基础知识

1.加密三要素

- 明文/密文
- 密钥
- 算法
  - 加密算法
  - 解密算法

> 密钥：其实就是定长的字符串，需要根据算法确定其长度；
>
> 算法：加密算法和解密算法有可能是互逆的，也有可能是相同的；
>
> 因为解密和加密是互逆操作，因此下面阐述具体操作中，仅从加密操作描述。

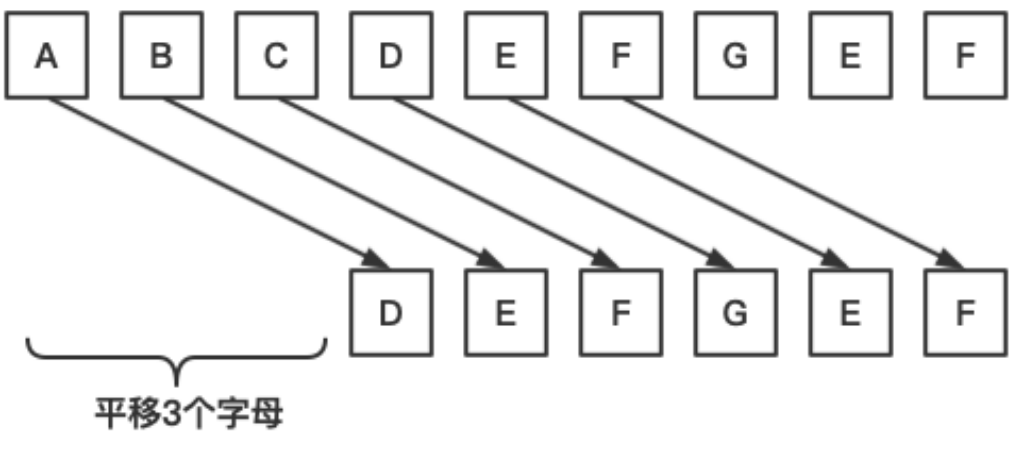
2.常用的两种加密方式⭐

- **对称加密**
- **非对称加密**
  - 公钥—可以公开是密钥；
  - 私钥—需要妥善保存的密钥，知道的人越少越好；

> 所谓的对称加密指的是在加解密时使用的是同一个密钥，而非对称加解密时使用的是不同的密钥，即有两个密钥—公钥和私钥，公钥加密数据，就需要使用私钥进行解密，如果用私钥加密就需要公钥进行解密。

> 对称加密在密钥不泄露的情况下保护性是双向的，而非对称加密如果其公钥暴露则其保护性是单向的。
>
> 对称加密的效率比较高，适用于大数据的情况。但是对称加密的加密强度相比于非对称加密要低一些。

3.凯撒密码



凯撒密码是一种相传尤里乌斯·撒旦使用过密码。撒旦于公元前100年左右于古罗马，是一位著名的军事统帅。凯撒密码是通过将明文中所使用的字母表按照一定的字数“平移”来进行加密的。公式如下：

- 加密公式：
  - $E_n(x) = (x + n) mod 26$
  - n相当于是上图中平移的个数
- 解密公式：
  - $E_n(x) = (x - n) mod 26$
  - n相当于是上图中平移的个数

> 试分析上面凯撒密码的加密三要素？

对称加密

> 以分组为单位进行处理的密码算法称为**分组密码（blockipher）**

1. 编码概念

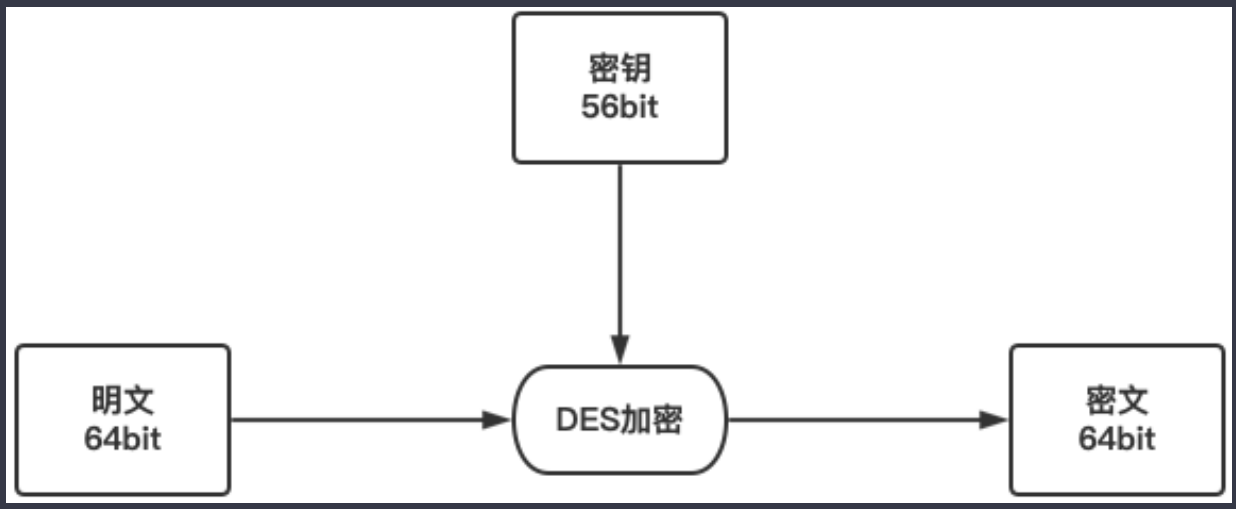
计算机的操作对象并不是文字，而是由0和1排列而成的**比特序列**，将现实世界中的东西映射为比特序列的操作称之为编码（encoding）。

2. DES — Data Encryption Standard

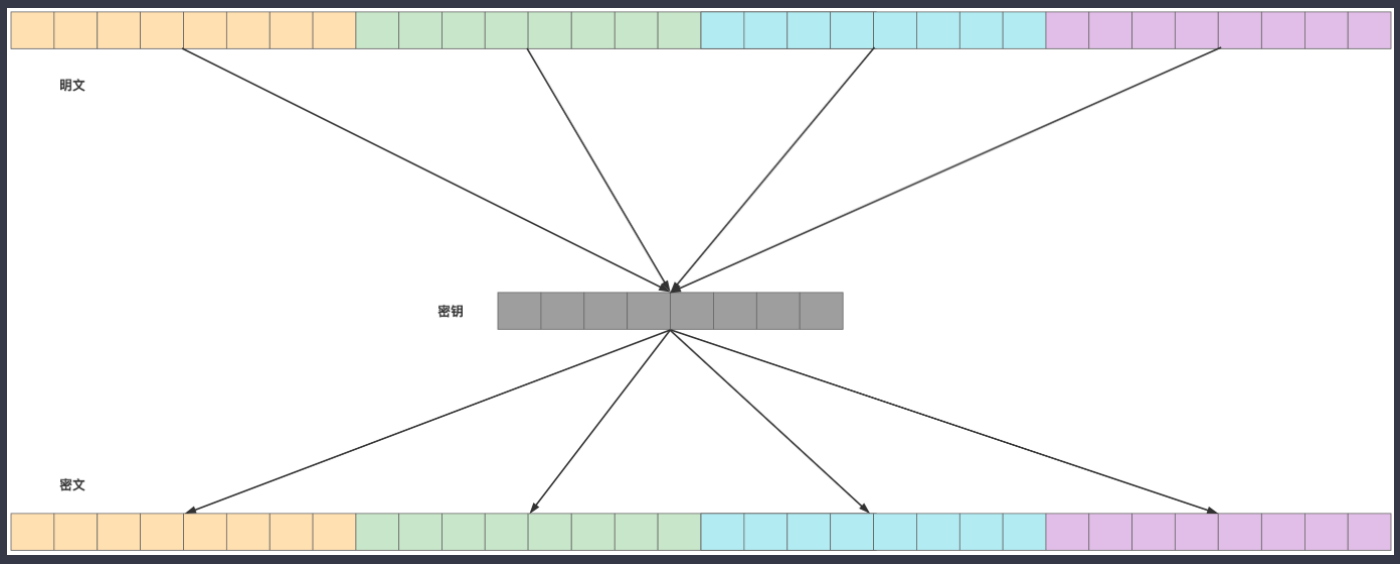
- 什么是DES

DES是1977年美国联邦信息处理标准中所采用的一种对称密码，DES一直以来被美国和其他国家的政府和银行等广泛使用。然后随着计算机的进步，现在DES已经能够被暴力破解了，强度大不如前。因此么现在我们基本上已经不使用DES了。

