比特币白皮书解读（扩展版）

本文是对比特币白皮书进一步深入的解读，其实是否读懂比特币白皮书并不会影响您炒币，只会影响你是否能够真正的拿住比特币。

——xfli5

来源：网络

编辑：xfli5

推特：@xfli5

目录

1 货币系统发展

货币系统发展：

（1）以物易物

（2）实物货币：如黄金作为实物货币

（3）符号货币：如纸币

（4）中央系统虚拟货币：

（5）分布式虚拟货币：bitcoin 登场

2 私钥、公钥、公钥 hash 及钱包地址

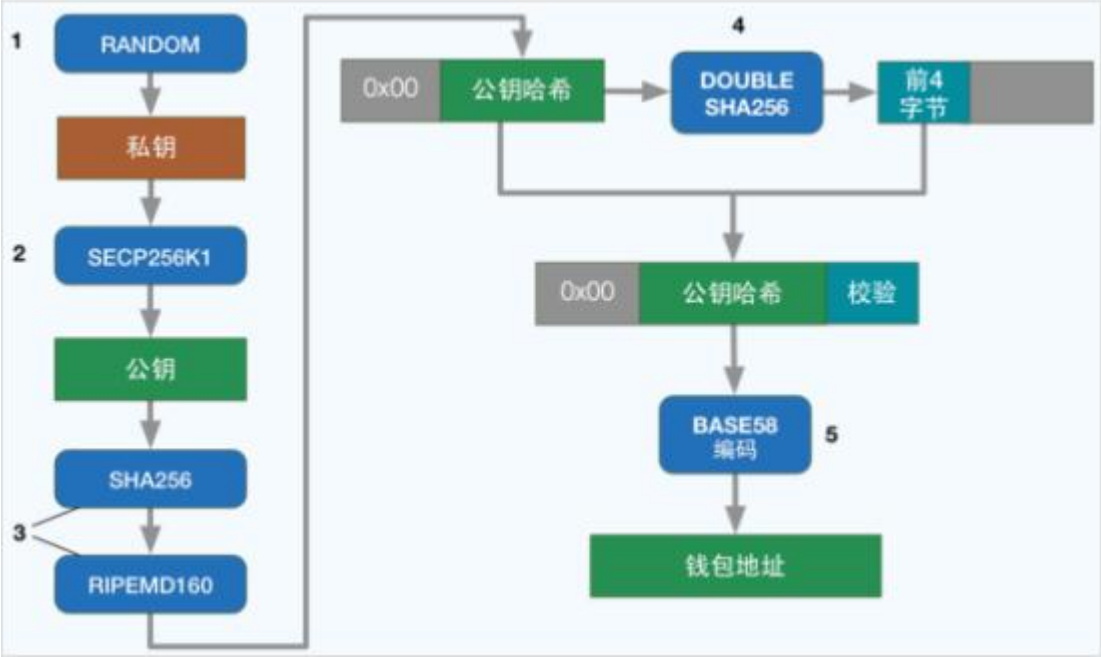
2.1 概述

私钥（Private Key)、公钥（Public Key)、公钥 Hash (Public Key Hash)、钱包地址(Bitcoin Address)。

比特币系统使用了**椭圆曲线签名算法**，算法的**私钥**由 32 个字节随机数组成，**私钥通过 SECP256K1 算出公钥**，**公钥**经过 SHA256 和 RIPEMD160 得到公钥哈希，公钥哈希再经过一系列处理得到比特币地址（也就是常说的**钱包地址**）。

通常情况下，我们都直接将钱包地址说成公钥，但从上述描述中，我们知道这是不恰当的，实际上钱包地址其实是公钥经过一系列哈希算法和编码算法后得到的。

2.2 钱包地址生成过程



（1）首先使用随机数生成器（RANDOM）生成一个『私钥』。一般来说这是一个 256bits 的数，拥有了这串数字就可以对相应『钱包地址』中的比特币进行操作，所以必须被安全地保存起来。

（2）『私钥』经过 SECP256K1 算法处理生成了『公钥』。SECP256K1 是一种椭圆曲线算法，通过一个已知『私钥』时可以算得『公钥』，而『公钥』已知时却无法反向计算出『私钥』。这是保障比特币安全的算法基础。

（3）Public Key Hash，由 Public Key，经过 Hash160 的计算得到。所谓 Hash160，是指先进行 1 次 SHA256 运算，再进行 1 次 RIPEMD160 运算，最后算出来的 Public Key Hash 是 160 bits。

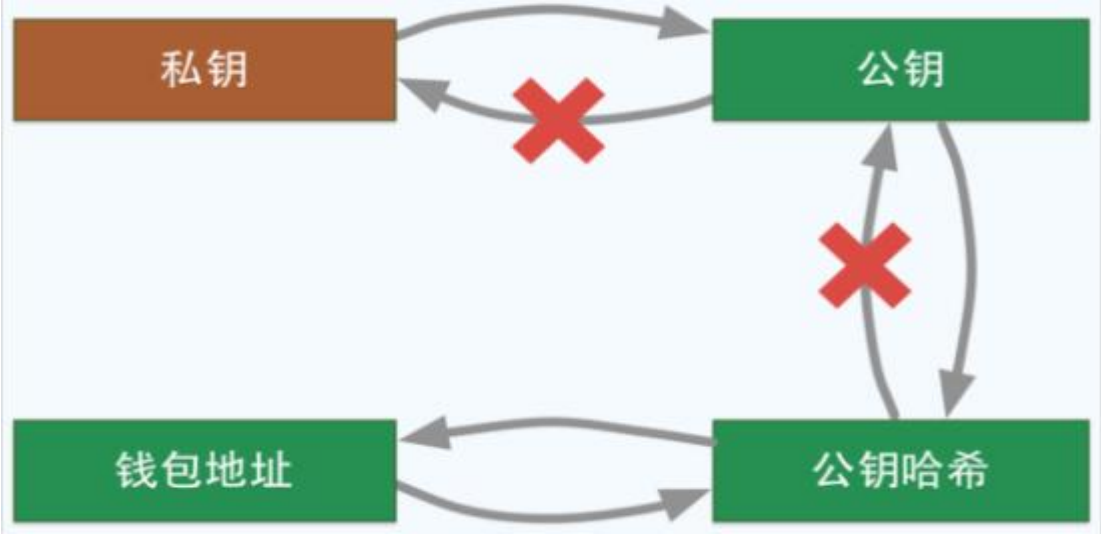
（4）将一个字节的地址版本号连接到『公钥哈希』头部（对于比特币网络的 pubkey 地址，这一字节为“0”），然后对其进行两次 SHA256 运算，将结果的前 4 字节作为『公钥哈希』的校验值，连接在其尾部。

（5）将上一步结果使用 BASE58 进行编码(比特币定制版本)，就得到了『钱包地址』。比如, 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa

2.3 私钥、公钥、公钥哈希及钱包地址关系

在上述的五个步骤里只有“BASE58 编码”有相应的可逆算法（“BASE58 解码”），其他算法都是不可逆的，所以这些数据之间的关系可以表示为：

可以看到：



通过『私钥』可以得到上述计算过程中所有的值。

『公钥哈希』和『钱包地址』可以通过互逆运算进行转换，所以它们是等价的。

那为什么不直接拿 Public Key 当 Bitcoin Address，而要再加 1 层 Public Key Hash 呢？这就涉及到 1 个抽象的问题。在比特币网络中，Bitcoin Address 是一个抽象的概念，在这个场景中，背后代表的是 Public Key；在其他场景中，可能是一个 Script Hash，而不是 Public Key Hash。

2.4 比特币钱包分类

本处以比特币钱包为例，其他数字货币钱包分类类似。

我们按照**私钥存储方式**进行分类，可以分为**中心化钱包与去中心化钱包**两个大类。去中心化钱包也称之为链上钱包（onchain ），中心化钱包也称之为链下钱包（offchain）。

类别	去中心化钱包				中心化钱包
子类	全节点钱包	轻钱包	纸钱包	脑钱包	交易所等
私钥存储位置	用户端	用户端	用户端	用户端	用户不掌握私钥
资产存储位置	区块链中	区块链中	区块链中	区块链中	提供中心化钱包的机构手中

2.4.1 去中心化钱包

私钥由用户维护，资产存储在区块链中。去中心化钱包通常被称为 Onchain 钱包。私钥被移交给用户。如果私钥丢失，钱包将无法帮助用户恢复，资金将永远丢失。但是去中心化的钱包很难被黑客集中攻击，用户不必担心钱包服务提供

商的自我窃取。

（1）全节点钱包

全节点钱包，也称为完整钱包，**是指存储比特币交易所有历史记录的客户端（曾经的每个用户的每一笔交易），私钥由用户自己保管。**全节点钱包安装完成后，一般安装在我们的电脑端（window、linux），会从网上下载比特币网络所有的节点数据，从创世区块至今所有的数据，然后这个钱包自己来维护全网的数据，自己验证竞争挖矿的结果，是完全的去中心化的钱包。

全节点钱包，可直接在比特币网络发起交易，可处理比特币协议的所有方面，能独立验证整个区块链和任何交易。全节点钱包一般要常开着，来及时同步网络中的新数据，否则每次使用时候要先同步数据。

全节点钱包 Bitcoin Core 就是典型的全节点钱包，由比特币官方开发 Core 团队开发的。

全节点钱包的优点：私钥用户掌握，具有更好的安全性与隐私性。

全节点钱包的缺点：占用很大的磁盘空间保存区块数据；每次使用都需要同步到最新的区块；不支持多种数字资产。

（2）轻钱包

轻钱包，也称为简单支付验证（SPV：simple-payment-verification）钱包，私钥由用户保管，不保存所有区块的数据，只保存跟自己相关的数据。通过连接完整客户端获得比特币的交易信息，可独立创建、验证和传输交易。轻钱包直接与比特币网络交互，不需要中介。轻钱包可以运行在电脑，手机，网页等地方。比如手机端轻钱包有 imtoken、trust 及火星钱包等

轻钱包的优点：用户体验好；很多轻钱包都支持多种数字资产；占用空间少。

轻钱包的缺点：验证速度会慢一些。

（3）纸钱包与脑钱包

纸钱包和脑钱包并不是完整意义上的比特币钱包，因为使用纸钱包或脑钱包，你无法使用上面的比特币，发币时还得借助其他钱包导入私钥。纸钱包和脑钱包仅仅是保存比特币私钥的两种方式。

纸钱包，就是把比特币的私钥打印在纸上，然后保存起来，一般用于冷藏比特币，也可以作为本地钱包或在线钱包的私钥备份。如果生成、打印私钥的环境是安全的，那么纸钱包安全性是比较高的，成本也很低。但是要注意防火防水防盗，如果是热敏纸打印的，还要注意热敏纸字迹容易褪掉。

此外，私钥也可以备份在磁介质中。像 U 盘、SD 卡的寿命都很长，质量好的保存几十年没问题，移动硬盘当然也可以，就是有些浪费。光盘最便宜，但是容易氧化，质量普遍不高，可能几年就坏掉了。

我们知道，比特币是私钥就是一串数字和大小写字母的组合，50 来个字符，如果我记忆力强，硬硬地把它记住，也算是一种保存方式。不过这些字母数字毫无规律，要想一字不差的长时间记忆，一般人恐怕做不到。那我们可以找一种变

通的方式，想一句你能记住的密语，这句话是只有你知道，而且很容易记忆的，比如你向女朋友表白时说的一句话，你觉得肯定忘不了而且一字不差，也没其他人知道，那可以用这句话做密语，通过数学计算转换为一组比特币的私钥和地址，就可以将比特币保存到你的脑子里了。如果你经常表白，那就算了吧。

