第十三章 Go语言加密算法

本章重点为大家介绍如下的内容：

* 哈希算法（MD5 RipeMD160 SHA256 ）
* DES、3DES对称加密
* RSA非对称加密算法
* 椭圆曲线数字签名算法ECDSA
* 编码解码 （base64 base58 base58check）

13.1 哈希算法

13.1.1 Hash的定义

hash （哈希或散列）算法是信息技术领域非常基础也非常重要的技术。它能任意长度的二进制值（明文）映射为较短的固定长度的二进制值（Hash 值），并且不同的明文很难映射为相同的 Hash 值。hash 值在应用中又被称为数字指纹（fingerprint）或数字摘要（digest）、消息摘要。

例如计算一段话“hello blockchain”的 MD5 hash 值为：78e6a8bcdef7a4a254c16054b082c783

这意味着我们只要对某文件进行 MD5 Hash 计算，得到结果为 78e6a8bcdef7a4a254c16054b082c783，这就说明文件内容极大概率上就是 “hello blockchain”。可见，Hash 的核心思想十分类似于基于内容的编址或命名。

一个优秀的 hash 算法，将能实现：

正向快速：给定明文和 hash 算法，在有限时间和有限资源内能计算出 hash 值。

逆向困难：给定（若干） hash 值，在有限时间内很难（基本不可能）逆推出明文。

输入敏感：原始输入信息修改一点信息，产生的 hash 值看起来应该都有很大不同。

抗冲突：对不同的关键字可能得到同一散列地址，或者说两段内容不同的明文，它们的 hash 值可能一致。这种现象称冲突或者碰撞。具有相同函数值的关键字对该散列函数来说称做同义词。

抗冲突又称为“抗碰撞性”或冲突避免。哈希函数抗冲突就是不同的输入不能产生相同的输出。

抗冲突并不是不会有冲突，只不过找到有冲突的两个输入的代价非常大。就好像暴力破解一个有效期为20年的密码，整个破解过程或许长达30年。这样虽然最后密码被破解了，但是也失去意义了。

13.1.2 流行的哈希算法

目前流行的 Hash 算法包括 MD5、SHA-1 和 SHA-2系列（SHA-224、SHA-256、SHA-384，和 SHA-512 ）。

1.MD4

MD4（RFC 1320）是 MIT (Massachusetts Institute of Technology , 麻州理工大学) 的 Ronald L. Rivest （荣获2002年图灵奖）在 1990 年设计的，MD 是 Message Digest 的缩写。其输出为 128 位。

2.MD5

MD5（RFC 1321）是 Rivest 于1991年对 MD4 的改进版本。其输出是 128 位。MD5 比 MD4 复杂，并且计算速度要慢一点，更安全一些。MD5 已被证明不具备“强抗碰撞性”。

MD5 是一个经典的 hash 算法，其和 SHA-1 算法都已被证明安全性不足应用于商业场景。



图 13.1

3.SHA

SHA （Secure Hash Algorithm， [sɪ'kjʊə(r)] ['ælɡərɪðəm] ，加密哈希算法）是一个 Hash 函数族，由 NIST（National Institute of Standards and Technology ， 美国国家标准技术研究所）于 1993 年发布第一个算法。SHA-1 在 1995 年面世，它的输出为长度 160 位的 hash 值，SHA-1 设计时基于和 MD4 相同原理，SHA-1 已被证明不具备“强抗碰撞性”。

为了提高安全性，NIST 还设计出了 SHA-224、SHA-256、SHA-384，和 SHA-512 算法（统称为 SHA-2），跟 SHA-1 算法原理类似。

目前，一般认为 MD5 和 SHA1 已经不够安全，推荐至少使用 SHA-256 算法。

4.RIPEMD-160(RACE Integrity Primitives Evaluation Message Digest 160 ， RACE完整的原始评估信息摘要 )是一个160位的加密哈希函数，旨在替代128位哈希函数MD4、MD5。

13.1.3 Hash与加密解密的区别

1.哈希（Hash）与加密（Encrypt）的区别

这两个概念不是很清晰，容易混淆两者。而正确区别两者是正确选择和使用哈希与加密的基础。

哈希（Hash）是将目标文本转换成具有相同长度的、不可逆的杂凑字符串（或叫做消息摘要），而加密（Encrypt）是将目标文本转换成具有不同长度的、可逆的密文。

哈希算法往往被设计成生成具有相同长度的文本，而加密算法生成的文本长度与明文本身的长度有关。

哈希的结果具有相同的长度，而加密的结果则长度不同。实际上，如果使用相同的哈希算法，不论你的输入有多么长，得到的结果长度是一个常数，而加密算法往往与明文的长度成正比。

哈希算法是不可逆的，而加密算法是可逆的。

哈希不是一一映射，是不可逆的。而加密算法是一一映射，是可逆的。

哈希是不可能可逆的，因为如果可逆，那么哈希就是世界上最强悍的压缩方式了——能将任意大小的文件压缩成固定大小。

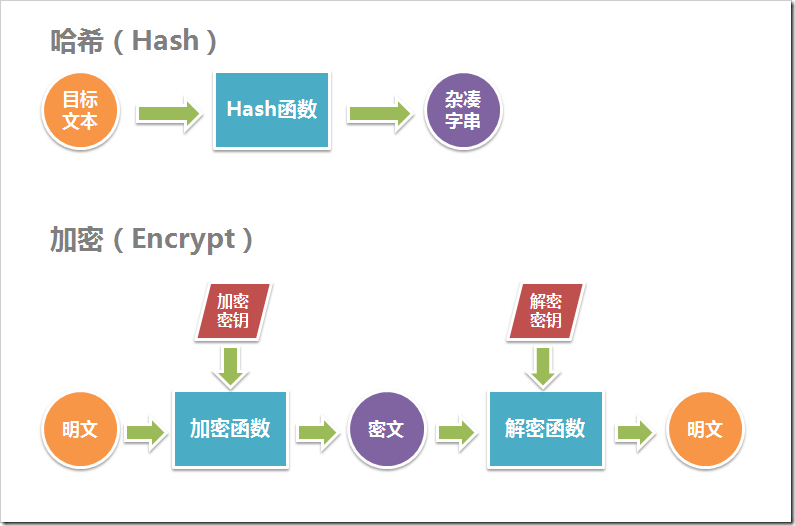


图 13.2

2.哈希（Hash）与加密（Encrypt）的选择

要实现数据保护，可以选择使用哈希或加密两种方式。那么在什么时候该选择哈希、什么时候该选择加密呢？

基本原则是：

如果被保护数据仅仅用作比较验证，在以后不需要还原成明文形式，则使用哈希；

如果被保护数据在以后需要被还原成明文，则需要使用加密。

例如，你正在做一个系统，你打算当用户忘记自己的登录口令时，重置此用户口令为一个随机口令，而后将此随机口令发给用户，让用户下次使用此口令登录，则适合使用哈希。实际上很多网站都是这么做的，想想你以前登录过的很多网站，是不是当你忘记口令的时候，网站并不是将你忘记的口令发送给你，而是发送给你一个新的、随机的口令，然后让你用这个新口令登录。这是因为你在注册时输入的口令被哈希后存储在数据库里，而哈希算法不可逆，所以即使是网站管理员也不可能通过哈希结果复原你的口令，而只能重置口令。

