Lập trình Blockchain với Golang. Part 1: Cơ bản

>Bài dịch từ _Building Blockchain in Go_ của tác giả _Ivan Kuznetsov_. Khi sử dụng vui lòng trích dẫn nguồn
[@hlongvu](https://github.com/hlongvu/blockchain-go-vietnamese)

Mục lục

Blockchain
Blockchain được xem là một trong những công nghệ mang tính cách mạng trong thế kỷ 21, đang dần trưởng thành và có những tiềm năng chưa khai phá hết. Về bản chất, blockchain chỉ là một bộ dữ liệu phân tán. Nhưng điều đặc biệt là nó không phải một cơ sở dữ liệu riêng tư (private database), mà nó là cơ sở dữ liệu mở (public database), bất kì ai cũng có thể sao chép tất cả (hoặc từng phần) bộ cơ sở dữ liệu này. Hơn nữa, nhờ có blockchain thì cryptocurrency (tiền-mã-hoá) và smart contracts mới có thể hiện thực hoá được.

Trong seri bài hướng dẫn này, chúng ta sẽ xây dựng một cryptocurrency đạng đơn giản, dựa trên công nghệ blockchain.

Block

Trong blockchain, block chính là phần chứa những dữ liệu có giá trị. Ví dụ, các block của Bitcoin chứa các transactions (giao dịch), phần cốt yếu của bất kì cryptocurrency nào. Ngoài những giao dịch này, block còn chứa những thông tin kỹ thuật khác, ví dụ version, timestamp và hash (mã băm) của block trước.

Trong hướng dẫn này chúng ta sẽ không xây dựng đầy đủ theo đặc tả của Bitcoin, mà chúng ta sẽ làm theo một cách đơn giản hơn, chỉ chứa những thông tin quan trọng và cốt yếu để blockchain có thể hoạt động được. Như sau:

```
```

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

Timestamp là thời điểm block được tạo, **Data** là những thông tin quan trọng mà block cần chứa, **PrevBlockHash** là hash của block trước đó, **Hash** chính là hash của block này.

Trong đặc tả kỹ thuật của Bitcoin, **Timestamp**, **PrevBlockHash**, **Hash** nằm trong phần block headers, còn các transactions (như **Data** ở trên) được lưu riêng ở phần dữ liệu khác. Hiện tại chúng ta đang lưu các trường chung trong một struct để đơn giản hoá.

Làm cách nào để tính các hash? Thuật toán để tính ra hash (hàm băm) đóng một vai trò rất quan trọng và chính là để bảo vệ blockchain. Hàm băm phải đòi hỏi tiêu tốn nhiều công sức (computationally difficult operation), thậm chí phải đầu tư CPU và GPU, ASICS cho việc tính toán này. Đây không phải là ngẫu nhiên mà đã được thiết kế có chủ định (by design), nhằm ngăn chặn việc có thể tạo ra các block một cách dễ dàng, khiến blockchain có thể bị thay đổi bằng các dữ liệu giả mạo. Chúng ta sẽ thảo luận thêm về thuật toán băm này trong các phần sau.

Hiện tại chúng ta chỉ cần lấy hàm băm SHA-256 của phần dữ liệu trong Block. Các dữ liệu được chuyển sang byte và ghép nối với nhau, sau đó lấy mã băm sha256.

```
func (b *Block) SetHash() {
	timestamp := []byte(strconv.FormatInt(b.Timestamp, 10))
	headers := bytes.Join([][]byte{b.PrevBlockHash, b.Data, timestamp},
[]byte{})
	hash := sha256.Sum256(headers)

	b.Hash = hash[:]
}
```

Và đây là hàm tạo một Block mới:

```
func NewBlock(data string, prevBlockHash []byte) *Block {
	block := &Block{time.Now().Unix(), []byte(data), prevBlockHash,
[]byte{}}
	block.SetHash()
	return block
}
```

Blockchain

Về cơ bản blockchain là một bộ dữ liệu có cấu trúc như một linked-list. Các block được lưu theo thứ tự và mỗi block đều có con trỏ link tới block liền kề phía trước. Chúng ta có thể truy cập được ngay block cuối cùng, và lấy block theo mã hash của block đó.

Chúng ta có thể xây dựng cấu trúc này bằng một array và một map: array sẽ lưu hash của các block, map sẽ lưu (hash: block). Blockchain đơn giản của chúng ta đang chỉ cần lưu array các block. Như sau:

```
type Blockchain struct {
	blocks []*Block
}
```

Đây chính là phiên bản blockchain đầu tiên của chúng ta ☺.

Hãy tạo hàm để thêm block vào blockchain:

```
func (bc *Blockchain) AddBlock(data string) {
	prevBlock := bc.blocks[len(bc.blocks)-1]
	newBlock := NewBlock(data, prevBlock.Hash)
	bc.blocks = append(bc.blocks, newBlock)
}
```

Để thêm block mới vào blockchain chúng ta cần có block liền trước nó. Vậy lúc blockchain chưa có block nào thì sao? Trong bất kì blockchain nào cũng phải có block đầu tiên này, và nó được gọi là **genesis block**. Đây là hàm để khởi tạo genesis block này:

```
func NewGenesisBlock() *Block {
	return NewBlock("Genesis Block", []byte{})
}
```

Tiếp theo là hàm tạo blockchain với genesis block này:

```
func NewBlockchain() *Blockchain {
	return &Blockchain{[]*Block{NewGenesisBlock()}}
}
```

Như vậy là đã hình thành blockchain đơn giản: có genesis block và có thể thêm các block tiếp theo vào blockchain.
Hãy kiểm tra xem blockchain này đã hoạt động chưa:

```
func main() {
	bc := NewBlockchain()

	bc.AddBlock("Send 1 BTC to Ivan")
	bc.AddBlock("Send 2 more BTC to Ivan")
```

```
        for _, block := range bc.blocks {
                fmt.Printf("Prev. hash: %x\n", block.PrevBlockHash)
                fmt.Printf("Data: %s\n", block.Data)
                fmt.Printf("Hash: %x\n", block.Hash)
                fmt.Println()
        }
}
```

Kết quả:

```
Prev. hash:
Data: Genesis Block
Hash: aff955a50dc6cd2abfe81b8849eab15f99ed1dc333d38487024223b5fe0f1168

Prev. hash: aff955a50dc6cd2abfe81b8849eab15f99ed1dc333d38487024223b5fe0f1168
Data: Send 1 BTC to Ivan
Hash: d75ce22a840abb9b4e8fc3b60767c4ba3f46a0432d3ea15b71aef9fde6a314e1

Prev. hash: d75ce22a840abb9b4e8fc3b60767c4ba3f46a0432d3ea15b71aef9fde6a314e1
Data: Send 2 more BTC to Ivan
Hash: 561237522bb7fcfbccbc6fe0e98bbbde7427ffe01c6fb223f7562288ca2295d1
```

Kết luận

Chúng ta vừa xây dựng một blockchain đơn giản: chỉ bao gồm một array các block, mỗi block có prevBlockHash trỏ tới block liền kề trước nó. Trong thực tế blockchain sẽ phức tạp hơn nhiều. Blockchain đơn giản trên có thể thêm block vào tuỳ ý và rất nhanh bởi hàm AddBlock, nhưng một blockchain thực thụ cần phải qua các bước tính toán nặng hơn nhiều: mỗi máy tính phải xử lí một khối lượng công việc rất lớn và đáp ứng điều kiện mới được quyền thêm block vào blockchain (gọi là Proof-of-Work). Ngoài ra blockchain còn là một bộ dữ liệu phân tán, và không ai có quyền làm chủ. Do đó mỗi block mới muốn được thêm vào blockchain phải được tất cả các máy tính khác trong mạng cho phép bằng một luật chung (consensus).

Blockchain của chúng ta cũng chưa có một transaction nào cả. Trong phần sau chúng ta sẽ thêm các tính năng mới vào blockchain này.

Links

1. Full source codes: https://github.com/Jeiwan/blockchain_go/tree/part_1
2. Block hashing algorithm: https://en.bitcoin.it/wiki/Block_hashing_algorithm

Mục lục

2. [Lập trình Blockchain với Golang. Part 2: Proof-of-work](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part2.md)
3. [Lập trình Blockchain với Golang. Part 3: Lưu trữ và tương tác CLI](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part3.md)
4. [Lập trình Blockchain với Golang. Part 4: Transactions 1](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part4.md)
5. [Lập trình Blockchain với Golang. Part 5: Address](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part5.md)
6. [Lập trình Blockchain với Golang. Part 6: Transaction 2](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part6.md)
7. [Lập trình Blockchain với Golang. Part 7: Network](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part7.md)

Lập trình Blockchain với Golang. Part 2: Proof-of-work

>Bài dịch từ _Building Blockchain in Go_ của tác giả _Ivan Kuznetsov_. Khi sử dụng vui lòng trích dẫn nguồn
[@hlongvu](https://github.com/hlongvu/blockchain-go-vietnamese)

Mục lục

Giới thiệu
Trong phần trước chúng ta đã xây dựng một cấu trúc dữ liệu đơn giản, là phần cốt lõi của blockchain. Chúng ta đã có thể thêm các block vào blockchain theo kiểu linked-list: mỗi block trỏ tới block liền kề trước đó. Tuy nhiên, blockchain của chúng ta mắc phải một lỗi cực kì nghiêm trọng: thêm block vào chuỗi hết sức dễ dàng. Một trong những chìa khoá quan trọng trong thiết kế của Bitcoin đó là việc thêm các block phải trải qua nhiều công sức tính toán. Trong phần này chúng ta sẽ sửa chữa khiếm khuyết trên.

Proof-of-work

Một trong những ý tưởng cốt lõi của blockchain đó là bất kì ai cũng phải bỏ ra nhiều công sức tính toán thì mới có thể thêm dữ liệu vào blockchain. Sự an toàn và đồng nhất của blockchain là nhờ thiết kế này. Thêm vào đó, khi bỏ công sức ra để thêm dữ liệu vào blockchain thì người tham gia đó sẽ được trao một phần thưởng (đây là lý do mọi người tham gia đào coin).

Cơ chế này tương tự trong đời thực: mỗi người bỏ công sức ra làm việc thì sẽ kiếm được tiền để chu cấp cho cuộc sống. Trong hệ thống blockchain, một số người tham gia đào coin (miners) phải liên tục tính toán để giúp hệ thống tiếp tục hoạt động, thêm block vào chuỗi và nhận phần thưởng (là coin đào được). Kết quả của việc đào coin (mining) là mỗi block được thêm vào blockchain sẽ càng thêm an toàn, giúp toàn bộ dữ liệu blockchain được ổn định và khó bị giả mạo. Nhưng đáng chú ý là những miners này phải chứng minh là họ đã bỏ công sức ra để mining.

Việc "bỏ công sức tính toán và chứng minh mình đã làm" này được gọi là **proof-of-work**. Việc này rất khó và thậm chí nhiều máy tính tốc độ cao cũng không thể thực hiện nhanh chóng. Thêm nữa, độ khó của công việc còn được tăng theo thời gian nhằm đảm bảo chỉ có 6 block được đào trong mỗi giờ. Trong Bitcoin, công việc là tìm ra một mã hash thoả mãn một số điều kiện. Mã hash thoả mãn đó được gọi là proof.

Một chú ý nữa, thuật toán **Proof-of-work** phải thoả mãn một điều kiện: tìm ra proof thì khó, nhưng xác nhận proof đó là đúng (verify) thì phải nhanh và dễ. Khi một người tìm ra proof đó và đưa cho những người trong mạng lưới kiểm tra thì họ có thể dễ dàng verify là đúng.

Hashing

Mục này chúng ta sẽ tìm hiểu về hàm băm (Hashing). Nếu bạn đã quen thuộc thì có thể bỏ qua.

Hashing là tiến trình để lấy mã băm (hash) của một dữ liệu nhất định. Mỗi hash đối với một dữ liệu (data) là duy nhất. Hàm băm (hash function) là một hàm với đầu vào là dữ liệu có độ lớn tuỳ ý, đầu ra là một hash có kích thước cố định. Sau đây là những tính chất của hàm băm:

1. Dữ liệu gốc không thể tái tạo lại chỉ dựa vào hash của dữ liệu đó. Vì thế hash không phải là mã hoá.
2. Một dữ liệu cố định thì chỉ sinh ra một hash cố định và không trùng lặp với hash của dữ liệu khác.
3. Thay đổi thậm chí 1 bit dữ liệu nhỏ trong data cũng sinh ra một hash tương ứng hoàn toàn khác ban đầu.

Hàm băm được sử dụng nhiều để kiểm tra tính toàn vẹn của dữ liệu. Ví dụ một số phần mềm sẽ có mã hash đi kèm để kiểm tra gói phần mềm chưa qua chỉnh sửa. Khi tải gói phần mềm về bạn có thể tính lại hash của nó đó để so sánh với mã hash mà nhà phát triển phần mềm cung cấp.

Trong hệ thống blockchain, Hashing được sử dụng để đảm bảo tính toàn vẹn của các block. Đầu vào của hàm băm lại có thêm mã hash của block liền kề trước, khiến cho việc thay đổi một block trong toàn bộ chuỗi là không thể (hoặc gần như không thể): Một khi một block thay đổi thì phải tính toán lại toàn bộ hash của block đó và các block liên tiếp phía sau.

>Các hash tính toán lại lại phải luôn đảm bảo proof-of-work, khiến cho khối lượng tính toán vô cùng lớn - Người dịch

Hashcash

Bitcoin sử dụng [Hashcash](https://en.wikipedia.org/wiki/Hashcash), một thuật toán Proof-of-Work vốn dùng để ngăn chặn thư rác. Thuật toán được chia làm các bước:

1. Lấy các dữ liệu đã biết (trong email thì là địa chỉ người nhận, trong Bitcoin thì là block headers).
2. Thêm một biến đếm (counter) vào dữ liệu đó, khởi đầu bằng 0.
3. Tính hash của **data + counter**
4. Kiểm tra xem hash tính được có thoả mãn điều kiện không:

	- Nếu thoả mãn thì hash đó là kết quả
	- Nếu không thoả mãn, tăng biến đếm counter lên và tiếp tục làm lại bước 3, 4

Việc lặp lại tính toán với biến đếm counter cho đến khi tìm được hash thoả mãn này là một quá trình tính toán với khối lượng lớn.

Bây giờ hãy xem điều kiện hash thoả mãn là gì. Trong phiên bản gốc của Hashcash, điều kiện một hash được chấp nhận là "20 bit đầu tiên của hash phải bằng 0". Ở Bitcoin, điều kiện này lại được thay đổi theo thời gian, bởi, theo thiết kế, mỗi block phải được tạo ra trong 10 phút, dù cho khả năng tính toán càng ngày càng lớn theo thời gian và số lượng miners.

Ví dụ với thuật toán này, data ban đầu là (“I like donuts”) khi kết hợp với counter là ca07ca thì kết quả hash thu được có 3 bytes bằng 0:

ca07ca là giá trị counter trong hệ số 16, trong hệ cơ số 10 nó tương đương 13240266

> Như vậy vòng lặp thuật toán phải chạy tới counter = 13240266 mới tìm ra một hash có 3 byte bằng 0. Điều đó cho thấy bạn phải tốn nhiều công sức tính toán thì mới tìm đc một hash thoả mãn điều kiện - Người dịch

Code

Như vậy chúng ta đã điểm qua phần thiết kế, hãy bắt tay xây dựng phần code. Đầu tiên chúng ta định nghĩa độ khó:

```
const targetBits = 24
```

Ở Bitcoin thì targetBits là một biến nằm trong block header, thể hiện độ khó của block đó khi được đào. Chúng ta không xây dựng phần thay đổi độ khó cho blockchain đơn giản này, nên targetBits được thể hiện là một hằng số.

24 cũng là một hằng số được chọn ngẫu nhiên, mục đích của chúng ta là chọn một độ khó bé hơn 256 bits, không nhỏ quá và cũng không quá lớn vì targetBits càng lớn thì độ khó càng tăng thêm nhiều lần.

```
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
}
```

```

Ở đây **ProofOfWork** là một struct chứa pointer tới 1 block và một target. Biến target này chính là độ khó. Chúng ta sử dụng big.Int để so sánh hash với target: chuyển hash thành big.Int và kiểm tra xem hash có bé hơn target hay không.

Trong hàm NewProofOfWork chúng ta khởi tạo target bằng 1 và shift left (dịch trái) 256 - targetBits bits. 256 là độ dài của một hash sinh ra bởi thuật toán băm SHA-256. Biểu diễn của target trong hệ 16 là:

```
0x0000100000000000000000000000000000000000000000000000000000000000
```

So sánh với 2 mã hash trong 2 ví dụ trước:

```
0fac49161af82ed938add1d8725835cc123a1a87b1b196488360e58d4bfb51e3
0000010000000000000000000000000000000000000000000000000000000000
0000008b0f41ec78bab747864db66bcb9fb89920ee75f43fdaaeb5544f7f76ca
```

Mã hash đầu tiên sinh ra từ data (“I like donuts”) lớn hơn target, do vậy không thoả mãn điều kiện. Mã hash thứ 2 sinh ra từ data (“I like donutsca07ca”) nhỏ hơn target, vì vậy nó thoả mãn điều kiện.

Bạn có thể hình dung target là chặn trên, nếu hash bé hơn chặn trên này thì thoả mãn, ngược lại thì không. Khi chặn trên này ngày càng hạ xuống, thì số lượng hash thoả mãn sẽ giảm xuống, đồng nghĩa với việc độ khó tăng lên và cần nhiều tính toán hơn để tìm hash.

Hàm dưới đây chuẩn bị data để tìm hash:

```
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
}
```

Chúng ta đơn giản chỉ nhập phần block với targetBits và nonce thành một mảng bytes, hàm hash sẽ thực hiện trên mảng bytes này.

