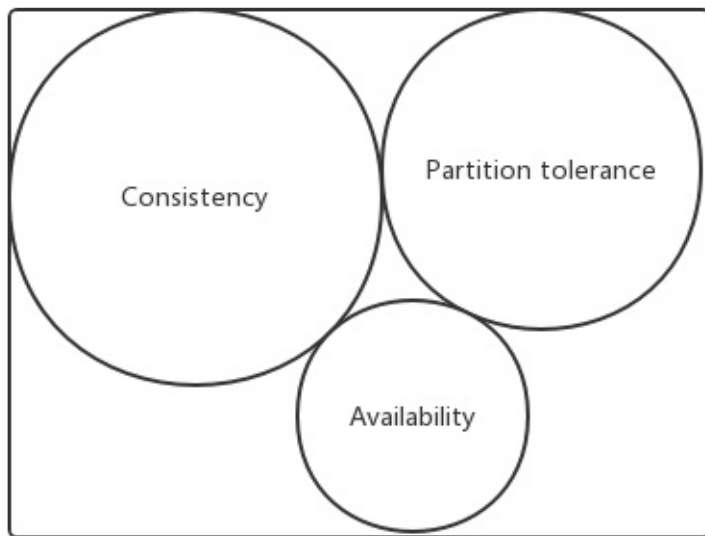
高能耗密度、原子尺寸的物理限制、高密度工艺的稳定性、比例缩小性能提升差越来越小，以及高昂的研发成本。

而有先见之明的计算机科学家，早在上世纪70年代开始就已经开始研究多处理器并行技术，到现在分布式系统的百花齐放，特别是Google的三大论文的发表，奠定了分布式系统的架构的大格局，之后该类型应用如雨后春笋般涌现。如，Google File System、MapReduce、Bigtable、Hadoop、Zookeeper、Spark、Storm、Hive、Hbase、Kafka、Redis等等，这些应用无一不是为了解决单一服务器解决不了的大量计算、海量存储、高效通信等问题。

分布式系统由于需要多台“计算机”协同工作，在逻辑处理、资源分配、信息安全以及性能上与我们常见单一计算机有很大的不同，而在这些问题中重要的就是各个节点之间的一致性，如何保证整个集群的一致性（Consistency），是确定一个系统是否是分布式系统的关键。

为此计算机科学家埃瑞克·布鲁尔在1999年提出了CAP原理（Consistency、Availability、Partition tolerance），分布式系统的一致性、可用性以及分区容错性。

- 一致性：分布式系统不同节点的备份数据，在同一时刻值是否相同。
- 可用性：分布式系统良好、高效的响应性能。
- 分区容错性：分布式系统中，若节点出现故障，是否能保证整个系统正常运行的可靠性。



他认为在一个分布式系统中，系统的这三个参数只能同时满足其中两个；也就是说，如果提高一致性和分区容错性，则可用性就会降低，同理为了可用性，需要降低一致性或分区容错性其中之一。因此三者相辅相成，架构师不要企图构建“完美”的分布式系统，要根据不同场景相互取舍。这一理论很快被业界所接受并广泛采用。

去中心化

去中心化作为新型词汇，与前期的云计算、大数据、人工智能一样被推上了网络信息时代的舞台，在这个时代扮演者“改变世界的”角色，它被誉为区块链存在的最核心因素，但到底是什么是去中心化，各界众说纷纭，存在很多异议。

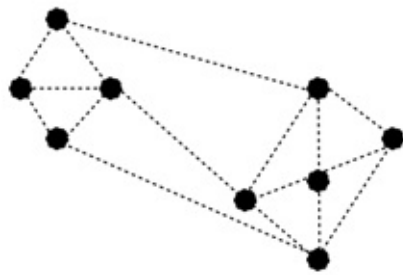
抛开应用场景，从概念上来讲：

- 广义概念：在多个参与者的系统中，不存在特殊权值的个体，信息的采纳由多数个体确认并被集体认可。
- 狭义概念：在多个节点的系统中，每个节点的数据相同，且系统操作具有强一致性。

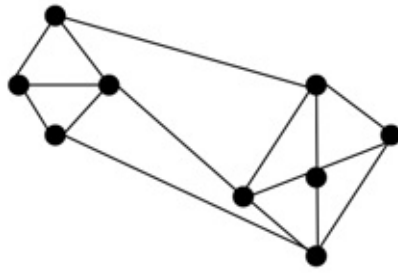
因此，去中心化是每个个体参与和数据一致性的体现，它的实现可以有效避免任何一个权威性的中心点参与、发布、左右结果，也不用担心信息的伪造、篡改和否认。

去中心化的系统与之前讲述的分布式系统有何不同呢？

其中，关键在于概念以及应用场景的不同，导致对节点数据的状态需求不同。分布式系统的数据副本是分散、分片存储，而去中心化系统的数据是完整副本，因此衍生出不同的系统机制。



(a) Distributed



(b) Decentralized

如上图可以看出，分布式系统中的连接使用虚线，因为每次请求被执行或分配的节点有限，为了保证有效资源利用，不会使用到所有节点，例如HDFS、Spark等等，同时它们也是串行化器（Serializer）的一种体现。

串行化器实际上是保证节点状态能同步的一种方法，有序的发布命令，让节点之间不易发生冲突。但是有单点故障问题（Single Point of Failure）。

而去中心化图中的实线代表，每一个请求（Update、Delete等）的执行会波及整个系统的所有节点，如：Zookeeper，又或是HyperLedger和Ethereum等基于区块链的服务。

实际上去中心化与分布式系统有着相似的主体结构，因此分布式系统的一些概念、特点、特性对于区块链也是适用的，如：一致性、可用性、分区容错性以及CAP原则。

需要注意的是，在广义概念中添加了“共识”的理念；而对于狭义来说，就是保证多个节点之间的一致性。

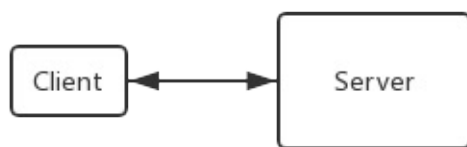
去中心化和共识是两个不同的概念，两者结合才是真正意义上的“去中心化”，去中心化有了共识才具备了完全的可信度，否则只是狭义概念的数据共享，没人知道这些共享的数据是否真实，或者被集体认可，仅仅只是拥有这些信息；共识有了去中心化，就具备了安全性，系统很难被攻破，信息也难被篡改；再加上特殊的数据结构，而这一切的实现就是区块链。

实际上，对于现阶段区块链发展，就是因为有了良好的共识机制，才会被世界广泛关注。不同的共识算法，受上述理论的影响，就会有不同的效果，这些内容会在后续共识算法中讲述。

Paxos算法

Paxos算法是解决分布式系统共识问题的重要算法，同样也衍生了大多数区块链共识算法，由Leslie Lamport在1989年提出这个名字，其实它的出生要早几年，对于它的历史，在这里不做详述。而Paxos算法对于现代互联网的发展作用，借用Google在2006年Chubby论文中的一句话：“事实上，到目前为止我们所遇到用于解决异步共识问题的算法，其核心都是Paxos”。

为了很好的讲述Paxos算法的内容，我们从简单结构模型开始——客户端/服务器（C/S）。



上图的功能是，Client给Server发送请求，Server接收到信息后，执行并返回结果。

但如何应对C/S之间的消息传递过程中出现消息丢失呢？如何避免重复发送同一个消息呢？

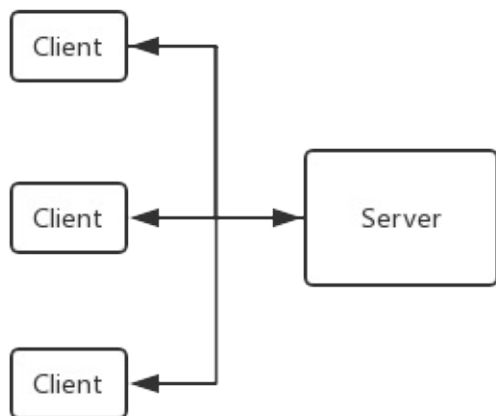
实际上如果我们在传输过程中添加确认机制（Acknowledgements），也就是说，每次Client在发送请求后，若Server接收到请求则会发送一个AckMessage返回给Client，如果超时还未接收到该消息，则Client补发请求；同时为了避免Server重复执行同一个消息，需要Client在发送的消息中添加一个MessageID，用于识别是否是重复消息。

Acknowledgements :

- Client向Server发送一条带MessageID的命令。
- Server接收到命令，会返回一条确认消息AckMessage。
- 如果Client在一段时间内没有收到确认消息，则重新发送该命令。

该算法的应用非常广泛，常见的有TCP协议，应用服务有Storm确认消息是否发送的Ack函数等等。

但是这又出现了另一个问题，如果有多个Client呢？



这样的结构带来一个新的问题，如果出现消息延迟，那Server端接收到的命令顺序也会不一样。

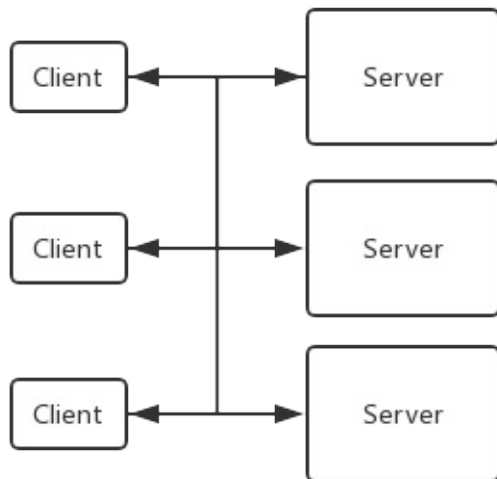
为了解决这个问题，Server端提供了一个锁机制（Locking）。每一个Client只有在拿到Server端的锁（Lock），才能够发送请求消息，没有拿到锁的Client需要等待其他Client释放锁，并重新抢占锁。这就是两段式协议（Two Phase Protocol）的雏形，通过两段才能发送/执行消息，保证了在同一时间只能有一个Client可以操作Server。避免了消息延迟、多客户端抢占Server带来的问题。

Locking :

- Client向Server请求Lock。（第一段）
- 如果Client获得Lock，则发送消息给Server。（第二段）
- 如果Client没有获得Lock，则等待其他Client释放Lock，并重新获取Lock。

实际上，这个问题也是一致性问题的解决，在经典数据库的系统中，两段提交协议（2PC）就是这种机制的实现。

但是，我们再次扩展模型，增加Server的个数，一个Client，需要发送消息到不同的Server时，这个两段式协议就出现了很大的问题。



每个Server都有各自的Lock，一个Client在提交时，需要获取所有的Lock才能发送消息。

如果Client宕机还未完全释放所有的Lock，如果Client只能获取到部分Lock，会不会出现死锁？

在这样的情况下如何保证各个Server的消息一致性呢？或者又如何保证数据的同步呢？

伴随着这些问题，Paxos算法中提出了一个替代Lock的概念：票据（Ticket），其特性如下。

Ticket：

- 由Server端发出最新的t，无论之前是否已发过。
- t可以过期，过期作废。

正常情况下，一个Client会先获取每个Server的最新的Ticket——t，并在请求的消息中携带该t，消息结构是最新t和命令c（t，c）。如果消息中的t不是该Server最新的t，Client会重新获取t。

所以无论Client是否宕机都不会影响其他Client的操作，因为每个Server只接收携带最新t的消息。

同时，为了避免Server端发生宕机，我们认为当一个Client收到超过一半以上的Server确认的最新t，就可以向所有Server发送确认执行的命令消息。

简化Paxos算法：

T，C为Client端的ticket和command参数；Ticket由每个Client维护，每次增加，由Server端验证、保存。

Server端最大ticket T.max，存储的命令携带ticket T.store，命令 C.store。

初始化，

Client	Server
T=0，C=c0	T.max=t，T.store=t，C.store=c1

请求，

Client	Server
向所有Server发送请求票据，T=T+1，并发送给所有Server。	
	if (T>T.max) { T.max = T; return prepare(T.store, C.store); }
if(获得半数以上Server回复prepare){ 选择最大的T.store的(T.store，C.store); if(T.store>0){ C=C.store; } 向回复的Server发送消息propose(T,C); }	
	获得propose请求。 if(T=T.max){ C.store=C; T.store=T; return success; }

执行，

Client	Server
if(获得半数以上Server回复success){ 向所有Server发送execute(C); }	
	获得execute请求。 if(C=C.store){ 执行命令C; T.store = 0; return result(T.store, C.store); }

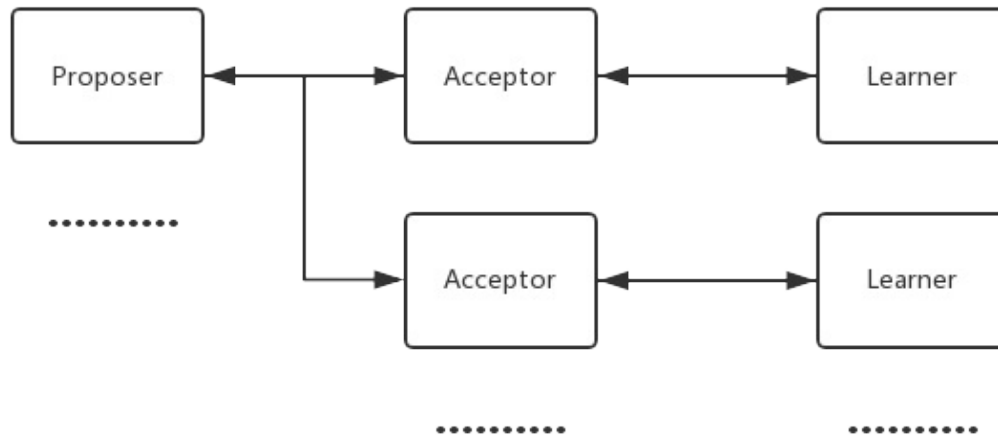
以上是Paxos算法的骨干逻辑，其中T.store作为是否已有准备好的命令存储的标志，若T.store不为零，则需要当前客户端提交执行已存储的命令C.store。这样如果一个Client执行到一半，另一个Client介入时就会继续执行上一个存储的Client命令，根据最终返回result来判断自己的请求是否被执行。

实际上，完整的Paxos算法中包含了Proposer、Acceptor和Learner角色，它们拥有着不同的功能。

Proposer：提案者或申请人，提出要表决的提案（Value 对应信息）。

Acceptor：接受者，接受了一个提案后就会把它记录下来。

Learner：学者，接受者接受了那个提案，他就会学习并执行那个提案。



步骤如下：

1. Proposer向有一定法定人数（quorum）的Acceptor发出新提案请求。
2. Acceptor确定该新提案的编号大于已记录的提案编号（如果没有已记录的提案则编号为0），记录新提案编号，并返回已记录的提案。
3. Proposer如果有一半以上的Acceptor响应，选择其中已记录的提案的最大编号的提案并使用新提案的编号，如果编号为0则使用新提案，并向返回该信息的Acceptor发出Prepare提案请求。
4. Acceptor判断准备请求的提案编号是否等于最大提案编号（新提案编号），如果是则接受该提案，并返回已接受该提案信息。
5. Proposer接收到超过一半Acceptor发出的接受提案消息，则确定有提案需要被执行（此时可能是提案也有可能是前一个已接受的提案）；发出Execute命令。
6. Acceptor执行提案，发送给Learner学习提案内容。

算法总体上可以分为两大步骤prepare和commit（发出准备提案和执行提交操作），而在实现过程中需要考虑很多问题，如出现多个Proposer同时争抢最大提案编号（ticket），这样Acceptor中记录的最大提案编号永远无法确定，甚至陷入死循环；大多数分布式系统也不会把Proposer定义为Client，处于安全等问题考虑，提供Clients访问Proposer，由Proposer提出提案；同时，Paxos算法也没有提供信息内容的可靠性保障等等。

Paxos算法是一个保证分布式系统一致性的基础算法，在应用过程中，面对各种各样的问题，我们借鉴该算法，并在此之上进行改进。在区块链中，Paxos算法同样也是借鉴的基础算法之一，但是仅仅只能让私有链能获得最大性能，对于联盟链和公有链则不能适用，因此，这将引出本章第二节和第三节的内容。

云共识

云共识，通过去中心化系统实现的不同区块链之间的互访机制。

区块链的发展从原先的Bitcoin——区块链1.0到现有的区块链2.0，短短不过几年时间，区块链应用已经快速发展到各行各业。

从理论角度来看，两个版本的最大区别在于去激励层，在区块链2.0版本中没有了激励层，对于Bitcoin来说就是没有“代币”了。

可以想象为此带来的革新是革命的，在去掉“代币”后，留下的是一个可以广泛应用的区块链架构，保留了去中心化、防伪、防篡改、安全、可追溯等优点，整个系统的运行不再依赖“代币”的驱动，由参与者共同维护。在技术驱动下，区块链2.0的应用前景更加广泛，对于上述优点，区块链都可以提供相对有效完美的解决方案。

已经有机构在利用“云”的虚拟化平台之上，建立一个区块链应用平台，也称为BaaS。致力于在同一个平台系统下提供多租户、多协议、多模式的服务。用户可以在BaaS之上建立自己的区块链服务，并向客户提供该服务；同已商业运行的PaaS平台一样，避免了小企业或非专业企业的部署难题。

很多大公司都在致力于公有链的构建，

区块链（BlockChain）的出现导致了去中心化概念的崛起，同时这样也保证了参与者的“公平性”，同时省去了很多中间环节，可以看成是甲乙双方的直接对接，由其他参与者“做证”。但如何解释什么是区块链？区块链系统有哪些部分组成？

本节将讲述区块链的定义与结构，区块链现有的类型以及特点，区块链应用。

概念

区块链概念是由日裔中本聪在2008年中提出的，目的是实现点对点的电子现金系统——比特币（Bitcoin）。该系统讲述了他对电子货币的构想，将不通过第三方作为中心来进行交易，并确保交易的可信、安全。中本聪是比特币的开发者兼创始人，不过很可惜，后来他退隐出了这个平台。

区块链的理念和P2P（Peer to Peer）是有本质区别的，尽管两者都有去中心化的概念，也用到一些算法，但是两者解决不同维度的问题。区块链是解决共识问题，P2P是解决资源共享。例如A和B是借贷关系，P2P的状态是A知道B有资金，A和B之间达成协议，平台上的其他用户并不关心此次交易，只关心B资金余量；而区块链的状态是A借B的资金，A和B之间达成协议，并且要告知平台上的所有用户，大家作为见证：记录A和B此时的资金，记录此次交易。

对于区块链来说比特币只是去中心化的起点，它的协议局限了它的功能，因为其中只有一种符号——比特币，合约也非常简单，满足正常的货币交易。但实际上，这些特点局限了它的发展。

后续的以太坊和超级账本的出现，都是区块链的进一步发展，弥补了早期区块链扩展性的问题，更加符合推广和应用。

图：区块链1.0和2.0的对比 <http://blog.csdn.net/sportshark/article/details/53364690>

区块链 V1.0	区块链 V2.0

- 广义概念：区块链是去中心化结构、安全传输，并由多方达成共识、参与、记录的一种存储方式。
- 狭义概念：区块链是能够保证每个节点间一致性的链式结构存储的系统。

链式的存储结构，表示每一个节点存储参与者提案的记录是以区块的提交顺序连续存储。就像一个铁链，每一环就代表一个存储的数据。

图：区块链的存储结构（一层一层的区块）

也就是说，区块链在应用过程中，无论采用什么样的共识算法、智能合约、安全协议，必须保证每一个在链上的参与者都参与到每一次提议与记录中来，并有同样效力的决定权。

根据概念，区块链应具有以下三个核心能力：去中心化、防篡改、可追溯。

图：区块链核心以及每个核心的实现（共识（paxos、Pow、PoS、DPos、PBFT），签名认证、链式存储（一致性哈希，超立方体网络、DHT和Churn））

- 去中心化：整个服务没有中心，意味着整个服务无需访问第三方机构，达成共识后，可以直接点对点的对接，具有一定的公平、公开的特点。但对外界来说，整个服务中的每

个参与节点都是“中心”，同时也构成了一个“中心”。因此，是否真的是去中心化，还是半去中心化、伪去中心化要看提交者所部署的位置以及职责。

- 防篡改：主要指的是已存储的区块防止被非法修改，通过非对称加密等实现，但其实，在整个提交过程中，既要防止请求被篡改，也要防止非法消息的提交。
- 可追溯：依靠区块链的链式存储结构，每一次提交区块都是顺序执行。因此可以通过链式结构追溯到任意一次提交的信息，不用担心任何一个参与者逾期抵赖。

结构

<http://www.doit.com.cn/p/274630.html>

为了满足区块链的特性，从架构上来说，需要具备数据层、网络层、核心层、服务层、用户层。

区块是区块链操作的最小存储单元，每一个区块都必须具备区块头与体，以便于适合区块链系统的运行。区块的结构如下：

(校对"工信部发布 区块链 数据格式规范") *具体参考一下比特币代码什么的*

区块体结构：

数据项	描述
Blocksize (区块大小)	区块开始到结束的字节长度
Blockheader	包含具体区块信息的描述

数据项 描述 长度

Blocksize (区块大小) 到区块结束的字节长度

Blockheader (区块头) 包含6个数据项

Transaction counter (交易数量)

Transactions (交易)

交易列表 (非空)

区块头描述：

数据项 目的

Version (版本) 区块版本号

hashPrevBlock (前一区块的Hash) 前一区块的256位Hash值

hashMerkleRoot Merkle (根节点Hash值) 基于一个区块中所有交易的256位Hash值

Time (时间戳) 从1970-01-01 00:00 UTC开始到现在，以秒为单位的当前时间戳

Bits (当前目标的Hash值) 压缩格式的当前目标Hash值

每个区块都包括了一个被称为魔法数的常数、区块的大小、区块头、区块所包含的交易数量及部分或所有的近期新交易。在每个区块中，对整个区块链起决定作用的是区块头。

分类

区块链按照应用场景可以分为：私有链、公有链、联盟链，三个类型的应用场景不同，而不是解释它的属性。

私有链：一般适用于一个团体内使用，容错、安全可以酌情考虑，如果节点可信度较高，在选择共识算法上可以考虑无“不可控节点”（拜占庭）问题存在的paxos算法，实际上，这样也带来了性能的提升。这里的节点是proxy服务器（消息转发的服务器，非用户机器），应用功能类似zookeeper，如果直接接入或用户信用度不高，建议不要使用单纯的一致性算法。

公有链：适用于公开使用，所有参与者都是不可控的，每个节点都是直接的参与者，并且对待每个参与者都是无差别的，这样平台需要面对更多的问题，例如节点频繁加入、退出平台，或发送虚假消息，甚至被黑客劫持。因此平台对安全、容错性要求较高，同时“复杂”的共识算法也会导致整个共识过程非常缓慢，如，PoW。

联盟链：适用于多个团体之间的链，参与者在进入服务平台之前，需要进行认证，确定参与者的属性后，才能使用平台。公有链也可以进行认证，但是并不会给参与者打上某种属性的标签。联盟链具有一定的约定性，可以接受“多中心化”的共识算法的介入，如果团体之间并不认可，也可以采取完全中心化的共识算法。

封闭式、开放式、半开放式

在区块链中，共识机制是其核心功能，它决定了整个系统的一致性、可用性以及分区容错性，并且提供独有的特性，来满足不同场景的需求。合格的共识算法既要快速达成共识，也要保证好容错能力，特别是节点在共识过程出现宕机，也要保证不影响共识结果的一致性。目前已构成区块链的共识算法有很多，这些算法实现的共识机制应用在多个领域中，但其大多数是基于Paxos理念的实现，在此基础上加以优化，添加功能，解决问题等等，例如：PoW、PoS、DPoS、PBFT算法等。

本章节主要讲述共识的特点，影响达成共识的情况。在广泛应用的算法中，选取两个经典的PoW和PBFT算法详细讲述。

共识算法

<http://blog.csdn.net/paxhujing/article/details/51605635>

共识（Consensus）是共同认知的表现，区块链的共识机制，其实就是对应分布式系统中的一致性，但相比分布式系统，区块链的共识算法更加丰满。因为，现有大多数分布式系统对于用户来说是透明的，多构建主从结构；因此，集群内部具有很高的可控性；而区块链的共识机制，为了体现去中心化特点，就需要所有注册用户，在“集群”系统内部开放信息的读写功能。为防止用户对信息进行伪装、篡改等等问题，算法的稳定性、安全性需要考虑的更加全面，同时这也是区块链能被广泛关注和应用的优点。

在达成共识的过程中，需要解决各种各样的问题，

First Chapter

GitBook allows you to organize your book into chapters, each chapter is stored in a separate file like this one.

Blockchain是21世纪最具革命性的技术之一，它仍在不断成长发展，潜力无限。本质上来讲，Blockchain是一个分布式数据库。但Blockchain的独特之处在于其是一个公开的数据库，而非私有数据库，每个人都可以拥有部分甚至整个数据库。如果想在数据库中添加一个新记录，需要征得其他数据库拥有者的同意。此外，Blockchain技术使数字货币和智能合约成为可能。

本文将基于Go语言构建简化版的blockchain，来实现数字货币。

让我们从“blockchain”中的“block”开始整个旅程。block存储价值信息，例如，比特币中的block存储交易信息。除此之外，block中包含一些其他信息，如版本、时间戳和上一个Block的hash值等

本文中，我们并不会按照比特币规范实现一个完整的blockchain，而是实现一个简化版本，仅仅包含一些重要的信息。block结构如下所示：

```
type Block struct {
    Timestamp    int64
    Data         []byte
    PrevBlockHash []byte
    Hash         []byte
}
```

Timestamp表示Block被创建的时间戳，**Data**表示block中实际的价值信息，**PrevBlockHash**表示上一个Block的hash值，**Hash**表示该block本身的Hash值。比特币规范中，Timestamp、PrevBlockHash、Hash存储在独立的数据结构中，作为block的头信息；Data存储在另一个独立的数据结构种。本文中，我们将它们放在一个数据结构中，从而简化实现。

Hash值用于确保blockchain的安全。Hash计算是计算敏感的操作，即使在高性能电脑也需要花费一段时间来完成计算(这也就是为什么人们购买高性能GPU进行比特币挖矿的原因)。blockchain架构设计有意使Hash计算变得困难，这样做是为了加大新增一个block的难度，进而防止block在增加后被随意修改。

现在，我们首先**SetHash**方法，将Block中的字段信息组合后，生成该block的SHA-256 Hash值：

```
func (b *Block) SetHash() {
    timestamp := []byte(strconv.FormatInt(b.Timestamp, 10))
    headers := bytes.Join([][]byte{b.PrevBlockHash, b.Data, timestamp}, []byte{})
    hash := sha256.Sum256(headers)

    b.Hash = hash[:]
}
```

接下来，实现**NewBlock**方法用于创建一个Block：

```
func NewBlock(data string, prevBlockHash []byte) *Block {
    block := &Block{time.Now().Unix(), []byte(data), prevBlockHash, []byte{}}
    block.SetHash()
    return block
}
```

OK，block实现完成啦！

现在，让我们来实现blockchain吧。本质上，blockchain仅仅是具备某种特殊结构的数据库：有序，反向链接链表。这意味着，block按照插入的顺序存放，同时每个block都保存指向上一个block的链接。这种结构保证可以快速获取最新插入的block同时获取它的hash值。

Go语言中，可以通过slice和map来实现该结构。slice(有序)用于保存有序的hash值，map(无序)用于保存hash->block对。对于我们的blockchain原型，目前不需要根据hash值来获取block，因此仅仅使用slice即可满足需求。blockchain结构如下：

```
type Blockchain struct {  
    blocks []*Block  
}
```

这就是我们的Blockchain，怎么样够简单吧😊

接下来，我们事先AddBlock方法，用于将block添加到blockchain中：

```
func (bc *Blockchain) AddBlock(data string) {  
    prevBlock := bc.blocks[len(bc.blocks)-1]  
    newBlock := NewBlock(data, prevBlock.Hash)  
    bc.blocks = append(bc.blocks, newBlock)  
}
```

新增一个block的前提是另一个block已经存在，但是一开始blockchain中并没有任何block。因此，在任何blockchain中都必须有一个特殊的block存在，称之为**GenesisBlock**。下面实现**NewGenesisBlock**方法用于创建**GenesisBlock**：

```
func NewGenesisBlock() *Block {  
    return NewBlock("Genesis Block", []byte{})  
}
```

接下来，实现**NewBlockchain**方法，该方法会创建一个包含Genesis Block的blockchain：

```
func NewBlockchain() *Blockchain {  
    return &Blockchain{[]*Block{NewGenesisBlock()}}  
}
```

最后让我们看看Blockchain是否可以正常工作吧：

```
func main() {  
    bc := NewBlockchain()  
  
    bc.AddBlock("Send 1 BTC to Ivan")  
    bc.AddBlock("Send 2 more BTC to Ivan")  
  
    for _, block := range bc.blocks {  
        fmt.Printf("Prev. hash: %x\n", block.PrevBlockHash)  
        fmt.Printf("Data: %s\n", block.Data)  
        fmt.Printf("Hash: %x\n", block.Hash)  
        fmt.Println()  
    }  
}
```

输出：

```
Prev. hash:  
Data: Genesis Block  
Hash: aff955a50dc6cd2abfe81b8849eab15f99ed1dc333d38487024223b5fe0f1168  
  
Prev. hash: aff955a50dc6cd2abfe81b8849eab15f99ed1dc333d38487024223b5fe0f1168  
Data: Send 1 BTC to Ivan  
Hash: d75ce22a840abb9b4e8fc3b60767c4ba3f46a0432d3ea15b71aef9fde6a314e1  
  
Prev. hash: d75ce22a840abb9b4e8fc3b60767c4ba3f46a0432d3ea15b71aef9fde6a314e1  
Data: Send 2 more BTC to Ivan  
Hash: 561237522bb7fcfbccbc6fe0e98bbbde7427ffe01c6fb223f7562288ca2295d1
```

