

《密码学基础原理》实验报告

课程: 密码学基础原理 实验名称: SHA1 算法

姓名: 陈钦 实验日期: 2022/11/5

学号: 2021131094 实验报告日期: 2022/11/5

班级: 区块链工程 213

教师评语:	成绩:
签名:	
日期:	

一、实验名称

SHA1 算法

二、实验环境（详细说明运行的系统、平台及代码等）

1. 系统: go version go1.19 windows/amd64
2. IDE : GoLand 2022.2.2

三、实验目的

- (1) 加深对 *SHAI* 算法的理解;
- (2) 阅读标准和文献, 提高自学能力;
- (3) 加深对模块化设计的理解, 提高编程实践能力。

四、实验内容、步骤及结果

1. 实验内容

根据 SHA1 算法的原理，编写程序进行计算

2. 实验步骤

- (1) 首先将我们的输入转换成二进制串

```

func stringToBinary(str string) string { //字符串转二进制如: "abc"==>011000010110001001100011
    result := ""
    sliceTep := make([]string, len(str))
    for i := 0; i < len(str); i++ {
        b := str[i] //a,char
        x := int64(b) //a的ASC码值,int64
        y := strconv.FormatInt(x, base: 2) //a的ASC码值的二进制,string
        if len(y) != 8 {
            temp := string(y)
            for j := 0; j < 8-len(y); j++ { //看看我们缺8-len(y)个bit位
                sliceTep[i] = "0" + temp
            }
            result += sliceTep[i]
        }
    }
    return result
} //字符串转二进制如: "abc"==>011000010110001001100011

```

(2) 将该二进制串划分为若干个 512bit 块，存放到一个切片中（这里展示部分代码）

```

func div512BitBlock(str string) []string { //传入的字符串会被按照512bit来分成多个组，每个组512bit，用一维数组来装
    strLength := len(str)
    blockGroupNum := int(strLength / 512) //512组数目: blockGroupNum
    residue := strLength % 512 //不足512bit的部分长度
    fullFill512Bit_0 := str[512*blockGroupNum:] //取出不足512bit的部分

    //下面开始对这一部分进行补齐512bit
    //计算需要补0的个数
    full0Num := 0
    notEnough65 := 0
    if 512-residue < 65 { //如果剩余位置不足65位
        full0Num = 960 - (residue + 1) //需要0的个数: full0Num个
        notEnough65 = 1
    } else { //如果剩余位置够65位
        full0Num = 448 - (residue + 1) //需要0的个数: full0Num个
    }

    //对不足512bit的部分补1
    fullFill512Bit_1 := fullFill512Bit_0 + "1"
    //对不足512bit的部分补0
    fullFill512Bit_2 := fullFill512Bit_1
    for i := 0; i < full0Num; i++ {
        fullFill512Bit_2 += "0"
    }
}

```

(3) 预处理：将每一个 512bit 块扩展成 80 个 W 块（这里展示部分代码）

```

func preHandle_94(str string) [80]string { //预处理部分：将被一个512bit块扩展80个W，每个W用二进制表示
    //0 <= t <= 15块：照搬
    result := [80]string{}
    for i := 0; i < 16; i++ {
        result[i] = str[i*32 : (i+1)*32]
    }
    //16 <= t <= 79：根据公式计算
    for i := 16; i < 80; i++ {
        temp := make([]string, 0)
        result_ := [32]string{}
        for j := 0; j < 32; j++ {
            i_3, _ := strconv.Atoi(string(result[i-3][j]))
            i_8, _ := strconv.Atoi(string(result[i-8][j]))
            i_14, _ := strconv.Atoi(string(result[i-14][j]))
            i_16, _ := strconv.Atoi(string(result[i-16][j]))
            temp = append(temp, strconv.Itoa((i_3 ^ i_8 ^ i_14 ^ i_16)))
        }
        for k := 0; k < len(temp); k++ {
            result_[k] = temp[k]
        }
        //循环左移一位
        RESULT := ""
    }
}

