Java 安全专题

Java 安全专题

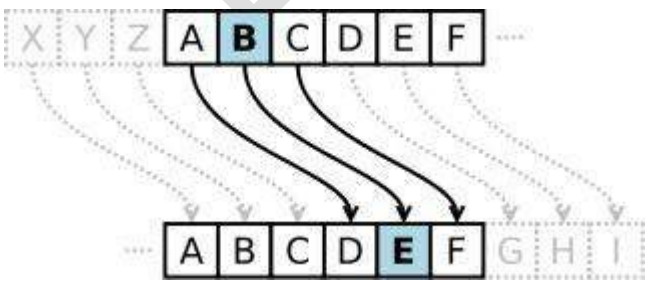
- 一些基本的安全知识
- 对称加密
- 非对称加密
- 数字签名
- 数字证书
- Keytool 工具的使用
- 基于 https 的网络请求

1.1 手写简单的加密（凯撒密码）

1.1.1 介绍

凯撒密码作为一种最为古老的对称加密体制，在古罗马的时候都已经很流行，他的基本思想是：通过把字母移动一定的位数来实现加密和解密。明文中的所有字母都在字母表上向后（或向前）按照一个固定数目进行偏移后被替换成密文。例如，当偏移量是 3 的时候，所有的字母 A 将被替换成 D，B 变成 E，由此可见，位数就是凯撒密码加密和解密的密钥。

例如：字符串”ABC”的每个字符都右移 3 位则变成”DEF”，解密的时候”DEF”的每个字符左移 3 位即能还原，如下图所示：



1.1.2 准备知识

```
1.  //字符转换成 ASCII 码数值
2.  char charA = 'a';
3.  int intA = charA; //char 强转为 int 即得到对应的 ASCII 码值，’a’的值为 97
4.
5.  //ASCII 码值转成 char
6.  int intA = 97;//97 对应的 ASCII 码’a’
7.  char charA = (char) intA; //int 值强转为 char 即得到对应的 ASCII 字符，即'a'
```

ASCII 字符代码表 一

高四位 / 低四位		ASCII非打印控制字符										ASCII 打印字符												
		0000					0001					0010		0011		0100		0101		0110		0111		
		0					1					2		3		4		5		6		7		
		+进制	字符	ctrl	代码	字符解释	+进制	字符	ctrl	代码	字符解释	+进制	字符	+进制	字符	+进制	字符	+进制	字符	+进制	字符	+进制	字符	ctrl
0000	0	0	BLANK NULL	^@	NUL	空	16	►	^P	DLE	数据链路转意	32		48	0	64	@	80	P	96	`	112	p	
0001	1	1	☺	^A	SOH	头标开始	17	◄	^Q	DC1	设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q	
0010	2	2	☻	^B	STX	正文开始	18	↕	^R	DC2	设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r	
0011	3	3	♥	^C	ETX	正文结束	19	‼	^S	DC3	设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s	
0100	4	4	♦	^D	EOT	传输结束	20	¶	^T	DC4	设备控制 4	36	$	52	4	68	D	84	T	100	d	116	t	
0101	5	5	♣	^E	ENQ	查询	21	§	^U	NAK	反确认	37	%	53	5	69	E	85	U	101	e	117	u	
0110	6	6	♠	^F	ACK	确认	22	■	^V	SYN	同步空闲	38	&	54	6	70	F	86	V	102	f	118	v	
0111	7	7	•	^G	BEL	震铃	23	↨	^W	ETB	传输块结束	39	'	55	7	71	G	87	W	103	g	119	w	
1000	8	8	◘	^H	BS	退格	24	↑	^X	CAN	取消	40	(	56	8	72	H	88	X	104	h	120	x	
1001	9	9	○	^I	TAB	水平制表符	25	↓	^Y	EM	媒体结束	41	)	57	9	73	I	89	Y	105	i	121	y	
1010	A	10	◙	^J	LF	换行/新行	26	→	^Z	SUB	替换	42	*	58	:	74	J	90	Z	106	j	122	z	
1011	B	11	♂	^K	VT	竖直制表符	27	←	^[	ESC	转意	43	+	59	;	75	K	91	[	107	k	123	{	
1100	C	12	♀	^L	FF	换页/新页	28	∟	^\	FS	文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|	
1101	D	13	♪	^M	CR	回车	29	↔	^]	GS	组分隔符	45	-	61	=	77	M	93	]	109	m	125	}	
1110	E	14	♫	^N	SO	移出	30	▲	^6	RS	记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~	
1111	F	15	☼	^O	SI	移入	31	▼	^-	US	单元分隔符	47	/	63	?	79	O	95	_	111	o	127	Δ	Back space

注：表中的ASCII字符可以用:ALT + "小键盘上的数字键" 输入

1.1.3 凯撒密码的简单代码实现

```
1.      /**
2.       * 加密
3.       * @param input 数据源（需要加密的数据）
4.       * @param key 秘钥，即偏移量
5.       * @return 返回加密后的数据
6.       */
7.      public static String encrypt(String input, int key) {
8.          //得到字符串里的每一个字符
9.          char[] array = input.toCharArray();
```

```
        for (int i = 0; i < array.length; ++i) {
            //字符转换成 ASCII 码值
            int ascii = array[i];
            //字符偏移，例如 a->b
            ascii = ascii + key;
            //ASCII 码值转换为 char
            char newChar = (char) ascii;
            //替换原有字符
            array[i] = newChar;

            //以上 4 行代码可以简写为一行
            //array[i] = (char) (array[i] + key);
        }

        //字符数组转换成 String
        return new String(array);
    }

    /**
     * 解密
     * @param input 数据源（被加密后的数据）
     * @param key 秘钥，即偏移量
     * @return 返回解密后的数据
     */
    public static String decrypt(String input, int key) {
        //得到字符串里的每一个字符
        char[] array = input.toCharArray();
        for (int i = 0; i < array.length; ++i) {
            //字符转换成 ASCII 码值
            int ascii = array[i];
            //恢复字符偏移，例如 b->a
            ascii = ascii - key;
            //ASCII 码值转换为 char
            char newChar = (char) ascii;
            //替换原有字符
            array[i] = newChar;

            //以上 4 行代码可以简写为一行
            //array[i] = (char) (array[i] - key);
        }

        //字符数组转换成 String
        return new String(array);
    }
```

代码输出结果：

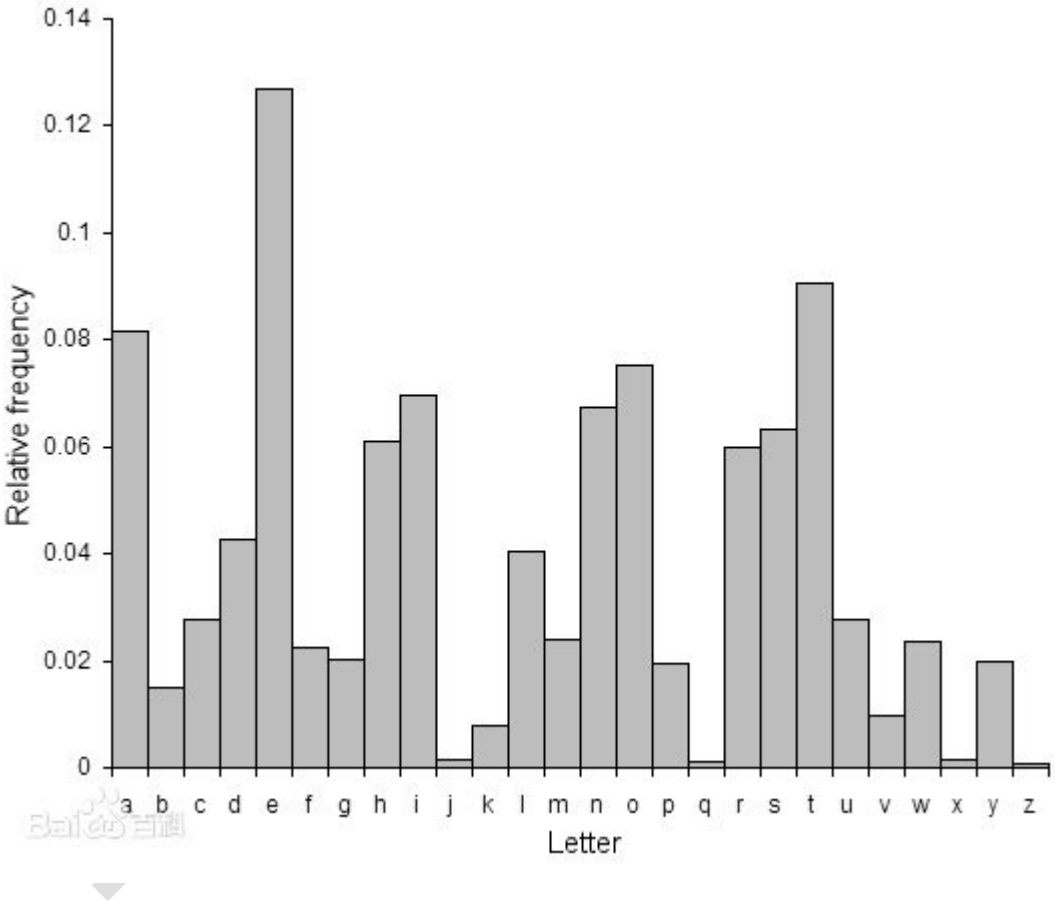
```
原文：Hei Ma
加密：Khl#Pd
解密：Hei Ma
```

1.1.4 破解凯撒密码：频率分析法

凯撒密码加密强度太低，只需要用频度分析法即可破解。

在任何一种书面语言中，不同的字母或字母组合出现的频率各不相同。而且，对于以这种语言书写的任意一段文本，都具有大致相同的特征字母分布。比如，在英语中，字母 E 出现的频率很高，而 X 则出现得较少。

