

## 常用密码技术

### 1 密码

#### 1.1 发送者、接收者和窃听者

#### 1.2 加密和解密

#### 1.3 密钥

##### 1.3.1 密码算法

##### 1.3.2 密钥

#### 1.4 凯撒密码

##### 1.4.1 凯撒密码的加密

##### 1.4.2 凯撒密码的解密

#### 1.5 密码信息安全常识与威胁

##### 1.5.1 密码信息安全常识

不要使用保密的密码算法

使用低强度的密码比不进行任何加密更危险

任何密码总有一天会被破译

密码只是信息安全的一部分

##### 1.5.2 密码信息威胁

### 2. 对称加密

#### 2.1 编码

#### 2.2 DES

##### 2.2.1 什么是DES

##### 2.2.2 加密和解密

##### 2.2.3 Go中对DES的操作

加解密实现思路

加解密的代码实现

#### 2.3 三重DES

##### 2.3.1 三重DES的加密

##### 2.3.2 Go中对3DES的操作

加解密实现思路

加解密的代码实现

#### 2.4 AES

##### 2.4.2 AES的加密和解密

##### 2.4.2 Go中对AES的使用

加解密实现思路

加解密的代码实现

#### 2.5 应选择哪种对称加密

本章小结

### 3. 分组密码的模式

#### 3.1 分组密码

#### 3.2 模式

明文分组和密文分组

#### 3.3 ECB 模式

#### 3.3 CBC模式

XOR

CBC模式

- 初始化向量
- 3.4 CFB 模式
  - 初始化向量
  - CFB模式与流密码
- 3.5 OFB 模式
  - 初始化向量
  - CFB模式和OFB模式对比
- 3.6 CTR 模式
  - 计数器的生成方法
  - OFB模式与CTR模式对比
  - CTR模式的特点
- 3.7 总结
- 4. 非对称加密
  - 4.1 非对称加密通信流程
  - 4.2 RSA
    - 4.2.1 RSA加密
    - 4.2.2 RSA解密
    - 4.2.3 Go中生成公钥和私钥
    - 4.2.4 Go中使用RSA
  - 4.3 ECC椭圆曲线
  - 4.4 非对称加密解惑
- 5. 单向散列函数
  - 5.1 什么是单向散列函数
  - 5.2 关于术语
  - 5.3 单向散列函数的性质
  - 5.4 单向散列函数的实际应用
    - 5.4.1 检测软件是否被篡改
    - 5.4.2 消息认证码
    - 5.4.3 数字签名
    - 5.4.6 伪随机数生成器
    - 5.4.7 一次性口令
  - 5.5 常用的单向散列函数
    - 5.5.1 MD4、MD5
    - 5.5.2 Go中使用MD5
    - 5.5.3 SHA-1、SHA-224、SHA-256、SHA-384、SHA-512
    - 5.5.4 Go中对SHA-1、SHA-2的使用
- 6. 消息认证码
  - 6.1 什么是消息认证码
  - 6.2 消息认证码的使用步骤
  - 6.3 HMAC
    - 6.3.1 HMAC介绍
    - 6.3.2 Go中对HMAC的使用
  - 6.3 消息认证码的密钥配送问题
  - 6.4 消息认证码无法解决的问题
    - 6.4.1 对第三方证明
    - 6.4.2 防止否认
  - 6.5 总结
- 7. 数字签名

- 7.1 从消息认证到数字签名
- 7.2 签名的生成和验证
- 7.3 非对称加密和数字签名
- 7.3 数字签名的方法
- 7.4 通过RSA实现数字签名
  - 7.4.1 生成数字签名
  - 7.4.2 验证数字签名
- 7.5 使用椭圆曲线实现数字签名
- 7.6 数字签名无法解决的问题
- 8. 证书
  - 8.1 证书的应用场景
  - 8.2 证书标准规范X.509
    - 8.2.1 证书规范
    - 8.2.2 证书格式
    - 8.2.3 CA证书
  - 8.3 公钥基础设施 (PKI)
    - 8.3.1 什么是公钥基础设施
    - 8.3.2 PKI的组成要素
      - 用户
      - 认证机构 (CA)
      - 仓库
    - 8.3.3 各种各样的PKI
- 9. SSL/TLS
  - 9.1 客户端与服务器
  - 9.2 用SSL/TLS承载HTTP
  - 9.3 https
    - 9.3.1 http和https
    - 9.3.2 https优缺点
- 参考资料
  - PKCS15个标准

# 常用密码技术

## 1 密码

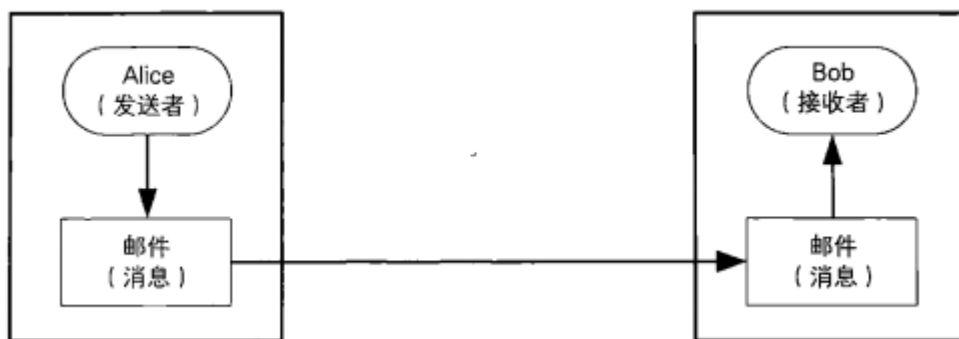
### 1.1 发送者、接收者和窃听者

#

请想象一个Alice向Bob发送电子邮件的场景。在这个场景中，发出邮件的Alice称为 **发送者 (sender)**，而收到邮件的Bob则称为 **接收者 (receiver)**。

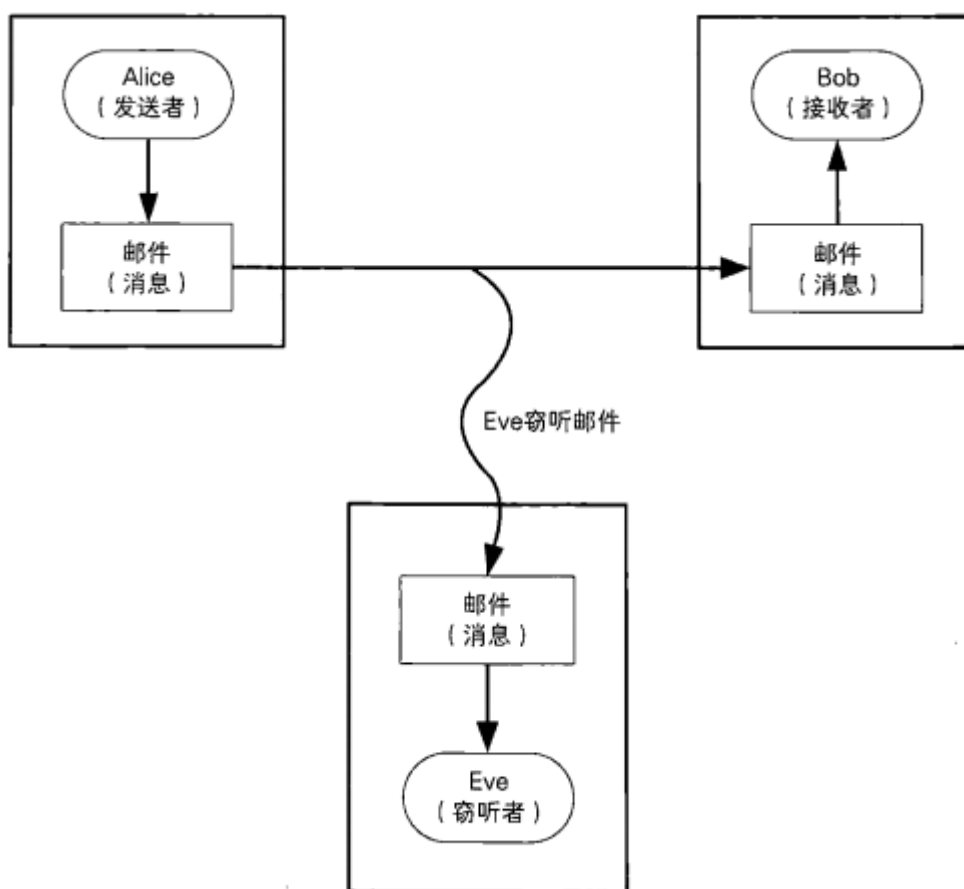
在讲解发送者、接收者的概念时，用邮件这个例子会比较便于理解，但实际上发送者和接收者这两个术语的使用范围并不仅仅局限于邮件。当某个人向另一个人发送信息时，发出信息的人称为发送者，而收到信息的人称为接收者。另外，被发送的信息有时也统称为 **消息 (message)**。

- Alice向Bob发送邮件



邮件是通过互联网从Alice的计算机发送到Bob的计算机的。在发送邮件时，邮件会经过许多台计算机和通信设备进行中转，在这个过程中，就存在被恶意窃听者（eavesdropper）偷看到的可能性。

- Eve（窃听者）看到邮件的内容



窃听者Eve并不一定是人类，有可能是安装在通信设备上的某种窃听器，也可能是安装在邮件软件和邮件服务器上的某些程序。

尽管邮件内容原本应该只有发送者和接收者两个人知道，但如果不采取相应的对策，就存在被第三方知道的风险。

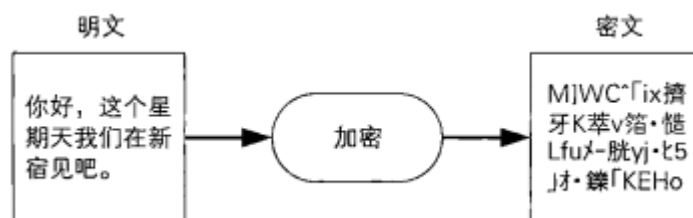
## 1.2 加密和解密

Alice不想让别人看到邮件的内容，于是她决定将邮件进行**加密**（encrypt）后再发送出去。

加密之前的消息称为**明文**（plaintext），加密之后的消息称为**密文**（cipher-text）。

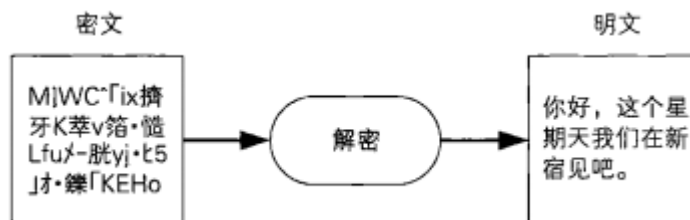
我们看到明文可以理解其中的含义，而看到密文则无法理解其中的含义。

- **明文加密之后就会变成看不懂的密文**



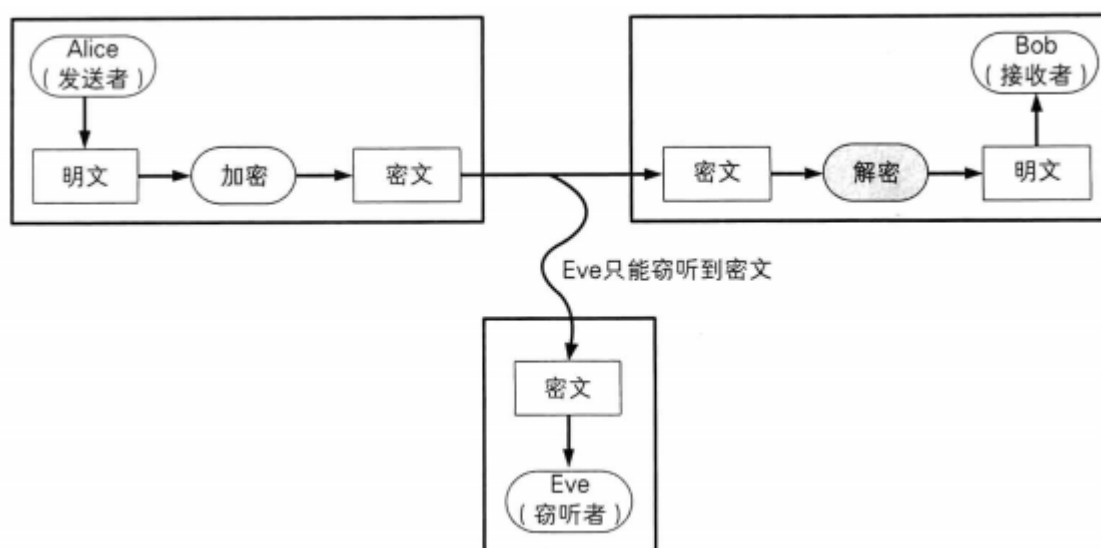
Bob收到了来自Alice的加密邮件，但作为接收者的Bob也是无法直接阅读密文的，于是 Bob需要对密文进行解密（decrypt）之后再阅读。解密就是将密文恢复成明文的过程。

- **密文解密之后就变成了原来的明文**



将消息加密后发送的话，即使消息被窃听，窃听者得到的也只是密文，而无法得知加密前的明文内容

- **将消息加密后发送, 窃听者只能得到密文**



在上述场景中，Alice将邮件进行加密，而Bob则进行解密，这样做的目的，是为了不让窃听者Eve读取邮件的内容Alice和Bob通过运用**密码**(cryptography)技术，保证了邮件的**机密性**（confidentiality）。

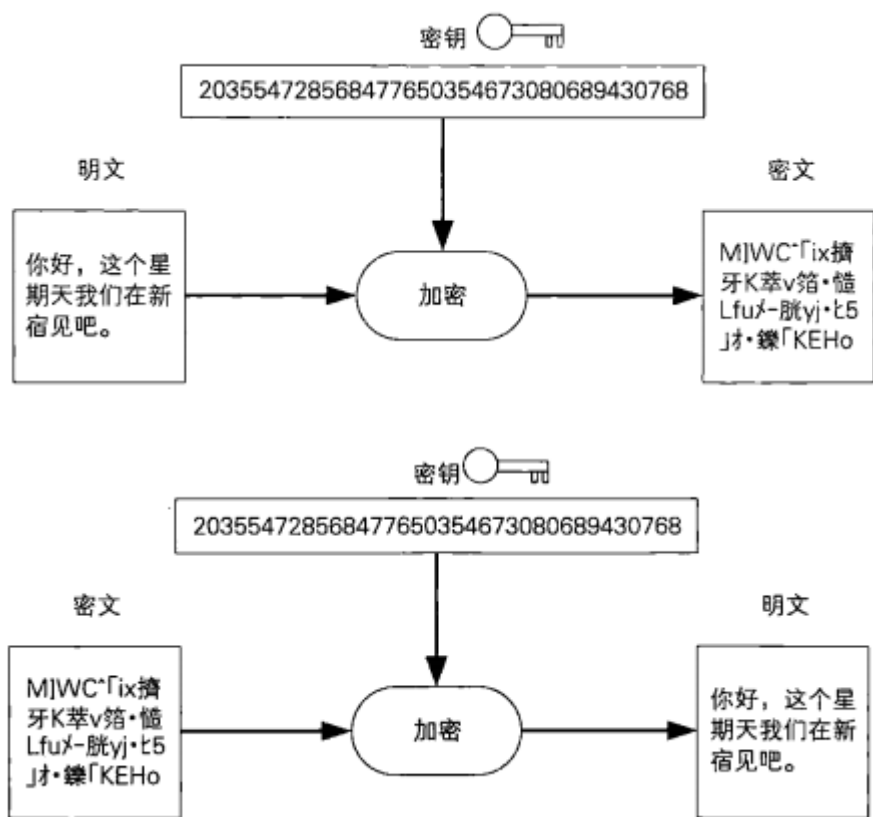
### 1.3.1 密码算法

用于解决复杂问题的步骤，通常称为**算法 (algorithm)**。从明文生成密文的步骤，也就是加密的步骤，称为“加密算法”，而解密的步骤则称为“解密算法”。加密、解密的算法合在一起统称为**密码算法**。

### 1.3.2 密钥

密码算法中需要**密钥 (key)**。现实世界中的“钥”，是像🔑这样的形状微妙而复杂的小金属片。然而，密码算法中的密钥，则是像203554728568477650354673080689430768这样的一串非常大的数字。

#### • 加密、解密与密钥



无论是在加密时还是在解密时，都需要知道密钥。

正如保险柜的钥匙可以保护保险柜中存放的贵重物品一样，密码中的密钥可以保护你的重要数据。即使保险箱再坚固，如果钥匙被盗，里面的贵重物品也会被盗。同样地我们也必须注意不要让密码的密钥被他人窃取。

## 1.4 凯撒密码

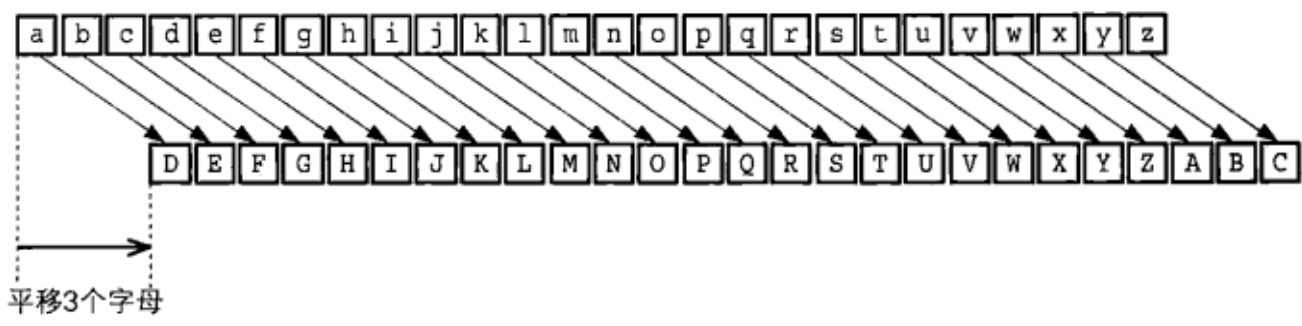
#

恺撒密码 (Caesar cipher) 是一种相传尤利乌斯·恺撒曾使用过的密码。恺撒于公元前100年左右诞生于古罗马，是一位著名的军事统帅。

**恺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。**比如在日语（例如平假名）或者汉语（例如汉语拼音）或者英文字母表中都可以用同样的思路来实现恺撒密码。

为了讲解方便，我们用小写字母 (a, b, c, ...) 来表小明文，用大写字母 (A, B, C, ...) 来表示密文。

现在我们将字母表平移3个字母，于是，明文中的a在加密后就变成了与其相隔3个字母的D，以此类推。b变成E，c变成F，d变成G.....v变成Y，w变成Z，而x则会回到字母表的开头而变成A，相应地，y变成B，z变成C。通过下图我们可以很容易地理解“平移”的具体工作方式。



1.4.1 凯撒密码的加密

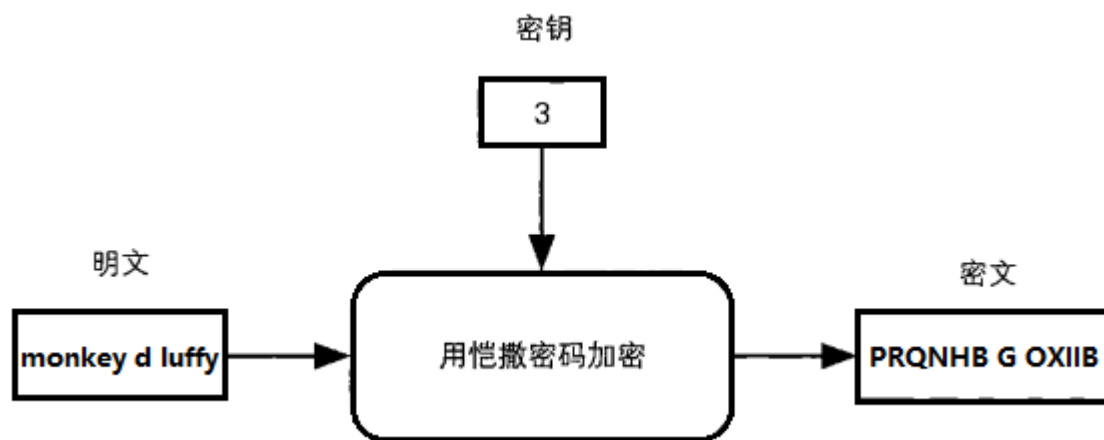
这里，我们假设要保密的信息为 `monkey d luffy` 这个男孩的名字。我们暂且不管这个名字到底代表一位真实的男性，还是只是一种暗号，只考虑将它在保密的状态下发送给接收者。

此时，明文包含下列12个字母：`monkey d luffy`，接下来我们对明文中的字母逐一加密：

1	m	---->	P
2	o	---->	R
3	n	---->	Q
4	k	---->	N
5	e	---->	H
6	y	---->	B
7	d	---->	G
8	l	---->	O
9	u	---->	X
10	f	---->	I
11	f	---->	I
12	y	---->	B

这样，明文 `monkey d luffy` 就被转换成了密文 `PRQNH B G OXIIB`，`monkey d luffy`这个词我们能够看懂，但 `PRQNH B G OXIIB`就看不懂了。

恺撒密码中，将字母表中的字母平移这个操作就是密码的算法，而平移的字母数量则相当于密钥。在上面的例子中，密钥为3（如下图）。



### 1.4.2 凯撒密码的解密

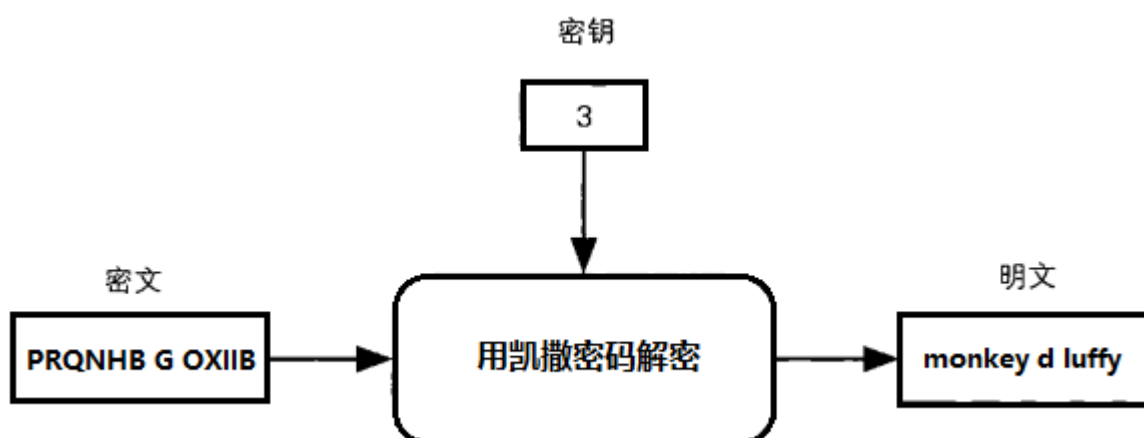
现在，假设接收者已经收到了密文PRQNH B G OXIIB，由于密文本身是看不懂的，因此必须将它解密成明文。

恺撒密码的解密过程是使用与加密时相同的密钥进行反向的平移操作。用刚才的例子来说，只要反向平移3个字母就可以解密了。

1	P	---->	m
2	R	---->	o
3	Q	---->	n
4	N	---->	k
5	H	---->	e
6	B	---->	y
7	G	---->	d
8	O	---->	l
9	X	---->	u
10	I	---->	f
11	I	---->	f
12	B	---->	y

这样我们就得到了明文monkey d luffy。

在这个场景中，密钥3必须由发送者和接收者事先约定好。





## 1.5 密码信息安全常识与威胁

### 1.5.1 密码信息安全常识

在继续下面的内容之前，我们先来介绍一些关于密码的常识。刚刚开始学习密码的人常常会对以下这几条感到不可思议，因为它们有悖于我们的一般性常识。

- 不要使用保密的密码算法
- 使用低强度的密码比不进行任何加密更危险
- 任何密码总有一天都会被破解
- 密码只是信息安全的一部分

#### 不要使用保密的密码算法

很多企业都有下面这样的想法：

“由公司自己开发一种密码算法，并将这种算法保密，这样就能保证安全。然而，这样的想法却是大错特错，使用保密的密码算法是无法获得高安全性的。我们不应该制作或使用任何保密的密码算法，而是应该使用那些已经公开的、被公认为强度较高的密码算法。

这样做的原因主要有以下两点：

#### • 密码算法的秘密早晚会公诸于世

从历史上看，密码算法的秘密最终无一例外地都会被暴露出来。例如：RSA公司开发的RC4密码算法曾经也是保密的，但最终还是有一位匿名人士开发并公开了与其等效的程序。

一旦密码算法的详细信息被暴露，依靠对密码算法本身进行保密来确保机密性的密码系统也就土崩瓦解了。反之，那些公开的算法从一开始就没有设想过要保密，因此算法的暴露丝毫不会削弱它们的强度。

#### • 开发高强度的密码算法是非常困难的

- 要比较密码算法的强弱是极其困难的，因为密码算法的强度并不像数学那样可以进行严密的证明。密码算法的强度只能通过事实来证明，如果专业密码破译者经过数年的尝试仍然没有破解某个密码算法，则说明这种算法的强度较高。
- 稍微聪明一点的程序员很容易就能够编写出“自己的密码系统”。这样的密码在外行看来貌似牢不可破，但在专业密码破译者的眼里，要破解这样的密码几乎是手到擒来。
- 现在世界上公开的被认为强度较高的密码算法，几乎都是经过密码破译者长期尝试破解未果而存活下来的。因此，如果认为“公司自己开发的密码系统比那些公开的密码系统更强”，那只能说是过于高估自己公司的能力了。
- 试图通过对密码算法本身进行保密来确保安全性的行为，一般称为隐蔽式安全性（securitybyobscurity），这种行为是危险且愚蠢的。
- 反过来说，将密码算法的详细信息以及程序源代码全部交给专业密码破译者，并且为其提供大量的明文和密文样本，如果在这样的情况下破译一段新的密文依然需要花上相当长的时间，就说明这是高强度的密码。

#### 使用低强度的密码比不进行任何加密更危险

一般人们会认为：就算密码的强度再低，也比完全不加密要强吧？其实这样的想法是非常危险的。

**正确的想法应该是：**与其使用低强度的密码，还不如从一开始就不使用任何密码这主要是由于用户容易通过“密码”这个词获得一种“错误的安全感”。对于用户来说，安全感与密码的强度无关，而只是由“信息已经被加密了”这一事实产生的，而这通常会导致用户在处理一些机密信息的时候麻痹大意。

### 任何密码总有一天会被破译

如果某种密码产品宣称“本产品使用了绝对不会被破解的密码算法”，那么你就要对这个产品的安全性打个问号了，这是因为绝对不会被破解的密码是不存在的。

无论使用任何密码算法所生成的密文，只要将所有可能的密钥全部尝试一遍，就总有一天可以破译出来。因此，破译密文所需要花费的时间，与要保密的明文的价值之间的权衡就显得非常重要。

### 密码只是信息安全的一部分

我们还是回到Alice给Bob发送加密邮件的例子。即便不去破解密码算法，也依然有很多方法能够知道Alice所发送的邮件内容，例如：

**攻击者可以不去试图破译经过加密的邮件，而是转而攻击Alice的电脑以获取加密之前的邮件明文。**

上面提到的攻击手段，都与密码的强度毫无关系。要保证良好的安全性，就需要理解“系统”这一概念本身的性质复杂的系统就像一根由无数个环节相连组成的链条，如果用力拉，链条就会从其中最脆弱的环节处断开。因此，系统的强度取决于其中最脆弱的环节的强度。

最脆弱的环节并不是密码，而是人类自己。

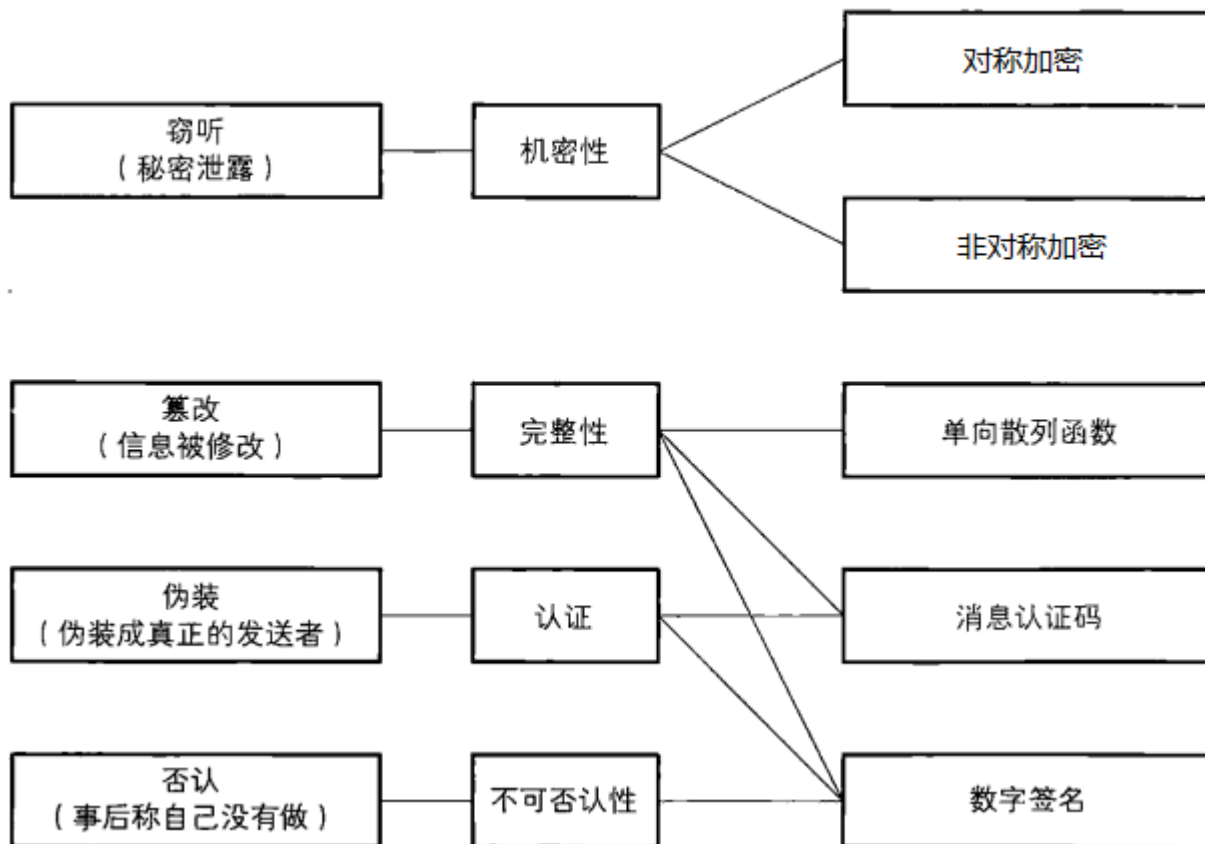
#### 1.5.2 密码信息威胁

我们将信息安全所面临的威胁与用来用对这些威胁的密码技术直接的关系用一张图标来表示出来。

信息安全所面临的威胁

受威胁的特性

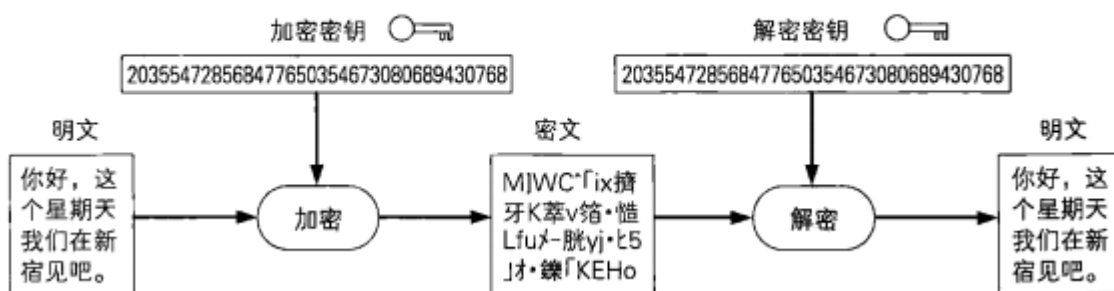
用来应对的密码技术



## 2. 对称加密

1 "对称加密：也称为对称密码，是指在加密和解码时使用同一密钥的加密方式"

对称密码中，加密用的密钥和  
解密用的密钥是相同的



### 2.1 编码

#

现代的密码都是建立在计算机的基础之上的，这是因为现代的密码所处理的数据量非常大，而且密码算法也非常复杂，不借助计算机的力量就无法完成加密和解密的操作。

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**。无论是文字、图像、声音、视频还是程序，在计算机中都是用比特序列来表示的。执行加密操作的程序，就是将表示明文的比特序列转换为表示密文的比特序列。

将现实世界中的东西映射为比特序列的操作称为**编码（encoding）**。例如midnight（深夜）这个词，我们可以对其中的每个字母逐一进行编码，这种编码规则叫作ASCII。

```
m → 01101101
i → 01101001
d → 01100100
n → 01101110
i → 01101001
g → 01100111
h → 01101000
t → 01110100
```

注意这里的m --> 01101101这一转换并不是加密而是编码。尽管在人类看来0和1的序列跟密码没什么两样，但计算机却可以“看懂”这些比特序列，并很快地反应出其所对应的字符 midnight

		109
HEX	6D	m的ASCII十进制值: 109 m的二进制值: 0110 1101
DEC	109	
OCT	155	
BIN	0110 1101	

## 2.2 DES

#

### 2.2.1 什么是DES

DES（Data Encryption Standard）是1977年美国联邦信息处理标准（FIPS）中所采用的一种对称密码（FIPS46.3）。DES一直以来被美国以及其他国家的政府和银行等广泛使用。然而，随着计算机的进步，现在DES已经能够被暴力破解，强度大不如前了。

RSA公司举办过破译DES密钥的比赛（DESChallenge），我们可以看一看RSA公司官方公布的比赛结果：

- 1997年的DES Challenge1中用了96天破译密钥
- 1998年的DES ChallengeI-1中用了41天破译密钥
- 1998年的DES ChallengeII-2中用了56小时破译密钥
- 1999年的DES ChallengeIII中只用了22小时15分钟破译密钥

由于DES的密文可以在短时间内被破译，因此除了用它来解密以前的密文以外，现在我不应该再使用DES了。

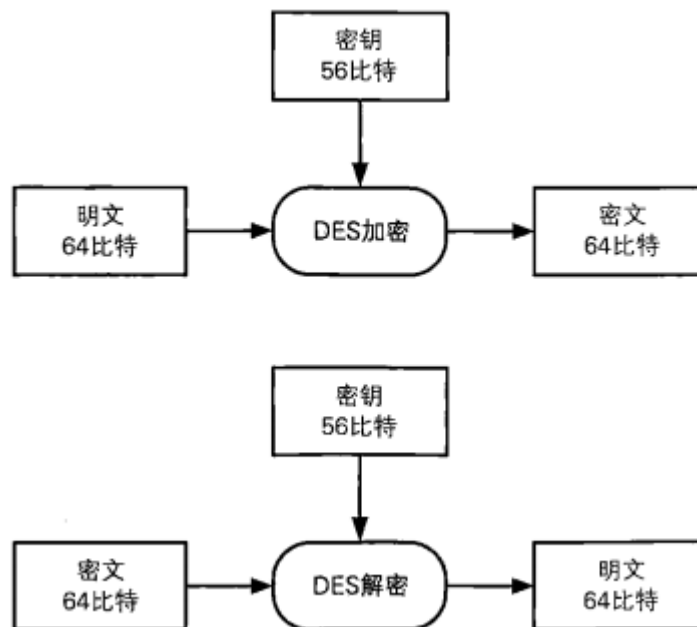
### 2.2.2 加密和解密

DES是一种将64比特的明文加密成64比特的密文的对称密码算法，它的密钥长度是56比特。尽管从规格上来说，DES的密钥长度是64比特，但由于每隔7比特会设置一个用于错误检查的比特，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）为一个单位来进行加密的，这个64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（blockcipher）**，DES就是分组密码的一种。

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（反复），而迭代的具体方式就称为模式（mode）。

- DES的加密与解密 - 图例



### 2.2.3 Go中对DES的操作

#### 加解密实现思路

- 加密 - CBC分组模式

1. 创建并返回一个使用DES算法的cipher.Block接口
  - 密钥长度为64bit, 即  $64/8 = 8$  字节(byte)
2. 对最后一个明文分组进行数据填充
  - DES是以64比特的明文（比特序列）为一个单位来进行加密的
  - 最后一组不够64bit, 则需要进行数据填充( 参考第三章)
3. 创建一个密码分组为链接模式的, 底层使用DES加密的BlockMode接口
4. 加密连续的数据块

- 解密

1. 创建并返回一个使用DES算法的cipher.Block接口
2. 创建一个密码分组为链接模式的, 底层使用DES解密的BlockMode接口
3. 数据块解密
4. 去掉最后一组的填充数据

#### 加解密的代码实现

在Go中使用DES需要导入的包:

```

1  import (
2      "crypto/des"
3      "crypto/cipher"
4      "fmt"
5      "bytes"
6  )