相反，如果你做的系统要求在用户忘记口令的时候必须将原口令发送给用户，而不是重置其口令，则必须选择加密而不是哈希。

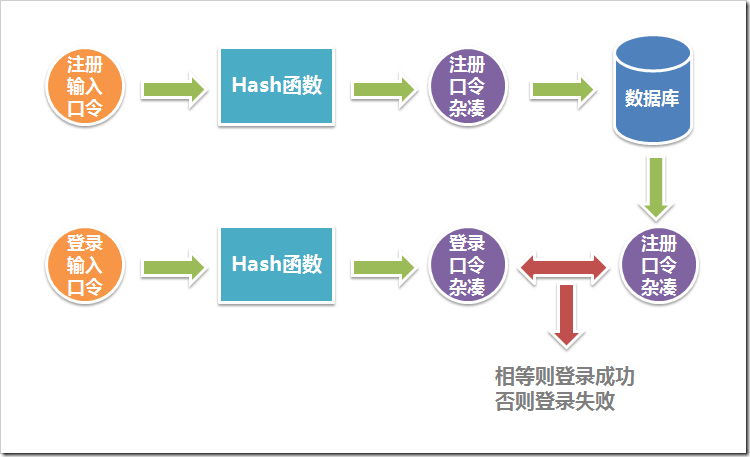


图 13.3

3.对简单哈希（Hash）的攻击

对于哈希的攻击，主要有寻找碰撞法和穷举法。

目前对于MD5和SHA1并不存在有效地寻找碰撞方法。我国杰出的数学家王小云教授曾经在国际密码学会议上发布了对于MD5和SHA1的碰撞寻找改进算法，但这种方法和“破解”相去甚远。该理论目前仅具有数学上的意义，她将破解MD5的预期步骤降低了好几个数量级，但对于实际应用来说仍然是一个天文数字。



图 13.4

穷举法（或暴力破解法），通俗来说，就是在一个范围内，如从000000到999999，将其中所有值一个一个用哈希算法哈希，然后将结果和杂凑串比较，如果相同，则这个值就一定是源字串或源字串的一个碰撞，于是就可以用这个值非法登录了。穷举法看似笨拙，但目前几乎所有的MD5破解机或MD5在线破解都是用这种穷举法。

纠其缘由，就是相当一部分口令是非常简单的，如“123456”或“000000”。穷举法是否能成功很大程度上取决于口令的复杂性。因为穷举法扫描的区间往往是单字符集、规则的区间，或者由字典数据进行组合，因此，如果使用复杂的口令，例如“!@#$%^&\*()”这种口令，穷举法就很难奏效了。

13.1.4 SHA256

SHA-256算法输入报文的最大长度不超过2^64 bit，输入按512-bit分组进行处理，产生的输出是一个256-bit的报文摘要。

SHA256算法包括以下几步：

step1:附加填充比特。

对报文进行填充使报文长度与448 模512 同余（长度=448 mod 512）,填充的比特数范围是1 到512，填充比特串的最高位为1,其余位为0。就是先在报文后面加一个 1，再加很多个0，直到长度满足mod 512=448.为什么是448，因为448+64=512. 第二步会加上一个 64bit的原始报文的 长度信息。

step2:附加长度值。

将用64-bit 表示的初始报文（填充前）的位长度附加在步骤1的结果 后（低位字节优先）。

step3:初始化缓存。

使用一个256-bit 的缓存来存放该散列函数的中间及最终结果。该缓存表示为A=0x6A09E667 , B=0xBB67AE85 , C=0x3C6EF372 , D=0xA54FF53A, E=0x510E527F , F=0x9B05688C , G=0x1F83D9AB , H=0x5BE0CD19 。

step4:处理512-bit（16 个字）报文分组序列。

该算法使用了六种基本逻辑函数，由64 步迭代运算组成。每步都以256-bit 缓存值ABCDEFGH 为输入，然后更新缓存内容。

每步使用一个32-bit 常数值Kt 和一个32-bit Wt。

13.1.5 核心代码

//1.基础工具函数

/\*\*

\* 将字节数组转成16进制字符串： []byte -> string

\*/

func BytesToHexString(arr []byte) string {

return hex.EncodeToString(arr)

}

/\*\*

\* 将16进制字符串转成字节数组： hex string -> []byte

\*/

func HexStringToBytes(s string) ([]byte, error) {

arr, err := hex.DecodeString(s)

return arr, err

}

/\*\*

\* 16进制字符串大端和小端颠倒

\*/

func ReverseHexString(hexStr string) string {

arr, \_ := hex.DecodeString(hexStr)

ReverseBytes(arr)

return hex.EncodeToString(arr)

}

/\*\*

\* 字节数组大端和小端颠倒

\*/

func ReverseBytes(data []byte) {

for i, j := 0, len(data)-1; i < j; i, j = i+1, j-1 {

data[i], data[j] = data[j], data[i]

}

}

//2.哈希函数

func SHA256HexString(text string) string {

sha256Instance := sha256.New()

arr, \_ := hex.DecodeString(text)

sha256Instance.Write(arr)

ciphertext := sha256Instance.Sum(nil)

return fmt.Sprintf("%x", ciphertext)

}

func SHA256Double(text string, isHex bool) string {

sha256Instance := sha256.New()

if isHex {

arr, \_ := hex.DecodeString(text)

sha256Instance.Write(arr)

} else {

sha256Instance.Write([]byte(text))

}

hashBytes := sha256Instance.Sum(nil)

sha256Instance.Reset()

sha256Instance.Write(hashBytes)

hashBytes = sha256Instance.Sum(nil)

return fmt.Sprintf("%x", hashBytes)

}

func HASH(text string, hashType string, isHex bool) string {

var hashInstance hash.Hash

switch hashType {

case "md4":

hashInstance = md4.New()

case "md5":

hashInstance = md5.New()

case "sha1":

hashInstance = sha1.New()

case "sha256":

hashInstance = sha256.New()

case "sha512":

hashInstance = sha512.New()

case "ripemd160":

hashInstance = ripemd160.New()

}

if isHex {

arr, \_ := hex.DecodeString(text)

hashInstance.Write(arr)

} else {

hashInstance.Write([]byte(text))

}

ciphertext := hashInstance.Sum(nil)

return fmt.Sprintf("%x", ciphertext)

}

13.2 对称加密算法

13.2.1 对称加密的概述

在对称加密算法中，数据发信方将明文（原始数据）和加密密钥一起经过特殊加密算法处理后，使其变成复杂的加密密文发送出去。收信方收到密文后，若想解读原文，则需要使用加密用过的密钥及相同算法的逆算法对密文进行解密，才能使其恢复成可读明文。在对称加密算法中，使用的密钥只有一个，发收信双方都使用这个密钥对数据进行加密和解密，这就要求解密方事先必须知道加密密钥。

对称加密(也叫私钥加密算法)指加密和解密使用相同密钥的加密算法。有时又叫传统密码算法。在大多数的对称算法中，加密密钥和解密密钥是相同的，所以也称这种加密算法为单密钥加密算法。它要求发送方和接收方在安全通信之前，商定一个密钥。对称算法的安全性依赖于密钥，泄漏密钥就意味着任何人都可以对他们发送或接收的消息解密，所以密钥的保密性对通信的安全性至关重要。

优势：对称加密算法的特点是算法公开、计算量小、加密速度快、加密效率高。

不足之处是，交易双方都使用同样钥匙，安全性得不到保证。此外，每对用户每次使用对称加密算法时，都需要使用其他人不知道的惟一钥匙，这会使得发收信双方所拥有的钥匙数量呈几何级数增长，密钥管理成为用户的负担。对称加密算法在分布式网络系统上使用较为困难，主要是因为密钥管理困难，使用成本较高。而与公开密钥加密算法比起来，对称加密算法能够提供加密和认证却缺乏了签名功能，使得使用范围有所缩小。

基于“对称密钥”的加密算法主要有DES、3DES（TripleDES）、AES、RC2、RC4、RC5和Blowfish等。本章将介绍最常用的对称加密算法DES、3DES（TripleDES）和AES。

