引入：

上一篇文章中，虽然我们只设计了一个简单的区块链原型，但它体现了区块链数据库的本质。我们可以向它里面添加区块，这些区块呈链状的，后一个区块指向前一个区块。然而，我们实现的区块链有一个非常明显的瑕疵：添加区块到这个链上特别容易。比特币或者其他区块链的一个要旨是向链中添加区块具有一定的困难性。今天我们就来修复这个瑕疵。

PoW(工作量证明):


区块链中一个关键的理念是你要想往里面添加数据需要完成一些艰难的工作（如解难题，或者叫挖矿）才行。这些艰难的工作保证着区块链安全和一致性。所以区块链会奖励这些完成艰难工作的人（这就是为什么人们挖矿会奖励比特币）。

这种机制与生活中的一个现象非常相似：一个人需要努力工作从而获得报酬去维持他的生活。在区块链中，一些网络参与者（矿工）努力工作从而维持此网络的运行，他们向链中添加区块，并且从中获得奖励(如比特币)。 他们的工作使得一个区块被安全的正确无误的添加到区块链中，这也维持着整个区块链数据库的稳定。值得注意的是，谁完成这件工作的必须有一个证明。

整个“解难题和证明”的机制称作“工作量证明”。说他难是因为“解难题”需要大量的计算：即使是性能很好的计算机也不能快速的完成。并且这个工作的难度会随着时间持续增大以保证大约6块/h的出块速度。在区块链中这种工作（挖矿工作）的目标是为一个区块找到一个满足一些要求的的hash值。并且这个hash值也起到证明的作用。因此，挖矿实际上就是寻找这种证明。

