Blockchains with Go

Introduction

For this assignment, you will be building functionality to create a simple blockchain system. Our blockchain will include a proof-of-work, so we will have to do some computation to create valid blocks.

As with the Go exercises, I have provided a code skeleton with some structure and hints.

Blockchain Basics

A single *block* in a blockchain is fundamentally a data structure that contains some information (more on what information later). That data is hashed with a cryptographic hash function. As long as the hash function matches the contents, we will say that it's a *valid* hash (but we'll refine the idea under "Proof-Of-Work" below).

Blocks become a blockchain when they are joined together. This happens because one of the pieces of data in each block is the hash value from the **previous block**.

That means that if you have the block 'n' from the chain and believe that its hash is valid, then you can also verify block 'n−1'. Block 'n−1' must hash to the "previous hash" from block 'n' . Similarly, you can verify blocks 'n−2', 'n−3', ….

As a result, we need a very small amount of information (one block) to be able to certify all steps in the chain since the beginning of time.

Proof-Of-Work Basics

If just verifying a sequence of blocks is all you want, then we're done. The *proof-of-work* forces a certain amount of computation to be required before someone can create a "valid" block.

The “proof” part of the proof-of-work requires that we have some value that is difficult to produce/find, but easy to verify once it is found. Any calculation with that property will do, but we will use the hash values we already need in the blockchain.

We will add a *proof* value to each block which can be any integer. We will select a difficulty, ‘d’ and insist that the last ‘d’ bits of the hash value (including the proof in the data that is hashed) be zero. A block can only be considered valid if its hash ends with ‘d’ zero bits.

That means in order to find a valid block, we have to iterate through possible “proof” values until we find one that makes the block's hash end in null bits. That work is (1) unavoidable as long as the hashing algorithm is good, and (2) easy to verify, since we just have to re-hash the block to check that it was done.

In a (simplified) diagram with d=16 (so the last 2 bytes = 4 hex characters are zeros), we end up with a blockchain that looks like this:

In each case, we had to choose a “proof” value that made all of the block's data (in the dashed box) hash to a value ending in ‘d’ zero bits. That hash becomes part of the data for the next block, and the chain continues.

Work Queue

We need to be able to compute the proof-of-work in parallel in order to calculate it as quickly as possible. We also need to be able to stop the calculations and return a valid proof as soon as we find one, without doing a bunch of (now useless) calculations.

I think this is a situation where solving the more general problem is easier than doing it in the context of blockchain mining.

See the provided `work_queue/work_queue.go` for an outline of how we need a work queue to behave for this assignment.

When a new queue is created, it needs to have channels created for jobs going into the queue, results coming out, and requests to halt the computations. (Our work queue doesn't distinguish between results from the tasks: they come out of the `Results` queue in whatever order they are produced.) Worker goroutines also need to be created which will watch the `Jobs` queue for work to do.

When a job (`Worker` instance) is put into the `Jobs` channel, it should be started (by calling `.Run()`) as soon as possible by one of the worker goroutines. Its result should be put into the `Results` channel.

Other code using the queue should be able to read from the `.Results` channel for results from any of the tasks, and deal with them as necessary.

Stopping the Queue

We will need our work queue to do a slightly non-standard trick: once we find a result (proof-of-work in this case), we want to stop processing work because all subsequent calculations are unnecessary.

In the `.worker()` method, that will mean processing jobs until the channel is closed and all messages in it are received (exactly what `for range` does).

When the queue's `.Shutdown` method is called, the `.Jobs` channel should be closed (so and `for range` loops exit, and no more messages can be sent) and any messages in `.Jobs` should be received (but not run).

Tests

The provided `work_queue_test.go` contains two tests. The first is for the basic “process jobs and give back results” behaviour of the queue. The second checks that the queue stops at the correct time when asked.

You do not need to add any additional tests to `work_queue_test.go`.

Blocks

In `blockchain/block.go`, we will start creating blocks. A `struct` that holds all of the data we need has been provided.