英语文本中典型的字母分布情况如下图所示：



1.1.5 破解流程

1，统计密文里出现次数最多的字符，例如出现次数最多的字符是是’h’。

2，计算字符’h’到’e’的偏移量，值为 3，则表示原文偏移了 3 个位置。

3，将密文所有字符恢复偏移 3 个位置。

注意点：统计密文里出现次数最多的字符时，需多统计几个备选，因为最多的可能是空格或者其他字符，例如下图出现次数最多的字符’#’是空格加密后的字符，’h’才是’e’偏移后的值。

```
字符'#'出现989次
字符'h'出现478次
字符'd'出现344次
字符'w'出现327次
```

解密时要多几次尝试，因为不一定出现次数最多的字符就是我们想要的目标字符，如下图，第二次解密的结果才是正确的。

```
字符'#'出现989次
猜测key = -66，解密生成第1个备选文件

字符'h'出现478次
猜测key = 3，解密生成第2个备选文件

字符'd'出现344次
猜测key = -1，解密生成第3个备选文件

字符'w'出现327次
猜测key = 18，解密生成第4个备选文件
```

代码实现请查看课程源码里的 demo

1.2 对称加密

1.2.1 介绍

加密和解密都使用同一把秘钥，这种加密方法称为对称加密，也称为单密钥加密。
简单理解为：加密解密都是同一把钥匙

1.1 里学到的凯撒密码就属于对称加密，他的字符偏移量即为秘钥。

1.2.2 对称加密常用算法

AES、DES、3DES、TDEA、Blowfish、RC2、RC4、RC5、IDEA、SKIPJACK 等。

DES：全称为 Data Encryption Standard，即数据加密标准，是一种使用密钥加密的块算法，1976 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），随后在国际上广泛流传开来。

3DES：也叫 Triple DES，是三重数据加密算法（TDEA，Triple Data Encryption Algorithm）块密码的通称。它相当于是对每个数据块应用三次 DES 加密算法。由于计算机运算能力的增强，原版 DES 密码的密钥长度变得容易被暴力破解；3DES 即是设计用来提供一种相对简单的方法，即通过增加 DES 的密钥长度来避免类似的攻击，而不是设计一种全新的块密码算法。

AES: 高级加密标准（英语：Advanced Encryption Standard，缩写：AES），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。经过五年的甄选流程，高级加密标准由美国国家标准与技术研究院（NIST）于 2001 年 11 月 26 日发布于 FIPS PUB 197，并在 2002 年 5 月 26 日成为有效的标准。2006 年，高级加密标准已然成为对称密钥加密中最流行的算法之一。

1.2.3 DES 算法简介

DES 加密原理（对比特位进行操作，交换位置，异或等等，无需详细了解）：

1.2.4 准备知识

Bit 是计算机最小的传输单位。以 0 或 1 来表示比特位的值
例如数字 3 对应的二进制数据为：00000011

代码示例：

```
1.          int i = 97;
2.          String bit = Integer.toBinaryString(i);
3.          //输出：97 对应的二进制数据为： 1100001
4.          System.out.println(i + "对应的二进制数据为： " + bit);
```

Byte 与 Bit 区别

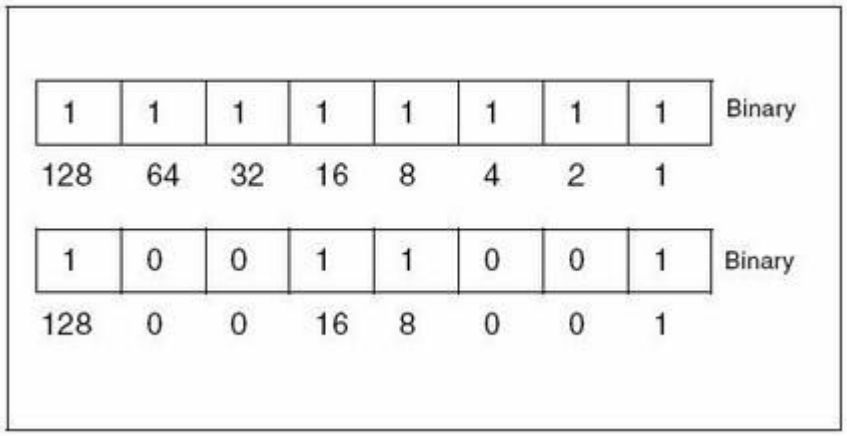
数据存储是以“字节”（Byte）为单位，数据传输是以大多是以“位”（bit，又名“比特”）为单位，一个位就代表一个 0 或 1（即二进制），每 8 个位（bit，简写为 b）组成一个字节（Byte，简写为 B），是最小一级的信息单位。

Byte 的取值范围：

```
1.          //byte 的取值范围：-128 到 127
2.          System.out.println(Byte.MIN_VALUE + "到" + Byte.MAX_VALUE);
```

即 10000000 到 01111111 之间，一个字节占 8 个比特位

二进制转十进制图示：



任何字符串都可以转换为字节数组

```
1.          String data = "1234abcd";
2.          byte[] bytes = data.getBytes();//内容为：49 50 51 52 97 98 99 100
```

上面数据 49 50 51 52 97 98 99 100 对应的二进制数据（即比特位为）：

00110001
00110010
00110011
00110100
01100001
01100010
01100011
01100100

将他们间距调大一点，可看做一个矩阵：

0 0 1 1 0 0 0 1
0 0 1 1 0 0 1 0
0 0 1 1 0 0 1 1
0 0 1 1 0 1 0 0

0 1 1 0 0 0 0 1
0 1 1 0 0 0 1 0
0 1 1 0 0 0 1 1
0 1 1 0 0 1 0 0

之后可对他们进行各种操作，例如交换位置、分割、异或运算等，常见的加密方式就是这样操作比特位的，例如下图的 IP 置换以及 S-Box 操作都是常见加密的一些方式：

