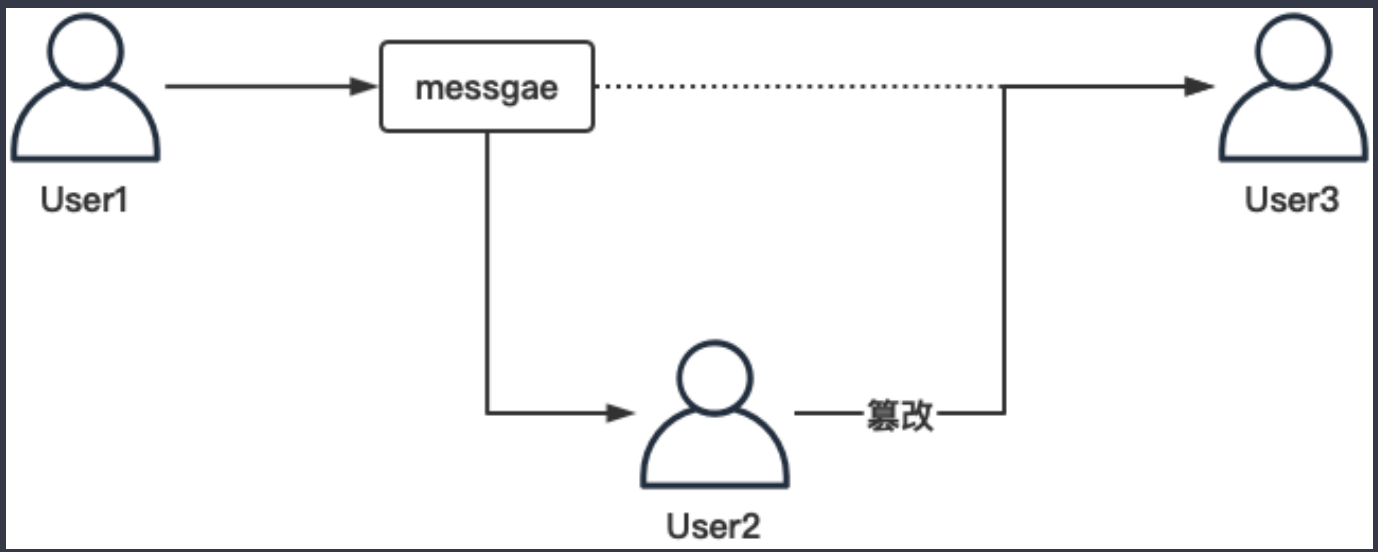
# 消息认证码及数字签名

## 1. 消息认证码

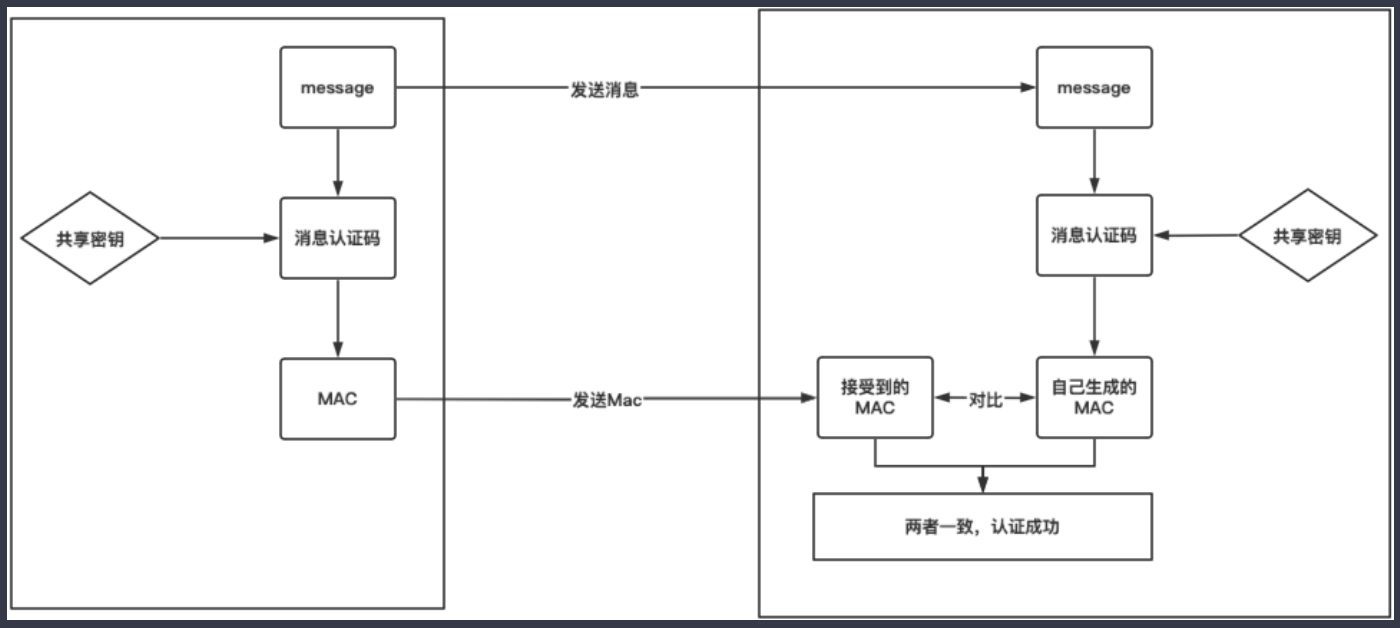
### 1.1 消息认证

消息认证码（message authentication）是一种确认完整性并进行认证的技术，取三个字母的首写简称为 MAC



> 思考针对上面这样一个场景如何去进行改进?
>
> 从哈希函数入手，将需要发送的数据进行哈希运算，将哈希值和原始值一并发送，需要在进行哈希运算的时候引入加密的步骤。在user1对数据进行哈希运算的时候引入一个密钥，让其参与哈希运算，生成的散列值一并发送。user2通过密钥和哈希算法对原始数据生成散列值，将其与user1发送过来的散列值进行对比，查看消息是否被篡改。

## 1.2 消息认证码的使用步骤




> 前提条件：在消息认证码生成的一方和校验的一方，必须有一个密钥。

## 1.3 go中对消息认证码的使用

go中使用消息认证码的包是 `crypto/hmac` ，

```go
package main

import (
  "crypto/hmac"
  "crypto/sha256"
  "fmt"
  "hash"
)
```

```go
// GenerateMessageAuthCode plainText 发送的消息 key 密钥 authMethod 消息码生成的方法
func GenerateMessageAuthCode(plainText, key []byte, authMethod func() hash.Hash) []byte {
  // 生成一个hash对象
  h := hmac.New(authMethod, key)
  // 写入消息内容:可以分批写入
  h.Write(plainText)
  // 获取消息码
  code := make([]byte, authMethod().Size())
  code = h.Sum(nil)

  return code
}

// CheckMessageAuthCode mac 接受到的消息验证码 plainText 接受到的消息 key 密钥
func CheckMessageAuthCode(mac, plainText, key []byte, authMethod func() hash.Hash) {
  // 生成一个hash对象
  h := hmac.New(authMethod, key)
  // 写入消息内容:可以分批写入
  h.Write(plainText)
  // 获取消息码
  code := make([]byte, authMethod().Size())
  code = h.Sum(nil)

  if hmac.Equal(mac, code) {
    fmt.Println("消息未被篡改.")
  } else {
    fmt.Println("消息已经被篡改.")
  }
}
```

```
func main() {
  key := []byte("my key")
  authMethod := func() hash.Hash { return sha256.New() }

  // --------------------模式生成消息码的过程-----------------
  message1 := []byte("今天的天气可太好了，珍惜这样的好天气!")
  code := GenerateMessageAuthCode(message1, key, authMethod)

  // --------------------模拟校验消息码的过程-----------------
  message2 := []byte("今天的天气可太好了，珍惜这样的好天气! ")
  CheckMessageAuthCode(code, message2, key, authMethod)
}
```