The first block in a blockchain is a little special. It will have generation 0, difficulty as specified in the argument to `Initial`, the empty string for data, and a previous hash of 32 zero bytes (`"\x00"`). **Create a function** `Initial` that generates the initial block for the chain.

To continue the chain, we will call a method on the last block on the chain. If we have a block `b`, calling `b.Next(message)` should create and return a new block that can go in the chain after `b`. That is, it will have one higher generation; the same difficulty; data as given in the argument; previous hash equal to `b`'s hash. **Create a method** `.Next` that creates a next-in-chain block.

Note that **these functions will not set the block's proof-of-work or hash** (and can leave them at the defaults when they are created by Go). That comes later…

Valid Blocks

Recall: a block is *valid* if it has a "proof" value that makes it hash to a value that ends in `.Difficulty` zero bits. We need a few things to check this: the actual data to hash, a function to calculate the hash, and a function to check if the hash ends in zero bits.

Hashing Blocks

These fields will go into our hash string, separated by colons. The integers will be in the usual decimal integer representation (i.e. the number ten is `"10"`):

- the previous hash, encoded to a string in hexadecimal;
- the generation;
- the difficulty;
- the data string;
- the proof-of-work.

After this code…

```
b0 := Initial(16)
b0.Mine(1)
b1 := b0.Next("message")
b1.Mine(1)
```

… my two hash strings are:

```
"0000000000000000000000000000000000000000000000000000000000000000:0:16::56231"
"6c71ff02a08a22309b7dbbcee45d291d4ce955caa32031c50d941e3e9dbd0000:1:16:message:2159"
```

[You might not have mining working as you read this, but you will soon.]

We will use the SHA256 hash function to hash these strings. **Write a method** `.CalcHash()` that calculates the hash value of a block.

My hashes with the code above, and hex-encoded (i.e. these are the result of `hex.EncodeToString(block.CalcHash())`) are:

```
"6c71ff02a08a22309b7dbbcee45d291d4ce955caa32031c50d941e3e9dbd0000"
"9b4417b36afa6d31c728eed7abc14dd84468fdb055d8f3cbe308b0179df40000"
```

Valid Hashes

We have a "valid" proof of work a block if the block's hash ends in `.Difficulty` zero bits. We need to check that.

Write a method `.ValidHash` that returns true if (and only if) the block's hash ends in `.Difficulty` zero bits.

The hash value will be a byte slice. The easiest way to check for `difficulty` zero bits at the end of the slice is probably something like:

- `nBytes = difficulty/8`
- `nBits = difficulty%8`
- Check each of the last `nBytes` bytes are `'\x00'`.

- Check that the next byte (from the end) is divisible by `1<<nBits` (one left-shifted `nBits` times is equal to
- 2nBits
-).

In the example above, we can see that if we create an initial block with `b0 := Initial(16)` then `b0.ValidHash()` should return `false`, but if we set `b0.Proof = 56231` then `b0.ValidHash()` should be `true`.

Another example, with this setup, `b1` is valid and has *exactly* 19 zero bits at the end of its hash:

```
b0 := Initial(19)
b0.SetProof(87745)
b1 := b0.Next("hash example 1234")
b1.SetProof(1407891)
```

But if we did this, the block is not valid. It has only 18 zero bits at the end of its hash:

```
b1.SetProof(346082)
```

Mining

Now that we have the basic structure for blocks, we need to do some mining. Put this functionality in `blockchain/mining.go`.

The process of mining a block is conceptually easy: check proof-of-work values until you find one that would make `block.ValidHash()` true. Once you have found it, set `block.Proof` and `block.Hash` to make the block valid. (The provided `.SetProof` does the last step.)

[It might be worth writing a simple method that loops over proof values starting a 0 until a valid one is found. You don't need this, but it completely avoids the need for the work queue, and may potentially be convenient for testing the more complex implementation we're about to create…]

A mining task will be a range of possible proof-of-work values: we want to check those values and return *any one of them* that makes the block's hash valid.

Mining Tasks

We will use our work queue for this. We will need a `struct` that implements the `Worker` interface, i.e. has a `.Run()` method. We can also use the `struct` for any other information needed to do the mining we want done.

