原址

区块链教程

区块链（Blockchain）技术源于比特币。在比特币中，为了保证每笔交易可信并不可篡改，中本聪发明了区块链，它通过后一个区块对前一个区块的引用，并以加密技术保证了区块链不可修改。

随着比特币的逐渐发展，人们发现区块链本质上其实是一个分布式的，不可篡改的数据库，天生具有可验证、可信任的特性，它不但可用于支持比特币，也可用于数字身份验证，清算业务等传统的必须由第三方介入的业务，从而降低交易成本。

虽然区块链近年来越来越火，各种概念和商业模式满天飞，但基于区块链底层技术的研究却很少。本教程从零基础开始，从底层开始研究区块链，彻底掌握区块链密码学原理、安全机制、共识技术与工程实现。最大的特色是：

零基础入门区块链，还能写代码！ 不仅掌握理论，还能写代码实现，这样就可以轻松识别真假区块链，同时对未来技术的发展有清晰的认识。

本教程代码主要用JavaScript编写，可在线运行，学习方便，省时省力！

比特币

比特币是人类历史上第一种数字货币。

什么是数字货币？一句话概括，数字货币是基于数学加密原理构建的不可伪造的货币系统，而比特币是第一个基于数学加密原理构建的分布式数字货币系统。

比特币和区块链有什么关系？一句话概括，比特币使用区块链技术实现了数字货币的可信支付。

比特币的历史可以追溯到2008年10月，一个名叫中本聪的神秘人物在一个密码学朋克论坛上发表了一篇比特币：一种点对点的电子现金系统的文章，这篇文章被看成是比特币的白皮书。

随后在08年11月，中本聪发布了比特币的第一版代码。09年1月，中本聪挖出了比特币的第一个区块——创世区块，比特币网络正式开始运行。

到现在，比特币已经运行了11年多。

数字货币 vs. 电子货币

说起货币，我们想到的就是日常生活中使用的纸币。但是，纸币并不是天生就出现的。如果追溯到三千多年前，人类社会并没有任何货币，部落之间的贸易是物物交换。随着经济和贸易的发展，迫切需要一种“一般等价物”来作为商品交换的“中介”，这种一般等价物就是货币。最早的货币是贝壳，后来由于金属冶炼技术的进步，出现了铜、铁铸造的货币。金属货币由于体积小，容易分割和铸造，逐渐获得了广泛的使用。最终，世界各国的金属货币都落到了金、银这几种贵金属上。

随着经济的继续发展，金属货币因为沉重并且不易携带，因此，人们发明了纸币。世界上最早的纸币出现在中国宋朝，称为“交子”。纸币的发行机制决定了必须由政府发行，并且强行推广使用，因此纸币又称法币。

随着计算机技术的发展，银行系统经过几十年的发展，已经用计算机系统完全代替了人工记账，纸币也实现了电子化。现在，我们可以自由地使用网银、支付宝这样的工具实现随时随地转账付款，就得益于纸币的电子化和网络化。

电子货币本质上仍然是法币，它仍然是由央行发行，只是以计算机技术把货币以实体纸币形式的流通变成了银行计算机系统的存款。和纸币相比，电子货币具有更高的流动性。我们每天使用的网上银行、支付宝、微信支付等，都是这种方式。

而比特币作为一种数字货币，它和电子货币不同的是，比特币不需要一个类似银行的中央信任机构，就可以通过全球P2P网络进行发行和流通，这一点听上去有点不可思议，但比特币正是一种通过密码学理论建立的不可伪造的货币系统。

比特币解决的问题

比特币通过技术手段解决了现金电子化以后交易的清结算问题。

传统的基于银行等金融机构进行交易，本质上是通过中央数据库，确保两个交易用户的余额一增一减。这些交易高度依赖专业的开发和运维人员，以及完善的风控机制。

比特币则是通过区块链技术，把整个账本全部公开，人手一份，全网相同，因此，修改账本不会被其他人承认。比特币的区块链就是一种存储了全部账本的链式数据库，通过一系列密码学理论进行防篡改，防双花。

如果我们从现金和存款的角度看，现金是M0，而银行存款是M1和M2。银行存款本质上已经不是现金，而是用户的资产，对应着银行的负债。因为银行只记录用户在银行的资产余额，因此，用户A通过银行把100元转账给用户B的时候，用户A的资产减少100元，相应的，用户B的资产增加100元，银行对用户A和用户B的总负债不变。换句话说，存款是用户的“提款期权”。

而现金则是由用户自己负责保存的货币。如果用户A把100元现金给用户B，那么此交易并不需要通过银行，因为使用现金时，用户与银行之间没有资产和负债关系。

通过银行转移存款，对用户来说很方便，但永远绕不过中央信任机构，并且用户必须信任银行不会篡改余额。通过现金交易，用户并不需要金融中介，但是需要当面交易，以及会遇到现钞的防伪、防盗等问题。

比特币解决的是现金电子化后无需中央信任机构的交易问题，即M0如何通过网络进行价值传输。我们已经习惯了通过互联网对数字化的新闻、音乐、视频进行信息传输，因为信息传输的本质是复制，但现实世界的现金可不能复制。想象一下我们如何把100元现金通过网络发送给另一个人，同时确保交易前后两个人的现金总额保持不变。所以，中本聪的白皮书把比特币定义为“点对点的电子现金系统”。

小结

总的来说，比特币具有以下特点：

- 创建了无需信任中心的货币发行机制；
- 发行数量由程序决定，无法随意修改；
- 交易账本完全公开可追溯，不可篡改；
- 密码学理论保证货币防伪造，防双花；
- 数字签名机制保证交易完整可信，不可抵赖和撤销。

区块链原理

区块链就是一个不断增长的全网总账本，每个完全节点都拥有完整的区块链，并且，节点总是信任最长的区块链，伪造区块链需要拥有超过51%的全网算力。

区块链的一个重要特性就是不可篡改。为什么区块链不可篡改？我们先来看区块链的结构。

区块链是由一个一个区块构成的有序链表，每一个区块都记录了一系列交易，并且，每个区块都指向前一个区块，从而形成一个链条：

如果我们观察某一个区块，就可以看到，每个区块都有一个唯一的哈希标识，被称为区块哈希，同时，区块通过记录上一个区块的哈希来指向上一个区块：

每一个区块还有一个Merkle哈希用来确保该区块的所有交易记录无法被篡改。

区块链中的主要数据就是一系列交易，第一条交易通常是Coinbase交易，也就是矿工的挖矿奖励，后续交易都是用户的交易。

区块链的不可篡改特性是由哈希算法保证的。

哈希算法

我们来简单介绍一下什么是哈希算法。

哈希算法，又称散列算法，它是一个单向函数，可以把任意长度的输入数据转化为固定长度的输出：

$h = H(x)$

例如，对`morning`和`bitcoin`两个输入进行某种哈希运算，得到的结果是固定长度的数字：

```
H("morning") = c7c3169c21f1d92e9577871831d067c8
H("bitcoin") = cd5b1e4947e304476c788cd474fb579a
```

我们通常用十六进制表示哈希输出。

因为哈希算法是一个单向函数，要设计一个安全的哈希算法，就必须满足：通过输入可以很容易地计算输出，但是，反过来，通过输出无法反推输入，只能暴力穷举。

```
H("???????") = c7c3169c21f1d92e9577871831d067c8
H("???????") = cd5b1e4947e304476c788cd474fb579a
```

想要根据上述结果反推输入，只能由计算机暴力穷举。

哈希碰撞

一个安全的哈希算法还需要满足另一个条件：碰撞率低。

碰撞是指，如果两个输入数据不同，却恰好计算出了相同的哈希值，那么我们说发生了碰撞：

```
H("data-123456") = a76b1fb579a02a476c789d9115d4b201
H("data-ABCDEF") = a76b1fb579a02a476c789d9115d4b201
```

因为输入数据长度是不固定的，所以输入数据是一个无限大的集合，而输出数据长度是固定的，所以，输出数据是一个有限的集合。把一个无限的集合中的每个元素映射到一个有限的集合，就必然存在某些不同的输入得到了相同的输出。

哈希碰撞的本质是把无限的集合映射到有限的集合时必然会产生碰撞。我们需要计算的是碰撞的概率。很显然，碰撞的概率和输出的集合大小相关。输出位数越多，输出集合就越大，碰撞率就越低。

安全哈希算法还需要满足一个条件，就是输出无规律。输入数据任意一个bit（某个字节的某一个二进制位）的改动，会导致输出完全不同，从而让攻击者无法逐步猜测输入，只能依赖暴力穷举来破解：

```
H("hello-1") = 970db54ab8a93b7173cb48f55e67fd2c
H("hello-2") = 8284353b768977f05ac600baad8d3d17
```

哈希算法有什么作用？假设我们相信一个安全的哈希算法，那么我们认为，如果两个输入的哈希相同，我们认为两个输入是相同的。

如果输入的内容就是文件内容，而两个文件的哈希相同，说明文件没有被修改过。当我们从网站上下载一个非常大的文件时，我们如何确定下载到本地的文件和官方网站发布的原始文件是完全相同，没有经过修改的呢？哈希算法就体现出了作用：我们只需要计算下载到本地的文件哈希，再和官方网站给出的哈希对比，如果一致，说明下载文件是正确的，没有经过篡改，如果不一致，则说明下载的文件肯定被篡改过。

大多数软件的官方下载页面会同时给出该文件的哈希值，以便让用户下载后验证文件是否被篡改：

MySQL Community Server 5.7.17

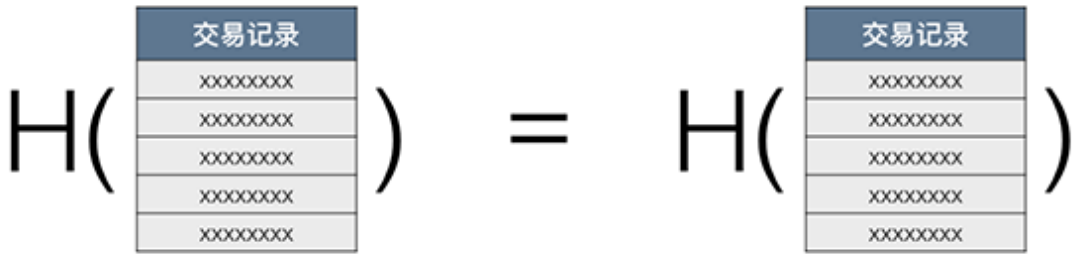
Select Platform:

Microsoft Windows

Other Downloads:

Windows (x86, 32-bit), ZIP Archive	5.7.17	341.3M	Download
(mysql-5.7.17-win32.zip)	MD5: d7497e614856d8f41b55b7ddabf82142 | Signature		
Windows (x86, 64-bit), ZIP Archive	5.7.17	355.3M	Download
(mysql-5.7.17-winx64.zip)	MD5: 95155e6addfbd35ec6624d5807f7a27d | Signature		
Windows (x86, 32-bit), ZIP Archive Debug Binaries & Test Suite	5.7.17	414.1M	Download
(mysql-5.7.17-win32-debug-test.zip)	MD5: 5845a8229da4f662eccbb5bdbbfacfbf | Signature		
Windows (x86, 64-bit), ZIP Archive Debug Binaries & Test Suite	5.7.17	423.5M	Download
(mysql-5.7.17-winx64-debug-test.zip)	MD5: 7d73bf1cbe9a2ae3097f244ef36616dc | Signature		

和文件类似，如果两份数据的哈希相同，则可以100%肯定，两份数据是相同的。比特币使用哈希算法来保证所有交易不可修改，就是计算并记录交易的哈希，如果交易被篡改，那么哈希验证将无法通过，说明这个区块是无效的。



常用哈希算法

常用的哈希算法以及它们的输出长度如下：

哈希算法	输出长度(bit)	输出长度(字节)
MD5	128 bit	16 bytes
RipeMD160	160 bits	20 bytes
SHA-1	160 bits	20 bytes
SHA-256	256 bits	32 bytes
SHA-512	512 bits	64 bytes

比特币使用的哈希算法有两种：`SHA-256`和`RipeMD160`

`SHA-256`的理论碰撞概率是：尝试2的130次方的随机输入，有99.8%的概率碰撞。注意2130是一个非常大的数字，大约是1361万亿亿亿亿。以现有的计算机的计算能力，是不可能在短期内破解的。

比特币使用两种哈希算法，一种是对数据进行两次`SHA-256`计算，这种算法在比特币协议中通常被称为`hash256`或者`dhash`。

另一种算法是先计算`SHA-256`，再计算`RipeMD160`，这种算法在比特币协议中通常被称为hash160。

```
const
    bitcoin = require('bitcoinjs-lib'),
    createHash = require('create-hash');
```

```
function standardHash(name, data) {
    let h = createHash(name);
    return h.update(data).digest();
}

function hash160(data) {
    let h1 = standardHash('sha256', data);
    let h2 = standardHash('ripemd160', h1);
    return h2;
}

function hash256(data) {
    let h1 = standardHash('sha256', data);
    let h2 = standardHash('sha256', h1);
    return h2;
}

let s = 'bitcoin is awesome';
console.log('ripemd160 = ' + standardHash('ripemd160', s).toString('hex'));
console.log('  hash160 = ' + hash160(s).toString('hex'));
console.log('   sha256 = ' + standardHash('sha256', s).toString('hex'));
console.log('  hash256 = ' + hash256(s).toString('hex'));
```

```
ripemd160 = 46c047bd035afb64dad2293cba29994a95b8b216
  hash160 = fe56649aa4f8fdb1edf6b88d2d41f3c1f72cf431
   sha256 = 23d4a09295be678b21a5f1dceae1f634a69c1b41775f680ebf8165266471401b
  hash256 = 1c78f53758ac96f43b99ed080f36327d2a823c4df4fa094e59b006d945bbb84d
```

运行上述代码，观察对一个字符串进行SHA-256、RipeMD160、hash256和hash160的结果。

区块链不可篡改特性

有了哈希算法的预备知识，我们来看比特币的区块链如何使用哈希算法来防止交易记录被篡改。

区块本身记录的主要数据就是一系列交易，所以，区块链首先要保证任何交易数据都不可修改。

Merkle Hash

在区块的头部，有一个Merkle Hash字段，它记录了本区块所有交易的Merkle Hash：

Version	536870912
Prev Hash	0000001ce749fb5b668ac54...
Merkle Hash	**174e90a4c40a8f2c0b2e5df3...**
Timestamp	1478073134
Bits	402937298
Nonce	0 ~ 0xffffffff

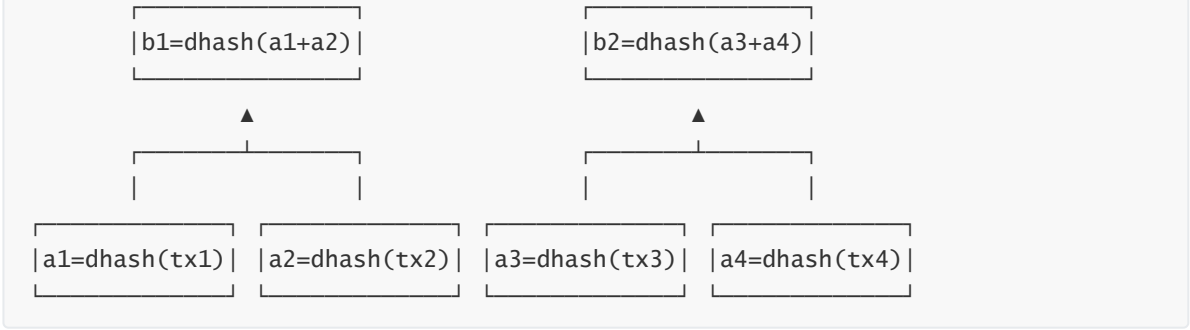
in	out	฿
小明	小红	2.0
汪星人	喵星人	8.8
...	...	...

Merkle Hash是把一系列数据的哈希根据一个简单算法变成一个汇总的哈希。

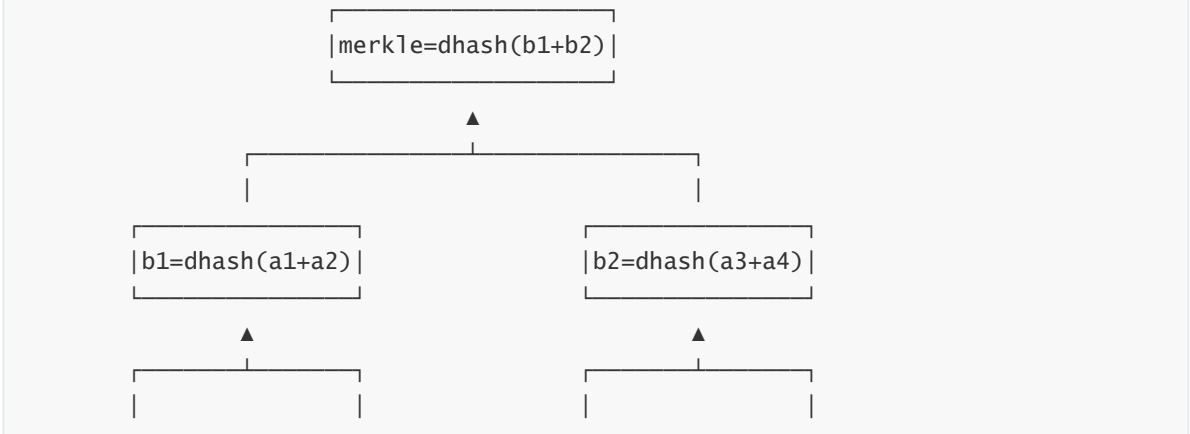
假设一个区块有4个交易，我们对每个交易数据做dhash，得到4个哈希值`a1`，`a2`，`a3`和`a4`：

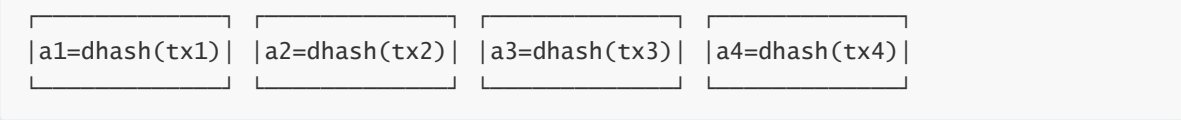
```
a1 = dhash(tx1)
a2 = dhash(tx2)
a3 = dhash(tx3)
a4 = dhash(tx4)
```