因为人的想法总是有迹可循的，想一句任何人都猜不出的话并牢牢记住，也不那么容易。所以脑钱包并不那么安全，有人曾用生日做脑钱包进行测试，结果币刚转进去就被黑客转走了，如果忘记密语或者干脆失忆了，比特币就永远找不回来了，变成了“脑残包”。

2.4.2 中心化钱包

中心化钱包是指私钥不在用户手中，全完全依赖于该中心化机构。中心化钱包通常被称为链下钱包（Offchain）。例如交易所的账号，你把币存到交易所，交易所给你记上帐，当你给其他用户发送，收币发币是在交易所的账簿上加加减减，并没有真正写入比特币区块链。只有用户存币和提币的时候，才会发生链上交易。

对于中心化的钱包，用户是没有自己钱包的私钥的。很多比特粉受去中心化思想影响比较深，认为什么都要去中心化，对中心化交易以及挖矿算力集中等比较排斥。实际上，链下钱包效率很高，可以实时到账，方便对接购物平台，国外很多商家就是对接 coinbase 的，国内的币付宝也有。中心化钱包将私钥保管的责任交给钱包公司，用户不必过多地考虑私钥安全。

需要说明的是，中心化钱包里的币，并不是比特币，而是信用币。钱包里的数字，并不表示你拥有这么多币，只是说明别人欠你这么多的比特币而已。

3 区块（block）

3.1 比特币区块整体架构

一个完整的区块结构主要由以下几部分构成：

数据项	字段	字节	描述
Magic NO	魔数	4	常数 0xD9B4BEF9
Blocksize	区块大小	4	用字节表示的该字段之后的区块大小
Blockheader	区块头	80	区块头包含六个字段
Transactons counter	交易计数器	1-9	该区块包含的交易数量，包含 coinbase 交易
Transactions	交易	不定	记录在区块里的交易信息，使用原生的交易信息格式，并且交易在数据流中的位置必须与 Merkle 树的叶子节点顺序一致

注：比特币的区块大小目前被严格限制在 1MB 以内。4 字节的区块大小字段不包含在此内。

3.2 比特币区块头结构

每个区块中包含 80byte 的区块头，区块头包含六个元素。区块头主要组成部分：

字节数	字段	说明
4	Version	区块版本号，表示本区块遵守的验证规则
32	hashPrevBlock	前一区块的哈希值=SHA256(SHA256(父区块头))
32	hashMerkleRoot	上一个 block 产生之后至新 block 生成此时间内所有交易的 merkle root 的 hash 值
4	Time	时间戳。自 1970 年 1 月 1 日 0 时 0 分 0 秒以来的秒数。每秒自增一，标记 Block 的生成时间，同时为 block hash 探寻引入一个频繁的变动因子
4	Bits	压缩格式的当前目标值（current target in compact format）。可以推算出难度值（difficulty），用于验证 block hash 难度是否达标
4	Nonce	32bit 的随机数（从 0 开始）。在上面数个字段都固定的情况下，不停地更换该随机数来探寻符合要求的 hash 值

bits 值是压缩存储的当前目标值，仅为 4bytes。通过 bits 得到 target 的计算公式是 **target = coefficient * 256^(exponent – 3)。目标值（target）是一个 256bit 数（也就是 256 位的二进制数）**。以 bits 等于 0x1903a30c 为例（0x 代指 16 进制意义），coefficient 是后三个字节：0x03a30c，exponent 是第一个字节：0x19。

Bits 的具体压缩过程：

（1）将数字转换为 256bit 数（或者说长度为 64（每一个十六进制相当于 4bit 的二进制）的十六进制）

（2）如果第一位数字大于 127(0x7f),则前面添加一个字节长度的零：00

（3）压缩结果中的第一位存放该十六进制数的位数

（4）后面三个数存放该十六进制数的前三位，如果不足三位，则后面补零

例 1:将数字 1000 压缩。

（1）先转换为长度为 64 的 16 进制数（256bit 二进制数）

1000 = 3*16^2+e*16+8*1

那么是由两个数字构成：

03　　e8

（2）第一个数未超过 0x7f,则不需填 0；

（3）从（1）16 进制的表示中可以看出位数（按字节计算位数）为 2，即 0x02；

（4）但长度两位低于三位，在后面补零，最终表示为：0x0203e800 。

例 2：创世区块中的 bits 为 0x1d00ffff，对应的数值为 2^(256-32)-1。

（1）将 2^(256-32)-1 转换为 16 进制（256bit 数）

ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff　　--总共 56 个十六进制的 f，每个 f 对应 1111，相当于 28byte 长度

（2）第一位已经超过 0x7f,前面添加一个字节长度的零：

00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

（3）现在长度为 28+1=29 (0x1d),则最终压缩结果为：0x1d00ffff。此时出现精度缺失，后面的 26 个 ff 被丢弃了，因为总共才 4 字节，前两字节已经被长度和添加的 0 所占用，只剩下 2 个字节来存储数字。如果我们将压缩结果 0x1d00ffff 解压，会是原值吗？实际结果为：

0x00ffff *256^(0x1d - 3)

=ffff0000000000000000000000000000000000000000000000000000 –共 56 个

而此数为比特币的最大 Target 值，此时难度最小为 1。将其结果前面添加 4 个字节长度的 0，其结果为：

0x00000000FFFF0000000000000000000000000000000000000000000000000000　--共 64 个

此最小难度值 1，在矿机上一般使用保留尾部的 FF，则为：

0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

称上面数字为 pool difficulty 1（矿池难度 1），矿池难度简称 pdiff。而比特币客户端表示如此精度，也许是困难的，所以不保留尾部的 FF,则结果为：

0x00000000FFFF0000000000000000000000000000000000000000000000000000

此值，在客户端上称之为 bdiff。

3.3 区块难度（difficulty）

通过上面的分析，在区块中并没有存放难度（difficulty）这个指标。

区块难度，是用来衡量挖出一个区块平均所需要的**运算次数**，反映了在一定难度下用多长时间才能挖到一定数量的区块，也是矿工挖矿时重要的参考指标。数据区块产生的难易程度是由**难度值（difficulty）**来衡量的。

在创世区块中，bits 值为 0x1d00ffff，根据公式计算可得 target 为：0x00000000ffff0000000000000000000000000000000000000000000000000000，其中前 32bit 为 0。我们以创世区块的 target 为参照物，同时定义创世区块的 difficult 为 1，其余的区块的 difficult 计算公式为：**当前区块_difficult = target_创世区块/target_当前区块**。由此可见，当前区块的 target 值越小，区块的 difficult 越大，也就是难度越高。

根据 difficult 的计算公式进一步推算（近似）：difficult = 0xffff*2^208/(2^256/2^x),其中 x 为当前区块 target 前导 0 的个数。于是可得 2^x = difficult*(2^32)。所以**为了打包一个区块，我们需计算哈希值的次数为：difficult*(2^32)**。

注：我们解释一下为什么上述的 2^x 可以表示打包一个区块所需要计算的哈希次数？其实这种说法是从概率角度来讲的，我们知道基于 SHA256 算法，目标值最多有 2^256 种结果，也就是说如果我们前端输入不同数值数量为 2^256+1，则其哈希值必定至少存在一个碰撞。假设 x 为当前区块 target 前导 0 的个数，则满足条件的目标值（哈希值）的个数为 2^(256-x)，总的哈希值为 2^256 种结果，那么满足条件的目标值（哈希值）占比总的哈希值=2^(256-x)/ 2^256=2^(-x),因此从概率上来讲，也就意味着我们前端输入 2^x 个数值就可以找到一个目标值。

这里简单的举个例子：现在有一道数学题，已知 X 是 0-99 中任意一个数字，求 X<100。答案非常简单，该范围内所有的数字都符合要求。再求 X<50，那么现在该范围内只有一半的数字符合要求，换句话说，现在的求解难度比之前大了。之前要想求解只需要尝试一次就可以了，现在求解则需要尝试两次，从而我们可得出 x<50 的难度是 x<100 的 2/1=2 倍。同样的，如果让我们求解 X<10，难度就是 x<100 的 100/10=10 倍。这里的参数 100、50、10 就相当于是难度值（difficulty），难度值 difficulty 的计算公式为：

难度值 = 最大目标值 / 当前目标值

目标值是一个 256bit 数。最大目标值是一个固定值 0x1d00ffff，也就是创世区块的 bits。难度值与当前目标值成反比，难度越大，当前目标值（bit）越小，难度越小，当前目标值越大。

目标值是当前区块生成所达成目标值的 hash 值，用于矿工的工作量证明。矿工挖掘的区块的头部 hash 值必须小于目标值，难度就符合要求，数据区块才能被挖掘成功。

当前目标值就是对应区块中的 bits，可以换算成正常格式。

最大目标值是一个定值，前 32 位都是 0，后面 224 位全是 1，十六进制表示如下：0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

我们也可以将难度值与时间进行换算，这样可以更直观的了解数据区块产生的难度，计算公式如下：

为了打包一个区块，我们需计算哈希值的次数为：difficult*(2^32)

一个 block 产生的时间=难度值(difficult)x2^32/hashrate

注：hashrate 是每秒运算的 hash 数量

数据区块难度是没有最大值的，每产生 2016 个区块后，数据区块运算难度会调整一次。以比特币为例，平均每隔 10 分钟会产生一个区块，那么每 14 天（2016*10/60/24=14）会调整一次区块难度，在未到调整周期时，区块难度是保持不变的。难度的调整是在每个完整节点中独立自动发生的。**每 2016 个区块，所有节点都会按统一的公式自动调整难度**，这个公式是由**最新 2016 个区块的花费时长与期望时长**（期望时长为 20160 分钟即两周，是按每 10 分钟一个区块的产生速率计算出的总时长）比较得出的，根据实际时长与期望时长的比值，进行相应调整（或变难或变易）。也就是说，如果区块产生的速率比 10 分钟快则增加难度，比 10 分钟慢则降低难度。这个公式可以总结为如下形式：

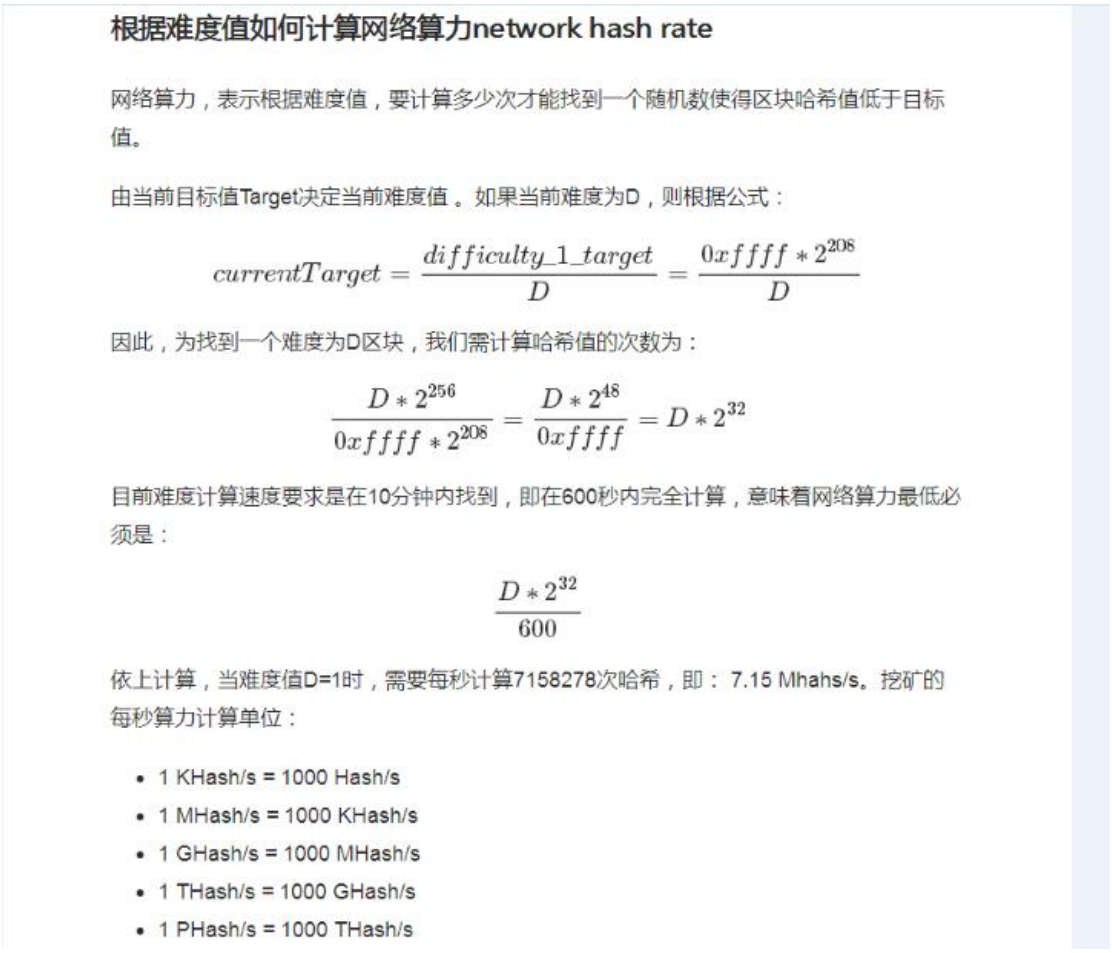
新难度值 = 旧难度值 *(过去 2016 个区块花费时长 / 20160 分钟)

