第11章

串

串或字符串（string）属于线性结构，自然地可直接利用向量或列表等序列结构加以实现。但字符串作为数据结构，特点也极其鲜明，这可归纳为：结构简单，规模庞大，元素重复率高。

所谓结构简单，是指字符表本身的规模不大，甚至可能极小。以生物信息序列为例，参与蛋白质（文本）合成的常见氨基酸（字符）只有20种，而构成DNA序列（文本）的碱基（字符）仅有4种。尽管就规模而言，地球系统模式的单个输出文件长达1~100GB，微软Windows系统逾4000万行的源代码长度累计达到40GB，但它们都只不过是由ASCII字符，甚至是可打印字符组成的。因此，以字符串形式表示的海量文本数据的高效处理技术，一直都是相关领域的研究重点。

鉴于字符串结构的上述特点，本章将直接利用C++本身所提供的字符数组，并转而将讲述的重点，集中于各种串匹配算法indexOf()的基本原理与高效实现。

§11.1 串及串匹配

11.1.1 串

■ 字符串

一般地，由n个字符构成的串记作：

$$S = \text{"}a_0\ a_1\ \ldots\ a_{n-1}\text{"},\quad \text{其中，} a_i \in \Sigma,\ 0 \le i < n$$

这里的Σ是所有可用字符的集合，称作字符表（alphabet），例如二进制比特集Σ = { 0, 1 }、ASCII字符集、Unicode字符集、构成DNA序列的所有碱基、组成蛋白质的所有氨基酸，等等。

字符串S所含字符的总数n，称作S的长度，记作|S| = n。这里只考虑长度有限的串，n < ∞。特别地，长度为零的串称作空串（null string）。请注意，空串并非由空格字符'□'组成的串，二者完全不同。

■ 子串

字符串中任一连续的片段，称作其子串（substring）。具体地，对于任意的0 ≤ i ≤ i + k < n，由字符串S中起始于位置i的连续k个字符组成的子串记作：

$$S.substr(i, k) = \text{"}a_i\ a_{i+1}\ \ldots\ a_{i+k-1}\text{"} = S[i, i + k)$$

有两种特殊子串：起始于位置0、长度为k的子串称为前缀（prefix），而终止于位置n - 1、长度为k的子串称为后缀（suffix），分别记作：

$$prefix(S, k) = S.substr(0, k) = S[0, k)$$

$$suffix(S, k) = S.substr(n - k, k) = S[n - k, n)$$

由上述定义可直接导出以下结论：空串是任何字符串的子串，也是任何字符串的前缀和后缀；任何字符串都是自己的子串，也是自己的前缀和后缀。此类子串、前缀和后缀分别称作平凡子串（trivial substring）、平凡前缀（trivial prefix）和平凡后缀（trivial suffix）。反之，字符串本身之外的所有非空子串、前缀和后缀，分别称作真子串（proper substring）、真前缀（proper prefix）和真后缀（proper suffix）。

■ 判等

最后，字符串S[0, n)和T[0, m)称作相等，当且仅当二者长度相等（n = m），且对应的字符分别相同（对任何0 ≤ i < n都有S[i] = T[i]）。

■ ADT

串结构主要的操作接口可归纳为表11.1。

表11.1 串ADT支持的操作看接口

操 作 接 口	功 能
length()	查询串的长度
charAt(i)	返回第i个字符
substr(i, k)	返回从第i个字符起、长度为k的子串
prefix(k)	返回长度为k的前缀
suffix(k)	返回长度为k的后缀
equal(T)	判断T是否与当前字符串相等
concat(T)	将T串接在当前字符串之后
indexOf(P)	若P是当前字符串的一个子串，则返回该子串的起始位置；否则返回-1

比如，依次对串S = "data structures"执行如下操作，结果依次如表11.2所示。

表11.2 串操作实例

操 作	输 出	字 符 串 S
length()	15	"data structures"
charAt(5)	's'	"data structures"
prefix(4)	"data"	"data structures"
suffix(10)	"structures"	"data structures"
concat("and algorithms")	"data structures and algorithms"	
equal("data structures")	false	"data structures and algorithms"
equal("data structures and algorithms")	true	"data structures and algorithms"
indexOf("string")	-1	"data structures and algorithms"
indexOf("algorithm")	20	"data structures and algorithms"

11.1.2 串匹配

■ 应用与问题

在涉及字符串的众多实际应用中，模式匹配是最常使用的一项基本操作。比如UNIX Shell的grep工具（General Regular Expression Parser）和DOS的find命令，基本功能都是在指定的字符串中查找[①]特定模式的字符串。又如生物信息处理领域，也经常需要在蛋白质序列中

① 这两个命令都是以文件形式来指定待查找的文本串，具体格式分别是：

```
% grep <pattern> <file>
c:\> find "pattern" <file>
```

寻找特定的氨基酸模式，或在DNA序列中寻找特定的碱基模式。再如，邮件过滤器也需根据事先定义的特征串，通过扫描电子邮件的地址、标题及正文来识别垃圾邮件。还有，反病毒系统也会扫描刚下载的或将要执行的程序，并与事先提取的特征串相比对，以判定其中是否含有病毒。

上述所有应用问题，本质上都可转化和描述为如下形式：

如何在字符串数据中，检测和提取以字符串形式给出的某一局部特征

这类操作都属于串模式匹配（string pattern matching）范畴，简称串匹配。一般地，即：

对基于同一字符表的任何文本串T（|T| = n）和模式串P（|P| = m）：
判定T中是否存在某一子串与P相同
若存在（匹配），则报告该子串在T中的起始位置

串的长度n和m本身通常都很大，但相对而言n更大，即满足2 << m << n。比如，若：

```
T  =  "Now is the time for all good people to come"
P  =  "people"
```

则匹配的位置应该是T.indexOf(P) = 29。

■ **问题分类**

根据具体应用的要求不同，串匹配问题可以多种形式呈现。

有些场合属于模式检测（pattern detection）问题：我们只关心是否存在匹配而不关心具体的匹配位置，比如垃圾邮件的检测。有些场合属于模式定位（pattern location）问题：若经判断的确存在匹配，则还需确定具体的匹配位置，比如带毒程序的鉴别与修复。有些场合属于模式计数（pattern counting）问题：若有多处匹配，则统计出匹配子串的总数，比如网络热门词汇排行榜的更新。有些场合则属于模式枚举（pattern enumeration）问题：在有多处匹配时，报告出所有匹配的具体位置，比如网络搜索引擎。

11.1.3 测评标准与策略

串模式匹配是一个经典的问题，有名字的算法已不下三十种。鉴于串结构自身的特点，在设计和分析串模式匹配算法时也必须做特殊的考虑。其中首先需要回答的一个问题就是，如何对任一串匹配算法的性能作出客观的测量和评估。

多数读者首先会想到采用评估算法性能的常规口径和策略：以时间复杂度为例，假设文本串T和模式串P都是随机生成的，然后综合其各种组合从数学或统计等角度得出结论。很遗憾，此类构思并不适用于这一问题。

以基于字符表Σ = { 0, 1 }的二进制串为例。任给长度为n的文本串，其中长度为m的子串不过n - m + 1个（m << n时接近于n个）。另一方面，长度为m的随机模式串多达2^m个，故匹配成功的概率为n / 2^m。以n = 100,000、m = 100为例，这一概率仅有

$$100{,}000 \,/\, 2^{100} \;<\; 10^{-25}$$

对于更长的模式串、更大的字符表，这一概率还将更低。因此，这一策略并不能有效地覆盖成功匹配的情况，所得评测结论也无法准确地反映算法的总体性能。

实际上，有效涵盖成功匹配情况的一种简便策略是，随机选取文本串T，并从T中随机取出长度为m的子串作为模式串P。这也是本章将采用的评价标准。

§11.2 蛮力算法

11.2.1 算法描述

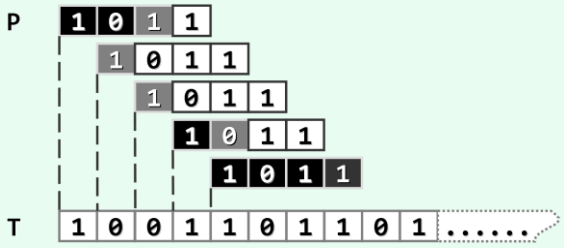

图11.1 串模式匹配的蛮力算法

蛮力串匹配是最直接最直觉的方法。如图11.1所示，可假想地将文本串和模式串分别写在两条印有等间距方格的纸带上，文本串对应的纸带固定，模式串纸带的首字符与文本串纸带的首字符对齐，二者都沿水平方向放置。于是，只需将P与T中长度为m的n - m + 1个子串逐一比对，即可确定可能的匹配位置。

不妨按自左向右的次序考查各子串。在初始状态下，T的前m个字符将与P的m个字符两两对齐。接下来，自左向右检查相互对齐的这m对字符：若当前字符对相互匹配，则转向下一对字符；反之一旦失配，则说明在此位置文本串与模式串不可能完全匹配，于是可将P对应的纸带右移一个字符，然后从其首字符开始与T中对应的新子串重新对比。图中，模式串P的每一黑色方格对应于字符对的一次匹配，每一灰色方格对应于一次失配，白色方格则对应于未进行的一次比对。若经过检查，当前的m对字符均匹配，则意味着整体匹配成功，从而返回匹配子串的位置。

蛮力算法的正确性显而易见：既然只有在某一轮的m次比对全部成功之后才成功返回，故不致于误报；反过来，所有对齐位置都会逐一尝试，故亦不致漏报。

11.2.2 算法实现

以下给出蛮力算法的两个实现版本。二者原理相同、过程相仿，但分别便于引入后续的不同改进算法，故在此先做一比较。

```
1  /*****************************************************************************************
2   * Text     :  0   1   2   .   .   .   i-j .   .   .   .   i   .   .   n-1
3   *             ------------------------|-------------------|------------
4   * Pattern  :                          0   .   .   .   .   j   .   .
5   *                                     |-------------------|
6   *****************************************************************************************/
7  int match ( char* P, char* T ) { //串匹配算法（Brute-force-1）
8     size_t n = strlen ( T ), i = 0; //文本串长度、当前接受比对字符的位置
9     size_t m = strlen ( P ), j = 0; //模式串长度、当前接受比对字符的位置
10    while ( j < m && i < n ) //自左向右逐个比对字符
11       if ( T[i] == P[j] ) //若匹配
12          { i ++;  j ++; } //则转到下一对字符
13       else //否则
14          { i -= j - 1; j = 0; } //文本串回退、模式串复位
15    return i - j; //如何通过返回值，判断匹配结果？
16 }
```