The `.Run()` call should return a `MiningResult` struct that includes (at least) a `.Proof` (proof-of-work value, if found) and `.Found` (indicating if the search was successful).

Once you have that, you can use it in `block.MineRange(start, end, workers, chunks)`. This should check proof values for this block, from `start` to `end` (**inclusive**). The calculation should be done in parallel by the given number of workers, and dividing the work into `chunks` approximately-equal parts.

This must be done using the work queue created above. If that is implemented correctly, it should be fairly easy to do this work in parallel, and to stop checking proof values (soon) after a valid proof is found.

A note on handling types and interfaces: the work queue will insist you return an `interface{}` value from `.Run()`. To actually use the result, it will have to be treated as a `MiningResult`. You will need a "type assertion" to get that to work.

Example Blocks

Here is an example of valid hashes being mined, and the proof values I get:

```
b0 := Initial(7)
b0.Mine(1)
fmt.Println(b0.Proof, hex.EncodeToString(b0.Hash))
b1 := b0.Next("this is an interesting message")
b1.Mine(1)
fmt.Println(b1.Proof, hex.EncodeToString(b1.Hash))
b2 := b1.Next("this is not interesting")
b2.Mine(1)
fmt.Println(b2.Proof, hex.EncodeToString(b2.Hash))
```

This outputs:

```
385     379bf2fb1a558872f09442a45e300e72f00f03f2c6f4dd29971f67ea4f3d5300
20      4a1c722d8021346fa2f440d7f0bbaa585e632f68fd20fed812fc944613b92500
40      ba2f9bf0f9ec629db726f1a5fe7312eb76270459e3f5bfdc4e213df9e47cd380
```

There are other valid proof values for these blocks, but these are the ones my code finds (and the numerically-smallest). Changing to difficulty 20 in the above code, it outputs:

```
1209938         19e2d3b3f0e2ebda3891979d76f957a5d51e1ba0b43f4296d8fb37c470600000
989099          a42b7e319ee2dee845f1eb842c31dac60a94c04432319638ec1b9f989d000000
1017262         6c589f7a3d2df217fdb39cd969006bc8651a0a3251ffb50470cbc9a0e4d00000
```

Valid Blockchains

In `blockchain.go`, we will finally put our blocks together into a blockchain, so we can have a chain and verify the validity of each block.

A simple `struct` for a blockchain is provided. Complete a method `.Add` that appends a new block to the chain.

The method `.IsValid` will do the important work: ensuring a valid chain where we know each message was created with full knowledge of everything that came before. It should return `true` only if:

- The initial block has previous hash all null bytes and is generation zero.
- Each block has the same difficulty value.
- Each block has a generation value that is one more than the previous block.
- Each block's previous hash matches the previous block's hash.
- Each block's hash value actually matches its contents.
- Each block's hash value ends in difficulty null bits.

Tests

The provided `blockchain_test.go` doesn't actually contain any tests, but it should.

Write some tests that check the core functionality of the `Block`. Your tests should ensure that at least some of the functionality described above is correct. There is no requirement for complete code coverage or testing every single thing mentioned.

You should **write at least three or four tests** (depending on the complexity of each test, of course) and cover a reasonable variety of the requirements for the assignment.

You can assume that the testify/assert package is available when we're marking. You can install it like:

```
go get github.com/stretchr/testify/assert
```

Summary

All of the functions provided in the code skeleton should be complete with arguments and return values as specified. We will expect them to be there when testing your implementation.

Pieces provided that shouldn't be changed:

- `blockchain.Block`, ….`MiningResult`, ….`Blockchain`
- `blockchain.Block.SetProof`, ….`Mine`

These structs/functions/methods as in the provided code skeleton should be completed to work as described above:

- `work_queue.Create`
- `work_queue.WorkQueue`
- `work_queue.WorkQueue.Enqueue`, ….`Shutdown`
- `blockchain.Initial`
- `blockchain.Block.Next`, ….`CalcHash`, ….`ValidHash`, ….`MineRange`
- `blockchain.Blockchain.Add`, ….`IsValid`,
- Some tests in `blockchain_test.go`.