注意到哈希值也可以看做数据，所以可以把`a1`和`a2`拼起来，`a3`和`a4`拼起来，再计算出两个哈希值`b1`和`b2`：

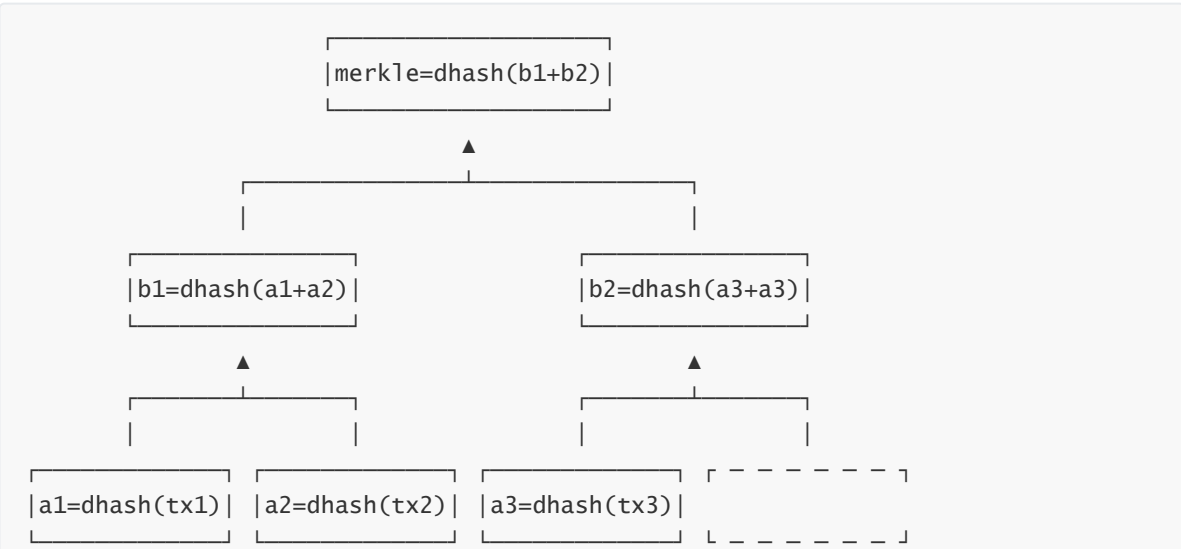


最后，把`b1`和`b2`这两个哈希值拼起来，计算出最终的哈希值，这个哈希就是Merkle Hash：

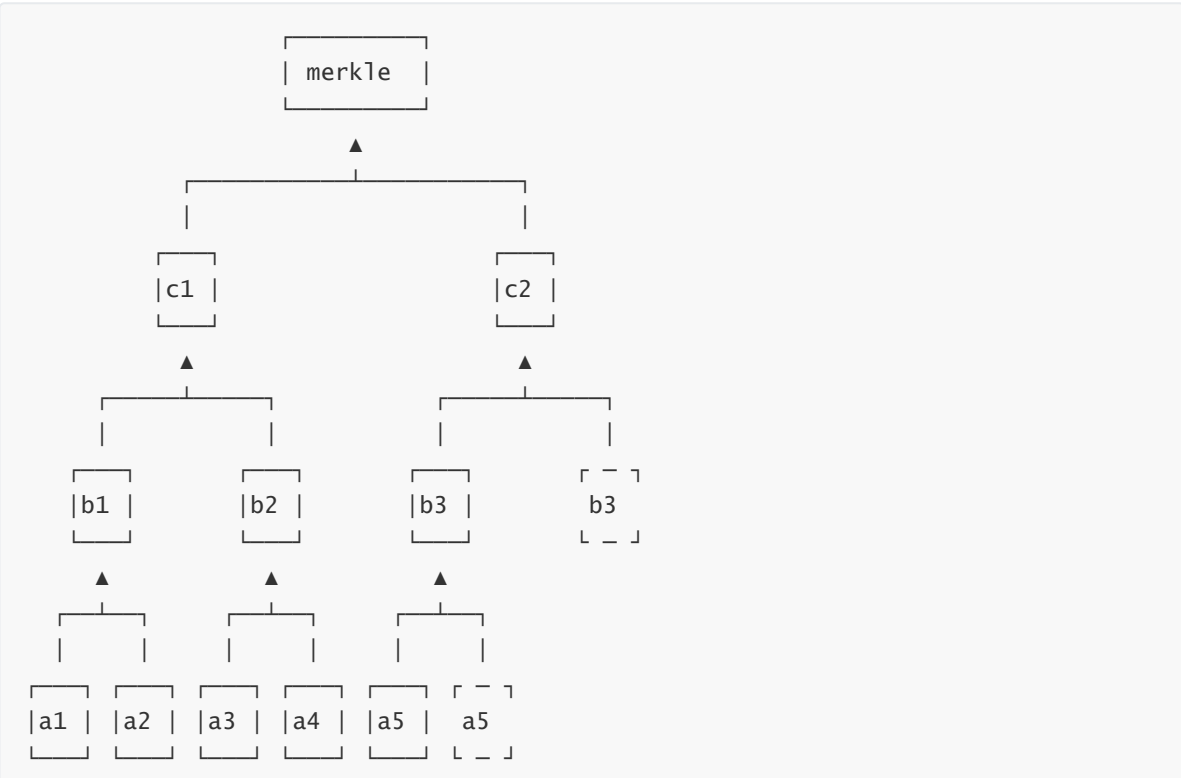




如果交易的数量不恰好是4个怎么办？例如，只有3个交易时，第一个和第二个交易的哈希`a1`和`a2`可以拼起来算出`b1`，第三个交易只能算出一个哈希`a3`，这个时候，就把a3直接复制一份，算出`b2`，这样，我们也能最终计算出Merkle Hash：



如果有5个交易，我们可以看到，`a5`被复制了一份，以便计算出`b3`，随后`b3`也被复制了一份，以便计算出`c2`。总之，在每一层计算中，如果有单数，就把最后一份数据复制，最后一定能计算出Merkle Hash：



从Merkle Hash的计算方法可以得出结论：修改任意一个交易哪怕一个字节，或者交换两个交易的顺序，都会导致Merkle Hash验证失败，也就会导致这个区块本身是无效的，所以，Merkle Hash记录在区块头部，它的作用就是保证交易记录永远无法修改。

Block Hash

区块本身用Block Hash——也就是区块哈希来标识。但是，一个区块自己的区块哈希并没有记录在区块头部，而是通过计算区块头部的哈希得到的：

区块头部的Prev Hash记录了上一个区块的Block Hash，这样，可以通过Prev Hash追踪到上一个区块。

由于下一个区块的Prev Hash又会指向当前区块，这样，每个区块的Prev Hash都指向自己的上一个区块，这些区块串起来就形成了区块链。

区块链的第一个区块（又称创世区块）并没有上一个区块，因此，它的Prev Hash被设置为`00000000...000`。

如果一个恶意的攻击者修改了一个区块中的某个交易，那么Merkle Hash验证就不会通过。所以，他只能重新计算Merkle Hash，然后把区块头的Merkle Hash也修改了。这时，我们就会发现，这个区块本身的Block Hash就变了，所以，下一个区块指向它的链接就断掉了。

由于比特币区块的哈希必须满足一个难度值，因此，攻击者必须先重新计算这个区块的Block Hash，然后，再把后续所有区块全部重新计算并且伪造出来，才能够修改整个区块链。

在后面的挖矿中，我们会看到，修改一个区块的成本就已经非常非常高了，要修改后续所有区块，这个攻击者必须掌握全网51%以上的算力才行，所以，修改区块链的难度是非常非常大的，并且，由于正常的区块链在不断增长，同样一个区块，修改它的难度会随着时间的推移而不断增加。

小结

区块链依靠安全的哈希算法保证所有区块数据不可更改；

交易数据依靠Merkle Hash确保无法修改，整个区块依靠Block Hash确保区块无法修改；

工作量证明机制（挖矿）保证修改区块链的难度非常巨大从而无法实现。

P2P交易原理

比特币的交易是一种无需信任中介参与的P2P（Peer-to-peer）交易。

传统的电子交易，交易双方必须通过银行这样的信任机构作为中介，这样可以保证交易的安全性，因为银行记录了交易双方的账户资金，能保证在一笔交易中，要么保证成功，要么交易无效，不存在一方到账而另一方没有付款的情况：

但是在比特币这种去中心化的P2P网络中，并没有一个类似银行这样的信任机构存在，要想在两个节点之间达成交易，就必须实现一种在零信任的情况下安全交易的机制。

创建交易有两种方法：我们假设小明和小红希望达成一笔交易，一种创建交易的方法是小红声称小明给了他1万块钱，显然这是不可信的：

还有一种创建交易的方法是：小明声称他给了小红一万块钱，只要能验证这个声明确实是小明作出的，并且小明真的有1万块钱，那么这笔交易就被认为是有效的：

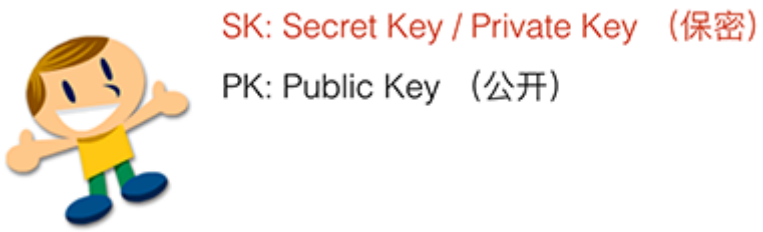
数字签名

如何验证这个声明确实是小明作出的呢？数字签名就可以验证这个声明是否是小明做的，并且，一旦验证通过，小明是无法抵赖的。

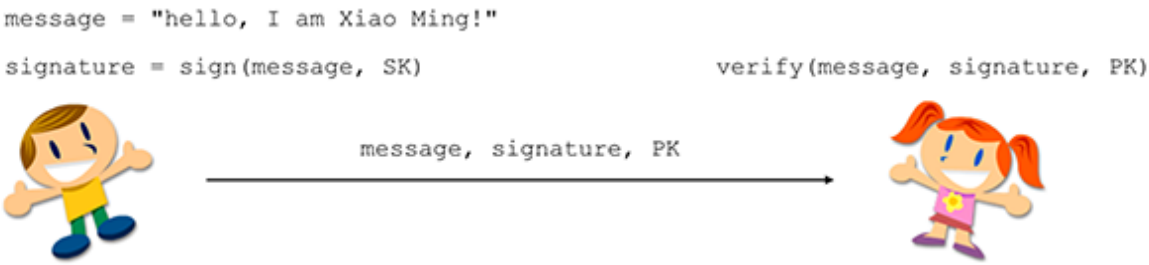
在比特币交易中，付款方就是通过数字签名来证明自己拥有某一笔比特币，并且，要把这笔比特币转移给指定的收款方。

使用签名是为了验证某个声明确实是由某个人做出的。例如，在付款合同中签名，可以通过验证笔迹的方式核对身份：

而在计算机中，用密码学理论设计的数字签名算法比验证笔迹更加可信。使用数字签名时，每个人都可以自己生成一个秘钥对，这个秘钥对包含一个私钥和一个公钥：私钥被称为Secret Key或者Private Key，私钥必须严格保密，不能泄漏给其他人；公钥被称为Public Key，可以公开给任何人：



当私钥持有人，例如，小明希望对某个消息签名的时候，他可以用自己的私钥对消息进行签名，然后，把消息、签名和自己的公钥发送出去：



其他任何人都可以通过小明的公钥对这个签名进行验证，如果验证通过，可以肯定，该消息是小明发出的。

数字签名算法在电子商务、在线支付这些领域有非常重要的作用：

首先，签名不可伪造，因为私钥只有签名人自己知道，所以其他人无法伪造签名。

其次，消息不可篡改，如果原始消息被人篡改了，那么对签名进行验证将失败。

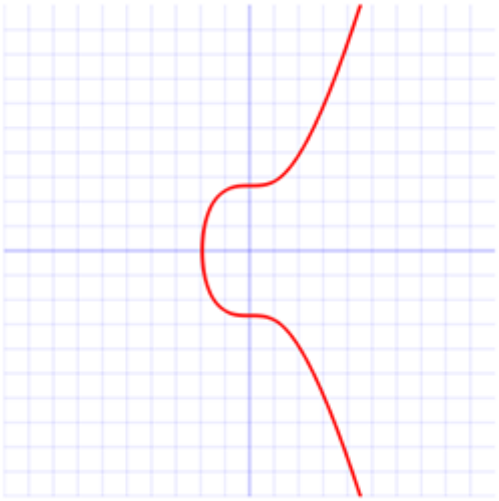
最后，签名不可抵赖。如果对签名进行验证通过了，那么，该消息肯定是由签名人自己发出的，他不能抵赖自己曾经发过这一条消息。

数字签名的三个作用：防伪造，防篡改，防抵赖。

数字签名算法

常用的数字签名算法有：RSA算法，DSA算法和ECDSA算法。比特币采用的签名算法是椭圆曲线签名算法：ECDSA，使用的椭圆曲线是一个已经定义好的标准曲线secp256k1：$y^2 = x^3 + 7$

这条曲线的图像长这样：



比特币采用的ECDSA签名算法需要一个私钥和公钥组成的秘钥对：私钥本质上就是一个1～2256的随机数，公钥是由私钥根据ECDSA算法推算出来的，通过私钥可以很容易推算出公钥，所以不必保存公钥，但是，通过公钥无法反推私钥，只能暴力破解。

比特币的私钥是一个随机的非常大的256位整数。它的上限，确切地说，比2256要稍微小一点：

```
0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036
4140
```

而比特币的公钥是根据私钥推算出的两个256位整数。

如果用银行卡作比较的话，比特币的公钥相当于银行卡卡号，它是两个256位整数：

比特币的私钥相当于银行卡密码，它是一个256位整数：

```
18E14A7B6A307F426A94F8114701E7C8E774E7F9A47E2C2035DB29A206321725
```

银行卡的卡号由银行指定，银行卡的密码可以由用户随时修改。而比特币“卡”和银行卡的不同点在于：密码（实际上是私钥）由用户先确定下来，然后计算出“卡号”（实际上是公钥），即卡号是由密码通过ECDSA算法推导出来的，不能更换密码，因为更换密码实际上相当于创建了一张新卡片。

由于比特币账本是全网公开的，所以，任何人都可以根据公钥查询余额，但是，不知道持卡人是谁。这就是比特币的匿名特性。

如果丢失了私钥，就永远无法花费对应公钥的比特币！

丢失了私钥和忘记银行卡密码不一样，忘记银行卡密码可以拿身份证到银行重新设置一个密码，因为密码是存储在银行的计算机中的，而比特币的P2P网络不存在中央节点，私钥只有持有人自己知道，因此，丢失了私钥，对应的比特币就永远无法花费。如果私钥被盗，黑客就可以花费对应公钥的比特币，并且这是无法追回的。

比特币私钥的安全性在于如何生成一个安全的256位的随机数。*不要试图自己想一个随机数*，而是应当使用编程语言提供的*安全随机数*算法，但绝对不能使用*伪随机数*。

绝不能自己想一个私钥或者使用伪随机数创建私钥！

那有没有可能猜到别人的私钥呢？这是不可能的。2256是一个非常大的数，它已经远远超过了整个银河系的原子总数。绝大多数人对数字大小的直觉是线性增长的，所以256这个数看起来不大，但是指数增长的2256是一个非常巨大的天文数字。

比特币钱包

比特币钱包实际上就是帮助用户管理私钥的软件。因为比特币的钱包是给普通用户使用的，它有几种分类：

- 本地钱包：是把私钥保存在本地计算机硬盘上的钱包软件，如Electrum；
- 手机钱包：和本地钱包类似，但可以直接在手机上运行，如Bitpay；
- 在线钱包：是把私钥委托给第三方在线服务商保存；
- 纸钱包：是指把私钥打印出来保存在纸上；
- 脑钱包：是指把私钥记在自己脑袋里。

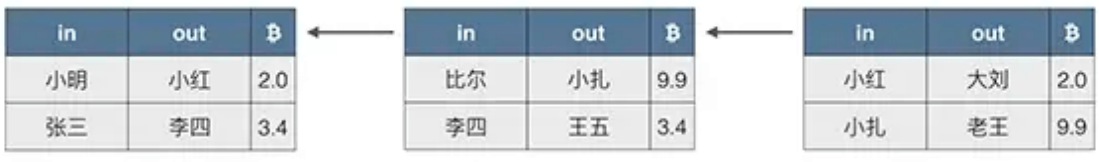
对大多数普通用户来说，想要记住私钥非常困难，所以*强烈不建议使用脑钱包*。

和银行账户不同，比特币网络没有账户的概念，任何人都可以从区块链查询到任意公钥对应的比特币余额，但是，并不知道这些公钥是由谁持有的，也就无法根据用户查询比特币余额。

作为用户，可以生成任意数量的私钥-公钥对，公钥是接收别人转账的地址，而私钥是花费比特币的唯一手段，钱包程序可以帮助用户管理私钥-公钥对。

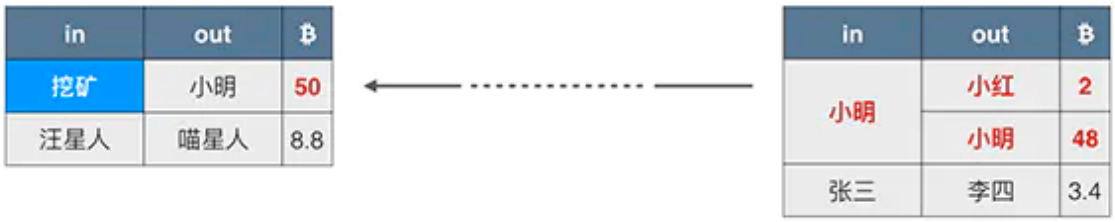
交易

我们再来看记录在区块链上的交易。每个区块都记录了至少一笔交易，一笔交易就是把一定金额的比特币从一个输入转移到一个输出：



例如，小明把两个比特币转移给小红，这笔交易的输入是小明，输出就是小红。实际记录的是双方的公钥地址。

如果小明有50个比特币，他要转给小红两个比特币，那么剩下的48个比特币应该记录在哪？比特币协议规定一个输出必须一次性花完，所以，小明给小红的两个比特币的交易必须表示成：



in	out	฿
挖矿	小明	50
汪星人	喵星人	8.8

in	out	฿
小明	小红	2
	小明	48
张三	李四	3.4

小明给小红2个比特币，同时小明又给自己48个比特币，这48个比特币就是找零。所以，一个交易中，一个输入可以对应多个输出。

当小红有两笔收入时，一笔2.0，一笔1.5，她想给小白转3.5比特币时，就不能单用一笔输出，她必须把两笔钱合起来再花掉，这种情况就是一个交易对应多个输入和1个输出：

in	out	฿
小明	小红	2.0
小军	小红	1.5

in	out	฿
小红	小白	3.5
小红		
张三	李四	3.4

如果存在找零，这笔交易就既包含多个输入也包含多个输出：

in	out	฿
小明	小红	2.0
小军	小红	1.5

in	out	฿
小红	小白	3.0
小红	小红	0.5
张三	李四	3.4

在实际的交易中，输入比输出要稍微大一点点，这个差额就是隐含的交易费用，交易费用会算入当前区块的矿工收入中作为矿工奖励的一部分：

in	out	฿
小明	小红	2.0
小军	小红	1.5

in	out	฿
小红	小白	2.99
小红	小红	0.49
张三	李四	3.4

计算出的交易费用：

```
交易费用 = 输入 - 输出 = (2.0 + 1.5) - (2.99 + 0.49) = 3.5 - 3.48 = 0.02
```

比特币实际的交易记录是由一系列交易构成，每一个交易都包含一个或多个输入，以及一个或多个输出。未花费的输出被称为UTXO（Unspent Transaction Ouptut）。

当我们要简单验证某个交易的时候，例如，对于交易`f36abd`，它记录的输入是`3f96ab`，索引号是`1`（索引号从`0`开始，`0`表示第一个输出，`1`表示第二个输出，以此类推），我们就根据`3f96ab`找到前面已发生的交易，再根据索引号找到对应的输出是0.5个比特币，所以，这笔交易的输入总计是0.5个比特币，输出分别是0.4个比特币和0.09个比特币，隐含的交易费用是0.01个比特币：

tx hash	IN UTXO:#	OUT Addr:฿
3f96ab	UTXO: 1d0c8f#0 SIGN: xxxxxx	1Te395s:฿2.0 1mPvuPA:฿0.5
1784a9	UTXO: 7a95d3#0 SIGN: xxxxxx UTXO: f90bd2#2 SIGN: xxxxxx	1sknWJD:฿1.2 1Sx9RmG:฿2.6
f36abd	UTXO: 3f96ab#1 SIGN: xxxxxx	16Gr9nB:฿0.40 1vg47TL:฿0.09
	...	...