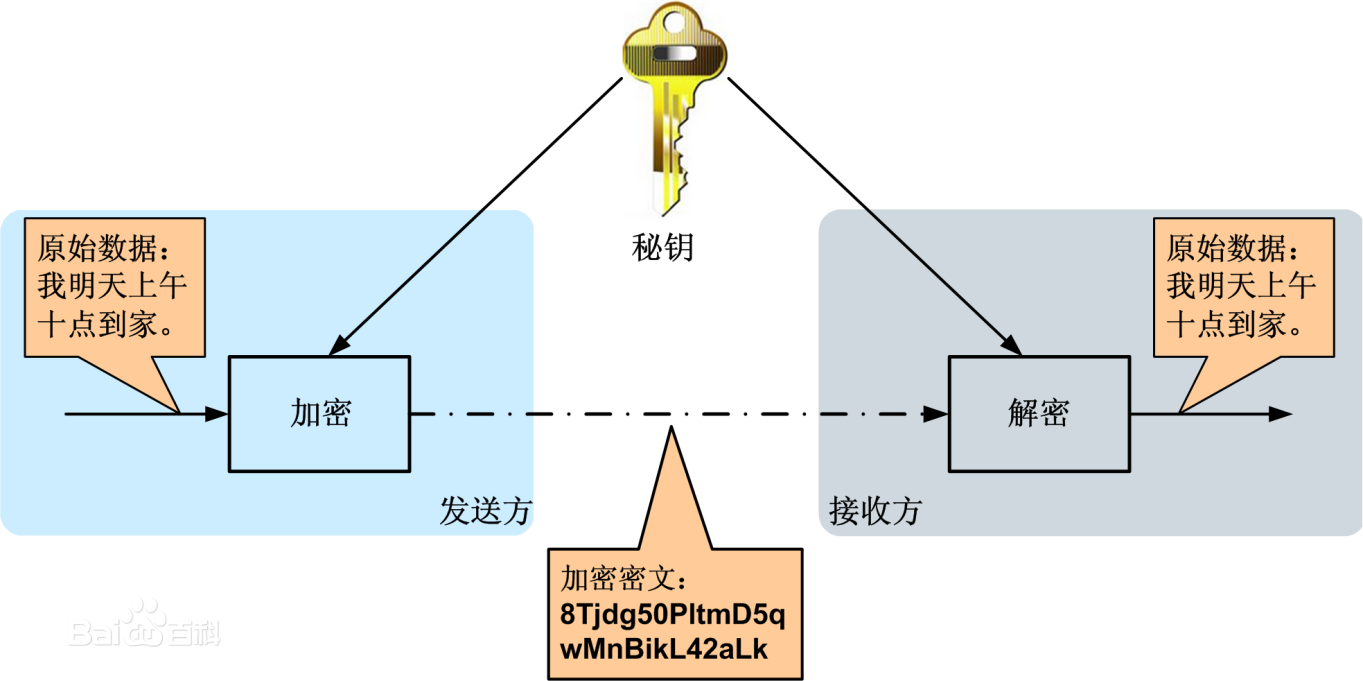


图 13.5

13.2.2 DES和TripleDES算法

DES算法全称为Data Encryption Standard，即数据加密标准算法。

DES是加密和解密使用相同密钥的加密算法，也叫做单密钥算法或私钥加密算法，传统密钥算法。它是IBM公司于1975年研究成功并公开发表的。

DES算法的入口参数有三个：Key、Data、Mode。

其中Key是DES算法的工作密钥，8个字节共64位；

Data是要被加密或被解密的数据；

Mode为DES的工作方式,有两种：加密或解密。

没有密钥的情况下，解密耗费时间非常长，基本上认为没有可能。

加密解密耗时和需要加密的文本大小成正比，这是P问题。

如果知道明文和对应的密文，求解所用的密钥，这是NP问题。

目前还没有NP的求解算法，但是很容易得到验证。想得到NP的解，只能暴力破解（穷举破解）

穷举验证成为对称加密仅有的求解方式，求解时间呈指数级增长。

DES算法把64位的明文输入块变为数据长度为64位的密文输出块，其中8位为奇偶校验位，另外56位作为密码的长度。

首先，DES把输入的64位数据块按位重新组合，并把输出分为L0、R0两部分，每部分各长32位，并进行前后置换，最终由L0输出左32位，R0输出右32位

根据这个法则经过16次迭代运算后，得到L16、R16，将此作为输入，进行与初始置换相反的逆置换，即得到密文输出。

DES算法具有极高的安全性，到目前为止，除了用穷举搜索法对DES算法进行攻击外，还没有发现更有效的办法。

56位长密钥的穷举空间为2^56，这意味着如果一台计算机的速度是每秒种检测100万个密钥，那么它搜索完全部密钥就需要将近2285年的时间，因此DES算法是一种很可靠的加密方法。

3DES密钥是24字节，即192位二进制。

13.2.3 AES加密

高级加密标准（英语：Advanced Encryption Standard，缩写：AES）,AES高级加密标准，是下一代的加密算法标准，速度快，安全级别高，目前 AES 标准的一个实现是Rijndael 算法；

Rijndael（读作rain-dahl）是由美国国家标准与技术协会（NIST）所选的高级加密标准（AES）的算法。

Rijndael由比利时计算机科学家Vincent Rijmen和Joan Daemen开发，近似读音：Rijn [rain] dael [del] (莱恩戴尔)。

这个标准用来替代原先的DES，已经被多方分析且广为全世界所使用。

AES高级加密标准由美国国家标准与技术研究院（NIST）于2001年11月26日发布，并在2002年5月26日成为有效的标准。2006年，高级加密标准已然成为对称密钥加密中最流行的算法之一。

该算法与其他对称密码算法相比更安全、效率更高等特点。

Rijndael是一个反复运算的加密算法，它允许可变动的数据区块及密钥的长度。数据区块与密钥长度的变动是各自独立的。

AES使用128位，192位或者256位的密钥长度（密钥分别是：16字节、24字节、32字节），使得它比密钥长度为56位的DES更健壮可靠。Rijndael被设计用来支持更多的密钥长度，然而除了上述3种密钥长度，其他密钥长度并没有被AES采用。

2^64 = 18446744073709551616

2^64这个数大于全球小麦1000年的产量。如果1微秒验证一个密码（1秒验证100万个），穷举需要费时58万年。

2^256 约= 10 ^ 77

10^80 是当前人类可见宇宙中所有物质原子数目的总和。

13.2.4 AES的加密模式

|  |  |
| --- | --- |
| 加密模式(英文名称及简写) | 中文名称 |
| Electronic Code Book(ECB) | 电子密码本模式 |
| Cipher Block Chaining(CBC) | 密码分组链接模式 |
| Cipher Feedback Mode(CFB) | 加密反馈模式 |
| Output Feedback Mode(OFB) | 输出反馈模式 |

ECB：最基本的加密模式，也就是通常理解的加密，相同的明文将永远加密成相同的密文，无初始向量，容易受到密码本重放攻击，一般情况下很少用。

CBC：明文被加密前要与前面的密文进行异或运算后再加密，因此只要选择不同的初始向量，相同的密文加密后会形成不同的密文，这是目前应用最广泛的模式。CBC加密后的密文是上下文相关的，但明文的错误不会传递到后续分组，但如果一个分组丢失，后面的分组将全部作废(同步错误)。

CFB：类似于自同步序列密码，分组加密后，按8位分组将密文和明文进行移位异或后得到输出同时反馈回移位寄存器，优点最小可以按字节进行加解密，也可以是n位的，CFB也是上下文相关的，CFB模式下，明文的一个错误会影响后面的密文(错误扩散)。

