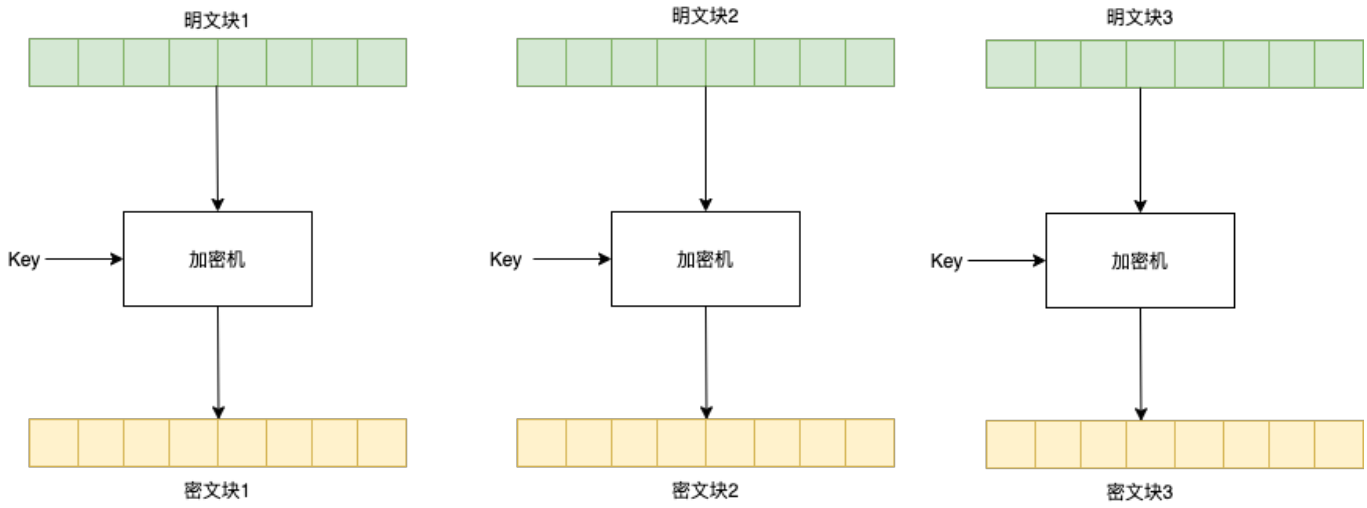
# 对称加密

- 采用单钥密码系统的加密方法，同一个密钥可以同时用作信息的加密和解密，这种加密方法称为对称加密，也称为单密钥加密。
- 示例
  - 我们现在有一个原文3要发送给B
  - 设置密钥为108, 3 * 108 = 324, 将324作为密文发送给B
  - B拿到密文324后, 使用324/108 = 3 得到原文
- 常见加密算法
  - DES : Data Encryption Standard，即数据加密标准，是一种使用密钥加密的块算法，1977年被美国联邦政府的国家标准局确定为联邦资料处理标准（FIPS），并授权在非密级政府通信中使用，随后该算法在国际上广泛流传开来。
  - AES : Advanced Encryption Standard, 高级加密标准 .在密码学中又称Rijndael加密法，是美国联邦政府采用的一种区块加密标准。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。
- 特点
  - 加密速度快, 可以加密大文件
  - 密文可逆, 一旦密钥文件泄漏, 就会导致数据暴露
  - 加密后编码表找不到对应字符, 出现乱码
  - 一般结合Base64使用