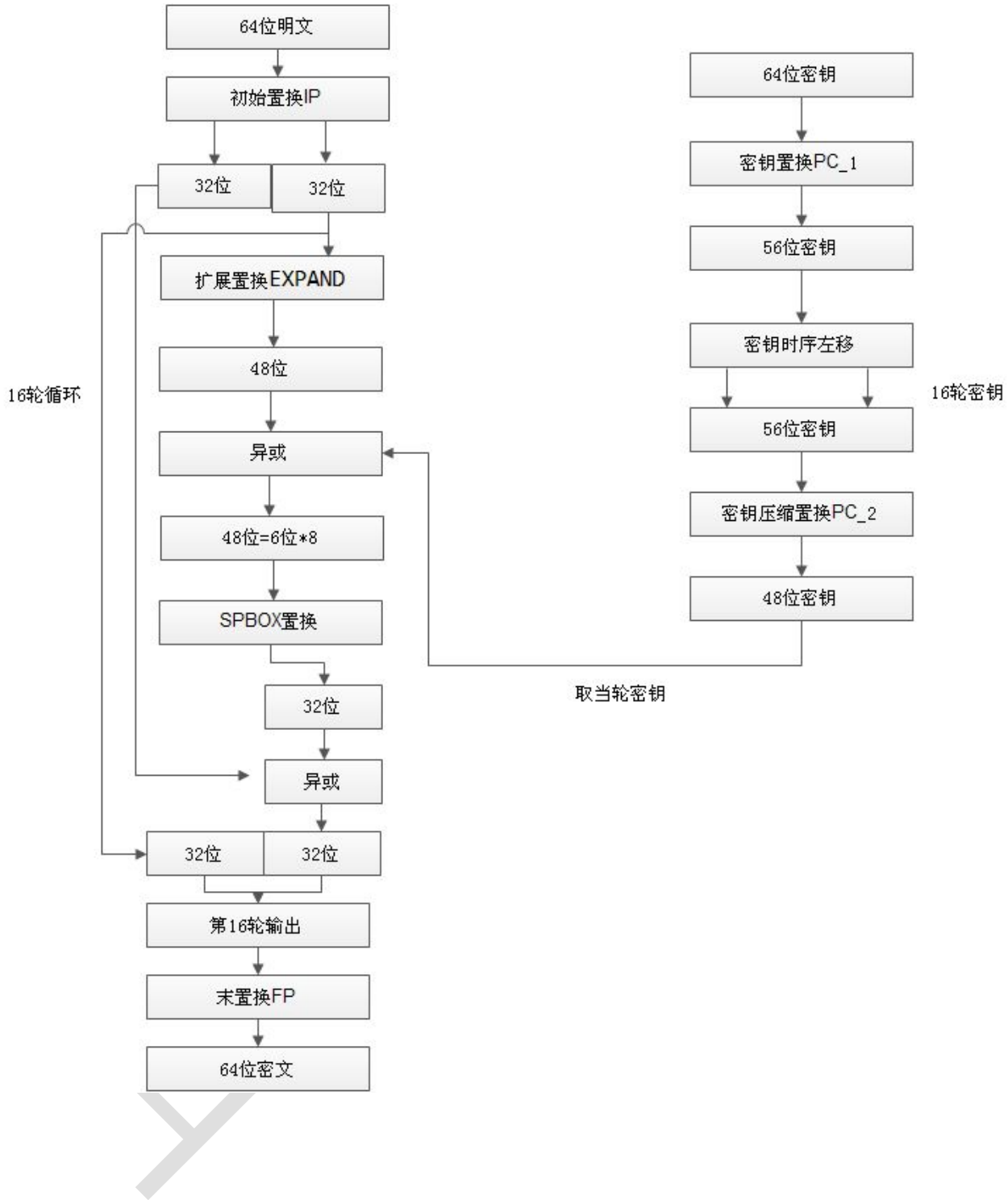
IP 置换：

IP								IP^{-1}							
58	50	42	34	26	18	10	2	40	8	48	16	56	24	64	32
60	52	44	36	28	20	12	4	39	7	47	15	55	23	63	31
62	54	46	38	30	22	14	6	38	6	46	14	54	22	62	30
64	56	48	40	32	24	16	8	37	5	45	13	53	21	61	29
57	49	41	33	25	17	9	1	36	4	44	12	52	20	60	28
59	51	43	35	27	19	11	3	35	3	43	11	51	19	59	27
61	53	45	37	29	21	13	5	34	2	42	10	50	18	58	26
63	55	47	39	31	23	15	7	33	1	41	9	49	17	57	25

S-BOX 置换：

–	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s_1	0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
s_2	0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
	1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
s_3	0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	2	13	6	4	9	8	15	3	0	11	1	2	12	5	13	14	7
	3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
s_4	0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
	1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
s_5	0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

DES 加密过程图解（流程很复杂，只需要知道内部是操作比特位即可）：



1.2.5 对称加密应用场景

1. 本地数据加密（例如加密 android 里 SharedPreferences 里面的某些敏感数据）
2. 网络传输：登录接口 post 请求参数加密{username=lisi,pwd=oJYa4i9VASRoxVLh75wPCg==}
3. 加密用户登录结果信息并序列化到本地磁盘(将 user 对象序列化到本地磁盘，下次登录时反序列化到内存里)
4. 网页交互数据加密（即后面学到的 Https）

1.2.6 DES 算法代码实现

```
1.      //1,得到 cipher 对象（可翻译为密码器或密码系统）
2.      Cipher cipher = Cipher.getInstance("DES");
3.      //2，创建秘钥
4.      SecretKey key = KeyGenerator.getInstance("DES").generateKey();
5.      //3，设置操作模式（加密/解密）
6.      cipher.init(Cipher.ENCRYPT_MODE, key);
7.      //4，执行操作
8.      byte[] result = cipher.doFinal("黑马".getBytes());
```

1.2.7 AES 算法代码实现

用法同上，只需把”DES”参数换成”AES”即可。

1.2.8 使用 Base64 编码加密后的结果

```
1.  byte[] result = cipher.doFinal("黑马".getBytes());
2.  System.out.println(new String(result));
```

输出：

```
U#◆◆◆◆.G◆◆◆VO◆7
```

加密后的结果是字节数组，这些被加密后的字节在码表（例如 UTF-8 码表）上找不到对应字符，会出现乱码，当乱码字符串再次转换为字节数组时，长度会变化，导致解密失败，所以转换后的数据是不安全的。

使用 Base64 对字节数组进行编码，任何字节都能映射成对应的 Base64 字符，之后能恢复到字节数组，利于加密后数据的保存于传输，所以转换是安全的。同样，字节数组转换成 16 进制字符串也是安全的。

密文转换成 Base64 编码后的输出结果：

```
VSO63Yqj1y5HrJXCVk+sNw==
```

密文转换成 16 进制编码后的输出结果：

```
5523badd8aa3d72e47ac95c2564fac37
```

Java 里没有直接提供 Base64 以及字节数组转 16 进制的 Api，开发中一般是自己手写或直接使用第三方提供的成熟稳定的工具类（例如 apache 的 commons-codec）。

Base64 字符映射表

Base64字符映射表

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v	(pad)	=
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y		

1.2.9 对称加密的具体应用方式

1，生成秘钥并保存到硬盘上，以后读取该秘钥进行加密解密操作，实际开发中用得比较少。

```
1.    //生成随机秘钥
2.    SecretKey secretKey = KeyGenerator.getInstance("AES").generateKey();
3.    //序列化秘钥到磁盘上
4.    FileOutputStream fos = new FileOutputStream(new File("heima.key"));
```

```
5.     ObjectOutputStream oos = new ObjectOutputStream(fos);
6.     oos.writeObject(secretKey);
7.
8.      //从磁盘里读取秘钥
9.      FileInputStream fis = new FileInputStream(new File("heima.key"));
10.     ObjectInputStream ois = new ObjectInputStream(fis);
11.     Key key = (Key) ois.readObject();
```

2，使用自定义秘钥（秘钥写在代码里）

```
1.      //创建密钥写法 1
2.     KeySpec keySpec = new DESKeySpec(key.getBytes());
3.      SecretKey secretKey = SecretKeyFactory.getInstance(ALGORITHM).
4.  generateSecret(keySpec);
5.
6.     //创建密钥写法 2
7.      //SecretKey secretKey = new SecretKeySpec(key.getBytes(), KEY_ALGORITHM);
8.
9.      Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
10.     cipher.init(Cipher.DECRYPT_MODE, secretKey);
11.     //得到 key 后，后续代码就是 Cipher 的写法，此处省略...
```

把秘钥写在代码里有一定风险，当别人反编译代码的时候，可能会看到秘钥，android 开发里建议用 JNI 把秘钥值写到 C 代码里，甚至拆分成几份，最后再组合成真正的秘钥

1.2.10 算法/工作模式/填充模式

初始化 cipher 对象时，参数可以直接传算法名：例如：
Cipher c = Cipher.getInstance("DES");

也可以指定更详细的参数，格式："algorithm/mode/padding" ，即"算法/工作模式/填充模式"
Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");

密码块工作模式

块密码工作模式(Block cipher mode of operation)，是对于按块处理密码的加密方式的一种扩充，不仅仅适用于 AES，包括 DES， RSA 等加密方法同样适用。

名称	英文	全名	方法	优点	缺点
ECB	Electronic codebook	电子密码本	每块独立加密	1.分块可以并	1.同样的原文得到相同的密

				行处理	文，容易被攻击
CBC	Cipher-block chaining	密码分组链接	每块加密依赖于前一块的密文	1.同样的原文得到不同的密文 2.原文微小改动影响后面全部密文	1.加密需要串行处理 2.误差传递
PCBC	Propagating cipher-block chaining	填充密码块链接	CBC 的扩种，较少使用	1.同样的原文得到不同的密文 2.互换两个邻接的密文块不会对后续块的解密造成影响	1.加密需要串行处理
CFB	Cipher feedback	密文反馈			
OFB	Output feedback	输出反馈模式	加密后密文与原文异或 XOR		1.能够对密文进行校验
CTR	Counter mode	计数器模式	增加一个序列函数对所有密文快做 XOR		

填充

填充(Padding)，是对需要按块处理的数据，当数据长度不符合块处理需求时，按照一定方法填充满块长的一种规则。

名称	方法	示例
Zero padding	最常见的方式，全填充 0x00	AA AA AA AA **00 00 00 00**
ANSI X.923	Zero 的改进，最后一个字节为填充字节个数	AA AA AA AA **00 00 00 04**
ISO 10126	随机填充	AA AA AA AA **81 A6 23 04**
PKCS7	ANSI X.923 的变体 填充 1 个字符就全 0x01 填充 2 个字符就全 0x02 不需要填充就增加一个块，填充块长度，块长为 8 就填充 0x08，块长为 16 就填充 0x10	AA AA AA AA AA AA AA **01** AA AA AA AA **04 04 04 04** AA AA AA AA AA AA AA AA**08 08 08 08 08 08 08 08**
ISO/IEC 7816-4	以 0x80 开始作为填充开始标记，后续全填充 0x00	AA AA AA AA AA AA AA **80** AA AA AA AA **80 00 00 00**

具体代码：

```
1       //秘钥算法
2       private static final String KEY_ALGORITHM = "DES";
3       //加密算法：algorithm/mode/padding  算法/工作模式/填充模式
4       private static final String CIPHER_ALGORITHM = "DES/ECB/PKCS5Padding";
5       //秘钥
6       private static final String KEY = "12345678";//DES 秘钥长度必须是 8 位或以上
7       //private static final String KEY = "1234567890123456";//AES 秘钥长度必须是 16 位
8
9      //初始化秘钥
10     SecretKey secretKey = new SecretKeySpec(KEY.getBytes(), KEY_ALGORITHM);
11      Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
```

```
12      //加密
13      cipher.init(Cipher.ENCRYPT_MODE, secretKey);
14      byte[] result = cipher.doFinal(input.getBytes());
```

