密码学

# 1.DES

## 1.1.DES的ECB模式实现

ECB模式的DES算法，输入8字节的明文/密文、8字节的密钥，可执行加密/解密操作得到密文/明文。

DES本身是一个对称算法，加密和解密使用同一组算法即可实现。

### 1.1.1.概述

算法原理见图1：

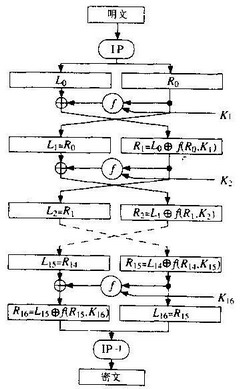


图1 DES算法原理图

详细描述如下：

**Algo. DES**

Input : unsigned char input[8];

Output : unsigned char output[8];

1. 将输入的8字节input内容，转换成64bit的表现形式后，通过IP\_table进行置换，得到全新序列的64bit数据；
2. 对step1中得到的数据，分组成为两个32bit的数据，表示为Li和Ri；
3. 对Li和Ri循环执行16轮变换，变换规则为：Li+1 = Ri; Ri+1 = Li xor f(Ri, Ki)；
4. 16轮循环结束后，将Left和Right的值互换；
5. 将互换后的left和right拼接成64bit数据后，通过IP\_inv\_table进行反置换；
6. 将step4得到的64bit数据组合成8字节的形式保存到output中，并返回；

可见，算法中除step3外，其他步骤比较容易理解。对于每个步骤中使用到的表格，将在程序中给出定义，且这些表格是DES标准中已经定义好的，无需做任何改动。IP\_table定义如下：

const static unsigned char IP\_table[DES\_BLOCK\_BITS\_SIZE] = {

58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6, 64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7

};

所谓的置换就是将原来的64bit数据位置打乱后重新放置，放置的规则就是上面的表格所规定的。例如原来的第58个bit，需要置换到第1个bit的位置上，等等。下面依据表格进行置换，都将按照这种规则进行。

IP\_inv\_table定义如下：

const static unsigned char ip\_inv\_table[DES\_BLOCK\_BITS\_SIZE] = {

40, 8,48,16,56,24,64,32,39, 7,47,15,55,23,63,31,

38, 6,46,14,54,22,62,30,37, 5,45,13,53,21,61,29,

36, 4,44,12,52,20,60,28,35, 3,43,11,51,19,59,27,

34, 2,42,10,50,18,58,26,33, 1,41, 9,49,17,57,25

};

step3中，Li+1无需赘述，主要的实现集中在Ri+1上。可以看到，要得到Ri+1，除了需要Li、Ri外，还需要一个Ki，该参数是由密钥扩展(1.1.2)得到的。

对于DES用户来说，无论是加密和解密，除去明文/密文外，自己肯定还要持有一个密钥，否则显然无法进行加解密的操作。这个密钥是64bit的数据。用户在进行加解密的时候，首先需要对这个密钥进行扩展，得到16个48bit的扩展密钥组合。这个扩展密钥集将在16轮循环中被依次使用，也就是上面step3中使用到的Ki。

除此之外，step3中对于还用到了一个函数：f()。该函数(1.1.3)将通过扩展置换、s盒代替、p盒置换三个步骤实现。

### 1.1.2.密钥扩展

如上所述，密钥扩展将8字节的用户手持密钥扩展得到[16][48bit]的密钥集，该密钥集将依次参与加密/解密过程。其流程图参见图2：

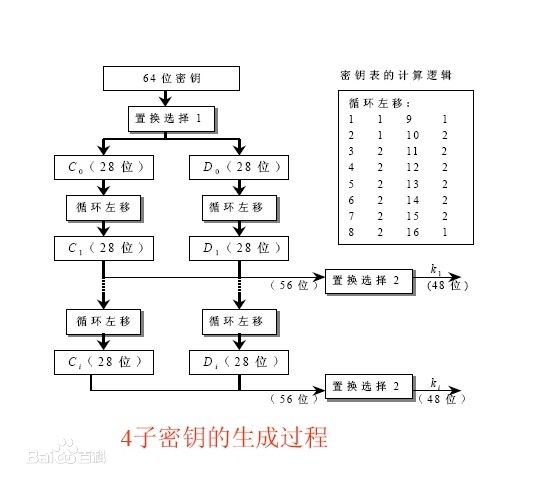


图2 密钥扩展(子密钥集合的生成)

其算法描述如下：

Input : key, 8byte data;

Output : subkey\_set, [16][ 48bit] data;

1. 对输入的8字节密钥，转换得到64bit形式；通过密钥置换表(key\_perm\_table)将其转换成56bit的数据；
2. 将step1中得到的56bit数据，分为Ci和Di两部分，长度都是28bit；
3. 对Ci和Di执行循环左移的操作，得到Ci+1和Di+1；循环左移的位数由表格rls\_table决定，16轮循环中每一轮对应各自的左移位数；
4. 得到的Ci+1和Di+1拼接成56bit的数据后，执行压缩置换；通过comp\_perm\_table(压缩置换表)压缩成为48bit的数据后保存；
5. 循环执行step3和step4，循环次数为16，最终得到16个48bit的数据集：subkey\_set[16][48bits]；

step1中用到了密钥置换表，该表的定义如下：

const static unsigned char key\_perm\_table[DES\_KEY\_PERM\_LEN] = {

57,49,41,33,25,17, 9, 1,58,50,42,34,26,18,

10, 2,59,51,43,35,27,19,11, 3,60,52,44,36,

63,55,47,39,31,23,15, 7,62,54,46,38,30,22,

14, 6,61,53,45,37,29,21,13, 5,28,20,12, 4

};

执行该操作后，将有选择性的将64bit的数据，置换成56bit的数据并投入使用。step3中使用到的rls\_table定义如下：

const static unsigned char rls\_table[DES\_ROTATE\_ROUNDS] = {

1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1

};

该表格定义了每一轮循环执行时，Ci和Di将要循环左移的位数。step4中定义的压缩置换表定义如下：

const static unsigned char comp\_perm\_table[DES\_EXTENSION\_LEN] = {

14,17,11,24, 1, 5, 3,28,15, 6,21,10,

23,19,12, 4,26, 8,16, 7,27,20,13, 2,

41,52,31,37,47,55,30,40,51,45,33,48,

44,49,39,56,34,53,46,42,50,36,29,32

};

该表格将56bit的数据，有选择性的进行压缩置换，置换后得到的数据将是48bit。

### 1.1.3.一轮DES详述

有了扩展密钥集后，将使用该密钥集和经过处理的明文/密文(1.1.1中step1)，执行16轮DES循环操作。每一轮操作的算法都是相同的，只不过输入参数是上一轮的输出，我们单独描述一轮DES的循环。

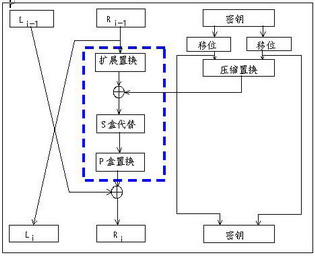


图3 一轮DES

详细描述如下：

Input : Li-1, Ri-1, Ki-1

Output : Li, Ri

1. 将Ri-1赋值给Li；
2. 通过扩展置换表ext\_perm\_table，将Ri-1从32bit扩展到48bit；
3. 将step2中得到的48bit数据，与对应的Ki-1做异或操作；
4. 将step3中得到的48bit数据，执行s盒代替，得到新的32bit数据；
5. 将step4中得到的32bit数据做p盒置换，得到32bit数据；
6. 将step5中得到的32bit数据，与32bit的Li-1进行异或操作后，结果赋值到Ri中；

扩展置换表将32bit的数据，扩展得到48bit的数据，其原理如图4所示：

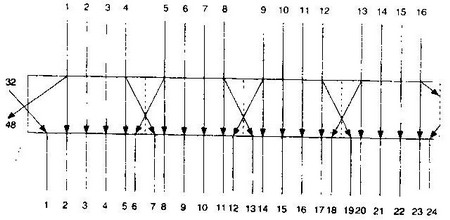


图4 扩展置换

其定义如下：

const static unsigned char ext\_perm\_table[DES\_EXTENSION\_LEN] = {

32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9,

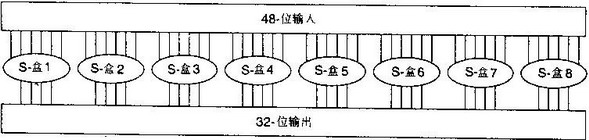
8, 9,10,11,12,13,12,13,14,15,16,17,

16,17,18,19,20,21,20,21,22,23,24,25,

24,25,26,27,28,29,28,29,30,31,32, 1

};

s盒代替是DES算法中的关键操作，因为整个算法中只有该操作不是线性的，也是算法安全性的保证。s盒代替将48bit的数据转换得到32bit的数据，其执行流程如图5所示：



