# Base64

Base64是网络上最常见的传输8bit字节代码的编码方式之一。Base64编码可用在http环境下传递较长的标识信息。Base64编码简短而且具有不可读性。

注意：标准的base64不适合直接放到url里传输。因为url编码器会把标准base64中的“/”和“+”字符变为形如“%XX”的形式。而这些“%”在存入数据库时还需要在进行转化，数据库中%会被作为通配符。

有一种用于url的改进的base64编码，它会在末尾填充“=”，并将标准的base64中的“+”和“/”改成了“-”和“\_”，这样免去了在url编解码和数据库存储时所作的转换。

用法：以org.apache.commons.codec.binary.Base64为例说明;

加密：Base64.encodeBase64URLSafe(byte[])；

解密：Base64.decodeBase64

还有一种用于正则表达式的改进的base64变种，它将”+”和“/”改为了”!”和“-”，因为”+”和”/”在正则表达式中具有特殊意义。

另外还有一些其他变种，都是将”+”和”/”改为其他字符用来避免和其他规范的冲突。

# 3DES

加密问题：

加密一般分为对称加密(Symmetric key encryption)和非对称加密(Asymmetric key encryption);

对称加密（加密的秘钥和解密的秘钥必须完全相同）

分为分组加密和序列加密：

分组加密，也叫块加密（block cyphers），将明文分解成为若干个块，一次加密明文中的一个块。分组时是将明文按一定的位长分组。

加解密过程是;先将明文分成若干组，在将明文组经过加密运算得到密文组，密文组经过解密运算，换原为明文组，再完全还原成明文。

序列加密：也叫流加密（stream cyphers）:一次加密明文中的一个位。是指利用少量的秘钥（制乱元素），通过密码算法产生大量伪随机位流，用于明文位流加密。

解密是用相同的秘钥和密码算法集与加密相同的伪随机位流，用以还原明文位流。

3DES或称为(Triple DES)是三重数据加密算法。相当于对每个数据块应用三次des加密算法。

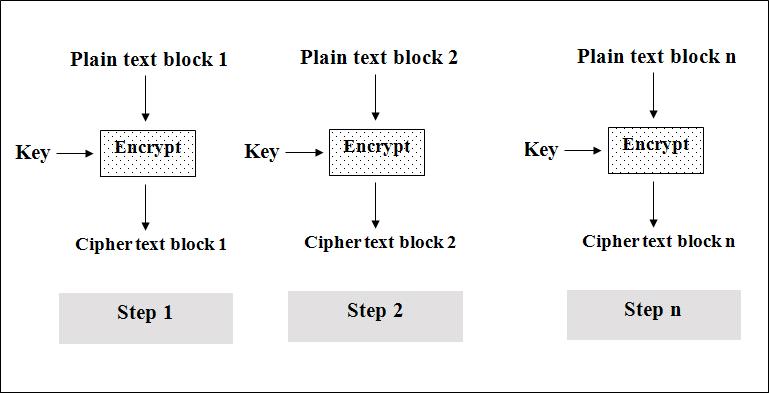
它使用3条56（64位说法不一，以百度百科56位为准）位的秘钥对数据进行加密。DES使用56位秘钥和密码块的方法，而在密码方法中，文本被分为64位大小的文本块然后进行加密。

