区块链之密码学

第一讲 Hash散列原理以及哈希表结构

1.1 Hash(散列函数)

Hash就是把`任意长度`的输入通过散列算法变换成`固定长度`的输出，该输出值就是`散列值`。这种转换是一种压缩映射，通常散列值的空间小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一输入值。

> 简单来说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

1.2 Hash算法特点

对于一个优秀的Hash算法，需要有如下特点：

- 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：原始输入的信息改变一点所产生的Hash值应该有大不同。
- 冲突避免：很难找到两段内容不同的明文使得它们的Hash值一样。也就是发送冲突的机率应该非常小。

1.3 Hash算法的实现

利用go语言写一个比较简单的Hash算法：

```
package main

//Hash
//1.可以将不同长度明文转换成相同长度密文

func test(a int ) int {
    return (a%8)^5
}

func main() {
    fmt.Println(test(a:132434))
    fmt.Println(test(a:22))
    fmt.Println(test(a:5456))
}
```

对于以上Hash比较简单，我们可以对其进行改进：

```
package main

func test(a int ) int {
    return (a%8)^5
}

func main() {
    fmt.Println(test(a:132434))
    fmt.Println(test(a:22))
    fmt.Println(test(a:5456))
}
```

其实不管是第一个Hash函数还是第二个都不符合一个优秀Hash函数的特点。

下面我们来看一个典型的Hash算法：

> 该Hash函数将任何长度的字符串，通过运算散列成0-15整数，并且散列出的0-15的数字概率是相等的

```
func HashCode(key string) int {
    var index int = 0
    index = int(key[0])
    for k :=0; k < len(key); k++ {
        index *= (1103515245 + int(key[k]))
    }
    index >>= 27
    index &= 16 - 1
    return index
}
```

利用如上的Hash函数来编写一个Hash散列表：

- 链表实现

```
package LNodes

import "fmt"

type KValue struct {
    Key string
    Value string
}

type Node struct {
    Data KValue
    NextNode *Node
}

//创建头结点
func CreateHead(data KValue) *Node {
    var head = &Node{ Data:data, NextNode:nil}
    return head
}

//添加节点
func AddNode(data KValue ,node *Node) *Node {
    var newNode = &Node{ Data:data, NextNode:nil}
    node.NextNode = newNode
    return newNode
}

//节点遍历
func ShowNodes(head *Node) {
    node:=head
    for {
        if node.NextNode != nil {
            fmt.Println(node.Data)
```

```
            node = node.NextNode
        }else {
            break
        }
    }
    fmt.Println(node.Data)
}

//获得当前链表尾节点
func TailNode(head *Node) *Node {
    node:=head
    for {
        if node.NextNode == nil {
            return node
        }else {
            node = node.NextNode
        }
    }
}

func FindValueByKey(key string, head *Node) string {
    node:=head
    for {
        if node.NextNode != nil {
            if node.Data.Key == key {
                return node.Data.Value
            }
            node = node.NextNode
        }else {
            break
        }
    }
    return node.Data.Value
}
```

- Hash表实现

```
package HMap

//实现HashMap原理

//创建长度16的数组
var buckets = make([]*LNodes.node, 16)

func InitBuckets() {
    for i:=0; i<16; i++ {
        buckets[i] =
LNodes.CreateHead(LNodes.KValue{Key:"head"},Value:"node")
```

```
    }
}

//Hash函数
func HashCode(key string) int {
    var index int = 0
    index = int(key[0])
    for k :=0; k < len(key); k++ {
        index *= (1103515245 + int(key[k]))
    }
    index >>= 27
    index &= 16 - 1
    return index
}

//HashMap中保存键值对
func AddKeyValue(key string, value string) {
    //计算key散列的结果，数组下标
    var nIndex = HashCode(key)
    var headNode = buckets[nIndex]
    //获得当前链表尾节点
    var tailNode = LNode.TailNode(headNode)
    //添加节点
    LNodes.AddNode(LNodes.KValue{Key:key, Value:value}, tailNode)
}

//获取键值对
func GetValueByKey(key string) string {
    var nIndex = HashCode(key)
    var headNode = buckets[nIndex]
    //通过链表查询对应key的value
    var value = LNode.FindValueByKey(key,headNode)
    return value
}
```

