如基本的单向加密算法：

* BASE64 严格地说，属于编码格式，而非加密算法
* MD5(Message Digest algorithm 5，信息摘要算法)
* SHA(Secure Hash Algorithm，安全散列算法)
* HMAC(Hash Message Authentication Code，散列消息鉴别码)

复杂的对称加密（DES、PBE）、非对称加密算法：

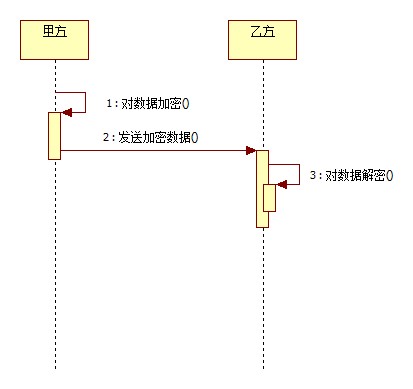
* DES(Data Encryption Standard，数据加密算法)
* PBE(Password-based encryption，基于密码验证)
* RSA(Ron Rivest, AdiShamir和Leonard Adleman,发明者名字命名)
* DH(Diffie-Hellman算法，密钥一致协议)
* DSA(Digital Signature Algorithm，数字签名)
* ECC(Elliptic Curves Cryptography，椭圆曲线密码编码学)

BASE64的加密解密是双向的，可以求反解。 MD5、SHA以及HMAC是单向加密，任何数据加密后只会产生唯一的一个加密串，通常用来校验数据在传输过程中是否被修改。其中HMAC算法有一个密钥，增强了数据传输过程中的安全性，强化了算法外的不可控因素。 **单向加密的用途主要是为了校验数据在传输过程中是否被修改。**

“数字签名”就是将原文通过MD5（或者SHA等不可逆加密）算法，生成一堆摘要消息MD。然后把这段摘要消息加到原文后面，然后对这整个信息进行再使用y私钥进行加密生成密文并发送给客户。客户拿到密文后使用x公钥进行解密，获得原文和摘要消息MD，然后客户再对原文进行MD5生成摘要消息，比对这段摘要消息和MD进行比较，如果相同说明信息完整，不相同就说明信息不完整，然后请求重发。

1）**BASE64**

按照RFC2045的定义，Base64被定义为：Base64内容传送编码被设计用来把任意序列的8位字节描述为一种不易被人直接识别的形式。（The Base64 Content-Transfer-Encoding is designed to represent arbitrary sequences of octets in a form that need not be humanly readable.）。常见于邮件、http加密，截取http信息，你就会发现登录操作的用户名、密码字段通过BASE64加密的。



通过java代码实现如下：

/\*\*

\* BASE64解密

\*/

**public** **static** **byte**[] decryptBASE64(String key) **throws** Exception {

**return** (**new** BASE64Decoder()).decodeBuffer(key);

}

/\*\*

\* BASE64加密

\*/

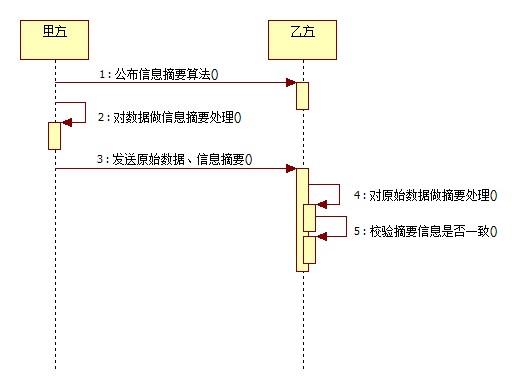
**public** **static** String encryptBASE64(**byte**[] key) **throws** Exception {

**return** (**new** BASE64Encoder()).encodeBuffer(key);

}

2）**MD5**

MD5 -- message-digest algorithm 5（信息-摘要算法）缩写，广泛用于加密和解密技术，常用于文件校验。校验？不管文件多大，经过MD5后都能生成唯一的MD5值。好比现在的ISO校验，都是MD5校验。通常我们不直接使用MD5加密。通常将MD5产生的字节数组交给BASE64再加密一把，得到相应的字符串。



通过java代码实现如下：

/\*\*

\* MD5加密

\*/

**public** **static** **byte**[] encryptMD5(**byte**[] data) **throws** Exception{

MessageDigest md5 = MessageDigest.getInstance(*KEY\_MD5*);