代码11.1 蛮力串匹配算法（版本一）

如代码11.1所示的版本借助整数i和j，分别指示T和P中当前接受比对的字符T[i]与P[j]。若当前字符对匹配，则i和j同时递增以指向下一对字符。一旦j增长到m则意味着发现了匹配，即可返回P相对于T的对齐位置i - j。一旦当前字符对失配，则i回退并指向T中当前对齐位置的下一字符，同时j复位至P的首字符处，然后开始下一轮比对。

```
/*****************************************************************************************
 * Text    :  0   1   2   .   .   .   i   i+1 .   .   .   i+j .   .   n-1
 *            ------------------------|-------------------|------------
 * Pattern :                          0   1   .   .   .   j   .   .
 *                                    |-------------------|
 *****************************************************************************************/
int match ( char* P, char* T ) { //串匹配算法（Brute-force-2）
   size_t n = strlen ( T ), i = 0; //文本串长度、与模式串首字符的对齐位置
   size_t m = strlen ( P ), j; //模式串长度、当前接受比对字符的位置
   for ( i = 0; i < n - m + 1; i++ ) { //文本串从第i个字符起，与
      for ( j = 0; j < m; j++ ) //模式串中对应的字符逐个比对
         if ( T[i + j] != P[j] ) break; //若失配，模式串整体右移一个字符，再做一轮比对
      if ( j >= m ) break; //找到匹配子串
   }
   return i; //如何通过返回值，判断匹配结果？
}
```

代码11.2 蛮力串匹配算法（版本二）

如代码11.2所示的版本，借助整数i指示P相对于T的对齐位置，且随着i不断递增，对齐的位置逐步右移。在每一对齐位置i处，另一整数j从0递增至m - 1，依次指示当前接受比对的字符为T[i + j]与P[j]。因此，一旦发现匹配，即可直接返回当前的对齐位置i。

11.2.3 时间复杂度

从理论上讲，蛮力算法至多迭代n - m + 1轮，且各轮至多需进行m次比对，故总共只需做不超过(n - m + 1)·m次比对。那么，这种最坏情况的确会发生吗？答案是肯定的。

考查如图11.2所示的实例。无论采用上述哪个版本的蛮力算法，都需做n - m + 1轮迭代，且各轮都需做m次比对。因此，整个算法共需做m·(n - m - 1)次字符比对，其中成功的和失败的各有(m - 1)·(n - m - 1) + 1和n - m - 2次。因m << n，渐进的时间复杂度应为$O(n \cdot m)$。

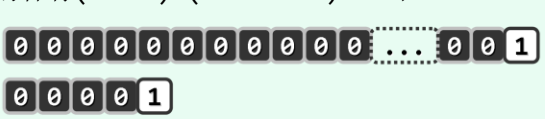

图11.2 蛮力算法的最坏情况
（也是基于坏字符策略BM算法的最好情况）

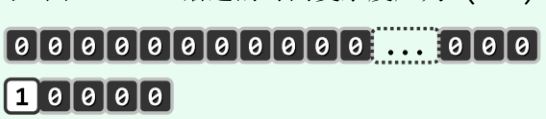

图11.3 蛮力算法的最好情况
（也是基于坏字符策略BM算法的最坏情况）

当然，蛮力算法的效率也并非总是如此低下。如图11.3所示，若将模式串P左右颠倒，则每经一次比对都可排除文本串中的一个字符，故此类情况下的运行时间将为$O(n)$。实际上，此类最好（或接近最好）情况出现的概率并不很低，尤其是在字符表较大时（习题[11-9]）。

§11.3 KMP算法

11.3.1 构思

上一节的分析表明，蛮力算法在最坏情况下所需时间，为文本串长度与模式串长度的乘积，故无法应用于规模稍大的应用环境，很有必要改进。为此，不妨从分析以上最坏情况入手。

稍加观察不难发现，问题在于这里存在大量的局部匹配：每一轮的m次比对中，仅最后一次可能失配。而一旦发现失配，文本串、模式串的字符指针都要回退，并从头开始下一轮尝试。

实际上，这类重复的字符比对操作没有必要。既然这些字符在前一轮迭代中已经接受过比对并且成功，我们也就掌握了它们的所有信息。那么，如何利用这些信息，提高匹配算法的效率呢？

以下以蛮力算法的前一版本（代码11.1）为基础进行改进。

■ 简单示例

如图11.4所示，用T[i]和P[j]分别表示当前正在接受比对的一对字符。

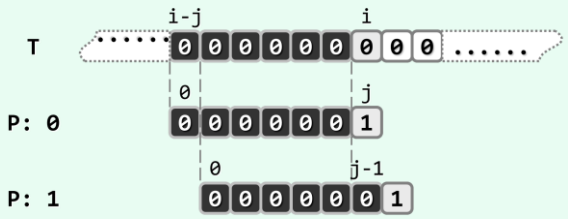

图11.4 利用以往的成功比对所提供的信息，可以避免文本串字符指针的回退

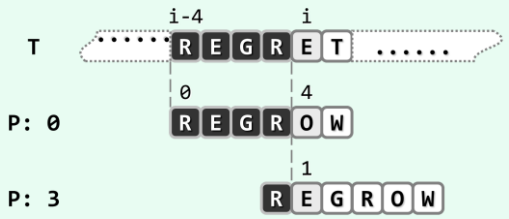

图11.5 利用以往的成功比对所提供的信息，有可能使模式串大跨度地右移

当本轮比对进行到最后一对字符并发现失配后，蛮力算法会令两个字符指针同步回退（即令i = i - j + 1和j = 0），然后再从这一位置继续比对。然而事实上，指针i完全不必回退。

■ 记忆 = 经验 = 预知力

经过前一轮比对，我们已经清楚地知道，子串T[i - j, i)完全由'0'组成。记住这一性质便可预测出：在回退之后紧接着的下一轮比对中，前j - 1次比对必然都会成功。因此，可直接令i保持不变，令j = j - 1，然后继续比对。如此，下一轮只需1次比对，共减少j - 1次！

上述“令i保持不变、j = j - 1”的含义，可理解为“令P相对于T右移一个单元，然后从前一失配位置继续比对”。实际上这一技巧可推而广之：利用以往的成功比对所提供的信息（记忆），不仅可避免文本串字符指针的回退，而且可使模式串尽可能大跨度地右移（经验）。

■ 一般实例

如图11.5所示，再来考查一个更具一般性的实例。

本轮比对进行到发现T[i] = 'E' ≠ 'O' = P[4]失配后，在保持i不变的同时，应将模式串P右移几个单元呢？有必要逐个单元地右移吗？不难看出，在这一情况下移动一个或两个单元都是徒劳的。事实上，根据此前的比对结果，此时必然有

T[i - 4, i) = P[0, 4) = "REGR"

若在此局部能够实现匹配，则至少紧邻于T[i]左侧的若干字符均应得到匹配——比如，当P[0]与T[i - 1]对齐时，即属这种情况。进一步地，若注意到i - 1是能够如此匹配的最左侧位置，即可直接将P右移4 - 1 = 3个单元（等效于i保持不变，同时令j = 1），然后继续比对。

11.3.2 next表

一般地，如图11.6假设前一轮比对终止于T[i] ≠ P[j]。按以上构想，指针i不必回退，而是将T[i]与P[t]对齐并开始下一轮比对。那么，t准确地应该取作多少呢？

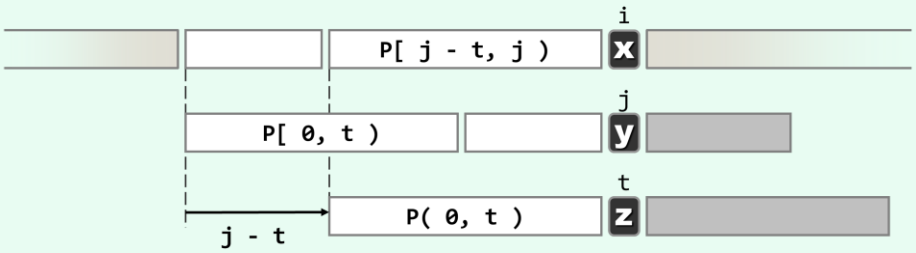

图11.6 利用此前成功比对所提供的信息，在安全的前提下尽可能大跨度地右移模式串

由图可见，经过此前一轮的比对，已经确定匹配的范围应为：

```
P[0, j)  =  T[i - j, i)
```

于是，若模式串P经适当右移之后，能够与T的某一（包含T[i]在内的）子串完全匹配，则一项必要条件就是：

```
P[0, t)  =  T[i - t, i)  =  P[j - t, j)
```

亦即，在P[0, j)中长度为t的真前缀，应与长度为t的真后缀完全匹配，故t必来自集合：

```
N(P, j)  =  { 0 ≤ t < j  |  P[0, t) = P[j - t, j) }
```

一般地，该集合可能包含多个这样的t。但需要特别注意的是，其中具体由哪些t值构成，仅取决于模式串P以及前一轮比对的首个失配位置P[j]，而与文本串T无关！

从图11.6还可看出，若下一轮比对将从T[i]与P[t]的比对开始，这等效于将P右移j - t个单元，位移量与t成反比。因此，为保证P与T的对齐位置（指针i）绝不倒退，同时又不致遗漏任何可能的匹配，应在集合N(P, j)中挑选最大的t。也就是说，当有多个值得试探的右移方案时，应该保守地选择其中移动距离最短者。于是，若令

```
next[j]  =  max( N(P, j) )
```

则一旦发现P[j]与T[i]失配，即可转而将P[next[j]]与T[i]彼此对准，并从这一位置开始继续下一轮比对。

既然集合N(P, j)仅取决于模式串P以及失配位置j，而与文本串无关，作为其中的最大元素，next[j]也必然具有这一性质。于是，对于任一模式串P，不妨通过预处理提前计算出所有位置j所对应的next[j]值，并整理为表格以便此后反复查询——亦即，将“记忆力”转化为“预知力”。

11.3.3 KMP算法

上述思路可整理为代码11.3，即著名的KMP算法[②]。

这里，假定可通过buildNext()构造出模式串P的next表。对照代码11.1的蛮力算法，只是在else分支对失配情况的处理手法有所不同，这也是KMP算法的精髓所在。