### 1.4 消息认证码的问题

- 弊端
  - 有密钥分发的困难
- 无法解决的问题
  - 不能进行第三方认证
  - 不能防止否认

## 2.数字签名

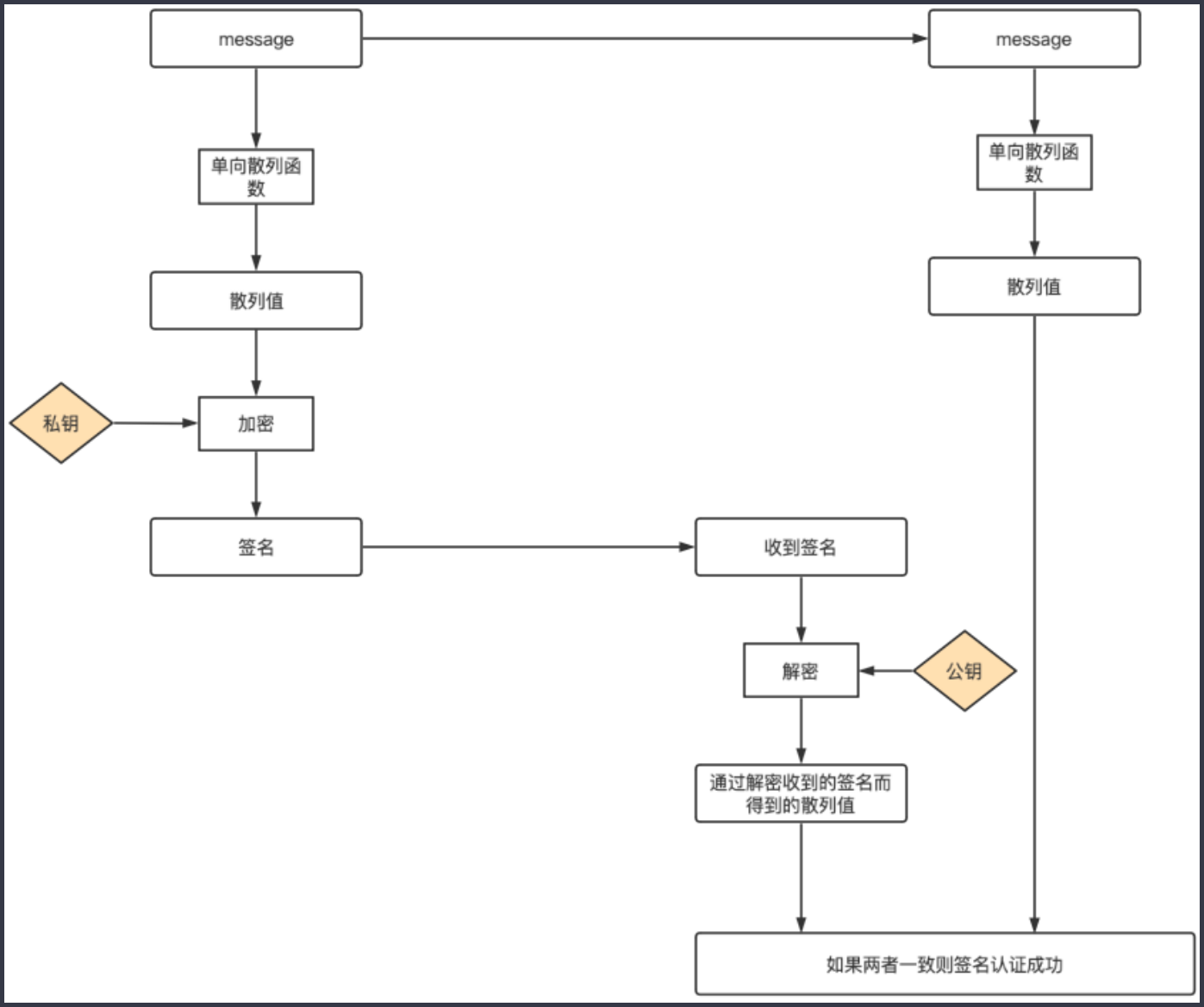
### 2.1 数字签名的生成和验证

> 在数字签名中，出现有以下两个行为
>
> - 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，也被称为对消息进行签名。生成签名就是对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”。

