**区块链与数字货币 作业一**

**3200105872 庄毅非 软工2002**

1. **实验目的**

实现sha256算法，并使用自己实现的算法计算特定的nonce。

1. **实验环境**

**编程语言**: Golang

**系统** : MacOS Monterey(Darwin Kernel Version 21.6.0)

1. **实验步骤**
2. **Sha256算法介绍**

SHA256是SHA-2下细分出的一种算法，能够从任何类型的数据中提取出256位对应的消息摘要，本质上是一个散列函数，并且具有雪崩效应，被广泛用在区块链共识链的形成中，也是工作量证明机制（pow）中至关重要的一部分。

1. **Sha256算法实现**

**(1)消息预处理**

对于输入的字符串类型的数据，因为一个ascii字符占据的空间为8位，我们首先计算输入字符串对应的二进制长度（8 \* 字符串长度，将二进制长度记为binlen），之后我们需要在原消息后进行比特位的填补。填补过程为首先在原数组最后加上一位1，之后加上若干位0，直到binlen对512取模得到的结果为448。之后在其最后加上表示消息原长度的64个bit，即可实现消息预处理，我的消息预处理函数如下。



1. **计算消息摘要**

这是sha256算法的核心，我们首先需要将输入的uint8数组切分为一系列大小为512bit的块，之后对于每一个块，先使用数组存储其16个大小为32bit的部分，然后使用递推公式计算其剩下的48个大小为32bit的单元，之后使用sha256中定义的迭代公式迭代64次，将得到的计算结果加到sha256中定义的返回结果初始值hashArr中。当遍历完所有的chunk后，我们就得到的所需的消息摘要。



1. **验证结果**

我们需要检测每一个消息摘要是否是我们所需的前30位、31位、32位均为0的对象，在我的实现中，首先使用位运算检测前29位是否均为0，如果是，那么逐次判断其前32位、31位、30位是否均为0，并返回对应的枚举值。主函数通过检查返回值来判断是否是所需的结果。



1. **循环函数**

在实现了上述功能函数之后，我在search函数循环体中先进行初始字符串”Blockchain@ZhejiangUniversity”和nonce之间的拼接，然后使用myhash函数计算其对应的sha256摘要值，将该摘要值输入verify进行检验，并根据返回值来判断本次的nonce是否是要寻找的能够使前30或31或32位均为0的特征数，如果是，那么将nonce和对应的时间输出，否则直接自增nonce，进入下一轮循环。

这里的0x5F5E100是1e8的十六进制值，我在实现中设定每一个go协程的检测区间为1亿，所以一旦nonce不小于原始值base加上一亿，说明本区间中的一亿个数字都已经检测完毕，那么本协程终止并返回主线程。



