概述

回文自动机描述了单个字符串的所有回文子串。

其分为两个类似 trie 的结构，分别代表字符串的奇回文子串集和偶回文子串集（显然是不重复的集合），这两个结构的每个节点都代表了一个回文子串。

每个节点 u 都有一个 Fail(u)，指向这个节点对应的回文子串的最大回文后缀对应的节点。（后缀不能是自己）

特殊地，偶回文子串集合的根的 Fail 指向奇回文子串集合的根。

构建方法和复杂度

对于一个字符串 s， 从它的最小前缀 s[0] 开始，每构建完 s[i<k] 的回文自动机，就开始构建 s[i<k+1] 的回文自动机。

虽然对于 s[i<k] 到 s[i<k+1] 的构建过程新增了 s[k] 这个字符，但对于字符串整体可能增加的回文串全都是“s[i<k] 的回文后缀 + s[k]”的形式。

不难证明，我们只需要从 s[i<k] 的最大回文后缀开始不断跳 Fail，就能找出可能新增的所有回文串。

剩下的问题是给这些新增的回文串代表的节点 u 找出 Fail(u)。

不难发现，新增的 Fail 只能指向新增的那些节点，也就是说我们在前面的跳 Fail 的过程中也可以顺便把 Fail 解决了。

读者可能会觉得这不是 $O(n^2)$ 的吗？这就涉及到关键了。

本质上，新增 s[k] 这个字符只会增加一个新的回文串。

由于回文串的性质这个其实很显然，就比如 x[aaaaaax] 必然已经在 [xaaaaaa]x 里出现过。

另外，具体实现中，需要顺便维护一下 s[i<k+1] 的最大回文后缀指向的节点是哪个。

例题

leetcode：最长回文子串

```
class Solution {
public:
    struct node{int len = 0, local = 0, son[276], fail = 0;} d[1021];
    int tot;
    // 回文自动机解法,O(n)
    string longestPalindrome(string s) {
        int anslen = -1, ansnode = 0;

        int n = s.size();
        d[0].fail = 1;
        d[1].len = -1; // 初始化，这个 fail 设置是服务于构建过程的，非常重要哦
        tot = 1;
        for(int i=0, las=0; i<n; ++i) // las 初始化为 0 而不是 1，你看懂了吗？
        {
            int p = las;
            while(i - d[p].len - 1 < 0 || s[i - d[p].len - 1] != s[i]) p = d[p].fail;
            // 注意，s[i - d[p].len -1] != s[i] 隐含了 p != 1
            if(!d[p].son[s[i]])
            {
```

```
                int pp = d[p].fail;
                while(i - d[pp].len - 1 < 0 || s[i - d[pp].len -1] != s[i]) pp =
d[pp].fail;

                d[++tot].fail = d[pp].son[s[i]];
                d[tot].len = d[p].len + 2;
                d[tot].local = i;
                d[p].son[s[i]] = tot;

                if(d[tot].len > anslen) anslen = d[tot].len, ansnode = tot;
            }
            las = d[p].son[s[i]];
        }
        return s.substr(d[ansnode].local - anslen + 1, anslen);
    }
};
```