区块链基础

很多人进入币圈是因为暴富神话，但实际上所有的暴富只不过是猎人伪装的陷阱，只是你自己不认为自己是猎物而已。

--xfli5

资料来源：网络

整理：xfli5

推特：@xfli5

一.区块链是个什么东西？

【严格定义】

区块链（Blockchain）是指通过基于密码学技术设计的共识机制方式，在 P2P 对等网络中多个节点共同维护一个持续增长，由时间戳和有序记录数据块所构建的链式列表账本的分布式数据库技术。该技术方案让参与系统中的任意多个节点，把一段时间系统内全部信息交流的数据，通过密码学算法计算并记录到一个数据块（block），并且生成该数据块的 hash 用于链接（chain）下个数据块和校验，系统所有参与节点来共同认定记录是否为真。

【通俗解释】

无论多大的系统或者多小的网站，一般在它背后都有数据库。那么这个数据库由谁来维护？在一般情况下，谁负责运营这个网络或者系统，那么就由谁来进行维护。如果是微信数据库肯定是腾讯团队维护，淘宝的数据库就是阿里的团队在维护。大家一定认为这种方式是天经地义的，但是区块链技术却不是这样。

如果我们把数据库想象成是一个账本 :比如支付宝就是很典型的账本，任何数据的改变就是记账型的。数据库的维护我们可以认为是很简单的记账方式。在区块链的世界也是这样，区块链系统中的每一个人都有机会参与记账。系统会在一段时间内，可能选择十秒钟内，也可能十分钟，选出这段时间记账最快最好的人，由这个人来记账，他会把这段时间数据库的变化和账本的变化记在一个区块（block）中，我们可以把这个区块想象成一页纸上，系统在确认记录正确后，会把过去账本的数据指纹链接（chain）这张纸上，然后把这张纸发给整个系统里面其他的所有人。然后周而复始，系统会寻找下一个记账又快又好的人，而系统中的其他所有人都会获得整个账本的副本。这也就意味着这个系统每一个人都有一模一样的账本，这种技术，我们就称之为区块链技术（Blockchain），也称为分布式账本技术。

由于每个人（节点）都有一模一样的账本，并且每个人（节点）都有着完全相等的权利，因此不会由于单个人（节点）失去联系或宕机，而导致整个系统崩溃。既然有一模一样的账本，就意味着所有的数据都是公开透明的，每一个人可以看到每一个账户上到底有什么数字变化。它非常有趣的特性就是，其中的数据无法篡改。因为系统会自动比较，会认为相同数量最多的账本是真的账本，少部分和别人数量不一样的账本是虚假的账本。在这种情况下，任何人篡改自己的账本是没有任何意义的，因为除非你能够篡改整个系统里面大部分节点。如果整个系统节点只有五个、十个节点也许还容易做到，但是如果有上万个甚至上十万个，并且还分布在互联网上的任何角落，除非某个人

能控制世界上大多数的电脑，否则不太可能篡改这样大型的区块链。

二.区块链有什么优良品质？

区块链有四个主要的特性：去中心化（Decentralized）、去信任（Trustless）、集体维护（Collectively maintain）、可靠数据库（Reliable Database）。并且由四个特性会引申出另外 2 个特性：开源（Open Source）、隐私保护（Anonymity）。如果一个系统不具备这些特征，将不能视其为基于区块链技术的应用。

（1）去中心化（Decentralized）

整个网络没有中心化的硬件或者管理机构，任意节点之间的权利和义务都是均等的，且任一节点的损坏或者失去都会不影响整个系统的运作。因此也可以认为区块链系统具有极好的健壮性。

（2）去信任（Trustless）

参与整个系统中的每个节点之间进行数据交换是无需互相信任的，整个系统的运作规则是公开透明的，所有的数据内容也是公开的，因此在系统指定的规则范围和时间范围内，节点之间是不能也无法欺骗其它节点。

（3）集体维护（Collectively maintain）

系统中的数据块由整个系统中所有具有维护功能的节点来共同维护的，而这些具有维护功能的节点是任何人都可以参与的。

（4）可靠数据库（Reliable Database）

整个系统将通过分布式数据库的形式，让每个参与节点都能获得一份

完整数据库的拷贝。除非能够同时控制整个系统中超过 51%的节点，否则单个节点上对数据库的修改是无效的，也无法影响其他节点上的数据内容。因此参与系统中的节点越多和计算能力越强，该系统中的数据安全性越高。

（5）开源（Open Source）

由于整个系统的运作规则必须是公开透明的，所以对于程序而言，整个系统必定会是开源的。

（6）隐私保护（Anonymity）

由于节点和节点之间是无需互相信任的，因此节点和节点之间无需公开身份，在系统中的每个参与的节点的隐私都是受到保护的。

三.三区块链基础

1 区块链基础--Hash（哈希）函数

1.1 基本概念

在实际的通信安全中，除了要求实现数据的保密性之外，还要求保证**数据的完整性**。密码学中的 **hash 函数**可为数据完整性提供保障。Hash 函数用来构造数据的短“指纹”（函数值）；一旦数据改变，指纹就不再正确。即使数据被存储在不安全的地方，通过重新计算数据的指纹并验证是否改变，就可以检测数据的完整性。

Hash 函数 h 用于将**任意长的消息（或数据）x** 映射为**较短的、固定长度**的一个值 h(x)。我们称 h(x)为数据摘要或者哈希值或者散列值等。通常，要求密码

学 hash 函数（安全 hash 函数）h 满足以下三点安全属性：

（1）抗原像攻击（Preimage Resistant）：已知 y ∈ Y（数据摘要集合），要找出 x ∈ X（数据报文集合），使得 h(x)=y 是困难的（在计算上是不可行的），也称单向性（One-way）。

（2）抗第二原像攻击(Second Preimage Resistant)：对于给定的 x ∈ X，找出另一个 x' ∈ X，使得 h(x')=h(x)是困难的（计算不可行），也称弱抗碰撞性(Weak Collision-Resistant)。

（3）抗碰撞性(Collision-Resistant)：找出任意两个不同的 x,x' ∈ X，使得 h(x)=h(x')是困难的（计算不可行），也称强抗碰撞性（Strong Collision-Resistant ）。

我们假设 x，x'是两个不同的消息，如果 h(x)=h(x'),则称 x，x'是 hash 函数的一个碰撞。如果这种情况很容易发生，这也表明该 hash 函数是不安全的。

举一个很棒的 hash 函数例子：如果把一段消息 x 比作一个人，这段消息的哈希值就是这个人的指纹，是这个人的独一无二的特征。

1.2 著名的 hash 算法

（1）SHA256：在比特币协议中，使用的哈希函数是 SHA256。2008 年，中本聪发明比特币时采用的这个 SHA-256 被公认为最安全最先进的算法之一。除了生成地址中有一个环节使用了 REPID160 算法，比特币系统中但凡有需要做 Hash 运算的地方都是用 SHA256。

SHA256 是安全散列算法 SHA（Secure Hash Algorithm）系列算法之一，**其摘要长度为 256bits，即 32 个字节，故称 SHA256**。SHA 系列算法是美国国家安全局 （NSA） 设计，美国国家标准与技术研究院（NIST） 发布的一系列密码散

列函数，包括 SHA-1、SHA-224、SHA-256、SHA-384 和 SHA-512 等变体。

类别	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
消息摘要长度	160	224	256	384	512
消息长度	小于2^{64}位	小于2^{64}位	小于2^{64}位	小于2^{128}位	小于2^{128}位
分组长度	512	512	512	1024	1024
计算字长度	32	32	32	64	64
计算步骤数	80	64	64	80	80

接下来我们详细介绍 SHA256 算法：

SHA-256 算法输入报文的最大长度不超过 2^64 bit，输入按 512-bit 分组进行处理，产生的输出是一个 256-bit 的报文摘要。该算法处理包括以下几步：

1）补位

消息必须进行补位，以使其长度（Lmod512=448，L<2^64）在对 512 取模以后的余数是 448。补位是这样进行的：先在消息的末尾补一个 1，然后再补 0，直到长度满足对 512 取模后余数是 448。总而言之，补位是至少补一位，最多补 512 位。以信息“ABC”为例显示补位的过程。

原始信息（ABC（ABC 是指 ASCII 码）对应的二进制）:01100001 01100010 01100011

补位第一步：0110000101100010 01100011 **1**

首先补一个“1”

补位第二步：0110000101100010 01100011 1**0……0**

然后补 423 个“0”

我们可以把最后补位完成后的数据用 16 进制写成下面的样子

```
61626380 00000000000000000 00000000

00000000 00000000000000000 00000000

00000000 00000000000000000 00000000

00000000 00000000
```

现在，数据的长度是 448 了，我们可以进行下一步操作。

注：为什么是 448，因为 448+64=512. 第二步会加上一个 64bit 的原始报文的长度信息。

2)补长度

所谓的补长度是将原始消息的**长度**（原始消息是 ABC，对应的二进制长度为 24）补到已经进行了补位操作的消息后面。通常用一个 **64 位的数据**来表示原始消息的**长度**。如果消息长度不大于 2^64，那么第一个字就是 0。在进行了补长度的操作以后，整个消息就变成下面这样了（16 进制格式）

```
61626380 00000000000000000 00000000
00000000 00000000000000000 00000000

00000000 00000000000000000 00000000

00000000 00000000000000000 00000018
```

如果原始的消息长度超过了 512，我们需要将它补成 512 的倍数。然后我们把整个消息分成一个一个 512 位的数据块，分别处理每一个数据块，从而得到消息摘要。

3）使用的常量

在 SHA256 算法中，用到 64 个常量，这些常量是对自然数中前 64 个质数的立方根的小数部分取前 32bit 而来。这 64 个常量用 16 进制表示如下：

```
 428a2f98 71374491 b5c0fbcf e9b5dba5
3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3
72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc
2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7
c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13
650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3
d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5
391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208
90befffa a4506ceb bef9a3f7 c67178f2
```

4）需要使用的函数

SHA-256 采用 6 个逻辑函数，每个函数均基于 32 位字运算，这些输入的 32 位字我们记为 x、y、z，同样的这些函数的计算结果也是一个 32 位字。这些逻辑函数表示如下：

```
CH (x, y, z) = (x AND y) XOR ( (NOT x) AND z)
MAJ( x, y, z) = (x AND y) XOR (x AND z) XOR (y AND z)
BSIG0(x) = ROTR^2(x) XOR ROTR^13(x) XOR ROTR^22(x)
BSIG1(x) = ROTR^6(x) XOR ROTR^11(x) XOR ROTR^25(x)
SSIG0(x) = ROTR^7(x) XOR ROTR^18(x) XOR SHR^3(x)
SSIG1(x) = ROTR^17(x) XOR ROTR^19(x) XOR SHR^10(x)
```

注：

其中 x、y、z 皆为 32bit 的字；

ROTR^k(x)是对 x 进行循环右移 k 位；例如针对二进制，循环右移就是形成环，例如 1110---循环右移两位---1011

SHR^3（x）是对 x 进行

XOR:相同为 0，不同为 1.例如 1110 XOR 0010 = 1100

5）计算消息摘要

基本思想：就是将消息分成 N 个 512bit 的数据块，八个哈希初值 H(0)n（n=0,1…7）经过第一个数据块得到 H（1）n，H（1）n 经过第二个数据块得到 H（2）n，……，依次处理，最后得到 H（N）n，然后将 H（N）n 的 8 个 32bit(H（N）0, H（N）1…H（N）7)连接成 256bit 消息摘要。

I、哈希初值 H（0）

SHA256 算法中用到的哈希初值 H（0）如下

```
H(0)0 = 6a09e667
H(0)1 = bb67ae85
H(0)2 = 3c6ef372
H(0)3 = a54ff53a
H(0)4 = 510e527f
H(0)5 = 9b05688c
H(0)6 = 1f83d9ab
H(0)7 = 5be0cd19
```

注：这些初值是对自然数中前 8 个质数 3、5、7、11 等的平方根的小数部分取前 32bit 而来，以 16 进制表示。

II、 计算过程中用到的三种中间值

1、64 个 32bit 字的 message schedule 标记为 w0、w1、…、w63。

2、8 个 32bit 字的工作变量标记为 a、b、c、d、e、f、g、h。

3、包括 8 个 32bit 字的哈希值标记为 H(i)0、…、H(i)7。

III、 工作流程

原始消息分为 N 个 512bit 的消息块。每个消息块分成 16 个 32bit 的字标记

为 M(i)0、M(i)1、M(i)2、…、M(i)15 然后对这 N 个消息块依次进行如下处理

对 N 个消息块依次进行以下四步操作后将最后得到的 H(N)0、H(N)1、H(N)2、…、H(N)7 串联起来即可得到最后的 256bit 消息摘要。

```
    For i=1 to N
1)    For t = 0 to 15
            Wt = M(i)t
        For t = 16 to 63

            Wt = SSIG1(W(t-2)) + W(t-7) + SSIG0(t-15) + W(t-16)
2)  a = H(i-1)0
      b = H(i-1)1
      c = H(i-1)2
      d = H(i-1)3
      e = H(i-1)4
      f = H(i-1)5
      g = H(i-1)6
      h = H(i-1)7
3) For t = 0 to 63
          T1 = h + BSIG1(e) + CH(e,f,g) + Kt + Wt
          T2 = BSIG0(a) + MAJ(a,b,c)
          h = g
          g = f
          f = e
          e = d + T1
          d = c
          c = b
          b = a
          a = T1 + T2

  4) H(i)0 = a + H(i-1)0
        H(i)1 = b + H(i-1)1
        H(i)2 = c + H(i-1)2
        H(i)3 = d + H(i-1)3
        H(i)4 = e + H(i-1)4
        H(i)5 = f + H(i-1)5
        H(i)6 = g + H(i-1)6
        H(i)7 = h + H(i-1)7
```

（2）MD5：对输入以 512 位(bit)分组，其输出是 4 个 32 位的级联。从密码分析的角度来看，MD5 被认为是易受攻击的。2004 年山东大学王小云采用“比特追踪法”可以很快的找到一个碰撞，破译 MD5。

2 区块链基础--二进制存储方式

2.1.二进制

二进制，有符号数，首位为 1 的是负数，首位为 0 的是正数。(规定）

无符号数没有正负之分，所以也没有首位的限制。(规定)

判断一个二进制是正数还是负数，要先看其在计算机中是**以有符号进行存储还是无符号进行存储**。

》如果是无符号存储，则为原码存储，其为一个正数。

》若是有符号存储，则为补码存储。看其最高位，最高位为 0，为正数，反之，为负数。

2.1.1.无符号存储

无符号数的原码、反码、补码都一样，皆为该数的二进制表示法

针对这种情形，均为原码存储，代表的是一个正数。比如针对一个字节的二进制最小值：00000000；最大值：11111111.范围 0~255.

针对无符号二进制 01100100 对应的十进制 1*2^6+1*2^5+1*2^2=100

2..1.2 有符号存储

我们以一个字节的二进制为例，其首位代指符号，0 表示正数，1 代指负数，后面七位指代数值。最小 1000 0000，最大 01111111，范围-128（1000 0000）~127（01111111）.

