

# 区块链教程

零基础入门区块链，顺便学习写代码！

廖雪峰

2025-06-16

<https://liaoxuefeng.com/books/blockchain/>

# 目录

1. 简介
2. 比特币
  - 2.1. 区块链原理
  - 2.2. P2P交易原理
    - 2.2.1. 私钥
    - 2.2.2. 公钥和地址
    - 2.2.3. 签名
  - 2.3. 挖矿原理
  - 2.4. 可编程支付原理
    - 2.4.1. 多重签名
  - 2.5. UTXO模型
  - 2.6. Segwit地址
  - 2.7. HD钱包
    - 2.7.1. 钱包层级
    - 2.7.2. 助记词
    - 2.7.3. 地址监控
3. 以太坊
  - 3.1. 账户
  - 3.2. 区块结构
  - 3.3. 交易
  - 3.4. 智能合约
    - 3.4.1. 编写合约
    - 3.4.2. 部署合约
    - 3.4.3. 调用合约
    - 3.4.4. 编写Dapp
    - 3.4.5. 常用合约

#### 3.4.5.1. [ERC-20](#)

#### 3.4.5.2. [Wrapped Ether](#)

# 简介

[原文链接](#)



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！

最后，请大家务必注意：

本教程为技术教程，教程的所有内容均不构成任何投资比特币或其他数字货币的意见和建议，也不赞成个人炒作任何数字货币！

本教程为技术教程，教程的所有内容均不构成任何投资比特币或其他数字货币的意见和建议，也不赞成个人炒作任何数字货币！

本教程为技术教程，教程的所有内容均不构成任何投资比特币或其他数字货币的意见和建议，也不赞成个人炒作任何数字货币！

重要的话说三遍，一心炒币，对技术不感兴趣的童鞋请自觉关闭页面离开，不要继续浪费时间学习。

[评论](#)

# 比特币

## 原文链接

什么是比特币？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16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。

[评论](#)



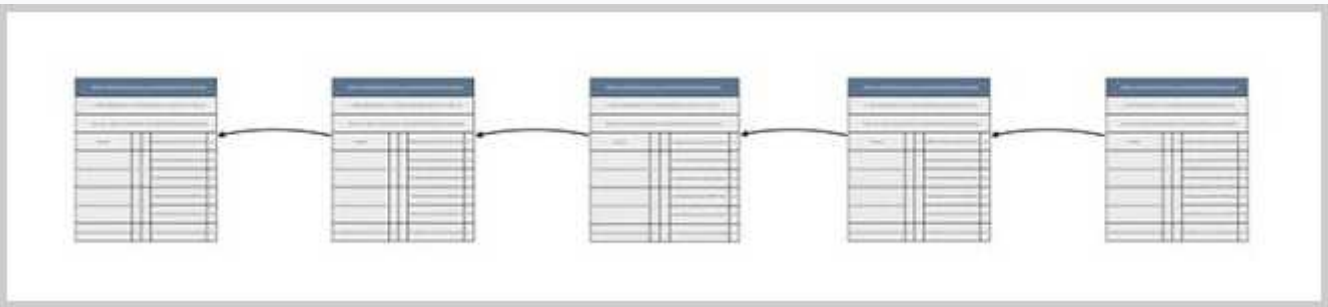
# 区块链原理

[原文链接](#)

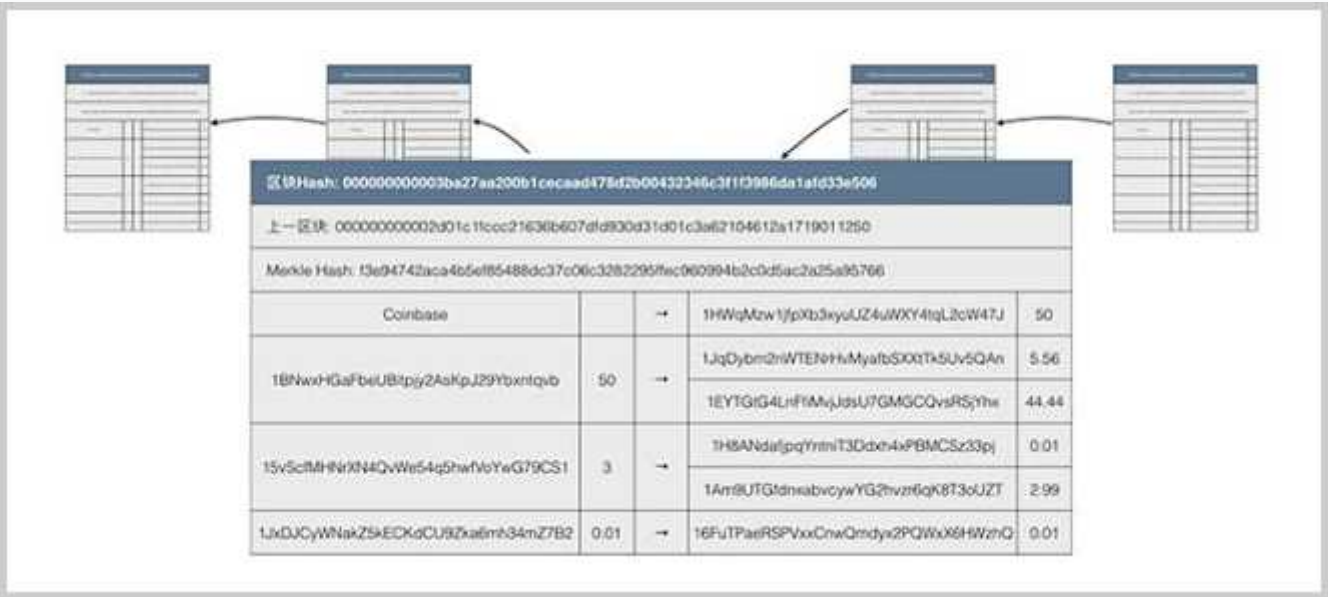
区块链就是一个不断增长的全网总账本，每个完全节点都拥有完整的区块链，并且，节点总是信任最长的区块链，伪造区块链需要拥有超过51%的全网算力。

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

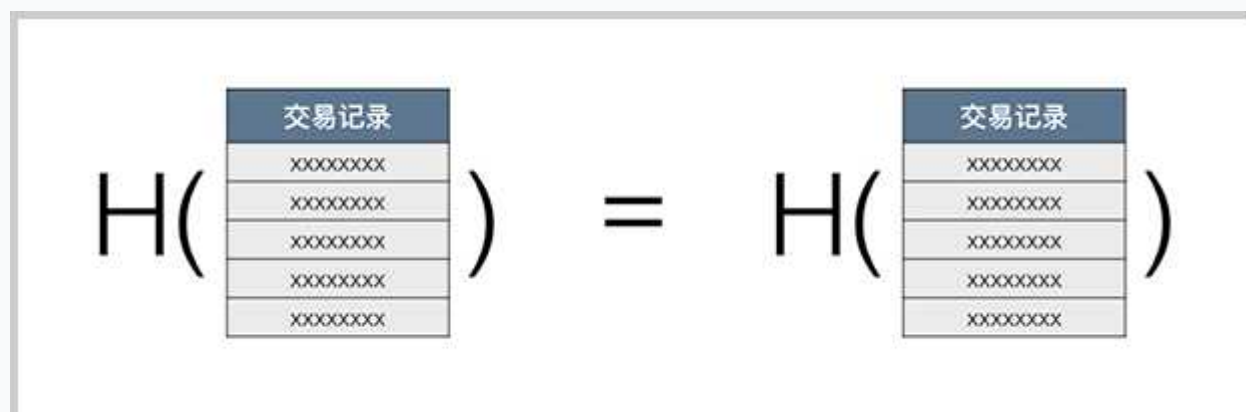
Select Platform:

Microsoft Windows

Other Downloads:

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

和文件类似，如果两份数据的哈希相同，则几乎可以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

哈希算法	输出长度(bit)	输出长度(字节)
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 $2^{130}$ 次方的随机输入，有99.8%的概率碰撞。注意 $2^{130}$ 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'));
```

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：

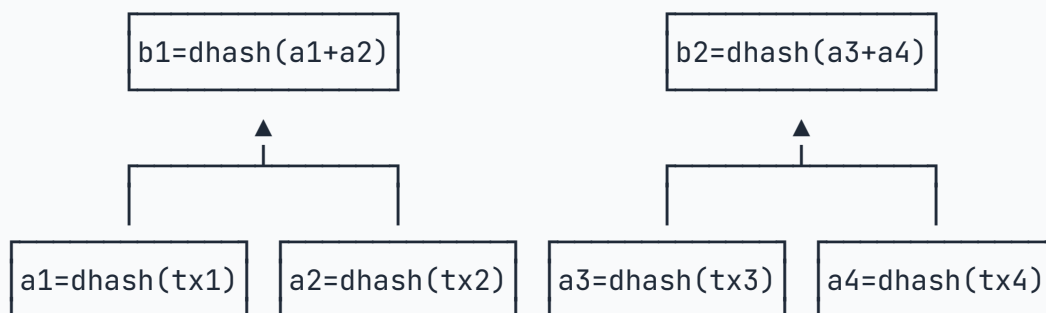


Merkle Hash是把一系列数据的哈希根据一个简单算法变成一个汇总的哈希。

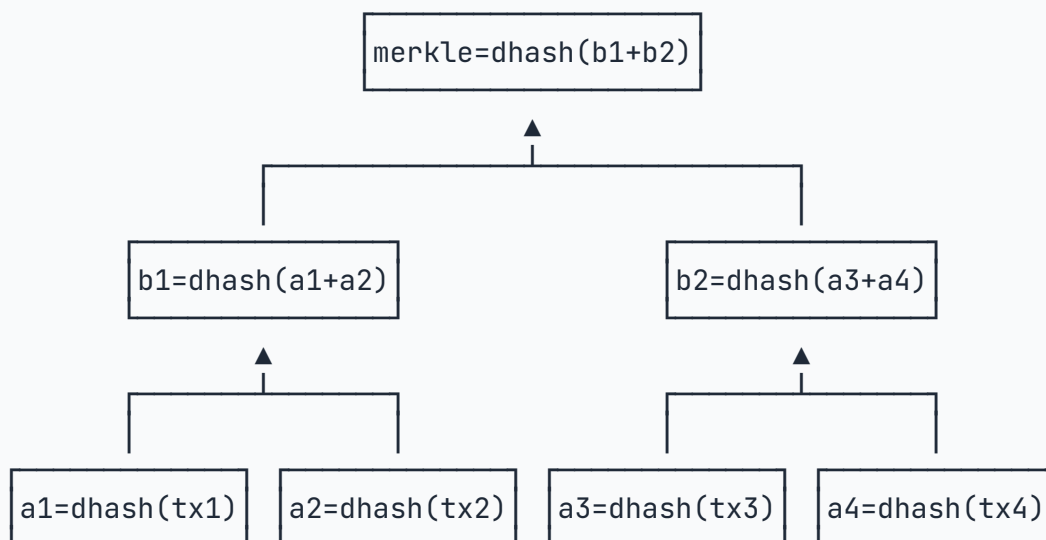
假设一个区块有4个交易，我们对每个交易数据做dhash，得到4个哈希值 `a1` , `a2` , `a3` 和 `a4` ：

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

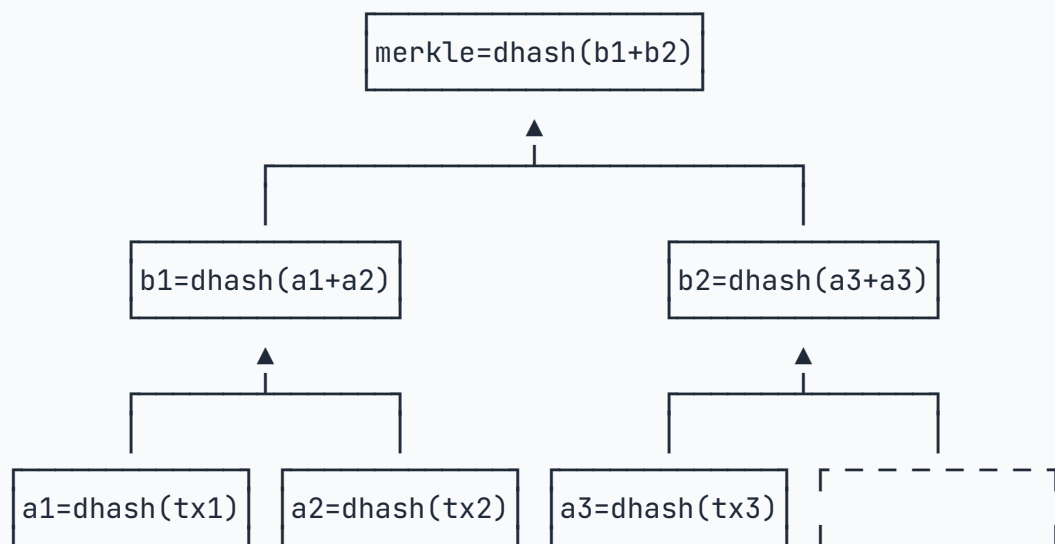
注意到哈希值也可以看做数据，所以可以把 `a1` 和 `a2` 拼起来，`a3` 和 `a4` 拼起来，再计算出两个哈希值 `b1` 和 `b2`：



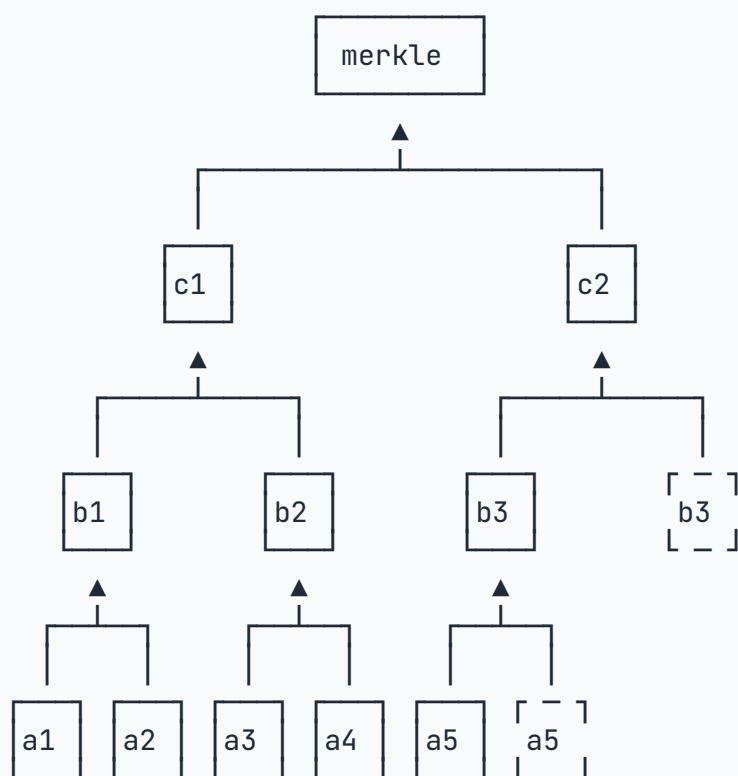
最后，把 `b1` 和 `b2` 这两个哈希值拼起来，计算出最终的哈希值，这个哈希就是Merkle Hash：



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：

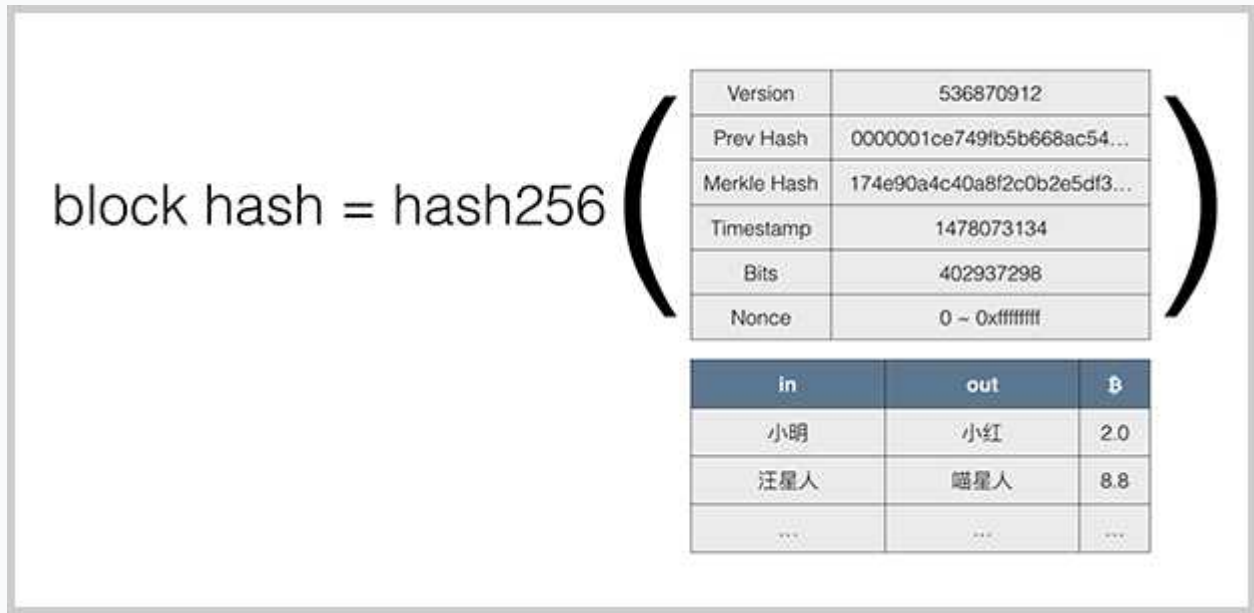


从Merkle Hash的计算方法可以得出结论：修改任意一个交易哪怕一个字节，或者交换两个交易的顺序，都会导致Merkle Hash验证失败，也会导致这个区块本身是无效的，所以，Merkle Hash记录在区块头部，它的作用就是保证交易记录永远无法修改。

## 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：

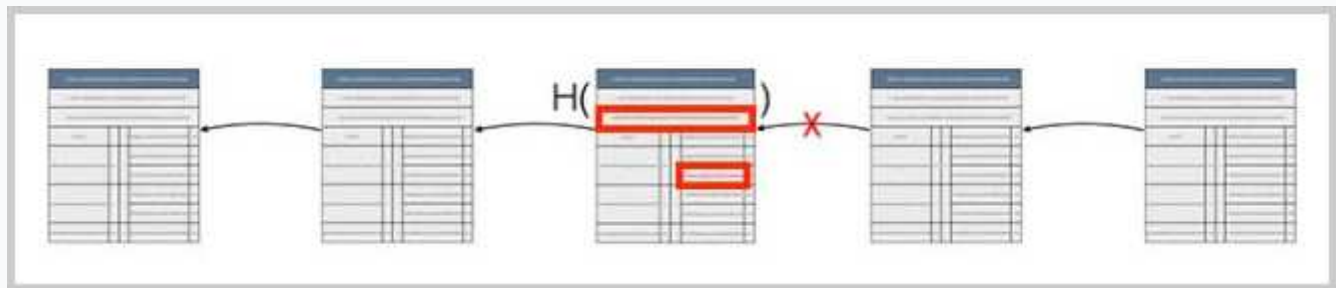


区块头部的Prev Hash记录了上一个区块的Block Hash，这样，可以通过Prev Hash追踪到上一个区块。

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。

## 评论

# P2P交易原理

## 原文链接

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密钥对，这个密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。

#### 💡 提示

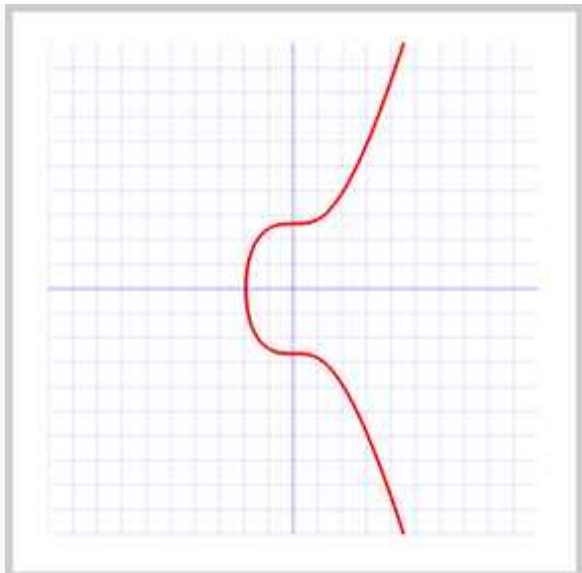
数字签名的三个作用：防伪造，防篡改，防抵赖。

## 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密钥对：私钥本质上就是一个 $1 \sim 2^{256}$ 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 $2^{256}$ 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。

### ⚠ 警告

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**。

### ⚠ 警告

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。 $2^{256}$ 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 $2^{256}$ 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；



- 钱包：是指把私钥记在自己脑袋里。

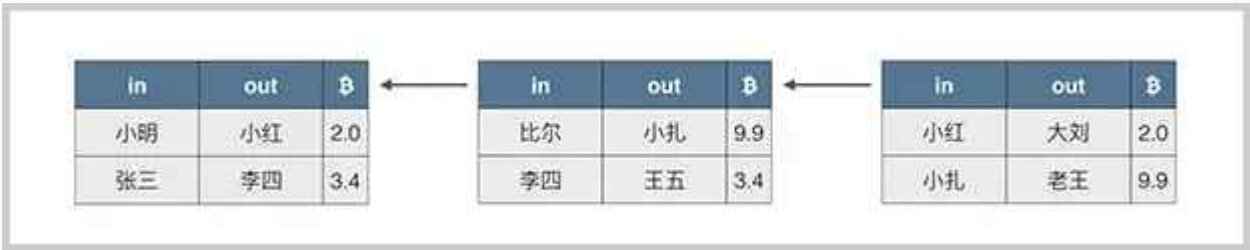
对大多数普通用户来说，想要记住私钥非常困难，所以**强烈不建议使用脑钱包**。

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：



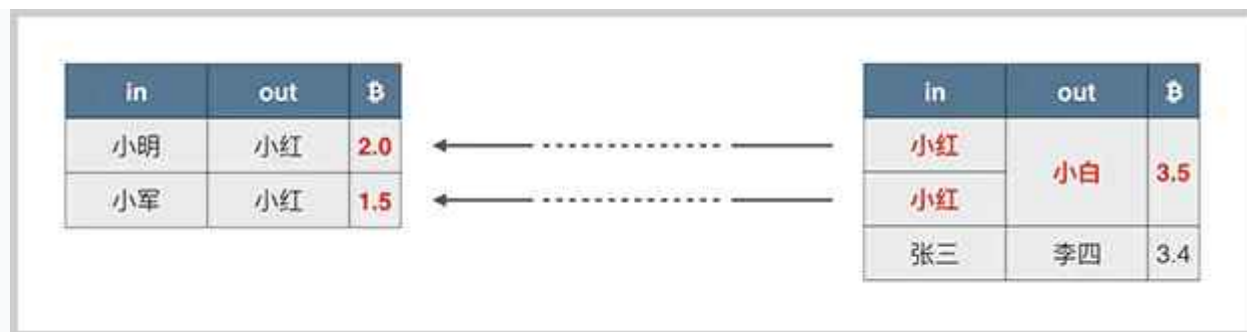
例如，小明把两个比特币转移给小红，这笔交易的输入是小明，输出就是小红。实际记录的是双方的公钥地址。

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：

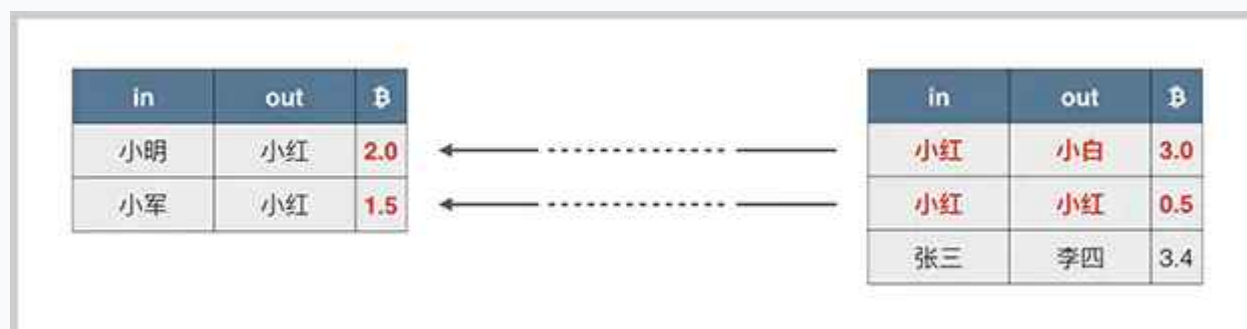


小明给小红 2 个比特币，同时小明又给自己 48 个比特币，这 48 个比特币就是找零。所以，一个交易中，一个输入可以对应多个输出。

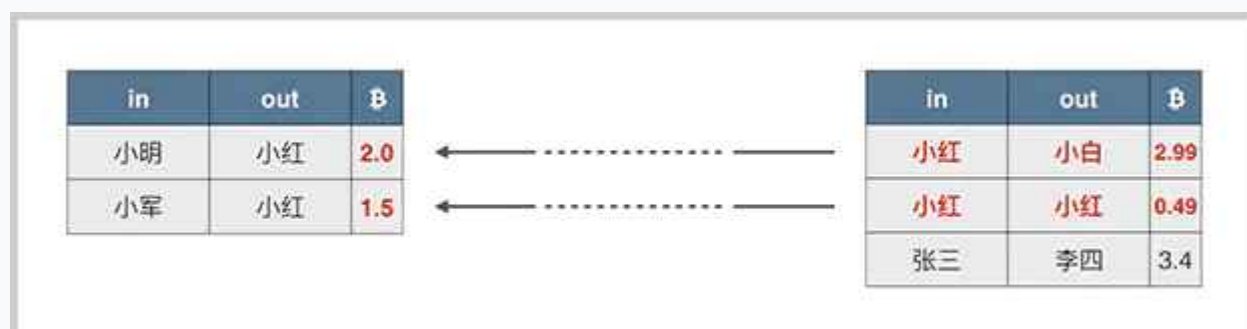
当小红有两笔收入时，一笔 2.0 ，一笔 1.5 ，她想给小白转 3.5 比特币时，就不能单用一笔输出，她必须把两笔钱合起来再花掉，这种情况就是一个交易对应多个输入和1个输出：



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：

$$\text{交易费用} = \text{输入} - \text{输出} = (2.0 + 1.5) - (2.99 + 0.49) = 3.5 - 3.48 = 0.02$$

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 Output。

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。

[评论](#)

# 私钥

## 原文链接

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，可以用数组来模拟。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));
```