是一种过渡的加密算法，用于DES到AES的过渡期。其中 AES已经在2006年成为全世界对称加密算法最流行的一种。

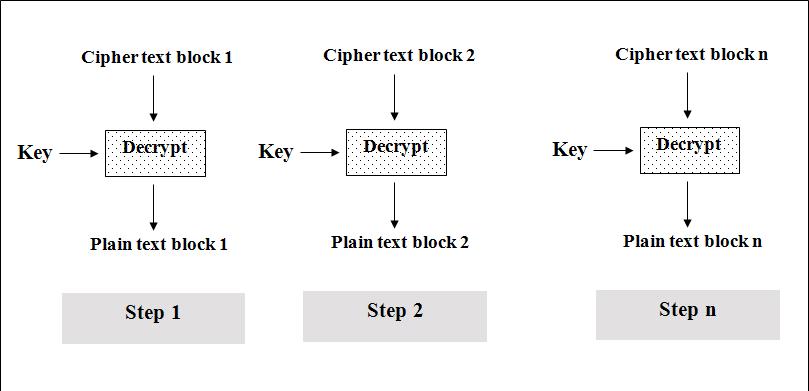
分组加密算法中EBC,CBC,CFB,OFB简介

**EBC（Electronic code book）电码本模式**

加密过程：



解密过程：



优点：

1. 简单
2. 有利于并行运算
3. 误差不会被传递，(如果其中一块加密中出现误差不会影响其他)。**重发机制待补充**

缺点：

1. 不能隐藏明文的模式

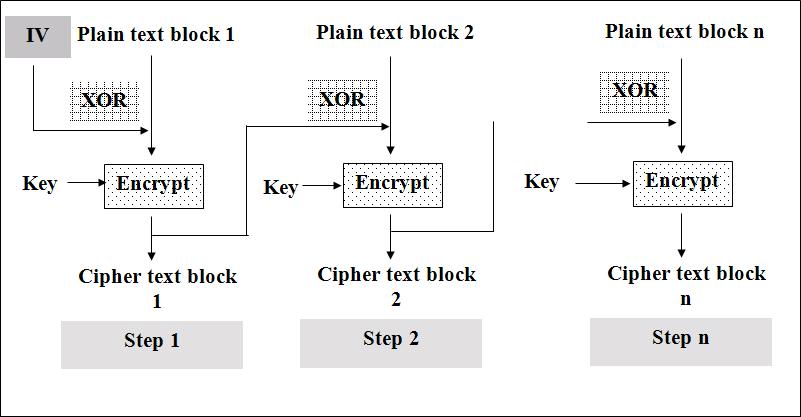
在秘钥中会出现明文消息的重复。

1. 可能对明文进行主动攻击

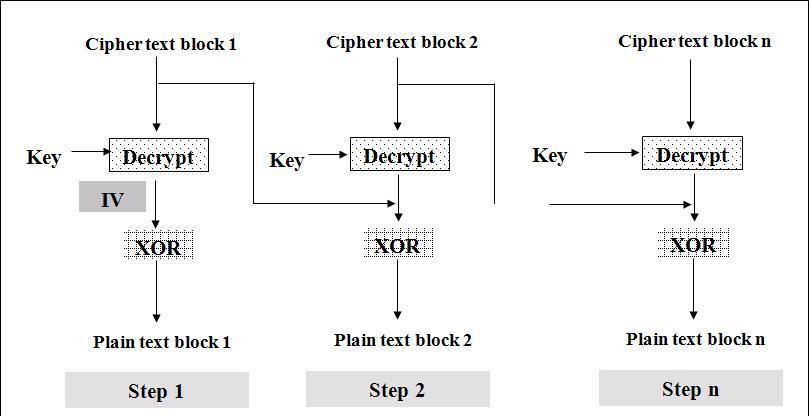
加密消息块相互独立成为被攻击的弱点。（个人理解：只要能解密其中一个，那所有的密文块都会被解密，而且解密其中一个密文组就能得到部分信息）。