```

DES加密代码:

```

1  // src -> 要加密的明文
2  // key -> 密钥, 大小为: 8byte
3  func DesEncrypt_CBC(src, key []byte) []byte{
4      // 1. 创建并返回一个使用DES算法的cipher.Block接口
5      block, err := des.NewCipher(key)
6      // 2. 判断是否创建成功
7      if err != nil{
8          panic(err)
9      }
10     // 3. 对最后一个明文分组进行数据填充
11     src = PKCS5Padding(src, block.BlockSize())
12     // 4. 创建一个密码分组为链接模式的, 底层使用DES加密的BlockMode接口
13     //     参数iv的长度, 必须等于b的块尺寸
14     tmp := []byte("helloAAA")
15     blackMode := cipher.NewCBCEncrypter(block, tmp)
16     // 5. 加密连续的数据块
17     dst := make([]byte, len(src))
18     blackMode.CryptBlocks(dst, src)
19
20     fmt.Println("加密之后的数据: ", dst)
21
22     // 6. 将加密数据返回
23     return dst
24 }

```

DES解密代码

```

1  // src -> 要解密的密文
2  // key -> 密钥, 和加密密钥相同, 大小为: 8byte
3  func DesDecrypt_CBC(src, key []byte) []byte {
4      // 1. 创建并返回一个使用DES算法的cipher.Block接口
5      block, err := des.NewCipher(key)
6      // 2. 判断是否创建成功
7      if err != nil{
8          panic(err)
9      }
10     // 3. 创建一个密码分组为链接模式的, 底层使用DES解密的BlockMode接口
11     tmp := []byte("helloAAA")
12     blockMode := cipher.NewCBCDecrypter(block, tmp)
13     // 4. 解密数据
14     dst := src
15     blockMode.CryptBlocks(src, dst)
16     // 5. 去掉最后一组填充的数据
17     dst = PKCS5UnPadding(dst)
18

```

```

19      // 6. 返回结果
20      return dst
21  }

```

最后一个分组添加填充数据和移除添加数据代码

```

1  // 使用pkcs5的方式填充
2  func PKCS5Padding(ciphertext []byte, blockSize int) []byte{
3      // 1. 计算最后一个分组缺多少个字节
4      padding := blockSize - (len(ciphertext)%blockSize)
5      // 2. 创建一个大小为padding的切片, 每个字节的值为padding
6      padText := bytes.Repeat([]byte{byte(padding)}, padding)
7      // 3. 将padText添加到原始数据的后边, 将最后一个分组缺少的字节数补齐
8      newText := append(ciphertext, padText...)
9      return newText
10 }
11
12 // 删除pkcs5填充的尾部数据
13 func PKCS5UnPadding(origData []byte) []byte{
14     // 1. 计算数据的总长度
15     length := len(origData)
16     // 2. 根据填充的字节值得到填充的次数
17     number := int(origData[length-1])
18     // 3. 将尾部填充的number个字节去掉
19     return origData[:length-number]
20 }

```

测试函数

```

1  func DESText() {
2      // 加密
3      key := []byte("11111111")
4      result := DesEncrypt_CBC([]byte("床前明月光，疑是地上霜。举头望明月，低头思故乡."), key)
5      fmt.Println(base64.StdEncoding.EncodeToString(result))
6      // 解密
7      result = DesDecrypt_CBC(result, key)
8      fmt.Println("解密之后的数据: ", string(result))
9  }

```

重要的函数说明

1. 生成一个底层使用DES加/解密的Block接口对象

```

1  函数对应的包: import "crypto/des"
2  func NewCipher(key []byte) (cipher.Block, error)
3      - 参数 key: des对称加密使用的密码, 密码长度为64bit, 即8byte
4      - 返回值 cipher.Block: 创建出的使用DES加/解密的Block接口对象

```

2. 创建一个密码分组为CBC模式, 底层使用b加密的BlockMode接口对象

```

1  函数对应的包: import "crypto/cipher"
2  func NewCBCEncrypter(b Block, iv []byte) BlockMode
3      - 参数 b: 使用des.NewCipher函数创建出的Block接口对象
4      - 参数 iv: 事先准备好的一个长度为一个分组长度的比特序列, 每个分组为64bit, 即8byte
5      - 返回值: 得到的BlockMode接口对象

```

### 3. 使用cipher包的BlockMode接口对象对数据进行加/解密

```

1  接口对应的包: import "crypto/cipher"
2  type BlockMode interface {
3      // 返回加密字节块的大小
4      BlockSize() int
5      // 加密或解密连续的数据块, src的尺寸必须是块大小的整数倍, src和dst可指向同一内存地址
6      CryptBlocks(dst, src []byte)
7  }
8  接口中的 CryptBlocks(dst, src []byte) 方法:
9      - 参数 dst: 传出参数, 存储加密或解密运算之后的结果
10     - 参数 src: 传入参数, 需要进行加密或解密的数据切片(字符串)

```

### 4. 创建一个密码分组为CBC模式, 底层使用b解密的BlockMode接口对象

```

1  函数对应的包: import "crypto/cipher"
2  func NewCBCDecrypter(b Block, iv []byte) BlockMode
3      - 参数 b: 使用des.NewCipher函数创建出的Block接口对象
4      - 参数 iv: 事先准备好的一个长度为一个分组长度的比特序列, 每个分组为64bit, 即8byte,
5                该序列的值需要和NewCBCEncrypter函数的第二个参数iv值相同
6      - 返回值: 得到的BlockMode接口对象

```

### 5. 自定义函数介绍

```

1  对称加密加密需要对数据进行分组, 保证每个分组的数据长度相等, 如果最后一个分组长度不够, 需要进行填充
2  func PKCS5Padding(ciphertext []byte, blockSize int) []byte
3      - 参数 ciphertext: 需要加密的原始数据
4      - 参数 blockSize: 每个分组的长度, 跟使用的加密算法有关系
5          * des: 64bit, 8byte
6          * 3des: 64bit, 8byte
7          * aes: 128bit, 16byte

```

## 2.3 三重DES

#

现在DES已经可以在现实的时间内被暴力破解, 因此我们需要一种用来替代DES的分组密码, 三重DES就是出于这个目的被开发出来的。

**三重DES (triple-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, 通常缩写为3DES。**

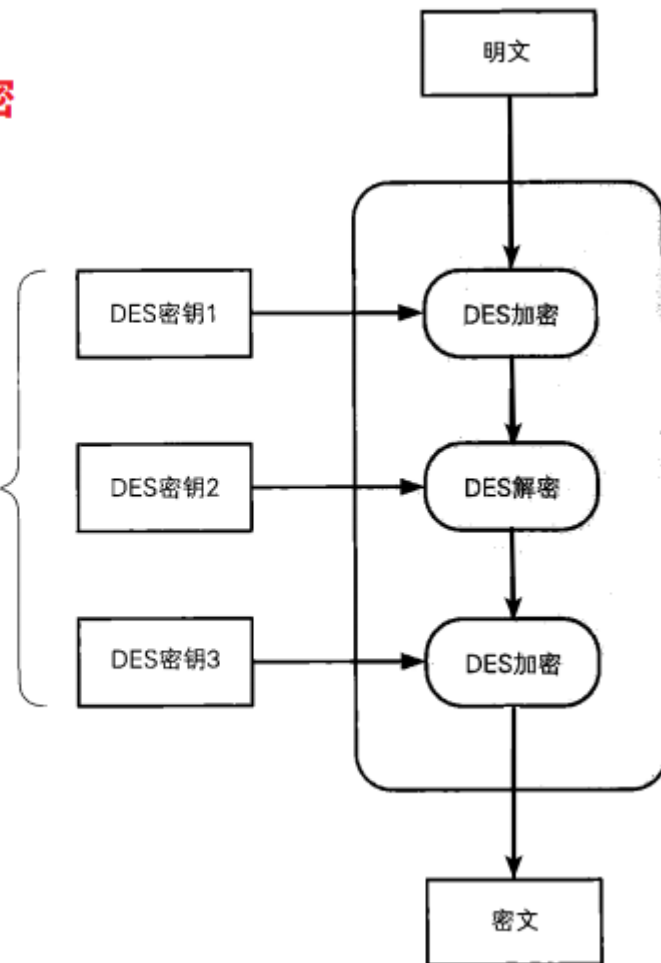
### 2.3.1 三重DES的加密

三重DES的加解密机制如图所示:

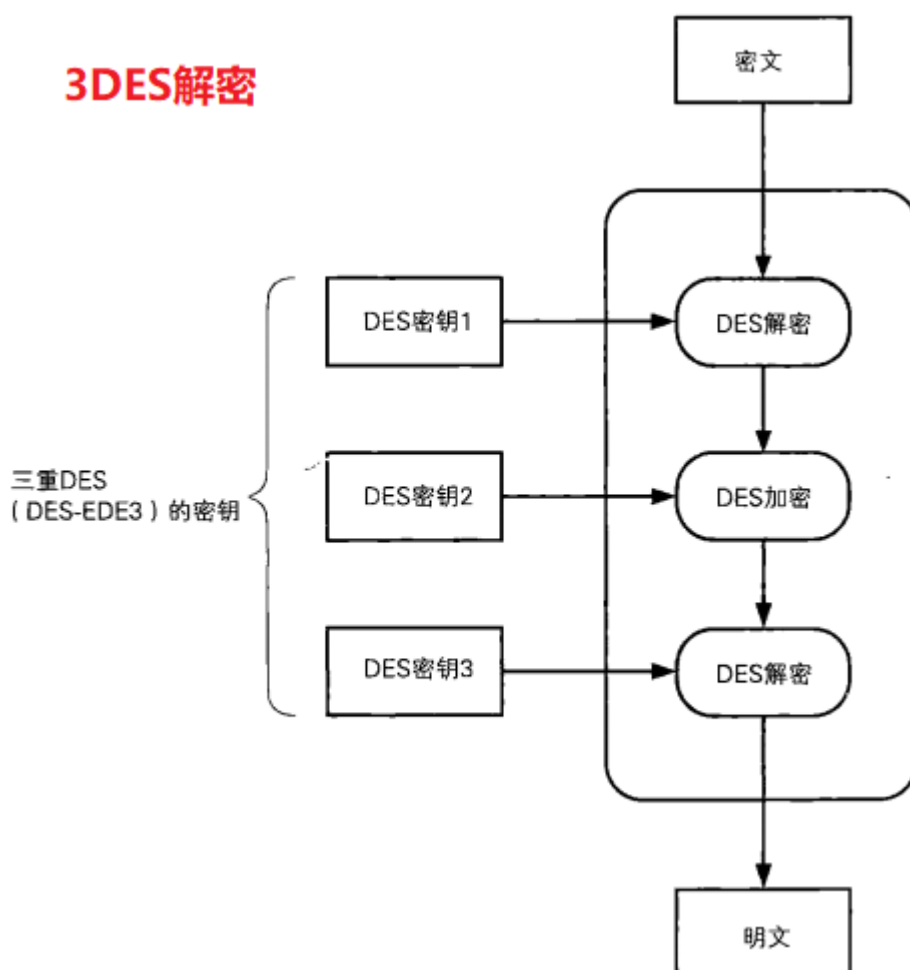


## 3DES加密

三重DES  
( DES-EDE3 ) 的密钥



## 3DES解密



明文经过三次DES处理才能变成最后的密文，由于DES密钥的长度实质上是56比特，因此三重DES的密钥长度就是 $56 \times 3 = 168$ 比特，加上用于错误检测的标志位 $8 \times 3$ ，共192bit。

从上图我们可以发现，三重DES并不是进行三次DES加密（加密-->加密-->加密），而是加密-->解密-->加密的过程。在加密算法中加入解密操作让人感觉很不可思议，实际上这个方法是IBM公司设计出来的，目的是为了让三重DES能够兼容普通的DES。

当三重DES中所有的密钥都相同时，三重DES也就等同于普通的DES了。这是因为在前两步加密-->解密之后，得到的就是最初的明文。因此，以前用DES加密的密文，就可以通过这种方式用三重DES来进行解密。也就是说，三重DES对DES具备向下兼容性。

如果密钥1和密钥3使用相同的密钥，而密钥2使用不同的密钥（也就是只使用两个DES密钥），这种三重DES就称为DES-EDE2。EDE表示的是加密（Encryption）-->解密（Decryption）-->加密（Encryption）这个流程。

密钥1、密钥2、密钥3全部使用不同的比特序列的三重DES称为DES-EDE3。

尽管三重DES目前还被银行等机构使用，但其处理速度不高，而且在安全性方面也逐渐显现出了一些问题。

### 2.3.2 Go中对3DES的操作

#### 加解密实现思路

- 加密 - CBC分组模式

1. 创建并返回一个使用3DES算法的cipher.Block接口

- 密钥长度为64bit\*3=192bit, 即 192/8 = 24字节(byte)
- 2. 对最后一个明文分组进行数据填充
  - 3DES是以64比特的明文（比特序列）为一个单位来进行加密的
  - 最后一组不够64bit, 则需要进行数据填充( 参考第三章)
- 3. 创建一个密码分组为链接模式的, 底层使用3DES加密的BlockMode接口
- 4. 加密连续的数据块

## • 解密

1. 创建并返回一个使用3DES算法的cipher.Block接口
2. 创建一个密码分组为链接模式的, 底层使用3DES解密的BlockMode接口
3. 数据块解密
4. 去掉最后一组的填充数据

## 加解密的代码实现

### 3DES加密代码

```
1 // 3DES加密
2 func TripleDESEncrypt(src, key []byte) []byte {
3     // 1. 创建并返回一个使用3DES算法的cipher.Block接口
4     block, err := des.NewTripleDESCipher(key)
5     if err != nil{
6         panic(err)
7     }
8     // 2. 对最后一组明文进行填充
9     src = PKCS5Padding(src, block.BlockSize())
10    // 3. 创建一个密码分组为链接模式, 底层使用3DES加密的BlockMode模型
11    blockMode := cipher.NewCBCEncrypter(block, key[:8])
12    // 4. 加密数据
13    dst := src
14    blockMode.CryptBlocks(dst, src)
15    return dst
16 }
```

### 3DES解密代码

```
1 // 3DES解密
2 func TripleDESDecrypt(src, key []byte) []byte {
3     // 1. 创建3DES算法的Block接口对象
4     block, err := des.NewTripleDESCipher(key)
5     if err != nil{
6         panic(err)
7     }
8     // 2. 创建密码分组为链接模式, 底层使用3DES解密的BlockMode模型
9     blockMode := cipher.NewCBCDecrypter(block, key[:8])
10    // 3. 解密
11    dst := src
12    blockMode.CryptBlocks(dst, src)
13    // 4. 去掉尾部填充的数据
14    dst = PKCS5UnPadding(dst)
```

```

15     return dst
16 }

```

## 重要的函数说明

### 1. 生成一个底层使用3DES加/解密的Block接口对象

```

1  函数对应的包: import "crypto/des"
2  func NewTripleDESCipher(key []byte) (cipher.Block, error)
3      - 参数 key: 3des对称加密使用的密码, 密码长度为(64*3)bit, 即(8*3)byte
4      - 返回值 cipher.Block: 创建出的使用DES加/解密的Block接口对象

```

### 2. 创建一个密码分组为CBC模式, 底层使用b加密的BlockMode接口对象

```

1  函数对应的包: import "crypto/cipher"
2  func NewCBCEncrypter(b Block, iv []byte) BlockMode
3      - 参数 b: 使用des.NewTripleDESCipher 函数创建出的Block接口对象
4      - 参数 iv: 事先准备好的一个长度为一个分组长度的比特序列, 每个分组为64bit, 即8byte
5      - 返回值: 得到的BlockMode接口对象

```

### 3. 使用cipher包的BlockMode接口对象对数据进行加/解密

```

1  接口对应的包: import "crypto/cipher"
2  type BlockMode interface {
3      // 返回加密字节块的大小
4      BlockSize() int
5      // 加密或解密连续的数据块, src的尺寸必须是块大小的整数倍, src和dst可指向同一内存地址
6      CryptBlocks(dst, src []byte)
7  }
8  接口中的 CryptBlocks(dst, src []byte) 方法:
9      - 参数 dst: 传出参数, 存储加密或解密运算之后的结果
10     - 参数 src: 传入参数, 需要进行加密或解密的数据切片(字符串)

```

### 4. 创建一个密码分组为CBC模式, 底层使用b解密的BlockMode接口对象

```

1  函数对应的包: import "crypto/cipher"
2  func NewCBCDecrypter(b Block, iv []byte) BlockMode
3      - 参数 b: 使用des.NewTripleDESCipher 函数创建出的Block接口对象
4      - 参数 iv: 事先准备好的一个长度为一个分组长度的比特序列, 每个分组为64bit, 即8byte,
5                该序列的值需要和NewCBCEncrypter函数的第二个参数iv值相同
6      - 返回值: 得到的BlockMode接口对象

```

## 2.4 AES

#

AES (Advanced Encryption Standard) 是取代其前任标准 (DES) 而成为新标准的一种对称密码算法。全世界的企业和密码学家提交了多个对称密码算法作为AES的候选, 最终在2000年从这些候选算法中选出了一名为 **Rijndael** 的对称密码算法, 并将其确定为了AES。

Rijndael是由比利时密码学家Joan Daemen和Vincent Rijmen设计的分组密码算法, 今后会有越来越多的密码软件支持这种算法。

Rijndael的分组长度为128比特, 密钥长度可以以32比特为单位在128比特到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)。

## 2.4.2 AES的加密和解密

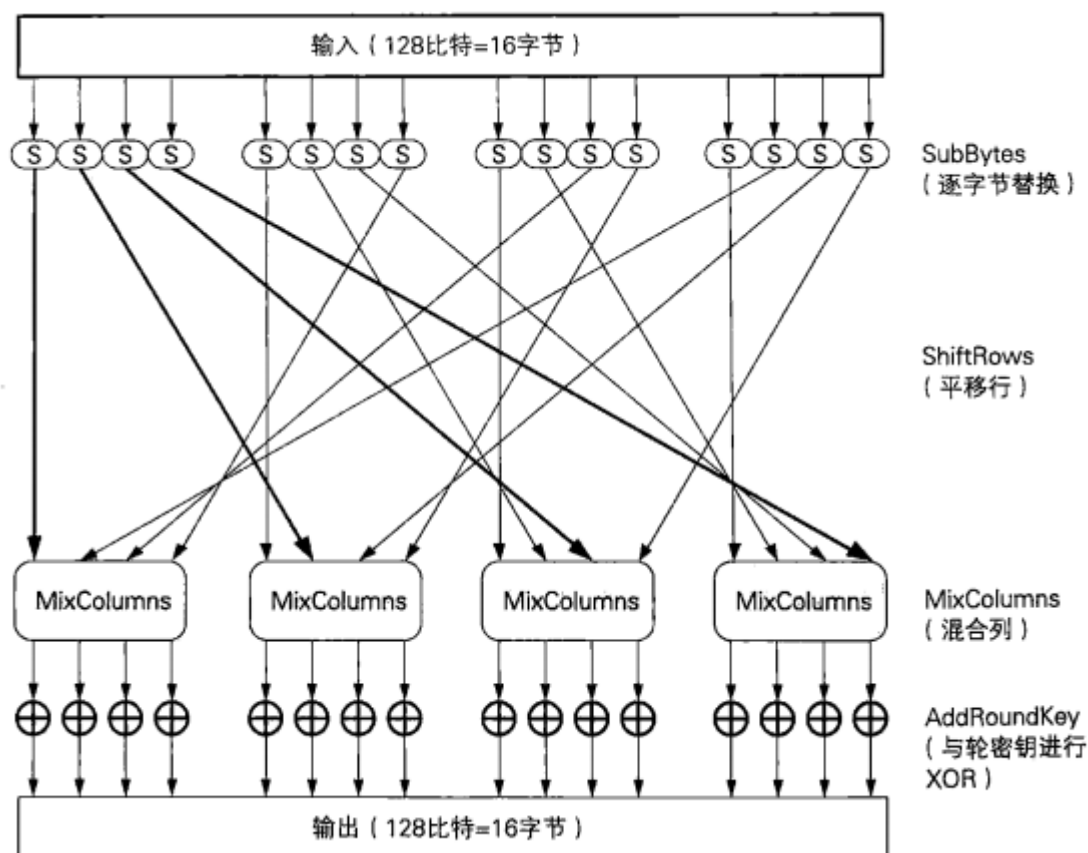
和DES一样，AES算法也是由多个轮所构成的，下图展示了每一轮的大致计算步骤。DES使用Feistel网络作为其基本结构，而AES没有使用Feistel网络，而是使用了SPN Rijndael的输入分组为128比特，也就是16字节。首先，需要逐个字节地对16字节的输入数据进行SubBytes处理。所谓SubBytes,就是以每个字节的值（0~255中的任意值）为索引，从一张拥有256个值的替换表（S-Box）中查找出对应值的处理，也就是说，将一个1字节的值替换成另一个1字节的值。

SubBytes之后需要进行ShiftRows处理，即将SubBytes的输出以字节为单位进行打乱处理。从下图的线我们可以看出，这种打乱处理是有规律的。

ShiftRows之后需要进行MixColumns处理，即对一个4字节的值进行比特运算，将其变为另外一个4字节值。

最后，需要将MixColumns的输出与轮密钥进行XOR，即进行AddRoundKey处理。到这里，AES的一轮就结束了。实际上，在AES中需要重复进行10~14轮计算。

通过上面的结构我们可以发现输入的所有比特在一轮中都会被加密。和每一轮都只加密一半输入的比特的Feistel网络相比，这种方式的优势在于加密所需要的轮数更少。此外，这种方式还有一个优势，即SubBytes, ShiftRows和MixColumns可以分别按字节、行和列为单位进行并行计算。



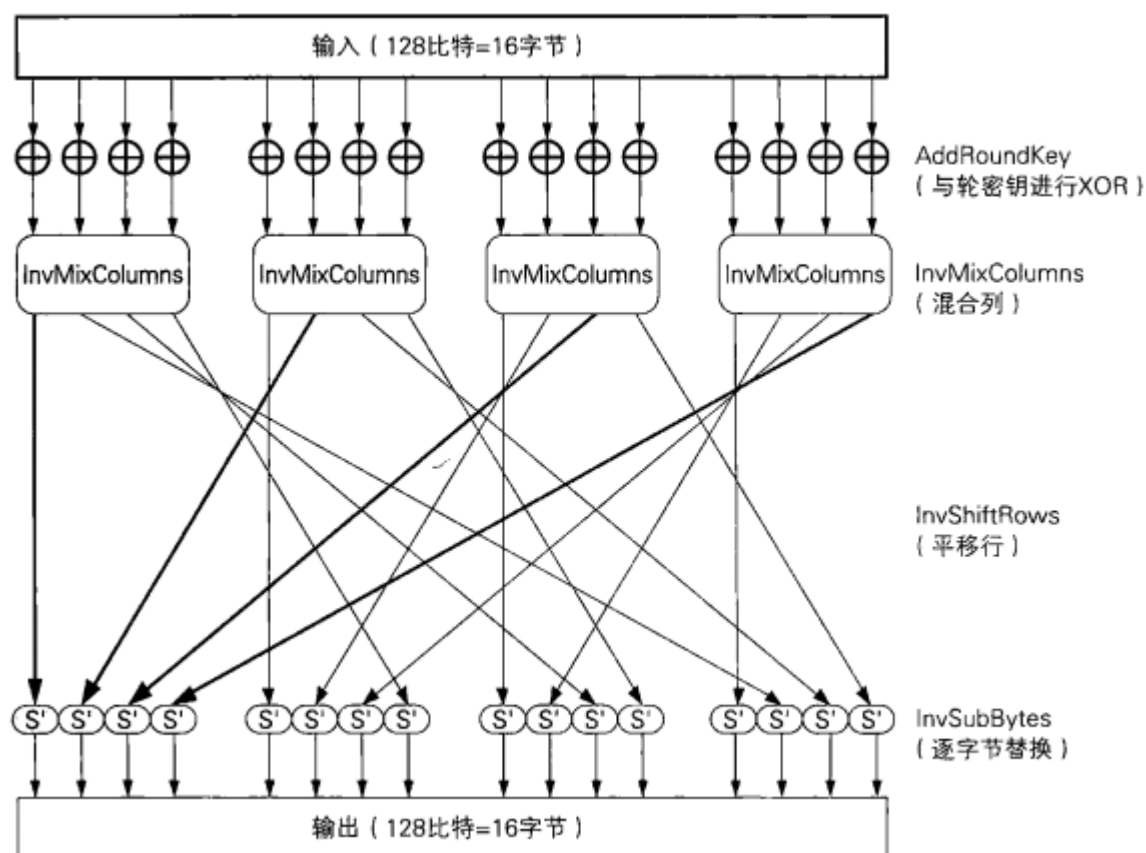
SubBytes -- 字节代换

ShiftRows -- 行移位代换

MixColumns -- 列混淆

AddRoundKey -- 轮密钥加

下图展示了AES中一轮的解密过程。从图中我们可以看出，SubBytes、ShiftRows、MixColumns分别存在反向运算 InvSubBytes、InvShiftRows、InvMixColumns，这是因为AES不像Feistel网络一样能够用同一种结构实现加密和解密。



InvSubBytes -- 逆字节替代

InvShiftRows -- 逆行移位

InvMixColumns -- 逆列混淆

## 2.4.2 Go中对AES的使用

### 加解密实现思路

#### • 加密 - CBC分组模式

1. 创建并返回一个使用AES算法的cipher.Block接口
  - 密钥长度为128bit, 即  $128/8 = 16$ 字节(byte)
2. 对最后一个明文分组进行数据填充
  - AES是以128比特的明文（比特序列）为一个单位来进行加密的
  - 最后一组不够128bit, 则需要进行数据填充( 参考第三章)
3. 创建一个密码分组为链接模式的, 底层使用AES加密的BlockMode接口
4. 加密连续的数据块

#### • 解密

1. 创建并返回一个使用AES算法的cipher.Block接口
2. 创建一个密码分组为链接模式的, 底层使用AES解密的BlockMode接口
3. 数据块解密
4. 去掉最后一组的填充数据

## 加解密的代码实现

### AES加密代码

```
1 // AES加密
2 func AESEncrypt(src, key []byte) []byte{
3     // 1. 创建一个使用AES加密的块对象
4     block, err := aes.NewCipher(key)
5     if err != nil{
6         panic(err)
7     }
8     // 2. 最后一个分组进行数据填充
9     src = PKCS5Padding(src, block.BlockSize())
10    // 3. 创建一个分组为链接模式, 底层使用AES加密的块模型对象
11    blockMode := cipher.NewCBCEncrypter(block, key[:block.BlockSize()])
12    // 4. 加密
13    dst := src
14    blockMode.CryptBlocks(dst, src)
15    return dst
16 }
```

### AES解密

```
1 // AES解密
2 func AESDecrypt(src, key []byte) []byte{
3     // 1. 创建一个使用AES解密的块对象
4     block, err := aes.NewCipher(key)
5     if err != nil{
6         panic(err)
7     }
8     // 2. 创建分组为链接模式, 底层使用AES的解密模型对象
9     blockMode := cipher.NewCBCDecrypter(block, key[:block.BlockSize()])
10    // 3. 解密
11    dst := src
12    blockMode.CryptBlocks(dst, src)
13    // 4. 去掉尾部填充的字
14    dst = PKCS5UnPadding(dst)
15    return dst
16 }
```

### 重要的函数说明

1. 生成一个底层使用AES加/解密的Block接口对象

```
1 函数对应的包: import "crypto/aes"
2 func NewCipher(key []byte) (cipher.Block, error)
3     - 参数 key: aes对称加密使用的密码, 密码长度为128bit, 即16byte
4     - 返回值 cipher.Block: 创建出的使用AES加/解密的Block接口对象
```

## 2. 创建一个密码分组为CBC模式, 底层使用b加密的BlockMode接口对象

```
1  函数对应的包: import "crypto/cipher"
2  func NewCBCEncrypter(b Block, iv []byte) BlockMode
3      - 参数 b: 使用aes.NewCipher函数创建出的Block接口对象
4      - 参数 iv: 事先准备好的一个长度为一个分组长度的比特序列, 每个分组为64bit, 即8byte
5      - 返回值: 得到的BlockMode接口对象
```

## 3. 使用cipher包的BlockMode接口对象对数据进行加/解密

```
1  接口对应的包: import "crypto/cipher"
2  type BlockMode interface {
3      // 返回加密字节块的大小
4      BlockSize() int
5      // 加密或解密连续的数据块, src的尺寸必须是块大小的整数倍, src和dst可指向同一内存地址
6      CryptBlocks(dst, src []byte)
7  }
8  接口中的 CryptBlocks(dst, src []byte) 方法:
9      - 参数 dst: 传出参数, 存储加密或解密运算之后的结果
10     - 参数 src: 传入参数, 需要进行加密或解密的数据切片(字符串)
```

## 4. 创建一个密码分组为CBC模式, 底层使用b解密的BlockMode接口对象

```
1  函数对应的包: import "crypto/cipher"
2  func NewCBCDecrypter(b Block, iv []byte) BlockMode
3      - 参数 b: 使用des.NewCipher函数创建出的Block接口对象
4      - 参数 iv: 事先准备好的一个长度为一个分组长度的比特序列, 每个分组为128bit, 即16byte,
5        该序列的值需要和NewCBCEncrypter函数的第二个参数iv值相同
6      - 返回值: 得到的BlockMode接口对象
```

## 2.5 应选择哪种对称加密

#

前面我们介绍了DES、三重DES和AES等对称密码, 那么我们到底应该使用哪一种对称密码算法呢?

1. 今后最好不要将DES用于新的用途, 因为随着计算机技术的进步, 现在用暴力破解法已经能够在现实的时间内完成对DES的破译。但是, 在某些情况下也需要保持与旧版本软件的兼容性。
2. 出于兼容性的因素三重DES在今后还会使用一段时间, 但会逐渐被AES所取代。
3. 今后大家应该使用的算法是AES (Rijndael), 因为它安全、快速, 而且能够在各种平台上工作。此外, 由于全世界的密码学家都在对AES进行不断的验证, 因此即便万一发现它有什么缺陷, 也会立刻告知全世界并修复这些缺陷。

一般来说, 我们不应该使用任何自制的密码算法, 而是应该使用AES。因为AES在其选定过程中, 经过了全世界密码学家所进行的高品质的验证工作, 而对于自制的密码算法则很难进行这样的验证。

## 本章小结

本章中我们介绍了对称密码, 以及DES、三重DES、AES和其他一些密码算法。

使用一种密钥空间巨大, 且在算法上没有弱点的对称密码, 就可以通过密文来确保明文的机密性。巨大的密钥空间能够抵御暴力破解, 算法上没有弱点可以抵御其他类型的攻击。



然而，用对称密码进行通信时，还会出现密钥的配送问题，即如何将密钥安全地发送给接收者。为了解决密钥配送问题，我们需要非对称加密技术。非对称加密，我们将在第四章进行讲解。

本章所介绍的几乎所有的密码算法，都只能将一个固定长度的分组进行加密当需要加密的明文长度超过分组长度时，就需要对密码算法进行迭代下一章我们将探讨对分组密码进行迭代的方法。

## 3. 分组密码的模式

1 "分组密码的模式 -- 分组密码是如何迭代的"

本章中我们将探讨一下分组密码的模式

我们在上一章中介绍的DES和AES都属于分组密码，它们只能加密固定长度的明文。如果需要加密任意长度的明文，就需要对分组密码进行迭代，而分组密码的迭代方法就称为分组密码的“模式”。

分组密码有很多种模式，如果模式的选择不当，就无法保证机密性。例如，如果使用ECB模式，明文中的一些规律就可以通过密文被识别出来。

分组密码的主要模式（ECB、CBC、CFB、OFB、CTR），最后再来考察一下到底应该使用哪一种模式。

### 3.1 分组密码

#

**分组密码（blockcipher）**是每次只能处理特定长度的一块数据的一类密码算法，这里的一块“就称为分组（block）”。此外，一个分组的比特数就称为分组长度（blocklength）。

例如，**DES和三重DES的分组长度都是64比特**。这些密码算法一次只能加密64比特的明文，并生成64比特的密文。

**AES的分组长度可以从128比特、192比特和256比特中进行选择。当选择128比特的分组长度时，AES一次可加密128比特的明文，并生成128比特的密文。**

### 3.2 模式

#

**分组密码算法只能加密固定长度的分组，但是我们需要加密的明文长度可能会超过分组密码的分组长度，这时就需要对分组密码算法进行迭代，以便将一段很长的明文全部加密。而迭代的方法就称为分组密码的模式（mode）。**

话说到这里，很多读者可能会说：“如果明文很长的话，将明文分割成若干个分组再逐个加密不就好了吗？”事实上可没有那么简单。将明文分割成多个分组并逐个加密的方法称为ECB模式，这种模式具有很大的弱点（稍后讲解）。对密码不是很了解的程序员在编写加密软件时经常会使用ECB模式，但这样做会在不经意间产生安全漏洞，**因此大家要记住千万不能使用ECB模式。**

模式有很多种类，分组密码的主要模式有以下5种：

- **ECB模式**：Electronic Code Book mode（电子密码本模式）
- **CBC模式**：Cipher Block Chaining mode（密码分组链接模式）
- **CFB模式**：Cipher FeedBack mode（密文反馈模式）
- **OFB模式**：Output FeedBack mode（输出反馈模式）

- CTR模式: CounTeR mode (计数器模式)

## 明文分组和密文分组

在介绍模式之前，我们先来学习两个术语。

**明文分组:** 是指分组密码算法中作为加密对象的明文。明文分组的长度与分组密码算法的分组长度是相等的。

**密文分组:** 是指使用分组密码算法将明文分组加密之后所生成的密文。



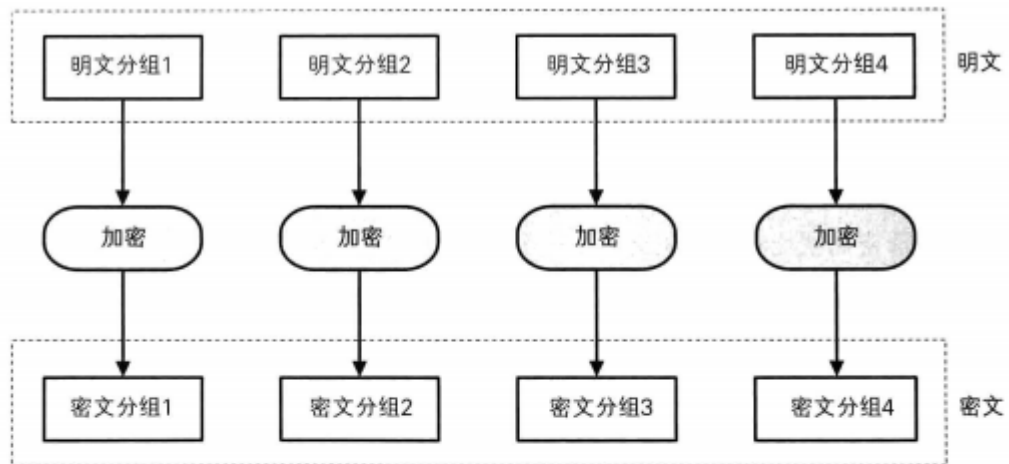
为了避免图示变得复杂，以后我们将“用分组密码算法加密”简写为“加密”，并省略对密钥的描述。

## 3.3 ECB 模式

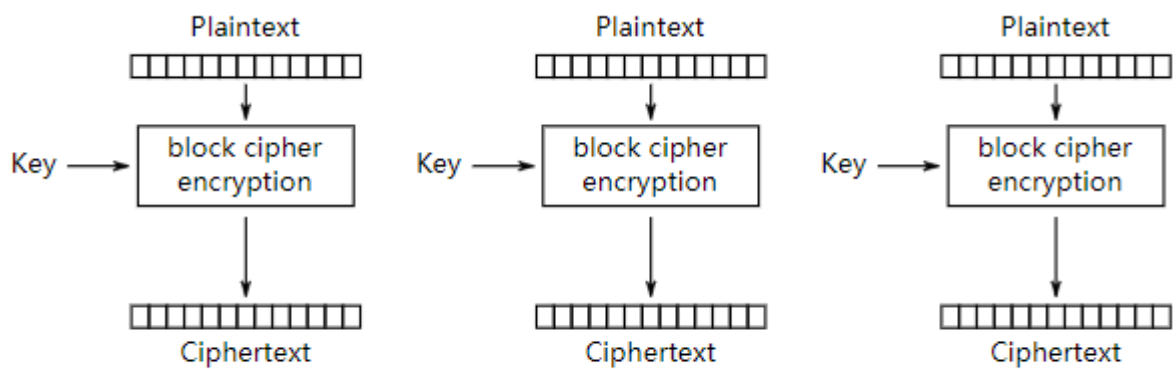
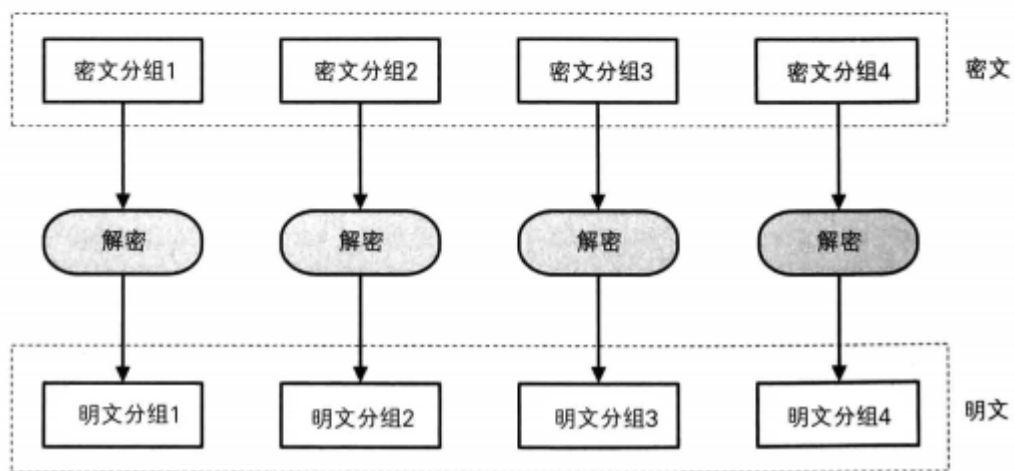
#