大功告成！

我们构建了一个简单的blockchain原型：目前该blockchain仅仅是一个block数组，每个block有一个指向上一个block的链接。而实际应用的blockchain更加复杂。在我们的blockchain中，添加新的block简单而快速，而实际应用中的blockchain添加block之前需要进行一些复杂计算操作（即工作量证明机制）。同时，blockchain是一个非单一决策者的分布式数据库。因此，添加新block需要被所有参与者同意（即一致性协议）。此外，在我们的blockchain中还没有任何Transaction信息！

在未来的文章中，我们讨论上述的所有特性。

在[基本原型](#)章节中，我们利用非常简单的数据结构构建了blockchain数据库。Block之间存在链式关系：每个block都保存上一个block的链接。但是，我们的blockchain实现存在一个严重的问题：添加block操作过于简单。而区块链和比特币系统的主旨是让新增block成为一个高成本的操作。接下来，我们将修改这个问题。

天道酬勤是blockchain的核心思想之一，即要想将block放到blockchain中，是需要付出一定努力的。这确保blockchain的安全性和一致性。与此同时，也会受到相应的奖赏（整个过程称作挖矿）。

这种机制与现实生活有异曲同工之妙：努力工作挣钱从而维持生计。在blockchain中，参与者（矿工）努力工作（挖矿）添加block进而获取奖励，从而维持整个blockchain网络。挖矿使block安全地加入blockchain，维持了整个blockchain的稳定。值得注意的是，挖矿是需要某种方式来证明的。

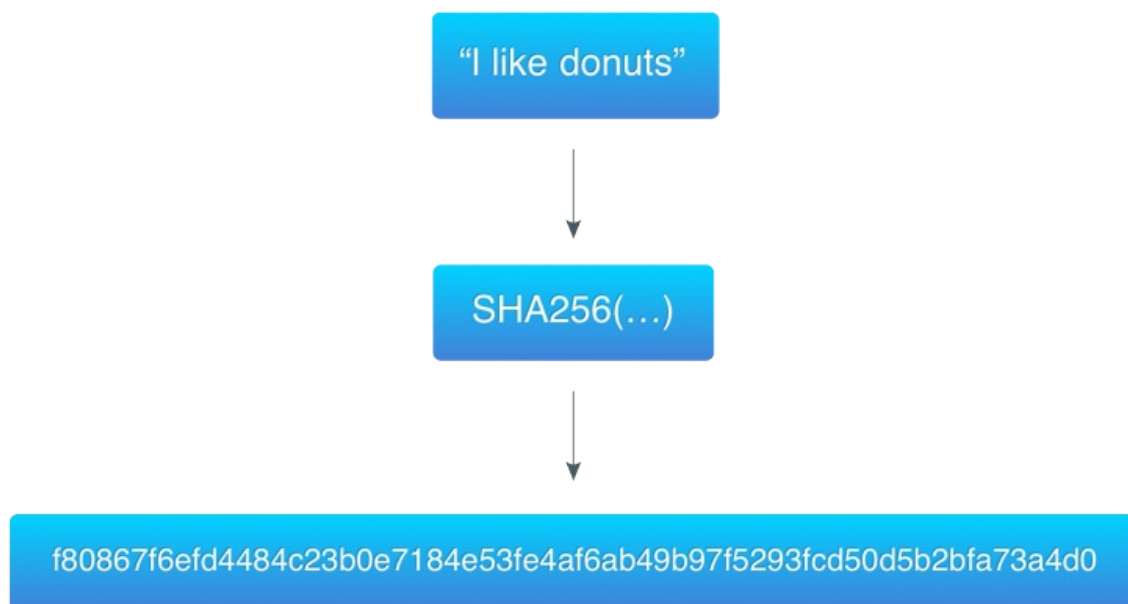
上述机制被称作“工作量证明”（即PoW）。PoW需要大量算力，即使是高性能计算机也不能非常快速地完成计算。此外，PoW随着时间的推移，难度也相应增加。

此外，PoW算法必须满足一个要求：工作量难度很高，但证明很容易。由于往往是其他人进行验证工作量，因此“快速证明”十分重要。

本节我们来讨论下hash，如果你熟悉hash，可以跳过此节。

hash是对于某个特定数据获得其hash值得过程。对于某个数据其hash值是唯一的。hash函数以任意长度数据作为输入，输出特定长度的hash值。对于hash来说，需要具备以下特性：

- 从hash值中无法恢复出原始数据。然而，hash并不是加密；
- 对于某个特定数据，hash值是唯一的；
- 输入数据即使改变一个字节，其hash值也会大不相同



Hash函数被广泛的用于数据一致性检查。例如，软件开发者提供软件包的同时，也会附带一个校验码。下载软件包后，可以使用hash函数计算出该软件包的hash值，并与软件开发者提供的校验码进行对比验证。

在blockchain中，hash用于保证block一致性。block的hash计算依赖于上一block的hash值。若想修改blockchain中的一个block，就需要把其后所有block全部重新计算一遍，这将消耗大量的算力（目前看是天文数字），这是得修改block变得非常非常困难，几乎不可实现。

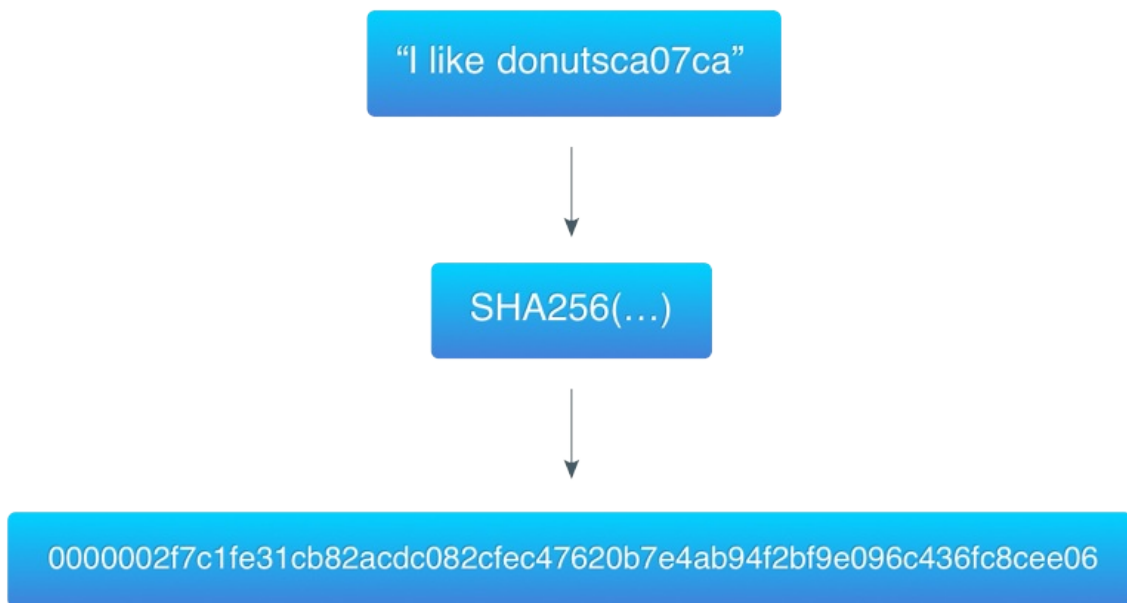
比特币使用Hashcash算法进行工作量证明，该算法一开始用于反垃圾邮件。该算法分为以下步骤：

1. 获取某种公开数据data（在反垃圾邮件场景下，使用收件人地址；在比特币中，使用block头信息）
2. 使用一个计数器counter，初始化为0
3. 计算data+counter的hash值
4. 检查hash值是否满足某种要求
 - i. 满足，结束
 - ii. 不满足，counter加1，然后重复3-4步骤

这是个暴力型算法：改变counter值，计算得到新hash值，判断该值是否满足要求，不满足，再改变counter值，再计算，再检查...如此反复尝试直到得到满足要求为止，要求很高的算力。

来看看hash要满足什么要求，hashcash最初实现中，hash值要满足“头20个bit全部为0”的要求。比特币设计中，hash值要求是动态变化的，随着时间和矿工的增多，算力要求也越来越多。

为了展示算法，使用上面示例中的数据（“I like donutsca07ca”），不停的计算该数据+counter的hash值，直到找到一个满足要求（“前3个字节为0”）的hash值为止，如下图所示：



“I like donutsca07ca”中的ca07ca是十六进制格式的值，即counter从0递增到13240266时计算得到满足要求的hash值。

OK，纸上谈兵到此为止，接下来实现该机制。首先，让我们来定义挖矿的难度：

```
const targetBits = 24
```

比特币中，“target bits”在block头信息中，表示该block被挖到的难度。我们并不实现难度自调整算法，而是定义一个全局的难度。

“target bits”只要小于256（因为我们要使用sha256算法计算hash值）即可，我们设置为24。这意味着，当hash值的前24位均为0即满足要求。当“target bits”过大时，满足要求将非常困难，因为符合条件的数据总数量变得非常小，因此需要更长时间计算才有可能得到满足要求的hash值。“target bits”的值越大难度越高，值越小难度越低。

```
type ProofOfWork struct {
    block    *Block
    target   *big.Int
}

func NewProofOfWork(b *Block) *ProofOfWork {
    target := big.NewInt(1)
    target.Lsh(target, uint(256-targetBits))

    pow := &ProofOfWork{b, target}

    return pow
}
```

ProofOfWork结构包括一个block指针和一个target指针。“target”表示困难度，其类型为big.Int，之所以使用big结构是为了方便和hash值做比较检查是否满足要求。

NewProofOfWork方法中，将target初始化为1，然后将其左移（256-target），其值为：

```
0x10000000000000000000000000000000000000000000000000000000
```

该值在内存中占用29字节，接下来将其与之前示例中的hash值进行比较：

```
0fac49161af82ed938add1d8725835cc123a1a87b1b196488360e58d4bfb51e3
000001000000000000000000000000000000000000000000000000000000
0000008b0f41ec78bab747864db66bcb9fb89920ee75f43fdaaeb5544f7f76ca
```

上述有3个hash值，第1行“I like donuts”计算得到hash值，第2行是目标值，第3行是基于“I like donutsca07ca”计算得到的hash值。第1行值大于目标值，不满足要求；第3行值小于目标值，满足要求，结束证明。

总结一下，hash值满足要求的规则：当hash值大于目标值，表示不满足要求；当hash值小于等于目标值，表示满足要求。

prepareData方法将block各个字段和nonce（counter值）作为输入，计算得到hash值作为输出：

```
func (pow *ProofOfWork) prepareData(nonce int) []byte {
    data := bytes.Join(
        [][]byte{
            pow.block.PrevBlockHash,
            pow.block.Data,
            IntToHex(pow.block.Timestamp),
            IntToHex(int64(targetBits)),
            IntToHex(int64(nonce)),
        },
        []byte{},
    )

    return data
}
```

准备工作一切完毕，接下来实现PoW核心算法：

```
func (pow *ProofOfWork) Run() (int, []byte) {
    var hashInt big.Int
    var hash [32]byte
    nonce := 0

    fmt.Printf("Mining the block containing \"%s\"\n", pow.block.Data)
    for nonce < maxNonce {
        data := pow.prepareData(nonce)
        hash = sha256.Sum256(data)
        fmt.Printf("\r%x", hash)
        hashInt.SetBytes(hash[:])

        if hashInt.Cmp(pow.target) == -1 {
            break
        } else {
            nonce++
        }
    }
    fmt.Print("\n\n")

    return nonce, hash[:]
}
```


最初，初始化变量：**hashInt**是hash值得整数形式；**nonce**是一个递增计数器，初始化为0。接下来，开始进行循环计算，**nonce**和**maxNonce**（设置为`math.MaxInt64`）进行比较，直到找到符合要求的hash值为止。

循环中进行如下操作：

1. 准备数据
2. 使用SHA-256算法计算hash值
3. 将hash值转换为整数
4. 整型hash值与目标值作比较

接下来，删除**SetHash**方法，并修改**NewBlock**方法：

```
func NewBlock(data string, prevBlockHash []byte) *Block {
    block := &Block{time.Now().Unix(), []byte(data), prevBlockHash, []byte{}, 0}
    pow := NewProofOfWork(block)
    nonce, hash := pow.Run()

    block.Hash = hash[:]
    block.Nonce = nonce

    return block
}
```

我们将**nonce**将入**Block**结构，这是因为**nonce**需要用于后续验证操作。**Block**结构结构如下：

```
type Block struct {
    Timestamp    int64
    Data         []byte
    PrevBlockHash []byte
    Hash         []byte
    Nonce        int
}
```

大功告成！让我们运行看看效果吧：

```
Mining the block containing "Genesis Block"
00000041662c5fc2883535dc19ba8a33ac993b535da9899e593ff98e1eda56a1

Mining the block containing "Send 1 BTC to Ivan"
00000077a856e697c69833d9effb6bdad54c730a98d674f73c0b30020cc82804

Mining the block containing "Send 2 more BTC to Ivan"
000000b33185e927c9a989cc7d5aaaed739c56dad9fd9361dea558b9bfaf5fbe

Prev. hash:
Data: Genesis Block
Hash: 00000041662c5fc2883535dc19ba8a33ac993b535da9899e593ff98e1eda56a1

Prev. hash: 00000041662c5fc2883535dc19ba8a33ac993b535da9899e593ff98e1eda56a1
Data: Send 1 BTC to Ivan
Hash: 00000077a856e697c69833d9effb6bdad54c730a98d674f73c0b30020cc82804

Prev. hash: 00000077a856e697c69833d9effb6bdad54c730a98d674f73c0b30020cc82804
Data: Send 2 more BTC to Ivan
Hash: 000000b33185e927c9a989cc7d5aaaed739c56dad9fd9361dea558b9bfaf5fbe
```

可以看到，所有block的hash值前3个字节均为0，同时生成block时需要花费一定时间。

目前还有一件事需要做，就是对工作量进行验证：

```
func (pow *ProofOfWork) Validate() bool {
    var hashInt big.Int

    data := pow.prepareData(pow.block.Nonce)
    hash := sha256.Sum256(data)
    hashInt.SetBytes(hash[:])

    isValid := hashInt.Cmp(pow.target) == -1

    return isValid
}
```

可以看到，验证过程需要nonce数据。

再运行一次，一切OK：

```
func main() {  
    ...  
  
    for _, block := range bc.blocks {  
        ...  
        pow := NewProofOfWork(block)  
        fmt.Printf("PoW: %s\n", strconv.FormatBool(pow.Validate()))  
        fmt.Println()  
    }  
}
```

输出：

```
...  
  
Prev. hash:  
Data: Genesis Block  
Hash: 00000093253acb814afb942e652a84a8f245069a67b5eaa709df8ac612075038  
PoW: true  
  
Prev. hash: 00000093253acb814afb942e652a84a8f245069a67b5eaa709df8ac612075038  
Data: Send 1 BTC to Ivan  
Hash: 0000003eeb3743ee42020e4a15262fd110a72823d804ce8e49643b5fd9d1062b  
PoW: true  
  
Prev. hash: 0000003eeb3743ee42020e4a15262fd110a72823d804ce8e49643b5fd9d1062b  
Data: Send 2 more BTC to Ivan  
Hash: 000000e42afddf57a3daa11b43b2e0923f23e894f96d1f24bfd9b8d2d494c57a  
PoW: true
```

我们的blockchain日趋成熟：现在添加block需要一定工作量，挖矿成为可能。但是仍然缺少一些关键特性：blockchain是非持久化的，没有钱包、地址、交易，缺乏一致性机制。所有这些特性将在后续章节实现。现在让我先尽情享受挖矿的乐趣吧！

目前为止，我们已经构建了一个具备工作量证明机制的blockchain。当前实现的blockchain仍然缺乏一些重要的特性。今天，我们会先将blockchain存储到数据库，然后实现一个命令行接口来操作blockchain。本质上blockchain是一个分布式数据库。现在我们先忽略“分布式”聚焦到“数据库”上。

到目前为止，我们实现的blockchain在每次程序运行时存储在内存中，不能重用或者与其他人共享该blockchain，因此需要使用一个数据来存储blockchain。

那么，使用哪种数据库呢？实际上，任何一种数据库都可以。在比特币的论文中，对数据库的使用没有任何要求，使用哪种数据库完全取决于开发者的意愿。比特币发布之初使用的是LevelDB，那么我们选用BoltDB。

为什么选用BoltDB？理由如下：

- 足够简洁
- 基于Go实现
- 不需要服务端

BoltDB在Github上的说明：

*Bolt*是基于纯Go语言开发的KV存储，灵感来自于Howard Chu的LMDB项目。该项目目标是开发一个简单、快速、可靠的无服务端的数据库。*API*非常小巧和简洁，仅仅关注如何获取或设置数据，这就是全部。

听起来和我们的需求非常匹配！让我们花几分钟研究一下。

BoltDB是K-V存储，没有关系型数据库中类似表、行、列结构，数据以key-value对存储（类似GO语言中的map）。相似的key-value对存储在同一个bucket中，类似于关系型数据库中的Table。因此，为了获取一个value，需要知道其所在的bucket以及对应的key。

BoltDB中没有数据类，key和value都是字节数组。我们的blockchain数据均为struct，因此为了在BoltDB中存取数据，需要对struct进行序列化和反序列化。虽然JSON, XML, Protocol Buffers均可以到达相同的目的，我们还是决定采用encoding/gob库来完成此工作，一方面是因为使用简单，另一方面是因为该模块是Go标准库的一部分。

实现持久化逻辑前，需要先确定在数据库中如何存储数据。对此，我们计划参考比特币的实现。

简单来说，比特币使用下面两个**bucket**来存储数据：

1. **blocks**存储用于描述所有block的元数据信息。
2. **chainstate**用于存储blockchain的状态，包括所有交易元数据和部分元数据。

同时，出于性能考虑，每个block存放在独立的文件，这样获取单个block时不需要加载所有block。但为了简化，我们还是把所有数据存储在一个文件中。

在blocks bucket中，存储如下k-v对：

1. 'b' + 32-byte block hash -> block index record
2. 'f' + 4-byte file number -> file information record
3. 'l' -> 4-byte file number: the last block file number used
4. 'R' -> 1-byte boolean: whether we're in the process of reindexing
5. 'F' + 1-byte flag name length + flag name string -> 1 byte boolean: various flags that can be on or off
6. 't' + 32-byte transaction hash -> transaction index record

在chainstate bucket中，存储如下k-v对：

1. 'c' + 32-byte transaction hash -> unspent transaction output record for that transaction
2. 'B' -> 32-byte block hash: the block hash up to which the database represents the unspent transaction outputs

(设计原理请参考[这里](#))

由于目前实现还没有设计交易，因此我们仅需要实现**blocks** bucket。同时，就如上面所说，所有数据存储在一个单一的文件中，因此不需要任何和文件号相关的信心。所以，我们仅需要一下k-v数据：

1. 32-byte block-hash -> Block structure (serialized)
2. 'l' -> the hash of the last block in a chain

这就是我们实现持久化所需的相关信息。

若前所述，BoltDB仅仅存储字节数组，因此我们实现Serializable方式（基于encoding/gob库）用于序列化Block结构：

```
func (b *Block) Serialize() []byte {
    var result bytes.Buffer
    encoder := gob.NewEncoder(&result)

    err := encoder.Encode(b)

    return result.Bytes()
}
```

过程很简单：首先声明一个buffer；然后初始化一个gob编码block；最后返回编码后的字节数组。

接下来，我们需要一个反序列化函数，用于将一个字节数组解码为一个block：

```
func DeserializeBlock(d []byte) *Block {
    var block Block

    decoder := gob.NewDecoder(bytes.NewReader(d))
    err := decoder.Decode(&block)

    return &block
}
```

让我们来实现 **NewBlockchain** 函数，该函数创建一个blockchain实例同时添加genesis block，步骤如下：

1. 打开数据库文件
2. 检查是否有blockchain
3. 如果有blockchain
 - i. 创建一个blockchain实例
 - ii. 将blockchain的tip指向最新的block
4. 如果没有blockchain
 - i. 创建genesis block
 - ii. 存储在数据库中
 - iii. 将genesis block的hash值保存为最新的block hash值
 - iv. 创建一个新的blockchain实例，同时tip指向最新的block

代码如下：

```
func NewBlockchain() *Blockchain {
    var tip []byte
    db, err := bolt.Open(dbFile, 0600, nil)

    err = db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))

        if b == nil {
            genesis := NewGenesisBlock()
            b, err := tx.CreateBucket([]byte(blocksBucket))
            err = b.Put(genesis.Hash, genesis.Serialize())
            err = b.Put([]byte("1"), genesis.Hash)
            tip = genesis.Hash
        } else {
            tip = b.Get([]byte("1"))
        }

        return nil
    })

    bc := Blockchain{tip, db}

    return &bc
}
```

让我们一段一段来看实现。

```
db, err := bolt.Open(dbFile, 0600, nil)
```

上面代码是打开BoltDB文件的标准方式，如果文件不存在不会反馈错误。

```
err = db.Update(func(tx *bolt.Tx) error {
    ...
})
```

在BoltDB中，数据库操作都在一个事务中运行。BoltDB有两种事务：只读事务和读写事务。在这里，由于可能会存在添加genesis block的操作，因此我们使用读写事务（**db.Update(...)**）。

```
b := tx.Bucket([]byte(blocksBucket))

if b == nil {
    genesis := NewGenesisBlock()
    b, err := tx.CreateBucket([]byte(blocksBucket))
    err = b.Put(genesis.Hash, genesis.Serialize())
    err = b.Put([]byte("l"), genesis.Hash)
    tip = genesis.Hash
} else {
    tip = b.Get([]byte("l"))
}
```

上述代码是核心逻辑。首先，我们尝试获取一个bucket，如果bucket存在，则将tip设置为key(l)的值；如果bucket不存在，则创建genesis block、创建bucket、存储genesis block、将tip设置为genesis block的hash值。

此外，注意创建blockchain的方法有了变化：

```
bc := Blockchain{tip, db}
```

现在不再存储任何block，而仅仅存储blockchain的tip，同时为了避免程序运行过程中数据库反复被打开，因此blockchain中会存储已打开的数据库链接，**blockchain**的结构修改如下：

```
type Blockchain struct {
    tip []byte
    db  *bolt.DB
}
```

接下来，修改**AddBlock**方法：新增一个block会变得复杂一些：

```
func (bc *Blockchain) AddBlock(data string) {
    var lastHash []byte

    err := bc.db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        lastHash = b.Get([]byte("l"))

        return nil
    })

    newBlock := NewBlock(data, lastHash)

    err = bc.db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        err := b.Put(newBlock.Hash, newBlock.Serialize())
        err = b.Put([]byte("l"), newBlock.Hash)
        bc.tip = newBlock.Hash

        return nil
    })
}
```

我们来一段一段来看：

```
err := bc.db.View(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte(blocksBucket))
    lastHash = b.Get([]byte("l"))

    return nil
})
```

首先使用只读事务获取当前数据库中最新block的hash值。

```
newBlock := NewBlock(data, lastHash)
b := tx.Bucket([]byte(blocksBucket))
err := b.Put(newBlock.Hash, newBlock.Serialize())
err = b.Put([]byte("l"), newBlock.Hash)
bc.tip = newBlock.Hash
```

在挖到一个新block后，我们将序列化后的结果存储到数据库文件中同时将key(l)值更新为最新block的hash值。

完成！一切简单明了。

现在所有block都保存在数据库文件中，因此我们可以重新打开一个blockchain并且新增一个block。但我们现在失去一个很好的特性：我们不能输出blockchain的内容。让我们修改这个缺陷！

BoltDB允许遍历bucket中的所有key，但是key按照字节序的顺序来存储的。我们想按照block被加入的顺序来打印blockchain内容。此外，我们不想将所有block全部加载到内存，因此将实现一个迭代器来逐个遍历block：

```
type BlockchainIterator struct {
    currentHash []byte
    db          *bolt.DB
}
```

当想遍历blockchain时，我们会创建一个迭代器。迭代器包括当前遍历到的block的hash值和数据库链接。

```
func (bc *Blockchain) Iterator() *BlockchainIterator {
    bci := &BlockchainIterator{bc.tip, bc.db}

    return bci
}
```

一开始迭代器指向blockchain的tip，意味着将从顶到底、从新到旧的遍历blockchain。实际上，选择tip就好比blockchain投票。Blockchain可能有多个分支，最长的那个分支被当做主分支。获得tip后，就可以重建整个blockchain并且得到blockchain的长度。因此，某种程度上可以说tip是blockchain的标识。

BlockchainIterator仅仅做一件事：返回先一个block。

```
func (i *BlockchainIterator) Next() *Block {
    var block *Block

    err := i.db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        encodedBlock := b.Get(i.currentHash)
        block = DeserializeBlock(encodedBlock)

        return nil
    })

    i.currentHash = block.PrevBlockHash

    return block
}
```

数据部分到此为止！

目前为止，程序还没有提供任何交互接口，仅仅在main函数中调用**NewBlockchain**, **bc.AddBlock**等函数。现在让我们实现以下命令：

```
blockchain_go addblock "Pay 0.031337 for a coffee"
blockchain_go printchain
```

所有命令行命令通过**CLI**结构处理：

```
type CLI struct {
    bc *Blockchain
}
```

CLI的入口是**Run**函数：

```
func (cli *CLI) Run() {
    cli.validateArgs()

    addBlockCmd := flag.NewFlagSet("addblock", flag.ExitOnError)
    printChainCmd := flag.NewFlagSet("printchain", flag.ExitOnError)

    addBlockData := addBlockCmd.String("data", "", "Block data")

    switch os.Args[1] {
    case "addblock":
        err := addBlockCmd.Parse(os.Args[2:])
    case "printchain":
        err := printChainCmd.Parse(os.Args[2:])
    default:
        cli.printUsage()
        os.Exit(1)
    }

    if addBlockCmd.Parsed() {
        if *addBlockData == "" {
            addBlockCmd.Usage()
            os.Exit(1)
        }
        cli.addBlock(*addBlockData)
    }

    if printChainCmd.Parsed() {
        cli.printChain()
    }
}
```

我们使用**flag**标准库处理命令行参数：


```
addBlockCmd := flag.NewFlagSet("addblock", flag.ExitOnError)
printChainCmd := flag.NewFlagSet("printchain", flag.ExitOnError)
addBlockData := addBlockCmd.String("data", "", "Block data")
```

首先，创建两个子命令：**addblock**、**printchain**，**addblock**具有一个命令参数**-data**，**printchain**没有任何命令参数。

```
switch os.Args[1] {
case "addblock":
    err := addBlockCmd.Parse(os.Args[2:])
case "printchain":
    err := printChainCmd.Parse(os.Args[2:])
default:
    cli.printUsage()
    os.Exit(1)
}
```

接下来，我们检查用户输入的命令以及与之相关的命令参数：

```
f addBlockCmd.Parsed() {
    if *addBlockData == "" {
        addBlockCmd.Usage()
        os.Exit(1)
    }
    cli.addBlock(*addBlockData)
}

if printChainCmd.Parsed() {
    cli.printChain()
}
```

对于解析成功的应命令执行对应的函数。

```
func (cli *CLI) addBlock(data string) {
    cli.bc.AddBlock(data)
    fmt.Println("Success!")
}

func (cli *CLI) printChain() {
    bci := cli.bc.Iterator()

    for {
        block := bci.Next()

        fmt.Printf("Prev. hash: %x\n", block.PrevBlockHash)
        fmt.Printf("Data: %s\n", block.Data)
        fmt.Printf("Hash: %x\n", block.Hash)
        pow := NewProofOfWork(block)
        fmt.Printf("PoW: %s\n", strconv.FormatBool(pow.Validate()))
        fmt.Println()

        if len(block.PrevBlockHash) == 0 {
            break
        }
    }
}
```

整个过程与之前非常相似，唯一区别在于现在使用**BlockchainIterator**遍历blockchain。

最后，修改一下main函数：

```
func main() {
    bc := NewBlockchain()
    defer bc.db.Close()

    cli := CLI{bc}
    cli.Run()
}
```

需要注意的是无论是否有命令行参数，程序一运行就会创建一个blockchain。

一切OK，尝试运行下吧：

```
$ blockchain_go printchain
No existing blockchain found. Creating a new one...
Mining the block containing "Genesis Block"
000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b

Prev. hash:
Data: Genesis Block
Hash: 000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b
PoW: true

$ blockchain_go addblock -data "Send 1 BTC to Ivan"
Mining the block containing "Send 1 BTC to Ivan"
000000d7b0c76e1001cdc1fc866b95a481d23f3027d86901eaeb77ae6d002b13

Success!

$ blockchain_go addblock -data "Pay 0.31337 BTC for a coffee"
Mining the block containing "Pay 0.31337 BTC for a coffee"
000000aa0748da7367dec6b9de5027f4fae0963df89ff39d8f20fd7299307148

Success!

$ blockchain_go printchain
Prev. hash: 000000d7b0c76e1001cdc1fc866b95a481d23f3027d86901eaeb77ae6d002b13
Data: Pay 0.31337 BTC for a coffee
Hash: 000000aa0748da7367dec6b9de5027f4fae0963df89ff39d8f20fd7299307148
PoW: true

Prev. hash: 000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b
Data: Send 1 BTC to Ivan
Hash: 000000d7b0c76e1001cdc1fc866b95a481d23f3027d86901eaeb77ae6d002b13
PoW: true

Prev. hash:
Data: Genesis Block
Hash: 000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b
PoW: true
```