- DES加密和解密



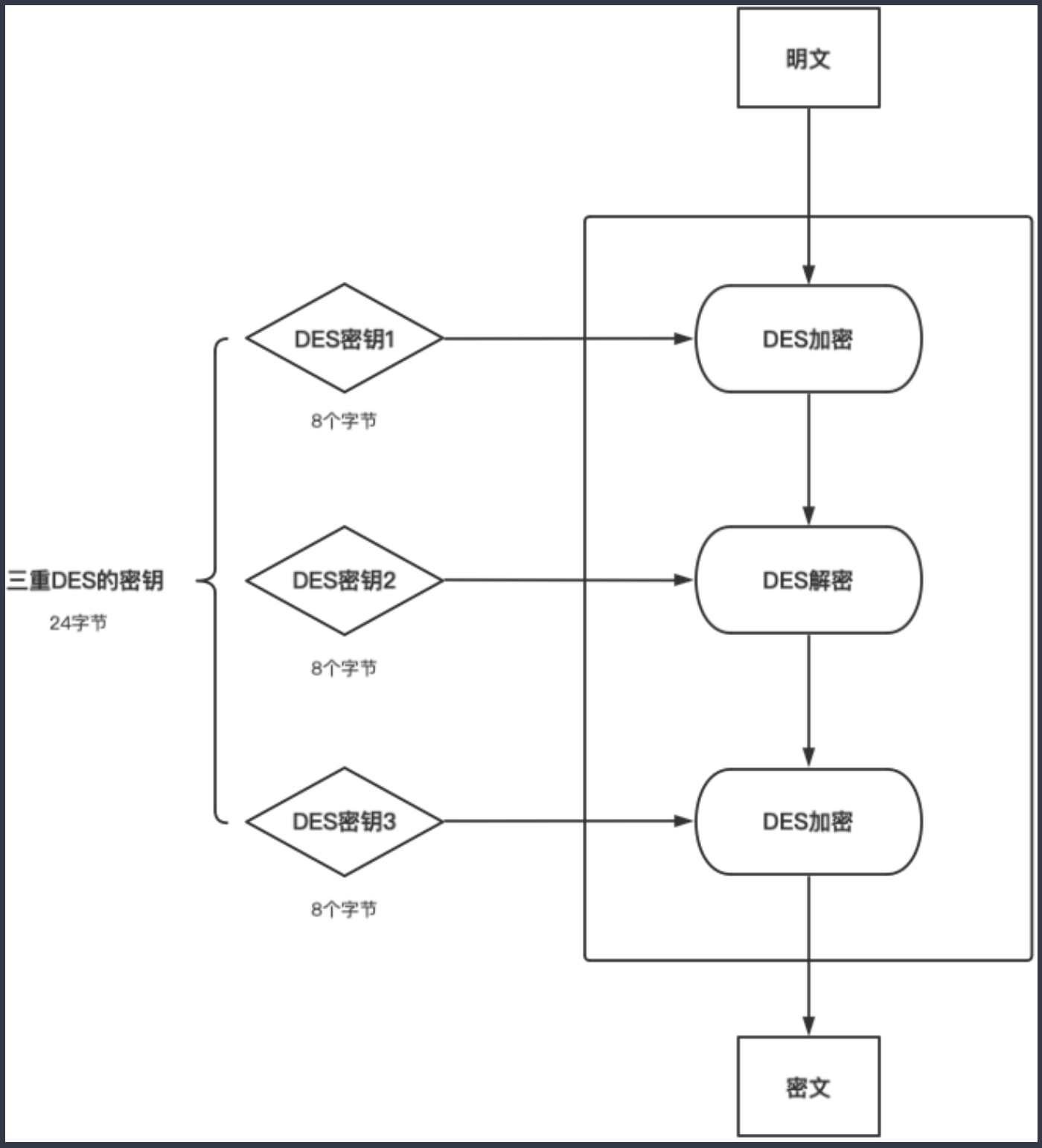
  - DES是一种将64位比特的明文加密成64比特的密文的对称密码算法。它的密钥长度为56比特。尽管从规格上来说，DES的密钥长度是64比特（8个字节），但由于每隔7比特会设置一个用于检查错误的比特，因此实质上其密钥长度是56比特。
  - DES是以64比特的明文（比特序列）为单位来进行加密的，这个64位比特的单位称为分组。一般来说，以分组为单位进行处理的密码算法称为分组密码，而DES就是分组密码的一种【（56+8）/8 = 8】。
  - DES每次只能加密64比特的数据，如果要加密的明文比较长，就需要对DES加密进行迭代，而迭代的具体方式称为模式（mode）。



3. 3DES — Triple-DES

> 现在DES已经可以被暴力破解。因此我们需要一种算法来替代DES的分组密码。三重DES就是出于这个目的被叫开发出来的。**三重DES**是为了增加DES的强度，将DES重复3次所得到的一种密码算法，简单的缩写就是为3DES。3DES架构兼容DES。

- 3DES的架构。注意3DES密钥的长度。



4. AES — Advanced Encryption Standard

> AES是取代其前任标准（DES）而成为新标准的一种对称加密算法。其分组长度为128比特（16个字节），密钥长度可以以32比特为单位在128到256比特的范围内进行选择（不过在AES的规格中，密钥长度只有128、192和256比特三种）

- AES的加密和解密

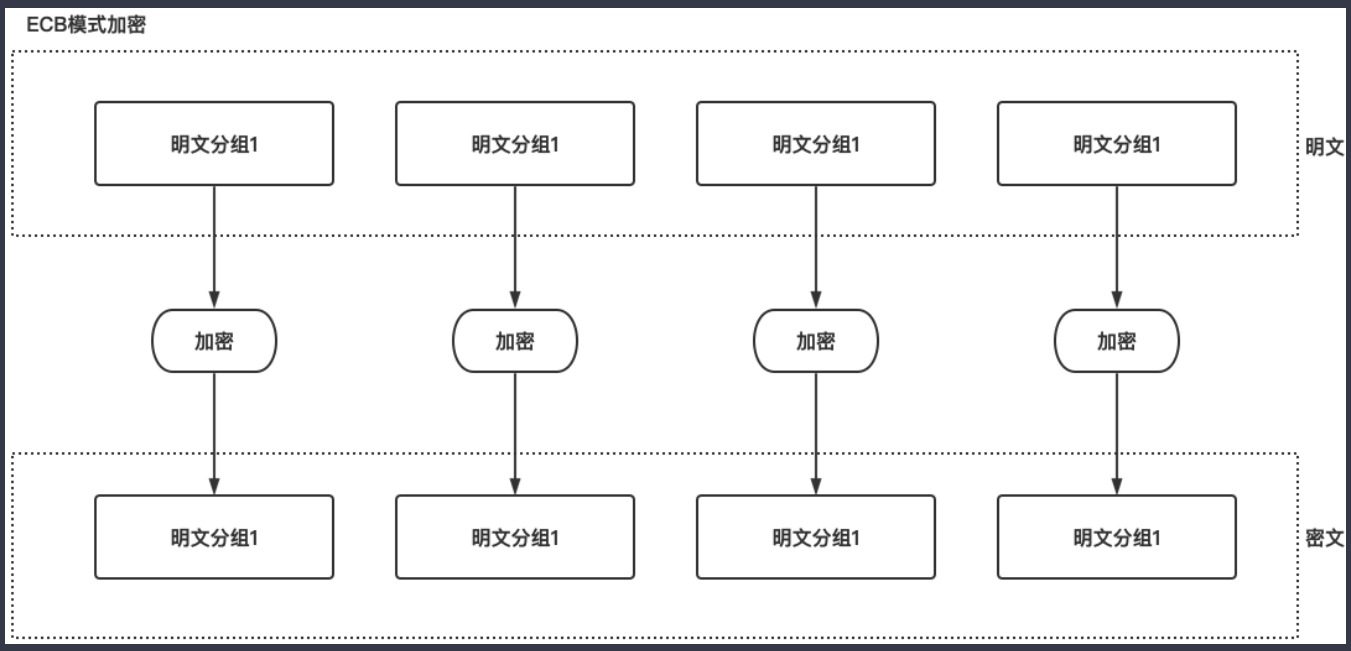
- SubBytes（逐字节替换）
- ShiftRows（平移化）
- MixColumns（混合列）
- AddRoundKey（与轮密钥进行XOR）

> AES的加密过程比较复杂，不需要对其中具体的加密细节进行掌握，但是需要清楚知道以下几点：
>
> 如果你的代码中需要使用对称加密，首选就是AES，其安全且效率高
>
> AES它是分组密码
>
> AES分组长度为128比特也就是16个字节
>
> AES的密钥长度有3种规格（128bit、192bit、256bit），golang中只支持128bit即16字节