```
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[:]
}
```

Chúng ta sẽ chạy một vòng lặp liên tục cho tới maxNonce (bằng math.MaxInt64). Thực tế thì chúng ta sẽ tìm ra hash thoả mãn sớm, không đến mức nonce phải tới giá trị math.MaxInt64 cực lớn này. Trong vòng lặp này chúng ta làm những việc sau:
1. Chuẩn bị data
2. Hash data với thuật toán SHA-256
3. Chuyển hash sang big.Int
4. So sánh con số nhận được với target

Trong phần 1 chúng ta có hàm SetHash của Block, giờ chúng ta thay thế bằng hash sinh ra từ proof-of-work này:

```
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
}
```

Biến nonce giờ được lưu trong Block, cần để verify lại kết quả một cách dễ dàng. Struct Block giờ được cập nhật lại như sau:

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

```
}
```

Và đây là kết quả khi chạy lại chương trình:

```
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
```

Bạn có thể thấy các hash đã có các bytes bằng 0 ở đầu, và phải mất 1 lúc mới tìm được các hash đó.

Còn một điều nữa, verify lại kết quả của proof-of-work như thế nào?

```
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
}
```

Như bạn có thể thấy biến nonce được lưu lại giúp việc verify trở nên dễ dàng.

Kiểm tra lại blockchain của chúng ta bằng hàm Validate():

```
func main() {
	...

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

```
                fmt.Println()
        }
}
```

Kết quả:

```
...

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
```

Kết luận

Blockchain chúng ta xây dựng đang dần giống với thực tế: thêm block mới cần có proof-of-work, là khởi đầu cho việc mining. Nhưng blockchain chúng ta còn thiếu nhiều tính năng thiết yếu khác: chưa lưu trữ data, chưa có wallet (ví), address (địa chỉ), transaction (giao dịch) và chưa có consensus (luật đồng thuận). Trong những phần tới chúng ta sẽ đề cập đến những vấn đề này.

Links

1. Full source code https://github.com/Jeiwan/blockchain_go/tree/part_2
2. Blockchain hashing algorithm https://en.bitcoin.it/wiki/Block_hashing_algorithm
3. Proof of work https://en.bitcoin.it/wiki/Proof_of_work
4. Hashcash https://en.bitcoin.it/wiki/Hashcash

Mục lục

3. [Lập trình Blockchain với Golang. Part 3: Lưu trữ và tương tác CLI](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part3.md)
4. [Lập trình Blockchain với Golang. Part 4: Transactions 1](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part4.md)
5. [Lập trình Blockchain với Golang. Part 5: Address](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part5.md)
6. [Lập trình Blockchain với Golang. Part 6: Transaction 2](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part6.md)
7. [Lập trình Blockchain với Golang. Part 7: Network](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part7.md)

Lập trình Blockchain với Golang. Part 3: Lưu trữ và tương tác CLI

>Bài dịch từ _Building Blockchain in Go_ của tác giả _Ivan Kuznetsov_. Khi sử dụng vui lòng trích dẫn nguồn
[@hlongvu](https://github.com/hlongvu/blockchain-go-vietnamese)

Mục lục

Giới thiệu
Phần này chúng ta sẽ tiến hành lưu trữ blockchain trong một database (DB) và xây dựng thêm command-line để tương tác với nó. Về cơ bản, blockchain là một bộ dữ liệu phân tán, tuy nhiên chúng ta sẽ lưu trữ dữ liệu trước và phần phân tán sẽ tiến hành sau.

Lựa chọn database
Bạn có thể sử dụng loại database nào? Sự thật là bạn có thể sử dụng bất cứ loại nào. Trong Bitcoin whitepaper cũng không hề chỉ định một loại database nào cụ thể, tuỳ vào lập trình viên lựa chọn mà thôi. Phiên bản [Bitcoin Core](https://github.com/bitcoin/bitcoin) được Satoshi Nakamoto lập trình sử dụng [LevelDB](https://github.com/google/leveldb). Còn với chúng ta khi sử dụng ngôn ngữ Golang thì sẽ lựa chọn...

BoltDB
Bởi:

1. Tính dễ dùng và tối giản
2. BoltDB được viết bởi Go
3. BoltDB không cần phải chạy trên một server
4. Các cấu trúc dữ liệu chúng ta muốn đều có thể sử dụng BoltDB

Từ giới thiệu của BoltDB trên Github:
> BoltDB là hệ lưu trữ key/value viết bằng Golang và được truyền cảm hứng bởi LMDB. Mục tiêu của nó là cung cấp một cơ sở dữ liệu đơn giản, nhanh và tin

cậy cho những project không cần tới hệ quản trị cơ sở dữ liệu đầy đủ như Postgres hoặc MySQL
>
> BoltDB được dùng ở những tác vụ cấp thấp nên tính đơn giản được đề cao trên hết. API của nó rất gọn nhẹ và chỉ tập trung vào việc lấy và lưu dữ liệu.

Xem chừng chỉ đó là đã đủ để chúng ta sử dụng trong project này. Hãy review lại đôi chút:

BoltDB là một bộ lưu dữ liệu theo key/value, có nghĩa là không cần tới table như ở SQL RDBMS (MySQL, PostgreSQL,...), không hàng, cột. Thay vào đó dữ liệu được lưu theo key-value (giống Golang map). Các cặp key-value này được lưu trong các bucket, mục đích là để gộp các dữ liệu giống nhau lại. Vậy, để lấy ra một giá trị (value), chúng ta cần biết bucket và key.

Một chú ý quan trọng trong BoltDB là nó không có kiểu dữ liệu: key và value đều ở dạng mảng byte. Để có thể lưu trữ Go struct, chúng ta phải serialize chúng - chuyển struct thành byte array và chuyển ngược lại byte array thành struct. Chúng ta sẽ sử dụng thư viện [encoding/gob](https://golang.org/pkg/encoding/gob/) để làm điều này. Mặc dù có thể dùng các kiểu dữ liệu khác JSON, XML, Protocol Bufers,... nhưng chúng ta sử dụng encoding/gob vì tính đơn giản và được hỗ trợ trực tiếp từ thư viện chuẩn của Go.

Cấu trúc dữ liệu

Trước khi tiến hành code phần lưu trữ dữ liệu, chúng ta hãy nghiên cứu xem nên lưu dữ liệu như thế nào. Hãy tham khảo xem Bitcoin Core thực hiện ra sao.

Bitcoin Core sử dụng 2 "buckets" để lưu dữ liệu:

1. blocks để lưu các dữ liệu về block trong chuỗi
2. chainstate để lưu trạng thái của chuỗi, ví dụ các giao dịch chưa sử dụng (unspent transaction outputs) và một số dữ liệu kỹ thuật khác

Thêm vào đó, blocks được lưu ra nhiều file trên ổ đĩa. Việc này là để tối ưu tốc độ đọc ghi. Chúng ta chưa cần thiết phải xây dựng phần này.

Trong **blocks**, các cặp **key->value** là:

1. **'b' + 32-byte block hash -> block index record**
2. **'f' + 4-byte file number -> file information record**
3. **'l' -> 4-byte file number: the last block file number used**
4. **'R' -> 1-byte boolean: whether we're in the process of reindexing**
5. **'F' + 1-byte flag name length + flag name string -> 1 byte boolean: various flags that can be on or off**
6. **'t' + 32-byte transaction hash -> transaction index record**

Trong **chainstats**, các cặp **key->value** là:

1. **'c' + 32-byte transaction hash -> unspent transaction output record for that transaction**
2. **'B' -> 32-byte block hash: the block hash up to which the database represents the unspent transaction outputs**

(Đặc tả cụ thể các bạn có thể xem [tại đây](https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_2):_Data_Storage))

Trong blockchain của chúng ta chưa có transaction nào cả, vì thế chúng ta chỉ sử dụng **blocks** bucket, và bucket này cũng chỉ lưu trong 1 file. Vậy các cặp **key->value** mà chúng ta có là:

1. **'b' + 32-byte block hash -> block index record**
3. **'l' -> hash của block cuối cùng trong chuỗi**

Vậy hãy bắt tay vào code.

Serialization

Như đã nói, boltDB chỉ có thể lưu trữ kiểu []byte trong khi đó chúng ta lại lưu Block kiểu struct. Sử dụng encoding/gob để serialize:

```
func (b *Block) Serialize() []byte {
        var result bytes.Buffer
        encoder := gob.NewEncoder(&result)

        err := encoder.Encode(b)

        return result.Bytes()
}
```
Tiếp theo là hàm Deserialize:

```
func DeserializeBlock(d []byte) *Block {
        var block Block

        decoder := gob.NewDecoder(bytes.NewReader(d))
        err := decoder.Decode(&block)

        return &block
}
```

>Các bạn có thể tham khảo thêm thư viện [encoding/gob](https://golang.org/pkg/encoding/gob/).

Lưu trữ

Bắt đầu với hàm **NewBlockchain**, hiện tại hàm này tạo ra một **Blockchain** và thêm genesis block vào đó. Chúng ra sẽ sửa lại như sau:

1. Mở một file DB
2. Kiểm tra xem đã có blockchain nào trong dữ liệu chưa
3. Nếu đã có:
        - Tạo một instance Blockchain mới
        - Đặt đỉnh (tip) của Blockchain này là hash cuối đã lưu trong DB
4. Nếu chưa có blockchain nào:
        - Tạo genesis block
        - Lưu lại trong DB
        - Lưu lại hash của genesis block vào hash cuối trong DB
        - Tạo một instance Blockchain với tip là hash đó

Xây dựng bằng code sẽ như sau:

```
func NewBlockchain() *Blockchain {
	var tip []byte
	db, err := bolt.Open(dbFile, 0600, nil)

	err = db.Update(func(tx *bolt.Tx) error {
		b := tx.Bucket([]byte(blocksBucket))

		if b == nil {
			genesis := NewGenesisBlock()
			b, err := tx.CreateBucket([]byte(blocksBucket))
			err = b.Put(genesis.Hash, genesis.Serialize())
			err = b.Put([]byte("l"), genesis.Hash)
			tip = genesis.Hash
		} else {
			tip = b.Get([]byte("l"))
		}

		return nil
	})

	bc := Blockchain{tip, db}

	return &bc
}
```

Hãy xem lại các bước:

```
db, err := bolt.Open(dbFile, 0600, nil)
```
Đây là câu lệnh để mở một file BoltDB. Chú ý là nó sẽ không trả về error nếu file không tồn tại.

```
err = db.Update(func(tx *bolt.Tx) error {
...
})
```
Trong BoltDB, mỗi thao tác với dữ liệu được chạy trong một transaction (Giao dịch). Có 2 loại transaction: chỉ đọc và đọc-ghi. Ở đây chúng ta mở một transaction đọc ghi (**db.Update(...)**) để có thể ghi genesis block vào DB.

```
b := tx.Bucket([]byte(blocksBucket))

if b == nil {
	genesis := NewGenesisBlock()
	b, err := tx.CreateBucket([]byte(blocksBucket))
	err = b.Put(genesis.Hash, genesis.Serialize())
	err = b.Put([]byte("l"), genesis.Hash)
	tip = genesis.Hash
} else {
	tip = b.Get([]byte("l"))
```

```
}
```

Đây là phần chính của hàm NewBlockchain. Ở đây chúng ta kiểm tra xem blockchain đã tồn tại chưa. Nếu chưa có, chúng ta tạo ra genesis block, tạo bucket và lưu block và cập nhật key "l" lưu lại hash của block cuối trong chuỗi.

Đồng thời chúng ta cũng tạo ra instance Blockchain:

```
bc := Blockchain{tip, db}
```

Chúng ta không lưu blockchain bằng mảng block nữa, chúng ta chỉ lưu lại đỉnh (tip), là hash của block cuối cùng. Chúng ta cũng giữ lại một biến db để có thể truy cập nhanh tới blockchain. Struct mới của chúng ta sẽ như sau:

```
type Blockchain struct {
	tip []byte
	db  *bolt.DB
}
```

Tiếp theo chúng ta sẽ cập nhật lại hàm **AddBlock**, không đơn giản như việc thêm một block mới vào mảng nữa, chúng ta phải lưu vào DB.

```
func (bc *Blockchain) AddBlock(data string) {
	var lastHash []byte

	err := bc.db.View(func(tx *bolt.Tx) error {
		b := tx.Bucket([]byte(blocksBucket))
		lastHash = b.Get([]byte("l"))

		return nil
	})

	newBlock := NewBlock(data, lastHash)

	err = bc.db.Update(func(tx *bolt.Tx) error {
		b := tx.Bucket([]byte(blocksBucket))
		err := b.Put(newBlock.Hash, newBlock.Serialize())
		err = b.Put([]byte("l"), newBlock.Hash)
		bc.tip = newBlock.Hash

		return nil
	})
}
```

Hãy xem lại từng bước:

```
err := bc.db.View(func(tx *bolt.Tx) error {
	b := tx.Bucket([]byte(blocksBucket))
	lastHash = b.Get([]byte("l"))

	return nil
```

```
})
```
Phía trên là 1 transaction read-only để lấy hash cuối cùng của blockchain. Hash này sẽ được dùng để mine block tiếp theo.

```
newBlock := NewBlock(data, lastHash)
b := tx.Bucket([]byte(blocksBucket))
err := b.Put(newBlock.Hash, newBlock.Serialize())
err = b.Put([]byte("l"), newBlock.Hash)
bc.tip = newBlock.Hash
```
Sau khi mine được newBlock chúng ta lưu lại vào DB và cập nhật key "l" là hash của block mới này.

Dễ phải không?

Kiểm tra Blockchain mới

Blockchain của chúng ta đã được lưu vào cơ sở dữ liệu, chúng ta có thể tắt mở chương trình và thêm block vào đó. Nhưng các block không lưu trong array nữa nên phần in ra các block đã không còn hoạt động. Hãy sửa phần này!

BoltDB cho phép chúng ta quét qua tất cả các key trong bucket, tuy nhiên các key lại sắp xếp theo thứ tự byte chứ không phải thứ tự block thêm vào. Đồng thời chúng ta cũng không muốn tải toàn bộ block vào bộ nhớ, vì vậy hãy đọc chúng từng block một. Để xây dựng tính năng này, chúng ta cần một bộ quét (iterator):

```
type BlockchainIterator struct {
        currentHash []byte
        db          *bolt.DB
}
```
Mỗi iterator sẽ được tạo lúc cần quét qua các blocks, iterator này sẽ lưu kết nối db và hash của block hiện tại đang trỏ tới. Iterator này được tạo ra từ Blockchain:

```
func (bc *Blockchain) Iterator() *BlockchainIterator {
        bci := &BlockchainIterator{bc.tip, bc.db}

        return bci
}
```
Chú ý là iterator này lúc khởi tạo cũng trỏ tới tip của blockchain, vì thế khi quét, các block sẽ lần lượt từ mới nhất tới cũ nhất.

Trong thực tế một blockchain có thể có nhiều nhánh, nhánh dài nhất sẽ được chọn làm nhánh chính. Tip này cũng có thể là bất kì block nào. Nên tip này có thể được hiểu như là một id của blockchain. Việc chọn một tip cũng là "voting" cho một blockchain.

BlockchainIterator chỉ thực hiện một nhiệm vụ: trả về block tiếp theo trong blockchain:

```
func (i *BlockchainIterator) Next() *Block {
	var block *Block

	err := i.db.View(func(tx *bolt.Tx) error {
		b := tx.Bucket([]byte(blocksBucket))
		encodedBlock := b.Get(i.currentHash)
		block = DeserializeBlock(encodedBlock)

		return nil
	})

	i.currentHash = block.PrevBlockHash

	return block
}
```

Phần DB đến đây cơ bản đã xong.

CLI

Blockchain hiện tại chỉ mới chạy trong hàm main mà chưa cho phép tương tác nào. Chúng ta sẽ tiếp tục xây dựng các command cho chương trình như sau:

```
blockchain_go addblock "Pay 0.031337 for a coffee"
blockchain_go printchain
```

Các command này sẽ được thực thi qua CLI struct:

```
type CLI struct {
	bc *Blockchain
}
```

Hàm **Run** sẽ chạy các command:

```
func (cli *CLI) Run() {
	cli.validateArgs()

	addBlockCmd := flag.NewFlagSet("addblock", flag.ExitOnError)
	printChainCmd := flag.NewFlagSet("printchain", flag.ExitOnError)

	addBlockData := addBlockCmd.String("data", "", "Block data")

	switch os.Args[1] {
	case "addblock":
		err := addBlockCmd.Parse(os.Args[2:])
	case "printchain":
		err := printChainCmd.Parse(os.Args[2:])
	default:
		cli.printUsage()
		os.Exit(1)
```

```
        }

        if addBlockCmd.Parsed() {
                if *addBlockData == "" {
                        addBlockCmd.Usage()
                        os.Exit(1)
                }
                cli.addBlock(*addBlockData)
        }

        if printChainCmd.Parsed() {
                cli.printChain()
        }
}
```

Các command được thư viện [flag](https://golang.org/pkg/flag/) parse.

```
addBlockCmd := flag.NewFlagSet("addblock", flag.ExitOnError)
printChainCmd := flag.NewFlagSet("printchain", flag.ExitOnError)
addBlockData := addBlockCmd.String("data", "", "Block data")
```
Hàm addblock sẽ có thêm trường data là dữ liệu muốn lưu.

```
switch os.Args[1] {
case "addblock":
        err := addBlockCmd.Parse(os.Args[2:])
case "printchain":
        err := printChainCmd.Parse(os.Args[2:])
default:
        cli.printUsage()
        os.Exit(1)
}
```

Tiếp theo chúng ta sẽ kiểm tra command và trường đi kèm.

```
if addBlockCmd.Parsed() {
        if *addBlockData == "" {
                addBlockCmd.Usage()
                os.Exit(1)
        }
        cli.addBlock(*addBlockData)
}

if printChainCmd.Parsed() {
        cli.printChain()
}
```
Tiếp theo là check và chạy hàm tương ứng.

```
func (cli *CLI) addBlock(data string) {
        cli.bc.AddBlock(data)
```

```
        fmt.Println("Success!")
}

func (cli *CLI) printChain() {
        bci := cli.bc.Iterator()

        for {
                block := bci.Next()

                fmt.Printf("Prev. hash: %x\n", block.PrevBlockHash)
                fmt.Printf("Data: %s\n", block.Data)
                fmt.Printf("Hash: %x\n", block.Hash)
                pow := NewProofOfWork(block)
                fmt.Printf("PoW: %s\n", strconv.FormatBool(pow.Validate()))
                fmt.Println()

                if len(block.PrevBlockHash) == 0 {
                        break
                }
        }
}
```

Các hàm trên tương tự như chúng ta đã xây dựng, chỉ có bây giờ chúng ta dùng BlockchainIterator để quét qua blockchain.