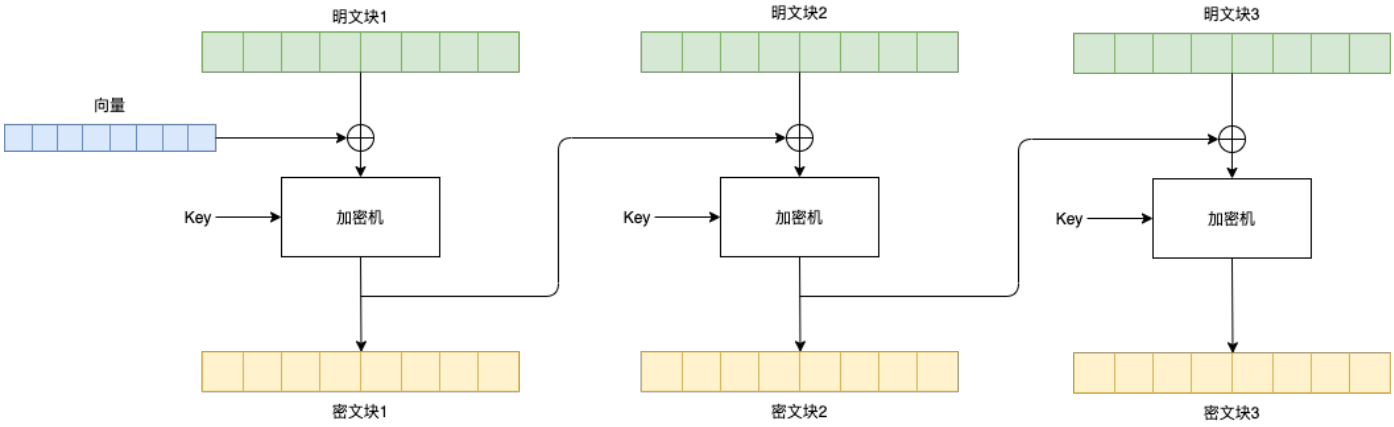
# 分组模式

## 1.ECB



### 2.CBC

需要**秘钥**和**向量**




## 填充模式

### 1.NoPadding

不填充，要求原文长度必须是16byte的整数倍

### 2.PKCS5Padding

对5字节的数据使用PKCS#5填充

```
data: 01 02 03 04 05
```

需要填充3字节,填充数据位03

```
padding: 03 03 03
```

填充后 data||padding

```
data_padded: 01 02 03 04 05 03 03 03
```

### 3.PKCS7Padding

https://datatracker.ietf.org/doc/html/rfc5652#section-6.3

> ```
> Some content-encryption algorithms assume the input length is a
> multiple of k octets, where k is greater than one.  For such
> algorithms, the input shall be padded at the trailing end with
> k-(lth mod k) octets all having value k-(lth mod k), where lth is
> ```

```
the length of the input.  In other words, the input is padded at
the trailing end with one of the following strings:

                01 -- if lth mod k = k-1
             02 02 -- if lth mod k = k-2
                 .
                 .
                 .
       k k ... k k -- if lth mod k = 0

The padding can be removed unambiguously since all input is padded,
including input values that are already a multiple of the block size,
and no padding string is a suffix of another.  This padding method is
well defined if and only if k is less than 256.
```

**缺几位补上几个几**

AES128算法其数据块大小为16字节

对11个字节的数据使用PKCS7填充

```
data: 00 11 22 33 44 55 66 77 88 99 AA
```

需要补充5个字节，每个字节的数据就是5

```
padding: 05 05 05 05 05
```

填充后 data || padding

```
data_padded: 00 11 22 33 44 55 66 77 88 99 AA 05 05 05 05 05
```

如果数据块是16个字节，或者n个16字节数据，则需要填充16字节（数据是16）

```
data: 00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF
```

需要补充16个字节，每个字节的数据就是16

```
padding: 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

填充后 data || padding

```
data_padded: 00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10
```

PKCS5就是块大小位8字节的PKCS7

密文 = aes(data, aes_key, aes_iv);

明文 = aes(cipherData, aes_key, aes_iv);

```
// 加密
public static String encrypt(String data, Key key, AlgorithmParameterSpec ivParam) {
  try {
    // 创建加密对象,
    // AES：加密算法
    // CBC：分组模式
    // PKCS5Padding：  填充模式
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    // 初始化,
    // param1：加密；param2：秘钥；param3：向量
    cipher.init(Cipher.ENCRYPT_MODE, key, ivParam);

    byte[] encrptyBytes = cipher.doFinal(data.getBytes(StandardCharsets.UTF_8));
    String encryptStr = Base64.getEncoder().encodeToString(encrptyBytes);
    return encryptStr;
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}