5. 分组密码模式

- 按位异或
  - 第一步将数据转换为二进制序列
  - 按位异或操作符:^、XOR【看两个标志位，相同为0，不同为1】
- ECB—Electronic Code Book 电子密码本模式【简单、效率高、容易被破解】

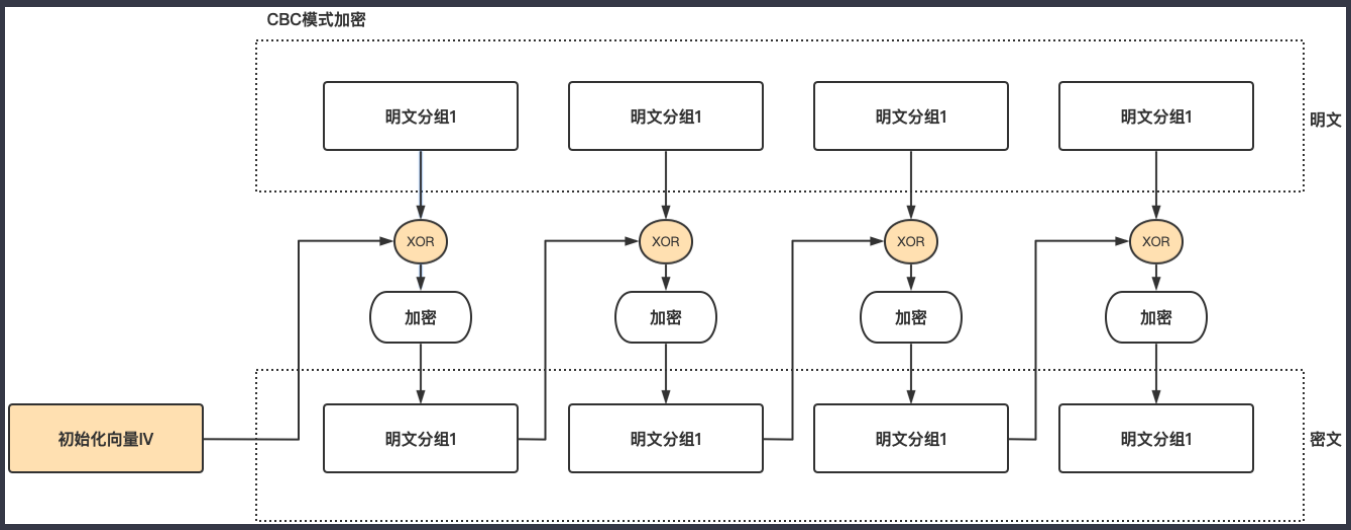
  ECB模式是最简单的加密模式，明文消息被分成固定大小的块（分组），并且每个块被单独加密。每个块的加密和解密都是独立的，且使用相同的方法进行解密，所以可以进行并行运算，但是这种方式一旦有一个块被破解，其他的块也可以使用相同的方法解密所有的明文数据，安全性较差，适用于数据较少的情形。加密前需要把明文数据填充到块大小的整数倍。



> 使用ECB模式加密时，相同的明文分组会被转换为相同的密文分组，也就是说，我们可以理解为是一个巨大的“明文分组->密文分组”的对应表，因此ECB模式也称为电子密码本模式，当最后一个明文分组的内容小于分组长度的时，需要用一特定的数据进行填充（padding），让其满足一个分组长度。
>
> ECB模式是所有模式中最简单的一种。ECB模式中，明文分组与密文分组是一一对应的关系，因此，如果明文中存在多个相同的明文分组，则这些明文分组最终都将被转换为相同的密文分组，这样一来，只要观察一下密文，就可以知道明文中，存在怎么的重复组合，并可以以此为线索来破译密码，因此ECB模式是存在一定的风险的。

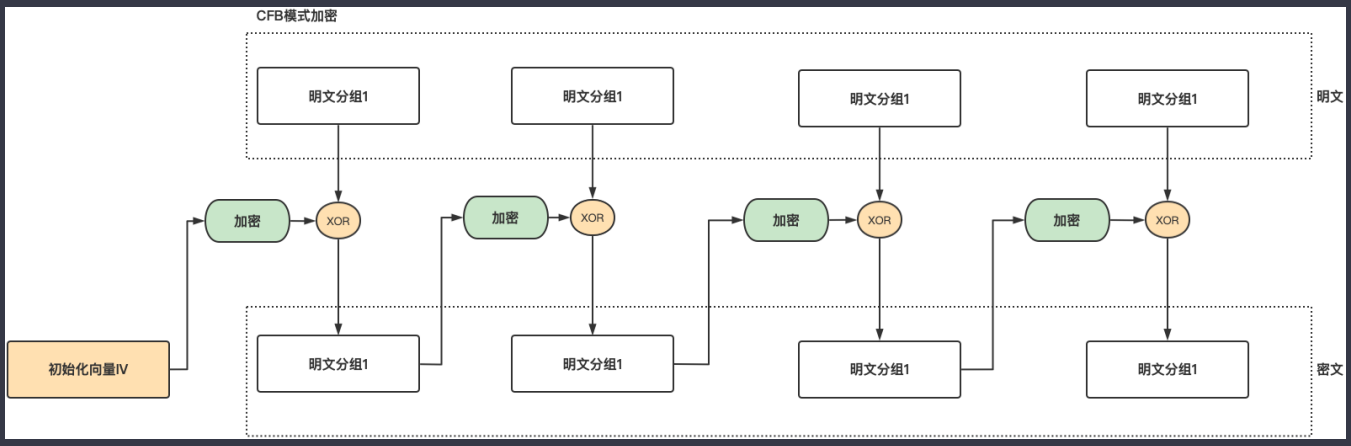
- CBC—Cipher Block Chaining 密码块链模式

CBC进行加密之前需要用初始化向量IV进行异或操作。CBC模式是一种最常用的加密模式，它最主要的缺点就是加密是连续的，不能并行处理，并且与ECB一样，消息块必须填充到块大小的整数倍。



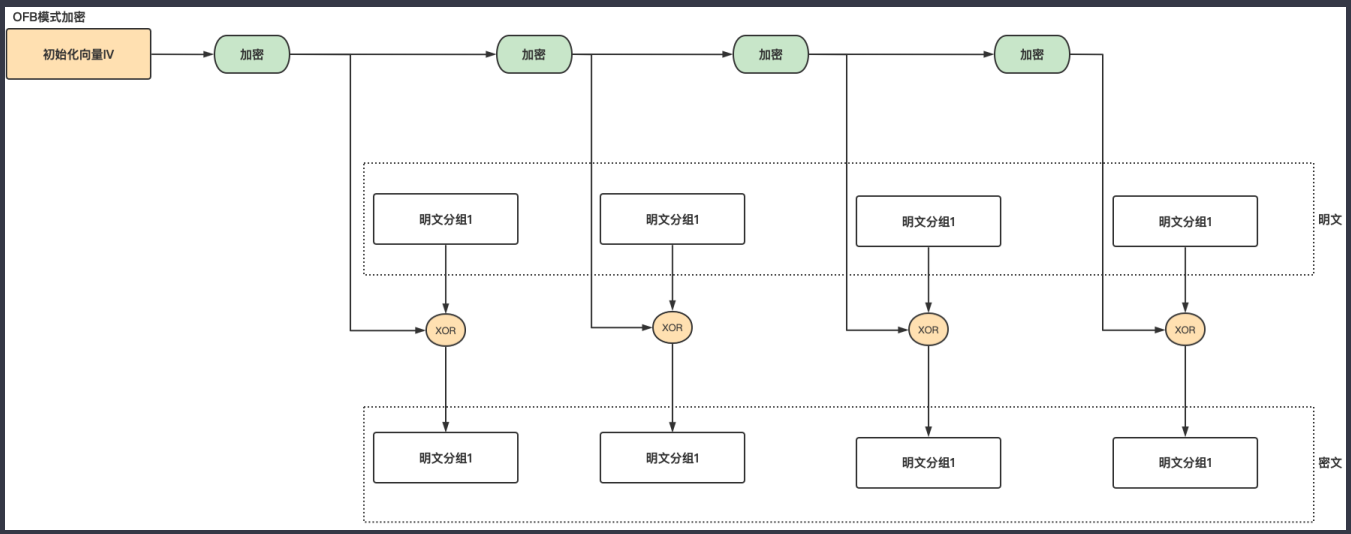
- CFB—Cipher FeedBack 密文反馈模式【密文没有规律】

  CFB模式中，前一个分组的密文加密后和当前分组的明文XOR异或操作生成当前分组的密文。而所谓的反馈指的是返回输入端的意思，即前一个密文分组会被送回到密码算法的输入端。CFB模式的加密与解密与CBC模式的加密与解密在流程上其实是非常相似的。【不需要padding操作】



