实验 1

区块链（Blockchain）是 21 世纪最具革命性的技术之一，它仍然处于不断成长的阶段，而且还有很多潜力尚未显现。作为比特币的底层技术，它本质上只是一个分布式数据库。不过使它独一无二的是，区块链是一个**公开**的而不是私人的数据库，每个使用它的人都有一个完整或者部分的副本。只有经过其他“数据库管理员”的同意，才能向其中添加新的记录。此外，也正是由于区块链，才使得**加密货币**和**智能合约**成为现实。

本实验将实现一个简化版的区块链，并基于此来构建一个简化版的加密货币。

准备工作：

1. Go 语言（可选）教程以及环境安装：
   http://www.runoob.com/go/go-environment.html
2. 安装数据库依赖包：$ go get -u github.com/boltdb/bolt

实验参考：

https://jeiwan.cc/
https://liuchengxu.gitbook.io/blockchain/

实验要求：

1. 认识区块链，了解其基本数据结构；
2. 理解共识算法 PoW 的基本原理和作用；
3. 自选语言完成基本原型，并实现持久化和命令行接口。

目录

1、 基本数据结构

首先从“区块”谈起。在区块链中，真正存储有效信息的是区块（block）。而在比特币中，真正有价值的信息就是交易（transaction）。实际上，交易信息是所有加密货币的价值所在。除此以外，区块还包含了一些技术实现的相关信息，比如版本，当前时间戳和前一个区块的哈希。

不过，我们要实现的是一个简化版的区块链，而不是一个像比特币技术规范所描述那样成熟完备的区块链。所以在我们目前的实现中，区块仅包含了部分关键信息，它的数据结构如下：

```go
// Block 由区块头和交易两部分构成
// Timestamp, PrevBlockHash, Hash 属于区块头 (block header)
// Timestamp     : 当前时间戳，也就是区块创建的时间
// PrevBlockHash : 前一个块的哈希
// Hash          : 当前块的哈希
// Data          : 区块实际存储的信息，比特币中也就是交易
type Block struct {
    Timestamp     int64
    PrevBlockHash []byte
    Hash          []byte
    Data          []byte
}
```

在我们的简化版区块中，还有一个 `Hash` 字段，那么，要如何计算哈希呢？哈希计算，是区块链一个非常重要的部分。正是由于它，才保证了区块链的安全。计算一个哈希，是在计算上非常困难的一个操作。即使在高速电脑上，也要耗费很多时间（这就是为什么人们会购买 GPU，FPGA，ASIC 来挖比特币）。这是一个架构上有意为之的设计，它故意使得加入新的区块十分困难，继而保证区块一旦被加入以后，就很难再进行修改。在接下来的内容中，我们将会讨论和实现这个机制。

目前，我们仅取了 `Block` 结构的部分字段（`Timestamp`, `Data` 和 `PrevBlockHash`），并将它们相互拼接起来，然后在拼接后的结果上计算一个 SHA-256，然后就得到了哈希.

```go
// SetHash 设置当前块哈希
// Hash = sha256(PrevBlockHash + Data + Timestamp)
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[:]
}
```

接下来，按照 Golang 的惯例，我们会实现一个用于简化创建区块的函数 `NewBlock`：

```go
// NewBlock 用于生成新块，参数需要 Data 与 PrevBlockHash
// 当前块的哈希会基于 Data 和 PrevBlockHash 计算得到
func NewBlock(data string, prevBlockHash []byte) *Block {
    block := &Block{
        Timestamp:     time.Now().Unix(),
        PrevBlockHash: prevBlockHash,
        Hash:          []byte{},
        Data:          []byte(data)}

    block.SetHash()

    return block
}
```

有了区块，下面让我们来实现区块**链**。本质上，区块链就是一个有着特定结构的数据库，是一个有序，每一个块都连接到前一个块的链表。也就是说，区块按照插入的顺序进行存储，每个块都与前一个块相连。这样的结构，能够让我们快速地获取链上的最新块，并且高效地通过哈希来检索一个块。