针对这种情形，均为补码存储。

(1)正数

正数的原码为该数的二进制表示法。 正数的反码与原码一样。 正数的补码与原码一样。

例如：1==>0000 0001(原码)==>0000 0001(反码)==>0000 0001(补码)

（首位为符号位，首位 0 表示正数，所以原码、反码和补码的首位都是 0）

(2)负数

（1）负数原码

负数的原码为该数对应的无符号数的二进制，将首位置 1。

例如：-1==>1(无符号数)==>0000 0001(无符号数的二进制)==>1000 0001(原码)。

(首位为符号位，首位 1 表示负数，后七位表示数值)

（2）负数反码

负数的反码为该数原码的符号位不变，其它位取反。

例如：-1==>1000 0001(原码)==>1111 1110(反码，符号位不变，其它位取反)。

（3）负数补码

》负数的补码为该数对应的无符号数的二进制取反加一。

例如：

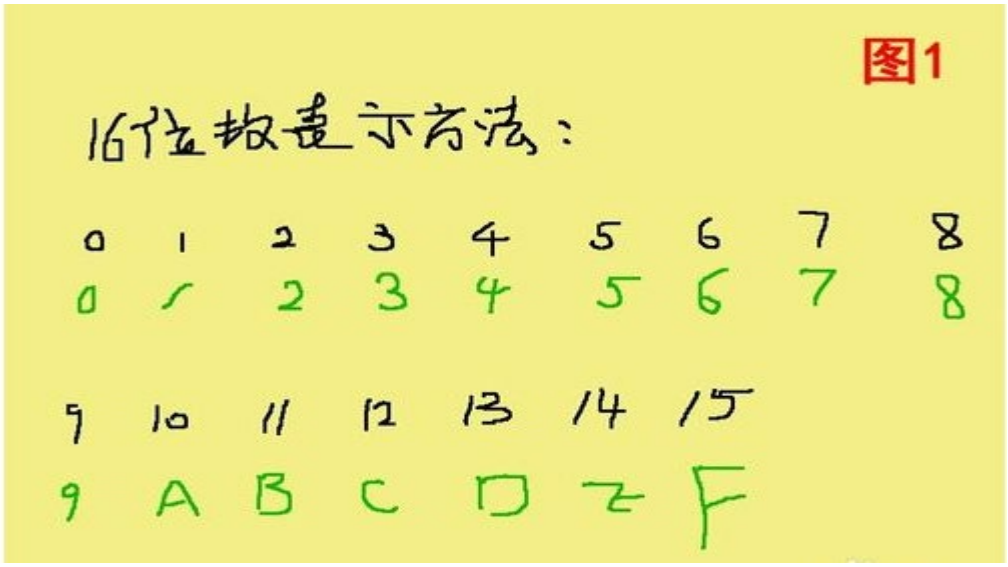
-128==>128(无符号数)==>1000 0000(-128 的无符号数的二进制)==>0111 1111(取反)==>1000 0000(补码，加 1)(负数的首位为 1)

-1==>1(无符号数)==>0000 0001(-1 对应无符号数的二进制)==>1111

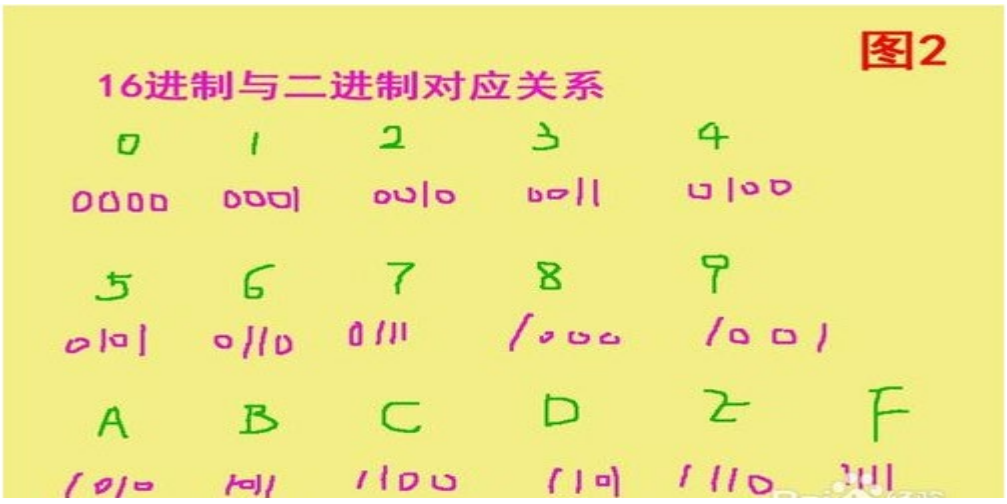
1110(取反)==>1111 1111(补码，加 1)(负数的首位为 1)

2.2.二进制与十六进制转换

1.首先呢，先要看看十六位数的表示方法，如图 1 所示。

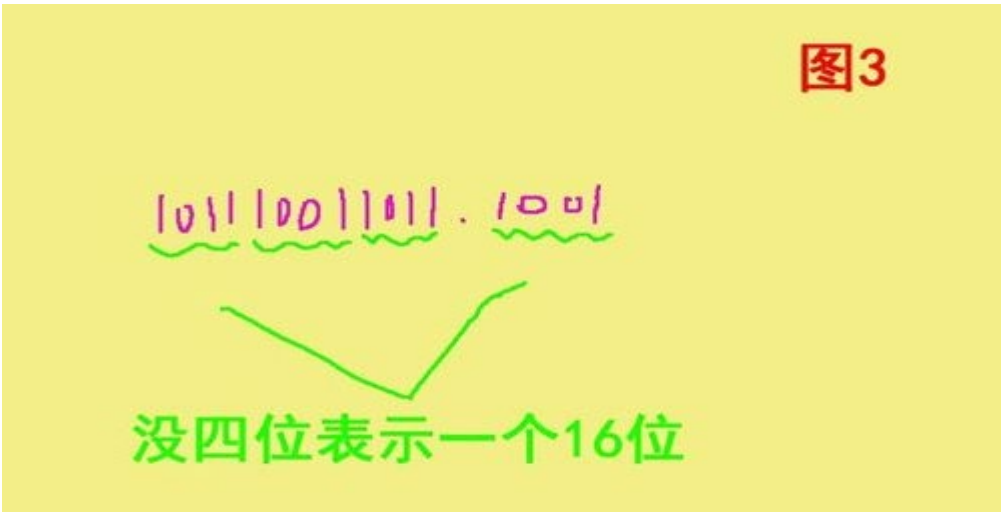


2. 再来掌握二进制数与十六进制数之间的对应关系表，如图 2 所示。只有牢牢掌握的对应关系，在转换的过程中才会事半功倍。

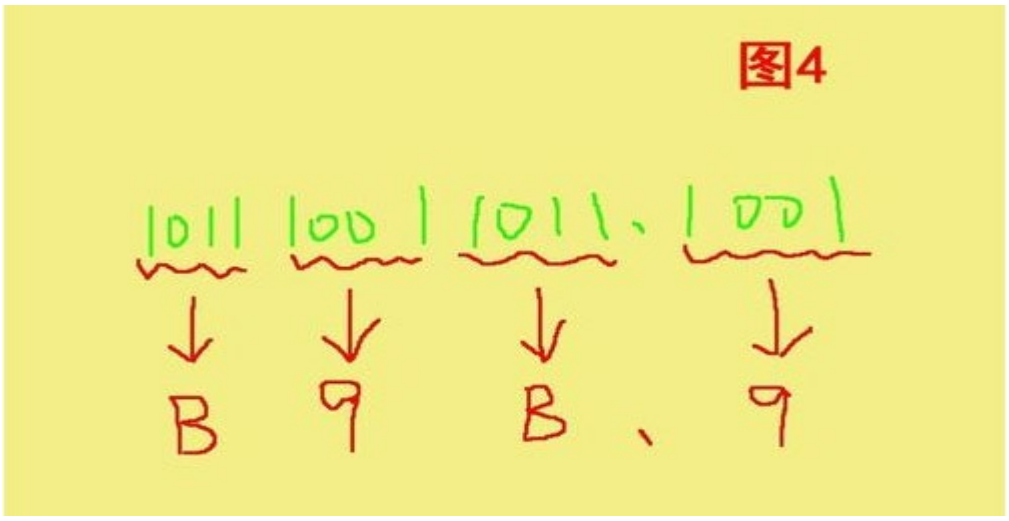


3. 二进制转换成十六进制的方法是，取四合一法，即从二进制的小数点为分界点，

向左（或向右）每四位取成一位，如图 3 所示。（注：小数点左侧向左，小数点右侧向右）

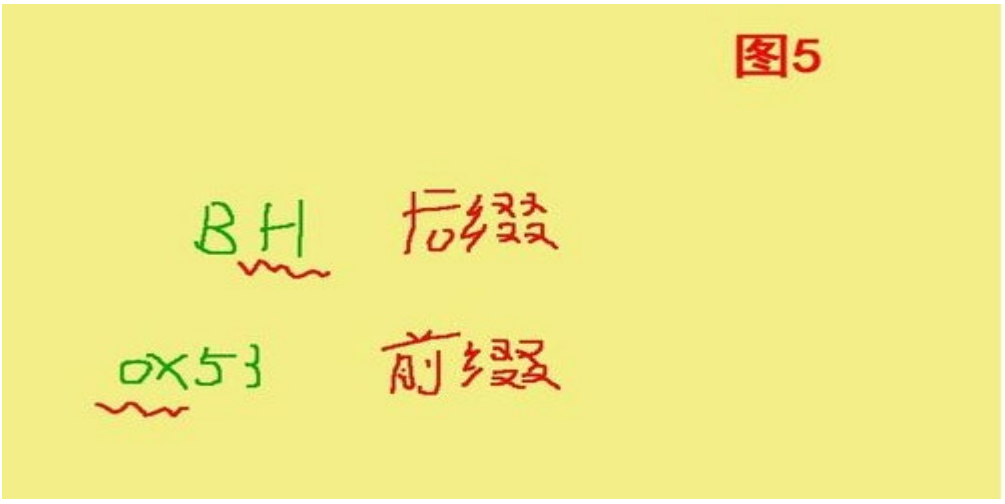


4. 组分好以后，对照二进制与十六进制数的对应表（如图 2 中所示），将四位二进制按权相加，得到的数就是一位十六进制数，然后按顺序排列，小数点的位置不变哦，最后得到的就是十六进制数哦，如图 4 所示。