影响区块难度的因素有很多，这里主要介绍一下难度与算力的关系。区块难

度的计算是与算力成正比的，当全网的算力越高时，区块难度就越高，反之当未来算力降低时，区块难度也会随之降低。你可以想象，如果算力突然暴涨，是原来的一倍，那么原本需要 14 天才能挖完 2016 块区块，现在只需要 7 天就能完成，到调整周期时难度增加一倍；如果发生意外，算力损失一半，那么原来 14 天的工作需要 28 天才能完成，到调整周期时，难度降低为原来的一半就可以了。因此，我们可以得出结论，区块难度的调整，可以通过算力情况进行自动匹配，当算力增加时，反应调整会加快；当算力降低时，反应调整会缓慢。

3.4 根据难度值计算网络算力

注：下图中的 difficulty_1_target=最大目标值=创世区块的 bit=0x1d00ffff
2*256 是最大 hash 计算数，除以当前的目标值,就可以得到在难度为 D 的情形下，计算多少次可以打包一个区块。



根据难度值如何计算网络算力network hash rate

网络算力，表示根据难度值，要计算多少次才能找到一个随机数使得区块哈希值低于目标值。

由当前目标值Target决定当前难度值 。如果当前难度为D，则根据公式：

$$currentTarget = \frac{difficulty\_1\_target}{D} = \frac{0xffff * 2^{208}}{D}$$

因此，为找到一个难度为D区块，我们需计算哈希值的次数为：

$$\frac{D * 2^{256}}{0xffff * 2^{208}} = \frac{D * 2^{48}}{0xffff} = D * 2^{32}$$

目前难度计算速度要求是在10分钟内找到，即在600秒内完全计算，意味着网络算力最低必须是：

$$\frac{D * 2^{32}}{600}$$

依上计算，当难度值D=1时，需要每秒计算7158278次哈希，即： 7.15 Mhahs/s。挖矿的每秒算力计算单位：

- 1 KHash/s = 1000 Hash/s
- 1 MHash/s = 1000 KHash/s
- 1 GHash/s = 1000 MHash/s
- 1 THash/s = 1000 GHash/s
- 1 PHash/s = 1000 THash/s

4 交易（Transaction）

4.1 比特币交易构成

（1）比特币交易简介及分类

交易(Transaction)是比特币系统的信息载体，最小单元。而块(Block)就是将这些基础单元打包装箱，贴上封条，并串联起来。巨大算力保障了块的安全，也就保障了单个交易的安全。交易（transaction）有三种常见类型：生成交易(Generation TX)，合成地址交易(Script Hash TX)，通用地址交易(Pub key Hash TX)。该分类并非严格意义的，只是根据交易的输入输出做的简单区分。

1）Generation TX

每个 Block 都对应一个生成交易(Generation TX)，该类交易是没有输入交易的，挖出的新币是所有币的源头。

2）Script Hash TX

该类交易目前不是很常见，大部分人可能没有听说过，但是非常有意义。未来应该会在某些场合频繁使用。该类交易的接受地址不是通常意义的地址，而是一个合成地址，以 3 开头（对，以 3 开头的也是比特币地址！）。三对公私钥，可以生成一个合成地址。在生成过程时指定 n of 3 中的 n，n 范围是[1, 3]，若 n=1，则仅需一个私钥签名即可花费该地址的币，若 n=3，则需要三把私钥依次签名才可以。

3）Pubkey Hash TX

该类是最常见的交易类型，由 N 个输入、M 个输出构成。

（2）基本交易类型

有了比特币地址，其实也就是 Public Key Hash，就有了一个最常见的交易类型，叫做 P2PKH(Pay to Public Key Hash)，付款给对方的 Public Key Hash。

另外一个交易类型，就是 P2SH（Pay to Scirpt Hash)，后面会专门讲述。

（3）交易（transaction）数据结构

交易中存放的是货币所有权的流转信息，所有权登记在比特币地址上(Public Key)。这些信息是全网公开的，以明文形式存储（比特币系统里的所有数据都是明文的），只有当需要转移货币所有权时，才需要用私钥签名来验证。交易（transaction）的数据结构如下：

lock_time 是一个多意字段，表示在某个高度的 Block 之前或某个时间点之前

字段大小	描述	数据类型	解释
4	version, 版本	uint32_t	交易数据结构的版本号
1+	tx_in count, 输入数量	var_int	输入交易的数量
41+	tx_in	tx_in[]	输入交易的数组，每个输入>=41字节
1+	tx_out count, 输出数量	var_int	输出地址的数量
9+	tx_out	tx_out[]	输入地址的数组，每个输入>=9字节
4	lock_time, 锁定时间	uint32_t	见下方解释

该交易处于锁定态，无法收录进 Block。关于 lock_time 取值及含义如下：

值	含义
0	立即生效，代表这笔交易不会积压，节点收到这笔交易之后，立即会进入 Memory Pool，之后开始打包
< 500000000（5 亿）	含义为 Block 高度，处于该 Block 之前为锁定（不生效）
>= 500000000（5 亿）	含义为 Unix 时间戳，处于该时刻之前为锁定（不生效）

若该笔交易的所有输入交易的 sequence 字段，均为 INT32 最大值(0xffffffff)，则忽略 lock_time 字段。否则，该交易在未达到 Block 高度或达到某个时刻之前，是不会被收录进 Block 中的。

（4）通过具体区块进一步分析

为了演示方便，我们读取稍早期的块数据，以高度 116219 Block 为例（具体参考 http://v1.8btc.com/wiki/bitcoin-technical-principles）。

可以通过 https://www.blockchain.com/explorer 网站查询区块及交易等详情

该 Block 里面有 5 笔交易，第一笔为 Generation TX

```
# ~ bitcoind getblock 0000000000007c639f2cbb23e4606a1d022fa4206353b9d92e99f5144bd74611
{
"hash" : "0000000000007c639f2cbb23e4606a1d022fa4206353b9d92e99f5144bd74611",
"confirmations" : 144667,
"size" : 1536,
"height" : 116219,
"version" : 1,
"merkleroot" : "587fefd748f899f84d0fa1d8a3876fdb406a4bb8f54a31445cb72564701daea6",
"tx" : [
"be8f08d7f519eb863a68cf292ca51dbab7c9b49f50a96d13f2db32e432db363e",
"a387039eca66297ba51ef2da3dcc8a0fc745bcb511e20ed9505cc6762be037bb",
"2bd83162e264abf59f9124ca517050065f8c8eed2a21fbf85d454ee4e0e4c267",
"028cfae228f8a4b0caee9c566bd41aed36bcd237cdc0eb18f0331d1e87111743",
"3a06b6615756dc3363a8567fbfa8fe978ee0ba06eb33fd844886a0f01149ad62"
],
"time" : 1301705313,
"nonce" : 1826107553,
"bits" : "1b00f339",
"difficulty" : 68977.78463021,
"previousblockhash" : "00000000000010d549135eb39bd3bbb1047df8e1512357216e8a85c57a1efbfb",
"nextblockhash" : "000000000000e9fcc59a6850f64a94476a30f5fe35d6d8c4b4ce0b1b04103a77"
}
```

（be8f08d7f519eb863a68cf292ca51dbab7c9b49f50a96d13f2db32e432db363e），解析出来看一下具体内容：

Generation TX 的输入不是一个交易，而带有 coinbase 字段的结构。该字段


```
# ~ bitcoind getrawtransaction be8f08d7f519eb863a68cf292ca51dbab7c9b49f50a96d13f2db32e432db363e 1
{
"hex" :
"01000000010000000000000000000000000000000000000000000000000000000000000000ffffffff070439f3001b0134ffffffff014034152a0100000043410 45b3aaa284d169c5ae2d20d0b0673468ed3506aa8fea
5976eacaf1ff304456f6522fbce1a646a24005b8b8e771a671f564ca6c03e484a1c394bf96e2a4ad01dceac00000000",
"txid" : "be8f08d7f519eb863a68cf292ca51dbab7c9b49f50a96d13f2db32e432db363e",
"version" : 1,
"locktime" : 0,
"vin" : [
{
"coinbase" : "0439f3001b0134",
"sequence" : 4294967295
}
],
"vout" : [
{
"value" : 50.01000000,
"n" : 0,
"scriptPubKey" : {
"asm" : "045b3aaa284d169c5ae2d20d0b0673468ed3506aa8fea5976eacaf1ff304456f6522fbce1a646a24005b8b8e771a671f564ca6c03e484a1c394bf96e2a4ad01dce OP_CHECKSIG",
"hex" : "41045b3aaa284d169c5ae2d20d0b0673468ed3506aa8fea5976eacaf1ff304456f6522fbce1a646a24005b8b8e771a671f564ca6c03e484a1c394bf96e2a4ad01dceac",
"reqSigs" : 1,
"type" : "pubkey",
"addresses" : [
"1LgZTvoTJ6quJNCURmBUaJJkWWQZXkQnDn"
]
}
}
],
"blockhash" : "00000000000007c639f2cbb23e4606a1d022fa4206353b9d92e99f5144bd74611",
"confirmations" : 145029,
"time" : 1301705313,
"blocktime" : 1301705313
}
```


的值由挖出此 Block 的人填写，这是一种“特权”：可以把信息写入货币系统（大家很喜欢用系统中的数据结构字段名来命名站点，例如 blockchain、coinbase 等，这些词的各种后缀域名都被抢注一空）。中本聪在比特币的第一个区块的 generation TX 中的写入的 coinbase 值是：

```
1  "coinbase":"04ffff001d0104455468652054696d65732030332f4a616e2f32303039204368616e63656c6c6f72206f6e206272696e6b206f66207365636f6e642062616 96c6f757420666f722062616e6b73"
```

将该段16进制转换为ASCII字符，就是那段著名的创世块留言：

```
1  The Times 03/Jan/2009 Chancellor on brink of second bailout for banks
```