OFB：将分组密码作为同步序列密码运行，和CFB相似，不过OFB用的是前一个n位密文输出分组反馈回移位寄存器，OFB没有错误扩散问题。

13.2.5 核心代码

//1.DES加密解密

//DES加密字节数组，返回字节数组

func DesEncrypt(originalBytes, key []byte) ([]byte, error) {

block, err := des.NewCipher(key)

if err != nil {

return nil, err

}

originalBytes = PKCS5Padding(originalBytes, block.BlockSize())

blockMode := cipher.NewCBCEncrypter(block, key)

cipherArr := make([]byte, len(originalBytes))

blockMode.CryptBlocks(cipherArr, originalBytes)

return cipherArr, nil

}

//DES解密字节数组，返回字节数组

func DesDecrypt(cipherBytes, key []byte) ([]byte, error) {

block, err := des.NewCipher(key)

if err != nil {

return nil, err

}

blockMode := cipher.NewCBCDecrypter(block, key)

originalText := make([]byte, len(cipherBytes))

blockMode.CryptBlocks(originalText, cipherBytes)

originalText = PKCS5UnPadding(originalText)

return originalText, nil

}

//DES加密文本，返回加密后文本

func DesEncryptString(originalText string, key []byte) (string, error) {

cipherArr, err := DesEncrypt([]byte(originalText), key)

if err != nil {

return "", err

}

base64str := base64.StdEncoding.EncodeToString(cipherArr)

return base64str, nil

}

//对加密文本进行DES解密，返回解密后明文

func DesDecryptString(cipherText string, key []byte) (string, error) {

cipherArr, \_ := base64.StdEncoding.DecodeString(cipherText)

cipherArr, err := DesDecrypt(cipherArr, key)

if err != nil {

return "", err

}

return string(cipherArr), nil

}

//2.3DES加密解密

//----------------------------------

// 3DES加密字节数组，返回字节数组

func TripleDesEncrypt(originalBytes, key []byte) ([]byte, error) {

block, err := des.NewTripleDESCipher(key)

if err != nil {

return nil, err

}

originalBytes = PKCS5Padding(originalBytes, block.BlockSize())

// originalBytes = ZeroPadding(originalBytes, block.BlockSize())

blockMode := cipher.NewCBCEncrypter(block, key[:8])

cipherArr := make([]byte, len(originalBytes))

blockMode.CryptBlocks(cipherArr, originalBytes)

return cipherArr, nil

}

// 3DES解密字节数组，返回字节数组

func TripleDesDecrypt(cipherBytes, key []byte) ([]byte, error) {

block, err := des.NewTripleDESCipher(key)

if err != nil {

return nil, err

}

blockMode := cipher.NewCBCDecrypter(block, key[:8])

originalArr := make([]byte, len(cipherBytes))

blockMode.CryptBlocks(originalArr, cipherBytes)

originalArr = PKCS5UnPadding(originalArr)

// origData = ZeroUnPadding(origData)

return originalArr, nil

}

// 3DES加密字符串，返回base64处理后字符串

func TripleDesEncrypt2Str(originalText string, key []byte) (string, error) {

block, err := des.NewTripleDESCipher(key)

if err != nil {

return "", err

}

originalData := PKCS5Padding([]byte(originalText), block.BlockSize())

// originalData = ZeroPadding(originalData, block.BlockSize())

blockMode := cipher.NewCBCEncrypter(block, key[:8])

cipherArr := make([]byte, len(originalData))

blockMode.CryptBlocks(cipherArr, originalData)

cipherText := base64.StdEncoding.EncodeToString(cipherArr)

return cipherText, nil

}

// 3DES解密base64处理后的加密字符串，返回明文字符串

func TripleDesDecrypt2Str(cipherText string, key []byte) (string, error) {

cipherArr, \_ := base64.StdEncoding.DecodeString(cipherText)

block, err := des.NewTripleDESCipher(key)

if err != nil {

return "", err

}

blockMode := cipher.NewCBCDecrypter(block, key[:8])

originalArr := make([]byte, len(cipherArr))

blockMode.CryptBlocks(originalArr, cipherArr)

originalArr = PKCS5UnPadding(originalArr)

// origData = ZeroUnPadding(origData)

return string(originalArr), nil

}

//3.AES加密解密

//AES加密字节数组，返回字节数组

func AesEncrypt(originalBytes, key []byte) ([]byte, error) {

block, err := aes.NewCipher(key)

if err != nil {

return nil, err

}

blockSize := block.BlockSize()

originalBytes = PKCS5Padding(originalBytes, blockSize)

// originalBytes = ZeroPadding(originalBytes, block.BlockSize())

blockMode := cipher.NewCBCEncrypter(block, key[:blockSize])

cipherBytes := make([]byte, len(originalBytes))

// 根据CryptBlocks方法的说明，如下方式初始化crypted也可以

// crypted := originalBytes

blockMode.CryptBlocks(cipherBytes, originalBytes)

return cipherBytes, nil

}

//AES解密字节数组，返回字节数组

func AesDecrypt(cipherBytes, key []byte) ([]byte, error) {

block, err := aes.NewCipher(key)

if err != nil {

return nil, err

}

blockSize := block.BlockSize()

blockMode := cipher.NewCBCDecrypter(block, key[:blockSize])

originalBytes := make([]byte, len(cipherBytes))

// origData := cipherBytes

blockMode.CryptBlocks(originalBytes, cipherBytes)

originalBytes = PKCS5UnPadding(originalBytes)

// origData = ZeroUnPadding(origData)

return originalBytes, nil

}

//AES加密文本，返回对加密后字节数组进行base64处理后字符串

func AesEncryptString(originalText string, key []byte) (string, error) {

cipherBytes, err := AesEncrypt([]byte(originalText), key)

if err != nil {

return "", err

}

base64str := base64.StdEncoding.EncodeToString(cipherBytes)

return base64str, nil

}

//对Base64处理后的加密文本进行DES解密，返回解密后明文

func AesDecryptString(cipherText string, key []byte) (string, error) {

cipherBytes, \_ := base64.StdEncoding.DecodeString(cipherText)

cipherBytes, err := AesDecrypt(cipherBytes, key)

if err != nil {

return "", err

}

return string(cipherBytes), nil

}

//4.填充函数

//对称加密需要的填充函数

func PKCS5Padding(data []byte, blockSize int) []byte {

padding := blockSize - len(data)%blockSize

padtext := bytes.Repeat([]byte{byte(padding)}, padding)

return append(data, padtext...)

}

func PKCS5UnPadding(data []byte) []byte {

length := len(data)

// 去掉最后一个字节 unpadding 次

unpadding := int(data[length-1])

return data[:(length - unpadding)]

}

func ZeroPadding(data []byte, blockSize int) []byte {

padding := blockSize - len(data)%blockSize

padtext := bytes.Repeat([]byte{0}, padding)

return append(data, padtext...)

}

func ZeroUnPadding(data []byte) []byte {

return bytes.TrimRightFunc(data, func(r rune) bool {

return r == rune(0)

})

}

13.3 非对称加密算法

13.3.1 非对称加密算法的发展史

该思想最早由瑞夫·墨克（Ralph C. Merkle）在1974年提出，之后在1976年，惠特菲尔德·迪菲（Whitfield Diffie）与马丁·赫尔曼（Martin Hellman）两位斯坦福的学者以单向函数为基础，创建了DH密码交换算法。

RSA公钥加密算法是1977年美国麻省理工学院的Rivest（里维斯特）、Shamir和Adleman三位教授开发的。RSA取名来自开发他们三者的名字。RSA是目前最有影响力的公钥加密算法，它能够抵抗到目前为止已知的所有密码攻击，已被ISO推荐为公钥数据加密标准。

其他常见的公钥加密算法有：ElGamal、背包算法、Rabin（RSA的特例）、椭圆曲线加密算法（Elliptic Curve Cryptography, ECC）。

13.3.2 非对称加密（Asymmetric Cryptography）

非对称加密又叫做公开密钥加密（Public key cryptography）或公钥加密。

指加密和解密使用不同密钥的加密算法，也称为公私钥加密。

假设两个用户要加密交换数据，双方交换公钥，使用时一方用对方的公钥加密，另一方即可用自己的私钥解密。如果企业中有n个用户，企业需要生成n对密钥，并分发n个公钥。由于公钥是可以公开的，用户只要保管好自己的私钥即可，因此加密密钥的分发将变得 十分简单。同时，由于每个用户的私钥是唯一的，其他用户除了可以可以通过信息发送者的公钥来验证信息的来源是否真实，还可以确保发送者无法否认曾发送过该信息。