```

(4) 进行 80 轮转换（这里展示部分代码）。N 个 80 轮转换的输出就是最终结果

```

func _80translate_94(a string, b string, c string, d string, e string, str [80]string, time int) string {
    fmt.Println(a, "====80轮转换输出如下====")
    fmt.Printf(binaryToHex(a) + ",")
    fmt.Printf(binaryToHex(b) + ",")
    fmt.Printf(binaryToHex(c) + ",")
    fmt.Printf(binaryToHex(d) + ",")
    fmt.Printf(binaryToHex(e) + ",")
    fmt.Println()

    //记录80轮转换之前的结果
    a_old := a
    b_old := b
    c_old := c
    d_old := d
    e_old := e

    for i := 0; i < 80; i++ {
        T := T_94(a, b, c, d, e, str[i], i)
        e = d
        d = c
        c = moveNBit(n: 30, b)
    }
}

```

(5) 一些细节：

对各种方法进行封装，成为 SHA1 方法

```
func SHA1_94(str1 string) {
    //将参数转换为二进制
    str2 := stringToBinary(str1)
    //将二进制划分为若干个512bit块，存放到一个切片中
    str3 := div512BitBlock(str2) //len(str3)是512bit块的数量
    //将每一个512bit扩展成80个W，用切片存放每一个80w
    str__ := "0110011101000101001000110000000011101111110011011010101110001001100010011101011011100111111100001000"
    fmt.Println("a...: "第1个512bit块的80轮转换: ")
    x := _80translate_94(str__[0:32],
        str__[32:64],
        str__[64:96],
        str__[96:128],
        str__[128:160], preHandle_94(str3[0]), time: 1)
    for i := 1; i < len(str3); i++ {
        fmt.Printf("format: "第%v个512bit块: \n", i+1)
        x = _80translate_94(x[0:32], //1-32
            x[32:64], //33-64
            x[64:96], //65-96
            x[96:128], //97-128
            x[128:160], //129-160
            preHandle_94(str3[i]), time: 0)
        fmt.Println(binaryToHex_(x))
    }
}
```

主要步骤（方法）:

```
func SHA1_94(str1 string) {...} //SHA1算法封装

func div512BitBlock(str string) []string {...} //传入的字符串会被按照512bit来分成多个组，每个组512bit，用一维数组来装

func stringToBinary(str string) string {...} //字符串转二进制如: "abc"==>011000010110001001100011

func preHandle_94(str string) [80]string {...} //预处理部分：将被一个512bit块分成80个W，每个W用二进制表示

func _80translate_94(a string, b string, c string, d string, e string, str [80]string, time int) string {...} //80轮转换
```

配套使用的方法（主要步骤需要用到的小步骤）:

```
180 func T_94(a string, b string, c string, d string, e string, Wt string, t int) string {...} //T函数
191 func f_94(x string, y string, z string, t int) string {...} //f函数
204 func Kt_94(t int) string {...} //Kt函数
217 func XOR_94(x string, y string) string {...} //两字符串异或
236 func reverse(str string) string {...} //将字符串取反
248 func and_94(x string, y string) string {...} //两字符串与运算
276 func moveNBit(n int, str string) string {...} //循环左移n位
303 func addModW(str01 string, str02 string, str03 string, str04 string, str05 string,
304 h0 int, h1 int, h2 int, h3 int, h4 int) string {...} //mode 2^32加法
322 func add(str01 int, str02 int) string {...} //相加并取模,w位的数，就取模2^w
337 func stringToInt(str string) int {...} // 现在我们的问题是：如何将string转为int 。比如"101"，要转为int的0b101
402 func binaryToHex(str string) string {...} //二进制转十六进制
442 func binaryToHex_(str string) string {...} //二进制转十六进制
```

3.实验结果

测试 1:

```
func main() {
    SHA1_94(str1: "abc")
    //SHA1_94("2021131094")
    //SHA1_94("abcbcbcdcedefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq")
}
```

