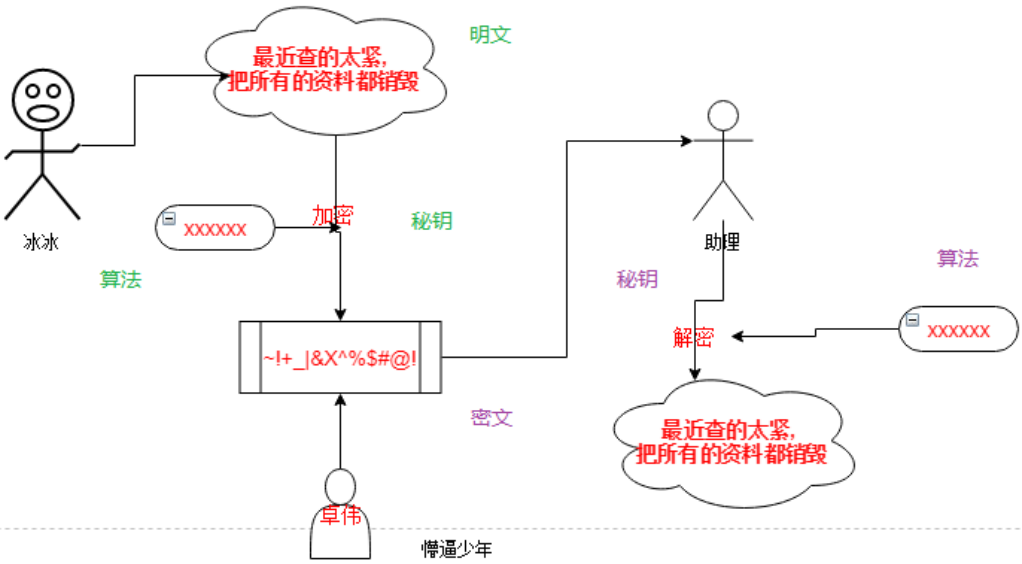
现代网络存在问题

1. 数据被窃取
2. 数据被篡改
3. 身份伪装

对称加密 ==》 非对称加密

对称加解密三要素



对称加密三要素：

1. 明文==》 1111
2. 秘钥==》 100
3. 算法==》加密算法1

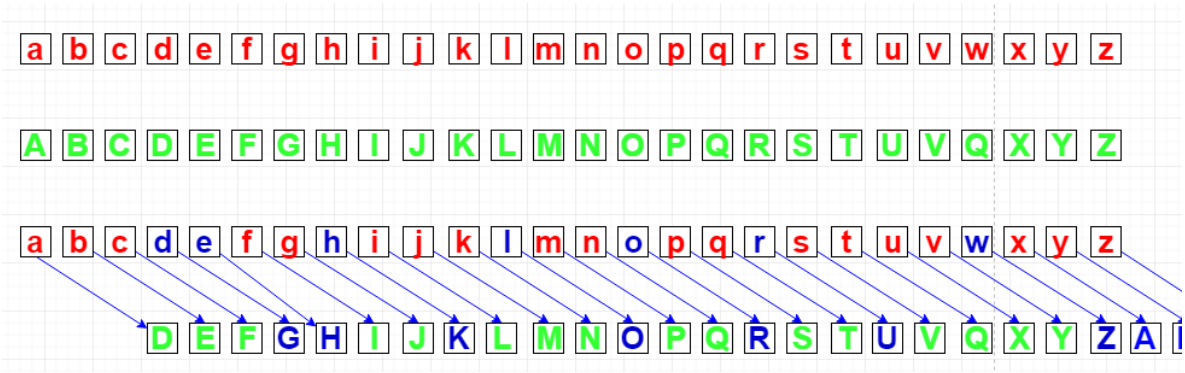
对称解密三要素：

1. 秘钥文==》 xxx&*^%$
2. 秘钥 ==>100
3. 算法 ==>解密算法，解密算法可能与加密算法相同，也可能不同

对称的意思：是指加密和解密的钥匙是相同的

非对称的意思：是指加密和解密的钥匙是不相同

凯撒密码



对称加密三要素：

1. 明文==》 hello world
2. 秘钥==》3
3. 算法==》向右移动

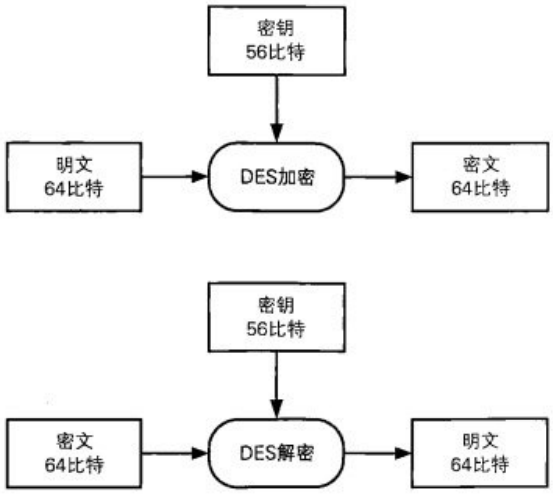
对称解密三要素：

1. 秘钥文==》 xxx&*^%$
2. 秘钥 ==> 3
3. 算法 ==> 向左移动

常用的加密算法

DES（Data Ecryption Standard）：数据加密标准

- **DES的加密与解密 - 图例**