非对称加密的缺点是加解密速度要远远慢于对称加密，在某些极端情况下，甚至能比非对称加密慢上1000倍。

公钥加密，需要两个密钥，一个是公开密钥，另一个是私有密钥；一个用作加密的时候，另一个则用作解密。

使用其中一个密钥把明文加密后所得的密文，只能用相对应的另一个密钥才能解密得到原本的明文；甚至连最初用来加密的密钥也不能用作解密。由于加密和解密需要两个不同的密钥，故被称为非对称加密。

不同于加密和解密都使用同一个密钥的对称加密。虽然两个密钥在数学上相关，但如果知道了其中一个，并不能凭此计算出另外一个。因此其中一个可以公开，称为公钥，任意向外发布；不公开的密钥为私钥，必须由用户自行严格秘密保管，绝不通过任何途径向任何人提供。

公钥用来加密消息、验证签名；私钥用来解密信息和进行签名。加密消息的密钥是不能解密消息的。

13.3.3 对称加密与非对称加密的区别

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法类型 | 特点 | 优势 | 缺陷 | 代表算法 |
| 对称加密 | 加解密密钥相同或可推算 | 计算效率高，加密强度高 | 需提前共享密钥；易泄露 | DES、3DES、AES、IDEA |
| 非对称加密 | 加解密密钥不相关 | 无需提前共享密钥 | 计算效率低，中间人攻击可能性低 | RSA、ElGamal、椭圆曲线系列算法ECC |

13.3.4 核心代码

//1.生成RSA密钥文件

func main() {

var bits int

flag.IntVar(&bits, "b", 2048, "密钥长度，默认为1024位")

if err := GenRsaKey(bits); err != nil {

log.Fatal("密钥文件生成失败！")

}

log.Println("密钥文件生成成功！")

}

func GenRsaKey(bits int) error {

// 生成私钥文件

privateKey, err := rsa.GenerateKey(rand.Reader, bits)

if err != nil {

return err

}

derStream := x509.MarshalPKCS1PrivateKey(privateKey)

block := &pem.Block{

Type: "私钥",

Bytes: derStream,

}

file, err := os.Create("private.pem")

if err != nil {

return err

}

err = pem.Encode(file, block)

if err != nil {

return err

}

// 生成公钥文件

publicKey := &privateKey.PublicKey

derPkix, err := x509.MarshalPKIXPublicKey(publicKey)

if err != nil {

return err

}

block = &pem.Block{

Type: "公钥",

Bytes: derPkix,

}

file, err = os.Create("public.pem")

if err != nil {

return err

}

err = pem.Encode(file, block)

if err != nil {

return err

}

return nil

}

//2.RSA加密与解密

var decrypted string

var privateKey, publicKey []byte

func init() {

var err error

flag.StringVar(&decrypted, "d", "", "加密过的数据")

flag.Parse()

publicKey, err = ioutil.ReadFile("public.pem")

if err != nil {

os.Exit(-1)

}

privateKey, err = ioutil.ReadFile("private.pem")

if err != nil {

os.Exit(-1)

}

}

// 加密字节数组，返回字节数组

func RsaEncrypt(origData []byte) ([]byte, error) {

block, \_ := pem.Decode(publicKey)

if block == nil {

return nil, errors.New("public key error")

}

pubInterface, err := x509.ParsePKIXPublicKey(block.Bytes)

if err != nil {

return nil, err

}

pub := pubInterface.(\*rsa.PublicKey)

return rsa.EncryptPKCS1v15(rand.Reader, pub, origData)

}

// 解密字节数组，返回字节数组

func RsaDecrypt(ciphertext []byte) ([]byte, error) {

block, \_ := pem.Decode(privateKey)

if block == nil {

return nil, errors.New("private key error!")

}

priv, err := x509.ParsePKCS1PrivateKey(block.Bytes)

if err != nil {

return nil, err

}

return rsa.DecryptPKCS1v15(rand.Reader, priv, ciphertext)

}

// 加密字符串，返回base64处理的字符串

func RsaEncryptString(origData string) (string, error) {

block, \_ := pem.Decode(publicKey)

if block == nil {

return "", errors.New("public key error")

}

pubInterface, err := x509.ParsePKIXPublicKey(block.Bytes)

if err != nil {

return "", err

}

pub := pubInterface.(\*rsa.PublicKey)

cipherArr, err := rsa.EncryptPKCS1v15(rand.Reader, pub, []byte(origData))

if err != nil {

return "", err

} else {

return base64.StdEncoding.EncodeToString(cipherArr), nil

}

}

// 解密经过base64处理的加密字符串，返回加密前的明文

func RsaDecryptString(cipherText string) (string, error) {

block, \_ := pem.Decode(privateKey)

if block == nil {

return "", errors.New("private key error!")

}

priv, err := x509.ParsePKCS1PrivateKey(block.Bytes)

if err != nil {

return "", err

}

cipherArr, \_ := base64.StdEncoding.DecodeString(cipherText)

originalArr, err := rsa.DecryptPKCS1v15(rand.Reader, priv, cipherArr)

if err != nil {

return "", err

} else {

return string(originalArr), nil

}

}

13.4 椭圆曲线加密算法ECC和椭圆曲线数字签名算法ECDSA

13.4.1 椭圆曲线密码学概述

椭圆曲线密码学（Elliptic curve cryptography，缩写为 ECC），是基于椭圆曲线数学理论实现的一种非对称加密算法。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。

比特币使用椭圆曲线算法生成公钥和私钥，选择的是secp256k1曲线。与RSA（Ron Rivest，Adi Shamir，Len Adleman三位天才的名字）一样，ECC（Elliptic Curves Cryptography，椭圆曲线加密）也属于公开密钥算法。

ECC与RSA算法的优势对比

椭圆曲线公钥系统是代替RSA的强有力的竞争者。

与经典的RSA、DSA等公钥密码体制相比，椭圆密码体制有以下优点：

（1）安全性能更高（ECC可以使用更短的密钥）：

160位ECC加密算法的安全强度相当于1024位RSA加密；

210位ECC加密算法的安全强度相当于2048位RSA加密。

（2）处理速度快：计算量小，处理速度快 在私钥的处理速度上（解密和签名），ECC远 比RSA、DSA快得多。

（3）存储空间占用小： ECC的密钥尺寸和系统参数与RSA、DSA相比要小得多， 所以占用的存储空间小得多。

（4）带宽要求低使得ECC具有广泛的应用前景。

ECC的这些特点使它必将取代RSA，成为通用的公钥加密算法。

13.4.2 数字签名

所谓数字签名(Digital Signature)（又称公开密钥数字签名、电子签章）,是一种类似写在纸上的普通的物理签名，但是使用了公钥加密领域的技术实现，用于鉴别数字信息的方法。

