视频位置：P:\00\_传智播客\黑马程序员训练营\_公开课视频教程\中关村黑马程序员训练营\_公开课视频教程\_Java安全

# 对称加密与非对称加密

对称加密：加密速度快

非对称加密：相比对称加密算法上要更复杂，所以加密速度慢

非对称加密应用场景：多个人可以加密，但是只有一个人有解密。比如我们访问工商银行，数据传给工商银肯定需要加密的，假设工商银行和用户A、用户B分别约定了两个密码，用户A用密码一加密后把数据传工商银行，用户B用密码二加密后把数据传给工商银行，银行使用相应的密码进行解密，如果此时来了用户C，银行又得跟用户C商量好一个密码，如果有一万个用户呢？这时，使用非对称加密就能解决问题：工商银行把公钥放在网站上，所有人都使用这个公钥进行加密，工商银行使用私钥进行解密。

密钥类型：对称加密的密码称为密钥，非对称加密的有公钥和私钥

Key：SecretKey，PublicKey，PrivateKey

## 对称加密示例

对称加密也叫密钥加密，英文中对应的密钥类是：SecretKey

**public** **class** Demo {

**public** **static** **void** main(String[] args) **throws** Exception {

// createKey();

// encript();

*decript*();

}

/\*\* 创建密钥并保存起来，方便下次使用 \*/

**private** **static** **void** createKey() **throws** Exception {

Key key = KeyGenerator.*getInstance*("DES").generateKey();// 创建密钥用于DES算法的密钥

ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("EvenDai.key"));

oos.writeObject(key);

oos.close();

}

/\*\* 读取密钥 \*/

**private** **static** Key readKey() **throws** Exception {

ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("EvenDai.key"));

Key key = (Key) ois.readObject();

ois.close();

**return** key;

}

/\*\* 加密 \*/

**private** **static** **void** encript() **throws** Exception {

Cipher cipher = Cipher.*getInstance*("DES"); // 创建DES算法的加密对象

cipher.init(Cipher.***ENCRYPT\_MODE***, *readKey*()); // 初始化为加密模式

String data = "好好学习，天天向上！"; // 要加密的数据

**byte**[] encriptBytes = cipher.doFinal(data.getBytes()); // 进行加密

System.***out***.println(**new** String(encriptBytes)); // 显示加密后的数据

// 保存加密后的数据

FileOutputStream fos = **new** FileOutputStream("encript.data");

fos.write(encriptBytes);

fos.close();

}

**private** **static** **void** decript() **throws** Exception {

Cipher cipher = Cipher.*getInstance*("DES"); // 创建DES算法的加密对象

cipher.init(Cipher.***DECRYPT\_MODE***, *readKey*()); // 初始化为加密模式

// 读取加密的数据

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

FileInputStream fis = **new** FileInputStream("encript.data");

**byte**[] buf = **new** **byte**[1024];

**int** len;

**while**((len = fis.read(buf)) != -1) {

baos.write(buf, 0, len);

}

fis.close();

**byte**[] decriptBytes = cipher.doFinal(baos.toByteArray());// 进行解密

System.***out***.println(**new** String(decriptBytes)); // 显示解密后的数据

}

}

上面代码生产的Key，就相当于是银行的U盾 ，即U盘里的证书就是Key。

还有另一种加密的方式是使用密码，如银行卡取款的密码，这时不能使用DES算法了，要用另一种叫“**PBEWithMD5AndDES**”的算法，其中的PBE代表“password-based encryption（基于密码的加密）”，代码和上面基本相同，需要改变的地方只有两处如下：

1、Key的创建：

注：这里的密码要求必须是8位长度，否则化抛异常

**private** **static** Key createKey(String password) **throws** Exception {

KeySpec keySpec = **new** PBEKeySpec(password.toCharArray());

SecretKey key = SecretKeyFactory.*getInstance*("PBEWithMD5AndDES").generateSecret(keySpec);