② Knuth和Pratt师徒，与Morris几乎同时发明了这一算法。他们稍后联合署名发表[60]该算法，并以其姓氏首字母命名

```
int match ( char* P, char* T ) {  //KMP算法
   int* next = buildNext ( P ); //构造next表
   int n = ( int ) strlen ( T ), i = 0; //文本串指针
   int m = ( int ) strlen ( P ), j = 0; //模式串指针
   while ( j < m  && i < n ) //自左向右逐个比对字符
      if ( 0 > j || T[i] == P[j] ) //若匹配，或P已移出最左侧（两个判断的次序不可交换）
         { i ++;  j ++; } //则转到下一字符
      else //否则
         j = next[j]; //模式串右移（注意：文本串不用回退）
   delete [] next; //释放next表
   return i - j;
}
```

代码11.3 KMP主算法（待改进版）

11.3.4 next[0] = -1

不难看出，只要$j > 0$则必有$0 \in N(P, j)$。此时$N(P, j)$非空，从而可以保证“在其中取最大值”这一操作的确可行。但反过来，若$j = 0$，则即便集合$N(P, j)$可以定义，也必是空集。此种情况下，又该如何定义$next[j = 0]$呢？

表11.3 next表实例：假想地附加一个通配符P[-1]

rank	-1	0	1	2	3	4	5	6	7	8	9
P[]	*	C	H	I	N	C	H	I	L	L	A
next[]	N/A	-1	0	0	0	0	1	2	3	0	0

反观串匹配的过程。若在某一轮比对中首对字符即失配，则应将P直接右移一个字符，然后启动下一轮比对。因此如表11.3所示，不妨假想地在P[0]的左侧“附加”一个P[-1]，且该字符与任何字符都是匹配的。就实际效果而言，这一处理方法完全等同于“令$next[0] = -1$”。

11.3.5 next[j + 1]

那么，若已知$next[0, j]$，如何才能递推地计算出$next[j + 1]$？是否有高效方法？

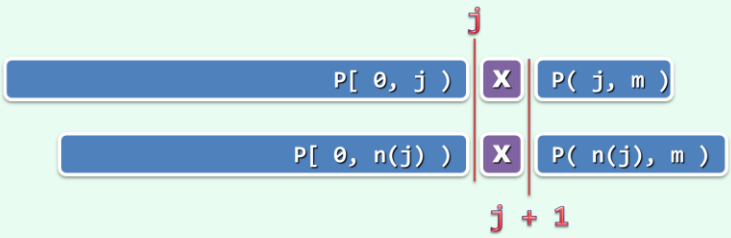

图11.7 P[j] = P[next[j]]时，必有next[j + 1] = next[j] + 1

若$next[j] = t$，则意味着在$P[0, j)$中，自匹配的真前缀和真后缀的最大长度为t，故必有$next[j + 1] \le next[j] + 1$——而且特别地，当且仅当$P[j] = P[t]$时如图11.7取等号。

那么一般地，若$P[j] \ne P[t]$，又该如何得到$next[j + 1]$？

此种情况下如图11.8，由next表的功能定义，next[j + 1]的下一候选者应该依次是

next[next[j]] + 1, next[next[next[j]]] + 1, ...

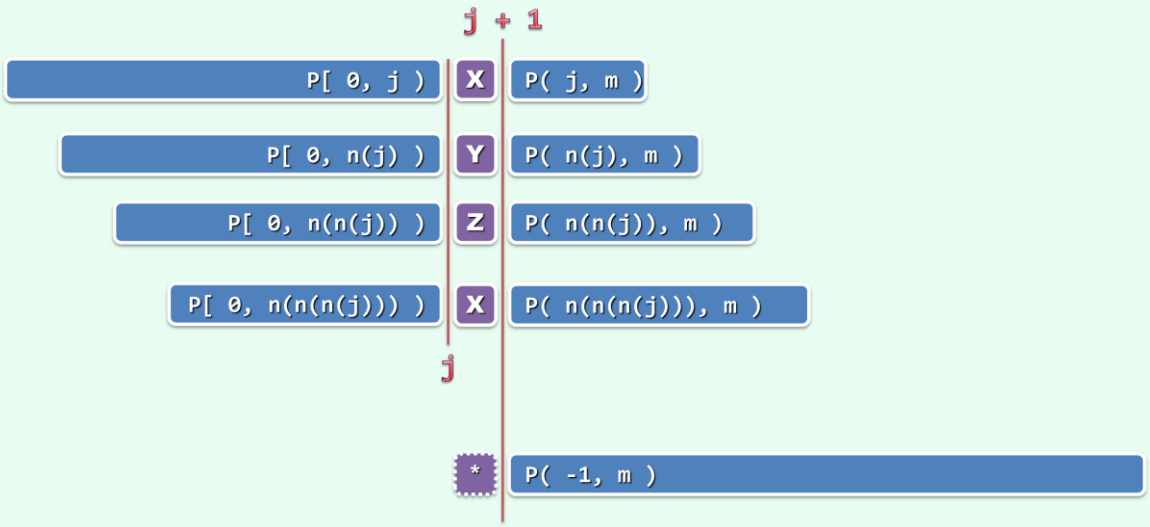

图11.8 P[j] ≠ P[next[j]]时，必有next[j + 1] = next[... next[j] ...] + 1

因此，只需反复用next[t]替换t（即令t = next[t]），即可按优先次序遍历以上候选者；一旦发现P[j]与P[t]匹配（含与P[t = -1]的通配），即可令next[j + 1] = next[t] + 1。

既然总有next[t] < t，故在此过程中t必然严格递减；同时，即便t降低至0，亦必然会终止于通配的next[0] = -1，而不致下溢。如此，该算法的正确性完全可以保证。

11.3.6 构造next表

按照以上思路，可实现next表构造算法如代码11.4所示。

```
1  int* buildNext ( char* P ) { //构造模式串P的next表
2     size_t m = strlen ( P ), j = 0; // "主"串指针
3     int* N = new int[m]; //next表
4     int t = N[0] = -1; //模式串指针
5     while ( j < m - 1 )
6        if ( 0 > t || P[j] == P[t] ) { //匹配
7           j ++; t ++;
8           N[j] = t; //此句可改进...
9        } else //失配
10          t = N[t];
11    return N;
12 }
```

代码11.4 next表的构造

可见，next表的构造算法与KMP算法几乎完全一致。实际上按照以上分析，这一构造过程完全等效于模式串的自我匹配，因此两个算法在形式上的近似亦不足为怪。

11.3.7 性能分析

由上可见，KMP算法借助next表可避免大量不必要的字符比对操作，但这意味着渐进意义上的时间复杂度会有实质改进吗？这一点并非一目了然，甚至乍看起来并不乐观。比如就最坏情况而言，共有Ω(n)个对齐位置，而且在每一对齐位置都有可能需要比对多达Ω(m)次。

如此说来，难道在最坏情况下，KMP算法仍可能共需执行Ω(nm)次比对？不是的。以下更为精确的分析将证明，即便在最坏情况下，KMP算法也只需运行线性的时间！

为此，请留意代码11.3中用作字符指针的变量i和j。若令k = 2i - j并考查k在KMP算法过程中的变化趋势，则不难发现：while循环每迭代一轮，k都会严格递增。

实际上，对应于while循环内部的if-else分支，无非两种情况：若转入if分支，则i和j同时加一，于是k = 2i - j必将增加；反之若转入else分支，则尽管i保持不变，但在赋值j = next[j]之后j必然减小，于是k = 2i - j也必然会增加。

纵观算法的整个过程：启动时有i = j = 0，即k = 0；算法结束时i ≤ n且j ≥ 0，故有k ≤ 2n。在此期间尽管整数k从0开始持续地严格递增，但累计增幅不超过2n，故while循环至多执行2n轮。另外，while循环体内部不含任何循环或调用，故只需$\mathcal{O}$(1)时间。因此，若不计构造next表所需的时间，KMP算法本身的运行时间不超过$\mathcal{O}$(n)。也就是说，尽管可能有Ω(n)个对齐位置，但就分摊意义而言，在每一对齐位置仅需$\mathcal{O}$(1)次比对（习题[11-4]）。

既然next表构造算法的流程与KMP算法并无实质区别，故仿照上述分析可知，next表的构造仅需$\mathcal{O}$(m)时间。综上可知，KMP算法的总体运行时间为$\mathcal{O}$(n + m)。

11.3.8 继续改进

尽管以上KMP算法已可保证线性的运行时间，但在某些情况下仍有进一步改进的余地。

考查模式串P = "000010"。按照11.3.2节的定义，其next表应如表11.4所示。

表11.4 next表仍有待优化的实例

rank	-1	0	1	2	3	4	5
P[]	*	0	0	0	0	1	0
next[]	N/A	-1	0	1	2	3	0

在KMP算法过程中，假设如图11.9前一轮比对因T[i] = '1' ≠ '0' = P[3]失配而中断。于是按照以上的next表，接下来KMP算法将依次将P[2]、P[1]和P[0]与T[i]对准并做比对。

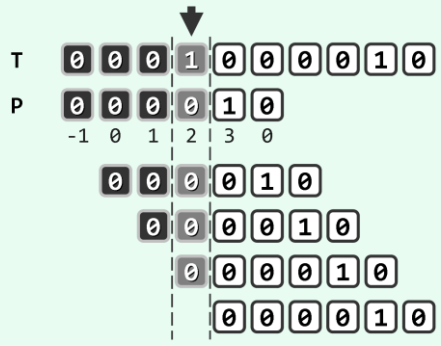

图11.9 按照此前定义的next表，仍有可能进行多次本不必要的字符比对操作

从图11.9可见，这三次比对都报告“失配”。那么，这三次比对的失败结果属于偶然吗？进一步地，这些比对能否避免？

实际上，即便说P[3]与T[i]的比对还算必要，后续的这三次比对却都是不必要的。实际上，它们的失败结果早已注定。

只需注意到P[3] = P[2] = P[1] = P[0] = '0'，就不难看出这一点——既然经过此前的比对已发现T[i] ≠ P[3]，那么继续将T[i]和那些与P[3]相同的字符做比对，既重蹈覆辙，更徒劳无益。

■ 记忆 = 教训 = 预知力

就算法策略而言，11.3.2节引入next表的实质作用，在于帮助我们利用以往成功比对所提供的“经验”，将记忆力转化为预知力。然而实际上，此前已进行过的比对还远不止这些，确切地说还包括那些失败的比对——作为“教训”，它们同样有益，但可惜此前一直被忽略了。