ECB(Electronic Code Book, 电子密码本)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加密，所以可以进行并行计算，但是这种方法一旦有一个块被破解，使用相同的方法可以解密所有的明文数据，安全性比较差。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倍数。

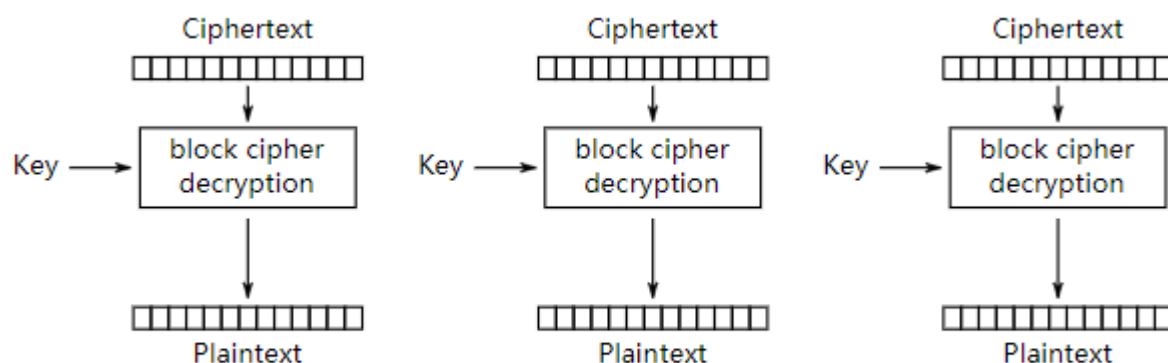
ECB模式的加密



ECB模式的解密



Electronic Codebook (ECB) mode encryption



## Electronic Codebook (ECB) mode decryption

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，我们可以将其理解为一个巨大的“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时，需要用一特定的数据进行填充（padding），让值一个分组长度等于分组长度。**

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，就可以知道明文存在怎样的重复组合，并可以以此为线索来破译密码，因此ECB模式是存在一定风险的。

### 3.3 CBC模式

#

#### XOR

为了让大家理解比特序列运算的概念，我们来介绍一下XOR运算。XOR的全称是exclusive or，在中文里叫作异或。尽管名字看起来很复杂，但这种运算本身一点都不难。

1个比特（bit）的位运算规则如下：

0 XOR 0 = 0	（0 与 0 的 XOR 结果为 0）
0 XOR 1 = 1	（0 与 1 的 XOR 结果为 1）
1 XOR 0 = 1	（1 与 0 的 XOR 结果为 1）
1 XOR 1 = 0	（1 与 1 的 XOR 结果为 0）

如果将0理解为偶数，1理解为奇数，就可以将XOR和一般的加法运算等同起来。

偶数（0）+ 偶数（0）= 偶数（0）  
 偶数（0）+ 奇数（1）= 奇数（1）  
 奇数（1）+ 偶数（0）= 奇数（1）  
 奇数（1）+ 奇数（1）= 偶数（0）

由于XOR和加法运算很相似，因此一般用+和0组合而成的符号⊕来表示XOR。

$0 \oplus 0 = 0$	(0 与 0 的 XOR 结果为 0)
$0 \oplus 1 = 1$	(0 与 1 的 XOR 结果为 1)
$1 \oplus 0 = 1$	(1 与 0 的 XOR 结果为 1)
$1 \oplus 1 = 0$	(1 与 1 的 XOR 结果为 0)

为了更加直观地理解XOR，大家可以想象一下黑白棋（奥赛罗棋）中的棋子。

- 将一个棋子保持原状（不翻转）看做0
- 将一个棋子翻转到另一面看做1

那么XOR运算就相当于将黑白棋的一个棋子进行翻转的操作。

不翻转(0) $\oplus$ 不翻转(0) = 不翻转(0)	——没有翻转
不翻转(0) $\oplus$ 翻转(1) = 翻转(1)	——翻转了一次
翻转(1) $\oplus$ 不翻转(0) = 翻转(1)	——翻转了一次
翻转(1) $\oplus$ 翻转(1) = 不翻转(0)	——翻转了两次，就等于没有翻转

通过上述场景，大家应该能够理解这样一个规律，即两个相同的数进行XOR运算的结果一定为0，因为棋子翻转两次和一次都没有翻转的结果是一样的。

上面我们介绍了1个比特之间的XOR运算，而如果是长比特序列之间的运算，则只要对其中每个相对应的比特进行XOR运算就可以了。假设我们将01001100这个比特序列称为A，将10101010这个比特序列称为B，那么A与B的XOR运算就可以像下面这样逐一对各个比特进行计算。和加法运算不同的是，XOR中不需要进位。

$$\begin{array}{r}
 0\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ \cdots\ A \\
 \oplus\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ \cdots\ B \\
 \hline
 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ \cdots\ A \oplus B
 \end{array}$$

由于两个相同的数进行XOR运算的结果一定为0，因此如果将 $A \oplus B$ 的结果再与B进行XOR运算，则结果会变回A。也就是说，两个公式中的B会相互抵消。

$$\begin{array}{r}
 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ \cdots\ A \oplus B \\
 \oplus\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ \cdots\ B \\
 \hline
 0\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ \cdots\ A\quad\quad\quad (\text{变回了 } A)
 \end{array}$$

可能大家已经发现了，上面的计算和加密、解密的步骤非常相似。

- 将明文A用密钥B进行加密，得到密文 $A \oplus B$
- 将密文 $A \oplus B$ 用密钥B进行解密，得到明文A

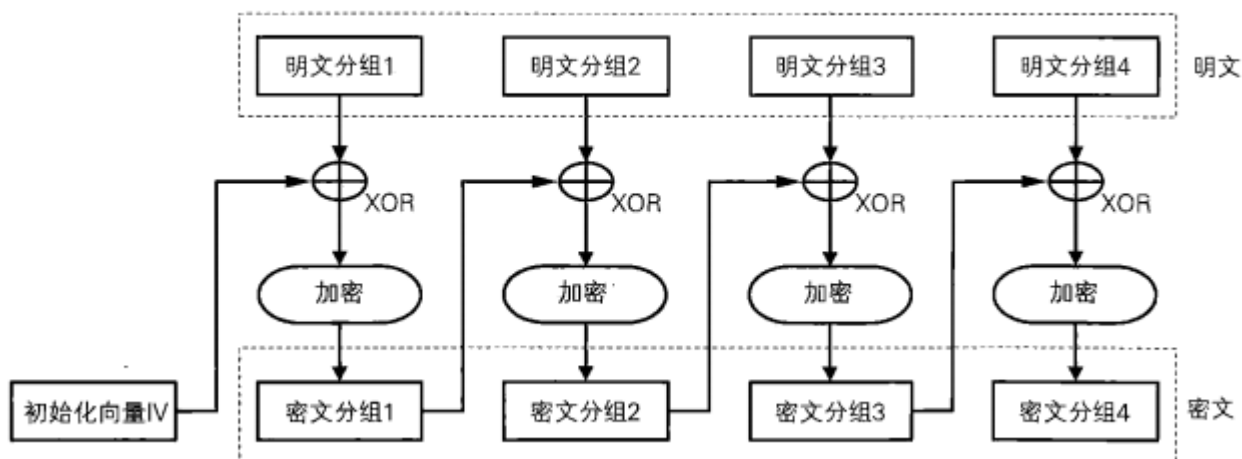
实际上，只要选择一个合适的B，仅仅使用XOR就可以实现一个高强度的密码。

对同一个比特序列进行两次XOR之后就会回到最初的状态。

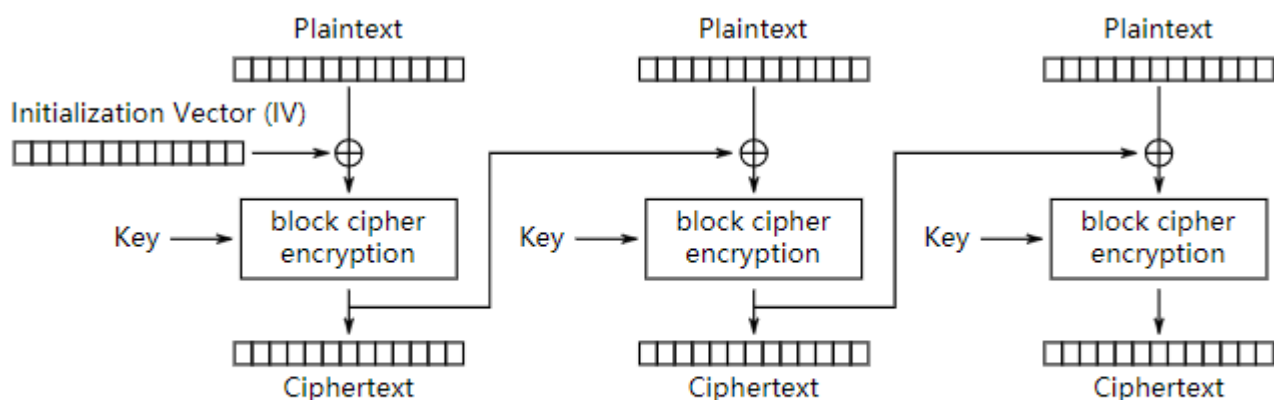
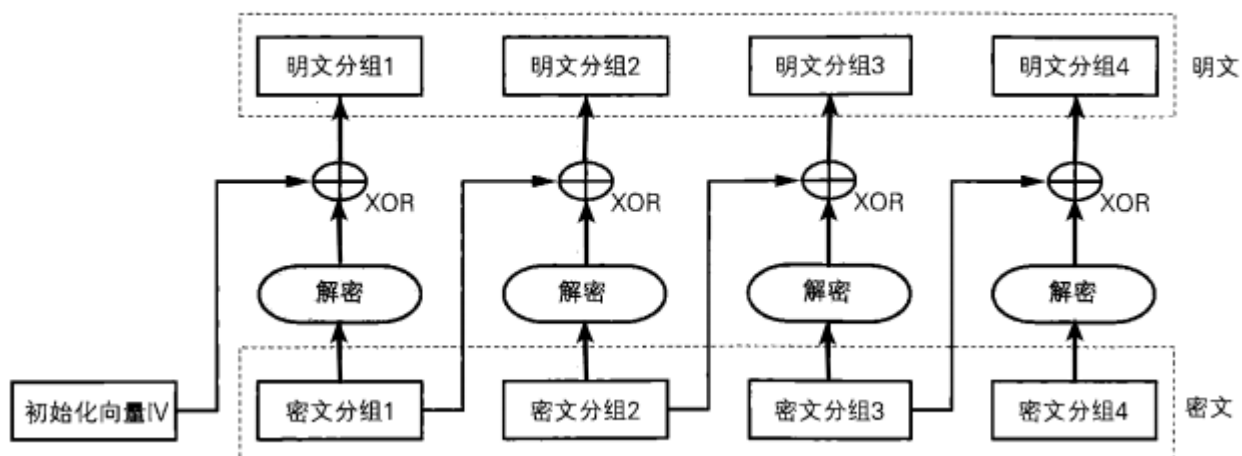
## CBC模式

CBC(Cipher Block Chaining, 密码块链)模式中每一个分组要先和前一个分组加密后的数据进行XOR异或操作, 然后再进行加密。这样每个密文块依赖该块之前的所有明文块, 为了保持每条消息都具有唯一性, 第一个数据块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, 它主要缺点是加密是连续的, 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倍数。

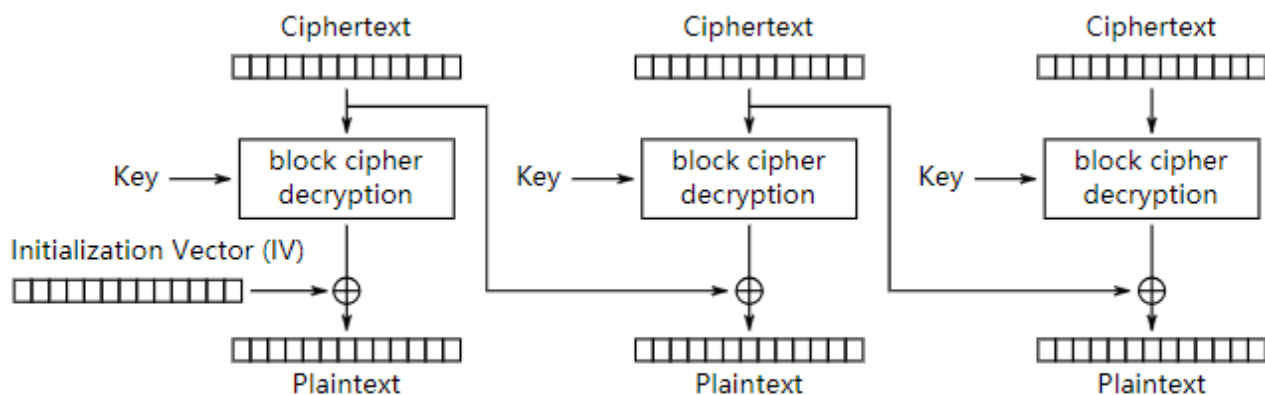
CBC模式的加密



CBC模式的解密

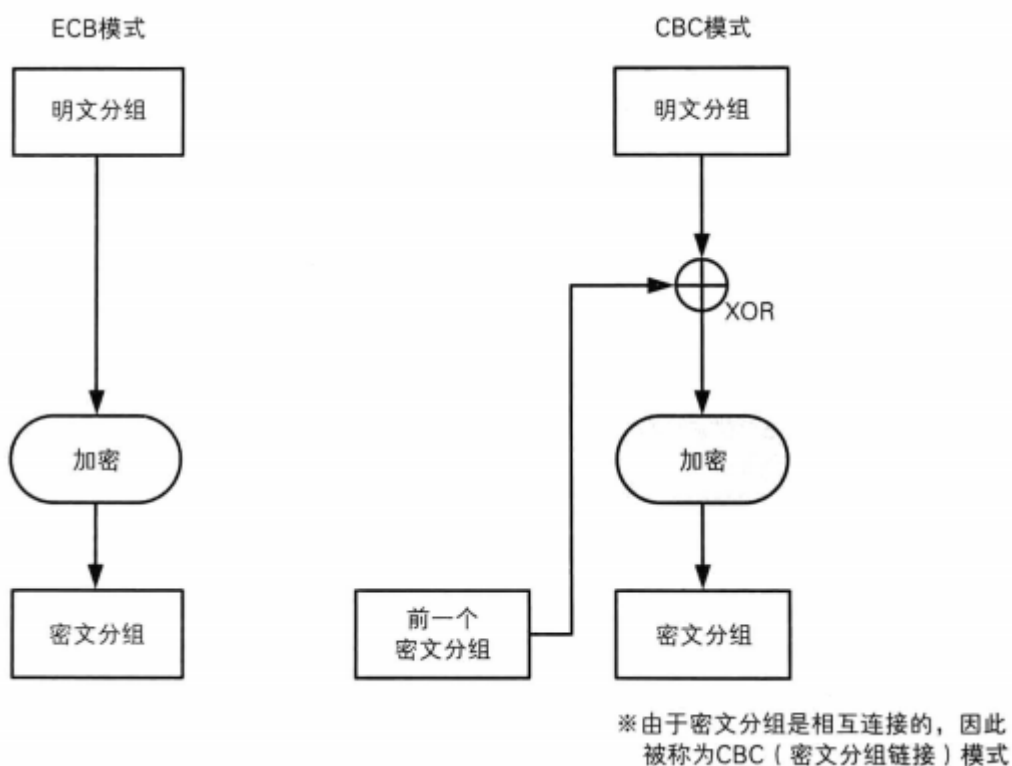


Cipher Block Chaining (CBC) mode encryption



## Cipher Block Chaining (CBC) mode decryption

如果将一个分组的加密过程分离出来，我们就可以很容易地比较出ECB模式和CBC模式的区别。ECB模式只进行了加密，而CBC模式则在加密之前进行了一次XOR。



## 初始化向量

当加密第一个明文分组时，由于不存在“前一个密文分组”，因此**需要事先准备一个长度为一个分组的比特序列来代替“前一个密文分组”**，这个比特序列称为初始化向量（initialization vector）

通常缩写为 IV 一般来说，每次加密时都会随机产生一个不同的比特序列来作为初始化向量。

明文分组在加密之前一定会与“前一个密文分组”进行 XOR 运算，因此即便明文分组1和2的值是相等的，密文分组1和2的值也不一定是相等的。这样一来，ECB模式的缺陷在CBC模式中就不存在了。

## 3.4 CFB 模式

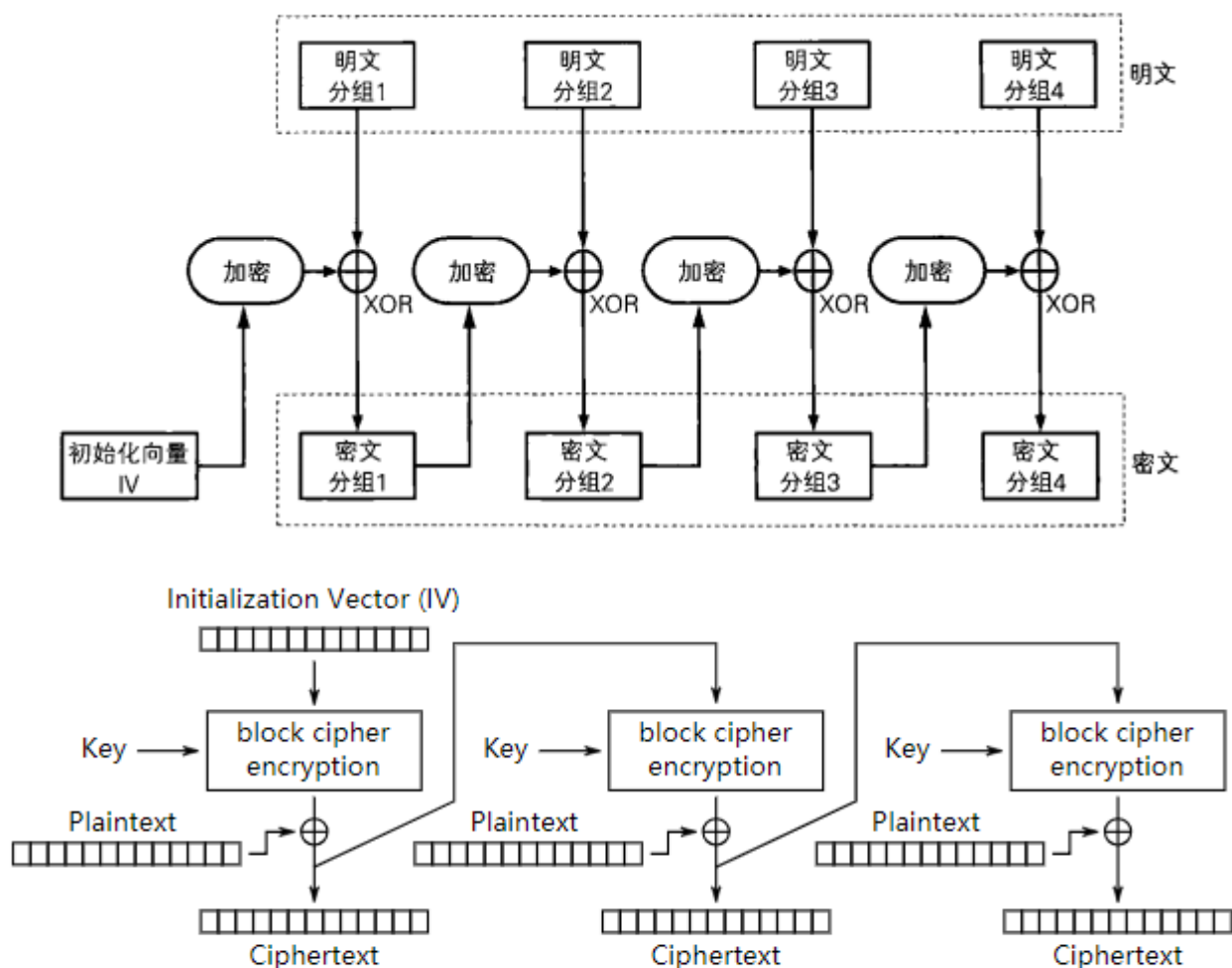
#

CFB模式的全称是Cipher FeedBack模式（密文反馈模式）。在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。**

所谓反馈，这里指的就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模式的解密和CBC模式的加密在流程上其实是非常相似的。

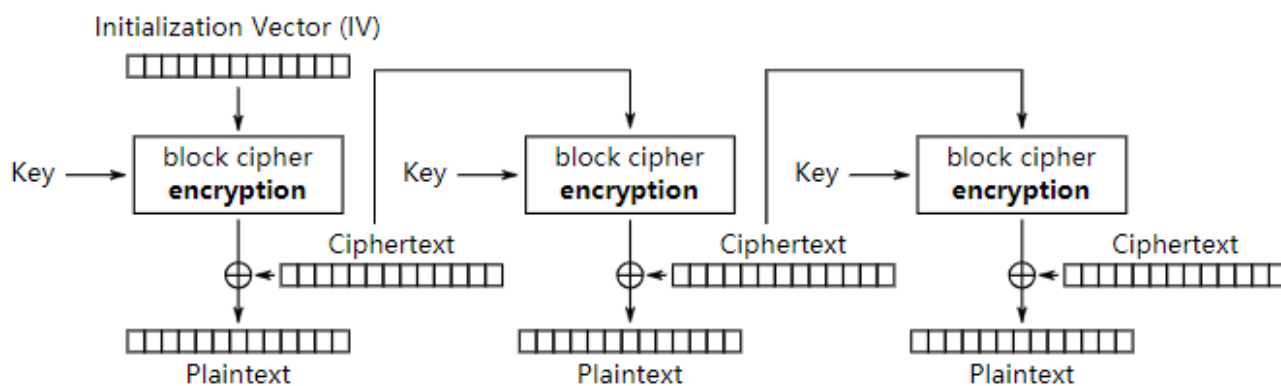
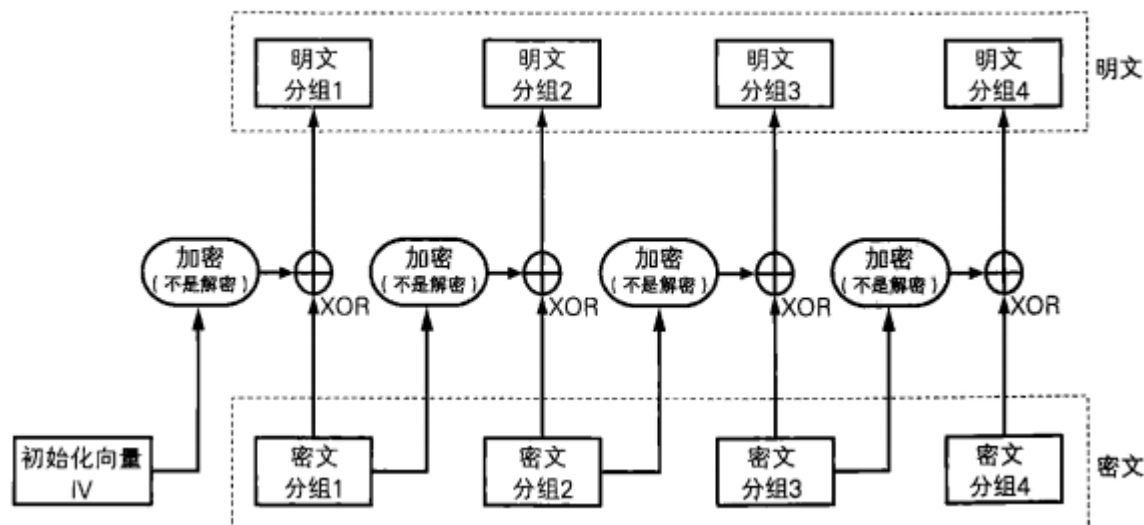
CFB模式的加密



Cipher Feedback (CFB) mode encryption



## CFB模式的解密



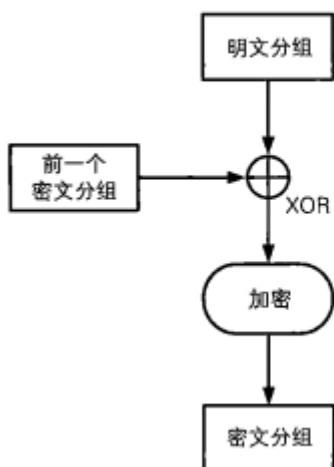
## Cipher Feedback (CFB) mode decryption

在ECB模式和CBC模式中，明文分组都是通过密码算法进行加密的，然而，在CFB模式中，明文分组并没有通过密码算法来直接进行加密。

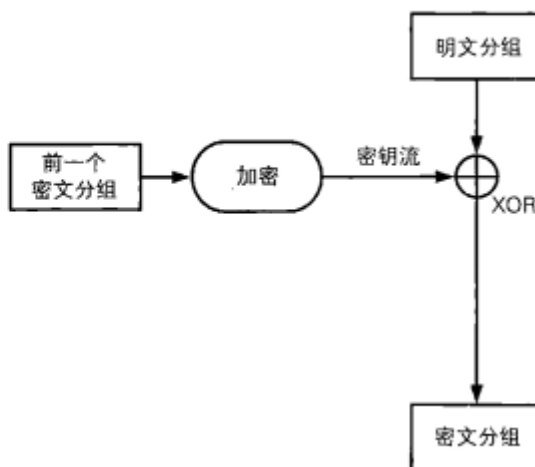
从上图可以看出，明文分组和密文分组之间并没有经过“加密”这一步骤。在CFB模式中，明文分和密文分组之间只有一个XOR。

我们将CBC模式与CFB模式对比一下，就可以看出其中的差异了（如下图）。在CBC模式中，明文分组和密文分组之间有XOR和密码算法两个步骤，而在CFB模式中，明文分组和密文分组之间则只有XOR。

CBC模式



CFB模式



## 初始化向量

在生成第一个密文分组时，由于不存在前一个输出的数据，因此需要使用初始化向量（IV）来代替，这一点和CBC模式是相同的。一般来说，我们需要在每次加密时生成一个不同的随机比特序列用作初始化向量。

## CFB模式与流密码

CFB模式是通过将“明文分组”与“密码算法的输出”进行XOR运算来生成“密文分组”的。

在CFB模式中，密码算法的输出相当于一个随机比特序列。由于密码算法的输出是通过计算得到的，并不是真正的随机数，因此CFB模式不可能具备理论上不可破译的性质。

CFB模式中由密算法所生成的比特序列称为密钥流（key stream）。在CFB模式中，密码算法就相当于用来生成密钥流的伪随机数生成器，而初始化向量相当于伪随机数生成器的“种子”。

在CFB模式中，明文数据可以被逐比特加密，因此我们可以将CFB模式看做是一种使用分组密码来实现流密码的方式。

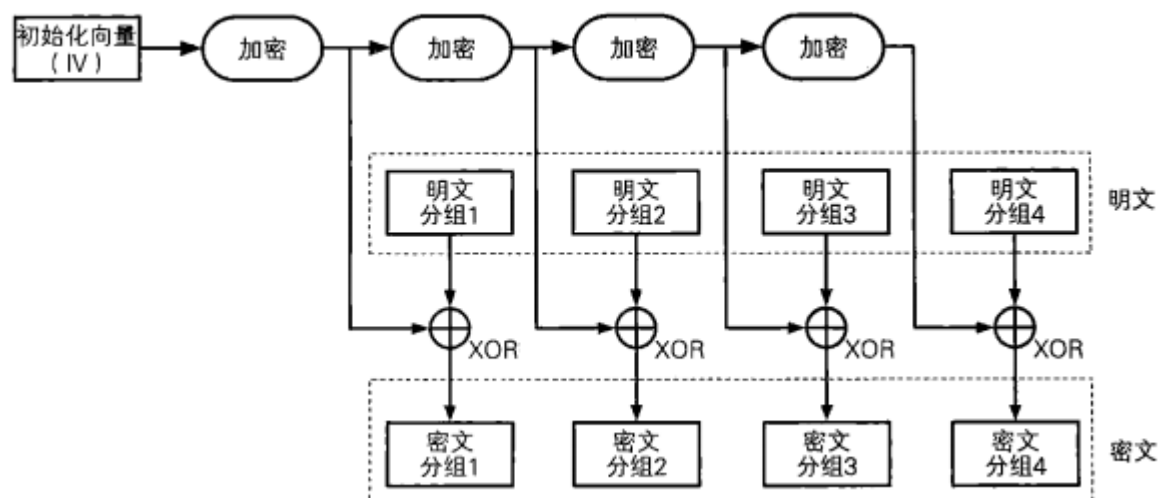
## 3.5 OFB 模式

#

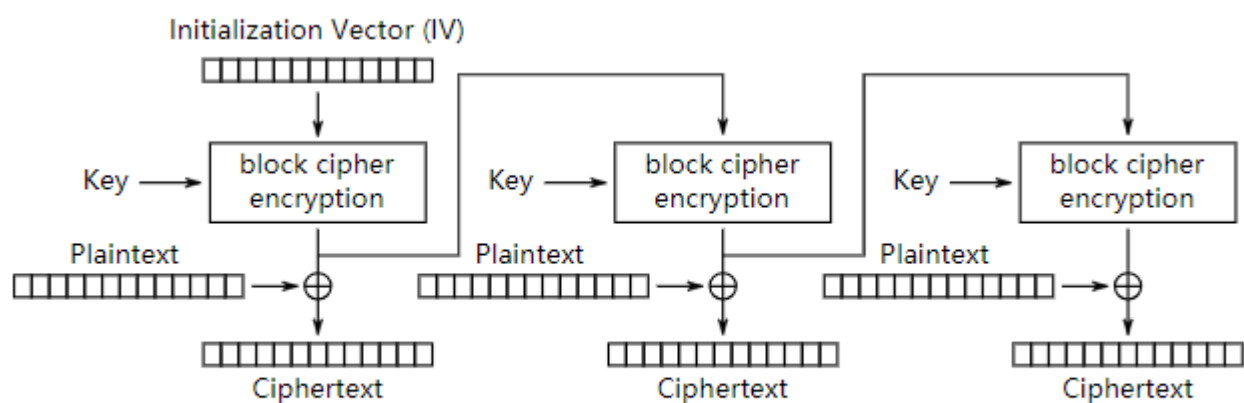
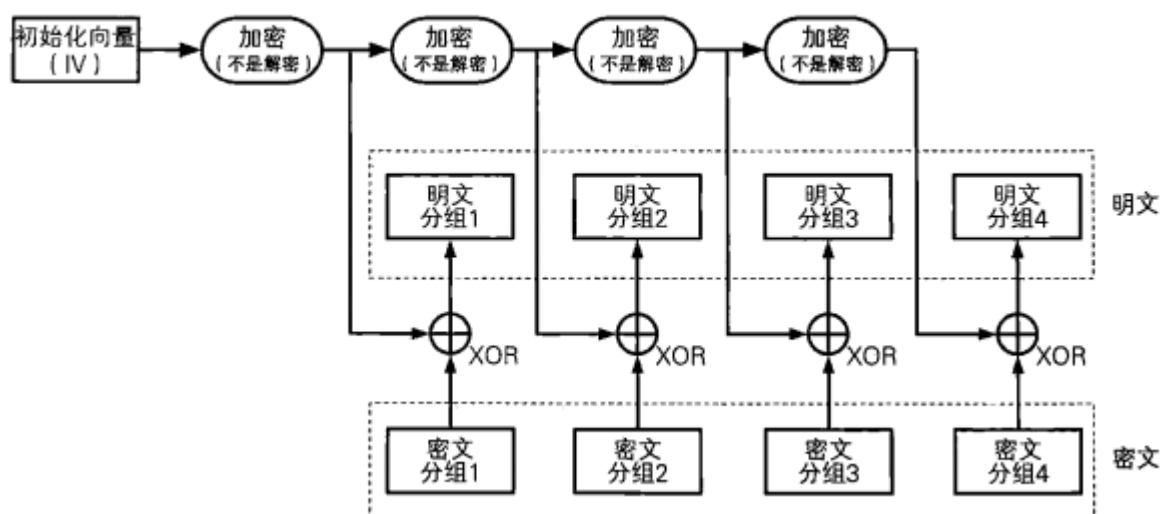
OFB式的全称是Output-Feedback模式（输出反馈模式）。在OFB模式中，密码算法的输出会反馈到密码算法的输入中，即上一个分组密码算法的输出是当前分组密码算法的输入（下图）。

OFB模式并不是通过密码算法对明文直接进行加密的，而是通过将“明文分组”和“密码算法的输出”进行XOR来产生“密文分组”的，在这一点上OFB模式和CFB模式非常相似。

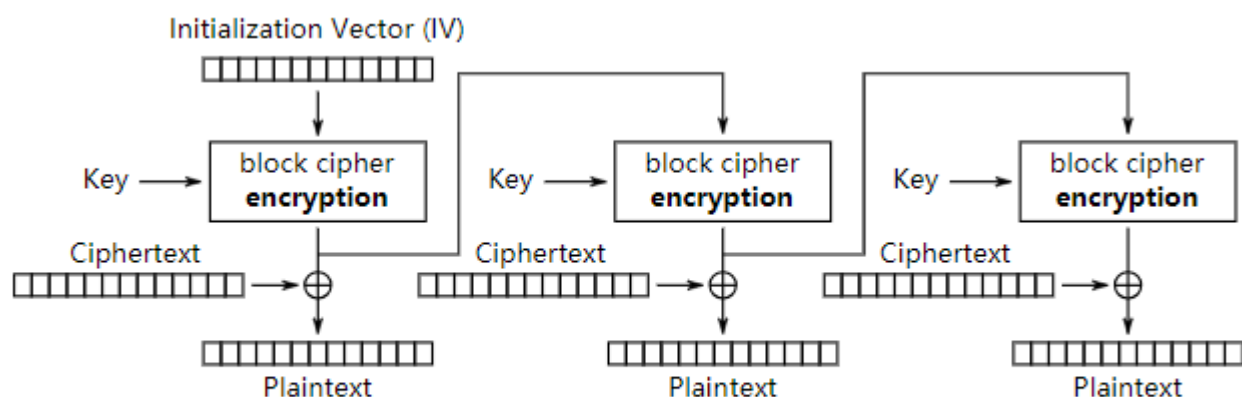
OFB模式的加密



OFB模式的解密



Output Feedback (OFB) mode encryption



Output Feedback (OFB) mode decryption

## 初始化向量

和CBC模式、CFB模式一样，OFB模式中也需要使用初始化向量（IV）。一般来说，我们需要在每次加密时生成一个不同的随机比特序列用作初始化向量。

## CFB模式和OFB模式对比

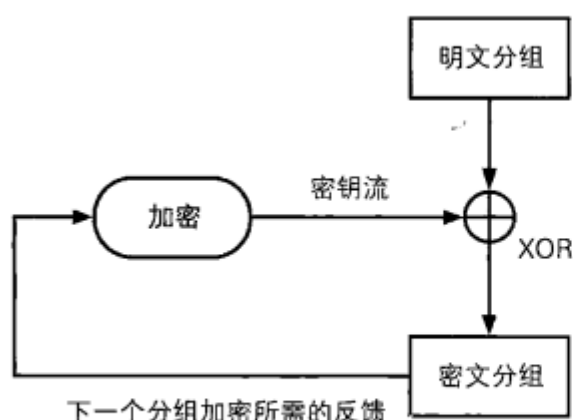
OFB模式和CFB模式的区别仅仅在于密码算法的输入。

CFB式中，密码算法的输入是前一个密文分组，也就是将密文分组反馈到密算法中，因此就有了“密文反馈模式”这个名字。

相对地，OFB模式中，密码算法的输入则是密码算法的前一个输出，也就是将输出反馈给密码算法，因此就有了“输出反馈模式”这个名字。

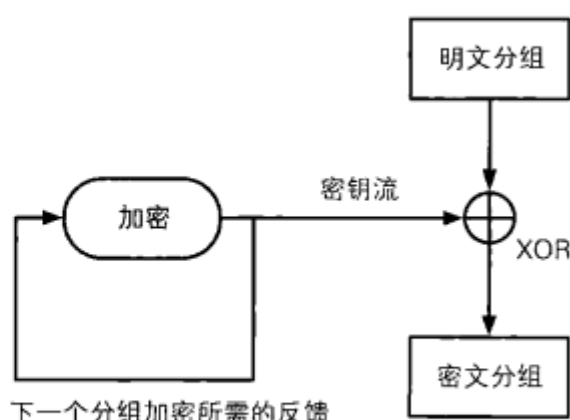
如果将一个分组抽出来对CFB模式和OFB模式进行一个对比，就可以很容易看出它们之间的差异（下图）。