注意：AES、DES 在 CBC 操作模式下需要 iv 参数

```
15      //AES、DES 在 CBC 操作模式下需要 iv 参数
16      IvParameterSpec iv = new IvParameterSpec(key.getBytes());
17
18      //加密
19      cipher.init(Cipher.ENCRYPT_MODE, secretKey, iv);
```

1.2.11 总结

DES 安全度在现代已经不够高，后来又出现的 3DES 算法强度提高了很多，但是其执行效率低下，AES 算法加密强度大，执行效率高，使用简单，实际开发中建议选择 AES 算法。

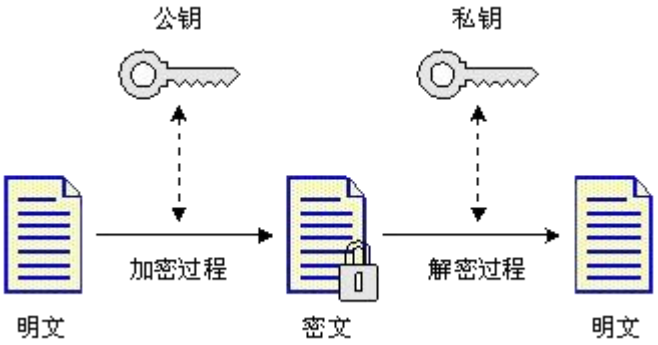
实际 android 开发中可以用对称加密（例如选择 AES 算法）来解决很多问题，例如：
1，做一个管理密码的 app，我们在不同的网站里使用不同账号密码，很难记住，想做个 app 统一管理，但是账号密码保存在手机里，一旦丢失了容易造成安全隐患，所以需要一种加密算法，将账号密码信息加密起来保管，这时候如果使用对称加密算法，将数据进行加密，秘钥我们自己记载心里，只需要记住一个密码。需要的时候可以还原信息。
2，android 里需要把一些敏感数据保存到 SharedPrefrence 里的时候，也可以使用对称加密，这样可以在需要的时候还原。
3，请求网络接口的时候，我们需要上传一些敏感数据，同样也可以使用对称加密，服务端使用同样的算法就可以解密。或者服务端需要给客户端传递数据，同样也可以先加密，然后客户端使用同样算法解密。

1.3 非对称加密

1.3.1 介绍

与对称加密算法不同，非对称加密算法需要两个密钥：公钥（publickey）和私钥（privatekey）。公钥与私钥是一对，如果用公钥对数据进行加密，只有用对应的私钥才能解密；如果用私钥对数据进行加密，那么只有用对应的公钥才能解密。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。

简单理解为：加密和解密是不同的钥匙



1.3.2 常见算法

RSA、Elgamal、背包算法、Rabin、D-H、ECC（椭圆曲线加密算法）等

1.3.3 RSA 算法原理

质因数、欧拉函数、模反元素

原理很复杂，只需要知道内部是基于分解质因数和取模操作即可

1.3.4 使用步骤

```
1.      //1，获取 cipher 对象
2.      Cipher cipher = Cipher.getInstance("RSA");
3.      //2，通过秘钥对生成器 KeyPairGenerator 生成公钥和私钥
4.      KeyPair keyPair = KeyPairGenerator.getInstance("RSA").generateKeyPair();
5.      //使用公钥进行加密，私钥进行解密（也可以反过来使用）
6.      PublicKey publicKey = keyPair.getPublic();
7.      PrivateKey privateKey = keyPair.getPrivate();
8.      //3,使用公钥初始化密码器
9.      cipher.init(Cipher.ENCRYPT_MODE, publicKey);
10.     //4，执行加密操作
11.     byte[] result = cipher.doFinal(content.getBytes());
12.     //使用私钥初始化密码器
13.     cipher.init(Cipher.DECRYPT_MODE, privateKey);
14.     //执行解密操作
15.     byte[] deResult = cipher.doFinal(result);
```

1.3.5 注意点

```
1.  //一次性加密数据的长度不能大于 117 字节
```

```
2.  private static final int ENCRYPT_BLOCK_MAX = 117;
3.  //一次性解密的数据长度不能大于 128 字节
4.  private static final int DECRYPT_BLOCK_MAX = 128;
```

分批操作

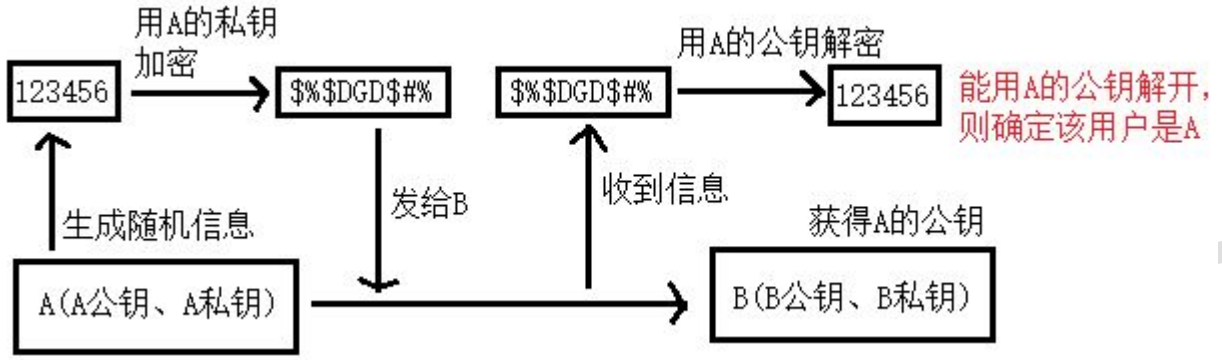
```
5.  /**
6.   * 分批操作
7.   *
8.   * @param content 需要处理的数据
9.   * @param cipher 密码器（根据 cipher 的不同，操作可能是加密或解密）
10.  * @param blockSize 每次操作的块大小，单位为字节
11.  * @return 返回处理完成后的结果
12.  * @throws Exception
13.  */
14. public static byte[] doFinalWithBatch(byte[] content, Cipher cipher, int blockSize) throws
    Exception {
15.     int offset = 0;//操作的起始偏移位置
16.     int len = content.length;//数据总长度
17.     byte[] tmp;//临时保存操作结果
18.     ByteArrayOutputStream baos = new ByteArrayOutputStream();
19.     //如果剩下数据
20.     while (len - offset > 0) {
21.         if (len - offset >= blockSize) {
22.             //剩下数据还大于等于一个 blockSize
23.             tmp = cipher.doFinal(content, offset, blockSize);
24.         }else {
25.             //剩下数据不足一个 blockSize
26.             tmp = cipher.doFinal(content, offset, len - offset);
27.         }
28.         //将临时结果保存到内存缓冲区里
29.         baos.write(tmp);
30.         offset = offset + blockSize;
31.     }
32.     baos.close();

33.     return baos.toByteArray();
34. }
```

1.3.6 非对称加密用途

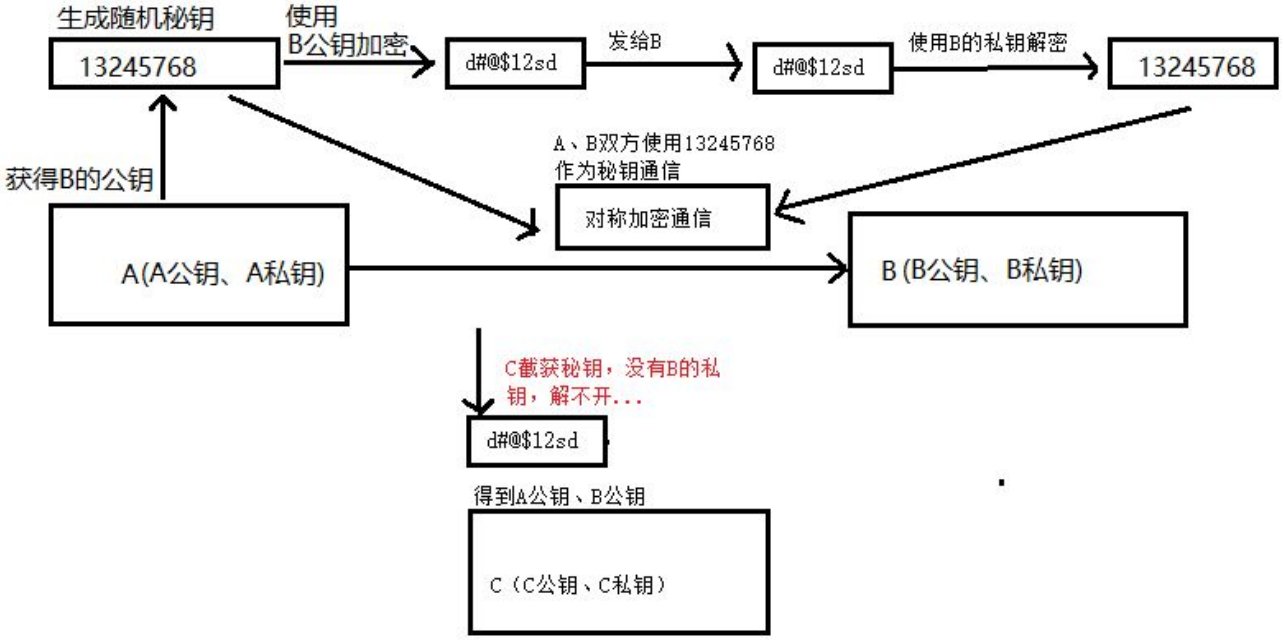
1，身份认证

一条加密信息若能用 A 的公钥能解开，则该信息一定是用 A 的私钥加密的，该能确定该用户是 A。



2，陌生人通信：A 和 B 两个人互不认识，A 把自己的公钥发给 B，B 也把自己的公钥发给 A，则双方可以通过对方的公钥加密信息通信。C 虽然也能得到 A、B 的公钥，但是他解不开密文。

