工作量证明

在上一节，我们构造了一个非常简单的数据结构 -- 区块，它也是整个区块链数据库的核心。目前所完成的区块链原型，已经可以通过链式关系把区块相互关联起来：每个块都与前一个块相关联。

但是，当前实现的区块链有一个巨大的缺陷：向链中加入区块太容易，也太廉价了。而区块链和比特币的其中一个核心就是，要想加入新的区块，必须先完成一些非常困难的工作。在本文，我们将会弥补这个缺陷。

工作量证明

区块链的一个关键点就是，一个人必须经过一系列困难的工作，才能将数据放入到区块链中。正是由于这种困难的工作，才保证了区块链的安全和一致。此外，完成这个工作的人，也会获得相应奖励（这也就是通过挖矿获得币）。

这个机制与生活现象非常类似：一个人必须通过努力工作，才能够获得回报或者奖励，用以支撑他们的生活。在区块链中，是通过网络中的参与者（矿工）不断的工作来支撑起了整个网络。矿工不断地向区块链中加入新块，然后获得相应的奖励。在这种机制的作用下，新生成的区块能够被安全地加入到区块链中，它维护了整个区块链数据库的稳定性。值得注意的是，完成了这个工作的人必须要证明这一点，即他必须要证明他的确完成了这些工作。

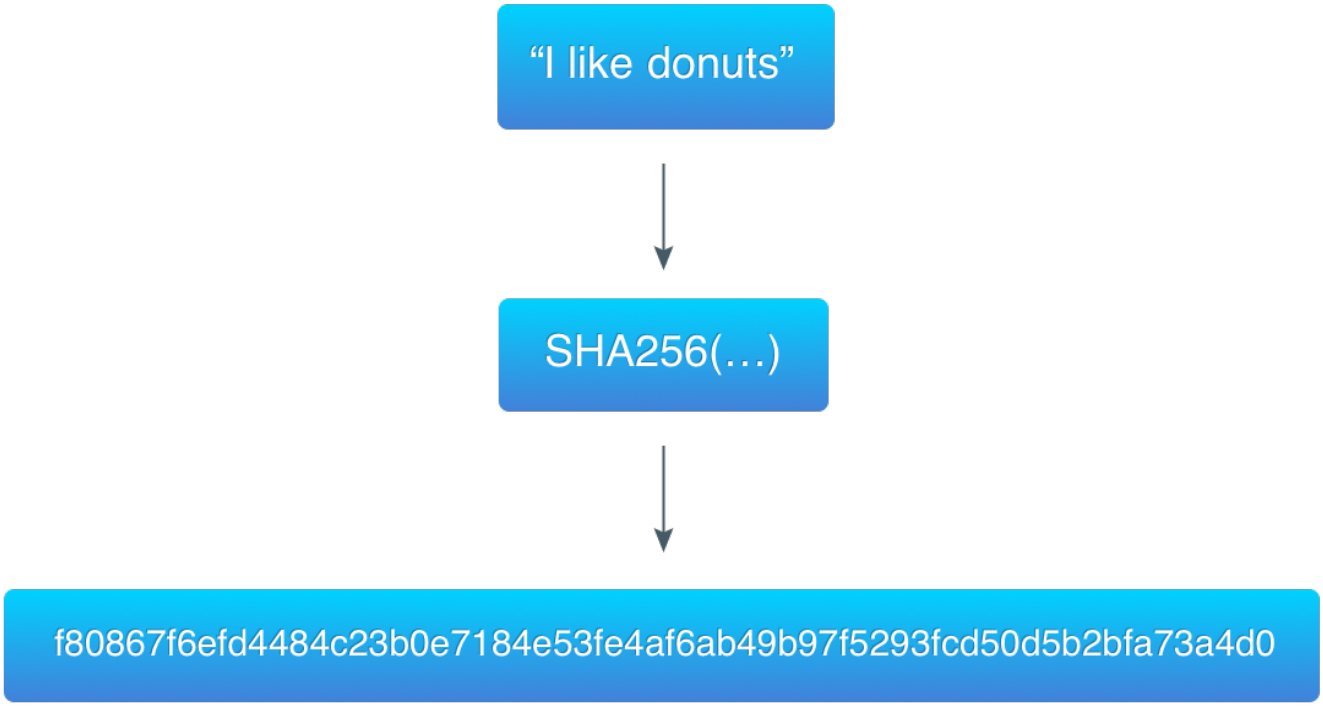
整个 “努力工作并进行证明” 的机制，就叫做工作量证明（proof-of-work）。要想完成工作非常地不容易，因为这需要大量的计算能力：即便是高性能计算机，也无法在短时间内快速完成。另外，这个工作的困难度会随着时间不断增长，以保持每 10 分钟出 1 个新块的速度。**在比特币中，这个工作就是找到一个块的哈希**，同时这个哈希满足了一些必要条件。这个哈希，也就充当了证明的角色。因此，寻求证明（寻找有效哈希），就是矿工实际要做的事情。

