# 第一部分 字典树

核心要旨：  
trie建立的过程就是给各个状态字串确定编号的过程，而某个状态的所有后继状态的**编号分布与各变量编码分布平行**；所以需要基址去偏移到一个足够**平行**的地方。  
tire建立后查找的过程就是从根节点如肉眼匹配的过程，而不是树的什么广度、深度的查找。

## 1.1 Trie创建（编号的过程）

确定有限自动机（deterministic finite automaton ，DFA）：所谓“确定有限自动机”是指给定一个状态和一个变量时，它能跳转到的下一个状态也就确定下来了，同时状态是有限的。

变量：激发状态转移的值  
状态：理解为从根节点到当前节点所有变量的顺序序列串  
举个例子，假设我们一共有10个汉字，每个汉字就是一个“变量”。我们为每个汉字编码。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 啊 | 阿 | 埃 | 根 | 胶 | 拉 | 及 | 廷 | 伯 | 人 |

表1. “变量”的编码

这10个汉字一共可以构成6个词语（本身就是六个状态）：啊，埃及，阿胶，阿根廷，阿拉伯，阿拉伯人。  
这里的每个词以及它的任意前缀都是一个“状态”，“状态”一共有10个：啊,阿,埃,阿根,阿根廷,阿胶,阿拉,阿拉伯,阿拉伯人,埃及。

我们把DFA图画出来:

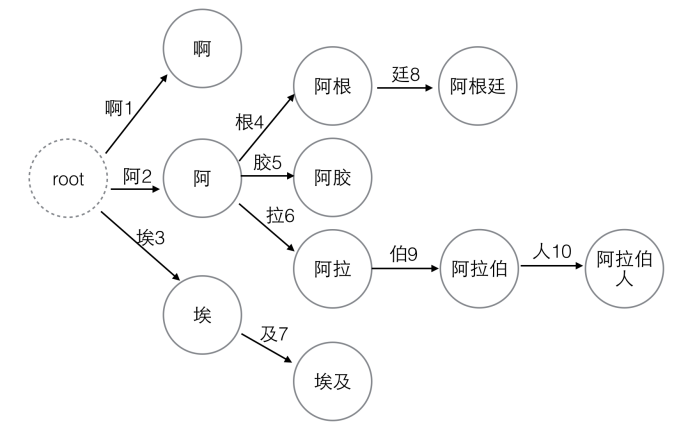


图1. DFA，同时也是Trie树

在图中每个节点代表一个“状态”，每条边代表一个“变量”，并且我们把变量的编码也标在了图中。状态的转换时因为变量的加入。  
下面我们构造两个int数组：base和check，它们的长度始终是一样的。数组的长度定多少并没有严格的规定，反正随着词语的插入，数组肯定是要扩容的。  
回到正题，我们不妨把double array的初始长度就定得大一些。两数组元素初始值均为0。  
double array的初始状态：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态编号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| base | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| check | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 状态字串 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

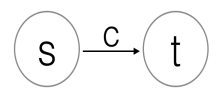
把词添加到词典的过程就给base和check数组中各元素赋值的过程。下面我们层次遍历图1所示的Trie树。

### **step1.第一层节点入树**

第一层上取到3个“状态”：啊,阿,埃。把这3个状态按照其对应的状态编号（可查表1，因为对于顶层状态从空状态自动转换而来，顶层的状态编号等于变量编码）放到state数组中。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态编号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| base | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| check | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 状态字串 | 啊 | 阿 | 埃 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

### **step2.第2层状态入树**

当存在状态转移

南无表达式成立  
其中s和t代表某个状态在数组中的下标，c代表变量的编号。

此时层次遍历来到了图1所示DFA的第二层，我们看到“阿”的子节点有“阿根”、“阿胶”、“阿拉”，已知状态“阿”的状态是2，变量“根”、“胶”、“拉”的编码依次是4、5、6，下面我们要给base[2]赋值：从小到大遍历所有的正整数，直到发现某个数正整k满足：base[k+4]=base[k+5]=base[k+6]=check[k+4]=check[k+5]=check[k+6]=0。  
得到k=1，那么就把1赋给base[2]，同时也确定了状态“阿根”、“阿胶”、“阿拉”的下标依次是k+4、k+5、k+6，即5、6、7，而且check[5]=check[6]=check[7]=2。