在 Golang 中，可以通过一个 array 和 map 来实现这个结构：array 存储有序的哈希（Golang 中 array 是有序的），map 存储 **hash -> block** 对(Golang 中, map 是无序的)。 但是在基本的原型阶段，我们只用到了 array，因为现在还不需要通过哈希来获取块。

```go
// BlockChain 是一个 Block 指针数组
type BlockChain struct {
    blocks []*Block
}
```

这就是我们的第一个区块链！是不是出乎意料地简单？就是一个 `Block` 数组。现在，让我们能够给它添加一个区块：

```go
// AddBlock 向链中加入一个新块
// data 在实际中就是交易
func (bc *BlockChain) AddBlock(data string) {
    prevBlock := bc.blocks[len(bc.blocks)-1]
    newBlock := NewBlock(data, prevBlock.Hash)
    bc.blocks = append(bc.blocks, newBlock)
}
```

结束！不过，就这样就完成了吗？

为了加入一个新的块，我们必须要有一个已有的块，但是，初始状态下，我们的链是空的，一个块都没有！所以，在任何一个区块链中，都必须至少有一个块。这个块，也就是链中的第一个块，通常叫做创世块（**genesis block**）. 让我们实现一个方法来创建创世块：

```go
// NewGenesisBlock 生成创世块
func NewGenesisBlock() *Block {
    return NewBlock("Genesis Block", []byte{})
}
```

现在，我们可以实现一个函数来创建有创世块的区块链：

```go
// NewBlockChain 创建一个有创世块的链
func NewBlockChain() *BlockChain {
    return &BlockChain{[]*Block{NewGenesisBlock()}}
}
```

检查一下我们的区块链是否如期工作：

```go
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()
    }
}
```

以上涉及到的 Go 语言包：

```go
package main

import (
    "bytes"
    "crypto/sha256"
    "strconv"
    "time"
    "fmt"
)
```

```
$ go build -o blockchain
$ ./blockchain
Prev hash:
Data: Genesis Block
Hash: e571c362ae3a8c77900b5e981e5faf577f06e54932404148149d9b7cc9f666a6

Prev hash: e571c362ae3a8c77900b5e981e5faf577f06e54932404148149d9b7cc9f666a6
Data: Send 1 BTC to Ivan
Hash: 9352ba0c09f2ec379e1be0501844c94de47eaf7aac1d4dcfce3201848d9c26a8

Prev hash: 9352ba0c09f2ec379e1be0501844c94de47eaf7aac1d4dcfce3201848d9c26a8
Data: Send 2 more BTC to Ivan
Hash: 6b552f19a8852620e43e6abfaeb0f6ecc21d9d0693ad71fbcd581cca212e203a
```

我们创建了一个非常简单的区块链原型：它仅仅是一个数组构成的一系列区块，每个块都与前一个块相关联。真实的区块链要比这复杂得多。在我们的区块链中，加入新的块非常简单，也很快，但是在真实的区块链中，加入新的块需要很多工作：你必须要经过十分繁重的计算（这个机制叫做工作量证明），来获得添加一个新块的权力。并且，区块链是一个分布式数据库，并且没有单一决策者。因此，要加入一个新块，必须要被网络的其他参与者确认和同意（这个机制叫做共识（consensus））。还有一点，我们的区块链还没有任何的交易！