3，秘钥交换： A 先得到 B 的公钥，然后 A 生成一个随机秘钥，例如 13245768，之后 A 用 B 的公钥加密该秘钥，得到加密后的秘钥，例如 d#@$12sd，之后将该密文发给 B，B 用自己的私钥解密得到 123456，之后双方使用 13245768 作为对称加密的秘钥通信。C 就算截获加密后的秘钥 d#@$12*asd，自己也解不开，这样 A、B 二人能通过对称加密进行通信。



1.3.7 总结

非对称加密一般不会单独拿来使用，他并不是为了取代对称加密而出现的，非对称加密速度比对称加密慢很多，极端情况下会慢 1000 倍，所以一般不会用来加密大量数据，通常我们经常会将对称加密和非对称加密两种技术联合起来使用，例如用非对称加密来给称加密里的秘钥进行加密（即秘钥交换）。

1.4 消息摘要（Message Digest）

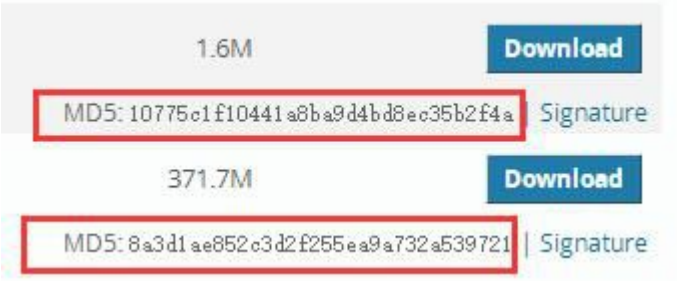
1.4.1 常见算法

MD5、SHA、CRC 等

1.4.2 使用场景

1. 对用户密码进行 md5 加密后保存到数据库里
2. 软件下载站使用消息摘要计算文件指纹，防止被篡改
3. 数字签名（后面知识点）

例如软件下载站数据指纹：
http://dev.mysql.com/downloads/installer/



1.4.3 使用步骤

```
1.      //常用算法：MD5、SHA、CRC
2.      MessageDigest digest = MessageDigest.getInstance("MD5");
3.      byte[] result = digest.digest(content.getBytes());
4.      //消息摘要的结果一般都是转换成 16 进制字符串形式展示
5.      String hex = Hex.encode(result);

6.      //MD5 结果为 16 字节（128 个比特位）、转换为 16 进制表示后长度是 32 个字符
7.      //SHA 结果为 20 字节（160 个比特位）、转换为 16 进制表示后长度是 40 个字符
8.      System.out.println(hex);
```

消息摘要后的结果是固定长度，无论你的数据有多大，哪怕是只有一个字节或者是一个 G 的文件，摘要后的结果都是固定长度。

经常听到有人问这样的问题，MD5 摘要后结果到底是多少位？有的人说是 16 位，有的说是 128 位，有的说是 32 位。到底是多长，这个时候我们就要明白，16 位指的是字节位数，128 位指的是比特位，32 位指的结果转换成 16 进制展示的字符位数。

1.5 数字签名

数字签名是**非对称加密**与**数字摘要**的组合应用

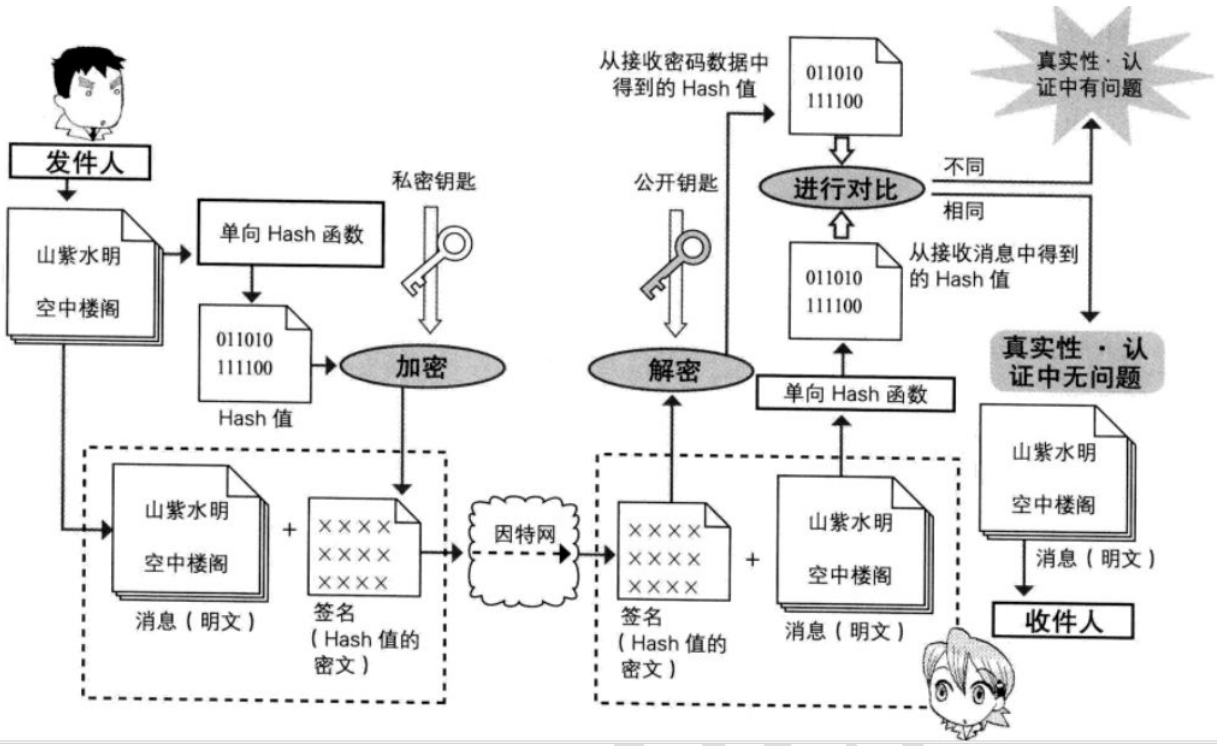
1.5.1 应用场景

1. 校验用户身份（使用私钥签名，公钥校验，只要用公钥能校验通过，则该信息一定是私钥持有者发布的）
2. 校验数据的完整性（用解密后的消息摘要跟原文的消息摘要进行对比）

1.5.2 签名过程

“发送报文时，发送方用一个哈希函数从报文文本中生成报文摘要,然后用自己的私人密钥对这个摘要进行加密，这个加密后的摘要将作为报文的数字签名和报文一起发送给接收方，接收方首先用与发送方一样的哈希函数从接收到的原始报文中计算出报文摘要，接着再用发送方的公用密钥来对报文附加的数字签名进行解密，如果这两个摘要相同、那么接收方就能确认该数字签名是发送方的。

数字签名有两种功效：一是能确定消息确实是由发送方签名并发出来的，因为别人假冒不了发送方的签名。二是数字签名能确定消息的完整性。因为数字签名的特点是它代表了文件的特征，文件如果发生改变，数字摘要的值也将发生变化。不同的文件将得到不同的数字摘要。 一次数字签名涉及到一个哈希函数、发送者的公钥、发送者的私钥。”



1.5.3 使用步骤

```
1.          //获取 signature 对象，初始化算法：MD2withRSA, MD5withRSA, or SHA1withRSA
2.          Signature signature = Signature.getInstance("MD5withRSA");
3.          //创建私钥（从磁盘上读取）
4.          PrivateKey  privateKey = (PrivateKey)SerializableUtil.readObject(
5.  "heima.privateKey");
6.          //使用私钥进行初始化
7.          signature.initSign(privateKey);
8.          //传入需要签名的数据
9.          signature.update(content.getBytes());
10.         //执行签名
11.         byte[] sign = signature.sign();
12.
13.         //创建公钥（从磁盘上读取）
14.        PublicKey publicKey = (PublicKey) SerializableUtil.readObject(
15. "heima.publicKey");
16.         //使用公钥进行初始化
17.         signature.initVerify(publicKey);
18.         //传入需要校验的数据（即上面的原文）
19.         signature.update(content.getBytes());
20.         //执行校验
21.         boolean verify = signature.verify(sign);
```