**CBC(cipher block chauning)密文分组连接（中信银行加密用到的）**

加密过程



解密过程：



这种方式在一个明文块被加密之前要与前一个密文分组进行异或运算。使用这种方式加密除了秘钥意外，还要协商一个初始化向量。

注：这个向量只在第一次计算时用到，并没有实际意义。

优点：

不容易主动攻击，安全性比ECB要好，适合传输长度较长的报文，是ssl，ipsec标准。

注：每个密码块都依赖于其他密码块，明文消息一个改变会影响所有密文块。

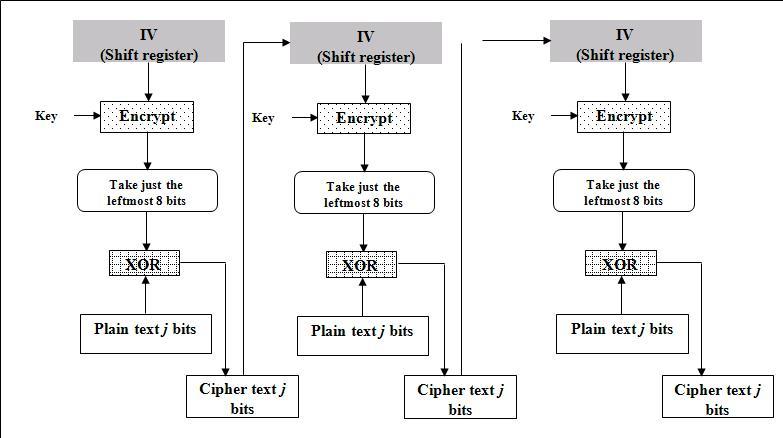
缺点：

不能并行处理。只能串行。（解密时可以并行化，解密时需要的是前一个密文组的信息而不需要前一个明文的信息）。

误差会被传递。

CFB(Cipher feedback) 密文反馈；

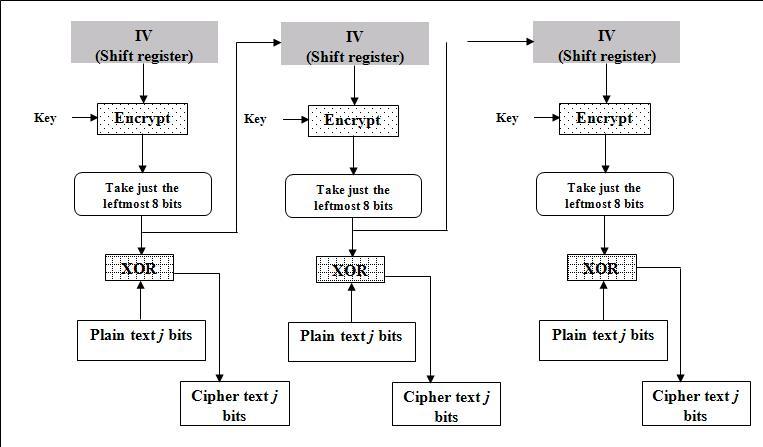
CFB类似CBC，可以将块密码变为自同步流密码；CFB解密过程几乎是颠倒的CBC加密过程；



需要使用一个与块的大小相同的移位寄存器，并用IV将寄存器初始化。然后，将寄存器内容使用块密码加密，然后将结果的最高x位与明文的x进行异或，以产生密文的x位。下一步将生成的x位密文移入寄存器中，并对下面的x位平文重复这一过程。解密过程与加密过程相似，以IV开始，对寄存器加密，将结果的高x与密文异或，产生x位明文，再将密文的下面x位移入寄存器。  
与CBC相似，明文的改变会影响接下来所有的密文，因此加密过程不能并行化；而同样的，与CBC类似，解密过程是可以并行化的。

OFB(output feedback)输出反馈

加密过程：

  
输出反馈模式（Output feedback, OFB）可以将块密码变成同步的流密码。它产生密钥流的块，然后将其与明文块进行异或，得到密文。与其它流密码一样，密文中一个位的翻转会使明文中同样位置的位也产生翻转。这种特性使得许多错误校正码，例如奇偶校验位，即使在加密前计算而在加密后进行校验也可以得出正确结果。  
每个使用OFB的输出块与其前面所有的输出块相关，因此不能并行化处理。然而，由于明文和密文只在最终的异或过程中使用，因此可以事先对IV进行加密，最后并行的将明文或密文进行并行的异或处理。  
可以利用输入全0的CBC模式产生OFB模式的密钥流。这种方法十分实用，因为可以利用快速的CBC硬件实现来加速OFB模式的加密过程。

