

## 一、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。

## 二、实验环境

编程语言：Golang

系统：MacOS Monterey(Darwin Kernel Version 21.6.0)

## 三、实验步骤

### 1. Sha256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。

### 2. 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 // 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
2 var precache [][]uint8
3
4 // behcache存储若干个uint8数组，功能和precache类似
5 var behcache [][]uint8
```

```

6
7 func preProcessing(inarr *[]uint8) {
8     arrlen := (uint64)(len(*inarr))
9     // 获得字符串长度，进而计算二进制长度
10    var binlen uint64 = arrlen * 8
11    // 取模
12    mod := binlen % 512
13    if mod ≤ 447 {
14        // 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
15        *inarr = append(*inarr, 0x80)
16        *inarr = append(*inarr, precache[(440-mod)/8]...)
17    } else {
18        // 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
19        *inarr = append(*inarr, 0x80)
20        *inarr = append(*inarr, behcache[(952-mod)/8]...)
21    }
22    // 进行位运算，将长度对应的uint8值添加到消息末尾
23    var base uint64 = 0x00000000000000ff
24    for move := 0; move < 8; move++ {
25        *inarr = append(*inarr, uint8(binlen&(base<<(56-move*8))>>(56-8*move)))
26    }
27 }

```

## (2) 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 func compute_chunk(hashval *[]uint32, inputchunk *[]uint32) {
2
3     // 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
4     const_head := []uint32{
5         0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,

```

```

6      0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
7      0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
8      0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
9      0x27b70a85, 0x2e1b2138, 0x4d2c6dfe, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
10     0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
11     0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
12     0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befeafa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2,
13 }
14 x0 := (*hashval)[0]
15 x1 := (*hashval)[1]
16 x2 := (*hashval)[2]
17 x3 := (*hashval)[3]
18 x4 := (*hashval)[4]
19 x5 := (*hashval)[5]
20 x6 := (*hashval)[6]
21 x7 := (*hashval)[7]
22 // 使用sha256中规定的算法迭代64次
23 for it := 0; it ≤ 63; it++ {
24     temp1 := const_head[it] + (*inputchunk)[it] + (((x4 >> 6) | (x4 << 26)) ^ ((x4 >> 11) | (x4 << 21)) ^ ((x4 >> 25) |
(x4 << 7))) + ((x4 & x5) ^ ((^x4) & x6)) + x7
25     temp2 := (((x0 >> 2) | (x0 << 30)) ^ ((x0 >> 13) | (x0 << 19)) ^ ((x0 >> 22) | (x0 << 10))) + ((x0 & x1) ^ (x0 & x2)
^ (x1 & x2))
26     x7, x6, x5 = x6, x5, x4
27     x4 = x3 + temp1
28     x3, x2, x1 = x2, x1, x0
29     x0 = temp1 + temp2
30 }
31 // 修改hasharr中存储的值
32 (*hashval)[0] += x0
33 (*hashval)[1] += x1
34 (*hashval)[2] += x2
35 (*hashval)[3] += x3
36 (*hashval)[4] += x4
37 (*hashval)[5] += x5
38 (*hashval)[6] += x6
39 (*hashval)[7] += x7

```

```
40 }
41 func myhash(inarr *[]uint8) []uint32 {
42     // 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
43     hashArr := []uint32{
44         0x6a09e667,
45         0xbb67ae85,
46         0x3c6ef372,
47         0xa54ff53a,
48         0x510e527f,
49         0x9b05688c,
50         0x1f83d9ab,
51         0x5be0cd19,
52     }
53     // 之后, 将输入的消息分割为一系列的chunk
54     arrlen := len(*inarr)
55     // 遍历每一个chunk, 每一个chunk 512bits
56     for move := 0; move < arrlen/64; move++ {
57         // 遍历每一个chunk, 每一个chunk 512bits
58         temparr := make([]uint32, 64)
59         // 构造chunk
60         for it := 0; it < 16; it++ {
61             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])
62         }
63         // 使用递推公式计算chunk剩下的位数
64         for it := 16; it ≤ 63; it++ {
65             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))
66         }
67         temp := make([]uint32, 8)
68         copy(temp, hashArr)
69         compute_chunk(&hashArr, &temparr)
70     }
71     // 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
72     return hashArr
```