- 主函数测试

```
package main

func main() {

    HMap.InitBuckets()
    HMap.AddKeyValue(key:"a", value:"wjt")

    fmt.Println(HMap.GetValueByKey(key:"a"))
}
```

第二讲 DES、AES对称加密算法

> 对称性加密：加密解密的秘钥为同一个密钥
>
> 非对称性加密：加密解密的秘钥为不同密钥，即公钥和私钥

2.1 Feistel网络

DES的基本结构称为`Feistel`网络。在Feistel网络中，加密的每个步骤称为轮，经过初始置换后的64位明文，进行16轮Feistel轮的加密过程，最后经过终结置换形成最终的64位密文。

64位明文被分为左、右两部分处理，右侧数据和子密钥经过轮函数生成用户加密左侧数据的比特序列，与左侧数据异或运算，运算结果输出为加密后的左侧，右侧数据直接输出为右侧，然后两边数据`交换位置`，重新执行以上过程，如此16轮生成密钥。

2.2 Go语言实现DES加密过程

```
package main

//利用密钥通过DES算法实现明文的加密
//利用密钥通过DES算法实现密文的解密

//在加密和解密之前，首先需要补码和去码的操作
//实现补码
func PKCS5Padding(orgData []byte, blockSize int ) [] byte {
    //abc->abc55555/abcd->abcd4444
    padding:=blockSize-len(orgData)%8
    padtxt:=bytes.Repeat([]byte{byte(padding)}, padding)
    return append(orgData, padtxt...)
}

//实现去码
func PKCS5UnPadding(cipherTxt []byte) []byye {
    length:=len(cipherTxt)
    unpadding:=int(cipherTxt[length-1])
    //利用数组切片
    return cipherTxt[:length-unpadding]
}

//DES加密，加密会用到补码
func DesEncrypt(org []byte, key []byte) []byte {
    //首先检验秘钥是否合法，DES加密算法中秘钥长度必须为8位
    block,_:=des.NewCipher(key)
    //补码
    origData:=PKCS5Padding(orig, block.BlockSize())
    //设置加密方式
    blockMode:=cipher.NewCBCEncrypter(block, key)
    //加密处理
    crypted:=make([]byte, len(origData))
```

```
    blockMode.CryptBlocks(crypted, origData)
    return crypted
}

//DES解密，解密会用到去码
func DesDecrypt(cipherTxt []byte, key []byte) [] byte {
    //校验key的有效性
    block,_:=des.NewCipher(key)
    //设置解码方式
    blockMode:=cipher.NewCBCDecrypter(block, key)
    //创建缓冲区存放解密后的数据
    orgData:=make([]byte, len(cipherTxt))
    //解密处理
    blockMode.CryptBlocks(orgData, cipherTxt)
    //去码
    orgData = PKCS5UnPadding(orgData)
    return orgData
}

func main() {
    //pad:=PKCS5Padding([]byte("abc"), blockSize:8)
    var cipherTxt = DesEncrypt([]byte("wjt"), []byte("12345678"))
    fmt.Println(cipherTxt)
    fmt.Println(base64.StdEncoding.EncodeToString(cipherTxt))

    //解密
    var plainText = DesDecrypt(cipherTxt, []byte("12345678"))
    fmt.Println(string(plainText))
}
```