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。

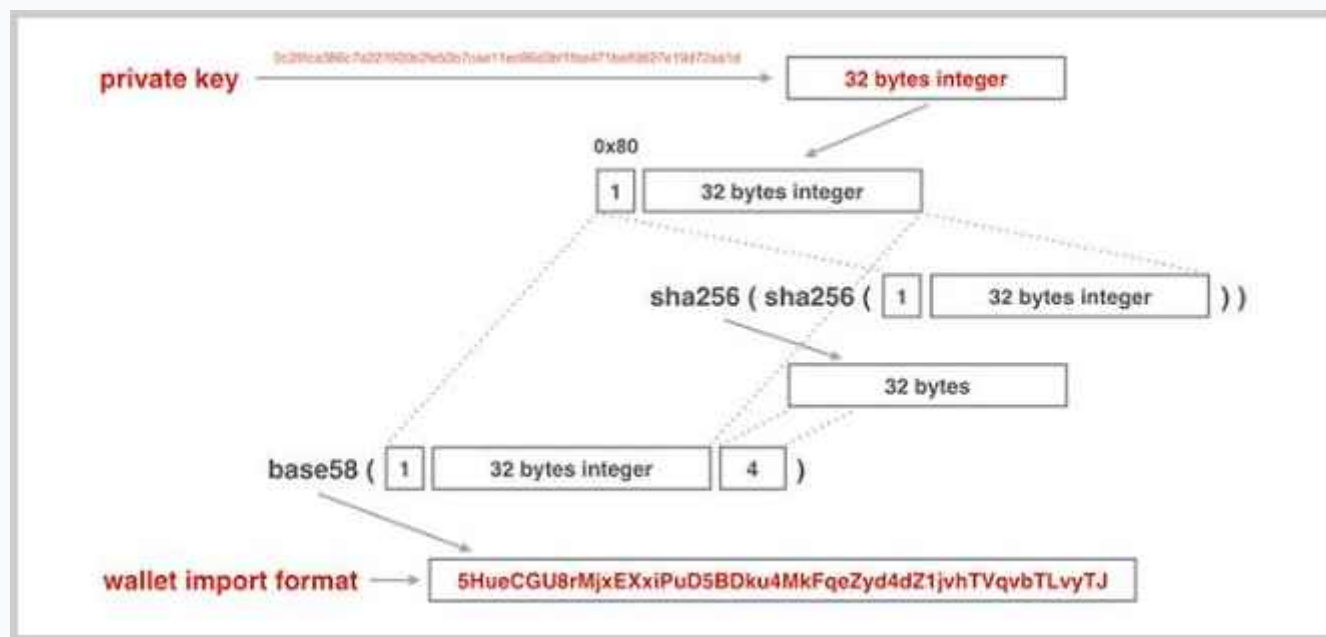
对私钥进行Base58编码有两种方式，一种是非压缩的私钥格式，一种是压缩的私钥格式，它们分别对应非压缩的公钥格式和压缩的公钥格式。

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：



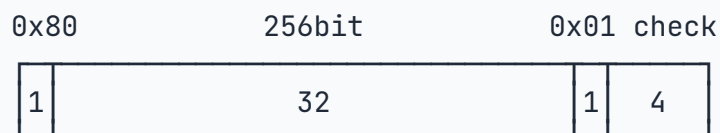
可以使用wif这个库实现WIF编码：

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

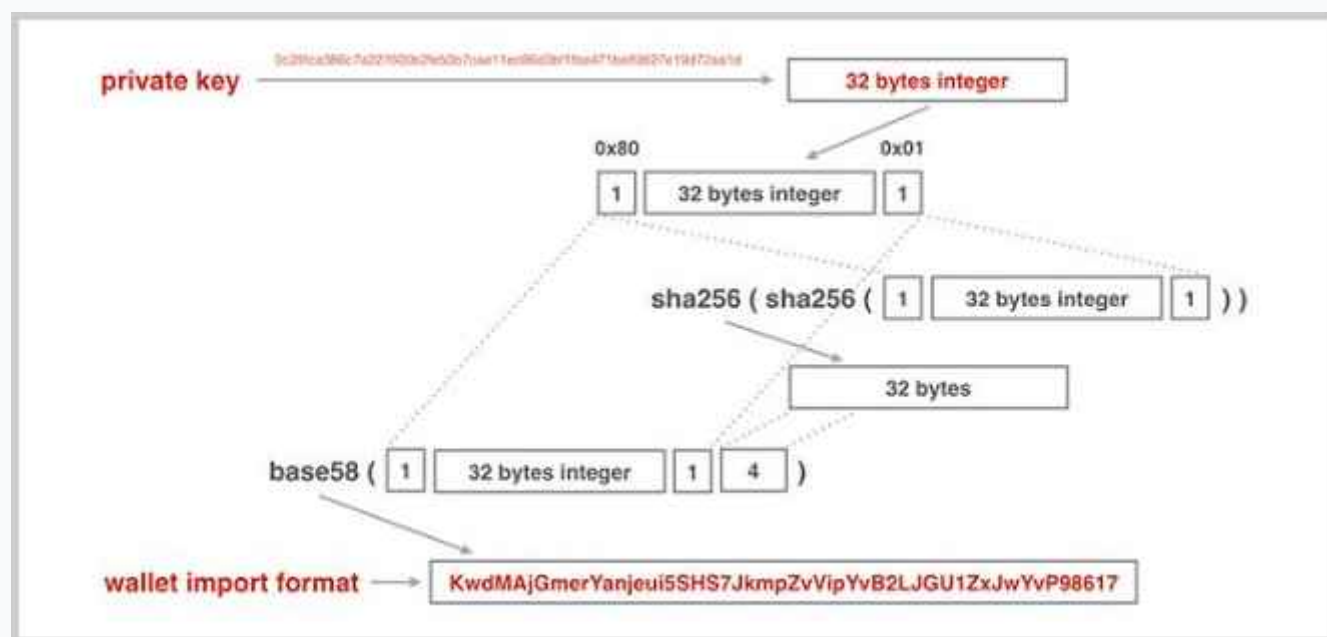
// 十六进制表示的私钥：
let privateKey =
'0c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1d';
// 对私钥编码：
let encoded = wif.encode(
    0x80, // 0x80前缀
    Buffer.from(privateKey, 'hex'), // 转换为字节
    false // 非压缩格式
);
console.log(encoded);
```

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：



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);
```

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());
```

## 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。

[评论](#)



# 公钥和地址

[原文链接](#)

## 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
```

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

## 地址

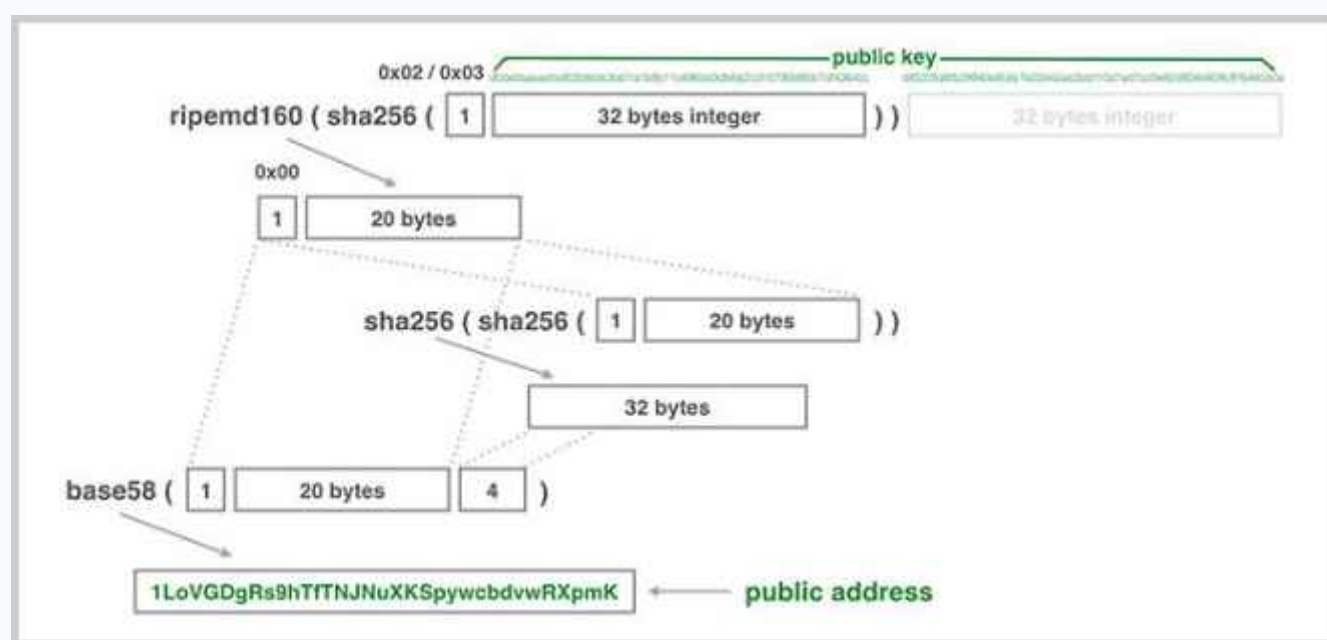
要特别注意，比特币的地址并不是公钥，而是公钥的哈希，即从公钥能推导出地址，但从地址不能反推公钥，因为哈希函数是单向函数。



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 =
    '02d0de0aaefad02b8bdc8a01a1b8b11c696bd3d66a2c5f10780d95b7df42645c',
    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
```

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

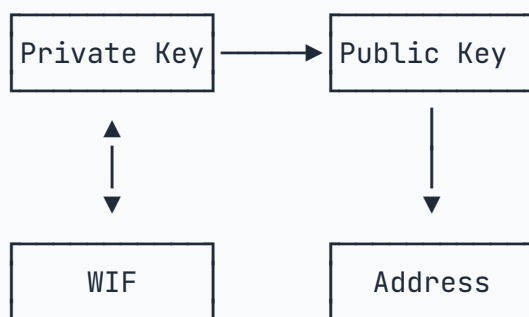
`ECPair.fromPublicKeyBuffer` 构造一个不带私钥的 `ECPair` 即可计算出地址。

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。

## 评论

# 签名

## 原文链接

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'));
```