一套数字签名通常定义两种互补的运算，一个用于签名，另一个用于验证。数字签名可以验证数据的来源，可以验证数据传输过程中是否被修改。

数字签名由两部分组成：第一部分是使用私钥（签名密钥）从消息（交易）创建签名的算法； 第二部分是允许任何人验证签名的算法。

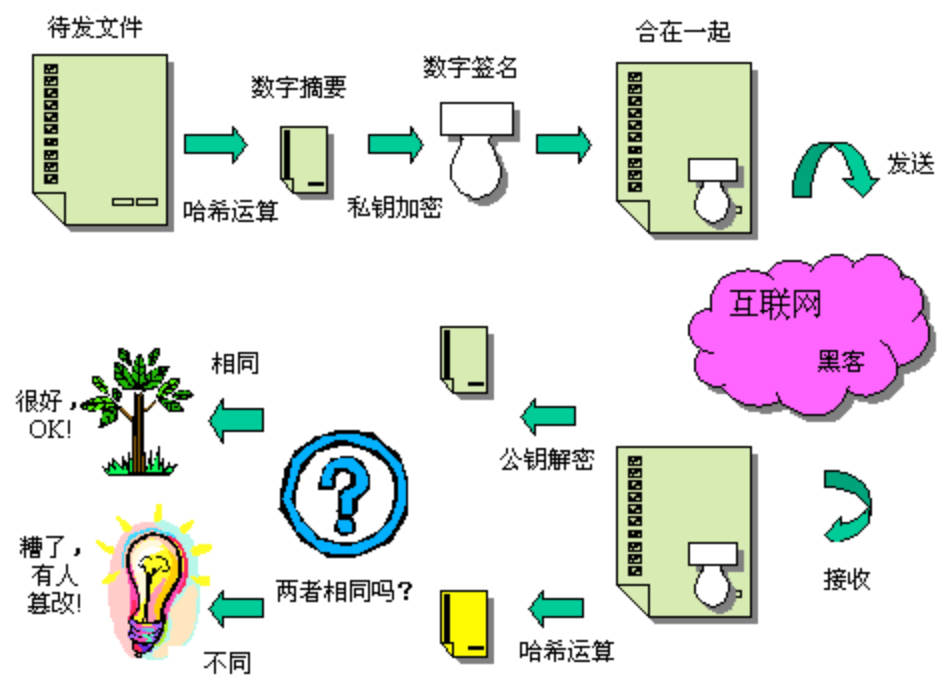


图 13.6

数字签名应该满足如下要求：

* 签名不可伪造性；
* 签名不可抵赖的；
* 签名可信性，签名的识别和应用相对容易，任何人都可以验证签名的有效性；
* 签名是不可复制的，签名与原文是不可分割的整体；
* 签名消息不可篡改，因为任意比特数据被篡改，其签名便被随之改变，那么任何人可以验证而拒绝接受此签名。

数字签名和验证过程

1.只有转账人才能生成的一段防伪造的字符串。通过验证该字符串，一方面证明该交易是转出方本人发起的，另一方面证明交易信息在传输过程中没有被更改。

2.数字签名由：数字摘要和非对称加密技术组成。数字摘要把交易信息hash成固定长度的字符串；再用私钥对hash后的交易信息进行加密形成数字签名。

3.交易中，需要将完整的交易信息和数字签名一起广播给矿工。矿工节点用转账人公钥对签名验证，验证成功说明该交易确实是转账人发起的；矿工节点将交易信息进行hash后与签名中的交易信息摘要进行比对，如果一致，则说明交易信息在传输过程中没有被篡改。

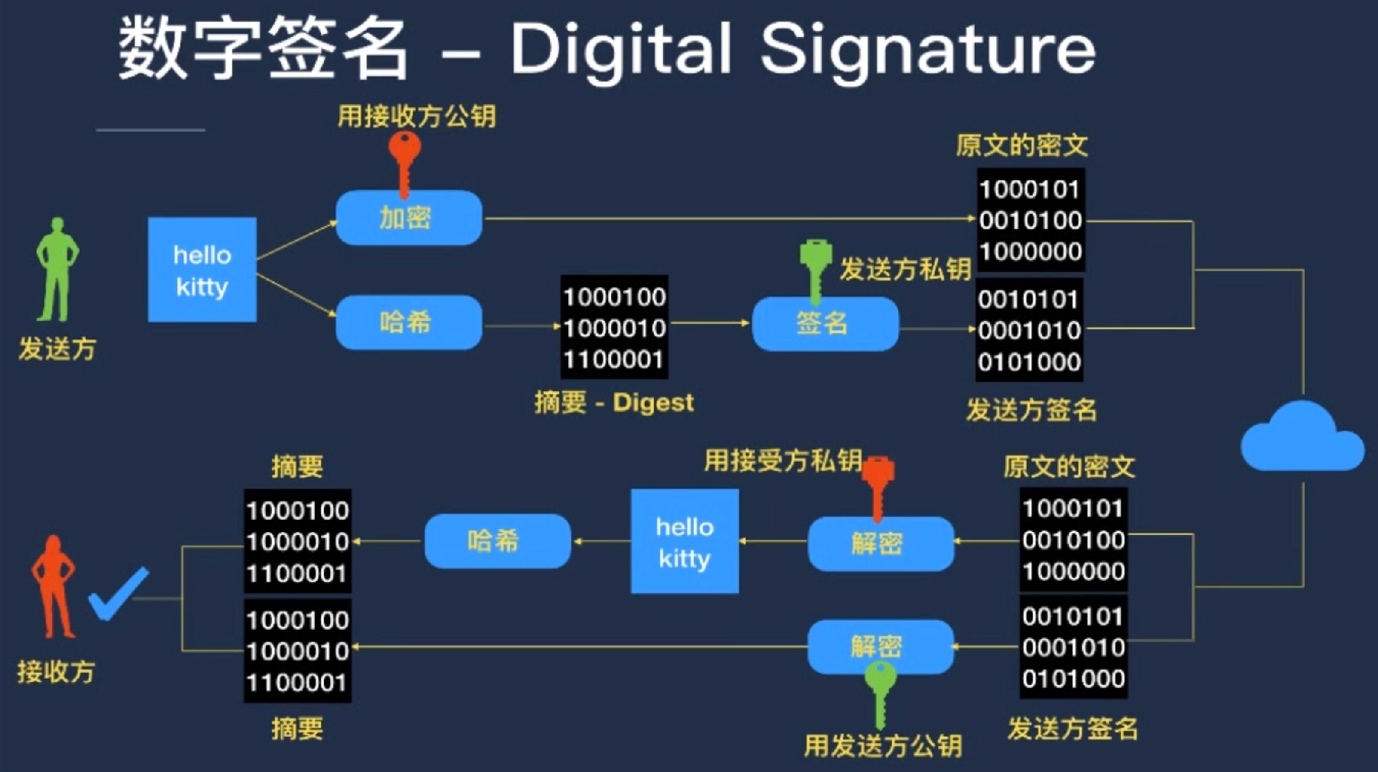


图 13.7

13.4.3 ECC数字签名(ECDSA)核心代码

//生成私钥和公钥，生成的私钥为结构体ecdsa.PrivateKey的指针

func NewKeyPair() (ecdsa.PrivateKey, []byte) {

//生成secp256椭圆曲线

curve := elliptic.P256()

//产生的是一个结构体指针，结构体类型为ecdsa.PrivateKey

private, err := ecdsa.GenerateKey(curve, rand.Reader)

if err != nil {

log.Panic(err)

}

fmt.Printf("私钥：%x\n", private)

fmt.Printf("私钥X：%x\n", private.X.Bytes())

fmt.Printf("私钥Y：%x\n", private.Y.Bytes())

fmt.Printf("私钥D：%x\n", private.D.Bytes())

//x坐标与y坐标拼接在一起，生成公钥

pubKey := append(private.X.Bytes(), private.Y.Bytes()...)