1.5.4 总结

数字签名一般不单独使用，基本都是用在数字证书里实现 SSL 通信协议。下面将学习的数字证书就是基于数字签名技术实现的。

1.6 数字证书

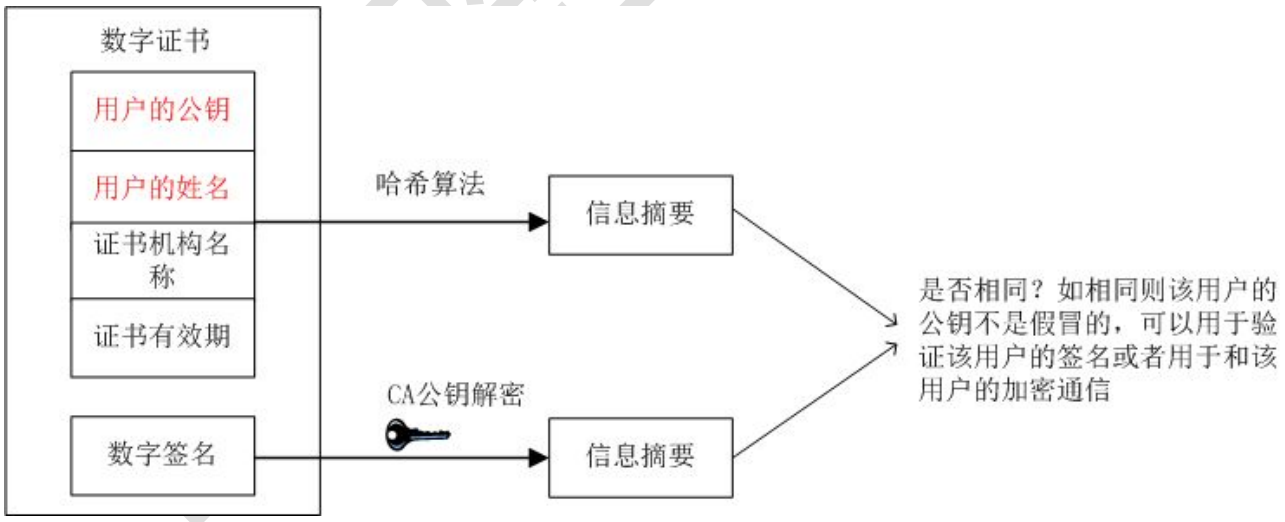
数字证书就是互联网通讯中标志通讯各方身份信息的一串数字，提供了一种在 Internet 上验证通信实体身份的方式，数字证书不是数字身份证，而是身份认证机构盖在数字身份证上的一个章或印（或者说加在数字身份证上的一个签名）。它是由权威机构——CA 机构，又称为证书授权（Certificate Authority）中心发行的，人们可以在网上用它来识别对方的身份。

1.6.1 应用场景

1，交易者身份的确定性、不可否认性、不可修改性

2，对应用进行签名认证（例如 Android 的 apk）

1.6.2 数字证书格式

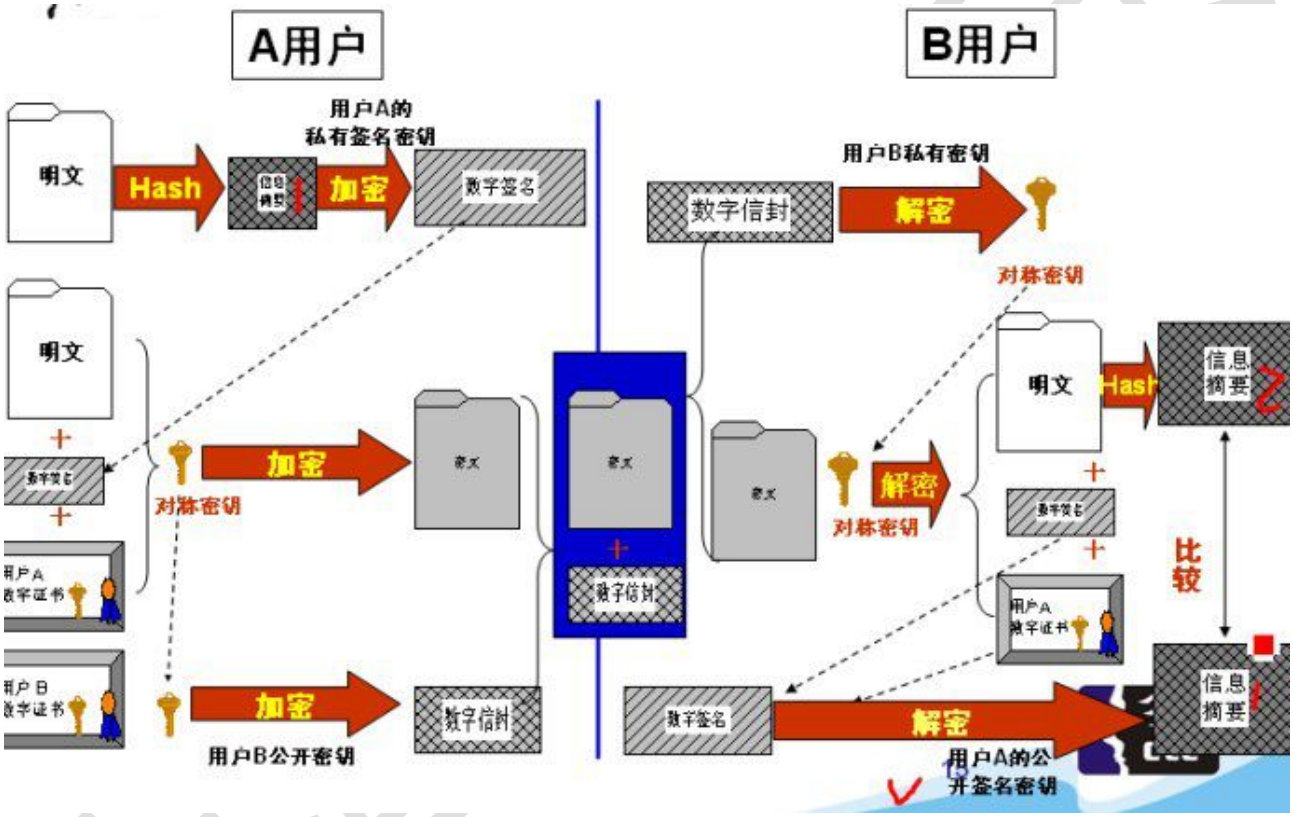


数字证书的格式普遍采用的是 X.509V3 国际标准，一个标准的 X.509 数字证书包含以下一些内容：

证书的版本信息；
证书的序列号，每个证书都有一个唯一的证书序列号；
证书所使用的签名算法；
证书的发行机构名称，命名规则一般采用 X.500 格式；

证书的有效期，通用的证书一般采用 UTC 时间格式，它的计时范围为 1950-2049；
证书所有人的名称，命名规则一般采用 X.500 格式；
证书所有人的公开密钥；
证书发行者对证书的签名。

1.6.3 数字证书原理



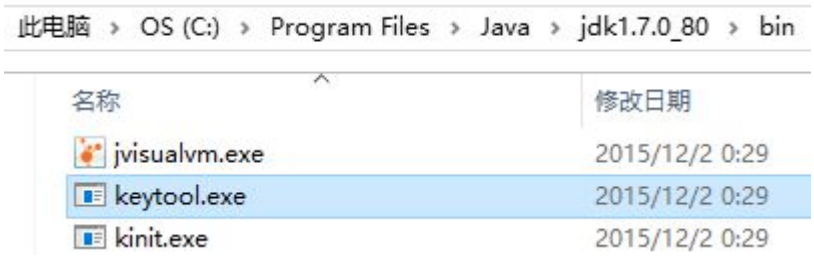
数字证书是安全领域里的终极武器，SSL 通信协议里最核心的东西就是数字证书。他涉及到前面提到的所有知识：对称加密、非对称加密、消息摘要、数字签名等。

数字证书可以通过 java 自带的 KeyTool 工具生成，生成后的数字证书一般保管在 KeyStore 里。KeyStore 可以叫做秘钥仓库。

秘钥仓库可以保管 3 种类型的数据：KeyStore.PrivateKeyEntry（非对称机密里的私钥）、KeyStore.SecretKeyEntry （对称加密里的秘钥）、KeyStore.TrustedCertificateEntry（受信任的证书）