`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 =  
    '02d0de0aaeafad02b8bdc8a01a1b8b11c696bd3d66a2c5f10780d95b7df42645c',  
    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);
```

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。

[评论](#)

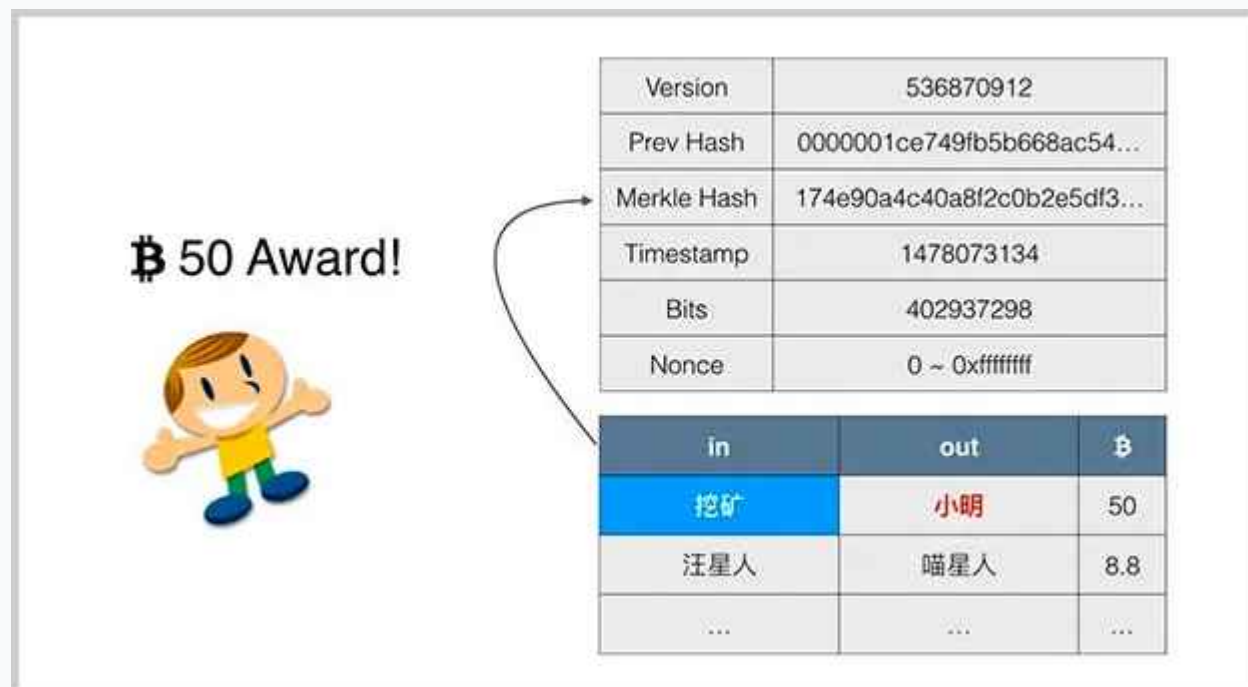
# 挖矿原理

## 原文链接

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。

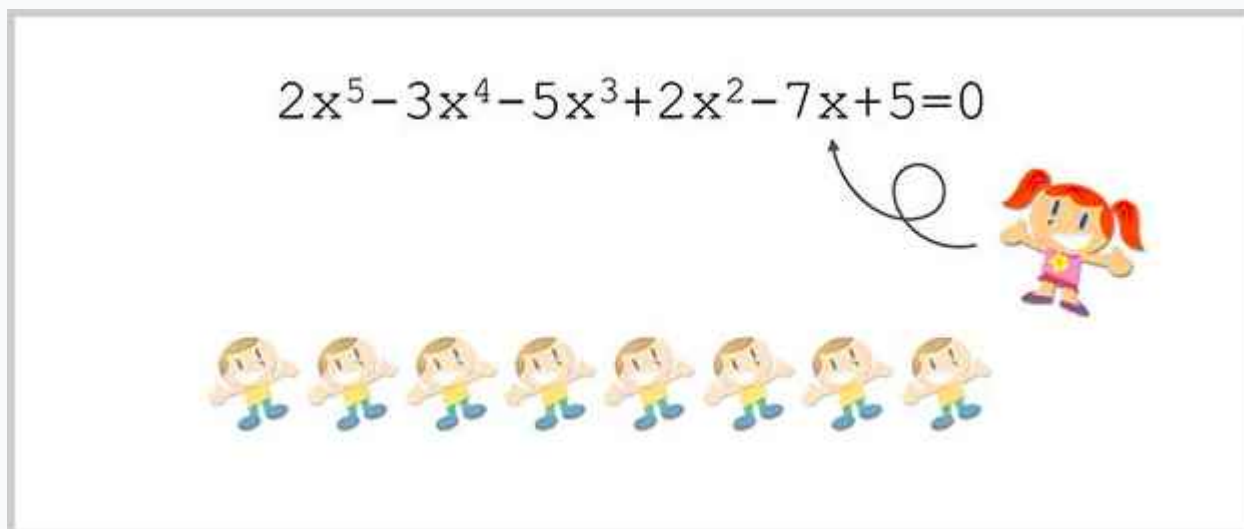
间：



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。

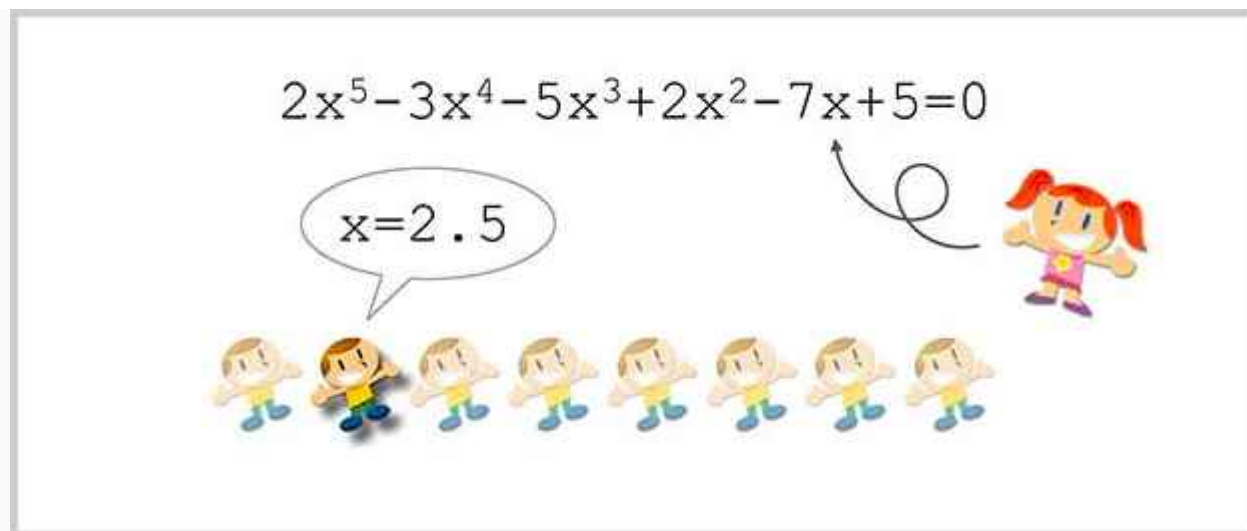
假设某个学校的一个班里，只有一个女生叫小红，其他都是男生。每个男生都想约小红看电影，但是，能实现愿望的只能有一个男生。

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：

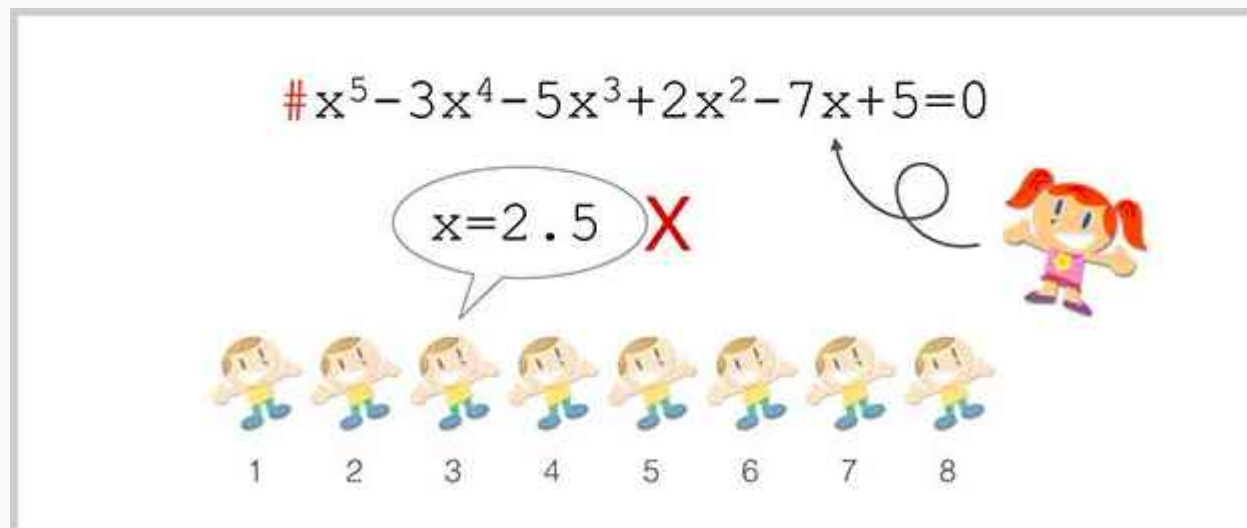




可以看出，解方程很困难，但是，验证结果却比较简单。所以，一个有效的工作量证明在于：计算过程非常复杂，需要消耗一定的时间，但是，验证过程相对简单，几乎可以瞬间完成。

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。

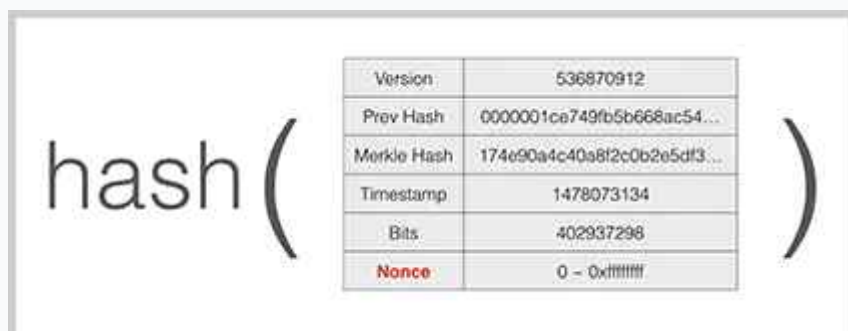
通过工作量证明，可以有效地验证每个人确实都必须花费一定时间做了计算。

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：



Version	536870912
Prev Hash	0000001ce749fb5b668ac54...
Merkle Hash	174e90a4c40a8f2c0b2e5df3...
Timestamp	1478073134
Bits	402937296
Nonce	0 - 0xffffffff

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) =
291656f37cdc493c4bb7b926e46fee5c14f9b76aff28f9d00f5cca0e54f376f
hash256(block data, nonce=1) =
f7b2c15c4de7f482edee9e8db7287a6c5def1c99354108ef33947f34d891ea8d
hash256(block data, nonce=2) =
b6eebc5faa4c44d9f5232631f39ddf4211443d819208da110229b644d2a99e12
hash256(block data, nonce=3) =
00aeaaaf01166a93a2217fe01021395b066dd3a81daffcd16626c308c644c5246
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") =  
652dcd7b75d499bcd6c61d0c4eda96012e3830557de01426da5b01e214b95cd7a  
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 `00` 的哈希值，理论上需要计算256次，因为 `00` 出现的概率是 $16^2=256$ ，实际计算44次：

```
hash256("Hello44") =  
00e477f95283a544ffac7a8efc7dec887f5c073e0f3b43b3797b5dafabb49b5
```

如果我们要找出前3位是 0 的哈希值，理论上需要计算 $16^3=4096$ 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 $16^4=6$ 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 $16^5=104$ 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 $16^6=1677$ 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 $16^{17}$ 次，大约需要计算2.9万亿亿次。

```
17个0 =  $16^{17}$  = 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)
            = 106299667504289830835845558415962632664710558339861315584
            =
0x000000000000000000455d200000000000000000000000000000000000000000
```

注意，难度值的**数值**越小，说明哈希值前面的 `0` 越多，计算的难度越大。

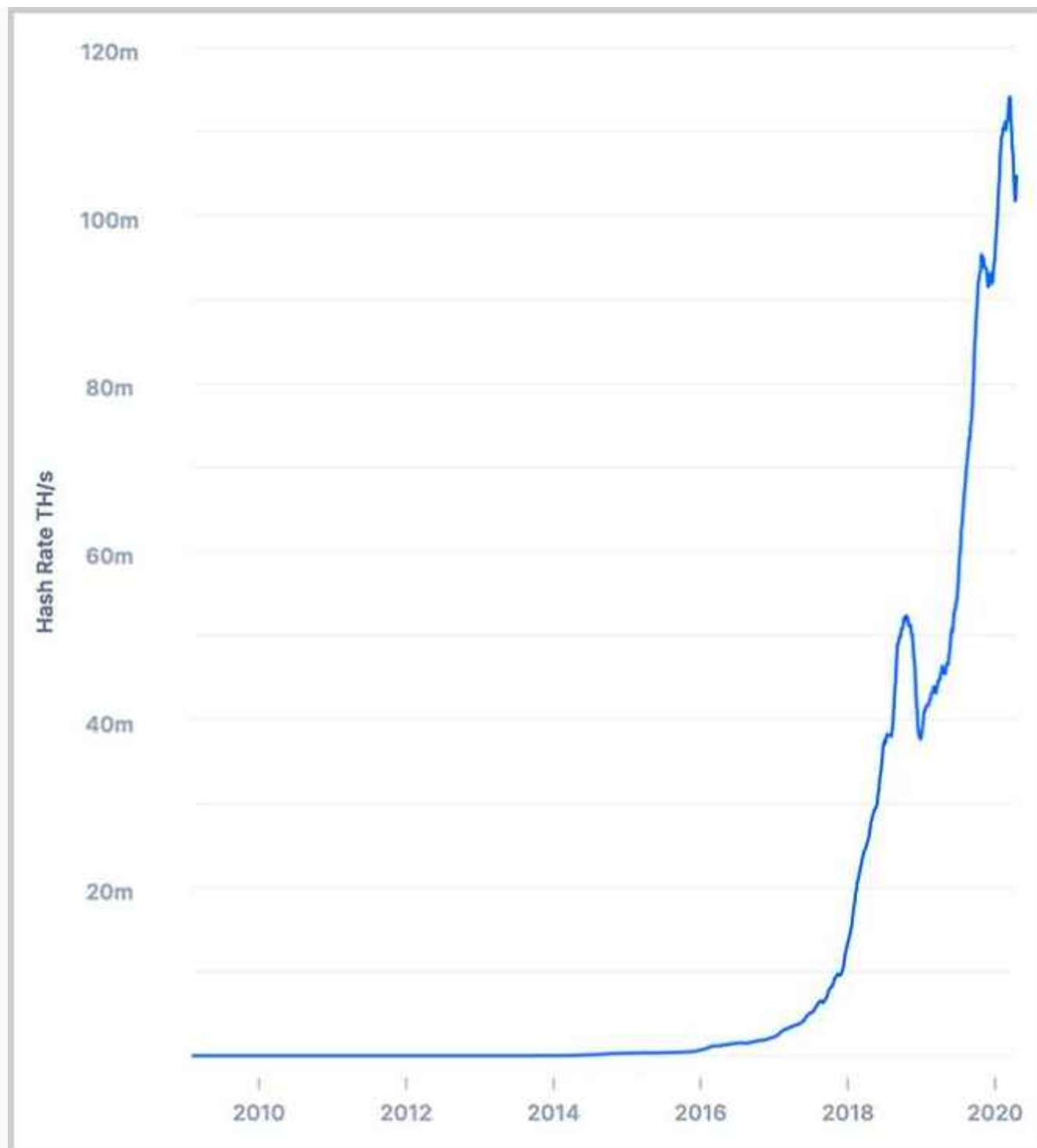
比特币网络的难度是不断变化的，它的难度保证大约每10分钟产生一个区块，而难度值在每2015个区块调整一次：如果区块平均生成时间小于10分钟，说明全网算力增加，难度也会增加，如果区块平均生成时间大于10分钟，说明全网算力减少，难度也会减少。因此，难度随着全网算力的增减会动态调整。

#### ⚠ 比特币难度调整周期

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。

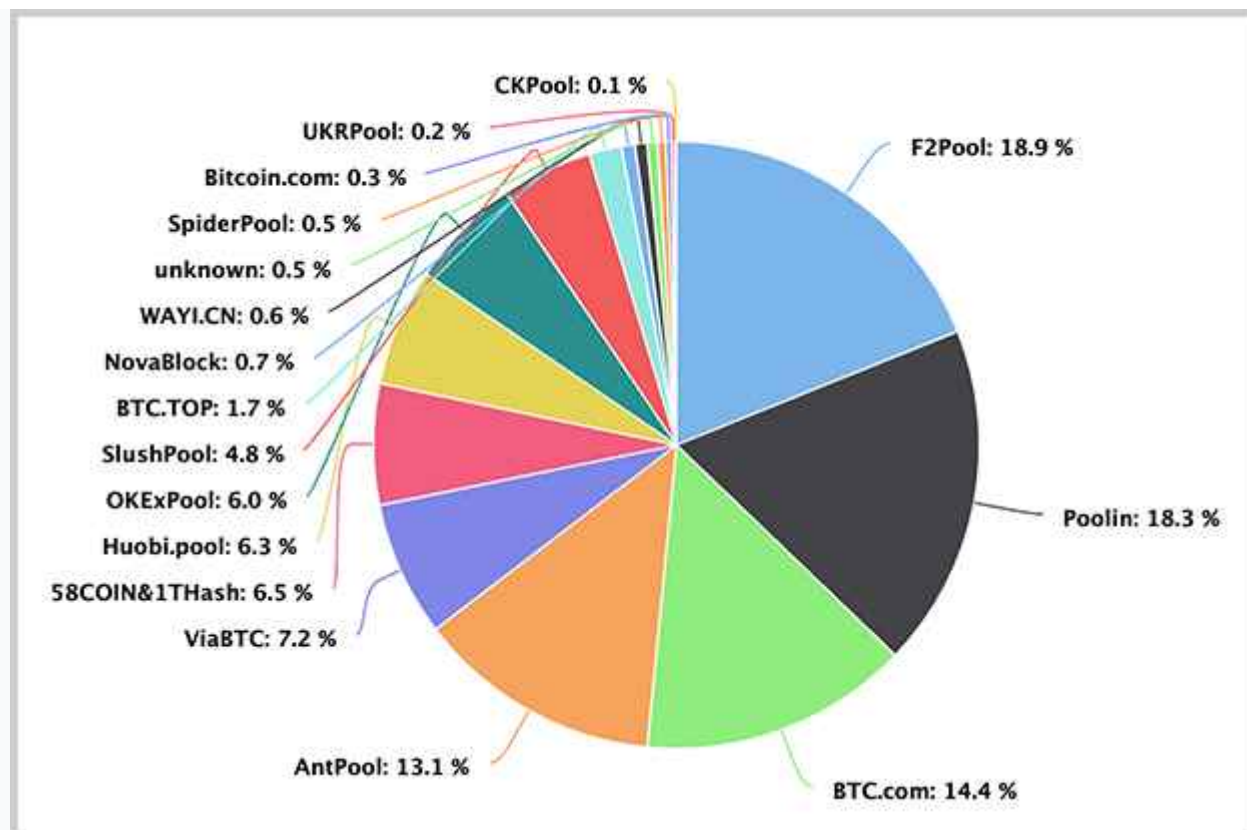
根据比特币每个区块的难度值和产出时间，就可以推算出整个比特币网络的全网算力。

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。

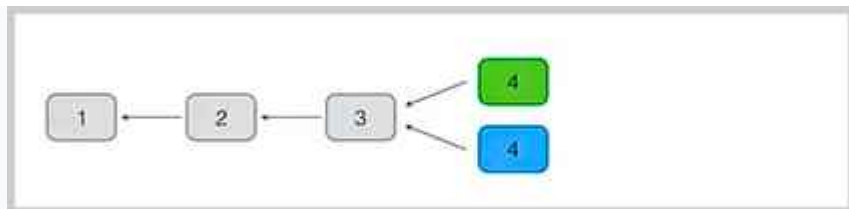
为什么区块可以安全广播？因为Merkle Hash锁定了该区块的所有交易，而该区块的第一个coinbase交易输出地址是该矿工地址。每个矿工在挖矿时产生的区块数据都是不同的，所以无法窃取别人的工作量。

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。

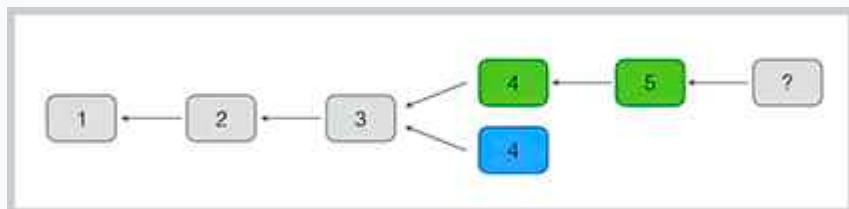
## 共识算法

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：

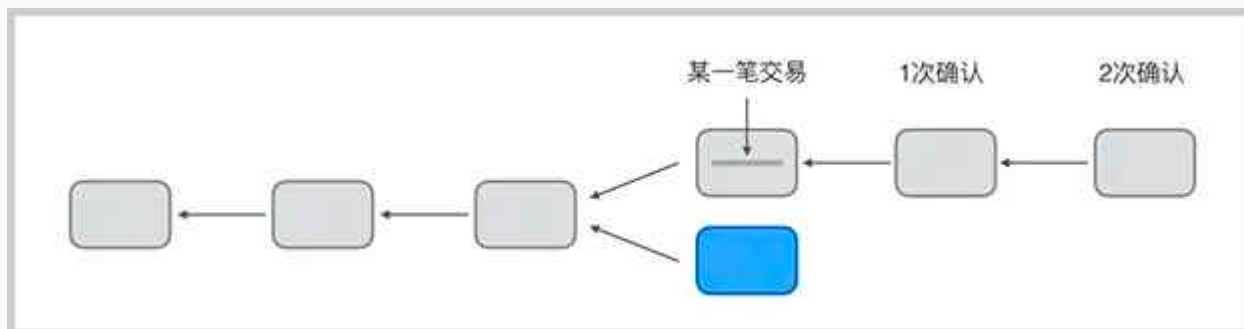


在分叉的情况下，有的矿工在绿色的分叉上继续挖矿，有的矿工在蓝色的分叉上继续挖矿：



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。

## 评论



# 可编程支付原理

## 原文链接

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	
3JXRVxhrk2o9f4w3cQchBLwUeegJBj6BEp	1M6Bzo23yqad8YwzTeRapGXQ76Pb9RRJYJ
	18gJ3jeLdMnr9g3EcbRzXwNssYEN5yFHKE
	1A5Mp8jHcMJEqZUmcsbmtqXfsiGdWYmp6y
	3JXRVxhrk2o9f4w3cQchBLwUeegJBj6BEp

script: 76a914dc5dc65c7e6cc3c404c6dd79d83b22b2fe9f489c88ac

tx: 55142366a67beda9d3ba9bfbd6166e8e95c4931a2b44e5b44b5685597e4c8774	
1M6Bzo23yqad8YwzTeRapGXQ76Pb9RRJYJ	13Kb2ykVGpNTJbxwnrfoyzAwgd4ZpXHv2q

script: 4830450221008ecb5ab06e62a67e320880db70ee8a7020503a055d7c45b73dccc41adf01ea9f602203a0d8f4314342636a6a473fc0b4dd4e49b62be288f0a4d5a23a8f488a768fa9b012103dd8763f8c3db6b77bee743ddafd33c969a99cde9278deb441b09ad7c14cf740f

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
```