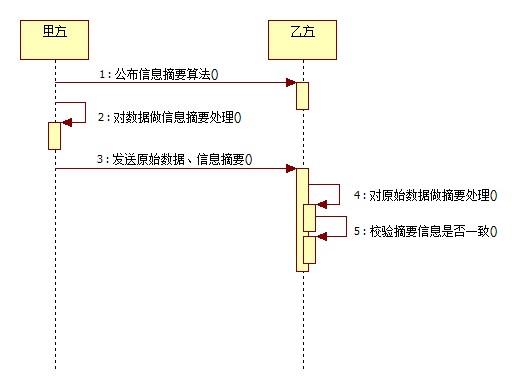
md5.update(data);

**return** md5.digest();

}

3）**SHA**

SHA(Secure Hash Algorithm，安全散列算法），数字签名等密码学应用中重要的工具，被广泛地应用于电子商务等信息安全领域。虽然，SHA与MD5通过碰撞法都被破解了，但是SHA仍然是公认的安全加密算法，较之MD5更为安全。



通过java代码实现如下：

/\*\*

\* SHA加密

\*/

**public** **static** **byte**[] encryptSHA(**byte**[] data) **throws** Exception{

MessageDigest sha = MessageDigest.getInstance(*KEY\_SHA*);

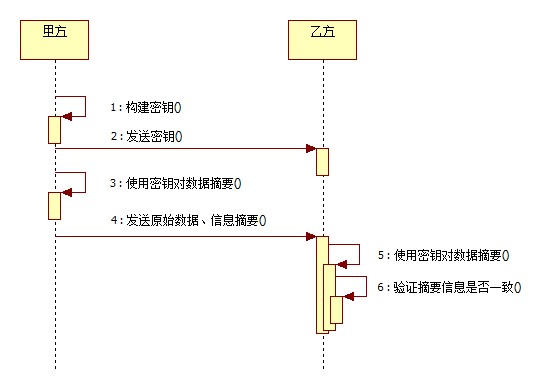
sha.update(data);

**return** sha.digest();

}

4）**HMAC**

HMAC(Hash Message Authentication Code，散列消息鉴别码，基于密钥的Hash算法的认证协议。消息鉴别码实现鉴别的原理是，用公开函数和密钥产生一个固定长度的值作为认证标识，用这个标识鉴别消息的完整性。使用一个密钥生成一个固定大小的小数据块，即MAC，并将其加入到消息中，然后传输。接收方利用与发送方共享的密钥进行鉴别认证等。



通过java代码实现如下：

/\*\*

\* 初始化HMAC密钥

\*/

**public** **static** String initMacKey() **throws** Exception {

KeyGenerator keyGenerator = KeyGenerator.getInstance(*KEY\_MAC*);

SecretKey secretKey = keyGenerator.generateKey();

**return** encryptBASE64(secretKey.getEncoded());

}

/\*\*

\* HMAC加密

\*/

**public** **static** **byte**[] encryptHMAC(**byte**[] data, String key) **throws** Exception {

SecretKey secretKey = **new** SecretKeySpec(decryptBASE64(key), *KEY\_MAC*);

Mac mac = Mac.getInstance(secretKey.getAlgorithm());

mac.init(secretKey);

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

}

5）**DES**

DES-Data Encryption Standard,即数据加密算法。DES算法的入口参数有三个:**Key、Data、Mode**。其中Key为8个字节共64位,是DES算法的工作密钥;Data也为8个字节64位,是要被加密或被解密的数据;Mode为DES的工作方式,有两种:加密或解密。DES算法把64位的明文输入块变为64位的密文输出块,它所使用的密钥也是64位。

通过java代码实现如下：

/\*\*

\* 生成密钥

\*/

**public** **static** String initKey(String seed) **throws** Exception {

SecureRandom secureRandom = **null**;

**if** (seed != **null**) {

secureRandom = **new** SecureRandom(decryptBASE64(seed));

} **else** {

secureRandom = **new** SecureRandom();

}

KeyGenerator kg = KeyGenerator.getInstance(*ALGORITHM*);

kg.init(secureRandom);

SecretKey secretKey = kg.generateKey();