详细描述如下：

Input : 48bit data;

Output : 32bit data;

1. 48bit数据分成8个6bit的数据；
2. 对每个6bit的数据，第1个bit和第6个bit组合，得到0—3之间的数字，标记为row\_num；中间4个bit组合得到0—f之间的数字，标记为column\_num；
3. 根据row\_num和column\_num，在s\_box中寻找对应的数值(1字节)，并将该值分解成4bit的形式，替换之前的6bit数据的位置；
4. 循环执行step2和step3两个操作，直到所有的8个6bit数据都替换成了4bit数据；

s\_box\_table的定义如下所示：

const static unsigned char s\_box\_table[DES\_BLOCK\_BYTES\_SIZE][DES\_BLOCK\_BITS\_SIZE] = {

{//S1

14, 4,13, 1, 2,15,11, 8, 3,10, 6,12, 5, 9, 0, 7,

0,15, 7, 4,14, 2,13, 1,10, 6,12,11, 9, 5, 3, 8,

4, 1,14, 8,13, 6, 2,11,15,12, 9, 7, 3,10, 5, 0,

15,12, 8, 2, 4, 9, 1, 7, 5,11, 3,14,10, 0, 6,13

},

{//S2

15, 1, 8,14, 6,11, 3, 4, 9, 7, 2,13,12, 0, 5,10,

3,13, 4, 7,15, 2, 8,14,12, 0, 1,10, 6, 9,11, 5,

0,14, 7,11,10, 4,13, 1, 5, 8,12, 6, 9, 3, 2,15,

13, 8,10, 1, 3,15, 4, 2,11, 6, 7,12, 0, 5,14, 9

},

{//S3

10, 0, 9,14, 6, 3,15, 5, 1,13,12, 7,11, 4, 2, 8,

13, 7, 0, 9, 3, 4, 6,10, 2, 8, 5,14,12,11,15, 1,

13, 6, 4, 9, 8,15, 3, 0,11, 1, 2,12, 5,10,14, 7,

1,10,13, 0, 6, 9, 8, 7, 4,15,14, 3,11, 5, 2,12

},

{//S4

7,13,14, 3, 0, 6, 9,10, 1, 2, 8, 5,11,12, 4,15,

13, 8,11, 5, 6,15, 0, 3, 4, 7, 2,12, 1,10,14, 9,

10, 6, 9, 0,12,11, 7,13,15, 1, 3,14, 5, 2, 8, 4,

3,15, 0, 6,10, 1,13, 8, 9, 4, 5,11,12, 7, 2,14

},

{//S5

2,12, 4, 1, 7,10,11, 6, 8, 5, 3,15,13, 0,14, 9,

14,11, 2,12, 4, 7,13, 1, 5, 0,15,10, 3, 9, 8, 6,

4, 2, 1,11,10,13, 7, 8,15, 9,12, 5, 6, 3, 0,14,

11, 8,12, 7, 1,14, 2,13, 6,15, 0, 9,10, 4, 5, 3

},

{//S6

12, 1,10,15, 9, 2, 6, 8, 0,13, 3, 4,14, 7, 5,11,

10,15, 4, 2, 7,12, 0, 5, 6, 1,13,14, 0,11, 3, 8,

9,14,15, 5, 2, 8,12, 3, 7, 0, 4,10, 1,13,11, 6,

4, 3, 2,12, 9, 5,15,10,11,14, 1, 7, 6, 0, 8,13

},

{//S7

4,11, 2,14,15, 0, 8,13, 3,12, 9, 7, 5,10, 6, 1,

13, 0,11, 7, 4, 0, 1,10,14, 3, 5,12, 2,15, 8, 6,

1, 4,11,13,12, 3, 7,14,10,15, 6, 8, 0, 5, 9, 2,

6,11,13, 8, 1, 4,10, 7, 9, 5, 0,15,14, 2, 3,12

},

{//S8

13, 2, 8, 4, 6,15,11, 1,10, 9, 3,14, 5, 0,12, 7,

1,15,13, 8,10, 3, 7, 4,12, 5, 6,11, 0,14, 9, 2,

7,11, 4, 1, 9,12,14, 2, 0, 6,10,13,15, 3, 5, 8,

2, 1,14, 7, 4,10, 8,13,15,12, 9, 0, 3, 5, 6,11

}

};

最后执行p\_box的置换操作，将32位数据进行再一次的置换，该表格的定义如下：

const static unsigned char p\_box\_table[DES\_HALF\_PART\_LEN] = {

16, 7,20,21,29,12,28,17, 1,15,23,26, 5,18,31,10,

2, 8,24,14,32,27, 3, 9,19,13,30, 6,22,11, 4,25

};

### 1.1.4.加密、解密

解密与加密的流程大致相同，不同之处是：将subkey\_set按照15->0的顺序参与循环运算；

### 1.1.5.Key的长度

DES算法，允许key的长度不是8，当key的长度不是8的时候:

* Key的长度小于8，补0在最后，补齐到8bytes；
* Key的长度大于8，只使用前8个字节作为key；

### 1.1.6.局限性

由于DES算法严格要求输入数据的长度，是8字节的整数倍，因此限制了算法的使用。经过网上查看资料，有一些算法可以解决这个问题，但是会使得算法本身更加复杂，似乎得不偿失，不如直接在调用者调用的时候，就进行填充来的容易些。

填充算法，简单来说，可以在最后一个不足8bytes的块，补充0x00或者0xff，记录补充的字节数后，解密后将这些填充位忽略；但该操作由应用来做更加合适，可适当选择使用的填充值，算法自身严格要求8字节的倍数作为输入长度；

## 1.2.DES3算法的ECB模式

由于DES算法提出年代较早，安全性随着计算机的发展受到了很大挑战，所以3des算法被提出，通过3轮密切相关的des运算，提高安全性。

算法原理描述如下：

3DES Algo.

input : key 16byte or 24byte, input, 8byte;

output : output, 8byte;

1. 对key按照8个字节为一个单位，执行密钥扩展，得到3个(或只有2个)subkey\_set；
2. 使用subkey\_set0/subkey\_set2对input执行加密/解密操作，得到ret0；
3. 使用subkey\_set1对ret0执行解密/加密操作，得到ret1；
4. 使用subkey\_set2/subkey\_set0对ret1执行加密/解密操作，得到ret2；
5. ret2即为所需要的结果；

对该算法有如下需要注意的地方：

* 执行3DES加密操作时，step2-4的顺序是：加密、解密、加密；执行解密时顺序是：解密、加密、解密；
* 使用密钥的顺序也不同，加密操作是：subkey\_set0、1、2；解密操作时，subkey\_set2、1、0；
* 使用3DES算法时，密钥的有效长度可以是16byte和24byte两种。如果是16byte，将生成两个密钥子集set1和set2，set1将同样被当做set3使用；而24byte则使用3个完全不同的set；

## 1.3.DES算法的CBC模式

相比于ECB模式(电子密码本)模式，CBC模式(密码分组链接模式)密级更高，但也更加复杂；原理如下:

加密：

1. 将输入数据，按照8bytes一组，分为D1,D2,…,Dn；
2. 第一组数据D1，先与初始化向量Iv异或，再进行DES加密，得到第一组明文C1；
3. D2与C1异或后，在进行DES加密，得到C2；
4. 以此类推，直至Dn；
5. 将C1,C2,…,Cn连接起来，即为加密结果；

解密：

1. 将密文数据，按照8bytes一组，分为C1,C2,…,Cn；
2. 将C1先使用DES解密，再与初始化向量Iv执行异或，得到D1；
3. 将C2先解密，再与C1执行异或后得到D2；
4. 以此类推，直至Dn；
5. 将D1,D2,…,Dn连接起来就是明文；