接下来我们再来看上述第四个交易（028cfae228f8a4b0caee9c566bd41aed36bcd237cdc0eb18f0331d1e87111743），包含三个输入和两个输出的普通交易：


```
# ~ bitcoind getrawtransaction 028cfae228f8a4b0caee9c566bd41aed36bcd237cdc0eb18f0331d1e87111743 1
{
"hex" :
"0100000003c9f3b07ebfca68fd1a6339d0808fbb013c90c6095fc93901ea77410103489ab7000000008a473044022055bac1856ecbc377dd5e869b1a84ed1d5228c987b098c095030c12431a4d524902205552313
0a9d0af5fc27828aba43b464ecb1991172ba2a509b5fbd6cac97ff3af0141048aefd78bba80e2d1686225b755dacea890c9ca1be10ec98173d7d5f2fefbbf881a6e918f3b051f8aaaa3fcc18bbf65097ce8d30d5a7e5ef8d
1005eaafd4b3fbeffffffffc9f3b07ebfca68fd1a6339d0808fbb013c90c6095fc93901ea77410103489ab7010000008a47304402206b993231adec55e6085e75f7dc5ca6c19e42e744cd60abaff957b1c352b3ef9a022022
a22fec37dfa2c646c78d9a0753d56cb4393e8d0b22dc580ef1aa6cccef208d0141042ff65bd6b3ef04253225405ccc3ab2dd926ff2ee48aac210819698440f35d785ec3cec92a51330eb0c76cf49e9e474fb9159ab41653
a9c1725c031449d31026afffffffffc98620a6c40fc7b3a506ad79af339541762facd1dd80ff0881d773fb72b230da010000008b483045022040a5d957e087ed61e80f1110bcaf4901b5317c257711a6cbc54d6b98b6a856
3f02210081e3697031fe82774b8f44dd3660901e61ac5a99bff2d0efc83ad261da5b4f1d014104a7d1a57e650613d3414ebd59e3192229dc09d3613e547bdd1f83435cc4ca0a11c679d96456cae75b1f5563728ec7da
1c1f42606db15bf554dbe8a829f3a8fe2fffffffff0200bd0105000000001976a914634228c26cf40a02a05db93f2f98b768a8e0e61b88acc096c7a6030000001976a9147514080ab2fcac0764de3a77d10cb790c71c74c2
88ac00000000",
"txid" : "028cfae228f8a4b0caee9c566bd41aed36bcd237cdc0eb18f0331d1e87111743",
"version" : 1,
"locktime" : 0,
```

```
"vout" : [
{
"value" : 0.84000000,
"n" : 0,
"scriptPubKey" : {
"asm" : "OP_DUP OP_HASH160 634228c26cf40a02a05db93f2f98b768a8e0e61b OP_EQUALVERIFY OP_CHECKSIG",
"hex" : "76a914634228c26cf40a02a05db93f2f98b768a8e0e61b88ac",
"reqSigs" : 1,
"type" : "pubkeyhash",
"addresses" : [
```

```
"vin" : [
{
"txid" : "b79a4803014177ea0139c95f09c6903c01bb8f80d039631afd68cabf7eb0f3c9",
"vout" : 0,
"scriptSig" : {
"asm" : "3044022055bac1856ecbc377dd5e869b1a84ed1d5228c987b098c095030c12431a4d5249022055523130a9d0af5fc27828aba43b464ecb1991172ba2a509b5fbd6cac97ff3af01
048aefd78bba80e2d1686225b755dacea890c9ca1be10ec98173d7d5f2fefbbf881a6e918f3b051f8aaaa3fcc18bbf65097ce8d30d5a7e5ef8d1005eaafd4b3fbe",
"hex" :
"473044022055bac1856ecbc377dd5e869b1a84ed1d5228c987b098c095030c12431a4d5249022055523130a9d0af5fc27828aba43b464ecb1991172ba2a509b5fbd6cac97ff3af0141048aefd78bba80e2d1686225
b755dacea890c9ca1be10ec98173d7d5f2fefbbf881a6e918f3b051f8aaaa3fcc18bbf65097ce8d30d5a7e5ef8d1005eaafd4b3fbe"
},
"sequence" : 4294967295
},
{
"txid" : "b79a4803014177ea0139c95f09c6903c01bb8f80d039631afd68cabf7eb0f3c9",
"vout" : 1,
"scriptSig" : {
"asm" : "304402206b993231adec55e6085e75f7dc5ca6c19e42e744cd60abaff957b1c352b3ef9a022022a22fec37dfa2c646c78d9a0753d56cb4393e8d0b22dc580ef1aa6cccef208d01
042ff65bd6b3ef04253225405ccc3ab2dd926ff2ee48aac210819698440f35d785ec3cec92a51330eb0c76cf49e9e474fb9159ab41653a9c1725c031449d31026a",
"hex" :
"47304402206b993231adec55e6085e75f7dc5ca6c19e42e744cd60abaff957b1c352b3ef9a022022a22fec37dfa2c646c78d9a0753d56cb4393e8d0b22dc580ef1aa6cccef208d0141042ff65bd6b3ef04253225405ccc
3ab2dd926ff2ee48aac210819698440f35d785ec3cec92a51330eb0c76cf49e9e474fb9159ab41653a9c1725c031449d31026a"
},
"sequence" : 4294967295
},
{
"txid" : "da30b272fb73d78108ff80ddd1ac2f76419533af79ad06a5b3c70fc4a62086c9",
"vout" : 1,
"scriptSig" : {
"asm" : "3045022040a5d957e087ed61e80f1110bcaf4901b5317c257711a6cbc54d6b98b6a8563f02210081e3697031fe82774b8f44dd3660901e61ac5a99bff2d0efc83ad261da5b4f1d01
04a7d1a57e650613d3414ebd59e3192229dc09d3613e547bdd1f83435cc4ca0a11c679d96456cae75b1f5563728ec7da1c1f42606db15bf554dbe8a829f3a8fe2f",
"hex" :
"483045022040a5d957e087ed61e80f1110bcaf4901b5317c257711a6cbc54d6b98b6a8563f02210081e3697031fe82774b8f44dd3660901e61ac5a99bff2d0efc83ad261da5b4f1d014104a7d1a57e650613d3414eb
d59e3192229dc09d3613e547bdd1f83435cc4ca0a11c679d96456cae75b1f5563728ec7da1c1f42606db15bf554dbe8a829f3a8fe2f"
},
"sequence" : 4294967295
}
],
```


解析：

（1）”vin”:[ ]代指本次交易的输入，就是上一次的交易输出，其中的每一个 vout 都是代指上一次交易输出编号。

》scriptSig: 拥有者对该交易的 ECDSA 签名。在 scriptSig 中包含 asm 和 hex 这两个字段。下面我们解释一下这两个字段，上面的 vin[]里面，第 1 个输入，这 2 个字段的值，分别为：

asm：

3044022055bac1856ecbc377dd5e869b1a84ed1d5228c987b098c095030c12431a4d5249022055523130a9d0af5fc27828aba43b464ecb1991172ba2a509b5fbd6cac97ff3af01048aefd78bba80e2d1686225b755dacea890c9ca1be10ec98173d7d5f2fefbbf881a6e918f3b051f8aaaa3fcc18bbf65097ce8d30d5a7e5ef8d1005eaafd4b3fbe

hex：

473044022055bac1856ecbc377dd5e869b1a84ed1d5228c987b098c095030c12431a4d5249022055523130a9d0af5fc27828aba43b464ecb1991172ba2a509b5fbd6cac97ff3af0141048aefd78bba80e2d1686225b755dacea890c9ca1be10ec98173d7d5f2fefbbf881a6e918f3b051f8aaaa3fcc18bbf65097ce8d30d5a7e5ef8d1005eaafd4b3fbe

这 2 个字段的值其实是一样的，只是 hex 在开头插入了字符 47，中间插入了一个字符 41，其他部分完全一样。

asm 为 assembly（拼装，或者汇编的意思），hex 为其 16 进制的表达。因此，这里的 asm 和 hex 就是同一个东西，里面包含了两部分：

signature（付款人的私钥的签名，0x47=71bit）+ pub key（付款人的公钥，0x47=65bit）

signature：

3044022055bac1856ecbc377dd5e869b1a84ed1d5228c987b098c095030c12431a4d5249022055523130a9d0af5fc27828aba43b464ecb1991172ba2a509b5fbd6cac97ff3af01

Pub key：

048aefd78bba80e2d1686225b755dacea890c9ca1be10ec98173d7d5f2fefbbf881a6e918f3b051f8aaaa3fcc18bbf65097ce8d30d5a7e5ef8d1005eaafd4b3fbe

（2）”vout”:[ ]代指本次交易的输出

》scriptPubKey: 接收方的公钥。scriptPubKey 里面有 asm, hex, reqSigs(暂时忽略）, type, address 几个字段。同样，asm 和 hex 当中同一个意思，只是不同的编码格式；type: 也就是前面我们说的 P2PKH，Pay to Public Key hash；address: 也就是对方的收款地址，或者说钱包地址。

第 1 个 vout 的 scriptPubKey 的值如下：

OP_DUP OP_HASH160 634228c26cf40a02a05db93f2f98b768a8e0e61b OP_EQUALVERIFY OP_CHECKSIG

address 值为：

"1A3q9pDtR4h8wpvyb8SVpiNPpT8ZNbHY8h"

（3）开头的字段 hex 记录了所有相关信息，后面显示的是 hex 解析出来的各类字段信息。下面把逐个分解 hex 内容（hex 可以从上面的直接看到）：

Tx Hash，俗称交易 ID，由 hex 得出：Tx Hash = SHA256(SHA256(hex))。由于

```
01000000 // 版本号，UINT32
03 // Tx输入数量，变长INT。3个输入。

/*** 第一组Input Tx ***/
// Tx Hash，固定32字节
c9f3b07ebfca68fd1a6339d0808fbb013c90c6095fc93901ea77410103489ab7
00000000 // 消费的Tx位于前向交易输出的第0个，UINT32，固定4字节
8a // 签名的长度，0x8A = 138字节
// 138字节长度的签名，含有两个部分：公钥+签名
47 // 签名长度，0x47 = 71字节
3044022055bac1856ecbc377dd5e869b1a84ed1d5228c987b098c095030c12431a4d5249022055523130a9d0af5fc27828aba43b464ecb1991172ba2a509b5fbd6cac97ff3af01
41 // 公钥长度，0x41 = 65字节
048aefd78bba80e2d1686225b755dacea890c9ca1be10ec98173d7d5f2fefbbf881a6e918f3b051f8aaaa3fcc18bbf65097ce8d30d5a7e5ef8d1005eaafd4b3fbe
ffffffff // sequence，0xffffffff = 4294967295，UINT32, 固定4字节

02 // Tx输出数量，变长INT。两个输出。

/*** 第一组输出 ***/
00bd010500000000 // 输出的币值，UINT64，8个字节。字节序需翻转，~= 0x000000000501bd00 = 84000000 satoshi
19 // 输出目的地址字节数，0x19 = 25字节，由一些操作码与数值构成
// 目标地址
// 0x76 -> OP_DUP(stack ops)
// 0xa9 -> OP_HASH160(crypto)
// 0x14 -> 长度，0x14 = 20字节
76 a9 14
// 地址的HASH160值，20字节
634228c26cf40a02a05db93f2f98b768a8e0e61b
// 0x88 -> OP_EQUALVERIFY(bit logic)
// 0xac -> OP_CHECKSIG(crypto)
88 ac

/*** 第二组输出 ***/
c096c7a603000000
19
76 a9 14 7514080ab2fcac0764de3a77d10cb790c71c74c2 88 ac

00000000 // lock_time，UINT32，固定4字节
```

每个交易只能成为下一个的输入，有且仅有一次，那么不存在输入完全相同的交易，那么就不存在相同的 Tx Hash（SHA256 碰撞概率极小，所以无需考虑 Hash 碰撞的问题，就像无需考虑地址私钥被别人撞到一样）。