1.6.4 Keyto 工具

路径：jre\bin\keytool.exe



常用命令：

生成 keypair

keytool -genkeypair

keytool -genkeypair -alias lisi（后面部分是为证书指定别名，否则采用默认的名称为 mykey）

看看 keystore 中有哪些项目：

keytool -list 或 keytool -list -v

keytool -exportcert -alias lisi -file lisi.cer

生成可打印的证书：

keytool -exportcert -alias lisi -file lisi.cer –rfc

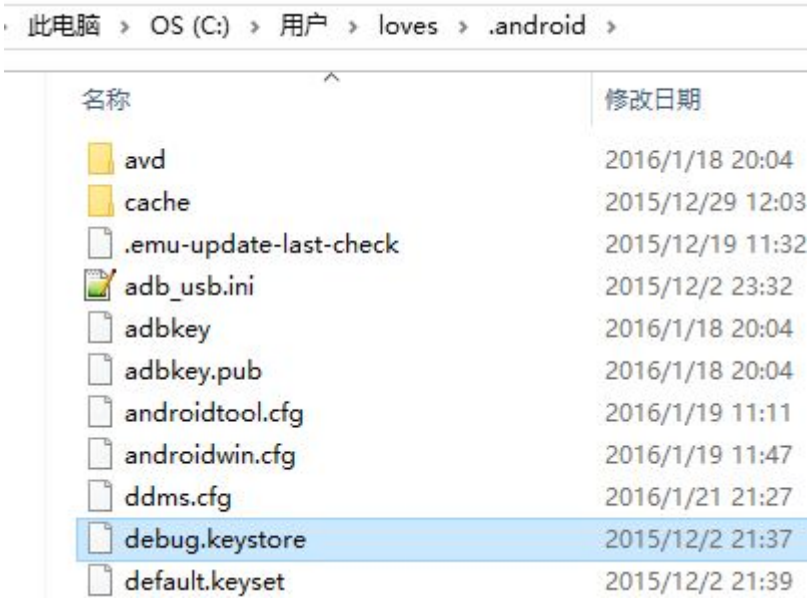
显示数字证书文件中的证书信息：

keytool -printcert -file lisi.cer

直接双击 lisi.cer，用 window 系统的内置程序打开 lisi.cer

1.6.5 Android 的 keystore 相关知识：

debug 签名路径：user\.android\debug.keystore



debug.keystore 的别名（alias）及密码:
别名：androiddebugkey，密码：android

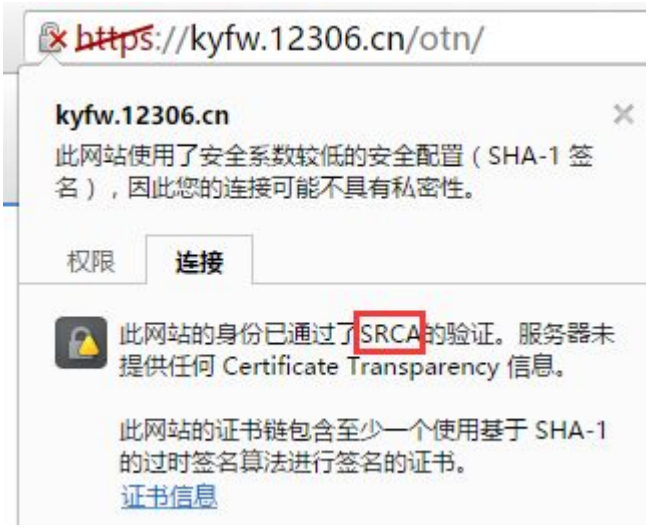
签名命令(jdk1.6)：
jarsigner -verbose -keystore debug.keystore -signedjar 1signed.apk 1.apk androiddebugkey
签名命令(jdk1.7)：
jarsigner -verbose -keystore debug.keystore -signedjar 1signed.apk 1.apk androiddebugkey -digestalg SHA1 -sigalg MD5withRSA

优化命令：zipalign -v 4 1signed.apk 1signedaligned.apk
验证签名是否成功：jarsigner -verify 1signed.apk

1.6.6 补充

签名证书：
由权威颁发机构颁发给服务器或者个人用于证明自己身份的东西，默认客户端都是信任的。主要目的是用来加密和保证数据的完整性和不可抵赖性
例如根证书机构 Symantec 颁发给百度的就是签名证书，是受信任的。

自签名证书：
由服务器自己颁发给自己，用于证明自己身份的东西，非权威颁发机构发布，默认客户端都是不信任的，主要目的是用来加密和保证数据的完整性和不可抵赖性,与签名证书相同.
例如中铁集团（SRCA）办法给 12306 的证书就是自签名证书，自己给自己颁发的。



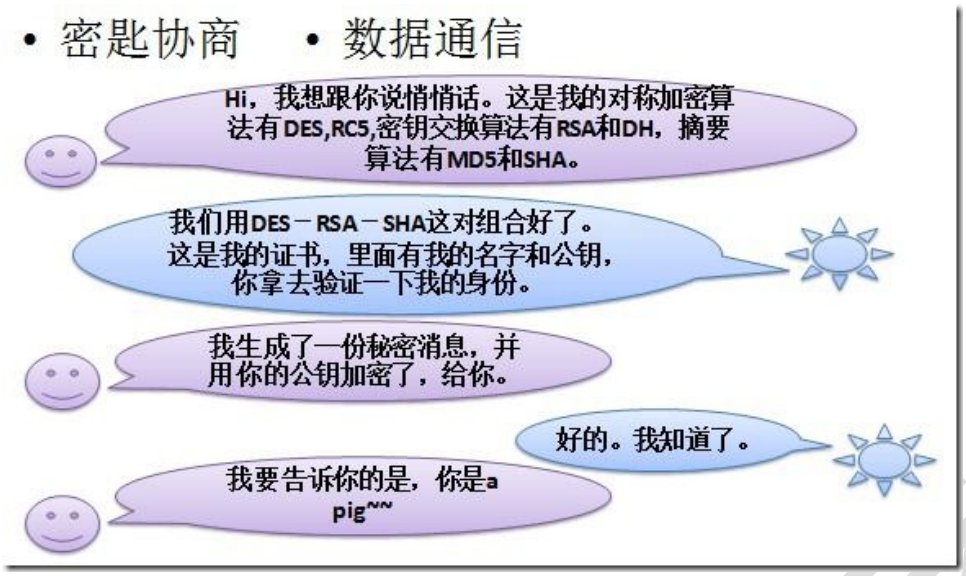
1.7 Https 编程

1.7.1 介绍

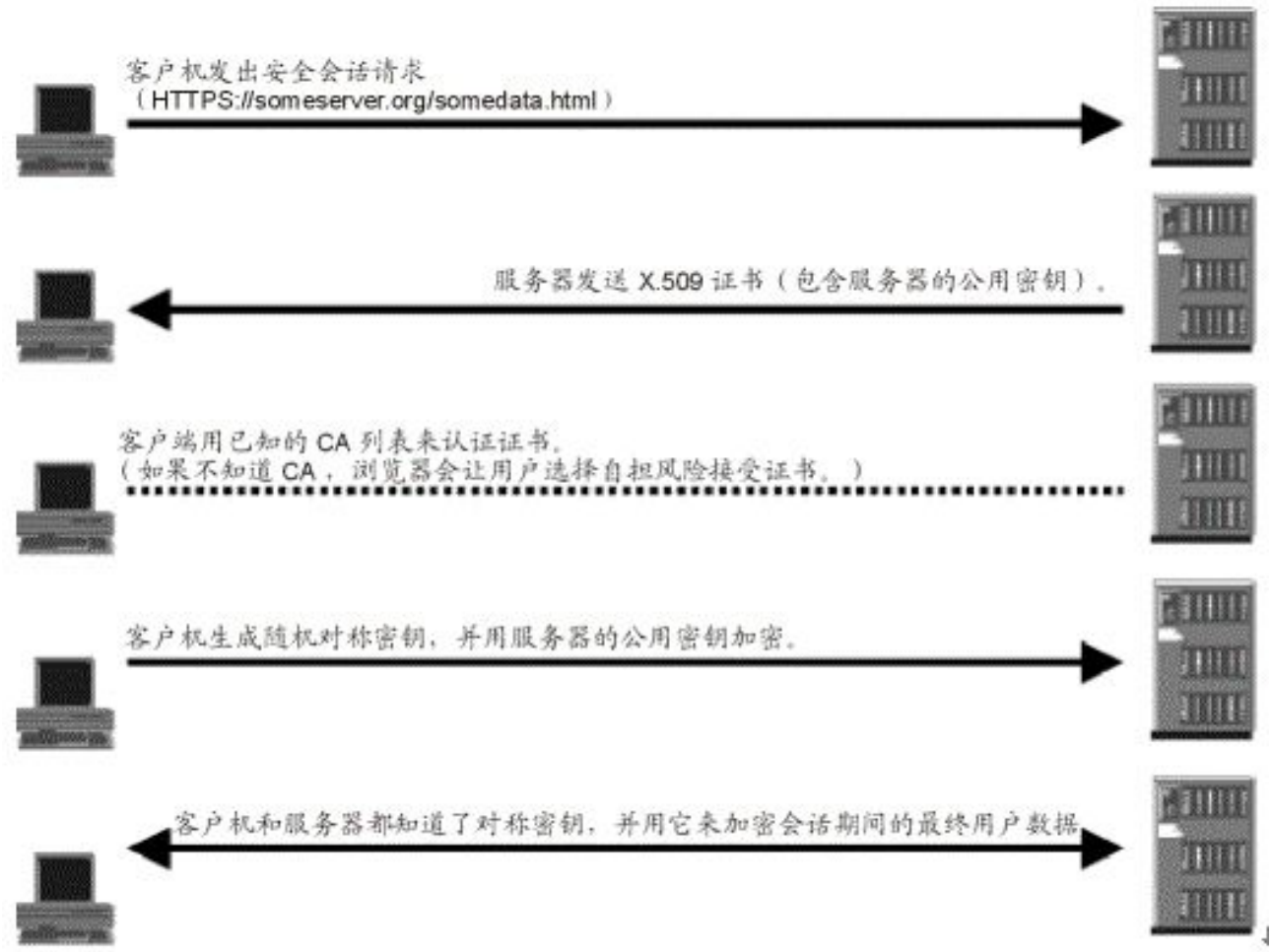
SSL(Secure Sockets Layer 安全套接层)，为网景公司(Netscape)所研发，用以保障在 Internet 上数据传输之安全，利用数据加密(Encryption)技术，可确保数据在网络上之传输过程中不会被截取及窃听。一般通用之规格为 40 bit 之安全标准，美国则已推出 128 bit 之更高安全标准，但限制出境。只要 3.0 版本以上之 I.E.或 Netscape 浏览器即可支持 SSL。