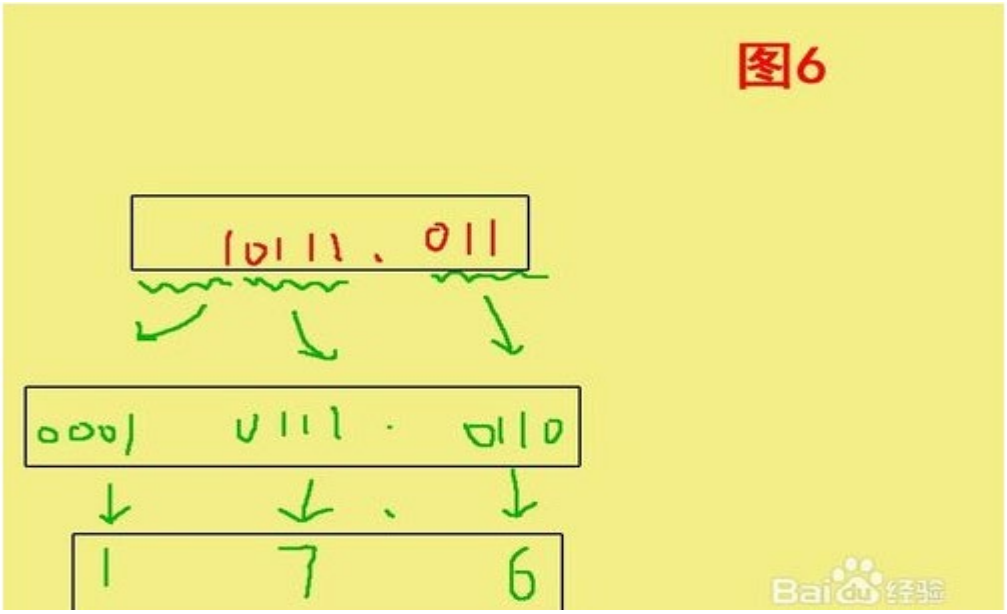


5. 注意 16 进制的表示法，用字母 H 后缀表示，比如 BH 就表示 16 进制数 11；

也可以用 0X 前缀表示，比如 0X53 就是 16 进制的 23.直观表示法如图 5 所示。

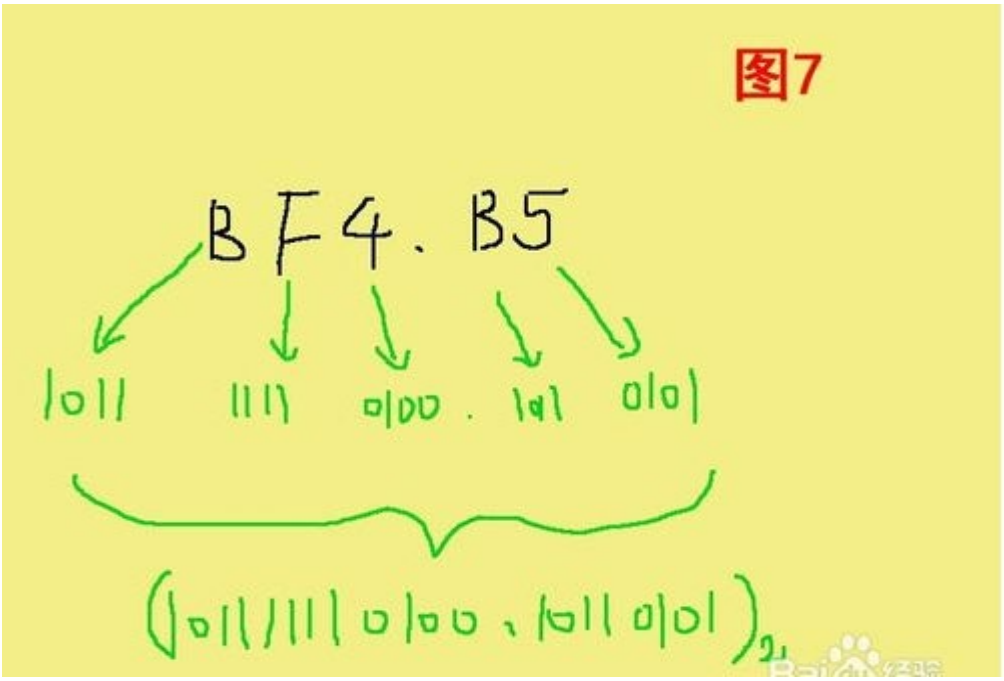


6. 这里需要注意的是，在向左（或向右）取四位时，取到最高位（最低位）如果无法凑足四位，就可以在小数点的最左边（或最右边）补 0，进行换算，如图 6 所示。



7. 下面看看将 16 进制转为二进制，反过来啦，方法就是一分四，即一个十六进制数分成四个二进制数，用四位二进制按权相加，最后得到二进制，小数点依旧

就可以啦。如图 7 所示。



2.3 二进制与八进制转换

类似二进制与十六进制的转换

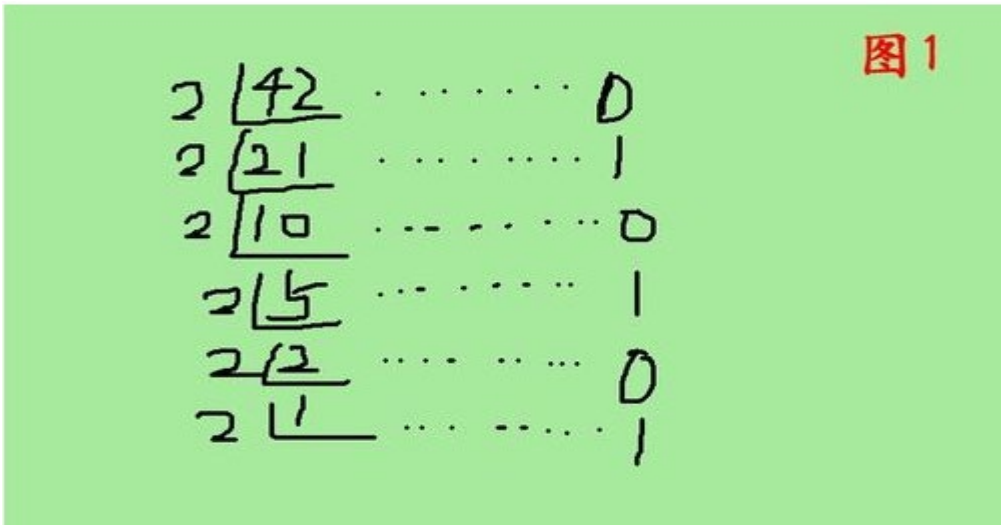
2.4 二进制与十进制转换

转成二进制主要有以下几种 :正整数转二进制, 负整数转二进制, 小数转二进制；

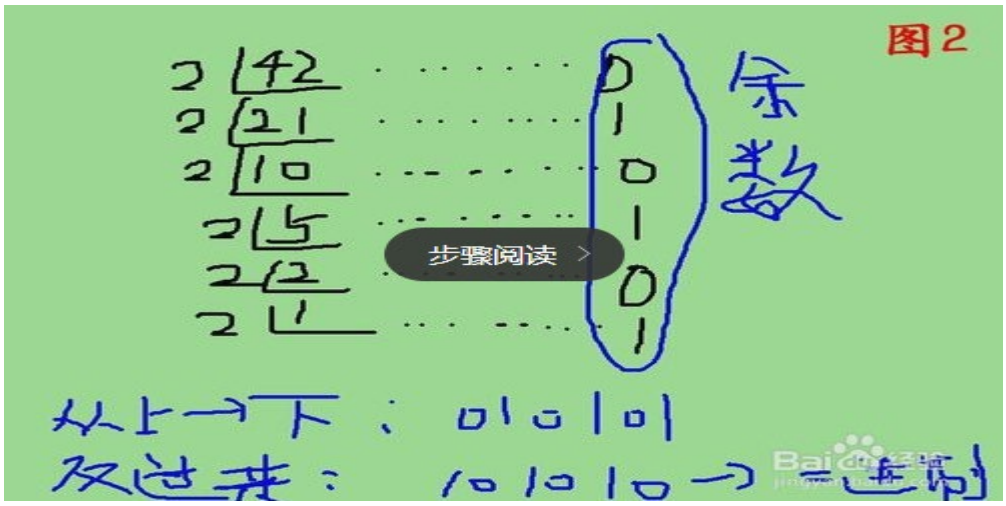
2.4.1.正整数转成二进制

基本法则：除二取余，然后倒序排列，高位补零。

也就是说，将正的十进制数除以二，得到的商再除以二，依次类推知道商为零或一时为止，然后在旁边标出各步的余数，最后倒着写出来，高位补零就 OK 咧。哎呀，还是举例说明吧，比如 42 转换为二进制，如图 1 所示操作。



42 除以 2 得到的余数分别为 010101，然后咱们倒着排一下，42 所对应二进制就是 101010.如图 2 所示更直观的表达。



计算机内部表示数的字节单位是定长的，1B=8bit，2B=16bit…。所以，位数不够时，高位补零，所说，如图 3 所示，42 转换成二进制以后就是。00101010，也即规范的写法为（42）10=（00101010）2。

2.4.2.负整数转换成二进制

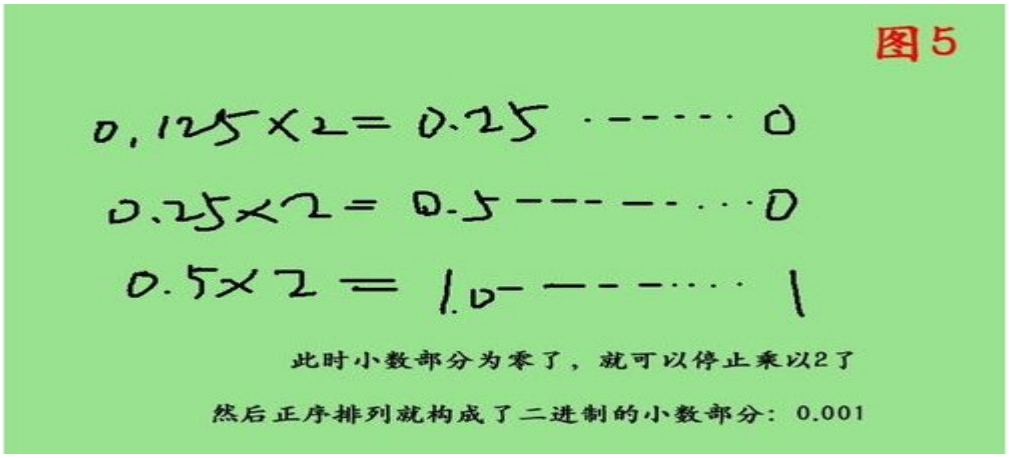
因为在有符号存储中，计算机存储的是补码，**所以此处我们转化的二进制实际上是负整数的补码！！**

方法：先是将对应的无符号整数转换成二进制后，对二进制取反，然后对结果再加 1。

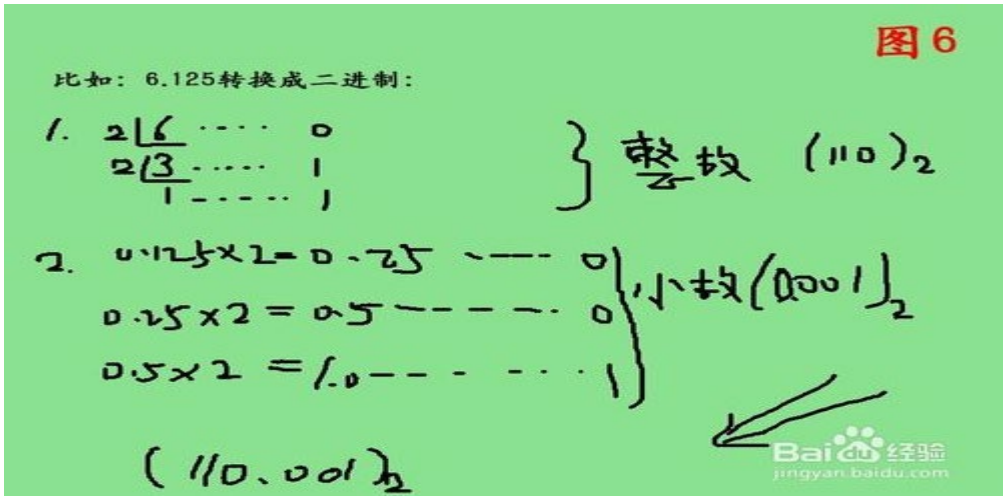
还以 42 为例，负整数就是-42，如图 4 所示为方法解释。最后即为：(-42) 10=（ 11010110）2.

2.4.3.小数转换为二进制的方法

对小数点以后的数乘以 2，取结果的整数部分（不是 1 就是 0 喽），然后再用小数部分再乘以 2，再取结果的整数部分……以此类推，**直到小数部分为 0 或者位数已经够了就 OK 了**。然后把取的整数部分按先后次序排列就 OK 了，就构成了二进制小数部分的序列，举个例子吧，比如 0.125，如图 5 所示。



如果小数的整数部分有大于 0 的整数时该如何转换呢？如以上整数转换成二进制，小数转换成二进制，然后加在一起就 OK 了，如图 6 所示。



2.4.4. 正整数二进制转换为十进制

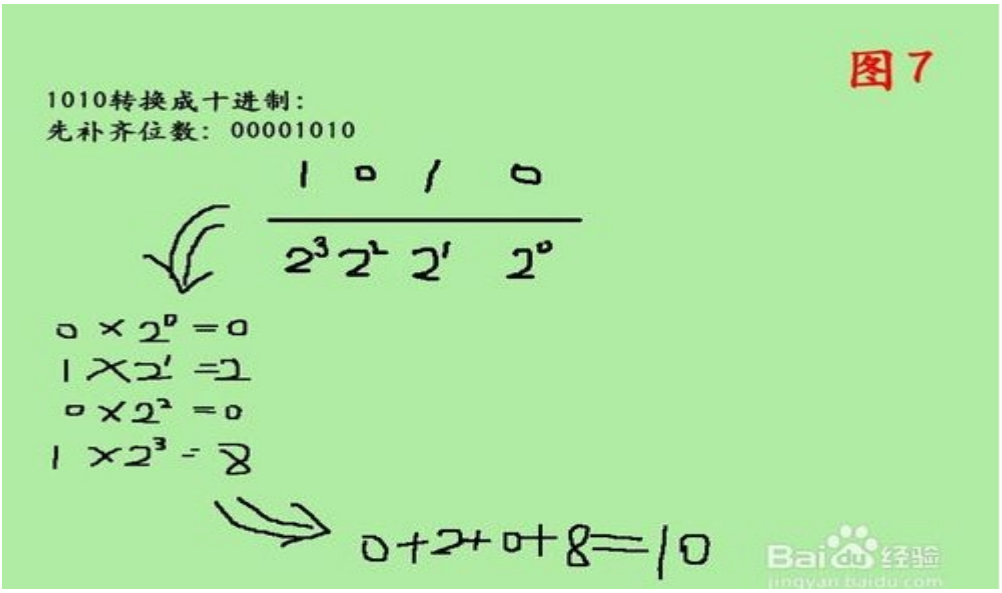
判断一个二进制是正数还是负数，要先看其在计算机中是**以有符号进行存储还是无符号进行存储**。

》如果是无符号存储，则为原码存储，其为一个正数。

》若是有符号存储，则为补码存储。看其最高位，最高位为 0，为正数，反之，为负数。

正数的原码=反码=补码。负数的补码是原码的反码再加 1

一个整数按照绝对值大小转换成的二进制数，是为原码。



2.4.5.负整数二进制转换为十进制

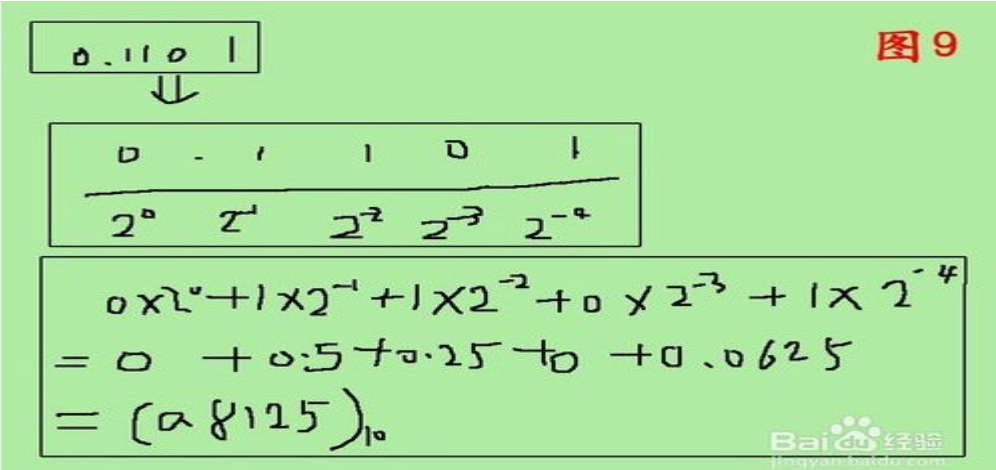
一个负数的二进制转换为十进制就是上面 3.小数转换为二进制的方法的逆过程

注：在计算机中，正数与负数的存储都是以补码形式存储，例如 10000001 表示 -127 的补码。

2.4.6.小数部分二进制转换为十进制

将有小数的二进制转换为十进制时：例如 0.1101 转换为十进制的方法：将

二进制中的四位数分别于下边（如图 9 所示）对应的值相乘后相加得到的值即为换算后的十进制。



3 区块链基础--对称秘钥与非对称秘钥

非对称加密解决了电子货币的所有权问题！

3.1 基本概念

北京的 Bob 发了一个快递到广州的 Alice，途中经过了上海，上海快递中心出现了一个黑客 H，他偷偷打开了 Bob 给 Alice 的快递，然后偷偷把里边的衣服剪烂，再按照原样包装好发往广州，可以看到对于这样简单包装的传输在中途是可以偷偷修改里边的东西。

HTTP 的数据包是明文传输，如果中途某个黑客嗅探到这个 HTTP 包，他可以偷偷修改里边包的内容，至于 Bob 跟 Alice 是互相不知道这个动作的，因此我们必须要有一个方案来防止这种不安全的篡改行为，有个方法就是加密！

（1）对称加密

采用**单钥密码系统**的加密方法，同一个密钥可以同时用作信息的加密和解密，这种加密方法称为对称加密。所谓对称就是**加密和解密的过程使用的是相同的密钥.**

例如 A 用户新建了一个文档只想自己和 B 用户看,于是 A 为这个文档设置了密码(加密)，然后再发送给 B.用户 B 只要知道文档的密码就可以查看其中内容(解密).别人即使拿到文档想看,但没有密码(密钥)也无从查看里面的内容.

当然对称加密也存在严重的秘钥泄露问题，上例中用户 A 必须通过某种方式（打电话、邮件等）告诉用户 B 密码，但这个过程中密码就可能泄露。

（2）非对称加密

最早在 1976 年由美国学者 Dime 和 Henman 提出。他们为解决信息公开传送和密钥管理问题，提出一种新的密钥交换协议，允许在不安全的媒体上的通讯双方交换信息，安全地达成一致的密钥，这就是“公开密钥系统”。

与对称加密不同,非对称加密算法的**加密和解密**使用**不同的两个密钥**.这两个密钥就是我们经常听到的**"公开密钥"(公钥)和"私有密钥"(私钥).**

公钥和私钥的关系是:

（1）公钥和私钥成对出现,

（2）如果你的消息使用公钥加密,那么需要该公钥对应的私钥才能解密;

（3）如果你的消息使用私钥加密,那么需要该私钥对应的公钥才能解密.

