Go语言 标准库 crypto包

crypto包搜集了常用的密码（算法）常量。

目前常用的加解密的方式无非三种：

1. 对称加密，加解密都使用的是同一个密钥, 其中的代表就是 AES、DES；
2. 非对加解密，加解密使用不同的密钥, 其中的代表就是 RSA、椭圆曲线；
3. 签名算法， 如 MD5, SHA1, HMAC等, 主要用于验证，防止信息被修改, 如：文件校验、数字签名；

md5

md5在crypto/md5包中，md5包提供了New和Sum方法。

```
func New() hash.Hash

func Sum(data []byte) [Size]byte
```

hash.Hash继承了io.Writer，因此可以将其当成一个输入流进行内容的更新。

```
type Writer interface {
   Write(p []byte) (n int, err error)
}
```

Write方法将p中的内容读入后存入到hash.Hash，最后在Sum方法通过内部函数checkSum计算出其校验和。

Sum函数是对hash.Hash对象内部存储的内容进行校验和计算然后将其追加到data的后面形成一个新的byte切片。通常将data置为nil。

Sum方法返回一个Size大小的byte数组，对于MD5返回一个128bit的16字节byte数组。

示例：

```
package main

import (
    "crypto/md5"
    "fmt"
    "io"
)

func MD5FromWriter() {
    h := md5.New()
    io.WriteString(h, "包子是个大帅哥！")
    io.WriteString(h, "咋就这么帅呢!")
    fmt.Printf("%x\n", h.Sum(nil))
}

func MD5FromSum() {
    data := []byte("包子是个大帅哥！")
    fmt.Printf("%x\n", md5.Sum(data))
```

```
}

func main() {
    MD5FromWriter()
    MD5FromSum()
}
#结果
9390b646a7efafa9e4bffa61faca6495
474002fba188efc3dccc60dce69927a8
```

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，输出一个256-bit的报文摘要。

SHA-256支持根据输入生成SHA256和SHA224的报文摘要，主要方法如下：

```
func New() hash.Hash

func New224() hash.Hash

func Sum256(data []byte) [Size]byte

func Sum224(data []byte) (sum224 [Size224]byte)
```

示例：

```
package main

import (
    "crypto/sha256"
    "fmt"
    "io"
)

func SHA256FromSum256() {
    sum := sha256.Sum256([]byte("hello world\n"))
    fmt.Printf("%x\n", sum)
}

func SHA256FromWriter() {
    h := sha256.New()
    h.Write([]byte("hello world\n"))
    fmt.Printf("%x\n", h.Sum(nil))
}

func main() {
    SHA256FromSum256()
    SHA256FromWriter()
}
#结果
a948904f2f0f479b8f8197694b30184b0d2ed1c1cd2a1ec0fb85d299a192a447
446591c52828f40adb3eba2168d1cc53bf69e9f806be49e6fd4ae20cca7c945b
```

AES

AES（Advanced Encryption Standard），即高级加密标准，是DES的替代标准。AES加密算法经历了公开选拔，最终在2000年由比利时密码学家Joan Daemen和Vincent Rijmen设计的Rijndael算法被选中，成为AES标准。

AES算法基于排列和置换运算。排列是对数据重新进行安排，置换是将一个数据单元替换为另一个。AES使用几种不同的方法来执行排列和置换运算。 AES是一个迭代的、对称密钥分组的密码，可以使用128、192和256位密钥，并且用128位（16字节）分组加密和解密数据。

AES是目前比较流行的对称加密算法，是一种分组密码（block cipher）算法，AES的分组长度为128比特（16字节），而密钥长度可以是128比特、192比特或256比特。

NewCipher

NewCipher创建并返回一个新的cipher.Block。参数是AES密钥，可以是AES-128，AES-192或AES-256。

```
func NewCipher(key []byte) (cipher.Block, error)
```

示例：

```
package main

import (
    "crypto/aes"
    "encoding/hex"
    "fmt"
    "log"
)

func EncryptAES(key string, plainText string) (string, error) {
    cipher, err := aes.NewCipher([]byte(key))
    if err != nil {
        return "", err
    }
    out := make([]byte, len(plainText))
    cipher.Encrypt(out, []byte(plainText))
    return hex.EncodeToString(out), nil
}

func DecryptAES(key string, encryptText string) (string, error) {
    decodeText, _ := hex.DecodeString(encryptText)
    cipher, err := aes.NewCipher([]byte(key))
    if err != nil {
        return "", err
    }
    out := make([]byte, len(decodeText))
    cipher.Decrypt(out, decodeText)
    return string(out[:]), nil
}

func main() {
    // 加密
```

```
    key := "thisisakeymustmorethan16"
    // plaintext
    text := "This is a secret"

    encrypt, err := EncryptAES(key, text)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(encrypt)

    // 解密
    plaintext, err := DecryptAES(key, encrypt)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(plaintext)
}
#结果
db647f5df56904ef3463834abc019c1d
This is a secret
```