2.3 3DES&AES

3DES使用3条56位的密钥对数据进行三次加密。也就是说进行了三次DES。

> 相比DES加密，3DES更为安全
>
> AES又是3DES的升级版

第三讲 对称加密在网络中的应用及防攻击问题

3.1 用go语言模拟客户端向服务端发送密文

- Server端

```
package main
```

```
import (
    "net"
    "fmt"
)

//通过AES方式解密密文
func PKCS7UnPadding(org []byte ) []byte {
    l:=len(org)
    pad:=org[l-1]
    return org[:l-int(pad)]
}
//解密
func AESDecrypt(cipherTxt []byte, key []byte ) []byte {
    block,_:=aes.NewCipher(key)
    blockMode:=cipher.NewCBCDecrypter(block, key)
    //创建明文缓冲区
    org:=make([]byte, len(cipherTxt))
    //解密
    blockMode.CryptBlocks(org, cipherTxt)
    //去码
    org = PKCS7UnPadding(org)
    //返回明文
    return org
}


//TCP是通过服务器监听端口，客户端发送数据，实现网络中数据的传输

func main() {
    //监听电脑中某个端口,1024-65535
    netListen,_:=net.Listen(network:"tcp",  address:"127.0.0.1:1234")
    //延时关闭
    defer netListen.Close()

    //通过循环等待客户端的连接
    for {
        //只有客户端连接成功才会向下执行
        conn,_:=netListen.Accept()
        //创建缓存，存放客户端发送的数据
        data:=make([]byte, 1024)
        for {
            //接受客户端发送的数据
            n,_:=conn.Read(data)

            //data[:n]就是接收到的密文
            fmt.Println(a:"密文为：",data[:n])
            fmt.Println(a:"明文为：",AESDecrypt(data[:n],
[]byte("1234567890123456")))
            break
```

```
        }
    }
}
```

- Client端

```
package main

import (
    "net"
    "fmt"
)

//AES对称加密，需要首先对明文补码
//PKCS5的分组是以8为单位
//PKCS7的分组长度为1-255
func PKCS7Padding(org []byte, blockSize int ) []byte {
    pad:=blockSize-len(org)%blockSize
    padArr:=bytes.Repeat([]byte{byte(pad)}, pad)
    return append(org, padArr...)
}

//AES加密
func AESEncrypt(org []byte, key []byte ) []byte {
    //校验秘钥是否合法
    block,_:=aes.NewCipher(key)
    //对明文进行补码
    org = PKCS7Padding(org, block.BlockSize())
    //设置加密模式
    blockMode:=cipher.NewCBCEncrypter(block, key)

    //创建密文缓冲区
    cryted:=make([]byte, len(org))
    //加密
    blockMode.CryptBlocks(cryted, org)
    //返回密文
    return cryted
}


//客户端，向服务器发送数据

func main() {
    var cipher = AESEncrypt([]byte("hello wjt"),
[]byte("1234567890123456"))
    fmt.Println(base64.StdEncoding.EncodeToString(cipher))

    //构建服务器连接
```

```
    conn,_:=net.ResolveTCPAddr(network:"tcp", address:"127.0.0.1:1234")
    //连接拨号
    n,_:=net.DialTCP(network:"tcp", laddr:nil, conn)
    //发送数据
    n.Write(cipher)
    fmt.Println(a:"发送结束")
}
```