使用说明（以3DES的CBC模式为例）：

**public** **static** **byte**[] des3ENCDecCBC(**byte**[] key, **byte**[] keyiv, **byte**[] data,

**int** mode) **throws** Exception {

Key deskey = **null**;

DESedeKeySpec spec = **new** DESedeKeySpec(key);

SecretKeyFactory keyfactory = SecretKeyFactory.*getInstance*("desede");

deskey = keyfactory.generateSecret(spec);

Cipher cipher = Cipher.*getInstance*("desede/CBC/PKCS5Padding");

// 注意desede后边的/CBC/PKCS5Padding部分不能省略，一旦省略则加解密时默认使用ECB

IvParameterSpec ips = **new** IvParameterSpec(keyiv);

cipher.init(mode, deskey, ips);

**return** cipher.doFinal(data);

}

其中key是密码，keyiv是偏移量，data是密文。Mode是加解密标识（加密：Cipher.ENCRYPT\_MODE，解密：Cipher.DECRYPT\_MODE，不可以使用其他值）;

注：密码，偏移量和密文。都必须先转换为二进制数组。

注：PKCS5Padding表示block大小是8位。

3des加密时对密码有长度限制

# RSA

RSA 是目前最有影响力的公钥加密算法，它能抵抗到目前为止绝大多数的密码攻击。被ISO推荐为公钥数据加密标准。

RSA算法基础：将两个大素数相乘十分容易，但是想要对其乘积进行因式分解却极其困难。因此可以将乘积公开作为加密的秘钥。

RSA公开秘钥密码体制，所谓公开的秘钥密码体制就是使用不同的加密秘钥与解密秘钥，是一种“由已知加密秘钥导出解密秘钥在计算机上不可行的”密码体制。

注：个人理解。即使得到加密秘钥也无法推导出解密秘钥。

加密秘钥（即公开秘钥）pk是公开信息，解密秘钥（即秘密秘钥）是需要保密的。加密算法E和解密算法D也是公开的。虽然解密秘钥SK是由公开秘钥PK决定的但却不能根据PK计算出来。

RSA算法通常先生成一对RSA秘钥，其中之一是保密秘钥由用户保存，另一个是公开秘钥，可对外公开，甚至可在网络服务器中注册。为了提高保密强度，RSA密码至少500位长，一般推荐为1024位。

但是密码长度越长加密的计算量也就越大，所以为了提高效率，减少计算量，加密时一般采用传统对称加密和公开秘钥加密相结合的方式。即：先用传统的对称加密使用密码对信息进行加密，然后采用RSA秘钥对密码进行加密。对方收到信息后，用不同的秘钥解密 并核对信息摘要。

RSA算法是第一个能同时用于加密和数字签名的算法（中信银行采用私钥进行签名，使用公钥对3des的加密时用到的密码进行加密）；

使用RSA加密时对代加密的字符串有长度限制，以1024位为例，则待加密字符串的长度不能超过117位。计算公式：1024 / 8 - 11 = 117。所以如果需要加密长字符串就必须使用分片加密。 **分片加密待查**

数字证书集合了众多的密码算法；自身带有公钥信息，可以完成相应的加密和解密操作，同时，还带有数字签名可以鉴别消息来源。自身带有消息摘要信息，可以验证证书完整性，证书含有用户身份信息，因而具有认证性。

数字证书又被称为电子证书。网络用户的电子证书需要由数字证书颁发机构（Certificate Authority）签发。只有经过CA签发的证书在网络中才具备可认证性。

