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：

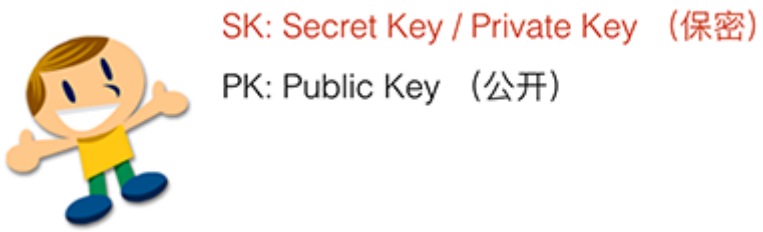
数字签名

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。

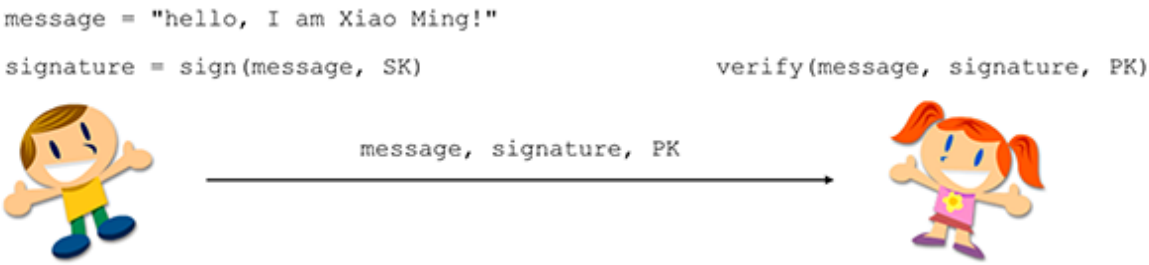
在比特币交易中，付款方就是通过数字签名来证明自己拥有某一笔比特币，并且，要把这笔比特币转移给指定的收款方。

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。

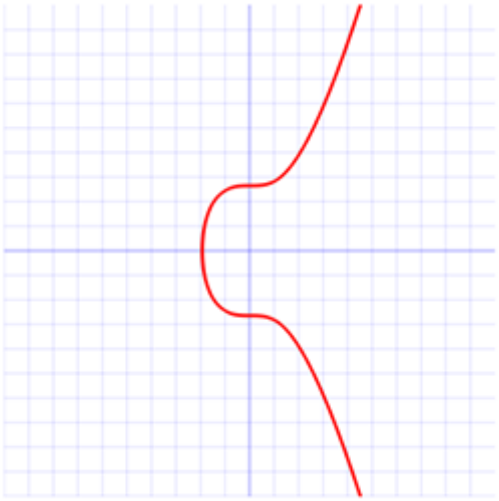
最后，签名不可抵赖。如果对签名进行验证通过了，那么，该消息肯定是由签名人自己发出的，他不能抵赖自己曾经发过这一条消息。