CFB模式



※密文分组被反馈到加密算法的输入，因此被称为CFB（密文反馈）

OFB模式



※加密算法的输出被反馈到加密算法的输入，因此被称为OFB（输出反馈）

由于CFB模式中是对密文分组进行反馈的，因此必须从第一个明文分组开始按顺序进行加密，也就是说无法跳过明文分组1而先对明文分组2进行加密。

相对地，在OFB模式中，XOR所需要的比特序列（密钥流）可以事先通过密码算法生成，和明文分组无关。只要提前准备好所需的密钥流，则在实际从明文生成密文的过程中，就完全不需要动用密码算法了。只要将明文与密钥流进行XOR就可以了。和AES等密码算法相比，XOR运算的速度是非常快的。这就意味着只要提前准备好密钥流就可以快速完成加密。换个角度来看，生成密钥流的操作和进行XOR运算的操作是可以并行的。

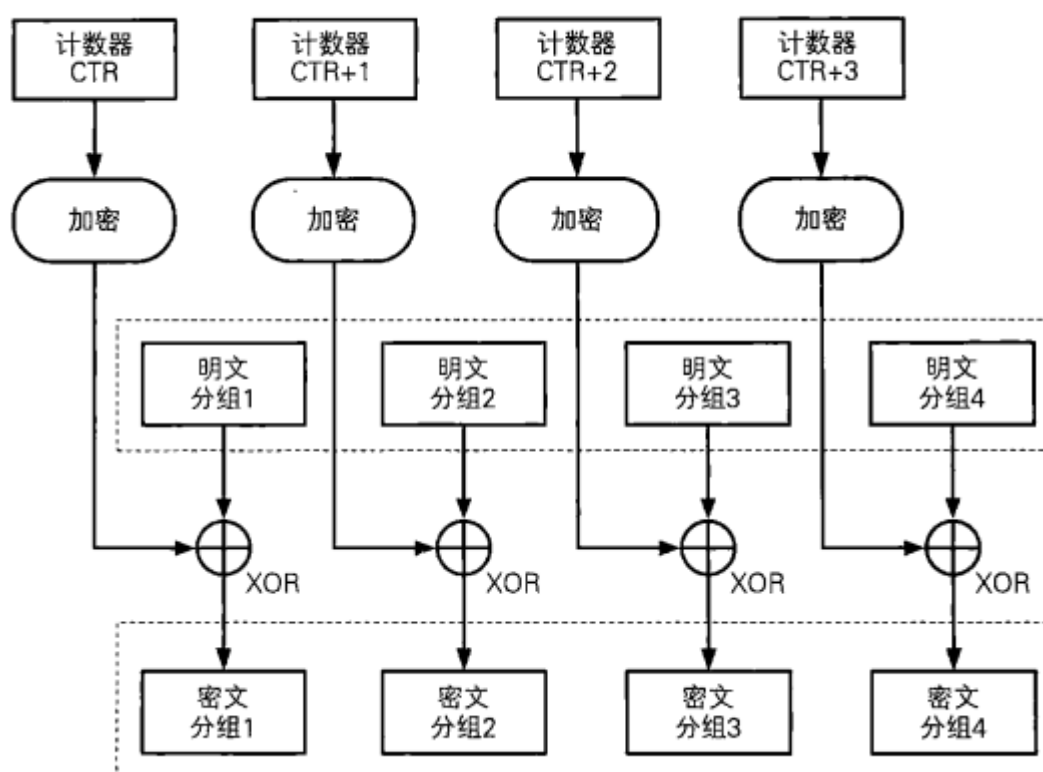
### 3.6 CTR 模式

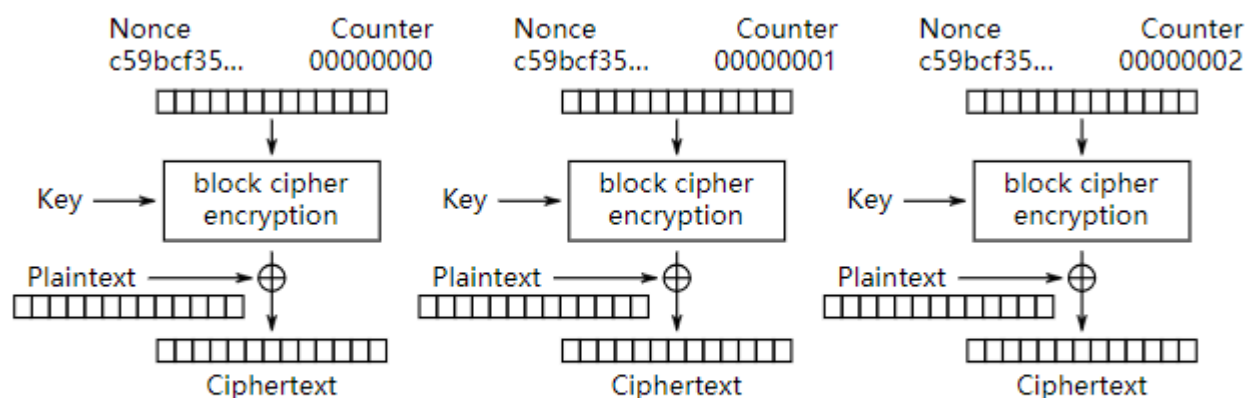
#

CTR模式的全称是CounTeR模式（计数器模式）。CTR模式是一种通过将逐次累加的计数器进行加密来生成密钥流的流密码（下图）。

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，最终的密文分组是通过将计数器加密得到的比特序列，与明文分组进行XOR而得到的。

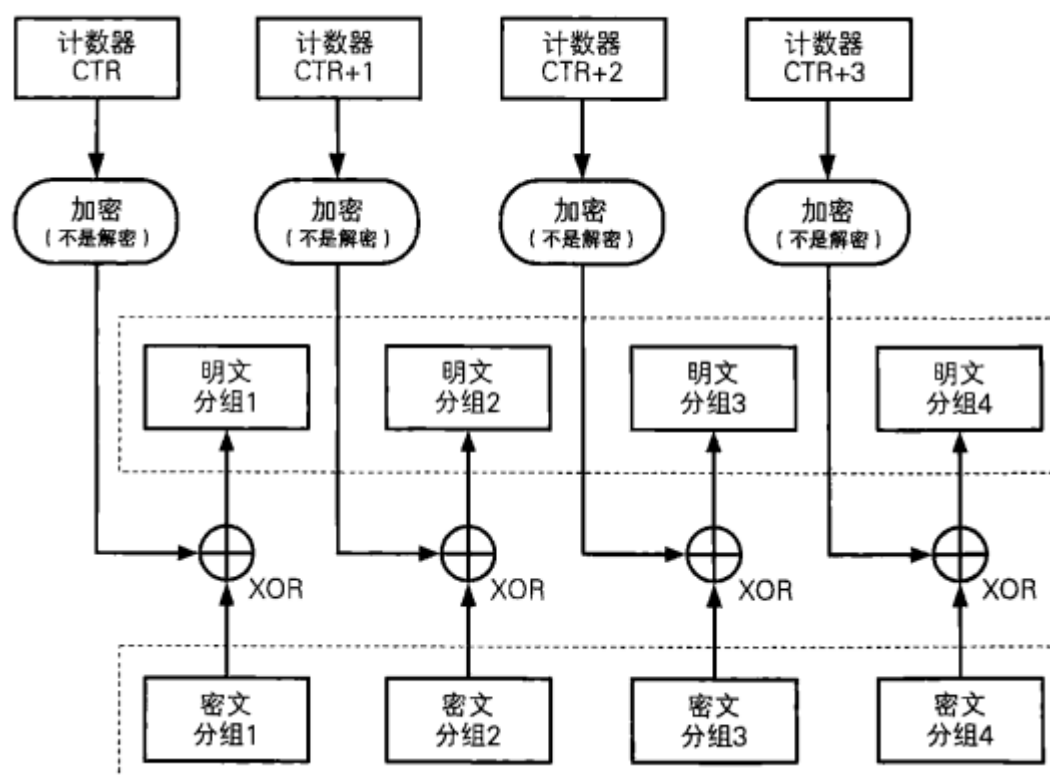
CTR模式的加密

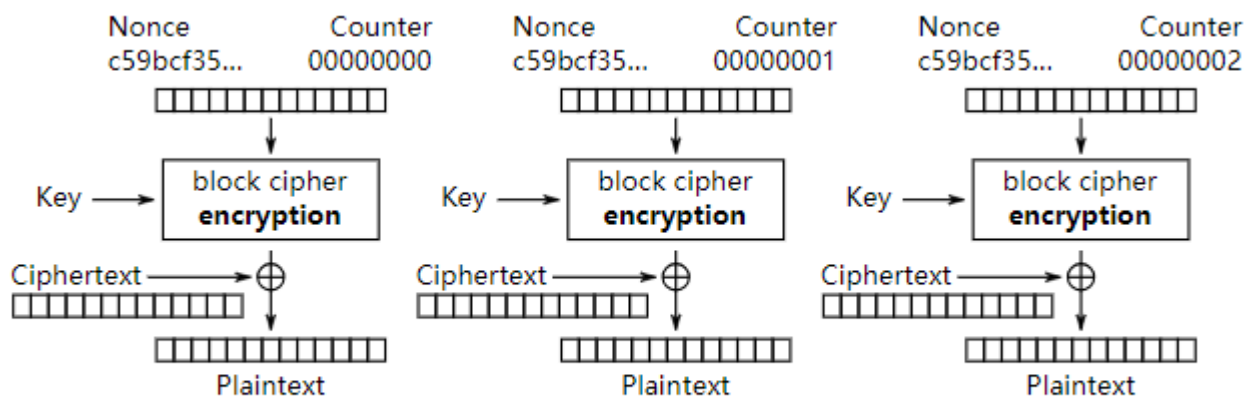




Counter (CTR) mode encryption

CTR模式的解密





Counter (CTR) mode decryption

## 计数器的生成方法

每次加密时都会生成一个不同的值（nonce）来作为计数器的初始值。当分组长度为128比特（16字节）时，计数器的初始值可能是像下面这样的形式。

```
66 1F 98 CD 37 A3 8B 4B 00 00 00 00 00 00 00 01
└────────── nonce ─────────┘ └──────── 分组序号 ─────────┘
```

其中前8个字节为nonce（随机数），这个值在每次加密时必须都是不同的，后8个字节为分组序号，这个部分是会逐次累加的。在加密的过程中，计数器的值会产生如下变化：

```
66 1F 98 CD 37 A3 8B 4B 00 00 00 00 00 00 00 01 明文分组 1 的计数器（初始值）
66 1F 98 CD 37 A3 8B 4B 00 00 00 00 00 00 00 02 明文分组 2 的计数器
66 1F 98 CD 37 A3 8B 4B 00 00 00 00 00 00 00 03 明文分组 3 的计数器
66 1F 98 CD 37 A3 8B 4B 00 00 00 00 00 00 00 04 明文分组 4 的计数器
```

按照上述生成方法，可以保证计数器的值每次都不同。由于计数器的值每次都不同，因此每个分组中将计数器进行加密所得到的密钥流也是不同的。也就是说，这种方法就是用分组密码来模拟生成随机的比特序列。

## OFB模式与CTR模式对比

CTR模式和OFB模式一样，都属于流密码。如果我们将单个分组的加密过程拿出来，那么OFB模式和CTR模式之间的差异还是很容易理解的（下图）。OFB模式是将加密的输出反馈到输入，而CTR模式则是将计数器的值用作输入。

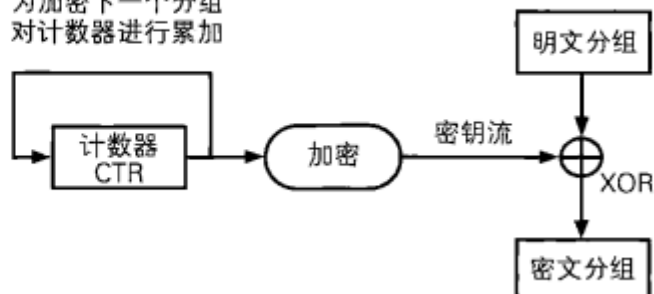
### OFB模式

加密下一个分组  
所需的反馈



### CTR模式

为加密下一个分组  
对计数器进行累加



## CTR模式的特点

CTR模式的加密和解密使用了完全相同的结构，因此在程序实现上比较容易。这一特点和同为流密码的OFB模式是一样的。

此外，CTR模式中可以对分组进行任意顺序的加密和解密，因此在加密和解密时需要用到的“计数器”的值可以由nonce和分组序号直接计算出来。这一性质是OFB模式所不具备的。

能够以任意顺序处理分组，就意味着能够实现并行计算。在支持并行计算的系统中，CTR模式的速度是非常快的。

## 3.7 总结

#

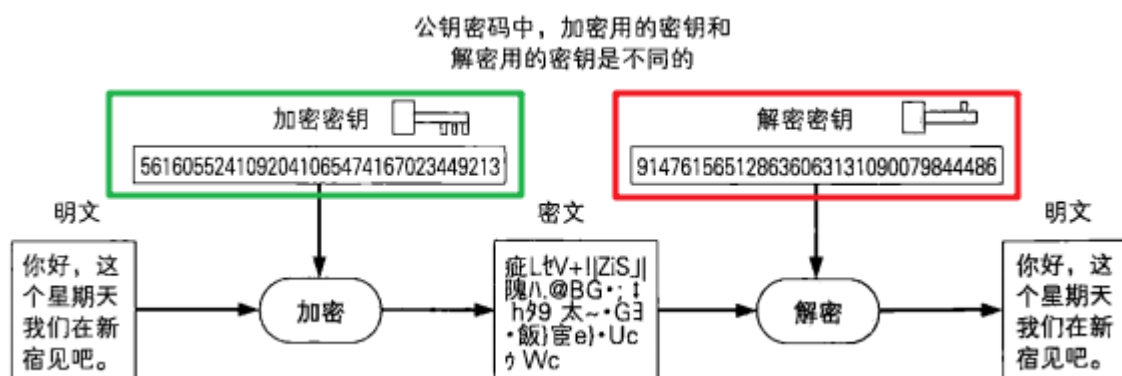
我们已经介绍了ECB、CBC、CFB、OFB和CTR模式，下面我们对这些模式的特点做一下整理。



模式	名称	优点	缺点	备注
ECB 模式	Electronic CodeBook 电子密码本模式	<ul style="list-style-type: none"> <li>简单</li> <li>快速</li> <li>支持并行计算（加密、解密）</li> </ul>	<ul style="list-style-type: none"> <li>明文中的重复排列会反映在密文中</li> <li>通过删除、替换密文分组可以对明文进行操作</li> <li>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</li> <li>不能抵御重放攻击</li> </ul>	不应使用
CBC 模式	Cipher Block Chaining 密文分组链接模式	<ul style="list-style-type: none"> <li>明文的重复排列不会反映在密文中</li> <li>支持并行计算（仅解密）</li> <li>能够解密任意密文分组</li> </ul>	<ul style="list-style-type: none"> <li>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</li> <li>加密不支持并行计算</li> </ul>	推荐使用
CFB 模式	Cipher-FeedBack 密文反馈模式	<ul style="list-style-type: none"> <li>不需要填充（padding）</li> <li>支持并行计算（仅解密）</li> <li>能够解密任意密文分组</li> </ul>	<ul style="list-style-type: none"> <li>加密不支持并行计算</li> <li>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</li> <li>不能抵御重放攻击</li> </ul>	<ul style="list-style-type: none"> <li>现在已不使用</li> <li>推荐用 CTR 模式代替</li> </ul>
OFB 模式	Output-FeedBack 输出反馈模式	<ul style="list-style-type: none"> <li>不需要填充（padding）</li> <li>可事先进行加密、解密的准备</li> <li>加密、解密使用相同结构</li> <li>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</li> </ul>	<ul style="list-style-type: none"> <li>不支持并行计算</li> <li>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</li> </ul>	推荐用 CTR 模式代替
CTR 模式	CounTeR 计数器模式	<ul style="list-style-type: none"> <li>不需要填充（padding）</li> <li>可事先进行加密、解密的准备</li> <li>加密、解密使用相同结构</li> <li>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</li> <li>支持并行计算（加密、解密）</li> </ul>	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

## 4. 非对称加密

1 "非对称加密也叫公钥密码：使用公钥加密，使用私钥解密"



在对称密码中，由于加密和解密的密钥是相同的，因此必须向接收者配送密钥。用于解密的密钥必须被配送给接收者，这一问题称为密钥配送问题。如果使用非对称加密也可以称为公钥密码，则无需向接收者配送用于解密的密钥，这样就解决了密钥配送问题。可以说非对称加密是密码学历史上最伟大的发明。

非对称加密中，密钥分为加密密钥和解密密钥两种。发送者用加密密钥对消息进行加密，接收者用解密密钥对密文进行解密。要理解公钥密码，清楚地区分加密密钥和解密密钥是非常重要的。加密密钥是发送者加密时使用的，而解密密钥则是接收者解密时使用的。

仔细思考一下加密密钥和解密密钥的区别，我们可以发现：

- 发送者只需要加密密钥
- 接收者只需要解密密钥
- 解密密钥不可以被窃听者获取
- 加密密钥被窃听者获取也没问题

也就是说，解密密钥从一开始就是由接收者自己保管的，因此只要将加密密钥发给发送者就可以解决密钥配送问题了，而根本不需要配送解密密钥。

非对称加密中，加密密钥一般是公开的。正是由于加密密钥可以任意公开，因此该密钥被称为**公钥**（**publickey**）。公钥可以通过邮件直接发送给接收者，也可以刊登在报纸的广告栏上，做成看板放在街上，或者做成网页公开给世界上任何人，而完全不必担心被窃听者窃取。

当然，我们也没有必要非要将公钥公开给全世界所有的人，但至少我们需要将公钥发送给需要使用公钥进行加密的通信对象（也就是给自己发送密文的发送者）。

相对地，**解密密钥是绝对不能公开的，这个密钥只能由你自己来使用，因此称为私钥（privatekey）**。私钥不可以被别人知道，也不可以将它发送给别人，甚至也不能发送给自己的通信对象。

公钥和私钥是一一对应的，一对公钥和私钥统称为**密钥对（keypair）**。由公钥进行加密的密文，必须使用与该公钥配对的私钥才能够解密。密钥对中的两个密钥之间具有非常密切的关系(数学上的关系)——因此公钥和私钥是不能分别单独生成的。

公钥密码的使用者需要生成一个包括公钥和私钥的密钥对，其中公钥会被发送给别人，而私钥则仅供自己使用。稍后我们将具体尝试生成一个密钥对。

## 4.1 非对称加密通信流程

#

下面我们来看一看使用公钥密码的通信流程。和以前一样、我们还是假设Alice要给Bob发送一条消息，Alice是发送者，Bob是接收者，而这一次窃听者Eve依然能够窃听到他们之间的通信内容。

在非对称加密通信中，通信过程是由接收者Bob来启动的。

1. Bob生成一个包含公钥和私钥的密钥对。

私钥由Bob自行妥善保管。

2. Bob将自己的公钥发送给Alice

Bob的公钥被窃听者Eve截获也没关系。

将公钥发送给Alice，表示Bob请Alice用这个公钥对消息进行加密并发送给他。

3. Alice用Bob的公钥对消息进行加密。

加密后的消息只有用Bob的私钥才能够解密。

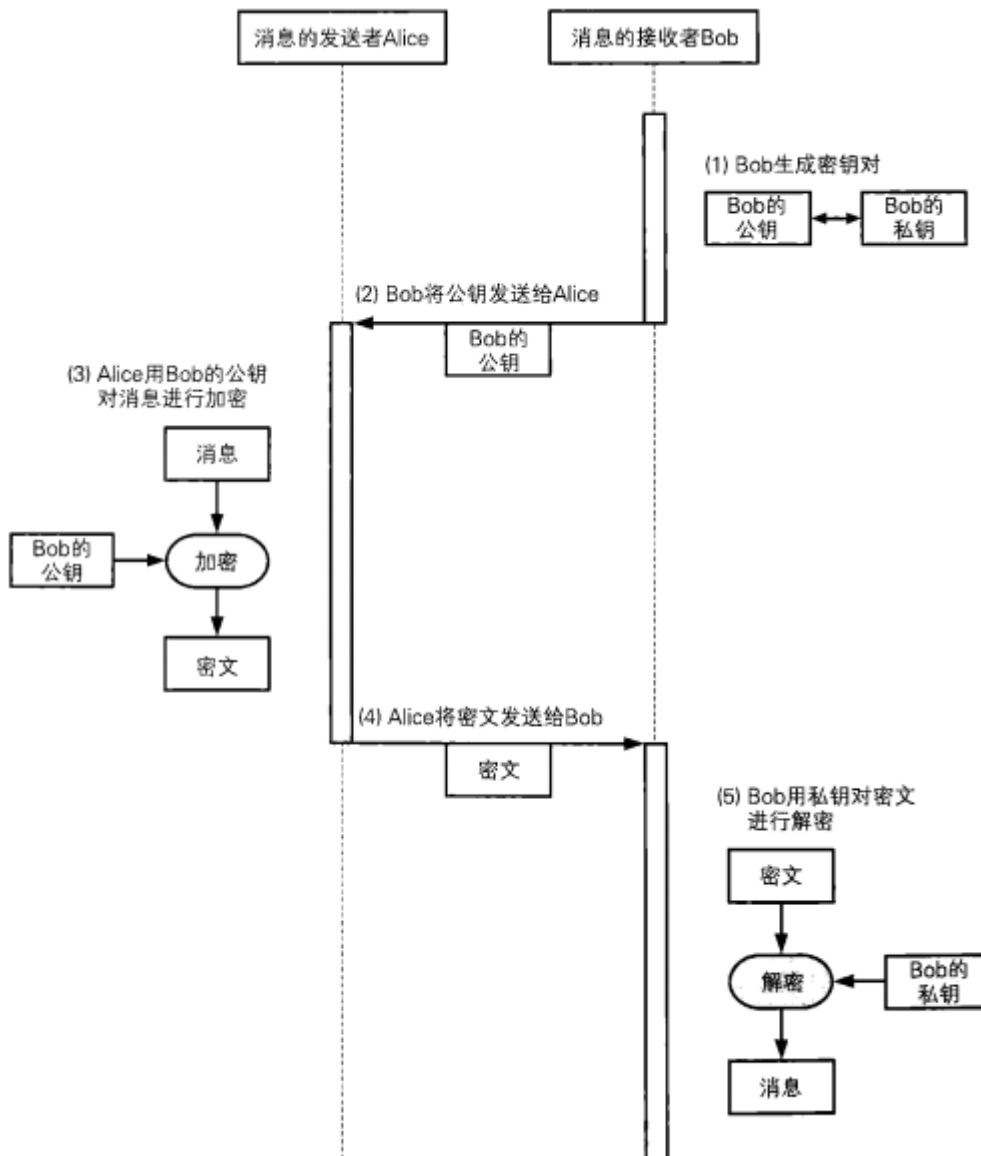
虽然Alice拥有Bob的公钥，但用Bob的公钥是无法对密文进行解密的。

#### 4. Alice将密文发送给Bob

密文被窃听者Eve截获也没关系。Eve可能拥有Bob的公钥，但是用Bob的公钥是无法进行解密的。

#### 5. Bob用自己的私钥对密文进行解密。

请参考下图, 看一看在Alice和Bob之间到底传输了哪些信息。其实它们之间所传输的信息只有两个：Bob的公钥以及用Bob的公钥加密的密文。由于Bob的私钥没有出现在通信内容中，因此窃听者Eve无法对密文进行解密。



窃听者Eve可能拥有Bob的公钥，但是Bob的公钥只是加密密钥，而不是解密密钥，因此窃听者Eve就无法完成解密操作。

非对称加密的密钥分为加密密钥和解密密钥，但这到底是怎样做到的呢？本节中我们来讲解现在使用最广泛的公钥密码算法——RSA。

RSA是一种非对称加密算法，它的名字是由它的三位开发者，即Ron Rivest、Adi Shamir和Leonard Adleman 的姓氏的首字母组成的（Rivest-Shamir-Leonard）。

RSA可以被用于非对称加密和数字签名，关于数字签名我们将在后面章节进行讲解。

1983年，RSA公司为RSA算法在美国取得了专利，但现在该专利已经过期。

### 4.2.1 RSA加密

下面我们终于可以讲一讲非对称加密的代表—RSA的加密过程了。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，如下。

$$\text{密文} = \text{明文}^E \bmod N \quad (\text{RSA加密}) \quad (1)$$

也就是说，RSA的密文是对代表明文的数字的E次方求modN的结果。换句话说，就是将明文自己做E次乘法，然后将其结果除以N求余数，这个余数就是密文。

噢，就这么简单？

对，就这么简单。仅仅对明文进行乘方运算并求mod即可，这就是整个加密的过程。在对称密码中，出现了很多复杂的函数和操作，就像做炒鸡蛋一样将**比特序列**挪来挪去，还要进行XOR(按位异或)等运算才能完成，但RSA却不同，它非常简洁。

对了，加密公式中出现的两个数——E和N，到底都是什么数呢？RSA的加密是求明文的E次方modN，因此只要知道E和N这两个数，任何人都可以完成加密的运算。所以说，E和N是RSA加密的密钥，也就是说，**E和N的组合就是公钥**。

不过，E和N并不是随便什么数都可以的，它们是经过严密计算得出的。顺便说一句，**E是加密 (Encryption) 的首字母，N是数字 (Number)的首字母**。

有一个很容易引起误解的地方需要大家注意——E和N这两个数并不是密钥对（公钥和私钥的密钥对）。E和N两个数才组成了一个公钥，因此我们一般会写成“公钥是(E, N)”或者“公钥是{E, N}”这样的形式，将E和N用括号括起来。

现在大家应该已经知道，**RSA的加密就是“求E次方的modN”**，接下来我们来看看RSA的解密。

### 4.2.2 RSA解密

RSA的解密和加密一样简单，可以用下面的公式来表达：

$$\text{明文} = \text{密文}^D \bmod N \quad (\text{RSA解密}) \quad (2)$$

也就是说，对表示密文的数字的D次方求modN就可以得到明文。换句话说，将密文自己做D次乘法，再对其结果除以N求余数，就可以得到明文。

这里所使用的数字N和加密时使用的数字N是相同的。**数D和数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的运算。

大家应该已经注意到，在RSA中，加密和解密的形式是相同的。加密是求“E次方的mod N”，而解密则是求“D次方的mod N”，这真是太美妙了。

当然，D也并不是随便什么数都可以的，作为解密密钥的D，和数字E有着相当紧密的联系。否则，用E加密的结果可以用D来解密这样的机制是无法实现的。

顺便说一句，D是解密（Decryption）的首字母，N是数字（Number）的首字母。

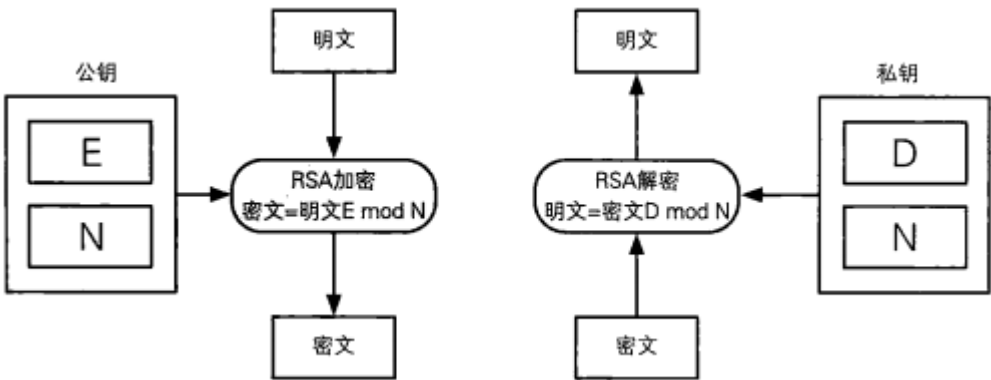
我们将上面讲过的内容整理一下，如下表所示。

密钥对	公钥	数 E 和数 N
	私钥	数 D 和数 N
加密	密文 = 明文 E mod N （明文的 E 次方除以 N 的余数）	
解密	明文 = 密文 D mod N （密文的 D 次方除以 N 的余数）	

1

解密\*\*

\*\*RSA的加密和



### 4.2.3 Go中生成公钥和私钥

需要引入的包

```
1 import (  
2     "crypto/rsa"  
3     "crypto/rand"  
4     "crypto/x509"  
5     "encoding/pem"  
6     "os"  
7 )
```

生成私钥操作流程概述

1. 使用rsa中的GenerateKey方法生成私钥
2. 通过x509标准将得到的ras私钥序列化为ASN.1 的 DER编码字符串
3. 将私钥字符串设置到pem格式块中
4. 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

1. 从得到的私钥对象中将公钥信息取出
2. 通过x509标准将得到 的rsa公钥序列化为字符串

3. 将公钥字符串设置到pem格式块中
4. 通过pem将设置好的数据进行编码,并写入磁盘文件

## 生成公钥和私钥的源代码

```
1 // 参数bits: 指定生成的密钥的长度, 单位: bit
2 func RsaGenKey(bits int) error{
3     // 1. 生成私钥文件
4     // GenerateKey函数使用随机数据生成器random生成一对具有指定字位数的RSA密钥
5     // 参数1: Reader是一个全局、共享的密码用强随机数生成器
6     // 参数2: 密钥的位数 - bit
7     privateKey, err := rsa.GenerateKey(rand.Reader, bits)
8     if err != nil{
9         return err
10    }
11    // 2. MarshalPKCS1PrivateKey将rsa私钥序列化为ASN.1 PKCS#1 DER编码
12    derStream := x509.MarshalPKCS1PrivateKey(privateKey)
13    // 3. Block代表PEM编码的结构, 对其进行设置
14    block := pem.Block{
15        Type: "RSA PRIVATE KEY", //"RSA PRIVATE KEY",
16        Bytes: derStream,
17    }
18    // 4. 创建文件
19    privFile, err := os.Create("private.pem")
20    if err != nil{
21        return err
22    }
23    // 5. 使用pem编码, 并将数据写入文件中
24    err = pem.Encode(privFile, &block)
25    if err != nil{
26        return err
27    }
28    // 6. 最后的时候关闭文件
29    defer privFile.Close()
30
31    // 7. 生成公钥文件
32    publicKey := privateKey.PublicKey
33    derPkix, err := x509.MarshalPKIXPublicKey(&publicKey)
34    if err != nil{
35        return err
36    }
37    block = pem.Block{
38        Type: "RSA PUBLIC KEY", //"PUBLIC KEY",
39        Bytes: derPkix,
40    }
41    pubFile, err := os.Create("public.pem")
42    if err != nil{
43        return err
44    }
45    // 8. 编码公钥, 写入文件
46    err = pem.Encode(pubFile, &block)
47    if err != nil{
48        panic(err)
49    }
```

```

49         return err
50     }
51     defer pubFile.Close()
52
53     return nil
54
55 }

```

重要的函数介绍:

1. GenerateKey函数使用随机数据生成器random生成一对具有指定字位数的RSA密钥。

```

1  "crypto/rsa" 包中的函数
2  func GenerateKey(random io.Reader, bits int) (priv *PrivateKey, err error)
3      - 参数1: io.Reader: 赋值为: rand.Reader
4        -- rand包实现了用于加解密的更安全的随机数生成器。
5        -- var Reader io.Reader (rand包中的变量)
6      - 参数2: bits: 秘钥长度
7      - 返回值1: 代表一个RSA私钥。
8      - 返回值2: 错误信息
9

```

2. 通过x509 将rsa私钥序列化为ASN.1 PKCS#1 DER编码

```

1  "crypto/x509" 包中的函数 (x509包解析X.509编码的证书和密钥)。
2  func MarshalPKCS1PrivateKey(key *rsa.PrivateKey) []byte
3      - 参数1: 通过rsa.GenerateKey得到的私钥
4      - 返回值: 将私钥通过ASN.1序列化之后得到的私钥编码数据

```

3. 设置Pem编码结构

```

1  Block代表PEM编码的结构。
2  type Block struct {
3      Type      string           // 得自前言的类型 (如"RSA PRIVATE KEY")
4      Headers   map[string]string // 可选的头项, Headers是可为空的多行键值对。
5      Bytes     []byte           // 内容解码后的数据, 一般是DER编码的ASN.1结构
6  }

```

4. 将得到的Pem格式私钥通过文件指针写入磁盘中

```

1  "encoding/pem" 包中的函数
2  func Encode(out io.Writer, b *Block) error
3      - 参数1: 可进行写操作的IO对象, 此处需要指定一个文件指针
4      - 参数2: 初始化完成的Pem块对象, 即Block对象

```

5. 通过RSA私钥得到公钥

```

1  // 私钥
2  type PrivateKey struct {
3      PublicKey      // 公钥
4      D              *big.Int // 私有的指数
5      Primes         []*big.Int // N的素因子, 至少有两个
6      // 包含预先计算好的值, 可在某些情况下加速私钥的操作
7      Precomputed    PrecomputedValues
8  }

```



```

9 // 公钥
10 type PublicKey struct {
11     N *big.Int // 模
12     E int      // 公开的指数
13 }
14 通过私钥获取公钥
15 publicKey := privateKey.PublicKey // privateKey为私钥对象

```

6. 通过x509将公钥序列化为PKIX格式DER编码。

```

1 "crypto/x509" 包中的函数
2 func MarshalPKIXPublicKey(pub interface{}) ([]byte, error)
3     - 参数1: 通过私钥对象得到的公钥
4     - 返回值1: 将公钥通过ASN.1序列化之后得到的编码数据
5     - 返回值2: 错误信息

```

7. 将公钥编码之后的数据格式化为Pem结构, 参考私钥的操作

8. 将得到的Pem格式公钥通过文件指针写入磁盘中

9. 生成的私钥和公钥文件数据

```

1 // 私钥文件数据
2 -----BEGIN RSA PRIVATE KEY-----
3 MIICXgIBAAKBgQC5bm0DCEV+EFeiLUqSshziqhSB30jXy5BWbPV5S1Mq4aWiEknM
4 i+Mw1aXic4bEsM3YyT73eWsisfQZNSc/4fRaV4qz50L8IIe9AZoGDSLX/Ar9AQMjF
5 0HbAtdI1CGQ4d80KjpDpPs2wZkTq1lWCg31d7U3DVE5mkqTgtSYIu9e7JQIDAQAB
6 AoGAR6dn72ZtvEnREHiEufjaJwM07Zng1TpS1I79PvEcHQWAhHkaoEo6VR17SD41
7 yPkv9njGsaQo0WDHGFvSTGhYm/EWGrBWRPc5xXbSBg7ty9Iza9B1ekAj8VfWryen
8 Wje3xD0CVCdUicCYdaSfPiJPYUwMSnNMNa+0cR921zBQg0ECQQDpCm1juH7LrpbC
9 NDF5q+LbUWMAE2KLDPX4WmDSdZdIO3mPux3Mdw0UEfrcvSBGZNB7gyaEG7goZL8G
10 BqL22MJHAKAey7SqVPoPbMPHuLI52VQ2FDp6xxSWLhjmV1ePCHGo28MDCaHeVzZ
11 QaxyuIbnY8A6NHfu/QGwz/eB941IjYNBMwJBAI9XEE1+mr++zIz4fdZRnGE7VqId
12 SmgtuL7jGNtb6YpMyFV/6ZdLp5N0PkmfEvQh0zyBycLxeNS1Q1n16Xu/tECQQCZ
13 dF42wdDgOfWYFMu31VETw9CTtuApya3vYhMNRXx4Pf1bYeMIf/OCT8CUVbwWHwc5
14 42d73TwwTorvy9TuFgSVAKEA6F69TH1Tn5oIP8IWHcHuqS01fIR/vGfEwQ4cFZGR
15 ketfieyeeF8rjn4qzwT/ugwRNjkhfKmoILnIC8UheEJdJA==
16 -----END RSA PRIVATE KEY-----

```

```

1 // 公钥文件数据
2 -----BEGIN RSA PUBLIC KEY-----
3 MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC5bm0DCEV+EFeiLUqSshziqhSB
4 30jXy5BWbPV5S1Mq4aWiEknMi+Mw1aXic4bEsM3YyT73eWsisfQZNSc/4fRaV4qz5
5 0L8IIe9AZoGDSLX/Ar9AQMjF0HbAtdI1CGQ4d80KjpDpPs2wZkTq1lWCg31d7U3D
6 VEm5kqTgtSYIu9e7JQIDAQAB
7 -----END RSA PUBLIC KEY-----

```

#### 4.2.4 Go中使用RSA

1. 操作步骤

- 公钥加密



1. 将公钥文件中的公钥读出, 得到使用pem编码的字符串
2. 将得到的字符串解码
3. 使用x509将编码之后的公钥解析出来
4. 使用得到的公钥通过rsa进行数据加密

- 私钥解密

1. 将私钥文件中的私钥读出, 得到使用pem编码的字符串
2. 将得到的字符串解码
3. 使用x509将编码之后的私钥解析出来
4. 使用得到的私钥通过rsa进行数据解密

## 2. 代码实现

- RSA公钥加密

```
1 func RSAEncrypt(src, filename []byte) []byte {
2     // 1. 根据文件名将文件内容从文件中读出
3     file, err := os.Open(string(filename))
4     if err != nil {
5         return nil
6     }
7     // 2. 读文件
8     info, _ := file.Stat()
9     allText := make([]byte, info.Size())
10    file.Read(allText)
11    // 3. 关闭文件
12    file.Close()
13
14    // 4. 从数据中查找到下一个PEM格式的块
15    block, _ := pem.Decode(allText)
16    if block == nil {
17        return nil
18    }
19    // 5. 解析一个DER编码的公钥
20    pubInterface, err := x509.ParsePKIXPublicKey(block.Bytes)
21    if err != nil {
22        return nil
23    }
24    pubKey := pubInterface.(*rsa.PublicKey)
25
26    // 6. 公钥加密
27    result, _ := rsa.EncryptPKCS1v15(rand.Reader, pubKey, src)
28    return result
29 }
```

- RSA私钥解密

```
1 func RSADecrypt(src, filename []byte) []byte {
2     // 1. 根据文件名将文件内容从文件中读出
3     file, err := os.Open(string(filename))
4     if err != nil {
```

```

5         return nil
6     }
7     // 2. 读文件
8     info, _ := file.Stat()
9     allText := make([]byte, info.Size())
10    file.Read(allText)
11    // 3. 关闭文件
12    file.Close()
13    // 4. 从数据中查找到下一个PEM格式的块
14    block, _ := pem.Decode(allText)
15    // 5. 解析一个pem格式的私钥
16    privateKey, err := x509.ParsePKCS1PrivateKey(block.Bytes)
17    // 6. 私钥解密
18    result, _ := rsa.DecryptPKCS1v15(rand.Reader, privateKey, src)
19
20    return result
21 }

```

## ○ 重要的函数介绍

1. 将得到的Pem格式私钥通过文件指针写入磁盘中

```

1  "encoding/pem" 包中的函数
2  func Decode(data []byte) (p *Block, rest []byte)
3      - 参数 data: 需要解析的数据块
4      - 返回值1: 从参数中解析出的PEM格式的块
5      - 返回值2: 参数data剩余的未被解码的数据

```

2. 解析一个DER编码的公钥, pem中的Block结构体中的数据格式为ASN.1编码

```

1  函数所属的包: "crypto/x509"
2  func ParsePKIXPublicKey(derBytes []byte) (pub interface{}, err error)
3      - 参数 derBytes: 从pem的Block结构体中取的ASN.1编码数据
4      - 返回值 pub: 接口对象, 实际是公钥数据
5      - 参数 err: 错误信息

```

3. 解析一个DER编码的私钥, pem中的Block结构体中的数据格式为ASN.1编码

```

1  函数所属的包: "crypto/x509"
2  func ParsePKCS1PrivateKey(der []byte) (key *rsa.PrivateKey, err error)
3      - 参数 der: 从pem的Block结构体中取的ASN.1编码数据
4      - 返回值 key: 解析出的私钥
5      - 返回值 err: 错误信息

```

4. 将接口转换为公钥

```

1  pubKey := pubInterface.(*rsa.PublicKey)
2      - pubInterface: ParsePKIXPublicKey函数返回的 interface{} 对象
3      - pubInterface.(*rsa.PublicKey): 将pubInterface转换为公钥类型
4      rsa.PublicKey

```

5. 使用公钥加密数据

```

1  函数所属的包: "crypto/rsa"
2  func EncryptPKCS1v15(rand io.Reader, pub *PublicKey, msg []byte) (out
    []byte, err error)
3      - 参数 rand: 随机数生成器, 赋值为 rand.Reader
4      - 参数 pub: 非对称加密加密使用的公钥
5      - 参数 msg: 要使用公钥加密的原始数据
6      - 返回值 out: 加密之后的数据
7      - 返回值 err: 错误信息

```

#### 6. 使用私钥解密数据

```

1  函数所属的包: "crypto/rsa"
2  func DecryptPKCS1v15(rand io.Reader, priv *PrivateKey, ciphertext []byte)
    (out []byte, err error)
3      - 参数 rand: 随机数生成器, 赋值为 rand.Reader
4      - 参数 priv: 非对称加密解密使用的私钥
5      - 参数 ciphertext: 需要使用私钥解密的数据
6      - 返回值 out: 解密之后得到的数据
7      - 返回值 err: 错误信

```

## 4.3 ECC椭圆曲线

#

### 1. 概念

椭圆曲线密码学（英语：Elliptic curve cryptography，缩写为 ECC），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年由Neal Koblitz和Victor Miller分别独立提出的。

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——提供相当的或更高等级的安全。

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的加密方法不同，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相当于RSA 1024位密钥提供的保密强度，而且计算量较小，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 位的椭圆曲线密码，虚拟货币 **比特币** 也选择ECC作为加密算法。

具体算法详解参考：

- <https://www.cnblogs.com/Kalafinaian/p/7392505.html>
- <https://blog.csdn.net/taifei/article/details/73277247>

### 2. 数学原理

不管是RSA还是ECC或者其它，公钥加密算法都是依赖于某个正向计算很简单（多项式时间复杂度），而逆向计算很难（指数级时间复杂度）的数学问题。

椭圆曲线依赖的数学难题是：

$k$  为正整数， $P$  是椭圆曲线上的点（称为基点）， $k \cdot P = Q$ ，已知  $Q$  和  $P$ ，很难计算出  $k$

## 4.4 非对称加密解惑

#

- 非对称加密比对称加密机密性更高吗？

这个问题无法回答, 以为机密性高低是根据密钥长度而变化的

- 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, 是密钥更长的非对称加密更安全吗?