Hàm main của chúng ta giờ được cập nhật lại như sau:

```
func main() {
        bc := NewBlockchain()
        defer bc.db.Close()

        cli := CLI{bc}
        cli.Run()
}
```

Chú ý: Dù bạn chạy command nào thì một Blockchain cũng được tạo ra.

Bây giờ hãy kiểm tra xem chương trình có chạy đúng không:

```
$ blockchain_go printchain
No existing blockchain found. Creating a new one...
Mining the block containing "Genesis Block"
000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b

Prev. hash:
Data: Genesis Block
Hash: 000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b
PoW: true

$ blockchain_go addblock -data "Send 1 BTC to Ivan"
Mining the block containing "Send 1 BTC to Ivan"
000000d7b0c76e1001cdc1fc866b95a481d23f3027d86901eaeb77ae6d002b13
```

```
Success!

$ blockchain_go addblock -data "Pay 0.31337 BTC for a coffee"
Mining the block containing "Pay 0.31337 BTC for a coffee"
000000aa0748da7367dec6b9de5027f4fae0963df89ff39d8f20fd7299307148

Success!

$ blockchain_go printchain
Prev. hash: 000000d7b0c76e1001cdc1fc866b95a481d23f3027d86901eaeb77ae6d002b13
Data: Pay 0.31337 BTC for a coffee
Hash: 000000aa0748da7367dec6b9de5027f4fae0963df89ff39d8f20fd7299307148
PoW: true

Prev. hash: 000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b
Data: Send 1 BTC to Ivan
Hash: 000000d7b0c76e1001cdc1fc866b95a481d23f3027d86901eaeb77ae6d002b13
PoW: true

Prev. hash:
Data: Genesis Block
Hash: 000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b
PoW: true
```

Kết luận

Ở phần sau chúng ta sẽ xây dựng address, wallet và transactions.

Links

1. [Full source codes](https://github.com/Jeiwan/blockchain_go/tree/part_3)
2. [Bitcoin Core Data Storage](https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_2):_Data_Storage)
3. [boltdb](https://github.com/boltdb/bolt)
4. [encoding/gob](https://golang.org/pkg/encoding/gob/)
5. [flag](https://golang.org/pkg/flag/)

Mục lục

5. [Lập trình Blockchain với Golang. Part 5: Address](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part5.md)
6. [Lập trình Blockchain với Golang. Part 6: Transaction 2](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part6.md)
7. [Lập trình Blockchain với Golang. Part 7: Network](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part7.md)

Lập trình Blockchain với Golang. Part 4: Transactions 1

>Bài dịch từ _Building Blockchain in Go_ của tác giả _Ivan Kuznetsov_. Khi sử dụng vui lòng trích dẫn nguồn
[@hlongvu](https://github.com/hlongvu/blockchain-go-vietnamese)

Mục lục

Giới thiệu
Transaction là trái tim của Bitcoin và nhiệm vụ duy nhất của blockchain là lưu trữ transaction một cách an toàn và tin cậy,để không ai có thể thay đổi chúng sau khi được tạo. Phần này chúng ta sẽ xây dựng transaction. Nhưng đây là một topic khá lớn nên sẽ được chia thành 2 phần. Phần 1 này chúng ta sẽ xây dựng kiến trúc tổng thể của transaction và sẽ đi vào cụ thể ở phần tới.

Hàm lượng code thay đổi là rất lớn nên các bạn có thể theo dõi tất cả ở [đây](https://github.com/Jeiwan/blockchain_go/compare/part_3...part_4#files_bucket)

There is no spoon
> Quote từ phim [Ma trận](https://www.youtube.com/watch?v=uAXtO5dMqEI)

Nếu bạn từng xây dựng một web app, để có phần thanh toán (payment) thường thì bạn phải có 2 tables này trong DB: **accounts** và **transactions**. Một account thường sẽ lưu thông tin user, bao gồm thông tin cá nhân, số dư tài khoản, và mỗi transaction sẽ là thông tin về việc chuyển tiền từ tài khoản này qua tài khoản kia. Trong Bitcoin, việc thanh toán được thể hiện bằng một cách hoàn toàn khác:

1. Không có accounts
2. Không có balance
3. Không có address
4. Không có coins
5. Không có người gửi và người nhận

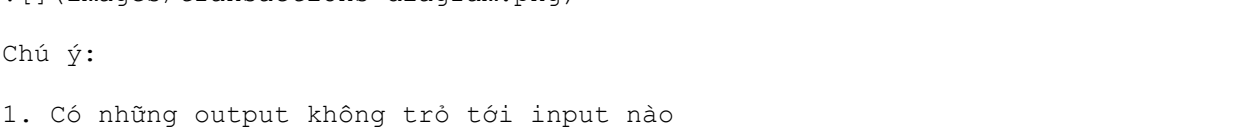
Vì Blockchain là một bộ dữ liệu mở, chúng ta hoàn toàn không muốn lưu giữ các thông tin nhạy cảm của người dùng. Coins không có trong các accounts. Transaction thì không phải từ người này qua người kia. Thậm chí không có trường nào để lưu balance. Tất cả chỉ là các transaction.

Vậy trong transaction có gì?

Bitcoin Transaction
Một transaction là tổng hợp của các inputs và outputs

```
type Transaction struct {
        ID   []byte
        Vin  []TXInput
        Vout []TXOutput
}
```

Inputs của transaction này lại chính là outputs của một transaction trước đó (có một số ngoại lệ, chúng ta sẽ thảo luận sau). Outputs chính là nơi mà coins được lưu giữ. Biểu đồ sau thể hiện mối liên kết giữa các transaction:



Chú ý:

1. Có những output không trỏ tới input nào
2. Trong một transaction, inputs có thể trỏ tới nhiều outputs của nhiều transaction trước
3. Mỗi input phải trỏ tới 1 output

Chúng ta có thể sử dụng các thuật ngữ như "money", "coin","spend","send","account",... nhưng các khái niệm này không tồn tại trong Bitcoin. Các transaction khoá các giá trị (value) lại, và chỉ có người khoá mới mở được khoá này, để có thể "spend" (sử dụng).

Transaction Outputs
Hãy xem cấu trúc của một transaction output:

```
type TXOutput struct {
        Value        int
        ScriptPubKey string
}
```

Hãy chú ý trường **Value** ở trên, đây chính là coins mà outputs lưu giữ (store). Store chính là khoá value lại với một puzzle (**ScriptPubKey**), đòi hỏi phải giải được mới có quyền sử dụng value này.

Trong thiết kế của Bitcoin, có một ngôn ngữ được sử dụng là *Script*, để định nghĩa logic khoá và mở khoá. Ngôn ngữ này khá sơ khai (được tạo nên có chủ ý nhằm tránh bị hack và dùng sai), và chúng ta không đi sâu vào. Nếu bạn quan tâm có thể tham khảo ở [đây](https://en.bitcoin.it/wiki/Script).

> Bitcoin lưu value theo đơn vị _satoshis_, không phải đơn vị BTC. Mỗi satoshi tương ứng 0.00000001 BTC, là đơn vị nhỏ nhất của Bitcoin.

> Bitcoin cũng có khả năng tạo smart-contract nhờ vào ngôn ngữ Script này

Chúng ta chưa xây dựng address vì thế để tránh các logic về khoá và mở khoá, chúng ta sử dụng ScriptPubKey để lưu một string (address của user)

Một điều quan trọng cần nhớ là output không thể chia nhỏ ra, nghĩa là bạn không thể trỏ tới mỗi 1 phần nhỏ của output. Khi một output được trỏ tới trong transaction thì nó sẽ được sử dụng toàn phần. Nếu giá trị của output đó lớn hơn giá trị cần sử dụng, một phần dư sẽ được tạo ra và gửi lại cho chính chủ. Tương tự với tình huống trong thực tế, khi bạn thanh toán 1$ bằng tờ 5$ thì bạn được trả lại 4$.

Transaction Inputs
Đây là cấu trúc của một transaction input:

```
type TXInput struct {
        Txid      []byte
        Vout      int
        ScriptSig string
}
```

Như đã đề cập ở phần trên, mỗi input trỏ tới 1 output của transaction khác. **Txid** lưu giữ id của transaction đó, **Vout** là index (vị trí) của output trong transaction đó. **ScriptSig** là một script lưu trữ data để sử dụng cùng với **ScriptPubKey** của output. Nếu data đúng thì output có thể được mở (unlock) và value của output có thể sử dụng để tạo nên output mới. Nếu không đúng, output sẽ không được quyền sử dụng trong input này. Đây là cơ chế giúp một người không thể sử dụng coins của người khác.

Chúng ta chưa xây dựng address, vậy nên **ScriptSig** cũng chỉ lưu một string thể hiện address của user. Phần public key và kiểm tra signature sẽ được đề cập trong hướng dẫn sau.

Tóm tắt lại như sau. Output là nơi "coins" được lưu trữ. Mỗi output đi kèm với một mã khoá. Các transaction phải có ít nhất 1 input và 1 output. Mỗi input trỏ tới 1 output từ transaction khác trước đó và phải có phần **ScriptSig** để mở khoá output đó.

Vậy cái gì có trước: input hay output?

Con gà và quả trứng

Có input thì mới có output và có output mới có input là vấn đề con gà quả trứng kinh điển. Trong Bitcoin thì output có trước input.

Khi miner đào một block, có một transaction đặc biệt trong block đó gọi là **coinbase transaction**. Đó là transaction không cần có input. Nó tạo ra output từ hư không. Đây chính là phần thưởng mà miner nhận được khi tham gia đào.

Như bạn đã biết, có genesis block ở đầu blockchain. Block này chứa output đầu tiên của blockchain. Output này không cần tới output hay input nào của hệ thống vì trước đó không có gì cả.

Đây là cấu trúc của coinbase transaction:

```
func NewCoinbaseTX(to, data string) *Transaction {
	if data == "" {
		data = fmt.Sprintf("Reward to '%s'", to)
	}

	txin := TXInput{[]byte{}, -1, data}
	txout := TXOutput{subsidy, to}
	tx := Transaction{nil, []TXInput{txin}, []TXOutput{txout}}
	tx.SetID()

	return &tx
}
```

Một coinbase transaction chỉ có 1 input. Trong blockchain của chúng ta nó được định nghĩa là **Txid** rỗng và **Vout = -1**. Đồng thời không có **ScriptSig** mà chỉ là data tuỳ ý.

> Coinbase transaction đầu tiên của Bitcoin lưu giữ thông điệp này: "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks". [Bạn có thể thấy ở
đây](https://blockchain.info/tx/4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b?show_adv=true).

Hãy chú ý trường **subsidy** ở trên. Đây chính là số coins phần thưởng dành cho miner. Chúng ta lưu với biến hằng số. Tuy nghiên trong Bitcoin thì nó được tính dựa vào số blocks hiện tại. Đào block đầu tiên của Bitcoin bạn nhận được 50 BTC và sau đó cứ **210000** blocks thì sẽ giảm một nửa.

Lưu trữ Transaction trong Blockchain

Từ giờ mỗi block phải chứa ít nhất một transaction và không thể đào block rỗng nữa. Trường **Data** sẽ được thay bằng mảng transaction:

```
type Block struct {
	Timestamp     int64
	Transactions  []*Transaction
	PrevBlockHash []byte
	Hash          []byte
	Nonce         int
}
```

Các hàm **NewBlock** và **NewGenesisBlock** cũng sẽ được cập nhật lại:

```
func NewBlock(transactions []*Transaction, prevBlockHash []byte) *Block {
```

```
        block := &Block{time.Now().Unix(), transactions, prevBlockHash,
[]byte{}, 0}
        ...
}

func NewGenesisBlock(coinbase *Transaction) *Block {
        return NewBlock([]*Transaction{coinbase}, []byte{})
}
```

Hàm tạo blockchain mới cũng được cập nhật:

```
func CreateBlockchain(address string) *Blockchain {
        ...
        err = db.Update(func(tx *bolt.Tx) error {
                cbtx := NewCoinbaseTX(address, genesisCoinbaseData)
                genesis := NewGenesisBlock(cbtx)

                b, err := tx.CreateBucket([]byte(blocksBucket))
                err = b.Put(genesis.Hash, genesis.Serialize())
                ...
        })
        ...
}
```

Hàm này sẽ có tham số là address, là địa chỉ nhận reward cho genesis block.

Proof-of-work

Thuật toán proof-of-work cũng phải thực hiện trên phần transaction của block, để có thể đảm bảo các transaction này an toàn. Hàm **ProofOfWork.prepareData** sẽ được cập nhật lại:

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

        return data
}
```

Thay vì **pow.block.Data** chúng ta dùng **pow.block.HashTransactions()** được tạo nên từ các transactions:

```
func (b *Block) HashTransactions() []byte {
        var txHashes [][]byte
        var txHash [32]byte
```

```
        for _, tx := range b.Transactions {
                txHashes = append(txHashes, tx.ID)
        }
        txHash = sha256.Sum256(bytes.Join(txHashes, []byte{}))

        return txHash[:]
}
```

Chúng ta muốn tất cả các transaction trong block đều được gộp lại và cho ra một hash duy nhất. Để làm được điều này chúng ta hash mỗi transaction rồi ghép lại với nhau, cuối cùng lấy hash của dữ liệu gộp đó.

> Bitcoin sử dụng một kỹ thuật cao hơn, các transaction được lưu trữ trong một [Merkle tree](https://en.wikipedia.org/wiki/Merkle_tree), mã hash ở gốc tree đó sẽ được dùng cho thuật toán proof-of-work. Nhờ cấu trúc này có thể kiểm tra nhanh chóng 1 transaction có nằm trong block hay không. Và chỉ cần lưu trữ mã gốc thay vì lưu trữ toàn bộ transactions.

Bây giờ hãy kiểm tra xem blockchain mới có hoạt động hay không:

```
$ blockchain_go createblockchain -address Ivan
00000093450837f8b52b78c25f8163bb6137caf43ff4d9a01d1b731fa8ddcc8a

Done!
```

Chúng ta đã tạo được blockchain và reward tới địa chỉ Ivan. Nhưng làm cách nào để kiểm tra số dư (balance)?

Unspent Transaction Outputs - output chưa sử dụng

Unspent nghĩa là output này chưa được input nào trỏ tới. Chúng ta cần tìm hết tất cả các output chưa sử dụng này (UTXO). Trong sơ đồ ở hình ảnh trước, UTXO bao gồm:

1. tx0, output 1;
2. tx1, output 0;
3. tx3, output 0;
4. tx4, output 0.

Tất nhiên lúc kiểm tra balance cho một địa chỉ, chúng ta không cần tới tất cả các UTXO, mà chỉ các UTXO có thể mở khoá được bởi địa chỉ đó. Đầu tiên hãy định nghĩa cách khoá - mở khoá cho input và output:

```
func (in *TXInput) CanUnlockOutputWith(unlockingData string) bool {
        return in.ScriptSig == unlockingData
}

func (out *TXOutput) CanBeUnlockedWith(unlockingData string) bool {
        return out.ScriptPubKey == unlockingData
}
```

Ở đây chúng ta chỉ đơn giản so sánh mã script với **unlockingData**. Chúng sẽ được cập nhật trong các phần hướng dẫn tiếp theo khi đã có private keys và address.

Bước tiếp theo chúng ta sẽ tìm các transaction có chứa UTXO:

```
func (bc *Blockchain) FindUnspentTransactions(address string) []Transaction {
  var unspentTXs []Transaction
  spentTXOs := make(map[string][]int)
  bci := bc.Iterator()

  for {
    block := bci.Next()

    for _, tx := range block.Transactions {
      txID := hex.EncodeToString(tx.ID)

    Outputs:
      for outIdx, out := range tx.Vout {
        // Was the output spent?
        if spentTXOs[txID] != nil {
          for _, spentOut := range spentTXOs[txID] {
            if spentOut == outIdx {
              continue Outputs
            }
          }
        }

        if out.CanBeUnlockedWith(address) {
          unspentTXs = append(unspentTXs, *tx)
        }
      }

      if tx.IsCoinbase() == false {
        for _, in := range tx.Vin {
          if in.CanUnlockOutputWith(address) {
            inTxID := hex.EncodeToString(in.Txid)
            spentTXOs[inTxID] = append(spentTXOs[inTxID], in.Vout)
          }
        }
      }
    }

    if len(block.PrevBlockHash) == 0 {
      break
    }
  }

  return unspentTXs
}
```

Các transaction đều được lưu trong block, vì vậy chúng ta phải kiểm tra hết tất cả các block trong blockchain. Bắt đầu bằng outputs:

```
if out.CanBeUnlockedWith(address) {
        unspentTXs = append(unspentTXs, tx)
}
```

Nếu output được khoá bởi cùng địa chỉ mà chúng ta đang tính balance thì đó là output cần tìm. Nhưng trước khi lấy output đó, cần kiểm tra thêm xem output đó đã bị trỏ bởi input nào chưa:

```
if spentTXOs[txID] != nil {
        for _, spentOut := range spentTXOs[txID] {
                if spentOut == outIdx {
                        continue Outputs
                }
        }
}
```

Những output đã bị trỏ tới bởi input khác có nghĩa là nó đã được sử dụng và không thể spend được nữa. Mảng spentTXOs lưu các output đã được spent bởi txId. Trường hợp coinbase transaction thì đặc biệt hơn vì nó không có input.