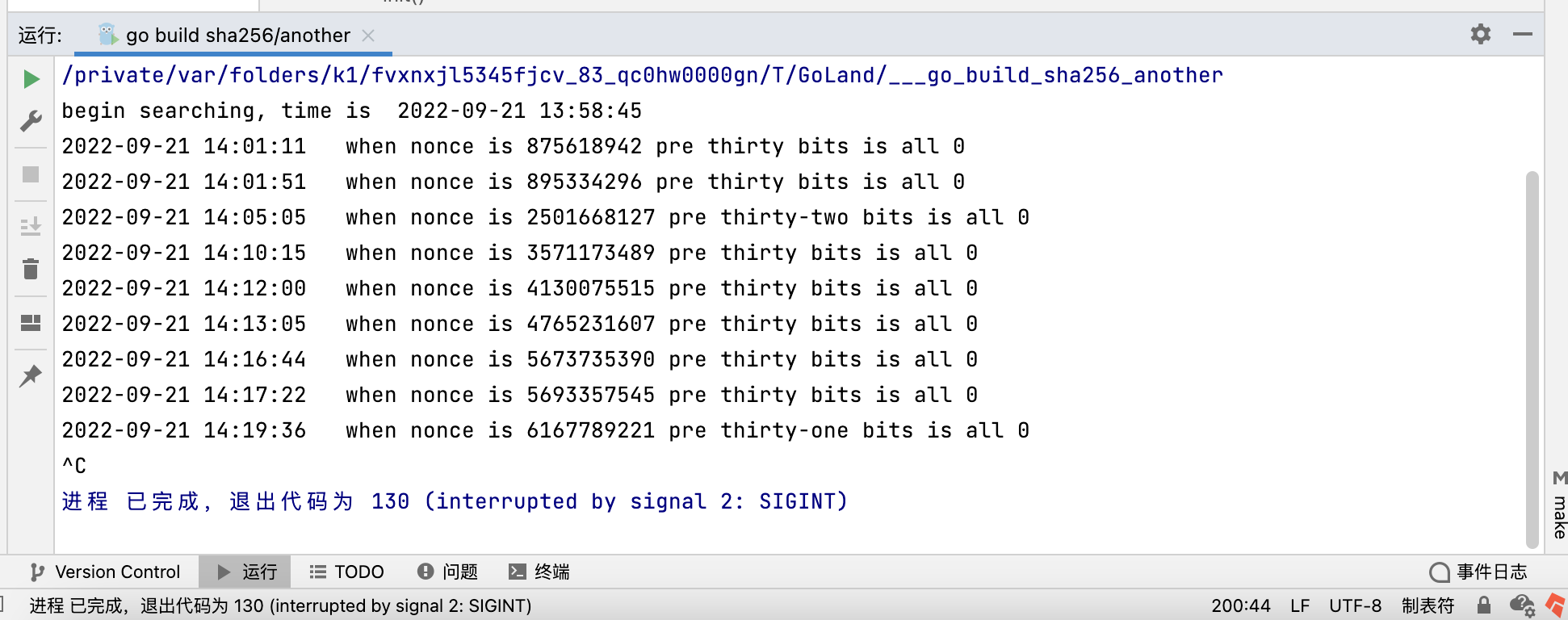
1. **主函数**

在主函数中，程序会自动执行init函数，进行precache和behcache的初始化，为字符串预处理函数作准备。然后会从0开始，每开辟一个协程，就将下一个协程对应的base值加上一亿，检查当前协程的总数是否大于等于11，如果是，那么就暂停主线程（这里是为了避免开辟过多协程导致协程之间切换开销影响程序运行），否则开辟一个协程计算对应base区间中是否有nonce符合条件。



1. **实验结果**

通过运行程序，我大概在3min左右找到了能够使前30位为0的数 875618942，之后在6min左右找到了能够使前32位为0的数 2501668127，在20min 左右找到了能够使前31位为0的数6167789221，运行截图如下。



1. **源代码**

将以下代码拷贝到go源文件（比如index.go）中，在对应文件夹执行go run index.go即可运行sha256程序。

package main  
  
import (  
 "fmt"  
 "runtime"  
 "strconv"  
 "time"  
)  
  
type RETURNTYPE uint32  
  
const (  
 NOT RETURNTYPE = 0  
 THIRTY RETURNTYPE = 1  
 ThirtyOne RETURNTYPE = 2  
 ThirtyTwo RETURNTYPE = 3  
)  
  
*// precache存储若干个uint8数组，其中每一个数组存储特定个数的0，方便使用append函数进行0的填充*var precache [][]uint8  
  