2、 工作量证明（Proof-of-Work）

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

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

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

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

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

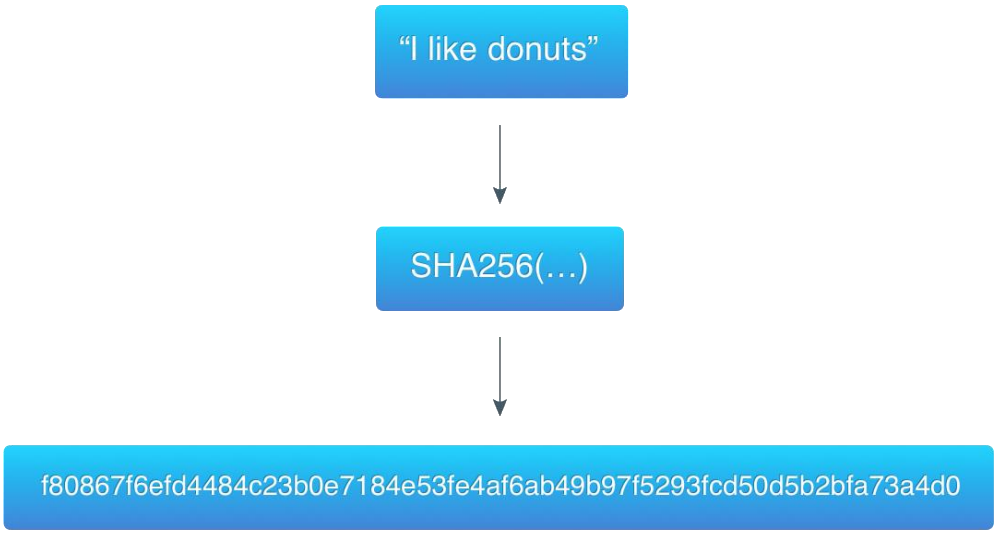
2.1 哈希计算

在本节，我们会讨论哈希计算。如果你已经熟悉了这个概念，可以直接跳过。

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

1. 无法从一个哈希值恢复原始数据。也就是说，哈希并不是加密。
2. 对于特定的数据，只能有一个哈希，并且这个哈希是唯一的。

3. 即使是仅仅改变输入数据中的一个字节，也会导致输出一个完全不同的哈希。



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

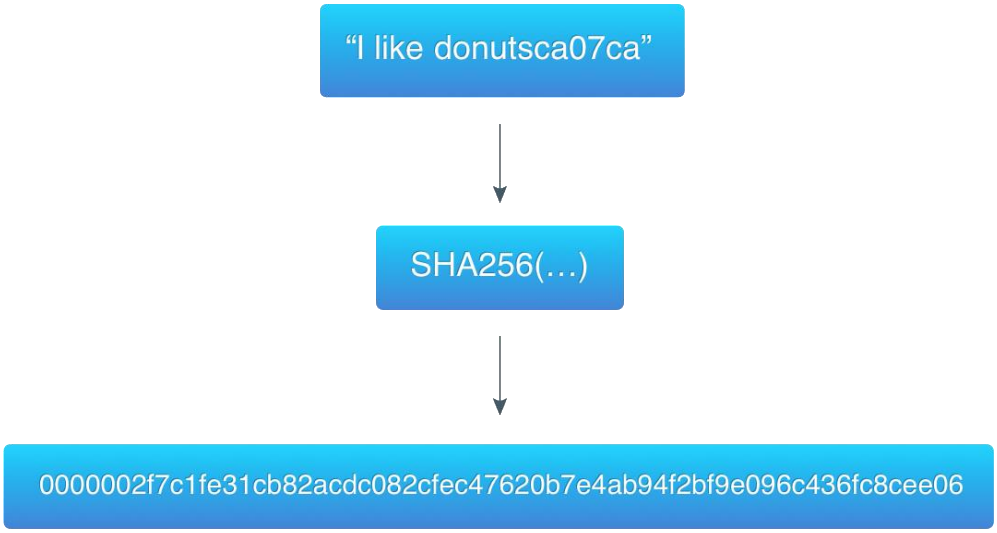
在区块链中，哈希被用于保证一个块的一致性。哈希算法的输入数据包含了前一个块的哈希，因此使得不太可能（或者，至少很困难）去修改链中的一个块：因为如果一个人想要修改前面一个块的哈希，那么他必须要重新计算这个块以及后面所有块的哈希。

2.2 Hashcash

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

1. 取一些公开的数据（比如，如果是 email 的话，它可以是接收者的邮件地址；在比特币中，它是区块头）
2. 给这个公开数据添加一个计数器。计数器默认从 0 开始

3. 将 **data(数据)** 和 **counter(计数器)** 组合到一起，获得一个哈希
4. 检查哈希是否符合一定的条件：
    1. 如果符合条件，结束
    2. 如果不符合，增加计数器，重复步骤 3-4



ca07ca 是计数器的 16 进制值，十进制的话是 13240266.