```
if tx.IsCoinbase() == false {
    for _, in := range tx.Vin {
        if in.CanUnlockOutputWith(address) {
            inTxID := hex.EncodeToString(in.Txid)
            spentTXOs[inTxID] = append(spentTXOs[inTxID], in.Vout)
        }
    }
}
```

Hàm trên sẽ trả lại một list các transaction có chưa UTXO. Để tính balance chúng ta cần thêm một hàm khác lọc ra các UTXO từ list trên:

```
func (bc *Blockchain) FindUTXO(address string) []TXOutput {
       var UTXOs []TXOutput
       unspentTransactions := bc.FindUnspentTransactions(address)

       for _, tx := range unspentTransactions {
                for _, out := range tx.Vout {
                        if out.CanBeUnlockedWith(address) {
                                UTXOs = append(UTXOs, out)
                        }
                }
       }

       return UTXOs
}
```

Giờ chúng ta đã có thể tạo hàm **getbalance**:

```
func (cli *CLI) getBalance(address string) {
	bc := NewBlockchain(address)
	defer bc.db.Close()

	balance := 0
	UTXOs := bc.FindUTXO(address)

	for _, out := range UTXOs {
		balance += out.Value
	}

	fmt.Printf("Balance of '%s': %d\n", address, balance)
}
```

Balance của một address chính là tổng của value từ các UTXO khoá bởi address đó.

Hãy kiểm tra balance sau khi đào genesis block:

```
$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 10
```

Đây là những đồng tiền đầu tiên trên blockchain.

Gửi tiền

Bây giờ chúng ta muốn gửi coins từ người này tới người kia. Vậy phải tạo ra transaction, đặt vào block và đào block đó. Chúng ta mới chỉ có coinbase transaction là loại transaction đặc biệt. Tổng quát lên một transaction được tạo như sau:

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain)
*Transaction {
	var inputs []TXInput
	var outputs []TXOutput

	acc, validOutputs := bc.FindSpendableOutputs(from, amount)

	if acc < amount {
		log.Panic("ERROR: Not enough funds")
	}

	// Build a list of inputs
	for txid, outs := range validOutputs {
		txID, err := hex.DecodeString(txid)

		for _, out := range outs {
			input := TXInput{txID, out, from}
			inputs = append(inputs, input)
		}
	}
```

```
        // Build a list of outputs
        outputs = append(outputs, TXOutput{amount, to})
        if acc > amount {
                outputs = append(outputs, TXOutput{acc - amount, from}) // a
change
        }

        tx := Transaction{nil, inputs, outputs}
        tx.SetID()

        return &tx
}
```

Trước khi tạo output mới, chúng ta phải tìm tất cả các UTXO và đảm bảo tổng giá trị phải đủ. Đây là công việc của hàm **FindSpendableOutputs**. Sau đó, với mỗi UTXO, một input mới sẽ được tạo ra. Tiếp theo chúng ta tạo ra 2 output:

1. Output chuyển tiền cho người nhận, sẽ được khoá bởi người nhận address
2. Ouput chuyển tiền dư lại cho người gửi, sẽ được khoá bởi người gửi address. Output này chỉ có khi tổng giá trị của các UTXO tìm được lớn hơn số tiền cần chuyển. Và hãy nhớ, các output không chia nhỏ ra được.

Hàm **FindSpendableOutputs** cũng giống như hàm **FindUnspentTransactions** chúng ta tạo trên kia:

```
func (bc *Blockchain) FindSpendableOutputs(address string, amount int) (int,
map[string][]int) {
        unspentOutputs := make(map[string][]int)
        unspentTXs := bc.FindUnspentTransactions(address)
        accumulated := 0

Work:
        for _, tx := range unspentTXs {
                txID := hex.EncodeToString(tx.ID)

                for outIdx, out := range tx.Vout {
                        if out.CanBeUnlockedWith(address) && accumulated <
amount {
                                accumulated += out.Value
                                unspentOutputs[txID] =
append(unspentOutputs[txID], outIdx)

                                if accumulated >= amount {
                                        break Work
                                }
                        }
                }
        }

        return accumulated, unspentOutputs
}
```

```

Hàm này quét qua tất cả các UTXO và tính value của chúng. Khi tổng value đã đủ thì hàm trả lại tổng giá trị kèm với các UTXOs theo các nhóm bởi transaction Id. Chúng ta cũng không cần quá nhiều hơn so với số tiền cần gửi.

Hàm đào block cũng được cập nhật lại:

```
func (bc *Blockchain) MineBlock(transactions []*Transaction) {
        ...
        newBlock := NewBlock(transactions, lastHash)
        ...
}
```

Và đây là hàm **send** gửi tiền:

```
func (cli *CLI) send(from, to string, amount int) {
        bc := NewBlockchain(from)
        defer bc.db.Close()

        tx := NewUTXOTransaction(from, to, amount, bc)
        bc.MineBlock([]*Transaction{tx})
        fmt.Println("Success!")
}
```

Gửi tiền là tiến trình tạo một transaction mới, cho vào block và cập nhật lên blockchain. Bitcoin thì không làm việc này ngay lập tức. Thay vào đó, Bitcoin cho tất cả các transaction mới vào bộ nhớ (mempool). Các miner sẽ lấy các transaction này cho vào block. Khi block được đào xong được gọi là confirmed và được thêm vào blockchain.

Hãy kiểm tra xem **sendcoin** đã hoạt động chưa:

```
$ blockchain_go send -from Ivan -to Pedro -amount 6
00000001b56d60f86f72ab2a59fadb197d767b97d4873732be505e0a65cc1e37

Success!

$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 4

$ blockchain_go getbalance -address Pedro
Balance of 'Pedro': 6
```

Hãy tạo thêm nhiều transaction và kiểm tra xem các unspent outputs có hoạt động không:

```
$ blockchain_go send -from Pedro -to Helen -amount 2
00000099938725eb2c7730844b3cd40209d46bce2c2af9d87c2b7611fe9d5bdf
```
```

```
Success!

$ blockchain_go send -from Ivan -to Helen -amount 2
000000a2edf94334b1d94f98d22d7e4c973261660397dc7340464f7959a7a9aa

Success!
```

Helen có tiền được chuyển từ Pedro và Ivan, giờ gửi chúng tới người khác:

```
$ blockchain_go send -from Helen -to Rachel -amount 3
000000c58136cffa669e767b8f881d16e2ede3974d71df43058baaf8c069f1a0

Success!

$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 2

$ blockchain_go getbalance -address Pedro
Balance of 'Pedro': 4

$ blockchain_go getbalance -address Helen
Balance of 'Helen': 1

$ blockchain_go getbalance -address Rachel
Balance of 'Rachel': 3
```

Giờ hãy test một trường hợp gửi sai:

```
$ blockchain_go send -from Pedro -to Ivan -amount 5
panic: ERROR: Not enough funds

$ blockchain_go getbalance -address Pedro
Balance of 'Pedro': 4

$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 2

```

Kết luận

Chúng ta đã thêm được transaction vào blockchain. Tuy nhiên còn thiếu nhiều tính năng cơ bản trong Bitcoin như sau:

1. Address. Chúng ta chưa có address thật dựa vào private key
2. Rewards. Chưa thưởng khi mine block.
3. UTXO set. Hiện tại muốn kiểm tra balance của một địa chỉ cần quét qua toàn bộ blockchain, việc này rất chậm và tốn nhiều tính toán. UTXO set sẽ giúp việc này dễ dàng hơn nhiều.
4. Mempool. Là nơi tập hợp các transaction chưa được cho vào block. Trong cài đặt của chúng ta mỗi block chỉ có 1 transaction nên chưa cần thiết

Links:

1. [Full source codes](https://github.com/Jeiwan/blockchain_go/tree/part_4)
2. [Transaction](https://en.bitcoin.it/wiki/Transaction)
3. [Merkle tree](https://en.bitcoin.it/wiki/Protocol_documentation#Merkle_Trees)
4. [Coinbase](https://en.bitcoin.it/wiki/Coinbase)

Mục lục

Lập trình Blockchain với Golang. Part 5: Address

>Bài dịch từ _Building Blockchain in Go_ của tác giả _Ivan Kuznetsov_. Khi sử dụng vui lòng trích dẫn nguồn
[@hlongvu](https://github.com/hlongvu/blockchain-go-vietnamese)

Mục lục

Giới thiệu

Trong phần trước chúng ta đã xây dựng được transaction. Các bạn cũng đã được giới thiệu về tính ẩn danh của transaction: không có user account, thông tin cá nhân không hề được lưu trữ trên hệ thống Bitcoin. Nhưng vậy phải có thứ gì đó chứng minh bạn là chủ sở hữu của những transaction chưa sử dụng này. Đây là nhiệm vụ mà address trong Bitcoin đảm nhiệm. Ở phần code trước chúng ta đã dùng string làm username để làm khoá trong các UTXO. Bây giờ chúng ta sẽ tiến hành xây dựng các address thật, tương tự như trong Bitcoin.

> Hàm lượng code thay đổi là rất lớn nên các bạn có thể theo dõi tất cả ở [đây](https://github.com/Jeiwan/blockchain_go/compare/part_4...part_5#files_bucket)

Bitcoin Address

Đây là một ví dụ của Bitcoin address: [1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa](https://blockchain.info/address/1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa). Đây là địa chỉ đầu tiên của Bitcoin, thuộc về Satoshi Nakamoto. Địa chỉ Bitcoin luôn luôn công khai. Nếu bạn muốn gửi coins cho bất kì ai, bạn phải biết địa chỉ của người đó. Nhưng việc có địa chỉ không chứng minh bạn là chủ của "ví" (wallet) đó. Address thực ra là thể hiện string của public key. Trong Bitcoin, định danh là một cặp private, public key lưu trong máy tính (hoặc một nơi nào đó bạn có quyền truy cập). Bitcoin dựa vào các tổ hợp của các thuật toán mã hoá để tạo nên các keys này, và đảm

bảo rằng không ai khác có thể sử dụng coin của bạn mà khi không biết keys của bạn. Hãy xem các thuật toán đó là gì.

Public-key Cryptography
Public-key Cryptography (mã hoá công khai) sử dụng một cặp keys: public key và private key. Public key thì không cần bảo mật và có thể cho bất kì ai biết. Ngược lại, private key thì cần phải bảo mật, không ai ngoài chủ sở hữu được biết vì private key có tác dụng như chìa khoá.

Về bản chất, ví Bitcoin là một cặp key đó. Khi bạn sử dụng một app wallet hoặc dùng một Bitcoin client để tạo address, một cặp key sẽ được tạo ra cho bạn. Bất kì ai nắm private key đều có quyền sử dụng tất cả coins được gửi tới key này trong Bitcoin.

Private key và Public key đều chỉ là các mảng bytes ngẫu nhiên, nên không thể in ra trên màn hình một cách dễ đọc. Vì thế Bitcoin sử dụng một thuật toán để chuyển đổi public key sang dạng string dễ đọc hơn.

> Nếu bạn từng sử dụng Bitcoin wallet, bạn có thể thấy chương trình thường đưa cho bạn một chuỗi các pass phrase (gồm 12 hoặc 24 từ tiếng Anh). Pass phrase đó được dùng thay cho private key và có thể tạo ra private key. Cơ chế này được xây dựng ở [BIP-039](https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki).

> BIP: Bitcoin Improvement Proposal

Chúng ta biết key định danh user trong Bitcoin. Vậy làm cách nào Bitcoin kiểm tra sở hữu của các outputs chưa sử dụng UTXO?

Digital Signature - Chữ Ký Số

Trong toán học và mã hoá, có một khái niệm về chữ ký số (digital signature) có thể đảm bảo các tính chất sau:

1. Đảm bảo dữ liệu không bị sửa chữa khi được gửi từ người này tới người kia
2. Xác minh được dữ liệu được tạo bởi 1 người xác định
3. Đảm bảo được việc một người gửi dữ liệu đi không phủ nhận được đó không phải của mình

Bằng cách áp dụng chữ ký số lên dữ liệu (Signing), chúng ta được một chữ ký, mà sau đó có thể xác thực được. Signing bằng cách sử dụng private key, và xác minh chỉ cần public key.

Để sign dữ liệu chúng ta cần:

1. Dữ liệu để sign
2. Private key

Để xác minh một chữ ký (được lưu trong các transaction input), chúng ta cần:

1. Dữ liệu đã được sign
2. Chữ ký (signature)
3. Public key

Nói ngắn gọn thì việc xác minh chữ ký là: kiểm tra xem chữ ký này có đúng với dữ liệu và private key tương ứng với public key này không.

> Chữ ký số không phải là mã hoá dữ liệu, chúng ta không thể tái tạo lại dữ liệu ban đầu từ chữ ký. Tương tự như hàm băm, dữ liệu chạy qua một hàm băm để cho ra một biểu diễn độc nhất tương ứng. Sự khác nhau giữa chữ ký và hash đó là cặp keys, chúng giúp việc xác minh chữ ký mà không lộ private key.
>
> Cặp keys thực ra cũng có thể dùng để mã hoá dữ liệu. Public key dùng để mã hoá và private key dùng để giải mã. Nhưng Bitcoin không sử dụng tính năng này.

Các transaction input trong Bitcoin được sign bởi người tạo transaction. Tất cả các transaction đều phải được xác minh trước khi cho vào block. Xác minh bao gồm:

1. Kiểm tra các input có quyền sử dụng các output trong các transactions nó trỏ tới.
2. Kiểm tra transaction signature là đúng

Quá trình này về cơ bản như sau:

Bây giờ hãy xem lại vòng đời của một transaction:

1. Lúc khởi tạo, blockchain chỉ có một genesis block chứa coinbase transaction. Trong coinbase transaction này không có input nên không cần việc signing. Output của coinbase transaction chứa hash của một public key. (RIPEMD16(SHA256(PubKey) thuật toán này được sử dụng để tính hash của public key).

2. Khi một người gửi coin, một transaction được khởi tạo. Các input của transaction này phải trỏ tới các outputs của transaction đã có. Mỗi input sẽ chứa một public key (không hash) và một chữ ký của cả transaction.

3. Những người khác trong mạng Bitcoin (node) khi nhận được transaction trên sẽ phải xác minh nó. Nó phải 1. kiểm tra hash của public key trong input giống với hash của output mà nó trỏ tới; 2. kiểm tra chữ ký của transaction.

4. Khi miner sẵn sàng mine block mới, nó sẽ cho transaction này vào block và khởi động việc mining.

5. Khi block được mine xong, tất cả các node khác trong mạng sẽ nhận được message thông báo có block mới và thêm block này vào blockchain.

6. Sau khi block được thêm vào blockchain, transaction này đã hoàn thành. Các outputs trong transaction này đã có thể được dùng trong transaction mới trong tương lai.

Elliptic Curve Cryptography

Như đã đề cập ở trên, public và private key là các chuỗi bytes ngẫu nhiên. Bởi private key được dùng để quyết định quyền sở hữu coin nên có một điều kiện phải thoả mãn: thuật toán random để sinh ra key phải thật chuẩn xác.

Chúng ta hoàn toàn không muốn ngẫu nhiên tạo ra một private key trùng với một người khác.

Bitcoin sử dụng mã hoá Elliptic Curve để tạo private key. Elliptic Curve là một mô hình toán học phức tạp, chúng ta không giải thích ở đây (nếu bạn tò mò muốn biết có thể xem tại [đây](http://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/)). Bạn chỉ cần biết là mô hình toán học này có thể dùng để tạo ra các số rất lớn và ngẫu nhiên. Ở Bitcoin có thể tạo ra số trong khoảng 0 đến 2^{256} (gần bằng 10^{77}, trong khi đó chỉ có 10^{78} đến 10^{82} nguyên tử trong vũ trụ). Đó là một con số rất lớn có thể đảm bảo không bao giờ random ra hai private key giống nhau.

Ngoài ra, Bitcoin dùng ECDSA (Elliptic Curve Digital Signature Algorithm) để sign các transaction. Chúng ta cũng sẽ sử dụng thuật toán này.

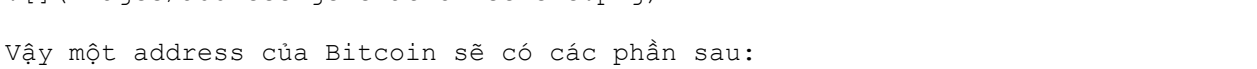
Base58

Hãy quay lại ví dụ về địa chỉ Bitcoin trước: 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa. Đây là dạng string đơn giản có thể đọc được của public key. Nếu giải mã nó thì chúng ta sẽ được chuỗi sau (mảng byte viết dưới hệ 16):

```
0062E907B15CBF27D5425399EBF6F0FB50EBB88F18C29B7D93
```

Bitcoin sử dụng mã hoá Base58 để chuyển public key sang dạng dễ đọc (address). Thuật toán này rất giống với Base64, nhưng bỏ bớt một số chữ cái nhằm tránh việc đọc nhầm lẫn dễ gây lỗi. Do đó, Base58 không có các chữ sau: 0 (zero), O (capital o), I (capital i), l (lowercase L) . Ngoài ra không có + và /.

Dưới đây là tiến trình chuyển public key sang address:



Vậy một address của Bitcoin sẽ có các phần sau:

```
Version  Public key hash                             Checksum
00       62E907B15CBF27D5425399EBF6F0FB50EBB88F18  C29B7D93
```

Thuật toán hash là một chiều, chúng ta không thể lấy lại public key chỉ dựa vào hash. Nhưng chúng ta có thể kiểm tra một public key có tạo ra hash đúng thế không (bằng cách chạy lại hàm trên và so sánh).

Đi sâu vào code chúng ta sẽ thấy các khái niệm trên dễ hiểu hơn.

