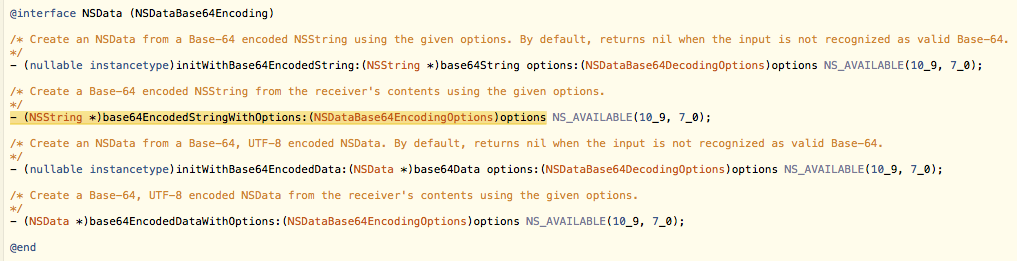
**参考文章**

* [Base64](http://www.cnblogs.com/yiki/archive/2009/01/18/1377828.html)
* [Base64编码/解码](http://base64.xpcha.com/)
* [填充算法，mac与java的区别](http://www.douban.com/note/75830899/)
* [块密码的工作模式](http://zh.wikipedia.org/wiki/%E5%9D%97%E5%AF%86%E7%A0%81%E7%9A%84%E5%B7%A5%E4%BD%9C%E6%A8%A1%E5%BC%8F)
* [分组密码](http://zh.wikipedia.org/wiki/%E5%88%86%E7%BB%84%E5%AF%86%E7%A0%81)
* [PKCS](http://en.wikipedia.org/wiki/PKCS)

**Base64编码**

感谢[Lonely\_\_](http://www.jianshu.com/users/8df293c6d308" \t "_blank)和[angelababa](http://www.jianshu.com/users/e7a8df26ee1b" \t "_blank)的提醒,苹果是有Base64的[API](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSData_Class/" \l "//apple_ref/occ/instm/NSData/base64EncodedStringWithOptions:" \t "_blank),截图如下:



苹果提供Base64API.png

Base64编码的思想是采用64个基本的ASCII码字符对数据进行重新编码。它将需要编码的数据拆分成字节数组。以3个字节为一组。按顺序排列24 位数据，再把这24位数据分成4组，即每组6位。再在每组的的最高位前补两个0凑足一个字节。这样就把一个3字节为一组的数据重新编码成了4个字节。当所要编码的数据的字节数不是3的整倍数，也就是说在分组时最后一组不够3个字节。这时在最后一组填充1到2个0字节。并在最后编码完成后在结尾添加1到2个 “=”。

例：将对ABC进行BASE64编码：

1、首先取ABC对应的ASCII码值。A（65）B（66）C（67）；

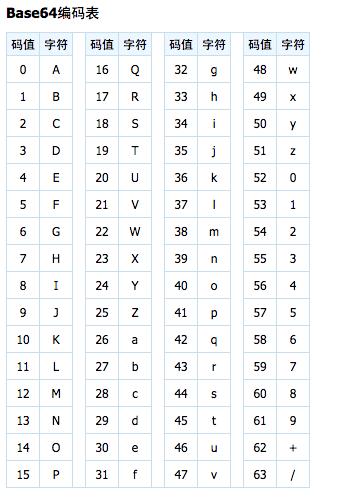
2、再取二进制值A（01000001）B（01000010）C（01000011）；

3、然后把这三个字节的二进制码接起来（010000010100001001000011）；

4、 再以6位为单位分成4个数据块,并在最高位填充两个0后形成4个字节的编码后的值，（00010000）（00010100）（00001001）（00000011），其中加色部分为真实数据；

5、再把这四个字节数据转化成10进制数得（16）（20）（9）（3）；

6、最后根据BASE64给出的64个基本字符表，查出对应的ASCII码字符（Q）（U）（J）（D），这里的值实际就是数据在字符表中的索引。



Base64编码表

解码过程就是把4个字节再还原成3个字节再根据不同的数据形式把字节数组重新整理成数据。

Base64很直观的目的就是让二进制文件转化为64个基本的ASCII码字符，（me:使得不可打印的数据能够被打印出来）。

**AES**

系统也并没有直接提供诸如DES、AES的API，但是提供了加密解密的相关操作CommonCrypto,DES或者AES的实现，需要我们自己封装一下。

加密是由算法/模式/填充组成的，算法是DES,AES等, 模式是EBC,CBC等，iOS和Android的填充是不一样的：

mac支持：

NoPadding (NoPadding就是不填充，相当于自定义填充)

