

# AC自动机

## 算法思想

AC 自动机 = TRIE + KMP

所有的模式串构成一棵 TRIE 并在 Trie 树上所有结点构造失配指针 KMP 思想。

为了进行多模式匹配。

Trie 构建操作和 trie 的 insert 的操作一模一样，其中每个结点代表某个字符串（也有可能是很多个）的某个前缀。

构建 fail 指针：

和 KMP 一样，是失配的时候用于跳转的指针。

KMP 要求最长相同真前后缀，但是 AC 自动机只需要相同后缀。即状态  $u$  的 fail 指针指向另一个状态  $v$ ， $v$  是  $u$  的最长后缀，有可能来自不同的字符串。

现在假设字典树中有一个结点  $u$ ， $u$  的父节点是  $p$ ， $p$  通过字符  $c$  的边指向  $u$ （即  $ch[p][c]=u$ ）现在分情况讨论：

如果  $ch[fail[p]][c]$  存在，那么相当于失配那个状态后面也恰好有字符  $c$ ，所以现在这个状态往下延伸字符  $c$ ，刚好失配那里也可以匹配到，于是  $u$  的 fail 指针指向  $ch[fail[p]][c]$

如果  $ch[fail[p]][c]$  不存在，说明这个失配的状态不满足，需要继续跳 fail，就是  $ch[fail[fail[p]]][c]$ ，一直重复下去，直到 fail 指针跳到根节点，此时没有办法，将 fail 指针指向根节点。

## 算法实现

建树函数  $build()$  目标是构建 fail 指针以及构建自动机。我们采用 BFS 来遍历字典树。

先给根节点自己连 fail 指针，指向自己，接着对根节点连出去的边开始操作。

如果根节点连向某个字符，就把 fail 指针指到根节点，毕竟长度为 1，失配就无了。然后把这个深度为 1 的点入队。

接下来，当队列不为空时我们每次取队首，就是 BFS 啦，并且取出的点在之前已经求过了 fail 指针。之后这个点就相当于上文提到的  $p$ ，如果他连出了某一条边，那么他这个儿子的失配位置，就是他的失配位置向下连这个字符的边（这里按照上面所说应该分类讨论不存在一直跳 fail，但是我们做了一些操作简化），并且将这个儿子入队。如果不存在这个儿子，就连到失配位置的对应字符的边，这样前一种指向的边其实已经是一直跳 fail 的结果了，相当于路径压缩。

匹配函数  $query()$

循环遍历匹配串，用一个变量  $nownode$  记录当前位置，利用 fail 指针找出所有匹配的模式串，累加到答案中，然后清零。再来一个  $tmpnode$  一直跳 fail 指针，而不动  $nownode$

时间复杂度：连了 \$trie\$ 图的复杂度是  $O(\sum |s_{i}|+n|\sum|+|S|)$   $\square$   $n$  是 \$AC\$ 自动机的结点数目，最大可以到  $O(\sum |s_{i}|)$   $\square$

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define maxn 1000100
//一种理解是AC自动机是把kmp放在了Trie树上
//end数组的意思是以这个节点为结尾的单词一共有多少个（任意字母都算在内）
//nxt数组的意思是节点编号的以不同字母结尾的下一个节点编号
struct Trie {
    int nxt[maxn][26], fail[maxn], end[maxn], vis[maxn];
    int root, size;
    int new_node() {
        for(int i=0; i<26; i++) nxt[size][i]=-1;
        end[size++]=0;
        return size-1;
    }
    void init() {
        size=0;
        root=new_node();
        memset(vis, 0, sizeof(vis));
    }
    void insert(char tmp[]) {
        int len=strlen(tmp), now_node=root;
        for(int i=0; i<len; i++) {
            if(nxt[now_node][tmp[i]-'a']==-1) nxt[now_node][tmp[i]-
'a']=new_node();
            now_node=nxt[now_node][tmp[i]-'a'];
        }
        end[now_node]++;
    }
    void build_Trie() {
        queue <int> q_trie;
        fail[root]=root;
        for(int i=0; i<26; i++) {
            if(!(~nxt[root][i])) nxt[root][i]=root;
            else {
                fail[nxt[root][i]]=root;
                q_trie.push(nxt[root][i]);
            }
        }
        while(!q_trie.empty()) {
            int now_node=q_trie.front();
            q_trie.pop();
            for(int i=0; i<26; i++) {
                if(!(~nxt[now_node][i]))
nxt[now_node][i]=nxt[fail[now_node]][i];
                else {
```

```

        fail[nxt[now_node][i]]=nxt[fail[now_node]][i];
        q_trie.push(nxt[now_node][i]);
    }
}
}
}
int query(char tmp[]) {
    int len=strlen(tmp),now_node=root,ans=0;
    for(int i=0; i<len; i++) {
        now_node=nxt[now_node][tmp[i]-'a'];
        int tmp_node=now_node;
        while(tmp_node!=root&&!vis[tmp_node]) {
            ans+=end[tmp_node];
            end[tmp_node]=0;
            vis[tmp_node]=1;
            tmp_node=fail[tmp_node];
        }
    }
    return ans;
}
} AC;
int n;
char str[1000100];
int main() {
    scanf("%d",&n);
    AC.init();
    for(int i=0; i<n; i++) {
        scanf("%s",str);
        AC.insert(str);
    }
    AC.build_Trie();
    scanf("%s",str);
    printf("%d",AC.query(str));
    return 0;
}

```

From:  
<https://wiki.cvbbacm.com/> - CVBB ACM Team

Permanent link:  
[https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal\\_string:%E7%8E%8B%E6%99%BA%E5%BD%AA:ac%E8%87%AA%E5%8A%A8%E6%9C%BA&rev=1627031048](https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal_string:%E7%8E%8B%E6%99%BA%E5%BD%AA:ac%E8%87%AA%E5%8A%A8%E6%9C%BA&rev=1627031048)

Last update: 2021/07/23 17:04