区块链是目前最热门的话题，广大读者都听说过比特币，或许还有智能合约，相信大家都非常想了解这一切是如何工作的。这篇文章就是帮助你使用 Go 语言来实现一个简单的区块链，用不到 200 行代码来揭示区块链的原理！高可用架构也会持续推出更多区块链方面文章，欢迎点击上方蓝色『高可用架构』关注。

“用不到200行 Go 代码就能实现一个自己的区块链！” 听起来有意思吗？有什么能比开发一个自己的区块链更好的学习实践方法呢？那我们就一起来实践下！

因为我们是一家从事医疗健康领域的科技公司，所以我们采用人类平静时的心跳数据（BPM心率）作为这篇文章中的示例数据。让我们先来统计一下你一分钟内的心跳数，然后记下来，这个数字可能会在接下来的内容中用到。

通过本文，你将可以做到：

创建自己的区块链

理解 hash 函数是如何保持区块链的完整性

如何创造并添加新的块

多个节点如何竞争生成块

通过浏览器来查看整个链

所有其他关于区块链的基础知识

但是，对于比如工作量证明算法（PoW）以及权益证明算法（PoS）这类的共识算法文章中将不会涉及。同时为了让你更清楚得查看区块链以及块的添加，我们将网络交互的过程简化了，关于 P2P 网络比如“全网广播”这个过程等内容将在下一篇文章中补上。

让我们开始吧！

设置

我们假设你已经具备一点 Go 语言的开发经验。在安装和配置 Go 开发环境后之后，我们还要获取以下一些依赖：

```
go get github.com/davecgh/go-spew/spew
```

spew 可以帮助我们在 console 中直接查看 struct 和 slice 这两种数据结构。

go get github.com/gorilla/mux

Gorilla 的 mux 包非常流行， 我们用它来写 web handler。

go get github.com/joho/godotenv

godotenv 可以帮助我们读取项目根目录中的 .env 配置文件，这样我们就不用将 http port 之类的配置硬编码进代码中了。比如像这样：

ADDR=8080

接下来，我们创建一个 main.go
文件。之后我们的大部分工作都围绕这个文件，让我开始编码吧！

导入依赖

我们将所有的依赖包以声明的方式导入进去：

```
package main

import (

"crypto/sha256"

"encoding/hex"

"encoding/json"

"io"

"log"

"net/http"

"os"

"time"
```

"github.com/davecgh/go-spew/spew"

"github.com/gorilla/mux"

"github.com/joho/godotenv"

)

数据模型

接着我们来定义一个结构体，它代表组成区块链的每一个块的数据模型：

type Block struct {

Index int

Timestamp string

BPM int

Hash string

PrevHash string

}

Index 是这个块在整个链中的位置

Timestamp 显而易见就是块生成时的时间戳

Hash 是这个块通过 SHA256 算法生成的散列值

PrevHash 代表前一个块的 SHA256 散列值

BPM 每分钟心跳数，也就是心率。还记得文章开头说到的吗？

接着，我们再定义一个结构表示整个链，最简单的表示形式就是一个 Block 的 slice：

var Blockchain Block

我们使用散列算法（SHA256）来确定和维护链中块和块正确的顺序，确保每一个块的 PrevHash 值等于前一个块中的 Hash 值，这样就以正确的块顺序构建出链：

散列和生成块

我们为什么需要散列？主要是两个原因：

在节省空间的前提下去唯一标识数据。散列是用整个块的数据计算得出，在我们的例子中，将整个块的数据通过 SHA256 计算成一个定长不可伪造的字符串。

维持链的完整性。通过存储前一个块的散列值，我们就能够确保每个块在链中的正确顺序。任何对数据的篡改都将改变散列值，同时也就破坏了链。以我们从事的医疗健康领域为例，比如有一个恶意的第三方为了调整“人寿险”的价格，而修改了一个或若干个块中的代表不健康的 BPM 值，那么整个链都变得不可信了。

我们接着写一个函数，用来计算给定的数据的 SHA256 散列值：

```
func calculateHash(block Block) string {

record := string(block.Index) + block.Timestamp + string(block.BPM) +
block.PrevHash

h := sha256.New

h.Write(byte(record))

hashed := h.Sum(nil)
```

```
return hex.EncodeToString(hashed)

}
```

这个 calculateHash 函数接受一个块，通过块中的 Index，Timestamp，BPM，以及 PrevHash 值来计算出 SHA256 散列值。接下来我们就能便携一个生成块的函数：

```
func generateBlock(oldBlock Block, BPM int) (Block, error) {

var newBlock Block

t := time.Now

newBlock.Index = oldBlock.Index + 1

newBlock.Timestamp = t.String

newBlock.BPM = BPM

newBlock.PrevHash = oldBlock.Hash

newBlock.Hash = calculateHash(newBlock)

return newBlock, nil

}
```