接下来，我们将实现地址、钱包、交易等概念。

交易是比特币的核心，而blockchain的唯一目的就是安全可靠地存储交易信息，确保创建交易后，没人可以修改该交易信息。今天，让我们来实现交易。由于交易是个非常大的主题，因此将分两部分介绍。第一部分实现交易的框架，第二部分实现更过细节。

Web应用一般需要创建如下数据表用于实现支付逻辑：账户表（**accounts**）、交易表（**transactions**）。**accounts**用于存储用户信息，如个人信息、账户余额；**transactions**用于存储资金的流动信息，如由谁支付给谁多少钱。比特币的实现则完全不同：

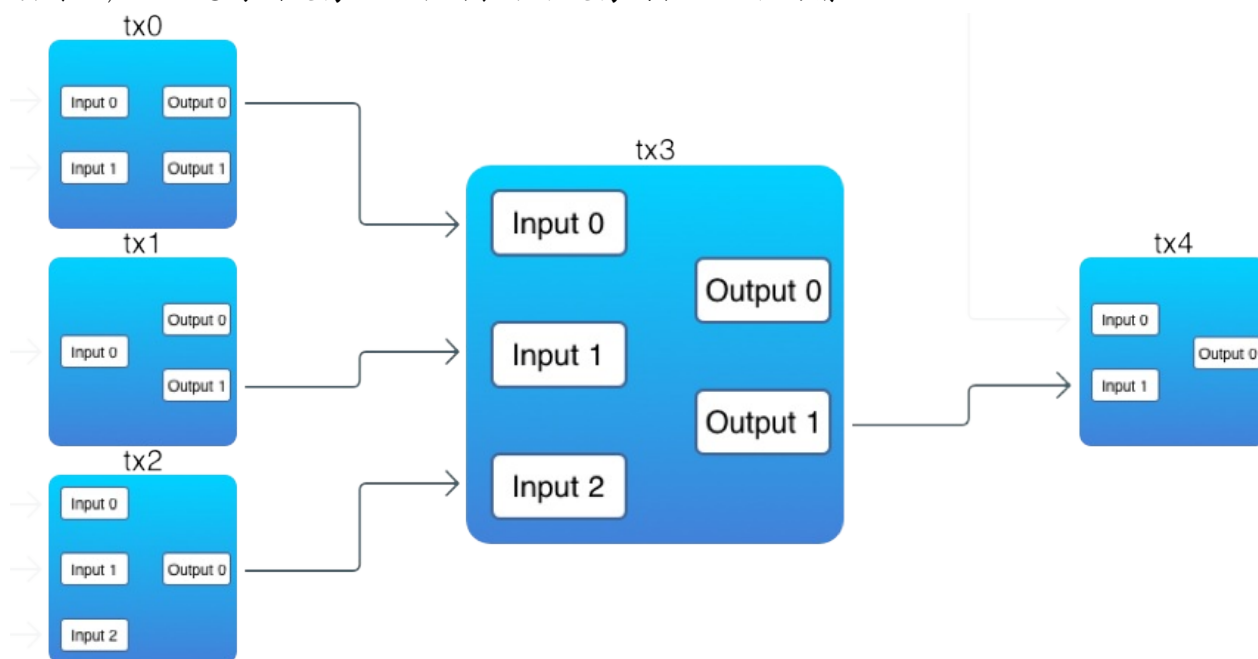
1. 没有账户
2. 没有余额
3. 没有地址信息
4. 没有货币信息
5. 没有付款人、收款人

由于**blockchain**是完全开放、公开的，因此我们不希望存储任何用户敏感信息。账户中不包含任何交易额信息。交易也不会将钱从一个账户转到另一个账户，也不存储任何账户余额信息。仅有的就是交易信息，交易信息到底有什么呢？

一个交易由多个输入和输出：

```
type Transaction struct {  
    ID    []byte  
    Vin   []TXInput  
    Vout  []TXOutput  
}
```

交易输入（下文称TXI）均与之前交易的输出（下文称TXO）相关联（存在一个例外情况，后续讨论）；TXO存储交易额。下图展示了交易间相互关联的情况：



需注意的是：

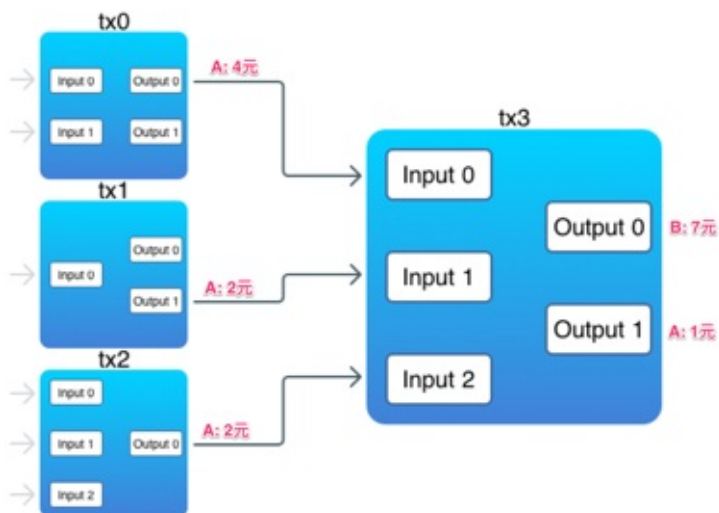
1. 有些TXO没有与任何其他TXI相关联
2. 某个TXI可能与之前多个TXO相关联
3. 一个TXI仅与一个TXO相关联

本文中，我们将使用“money”，“coins”，“spend”，“send”，“account”等术语用于描述交易过程，但比特币中就没有相关概念存在的。比特币通过一个脚本对交易额进行加锁，而仅仅加锁者才能解锁。

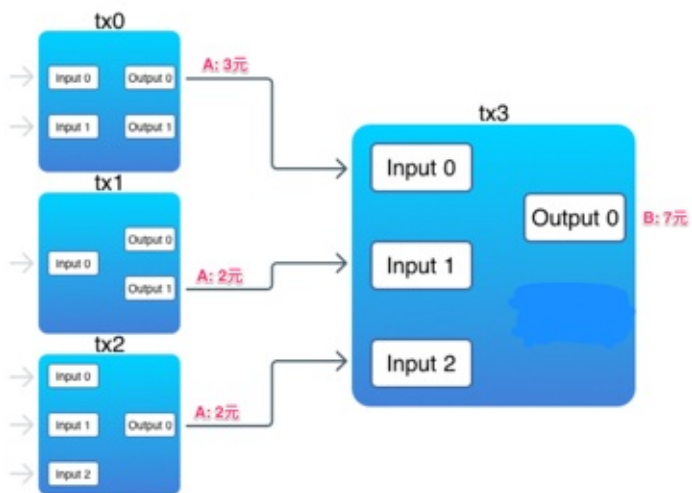
一个交易包括付款方和收款方，因此交易中的TXI都是付款方，而交易中的TXO有两种情况：若TXI的总额正好和所需值相等，那么交易只有一个TXO，该TXO属于收款方；若TXI的总额大于所需值，那么交易有两个TXO，一个属于收款方，一个属于付款方。

（FindUnspentTransactions方法利用这个特点来获取某个地址（账户）的UTXO）：

TXI大于所需值，有两个TXO



TXI等于所需值，有一个TXO



TXO结构如下：

```
type TXOutput struct {  
    Value      int  
    ScriptPubKey string  
}
```

TXO存储货币信息（TXOutput的Value字段），同时通过一个谜题进行锁定，谜题存放在**ScriptPubKey**字段中。比特币使用一个叫做script的脚本语言进行加锁和解锁操作，script中定义了加锁和解锁逻辑。该语言是非常底层的（这是有意为之，主要是为了避免黑客攻击或者误用）。对于该语言本身不进行过多讨论，详细可以参考[BitcoinScript](#)。

“比特币中，*value*字段存储的是*satoshis*数量，而不是*BTC*的数量。一个*satoshis*表示0.00000001 BTC，是最小的货币单位（如同一分钱）。”

由于尚未实现地址（钱包）概念，因此暂时不涉及脚本相关的逻辑，**ScriptPubKey**仅仅存储用户定义的字符串，而非钱包信息。

正因为由此脚本语言，比特币可以用作智能合约平台

值得注意的是：一个TXO作为一个整体使用，是不可分割的，不能只使用该TXO的一部分。如果该TXO的交易额大于所需，则发生“找零”，这同现实世界是一样的：用5块钱购买价值1块钱的商品，获得商品的同时还会获得4元找零；而不是把5块钱撕开成1/5和4/5，只是用4/5，这是不允许的。

TXI结构如下：

```
type TXInput struct {  
    Txid      []byte  
    Vout       int  
    ScriptSig string  
}
```

如上所述，TXI与之前的某个TXO相关联：**Txid**存储输出所属的交易的ID，**Vout**存储输出的序号（一个交易可以包括多个TXO）。**ScriptSig**存储一个脚本，与之关联的TXO的

ScriptPubKey就来源于该脚本，因此两者可以进行校验：如果校验正确，与之关联的TXO被解锁并生成新的TXO；如果校验不正确，TXO压根不能够被TXI所引用。该机制保证钱只能被其拥有者使用，而不能被其他人使用。

由于我们还没有实现地址（钱包），**ScriptSig**仅仅存储用户自定义的字符串而已，后续我们将实现公钥和签名核查。

综上所述，TXO存储交易额，同时拥有一个解锁脚本。每个交易至少拥有一个TXI和一个TXO。TXI的**ScriptSig**和TXO的**ScriptPubKey**进行校验，验证成功后可以解锁交易输出，并创建新的TXO。

那么问题来了：先有TXI还是先有TXO？

比特币中，TXI关联TXO的逻辑和经典的“先有鸡还是先有蛋”的问题是一样的。比特币是先有蛋（TXO）后有鸡（TXI），TXO先于TXI出现。

当blockchain的首个block（即genesis block）被挖到后，会生成一个coinbase交易。coinbase交易是一种特殊的交易，该TXI不会引用任何TXO，而会直接生成一个TXO，这是作为奖励给矿工的。

Blockchain以genesis block开头，该block生成blockchain中第一个TXO。由于之前没有任何交易，因此该TXI不会与任何TXO关联。

下面创建一个coinbase交易：

```
func NewCoinbaseTX(to, data string) *Transaction {
    if data == "" {
        data = fmt.Sprintf("Reward to '%s'", to)
    }

    txin := TXInput{[]byte{}, -1, data}
    txout := TXOutput{subsidy, to}
    tx := Transaction{nil, []TXInput{txin}, []TXOutput{txout}}
    tx.SetID()

    return &tx
}
```

coinbase仅有一个TXI，该TXI的Txid为空，Vout设置为-1，同时ScriptSig中存储的不是脚本，而仅仅是一个普通字符串。

比特币中，第一个coinbase交易包含如下信息“The Times 03/Jan/2009 Chancellor on brink of second bailout for banks”

Subsidy是挖矿的奖励值，比特币中，该奖励值是基于总block数量计算得到的。挖出genesis奖励50BTC，每挖出210000个block，奖励值减半。我们的实现中，该奖励值是一个常量。

从现在开始，每个block至少包含一个交易，**Block**结构中的**Data**字段江永**Transactions**字段代替。

```
type Block struct {
    Timestamp      int64
    Transactions   []*Transaction
    PrevBlockHash []byte
    Hash           []byte
    Nonce          int
}
```

NewBlock和**NewGenesisBlock**方法也需要修改：

```
func NewBlock(transactions []*Transaction, prevBlockHash []byte) *Block {
    block := &Block{time.Now().Unix(), transactions, prevBlockHash, []byte{}, 0}
    ...
}

func NewGenesisBlock(coinbase *Transaction) *Block {
    return NewBlock([]*Transaction{coinbase}, []byte{})
}
```

CreateBlockchain也需要修改：

```
func CreateBlockchain(address string) *Blockchain {
    ...
    err = db.Update(func(tx *bolt.Tx) error {
        cbtx := NewCoinbaseTX(address, genesisCoinbaseData)
        genesis := NewGenesisBlock(cbtx)

        b, err := tx.CreateBucket([]byte(blocksBucket))
        err = b.Put(genesis.Hash, genesis.Serialize())
        ...
    })
    ...
}
```

CreateBlockchain接受一个地址，该地址会挖到genesis block因此将获取奖励。

为了保证blockchain的一致性和可靠性，PoW算法必须要考虑block中的交易信息，**ProofOfWork.prepareData**需要做如下修改：

```
func (pow *ProofOfWork) prepareData(nonce int) []byte {
    data := bytes.Join(
        [][]byte{
            pow.block.PrevBlockHash,
            pow.block.HashTransactions(), // This line was changed
            IntToHex(pow.block.Timestamp),
            IntToHex(int64(targetBits)),
            IntToHex(int64(nonce)),
        },
        []byte{},
    )

    return data
}
```

现在使用**pow.block.HashTransactions()**代替**pow.block.Data**：

```
func (b *Block) HashTransactions() []byte {
    var txHashes [][]byte
    var txHash [32]byte

    for _, tx := range b.Transactions {
        txHashes = append(txHashes, tx.ID)
    }
    txHash = sha256.Sum256(bytes.Join(txHashes, []byte{}))

    return txHash[:]
}
```

我们遍历所有交易，获取每个交易的hash值，然后将所有交易的hash值组合后最终得到一个代表block中整个交易的hash值，计算整个block的hash值时会引用该值。

比特币使用了一种更加优雅的技术：以**Merkle Tree**（默克尔树）表示block中的所有交易，PoW算法中使用树根的hash值。该方法仅仅需要树根hash值而不用下载所有交易信息，因此可以非常快速检查block中是否包含某些交易，

使用Ivan创建一个blockchain：

```
$ blockchain_go createblockchain -address Ivan
00000093450837f8b52b78c25f8163bb6137caf43ff4d9a01d1b731fa8ddcc8a

Done!
```

我们收到了首次挖矿的奖励，但是我们如何知道余额信息呢？

需要找到所有未消费的TXO（下文称为UTXO）。未消费表示这些TXO未被任何TXI所引用，在上图中一下TXO是UTXO：

1. tx0, output 1;
2. tx1, output 0;
3. tx3, output 0;
4. tx4, output 0.

下面是加锁/解锁方法：

```
func (in *TXInput) CanUnlockOutputWith(unlockingData string) bool {  
    return in.ScriptSig == unlockingData  
}  
  
func (out *TXOutput) CanBeUnlockedWith(unlockingData string) bool {  
    return out.ScriptPubKey == unlockingData  
}
```

目前，仅仅将**ScriptSig**、**ScriptPubKey**与**unlockingData**作比较，待后续实现地址（钱包）后，会做进一步改进。

接下来查找包含UTXO的交易，过程有些复杂：


```

func (bc *Blockchain) FindUnspentTransactions(address string) []Transaction {
    var unspentTXs []Transaction
    spentTXOs := make(map[string][]int)
    bci := bc.Iterator()

    for {
        block := bci.Next()

        for _, tx := range block.Transactions {
            txID := hex.EncodeToString(tx.ID)

            Outputs:
            for outIdx, out := range tx.Vout {
                // Was the output spent?
                if spentTXOs[txID] != nil {
                    for _, spentOut := range spentTXOs[txID] {
                        if spentOut == outIdx {
                            continue Outputs
                        }
                    }
                }

                if out.CanBeUnlockedWith(address) {
                    unspentTXs = append(unspentTXs, *tx)
                }
            }

            if tx.IsCoinbase() == false {
                for _, in := range tx.Vin {
                    if in.CanUnlockOutputWith(address) {
                        inTxID := hex.EncodeToString(in.Txid)
                        spentTXOs[inTxID] = append(spentTXOs[inTxID], in.Vout)
                    }
                }
            }
        }

        if len(block.PrevBlockHash) == 0 {
            break
        }
    }

    return unspentTXs
}

```

交易存储在block中，我们需要遍历blockchain中的每一个block：

```

if out.CanBeUnlockedWith(address) {
    unspentTXs = append(unspentTXs, tx)
}

```

如果TXO是被指定地址锁定的，该TXO会作为候选TXO继续进行处理：

```
if spentTXOs[txID] != nil {
    for _, spentOut := range spentTXOs[txID] {
        if spentOut == outIdx {
            continue Outputs
        }
    }
}
```

对于已经被TXI引用的TXO，不做处理；对于未被TXI引用的TXO，即UTXO，将包含其的交易保存到交易列表中。对于coinbase交易，不需要遍历TXI，因为其TXI不会引用任何TXO。

```
if tx.IsCoinbase() == false {
    for _, in := range tx.Vin {
        if in.CanUnlockOutputWith(address) {
            inTxID := hex.EncodeToString(in.Txid)
            spentTXOs[inTxID] = append(spentTXOs[inTxID], in.Vout)
        }
    }
}
```

最终，该函数返回包含UTXO的交易列表。通过接下来的函数，进一步处理，最终返回TXO列表。

```
func (bc *Blockchain) FindUTXO(address string) []TXOutput {
    var UTXOs []TXOutput
    unspentTransactions := bc.FindUnspentTransactions(address)

    for _, tx := range unspentTransactions {
        for _, out := range tx.Vout {
            if out.CanBeUnlockedWith(address) {
                UTXOs = append(UTXOs, out)
            }
        }
    }

    return UTXOs
}
```

OK！下面实现**getbalance**命令：

```
func (cli *CLI) getBalance(address string) {
    bc := NewBlockchain(address)
    defer bc.db.Close()

    balance := 0
    UTXOs := bc.FindUTXO(address)

    for _, out := range UTXOs {
        balance += out.Value
    }

    fmt.Printf("Balance of '%s': %d\n", address, balance)
}
```

账户余额就是属于该账户的所有UTXO的总和。

Ivan挖到genesis block后，其账户余额为10：

```
$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 10
```

货币流动起来才创造价值，为了达成此目的，我们需要创建一个交易，将该交易放到一个block中，然后通过挖矿挖到（PoW）该block。目前为止，我们仅仅实现了coinbase这种特殊的交易，下面我们实现通用的交易：

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain) *Transaction {
    var inputs []TXInput
    var outputs []TXOutput

    acc, validOutputs := bc.FindSpendableOutputs(from, amount)

    if acc < amount {
        log.Panic("ERROR: Not enough funds")
    }

    // Build a list of inputs
    for txid, outs := range validOutputs {
        txID, err := hex.DecodeString(txid)
        if err != nil {
            return nil
        }

        for _, out := range outs {
            input := TXInput{txID, out, from}
            inputs = append(inputs, input)
        }
    }

    // Build a list of outputs
    outputs = append(outputs, TXOutput{amount, to})
    if acc > amount {
        outputs = append(outputs, TXOutput{acc - amount, from}) // a change
    }

    tx := Transaction{nil, inputs, outputs}
    tx.SetID()

    return &tx
}
```

创建TXO前，我们需要找到该账户之前所有交易中的UTXO（**FindSpendableOutputs**函数），然后对于每一个UTXO创建一个TXI引用它，直到交易额满足要求为止。此时所有被引用的UTXO变为TXO。随后，在当前交易我们需要创建TXO：

- 如果引用的UTXO的交易额小于所需，则当前交易创建失败；
- 如果引用的UTXO的交易额等于所需，则当前交易仅有一个TXO；
- 如果引用的UTXO的交易额大于所需，则当前交易有两个TXO。

FindSpendableOutputs函数基于**FindUnspentTransactions**实现：

```

func (bc *Blockchain) FindSpendableOutputs(address string, amount int) (int, map[string]
[]int) {
    unspentOutputs := make(map[string][]int)
    unspentTXs := bc.FindUnspentTransactions(address)
    accumulated := 0

Work:
    for _, tx := range unspentTXs {
        txID := hex.EncodeToString(tx.ID)

        for outIdx, out := range tx.Vout {
            if out.CanBeUnlockedWith(address) && accumulated < amount {
                accumulated += out.Value
                unspentOutputs[txID] = append(unspentOutputs[txID], outIdx)

                if accumulated >= amount {
                    break Work
                }
            }
        }
    }

    return accumulated, unspentOutputs
}

```

此方法遍历账户所有UTX，遍历过程中进行交易额累加，同时计算累加交易额是否满足需求，当满足需求时停止遍历，并返回UTXO列表以及累加交易额。

接下来修改**Blockchain.MineBlock**方法：

```

func (bc *Blockchain) MineBlock(transactions []*Transaction) {
    ...
    newBlock := NewBlock(transactions, lastHash)
    ...
}

```

最后，实现**send**命令：

```

func (cli *CLI) send(from, to string, amount int) {
    bc := NewBlockchain(from)
    defer bc.db.Close()

    tx := NewUTXOTransaction(from, to, amount, bc)
    bc.MineBlock([]*Transaction{tx})
    fmt.Println("Success!")
}

```

消费意味着创建一个交易，然后挖一个block存储该交易，并将block添加到blockchain中。但是比特币的做法不同：比特币不会为一个新的交易马上去挖矿，而是会先将新交易缓存内存池mempool，当矿工即将挖矿时，从内存池中将所有交易取出，整体放到block中，并添加到blockchain。

让我们尝试进行一些交易：

```
$ blockchain_go send -from Ivan -to Pedro -amount 6
00000001b56d60f86f72ab2a59fadb197d767b97d4873732be505e0a65cc1e37

Success!

$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 4

$ blockchain_go getbalance -address Pedro
Balance of 'Pedro': 6
```

Ivan给了Pedro 6元，此时Ivan余4元，Pedro余6元。然后，Pedro给Helen 2元，Ivan给Helen 2元：

```
$ blockchain_go send -from Pedro -to Helen -amount 2
00000099938725eb2c7730844b3cd40209d46bce2c2af9d87c2b7611fe9d5bdf

Success!

$ blockchain_go send -from Ivan -to Helen -amount 2
000000a2edf94334b1d94f98d22d7e4c973261660397dc7340464f7959a7a9aa

Success!
```

此时，Helen锁定两个TXO：一个来自Pedro，一个来自Ivan。Helen给Rachel 3元，此时Ivan余2元，Pedro余4元，Helen余1元，Rachel余3元：

```
$ blockchain_go send -from Helen -to Rachel -amount 3
000000c58136cffa669e767b8f881d16e2ede3974d71df43058baaf8c069f1a0

Success!

$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 2

$ blockchain_go getbalance -address Pedro
Balance of 'Pedro': 4

$ blockchain_go getbalance -address Helen
Balance of 'Helen': 1

$ blockchain_go getbalance -address Rachel
Balance of 'Rachel': 3
```

一切OK！尝试一种异常情况：Pedro给Ivan5元，但是Pedro只有4元，消费失败。交易失败前后，Pedro和Ivan的余额未发生变化。

```
$ blockchain_go send -from Pedro -to Ivan -amount 5
panic: ERROR: Not enough funds

$ blockchain_go getbalance -address Pedro
Balance of 'Pedro': 4

$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 2
```

哇！经历重重困难，我们现在终于实现了交易了！不过，我们仍然缺少了比特币这类数字货币的一些关键特性：

1. 地址（账户/钱包）：仍没有基于私钥的地址
2. 奖励：挖矿应该给予响应的奖励
3. UTXO集合：在我们的实现中，为了获取余额需要遍历整个blockchain，效率非常低。此外，如果我们想验证交易，也会很慢。UTXO集合可以解决上述问题。
4. Mempool：在我们的实现中，一个block仅仅包含一个交易，利用率不高。Mempool用于缓存多个交易然后打包到一个block中，从而提高利用率。

在[交易](#)章节中实现了交易，交易不涉及用户特征信息，但仍旧有一些数据用于表示TXO的拥有者。之前我们使用普通字符串来标识地址，下面将实现正式的地址概念了。

一个比特币地址的例子：[1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa](#)。这是世界上首个比特币地址，据说属于比特币发明人中本聪。比特币地址是公开的，如果你想转给某人一些BTC，那么就需要知道其地址。这些地址并不代表钱包，仅仅是具备可读格式的公钥。在比特币世界中，你的ID是一个密钥对（公/私钥），该密钥对需要保存在你的电脑或者其他你可以直接存取到它的设备。比特币通过密码学算法来创建密钥对，从而保证在不能直接存取该密钥情况下，没人可以动你的钱。下面让我们来讨论一下这些算法。

公钥加密算法使用一对密钥：公钥和私钥。公钥可以给其他人，而私钥不应该给其他人，你的私钥就代表你本人。

本质上，比特币钱包就是一个密钥对。当你安装钱包客户端或者比特币客户端创建一个新地址时，一个密钥对将会被创建。在比特币世界，谁拥有密钥，谁就可以掌控属于该密钥的钱。

公钥和私钥仅仅是一些随机字节序列，人们无法直接读取，因此比特币使用一个算法用于将公钥转换成可读的字符串。

如果之前使用过比特币钱包应用，你可能会有个短密码，该短密码用于生成私钥，该机制在 [BIP-039](#) 实现。

那么比特币是如何校验TXO的所有者呢？

在数学和密码学中，数字签名算法保证：

1. 当数据从发送者传给接受者时，数据不会被篡改
2. 数据被某个发送者创建
3. 该发送者不能拒绝发送数据

通过签名算法对数据进行处理后得到一个签名，该签名用于后续验证数据使用。数字签名使用私钥签名，公钥验证。

数据签名需要以下信息：

1. 要签名的数据
2. 私钥

数据签名过程生成一个签名，签名存储于TXI中。为了校验签名，需要如下信息：

1. 已经被签名的数据
2. 签名信息
3. 公钥

认证过程简而言之就是：核查用私钥加密数据后生成的签名，是否可以用来生成公钥。

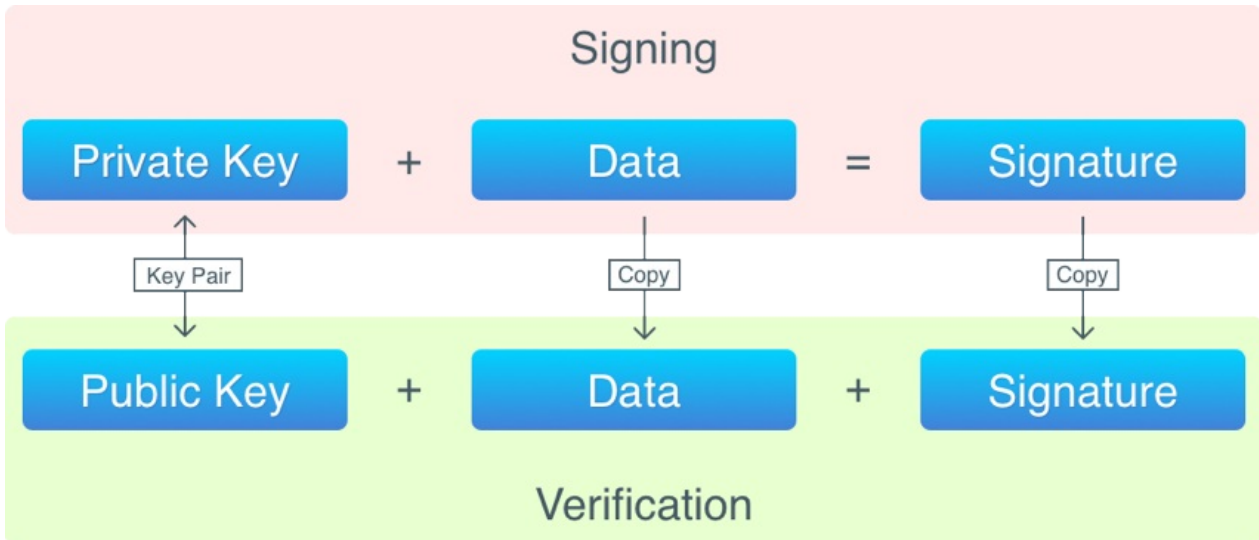
数字签名并不是加密，因此无法从签名中重构或者说解密出原始数据。与`hash`类似：通过`hash`算法生成表示数据的唯一`hash`值。签名和哈希是密钥对：正因为有密钥对，使得签名认证成为可能。

但是，密钥对也可以用于加密数据：私钥加密，公钥解密。但是比特币中并不使用该特性。

在比特币中，交易的创建者会对每个交易进行签名。交易在放到`block`之前，需要对每个交易进行验证。验证意味着：

1. 检查TXI是否有权限使用与之关联的TXO
2. 检查该交易的签名是否正确

形象来说，数据签名和签名校验的过程如下：



让我们回顾一个交易的整个声明周期：

1. 伊始，仅仅有genesis block，该block包含一个coinbase交易。Coinbase交易不包含真正的TXI，因此不需要签名。Coinbase交易中的TXO包含哈希后的公钥（使用RIPEMD16(SHA256(PubKey)算法）。
2. 当有人消费时，新的交易被创建。该交易的TXI引用之前交易的TXO。每个TXI存储一个未哈希的公钥，和整个交易的签名。
3. 比特币中收到此交易的其他节点将进行验证，将核查：TXI中公钥的hash值是否和其引用的TXO的公钥匹配（这保证发送者只能使用属于其拥有的货币）；签名是否正确（这保证该交易是被一个真实的货币所有者创建的）。
4. 当一个矿工准备好要挖一个新block时，该交易将放入到block中并开始挖矿。
5. 当挖到一个新block后，网络中的其它节点会收到一个消息：该block被挖到了，同时将该block添加到blockchain中。
6. 在block被添加到blockchain之后，交易完成，其TXO可以被新的交易所引用了。

如前所述，公钥和私钥都是随机字节序列，私钥代表获取的所有者，这一切有一个必要条件：随机算法产生的是真实随机值，一定不能发生这种情况：产生一个已经属于别人的私钥。

比特币使用椭圆曲线算法生成私钥。椭圆曲线是一个复杂的数学概念，在这不对其进行详述（如果非常感兴趣，可以参考[this gentle introduction to elliptic curves](#)，再次提醒：一大堆数学公式！）我们所需要知道的就是这些曲线可以生成非常大的随机数。比特币所使用的曲线算法将从0到2256随机选择一个数（这意味着有大概1077个随机数可供挑选，而宇宙可见范围内存在大约1078到1082个原子），这使得同一私钥被生成两次的可能为0。

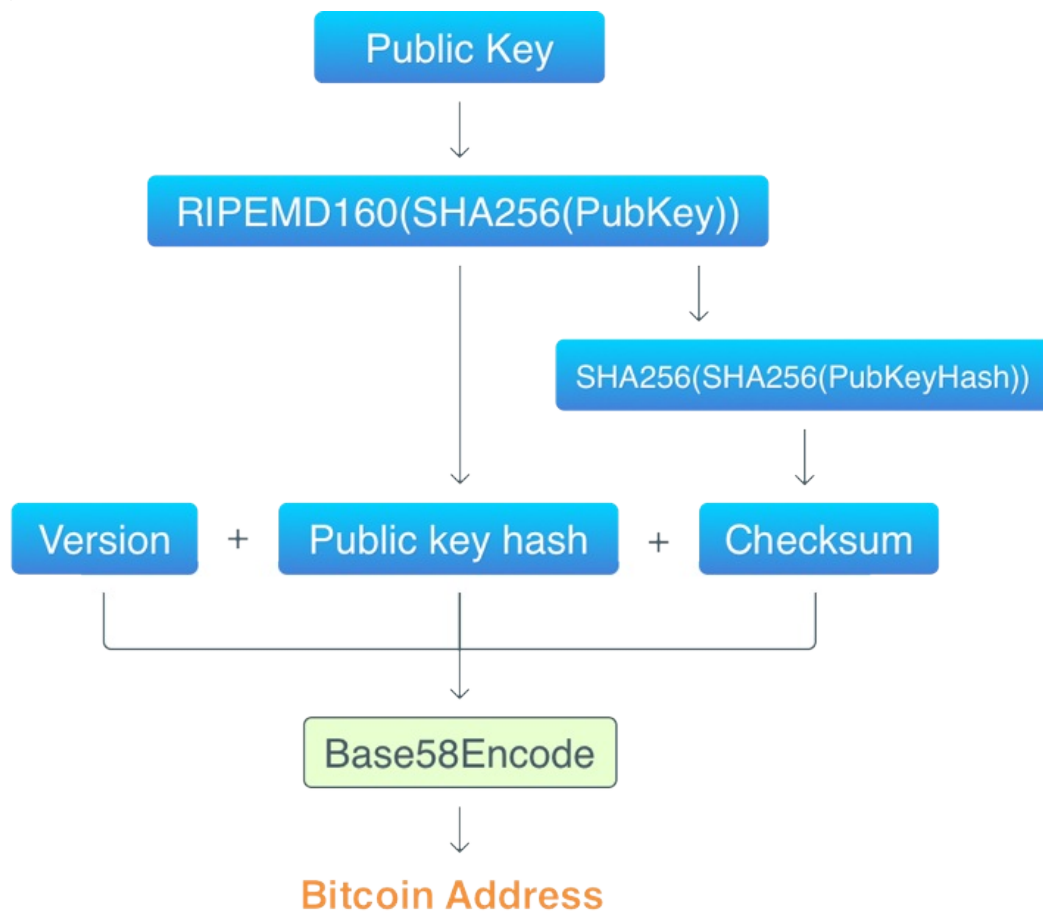
此外，比特币使用ECDSA算法（Elliptic Curve Digital Signature Algorithm）对交易进行签名。

现在，让我们回顾上面提到的比特币地址：1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa，该地址是公钥经过转换后的可读格式的字符串。如果解码后，公钥原始是这样的（以16进制表示）：