哈希计算

在本节，我们会讨论哈希计算。如果你已经熟悉了这个概念，可以直接跳过。

获得指定数据的一个哈希值的过程，就叫做哈希计算。一个哈希，就是对所计算数据的一个唯一表示。对于一个哈希函数，输入任意大小的数据，它会输出一个固定大小的哈希值。下面是哈希的几个关键特性：

1. 无法从一个哈希值恢复原始数据。也就是说，哈希并不是加密。
2. 对于特定的数据，只能有一个哈希，并且这个哈希是唯一的。
3. 即使是仅仅改变输入数据中的一个字节，也会导致输出一个完全不同的哈希。



哈希函数被广泛用于检测数据的一致性。软件提供者常常在除了提供软件包以外，还会发布校验和。当下载完一个文件以后，你可以用哈希函数对下载好的文件计算一个哈希，并与作者提供的哈希进行比较，以此来保证文件下载的完整性。

在区块链中，哈希被用于保证一个块的一致性。哈希算法的输入数据包含了前一个块的哈希，因此使得不太可能（或者，至少很困难）去修改链中的一个块：因为如果一个人想要修改前面一个块的哈希，那么他必须要重新计算这个块以及后面所有块的哈希。

Hashcash

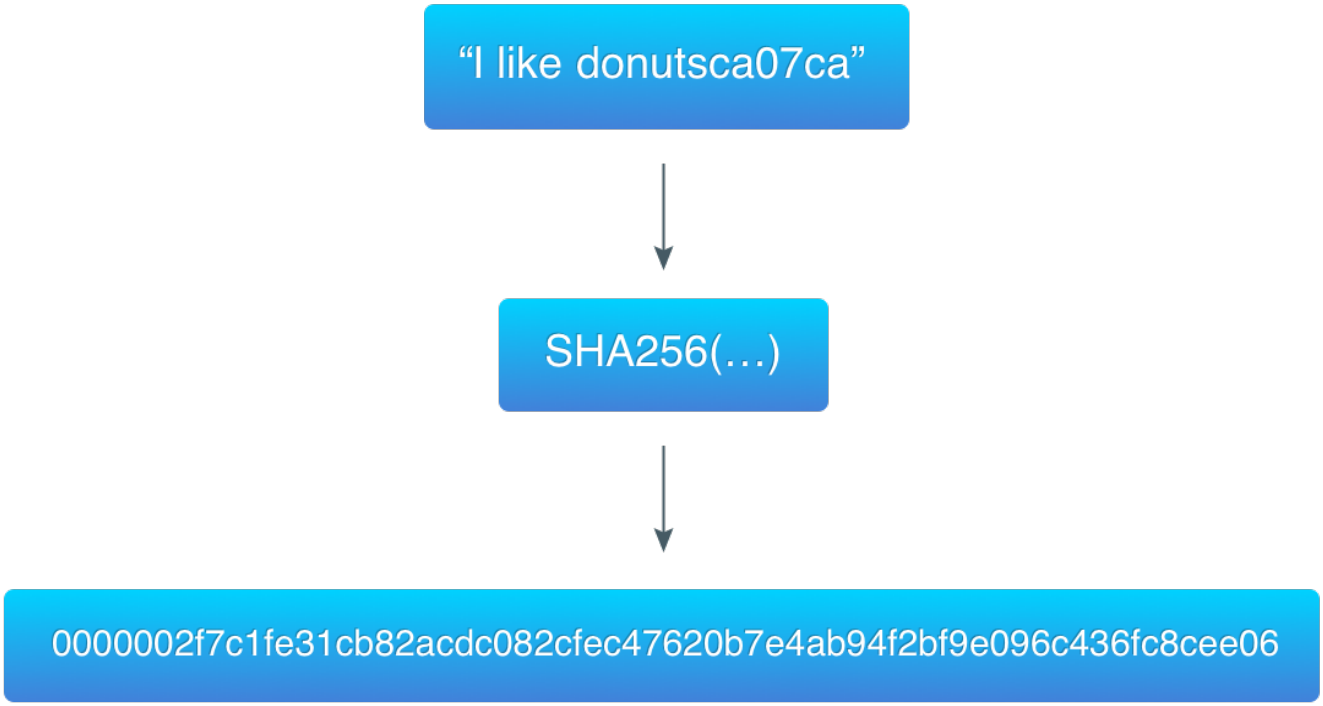
比特币使用 Hashcash ，一个最初用来防止垃圾邮件的工作量证明算法。它可以被分解为以下步骤：