同理，“埃”的子节点是“埃及”，状态“埃”的编码是3，变量“及”的编码是7，此时有check[1+7]=base[1+7]=0，所以base[3]=1，状态“埃及”的状态是8，check[8]=3。  
遍历完DFA的第二层后得到下表：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态编号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| base | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| check | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 状态字串 | 啊 | 阿 | 埃 |  | 阿 根 | 阿 胶 | 阿 拉 | 埃 及 |  |  |  |  |  |  |  |  |  |  |  |

### **step3.第n层状态入树**

重复step2，层次遍历完整查询树之后，得到：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态编号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| base | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| check | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 3 | 5 | 7 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 状态字串 | 啊 | 阿 | 埃 |  | 阿 根 | 阿 胶 | 阿 拉 | 埃 及 | 阿 根 廷 | 阿 拉 伯 | 阿 拉 伯 人 |  |  |  |  |  |  |  |  |

### **step4.借助基址标注单词**

最后遍历一次DFA，当某个状态已无后续状态（是一个词的结尾时），按下列方法修改其base值。

|  |  |
| --- | --- |
| 1  2  3  4 | if(base[i]==0)      base[i]=-i  else      base[i]=-base[i] |

得到：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态编号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| base | -1 | 1 | 1 | 0 | 1 | -6 | 1 | -8 | -9 | -1 | -11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| check | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 3 | 5 | 7 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 状态字串 | 啊 | 阿 | 埃 |  | 阿 根 | 阿 胶 | 阿 拉 | 埃 及 | 阿 根 廷 | 阿 拉 伯 | 阿 拉 伯 人 |  |  |  |  |  |  |  |  |

double array建好之后，如果词典中又动态地添加了一个新词，比如“阿拉根”，那么“阿拉”的所有子孙节点在double array中的位置要重新分配。

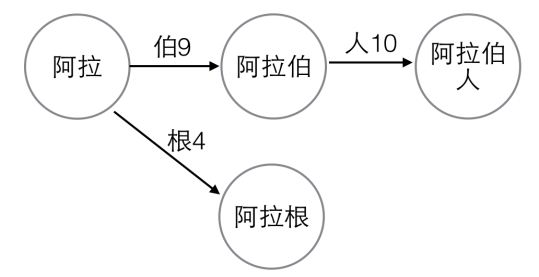


图2. 动态添加一个词

首先，把“阿拉伯”和“阿拉伯人”对应的base、check值清0，把“阿拉伯”和“阿拉伯人”从state数组中删除掉，把“阿拉”的base值清0。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态编号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| base | -1 | 1 | 1 | 0 | 1 | -6 | 0 | -8 | -9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| check | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 3 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 状态字串 | 啊 | 阿 | 埃 |  | 阿根 | 阿胶 | 阿拉 | 埃及 | 阿根廷 |  |  |  |  |  |  |  |  |  |  |

然后，按照上面step2所述的方法把“阿拉伯”、“阿拉根”插入到double array中。变量“根”、“伯”的编号是4和9，满足base[k+4]=base[k+9]=check[k+4]=check[k+9]=0的最小的k是6，所以base[7]=6，“阿拉伯”和“阿拉根”对应的下标是10和15。同理把“阿拉伯人”插入到double array中。

