用Go构建区块链——3.持久化和命令行

本篇是"用Go构建区块链"系列的第三篇，主要对原文进行翻译。对应原文如下：

```
https://jeiwan.cc/posts/building-blockchain-in-go-part-3/
```

1、介绍

到目前为止，我们已经构建了一个带有工作量证明的区块链，这使得挖矿称为可能。我们的实现越来越接近功能完整的区块链，但它仍然缺乏一些重要功能。今天将开始在数据库中存储区块链，之后我们将制作一个简单的命令行界面来执行区块链操作。本质上，区块链是一个分布式数据库。现在我们将省略"分布式"部分，并专注于"数据库"部分。

2、数据库选择

目前，我们实现的区块链中没有数据库; 相反，我们每次运行程序时都会创建区块并将它们存储在内存中。我们不能重复使用区块链，我们无法与其他人共享，因此我们需要将其存储在磁盘上。

我们需要哪个数据库？实际上，他们中的任何一个都可以。在最初的比特币原始论文中，关于使用某个数据库没有任何说法，因此它完全取决于开发者如何选择。 Bitcoin Core，最初由中本聪发布，也是目前比特币实现的参考版本。它使用LevelDB（尽管它仅在2012年引入客户端）。我们将使用...

3、BoltDB

因为：

1.它简单而很小；
2.它用Go语言实现；
3.它不需要运行服务器；
4.它允许构建我们想要的数据结构。

以下摘自 BoltDB在 Github的README ：

> Bolt是一个纯Go语言写的Key/Value存储，受到了Howard Chu的LMDB项目的启发。该项目的目标是为不需要完整数据库服务器（如Postgres或MySQL）的项目提供一个简单，快速且可靠的数据库。
>
> 由于Bolt旨在用作这种低级别的功能，因此简单性是关键。该API将很小，只专注于获取值

> 和设置值。仅此而已。

听起来非常适合我们的需求！让我们花一分钟审查一下。

BoltDB是一个键/值存储，这意味着没有像SQL RDBMS（MySQL，PostgreSQL等）那样的表，没有行，没有列。相反，数据存储为键值对（如Golang中的map）。键值对存储在桶(bucket)中，这些桶(bucket)用于将类似的键值对进行分组（这与RDBMS中的表类似）。因此，为了获得价值，你需要知道一个桶(bucket)和一个键(key)。

关于BoltDB的一个重要的事情是它没有数据类型：键和值是字节数组(byte arrays)。由于我们将Go结构（也就是区块 `Block` ）存储在其中，因此我们需要将它们序列化，即实现将Go结构转换为字节数组并将其从字节数组恢复的机制。我们将使用 encoding/gob 做这件事情。不过， `JSON` ， `XML` ， `Protocol Buffers` 等均可使用。我们使用 encoding/gob 是因为它很简单，并且是标准Go库的一部分。

4、数据库结构

在开始实施持久性逻辑之前，我们首先需要决定如何将数据存储在数据库中。为此，我们将介绍比特币核心的做法。

简而言之，Bitcoin Core使用两个"桶(buckets)"来存储数据：

1. `blocks` 存储描述链中所有块的元数据。
2. `chainstate` 存储链的状态，这是目前所有未使用的事务输出和一些元数据。

另外，区块在磁盘上作为单独的文件存储。这是为了达到性能目的而完成的：读取单个块不需要将全部（或部分）全部加载到内存中。我们不会实现这个。

在 `blocks` ， `key -> value` 对是：

1.'b' + 32 字节的区块 hash -> 区块索引记录
2.'f' + 4 字节的文件数字 -> 文件信息记录
3.'l' -> 4 字节的文件数字:最后一个使用过的区块文件数字
4.'R' -> 1 字节的布尔值: 我们是否要去重新索引
5.'F' + 1 字节的标志名长度 + 标志名字符串 -> 1 字节布尔值: 开或关的多种标志
6.'t' + 32 字节的交易hash -> 交易索引记录

在 `chainstate` ， `key -> value` 对是：
1.'c' + 32 字节的交易hash -> 未使用的交易出账记录
2.'B' -> 32 字节的区块hash: 数据库应该表示的未使用交易出账的区块哈希

（详细解释可以在这里找到）

因为我们暂时并没有交易信息，我们可以只有一个 `blocks` 桶。另外，如上所述，我们将整个数据库存储为单个文件，而不将区块存储在单独的文件中。所以我们不需要任何与文件编号相关的东西。所以这些是我们将要使用的 `key -> value` （键值）对：

1.32 字节的区块hash -> 区块结构(序列化后的)
2.'l' -> 链上最后一个区块的 hash

这就是我们开始实现持久性机制所需要知道的。

5、序列化

如前所述，BoltDB中的值只能是 `[]byte` 类型，而我们想要将 `Block` 结构存储在数据库中。我们将使用encoding/gob来序列化结构。

让我们来实现 `Serialize` 的方法 `Block` （错误处理此不再赘述）：

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

    err := encoder.Encode(b)

    return result.Bytes()
}
```

这篇文章很简单：首先，我们声明一个将存储序列化数据的缓冲区; 然后我们初始化一个 `gob` 编码器并对区块进行编码; 结果以字节数组的形式返回。

接下来，我们需要一个反序列化函数，它将接收一个字节数组作为输入并返回一个 `Block` 。这不是一个方法，而是一个独立的功能：