例如：用户 A 和用户 B 互发邮件,并且邮件内容需要保密.用户 A 和用户 B 采用非对称加密方式，这样用户 A 和用户 B 都各自有一把公钥和一把私钥。并且用户 A 和用户 B 可以通过公开的方式告知对方自己的公钥，各自的私钥都保存在自己的手中，不让外人知道。当用户 A 给用户 B 发信息时，可以用用户 B 的公钥加

密邮件内容发送给 B，用户 B 用自己的私钥解密邮件，就可以看到邮件内容了。当然，用户 B 给用户 A 发送消息，用用户 A 的公钥加密邮件内容，用户 A 用自己的私钥就可以解密邮件。这个过程中私钥都在自己的手里，可以有效的避免泄露。

3.2 非对称加密案例

在非对称加密中使用的主要算法有：RSA、Elgamal、背包算法、Rabin、D-H、ECC（椭圆曲线加密算法）等。为了更加具体形象的理解公私钥，我们接下来通过一个案例来描述，以 RSA 为例。

假设信息传递的双方是 A 和 B。假设 B 想让 A 传递加密信息给自己，就是 A 传递信息给 B。使用非对称加密方式为 RSA。步骤如下：

（一）登录可以生成非对称秘钥的网址

http://web.chacuo.net/netrsakeypair

（二）选择 RSA 密钥对，如图点“生成密钥对”，密钥的长度选 4096，密钥长度越长，破解难度就越高。点完之后，分别把下面两个大文本框内的公开密钥（第一个方框内的全部内容，图上公钥 1）和私有密钥（第二个方框的全部内容，图上私钥 1）拷贝或者保存到一个稳妥的地方。

（三）B 把公钥 1（上图中的简称公钥 1）用任何方式（邮件，QQ，微信）发给 A。

（四）A 拿到了公钥 1 之后，就可以把自己想给 B 的信息，来进行加密了。点击左侧 RSA 公钥加密解密，复制从 B 处得到的公钥 1 的内容，添加需要加密给 B 的文本内容，点击最后的 RSA 公钥加密，生成加密文本。

（五）A 把加密完后生成的加密文本，用公开途径发给 B，A 的情报传递工作就完成了。

解密流程：B 如何解密 A 的情报内容呢？请看下面的操作。

（六）点击左侧 RSA 私钥加密解密 B 首先把最早生成保存好的私钥 1 复制到第一个方框内，然后把 A 公开发过来的加密后文本复制到第二个方框内，点击 RSA 私钥解密，最后生成了 A 加密的文字内容：

4 区块链基础--数字签名与数字证书

4.1 基本概念

数字签名是用于确定信息的所有者和信息的完整性 ;数字证书用于确保公钥的所有者的真实性。

以比特币来说明数字签名 :数字签名就是只有比特币转账中转出的人才能生成的一段防伪造的字符串。通过验证该数字串，一方面证明交易是转出方本人发起的，另一方面证明交易信息在传输中没有被篡改。数字签名由数字摘要和非对称加密技术组成。

（1）通过数字摘要技术，把交易信息缩短成固定长度的字符串，然后用自己的私钥对摘要进行加密，形成数字签名。完成后需要将完整交易信息和数字签名一起广播给矿工。矿工用牛牛的公钥进行验证，如果验证成功，说明该笔交易确实是牛牛发出的，且信息未被更改。

（2）非对称加密技术是指数字签名的私钥和解密的公钥不一致

我们通过以下案例来解释（具体可以参考文献[2]），案例 1-9 步中描述了数字签名；案例 10-13 阐述了数字证书。

（1）非对称加密：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。

（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。

（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，生成信件的数据摘要。

（6）然后，鲍勃使用私钥，对这个数据摘要加密，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，也就是保证了数据的完整性。

（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。

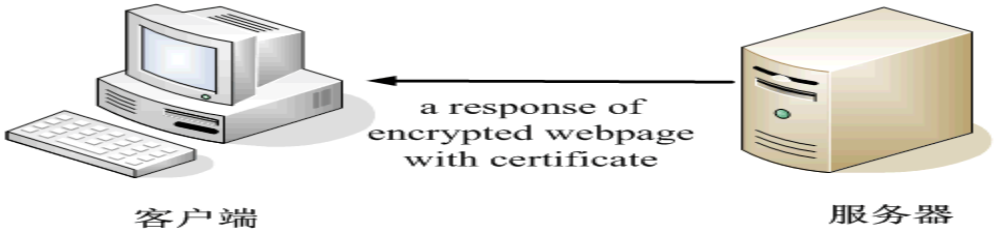
（13）苏珊收信后，用 CA 的公钥解开数字证书，就可以拿到鲍勃真实的公钥了，然后就能证明"数字签名"是否真的是鲍勃签的。

4.2 具体案例-https

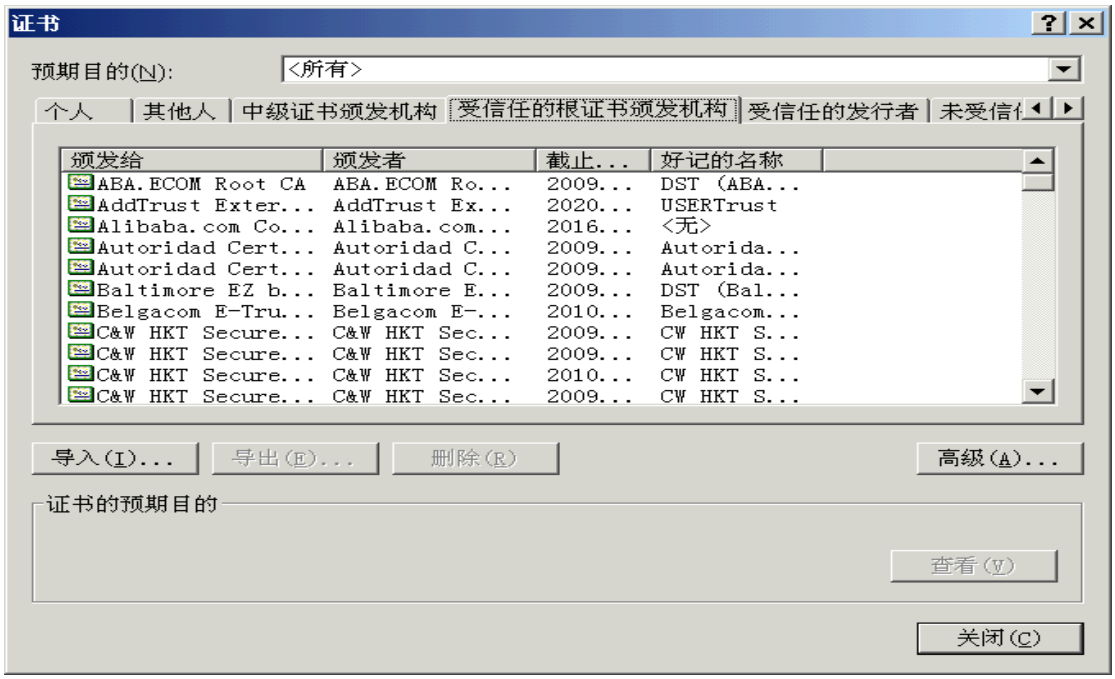
接下来我们以 https 为例再看看数字证书的使用，这个协议主要用于网页加密：

（1）首先，客户端向服务器发出加密请求。

（2）服务器用自己的私钥加密网页以后，连同本身的数字证书，一起发送给客户端。

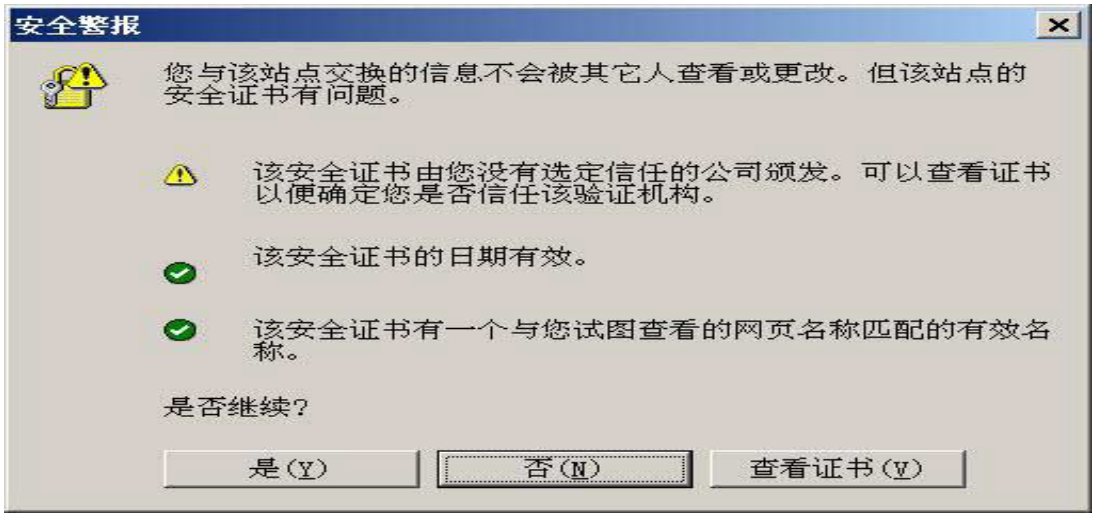


（3）客户端（浏览器）的"证书管理器"，有"受信任的根证书颁发机构"列表。客户端会根据这张列表，查看解开数字证书的公钥是否在列表之内。

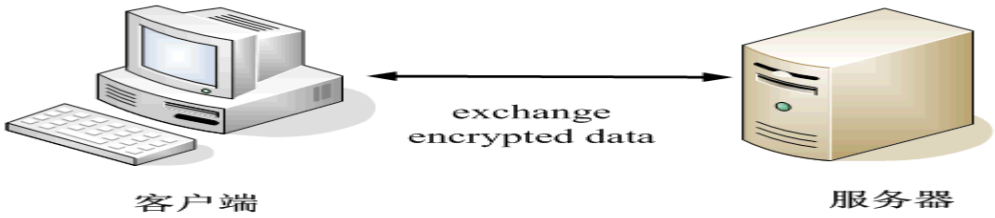


（4）如果数字证书记载的网址，与你正在浏览的网址不一致，就说明这张证书可能被冒用，浏览器会发出警告。

(5)如果这张数字证书不是由受信任的机构颁发的，浏览器会发出另一种警告。



（6）如果数字证书是可靠的，客户端就可以使用证书中的服务器公钥，对信息进行加密，然后与服务器交换加密信息。



5 区块链基础--P2P 网络

比特币、以太坊等众多数字货币都实现了属于自己的 P2P 网络协议，P2P 网络模块作为所有区块链的最底层模块，直接决定了整个区块链网络的稳定性。今天讨论的重点主要是区块链技术的 P2P 技术，也就是比特币和以太坊的 P2P 网络。由于区块链的 P2P 网络技术知识繁多，我们主要提炼其中的四个内容进行讲解：区块链的网络连接与拓扑结构、节点发现、局域网穿透与节点交互协议。

5.1 网络连接与拓扑结构

（1）网络连接

除去少数支持 UDP 协议的区块链项目外，绝大部分的区块链项目所使用的底层网络协议依然是 TCP/IP 协议。所以从网络协议的角度来看，区块链其实是基于 **TCP/IP 网络协议**的，这与 HTTP 协议、SMTP 协议是处在同一层，也就是应用层。以 HTTP 协议为代表的、与服务端的交互模式在区块链上被彻底打破了，变更为完全的点对点拓扑结构，这也是以太坊提出的 Web3.0 的由来。

比特币的 P2P 网络是一个非常复杂的结构，考虑到矿池内部的挖矿交互协议与轻节点。我们仅仅讨论全节点这种场景下的 P2P 网络发现与路由。

比特币的 P2P 网络基于 TCP 构建，主网默认通信端口为 8333。以太坊的 P2P 网络则与比特币不太相同，以太坊 P2P 网络是一个完全加密的网络，提供 UDP 和 TCP 两种连接方式，主网默认 TCP 通信端口是 30303，推荐的 UDP 发现端口为 30301。

（2）拓扑结构

P2P 网络拓扑结构有很多种，有些是中心化拓扑，有些是半中心化拓扑，有些是全分布式拓扑结构。比特币全节点组成的网络是一种全分布式的拓扑结构，节点与节点之间的传输过程更接近“泛洪算法”，即：交易从某个节点产生，接着广播到临近节点，临近节点一传十十传百，直至传播到全网。全节点与 SPV 简化支付验证客户端之间的交互模式，更接近半中心化的拓扑结构，也就是 SPV 节点可以随机选择一个全节点进行连接，这个全节点会成为 SPV 节点的代理，帮助 SPV 节点广播交易。

5.2 节点发现

节点发现是任何区块链节点接入区块链 P2P 网络的第一步。这与你孤身一人去陌生地方旅游一样，如果没有地图和导航，那你只能拽附近的人问路，“拽附近的人问路”的这个动作就可以理解成节点发现。

节点发现可分为初始节点发现，和启动后节点发现。初始节点发现就是说你的全节点是刚下载的，第一次运行，什么节点数据都没有。启动后节点发现表示正在运行的钱包已经能跟随网络动态维护可用节点。

（1）初始节点发现

在比特币网络中，初始节点发现一共有两种方式。第一种叫做 DNS-seed，又称 DNS 种子节点，DNS 就是中心化域名查询服务，比特币的社区维护者会维护一些域名。比如 seed.bitcoin.sipa.be 这个域名就是由比特币的核心开发者 Sipa 维护的，如果我们通过 nslookup 会发现大约二十多个 A 纪录的 IPv4 主机地址。我们通过 nc 命令尝试连接域名下的某个主机的 8333 端口会发现连接成功，运行结构如下。

```
✗ chenhao@chenhaodeMacBook-Pro  ~  nc -nvv 149.202.179.35 8333
found 0 associations
found 1 connections:
     1: flags=82<CONNECTED,PREFERRED>
 outif en0
 src 192.168.1.104 port 62125
 dst 149.202.179.35 port 8333
 rank info not available
 TCP aux info available
Connection to 149.202.179.35 port 8333 [tcp/*] succeeded!
```

好的，到目前为止我们已经手动做了一遍初始节点发现的工作，这些操作是由比特币的代码完成的。

第二种方式就是，代码中硬编码（ hard-code ）了一些地址，这些地址我

们称之为种子节点（seed-node），当所有的 DNS 种子节点全部失效时，全节点会尝试连接这些种子节点。用在以太坊中，思路也大致相同，也是在代码中硬编码（hard-code）了一些种子节点做类似的工作。

（2）启动后节点发现

在 Bitcoin 的网络中，一个节点可以将自己维护的对等节点列表(peer list)发送给临近节点，所以在初始节点发现之后，你的节点要做的第一件事情就是向对方要列表："快把你的节点列表给我复制一份。"所以在每次需要发送协议消息的时候，它会花费固定的时间尝试和已存的节点列表中的节点建立链接，如果有任何一个节点在超时之前可以连接上，就不用去 DNS seed 获取地址，一般来说，这种可能性很小，尤其是全节点数目非常多的情况下。

而在以太坊网络中，也会维护类似的一个节点列表(NodeTable)，但是这个节点列表与比特币的简单维护不同，它采用了 P2P 网络协议中一个成熟的算法，叫做 Kademlia 网络，简称 KAD 网络。它使用了 DHT 来定位资源，全称 Distributed Hash Table，中文名为分布式哈希表。KAD 网络会维护一个路由表，用于快速定位目标节点。由于 KAD 网络基于 UDP 通信协议，所以以太坊节点的节点发现是基于 UDP 的，如果找到节点以后，数据交互又会切换到 TCP 协议上。

（3）黑名单与长连接

公有区块链面临的网络环境是非常开放的，任何人只要下载好钱包，打开运行就进入了这个 P2P 网络，这也会带来被攻击的可能。所以在比特币的代码中，会有一段去控制逻辑，你可以手动将你认为可疑的节点移除并加入禁止列表，同时去配置可信的节点。当然，以上并不属于客户端的标准协议的一部分，任何人都可以实现属于自己的 P2P 网络层。以太坊上有针对账户进行的黑名单处理，

但是这属于业务层。我没有找到很详尽的资料，所以你有兴趣的话，可以自己尝试一下。不过总的来说，黑名单我们也可以通过操作系统的防火墙去处理，这并不算一个特别棘手的问题。

5.3 局域网穿透

前面我们说到了区块链的 P2P 网络结构是一种全分布式的拓扑结构。但是，如今我们的网络环境是由局域网和互联网组成的。也就是说，当你在局域网运行一个区块链节点，在公网是发现不了的，公网上的节点只能被动接受连接，并不能主动发起连接。

如果这个局域网是你可以控制的，那么很好说，咱们只需要在 VPC 网络中配置路由，将公网 IP 和端口映射到局域网中你的 IP 和端口即可。这个条件是非常苛刻的，那么到底有没有一种方案可以自行建立映射呢？答案是：有，就是 NAT 技术和 UPnP 协议。