RSA

1977年，Ron Rivest、Adi Shamir、Leonard Adleman三人在美国公布了一种公钥加密算法，即RSA公钥加密算法。RSA是目前最有影响力和最常用的公钥加密算法，可用于数据加密和数字签名。

OpenSSL生成私钥：

```
openssl genrsa -out private.pem 1024
```

OpenSSL生成公钥：

```
openssl rsa -in private.pem -pubout -out public.pem
```

RSA私钥中包含公钥，私钥的数据结构如下：

```
type PrivateKey struct {
   PublicKey            // public part.
   D         *big.Int   // private exponent
   Primes    []*big.Int // prime factors of N, has >= 2 elements.

   // Precomputed contains precomputed values that speed up private
   // operations, if available.
   Precomputed PrecomputedValues
}
```

rsa常用方法

EncryptOAEP

使用RSA-OAEP算法对信息进行加密

```
func EncryptOAEP(hash hash.Hash, random io.Reader, pub *PublicKey, msg []byte,
label []byte) ([]byte, error)
```

DecryptOAEP

使用RSA-OAEP算法对信息进行解密

```
func DecryptOAEP(hash hash.Hash, random io.Reader, priv *PrivateKey, ciphertext
[]byte, label []byte) ([]byte, error)
```

GenerateKey

生成私钥

```
func GenerateKey(random io.Reader, bits int) (*PrivateKey, error)
```

Public

根据私钥获取公钥

```
func (priv *PrivateKey) Public() crypto.PublicKey
```

Sign

生成私钥的签名

```
func (priv *PrivateKey) Sign(rand io.Reader, digest []byte, opts
crypto.SignerOpts) ([]byte, error)
```

Decrypt

利用私钥对加密信息进行解密

```
func (priv *PrivateKey) Decrypt(rand io.Reader, ciphertext []byte, opts
crypto.DecrypterOpts) (plaintext []byte, err error)
```

rsa加解密示例

```
package main

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

```
//生成RSA私钥和公钥，保存到文件中
func GenerateRSAKeyPair(bits int, private string, public string) {
   //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
   //Reader是一个全局、共享的密码用强随机数生成器
   privateKey, err := rsa.GenerateKey(rand.Reader, bits)
   if err != nil {
      panic(err)
   }
   //保存私钥
   //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
   X509PrivateKey := x509.MarshalPKCS1PrivateKey(privateKey)
   //使用pem格式对x509输出的内容进行编码
   //创建文件保存私钥
   privateFile, err := os.Create(private)
   if err != nil {
      panic(err)
   }
   defer privateFile.Close()
   //构建一个pem.Block结构体对象
   privateBlock := pem.Block{Type: "RSA Private Key", Bytes: X509PrivateKey}
   //将数据保存到文件
   pem.Encode(privateFile, &privateBlock)

   //保存公钥
   //获取公钥的数据
   publicKey := privateKey.PublicKey
   //X509对公钥编码
   X509PublicKey, err := x509.MarshalPKIXPublicKey(&publicKey)
   if err != nil {
      panic(err)
   }
   //pem格式编码
   //创建用于保存公钥的文件
   publicFile, err := os.Create(public)
   if err != nil {
      panic(err)
   }
   defer publicFile.Close()
   //创建一个pem.Block结构体对象
   publicBlock := pem.Block{Type: "RSA Public Key", Bytes: X509PublicKey}
   //保存到文件
   pem.Encode(publicFile, &publicBlock)
}

//RSA加密
func RSAEncrypt(plainText []byte, path string) []byte {
   //打开文件
   file, err := os.Open(path)
   if err != nil {
      panic(err)
   }
   defer file.Close()
   //读取文件的内容
   info, _ := file.Stat()
```

```
    buf := make([]byte, info.Size())
    file.Read(buf)
    //pem解码
    block, _ := pem.Decode(buf)
    //x509解码

    publicKeyInterface, err := x509.ParsePKIXPublicKey(block.Bytes)
    if err != nil {
        panic(err)
    }
    //类型断言
    publicKey := publicKeyInterface.(*rsa.PublicKey)
    //对明文进行加密
    cipherText, err := rsa.EncryptPKCS1v15(rand.Reader, publicKey, plainText)
    if err != nil {
        panic(err)
    }
    //返回密文
    return cipherText
}

//RSA解密
func RSADecrypt(cipherText []byte, path string) []byte {
    //打开文件
    file, err := os.Open(path)
    if err != nil {
        panic(err)
    }
    defer file.Close()
    //获取文件内容
    info, _ := file.Stat()
    buf := make([]byte, info.Size())
    file.Read(buf)
    //pem解码
    block, _ := pem.Decode(buf)
    //X509解码
    privateKey, err := x509.ParsePKCS1PrivateKey(block.Bytes)
    if err != nil {
        panic(err)
    }
    //对密文进行解密
    plainText, _ := rsa.DecryptPKCS1v15(rand.Reader, privateKey, cipherText)
    //返回明文
    return plainText
}
func main() {
    //生成密钥对，保存到文件
    GenerateRSAKeyPair(2048, "private.pem", "public.pem")
    message := []byte("hello world")
    //加密
    cipherText := RSAEncrypt(message, "public.pem")
    fmt.Println("Encrypt:", cipherText)
    //解密
    plainText := RSADecrypt(cipherText, "private.pem")
    fmt.Println("DeEncrypt:", string(plainText))
```

