串的模式匹配就是从源字符串中查找子字符串，如果含有返回匹配成功的位置。

串的模式匹配问题有几个算法：

1.Brute-Force算法

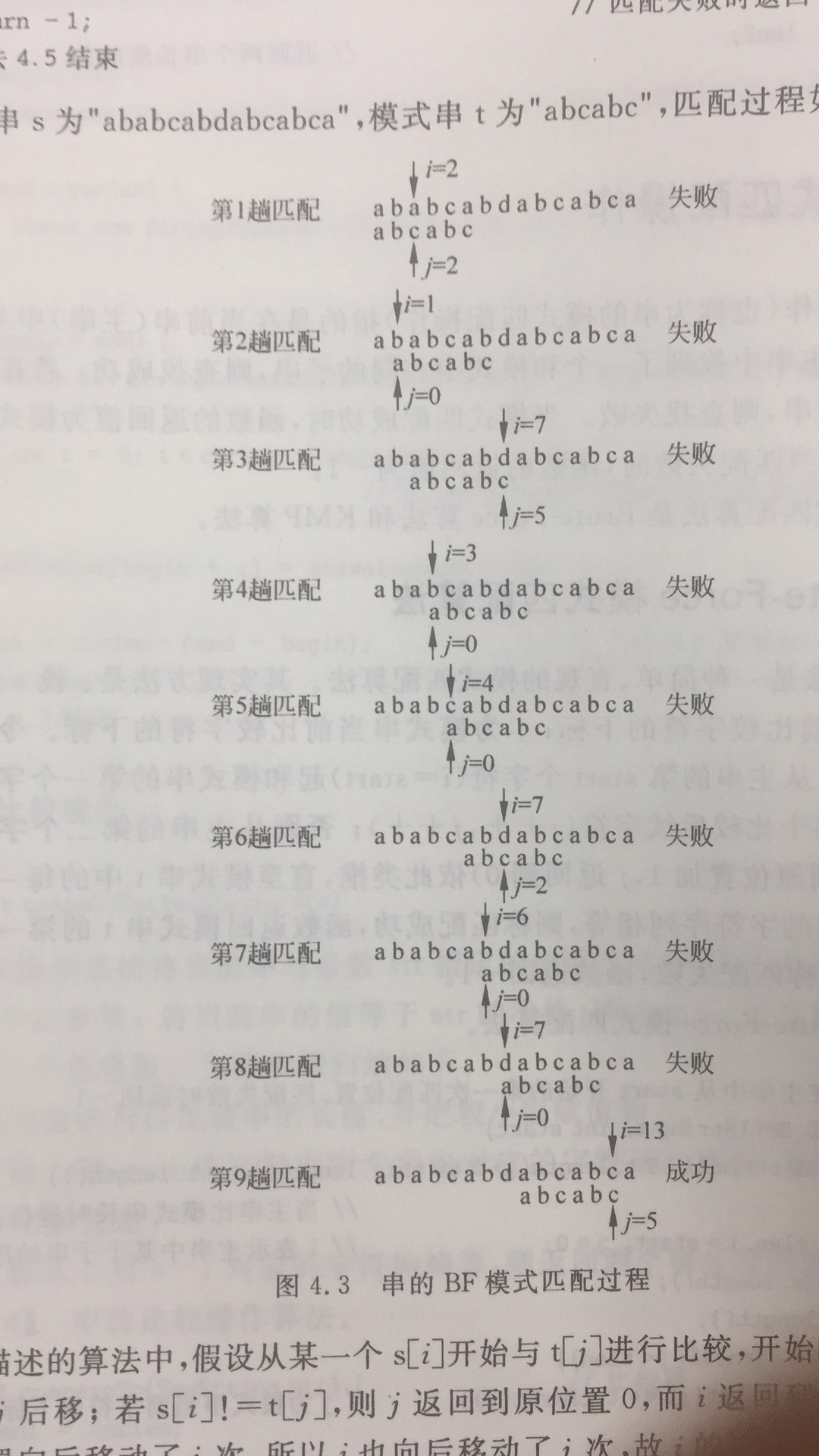
BF算法是一种简单、直观的模式匹配算法

从源字符串的第一位开始逐位进行子字符串的匹配。如果子字符串位数走到了末尾则匹配成功。当有一位匹配不成功，则退出当前匹配。然后在到源字符串的第二位进行匹配，依次类推。