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。

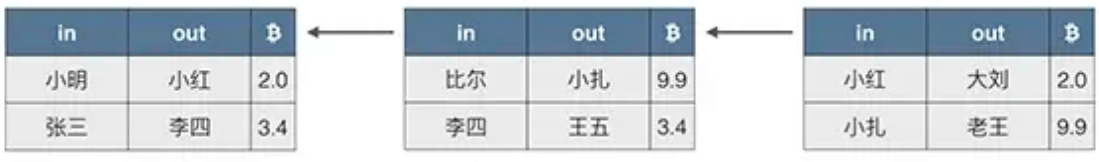
对大多数普通用户来说，想要记住私钥非常困难，所以*强烈不建议使用脑钱包*。

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。

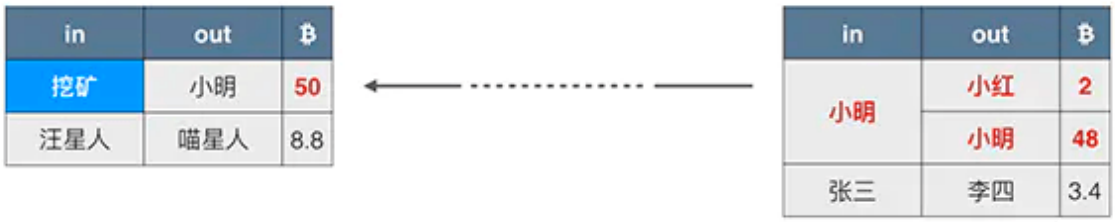
交易

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：



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：

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：

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：

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。

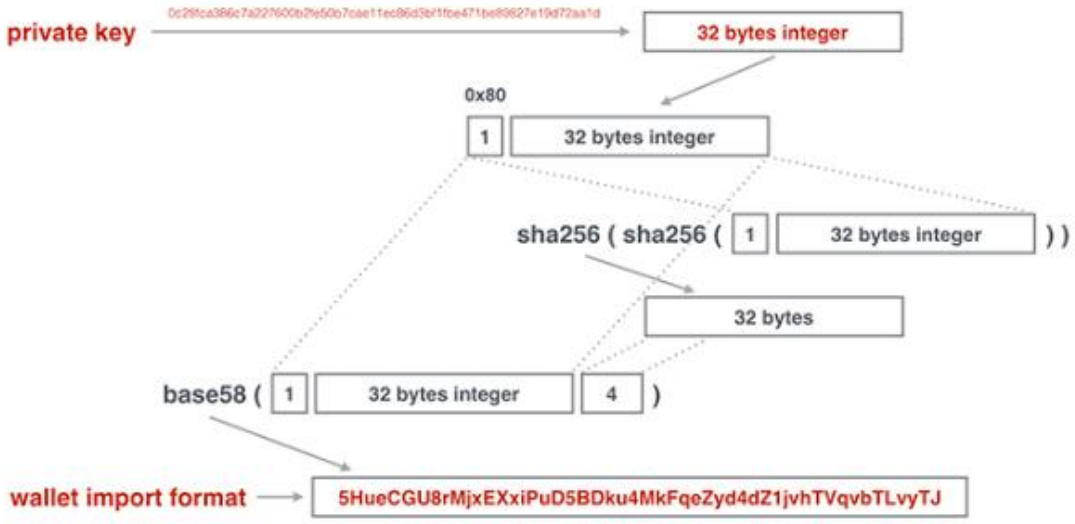
对私钥进行Base58编码有两种方式，一种是非压缩的私钥格式，一种是压缩的私钥格式，它们分别对应非压缩的公钥格式和压缩的公钥格式。