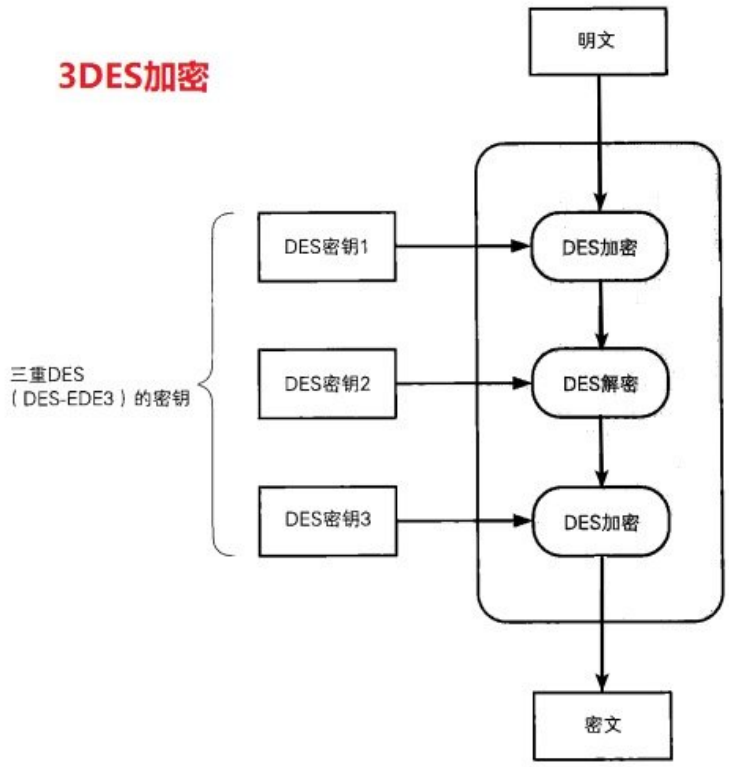


特点：

1. 已经不适用了，不安全

2. 秘钥长度：8字节（64 bit），在算法内部，每7 bit会自动提供一个校验位，真正参与加解密的只有56比特。
3. 对大文件进行加密时，需要对数据进行分组切割，每一个分组的长度是：8byte

3DES：Triple Data Encryption Standard: 三重DES



特点：

1. 进行了3次des操作
2. 加密过程：
    1. 加密=》解密=》加密
    2. 第二步，进行的解密，是为了与DES算法兼容（秘钥1 与秘钥2相同）
3. 解密过程：
    1. 解密=》加密=》解密
4. 秘钥长度：8 byte * 3 = 24 byte
5. 数据切割（分组）时，分组长度：8字节
6. 三个秘钥的关系：
    1. key1 与key2相同，或者key2与key3相同，此时是兼容des算法
    2. 如果key1、key2、key3各不相同，此时叫做3des-ede3
    3. 如果key1余key3相同，但是与key2，此时叫做3des-ed2
7. 过渡的算法，可以使用。