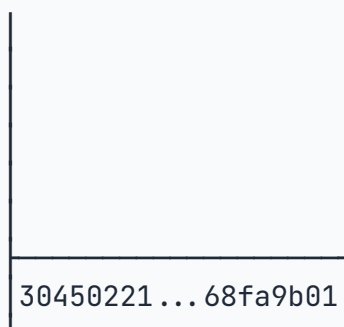
我们再来看解锁脚本。解锁脚本的第一个字节 `48` 表示一个72字节长度的数据，因为十六进制的 `48` 换算成十进制是72。接下来的字节 `21` 表示一个33字节长度的数据。因此，该解锁脚本实际上只有两个数据。

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

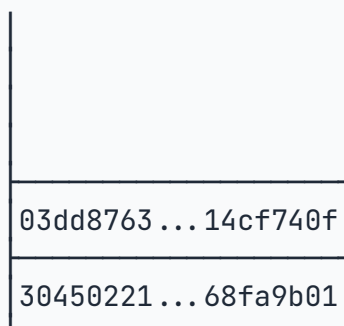
接下来，我们就需要验证这个交易是否有效。要验证这个交易，首先，我们要把解锁脚本和锁定脚本拼到一块，然后，开始执行这个脚本：

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

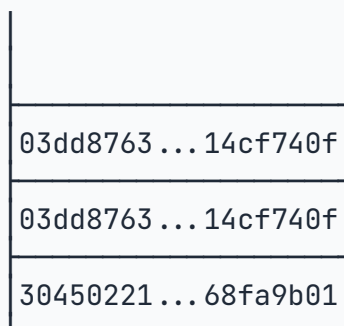
比特币脚本是一种基于栈结构的编程语言，所以，我们要先准备一个空栈，用来执行比特币脚本。然后，我们执行第一行代码，由于第一行代码是数据，所以直接把数据压栈：



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：



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：



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
```

```
T0: 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。

[评论](#)

# 多重签名

## 原文链接

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
```

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。

### 💡 多重签名地址

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（ $1 \leq M \leq N \leq 15$ ）。

## 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。

[评论](#)



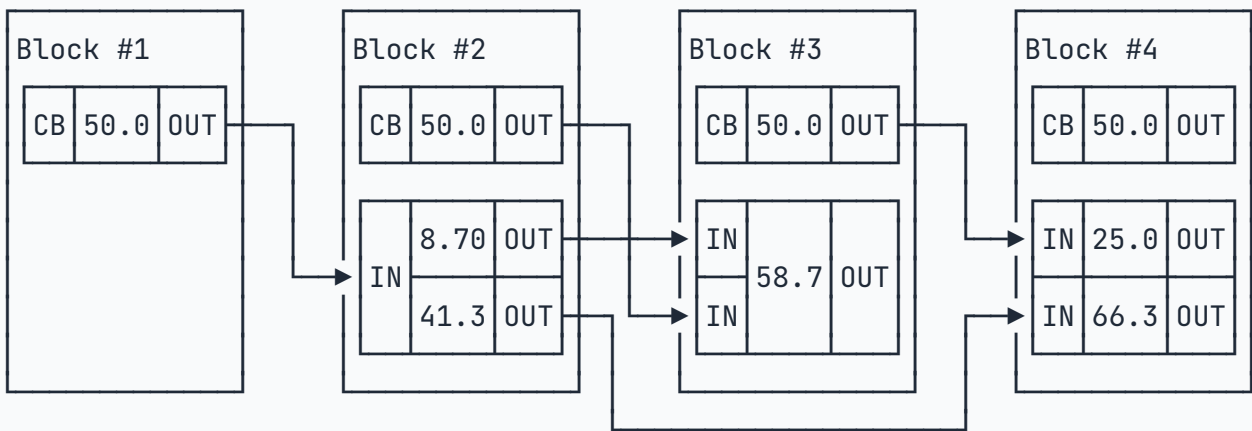
# UTXO模型

[原文链接](#)

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UTXO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。

[评论](#)

# Segwit地址

## 原文链接

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

`bc1qmy63mjadtW8nhz169ukdepwzsyvv4yex5qlmkd` 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 =
  '02d0de0aaeafad02b8bdc8a01a1b8b11c696bd3d66a2c5f10780d95b7df42645c';

// 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); // bc1qmy63mjadt8nhzl69ukdepwzsyvv4yex5qlmkd
```

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， $(r, -s \bmod N)$  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  $(r, -s \bmod N)$ ，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 `bc1` 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）。

评论

# HD钱包

## 原文链接

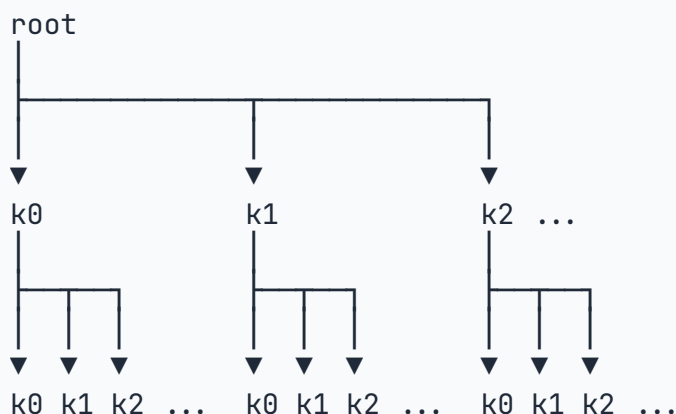
在比特币的链上，实际上并没有账户的概念，某个用户持有的比特币，实际上是其控制的一组UTXO，而这些UTXO可能是相同的地址（对应相同的私钥），也可能是不同的地址（对应不同的私钥）。

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 $k_1$ ，然后计算出 $k_2$ 、 $k_3$ 、 $k_4$ .....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 $k_n$ ：

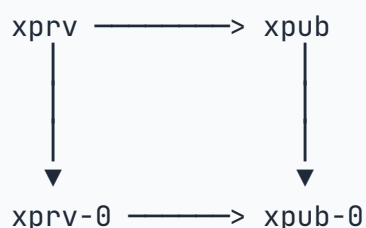
$$k_n = hdkey(k, n)$$

即HD层级实际上是无限的，每一层索引从 $0 \sim 2^{32}$ ，约43亿个子key。这种计算被称为衍生（Derivation）。

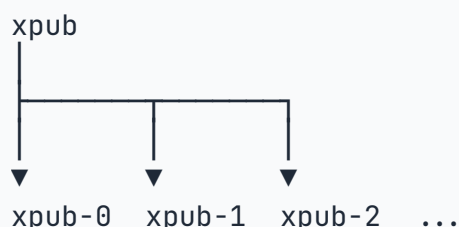
现在问题来了：如何根据某个私钥计算下一层的子私钥？即函数 `hdkey(k, n)` 如何实现？

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。

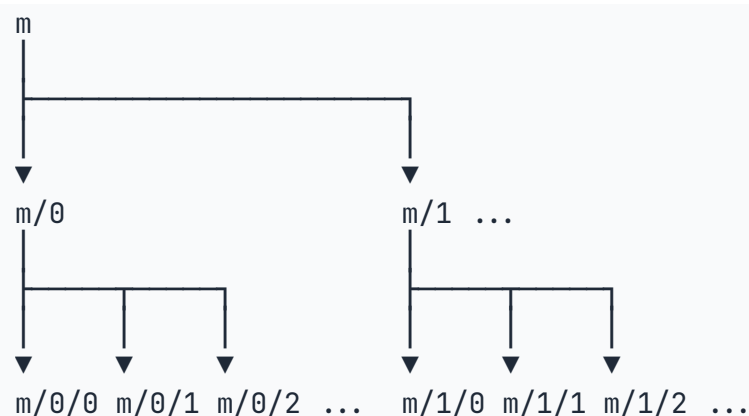
因此，HD钱包通过分层确定性算法，实现了以下功能：

- 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 $0 \sim 2^{32}$ ，原因是扩展私钥中只有4字节作为索引，因此索引范围是 $0 \sim 2^{32}$ 。

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 $0 \sim 2^{31}$ 作为普通衍生索引，而索引 $2^{31} \sim 2^{32}$ 作为硬化衍生索引，硬化衍生索引通常记作 $0'$ 、 $1'$ 、 $2'$ .....，即索引 $0' = 2^{31}$ ， $1' = 2^{31} + 1$ ， $2' = 2^{31} + 2$ ，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  $2^{31} + 44 = 2147483692$ 。从 `m/44'` 无法反向推导出 `m`。

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 $0 \sim 2^{31}$ 。因此，观察钱包能使用的索引是 $0 \sim 2^{31}$ 。

## 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。

[评论](#)

# 钱包层级

## 原文链接

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 =
    'xprv9s21ZrQH143K4EKMS3q1vbJo564QAbs98BfXQME6nk8UCrnXnv8vWg9qmtup3kTug96p5E
    3AvarBhPMScQDqMhEEm41rpYEdXBL8qzVZtwz',
    root = bitcoin.HDNode.fromBase58(xprv);

// m/0:
let 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:
let m_1 = root.derive(1);
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());
```

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 =
    'xprv9s21ZrQH143K4EKMS3q1vbJo564QAbs98BfXQME6nk8UCrnXnv8vWg9qmtup3kTug96p5E
```

```
3AvarBhPMScQDqMhEEm41rpYEdXBL8qzVZtwz',
    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);
```

但不能从xpub推算硬化的子公钥：

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

let
    xprv =
    'xprv9s21ZrQH143K4EKMS3q1vbJo564QAbs98BfXQME6nk8UCrnXnv8vWg9qmtup3kTug96p5E
3AvarBhPMScQDqMhEEm41rpYEdXBL8qzVZtwz',
    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();
```

## 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`，没有一个统一的规范，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 $0 \sim 2^{31}$ 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。

评论

# 助记词

## 原文链接

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。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。

### ⚠ 警告

千万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));
```

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:
let words = bip39.generateMnemonic(256, null,
bip39.wordlists.chinese_simplified);
console.log(words);
```

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：

```
b59a8078d4ac5c05b0c92b775b96a466cd136664bfe14c1d49aff3ccc94d52dfb1d59ee6284
26192eff5535d6058cb64317ef2992c8b124d0f72af81c9ebfaaa
```

该种子即为HD钱包的种子。



### ⚠ 助记词口令

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);
```

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);
```

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
```

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。

[评论](#)

# 地址监控

## 原文链接

一个HD钱包管理的是一组自动计算的地址。以比特币为例，在确定了根扩展私钥 `m` 后，得到一组地址为 `m/44'/0'/0'/0/x`，其中  $x=0\sim 2^{31}$ 。

HD钱包需要在链上监控每个TX的输入和输出，看看上述管理的一组地址是否存在与输入和输出中。如果作为输入，则钱包余额减少，如果作为输出，则钱包余额增加。

现在问题来了：如何根据TX的输入和输出地址快速判断这些地址中是否存在HD钱包管理的地址？

首先，可用的地址高达  $2^{31}$  个，这个数太大了，用户不可能用完，因此，HD钱包只会预生成前1000个地址（即索引号为0~999）并保存在本地数据库中，如果不够了，再继续扩展1000个，这样，HD钱包管理的地址数量不会太大。

其次，要上千个地址集合中快速判断某个地址是否存在，查询数据库是一个非常低效的方式。以哈希表存储在内存中虽然效率很高，但管理的集合数量太多，占用的内存会非常大。

要做到高效的查询和低空间占用率，可以使用布隆过滤器（Bloom Filter），它是由Burton Howard Bloom在1970年提出的，其原理是将每个元素通过若干个哈希函数映射成一个位数组的若干个点，将其置1。检索的时候，先计算给定元素对应位是否全1，如果是全1，则给定元素很可能存在，否则，元素必定不存在。

因此，Bloom Filter有个重要特点，就是判断元素时，如果不存在，那么肯定不存在，如果存在，实际上是以一定概率存在（例如，99%），还需要再次从数据库查询以确定元素真的存在。

Bloom Filter的缺点就是它无法100%准确判断存在，此外，添加新的元素到Bloom Filter很容易，但删除元素就非常困难。构造Bloom Filter时，要先预估元素个数并给定存在概率，才能计算所需的内存空间。

Bloom Filter广泛用于垃圾邮件地址判断，CDN服务等。Bloom Filter也非常适合HD钱包监控链上每个交易的地址。

## 小结

HD钱包通过Bloom Filter可以高效监控链上的所有地址，并根据是否是本地管理的地址决定如何计算钱包余额。

## 评论

# 以太坊

## 原文链接

以太坊（Ethereum）是一个支持智能合约的区块链平台，它与比特币最大的不同是，以太坊通过一个虚拟机（EVM）可以运行智能合约。

以太坊是Vitalik Buterin（维塔利克·布特林，人称V神）在2013年提出的概念，Vitalik最早参与了比特币社区的开发，并希望比特币把功能受限的脚本扩展成图灵完全的编程环境，但没有得到比特币开发社区的认同，于是他决定另起炉灶，打造一个新的区块链平台，目标是运行去中心化的程序。

以太坊从2015年正式启动并运行，期间经历过DAO攻击造成的硬分叉。和比特币类似，以太坊也通过PoW进行挖矿，后改为PoS挖矿，其挖出的平台币叫以太币（Ether），目前每个区块奖励是2 Ether，约13~15秒左右出一个块。

和比特币相比，以太坊在以下几点上有所不同：

## 账户模型

比特币使用的UTXO模型是一种对开发友好、易于实现清结算的模型，但对用户不友好，因为普通用户所认知的账户是一个账号、对应余额变动的模型。以太坊的账户模型和比特币不同，它就是余额模型，即交易引发账户余额的变动，这与传统金融账户一致。

## 智能合约

从比特币的可编程支付原理可知，任何支付实际上都是在执行比特币脚本，只有脚本成功执行，支付才能成功。

以太坊的交易与之类似，并且更进一步，它实现了一个图灵完备的脚本语言，运行在EVM（Ethereum Virtual Machine，以太坊虚拟机）中，任何人都可以编写合法的脚本来执行任意逻辑（有很多限制），例如，定义一种新的代币，抵押贷款等。