2.3 实现

与该部分相关的包：

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

完成了理论层面，来动手写代码吧！首先，定义挖矿的难度值

```
// 难度值，这里表示哈希的前 24 位必须是 0
const targetBits = 24
```

在比特币中，当一个块被挖出来以后，“target bits”代表了区块头里存储的难度，也就是开头有多少个 0。这里的 24 指的是算出来的哈希前 24 位必须是 0，如果用 16 进制表示，就是前 6 位必须是 0，这一点从最后的输出可以看出来。

目前我们并不会实现一个动态调整目标的算法，所以将难度定义为一个全局的常量即可。

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

```go
// 每个块的工作量都必须要证明，所有有个指向 Block 的指针
// target 是目标，我们最终要找的哈希必须要小于目标
type ProofOfWork struct {
    block  *Block
    target *big.Int
}

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

    pow := &ProofOfWork{b, target}

    return pow
}
```

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

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

```
0x10000000000000000000000000000000000000000000000000000000000000
```

它在内存上占据了 29 个字节。下面是与前面例子哈希的形式化比较：

```
0fac49161af82ed938add1d8725835cc123a1a87b1b196488360e58d4bfb51e3
0000010000000000000000000000000000000000000000000000000000000000
0000008b0f41ec78bab747864db66bcb9fb89920ee75f43fdaaeb5544f7f76ca
```

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

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

现在，我们需要有数据来进行哈希，准备数据：

```
// 工作量证明用到的数据有: PrevBlockHash, Data, Timestamp, targetBits, nonce
func (pow *ProofOfWork) prepareData(nonce int) []byte {
    data := bytes.Join(
        [][]byte{
            pow.block.PrevBlockHash,
            pow.block.Data,
            IntToHex(pow.block.Timestamp),
            IntToHex(int64(targetBits)),
            IntToHex(int64(nonce)),
        },
        []byte{},
    )

    return data
}
```

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

很好，到这里，所有的准备工作就完成了，下面来实现 PoW 算法的核心：

```
// 工作量证明的核心就是寻找有效哈希
func (pow *ProofOfWork) Run() (int, []byte) {
    var hashInt big.Int
    var hash [32]byte
    nonce := 0

    fmt.Printf("Mining the block containing \"%s\"\n", pow.block.Data)
    for nonce < maxNonce {
        data := pow.prepareData(nonce)

        hash = sha256.Sum256(data)
        hashInt.SetBytes(hash[:])

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

    return nonce, hash[:]
}
```

首先我们对变量进行初始化：

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

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

在这个循环中，我们做的事情有：

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

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

```
// 创建新块时需要运行工作量证明找到有效哈希
func NewBlock(data string, prevBlockHash []byte) *Block {
    block := &Block{
        Timestamp:     time.Now().Unix(),
        PrevBlockHash: prevBlockHash,
        Hash:          []byte{},
        Data:          []byte(data),
        Nonce:         0}
    pow := NewProofOfWork(block)
    nonce, hash := pow.Run()

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

    return block
}
```

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

```
// Nonce 在对工作量证明进行验证时用到
type Block struct {
    Timestamp     int64
    PrevBlockHash []byte
    Hash          []byte
    Data          []byte
    Nonce         int
}
```

还剩下一件事情需要做，对工作量证明进行验证：

```
// 验证工作量，只要哈希小于目标就是有效工作量
func (pow *ProofOfWork) Validate() bool {
    var hashInt big.Int

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

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

    return isValid
}
```