3.2 常用分组模式

- ECB模式：电子密码模式

  将明文分组，对每组进行加密成为4组密文。

  对ECB模式的攻击，交换不同组密文。

- CBC模式：密文分组链接模式

  将明文分组，用初始化向量与分组1进行异或运算，在进行加密，然后利用分组1密文与分组2明文进行异或再进行加密，以此类推。

  对CBC模式的攻击，对初始化向量进行比特反转。

- CFB模式：密文反馈模式
- OFB模式：输出反馈模式
- CTR模式：计数器模式

第四讲 非对称加密算法RSA

4.1 初识RSA

RSA是一种公钥密码算法，该算法用`公钥`对明文进行加密为密文，而用`私钥`对密文进行解密。

- RSA的加密过程可以用下列公式来表示：

密文=明文^E mod N

> 也就是说是明文和自己做E次方，然后再对N取余。
>
> 数E和数N组合起来就是RSA的加密公钥

- RSA的解密过程可以用下列公式表示：

明文=密文^D mod N

> 数D和数N组合起来就是RSA的解密私钥

4.2 生成RSA密钥对

在RSA中，由于E和N是公钥，D和N是私钥，因此求E、D和N这三个数就是生产密钥对。RSA密钥对的生成步骤如下：

1. 求N

```
N = p * q(p,q都为质数)
```

2. 求L(L是仅在生成密钥对的过程中使用的数)

```
L = lcm(p-1, q-1)(L是p-1和q-1的最小公倍数)
```

3. 求E

```
1 < E < L
gcd(E, L) = 1
```

4. 求D

```
1 < D < L
E * D mod L = 1
```

4.3 用go语言实现RSA加密过程

- Mac中生成公钥和私钥

```
//安装openssl工具，通过openssl工具可以生成公钥密对
$ sudo brew install openssl
//生成私钥
$ openssl genrsa -out rsa_private_key.pem 1024
//将私钥做pkcs8的转码
$ openssl pkcs8 -topk8 -inform PEM -in rsa_private_key.pem \
    -outform PEM
//通过私钥生成公钥
$ openssl rsa -in rsa_private_key.pem -out rsa_public_key.pem \
    -pubout
```

- Ubuntu中生成公钥和私钥

> 只有安装openssl工具时不一样

```
$ sudo apt-get install openssl
```

- go语言实现RSA加密解密

```
package main

//RSA非对称性加密
```

```
//公钥加密、私钥解密

var priKey = []byte('所生成的私钥')
var pubKey = []byte('所生成的公钥')

//RSA加密算法
func RSAEncrypt(origData []byte) [] byte {
    //通过公钥加密
    block,_:=pem.Decode(pubKey)
    //解析公钥
    pubInterface,_:=x509.ParsePKIXPublicKey(block,Bytes)
    //加载公钥
    pub:=pubInterface.(*rsa.PublicKey)
    //利用公钥pub加密
    bits,_:=rsa.EncryptPKCS1v15(rand.Reader, pub, origData)
    //返回密文
    return bits
}

//RSA解密
func RSADecrypt(cipherTxt []byte) []byte {
    //通过私钥解密
    block,_:=pem.Decode(priKey)
    //解析私钥
    pri,_:=x509.ParsePKCS1PrivateKey(block.Bytes)
    //解密
    bits,_:=rsa.DecryptPKCS1v15(rand.Reader, pri, cipherTxt)
    //返回明文
    return bits
}

func main() {
    //加密
    cipher:=RSAEncrypt([]byte("hello wjt"))
    fmt.Println(cipher)

    //解密
    plain:=RSADecrypt(cipher)
    fmt.Println(plain)
}
```

4.4 中间人攻击

中间人攻击虽然不能破译RSA，但却是一种针对机密性的有效攻击。所谓中间人攻击，就是主动攻击者混入发送者和接收者中间，对发送者伪装成接收者，对接收者伪装成发送者的攻击方式。