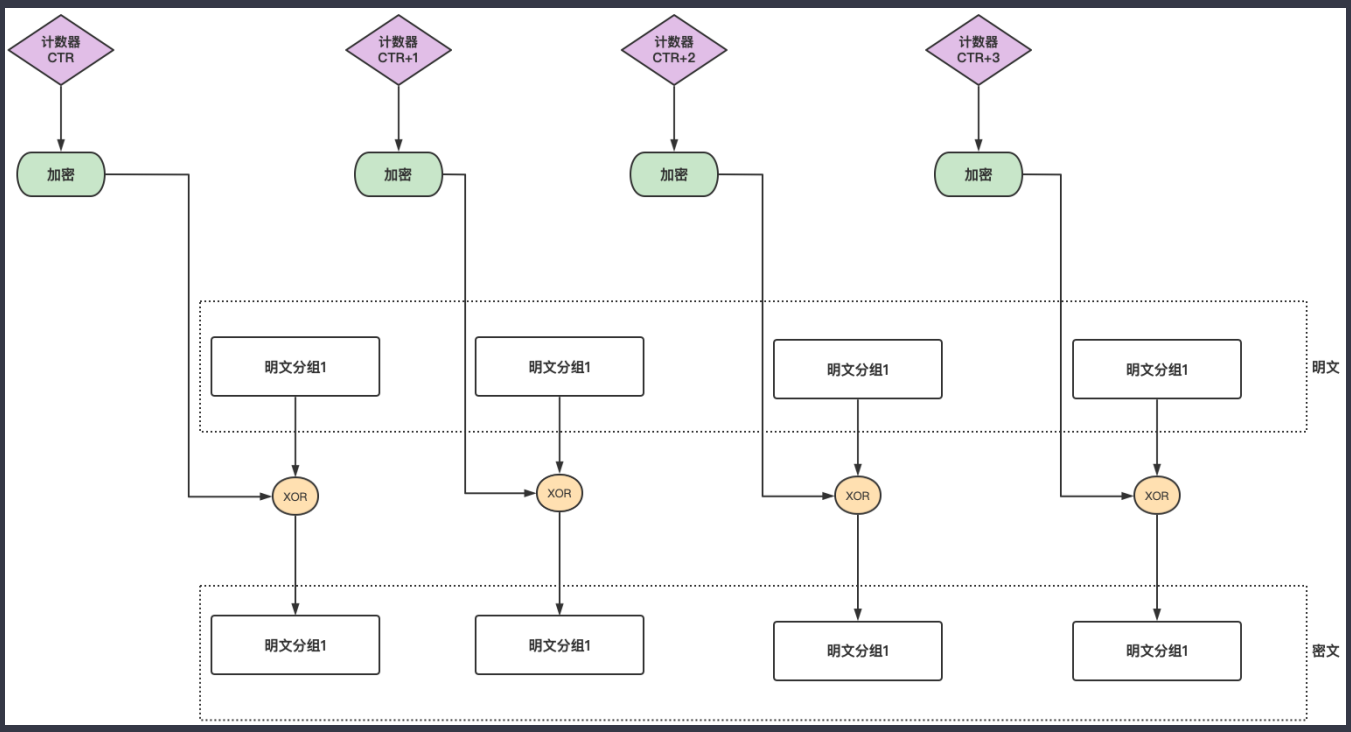
- OFB—Output FeedBack 输出反馈模式【密文没有规律】

  在OFB模式中，密码算法的输出会反馈到密码算法输入中，输上一个分组密码算法的输出是当前的分组密码算法的输入。OFB模式并不是通过密码算法对明文直接加密的，而是通过将“明文分组”和“密码算法的输出“进行XOR产生密文分组，在这一点上OFB和CFB模式非常相似。



- CTR—CounTeR 计数器模式【密文没有规律】

  CTR模式的全城是CounTeR模式（计数器模式）。CTR模式是一种将逐次累加的计算器进行加密来生成的密钥的加密模式。在CTR模式中，每个分组对应一个逐次累加的计数器，并通过对计数器进行加密来生成密钥流。也就是说，最终的密文是通过将计数器加密得到比特序列。



> ECB：明文分组的长度有一定的要求，如果不满足要求，需要使用padding进行操作；【不安全】
>
> CBC：明文分组的长度有一定的要求，且需要初始化向量；【安全】
>
> CFB、OFB：都需要准备初始向量，但是不对明文的长度作要求，简单来说就是没有pading操作；
>
> CTR：不需要初始化向量，也不对明文的长度作要求；
>
> 显然，上述的过程一个条件泛化的过程。

6. 在golang中使用对称加密⭐

基本的加密流程如下【查询具体crypto都可以看到具体的API接口】：

1. 创建一个底层使用的DES3/3DES/AES的密码接口；
2. 如果使用的是CBC/ECB分组模式需要对明文分组进行填充；
3. 创建一个密码分组模式的接口对象CBC、CFB、OFB、OTR；
4. 加密，得到密文；

6.1 填充函数

编写填充函数：如果最后一个分组字节数不够，则需要填充。另外如果明文分组刚刚好不需要填充，则我们在尾部添加一个完成的分组，分组的字节值可以为一个分组大小，解密时同样获取这个字节值，来完成删除。

```
// plainText 原始数据
// blockSize 分组字节数
func padding(plainText []byte,blockSize int)[]byte{
  // 求出需要填充的数目
  padNum := blockSize - (len(plainText) % blockSize)
  // 创建一个新的切片
  padObj := []byte{byte(padNum)}
  // 元素拼接
  plainText = append(plainText,bytes.Repeat(padObj,padNum)...)
  return plainText
}
```

去除填充函数：一个填充技巧是说如果你的填充数为5，则将这个填充元素设置为5（int->byte）最好，这样后面可以通过这个ASCII码值（byte-> int）确定解密时候需要删除的的字节数。

```
// plainText 解密后明文
func unPadding(plainText []byte)[]byte{
  // 获取切片最后一个字节
  lastChar := plainText[len(plainText)-1]
  // 将char转换为int
  padNum := int(lastChar)
  // 去除尾部填充
  return plainText[:len(plainText)-padNum]
}
```

6.2 DES+CBC API接口

在实现其他操作的时候，直接换成指定的API即可，大致的框架是如下的：

- 加密操作

```
// plainText 原始明文
// key 密钥
func desEncrypt(plainText,key []byte)[]byte{
  // 使用底层提供的一个des接口
  block,err := des.NewCipher(key)
  if err != nil {
    panic(err)
  }
  // 明文填充
  newText := padding(plainText,block.BlockSize())
  // 创建一个使用cbc加密分组的接口
  iv := []byte("12345678") // 相当于初始化一个向量
  blockMode := cipher.NewCBCEncrypter(block,iv)
  // 加密
  cipherText := make([]byte,len(newText)) // 密文的长度与明文的长度一致
  blockMode.CryptBlocks(cipherText,newText)
  // 将加密后密文返回
  return cipherText
}
```

- 解密操作

```
// cipherText 加密密文
// key 密钥
func desDecrypt(cipherText,key []byte)[]byte{
  // 使用底层提供的一个des接口
  block,err := des.NewCipher(key)
  if err != nil {
    panic(err)
```

```
    }
    // 创建一个使用cbc解密分组的接口
    iv := []byte("12345678") // 相当于初始化一个向量
    blockMode := cipher.NewCBCDecrypter(block,iv)
    // 解密
    blockMode.CryptBlocks(cipherText,cipherText) // 覆盖操作
    // 去除填充
    plainText := unPadding(cipherText)
    return plainText
}
```

- 测试

```
func main() {
    password := "testabc!34"
    fmt.Println("password:", password)
    key := "mykey234"
    EncryptPw := desEncrypt([]byte(password), []byte(key))
    fmt.Println("encrypt password:", string(EncryptPw))
    passwordTemp := desDecrypt(EncryptPw, []byte(key))
    fmt.Println("password:", string(passwordTemp))
}
```