1. 取一些公开的数据（比如，如果是 email 的话，它可以是接收者的邮件地址；在比特币中，它是区块头）
2. 给这个公开数据添加一个计数器。计数器默认从 0 开始
3. 将 **data(数据)** 和 **counter(计数器)** 组合到一起，获得一个哈希
4. 检查哈希是否符合一定的条件：
    1. 如果符合条件，结束
    2. 如果不符合，增加计数器，重复步骤 3-4

因此，这是一个暴力算法：改变计数器，计算新的哈希，检查，增加计数器，计算哈希，检查，如此往复。这也是为什么说它的计算成本很高，因为这一步需要如此反复不断地计算和检查。

现在，让我们来仔细看一下一个哈希要满足的必要条件。在原始的 Hashcash 实现中，它的要求是 “一个哈希的前 20 位必须是 0”。在比特币中，这个要求会随着时间而不断变化。因为按照设计，必须保证每 10 分钟生成一个块，而不论计算能力会随着时间增长，或者是会有越来越多的矿工进入网络，所以需要动态调整这个必要条件。

为了阐释这一算法，我从前一个例子（“I like donuts”）中取得数据，并且找到了一个前 3 个字节是全是 0 的哈希。



ca07ca 是计数器的 16 进制值，十进制的话是 13240266.

实现

好了，完成了理论层面，来动手写代码吧！首先，定义挖矿的难度值：

```
const targetBits = 24
```

在比特币中，当一个块被挖出来以后，“target bits” 代表了区块头里存储的难度，也就是开头有多少个 0。这里的 24 指的是算出来的哈希前 24 位必须是 0，如果用 16 进制表示，就是前 6 位必须是 0，这一点从最后的输出可以看出来。目前我们并不会实现一个动态调整目标的算法，所以将难度定义为一个全局的常量即可。

24 其实是一个可以任意取的数字，其目的只是为了有一个目标（target）而已，这个目标占据不到 256 位的内存空间。同时，我们想要有足够的差异性，但是又不至于大的过分，因为差异性越大，就越难找到一个合适的哈

希。

```
type ProofOfWork struct {
    block  *Block
    target *big.Int
}

func NewProofOfWork(b *Block) *ProofOfWork {
    target := big.NewInt(1)
    target.Lsh(target, uint(256-targetBits))

    pow := &ProofOfWork{b, target}

    return pow
}
```

这里，我们构造了 **ProofOfWork** 结构，里面存储了指向一个块(`block`)和一个目标(`target`)的指针。这里的 “目标”，也就是前一节中所描述的必要条件。这里使用了一个 大整数 ，我们会将哈希与目标进行比较：先把哈希转换成一个大整数，然后检测它是否小于目标。

在 **NewProofOfWork** 函数中，我们将 **big.Int** 初始化为 1，然后左移 `256 - targetBits` 位。**256** 是一个 SHA-256 哈希的位数，我们将要使用的是 SHA-256 哈希算法。**target（目标）** 的 16 进制形式为：

```
0x1000000000000000000000000000000000000000000000000000000000000000
```

它在内存上占据了 29 个字节。下面是与前面例子哈希的形式化比较：

```
0fac49161af82ed938add1d8725835cc123a1a87b1b196488360e58d4bfb51e3
0000010000000000000000000000000000000000000000000000000000000000
0000008b0f41ec78bab747864db66bcb9fb89920ee75f43fdaaeb5544f7f76ca
```

第一个哈希（基于 “I like donuts” 计算）比目标要大，因此它并不是一个有效的工作量证明。第二个哈希（基于 “I like donutsca07ca” 计算）比目标要小，所以是一个有效的证明。