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：

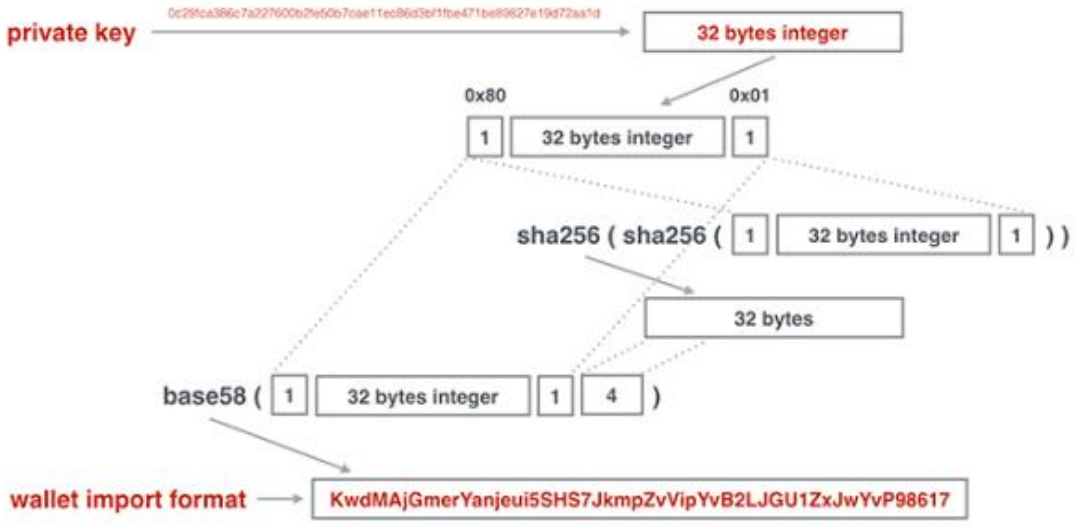
```
// 十六进制表示的私钥:
let privateKey =
'0c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1d';
// 对私钥编码:
let encoded = wif.encode(
        0x80, // 0x80前缀
        Buffer.from(privateKey, 'hex'), // 转换为字节
        false // 非压缩格式
);
console.log(encoded);
```

```
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。

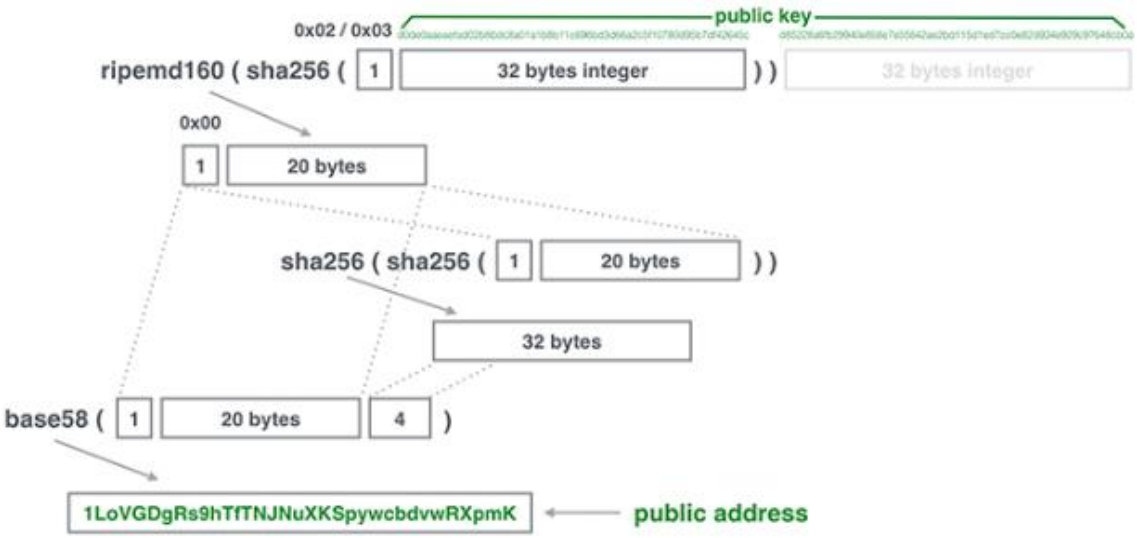
地址

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
```

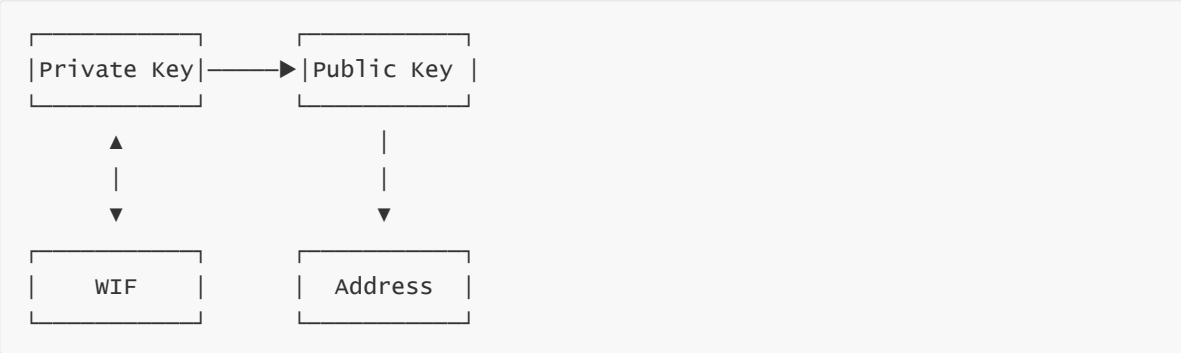
计算地址的时候，不必知道私钥，可以直接从公钥计算地址，即通过`ECPair.fromPublicKeyBuffer`构造一个不带私钥的`ECPair`即可计算出地址。