*// behcache存储若干个uint8数组，功能和precache类似*var behcache [][]uint8  
  
func preProcessing(inarr \*[]uint8) {  
 arrlen := (uint64)(len(\*inarr))  
 *// 获得字符串长度，进而计算二进制长度* var binlen uint64 = arrlen \* 8  
 *// 取模* mod := binlen % 512  
 if mod <= 447 {  
 *// 如果mod <= 447，说明补上一位1之后取模运算不会超过448，那么直接将其补上要达到448所需的0即可* \*inarr = append(\*inarr, 0x80)  
 \*inarr = append(\*inarr, precache[(440-mod)/8]...)  
 } else {  
 *// 如果超过448，那么需要补充(512 - (mod - 448 - 8)) = 952 - mod位0，这里-8是因为0x80已经用去了8位* \*inarr = append(\*inarr, 0x80)  
 \*inarr = append(\*inarr, behcache[(952-mod)/8]...)  
 }  
 *// 进行位运算，将长度对应的uint8值添加到消息末尾* var base uint64 = 0x00000000000000ff  
 for move := 0; move < 8; move++ {  
 \*inarr = append(\*inarr, uint8(binlen&(base<<(56-move\*8))>>(56-8\*move)))  
 }  
}  
func printUint8Array(charArr []uint8) {  
 count := 0  
 for i := range charArr {  
 fmt.Printf("%.2x", charArr[i])  
 count++  
 if count%4 == 0 {  
 fmt.Printf(" ")  
 }  
 if count%16 == 0 {  
 fmt.Printf("\n")  
 }  
 }  
}  
func printUint32Array(uint32Arr []uint32) {  
 count := 0  
 for i := range uint32Arr {  
 fmt.Printf("%.8x ", uint32Arr[i])  
 count++  
 if count%4 == 0 {  
 fmt.Printf("\n")  
 }  
 }  
}  
func compute\_chunk(hashval \*[]uint32, inputchunk \*[]uint32) {  
  
 *// sha256算法中定义的8个32位初始值，是对前64个质数立方根小数部分的表示的前32位* const\_head := []uint32{  
 0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,  
 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,  
 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,  
 0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,  
 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,  
 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,  
 0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,  
 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2,  
 }  
 x0 := (\*hashval)[0]  
 x1 := (\*hashval)[1]  
 x2 := (\*hashval)[2]  
 x3 := (\*hashval)[3]  
 x4 := (\*hashval)[4]  
 x5 := (\*hashval)[5]  
 x6 := (\*hashval)[6]  
 x7 := (\*hashval)[7]  
 *// 使用sha256中规定的算法迭代64次* for it := 0; it <= 63; it++ {  
 temp1 := const\_head[it] + (\*inputchunk)[it] + (((x4 >> 6) | (x4 << 26)) ^ ((x4 >> 11) | (x4 << 21)) ^ ((x4 >> 25) | (x4 << 7))) + ((x4 & x5) ^ ((^x4) & x6)) + x7  
 temp2 := (((x0 >> 2) | (x0 << 30)) ^ ((x0 >> 13) | (x0 << 19)) ^ ((x0 >> 22) | (x0 << 10))) + ((x0 & x1) ^ (x0 & x2) ^ (x1 & x2))  
 x7, x6, x5 = x6, x5, x4  
 x4 = x3 + temp1  
 x3, x2, x1 = x2, x1, x0  
 x0 = temp1 + temp2  
 }  
 *// 修改hasharr中存储的值* (\*hashval)[0] += x0  
 (\*hashval)[1] += x1  
 (\*hashval)[2] += x2  
 (\*hashval)[3] += x3  
 (\*hashval)[4] += x4  
 (\*hashval)[5] += x5  
 (\*hashval)[6] += x6  
 (\*hashval)[7] += x7  
}  
func myhash(inarr \*[]uint8) []uint32 {  
 *// sha256算法中定义的8个32位初始值，是对前8个质数平方根小数部分的表示的前32位* hashArr := []uint32{  
 0x6a09e667,  
 0xbb67ae85,  
 0x3c6ef372,  
 0xa54ff53a,  
 0x510e527f,  
 0x9b05688c,  
 0x1f83d9ab,  
 0x5be0cd19,  
 }  
 *// 之后，将输入的消息分割为一系列的chunk* arrlen := len(\*inarr)  
 *// 遍历每一个chunk，每一个chunk 512bits* for move := 0; move < arrlen/64; move++ {  
 *// 遍历每一个chunk，每一个chunk 512bits* temparr := make([]uint32, 64)  
 *// 构造chunk* for it := 0; it < 16; it++ {  
 temparr[it] = (uint32)((\*inarr)[move\*64+4\*it])<<24 | (uint32)((\*inarr)[move\*64+4\*it+1])<<16 | (uint32)((\*inarr)[move\*64+4\*it+2])<<8 | (uint32)((\*inarr)[move\*64+4\*it+3])  
 }  
 *// 使用递推公式计算chunk剩下的位数* for it := 16; it <= 63; it++ {  
 temparr[it] = temparr[it-16] + temparr[it-7] + (((temparr[it-15] >> 7) | (temparr[it-15] << 25)) ^ ((temparr[it-15] >> 18) | (temparr[it-15] << 14)) ^ (temparr[it-15] >> 3)) + (((temparr[it-2] >> 17) | (temparr[it-2] << 15)) ^ ((temparr[it-2] >> 19) | (temparr[it-2] << 13)) ^ (temparr[it-2] >> 10))  
 }  
 temp := make([]uint32, 8)  
 copy(temp, hashArr)  
 compute\_chunk(&hashArr, &temparr)  
 }  
 *// 退出循环时，hashArr中存储的值进行拼接即可得到对应的256位消息摘要* return hashArr  
}  
  