实验要求的输出

第 1 个 512bit 块的 80 轮转换：

===== 扩展的 80 个 W 块开始输出 =====

W[0]=61626380

W[1]=00000000

W[14]=00000000

W[15]=00000018

W[16]=C2C4C700

W[79]=822E0879

===== 扩展的 80 个 W 块输出结束 =====

===== 80 轮转换输出如下 =====

SHA1 算法的最终结果输出

```
go build crypto_lab_03.go x
3F2588C2,497093C0,DE37534A,030F7CAD,4405957E,[69]
C199F8C7,3F2588C2,125C24F0,DE37534A,030F7CAD,[70]
39859DE7,C199F8C7,8FC96230,125C24F0,DE37534A,[71]
EDB42DE4,39859DE7,F0667E31,8FC96230,125C24F0,[72]
11793F6F,EDB42DE4,CE616779,F0667E31,8FC96230,[73]
5EE76897,11793F6F,3B6D0B79,CE616779,F0667E31,[74]
63F7DAB7,5EE76897,C45E4FDB,3B6D0B79,CE616779,[75]
A079B7D9,63F7DAB7,D7B9DA25,C45E4FDB,3B6D0B79,[76]
860D21CC,A079B7D9,D8FD6AD,D7B9DA25,C45E4FDB,[77]
5738D5E1,860D21CC,681E6DF6,D8FD6AD,D7B9DA25,[78]
42541B35,5738D5E1,21834873,681E6DF6,D8FD6AD,[79] 正确

abc的SHA1加密结果为:
A9993E364706816ABA3E25717850C26C9CD0D89D
```

t = 78 : 5738d5e1 860d21cc 681e6df6 d8fdf6ad d7b9de
t = 79 : 42541b35 5738d5e1 21834873 681e6df6 d8fdf6

That completes the processing of the first and only message block, $M^{(1)}$. The final hash $H^{(1)}$ is calculated to be

$$H_0^{(1)} = 67452301 + 42541b35 = a9993e36$$

$$H_1^{(1)} = efcdab89 + 5738d5e1 = 4706816a$$

$$H_2^{(1)} = 98badcfe + 21834873 = ba3e2571$$

$$H_3^{(1)} = 10325476 + 681e6df6 = 7850c26c$$

$$H_4^{(1)} = c3d2e1f0 + d8fdf6ad = 9cd0d89d.$$

The resulting 160-bit message digest is

a9993e36 4706816a ba3e2571 7850c26c 9cd0d89d.

A.2 SHA-1 Example (Multi-Block Message)

Let the message, M , be the 448-bit ($\ell = 448$) ASCII string

"abcdbcdecdefdefgefghfghighijhijkijklklmklmnlmnomnopnopq".

测试 2:

```
func main() {
    //SHA1_94("abc")
    SHA1_94(str1: "2021131094")
    //SHA1_94("abcdbcdecdefdefgefghfghighijhijkijklklmklmnlmnomnopnopq")
}
```

实验要求的输出

第 1 个 512bit 块的 80 轮转换：

===== 扩展的 80 个 W 块开始输出 =====

W[0]=64C19316

W[1]=2CD8B072

W[14]=00000000

W[15]=00000046

W[16]=6D83262D

W[79]=2F8A8D4D

===== 扩展的 80 个 W 块输出结束 =====

===== 80 轮转换输出如下 =====

SHA1 算法的最终结果输出

```
F6073473,EA667CAC,27653D0E,F7B74869,1A22DE9F,[74]
813EB7C3,F6073473,3A999F2B,27653D0E,F7B74869,[75]
AE8998D5,813EB7C3,FD81CD1C,3A999F2B,27653D0E,[76]
0C9754BA,AE8998D5,E04FADF0,FD81CD1C,3A999F2B,[77]
6869BD84,0C9754BA,6BA26635,E04FADF0,FD81CD1C,[78]
8C216C4B,6869BD84,8325D52E,6BA26635,E04FADF0,[79]
```

2021131094的 SHA1加密结果为：

F3668F4C5837690D1BE0B22C7BD4BAABA4228FE0

测试 3:

```
func main() {
    //SHA1_94("abc")
    //SHA1_94("2021131094")
    SHA1_94( str1: "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq")
}
```

实验要求的输出

```
第 1个 512bit块的 80轮 转换：
=====扩展的 80个 W块 开始 输出 =====
W[0]=61626364
W[1]=62636465
W[14]=80000000
W[15]=00000000
W[16]=0A063A3E
W[79]=1FF69958
=====扩展的 80个 W块 输出 结束 =====
```

