用Go构建区块链——2.工作量证明

本篇是"用Go构建区块链"系列的第二篇，主要对原文进行翻译。对应原文如下：

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

1、介绍

在上一篇文章，我们构建了一个非常简单的数据结构，这是区块链数据库的本质。我们可以通过它们之间的链式关系为它添加区块：每个区块和前一个区块相链接。哎，我们实现的区块链有一个严重的缺陷：添加区块到链上非常简单而且成本很低。区块链和比特币的关键之一在于添加新的区块是一项非常困难的工作。今天，我们要解决这个缺陷。

2、工作量证明

区块链的一个宗旨是，人们必须执行一些困难的工作才能将数据放入到区块链中。正是这项困难的工作才使得区块链更加安全和一致。此外，这项困难的工作有相应的奖励（这是人们如何通过挖矿获得币）。

这种机制与现实生活中的机制非常相似：必须努力工作以获得报酬并维持生活。在区块链中，网络的一些参与者（矿工）负责维护网络，为其添加新的区块，并为这份工作获得相应的奖励。他们工作的结果是，一个区块以一种安全的方式加入到区块链中，从而保持整个区块链数据库的稳定性。值得注意的是，完成这项工作的人必须证明这一点。

这整个“努力工作和证明”的机制被称为工作量证明。这很难，因为它需要大量的计算能力：即使是高性能的计算机也无法快速完成。此外，这项工作的难度不断增加，以保持新的区块在每小时6个区块的速率。在比特币中，这种工作的目标是为一个区块找到一个符合要求的哈希。这就是这个哈希作为证明。因此，找到一个证明就是实际的工作。

最后要注意的是。工作量证明算法必须满足这样一项要求：完成工作很难，但验证很容易。证明通常交给其他人，所以对他们来说，不需要太多时间来验证它。

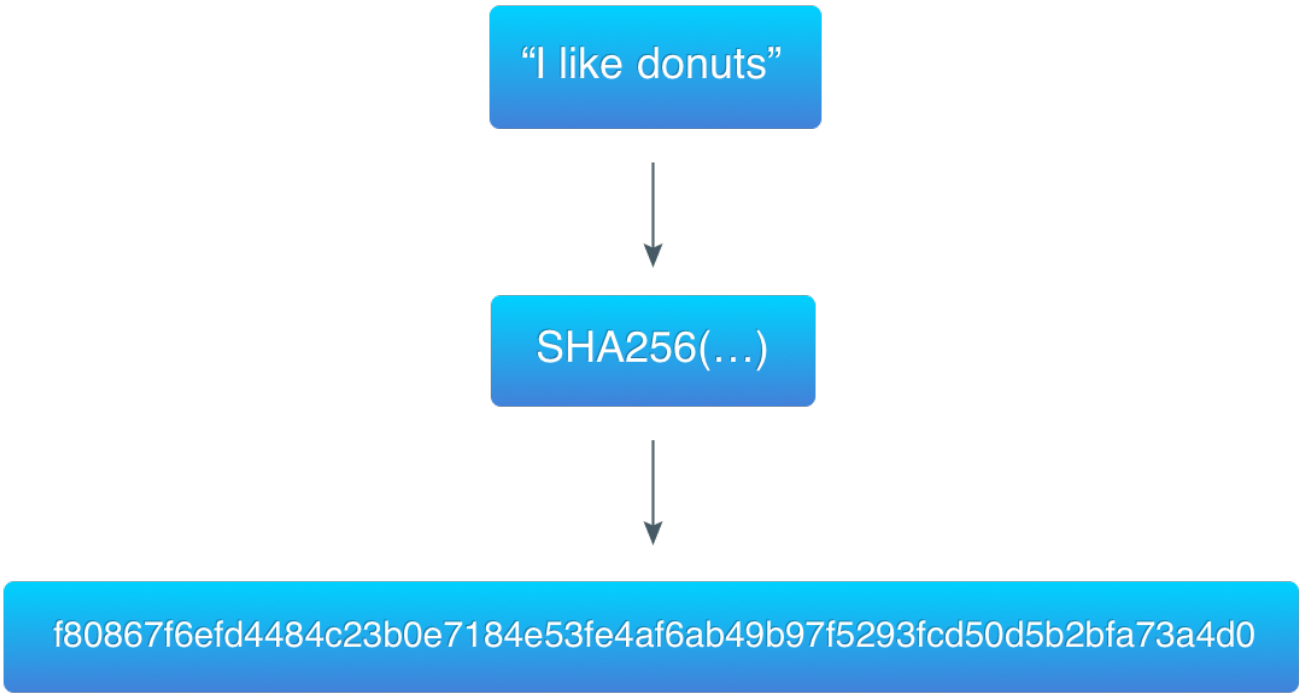
3、哈希

在这一段中，我们将讨论哈希。如果您熟悉这个概念，您可以跳过这个部分。

哈希计算是指一个获得指定数据的哈希值的过程。哈希值，是它所计算数据的唯一表示。哈希函数是一种可以获取任意大小数据并生成固定长度哈希值的函数。以下是哈希的一些关键特性：