- 完整代码

```
package main

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

// plainText 原始数据
```

```go
// blockSize 分组字节数
func padding(plainText []byte, blockSize int) []byte {
  // 求出需要填充的数目
  padNum := blockSize - (len(plainText) % blockSize)
  // 创建一个新的切片
  padObj := []byte{byte(padNum)}
  // 元素拼接
  plainText = append(plainText, bytes.Repeat(padObj,
padNum)...)
  return plainText
}

// plainText 解密后明文
func unPadding(plainText []byte) []byte {
  // 获取切片最后一个字节
  lastChar := plainText[len(plainText)-1]
  // 将char转换为int
  padNum := int(lastChar)
  // 去除尾部填充
  return plainText[:len(plainText)-padNum]
}

// plainText 原始明文
// key 密钥
func desEncrypt(plainText, key []byte) []byte {
  // 使用底层提供的一个des接口
  block, err := des.NewCipher(key)
  if err != nil {
    panic(err)
  }
  // 明文填充
  newText := padding(plainText, block.BlockSize())
  // 创建一个使用cbc加密分组的接口
```

```
  iv := []byte("12345678") // 相当于初始化一个向量
  blockMode := cipher.NewCBCEncrypter(block, iv)
  // 加密
  cipherText := make([]byte, len(newText)) // 密文的长度与明文的长度一致
  blockMode.CryptBlocks(cipherText, newText)
  // 将加密后密文返回
  return cipherText
}

// cipherText 加密密文
// key 密钥
func desDecrypt(cipherText, key []byte) []byte {
  // 使用底层提供的一个des接口
  block, err := des.NewCipher(key)
  if err != nil {
    panic(err)
  }
  // 创建一个使用cbc解密分组的接口
  iv := []byte("12345678") // 相当于初始化一个向量
  blockMode := cipher.NewCBCDecrypter(block, iv)
  // 解密
  blockMode.CryptBlocks(cipherText, cipherText) // 覆盖操作
  // 去除填充
  plainText := unPadding(cipherText)
  return plainText
}

func main() {
  password := "testabc!34"
  fmt.Println("password:", password)
  key := "mykey234"
  EncryptPw := desEncrypt([]byte(password), []byte(key))
```

```
  fmt.Println("encrypt password:", string(EncryptPw))
  passwordTemp := desDecrypt(EncryptPw, []byte(key))
  fmt.Println("password:", string(passwordTemp))
}
```

6.3 AES+CTR API接口⭐

CTR分组加密对原始明文长度不做要求，因此没有padding操作。

- 加密操作

```
// plainText 原始明文
// key 密钥
func aesEncrypt(plainText, key []byte) []byte {
  // 使用底层提供的一个aes接口
  block, err := aes.NewCipher(key)
  if err != nil {
    panic(err)
  }
  // CTR分组加密对原始明文长度不做要求
  // 创建一个使用CTR加密分组的接口
  iv := []byte("1234567812345678") // 相当于初始化一个向量
  blockMode := cipher.NewCTR(block, iv)
  // 加密
  cipherText := make([]byte, len(plainText)) // 密文的长度与明文的长度一致
  blockMode.XORKeyStream(cipherText, plainText)
  // 将加密后密文返回
  return cipherText
}
```

- 解密操作

```
// cipherText 加密密文
// key 密钥
func aesDecrypt(cipherText, key []byte) []byte {
```

```
    // 使用底层提供的一个des接口
    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
    }
    // 创建一个使用CTR解密分组的接口
    iv := []byte("1234567812345678")
    blockMode := cipher.NewCTR(block, iv)
    // 解密
    plainText := make([]byte, len(cipherText))
    blockMode.XORKeyStream(plainText, cipherText) // 覆盖操作
    return plainText
}
```

- 测试

```
func main() {
    password := "testabc!34"
    fmt.Println("password:", password)
    key := "mykey234mykey234"
    EncryptPw := aesEncrypt([]byte(password), []byte(key))
    fmt.Println("encrypt password:", string(EncryptPw))
    passwordTemp := aesDecrypt(EncryptPw, []byte(key))
    fmt.Println("password:", string(passwordTemp))
}
```

- 完整代码

```
package main

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

```go
// plainText 原始明文
// key 密钥
func aesEncrypt(plainText, key []byte) []byte {
  // 使用底层提供的一个des接口
  block, err := aes.NewCipher(key)
  if err != nil {
    panic(err)
  }
  // CTR分组加密对原始明文长度不做要求
  // 创建一个使用cbc加密分组的接口
  iv := []byte("1234567812345678")
  blockMode := cipher.NewCTR(block, iv)
  // 加密
  cipherText := make([]byte, len(plainText)) // 密文的长度与明文的长度一致
  blockMode.XORKeyStream(cipherText, plainText)
  // 将加密后密文返回
  return cipherText
}

// cipherText 加密密文
// key 密钥
func aesDecrypt(cipherText, key []byte) []byte {
  // 使用底层提供的一个aes接口
  block, err := aes.NewCipher(key)
  if err != nil {
    panic(err)
  }
  // 创建一个使用CTR解密分组的接口
  iv := []byte("1234567812345678")
  blockMode := cipher.NewCTR(block, iv)
  // 解密
```

```
    plainText := make([]byte, len(cipherText))
    blockMode.XORKeyStream(plainText, cipherText) // 覆盖操作
    return plainText
}

