DES加密解密go语言实现

```go
package main

import (
    "fmt"
)

// 是否为debug模式
var debug bool = false

// 二进制字符串转换至 Int, Left Low, Right High, Bigger模式输入
func BinStrToIntB(bin string) uint64 {
    var ret uint64 = 0
    length := len(bin)

    j := 0
    for i := 0; i < length; i++ {
        if bin[i] == ' ' {
            continue
        }
        ret |= (uint64)(bin[i]-'0') << j
        j++
    }
    return ret
}

// 以大端模式打印64位二进制bit位
func PrintInt64B(str string, num uint64) {
    // 反转bit位
    var out uint64 = 0
    t := 64
    for i := 0; i < t; i++ {
        out |= ((uint64)((num >> i) & 0x1)) << (t - i - 1)
    }
    fmt.Printf("%s : %064b\n", str, out)
}

// 以大端模式打印56位二进制bit位
func PrintInt56B(str string, num uint64) {
    // 反转bit位
    var out uint64 = 0
    t := 56
    for i := 0; i < t; i++ {
        out |= ((uint64)((num >> i) & 0x1)) << (t - i - 1)
    }
    fmt.Printf("%s : %056b\n", str, out)
}

// 以大端模式打印48位二进制bit位
func PrintInt48B(str string, num uint64) {
    // 反转bit位
    var out uint64 = 0
    t := 48
    for i := 0; i < t; i++ {
        out |= ((uint64)((num >> i) & 0x1)) << (t - i - 1)
    }
    fmt.Printf("%s : %048b\n", str, out)
}

// 以大端模式打印32位二进制bit
func PrintInt32B(str string, num uint32) {
    // 反转bit位
    var out uint64 = 0
    t := 32
    for i := 0; i < t; i++ {
        out |= ((uint64)((num >> i) & 0x1)) << (t - i - 1)
    }
    fmt.Printf("%s : %032b\n", str, out)
}

// 以大端模式打印28位二进制bit
func PrintInt28B(str string, num uint32) {
    // 反转bit位
    var out uint64 = 0
    t := 28
    for i := 0; i < t; i++ {
        out |= ((uint64)((num >> i) & 0x1)) << (t - i - 1)
    }
    fmt.Printf("%s : %028b\n", str, out)
}

// IP置换
/*
IP置换表中的值代表是第几bit的值，采用目前第几个bit索引值i1 进行查IP置换表后得到bit索引 i2，再根据这个i2查本身自己对应的该bit位，替换当前 i1位置的bit值。
*/
func IPRplace(num uint64) uint64 {
    var IPTable = [64]uint8{
        58, 50, 42, 34, 26, 18, 10, 2,
        60, 52, 44, 36, 28, 20, 12, 4,
        62, 54, 46, 38, 30, 22, 14, 6,
        64, 56, 48, 40, 32, 24, 16, 8,
        57, 49, 41, 33, 25, 17, 9, 1,
        59, 51, 43, 35, 27, 19, 11, 3,
        61, 53, 45, 37, 29, 21, 13, 5,
        63, 55, 47, 39, 31, 23, 15, 7}
    var out uint64 = 0
    for i := 0; i < 64; i++ {
        out |= (num >> (uint64(IPTable[i] - 1)) & 0x1) << i
    }
    return out
}

// E盒子拓展:
/*
   从32bit拓展到48bit
   将该eBox表中对于的bit位放入该索引bit位置中
*/
func E_Expand(num uint32) uint64 {
    var eBox = [48]uint8{
        32, 1, 2, 3, 4, 5,
        4, 5, 6, 7, 8, 9,
        8, 9, 10, 11, 12, 13,
        12, 13, 14, 15, 16, 17,
        16, 17, 18, 19, 20, 21,
        20, 21, 22, 23, 24, 25,
        24, 25, 26, 27, 28, 29,
        28, 29, 30, 31, 32, 1,
    }
    var out uint64 = 0
    for i := 0; i < 48; i++ {
        out |= (uint64)((num>>(eBox[i]-1))&1) << i
    }
    return out
}

/*
   S 盒压缩，将48bit压缩为32bit
   将48bit分为8组，每组6bit
   该组的第一个bit位与第6个bit位 组成S盒行号
   中间4bit位 组成S盒列号

   计算公式：
   r = b1 * 2 + b6
   c = b2 << 3 + b3 << 2 + b3 << 1 + b4

   根据行号和列好查询的到4bit的二进制数，对该二进制数进行大端处理即可完毕S盒
*/
func SBox(num uint64) uint32 {
    var sBox = [8][4][16]uint8{
        {{14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7},
            {0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8},
            {4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0},
            {15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13}},

        {{15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10},
            {3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5},
            {0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15},