不是。

非对称加密的密钥长度不能与对称加密的密钥长度进行直接比较。下表是一张密钥长度的比较表（本表摘自《应用密码学》），根据这张表我们可以看出，1024比特的公钥密码与128比特的对称密码相比，反而是128比特的对称密码抵御暴力破解的能力更强。

对称加密密钥长度	非对称加密密钥长度
128 比特	2304 比特
112 比特	1792 比特
80 比特	768 比特
64 比特	512 比特
56 比特	384 比特

- 有了非对称加密，以后对称加密会被替代吗？

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，非对称加密并不适合用来对很长的消息内容进行加密。根据目的的不同，还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。

## 5. 单向散列函数

1 "单向散列函数 --- 获取消息的指纹"

在刑事侦查中，侦查员会用到指纹。通过将某个特定人物的指纹与犯罪现场遗留的指纹进行对比，就能够知道该人物与案件是否存在关联。

针对计算机所处理的消息，有时候我们也需要用到“指纹”。当需要比较两条消息是否一致时，我们不必直接对比消息本身的内容，只要对比它们的“指纹”就可以了。

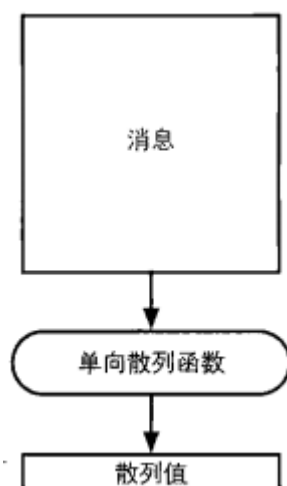
本章中，我们将学习单向散列函数的相关知识。使用单向散列函数就可以获取消息的“指纹”，通过对比“指纹”，就能够知道两条消息是否一致。

下面，我们会先简单介绍一下单向散列函数，并给大家展示具体的例子。然后我们将详细介绍现在使用非常广泛的SHA-1单向散列函数。

### 5.1 什么是单向散列函数

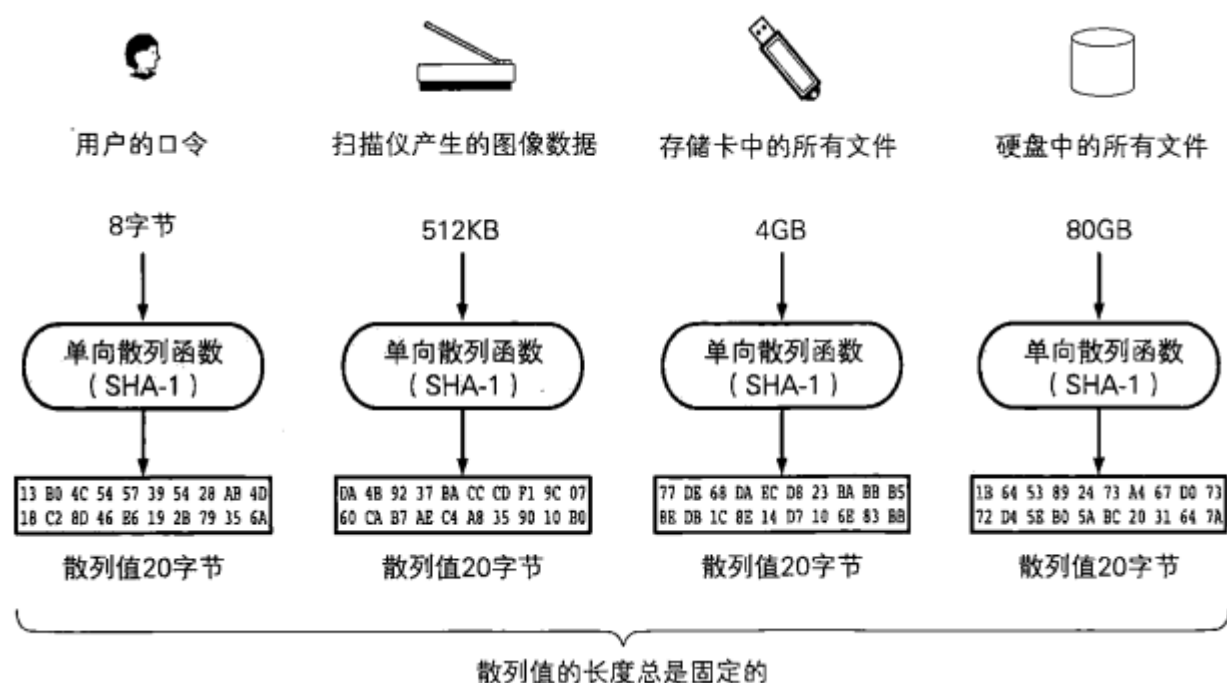
#

单向散列函数（one-way function）有一个输入和一个输出，其中输入称为消息（message），输出称为散列值（hash value）。单向散列函数可以根据消息的内容计算出散列值，而散列值就可以被用来检查消息的完整性。



这里的消息不一定是人类能够读懂的文字，也可以是图像文件或者声音文件。单向散列函数不需要知道消息实际代表的含义。无论任何消息，单向散列函数都会将它作为单纯的比特序列来处理，即根据比特序列计算出散列值。

散列值的长度和消息的长度无关。无论消息是1比特，还是100MB，甚至是100GB，单向散列函数都会计算出固定长度的散列值。以SHA-1单向散列函数为例，它所计算出的散列值的长度永远是160比特（20字节）。



## 5.2 关于术语

#

单向散列函数的相关术语有很多变体，不同参考资料中所使用的术语也不同，下面我们就介绍其中的几个。

单向散列函数也称为**消息摘要函数**（message digest function）、**哈希函数**或者**杂凑函数**。

输入单向散列函数的消息也称为**原像**（pre-image）。

单向散列函数输出的散列值也称为**消息摘要**（message digest）或者**指纹**（fingerprint）。

**完整性**也称为一致性。

顺便说一句，单向散列函数中的“散列”的英文“hash”一词，原意是古法语中的“斧子”，后来被引申为“剁碎的肉末”，也许是用斧子一通乱剁再搅在一起的那种感觉吧。单向散列函数的作用，实际上就是将很长的消息剁碎，然后再混合成固定长度的散列值。

## 5.3 单向散列函数的性质

#

通过使用单向散列函数，即便是确认几百MB大小的文件的完整性，也只要对比很短的散列值就可以了。那么，单向散列函数必须具备怎样的性质呢？我们来整理一下。

- **根据任意长度的消息计算出固定长度的散列值**

首先，单向散列函数的输入必须能够是任意长度的消息。

其次，无论输入多长的消息，单向散列函数必须都能够生成长度很短的散列值，如果消息越长生成的散列值也越长的话就不好用了。从使用方便的角度来看，散列值的长度最好是短且固定的。

- **能够快速计算出散列值**

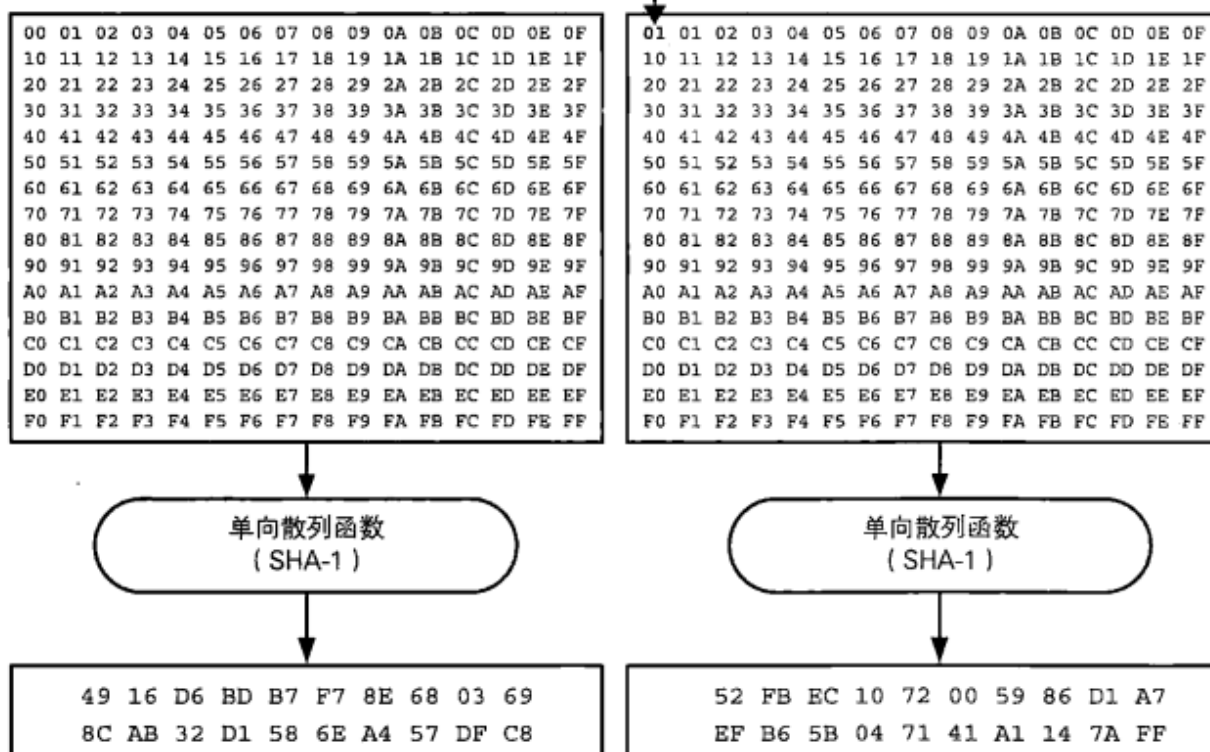
计算散列值所花费的时间必须要短。尽管消息越长，计算散列值的时间也会越长，但如果不能在现实的时间内完成计算就没有意义了。

- **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，也必须有很高的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的检查，则需要确保在事实上不可能被人为地发现碰撞。

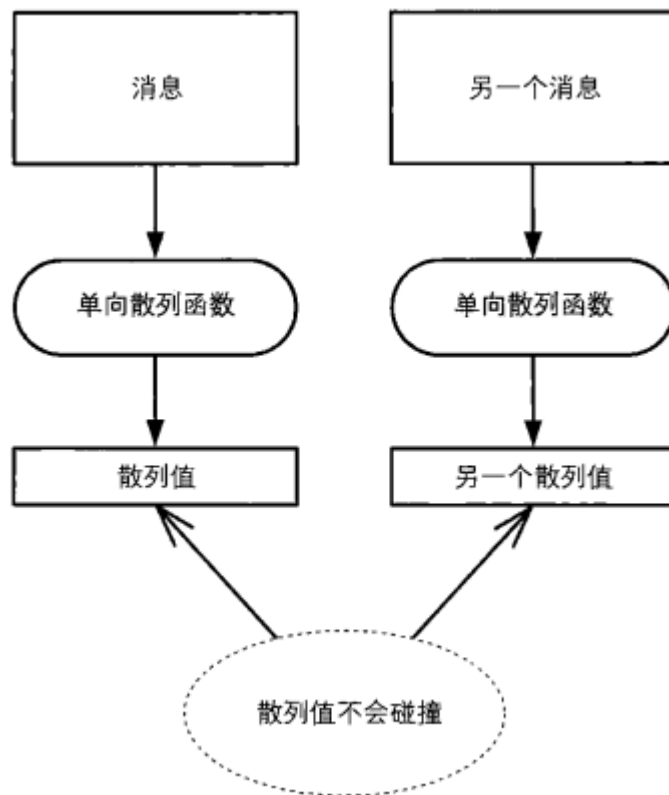
将消息的00改为01  
(1比特的改变)



哪怕消息只有1比特的区别，也会产生完全不同的散列值

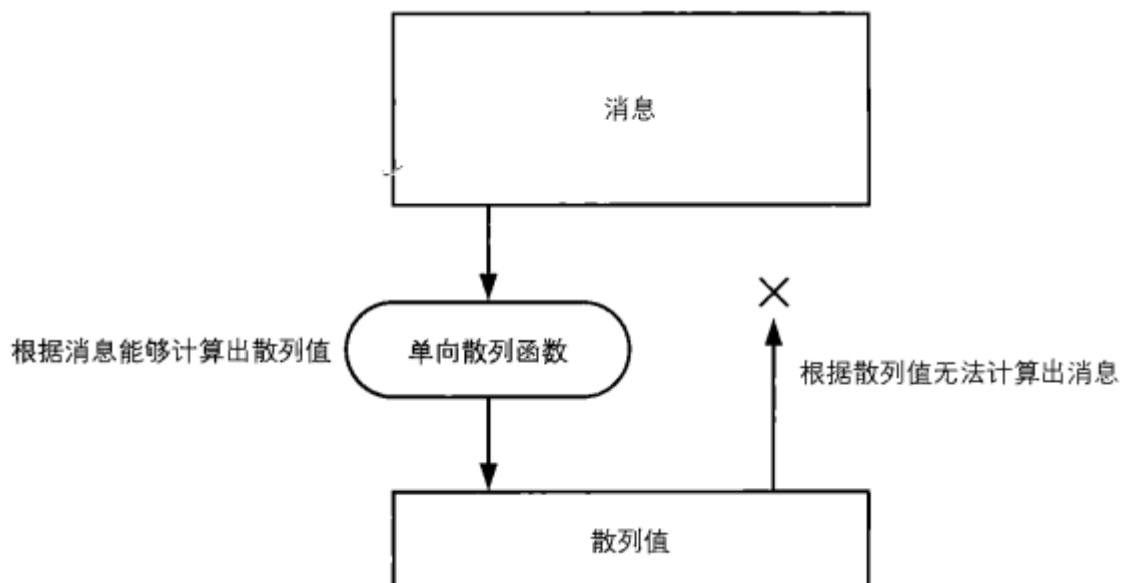
难以发现碰撞的性质称为抗碰撞性 (collision resistance)。密码技术中所使用的单向散列函数，都需要具备抗碰撞性。

强抗碰撞性，是指要找到散列值相同的两条不同的消息是非常困难的这一性质。在这里，散列值可以是任意值。密码技术中的单向散列函数必须具备强抗碰撞性。



- 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反过来走的。



正如同将玻璃砸得粉碎很容易，但却无法将碎片还原成完整的玻璃一样，根据消息计算出散列值很容易，但根据散列值却无法反算出消息。

在这里需要注意的一点是，尽管单向散列函数所产生的散列值是和原来的消息完全不同的比特序列，但是**单向散列函数并不是一种加密，因此无法通过解密将散列值还原为原来的消息。**



### 5.4 单向散列函数的实际应用

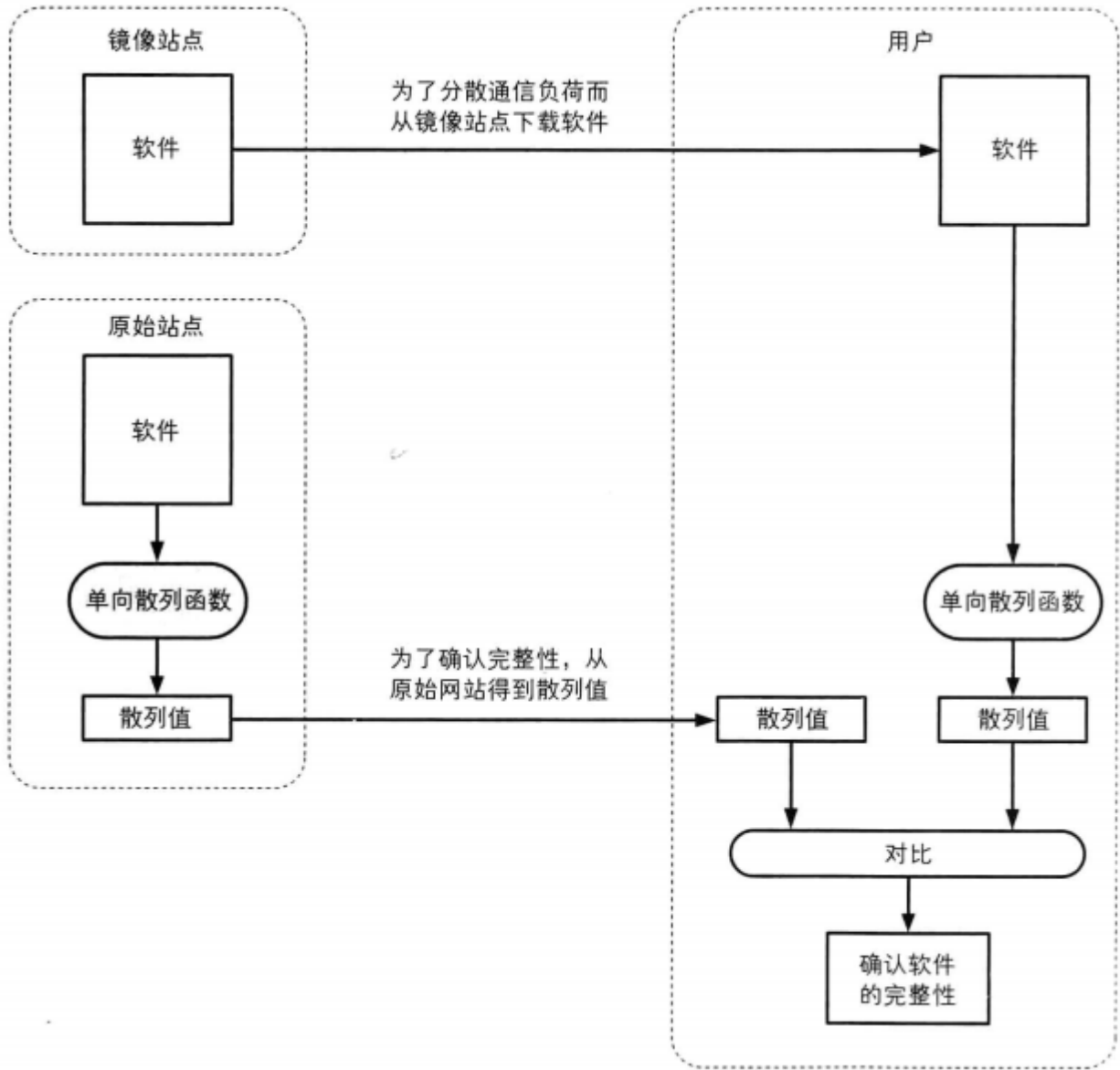
下面我们来看一下实际应用单向散列函数的例子。

#### 5.4.1 检测软件是否被篡改

我们可以使用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把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。用户在下载到软件之后，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，用户可以确认自己所下载到的文件与软件作者所提供的文件是否一致。

这样的方法，在可以通过多种途径得到软件的情况下非常有用。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。



#### 5.4.2 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“发送者和接收者之间的共享密钥”和“消息，进行混合后计算出的散列值。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我们将后边章节中介绍。

### 5.4.3 数字签名

在进行数字签名时也会使用单向散列函数。

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。数字签名的处理过程非常耗时，因此一般不会对整个消息内容直接施加数字签名，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。

### 5.4.6 伪随机数生成器

使用单向散列函数可以构造伪随机数生成器。

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。

### 5.4.7 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客户端的合法性认证。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（one-time），因此即使窃听者窃取了口令，也无法使用。

## 5.5 常用的单向散列函数

#

### 5.5.1 MD4、MD5

MD4是由Rivest于1990年设计的单向散列函数，能够产生128比特的散列值（RFC1186，修订版RFC1320）。不过，随着Dobbertin提出寻找MD4散列碰撞的方法，因此现在它已经不安全了。

MD5是由Rwest于1991年设计的单项散列函数，能够产生128比特的散列值（RFC1321）。

MD5的强抗碰撞性已经被攻破，也就是说，现在已经能够产生具备相同散列值的两条不同的消息，因此它也已经不安全了。

MD4和MD5中的MD是消息摘要（Message Digest）的缩写。

### 5.5.2 Go中使用MD5

- 需要导入的包

```

1 import (
2     "crypto/md5"
3     "encoding/hex"
4 )

```

## • 计算Md5的方式1

```

1 func getMD5_1(str []byte) string {
2     // 1. 计算数据的md5
3     result := md5.Sum(str)
4     fmt.Println(result)
5     fmt.Printf("%x\n", result)
6     // 2. 数据格式化为16进制格式字符串
7     res := fmt.Sprintf("%x", result)
8     fmt.Println(res)
9     // --- 这是另外一种格式化切片的方式
10    res = hex.EncodeToString(result[:])
11    fmt.Println("res: ", res)
12    return res
13 }

```

重要函数说明:

### 1. 返回数据data的MD5校验和

```

1 函数所属的包: "crypto/md5"
2 func Sum(data []byte) [Size]byte
3     - 参数 data: 原始数据
4     - 返回值: 经过md5计算之后得到的数据, 长度为 16字节(byte)

```

### 2. 将字符串编码为16进制格式

```

1 函数所属的包: "encoding/hex"
2 func EncodeToString(src []byte) string
3     - 参数 src: 要转换的数据
4     - 返回值: 转换之后得到的16进制格式字符串

```

## • 计算Md5的方式2

```

1 func getMD5_2(str []byte) string {
2     // 1. 创建一个使用MD5校验的Hash对象
3     myHash := md5.New()
4     // 2. 通过io操作将数据写入hash对象中
5     io.WriteString(myHash, "hello")
6     //io.WriteString(myHash, ", world")
7     myHash.Write([]byte(", world"))
8     // 3. 计算结果
9     result := myHash.Sum(nil)
10    fmt.Println(result)
11    // 4. 将结果转换为16进制格式字符串
12    res := fmt.Sprintf("%x", result)
13    fmt.Println(res)
14    // --- 这是另外一种格式化切片的方式
15    res = hex.EncodeToString(result)

```

```

16     fmt.Println(res)
17
18     return res
19 }

```

重要函数说明:

#### 1. 创建一个新的使用MD5校验的hash.Hash接口

```

1  函数所属的包: "crypto/md5"
2  func New() hash.Hash

```

Hash是一个被所有hash函数实现的公共接口。

```

1  type Hash interface {
2      // 通过嵌入的匿名io.Writer接口的Write方法向hash中添加更多数据, 永远不返回错误
3      io.Writer
4      // 返回添加b到当前的hash值后的新切片, 不会改变底层的hash状态
5      Sum(b []byte) []byte
6      // 重设hash为无数据输入的状态
7      Reset()
8      // 返回Sum会返回的切片的长度
9      Size() int
10     // 返回hash底层的块大小; Write方法可以接受任何大小的数据,
11     // 但提供的数据是块大小的倍数时效率更高
12     BlockSize() int
13 }
14
15 "io" 包中 Writer 接口用于包装基本的写入方法。
16 type Writer interface {
17     Write(p []byte) (n int, err error)
18 }

```

#### 2. 通过io操作将数据写入hash对象中

```

1  # 第一种方式
2  函数所属的包: "io"
3  func WriteString(w Writer, s string) (n int, err error)
4      - 参数 w: 实现了/包含Writer接口的对象
5      - 参数 s: 要添加到IO对象中的数据
6      - 返回值 n: 数据长度
7      - 返回值 err: 错误信息
8  # 第二种方式
9  使用md5包中的New()方法得到的hash.Hash接口(假设名为: myHash)添加数据
10 myHash.Write([]byte("测试数据"))

```

#### 3. 使用hash.Hash接口中的Sum方法计算结果

```

1  Sum(b []byte) []byte
2      - 参数 b: 将b中的数据进行哈希计算, 结果添加到原始数据的前面,
3      一般情况下该参数指定为空, 即: nil
4      - 返回值: 进行哈希运算之后得到的结果

```

### 5.5.3 SHA-1、SHA-224、SHA-256、SHA-384、SHA-512

SHA-1是由NIST（NationalInstituteOfStandardsandTechnology，美国国家标准技术研究所）设计的一种能够产生160比特的散列值的单向散列函数。1993年被作为美国联邦信息处理标准规格（FIPS PUB 180）发布的是SHA,1995年发布的修订版FIPS PUB 180-1称为SHA-1。

SHA-1的消息长度存在上限，但这个值接近于 $2^{64}$ 比特，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，它们的散列值长度分别为256比特、384比特和512比特。这些单向散列函数合起来统称SHA-2，它们的消息长度也存在上限（SHA-256的上限接近于 $2^{64}$ 比特，SHA-384和SHA-512的上限接近于 $2^{128}$ 比特）。这些单向散列函数是于2002年和SHA-1一起作为FIPS PUB 180-2发布的SHA-1的强抗碰撞性已于2005年被攻破,也就是说，现在已经能够产生具备相同散列值的两条不同的消息。不过，SHA-2还尚未被攻破。

	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

## 5.5.4 Go中对SHA-1、SHA-2的使用

- 需要导入的包

```
1 import (  
2     "crypto/sha1"  
3     "encoding/hex"  
4     "crypto/sha256"  
5     "crypto/sha512"  
6 )
```

- 使用sha1计算文件指纹

上一小节介绍了如何使用go提供的API计算数据的md5指纹, sha1的计算方式和md5的套路是一样的, 需要将md5包, 替换为sh1, 下面给大家介绍一下如何使用sha1计算文件的指纹(md5亦如此)

```
1 func getSha1(src string) string {  
2     // 1. 打开文件  
3     fp, err := os.Open(src)  
4     if err != nil {  
5         return "文件打开失败"  
6     }  
7     // 2. 创建基于sha1算法的Hash对象  
8     myHash := sha1.New()
```

```

9      // 3. 将文件数据拷贝给哈希对象
10     num, err := io.Copy(myHash, fp)
11     if err != nil {
12         return "拷贝文件失败"
13     }
14     fmt.Println("文件大小: ", num)
15     // 4. 计算文件的哈希值
16     tmp1 := myHash.Sum(nil)
17     // 5. 数据格式转换
18     result := hex.EncodeToString(tmp1)
19     fmt.Println("sha1: ", result)
20
21     return result
22 }

```

## 6. 消息认证码

1 "消息认证码 --- 消息被正确传送了吗?"

### 6.1 什么是消息认证码

#

- Alice 和 Bob 的故事

像以前一样，我们还是从一个Alice和Bob的故事开始讲起。不过，这一次Alice和Bob分别是两家银行，Alice银行通过网络向Bob银行发送了一条汇款请求，Bob银行收到的请求内容是：

从账户  $A - 5374$  向账户  $B - 6671$  汇款 1000万元 (3)

当然，Bob银行所收到的汇款请求内容必须与Alice银行所发送的内容是完全一致的。如果主动攻击者Mallory在中途将Alice银行发送的汇款请求进行了篡改，那么Bob银行就必须能够识别出这种篡改，否则如果Mallory将收款账户改成了自己的账户，那么1000万元就会被盗走。

话说回来，这条汇款请求到底是不是Alice银行发送的呢？有可能Alice银行根本就没有发送过汇款请求，而是由主动攻击者Mallory伪装成Alice银行发送的。如果汇款请求不是来自Alice银行，那么就绝对不能执行汇款。

现在我们需要关注的问题是汇款请求（消息）的“完整性”和“认证”这两个性质。

**消息的完整性 (integrity)**, 指的是“消息没有被篡改”这一性质，完整性也叫一致性。如果能够确认汇款请求的内容与Alice银行所发出的内容完全一致，就相当于是确认了消息的完整性，也就意味着消息没有被篡改。

**消息的认证 (authentication)** 指的是“消息来自正确的发送者”这一性质。如果能够确认汇款请求确实来自Alice银行，就相当于对消息进行了认证，也就意味着消息不是其他人伪装成发送者所发出的。

通过使用本章中要介绍的消息认证码，我们就可以同时识别出篡改和伪装，也就是既可以确认消息的完整性，也可以进行认证。

- 什么是消息认证码

**消息认证码（message authentication code）是一种确认完整性并进行认证的技术，取三个单词的首字母，简称为MAC。**

消息认证码的输入包括任意长度的消息和一个发送者与接收者之间**共享的密钥**，它可以输出固定长度的数据，这个数据称为**MAC值**。

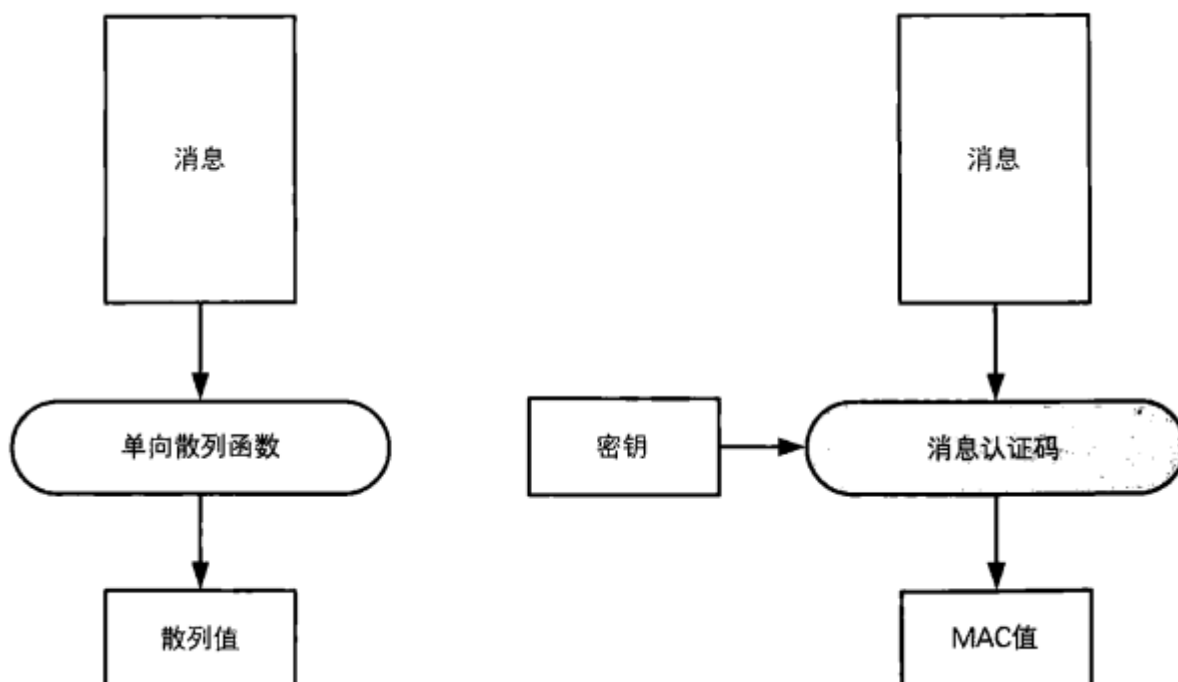
根据任意长度的消息输出固定长度的数据，这一点和单向散列函数很类似。但是单向散列函数中计算散列值时不需要密钥，而消息认证码中则需要使用发送者与接收者之间共享的密钥。

要计算MAC必须持有共享密钥，没有共享密钥的人就无法计算MAC值，消息认证码正是利用这一性质来完成认证的。此外，和单向散列函数的散列值一样，哪怕消息中发生1比特的变化，MAC值也会产生变化，消息认证码正是利用这一性质来确认完整性的。