## 评论

# 账户

## 原文链接

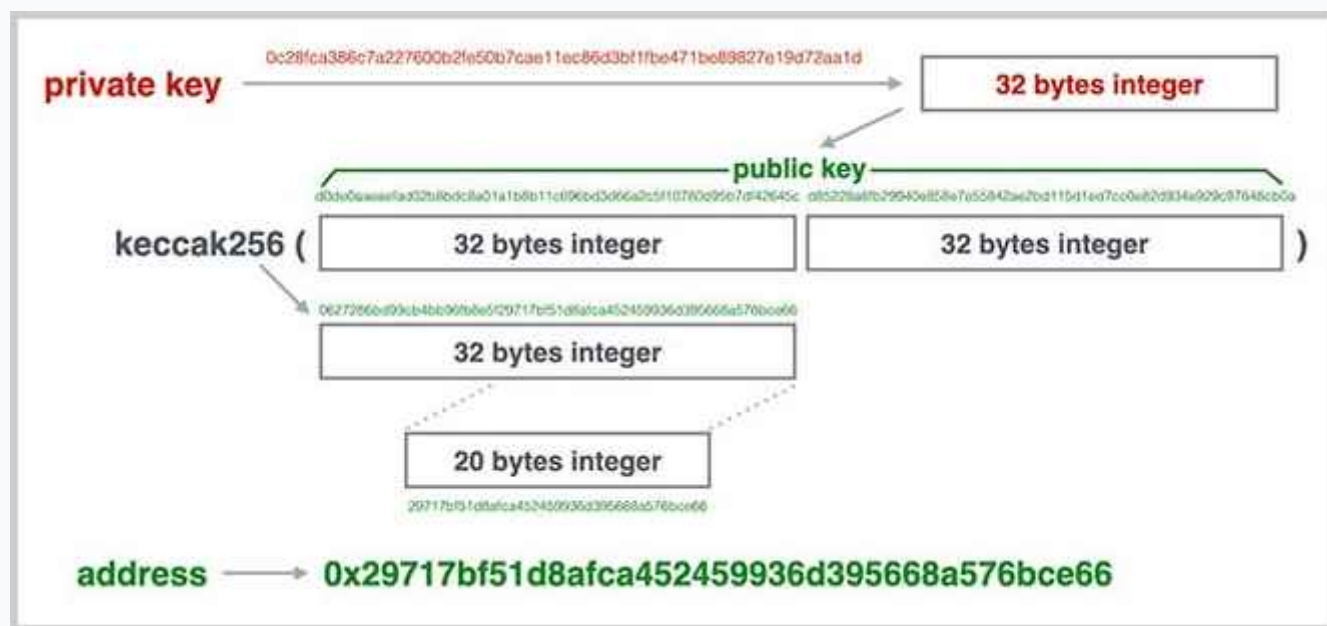
以太坊账户负责存储用户的以太坊余额。对大多数普通用户来说，以太坊账户和银行账户非常类似，通常只需要一个账户即可。

确切地说，以太坊账户分为外部账户和合约账户两类：

- 外部账户：即普通用户用私钥控制的账户；
- 合约账户：一种拥有合约代码的账户，它不属于任何人，也没有私钥与之对应。

本节我们仅讨论普通用户使用的外部账户。

和比特币类似，一个以太坊账户就是一个公钥哈希后得到的地址，它是由一个私钥推导出对应的公钥，然后再计算出地址。其中，私钥与公钥算法与比特币完全相同，均为secp256k1椭圆曲线，但和比特币不同的是，以太坊采用非压缩公钥，然后直接对公钥做keccak256哈希，得到32字节的哈希值，取后20字节加上 `0x` 前缀即为地址：



用代码实现如下：

```
const
  randomBytes = require('randombytes'),
  ethUtil = require('ethereumjs-util');

// 生成256bit的随机数作为私钥：
```

```
let priKey = randomBytes(32).toString('hex');
// 计算公钥(非压缩格式):
let pubKey = ethUtil.privateToPublic(new Buffer(priKey,
'hex')).toString('hex');
// 计算地址:
let addr = ethUtil.pubToAddress(new Buffer(pubKey, 'hex')).toString('hex');

console.log('Private key: 0x' + priKey);
console.log('Public key: 0x' + pubKey);
console.log('Address: 0x' + addr);
```

和比特币采用Base58或Bech32编码不同，以太坊对私钥和地址均采用十六进制编码，因此它没有任何校验，如果某一位写错了，仍然是一个有效的私钥或地址。

keccak256哈希算法在以太坊中也被称为SHA3算法，但是要注意，keccak算法原本是SHA3的候选算法，然而在SHA3最后的标准化时，对keccak做了改进，因此，标准的SHA3算法和keccak是不同的，只是以太坊在开发时就选择了尚未成为SHA3标准的keccak算法。后续我们在讨论以太坊的哈希算法时，一律使用keccak256而不是SHA3-256。

## 带校验的地址

因为以太坊的地址就是原始哈希的后20字节，并且以十六进制表示，这种方法简单粗暴，但没有校验。地址中任何数字出错都仍是一个有效地址。为了防止抄错，以太坊通过EIP-55实现了一个带校验的地址格式，它的实现非常简单，即对地址做一个keccak256哈希，然后按位对齐，将哈希值 $\geq 8$ 的字母变成大写：

```
original addr = 0x29717bf51d8afca452459936d395668a576bce66
keccak hash =   e72ecce2eb2ed0ffab5e05f043ee68fab3df759d...
checksum addr = 0x29717BF51D8AFcA452459936d395668A576Bce66
```

因此，以太坊地址就是依靠部分变成大写的字母进行校验，它的好处是带校验的地址和不带校验的地址对钱包软件都是一样的格式，缺点是有很小的概率无法校验全部小写的地址。

```
const ethUtil = require('ethereumjs-util');

console.log('is valid address: ' +
ethUtil.isValidAddress('0x29717bf51d8afca452459936d395668a576bce66')); //
true
console.log('is valid checksum address: ' +
```



```
ethUtil.isValidChecksumAddress('0x29717BF51D8AFcA452459936d395668A576Bce66')
)); // true
console.log('is valid checksum address: ' +
ethUtil.isValidChecksumAddress('0x29717BF51D8AFcA452459936d395668A576BcE66')
)); // false
```

下面这个程序可以自动搜索指定前缀地址的私钥：

```
const randomBytes = require('randombytes');
const ethUtil = require('ethereumjs-util');

// 搜索指定前缀为'0xAA...'的地址：
let prefix = '0xAA';

if (/^0x[a-fA-F0-9]{1,2}$/.test(prefix)) {
  let
    max = parseInt(Math.pow(32, prefix.length-2)),
    qPrefix = prefix.toLowerCase().substring(2),
    prettyPriKey = null,
    prettyAddress = null,
    priKey, pubKey, addr, cAddr, i;

  for (i=0; i<max; i++) {
    priKey = randomBytes(32).toString('hex');
    pubKey = ethUtil.privateToPublic(new Buffer(priKey,
'hex')).toString('hex');
    addr = ethUtil.pubToAddress(new Buffer(pubKey,
'hex')).toString('hex');
    if (addr.startsWith(qPrefix)) {
      cAddr = ethUtil.toChecksumAddress('0x' + addr);
      if(cAddr.startsWith(prefix)) {
        prettyPriKey = priKey;
        prettyAddress = cAddr;
        break;
      }
    }
  }

  if (prettyPriKey === null) {
    console.error('Not found.');
```

```
} else {
```

```
        console.log('Private key: 0x' + prettyPriKey);
        console.log('Address: ' + prettyAddress);
    }
} else {
    console.error('Invalid prefix.');
```

原理是不断生成私钥和对应的地址，直到生成的地址前缀满足指定字符串。一个可能的输出如下：

```
Private key:
0x556ba88aea1249a1035bdd3ec2d97f8c60404e26ecfcd7757e0906885d40322e
Address: 0xAA6f2ea881B96F87152e029f69Bd443834D99f97
```

### ⚠ 警告

如果你想用这种方式生成地址，请确保电脑无恶意软件，并在断网环境下用Node执行而不是在浏览器中执行。

## HD钱包

因为以太坊和比特币的非对称加密算法是完全相同的，不同的仅仅是公钥和地址的表示格式，因此，比特币的HD钱包体系也完全适用于以太坊。用户通过一套助记词，既可以管理比特币钱包，也可以管理以太坊钱包。

以太坊钱包的派生路径是 `m/44'/60'/0'/0/0`，用代码实现如下：

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

// 助记词和口令：
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...uLg9r9kF
console.log('xpub: ' + root.neutered().toBase58()); //
xpub661MyMwA...oy32fcRG

// 生成派生key:
let child0 = root.derivePath("m/44'/60'/0'/0/0");
let prvKey = child0.keyPair.d.toString(16);
let pubKey = ethUtil.privateToPublic(new Buffer(prvKey,
'hex')).toString('hex');
let address = '0x' + ethUtil.pubToAddress(new Buffer(pubKey,
'hex')).toString('hex');
let checksumAddr = ethUtil.toChecksumAddress(address);

console.log("      prv m/44'/60'/0'/0/0: 0x" + prvKey); //
0x6c03e50ae20af44b9608109fc978bdc8f081e7b0aa3b9d0295297eb20d72c1c2
console.log("      pub m/44'/60'/0'/0/0: 0x" + pubKey); //
0xff10c2376a9ff0974b28d97bc70daa42cf85826ba83e985c91269e8c975f75f7d56b9f507
1911fb106e48b2dbb2b30e0558faa2fc687a813113632c87c3b051c
console.log("      addr m/44'/60'/0'/0/0: " + address); //
0x9759be9e1f8994432820739d7217d889918f2f07
console.log("check-addr m/44'/60'/0'/0/0: " + checksumAddr); //
0x9759bE9e1f8994432820739D7217D889918f2f07
```

因为以太坊采用账户余额模型，通常情况下一个以太坊地址已够用。如果要生成多个地址，可继续派生 `m/44'/60'/0'/0/1` 、 `m/44'/60'/0'/0/2` 等。

## 小结

以太坊的私钥和公钥采用和比特币一样的ECDSA算法和secp256k1曲线，并且可以复用比特币的HD钱包助记词；

以太坊的地址采用对非压缩公钥的keccak256哈希后20字节，并使用十六进制编码，可以通过大小写字母实现地址的校验。

## 评论

# 区块结构

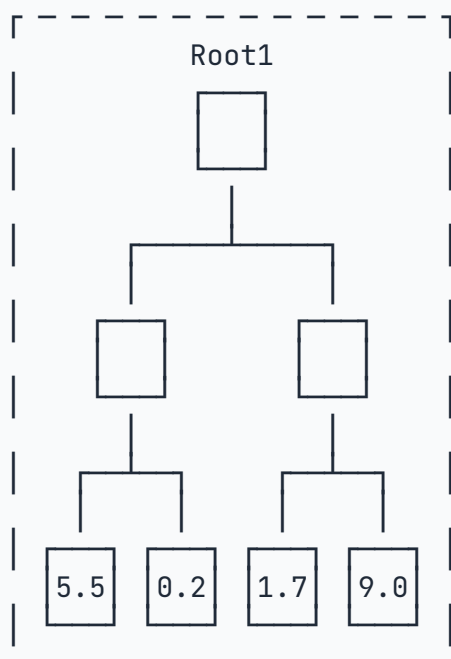
[原文链接](#)

比特币的区块链是由PoW保证每个区块都指向前一个区块，而在每一个区块内部，由一个独立的Merkle Tree来保证所有交易的不可篡改。用户的比特币是以UTXO的方式存储的，因此，比特币的交易就是不断地消耗现有的UTXO，并产生新的UTXO。

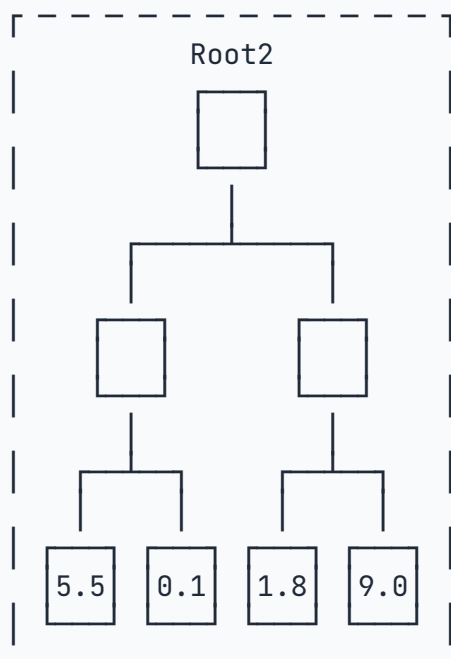
而以太坊采用的是账户模型，如果小明的账户在某个区块的资产是1 ETH，当小明给小红转账0.2 ETH后，刨除手续费，他的账户还剩下约0.8 ETH。由于小明的账户地址不变，所以，以太坊的区块结构必须能在每个区块持续地跟踪并记录小明的账户余额变动。因此，和比特币相比，以太坊的区块数据结构更加复杂。

## Merkle Patricia Tree

以太坊存储账户数据的数据结构是MPT：Merkle Patricia Tree，它是一种改进的Merkle Tree。当MPT的每个叶子结点的值确定后，计算出的Root Hash就是完全确定的。例如，在第一个区块中，4个账户的余额确定后，即可确定 **Root1**：



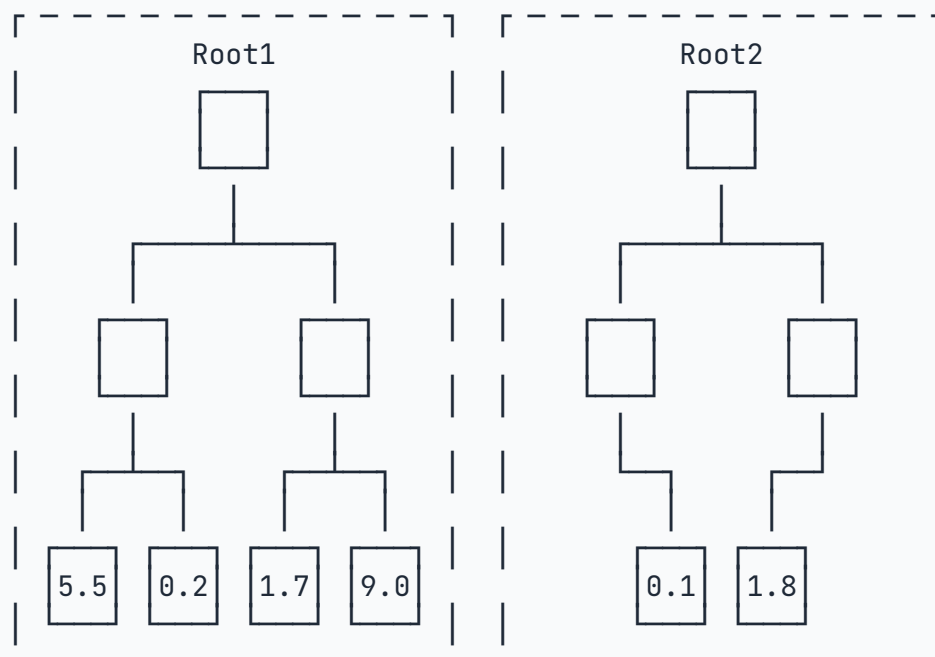
在第二个区块中，如果发生了转账，将计算出一个新的 **Root2**：



每一个区块通过Root Hash将完全确定所有账户的状态，所以，从全局看，以太坊就是一个状态机，每个区块通过记录一个 `stateRoot` 来表示一个新状态。如果给定某个区块的 `stateRoot`，我们肯定能完全确定所有账户的所有余额等信息。因此，`stateRoot` 就被称为当前的世界状态。

有的同学可能会思考，如果第一个区块只有几个账户，随着账户的增加，如果有数百万个账户，到后面岂不是区块存储的数据量会越来越大？

实际上，每个区块的 `stateRoot` 表示的是一个完全状态的逻辑树，但每个区块记录的数据只包括修改的部分，如果我们观察第二个区块的树，它实际上只记录修改的两个账户，以及两个账户因修改后导致的上层路径的Hash发生的变化：



想要将一个有几百万节点的树完整地放入内存需要消耗大量的内存，而以太坊全节点也并不会将整颗逻辑树放入内存。实际上，每个节点的数据被存放到LevelDB中，节点仅在内存中存储当前活动的一些账户信息。如果需要操作某个不在内存的账户，则会将其从LevelDB加载到内存。如果内存不够，也会将长期不活动的节点从内存中移除，因为将来可以通过节点的路径再次从LevelDB加载。

## 账户数据

一个以太坊账户由4部分数据构成：

- nonce
- balance
- storageRoot
- codeHash

其中，`nonce` 是一个递增的整数，每发送一次交易，`nonce` 递增 1，因此，`nonce` 记录的就是交易次数。

`balance` 记录的就是账户余额，以 `wei` 为单位，1 Ether等于 $10^{18}$ wei。

如果一个账户是合约账户，则 `storageRoot` 存储合约相关的状态数据，`codeHash` 存储合约代码的Hash。对于外部账户，这两部分数据都是空。

## 区块数据

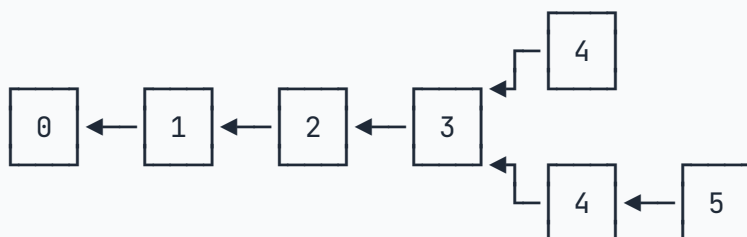
一个以太坊区块由区块头和一系列交易构成。区块头除了记录parentHash（上一个区块的Hash）、stateRoot（世界状态）外，还包括：

- sha3Uncles：记录引用的叔块；
- transactionRoot：记录当前区块所有交易的Root Hash；
- receiptsRoot：记录当前区块所有交易回执的Root Hash；
- logsBloom：一个Bloom Filter，用于快速查找Log；
- difficulty：挖矿难度值；
- number：区块高度，严格递增的整数；
- timestamp：区块的时间戳（以秒为单位）；
- .....

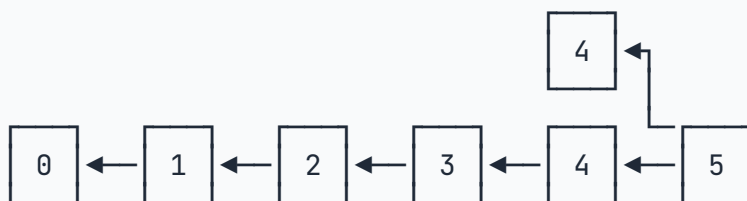
`transactionRoot` 和 `receiptsRoot` 也是两个MPT树，但他两和 `stateRoot` 不同，他们仅表示当前区块的两棵树，与前面的区块状态无关。

## 叔块

在2022年9月升级之前，以太坊采用和比特币类似的PoW挖矿，只是算法为改进的Ethash算法。PoW挖矿肯定会产生分叉，但由于最长链共识，最终某个分叉将胜出：



但是和比特币不同的是，虽然 `#4` 的竞争导致一个胜出另一个失败，但以太坊鼓励后续的区块引用另一个废弃的 `#4` 块，这种引用的废弃块被称为叔块（Uncle Block）：





区块头记录的 `sha3Uncles` 就是叔块，一个区块可引用0~2个叔块，且叔块高度必须在前7层之内。

叔块的目的是给予竞争失败的矿工部分奖励，避免出现较长的分叉。

## 小结

以太坊的核心数据结构是以Merkle Patricia Tree记录的世界状态，每个区块通过打包新的交易，从而导致世界状态的变化。

### 评论

# 交易

## 原文链接

在比特币中，交易就是消耗已有UTXO，并通过执行脚本产生新的UTXO，其中隐含的新旧差额即为矿工手续费。

在以太坊中，交易也需要手续费，手续费被称为Gas（汽油），它的计算比比特币要复杂得多。

以太坊除了最基本的转账：即从一个账户支付Ether到另一个账户，还支持执行合约代码。合约代码是图灵完备的编程语言，通过EVM（以太坊虚拟机）执行。如果某个合约编写了一个无限循环，那么所有节点执行该合约代码，岂不是永远无法结束？

为了保证合约代码的可靠执行，以太坊给每一个虚拟机指令都标记了一个Gas基本费用，称为gasUsed。例如，加减计算的费用是3，计算SHA3的费用是30，输出日志的费用是375，写入存储的费用高达20000。总的来说，消耗CPU比消耗存储便宜，简单计算比复杂计算便宜，读取比写入便宜。

除了gasUsed外，用户还需要提供一个gasPrice，以Gwei（1Gwei=10<sup>9</sup>Wei）为单位。通过竞价得到一个矿工愿意接受的gasPrice。如果一个交易消耗了120000的gasUsed，而gasPrice是50 Gwei，则交易费用是：