TLS（Transport Layer Security 传输层安全），用于在两个通信应用程序之间提供保密性和数据完整性。TLS 是 SSL 的标准化后的产物，有 1.0 ，1.1 ，1.2 三个版本，默认使用 1.0。TLS1.0 和 SSL3.0 几乎没有区别，事实上我们现在用的都是 TLS，但因为历史上习惯了 SSL 这个称呼。

SSL 通信简单图示：



SSL 通信详细图示：



当请求使用自签名证书的网站数据时，例如请求 12306 的客运服务页面：https://kyfw.12306.cn/otn/，

则会报下面的错误，原因是客户端的根认证机构不能识别该证书

错误信息：unable to find valid certification path to requested target

1.7.2 解决方案 1

一个证书可不可信，是由 TrustManager 决定的，所以我们只需要自定义一个什么都不做的 TrustManager 即可，服务器出示的所有证书都不做校验，一律放行。

```
1.      public static void main(String[] args) throws Exception {
2.          //协议传输层安全 TLS(transport layer secure)
3.          SSLContext sslContext = SSLContext.getInstance("TLS");
4.          //创建信任管理器（TrustManager 负责校验证书是否可信）
5.          TrustManager[] tm = new TrustManager[]{new EmptyX509TrustManager()};
6.          //使用自定义的信任管理器初始化 SSL 上下文对象
7.          sslContext.init(null, tm, null);
8.          //设置全局的 SSLSocketFactory 工厂（对所有 ssl 链接都产生影响）
9.          HttpsURLConnection.setDefaultSSLSocketFactory(sslContext.getSocketFactory());
10.
11.         //URL url = new URL("https://www.baidu.com");
12.         URL url = new URL("https://kyfw.12306.cn/otn/");
13.         HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
14.         InputStream in = conn.getInputStream();
15.         System.out.println(Util.inputstream2String(in));
16.     }
17.
18.     /**
19.      * 自定义一个什么都不做的信任管理器，所有证书都不做校验，一律放行
20.      */
21.     private static class EmptyX509TrustManager implements X509TrustManager{
22.         @Override
23.         public void checkClientTrusted(X509Certificate[] chain, String authType)
24.                 throws CertificateException {
25.         }

26.         @Override
27.         public void checkServerTrusted(X509Certificate[] chain, String authType)
28.                 throws CertificateException {
29.         }

30.         @Override
31.         public X509Certificate[] getAcceptedIssuers() {
32.             return null;
33.         }
```

```
34.
35.     }
```

1.7.3 解决方案 2

12306 服务器出示的证书是中铁集团 SRCA 给他颁发的，所以 SRCA 的证书是能够识别 12306 的证书的，所以只需要把 SRCA 证书导入系统的 KeyStore 里，之后交给 TrustManagerFactory 进行初始化，则可把 SRCA 添加至根证书认证机构，之后校验的时候，SRCA 对 12306 证书校验时就能通过认证。

这种解决方案有两种使用方式：一是直接使用 SRCA.cer 文件，二是使用改文件的 RFC 格式数据，将其写在代码里。

```
1.      //12306 证书的 RFC 格式（注意要记得手动添加两个换行符）
2.      private static final String CERT_12306_RFC = "-----BEGIN CERTIFICATE-----\n"
3.              +
    "MIICmjCCAgOgAwIBAgIIbyZr5/jKH6QwDQYJKoZIhvcNAQEFBQAwRzELMAkGA1UEBhMCQ04xKTAn"
4.              +
    "BgNVBAoTIFNpbm9yYWlsIENlcnRpZmljYXRpb24gQXV0aG9yaXR5MQ0wCwYDVQQDEwRTUkNBMB4X"
5.              +
    "DTA5MDUyNTA2NTYwMFoXDTI5MDUyMDA2NTYwMFowRzELMAkGA1UEBhMCQ04xKTAnBgNVBAoTIFNp"
6.              +
    "bm9yYWlsIENlcnRpZmljYXRpb24gQXV0aG9yaXR5MQ0wCwYDVQQDEwRTUkNBMIGfMA0GCSqGSIb3"
7.              +
    "DQEBAQUAA4GNADCBiQKBgQDMpbNeb34p0GvLkZ6t72/OOba4mX2K/eZRWFfnuk8e5jKDH+9BgCb2"
8.              +
    "9bSotqPqTbxXWPxIOz8EjyUO3bfR5pQ8ovNTOlks2rS5BdMhoi4sUjCKi5ELiqtyww/XgY5iFqv6"
9.              +
    "D4Pw9QvOUcdRVSbPWo1DwMmH75It6pk/rARIFHEjWwIDAQABo4GOMIGLMB8GA1UdIwQYMBaAFHle"
10.             +
    "tne34lKDQ+3HUYhMY4UsAENYMAwGA1UdEwQFMAMBAf8wLgYDVR0fBCcwJTAjoCGgH4YdaHR0cDov"
11.             +
    "LzE5Mi4xNjguOS4xNDkvY3JsMS5jcmwwCwYDVR0PBAQDAgH+MB0GA1UdDgQWBBR5XrZ3t+JSg0Pt"
12.             +
    "x1GITGOFLABDWDANBgkqhkiG9w0BAQUFAAOBgQDGrAm2U/of1LbOnG2bnnQtgcVaBXiVJF8LKPaV"
13.             +
    "23XQ96HU8xfgSZMJS6U00WHAI7zp0q208RSUft9wDq9ee///VOhzR6Tebg9QfyPSohkBrhXQenvQ"
14.             + "og555S+C3eJAAVeNCTeMS3N/M5hzBRJAoffn3qoYdAO1Q8bTguOi+2849A=="
15.             + "-----END CERTIFICATE-----\n";

16.     public static void main(String[] args) throws Exception {
17.         // 使用传输层安全协议 TLS(transport layer secure)
18.         SSLContext sslContext = SSLContext.getInstance("TLS");
19.         //使用 SRCA.cer 文件的形式
```

```
20.         //FileInputStream certInputStream = new FileInputStream(new File("srca.cer"));
21.         //也可以通过 RFC 字符串的形式使用证书
22.         ByteArrayInputStream certInputStream = new
    ByteArrayInputStream(CERT_12306_RFC.getBytes());
23.         // 初始化 keyStore，用来导入证书
24.         KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
25.         //参数 null 表示使用系统默认 keystore，也可使用其他 keystore（需事先将 srca.cer 证书导入
    keystore 里）
26.         keyStore.load(null);
27.         //通过流创建一个证书
28.         Certificate certificate = CertificateFactory.getInstance("X.509")
29.                 .generateCertificate(certInputStream);
30.         // 把 srca.cer 这个证书导入到 KeyStore 里，别名叫做 srca
31.         keyStore.setCertificateEntry("srca", certificate);
32.         // 设置使用 keyStore 去进行证书校验
33.         TrustManagerFactory trustManagerFactory = TrustManagerFactory
34.                 .getInstance(TrustManagerFactory.getDefaultAlgorithm());
35.         trustManagerFactory.init(keyStore);
36.         //用我们设定好的 TrustManager 去做 ssl 通信协议校验，即证书校验
37.         sslContext.init(null, trustManagerFactory.getTrustManagers(), null);
38.         HttpsURLConnection.setDefaultSSLSocketFactory(sslContext
39.                 .getSocketFactory());

40.         URL url = new URL("https://kyfw.12306.cn/otn/");
41.         HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
42.         InputStream in = conn.getInputStream();
43.         System.out.println(Util.inputstream2String(in));
44.     }
```

1.7.4 Android 里的 https 请求：

把 scra.cer 文件考到 assets 或 raw 目录下，或者直接使用证书的 RFC 格式，接下来的做法和 java 工程代码一样

1.8 课程总结

以上学习所有内容，对称加密、非对称加密、消息摘要、数字签名等知识都是为了理解数字证书工作原理而作为一个预备知识。数字证书是密码学里的终极武器，是人类几千年历史总结的智慧的结晶，只有

在明白了数字证书工作原理后，才能理解 Https 协议的安全通讯机制。最终才能在 SSL 开发过程中得心应手。

另外，对称加密和消息摘要这两个知识点是可以单独拿来使用的。

知识点串联：

数字证书使用到了以上学习的所有知识

1，对称加密与非对称加密结合使用实现了秘钥交换，之后通信双方使用该秘钥进行对称加密通信。

2，消息摘要与非对称加密实现了数字签名，根证书机构对目标证书进行签名，在校验的时候，根证书用公钥对其进行校验。若校验成功，则说明该证书是受信任的。

3，Keytool 工具可以创建证书，之后交给根证书机构认证后直接使用自签名证书，还可以输出证书的 RFC 格式信息等。

4，数字签名技术实现了身份认证与数据完整性保证。

5，加密技术保证了数据的保密性，消息摘要算法保证了数据的完整性，对称加密的高效保证了数据处理的可靠性，数字签名技术保证了操作的不可否认性。

通过以上内容的学习，我们要能掌握以下知识点：

1. 基础知识：bit 位、字节、字符、字符编码、进制转换、io
2. 知道怎样在实际开发里怎样使用对称加密解决问题
3. 知道对称加密、非对称加密、消息摘要、数字签名、数字证书是为了解决什么问题而出现的
4. 了解 SSL 通讯流程
5. 实际开发里怎样请求 Https 的接口