←Home

# Building Blockchain in Go. Part 3: Persistence and CLI

AUGUST 29, 2017
GOLANG BLOCKCHAIN BITCOIN



Chinese translations: by liuchengxu, by zhangli1.

## Introduction

So far, we've built a blockchain with a proof-of-work system, which makes mining possible. Our implementation is getting closer to a fully functional blockchain, but it still lacks some important features. Today will start storing a blockchain in a database, and after that we'll make a simple command-line interface to perform operations with the blockchain. In its essence, blockchain is a distributed database. We're going to omit the "distributed" part for now and focus on the "database" part.

## Database Choice

Currently, there's no database in our implementation; instead, we create blocks every time we run the program and store them in memory. We cannot reuse a blockchain, we cannot share it with others, thus we need to store it on the disk.

Which database do we need? Actually, any of them. In the original Bitcoin paper, nothing is said about using a certain database, so it's up to a developer what DB to use. Bitcoin

Core, which was initially published by Satoshi Nakamoto and which is currently a reference implementation of Bitcoin, uses LevelDB (although it was introduced to the client only in 2012). And we'll use…

## BoltDB

Because:

1. It's simple and minimalistic.
2. It's implemented in Go.
3. It doesn't require to run a server.
4. It allows to build the data structure we want.

From the BoltDB's README on Github:

> Bolt is a pure Go key/value store inspired by Howard Chu's LMDB project. The goal of the project is to provide a simple, fast, and reliable database for projects that don't require a full database server such as Postgres or MySQL.
>
> Since Bolt is meant to be used as such a low-level piece of functionality, simplicity is key. The API will be small and only focus on getting values and setting values. That's it.

Sounds perfect for our needs! Let's spend a minute reviewing it.

BoltDB is a key/value storage, which means there're no tables like in SQL RDBMS (MySQL, PostgreSQL, etc.), no rows, no columns. Instead, data is stored as key-value pairs (like in Golang maps). Key-value pairs are stored in buckets, which are intended to group similar pairs (this is similar to tables in RDBMS). Thus, in order to get a value, you need to know a bucket and a key.

One important thing about BoltDB is that there are no data types: keys and values are byte arrays. Since we'll store Go structs ( `Block` , in particular) in it, we'll need to serialize them, i.e. implement a mechanism of converting a Go struct into a byte array and restoring it back from a byte array. We'll use encoding/gob for this, but `JSON` , `XML` , `Protocol Buffers` , etc. can be used as well. We're using `encoding/gob` because it's simple and is a part of the standard Go library.

# Database Structure

Before starting implementing persistence logic, we first need to decide how we'll store data in the DB. And for this, we'll refer to the way Bitcoin Core does that.

In simple words, Bitcoin Core uses two "buckets" to store data:

1. `blocks` stores metadata describing all the blocks in a chain.
2. `chainstate` stores the state of a chain, which is all currently unspent transaction outputs and some metadata.

Also, blocks are stored as separate files on the disk. This is done for a performance purpose: reading a single block won't require loading all (or some) of them into memory. We won't implement this.

In `blocks`, the `key -> value` pairs are:

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`

In `chainstate`, the `key -> value` pairs are:

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`

*(Detailed explanation can be found here)*

Since we don't have transactions yet, we're going to have only `blocks` bucket. Also, as said above, we will store the whole DB as a single file, without storing blocks in separate files. So we won't need anything related to file numbers. So these are `key -> value` pairs we'll use:

1. `32-byte block-hash -> Block structure (serialized)`
2. `'l' -> the hash of the last block in a chain`

That's all we need to know to start implementing the persistence mechanism.

## Serialization

As said before, in BoltDB values can be only of `[]byte` type, and we want to store `Block` structs in the DB. We'll use encoding/gob to serialize the structs.

Let's implement `Serialize` method of `Block` (errors processing is omitted for brevity):

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

        err := encoder.Encode(b)

        return result.Bytes()
}
```

The piece is straightforward: at first, we declare a buffer that will store serialized data; then we initialize a `gob` encoder and encode the block; the result is returned as a byte array.

Next, we need a deserializing function that will receive a byte array as input and return a `Block`. This won't be a method but an independent function:

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

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

        return &block
}
```

And that's it for the serialization!

# Persistence

Let's start with the `NewBlockchain` function. Currently, it creates a new instance of `Blockchain` and adds the genesis block to it. What we want it to do is to:

1. Open a DB file.
2. Check if there's a blockchain stored in it.
3. If there's a blockchain:
    1. Create a new `Blockchain` instance.
    2. Set the tip of the `Blockchain` instance to the last block hash stored in the DB.
4. If there's no existing blockchain:
    1. Create the genesis block.
    2. Store in the DB.
    3. Save the genesis block's hash as the last block hash.
    4. Create a new `Blockchain` instance with its tip pointing at the genesis block.