我们用i记录源字符串s的当前位，用j记录子字符串t的位数。我们把子字符串t的头部对齐到源字符串s的i位。子字符串t的j位对应的是源字符串s的（i+j）位，也就是说我们比较t.charAt(j) 和 s.charAt(i+j) 是否相同。相同则往下一位比较j+1，不相同跳出子字符串遍历，源字符串的i+1。当子字符串的位数j等于子字符串的长度的时候，匹配成功。当源字符串全部匹配完子字符串还没匹配成功的时候失败。

从以上过程可知，源字符串位i和子字符串位j，每次的比较都会回退。

过程：如下图



代码：

for(int i=0;;i++){ // i记录源字符串的位数

for(int j=0;;j++){

if(j==t.length())//子字符串从t.charAt(0)到t.charAt(t.length-1)都等于源字符

return i; //匹配成功，返回匹配成功的位置

if(i+j>s.length()-1) //i+j是源字符串的目前位置,大于了s.length-1,源字符串走完

return -1; //源字符串都走完了，还没有匹配成功则匹配失败

if(t.charAt(j)!=s.charAt(i+j))//源字符串和子字符串对应位不相同

break; //跳出当前i位的子字符串匹配，进行下一位i++

/\* if(t.charAt(j)==s.charAt(i+j)) //这步可以不用写

countiue; //相同则继续j的循环，j++ \*/

}

}

2.KMP算法

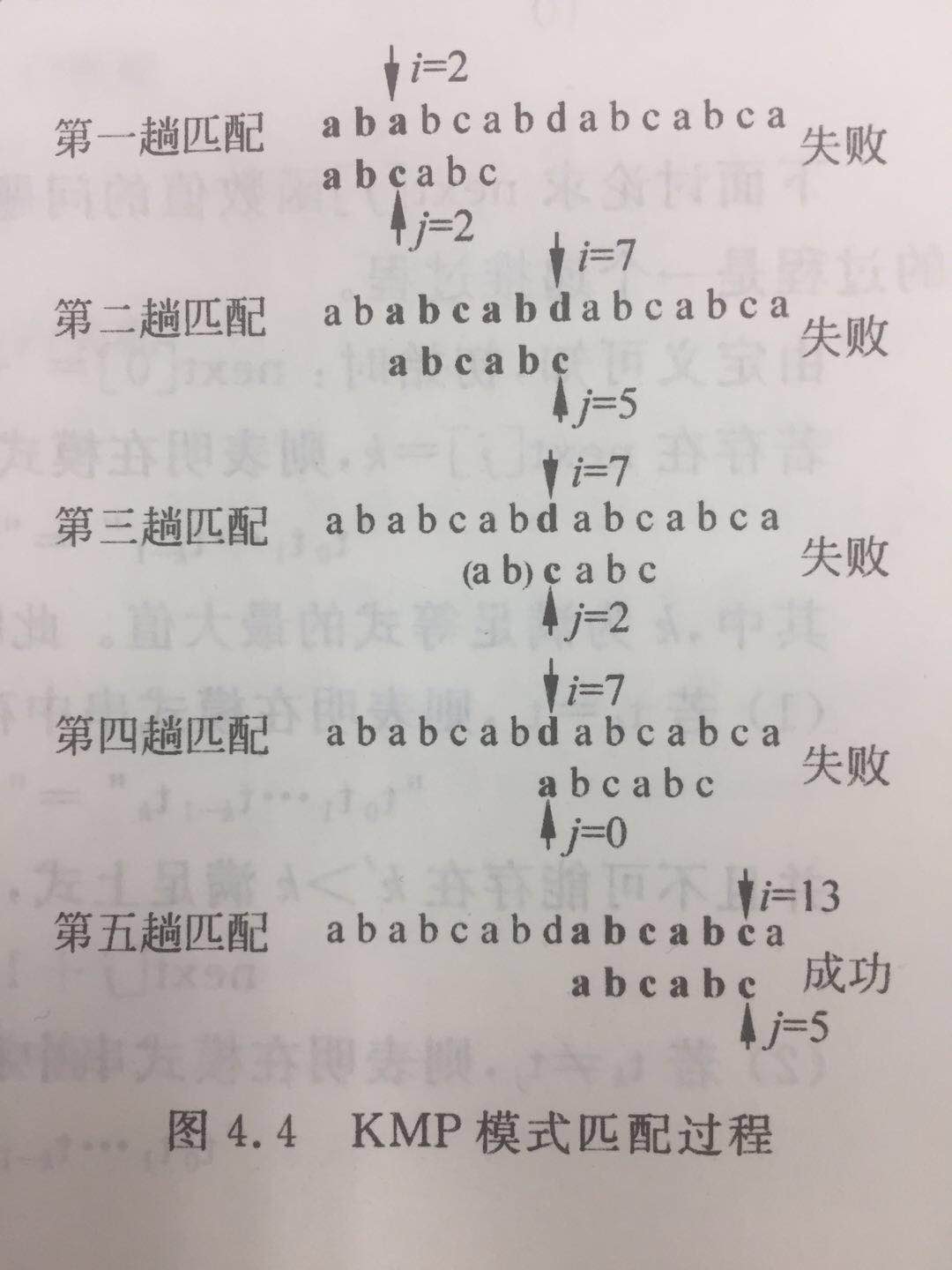
KMP算法是对BF算法的优化。当匹配失败的时候BF算法则是走到下一位也就是i++，开始匹配；而KMP算法则源字符串位置指针i不回退走到与子字符串匹配不相同的那一位开始下一次的匹配。而子字符串的位置指针j回退，但是不是根据BF算法那样回退到0，而是根据一个next[j]函数找到回退的位置，然后进行下次比较。源字符串匹配失败，子字符串回退比较，失败，子字符串再回退，比较，一直到j=next[0]，也就是j==-1这个回退的位置无法回退的时候，源字符串当前位置匹配失败，源字符串移动。