消息认证码有很多种实现方法，大家可以暂且这样理解：消息认证码是一种与密钥相关联的单向散列函数。

1

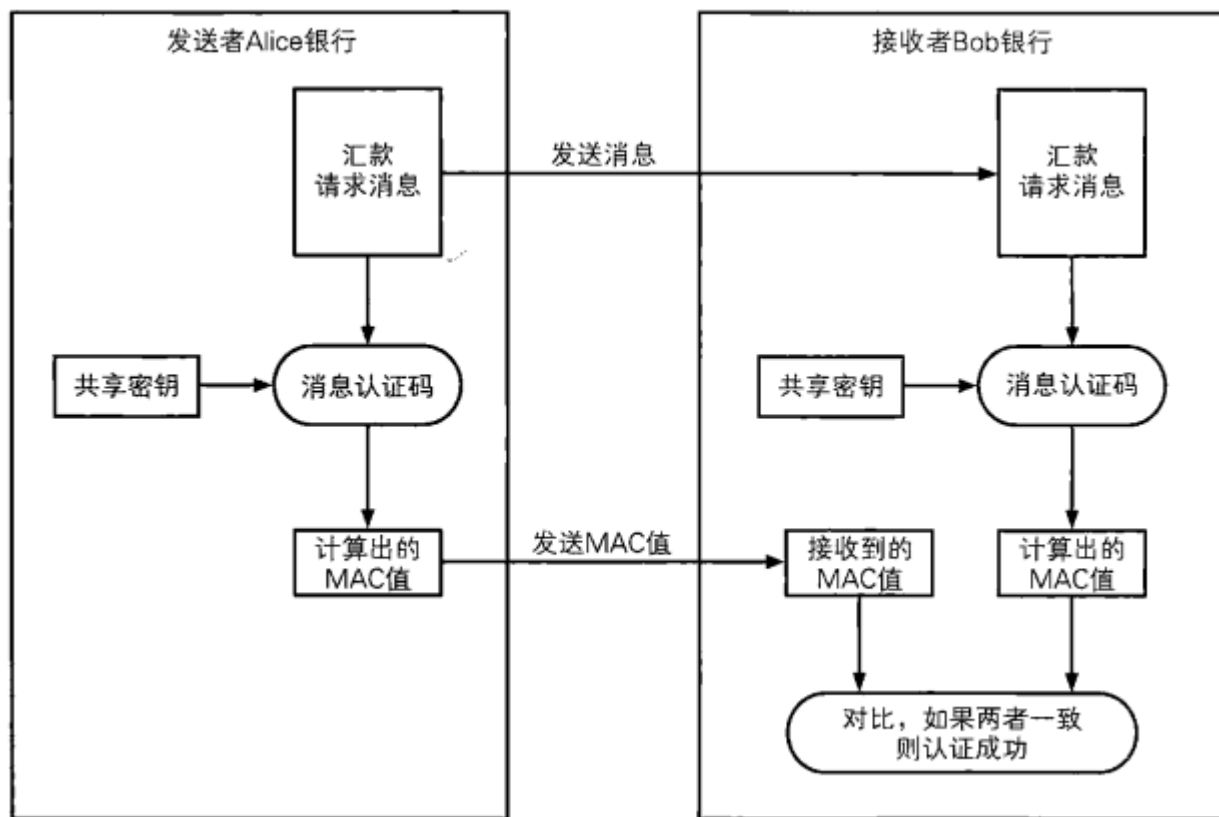
**\*\*单向散列函数与消息认证码的比较\*\***



## 6.2 消息认证码的使用步骤

#

我们还是以Alice银行和Bob银行的故事为例，来讲解一下消息认证码的使用步骤：



1. 发送者Alice与接收者Bob事先共享密钥。
2. 发送者Alice根据汇款请求消息计算MAC值（使用共享密钥）。
3. 发送者Alice将汇款请求消息和MAC值两者发送给接收者Bob。
4. 接收者Bob根据接收到的汇款请求消息计算MAC值（使用共享密钥）。
5. 接收者Bob将自己计算的MAC值与从Alice处收到的MAC值进行对比。
6. 如果两个MAC值一致，则接收者Bob就可以断定汇款请求的确来自Alice（认证成功）；如果不一致，则可以断定消息不是来自Alice（认证失败）。

## 6.3 HMAC

#

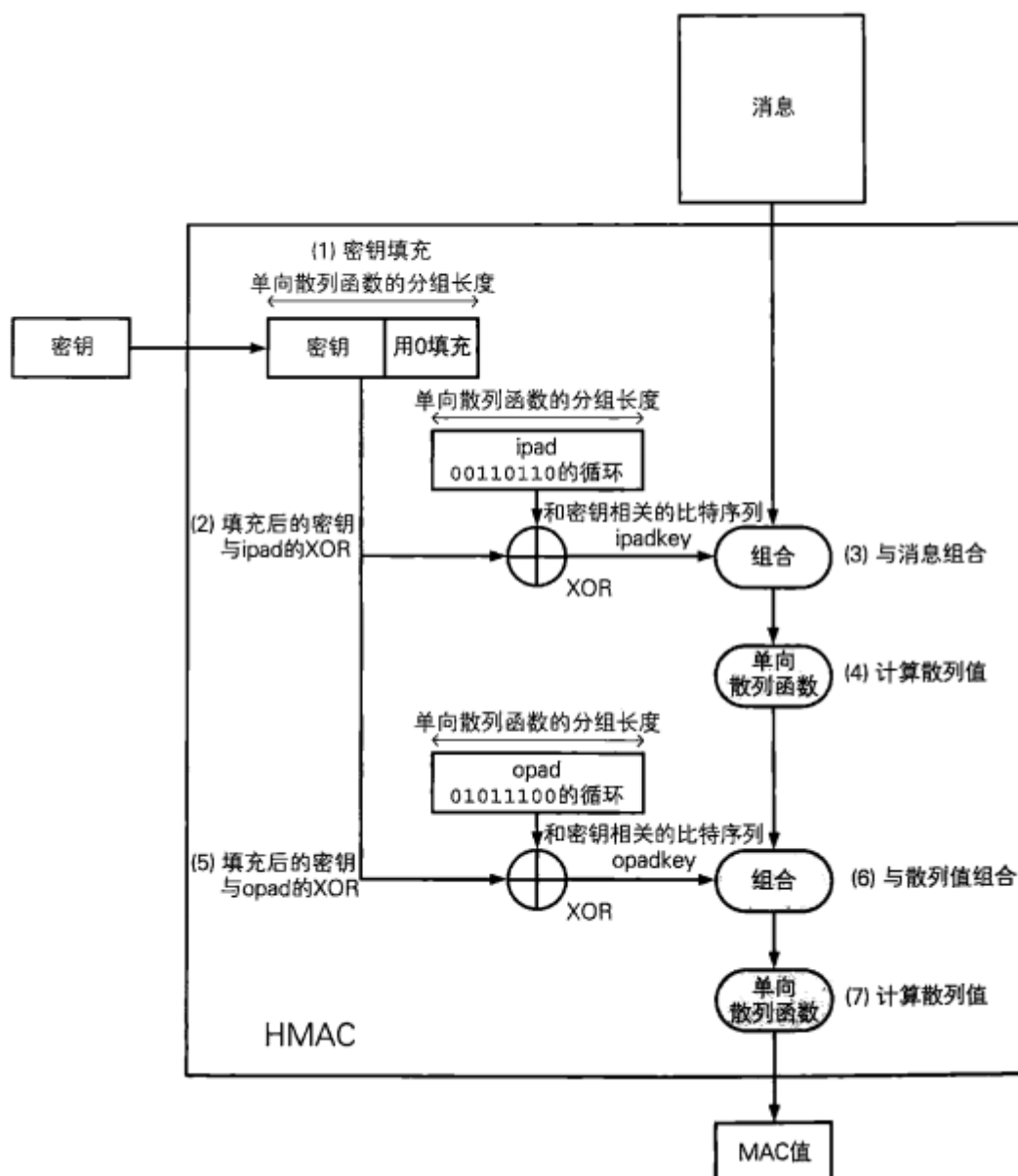
### 6.3.1 HMAC介绍

HMAC是一种使用单向散列函数来构造消息认证码的方法（RFC2104），其中HMAC的H就是Hash的意思。

HMAC中所使用的单向散列函数并不仅限于一种，任何高强度的单向散列函数都可以被用于HMAC,如果将来设计出新的单向散列函数，也同样可以使用。

使用SHA-1、MD5、RIPEMD-160所构造的HMAC，分别称为HMAC-SHA-1、HMAC-MD5和HMAC-RIPEMD。





通过上述流程我们可以看出，最后得到的MAC值，一定是一个和输入的消息以及密钥都相关的长度固定的比特序列。

### 6.3.2 Go中对HMAC的使用

需要使用的包

```
1 import "crypto/hmac"
```

使用的函数

```
1 func New(h func() hash.Hash, key []byte) hash.Hash
2 func Equal(mac1, mac2 []byte) bool
```

#### 1. hamc.New 函数

- 参数1: 创建一个新的使用哈希校验算法的hash.Hash接口, 如:
  - md5.New()

- sha1.New()
  - sha256.New()
  - 参数2: 使用的密钥
  - 返回值: 通过该哈希接口添加数据和计算消息认证码
    - 添加数据: Write(p []byte) (n int, err error)
    - 计算结果: Sum(b []byte) []byte
2. hmac.Equal 函数
- 比较两个MAC是否相同

生成消息认证码:

```

1 // 生成消息认证码
2 func GenerateHMAC(src, key []byte) []byte {
3     // 1. 创建一个底层采用sha256算法的 hash.Hash 接口
4     myHmac := hmac.New(sha256.New, key)
5     // 2. 添加测试数据
6     myHmac.Write(src)
7     // 3. 计算结果
8     result := myHmac.Sum(nil)
9
10    return result
11 }
```

重要函数说明

#### 1. 创建一个底层采用哈希算法的 hash.Hash 接口

```

1 函数对应的包: "crypto/hmac"
2 func New(h func() hash.Hash, key []byte) hash.Hash
3     - 参数 h: 函数指针, 返回值为hash.Hash, 可以使用哈希算法对应的New方法, 如:
4         -- md5.New
5         -- sha1.New
6         -- sha256.New
7         -- sha256.New224
8         -- sha512.New
9         -- sha512.New384
10    - 参数 key: 和数据进行混合运算使用的密钥
11    - 返回值: hash.Hash 接口
```

验证消息认证码

```

1 func VerifyHMAC(res, src, key []byte) bool {
2
3     // 1. 创建一个底层采用sha256算法的 hash.Hash 接口
4     myHmac := hmac.New(sha256.New, key)
5     // 2. 添加测试数据
6     myHmac.Write(src)
7     // 3. 计算结果
8     result := myHmac.Sum(nil)
9     // 4. 比较结果
10    return hmac.Equal(res, result)
11 }
```

重要函数说明:

#### 1. 比较两个MAC是否相同

```
1  函数对应的包: "crypto/hmac"
2  func Equal(mac1, mac2 []byte) bool
3      - 参数 mac1, mac2: 通过哈希算法计算得到的消息认证码
4      - 返回值: 如果mac1==mac2, 返回 true; 否则, 返回 false
```

测试代码

```
1  func HMacTest() {
2      key := []byte("我是消息认证码密钥")
3      src := []byte("我是消息认证码测试数据")
4      result := GenerateHMAC(src, key)
5      final := VerifyHMAC(result, src, key)
6      if final {
7          fmt.Println("消息认证码认证成功!!!")
8      } else {
9          fmt.Println("消息认证码认证失败 .....")
10     }
11 }
```

## 6.3 消息认证码的密钥配送问题

#

在消息认证码中, 需要发送者和接收者之间共享密钥, 而这个密钥不能被主动攻击者Mallory获取。如果这个密钥落入Mallory手中, 则Mallory也可以计算出MAC值, 从而就能够自由地进行篡改和伪装攻击, 这样一来消息认证码就无法发挥作用了。

发送者和接收者需要共享密钥, 这一点和我们介绍的对称加密很相似。实际上, 对称加密的密钥配送问题在消息认证码中也同样会发生。关于密钥的配送后边章节会介绍如何使用非对称加密的方式进行解决。

## 6.4 消息认证码无法解决的问题

#

假设发送者Alice要向接收者Bob发送消息, 如果使用了消息认证码, 接收者Bob就能够断定自己收到的消息与发送者Alice所发出的消息是一致的, 这是因为消息中的MAC值只有用Alice和Bob之间共享的密钥才能够计算出来, 即便主动攻击者Mallory篡改消息, 或者伪装成Alice发送消息, Bob也能够识别出消息的篡改和伪装。

但是, 消息认证码也不能解决所有的问题, 例如“对第三方证明”和“防止否认”, 这两个问题就无法通过消息认证码来解决。下面我们来逐一解释一下。

### 6.4.1 对第三方证明

假设Bob在接收了来自Alice的消息之后, 想要向第三方验证者Victor证明这条消息的确是Alice发送的, 但是用消息认证码无法进行这样的证明, 这是为什么呢?

首先, Victor要校验MAC值, 就需要知道Alice和Bob之间共享的密钥。

假设Bob相信Victor, 同意将密钥告诉Victor,即便如此, Victor也无法判断这条消息是由Alice发送的, 因为Victor可以认为: “即使MAC值是正确的, 发送这条消息的人也不一定是Alice, 还有可能是Bob。”

能够计算出正确MAC值的人只有Alice和Bob, 在他们两个人之间进行通信时, 可以断定是对方计算了MAC值, 这是因为共享这个密钥的双方之中, 有一方就是自己。然而, 对于第三方Victor、Alice或Bob却无法证明是对方计算了MAC值, 而不是自己。

使用第7章中将要介绍的数字签名就可以实现对第三方的证明。

### 6.4.2 防止否认

假设Bob收到了包含MAC值的消息, 这个MAC值是用Alice和Bob共享的密钥计算出来的, 因此Bob能够判断这条消息的确来自Alice。

但是, 上面我们讲过, Bob无法向验证者Victor证明这一点, 也就是说, 发送者Alice可以向Victor声称: “我没有向Bob发送过这条消息。”这样的行为就称为否认 (repudiation) 。

Alice可以说“这条消息是Bob自己编的吧”, “说不定Bob的密钥被主动攻击者Mallory给盗取了, 我的密钥可是妥善保管着呢” 等。说白了, 就是Alice和Bob吵起来了。

即便Bob拿MAC值来举证, Victor也无法判断Alice和Bob谁的主张才是正确的, 也就是说, **用消息认证码无法防止否认 (nonrepudiatlon)** 。

## 6.5 总结

#

消息认证码是对消息进行认证并确认其完整性的技术。通过使用发送者和接收者之间共享的密钥, 就可以识别出是否存在伪装和篡改行为。

消息认证码可以使用单向散列函数HMAC, 对称加密也可以实现, 这里不再进行介绍。

消息认证码中, 由于发送者和接收者共享相同的密钥, 因此会产生无法对第三方证明以及无法防止否认等问题。在下一章中, 我们将介绍能够解决这些问题的数字签名。

## 7. 数字签名

1 "数字签名 --- 消息到底是谁写的"

本章中我们将学习数字签名的相关知识。数字签名是一种将相当于现实世界中的盖章、签字的功能在计算机世界中进行实现的技术。使用数字签名可以识别篡改和伪装, 还可以防止否认。

### 7.1 从消息认证到数字签名

#

- 消息认证码的局限性

通过使用第6章中介绍的消息认证码, 我们可以识别消息是否被篡改或者发送者身份是否被伪装, 也就是可以校验消息的完整性, 还可以对消息进行认证。然而, 比如在出具借条的场景中却无法使用消息认证码, 因为消息认证码无法防止否认。

消息认证码之所以无法防止否认，是因为消息认证码需要在发送者Alice和接收者Bob两者之间共享同一个密钥。正是因为密钥是共享的，所以能够使用消息认证码计算出正确MAC值的并不只有发送者Alice，接收者Bob也可以计算出正确的MAC值。由于Alice和Bob双方都能够计算出正确的MAC值，因此对于第三方来说，我们无法证明这条消息的确是由Alice生成的。

## • 通过数字签名解决问题

下面请大家开动一下脑筋。假设发送者Alice和接收者Bob不需要共享一个密钥，也就是说，Alice和Bob各自使用不同的密钥。

我们假设Alice使用的密钥是一个只有Alice自己才知道的私钥。当Alice发送消息时，她用私钥生成一个“签名”。相对地，接收者Bob则使用一个和Alice不同的密钥对签名进行验证。使用Bob的密钥无法根据消息生成签名，但是用Bob的密钥却可以对Alice所计算的签名进行验证，也就是说可以知道这个签名是否是通过Alice的密钥计算出来的。如果真有这么一种方法的话，那么不管是识别篡改、伪装还是防止否认就都可以实现了吧？

实际上，这种看似很神奇的技术早就已经问世了，这就是数字签名（digital signature）。

## 7.2 签名的生成和验证

#

让我们来稍微整理一下。

在数字签名技术中，出现了下面两种行为：

- 生成消息签名的行为
- 验证消息签名的行为

**生成消息签名**这一行为是由消息的发送者Alice来完成的，也称为“对消息签名”。生成签名就是根据消息内容计算数字签名的值，这个行为意味着“我认可该消息的内容”。

**验证数字签名**这一行为一般是由消息的接收者Bob来完成的，但也可以由需要验证消息的第三方来完成，这里的第三方我们暂且将其命名为验证者Victor。验证签名就是检查该消息的签名是否真的属于Alice，验证的结果可以是成功或者失败，成功就意味着这个签名是属于Alice的，失败则意味着这个签名不是属于Alice的。

在数字签名中，生成签名和验证签名这两个行为需要使用各自专用的密钥来完成。

Alice使用“签名密钥”来生成消息的签名，而Bob和Victor则使用“验证密钥”来验证消息的签名。数字签名对签名密钥和验证密钥进行了区分，使用验证密钥是无法生成签名的。这一点非常重要。此外，**签名密钥只能由签名的人持有，而验证密钥则是任何需要验证签名的人都可以持有。**

刚才讲的这部分内容，是不是觉得似曾相识呢？

没错，这就是我们讲过的非对称加密。公钥密码和上面讲的数字签名的结构非常相似。在非对称加密中，密钥分为加密密钥和解密密钥，用加密密钥无法进行解密。此外，解密密钥只能由需要解密的人持有，而加密密钥则是任何需要加密的人都可以持有。你看，数字签名和非对称加密是不是很像呢？

实际上，数字签名和非对称加密有着非常紧密的联系，简而言之，数字签名就是通过将非对称加密“反过来用”而实现的。下面我们来将密钥的使用方式总结成一张表：

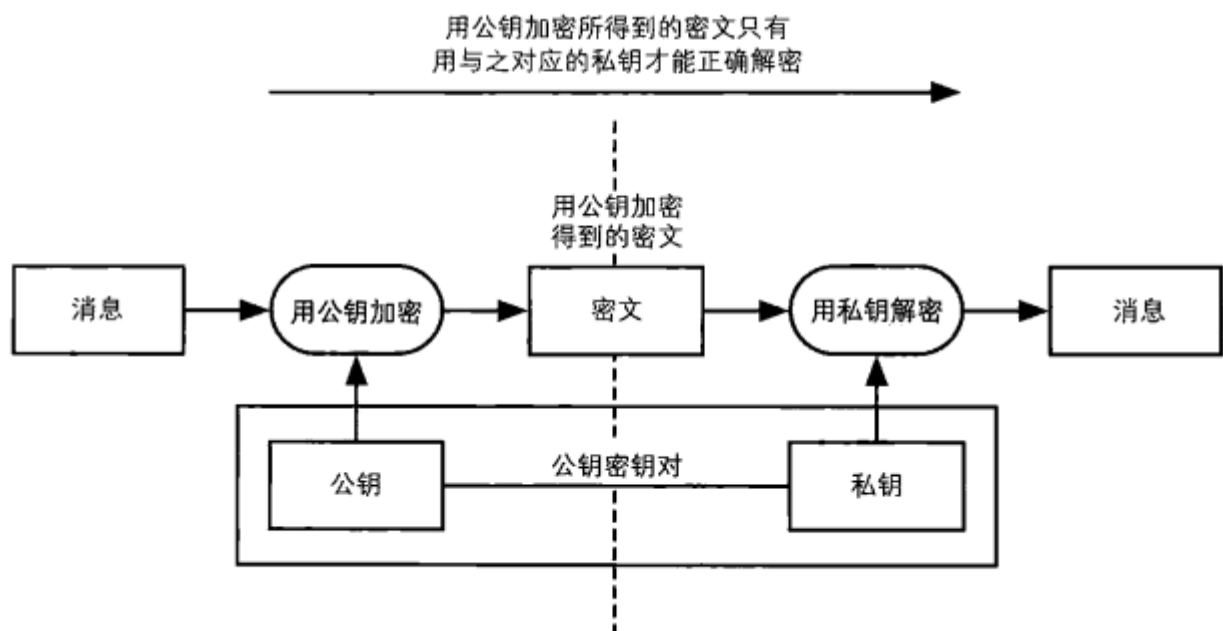
	私钥	公钥
非对称加密	接收者解密时使用	发送者加密时使用
数字签名	签名者生成签名时使用	验证者验证签名时使用
谁持有秘钥?	个人持有	只要需要, 任何人都可以持有

## 7.3 非对称加密和数字签名

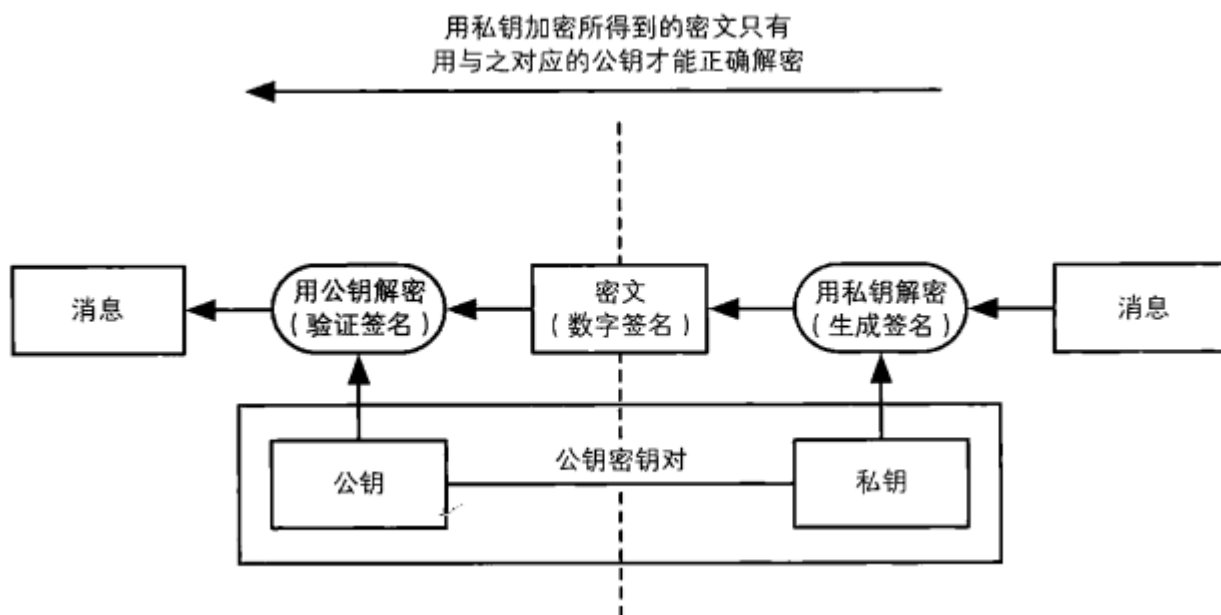
#

下面我们再详细讲一讲非对称加密与数字签名之间的关系。

要实现数字签名, 我们可以使用第4章中介绍的非对称加密。非对称加密包括一个由公钥和私钥组成的密钥对, 其中公钥用于加密, 私钥用于解密。



数字签名中也同样会使用公钥和私钥组成的密钥对, 不过这两个密钥的用法和非对称加密是相反的, 即用私钥加密相当于生成签名, 而用公钥解密则相当于验证签名。请大家通过比较两张图示来理解一下“反过来用”到底是什么样的情形。

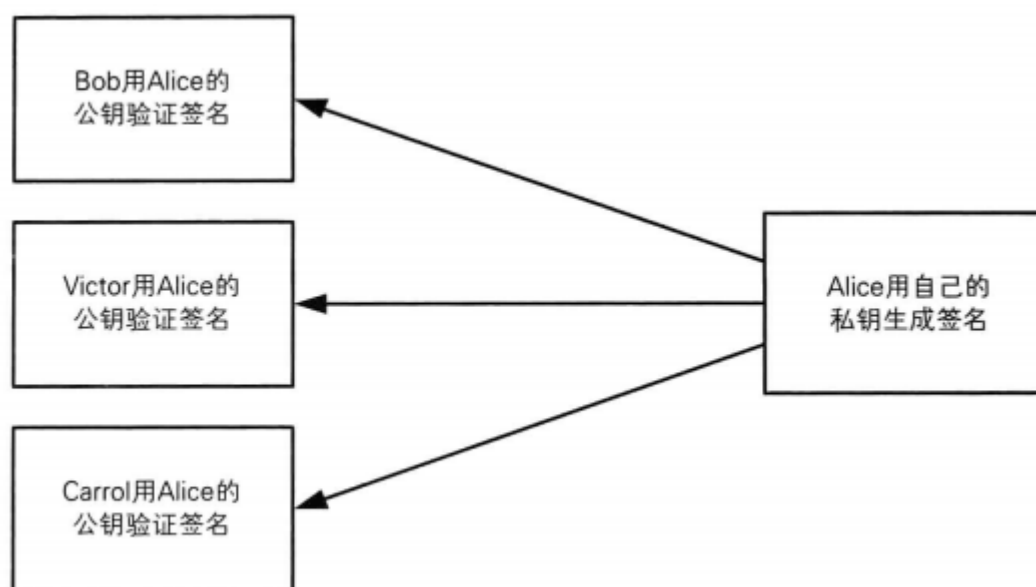


那么为什么加密相当于生成签名，而解密相当于验证签名呢？要理解这个问题，我们需要回想一下非对称加密中讲过的知识，即组成密钥对的两个密钥之间存在严密的数学关系，它们是一对无法拆散的伙伴。

用公钥加密所得到的密文，只能用与该公钥配对的私钥才能解密：同样地，用私钥加密所得到的密文，也只能用与该私钥配对的公钥才能解密。也就是说，如果用某个公钥成功解密了密文，那么就能够证明这段密文是用与该公钥配对的私钥进行加密所得到的。

用私钥进行加密这一行为只能由持有私钥的人完成，正是基于这一事实，我们才可以将用私钥加密的密文作为签名来对待。

由于公钥是对外公开的，因此任何人都能够用公钥进行解密，这就产生了一个很大的好处，即任何人都能够对签名进行验证。



## 7.3 数字签名的方法

#

下面我们来具体介绍两种生成和验证数字签名的方法。

- 直接对消息签名的方法
- 对消息的散列值签名的方法

直接对消息签名的方法比较容易理解，但实际上并不会使用；对消息的散列值签名的方法稍微复杂一点，但实际中我们一般都使用这种方法。

使用直接对消息签名的方法，需要对整个消息进行加密，非常耗时，这是因为非对称加密算法本来就非常慢。那么，我们能不能生成一条很短的数据来代替消息本身呢？这就是单向散列函数。

于是我们不必再对整个消息进行加密（即对消息签名），而是只要先用单向散列函数求出消息的散列值，然后再将散列值进行加密（对散列值签名）就可以了。无论消息有多长，散列值永远都是这么短，因此对其进行加密（签名）是非常轻松的。

**(1) Alice用单向散列函数计算消息的散列值。**

**(2) Alice用自己的私钥对散列值进行加密。**

- 1 用私钥加密散列值得到的密文就是Alice对这条散列值的签名，由于只有Alice才持有自己的私钥因此，
- 2
- 3 除了Alice以外，其他人是无法生成相同的签名（密文）的。

**(3) Alice将消息和签名发送给Bob。**

**(4) Bob用Alice的公钥对收到的签名进行解密。**

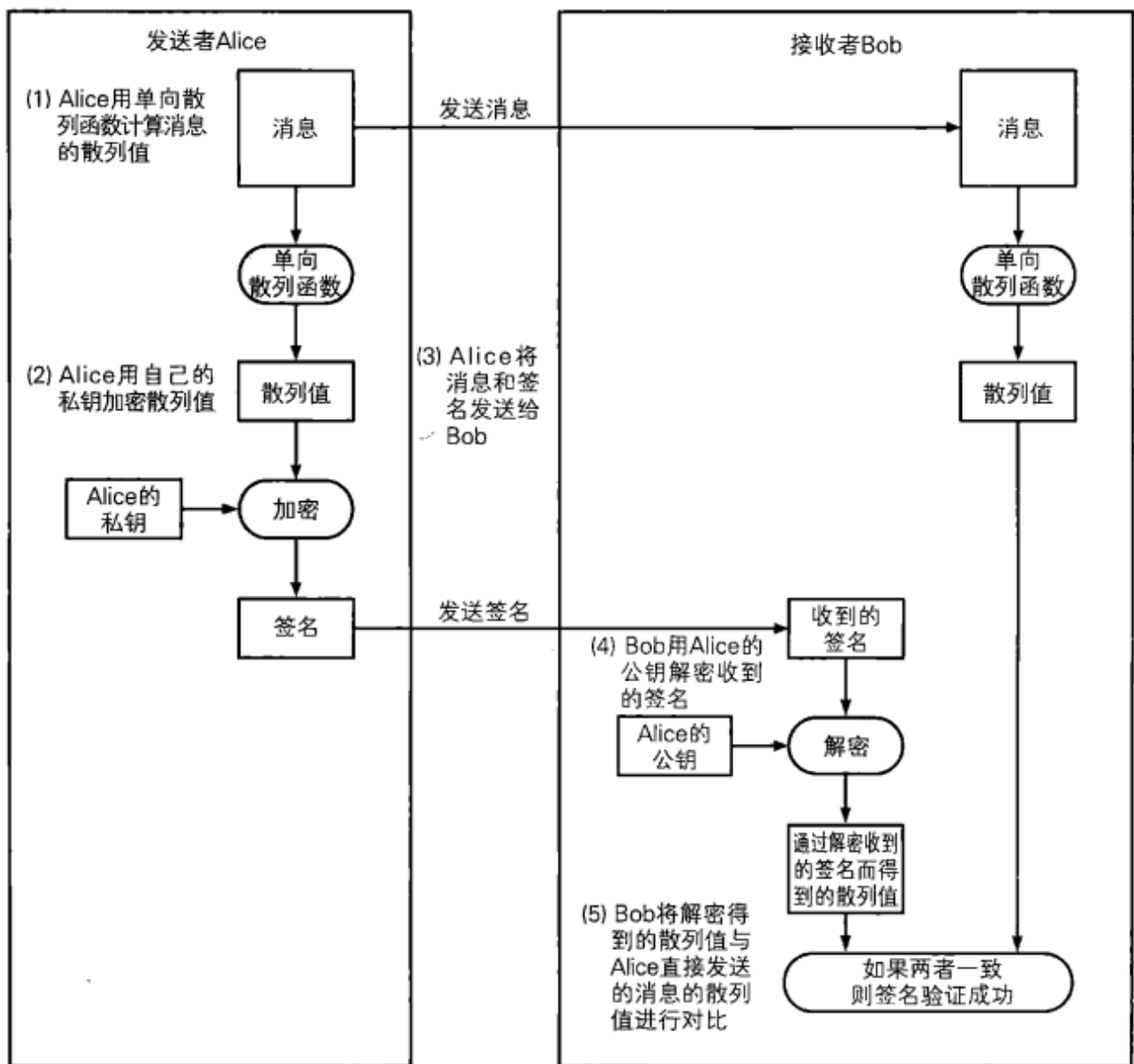
- 1 如果收到的签名确实是用Alice的私钥进行加密而得到的密文（签名），那么用Alice的公钥应该能够正确
- 2
- 3 解密，解密的结果应该等于消息的散列值。如果收到的签名不是用Alice的私钥进行加密而得到的密文，
- 4
- 5 那么就无法用Alice的公钥正确解密（解密后得到的数据看起来是随机的）。

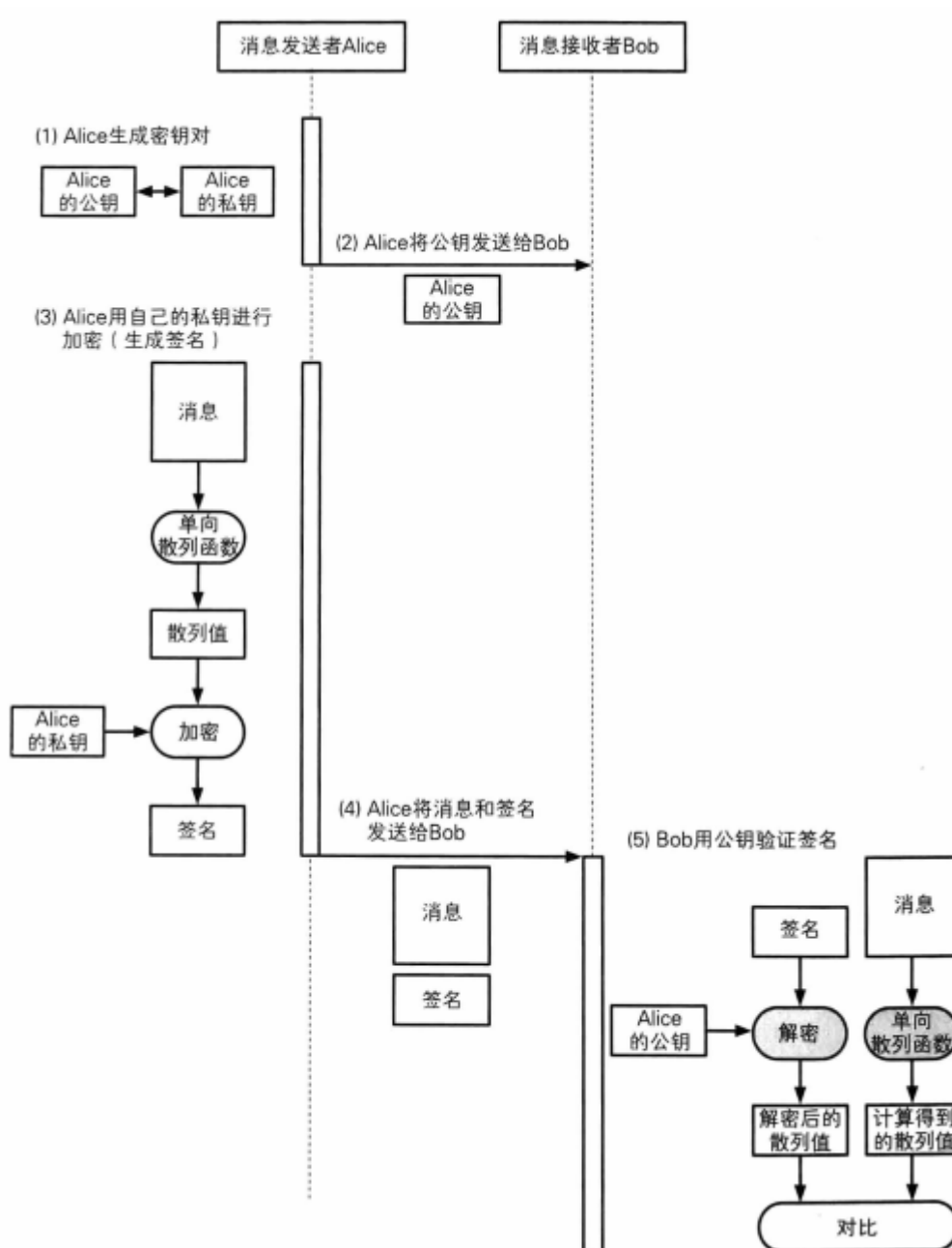
**(5) Bob将签名解密后得到的散列值与Alice直接发送的消息的散列值进行对比。**

- 1 如果两者一致，则签名验证成功；如果两者不一致，则签名验证失败。

我们将数字签名中生成签名和验证签名的过程整理成一张时间流程图。







## 7.4 通过RSA实现数字签名

#

前边章节已经介绍过了如何通过自己编写的go代码生成非对称加密算法RSA的公钥和私钥文件, 假设公钥文件的文件名为 `public.pem`, 私钥文件对应的文件名为 `private.pem`。

### 7.4.1 生成数字签名

```
1 func SignatureRSA() ([]byte, error) {
2     // 1. 从私钥文件中读生成的私钥内容
3     fp, err := os.Open("private.pem")
4     if err != nil {
5         return nil, errors.New("打开私钥文件 - private.pem 失败!!!")
6     }
7     // 2. 读文件内容
8     fileInfo, _ := fp.Stat()
```

```

9      all := make([]byte, fileInfo.Size())
10     _, err = fp.Read(all)
11     if err != nil {
12         return nil, errors.New("读文件内容失败!!!")
13     }
14     fmt.Println("文件内容: ", string(all))
15     // 3. 关闭文件
16     defer fp.Close()
17     // 4. 将数据解析成pem格式的数据块
18     block, _ := pem.Decode(all)
19     // 5. 解析pem数据块, 得到私钥
20     priv_Key, err := x509.ParsePKCS1PrivateKey(block.Bytes)
21     if err != nil {
22         return nil, errors.New("解析私钥失败!!!")
23     }
24
25     // 待签名数据
26     myData := []byte("渡远荆门外, 来从楚国游。山随平野尽, 江入大荒流。月下飞天境, 云生结海楼。仍怜故乡水, 万里送行舟。")
27     // 6. 将数据通过哈希函数生成信息摘要
28     myHash := sha256.New()
29     myHash.Write(myData)
30     result := myHash.Sum(nil)
31     // 7. 生成签名
32     mySignature, err := rsa.SignPKCS1v15(rand.Reader, priv_Key, crypto.SHA256, result)
33     if err != nil {
34         return nil, errors.New("生成签名失败!!!")
35     }
36
37     return mySignature, nil
38 }

```

## 7.4.2 验证数字签名

```

1  func VerifyRSA(src []byte) (error){
2      // 1. 从密钥文件中读生成的密钥内容
3      fp, err := os.Open("public.pem")
4      if err != nil {
5          return errors.New("打开公钥文件 - public.pem 失败!!!")
6      }
7      // 2. 读文件内容
8      fileInfo, _ := fp.Stat()
9      all := make([]byte, fileInfo.Size())
10     num, err := fp.Read(all)
11     if err != nil {
12         return errors.New("读文件内容失败!!!")
13     }
14     fmt.Println("文件大小: ", num)
15     // 3. 关闭文件
16     defer fp.Close()
17     // 4. 将公钥数据解析为pem格式的数据块
18     block, _ := pem.Decode(all)