NAT 技术非常常见，这里使用的是源 NAT，简而言之就是替换 TCP 报文中的源地址并映射到共网地址。

UPnP 是通用即插即用（Universal Plug and Play）的缩写，它主要用于设备的智能互联互通，所有在网络上的设备马上就能知道有新设备加入。

这些设备彼此之间能互相通信，更能直接使用或者控制它，一切都不需要人工设置。有关 UPnP 的资料比较多，这里就不赘述了，你可以自行搜索相关的信息。

比特币和以太坊均使用了 UPnP 协议作为局域网穿透工具，只要局域网中的路由设备支持 NAT 网关功能、支持 UPnP 协议，即可将你的区块链节点自动映

射到公网上。

5.4 节点交互协议

一旦节点建立连接以后，节点之间的交互是遵循一些特定的命令，这些命令写在消息的头部，消息体写的则是消息内容。命令分为两种，一种是请求命令，一种是数据交互命令。

节点连接完成要做的第一件事情叫做握手操作。这一点在比特币和以太坊上的流程是差不多的，就是相互问候一下，提供一些简要信息。比如先交换一下版本号，看看是否兼容。只是以太坊为握手过程提供了对称加密，而比特币没有。握手完毕之后，无论交互什么信息，都是需要保持长连接的，在比特币上有 PING/PONG 这两种类型的消息，这很明显就是用于保持节点之间长连接的心跳而设计的 ;而在以太坊的设计中, 将 PING/PONG 协议移到了节点发现的过程中。

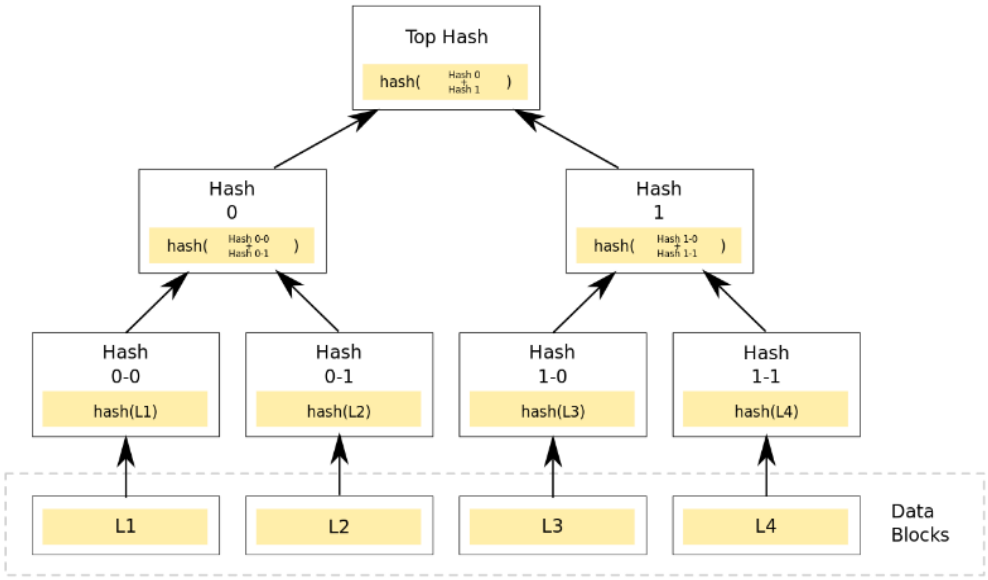
请求命令一般分为发起者请求，比如比特币中的 getaddr 命令是为了获取对方的可用节点列表，inv 命令则提供了数据传输，消息体中会包含一个数据向量。

我们说区块链最重要的功能就是同步区块链，而同步区块恰巧是最考验 P2P 网络能力的。区块同步方式分为两种，第一种叫做 HeaderFirst，它提供了区块头先同步，同步完成以后再从其他节点获得区块体。第二种叫做 BlockFirst，这种区块同步的方式比较简单粗暴，就是从其他节点获取区块必须是完整的。第一种方案提供了较好的交互过程，减轻了网络负担。这两种同步方式会直接体现在节点交互协议上，他们使用的命令逻辑完全不同。

6 区块链基础--Merkle Tree

6.1 Merkle Tree 基本概念

Merkle Tree，通常也被称作 Hash Tree，顾名思义，就是存储 hash 值的一棵树。**Merkle 树的叶子**是数据块(例如，文件或者文件的集合)的 **hash 值**。非叶节点是其对应子节点串联字符串的 hash。



在最底层，把数据分成小的数据块，有相应地哈希（hash）和它对应。但是往上走，并不是直接去运算根哈希，而是把相邻的两个哈希合并成一个字符串，然后运算这个字符串的哈希，这样得到了上一层的哈希（hash）。如果最底层的哈希总数是**单数**，那到最后必然出现一个单身哈希，这种情况就直接对它进行哈希运算，所以也能得到它的上一层哈希。于是往上推，依然是一样的方式，可以得到数目更少的新一级哈希，最终必然形成一棵倒挂的树，到了树根的这个位置，这一代就剩下一个根哈希了，我们把它叫做 **Merkle Root**。

6.2 Merkle Tree 的特点

（1）MT 是一种树，大多数是二叉树，也可以多叉树，无论是几叉树，它都具有树结构的所有特点；

（2）Merkle Tree 的叶子节点的 value 是数据集合的单元数据或者单元数据 HASH。

（3）非叶子节点的 value 是根据它下面所有的叶子节点值，然后按照 Hash 算法计算而得出的。

通常，加密的 hash 方法像 SHA-2 和 MD5 用来做 hash。但如果仅仅防止数据不是蓄意的损坏或篡改，可以改用一些安全性低但效率高的校验和算法，如 CRC。

6.3Merkle Tree 基本操作

（1）Merkle Tree 的创建

Step1：加入最底层有 9 个数据块。

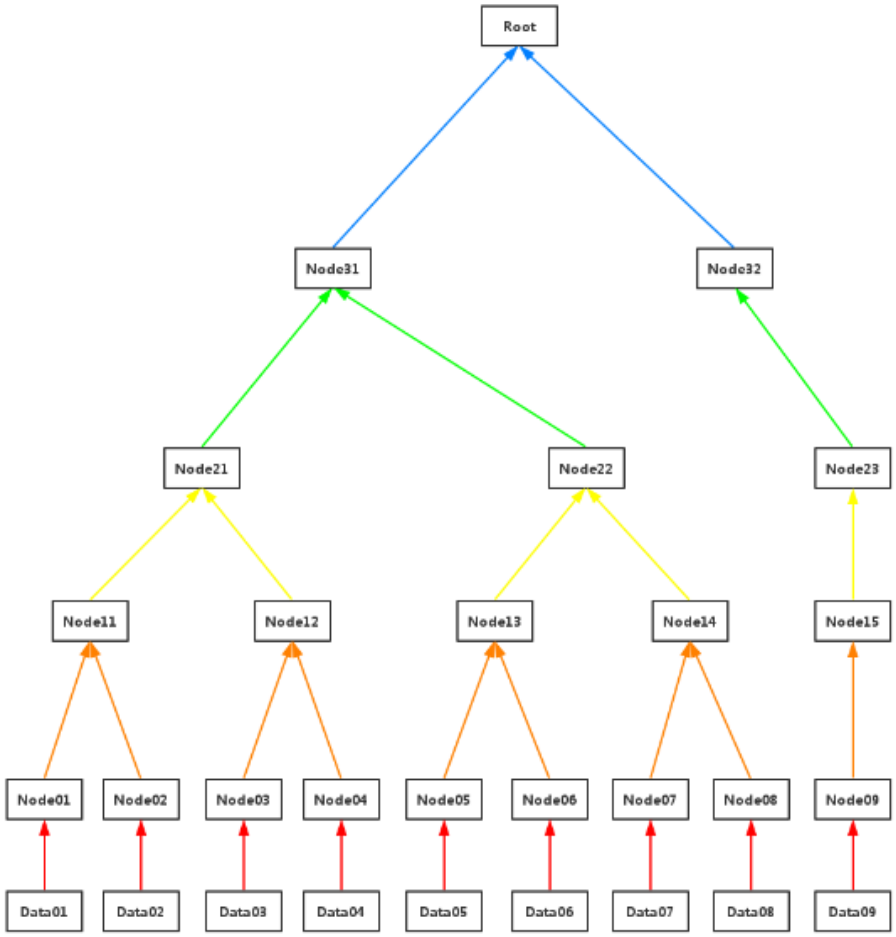
Step2：（红色线）对数据块做 hash 运算，Node0i = hash(Data0i), i=1,2,⋯,9

Step3: （橙色线）相邻两个 hash 块**串联**，然后做 hash 运算，Node1((i+1)/2) = hash(Node0i+Node0(i+1)), i=1,3,5,7;对于 i=9, Node1((i+1)/2) = hash(Node0i)

Step4: （黄色线）重复 step2

Step5：（绿色线）重复 step2

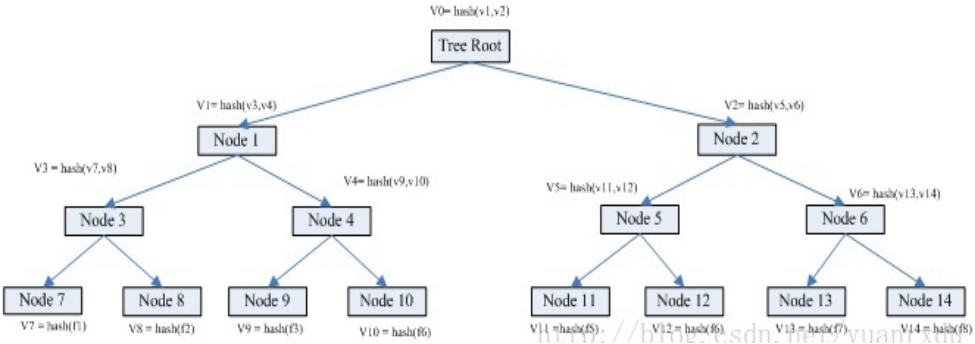
Step6：（蓝色线）重复 step2，生成 Merkle Tree Root



易得，创建 Merkle Tree 是 O(n)复杂度(这里指 O(n)次 hash 运算)，n 是数据块的数量。得到 Merkle Tree 的树高是 log(n)+1（此处的 log 以 2 为底）

（2）数据块检索

我们假设有 A 和 B 两台机器, A 与 B 都有 8 个文件, 文件分别是 f1 f2 f3 ....f8。这个时候我们就可以通过 Merkle Tree 来进行快速比较。假设我们在文件创建的时候每个机器都构建了一个 Merkle Tree。



从上图可得知，叶子节点 node7 的 value = hash(f1),是 f1 文件的 HASH;而其父亲节点 node3 的 value = hash(v7, v8)，也就是其子节点 node7 node8 的值得 HASH。就是这样表示一个层级运算关系。root 节点的 value 其实是所有叶子节点的 value 的唯一特征。

假如 A 上的文件 f5 与 B 上的 f5 不一样,其他七个文件一模一样。我们怎么通过两个机器的 merkle tree 信息找到不相同的文件？这个比较检索过程如下:

Step1. 首先比较 v0 是否相同,如果不同，检索其孩子 node1 和 node2.

Step2. v1 相同，v2 不同。检索 node2 的孩子 node5 node6;

Step3. v5 不同，v6 相同，检索比较 node5 的孩子 node 11 和 node 12

Step4. v11 不同，v12 相同。node 11 为叶子节点，获取其目录信息。

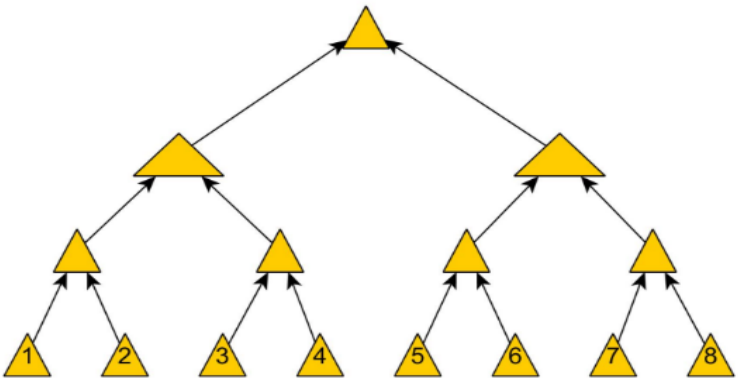
Step5. 检索比较完毕。

以上过程的理论复杂度是 Log(N)。

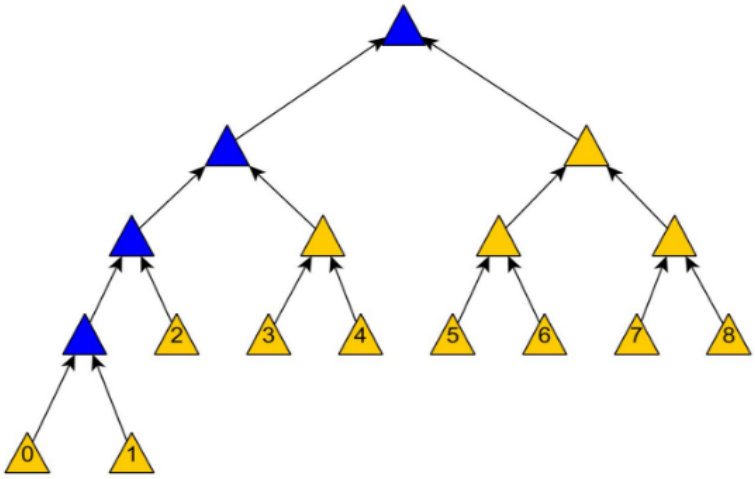
(3) 更新、插入及删除

对于 Merkle Tree 数据块的更新操作其实是很简单的，更新完数据块，然后接着更新其到树根路径上的 Hash 值就可以了，这样不会改变 Merkle Tree 的结构。但是，插入和删除操作肯定会改变 Merkle Tree 的结构，如下图，一种插入

操作是这样的：



插入数据块0后(考虑数据块的位置)，Merkle Tree的结构是这样的：



而文献[7]中的同学在考虑一种插入的算法，满足下面条件：

1)re-hashing 操作的次数控制在 log(n)以内

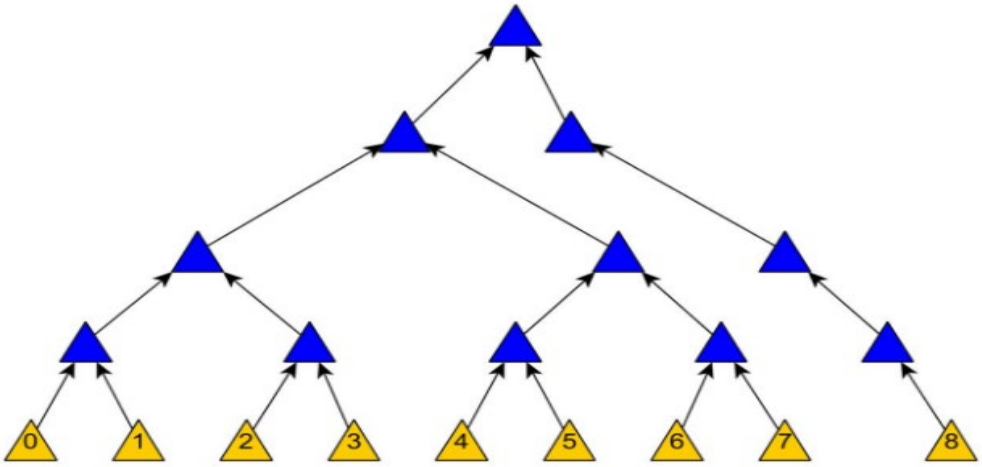
2)数据块的校验在 log(n)+1 以内

3)除非原始树的 n 是偶数，插入数据后的树没有孤儿，并且如果有孤儿，那么孤儿是最后一个数据块

4)数据块的顺序保持一致

5)插入后的 Merkle Tree 保持平衡

然后上面的插入结果就会变成这样：

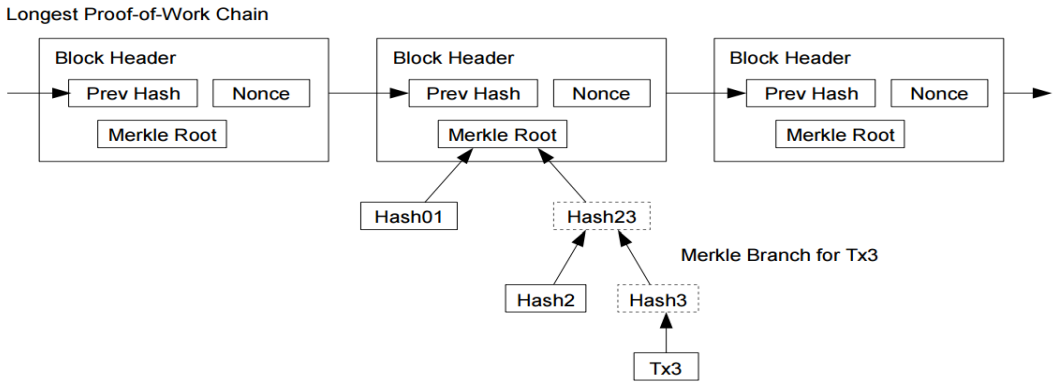


根据文献[7]中回答者所说，Merkle Tree 的插入和删除操作其实是一个工程上的问题，不同问题会有不同的插入方法。如果要确保树是平衡的或者是树高是 log(n)的，可以用任何的标准的平衡二叉树的模式，如 AVL 树，红黑树，伸展树，2-3 树等。这些平衡二叉树的更新模式可以在 O(lgn)时间内完成插入操作，并且能保证树高是 O(lgn)的。那么很容易可以看出更新所有的 Merkle Hash 可以在 O((lgn)2)时间内完成（对于每个节点如要更新从它到树根 O(lgn)个节点，而为了满足树高的要求需要更新 O(lgn)个节点）。如果仔细分析的话，更新所有的 hash 实际上可以在 O(lgn)时间内完成，因为要改变的所有节点都是相关联的，即他们要不是都在从某个叶节点到树根的一条路径上，或者这种情况相近。