```
120000 x 50 Gwei = 6000000 Gwei = 0.006 Ether
```

但是在执行代码的时候，存在条件判断、循环等语句，同一段代码，执行的结果也可能不同，因此，事前预计一个合约执行要花费多少Gas，不现实。

所以以太坊规定，一笔交易，先给出gasPrice和gasLimit，如果执行完成后有剩余，剩余的退还，如果执行过程中消耗殆尽，那么交易执行失败，但已执行的Gas不会退。

太复杂了是不是？我们还是举个例子。

假定某个账户想执行一笔交易，他给出gasPrice为50 Gwei，预估gasUsed约120000，设定gasLimit为150000，则预支付的Ether为：

```
150000 x 50 Gwei = 7500000 Gwei = 0.0075 Ether
```

如果账户的Ether余额不足0.0075，则该交易根本无法发送。如果账户余额等于或超过0.0075，例如余额为0.008，则矿工可以将该交易打包。假设实际执行消耗的gasUsed为120000，则交易

费用0.006，账户剩余0.002。

很少有交易能准确预估gasUsed，只有标准转账交易是21000，因此，标准的转账交易gasLimit可以设置为21000（即恰好消耗完毕无剩余）。

Gas Price是全网用户竞价产生的，它时刻在波动。如果交易少，Gas Price将下降，如果交易多，网络拥堵，则Gas Price将上升。以太坊的Gas价格可以在[Etherscan](#)跟踪。

## 交易回执

以太坊区块为每一笔交易都会产生一笔回执（Receipt），表示交易的最终状态。一个回执信息主要包括：

- status：执行结果，1表示成功，0表示失败；
- gasUsed：已消耗的Gas数量；
- txHash：交易Hash；
- logs：交易产生的日志；
- .....

## 转账交易

转账交易是指两个外部账号转移Ether，我们以交易[0xb940...4ad7](#)为例，可以看到：

- Transaction Hash: 0xb940...4ad7，这是交易Hash，即交易的唯一标识；
- Status: Success，表示交易成功；
- From: 0x0FFf...bBc4，交易的发送方；
- To: 0x5b2a...5a46，交易的接收方；
- Value: 1.6912 Ether，交易发送的Ether；
- Gas Price: 82 Gwei，Gas的价格；
- Gas Limit: 21,000，转账交易恰好消耗21000Gas，因此总是21000；
- Usage by Txn: 21,000 (100%)，消耗的Gas占比，这里恰好全部消耗完；
- Nonce: 0，发送方的nonce，0表示第1笔交易；
- Input Data: 0x，因为是转账交易，没有输入数据，因此为空。

## 合约交易

合约交易就是指一个外部账号调用某个合约的某个public函数。我们以交易0x8aff...8cd0为例，可以看到：

- From: 0x2329...BA3a，交易的发起方，该地址一定是外部账户；
- To: 0x7a25...488D，交易的接收方，这里地址是一个合约地址；
- Value: 4.5 Ether，即向合约发送4.5 Ether；
- Gas Limit: 152,533，这是交易发起前设定的最大Gas；
- Usage by Txn: 125,290 (82.14%)，这是交易实际消耗的Gas；
- Input Data: 0x7ff36ab5...，这是交易的输入数据，其中包含了调用哪个函数，以及传递的参数，解码后可知调用函数是 `swapExactETHForTokens`。

可见，转账交易的Gas费用是固定的，而合约交易只能预估，具体费用以实际执行后消耗的为准。

## 小结

以太坊的交易需要消耗Gas，而Gas价格和实际消耗的数量决定了一个交易实际消耗的Ether，即交易成本。

合约交易无法精确地确定Gas数量，只能预估并给出Gas Limit。

## 评论

# 智能合约

## 原文链接

以太坊相比比特币的一个重大创新就是它支持智能合约（Smart Contract）。

所谓智能合约，就是一种运行在区块链上的程序。和普通程序不同的是，智能合约要保证在区块链网络的每一个节点中运行的结果完全相同，这样才能使任何一个节点都可以验证挖矿产出节点生成的区块里，智能合约执行的结果对不对。

因此，以太坊提供了一个EVM（Ethereum Virtual Machine）虚拟机来执行智能合约的字节码，并且，和普通程序相比，为了消除程序运行的不确定性，智能合约有很多限制，例如，不支持浮点运算（因为浮点数有不同的表示方法，不同架构的CPU运行的浮点计算精度都不同），不支持随机数，不支持从外部读取输入等等。

类似于Java源码被编译为JVM可执行的字节码，我们也需要一种高级语言来编写智能合约，然后编译成EVM的字节码。最常用的开发智能合约的语言是以太坊专门为其定制的Solidity语言，后续我们会详细介绍Solidity的用法。

一个智能合约被编译后就是一段EVM字节码，将它部署在以太坊的区块链时，会根据部署者的地址和该地址的nonce分配一个合约地址，合约地址和账户地址的格式是没有区别的，但合约地址没有私钥，也就没有人能直接操作该地址的合约数据。要调用合约，唯一的方法是调用合约的公共函数。

这也是合约的一个限制：合约不能主动执行，它只能被外部账户发起调用。如果一个合约要定期执行，那只能由线下服务器定期发起合约调用。

此外，合约作为地址，可以接收Ether，也可以发送Ether。合约内部也可以存储数据。合约的数据存储在合约地址关联的存储上，这就使得合约具有了状态，可以实现比较复杂的逻辑，包括存款、取款等。

合约在执行的过程中，可以调用其他已部署的合约，前提是知道其他合约的地址和函数签名，这就大大扩展了合约的功能。例如，一个合约可以调用另一个借贷合约的借款方法，再调用交易合约，最后再调用还款方法，实现所谓的“闪电贷”（即在一个合约调用中实现借款-交易-还款）功能。多个合约的嵌套调用也使得因为代码编写的漏洞导致黑客攻击的可能性大大增加。为了避免漏洞，编写合约时需要更加小心。

## 小结

以太坊通过EVM虚拟机执行智能合约代码；

合约被部署后将自动获得一个地址，并可像外部账户一样存取Ether，还可以存储状态数据；

合约只能被动地被外部账户调用，但在执行时可以调用其他合约的公共函数。

[评论](#)

# 编写合约

## 原文链接

以太坊的智能合约就是一段由EVM虚拟机执行的字节码，类似于Java虚拟机执行Java字节码。直接编写字节码非常困难，通常都是由编译器负责把高级语言编译为字节码。

编写以太坊合约最常用的高级语言是Solidity，这是一种类似JavaScript语法的高级语言。通过Solidity的[官方网站](#)可以快速上手，我们不会详细讨论Solidity的语法细节，这里给出一个简单的投票合约：

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity =0.8.7;

contract Vote {

    event Voted(address indexed voter, uint8 proposal);

    mapping(address => bool) public voted;

    uint256 public endTime;

    uint256 public proposalA;
    uint256 public proposalB;
    uint256 public proposalC;

    constructor(uint256 _endTime) {
        endTime = _endTime;
    }

    function vote(uint8 _proposal) public {
        require(block.timestamp < endTime, "Vote expired.");
        require(_proposal >= 1 && _proposal <= 3, "Invalid proposal.");
        require(!voted[msg.sender], "Cannot vote again.");
        voted[msg.sender] = true;
        if (_proposal == 1) {
            proposalA ++;
        }
        else if (_proposal == 2) {
```

```
        proposalB ++;
    }
    else if (_proposal == 3) {
        proposalC ++;
    }
    emit Voted(msg.sender, _proposal);
}

function votes() public view returns (uint256) {
    return proposalA + proposalB + proposalC;
}
}
```

Solidity注释和JavaScript一致，第一行通常是版权声明的注释，然后声明编译器版本：

```
pragma solidity =0.8.7; // 指定编译器版本为0.8.7
```

也可以指定版本范围，如：

```
pragma solidity >=0.8.0 <0.9.0; // 指定编译器版本为0.8.x
```

紧接着，由关键字 `contract` 声明一个合约：

```
contract Vote {
    ... // 合约代码
}
```

虽然一个Solidity文件可以包含多个合约，但最好还是遵循一个文件一个合约，且文件名保持与合约一致。这里的合约名是 `Vote`。

熟悉面向对象编程的小伙伴对类、成员变量、成员方法一定不陌生。一个合约就相当于一个类，合约内部可以有成员变量：

```
contract Vote {
    // 记录已投票的地址：
    mapping(address => bool) public voted;

    // 记录投票终止时间：
    uint256 public endTime;
```



```
// 记录得票数量：
uint256 public proposalA;
uint256 public proposalB;
uint256 public proposalC;

...

}
```

Solidity支持整型（细分为 `uint256` 、 `uint128` 、 `uint8` 等）、`bytes32` 类型、映射类型（相当于Java的Map）、布尔型（`true` 或 `false`）和特殊的 `address` 类型表示一个以太坊地址。

以太坊合约 **不支持**浮点数类型，是为了保证每个节点运行合约都能得到完全相同的结果。浮点数运算在不同的ISA体系下存在表示方式、运算精度的不同，无法保证两个节点执行浮点运算会得到相同的结果。

所有的成员变量都默认初始化为 `0` 或 `false`（针对bool）或空（针对mapping）。

如果某个成员变量要指定初始值，那么需要在构造函数中赋值：

```
contract Vote {
    ...

    // 构造函数：
    constructor(uint256 _endTime) {
        endTime = _endTime; // 设定成员变量endTime为指定参数值
    }

    ...
}
```

以太坊合约支持读、写两种类型的成员函数，以 `view` 修饰的函数是只读函数，它不会修改成员变量，即不会改变合约的状态：

```
contract Vote {
    ...

    function votes() public view returns (uint256) {
        return proposalA + proposalB + proposalC;
    }
}
```

```
...  
}
```

没有 `view` 修饰的函数是写入函数，它会修改成员变量，即改变了合约的状态：

```
contract Vote {  
    ...  
  
    function vote(uint8 _proposal) public {  
        require(block.timestamp < endTime, "Vote expired.");  
        require(_proposal >= 1 && _proposal <= 3, "Invalid proposal.");  
        require(!voted[msg.sender], "Cannot vote again.");  
        // 给mapping增加一个key-value:  
        voted[msg.sender] = true;  
        if (_proposal == 1) {  
            // 修改proposalA:  
            proposalA ++;  
        }  
        else if (_proposal == 2) {  
            // 修改proposalB:  
            proposalB ++;  
        }  
        else if (_proposal == 3) {  
            // 修改proposalC:  
            proposalC ++;  
        }  
        emit Voted(msg.sender, _proposal);  
    }  
  
    ...  
}
```

合约可以定义事件（Event），我们在Vote合约中定义了一个 `Voted` 事件：

```
contract Vote {  
    // Voted事件，有两个相关值：  
    event Voted(address indexed voter, uint8 proposal);  
}
```

```
...  
}
```

只定义事件还不够，触发事件必须在合约的写函数中通过 `emit` 关键字实现。当调用 `vote()` 写方法时，会触发 `Voted` 事件：

```
contract Vote {  
    ...  
  
    function vote(uint8 _proposal) public {  
        ...  
        emit Voted(msg.sender, _proposal);  
    }  
  
    ...  
}
```

事件可用于通知外部感兴趣的第三方，他们可以在区块链上监听产生的事件，从而确认合约某些状态发生了改变。

以上就是用Solidity编写一个完整的合约所涉及的几个要素：

- 声明版权（可选）；
- 声明编译器版本；
- 以`contract`关键字编写一个合约；
- 可以包含若干成员变量；
- 可以在构造函数中对成员变量初始化（可选）；
- 可以编写只读方法；
- 可以编写写入方法；
- 可以声明Event并在写入方法中触发。

函数又可以用 `public` 或 `private` 修饰。顾名思义，`public` 函数可以被外部调用，而 `private` 函数不能被外部调用，他们只能被 `public` 函数在内部调用。

## 合约执行流程

当一个合约编写完成并成功编译后，我们就可以把它部署到以太坊上。合约部署后将自动获得一个地址，通过该地址即可访问合约。

把 `contract Vote {...}` 看作一个类，部署就相当于一个实例化。如果部署两次，将得到两个不同的地址，相当于实例化两次，两个部署后的合约对应的成员变量是完全独立的，互不影响。

构造函数在部署合约时就会立刻执行，且仅执行一次。合约部署后就无法调用构造函数。

任何外部账户都可以发起对合约的函数调用。如果调用只读方法，因为不改变合约状态，所以任何时刻都可以调用，且不需要签名，也不需要消耗Gas。但如果调用写入方法，就需要签名提交一个交易，并消耗一定的Gas。

在一个交易中，只能调用一个合约的一个写入方法。无需考虑并发和同步的问题，因为以太坊交易的写入是严格串行的。

## 验证

由于任何外部账户都可以发起对合约的函数调用，所以任何验证工作都必须在函数内部自行完成。最常用的 `require()` 可以断言一个条件，如果断言失败，将抛出错误并中断执行。

常用的检查包括几类：

参数检查：

```
// 参数必须为1,2,3:
require(_proposal >= 1 && _proposal <= 3, "Invalid proposal.");
```

条件检查：

```
// 当前区块时间必须小于设定的结束时间:
require(block.timestamp < endTime, "Vote expired.");
```

调用方检查：

```
// msg.sender表示调用方地址:
require(!voted[msg.sender], "Cannot vote again.");
```

以太坊合约具备类似数据库事务的特点，如果中途执行失败，则整个合约的状态保持不变，不存在修改某个成员变量后，后续断言失败导致部分修改生效的问题：

```
function increment() {
    // 假设a,b均为成员变量:
```

```
a++;  
emit AChanged(a);  
// 如果下面的验证失败, a不会被更新, 也没有AChanged事件发生:  
require(b < 10, 'b >= 10');  
b++;  
}
```

即合约如果执行失败, 其状态不会发生任何变化, 也不会有任何事件发生, 仅仅是调用方白白消耗了一定的Gas。

## 小结

编写一个以太坊合约相当于编写一个类, 一个合约可以包含多个成员变量和若干函数, 以及可选的构造函数;

部署一个合约相当于实例化, 部署时刻将执行构造函数;

任何外部账户均可发起对合约函数的调用, 但一个交易仅限一个函数调用;

所有检查都必须在合约的函数内部完成。

## 评论

# 部署合约

## 原文链接

当我们编写完Vote合约后，如何把它部署到以太坊的链上？

实际上，部署合约也是一个交易，需要一个外部账户，花费一定的Gas，就可以把合约部署到链上。

因此，我们首先需要有一个便于开发和测试的钱包，才能创建一个外部账户，并且账户上要有一定的Ether。

## MetaMask

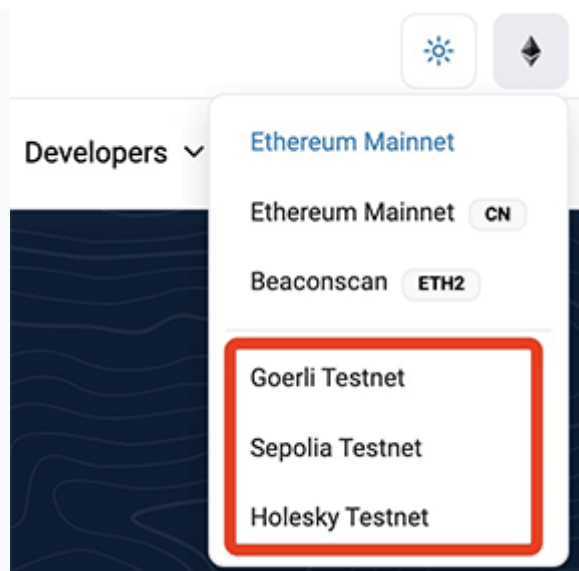
我们强烈推荐使用[MetaMask](#)钱包，这是一个基于浏览器插件的钱包，支持Chrome、Firefox等浏览器。使用的时候，通过Dapp网站的JavaScript可以发起交易，用户通过MetaMask确认后即可将交易发送至链上。

安装MetaMask非常简单，请参考官方文档。安装完成后，第一次启动MetaMask需要创建或导入一个钱包，设置一个解锁口令，MetaMask允许创建多个账号，可随时切换账号，还可切换不同的链，例如，以太坊主网、Ropsten测试网、Rinkeby测试网等。

在开发阶段，直接使用主网太费钱，可以使用测试网，并从[faucet.egorfine.com](#)或[faucet.dimensions.network](#)获取一些测试网的Ether。

### ⚠ 注意

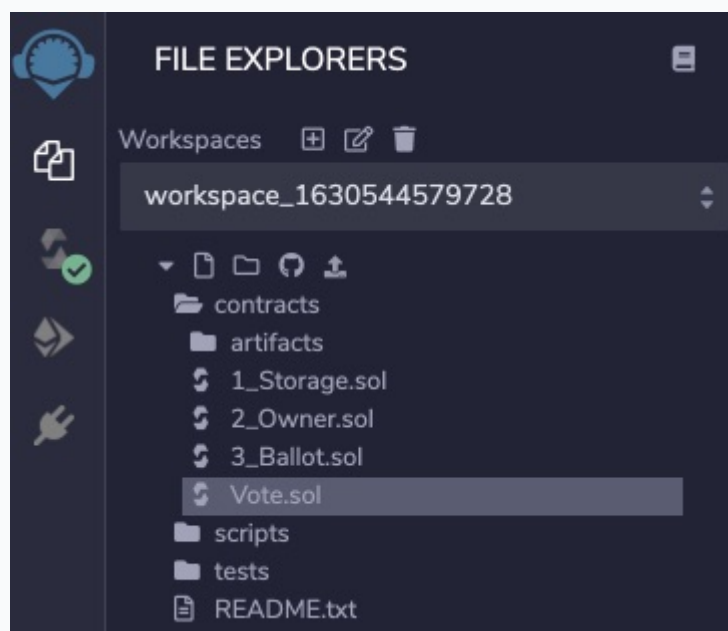
以太坊有多个测试网，开发前请在Etherscan确认使用哪个活动的测试网。



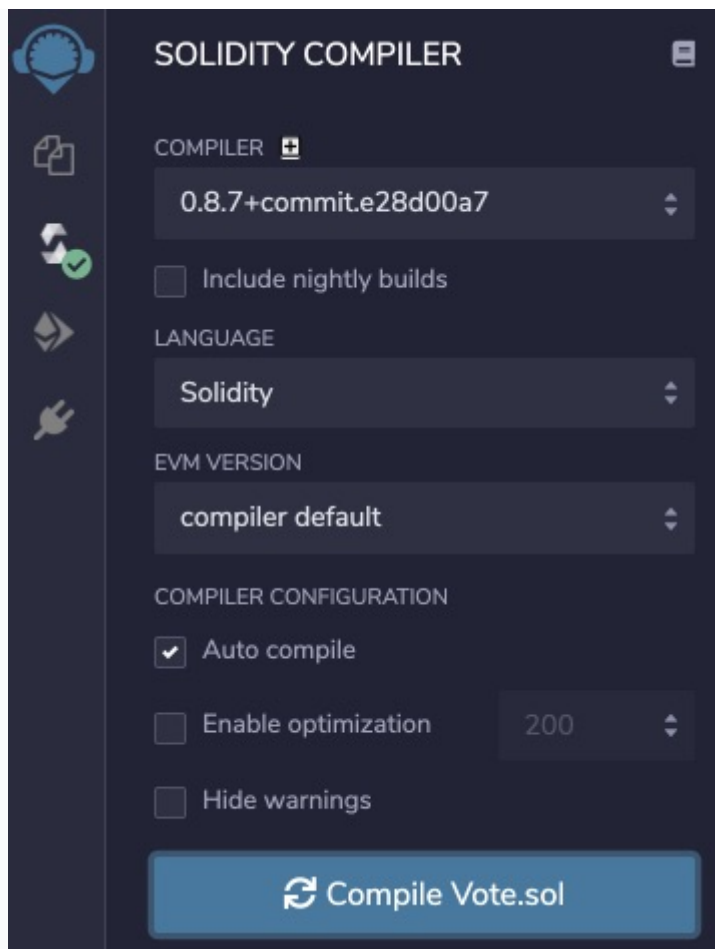
## 部署合约

以太坊官方提供了一个[Remix](#)的在线IDE，用于编写、编译和部署以太坊合约。这是从零开始部署一个合约的最简单的方式。

我们访问[Remix](#)（注意：要部署合约，只能通过http访问，不能使用https），在左侧选择“File explorers”，在默认的Workspace的 `contracts` 目录下新建文件 `Vote.sol`，然后贴入上一节我们编写的代码：