// 解密
public static String decrypt(String clipherString, Key key, AlgorithmParameterSpec
ivParam) {
  String decrypted = "";
  try {
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, key, ivParam);
    byte[] decode = Base64.getDecoder().decode(clipherString);
    byte[] bytes = cipher.doFinal(decode);
    decrypted = new String(bytes, "UTF-8");
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
  return decrypted;
}
```

# 单向散列

- 消息摘要（Message Digest）又称为数字摘要(Digital Digest)
- 它是一个唯一对应一个消息或文本的固定长度的值，它由一个单向Hash加密函数对消息进行作用而产生
- 使用数字摘要生成的值是不可以篡改的，为了保证文件或者值的安全

无论输入的消息有多长，计算出来的消息摘要的长度总是固定的。例如应用MD5算法摘要的消息有128个比特位，用SHA-1算法摘要的消息最终有160比特位的输出

只要输入的消息不同，对其进行摘要以后产生的摘要消息也必不相同；但相同的输入必会产生相同的输出

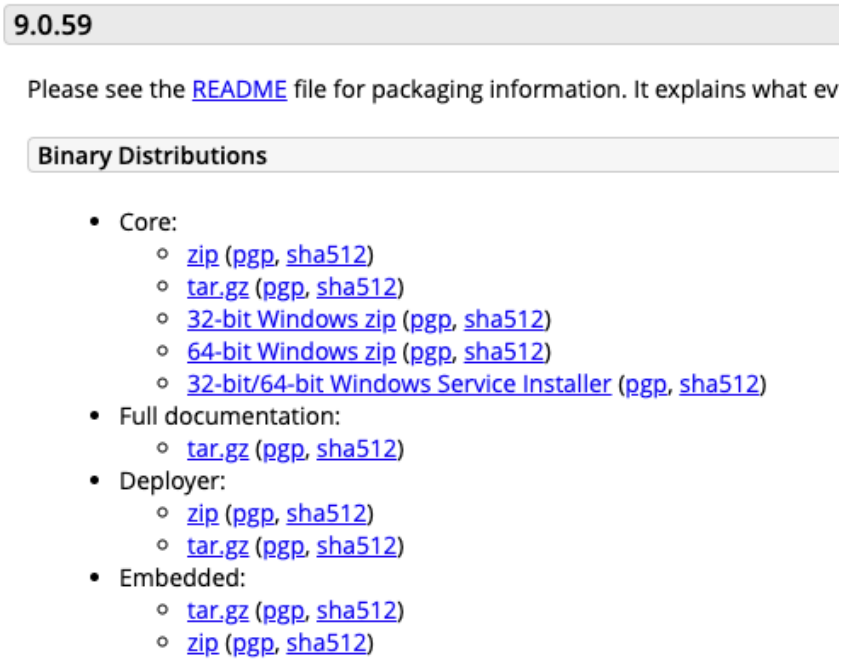
**消息摘要是单向、不可逆的**

常见算法:

```
- MD5
- SHA1
- SHA256
- SHA512
```

在线生产摘要值

百度搜索 `tomcat` ， 进入官网下载， 会经常发现有 `sha1`， `sha512` , 这些都是数字摘要





用途:

```
1. 验签
2. 数据完整性验证
3. 对登录密码做摘要
```

md5

```
MessageDigest messageDigest = MessageDigest.getInstance("MD5");
byte[] digest = messageDigest.digest(str.getBytes(StandardCharsets.UTF_8));
return HexUtils.toHexString(digest);
```

sha256