证书的签发过程实际上是对申请数字证书的公钥做数字签名，证书的验证过程实际上是对数字证书的公钥做验证签名，其中还包括证书有效期验证。

数字证书是采用了公钥基础设施(public key infrastructure,PKI)，使用了相应的加密算法确保网络应用安全性。

除了RSA之外还可以使用DSA算法，但是使用DSA算法无法实现加解密。即使用DSA算法的证书无法实现加解密功能。

PKCS（public-key cryptography standards，公钥加密标准）最常用的由三个

公钥加密标准 描述 文件名后缀

PKCS#7 密码消息语法标准 .p7b , .p7c ,.spc

PKCS#10 证书请求语法标准 .p10 , .csr

PKCS#12 个人信息交换语法标准 .p12 , .pfx

我们用到的是PKCS#12 ，PKCS#12文件可以最为java中秘钥库或信任库直接使用。

另外：通常使用base64编码格式作为数字证书文件存储格式。

构建自签名证书：

keytool -genkeypair -keyalg RSA -keysize 1024 -sigalg MD5withRSA -validity 36000 -alias www.zlex.org -keystore E:\cer\zlex.keystore -dname "CN=www.zlex.org,OU=zlex,O=zlex,L=BJ,ST=BJ,C=CN" -storepass 123456

参数说明：

-genkeypair 该参数表示生成秘钥。

-keyalg 指定秘钥算法，此处使用RSA, 还可以使用DSA（只是DSA不具有加解密功能），而且DSA是默认的算法

-keysize 秘钥长度，默认1024位，

-sigalg 指定签名算法，此处使用MD5withRSA。

-validity 指定证书有效期，这里指定36000天

-alias 指定别名。（在java中调用时用到了）

-keystore 指定秘钥库的存放位置

-storepass 指定密码

-dname 指定用户信息

注：这里的用户姓名并不是指现代意义中的用户个体，而是指网络环境中的个体，因此通常使用域名或带有通配符“\*”的泛域名，如“\*.zlex.org”

注： 第一个CN 表示姓名；

OU 表示组织单位名称

O表示组织名称

L 表示城市或区域名称

ST 表示州或省份名称

C 表示国家代码

**注意：命令必须全部使用英文符号。**

以上命令可以在秘钥库中创建数字证书，虽然没有经过CA认证但并不影响自己使用。

导出命令如下：

keytool -exportcert -alias www.zlex.org -storepass 123456 -keystore E:\cer\zlex.keystore -file E:\cer\zlex.cer -rfc

参数介绍

-exportcert 表示导出证书

-alias 指定别名

-keystore 指定秘钥库文件，需要从秘钥库文件中导出秘钥

-file 指定导出文件的文件名和路径

-rfc 指定以BASE64位编码格式输出.

-storepass 指定密码

注意：这里通过keytool工具导出的证书，是一个自签名的X.509第三版类型的根证书，并以base64编码保存。自签名证书虽然能使用，但未经CA机构认证，几乎没有任何法律效力。

使用自签名证书时，我们需要将其导入系统

另：可以通过命令打印出证书内容，命令如下

keytool -printcert -file E:\cer\zlex.cer

参数说明：

-printcert 表示打印证书内容

-file 表示证书所在的位置和名称

获得签发证书之后需要将证书导入信任库中。以及使用openssl管理证书由于暂时未用到并为深入研究，以后补上。

证书的使用

从java6，便提供了完善的数字证书管理实现。无需关注具体是算法，仅通过操作秘钥库和数字证书就可以完成相应的加解密和签名研签操作。秘钥库管理私钥，数字证书管理公钥。公钥和私钥分属消息的传递两方，进行加密消息传递。

以下以keytool产生的秘钥库和证书为例，演示证书使用的相关操作。加载秘钥库需要提供秘钥库文件的路径和秘钥库密码。

首先获取秘钥库

/\*\*

\* 将秘钥库实例化

\*

\* **@param** keyStorePath

\* **@param** password

\* **@return**

\*/

**public** **static** KeyStore getKeyStore(String keyStorePath, String password) {

KeyStore keyStore = **null**;

**try** {

// 实例化秘钥库,此处传入秘钥库类型

keyStore = KeyStore.*getInstance*(KeyStore.*getDefaultType*());

FileInputStream is = **new** FileInputStream(keyStorePath);

keyStore.load(is, password.toCharArray());

is.close();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** keyStore;