**return** key; // 这时的key没有必要保存了，我们只要记得自己设置的password字符串密码即可。

}

2、Cipher类的创建：

上面的PBEParameterSpec就是传说中的加盐，解密的时候也得加这个参数。

PBEParameterSpec parameterSpec = **new** PBEParameterSpec(**new** **byte**[]{1,2,3,4,5,6,7,8},1000);

Cipher cipher = Cipher.*getInstance*("PBEWithMD5AndDES"); // 创建DES算法的加密对象

cipher.init(Cipher.***ENCRYPT\_MODE***, *createKey*("12345678"), parameterSpec); // 初始化为加密模式

## 非对称加密

一般都是公钥加密，私钥解密。代码与对称加密的方式基本相同，不同的是算法改成RSA，密钥的生成使用KeyPairGenerator来生成公钥和私钥，加密用公钥，解密就用私钥。

**public** **class** Demo {

**public** **static** **void** main(String[] args) **throws** Exception {

// createKey();

*encript*();

*decript*();

}

/\*\* 创建密钥并保存起来，方便下次使用 \*/

**private** **static** **void** createKey() **throws** Exception {

KeyPairGenerator generator = KeyPairGenerator.*getInstance*("RSA");

KeyPair keyPair = generator.generateKeyPair();

PublicKey publicKey = keyPair.getPublic();

PrivateKey privateKey = keyPair.getPrivate();

*saveKey*(publicKey, "publicKey.key");

*saveKey*(privateKey, "privateKey.key");

}

**private** **static** **void** saveKey(Key key, String keyFileName) **throws** Exception {

ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream(keyFileName));

oos.writeObject(key);

oos.close();

}

/\*\* 读取密钥 \*/

**private** **static** Key readKey(String keyFileName) **throws** Exception {

ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream(keyFileName));

Key key = (Key) ois.readObject();

ois.close();

**return** key;

}

/\*\* 加密 \*/

**private** **static** **void** encript() **throws** Exception {

Cipher cipher = Cipher.*getInstance*("RSA"); // 创建DES算法的加密对象

cipher.init(Cipher.***ENCRYPT\_MODE***, *readKey*("publicKey.key")); // 初始化为加密模式

String data = "好好学习，天天向上！"; // 要加密的数据

**byte**[] encriptBytes = cipher.doFinal(data.getBytes()); // 进行加密

System.***out***.println(**new** String(encriptBytes)); // 显示加密后的数据

// 保存加密后的数据

FileOutputStream fos = **new** FileOutputStream("encript.data");

fos.write(encriptBytes);

fos.close();

}

**private** **static** **void** decript() **throws** Exception {

Cipher cipher = Cipher.*getInstance*("RSA"); // 创建DES算法的加密对象

cipher.init(Cipher.***DECRYPT\_MODE***, *readKey*("privateKey.key"));// 初始化为加密模式

// 读取加密的数据

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

FileInputStream fis = **new** FileInputStream("encript.data");

**byte**[] buf = **new** **byte**[1024];

**int** len;

**while**((len = fis.read(buf)) != -1) {

baos.write(buf, 0, len);

}

fis.close();

**byte**[] decriptBytes = cipher.doFinal(baos.toByteArray());// 进行解密

System.***out***.println(**new** String(decriptBytes)); // 显示解密后的数据

}

}

注：cipher.update方法在RSA加密中不起作用，RSA加密只能使用cihper.doFinal(byte[] data)方法

CipherInputStream和CipherOutputStream是用于在读或写的时候就解密或者解密，就看会传入的chipher对象的初始化模式，如果设置为解密，则CipherInputStream用于把加密的文件读进来，读进来就解密了，而CipherOutputStream用于把加密的数据写到文件，写的时候就解密了，也就是说这两个流这时都是用在解密：具体例子如下：

**private** **static** **void** decript() **throws** Exception {

Cipher cipher = Cipher.*getInstance*("RSA"); // 创建DES算法的加密对象

cipher.init(Cipher.***DECRYPT\_MODE***, *readKey*("privateKey.key"));// 初始化为加密模式

FileInputStream fis = **new** FileInputStream("encript.data");