两种模式的差别是：

ECB模式更加简单、有利于并行计算、无差不会被传递；但也不能隐藏明文的模式，可对明文产生攻击；

CBC模式不容易主动攻击，安全性更高；但不利于并行计算，影响效率，误差会传递，需要初始化向量Iv；

Iv的选择，一般是随机生成的；每一次加解密使用相同的Iv，下一次加解密时更换该值；这样就可以保证即便是相同的明文和密钥，得到的密文也是不同的了；密级也因此更高一些。

# 2.AES算法

AES算法中，一个字节占用8bit，一个字是4个字节，占用32bit的空间。下面的所有描述都是基于这个基本概念的。

## 2.1.AES算法的ECB模式实现

AES算法属于分组密码算法，它的输入分组、输出分组和加解密过程中产生的中间分组都是128bit的数据。密钥的长度可以由如下3个选择：128bit，192bit和256bit；也可分别描述为4、6、8个字。我们用Nk来表示密钥的字数。

一般来讲，用Nr表示对一个数据分组加密的轮数，Nr与Nk有一一对应的关系，如下表2.1所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AES类型 | 密钥长度 | 分组大小 | 加密轮数 | 扩展密钥字节数 |
| AES-128 | 4字 | 4字 | 10 | 176 |
| AES-192 | 6字 | 4字 | 12 | 208 |
| AES-256 | 8字 | 4字 | 14 | 240 |

表2.1 密钥长度及轮数对应关系

AES算法，ECB模式下的加解密原理可通过图2.1概要描述：



图2.1 AES算法概述

显然，每一轮都需要一个和输入分组具有相同长度的扩展密钥Ke的参与。由于外部输入的加密密钥K长度有限，所以在AES中同样要使用密钥扩展程序(key\_expansion)把外部密钥K扩展成更长的数组，以完成各轮的操作。

# 3.SHA1散列算法

Algorithm SHA1

Function : 计算给定字符串的SHA1散列值；

Input : 字符集合(一般以字符串的形式体现)str\_input；

Output : 该字符集的SHA1散列值digest，确定为160bit，也就是20byte；

1. 补位、补长度(详见3.1)，使str\_input🡪exp\_input；
2. 将exp\_input按照指定规则(详见3.2)分块，得到M0，M1，……，Mn-1；
3. 初始化数据：i = 0；H[5]、K[4]分别进行初始化(详见3.3)；
4. 使用Mi对H[5]进行刷新(详见3.4)，得到新一组H[5]的值；
5. 如果I == n – 1 : 执行step 6；

否则：执行step 4；

1. 将H[5]分解成20个byte的数据，赋值到digest中(详见3.5)，就是输入数据的散列值；

## 3.1.补位、补长度

### 3.1.1.补位

所谓补位是指输入数据必须进行补位，以使其长度在对512bit取模以后的余数是448bit。也就是说，（补位后的消息长度）%512 = 448。即使原始长度已经满足对512bit取模后余数是448bit，补位也必须要进行。

补位是这样进行的：先补一个bit的1，然后再补0，直到长度满足对512取模后余数是448。总而言之，补位是至少补一位，最多补512位。还是以前面的“abc”为例显示补位的过程。

原始信息： 01100001 01100010 01100011

补位第一步：01100001 01100010 01100011 1 //首先补一个“1”

补位第二步：01100001 01100010 01100011 10…..0  //然后补423个“0”

我们可以把最后补位完成后的数据用16进制写成下面的样子

61626380 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000

### 3.1.2.补长度

执行过上面的补位操作后，原始数据已经满足了长度对512bit取模后的余数是448bit，还需要在最后再补充64bit的长度值，使原始数据的长度满足时512bit的长度。

这里的长度值是原始数据的长度，不包括补位和补长度的值，单位是bit。

通常用一个64位的数据来表示原始消息的长度。如果消息长度不大于2^64，那么第一个字就是0。在进行了补长度的操作以后，整个消息就变成下面这样了（16进制格式）

61626380 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000018

## 3.2.分块

经过3.1中的操作，输入数据input已经被补充成为了exp\_input，且exp\_input的长度显然是512bit(64byte)的整数倍。

这里所谓的分块就是将exp\_input按照512bit(64byte)为基本单位进行划分，得到N个分块，表示成为：M[0], ……, M[N-1]。显然任何一个M[i]的长度都是64个byte。

## 3.3.初始化数据

进行SHA1的计算之前，还需要进行一些初始化操作，主要指的是：

### 3.3.1.一系列常量

在后面3.4中对H进行刷新的时候，要进行80轮相关运算，运算过程中需要使用到一下的K值。针对不同的轮数，K的值不同，t代表了轮数。

Kt = 0X5A827999 (0 <= t <= 19)

Kt = 0X6ED9EBA1 (20 <= t <= 39)

Kt = 0X8F1BBCDC (40 <= t <= 59)

Kt = 0XCA62C1D6 (60 <= t <= 79)

### 3.3.2.使用到的函数

同样在3.4中的80轮操作中，需要使用到一些定义好的函数，这些函数针对不同的轮数，也有不同的定义：

ft(B,C,D) = (B and C) or ( (not B) and D ) ( 0 <= t <= 19)

ft(B,C,D) = B xor C xor D              (20 <= t <= 39)

ft(B,C,D) = (B and C) or (B and D) or (C and D) (40 <= t <= 59)

ft(B,C,D) = B xor C xor D                     (60 <= t <= 79).

### 3.3.3.变量初值

由上可知，3.4中主要对H[5]通过某种方法不断改变其值，最终H[5]将构造出结果。

但显然，H[5]需要有一个初值，这里算法规定如下：

H[0] = 0X67452301;

H[1] = 0XEFCDAB89;

H[2] = 0X98BADCFE;

H[3] = 0X10325476;

H[4] = 0XC3D2E1F0;

## 3.4.刷新H[5](算法核心)

对每一个M[i]，都需要执行本部分操作，并借此刷新H[5]的值。直到M[N-1]执行完，H[5]将作为最终值，稍作格式变化后输出。这里描述针对每一个M[i]的操作。

这里引入SHA1算法中关于字的定义：**一个字**指的是占用4个字节的基本单位，也就是一个word占用了32个bit。

### 3.4.1.循环左移函数RSL

给定数字number，给定移位数量shift\_bit，循环左移函数可如下实现：

#define RSL(number, shift\_bit) ((number << shift\_bit) | (number >> (32 – shift\_bit)))

这里注意，在C语言中，如果是有符号数，进行右移时编译器会将符号位考虑进来，写程序时应特别注意这点。

### 3.4.2.W[80]的生成

1. 将M[i]中64个byte的数据，每四个组成一个字后，按照顺序赋值到W[0]—W[15]中；例如，M[i] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, …} 🡪 W[0]=0x01020304, W[1] = 0x05060708,….；
2. 对于W[16]—W[79]的生成，遵循如下规则：W[i] = RSL( (W[i-3] xor W[i-8] xor W[i-14] xor W[i-16]), 1 )；

### 3.4.3.H[5]的刷新

取临时变量：A = H[0], B = H[1], C= H[2], D = H[3], E = H[4]。

如下操作，循环执行80次：

temp = RSL(A, 5) + ft(B, C, D) + E + W[i] + K[t];

E = D; D = C; C = RSL(B, 30); B = A; A = temp;

第一条语句中ft()函数、K[t]常量参见3.3.1和3.3.2，W[i]参见3.4.2，RSL函数参见3.4.1,。在执行完上述80次循环后，执行如下操作：

H0 = H0 + A

H1 = H1 + B,

H2 = H2 + C,

H3 = H3 + D,

H4 = H4 + E

这样H[5]就完成了针对M[i]的刷新操作。

## 3.5.构造结果

得到的H[5]是5个字，SHA1算法规定的输出时20个byte，所以需要将每个字变换成4个byte的形式返回。

此时算法结束。

# 4.MD5散列算法

MD5算法全程是Message Digest Algorithm 5，中文名是消息摘要算法第五版，是一种散列算法，用来提供信息的完整性保护。

## 4.1.算法步骤

MD5算法以512位为一个分组处理输入的信息，每一个分组又被划分为16个32个子分组，经过一系列数学运算后，输出四个32位的分组数据，这4个32位的分组数据级联成128bit的散列。

### 4.1.1.填充

如果输入信息的长度(单位为bit！)对512求余的结果不等于448，就需要填充，使其长度对512求余结果为448。填充方法是一个bit的1和若干个bit的0.填充完成后，信息的长度满足：N\*512+448(bit)；