```
0062E907B15CBF27D5425399EBF6F0FB50EBB88F18C29B7D93
```

比特币使用Base58算法将公钥编码成可读格式的字符串。该算法与著名的Base64算法类似，但为了避免利用相似性进行攻击，Base58算法将一些字符移除了，因此没有0（数值0）、O（o的大写形式）、l（i的大写形式）、I（L的小写形式）。此外也没有+和/。

下面形象的描述从公钥获取地址的过程：



比特币地址由三部分组成：

Version	Public key hash	Checksum
00	62E907B15CBF27D5425399EBF6F0FB50EBB88F18	C29B7D93

由于哈希函数是单向的，因此不可能从公钥的hash值中获取公钥。但是，我们可以利用哈希函数计算出公钥的hash值，然后和已经保存的公钥hash值进行对比，从而验证是否匹配。

OK，所有背景已介绍完毕，让我们开始动手写代码吧，这会让上述概念变得更加清晰。

Wallet结构如下：

```
type Wallet struct {
    PrivateKey ecdsa.PrivateKey
    PublicKey  []byte
}

type Wallets struct {
    Wallets map[string]*Wallet
}

func NewWallet() *Wallet {
    private, public := newKeyPair()
    wallet := Wallet{private, public}

    return &wallet
}

func newKeyPair() (ecdsa.PrivateKey, []byte) {
    curve := elliptic.P256()
    private, err := ecdsa.GenerateKey(curve, rand.Reader)
    pubKey := append(private.PublicKey.X.Bytes(), private.PublicKey.Y.Bytes()...)

    return *private, pubKey
}
```

钱包就是一个密钥对。**Wallets**代表钱包集合，**Wallets**可以被存储到文件中，同时也可以从文件加载到内存中。**NewWallet**函数用于生成一个新的钱包。**newKeyPair**函数使用椭圆曲线算法生成私钥，紧接着通过私钥生成公钥。需要注意一点：椭圆曲线算法中，公钥是曲线上的点集合。因此，公钥由X,Y坐标混合而成。比特币中，这些坐标被组合在一起，生成公钥。

现在，创建一个地址：

```
func (w Wallet) GetAddress() []byte {
    pubKeyHash := HashPubKey(w.PublicKey)

    versionedPayload := append([]byte{version}, pubKeyHash...)
    checksum := checksum(versionedPayload)

    fullPayload := append(versionedPayload, checksum...)
    address := Base58Encode(fullPayload)

    return address
}

func HashPubKey(pubKey []byte) []byte {
    publicSHA256 := sha256.Sum256(pubKey)

    RIPEMD160Hasher := ripemd160.New()
    _, err := RIPEMD160Hasher.Write(publicSHA256[:])
    publicRIPEMD160 := RIPEMD160Hasher.Sum(nil)

    return publicRIPEMD160
}

func checksum(payload []byte) []byte {
    firstSHA := sha256.Sum256(payload)
    secondSHA := sha256.Sum256(firstSHA[:])

    return secondSHA[:addressChecksumLen]
}
```

通过Base58将公钥转换成可读格式，包括以下步骤：

1. 使用 RIPEMD160(SHA256(PubKey))对公钥进行两次哈希，生成pubKeyHash
2. 将版本信息追加到pubKeyHash之前，生成versionedPayload，此时
versionedPayload=version+pubKeyHash
3. 使用SHA256(SHA256(versionedPayload))进行两次哈希得到一个hash值，取该值的前n个字节最终生成checksum
4. 将checksum追加到versionedPayload之后，生成编码前的地址，此时地址=
version+pubKeyHash+checksum
5. 最后使用Base58对version+pubKeyHash+checksum编码生成最终的地址。

最终，我们得到一个真实的比特币地址，甚至可以在blockchain.info上查询该地址的余额。当然，无论查询多少次，该地址的余额都会是0。好的公钥加密算法是非常关键，它使产生相同私钥的概率尽可能低，低到不可能发生。同时，比特币地址生成算法使用一组开放的算法集合，因此不需要连接到比特币网络就可以生成比特币地址。

现在，修改TXI和TXO结构，在其中加入地址：

```

type TXInput struct {
    Txid      []byte
    Vout      int
    Signature []byte
    PubKey    []byte
}

func (in *TXInput) UsesKey(pubKeyHash []byte) bool {
    lockingHash := HashPubKey(in.PubKey)

    return bytes.Compare(lockingHash, pubKeyHash) == 0
}

type TXOutput struct {
    Value      int
    PubKeyHash []byte
}

func (out *TXOutput) Lock(address []byte) {
    pubKeyHash := Base58Decode(address)
    pubKeyHash = pubKeyHash[1 : len(pubKeyHash)-4]
    out.PubKeyHash = pubKeyHash
}

func (out *TXOutput) IsLockedWithKey(pubKeyHash []byte) bool {
    return bytes.Compare(out.PubKeyHash, pubKeyHash) == 0
}

```

由于我们不打算实现一个脚本语言，因此不再使用**ScriptPubKey**和**ScriptSig**。**ScriptSig**被**Signature**和**PubKey**代替；**ScriptPubKey**被**PubKeyHash**代替。通过下面的方法中实现比特币中TXO的加解锁和TXI的签名逻辑：

UsesKey方法核查TXI是否可以使用特定的未哈希的公钥来解锁一个TXO。**IsLockedWithKey**核查TXO是否被特定的哈希后的公钥锁定。**UsesKey**和**IsLockedWithKey**是互补函数，**FindUnspentTransactions**函数使用这两个函数来构建交易间的关系。

Lock方法是对TXO进行加锁。当发给某人货币时，仅仅知道他们的地址，因此该方法唯一入参就是地址信息。从地址中从解码出哈希后的公钥，将其保存到**PubKeyHash**中。

现在，让我们看看是否能正常工作：

```
$ blockchain_go createwallet
Your new address: 13Uu7B1vDP4ViXqHFswtbraM3EfQ3UkWXt

$ blockchain_go createwallet
Your new address: 15pUhCbtrGh3JUx5iHnXjfpYHyTgawvG5h

$ blockchain_go createwallet
Your new address: 1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy

$ blockchain_go createblockchain -address 13Uu7B1vDP4ViXqHFswtbraM3EfQ3UkWXt
0000005420fbfdafa00c093f56e033903ba43599fa7cd9df40458e373eee724d

Done!

$ blockchain_go getbalance -address 13Uu7B1vDP4ViXqHFswtbraM3EfQ3UkWXt
Balance of '13Uu7B1vDP4ViXqHFswtbraM3EfQ3UkWXt': 10

$ blockchain_go send -from 15pUhCbtrGh3JUx5iHnXjfpYHyTgawvG5h -to 13Uu7B1vDP4ViXqHFswt
braM3EfQ3UkWXt -amount 5
2017/09/12 13:08:56 ERROR: Not enough funds

$ blockchain_go send -from 13Uu7B1vDP4ViXqHFswtbraM3EfQ3UkWXt -to 15pUhCbtrGh3JUx5iHnX
jfpYHyTgawvG5h -amount 6
00000019afa909094193f64ca06e9039849709f5948fbac56cae7b1b8f0ff162

Success!

$ blockchain_go getbalance -address 13Uu7B1vDP4ViXqHFswtbraM3EfQ3UkWXt
Balance of '13Uu7B1vDP4ViXqHFswtbraM3EfQ3UkWXt': 4

$ blockchain_go getbalance -address 15pUhCbtrGh3JUx5iHnXjfpYHyTgawvG5h
Balance of '15pUhCbtrGh3JUx5iHnXjfpYHyTgawvG5h': 6

$ blockchain_go getbalance -address 1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy
Balance of '1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy': 0
```

对交易进行签名是保证某人只能花费属于自己货币的唯一方法。签名无效代表交易无效，该交易就无法被加入到blockchain中。

对于实现交易签名，万事俱备只欠东风：基于什么数据进行签名。是基于部分交易进行签名还是对整个交易进行签名？选择数据非常重要。签名的数据中必须包含可以唯一标识该数据的信息。例如，仅仅对TXO的交易额进行签名，这就不合理，因为签名中没有考虑发送者和接受者信息。

交易会解锁之前的TXO，生成新的TXO，因此必须将对以下数据进行签名：

1. 被解锁的TXO的公钥hash值。该值标识“发送者”
2. 新生成的，被加锁的TXO的公钥hash值。该值标识“接收者”
3. 新TXO的交易额

比特币的加锁、解锁逻辑存放在脚本中，分别存储在TXI的ScriptSig和TXO的ScriptPubKey字段。由于比特币支持多种类型的脚本，因此ScriptPubKey的整体也需要被签名

我们并不需要对TXI中的公钥进行签名。因此并非对“原始”交易进行签名，而是对TrimmedCopy交易进行签名。什么是TrimmedCopy交易？还是让我们来看代码理解下，从Sign方法开始：

TrimmedCopy交易的处理过程参见[此文](#)，可能有些过程，但是还是可以看出些端倪的。

```
func (tx *Transaction) Sign(privKey ecdsa.PrivateKey, prevTXs map[string]Transaction)
{
    if tx.IsCoinbase() {
        return
    }

    txCopy := tx.TrimmedCopy()

    for inID, vin := range txCopy.Vin {
        prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
        txCopy.Vin[inID].Signature = nil
        txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
        txCopy.ID = txCopy.Hash()
        txCopy.Vin[inID].PubKey = nil

        r, s, err := ecdsa.Sign(rand.Reader, &privKey, txCopy.ID)
        signature := append(r.Bytes(), s.Bytes()...)

        tx.Vin[inID].Signature = signature
    }
}
```

该方法接受两个参数：私钥以及所引用的之前交易的集合。对交易进行签名时，需要获取该交易所有TXI引用的TXO列表，因此需要存储这些TXO对应的交易。

我们一步一步来看：

```
if tx.IsCoinbase() {  
    return  
}
```

Coinbase交易没有真实的TXI，因此此交易不进行签名。

```
txCopy := tx.TrimmedCopy()
```

基于TrimmedCopy交易进行签名：

```
func (tx *Transaction) TrimmedCopy() Transaction {  
    var inputs []TXInput  
    var outputs []TXOutput  
  
    for _, vin := range tx.Vin {  
        inputs = append(inputs, TXInput{vin.Txid, vin.Vout, nil, nil})  
    }  
  
    for _, vout := range tx.Vout {  
        outputs = append(outputs, TXOutput{vout.Value, vout.PubKeyHash})  
    }  
  
    txCopy := Transaction{tx.ID, inputs, outputs}  
  
    return txCopy  
}
```

TrimmedCopy交易包括复制原始交易中所有的TXI和TXO，区别在于TrimmedCopy中TXI的**Signature**和**PubKey**被设置为nil。

接下来对TrimmedCopy交易中所有的TXI进行遍历：

```
for inID, vin := range txCopy.Vin {  
    prevTx := prevTXs[hex.EncodeToString(vin.Txid)]  
    txCopy.Vin[inID].Signature = nil  
    txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash  
}
```

对于每个TXI，为了确保万无一失，再次将**Signature**字段设置为nil，同时将**PubKey**设置为其所引用的TXO的**PubKeyHash**。此时，除了当前被处理的交易外，其他所有交易都是“空”交易（**Signature**和**PubKey**字段为nil）。由于比特币允许交易包含来自于不同地址的TXI，因此需要单独地对每个TXI进行签名，虽然对于我们的应用来说没必要这么做（在我们的应用中，一个交易包中的TXI均来自同一地址）。

```
txCopy.ID = txCopy.Hash()
txCopy.Vin[inID].PubKey = nil
```

Hash方法将交易序列化并通过SHA-256进行哈希。生成的哈希结果就是待签名的数据。此后，我们将**PubKey**字段重新设置为nil避免影响后续的迭代。下面是核心部分：

```
r, s, err := ecdsa.Sign(rand.Reader, &privKey, txCopy.ID)
signature := append(r.Bytes(), s.Bytes()...)

tx.Vin[inID].Signature = signature
```

我们使用ECDSA签名算法通过私钥 **privKey** 对**txCopy.ID**进行签名，生成一对数字序列。这些数字序列组合后放在**Signature**字段。

下面是校验过程：

```
func (tx *Transaction) Verify(prevTXs map[string]Transaction) bool {
    txCopy := tx.TrimmedCopy()
    curve := elliptic.P256()

    for inID, vin := range tx.Vin {
        prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
        txCopy.Vin[inID].Signature = nil
        txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
        txCopy.ID = txCopy.Hash()
        txCopy.Vin[inID].PubKey = nil

        r := big.Int{}
        s := big.Int{}
        sigLen := len(vin.Signature)
        r.SetBytes(vin.Signature[:sigLen / 2])
        s.SetBytes(vin.Signature[sigLen / 2:])

        x := big.Int{}
        y := big.Int{}
        keyLen := len(vin.PubKey)
        x.SetBytes(vin.PubKey[:keyLen / 2])
        y.SetBytes(vin.PubKey[keyLen / 2:])

        rawPubKey := ecdsa.PublicKey{curve, &x, &y}
        if ecdsa.Verify(&rawPubKey, txCopy.ID, &r, &s) == false {
            return false
        }
    }

    return true
}
```


首先，生成TrimmedCopy交易：

```
txCopy := tx.TrimmedCopy()
```

接下来，创建椭圆曲线用于生成键值对：

```
curve := elliptic.P256()
```

对于每个TXI的签名进行验证：

```
for inID, vin := range tx.Vin {
    prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
    txCopy.Vin[inID].Signature = nil
    txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
    txCopy.ID = txCopy.Hash()
    txCopy.Vin[inID].PubKey = nil
}
```

这个过程和**Sign**方法是一致的，因为验证的数据需要和签名的数据是一致的。

```
r := big.Int{}
s := big.Int{}
sigLen := len(vin.Signature)
r.SetBytes(vin.Signature[: (sigLen / 2)])
s.SetBytes(vin.Signature[(sigLen / 2):])

x := big.Int{}
y := big.Int{}
keyLen := len(vin.PubKey)
x.SetBytes(vin.PubKey[: (keyLen / 2)])
y.SetBytes(vin.PubKey[(keyLen / 2):])
```

这里，。之前，将签名生成的两个字节序列组合生成TXI的**Signature**；将椭圆曲线的X,Y坐标点集合（其实也是两个字节序列）组合生成TXI的**PubKey**。现在，我们需要将TXI的**Signature**和**PubKey**中的数据进行“拆包”，用于**crypto/ecdsa**库进行验证使用。

```
rawPubKey := ecdsa.PublicKey{curve, &x, &y}
if ecdsa.Verify(&rawPubKey, txCopy.ID, &r, &s) == false {
    return false
}
return true
```

现在，需要一个根据交易ID获取交易的函数，由于需要访问blockchain，因此作为blockchain的一个方法实现：

```

func (bc *Blockchain) FindTransaction(ID []byte) (Transaction, error) {
    bci := bc.Iterator()

    for {
        block := bci.Next()

        for _, tx := range block.Transactions {
            if bytes.Compare(tx.ID, ID) == 0 {
                return *tx, nil
            }
        }

        if len(block.PrevBlockHash) == 0 {
            break
        }
    }

    return Transaction{}, errors.New("Transaction is not found")
}

func (bc *Blockchain) SignTransaction(tx *Transaction, privKey ecdsa.PrivateKey) {
    prevTXs := make(map[string]Transaction)

    for _, vin := range tx.Vin {
        prevTX, err := bc.FindTransaction(vin.Txid)
        prevTXs[hex.EncodeToString(prevTX.ID)] = prevTX
    }

    tx.Sign(privKey, prevTXs)
}

func (bc *Blockchain) VerifyTransaction(tx *Transaction) bool {
    prevTXs := make(map[string]Transaction)

    for _, vin := range tx.Vin {
        prevTX, err := bc.FindTransaction(vin.Txid)
        prevTXs[hex.EncodeToString(prevTX.ID)] = prevTX
    }

    return tx.Verify(prevTXs)
}

```

FindTransaction方法根据ID查找并返回交易；**SignTransaction**对于一个交易找到其所有引用的交易后，进行签名；**VerifyTransaction**对于一个交易到其所有引用的交易后，进行验证。

交易签名在**NewUTXOTransaction**中进行：

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain) *Transaction {
    ...

    tx := Transaction{nil, inputs, outputs}
    tx.ID = tx.Hash()
    bc.SignTransaction(&tx, wallet.PrivateKey)

    return &tx
}
```

在交易加入到block前进行交易验证：

```
func (bc *Blockchain) MineBlock(transactions []*Transaction) {
    var lastHash []byte

    for _, tx := range transactions {
        if bc.VerifyTransaction(tx) != true {
            log.Panic("ERROR: Invalid transaction")
        }
    }
    ...
}
```

现在执行看看效果：

```
$ blockchain_go createwallet
Your new address: 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR

$ blockchain_go createwallet
Your new address: 1NE86r4Esjf53EL7fR86CsftZpNN42Sfab

$ blockchain_go createblockchain -address 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR
000000122348da06c19e5c513710340f4c307d884385da948a205655c6a9d008

Done!

$ blockchain_go send -from 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR -to 1NE86r4Esjf53EL7fR86
CsftZpNN42Sfab -amount 6
0000000f3dbb0ab6d56c4e4b9f7479afe8d5a5dad4d2a8823345a1a16cf3347b

Success!

$ blockchain_go getbalance -address 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR
Balance of '1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR': 4

$ blockchain_go getbalance -address 1NE86r4Esjf53EL7fR86CsftZpNN42Sfab
Balance of '1NE86r4Esjf53EL7fR86CsftZpNN42Sfab': 6
```

一切OK !

把 **NewUTXOTransaction** 中的以下语句注释，这样未被签名的交易不会被叫入到blockchain中。

bc.SignTransaction(&tx, wallet.PrivateKey)

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain) *Transaction {
    ...
    tx := Transaction{nil, inputs, outputs}
    tx.ID = tx.Hash()
    // bc.SignTransaction(&tx, wallet.PrivateKey)

    return &tx
}
$ go install
$ blockchain_go send -from 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR -to 1NE86r4Esjf53EL7fR86
CsftZpNN42Sfab -amount 1
2017/09/12 16:28:15 ERROR: Invalid transaction
```

目前为止，除了网络特性外，比特币的许多关键特性均已实现。下一章，对交易实现再做一些完善。

本文一开始提到**blockchain**是一个分布式数据库。目前为止，我们一直聚焦于“数据库”，暂时忽略了“分布式”。现在，我们已经实现了**blockchain**数据库的所有部分。本章中，我们将讨论前面章节中未涉及的一些机制；下一章，我们将实现**blockchain**的分布式特性。

前文中，我们没有讨论挖矿奖励。现在让我们来实现此机制。

所谓的奖励其实是一个coinbase交易。当矿工挖到一个新的block时，除了一个通用交易外，还会包含一个coinbase交易。Coinbase的TXO中包括矿工的公钥。

修改**send**方法，实现挖矿奖励：

```
func (cli *CLI) send(from, to string, amount int) {
    ...
    bc := NewBlockchain()
    UTXOSet := UTXOSet{bc}
    defer bc.db.Close()

    tx := NewUTXOTransaction(from, to, amount, &UTXOSet)
    cbTx := NewCoinbaseTX(from, "")
    txs := []*Transaction{cbTx, tx}

    newBlock := bc.MineBlock(txs)
    fmt.Println("Success!")
}
```

我们的实现中，交易创建者进行挖矿，挖到一个新block时，就会收到一个奖励。

在[持久化和CLI](#)中，曾经提到比特币将block存储在数据库：block存储在**blocks**数据库；交易TXO存储在**chainstate**数据库。现在重新回顾下**chainstate**的结构：

1. 'c' + 32-byte transaction hash -> unspent transaction output record for that transaction
2. 'B' -> 32-byte block hash: the block hash up to which the database represents the unspent transaction outputs

在此之前，我们实现了交易，但是没有将交易信息存储到**chainstate**数据库。现在，我们来实现此机制。

chainstate数据库不存储交易本身，而是存储所有UTXO，称为UTXO集合。此外，**chainstate**数据库也会存储“the block hash up to which the database represents the unspent transaction outputs”，不过目前用不到block高度的信息，故暂不实现。

那么，UTXO集合有什么用呢？

让我们回顾下**FindUnspentTransactions**方法：

```
func (bc *Blockchain) FindUnspentTransactions(pubKeyHash []byte) []Transaction {
    ...
    bci := bc.Iterator()

    for {
        block := bci.Next()

        for _, tx := range block.Transactions {
            ...
        }

        if len(block.PrevBlockHash) == 0 {
            break
        }
    }
    ...
}
```

该方法返回所有UTXO。由于交易存储在block中，因此需要遍历blockchain中的所有block，对block中的所有交易进行检查。截止2017/9/18，比特币有485860个block，整个数据库大小超过140GB。这意味着需要遍历整个数据库中的所有block对交易进行检查。可想而知效率会非常低！

为了解决此问题，我们需要对UTXO建立索引，这就是所谓的UTXO集合。UTXO集合其实是一个缓存，遍历blockchain中所有交易得到的，用于计算余额以及交易验证。截止2017/9，比特币的UTXO集合大小约为2.7GB。

目前，下述方法用于查找交易：

1. Blockchain.FindUnspentTransactions – 遍历所有block返回UTXO
2. Blockchain.FindSpendableOutputs – 新交易创建时，通过此函数找到满足要求的可以供消费的交易。此函数使用到了Blockchain.FindUnspentTransactions
3. Blockchain.FindUTXO – 返回指定公钥hash值的所有UTXO，用于计算余额。此函数使用到了Blockchain.FindUnspentTransactions
4. Blockchain.FindTransaction - 根据交易ID查找交易。该函数会遍历所有block进行查找。

上述方法均会遍历整个数据库中所有block，但由于UTXO集合仅仅存储UTXO不存储所有交易，因此UTXO集合无法为上述所有方法带来改善，如Blockchain.FindTransaction方法。

所以，我们考虑使用下面的新方法：

1. Blockchain.FindUTXO – 遍历block返回所有UTXO
2. UTXOSet.Reindex – 将Blockchain.FindUTXO结果存储到数据库
3. UTXOSet.FindSpendableOutputs - 和Blockchain.FindSpendableOutputs类似，只不过基于UTXO集合进行计算
4. UTXOSet.FindUTXO - 和Blockchain.FindUTXO 类似，只不过基于UTXO集合进行计算
5. Blockchain.FindTransaction保持不变

现在，两个使用最频繁的方法均使用了UTXO集合做缓存：

```
type UTXOSet struct {  
    Blockchain *Blockchain  
}
```

我们仍使用同一个数据库，将UTXO集合存储在不同的bucket中。因此，UTXOSet和Blockchain相伴而生。

```
func (u UTXOSet) Reindex() {
    db := u.Blockchain.db
    bucketName := []byte(utxoBucket)

    err := db.Update(func(tx *bolt.Tx) error {
        err := tx.DeleteBucket(bucketName)
        _, err = tx.CreateBucket(bucketName)
    })

    UTXO := u.Blockchain.FindUTXO()

    err = db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket(bucketName)

        for txID, outs := range UTXO {
            key, err := hex.DecodeString(txID)
            err = b.Put(key, outs.Serialize())
        }
    })
}
```

首先，若UTXO集合存在，将其删除；然后获取所有UTXO；最后将其保存到bucket中。

Blockchain.FindUTXO几乎与**Blockchain.FindUnspentTransactions**完全一致，只不过现在该方法返回**TransactionID** → **TransactionOutputs**对。

现在，发送货币过程可以用到UTXO集合：

```
func (u UTXOSet) FindSpendableOutputs(pubkeyHash []byte, amount int) (int, map[string]
[]int) {
    unspentOutputs := make(map[string][]int)
    accumulated := 0
    db := u.Blockchain.db

    err := db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(utxoBucket))
        c := b.Cursor()

        for k, v := c.First(); k != nil; k, v = c.Next() {
            txID := hex.EncodeToString(k)
            outs := DeserializeOutputs(v)

            for outIdx, out := range outs.Outputs {
                if out.IsLockedWithKey(pubkeyHash) && accumulated < amount {
                    accumulated += out.Value
                    unspentOutputs[txID] = append(unspentOutputs[txID], outIdx)
                }
            }
        }
    })

    return accumulated, unspentOutputs
}
```

获取余额过程也可以用到UTXO集合：

```
func (u UTXOSet) FindUTXO(pubKeyHash []byte) []TXOutput {
    var UTXOs []TXOutput
    db := u.Blockchain.db

    err := db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(utxoBucket))
        c := b.Cursor()

        for k, v := c.First(); k != nil; k, v = c.Next() {
            outs := DeserializeOutputs(v)

            for _, out := range outs.Outputs {
                if out.IsLockedWithKey(pubKeyHash) {
                    UTXOs = append(UTXOs, out)
                }
            }
        }

        return nil
    })

    return UTXOs
}
```

之前**Blockchain**的那些方法已经不需要了，现在开始使用这些新方法。

UTXO集合的出现意味着交易信息分为两部分存储：实际交易数据存储在**blockchain**中，UTXO数据存储在UTXO集合中。为了保持数据一致性的同时避免每次挖到一个新的**block**都重建UTXO集合（不希望频繁扫描**blockchain**），需要一个数据同步机制：

```
func (u UTXOSet) Update(block *Block) {
    db := u.Blockchain.db

    err := db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(utxoBucket))

        for _, tx := range block.Transactions {
            if tx.IsCoinbase() == false {
                for _, vin := range tx.Vin {
                    updatedOuts := TXOutputs{}
                    outsBytes := b.Get(vin.Txid)
                    outs := DeserializeOutputs(outsBytes)

                    for outIdx, out := range outs.Outputs {
                        if outIdx != vin.Vout {
                            updatedOuts.Outputs = append(updatedOuts.Outputs, out)
                        }
                    }

                    if len(updatedOuts.Outputs) == 0 {
                        err := b.Delete(vin.Txid)
                    } else {
                        err := b.Put(vin.Txid, updatedOuts.Serialize())
                    }
                }
            }

            newOutputs := TXOutputs{}
            for _, out := range tx.Vout {
                newOutputs.Outputs = append(newOutputs.Outputs, out)
            }

            err := b.Put(tx.ID, newOutputs.Serialize())
        }
    })
}
```

该方法看上去很长，但实际很简单。当新的block生成时，需要更新UTXO集合。更新意味着删除已消费的TXO，同时将新生成的交易中的UTXO添加进来。若某个交易的UTXO全部被删除，该交易也会随之被移除。

```
func (cli *CLI) createBlockchain(address string) {  
    ...  
    bc := CreateBlockchain(address)  
    defer bc.db.Close()  
  
    UTXOSet := UTXOSet{bc}  
    UTXOSet.Reindex()  
    ...  
}
```

当blockchain被创建时，需要创建UTXO集合。现在，这是唯一需要进行**Reindex**的地方。虽然blockchain一开始仅有一个block以及一个交易，这时仅需要执行**Update**没必要进行**Reindex**，但是将来可能需要这个重建机制。

```
func (cli *CLI) send(from, to string, amount int) {  
    ...  
    newBlock := bc.MineBlock(txs)  
    UTXOSet.Update(newBlock)  
}
```

新的block生成时，更新UTXO集合。

下面看看运行情况：

```
$ blockchain_go createblockchain -address 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1
00000086a725e18ed7e9e06f1051651a4fc46a315a9d298e59e57aeacbe0bf73

Done!