            {13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9}},

        {{10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8},
            {13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1},
            {13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7},
            {1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12}},

        {{7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15},
            {13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9},
            {10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4},
            {3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14}},

        {{2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9},
            {14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6},
            {4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14},
            {11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3}},

        {{12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11},
            {10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8},
            {9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6},
            {4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13}},

        {{4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1},
            {13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6},
            {1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2},
            {6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12}},

        {{13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7},
            {1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2},
            {7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8},
            {2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11}},
    }

    var out uint32 = 0
    for i := 0; i < 8; i++ {
        b := (uint8)(num>>(i*6)) & 0x3f
        r := (b&1)<<1 + (b >> 5)
        c := ((b>>1)&1)<<3 + ((b>>2)&1)<<2 + ((b>>3)&1)<<1 + ((b >> 4) & 1)
        o := sBox[i][r][c]

        // 由于查表是小端模式，需要转换至大端
        var o2 uint8 = 0

        for j := 0; j < 4; j++ {
            o2 |= ((o >> j) & 1) << (3 - j)
        }

        out |= uint32(o2) << (i * 4)

        if debug == true {
            //fmt.Printf("b: %06b r: %d c: %d, o: %04b o2: %04b\n", b, r, c, o, o2)
        }
    }

    return out
}

/*
   P和置换
   与IP置换原理一样
*/
func PBox(num uint32) uint32 {
    var pTable = [32]uint8{
        16, 7, 20, 21,
        29, 12, 28, 17,
        1, 15, 23, 26,
        5, 18, 31, 10,
        2, 8, 24, 14,
        32, 27, 3, 9,
        19, 13, 30, 6,
        22, 11, 4, 25,
    }
    var out uint32 = 0
    for i := 0; i < 32; i++ {
        out |= (num >> (uint32(pTable[i] - 1)) & 0x1) << i
    }
    return out
```

```go
}

/*
    子密钥生成部分
    获取64bit的密钥后，经过PC1置换，获取56bit有效位，分成两组28bit，分别为C0，D0。
    C0，D0通过循环左移得到C1，D1，组装在一起，经过PC2置换得到第一轮子密钥。

    C1与D1经过循环左移，得到C2，D2,组装在一起，经过PC2置换得到第二轮密钥，以此类推，得到下一轮密钥。
*/

/*
    将64bit的密钥压缩生成56bit
*/
func PC1(num uint64) uint64 {
    var p1Table = [56]uint8{
        57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,
        19, 11, 3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
        7, 62, 54, 46, 38, 30, 22,
        14, 6, 61, 53, 45, 37, 29,
        21, 13, 5, 28, 20, 12, 4,
    }
    var out uint64 = 0
    for i := 0; i < 56; i++ {
        out |= (uint64)((num>>(p1Table[i]-1))&1) << i
    }
    return out
}

/*
    将56bit的密钥压缩生成48bit
*/
func PC2(num uint64) uint64 {
    var p2Table = [48]uint8{