next[j]函数原理：每一个子字符串斗殴一个k值对应，这个k值仅与模式串本身有关，与源字符串无关。

定义next[0]=-1,next[1]=0，j是主字符串指针，i为模式串指针，j=1从1开始，i=0，从0开始。每次匹配的是t.charAt(j)和t.charAt(k)。匹配成功令next[j+1]=next[j]+1=k+1，j和k都向下走一步：j++，k++。匹配失败了用next[k]的值代替当前k的值再次进行匹配，直到k回退到0的时候彻底匹配失败，定义next[j+1]=0，主字符串向下走一位，开始下次匹配。可知，每次的j位置的匹配得出来得数值是j+1位置得next数组的值。

可知，j位置的匹配，得出j+1位置的next数组值。j无论匹配成功与否从1开始一直在走(j++)，而k则是匹配成功了才走(k++),匹配失败了则用next[k]的值代替当前k再次进行匹配(k=next[k])，直到k回退到0为止，如果失败则填0.

过程：



Next[j]函数的代码：

public int[] getNext(String t){

int[] next = new int[t.length()];

next[0]=-1;

next[1]=0;

int j = 1; //主串指针

int k = 0; //模式串指针

while (j<t.length()-1){ //从1开始，每次匹配都是定下一位的next[]数据的值

//每次匹配出来的数值是下一个位置的next值，到t.length-2就可以出t.length-1的值了

if(t.charAt(j)==t.charAt(k)){ //匹配成功

next[j+1]=k+1;

j++;

k++;

}

else if(k == 0){

next[j+1] = 0;

j++;

}

else

k=next[k];

}

return next;

}

KMP匹配:

public static int Kmp(String s,String t){

int[] next = getNext(t);

int i = 0;

int j = 0;

while(i<s.length()&&j<t.length()){ //j==-1 这行代码很关键，证明子字符串已经 回退到 0，主字符串可以进行下一位的匹配

if(j==-1||s.charAt(i)==t.charAt(j)) { //j==-1的时候说明子字符串已经回退到0后，t[0]!=s[j],又回退到-1,当j等于-1的时候说明彻底匹配失败

i++; //失败也都向下走一位，开始新的匹配。

j++; //i++向下走一位，j++等于0，回到子字符串头t.charAt[0]

}

else {

j=next[j]; //j指针回退，回退位置由next[j]数组决定

}

}

if(j<t.length()) //j没走完整个t,匹配失败 匹配成功j一定走完t

return -1;

else

return i-t.length();//匹配成功的时候，返回当前i位置减去t的长度，不能减j，以为j没有回退到0.