> **验证消息签名**这一行为一般是有消息的接受者来完成的，但也可以由需要验证消息的第三方来完成。验证签名就是检查该消息是否真的属于Alice，验证成功即表明该消息是属于Alice的，反之则不是

> 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”来完成消息的签名，而验证者或者第三方则使用验证密钥来验证消息。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**。
>
> 事实上，这种方式与非对称加密非常相似，只不过是反向使用，即使用私钥加密，只要能用公钥完成解密即可信任的。

## 2.2 数字签名的流程




## 2.3 Go使用RSA进行数字签名

- 在进行这一步的前提是你已经生成了这样一对的RSA的密钥对（以文件的形式）。如果没有可以参照如下的方式进行生成。

```
// GenerateRsaKey bits密钥长度
func GenerateRsaKey(bits int) {
  // ========================生成私钥============================
  // 使用rsa中的GenerateKey方法生成密钥
  eKey, err := rsa.GenerateKey(rand.Reader, bits)
  if err != nil {
    panic(err)
  }
```

```
// 通过x509标准将得到的rsa私钥序列序列化为ASN.1的编码字符串
data := x509.MarshalPKCS1PrivateKey(eKey)
// 将私钥设置到pem格式块中
block := &pem.Block{
  Type:  "RSA PRIVATE KEY",
  Bytes: data,
}
// 通过pem将设置好的数据进行编码，并写入磁盘
eFile, err := os.Create("private.pem")
if err != nil {
  panic(err)
}
if err := pem.Encode(eFile, block); err != nil {
  panic(err)
}

// ========================生成私钥==========================
// 从私钥中取出公钥
dKey := eKey.PublicKey
// 通过x509标准将得到rsa公钥序列序列化为ASN.1的编码字符串
data, err = x509.MarshalPKIXPublicKey(&dKey)
if err != nil {
  panic(err)
}
// 将私钥设置到pem格式块中
block = &pem.Block{
  Type:  "RSA PUBLIC KEY",
  Bytes: data,
}
// 通过pem将设置好的数据进行编码，并写入磁盘
dFile, err := os.Create("public.pem")
if err != nil {
  panic(err)
```

```
    }
    if err := pem.Encode(dFile, block); err != nil {
        panic(err)
    }
}
```

- 从文件中获取密钥

```
func GetRsaPrivateKey(privateFile string) *rsa.PrivateKey {
    // 读取私钥文件
    file, err := os.Open(privateFile)
    if err != nil {
        panic(err)
    }
    fileInfo, err := os.Stat(privateFile)
    if err != nil {
        panic(err)
    }
    data := make([]byte, fileInfo.Size())
    if _, err := file.Read(data); err != nil {
        panic(err)
    }
    // 使用pem对数据进行解密
    block, _ := pem.Decode(data)
    if block == nil || block.Type != "RSA PRIVATE KEY" {
        panic("failed to decode PEM block containing private key")
    }
    // 使用x509对block中的数据进行解析
    eKey, err := x509.ParsePKCS1PrivateKey(block.Bytes)
    if err != nil {
        panic(err)
    }
    return eKey
```

```
}

func GetRsaPublicKey(publicFile string) *rsa.PublicKey {
  // 读取私钥文件
  file, err := os.Open(publicFile)
  if err != nil {
    panic(err)
  }
  fileInfo, err := os.Stat(publicFile)
  if err != nil {
    panic(err)
  }
  data := make([]byte, fileInfo.Size())
  if _, err := file.Read(data); err != nil {
    panic(err)
  }
  // 使用pem对数据进行解密
  block, _ := pem.Decode(data)
  if block == nil || block.Type != "RSA PUBLIC KEY" {
    panic("failed to decode PEM block containing private key")
  }
  // 使用x509对block中的数据进行解析
  pub, err := x509.ParsePKIXPublicKey(block.Bytes)
  if err != nil {
    panic(err)
  }
  dKey, ok := pub.(*rsa.PublicKey)
  if !ok {
    panic("断言失败")
  }
  return dKey
}
```