文献[7]的回答者说实际上 Merkle Tree 的结构(是否平衡，树高限制多少)在大多数应用中并不重要，而且保持数据块的顺序也在大多数应用中也不需要。因此，可以根据具体应用的情况，设计自己的插入和删除操作。一个通用的 Merkle Tree 插入删除操作是没有意义的。**具体参考文献[7]**

6.4Merkle Tree 具体应用

（1）bitcoin



Merkle tree 最早的应用是 Bitcoin，它是由中本聪在 2009 年描述并创建的。Bitcoin 的 Blockchain 利用 Merkle tree 来存储每个区块的交易。而这样做的好处，也就是中本聪描述到的“简化支付验证”（Simplified Payment Verification，SPV）的概念:一个“轻客户端”（light client）可以仅下载链的区块头即每个区块中的 80byte 的数据块，包含六个元素，而不是下载每一笔交易以及每一个区块。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	目标值。可以推算出难度值，用于验证 block hash 难度是否达标
4	Nonce	随机数。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

如果客户端想要确认一个交易的状态，它只需简单的发起一个 Merkle proof 请求，这个请求显示出这个特定的交易在 Merkle trees 的一个之中，而且这个 Merkle Tree 的树根在主链的一个区块头中。但是 Bitcoin 的轻客户端有它的局限。一个局限是，尽管它可以证明包含的交易，但是它不能进行涉及当前状态的证明（如数字资产的持有，名称注册，金融合约的状态等）。Bitcoin 如何查询你当前有多少币？一个比特币轻客户端，可以使用一种协议，它涉及查询多个节点，并相信其中至少会有一个节点会通知你，关于你的地址中任何特定的交易支出，而这可以让你实现更多的应用。但对于其他更为复杂的应用而言，这些远远是不够的。一笔交易影响的确切性质（precise nature），可以取决于此前的几笔交易，而这些交易本身则依赖于更为前面的交易，所以最终你可以验证整个链上的每一笔交易。为了解决这个问题，Ethereum 的 Merkle Tree 的概念，会更进一步。

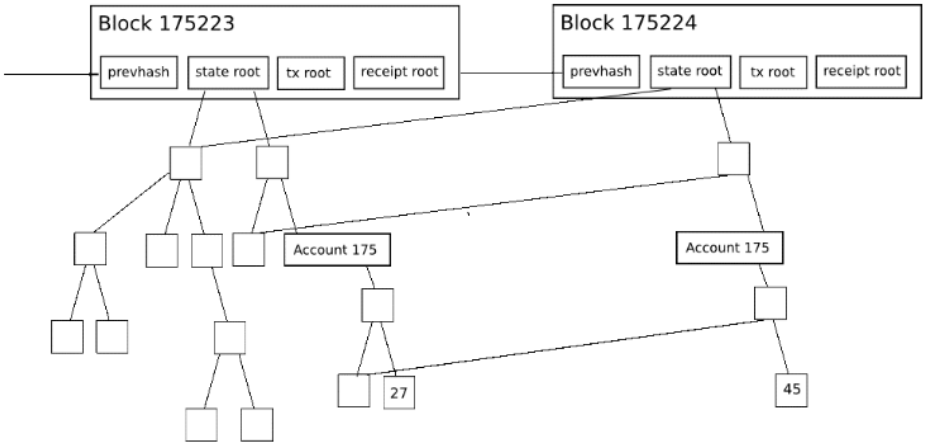
（2）Ethereum

每个以太坊区块头不是包括一个 Merkle 树，而是为三种对象设计的三棵树：

1）交易 Transaction

2）收据 Receipts(本质上是显示每个交易影响的多块数据)

3）状态 State



这使得一个非常先进的轻客户端协议成为了可能，它允许轻客户端轻松地进行并核实以下类型的查询答案：

1）这笔交易被包含在特定的区块中了么？

2）告诉我这个地址在过去 30 天中，发出 X 类型事件的所有实例（例如，一个众筹合约完成了它的目标）

3）目前我的账户余额是多少？

4）这个账户是否存在？

5）假如在这个合约中运行这笔交易，它的输出会是什么？

第一种是由交易树（transaction tree）来处理的；**第三和第四**种则是由状态树（state tree）负责处理，**第二种**则由收据树（receipt tree）处理。计算前四个查询任务是相当简单的。服务器简单地找到对象，获取 Merkle 分支，并通过分支来回复轻客户端。**第五种**查询任务同样也是由状态树处理，但它的计算方式会比较复杂。这里，我们需要构建一个 Merkle 状态转变证明（Merkle state transition proof）。从本质上来讲，这样的证明也就是在说“如果你在根 S 的状态树上运行交易 T，其结果状态树将是根为 S'，log 为 L，输出为 O”（“输出”作为存在于以

太坊的一种概念，因为每一笔交易都是一个函数调用 ;它在理论上并不是必要的)。

为了推断这个证明，服务器在本地创建了一个假的区块，将状态设为 S，并在请求这笔交易时假装是一个轻客户端。也就是说，如果请求这笔交易的过程，需要客户端确定一个账户的余额，这个轻客户端(由服务器模拟的)会发出一个余额查询请求。如果需要轻客户端在特点某个合约的存储中查询特定的条目，这个轻客户端就会发出这样的请求。也就是说服务器(通过模拟一个轻客户端)正确回应所有自己的请求，但服务器也会跟踪它所有发回的数据。然后，服务器从上述的这些请求中把数据合并并把数据以一个证明的方式发送给客户端。然后，客户端会进行相同的步骤，但会将服务器提供的证明作为一个数据库来使用。如果客户端进行步骤的结果和服务器提供的是一样的话，客户端就接受这个证明。

7 区块链基础--零知识证明

7.1 零知识证明定义

20 世纪 80 年代初，S.Goldwasser、S.Micali 以及 C.Rackoff 提出零知识证明（Zero-Knowledge Proof）概念。零知识证明是指被验证者能够在不向验证者提供任何有用的信息的情况下，使验证者相信某个论断是正确的。零知识证明实质上是一种涉及两方或更多方的协议，即两方或更多方完成一项任务所需采取的一系列步骤。零知识证明技术进一步增强了隐私性。

7.2 零知识证明基本原理

我们知道比特币其实是伪匿名的，因为比特币全网数据都是公开透明的，只要知道一个比特币地址，我们就可以通过区块链浏览器来查看完整的交易信息。

大零币（ZEC）是基于零知识证明技术来实现完全的隐私性，下面我们基于大零币来讲述一下零知识证明基本原理。

假设用户 Alice 需要向 Bob 转账 1 个 ZEC，基于零知识证明操作过程：

（1）用户 Alice 会先将自己的这 1 个 ZEC 拆分成若干份，具体份数可以根据设置来定。

（2）大零币的公有链同时也会将其他交易输出与 Alice 的若干份 ZEC 进行混合拆分，最后从中取出合计为 1 个 ZEC 的若干份发送给 Bob 的收款地址。

经过这条公有链一系列的“混币”过程，就使得包括交易地址和具体金额在内的交易信息具有很强的隐匿性。另外，从上述过程来看要实现匿名性，其所花费的计算资源就非常多，带来了大量的资源浪费，也导致了其可扩展性面临巨大挑战。当然匿名性会带来监管问题，将会给追踪与监管带来非常大的挑战，造成一系列社会问题。

7.3 阿里巴巴的零知识证明例子

网上有一个关于理解零知识证明的例子，即“阿里巴巴的零知识证明”，可以帮助我们理解“零知识证明”的原理。

一天，阿里巴巴被强盗抓住了，强盗向阿里巴巴拷问进入山洞的咒语。面对强盗，阿里巴巴是这么想的：如果我把咒语告诉了他们，他们就会认为我没有价值了，就会杀了我省粮食；但如果我死活不说，他们也会认为我没有价值而杀了我。怎样才能做到既让他们确信我知道咒语，但又一丁点咒语内容也不泄露给他们呢？

这的确是一个令人纠结的问题，但阿里巴巴想了一个好办法，当强盗向他拷

问打开山洞石门的咒语时，他对强盗说："你们在离开我一箭远的地方，用弓箭指着我，当你们举起右手我就念咒语打开石门，举起左手我就念咒语关上石门，如果我做不到或逃跑，你们就用弓箭射死我。"

强盗们当然会同意，因为这个方案不仅对他们没有任何损失，而且还能帮助他们搞清楚阿里巴巴到底是不是真的知道咒语这个问题。阿里巴巴也没有损失，因为处于一箭之地的强盗们听不到他念的咒语，不必担心泄露了秘密，同时他又确信自己的咒语有效，也不会发生被射死的杯具。

强盗举起了右手，只见阿里巴巴的嘴动了几下，石门果真打开了，强盗举起了左手，阿里巴巴的嘴动了几下后石门又关上了。强盗还是有点不信，说不准这是巧合呢，他们不断地换着节奏举右手举左手，石门跟着他们的节奏开开关关，最后强盗们想，如果还认为这只是巧合，自己未免是个傻瓜，那还是相信了阿里巴巴吧。这样，阿里巴巴既没有告诉强盗进入山洞石门的咒语，同时又向强盗们证明了，他是知识这个咒语的。这就是零知识证明的一个重要实例。

8 区块链基础--女巫攻击

1）女巫攻击名称来源

根据 Flora Rhea Schreiberie 在 1973 年的小说《女巫》（Sybil）改编的同名电影，是一个化名 Sybil Dorsett 的女人心理治疗的故事。她被诊断为分离性身份认同障碍，兼具 16 种人格。我的那个天。高度精分一枚。

（2）什么是女巫攻击

在对等网络中，但节点通常具有多个身份标识，通过控制系统的大部分节点来消弱冗余备份的作用。

在 P2P 网络中，因为节点随时加入退出等原因，为了维持网络稳定，同一份数据通常需要备份到多个分布式节点上，这就是数据冗余机制。女巫攻击是**攻击数据冗余机制**的一种有效手段。

如果网络中存在一个恶意节点，那么同一个恶意节点可以具有多重身份，就如电影了的女主角都可以分裂出 16 个身份，但是恶意节点比它还能分。这一分可好，原来需要备份到多个节点的数据被欺骗地备份到了同一个恶意节点（该恶意节点伪装成多重身份），这就是女巫攻击。

（3）女巫攻击的解决方案

一种方法是工作量证明机制，即证明你是一个节点，别只说不练，而是要用计算能力证明，这样极大地增加了攻击的成本。

另一种方法是身份认证（相对于 PoW 协议，女巫攻击是基于 BFT 拜占庭使用容错协议的 Blockchain 需要考虑的问题，需要采用相应的身份认证机制）。

认证机制分为二类：

1）基于第三方的身份认证

每加入一个新的节点都需要与某一个可靠的第三方节点进行身份验证。

2）纯分布式的身份认证

每加入一个新的节点都需要获得当前网络中所有可靠节点的认证，这种方法采用了随机密钥分发验证的公钥体制的认证方式，需要获得网络中大多数节点的认证才能加入该网络。

9 区块链基础--51%攻击

比特币系统是基于 POW 共识机制，也就是需要使用大量的算力来维护网络

安全，但这种安全都是概率安全。在比特币白皮书中最后一节中，中本聪论述了当攻击节点所拥有的算力低于诚实节点所拥有的算力情况下，随着确认块数的增加，攻击节点获胜的概率呈指数下降，概率值快速趋近零，这是很多应用等待六个甚至六个以上确认的原因。

51%算力攻击是指掌握了比特币全网的 51%算力之后，用这些算力来重新计算已经确认过的区块，使区块链产生分叉并且获得利益的行为。

1.51%算力攻击流程

我们本处介绍一下如何发动 51%攻击，来进一步了解 51%攻击。

1.1 攻击前准备

假设攻击者要发送攻击，该攻击者已经具备以下条件：

(1) 该攻击者掌握了比特币全网的 51%算力，并能在攻击过程中一直保持。

（2）该攻击者持有 100 个比特币（可以更多）。

（3）该攻击者是个理性的人，攻击的目的只是为了获取利益，而并不是其他。

1.2 攻击者发动攻击

（1）攻击者将 100 个比特币转到交易所（此处我们假设是交易所，当然也可以是一些机构或者个人），然后将 100 个比特币以 10000 美金的价格卖出，获得 100W 美金。这个过程的时间越短越好，能大大节省攻击时间。

（2）攻击者用 51%算力从还没向交易所转币的区块开始重新计算并生成区块。假设攻击者向交易所转币的交易记录的区块为第 20 万个区块，攻击者就从第 199999 个区块开始重新计算生成区块。

(3)按照上述第一个假设条件，攻击者生成的攻击块链一定能追上原块链，

攻击者的攻击一定成功。

（4）当攻击者的区块链的长度超过原区块链 2 个或者更多区块，所有其他节点将丢弃原区块链（丢弃从 199999 之后的区块），接受攻击区块链。

攻击者成功发动了 51%算力攻击，让我们看看攻击后都发生了什么：

（1）原区块链上第 199999 个区块之后交易全部作废。

（2）第 199999 个区块之后没有交易的客户的币数量不受任何影响。

（3）第 199999 个区块之后转出比特币的人会发现币回来了，此处攻击者就可以拿回转向交易所的 100 个比特币。

（4）第 199999 个区块之后接收到比特币的人会发现币消失了。

通过分析我们发现，攻击者在获得 100W 美金的同时，自己转向交易所的 100 个比特币又回来了，这样就获得了 100W 美金。

1.3 51%算力攻击再分析

（1）在上述中，我们假设攻击者是理性的，只为了获得利益，所以上述攻击是可以获得最大利益的。但是如果攻击者是个疯子，那么攻击者就可以击垮整个比特币网络。

（2）在上述中，我们假设攻击者在整个攻击过程中一直保持 51%算力，这是可以确定攻击者一定攻击成功。但同样也存在无需 51%的算力也可以攻击成功可能，算力高只是表明从概率角度上出块快，在有限个区块内，算力稍低并不一定出块少。