In code, it looks like this:

```
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
}
```

Let's review this piece by piece.

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

This is a standard way of opening a BoltDB file. Notice that it won't return an error if there's no such file.

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

In BoltDB, operations with a database are run within a transaction. And there are two types of transactions: read-only and read-write. Here, we open a read-write transaction (`db.Update(...)`), because we expect to put the genesis block in the 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"))
}
```

This is the core of the function. Here, we obtain the bucket storing our blocks: if it exists, we read the `l` key from it; if it doesn't exist, we generate the genesis block, create the bucket, save the block into it, and update the `l` key storing the last block hash of the chain.

Also, notice the new way of creating a `Blockchain`:

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

We don't store all the blocks in it anymore, instead only the tip of the chain is stored. Also, we store a DB connection, because we want to open it once and keep it open while the program is running. Thus, the `Blockchain` structure now looks like this:

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

Next thing we want to update is the `AddBlock` method: adding blocks to a chain now is not as easy as adding an element to an array. From now on we'll store blocks in the 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
        })
}
```

Let's review this piece by piece:

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

            return nil
    })
```

This is the other (read-only) type of BoltDB transactions. Here we get the last block hash from the DB to use it to mine a new block hash.

```
    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
```

After mining a new block, we save its serialized representation into the DB and update the `l` key, which now stores the new block's hash.

Done! It wasn't hard, was it?

## Inspecting Blockchain

All new blocks are now saved in a database, so we can reopen a blockchain and add a new block to it. But after implementing this, we lost a nice feature: we cannot print out blockchain blocks anymore because we don't store blocks in an array any longer. Let's fix this flaw!

BoltDB allows to iterate over all the keys in a bucket, but the keys are stored in byte-sorted order, and we want blocks to be printed in the order they take in a blockchain. Also, because we don't want to load all the blocks into memory (our blockchain DB could be huge!.. or let's just pretend it could), we'll read them one by one. For this purpose, we'll need a blockchain iterator:

```
    type BlockchainIterator struct {
            currentHash []byte
```

```
        db           *bolt.DB
}
```

An iterator will be created each time we want to iterate over blocks in a blockchain and it'll store the block hash of the current iteration and a connection to a DB. Because of the latter, an iterator is logically attached to a blockchain (it's a `Blockchain` instance that stores a DB connection) and, thus, is created in a `Blockchain` method:

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

        return bci
}
```

Notice that an iterator initially points at the tip of a blockchain, thus blocks will be obtained from top to bottom, from newest to oldest. In fact, choosing a tip means "voting" for a blockchain. A blockchain can have multiple branches, and it's the longest of them that's considered main. After getting a tip (it can be any block in the blockchain) we can reconstruct the whole blockchain and find its length and the work required to build it. This fact also means that a tip is a kind of an identifier of a blockchain.

`BlockchainIterator` will do only one thing: it'll return the next block from a 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
}
```

That's it for the DB part!

## CLI

Until now our implementation hasn't provided any interface to interact with the program: we've simply executed `NewBlockchain`, `bc.AddBlock` in the `main` function. Time to improve this! We want to have these commands:

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

All command-line related operations will be processed by the `CLI` struct:

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

Its "entrypoint" is the `Run` function:

```
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()
        }
}
```

We're using the standard flag package to parse command-line arguments.

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

First, we create two subcommands, `addblock` and `printchain`, then we add `-data` flag to the former. `printchain` won't have any flags.

```
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)
}
```

Next we check the command provided by user and parse related `flag` subcommand.

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

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

Next we check which of the subcommands were parsed and run related functions.

```
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
                }
        }
}
```

This piece is very similar to the one we had before. The only difference is that we're now using a `BlockchainIterator` to iterate over blocks in a blockchain.

Also let's not forget to modify the `main` function accordingly:

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

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

Note that a new `Blockchain` is created no matter what command-line arguments are provided.

And that's it! Let's check that everything works as expected:

```
$ 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
```

*(sound of a beer can opening)*

## Conclusion

Next time we'll implement addresses, wallets, and (probably) transactions. So stay tuned!

## Links

1. Full source codes
2. Bitcoin Core Data Storage
3. boltdb
4. encoding/gob
5. flag



Ivan Kuznetsov
Write things



## Read more

What is Lightning Network and How to Try It Today Mar 2 2018

Building Blockchain in Go. Part 7: Network Oct 6 2017

Building Blockchain in Go. Part 6: Transactions 2 Sep 18 2017

| | |
|---|---|
| Building Blockchain in Go. Part 5: Addresses | Sep 11 2017 |
| Building Blockchain in Go. Part 4: Transactions 1 | Sep 4 2017 |
| Building Blockchain in Go. Part 2: Proof-of-Work | Aug 22 2017 |
| Building Blockchain in Go. Part 1: Basic Prototype | Aug 16 2017 |
| TIL: Convolutional Filters Are Weights | Aug 5 2017 |