> 解决方式：可以通过数字签名的方式解决

第五讲 RSA非对称加密在网络中的应用

5.1 什么是数字签名

`数字签名`就是只有信息的发送者才能产生的别人`无法伪造`的一段数字串，这段数字串同时也是对信息的发送者发送信息真实性的一个有效证明。

5.2 数字签名生成与验证

- 数字签名生成

  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`。

- 数字签名验证

  验证签名就是检查该消息的签名是否真的属于发送者，验证的结果可以是成功或者失败，成为就意味着这个签名是属于发送者的，失败则意味着这个签名不是属于发送者的。

5.3 非对称密码机制

公钥密码包括一个由公钥和私钥组成的密钥对。

- 用公钥加密，私钥解密
- 用私钥签名，公钥验证

5.4 用go语言实现数字签名过程

- 单机

```
package main

//用公钥加密，私钥解密
//用私钥签名，公钥验证
//私钥非公开

//编程实现公钥加密，私钥解密过程
func crypt() {
    //创建私钥
    priv,_:=rsa.GenerateKey(rand.Reader, bits:1024)
    fmt.Println(a:"输出系统自动产生的私钥", priv)
    //创建公钥
    pub:=priv.PublicKey
    //准备加密的明文
    org:=[]byte("hello wjt")
    //公钥加密
    cipherTxt,_:=rsa.EncryptOAEP(md5.New(), rand.Reader, &pub, org,
label:nil)
    //打印密文
    fmt.Println(a:"密文为", cipherTxt)
    fmt.Println(a:"密文为", base64.StdEncoding.EncodeToString(cipherTxt))
```

```
    //用私钥解密
    plaintext,_:=rsa.DecryptOAEP(md5.New(), rand.Reader, priv, cipherTxt,
label:nil)
    //打印解密后的结果
    fmt.Println(a:"明文为", string(plaintext))
}

//编程实现私钥签名，公钥验证过程
func sign() {
    //创建私钥
    priv,_:=rsa.GenerateKey(rand.Reader, bits:1024)
    //创建公钥
    pub:=&priv.PublicKey

    plaintxt:=[]byte("wangjitao")

    //实现Hash散列
    h:=md5.New()
    h.Write(plaintxt)
    hashed:=h.Sum(b:nil)

    //通过RSA实现数字签名，数字签名的作用为验证是否被篡改
    opts:=rsa.PSSOptions{rsa.PSSSaltLengthAuto, crypt.MD5}
    sig,_=rsa.SignPSS(rand.Reader,priv,crypto.MD5,hashed,&opts)

    fmt.Println(a:"签名的结果", sig)

    //通过公钥实现验证签名
    err:=rsa.VerifyPSS(pub,crypto.MD5,hashed,sig,&opts)
    if err == nil {
        fmt.Println(a:"验证成功")
    }
}

func main() {

}
```

- 模拟客户端与服务端

Client端

```
package main

//发送消息，需要对数据进行签名
var privateKey = []byte('所生成的私钥')

func SignData() []byte {
```

```
    //准备签名的数据
    plaintxt:=[]byte("wangjitao")
    h:=md5.New()
    h.Write(plaintxt)
    hashed:=h.Sum(b:nil)

    //将字节数组转换成私钥类型
    block,_:=pem.Decode(privateKey)
    priv,_:=x509.ParsePKCS1PrivateKey(block.Bytes)

    //通过RSA实现数字签名
    opts:=rsa.PSSOptions{rsa.PSSSaltLengthAuto, crypt.MD5}
    sig,_=rsa.SignPSS(rand.Reader,priv,crypto.MD5,hashed,&opts)

    //返回签名结果
    return sig
}

//通过TCP将数据和签名结果发送给接收端
func Send(data []byte) {
    conn,_:=net.ResolveTCPAddr("tcp4", "127.0.0.1:1234")
    n,_:=net.DialTCP("tcp", nil, conn)
    //将数据通过tcp协议发送给接收方
    n.Write(data)
    fmt.Println("发送结束")
}

func main() {
    //获得签名结果
    sg:=SignData()
    //dt是准备发送出去的数据
    var dt = "hello world"
    var data = make([]byte, len(dt)+len(sg))
    copy(data[0:11], []byte(dt))
    copy(data[11:], sg)
    //data数组其实有两部分组成，发出去的数据+签名结果
    Send(data)
}
```

Server端

```
package main

var publicKey = []byte('所生成的公钥')

func Recive() []byte {
    netListen,_:=net.Listen("tcp", "127.0.0.1:1234")
    defer netListen.Close()
```

```
    //监听端口，并接收数据
    for {
        conn,_:=netListen.Accept()
        //设置接收数据的内存缓存
        data:=make([]byte, 2048)
        for {
            n,_:=conn.Read(data)
            //返回接收的数据
            return data[:n]
        }
    }
}

func main() {
    //fmt.Println(Recive())
    //获得接收到的数据
    data:=Recive()
    //拆分数据
    plaintxt:=data[:11]
    fmt.Println(a:"接收的明文为",string(plaintxt))

    //获得接收到的数字签名的结果
    sig:=data[11:]

    //通过公钥做验证
    block,_:=pem.Decode(publicKey)
    pubInterface,_:=x509.ParsePKIXPublicKey(block.Bytes)
    pub:=pubInterface.(*rsa.PublicKey)

    h:=md5.New()
    h.Write([]byte("wangjitao"))
    hashed:=h.Sum(nil)

    e:=rsa.VerifyPSS(pub,cryptp.MD5,hashed,sig,nil)
    if e == nil {
        fmt.Println("接收数据成功，数据缺失是wangjitao发送的")
    }
}
```

5.5 数字签名算法

- RSA
- DSA
- ECDSA