```
MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
byte[] bytes = messageDigest.digest(str.getBytes(StandardCharsets.UTF_8));
return HexUtils.toHexString(bytes);
```

# 非对称加密

RSA，由罗纳德·李维斯特（Ron **R**ivest）、阿迪·萨莫尔（Adi **S**hamir）和伦纳德·阿德曼（Leonard **A**dleman）三位科学家提出。

- 与对称加密算法不同，非对称加密算法需要两个密钥：`公开密钥(publickey)` 和 `私有密(privatekey)`
- 公开密钥和私有密钥是`一对`
- 如果用`公开密钥`对数据进行`加密`，只有用`对应的私有密钥`才能`解密`。
- 如果用`私有密钥`对数据进行`加密`，只有用`对应的公开密钥`才能`解密`。
- 因为加密和解密使用的是两个`不同`的密钥，所以这种算法叫作`非对称加密算法`。

- 示例
  - 首先生成密钥对, 公钥为(5,14), 私钥为(11,14)
  - 现在A希望将原文2发送给B
  - A使用公钥加密数据. 2的5次方mod 14 = 4 , 将密文4发送给B
  - B使用私钥解密数据. 4的11次方mod14 = 2, 得到原文2
- 特点
  - 加密和解密使用不同的密钥
  - 如果使用私钥加密, 只能使用公钥解密
  - 如果使用公钥加密, 只能使用私钥解密
  - 处理数据的速度较慢, 因为安全级别高
- 常见算法
  - RSA
  - ECC

欧拉函数：

小于n的正整数中与n互质的数的数目

互质是公约数只有1的两个整数叫作互质整数

质数是指在大于1的自然数中，除了1和它本身以外不再有其他因数的自然数

> 

> φ(6) = 2;
>
> 小于6的正整数：5，4，3，2，1；只有5 和 1与6互质
>
> 如果n是质数，φ(n) = n - 1
>
> φ(7) = 6;
>
> 小于7的正整数：6，5，4，3，2，1，都与7互质

如果n可以分解成2个互质的整数之积，那么n的欧拉函数等于这两个因子的欧拉函数之积。

即若n = p * q，且p,q互质，则φ(n) = φ(p * q) = φ(p) * φ(q) = (p -1) * (q - 1)

模反元素d

如果两个正整数e和φ(n)互质，那么一定可以找到一个整数d，使得ed-1被φ(n)整除，或者说ed除以φ(n)余1，此时d就叫做e的模反元素

> ed - 1= kφ(n)
>
> ed mod φ(n) = 1
>
> 3 和 11 互质， 3 * 4 - 1 = 11 * 1 ，4就是3的模反元素；d = (11 * k + 1) / 3
>
> 4 ±k*11都是3的模反元素；
>
> d +k * φ(n)都是e的模反元素

RSA加密过程

| 步骤 | 说明 | 描述 |
| --- | --- | --- |
| 1 | 选择一对不相等且足够大的质数 | p,q |
| 2 | 计算p,q的乘积 | n=p*q |
| 3 | 计算n的欧拉函数 | φ(n)=(p-1)*(q-1) |
| 4 | 选一个φ(n)与互质的整数e | 1<e<φ(n) |
| 5 | 计算出e对于φ(n)的模反元素d | de mode φ(n) = 1 |
| 6 | 公钥 | KU = (e, n) |
| 7 | 私钥 | KR = (d, n) |

明文M　加密

$$M^e mod n = C$$

密文C　解密

$$C^d mod n = M$$

p = 3, q = 11

n = 3 * 11 = 33

ϕ(33) = ϕ(3) * ϕ(11) = 2 * 10 = 20

e = 3

e*d - 1 = kϕ(33)； 3 * d - 1 = k * 20； d = (20k + 1) / 3 = (20 * 1 + 1) / 3 = 7; d = 7

KU = (3, 33)

KR = (7, 33)

明文 M = 6，加密：6^3 % 33 = 18（密文）

解密：18^7 % 33 = 6

用途:

1. 验证用户身份
2. 交换秘钥

1.生成公私钥对

```
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
KeyPair keyPair = keyPairGenerator.generateKeyPair();
PublicKey aPublic = keyPair.getPublic();
PrivateKey aPrivate = keyPair.getPrivate();

byte[] aPublicEncoded = aPublic.getEncoded();
String publicKeyStr = Base64Utils.encodeToString(aPublicEncoded);
System.out.println(publicKeyStr);
String publicKeyHexString = HexUtils.toHexString(aPublicEncoded);
System.out.println(publicKeyHexString);

byte[] aPrivateEncoded = aPrivate.getEncoded();
String privateKeyStr = Base64Utils.encodeToString(aPrivateEncoded);
System.out.println(privateKeyStr);
String privateKeyHexStr = HexUtils.toHexString(aPrivateEncoded);
System.out.println(privateKeyHexStr);
```

2.私钥加密，公钥解密