译者注：上面的形式化比较有些“言不符实”，其实它应该并非由 “I like donuts” 而来，但是原文表达的意思是没问题的，可能是疏忽而已。下面是我做的一个小实验：

```
package main

import (
    "crypto/sha256"
    "fmt"
    "math/big"
)

func main() {

    data1 := []byte("I like donuts")
    data2 := []byte("I like donutsca07ca")
    targetBits := 24
    target := big.NewInt(1)
    target.Lsh(target, uint(256-targetBits))
    fmt.Printf("%x\n", sha256.Sum256(data1))
    fmt.Printf("%64x\n", target)
    fmt.Printf("%x\n", sha256.Sum256(data2))

}
```

输出：

```
xlc@xlc-mbp:~/.../go/word >>> make
go build -o blockchain
./blockchain
f80867f6efd4484c23b0e7184e53fe4af6ab49b97f5293fcd50d5b2bfa73a4d0
     1000000000000000000000000000000000000000000000000000000000
0000002f7c1fe31cb82acdc082cfec47620b7e4ab94f2bf9e096c436fc8cee06
```

你可以把目标想象为一个范围的上界：如果一个数（由哈希转换而来）比上界要小，那么是有效的，反之无效。因为要求比上界要小，所以会导致有效数字并不会很多。因此，也就需要通过一些困难的工作（一系列反复地计算），才能找到一个有效的数字。

现在，我们需要有数据来进行哈希，准备数据：

```
func (pow *ProofOfWork) prepareData(nonce int) []byte {
    data := bytes.Join(
        [][]byte{
            pow.block.PrevBlockHash,
            pow.block.Data,
            IntToHex(pow.block.Timestamp),
            IntToHex(int64(targetBits)),
            IntToHex(int64(nonce)),
        },
        []byte{},
    )

    return data
}
```

这个部分比较直观：只需要将 target ， nonce 与 Block 进行合并。这里的 `nonce` ，就是上面 Hashcash 所提到的计数器，它是一个密码学术语。

很好，到这里，所有的准备工作就完成了，下面来实现 PoW 算法的核心：

```
func (pow *ProofOfWork) Run() (int, []byte) {
    var hashInt big.Int
    var hash [32]byte
    nonce := 0

    fmt.Printf("Mining the block containing \"%s\"\n", pow.block.Data)
    for nonce < maxNonce {
        data := pow.prepareData(nonce)
        hash = sha256.Sum256(data)
        hashInt.SetBytes(hash[:])

        if hashInt.Cmp(pow.target) == -1 {
            fmt.Printf("\r%x", hash)
            break
        } else {
            nonce++
        }
    }
    fmt.Print("\n\n")

    return nonce, hash[:]
}
```

首先我们对变量进行初始化：

- `HashInt` 是 `hash` 的整形表示；
- `nonce` 是计数器。

然后开始一个 “无限” 循环： `maxNonce` 对这个循环进行了限制, 它等于 `math.MaxInt64` ，这是为了避免 `nonce` 可能出现的溢出。尽管我们 PoW 的难度很小，以至于计数器其实不太可能会溢出，但最好还是以防万一检查一下。

在这个循环中，我们做的事情有：

1. 准备数据
2. 用 SHA-256 对数据进行哈希
3. 将哈希转换成一个大整数
4. 将这个大整数与目标进行比较

跟之前所讲的一样简单。现在我们可以移除 `Block` 的 `SetHash` 方法，然后修改 `NewBlock` 函数：

```
func NewBlock(data string, prevBlockHash []byte) *Block {
    block := &Block{time.Now().Unix(), []byte(data), prevBlockHash, []byte{}, 0}
    pow := NewProofOfWork(block)
    nonce, hash := pow.Run()

    block.Hash = hash[:]
    block.Nonce = nonce

    return block
}
```

在这里，你可以看到 `nonce` 被保存为 `Block` 的一个属性。这是十分有必要的，因为待会儿我们对这个工作量进行验证时会用到 `nonce` 。 `Block` 结构现在看起来像是这样：

```
type Block struct {
    Timestamp     int64
    Data          []byte
    PrevBlockHash []byte
    Hash          []byte
    Nonce         int
}
```