=====扩展的80个W块输出结束=====

$r = 78 :$	58c0aac0	da955482	74e9b89d	9af00b7f	74a36c
$r = 79 :$	906fd62c	58c0aac0	b6a55520	74e9b89d	9af00b7f

That completes the processing of the second and final message block, $M^{(2)}$. The final hash value $H^{(2)}$, is calculated to be

$$\begin{aligned}
 H_0^{(2)} &= f4286818 + 906fd62c = 84983e44 \\
 H_1^{(2)} &= c37b27ae + 58c0aac0 = 1c3bd26e \\
 H_2^{(2)} &= 0408f581 + b6a55520 = baae4aa1 \\
 H_3^{(2)} &= 84677148 + 74e9b89d = f95129e5 \\
 H_4^{(2)} &= 4a566572 + 9af00b7f = e54670f1.
 \end{aligned}$$

The resulting 160-bit message digest is

84983e44 1c3bd26e baae4aa1 f95129e5 e54670f1.

A.3 SHA-1 Example (Long Message)

Let the message M be the binary-coded form of the ASCII string which consists of 1,000 repetitions of the character "a". The resulting SHA-1 message digest is

34aa973c d4cd4aa4 f61eeb2b dbad2731 6534016f.

五、实验中的问题及心得

本人使用 go 语言进行 SHA1 实现，在实验要求的基础上（实验要求输出扩展的 80 个 W），再进一步探究 SHA1，完整实现了 SHA1。

SHA 算法主要分为四个大步骤：（1）将输入转换为二进制串，（2）将该二进制串划分为若干个 512bit 块，（3）预处理：将每一个 512bit 块扩展成 80 个 W 块，（4）进行 80 轮转换。其中步骤（3）的输出是本实验所要求的，步骤（4）为本人对 SHA1 加密算法的扩展探究。当然其中比较难的是步骤（4）

编写代码过程中遇到了各种各样的问题。比如：需要将二进制串转为十六进制输出，进制转换之后 int 会变成 string，还需要写方法来再变回 int。这种进制和类型之间的转换关系很是复杂。又如 8 轮转换中的 T 函数和 F 函数：这是两个关键的方法，我们要用到异或运算与运算等来编写，很关键，本人因为这里写错卡了很久。又如每一个加法：这里的加法是取模操作，要求是取模 2^{32} 。又如假如进行多轮的 80 轮转换该如何进行衔接？等等。。。

收获：（1）代码模块化编写的思想得到了很大的提高，模块化思想真的很重要！不仅在设计算法思路方面有帮助，进行 debug 调试找错误的时候也起到了很大的作用。（2）耐心得到了锻炼：整个实验三至少花费了 13 个小时，最终也是完成了。（3）对于复杂的项目，方法之间的关系梳理能力得到很大提高

注：详细的代码实现细节和思路已经注释到程序源代码当中

以上

附件：程序代码

```
package main

import (
    "fmt"
    "math"
    "strconv"
)

func main() {
    SHA1_94("abc")
    SHA1_94("2021131094")
}
```



```

    SHA1_94("abcdbcdecdefdefgefghfghighijhijkijklmklmnlmnomnopnopq")
}

func SHA1_94(str1 string) {
    //将参数转换为二进制
    str2 := stringToBinary(str1)
    //将二进制划分为若干个 512bit 块，存放到一个切片中
    str3 := div512BitBlock(str2) //len(str3)是 512bit 块的数量
    //将每一个 512bit 扩展成 80 个 W，用切片存放每一个 80w
    str__ :=
    "0110011101000101001000110000000111101111110011011010101110001001100110001011101
    01101110011111110000100000011001001010100011101101100001111010010111000011111000
    0"

    fmt.Println("第 1 个 512bit 块的 80 轮转换：")
    x := _80translate_94(str__[0:32],
        str__[32:64],
        str__[64:96],
        str__[96:128],
        str__[128:160], preHandle_94(str3[0]), 1)
    for i := 1; i < len(str3); i++ {
        fmt.Printf("第 %v 个 512bit 块：\n", i+1)
        x = _80translate_94(x[0:32], //1-32
            x[32:64], //33-64
            x[64:96], //65-96
            x[96:128], //97-128
            x[128:160], //129-160
            preHandle_94(str3[i]), 0)
        fmt.Println(binaryToHex_(x))
    }

    fmt.Printf("\n %v 的 SHA1 加密结果为:\n %v", str1, binaryToHex_(x))
} //SHA1 算法封装