**return** encryptBASE64(secretKey.getEncoded());

}

/\*\*

\* 转换密钥

\*/

**private** **static** Key toKey(byte[] key) **throws** Exception {

DESKeySpec dks = **new** DESKeySpec(key);

SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(*ALGORITHM*);

SecretKey secretKey = keyFactory.generateSecret(dks);

// 当使用其他对称加密算法时，如AES、Blowfish等算法时，用下述代码替换上述三行代码

// SecretKey secretKey = new SecretKeySpec(key, ALGORITHM);

**return** secretKey;

}

/\*\*

\* 解密

\*/

**public** **static** **byte**[] decrypt(byte[] data, String key) **throws** Exception {

Key k = toKey(decryptBASE64(key));

Cipher cipher = Cipher.getInstance(*ALGORITHM*);

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

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

}

/\*\*

\* 加密

\*/

**public** **static** **byte**[] encrypt(byte[] data, String key) **throws** Exception {

Key k = toKey(decryptBASE64(key));

Cipher cipher = Cipher.getInstance(*ALGORITHM*);

cipher.init(Cipher.*ENCRYPT\_MODE*, k);

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

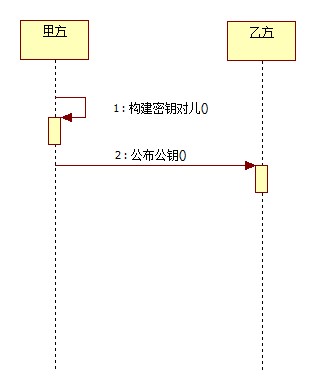
}

6）**RSA**

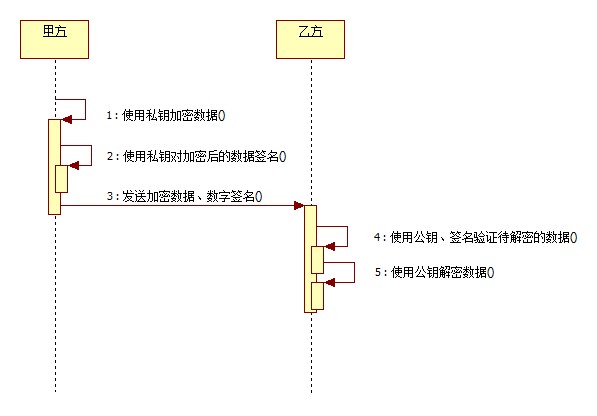
**它是一个既能用于数据加密也能用于数字签名的算法**。它易于理解和操作，也很流行。算法的名字以发明者的名字命名：Ron Rivest, AdiShamir 和Leonard Adleman。 这种加密算法的特点主要是密钥的变化，上文我们看到DES只有一个密钥。相当于只有一把钥匙，如果这把钥匙丢了，数据也就不安全了。RSA同时有两把钥匙，公钥与私钥。同时支持数字签名。数字签名的意义在于，对传输过来的数据进行校验。确保数据在传输工程中不被修改。

流程分析：

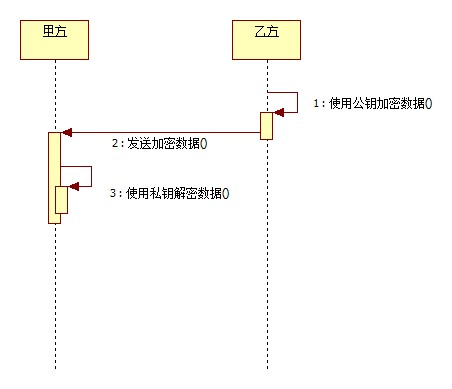
1.甲方构建密钥对儿，将公钥公布给乙方，将私钥保留。



2.甲方使用私钥加密数据，然后用私钥对加密后的数据签名，发送给乙方签名以及加密后的数据；乙方使用公钥、签名来验证待解密数据是否有效，如果有效使用公钥对数据解密。



3.乙方使用公钥加密数据，向甲方发送经过加密后的数据；甲方获得加密数据，通过私钥解密。



通过java代码实现如下：

/\*\*

\* 用私钥对信息生成数字签名

\*/

**public** **static** String sign(**byte**[] data, String privateKey) **throws** Exception {

// 解密由base64编码的私钥

**byte**[] keyBytes = decryptBASE64(privateKey);

// 构造PKCS8EncodedKeySpec对象

PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(keyBytes);