依然以图11.9为例，以往所做的失败比对，实际上已经为我们提供了一条极为重要的信息——T[i] ≠ P[4]——可惜我们却未能有效地加以利用。原算法之所以会执行后续四次本不必要的比对，原因也正在于未能充分汲取教训。

■ 改进

为把这类“负面”信息引入next表，只需将11.3.2节中集合N(P, j)的定义修改为：

$$N(P, j) = \{ 0 \le t < j \mid P[0, t) = P[j - t, j) \text{ 且 } P[t] \ne P[j] \}$$

也就是说，除“对应于自匹配长度”以外，t只有还同时满足“当前字符对不匹配”的必要条件，方能归入集合N(P, j)并作为next表项的候选。

相应地，原next表构造算法（代码11.4）也需稍作修改，调整为如下改进版本。

```
1  int* buildNext ( char* P ) { //构造模式串P的next表（改进版本）
2     size_t m = strlen ( P ), j = 0; // “主”串指针
3     int* N = new int[m]; //next表
4     int t = N[0] = -1; //模式串指针
5     while ( j < m - 1 )
6        if ( 0 > t || P[j] == P[t] ) { //匹配
7           j ++; t ++;
8           N[j] = ( P[j] != P[t] ? t : N[t] ); //注意此句与未改进之前的区别
9        } else //失配
10          t = N[t];
11    return N;
12 }
```

代码11.5 改进的next表构造算法

由代码11.5可见，改进后的算法与原算法的唯一区别在于，每次在P[0, j)中发现长度为t的真前缀和真后缀相互匹配之后，还需进一步检查P[j]是否等于P[t]。唯有在P[j] ≠ P[t]时，才能将t赋予next[j]；否则，需转而代之以next[t]。

仿照11.3.7节的分析方法易知，改进后next表的构造算法同样只需$O(m)$时间。

■ 实例

仍以P = "000010"为例，改进之后的next表如表11.5所示。读者可参照图11.9，就计算效率将新版本与原版本（表11.4）做一对比。

表11.5 改进后的next表实例

rank	-1	0	1	2	3	4	5
P[]	*	0	0	0	0	1	0
next[]	N/A	-1	-1	-1	-1	3	-1

利用新的next表针对图11.9中实例重新执行KMP算法，在首轮比对因T[i] = '1' ≠ '0' = P[3]失配而中断之后，将随即以P[next[3]] = P[-1]（虚拟通配符）与T[i]对齐，并启动下一轮比对。将其效果而言，等同于聪明且安全地跳过了三个不必要的对齐位置。

§11.4 *BM算法

11.4.1 思路与框架

■ 构思

KMP算法的思路可概括为：当前比对一旦失配，即利用此前的比对（无论成功或失败）所提供的信息，尽可能长距离地移动模式串。其精妙之处在于，无需显式地反复保存或更新比对的历史，而是独立于具体的文本串，事先根据模式串预测出所有可能出现的失配情况，并将这些信息“浓缩”为一张next表。就其总体思路而言，本节将要介绍的BM算法[③]与KMP算法类似，二者的区别仅在于预测和利用“历史”信息的具体策略与方法。

BM算法中，模式串P与文本串T的对准位置依然“自左向右”推移，而在每一对准位置却是“自右向左”地逐一比对各字符。具体地，在每一轮自右向左的比对过程中，一旦发现失配，则将P右移一定距离并再次与T对准，然后重新一轮自右向左的扫描比对。为实现高效率，BM算法同样需要充分利用以往的比对所提供的信息，使得P可以“安全地”向后移动尽可能远的距离。

■ 主体框架

BM算法的主体框架，可实现如代码11.6所示。

```
1  int match ( char* P, char* T ) { //Boyer-Morre算法（完全版，兼顾Bad Character与Good Suffix）
2     int* bc = buildBC ( P ); int* gs = buildGS ( P ); //构造BC表和GS表
3     size_t i = 0; //模式串相对于文本串的起始位置（初始时与文本串左对齐）
4     while ( strlen ( T ) >= i + strlen ( P ) ) { //不断右移（距离可能不止一个字符）模式串
5        int j = strlen ( P ) - 1; //从模式串最末尾的字符开始
6        while ( P[j] == T[i + j] ) //自右向左比对
7           if ( 0 > --j ) break;
8        if ( 0 > j ) //若极大匹配后缀 == 整个模式串（说明已经完全匹配）
9           break; //返回匹配位置
10       else //否则，适当地移动模式串
11          i += __max ( gs[j], j - bc[ T[i + j] ] ); //位移量根据BC表和GS表选择大者
12    }
13    delete [] gs; delete [] bc; //销毁GS表和BC表
14    return i;
15 }
```

代码11.6 BM主算法

可见，这里采用了蛮力算法后一版本（310页代码11.2）的方式，借助整数i和j指示文本串中当前的对齐位置T[i]和模式串中接受比对的字符P[j]。不过，一旦局部失配，这里不再是机械地令i += 1并在下一字符处重新对齐，而是采用了两种启发式策略确定最大的安全移动距离。为此，需经过预处理，根据模式串P整理出坏字符和好后缀两类信息。

与KMP一样，算法过程中指针i始终单调递增；相应地，P相对于T的位置也绝不回退。

[③] 由R. S. Boyer和J. S. Moore于1977年发明[61]

11.4.2 坏字符策略

■ 坏字符

如图11.10(a)和(b)所示，若模式串P当前在文本串T中的对齐位置为i，且在这一轮自右向左将P与substr(T, i, m)的比对过程中，在P[j]处首次发现失配：

T[i + j] = 'X' ≠ 'Y' = P[j]

则将'X'称作坏字符（bad character）。问题是：

接下来应该选择P中哪个字符对准T[i + j]，然后开始下一轮自右向左的比对？

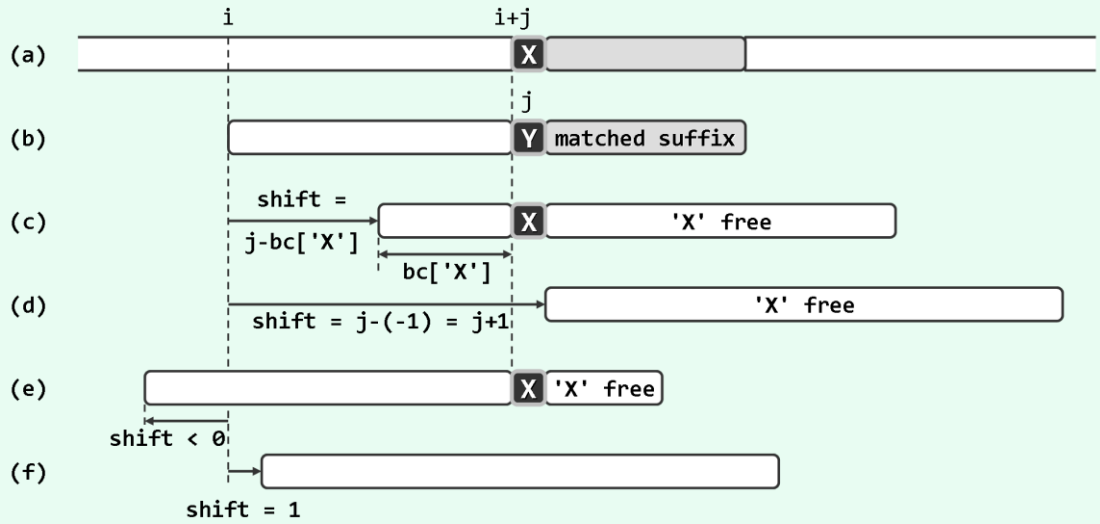

图11.10 坏字符策略：通过右移模式串P，使T[i + j]重新得到匹配

若P与T的某一（包括T[i + j]在内的）子串匹配，则必然在T[i + j] = 'X'处匹配；反之，若与T[i + j]对准的字符不是'X'，则必然失配。故如图11.10(c)所示，只需找出P中的每一字符 'X'，分别与T[i + j] = 'X'对准，并执行一轮自右向左的扫描比对。不难看出，对应于每个这样的字符'X'，P的位移量仅取决于原失配位置j，以及'X'在P中的秩，而与T和i无关！

■ bc[]表

若P中包含多个'X'，则是否真地有必要逐一尝试呢？实际上，这既不现实——如此将无法确保文本串指针i永不回退——更不必要。一种简便而高效的做法是，仅尝试P中最靠右的字符'X'（若存在）。与KMP算法类似，如此便可在确保不致遗漏匹配的前提下，始终单向地滑动模式串。具体如图11.10(c)所示，若P中最靠右的字符'X'为P[k] = 'X'，则P的右移量即为j - k。

同样幸运的是，对于任一给定的模式串P，k值只取决于字符T[i + j] = 'X'，因此可将其视作从字符表到整数（P中字符的秩）的一个函数：

$$bc(c) = \begin{cases} k & \text{（若P[k] = c，且对所有的i > k都有P[i] ≠ c）} \\ -1 & \text{（若P[]中不含字符c）} \end{cases}$$

故如代码11.6所示，如当前对齐位置为i，则一旦出现坏字符P[j] = 'Y'，即重新对齐于：

i += j - bc[T[i + j]]

并启动下一轮比对。为此可仿照KMP算法，预先将函数bc()整理为一份查询表，称作BC表。

■ 特殊情况

可用的BC表，还应足以处理各种特殊情况。比如，若P根本就不含坏字符'X'，则如图11.10(d)所示，应将该串整体移过失配位置T[i + j]，用P[0]对准T[i + j + 1]，再启动下一轮比对。实际上，上述对bc()函数的定义已给出了应对方法——将BC表中此类字符的对应项置为-1。这种处理手法与KMP算法类似，其效果也等同于在模式串的最左端，增添一个通配符。

另外，即使P串中含有坏字符'X'，但其中最靠右者的位置也可能太靠右，以至于k = bc['X'] ≥ j。此时的j - k不再是正数，故若仍以此距离右移模式串，则实际效果将如图11.10(e)所示等同于左移。显然，这类移动并不必要——匹配算法若果真能够进行至此，则此前左侧的所有位置都已被显式或隐式地否定排除了。因此，这种情况下不妨如图11.10(f)所示，简单地将P串右移一个字符，然后启动下一轮自右向左的比对。