```

```

func div512BitBlock(str string) []string { //传入的字符串会被按照 512bit 来分成多个组，每个
    组 512bit，用一维数组来装

```

```

strLength := len(str)
blockGroupNum := int(strLength / 512)           //512 组数目: blockGroupNum
residue := strLength % 512                       //不足 512bit 的部分长度
fullFill512Bit_0 := str[512*blockGroupNum:] //取出不足 512bit 的部分

//下面开始对这一部分进行补齐 512bit
//计算需要补 0 的个数
full0Num := 0
notEnough65 := 0
if 512-residue < 65 { //如果剩余位置不足 65 位
    full0Num = 960 - (residue + 1) //需要 0 的个数: full0Num 个
    notEnough65 = 1
} else { //如果剩余位置够 65 位
    full0Num = 448 - (residue + 1) //需要 0 的个数: full0Num 个
}
//对不足 512bit 的部分补 1
fullFill512Bit_1 := fullFill512Bit_0 + "1"
//对不足 512bit 的部分补 0
fullFill512Bit_2 := fullFill512Bit_1
for i := 0; i < full0Num; i++ {
    fullFill512Bit_2 += "0"
}
//添加 64bit 的数值: 内容长度
residueToBinary := strconv.FormatInt(int64(residue), 2) //将内容长度从十进制转为二进制
fullFill512Bit_3 := residueToBinary                     //声明等待完善的 64bit 部分
//前面要补 0
if len(residueToBinary) != 64 { //如果这个内容长度不足 64bit
    for j := 0; j < 64-len(residueToBinary); j++ { //看看我们缺 64-len(residueToBinary)个
bit 位
        fullFill512Bit_3 = "0" + fullFill512Bit_3
    }
}
fullFill512Bit_4 := fullFill512Bit_2 + fullFill512Bit_3 //不足 512bit 的部分补足完毕
//下面我们来将输入转换成的若干个 512bit 块输出, 输出到一个一维数组,这个一维数组

```

的每一个值就是 512bit 块中的二进制串

```
arr := make([]string, 0)
for i := 0; i < blockGroupNum+1+notEnough65; i++ { //每一个 512 块（不包括补的）
    arr = append(arr, fullFill512Bit_4[i*512:(i+1)*512])
}
//arr = append(arr, fullFill512Bit_4) //加上补的那块
return arr
} //传入的字符串会被按照 512bit 来分成多个组，每个组 512bit，用一维数组来装
```

func stringToBinary(str string) string { // 字符串转二进制如：
"abc"==>011000010110001001100011

```
    result := ""
    sliceTep := make([]string, len(str))
    for i := 0; i < len(str); i++ {
        b := str[i] //a,char
        x := int64(b) //a 的 ASC 码值,int64
        y := strconv.FormatInt(x, 2) //a 的 ASC 码值的二进制,string
        if len(y) != 8 {
            temp := string(y)
            for j := 0; j < 8-len(y); j++ { //看看我们缺 8-len(y)个 bit 位
                sliceTep[i] = "0" + temp
            }
            result += sliceTep[i]
        }
    }
    return result
} //字符串转二进制如: "abc"==>011000010110001001100011
```