$ blockchain_go send -from 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1 -to 12DkLzLQ4B3gnQt62EPR
JGZ38n3zF4Hzt5 -amount 6
0000001f75cb3a5033aeecbf6a8d378e15b25d026fb0a665c7721a5bb0faa21b

Success!

$ blockchain_go send -from 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1 -to 12ncZhA5mFTTnTmHq1aT
PYBri4jAK8TacL -amount 4
000000cc51e665d53c78af5e65774a72fc7b864140a8224bf4e7709d8e0fa433

Success!

$ blockchain_go getbalance -address 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1
Balance of '1F4MbuqjcuJGymjcuYQMUVYB37AWKkSLif': 20

$ blockchain_go getbalance -address 12DkLzLQ4B3gnQt62EPRJGZ38n3zF4Hzt5
Balance of '1XWu6nitBWe6J6v6MXmd5rhdP7dZsExbx': 6

$ blockchain_go getbalance -address 12ncZhA5mFTTnTmHq1aTPYBri4jAK8TacL
Balance of '13UASQpCR8Nr41PoJH8Bz4K6cmTCqweskL': 4
```

1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1收到3次奖励：

1. 第1次奖励，挖到了genesis block
2. 第2次奖励，挖到了
block:0000001f75cb3a5033aeecbf6a8d378e15b25d026fb0a665c7721a5bb0faa21b
3. 第3次奖励，挖到了
block:000000cc51e665d53c78af5e65774a72fc7b864140a8224bf4e7709d8e0fa433

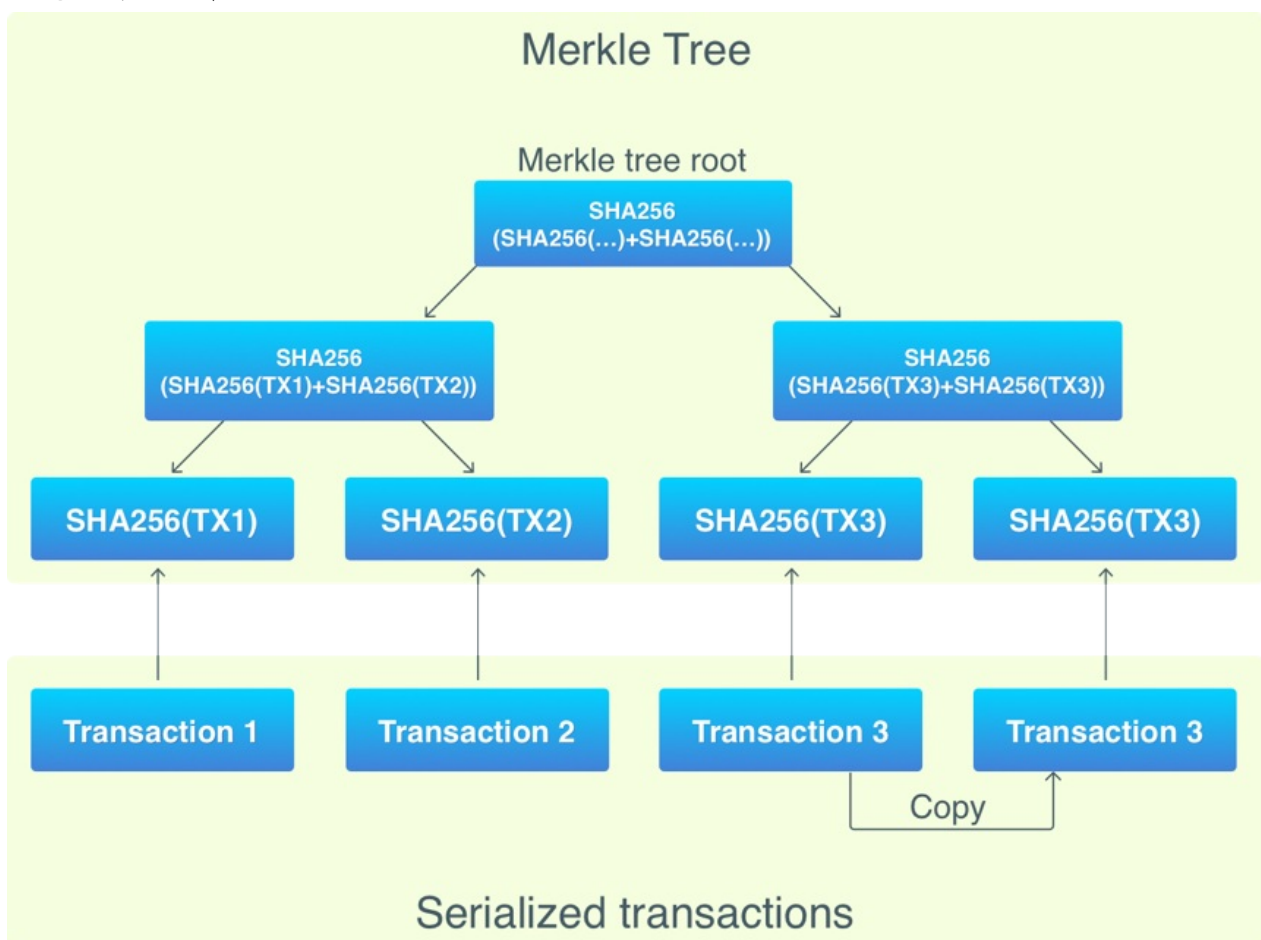
下面我们讨论另外一个优化技术。

如前所述，比特币的数据库大小已经超过140GB，由于天生的去中心化特性，比特币网络中的每个节点都是独立、自立的，这意味着每个节点必须存储整个blockchain。随着比特币使用者的日益增多，人人拥有整个blockchain变得越来越困难。同时，作为完全的参与者，富有对交易和block进行验证的责任，这需要节点之间通过网络进行交互从而下载新的block。

在中本聪发布的[比特币白皮书](#)中，针对此问题提供了一个解决方案：SPV（Simplified Payment Verification）。SPV是一个轻量级的比特币节点，该节点不会下载整个blockchain，也不会对block和交易进行验证，而是通过查找block中的交易来验证支付，同时连接到一个完整节点获取所需数据。SPV机制允许多个轻量级节点对应一个完整节点。为了实现SPV，需要一种交易检测方法，若包含该交易就无需下载整个block，此时默克尔树派上用场了。

比特币使用默克尔树获取交易hash值，交易hash值保存在block的头信息中，应用于PoW。目前，仅仅将一个block中所有交易的hash值简单地组合在一起，然后再使用SHA256算法生成hash值，虽然这也可以为block的交易集合创建唯一标识，但默克尔树有一些额外的优势。

默克尔树如下所示：



每个block有一个默克尔树，树中每个叶子节点是一个交易的hash值（比特币使用双重SHA256哈希）。叶子节点的数量一定是偶数，然后并非每个block都恰好有偶数个交易。当block有奇数个交易时，最后一个交易会被复制一次（复制仅仅发生在默克尔树中而不是block中！）。

默克尔树自下而上的进行组织，叶子节点成对分组后将两个hash值组合后生成新的hash值，形成上层的树节点，重复整个过程直到只有一个树节点为止，也就是所说的根节点。根节点的hash值是整个交易集的唯一标识，保存在block头信息中，用于PoW过程。

默克尔树的好处是：验证某个交易不需要现在整个block，而仅仅需要交易hash值、默克尔树根节点hash值以及默克尔路径即可。

下面让我们开始写代码：

```
type MerkleTree struct {
    RootNode *MerkleNode
}

type MerkleNode struct {
    Left  *MerkleNode
    Right *MerkleNode
    Data []byte
}

func NewMerkleNode(left, right *MerkleNode, data []byte) *MerkleNode {
    mNode := MerkleNode{}

    if left == nil && right == nil {
        hash := sha256.Sum256(data)
        mNode.Data = hash[:]
    } else {
        prevHashes := append(left.Data, right.Data...)
        hash := sha256.Sum256(prevHashes)
        mNode.Data = hash[:]
    }

    mNode.Left = left
    mNode.Right = right

    return &mNode
}
```

NewMerkleNode方法用于创建**MerkleNode**：当节点为叶子节点时，**Data**保存某个交易的hash值；当节点为非叶子节点时，**Data**保存两个子节点生成的hash值。

```

func NewMerkleTree(data [][]byte) *MerkleTree {
    var nodes []MerkleNode

    if len(data)%2 != 0 {
        data = append(data, data[len(data)-1])
    }

    for _, datum := range data {
        node := NewMerkleNode(nil, nil, datum)
        nodes = append(nodes, *node)
    }

    for i := 0; i < len(data)/2; i++ {
        var newLevel []MerkleNode

        for j := 0; j < len(nodes); j += 2 {
            node := NewMerkleNode(&nodes[j], &nodes[j+1], nil)
            newLevel = append(newLevel, *node)
        }

        nodes = newLevel
    }

    mTree := MerkleTree{&nodes[0]}

    return &mTree
}

```

创建一个默克尔树之前，需要确保有偶数个叶子节点，然后将数据转换为叶子节点，最后生成整个默克尔树。

译者注：上述实现存在一个问题：当叶子节点的数量是 $2n$ 时，可以正常创建树（*for j := 0; j < len(nodes); j += 2...*该段代码体现）；如果不是 $2n$ 时，创建树会发生异常！

修改**Block.HashTransactions**方法，用于获取在PoW过程获取交易hash值。

```

func (b *Block) HashTransactions() []byte {
    var transactions [][]byte

    for _, tx := range b.Transactions {
        transactions = append(transactions, tx.Serialize())
    }
    mTree := NewMerkleTree(transactions)

    return mTree.RootNode.Data
}

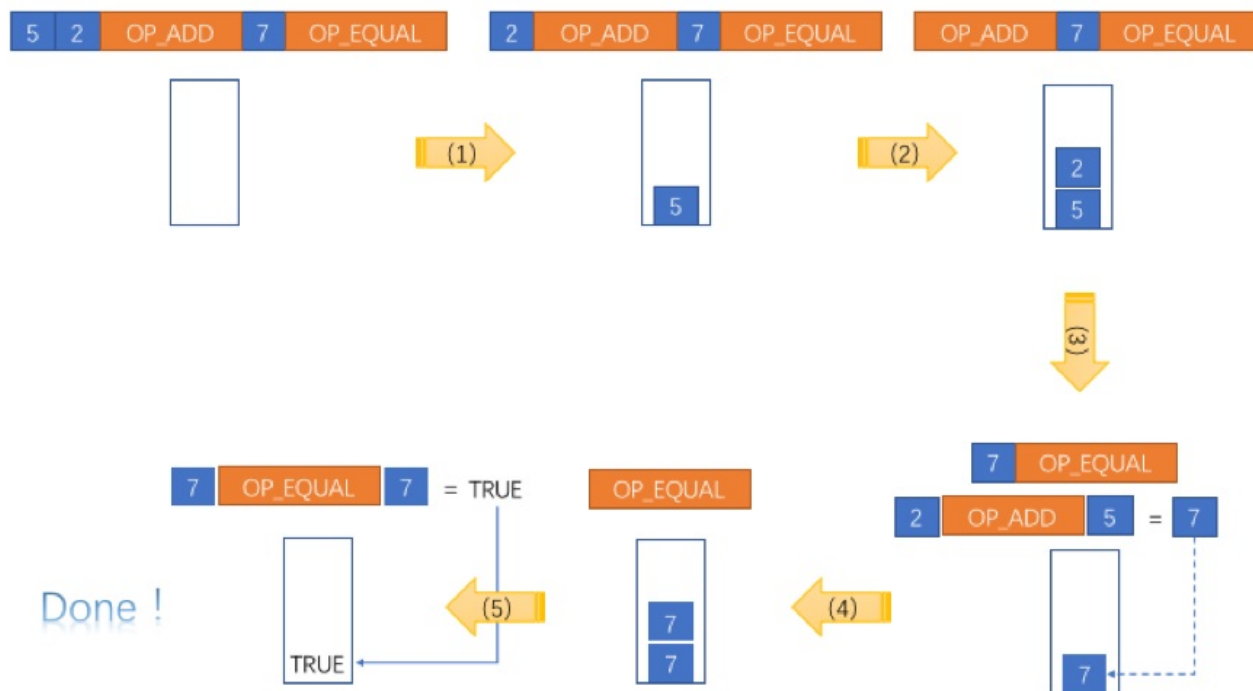
```


比特币拥有一个称作**script**的脚本语言。通过**script**，可以对TXO加锁，同时支持TXI解锁TXO。Script非常简单，就是一些代码和操作符的序列，例如：

```
5 2 OP_ADD 7 OP_EQUAL
```

5, 2, 7是数据，**OP_ADD**和**OP_EQUAL**是操作符。**script**从左至右执行代码：遇到数据放入堆栈（先进后出）；遇到操作符就从堆栈顶部取出所需的数据，并将结果放入堆栈。

上面的代码执行过程如下：



OP_ADD从堆栈中获取两个数据然后求和，将结果放入堆栈中。**OP_EQUAL**从堆栈获取两个数据进行比较：如果相等将**true**放入堆栈；否则将**false**放入堆栈。栈顶数据就是代码的执行结果。

比特币中用于执行交付的脚本：

```
<signature> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

该脚本是比特币中最常用的脚本，称为**P2PKH**（Pay to Public Key Hash）。此脚本用于向一个公钥**hash**值进行支付，例如使用一个公钥对货币进行加锁。比特币支付的核心就在于此：无账户，无基金转移；仅有一个脚本用于检查所提供的签名和公钥是否正确。

此脚本数据分为两部分：

1. 第一部分，`<signature> <pubKey>`，被存放在TXI的ScriptSig 中
2. 第二部分，`OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG`，被存放在TXO的ScriptPubKey 中

TXO定义解锁逻辑，TXI提供数据解锁TXO，脚本执行如下：

1. Stack: **empty** Script: **<signature> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG**
2. Stack: **<signature>** Script: **<pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG**
3. Stack: **<signature> <pubKey>** Script: **OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG**
4. Stack: **<signature> <pubKey> <pubKey>** Script: **OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG**
5. Stack: **<signature> <pubKey> <pubKeyHash>** Script: **<pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG**
6. Stack: **<signature> <pubKey> <pubKeyHash> <pubKeyHash>** Script: **OP_EQUALVERIFY OP_CHECKSIG**
7. Stack: **<signature> <pubKey>** Script: **OP_CHECKSIG**
8. Stack: **true or false.** Script: **empty.**

OP_DUP复制复制栈顶数据，**OP_HASH160**获取栈顶数据、使用**RIPEMD160**生成hash值并放入堆栈。**OP_EQUALVERIFY**比较两个栈顶数据，不相等则结束脚本。**OP_CHECKSIG**对交易进行哈希处理并使用**<signature>**、**<pubKey>**验证签名。

正因为有了script脚本语言，比特币可以成为智能合约平台，如该脚本可以支持多种支付模式，不仅仅是单个数字了。

我们已经实现了基于blockchain的数字货币的所有关键特性。我们实现了blockchain、地址、挖矿和交易。但是还有一个特性尚未实现，该特性使得比特币成为全球系统：共识机制。下章我们开始实现blockchain的“去中心化”特性。

坚持就是胜利！

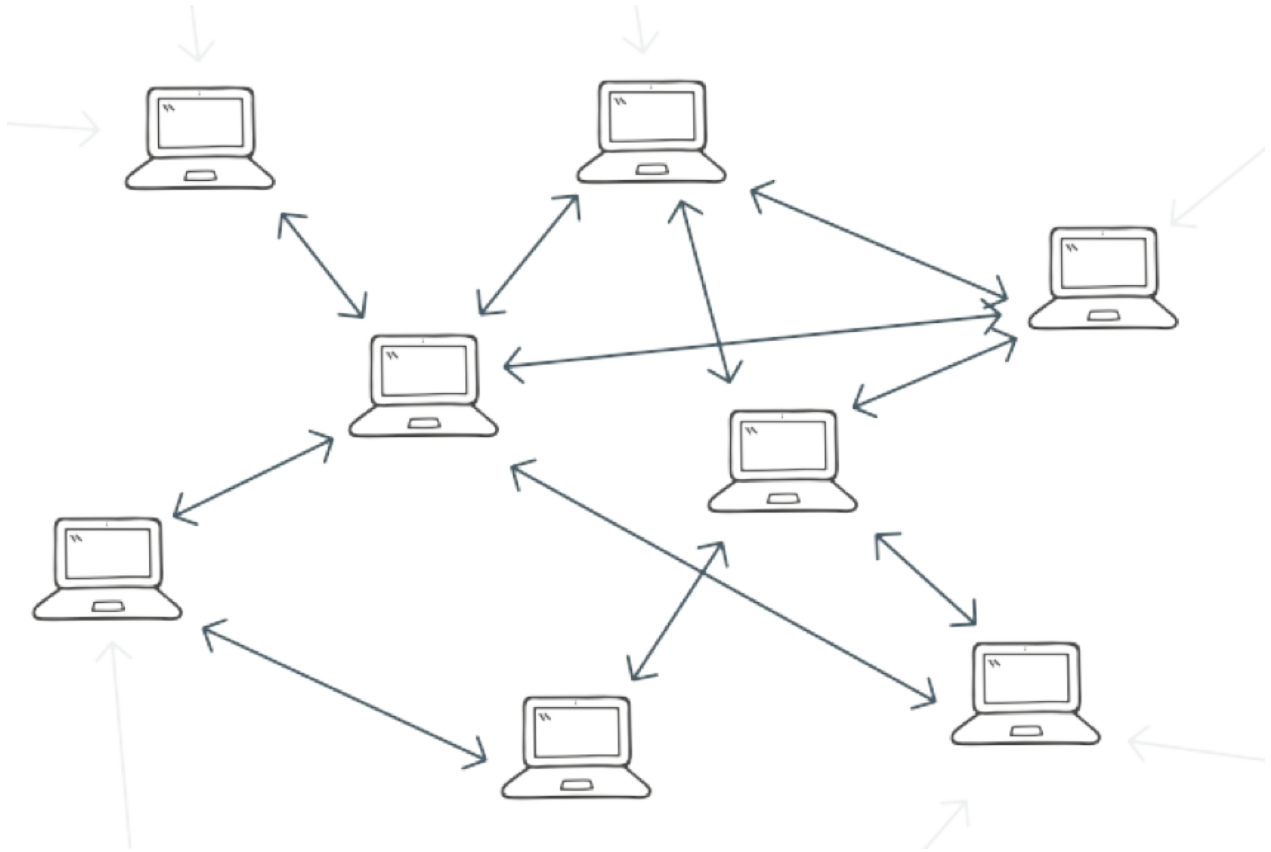
目前为止，取款连的所有特性均以实现：匿名的、安全的、随机生成的地址；区块链存储数据；PoW机制；可靠地存储交易。虽然这些特性都非常关键，但是使数字货币成为可能的是网络。

区块链网络如同人类社会一样，只有参与者对所有规则达成一致并严格遵循这些规则时，才能使整个网络繁荣发展；否则，制定再多再好的规则亦是徒劳。

原作者注：由于时间原因，本文不会实现一个真实的P2P网络，而仅仅基于一个最通用的场景进行讲解。该场景会涉及多种类型的节点，不过最终的实现也不一定能否保证可以正常工作，敬请谅解！不过，对于读者来说，完善该场景以及实现一个真实的P2P网络将会是一个十分有趣的课题☺。

Blockchain网络是一个去中心化网络，没有Server和Client的概念。Blockchain网络由许多节点组成，每个节点都是一个完备的、自治的成员。节点既是Client又是Server，与通常意义的Web应用相比，这是最主要的区别。

Blockchain网络是一个P2P网络：网络中各节点之间直接连接，不存在任何层级结构，拓扑结构示意图如下：



(Business vector created by Dooder - Freepik.com)

Blockchain网络节点需要同其他节点进行交互，获取其他节点的状态并维护状态信息，这使得实现变得更加复杂。

Blockchain节点需要扮演不同的角色：

(1) 矿工

这类节点运行在专有机，如ASIC机器，唯一的目标就是尽可能快的挖出新的block。挖矿实际上就是解决PoW谜题，因此矿工节点仅存在于使用PoW机制的blockchain中，对于使用PoS机制的blockchain，就不存在挖矿一说。

(2) 完整节点

这类节点拥有整个blockchain数据库，从而对block或交易进行有效性验证。此外，这类节点支持路由操作，借助此操作实现节点间相互发现。这对于blockchain网络十分关键，因为正是这些节点来共同决策block或交易是否有效。

(3) SPV(Simplified Payment Verification)

这类节点具备对交易进行验证的能力，但不存储完整的blockchain数据库。SPV节点并非对所有交易都能进行验证，例如，某个SPV节点仅能对发送给特定地址的交易进行验证，这取决于SPV节点连接到哪个完整节点，可以获取到哪些数据（多个SPV节点可能连接到同一个完整节点）。SPV是钱包应用的基础，使得不必下载完整blockchain的情况下仍然可以进行交易验证。

为了便于实现，需要对一些事情进行简化。由于模拟blockchain网络需要许多机器，然而我们无法找到那么多机器。一个解决方法是通过虚拟机或者Docker进行仿真，但是这将使我们不得不处理许多关于虚拟路或者Docker的问题，从而使实现变得更加困难。因此，我们打算用单台机器运行多个节点的方式来模拟blockchain网络。为了达到此目的，将不再使用IP作为节点标识，而是使用端口号作为节点标识，例如节点地址为127.0.0.1:3001、127.0.0.1:3002、127.0.0.1:3003等。此时，我们使用NODE_ID环境变量保存端口号，用来标识一个节点。这样可以打开不同的终端，设置不同的NODE_ID，这样就拥有了多个节点。

节点采用端口号作为标识，这使得每个节点的blockchain和钱包文件也不相同，文件名需要和端口号对应起来，例如，**blockchain_3000.db**, **blockchain_30001.db** and **wallet_3000.db**, **wallet_30001.db**等等。

当Bitcoin Core首次运行时，需要从某些节点下载最新状态的blockchain，从哪些节点下载呢？

对于节点地址绝不能采用硬编码方式，因为一旦节点被攻击或者关停，会导致新的节点无法加入网络。Bitcoin Core中会硬编码一些DNS Seeds地址。这些不是某种节点而是DNS服务器，这些DNS服务器知道节点地址。当启动一次完全干净的Bitcoin Core时，将连接到某个seed，然后获取所有节点的列表，最终从这些节点下载blockchain。

然而，我们将采用集中化的实现方式，将包括以下节点：

1. 中心节点：所有其他节点均连接到此节点；该节点向其他节点发送数据
2. 矿工节点：将交易存储在mempool中，当交易数量满足要求时，开始挖新的block
3. 钱包节点：用于在钱包之间发送钱币。与SPV节点不同，该节点存储完整的blockchain

场景

本文的目标是实现以下场景：

1. 中心节点创建blockchain
2. 其他钱包节点连接到中心节点，并下载blockchain
3. 一个或多个矿工节点连接到中心节点，并下载blockchain
4. 钱包节点创建一个交易
5. 矿工节点收到此交易并保存在mempool中
6. 当mempool有足够交易时，矿工开始挖矿
7. 当新的block生成时，其将被发送到中心节点
8. 钱包节点与中心节点进行同步
9. 钱包节点的拥有者将检查支付是否成功

整个过程与比特币很像，虽然没有实现一个真实的P2P网络，但是该场景仍然是比特币中最重要、最主要的用例。

version消息

节点间借助于消息进行通信。当新节点启动后，将从一个DNS seed获取节点列表，并向这些节点发送**version**消息。version结构如下：

```

type version struct {
    Version      int
    BestHeight   int
    AddrFrom     string
}

```

由于仅仅有一个blockchain版本，因此**Version**字段不存储任何重要信息。**BestHeight**存放该节点的blockchain的长度。**AddrFrom**存放消息发送者的地址。

当一个节点收到**version**消息后，需要做些什么呢？它将发送自身的**version**消息给对方。这是一种握手协议：不握手不能做其他任何事情。**version**用于发现一个更长的blockchain。当一个节点收到**version**消息后，将会检查发送节点的**version.BestHeight**值是否更大（即blockchain更长），如果发送节点的blockchain更长的话，接收节点将会发送请求获取缺失的block。

创建一个Server用于接收消息：

```

var nodeAddress string
var knownNodes = []string{"localhost:3000"}

func StartServer(nodeID, minerAddress string) {
    nodeAddress = fmt.Sprintf("localhost:%s", nodeID)
    miningAddress = minerAddress
    ln, err := net.Listen(protocol, nodeAddress)
    defer ln.Close()

    bc := NewBlockchain(nodeID)

    if nodeAddress != knownNodes[0] {
        sendVersion(knownNodes[0], bc)
    }

    for {
        conn, err := ln.Accept()
        go handleConnection(conn, bc)
    }
}

```

首先，将中心节点地址硬编码到程序中：一开始每个节点必须要知道该地址。**minerAddress**参数表示接收挖矿奖励的地址。

```

if nodeAddress != knownNodes[0] {
    sendVersion(knownNodes[0], bc)
}

```

若当前节点不是中心节点，需要向中心节点发送**version**消息，用于检查当前节点的**blockchain**是否过时。

```
func sendVersion(addr string, bc *Blockchain) {
    bestHeight := bc.GetBestHeight()
    payload := gobEncode(version{nodeVersion, bestHeight, nodeAddress})

    request := append(commandToBytes("version"), payload...)

    sendData(addr, request)
}
```

消息由字节数组构成：前12个字节表示命令名（此时是“version”），紧接着是gob编码的消息体。

```
func commandToBytes(command string) []byte {
    var bytes [commandLength]byte

    for i, c := range command {
        bytes[i] = byte(c)
    }

    return bytes[:]
}
```

commandToBytes函数创建长度为12的字节数组，然后将命令名填充到其中。

```
func bytesToCommand(bytes []byte) string {
    var command []byte

    for _, b := range bytes {
        if b != 0x0 {
            command = append(command, b)
        }
    }

    return fmt.Sprintf("%s", command)
}
```

bytesToCommand与**commandToBytes**作用正好相反：当一个节点接收到一个命令时，通过**bytesToCommand**函数从中解析出命令名和消息体。**handleConnection**函数会根据解析出的命令名来调用不同的消息处理函数。

```
func handleConnection(conn net.Conn, bc *Blockchain) {
    request, err := ioutil.ReadAll(conn)
    command := bytesToCommand(request[:commandLength])
    fmt.Printf("Received %s command\n", command)

    switch command {
    ...
    case "version":
        handleVersion(request, bc)
    default:
        fmt.Println("Unknown command!")
    }

    conn.Close()
}
```

version的消息处理函数如下：

```
func handleVersion(request []byte, bc *Blockchain) {
    var buff bytes.Buffer
    var payload version

    buff.Write(request[commandLength:])
    dec := gob.NewDecoder(&buff)
    err := dec.Decode(&payload)

    myBestHeight := bc.GetBestHeight()
    foreignerBestHeight := payload.BestHeight

    if myBestHeight < foreignerBestHeight {
        sendGetBlocks(payload.AddrFrom)
    } else if myBestHeight > foreignerBestHeight {
        sendVersion(payload.AddrFrom, bc)
    }

    if !nodeIsKnown(payload.AddrFrom) {
        knownNodes = append(knownNodes, payload.AddrFrom)
    }
}
```

首先，对请求中的数据进行解码获取消息体，这对所有消息处理函数都是一样的，后续介绍其他消息处理函数时将不再进行介绍。

随后，接收节点将自身的**BestHeight**与消息体（即发送节点）中的**BestHeight**进行比较。若接收节点的blockchain更长，将发送**version**消息；若发送节点的blockchain更长，将发送**getblocks**消息。

getblocks消息

```
type getblocks struct {  
    AddrFrom string  
}
```

getblocks消息（比特币实现中更加复杂）：返回当前节点所拥有的block的hash值列表，而不是所有block的详细信息。出于降低网络负荷的目的，若需要下载block，可以从多个节点同时下载，没必要从单个节点下载。相应的消息处理函数也很简单：

```
func handleGetBlocks(request []byte, bc *Blockchain) {  
    ...  
    blocks := bc.GetBlockHashes()  
    sendInv(payload.AddrFrom, "block", blocks)  
}
```

我们的实现中，该消息处理函数通过发送inv消息返回所有block的hash值。

inv消息

```
type inv struct {  
    AddrFrom string  
    Type     string  
    Items    [][]byte  
}
```

inv消息包含发送节点所拥有的block或交易的hash值列表。**Type**用于表示是block还是交易。该消息处理函数如下：


```

func handleInv(request []byte, bc *Blockchain) {
    ...
    fmt.Printf("Receieved inventory with %d %s\n", len(payload.Items), payload.Type)

    if payload.Type == "block" {
        blocksInTransit = payload.Items

        blockHash := payload.Items[0]
        sendGetData(payload.AddrFrom, "block", blockHash)

        newInTransit := [][]byte{}
        for _, b := range blocksInTransit {
            if bytes.Compare(b, blockHash) != 0 {
                newInTransit = append(newInTransit, b)
            }
        }
        blocksInTransit = newInTransit
    }

    if payload.Type == "tx" {
        txID := payload.Items[0]

        if mempool[hex.EncodeToString(txID)].ID == nil {
            sendGetData(payload.AddrFrom, "tx", txID)
        }
    }
}