### (3) 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 func verify(inputarr []uint32) RETURNTYPE {
2     //printUint32Array(inputarr)
3     if inputarr[0]&0xffffffff8 != 0 {
4         return NOT
5     } else {
6         last := inputarr[0] & 0x00000007
7         if last == 0 {
8             return ThirtyTwo
9         } else if last < 2 {
10            return ThirtyOne
11        } else if last < 4 {
12            return THIRTY
13        }
14    }
15    return NOT
16 }
```

### (4) 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 type RETURNTYPE uint32
```

```
2
3 const (
4     NOT      RETURNTYPE = 0
5     THIRTY   RETURNTYPE = 1
6     ThirtyOne RETURNTYPE = 2
7     ThirtyTwo RETURNTYPE = 3
8 )
9 func search(base uint64) {
10     //fmt.Println("enter base:", base)
11     //nonce := base
12     var nonce uint64 = base
13     for ; ; nonce++ {
14         var cur string = "Blockchain@ZhejiangUniversity" + strconv.FormatUint(nonce, 10)
15         //fmt.Println(cur)
16         var charArr []uint8 = []uint8(cur)
17         preProcessing(&charArr)
18         sha256_result := verify(myhash(&charArr))
19
20         switch sha256_result {
21             case THIRTY:
22                 {
23                     var input string = "when nonce is" + strconv.FormatUint(nonce, 10) + " pre thirty bits is all 0"
24                     timeStr := time.Now().Format("2006-01-02 15:04:05")
25                     fmt.Println(timeStr, " ", input)
26                     break
27                 }
28             case ThirtyOne:
29                 {
30                     var input string = "when nonce is" + strconv.FormatUint(nonce, 10) + " pre thirty-one bits is all 0"
31                     timeStr := time.Now().Format("2006-01-02 15:04:05")
32                     fmt.Println(timeStr, " ", input)
33                     break
34                 }
35             case ThirtyTwo:
36                 {
37                     var input string = "when nonce is" + strconv.FormatUint(nonce, 10) + " pre thirty-two bits is all 0"
```

```

38         timeStr := time.Now().Format("2006-01-02 15:04:05")
39         fmt.Println(timeStr, " ", input)
40         break
41     }
42 }
43 if nonce == base+0x5F5E100 {
44     return
45 }
46 }
47 }

```

## (5) 主函数

在主函数中，程序会先自动执行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 func init() {
2     timeStr := time.Now().Format("2006-01-02 15:04:05")
3     fmt.Println(timeStr, "begin searching")
4     for move := 0; move ≤ 440; move += 8 {
5         var temp []uint8 = make([]uint8, move/8)
6         for i := range temp {
7             temp[i] = 0
8         }
9         precache = append(precache, temp)
10    }
11
12    for move := 0; move ≤ 952; move += 8 {
13        var temp []uint8 = make([]uint8, (move)/8)
14        for i := range temp {
15            temp[i] = 0
16        }
17        behcache = append(behcache, temp)
18    }
19 }