PKCS7Padding

而java支持:

NoPadding

ISO10126Padding

OAEPPadding, OAEPWith<digest>And<mgf>Padding

PKCS1Padding

PKCS5Padding

SSL3Padding

**接下来我们引入一些背景知识：**

在密码学中，分组加密（Block cipher，又称分块加密），是一种对称密钥算法。它将明文分成多个等长的模块（block），使用确定的算法和对称密钥对每组分别加密解密。分组加密是极其重要的加密协议组成，其中典型的如DES和AES作为美国政府核定的标准加密算法，应用领域从电子邮件加密到银行交易转帐，非常广泛。

密码学中的工作模式：

最早出现的工作模式，ECB，CBC，OFB和CFB可以追溯到1981年。2001年，NIST修订了其早先发布的工作模式工作列表，加入了AES，并加入了CTR模式。最后，在2010年1月，NIST加入了XTS-AES，而其余的可信模式并没有为NIST所认证。例如CTS是一种密文窃取的模式，许多常见的密码学运行库提供了这种模式。

密码学中，块密码的工作模式允许使用同一个块密码密钥对多于一块的数据进行加密，并保证其安全性。块密码自身只能加密长度等于密码块长度的单块数据，若要加密变长数据，则数据必须先被划分为一些单独的密码块。通常而言，最后一块数据也需要使用合适填充方式将数据扩展到符合密码块大小的长度。一种工作模式描述了加密每一数据块的过程，并常常使用基于一个通常称为初始化向量的附加输入值以进行随机化，以保证安全。

初始化向量：

初始化向量（IV，Initialization Vector）是许多工作模式中用于随机化加密的一块数据，因此可以由相同的明文，相同的密钥产生不同的密文，而无需重新产生密钥，避免了通常相当复杂的这一过程。

初始化向量与密钥相比有不同的安全性需求，因此IV通常无须保密，然而在大多数情况中，不应当在使用同一密钥的情况下两次使用同一个IV。对于CBC和CFB，重用IV会导致泄露明文首个块的某些信息，亦包括两个不同消息中相同的前缀。对于OFB和CTR而言，重用IV会导致完全失去安全性。另外，在CBC模式中，IV在加密时必须是无法预测的；特别的，在许多实现中使用的产生IV的方法，例如SSL2.0使用的，即采用上一个消息的最后一块密文作为下一个消息的IV，是不安全的。

**注意：ECB模式不需要初始化向量，之所以提一句，是因为我用的ECB模式。**

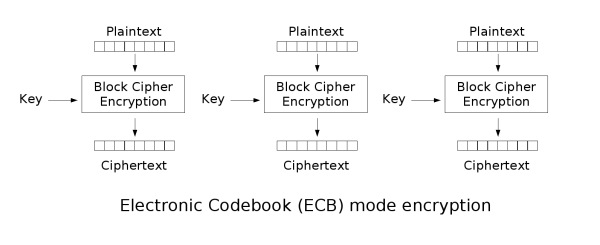
填充：

块密码（me：块加密，将被加密数据分为多个定长的数据块分别应用算法和模式进行加密，如果最后块的长度不够需要进行填充）只能对确定长度的数据块进行处理，而消息的长度通常是可变的。因此部分模式（即ECB和CBC）需要最后一块在加密前进行填充。有数种填充方法，其中最简单的一种是在平文的最后填充空字符以使其长度为块长度的整数倍，但必须保证可以恢复平文的原始长度；例如，若平文是C语言风格的字符串，则只有串尾会有空字符。稍微复杂一点的方法则是原始的DES使用的方法，即在数据后添加一个1位，再添加足够的0位直到满足块长度的要求；若消息长度刚好符合块长度，则添加一个填充块。最复杂的则是针对CBC的方法，例如密文窃取，残块终结等，不会产生额外的密文，但会增加一些复杂度。布鲁斯·施奈尔和尼尔斯·弗格森提出了两种简单的可能性：添加一个值为128的字节（十六进制的80），再以0字节填满最后一个块；或向最后一个块填充n个值均为n的字节。