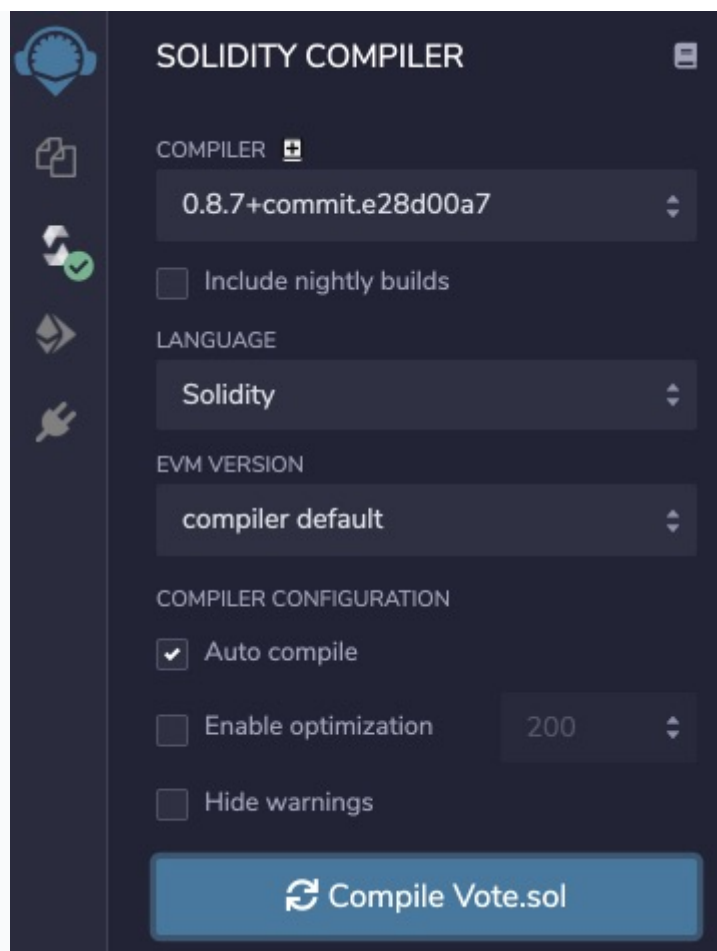


接下来是编译该合约。选择左侧的“Solidity compiler”，点击“Compile Vote.sol”开始编译：



如果没有编译错误，可以看到编译成功的标志。接下来选择“Deploy & run transactions”：





在“ENVIRONMENT”中，选择“Injected Web3”，表示我们要使用MetaMask注入的Web3环境，如果已正确连接MetaMask，可以看到“Ropsten (3) network”，表示已连接到Ropsten测试网。

在“CONTRACT”中，选择“Vote - contracts/Vote.sol”，这是我们将要部署的合约。

在“Deploy”按钮左侧，填入构造函数的参数，例如 `1735719000`，然后点击“Deploy”按钮开始部署，此时会弹出MetaMask的交易签名确认，确认后部署合约的交易即被发送至测试链。在MetaMask的账户 - 活动中可以看到正在发送的交易，查看详情可以在Etherscan查看[该交易的详细信息](#)。当交易被打包确认后，即可获得部署后合约的地址`0x5b2a...5a46`。

至此，我们就成功地部署了一个以太坊合约。

对于熟练的Solidity开发者，可以使用Truffle这个JavaScript工具通过JavaScript脚本全自动部署合约，减少手动操作导致的出错的可能。

## 小结

编写Solidity合约后，可以通过Remix在线编译、部署；

可以使用Truffle完成合约的自动化编译、测试、部署。

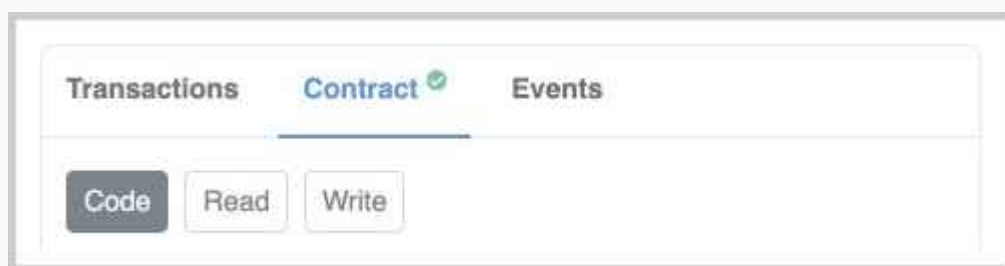
[评论](#)

# 调用合约

## 原文链接

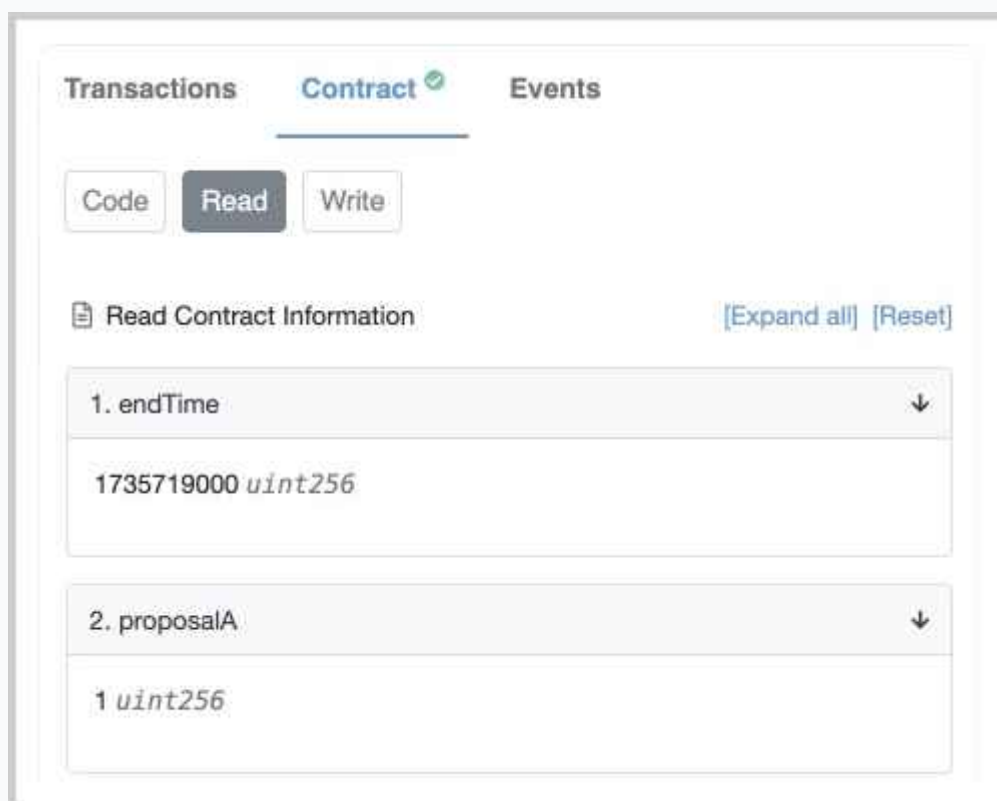
当我们已经成功地将一个合约部署到以太坊链上时，如何调用该合约？

首先，我们通过Etherscan这个网站可以查看已部署合约的详细信息，例如0x5b2a...5a46。在调用合约前，我们可以通过“Verify and Publish”这个链接将源码上传到Etherscan并验证。首先选择正确的Solidity编译器版本，贴入源码，Etherscan自动编译后，如果二进制的字节码完全匹配已部署的合约，则验证通过，该合约有一个绿色小勾的标志：



## 读取合约

访问合约的只读函数时，无需消耗Gas，也无需连接钱包，直接切换到“Read”面板，即可看到只读函数的返回值：



从结果可知, `endTime()` 返回 `1735719000` , `proposalA()` 返回 `1` 。

有的童鞋会问, 我们在Vote合约只有 `endTime` 字段并没有 `endTime()` 函数, 为什么可以访问 `endTime()` ? 原因是public字段会自动对应一个同名的只读函数, 即:

```
contract Vote {
    uint256 public endTime;
}
```

完全等价于:

```
contract Vote {
    uint256 private _endTime;

    function endTime() public view returns (uint256) {
        return _endTime;
    }
}
```

对于需要填入参数的只读函数, 可以直接在对应的输入框填入参数, 然后点击“Query”调用并获取结果:



5. voted

<input> (address)

0x9759bE9e1f8994432820739D7217D889918f2f07

Query

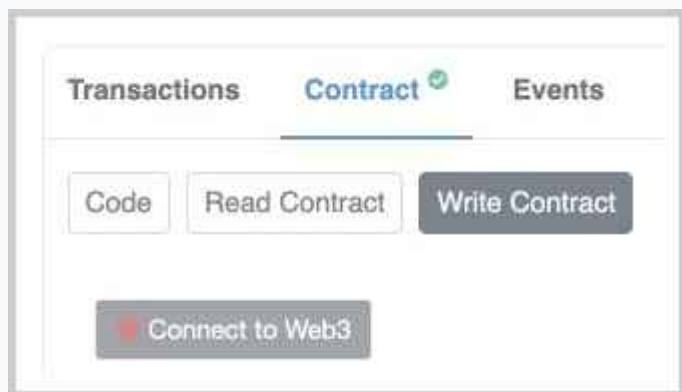
bool

[ voted(address) method Response ]

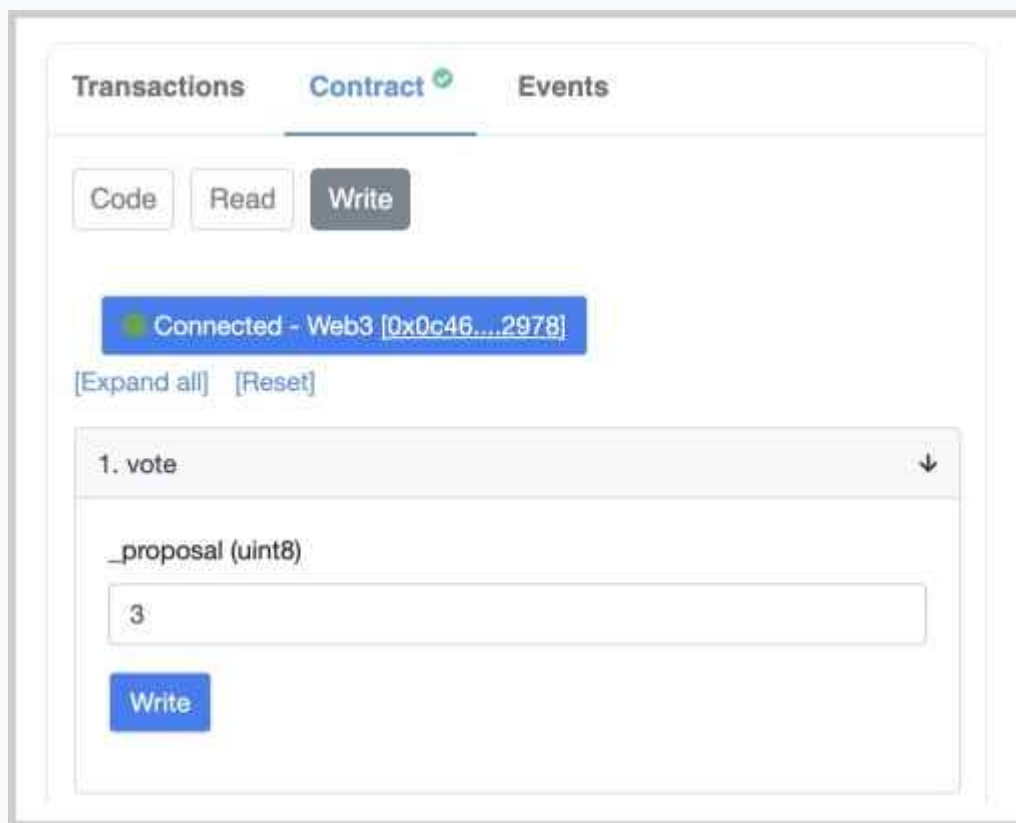
>> bool: true

## 写入合约

当我们要写入合约时，就必须提交一个签名的交易，并消耗一定的Gas。我们在Etherscan的合约页选择“Write”，会出现一个“Connect to Web3”的链接：



点击并连接MetaMask后，我们就可以选择一个写入函数，填入参数，然后点击“Write”：



在MetaMask中确认该交易后，交易被发送至链上。等待打包成功后，我们就可以读取到合约内部更新后的状态。

## 小结

调用合约的只读函数无需签名，也无需Gas，任何时候均可调用；

调用合约的写入函数需要签名发送交易，并消耗一定的Gas。只有等交易成功落块后，写入才算成功。

评论

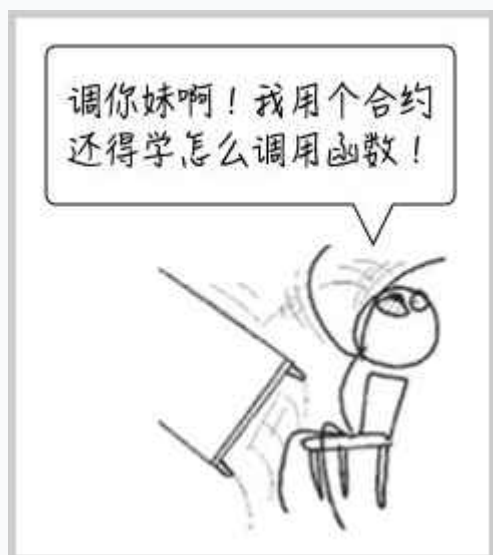
# 编写Dapp

[原文链接](#)

上一节我们讲了如何调用已部署在以太坊链上的合约。



通过Etherscan这个网站不仅可以查看合约代码，还可以调用合约的读取和写入方法。可见，调用一个合约是非常简单的。

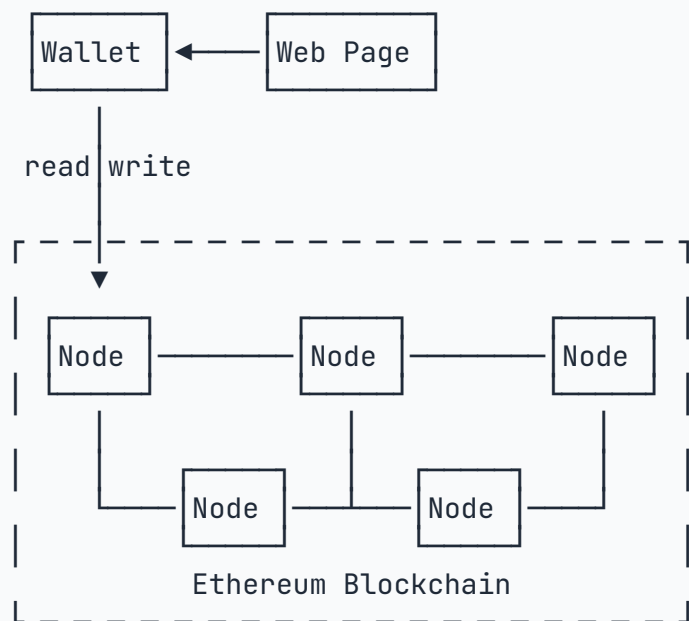


但是对于普通用户来说，这种调用方式就有点耍流氓了。大家平时上网发个微博，也没说要调用一个JSON REST API。如果发微博必须调用API，那微博用户肯定个个都是开发高手，根本没空撕来撕去。