func verify(inputarr []uint32) RETURNTYPE {  
 *//printUint32Array(inputarr)* if inputarr[0]&0xfffffff8 != 0 {  
 return NOT  
 } else {  
 last := inputarr[0] & 0x00000007  
 if last == 0 {  
 return ThirtyTwo  
 } else if last < 2 {  
 return ThirtyOne  
 } else if last < 4 {  
 return THIRTY  
 }  
 }  
 return NOT  
}  
func search(base uint64) {  
 *//fmt.Println("enter base:", base)  
 //nonce := base* var nonce uint64 = base  
 for ; ; nonce++ {  
 var cur string = "Blockchain@ZhejiangUniversity" + strconv.FormatUint(nonce, 10)  
 *//fmt.Println(cur)* var charArr []uint8 = []uint8(cur)  
 preProcessing(&charArr)  
 sha256\_result := verify(myhash(&charArr))  
  
 switch sha256\_result {  
 case THIRTY:  
 {  
 var input string = "when nonce is" + strconv.FormatUint(nonce, 10) + " pre thirty bits is all 0"  
 timeStr := time.Now().Format("2006-01-02 15:04:05")  
 fmt.Println(timeStr, " ", input)  
 break  
 }  
 case ThirtyOne:  
 {  
 var input string = "when nonce is" + strconv.FormatUint(nonce, 10) + " pre thirty-one bits is all 0"  
 timeStr := time.Now().Format("2006-01-02 15:04:05")  
 fmt.Println(timeStr, " ", input)  
 break  
 }  
 case ThirtyTwo:  
 {  
 var input string = "when nonce is" + strconv.FormatUint(nonce, 10) + " pre thirty-two bits is all 0"  
 timeStr := time.Now().Format("2006-01-02 15:04:05")  
 fmt.Println(timeStr, " ", input)  
 break  
 }  
 }  
 if nonce == base+0x5F5E100 {  
 return  
 }  
 }  
}  
func init() {  
 timeStr := time.Now().Format("2006-01-02 15:04:05")  
 fmt.Println(timeStr, "begin searching")  
 for move := 0; move <= 440; move += 8 {  
 var temp []uint8 = make([]uint8, move/8)  
 for i := range temp {  
 temp[i] = 0  
 }  
 precache = append(precache, temp)  
 }  
  
 for move := 0; move <= 952; move += 8 {  
 var temp []uint8 = make([]uint8, (move)/8)  
 for i := range temp {  
 temp[i] = 0  
 }  
 behcache = append(behcache, temp)  
 }  
}  
func main() {  
 *//一个协程检查1亿的窗口  
 //0x3B9ACA00  
 //ffffffffffffffff  
 //var number uint64 = 0* var number uint64 = 0  
 for ; number <= 0xFFFFFFFFFA0A2000; number += 0x5F5E100 {  
 for runtime.NumGoroutine() >= 11 {  
 *//协程过多可能带来性能的下降* time.Sleep(10)  
 }  
 go search(number)  
 }  
}