func preHandle_94(str string) [80]string { //预处理部分：将被一个 512bit 块扩展 80 个 W，每个 W 用二进制表示

```
    //0 <= t <= 15 块：照搬
    result := [80]string{}
    for i := 0; i < 16; i++ {
```

```

        result[i] = str[i*32 : (i+1)*32]
    }
    //16 <= t <= 79: 根据公式计算
    for i := 16; i < 80; i++ {
        temp := make([]string, 0)
        result_ := [32]string{ }
        for j := 0; j < 32; j++ {
            i_3, _ := strconv.Atoi(string(result[i-3][j]))
            i_8, _ := strconv.Atoi(string(result[i-8][j]))
            i_14, _ := strconv.Atoi(string(result[i-14][j]))
            i_16, _ := strconv.Atoi(string(result[i-16][j]))
            temp = append(temp, strconv.Itoa((i_3 ^ i_8 ^ i_14 ^ i_16)))
        }
        for k := 0; k < len(temp); k++ {
            result_[k] = temp[k]
        }
        //循环左移一位
        RESULT := ""
        for i := 0; i < 32; i++ {
            RESULT += result_[i]
        }
        result[i] = moveNBit(1, RESULT)
    }
    //实验作业要求输出
    fmt.Println("=====扩展的 80 个 W 块开始输出=====")
    for i := 0; i < 80; i++ {
        if i == 0 || i == 1 || i == 14 || i == 15 || i == 16 || i == 79 {
            fmt.Printf("W[%v]=%v\n", i, binaryToHex(result[i]))
        }
    }

}

fmt.Println("=====扩展的 80 个 W 块输出结束=====")

return result
} //预处理部分：将被一个 512bit 块分成 80 个 W，每个 W 用二进制表示

```

```

func _80translate_94(a string, b string, c string, d string, e string, str [80]string, time int) string {
    fmt.Println("=====80 轮转换输出如下=====")
    fmt.Printf(binaryToHex(a) + ",")
    fmt.Printf(binaryToHex(b) + ",")
    fmt.Printf(binaryToHex(c) + ",")
    fmt.Printf(binaryToHex(d) + ",")
    fmt.Printf(binaryToHex(e) + ",")
    fmt.Println()

    //记录 80 轮转换之前的结果
    a_old := a
    b_old := b
    c_old := c
    d_old := d
    e_old := e

    for i := 0; i < 80; i++ {
        T := T_94(a, b, c, d, e, str[i], i)
        e = d
        d = c
        c = moveNBit(30, b)
        b = a
        a = T
        fmt.Printf(binaryToHex(a) + ",")
        fmt.Printf(binaryToHex(b) + ",")
        fmt.Printf(binaryToHex(c) + ",")
        fmt.Printf(binaryToHex(d) + ",")
        fmt.Printf(binaryToHex(e) + ",")
        fmt.Printf("[%v]\n", i)
    }
    if time == 1 {
        return addModW(a, b, c, d, e, 0b011001111010001010010001100000001,
0b11101111110011011010101110001001,

```

```

0b1001100010111010110011111110,
0b00010000001100100101010001110110, 0b11000011110100101110000111110000)
    } else {
        return addModW(a, b, c, d, e, stringToInt(a_old), stringToInt(b_old), stringToInt(c_old),
stringToInt(d_old), stringToInt(e_old))
    }

} //80 轮转换

```

```

func T_94(a string, b string, c string, d string, e string, Wt string, t int) string {
    a_ := moveNBit(5, a)
    f := f_94(b, c, d, t)
    Kt := Kt_94(t)
    //return a_ + f + e + Kt + Wt
    a1 := add(stringToInt(a_), stringToInt(f)) //a_ + f
    a2 := add(stringToInt(a1), stringToInt(e)) //a_ + f + e
    a3 := add(stringToInt(a2), stringToInt(Kt)) //a_ + f + e + Kt
    a4 := add(stringToInt(a3), stringToInt(Wt)) //a_ + f + e
    return a4
} //T 函数

```

```

func f_94(x string, y string, z string, t int) string { //输入是二进制串，共 32bit
    if 0 <= t && t <= 19 {
        return XOR_94(and_94(x, y), and_94(reverse(x), z))
    } else if 20 <= t && t <= 39 {
        return XOR_94(XOR_94(x, y), z)
    } else if 40 <= t && t <= 59 {
        return XOR_94(XOR_94(and_94(x, y), and_94(x, z)), and_94(y, z))
    } else if 60 <= t && t <= 79 {
        return XOR_94(XOR_94(x, y), z)
    } else {
        return "-1"
    }
} //f 函数

```

```

func Kt_94(t int) string {
    if 0 <= t && t <= 19 {
        return "5a827999"
    } else if 20 <= t && t <= 39 {
        return "6ed9eba1"
    } else if 40 <= t && t <= 59 {
        return "8f1bbcdc"
    } else if 60 <= t && t <= 79 {
        return "ca62c1d6"
    } else {
        return "-1"
    }
} //Kt 函数