好了！现在让我们来运行一下是否正常工作：

```
Mining the block containing "Genesis Block"
00000041662c5fc2883535dc19ba8a33ac993b535da9899e593ff98e1eda56a1

Mining the block containing "Send 1 BTC to Ivan"
00000077a856e697c69833d9effb6bdad54c730a98d674f73c0b30020cc82804

Mining the block containing "Send 2 more BTC to Ivan"
000000b33185e927c9a989cc7d5aaaed739c56dad9fd9361dea558b9bfaf5fbe

Prev. hash:
Data: Genesis Block
Hash: 00000041662c5fc2883535dc19ba8a33ac993b535da9899e593ff98e1eda56a1

Prev. hash: 00000041662c5fc2883535dc19ba8a33ac993b535da9899e593ff98e1eda56a1
Data: Send 1 BTC to Ivan
Hash: 00000077a856e697c69833d9effb6bdad54c730a98d674f73c0b30020cc82804

Prev. hash: 00000077a856e697c69833d9effb6bdad54c730a98d674f73c0b30020cc82804
Data: Send 2 more BTC to Ivan
Hash: 000000b33185e927c9a989cc7d5aaaed739c56dad9fd9361dea558b9bfaf5fbe
```

成功了！你可以看到每个哈希都是 3 个字节的 0 开始，并且获得这些哈希需要花费一些时间。

还剩下一件事情需要做，对工作量证明进行验证：

```
func (pow *ProofOfWork) Validate() bool {
    var hashInt big.Int

    data := pow.prepareData(pow.block.Nonce)
    hash := sha256.Sum256(data)
    hashInt.SetBytes(hash[:])

    isValid := hashInt.Cmp(pow.target) == -1

    return isValid
}
```

这里，就是我们就用到了上面保存的 `nonce` 。

再来检测一次是否正常工作：

```
func main() {
    ...

    for _, block := range bc.blocks {
        ...
        pow := NewProofOfWork(block)
        fmt.Printf("PoW: %s\n", strconv.FormatBool(pow.Validate()))
        fmt.Println()
    }
}
```

输出：

```
...

Prev. hash:
Data: Genesis Block
Hash: 00000093253acb814afb942e652a84a8f245069a67b5eaa709df8ac612075038
PoW: true

Prev. hash: 00000093253acb814afb942e652a84a8f245069a67b5eaa709df8ac612075038
Data: Send 1 BTC to Ivan
Hash: 0000003eeb3743ee42020e4a15262fd110a72823d804ce8e49643b5fd9d1062b
PoW: true

Prev. hash: 0000003eeb3743ee42020e4a15262fd110a72823d804ce8e49643b5fd9d1062b
Data: Send 2 more BTC to Ivan
Hash: 000000e42afddf57a3daa11b43b2e0923f23e894f96d1f24bfd9b8d2d494c57a
PoW: true
```

从下图可以看出，这次我们产生三个块花费了一分多钟，比没有工作量证明之前慢了很多（也就是成本高了很多）：

```
xlc@xlc-mbp:~/.../go/blockchain >>> time ./blockchain
Mining the block containing "Genesis Block"
000000e278c44cf308a704a0e1484d0fe70449cac49af5d1bf7844244863e641

Mining the block containing "Send 1 BTC to Ivan"
0000002c792d0932db9099da1e54672d291dd602cd390b1c0fc4b424f2839144

Mining the block containing "Send 2 more BTC to Ivan"
00000078ee359df77069c8d4c51058cf9d5833aad66ae51664944e58c6c48242

Prev hash:
Data: Genesis Block
Hash: 000000e278c44cf308a704a0e1484d0fe70449cac49af5d1bf7844244863e641
PoW: true

Prev hash: 000000e278c44cf308a704a0e1484d0fe70449cac49af5d1bf7844244863e641
Data: Send 1 BTC to Ivan
Hash: 0000002c792d0932db9099da1e54672d291dd602cd390b1c0fc4b424f2839144
PoW: true

Prev hash: 0000002c792d0932db9099da1e54672d291dd602cd390b1c0fc4b424f2839144
Data: Send 2 more BTC to Ivan
Hash: 00000078ee359df77069c8d4c51058cf9d5833aad66ae51664944e58c6c48242
PoW: true


real    1m20.614s
user    1m20.979s
sys     0m1.158s
```

总结

我们离真正的区块链又进了一步：现在需要经过一些困难的工作才能加入新的块，因此挖矿就有可能了。但是，它仍然缺少一些至关重要的特性：区块链数据库并不是持久化的，没有钱包，地址，交易，也没有共识机制。不过，所有的这些，我们都会在接下来的文章中实现，现在，愉快地挖矿吧！

参考：

- Full source codes
- Blockchain hashing algorithm

- Proof of work
- Hashcash
- Building Blockchain in Go. Part 2: Proof-of-Work
- part_2

- 上一节: 基本原型
- 下一节: 持久化