1.原始数据无法从哈希值中恢复。因此，哈希不是加密。
2.特定的数据只能有一个哈希值，而且哈希值是唯一的。

3.更改输入数据中的一个字节将导致完全不同的哈希值。



哈希函数广泛用于检查数据的一致性。除软件包外，某些软件提供商还发布校验和。下载文件后，您可以将其提供给哈希函数，并将生成的哈希值与软件开发人员提供的哈希值进行比较。

4、Hashcash

比特币使用 Hashcash ，这是一种最初用于防止电子邮件垃圾邮件的工作量证明算法。它可以分成以下几个步骤：

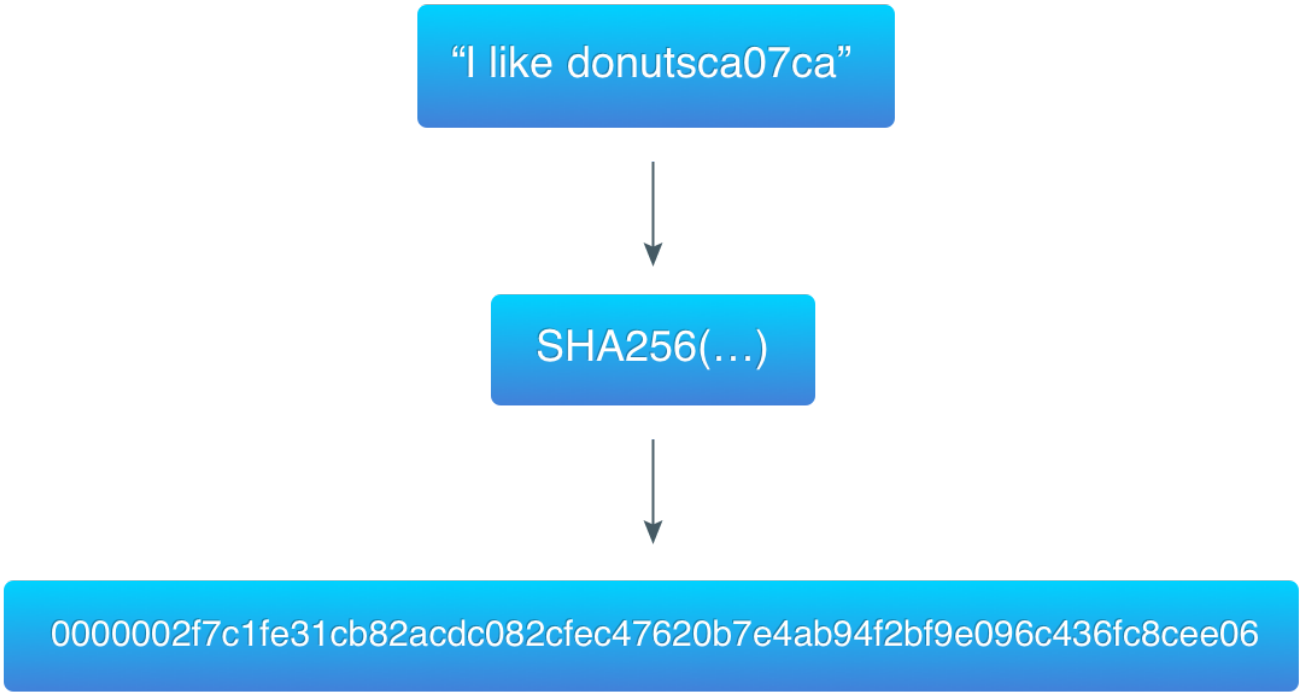
1.拿一些公开的数据（如电子邮件，它是接收者的电子邮件地址;比特币的情况下，它是区块头）。
2.给它添加一个计数器。计数器从0开始。
3.获取 `data + counter` 组合的哈希。
4.检查哈希是否符合某些要求。