AES：Advanced Encrytion Standard：

特点：

1. 秘钥长度： 128（16字节），192（24字节）,256（32字节）
2. 分组长度：16字节（128比特）
3. 建议使用，效率高，加密更安全

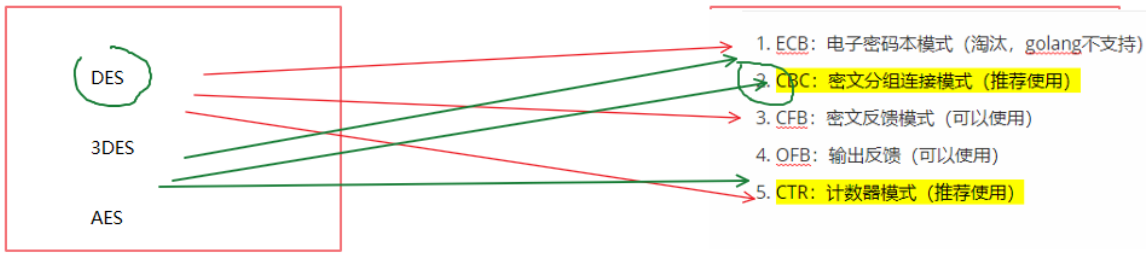
小结

算法	秘钥长度	分组	备注
DES	8字节	8字节	不要使用
3DES	24字节	8字节	可以使用
AES	16,24,32字节	16字节	建议使用

五种分组模式

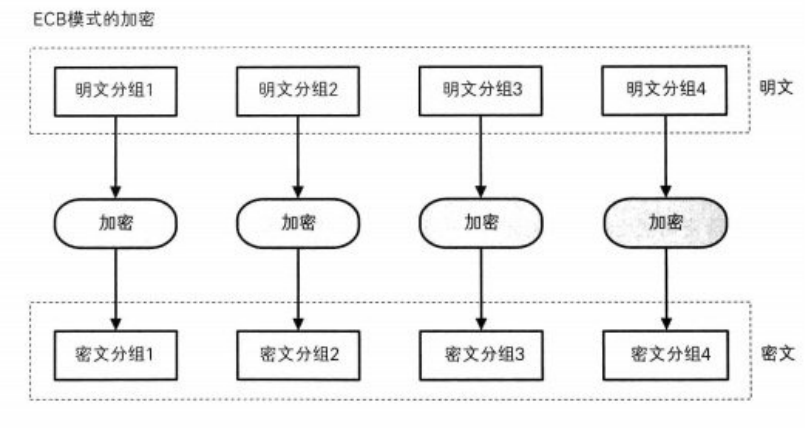
1. ECB：电子密码本模式（淘汰，golang不支持）
2. CBC：密文分组连接模式（推荐使用）
3. CFB：密文反馈模式（可以使用）
4. OFB：输出反馈（可以使用）
5. CTR：计数器模式（推荐使用）

密码算法与分组模式的关系



具体模式分析

1. ECB模式



特点：

1. 需要对明文进行切割，具体分组长度依赖加密算法：
    1. des==》 8字节分组
    2. 3des ==》 8字节
    3. aes ==》 16字节
2. 对每一个分组数据进行单独的加密：
    1. 效率高
    2. 可以通过输入有规律的输入得到有规律的输出：不安全，加密不彻底
3. 由于明文分组后直接加密，所以需要对不足分组长度进行尾部填充。