```

为了跟踪已下载block信息，当block的hash值完成传输后，将被保存到**blocksInTransit**中，这样做可以使我们从不同的节点下载block。一旦block置为transit状态后，将向inv消息的发送节点发送**getdata**消息并更新**blocksInTransit**。在真实的P2P网络中，将从不同的节点传输block信息。

在我们的实现中，inv消息中仅有一个block或交易的hash值，这也就是当**payload.Type == "tx"**时仅仅获取第一个hash值的原因。当mempool没有找到对应block或交易时，将发送**getdata**消息获取相关信息。

getdata消息

```

type getdata struct {
    AddrFrom string
    Type      string
    ID        []byte
}

```

getdata消息用于获取某个block或交易信息

```
func handleGetData(request []byte, bc *Blockchain) {
    ...
    if payload.Type == "block" {
        block, err := bc.GetBlock([]byte(payload.ID))

        sendBlock(payload.AddrFrom, &block)
    }

    if payload.Type == "tx" {
        txID := hex.EncodeToString(payload.ID)
        tx := mempool[txID]

        sendTx(payload.AddrFrom, &tx)
    }
}
```

该消息处理函数根据消息中的数据类型，返回block或者交易。需要注意的是，该函数没有检查对应的block或交易是否存在。

block和tx消息

```
type block struct {
    AddrFrom string
    Block    []byte
}

type tx struct {
    AddFrom    string
    Transaction []byte
}
```

这些消息表示实际的数据（block或交易信息）。

block消息处理函数很简单：

```

func handleBlock(request []byte, bc *Blockchain) {
    ...

    blockData := payload.Block
    block := DeserializeBlock(blockData)

    fmt.Println("Receieved a new block!")
    bc.AddBlock(block)

    fmt.Printf("Added block %x\n", block.Hash)

    if len(blocksInTransit) > 0 {
        blockHash := blocksInTransit[0]
        sendGetData(payload.AddrFrom, "block", blockHash)

        blocksInTransit = blocksInTransit[1:]
    } else {
        UTXOSet := UTXOSet{bc}
        UTXOSet.Reindex()
    }
}

```

当收到一个新block时，将该block加入blockchain中。如果还有其他待下载的block，将向相同的节点发送消息来获取block信息。当所有block均已下载，UTXO将被重建。

待完善:目前实现认为新block是有效的，后续在新block加入blockchain之前，应该对其进行有效性验证。

待完善:目前实现使用UTXOSet.Reindex()对UTXO进行重建，后续应该使用UTXOSet.Update(block)，避免重建整个blockchain带来的性能影响。

tx消息处理函数更复杂一些：

```

func handleTx(request []byte, bc *Blockchain) {
    ...
    txData := payload.Transaction
    tx := DeserializeTransaction(txData)
    mempool[hex.EncodeToString(tx.ID)] = tx

    if nodeAddress == knownNodes[0] {
        for _, node := range knownNodes {
            if node != nodeAddress && node != payload.AddFrom {
                sendInv(node, "tx", [][]byte{tx.ID})
            }
        }
    } else {
        if len(mempool) >= 2 && len(miningAddress) > 0 {
            MineTransactions:
            var txs []*Transaction

```

```

    for id := range mempool {
        tx := mempool[id]
        if bc.VerifyTransaction(&tx) {
            txs = append(txs, &tx)
        }
    }

    if len(txs) == 0 {
        fmt.Println("All transactions are invalid! Waiting for new ones...")
        return
    }

    cbTx := NewCoinbaseTX(miningAddress, "")
    txs = append(txs, cbTx)

    newBlock := bc.MineBlock(txs)
    UTXOSet := UTXOSet{bc}
    UTXOSet.Reindex()

    fmt.Println("New block is mined!")

    for _, tx := range txs {
        txID := hex.EncodeToString(tx.ID)
        delete(mempool, txID)
    }

    for _, node := range knownNodes {
        if node != nodeAddress {
            sendInv(node, "block", [][]byte{newBlock.Hash})
        }
    }

    if len(mempool) > 0 {
        goto MineTransactions
    }
}
}

```

首先，将新交易放入mempool（放入前应该进行有效性验证）。

```

if nodeAddress == knownNodes[0] {
    for _, node := range knownNodes {
        if node != nodeAddress && node != payload.AddFrom {
            sendInv(node, "tx", [][]byte{tx.ID})
        }
    }
}

```

检查当前节点是否为中心节点，在我们的视线中由于中心节点不进行挖矿操作，因此若是中心节点则将新交易转发给网络中的其他节点。

若当前节点为矿工节点，将进行如下操作。让我们一步一步来看：

```
if len(mempool) >= 2 && len(miningAddress) > 0 {
```

仅仅矿工节点设置 **miningAddress**，若当前矿工节点的mempool包含两个以上的交易时，开始挖矿：

```
for id := range mempool {
    tx := mempool[id]
    if bc.VerifyTransaction(&tx) {
        txs = append(txs, &tx)
    }
}

if len(txs) == 0 {
    fmt.Println("All transactions are invalid! Waiting for new ones...")
    return
}
```

首先，对mempool中的所有交易进行验证，忽略所有无效交易，若没有任何有效交易，则停止挖矿。

```
cbTx := NewCoinbaseTX(miningAddress, "")
txs = append(txs, cbTx)

newBlock := bc.MineBlock(txs)
UTXOSet := UTXOSet{bc}
UTXOSet.Reindex()

fmt.Println("New block is mined!")
```

所有有效交易和一个用于奖励的coinbase交易将被放到一个block中，当完成挖矿后，UTXO将被重建。

待完善:与之前一样，后续应该使用 *UTXOSet.Update(block)*，避免重建整个blockchain带来的性能影响。

```
for _, tx := range txs {
    txID := hex.EncodeToString(tx.ID)
    delete(mempool, txID)
}

for _, node := range knownNodes {
    if node != nodeAddress {
        sendInv(node, "block", [][]byte{newBlock.Hash})
    }
}

if len(mempool) > 0 {
    goto MineTransactions
}
```

当挖矿结束后，从mempool中移除交易信息，同时当前节点将向其他节点发送**inv**消息，其中包含最新挖到的block的hash值。

让我们对之前的场景进行演练。

首先，打开一个终端，通过**export NODE_ID=3000**命令将**NODE_ID**设置为3000。后续将使用**NODE 3000**或**NODE 3001**来表示哪个节点。

```
$ blockchain_go createblockchain -address CENTREAL_NODE
```

在节点上创建blockchain之后，blockchain中包含一个genesis block，该genesis block作为该节点的blockchain的标识符（在Bitcoin Core中，genesis block是硬编码的）。我们需要将该block单独保存，其他节点会用到该信息：

```
$ cp blockchain_3000.db blockchain_genesis.db
```

在**NODE 3001**节点上

打开一个新终端，将node ID设置为3001，该节点为钱包节点。使用**createwallet**命令创建3个钱包地址：**WALLET_1, WALLET_2, WALLET_3**。

在**NODE 3000**节点上

向这些钱包地址发送一些货币：

```
$ blockchain_go send -from CENTREAL_NODE -to WALLET_1 -amount 10 -mine
$ blockchain_go send -from CENTREAL_NODE -to WALLET_2 -amount 10 -mine
```

-mine参数表示将立即开始进行挖矿，由于初始时网络中没有矿工节点，因此需要该命令行参数。

启动节点：

```
$ blockchain_go startnode
```

该节点将一直运行直到该场景结束。

在**NODE 3001**节点上

将上面保存的genesis block作为该节点的blockchain的初始化信息：

```
$ cp blockchain_genesis.db blockchain_3001.db
```

启动节点：

```
$ blockchain_go startnode
```

该节点将从中心节点下载所有block。为了检查一切是否正常，停止该节点，然后查看钱包地址的余额：

```
$ blockchain_go getbalance -address WALLET_1
Balance of 'WALLET_1': 10

$ blockchain_go getbalance -address WALLET_2
Balance of 'WALLET_2': 10
```

由于**NODE 3001**中已经有blockchain，因此也可以查看**CENTRAL_NODE**地址的余额：

```
$ blockchain_go getbalance -address CENTRAL_NODE
Balance of 'CENTRAL_NODE': 10
```

在**NODE 3002**节点上

打开一个新终端，将node ID设置为3002，并生成一个钱包，该节点作为矿工节点使用。同理，将上面保存的genesis block作为该节点的blockchain的初始化信息：

```
$ cp blockchain_genesis.db blockchain_3002.db
```

启动节点：

```
blockchain_go startnode -miner MINER_WALLET
```

在**NODE 3001**节点上

发送一些货币：

```
$ blockchain_go send -from WALLET_1 -to WALLET_3 -amount 1
$ blockchain_go send -from WALLET_2 -to WALLET_4 -amount 1
```

在**NODE 3002**节点上

然后，快速切换到矿工节点所在终端，可以看到正在挖新的block！同样，也可以查看中心节点所在终端有何信息输出。

在**NODE 3001**节点上

切换到钱包节点所在终端，然后启动该节点：

```
$ blockchain_go startnode
```


该节点将下载新的block！

停止该节点并检查余额：

```
$ blockchain_go getbalance -address WALLET_1
Balance of 'WALLET_1': 9

$ blockchain_go getbalance -address WALLET_2
Balance of 'WALLET_2': 9

$ blockchain_go getbalance -address WALLET_3
Balance of 'WALLET_3': 1

$ blockchain_go getbalance -address WALLET_4
Balance of 'WALLET_4': 1

$ blockchain_go getbalance -address MINER_WALLET
Balance of 'MINER_WALLET': 10
```

一切如预期！

由于时间关系，没有描述如何实现一个真实的P2P网络协议。对于想了解比特币结束的读者，希望本文可以带来一些帮助，同时也希望给大家带来一些启示来更好的学习比特币技术！

附注：大家可以参考下面比特币网络协议的链接，通过实现**addr**消息来完善整个网络。**addr**消息十分重要，该消息使得网络中的节点可以发现彼此。

介绍以太坊的发展和现状，以太坊还处于发展中，变化较多，描述清楚现状以及未来的发展对于看清以太坊未来应用的场景有很大的好处

硬件环境：

- 处理器：Intel Core i5 2.6 GHz
- 内存：DDR3 8 GB 1600 MHz
- 显卡：Intel Iris 1536 MB

软件环境：

- 操作系统：OSX 10.11.6
- Brew版本：<https://brew.sh>安装最新版本，需提前安装好
- Go版本：1.9.2，需提前安装好
- Geth版本：1.7.3
- Mist版本：0.9.3

以太坊(Ethereum)客户端实现以太坊协议，用于接入以太坊网络进行各种操作，例如挖矿、交易等。网上有各种语言实现的以太坊客户端，其中官方提供了基于Go和C++的CLI客户端，即go-ethereum和cpp-ethereum。由于go-ethereum实现相对较完备且在学习Go语言，因此将使用go-ethereum作为客户端。

go-ethereum中包括许多命令，其中最重要的命令是Geth，该命令具备初始化、启动、挖矿、创建账户等许多重要功能，因此一般以Geth来代表以太坊客户端。

在Mac上安装Geth十分简单，有3种方法：

- 下载预编译好的安装包<https://geth.ethereum.org/downloads>
- 基于源码编译<https://github.com/ethereum/go-ethereum>
- 通过brew安装

本文将使用brew的安装方法。安装Geth执行如下命令：

```
# brew tap ethereum/ethereum
# brew install ethereum
需要等待一段时间...
```

说明：由于brew官方源中不包含ethereum，因此需要通过**brew tap**添加以太坊源（类似yum添加第三方源）。

安装完成后执行**geth version**命令查看版本信息：

```
Geth
Version: 1.7.3-stable
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.9.2
Operating System: darwin
...
```

当完成一个智能合约(Smart Contract)或DApp(Decentralization Application)应用时需要进行测试，可以接入以太坊网络进行测试，但是一方面效率很低，需要和公有链上的其他节点进行竞争；另一方面需要消耗真实的以太币(Ether)，后者恐怕是最不希望看到的情况。

这种情况下，可以使用以太坊建立私有链(Private Network)对智能合约和DApp应用在本地进行测试，不仅高效而且不用消耗真实的以太币。

Genesis文件

Genesis block（世纪块或始祖块）是区块链中第一个块，是唯一一个没有前序块的区块。以太坊允许通过自定义Genesis文件来创建Genesis block。拥有不同Genesis block的节点一定不会达成一致，这是由一致性算法保证的。

Genesis文件示例如下：

```
{
  "nonce": "0x0000000000000042",
  "difficulty": "0x020000",
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0x000000000000000000000000000000000000000000000000",
  "timestamp": "0x00",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extraData": "0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbbdb7a38e1e50b1b82fa",
  "gasLimit": "0x4c4b40",
  "config": {
    "chainId": 15,
    "homesteadBlock": 0,
    "eip155Block": 0,
    "eip158Block": 0
  },
  "alloc": { }
}
```

mixhash 和 nonce

mixhash长度为256位，nonce长度为64位，两者用于PoW系统。矿工挖矿时需要借助于两者不断计算得到一个值，当该值在数学意义上满足某种条件（参考Yellowpaper, 4.3.4. Block Header Validity, (44)、Yellowpaper, 11.5. Mining Proof-of-Work）时则证明挖到block。

difficulty

挖矿难度等级，通过该值确定挖矿目标。挖矿目标是根据前序block的难度等级和timestamp计算得到的。难度等级越高，挖矿时间越长，进而交易确认生效时间也越长。对于私有链，可以降低难度等级便于快速测试。

alloc

alloc可以包含一些钱包，这样通过以太坊特有的“预售”特性，可以提前给这些钱包充一定数量的ether。私有链中挖矿难度很低，因此不需要特别设置该选项。

coinbase

The 160-bit address to which all rewards (in Ether) collected from the successful mining of this block have been transferred. They are a sum of the mining reward itself and the Contract transaction execution refunds. Often named “beneficiary” in the specifications, sometimes “etherbase” in the online documentation. This can be anything in the Genesis Block since the value is set by the setting of the Miner when a new Block is created.

timestamp

A scalar value equal to the reasonable output of Unix time() function at this block inception. This mechanism enforces a homeostasis in terms of the time between blocks. A smaller period between the last two blocks results in an increase in the difficulty level and thus additional computation required to find the next valid block. If the period is too large, the difficulty, and expected time to the next block, is reduced. The timestamp also allows verifying the order of block within the chain (Yellowpaper, 4.3.4. (43)).

parentHash

长度256位，指向前序block，Genesis block没有前序block，填充为0。

extraData

选填项，长度32字节，可以填充一些除了交易外的数据。

gasLimit

A scalar value equal to the current chain-wide limit of Gas expenditure per block. High in our case to avoid being limited by this threshold during tests. Note: this does not indicate that we should not pay attention to the Gas consumption of our Contracts.

创建一个目录用于保存私有链相关信息：

```
# mkdir -p $HOME/share/q-btc/data
```

将上面内容保存到文件中，命名为PrivateGenesis.json：

```
# cd $HOME/share/q-btc
# vi PrivateGenesis.json
```

初始化节点

使用上面创建的Genesis文件初始化节点：

```
# cd $HOME/share/q-btc/data
# geth init PrivateGenesis.json --datadir $HOME/share/q-btc/data
WARN [xx-xx|13:26:56] No etherbase set and no accounts found as default
INFO [xx-xx|13:26:56] Allocated cache and file handles          database=$HOME/share/q-
btc/data/geth/chaindata cache=16 handles=16
INFO [xx-xx|13:26:56] Writing custom genesis block
INFO [xx-xx|13:26:56] Successfully wrote genesis state          database=chaindata hash
=611596...424d04
INFO [xx-xx|13:26:56] Allocated cache and file handles          database=$HOME/share/q-
btc/data/geth/lightchaindata cache=16 handles=16
INFO [xx-xx|13:26:56] Writing custom genesis block
INFO [xx-xx|13:26:56] Successfully wrote genesis state          database=lightchaindata
hash=611596...424d04
```

初始化后，在\$HOME/share/q-btc/data生成一系列目录及文件：

```
data
├── geth
│   ├── LOCK
│   ├── chaindata
│   │   ├── 000002.ldb
│   │   ├── 000003.log
│   │   ├── CURRENT
│   │   ├── LOCK
│   │   ├── LOG
│   │   └── MANIFEST-000004
│   ├── lightchaindata
│   │   ├── 000001.log
│   │   ├── CURRENT
│   │   ├── LOCK
│   │   ├── LOG
│   │   └── MANIFEST-000000
│   ├── nodekey
│   ├── nodes
│   │   ├── 000001.log
│   │   ├── CURRENT
│   │   ├── LOCK
│   │   ├── LOG
│   │   └── MANIFEST-000000
│   └── transactions.rlp
└── keystore
```

注意：由于使用--datadir自定义数据目录，后续geth命令使用时都需要加上该参数。默认数据目录为\$HOME/Library/Ethereum。

创建账户

geth account命令用于管理账户，支持许多子命令：

- geth account list 打印所有账户
- geth account new 创建新账户
- geth account update 更新账户信息
- geth account import 导入私钥生成新的账户
- geth account help 打印帮助

初始化私有链后没有任何账户：

```
# geth --datadir $HOME/share/q-btc/data account list
WARN [xx-xx|22:27:19] No etherbase set and no accounts found as default
```

生成一个包含明文密码的文件，并创建一个账户：

```
# cd $HOME/share/q-btc
# echo 123 > account.pwd
# geth --datadir $HOME/share/q-btc/data --password account.pwd account new
Address: { 205c6e56f2b809d686b4afc42b241004c985c900 }
# geth --datadir $HOME/share/q-btc/data account list
Account #0: {205c6e56f2b809d686b4afc42b241004c985c900} keystore://$HOME/share/q-btc/data/keystore/UTC--2017-xx-xxT14-50-17.692945588Z--205c6e56f2b809d686b4afc42b241004c985c900
```

此时生成了一个新账户，账户地址为{ **205c6e56f2b809d686b4afc42b241004c985c900** }。

启动节点

若之前部署过以太坊，请先删除\$HOME/.ethash目录，通过如下命令启动节点：

```
# geth --datadir $HOME/share/q-btc/data \  
--identity "PrivateETH" \  
--nodiscover \  
--maxpeers 25 \  
--rpc \  
--rpcapi "*" \  
--rpcport 8545 \  
--rpccorsdomain "*" \  
--port 30303 \  
--syncmode "fast" \  
--cache=1024 \  
--networkid 1999 \  
console
```

命令行参数的含义：

--datadir "\$HOME/share/q-btc/data"

该选项设置私有链数据存储目录，默认目录为\$HOME/Library/Ethereum。建议选择一个独立于存放以太坊公有链数据目录的其它目录。

--identity

节点身份标识。

--nodiscover

若不设置该选项，当私有链节点的networkid与其他节点的networkid相同时，其他节点会将私有链节点自动加入其所在的私有链中，这是不希望看到的（当然若genesis block不同也会不添加成功，不过通过admin.peers可以看到一会有peer一会又没有了，这就是被发现添加，但是genesis block不同而添加失败）。通过该选项，可以禁止私有链节点被其他节点发现，除非手动加入。

--maxpeers 25

该选项指定可以加入到私有链网络的最大节点数，默认25。当设置为0时，任何节点都无法加入私有链网络。

--rpc

该选项将启用HTTP-RPC接口，geth默认启用。

--rpcapi "db,eth,net,web3"

该选项指定可以通过HTTP-RPC接口访问哪些API。默认情况下，geth启用所有IPC接口的所有API，以及RPC接口的db,eth,net,web3这几类API。

--rpcport "8545"

HTTP-RPC服务监听端口，默认为8545。

--rpccorsdomain "*"

该选项指定接收来自于哪些URL连接的初始请求。真实环境中，应该设置一个特定的URL。由于私有链或私有链不涉及真实的以太币，因此可以设置为*通配符便于测试，这样可以使用站点（如Browser Solidity）或者DApps（如Notareth）连接网络。

--port "30303"

P2P网络监听端口，通过该端口可以手动地连接到其它节点。

--syncmode "fast"

blockchain数据同步模式：full，fast或light。

--cache=1024

缓存大小，单位MB，最小16，默认120。

--networkid 1999

网络ID，用于标识私有链网络，不同私有链的网络ID不同。默认为1，1表示太坊公有链；2表示Morden私有链网络，已不使用；3表示Ropsten私有链网路；4表示Rinkeby私有链网络。

私有链启动后，会打印如下信息：

```

INFO [xx-xx|23:40:46] Starting peer-to-peer node           instance=Geth/PrivateET
H/v1.7.3-stable/darwin-amd64/go1.9.2
INFO [xx-xx|23:40:46] Allocated cache and file handles       database=$HOME/share/q-
btc/data/geth/chaindata cache=1024 handles=1024
INFO [xx-xx|23:40:46] Initialised chain configuration   config="{ChainID: 15 Ho
mestead: 0 DAO: <nil> DAOSupport: false EIP150: <nil> EIP155: 0 EIP158: 0 Byzantium: <
nil> Engine: unknown}"
INFO [xx-xx|23:40:46] Disk storage enabled for ethash caches  dir=$HOME/share/q-btc/d
ata/geth/ethash count=3
INFO [xx-xx|23:40:46] Disk storage enabled for ethash DAGs   dir=$HOME/.ethash
count=2
INFO [xx-xx|23:40:46] Initialising Ethereum protocol   versions="[63 62]" netw
ork=1999
INFO [xx-xx|23:40:46] Loaded most recent local header       number=0 hash=611596...42
4d04 td=131072
INFO [xx-xx|23:40:46] Loaded most recent local full block    number=0 hash=611596...42
4d04 td=131072
INFO [xx-xx|23:40:46] Loaded most recent local fast block   number=0 hash=611596...42
4d04 td=131072
INFO [xx-xx|23:40:46] Loaded local transaction journal   transactions=0 dropped=
0
INFO [xx-xx|23:40:46] Regenerated local transaction journal transactions=0 accounts
=0
INFO [xx-xx|23:40:46] Starting P2P networking
INFO [xx-xx|23:40:47] RLPx listener up                self="enode://2d59b92a8
45dc555276ae565a5b26d43c2c4e1505efc84666b1bd893d7a0e050d0a26af72618d32aa33b35c7034d9f0
41bbbc019b19c4d21c23393af0dad6ef0@[::]:30303?discport=0"
INFO [xx-xx|23:40:47] IPC endpoint opened: $HOME/share/q-btc/data/geth.ipc
INFO [xx-xx|23:40:47] HTTP endpoint opened: http://127.0.0.1:8545
Welcome to the Geth JavaScript console!

instance: Geth/PrivateETH/v1.7.3-stable/darwin-amd64/go1.9.2
coinbase: 0x205c6e56f2b809d686b4afc42b241004c985c900
at block: 0 (Thu, 01 Jan 1970 08:00:00 CST)
datadir: $HOME/share/q-btc/data
modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.
0 web3:1.0

> INFO [xx-xx|23:40:49] Mapped network port           proto=tcp extport=303
03 intport=30303 interface="UPNP IGDv1-PPP1"

```

打印的信息主要包含私有链的配置信息，如缓存存放目录、数据存放目录、DAG存放目录、IPC地址等。随后进入Geth JavaScript控制台（>为控制台提示符），该控制台可执行JavaScript代码，同时内置的许多以太坊对象：

- eth 用于操作区块链
- net 用于查看p2p网络状态
- admin 用于管理节点
- miner 用于启动&停止挖矿
- personal 用于管理账户
- txpool 用于查看交易内存池
- web3 上述对象的父对象，此外包括一些单位换算对象

挖矿

启动节点进入控制台后，就可以开始挖矿：

```

> miner.start(1)
INFO [xx-xx|00:01:06] Updated mining threads          threads=1
INFO [xx-xx|00:01:06] Transaction pool price threshold updated price=18000000000
INFO [xx-xx|00:01:06] Starting mining operation
null
> INFO [xx-xx|00:01:06] Commit new mining work          number=1 txs=0 uncles
=0 elapsed=2.244ms
INFO [xx-xx|00:01:21] Generating DAG in progress        epoch=0 percentage=1 el
apsed=13.218s
INFO [xx-xx|00:01:22] Generating DAG in progress        epoch=0 percentage=2 el
apsed=14.693s
INFO [xx-xx|00:01:24] Generating DAG in progress        epoch=0 percentage=3 el
apsed=16.200s
.....
INFO [xx-xx|00:06:54] Generating DAG in progress        epoch=1 percentage=97 e
lapsed=3m0.262s
INFO [xx-xx|00:06:55] Generating DAG in progress        epoch=1 percentage=98 e
lapsed=3m1.942s
INFO [xx-xx|00:06:57] Generating DAG in progress        epoch=1 percentage=99 e
lapsed=3m3.726s
INFO [xx-xx|00:06:57] Generated ethash verification cache epoch=1 elapsed=3m3.729
s
INFO [xx-xx|00:07:07] mined potential block            number=2 hash=af3ac8...f49
b4d
INFO [xx-xx|00:07:07] Commit new mining work          number=3 txs=0 uncles=0
elapsed=2.094ms
INFO [xx-xx|00:07:07] Successfully sealed new block    number=3 hash=7498af...37
9059
INFO [xx-xx|00:07:07] mined potential block            number=3 hash=7498af...379
059
INFO [xx-xx|00:07:07] Commit new mining work          number=4 txs=0 uncles=0
elapsed=192.358μs
INFO [xx-xx|00:07:08] Successfully sealed new block    number=4 hash=c3e532...36
a2ba
.....