```
String input = "spdb浦发银行";
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE, aPrivate);
byte[] entryBytes = cipher.doFinal(input.getBytes(StandardCharsets.UTF_8));
String entryStr = Base64Utils.encodeToString(entryBytes);
System.out.println("私钥加密后的数据:  " + entryStr);


// 公钥解密
cipher.init(Cipher.DECRYPT_MODE, aPublic);
byte[] decryptBytes = cipher.doFinal(entryBytes);
String plain = new String(decryptBytes, "UTF-8");
System.out.println("解密后的明文:  " + plain);
```

**私钥解密，私钥解密会报错**

```
私钥加密后的数据:  wxnPoeDjpry8GMlzO5covcX39WLRWWLB28pcnvPL349oLLgkmOMkB8m0amtohOHGH9qGnRUcVZipkN
公钥解密后的明文:  spdb浦发银行
Exception in thread "main" javax.crypto.BadPaddingException Create breakpoint : Decryption error
    at java.base/sun.security.rsa.RSAPadding.unpadV15(RSAPadding.java:378)
    at java.base/sun.security.rsa.RSAPadding.unpad(RSAPadding.java:290)
    at java.base/com.sun.crypto.provider.RSACipher.doFinal(RSACipher.java:366)
    at java.base/com.sun.crypto.provider.RSACipher.engineDoFinal(RSACipher.java:392)
    at java.base/javax.crypto.Cipher.doFinal(Cipher.java:2205)
    at com.spdb.encryptdemo.utils.RSA.main(RSA.java:50)
```

**私钥加密，只能使用公钥解密**

3.公钥加密，私钥解密

```
String input = "spdb浦发银行";

Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE, aPublic);
byte[] entryBytes = cipher.doFinal(input.getBytes(StandardCharsets.UTF_8));
// 公钥加密的数字符串
System.out.println(Base64Utils.encodeToString(entryBytes));

// 使用私钥解密
cipher.init(Cipher.DECRYPT_MODE, aPrivate);
byte[] decryptBytes = cipher.doFinal(entryBytes);
String plain = new String(decryptBytes, "UTF-8");
System.out.println("私钥解密后的明文:  " + plain);
```

使用公钥加密只能使用私钥解密
公钥加密，公钥解密报的异常如下

```
私钥解密后的明文:  spdb浦发银行
Exception in thread "main" javax.crypto.BadPaddingException Create breakpoint : Decryption error
    at java.base/sun.security.rsa.RSAPadding.unpadV15(RSAPadding.java:378)
    at java.base/sun.security.rsa.RSAPadding.unpad(RSAPadding.java:290)
    at java.base/com.sun.crypto.provider.RSACipher.doFinal(RSACipher.java:359)
    at java.base/com.sun.crypto.provider.RSACipher.engineDoFinal(RSACipher.java:392)
    at java.base/javax.crypto.Cipher.doFinal(Cipher.java:2205)
    at com.spdb.encryptdemo.utils.RSA.main(RSA.java:51)
```

epcs中使用指数和模数初始化的私钥

```
// 从私钥中获取指数和模数
BigInteger publicExponent1 = ((RSAPrivateCrtKey) aPrivate).getPublicExponent();
BigInteger modulus1 = ((RSAPrivateCrtKey) aPrivate).getModulus();
```

```
// 公钥的指数
BigInteger publicExponent = ((RSAPublicKey) aPublic).getPublicExponent();
System.out.println("公钥的指数:  " + publicExponent);

// 公钥的模数
BigInteger modulus = ((RSAPublicKey) aPublic).getModulus();
System.out.println("公钥的模数:  " + modulus);
System.out.println("公钥的format:  " + ((RSAPublicKey) aPublic).getFormat());
// 生成公钥
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
RSAPublicKeySpec rsaPublicKeySpec = new RSAPublicKeySpec(modulus, publicExponent);
PublicKey generatePublic = keyFactory.generatePublic(rsaPublicKeySpec);

cipher.init(Cipher.ENCRYPT_MODE, generatePublic);
entryBytes = cipher.doFinal(input.getBytes(StandardCharsets.UTF_8));