```
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！

6、持久化

我们从这个NewBlockchain函数开始。现在，它创建一个 `Blockchain` 的实例，并添加一个创世区块到里面。我们想做的是这样的：

1.打开一个数据库文件
2.检查是否存在区块链
3.如果有区块链：

1. 创建一个新的 `Blockchain` 实例。
2. 将 `Blockchain` 实例的提示设置为存储在数据库中的最后一个区块哈希。

4.如果不存在区块链：

1. 创建创世区块
2. 存储在数据库中
3. 将创世区块的哈希保存为最后一个区块的哈希
4. 创建一个新的 `Blockchain` 实例，其指向创世区块

在代码中，它看起来像这样：

```
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
}
```

让我们来一段一段地来回顾一下。

```
db, err := bolt.Open(dbFile, 0600, nil)
```

这是打开BoltDB文件的标准方式。注意，如果没有这样的文件，它不会返回错误。

```
err = db.Update(func(tx *bolt.Tx) error {
...
})
```

在BoltDB中，数据库操作在事务(transaction)中运行。有两种类型的事务：只读和读写。在这里，我们打开一个读写事务（`db.Update(...)`），因为我们希望将创世区块放在数据库中。

```
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"))
}
```

这是该函数的核心。在这里，我们获得桶(bucket)来存储我们的区块：如果存在，我们读 `l` 键；如果不存在，就生成创世块，创建bucket，将区块保存到其中，并更新 `l` 键来存储链中的最后一个区块的哈希。

另外，注意一下创建 `Blockchain` 的新方式：

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

我们不再存储所有的区块了，而只存储链的顶端。另外，我们存储一个数据库的连接。因为我们想一旦打开它，就在程序运行时保持打开状态。因此， `Blockchain` 结构现在看起来像这样：

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

接下来我们要更新的 `AddBlock` 方法：现在向链中添加区块并不像向数组中添加元素那么简

单。从现在开始，我们将在DB中存储区块：

```
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
    })
}
```

让我们来一段一段地来回顾一下：

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

    return nil
})
```

这是另一种（只读）类型的BoltDB事务。在这里，我们从数据库中获取最后一个区块哈希来使用它来挖掘一个新的区块哈希。

```
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
```

挖掘新块后，我们将其序列化表示保存到数据库中，并更新 `l` 的键(key)，它现在存储了新区块

的哈希。

完成！这并不难，是吗?

7、检查区块链

所有新块现在都保存在数据库中，因此我们可以重新打开区块链并为其添加新块。但是在实现这个之后，我们失去了一个重要的缺陷：我们不能再打印出区块链块，因为我们不再将块存储在数组中。让我们来修复这个问题！

BoltDB允许迭代桶(bucket)中的所有键，不过这些键是以字节排序的顺序存储，我们希望区块按照它们在区块链中的顺序进行打印。另外，因为我们不想将所有区块加载到内存中（我们的区块链数据库可能很大，或者我们假装它可以），我们将逐个读取它们。为此，我们需要一个区块链迭代器：

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

每次我们想要遍历区块链中的区块时，都会创建一个迭代器，它将存储当前迭代的区块哈希和到数据库的连接。因为后者，一个迭代器在逻辑上被依加到区块链的(它是一个 `Blockchain` 存储数据库连接的实例)，所以了，我们在 `Blockchain` 里创建方法:

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

    return bci
}
```

请注意，迭代器最初指向区块链的顶端(tip)，因此区块将从上到下，从最新到最旧获取。事实上， `选择提示意味着区块链的"投票"` 。区块链可以有多个分支，并且它们中被认为是最长的分支。在找到顶端(tip)（可以是区块链中的任何区块）后，我们可以重新构建整个区块链并计算其长度以及构建区块链所需的工作。这个事实意味着，一个顶端(tip)也就是区块链的一种标识符。

`BlockchainIterator` 只会做一件事：它会从区块链返回下一个区块。

```
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！

8、CLI（命令行）

到现在为止我们的实现并没有提供任何接口与程序交互：我们只是执行 `NewBlockchain`，`bc.AddBlock` 在 `main` 函数。是时候来改善这一点了！我们想要这些命令：

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

有的命令行依赖的操作都会被在 `CLI` 结构中进行：

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

它的 "入口" 是 `Run` 函数：

```
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()
    }
}
```

我们使用标准的 flag 包来解析命令行参数。

```
addBlockCmd := flag.NewFlagSet("addblock", flag.ExitOnError)
printChainCmd := flag.NewFlagSet("printchain", flag.ExitOnError)
addBlockData := addBlockCmd.String("data", "", "Block data")
```

首先，我们创建了两个子命令， `addblock` 和 `printchain` ，我们再加入 `-data` 标志前者。`printchain` 将不会有任何标志。

```
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)
}
```

接下来我们检查用户提供的命令并解析相关的 `flag` 子命令。

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

```
    cli.addBlock(*addBlockData)
}

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

接下来我们检查哪些子命令被解析并运行相关函数。

```
    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
        }
    }
}
```

这部分内容和我们之前的内容很像，唯一的区别是我们现在使用的是 `BlockchainIterator` 遍历区块链中的区块：

我们也不要忘记修改相应的 `main` 函数：

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

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

请注意，`Blockchain` 无论提供什么命令行参数，都会创建一个新的链。

就是这样！让我们来检查一切是否按预期工作：

```
$ 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
```

（啤酒的声音可以打开）

9、总结

下次我们将实现地址，钱包和（可能还有）交易。敬请期待！

链接：

1.完整的源代码

2.Bitcoin Core数据存储

3.boltdb

4.encoding/gob

5.flag