// KEY\_ALGORITHM 指定的加密算法

KeyFactory keyFactory = KeyFactory.getInstance(*KEY\_ALGORITHM*);

// 取私钥匙对象

PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);

// 用私钥对信息生成数字签名

Signature signature = Signature.getInstance(*SIGNATURE\_ALGORITHM*);

signature.initSign(priKey);

signature.update(data);

**return** encryptBASE64(signature.sign());

}

/\*\*

\* 校验数字签名

\*/

**public** **static** **boolean** verify(**byte**[] data, String publicKey, String sign) **throws** Exception {

// 解密由base64编码的公钥

**byte**[] keyBytes = decryptBASE64(publicKey);

// 构造X509EncodedKeySpec对象

X509EncodedKeySpec keySpec = **new** X509EncodedKeySpec(keyBytes);

// KEY\_ALGORITHM 指定的加密算法

KeyFactory keyFactory = KeyFactory.getInstance(*KEY\_ALGORITHM*);

// 取公钥匙对象

PublicKey pubKey = keyFactory.generatePublic(keySpec);

Signature signature = Signature.getInstance(*SIGNATURE\_ALGORITHM*);

signature.initVerify(pubKey);

signature.update(data);

// 验证签名是否正常

**return** signature.verify(decryptBASE64(sign));

}

/\*\*

\* 用私钥解密

\*/

**public** **static** **byte**[] decryptByPrivateKey(**byte**[] data, String key) **throws** Exception {

// 对密钥解密

**byte**[] keyBytes = decryptBASE64(key);

// 取得私钥

PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(keyBytes);

KeyFactory keyFactory = KeyFactory.getInstance(*KEY\_ALGORITHM*);

Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);

// 对数据解密

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

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

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

}

/\*\*

\* 用公钥解密

\*/

**public** **static** **byte**[] decryptByPublicKey(**byte**[] data, String key) **throws** Exception {

// 对密钥解密

**byte**[] keyBytes = decryptBASE64(key);

// 取得公钥

X509EncodedKeySpec x509KeySpec = **new** 509EncodedKeySpec(keyBytes);

KeyFactory keyFactory = KeyFactory.getInstance(*KEY\_ALGORITHM*);

Key publicKey = keyFactory.generatePublic(x509KeySpec);

// 对数据解密

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

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

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

}

/\*\*

\* 用公钥加密

\*/

**public** **static** **byte**[] encryptByPublicKey(**byte**[] data, String key) throws Exception {

// 对公钥解密

**byte**[] keyBytes = decryptBASE64(key);

// 取得公钥

X509EncodedKeySpec x509KeySpec = **new** X509EncodedKeySpec(keyBytes);

KeyFactory keyFactory = KeyFactory.getInstance(*KEY\_ALGORITHM*);

Key publicKey = keyFactory.generatePublic(x509KeySpec);

// 对数据加密

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

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

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

}

/\*\*

\* 用私钥加密

\*/

**public** **static** **byte**[] encryptByPrivateKey(**byte**[] data, String key) **throws** Exception {

// 对密钥解密

**byte**[] keyBytes = decryptBASE64(key);

// 取得私钥

PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(keyBytes);

KeyFactory keyFactory = KeyFactory.getInstance(*KEY\_ALGORITHM*);

Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);

// 对数据加密

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

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

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

}

/\*\*

\* 初始化密钥

\*/

**public** **static** Map<String, Object> initKey() **throws** Exception {

KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance(*KEY\_ALGORITHM*);

keyPairGen.initialize(1024);

KeyPair keyPair = keyPairGen.generateKeyPair();

// 公钥

RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();

// 私钥

RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();

Map<String, Object> keyMap = **new** HashMap<String, Object>(2);

keyMap.put(*PUBLIC\_KEY*, publicKey);

keyMap.put(*PRIVATE\_KEY*, privateKey);