Cài đặt Addresses

Chúng ta bắt đầu với Wallet:

```
type Wallet struct {
        PrivateKey ecdsa.PrivateKey
```

```
        PublicKey  []byte
}

type Wallets struct {
        Wallets map[string]*Wallet
}

func NewWallet() *Wallet {
        private, public := newKeyPair()
        wallet := Wallet{private, public}

        return &wallet
}

func newKeyPair() (ecdsa.PrivateKey, []byte) {
        curve := elliptic.P256()
        private, err := ecdsa.GenerateKey(curve, rand.Reader)
        pubKey := append(private.PublicKey.X.Bytes(),
private.PublicKey.Y.Bytes()...)

        return *private, pubKey
}
```

Một ví (wallet) không có gì hơn ngoài một cặp key. Vậy nên chúng ta cần thêm **Wallets**, gồm một list các wallet, lưu lại được vào file và đọc lại. Hàm **newKeyPair** được dùng để tạo một cặp key mới: Chúng ta khởi tạo một ECDSA và tạo một private key, public key được tạo ra từ private key này. Chú ý: Trong thuật toán eliptic curve, public key là các điểm trên đường cong đồ thị. Do đó public key là tổ hợp của các toạ độ X, Y. Trong Bitcoin, các toạ độ này được ghép lại với nhau tạo thành public key.

Bây giờ hãy tạo address:

```
func (w Wallet) GetAddress() []byte {
        pubKeyHash := HashPubKey(w.PublicKey)

        versionedPayload := append([]byte{version}, pubKeyHash...)
        checksum := checksum(versionedPayload)

        fullPayload := append(versionedPayload, checksum...)
        address := Base58Encode(fullPayload)

        return address
}

func HashPubKey(pubKey []byte) []byte {
        publicSHA256 := sha256.Sum256(pubKey)

        RIPEMD160Hasher := ripemd160.New()
        _, err := RIPEMD160Hasher.Write(publicSHA256[:])
        publicRIPEMD160 := RIPEMD160Hasher.Sum(nil)

        return publicRIPEMD160
}
```

```
func checksum(payload []byte) []byte {
        firstSHA := sha256.Sum256(payload)
        secondSHA := sha256.Sum256(firstSHA[:])

        return secondSHA[:addressChecksumLen]
}
```

Các bước để tạo address từ public key như sau:

1. Lấy public key và hash nó với thuật toán **RIPEMD160(SHA256(PubKey))**
2. Thêm các byte version vào đầu hash trên
3. Tính checksum bằng cách hash tiếp **SHA256(SHA256(payload))**. Lấy 4 bytes từ hash này.
4. Ghép checksum vào hash ở bước 2 **version+PubKeyHash**
5. Mã hoá lại **version+PubKeyHash+checksum** theo Base58

Kết quả là chúng ta sẽ được một **địa chỉ Bitcoin thật**, bạn có thể kiểm tra balance của địa chỉ này tại blockchain.info. Tôi có thể dám chắc balance này luôn bằng 0 cho dù bạn thử bao nhiêu lần. Đây là lí do tại sao chúng ta cần chọn một thuật toán mã hoá tốt: Thử hình dung việc sinh private key, phải đảm bảo cơ hội sinh ra trùng private key là cực kì thấp. Một cách lí tưởng thì phải là "không bao giờ".

Cũng chú ý là bạn không cần phải kết nối tới một máy tính chạy mạng Bitcoin để tạo được một address. Thuật toán tạo address này là chung và công khai, không phụ thuộc vào tính và ngôn ngữ sử dụng.

Chúng ta phải cập nhật lại input và output trong transaction để sử dụng address:

```
type TXInput struct {
        Txid      []byte
        Vout      int
        Signature []byte
        PubKey    []byte
}

func (in *TXInput) UsesKey(pubKeyHash []byte) bool {
        lockingHash := HashPubKey(in.PubKey)

        return bytes.Compare(lockingHash, pubKeyHash) == 0
}

type TXOutput struct {
        Value      int
        PubKeyHash []byte
}

func (out *TXOutput) Lock(address []byte) {
        pubKeyHash := Base58Decode(address)
        pubKeyHash = pubKeyHash[1 : len(pubKeyHash)-4]
        out.PubKeyHash = pubKeyHash
}
```

```
func (out *TXOutput) IsLockedWithKey(pubKeyHash []byte) bool {
        return bytes.Compare(out.PubKeyHash, pubKeyHash) == 0
}
```

Chú ý là chúng ta không còn sử dụng **ScriptPubKey** và **ScriptSig** vì chúng ta sẽ không xây dựng một ngôn ngữ Script như Bitcoin. Thay vào đó **ScriptSig** chia thành **Signature** và **PubKey** còn **ScriptPubKey** đổi tên thành **PubKeyHash**. Chúng ta sẽ xây dựng cơ chế khoá, mở khoá và chữ ký inputs như ở Bitcoin, nhưng đặt trong các hàm thay vì Script.

Hàm **UsesKey** kiểm tra xem input có sử dụng được key để mở khoá được không. Chú ý là trong input là public key chưa hash, còn hàm sử dụng một key đã hash. **IsLockedWithKey** kiểm tra xem key có dùng để mở khoá output này được không. Hàm này bổ sung cho **UsesKey** và sẽ được dùng trong **FindUnspentTransactions** để kiểm tra liên kết giữa các transaction.

Hàm **Lock** dùng để khoá output. Khi chúng ta gửi coin cho ai đó, chúng ta chỉ biết địa chỉ của họ. Hàm này nhận tham số là địa chỉ đó. Địa chỉ được giải mã để lấy public key hash và lưu lại trong trường **out.PubKeyHash**.

Giờ hãy kiểm tra xem mọi thứ có hoạt động không:

```
$ blockchain_go createwallet
Your new address: 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt

$ blockchain_go createwallet
Your new address: 15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h

$ blockchain_go createwallet
Your new address: 1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy

$ blockchain_go createblockchain -address 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt
0000005420fbfdafa00c093f56e033903ba43599fa7cd9df40458e373eee724d

Done!

$ blockchain_go getbalance -address 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt
Balance of '13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt': 10

$ blockchain_go send -from 15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h -to
13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt -amount 5
2017/09/12 13:08:56 ERROR: Not enough funds

$ blockchain_go send -from 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt -to
15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h -amount 6
00000019afa909094193f64ca06e9039849709f5948fbac56cae7b1b8f0ff162

Success!

$ blockchain_go getbalance -address 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt
Balance of '13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt': 4

$ blockchain_go getbalance -address 15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h
Balance of '15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h': 6
```

```
$ blockchain_go getbalance -address 1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy
Balance of '1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy': 0
```

Tiếp theo chúng ta sẽ cài đặt phần chữ ký transaction.

Cài Đặt Signatures

Transaction phải được sign vì đây là cách duy nhất Bitcoin đảm bảo việc một người không thể dùng coin của người khác. Nếu chữ ký (signature) là sai, cả transaction cũng được coi là vô hiệu và không được cho vào blockchain.

Chúng ta đã có các thành phần để tạo signature cho transaction, trừ phần dữ liệu. Phần nào của transaction cần được sign? Hay phải sign toàn bộ transaction? Chọn lựa dữ liệu để sign cũng khá quan trọng. Vấn đề là phải chọn lựa dữ liệu chứa các thông tin đủ và cần thiết. Ví dụ, sẽ vô lí khi chỉ sign các output vì chưa đủ để bảo vệ người gửi và người nhận.

Các transaction phải mở khoá các output, tính toán lượng coin cần gửi, khoá lại trong output mới. Những dữ liệu cần sign bao gồm:

1. Public key của các output cần mở khoá. Đây là phần thể hiện "người gửi" của transaction
2. Public key hash của output bị khoá. Đây là phần thể hiện "người nhận" của transaction
3. Value của output mới (giá trị coin sẽ gửi đi)

> Trong Bitcoin, logic khoá và mở khoá được lưu trong các đoạn mã, nằm ở **ScriptSig** và **ScriptPubKey**. Bitcoin sẽ sign cả đoạn mã **ScriptPubKey**.

Như bạn thấy, chúng ta không cần sign các public key trong các input. Vì thế nên trong Bitcoin, không phải transaction được sign mà chỉ là một phần của nó với input chứa **ScriptPubKey**.

> Chi tiết của tiến trình lấy một phần transaction để sign được mô tả tại [đây](https://en.bitcoin.it/wiki/File:Bitcoin_OpCheckSig_InDetail.png). Có thể nó là bản cũ, nhưng tôi chưa tìm được một mô tả nào khác tin cậy hơn.

Có vẻ khá loằng ngoằng, nhưng hãy bắt đầu code hàm **Sign**:

```
func (tx *Transaction) Sign(privKey ecdsa.PrivateKey, prevTXs
map[string]Transaction) {
	if tx.IsCoinbase() {
		return
	}

	txCopy := tx.TrimmedCopy()

	for inID, vin := range txCopy.Vin {
		prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
		txCopy.Vin[inID].Signature = nil
		txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
		txCopy.ID = txCopy.Hash()
		txCopy.Vin[inID].PubKey = nil
```

```
                r, s, err := ecdsa.Sign(rand.Reader, &privKey, txCopy.ID)
                signature := append(r.Bytes(), s.Bytes()...)

                tx.Vin[inID].Signature = signature
        }
}
```

Hàm này lấy private key và một map của các transaction trước. Như đã nhắc ở trước, để sign một transaction chúng ta cần các output được trỏ tới bởi các input của transaction này, cho nên chúng ta cần các transaction lưu giữ các output này (prevTXs).

Hãy xem lại các bước:

```
if tx.IsCoinbase() {
        return
}
```

Coinbase transaction không cần sign vì nó không có input.

```
txCopy := tx.TrimmedCopy()
```

Một phần của transaction được sao chép lại để sign, không phải toàn bộ:

```
func (tx *Transaction) TrimmedCopy() Transaction {
        var inputs []TXInput
        var outputs []TXOutput

        for _, vin := range tx.Vin {
                inputs = append(inputs, TXInput{vin.Txid, vin.Vout, nil, nil})
        }

        for _, vout := range tx.Vout {
                outputs = append(outputs, TXOutput{vout.Value,
vout.PubKeyHash})
        }

        txCopy := Transaction{tx.ID, inputs, outputs}

        return txCopy
}
```

Phiên bản sao chép này chứa toàn bộ input và output nhưng
TXInput.Signature và **TXInput.PubKey**
được chuyển thành nil.

Sau đó chúng ta quét qua mảng input:

```
for inID, vin := range txCopy.Vin {
        prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
        txCopy.Vin[inID].Signature = nil
        txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash

```

Tại mỗi input, **Signature** được đặt lại nil (double-check) and **PubKey** được đặt thành **PubKeyHash** của output được trỏ tới. **Các input được sign riêng rẽ nhau**, blockchain của chúng ta không cần điều này, tuy nhiên ở Bitcoin thì các input có thể trỏ tới các address khác nhau.

```
txCopy.ID = txCopy.Hash()
txCopy.Vin[inID].PubKey = nil

```

Hàm **Hash** serialize transaction và lấy hash theo thuật toán SHA-256. Chúng ta sẽ lấy hash này để sign. Sau khi lấy hash chúng ta trả lại giá trị cho **PubKey** để không ảnh hưởng tới vòng lặp khác.

Đây là phần chính:

```
r, s, err := ecdsa.Sign(rand.Reader, &privKey, txCopy.ID)
signature := append(r.Bytes(), s.Bytes()...)
tx.Vin[inID].Signature = signature

```
Chúng ta sign **txCopy.ID** bằng **privKey**. Một chữ ký ECDSA là một cặp số, được gộp lại và lưu vào   **input.Signature**.

Tiếp theo là hàm để xác minh signature:

```
func (tx *Transaction) Verify(prevTXs map[string]Transaction) bool {
        txCopy := tx.TrimmedCopy()
        curve := elliptic.P256()

        for inID, vin := range tx.Vin {
                prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
                txCopy.Vin[inID].Signature = nil
                txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
                txCopy.ID = txCopy.Hash()
                txCopy.Vin[inID].PubKey = nil

                r := big.Int{}
                s := big.Int{}
                sigLen := len(vin.Signature)
                r.SetBytes(vin.Signature[:(sigLen / 2)])
                s.SetBytes(vin.Signature[(sigLen / 2):])

                x := big.Int{}
                y := big.Int{}
```

```
                keyLen := len(vin.PubKey)
                x.SetBytes(vin.PubKey[:(keyLen / 2)])
                y.SetBytes(vin.PubKey[(keyLen / 2):])

                rawPubKey := ecdsa.PublicKey{curve, &x, &y}
                if ecdsa.Verify(&rawPubKey, txCopy.ID, &r, &s) == false {
                        return false
                }
        }

        return true
}
```

Tương tự như phần sign, chúng ta cần sao chép một phần của transaction:

```
txCopy := tx.TrimmedCopy()
```

Khởi tạo thuật toán mã hoá:

```
curve := elliptic.P256()
```

Sau đó chúng ta kiểm tra signature ở mỗi input:

```
for inID, vin := range tx.Vin {
        prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
        txCopy.Vin[inID].Signature = nil
        txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
        txCopy.ID = txCopy.Hash()
        txCopy.Vin[inID].PubKey = nil
```

Phần này tương tự với hàm **Sign**, xác minh cũng cần dữ liệu giống với lúc tạo chữ ký:

```
r := big.Int{}
        s := big.Int{}
        sigLen := len(vin.Signature)
        r.SetBytes(vin.Signature[:(sigLen / 2)])
        s.SetBytes(vin.Signature[(sigLen / 2):])

        x := big.Int{}
        y := big.Int{}
        keyLen := len(vin.PubKey)
        x.SetBytes(vin.PubKey[:(keyLen / 2)])
        y.SetBytes(vin.PubKey[(keyLen / 2):])
```

Ở đây chúng ta tách **TXInput.Signature** và **TXInput.PubKey** thành hai phần tương ứng với 2 toạ độ của public key để sử dụng theo thư viện **crypto/ecdsa**.

```
        rawPubKey := ecdsa.PublicKey{curve, &x, &y}
        if ecdsa.Verify(&rawPubKey, txCopy.ID, &r, &s) == false {
                return false
        }
}

return true
```

Chúng ta khởi tạo lại **ecdsa.PublicKey** và chạy hàm **ecdsa.Verify**. Nếu tất cả các input đều đúng sẽ trả lại true, ngược lại có 1 input sai thì sẽ trả về false.

Bây giờ chúng ta cần có thêm hàm để tìm kiếm các transaction trước (tương ứng với các UTXO). Hàm này sẽ yêu cầu tham chiếu tới blockchain, vì thế nó được đặt làm hàm con của Blockchain:

```
func (bc *Blockchain) FindTransaction(ID []byte) (Transaction, error) {
        bci := bc.Iterator()

        for {
                block := bci.Next()

                for _, tx := range block.Transactions {
                        if bytes.Compare(tx.ID, ID) == 0 {
                                return *tx, nil
                        }
                }

                if len(block.PrevBlockHash) == 0 {
                        break
                }
        }

        return Transaction{}, errors.New("Transaction is not found")
}

func (bc *Blockchain) SignTransaction(tx *Transaction, privKey
ecdsa.PrivateKey) {
        prevTXs := make(map[string]Transaction)

        for _, vin := range tx.Vin {
                prevTX, err := bc.FindTransaction(vin.Txid)
                prevTXs[hex.EncodeToString(prevTX.ID)] = prevTX
        }

        tx.Sign(privKey, prevTXs)
}

func (bc *Blockchain) VerifyTransaction(tx *Transaction) bool {
```

```
        prevTXs := make(map[string]Transaction)

        for _, vin := range tx.Vin {
                prevTX, err := bc.FindTransaction(vin.Txid)
                prevTXs[hex.EncodeToString(prevTX.ID)] = prevTX
        }

        return tx.Verify(prevTXs)
}
```

Nội dung của các hàm này khá đơn giản: **FindTransaction** tìm kiếm các transaction theo ID; **SignTransaction** lấy một transaction, tìm kiếm các transaction liên quan và sign; **VerifyTransaction** tương tự, nhưng là xác minh.

Việc sign các transaction diễn ra ở **NewUTXOTransaction**:

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain)
*Transaction {
        ...

        tx := Transaction{nil, inputs, outputs}
        tx.ID = tx.Hash()
        bc.SignTransaction(&tx, wallet.PrivateKey)

        return &tx
}
```

Còn xác minh các transaction diễn ra trước khi các transaction được cho vào block:

```
func (bc *Blockchain) MineBlock(transactions []*Transaction) {
        var lastHash []byte

        for _, tx := range transactions {
                if bc.VerifyTransaction(tx) != true {
                        log.Panic("ERROR: Invalid transaction")
                }
        }
        ...
}
```

Xong, hãy kiểm tra lại chương trình một lần nữa:

```
$ blockchain_go createwallet
Your new address: 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR

$ blockchain_go createwallet
Your new address: 1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab

$ blockchain_go createblockchain -address 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR
```

```
000000122348da06c19e5c513710340f4c307d884385da948a205655c6a9d008

Done!

$ blockchain_go send -from 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR -to
1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab -amount 6
0000000f3dbb0ab6d56c4e4b9f7479afe8d5a5dad4d2a8823345a1a16cf3347b

Success!

$ blockchain_go getbalance -address 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR
Balance of '1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR': 4

$ blockchain_go getbalance -address 1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab
Balance of '1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab': 6
```

Hãy comment lại phần **bc.SignTransaction(&tx, wallet.PrivateKey)** để xem các transaction chưa sign có được sign hay không:

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain)
*Transaction {
   ...
        tx := Transaction{nil, inputs, outputs}
        tx.ID = tx.Hash()
        // bc.SignTransaction(&tx, wallet.PrivateKey)

        return &tx
}
```

```
$ go install
$ blockchain_go send -from 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR -to
1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab -amount 1
2017/09/12 16:28:15 ERROR: Invalid transaction
```

Awesome!