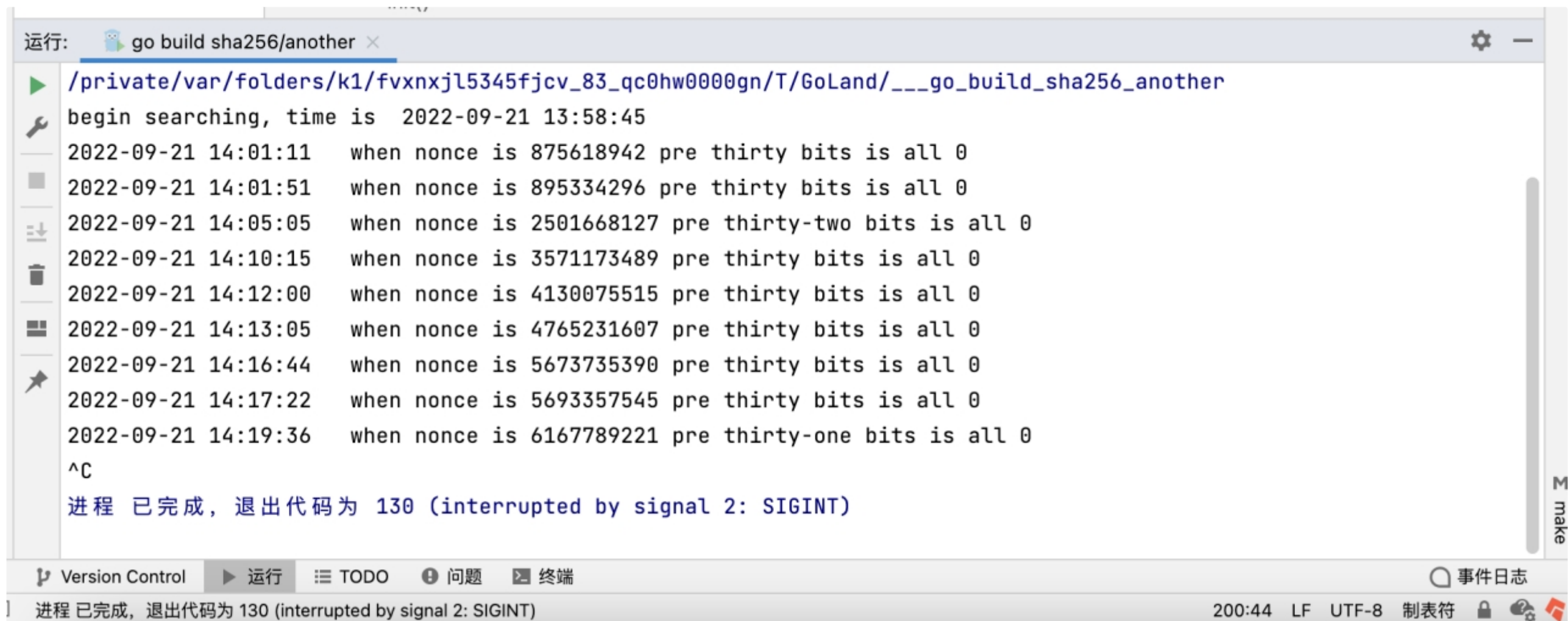
```

```
20 func main() {
21     //一个协程检查1亿的窗口
22     var number uint64 = 0
23     for ; number ≤ 0xFFFFFFFFFA0A2000; number += 0x5F5E100 {
24         for runtime.NumGoroutine() ≥ 11 {
25             //协程过多可能带来性能的下降
26             time.Sleep(10)
27         }
28         go search(number)
29     }
30 }
31
```

## 四、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。





```
运行: go build sha256/another x
/private/var/folders/k1/fvxn timer 5345fjcv_83_qc0hw0000gn/T/GoLand/___go_build_sha256_another
begin searching, time is 2022-09-21 13:58:45
2022-09-21 14:01:11 when nonce is 875618942 pre thirty bits is all 0
2022-09-21 14:01:51 when nonce is 895334296 pre thirty bits is all 0
2022-09-21 14:05:05 when nonce is 2501668127 pre thirty-two bits is all 0
2022-09-21 14:10:15 when nonce is 3571173489 pre thirty bits is all 0
2022-09-21 14:12:00 when nonce is 4130075515 pre thirty bits is all 0
2022-09-21 14:13:05 when nonce is 4765231607 pre thirty bits is all 0
2022-09-21 14:16:44 when nonce is 5673735390 pre thirty bits is all 0
2022-09-21 14:17:22 when nonce is 5693357545 pre thirty bits is all 0
2022-09-21 14:19:36 when nonce is 6167789221 pre thirty-one bits is all 0
^C
进程 已完成, 退出代码为 130 (interrupted by signal 2: SIGINT)
```

Version Control 运行 TODO 问题 终端 事件日志

进程 已完成, 退出代码为 130 (interrupted by signal 2: SIGINT) 200:44 LF UTF-8 制表符

## 五、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。