func main() {
    password := "testabc!34"
    fmt.Println("password:", password)
    key := "mykey234mykey234"
    EncryptPw := aesEncrypt([]byte(password), []byte(key))
    fmt.Println("encrypt password:", string(EncryptPw))
    passwordTemp := aesDecrypt(EncryptPw, []byte(key))
    fmt.Println("password:", string(passwordTemp))
}
```

7.总结

对称加密

加密算法	分组长度	密钥长度
DES	8字节	8字节
3DES	8字节	24字节
AES	16字节	16字节、24字节、32字节

分组模式

分组模式	推荐	条件
EBC	不推荐	
CBC	常用方式	初始化向量（长度等于明文分组长度）
OFB	不推荐	
CFB	不推荐	
CTR	推荐，效率高	初始化向量（长度等于明文分组长度）

非对称加密

1.对称加密的弊端

- 对称加密中，密钥分发困难，不能通过网络来分发密钥；
- 如果非要发，可以通过非对称加密的方式进行密钥的分发；

> 一个实际的场景：Kevin和Bob进行通信，Kevin给Bob发送数据，使用对称加密的方式：
>
> - Bob生成一个非对称的密钥对；
> - Bob将密钥对中的公钥发送给Kevin；
> - Kevin生成一个用于对称加密的密钥；
> - Kevin使用Bob的公钥对对称加密的密钥进行加密，然后发送给Bob；
> - Bob使用私钥对数据进行解密，得到对称加密的密钥；
> - 通信双方使用写好的密钥进行对称加密数据加密；

2.非对称加密的密钥

- 不存在密钥分发困难的问题，因为可以通过网络来分发公钥；

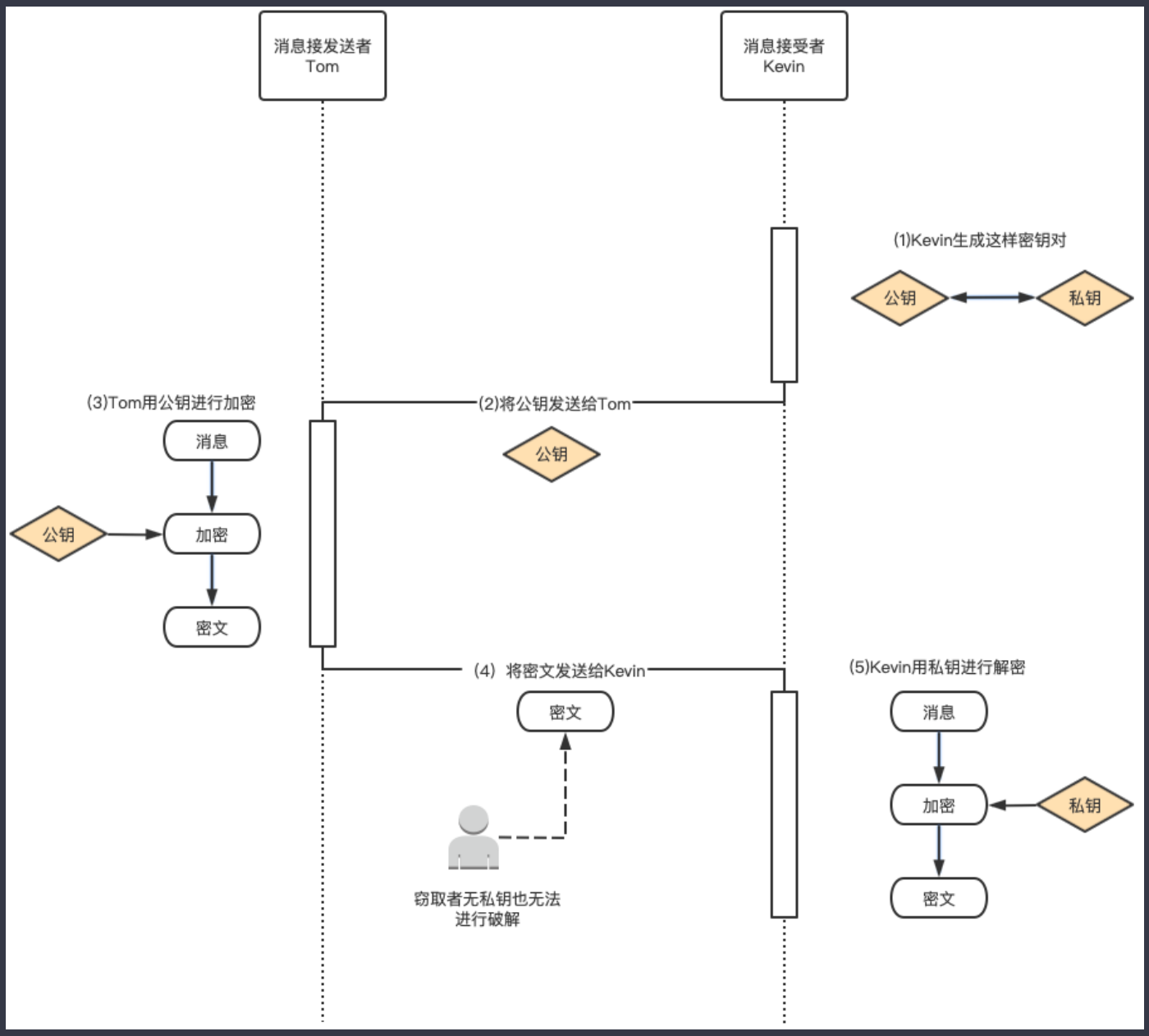
2.1 场景分析

> 1. 信息加密（A写数据，发送给B，信息只允许B读）
>
>   A：公钥
>
>   B：私钥
> 2. 登录认证（客户端要登录，连接服务器，向服务器请求个人数据）
>
>   客户端：私钥
>
>   服务器：公钥
> 3. 数字签名（表明信息没有受到伪造，缺失是信息拥有者发出来的，附在信息原文的后面）
>
>   发送信息的人：私钥
>
>   收到信息的人：公钥
> 4. 网银U盾
>
>   U盾：私钥
>
>   银行：公钥

通过以上场景的分析，我们知道，数据对谁重要，那么那一方就持有私钥。另外直观上来说，私钥文件闭公钥文件长。使用第三方工具生成密钥对：公钥文件`xxx.pub`，

3.非对称加密通信流程

用part1的部分的例子来说明：



4.RSA

非对称加密的密钥分为加密密钥和解密密钥。而这其中使用最广泛的就是公钥密码算法—RSA，其被用于非对称加密和数字签名。

4.1 RSA加密

在RSA中，明文、密钥和密文都是数字。RSA的加密过程可以用下列公式来表达:

$$密文 = 明文^{E} mod N$$

也就是说，RSA的密文是对代表明文的数字的E次方对N求余的结果，这相比于对称加密的复杂流程来说RSA的加密过程非常的简单。因此任何知道这两个数`E`和`N`都可以完成运算，所以加密公式中出现的两个数E和N就是RSA的加密密钥，也就是说E和N的组合就是公钥。而加解密中出现的`E`是Eecryption的首字母，N是Number的首字母。

4.2 RSA解密

和RSA加密一样简单，解密的过程可以通过下面的公式来说明：

$$明文 = 密文^{D} mod N$$

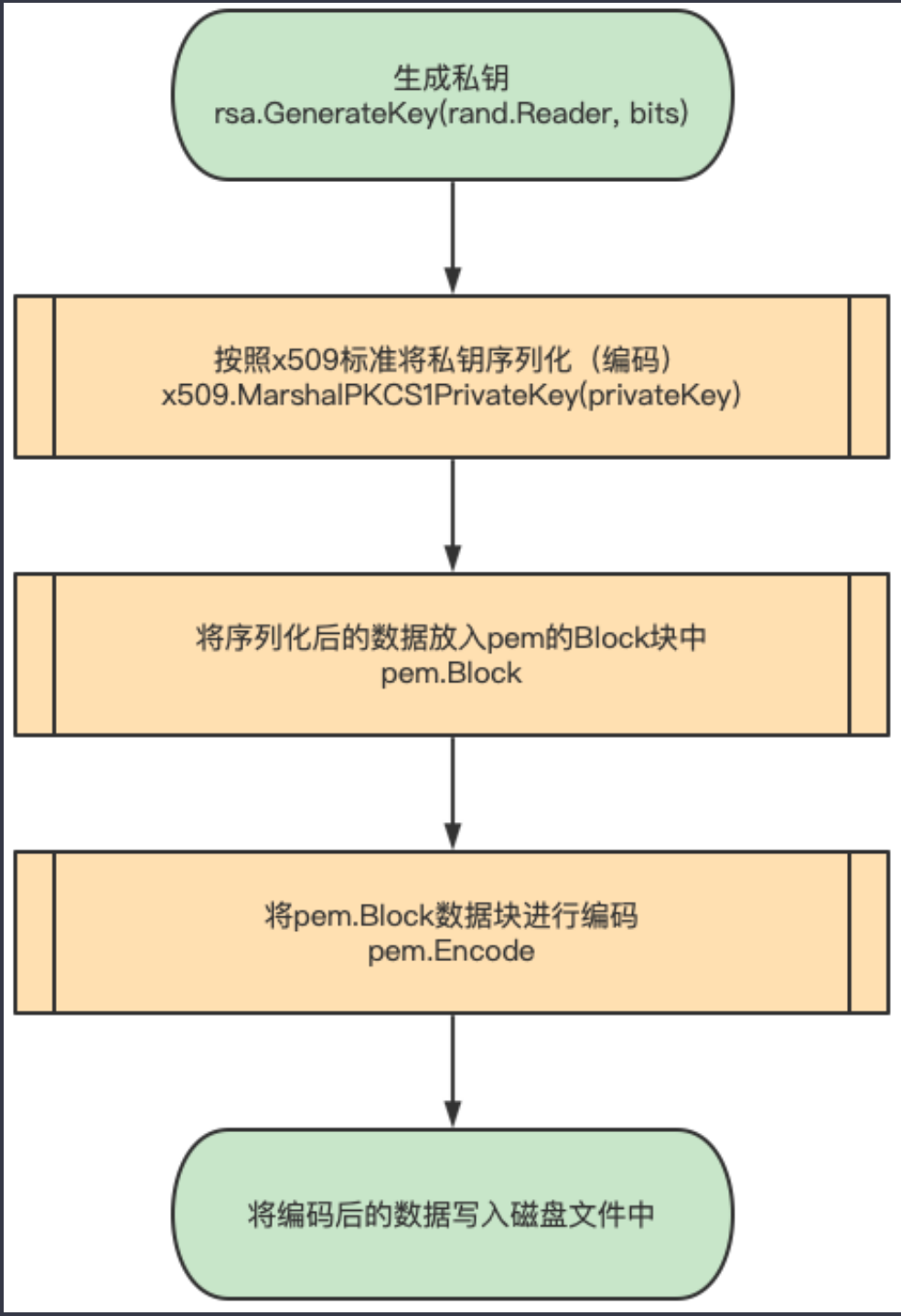
也就是说对密文的数字的D次方再对N求余的结果。这里所使用的数字N和加密时使用N是一样的。数字D和数字N组合起来就是RSA的解密密钥，因此D和N的组合就是私钥。只有知道D和N两个数的人才能够完成解密运算。而加解密中出现的`D`是Decryption的首字母，N是Number的首字母。

4.3 生成RSA密钥对

- 一些基本的概念
  - x509证书规范、pem、base64

    pem和base64都是一种数据编码方式，不过后者是可逆的。
  - ASN.1抽象语法标记
  - PKCS1标准
- 密钥对生成流程

  其实后续进行编码的工作主要是为了能够保存到文件中。其大致流程如下：



- 生成私钥操作流程的概述【go提供下述的操作的API】

  - 使用rsa中的GenerateKey方法生成私钥；
  - 通过x509标准将得到rsa私钥序列序列化为ASN.1的编码字符串；
  - 将私钥字符串设置到pem格式块(pem.block)中；
  - 通过pem将设置好的数据进行编码，并写入磁盘文件中；

- 生成公钥操作流程的概述【go提供了下述操作的API】

  - 从得到的私钥对象中将公钥信息取出；

> - 通过x509标准得到的rsa公钥序列化为字符串；
> - 将公钥字符串设置为pem格式块中；
> - 通过pem将设置好的数据进行编码，并写入磁盘文件中；

- golang生成rsa密钥对