这里，就是我们就用到了上面保存的 `nonce`。

好了！现在让我们来运行一下是否正常工作：

```
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)
        pow := NewProofOfWork(block)
        fmt.Printf("PoW: %s\n", strconv.FormatBool(pow.Validate()))
        fmt.Println()
    }
}
```

```
-$ go build -o blockchain
-$ ./blockchain
Mining the block containing "Genesis Block"
000000f2e210a44016a40d3dd700d7d58e5d9f4f52742eab2b26c5b5fc6007c3

Mining the block containing "Send 1 BTC to Ivan"
000000ca39d4a16310af2ed3211f3dcd8916b9d91387293fa144b52de4bfdab9

Mining the block containing "Send 2 more BTC to Ivan"
0000008bf5651599009e64845e6145376626299c987fa82991b35e02a9270777

Prev hash:
Data: Genesis Block
Hash: 000000f2e210a44016a40d3dd700d7d58e5d9f4f52742eab2b26c5b5fc6007c3
PoW: true

Prev hash: 000000f2e210a44016a40d3dd700d7d58e5d9f4f52742eab2b26c5b5fc6007c3
Data: Send 1 BTC to Ivan
Hash: 000000ca39d4a16310af2ed3211f3dcd8916b9d91387293fa144b52de4bfdab9
PoW: true

Prev hash: 000000ca39d4a16310af2ed3211f3dcd8916b9d91387293fa144b52de4bfdab9
Data: Send 2 more BTC to Ivan
Hash: 0000008bf5651599009e64845e6145376626299c987fa82991b35e02a9270777
PoW: true
```

成功了！你可以看到每个哈希都是 3 个字节的 0 开始，并且获得这些哈希需要花费一些时间，这次我们产生三个块花费了一分多钟，比没有工作量证明之前慢了很多（也就是成本高了很多）。

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

3、 持久化和命令行接口

到目前为止，我们已经构建了一个有工作量证明机制的区块链。有了工作量证明，挖矿也就有了着落。虽然目前距离一个有着完整功能的区块链越来越近了，但是它仍然缺少了一些重要的特性。在今天的内容中，我们会将区块链持久化到一个数据库中，然后会提供一个简单的命令行接口，用来完成一些与区块链的交互操作。本质上，区块链是一个分布式数据库，不过，我们暂时先忽略“分布式”这个部分，仅专注于“存储”这一点。

目前，我们的区块链实现里面并没有用到数据库，而是在每次运行程序时，简单地将区块链存储在内存中。那么一旦程序退出，所有的内容就都消失了。我们没有办法再次使用这条链，也没有办法与其他人共享，所以我们需要把它存储到磁盘上。

那么，我们要用哪个数据库呢？实际上，任何一个数据库都可以。在 比特币原始论文 中，并没有提到要使用哪一个具体的数据库，它完全取决于开发者如何选择。 Bitcoin Core ，最初由中本聪发布，现在是比特币的一个参考实现，它使用的是 LevelDB。而我们将要使用的是...

3.1 BoltDB

因为它：

1. 非常简洁
2. 用 Go 实现
3. 不需要运行一个服务器
4. 能够允许我们构造想要的数据结构

本部分涉及到的包：

```
import (
    "github.com/boltdb/bolt"
)
```

并且使用前要安装对应依赖：

$ go get -u github.com/boltdb/bolt

Bolt 使用键值存储，这意味着它没有像 SQL RDBMS （MySQL，PostgreSQL 等等）的表，没有行和列。相反，数据被存储为键值对（key-value pair，就像 Golang 的 map）。键值对被存储在 bucket 中，这是为了将相似的键值对进行分组（类似 RDBMS 中的表格）。因此，为了获取一个值，你需要知道一个 bucket 和一个键（key）。