**return** keyMap;

}

**数字信封**

数字信封用加密技术来保证只有特定的收信人才能阅读信的内容。

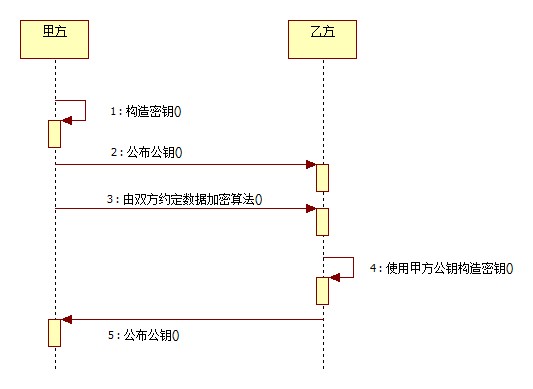
流程：信息发送方采用对称密钥来加密信息，然后再用接收方的公钥来加密此对称密钥（这部分称为数字信封），再将它和信息一起发送给接收方；接收方先用相应的私钥打开数字信封，得到对称密钥，然后使用对称密钥再解开信息。

7）**DH**

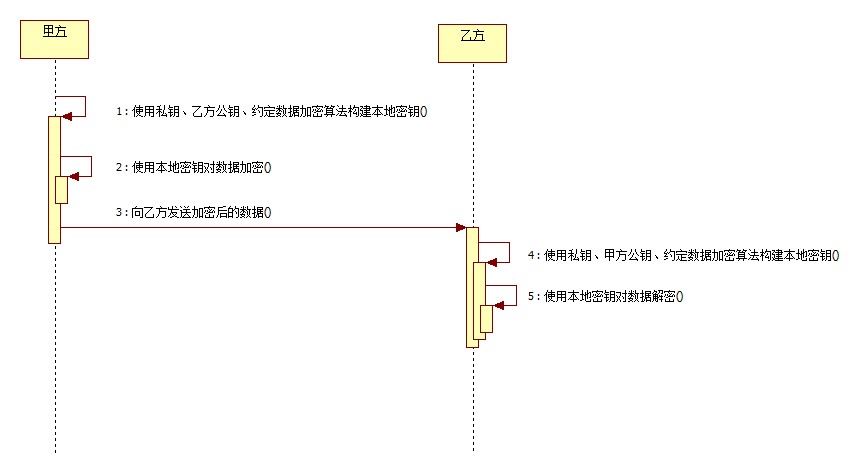
Diffie- Hellman算法(D-H算法)，密钥一致协议。简单的说就是允许两名用户在公开媒体上交换信息以生成"一致"的、可以共享的密钥。换句话说，就是由甲方产出一对密钥（公钥、私钥），乙方依照甲方公钥产生乙方密钥对（公钥、私钥）。以此为基线，作为数据传输保密基础，同时双方使用同一种对称加密算法构建本地密钥（SecretKey）对 数据加密。这样，在互通了本地密钥（SecretKey）算法后，甲乙双方公开自己的公钥，使用对方的公钥和刚才产生的私钥加密数据，同时可以使用对方的公钥和自己的私钥对数据解密。不单单是甲乙双方两方，可以扩展为多方共享数据通讯，这样就完成了网络交互数据的安全通讯！

流程分析：

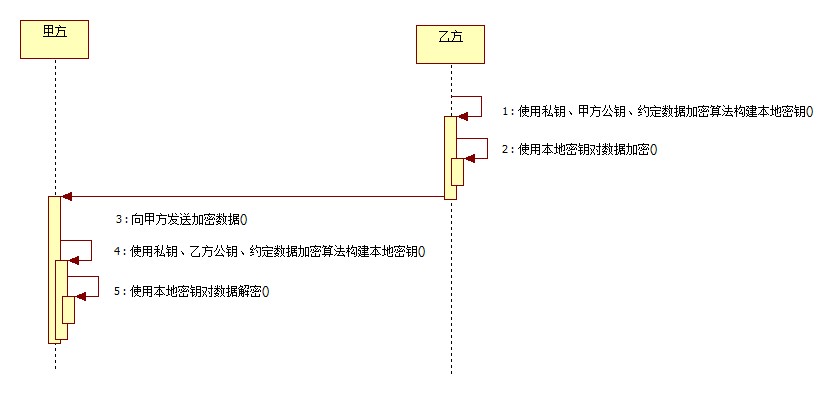
1.甲方构建密钥对儿，将公钥公布给乙方，将私钥保留；双方约定数据加密算法；乙方通过甲方公钥构建密钥对儿，将公钥公布给甲方，将私钥保留。