最需要强调的是，“工作量证明算法”必需满足一个需求：求解这个难题很困难，但是验证它很简单。

哈希：

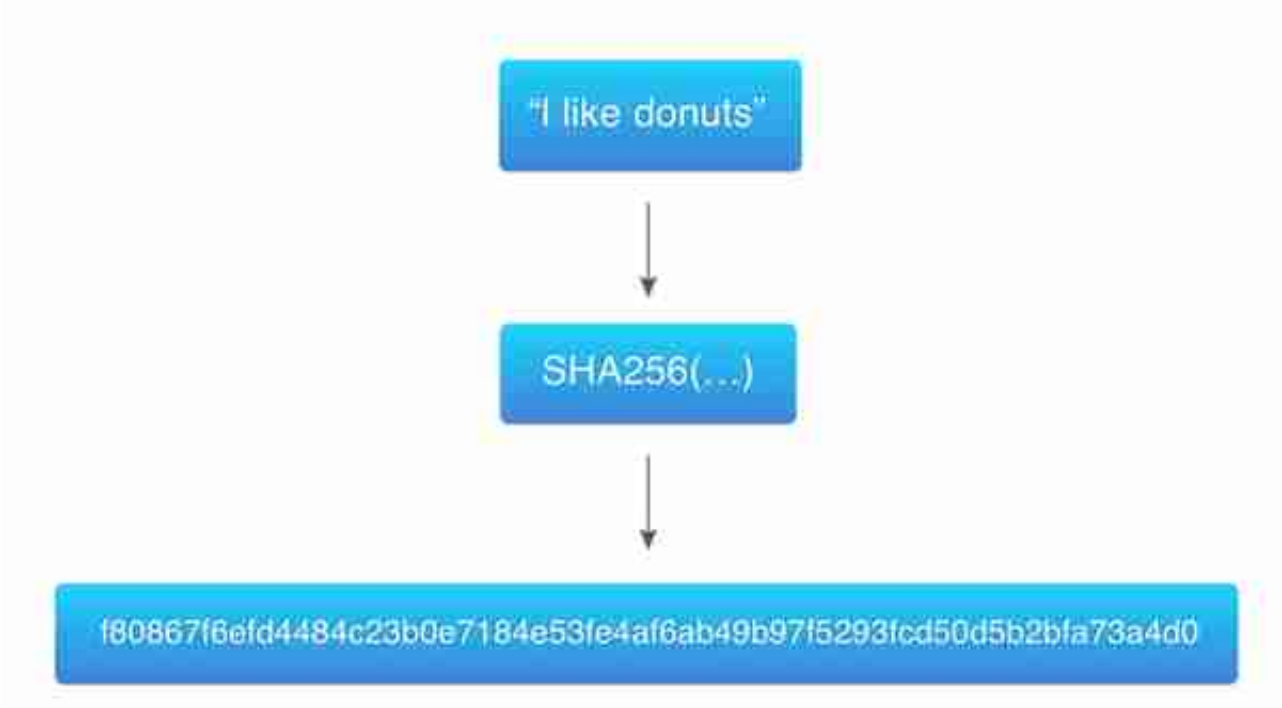

在这一小节中，我们将讨论哈希。如果你对这一块很熟悉，请跳过这一部分。

哈希的过程就是给某一数据找到对应的一个hash值的过程。哈希函数可以接受任意大小的数据并生成固定大小的hash值。哈希有如下几个特性：

1.给定一个hash值，不能由此反推出原始数据。因此，哈希不是加密。

2.确定的数据只有一个hash值，并且这个hash值是唯一的。

3.仅仅改动数据中的一个字节，得到的hash值就完全不同。




哈希函数在数据一致性方面有着广泛的应用。一些软件提供商将软件的验证码印在软件的包装上面。

用户下载文件后可以用一个给定的哈希函数生成一个哈希值与软件提供商提供的hash对比一下，看是否一致。

在区块链中，哈希用来保证区块的（添加到区块链前后的）一致性。区块链中哈希算法的入参包含前一个区块的哈希值，这使得恶意修改一个区块变的不可能，或者至少非常困难。因为这需要重新计算这个区块的哈希值以及所有之后的区块的哈希值。

哈希现金:

比特币中使用了Hashcash算法，这是一个PoW算法，设计的初衷是用来防止垃圾邮件。该算法可分解为以下几个步骤：

1.准备公开的数据(data):在邮件系统中为接收方的地址；在比特币中为区块头

2.添加一个计数器(counter)。从0开始计数。

3.计算data+counter组合的hash值

4.校验hash值是否满足特定的要求

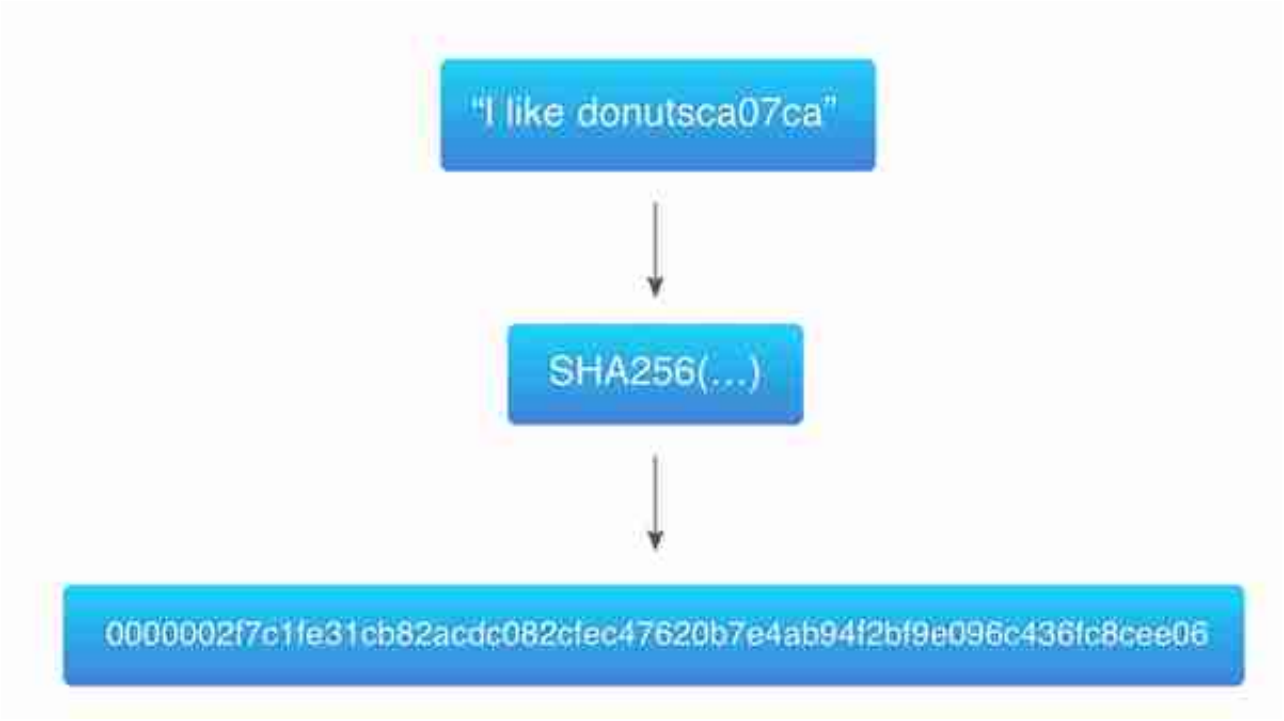
1).是，恭喜你成功了

2).否，counter++ 并且重复步骤3和4

所以，这是一个强制暴力计算的算法：改变counter，计算一个新的哈希值，检验它，不符合继续增加counter，继续计算一个新的哈希，等等，所以这是一个计算困难型的算法。