需要注意的一个事情是，Bolt 数据库没有数据类型：键和值都是字节数组（ byte array ）。鉴于需要在里面存储 Go 的结构（ 准确来说 ，也就是存储 **Block(块)** ），

我们需要对它们进行序列化，也就说，实现一个从 Go struct 转换到一个 byte array 的机制，同时还可以从一个 byte array 再转换回 Go struct。虽然我们将会使用 encoding/gob 来完成这一目标，但实际上也可以选择使用 **JSON**, **XML**, **Protocol Buffers** 等等。之所以选择使用 **encoding/gob**，是因为它很简单，而且是 Go 标准库的一部分。

虽然 BoltDB 的作者出于个人原因已经不在对其维护（见 README），不过关系不大，它已经足够稳定了，况且也有活跃的 fork:coreos/bblot。

3.2 数据库结构

在开始实现持久化的逻辑之前，我们首先需要决定到底要如何在数据库中进行存储。为此，我们可以参考 Bitcoin Core 的做法：

简单来说，Bitcoin Core 使用两个“bucket”来存储数据：

1. 其中一个 bucket 是 **blocks**，它存储了描述一条链中所有块的元数据
2. 另一个 bucket 是 **chainstate**，存储了一条链的状态，也就是当前所有的未花费的交易输出，和一些元数据

此外，出于性能的考虑，Bitcoin Core 将每个区块（block）存储为磁盘上的不同文件。如此一来，就不需要仅仅为了读取一个单一的块而将所有（或者部分）的块都加载到内存中。但是，为了简单起见，我们并不会实现这一点。详情可见(https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_2):_Data_Storage)。

因为目前还没有交易，所以我们只需要 **blocks** bucket。另外，正如上面提到的，我们会将整个数据库存储为单个文件，而不是将区块存储在不同的文件中。所以，我们也不会需要文件编号（file number）相关的东西。最终，我们会用到的键值对有：

1. 32 字节的 block-hash -> block 结构
2. l -> 链中最后一个块的 hash

这就是实现持久化机制所有需要了解的内容了。

3.3 序列化

本部分涉及到的包：

```
import (
    "bytes"
    "encoding/gob"
)
```

上面提到，在 BoltDB 中，值只能是 `[]byte` 类型，但是我们想要存储 `Block` 结构。所以，我们需要使用 encoding/gob 来对这些结构进行序列化。
让我们来实现 `Block` 的 `Serialize` 方法（为了简洁起见，此处略去了错误处理）：

```
// 将 Block 序列化为一个字节数组
func (b *Block) Serialize() []byte {
    var result bytes.Buffer
    encoder := gob.NewEncoder(&result)

    err := encoder.Encode(b)

    return result.Bytes()
}
```

这个部分比较直观：首先，我们定义一个 buffer 存储序列化之后的数据。然后，我们初始化一个 gob `encoder` 并对 block 进行编码，结果作为一个字节数组返回。

接下来，我们需要一个解序列化的函数，它会接受一个字节数组作为输入，并返回一个 `Block`. 它不是一个方法（method），而是一个单独的函数（function）：

```
// 将字节数组反序列化为一个 Block
func DeserializeBlock(d []byte) *Block {
    var block Block

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

    return &block
}
```

这就是序列化部分的内容了。

3.4 持久化

让我们从 `NewBlockchain` 函数开始。在之前的实现中，`NewBlockchain` 会创建一个新的 `Blockchain` 实例，并向其中加入创世块。而现在，我们希望它做的事情有：

1. 打开一个数据库文件
2. 检查文件里面是否已经存储了一个区块链
3. 如果已经存储了一个区块链：
    1. 创建一个新的 `Blockchain` 实例
    2. 设置 `Blockchain` 实例的 tip 为数据库中存储的最后一个块的哈希
4. 如果没有区块链：
    1. 创建创世块
    2. 存储到数据库
    3. 将创世块哈希保存为最后一个块的哈希
    4. 创建一个新的 `Blockchain` 实例，初始时 tip 指向创世块（tip 有尾部，尖端的意思，在这里 tip 存储的是最后一个块的哈希）