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 =
3044022052d0b6e817e01e22ba6ab19c0ab9cdbb2dbcd0612c5b8f990431dd0634f5a96530220188b
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 = '3044022052d0b6e817e01e22ba6ab19c0'
              + '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

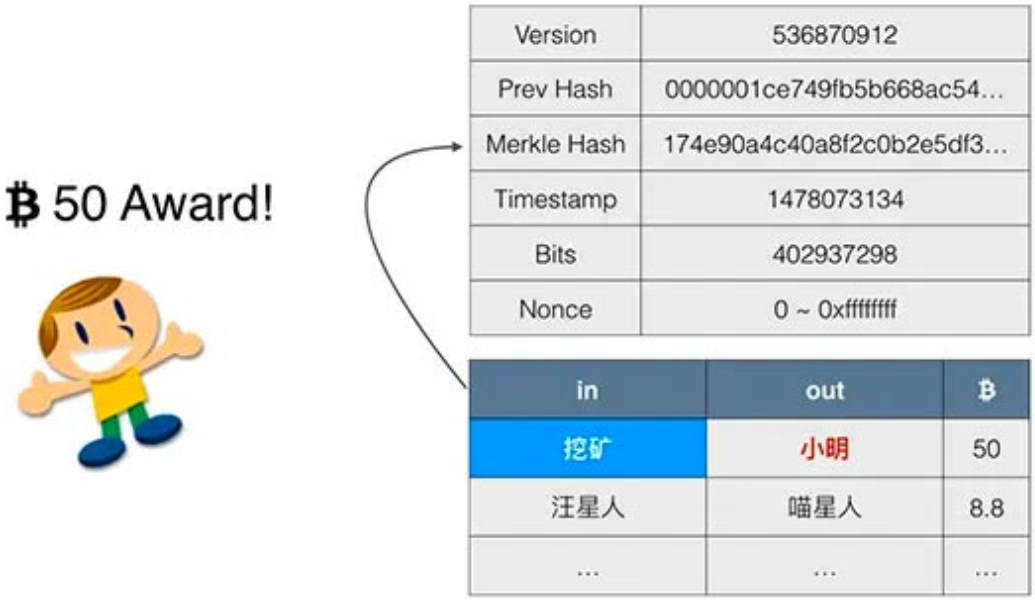
通过私钥可以对消息进行签名，签名可以保证消息防伪造，防篡改，防抵赖。

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。