小结

比特币使用数字签名保证零信任的可靠P2P交易：

- 私钥是花费比特币的唯一手段；
- 钱包软件是用来帮助用户管理私钥；
- 所有交易被记录在区块链中，可以通过公钥查询所有交易信息。

私钥

在比特币中，私钥本质上就是一个256位的随机整数。我们以JavaScript为例，演示如何创建比特币私钥。

在JavaScript中，内置的Number类型使用56位表示整数和浮点数，最大可表示的整数最大只有`9007199254740991`。其他语言如Java一般也仅提供64位的整数类型。要表示一个256位的整数，只能使用数组来模拟。bitcoinjs使用bigi这个库来表示任意大小的整数。

下面的代码演示了通过`ECPair`创建一个新的私钥后，表示私钥的整数就是字段`d`，我们把它打印出来：

```
const bitcoin = require('bitcoinjs-lib');
```

```
let keyPair = bitcoin.ECPair.makeRandom();
// 打印私钥:
console.log('private key = ' + keyPair.d);
// 以十六进制打印:
console.log('hex = ' + keyPair.d.toHex());
// 补齐32位:
console.log('hex = ' + keyPair.d.toHex(32));
```

```
private key =
566600809736945060093184178807792298386970933733320546535934583407349469577 3
hex = 7d44782875f3b67d04ee800fbfc0150a67b59f0529329e10acb11be709dfef5d
hex = 7d44782875f3b67d04ee800fbfc0150a67b59f0529329e10acb11be709dfef5d
```

注意：每次运行上述程序，都会生成一个随机的`ECPair`，即每次生成的私钥都是不同的。

256位的整数通常以十六进制表示，使用`toHex(32)`我们可以获得一个固定64字符的十六进制字符串。注意每两个十六进制字符表示一个字节，因此，64字符的十六进制字符串表示的是32字节=256位整数。

想要记住一个256位的整数是非常困难的，并且，如果记错了其中某些位，这个记错的整数仍然是一个*有效的私钥*，因此，比特币有一种对私钥进行编码的方式，这种编码方式就是带校验的Base58编码。

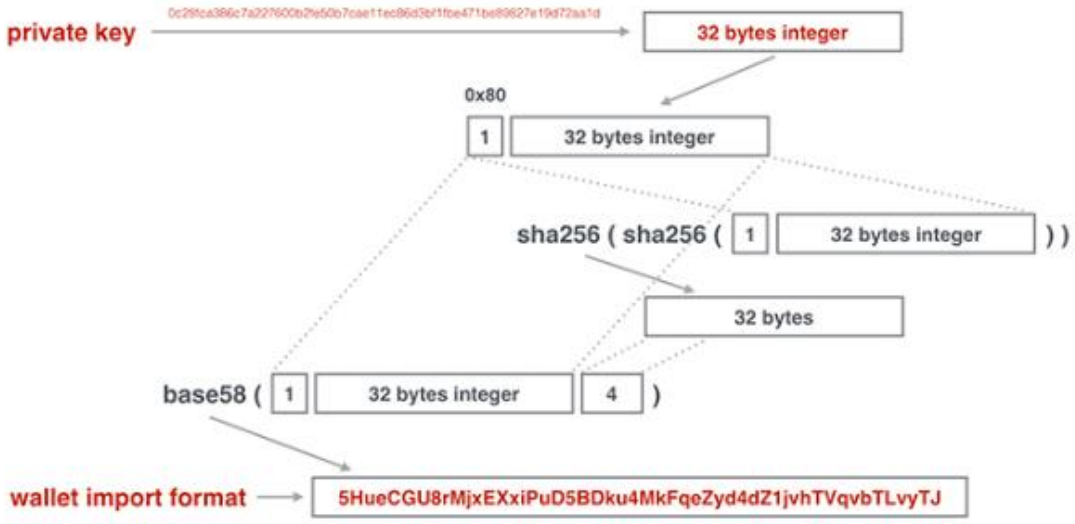
对私钥进行Base58编码有两种方式，一种是非压缩的私钥格式，一种是压缩的私钥格式，它们分别对应非压缩的公钥格式和压缩的公钥格式。

具体地来说，非压缩的私钥格式是指在32字节的私钥前添加一个`0x80`字节前缀，得到33字节的数据，对其计算4字节的校验码，附加到最后，一共得到37字节的数据：

```
0x80          256bit             check
┌──┬──────────────────────────────┬────┐
|1|              32              |  4  |
└──┴──────────────────────────────┴────┘
```

计算校验码非常简单，对其进行两次SHA256，取开头4字节作为校验码。

对这37字节的数据进行Base58编码，得到总是以`5`开头的字符串编码，这个字符串就是我们需要非常小心地保存的私钥地址，又称为钱包导入格式：WIF（Wallet Import Format），整个过程如下图所示：



可以使用wif这个库实现WIF编码：

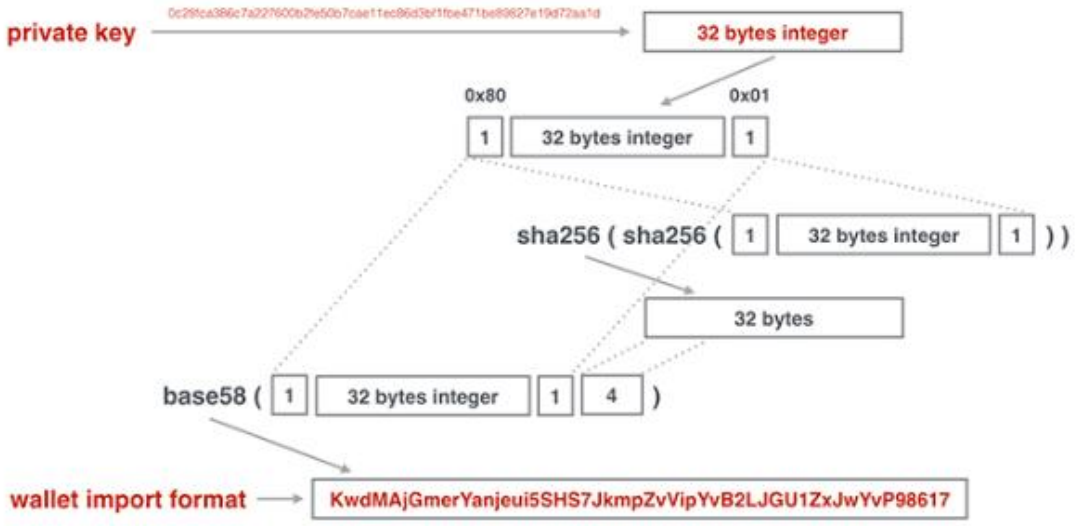
```
// 十六进制表示的私钥:
let privateKey =
'0c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1d';
// 对私钥编码:
let encoded = wif.encode(
        0x80, // 0x80前缀
        Buffer.from(privateKey, 'hex'), // 转换为字节
        false // 非压缩格式
);
console.log(encoded);
```

```
5HueCGU8rMjxEXxiPuD5BDku4MkFqeZyd4dZ1jvhTVqvbTLvyTJ
```

另一种压缩格式的私钥编码方式，与非压缩格式不同的是，压缩的私钥格式会在32字节的私钥前后各添加一个`0x80`字节前缀和`0x01`字节后缀，共34字节的数据，对其计算4字节的校验码，附加到最后，一共得到38字节的数据：

```
0x80          256bit          0x01 check
┌──┬──────────────────────────────┬──┬──────┐
|1|              32              |1|  4   |
└──┴──────────────────────────────┴──┴──────┘
```

对这38字节的数据进行Base58编码，得到总是以`K`或`L`开头的字符串编码，整个过程如下图所示：



通过代码实现压缩格式的WIF编码如下：

```
const wif = require('wif');
```

```
// 十六进制表示的私钥:
let privateKey =
'0c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1d';
// 对私钥编码:
let encoded = wif.encode(
        0x80, // 0x80前缀
        Buffer.from(privateKey, 'hex'), // 转换为字节
        true // 压缩格式
);
console.log(encoded);
```

```
KwdMAjGmerYanjeui5SHS7JkmpZvVipYvB2LJGU1ZxJwYvP98617
```

目前，非压缩的格式几乎已经不使用了。bitcoinjs提供的`ECPair`总是使用压缩格式的私钥表示：

```
const
    bitcoin = require('bitcoinjs-lib'),
    BigInteger = require('bigi');
```

```
let
    priv = '0c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1d',
    d = BigInteger.fromBuffer(Buffer.from(priv, 'hex')),
    keyPair = new bitcoin.ECPair(d);
// 打印WIF格式的私钥:
console.log(keyPair.toWIF());
```

```
KwdMAjGmerYanjeui5SHS7JkmpZvVipYvB2LJGU1ZxJwYvP98617
```

小结

比特币的私钥本质上就是一个256位整数，对私钥进行WIF格式编码可以得到一个带校验的字符串。

使用非压缩格式的WIF是以`5`开头的字符串。

使用压缩格式的WIF是以`K`或`L`开头的字符串。

公钥和地址

公钥

比特币的公钥是根据私钥计算出来的。

私钥本质上是一个256位整数，记作`k`。根据比特币采用的ECDSA算法，可以推导出两个256位整数，记作`(x, y)`，这两个256位整数即为非压缩格式的公钥。

由于ECC曲线的特点，根据非压缩格式的公钥`(x, y)`的`x`实际上也可推算出`y`，但需要知道`y`的奇偶性，因此，可以根据`(x, y)`推算出`x'`，作为压缩格式的公钥。

压缩格式的公钥实际上只保存`x`这一个256位整数，但需要根据`y`的奇偶性在`x`前面添加`02`或`03`前缀，`y`为偶数时添加`02`，否则添加`03`，这样，得到一个1+32=33字节的压缩格式的公钥数据，记作`x'`。

注意压缩格式的公钥和非压缩格式的公钥是可以互相转换的，但均不可反向推导出私钥。

非压缩格式的公钥目前已很少使用，原因是非压缩格式的公钥签名脚本数据会更长。

我们来看看如何根据私钥推算出公钥：

```
const bitcoin = require('bitcoinjs-lib');
```

```
let
    wif = 'KwdMAjGmerYanjeui5SHS7JkmpZvVipYvB2LJGU1ZxJwYvP98617',
    ecPair = bitcoin.ECPair.fromWIF(wif); // 导入私钥
// 计算公钥:
let pubKey = ecPair.getPublicKeyBuffer(); // 返回Buffer对象
console.log(pubKey.toString('hex')); // 02或03开头的压缩公钥
```

```
02d0de0aaeaefad02b8bdc8a01a1b8b11c696bd3d66a2c5f10780d95b7df42645c
```

构造出`ECPair`对象后，即可通过`getPublicKeyBuffer()`以`Buffer`对象返回公钥数据。

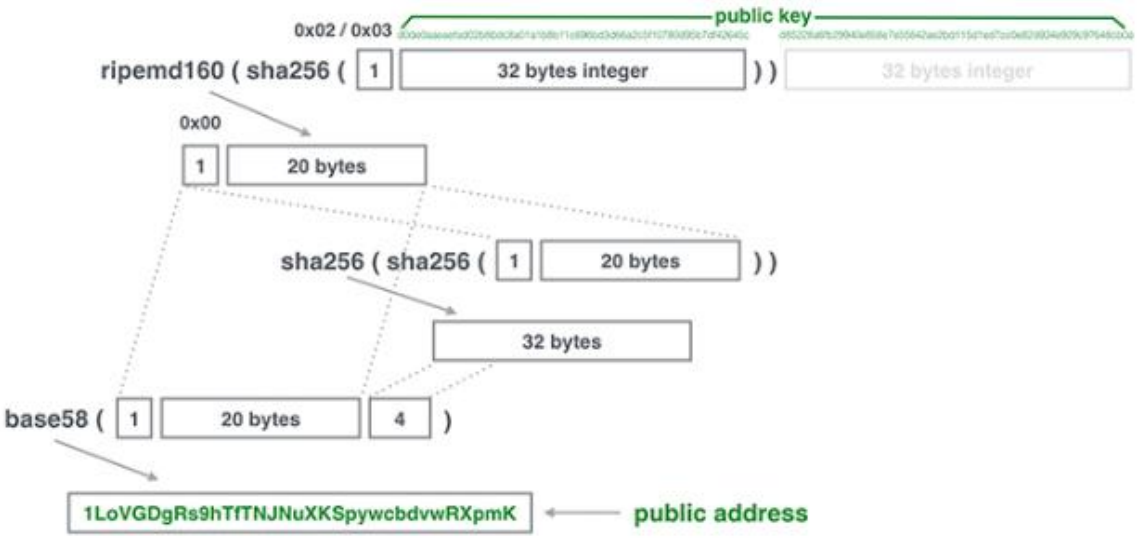
地址

要特别注意，比特币的地址并不是公钥，而是公钥的哈希，即从公钥能推导出地址，但从地址不能反推公钥，因为哈希函数是单向函数。

以压缩格式的公钥为例，从公钥计算地址的方法是，首先对1+32=33字节的公钥数据进行Hash160（即先计算SHA256，再计算RipeMD160），得到20字节的哈希。然后，添加`0x00`前缀，得到1+20=21字节数据，再计算4字节校验码，拼在一起，总计得到1+20+4=25字节数据：

```
0x00       hash160          check
┌──┬──────────────────────┬──────┐
|1|          20          |  4   |
└──┴──────────────────────┴──────┘
```

对上述25字节数据进行Base58编码，得到总是以`1`开头的字符串，该字符串即为比特币地址，整个过程如下：



使用JavaScript实现公钥到地址的编码如下：

```
const bitcoin = require('bitcoinjs-lib');
```

```
let
    publicKey =
'02d0de0aaeaefad02b8bdc8a01a1b8b11c696bd3d66a2c5f10780d95b7df42645c',
    ecPair = bitcoin.ECPair.fromPublicKeyBuffer(Buffer.from(publicKey, 'hex'));
// 导入公钥
// 计算地址:
let address = ecPair.getAddress();
console.log(address); // 1开头的地址
```

```
1LoVGDgRs9hTfTNJNuXKSpywcbdvwRXpmK
```

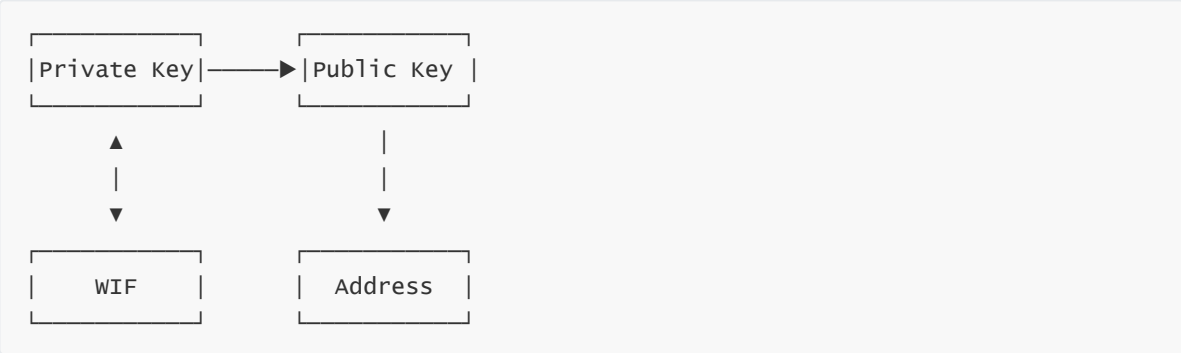
计算地址的时候，不必知道私钥，可以直接从公钥计算地址，即通过`ECPair.fromPublicKeyBuffer`构造一个不带私钥的`ECPair`即可计算出地址。

要注意，对非压缩格式的公钥和压缩格式的公钥进行哈希编码得到的地址，都是以`1`开头的，因此，从地址本身并无法区分出使用的是压缩格式还是非压缩格式的公钥。

以`1`开头的字符串地址即为比特币收款地址，可以安全地公开给任何人。

仅提供地址并不能让其他人得知公钥。通常来说，公开公钥并没有安全风险。实际上，如果某个地址上有对应的资金，要花费该资金，就需要提供公钥。如果某个地址的资金被花费过至少一次，该地址的公钥实际上就公开了。

私钥、公钥以及地址的推导关系如下：



小结

比特币的公钥是根据私钥由ECDSA算法推算出来的，公钥有压缩和非压缩两种表示方法，可互相转换。

比特币的地址是公钥哈希的编码，并不是公钥本身，通过公钥可推导出地址。

通过地址不可推导出公钥，通过公钥不可推导出私钥。

签名

签名算法是使用私钥签名，公钥验证的方法，对一个消息的真伪进行确认。如果一个人持有私钥，他就可以使用私钥对任意的消息进行签名，即通过私钥`sk`对消息`message`进行签名，得到`signature`：

```
signature = sign(message, sk);
```

签名的目的是为了证明，该消息确实是由持有私钥`sk`的人发出的，任何其他人都可以对签名进行验证。验证方法是，由私钥持有人公开对应的公钥`pk`，其他人用公钥`pk`对消息`message`和签名`signature`进行验证：

```
isValid = verify(message, signature, pk);
```

如果验证通过，则可以证明该消息确实是由持有私钥`sk`的人发出的，并且未经过篡改。

数字签名算法在电子商务、在线支付这些领域有非常重要的作用，因为它能通过密码学理论证明：

1. 签名不可伪造，因为私钥只有签名人自己知道，所以其他人无法伪造签名；
2. 消息不可篡改，如果原始消息被人篡改了，对签名进行验证将失败；
3. 签名不可抵赖，如果对签名进行验证通过了，签名人不能抵赖自己曾经发过这一条消息。

简单地说来，数字签名可以防伪造，防篡改，防抵赖。

对消息进行签名，实际上是对消息的哈希进行签名，这样可以使任意长度的消息在签名前先转换为固定长度的哈希数据。对哈希进行签名相当于保证了原始消息的不可伪造性。

我们来看看使用ECDSA如何通过私钥对消息进行签名。关键代码是通过`sign()`方法签名，并获取一个`ECSignature`对象表示签名：

```
const bitcoin = require('bitcoinjs-lib');
```

```
let
    message = 'a secret message!', // 原始消息
    hash = bitcoin.crypto.sha256(message), // 消息哈希
    wif = 'KwdMAjGmerYanjeui5SHS7JkmpZvVipYvB2LJGU1ZxJwYvP98617',
    keyPair = bitcoin.ECPair.fromWIF(wif);
// 用私钥签名:
let signature = keyPair.sign(hash).toDER(); // ECSignature对象
// 打印签名:
console.log('signature = ' + signature.toString('hex'));
// 打印公钥以便验证签名:
console.log('public key = ' + keyPair.getPublicKeyBuffer().toString('hex'));
```

```
signature =
304402205d0b6e817e01e22ba6ab19c0ab9cdbb2dbcd0612c5b8f990431dd0634f5a96530220188b
989017ee7e830de581d4e0d46aa36bbe79537774d56cbe41993b3fd66686
public key = 02d0de0aaeaefad02b8bdc8a01a1b8b11c696bd3d66a2c5f10780d95b7df42645c
```

`ECSignature`对象可序列化为十六进制表示的字符串。

在获得签名、原始消息和公钥的基础上，可以对签名进行验证。验证签名需要先构造一个*不含*私钥的`ECPair`，然后调用`verify()`方法验证签名：

```
const bitcoin = require('bitcoinjs-lib');
```

```
let signAsStr = '304402205d0b6e817e01e22ba6ab19c0'
              + 'ab9cdbb2dbcd0612c5b8f990431dd063'
              + '4f5a96530220188b989017ee7e830de5'
              + '81d4e0d46aa36bbe79537774d56cbe41'
              + '993b3fd66686'

let
    signAsBuffer = Buffer.from(signAsStr, 'hex'),
    signature = bitcoin.ECSignature.fromDER(signAsBuffer), // ECSignature对象
    message = 'a secret message!', // 原始消息
    hash = bitcoin.crypto.sha256(message), // 消息哈希
    pubKeyAsStr =
'02d0de0aaeaefad02b8bdc8a01a1b8b11c696bd3d66a2c5f10780d95b7df42645c',
    pubKeyAsBuffer = Buffer.from(pubKeyAsStr, 'hex'),
    pubKeyOnly = bitcoin.ECPair.fromPublicKeyBuffer(pubKeyAsBuffer); // 从public
key构造ECPair

// 验证签名:
let result = pubKeyOnly.verify(hash, signature);
console.log('Verify result: ' + result);
```

```
Verify result: true
```

注意上述代码只引入了公钥，并没有引入私钥。

修改`signAsStr`、`message`和`pubKeyAsStr`的任意一个变量的任意一个字节，再尝试验证签名，看看是否通过。

比特币对交易数据进行签名和对消息进行签名的原理是一样的，只是格式更加复杂。对交易签名确保了只有持有私钥的人才能够花费对应地址的资金。

小结

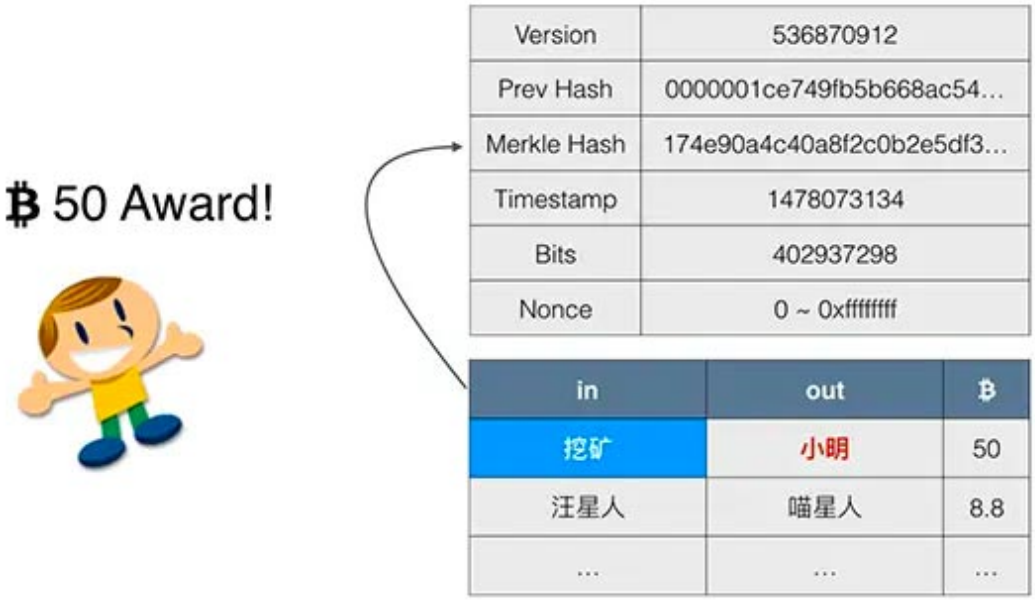
通过私钥可以对消息进行签名，签名可以保证消息防伪造，防篡改，防抵赖。

挖矿原理

在比特币的P2P网络中，有一类节点，它们时刻不停地进行计算，试图把新的交易打包成新的区块并附加到区块链上，这类节点就是矿工。因为每打包一个新的区块，打包该区块的矿工就可以获得一笔比特币作为奖励。所以，打包新区块就被称为挖矿。