```

```

19 // 5. 将公钥从pem数据块中提取出来
20 pubInterface, err := x509.ParsePKIXPublicKey(block.Bytes)
21 if err != nil {
22     return errors.New("解析公钥失败!!!")
23 }
24 // 6. 公钥接口转换为公钥对象
25 pubKey := pubInterface.(*rsa.PublicKey)
26 // 待认证数据
27 myData := []byte("渡远荆门外，来从楚国游。山随平野尽，江入大荒流。月下飞天境，云生结海楼。仍怜故乡水，万里送行舟。")
28 // 7. 将数据通过哈希函数生成信息摘要
29 myHash := sha256.New()
30 myHash.Write(myData)
31 result := myHash.Sum(nil)
32
33 // 7. 数据认证
34 err = rsa.VerifyPKCS1v15(pubKey, crypto.SHA256, result, src)
35 if err != nil {
36     return err
37 }
38
39 fmt.Println("数字签名验证成功，恭喜o(*￣▽￣*)o恭喜")
40 return nil
41 }

```

## 7.5 使用椭圆曲线实现数字签名

#

## 7.6 数字签名无法解决的问题

#

用数字签名既可以识别出篡改和伪装，还可以防止否认。也就是说，我们同时实现了确认消息的完整性、进行认证以及否认防止。现代社会中的计算机通信从这一技术中获益匪浅。

然而，**要正确使用数字签名，有一个大前提，那是用于验证签名的公钥必须属于真正的发送者。**即便数字签名算法再强大，如果你得到的公钥是伪造的，那么数字签名也会完全失效。

现在我们发现自己陷入了一个死循环——数字签名是用来识别消息篡改、伪装以及否认的，但是为此我们又必须从没有被伪装的发送者得到没有被篡改的公钥才行。

为了能够确认自己得到的公钥是否合法，我们需要使用证书。所谓证书，就是将公钥当作一条消息，由一个可信的第三方对其签名后所得到的公钥。

当然，这样的方法只是把问题转移了而已。为了对证书上施加的数字签名进行验证，我们必定需要另一个公钥，那么如何才能构筑一个可信的数字签名链条呢？又由谁来颁发可信的证书呢？到这一步，我们就已经踏入了社会学的领域。我们需要让公钥以及数字签名技术成为一种社会性的基础设施，即公钥基础设施（Public Key Infrastructure），简称PKI关于证书和PKI我们将在第8章中介绍。

## 8. 证书

要开车得先考驾照。驾照上面记有本人的照片、姓名、出生日期等个人信息，以及有效期、准驾车辆的类型等信息，并由公安局在上面盖章。我们只要看到驾照，就可以知道公安局认定此人具有驾驶车辆的资格。

公钥证书 (Public-Key Certificate, PKC) 其实和驾照很相似，里面记有姓名、组织、邮箱地址等个人信息，以及属于此人的公钥，并由认证机构 (Certification Authority、Certifying Authority, CA) 施加数字签名。只要看到公钥证书，我们就可以知道认证机构认定该公钥的确属于此人。公钥证书也简称为证书 (certificate)。

可能很多人都没听说过认证机构，认证机构就是能够认定“公钥确实属于此人”，并能够生成数字签名的个人或组织。认证机构中有国际性组织和政府所设立的组织，也有通过提供认证服务来盈利的一般企业，此外个人也可以成立认证机构。

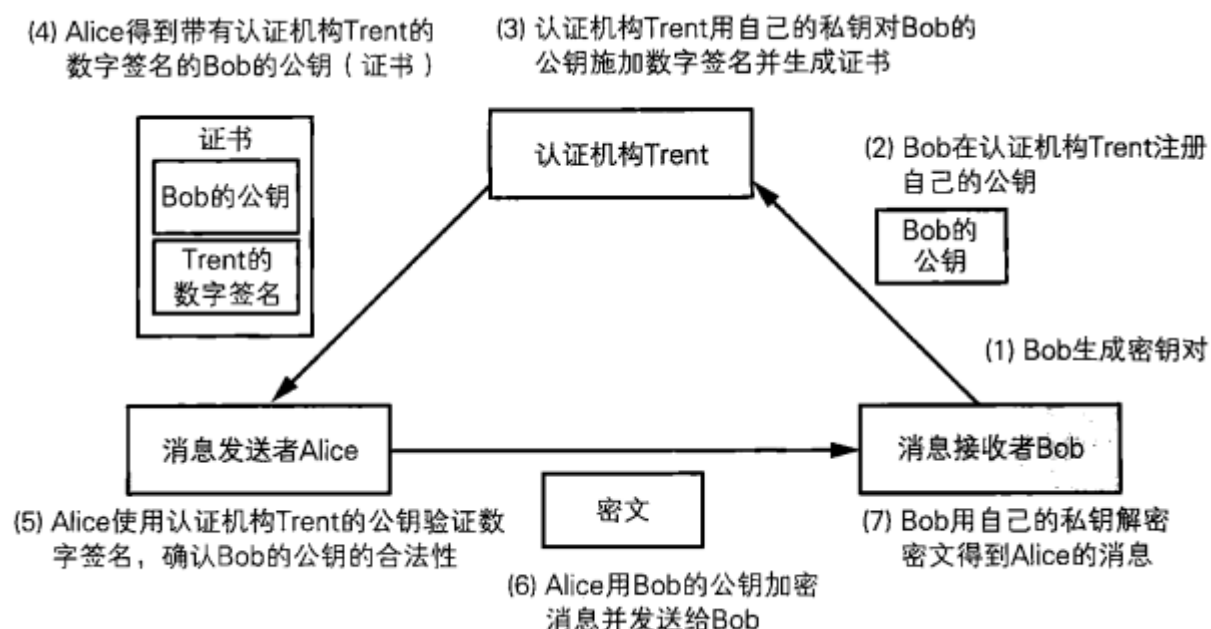
## 8.1 证书的应用场景

#

下面我们来通过证书的代表性应用场景来理解证书的作用。

下图展示了Alice向Bob发送密文的场景，在生成密文时所使用的Bob的公钥是通过认证机构获取的。

认证机构必须是可信的，对于“可信的第三方”，下图中会使用Trent这个名字，这个词是从trust（信任）一词演变而来的。



下面让我们对照着上图来看一看这些步骤具体都做了些什么。

### 1. Bob生成密钥对

要使用公钥密码进行通信，首先需要生成密钥对。Bob生成了一对公钥和私钥，并将私钥自行妥善保管。在这里，密钥对是由Bob自己生成的，也可以由认证机构代为生成。

### 2. Bob在认证机构Trent注册自己的公钥

- 在这里Bob则将公钥发送给了认证机构Trent，这是因为Bob需要请认证机构Trent对他的公钥加上数字签名（也就是生成证书）。

- Trent收到Bob的公钥后，会确认所收到的公钥是否为Bob本人所有（参见专栏：身份确认和认证业务准则）

## 专栏：身份确认和认证业务准则

认证机构确认“本人”身份的方法和认证机构的认证业务准则（Certificate Practice Statement, CPS，的内容有关。如果认证机构提供的是测试用的服务，那么可能完全不会进行任何身份确认。如果是政府部门运营的认证机构，可能就需要根据法律规定来进行身份确认。如果是企业面向内部设立的认证机构，那就可能会给部门负责人打电话直接确认。

例如，VeriSign的认证业务准则中将身份确认分为Class1 ~ 3共三个等级

- Class1：通过向邮箱发送件来确认本人身份
- Class2：通过第三方数据库来确认本人身份
- Class3：通过当面认证和身份证明来确认本人身份

等级越高，身份确认越严格。

### 3. 认证机构Trent用自己的私钥对Bob的公钥施加数字签名并生成证书

Trent对Bob的公钥加上数字签名。为了生成数字签名，需要Trent自身的私钥，因此Trent需要事先生成好密钥对。

### 4. Alice得到带有认证机构Trent的数字签名的Bob的公钥（证书）

现在Alice需要向Bob发送密文，因此她从Trent处获取证书。证书中包含了Bob的公钥。

### 5. Alice使用认证机构Trent的公钥验证数字签名，确认Bob的公钥的合法性

Alice使用认证机构Trent的公钥对证书中的数字签名进行验证。如果验证成功，就相当于确认了证书中所包含的公钥的确是属于Bob的。到这里，Alice就得到了合法的Bob的公钥。

### 6. Alice用Bob的公钥加密消息并发送给Bob

Alice用Bob的公钥加密要发送的消息，并将消息发送给Bob。

### 7. Bob用自己的私钥解密密文得到Alice的消息

Bob收到Alice发送的密文，然后用自己的私钥解密，这样就能够看到Alice的消息了。

上面就是利用认证机构Trent进行公钥密码通信的流程。其中1、2、3这几个步骤仅在注册新公钥时才会进行，并不是每次通信都需要。此外，步骤4仅在Alice第一次用公钥密码向Bob发送消息时才需要进行，只要Alice将Bob的公钥保存在电脑中，在以后的通信中就可以直接使用了。

## 8.2 证书标准规范X.509

#

证书是由认证机构颁发的，使用者需要对证书进行验证，因此如果证书的格式千奇百怪那就不方便了。于是，人们制定了证书的标准规范，其中使用最广泛的是由ITU（International Telecommunication Union，国际电信联盟）和ISO（International Organization for Standardization，国际标准化组织）制定的X.509规范。很多应用程序都支持X.509并将其作为证书生成和交换的标准规范。

X.509是一种非常通用的证书格式。所有的证书都符合ITU-T X.509国际标准，因此(理论上)为一种应用创建的证书可以用于任何其他符合X.509标准的应用。X.509证书的结构是用ASN.1 (Abstract Syntax Notation One) 进行描述数据结构，并使用ASN.1语法进行编码。

在一份证书中，必须证明公钥及其所有者的姓名是一致的。对X.509证书来说，认证者总是 **CA** 或由CA指定的人，一份X.509证书是一些标准字段的集合，这些字段包含有关用户或设备及其相应公钥的信息。X.509标准定义了证书中应该包含哪些信息，并描述了这些信息是如何编码的(即数据格式)

一般来说，一个数字证书内容可能包括基本数据（版本、序列号）、所签名对象信息（签名算法类型、签发者信息、有效期、被签发人、签发的公开密钥）、CA的数字签名，等等。

### 8.2.1 证书规范

前使用最广泛的标准为ITU和ISO联合制定的X.509的 v3版本规范 (RFC5280)，其中定义了如下证书信息域：

- **版本号(Version Number)**：规范的版本号，目前为版本3，值为0x2；
- **序列号 (Serial Number)**：由CA维护的为它所发的每个证书分配的一的列号，用来追踪和撤销证书。只要拥有签发者信息和序列号，就可以唯一标识一个证书，最大不能超过20个字节；
- **签名算法 (Signature Algorithm)**：数字签名所采用的算法，如：
  - sha256-with-RSA-Encryption
  - ccdsa-with-SHA256；
- **颁发者 (Issuer)**：发证书单位的标识信息，如 "C=CN, ST=Beijing, L=Beijing, O=org.example.com, CN=ca.org. example.com"；
- **有效期(Validity)**: 证书的有效期限，包括起止时间。
- **主体(Subject)**: 证书拥有者的标识信息 (Distinguished Name)，如: "C=CN, ST=Beijing, L=Beijing, CN=person.org.example.com"；
- **主体的公钥信息(SubJect Public Key Info)**：所保护的公钥相关的信息：
  - 公钥算法 (Public Key Algorithm) 公钥采用的算法；
  - 主体公钥 (Subject Unique Identifier)：公钥的内容。
- **颁发者唯一号 (Issuer Unique Identifier)**：代表颁发者的唯一信息，仅2、3版本支持，可选；
- **主体唯一号 (Subject Unique Identifier)**：代表拥有证书实体的唯一信息，仅2、3版本支持，可选；
- **扩展 (Extensions, 可选)**：可选的一些扩展。中可能包括：
  - Subject Key Identifier：实体的密钥标识符，区分实体的多对密钥；
  - Basic Constraints：一指明是否属于CA；
  - Authority Key Identifier：证书颁发者的公钥标识符；
  - CRL Distribution Points: 撤销文件的颁发地址；
  - Key Usage：证书的用途或功能信息。

此外，证书的颁发者还需要对证书内容利用自己的私钥添加签名，以防止别人对证书的内容进行篡改。

### 8.2.2 证书格式

X.509规范中一般推荐使用PEM(Privacy Enhanced Mail) 格式来存储证书相关的文件。证书文件的文件名后缀一般为 .crt 或 .cer。对应私钥文件的文件名后缀一般为 .key。证书请求文件的文件名后缀为 .csr。有时候也统一用 pem作为文件名后缀。

PEM格式采用文本方式进行存储。一般包括首尾标记和内容块，内容块采用Base64进行编码。

**编码格式总结:**



- X.509 DER(Distinguished Encoding Rules)编码, 后缀为: .der .cer .crt
- X.509 BASE64编码(PEM格式), 后缀为: .pem .cer .crt

例如, 一个PEM格式 (base64编码) 的示例证书文件内容如下所示:

```

1  -----BEGIN CERTIFICATE-----
2  MIIDYjCCArKgAwIBAgIQdZfkKrISoINLporOrZLXPTANBgkqhkiG9w0BAQsFADBN
3  MSswKQYDVQQLDCJDcmVhdGVkIGJ5IGh0dHA6Ly93d3cuZm1kZGx1c2IuY29tMRUw
4  EwYDVQKDAxET190T1RfVfJVU1QxITAfBgNVBAMMERPX05PVF9UU1VTVF9GaWRk
5  bGVyUm9vdDAeFw0xNzA0MTEwNjQ0MzhaFw0yMzA0MTEwNjQ0MzhaMFoxKzApBgNV
6  BAsMIkNyZWZ0ZWQgYnkgHR0cDovL3d3dy5maWRkbGVyMi5jb20xFTATBgNVBAoM
7  DERPX05PVF9UU1VTVDEUMBIGA1UEAwWLK15iYWlkS5jb20wggEiMA0GCSqGSIsb3
8  DQEBAAQAA4IBDwAwggEKAoIBAQDX0AM198jxwRoKgwWsd9oj5vI0and9v9SB9Ch1
9  gZEU6G9ZA0C7BucsBzJ2b10Mf6qq0Iee1DfeydfEKyTmBKTafgb2DoQE30HZjy0B
10 QTJrs0df5s636W5gJp4f7CUYYA/3e1nxr/+AuG44Id1si17TWodVKjsQhjh+bK6
11 8ukQZye11SgBeQ0ivzxXe0rhXzrocoeKZFmUxLkUpm+/mX1syDTdaCmQ6LT4KYYi
12 soKe4f+r2tLbUzPKxtk2F1v3ZL0jiRdzCOA27e5n88zdAFrCmMB4teG/azCSAH3g
13 Yb6vaAGA0nKyDLGunW51sSesWBpHceJnMfrhwxCjiv707JZtAgMBAAGjfb9MA4G
14 A1UdDwEB/wQEAwIESDATBgNVHSUEDDAKBggrBgEFBQCcDATAwBgNVHREEDzANGgsq
15 LmJhaWR1LmNvbTAFBgNVHSMEGDAWgBQ9UIffUQSuwWG0m+o74JffZJNadjAdBgNV
16 HQ4EFgQUQh8IksZqcMVmKrIibTHLbAgLRGgwdQYJKoZIhvcNAQELBQADggEBAC5Y
17 JndwXpm0W+9SUIQhAUSE9LZh+DzcSmlCWtBk+SKBwmAegbfNSf6CgCh0VY6iIhbn
18 GlszqgAOAqVMxAED1R/YJT0LAUXFw8KICsWdvE01xtHqhk1tCK1540tci60Wu+tz
19 1t8999GPbJskecbRDGRDSA/gQGZJuL0rnmIuz3macSVn6tH7NwdoNeN68Uj3Qyt5
20 orYv1IFm8t5522ga8ac1y90hK4R5HcvN71aIjMKrikgynK0E+g45QypHRIe/z0S
21 /1W/6rqTgfN60Wc0c15hPeJbTtkntB5Fqd0sfsnKkW6jPsKQ+z/+vZ5XqzdlFupQ
22 29F14ei8ZH19aLIHP5s=
23  -----END CERTIFICATE-----

```

证书中的解析出来的内容:

```

1  Certificate:
2      Data:
3          Version: 3 (0x2)
4          Serial Number:
5              10:e6:fc:62:b7:41:8a:d5:00:5e:45:b6
6          Signature Algorithm: sha256WithRSAEncryption
7          Issuer: C=BE, O=GlobalSign nv-sa, CN=GlobalSign Organization Validation CA-SHA256-
8  G2
9          Validity
10             Not Before: Nov 21 08:00:00 2016 GMT
11             Not After : Nov 22 07:59:59 2017 GMT
12             Subject: C=US, ST=California, L=San Francisco, O=Wikimedia Foundation, Inc.,
13             CN=*.wikipedia.org
14             Subject Public Key Info:
15                 Public Key Algorithm: id-ecPublicKey
16                 Public-Key: (256 bit)
17                 pub:
18                     04:c9:22:69:31:8a:d6:6c:ea:da:c3:7f:2c:ac:a5:
19                     af:c0:02:ea:81:cb:65:b9:fd:0c:6d:46:5b:c9:1e:
20                     ed:b2:ac:2a:1b:4a:ec:80:7b:e7:1a:51:e0:df:f7:
21                     c7:4a:20:7b:91:4b:20:07:21:ce:cf:68:65:8c:c6:
22                     9d:3b:ef:d5:c1

```



```

21         ASN1 OID: prime256v1
22         NIST CURVE: P-256
23     X509v3 extensions:
24         X509v3 Key Usage: critical
25             Digital Signature, Key Agreement
26         Authority Information Access:
27             CA Issuers -
URI:http://secure.globalsign.com/cacert/gsorganizationvalsha2g2r1.crt
28         OCSP - URI:http://ocsp2.globalsign.com/gsorganizationvalsha2g2
29
30     X509v3 Certificate Policies:
31         Policy: 1.3.6.1.4.1.4146.1.20
32         CPS: https://www.globalsign.com/repository/
33         Policy: 2.23.140.1.2.2
34
35     X509v3 Basic Constraints:
36         CA:FALSE
37     X509v3 CRL Distribution Points:
38
39         Full Name:
40             URI:http://crl.globalsign.com/gs/gsorganizationvalsha2g2.crl
41
42     X509v3 Subject Alternative Name:
43         DNS:*.wikipedia.org, DNS:*.m.mediawiki.org, DNS:*.m.wikibooks.org,
DNS:*.m.wikidata.org, DNS:*.m.wikimedia.org, DNS:*.m.wikimediafoundation.org,
DNS:*.m.wikinews.org, DNS:*.m.wikipedia.org, DNS:*.m.wikiquote.org,
DNS:*.m.wikisource.org, DNS:*.m.wikiversity.org, DNS:*.m.wikivoyage.org,
DNS:*.m.wiktionary.org, DNS:*.mediawiki.org, DNS:*.planet.wikimedia.org,
DNS:*.wikibooks.org, DNS:*.wikidata.org, DNS:*.wikimedia.org,
DNS:*.wikimediafoundation.org, DNS:*.wikinews.org, DNS:*.wikiquote.org,
DNS:*.wikisource.org, DNS:*.wikiversity.org, DNS:*.wikivoyage.org, DNS:*.wiktionary.org,
DNS:*.wmfusercontent.org, DNS:*.zero.wikipedia.org, DNS:mediawiki.org, DNS:w.wiki,
DNS:wikibooks.org, DNS:wikidata.org, DNS:wikimedia.org, DNS:wikimediafoundation.org,
DNS:wikinews.org, DNS:wikiquote.org, DNS:wikisource.org, DNS:wikiversity.org,
DNS:wikivoyage.org, DNS:wiktionary.org, DNS:wmfusercontent.org, DNS:wikipedia.org
44     X509v3 Extended Key Usage:
45         TLS Web Server Authentication, TLS Web Client Authentication
46     X509v3 Subject Key Identifier:
47         28:2A:26:2A:57:8B:3B:CE:B4:D6:AB:54:EF:D7:38:21:2C:49:5C:36
48     X509v3 Authority Key Identifier:
49         keyid:96:DE:61:F1:BD:1C:16:29:53:1C:C0:CC:7D:3B:83:00:40:E6:1A:7C
50
51     Signature Algorithm: sha256WithRSAEncryption
52         8b:c3:ed:d1:9d:39:6f:af:40:72:bd:1e:18:5e:30:54:23:35:
53     ...

```

### 8.2.3 CA证书

证书是用来证明某某东西确实是某某东西的东西（是不是像绕口令？）。通俗地说，证书就好比上文里面的公章。通过公章，可以证明对应的证件的真实性。

理论上，人人都可以找个证书工具，自己做一个证书。那如何防止坏人自己制作证书出来骗人捏？请看后续 CA 的介绍。

CA是Certificate Authority的缩写，也叫“证书授权中心”。

它是负责管理和签发证书的第三方机构, 好比一个可信任的中介公司。一般来说，CA必须是所有行业 and 所有公众都信任的、认可的。因此它必须具有足够的权威性。就好比A、B两公司都必须信任C公司，才会找C公司作为公章的中介。

- CA证书

CA 证书，顾名思义，就是CA颁发的证书。

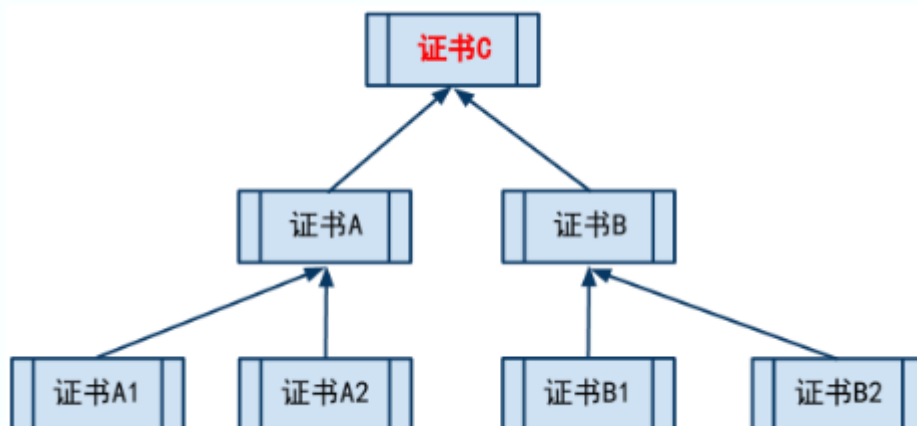
前面已经说了，人人都可以找工具制作证书。但是你一个小破孩制作出来的证书是没啥用处的。因为你不是权威的CA机关，你自己搞的证书不具有权威性。

比如，某个坏人自己刻了一个公章，盖到介绍信上。但是别人一看，不是受信任的中介公司的公章，就不予理睬。坏蛋的阴谋就不能得逞啦。

- 证书信任链

证书直接是可以有信任关系的, 通过一个证书可以证明另一个证书也是真实可信的. 实际上，证书之间的信任关系，是可以嵌套的。比如，C 信任 A1, A1 信任 A2, A2 信任 A3.....这个叫做证书的信任链。只要你信任链上的头一个证书，那后续的证书，都是可以信任滴。

假设 C 证书信任 A 和 B；然后 A 信任 A1 和 A2；B 信任 B1 和 B2。则它们之间，构成如下的一个树形关系（一个倒立的树）。



处于最顶上的树根位置的那个证书，就是“**根证书**”。除了根证书，其它证书都要依靠上一级的证书，来证明自己。那谁来证明“根证书”可靠捏？实际上，根证书自己证明自己是可靠滴（或者换句话说，根证书是不需要被证明滴）。

聪明的同学此刻应该意识到了：根证书是整个证书体系安全的根本。所以，如果某个证书体系中，根证书出了问题（不再可信了），那么所有被根证书所信任的其它证书，也就不再可信了。

- 证书有啥用

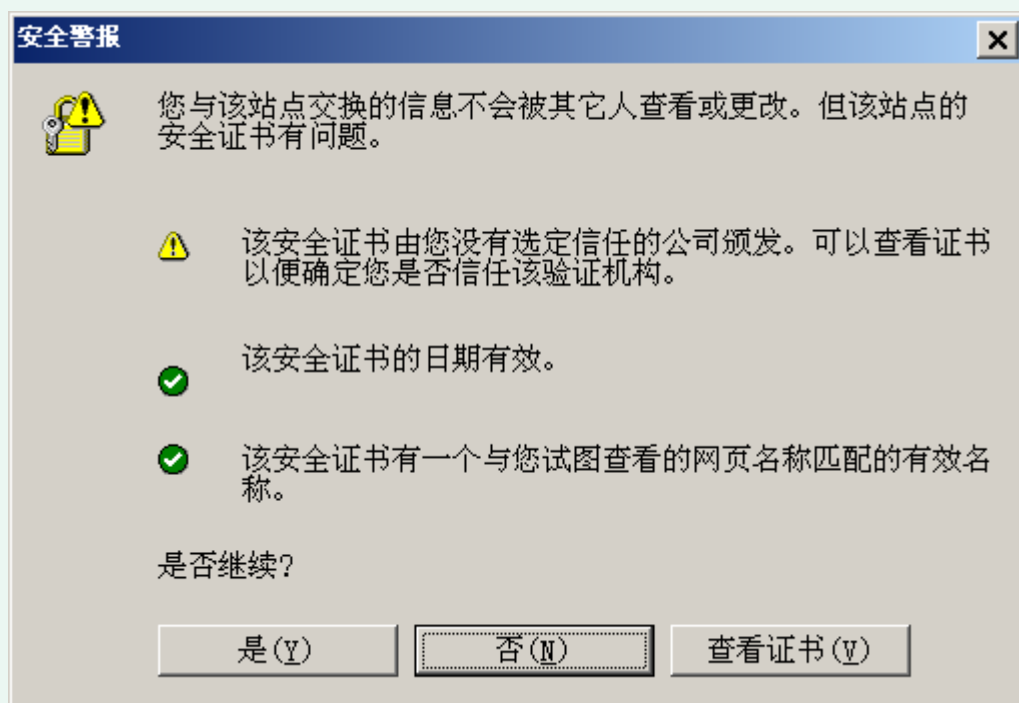
1. 验证网站是否可信（针对HTTPS）

通常，我们如果访问某些敏感的网页（比如用户登录的页面），其协议都会使用 HTTPS 而不是 HTTP。因为 HTTP 协议是明文的，一旦有坏人在偷窥你的网络通讯，他/她就可以看到网络通讯的内容（比如你的密码、银行帐号、等）；而 HTTPS 是加密的协议，可以保证你的传输过程中，坏蛋无法偷窥。

但是，千万不要以为，HTTPS 协议有了加密，就可高枕无忧了。俺再举一个例子来说明，光有加密是不够滴。假设有一个坏人，搞了一个假的网银的站点，然后诱骗你上这个站点。假设你又比较单纯，一不留神，就把你的帐号，口令都输入进去了。那这个坏蛋的阴谋就得逞鸟。

为了防止坏人这么干，HTTPS 协议除了有加密的机制，还有一套证书的机制。通过证书来确保，某个站点确实就是某个站点。

有了证书之后，当你的浏览器在访问某个 HTTPS 网站时，会验证该站点上的 CA 证书（类似于验证介绍信的公章）。如果浏览器发现该证书没有问题（证书被某个根证书信任、证书上绑定的域名和该网站的域名一致、证书没有过期），那么页面就直接打开；否则的话，浏览器会给出一个警告，告诉你该网站的证书存在某某问题，是否继续访问该站点？下面给出 IE 和 Firefox 的抓图：





## 此连接是不受信任的

您想使用 Firefox 来安全连接至 [www.enum.cn](http://www.enum.cn)，但是我们无法确认此连接为安全的。

通常，当您尝试安全连接时，站点会出示受信任的标识，以证明您访问的是正确的地址。然而，现在无法验证此网站的标识。

### 怎么办？

如果您过去连接到此网站并且没有发现问题，那么此错误信息表示可能有人想冒充该网站，所以您应该停止浏览。

立即离开！

### ▼ 技术细节

[www.enum.cn](http://www.enum.cn) 使用了无效的安全证书。

该证书因为其发行者证书不受信而不被信任。

(错误码: sec\_error\_untrusted\_issuer)

### ▼ 我已充分了解可能的风险

如果您了解现在所发生的一切，您可以告诉 Firefox 并让它信任此站点的标识。**即使您信任此站点，这个错误还表明可能有人尝试干扰您的连接。**

不要随便添加例外，除非您知道并认同该网站不使用受信任标识的理由。

添加例外...

大多数知名的网站，如果用了 HTTPS 协议，其证书都是可信的（也就不会出现上述警告）。所以，今后你如果上某个知名网站，发现浏览器跳出上述警告，你就要小心啦！

## 2. 验证某文件是否可信（是否被篡改）

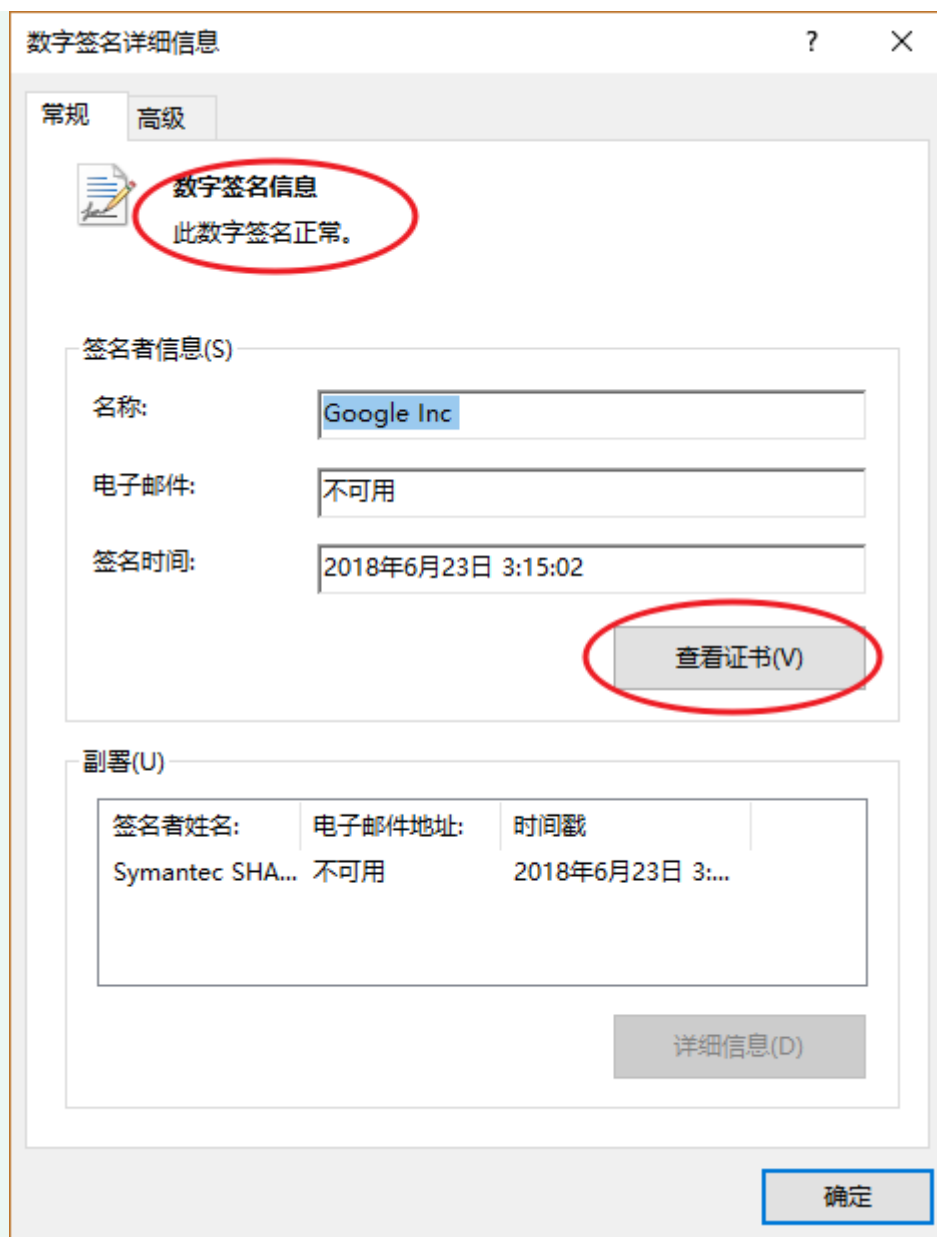
证书除了可以用来验证某个网站，还可以用来验证某个文件是否被篡改。具体是通过证书来制作文件的数字签名。制作数字签名的过程太专业，咱就不说了。后面专门告诉大家如何验证文件的数字签名。考虑到大多数人用 Windows 系统，俺就拿 Windows 的例子来说事儿。

比如，俺手头有一个 Google Chrome 的安装文件（带有数字签名）。当俺查看该文件的属性，会看到如下的界面。眼神好的同学，会注意到到上面有个“**数字签名**”的标签页。如果没有出现这个标签页，就说明该文件没有附带数字签名。

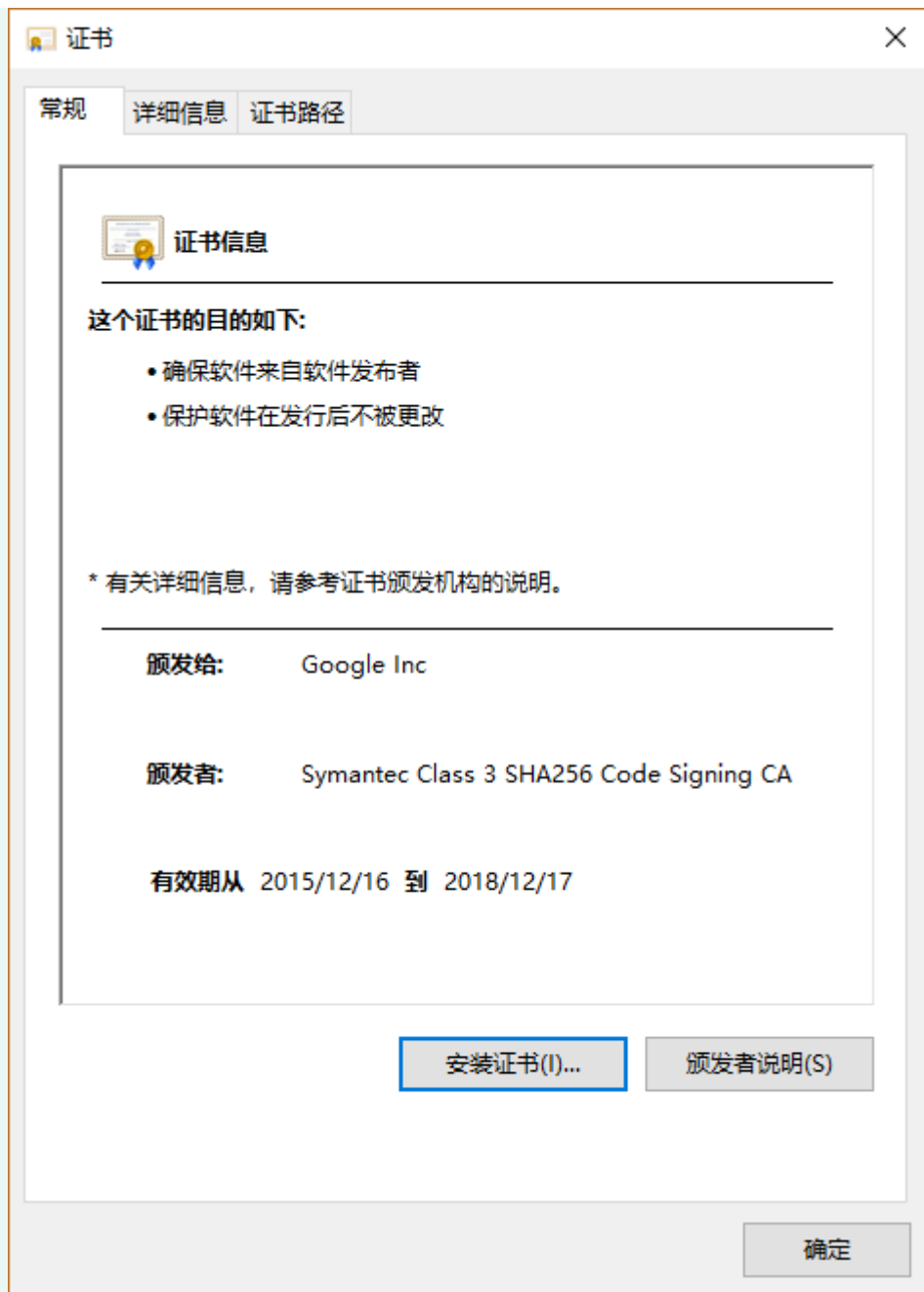


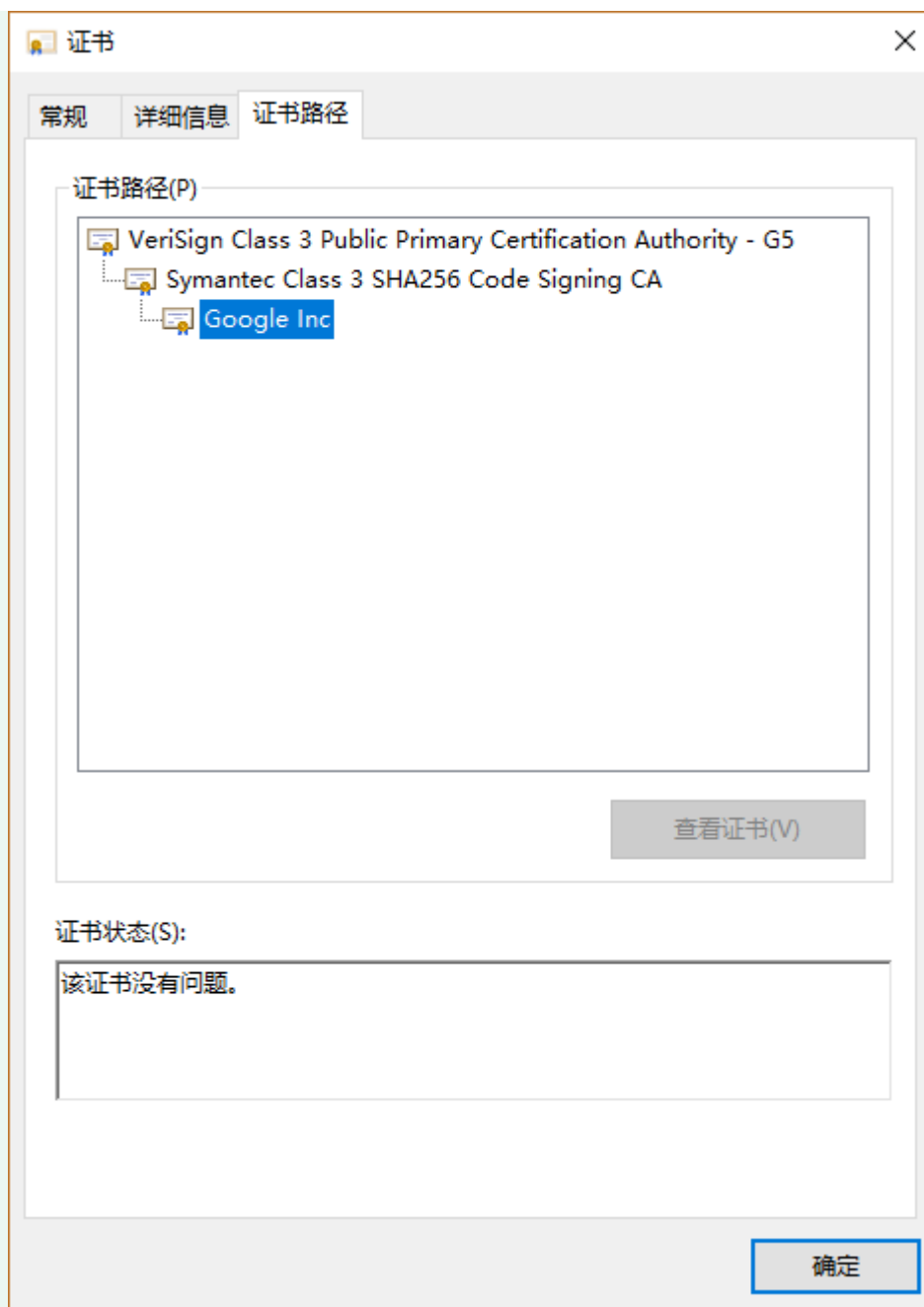
一般来说，签名列表中，有且仅有一个签名。选中它，点“**详细信息**”按钮。跳出如下界面：