填充分析

异或操作

与：只要有0，结果为0 =》&

或：只有有1，结果为1 =》|

异或：相同为0，不同为1=》^

A：8， B：9

8：0000,1000

9：0000,1001 ^

0000,0001 ==>1

加密过程：

密文：8

算法：异或

秘钥：9

密文：1

解密过程：

密文：1

算法：异或

秘钥：9

1:0000,0001

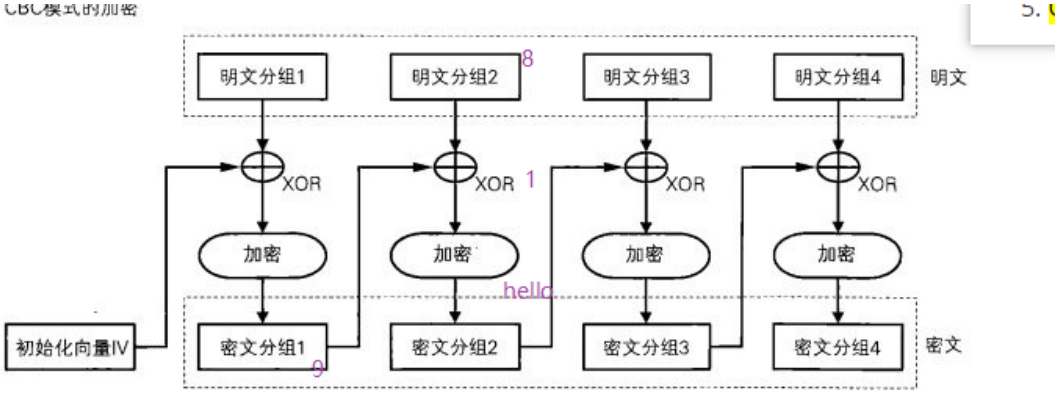
9:0000,1001 ^

0000,1000 ==> 8

小结：

1. 异或加密，秘钥相同，算法相同
2. 相同为0，不同为1（同龄人）

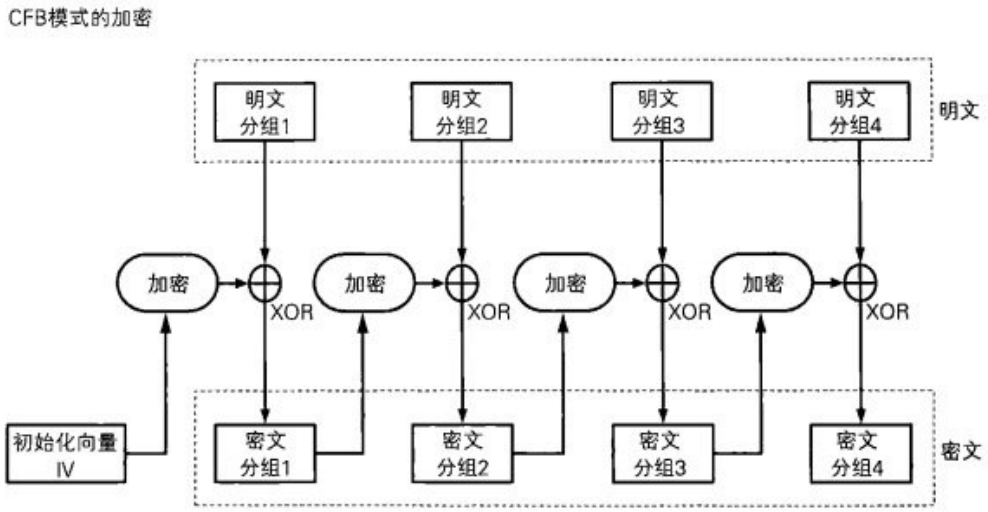
CBC分组模式



特点：

1. 需要分组，长度与选择的算法一致
2. 也需要对分组数据进行填充
3. 后一个分组的加密输入，依赖前一个加密的输出，所以不能并行加密
4. 先异或，后加密
5. 对于第一个分组，需要额外提供一个初始化向量iv（init vector），要求iv长度与分组长度一致
6. cbc模式推荐使用。

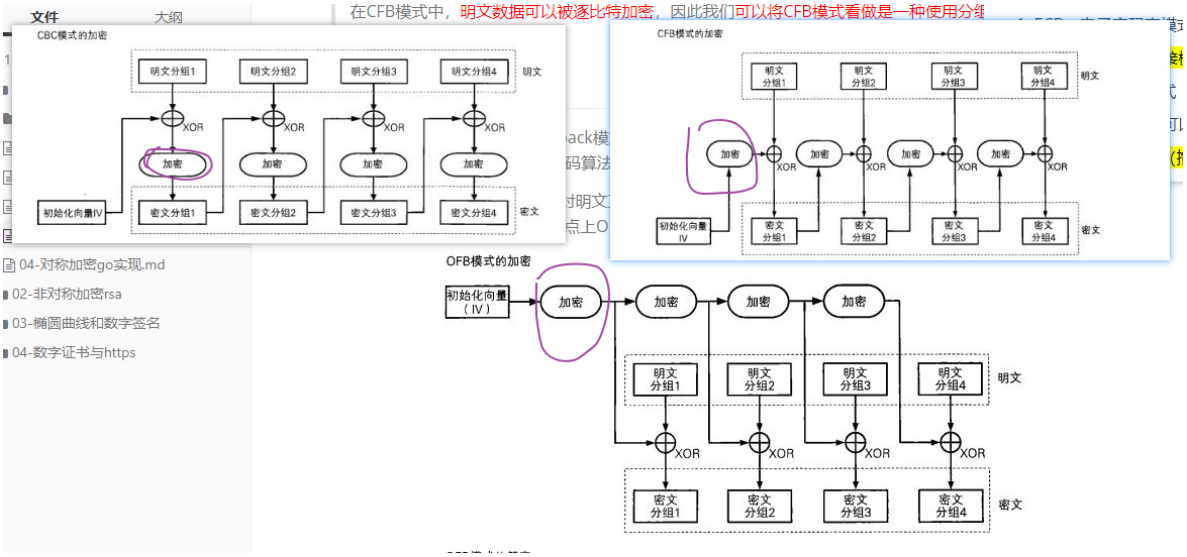
CFB分组模式



特点：

1. 需要分组，长度与选择算法一致
2. 不需要对分组数据进行填充
3. 先加密，后异或
4. 后一个分组的加密输入，依赖前一个加密的输出，所以不能并行加密
5. 对于第一个分组，需要额外提供一个初始化向量iv（init vector），要求iv长度与分组长度一致
6. 可以使用，但是推荐使用CTR模式