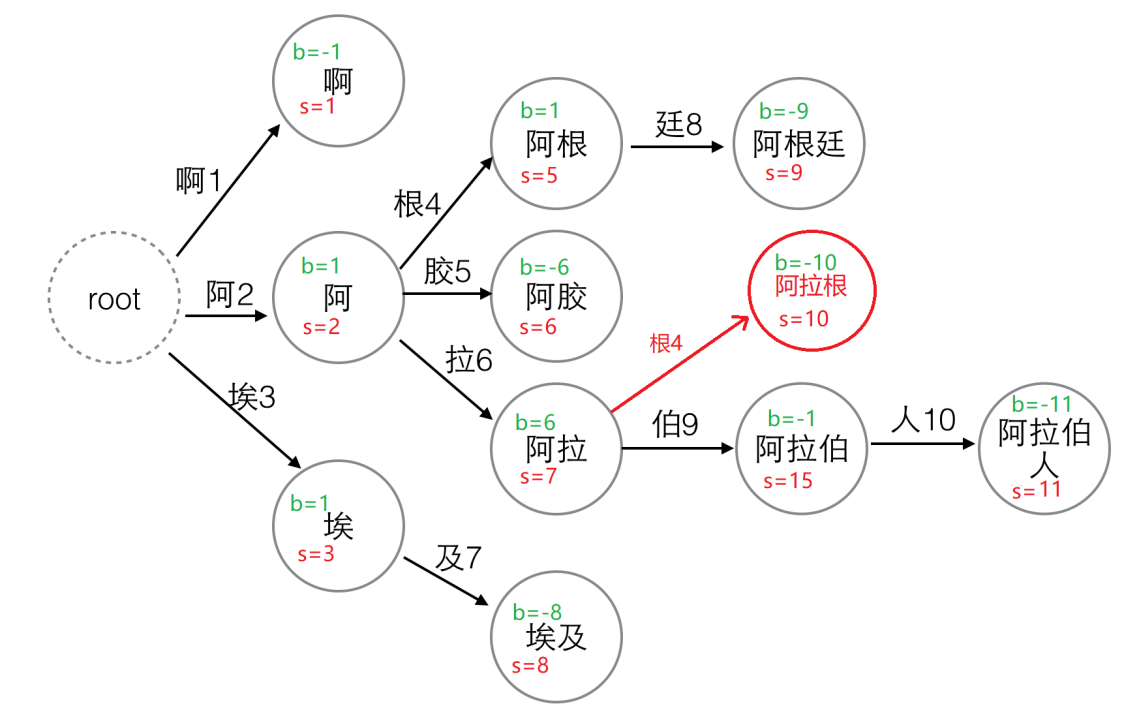
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态编号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| base | -1 | 1 | 1 | 0 | 1 | -6 | 6 | -8 | -9 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| check | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 3 | 5 | 7 | 15 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 |
| 状态字串 | 啊 | 阿 | 埃 |  | 阿根 | 阿胶 | 阿拉 | 埃及 | 阿根廷 | 阿拉根 | 阿拉伯人 |  |  |  | 阿拉伯 |  |  |  |  |

最后，遍历图2所示的DFA，当某个节点已经是一个词的结尾时按照step4中的方法修改其base值。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态编号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| base | -1 | 1 | 1 | 0 | 1 | -6 | 6 | -8 | -9 | -10 | -11 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 0 |
| check | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 3 | 5 | 7 | 15 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 |
| 状态字串 | 啊 | 阿 | 埃 |  | 阿根 | 阿胶 | 阿拉 | 埃及 | 阿根廷 | 阿拉根 | 阿拉伯人 |  |  |  | 阿拉伯 |  |  |  |  |

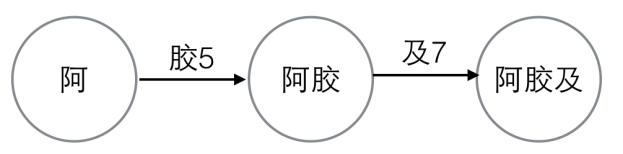
**核心要旨：**阿拉后续状态**阿拉伯、阿拉根状态的分布（10+（0+5））与根、伯的编码分布（4+（0+5））平行，这样才能存在一个基值6。**

最后，构建出有状态编号和基址的树如下，使用**基址、状态编号、变量编码**就能快速的判断某个输入的状态是否存于该树；简单得如同查表一样。



## 1.2 查找（匹配的过程）

double array建好之后，如何查询一个词是否在词典中呢？  
比如要查“阿胶及”，每个字的编号是已知的，我们画出状态转移图。



变量“阿”的编号是2，base[2]=1，变量“胶”的编号是5，base[2]+5=6，我们检查一下check[6]是否等于2。check[6]确实等于2，则继续看下一个状态转移。同时我们发现base[6]是负数，这说明“阿胶”已经是一个完整的词了。