### 4.1.2.记录长度

用64bit的数据记录**填充之前**的数据长度，这64bit的数据紧随4.1.1中填充之后的数据，这样信息的长度就成为了：（N+1）\*512bit；

要注意的是，这里添加长度时的顺序一定是小端的，例如“abc”这个字符串，三个字节的长度(\0不计算在长度内)，换算成bit就是24，折合成十六进制就是0x18。按照正常的大端，这里要添加的就会是0x0000 0000 0000 0018，但是在MD5算法中要求的一定是：0x1800 0000 0000 0000的形式；

### 4.1.3.装入标准幻数

标准幻数(物理顺序)是：A=(0x01234567), B=(0x89ABCDEF), C=(0Xfedcba98), D=(0x76543210);

但是在程序中要注意，能够使用的应该是字节序相反的变量定义：A=(0x67452301), B=(0xefcdab89), C=(0x98badcfe), D=(0x10325476);

### 4.1.4.四轮循环运算

首先做如下一些基本函数定义：

F(X, Y, Z) = (X & Y) | ((~X) & Z);

G(X, Y, Z) = (X & Z) | (Y & (~Z));

H(X, Y, Z) = (X ^ Y ^ Z);

I(X, Y, Z) = Y ^ (X | (~Z));

这是四个非线性函数，这四个函数说明，如果X、Y和Z的对应位是独立和均匀的，那么结果的每一位也是独立和均匀的；

假设M[j]表示消息的第j个子分组，<<<表示循环左移，那么：

FF(a, b, c, d, M[j], s, ti)表示：a = b + ( (a + F(b, c, d) + M[j] + ti) <<< s );

GG(a, b, c, d, M[j], s, ti)表示：a = b + ( (a + G(b, c, d) + M[j] + ti) <<< s );

HH(a, b, c, d, M[j], s, ti)表示：a = b + ( (a + H(b, c, d) + M[j] + ti) <<< s );

II(a, b, c, d, M[j], s, ti)表示：a = b + ( (a + I(b, c, d) + M[j] + ti) <<< s );

1. for(i = 0; i < N+1; i++) //数据长度为(N+1)\*512bits，对每一个512bits的分组执行如下的操作
2. 将512bits分为16个分组，每个分组的size为32bits，定义这些分组为M[0]—M[15]，注意这些分组组合成32bit的数字的顺序也是小端的！例如，32bit的char是[0x12, 0x34, 0x56, 0x78];那么M[x]=0x78563412,而不是0x12345678！
3. 对这16个分组进行判断：

a = A; b = B; c = C; d = D;

//传说中的对M[j]的第一轮循环

        FF(a,b,c,d,M[0],7,0xd76aa478);

        FF(d,a,b,c,M[1],12,0xe8c7b756);

        FF(c,d,a,b,M[2],17,0x242070db);

        FF(b,c,d,a,M[3],22,0xc1bdceee);

        FF(a,b,c,d,M[4],7,0xf57c0faf);

        FF(d,a,b,c,M[5],12,0x4787c62a);

        FF(c,d,a,b,M[6],17,0xa8304613);

        FF(b,c,d,a,M[7],22,0xfd469501) ;

        FF(a,b,c,d,M[8],7,0x698098d8) ;

        FF(d,a,b,c,M[9],12,0x8b44f7af) ;

        FF(c,d,a,b,M[10],17,0xffff5bb1) ;

        FF(b,c,d,a,M[11],22,0x895cd7be) ;

        FF(a,b,c,d,M[12],7,0x6b901122) ;

        FF(d,a,b,c,M[13],12,0xfd987193) ;

        FF(c,d,a,b,M[14],17,0xa679438e) ;

        FF(b,c,d,a,M[15],22,0x49b40821);

        //传说中对M[j]的第二轮循环

        GG(a,b,c,d,M[1],5,0xf61e2562);

        GG(d,a,b,c,M[6],9,0xc040b340);

        GG(c,d,a,b,M[11],14,0x265e5a51);

        GG(b,c,d,a,M[0],20,0xe9b6c7aa) ;

        GG(a,b,c,d,M[5],5,0xd62f105d) ;

        GG(d,a,b,c,M[10],9,0x02441453) ;

        GG(c,d,a,b,M[15],14,0xd8a1e681);

        GG(b,c,d,a,M[4],20,0xe7d3fbc8) ;

        GG(a,b,c,d,M[9],5,0x21e1cde6) ;

        GG(d,a,b,c,M[14],9,0xc33707d6) ;

        GG(c,d,a,b,M[3],14,0xf4d50d87) ;

        GG(b,c,d,a,M[8],20,0x455a14ed);

        GG(a,b,c,d,M[13],5,0xa9e3e905);

        GG(d,a,b,c,M[2],9,0xfcefa3f8) ;

        GG(c,d,a,b,M[7],14,0x676f02d9) ;

        GG(b,c,d,a,M[12],20,0x8d2a4c8a);

        //传说中对M[j]的第三轮循环

        HH(a,b,c,d,M[5],4,0xfffa3942);

        HH(d,a,b,c,M[8],11,0x8771f681);

        HH(c,d,a,b,M[11],16,0x6d9d6122);

        HH(b,c,d,a,M[14],23,0xfde5380c) ;

        HH(a,b,c,d,M[1],4,0xa4beea44) ;

        HH(d,a,b,c,M[4],11,0x4bdecfa9) ;

        HH(c,d,a,b,M[7],16,0xf6bb4b60) ;

        HH(b,c,d,a,M[10],23,0xbebfbc70);

        HH(a,b,c,d,M[13],4,0x289b7ec6);

        HH(d,a,b,c,M[0],11,0xeaa127fa);

        HH(c,d,a,b,M[3],16,0xd4ef3085);

        HH(b,c,d,a,M[6],23,0x04881d05);

        HH(a,b,c,d,M[9],4,0xd9d4d039);

        HH(d,a,b,c,M[12],11,0xe6db99e5);

        HH(c,d,a,b,M[15],16,0x1fa27cf8) ;

        HH(b,c,d,a,M[2],23,0xc4ac5665);

        //传说中对M[j]的第四轮循环

        II(a,b,c,d,M[0],6,0xf4292244) ;

        II(d,a,b,c,M[7],10,0x432aff97) ;

        II(c,d,a,b,M[14],15,0xab9423a7);

        II(b,c,d,a,M[5],21,0xfc93a039) ;

        II(a,b,c,d,M[12],6,0x655b59c3) ;

        II(d,a,b,c,M[3],10,0x8f0ccc92) ;

        II(c,d,a,b,M[10],15,0xffeff47d);

        II(b,c,d,a,M[1],21,0x85845dd1) ;

        II(a,b,c,d,M[8],6,0x6fa87e4f) ;

        II(d,a,b,c,M[15],10,0xfe2ce6e0);

        II(c,d,a,b,M[6],15,0xa3014314) ;

        II(b,c,d,a,M[13],21,0x4e0811a1);

        II(a,b,c,d,M[4],6,0xf7537e82) ;

        II(d,a,b,c,M[11],10,0xbd3af235);

        II(c,d,a,b,M[2],15,0x2ad7d2bb);

        II(b,c,d,a,M[9],21,0xeb86d391);

        A += a;

        B += b;

        C += c;

        D += d;

1. 对所有子分组执行如上的循环操作；

### 4.1.5.最终结果

处理完所有512分组的内容后，得到一组新的A B C D的值，将这些值按照ABCD的级联顺序排序，然后输出即为最终的MD5值。

需要注意的是，MD5是小端序列，所以输出时要做一次转换，例如A有32bit，分为A1A2A3A4，那么输出的就应该是A4A3A2A1。

### 4.1.6.一些典型的MD5值

md5(“”) = d41d8cd98f00b204e9800998ecf8427e

MD5 ("a") = 0cc175b9c0f1b6a831c399e269772661

MD5 (“ab”) = 187ef4436122d1cc2f40dc2b92f0eba0

MD5 ("abc") = 900150983cd24fb0d6963f7d28e17f72

MD5 (“abcde”) = ab56b4d92b40713acc5af89985d4b786

MD5 ("message digest") = f96b697d7cb7938d525a2f31aaf161d0

MD5 ("abcdefghijklmnopqrstuvwxyz") = c3fcd3d76192e4007dfb496cca67e13b

MD5 ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz") = f29939a25efabaef3b87e2cbfe641315