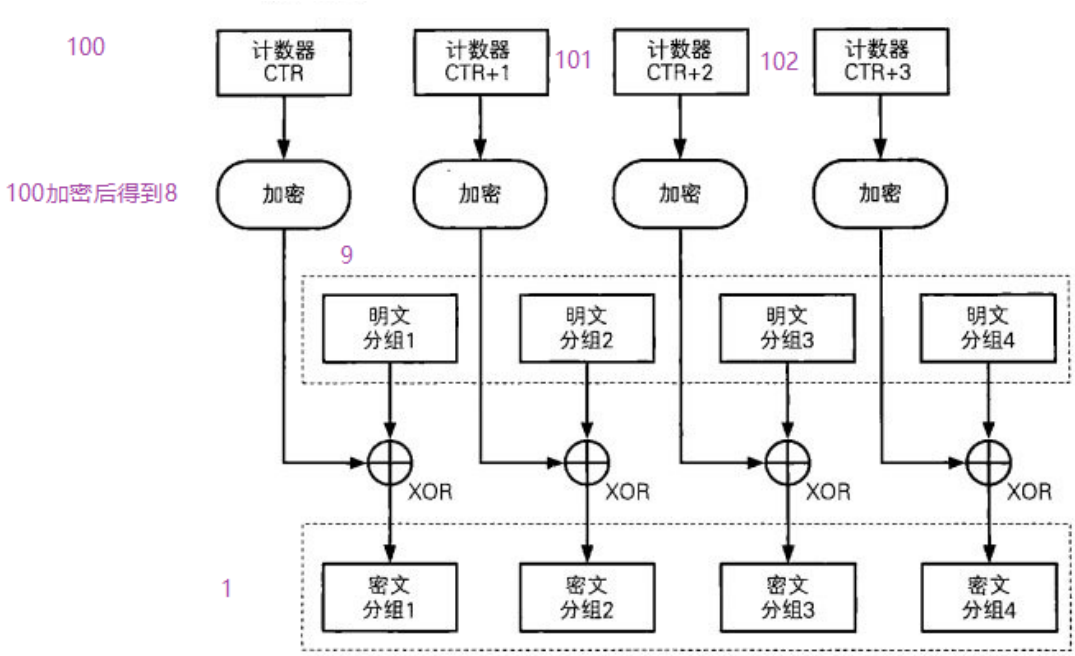
OFB模式



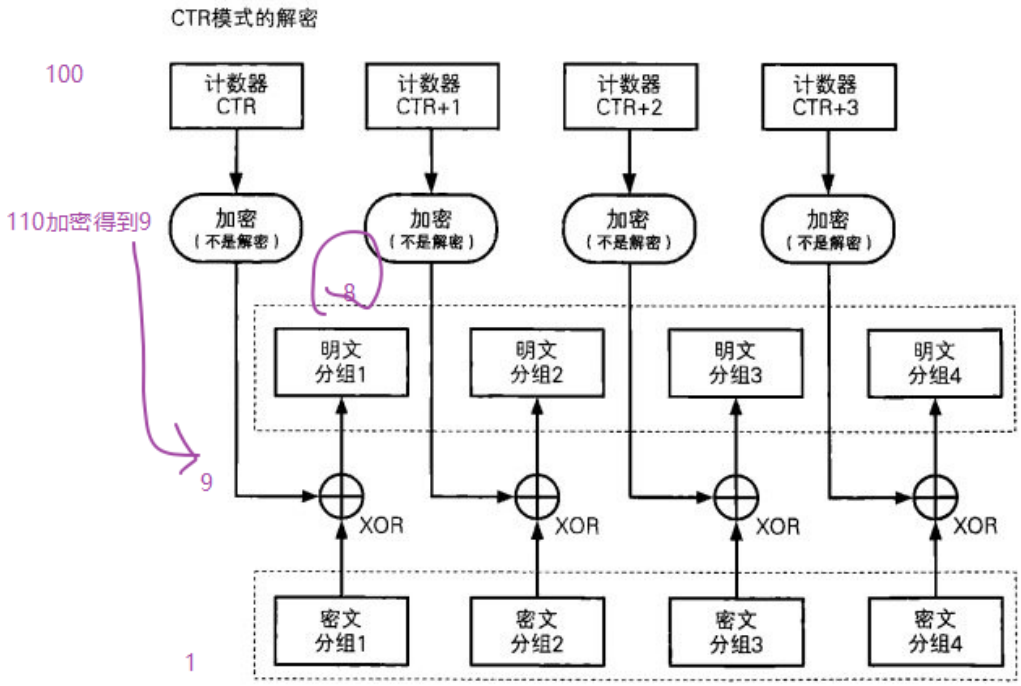
特点：

1. 分组数据不需要填充
2. 反复对初始向量的结果进行加密，作为下一次加密的输入
3. 可以使用，推荐使用CTR模式

CTR模式



密文：1



特点：

1. 需要提供一个数字，每个分组会自动加1
2. 不需要填充
3. 可以并行加密解密
4. 推荐使用

小结

> 我们已经介绍了ECB、CBC、CFB、OFB和CTR模式，下面我们对这些模式的特点做一下整理。

需要填充

模式	名称	优点	缺点	备注
ECB 模式	Electronic CodeBook 电子密码本模式	• 简单 • 快速 • 支持并行计算（加密、解密）	• 明文中的重复排列会反映在密文中 • 通过删除、替换密文分组可以对明文进行操作 • 对包含某些比特错误的密文进行解密时，对应的分组会出错 • 不能抵御重放攻击	不应使用
CBC 模式	Cipher Block Chaining 密文分组链接模式	• 明文的重复排列不会反映在密文中 • 支持并行计算（仅解密） • 能够解密任意密文分组	• 对包含某些错误比特的密文进行解密时，第一个分组的全部比特以及后一个分组的相应比特会出错 • 加密不支持并行计算	推荐使用
CFB 模式	Cipher-FeedBack 密文反馈模式	• 不需要填充（padding） • 支持并行计算（仅解密） • 能够解密任意密文分组	• 加密不支持并行计算 • 对包含某些错误比特的密文进行解密时，第一个分组的全部比特以及后一个分组的相应比特会出错 • 不能抵御重放攻击	• 现在已不使用 • 推荐用 CTR 模式代替
OFB 模式	Output-FeedBack 输出反馈模式	• 不需要填充（padding） • 可事先进行加密、解密的准备 • 加密、解密使用相同结构 • 对包含某些错误比特的密文进行解密时，只有明文中相对应的比特会出错	• 不支持并行计算 • 主动攻击者反转密文分组中的某些比特时，明文分组中相对应的比特也会被反转	推荐用 CTR 模式代替
CTR 模式	CounTeR 计数器模式	• 不需要填充（padding） • 可事先进行加密、解密的准备 • 加密、解密使用相同结构 • 对包含某些错误比特的密文进行解密时，只有明文中相对应的比特会出错 • 支持并行计算（加密、解密）	主动攻击者反转密文分组中的某些比特时，明文分组中相对应的比特也会被反转	推荐使用

代码认证

AES + CTR ==》 不需要填充

aes特点：

1. 秘钥长度： 128（16字节），192（字节）,256（32字节）
2. 分组长度：16字节（128比特）
3. 建议使用，效率高，加密更安全

ctr特点：

1. 需要提供一个数字，每个分组会自动加1
2. 不需要填充
3. 可以并行加密解密
4. 推荐使用

在线手册：https://studygolang.com/pkgdoc

实现加密逻辑

```
package main
```

```go
import (
    "bytes"
    "crypto/aes"
    "crypto/cipher"