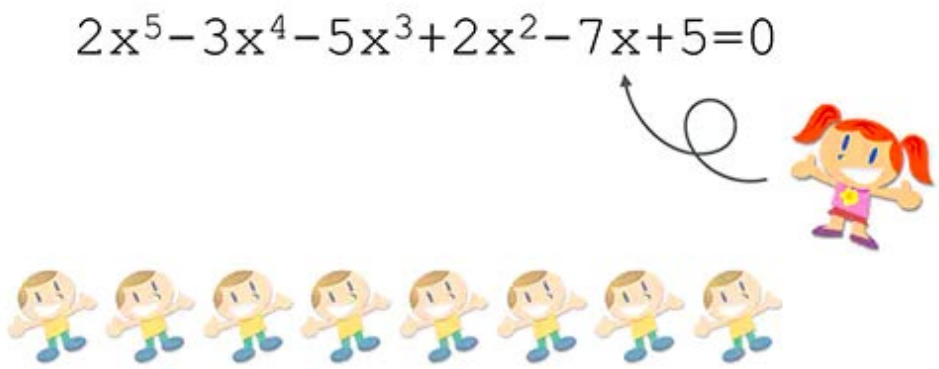
工作量证明

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：

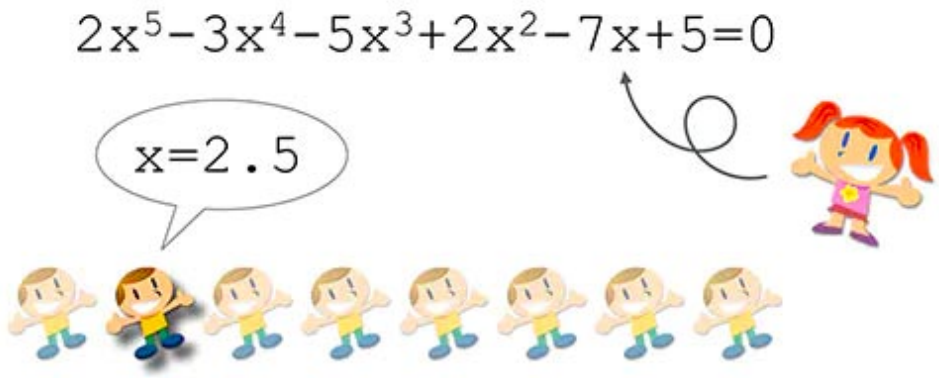
比特币的工作量证明需要归结为计算机计算，也就是数学问题。如何构造一个数学问题来实现工作量证明？我们来看一个简单的例子。

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：



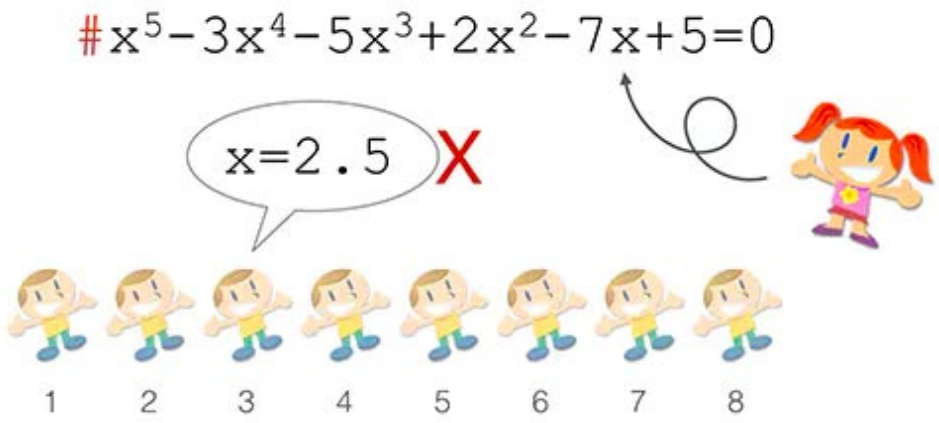
因为解高次方程没有固定的公式，需要进行大量的计算，才能算出正确的结果，这个计算过程就需要一定的工作量。假设小明率先计算出了结果x = 2.5，小红可以简单地验证这个结果是否正确：