# 5.RC4算法

RC4算法是一种在电子信息领域加密的技术手段，常用于无线通信领域。和DES算法一样，它是一种对称加密算法，也就是说使用的密钥为私钥。

不同于DES的是，RC4不是对明文进行分组处理，而是以字节流的形式加密每一个字节，解密的时候也是依次对密文中的每一个字节执行解密。

算法流程如下：

## 5.1.关键变量

### 5.1.1.密钥流

RC4算法的关键是根据明文和密钥生成相应的密钥流，密钥流的长度和明文的长度是对应的，也就是说明文的长度是500字节，那么密钥流也是500字节。当然，加密生成的密文也是500字节，因为密文第i字节=明文第i字节^密钥流第i字节；

### 5.1.2.状态向量S

长度为256，S[0],S[1].....S[255]。每个单元都是一个字节，算法运行的任何时候，S都包括0-255的8比特数的排列组合，只不过值的位置发生了变换；

### 5.1.3.临时向量T

长度也为256，每个单元也是一个字节。如果密钥的长度是256字节，就直接把密钥的值赋给T，否则，轮转地将密钥的每个字节赋给T；

### 5.1.4.密钥K

长度为1-256字节，注意密钥的长度 keylen 与明文长度、密钥流的长度没有必然关系，通常密钥的长度取值为16字节（128比特）。

## 5.2.算法原理

### 5.2.1.初始化S和T

for i=0 to 255 do

S[i] =i;

T[i]=K[ I mod keylen ];

### 5.2.2.初始排列S

for i=0 to 255 do

j= ( j+S[i]+T[i])mod256;

swap(S[i],S[j]);

### 5.2.3.产生密钥流

for r=0 to len do //r为明文长度，r字节

i=(i+1) mod 256;

j=(j+S[i])mod 256;

swap(S[i],S[j]);

t=(S[i]+S[j])mod 256;

k[r]=S[t];

### 5.2.4.生成密文/明文

密文第i字节=明文第i字节^密钥流第i字节；

或者：明文第i字节=密文第i字节^密钥流第i字节；

# 6.BASE64算法

很多场合下，特殊字符的存在不被允许，例如邮件只允许可见字符的传送，但很多ASCII码的控制字符并不是可见字符，就不能通过邮件传送了。BASE64就是一种转换算法，将所有字符都转换为可见字符。

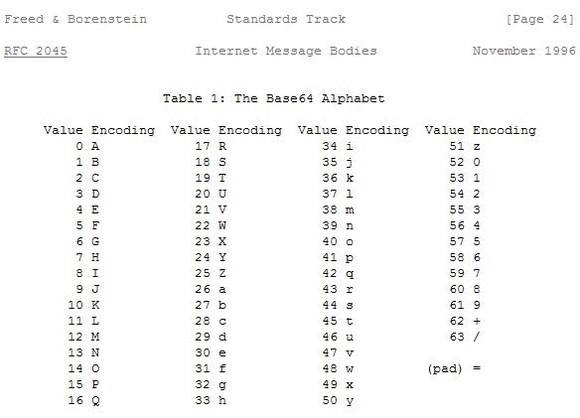
之所以被取名为base64，是因为其加密后的密文只会由A-Z, a-z, 0-9和+ / 这64个可见字符组成，不会出现其他字符；

## 6.1.原理

查阅网上存在的几种原理说明文档后，发现如下的这种通俗的举例式方法更容易被人理解，引用之：

1. 假设有3个字符要做base64加密，首先将其转换成二进制，那么此时得到一个24bits的格式；
2. Base64只支持64个有效字符，二进制的话就是26个，也就是6个bit；所以24个bit的话，就需要4 \* 6bit；因此将这个24bits的数据分成4份，每份6个bit；
3. 对每6个bits的数据，前面增加00后填充为8bit的数据，这就是一个字节的数据了，得到这个数据的值；
4. 按照下图所示的Key-value的字典，将这个数值转换为base64支持的字符；
5. 得到了4个字符，就是BASE64加密后的数据了。

下图为标准协议(RFC2045)规定的Key-value的字典：



按照上面的流程，可以轻松处理3\*N个字符的base64加密；但是并不总是需要对3N个字节的数据进行处理，针对3N+1和3N+2个字节的数据的加密，处理方式如下所述：

* BASE64规定的处理原则是：不足之处使用”=”补齐；
* 比如，对A这个字符进行处理:
  + 0xA=(01000001)2;
  + 转换为两个base64支持的字符就是：00010000, 00010000；
  + 查找map表，这两个的value是Q；到此得到了QQ；
  + 但是还不满足BASE64的密文长度是4的整数倍，缺少2个字节的数据，按照协议规定，此时补充=，因此最终结果就是:QQ==;

## 6.2.变种

所谓的BASE64的变种，大概有两种方式：

1. 对原始内容进行修改，按照自行约定的规则，在原始字符串的指定位置增加一些信息，解密后忽略这些信息即可；
2. 对key-value的map进行调整；

排序和查找

# 1.基本排序算法

按照是否使用外存，排序算法可分为内部排序和外部排序两种。

内部排序又分为如下五种：

* 插入排序
  + 直接插入排序
  + 希尔排序
* 选择排序
  + 简单选择排序
  + 堆排序
* 交换排序
  + 冒泡排序
  + 快速排序
* 归并排序
* 基数排序

## 1.1.直接插入排序

### 1.1.1.基本思想

在要排序的一组数中，假设前面(n-1)[n>=2]个数已经是排好了顺序的，现在要把第n个数字插到前面的有序数组中，使得这n个数也是排好顺序的。如此反复循环，直到全部都排好了顺序。

### 1.1.2.实例

初始状态：57 68 59 52

处理：

1. 57 68 59 52 由于57小于68，不处理
2. 57 59 68 52 将59插入到有序数列中，保持有序状态
3. 52 57 59 68 将52插入有序数列

### 1.1.3.复杂度分析

时间复杂度应为O(n2)；在极端情况(数组本来就是有序的)下，可以达到O(n)的最好结果。

## 1.2.希尔排序(最小增量排序)

### 1.2.1.基本思想

希尔排序又称为缩小增量排序，是直接插入排序的改进和优化。

其基本思想是先将序列分成若干子序列分别进行直接插入排序，待整个序列基本有序时再对整体进行一次直接插入排序。

算法特点是子序列不是简单的“逐段分割”，而是相邻某个“增量”的记录构成一个子序列。一般增量序列的选择直接使用上次的一半，例如整个序列长度为N，则第一次的增量序列可以选择为N/2。并注意，最后一个增量必须是1，这也是结束条件。

### 1.2.2.实例

Org, n = 10: 57 68 59 52 73 61 29 84 65 32

Round 0, d = 5: 57---------------61

68--------------29

59-------------84

52---------------65

73---------------32

57 29 59 52 32 61 68 84 65 73

Round 1, d = 2: 57----59----32----68-----65

29----52---61----84-----73

32 29 57 52 59 61 65 73 68 84

Round 2, d = 1(结束条件): 直接插入排序

## 1.3.简单选择排序

### 1.3.1.基本思想

在要排序的一组数中，选出最小的一个数与第一个位置的数交换；然后在剩下的数中再找出最小的与第二个位置的数交换，如此循环到倒数第二个数和最后一个数比较为止。

### 1.3.2.实例

Org: 57 68 59 52

Round 0 : 52 68 59 57

Round 1: 52 57 68 59

Round 2: 52 57 59 68

## 1.4.堆排序

### 1.4.1.基本思想

当且仅当满足以下条件时，关键字序列K1—Kn可以称为堆：树中任一非叶子结点的关键字均不大于(不小于)其左右孩子节点的关键字。

用大顶堆排序的基本思想：

1. 先将输入的数字序列初始化成为大顶堆；
2. 将堆顶(也肯定是序列里最大的关键字)R[1]和序列最后一个关键字R[n]交换，由此得到新的无需区R[1]..R[n]，而R[n]显然已经是有序区；
3. 调整R[1]..R[n]为有序堆，再将其堆顶与最末的元素交换，循环执行，直至所有元素都执行过排序为止。