}

然后从秘钥库中获取私钥

/\*\*

\* 获取私钥

\*

\* **@param** keyStorePath

\* **@param** password

\* **@return**

\*/

**public** **static** PrivateKey getProvateKey(String keyStorePath,

String password, String alias) {

PrivateKey privateKey = **null**;

**try** {

KeyStore keyStore = *getKeyStore*(keyStorePath, password);

// 此处传入的是私钥的别名和的密码

privateKey = (PrivateKey) keyStore.getKey(alias,

password.toCharArray());

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** privateKey;

}

获取私钥以后就可以实现加解密操作。

如果需要从秘钥库中获取签名算法，只能通过从秘钥库获取数字证书并强转成X509Certificate实例，通过getSigAlgName()方法获得对应的签名算法。

代码如下：

/\*\*

\* 获取签名算法

\*

\* **@param** keyStorePath

\* **@param** password

\* **@param** alias

\* **@return**

\*/

**public** **static** Signature getSignature(String keyStorePath, String password,

String alias) {

Signature signature = **null**;

**try** {

KeyStore keyStore = *getKeyStore*(keyStorePath, password);

X509Certificate x509Certificate = (X509Certificate) keyStore

.getCertificate(alias);

signature = Signature.*getInstance*(x509Certificate.getSigAlgName());

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** signature;

}

数字证书操作比秘钥库操作更为简单，但是java6只支持X.509类型的数字证书.

Java代码加载公钥如下所示:

/\*\*

\* 从外部读取数字证书

\*

\* **@param** certificatePath

\* **@return**

\*/

**public** **static** Certificate getCertificate(String certificatePath) {

Certificate certificate = **null**;

**try** {

CertificateFactory certificateFactory = CertificateFactory

.*getInstance*("X.509");

FileInputStream in = **new** FileInputStream(certificatePath);

certificate = certificateFactory.generateCertificate(in);

in.close();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** certificate;

}

将数字证书实例化以后，从数字证书中得到公钥

代码如下：

/\*\*

\* 从数字证书中得到公钥

\*

\* **@param** certificatePath

\* **@return**

\*/

**public** **static** PublicKey getPublicKey(String certificatePath) {

PublicKey publicKey = **null**;

Certificate certificate = *getCertificate*(certificatePath);

publicKey = certificate.getPublicKey();

**return** publicKey;

}

使用数字证书进行研签的时候，需要将证书进行强转，强转成X509Certificate

具体如下:

**public** **static** Signature getSignature(String certificatePath) {

Signature signature = **null**;

**try** {

X509Certificate certificate = (X509Certificate) *getCertificate*(certificatePath);

signature = Signature.*getInstance*(certificate.getSigAlgName());

signature.initVerify(certificate);

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** signature;

}

注意：数字签名对象初始化（initVerify方法）传入的对象时数字证书而不是公钥。但是实际上是使用数字证书的公钥。

得到证书的签名算法之后就可以进行研签操作。

测试代码如下：

加解密工具类：

package RSATestCER;

import java.io.FileInputStream;

import java.security.KeyStore;

import java.security.KeyStoreException;

import java.security.PrivateKey;

import java.security.PublicKey;

import java.security.Signature;

import java.security.cert.Certificate;

import java.security.cert.CertificateFactory;

import java.security.cert.X509Certificate;

import javax.crypto.Cipher;

public class CertificateCoder {

// 证书的类型X509

public static final String CERT\_TYPE = "X.509";

/\*\*

\* 由keystore获得私钥

\*

\* @param keyStorePath

\* 秘钥库地址

\* @param alias

\* 别名

\* @param password

\* 密码

\* @return PrivateKey 私钥

\*/

private static PrivateKey getPrivateKeyByKeyStore(String keyStorePath,

String alias, String password) {

PrivateKey privateKey = null;

try {

// 获取外部秘钥库

KeyStore ks = getKeyStore(keyStorePath, password);

privateKey = (PrivateKey) ks.getKey(alias, password.toCharArray());