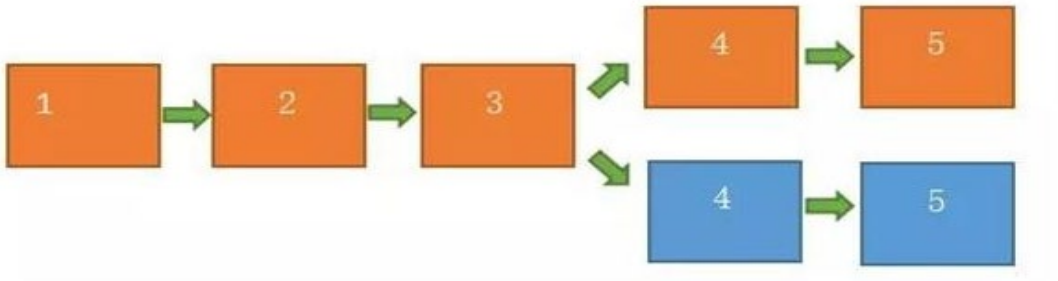
（3）比特币系统是一个世界范围参与的，51%算力攻击可以造成比特币最致命的伤害，需要加强对比特币网络的监控，保障比特币网络安全。

10 区块链基础--区块链分叉

我们在这里全面的讨论区块链分叉的情形，主要包括两大类。

10.1 区块链延伸过程中的不断分叉

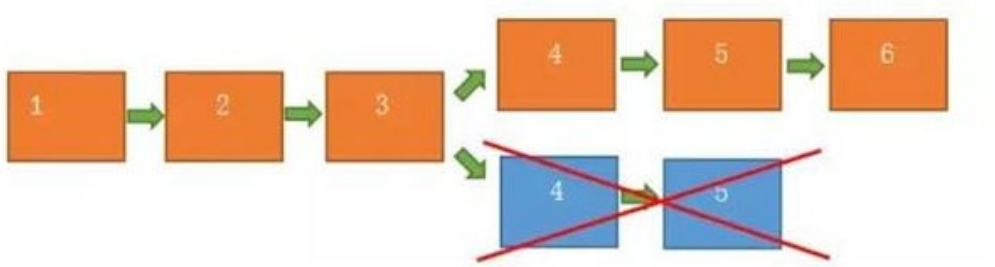
我们以比特币进行解释。当矿工使用 POW 共识机制寻找下一个满足**那个包含多个 0 的 HASH 值**的矿工可以生成这个区块，但是可能存在两个矿工同时找到符合条件的 HASH 值，并将自己的结果进行广播。区块链在这个时刻，出现了两个都满足要求的不同区块。那么，全体矿工这时该怎么办呢?由于距离远近，不同的矿工看到这两个区块是有先后顺序的。通常情况下，矿工们会把自己先看到的区块复制过来，然后接着在这个区块开始新的挖矿工作。于是，出现了这样的情景：



在以工作量证明机制为共识算法的区块链系统中，这个问题是这样被解决的：从分叉的区块起，由于不同的矿工跟从了不同的区块，在分叉出来的两条不同链上，算力是有差别的。形象地说，就是跟从两个链矿工的数量是不同的。

由于解题能力和矿工的数量成正比，因此两条链的增长速度也是不一样的，在一段时间之后，总有一条链的长度要超过另一条。当矿工发现全网有一条更长

的链时，他就会抛弃他当前的链，把新的更长的链全部复制回来，在这条链的基础上继续挖矿。所有矿工都这样操作，这条链就成为了主链，分叉出来被抛弃掉的链就消失了。



这种类型分叉是一种正常现象，分叉都会很快消失。最终，**只有一条链会被保留下来，成为真正有效的账本，其他都是无效的，所以整个区块链仍然是唯一的。**从 block hash 算法我们知道，合理的 block 并不是唯一的，同一高度存在多个 block 的可能性。那么，当同一个高度出现多个时，主链即出现分叉(Fork)。遇到分叉时，网络会根据下列原则选举出 Best Chain：

》不同高度的分支，总是接受最高（即最长）的那条分支

》相同高度的，接受难度最大的

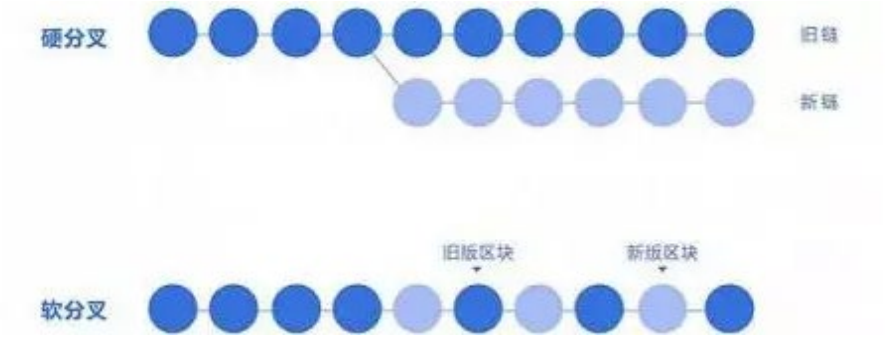
》高度相同且难度一致的，接受时间最早的

》若所有均相同，则按照从网络接受的顺序

》等待 Block Chain 高度增一，则重新选择 Best Chain

10.2 软分叉与硬分叉（由于软件升级导致的分叉）

下面这张示意图，可以很形象的看出硬分叉与软分叉两者的差别：



（1）软分叉

我们知道比特币网络是一个 P2P 网络，每个节点部署了 1 个比特币网络的客户端，类似 BT 下载软件。既然是软件，就避免不了要升级，不管是发布新 Feature，还是说 Bug Fix，软件必然是要一版有一版的升级。但问题就来了：全球那么多节点，不可能所有节点的客户端软件都同时升级。必然有的节点在用老版本的，有的节点在用新版本的，这个时候，软件兼容性问题就来了。

所谓软分叉，就是指这次升级是“前向兼容”的（forward-compatiable）。有的客户端升级到了新版本，有的客户端还用老版本，老版本的客户端可以正确处理新版本客户端产生的数据。具体点就是，新版本产生的 Block 和 Transaction，老版本的节点能正确处理；反过来，当然同样可以。

所以大家可以看到：软分叉其实并没有分叉，还是一条链，只是这条链上面的 Block 有不同的版本。 软分叉这个词，主要是相对接下来要讲的“硬分叉”来说的。软分叉看起来很好，新旧版本一起工作，但它也有很多争议。

软分叉的争议点：

1）软分叉加大了技术复杂度。在设计新版本的一致性协议的时候，还要考虑这个协议要被老版本的兼容，因为历史包袱，设计自然带有折中妥协。导致新

版本的代码可维护性变差，自然出 Bug 的概念变高，安全性变差。

2）软分叉升级不能回滚

举个例子：现在整个网络所有节点的版本都是 v1，然后某些节点升级到了 v2，在 v2 这个版本上面成交了一些 Block 和 Transaction。但现在你发现 v2 这个版本有个安全的 Bug，你要回滚 v2，让所有节点都回到 v1。那 v2 上面挖出的 Block 和 Transaction，在 v1 版本上面虽然可以验证通过，但其实是放松了限制的，这会导致安全问题。所以这就意味着 v2 上面挖出的 Block 和 Transaction 要作废。

（2）硬分叉

硬分叉就比较容易理解了，新发布的版本和旧版本不兼容。意味着：要么所有节点都升级到新版本，这样整个网络还是一条链；要么一部分节点用旧版本，一部分节点用新版本，大家互相不认可对方挖出的 Block，整个网络分裂成 2 条链，这相当于发行了一种新的货币！！！

而这就要说到比特币的 2 大门派：隔离见证派 与 矿池派的撕逼大战，也正是这场撕逼大战，导致现在的比特币出现了 1 次硬分叉，搞出了 2 种货币：旧的 BTC(Bitcoin Coin) 与 新的 BCH（Bitcoin Cash）。具体参考[15]

11 区块链基础--矿池

11.1 矿池产生背景

以比特币为例说明，中本聪希望每个人都能通过自己电脑的 cpu 来实现挖矿，尽量实现去中心化及分散性。但随着挖矿人数增长及高性能矿机 ASCI 投入挖矿，挖矿算力大幅度上涨，挖矿难度也呈指数级别上涨，个体矿工找到一个

区块从而获得奖励的可能性已经非常非常小。矿工希望追求持续稳定的收益，于是矿池应运而生，矿池通过汇集所有参与者们的算力一起挖矿，提高爆块的机率，然后根据每个矿工的贡献比例分享奖励。

矿池就是通过将矿工算力汇集在一起挖矿，这样就能挖到区块的概率会大大增加，然后把收益根据每个人的算力占比去分配。矿池的核心工作是给矿工分配任务，统计工作量并分发收益。和 Solo 模式相比，矿工收益的期望值没有变，但收益更加持续稳定。

11.2 矿池的工作原理

矿池通过专用挖矿协议连接成百上千的矿工，矿工设置矿机的挖矿软件连接到矿池指定的域名和端口。矿机在挖矿时保持和矿池服务器的连接，和其他矿工同步各自的工作，这样矿池中的矿工拿到不同的挖矿任务，之后分享奖励，成功爆块的奖励支付到矿池的私有钱包地址，而不是单个矿工的。矿池每天按矿工贡献支付奖励到矿工的钱包地址，但是因为支付有手续费的缘故，一般以太坊矿池将 0.1 eth 设为最低支付标准，当矿工当天的奖励金额小于 0.1eth 时，矿池会将这部分金额累计，直到某天矿工待支付的奖励金额大于 0.1。

矿池把搜寻候选区块的工作量分割，并根据矿工挖矿的贡献计算相应的“份额(share)”。矿池为“份额(share)”设置了一个低难度的目标，通常比公网难度低 1000 倍以上，矿池给矿工低难度的新任务，每次计算完成之后，便提交给矿池一个 share。当矿池验证这些 share 没有问题则接收并统计数量。矿池分配在收益时，则根据各个矿工提交的 share 数量的多少，按占比的比例，来分配这些新币。

矿池任务分配原理：

通过前面学习我们知道，挖矿实际就是计算机不停进行哈希运算，直到找到符合条件的目标值，目标值是一个总长度 256 位的二进制数，要求前 n 位为连续 0。矿池将区块难度分成很多难度更小的任务下发给矿工计算，矿工完成一个任务后将结果提交给矿池，叫提交一个 share。假设全网难度要求 n 的值为 100，即前 100 个比特位为 0，矿池可能会给矿工分配一个任务，要求前 30 位为 0，然后再从所有提交的任务中，寻找有没有凑巧前 100 位为 0 的目标值。

不同矿机算力大小不同，矿池会根据大家的算力大小分配难度不同的任务。比如 A 矿机算力 1T，B 矿机算力 10T，那么矿池给 A 矿机分配任务要求前 10 个比特位为 0，B 矿机的任务可能会是前 20 个比特位为 0。前 20 个比特位为 0 成为符合条件目标值的概率肯定是大于前 10 个比特位为 0 的概率。

11.3 矿池难度

大家在矿池的介绍都会看到一个 Starting Difficulty（起始难度）参数，一些矿池的不同端口，起始难度也是不同的。**矿池的难度，就是每次提交的 shares 数量**，比如难度 15000，那么一个成功的 result 提交的是 15000 个 shares。

如何选择这个起始难度，对收益还是有一定的影响的，基本道理就是：如果算力低，选择起始难度低的端口，如果算力大，选择起始难度高的端口。

这个起始难度是由矿池决定的，挖着挖着，矿池也会更改难度的大小，但是我们可以设置一个固定值。以下部分资料通过网络查阅得知，并非官方的资料，可以作为参考：

难度系数最大的作用是为了减少矿池被类似 DDoS，如果难度设置比较低，

矿工一秒可以提交很多的 shares 上来，矿池处理不了，就像被 DDoS 了。但是从另一个角度，如果矿池的起始难度太高，在矿池挖到一个块（Block）的时候，你一个 share 都没提交，那么根据 PPLNS 的收益分配规则，你是得不到收益的，那么就白挖了。所以在寻找矿池的时候，还要仔细查看矿池的起始难度，这个难度值与矿机要有一定的匹配性。

那如何选择起始难度呢，有网友说是**算力的 12 倍**左右（这个是不确定的，有的说 20 倍，不是官方的说法，有机会可以自己摸索），比如你的算力是 200 Hashes/s，那么选择难度在 2400 左右的矿池端口，但是现在矿池的起始难度感觉都比较高，比如 minexmr 矿池的起始难度是 15000。

难度越大，提交一个 result 就需要更多的时间，当然这个 result 代表的 shares 也更多。既然 minexmr 的起始难度比较高，如果我们的矿机性能不是特别好，我们可以限制矿池调整难度，使用最低的起始难度。不同的矿池有不同的设置方法，具体可以查看矿池的介绍页面，一般都是在钱包地址后面+难度值，比如 minexmr 的矿工可以设置为钱包地址+15000。

11.4 矿池不同的结算方式

矿池中常见的不同结算方式：PPS/PPLNS/P2P/SOLO。如果你是普通的挖矿者，只想挖矿赚钱，那么推荐使用 PPS 结算方式，这样收益会很稳定。如果你是个小赌徒，喜欢中奖的感觉，可以考虑 PPLNS 模式，手续费更少，但收益如何，就看运气咯。如果你是大算力用户，并且对自己的运气很有信心，那么可选择自己 SOLO，如果挖到了全是自己的，如果没挖到就什么也没有。适合大算力矿场的专业级矿工。目前 P2P 基本上已经不再使用了。

（1）PPLNS 方式

(最纯正的组队挖矿)全称 Pay Per Last N Shares，意思是说“根据过去的 N 个 share 来支付收益”，这意味着，所有的矿工一旦发现了一个区块，大家将根据每个人自己贡献的股份数量占比来分配区块中的货币。(share 就是股份的意思)

举个例子：假设，张三、李四、王五，这三个人在同一个 PPLNS 矿池中挖矿，在过去的一段时间里，张三贡献了 10 个 share，李四贡献 3 个，王五贡献 12 个，加起来是 25 个 share，这时矿池发现了一个区块，区块中含有 25 个比特币，那么，张三就会分到 10/25 个区块的奖励，也就是 10 个比特币，而李四获得 3 个，王五获得 12 个。

在 PPLNS 模式下，运气成份非常重要，如果矿池一天能够发现很多个区块，那么大家的分红也会非常多，如果矿池一天下来都没有能够发现区块，那么大家也就没有任何收益。同时，由于 PPLNS 下，具有一定的滞后惯性，你的挖矿收益会有一定的延迟，比如说，你加入到一个新的 PPLNS 矿池，这个时候你会发现前面几个小时的收益比较低，那是因为别人在这个矿池里已经贡献了很多个 share 了，你是新来的，你的贡献还很少，所以分红时你的收益都是比较低的。随着时间的推移，该结算的也结算了，大家又开始进行了新一轮的运算时，你就回到和别人一样的水平了。同样道理，若你离开了 PPLNS 矿池不再挖矿，你贡献的 share 还在，在此后的一段时间里，你依然会得到分红收益，直到你的 share 被结算完毕。一般情况下，PPLNS 模式的矿池收费在 1%~4%不等。

（2）PPS 方式

PPS 矿池(类似于打工模式)PPS 全称为 Pay Per Share。为了解决 PPLNS 那种有时候收益很高，有时候没有收益的情况，PPS 采用了新的算法。PPS 根据你的