代码大概是这样：

```go
func NewBlockchain() *Blockchain {
    var tip []byte
    // 这是打开一个 BoltDB 文件的标准做法。注意，即使不存在这样的文件，它也不会返回错误。
    db, err := bolt.Open(dbFile, 0600, nil)

    // 在 BoltDB 中，数据库操作通过一个事务（transaction）进行操作。
    // 这里打开的是一个读写事务（db.Update(...)），因为我们可能会向数据库中添加创世块。
    err = db.Update(func(tx *bolt.Tx) error {
        // 函数的核心，先获取存储区块的 bucket
        b := tx.Bucket([]byte(blocksBucket))

        // 如果数据库中不存在区块链就创建一个，否则直接读取最后一个块的哈希
        if b == nil {
            fmt.Println("No existing blockchain found. Creating a new one...")
            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
    })

    // 注意创建 Blockchain 一个新的方式：
    bc := Blockchain{tip, db}

    return &bc
}
```

这次，我们不在里面存储所有的区块了，而是仅存储区块链的 `tip`。另外，我们存储了一个数据库连接。因为我们想要一旦打开它的话，就让它一直运行，直到程序运行结束。因此，`Blockchain` 的结构现在看起来是这样：

```go
// tip 这个词本身有事物尖端或尾部的意思，这里指的是存储最后一个块的哈希
// 在链的末端可能出现短暂分叉的情况，所以选择 tip 其实也就是选择了哪条链
// db 存储数据库连接
type Blockchain struct {
    tip []byte
    db  *bolt.DB
}
```

接下来我们想要更新的是 `AddBlock` 方法：现在向链中加入区块，就不是像之前向一个数组中加入一个元素那么简单了。从现在开始，我们会将区块存储在数据库里面：

```go
// 加入区块时，需要将区块持久化到数据库中
func (bc *Blockchain) AddBlock(data string) {
    var lastHash []byte
    // 这是 BoltDB 事务的另一个类型（只读）
    // 首先获取最后一个块的哈希用于生成新块的哈希
    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
    })
}
```

3.5 检查区块链

现在，产生的所有块都会被保存到一个数据库里面，所以我们可以重新打开一个链，然后向里面加入新块。但是在实现这一点后，我们失去了之前一个非常好的特性：再也无法打印区块链的区块了，因为现在不是将区块存储在一个数组，而是放到了数据库里面。让我们来解决这个问题！

BoltDB 允许对一个 bucket 里面的所有 key 进行迭代，但是所有的 key 都以字节序进行存储，而且我们想要以区块能够进入区块链中的顺序进行打印。此外，因为我们不想将所有的块都加载到内存中（因为我们的区块链数据库可能很大！或者现在可以假装它可能很大），我们将会一个一个地读取它们。故而，我们需要一个区块链迭代器（`BlockchainIterator`）：

```go
type BlockchainIterator struct {
    currentHash []byte
    db          *bolt.DB
}
```

每当要对链中的块进行迭代时，我们就会创建一个迭代器，里面存储了当前迭代的块哈希（`currentHash`）和数据库的连接（`db`）。通过 `db`，迭代器逻辑上被附属到一个区块链上（这里的区块链指的是存储了一个数据库连接的 `Blockchain` 实例），并且通过 `Blockchain` 方法进行创建：

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

    return bci
}
```

注意，迭代器的初始状态为链中的 tip，因此区块将从尾到头（创世块为头），也就是从最新的到最旧的进行获取。实际上，**选择一个 tip 就是意味着给一条链“投票”**。一条链可能有多个分支，最长的那条链会被认为是主分支。在获得一个 tip （可以是链中的任意一个块）之后，我们就可以重新构造整条链，找到它的长度和需要构建它的工作。这同样也意味着，一个 tip 也就是区块链的一种标识符。

`BlockchainIterator` 只会做一件事情：返回链中的下一个块。