比特币的挖矿原理就是一种工作量证明机制。工作量证明POW是英文Proof of Work的缩写。

在讨论POW之前，我们先思考一个问题：在一个新区块中，凭什么是小明得到50个币的奖励，而不是小红或者小军？



当小明成功地打包了一个区块后，除了用户的交易，小明会在第一笔交易记录里写上一笔“挖矿”奖励的交易，从而给自己的地址添加50个比特币。为什么比特币的P2P网络会承认小明打包的区块，并且认可小明得到的区块奖励呢？

因为比特币的挖矿使用了工作量证明机制，小明的区块被认可，是因为他在打包区块的时候，做了一定的工作，而P2P网络的其他节点可以验证小明的工作量。

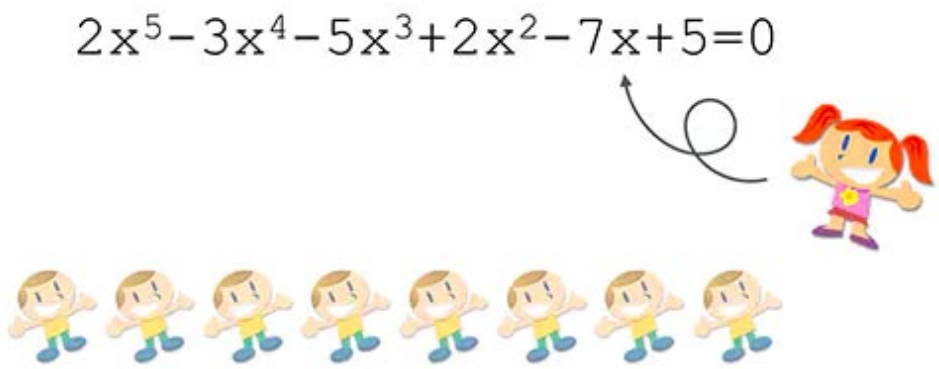
工作量证明

什么是工作量证明？工作量证明是指，证明自己做了一定的工作量。例如，在驾校学习了50个小时。而其他人可以简单地验证该工作量。例如，出示驾照，表示自己确实在驾校学习了一段时间：

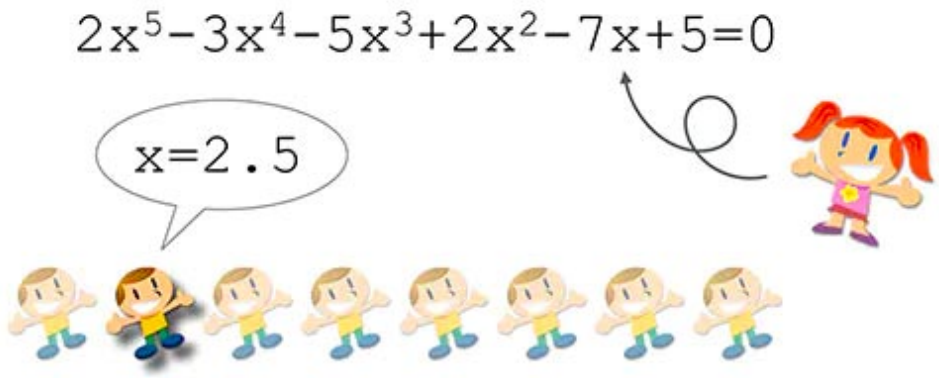
比特币的工作量证明需要归结为计算机计算，也就是数学问题。如何构造一个数学问题来实现工作量证明？我们来看一个简单的例子。

假设某个学校的一个班里，只有一个女生叫小红，其他都是男生。每个男生都想约小红看电影，但是，能实现愿望的只能有一个男生。

到底选哪个男生呢？本着公平原则，小红需要考察每个男生的诚意，考察的方法是，出一道数学题，比如说解方程，谁第一个解出这个方程，谁就有资格陪小红看电影：



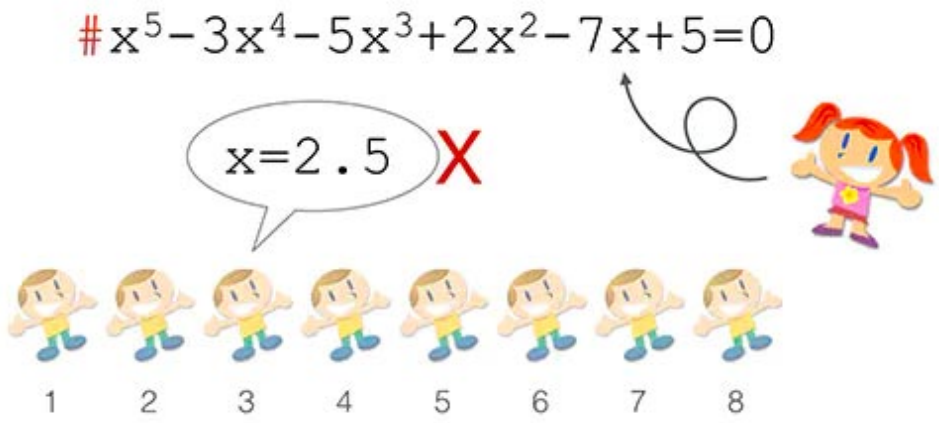
因为解高次方程没有固定的公式，需要进行大量的计算，才能算出正确的结果，这个计算过程就需要一定的工作量。假设小明率先计算出了结果x = 2.5，小红可以简单地验证这个结果是否正确：



可以看出，解方程很困难，但是，验证结果却比较简单。所以，一个有效的工作量证明在于：计算过程非常复杂，需要消耗一定的时间，但是，验证过程相对简单，几乎可以瞬间完成。

现在出现了另一个问题：如果其他人偷看了小明的答案并且抢答了怎么办？

要解决这个问题也很容易，小红可以按照男生的编号，给不同的男生发送不同的方程，方程的第一项的系数就是编号。这样，每个人要解的方程都是不一样的。小明解出的$x=2.5$对于小军来说是无效的，因为小军的编号是3，用小明的结果验证小军的方程是无法通过验证的。



事实上如果某个方程被验证通过了，小红可以直接从方程的第一项系数得知是谁解出的方程。所以，窃取别人的工作量证明的结果是没有用的。

通过工作量证明，可以有效地验证每个人确实都必须花费一定时间做了计算。

在比特币网络中，矿工的挖矿也是一种工作量证明，但是，不能用解多项式方程来实现，因为解多项式方程对人来说很难计算，对计算机来说非常容易，可以在1秒钟以内完成。

要让计算机实现工作量证明，必须找到一种工作量算法，让计算机无法在短时间内算出来。这种算法就是哈希算法。

通过改变区块头部的一个`nonce`字段的值，计算机可以计算出不同的区块哈希值：

直到计算出某个特定的哈希值的时候，计算结束。这个哈希和其他的哈希相比，它的特点是前面有好几个0：

```
hash256(block data, nonce=0) =
291656f37cdcf493c4bb7b926e46fee5c14f9b76aff28f9d00f5cca0e54f376f
hash256(block data, nonce=1) =
f7b2c15c4de7f482edee9e8db7287a6c5def1c99354108ef33947f34d891ea8d
hash256(block data, nonce=2) =
b6eebc5faa4c44d9f5232631f39ddf4211443d819208da110229b644d2a99e12
hash256(block data, nonce=3) =
00aeaaf01166a93a2217fe01021395b066dd3a81daffcd16626c308c644c5246
hash256(block data, nonce=4) =
26d33671119c9180594a91a2f1f0eb08bdd0b595e3724050acb68703dc99f9b5
hash256(block data, nonce=5) =
4e8a3dcab619a7ce5c68e8f4abdc49f98de1a71e58f0ce9a0d95e024cce7c81a
hash256(block data, nonce=6) =
185f634d50b17eba93b260a911ba6dbe9427b72f74f8248774930c0d8588c193
hash256(block data, nonce=7) =
09b19f3d32e3e5771bddc5f0e1ee3c1bac1ba4a85e7b2cc30833a120e41272ed
...
hash256(block data, nonce=124709132) =
00000000fba7277ef31c8ecd1f3fef071cf993485fe5eab08e4f7647f47be95c
```

比特币挖矿的工作量证明原理就是，不断尝试计算区块的哈希，直到计算出一个特定的哈希值，它比难度值要小。

比特币使用的SHA-256算法可以看作对随机输入产生随机输出，例如，我们对字符串`Hello`再加上一个数字计算两次SHA-256，根据数字的不同，得到的哈希是完全无规律的256位随机数：

```
hash256("Hello?") =
????????????????????????????????????????????????????????????????
```

大约计算16次，我们可以在得到的哈希中找到首位是`0`的哈希值，因为首位是0出现的概率是1/16：

```
hash256("Hello1") =
ffb7a43d629d363026b3309586233ab7ffc1054c4f56f43a92f0054870e7ddc9
```

```
hash256("Hello2") =
e085bf19353eb3bd1021661a17cee97181b0b369d8e16c10ffb7b01287a77173
hash256("Hello3") =
c5061965d37b8ed989529bf42eaf8a90c28fa00c3853c7eec586aa8b3922d404
hash256("Hello4") =
42c3104987afc18677179a4a1a984dbfc77e183b414bc6efb00c43b41b213537
hash256("Hello5") =
652dcd7b75d499bcdc61d0c4eda96012e3830557de01426da5b01e214b95cd7a
hash256("Hello6") =
4cc0fbe28abb820085f390d66880ece06297d74d13a6ddbbab3b664582a7a582
hash256("Hello7") =
c3eef05b531b56e79ca38e5f46e6c04f21b0078212a1d8c3500aa38366d9786d
hash256("Hello8") =
cf17d3f38036206cfce464cdcb44d9ccea3f005b7059cff1322c0dd8bf398830
hash256("Hello9") =
1f22981824c821d4e83246e71f207d0e49ad57755889874d43def42af693a077
hash256("Hello10") =
8a1e475d67cfbcea4bcf72d1eee65f15680515f65294c68b203725a9113fa6bf
hash256("Hello11") =
769987b3833f082e31476db0f645f60635fa774d2b92bf0bab00e0a539a2dede
hash256("Hello12") =
c2acd1bb160b1d1e66d769a403e596b174ffab9a39aa7c44d1e670feaa67ab2d
hash256("Hello13") =
dab8b9746f1c0bcf5750e0d878fc17940db446638a477070cf8dca8c3643618a
hash256("Hello14") =
51a575773fccbb5278929c08e788c1ce87e5f44ab356b8760776fd816357f6ff
hash256("Hello15") =
0442e1c38b810f5d3c022fc2820b1d7999149460b83dc680abdebc9c7bd65cae
```

如果我们要找出前两位是`0`的哈希值，理论上需要计算256次，因为`00`出现的概率是162=256，实际计算44次：

```
hash256("Hello44") =
00e477f95283a544ffac7a8efc7decb887f5c073e0f3b43b3797b5dafabb49b5
```

如果我们要找出前3位是`0`的哈希值，理论上需要计算163=4096次，实际计算6591次：

```
hash256("Hello6591") =
0008a883dacb7094d6da1a6cefc6e7cbc13635d024ac15152c4eadba7af8d11c
```

如果我们要找出前4位是`0`的哈希值，理论上需要计算164=6万5千多次，实际计算6万7千多次：

```
hash256("Hello67859") =
00002e4af0b80d706ae749d22247d91d9b1c2e91547d888e5e7a91bcc0982b87
```

如果我们要找出前5位是`0`的哈希值，理论上需要计算165=104万次，实际计算158万次：

```
hash256("Hello1580969") =
00000ca640d95329f965bde016b866e75a3e29e1971cf55ffd1344cdb457930e
```

如果我们要找出前6位是`0`的哈希值，理论上需要计算166=1677万次，实际计算1558万次：

```
hash256("Hello15583041") =
0000009becc5cf8c9e6ba81b1968575a1d15a93112d3bd67f4546f6172ef7e76
```

对于给定难度的SHA-256：假设我们用难度1表示必须算出首位1个`0`，难度2表示必须算出首位两个`0`，难度N表示必须算出首位N个`0`，那么，每增加一个难度，计算量将增加16倍。

对于比特币挖矿来说，就是先给定一个难度值，然后不断变换`nonce`，计算Block Hash，直到找到一个比给定难度值低的Block Hash，就算成功挖矿。

我们用简化的方法来说明难度，例如，必须计算出连续17个`0`开头的哈希值，矿工先确定Prev Hash，Merkle Hash，Timestamp，bits，然后，不断变化`nonce`来计算哈希，直到找出连续17个`0`开头的哈希值。我们可以大致推算一下，17个十六进制的`0`相当于计算了1617次，大约需要计算2.9万亿亿次。

```
17个0 = 1617 = 295147905179352825856 = 2.9万亿亿次
```

实际的难度是根据`bits`由一个公式计算出来，比特币协议要求计算出的区块的哈希值比难度值要小，这个区块才算有效：

```
Difficulty = 402937298
           = 0x18 0455d2
           = 0x0455d2 * 28 * (0x18 - 3)
           = 10629966750428983083584555841596263266471055833986131558 4
           = 0x00000000000000000455d2000000000000000000000000000000000000000000000000
```

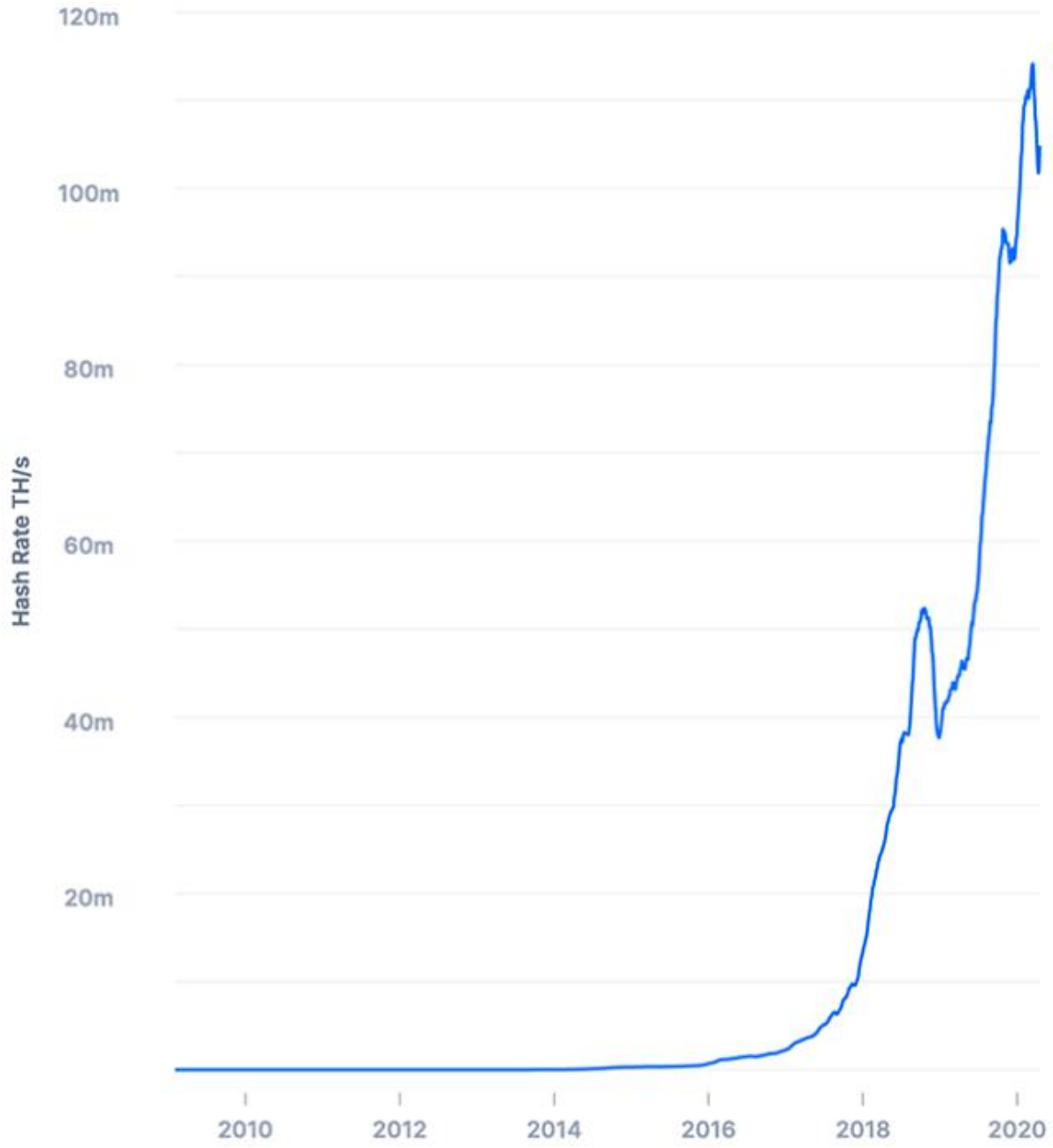
注意，难度值越小，说明哈希值前面的`0`越多，计算难度越大。

比特币网络的难度值是不断变化的，它的难度值保证大约每10分钟产生一个区块，而难度值在每2015个区块调整一次：如果区块平均生成时间小于10分钟，说明全网算力增加，难度值也会增加，如果区块平均生成时间大于10分钟，说明全网算力减少，难度值也会减少。因此，难度值随着全网算力的增减会动态调整。

比特币设计时本来打算每2016个区块调整一次难度，也就是两周一次，但是由于第一版代码的一个bug，实际调整周期是2015个区块。

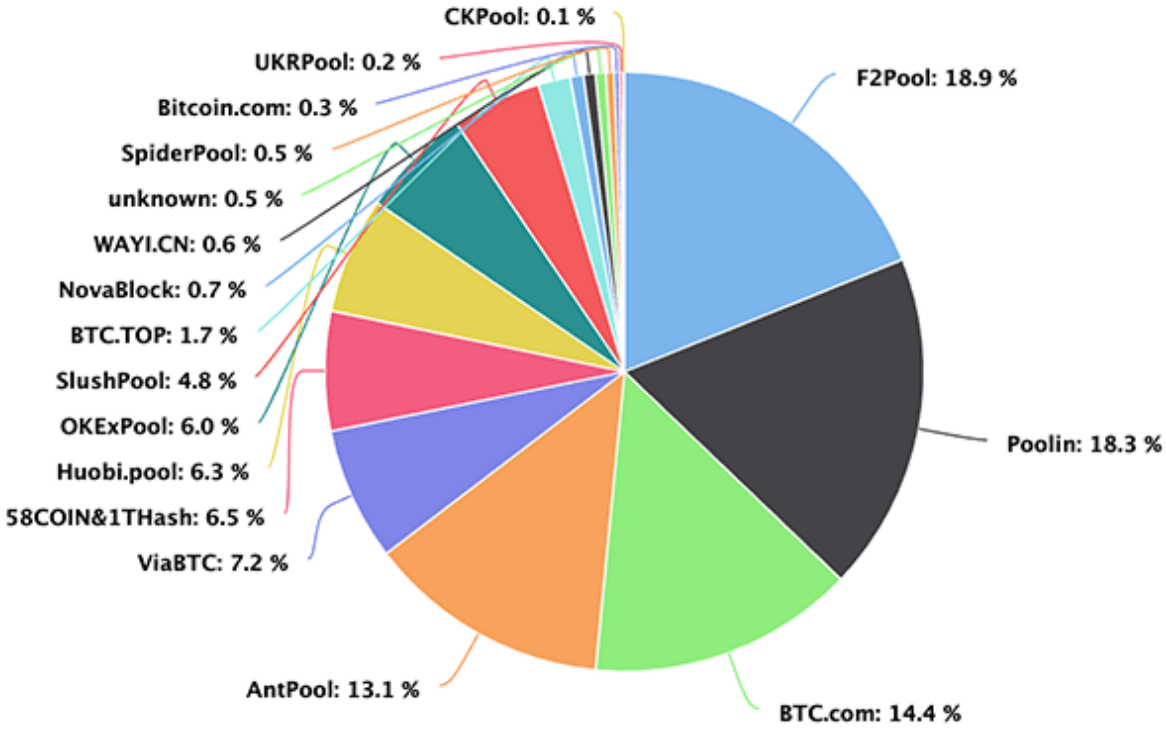
根据比特币每个区块的难度值和产出时间，就可以推算出整个比特币网络的全网算力。

比特币网络的全网算力一直在迅速增加。目前，全网算力已经超过了100EH/每秒，也就是大约每秒钟计算1万亿亿次哈希：



所以比特币的工作量证明被通俗地称之为挖矿。在同一时间，所有矿工都在努力计算下一个区块的哈希。而挖矿难度取决于全网总算力的百分比。举个例子，假设小明拥有全网总算力的百分之一，那么他挖到下一个区块的可能性就是1%，或者说，每挖出100个区块，大约有1个就是小明挖的。