- 数字签名

```
func GenerateRsaSign(plainText []byte) []byte {
  // 使用sha256计算plainText的散列值
  hash := sha256.New()
  hash.Write(plainText)
  hashValue := make([]byte, len(plainText))
  hashValue = hash.Sum(nil)

  // 获取RSA私钥并对散列值进行数字签名
  eKey := GetRsaPrivateKey("private.pem")
  sign, err := rsa.SignPKCS1v15(rand.Reader, eKey, 5,
hashValue)
  if err != nil {
    panic(err)
  }
  return sign
}
```

- 数字签名的认证

```
func CheckRsaSign(plainText, sign []byte) {
  // 使用sha256计算plainText
  hash := sha256.New()
  hash.Write(plainText)
  hashValue := make([]byte, len(plainText))
  hashValue = hash.Sum(nil)

  // 获取公钥并对签名进行解密
  dKey := GetRsaPublicKey("public.pem")
  if err := rsa.VerifyPKCS1v15(dKey, 5, hashValue, sign); err
== nil {
    fmt.Println("签名合法")
  } else {
    fmt.Println("签名不合法:", err.Error())
```

```
    }
}
```

- 完整代码

```
package main

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

// GenerateRsaKey bits密钥长度
func GenerateRsaKey(bits int) {
    // ========================生成私钥============================
    // 使用rsa中的GenerateKey方法生成密钥
    eKey, err := rsa.GenerateKey(rand.Reader, bits)
    if err != nil {
        panic(err)
    }
    // 通过x509标准将得到的rsa私钥序列序列化为ASN.1的编码字符串
    data := x509.MarshalPKCS1PrivateKey(eKey)
    // 将私钥设置到pem格式块中
    block := &pem.Block{
        Type:  "RSA PRIVATE KEY",
        Bytes: data,
    }
    // 通过pem将设置好的数据进行编码，并写入磁盘
    eFile, err := os.Create("private.pem")
```

```
    if err != nil {
        panic(err)
    }
    if err := pem.Encode(eFile, block); err != nil {
        panic(err)
    }

    // ========================生成私钥============================
    // 从私钥中取出公钥
    dKey := eKey.PublicKey
    // 通过x509标准将得到rsa公钥序列序列化为ASN.1的编码字符串
    data, err = x509.MarshalPKIXPublicKey(&dKey)
    if err != nil {
        panic(err)
    }
    // 将私钥设置到pem格式块中
    block = &pem.Block{
        Type:  "RSA PUBLIC KEY",
        Bytes: data,
    }
    // 通过pem将设置好的数据进行编码，并写入磁盘
    dFile, err := os.Create("public.pem")
    if err != nil {
        panic(err)
    }
    if err := pem.Encode(dFile, block); err != nil {
        panic(err)
    }
}

func GetRsaPrivateKey(privateFile string) *rsa.PrivateKey {
    // 读取私钥文件
    file, err := os.Open(privateFile)
```

```
  if err != nil {
    panic(err)
  }
  fileInfo, err := os.Stat(privateFile)
  if err != nil {
    panic(err)
  }
  data := make([]byte, fileInfo.Size())
  if _, err := file.Read(data); err != nil {
    panic(err)
  }
  // 使用pem对数据进行解密
  block, _ := pem.Decode(data)
  if block == nil || block.Type != "RSA PRIVATE KEY" {
    panic("failed to decode PEM block containing private key")
  }
  // 使用x509对block中的数据进行解析
  eKey, err := x509.ParsePKCS1PrivateKey(block.Bytes)
  if err != nil {
    panic(err)
  }
  return eKey
}
func GetRsaPublicKey(publicFile string) *rsa.PublicKey {
  // 读取私钥文件
  file, err := os.Open(publicFile)
  if err != nil {
    panic(err)
  }
  fileInfo, err := os.Stat(publicFile)
  if err != nil {
    panic(err)
  }
```

```
  data := make([]byte, fileInfo.Size())