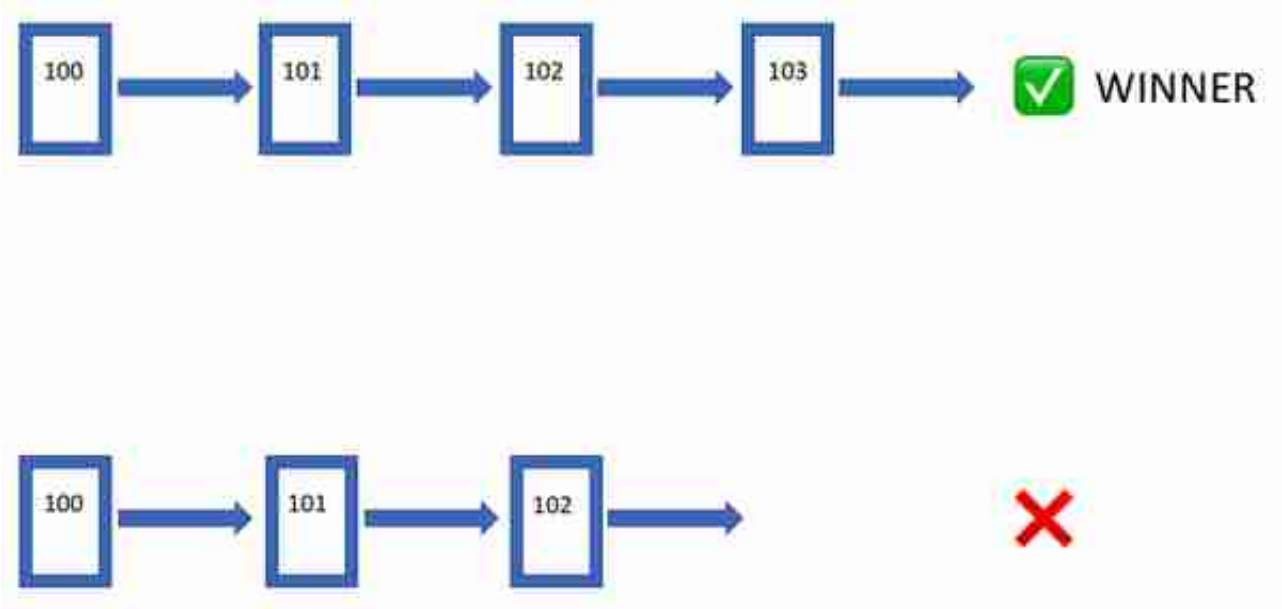
其中，Index 是从给定的前一块的 Index 递增得出，时间戳是直接通过 time.Now 函数来获得的，Hash 值通过前面的 calculateHash 函数计算得出，PrevHash 则是给定的前一个块的 Hash 值。

校验块

搞定了块的生成，接下来我们需要有函数帮我们判断一个块是否有被篡改。检查 Index 来看这个块是否正确得递增，检查 PrevHash 与前一个块的 Hash 是否一致，再来通过 calculateHash 检查当前块的 Hash 值是否正确。通过这几步我们就能写出一个校验函数：

```
func isBlockValid(newBlock, oldBlock Block) bool {

if oldBlock.Index+1 != newBlock.Index {

return false

}

if oldBlock.Hash != newBlock.PrevHash {

return false

}

if calculateHash(newBlock) != newBlock.Hash {

return false

}

return true

}
```

除了校验块以外，我们还会遇到一个问题：两个节点都生成块并添加到各自的链上，那我们应该以谁为准？这里的细节我们留到下一篇文章，这里先让我们记住一个原则：始终选择最长的链。



通常来说，更长的链表示它的数据（状态）是更新的，所以我们需要一个函数

能帮我们将本地的过期的链切换成最新的链：

```
func replaceChain(newBlocks []Block) {

if len(newBlocks) > len(Blockchain) {

Blockchain = newBlocks

}

}
```

到这一步，我们基本就把所有重要的函数完成了。接下来，我们需要一个方便直观的方式来查看我们的链，包括数据及状态。通过浏览器查看 web
页面可能是最合适的方式！

Web 服务

我猜你一定对传统的 web
服务及开发非常熟悉，所以这部分你肯定一看就会。借助 Gorilla/mux
包，我们先写一个函数来初始化我们的 web 服务：

```
func run error {
```

```
mux := makeMuxRouter

httpAddr := os.Getenv("ADDR")

log.Println("Listening on ", os.Getenv("ADDR"))

s := &http.Server{

Addr: ":" + httpAddr,

Handler: mux,

ReadTimeout: 10 * time.Second,

WriteTimeout: 10 * time.Second,

MaxHeaderBytes: 1
```