由于目前全网算力超过了100EH/s，而单机CPU算力不过几M，GPU算力也不过1G，所以，单机挖矿的成功率几乎等于0。比特币挖矿已经从早期的CPU、GPU发展到专用的ASIC芯片构建的矿池挖矿。



当某个矿工成功找到特定哈希的新区块后，他会立刻向全网广播该区块。其他矿工在收到新区块后，会对新区块进行验证，如果有效，就把它添加到区块链的尾部。同时说明，在本轮工作量证明的竞争中，这个矿工胜出，而其他矿工都失败了。失败的矿工会抛弃自己当前正在计算还没有算完的区块，转而开始计算下一个区块，进行下一轮工作量证明的竞争。

为什么区块可以安全广播？因为Merkle Hash锁定了该区块的所有交易，而该区块的第一个coinbase交易输出地址是该矿工地址。每个矿工在挖矿时产生的区块数据都是不同的，所以无法窃取别人的工作量。

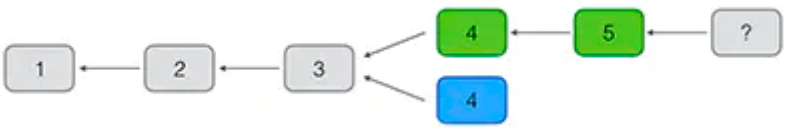
比特币总量被限制为约2100万个比特币，初始挖矿奖励为每个区块50个比特币，以后每4年减半。

共识算法

如果两个矿工在同一时间各自找到了有效区块，注意，这两个区块是不同的，因为coinbase交易不同，所以Merkle Hash不同，区块哈希也不同。但它们只要符合难度值，就都是有效的。这个时候，网络上的其他矿工应该接收哪个区块并添加到区块链的末尾呢？答案是，都有可能。

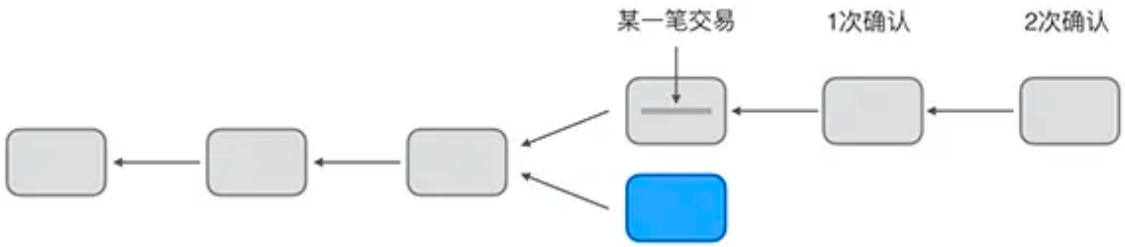
通常，矿工接收先收到的有效区块，由于P2P网络广播的顺序是不确定的，不同的矿工先收到的区块是有可能的不同的。这个时候，我们说区块发生了分叉：

在分叉的情况下，有的矿工在绿色的分叉上继续挖矿，有的矿工在蓝色的分叉上继续挖矿：



但是最终，总有一个分叉首先挖到后续区块，这个时候，由于比特币网络采用最长分叉的共识算法，绿色分叉胜出，蓝色分叉被废弃，整个网络上的所有矿工又会继续在最长的链上继续挖矿。

由于区块链虽然最终会保持数据一致，但是，一个交易可能被打包到一个后续被孤立的区块中。所以，要确认一个交易被永久记录到区块链中，需要对交易进行确认。如果后续的区块被追加到区块链上，实际上就会对原有的交易进行确认，因为链越长，修改的难度越大。一般来说，经过6个区块确认的交易几乎是不可能被修改的。



小结

比特币挖矿是一种带经济激励的工作量证明机制；

工作量证明保证了修改区块链需要极高的成本，从而使得区块链的不可篡改特性得到保护；

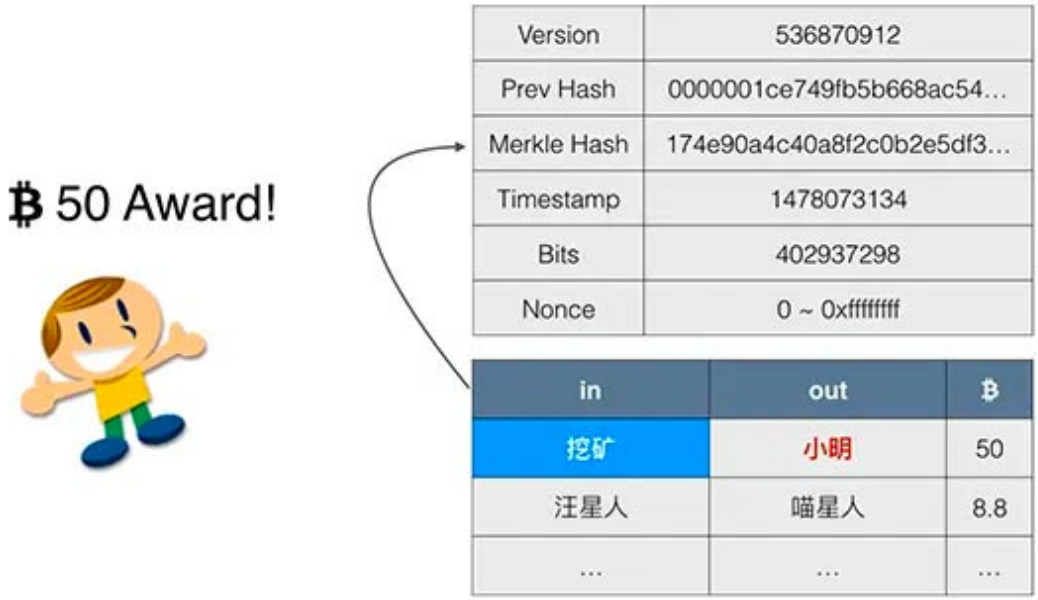
比特币的网络安全实际上就是依靠强大的算力保障的。

挖矿原理

在比特币的P2P网络中，有一类节点，它们时刻不停地进行计算，试图把新的交易打包成新的区块并附加到区块链上，这类节点就是矿工。因为每打包一个新的区块，打包该区块的矿工就可以获得一笔比特币作为奖励。所以，打包新区块就被称为挖矿。

比特币的挖矿原理就是一种工作量证明机制。工作量证明POW是英文Proof of Work的缩写。

在讨论POW之前，我们先思考一个问题：在一个新区块中，凭什么是小明得到50个币的奖励，而不是小红或者小军？



当小明成功地打包了一个区块后，除了用户的交易，小明会在第一笔交易记录里写上一笔“挖矿”奖励的交易，从而给自己的地址添加50个比特币。为什么比特币的P2P网络会承认小明打包的区块，并且认可小明得到的区块奖励呢？

因为比特币的挖矿使用了工作量证明机制，小明的区块被认可，是因为他在打包区块的时候，做了一定的工作，而P2P网络的其他节点可以验证小明的工作量。

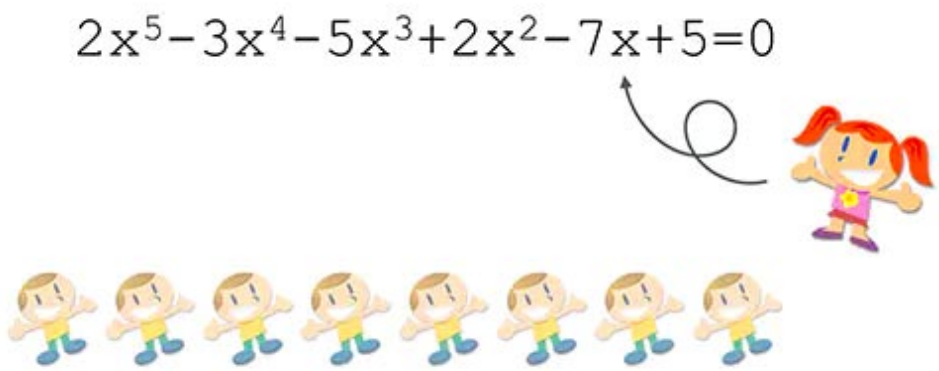
工作量证明

什么是工作量证明？工作量证明是指，证明自己做了一定的工作量。例如，在驾校学习了50个小时。而其他人可以简单地验证该工作量。例如，出示驾照，表示自己确实在驾校学习了一段时间：

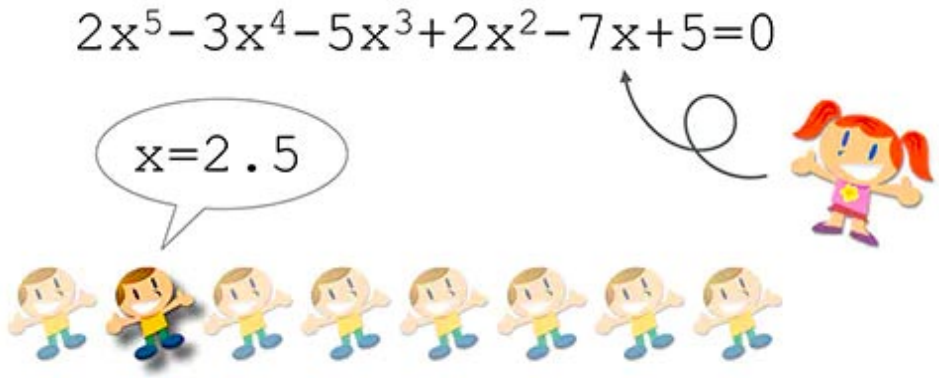
比特币的工作量证明需要归结为计算机计算，也就是数学问题。如何构造一个数学问题来实现工作量证明？我们来看一个简单的例子。

假设某个学校的一个班里，只有一个女生叫小红，其他都是男生。每个男生都想约小红看电影，但是，能实现愿望的只能有一个男生。

到底选哪个男生呢？本着公平原则，小红需要考察每个男生的诚意，考察的方法是，出一道数学题，比如说解方程，谁第一个解出这个方程，谁就有资格陪小红看电影：



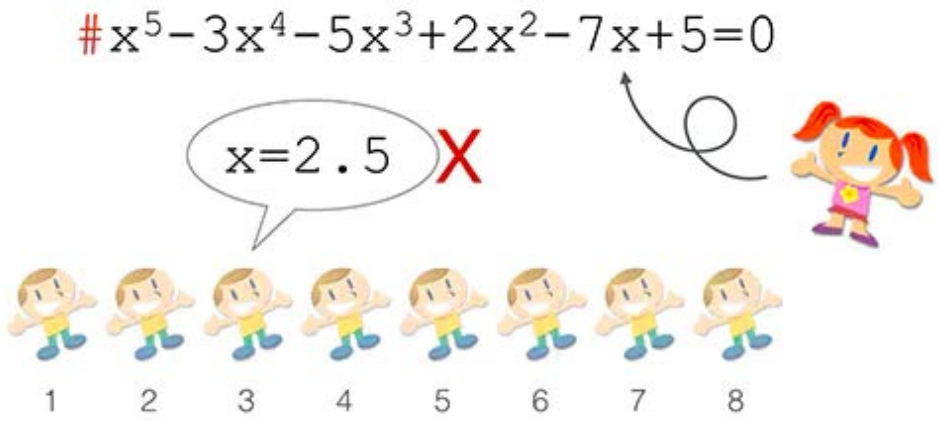
因为解高次方程没有固定的公式，需要进行大量的计算，才能算出正确的结果，这个计算过程就需要一定的工作量。假设小明率先计算出了结果x = 2.5，小红可以简单地验证这个结果是否正确：



可以看出，解方程很困难，但是，验证结果却比较简单。所以，一个有效的工作量证明在于：计算过程非常复杂，需要消耗一定的时间，但是，验证过程相对简单，几乎可以瞬间完成。

现在出现了另一个问题：如果其他人偷看了小明的答案并且抢答了怎么办？

要解决这个问题也很容易，小红可以按照男生的编号，给不同的男生发送不同的方程，方程的第一项的系数就是编号。这样，每个人要解的方程都是不一样的。小明解出的$x=2.5$对于小军来说是无效的，因为小军的编号是3，用小明的结果验证小军的方程是无法通过验证的。



事实上如果某个方程被验证通过了，小红可以直接从方程的第一项系数得知是谁解出的方程。所以，窃取别人的工作量证明的结果是没有用的。

通过工作量证明，可以有效地验证每个人确实都必须花费一定时间做了计算。

在比特币网络中，矿工的挖矿也是一种工作量证明，但是，不能用解多项式方程来实现，因为解多项式方程对人来说很难计算，对计算机来说非常容易，可以在1秒钟以内完成。

要让计算机实现工作量证明，必须找到一种工作量算法，让计算机无法在短时间内算出来。这种算法就是哈希算法。

通过改变区块头部的一个`nonce`字段的值，计算机可以计算出不同的区块哈希值：

直到计算出某个特定的哈希值的时候，计算结束。这个哈希和其他的哈希相比，它的特点是前面有好几个0：

```
hash256(block data, nonce=0) =
291656f37cdcf493c4bb7b926e46fee5c14f9b76aff28f9d00f5cca0e54f376f
hash256(block data, nonce=1) =
f7b2c15c4de7f482edee9e8db7287a6c5def1c99354108ef33947f34d891ea8d
hash256(block data, nonce=2) =
b6eebc5faa4c44d9f5232631f39ddf4211443d819208da110229b644d2a99e12
hash256(block data, nonce=3) =
00aeaaf01166a93a2217fe01021395b066dd3a81daffcd16626c308c644c5246
hash256(block data, nonce=4) =
26d33671119c9180594a91a2f1f0eb08bdd0b595e3724050acb68703dc99f9b5
hash256(block data, nonce=5) =
4e8a3dcab619a7ce5c68e8f4abdc49f98de1a71e58f0ce9a0d95e024cce7c81a
hash256(block data, nonce=6) =
185f634d50b17eba93b260a911ba6dbe9427b72f74f8248774930c0d8588c193
hash256(block data, nonce=7) =
09b19f3d32e3e5771bddc5f0e1ee3c1bac1ba4a85e7b2cc30833a120e41272ed
...
hash256(block data, nonce=124709132) =
00000000fba7277ef31c8ecd1f3fef071cf993485fe5eab08e4f7647f47be95c
```

比特币挖矿的工作量证明原理就是，不断尝试计算区块的哈希，直到计算出一个特定的哈希值，它比难度值要小。

比特币使用的SHA-256算法可以看作对随机输入产生随机输出，例如，我们对字符串`Hello`再加上一个数字计算两次SHA-256，根据数字的不同，得到的哈希是完全无规律的256位随机数：

```
hash256("Hello?") =
????????????????????????????????????????????????????????????????
```

大约计算16次，我们可以在得到的哈希中找到首位是`0`的哈希值，因为首位是0出现的概率是1/16：

```
hash256("Hello1") =
ffb7a43d629d363026b3309586233ab7ffc1054c4f56f43a92f0054870e7ddc9
hash256("Hello2") =
e085bf19353eb3bd1021661a17cee97181b0b369d8e16c10ffb7b01287a77173
```

```
hash256("Hello3") =
c5061965d37b8ed989529bf42eaf8a90c28fa00c3853c7eec586aa8b3922d404
hash256("Hello4") =
42c3104987afc18677179a4a1a984dbfc77e183b414bc6efb00c43b41b213537
hash256("Hello5") =
652dcd7b75d499bcdc61d0c4eda96012e3830557de01426da5b01e214b95cd7a
hash256("Hello6") =
4cc0fbe28abb820085f390d66880ece06297d74d13a6ddbbab3b664582a7a582
hash256("Hello7") =
c3eef05b531b56e79ca38e5f46e6c04f21b0078212a1d8c3500aa38366d9786d
hash256("Hello8") =
cf17d3f38036206cfce464cdcb44d9ccea3f005b7059cff1322c0dd8bf398830
hash256("Hello9") =
1f22981824c821d4e83246e71f207d0e49ad57755889874d43def42af693a077
hash256("Hello10") =
8a1e475d67cfbcea4bcf72d1eee65f15680515f65294c68b203725a9113fa6bf
hash256("Hello11") =
769987b3833f082e31476db0f645f60635fa774d2b92bf0bab00e0a539a2dede
hash256("Hello12") =
c2acd1bb160b1d1e66d769a403e596b174ffab9a39aa7c44d1e670feaa67ab2d
hash256("Hello13") =
dab8b9746f1c0bcf5750e0d878fc17940db446638a477070cf8dca8c3643618a
hash256("Hello14") =
51a575773fccbb5278929c08e788c1ce87e5f44ab356b8760776fd816357f6ff
hash256("Hello15") =
0442e1c38b810f5d3c022fc2820b1d7999149460b83dc680abdebc9c7bd65cae
```

如果我们要找出前两位是`0`的哈希值，理论上需要计算256次，因为`00`出现的概率是162=256，实际计算44次：

```
hash256("Hello44") =
00e477f95283a544ffac7a8efc7decb887f5c073e0f3b43b3797b5dafabb49b5
```

如果我们要找出前3位是`0`的哈希值，理论上需要计算163=4096次，实际计算6591次：

```
hash256("Hello6591") =
0008a883dacb7094d6da1a6cefc6e7cbc13635d024ac15152c4eadba7af8d11c
```

如果我们要找出前4位是`0`的哈希值，理论上需要计算164=6万5千多次，实际计算6万7千多次：

```
hash256("Hello67859") =
00002e4af0b80d706ae749d22247d91d9b1c2e91547d888e5e7a91bcc0982b87
```

如果我们要找出前5位是`0`的哈希值，理论上需要计算165=104万次，实际计算158万次：

```
hash256("Hello1580969") =
00000ca640d95329f965bde016b866e75a3e29e1971cf55ffd1344cdb457930e
```

如果我们要找出前6位是`0`的哈希值，理论上需要计算166=1677万次，实际计算1558万次：

```
hash256("Hello15583041") =
0000009becc5cf8c9e6ba81b1968575a1d15a93112d3bd67f4546f6172ef7e76
```

对于给定难度的SHA-256：假设我们用难度1表示必须算出首位1个`0`，难度2表示必须算出首位两个`0`，难度N表示必须算出首位N个`0`，那么，每增加一个难度，计算量将增加16倍。

对于比特币挖矿来说，就是先给定一个难度值，然后不断变换`nonce`，计算Block Hash，直到找到一个比给定难度值低的Block Hash，就算成功挖矿。

我们用简化的方法来说明难度，例如，必须计算出连续17个`0`开头的哈希值，矿工先确定Prev Hash，Merkle Hash，Timestamp，bits，然后，不断变化`nonce`来计算哈希，直到找出连续17个`0`开头的哈希值。我们可以大致推算一下，17个十六进制的`0`相当于计算了1617次，大约需要计算2.9万亿亿次。

```
17个0 = 1617 = 295147905179352825856 = 2.9万亿亿次
```

实际的难度是根据`bits`由一个公式计算出来，比特币协议要求计算出的区块的哈希值比难度值要小，这个区块才算有效：

```
Difficulty = 402937298
           = 0x18 0455d2
           = 0x0455d2 * 28 * (0x18 - 3)
           = 10629966750428983083584555841596263266471055833986131558 4
           = 0x00000000000000000455d2000000000000000000000000000000000000000000000000
```