```
1
2
3 package main
4
5 import (
6     "fmt"
7     "runtime"
8     "strconv"
9     "time"
10 )
11
12 type RETURNTYPE uint32
13
```

```
14 const (
15     NOT     RETURNTYPE = 0
16     THIRTY   RETURNTYPE = 1
17     ThirtyOne RETURNTYPE = 2
18     ThirtyTwo RETURNTYPE = 3
19 )
20
21 // 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
22 var precache [][]uint8
23
24 // behcache存储若干个uint8数组，功能和precache类似
25 var behcache [][]uint8
26
27 func preProcessing(inarr []uint8) {
28     arrlen := (uint64)(len(inarr))
29     // 获得字符串长度，进而计算二进制长度
30     var binlen uint64 = arrlen * 8
31     // 取模
32     mod := binlen % 512
33     if mod ≤ 447 {
34         // 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
35         inarr = append(inarr, 0x80)
36         inarr = append(inarr, precache[(440-mod)/8]...)
37     } else {
38         // 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
39         inarr = append(inarr, 0x80)
40         inarr = append(inarr, behcache[(952-mod)/8]...)
41     }
42     // 进行位运算，将长度对应的uint8值添加到消息末尾
43     var base uint64 = 0x00000000000000ff
44     for move := 0; move < 8; move++ {
45         inarr = append(inarr, uint8(binlen&(base<<(56-move*8))>>(56-8*move)))
46     }
47 }
48 func printUint8Array(charArr []uint8) {
49     count := 0
```

```
50 for i := range charArr {
51     fmt.Printf("%.2x", charArr[i])
52     count++
53     if count%4 == 0 {
54         fmt.Printf(" ")
55     }
56     if count%16 == 0 {
57         fmt.Printf("\n")
58     }
59 }
60 }
61 func printUint32Array(uint32Arr []uint32) {
62     count := 0
63     for i := range uint32Arr {
64         fmt.Printf("%.8x ", uint32Arr[i])
65         count++
66         if count%4 == 0 {
67             fmt.Printf("\n")
68         }
69     }
70 }
71 func compute_chunk(hashval []uint32, inputchunk []uint32) {
72
73     // 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
74     const_head := []uint32{
75         0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
76         0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
77         0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
78         0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
79         0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
80         0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
81         0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
82         0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2,
83     }
84     x0 := (hashval)[0]
85     x1 := (hashval)[1]
```

```

86  x2 := (hashval)[2]
87  x3 := (hashval)[3]
88  x4 := (hashval)[4]
89  x5 := (hashval)[5]
90  x6 := (hashval)[6]
91  x7 := (hashval)[7]
92  // 使用sha256中规定的算法迭代64次
93  for it := 0; it ≤ 63; it++ {
94      temp1 := const_head[it] + (inputchunk)[it] + (((x4 >> 6) | (x4 << 26)) ^ ((x4 >> 11) | (x4 << 21)) ^ ((x4 >> 25) | (x4 <<
7))) + ((x4 & x5) ^ ((^x4) & x6)) + x7
95      temp2 := (((x0 >> 2) | (x0 << 30)) ^ ((x0 >> 13) | (x0 << 19)) ^ ((x0 >> 22) | (x0 << 10))) + ((x0 & x1) ^ (x0 & x2) ^
(x1 & x2))
96      x7, x6, x5 = x6, x5, x4
97      x4 = x3 + temp1
98      x3, x2, x1 = x2, x1, x0
99      x0 = temp1 + temp2
100 }
101 // 修改hasharr中存储的值
102 (hashval)[0] += x0
103 (hashval)[1] += x1
104 (hashval)[2] += x2
105 (hashval)[3] += x3
106 (hashval)[4] += x4
107 (hashval)[5] += x5
108 (hashval)[6] += x6
109 (hashval)[7] += x7
110 }
111 func myhash(inarr []uint8) []uint32 {
112     // 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
113     hashArr := []uint32{
114         0x6a09e667,
115         0xbb67ae85,
116         0x3c6ef372,
117         0xa54ff53a,
118         0x510e527f,
119         0x9b05688c,

```