}

3.Sunday算法

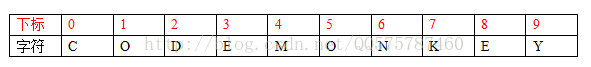
Sunday算法是Daniel M.Sunday于1990年提出的字符串模式匹配。其效率在匹配随机的字符串时比其他匹配算法还要更快。

过程：

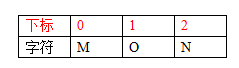
我们定义i为文本字符串s的指针，j为模式字符串t的指针，i=0，j=0。两字符串头部对齐，我们开始逐为比较，没有匹配成功，把t的尾巴与下一个位置k对齐，比较是否含有下一个位置的元素，t则从尾巴开始向前依次与s.charAt(k)比较。如果相等，停下。比较这时候与t头部对齐的位置开始进行下次的比较；如果都不相等，t向后再移动一位，开始下次的比较。

示例：

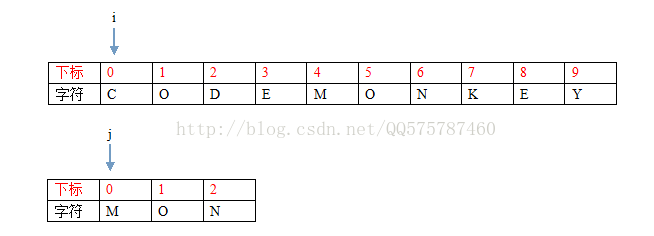
s：



t：



（1）i=0，j=0



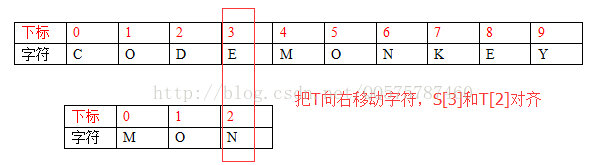
首先将s与t的字符串头部对齐，定义i为指向s头部的指针，j为指向t头部的指针。

（2）比较s.charAt(i)是否等于t.charAt(j)

相等：i++，j++，再次比较

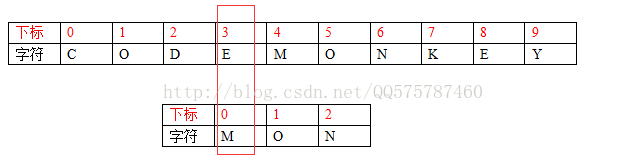
不相等，我们把t的尾巴移动到从s和t在当前位置末尾对齐的下一个位置

m=i+t.length,j=t.length-1

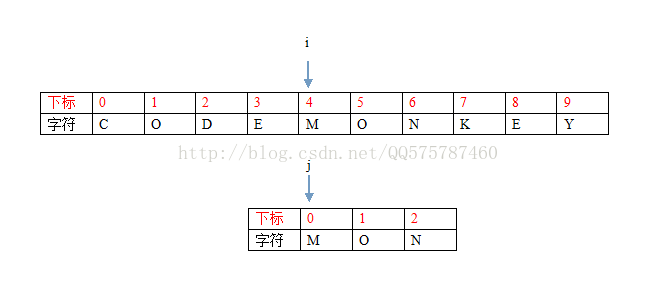


（3）比较s.charAt(i)和t.charAt(j)

不相等：t从尾向头移动一位，再次比较，直到到头为止



如果头也不相等，s向下移动一个为止，t向下移动一个位置。i++，j++



相等：停止，比较这个时候对齐的两字符串的头s.charAt(i)和t.charAt(j)开始比较

（4）循环比较

注意：s.charAt(i)和t.charAt(j)始终是对齐的。

代码：

public static int contains(char[] str,char ch){ //查找k位置的元素在t中的位置

for(int i = str.length-1;i>=0;i--){ //从t的尾巴开始向前遍历

if(str[i]==ch) //找到等于的s.charAt(k)的那个在t中的相对位置

return i; //找到返回该位置

}

return -1;//没找到返回-1

}

public static int Sunday(String s,String t){

int i = 0;

int j = 0;

int k;

while(i<s.length()-t.length()+j){

if(s.charAt(i)!=t.charAt(j)){ //不相等的时候

k=contains(t.toCharArray(),s.charAt(i+t.length()-j));

if(k==-1){ //不包含的时候i先移动到与j相同的位置

i=i+t.length()-j+1; //j每次移动的距离是t.length-j，i移动也一样

//但是这次i比j要多动一格，i=i+t.length-j+1

j=0; //j回调到t的头部

}

else {

i=i+t.length()-j-k; //因为在第k的位置相等，所以j少移动了k格，//j移动了t.length-j-k

j=0; //j回调到t的头部

}

}

else {

if(j==t.length()-1){ //j已经走到了字符串的最后一位,匹配成功

return i-j; //返回最开始匹配的位置，因为i走的长度始终等于j走

//的长度，j是从0开始走的所以j走的长度为j

//i走的长度也为j，i的起始点为i-j

}

else {

i++;

j++;

}

}

}

return -1;

}

注意：i走的长度始终等于j走的长度