```

```

func XOR_94(x string, y string) string { //两字符串异或,每个字符串 32bit
    temp := make([]string, 0)
    for j := 0; j < 32; j++ {
        if j == 31 {
            a, _ := strconv.Atoi(string(x[31]))
            b, _ := strconv.Atoi(string(y[31]))
            temp = append(temp, strconv.Itoa(a^b))
            continue
        }
        a, _ := strconv.Atoi(x[j : j+1]) //将每个数字取出， string=>int
        b, _ := strconv.Atoi(y[j : j+1])
        temp = append(temp, strconv.Itoa(a^b)) //取出的数字异或，然后添加到切片。
    }
    int=>string
    }
    var str string = "" //用于输出
    for k := 0; k < len(temp); k++ {
        str += temp[k]
    }
    return str
}

```

```
} //两字符串异或
```

```
func reverse(str string) string {  
    var result string = ""  
    for i := 0; i < 32; i++ {  
        a, _ := strconv.Atoi(str[i : i+1])  
        if a == 0 {  
            result += "1"  
        } else {  
            result += "0"  
        }  
    }  
    return result  
} //将字符串取反
```

```
func and_94(x string, y string) string {  
    //1010 and 1101 =  
    temp := make([]string, 0)  
    for j := 0; j < 32; j++ {  
        if j == 31 {  
            a, _ := strconv.Atoi(string(x[31]))  
            b, _ := strconv.Atoi(string(y[31]))  
            c := 0  
            if a == 1 && b == 1 {  
                c = 1  
            }  
            temp = append(temp, strconv.Itoa(c))  
            continue  
        }  
        a, _ := strconv.Atoi(x[j : j+1]) //将每个数字取出， string=>int  
        b, _ := strconv.Atoi(y[j : j+1])  
        c := 0  
        if a == 1 && b == 1 {  
            c = 1  
        }  
    }  
    return strings.Join(temp, "")  
}
```



```

    }
    temp = append(temp, strconv.Itoa(c)) //取出的数字异或，然后添加到切片。int=>string
}
var str string = "" //用于输出
for k := 0; k < len(temp); k++ {
    str += temp[k]
}
return str
} //两字符串与运算

```

```

func moveNBit(n int, str string) string {
    /*//左移实现
    var result string = ""
    var j int = n
    for i := 0; i < len(str)-n; i++ {
        result += str[j : j+1]
        j++
    }
    for i := 0; i < n; i++ {
        result += "0"
    }
    return result
    */

```

```

//循环左移 n 位实现
//10010=>01010:循环左移两位
var result string = ""
var j int = n
for i := 0; i < len(str)-n; i++ {
    result += str[j : j+1]
    j++
}
for i := 0; i < n; i++ {
    result += str[i : i+1]
}

```

```

    }

    return result
} //循环左移 n 位

func addModW(str01 string, str02 string, str03 string, str04 string, str05 string,
    h0 int, h1 int, h2 int, h3 int, h4 int) string {
    /*
        a := 0b011001111010001010010001100000001
        b := 0b11101111110011011010101110001001
        c := 0b10011000101110101101110011111110
        d := 0b00010000001100100101010001110110
        e := 0b11000011110100101110000111110000
    */
    //应该将 string 转成 int，然后相加，取模 2^32
    //现在我们的问题是：如何将 string 转为 int 。比如"101"，要转为 int 的 101
    H0_ := add(h0, stringToInt(str01))
    H1_ := add(h1, stringToInt(str02))
    H2_ := add(h2, stringToInt(str03))
    H3_ := add(h3, stringToInt(str04))
    H4_ := add(h4, stringToInt(str05))
    //return binaryToHex(H0_) + binaryToHex(H1_) + binaryToHex(H2_) + binaryToHex(H3_)
+ binaryToHex(H4_)
    return H0_ + H1_ + H2_ + H3_ + H4_
} //mode 2^32 加法