// 私钥的指数和模数，其中公私钥的模数是相同的
BigInteger publicExponent1 = ((RSAPrivateCrtKey) aPrivate).getPrivateExponent();
BigInteger modulus1 = ((RSAPrivateCrtKey) aPrivate).getModulus();
System.out.println(publicExponent1);
System.out.println(modulus1);
System.out.println(((RSAPrivateCrtKey) aPrivate).getFormat());

// 更具模数和指数生成私钥
KeySpec keySpec = new RSAPrivateKeySpec(modulus1, publicExponent1);
PrivateKey generatePrivate = keyFactory.generatePrivate(keySpec);
// 解密
cipher.init(Cipher.DECRYPT_MODE, generatePrivate);
byte[] doFinal = cipher.doFinal(entryBytes);
```

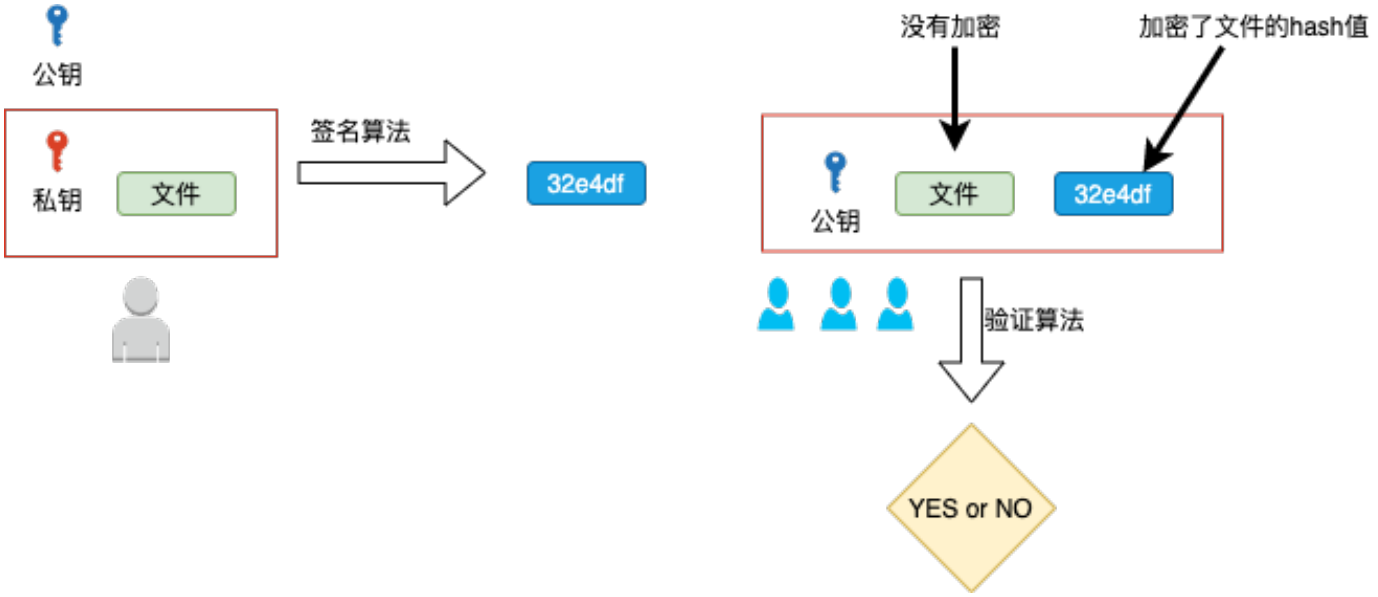
```
System.out.println(new String(doFinal, "UTF-8"));
```

# 数字签名

数字签名（又称公钥数字签名）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。它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来实现的，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。数字签名是非对称密钥加密技术与数字摘要技术的应用。

数字签名的主要作用就是保证了数据的有效性（验证是谁发的）和完整性（证明信息没有被篡改）。

签名和验签的过程：



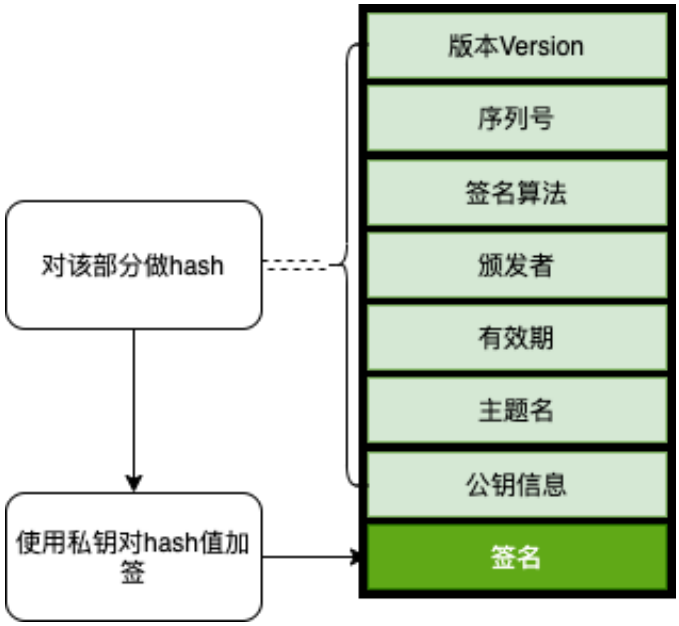