■ bc[]表实例

以由大写英文字母和空格组成的字符表Σ = { '□', 'A'~'Z' }为例。按照以上定义，与模式串"DATA STRUCTURES"相对应的BC表应如表11.6所示。

表11.6 模式串P = "DATA STRUCTURES"及其对应的BC表

rank	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P[]	*	D	A	T	A	□	S	T	R	U	C	T	U	R	E	S

char	□	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
bc[]	4	3	-1	9	0	13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	12	14	10	11	-1	-1	-1	-1	-1

其中，字符'A'在秩为1和3处出现了两次，bc['A']取作其中的大者3；字符'T'则在秩为2、6和10处出现了三次，bc['T']取作其中的最大者10。在该字符串中并未出现的字符，对应的BC表项均统一取作-1，等效于指向在字符串最左端假想着增添的通配符。

■ bc[]表构造算法

按照上述思路，BC表的构造算法可实现如代码11.7所示。

```
1  //*****************************************************************************************
2  //    0                      bc['X']                              m-1
3  //    |                      |                                    |
4  //    ......................X************************************
5  //                           .|<------------ 'X' free ------------>|
6  //*****************************************************************************************
7  int* buildBC ( char* P ) { //构造Bad Charactor Shift表：O(m + 256)
8     int* bc = new int[256]; //BC表，与字符表等长
9     for ( size_t j = 0; j < 256; j ++ ) bc[j] = -1; //初始化：首先假设所有字符均未在P中出现
10    for ( size_t m = strlen ( P ), j = 0; j < m; j ++ ) //自左向右扫描模式串P
11       bc[ P[j] ] = j; //将字符P[j]的BC项更新为j（单调递增）——画家算法
12    return bc;
13 }
```

代码11.7 BC表的构造

该算法在对BC初始化之后，对模式串P做一遍线性扫描，并不断用当前字符的秩更新BC表中的对应项。因为是按秩递增的次序从左到右扫描，故只要字符c在P中出现过，则最终的bc[c]必将如我们的所期望的那样，记录下其中最靠右者的秩。

若将BC表比作一块画布，则其中各项的更新过程，就犹如画家在不同位置堆积不同的油彩。而画布上各处最终的颜色，仅取决于在对应位置所堆积的最后一笔——这类算法，也因此称作“画家算法”（painter's algorithm）。

代码11.7的运行时间可划分为两部分，分别消耗于其中的两个循环。前者是对字符表Σ中的每个字符分别做初始化，时间量不超过$O(|\Sigma|)$。后一循环对模式串P做一轮扫描，其中每个字符消耗$O(1)$时间，故共需$O(m)$时间。由此可知，BC表可在$O(|\Sigma| + m)$时间内构造出来，其中$|\Sigma|$为字符表的规模，m为模式串的长度。

■ 匹配实例

一次完整的查找过程，如图11.11所示。这里的文本串T长度为12(b)，模式串P长度为4(a)。模式串P中各字符所对应的bc[]表项，如图(a)所示。

因这里的字符表涵盖常用的汉字，规模很大，故为节省篇幅，除了模式串所含的四个字符，其余大量字符的bc[]表项均默认统一为-1，在此不再逐个标出。

以下，首先如图(c1)所示，在第一个对齐位置，经1次后比较发现P[3] = '常' ≠ '非' = T[3]。于是如图(c2)所示，将P[bc['非']] = P[2]与T[3]对齐，并经3次比较后发现P[1] = '名' ≠ '道' = T[2]。于是如图(c3)所示，将P[bc['道']] = P[-1]与T[2]对齐，并经1次比较发现P[3] = '常' ≠ '名' = T[6]。于是如图(c4)所示，将P[bc['名']] = P[1]与T[6]对齐，并经过1次比较发现P[3] = '常' ≠ '名' = T[8]。最后如图(c5)所示，将P[bc['名']] = P[1]与T[8]对齐，并经4次比较后匹配成功。

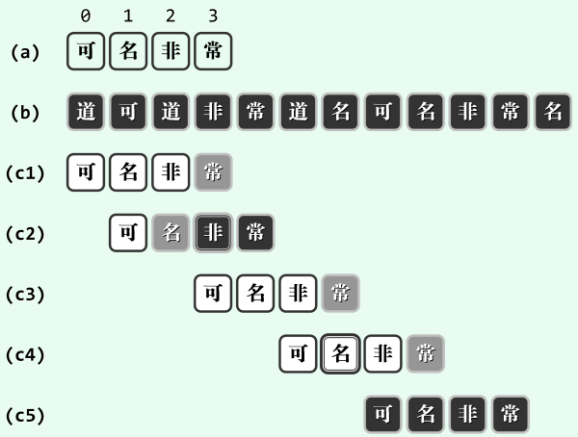

图11.11 借助bc[]表的串匹配

可见，整个过程中总共做过6次成功的（黑色字符）和4次失败的（灰色字符）比较，累计10次，文本串的每个有效字符平均为10/11不足一次。

■ 复杂度

若暂且不计构造BC表的过程，BM算法本身进行串模式匹配所需的时间与具体的输入十分相关。若将文本串和模式串的长度分别记作n和m，则在通常情况下的实际运行时间往往低于$O(n)$。而在最好的情况下，每经过常数次比对，BM算法就可以将模式串右移m个字符（即整体右移）。比如，图11.2中蛮力算法的最坏例子，却属于BM算法的最好情况。此类情况下，只需经过n / m次比对算法即可终止，故运行时间不超过$O(n / m)$。

反之，若如图11.3模式串P左右颠倒，则在每一轮比对中，P总要完整地扫描一遍才发现失配并向右移动一个字符。此类情况下的总体运行时间将为$O(n \times m)$，属于最坏情况。

11.4.3 好后缀策略

■ 构思

上述基于坏字符的启发策略，充分体现了“将教训转化为预知力”的构思：一旦发现P[j]与T[i + j]失配，就将P与T重新对齐于至少可使T[i + j]恢复匹配（含通配）的位置。然而正如上例所揭示的，这一策略有时仍显得不够“聪明”，计算效率将退化为几乎等同于蛮力算法。

参照KMP算法的改进思路不难发现，坏字符策略仅利用了此前（最后一次）失败比对所提供的“教训”。而实际上在此之前，还做过一系列成功的比对，而这些“经验”却被忽略了。

回到如图11.3所示的最坏情况，每当在P[0] = '1' ≠ '0'处失配，自然首先应该考虑将其替换为字符'0'（或通配符）。但既然本轮比对过程中已有大量字符'0'的成功匹配，则无论将P[0]对准其中的任何一个都注定会失配。故此时更明智地，应将P整体“滑过”这段区间，直接以P[0]对准T中尚未接受比对的首个字符。果真如此，算法的运行时间将有望降回至$O(n)$！

■ 好后缀

每轮比对中的若干次（连续的）成功匹配，都对应于模式串P的一个后缀，称作“好后缀”（good suffix）。按照以上分析，必须充分利用好好后缀所提供的“经验”。

一般地，如图11.12(a)和(b)所示，设本轮自右向左的扫描终止于失配位置：

```
T[i + j]  =  'X'  ≠  'Y'  =  P[j]
```

若分别记

```
W  =  substr(T, i + j + 1, m - j - 1)  =  T[i + j + 1, m + i)
U  =  suffix(P, m - j - 1)  =  P[j + 1, m)
```

则U即为当前的好后缀，W为T中与之匹配的子串。

好后缀U长度为m - j - 1，故只要j ≤ m - 2，则U必非空，且有U = W。此时具体地：

根据好后缀所提供的信息应如何确定，P中有哪个（哪些）字符值得与上一失配字符T[i + j]对齐，然后启动下一轮比对呢？

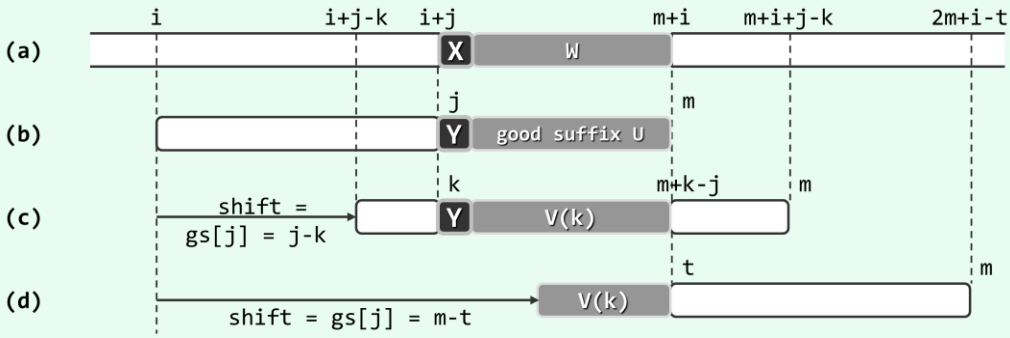

图11.12 好后缀策略：通过右移模式串P，使与P后缀U匹配的W重新得到匹配

如图11.12(c)所示，设存在某一整数k，使得在将P右移j - k个单元，并使P[k]与T[i + j]相互对齐之后，P能够与文本串T的某一（包含T[m + i - 1]在内的）子串匹配，亦即：

```
P  =  substr(T, i + j - k, m)  =  T[i + j - k, m + i + j - k)
```

于是，若记：

```
V(k)  =  substr(P, k + 1, m - j - 1)  =  P[k + 1, m - j + k)
```

则必然有：

V(k) = W = U

也就是说，若值得将P[k]与T[i + j]对齐并做新的一轮比对，则P的子串V(k)首先必须与P自己的后缀U相互匹配——这正是从好后缀中“挖掘”出来的“经验”。

此外，还有另一必要条件：P中这两个自匹配子串的前驱字符不得相等，即P[k] ≠ P[j]。否则，与第11.3.8节KMP算法的改进同理，在此对齐位置也注定不会出现与P的整体匹配。

当然，若模式串P中同时存在多个满足上述必要条件的子串V(k)，则不妨选取其中最靠右者（对应于最大的k、最小的右移距离j - k）。这一处理手法的原理，依然与KMP算法类似——如此既不致遗漏匹配位置，亦可保证始终单向地“滑动”模式串，而不致回退。

■ gs[]表

如图11.12(c)所示，若满足上述必要条件的子串V(k)起始于P[k + 1]，则模式串对应的右移量应就是j - k。表面上，此右移量同时取决于失配位置j以及k；然而实际上，k本身（也因此包括位移量j - k）仅取决于模式串P以及j值。因此可以仿照KMP算法的做法，通过预处理，将模式串P事先转换为另一张查找表gs[0, m)，其中gs[j] = j - k分别记录对应的位移量。