```

通过设置`mine.start(n)`的参数`n`来控制同时挖矿的线程数，默认使用创建的第一个账户进行挖矿。启动挖矿之初，先要创建DAG文件，当创建完成后（`percentage=99`）会打印DAG验证信息（上面红色字体），然后就会开始挖矿。此时会发现挖矿速度非常快，远远高于以太坊公有链的速度，这是因为有意在Genesis文件中将挖矿难度变得非常低：**"nonce":**

"0x000000000000000042"，这样便于在测试网络中快速挖到以太币进行测试。

什么是DAG？

Ethash是一个工作量证明(PoW)的系统，为了实现PoW需要准备大小约为1GB的数据集，称为DAG。由于DAG数据集的生成需要花费一定的时间，因此第一次生成后该数据集将被缓存起来便于其他客户端共享该数据集。

DAG数文件存放在：

- Mac/Linux：\$HOME/.ethash/full-R<REVISION>-<SEEDHASH>
- Windows：\$HOME/AppData/Local/Ethash/full-R<REVISION>-<SEEDHASH>

DAG文件以8字节的魔数0xfee1deadbaddcafe开头（大端存储，fe ca dd ba ad de e1 fe）。

可以使用如下命令停止挖矿，成功会返打印true：

```
> miner.stop()  
true
```

查看当前账户列表，目前仅有一个账户：

```
> eth.accounts  
["0x205c6e56f2b809d686b4afc42b241004c985c900"]
```

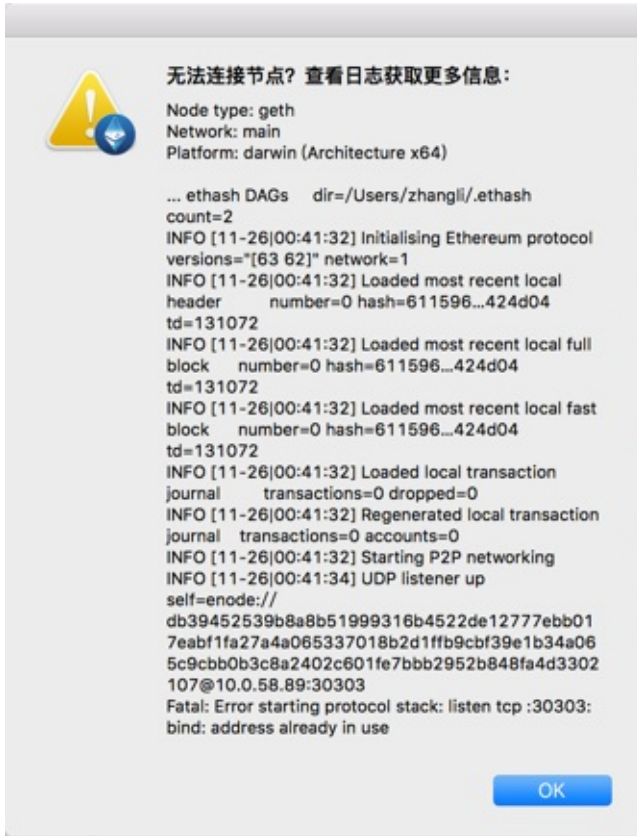
查看该用户/矿工的余额，eth.getBalance返回值是以太币最小单位（Wei，Ether = 1018Wei）的数量，可以使用web3.fromWei转换成实际的以太币数量：

```
> eth.getBalance(eth.accounts[0])  
1.59e+21  
> primary = eth.accounts[0]  
"0x205c6e56f2b809d686b4afc42b241004c985c900"  
> balance = web3.fromWei(eth.getBalance(primary), "ether")  
1590
```

Mist是以太坊官方提供的钱包应用，为geth的GUI版本。官方下载地址：
<https://github.com/ethereum/mist/releases>，本文中使用的0.9.3版本。

连接私有链

启动Mist，过一段时间后会出現如下错误：



为什么会这样呢？原因如下：

Mist启动后查找本地默认ipc是否存在（Mac默认文件为\$HOME/Library/Ethereum/geth.ipc；Windows默认文件为.\pipe\geth.ipc；Linux默认文件为/.ethereum/geth.ipc），如果存在则发送rpc请求获取相关数据，否则将从公有链网络下载Genesis block等信息初始化本地链并启动节点。

而在本例中，节点初始化和启动均使用了自定义目录\$HOME/share/q-btc/data，因此Mist没有连接到私有链网络，而是连接到公有链获取信息初始化并启动节点。

此时，本地节点和Mist启动的新节点均使用的是默认端口30303，发生了端口冲突，因此出现上面的错误。

为了使Mist连接到本地节点，可以在启动Mist时设--rpc参数指定ipc文件：

```
# cd /Applications/Mist.app/Contents/  
# ./Mist --rpc $HOME/share/q-btc/data/geth.ipc
```

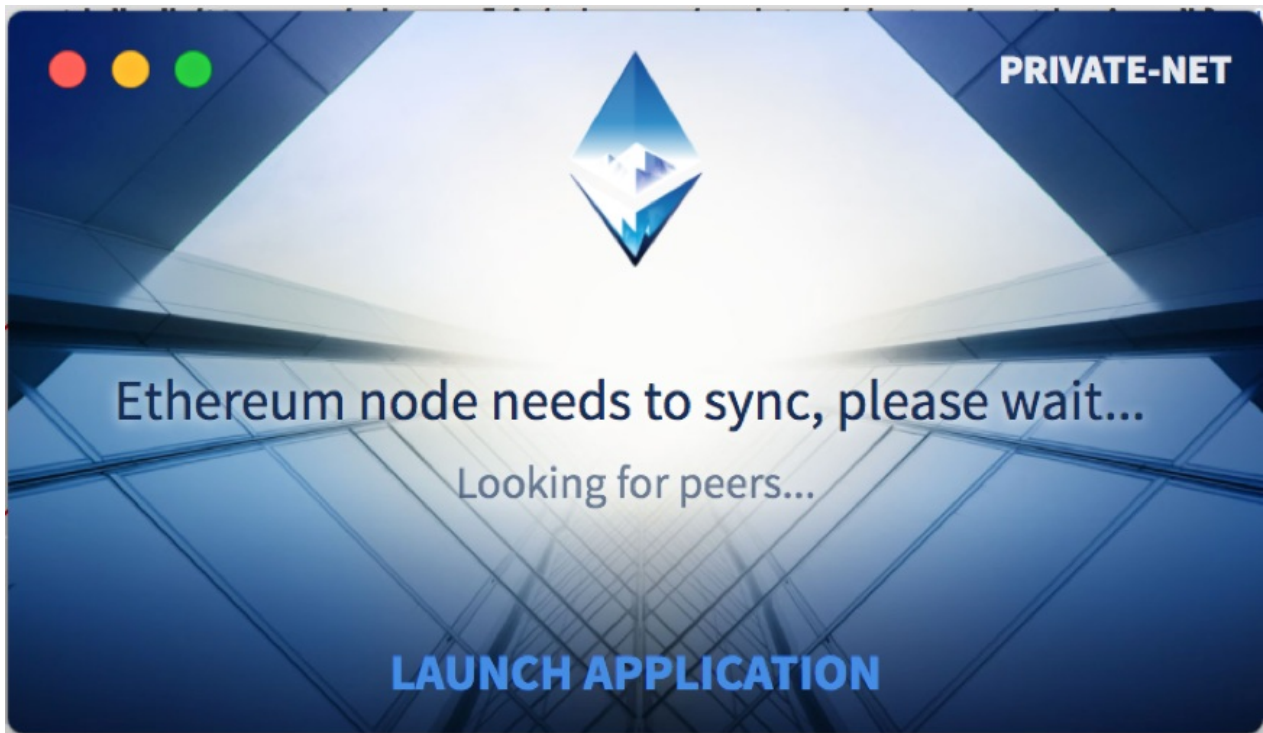
```

[xxxx-xx-xx 01:04:34.881] [INFO] main - Running in production mode: true
[xxxx-xx-xx 01:04:34.912] [INFO] EthereumNode - undefined null 'fast'
[xxxx-xx-xx 01:04:34.913] [INFO] EthereumNode - Defaults loaded: geth main fast
[xxxx-xx-xx 01:04:35.258] [INFO] main - Starting in Mist mode
[xxxx-xx-xx 01:04:35.349] [INFO] Db - Loading db: $HOME/Library/Application Support/Mist/mist.lokidb
[xxxx-xx-xx 01:04:35.358] [INFO] Windows - Creating commonly-used windows
[xxxx-xx-xx 01:04:35.358] [INFO] Windows - Create secondary window: loading, owner: notset
[xxxx-xx-xx 01:04:35.456] [INFO] updateChecker - Check for update...
[xxxx-xx-xx 01:04:38.396] [INFO] Windows - Create primary window: main, owner: notset
[xxxx-xx-xx 01:04:38.404] [INFO] Windows - Create primary window: splash, owner: notset
[xxxx-xx-xx 01:04:38.786] [INFO] ipcCommunicator - Backend language set to: zh
[xxxx-xx-xx 01:04:39.208] [INFO] (ui: splashscreen) - Web3 already initialized, re-using provider.
[xxxx-xx-xx 01:04:39.275] [INFO] (ui: splashscreen) - Meteor starting up...
[xxxx-xx-xx 01:04:39.453] [INFO] ClientBinaryManager - Initializing...
[xxxx-xx-xx 01:04:39.454] [INFO] ClientBinaryManager - Checking for new client binaries config from: https://raw.githubusercontent.com/ethereum/mist/master/clientBinaries.json
[xxxx-xx-xx 01:04:40.536] [INFO] ClientBinaryManager - No "skippedNodeVersion.json" found.
[xxxx-xx-xx 01:04:40.537] [INFO] ClientBinaryManager - Initializing...
[xxxx-xx-xx 01:04:40.538] [INFO] ClientBinaryManager - Resolving platform...
[xxxx-xx-xx 01:04:40.538] [INFO] ClientBinaryManager - Calculating possible clients...
[xxxx-xx-xx 01:04:40.539] [INFO] ClientBinaryManager - 1 possible clients.
[xxxx-xx-xx 01:04:40.540] [INFO] ClientBinaryManager - Verifying status of all 1 possible clients...
[xxxx-xx-xx 01:04:40.540] [INFO] ClientBinaryManager - Verify Geth status ...
[xxxx-xx-xx 01:04:40.574] [INFO] ClientBinaryManager - Checking for Geth sanity check ...
[xxxx-xx-xx 01:04:40.574] [INFO] ClientBinaryManager - Checking for Geth sanity check ...
[xxxx-xx-xx 01:04:40.574] [INFO] ClientBinaryManager - Checking sanity for Geth ...
[xxxx-xx-xx 01:04:40.579] [INFO] ClientBinaryManager - Checking sanity for Geth ...
[xxxx-xx-xx 01:04:40.662] [ERROR] ClientBinaryManager - Sanity check failed for Geth Error: Unable to find "1.7.2" in Geth output
    at Promise.resolve.then.then.then (/Applications/Mist.app/Contents/Resources/app.asar/node_modules/ethereum-client-binaries/src/index.js:635:17)
    at <anonymous>
    at process._tickCallback (internal/process/next_tick.js:109:7)
[xxxx-xx-xx 01:04:40.795] [INFO] Sockets/node-ipc - Connect to {"path": "$HOME/share/q-btc/data/geth.ipc"}
[xxxx-xx-xx 01:04:40.797] [INFO] Sockets/node-ipc - Connected!
[xxxx-xx-xx 01:04:40.798] [INFO] NodeSync - Ethereum node connected, re-start sync
[xxxx-xx-xx 01:04:40.798] [INFO] NodeSync - Starting sync loop
[xxxx-xx-xx 01:04:40.799] [INFO] Sockets/3 - Connect to {"path": "$HOME/share/q-btc/data/geth.ipc"}
[xxxx-xx-xx 01:04:40.810] [INFO] Sockets/3 - Connected!
[xxxx-xx-xx 01:04:40.835] [INFO] (ui: splashscreen) - Network is privatenet
[xxxx-xx-xx 01:04:40.836] [INFO] (ui: splashscreen) - Network is privatenet
[xxxx-xx-xx 01:04:43.383] [INFO] main - Connected via IPC to node.

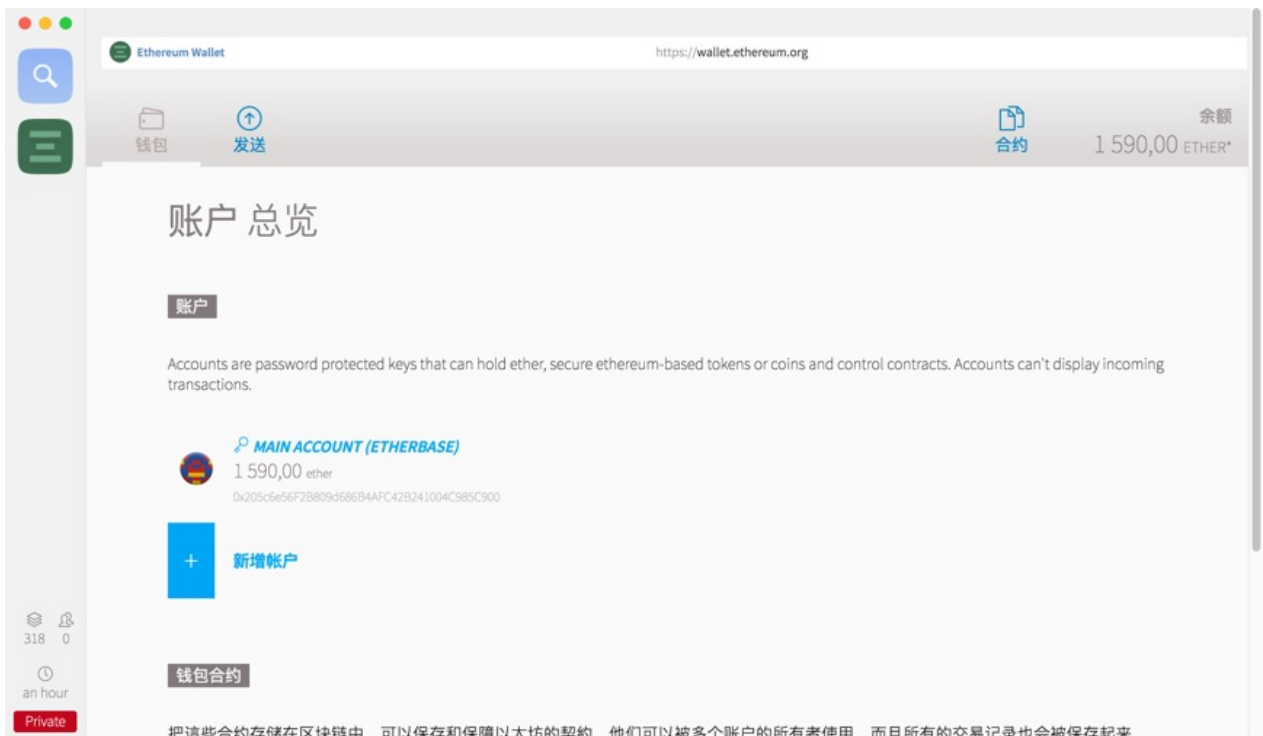
```

```
[xxxx-xx-xx 01:04:43.515] [INFO] updateChecker - App is up-to-date.
```

注意：“Sanity check failed for Geth Error”错误信息标识Mist所需的geth版本信息与从私有链获取到的geth版本信息不匹配，暂未发现该错误对后续应用造成影响，可以先忽略。



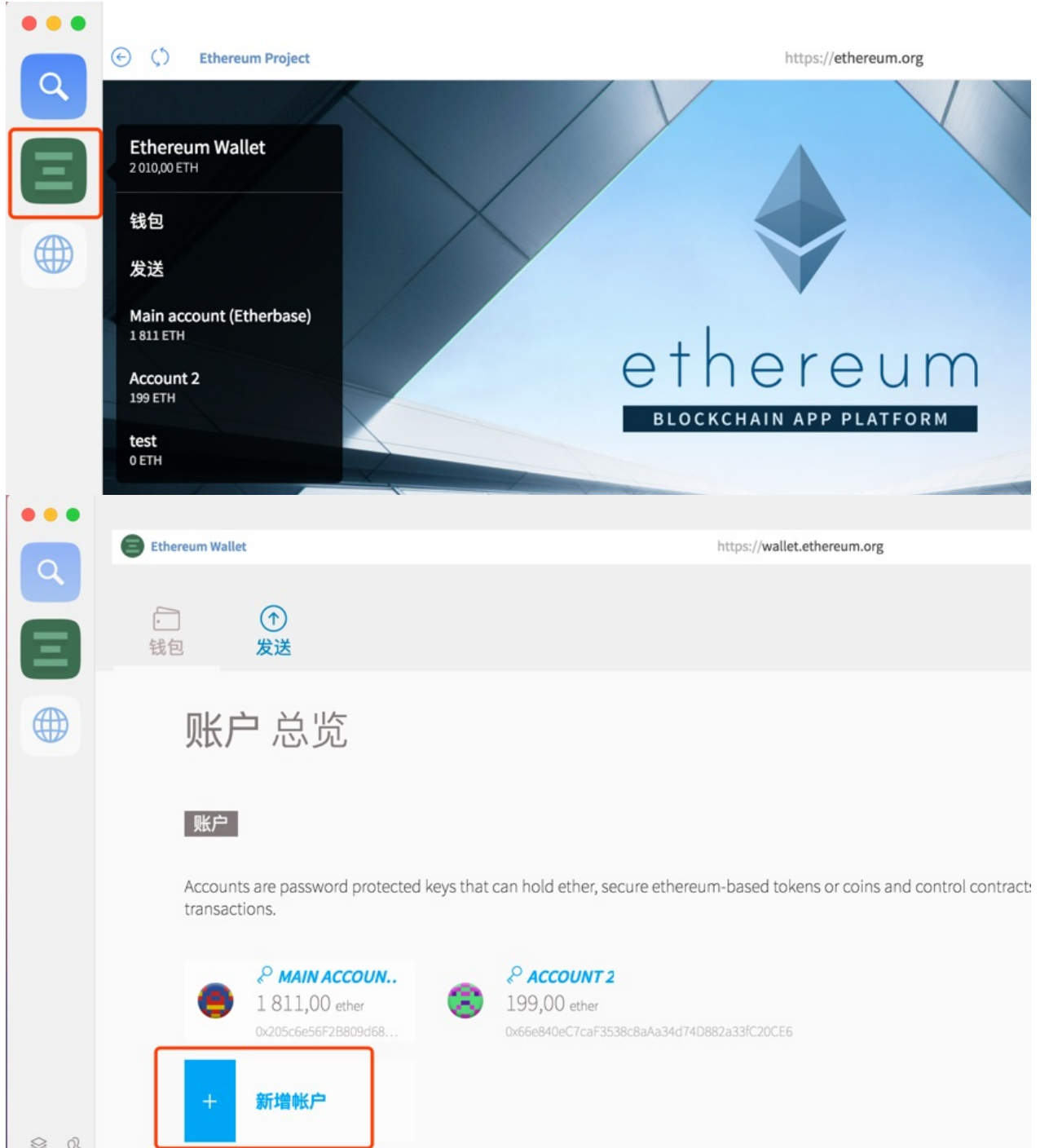
Mist界面启动后，右上角显示PRIVATE-NET，已经成功连接到私有链网络中了，点击LAUNCH APPLICATION，进入Mist主界面：

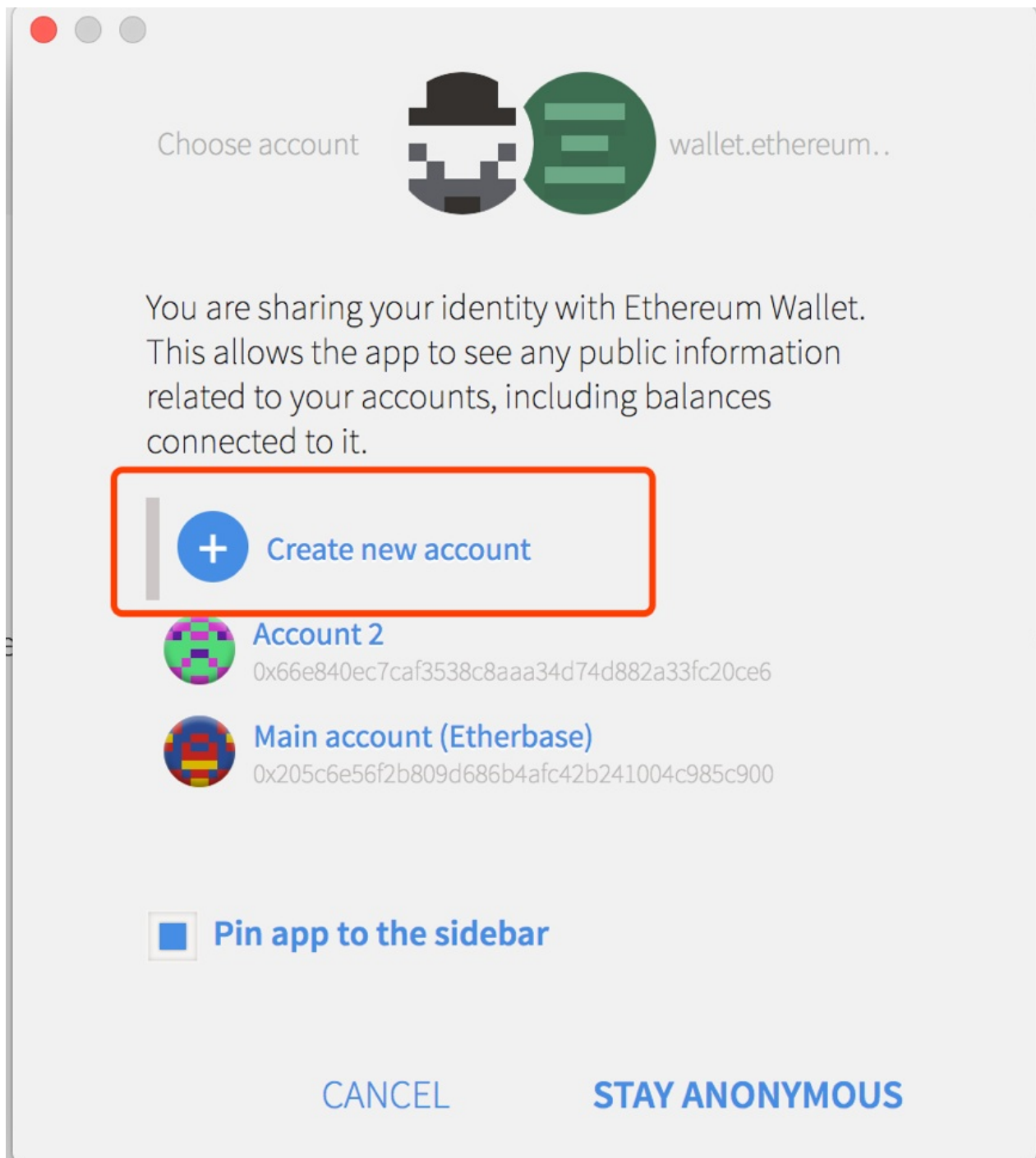


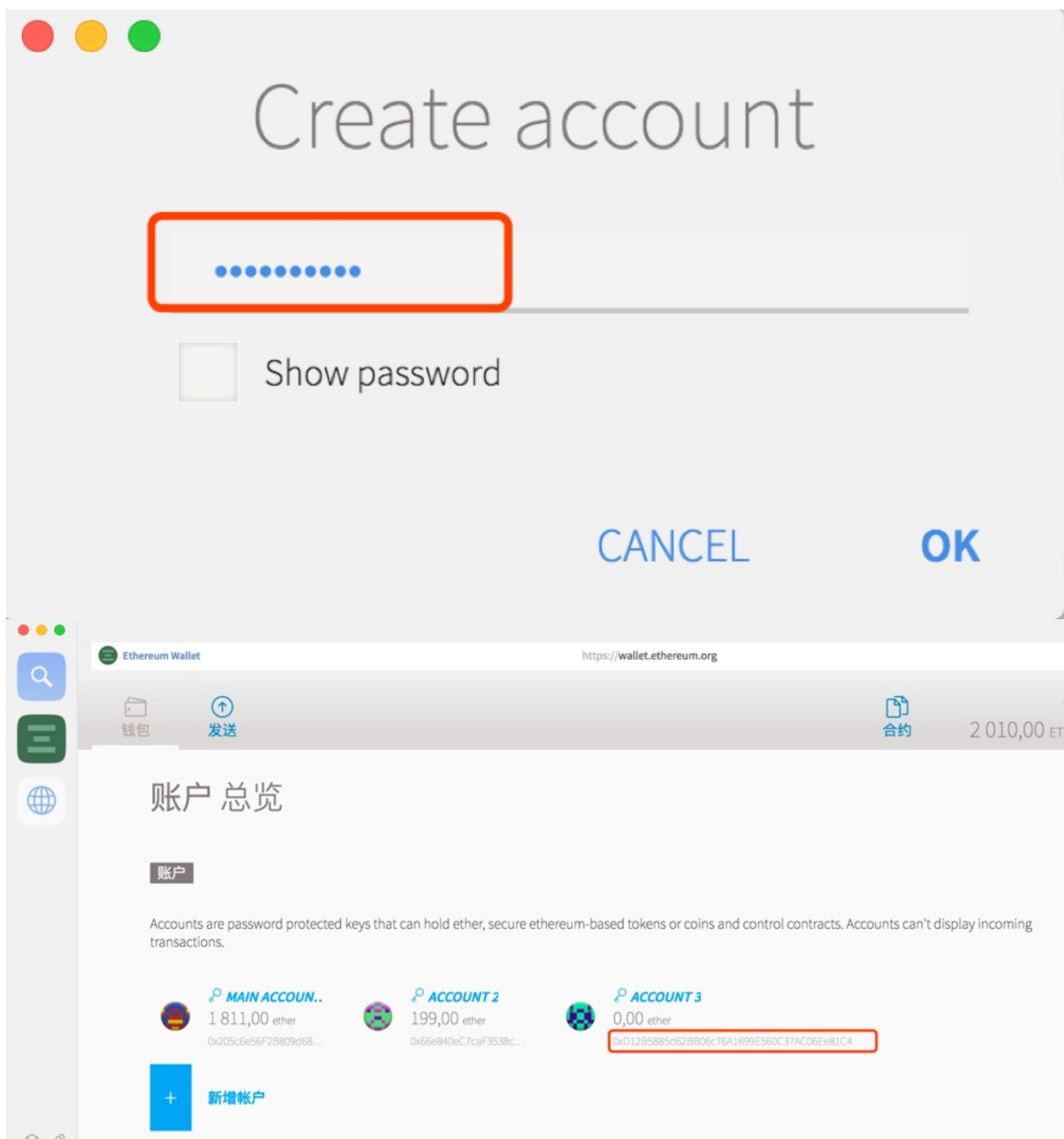
可以看到界面显示已经有一个账户，拥有1590个以太币，与之前geth命令行返回的信息一致。

创建账户

账户是一对公钥/私钥对，用于进行挖矿、交易、存储ether等活动。之前已经通过Geth创建了一个账户，该账户为Main Account，挖矿获得的ether默认存在该账户。账户可以有多个，下面通过Mist创建另一个账户







至此，新创建了一个账户 ACCOUNT3，该账户地址为

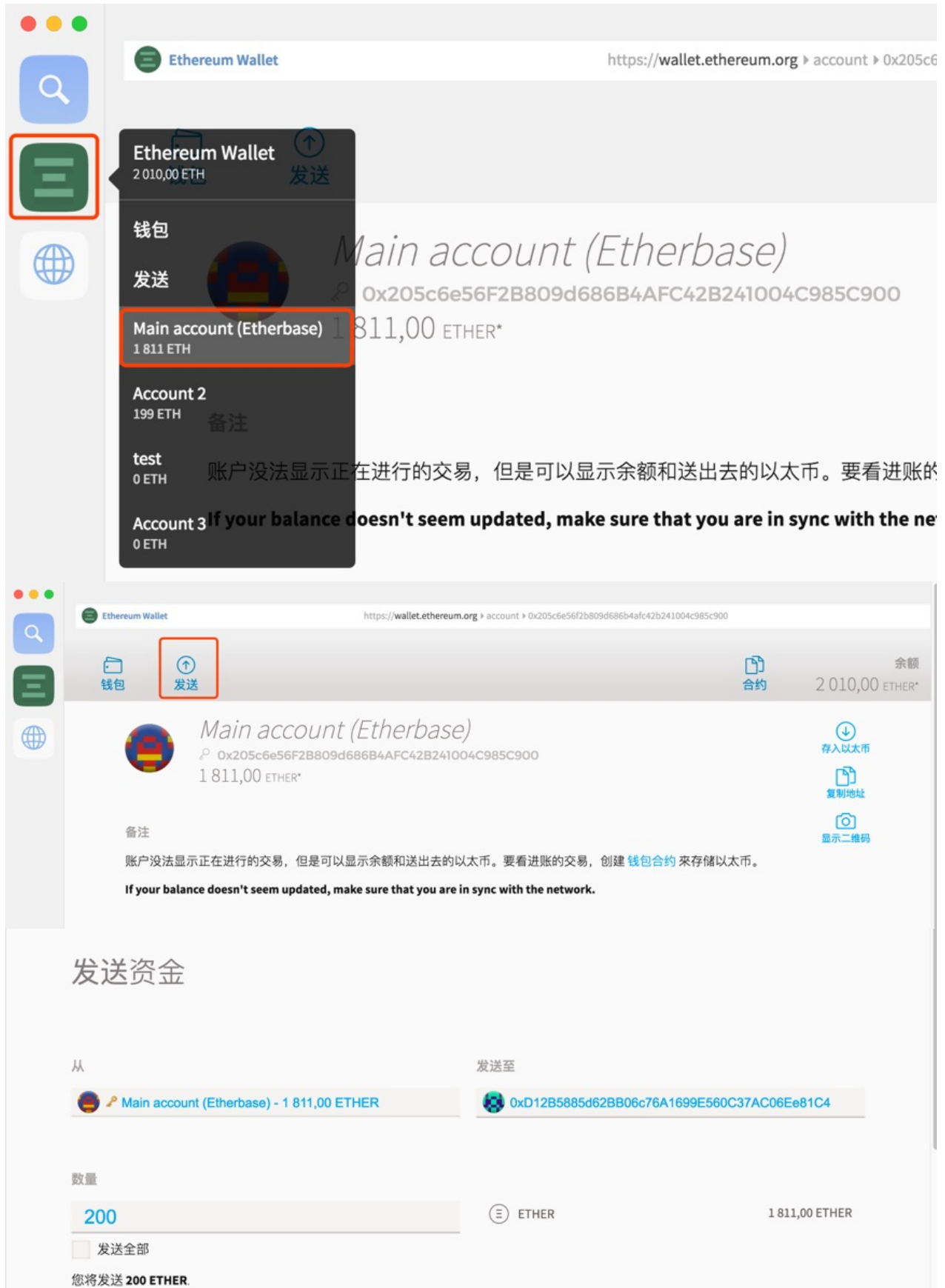
0xD12B5885d62BB06c76A1699E560C37AC06Ee81C4。

账户文件保存在--datadir目录中的keystore目录中：

```
Chris:keystore zhangli$ ls $HOME/share/q-btc/private_chain/data/keystore
UTC--2017-11-25T14-50-17.692945588Z--205c6e56f2b809d686b4afc42b241004c985c900
UTC--2017-11-26T13-28-26.329917826Z--66e840ec7caf3538c8aaa34d74d882a33fc20ce6
UTC--2017-11-26T14-17-13.507560584Z--d12b5885d62bb06c76a1699e560c37ac06ee81c4
```

创建交易

从MAIN ACCOUNT向ACCOUNT3转入一些ether，会创建一个交易：



以太坊 Mist

SHOW MORE OPTIONS

选择手续费

0,000378 ETHER

更便宜

更快


此数量为本笔交易手续费上限，您的交易耗时 约小于30 秒。

总共

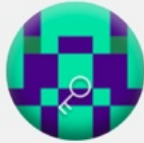
200,000378 ETHER

发送

Send transaction


0x205c...c900

200.00 ETHER


0xd12b...81c4

Estimated fee consumption

0.000378 ether (21,000 gas)

Provide maximum fee

0.002178 ether (121,000 gas)

Gas price

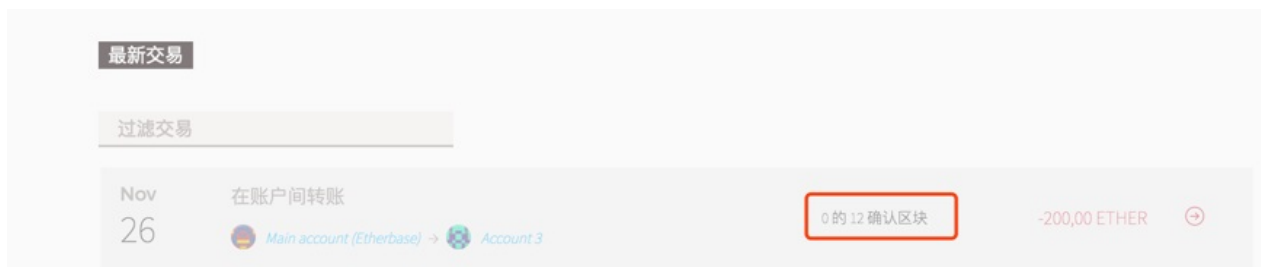
0.018 ether per million gas

.....