现在我们来看看目标哈希值需要满足的条件。在最早的Hashcash实现中，目标哈希值需要满足前20个比特都是0这样的要求。在比特币中，目标哈希满足的条件是随着时间变化的，因为在比特币的设计中，不管是计算能力随着时间推移的增长还是越来越多的员工加入到网络当中，出块的速度需要保持在每10分钟一个。

为了演示这一算法过程，我们使用前面例子中的数据("I like donuts")，目标哈希满足前三个字节全部为0:




图中
ca07ca是十六进制表示的计数器(counter)的值，相当于十进制中的13240266。

实现：

上面我们分析了理论基础，现在我们来编写代码。首先，我们定义挖矿的难度：

```
const targetBits = 24
```

比特币中，是在被挖出的区块的区块头中存储挖矿难度的。我们不打算实现一个目标哈希调整(挖矿难度的调整)的算法，现在我们仅仅定义一个全局的常量表示挖矿难度。

24是随意选取的，我们的目标是目标值占用内存空间在256个比特以内。并且我们希望这个"难度"足够有意义但是不能太大，因为"难度"越大就越难找到一个合适的哈希值。

```
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
}
```

创建一个ProofOfWork结构，它持有两个指针，一个指向区块，一个指向“目标值”。“目标值”就是算出的哈希满足的需求。我们之所以使用big.Int类型是因为我们需要比较哈希值和“目标值”：我们将哈希值转换为一个bigInt类型然后检验它是否比目标值小。

在NewProofOfWork函数中，我们初始化一个big.Int类型的变量，设置其值为1，然后左移256-targetBits 个比特位。256指的是SHA-256的比特位数，我的区块链中将使用SHA-256哈希算法。十六进制表示的“目标值”如下图所示：

```
0x1000000000000000000000000000000000000000000000000000000000000000
```

它占用29个字节的内存空间。下面是之前计算的哈希与“目标值”的比较：

```
0fac49161af82ed938add1d8725835cc123a1a87b1b196488360e58d4bfb51e3
0000010000000000000000000000000000000000000000000000000000000000
0000008b0f41ec78bab747864db66bcb9fb89920ee75f43fdaaeb5544f7f76ca
```

第一个哈希值("I like
donuts"计算得出的哈希值)大于目标值，因此它是无效的。第二个哈希值（ "I like donutsca07ca"计算得出的哈希值）小于目标值，因此它是有效的。

你可以认为“目标值”是有效哈希的上边界：如果一个哈希值小于这个边界值，它是有效的，反之则反。降低这个边界值意味着有效值的减少，因此，找到一个有效值需要付出更多的工作。

现在，我们准备计算哈希的数据。

```
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
}
```

这段代码很简单：我们仅仅将区块中的字段与目标值及nonce合并。nonce就是上文Hashcash中描述的计数器，这是一个密码学上的名词。

好了，所有的准备工作都完成了，我们可以开始实现PoW算法的核心了。

```
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，我们定义几个变量，hashInt：hash转换为整型后的值；nonce：计数器。接下来是一个(伪)"无限循环"，循环的最大次数为maxNonce，它与math.MaxInt64相等；这样是为了防止nonce的溢出。尽管在我们的Pow算法实现中计数器的溢出很难，但是检查一下更好，以防万一。

在循环中执行的步骤：

1.准备数据

2.用SHA-256计算数据的hash

3.得到hash的big integer 类型

4.将hash的整型值与目标值比较

现在我们可以Block的SetHash方法删掉了并且我们将NewBlock方法修改如下：

```
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
}
```

如你所见nonce将作为Block的一个属性保存在其中。这样做是必需的，因为工作量证明需要用到noce。Block结构变为如下所示的样子：

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

让我们启动程序看看我们的程序是否能够很好的工作。

```
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
```

非常好！现在你可以看到所有区块的hash都是以三个字节的0开头，并且生成这些hash值需要一些时间。

事情还没完，我们还有一件事要做：验证工作量。

```
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
}
```

这就是我们使用区块中存储的nonce的地方。

再看看我们的代码是否能正常运行：

```
func main() {
        ...

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

结果：

```
...

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
```

结论：

我们的区块链进一步接近它的真实架构了：添加区块需要一定的工作量证明，因此可以挖矿了。但是它依然缺少一些至关重要的特征：区块链数据没有持久化，缺少钱包，地址，交易以及共识机制。所有的这些特性，我们会在以后的文章中一一介绍。

- 尽力而为 -