如图11.12(d)所示，若P中没有任何子串V(k)可与好后缀U完全匹配呢？此时需从P的所有前缀中，找出可与U的某一（真）后缀相匹配的最长者，作为V(k)，并取gs[j] = m - |V(k)|。

表11.7 模式串P = "ICED RICE PRICE"对应的GS表

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P[j]	I	C	E	D	□	R	I	C	E	□	P	R	I	C	E
gs[j]	12	12	12	12	12	12	12	12	12	12	6	12	15	15	1

考查如表11.7所示的实例。其中的gs[10] = 6可理解为：一旦在P[10] = 'P'处发生失配，则应将模式串P右移6个字符，即用P[10 - 6] = P[4] = '□'对准文本串T的失配字符，然后启动下一轮比对。类似地，gs[5] = 12意味着：一旦在P[5] = 'R'处发生失配，则应将模式串P整体右移12个字符，然后继续启动下一轮比对。当然，也可以等效地认为，以P[5 - 12] = P[-7]对准文本串中失配的字符，或以P[0]对准文本串中尚未对准过的最左侧字符。

■ 匹配实例

基于好后缀策略的匹配实例，如图11.13所示。首先如图(c1)所示，在第一个对齐位置，经1次比较发现：

P[7] = '也' ≠ '静' = T[7]

于是如图(c2)所示，将P右移gs[7] = 1位，经3次比较发现：

P[5] = '故' ≠ '曰' = T[6]

于是如图(c3)所示，将P右移gs[5] = 4位，经8次比较后匹配成功。

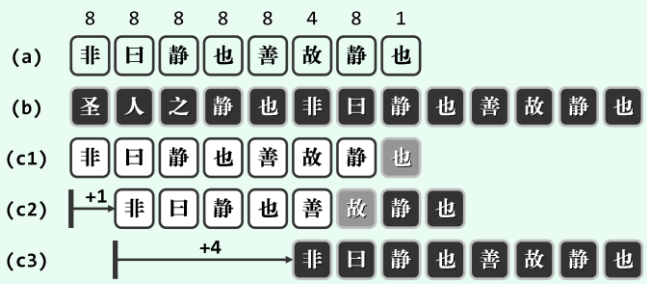

图11.13 借助gs[]表的串匹配：
(a) 模式串P及其gs[]表；(b) 文本串T

可见，整个过程中总共做10次成功的（黑色字符）和2次失败的（灰色字符）比较，累计12次比较。文本串的每个字符，平均（12/13）不足一次。

■ 复杂度

如317页代码11.6所示，可以同时结合以上BC表和GS表两种启发策略，加快模式串相对于文本串的右移速度。可以证明，对于匹配失败的情况，总体比对的次数不致超过$O(n)$[60][62][63]。

若不排除完全匹配的可能，则该算法在最坏情况下的效率，有可能退化至与蛮力算法相当。所幸，只要做些简单的改进，依然能够保证总体的比对次数不超过线性（习题[11-7]）。

综上所述，在兼顾了坏字符与好后缀两种策略之后，BM算法的运行时间为$O(n + m)$。

11.4.4 gs[]表构造算法

■ 蛮力算法

根据以上定义，不难直接导出一个构造gs[]表的“算法”：对于每个好后缀P(j, m)，按照自后向前（k从j - 1递减至0）的次序，将其与P的每个子串P(k, m + k - j)逐一对齐，并核对是否出现如图11.12(c~d)所示的匹配。一旦发现匹配，对应的位移量即是gs[j]的取值。

然而遗憾的是，这里共有$O(m)$个好后缀，各需与$O(m)$个子串对齐，每次对齐后在最坏情况下都需要比对$O(m)$次，因此该“算法”可能需要$O(m^3)$的时间。

实际上，仅需线性的时间即可构造出gs[]表（习题[11-6]）。为此，我们需要引入ss[]表。

■ MS[]串与ss[]表

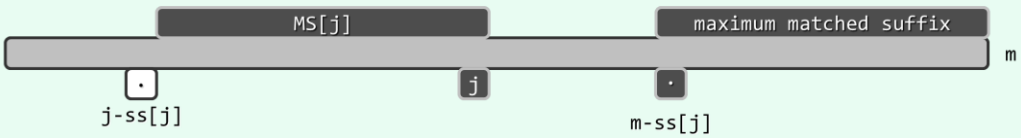

图11.14 MS[j]和ss[j]表的定义与含义

如图11.14所示，对于任一整数j ∈ [0, m)，在P[0, j]的所有后缀中，考查那些与P的某一后缀匹配者。若将其中的最长者记作MS[j]，则ss[j]就是该串的长度|MS[j]|。特别地，当MS[j]不存在时，取ss[j] = 0。

综上所述，可定义ss[j]如下：

$$ss[j] = \max\{\ 0 \le s \le j + 1 \ \mid\ P(j - s, j] = P[m - s, m)\ \}$$

特别地，当j = m - 1时，必有s = m——此时，有P(-1, m - 1] = P[0, m)。

■ 实例

表11.8 模式串P = "ICED RICE PRICE"对应的SS表

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P[i]	I	C	E	D	□	R	I	C	E	□	P	R	I	C	E
ss[i]	0	0	3	0	0	0	0	0	4	0	0	0	0	0	15

仍以表11.7中的模式串P为例，按照如上定义，P所对应的ss[]表应如表11.8所示。

比如，其中之所以有ss[8] = 4，是因为若取j = 8和s = 4，则有：

P(8 - 4, 8] = P(4, 8] = "RICE" = P[11, 15) = P[15 - 4, 15)

实际上，ss[]表中蕴含了gs[]表的所有信息，由前者足以便捷地构造出后者。

■ 由ss[]表构造gs[]表

如图11.15所示，任一字符P[j]所对应的ss[j]值，可分两种情况提供有效的信息。

第一种情况如图(a)所示，设该位置j满足：

$$ss[j] = j + 1$$

也就是说，MS[j]就是整个前缀P[0, j]。此时，对于P[m - j - 1]左侧的每个字符P[i]而言，对应于如图11.12(d)所示的情况，m - j - 1都应该是gs[i]取值的一个候选。

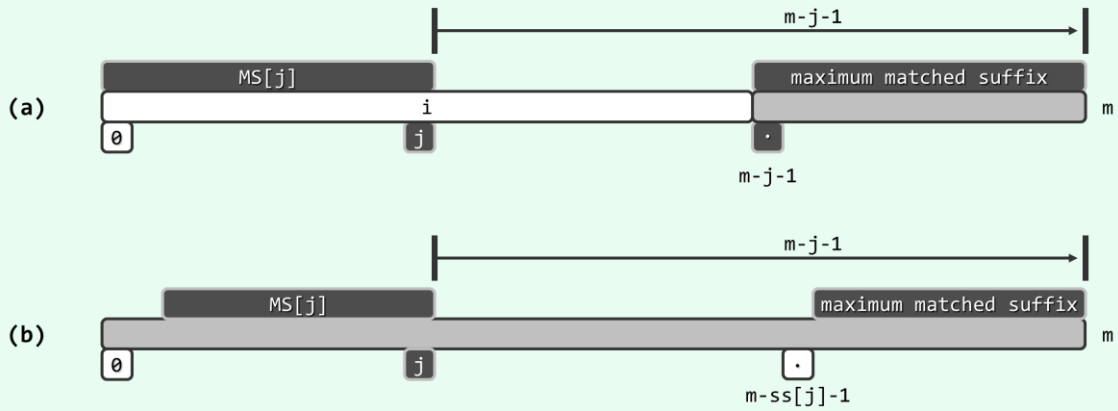

图11.15 由ss[]表构造gs[]表

第二种情况如图(b)所示，设该位置j满足：

$$ss[j] \leq j$$

也就是说，MS[j]只是P[0, j]的一个真后缀。此时，对于字符P[m - ss[j] - 1]而言，对应于如图11.12(c)所示的情况，若同时还满足：

$$P[m - ss[j] - 1] \neq P[j - ss[j]]$$

则m - j - 1也应是gs[m - ss[j] - 1]取值的一个候选。

反过来，根据此前所做的定义，每一位置i所对应的gs[i]值只可能来自于以上候选。进一步地，既然gs[i]的最终取值是上述候选中的最小（最安全）者，故仿照构造bc[]表的画家算法，累计用时将不超过$O(m)$（习题[11-6]）。

■ ss[]表的构造

由上可见，ss[]表的确是构造gs[]表的基础与关键。同样地，若采用蛮力策略，则对每个字符P[j]都需要做一趟扫描对比，直到出现失配。如此，累计需要$O(m^2)$时间。

为了提高效率，我们不妨自后向前地逆向扫描，并逐一计算出各字符P[j]对应的ss[j]值。如图11.16所示，因此时必有P[j] = P[m - hi + j - 1]，故可利用此前已计算出的ss[m - hi + j - 1]，分两种情况快速地导出ss[j]。在此期间，只需动态地记录当前的极长匹配后缀：