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
```

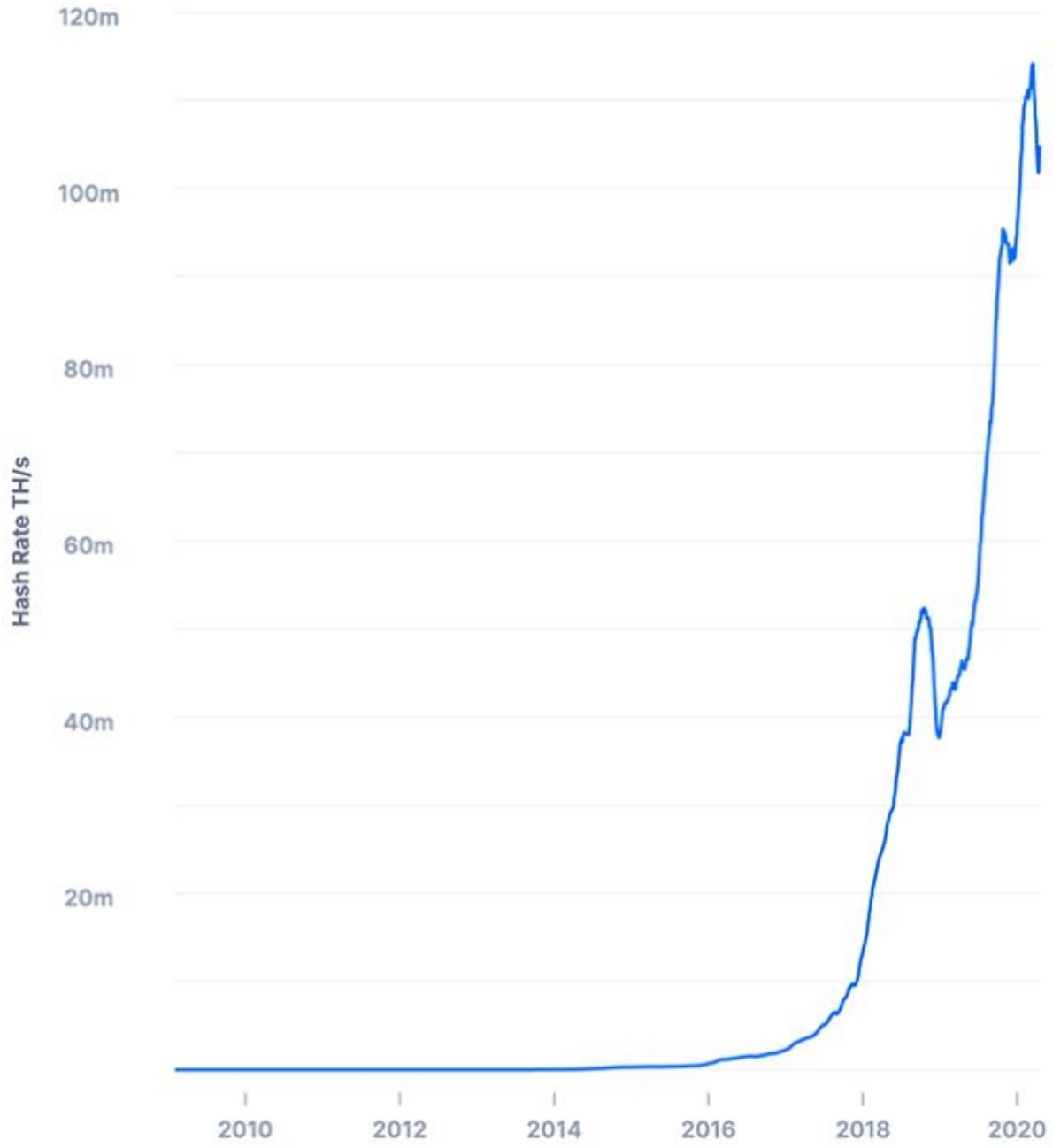
注意，难度值越小，说明哈希值前面的`0`越多，计算难度越大。

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。

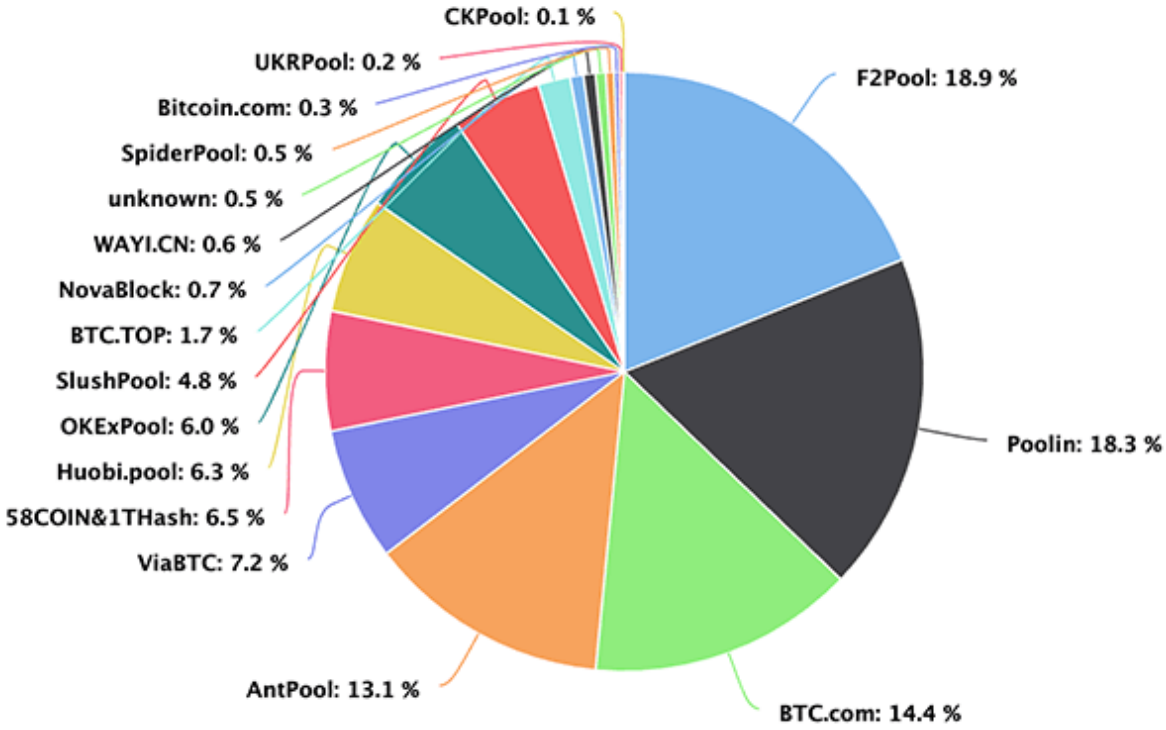
根据比特币每个区块的难度值和产出时间，就可以推算出整个比特币网络的全网算力。

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。