CipherInputStream cis = **new** CipherInputStream(fis, cipher); // 从这个流中读取的内容就是解密的

// 读取加密的数据

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

**byte**[] buf = **new** **byte**[1024];

**int** len;

**while**((len = cis.read(buf)) != -1) {

baos.write(buf, 0, len);

}

cis.close();

fis.close();

System.***out***.println(**new** String(baos.toByteArray()));

FileInputStream fis2 = **new** FileInputStream("encript.data");

FileOutputStream fos2 = **new** FileOutputStream("unencript.data");

CipherOutputStream cos = **new** CipherOutputStream(fos2, cipher); // 用这个流写出去的内容就是解密的

**while**((len = fis2.read(buf)) != -1) {

cos.write(buf, 0, len);

}

cos.close();

fos2.close();

fis2.close();

}

# 数字摘要

* **消息摘要是一种算法：无论原始数据多长，消息摘要的结果都是固定长度的；原始数据任意bit位的变化，都会导致消息摘要的结果有很大的不同，且根据结果推算出原始数据的概率极低。消息摘要可以看作原始数据的指纹，指纹不同则原始数据不同。**

比如MD5，给MD5传入一堆二进制数据，这些数据无论多少（如1K、1M、1G），通过Md5算法对这些二进制进行计算，计算的结果长度是不固定不变，都是128位，即16个字节。只要传进来的数据有一点点变化，则算出来的结果就大相径庭。既然结果总是128位，能表示的结果组合是有限的，而数据是无限的，所以不同数据产生的MD5结果还是有可能相同，但是这是很难碰得到相同的情况的，可以略等于不会有相同的，比如人的指纹是不可能有一样的，如果哪一天出现了一个罪犯，他的指纹和你一样，然后你被抓了，警察认为你就是罪犯把你枪毕了也不怨枉，你太幸运，活该！

SHA产生的数字摘要是20个字节，所以相比MD5要更安全。

MD5不是加密，因为通过MD5结果是不能恢复出原始数据的。

我们在注册用户时，不需要把密码发给服务器端，服务器端也不希望保存明文密码，所以我们只需要把md5后的密码发给服务器端保存即可。

## MD5产生数字摘要示例：

**public** **class** Demo {

**public** **static** **void** main(String[] args) **throws** Exception {

**byte**[] bytes = "zxx".getBytes();

String md5 = *md5*(bytes);

System.***out***.println("md5的16进制字符串：" + md5 + ", 长度：" + md5.length());

}

**private** **static** String md5(**byte**[] bytes) **throws** Exception {

MessageDigest md = MessageDigest.*getInstance*("MD5");

**byte**[] md5Bytes = md.digest(bytes); // 产生的md5数字摘要长度是16个字节

System.***out***.println("md5字节数组长度为：" + md5Bytes.length);

// 1个字节是8个比特位，如果把每个字节用10进制表示，则每个字节范围是-128 ~ 127，假设每个字节都是最大值，则会有16个127，

// 把16个127连接起来显示会很长，所以不使用10进制，使用16进制，4个位正好可以表示0 ~ 15，所以 1个字节会使用两个16进制数来表示，

// md5长度是16个字节，转换为16进制显示则需要32个16进制数

String hexMd5 = *byteToHex*(md5Bytes);

**return** hexMd5;

}

/\*\* 把字节转换为16进制字符串 \*/

**private** **static** String byteToHex(**byte**[] bytes) {

StringBuilder sb = **new** StringBuilder();