```
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
KeyPair keyPair = keyPairGenerator.generateKeyPair();
RSAPublicKey aPublic = (RSAPublicKey)keyPair.getPublic();
RSAPrivateKey aPrivate = (RSAPrivateKey)keyPair.getPrivate();

String input = "你好，我是tom";
// 获取签名对象
Signature signature = Signature.getInstance("sha256withrsa");
// 初始化签名对象
signature.initSign(aPrivate);
// 传入原文
signature.update(input.getBytes(StandardCharsets.UTF_8));
// 开始签名
byte[] sign = signature.sign();
System.out.println("签名数据：  " + Base64Utils.encodeToString(sign));
```

```
/// 验签操作
// 获取Signature对象
Signature signature1 = Signature.getInstance("sha256withrsa");
// 使用公钥初始化，
signature1.initVerify(aPublic);
// 传入原文数据
signature1.update(input.getBytes(StandardCharsets.UTF_8));
// 开始验签，传入签名数据
boolean verify = signature1.verify(sign);
System.out.println(verify ? "验签成功": "验签失败");
```

# 数字证书

数字证书是一个经`数字证书认证机构CA`（Certificate Authority）认证签名的文件，包含拥有者的公钥以及相关的身份信息。用户想要获得证书，应该先向`CA`提出申请，`CA`验证申请者的身份后，为其分配一个公钥与其身份信息绑定，为该信息信息进行签名，作为证书的一部分，然后把整个证书发送给申请者。 当需要鉴别证书真伪时，只需要用`CA`的公钥对证书上的签名进行验证，验证通过则证书有效。

## 证书的结构




字段含义

- 版本： 使用`X.509`的版本，目前普遍使用`v3`版本；
- 序列号：`CA`分配给证书的一个整数，作为证书的唯一标识；
- 签名算法：`CA`颁发证书使用的签名算法；
- 有效期：包含证书的起止日期；
- 主体名：该证书拥有者的名称，如果与颁发者相同则说明证书是一个自签名证书；
- 公钥信息：对外公开的公钥以及公钥算法（RSA ECC）；
- 扩展信息：通常包含证书的用法，`证书吊销列表（Certificate Revocation List，CRL）`的发布地址等可

选字段；

- 签名：颁发者用私钥对证书信息的签名；

通过查看浏览器查看网站证书


## 证书类型

- 自签名证书：自签名证书又称根证书，是自己发给自己的证书，证书的颁发者和主体同名；
- 本地证书：`CA`颁发给申请者的证书；
- 设备本地：设备根据CA证书给自己颁发的证书，证书中的颁发者名称是CA服务器的名称。

## 证书格式

- `PKCS#12`：`#12`是标准号，常见后缀是`.P12`，可包含私钥也可不包含私钥；
- `DER`：二进制格式保存证书，不包含私钥，常见后缀`.DER`；
- `PEM`：以ASCII格式保存的证书，可包含私钥，也可不包含私钥，常见后缀`.PEM`；