$$P(lo, hi] = P[m - hi + lo, m)$$

第一种情况如图(a)所示，设：

$$ss[m - hi + j - 1] \leq j - lo$$

此时，ss[m - hi + j - 1]也是ss[j]可能的最大取值，于是便可直接得到：

$$ss[j] = ss[m - hi + j - 1]$$

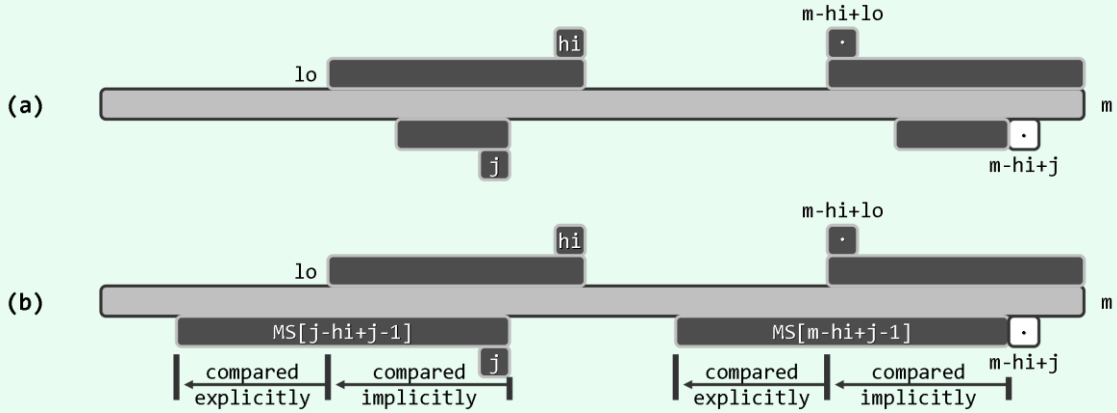

图11.16 构造ss[]表

第二种情况如图(b)所示，设：

```
j - lo  <  ss[m - hi + j - 1]
```

此时，至少仍有：

```
P(lo, j]  =  P[m - hi + lo, m - hi + j)
```

故只需将

```
P(j - ss[m - hi + j - 1], lo]
```

与

```
P[m - hi + j - ss[m - hi + j - 1], m - hi + lo)
```

做一比对，也可确定ss[j]。当然，这种情况下极大匹配串的边界lo和hi也需相应左移（递减）。

同样地，以上构思只要实现得当，也只需$O(m)$时间即可构造出ss[]表（习题[11-6]）。

■ 算法实现

按照上述思路，GS表的构造算法可实现如代码11.8所示。

```
1  int* buildSS ( char* P ) { //构造最大匹配后缀长度表：O(m)
2     int m = strlen ( P ); int* ss = new int[m]; //Suffix Size表
3     ss[m - 1]  =  m; //对最后一个字符而言，与之匹配的最长后缀就是整个P串
4  // 以下，从倒数第二个字符起自右向左扫描P，依次计算出ss[]其余各项
5     for ( int lo = m - 1, hi = m - 1, j = lo - 1; j >= 0; j -- )
6        if ( ( lo < j ) && ( ss[m - hi + j - 1] <= j - lo ) ) //情况一
7           ss[j] =  ss[m - hi + j - 1]; //直接利用此前已计算出的ss[]
8        else { //情况二
9           hi = j; lo = __min ( lo, hi );
10          while ( ( 0 <= lo ) && ( P[lo] == P[m - hi + lo - 1] ) ) //二重循环？
11             lo--; //逐个对比处于(lo, hi]前端的字符
12          ss[j] = hi - lo;
13       }
14    return ss;
15 }
16
```

```
17 int* buildGS ( char* P ) { //构造好后缀位移量表：O(m)
18    int* ss = buildSS ( P ); //Suffix Size table
19    size_t m = strlen ( P ); int* gs = new int[m]; //Good Suffix shift table
20    for ( size_t j = 0; j < m; j ++ ) gs[j] = m; //初始化
21    for ( size_t i = 0, j = m - 1; j < UINT_MAX; j -- ) //逆向逐一扫描各字符P[j]
22       if ( j + 1 == ss[j] ) //若P[0, j] = P[m - j - 1, m)，则
23          while ( i < m - j - 1 ) //对于P[m - j - 1]左侧的每个字符P[i]而言（二重循环？）
24             gs[i++] = m - j - 1; //m - j - 1都是gs[i]的一种选择
25    for ( size_t j = 0; j < m - 1; j ++ ) //画家算法：正向扫描P[]各字符，gs[j]不断递减，直至最小
26       gs[m - ss[j] - 1] = m - j - 1; //m - j - 1必是其gs[m - ss[j] - 1]值的一种选择
27    delete [] ss; return gs;
28 }
```

代码11.8 GS表的构造

11.4.5 算法纵览

■ 时间效率的变化范围

以上我们针对串匹配问题，依次介绍了蛮力、KMP、基于BC表、综合BC表与GS表等四种典型算法，其渐进复杂度的跨度范围，可概括如图11.17所示。

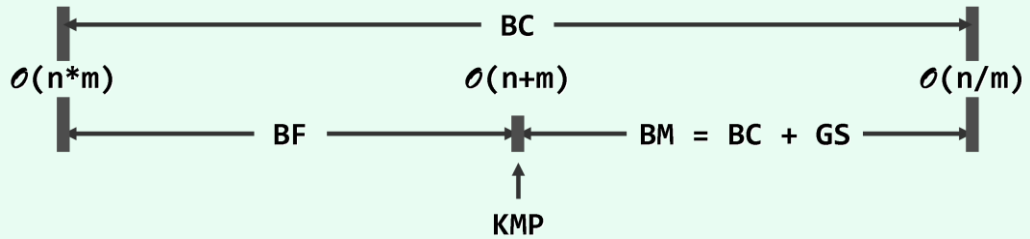

图11.17 典型串匹配算法的复杂度概览

其中，蛮力（BF）算法的时间效率介于$O(n * m)$至$O(n + m)$之间，而且其最好情况与KMP算法相当！当然，后者的优势在于，无论何种情况，时间效率均稳定在$O(n + m)$。因此在蛮力算法效率接近或达到最坏的$O(n * m)$时，KMP算法的优势才会十分明显。

仅采用坏字符启发策略（BC）的BM算法，时间效率介于$O(n * m)$至$O(n / m)$之间。可见，其最好情况与最坏情况相差悬殊。结合了好后缀启发策略（BC + GS）后的BM算法，则介于$O(n + m)$和$O(n / m)$之间。可见，在改进最低效率的同时，保持了最高效率的优势。

■ 单次比对成功概率

饶有意味的是，单次比对成功的概率，是决定串匹配算法时间效率的一项关键因素。

纵观以上串匹配算法，在每一对齐位置所进行的一轮比对中，仅有最后一次可能失败；反之，此前的所有比对（若的确进行过）必然都是成功的。反观诸如图11.2、图11.3的实例可见，各种算法的最坏情况均可概括为：因启发策略不够精妙甚至不当，在每一对齐位置都需进行多达$\Omega(m)$次成功的比对（另加最后一次失败的比对）。

若将单次比对成功的概率记作Pr，则以上算法的时间性能随Pr的变化趋势，大致如图11.18

所示。其中纵坐标为运行时间，分为$O(n / m)$、$O(n + m)$和$O(n * m)$三档——当然，此处只是大致示意，实际的增长趋势未必是线性的。

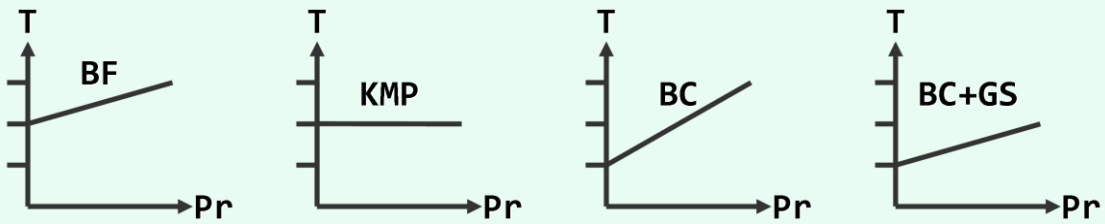

图11.18 随着单次比对成功概率（横轴）的提高，串匹配算法的运行时间（纵轴）通常亦将增加

可见，对于同一算法，计算时间与Pr具有单调正相关关系——这一点不难理解，正如以上分析，消耗于每一对齐位置的平均时间成本随Pr的提高而增加。

■ 字符表长度

实际上，在所有字符均等概率出现的情况下，Pr的取值将主要决定于字符表的长度|Σ|，并与之成反比关系：字符表越长，其中任何一对字符匹配的概率越低。

这一性质可用以解释：在通常的情况下，蛮力算法实际的运行效率并不算太低（习题[11-9]）；不同的串匹配算法，因何各自有其适用的场合（习题[11-10]）。

§11.5 *Karp-Rabin算法

11.5.1 构思

■ 凡物皆数

早在公元前500年，先贤毕达哥拉斯及其信徒即笃信"凡物皆数"[④]。近世以来，以克罗内克[⑤]为代表的构造主义数学家曾坚定地认为，唯有可直接构造的自然数才是万物的本源。而此后无论是康托[⑥]还是哥德尔[⑦]，都以他们杰出的发现，为这一思想添加了生动的注脚。

其实，即便是限于本书所涉及和讨论的计算机科学领域，循着这一思路也可导出优美、简洁和高效的数据结构及算法。比如，细细品味第9章后不难领悟到，散列技术亦可视作为这一思想的产物。从这一角度来看，散列之所以可实现极高的效率，正在于它突破了通常对关键码的狭义理解——允许操作对象不必支持大小比较——从而在一般类型的对象（词条）与自然数（散列地址）之间，建立起直接的联系。

那么，这一构思与技巧，可否转而运用于本章讨论的主题呢？答案是肯定的。

■ 串亦为数

为此，可以效仿康托的思路，将任一有限长度的整数向量视作自然数，进而在字符串与自然数之间建立联系。

④ "All things are numbers.", Pythagoras (570 ~ 495 B.C.)

⑤ "God made the integers; all else is the work of man.", L. Kronecker (1823 ~ 1891)

⑥ Geogr Cantor (1845 ~ 1918)

⑦ Kurt Godel (1906 ~ 1978)

若字母表规模|Σ| = d，则任一字符串都将对应于一个d + 1进制[®]的整数。以由大写英文字母组成的字母表为例，若将这些字符依次映射为[1, 26]内的自然数，则每个这样的字符串都将对应于一个26 + 1 = 27进制的整数，比如：

"CANTOR" = <3, 1, 14, 20, 15, 18>$_{(27)}$ = 43,868,727$_{(10)}$

"DATA" = <4, 1, 20, 1>$_{(27)}$ = 80002$_{(10)}$

从算法的角度来看，这一映射关系就是一个不折不扣的散列。

11.5.2 算法与实现

■ 算法

以上散列并非满射，但不含'0'的任一d + 1进制值自然数，均唯一地对应于某个字符串，故它几乎已是一个完美的算法。字符串经如此转换所得的散列码，称作其指纹（fingerprint）。

按照这一理解，“判断模式串P是否与文本串T匹配”的问题，可以转化为“判断T中是否有某个子串与模式串P拥有相同的指纹”的问题。具体地，只要逐一取出T中长度为m的子串，并将其对应的指纹与P所对应的指纹做一比对，即可确定是否存在匹配位置——这已经可以称作一个串匹配算法了，并以其发明者姓氏命名为Karp-Rabin算法。