**for** (**byte** b: bytes) {

// 我们希望1个byte用两个16进制数表示，通过位运算取高4位和低4位分别转换为16进制

// byte hight4Bit = (byte) (b >>> 4); 这样取高4位是不对的，原因如下：

// 位运算会提升为10进制再位移，如果这个byte是负数，则提升为10进制时，高24位全是11，

// 无符号右移4位，则高4位变成了0，但是高24位中还有20位还是1,而我们希望32个二进位上只保留后8位中的前面4位

**byte** hight4Bit = (**byte**) (b >> 4 & 0xF); // 把高4位右移，低4位就被移出去了，&F就是只保留最低的4个二进制位，其它全部变成0

**byte** low4Bit = (**byte**) (b & 0x0F); // 注：&0F跟&F没有任何区别，推荐使用&F。因为&0F很容易让人认为是把4位与0相与，把4位与1相与，其实是错的，是低4位与1相与，剩余的位与0相与

// 现在的hight4Bit和low4Bit都是正数，且值最大是15，也就是16进制的F

// 我们不知道ASCCI码表中字符对应的十进制值是多少，但是我们可以直接使用这些字符进行运算，实际上是用这些字符对应的十进制值进制运算

// 如果4个位的值是10，我们减去10再加上a，则结果还是字符a对应的十进制值

// 如果4个位的值是11，我们减去10再加上a，则结果肯定是a的下一个字母b对应的十进制值

// 如果4个位的值如果不大于10，则加上字符零对应的十进制

// 如果4个位的值是0，加上字符零，结果还是这个字符零对应的十进制

// 如果4个位的值是1，加上字符零，结果肯定是这个字符零的下一个字符对应的十进制值

// 为什么 > 9就去减10加a，因为我们希望大于9的值用a ~ f表示

// 为什么 <= 9 就加'0'，因为我们希望小于或等于9的值用0 ~ 9表示

**char** hight4BitHex = (**char**) (hight4Bit > 9 ? hight4Bit - 10 + 'a' : hight4Bit + '0');

**char** low4BitHex = (**char**) (low4Bit > 9 ? low4Bit - 10 + 'a' : low4Bit + '0');

sb.append(hight4BitHex).append(low4BitHex);

}

**return** sb.toString();

}

}

注：MessageDigest有update(byte[])方法，跟加密时用到的类的使用相同，就是可以先不停调用update方法不停地添加字节，添加够了再调用digest()无参方法生成数字摘要。

md5生成的结果也称为I 哈希值

下面是使用了Integer的方法将十进制转换为16进制字符，代码更简单：

/\*\* 把字节转换为16进制字符串 \*/

**private** **static** String byteToHex(**byte**[] bytes) {

StringBuilder sb = **new** StringBuilder();

**for** (**int** b: bytes) { // 这里把byte提升成了int，如果是负数，int中多余的位会补-1，所以需要把除了低8位，高24位都要变成0

**int** hight4Bit = b & 0xFF; // 保留下低8位，把高24位变成0

String hexString = Integer.*toHexString*(hight4Bit);

//如果一个byte只用到了低4位，则只需要一个16进制数值即可表示，但我们希望用两个，所以补0

**if** (hexString.length() == 1) {

sb.append("0");

}

sb.append(hexString);

}

**return** sb.toString();

}

反向操作，把16进制转换回二进制

/\*\* 把16进制的字符串转换回原始字节码 \*/

**private** **static** **byte**[] hex2byte(String str) {

**int** size = str.length() / 2; // 每两个字符转换为一个字节

**byte**[] bytes = **new** **byte**[size];

**for** (**int** i = 0; i < size; ++i) {

String \_2bitHex = str.substring(2 \* i, 2 \* i + 2); // 获取两位16进制

bytes[i] = (**byte**) (Integer.parseInt(\_2bitHex, 16) & 0xFF);

}

**return** bytes;

}

## MD5生成文件的数字摘要

我们看到MD5是接收字节的，所以不管什么数据，都是字节，都可以进行MD5，只要把文件读取成字节即可。但是有另外两个更便捷的类DigestInputStream、DigestOutputStream，它们的使用跟加密时类型的流的使用也差不多，这两个类是用流读数据或写数据，我们的MD5需要完整的字节码，所以这两个流需要获取到完整的字节吗，它们要包装基本的IO流，等到读或者写完成的时候，再调用digest方法获取md5的哈希值。不论是读还是写，都可以获取哈希值。读，用于获取已存在的文件的哈希值，写，用于在生成一个文件的时候就获取哈希值。示例如下：

