# 题目

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

**示例:**

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // 返回 true

trie.search("app"); // 返回 false

trie.startsWith("app"); // 返回 true

trie.insert("app");

trie.search("app"); // 返回 true

**说明:**

你可以假设所有的输入都是由小写字母 a-z 构成的。

保证所有输入均为非空字符串。

# 分析

**思路：**

Trie树每层节点保存的元素就是26个字母，并且需要定义一个是否为末尾的变量。

**代码：**

class Trie {

private:

bool isEnd;

Trie\* next[26];

public:

Trie() {

isEnd = false;

memset(next, 0, sizeof(next));

}

描述：向 Trie 中插入一个单词 word

实现：这个操作和构建链表很像。首先从根结点的子结点开始与 word 第一个字符进行匹配，一直匹配到前缀链上没有对应的字符，这时开始不断开辟新的结点，直到插入完 word 的最后一个字符，同时还要将最后一个结点isEnd = true;，表示它是一个单词的末尾。

void insert(string word) {

Trie\* node = this;

for (char c : word) {

if (node->next[c-'a'] == NULL) {

node->next[c-'a'] = new Trie();

}

node = node->next[c-'a'];

}

node->isEnd = true;

}

描述：查找 Trie 中是否存在单词 word

实现：从根结点的子结点开始，一直向下匹配即可，如果出现结点值为空就返回false，如果匹配到了最后一个字符，那我们只需判断node->isEnd即可。

bool search(string word) {

Trie\* node = this;

for (char c : word) {

node = node->next[c - 'a'];

if (node == NULL) {

return false;

}

}

return node->isEnd;

}

或：

bool search(string word) {

Trie\* node = this;

for (char c : word) {

if (node->next[c - 'a'] == NULL) {

return false;

}

node = node->next[c - 'a'];

}

return node->isEnd; //不能直接返回true

}

描述：判断 Trie 中是或有以 prefix 为前缀的单词

实现：和 search 操作类似，只是不需要判断最后一个字符结点的isEnd，因为既然能匹配到最后一个字符，那后面一定有单词是以它为前缀的。

bool startsWith(string prefix) {

Trie\* node = this;

for (char c : prefix) {

node = node->next[c-'a'];

if (node == NULL) {

return false;

}

}

return true;

}

};

**总结：**

通过以上介绍和代码实现我们可以总结出 Trie 的几点性质：

Trie 的形状和单词的插入或删除顺序无关，也就是说对于任意给定的一组单词，Trie 的形状都是唯一的。

查找或插入一个长度为 L 的单词，访问 next 数组的次数最多为 L+1，和 Trie 中包含多少个单词无关。

Trie 的每个结点中都保留着一个字母表，这是很耗费空间的。如果 Trie 的高度为 n，字母表的大小为 m，最坏的情况是 Trie 中还不存在前缀相同的单词，那空间复杂度就为 O(m^n)。

最后，关于 Trie 的应用场景，希望你能记住 8 个字：一次建树，多次查询。