## 1.5.冒泡排序

### 1.5.1.基本思想

冒泡算法的运作如下(**从后往前**)：

1. 比较相邻元素，如果第一个比第二个大，就交换他们；
2. 对每一对相邻元素做如此操作，从开始第一对到结尾最后一对。执行完之后，最后一个元素应该是最大的数；
3. 针对所有的元素重复以上操作，除了最后一个；
4. 持续每次对越来越少的元素重复之前步骤，直到没有任何一对数字要比较；

### 1.5.2.实例

Org: 57 68 59 52

Round 0 : 52 57 59 68

Round 1: 52 57 59 68

Round 2: 52 57 59 68

## 1.6.快速排序

### 1.6.1.基本思想

快速排序采用了分治的策略解决问题，分治法的核心思想是：将问题分解成若干个规模更小但结构与原问题相似的子问题。递归的解决这些子问题，最终将子问题的解组合成为原问题的解。

设当前待排序的无序区为R[low…high]，利用分治法可以将快速排序的基本思想描述为：

1. 在数列中随机选取一个记录作为基准pivot(一般直接选取第一个)，以此基准将当前序列划分为左右两个较小的子区间Rl和Rr，并使左区间中记录都小于pivot，右区间中值都大于pivot。此时pivot已经处在了正确的位置上，所以无需再次参加后续的排序。
2. 通过递归方法对左区间和右区间再次进行划分；
3. 递归返回时，序列已经成为有序，排序结束。

这里注意两点：

* 一般pivot会直接选择第一个记录，但随机选取的方法能够更好的照顾到算法对所有类型数列的时间消耗，对各种类型数列可以提高排序的效率；不论pivot选择哪里，总会有特殊情况导致算法耗时达到O(n2)的级别。
* 由于普遍认为20以内的数组排序，直接插入排序的效率是最高的，所以当数组长度小于20时，可以直接采用直接插入排序进行排序，而不是再次执行递归。

## 1.7.归并排序

归并排序是将两个(或以上)有序表合并成一个新的有序表，即把待排序的序列分成若干个有序的子序列，再把有序的子序列合并成整体有序的排序过程。

归并排序是建立在归并操作基础上的算法，该算法是采用分治法的一个典型应用。

### 1.7.1.归并操作实例

归并操作指的是将两个有序序列合并成一个有序序列的方法，例如有数列{6, 202, 100, 31, 38, 8, 1}:

|  |  |  |
| --- | --- | --- |
| 归并次数 | 结果 | 比较次数 |
| 第一次归并 | {6, 202}, {31, 100}, {8, 38}, {1} | 3 |
| 第二次归并 | {6, 31, 100, 202}， {1, 8, 38} | 4 |
| 第三次归并 | {1, 6, 8, 31, 38, 100, 202} | 5 |

由上表可以看出，总的比较次数是：3 + 4 + 5 = 12.

### 1.7.2.归并操作算法描述

1. 申请空间，申请的大小应是两个已经有序的序列的长度之和，用来存放合并后的序列；
2. 设定两个指针，最初位置分别是两个已经排序序列的起始位置；
3. 比较两个指针指向的元素，选择相对小的放入合并后的空间，指针顺势移动到下一个元素；
4. 重复步骤3，直至某一个指针超出序列尾；
5. 将另一个序列剩下的所有元素直接复制到合并空间中；

## 1.8.基数排序

## 1.9.八种排序算法的总结

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 类别 | 排序方法 | 时间复杂度 | | | 空间复杂度 | 稳定性 |
| 平均情况 | 最好情况 | 最坏情况 | 辅助存储 |
| 插入排序 | 直接插入 | O(n2) | O(n) | O(n2) | O(1) | 稳定 |
| 希尔排序 | O(n1.3) | O(n) | O(n2) | O(1) | 不稳定 |
| 选择排序 | 直接选择 | O(n2) | O(n2) | O(n2) | O(1) | 不稳定 |
| 堆排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(1) | 不稳定 |
| 交换排序 | 冒泡排序 | O(n2) | O(n) | O(n2) | O(1) | 稳定 |
| 快速排序 | O(nlog2n) | O(nlog2n) | O(n2) |  | 不稳定 |
| 归并排序 | | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(n) | 稳定 |
| 基数排序 | | O(d(r+n)) | O(d(rd+n)) | O(d(r+n)) | O(rd + n) | 稳定 |
| 基数排序中，r代表关键字的基数，d代表长度，n代表关键字的个数 | | | | | | |

## 1.10.效率比较

通过耗时测试，得出如下结果：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数组类型 | 数组长度 | 循环次数 | 排序算法 | 平均耗时(ms) |
| 顺序数组 | 5000 | 50 | 直接插入排序 | 0.003036 |
| 希尔排序 | 0.073832 |
| 简单选择排序 | 3.630109 |
| 堆排序 | 0.186747 |
| 冒泡排序 | 3.508217 |
| 快速排序 | 2.766099 |
| 归并排序 |  |
| 10000 | 10 | 直接插入排序 | 0.001236 |
| 希尔排序 | 0.055970 |
| 简单选择排序 | 2.896902 |
| 堆排序 | 0.090035 |
| 冒泡排序 | 2.286321 |
| 快速排序 | 2.277021 |
| 归并排序 |  |
| 反序数组 | 5000 | 50 | 直接插入排序 | 9.442874 |
| 希尔排序 | 17.744187 |
| 简单选择排序 | 3.761798 |
| 堆排序 | 0.177772 |
| 冒泡排序 | 9.382973 |
| 快速排序 | 2.726488 |
| 归并排序 |  |
| 10000 | 10 | 直接插入排序 | 7.351613 |
| 希尔排序 | 14.342757 |
| 简单选择排序 | 2.956789 |
| 堆排序 | 0.092651 |
| 冒泡排序 | 7.383245 |
| 快速排序 | 2.193542 |
| 归并排序 |  |
| 随机数组 | 5000 | 50 | 直接插入排序 | 4.614691 |
| 希尔排序 | 9.143959 |
| 简单选择排序 | 3.521202 |
| 堆排序 | 0.190860 |
| 冒泡排序 | 8.903948 |
| 快速排序 | 0.130144 |
| 归并排序 |  |
| 10000 | 10 | 直接插入排序 | 3.755292 |
| 希尔排序 | 7.189396 |
| 简单选择排序 | 3.819253 |
| 堆排序 | 0.108193 |
| 冒泡排序 | 7.143905 |
| 快速排序 | 0.069189 |
| 归并排序 |  |

测试平台：virtual box上装的ubuntu 12.04虚拟机，虚拟出768M内存；

# 2.基本查找算法

## 2.1.顺序查找

从表的一端开始，顺序的扫描表，如果找到要查找的值就结束，否则查找失败。

这种算法查找的效率低下，但适用于线性表的顺序存储结构和链式存储结构的查找。

## 2.2.二分查找

1. Mid = (low + high) / 2;
2. 比较待查值a与mid指向节点的值，如果相等则查找成功，返回；如果a<[mid]，则high= mid-1；否则low=mid+1；
3. 重复执行step1和step2。如果low>high时查找失败。

**该算法要求要查找的序列式有序序列！**

虽然二分查找的效率较高，但由于表本身要求按照关键字排序，且排序本身就耗时较大，所以二分法更加适合顺序存储结构，且是一经建立就很少改动的结构。

## 2.3.分块查找

其他相关算法知识

# 1.最大公约数和最小公倍数

辗转相除法，又称欧几里得算法(Euclidean algorithm)，是求两个正整数最大公约数的算法。

辗转相除法基于以下原理：两个整数的最大公约数等于其中较小的数和两数的相除余数的最大公约数。例如，252和105的最大公约数是21，而252/105=2余42，所以105和42的最大公约数也是21.在这个过程中，两个数中的较大的一直在减小，当余数变为0的时候，除数就是两个数的最大公约数。

最小公倍数和最大公约数的关系是：两数相乘的结果，除以这两个数的最大公约数，结果就是最小公倍数。

最大公约数：greatest common divisor

最小公倍数：least common multiple;

# 2.杨辉三角

所谓杨辉三角，就是类似于如下的数字排列组合：

1

1. 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

……..