Kết luận

Chúng ta đã cài đặt được khá nhiều chức năng của Bitcoin lên blockchain này, trừ các phần liên quan tới network. Trong phần tới chúng ta sẽ hoàn thiện các tính năng của transaction.

Links

1. [Full source codes](https://github.com/Jeiwan/blockchain_go/tree/part_5)
2. [Public-key cryptography](https://en.wikipedia.org/wiki/Public-key_cryptography)
3. [Digital signatures](https://en.wikipedia.org/wiki/Digital_signature)
4. [Elliptic curve](https://en.wikipedia.org/wiki/Elliptic_curve)
5. [Elliptic curve cryptography](https://en.wikipedia.org/wiki/Elliptic_curve_cryptography)

6.
[ECDSA](https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm)
7. [Technical background of Bitcoin addresses](https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses)
8. [Address](https://en.bitcoin.it/wiki/Address)
9. [Base58](https://en.bitcoin.it/wiki/Base58Check_encoding)
10. [A gentle introduction to elliptic curve cryptography](http://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/)

Mục lục

Lập trình Blockchain với Golang. Part 6: Transaction 2

>Bài dịch từ _Building Blockchain in Go_ của tác giả _Ivan Kuznetsov_. Khi sử dụng vui lòng trích dẫn nguồn
[@hlongvu](https://github.com/hlongvu/blockchain-go-vietnamese)

Mục lục

Giới thiệu

Trong những phần trước, chúng ta đã nói về blockchain là một bộ cơ sở dữ liệu phân tán, tuy nhiên chỉ mới xây dựng phần dữ liệu chứ chưa có phân tán. Phần này chúng ta sẽ tiếp tục bổ sung các cơ chế của transaction còn thiếu, tiếp tục trong phần sau sẽ đi vào tính phân tán của blockchain.

> Hàm lượng code thay đổi là rất lớn nên các bạn có thể theo dõi tất cả ở [đây](https://github.com/Jeiwan/blockchain_go/compare/part_5...part_6#files_bucket)

Reward - Phần thưởng

Một tính năng nhỏ chúng ta bỏ qua trong phần trước đó là reward cho việc mining. Tới lúc này thì chúng ta đã có đủ các yếu tố để cài đặt nó.

Reward đơn giản chỉ là một coinbase transaction. Khi một node tham gia đào block mới, nó gom các transaction đang chờ và thêm vào 1 coinbase transaction. Coinbase transaction này chỉ có một output chứa public key hash của người đào (- tự thưởng cho mình).

Cài đặt reward khá dễ bằng cách cập nhật hàm send:

```
```

```
func (cli *CLI) send(from, to string, amount int) {
    ...
    bc := NewBlockchain()
    UTXOSet := UTXOSet{bc}
    defer bc.db.Close()

    tx := NewUTXOTransaction(from, to, amount, &UTXOSet)
    cbTx := NewCoinbaseTX(from, "")
    txs := []*Transaction{cbTx, tx}

    newBlock := bc.MineBlock(txs)
    fmt.Println("Success!")
}
```

Trong blockchain của chúng ta, người tạo transaction cũng đào luôn block và nhận thưởng.

UTXO Set

Trong [phần 3]() chúng ta đã biết cách Bitcoin Core lưu trữ block trong database. Các block được lưu trong tập dữ liệu **blocks** và các transaction output được lưu trong **chainstate**. Hãy xem lại lần nữa:

1. 'c' + 32-byte transaction hash -> unspent transaction output record for that transaction (*Lưu các unspent transaction output cho transaction id đó*)
2. 'B' -> 32-byte block hash: the block hash up to which the database represents the unspent transaction outputs (block cuối)

Blockchain của chúng ta đã có bucket **blocks** lưu các block tuy nhiên chưa có **chainstate**. Giờ chúng ta sẽ xây dựng phần này.

chainstate không lưu giữ các transaction. Thay vào đó nó lưu trữ UTXO Set, là tập hợp của các transaction output chưa sử dụng (unspent transaction output). Ngoài ra khoá 'B' như trong Bitcoin chúng ta cũng chưa cần cài đặt vì blockchain của chúng ta chưa có tham số block-height (độ cao block).

Vậy tại sao lại cần UTXO Set?

Hãy xem lại hàm **Blockchain.FindUnspentTransactions** đã cài đặt:

```
func (bc *Blockchain) FindUnspentTransactions(pubKeyHash []byte)
[]Transaction {
    ...
    bci := bc.Iterator()

    for {
        block := bci.Next()

        for _, tx := range block.Transactions {
            ...
        }

        if len(block.PrevBlockHash) == 0 {
            break
        }
```

```
    }
    ...
}
```

Hàm này tìm kiếm các utxo. Tuy nhiên do transaction được lưu trong block, chúng ta phải quét qua toàn bộ blockchain và kiểm tra mọi transaction trong nó. Tính tới 18/9/2017, Bitcoin có tới 485,860 block và toàn bộ dữ liệu lớn tới 140+ Gb. Như vậy khối lượng tính toán là rất lớn.

Giải pháp cho vấn đề này là lưu trữ riêng lại các utxo, và đây chính là việc mà UTXO Set đảm nhiệm. Nó là bộ nhớ đệm xây dựng từ blockchain (phải quét lại từ đầu, nhưng chỉ phải làm 1 lần), sau đó được dùng để tính toán balance và xác minh transaction mới. UTXO Set này rơi vào khoảng 2.7Gb vào 9/2017.

Hãy xem chúng ta cần thay đổi gì để cài đặt UTXO Set này. Hiện tại các hàm này được sử dụng để tìm kiếm các transaction:

1. **Blockchain.FindUnspentTransactions** - hàm chính để tìm các transaction chứa utxo. Nó quét qua toàn bộ block.
2. **Blockchain.FindSpendableOutputs** - hàm này sử dụng khi một transaction mới được tạo. Nó sẽ tìm đủ utxo phù hợp với số coin cần gửi.
3. **Blockchain.FindUTXO** - tìm kiếm utxo cho một public key hash, dùng để tính balance.
4. **Blockchain.FindTransaction** - tìm kiếm các transaction trong blockchain bằng id. Nó quét qua tất cả block cho tới lúc tìm thấy.

Như chúng ta thấy các hàm trên đều phải quét qua toàn bộ blockchain. Bộ UTXO Set chỉ lưu các utxo nên sẽ không dùng được cho hàm **Blockchain.FindTransaction**

Vậy chúng ta cần các hàm sau:

1. **Blockchain.FindUTXO** – Tìm kiếm các utxo bằng cách quét qua block
2. **UTXOSet.Reindex** — dùng **FindUTXO** để tìm các utxo và lưu vào database. Đây là caching.
3. **UTXOSet.FindSpendableOutputs** – tương tự như **Blockchain.FindSpendableOutputs**, nhưng sử dụng UTXO Set.
4. **UTXOSet.FindUTXO** – tương tự như **Blockchain.FindUTXO**, nhưng dùng UTXO Set.
5. **Blockchain.FindTransaction** không đổi.

Như vậy các hàm dùng nhiều sẽ sử dụng qua cache này. Hãy bắtd đầu cài đặt:

```
type UTXOSet struct {
    Blockchain *Blockchain
}
```

Chúng ta sử dụng cùng một file database, tuy nhiên sẽ lưu UTXO Set vào một bucket khác. UTXOSet sẽ truy cập chung DB với Blockchain.

```
func (u UTXOSet) Reindex() {
    db := u.Blockchain.db
    bucketName := []byte(utxoBucket)
```

```
    err := db.Update(func(tx *bolt.Tx) error {
        err := tx.DeleteBucket(bucketName)
        _, err = tx.CreateBucket(bucketName)
    })

    UTXO := u.Blockchain.FindUTXO()

    err = db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket(bucketName)

        for txID, outs := range UTXO {
            key, err := hex.DecodeString(txID)
            err = b.Put(key, outs.Serialize())
        }
    })
}
```

Đây là hàm khởi tạo UTXO Set. Đầu tiên sẽ xoá bucket nếu đã có, sau đó sẽ quét qua toàn bộ Blockchain để tìm utxo và lưu lại trong database.

Blockchain.FindUTXO gần tương tự với **Blockchain.FindUnspentTransactions**, nhưng giờ sẽ trả về một map của cặp **TransactionID** → **TransactionOutputs**.

Bây giờ UTXO Set có thể dùng để send coin:

```
func (u UTXOSet) FindSpendableOutputs(pubkeyHash []byte, amount int) (int,
map[string][]int) {
    unspentOutputs := make(map[string][]int)
    accumulated := 0
    db := u.Blockchain.db

    err := db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(utxoBucket))
        c := b.Cursor()

        for k, v := c.First(); k != nil; k, v = c.Next() {
            txID := hex.EncodeToString(k)
            outs := DeserializeOutputs(v)

            for outIdx, out := range outs.Outputs {
                if out.IsLockedWithKey(pubkeyHash) && accumulated < amount {
                    accumulated += out.Value
                    unspentOutputs[txID] = append(unspentOutputs[txID],
outIdx)
                }
            }
        }
    })

    return accumulated, unspentOutputs
}
```

Hoặc kiểm tra balance:

```
func (u UTXOSet) FindUTXO(pubKeyHash []byte) []TXOutput {
    var UTXOs []TXOutput
    db := u.Blockchain.db

    err := db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(utxoBucket))
        c := b.Cursor()

        for k, v := c.First(); k != nil; k, v = c.Next() {
            outs := DeserializeOutputs(v)

            for _, out := range outs.Outputs {
                if out.IsLockedWithKey(pubKeyHash) {
                    UTXOs = append(UTXOs, out)
                }
            }
        }

        return nil
    })

    return UTXOs
}
```

Đây là những sửa đổi nhỏ của hai hàm tương tự trong **Blockchain**. Các hàm này sẽ chuyển sang cho UTXOSet chứ không nằm ở **Blockchain** nữa.

Bây giờ dữ liệu của chúng ta đã được chia ra 2 phần: tất cả các transaction được lưu trong blockchain và các utxo được lưu trong UTXO Set. Vì thế cần có sự đồng bộ giữa 2 bên để UTXO Set có thể luôn cập nhật với dữ liệu mới của blockchain. Nhưng chúng ta lại không muốn chạy hàm **Reindex()** mỗi khi có block mới. Chúng ta sẽ thêm hàm sau:

```
func (u UTXOSet) Update(block *Block) {
    db := u.Blockchain.db

    err := db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(utxoBucket))

        for _, tx := range block.Transactions {
            if tx.IsCoinbase() == false {
                for _, vin := range tx.Vin {
                    updatedOuts := TXOutputs{}
                    outsBytes := b.Get(vin.Txid)
                    outs := DeserializeOutputs(outsBytes)

                    for outIdx, out := range outs.Outputs {
                        if outIdx != vin.Vout {
                            updatedOuts.Outputs = append(updatedOuts.Outputs,
out)
                        }
```

```
                    }

                    if len(updatedOuts.Outputs) == 0 {
                        err := b.Delete(vin.Txid)
                    } else {
                        err := b.Put(vin.Txid, updatedOuts.Serialize())
                    }

                }
            }

            newOutputs := TXOutputs{}
            for _, out := range tx.Vout {
                newOutputs.Outputs = append(newOutputs.Outputs, out)
            }

            err := b.Put(tx.ID, newOutputs.Serialize())
        }
    })
}
```

Khi một block được đào, UTXO Set sẽ phải được cập nhật lại. Cập nhật bao gồm việc loại bỏ những utxo đã sử dụng và thêm vào các utxo mới phát sinh.

Khởi tạo UTXO Set lúc tạo blockchain:

```
func (cli *CLI) createBlockchain(address string) {
    ...
    bc := CreateBlockchain(address)
    defer bc.db.Close()

    UTXOSet := UTXOSet{bc}
    UTXOSet.Reindex()
    ...
}
```

Hàm reindex sẽ chạy ngay khi một blockchain mới được tạo. Hiện tại đây là nơi duy nhất Reindex được chạy. Lúc này mới blockchain cũng chỉ có một block nên gọi hàm Reindex có vẻ hơi quá, thay vào đó có thể là Update. Tuy nhiên hàm Reindex sẽ được dùng nhiều trong tương lai.

```
func (cli *CLI) send(from, to string, amount int) {
    ...
    newBlock := bc.MineBlock(txs)
    UTXOSet.Update(newBlock)
}
```

UTXO Set được cập nhật sau khi có block mới được tạo.

Hãy kiểm tra xem chương trình của chúng ta chạy chưa:

```
$ blockchain_go createblockchain -address 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1
00000086a725e18ed7e9e06f1051651a4fc46a315a9d298e59e57aeacbe0bf73

Done!

$ blockchain_go send -from 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1 -to
12DkLzLQ4B3gnQt62EPRJGZ38n3zF4Hzt5 -amount 6
0000001f75cb3a5033aeecbf6a8d378e15b25d026fb0a665c7721a5bb0faa21b

Success!

$ blockchain_go send -from 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1 -to
12ncZhA5mFTTnTmHq1aTPYBri4jAK8TacL -amount 4
000000cc51e665d53c78af5e65774a72fc7b864140a8224bf4e7709d8e0fa433

Success!

$ blockchain_go getbalance -address 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1
Balance of '1F4MbuqjcuJGymjcuYQMUVYB37AWKkSLif': 20

$ blockchain_go getbalance -address 12DkLzLQ4B3gnQt62EPRJGZ38n3zF4Hzt5
Balance of '1XWu6nitBWe6J6v6MXmd5rhdP7dZsExbx': 6

$ blockchain_go getbalance -address 12ncZhA5mFTTnTmHq1aTPYBri4jAK8TacL
Balance of '13UASQpCR8Nr41PojH8Bz4K6cmTCqweskL': 4
```

Địa chỉ **1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1** đã nhận 3 lần reward:
1. Lúc đào genesis block
2. Lúc đào được block
0000001f75cb3a5033aeecbf6a8d378e15b25d026fb0a665c7721a5bb0faa21b
3. Lúc đào được block
000000cc51e665d53c78af5e65774a72fc7b864140a8224bf4e7709d8e0fa433

Merkle Tree

Thêm một phương pháp tối ưu chúng ta sẽ thực hiện ở phần này.

Như đã nói ở trên, dung lượng lưu trữ của Bitcoin lên tới hơn 140Gb, và mỗi node tham gia hệ thống đều phải lưu đủ lượng dữ liệu này. Điều này khiến cho nhiều người tham gia Bitcoin gặp khó khăn và họ đều không muốn chạy một node đầy đủ (full node). Nhưng một điều chắc chắn là node nào cũng luôn phải đảm bảo tính chính xác khi xác nhận các transaction. Thêm nữa, do dung lượng quá lớn, sẽ cần rất nhiều băng thông để có thể tương tác với nhau trong hệ thống Bitcoin.

Trong [White Paper của Bitcoin](https://bitcoin.org/bitcoin.pdf) xuất bản bởi Satoshi Nakamoto, có một giải pháp cho vấn đề này: Simplified Payment Verification (SPV). SVP là một Bitcoin node không cần tải toàn bộ blockchain, không cần xác minh block và transaction. Thay vào đó, nó lại liên kết tới một full node để lấy các dữ liệu cần thiết. Cơ chế này giúp có thể chạy nhiều wallet nhẹ (light wallet) chỉ với một full node.

Để SPV có thể hoạt động, cần phải có phương pháp kiểm tra xem block có chứa một transaction hay không, mà không cần tải toàn bộ dữ liệu block. Đây chính là ứng dụng của Merkle tree.

Merkle tree được sử dụng để Bitcoin lưu giữ các transaction hash, sau đó được lưu vào block header, và cho làm input cho proof-of-work. Blockchain của chúng ta hiện tại chỉ đang ghép các hash của các transaction với nhau rồi lấy hash SHA-256. Cũng là một cách tốt để lấy hash, tuy nhiên không có được những lợi ích mà Merkle tree mang lại.

Hãy xem biểu diễn của một Merkle tree:

Merkle tree được tạo ra cho mỗi block, nó bắt đầu bởi các nhánh là với mỗi lá là một transaction hash. Số lượng lá phải là chẵn, nếu số lượng transaction là lẻ thì nó sẽ sao chép thêm transaction cuối vào.

Chạy từ dưới lên, các lá được ghép thành cặp, sau đó ghép 2 hash lại với nhau, sau đó lại hash tiếp kết quả này để tạo thành một node mới của tree. Lặp lại tiến trình này cho tới khi chỉ còn lại 1 node, gọi là gốc của tree (root). Hash của root này được dùng làm hash chung cho toàn bộ transaction và được lưu trong block header.

Lợi ích của Merkle tree là một node có thể xác minh một transaction thuộc block mà không cần tải toàn bộ block đó về. Chỉ cần có transaction hash, Merkle tree root hash, và một Merkle tree path (đường đi trên tree).

Hãy tiến hành code:

```
type MerkleTree struct {
    RootNode *MerkleNode
}

type MerkleNode struct {
    Left  *MerkleNode
    Right *MerkleNode
    Data  []byte
}
```

Đây là cấu trúc tree điển hình. Mỗi **MerkleNode** có hai lá. **MerkleTree** là root link tới node đầu tiên của Merkle tree.