任何时候，A 给 B 转账的一笔交易，都会先用 A 的私钥签名，再发给 B 的公钥（这里是 Public Key Hash，而不是直接是 Public Key）。以此，达到 2 个目的：（1）证明了这笔钱的确是来自 A　（2）这笔钱，只有 B 能解锁，能使用。

A 的私钥签名，叫做 scriptSig；发给 B 的公钥，叫做 scriptPubKey。

（5）locktime 与 sequence number

Locktime 在（3）交易（transaction）数据结构已经介绍，不做过多介绍！

sequence number：

LockTime 是 Transaction 级别的，每个 Transaction 有 1 个 LockTime 字段；Sequence Number 呢，是 Input 级别的，1 个 Transaction 里面多个 Input，每个都对应 1 个 Sequence Number，所以它的粒度比 LockTime 要细。

Sequence Number 是个整数，什么意思呢？

原理和 LockTime 类似，也是把交易 Hold 在那，等到该 Input 所引用的交易（也就是上 1 个交易的 UTXO）所在的 Block，其后面跟随了 sequence number 个 Block 之后，该交易才能被打包，被广播进区块链网络。说通俗点，就是上 1 个 Block 要足够成熟，后面跟了很多个块之后，该 Block 里面的 UTXO 才能被花出去。

从上面的解释可以看出，LockTime 和 sequence number，都是一个 Time Lock。要把当前的交易 Hold 在那，等到时间成熟了再能打包进区块链。两者都是关于时间的，但有一个很大差别：

LockTime ：绝对时间，用的是整个区块链的长度或者时间戳来表达的。

sequence number：相对时间，当前交易所引用的 UTXO 所在的块，后面追加了多少个块。

4.2transaction、UTXO 及钱包关系解析

（1）transaction

Transaction 之间的网状关系：一切交易可追溯

考虑如下场景：用户 A 和用户 B 之间发生了一个交易 T3：A 向 B 转 100 元。

那这 100 元，哪来的呢？ 来自 T1：C 向 A 转的 80 元 + T2：D 向 A 转的 30 元（共 110 元，但 A 只转了 100 元给 B，10 元找零返回给 A 的账号）。

同理，C 向 A 转的这 80 元，来自用户 E、F 的某次交易。

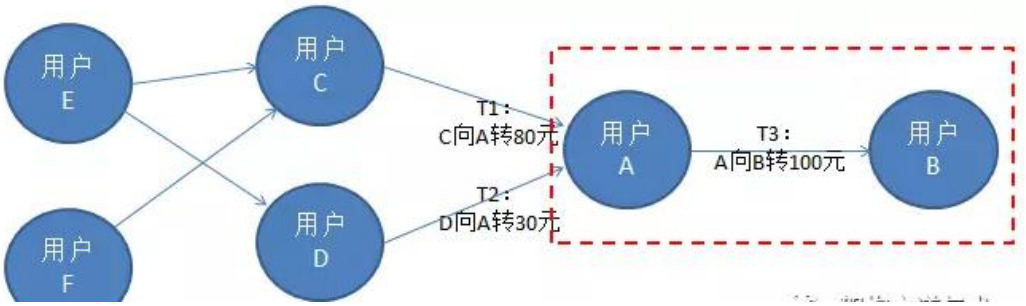
D 向 A 转的这 30 元，来自用户 E 的某次交易。。

举这个例子，是想说明一个问题：

交易与交易之间组成了网状关系，1 个交易的输出，成为了下 1 个交易的输入；下 1 个交易的输出，又成了下下 1 个交易的输入。

所有的钱，在这个网络中流动，每 1 笔钱的去向、来源，都是可追溯的，而这也是区块链网络的一个重要特点。

多个付款人 + 多个收款人



在现实生活的交易中，通常 1 笔交易就是 1 个付款人，1 个收款人。

但比特币网络的交易比这个灵活的多，支持 1 笔交易，有多个付款人 + 多个收款人，这个特性非常有用。

举个例子：

找零的场景：1 个用户的钱包里面有 100 元（来自 1 个 UTXO，什么是 UTXO，稍后解释），现在他要买 1 瓶矿泉水，只需要 1 块钱，但是 1 个 UTXO 有且只能花费 1 次，也就是说，他要把 100 元 1 次性花出去!!! 这怎么处理呢？

答案就是：

付款人：1 个，该用户，100 元

收款人：2 个，卖家：1 元　自己：99 元

也就是，把 100 元，分成 2 部分，打给了 2 个人，1 个是卖家，1 个是自己。

对应到交易里面，就是这笔交易有 1 个输入，2 个输出！

多个输入 + 多个输出

再次考虑最上面的场景：T3：A 向 B 转账 100 元

这个交易就有 2 个输入，2 个输出：

2 个输入（也就是 2 个 UTXO）：

T1: C 向 A 转的 80 元

T2：D 向 A 转的 30 元

2 个输出：

B：100 元

A：10 元（找零）

（2）UTXO　与 UTXO Set

UTXO 是一个非常非常核心的概念，搞明白了这个，也就搞明白了交易的内部结构、钱包等一系列东西。UTXO，全称是 Unspent Transaction Out，未花费的交易输出。

所有的钱都不是无源之水，别人给你的转账中（对应多笔交易的输出），有的被你花费了，有的没有被花费，没有被花费的叫 UTXO。

下面这张图，在上面的示意图的基础上，更详细的展示了交易之间的关联关系：

有 user1, user2, user3, user4, user5 5 个用户，

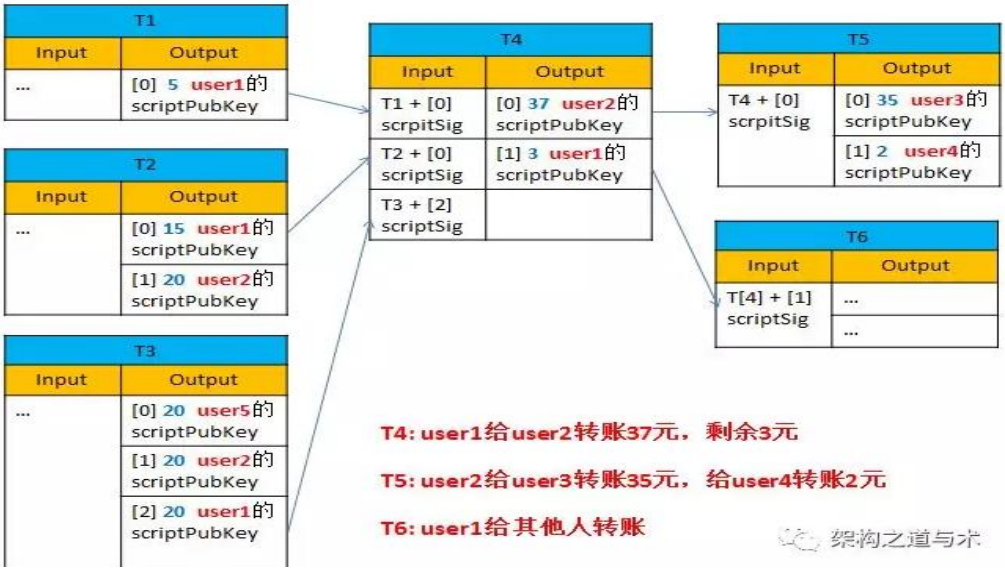
考虑 T4：user1 要给 user2 转账 37 元。user1 的 37 元来自哪呢？？

来自 T1，T2，T3 这 3 个交易：

T1 有 1 个 Output（也就是 1 个 UTXO，属于 user1，5 元），

T2 有 2 个 Output（也就是 2 个 UTXO，1 个属于 user1，15 元；1 个属于 user2，

20 元），
T3 有 3 个 Output（也就是 3 个 UTXO，分别属于 user5, 20 元；user2,20 元； user1，20 元)
在 T4 里，要凑够这 37 元，user1 需要 1 次性花费来自 T1, T2, T3 的 3 个 UTXO（5+15+20 = 40 元），同时产出了 2 个 UTXO，1 个给 user2 的 37 元，1 个是自己找零的 3 元。
这找零的 3 元，有被花到了下 1 个交易 T6。。。



通过上面的场景分析，我们知道了：

1）任何 1 笔 Transaction，会花费多个 UTXO（Input），同时也产生多个新的 UTXO（Output），属于多个不同的收款人。

2）1 个 UTXO，具有如下的表达形式：

1 个 UTXO = 1 个 Transaction ID + Output Index

3）旧的 UTXO 不断消亡，新的 UTXO 不断产生。所有的 UTXO，组成了 UTXO Set 的数据库，存在于每个节点

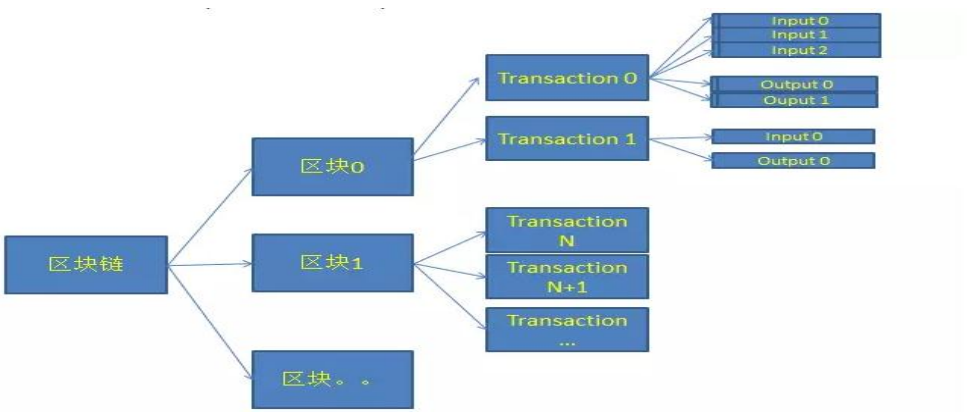
4）任何 1 笔 UTXO，有且仅可能被 1 个交易花费 1 次

（3）钱包

深刻理解了 UTXO 的概念，钱包就很容易理解了：

某个人的钱包的余额 = 属于他的 UTXO 的总和

在这里，你会发现一个不同于现实世界的“银行”里的一个概念：在银行里，会存储每个账号剩余多少钱。但这里，我们存储的并不是每个账号的余额，而**存的是一笔笔的交易**，也就是一笔笔的 UTXO，每个账户的余额是通过 UTXO 计算出来的，而不是直接存储余额!!