2.甲方使用私钥、乙方公钥、约定数据加密算法构建本地密钥，然后通过本地密钥加密数据，发送给乙方加密后的数据；乙方使用私钥、甲方公钥、约定数据加密算法构建本地密钥，然后通过本地密钥对数据解密。



3.乙方使用私钥、甲方公钥、约定数据加密算法构建本地密钥，然后通过本地密钥加密数据，发送给甲方加密后的数据；甲方使用私钥、乙方公钥、约定数据加密算法构建本地密钥，然后通过本地密钥对数据解密



通过java代码实现如下：

/\*\*

\* 初始化甲方密钥

\*/

**public** **static** Map<String, Object> initKey() **throws** Exception {

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(*ALGORITHM*);

keyPairGenerator.initialize(*KEY\_SIZE*);

KeyPair keyPair = keyPairGenerator.generateKeyPair();

// 甲方公钥

DHPublicKey publicKey = (DHPublicKey) keyPair.getPublic();

// 甲方私钥

DHPrivateKey privateKey = (DHPrivateKey) keyPair.getPrivate();

Map<String, Object> keyMap = **new** HashMap<String, Object>(2);

keyMap.put(*PUBLIC\_KEY*, publicKey);

keyMap.put(*PRIVATE\_KEY*, privateKey);

**return** keyMap;

}

/\*\*

\* 初始化乙方密钥

\*/

**public** **static** Map<String, Object> initKey(String key) **throws** Exception {

// 解析甲方公钥

**byte**[] keyBytes = decryptBASE64(key);

X509EncodedKeySpec x509KeySpec = **new** X509EncodedKeySpec(keyBytes);

KeyFactory keyFactory = KeyFactory.getInstance(*ALGORITHM*);

PublicKey pubKey = keyFactory.generatePublic(x509KeySpec);

// 由甲方公钥构建乙方密钥

DHParameterSpec dhParamSpec = ((DHPublicKey) pubKey).getParams();

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(keyFactory.getAlgorithm());

keyPairGenerator.initialize(dhParamSpec);

KeyPair keyPair = keyPairGenerator.generateKeyPair();

// 乙方公钥

DHPublicKey publicKey = (DHPublicKey) keyPair.getPublic();

// 乙方私钥

DHPrivateKey privateKey = (DHPrivateKey) keyPair.getPrivate();

Map<String, Object> keyMap = **new** HashMap<String, Object>(2);

keyMap.put(*PUBLIC\_KEY*, publicKey);

keyMap.put(*PRIVATE\_KEY*, privateKey);

**return** keyMap;

}

/\*\*

\* 加密

\*/

**public** **static** **byte**[] encrypt(**byte**[] data, String publicKey, String privateKey) **throws** Exception {

// 生成本地密钥

SecretKey secretKey = getSecretKey(publicKey, privateKey);

// 数据加密

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

cipher.init(Cipher.*ENCRYPT\_MODE*, secretKey);

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

}

/\*\*

\* 解密

\*/

**public** **static** **byte**[] decrypt(**byte**[] data, String publicKey, String privateKey) **throws** Exception {

// 生成本地密钥

SecretKey secretKey = getSecretKey(publicKey, privateKey);

// 数据解密

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

cipher.init(Cipher.*DECRYPT\_MODE*, secretKey);

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

}

/\*\*

\* 构建密钥

\*/

**private** **static** SecretKey getSecretKey(String publicKey, String privateKey) **throws** Exception {

// 初始化公钥

**byte**[] pubKeyBytes = decryptBASE64(publicKey);

KeyFactory keyFactory = KeyFactory.getInstance(*ALGORITHM*);

X509EncodedKeySpec x509KeySpec = **new** X509EncodedKeySpec(pubKeyBytes);

PublicKey pubKey = keyFactory.generatePublic(x509KeySpec);

// 初始化私钥

**byte**[] priKeyBytes = decryptBASE64(privateKey);

PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(priKeyBytes);

Key priKey = keyFactory.generatePrivate(pkcs8KeySpec);

KeyAgreement keyAgree = KeyAgreement.getInstance(keyFactory.getAlgorithm());

keyAgree.init(priKey);

keyAgree.doPhase(pubKey, **true**);