        14, 17, 11, 24, 1, 5,
        3, 28, 15, 6, 21, 10,
        23, 19, 12, 4, 26, 8,
        16, 7, 27, 20, 13, 2,
        41, 52, 31, 37, 47, 55,
        30, 40, 51, 45, 33, 48,
        44, 49, 39, 56, 34, 53,
        46, 42, 50, 36, 29, 32,
    }
    var out uint64 = 0
    for i := 0; i < 48; i++ {
        out |= (uint64)(num>>(p2Table[i]-1)&1) << i
    }
    return out
}

/*
    循环左移
*/

func ShiftLeft(num uint32, times int) uint32 {
    if times > 16 || times < 1 {
        fmt.Println("ShiftLeft Error")
        return num
    }

    var shiftTable = [16]int{
        1, 1, 2, 2, 2, 2, 2, 2,
        1, 2, 2, 2, 2, 2, 2, 1,
    }
    // 由于在数值中，高位在左，低位在右，所以采用右移，在大端模式下是左移
    var out uint32 = num
    for i := 0; i < shiftTable[times-1]; i++ {
        h := num & 1 // 获取最低位
        out >>= 1
        out |= h << 27 // 低位补高位
    }
    return out
}

/*
   IP逆置换，经过16轮变换之后，得到64bit数据，最后一步是IP逆置换。
   IP逆置换正好是IP置换的逆。
*/

func InverseIPRplace(num uint64) uint64 {
    var IPTable = [64]uint8{
        40, 8, 48, 16, 56, 24, 64, 32,
        39, 7, 47, 15, 55, 23, 63, 31,
        38, 6, 46, 14, 54, 22, 62, 30,
        37, 5, 45, 13, 53, 21, 61, 29,
        36, 4, 44, 12, 52, 20, 60, 28,
        35, 3, 43, 11, 51, 19, 59, 27,
        34, 2, 42, 10, 50, 18, 58, 26,
        33, 1, 41, 9, 49, 17, 57, 25}

    var out uint64 = 0
    for i := 0; i < 64; i++ {
        out |= (num >> (uint64(IPTable[i] - 1)) & 0x1) << i
    }
    return out
}

// 单轮加密实现
// m 为R(i - 1), key为本轮次子密钥
func SingalRound(l uint32, r uint32, key uint64) uint32 {
    o := E_Expand(r)
    o ^= key
    so := SBox(o)
    so = PBox(so)
    return so ^ l
}

// 子密钥生成器
func DesKeyGen(key uint64) [16]uint64 {
    var out = [16]uint64{0}
    o := PC1(key)
    l := (uint32)(o & 0xfffffff) // 获取低28bit
    r := (uint32)(o >> 28)       // 获取高28bit
    for i := 1; i <= 16; i++ {
        l = ShiftLeft(l, i)
        r = ShiftLeft(r, i)
        o = uint64(l)
        o |= (uint64)(r << 28)
        o = PC2(o)
        out[i-1] = o
    }
    return out
}

// Des加密函数实现
func DesEncode(m []byte, key uint64) []byte {
    keys := DesKeyGen(key)
    out := make([]byte, 0)

    length := len(m)
    if length%8 != 0 { // 补充0
        for i := 0; i < (8 - (length % 8)); i++ {
            m = append(m, 0)
        }
        length = len(m)
    }

    // 每8字节进行加密
    for i := 0; i < (length / 8); i++ {
        var d uint64 = 0

        // 将8字节转化为uint64类型
        for j := 0; j < 8; j++ {
            var c uint8 = m[i*8+j] // 获取当前字节
            // 由是小端模式，需要转换至大端
            var o2 uint8 = 0
            for k := 0; k < 8; k++ {
                o2 |= ((c >> k) & 1) << (7 - k)
            }
            d |= uint64(o2) << (j * 8)
        }
        //fmt.Printf("o  : %064b\n", d)
        //PrintInt64B("m ", d)

        // IP置换
        o := IPRplace(d)
        l := uint32(o)
        r := uint32(o >> 32)
        t := uint32(0)

        // 轮加密
        for j := 0; j < 16; j++ {
            t = r
            r = SingalRound(l, r, keys[j])
            l = t
        }

        //PrintInt32B("l0 ", l)
        //PrintInt32B("r0", r)
        // 左右交换合并
        d = uint64(r)
        d |= uint64(l) << 32

        //PrintInt64B("r0", d)

        // IP逆向置换
        d = InverseIPRplace(d)
        //PrintInt64B("IpInverse: ", d)

        // 追加到Bytes

        // 将uint64转化为8字节
        for j := 0; j < 8; j++ {
            //var c uint8 = m[i * 8 + j] // 获取当前字节
            // 大端模式，需要转换至小端
            c := uint8(d >> (j * 8) & 0xff)
            var o2 uint8 = 0
            for k := 0; k < 8; k++ {