4.3 交易安全性如何保证？ -- scriptSig/scriptPubKey/Script Engine

A 给 B 转账的时候，会先用 A 的私钥进行签名，再转账给 B 的公钥。这个过程，说明了 2 件事：用 A 的私钥签名，证明了这笔钱是 A 的；转账给 B 的公钥，证明了这笔钱是转给 B 的，不是转给别人的。

接下来，B 要花这笔钱，比如转给 C；同样的，要用 B 的私钥签名，转账给 C 的公钥。

下面就来详细的分析一下 1 个 Transaction 的内部结构，看看究竟都是如何用私钥签名的，又是如何转账给公钥的。

交易的详细结构

下图是用的比特币的命令行工具 getrawtransaction 来详细展示了 1 个 Transaction ID 为 028cfae228f8a4b0caee9c566bd41aed36bcd237cdc0eb18f0331d1e87111743 的交易的内部结构：

该交易有 3 个输入（vin 里面），2 个输出（vout）。

我们会看到，每 1 个 vin 里面，都有 1 个 txid 字段，1 个 vout 字段，这 2 个字段合在一起也就组成了我们上节课所说的 UTXO。1 个 UTXO = 1 个 Transaction Id + output Index。

scriptSig 字段就是我们所说的，私钥的签名。

每 1 个 vout，有 1 个字段 value，字段 n，字段 scriptPubKey。

value 就是转账给对方的钱数，n 是 output 里面的编号（也就是下 1 次花费这个 UTXO 时，对应的 vin 里面的字段 vout）。

scriptPubKey 就是我们所说的，对方的公钥。


```
{
    "vout" : [
        {
            "value" : 0.84000000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 634228c26cf40a02a05db93f2f98b768a8e0e61b OP_EQUAL
                "hex" : "76a914634228c26cf40a02a05db93f2f98b768a8e0e61b88ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1A3q9pDtR4h8wpvyb8SVpiNPpT8ZNbHY8h"
                ]
            }
        },
        {
            "value" : 156.83000000,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 7514080ab2fcac0764de3a77d10cb790c71c74c2 OP_EQUAL
                "hex" : "76a9147514080ab2fcac0764de3a77d10cb790c71c74c288ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1Bg44FZsoTeYteRykC1XHz8facWYKhGvQ8"
```

scriptSig 详细解释:

scriptSig 里面有 asm 和 hex2 个字段，都是什么意思呢？

上面的 vin 里面，第 1 个输入，这 2 个字段的值，分别为：

```
"asm" : "3044022055bac1856ecbc377dd5e869b1a84ed1d5228c987b098c095030c12431a4d52490
2055523130a9d0af5fc27828aba43b464ecb1991172ba2a509b5fbd6cac97ff3af01
048aefd78bba80e2d1686225b755dacea890c9ca1be10ec98173d7d5f2fefbbf881a6e918f3b05
1f8aaaa3fcc18bbf65097ce8d30d5a7e5ef8d1005eaafd4b3fbe",

"hex" : "473044022055bac1856ecbc377dd5e869b1a84ed1d5228c987b098c
095030c12431a4d5249022055523130a9d0af5fc27828aba43b464ecb1991172
ba2a509b5fbd6cac97ff3af0141048aefd78bba80e2d1686225b755dacea890c9ca1be10ec
98173d7d5f2fefbbf881a6e918f3b051f8aaaa3fcc18bbf65097ce8d30d5a7e5
ef8d1005eaafd4b3fbe"
```

细心的读者会发现，这 2 个字段的值其实是一样的，只是 hex 在开头插入了

```
            }
        }
    ],
    "blockhash" : "0000000000007c639f2cbb23e4606a1d022fa4206353b9d92e99f5144bd74611",
    "confirmations" : 147751,
    "time" : 1301705313,
    "blocktime" : 1301705313
}
```


字符 47，中间插入了一个字符 41，其他部分完全一样。

asm 为 assembly（拼装，或者汇编的意思），hex 为其 16 进制的表达。

所以呢，这里的 asm 和 hex 就是同 1 个东西，里面包含了 2 部分：

signature（付款人的私钥的签名） + pub key（付款人的公钥），以 41 这个字符隔开。

scriptPubKey 详细解释:

scriptPubKey 里面有 asm, hex, reqSigs(暂时忽略）, type, address 几个字段。

同样，asm 和 hex 当中同 1 个意思，只是不同的编码格式；

type: 也就是前面我们说的 P2PKH，Pay to Public Key hash；

address: 也就是对方的收款地址，或者说钱包地址。

第 1 个 vout 的 scriptPubKey 的值如下：

OP_DUP OP_HASH160 634228c26cf40a02a05db93f2f98b768a8e0e61b OP_EQUALVERIFY OP_CHECKSIG

address 值为：

"1A3q9pDtR4h8wpvyb8SVpiNPpT8ZNbHY8h"

这 2 者是什么关系呢？ scriptPubKey 的格式，看起来很奇怪，OP_DUP, OP_HASH160，都是啥？？？

这就是接下来要讲的 script Engine.

单纯的看 scriptPubKey 是看不懂的，我们需要把 scriptPubKey 和下 1 个，要花费它的交易的 scriptSig，拼接在一起来看。

scriptPubKey 代表的是收款人的公钥信息；下 1 个要花费这个 UTXO 的交易里面的 vin，必然有该公钥对应的私钥的前面信息，也就是 scriptSig。

2 部分拼接在一起，scriptSig 在前，scriptPubKey 在后（注意：scriptPubKey 是当前交易的，scriptSig 是下 1 个交易的，要花费这个 UTXO 的），就长这个样子：

<signature> <pub key>

OP_DUP OP_HASH160 634228c26cf40a02a05db93f2f98b768a8e0e61b

OP_EQUALVERIFY OP_CHECKSIG

其中，signature, pub key 这 2 部分的内容，就类似上面的 vin 里面的 asm 部分，因为字符串太长了，此处就省略了，用<signature> <pub key> 代替。

上面这串东西，就是 1 个 script，1 个简单的 DSL 语言（关于什么是 DSL，自己百度之）。这串 script，被丢进 Script Engine 里面执行，执行结果就是 TRUE/FALSE，表示这个人有没有资格花这笔钱，或者说这笔交易是否有效。

Script Engine 是一个基于栈的脚本解释器，下面就来详细解释一下这串脚本是如何执行的：

从左往右，扫描上面这个 Script（也就是一个拼接的字符串），中间用空格隔开的，共 7 部分（7 个子串）。遇到字符串，则入栈； 遇到 OP_打头的，要么是 1 元操作符，栈顶 1 个元素出栈，计算，结果再入栈；要么是 2 元操作符，栈顶的 2 个元素出栈，计算，结果再入栈。

下面看一下入栈/出栈的详细过程：

<signature> //第 1 个字符串，签名，入栈

<pub key> //第 2 个字符串，pub key，入栈

OP_DUP //第 3 个字符串：1 元操作符，DUP 就是拷贝的意思。也就是把栈顶元素拷贝 1 份，入栈。现在栈顶元素是 pub key，也就是把 pub key 拷贝 1 份，入栈。

OP_HASH160 //第 4 个字符串：1 元操作符，对栈顶元素，做 HASH160 运算。也就是对 pub key 做 HASH160 运算，再入栈

634228c26cf40a02a05db93f2f98b768a8e0e61b //第 5 个字符串：其实就是 pub key hash，入栈

OP_EQUALVERIFY//第 6 个字符串：2 元操作符，校验栈顶的 2 个元素是否相等。此时栈顶的 2 元素，一个是刚入栈的 pub key hash, 一个是经过前面的 Hash160 计算出来的 pub key hash。两者不等，整个就结束了，返回 FALSE；
两者相等，全部出栈，此时栈里还剩 2 个元素：signature, pub key

OP_CHECKSIG//第 7 个字符串：操作符，检测栈顶的 2 个元素，pub key 和 signature 是否能对应的上。对应的上，说明这个签名的私钥，和收款人的公钥可以对上。有资格花这笔钱。

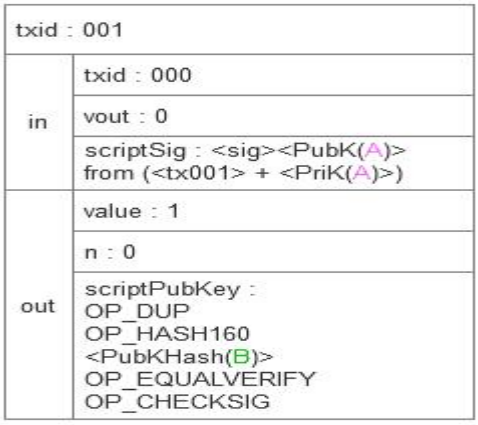
总结：

每 1 个节点上面都有这样 1 个 Script Engine，对于接收到的每 1 笔交易的每

个 vin，都会把其 scriptSig 和引用的 UTXO（也就是上 1 个交易的输出）的 scriptPubKey 拼接起来，形成 1 个字符串，塞入 Script Engine，得到结果是 True/False。如果是 False，说明该笔交易的付款人，没有资格花这笔钱，该笔交易不合法，会被直接丢弃。这个过程，也就是交易的验证过程。

4.4 比特币交易的签名与验证

（1）交易的构成



如上图所示，这是一个比特币交易的简化结构（忽略了一些参数，简化了 id）。比特币的交易主要由两部分构成：输入（input）和输出（output）。

1）Input 是要说明我打算花的这 UTXO 是从哪儿来的。具体的参数包括：

txid：引用的 UTXO 所在的那笔交易 ID

vout：引用的 UTXO 所在交易的输出中的序号（从 0 开始）

scriptSig：解锁脚本，包含付款人对本次交易的签名(<sig>)和付款人公钥(<PubK(A)>)。签名(<sig>):签名公式如下：

Sig = Fsig(Fhash(transaction), private key)

也就是，先对 Transaction 进行 1 次 Hash 运算，把这个 Hash 值 + private key，经过一个签名函数的计算，得到签名值。

2）Output 是要说明我打算生成几个 UTXO，分别给谁，每个 UTXO 里面有多少 BTC。具体的参数包括：

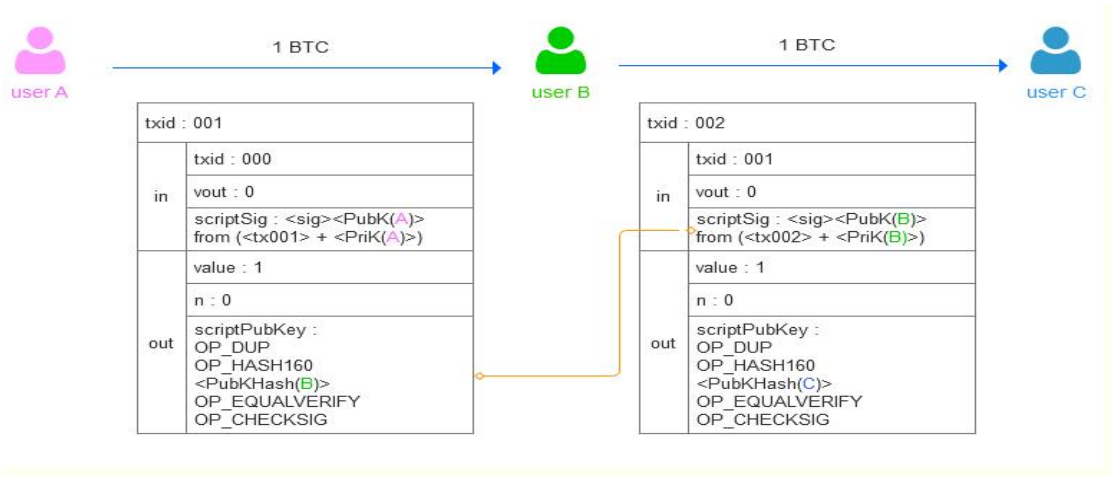
value：比特币数量

n：UTXO 序号（从 0 开始）

scriptPubkey：锁定脚本，包含命令（OP_DUP 等）和收款人的公钥哈希（<PubKHash(B)>)。

了解了交易的结构之后，现在我们通过一个交易的示例，来看看签名和验证是如何进行的。

（2）交易的签名



我们来看上面这张图，A 通过交易 001 转给 B 1BTC，B 通过交易 002 转给 C 1BTC，简化起见，忽略交易费的问题。

我们来重点分析交易 002：B 打算转给 C 1BTC，他先找到 A 转给他的那个 UTXO，即交易 001 的 out 中 n=0 的那个 UTXO，把相关参数写入交易 002 中的 in。然后在 out 中输入比特币数量，UTXO 序号，锁定脚本。**锁定脚本中的命令都是固定的**，C 的公钥哈希（<PubKHash(C)>）可通过 C 的钱包地址解码获得。

这样，交易 002 相关的数据都已经准备好了，就差最后的签名了。这个签名就类似于开支票时的签名，证明我同意把这笔钱给你。但是具体的实现要比签个字复杂很多，原因就在于互联网中一切都是可以复制的，如何证明你拥有这笔钱，如何证明这个交易是你创建的且没有被修改过，这背后都有严密的数学理论和算法来保证。我们先来看下**签名的过程**：