    "fmt"
)

//加密函数 : key:秘钥， plainText :明文
func aesCtrEncrypt(key, plainText []byte) ([]byte, error) {
    fmt.Println("开始加密,明文:", string(plainText))

    //第一步：创建密码接口
    //func NewCipher(key []byte) (cipher.Block, error) {
    block, err := aes.NewCipher(key)
    if err != nil {
        fmt.Println("NewCipher err:", err)
        return nil, err
    }

    //创建初始向量（数字ctr），长度与算法分组长度一致
    //使用repeat，创建一个16个"1"的切片
    iv := bytes.Repeat([]byte("1"), block.BlockSize())

    fmt.Println("iv :", iv)

    //第二步：创建ctr分组
    s := cipher.NewCTR(block, iv)

    //创建一个切片，用于存储密文数据
    //src :数据来源
    //dst : 目的地
    dst := make([]byte, len(plainText))

    //第三步：加密处理
    s.XORKeyStream(dst, plainText)

    return dst, nil
}

//解密函数
func aesCtrDecrypt(key, cipherText []byte) ([]byte, error) {
    fmt.Println("开始解密,密文:", string(cipherText))
    //TODO

    return nil, nil
}

func main() {
    key := []byte("1234567887654321") //16字节秘钥
    plainText := []byte("你好，昌平!")

    cipherData, err := aesCtrEncrypt(key, plainText)
    if err != nil {
        fmt.Println("加密失败:", err)
        return
    }
```

```
    fmt.Printf("加密后的数据为, hex :%x\n", cipherData)
    fmt.Printf("加密后的数据为, string :%s\n", cipherData)

    plainText, err = aesCtrDecrypt(key, cipherData)
    if err != nil {
        fmt.Println("解密失败:", err)
        return
    }

    fmt.Printf("解密后的数据为:%s\n", plainText)
}
```