继续看下一个状态转移，base[6]=-6，负数取其相反数，base[6]=6，变量“及”的编号是7，base[6]+7=13，我们检查一下check[13]是否等于6，发现不满足，则“阿胶及”不是一个词，甚至都是不是任意一个词的前缀。

## 1.3 Double Array Trie的实现

dat的实现中还需要额外的信息来提高遍历速度（如同前缀Key的获取）。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态编号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |  |
| base | -1 | 1 | 1 | 0 | 1 | -6 | 6 | -8 | -9 | -10 | -11 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 0 | 为负数代表key完整，绝对值与状态编号不等代表有尾串 |
| check | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 3 | 5 | 7 | 15 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 |  |
| 状态字串 | 啊 | 阿 | 埃 |  | 阿根 | 阿胶 | 阿拉 | 埃及 | 阿根廷 | 阿拉根 | 阿拉伯人 |  |  |  | 阿拉伯 |  |  |  |  |  |
| 上边只是将字典树搭建起，但没有考虑字段数的遍历效率  如果没有下边的信息，在统计同前缀的Key的时候，需要递归遍历check相同的来寻找。 | | | | | | | | | | | | | | | | | | | |  |
| sibling | 阿 | 埃 | 0 |  | 胶 | 拉 | 0 |  | 0 | 0 | 0 |  |  |  | 根 |  |  |  |  |  |
| child |  | 根 | 及 |  | 廷 |  | 伯 |  |  |  |  |  |  |  | 人 |  |  |  |  |  |

dat的实现中并不是将所有的Key的字符节点化，只是把公共前缀中的字符节点化，关键字剩余字符串和附加信息串化的作为尾串保存（达到压缩的效果，使用Base负值指定尾串在缓冲区的偏移）；由于Key有加入和移除尾串缓冲区就产生待回收利用的空间（待回收空间用?占位，而不是调整挪动尾串缓冲区），这些空间位置需要地方来保存，就会产生两个tail缓冲区（尾串缓冲区，尾串缓冲区空闲区域偏移位置存储缓冲区）

|  |  |
| --- | --- |
| 1357220624_8644 |  |

# 第二部分 后缀树

## 2.1 后缀树的建立

后缀树，就是包含一则字符串所有后缀的压缩Trie。  
以字符串S=XMADAMYX为例，我们得到下面的结构：

|  |  |
| --- | --- |
| S[1..8], XMADAMYX， 起始位置为1   S[2..8], MADAMYX，起始位置为2      S[3..8], ADAMYX，起始位置为3        S[4..8], DAMYX，起始位置为4           S[5..8], AMYX，起始位置为5             S[6..8], MYX，起始位置为6                S[7..8], YX，起始位置为7                  S[8..8], X，起始位置为8                                  空字串，记为$ | 0_1319297841O3WB |

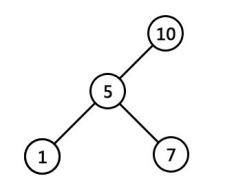
仔细观察上图，我们可以看到不少值得压缩的地方。比如蓝框标注的分支都是独苗，没有必要用单独的节点同边表示。如果我们允许任意一条边里包含多个字母，就可以把这种没有分叉的路径压缩到一条边。另外每条边已经包含了足够的后缀信息，我们就不用再给节点标注字符串信息了。我们只需要在叶节点上标注上每项后缀的起始位置。于是我们得到下左图。

|  |  |
| --- | --- |
| 0_1319300244pcg5 | 0_13193002862qt7 |

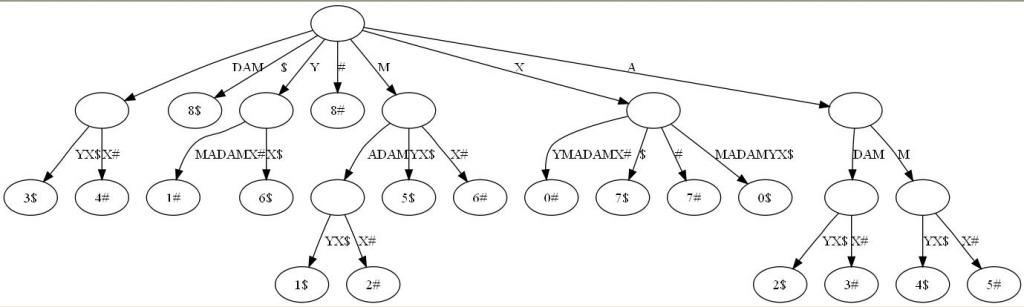
这样的结构丢失了某些后缀。比如后缀X在上图中消失了，因为它正好是字符串XMADAMYX的前缀。为了避免这种情况，我们也规定每项后缀不能是其它后缀的前缀。要解决这个问题其实挺简单，在待处理的子串后加一个空字串就行了。例如我们处理XMADAMYX前，先把XMADAMYX变为 XMADAMYX$，于是就得到suffix tree--后缀树了，如上右图所示。