```
}
```

rsa数字签名示例

RSA在用于数字签名时，发送方通常先对消息生成散列值，再利用私钥对散列值进行签名，接收方收到消息及签名时，也先对消息生成散列值（与发送方使用同种单向散列函数），利用发送方发的公钥、签名以及自己生成的散列值进行签名验证。

```
package main

import (
   "crypto"
   "crypto/rand"
   "crypto/rsa"
   "crypto/sha256"
   "crypto/x509"
   "encoding/pem"
   "fmt"
   "os"
)

//生成RSA私钥和公钥，保存到文件中
func GenerateRSAKey(bits int, private string, public string) {
   //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
   //Reader是一个全局、共享的密码用强随机数生成器
   privateKey, err := rsa.GenerateKey(rand.Reader, bits)
   if err != nil {
      panic(err)
   }
   //保存私钥
   //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
   X509PrivateKey := x509.MarshalPKCS1PrivateKey(privateKey)
   //使用pem格式对x509输出的内容进行编码
   //创建文件保存私钥
   privateFile, err := os.Create(private)
   if err != nil {
      panic(err)
   }
   defer privateFile.Close()
   //构建一个pem.Block结构体对象
   privateBlock := pem.Block{Type: "RSA Private Key", Bytes: X509PrivateKey}
   //将数据保存到文件
   pem.Encode(privateFile, &privateBlock)

   //保存公钥
   //获取公钥的数据
   publicKey := privateKey.PublicKey
   //X509对公钥编码
   X509PublicKey, err := x509.MarshalPKIXPublicKey(&publicKey)
   if err != nil {
      panic(err)
   }
   //pem格式编码
   //创建用于保存公钥的文件
```

```
    publicFile, err := os.Create(public)
    if err != nil {
        panic(err)
    }
    defer publicFile.Close()
    //创建一个pem.Block结构体对象
    publicBlock := pem.Block{Type: "RSA Public Key", Bytes: X509PublicKey}
    //保存到文件
    pem.Encode(publicFile, &publicBlock)
}

//读取RSA私钥
func GetRSAPrivateKey(path string) *rsa.PrivateKey {
    //读取文件内容
    file, err := os.Open(path)
    if err != nil {
        panic(err)
    }
    defer file.Close()
    info, _ := file.Stat()
    buf := make([]byte, info.Size())
    file.Read(buf)
    //pem解码
    block, _ := pem.Decode(buf)
    //X509解码
    privateKey, err := x509.ParsePKCS1PrivateKey(block.Bytes)
    return privateKey
}

//读取RSA公钥
func GetRSAPublicKey(path string) *rsa.PublicKey {
    //读取公钥内容
    file, err := os.Open(path)
    if err != nil {
        panic(err)
    }
    defer file.Close()
    info, _ := file.Stat()
    buf := make([]byte, info.Size())
    file.Read(buf)
    //pem解码
    block, _ := pem.Decode(buf)
    //x509解码
    publicKeyInterface, err := x509.ParsePKIXPublicKey(block.Bytes)
    if err != nil {
        panic(err)
    }
    publicKey := publicKeyInterface.(*rsa.PublicKey)
    return publicKey
}

//对消息的散列值进行数字签名
func GetSign(msg []byte, path string) []byte {
    //取得私钥
    privateKey := GetRSAPrivateKey(path)
```

```
    //计算散列值
    hash := sha256.New()
    hash.Write(msg)
    bytes := hash.Sum(nil)
    //SignPKCS1v15使用RSA PKCS#1 v1.5规定的RSASSA-PKCS1-V1_5-SIGN签名方案计算签名
    sign, err := rsa.SignPKCS1v15(rand.Reader, privateKey, crypto.SHA256, bytes)
    if err != nil {
        panic(sign)
    }
    return sign
}

//验证数字签名
func VerifySign(msg []byte, sign []byte, path string) bool {
    //取得公钥
    publicKey := GetRSAPublicKey(path)
    //计算消息散列值
    hash := sha256.New()
    hash.Write(msg)
    bytes := hash.Sum(nil)
    //验证数字签名
    err := rsa.VerifyPKCS1v15(publicKey, crypto.SHA256, bytes, sign)
    return err == nil
}

//测试RSA数字签名
func main() {
    //生成密钥文件
    GenerateRSAKey(2048, "private.pem", "public.pem")

    //模拟发送方
    //要发送的消息
    msg := []byte("hello world")
    //生成签名
    sign := GetSign(msg, "private.pem")

    //模拟接收方
    //接受到的消息
    acceptmsg := []byte("hello world")
    //接受到的签名
    acceptsign := sign
    //验证签名
    ok := VerifySign(acceptmsg, acceptsign, "public.pem")
    if ok {
        fmt.Println("Signature is accepted")
    } else {
        fmt.Println("Signature is not accepted")
    }
}
```