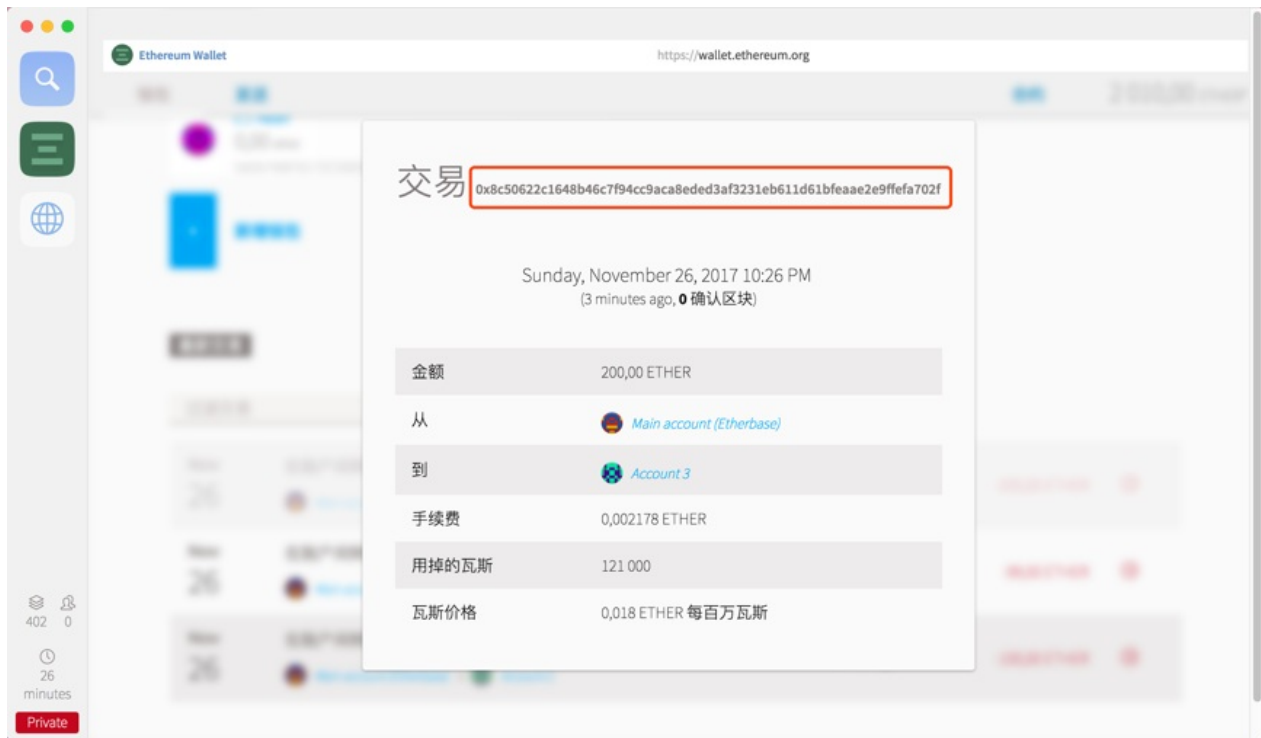
CANCEL

SEND TRANSACTION

输入MAIN ACCOUNT账户密码，SEND TRANSACTION后创建一个交易，可以在MAIN ACCOUNT账户的交易列表中看到：



点击该交易，可以看到交易信息：



交易ID为0x8c50622c1648b46c7f94cc9aca8eded3af3231eb611d61bfeaae2e9ffefa702f，此时console会显示日志：

```
> INFO [xx-xx|22:26:19] Submitted transaction fullhash=0x8c50622c1648b46c7f94cc9aca8eded3af3231eb611d61bfeaae2e9ffefa702f recipient=0xD12B5885d62BB06c76A1699E560C37AC06Ee81C4
```

该交易一直处于灰色，显示“0的12确认区块”，不会有任何变化，而ACCOUNT3的余额一直未0。这是因为目前没有任何节点进行挖矿操作，因此不会将该交易放入私有链中。



通过console启动挖矿：

```
> miner.start(1)
```



开始挖矿后，可以看到交易开始产生进度，信息也开始有所变化：

最新交易

过滤交易

Nov 26	在账户间转账  Main account (Etherbase) →  Account 3	3 的 12 确认区块	-200,00 ETHER	⊕
-----------	--	-------------	---------------	---


当交易完成时，交易状态如下，同时ACCOUNT3的余额变为200：


Nov 26 在账户间转账 7 minutes ago -200,00 ETHER ⊕
 Main account (Etherbase) →  Account 3

账户总览

账户

Accounts are password protected keys that can hold ether, secure ether transactions.

**MAIN ACCOUN..**
1 856,00 ether
0x205c6e56F2B809d68...

**ACCOUNT 3**
200,00 ether
0xD12B5885d62BB06c...

私有链中多个节点互联需要确保多个节点：

- 拥有相同的genesis block
- 拥有相同的networkid

同一机器内节点互联

可以在同一台机器上创建多个节点，只要节点间的端口不冲突即可，下面以创建两个节点为例。

目录规划如下：

```
# mkdir -p $HOME/share/q-btc/local-cluster/{data1,data2}
# cd $HOME/share/q-btc/local-cluster
# vi PrivateGenesis.json
```

打开一个终端，初始化、启动节点1：

```
# cd $HOME/share/q-btc/local-cluster
# geth init PrivateGenesis.json --datadir $HOME/share/q-btc/local-cluster/data1
# geth --datadir $HOME/share/q-btc/local-cluster/data1 \
--identity "PrivateETH Node1" \
--nodiscover \
--maxpeers 25 \
--rpc \
--rpcapi "*" \
--rpcport 8545 \
--rpccorsdomain "*" \
--port 30303 \
--syncmode "fast" \
--cache=1024 \
--networkid 1999 \
console
打开另一个终端，初始化、启动节点2：
# cd $HOME/share/q-btc/local-cluster
# geth init PrivateGenesis.json --datadir $HOME/share/q-btc/local-cluster/data2
# geth --datadir $HOME/share/q-btc/local-cluster/data2 \
--identity "PrivateETH Node2" \
--nodiscover \
--maxpeers 25 \
--rpc \
--rpcapi "*" \
--rpcport 8546 \
--rpccorsdomain "*" \
--port 30304 \
--syncmode "fast" \
--cache=1024 \
--networkid 1999 \
console
```

分别在节点1/2查看peer列表，此时均为空：

```
> admin.peers
[]
```

分别在节点1/2查看block数量，此时均为0：

```
> eth.blockNumber
0
```

分别在节点1/2查看账户情况，此时均没有账户：

```
> eth.accounts
[]
```

为了验证节点间可以同步信息，在节点1上进行挖矿获取一些block。先创建一个账户（之前通过geth命令创建，下面使用console来创建），然后启动挖矿：

```
> personal.newAccount()
Passphrase: 输入密码
Repeat passphrase: 输入密码
"0x9f6a75647cc8d382f1c236b87c4d37003fa54dd5"
> eth.accounts
["0x9f6a75647cc8d382f1c236b87c4d37003fa54dd5"]
> miner.start(1)
INFO [xx-xx|00:40:07] Successfully sealed new block      number=1 hash=28a8d2...ce7f9f
INFO [xx-xx|00:40:07] mined potential block            number=1 hash=28a8d2...ce7f9f
INFO [xx-xx|00:40:07] Commit new mining work          number=2 txs=0 uncles=0 elapsed=797.4µs
INFO [xx-xx|00:40:08] Successfully sealed new block      number=2 hash=ac7d21...abc77e
.....
> miner.stop()
> eth.blockNumber
12
```

最终挖到12个block。

由于启动节点时增加了--nodiscover选项，因此需要手动添加，在console中通过admin.addPeer(nodeurl)函数添加peer。首先，要知道两个peer的nodeurl。

分别在节点1/2查看node信息：

节点1 node信息：

```
> admin.nodeInfo.enode
"enode://01dfc3a6cec378232668fddd07479e5607d4e1dd0b50d7ddb7f06c89003e8dc44e2b930db18d0daeccdaec3fdddb3ea1dace2ca022b516834d957c635187d399@[::]:30303?discport=0"
```

节点2 node信息：

```
> admin.nodeInfo.enode
"enode://eb6901ddf5f563e8fe42e744c3c63e96de5417531b2194da3d0dbf8ee68773d024261330ed6aeebe3ee1eea60be4947676f421f03f64bf875ee5e968ab45449@[::]:30304?discport=0"
```

在节点2上添加节点1的信息：

```

> admin.addPeer("enode://01dfc3a6cec378232668fddd07479e5607d4e1dd0b50d7ddb7f06c89003e8dc44e2b930db18d0daeccdaec3fdddb3ea1dace2ca022b516834d957c635187d399@[::]:30303?discport=0")
true
> admin.peers
[
  {
    caps: ["eth/63"],
    id: "01dfc3a6cec378232668fddd07479e5607d4e1dd0b50d7ddb7f06c89003e8dc44e2b930db18d0daeccdaec3fdddb3ea1dace2ca022b516834d957c635187d399",
    name: "Geth/PrivateETH Node1/v1.7.3-stable/darwin-amd64/go1.9.2",
    network: {
      localAddress: "[::1]:60802",
      remoteAddress: "[::1]:30303"
    },
    protocols: {
      eth: {
        difficulty: 1707456,
        head: "0xf7d4e393ec7963da14aaeb1ef4e74e1b365a4da8c666cf3c954fef7268597d1b",
        version: 63
      }
    }
  }
]
> INFO [xx-xx|00:45:07] Block synchronisation started
INFO [xx-xx|00:45:07] Imported new state entries count=1 elapsed=297.665
µs processed=1 pending=0 retry=0 duplicate=0 unexpected=0
INFO [xx-xx|00:45:08] Imported new block headers count=12 elapsed=1.374s
number=12 hash=f7d4e3...597d1b ignored=0
INFO [xx-xx|00:45:08] Imported new chain segment blocks=12 txs=0 mgas=0.
000 elapsed=3.237ms mgasps=0.000 number=12 hash=f7d4e3...597d1b
INFO [xx-xx|00:45:08] Fast sync complete, auto disabling

> eth.blockNumber
12

```

在节点2上先通过`admin.addPeer`添加节点1；然后使用`admin.peer`查看节点信息，可以看到节点已经添加；接下来输出一系列信息，“**Fast sync complete, auto disabling**”表示node2从node1同步信息结束；通过`eth.blockNumber`可以看到同步了12个block。

此时，再在节点1上启动挖矿，当挖到新的block时，节点2会打印如下信息，表示节点2在同步更新私有链：

```

INFO [xx-xx|00:50:09] Imported new chain segment blocks=1 txs=0 mgas=0.
000 elapsed=5.147ms mgasps=0.000 number=13 hash=f0e25d...3fc77f
INFO [xx-xx|00:50:19] Imported new chain segment blocks=1 txs=0 mgas=0.
000 elapsed=3.802ms mgasps=0.000 number=14 hash=998152...644f12
INFO [xx-xx|00:50:19] Imported new chain segment blocks=1 txs=0 mgas=0.
000 elapsed=3.908ms mgasps=0.000 number=15 hash=d6d0f4...d51980

```

跨机器间节点互联

跨机器间的节点互联和同一台机器内的节点互联一样，区别在于添加节点，例如机器2的IP为192.168.1.2，节点标识为

enode://999fc3a6cec378232668fddd07479e5607d4e1dd0b50d7ddb7f06c89003e8dc44e2b930db18d0daeccdaec3fdddb3ea1dace2ca022b516834d957c635187d399@[::]:30303?discport=0，机器1添加节点调用如下：

```
> admin.addPeer("enode://01dfc3a6cec378232668fddd07479e5607d4e1dd0b50d7ddb7f06c89003e8dc44e2b930db18d0daeccdaec3fdddb3ea1dace2ca022b516834d957c635187d399@192.168.1.2:30303?discport=0")
true
```

只需要将机器2节点标识中的[::]替换为机器2的IP即可。

添加节点通常会出现失败的情况，通常是由于：

- 机器间时间不同步。以太坊网络对于时间同步有很高的要求，两台机器即使有10几秒的差异也会产生一定问题
- 防火墙禁止UDP协议

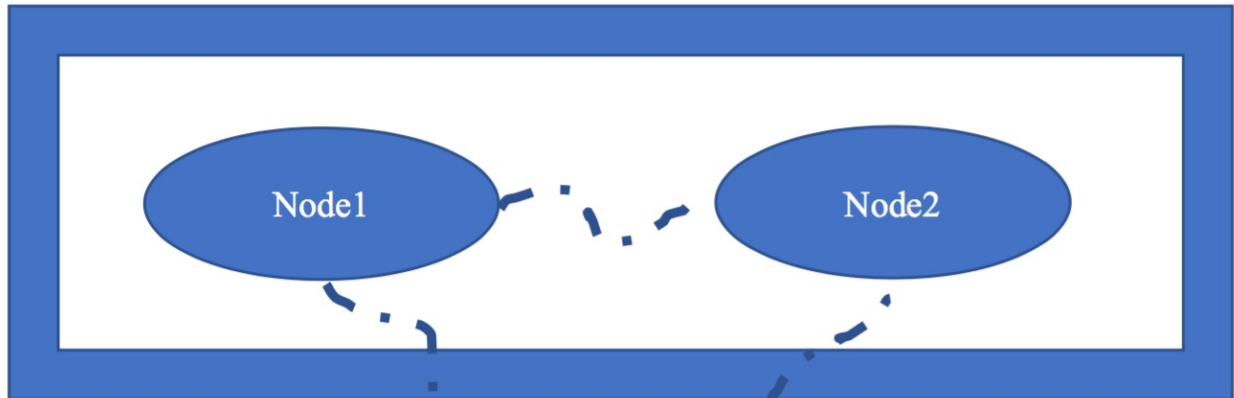
静态节点

上面通过admin.addPeer()来联机到这些节点，也可以通过配以一个文件static-node.js来声明静态节点，当节点启动后，会自动添加该文件中的节点。static-node.js放置于\${datadir}中，如\$HOME/share/q-btc/local-cluster/data1中，文件内容格式为：

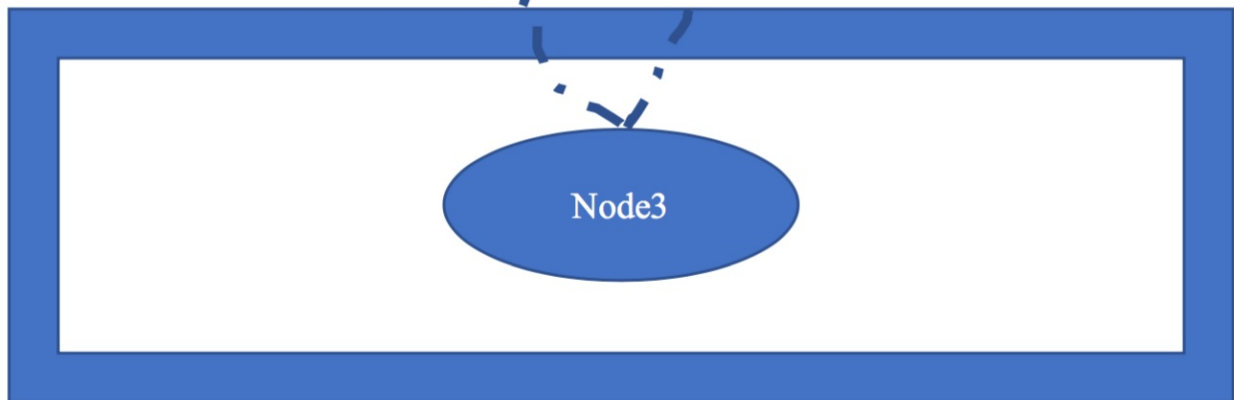
```
[
  enodeurl,
  enodeurl,
  ...
]
```

目前构成了一个**2机器3节点**的私有链网络：

机器 1 (192.168.1.1)



机器 2 (192.168.1.2)



Solidity是以太坊的智能合约语言，对齐语法和使用以实例的方式进行介绍

介绍如何通过以太坊开发、测试、部署、发布自己的DApp

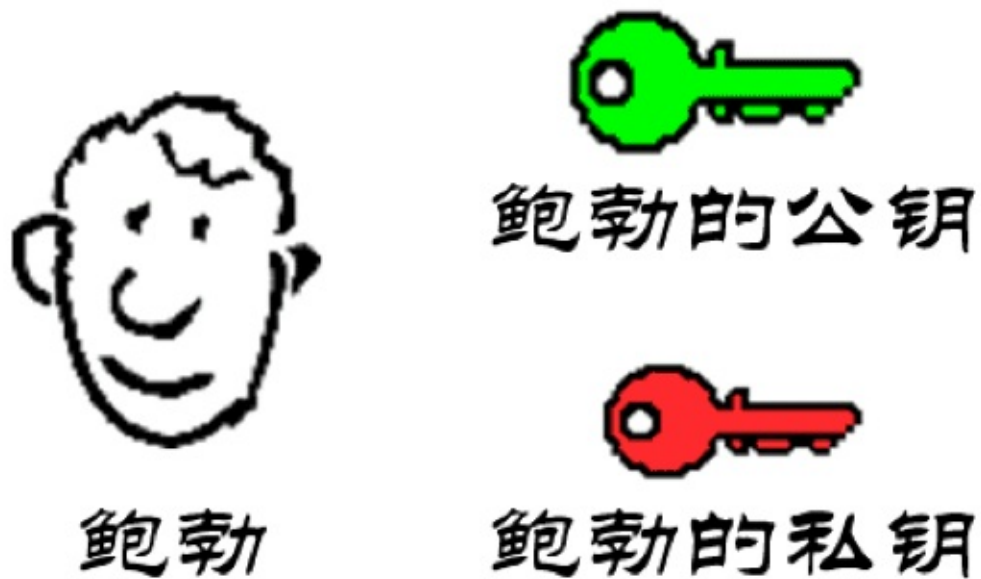
待续...

公钥和私钥俗称非对称加密，公钥和私钥称为密钥对：用公钥加密的内容只能用私钥解密；用私钥加密的内容只能用公钥解密。

公钥和私钥一般有两种用途：加密（公钥加密，私钥解密）和认证（私钥加密，公钥解密）。

以下示例（引用自[中文翻译](#)或[英文原文](#)）生动形象地说明两种应用场景，其中1-4是加密过程，5-9是认证过程，而10-12解释了认证中心的必要性：

(1) 鲍勃有两把钥匙，一把是公钥，另一把是私钥



(2) 鲍勃把公钥送给他的朋友们----帕蒂、道格、苏珊----每人一把。



(3) 苏珊要给鲍勃写一封保密的信。她写完后用鲍勃的公钥加密，就可以达到保密的效果。



苏珊

"Hey Bob,
how about
lunch at
Taco Bell. I
hear they
have free
refills!"



公钥加密

HNFmsEm6Un
BejhhyCGKO
KJUxhiygSBC
EiC0QYIh/Hn
3xgiKBcyLK1
UcYiYlxx2lCF
HDC/A

(4) 鲍勃收信后，用私钥解密，就看到了信件内容。这里要强调的是，只要鲍勃的私钥不泄露，这封信就是安全的，即使落在别人手里，也无法解密。



鲍勃

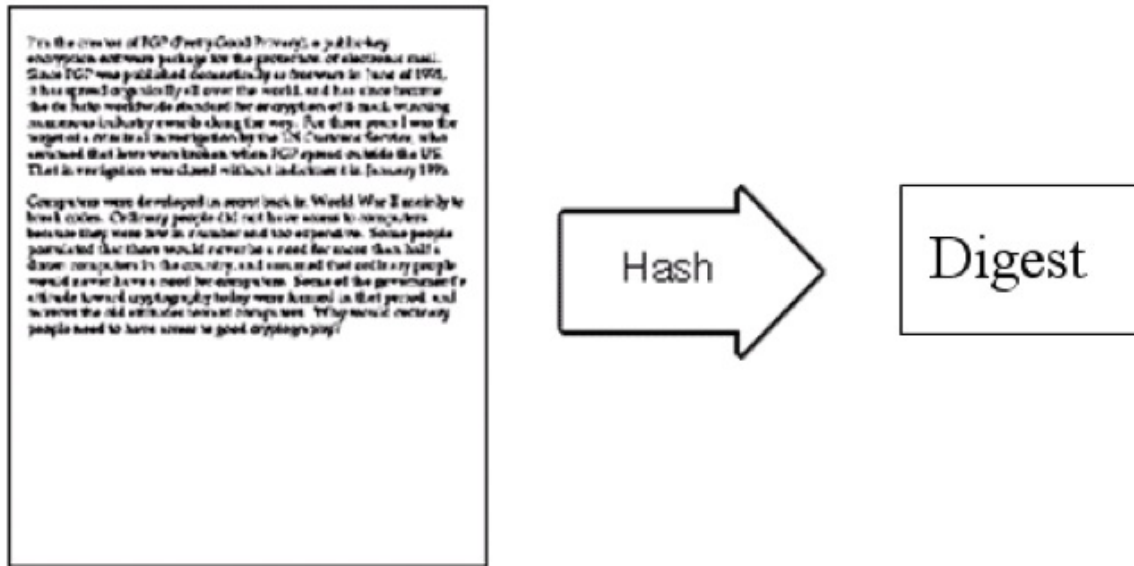
HNFmsEm6Un
BejhhyCGKO
KJUxhiygSBC
EiC0QYIh/Hn
3xgiKBcyLK1
UcYiYlxx2lCF
HDC/A



私钥解密

"Hey Bob,
how about
lunch at
Taco Bell. I
hear they
have free
refills!"

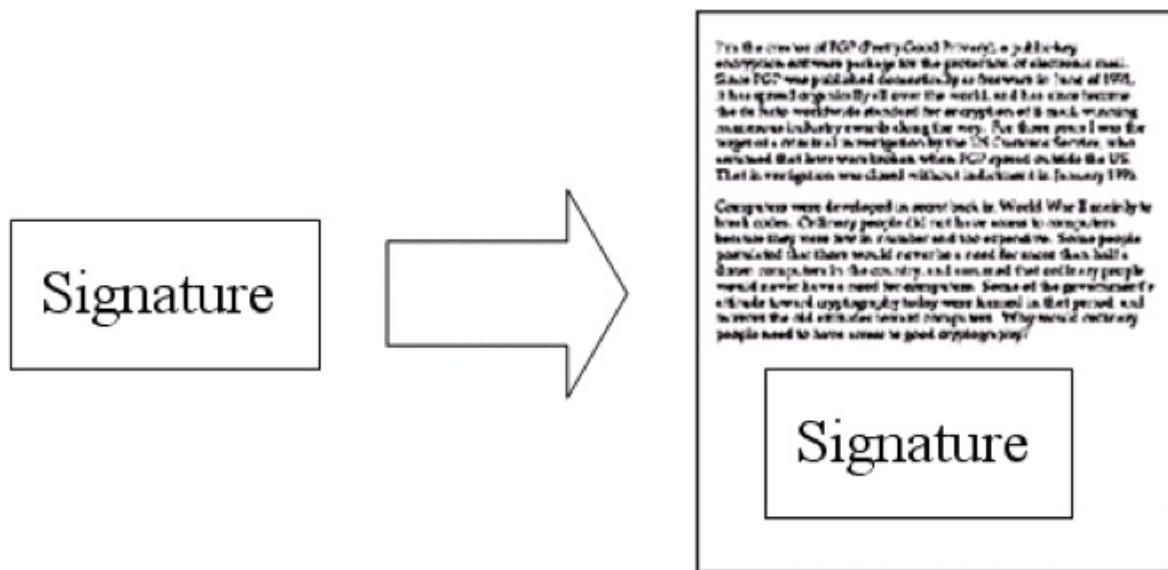
(5) 鲍勃给苏珊回信，决定采用"数字签名"。他写完后先用Hash函数，生成信件的摘要（digest）。



(6) 然后，鲍勃使用私钥，对这个摘要加密，生成"数字签名"（signature）。



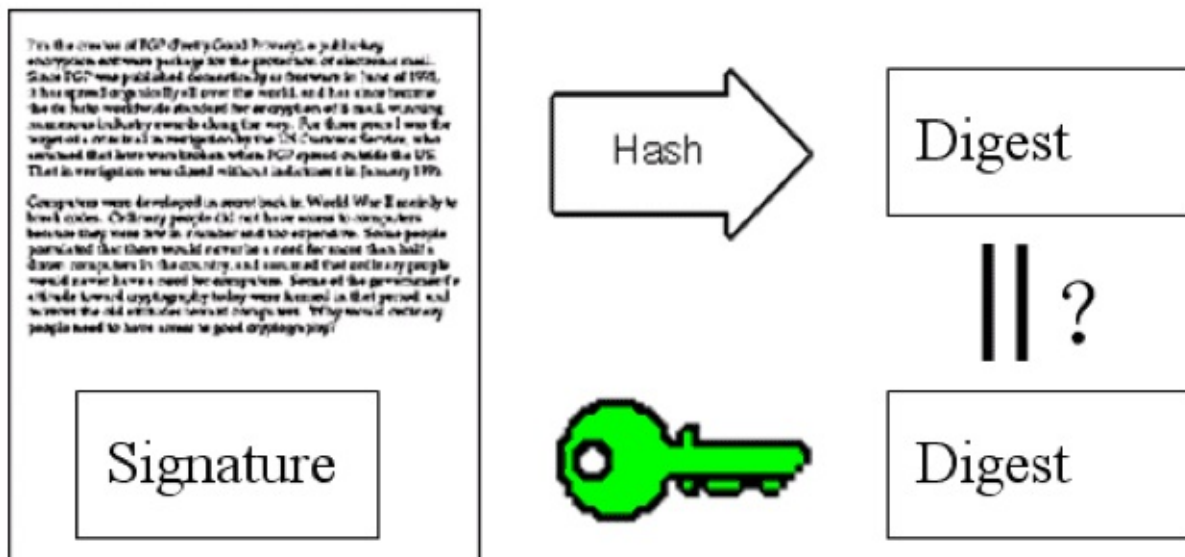
(7) 鲍勃将这个签名，附在信件下面，一起发给苏珊。



(8) 苏珊收信后，取下数字签名，用鲍勃的公钥解密，得到信件的摘要。由此证明，这封信确实是鲍勃发出的。



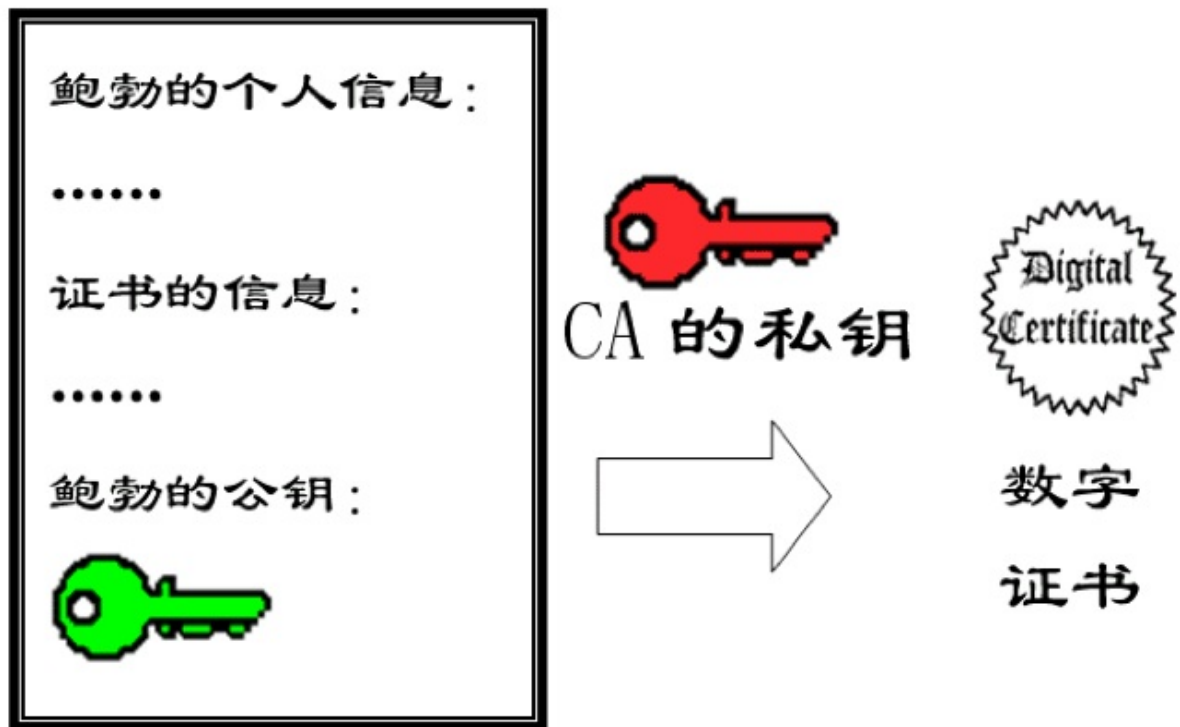
(9) 苏珊再对信件本身使用Hash函数，将得到的结果，与上一步得到的摘要进行对比。如果两者一致，就证明这封信未被修改过。



(10) 复杂的情况出现了。道格想欺骗苏珊，他偷偷使用了苏珊的电脑，用自己的公钥换走了鲍勃的公钥。此时，苏珊实际拥有的是道格的公钥，但是还以为这是鲍勃的公钥。因此，道格就可以冒充鲍勃，用自己的私钥做成"数字签名"，写信给苏珊，让苏珊用假的鲍勃公钥进行解密。



(11) 后来，苏珊感觉不对劲，发现自己无法确定公钥是否真的属于鲍勃。她想到了一个办法，要求鲍勃去找"证书中心"（certificate authority，简称CA），为公钥做认证。证书中心用自己的私钥，对鲍勃的公钥和一些相关信息一起加密，生成"数字证书"（Digital Certificate）。



(12) 鲍勃拿到数字证书以后，就可以放心了。以后再给苏珊写信，只要在签名的同时，再附上数字证书就行了。



(13) 苏珊收信后，用CA的公钥解开数字证书，就可以拿到鲍勃真实的公钥了，然后就能证明"数字签名"是否真的是鲍勃签的。