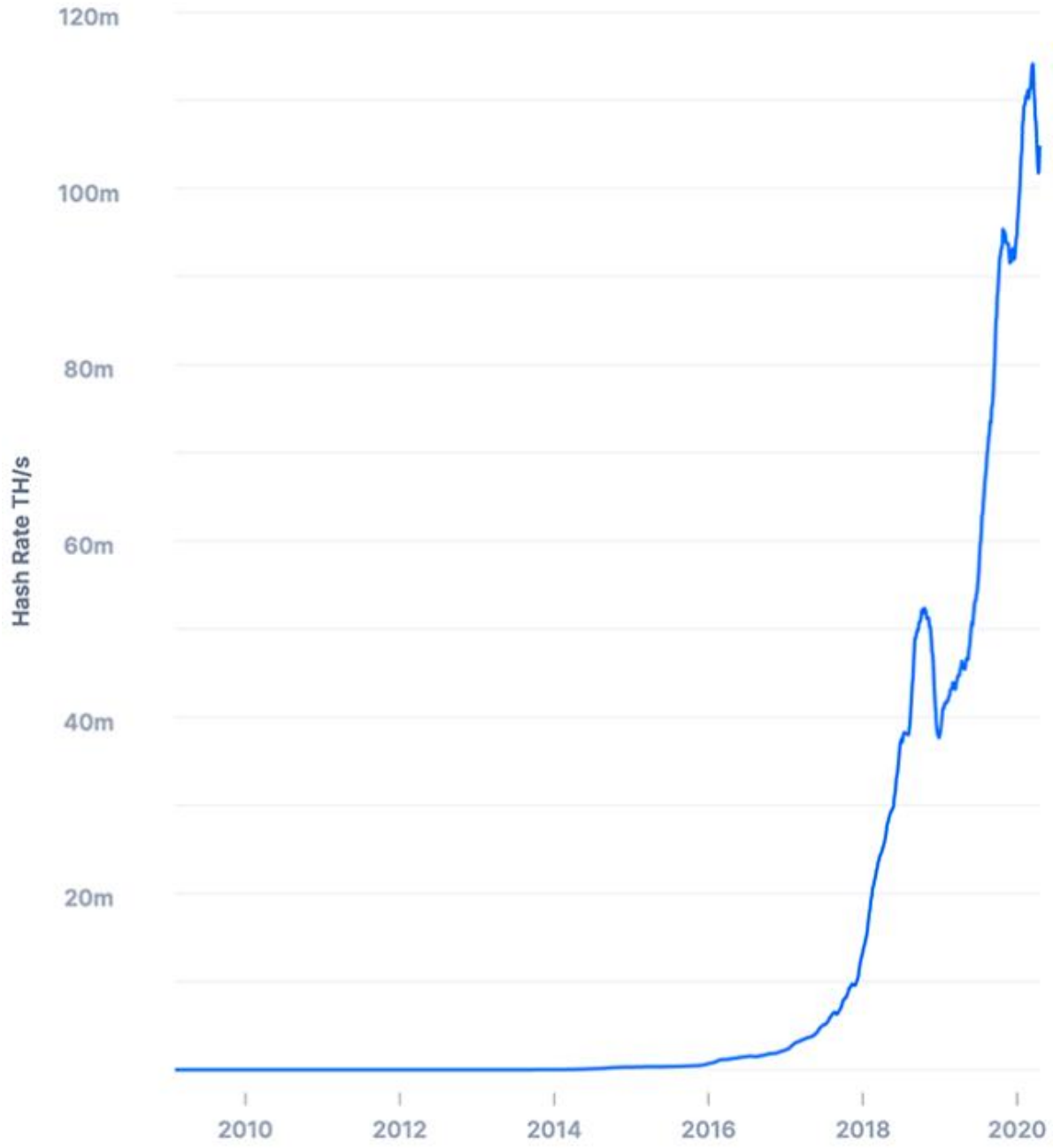
注意，难度值越小，说明哈希值前面的`0`越多，计算难度越大。

比特币网络的难度值是不断变化的，它的难度值保证大约每10分钟产生一个区块，而难度值在每2015个区块调整一次：如果区块平均生成时间小于10分钟，说明全网算力增加，难度值也会增加，如果区块平均生成时间大于10分钟，说明全网算力减少，难度值也会减少。因此，难度值随着全网算力的增减会动态调整。

比特币设计时本来打算每2016个区块调整一次难度，也就是两周一次，但是由于第一版代码的一个bug，实际调整周期是2015个区块。

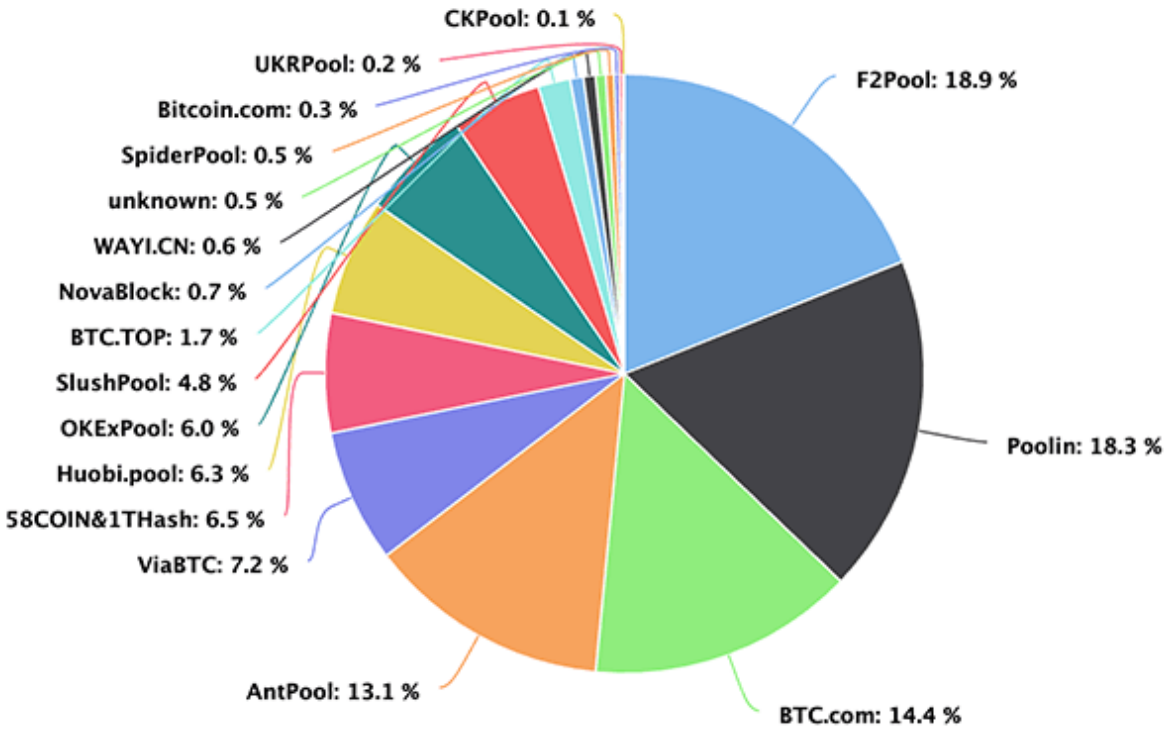
根据比特币每个区块的难度值和产出时间，就可以推算出整个比特币网络的全网算力。

比特币网络的全网算力一直在迅速增加。目前，全网算力已经超过了100EH/每秒，也就是大约每秒钟计算1万亿亿次哈希：



所以比特币的工作量证明被通俗地称之为挖矿。在同一时间，所有矿工都在努力计算下一个区块的哈希。而挖矿难度取决于全网总算力的百分比。举个例子，假设小明拥有全网总算力的百分之一，那么他挖到下一个区块的可能性就是1%，或者说，每挖出100个区块，大约有1个就是小明挖的。

由于目前全网算力超过了100EH/s，而单机CPU算力不过几M，GPU算力也不过1G，所以，单机挖矿的成功率几乎等于0。比特币挖矿已经从早期的CPU、GPU发展到专用的ASIC芯片构建的矿池挖矿。



当某个矿工成功找到特定哈希的新区块后，他会立刻向全网广播该区块。其他矿工在收到新区块后，会对新区块进行验证，如果有效，就把它添加到区块链的尾部。同时说明，在本轮工作量证明的竞争中，这个矿工胜出，而其他矿工都失败了。失败的矿工会抛弃自己当前正在计算还没有算完的区块，转而开始计算下一个区块，进行下一轮工作量证明的竞争。

为什么区块可以安全广播？因为Merkle Hash锁定了该区块的所有交易，而该区块的第一个coinbase交易输出地址是该矿工地址。每个矿工在挖矿时产生的区块数据都是不同的，所以无法窃取别人的工作量。

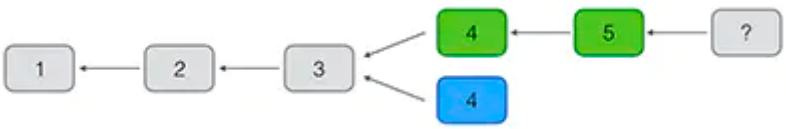
比特币总量被限制为约2100万个比特币，初始挖矿奖励为每个区块50个比特币，以后每4年减半。

共识算法

如果两个矿工在同一时间各自找到了有效区块，注意，这两个区块是不同的，因为coinbase交易不同，所以Merkle Hash不同，区块哈希也不同。但它们只要符合难度值，就都是有效的。这个时候，网络上的其他矿工应该接收哪个区块并添加到区块链的末尾呢？答案是，都有可能。

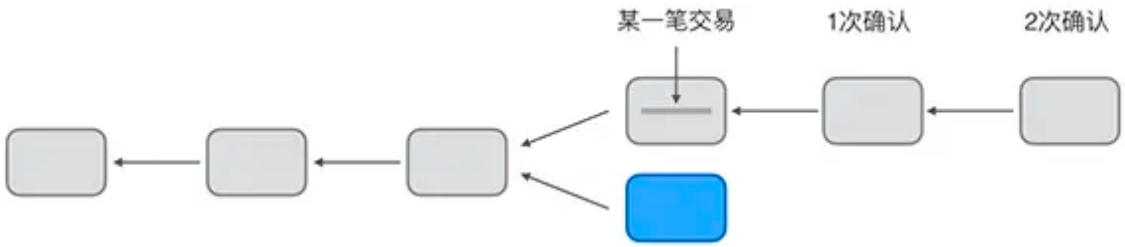
通常，矿工接收先收到的有效区块，由于P2P网络广播的顺序是不确定的，不同的矿工先收到的区块是有可能的不同的。这个时候，我们说区块发生了分叉：

在分叉的情况下，有的矿工在绿色的分叉上继续挖矿，有的矿工在蓝色的分叉上继续挖矿：



但是最终，总有一个分叉首先挖到后续区块，这个时候，由于比特币网络采用最长分叉的共识算法，绿色分叉胜出，蓝色分叉被废弃，整个网络上的所有矿工又会继续在最长的链上继续挖矿。

由于区块链虽然最终会保持数据一致，但是，一个交易可能被打包到一个后续被孤立的区块中。所以，要确认一个交易被永久记录到区块链中，需要对交易进行确认。如果后续的区块被追加到区块链上，实际上就会对原有的交易进行确认，因为链越长，修改的难度越大。一般来说，经过6个区块确认的交易几乎是不可能被修改的。



小结

比特币挖矿是一种带经济激励的工作量证明机制；

工作量证明保证了修改区块链需要极高的成本，从而使得区块链的不可篡改特性得到保护；

比特币的网络安全实际上就是依靠强大的算力保障的。

可编程支付原理

比特币的所有交易的信息都被记录在比特币的区块链中，任何用户都可以通过公钥查询到某个交易的输入和输出金额。当某个用户希望花费一个输出时，例如，小明想要把某个公钥地址的输出支付给小红，他就需要使用自己的私钥对这笔交易进行签名，而矿工验证这笔交易的签名是有效的之后，就会把这笔交易打包到区块中，从而使得这笔交易被确认。

但比特币的支付实际上并不是直接支付到对方的地址，而是一个脚本，这个脚本的意思是：谁能够提供另外一个脚本，让这两个脚本能顺利执行通过，谁就能花掉这笔钱：

```
FROM: UTXO Hash#index
AMOUNT: 0.5 btc
TO: OP_DUP OP_HASH160 <address> OP_EQUALVERIFY OP_CHECKSIG
```

所以，比特币交易的输出是一个锁定脚本，而下一个交易的输入是一个解锁脚本。必须提供一个解锁脚本，让锁定脚本正确运行，那么该输入有效，就可以花费该输出。

我们以真实的比特币交易为例，某个交易的某个输出脚本是`76a914dc...489c88ac`这样的二进制数据，注意这里的二进制数据是用十六进制表示的，而花费该输出的某个交易的输入脚本是`48304502...14cf740f`这样的二进制数据，也是十六进制表示的：

```
    ┌──────────────────────────────────────────────────────────────────────┐
    |tx:ada3f1f426ad46226fdce0ec8f795dcbd05780fd17f76f5dcf67cfbfd35d54de   |
    ├──────────────────────────────────┬───────────────────────────────────┤
    |                                  |1M6Bzo23yqad8YwzTeRapGXQ76Pb9RRJYJ|─┐
    |                                  ├───────────────────────────────────┤ |
    |                                  |18gJ3jeLdMnr9g3EcbRzXwNssYEN5yFHKE| |
    |3JXRVxhrk2o9f4w3cQchBLwUeegJBj6BEp|├──────────────────────────────────┤ |
    |                                  |1A5Mp8jHcMJEqZUmcsbmtqXfsiGdWYmp6y| |
    |                                  ├──────────────────────────────────┤ |
    |                                  |3JXRVxhrk2o9f4w3cQchBLwUeegJBj6BEp| |
    └──────────────────────────────────┴──────────────────────────────────┘ |
┌──────────────────────────────────────────────────────────────────────────┘
|   script: 76a914dc5dc65c7e6cc3c404c6dd79d83b22b2fe9f489c88ac
|
|   ┌──────────────────────────────────────────────────────────────────────┐
|   |tx:55142366a67beda9d3ba9bfbd6166e8e95c4931a2b44e5b44b5685597e4c8774   |
|   ├──────────────────────────────────┬───────────────────────────────────┤
└──>|1M6Bzo23yqad8YwzTeRapGXQ76Pb9RRJYJ|13Kb2ykVGpNTJbxwnrfoyZAwgd4ZpXHv2q|
    └──────────────────────────────────┴───────────────────────────────────┘
    script: 4830450221008ecb5ab06e62a67e320880db70ee8a7020503a055d7c45b7
            3dcc41adf01ea9f602203a0d8f4314342636a6a473fc0b4dd4e49b62be28
            8f0a4d5a23a8f488a768fa9b012103dd8763f8c3db6b77bee743ddafd33c
            969a99cde9278deb441b09ad7c14cf740f
```

我们先来看锁定脚本，锁定脚本的第一个字节`76`翻译成比特币脚本的字节码就是`OP_DUP`，`a9`翻译成比特币脚本的字节码就是`OP_HASH160`。`14`表示这是一个20字节的数据，注意十六进制的`14`换算成十进制是20，于是我们得到20字节的数据。最后两个字节，`88`表示字节码`OP_EQUALVERIFY`，`ac`表示字节码`OP_CHECKSIG`，所以整个锁定脚本是：

```
        OP_DUP 76
    OP_HASH160 a9
          DATA 14 (dc5dc65c...fe9f489c)
OP_EQUALVERIFY 88
   OP_CHECKSIG ac
```

我们再来看解锁脚本。解锁脚本的第一个字节`48`表示一个72字节长度的数据，因为十六进制的`48`换算成十进制是72。接下来的字节`21`表示一个33字节长度的数据。因此，该解锁脚本实际上只有两个数据。

```
DATA 48 (30450221...68fa9b01)
DATA 21 (03dd8763...14cf740f)
```

接下来，我们就需要验证这个交易是否有效。要验证这个交易，首先，我们要把解锁脚本和锁定脚本拼到一块，然后，开始执行这个脚本：

```
          DATA 48 (30450221...68fa9b01)
          DATA 21 (03dd8763...14cf740f)
        OP_DUP 76
    OP_HASH160 a9
          DATA 14 (dc5dc65c...fe9f489c)
OP_EQUALVERIFY 88
   OP_CHECKSIG ac
```

比特币脚本是一种基于栈结构的编程语言，所以，我们要先准备一个空栈，用来执行比特币脚本。然后，我们执行第一行代码，由于第一行代码是数据，所以直接把数据压栈：

```
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
├────────────────────┤
|30450221...68fa9b01|
└────────────────────┘
```

紧接着执行第二行代码，第二行代码也是数据，所以直接把数据压栈：

```
|                    |
|                    |
|                    |
|                    |
├────────────────────┤
|03dd8763...14cf740f|
├────────────────────┤
|30450221...68fa9b01|
└────────────────────┘
```

接下来执行`OP_DUP`指令，这条指令会把栈顶的元素复制一份，因此，我们现在的栈里面一共有3份数据：

```
|                   |
|                   |
├───────────────────┤
|03dd8763...14cf740f|
├───────────────────┤
|03dd8763...14cf740f|
├───────────────────┤
|30450221...68fa9b01|
└───────────────────┘
```

然后，执行`OP_HASH160`指令，这条指令会计算栈顶数据的hash160，也就是先计算SHA-256，再计算RipeMD160。对十六进制数据`03dd8763f8c3db6b77bee743ddafd33c969a99cde9278deb441b09ad7c14cf740f`计算hash160后得到结果`dc5dc65c7e6cc3c404c6dd79d83b22b2fe9f489c`，然后用结果替换栈顶数据：

```
|                   |
|                   |
├───────────────────┤
|dc5dc65c...fe9f489c|
├───────────────────┤
|03dd8763...14cf740f|
├───────────────────┤
|30450221...68fa9b01|
└───────────────────┘
```

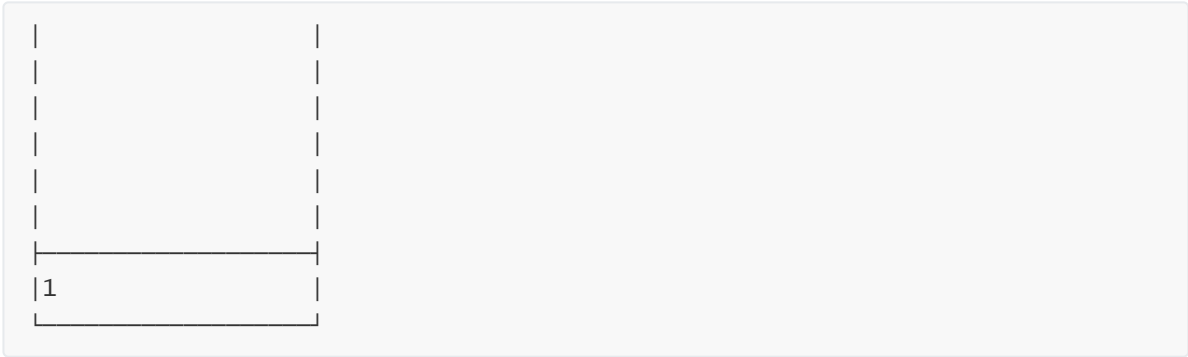
接下来的指令是一条数据，所以直接压栈：

```
├───────────────────┤
|dc5dc65c...fe9f489c|
├───────────────────┤
|dc5dc65c...fe9f489c|
├───────────────────┤
|03dd8763...14cf740f|
├───────────────────┤
|30450221...68fa9b01|
└───────────────────┘
```

然后，执行`OP_EQUALVERIFY`指令，它比较栈顶两份数据是否相同，如果相同，则验证通过，脚本将继续执行，如果不同，则验证失败，整个脚本就执行失败了。在这个脚本中，栈顶的两个元素是相同的，所以验证通过，脚本将继续执行：

```
|                   |
|                   |
|                   |
|                   |
├───────────────────┤
|03dd8763...14cf740f|
├───────────────────┤
|30450221...68fa9b01|
└───────────────────┘
```

最后，执行`OP_CHECKSIG`指令，它使用栈顶的两份数据，第一份数据被看作公钥，第二份数据被看作签名，这条指令就是用公钥来验证签名是否有效。根据验证结果，成功存入`1`，失败存入`0`：



最后，当整个脚本执行结束后，检查栈顶元素是否为`0`，如果不为`0`，那么整个脚本就执行成功，这笔交易就被验证为有效的。

上述代码执行过程非常简单，因为比特币的脚本不含条件判断、循环等复杂结构。上述脚本就是对输入的两个数据视作签名和公钥，然后先验证公钥哈希是否与地址相同，再根据公钥验证签名，这种标准脚本称之为P2PKH（Pay to Public Key Hash）脚本。

输出

当小明给小红支付一笔比特币时，实际上小明创建了一个锁定脚本，该锁定脚本中引入了小红的地址。要想通过解锁脚本花费该输出，只有持有对应私钥的小红才能创建正确的解锁脚本（因为解锁脚本包含的签名只有小红的私钥才能创建），因此，小红事实上拥有了花费该输出的权利。

使用钱包软件创建的交易都是标准的支付脚本，但是，比特币的交易本质是成功执行解锁脚本和锁定脚本，所以，可以编写各种符合条件的脚本。比如，有人创建了一个交易，它的锁定脚本像这样：

```
OP_HASH256
      DATA 6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
  OP_EQUAL
```

这有点像一个数学谜题。它的意思是说，谁能够提供一个数据，它的hash256等于`6fe28c0a...`，谁就可以花费这笔输出。所以，解锁脚本实际上只需要提供一个正确的数据，就可以花费这笔输出。点这里查看谁花费了该输出。