编写程序时应注意到的规律：

1. 第一列和对角线元素都是1；
2. 其余数字都等于本列上一行数据与前一列上一行数字之和；

# 3.素数判断

素数(质数)指的是能被1和自身整除的数字。

素数判断只需判断[2, sqrt(n)]之间是否存在数字能够被n整除即可，无需判断sqrt(n)—n-1之间的其他数字，原因是如果有一个数字m能够被n整除，那么n/m和m之间一定有一个是存在于[2, sqrt(n)]之间的。

这种判断适用于小数，对于大数应有更加有效率的算法来进行判断。

字符串匹配算法(moUtils\_Search)

字符串匹配算法，指的是从一个字符串(主串)中，查找其是否包含另外一个字符串(模式串)；

该匹配原理可以扩展，并不一定局限于字符串(“\0”结尾)，而是所有char类型的数组，都可以使用该算法进行匹配；

# 1.BF算法

Brute Force算法(暴力破解算法)，时间复杂度为O(n\*m)，其中n为主串的长度，m为模式串的长度；

从主串的第一个字节开始进行匹配，如果匹配失败了跳转到下一个字符继续进行匹配；

# 2.KMP算法

KMP算法是三位发明者的名字简称组成的，分别是：Knuth，Morris，Pratt；

## 2.1.部分匹配表

KMP算法的关键，是一张部分匹配表(Partial Match Table)；

### 2.1.1.前缀、后缀

两个概念需要明确：

* 前缀：除了最后一个字符外，一个字符串的全部的头部组合；
* 后缀：除了第一个字符外，一个字符串的全部的尾部组合；

例如，字符串bread，前缀就是：b, br, bre, brea；后缀就是：read, ead, ad, d；

### 2.1.2.部分匹配值

部分匹配值就是这个字符串的“前缀”和“后缀”的**最长的**共有元素的长度，例如对字符串“ABCDABD”来说：

* A:前缀为空，后缀为空，共有元素为0个，长度自然是0；
* AB:前缀为["A”]，后缀为[“B”]，共有元素为0个，长度自然是0；
* ABC:前缀为["A”, “AB”]，后缀为[“BC”, “C”]，共有元素为0个，长度自然是0；
* ABCD:前缀为["A”, “AB”, “ABC”]，后缀为[“BCD”, “CD”, “D”]，共有元素为0个，长度自然是0；
* ABCDA:前缀为["A”, “AB”, “ABC”, “ABCD”]，后缀为[“BCDA”, “CDA”, “DA”, “A”]，共有元素为1个(“A”)，长度是1；
* ABCDAB:前缀为["A”, “AB”, “ABC”, “ABCD”, “ABCDA”]，后缀为[“BCDAB”, “CDAB”, “DAB”, “AB”, “B”]，共有元素为1个(“AB”)，长度是2；
* ABCDABD:前缀为["A”, “AB”, “ABC”, “ABCD”, “ABCDA”, “ABCDAB”]，后缀为[“BCDABD”, “CDABD”, “DABD”, “ABD”, “BD”, “D”]，共有元素为0个，长度是0；

部分匹配的实质，是有时候字符串的头部和尾部会有所重复。比如，“ABCDAB”中有两个AB，那么部分匹配值就是2。搜索词移动的时候，第一个AB向后移动4个字节，就可以来到第二个AB的位置。

### 2.1.3.部分匹配表

将所有字符的部分匹配值组织在一起，为一张部分匹配表；

如2.1.2.中ABCDABD的部分匹配表就是：

A B C D A B D

0 0 0 0 1 2 0

## 2.2.匹配过程

将模式串与主串从第一个字节开始、从前向后进行比较:

* 如果模式串的第一个字符与当前的主串对应处不匹配，模式串后移一位；
* 如果模式串的前m个字节已经与主串匹配，第m+1的字节匹配失败，那么**移动的数量 = 已匹配的字符数 - 最后一个匹配成功的字符对应的部分匹配值；**

例如，主串为BBC ABCDAB ABCD，模式串为2.1.2.中所述的ABCDABD，那么匹配过程为：

1. 第一个字符B,与模式串的第一个字符匹配失败，后移一位继续比较；
2. 后三个字节：B B 空格，都不能匹配，依然后移；
3. 字符A，第一个字节匹配成功了，继续匹配，知道空格和模式串最后一个字符“D”匹配失败了，此时匹配成功的数量是6，最后一个匹配成功的字符(B)对应的部分匹配值是2，因此移动位数为4；
4. 。。。。。。

# 3.BM算法

BM算法(Boyer Mooer algorithm)，通过引入坏字符表(bad character table)和好后缀表(good suffix table)，并采用自后向前的比较方式，进一步减少比较次数，可以取得一个较好的时间复杂度：O(n)；

所谓坏字符，指的是从后向前开始比较，一旦出现字节不相同的，就是坏字符，例如主串是“abc”，模式串是“de”，首先将ab和de进行比较，从后向前进行比较的话，第一个字符是b和e，不相等，那么b就是相对于模式串de的一个坏字符；

所谓好后缀，指的是从后向前开始比较，有部分数据是相同的，这一些相同的数据就是好后缀，例如：主串是“abcdefg”，模式串是“xbc”；开始比较后，将“abc”和“xbc”进行比较，从后向前，c匹配成功，b匹配成功，a和x匹配失败，那么“bc”就是好后缀，而如前所述，a就是坏字符；

## 3.1.坏字符规则

当发现坏字符时，需要将模式串向右移动，移动规则是:

* 后移位数 = 坏字符出现的位置 – 搜索词中**上一次出现**的位置；
* 如果坏字符不包含在搜索词中，“上一次出现的位置”为-1；

所谓上一次出现，指的是从坏字符出现的位置开始、向前找到的最后一次出现的位置。例如，主串是xaa，模式串是axa，那么坏字符“a”出现的位置是1，且上一次出现的位置是0，而不是2，那么移动位数就应该是（1-0）=1；

## 3.2.好后缀规则

好后缀规则表述为:

后移位数 = 好后缀的位置 – 搜索词中**上一次出现**的位置；

例如，假设字符串“ABCDAB”的后一个“AB”是好后缀，那么“好后缀的位置”是5**(从0开始计数，取最后一个字符的位置值)**，在“搜索词中上一次出现的位置”是1(第一个B出现的位置)，那么后移位数就应该是：5 - 1 = 4；

又比如，如果“ABCDEF”中的“EF”是好后缀，那么位置同样是5，上一次出现的位置就是-1(因为并未出现)，所以后移位数就是5 - (-1) = 6；

需要注意如下几个关键点：

* “好后缀的位置”以**最后一个字符**为准，从0开始计数；
* 如果好后缀在搜索词中只出现一次，那么“上一次出现的位置”为-1；
* 如果好后缀有多个
  + 对于最长的那个好后缀，“上一次出现的位置”可以出现在字符串的任何位置，例如“ABCDABCDAB”，如果好后缀是“AB”，那么上一次出现的位置就是4，而不是0；
  + 其他的好后缀，“上一次出现的位置”必须在头部：例如“BABCDAB”，如果好后缀是DAB、AB、B，那么由于DAB没有重复出现，AB虽然有重复出现但是并不在头部，因此好后缀使用的是B，且上一次出现的位置是0；

## 3.3.坏字符表和好后缀表

根据好后缀规则和好后缀规则，可以看到，这两个规则都只是和模式串有关，和主串并无关联，因此可以在进行比较之前，初始化这两个表格。一旦需要移位，直接从表中获取移动位数即可；

后移位数，取 坏字符表 和 好后缀表 中移动位数更多的值；

实际的算法实现中，后移位数并不是如上的实现，而是如果最后一个字节就没有匹配上，那么就对其使用了坏字符规则；如果有好后缀，就使用好后缀规则；

# 4.Sunday算法

Sunday算法同样是一种高效的字符串查找算法，其效率远高于KMP算法，对于随机字符串，很多情况下效率比BM算法更高，最关键的是，该算法的原理更为简单。

Sunday算法与BM算法很相似，只不过Sunday算法是从前向后匹配，一旦发现匹配失败的字符，关注的是主串中相对于模式串最末字符的下一个字符。例如，主串是“abc”，模式串是“d”，那么其开始比较后，将a和d首先比较，比较失败后关注的将是“b”；

后移规则是：

* 所关注的字符存在于模式串中：后移位数 = 模式串长度 – 该字符在模式串中的位置(从0开始计数)；
* 所关注的字符不存在于模式串中：后移位数= 模式串长度 + 1；

特别需要注意，该算法对最后一个字符的下一个字符的处理逻辑，谨防溢出！

校验算法(moUtils\_Check)