  if _, err := file.Read(data); err != nil {
    panic(err)
  }
  // 使用pem对数据进行解密
  block, _ := pem.Decode(data)
  if block == nil || block.Type != "RSA PUBLIC KEY" {
    panic("failed to decode PEM block containing private key")
  }
  // 使用x509对block中的数据进行解析
  pub, err := x509.ParsePKIXPublicKey(block.Bytes)
  if err != nil {
    panic(err)
  }
  dKey, ok := pub.(*rsa.PublicKey)
  if !ok {
    panic("断言失败")
  }
  return dKey
}

func GenerateRsaSign(plainText []byte) []byte {
  // 使用sha256计算plainText的散列值
  hash := sha256.New()
  hash.Write(plainText)
  hashValue := make([]byte, len(plainText))
  hashValue = hash.Sum(nil)

  // 获取RSA私钥并对散列值进行数字签名
  eKey := GetRsaPrivateKey("private.pem")
  sign, err := rsa.SignPKCS1v15(rand.Reader, eKey, 5,
hashValue)
  if err != nil {
```

```
    panic(err)
  }
  return sign
}

func CheckRsaSign(plainText, sign []byte) {
  // 使用sha256计算plainText
  hash := sha256.New()
  hash.Write(plainText)
  hashValue := make([]byte, len(plainText))
  hashValue = hash.Sum(nil)

  // 获取公钥并对签名进行解密
  dKey := GetRsaPublicKey("public.pem")
  if err := rsa.VerifyPKCS1v15(dKey, 5, hashValue, sign); err
== nil {
    fmt.Println("签名合法")
  } else {
    fmt.Println("签名不合法:", err.Error())
  }
}

func main() {
  // 先生成密钥对
  GenerateRsaKey(1024)

  // 生成数字签名
  sendMessage := []byte("今天的天气真好，要珍惜如此美好的一天！")
  sign := GenerateRsaSign(sendMessage)

  // 检查数字签名
  recMessage := []byte("今天的天气真好，要珍惜如此美好的一天!")
  CheckRsaSign(recMessage, sign)
```

```
}
```

## 2.4 Go使用椭圆曲线进行数字签名

> 美国FIPS186-2标准，推荐使用5个素数上的椭圆曲线，这5个素数分别是：
>
> $P_{192} = 2^{192} - 2^{64} - 1$
>
> $P_{224} = 2^{224} - 2^{96} + 1$
>
> $P_{256} = 2^{256} - 2^{224} + 2^{192} - 2^{96} - 1$
>
> $P_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
>
> $P_{521} = 2^{521} - 1$
>
> 同时这些数字越大，机密性越高，但是效率越低
>
> 在Golang中，椭圆曲线对应的包:" `crypto/elliptic` ";
>
> 在Golang中，使用椭圆曲线进行数字签名使用：" `crypto/ecdsa` "
>
> 相比于上面的RSA加密方式，如果你的message比较大，需要设置一个合适的bits，但是椭圆曲线不需要这样的问题。

- 生成基于椭圆曲线的密钥对

```
func GenerateCurveKey() {
  // ============================生成私钥
============================
  // 生成私钥
  eKey, err := ecdsa.GenerateKey(elliptic.P521(), rand.Reader)
  if err != nil {
    panic(err)
  }
  // 使用x509进行编码
  data, err := x509.MarshalECPrivateKey(eKey)
  if err != nil {
    panic(err)
```

```
    }
    // 构造block块
    block := &pem.Block{
        Type:  "CURVE PRIVATE KEY",
        Bytes: data,
    }
    // 使用pem进行编码并写入文件
    file, err := os.Create("private.pem")
    if err := pem.Encode(file, block); err != nil {
        panic(err)
    }
    defer file.Close()
    // ============================生成公钥
============================
    dKey := eKey.PublicKey
    // 使用x509进行编码
    data, err = x509.MarshalPKIXPublicKey(&dKey)
    if err != nil {
        panic(err)
    }
    // 构造block
    block = &pem.Block{
        Type:  "CURVE PUBLIC KEY",
        Bytes: data,
    }
    // 使用pem进行编码并写入文件
    file, err = os.Create("public.pem")
    if err := pem.Encode(file, block); err != nil {
        panic(err)
    }
}
```