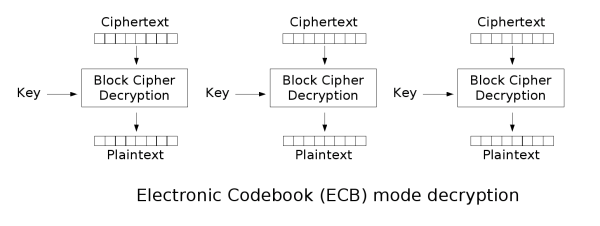
CFB，OFB和CTR模式不需要对长度不为密码块大小整数倍的消息进行特别的处理。因为这些模式是通过对块密码的输出与平文进行异或工作的。最后一个平文块（可能是不完整的）与密钥流块的前几个字节异或后，产生了与该平文块大小相同的密文块。流密码的这个特性使得它们可以应用在需要密文和平文数据长度严格相等的场合，也可以应用在以流形式传输数据而不便于进行填充的场合。

**注意：ECB模式是需要填充的。**

ECB:  
最简单的加密模式即为电子密码本（Electronic codebook，ECB）模式。需要加密的消息按照块密码的块大小被分为数个块，并对每个块进行独立加密。



ECB加密

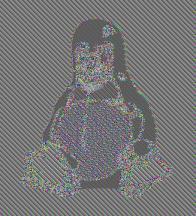


ECB解密

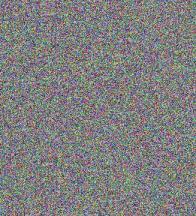
本方法的缺点在于同样的平文块会被加密成相同的密文块；因此，它不能很好的隐藏数据模式。在某些场合，这种方法不能提供严格的数据保密性，因此并不推荐用于密码协议中。下面的例子显示了ECB在密文中显示平文的模式的程度：该图像的一个位图版本（上图）通过ECB模式可能会被加密成中图，而非ECB模式通常会将其加密成最下图。



原图



使用ECB模式加密



提供了伪随机性的非ECB模式

原图是使用CBC，CTR或任何其它的更安全的模式加密最下图可能产生的结果——与随机噪声无异。注意最下图看起来的随机性并不能表示图像已经被安全的加密；许多不安全的加密法也可能产生这种“随机的”输出。

ECB模式也会导致使用它的协议不能提供数据完整性保护，易受到重放攻击的影响，因此每个块是以完全相同的方式解密的。例如，“梦幻之星在线：蓝色脉冲”在线电子游戏使用ECB模式的Blowfish密码。在密钥交换系统被破解而产生更简单的破解方式前，作弊者重复通过发送加密的“杀死怪物”消息包以非法的快速增加经验值。

其他模式在此就不展开了，详情请转[块密码的工作模式](http://zh.wikipedia.org/wiki/%E5%9D%97%E5%AF%86%E7%A0%81%E7%9A%84%E5%B7%A5%E4%BD%9C%E6%A8%A1%E5%BC%8F) ，进一步了解CBC、CFB、OFB、CTR等模式。

把最重要的函数摘出来解释一下：

/\*!

@function CCCrypt

@abstract Stateless, one-shot encrypt or decrypt operation.

This basically performs a sequence of CCCrytorCreate(),

CCCryptorUpdate(), CCCryptorFinal(), and CCCryptorRelease().

@param alg Defines the encryption algorithm.

@param op Defines the basic operation: kCCEncrypt or

kCCDecrypt.

@param options A word of flags defining options. See discussion

for the CCOptions type.

@param key Raw key material, length keyLength bytes.

@param keyLength Length of key material. Must be appropriate

for the select algorithm. Some algorithms may

provide for varying key lengths.

@param iv Initialization vector, optional. Used for

Cipher Block Chaining (CBC) mode. If present,

must be the same length as the selected

algorithm's block size. If CBC mode is

selected (by the absence of any mode bits in

the options flags) and no IV is present, a

NULL (all zeroes) IV will be used. This is

ignored if ECB mode is used or if a stream

cipher algorithm is selected.

@param dataIn Data to encrypt or decrypt, length dataInLength

bytes.

@param dataInLength Length of data to encrypt or decrypt.

@param dataOut Result is written here. Allocated by caller.

Encryption and decryption can be performed

"in-place", with the same buffer used for

input and output.

@param dataOutAvailable The size of the dataOut buffer in bytes.

@param dataOutMoved On successful return, the number of bytes

written to dataOut. If kCCBufferTooSmall is

returned as a result of insufficient buffer

space being provided, the required buffer space

is returned here.