```

120     0x1f83d9ab,
121     0x5be0cd19,
122 }
123 // 之后, 将输入的消息分割为一系列的chunk
124 arrlen := len(inarr)
125 // 遍历每一个chunk, 每一个chunk 512bits
126 for move := 0; move < arrlen/64; move++ {
127     // 遍历每一个chunk, 每一个chunk 512bits
128     temparr := make([]uint32, 64)
129     // 构造chunk
130     for it := 0; it < 16; it++ {
131         temparr[it] = (uint32)((inarr)[move64+4it])<<24 | (uint32)((inarr)[move64+4it+1])<<16 | (uint32)((inarr)[move64+4it+2])
<<8 | (uint32)((inarr)[move64+4it+3])
132     }
133     // 使用递推公式计算chunk剩下的位数
134     for it := 16; it ≤ 63; it++ {
135         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))
136     }
137     temp := make([]uint32, 8)
138     copy(temp, hashArr)
139     compute_chunk(&hashArr, &temparr)
140 }
141 // 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
142 return hashArr
143 }
144
145 func verify(inputarr []uint32) RETURNTYPE {
146     //printUint32Array(inputarr)
147     if inputarr[0]&0xffffffff8 ≠ 0 {
148         return NOT
149     } else {
150         last := inputarr[0] & 0x00000007
151         if last = 0 {
152             return ThirtyTwo

```

```
153     } else if last < 2 {
154         return ThirtyOne
155     } else if last < 4 {
156         return THIRTY
157     }
158 }
159 return NOT
160 }
161 func search(base uint64) {
162     //fmt.Println("enter base:", base)
163     //nonce := base
164     var nonce uint64 = base
165     for ; ; nonce++ {
166         var cur string = "Blockchain@ZhejiangUniversity" + strconv.FormatUint(nonce, 10)
167         //fmt.Println(cur)
168         var charArr []uint8 = []uint8(cur)
169         preProcessing(&charArr)
170         sha256_result := verify(myhash(&charArr))
171
172         switch sha256_result {
173         case THIRTY:
174             {
175                 var input string = "when nonce is" + strconv.FormatUint(nonce, 10) + " pre thirty bits is all 0"
176                 timeStr := time.Now().Format("2006-01-02 15:04:05")
177                 fmt.Println(timeStr, " ", input)
178                 break
179             }
180         case ThirtyOne:
181             {
182                 var input string = "when nonce is" + strconv.FormatUint(nonce, 10) + " pre thirty-one bits is all 0"
183                 timeStr := time.Now().Format("2006-01-02 15:04:05")
184                 fmt.Println(timeStr, " ", input)
185                 break
186             }
187         case ThirtyTwo:
188             {
```

```
189     var input string = "when nonce is" + strconv.FormatUint(nonce, 10) + " pre thirty-two bits is all 0"
190     timeStr := time.Now().Format("2006-01-02 15:04:05")
191     fmt.Println(timeStr, " ", input)
192     break
193 }
194 }
195 if nonce == base+0x5F5E100 {
196     return
197 }
198 }
199 }
200 func init() {
201     timeStr := time.Now().Format("2006-01-02 15:04:05")
202     fmt.Println(timeStr, "begin searching")
203     for move := 0; move ≤ 440; move += 8 {
204         var temp []uint8 = make([]uint8, move/8)
205         for i := range temp {
206             temp[i] = 0
207         }
208         precache = append(precache, temp)
209     }
210
211     for move := 0; move ≤ 952; move += 8 {
212         var temp []uint8 = make([]uint8, (move)/8)
213         for i := range temp {
214             temp[i] = 0
215         }
216         behcache = append(behcache, temp)
217     }
218 }
219 func main() {
220     //一个协程检查1亿的窗口
221     //0x3B9ACA00
222     //fffffffffffffffffff
223     //var number uint64 = 0
224     var number uint64 = 0
```

```
225 for ; number ≤ 0xFFFFFFFFFA0A2000; number += 0x5F5E100 {
226     for runtime.NumGoroutine() ≥ 11 {
227         // 协程过多可能带来性能的下降
228         time.Sleep(10)
229     }
230     go search(number)
231 }
232 }
```