1. 如果符合，则完成
2. 如果不符合，增加计数器，重复步骤3和4

因此，这是一个暴力算法：改变计数器，计算一个新的哈希值，检查它，增加计数器，计算哈希值等。这就是为什么它的计算成本很高。

现在我们来看看哈希必须满足的要求。在最初的Hashcash实现中，需求看起来是“哈希值的前20位必须为零”。在比特币中，需求会随时调整，因为在设计上，尽管计算能力随着时间的推移而增加，越来越多的矿工加入到网络中，但必须每隔10分钟生成一个区块。

为了演示这种算法，我从前面的例子（"I like donuts"）中获取数据，并找到一个前3个字节为0的哈希。



`ca07ca` 是计数器的十六进制值，即十进制系统中的13240266。

5、实现

好了，我们已经完成了理论，让我们编写代码！首先，我们来定义挖矿的难度：

```
const targetBits = 24
```

在比特币中，"target bits"是在每个被挖出来的区块头上存储的困难度。目前我们不会实现目标调整算法，因此我们可以将难度定义为全局常量。

24是一个任意数字，我们的目标是在内存中占用少于256位的目标。我们希望差异足够大，但不要太大，因为差异越大，找到合适的哈希越困难。

```
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
}
```

这里创建 `ProofOfWork` 一个保存指向区块的指针和指向目标的指针的结构。"target"是前一段描述的要求的另一个名称。我们使用一个大整数，因为我们将哈希与目标进行比较：我们将哈希转换为大整数，并检查它是否小于目标。

在 `NewProofOfWork` 函数中，我们 `big.Int` 用值1初始化a，并将它左移位 `256 - targetBits` 。 `256` 是以位为单位的SHA-256哈希长度，而且是我们要使用的SHA-256哈希算法。 `targetis` 的十六进制表示是：

```
0x1000000000000000000000000000000000000000000000000000000000000
```

它在内存中占用29个字节。下面是它与前面例子中的哈希值的x形式化比较：

```
0fac49161af82ed938add1d8725835cc123a1a87b1b196488360e58d4bfb51e3
0000010000000000000000000000000000000000000000000000000000000000
0000008b0f41ec78bab747864db66bcb9fb89920ee75f43fdaaeb5544f7f76ca
```

第一个哈希（根据"I like donuts"计算）大于目标，因此它不是有效的工作证明。第二个哈希（根据"I like donutsca07ca"计算）小于目标，因此这是一个有效的证明。

您可以将目标视为范围的上限：如果数字（哈希）低于边界，则该数字有效，反之亦然。降低边界将导致更少的有效数字，因此，找到一个有效数字需要更困难的工作。

现在，我们需要数据来哈希。让我们来准备一下：

```
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
}
```

这件事很简单：我们只是将区块字段与目标和随机数nonce合并。 这里的 `nonce` 是来自上面Hashcash描述的计数器，这是一个密码学术语。

好了，所有的准备工作都完成了，我们来实现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)
        fmt.Printf("\r%x", hash)
        hashInt.SetBytes(hash[:])

        if hashInt.Cmp(pow.target) == -1 {
            break
        } else {
            nonce++
        }
    }
    fmt.Print("\n\n")

    return nonce, hash[:]
}
```

首先，我们初始化变量：`hashInt` 是 `hash` 的整型表示; `nonce` 是计数器。接下来，我们运行一个"无限"循环：它受限于 `maxNonce` , 等于 `math.MaxInt64` ;这是为了避免 `nonce` 可能出现的溢出。尽管我们的PoW实现的难度太低而不足以使计数器溢出，但为了以防万一，进行此项检查仍然更好。

在循环中我们：

1.准备数据。
2.用SHA-256计算哈希值。
3.将哈希转换为一个大整数。
4.将这个大整数与目标进行比较。

和前面解释的一样简单。现在我们可以删掉 `Block` 里的 `SetHash` 函数，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， 您可以看到 `nonce` 被保存为 `Block` 的一个属性。这是很有必要的，因为需要 `nonce` 去验证证明。现在 `Block` 看起来像是这样：

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

好的！让我们运行该程序，看看是否一切正常。

```
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
```

欧耶！您可以看到每个哈希都是以3个字节的0开头，并且获得这些哈希需要花费一些时间。

还有一件事要做：让我们可以验证工作量的证明。

```
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
}
```

这就是我们需要保存的随机数的地方。

让我们再次检查一切是否正常：

```
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
```

6、结论

我们的区块链离真实的架构更近了一步：现在添加区块需要困难的工作，这就让挖坑成为了可能。但它仍然缺乏一些关键特征：区块链数据库不是持久的，没有钱包，地址，交易，并且没有共识机制。所有这些我们将在未来的文章中实现，而现在，快乐的挖掘！

链接：

1.完整的源代码

2.区块链哈希算法

3.Proof of work(工作量证明)

4.Hashcash