## 2.2 后缀树与回文问题的关联

**最低共有祖先，LCA（Lowest Common Ancestor)**，也就是任意两节点（多个也行）最长的共有前缀。比如下图中，节点7同节点1的共同祖先是节点5与节点10，但最低共同祖先是5。 查找LCA的算法是O(1)的复杂度，当然，代价是需要对后缀树做复杂度为O(n)的预处理。



**广义后缀树(Generalized Suffix Tree)**。传统的后缀树处理一坨单词的所有后缀。广义后缀树存储任意多个单词的所有后缀。例如下图是单词XMADAMYX与XYMADAMX的广义后缀 树。注意我们需要区分不同单词的后缀，所以叶节点用不同的特殊符号与后缀位置配对。



## 2.3最长回文问题的解决

有了上面的概念，本文引言中提出的查找最长回文问题就相对简单了。咱们来回顾下引言中提出的回文问题的具体描述：找出给定字符串里的最长回文。例如输入XMADAMYX，则输出MADAM。

思维的突破点在于考察回文的半径，而不是回文本身。所谓半径，就是回文对折后的字串。比如回文MADAM 的半径为MAD，半径长度为3，半径的中心是字母D。显然，最长回文必有最长半径，且两条半径相等。还是以MADAM为例，以D为中心往左，我们得到半径 DAM；以D为中心向右，我们得到半径DAM。二者肯定相等。因为MADAM已经是单词XMADAMYX里的最长回文，我们可以肯定从D往左数的字串 DAMX与从D往右数的子串DAMYX共享最长前缀DAM。而这，正是解决回文问题的关键。现在我们有后缀树，怎么把从D向左数的字串DAMX变成后缀 呢？

到这个地步，答案应该明显：把单词XMADAMYX翻转（XMADAMYX=>XYMADAMX，DAMX就变成后缀了）就行了。于是我们把寻找回文的问题转换成了寻找两坨后缀的LCA的问题。当然，我们还需要知道到底查询那些后缀间的LCA。很简单，给定字符串S，如果最长回文的中心在i，那从位置i向右数的后缀刚好是S(i)，而向左数的字符串刚好是翻转S后得到的字符串S‘的后缀S'(n-i+1)。这里的n是字符串S的长度。

## 2.4后缀树的应用

后缀树的用途，总结起来大概有如下几种

查找字符串o是否在字符串S中。   
方案：用S构造后缀树，按在trie中搜索字串的方法搜索o即可。   
原理：若o在S中，则o必然是S的某个后缀的前缀。   
例如S: leconte，查找o: con是否在S中,则o(con)必然是S(leconte)的后缀之一conte的前缀.有了这个前提，采用trie搜索的方法就不难理解了。

指定字符串T在字符串S中的重复次数。   
方案：用S+’$'构造后缀树，搜索T节点下的叶节点数目即为重复次数   
原理：如果T在S中重复了两次，则S应有两个后缀以T为前缀，重复次数就自然统计出来了。

字符串S中的最长重复子串   
方案：原理同2，具体做法就是找到最深的非叶节点。   
这个深是指从root所经历过的字符个数，最深非叶节点所经历的字符串起来就是最长重复子串。   
为什么要非叶节点呢?因为既然是要重复，当然叶节点个数要>=2。

两个字符串S1，S2的最长公共部分   
方案：将S1#S2$作为字符串压入后缀树，找到最深的非叶节点，且该节点的叶节点既有#也有$(无#)。