@result kCCBufferTooSmall indicates insufficent space in the dataOut

buffer. In this case, the \*dataOutMoved

parameter will indicate the size of the buffer

needed to complete the operation. The

operation can be retried with minimal runtime

penalty.

kCCAlignmentError indicates that dataInLength was not properly

aligned. This can only be returned for block

ciphers, and then only when decrypting or when

encrypting with block with padding disabled.

kCCDecodeError Indicates improperly formatted ciphertext or

a "wrong key" error; occurs only during decrypt

operations.

\*/

CCCryptorStatus CCCrypt(

CCOperation op, /\* 枚举值，确认是加密操作，还是解密操作 \*/

CCAlgorithm alg, /\* 枚举值，确认加解密的算法，如kCCAlgorithmAES128、kCCAlgorithmDES \*/

CCOptions options, /\* 枚举值，kCCOptionPKCS7Padding | kCCOptionECBMode，经我调查，这样就是ECB模式，并以PKCS7来填充\*/

const void \*key,

size\_t keyLength,

const void \*iv, /\* 初始化向量（NULLoptional initialization vector），ECB模式写NULL就行 \*/

const void \*dataIn, /\* optional per op and alg \*/

size\_t dataInLength,

void \*dataOut, /\* data RETURNED here \*/

size\_t dataOutAvailable,

size\_t \*dataOutMoved)

上面说到，iOS和Android填充是不一样的，那怎么办？据说，**PKCS7Padding是兼容PKCS5Padding的**，我在与安卓联合测试中，确实没有问题。

把我用的AES加密摘出来吧：

我用的是一个NSData类目NSData+AES，密钥是128位的，即16个字节，加密解密方法的实现如下（记得引#import <CommonCrypto/CommonCryptor.h>）：

加密：

- (NSData \*)AES128EncryptWithKey:(NSString \*)key

{

// 'key' should be 32 bytes for AES256, will be null-padded otherwise

char keyPtr[kCCKeySizeAES128+1]; // room for terminator (unused)

bzero(keyPtr, sizeof(keyPtr)); // fill with zeroes (for padding)

// fetch key data

[key getCString:keyPtr maxLength:sizeof(keyPtr) encoding:NSUTF8StringEncoding];

NSUInteger dataLength = [self length];

//See the doc: For block ciphers, the output size will always be less than or

//equal to the input size plus the size of one block.

//That's why we need to add the size of one block here

size\_t bufferSize = dataLength + kCCBlockSizeAES128;

void \*buffer = malloc(bufferSize);

size\_t numBytesEncrypted = 0;

CCCryptorStatus cryptStatus = CCCrypt(kCCEncrypt, kCCAlgorithmAES128, kCCOptionPKCS7Padding | kCCOptionECBMode,

keyPtr, kCCKeySizeAES128,

NULL /\* initialization vector (optional) \*/,

[self bytes], dataLength, /\* input \*/

buffer, bufferSize, /\* output \*/

&numBytesEncrypted);

if (cryptStatus == kCCSuccess) {

//the returned NSData takes ownership of the buffer and will free it on deallocation

return [NSData dataWithBytesNoCopy:buffer length:numBytesEncrypted];

}

free(buffer); //free the buffer;

return nil;

}

解密：

- (NSData \*)AES128DecryptWithKey:(NSString \*)key {

// 'key' should be 32 bytes for AES256, will be null-padded otherwise

char keyPtr[kCCKeySizeAES128+1]; // room for terminator (unused)

bzero(keyPtr, sizeof(keyPtr)); // fill with zeroes (for padding)

// fetch key data

[key getCString:keyPtr maxLength:sizeof(keyPtr) encoding:NSUTF8StringEncoding];

NSUInteger dataLength = [self length];

//See the doc: For block ciphers, the output size will always be less than or

//equal to the input size plus the size of one block.

//That's why we need to add the size of one block here

size\_t bufferSize = dataLength + kCCBlockSizeAES128;

void \*buffer = malloc(bufferSize);

size\_t numBytesDecrypted = 0;

CCCryptorStatus cryptStatus = CCCrypt(kCCDecrypt, kCCAlgorithmAES128, kCCOptionPKCS7Padding| kCCOptionECBMode,

keyPtr, kCCKeySizeAES128,

NULL /\* initialization vector (optional) \*/,

[self bytes], dataLength, /\* input \*/

buffer, bufferSize, /\* output \*/

&numBytesDecrypted);

if (cryptStatus == kCCSuccess) {

//the returned NSData takes ownership of the buffer and will free it on deallocation

return [NSData dataWithBytesNoCopy:buffer length:numBytesDecrypted];

}

free(buffer); //free the buffer;

return nil;

}