- 编写获取密钥对的方法

```
```

```go
func GetCurvePrivateKey(filePath string) *ecdsa.PrivateKey {
  // 读取私钥文件
  file, err := os.Open(filePath)
  if err != nil {
    panic(err)
  }
  fileInfo, err := file.Stat()
  if err != nil {
    panic(err)
  }
  data := make([]byte, fileInfo.Size())
  if _, err = file.Read(data); err != nil {
    panic(err)
  }

  // 使用pem进行解析
  block, _ := pem.Decode(data)
  if block == nil || block.Type != "CURVE PRIVATE KEY" {
    panic("pem解析私钥错误.")
  }

  // 使用x509进行解析
  eKey, err := x509.ParseECPrivateKey(block.Bytes)
  if err != nil {
    panic(err)
  }
  return eKey
}

func GetCurvePublicKey(filePath string) *ecdsa.PublicKey {
  // 读取私钥文件
  file, err := os.Open(filePath)
  if err != nil {
```

```
    panic(err)
  }
  fileInfo, err := file.Stat()
  if err != nil {
    panic(err)
  }
  data := make([]byte, fileInfo.Size())
  if _, err = file.Read(data); err != nil {
    panic(err)
  }

  // 使用pem进行解析
  block, _ := pem.Decode(data)
  if block == nil || block.Type != "CURVE PUBLIC KEY" {
    panic("pem解析公钥错误.")
  }

  // 使用x509进行解析
  pub, err := x509.ParsePKIXPublicKey(block.Bytes)
  if err != nil {
    panic(err)
  }
  dKey, ok := pub.(*ecdsa.PublicKey)
  if !ok {
    panic("公钥断言失败.")
  }
  return dKey
}
```

- 编写数字签名及验证数字签名的方法

```
func GenerateCurveSign(plainText []byte) (*big.Int, *big.Int) {
  // 获取消息的散列值
  h := sha256.New()
```

```
  h.Write(plainText)
  hashText := make([]byte, len(plainText))
  hashText = h.Sum(nil)

  // 获取椭圆曲线的私钥匙
  eKey := GetCurvePrivateKey("private.pem")
  r, s, err := ecdsa.Sign(rand.Reader, eKey, hashText)
  if err != nil {
    panic(err)
  }
  return r, s
}

func CheckCurveSign(plainText []byte, r, s *big.Int) {
  // 获取消息的散列值
  h := sha256.New()
  h.Write(plainText)
  hashText := make([]byte, len(plainText))
  hashText = h.Sum(nil)

  // 获取椭圆曲线的公钥
  dKey := GetCurvePublicKey("public.pem")
  if ecdsa.Verify(dKey, hashText, r, s) {
    fmt.Println("签名合法")
  } else {
    fmt.Println("签名不合法")
  }
}
```