```

```

func add(str01 int, str02 int) string {
    s := str01 + str02
    x := strconv.FormatInt(int64(s), 2)
    c := x
    if len(x) == 32 { //如果没进位就不变
        c = x
    } else if len(x) == 33 { //如果进位就取模
        c = x[1:]
    } else { //不足 32 位要补位，前面补 0

```

```

        for i := 0; i < 32-len(x); i++ {
            c = "0" + c
        }
    }
    return c
} //相加并取模,w 位的数, 就取模 2^w

```

func stringToInt(str string) int { // 现在我们的问题是: 如何将 string 转为 int 。比如"101", 要转为十进制的 5

```

    var num float64 = 0
    var j int = 0

    //如果传入的是 Kt (十六进制)
    if len(str) == 8 {
        for i := len(str) - 1; i >= 0; i-- {
            if string(str[i]) == "1" {
                num += math.Pow(2, float64(j))
            }
            if string(str[i]) == "2" {
                num += 2 * math.Pow(2, float64(j))
            }
            if string(str[i]) == "3" {
                num += 3 * math.Pow(2, float64(j))
            }
            if string(str[i]) == "4" {
                num += 4 * math.Pow(2, float64(j))
            }
            if string(str[i]) == "5" {
                num += 5 * math.Pow(2, float64(j))
            }
            if string(str[i]) == "6" {
                num += 6 * math.Pow(2, float64(j))
            }
            if string(str[i]) == "7" {

```

```

        num += 7 * math.Pow(2, float64(j))
    }
    if string(str[i]) == "8" {
        num += 8 * math.Pow(2, float64(j))
    }
    if string(str[i]) == "9" {
        num += 9 * math.Pow(2, float64(j))
    }
    if string(str[i]) == "a" {
        num += 10 * math.Pow(2, float64(j))
    }
    if string(str[i]) == "b" {
        num += 11 * math.Pow(2, float64(j))
    }
    if string(str[i]) == "c" {
        num += 12 * math.Pow(2, float64(j))
    }
    if string(str[i]) == "d" {
        num += 13 * math.Pow(2, float64(j))
    }
    if string(str[i]) == "e" {
        num += 14 * math.Pow(2, float64(j))
    }
    if string(str[i]) == "f" {
        num += 15 * math.Pow(2, float64(j))
    }
    j = j + 4
}
return int(num)
}

```

```

for i := 31; i >= 0; i-- {
    if string(str[i]) == "1" {
        num += math.Pow(2, float64(j))
    }
}

```

```
    }  
    j++  
}  
  
return int(num)
```

} // 现在我们的问题是：如何将 string 转为 int 。比如"101"，要转为 int 的 0b101

```
func binaryToHex(str string) string {  
    result := ""  
    for i := 0; i < 8; i++ {  
        switch str[i*4 : i*4+4] {  
            case "0000":  
                result += "0"  
            case "0001":  
                result += "1"  
            case "0010":  
                result += "2"  
            case "0011":  
                result += "3"  
            case "0100":  
                result += "4"  
            case "0101":  
                result += "5"  
            case "0110":  
                result += "6"  
            case "0111":  
                result += "7"  
            case "1000":  
                result += "8"  
            case "1001":  
                result += "9"  
            case "1010":  
                result += "A"  
            case "1011":  
                result += "B"
```

```

        case "1100":
            result += "C"
        case "1101":
            result += "D"
        case "1110":
            result += "E"
        case "1111":
            result += "F"
        }
    }
    return result
} //二进制转十六进制

```

```

func binaryToHex_(str string) string {
    result := ""
    for i := 0; i < 40; i++ {
        switch str[i*4 : i*4+4] {
            case "0000":
                result += "0"
            case "0001":
                result += "1"
            case "0010":
                result += "2"
            case "0011":
                result += "3"
            case "0100":
                result += "4"
            case "0101":
                result += "5"
            case "0110":
                result += "6"
            case "0111":
                result += "7"
            case "1000":

```

```
        result += "8"
    case "1001":
        result += "9"
    case "1010":
        result += "A"
    case "1011":
        result += "B"
    case "1100":
        result += "C"
    case "1101":
        result += "D"
    case "1110":
        result += "E"
    case "1111":
        result += "F"
    }
}

return result
} //二进制转十六进制
```