**public** **class** Demo {

**public** **static** **void** main(String[] args) **throws** Exception {

String md5 = *md5*("privateKey.key");

System.***out***.println("md5的16进制字符串：" + md5 + ", 长度：" + md5.length());

}

**private** **static** String md5(String filePath) **throws** Exception {

MessageDigest md = MessageDigest.*getInstance*("MD5");

FileInputStream fis = **new** FileInputStream(filePath);

DigestInputStream dis = **new** DigestInputStream(fis, md);

**byte**[] buf = **new** **byte**[1024];

**while** (dis.read(buf) != -1);

dis.close();

fis.close();

**byte**[] md5Bytes = md.digest();

String hexMd5 = *byteToHex*(md5Bytes);

**return** hexMd5;

}

**private** **static** String byteToHex(**byte**[] bytes) {

StringBuilder sb = **new** StringBuilder();

**for** (**int** b: bytes) {

**int** hight4Bit = b & 0xFF;

String hexString = Integer.*toHexString*(hight4Bit);

**if** (hexString.length() == 1) {

sb.append("0");

}

sb.append(hexString);

}

**return** sb.toString();

}

}

## 基于MAC（消息验证码）的数字摘要

MAC，Message Authentication Code ，密钥被用作消息摘要生成过程的一部分。

防止内容和摘要同时被篡改，在一定程度上起到了验证发送者身份。

步骤： KeyGenerator.getInstance🡪KeyGeneraotr.generateKey 🡪 Mac.getInstance 🡪 Mac.init(secretyKey) 🡪Mac.update🡪Mac.doFinal

简单理解：

就是把md5当成签名，我们把一些数据传给服务器，可以给这些数据生成md5，但是这样并没有什么用，因为别人修改了数据之后可以重新生成md5，解决办法就是我们可以在生成md5中加入一些其它的字符（密码），这样别人修改了数据虽然可以再生成md5，但是它没有密码，所以我们在验证的时候能知道数据被修改了。

示例代码如下：

**private** **static** String macDigest(**byte**[] bytes) **throws** Exception{

String password = "123456f";

PBEKeySpec keySpec = **new** PBEKeySpec(password.toCharArray());

SecretKeyFactory keyFactory = SecretKeyFactory.*getInstance*("PBEWithMD5AndDES");

SecretKey key = keyFactory.generateSecret(keySpec);

Mac mac = Mac.*getInstance*("HmacMD5");

mac.init(key);

**byte**[] md5Bytes = mac.doFinal(bytes);

String hexMd5 = *byteToHex*(md5Bytes);

**return** hexMd5;

}

## SHA 1生产数字摘要

使用SHA1算法产生的数字摘要长度是20个字节。

与MD5 的使用一模一样，只需要把MessageDigest.*getInstance*("MD5")中的参数改成："SHA-1"

如果是要生成基于MAC的数字摘要，则把Mac mac = Mac.*getInstance*("HmacMD5");中的参数改成："HmacSHA1"

# 数字签名

数字签名的基础是公钥和私钥的非对称加密。

对于加密来说，加密一般使用公钥加密，私钥解密，如果使用私钥加密，则这个加密就没有意义了，因为很多人都有公钥，都可以解密。

对于数字签名来说，则使用私钥加密是有用的，作用就是别人可以公钥来认证你的身份，因为公钥能解密，说明你就是拥有私钥的那个人，如果你的私钥被别人偷了，那也没办法。就像银行的公章，被偷来盖章转走了钱了，银行的人说不是他们干的，那也没办法，确实是你们的公章，承让你们给别人偷走呢。所以私钥加密公钥解密可以用来认证一个人的身份。