Hàm tạo node mới:

```
func NewMerkleNode(left, right *MerkleNode, data []byte) *MerkleNode {
    mNode := MerkleNode{}

    if left == nil && right == nil {
        hash := sha256.Sum256(data)
        mNode.Data = hash[:]
    } else {
        prevHashes := append(left.Data, right.Data...)
```

```
        hash := sha256.Sum256(prevHashes)
        mNode.Data = hash[:]
    }

    mNode.Left = left
    mNode.Right = right

    return &mNode
}
```

Mỗi node sẽ chứa một trường dữ liệu **Data**. Khi node là lá, nó sẽ là dữ liệu của transaction hash. Khi nó trỏ tới hai nhánh con, dữ liệu sẽ là hash của hai nhánh con đó.

```
func NewMerkleTree(data [][]byte) *MerkleTree {
    var nodes []MerkleNode

    if len(data)%2 != 0 {
        data = append(data, data[len(data)-1])
    }

    for _, datum := range data {
        node := NewMerkleNode(nil, nil, datum)
        nodes = append(nodes, *node)
    }

    for i := 0; i < len(data)/2; i++ {
        var newLevel []MerkleNode

        for j := 0; j < len(nodes); j += 2 {
            node := NewMerkleNode(&nodes[j], &nodes[j+1], nil)
            newLevel = append(newLevel, *node)
        }

        nodes = newLevel
    }

    mTree := MerkleTree{&nodes[0]}

    return &mTree
}
```

Khi một tree mới được tạo, điều đầu tiên phải có là số lá chẵn. Sau đó, **data** (là các hash của transaction) được chuyển thành các lá. Từ đó xây dựng nên các nhánh đi dần lên gốc.

Sau đó hàm **Block.HashTransactions** sẽ được cập nhật lại.

```
func (b *Block) HashTransactions() []byte {
    var transactions [][]byte

    for _, tx := range b.Transactions {
        transactions = append(transactions, tx.Serialize())
```

```
    }
    mTree := NewMerkleTree(transactions)

    return mTree.RootNode.Data
}
```

Các transaction được serialized thành mảng data dùng cho Merkle tree. Hàm này sẽ lấy hash của root Merkle tree này dùng cho proof-of-work.

P2PKH

Thêm một tính năng bạn cần biết của Bitcoin.

Ở các phần trước bạn đã biết Bitcoin sử dụng ngôn ngữ _Script_, dùng để khoá và mở khoá các transaction output; và các transaction input chứa dữ liệu để mở các khoá này. Ngôn ngữ này rất đơn giản, mã code chỉ là một chuỗi các toán tử và dữ liệu.

```
5 2 OP_ADD 7 OP_EQUAL

```
5, **2** và **7** là dữ liệu. **OP_ADD** và **OP_EQUAL** là các toán tử. Ngôn ngữ Script được chạy từ trái qua phải: dữ liệu được bỏ vào stack và toán tử thì thực hiện trên các thành phần của stack. Stack này giống như bộ nhớ FILO (First Input Last Output): thành phần vào trước thì sẽ lấy ra sau cùng, các thành phần tiếp theo sẽ được đặt nằm trên.

Các bước thực hiện đoạn mã trên như sau:

1. Stack: empty. Script: 5 2 OP_ADD 7 OP_EQUAL.
2. Stack: 5. Script: 2 OP_ADD 7 OP_EQUAL.
3. Stack: 5 2. Script: OP_ADD 7 OP_EQUAL.
4. Stack: 7. Script: 7 OP_EQUAL.
5. Stack: 7 7. Script: OP_EQUAL.
6. Stack: true. Script: empty.

Trong đó, **OP_ADD** sẽ lấy 2 thành phần trên cùng của stack, cộng lại rồi đưa kết quả vào stack. **OP_EQUAL** lấy 2 thành phần trên cùng của stack, so sánh chúng, nếu bằng nhau thì đặt **true** vào stack, ngược lại thì đặt **false**. Kết quả của một đoạn mã là giá trị cuối cùng của stack, trong trường hợp này là **true**, có nghĩa là đoạn mã trên đã chạy đúng.

Giờ hãy xem đoạn mã thực tế của Bitcoin:

```
<signature> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY
OP_CHECKSIG
```

Đoạn mã trên gọi là Pay to Public Key Hash (P2PKH), được sử dụng nhiều nhất trong Bitcoin. Nghĩa của nó là: trả tiền cho một publick key hash (khoá coin lại cho key này). Đây chính là trái tim của việc thanh toán trong Bitcoin: không hề có account, không hề chuyển tiền qua lại, chỉ có một đoạn mã kiểm tra xem signature và public key có đúng hay không.

Đoạn mã trên thực tế được lưu bằng hai phần:

1. Phần đầu tiên **\<signature\>** **\<pubKey\>** được lưu trong **input.ScriptSig**
2. Phần còn lại, **OP_DUP OP_HASH160 \<pubKeyHash\> OP_EQUALVERIFY OP_CHECKSIG** được lưu trong **output.ScriptPubKey**

Có thể thấy, output chính là logic mở khoá, còn input chứa dữ liệu để mở khoá output.
Khi chạy đoạn mã trên sẽ như sau:

1. Stack: empty

   Script: \<signature\> \<pubKey\> OP_DUP OP_HASH160 \<pubKeyHash\> OP_EQUALVERIFY OP_CHECKSIG

2. Stack: \<signature\>

   Script: \<pubKey\> OP_DUP OP_HASH160 \<pubKeyHash\> OP_EQUALVERIFY OP_CHECKSIG

3. Stack: \<signature\> \<pubKey\>

   Script: OP_DUP OP_HASH160 \<pubKeyHash\> OP_EQUALVERIFY OP_CHECKSIG

4. Stack: \<signature\> \<pubKey\> \<pubKey\>

   Script: OP_HASH160 \<pubKeyHash\> OP_EQUALVERIFY OP_CHECKSIG

5. Stack: \<signature\> \<pubKey\> \<pubKeyHash\>

   Script: \<pubKeyHash\> OP_EQUALVERIFY OP_CHECKSIG

6. Stack: \<signature\> \<pubKey\> \<pubKeyHash\> \<pubKeyHash\>

   Script: OP_EQUALVERIFY OP_CHECKSIG

7. Stack: \<signature\> \<pubKey\>

   Script: OP_CHECKSIG

8. Stack: true or false. Script: empty.

Trong đó:

- **OP_DUP** nhân đôi thành phần đầu stack
- **OP_HASH160** lấy thành phần đầu stack và hash nó với **RIPEMD160**
- **OP_EQUALVERIFY** so sánh hai thành phần đầu tiên của stack, nếu không bằng nhau thì thoát luôn
- **OP_CHECKSIG** kiểm tra signature so với pubKey. Lệnh này khá phức tạp: nó sẽ lấy một phần của transaction, hash nó, và kiểm tra signature có đúng không, dựa vào **\<signature\>** và **\<pubKey\>**

Dựa vào ngôn ngữ _Script_ trên, Bitcoin cũng có thể được coi là hệ thống smart-contract. Nhờ nó, một số hình thức thanh toán khác cũng khả thi.

Kết luận

Như vậy chúng ta đã xây dựng được hầu hết các tính năng chủ chốt của một blockchain-based cyptocurrency. Chúng ta có blockchain, address, mining, và transaction. Nhưng có thêm một cơ chế giúp Bitcoin trở thành một hệ thống toàn cầu: sự đồng thuận (consensus). Trong phần tới chúng ta sẽ xây dựng phần "phân tán" của blockchain.

Links

1. [Full source codes](https://github.com/Jeiwan/blockchain_go/tree/part_6)
2. [The UTXO Set](https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_2):_Data_Storage#The_UTXO_set_.28chainstate_leveldb.29)
3. [Merkle Tree](https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_2):_Data_Storage#The_UTXO_set_.28chainstate_leveldb.29)
4. [Script](https://en.bitcoin.it/wiki/Script)
5. [“Ultraprune” Bitcoin Core commit](https://github.com/sipa/bitcoin/commit/450cbb0944cd20a06ce806e6679a1f4c83c50db2)
6. [UTXO set statistics](https://statoshi.info/dashboard/db/unspent-transaction-output-set)
7. [Smart contracts and Bitcoin](https://medium.com/@maraoz/smart-contracts-and-bitcoin-a5d61011d9b1)
8. [Why every Bitcoin user should understand “SPV security”](https://medium.com/@jonaldfyookball/why-every-bitcoin-user-should-understand-spv-security-520d1d45e0b9)

Mục lục

1. [Lập trình Blockchain với Golang. Part 1: Cơ bản](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part1.md)
2. [Lập trình Blockchain với Golang. Part 2: Proof-of-work](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part2.md)
3. [Lập trình Blockchain với Golang. Part 3: Lưu trữ và tương tác CLI](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part3.md)
4. [Lập trình Blockchain với Golang. Part 4: Transactions 1](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part4.md)
5. [Lập trình Blockchain với Golang. Part 5: Address](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part5.md)
6. [Lập trình Blockchain với Golang. Part 6: Transaction 2](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part6.md)

7. [Lập trình Blockchain với Golang. Part 7: Network](https://github.com/hlongvu/blockchain-go-vietnamese/blob/master/Blockchain-go-part7.md)

Lập trình Blockchain với Golang. Part 7: Network

>Bài dịch từ _Building Blockchain in Go_ của tác giả _Ivan Kuznetsov_. Khi sử dụng vui lòng trích dẫn nguồn
[@hlongvu](https://github.com/hlongvu/blockchain-go-vietnamese)

Mục lục

Giới thiệu

Tới giờ chúng ta đã xây dựng được một blockchain với các tính năng chủ chốt như: ẩn danh, an toàn, địa chỉ ngẫu nhiên, lưu trữ dữu liệu, Proof-Of-Work, UTXO Set. Những tính năng này là thiết yếu, tuy nhiên vẫn chưa đủ. Thứ giúp cho những tính năng này nổi bật, thứ giúp tiền mã hoá có thể thành công đó là network. Thử hỏi chúng ta dùng blockchain làm gì nếu chỉ chạy trên một máy tính. Tiền mã hoá cũng có tác dụng gì nếu chỉ có một người sử dụng. Chính network là thứ khiến các tính năng này hữu dụng.

Các bạn có thể nghĩ tới blockchain như là các luật lệ, giống như trong cuộc sống con người có những quy tắc và luật riêng để cùng sinh sống và phát triển với nhau. Một loại trật tự xã hội. Mạng lưới blockchain chính là một cộng đồng các chương trình cùng theo một luật, cũng vì cùng tuân theo một luật này khiến cho mạng lưới tồn tại. Tương tự như loài người cùng có chung một lí tưởng thì sẽ trở nên mạnh mẽ hơn và có thể xây dựng một cuộc sống tốt đẹp hơn.
Nếu có ai đó lại đi theo các luật lệ khác, họ sẽ phải sống trong một xã hội khác. Cũng như vậy trong blockchain, nếu có một node đi theo một luật khác, nó cũng sẽ tạo nên một mạng lưới hoàn toàn khác.

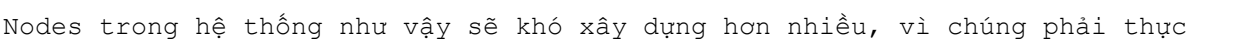
Điều này rất quan trọng: Nếu không có network và số đông các node tham gia theo cùng một luật chung, thì những luật lệ này cũng vô tác dụng.

> Không may là tôi chưa có thời gian để xây dựng một mạng lưới P2P hoàn chỉnh. Trong phần này tôi chỉ đề cập tới một trường hợp thường gặp, đó là một network có các kiểu node khác nhau. Bổ sung thêm vào kiến trúc này để xây dựng một mạng P2P sẽ là phần dành cho bạn nghiên cứu và khám phá. Tôi cũng không dám chắc là phần code này có thể hoạt động trong những tình huống khác, ngoài những thứ đề cập trong phần này.
>
> Hàm lượng code thay đổi là rất lớn nên các bạn có thể theo dõi tất cả ở [đây](https://github.com/Jeiwan/blockchain_go/compare/part_5...part_6#files_bucket)

Blockchain Network - Mạng lưới blockchain

Blockchain network được phân tán (decentralized), nghĩa là không có server xử lí riêng, không có client lấy dữ liệu từ server. Trong blockchain network có các node, và mỗi node là một thành viên đầy đủ của cả hệ thống. Mỗi node đảm nhiệm cả server và client. Đây là điều cần nhớ vì nó sẽ rất khác với kiến trúc của web app.

Blockchain network là một P2P (Peer-to-Peer) network, có nghĩa là các node kết nối trực tiếp với nhau. Cấu trúc liên kết (topology) của nó là phẳng vì không có sự phân chia giữa các node. Hãy xem mô tả dưới đây:



Nodes trong hệ thống như vậy sẽ khó xây dựng hơn nhiều, vì chúng phải thực hiện rất nhiều chức năng. Mỗi node phải tương tác với nhiều node khác, yêu cầu trạng thái của node kia, so sánh với trạng thái của chính nó và thực hiện cập nhật.

Node Roles

Mặc dù mỗi node đều là các thành viên đầy đủ, nhưng mỗi node có thể đảm nhiệm các chức năng khác nhau trong hệ thống. Đây là các loại:

1. Miner

Những node này chạy trên những cỗ máy tính toán mạnh (như ASIC), và mục đích duy nhất của nó là tính toán để tìm ra block mới một cách nhanh nhất có thể. Chỉ có các blockchain có Proof-of-Work mới có miner, vì thực chất việc đào coin là giải các bài toán PoW. Trong blockchain sử dụng Proof-of-Stake thì không có việc đào.

2. Full node

Những node này đảm nhiệm việc xác minh block được đào bởi miner và xác minh các transaction. Để làm được điều này, các full node phải có bản sao đầy đủ của blockchain. Cùng với đó, chúng cũng có nhiệm vụ định tuyến cho các node khác, giúp các node khác tìm thấy nhau.

Việc có nhiều full node là rất cần thiết cho cả hệ thống, vì đây là những node đưa ra quyết định một block hay transaction có đúng hay không.

3. SPV

Các SPV (Simplified Payment Verification) node không lưu toàn bộ bản sao của blockchain, nhưng chúng vẫn thực hiện việc xác minh (không phải tất

cả mà chỉ một phần nhỏ hơn, ví dụ: chỉ xác minh các transaction gửi tới address xác định). Một node SPV phải dựa vào một full node để lấy dữ liệu. Nhiều node SPV có thể kết nối tới một full node. SPV giúp cho các ứng dụng ví có thể chạy được một cách dễ dàng hơn.

Network Đơn Giản Hoá
Để xây dựng blockchain của chúng ta, có một số tính năng cần đơn giản hoá. Vì chúng ta không có nhiều máy tính để mô phỏng cả network với nhiều node. Có thể sử dụng máy ảo (VM) hoặc Docker để giải quyết vấn đề này, tuy nhiên sẽ làm việc xây dựng trở nên phức tạp hơn. Vậy chúng ta vừa muốn chạy nhiều node trong cùng một máy, lại vừa muốn các node này hoạt động riêng biệt với nhau. Để làm điều này, chúng ta sẽ **dùng port làm id cho các node**, thay vì dùng IP. Ví dụ, sẽ có các node với địa chỉ: **127.0.0.1:3000**, **127.0.0.1:3001**, **127.0.0.1:3002**,... Chúng ta sử dụng port biến env (environment variable) **NODE_ID** để đặt tên cho các node. Sau đó chúng ta có thể bật nhiều cửa sổ terminal, đặt các **NODE_ID** khác nhau và chạy nhiều node.

Cách này cũng đòi hỏi phải sử dụng file dữ liệu khác nhau cho blockchain và wallet. Chúng sẽ có thêm **NODE_ID** trong các tên file như sau: **blockchain_3000.db**, **blockchain_30001.db**, **wallet_3000.db**, **wallet_30001.db**, ...

Code

Điều gì xảy ra khi bạn tải Bitcoin Core và chạy chương trình lần đầu tiên? Nó sẽ phải kết nối tới node nào đó để tải bản sao mới nhất của blockchain. Vậy làm thế nào để máy tính của bạn biết được địa chỉ của các node để kết nối tới?

Hardcode địa chỉ node trong Bitcoin không phải là cách tốt, nếu node đó bị tấn công hoặc bị tắt, sẽ dẫn tới việc node mới không thể kết nối với mạng. Thay vào đó, Bitcoin Core sử dụng **DNS seeds**. Đây không phải là node mà là nơi để lấy địa chỉ của một số node. Khi bạn bật Bitcoin Core lần đầu tiên, nó sẽ kết nối tới các seeds này và lấy về một list các node để kết nối tới.

Trong blockchain của chúng ta sẽ có sự tập trung (centralization). Chúng ta sẽ có 3 nodes:

1. Central Node: Node trung tâm, các node còn lại sẽ kết nối tới, và gửi dữ liệu tới các node khác.
2. Miner Node: Node này sẽ lưu các transaction mới trong mempool và khi có đủ transaction sẽ tiến hành đào block mới.
3. Wallet node: Node này dùng để gửi coin giữa các wallet. Không giống với SPV, node này lưu giữ toàn bộ blockchain.

Tình huống

Phần này chúng ta sẽ xây dựng hệ thống theo tình huống sau:
1. Central node sẽ tạo một blockchain
2. Các node khác (wallet) kết nối tới nó và tải toàn bộ blockchain
3. Một vài miner kết nối tới central node và tải toàn bộ blockchain
4. Wallet node tạo một transaction
5. Miner node nhận transaction và giữ trong mempool
6. Khi có đủ transaction trong mempool, miner bắt đầu đào block mới
7. Sau khi block được đào, nó được gửi cho central node
8. Wallet node sẽ đồng bộ cùng với central node

9. Người dùng ở wallet node sẽ kiểm tra xem việc gửi tiền có thành công hay không

Dù tình huống trên không xảy ra trên hệ thống P2P thực tế nhưng cũng là một use case quan trọng nhất giống với Bitcoin.

Version

Các node giao tiếp với nhau thông qua các message. Khi một node mới chạy, nó lấy được một vài địa chỉ node khác qua DNS seed, gửi message **version**, như sau:

```
type version struct {
    Version    int
    BestHeight int
    AddrFrom   string
}
```

Chúng ta chỉ có một version blockchain, nên biến **Version** cũng không có thông tin gì quan trọng (trong Bitcoin có Main net và một số test net). **BestHeight** lưu độ cao của blockchain mà node đang có. **AddFrom** là địa chỉ của node gửi message.