```
// GenerateRsaKey bits 密钥长度
func GenerateRsaKey(bits int) {
  // ====================生成私钥==================
  // 使用rsa中的GenerateKey方法生成私钥
  eKey, err := rsa.GenerateKey(rand.Reader, bits)
  if err != nil {
    panic(err)
  }
  // 通过x509标准将得到rsa私钥序列序列化为ASN.1的编码字符串
  data := x509.MarshalPKCS1PrivateKey(eKey)
  // 将私钥字符串设置到pem格式块中
    block := &pem.Block{
    Type:  "RSA PRIVATE KEY", // 得自前言的类型（如"RSA PRIVATE KEY"）
    Bytes: data,              // 内容解码后的数据，一般是DER编码的ASN.1结构
  }
  // 通过pem将设置好的数据进行编码，并写入磁盘文件中
  efile, err := os.Create("private.pem")          // 创建一个写入文件流
  if err = pem.Encode(efile, block); err != nil { // 通过一个encode写入文件
    return
  }

  // ====================生成公钥==================
  // 从私钥中取出公钥
  dKey := privateKey.PublicKey
```

```
  // 通过x509标准将得到rsa私钥序列序列化为ASN.1的编码字符串
  data,err = x509.MarshalPKIXPublicKey(&dKey) // 这里的序列化公钥的API和私钥的API有差异
  if err != nil {
    panic(err)
  }
  // 将私钥字符串设置到pem格式块中
  block = &pem.Block{
    Type: "RSA PUBLIC KEY",
    Bytes: data,
  }
  // 通过pem将设置好的数据进行编码，并写入磁盘文件中
  dfile, err := os.Create("public.pem")// 创建一个写入文件流
  if err = pem.Encode(dfile,block);err != nil {
    return
  }
}
```

4.4 RSA加解密

- RSA加密
  - 将公钥文件中的公钥读出，得到使用pem编码的字符串；
  - 将得到的字符串解码；
  - 使用x509将编码之后的公钥解析出来；
  - 使用得到的公钥通过RSA进行数据加密；
- RSA解密
  - 将私钥文件中私钥独处，得到使用pem编码的字符串；
  - 将得到的字符串解码；
  - 使用x509将编码后的私钥解析出来；
  - 使用得到的私钥来解析密文得到明文；

- golang实现rsa加解密

```
func rsaEncrypt(plainText []byte, fileName string) []byte {
  // 读取私钥匙文件
  file, err := os.Open(fileName)
  if err != nil {
    panic(err)
  }
  fileInfo, _ := os.Stat(fileName)
  data := make([]byte, fileInfo.Size())
  if _, err = file.Read(data); err != nil {
    panic(err)
  }
  if err = file.Close(); err != nil {
    panic(err)
  }

  //使用pem进行数据解码
  block, _ := pem.Decode(data)
  if block == nil || block.Type != "RSA PUBLIC KEY" {
    log.Fatal("failed to decode PEM block containing public
key")
  }

  // 使用x509进行解析获取到公钥
  pub, err := x509.ParsePKIXPublicKey(block.Bytes)
  if err != nil {
    log.Fatal(err)
  }

  // 使用公钥进行加密
  dKey, ok := pub.(*rsa.PublicKey) // 进行类型断言
  if !ok {
    panic(err)
```

```
  }
  cipherText, err := rsa.EncryptPKCS1v15(rand.Reader, dKey,
plainText)
  if err != nil {
    panic(err)
  }
  return cipherText
}

func rsaDecrypt(cipherText []byte, fileName string) []byte {
  // 读取私钥
  file, err := os.Open(fileName)
  if err != nil {
    panic(err)
  }
  fileInfo, err := os.Stat(fileName)
  data := make([]byte, fileInfo.Size())
  if _, err = file.Read(data); err != nil {
    panic(err)
  }
  if err = file.Close(); err != nil {
    panic(err)
  }

  // 使用pem对数据进行解密
  block, _ := pem.Decode(data)
  if block == nil || block.Type != "RSA PRIVATE KEY" {
    log.Fatal("failed to decode PEM block containing private
key")
  }

  // 使用x509进行解析获得私钥
  eKey, err := x509.ParsePKCS1PrivateKey(block.Bytes)
```

```
    if err != nil {
        panic(err)
    }

    // 使用私钥对密文进行解密
    plainText, err := rsa.DecryptPKCS1v15(rand.Reader, eKey,
cipherText)
    if err != nil {
        panic(err)
    }
    return plainText
}
```

4.5 RSA完整代码⭐

```
package main

import (
    "crypto/rand"
    "crypto/rsa"
    "crypto/x509"
    "encoding/pem"
    "fmt"
    "log"
    "os"
)


// GenerateRsaKey bits 密钥长度
// GenerateRsaKey bits 密钥长度
func GenerateRsaKey(bits int) {
    // ====================生成私钥===================
    // 使用rsa中的GenerateKey方法生成私钥
```

```
  eKey, err := rsa.GenerateKey(rand.Reader, bits)
  if err != nil {
    panic(err)
  }
  // 通过x509标准将得到rsa私钥序列序列化为ASN.1的编码字符串
  data := x509.MarshalPKCS1PrivateKey(eKey)
  // 将私钥字符串设置到pem格式块中
  block := &pem.Block{
    Type:  "RSA PRIVATE KEY", // 得自前言的类型（如"RSA PRIVATE KEY"）
    Bytes: data,              // 内容解码后的数据，一般是DER编码的ASN.1结构
  }
  // 通过pem将设置好的数据进行编码，并写入磁盘文件中
  eFile, err := os.Create("private.pem")           // 创建一个写入文件流
  if err = pem.Encode(eFile, block); err != nil { // 通过一个encode写入文件
    return
  }

  // ====================生成公钥====================
  // 从私钥中取出公钥
  dKey := eKey.PublicKey
  // 通过x509标准将得到rsa私钥序列序列化为ASN.1的编码字符串
  data, err = x509.MarshalPKIXPublicKey(&dKey) // 这里的序列化公钥的API和私钥的API有差异
  if err != nil {
    panic(err)
  }
  // 将私钥字符串设置到pem格式块中
  block = &pem.Block{
    Type:  "RSA PUBLIC KEY",
```

```
    Bytes: data,
  }
  // 通过pem将设置好的数据进行编码，并写入磁盘文件中
  dFile, err := os.Create("public.pem") // 创建一个写入文件流
  if err = pem.Encode(dFile, block); err != nil {
    return
  }
}

func rsaEncrypt(plainText []byte, fileName string) []byte {
  // 读取私钥匙文件
  file, err := os.Open(fileName)
  if err != nil {
    panic(err)
  }
  fileInfo, _ := os.Stat(fileName)
  data := make([]byte, fileInfo.Size())
  if _, err = file.Read(data); err != nil {
    panic(err)
  }
  if err = file.Close(); err != nil {
    panic(err)
  }

  //使用pem进行数据解码
  block, _ := pem.Decode(data)
  if block == nil || block.Type != "RSA PUBLIC KEY" {
    log.Fatal("failed to decode PEM block containing public key")
  }

  // 使用x509进行解析获取到公钥
  pub, err := x509.ParsePKIXPublicKey(block.Bytes)
  if err != nil {
```

```
    log.Fatal(err)
  }

  // 使用公钥进行加密
  dKey, ok := pub.(*rsa.PublicKey) // 进行类型断言
  if !ok {
    panic(err)
  }
  cipherText, err := rsa.EncryptPKCS1v15(rand.Reader, dKey,
plainText)
  if err != nil {
    panic(err)
  }
  return cipherText
}

func rsaDecrypt(cipherText []byte, fileName string) []byte {
  // 读取私钥
  file, err := os.Open(fileName)
  if err != nil {
    panic(err)
  }
  fileInfo, err := os.Stat(fileName)
  data := make([]byte, fileInfo.Size())
  if _, err = file.Read(data); err != nil {
    panic(err)
  }
  if err = file.Close(); err != nil {
    panic(err)
  }

  // 使用pem对数据进行解密
  block, _ := pem.Decode(data)
```

```
  if block == nil || block.Type != "RSA PRIVATE KEY" {
    log.Fatal("failed to decode PEM block containing private
key")
  }

  // 使用x509进行解析获得私钥
  eKey, err := x509.ParsePKCS1PrivateKey(block.Bytes)
  if err != nil {
    panic(err)
  }