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return privateKey;

}

/\*\*

\* 通过证书获取公钥

\*

\* @param certificatePath

\* 证书目录

\* @return PublicKey 公钥

\*/

private static PublicKey getPublicKeyByCertificate(String certificatePath) {

PublicKey publicKey = null;

Certificate certificate = getCertificate(certificatePath);

publicKey = certificate.getPublicKey();

return publicKey;

}

/\*\*

\* 从外部文件获取证书

\*

\* @param certificatePath

\* 证书路径

\* @return Certificate 证书实例

\*/

private static Certificate getCertificate(String certificatePath) {

Certificate certificate = null;

try {

CertificateFactory certificateFactory = CertificateFactory

.getInstance(CERT\_TYPE);

FileInputStream in = new FileInputStream(certificatePath);

certificate = (Certificate) certificateFactory

.generateCertificate(in);

in.close();

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return certificate;

}

/\*\*

\* 从秘钥库获取证书

\*

\* @param keyStorePath

\* 秘钥库位置

\* @param alias

\* 别名

\* @param password

\* 密码

\* @return Certificate 证书

\*/

private static Certificate getCertificate(String keyStorePath,

String alias, String password) {

Certificate certificate = null;

try {

KeyStore ks = getKeyStore(keyStorePath, password);

certificate = (Certificate) ks.getCertificate(alias);

} catch (KeyStoreException e) {

e.printStackTrace();

}

return certificate;

}

/\*\*

\* 从外部读取秘钥库

\*

\* @param keytorePath

\* 秘钥库位置

\* @param password

\* 密码

\* @return KeyStore 秘钥库实例

\*/

private static KeyStore getKeyStore(String keytorePath, String password) {

KeyStore ks = null;

try {

ks = KeyStore.getInstance(KeyStore.getDefaultType());

FileInputStream in = new FileInputStream(keytorePath);

ks.load(in, password.toCharArray());

in.close();

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return ks;

}

/\*\*

\* 利用私钥进行加密

\*

\* @param data

\* 待加密数据

\* @param keyStorePath

\* 秘钥库的路径

\* @param alias

\* 别名

\* @param password

\* 密码

\* @return byte[] 加密后的数据

\*/

public static byte[] encryptByPrivateKey(byte[] data, String keyStorePath,

String alias, String password) {

byte[] bytes = null;

try {

PrivateKey privateKey = getPrivateKeyByKeyStore(keyStorePath,

alias, password);

Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());

cipher.init(Cipher.ENCRYPT\_MODE, privateKey);

bytes = cipher.doFinal(data);

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return bytes;

}

/\*\*

\* 私钥解密

\*

\* @param data

\* 待解密数据

\* @param keyStorePath

\* 秘钥库地址

\* @param alias

\* 别名

\* @param password

\* 密码

\* @return 解密后的数据

\*/

public static byte[] decryptByPrivateKey(byte[] data, String keyStorePath,

String alias, String password) {

byte[] bytes = null;

try {

PrivateKey privateKey = getPrivateKeyByKeyStore(keyStorePath,

alias, password);

Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());

cipher.init(Cipher.DECRYPT\_MODE, privateKey);

bytes = cipher.doFinal(data);

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return bytes;

}

/\*\*

\* 公钥加密

\*

\* @param data

\* 带数据加密

\* @param certificatePath

\* 证书路径

\* @return byte[] 加密数据

\*/

public static byte[] encryptByPublicKey(byte[] data, String certificatePath) {

byte[] bytes = null;

try {

PublicKey publicKey = getPublicKeyByCertificate(certificatePath);

Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());

cipher.init(Cipher.ENCRYPT\_MODE, publicKey);

bytes = cipher.doFinal(data);

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return bytes;

}

/\*\*

\* 公钥解密

\*

\* @param data

\* 待解密数据

\* @param certificatePath

\* 证书路径

\* @return byte[] 解密后的数据

\*/

public static byte[] decryptByPublicKey(byte[] data, String certificatePath) {

byte[] bytes = null;

try {

PublicKey publicKey = getPublicKeyByCertificate(certificatePath);