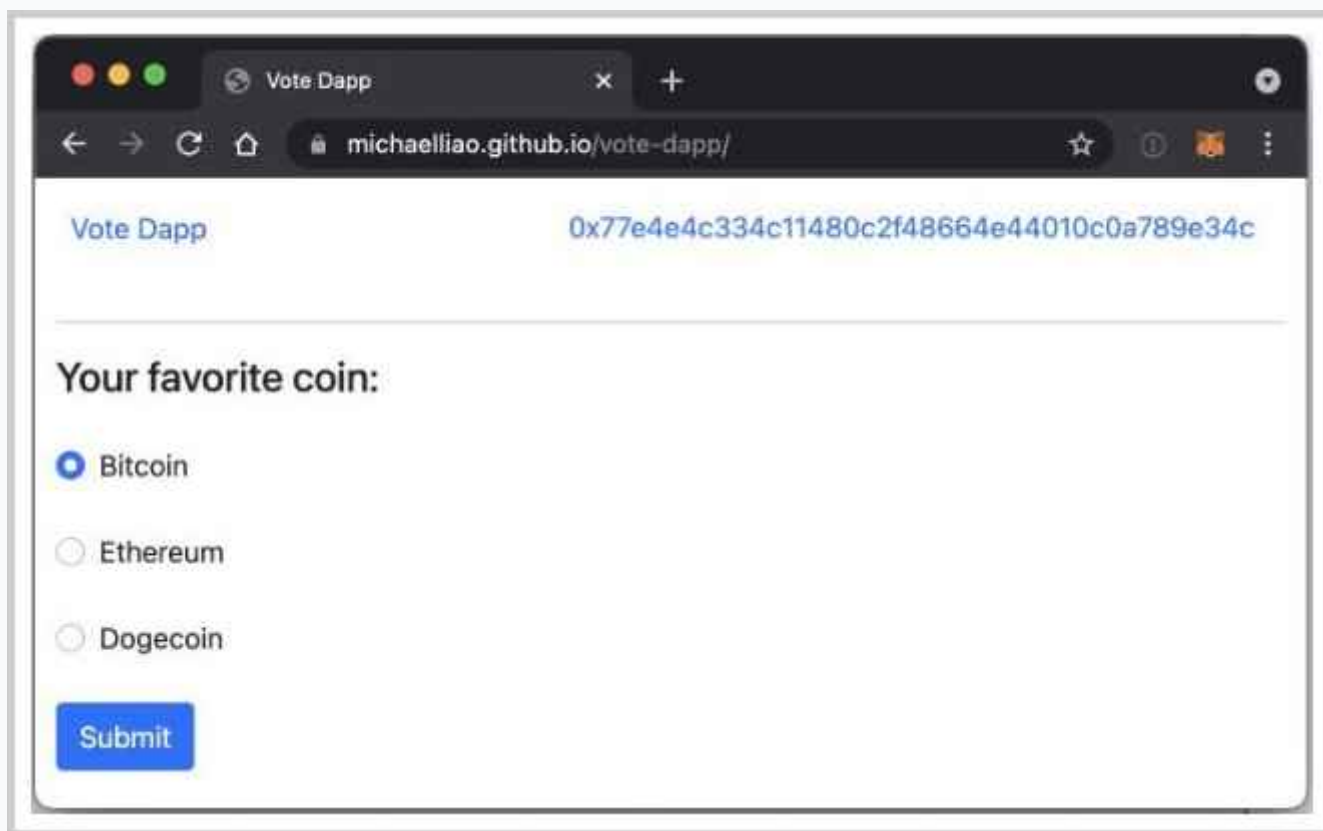
所以，要求普通用户自己去Etherscan调用合约不现实。本节我们就来介绍如何开发一个Dapp，让用户通过页面，点击按钮来调用合约。

## Dapp架构

一个Dapp的架构实际上包含以下部分：



以太坊的区块链网络实际上是一个由若干节点构成的P2P网络，所谓读写合约，实际上是向网络中的某个节点发送JSON-RPC请求。当我们想要做一个基于Vote合约的Dapp时，我们需要开发一个页面，并连接到浏览器的MetaMask钱包，这样，页面的JavaScript就可以通过MetaMask读写Vote合约，页面效果如下：



可以访问<https://michaelliao.github.io/vote-dapp/>查看页面并与合约交互。

编写Dapp的页面可以按以下步骤进行：



第一步，引入相关库，这里我们引入`ethers.js`这个库，它封装了读写合约的逻辑。在页面中用`<script>`引入如下：

```
<script src="https://cdn.jsdelivr.net/npm/ethers@5.0.32/dist/ethers.umd.min.js">
```

第二步，我们需要获取MetaMask注入的Web3，可以通过一个简单的函数实现：

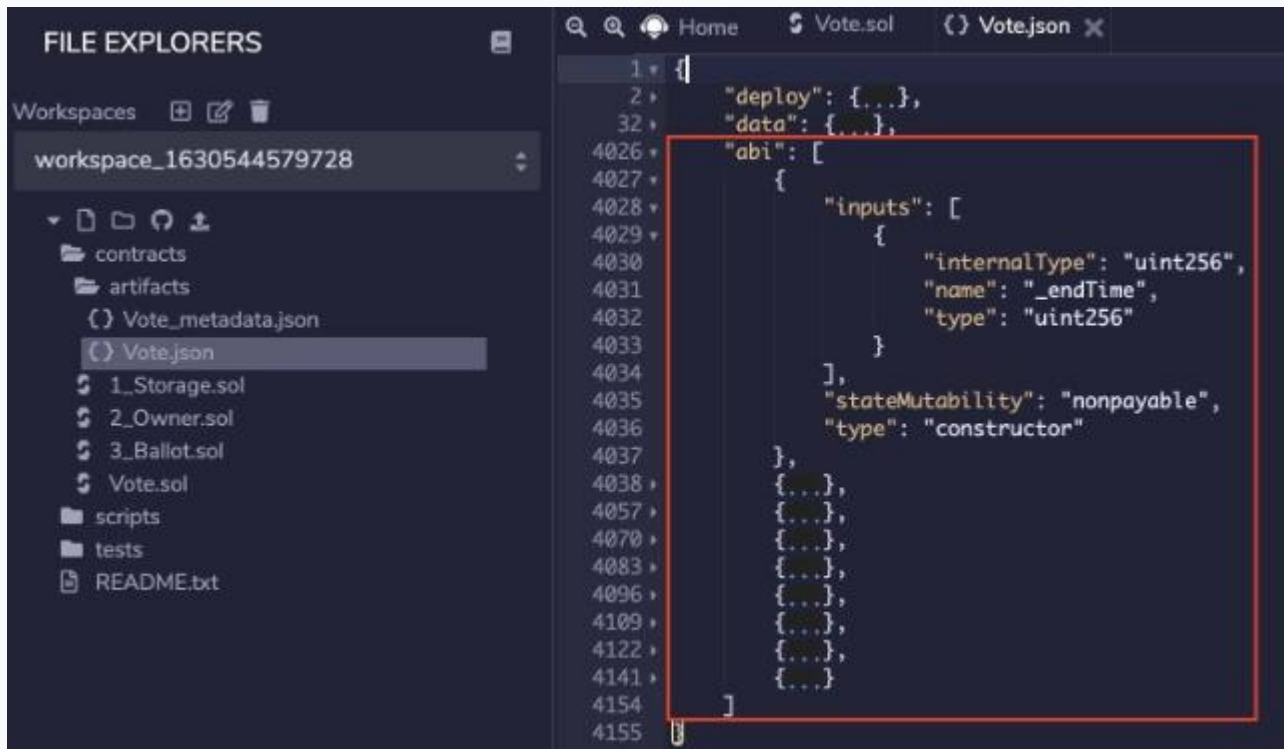
```
function getWeb3Provider() {  
  if (!window.web3Provider) {  
    if (!window.ethereum) {  
      console.error("there is no web3 provider.");  
      return null;  
    }  
    window.web3Provider = new  
ethers.providers.Web3Provider(window.ethereum, "any");  
  }  
  return window.web3Provider;  
}
```

第三步，在用户点击页面“Connect Wallet”按钮时，尝试连接MetaMask：

```
async function () {  
  if (window.getWeb3Provider() === null) {  
    console.error('there is no web3 provider.');    return false;  
  }  
  try {  
    // 获取当前连接的账户地址：  
    let account = await window.ethereum.request({  
      method: 'eth_requestAccounts',  
    });  
    // 获取当前连接的链ID：  
    let chainId = await window.ethereum.request({  
      method: 'eth_chainId'  
    });  
    console.log('wallet connected.');    return true;  
  } catch (e) {  
    console.error('could not get a wallet connection.', e);  
    return false;  
  }  
}
```

```
}  
}
```

最后一步，当我们已经连接到MetaMask钱包后，即可写入合约。写入合约需要合约的ABI (Application Binary Interface) 信息，即合约函数调用的接口信息，这些信息在Remix部署时产生。我们需要回到Remix，在 `contracts` - `artifacts` 目录下找到 `Vote.json` 文件，它是一个JSON，右侧找到 `"abi": [...]`，把 `abi` 对应的部分复制出来：



以常量的形式引入Vote合约的地址和ABI：

```
const VOTE_ADDR = '0x5b2a057e1db47463695b4629114cbdae99235a46';  
const VOTE_ABI = [{ "inputs": [{ "internalType": "uint256", "name":  
"_endTime", "type": "uint256" }], ...
```

现在，我们就可以在页面调用 `vote()` 写入函数了：

```
async function vote(proposal) {  
  // TODO: 检查MetaMask连接信息  
  // 根据地址和ABI创建一个Contract对象：  
  let contract = new ethers.Contract(VOTE_ADDR, VOTE_ABI,  
window.getWeb3Provider().getSigner());  
  // 调用vote()函数，并返回一个tx对象：  
  let tx = await contract.vote(proposal);  
  // 等待tx落块，并至少1个区块确认：
```

```
    await tx.wait(1);  
}
```

以上就是调用合约的全部流程。

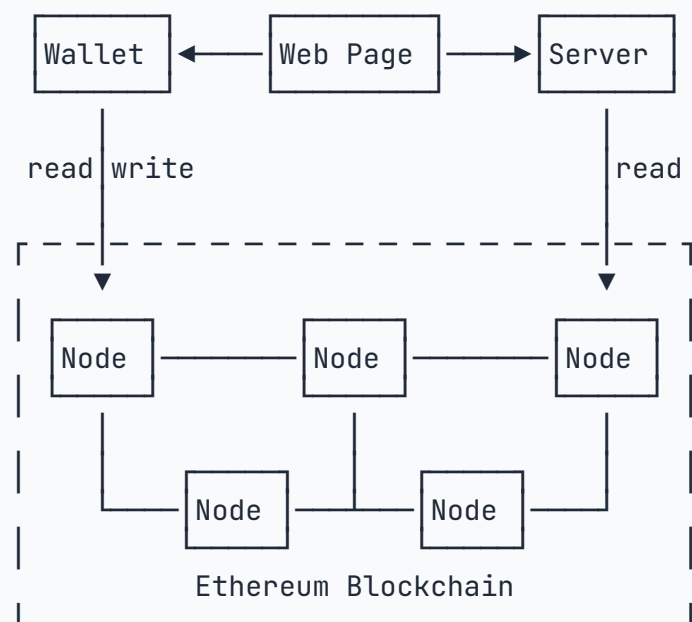
我们需要明确几个要点：

1. 页面的JavaScript代码无法直接访问以太坊网络的P2P节点，只能间接通过MetaMask钱包访问；
2. 钱包之所以能访问以太坊网络的节点，是因为它们内置了某些公共节点的域名信息；
3. 如果用户的浏览器没有安装MetaMask钱包，则页面无法通过钱包读取合约或写入合约。

因此引出了第二个问题：一个Dapp到底需不需要服务器端？

对于大多数的Dapp来说，是需要服务器端的，这是因为，当用户浏览器没有安装钱包，或者钱包并没有连接到Dapp期待的网络时，页面将无法获得合约的任何数据。例如，上述Dapp就无法读取到三种投票的数量，因此无法在页面上绘制对比图。

如果部署一个服务器端，由服务器连接P2P网络的节点并读取合约，然后以JSON API的形式给前端提供相关数据，则可以实现一个更完善的Dapp。因此，完整的Dapp架构如下：



为Dapp搭建后端服务器时要严格遵循以下规范：

1. 后端服务器只读取合约，不存储任何私钥，因此无法写入合约，保证了安全性；
2. 后端服务器要读取合约，就必须连接到P2P节点，要么选择公共的节点服务（例如Infura），要么自己搭建一个以太坊节点（维护的工作量较大）；

3. 后端服务器应该通过合约产生的日志（即合约运行时触发的event）监听合约的状态变化，而不是定期扫描。监听日志需要通过P2P节点创建Filter并获取Filter返回的日志；
4. 后端服务器应该将从日志获取的数据做聚合、缓存，以便前端页面能快速展示相关数据。

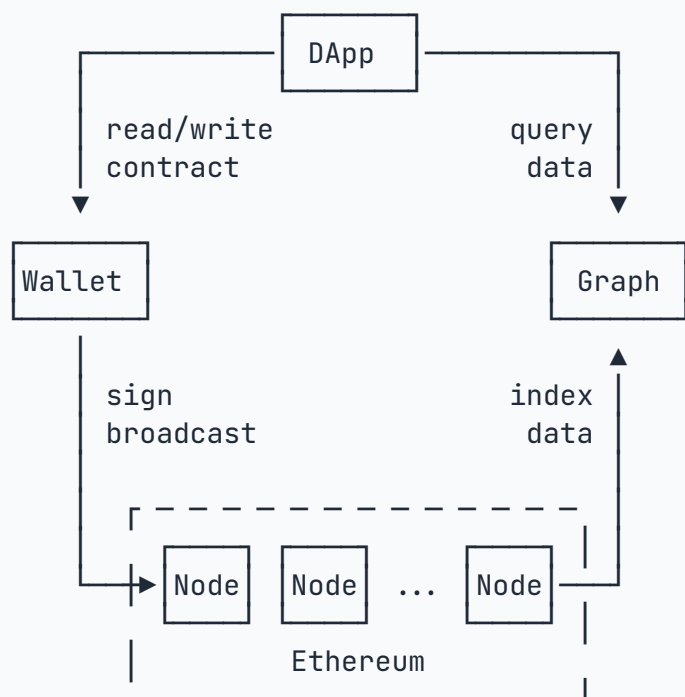
因此，设计Dapp时，既要考虑将关键业务逻辑写入合约，又要考虑日志输出有足够的信息让后端服务器能聚合历史数据。前端、后端和合约三方开发必须紧密配合。

不同的编程语言实现后端服务时，可以选择封装好的第三方库，例如Java可以使用[Web3j](#)，Python可以使用[Web3.py](#)，详细的说明可以参考[官方文档](#)。

## Graph

还有一种托管的后端服务：[The Graph](#)。它本身也可看作是一个基础设置。The Graph可以让我们部署一个Graph查询服务，如何定义表结构以及如何更新则由我们提供一个预编译的WASM。整个配置、WASM代码以及查询服务都托管在The Graph中，无需自己搭建服务器，非常方便。

因此，使用Graph的一个完整的DApp架构如下：



我们在Vote这个Dapp中并未提供后端服务，请自行实现。

## 小结

一个Dapp应用通常由前端、后端和部署的合约三部分构成。

不懂前端的后端不是一个合格的合约开发者。

[评论](#)

# 常用合约

## 原文链接

本节我们介绍以太坊上常见的几种合约：

- ERC-20：以太坊标准代币合约；
- Wrapped Ether：将以太坊封装为ERC20的合约；
- ERC-721：以太坊NFT标准合约；
- ERC-1155：相同NFT允许多个持有者的合约。

## 评论

# ERC-20

## 原文链接

ERC-20是以太坊定义的一个合约接口规范，符合该规范的合约被称为以太坊代币。

一个ERC-20合约通过 `mapping(address => uint256)` 存储一个地址对应的余额：

```
contract MyERC20 {  
    mapping(address => uint256) public balanceOf;  
}
```

如果要在两个地址间转账，实际上就是对 `balanceOf` 这个 `mapping` 的对应的kv进行加减操作：

```
contract MyERC20 {  
    mapping(address => uint256) public balanceOf;  
  
    function transfer(address recipient, uint256 amount) public returns  
(bool) {  
        // 不允许转账给0地址：  
        require(recipient != address(0), "ERC20: transfer to the zero  
address");  
        // sender的余额必须大于或等于转账额度：  
        require(balanceOf[msg.sender] >= amount, "ERC20: transfer amount  
exceeds balance");  
        // 更新sender转账后的额度：  
        balanceOf[msg.sender] -= amount;  
        // 更新recipient转账后的额度：  
        balanceOf[recipient] += amount;  
        // 写入日志：  
        emit Transfer(sender, recipient, amount);  
        return true;  
    }  
}
```

## 安全性

早期ERC20转账最容易出现的安全漏洞是加减导致的溢出，即两个超大数相加溢出，或者减法得到了负数导致结果错误。从Solidity 0.8版本开始，编译器默认就会检查运算溢出，因此，不要使用早期的Solidity编译即可避免溢出问题。

没有正确实现 `transfer()` 函数会导致交易成功，却没有任何转账发生，此时外部程序容易误认为已成功，导致假充值：

```
function transfer(address recipient, uint256 amount) public returns (bool)
{
    if (balanceOf[msg.sender] >= amount) {
        balanceOf[msg.sender] -= amount;
        balanceOf[recipient] += amount;
        emit Transfer(sender, recipient, amount);
        return true;
    } else {
        return false;
    }
}
```

实际上 `transfer()` 函数返回 `bool` 毫无意义，因为条件不满足必须抛出异常回滚交易，这是ERC20接口定义冗余导致部分开发者未能遵守规范导致的。

ERC-20另一个严重的安全性问题来源于重入攻击：

```
function transfer(address recipient, uint256 amount) public returns (bool)
{
    require(recipient != address(0), "ERC20: transfer to the zero address");
    uint256 senderBalance = balanceOf[msg.sender];
    require(senderBalance >= amount, "ERC20: transfer amount exceeds balance");
    // 此处调用另一个回调：
    callback(msg.sender);
    // 更新转账后的额度：
    balanceOf[msg.sender] = senderBalance - amount;
    balanceOf[recipient] += amount;
    emit Transfer(sender, recipient, amount);
    return true;
}
```



先回调再更新的方式会导致重入攻击，即如果 `callback()` 调用了外部合约，外部合约回调 `transfer()`，会导致重复转账。防止重入攻击的方法是一定要在校验通过后立刻更新数据，不要在校验-更新中插入任何可能执行外部代码的逻辑。

[评论](#)

# Wrapped Ether

## 原文链接

如果一个合约既支持ETH付款，也支持ERC-20代币付款，我们会发现这两种付款方式处理逻辑是不一样的：

```
contract Shop {
    function pay(uint productId) public payable {
        // pay ETH...
    }
    function pay(uint productId, address erc, uint256 amount) public {
        // pay ERC...
    }
}
```

两种逻辑混在一起用，代码就会复杂，就容易出问题。

一个简单的解决方法是将ETH也变成一种代币，可以用一个简单的 `WETH` 合约实现：

```
contract WETH {
    string public name      = "Wrapped Ether";
    string public symbol    = "WETH";
    uint8  public decimals = 18;

    mapping (address => uint) public  balanceOf;

    function deposit() public payable {
        balanceOf[msg.sender] += msg.value;
    }

    function withdraw(uint wad) public {
        require(balanceOf[msg.sender] >= wad);
        balanceOf[msg.sender] -= wad;
        msg.sender.transfer(wad);
    }

    ...
}
```

这样，处理ETH付款就可以简单地将它变成WETH，然后走同一种逻辑：

```
contract Shop {
    function pay(uint productId) public payable {
        // ETH -> WETH:
        WETH.deposit.value(msg.value)();
        _pay(productId, address(this), WETH.address, msg.value);
    }

    function pay(uint productId, address erc, uint256 amount) public {
        _pay(productId, msg.sender, erc, amount);
    }

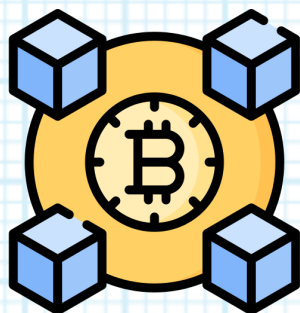
    function _pay(uint productId, address sender, address erc, uint256
amount) private {
        // pay ERC...
    }
}
```

不需要自己实现WETH，因为以太坊主网已经有一个通用的[WETH](#)。

## 小结

通过WETH，将ETH变成ERC20代币，可以简化处理代币的逻辑。

[评论](#)



# 区块链教程

零基础入门区块链，顺便学习写代码！

Author: 廖雪峰

Version: 2025-06-16

Website: <https://liaoxuefeng.com/books/blockchain/>