比特币的脚本通过不同的指令还可以实现更灵活的功能。例如，多重签名可以让一笔交易只有在多数人同意的情况下才能够进行。最常见的多重签名脚本可以提供3个签名，只要任意两个签名被验证成功，这笔交易就可以成功。

```
FROM: UTXO Hash#index
AMOUNT: 10.5 btc
TO: P2SH: OP_2 pk1 pk2 pk3 OP_3 OP_CHECKMULTISIG
```

也就是说，3个人中，只要任意两个人同意用他们的私钥提供签名，就可以完成交易。这种方式也可以一定程度上防止丢失私钥的风险。3个人中如果只有一个人丢失了私钥，仍然可以保证这笔输出是可以被花费的。

支付的本质

从比特币支付的脚本可以看出，比特币支付的本质是由程序触发的数字资产转移。这种支付方式无需信任中介的参与，可以在零信任的基础上完成数字资产的交易，这也是为什么数字货币又被称为可编程的货币。

由此催生出了智能合约：当一个预先编好的条件被触发时，智能合约可以自动执行相应的程序，自动完成数字资产的转移。保险、贷款等金融活动在将来都可以以智能合约的形式执行。智能合约以程序来替代传统的纸质文件条款，并由计算机强制执行，将具有更高的更低的信任成本和运营成本。

小结

比特币采用脚本的方式进行可编程支付：通过执行解锁脚本确认某个UTXO的资产可以被私钥持有人转移给其他人。

多重签名

由比特币的签名机制可知，如果丢失了私钥，没有任何办法可以花费对应地址的资金。

这样就使得因为丢失私钥导致资金丢失的风险会很高。为了避免一个私钥的丢失导致地址的资金丢失，比特币引入了多重签名机制，可以实现分散风险的功能。

具体来说，就是假设N个人分别持有N个私钥，只要其中M个人同意签名就可以动用某个“联合地址”的资金。

多重签名地址实际上是一个Script Hash，以2-3类型的多重签名为例，它的创建过程如下：

```
const bitcoin = require('bitcoinjs-lib');
```

```
let
    pubKey1 =
'026477115981fe981a6918a6297d9803c4dc04f328f22041bedff886bbc2962e01',
    pubKey2 =
'02c96db2302d19b43d4c69368babace7854cc84eb9e061cde51cfa77ca4a22b8b9',
    pubKey3 =
'03c6103b3b83e4a24a0e33a4df246ef11772f9992663db0c35759a5e2ebf68d8e9',
    pubKeys = [pubKey1, pubKey2, pubKey3].map(s => Buffer.from(s, 'hex')); // 注意把string转换为Buffer

// 创建2-3 RedeemScript:
let redeemScript = bitcoin.script.multisig.output.encode(2, pubKeys);
console.log('Redeem script: ' + redeemScript.toString('hex'));

// 编码:
let scriptPubKey =
bitcoin.script.scriptHash.output.encode(bitcoin.crypto.hash160(redeemScript));
let address = bitcoin.address.fromOutputScript(scriptPubKey);

console.log('Multisig address: ' + address); //
36NUkt6FWUi3LAWBqWRdDmdTWbt91Yvfu7
```

```
Redeem script:
522102647711598 1fe981a6918a6297d9803c4dc04f328f22041bedff886bbc2962e012102c96db2
302d19b43d4c69368babace7854cc84eb9e061cde51cfa77ca4a22b8b92103c6103b3b83e4a24a0e
33a4df246ef11772f9992663db0c35759a5e2ebf68d8e953ae
Multisig address: 36NUkt6FWUi3LAWBqWRdDmdTWbt91Yvfu7
```

首先，我们需要所有公钥列表，这里是3个公钥。然后，通过`bitcoin.script.multisig.output.encode()`方法编码为2-3类型的脚本，对这个脚本计算hash160后，使用Base58编码即得到总是以`3`开头的多重签名地址，这个地址实际上是一个脚本哈希后的编码。

> 以3开头的地址就是比特币的多重签名地址，但从地址本身无法得知签名所需的M/N。

如果我们观察Redeem Script的输出，它的十六进制实际上是：

```
52
21 026477115981fe981a6918a6297d9803c4dc04f328f22041bedff886bbc2962e01
21 02c96db2302d19b43d4c69368babace7854cc84eb9e061cde51cfa77ca4a22b8b9
21 03c6103b3b83e4a24a0e33a4df246ef11772f9992663db0c35759a5e2ebf68d8e9
53
ae
```

翻译成比特币的脚本指令就是：

```
OP_2
PUSHDATA(33) 026477115981fe981a6918a6297d9803c4dc04f328f22041bedff886bbc2962e01
PUSHDATA(33) 02c96db2302d19b43d4c69368babace7854cc84eb9e061cde51cfa77ca4a22b8b9
PUSHDATA(33) 03c6103b3b83e4a24a0e33a4df246ef11772f9992663db0c35759a5e2ebf68d8e9
OP_3
OP_CHECKMULTISIG
```

`OP_2` 和 `OP_3` 构成2-3多重签名，这两个指令中间的3个 `PUSHDATA(33)` 就是我们指定的3个公钥，最后一个 `OP_CHECKMULTISIG` 表示需要验证多重签名。

发送给多重签名地址的交易创建的是P2SH脚本，而花费多重签名地址的资金需要的脚本就是M个签名+Redeem Script。

注意：从多重签名的地址本身并无法得知该多重签名使用的公钥，以及M-N的具体数值。必须将Redeem Script公示给每个私钥持有人，才能够验证多重签名地址是否正确（即包含了所有人的公钥，以及正确的M-N数值）。要花费多重签名地址的资金，除了M个私钥签名外，必须要有Redeem Script（可由所有人的公钥构造）。只有签名，没有Redeem Script是不能构造出解锁脚本来花费资金的。因此，保存多重签名地址的钱包必须同时保存Redeem Script。

利用多重签名，可以实现：

- 1-2，两人只要有一人同意即可使用资金；
- 2-2，两人必须都同意才可使用资金；
- 2-3，3人必须至少两人同意才可使用资金；
- 4-7，7人中多数人同意才可使用资金。

最常见的多重签名是2-3类型。例如，一个提供在线钱包的服务，为了防止服务商盗取用户的资金，可以使用2-3类型的多重签名地址，服务商持有1个私钥，用户持有两个私钥，一个作为常规使用，一个作为应急使用。这样，正常情况下，用户只需使用常规私钥即可配合服务商完成正常交易，服务商因为只持有1个私钥，因此无法盗取用户资金。如果服务商倒闭或者被黑客攻击，用户可使用自己掌握的两个私钥转移资金。

大型机构的比特币通常都使用多重签名地址以保证安全。例如，某个交易所的3-6多重签名地址 3D2oetdNuZUqQHPJmcMDDHYoqkyNVsFk9r。

利用多重签名，可以使得私钥丢失的风险被分散到N个人手中，并且，避免了少数人窃取资金的问题。

比特币的多重签名最多允许15个私钥参与签名，即可实现1-2至15-15的任意组合（$1 \leqslant M \leqslant N \leqslant 15$）。

小结

多重签名可以实现N个人持有私钥，其中M个人同意即可花费资金的功能。

多重签名降低了单个私钥丢失的风险。

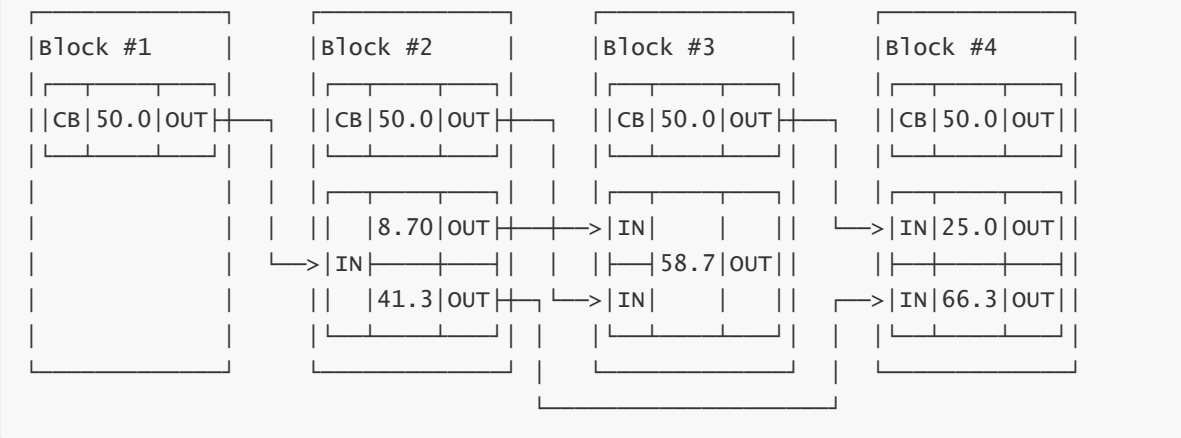
支付比特币到一个多重签名地址实际上是创建一个P2SH输出。

UTXO模型

比特币的区块链由一个个区块串联构成，而每个区块又包含一个或多个交易。

如果我们观察任何一个交易，它总是由若干个输入（Input）和若干个输出（Output）构成，一个Input指向的是前面区块的某个Output，只有Coinbase交易（矿工奖励的铸币交易）没有输入，只有凭空输出。所以，任何交易，总是可以由Input溯源到Coinbase交易。

这些交易的Input和Output总是可以串联起来：



还没有被下一个交易花费的Output被称为UTXO：Unspent TX Output，即未花费交易输出。给定任何一个区块，计算当前所有的UXTO金额之和，等同于自创世区块到给定区块的挖矿奖励之和。

因此，比特币的交易模型和我们平时使用的银行账号有所不同，它并没有账户这个说法，只有UTXO。想要确定某个人拥有的比特币，并无法通过某个账户查到，必须知道此人控制的所有UTXO金额之和。

在钱包程序中，钱包管理的是一组私钥，对应的是一组公钥和地址。钱包程序必须从创世区块开始扫描每一笔交易，如果：

1. 遇到某笔交易的某个Output是钱包管理的地址之一，则钱包余额增加；
2. 遇到某笔交易的某个Input是钱包管理的地址之一，则钱包余额减少。

钱包的当前余额总是钱包地址关联的所有UTXO金额之和。

如果刚装了一个新钱包，导入了一组私钥，在钱包扫描完整个比特币区块之前，是无法得知当前管理的地址余额的。

那么，给定一个地址，要查询该地址的余额，难道要从头扫描几百GB的区块链数据？

当然不是。

要做到瞬时查询，我们知道，使用关系数据库的主键进行查询，由于用了索引，速度极快。

因此，对区块链进行查询之前，首先要扫描整个区块链，重建一个类似关系数据库的地址-余额映射表。这个表的结构如下：

address	balance	lastUpdatedAtBlock
address-1	50.0	0

一开始，这是一个空表。每当扫描一个区块的所有交易后，某些地址的余额增加，另一些地址的余额减少，两者之差恰好为区块奖励：

address	balance	lastUpdatedAtBlock
address-1	50.0	0
address-2	40.0	3
address-3	50.0	3
address-4	10.0	3

这样，扫描完所有区块后，我们就得到了整个区块链所有地址的完整余额记录，查询的时候，并不是从区块链查询，而是从本地数据库查询。大多数钱包程序使用LevelDB来存储这些信息，手机钱包程序则是请求服务器，由服务器查询数据库后返回结果。

如果我们把MySQL这样的数据库看作可修改的，那么区块链就是不可修改，只能追加的只读数据库。但是，MySQL这样的数据库虽然其状态是可修改的，但它的状态改变却是由修改语句（INSERT/UPDATE/DELETE）引起的。把MySQL的binlog日志完整地记录下来，再进行重放，即可在另一台机器上完整地重建整个数据库。把区块链看作不可修改的binlog日志，我们只要把每个区块的所有交易重放一遍，即可重建一个地址-余额的数据库。

可见，比特币的区块链记录的是修改日志，而不是当前状态。

小结

比特币区块链使用UTXO模型，它没有账户这个概念；

重建整个地址-余额数据库需要扫描整个区块链，并按每个交易依次更新记录，即可得到当前状态。

Segwit地址

Segwit地址又称隔离见证地址。在比特币区块链上，经常可以看到类似`bc1qmy63mjadtw8nhzl69ukdepwzsyvv4yex5qlmkd`这样的以`bc`开头的地址，这种地址就是隔离见证地址。

Segwit地址有好几种，一种是以`3`开头的隔离见证兼容地址（Nested Segwit Address），从该地址上无法区分到底是多签地址还是隔离见证兼容地址，好处是钱包程序不用修改，可直接付款到该地址。

另一种是原生隔离见证地址（Native Segwit Address），即以`bc`开头的地址，它本质上就是一种新的编码方式。

我们回顾一下`1`开头的比特币地址是如何创建的：

1. 根据公钥计算hash160；
2. 添加固定头并计算带校验的Base58编码。

简单地概括就是使用Base58编码的公钥哈希。

而`bc`地址使用的不是Base58编码，而是Bech32编码，它的算法是：

1. 根据公钥计算hash160；
2. 使用Base32编码得到更长的编码；
3. 以`bc`作为识别码进行编码并带校验。

Bech32编码实际上由两部分组成：一部分是`bc`这样的前缀，被称为HRP（Human Readable Part，用户可读部分），另一部分是特殊的Base32编码，使用字母表`qpzry9x8gf2tvdw0s3jn54khce6mua7l`，中间用`1`连接。对一个公钥进行Bech32编码的代码如下：

```
const
    bitcoin = require('bitcoinjs-lib'),
    bech32 = require('bech32'),
    createHash = require('create-hash');
```

```
// 压缩的公钥:
let publicKey =
'02d0de0aaeaefad02b8bdc8a01a1b8b11c696bd3d66a2c5f10780d95b7df42645c';

// 计算hash160:
let
    sha256 = createHash('sha256'),
    ripemd160 = createHash('ripemd160'),
    hash256 = sha256.update(Buffer.from(publicKey, 'hex')).digest(),
    hash160 = ripemd160.update(hash256).digest();

// 计算bech32编码:
let words = bech32.toWords(hash160);
// 头部添加版本号0x00:
words.unshift(0);

// 对地址编码:
let address = bech32.encode('bc', words);
console.log(address); // bc1qmy63mjadtw8nhzl69ukdepwzsyvv4yex5qlmkd
```

```
bc1qmy63mjadtw8nhzl69ukdepwzsyvv4yex5qlmkd
```

和Base58地址相比，Bech32地址的优点有：

1. 不用区分大小写，因为编码用的字符表没有大写字母；
2. 有个固定前缀，可任意设置，便于识别；
3. 生成的二维码更小。

它的缺点是：

1. 和现有地址不兼容，钱包程序必须升级；
2. 使用`1`作为分隔符，却使用了字母`l`，容易混淆；
3. 地址更长，有42个字符。

那为什么要引入Segwit地址呢？按照官方说法，它的目的是为了解决比特币交易的延展性（Transaction Malleability）攻击。

延展性攻击

什么是延展性攻击呢？我们先回顾一下比特币的区块链如何保证一个交易有效并且不被修改：

1. 每个交易都必须签名才能花费输入（UTXO）；
2. 所有交易的哈希以Merkle Tree计算并存储到区块头。

我们再看每个交易的细节，假设有一个输入和一个输出，它类似：

```
tx = ... input#index ... signature ... output-script ...
```

而整个交易的哈希可直接根据交易本身计算：

```
tx-hash = dhash(tx)
```

因为只有私钥持有人才能正确地签名，所以，只要签名是有效的，tx本身就应该固定下来。

但问题出在ECDSA签名算法上。ECDSA签名算法基于私钥计算的签名实际上是两个整数，记作`(r, s)`，但由于椭圆曲线的对称性，`(r, -s mod N)`实际上也是一个有效的签名（N是椭圆曲线的固定参数之一）。换句话说，对某个交易进行签名，总是可以计算出两个有效的签名，并且这两个有效的签名还可以互相计算出来。

黑客可以在某一笔交易发出但并未落块的时间内，对签名进行修改，使之仍是一个有效的交易。注意黑客并无法修改任何输入输出的地址和金额，仅能修改签名。但由于签名的修改，使得整个交易的哈希被改变了。如果修改后的交易先被打包，虽然原始交易会被丢弃，且并不影响交易安全，但这个延展性攻击可用于攻击交易所。

要解决延展性攻击的问题，有两个办法，一是对交易签名进行归一化（Normalize）。因为ECDSA签名后总有两个有效的签名`(r, s)`和`(r, -s mod N)`，那只接受数值较小的那个签名，为此比特币引入了一个`SCRIPT_VERIFY_LOW_S`标志仅接受较小值的签名。

另一个办法是把签名数据移到交易之外，这样交易本身的哈希就不会变化。不含签名的交易计算出的哈希称为wtxid，为此引入了一种新的隔离见证地址。

小结

以`bc`开头的隔离见证地址使用了Bech32编码；

比特币延展性攻击的原因是ECDSA签名总是有两个有效签名，且可以相互计算；

规范ECDSA签名格式可强制使用固定签名（例如总是使用较小值的签名）。

HD钱包

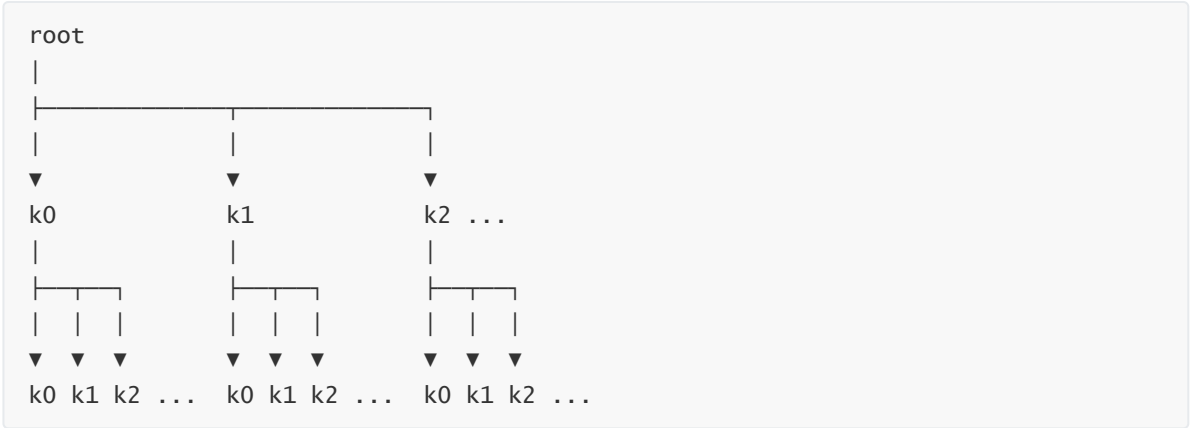
在比特币的链上，实际上并没有账户的概念，某个用户持有的比特币，实际上是其控制的一组UTXO，而这些UTXO可能是相同的地址（对应相同的私钥），也可能是不同的地址（对应不同的私钥）。

出于保护隐私的目的，同一用户如果控制的UTXO其地址都是不同的，那么很难从地址获知某个用户的比特币持币总额。但是，管理一组成千上万的地址，意味着管理成千上万的私钥，管理起来非常麻烦。

能不能只用一个私钥管理成千上万个地址？实际上是可以的。虽然椭圆曲线算法决定了一个私钥只能对应一个公钥，但是，可以通过某种确定性算法，先确定一个私钥k1，然后计算出k2、k3、k4……等其他私钥，就相当于只需要管理一个私钥，剩下的私钥可以按需计算出来。

这种根据某种确定性算法，只需要管理一个根私钥，即可实时计算所有"子私钥"的管理方式，称为HD钱包。

HD是Hierarchical Deterministic的缩写，意思是分层确定性。先确定根私钥root，然后根据索引计算每一层的子私钥：



对于任意一个私钥k，总是可以根据索引计算它的下一层私钥kn：

k_n=hdkey(k,\;n)$k_n = hdkey(k, n)$

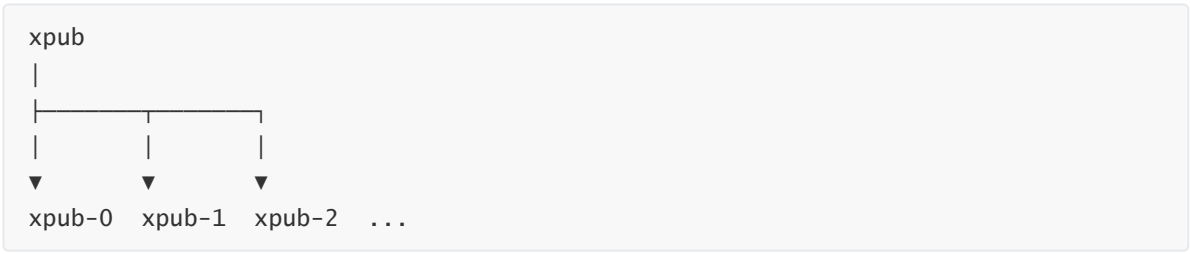
即HD层级实际上是无限的，每一层索引从0～232，约43亿个子key。这种计算被称为衍生（Derivation）。