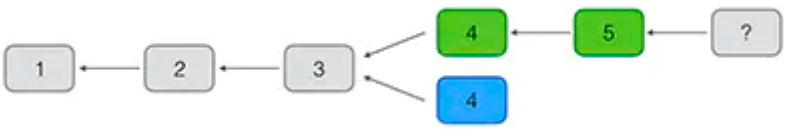
比特币总量被限制为约2100万个比特币，初始挖矿奖励为每个区块50个比特币，以后每4年减半。

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。

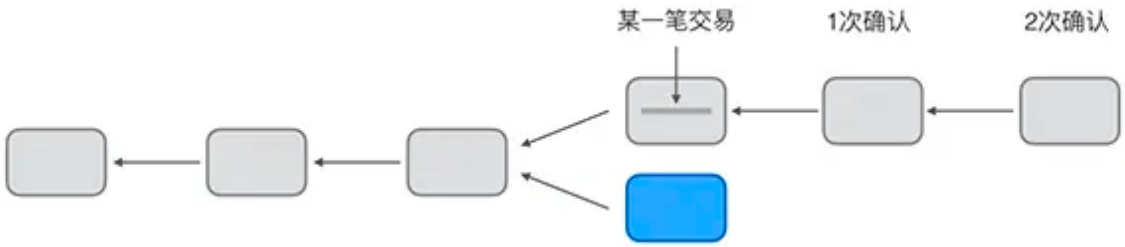
通常，矿工接收先收到的有效区块，由于P2P网络广播的顺序是不确定的，不同的矿工先收到的区块是有可能的不同的。这个时候，我们说区块发生了分叉：

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。