Vậy khi một node nào đó nhận được message **version** nó sẽ trả lời như thế nào? Nó sẽ trả lời với một message **version** khác. Giống như việc bắt tay (hankshake): không cần làm việc gì khác trước khi chào nhau. Nhưng nó không vô nghĩa, **version** được dùng để xem ai có blockchain dài hơn. Khi một node nhận message **version**, nó sẽ kiểm tra xem blockchain mà nó đang lưu giữ có dài hơn **BestHeight** hay không. Nếu không, node đó sẽ yêu cầu và tải thêm các block còn thiếu.

Để nhận các message, chúng ta cần một server:

```
var nodeAddress string
var knownNodes = []string{"localhost:3000"}

func StartServer(nodeID, minerAddress string) {
    nodeAddress = fmt.Sprintf("localhost:%s", nodeID)
    miningAddress = minerAddress
    ln, err := net.Listen(protocol, nodeAddress)
    defer ln.Close()

    bc := NewBlockchain(nodeID)

    if nodeAddress != knownNodes[0] {
        sendVersion(knownNodes[0], bc)
    }

    for {
        conn, err := ln.Accept()
        go handleConnection(conn, bc)
    }
}
```

Chúng ta hard-code địa chỉ của central node. **minerAddress** là địa chỉ hưởng phần thưởng từ việc đào coin.

```
if nodeAddress != knownNodes[0] {
    sendVersion(knownNodes[0], bc)
}
```

Đoạn mã trên kiểm tra xem nó có phải là central node hay không, nếu không sẽ gửi message **version** tới central node để kiểm tra xem blockchain đã đồng bộ chưa.

```
func sendVersion(addr string, bc *Blockchain) {
    bestHeight := bc.GetBestHeight()
    payload := gobEncode(version{nodeVersion, bestHeight, nodeAddress})

    request := append(commandToBytes("version"), payload...)

    sendData(addr, request)
}
```

Các message của node là một mảng byte. 12 byte đầu tiên là mã lệnh (command), tiếp theo là mã hoá gob của message struct. Hàm **commandToBytes** như sau:

```
func commandToBytes(command string) []byte {
    var bytes [commandLength]byte

    for i, c := range command {
        bytes[i] = byte(c)
    }

    return bytes[:]
}
```

Nó tạo ra 12 byte buffer và đặt command vào đó, các byte còn lại rỗng. Một hàm ngược lại:

```
func bytesToCommand(bytes []byte) string {
    var command []byte

    for _, b := range bytes {
        if b != 0x0 {
            command = append(command, b)
        }
    }

    return fmt.Sprintf("%s", command)
}
```

Khi một node nhận được command, nó sẽ chạy hàm **bytesToCommand** lấy command ra. Tương tứng với mỗi command sẽ có một phần xử lí riêng:

```
func handleConnection(conn net.Conn, bc *Blockchain) {
    request, err := ioutil.ReadAll(conn)
    command := bytesToCommand(request[:commandLength])
    fmt.Printf("Received %s command\n", command)

    switch command {
    ...
    case "version":
        handleVersion(request, bc)
    default:
        fmt.Println("Unknown command!")
    }

    conn.Close()
}
```

Đây là phần xử lí cho command **version**:

```
func handleVersion(request []byte, bc *Blockchain) {
    var buff bytes.Buffer
    var payload version

    buff.Write(request[commandLength:])
    dec := gob.NewDecoder(&buff)
    err := dec.Decode(&payload)

    myBestHeight := bc.GetBestHeight()
    foreignerBestHeight := payload.BestHeight

    if myBestHeight < foreignerBestHeight {
        sendGetBlocks(payload.AddrFrom)
    } else if myBestHeight > foreignerBestHeight {
        sendVersion(payload.AddrFrom, bc)
    }

    if !nodeIsKnown(payload.AddrFrom) {
        knownNodes = append(knownNodes, payload.AddrFrom)
    }
}
```

Đầu tiên chúng ta cần giải mã yêu cầu và lấy nội dung version vào payload. Phần decode này tương tự nhau trong các handler, sẽ không trích dẫn nữa ở các bước tiếp.

Sau đó, node sẽ so sánh **BestHeight** của nó với giá trị trong message. Nếu dài hơn nó sẽ trả lời bằng message **version**, ngược lại nó sẽ gửi **getblocks**.

getblocks

```
type getblocks struct {
    AddrFrom string
}
```

getblocks mang ý nghĩa "hãy cho tôi xem các block mà bạn có" (trong Bitcoin thì phức tạp hơn). Hãy chú ý, nó không nói "hãy đưa tôi tất cả block", thay vào đó nó chỉ yêu cầu các block hash. Như vậy để làm giảm tải hệ thống, block có thể tải từ nhiều node khác nhau, và chúng ta cũng không muốn tải cả Gb từ một nơi về.

Xử lí command này như sau:

```
func handleGetBlocks(request []byte, bc *Blockchain) {
    ...
    blocks := bc.GetBlockHashes()
    sendInv(payload.AddrFrom, "block", blocks)
}
```
Trong thiết kế đơn giản của chúng ta, node sẽ trả về tất cả block hash.

inv

```
type inv struct {
    AddrFrom string
    Type     string
    Items    [][]byte
}
```
Bitcoin sử dụng **inv** để cho các nodes khác xem block và transaction mà nó đang có. Cũng như trên, nó không chứa toàn bộ block và transaction mà chỉ là các hash. Trường **Type** cho phép chúng ta biết đó là block hay transaction.

Xử lí **inv** phức tạp hơn đôi chút:

```
func handleInv(request []byte, bc *Blockchain) {
    ...
    fmt.Printf("Recevied inventory with %d %s\n", len(payload.Items),
payload.Type)

    if payload.Type == "block" {
        blocksInTransit = payload.Items

        blockHash := payload.Items[0]
        sendGetData(payload.AddrFrom, "block", blockHash)

        newInTransit := [][]byte{}
        for _, b := range blocksInTransit {
            if bytes.Compare(b, blockHash) != 0 {
                newInTransit = append(newInTransit, b)
            }
        }
        blocksInTransit = newInTransit
```

```
    }

    if payload.Type == "tx" {
        txID := payload.Items[0]

        if mempool[hex.EncodeToString(txID)].ID == nil {
            sendGetData(payload.AddrFrom, "tx", txID)
        }
    }
}
```

Nếu là block hash được gửi tới, chúng ta sẽ lưu lại vào **blocksInTransit**. Điều này cho phép chúng ta theo dõi và tải block từ các node khác nhau. Sau khi đặt block vào trạng thái inTransit, chúng ta gửi command **getdata** ngược lại để cập nhật **blocksInTransit**. Trong thực tế hệ thống P2P, chúng ta sẽ tải các block từ nhiều node khác nhau.

Thiết kế của chúng ta không gửi **inv** với nhiều hash. Đó là lý do khi check **payload.Type == "tx"** cũng chỉ hash đầu tiên được lấy. Sau đó chúng ta kiểm tra xem hash đó đã có trong mempool chưa, nếu chưa **getdata** được gửi đi.

getdata

```
type getdata struct {
    AddrFrom string
    Type     string
    ID       []byte
}
```

Hàm **getdata** yêu cầu một block hoặc transaction, và nó chỉ chứa được một block/transaction id.

```
func handleGetData(request []byte, bc *Blockchain) {
    ...
    if payload.Type == "block" {
        block, err := bc.GetBlock([]byte(payload.ID))

        sendBlock(payload.AddrFrom, &block)
    }

    if payload.Type == "tx" {
        txID := hex.EncodeToString(payload.ID)
        tx := mempool[txID]

        sendTx(payload.AddrFrom, &tx)
    }
}
```

Hàm xử lí **getdata** khá đơn giản: nếu yêu cầu block thì trả về block, nếu yêu cầu transaction thì trả về transaction. Chú ý, chúng ta không kiểm tra xem node có chưa block hay transaction đó không. Đây là 1 lỗi :)

block and tx

```
type block struct {
    AddrFrom string
    Block    []byte
}

type tx struct {
    AddFrom     string
    Transaction []byte
}
```

Đây là hai message gửi dữ liệu.

Khi nhận được block:

```
func handleBlock(request []byte, bc *Blockchain) {
    ...

    blockData := payload.Block
    block := DeserializeBlock(blockData)

    fmt.Println("Recevied a new block!")
    bc.AddBlock(block)

    fmt.Printf("Added block %x\n", block.Hash)

    if len(blocksInTransit) > 0 {
        blockHash := blocksInTransit[0]
        sendGetData(payload.AddrFrom, "block", blockHash)

        blocksInTransit = blocksInTransit[1:]
    } else {
        UTXOSet := UTXOSet{bc}
        UTXOSet.Reindex()
    }
}
```

Khi nhận được block, chúng ta đặt nó vào blockchain. Nếu có thêm block cần tải, chúng ta lại tiếp tục yêu cầu từ node mới gửi. Khi blockchain đã đồng bộ đầy đủ, UTXO Set sẽ reindex.

> TODO: Thay vì tin tưởng block gửi tới vô điều kiện, chúng ta phải xác minh block mới nhận được có đúng hay không trước khi cho vào blockchain.
>
> TODO: Thay vì chạy UTXOSet.Reindex(), hãy chạy UTXOSet.Update(block) đối với mỗi block nhận được để giảm thiểu việc quét qua cả blockchain lãng phí.

Khi nhận được message **tx** cần xử lí khá nhiều:

```
func handleTx(request []byte, bc *Blockchain) {
    ...
    txData := payload.Transaction
    tx := DeserializeTransaction(txData)
    mempool[hex.EncodeToString(tx.ID)] = tx

    if nodeAddress == knownNodes[0] {
        for _, node := range knownNodes {
            if node != nodeAddress && node != payload.AddFrom {
                sendInv(node, "tx", [][]byte{tx.ID})
            }
        }
    } else {
        if len(mempool) >= 2 && len(miningAddress) > 0 {
        MineTransactions:
            var txs []*Transaction

            for id := range mempool {
                tx := mempool[id]
                if bc.VerifyTransaction(&tx) {
                    txs = append(txs, &tx)
                }
            }

            if len(txs) == 0 {
                fmt.Println("All transactions are invalid! Waiting for new
ones...")
                return
            }

            cbTx := NewCoinbaseTX(miningAddress, "")
            txs = append(txs, cbTx)

            newBlock := bc.MineBlock(txs)
            UTXOSet := UTXOSet{bc}
            UTXOSet.Reindex()

            fmt.Println("New block is mined!")

            for _, tx := range txs {
                txID := hex.EncodeToString(tx.ID)
                delete(mempool, txID)
            }

            for _, node := range knownNodes {
                if node != nodeAddress {
                    sendInv(node, "block", [][]byte{newBlock.Hash})
                }
            }

            if len(mempool) > 0 {
                goto MineTransactions
            }
        }
```

```
    }
}
```

Điều đầu tiên cần làm là đặt transaction vào mempool (như trên, transaction cũng cần xác minh trước khi cho vào mempool). Sau đó:

```
if nodeAddress == knownNodes[0] {
    for _, node := range knownNodes {
        if node != nodeAddress && node != payload.AddFrom {
            sendInv(node, "tx", [][]byte{tx.ID})
        }
    }
}
```

Kiểm tra xem node hiện tại có phải là central node không. Trong thiết kế của chúng ta, central node không đào block mới mà gửi các transaction mới tới node khác trong network.

Bước tiếp theo chỉ dành cho miner:

```
if len(mempool) >= 2 && len(miningAddress) > 0 {
```

miningAddress chỉ được thiết lập tại miner node. Khi có 2 transaction hoặc nhiều hơn trong mempool thì tiến trình đào sẽ được bắt đầu.

```
for id := range mempool {
    tx := mempool[id]
    if bc.VerifyTransaction(&tx) {
        txs = append(txs, &tx)
    }
}

if len(txs) == 0 {
    fmt.Println("All transactions are invalid! Waiting for new ones...")
    return
}
```

Đầu tiên các transaction trong mempool được xác minh. Transaction sai sẽ bị bỏ qua, nếu không có transaction nào thì tiến trình đào bị bỏ qua.

```
cbTx := NewCoinbaseTX(miningAddress, "")
txs = append(txs, cbTx)

newBlock := bc.MineBlock(txs)
UTXOSet := UTXOSet{bc}
UTXOSet.Reindex()

fmt.Println("New block is mined!")
```

Các transaction đúng được cho vào block, cũng như coinbase transaction. Sau khi đào được block, UTXO Set được cập nhật.

> TODO: Tương tự như trên, UTXOSet.Update nên được dùng thay vì UTXOSet.Reindex

```
for _, tx := range txs {
    txID := hex.EncodeToString(tx.ID)
    delete(mempool, txID)
}

for _, node := range knownNodes {
    if node != nodeAddress {
        sendInv(node, "block", [][]byte{newBlock.Hash})
    }
}

if len(mempool) > 0 {
    goto MineTransactions
}
```

Sau khi transaction được đào, nó được bỏ ra khỏi mempool. Các node khác sẽ được gửi message **inv** với mã block hash mới. Các node đó sẽ có thể yêu cầu block đó sau khi xử lí message.

Result

Hãy chạy chương trình theo tình huống chúng ta đã đưa ra ở trên:

Đầu tiên chạy node với **NODE_ID = 3000** (export NODE_ID=3000) trong terminal 1. Để tóm tắt chúng ta sẽ gọi tắt là **NODE 3000** hoặc **NODE 3001** để các bạn có thể biết các thao tác trên node nào.

NODE 3000

Tạo wallet và blockchain mới:

```
$ blockchain_go createblockchain -address CENTREAL_NODE
```

(Tôi sử dụng address giả để cho tiện và dễ theo dõi)

Sau đó blockchain sẽ chỉ có genesis block. Chúng ta cần lưu lại block và dùng trong các node khác. Genesis block được dùng để làm định danh cho blockchain (trong Bitcoin Core, genesis block được hard code).

```
$ cp blockchain_3000.db blockchain_genesis.db
```

NODE 3001

Mở tiếp một terminal và đặt NODE ID bằng 3001. Đây là wallet node. Khởi tạo một số address với **blockchain_go createwallet**, chúng ta gọi các ví này là **WALLET_1**, **WALLET_2**, **WALLET_3**.

NODE 3000

Gửi coin tới các wallet:

```
$ blockchain_go send -from CENTREAL_NODE -to WALLET_1 -amount 10 -mine
$ blockchain_go send -from CENTREAL_NODE -to WALLET_2 -amount 10 -mine
```

Cờ **-mine** mang ý nghĩa block sẽ được đào ngay trong node đó. Chúng ta phải có cờ này vì ban đầu chưa có miner node nào trong mạng. Khởi động node:

```
$ blockchain_go startnode
```

Node này sẽ chạy cho tới cuối cùng.

NODE 3001

Chạy node với genesis block được lưu ở trên:

```
$ cp blockchain_genesis.db blockchain_3001.db
$ blockchain_go startnode
```

Nó sẽ tải các block từ central node. Để kiểm tra xem mọi thứ có hoạt động không, dừng node lại và kiểm tra balance:

```
$ blockchain_go getbalance -address WALLET_1
Balance of 'WALLET_1': 10

$ blockchain_go getbalance -address WALLET_2
Balance of 'WALLET_2': 10
```

Đồng thời bạn cũng có thể kiểm tra balance của central node, bởi node 3001 này cũng đã cập nhật blockchain:

```
$ blockchain_go getbalance -address CENTRAL_NODE
Balance of 'CENTRAL_NODE': 10
```

NODE 3002

Mở một cửa sổ mới và đặt NODE_ID bằng 30002, tạo một wallet mới. Đây sẽ là miner node. Khởi tạo blockchain và chạy node:

```
$ cp blockchain_genesis.db blockchain_3002.db
$ blockchain_go startnode -miner MINER_WALLET

```

NODE 3001

Gửi tiền:

```
$ blockchain_go send -from WALLET_1 -to WALLET_3 -amount 1
$ blockchain_go send -from WALLET_2 -to WALLET_4 -amount 1
```

NODE 3002

Hãy chuyển nhanh sang miner node để xem nó đang đào block mới. Đồng thời kiểm tra ouput của central node.

NODE 3001

Chuyển sang wallet node và chạy:

```
$ blockchain_go startnode
```

Nó sẽ tải về block mới. Sau đó dừng lại và kiểm tra balance:

```
$ blockchain_go getbalance -address WALLET_1
Balance of 'WALLET_1': 9

$ blockchain_go getbalance -address WALLET_2
Balance of 'WALLET_2': 9

$ blockchain_go getbalance -address WALLET_3
Balance of 'WALLET_3': 1

$ blockchain_go getbalance -address WALLET_4
Balance of 'WALLET_4': 1

$ blockchain_go getbalance -address MINER_WALLET
Balance of 'MINER_WALLET': 10
```

Vậy là xong!

Kết luận

Đây là phần cuối của seri. Tôi có thể làm thêm phần P2P tuy nhiên không có đủ thời gian. Hy vọng loạt bài này giúp trả lời các câu hỏi của các bạn về hệ thống Bitcoin, cũng như mang đến nhiều câu hỏi mới mà các bạn có thể tự tìm câu trả lời. Còn có nhiều điều hấp dẫn giấu trong công nghệ của Bitcoin! Chúc may mắn!

P.S. Bạn có thể nâng cao hệ thống này hơn bằng việc thiết kế message **addr**, được đặc tả trong Bitcoin như link dưới đây, giúp các node có thể tìm thấy nhau. Tôi đã bắt tay cài đặt nhưng còn chưa xong!

Links

1. [Source codes](https://github.com/Jeiwan/blockchain_go/tree/part_7)
2. [Bitcoin protocol documentation](https://en.bitcoin.it/wiki/Protocol_documentation)
3. [Bitcoin network](https://en.bitcoin.it/wiki/Network)

>Bài dịch từ _Building Blockchain in Go_ của tác giả _Ivan Kuznetsov_. Khi sử dụng vui lòng trích dẫn nguồn [@hlongvu](https://github.com/hlongvu/blockchain-go-vietnamese)

Mục lục