效果:

```
duke@DUKEDU51C6 MINGW64 /c/goprojects/src/go5期/02-密码
$ go run 01-aes-ctr.go
开始加密,明文: 你好, 昌平!
iv : [49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49]
加密后的数据为, hex :d8d4e2ba48f5ca605da8dadba94887bc
加密后的数据为, string :????H??]??H?
开始解密,密文: ????H??]??H?
解密后的数据为:

duke@DUKEDU51C6 MINGW64 /c/goprojects/src/go5期/02-密码
```

解密函数实现:

与加密逻辑一致, 直接复用

```
//解密函数
func aesCtrDecrypt(key, cipherText []byte) ([]byte, error) {
    fmt.Println("开始解密,密文:", string(cipherText))

    //加密过程与解密过程是一样的
    return aesCtrEncrypt(key, cipherText)
}
```

效果:

```
$ go run 01-aes-ctr.go
开始加密,明文: 你好, 昌平!
iv : [49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49]
加密后的数据为, hex :d8d4e2ba48f5ca605da8dadba94887bc
加密后的数据为, string :????H??]??H?
开始解密,密文: ????H??]??H?
开始加密,明文: ????H??]??H?
iv : [49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49]
解密后的数据为:你好, 昌平!
```

DES + CBC ==》需要填充

框架：

```
package main

import (
    "fmt"
)

//加密函数 : key:秘钥， plainText :明文
func desCbcEncrypt(key, plainText []byte) ([]byte, error) {
    fmt.Println("开始加密,明文:", string(plainText))

    return nil, nil
}

//解密函数
func desCbcDecrypt(key, cipherText []byte) ([]byte, error) {
    fmt.Println("开始解密,密文:", string(cipherText))

    return nil, nil
}

func main() {
    key := []byte("1234567887654321") //16字节秘钥
    plainText := []byte("你好，昌平!")

    cipherData, err := desCbcEncrypt(key, plainText)
    if err != nil {
        fmt.Println("加密失败:", err)
        return
    }

    fmt.Printf("加密后的数据为, hex :%x\n", cipherData)
    fmt.Printf("加密后的数据为, string :%s\n", cipherData)

    plainText, err = desCbcDecrypt(key, cipherData)
    if err != nil {
        fmt.Println("解密失败:", err)
        return
    }

    fmt.Printf("解密后的数据为:%s\n", plainText)
}
```

des特点：

1. 8字节秘钥，8字节分组

cbc特点：

1. 需要填充

阶段1：加密满足条件长度的数据（8字节）

```
package main

import (
    "bytes"
    "crypto/cipher"
    "crypto/des"
    "fmt"
)

//des特点：
//1. 8字节秘钥，8字节分组
//
//cbc特点：
//1. 需要填充

//加密函数 : key:秘钥， plainText :明文
func desCbcEncrypt(key, plainText []byte) ([]byte, error) {
    fmt.Println("开始加密,明文:", string(plainText))

    //第一步：创建密码接口
    block, err := des.NewCipher(key)
    if err != nil {
        fmt.Println("des.NewCipher err:", err)
        return nil, err
    }

    fmt.Println("Block Size:", block.BlockSize()) //8

    //创建iv
    iv := bytes.Repeat([]byte("1"), block.BlockSize())

    //TODO填充数据

    //第二步：创建cbc加密分组
    blockMode := cipher.NewCBCEncrypter(block, iv)

    dst := make([]byte, len(plainText))
    //第三步：加密
    blockMode.CryptBlocks(dst /*密文*/, plainText /*明文*/)

    return dst, nil
}

//解密函数
func desCbcDecrypt(key, cipherText []byte) ([]byte, error) {
    fmt.Println("开始解密,密文:", string(cipherText))
    //第一步：创建密码接口
```

```
    block, err := des.NewCipher(key)
    if err != nil {
        fmt.Println("des.NewCipher err:", err)
        return nil, err
    }

    fmt.Println("Block Size:", block.BlockSize()) //8

    //创建iv
    iv := bytes.Repeat([]byte("1"), block.BlockSize())

    //第二步：创建cbc解密分组
    blockMode := cipher.NewCBCDecrypter(block, iv)   //<<<<========== 此处不同

    dst := make([]byte, len(cipherText))
    //第三步：解密
    blockMode.CryptBlocks(dst /*明文*/, cipherText /*密文*/)

    //去除填充
    //TODO
    return dst, nil
}

func main() {
    key := []byte("12345678")     //8字节秘钥
    plainText := []byte("你好，昌平!") //ok
    //plainText := []byte("你好，昌平!!") //不ok

    cipherData, err := desCbcEncrypt(key, plainText)
    if err != nil {
        fmt.Println("加密失败:", err)
        return
    }

    fmt.Printf("加密后的数据为, hex :%x\n", cipherData)
    fmt.Printf("加密后的数据为, string :%s\n", cipherData)

    plainText, err = desCbcDecrypt(key, cipherData)
    if err != nil {
        fmt.Println("解密失败:", err)
        return
    }

    fmt.Printf("解密后的数据为:%s\n", plainText)
}
```