通常这个界面会显示一行字：“**该数字签名正常**”（图中红圈标出）。如果有这行字，就说明该文件从出厂到你手里，中途没有被篡改过（是原装滴、是纯洁滴）。如果该文件被篡改过了（比如，感染了病毒、被注入木马），那么对话框会出现一个警告提示“**该数字签名无效**”



不论签名是否正常，你都可以点“查看证书”按钮。这时候，会跳出证书的对话框。如下：





从后一个界面，可以看到刚才说的证书信任链。图中的信任链有3层：

- 第1层是根证书（verisign）。
- 第2层是 symantec 专门用来签名的证书。
- 第3层是 Google 自己的证书。

目前大多数知名的公司（或组织机构），其发布的可执行文件（比如软件安装包、驱动程序、安全补丁），都带有数字签名。你可以自己去看一下。

建议大伙儿在安装软件之前，都先看看是否有数字签名？如果有，就按照上述步骤验证一把。一旦数字签名是坏的，那可千万别装。

## 8.3 公钥基础设施（PKI）

#



仅制定证书的规范还不足以支持公钥的实际运用，我们还需要很多其他的规范，例如证书应该由谁来颁发，如何颁发，私钥泄露时应该如何作废证书，计算机之间的数据交换应采用怎样的格式等。这一节我们将介绍能够使公钥的运用更加有效的公钥基础设施。

### 8.3.1 什么是公钥基础设施

公钥基础设施（Public-Key infrastructure）是为了能够更有效地运用公钥而制定的一系列规范和规格的总称。公钥基础设施一般根据其英语缩写而简称为PKI。

PKI只是一个总称，而并非指某一个单独的规范或规格。例如，RSA公司所制定的PKCS（Public-Key Cryptography Standards，公钥密码标准）系列规范也是PKI的一种，而互联网规格RFC（Request for Comments）中也有很多与PKI相关的文档。此外，X.509这样的规范也是PKI的一种。在开发PKI程序时所使用的由各个公司编写的API（Application Programming Interface, 应用程序编程接口）和规格设计书也可以算是PKI的相关规格。

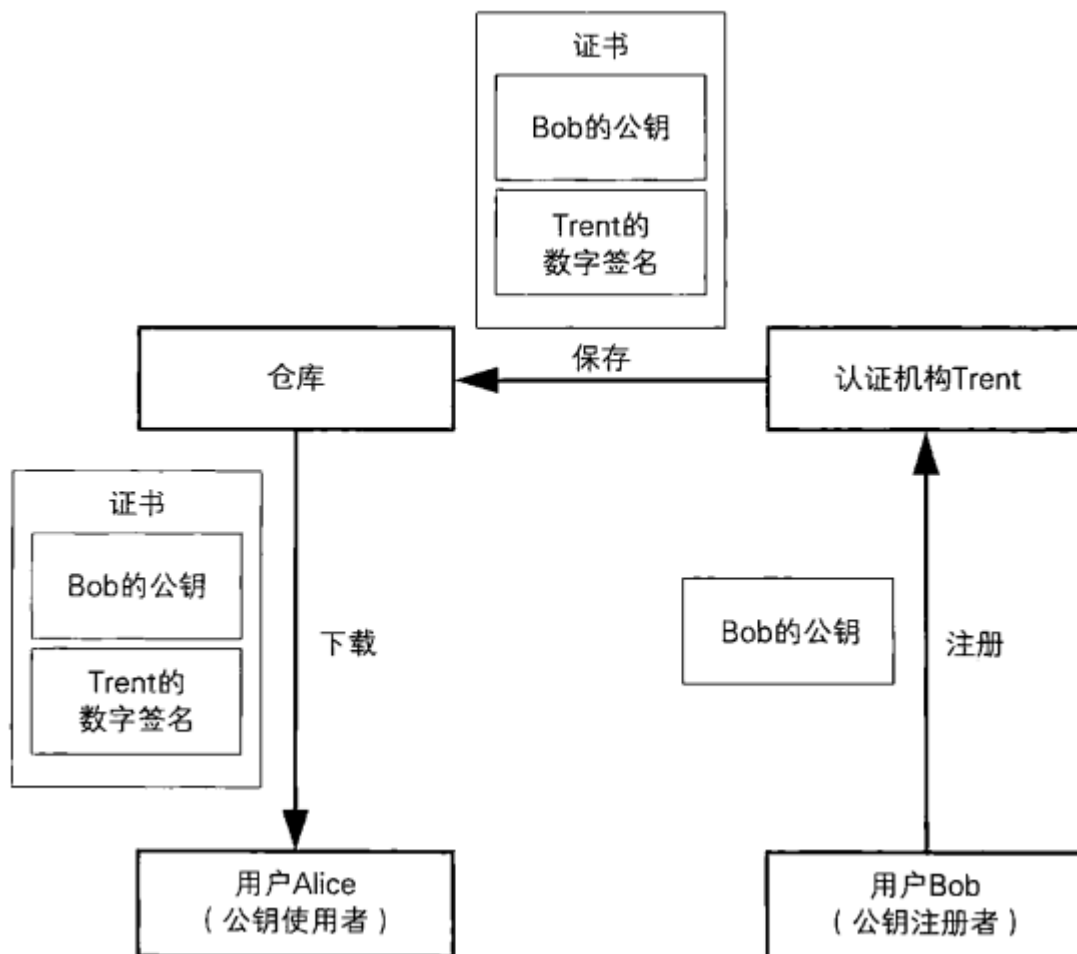
因此，根据具体所采用的规格，PKI也会有很多变种，这也是很多人难以整体理解PKI的原因之一。

为了帮助大家整体理解PKI,我们来简单总结一下PKI的基本组成要素（用户、认证机构、仓库）以及认证机构所负责的工作。

### 8.3.2 PKI的组成要素

PKI的组成要素主要有以下三个：

- 用户 --- 使用PKI的人
- 认证机构 --- 颁发证书的人
- 仓库 --- 保存证书的数据库



## 用户

用户就是像Alice、Bob这样使用PKI的人。用户包括两种：一种是希望使用PKI注册自己的公钥的人，另一种是希望使用已注册的公钥的人。我们来具体看一下这两种用户所要进行的操作。

### • 注册公钥的用户所进行的操作

- 生成密钥对（也可以由认证机构生成）
- 在认证机构注册公钥
- 向认证机构申请证书
- 根据需要申请作废已注册的公钥
- 解密接收到的密文
- 对消息进行数字签名

### • 使用已注册公钥的用户所进行的操作

- 将消息加密后发送给接收者
- 验证数字签名

```
1  /*
2  ===== 小知识点 =====
3  浏览器如何验证SSL证书
4  1. 在IE浏览器的菜单中点击“工具 /Internet选项”，选择“内容”标签，点击“证书”按钮，然后就可以看到IE
5  浏览器已经信任了许多“中级证书颁发机构”和“受信任的根证书颁发机构”。当我们在访问该网站时，浏览器
6  就会自动下载该网站的SSL证书，并对证书的安全性进行检查。
```

7 2. 由于证书是分等级的, 网站拥有者可能从根证书颁发机构领到证书, 也可能从根证书的下一级(如某个国家的  
8 认证中心, 或者是某个省发出的证书)领到证书。假设我们正在访问某个使用了 SSL技术的网站, IE浏  
9 览器就会收到了一个SSL证书, 如果这个证书是由根证书颁发机构签发的, IE浏览器就会按照下面的步骤来  
10 检查: 浏览器使用内 置的根证书中的公钥来对收到的证书进行认证, 如果一致, 就表示该安全证书是由可信  
11 的颁证机构签发的, 这个网站就是安全可靠的; 如果该SSL证书不是根服 务器签发的, 浏览器就会自动检  
12 查上一级的发证机构, 直到找到相应的根证书颁发机构, 如果该根证书颁发机构是可信的, 这个网站的SSL证  
13 书也是可信的。  
14 \*/

## 认证机构 (CA)

认证机构 (Certification Authority, CA) 是对证书进行管理的人。上面的图中我们给它起了一个名字叫作Trent。  
认证机构具体所进行的操作如下:

- 生成密钥对 (也可以由用户生成)

生成密钥对有两种方式: 一种是由PKI用户自行生成, 一种是由认证机构来生成。在认证机构生成用户密钥对的情况下, 认证机构需要将私钥发送给用户, 这时就需要使用PKCS#12 (Personal Information Exchange Syntax Standard) 等规范。

- 在注册公钥时对本人身份进行认证, 生成并颁发证书

在用户自行生成密钥对的情况下, 用户会请求认证机构来生成证书。申请证书时所使用的规范是由PKCS#10 (Certification Request Syntax Standard) 定义的。

认证机构根据其认证业务准则 (Certification Practice Statement, CPS) 对用户的身份进行认证, 并生成证书。在生成证书时, 需要使用认证机构的私钥来进行数字签名。生成的证书格式是由PKCS#6 (Extended-Certificate Syntax Standard) 和 X.509定义的。

- 作废证书

当用户的私钥丢失、被盗时, 认证机构需要对证书进行作废 (revoke)。此外, 即便私钥安然无恙, 有时候也需要作废证书, 例如用户从公司离职导致其失去私钥的使用权限, 或者是名称变更导致和证书中记载的内容不一致等情况。

纸质证书只要撕毁就可以作废了, 但这里的证书是数字信息, 即便从仓库中删除也无法作废, 因为用户会保存证书的副本, 但认证机构又不能入侵用户的电脑将副本删除。

要作废证书, 认证机构需要制作一张证书 **作废清单 (Certificate Revocation List), 简称为CRL**。

CRL是认证机构宣布作废的证书一览表, 具体来说, 是一张已作废的证书序列号的清单, 并由认证机构加上数字签名。证书序列号是认证机构在颁发证书时所赋予的编号, 在证书中都会记载。

PKI用户需要从认证机构获取最新的CRL,并查询自己要用于验证签名 (或者是用于加密) 的公钥证书是否已经作废这个步骤是非常重要的。

假设我们手上有Bob的证书, 该证书有合法的认证机构签名, 而且也在有效期内, 但仅凭这些还不能说明该证书一定是有效的, 还需要查询认证机构最新的CRL, 并确认该证书是否有效。一般来说, 这个检查不是由用户自身来完成的, 而是应该由处理该证书的软件来完成, 但有很多软件并没有及时更新CRL。

认证机构的工作中，公钥注册和本人身份认证这一部分可以由注册机构（Registration Authority，RA）来分担。这样一来，认证机构就可以将精力集中到颁发证书上，从而减轻了认证机构的负担。不过，引入注册机构也有弊端，比如说认证机构需要对注册机构本身进行认证，而且随着组成要素的增加，沟通过程也会变得复杂，容易遭受攻击的点也会增。

## 仓库

仓库（repository）是一个保存证书的数据库，PKI用户在需要的时候可以从中获取证书。它的作用有点像打电话时用的电话本。在本章开头的例子中，尽管没特别提到，但Alice获取Bob的证书时，就可以使用仓库。仓库也叫作证书目录。

### 8.3.3 各种各样的PKI

公钥基础设施（PKI）这个名字总会引起一些误解，比如说“面向公众的权威认证机构只有一个”，或者“全世界的公钥最终都是由一个根CA来认证的”，其实这些都是不正确的。认证机构只要对公钥进行数字签名就可以了，因此任何人都可以成为认证机构，实际上世界上已经有无数个认证机构了。

国家、地方政府、医院、图书馆等公共组织和团体可以成立认证机构来实现PKI，公司也可以出于业务需要在内部实现PKI，甚至你和你的朋友也可以以实验为目的来构建PKI。

在公司内部使用的情况下，认证机构的层级可以像上一节中一样和公司的组织层级一一对应，也可以不一一对应。例如，如果公司在东京、大阪、北海道和九州都成立了分公司，也可以采取各个分公司之间相互认证的结构。在认证机构的运营方面，可以购买用于构建PKI的软件产品由自己公司运营，也可以使用VeriSign等外部认证服务。具体要采取怎样的方式，取决于目的和规模，并没有一定之规。

## 9. SSL/TLS

1 "SSL/TLS --- 为了更安全的通信"

本章中我们将学习SSL/TLS的相关知识。

SSL/TLS是世界上应用最广泛的密码通信方法。比如说，当在网上商城中输入信用卡号时，我们的Web浏览器就会使用SSL/TLS进行密码通信。使用SSL/TLS可以对通信对象进行认证，还可以确保通信内容的机密性。

SSL/TLS中综合运用了之前所学习的对称密码、消息认证码、公钥密码、数字签名、伪随机数生成器等密码技术，大家可以在阅读本章内容的同时对这些技术进行复习。严格来说，SSL（Secure Socket Layer）与TLS（Transport Layer Security）是不同的，TLS相当于是SSL的后续版本。不过，本章中所介绍的内容，大多是SSL和TLS两者兼备的，因此除具体介绍通信协议的部分以外，都统一写作SSL/TLS。

### 9.1 客户端与服务器

#

Bob书店是Alice经常光顾的一家网店，因为在Bob书店她可以搜索到新出版的图书，还可以通过信用卡快速完成支付，购买的书还能快递到家，真的很方便。

有一天，Alice读了一本关于网络信息安全的书，书上说“互联网上传输的数据都是可以被窃听的”。Alice感到非常担心，自己在购买新书的时候输入的信用卡号会不会被窃听呢？

Alice看到Bob书店的网站下面写着一行字：“在以https://开头的网页中输入的信息将通过SSL/TLS发送以确保安全”。

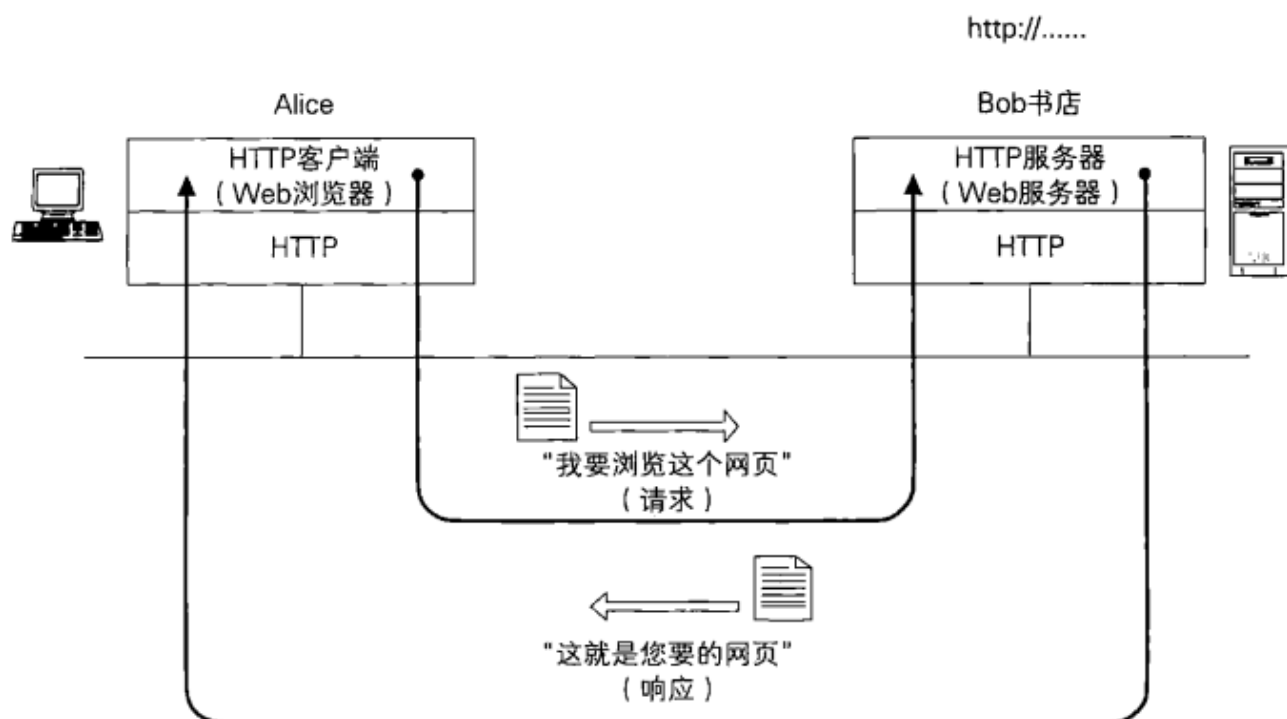
的确，输入信用卡号的网页的URL是以 **https://** 开头的，而不是一般的 **http://**。此外，在浏览这个网页时，Alice的web浏览器上还会显示一个小锁头的图标，看上去好像挺安全的。

但Alice心想，就算写着“通过SSL/TLS发送”我也不放心啊，到底在我的Web浏览器和Bob书店的网站之间都发生了哪些事呢？

本章将要介绍的技术——SSL/TLS就可以解答Alice的疑问。当进行SSL/TLS通信时，Web浏览器上就会显示一个小锁头的图标。

1

**\*\*Alice的Web浏览器（客户端）和Bob书店的网站（服务器）进行HTTP通信\*\***



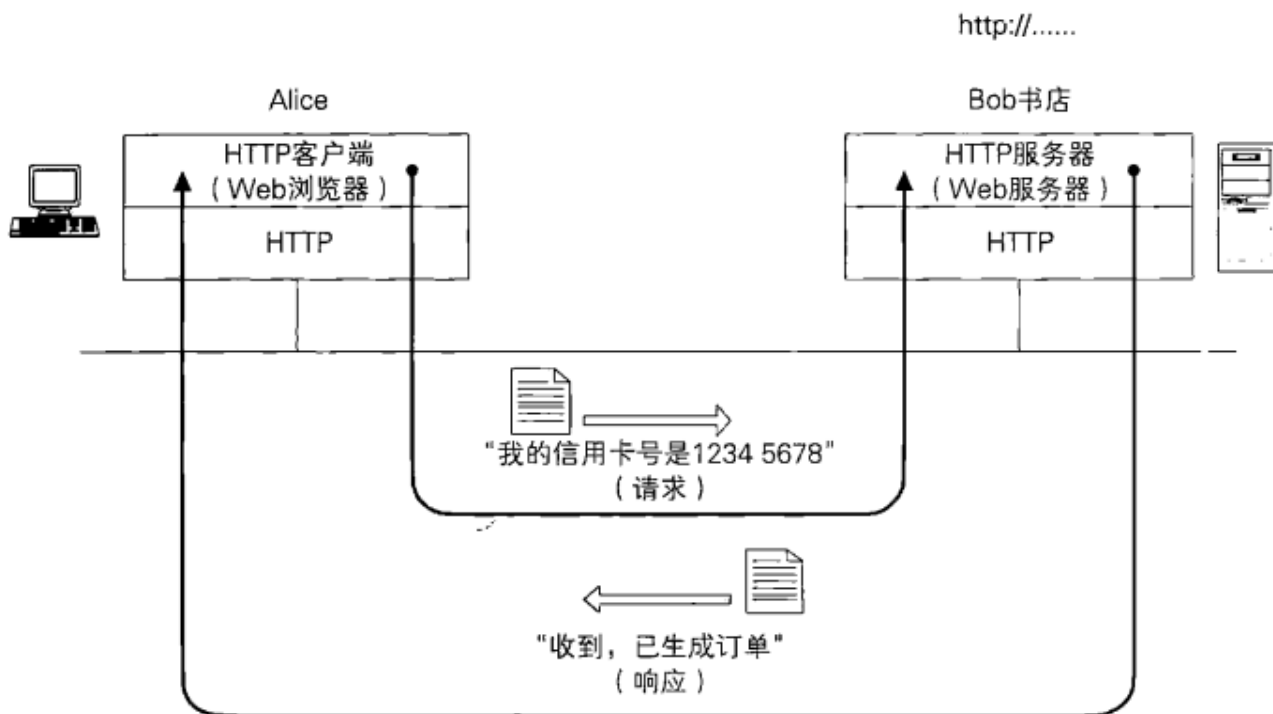
Alice和Bob书店之间的通信，实际上是Alice所使用的Web浏览器和Bob书店的Web服务器之间的通信。Web浏览器是Alice的计算机上运行的一个程序，而web服务器则是在Bob书店的计算机上运行的一个程序，它们都遵循一种叫作HTTP（Hyper Text Transfer Protocol, 超文本传输协议）的协议（protocol）来进行通信。其中，Web浏览器称为HTTP客户端，Web服务器称为HTTP服务器。

当Alice点击网页上的链接或者输入URL时，Web浏览器就会通过网络向Web服务器发送一个“我要浏览这个网页”，的请求（request）。

Web服务器则将请求的网页内容发送给Web浏览器，以便对请求作出响应（response）。服务器和客户端之间所进行的处理就是请求和响应的往复。HTTP可以认为是在HTTP客户端与HTTP服务器之间进行请求和响应的规范。

Alice向Bob书店发送信用卡号也是使用HTTP来完成的（下图）。Alice输入信用卡号之后按下提交按钮，这时客户端（Web浏览器）就会将信用卡号作为HTTP请求发送给服务器。服务器则会将“生成订单”的网页作为HTTP响应返回给客户端。

不过，如果直接发送请求的话，信用卡号就很可能被窃听。下一节我们将探讨针对这种风险的对策。



## 9.2 用SSL/TLS承载HTTP

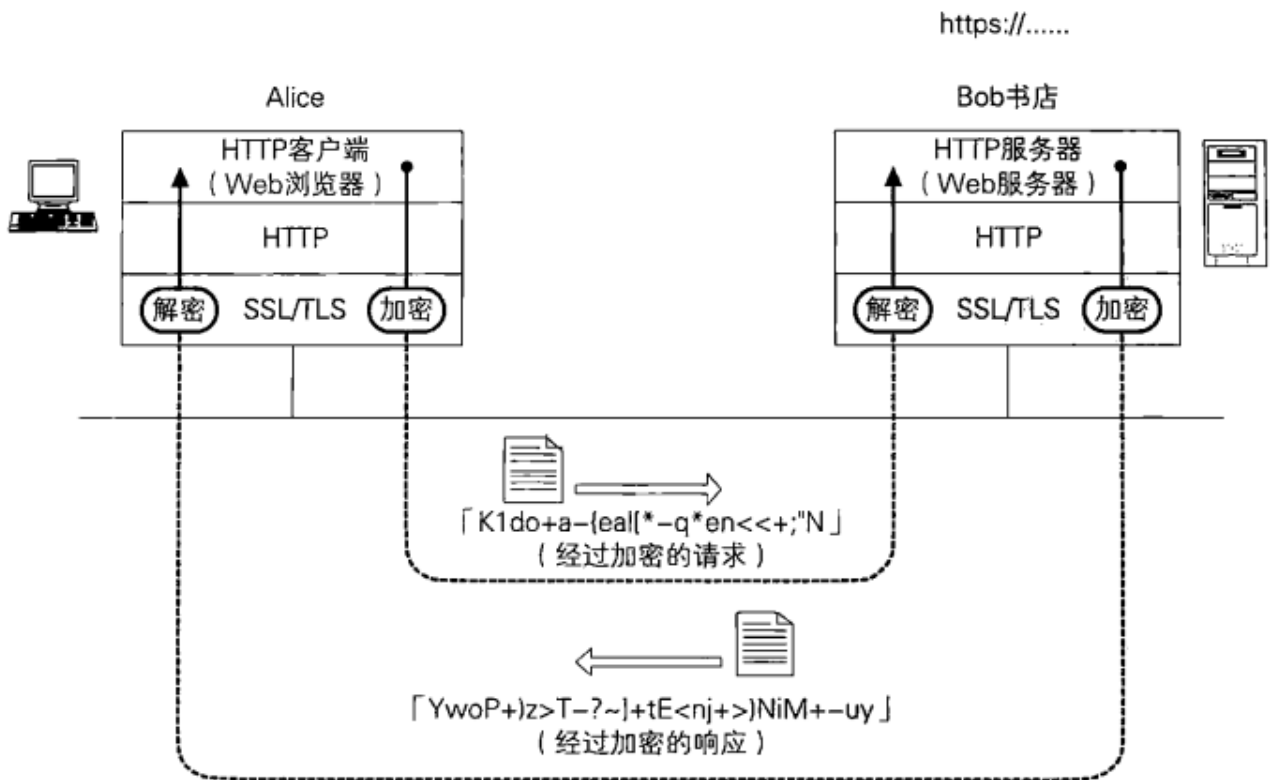
#

什么是SSL,什么是TLS呢? 官话说SSL是安全套接层(secure sockets layer), TLS是SSL的继任者, 叫传输层安全(transport layer security)。说白点, 就是在明文的上层和TCP层之间加上一层加密, 这样就保证上层信息传输的安全。如HTTP协议是明文传输, 加上SSL层之后, 就有了雅称HTTPS。它存在的唯一目的就是保证上层通讯安全的一套机制。

当Web浏览器发送信用卡号时, 信用卡号的数据会作为客户端请求发送给服务器。如果通信内容被窃听者Eve所窃取, Eve就会得到信用卡号。

于是, 我们可以用SSL (Secure Socket Layer) 或者TLS (Transport Layer Security) 作为对通信进行加密的协议, 然后在此之上承载HTTP (下图)。通过将两种协议进行叠加, 我们就可以对HTTP的通信 (请求和响应) 进行加密, 从而防止窃听。通过SSL/TLS进行通信时, URL不是以http://开头, 而是以https://开头。

以上就是SSL/TLS的简单介绍。



在大致了解了SSL/TLS之后，我们来整理一下SSL/TLS到底负责哪些工作。我们想要实现的是，通过本地的浏览器访问网络上的web服务器，并进行安全的通信。用上边的例子来说就是，Alice希望通过web浏览器向Bob书店发送信用卡号。在这里，我们有几个必须要解决的问题。

1. Alice的信用卡号和地址在发送到Bob书店的过程中不能被窃听。
2. Alice的信用卡号和地址在发送到Bob书店的过程中不能被篡改。
3. 确认通信对方的Web服务器是真正的Bob书店。

在这里，（1）是机密性的问题；（2）是完整性的问题；而（3）则是认证的问题。

要确保机密性，可以使用对称加密。由于对称加密算法的密钥不能被攻击者预测，因此我们使用伪随机数生成器来生成密钥。若要将对称加密的密钥发送给通信对象，可以使用非对称加密算法完成密钥交换。要识别篡改，对数据进行认证，可以使用消息认证码。消息认证码是使用单向散列函数来实现的。

要对通信对象进行认证，可以使用对公钥加上数字签名所生成的证书。

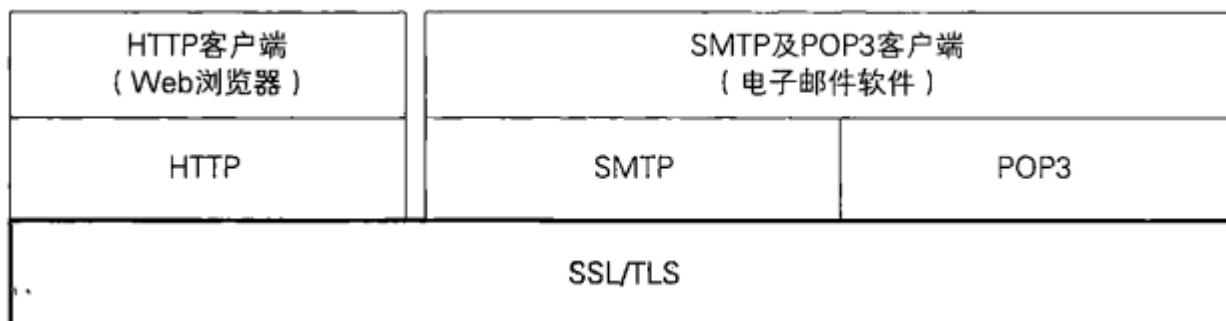
好，工具已经找齐了，下面只要用一个“框架”（framework）将这些工具组合起来就可以了。SSL/TIS协议其实就扮演了这样一种框架的角色。

## SSL/TLS也可以保护其他的协议

刚才我们提到用SSL/TLS承载HTTP通信，这是因为HTTP是一种很常用的协议。其实SSL/TLS上面不仅可以承载HTTP，还可以承载其他很多协议。例如，发送邮件时使用的SMTP（Simple Mail Transfer Protocol, 简单邮件传输协议）和接收邮件时使用的POP3（Post Office Protocol, 邮局协议）都可以用SSL/TLS进行承载。在这样的情况下，SSL/TLS就可以对收发的邮件进行保护。

用SSL/TLS承载HTTP、SMTP和POP3的结构如下图所示。一般的电子邮件软件都可以完成发送和接收邮件这两种操作，其实是同时扮演了SMTP客户端和POP3客户端这两种角色。





## 9.3 https

#

### 9.3.1 http和https

#

HTTP协议：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务端请求和应答的标准（TCP），用于从WWW服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

HTTPS：是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL/TLS层，HTTPS的安全基础是SSL/TLS，因此加密的详细内容就需要SSL/TLS。

HTTPS协议的主要作用可以分为两种：

- 建立一个信息安全通道，来保证数据传输的安全；
- 确认网站的真实性。

HTTPS和HTTP的区别主要如下：

- 1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。
- 2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl/tls加密传输协议。
- 3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
- 4、http的连接很简单，是无状态的；HTTPS协议是由SSL/TLS+HTTP协议构建的可进行加密传输、

1 身份认证的网络协议，比http协议安全。

### 9.3.2 https优缺点

#

#### • https的优点

尽管HTTPS并非绝对安全，掌握根证书的机构、掌握加密算法的组织同样可以进行中间人形式的攻击，但HTTPS仍是现行架构下最安全的解决方案，主要有以下几个好处：

1. 使用HTTPS协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；
2. HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全，可防止数据在传输过程中不被窃取、改变，确保数据的完整性。
3. HTTPS是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。
4. 谷歌曾在2014年8月份调整搜索引擎算法，并称“比起同等HTTP网站，采用HTTPS加密的网站在搜索结果中的排名将会更高”。



- https的缺点

虽然说HTTPS有很大的优势，但其相对来说，还是存在不足之处的：

1. HTTPS协议握手阶段比较费时，会使页面的加载时间延长近50%，增加10%到20%的耗电；
2. HTTPS连接缓存不如HTTP高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；
3. SSL/TLS证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。
4. SSL/TLS证书通常需要绑定IP，不能在同一IP上绑定多个域名，IPv4资源不可能支撑这个消耗。
5. HTTPS协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL证书的信用链体系并不安全，特别是在某些国家可以控制CA根证书的情况下，中间人攻击一样可行。

## 参考资料

### PKCS15个标准

#

PKCS 全称是 Public-Key Cryptography Standards，是由 RSA 实验室与其它安全系统开发商为促进公钥密码的发展而制订的一系列标准。

可以到官网上看看 [What is PKCS](#)

<http://www.rsa.com/rsalabs/node.asp?id=2308>

PKCS 目前共发布过 15 个标准：

1. **PKCS#1: RSA加密标准。** PKCS#1定义了RSA公钥函数的基本格式标准，特别是数字签名。它定义了数字签名如何计算，包括待签名数据和签名本身的格式；它也定义了PSA公/私钥的语法。
2. PKCS#2：涉及了RSA的消息摘要加密，这已被并入PKCS#1中。
3. PKCS#3: Diffie-Hellman密钥协议标准。PKCS#3描述了一种实现Diffie- Hellman密钥协议的方法。
4. PKCS#4：最初是规定RSA密钥语法的，现已经被包含进PKCS#1中。
5. PKCS#5: 基于口令的加密标准。PKCS#5描述了使用由口令生成的密钥来加密8位位组串并产生一个加密的8位位组串的方法。PKCS#5可以用于加密私钥，以便于密钥的安全传输（这在PKCS#8中描述）。
6. PKCS#6: 扩展证书语法标准。PKCS#6定义了提供附加实体信息的X.509证书属性扩展的语法（当PKCS#6第一次发布时，X.509还不支持扩展。这些扩展因此被包括在X.509中）。
7. PKCS#7: 密码消息语法标准。PKCS#7为使用密码算法的数据规定了通用语法，比如数字签名和数字信封。PKCS#7提供了许多格式选项，包括未加密或签名的格式化消息、已封装（加密）消息、已签名消息和既经过签名又经过加密的消息。
8. PKCS#8: 私钥信息语法标准。PKCS#8定义了私钥信息语法和加密私钥语法，其中私钥加密使用了PKCS#5标准。
9. PKCS#9: 可选属性类型。PKCS#9定义了PKCS#6扩展证书、PKCS#7数字签名消息、PKCS#8私钥信息和PKCS#10证书签名请求中要用到的可选属性类型。已定义的证书属性包括E-mail地址、无格式姓名、内容类型、消息摘要、签名时间、签名副本（counter signature）、质询口令和扩展证书属性。
10. PKCS#10: 证书请求语法标准。PKCS#10定义了证书请求的语法。证书请求包含了一个唯一识别名、公钥和可选的一组属性，它们一起被请求证书的实体签名（证书管理协议中的PKIX证书请求消息就是一个PKCS#10）。

11. PKCS#11: 密码令牌接口标准。PKCS#11或“Cryptoki”为拥有密码信息（如加密密钥和证书）和执行密码学函数的单用户设备定义了一个应用程序接口（API）。智能卡就是实现Cryptoki的典型设备。注意：Cryptoki定义了密码函数接口，但并未指明设备具体如何实现这些函数。而且Cryptoki只说明了密码接口，并未定义对设备来说可能有用的其他接口，如访问设备的文件系统接口。
12. PKCS#12: 个人信息交换语法标准。PKCS#12定义了个人身份信息（包括私钥、证书、各种秘密和扩展字段）的格式。PKCS#12有助于传输证书及对应的私钥，于是用户可以在不同设备间移动他们的个人身份信息。
13. PKCS#13: 椭圆曲线密码标准。PKCS#13标准当前正在完善之中。它包括椭圆曲线参数的生成和验证、密钥生成和验证、数字签名和公钥加密，还有密钥协定，以及参数、密钥和方案标识的ASN.1语法。
14. PKCS#14: 伪随机数产生标准。PKCS#14标准当前正在完善之中。为什么随机数生成也需要建立自己的标准呢？PKI中用到的许多基本的密码学函数，如密钥生成和Diffie-Hellman共享密钥协商，都需要使用随机数。然而，如果“随机数”不是随机的，而是取自一个可预测的取值集合，那么密码学函数就不再是绝对安全了，因为它的取值被限于一个缩小了的值域中。因此，安全伪随机数的生成对于PKI的安全极为关键。
15. **PKCS#15: 密码令牌信息语法标准。**PKCS#15通过定义令牌上存储的密码对象的通用格式来增进密码令牌的互操作性。在实现PKCS#15的设备上存储的数据对于使用该设备的所有应用程序来说都是一样的，尽管实际上在内部实现时可能所用的格式不同。PKCS#15的实现扮演了翻译家的角色，它在卡的内部格式与应用程序支持的数据格式间进行转换。