```go
// 返回链中的下一个块
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
}
```

这就是数据库部分的内容了！

3.6 CLI

本部分涉及到的包：

```go
import (
    "flag"
    "fmt"
    "os"
)
```

到目前为止，我们的实现还没有提供一个与程序交互的接口：目前只是在 `main` 函数中简单执行了 `NewBlockchain` 和 `bc.AddBlock` 。是时候改变了！现在我们想要拥有这些命令：

```
$ blockchain addblock "Pay 0.031337 for a coffee"
$ blockchain printchain
```

所有命令行相关的操作都会通过 `CLI` 结构进行处理：

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

它的“入口”是 `Run` 函数：

```
func (cli *CLI) Run() {
    cli.validateArgs()
    // 使用标准库里面的 flag 包来解析命令行参数:
    // 首先创建两个子命令: addblock 和 printchain
    addBlockCmd := flag.NewFlagSet("addblock", flag.ExitOnError)
    printChainCmd := flag.NewFlagSet("printchain", flag.ExitOnError)
    // 然后给 addblock 添加 -data 标志, printchain 没有任何标志
    addBlockData := addBlockCmd.String("data", "", "Block data")
    // 然后, 我们检查用户提供的命令, 解析相关的 flag 子命令:
    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)
    }
    // 接着检查解析是哪一个子命令, 并调用相关函数:
    if addBlockCmd.Parsed() {
        if *addBlockData == "" {
            addBlockCmd.Usage()
            os.Exit(1)
        }
        cli.bc.AddBlock(*addBlockData)
    }

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

这部分内容跟之前的很像，唯一的区别是我们现在使用的是 `BlockchainIterator` 对区块链中的区块进行迭代。

记得不要忘了对 `main` 函数作出相应的修改：

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

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

注意，无论提供什么命令行参数，都会创建一个新的链。

结束前再次提醒，记得安装依赖包！！！

$ go get -u github.com/boltdb/bolt

这就是今天的所有内容了！来看一下是不是如期工作：

```
-$ go get -u github.com/boltdb/bolt
-$ go build -o blockchain
-$ ./blockchain printchain
No existing blockchain found. Creating a new one...
Mining the block containing "Genesis Block"
000000e7ed07a1408ada0c5aa34f47f434344e9bb63a93aa84ed7d68f870a304

Prev hash:
Data: Genesis Block
Hash: 000000e7ed07a1408ada0c5aa34f47f434344e9bb63a93aa84ed7d68f870a304
PoW: true

-$ ./blockchain addblock -data "Send 1 BTC to xiaoming"
Mining the block containing "Send 1 BTC to xiaoming"
000000b9f30bbdaf9c4bcf5510f159e575721a5e99be6187891138a0c47b0fc2

-$ ./blockchain addblock -data "Pay 0.31337 BTC for a coffee"
Mining the block containing "Pay 0.31337 BTC for a coffee"
00000003bc556241ef36cf718bd93b7d5ae2415298d9335de529221b81fc9bfc

-$ ./blockchain printchain
Prev hash: 000000b9f30bbdaf9c4bcf5510f159e575721a5e99be6187891138a0c47b0fc2
Data: Pay 0.31337 BTC for a coffee
Hash: 00000003bc556241ef36cf718bd93b7d5ae2415298d9335de529221b81fc9bfc
PoW: true

Prev hash: 000000e7ed07a1408ada0c5aa34f47f434344e9bb63a93aa84ed7d68f870a304
Data: Send 1 BTC to xiaoming
Hash: 000000b9f30bbdaf9c4bcf5510f159e575721a5e99be6187891138a0c47b0fc2
PoW: true

Prev hash:
Data: Genesis Block
Hash: 000000e7ed07a1408ada0c5aa34f47f434344e9bb63a93aa84ed7d68f870a304
PoW: true
```