签名的输入：

1）待签名的交易数据（输入和输出），即<tx002>。

2）引用的 UTXO 相关信息（交易 ID、序号、锁定脚本）

3）B 的私钥，即<PriK(B)>。

4）签名类型

签名的输出：

scriptSig ，即解锁脚本，包含签名（<sig>）和 B 的公钥（<PubK(B)>）。

至此，一个完整的交易即创建成功，可以发送给其它节点验证了。

这里多说一句，细心的读者可能会发现，输入 2 的信息其实输入 1 已经包含了，或者可以根据输入 1 查的到，为什么还要单独列出呢。目前我也没有找到明确的可信服的解释，不知道是否还有其它深意。期待大神们的指教。

（3）签名的验证

交易发送至其它节点后，其它节点会对其进行验证，只有验证通过的交易才会被继续传播。交易验证的项目很多，这里只讲关于签名的验证。签名验证的目的有两个：

》证明交易所引用的 UTXO 的确属于付款人。

具体到本次交易，就是证明交易 001 的序号为 0 的 UTXO 的确是发给 B 的。

》证明交易的所有数据的确是付款人提供的，且未被修改过。

具体到本次交易，就是证明 B 的确创建了交易 002，且交易内的数据未被修改过。

下面我们来看验证是如何进行的，其实很简单，就是用**解锁脚本解锁对应UTXO 的锁定脚本**，对应上图就是**橙色线**所连接的两个脚本：

<sig><PubK(B)> OP_DUP OP_HASH160 <PubKHash(B)> OP_EQUALVERIFY OP_CHECKSIG

比特币脚本的执行基于堆栈模型，遵循**从左到右，后入先出**的原则。关于堆栈的介绍，文末的参考文章中有比较清晰的图示，不清楚的读者可以参考。本文为方便阐释各步骤的意义，采用文字方式描述。各步操作如下：

1）<sig> <sig> 入栈

2） <PubK(B)> <PubK(B)>入栈

3）OP_DUP 复制位于栈顶的<PubK(B)> ,将副本置于栈顶。

4）OP_HASH160 对位于栈顶的<PubK(B)>副本进行 HASH160,<PubK(B)>转变为<PubKHash(B)>。

5）<PubKHash(B)> <PubKHash(B)>入栈

6）OP_EQUALVERIFY 比较位于栈顶的两个元素是否相同，若相同则移除这两个元素，继续执行。若不同，则中断执行，返回失败。

7）OP_CHECKSIG 检查签名（注意栈内现有的元素为<sig><PubK(B)>）,根据结果返回成功或失败。

下面我们来分析下每一步的意义，步骤 1~6 的意义其实很明显，用 B 提供的公钥（<PubK(B)>）进行双哈希（HASH160），然后与锁定脚本中的公钥哈希（<PubKHash(B)>）作对比，相同则返回成功。我们知道公钥哈希（<PubKHash(B)>）就是 A 在创建交易时根据 B 的地址生成的，它就是 B 的公钥经过双哈希运算得来的，所以这一步只要提供了 B 的公钥，验证肯定是成功的。也就是证明了上文中提到的验证目的 1：证明交易 001 的序号为 0 的 UTXO 的确是发给 B 的。

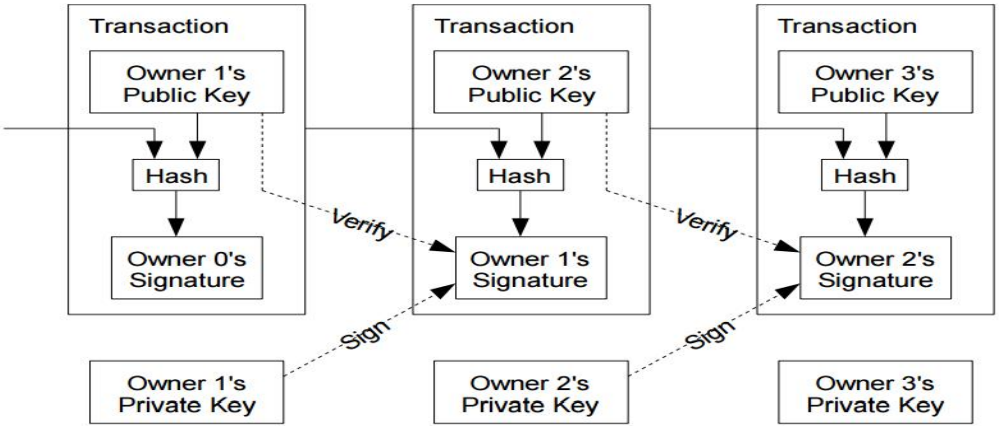
比较麻烦的是第 7 步，很多文章说到这里都只是泛泛而谈，或是一笔带过，我学习到这里的时候真是如堕雾中，四顾茫然啊。现在回过头再去看一些文章中的表述是非常不准确的。这一步简单的 CHECKSIG 操作，实际上蕴含了复杂的密码学和数学原理，证明的其实不是所有权的问题，而是证明了 B 的确创建了交易 002，且交易内的数据未被修改过，也就是上文中提到的验证目的 2。

那么，CHECKSIG 的验证是如何实现的呢？这里运用了椭圆曲线数字签名算法（ECDSA：The Elliptic Curve Digital Signature Algorithm ），一种利用椭圆曲线进行数字签名和验证的算法。下面将简单介绍这种算法是如何用来进行比特币交易的签名和验证的。涉及到的数学知识不作深入介绍，感兴趣的读者可参照文末的文章链接深入了解。

（4）参考

https://www.cnblogs.com/linguoguo/p/10392479.html

4.5 比特币原文交易解析



上图展示了比特币交易记录方式，是一个**基于时间序列的链式结构。**

（1）电子货币定义：将一枚电子货币定义为**一串数字签名**。

（2）如何将电子货币从当前所有者发送给下一位所有者？

货币的当前所有者通过将**上一次交易和下一位所有者的公钥**执行 hash，并对 **hash 值**进行数字签名（使用货币当前拥有者的私钥对 hash 值进行加密签名）及将该数字签名附加在货币的末尾的方式就完成了电子货币从当前所有者发送给下一位所有者过程。

（3）原文中“我们将一枚电子货币（an electronic coin）定义为一串数字签名”，其实**它是一条记录（数据）**，也就是说你**拥有的交易记录**才是你的数字资产。需要注意的是，这条记录里面没有你的姓名、身份证号等这些中心化系统中常见的识别所有者的字段，**唯一表示你身份的是你的公钥（钱包），证明你所有权的是你的私钥（密码）。**

（4）在比特币系统中解决双重支付问题。

实际上我们需要关注的**只是于本交易之前发生的交易，而不需要关注这笔交易发生之后是否会有双重支付的尝试**。为了确保某一次交易是不存在的，那么唯一的方法就是**获悉之前发生过的所有交易。交易信息就应当被公开宣布（publicly announced），我们需要整个系统内的所有参与者，都有唯一公认的历史交易序列**。收款人需要确保在交易期间绝大多数的节点都认同该交易是**首次出现**。

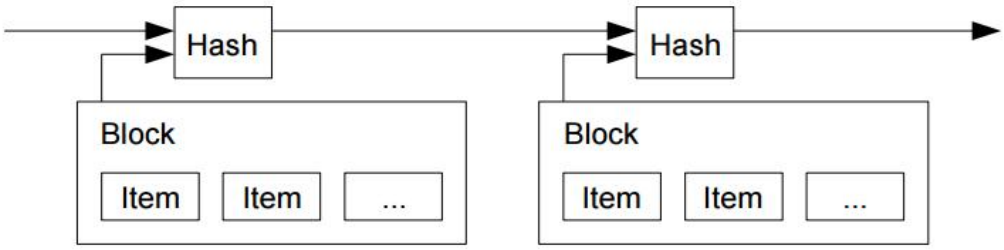
5 时间戳

时间戳解决了交易存在性问题。

【原文】

本解决方案首先提出一个“时间戳服务”。时间戳服务通过对以区块(block)

形式存在的一组 item 实施随机散列（hash）后加上时间戳 ，并将该随机散列（hash）进行广播，就像在新闻或世界性新闻组网络（Usenet）的发帖一样。显然，该时间戳能够**证实特定数据必然于某特定时刻是的确存在**的，因为只有在该时刻存在了才能获取相应的随机散列值（hash）。每个时间戳应当将前一个时间戳（也就是前一个区块的 hash 值）纳入其随机散列值（hash）中，每一个随后的时间戳都对之前的一个时间戳进行增强(reinforcing)，这样就形成了一个链条（Chain）。



【解读】

（1）时间戳的作用

时间戳的主要作用就是**证实特定数据必然于某特定时刻是的确存在。时间戳解决了交易存在性问题。**

（2）时间戳的获取来源

时间来自于连接的其他节点（node）时间的中位数（median，是个数学概念，比平均值更不受极端数字影响），要求连接的节点（node）数量至少为 5，中位数和本地系统时间差别不超过 70 分钟，否则会提醒你更新本机的时间。同时，在接收到新的 block 时会拒绝时间与自己差距+2 小时和-(前 11 个 block 时间中位数)的 block，**由此我们知道了这个时间的正确来源了，还是利用大多数人的机器时间，其实比特币的核心理念一直是认为“多数人是正义”的。**

假设新的发现的区块为 B(x),时间戳为 TB(x),则与前一个区块 B(n)的时间戳 TB(n)关系：-(前 11 个 block 时间中位数)<TB(x)- TB(n)<2

（3）区块链形成方式

区块的哈希实际上就是区块头的哈希。也就是说：

HASH(block)= HASH(block header)=

HASH(version,preblockhash,merkletreehash,bits,timestamp,nonce)

每一个区块头中都包含上一个区块的哈希值，这样就形成一条基于时间戳的区块链！

6 共识机制：工作量证明（POW）

6.1POW 的作用

2008 年 10 月，中本聪（Satoshi Nakamoto）发表了论文《Bitcoin: A Peer-to-Peer

Electronic Cash System》，在论文中设计了区块链的 POW 的共识机制。那 POW 共识机制主要的作用是什么？仔细分析比特币的整体设计思路，使用非对称秘钥解决了电子货币的所有权问题，使用区块时间戳解决了交易的存在性问题，用分布式账本解决了剔除第三方机构后交易的验证问题。最后就剩下双重支付问题，在点对点的分布式系统中，要保障所有节点的数据一致性，就必须引入一种机制来保障。中本聪设计了 POW 共识机制，通过该机制消除双重支付，保证所有节点数据的一致性。

6.2 比特币 POW 实现方式

比特币采用 SHA256 哈希算法，逻辑上是对整个区块进行哈希运算，实际上是对区块头执行哈希运算。也就是说，所谓的区块的哈希值，更确切的表述为区块头的哈希值。区块头大小为 80byte，包含六个字段：version、timestamp、bits、nonce、 hashpreblock 及 hashmerkleroot。如此设计首先带来的好处是方便哈希运算，每次运算只需要 80 字节的参数输入，而不是整个区块的数据，同时交易列表的任何变化又能体现在哈希运行结果上。

比特币采用 SHA256 哈希运算，且每次都是连续进行两次 SHA256 运算才能作为最终结果，前一次运算的结果作为后一次运算的输入，即 Double SHA256，一般简称 SHA256D，比特币合格区块判断依据如下：

SHA256D(Version,hashPreBlock,hashMerkleRoot,Timestamp,Bits,Nonce)<=MAXTARGET/Diff

比特币工作量证明（POW）的达成就是矿工计算出来的区块哈希值必须小于目标值。比特币工作量证明的过程，就是通过不停的变换区块头（即尝试不同的 nouce 值）作为输入进行 SHA256 哈希运算，找出一个特定格式哈希值的过程（即要求有一定数量的前导 0）。而要求的前导 0 的个数越多，代表难度越大。我们可以把比特币矿工解这道工作量证明迷题的步骤大致归纳如下：