- 完整的代码

```
package main

import (
```

```
  "crypto/ecdsa"
  "crypto/elliptic"
  "crypto/rand"
  "crypto/sha256"
  "crypto/x509"
  "encoding/pem"
  "fmt"
  "math/big"
  "os"
)

func GenerateCurveKey() {
  // ============================生成私钥
============================
  // 生成私钥
  eKey, err := ecdsa.GenerateKey(elliptic.P521(), rand.Reader)
  if err != nil {
    panic(err)
  }
  // 使用x509进行编码
  data, err := x509.MarshalECPrivateKey(eKey)
  if err != nil {
    panic(err)
  }
  // 构造block块
  block := &pem.Block{
    Type:  "CURVE PRIVATE KEY",
    Bytes: data,
  }
  // 使用pem进行编码并写入文件
  file, err := os.Create("private.pem")
  if err := pem.Encode(file, block); err != nil {
    panic(err)
```

```
  }
  defer file.Close()
  // ============================生成公钥
==========================
  dKey := eKey.PublicKey
  // 使用x509进行编码
  data, err = x509.MarshalPKIXPublicKey(&dKey)
  if err != nil {
    panic(err)
  }
  // 构造block
  block = &pem.Block{
    Type:  "CURVE PUBLIC KEY",
    Bytes: data,
  }
  // 使用pem进行编码并写入文件
  file, err = os.Create("public.pem")
  if err := pem.Encode(file, block); err != nil {
    panic(err)
  }
}

func GetCurvePrivateKey(filePath string) *ecdsa.PrivateKey {
  // 读取私钥文件
  file, err := os.Open(filePath)
  if err != nil {
    panic(err)
  }
  fileInfo, err := file.Stat()
  if err != nil {
    panic(err)
  }
  data := make([]byte, fileInfo.Size())
```

```
  if _, err = file.Read(data); err != nil {
    panic(err)
  }

  // 使用pem进行解析
  block, _ := pem.Decode(data)
  if block == nil || block.Type != "CURVE PRIVATE KEY" {
    panic("pem解析私钥错误.")
  }

  // 使用x509进行解析
  eKey, err := x509.ParseECPrivateKey(block.Bytes)
  if err != nil {
    panic(err)
  }
  return eKey
}

func GetCurvePublicKey(filePath string) *ecdsa.PublicKey {
  // 读取私钥文件
  file, err := os.Open(filePath)
  if err != nil {
    panic(err)
  }
  fileInfo, err := file.Stat()
  if err != nil {
    panic(err)
  }
  data := make([]byte, fileInfo.Size())
  if _, err = file.Read(data); err != nil {
    panic(err)
  }
```

```
  // 使用pem进行解析
  block, _ := pem.Decode(data)
  if block == nil || block.Type != "CURVE PUBLIC KEY" {
    panic("pem解析公钥错误.")
  }

  // 使用x509进行解析
  pub, err := x509.ParsePKIXPublicKey(block.Bytes)
  if err != nil {
    panic(err)
  }
  dKey, ok := pub.(*ecdsa.PublicKey)
  if !ok {
    panic("公钥断言失败.")
  }
  return dKey
}

func GenerateCurveSign(plainText []byte) (*big.Int, *big.Int) {
  // 获取消息的散列值
  h := sha256.New()
  h.Write(plainText)
  hashText := make([]byte, len(plainText))
  hashText = h.Sum(nil)

  // 获取椭圆曲线的私钥匙
  eKey := GetCurvePrivateKey("private.pem")
  r, s, err := ecdsa.Sign(rand.Reader, eKey, hashText)
  if err != nil {
    panic(err)
  }
  return r, s
}
```

```go
func CheckCurveSign(plainText []byte, r, s *big.Int) {
  // 获取消息的散列值
  h := sha256.New()
  h.Write(plainText)
  hashText := make([]byte, len(plainText))
  hashText = h.Sum(nil)

  // 获取椭圆曲线的公钥
  dKey := GetCurvePublicKey("public.pem")
  if ecdsa.Verify(dKey, hashText, r, s) {
    fmt.Println("签名合法")
  } else {
    fmt.Println("签名不合法")
  }
}

func main() {
  GenerateCurveKey()

  // ========================生成签名====================
  message1 := []byte("今天的天气真好，一定要珍惜这样的天气！")
  r, s := GenerateCurveSign(message1)

  // ========================检查签名====================
  message2 := []byte("今天的天气真好，一定要珍惜这样的天气!")
  CheckCurveSign(message2, r, s)
}
```

## 2.5 数字签名无法解决的问题