Cipher cipher = Cipher.getInstance(publicKey.getAlgorithm());

cipher.init(Cipher.DECRYPT\_MODE, publicKey);

bytes = cipher.doFinal(data);

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return bytes;

}

/\*\*

\* 签名

\*

\* @param sign

\* 待签名数据

\* @param keyStorePath

\* 秘钥库位置

\* @param alias

\* 别名

\* @param password

\* 秘钥

\* @return byte[] 签名得到的数据

\*/

public static byte[] sign(byte[] sign, String keyStorePath, String alias,

String password) {

byte[] bytes = null;

Signature signature;

try {

X509Certificate x509Certificate = (X509Certificate) getCertificate(

keyStorePath, alias, password);

signature = Signature.getInstance(x509Certificate.getSigAlgName());

PrivateKey privateKey = getPrivateKeyByKeyStore(keyStorePath,

alias, password);

signature.initSign(privateKey);

signature.update(sign);

bytes = signature.sign();

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return bytes;

}

/\*\*

\* 验证签名

\*

\* @param data

\* 原始数据

\* @param sign

\* 签名

\* @param certificatePath

\* 证书位置

\* @return boolean 比对结果。

\*/

public static boolean verify(byte[] data, byte[] sign,

String certificatePath) {

boolean flag = false;

X509Certificate certificate = (X509Certificate) getCertificate(certificatePath);

try {

Signature signature = Signature.getInstance(certificate

.getSigAlgName());

signature.initVerify(certificate);

signature.update(data);

flag = signature.verify(sign);

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return flag;

}

}

测试类代码如下：

**package** RSATestCER;

**public** **class** TestRSA {

**private** String password = "123456";

**private** String alias = "www.zlex.org";

**private** String certificatePath = "E:\\cer\\zlex.cer";

**private** String keyStorePath = "E:\\cer\\zlex.keystore";

**public** **void** test1(String inputStr) {

System.***out***.println("公钥加密 - 私钥解密 测试");

**byte**[] data = inputStr.getBytes();

// 公钥加密

**byte**[] encrypt = CertificateCoder.*encryptByPublicKey*(data,

certificatePath);

// 私钥进行解密

**byte**[] decrypt = CertificateCoder.*decryptByPrivateKey*(encrypt,

keyStorePath, alias, password);

String outPutstr = **new** String(decrypt);

System.***out***.println("加密前：" + inputStr);

System.***out***.println("加密后:" + **new** String(encrypt));

System.***out***.println("解密后:" + outPutstr);

// assertArrayEquals(data, decrypt);

}

**public** **void** test2(String inputStr) {

System.***out***.println("私钥加密--公钥解密");

**byte**[] data = inputStr.getBytes();

// 私钥进行加密

**byte**[] encoded = CertificateCoder.*encryptByPrivateKey*(data,

keyStorePath, alias, password);

// 公钥加密

**byte**[] decoded = CertificateCoder.*decryptByPublicKey*(encoded,

certificatePath);

String outPutStr = **new** String(decoded);

System.***out***.println("加密前：" + inputStr);

System.***out***.println("加密后: " + **new** String(decoded));

System.***out***.println("解密后： " + outPutStr);

}

**public** **void** testSign(String inputStr) {

System.***out***.println("私钥签名--公钥验签");

**byte**[] data = inputStr.getBytes();

// 私钥签名

**byte**[] sign = CertificateCoder

.*sign*(data, keyStorePath, alias, password);

System.***out***.println("签名：" + **new** String(sign));

// 公钥验证签名

**boolean** flag = CertificateCoder.*verify*(data, sign, certificatePath);

System.***out***.println("对比结果是:" + flag);

}

}

注：使用RSA编码之后得到的byte[] 必须经过base64转换之后才能使用转换为字符转，否则会出现乱码！原因待定！