算力在矿池中的占比，并估算了矿池每天可以获得的矿产，给你每天基本固定的收益。这么举例就很好理解：假设你的算力是 100M，而整个矿池的算力是10000M，那么你就占据了矿池算力的 1%，然后，假设矿池根据当前的难度和全球总算力，估算出矿池一天大约能够挖到 4 个区块，也就是 100 个比特币，那么，矿池会为你每天支付全矿池 1%，也就是 1 个比特币的报酬，这样，即使矿池今天只挖到了 1 个区块，你也是获得 1 个比特币(矿池亏本)，如果矿池超额发挥，挖到了 10 个区块，你还是只有 1 个比特币的收益(矿池大赚)。实际上，PPS模式的矿池为了避免亏本风险，往往会收取 3%-8%的高额手续费。

（3）P2P 方式

P2P 挖矿模式是继承 p2pool 开源项目而推出的挖矿模式，其为矿工提供了一种点对点的挖矿方式，防止算力集中在 某个中心化的矿池而对比特币网络进行 51%攻击。目前 p2p 网络情况不怎么好，所以已经基本不用了。

（4）SOLO 方式

SOLO 挖矿，就是自己单枪匹马的自己挖，自己计算区块，去碰撞 HASH 值，碰撞到了，那么此次块区奖励全部归你个人所有，同理，如果你一天里没有爆快，那么你一天就是颗粒无收。SOLO 模式目前都是算力比较大的用户选择，因为要和其他人抢，如果算力太低，很难抢得到。

12 区块链基础--拜占庭问题与两军问题

12.1 拜占庭将军问题描述

1982 年，由 Leslie Lamport 与另外两人提出拜占庭将军问题，被称为 The Byzantine Generals Problem 或者 Byzantine Failure。核心描述是军中可能有叛徒，

却要保证进攻一致性问题，由此引申到计算领域，发展成了一种容错理论。关于拜占庭将军问题，引用网上的描述：

拜占庭帝国想要进攻一个强大的敌人，为此派出了 10 支军队去包围这个敌人。这个敌人虽不比拜占庭帝国，但也足以抵御 5 支常规拜占庭军队的同时袭击。基于一些原因，这 10 支军队不能集合在一起单点突破，必须在分开的包围状态下同时攻击。他们任一支军队单独进攻都毫无胜算，除非有至少 6 支军队同时袭击才能攻下敌国。他们分散在敌国的四周，依靠通信兵相互通信来协商进攻意向及进攻时间。困扰这些将军的问题是，他们不确定他们中是否有叛徒，叛徒可能擅自变更进攻意向或者进攻时间。在这种状态下，拜占庭将军们能否找到一种分布式的协议来让他们能够远程协商，从而赢取战斗？这就是著名的拜占庭将军问题。

2. 拜占庭将军问题分析

（1）拜占庭将军问题的前提是消息传递的信道没有问题（也就是通信兵不会被截获或者无法传递问题）。

这是因为 Lamport 已经证明了在消息可能丢失的不可靠信道上试图通过消息传递的方式达到一致性是不可能的。所以，在研究拜占庭将军问题的时候，我们已经假定了**信道是没有问题的**。

（2）拜占庭将军问题的复杂性

拜占庭将军问题十分复杂，从上述描述中，我们可能觉察不到，接下来我们一起看看其复杂性。

1）假设没有叛徒的情形，也就是十个将军（A1、A2…A10）都是忠诚的。

如果将军 A1 提一个进攻提议，协议内容为“2020 年 5 月 20 日上午 10 点开

始进攻敌人，确认参加请签名”。由通信兵通信分别告诉其他的将军，如果很幸运，他收到了其他 6 位将军以上的同意，就可以成功发起进攻。如果不幸，其他的将军也在此时发出不同的进攻提议，如“2020 年 5 月 25 日上午 10 点开始进攻敌人，确认参加请签名”等。由于时间上的差异，不同的将军收到并认可的进攻提议可能是不一样的，此时可能出现 A1 提议有 3 个支持者，A2 提议有 4 个支持者，A3 提议有 2 个支持者等等。

2）假设有叛徒的情况，这种情况更加复杂。

在没有叛徒的情形下，达成一致性都特别困难，当 10 个将军中出现叛徒将军时，情况更加复杂。比如一个叛徒可以给不同的将军发送不同的进攻时间。假设 A1 是叛徒，那么他可能如下操作：

A1 给 A2 的协议：“2020 年 5 月 20 日上午 10 点开始进攻敌人，确认参加请签名”

A1 给 A3 的协议：“2020 年 5 月 28 日下午 13 点开始进攻敌人，确认参加请签名”

……

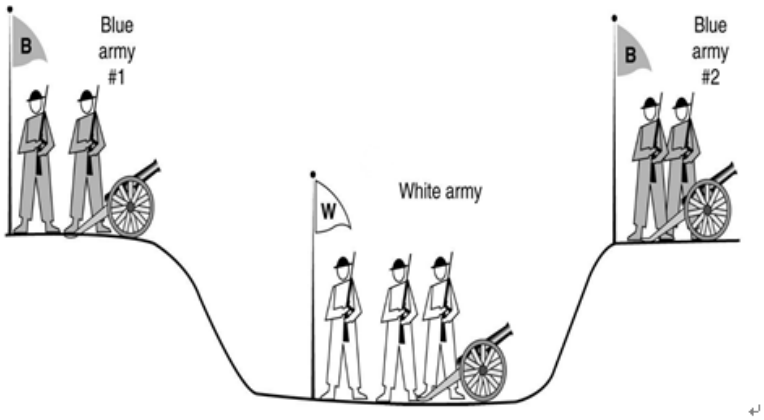
A1 给 A10 的协议：“2020 年 6 月 20 日上午 10 点开始进攻敌人，确认参加请签名”

叛徒发送前后不一致的进攻提议，被称为“拜占庭错误”，而能够处理拜占庭错误的这种容错性称为「Byzantine fault tolerance」，简称为 BFT。

12.2 两军问题

正如前文所说，拜占庭将军问题和两军问题实质是不一样的。国内大量解释

拜占庭将军问题的文章将两者混为一谈，其实是混淆了两个问题的实质，由此造成了许多误解。这两个问题看起来的确有点相似，但是**问题的前提和研究方向**都截然不同。



（图1：两军问题图示）

如图 1 所示，白军驻扎在沟渠里，蓝军则分散在沟渠两边。白军比任何一支蓝军都更为强大，但是蓝军若能同时合力进攻则能够打败白军。他们不能够远程的沟通，只能派遣通信兵穿过沟渠去通知对方蓝军协商进攻时间。**是否存在一个能使蓝军必胜的通信协议，这就是两军问题**。

看到这里您可能发现两军问题和拜占庭将军问题有一定的相似性，但我们必须注意的是，通信兵得经过敌人的沟渠，在这过程中他可能被捕，也就是说，**两军问题中信道是不可靠的，并且其中没有叛徒之说，这就是两军问题和拜占庭将军问题的根本性不同。**由此可见，大量混淆了拜占庭将军问题和两军问题的文章并没有充分理解两者。

两军问题的根本问题在于信道的不可靠，反过来说，如果传递消息的信道是可靠的，两军问题可解。然而，并不存在这样一种信道，所以两军问题在经典情

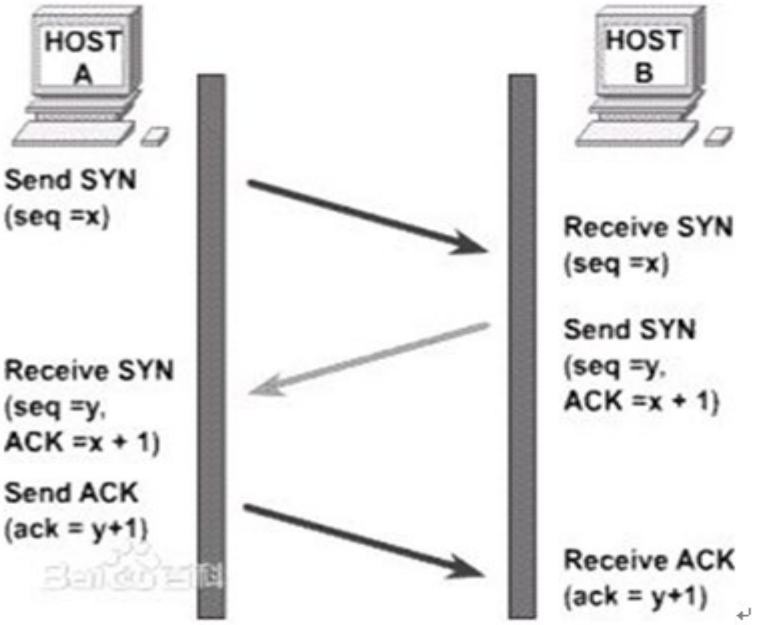
境下是不可解的，为什么呢？

倘若 1 号蓝军（简称 1）向 2 号蓝军（简称 2）派出了通信兵，若 1 要知道 2 是否收到了自己的信息，1 必须要求 2 给自己传输一个回执，说“你的信息我已经收到了，我同意你提议的明天早上 10 点 9 分准时进攻”。

然而，就算 2 已经送出了这条信息，2 也不能确定 1 就一定会在这个时间进攻，因为 2 发出的回执 1 并不一定能够收到。所以，1 必须再给 2 发出一个回执说“我收到了”，但是 1 也不会知道 2 是否收到了这样一个回执，所以 1 还会期待一个 2 的回执。

虽然看似很可笑，但在这个系统中永远需要存在一个回执，这对于两方来说都并不一定能够达成十足的确信。更要命的是，我们还没有考虑，通信兵的信息还有可能被篡改。由此可见，经典情形下两军问题是不可解的，并不存在一个能使蓝军一定胜利的通信协议。

不幸的是，两军问题作为现代通信系统中必须解决的问题，我们尚不能将之完全解决，这意味着你我传输信息时仍然可能出现丢失、监听或篡改的情况。但我们能不能通过一种相对可靠的方式来解决大部分情形呢？这需要谈到 **TCP 协议**。事实上，搜索“两军问题与三次握手”，您一定可以找到大量与 TCP 协议相关的内容。



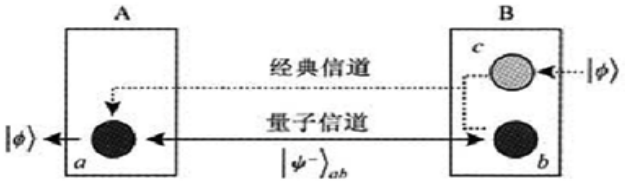
（图2：TCP协议的基本原理）

TCP 协议中，A 先向 B 发出一个随机数 x，B 收到 x 了以后，发给 A 另一个随机数 y 以及 x+1 作为答复，这样 A 就知道 B 已经收到了，因为要破解随机数 x 可能性并不大；然后 A 再发回 y+1 给 B，这样 B 就知道 A 已经收到了。这样，A 和 B 之间就建立一个可靠的连接，彼此相信对方已经收到并确认了信息。

而事实上，A 并不会知道 B 是否收到了 y+1；并且，由于信道的不可靠性，x 或者 y 都是可能被截获的，这些问题说明了即使是三次握手，也并不能够彻底解决两军问题，**只是在现实成本可控的条件下，我们把 TCP 协议当作了两军问题的现实可解方法**。

那么，是否能够找到一个理论方法来真正的破解两军问题呢？答案是有的，**量子通讯协议**，笔者并没有能力弄清这个颇为高深的问题。据我的理解，处于量子纠缠态的两个粒子，无论相隔多远都能够彼此同步，光是直观的来看，这个效应可以用来实现保密通讯。

但是由于测不准原理，一测量粒子状态就会改变其状态，所以通讯时还必须通过不可靠信道发送另一条信息。尽管这个“另一条信息”是不可靠的，但是由于已经存在了一条绝对可靠的信道（量子纠缠），保证了另一条信道即使不可靠也能保证消息是可靠的，否则至少被窃取了一定能够被发现。



（图3：量子隐形传态的原理图）

因此我们可以相信，至少理论上两军问题是可解的，即存在一种方法，即使利用了不可靠的信道，也能保证信息传递的可靠性。所以，在确保了信道可靠的基础上，我们可以回到拜占庭将军问题上继续讨论。

12.3 区块链解决拜占庭问题

拜占庭问题的实质就是共识机制问题。也就是忠诚的将军之间如何达成一致性的目标（一致性进攻或者撤退），当然这里假设都是忠诚的将军多于不忠诚的将军。

在出现比特币之前，解决分布式系统一致性问题主要是 Lamport 提出的 Paxos 算法或其衍生算法。Paxos 类算法仅适用于中心化的分布式系统，这样的系统的没有不诚实的节点（不会发送虚假错误消息，但允许出现网络不通或宕机出现的消息延迟）。

中本聪在比特币中创造性的引入了“工作量证明（POW : Proof of Work）”来

解决这个问题。通过工作量证明就增加了发送信息的成本，降低节点发送消息速率，这样就以保证在一个时间只有一个节点(或是很少)在进行广播，同时在广播时会附上自己的签名。

这个过程就像一位将军 A 在向其他的将军（B、C、D…）发起一个进攻提议一样，将军 B、C、D…看到将军 A 签过名的进攻提议书，如果是诚实的将军就会立刻同意进攻提议，而不会发起自己新的进攻提议。

以上就是比特币网络中是单个区块（账本）达成共识的方法（取得一致性）。理解了单个区块取得一致性的方法，那么整个区块链（总账本）如果达成一致也好理解。

我们稍微把将军问题改一下：假设攻下一个城堡需要多次的进攻，每次进攻的提议必须基于之前最多次数的胜利进攻下提出的（只有这样敌方已有损失最大，我方进攻胜利的可能性就更大），这样约定之后，将军 A 在收到进攻提议时，就会检查一下这个提议是不是基于最多的胜利提出的，如果不是（基于最多的胜利）将军 A 就不会同意这样的提议，如果是的，将军 A 就会把这次提议记下来。这就是比特币网络最长链选择

13 区块链基础—私钥、助记词及 keystore

区块链基础—私钥、助记词及 keystore

#科普探索

在区块链中，我们知道得私钥者得天下，拥有私钥就等于拥有对资产的所有权。比特币私钥是一个长度为 64 位的十六进制。助记词与 keystore 都是私钥

的不同展现形式。私钥、助记词及 keystore 关系如下：

“助记词”=“私钥”=“keystore+钱包密码”

14.1.助记词

助记词是私钥的明文展现形式，最早是由 BIP39 提案提出，其目的是为了帮助用户记忆复杂的私钥。助记词一般由 18、21 等多个单词构成，这些单词都取自一个固定词库，其生成顺序也是按照一定算法而来。助记词是未经加密的私钥，任何人得到了你的助记词，就可以夺走你的资产。做好助记词保护尤为重要。

（1）原则上做至少三个备份，比如纸质备份一份，电子备份两份。

（2）尽量在不联网的情形下，进行助记词备份，电子备份不建议截图方式。

（3）不同的备份一定要尽量分散放，不要放在同一个地方。

（4）电子备份需要进行加密处理，即使电子备份被盗也可以第一时间完成资金转移。

14.2keystore

Keystore 是用设置的钱包密码加密私钥后的文本，是私钥的加密展现形式，。当钱包密码修改后，keystore 也会发生变化。Keystore 需要配合钱包密码一起使用，备份了 keystore 同时别忘了备份钱包的密码。Keystore 常见于以太坊钱包。

此处 keystore 是加密的，所以可以在网络环境随意传播，只要你的钱包密码安全，其他人即使拿到你的 keystore 文件，也是无法操作你的资产。

四．参考

[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_BnLqHgusjR8QRUGGnbmCl0sQOgx7dnVz7ltSLdu7axz8Blbt57JF4qbMkBzO_xuwleugZVabWMUijelq2SdqMZcoFZabwhaYIclhlTQ_

[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