//打印公钥，公钥用16进制打印出来长度为128，包含了x轴坐标与y轴坐标。

fmt.Printf("公钥：%x \n", pubKey)

return \*private, pubKey

}

//生成签名的DER格式

func MakeSignatureDerString(r, s string) string {

// 获取R和S的长度

lenSigR := len(r) / 2

lenSigS := len(s) / 2

// 计算DER序列的总长度

lenSequence := lenSigR + lenSigS + 4

// 将10进制长度转16进制字符串

strLenSigR := DecimalToHex(int64(lenSigR))

strLenSigS := DecimalToHex(int64(lenSigS))

strLenSequence := DecimalToHex(int64(lenSequence))

// 拼凑DER编码

derString := "30" + strLenSequence

derString = derString + "02" + strLenSigR + r

derString = derString + "02" + strLenSigS + s

derString = derString + "01"

return derString

}

//验证签名1

func VerifySig(pubKey, message []byte, r, s \*big.Int) bool {

curve := elliptic.P256()

//公钥的长度

keyLen := len(pubKey)

//前一半为x轴坐标，后一半为y轴坐标

x := big.Int{}

y := big.Int{}

x.SetBytes(pubKey[:(keyLen / 2)])

y.SetBytes(pubKey[(keyLen / 2):])

rawPubKey := ecdsa.PublicKey{curve, &x, &y}

//根据交易哈希、公钥、数字签名验证成功。ecdsa.Verify func Verify(pub \*PublicKey, hash []byte, r \*big.Int, s \*big.Int) bool

res := ecdsa.Verify(&rawPubKey, message, r, s)

return res

}

//验证签名2

func VerifySignature(pubKey, message []byte, r, s string) bool {

curve := elliptic.P256()

//公钥的长度

keyLen := len(pubKey)

//前一半为x轴坐标，后一半为y轴坐标

x := big.Int{}

y := big.Int{}

x.SetBytes(pubKey[:(keyLen / 2)])

y.SetBytes(pubKey[(keyLen / 2):])

rawPubKey := ecdsa.PublicKey{curve, &x, &y}

//根据交易哈希、公钥、数字签名验证成功。ecdsa.Verify func Verify(pub \*PublicKey, hash []byte, r \*big.Int, s \*big.Int) bool

rint := big.Int{}

sint := big.Int{}

rByte, \_ := hex.DecodeString(r)

sByte, \_ := hex.DecodeString(s)

rint.SetBytes(rByte)

sint.SetBytes(sByte)

//fmt.Println("------", rint.SetBytes(rByte))

//fmt.Println("------", sint.SetBytes(sByte))

res := ecdsa.Verify(&rawPubKey, message, &rint, &sint)

return res

}

//验证过程

func main() {

//1、生成签名

fmt.Println("1、生成签名-------------------------------")

//调用函数生成私钥与公钥

privKey, pubKey := cryptotool.NewKeyPair()

//信息的哈希

msg := sha256.Sum256([]byte("hello world"))

//根据私钥和信息的哈希进行数字签名，产生r和s

r, s, \_ := ecdsa.Sign(rand.Reader, &privKey, msg[:])

//生成r、s字符串

fmt.Println("-------------------------------")

strSigR := fmt.Sprintf("%x", r)

strSigS := fmt.Sprintf("%x", s)

fmt.Println("r、s的10进制分别为：", r, s)

fmt.Println("r、s的16进制分别为：", strSigR, strSigS)

//r和s拼接在一起，形成数字签名的der格式

signatureDer := cryptotool.MakeSignatureDerString(strSigR, strSigS)

//打印数字签名的16进制显示

fmt.Println("数字签名DER格式为：", signatureDer)

fmt.Println()

//2、签名验证过程

fmt.Println("2、签名验证过程-------------------------------")

res := cryptotool.VerifySig(pubKey, msg[:], r, s)

fmt.Println("签名验证结果：" , res)

res = cryptotool.VerifySignature(pubKey, msg[:], strSigR, strSigS)

fmt.Println("签名验证结果：" , res)

}

13.5 字符编码/解码

13.5.1 Base64

Base64就是一种基于64个可打印字符来表示二进制数据的方法

Base64使用了26个小写字母、26个大写字母、10个数字以及两个符号（例如“+”和“/”），用于在电子邮件这样的基于文本的媒介中传输二进制数据。

Base64通常用于编码邮件中的附件。

Base64字符集：ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

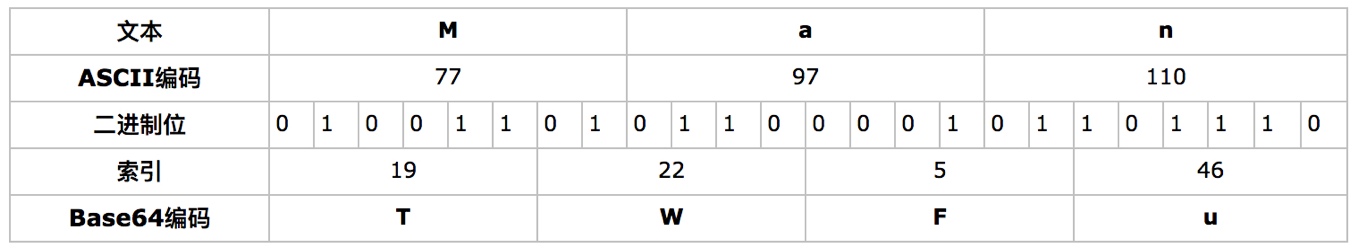


图 13.8

Base64的步骤：

1. 将每个字符转成ASCII编码（10进制）

2. 将10进制编码转成2进制编码

3. 将2进制编码按照6位一组进行平分

4. 将6位一组的2进制数高位补零，然后转成10进制数

5. 将10进制数作为索引，从Base64编码表中查找字符

6. 每3个字符的文本将编码为4个字符长度（3\*8=4\*6）

a. 若文本为3个字符，则正好编码为4个字符长度；

b. 若文本为2个字符，则编码为3个字符，由于不足4个字符，则在尾部用一个“=”补齐；

c. 若文本为1个字符，则编码为2个字符，由于不足4个字符，则在尾部用两个“=”补齐。

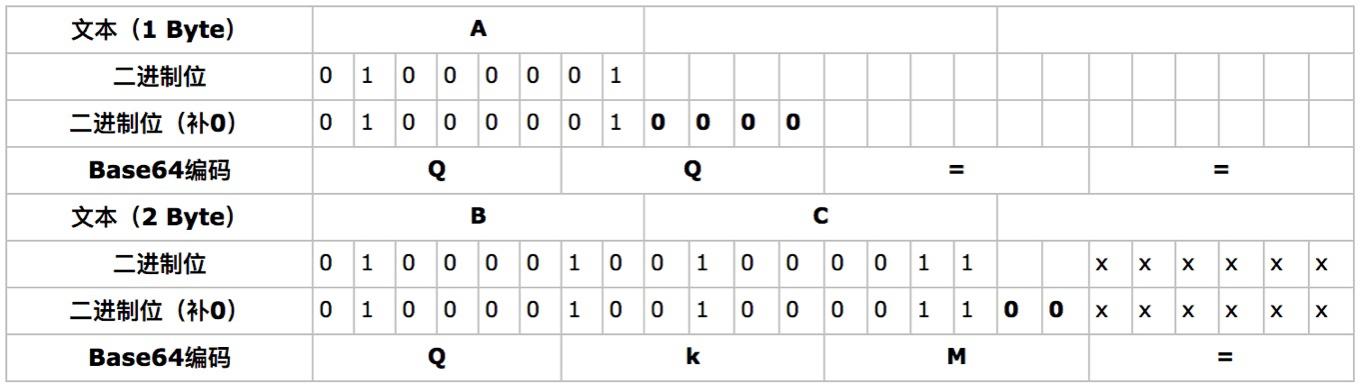


图 13.9

Base64相关的函数：

func Base64EncodeString(str string) string {

return base64.StdEncoding.EncodeToString([]byte(str))

}

func Base64DecodeString(str string) string {

result, \_ := base64.StdEncoding.DecodeString(str)

return string(result)

}

13.5.2 Base58

Base58是一种基于文本的二进制编码格式，用在比特币和其它的加密货币中。这种编码格式不仅实现了数据压缩，保持了易读性，还具有错误诊断功能。

Base58是Base64编码格式的子集，同样使用大小写字母和10个数字，但舍弃了一些容易错读和在特定字体中容易混淆的字符。

Base58不含Base64中的0（数字0）、O（大写字母o）、l（小写字母L）、I（大写字母i），以及“+”和“/”两个字符。目的就是去除容易混淆的字符。

简而言之，Base58就是由不包括（0，O，l，I）的大小写字母和数字组成。

比特币的Base58字母表：123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

13.5.3 Base58Check

Base58Check是一种常用在比特币中的Base58编码格式，增加了错误校验码来检查数据在转录中出现的错误。

校验码长4个字节，添加到需要编码的数据之后。

校验码是从需要编码的数据的哈希值中得到的，所以可以用来检测并避免转录和输入中产生的错误。

使用Base58check编码格式时，编码软件会计算原始数据的校验码并和结果数据中自带的校验码进行对比。二者不匹配则表明有错误产生，那么这个Base58Check格式的数据就是无效的。例如，一个错误比特币地址就不会被钱包认为是有效的地址，否则这种错误会造成资金的丢失。