现在问题来了：如何根据某个私钥计算下一层的子私钥？即函数`hdkey(k, n)`如何实现？

HD钱包采用的计算子私钥的算法并不是一个简单的SHA-256，私钥也不是普通的256位ECDSA私钥，而是一个扩展的512位私钥，记作xprv，它通过SHA-512算法配合ECC计算出子扩展私钥，仍然是512位。通过扩展私钥可计算出用于签名的私钥以及公钥。

简单来说，只要给定一个根扩展私钥（随机512位整数），即可计算其任意索引的子扩展私钥。扩展私钥总是能计算出扩展公钥，记作xpub：

从xprv及其对应的xpub可计算出真正用于签名的私钥和公钥。之所以要设计这种算法，是因为扩展公钥xpub也有一个特点，那就是可以直接计算其子层级的扩展公钥：



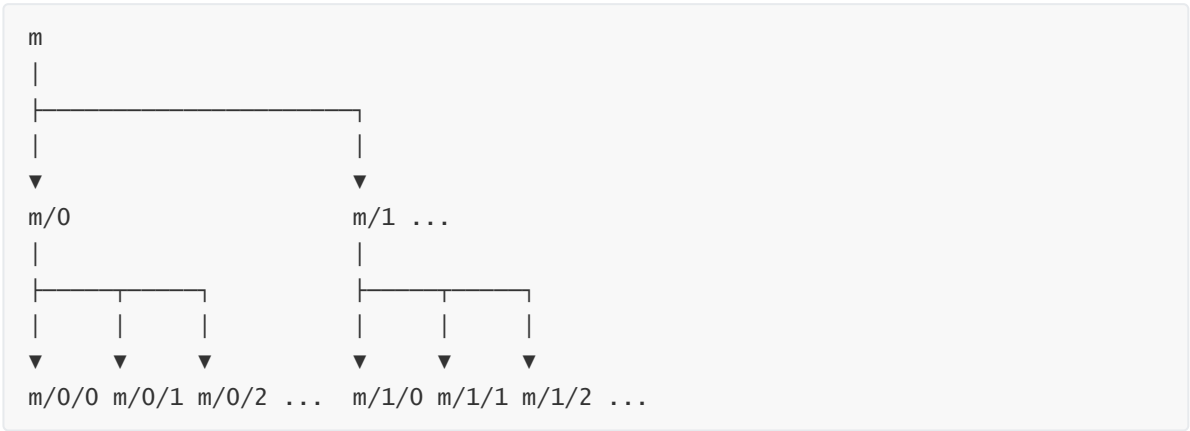
因为xpub只包含公钥，不包含私钥，因此，可以安全地把xpub交给第三方（例如，一个观察钱包），它可以根据xpub计算子层级的所有地址，然后在比特币的链上监控这些地址的余额，但因为没有私钥，所以只能看，不能花。

因此，HD钱包通过分层确定性算法，实现了以下功能：

- 只要确定了扩展私钥xprv，即可根据索引计算下一层的任何扩展私钥；
- 只要确定了扩展公钥xpub，即可根据索引计算下一层的任何扩展公钥；
- 用户只需保存顶层的一个扩展私钥，即可计算出任意一层的任意索引的扩展私钥。

从理论上说，扩展私钥的层数是没有限制的，每一层的数量被限制在0～232，原因是扩展私钥中只有4字节作为索引，因此索引范围是0～232。

通常把根扩展私钥记作`m`，子扩展私钥按层级记作`m/x/y/z`等：



例如，`m/0/2`表示从`m`扩展到`m/0`（索引为0）再扩展到`m/0/2`（索引为2）。

安全性

HD钱包给私钥管理带来了非常大的方便，因为只需要管理一个根扩展私钥，就可以管理所有层级的所有衍生私钥。

但是HD钱包的扩展私钥算法有个潜在的安全性问题，就是如果某个层级的xprv泄露了，可反向推导出上层的xprv，继而推导出整个HD扩展私钥体系。为了避免某个子扩展私钥的泄漏导致上层扩展私钥被反向推导，HD钱包还有一种硬化的衍生计算方式（Hardened Derivation），它通过算法“切断”了母扩展私钥和子扩展私钥的反向推导。HD规范把索引0～231作为普通衍生索引，而索引231～232作为硬化衍生索引，硬化衍生索引通常记作0'、1'、2'……，即索引0'=231，1'=231+1，2'=231+2，以此类推。

因此，`m/44'/0`表示的子扩展私钥，它的第一层衍生索引`44'`是硬化衍生，实际索引是231+44=2147483692。从`m/44'/0`无法反向推导出`m/44'`。

在只有扩展公钥的情况下，只能计算出普通衍生的子公钥，无法计算出硬化衍生的子扩展公钥，即可计算出的子扩展公钥索引被限制在0～231。因此，观察钱包能使用的索引是0～231。

BIP-32

比特币的BIP-32规范详细定义了HD算法原理和各种推导规则，可阅读此文档以便实现HD钱包。

小结

HD钱包采用分层确定性算法通过根扩展私钥计算所有层级的所有子扩展私钥，继而得到扩展公钥和地址；

可以通过普通衍生和硬化衍生两种方式计算扩展子私钥，后者更安全，但对应的扩展公钥无法计算硬化衍生的子扩展公钥；

通过扩展公钥可以在没有扩展私钥的前提下计算所有普通子扩展公钥，此特性可实现观察钱包。

钱包层级

HD钱包算法决定了只要给定根扩展私钥，整棵树的任意节点的扩展私钥都可以计算出来。

我们来看看如何利用bitcoinjs-lib这个JavaScript库来计算HD地址：

```
const bitcoin = require('bitcoinjs-lib');

let
    xprv =
'xprv9s21ZrQH143K4EKMS3q1vbJo564QAbs98BfXQME6nk8UCrnXnv8vWg9qmtup3kTug96p5E3AvarBhPMScQDqMhEEm41rpYEdXBL8qzVZtwz',
    root = bitcoin.HDNode.fromBase58(xprv);

// m/0:
var m_0 = root.derive(0);
console.log("xprv m/0: " + m_0.toBase58());
console.log("xpub m/0: " + m_0.neutered().toBase58());
console.log(" prv m/0: " + m_0.keyPair.toWIF());
console.log(" pub m/0: " + m_0.keyPair.getAddress());

// m/1:
var m_1 = root.derive(0);
console.log("xprv m/1: " + m_1.toBase58());
console.log("xpub m/1: " + m_1.neutered().toBase58());
console.log(" prv m/1: " + m_1.keyPair.toWIF());
console.log(" pub m/1: " + m_1.keyPair.getAddress());
```

```
xprv m/0:
xprv9vV8wfuBFqPwqk1QRQ6W8NWXV7fnTQmf56r5dYXz4YNDafrhGeg6N2dVF7MPhrPUomzdQSSZAj5X
XLCirjqKuEVSnbbqWd1BvkxkGPiNxY9
xpub m/0:
xpub69UVMBS56CxF4E5sXRdWVWTG39WGrsVWSKmgRvwbcsuCTUBqpBzLupwy6P98uDM99qVb6Y32drbS
Fz7iaj5TDYPuRgoHL4U1bQRFHygiMZC
 prv m/0: L3sZcLifBMDkutRFKy7HKNJ5hjJo8vKt8WrHW5mqenSf2e47SHaT
 pub m/0: 1EwGHVkkRNB98FZyhwpmxU9Ax9fVpnBCcZ
xprv m/1:
xprv9vV8wfuBFqPwqk1QRQ6W8NWXV7fnTQmf56r5dYXz4YNDafrhGeg6N2dVF7MPhrPUomzdQSSZAj5X
XLCirjqKuEVSnbbqWd1BvkxkGPiNxY9
xpub m/1:
xpub69UVMBS56CxF4E5sXRdWVWTG39WGrsVWSKmgRvwbcsuCTUBqpBzLupwy6P98uDM99qVb6Y32drbS
Fz7iaj5TDYPuRgoHL4U1bQRFHygiMZC
 prv m/1: L3sZcLifBMDkutRFKy7HKNJ5hjJo8vKt8WrHW5mqenSf2e47SHaT
 pub m/1: 1EwGHVkkRNB98FZyhwpmxU9Ax9fVpnBCcZ
```

注意到以`xprv`开头的`xprv9s21ZrQH...`是512位扩展私钥的Base58编码，解码后得到的就是原始扩展私钥。

可以从某个xpub在没有xprv的前提下直接推算子公钥：

```
const bitcoin = require('bitcoinjs-lib');

let
    xprv =
'xprv9s21ZrQH143K4EKMS3q1vbJo564QAbs98BfXQME6nk8UCrnXnv8vWg9qmtup3kTug96p5E3Avar
BhPMScQDqMhEEm41rpYEdXBL8qzVZtwz',
    root = bitcoin.HDNode.fromBase58(xprv);

// m/0:
let
    m_0 = root.derive(0),
    xprv_m_0 = m_0.toBase58(),
    xpub_m_0 = m_0.neutered().toBase58();

// 方法一：从m/0的扩展私钥推算m/0/99的公钥地址：
let pub_99a = bitcoin.HDNode.fromBase58(xprv_m_0).derive(99).getAddress();

// 方法二：从m/0的扩展公钥推算m/0/99的公钥地址：
let pub_99b = bitcoin.HDNode.fromBase58(xpub_m_0).derive(99).getAddress();

// 比较公钥地址是否相同：
console.log(pub_99a);
console.log(pub_99b);
```

```
1GvmxzNG5i8EqXQuD75XaGjqf2oCEmJ87M
1GvmxzNG5i8EqXQuD75XaGjqf2oCEmJ87M
```

但不能从xpub推算硬化的子公钥：

```
const bitcoin = require('bitcoinjs-lib');

let
```

```
    xprv =
'xprv9s21ZrQH143K4EKMS3q1vbJo564QAbs98BfXQME6nk8UCrnXnv8vWg9qmtup3kTug96p5E3Avar
BhPMScQDqMhEEm41rpYEdXBL8qzVZtwz',
    root = bitcoin.HDNode.fromBase58(xprv);

// m/0:
let
    m_0 = root.derive(0),
    xprv_m_0 = m_0.toBase58(),
    xpub_m_0 = m_0.neutered().toBase58();

// 从m/0的扩展私钥推算m/0/99'的公钥地址:
let pub_99a =
bitcoin.HDNode.fromBase58(xprv_m_0).deriveHardened(99).getAddress();
console.log(pub_99a);

// 不能从m/0的扩展公钥推算m/0/99'的公钥地址:
bitcoin.HDNode.fromBase58(xpub_m_0).deriveHardened(99).getAddress();
```

```
1PVoHWkWFk6uJYDERskP5HSwG7WHfYLacF
TypeError: Could not derive hardened child key
```

BIP-44

HD钱包理论上有无限的层级，对使用secp256k1算法的任何币都适用。但是，如果一种钱包使用`m/1/2/x`，另一种钱包使用`m/3/4/x`，没有一种统一的规范，就会乱套。

比特币的BIP-44规范定义了一种如何派生私钥的标准，它本身非常简单：

```
m / purpose' / coin_type' / account' / change / address_index
```

其中，`purpose`总是`44`，`coin_type`在SLIP-44中定义，例如，`0=BTC`，`2=LTC`，`60=ETH`等。`account`表示用户的某个“账户”，由用户自定义索引，`change=0`表示外部交易，`change=1`表示内部交易，`address_index`则是真正派生的索引为0～231的地址。

例如，某个比特币钱包给用户创建的一组HD地址实际上是：

- m/44'/0'/0'/0/0
- m/44'/0'/0'/0/1
- m/44'/0'/0'/0/2
- m/44'/0'/0'/0/3
- ...

如果是莱特币钱包，则用户的HD地址是：

- m/44'/2'/0'/0/0
- m/44'/2'/0'/0/1
- m/44'/2'/0'/0/2
- m/44'/2'/0'/0/3
- ...

小结

实现了BIP-44规范的钱包可以管理所有币种。相同的根扩展私钥在不同钱包上派生的一组地址都是相同的。

助记词

从HD钱包的创建方式可知，要创建一个HD钱包，我们必须首先有一个确定的512bit（64字节）的随机数种子。

如果用电脑生成一个64字节的随机数作为种子当然是可以的，但是恐怕谁也记不住。

如果自己想一个句子，例如`bitcoin is awesome`，然后计算SHA-512获得这个64字节的种子，虽然是可行的，但是其安全性取决于自己想的句子到底有多随机。像`bitcoin is awesome`本质上就是3个英文单词构成的随机数，长度太短，所以安全性非常差。

为了解决初始化种子的易用性问题，BIP-39规范提出了一种通过助记词来推算种子的算法：

以英文单词为例，首先，挑选2048个常用的英文单词，构造一个数组：

```
const words = ['abandon', 'ability', 'able', ..., 'zoo'];
```

然后，生成128~256位随机数，注意随机数的总位数必须是32的倍数。例如，生成的256位随机数以16进制表示为：

```
179e5af5ef66e5da5049cd3de0258c5339a722094e0fdbbbe0e96f148ae80924
```

在随机数末尾加上校验码，校验码取SHA-256的前若干位，并使得总位数凑成11的倍数，即。上述随机数校验码的二进制表示为`00010000`。

将随机数+校验码按每11 bit一组，得到范围是0~2047的24个整数，把这24个整数作为索引，就得到了最多24个助记词，例如：

```
bleak version runway tell hour unfold donkey defy digital abuse glide please omit
much cement sea sweet tenant demise taste emerge inject cause link
```

由于在生成助记词的过程中引入了校验码，所以，助记词如果弄错了，软件可以提示用户输入的助记词可能不对。

生成助记词的过程是计算机随机产生的，用户只要记住这些助记词，就可以根据助记词推算出HD钱包的种子。

注意：不要自己挑选助记词，原因一是随机性太差，二是缺少校验。

生成助记词可以使用bip39这个JavaScript库：

```
const bip39 = require('bip39');

let words = bip39.generateMnemonic(256);
console.log(words);

console.log('is valid mnemonic? ' + bip39.validateMnemonic(words));
```

```
leaf mansion night switch bread more dutch pigeon various humor cancel history
laundry divorce switch pretty canal tail deliver grape old child quit proud
is valid mnemonic? true
```

运行上述代码，每次都会得到随机生成的不同的助记词。

如果想用中文作助记词也是可以的，给`generateMnemonic()`传入一个中文助记词数组即可：

```
const bip39 = require('bip39');

// 第二个参数rng可以为null:
var words = bip39.generateMnemonic(256, null,
bip39.wordlists.chinese_simplified);
console.log(words);
```

```
亚 自 影 惩 兼 氯 粒 割 床 底 棒 润 铁 蒋 粒 到 畅 官 闷 抽 麻 腹 天 乡
```

注意：同样索引的中文和英文生成的HD种子是不同的。各种语言的助记词定义在[bip-0039-wordlists.md]。

根据助记词推算种子

根据助记词推算种子的算法是PBKDF2，使用的哈希函数是Hmac-SHA512，其中，输入是助记词的UTF-8编码，并设置Key为`mnemonic`+用户口令，循环2048次，得到最终的64字节种子。上述助记词加上口令`bitcoin`得到的HD种子是：

```
b59a8078d4ac5c05b0c92b775b96a466cd136664bfe14c1d49aff3ccc94d52dfb1d59ee628426192
eff5535d6058cb64317ef2992c8b124d0f72af81c9ebfaaa
```

该种子即为HD钱包的种子。

> 要特别注意：用户除了需要记住助记词外，还可以额外设置一个口令。HD种子的生成依赖于助记词和口令，丢失助记词或者丢失口令（如果设置了口令的话）都将导致HD钱包丢失！

用JavaScript代码实现为：

```
const bip39 = require('bip39');

let words = bip39.generateMnemonic(256);
console.log(words);

let seedBuffer = bip39.mnemonicToSeed(words);
let seedAsHex = seedBuffer.toString('hex');
// or use bip39.mnemonicToSeedHex(words)
console.log(seedAsHex);
```

```
tired blur mango myth bomb amount absurd travel box faint guess uniform kitten
crime spring orchard vital flee sustain fantasy squeeze flavor recall electric
293756ce6a417237c6907e0a41540d59d1bbdfeeb3c7522b6ffccca60df60df5b758670b30bc10f
3fada6952415efa62654674a95efe5343c80eb8827a8944c
```

根据助记词和口令生成HD种子的方法是在`mnemonicToSeed()`函数中传入password：

```
const bip39 = require('bip39');
let words = bip39.generateMnemonic(256);
console.log(words);

let password = 'bitcoin';

let seedAsHex = bip39.mnemonicToSeedHex(words, password);
console.log(seedAsHex);
```

```
shiver perfect during enlist van arm peasant funny pulp cherry soccer fix embody
moment until cushion clock vault increase way tumble chuckle twist survey
8e3cd4205660b5d772f5c3e79e7701f84bfdcbd1f8d38ac1eb04379ede7e28f1e912c82c137dc125
6c0a7bd61b5ffe137816a2e82cc01189b03caf55943aa302
```

从助记词算法可知，只要确定了助记词和口令，生成的HD种子就是确定的。

如果两个人的助记词相同，那么他们的HD种子也是相同的。这也意味着如果把助记词抄在纸上，一旦泄漏，HD种子就泄漏了。

如果在助记词的基础上设置了口令，那么只知道助记词，不知道口令，也是无法推算出HD种子的。

把助记词抄在纸上，口令记在脑子里，这样，泄漏了助记词也不会导致HD种子被泄漏，但要牢牢记住口令。

最后，我们使用助记词+口令的方式来生成一个HD钱包的HD种子并计算出根扩展私钥：

```
const
    bitcoin = require('bitcoinjs-lib'),
    bip39 = require('bip39');

let
    words = 'bleak version runway tell hour unfold donkey defy digital abuse
glide please omit much cement sea sweet tenant demise taste emerge inject cause
link',
    password = 'bitcoin';

// 计算seed:
let seedHex = bip39.mnemonicToSeedHex(words, password);
console.log('seed: ' + seedHex); // b59a8078...c9ebfaaa

// 生成root:
let root = bitcoin.HDNode.fromSeedHex(seedHex);
console.log('xprv: ' + root.toBase58()); // xprv9s21ZrQH...uLgyr9kF
console.log('xpub: ' + root.neutered().toBase58()); // xpub661MyMwA...oy32fcRG

// 生成派生key:
let child0 = root.derivePath("m/44'/0'/0'/0/0");
console.log("prv m/44'/0'/0'/0/0: " + child0.keyPair.toWIF()); //
KzuPk3PXKdnd6QwLqUCK38PrXoqJfJmACzxTaa6TFKzPJR7H7AFg
console.log("pub m/44'/0'/0'/0/0: " + child0.getAddress()); //
1PwKkrF366RdTuYsS8KWEbGxfP4bikegcS
```

```
seed:
b59a8078d4ac5c05b0c92b775b96a466cd136664bfe14c1d49aff3ccc94d52dfb1d59ee628426192
eff5535d6058cb64317ef2992c8b124d0f72af81c9ebfaaa
xprv:
xprv9s21ZrQH143K4ATirFwVJe2ZvmmBACdKcK1cWXtDdMhLFViGUDBDWJnWHLYGz5624M8zFsYKGoGt
8UVcPMwu9MR5gveP98ZoRq7uLgyr9kF
xpub:
xpub661MyMwAqRbcGeYBxHUVfmyJUobfZfMAyXwDJvHqBhEK8J3R1kVU476z8dvaQYCvwWPTQFzVZUuN
TAWWdhNQVY2TjTiFseh8j9goy32fcRG
prv m/44'/0'/0'/0/0: KzuPk3PXKdnd6QwLqUCK38PrXoqJfJmACzxTaa6TFKzPJR7H7AFg
pub m/44'/0'/0'/0/0: 1PwKkrF366RdTuYsS8KWEbGxfP4bikegcS
```

可以通过https://iancoleman.io/bip39/在线测试BIP-39并生成HD钱包。请注意，该网站仅供测试使用。生成正式使用的HD钱包*必须*在*可信任的离线环境*下操作。

小结

BIP-39规范通过使用助记词+口令来生成HD钱包的种子，用户只需记忆助记词和口令即可随时恢复HD钱包。

丢失助记词或者丢失口令均会导致HD钱包丢失。