（1）生成用于发行新比特币奖励的 Coinbase 交易，并与当前时间戳的其他所有准备打包进区块的交易组成交易列表，通过 Merkle Tree 算法生成 Merkle Root Hash；

（2）把 Merkle Root Hash 及其他相关五个字段组装成区块头，其中 nonce 置零，将区块头的 80 字节数据（Block Header）作为工作量证明的输入；

（3）不停的变更区块头中的随机数即 nonce 的数值（nonce 初始置零，每次增 1），并对每次变更后的的区块头做双重 SHA256 运算（即 SHA256(SHA256(Block_Header))），将每次结果值与当前网络的目标值做对比，如果小于目标值，则成功搜索到合适的随机数 nonce 并获得该区块的记账权，工作量证明完成。

（4）如果在当前时间戳未成功，则更新时间戳，重复上述步骤，直到找到符合条件的 nonce。

（5）在节点成功找到满足条件的哈希值之后，会马上对全网进行广播打包区块。

（6）网络的其他节点收到广播打包区块，会立刻对区块的哈希值及交易数据的有效性进行验证。如果验证通过，则表明已经有节点成功解迷，自己就不再竞争当前区块打包，而是选择接受这个区块，记录到自己的账本中，然后进行下一个区块的竞争猜谜。

7 网络

运行该网络的步骤如下：
1)新的交易（transaction）向全网节点进行广播；
2)每一个节点都将收到的交易信息纳入一个区块中；
3)每个节点都尝试为自己的区块中找到一个具有足够难度的工作量证明；
4)当一个节点找到了一个工作量证明（pow），它就向全网进行广播；
5)当且仅当包含在该区块中的所有交易都是有效的且之前未存在过的，其他节点才认同该区块的有效性；
6)其他节点表示他们接受该区块，而表示接受的方法，则是在跟随该区块的末尾，制造新的区块以延长该链条，而新区块的随机散列值（hash）则基于上一个区快的随机散列值。

节点始终都将最长的链条视为正确的链条，并持续工作和延长它。如果有两个节点同时广播不同版本的新区块，那么其他节点在接收到该区块的时间上将存在先后差别。当此情形，他们将在率先收到的区块基础上进行工作，**但也会保留另外一个链条，以防后者变成最长的链条**。该僵局（tie）的打破要等到下一个工作量证明被发现，而其中的一条链条被证实为是较长的一条，那么在另一条分支链条上工作的节点将转换阵营，开始在较长的链条上工作。

所谓“新的交易要广播”，实际上不需要抵达全部的节点。只要交易信息能够抵达足够多的节点，那么他们将很快被整合进一个区块中。而区块的广播对被丢弃的信息是具有容错能力的。如果一个节点没有收到某特定区块，那么该节点将会发现自己缺失了某个区块，也就可以提出自己下载该区块的请求。

8 激励机制

【原文】

我们约定如此：每个区块的第一笔交易进行特殊化处理，该交易产生由该区块创造者拥有的新的电子货币。这样就增加了节点支持该网络的激励，并在没有中央集权机构发行货币的情况下，提供了一种将电子货币分配到流通领域的一种方法。这种将一定数量新货币持续增添到货币系统中的方法，非常类似于耗费资源去挖掘金矿并将黄金注入到流通领域。此时，CPU 的时间和电力消耗就是消耗的资源。

另外一个激励的来源则是交易费（transaction fees）。如果某笔交易的输出值小于输入值，那么差额就是交易费，该交易费将被增加到该区块的激励中。只要既定数量的电子货币已经进入流通，那么激励机制就可以逐渐转换为完全依靠交易费，那么本货币系统就能够免于通货膨胀。

激励系统也有助于鼓励节点保持诚实。如果有一个贪婪的攻击者能够调集比

所有诚实节点加起来还要多的 CPU 计算力，那么他就面临一个选择：要么将其用于诚实工作产生新的电子货币，或者将其用于进行二次支付攻击。那么他就会发现，按照规则行事、诚实工作是更有利可图的。因为该等规则使得他能够拥有更多的电子货币，而不是破坏这个系统使得其自身财富的有效性受损。

【解读】

（1）比特币的来源

在 bitcoin 的系统中，中本聪已经规定了总量就是 2100W 个 bitcoin。而这些 bitcoin 的产生是每产生 210000 个区块就减半(以现在约定大约每 10 分钟产生一个区块的速度大约到 2140 年产生为 0)。而 bitcoin 产生的方法就是**给抢到记账权的人凭空给予一定量的货币**，这样就同时解决的货币发行和矿工记账的奖励(就像现实中挖黄金的黄金矿工一样)。”**每个区块的第一笔交易进行特殊化处理，该交易产生一枚由该区块创造者拥有的新的电子货币**“。这种凭空奖励的方法，就是每个区块的第一笔交易，是一个特殊的交易，这个交易就是只有 output，且对于输入这个 output 的 input(前一个交易的 output)是个空。

（2）比特币发行机制

简单的说，大概每十分钟发行一次。

新比特币在每个网络节点在解决了一定的数学计算（比如，创建新的 block）后生成。这个生成过程被认为是难以重现和 proof of work 的。解决问题后得到的回报是 automatically adjusted，因此在比特币网络的头 4 年，将会产生总额为 10,500,000 BTC 的比特币。这个数量每隔 4 年就自动减半，也就是说在第 4 至第 8 年会产生 5,250,000 BTC，第 8 至 12 年则只有 2,625,000 BTC，如此类推。到最后（2140），总共产生的比特币数量为接近 21,000,000 BTC。

另外，伴随着网络一同建立的还有一个系统。平均每隔 10 分钟，该系统就尝试去收集网络上产生的 block 里面的新比特币。创建新比特币的难度系数是随着参与尝试产生新比特币的人数而变化的。整个网络一致认可基于产生最前面的 2016 个 block 所花的时间实现这些行为。因此，难度系数与产生这些最早的 block 所花的时间内投入到产生这些新的比特币的平均计算资源有关。某个人“发现”一个 block 的可能性是他所用的计算资源和所有同时在网络上生成 block 的计算资源的比值。

（3）挖完所有比特币所需时间

最后一个产出比特币的区块将是#6,929,999 号区块，这大约会在公元 2140 年前后发生。届时流通中比特币的总数将恒定维持在 20,999,999,9769 BTC。

即使比特币的允许精度从目前的 8 位小数扩展，最终流通中的比特币将总是略低于 2100 万（假定其他参数不变）。例如，如果引入 16 位小数精度，最终的比特币总额将是 20999999.999999999496 BTC。

（4）激励机制的奖励主要有两部分：挖到新区块的报酬+交易手续费

9 回收磁盘空间

如果最近的交易已经被纳入了足够多的区块之中，那么就可以丢弃该交易之前的数据，以回收硬盘空间。为了同时确保不损害区块的随机散列值（hash），交易信息通过执行 hash 构建成一种 Merkle 树（Merkle tree）的形态，使得只有根(root)被纳入了区块的随机散列值（hash）。通过将该树（tree）的分支拔除

（stubbing）的方法，老区块就能被压缩。而内部的随机散列值（hash）是不必保存的。

不包含交易的区块头大约 80 字节。我们假设每 10 分钟生成一次区块，每年大小为 80 bytes * 6 * 24 * 365 = 4.2MB。2008 年 PC 系统通常的内存容量为 2GB，按照摩尔定理预言的每年增长 1.2GB 的大小，即使将全部的区块头存储在内存之中都不是问题。

10 简化的支付验证（SPV）

在不运行完整网络节点的情况下，也能够对支付进行检验。一个用户需要保留最长的工作量证明链条的区块头的拷贝，它可以不断向网络发起询问，直到它确信自己拥有最长的链条，并能够通过 merkle 的分支通向它被加上时间戳并纳入区块的那次交易。节点想要自行检验该交易的有效性是不可能的，但可以通过追溯到链条的某个位置，它就能看到某个节点曾经接受过它，并且于其后追加的区块也进一步证明全网曾经接受了它。

因此，只要诚实的节点控制了网络，检验机制就是可靠的。但是，当全网被一个计算力占优的攻击者攻击时，将变得较为脆弱。因为网络节点能够自行确认交易的有效性，只要攻击者能够持续地保持计算力优势，简化的机制会被攻击者焊接的（fabricated）交易欺骗。那么一个可行的策略就是，只要他们发现了一个无效的区块，就立刻发出警报，收到警报的用户将立刻开始下载被警告有问题的区块或交易的完整信息，以便对信息的不一致进行判定。对于日常会发生大量收付的商业机构，可能仍会希望运行他们自己的完整节点，以保持较大的独立完全性和检验的快速性。

11 组合与分割价值

虽然可以独立地对电子货币进行处理，但是对于每一枚电子货币单独发起一次交易将是一种笨拙的办法。为了使得价值易于组合与分割，**交易被设计为可以包含多个输入和输出。**一般而言是某次价值较大的前次交易构成的单一输入，或者由某几个价值较小的前次交易共同构成的并行输入，但是输出最多只有两种类型：一个类型用于支付，另一个类型用于找零（如有）。

需要指出的是，当一笔交易依赖于之前的多笔交易时，这些交易又各自依赖于多笔交易，但这并不存在任何问题。因为这个工作机制并不需要展开检验之前发生的所有交易历史。

12 隐私

传统的银行模型为交易的参与者提供了一定程度的隐私保护，因为试图向可信任的第三方索取交易信息是严格受限的。但是如果将交易信息向全网进行广播，就意味着这样的方法失效了。但是隐私依然可以得到保护：将公钥保持为匿名的方式。公众得知的信息仅仅是有某个人将一定数量的货币发给了另外一个人，但是难以将该交易同特定的人联系在一起，也就是说，公众难以确信，这些人究竟是谁。这同股票交易所发布的信息是类似的，股票交易发生的时间、交易量是记录在案且可供查询的，但是交易双方的身份信息却不予透露。

作为额外的预防措施，使用者可以让每次交易都生成一个新的密钥对，以确保这些交易不被追溯到一个共同的所有者。但是由于并行输入的存在，一定程度上的追溯还是不可避免的，因为并行输入表明这些货币都属于同一个所有者。此时的风险在于，如果某个人的某一个公钥被确认属于他，那么就可以追溯出此人的其它很多交易。

参考：

[1] http://chainb.com/?P=Cont&id=6

[2]

http://www.ruanyifeng.com/blog/2011/08/what_is_a_digital_signature

[3] https://bitcoin.org/bitcoin.pdf

[4]

http://www.8btc.com/wiki/bitcoin-a-peer-to-peer-electronic-cash-system

[5] http://www.8btc.com/bitcoin-story

[6] https://www.jianshu.com/p/ca0c0a0e0faa

[7]http://crypto.stackexchange.com/questions/22669/merkle-hash-tree-updates

[8] https://www.cnblogs.com/sanghai/p/7608701.html

[9]http://www.8btc.com/pos-white-book

[10]http://www.8btc.com/blackcoin-pos

[11] https://www.cnblogs.com/sueyyyy/p/9726812.html

[12]

https://baike.baidu.com/link?url=sgRa5dn5lzrg-1_eaRX1EtZeL3XucbEADxOjoBsYRyQSV_BnLqHgusjR8QRUGGnbmCl0sQOgx7dnVz7ItSLdu7axz8BIbt57JF4qbMkBzO_xuwIeugZVabWMUijeIq2SdqMZcoFZabwhaYIcIhlTQ_

[13] http://www.jianshu.com/p/d706b44b1e1e

[14] https://www.zhihu.com/question/52365546/answer/130192361

[15] http://www.8btc.com/tan90d97

[16] https://github.com/ethereum/go-ethereum/wiki/Private-network