该算法相关的预定义如代码11.9所示。这里仅考虑了阿拉伯数字串，故每个串的指纹都已一个R = 10进制数。同时，使用64位整数的散列码。

```
1 #define M 97 //散列表长度：既然这里并不需要真地存储散列表，不妨取更大的素数，以降低误判的可能
2 #define R 10 //基数：对于二进制串，取2；对于十进制串，取10；对于ASCII字符串，取128或256
3 #define DIGIT(S, i) ( (S)[i] - '0' ) //取十进制串S的第i位数字值（假定S合法）
4 typedef __int64 HashCode; //用64位整数实现散列码
5 bool check1by1 ( char* P, char* T, size_t i );
6 HashCode prepareDm ( size_t m );
7 void updateHash ( HashCode& hashT, char* T, size_t m, size_t k, HashCode Dm );
```

代码11.9 Karp-Rabin算法相关的预定义

算法的主体结构如代码11.10所示。除了预先计算模式串指纹hash(P)等预处理，至多包含|T| - |P| = n - m轮迭代，每轮都需计算当前子串的指纹，并与目标指纹比对。

```
1 int match ( char* P, char* T ) { //串匹配算法（Karp-Rabin）
2    size_t m = strlen ( P ), n = strlen ( T ); //assert: m <= n
3    HashCode Dm = prepareDm ( m ), hashP = 0, hashT = 0;
4    for ( size_t i = 0; i < m; i++ ) { //初始化
5       hashP = ( hashP * R + DIGIT ( P, i ) ) % M; //计算模式串对应的散列值
6       hashT = ( hashT * R + DIGIT ( T, i ) ) % M; //计算文本串（前m位）的初始散列值
7    }
8    for ( size_t k = 0; ; ) { //查找
```


[®] 之所以取d + 1而不是d，是为了回避'0'字符以保证这一映射是单射
否则若字符串中存在由'0'字符组成的前缀，则无论该前缀长度任何，都不会影响对应的整数取值

```
9       if ( hashT == hashP )
10         if ( check1by1 ( P, T, k ) ) return k;
11      if ( ++k > n - m ) return k; //assert: k > n - m，表示无匹配
12      else updateHash ( hashT, T, m, k, Dm ); //否则，更新子串散列码，继续查找
13   }
14 }
```

代码11.10 Karp-Rabin算法主体框架

请注意，这里并不需要真正地设置一个散列表，故空间复杂度与表长M无关。

■ 数位与字长

然而就效率而言，将上述方法称作算法仍嫌牵强。首先，直接计算各子串的指纹十分耗时。仍以上述大写英文字符母表为例，稍长的字符串就可能对应于数值很大的指纹，比如：

"HASHING" = <8, 1, 19, 8, 9, 14, 7>$_{(27)}$ = 3,123,974,608$_{(10)}$

"KARPRABIN" = <11, 1, 18, 16, 18, 1, 2, 9, 14>$_{(27)}$ = 3,124,397,993,144$_{(10)}$

另一方面，随着字母表规模d的增大，指纹的位数也将急剧膨胀。以d = 128 = 2^7的ASCII字符集为例，只要模式串长度m = |P| ≥ 10，其指纹的长度就会达到$m \cdot \log_2 d = 70$个比特，从而超出目前通常支持的32 ~ 64位字长。这就意味着，若指纹持续加长，即便不考虑存储所需的空间字长而仅就时间成本而言，无论是指纹的计算还是指纹的比对，都无法在$O(1)$时间内完成。确切地说，这些操作所需的时间都将线性正比于模式串长度m。于是整个“算法”的时间复杂度将高达$O(n * m)$——退回到11.2节的蛮力算法。

■ 散列压缩

不妨暂且搁置指纹的快速计算问题，首先讨论指纹的快速比对。既然上述指纹完全等效于字符串的散列码，上述问题也就与我们在9.3.2节中所面临的困境类似——若不能对整个散列空间进行有效的压缩，则以上方法将仅停留于朴素的构思，而将无法兑现为实用的算法。

仿照9.3.3节的思路和方法，这里不妨以除余法为示例，通过散列函数hash(key) = key % M，将指纹的数值压缩至一个可以接受的范围。以十进制数字串为例，字母表规模d = 10。

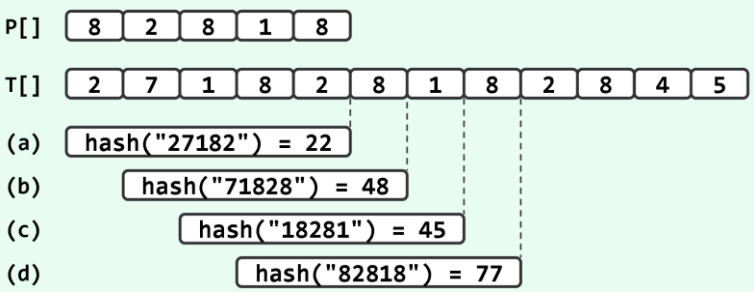

图11.19 Karp-Rabin串匹配算法实例：模式串指纹hash("82818") = 82,818 % 97 = 77

例如，如图11.19所示即为散列表长选作M = 97时，一次完整的匹配过程。

首先经预处理，提前计算出模式串P的指纹hash("82818") = 77。

此后，自左向右地依次取出文本串T中长度为m的各个子串，计算其指纹并与上述指纹对比。由图11.19可见，经过三次比对失败，最终确认匹配于substr(T, 3, 5) = P。

可见，经散列压缩之后，指纹比对所需的时间将仅取决于散列表长M，而与模式串长m无关。

■ 散列冲突

压缩散列空间的同时，必然引起冲突。就Karp-Rabin算法而言这体现为，文本串中不同子串的指纹可能相同，甚至恰好都与模式串的指纹相同。

仍考查以上实例，但如图11.20所示改换为P = "18284"，其指纹hash("18284") = 48。

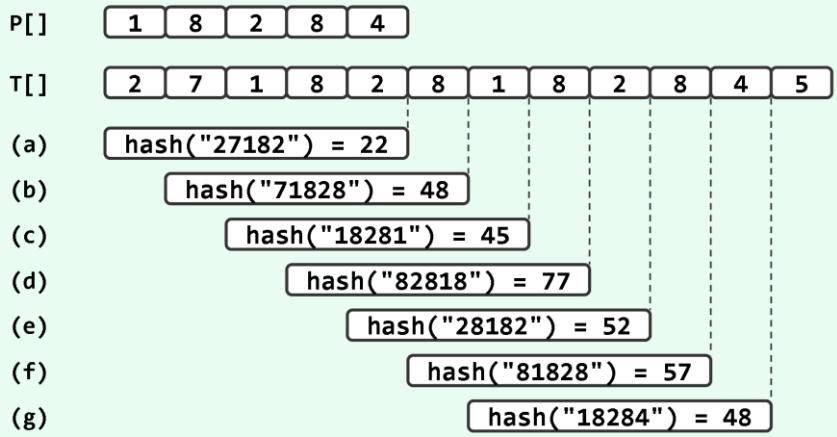

图11.20 Karp-Rabin串匹配算法实例：模式串指纹hash("18284") = 18,284 % 97 = 48

于是，尽管在第二次指纹比对时（图(b)）即发现hash("71828") = 48，与模式串的指纹相同，但真正的匹配却应该在第七次比对后（图(g)）才能确认。

既然指纹相同并不是匹配的充分条件，故在发现指纹相等之后，还必须如代码11.11所示，对原字符串做一次严格的逐位比对。

```
1 bool check1by1 ( char* P, char* T, size_t i ) { //指纹相同时，逐位比对以确认是否真正匹配
2    for ( size_t m = strlen ( P ), j = 0; j < m; j++, i++ ) //尽管需要O(m)时间
3       if ( P[j] != T[i] ) return false; //但只要散列得当，调用本例程并返回false的概率将极低
4    return true;
5 }
```

代码11.11 指纹相同时还需逐个字符地比对

尽管这种比对需耗时$O(m)$，但只要散列策略设计得当，即可有效地控制发生冲突以及执行此类严格比对的概率。以此处的除余法为例，若散列表容量选作M，则在“各字符皆独立且均匀分布”的假定条件下，指纹相同的可能性应为1/M；而随着M的增大，冲突的概率将急速下降。代码11.9中选取M = 97完全是出于演示的需要，实际应用中不妨适当地选用更长的散列表。

■ 快速指纹更新

最后，讨论快速指纹计算的实现。对图11.20等实例细加观察不难发现，按照自左向右的次序，任何两次相邻比对所对应的子串之间存在极强的相关性，子串的指纹亦是如此。

实际上，二者仅在首、末字符处有所出入。准确地如图11.21所示，前一子串删除首字符之后的后缀，与后一子串删除末字符之后的前缀完全相同。

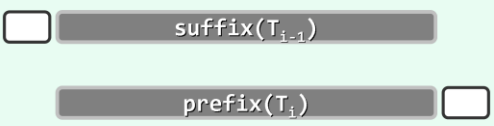

图11.21 相邻子串内容及指纹的相关性

利用这种相关性，可以根据前一子串的指纹，在常数时间内得到后一子串的指纹。也就是说，整个算法过程中消耗于子串指纹计算的时间，平均每次仅为$O(1)$。

该算法的具体实现，如代码11.12所示。

```
1 // 子串指纹快速更新算法
2 void updateHash ( HashCode& hashT, char* T, size_t m, size_t k, HashCode Dm ) {
3    hashT = ( hashT - DIGIT ( T, k - 1 ) * Dm ) % M; //在前一指纹基础上，去除首位T[k - 1]
4    hashT = ( hashT * R + DIGIT ( T, k + m - 1 ) ) % M; //添加末位T[k + m - 1]
5    if ( 0 > hashT ) hashT += M; //确保散列码落在合法区间内
6 }
```

代码11.12 串指纹的快速更新

这里，前一子串最高位对指纹的贡献量应为$P[0] \times M^{m-1}$。只要注意到其中的M^{m-1}始终不变，即可考虑如代码11.13所示，通过预处理提前计算出其对应的模余值。

为此尽管可采用代码1.8中的快速幂算法power2()，但考虑到此处仅需调用一次，同时兼顾算法的简洁性，故不妨直接以蛮力累乘的形式实现。

```
1 HashCode prepareDm ( size_t m ) { //预处理：计算R^(m - 1) % M （仅需调用一次，不必优化）
2    HashCode  Dm = 1;
3    for ( size_t i = 1; i < m; i++ ) Dm = ( R * Dm ) % M; //直接累乘m - 1次，并取模
4    return Dm;
5 }
```

代码11.13 提前计算M^(m-1)