预防数据被修改那使用私钥加密就行了，为什么还需要签名呢？因为非对称加密是比较耗时的，SHA签名的原来是这样的：使用SHA生成数字摘要(这个速度是比较快的)，防止数据被人修改，但是任何人都可以使用SHA生成数字摘要，怎么知道这个数字摘要就是你生成的呢？我们使用私钥对这个签名进行加密，虽然说加密能被所有拥有公钥的人打开，但是无所谓了，我们的目的是让有公钥的人能验证我的身份即可，他能打开，说明这个数字摘要就是拥有私钥的人生成的，这样身份得以确定。

SHA数字签名示例代码：

**public** **class** Demo {

**public** **static** **void** main(String[] args) **throws** Exception {

String jieTiao = "我是小明，向小红借300万，下个月1号前还";

**byte**[] sign = *sign*(jieTiao);

**boolean** result = *verify*(jieTiao, sign);

System.***out***.println(result ? "验证通过" : "验证失败");

}

/\*\* 读取密钥 \*/

**private** **static** Key readKey(String keyFileName) **throws** Exception {

ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream(keyFileName));

Key key = (Key) ois.readObject();

ois.close();

**return** key;

}

**public** **static** **byte**[] sign(String msg) **throws** Exception {

// 参数"SHA1withRSA"前面的SHA1，代表数字摘要是使用SHA1算法进行计算的，后面的RSA代表使用RSA对这个签名进行加密

// 所以参数也可以使用"MD5withRSA"（代表数字摘要是使用md5进行计算的的），当然查看文档还有更多参数可用

Signature signature = Signature.*getInstance*("SHA1withRSA");

signature.initSign((PrivateKey) *readKey*("privateKey.key"));

signature.update(msg.getBytes());

**byte**[] sign = signature.sign();

**return** sign;

}

**private** **static** **boolean** verify(String jieTiao, **byte**[] sign) **throws** Exception {

Signature signature = Signature.*getInstance*("SHA1withRSA");

signature.initVerify((PublicKey) *readKey*("publicKey.key"));

signature.update(jieTiao.getBytes());

**boolean** result = signature.verify(sign);

**return** result;

}

}

现在还有一个问题：就是安全的问题，张三生产了一对密钥，把公钥给了李四，然后使用私钥对一张借条进行了签名，签名给了李四，李四一验证通过，然后借了300万给张三。问题来了，到还款期的时候，张三说李四的公钥不是它给的，这怎么确定是不是张三给的呀，没法确定，因为张三也可以说它没有对应的私钥。所以为了安全，生产密钥的人需要把公钥拿到一个权威的机构进行认证，认证过的公钥就是安全的，认证通过会给你发一个证书，证书是一个文件形式，里面装着你的公钥，并且有权威机构的签名，所以可以把这个证书拿到这个权威机构的网站进行验证，验证通过则是安全的证书，则证书里的公钥也是安全的公钥

# 数字证书

数字证书原理：你把你的公钥拿到权威机构进行认证，权威机会会要求你提供身份证啊等各种信息，验证通过了，它就会把你的公钥进行签名，然后把你的公钥、签名、认证机构信息等封装成一个文件，这个文件就是证书。所以这个证书里包含了我们的公钥，还有机构信息和机构对这个公钥的签名。我们怎么确认这个证书是可信任的呢？我们可以根据证书上的机构信息找到权威机构网站，权威机构网站上提供了权威机构的公钥，使用这个公钥可以验证我们的证书里面的签名，如果能验证通过，则这个证书就是可信任的。

在国外有一些权威机构，全世界的人都信任它们，在任何的电脑在出厂时就已经信任了一些全球知名机构的证书，查看IE上的证书，如下：

 

“受信任的根证书颁发机构”，意思是这些机构都是直接颁发证书的机构。

假设中国的公安部拿着公钥去一个机构进行了认证得到了证书，那我们可以找公安部给我们的公钥进行认证，公安部认证后也给我们一个证书，这个证书是使用公安部的私钥进行签名的。电脑会不会信任我们的证书呢？会，因为电脑看到我们的证书，它能知道是公安部签名的，但是然后电脑能知道公安部是权威机构签名的，所以可以信任。 像这样类似公安部颁发的证书机构就称为“中间证书颁发机构”，如下图：