                o2 |= ((c >> k) & 1) << (7 - k)
            }
            out = append(out, o2)
            //d |= uint64(o2) << (j * 8)
        }
    }

    //GetUint64ByBytes(&m[8])

    //IPRplace()
    return out
}

// Des解密函数实现
func DesDecode(m []byte, key uint64) []byte {
    keys := DesKeyGen(key)
    out := make([]byte, 0)
    length := len(m)

    // 每8字节进行加密
    for i := 0; i < (length / 8); i++ {
        var d uint64 = 0

        // 将8字节转化为uint64类型
        for j := 0; j < 8; j++ {
            var c uint8 = m[i*8+j] // 获取当前字节
            // 由是小端模式，需要转换至大端
            var o2 uint8 = 0
            for k := 0; k < 8; k++ {
                o2 |= ((c >> k) & 1) << (7 - k)
            }
            d |= uint64(o2) << (j * 8)
        }

        // IP置换
        o := IPRplace(d)
        l := uint32(o)
        r := uint32(o >> 32)
        t := uint32(0)
        // 轮加密
        for j := 0; j < 16; j++ {
            t = r
            r = SingalRound(l, r, keys[15-j]) // 密钥顺序变化
            l = t
        }

        // 左右交换合并
        d = uint64(r)
        d |= uint64(l) << 32

        //PrintInt64B("r0", d)

        // IP逆向置换
        d = InverseIPRplace(d)
        //PrintInt64B("IpInverse: ", d)

        // 追加到Bytes

        // 将uint64转化为8字节
        for j := 0; j < 8; j++ {
            //var c uint8 = m[i * 8 + j] // 获取当前字节
            // 大端模式，需要转换至小端
            c := uint8(d >> (j * 8) & 0xff)
            var o2 uint8 = 0
            for k := 0; k < 8; k++ {
                o2 |= ((c >> k) & 1) << (7 - k)
            }
            out = append(out, o2)
            //d |= uint64(o2) << (j * 8)
        }
    }

    return out
}

func Test() {
    str := "01100011 01101111 01101101 01110000 01110101 01110100 01100101 01110010"

    subkey := "010100 000010 110010 101100 010101 000010 001101 000111"

    o := BinStrToIntB(str)

    key := BinStrToIntB(subkey)

    fmt.Printf("字符串: %s\n", str)
    fmt.Printf("Little: %064b\n", o)

    PrintInt64B("Input  ", o)
    o = IPRplace(o)
    PrintInt64B("IPTable ", o)

    l := uint32(o)
    r := uint32(o >> 32)
    PrintInt32B("l0 ", l)
    PrintInt32B("r0", r)
    PrintInt32B("E_Expand in  ", r)
    o = E_Expand(r)
    PrintInt48B("E_Expand out ", o)

    fmt.Println("\nS盒实现")
    PrintInt48B("key ", key)

    si := o ^ key
    PrintInt48B("SBox in ", si)

    so := SBox(si)
    PrintInt32B("SBox out ", so)

    po := PBox(so)
    PrintInt32B("PBox out ", po)

    l = l ^ po // 作为下一轮 l
    PrintInt32B("l1 ", l)

    fmt.Println("子密钥生成实现: 输入01234567")
    subkey = "00110000 00110001 00110010 00110011 00110100 00110101 00110110
00110111"
    key = BinStrToIntB(subkey)
    PrintInt64B("key ", key)
    o = PC1(key)
    PrintInt56B("PC1 ", o)
    l = (uint32)(o & 0xfffffff) // 获取低28bit
    r = (uint32)(o >> 28)       // 获取高28bit
    PrintInt28B("l ", l)
    PrintInt28B("r ", r)

    fmt.Println("循环左移")
    l = ShiftLeft(l, 1)
    PrintInt28B("l ", l)

    fmt.Println("IP逆置换")
    str = "11111111 10111000 01110110 01010111 00000000 11111111 00000110
10000011"
    o = BinStrToIntB(str)
    PrintInt64B("Input      ", o)
    o = InverseIPRplace(o)
    PrintInt64B("InverseIP  ", o)
}

func main() {
    //Test()
    data := []byte{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 'A', 'D'}
    var key uint64 = 0x1234
    fmt.Print("加密前data")
    fmt.Println(data)
    fmt.Printf("key: 0x%X\n", key)

    out := DesEncode(data, key)
    fmt.Println("加密后")
    fmt.Print(out)

    out = DesDecode(out, key)
    fmt.Printf("\n解密后")
    fmt.Print(out)
}
```