  // 使用私钥对密文进行解密
  plainText, err := rsa.DecryptPKCS1v15(rand.Reader, eKey,
cipherText)
  if err != nil {
    panic(err)
  }
  return plainText
}

func main() {
  // bits与加密明文长度有关联
  GenerateRsaKey(1024)
  text := "!@#qwe123"
  cipherData := rsaEncrypt([]byte(text), "public.pem")
  fmt.Println(string(cipherData))
  data := rsaDecrypt(cipherData, "private.pem")
  fmt.Println(string(data))
}
```

> 另外除了RSA，还有一个ECC椭圆曲线的加密方式。ECC是一种建立公开密钥加密的算法，基于椭圆曲线数学。椭圆曲线在密码学中的使用是在1985所创建的。ECC的主要优势是在某些情况下它比其他的方法使用更小的密钥——比如RSA加密算法提供更高等级的安全。椭圆曲线密码学的许多形式有稍微的不同，所有的都依赖于被广泛承认的解决椭圆曲线离散对数问题的困难性上。与传统的基于大质数因子分解困难性加密方法不同，ECC通过椭圆曲线方程式的性质产生密钥。
>
> ECC 164位的密钥产生的一个安全级别相当于RSA 1024位密钥提供的保密密度，而且计算量小，处理速度更快，存储空间和传输带宽占用较少。目前我国居民二代身份证正在使用256位的椭圆权限密码。虚拟货币比特币也选择ECC作为加密算法。椭圆曲线所依赖的数学难题是 k为整数，P是椭圆曲线上的点（称为基点，k*P=Q,已知P和Q，很难计算出k）

4.6 非对称加密的疑惑

- 非对称加密比对称加密的机密性更高吗?

  这个问题其实是无法回答的，因为机密性的高低与密钥长度和具体的加密方法都有关系。

- 采用1024bit密钥长度的非对称加密和采用128bit密钥长度的对称加密中，是密钥长度更长的非密钥加密更安全吗?

  不是。非对称加密的密钥长度不能于对称加密密钥长度进行直接比较。128bit的非对称加密的密码比起1024bit对称加密抵御暴力破解的能力更强。

- 有了非对称加密，对称加密就会被取代吗?

  不会。一般来说，在采用具备同等机密性的密钥长度的情况下，非对称加密的处理速度只有对称加密的几百分之一。因此，非对称加密并不适用于对很长的消息内容进行加密。根据目的的不同，还有可能会配合使用对称加密和非对称加密。例如，混合密码系统就是将这两种密码组合而成的。

6.哈希算法

`哈希算法`也被叫做单向散列函数、杂凑函数、消息摘要函数，通过单向散列函数，可以获取消息的指纹。接受的输入也叫做`原像`，输出叫做`三列值、哈希值、指纹、摘要`。

6.1 单项散列函数的定义

简单来说，就是通过哈希算法，我们可以得到一段标识的唯一信息。

6.2 单项散列函数的性质

通过使用单向散列函数，即便是确定几百MB大小的文件完整性，也只要对比很短的散列值就可以了。其一般具备以下的性质。

- 根据任意长度的消息计算出固定长度的散列值。

  即单向散列函数的输入必须能够是任意长度的消息。同时，无论输出的多么长的消息，单项散列函数必须能够生成长度很短的散列值，散列值的长度最好是短且固定的。

- 能够快速计算出散列值

  计算散列值所花费的时间必要短。尽管消息越长，计算散列值的时间也越长，但如果不能在现实时间内完成计算就没有意义了。

- 消息不同散列值也不同

  为了能够确认完整性，消息中哪怕只有1比特的改变，也必须有很高概率产生不同的散列值。

  如果单向散列函数计算出的散列值没有发生变化，那么消息很容易就会被篡改，这个单向散列函数也就无法被用于完整性的检查。**两个不同的消息产生同一个散列值的情况称为碰撞（collision）**。如果要将单向散列函数用于完整性检测，则需要确保在事实上不可能被认为地发生碰撞。

- 具备单向性

  单向散列函数必须具备单向性（one-way）。单向性指的是无法通过散列值反算出消息的性质，根据消息计算散列值可以很容易，但这条单行路是无法过来走的。因此严格的来说，单向散列函数并不是一种加密，因为无法通过解密将散列值还原为原来的消息。

6.3 单向散列函数的实际应用

以下是单向散列函数实际应用的场景。

- 检测软件是否被修改

  我们可以通过单向散列函数来确认自己下载的软件是否被篡改。很多软件，尤其是安全相关的软件都会通过单向散列函数计算出的散列值公布在自己的官方网站上。用户下载到软件之后，可以自行计算散列值，然后与官方网站上公布的散列值进行对比。通过散列值，用户可以确认自己下载到软件与软件作者所提供的软件是否一致。这样的方法，在可以通过多种途径得到团建的情况下非常有用。为了减轻服务器的压力，很多软件作者都会借助多个网站（镜像站点）来发布软件，在这种情况下，单向散列函数就会在检测软件是否被篡改方面发挥重要作用。

- 消息认证码

  使用单向散列函数可以构造消息认证码。消息认证码是将“发送者和接受者之间共享的密钥”和“消息”，进行混合计算出的散列值，使用消息认证码可以检测并防止通信过程中的错误、篡改以及伪装。消息认证码在SSL/TLS中也得到了运用，关于SSL/TLS我们将在后面几章中介绍。

- 数字签名

  在进行消息签名时，我们也会使用这样的单向散列函数。数字签名是现实社会中的签名（sign）和盖章这样的行为在数字世界中的实现。数字签名的处理非常耗时，因此一般不会对整个消息内容直接施加数字前面，而是先通过单向散列函数计算出消息的散列值，然后再对这个散列值施加数字签名。

- 伪随机数生成器

  使用单向散列函数可以伪造随机数生成器。密码技术中所使用的随机数需要具备事实上不可能根据过去的随机数列预测未来的随机数列这样的性质。为了保证不可预测性，可以利用单向散列函数的单向性。

- 一次性口令

  使用单向散列函数可以构造一次性口令（one-time password）。一次性口令经常被用于服务器对对客户端合法性认证。在这种方式中，通过使用单向散列函数可以保证口令只在通信链路上传送一次，因此即使窃听者窃听了口令，也无法使用。

6.4 常用的单向散列函数

6.4.1 MD4、MD5

MD4是于1990年设计的单向散列函数，能够产生128bit的散列值。不过，随着有人提出了寻找MD4的碰撞方法，因此现在它已经不安全了。MD5在1995应运而生，能够产生128bit的散列值。如今MD5的强碰撞性已经被攻破，也就说，现在已经能够产生具备相同的散列值的两条不同消息，因此它也已经不安全了。（MD是Message Digest）的缩写。

6.4.2 Go中使用MD5

```
data := []byte("These pretzels are making me thirsty.")
fmt.Printf("%x", md5.Sum(data)) //
Output:b0804ec967f48520697662a204f5fe72
```

6.4.3 SHA-1、SHA-224、SHA-256、SHA-384、SHA-512

SHA-1是有INST设计的一种能够产生160比特的散列值的单向散列函数。SHA-1的消息长度存在上线（接近于2^{64}bit）是个非常巨大的数值，因此在实际应用中没有问题。

SHA-256、SHA-384和SHA-512都是由NIST设计的单向散列函数，他们的散列值长度分别是256bit、384bit和512bit。这些单向散列函数合起来统称为SHA-2，他们的消息长度也存在上限，不过实际使用中应该没有问题。而SHA1的强碰撞性已于2005年被攻破。而SHA-2还尚未攻破。

单向散列函数	比特数	字节数
MD4	128bit	16byte
MD5	128bit	16byte
SHA-1	160bit	20byte
SHA-224	224bit	28byte
SHA-256	256bit	32byte
SHA-384	384bit	48byte
SHA-512	512bit	64byte

6.4.4 Go中使用SHA-1、SHA-2

- SHA-1【伪代码】

```
data := []byte("This page intentionally left blank.")
fmt.Printf("% x", sha1.Sum(data)) // Output:af 06 49 23 bb f2
30 15 96 aa c4 c2 73 ba 32 17 8e bc 4a 96
```

- SHA-256【伪代码】

```
import "encoding/hex"
import "crypto/sha256"

// 通过new得到一个hash对象
hash := sha256.New()
// 往hash对象中添加数据即可【如果数据过大可以多次添加】
hash.Hash.Write([]byte("1This page intentionally left blank."))
hash.Hash.Write([]byte("2This page intentionally left blank."))
// 计算结果
sha256 := hash.Sum()
// 散列值一般是一个二进制的字符串，有些字符不可见，需要格式化16进制的字符串0-9 a-f
// 注意进制转换后字节长度会翻倍
sha256_16 := hex.EncodeToString(sha256) // 32字节
```

其他的单向散列接口类似，使用的时候直接查询文档即可。