通讯过程的数据校验，指的是传输之前，在数据的尾部增加一些额外的附加信息，收到信息后，通过对这些信息进行校验，确认收到的数据和发出的数据是否相同。

# 1.奇偶校验

奇偶校验，是在要校验的数据后，补充0或者1，使数据中0或者1的bit总数为奇数或者偶数个。

例如，0x1a，二进制 0001 1010:

* 奇校验：1的总数为奇数个，后面补充0，1的个数是3，符合条件；
* 偶校验：1的总数为偶数个，后面补充1，1的个数就是4了，符合条件；

奇偶校验的优势很明显，位操作，容易实现，尤其容易硬件实现，常见的串口协议RS232，用的就是该校验方式；

缺点也很明显，有50%的可能性校验失败；

# 2.累加和校验

累加和校验，指的是将需要校验的数据累加，并将结果写入到最后一个字节中。这里累加的时候，将忽略进位的按照字节累加，如此才能保证最后的结果是一个字节的长度。

累加和实现起来非常简单，也经常被使用。但是这种校验方式，检验错误的能力一般，有1/256的机会出错。

# 3.CRC校验

**呃……**

**居然每个人对于算法的介绍都不同！！！**

**这个算法先搁置吧，受不了，找不到标准。**

CRC算法的基本思想，是将传输的数据当作一个位数很长的数字，使用这个数字除以另一个数，得到的余数作为校验值，加入到原始数据的后面。

最常用的几种生成多项式：

* CRC8 : X8 + X5 + X4 + X0；
* CRC12 = X12 + X11 + X3 + X2 + X0;
* CRC16 = X16 + X15 + X2 + X0;
* CRC-CCITT = X16 + X12 + X5 + X0;
* CRC-32 = X32 + X26 + X23 + X22 + X16 + X12 + X11 + X10 + X8 + X7 + X5 + X4 + X2 + X1 + X0;

位宽(width)，不是多项式对应的二进制数的位数，而是位数减1。比如CRC8的多项式，转换成二进制，就是1001 1001 0.这里说他的位数是8，其实是9-1；

普遍的数据，都采用16进制表示。由于生成多项式的最高位一定是1，且位宽限定了多项式的长度，因此第一位都被缩写了。例如CRC32的生成多项式，其实记作：0x04c1 1db7，最高位的1被省略了。

CRC算法规定，假设位宽为w，那么就要在原始数据后补充w个0，例如CRC8就需要补充8个0；

## 3.1.余数初始值

所谓余数初始值，就是在计算CRC值开始的时候，为CRC结果寄存器中赋的不是0，而是一个初始值。

|  |  |  |  |
| --- | --- | --- | --- |
|  | CRC-CCITT | CRC-16 | CRC-32 |
| 位宽w | 16 | 16 | 32 |
| 多项式值(除数) | 0x1021 | 0x8005 | 0x04c11db7 |
| 余数初始值 | 0xffff | 0x0000 | 0xffffffff |
| 结果异或值 | 0x0000 | 0x0000 | 0xffffffff |

## 3.2.算法描述

CRC计算过程算法描述：

1. 设置CRC寄存器的值，为余数初始值；
2. 将数据

数据结构与算法分析

# 第4章 树

定义树的一种自然的方式是递归的方法：一棵树是一些节点的集合。这个集合可是是空集；如果非空，那么一棵树由根节点r(root)以及0个或多个非空的子树T1，T2，……，Tk组成，这些子树的根节点都被来自r的一条有向的边(edge)所连接。

每一棵子树的根叫做r的儿子(child)，r是每一棵子树的父亲(parent)。从递归定义可以得知，一棵树是由N个节点和N-1条边组成的。

**深度**：对任意节点Ni，其深度指的是从根到Ni的唯一路径的长。特殊情况下，根节点的深度是0.

**高度**：对任意节点Ni，其高度指的是从Ni到一个树叶节点的最长路径的长度。特殊情况下，叶子结点的高是0，一棵树的高度等于根节点的高度。

一棵树的深度，总是等于其最深的叶子结点的深度；而这个深度，与这棵树的高度是相同的。

## 4.1.二叉树

二叉树是一棵树，其中每个节点都不能有超过2个的儿子节点。

二叉树有许多与搜索无关的重要应用，其主要用处之一是在编译器的设计领域，通过表达式树探索该问题。

### 4.1.1.表达式树

表达式树的树叶是操作数，其他的节点是操作符。例如如下的树：



预期的表达式结果是:(a + ( b \* c)) + (((d \* e) + f) \* g);

要得到这个预期的表达式，我们需要递归的产生一个带括号的左表达式，然后打印出根节点的运算符，再递归的产生一个右表达式。这种遍历方法(左-中-右)，称之为**中序遍历**。

另一个遍历策略，是递归打印出左子树、右子树，最后打印运算符，上面的树按照这种顺序应该输出：abc\*+de\*f+g\*+；这种方式(左-右-中)称为**后序遍历**。

第三种遍历策略，是先打印出运算符，再递归的打印子树，按照这种策略上面的树应输出：++a\*bc\*+\*defg；这种方式(中-左-右)称为**前序遍历**；

### 4.1.2.查找树ADT之二叉查找树

ADT, Attribute Distribute Tree，属性分散树；

二叉树的一个重要应用，是在查找中。

为了描述方便，假定每个节点都有一个关键字的值，这些关键字都假定为整数，且所有关键字互异，以后再处理有重复的情况。

使二叉树成为二叉查找树(Binary Search Tree)的性质是:对于树中的每个节点X，其左子树的所有关键字的值小于X的关键字值；右子树中所有节点关键字的值大于X的关键字的值。

这意味着，二叉查找树可以用某种统一的方式排序。

#### 插入操作

对于二叉查找树的插入操作，概念上是简单的。

为了将X插入到T中，首先使用find进行查找。如果找到了X就做一些属性值的刷新或什么都不做(做什么取决于T对于重复节点的处理策略)；否则将X插入到遍历的路径的最后一个节点上。

#### 删除操作

针对删除，需要考虑几种不同的情况：

* 叶子节点：直接删除；
* 有一个儿子节点的节点：该节点的父节点更改指向，指向其子节点，之后删除该节点即可；
* 有两个儿子节点的节点：首先找到该节点的右子树中最小的节点值；使用这个最小节点的内容更换到该节点中；删除该最小节点。由于该最小节点不可能有左儿子了，因此删除操作将异常简单。

删除更多时候可以考虑懒惰删除，也就是对节点增加一个标记位，执行删除操作时只是置位，并不实际删除。

### 4.1.3.AVL树

AVL(Adelson-Velskii and Landis)树，是其每个节点的左子树和右子树的高度最多差1的二叉查找树；

约定，空树的高度为-1；

对于AVL树，除去插入节点的动作，其他动作(查询、删除)都可以在L(logN)的时间复杂度下完成。插入操作之所以更加复杂，是因为插入一个新的节点，很有可能破坏原有的AVL特性，所以就需要调整树结构，以恢复AVL特性。这种修正AVL树特性的动作，一般称之为**旋转**(rotation)。

在插入发生时，只有那些从插入点到根节点的路径上的节点可能被打破平衡，因为只有这些节点的子树可能发生变化。当沿着这条路径，上行到根，并更新平衡信息时，我们可以找到一个节点，他的新平衡破坏了AVL条件。

我们假设这个必须重新平衡的节点是a。由于每个节点最多有2个儿子节点，因此高度不平衡时，a节点的两个子树的高度相差为2，共有4种可能性：

1. 对a节点的左儿子的左子树进行一次插入；
2. 对a节点的左儿子的右子树进行一次插入；
3. 对a节点的右儿子的左子树进行一次插入；
4. 对a节点的右儿子的右子树进行一次插入；

情况1和4，是针对a节点的镜像对称；情况2和3也是针对a节点的镜像对称。因此，理论上只需要讨论两种情况。

对于发生在“外边”的情况(情况1和4，也就是左左或右右)，通过对树进行一次单旋转(single rotation)；对于发生在“里边”的情况(情况2和3，也就是左右或右左)，通过双旋转(double rotation)来处理。

AVL树的平衡的重新建立，流程是：找到节点a(插入节点后，左右子树的高度相差2了)；分析情况；针对不同情况执行不同的旋转方式:左左单旋转、右右单旋转、左右双旋转、右左双旋转。