阶段2：输入不满足条件的数据（10字节），需要填充

填充函数：

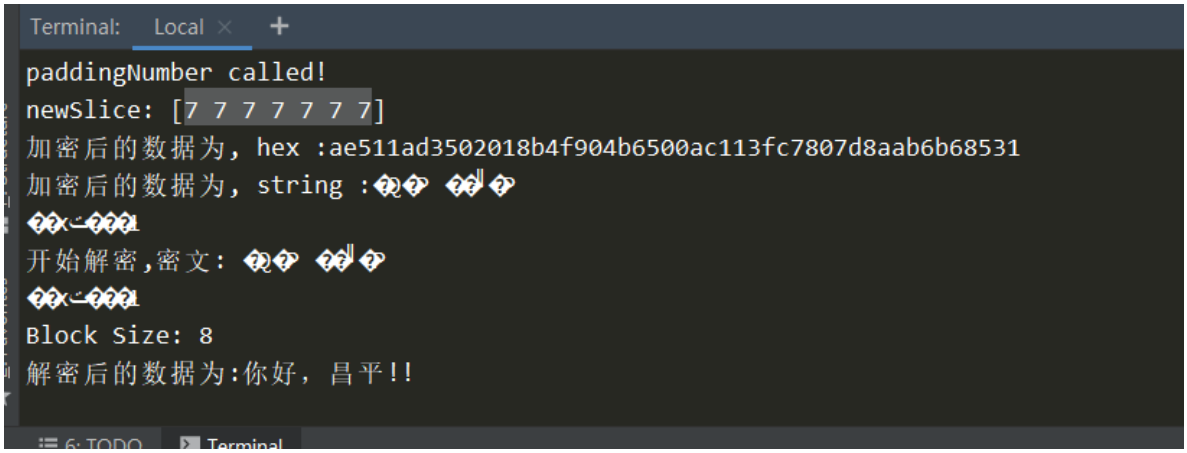
```
//填充函数
func paddingNumber(src []byte, blockSize int) ([]byte, error) {
    fmt.Println("paddingNumber called!")
```

```
    if src == nil {
        return nil, errors.New("填充数据为空!")
    }

    //src : 25byte，每个分组长度是:8byte 需要填充:7byte
    //a. 剩余字节数
    leftNumber := len(src) % blockSize //1

    //b. 求出需要填充的个数
    needNumber := blockSize - leftNumber

    //[]byte{7,7,7,7,7,7,7}
    newSlice := bytes.Repeat([]byte{byte(needNumber)}, needNumber)

    fmt.Println("newSlice:", newSlice)

    //将填好的数据追加到src后面
    src = append(src, newSlice...)
    return src, nil
}
```

加密之前调用：

```
    fmt.Println( a...: "Block Size:", block.BlockSize()) //8

    //创建iv
    iv := bytes.Repeat([]byte("1"), block.BlockSize())

    //在加密之前进行填充
    plainText, _ = paddingNumber(plainText, block.BlockSize())

    //第二步：创建cbc加密分组
    blockMode := cipher.NewCBCEncrypter(block, iv)

    dst := make([]byte, len(plainText))
    //第三步：加密
    blockMode.CryptBlocks(dst /*密文*/, plainText /*明文*/)
```

效果：



去除填充:

```
//去除填充函数
func unPaddingNumber(src []byte) []byte {
    fmt.Println("unPaddingNumber called!")

    //1. 获取最后一个字符
    lastByte := src[len(src)-1]
    num := int(lastByte)

    //2. 截取原文
    return src[0 : len(src)-num]
}
```

```
func desCbcDecrypt(key, cipherText []byte) ([]byte, error) {
    fmt.Println( a...: "开始解密,密文:", string(cipherText))
    //第一步：创建密码接口
    block, err := des.NewCipher(key)
    if err != nil {...}

    fmt.Println( a...: "Block Size:", block.BlockSize()) //8

    //创建iv
    iv := bytes.Repeat([]byte("1"), block.BlockSize())

    //第二步：创建cbc解密分组
    blockMode := cipher.NewCBCDecrypter(block, iv)

    dst := make([]byte, len(cipherText))
    //第三步：解密
    blockMode.CryptBlocks(dst /*明文*/, cipherText /*密文*/)

    //去除填充字符
    dst = unPaddingNumber(dst)
```

解密之后，调用去除填充函数