// 生成本地密钥

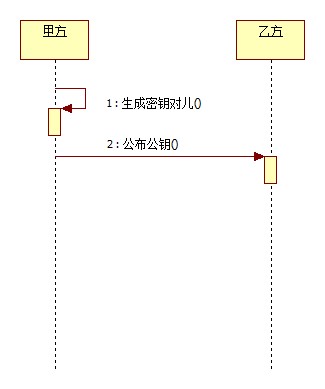
SecretKey secretKey = keyAgree.generateSecret(*SECRET\_ALGORITHM*);

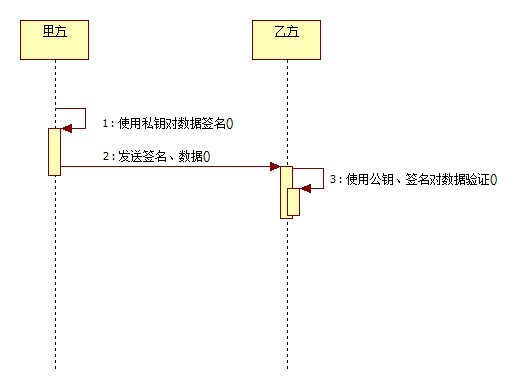
**return** secretKey;

}

8）**DSA**

DSA-Digital Signature Algorithm 是Schnorr和ElGamal签名算法的变种，被美国NIST作为DSS(DigitalSignature Standard)。简单的说，这是一种更高级的验证方式，用作数字签名。不单单只有公钥、私钥，还有数字签名。私钥加密生成数字签名，公钥验证数据及签 名。如果数据和签名不匹配则认为验证失败！数字签名的作用就是校验数据在传输过程中不被修改。数字签名，是单向加密的升级！





通过java代码实现如下：

/\*\*

\* 用私钥对信息生成数字签名

\*/

**public** **static** String sign(**byte**[] data, String privateKey) **throws** Exception {

// 解密由base64编码的私钥

**byte**[] keyBytes = decryptBASE64(privateKey);

// 构造PKCS8EncodedKeySpec对象

PKCS8EncodedKeySpec pkcs8KeySpec = **new** PKCS8EncodedKeySpec(keyBytes);

// KEY\_ALGORITHM 指定的加密算法

KeyFactory keyFactory = KeyFactory.getInstance(*ALGORITHM*);

// 取私钥匙对象

PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);

// 用私钥对信息生成数字签名

Signature signature = Signature.getInstance(keyFactory.getAlgorithm());

signature.initSign(priKey);

signature.update(data);

**return** encryptBASE64(signature.sign());

}

/\*\*

\* 校验数字签名

\*/

**public** **static** **boolean** verify(**byte**[] data, String publicKey,String sign) **throws** Exception {

// 解密由base64编码的公钥

**byte**[] keyBytes = decryptBASE64(publicKey);

// 构造X509EncodedKeySpec对象

X509EncodedKeySpec keySpec = **new** X509EncodedKeySpec(keyBytes);

// ALGORITHM 指定的加密算法

KeyFactory keyFactory = KeyFactory.getInstance(*ALGORITHM*);

// 取公钥匙对象

PublicKey pubKey = keyFactory.generatePublic(keySpec);

Signature signature = Signature.getInstance(keyFactory.getAlgorithm());

signature.initVerify(pubKey);

signature.update(data);

// 验证签名是否正常

**return** signature.verify(decryptBASE64(sign));

}

/\*\*

\* 生成密钥

\*/

**public** **static** Map<String, Object> initKey(String seed) **throws** Exception {

KeyPairGenerator keygen = KeyPairGenerator.getInstance(*ALGORITHM*);

// 初始化随机产生器

SecureRandom secureRandom = **new** SecureRandom();

secureRandom.setSeed(seed.getBytes());

keygen.initialize(*KEY\_SIZE*, secureRandom);

KeyPair keys = keygen.genKeyPair();

DSAPublicKey publicKey = (DSAPublicKey) keys.getPublic();

DSAPrivateKey privateKey = (DSAPrivateKey) keys.getPrivate();

Map<String, Object> map = **new** HashMap<String, Object>(2);

map.put(*PUBLIC\_KEY*, publicKey);

map.put(*PRIVATE\_KEY*, privateKey);

**return** map;

}