为了使用Base58Check编码格式对数据（数字）进行编码，首先我们要对数据添加一个称作“版本字节”的前缀，这个前缀用来明确需要编码的数据的类型。

例如，比特币地址的前缀是0（十六进制是0x00），而对私钥编码时前缀是128（十六进制是0x80）。

附加：

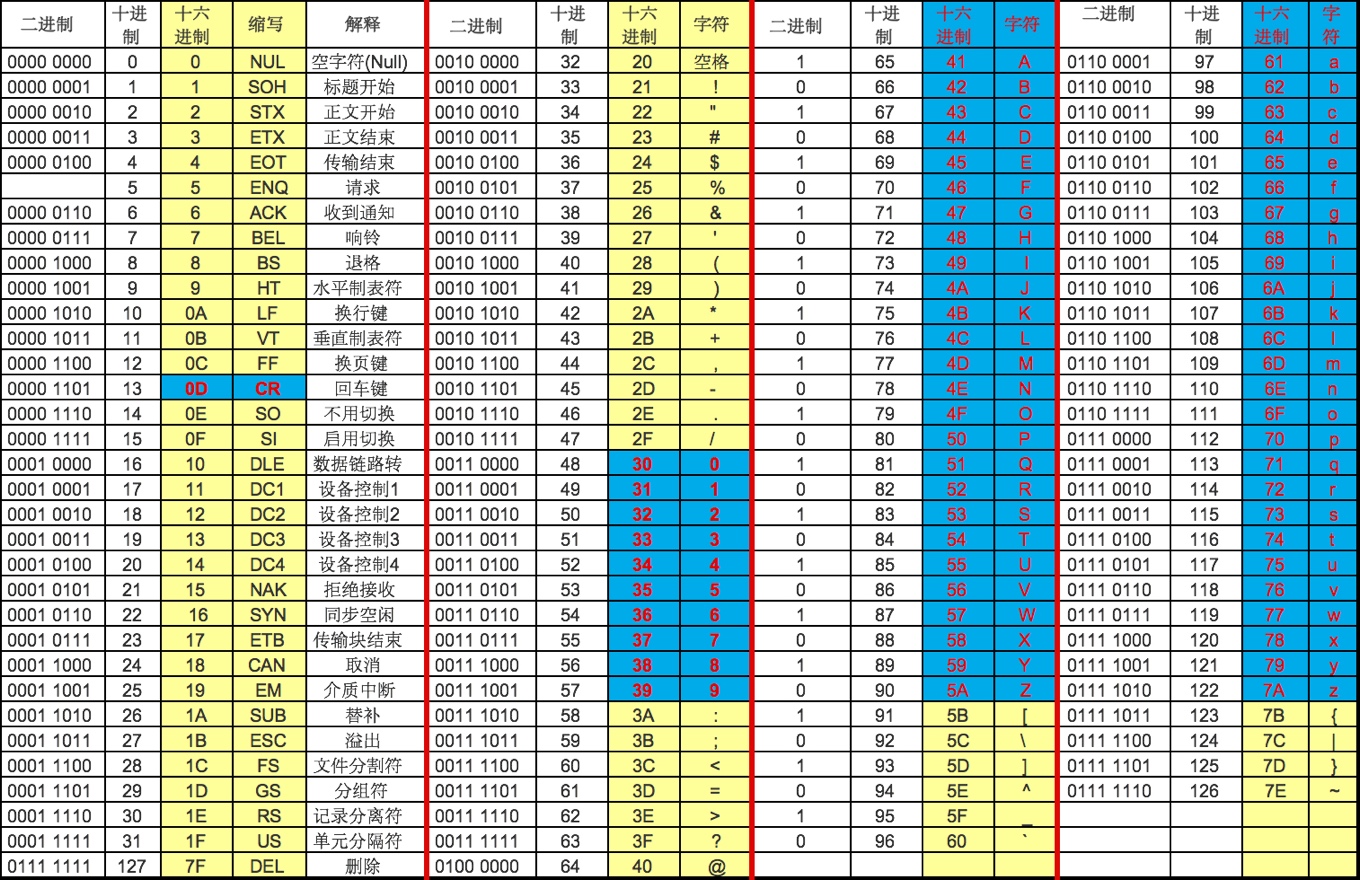


图 13.10

13.5.4 核心代码

var base58Alphabets = []byte("123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz")

// Base58Encode encodes a byte array to Base58

func Base58Encode(input []byte) []byte {

var result []byte

x := big.NewInt(0).SetBytes(input)

base := big.NewInt(int64(len(base58Alphabets)))

zero := big.NewInt(0)

mod := &big.Int{}

for x.Cmp(zero) != 0 {

x.DivMod(x, base, mod)

result = append(result, base58Alphabets[mod.Int64()])

}

// https://en.bitcoin.it/wiki/Base58Check\_encoding#Version\_bytes

if input[0] == 0x00 {

result = append(result, base58Alphabets[0])

}

ReverseBytes(result)

return result

}

// Base58Decode decodes Base58-encoded data

func Base58Decode(input []byte) []byte {

result := big.NewInt(0)

for \_, b := range input {

charIndex := bytes.IndexByte(base58Alphabets, b)

result.Mul(result, big.NewInt(58))

result.Add(result, big.NewInt(int64(charIndex)))

}

decoded := result.Bytes()

if input[0] == base58Alphabets[0] {

decoded = append([]byte{0x00}, decoded...)

}

return decoded

}

[第十三章 Go语言加密算法 1](#_Toc29641)

[13.1 哈希算法 1](#_Toc8383)

[13.1.1 Hash的定义 1](#_Toc6326)

[13.1.2 流行的哈希算法 1](#_Toc23928)

[13.1.3 Hash与加密解密的区别 3](#_Toc19888)

[13.1.4 SHA256 6](#_Toc13378)

[13.1.5 核心代码 7](#_Toc17103)

[13.2 对称加密算法 9](#_Toc22666)

[13.1.6 对称加密的概述 9](#_Toc31392)

[13.1.7 DES和TripleDES算法 10](#_Toc13370)

[13.1.8 AES加密 11](#_Toc2760)

[13.1.9 AES的加密模式 11](#_Toc5179)

[13.1.10 填充方式 12](#_Toc19197)

[13.1.11 核心代码 12](#_Toc32273)

[13.3 非对称加密算法 16](#_Toc11445)

[13.1.12 非对称加密算法的发展史 17](#_Toc8026)

[13.1.13 非对称加密（Asymmetric Cryptography） 17](#_Toc2345)

[13.1.14 对称加密与非对称加密的区别 17](#_Toc3634)

[13.1.15 RSA原理 18](#_Toc24742)

[13.1.16 核心代码 18](#_Toc2508)

[13.4 椭圆曲线加密算法ECC和椭圆曲线数字签名算法ECDSA 22](#_Toc7496)

[13.1.17 椭圆曲线密码学概述： 22](#_Toc9406)

[13.1.18 数字签名的概念 22](#_Toc15495)

[13.1.19 数字签名应该满足的要求 23](#_Toc20186)

[13.1.20 数字签名和验证过程 23](#_Toc7751)

[13.1.21 ECC数字签名(ECDSA)核心代码 24](#_Toc8833)

[13.5 字符编码/解码 27](#_Toc27973)

[13.1.22 Base64 27](#_Toc14799)

[13.1.23 Base58 29](#_Toc29724)

[13.1.24 Base58Check 29](#_Toc12836)

[13.1.25 核心代码 30](#_Toc2867)