0211. 添加与搜索单词 - 数据结构设计

ITCharge 大约 2 分钟

- 标签：深度优先搜索、设计、字典树、字符串
- 难度：中等

题目链接

- 0211. 添加与搜索单词 - 数据结构设计 - 力扣

题目大意

要求：设计一个数据结构，支持「添加新单词」和「查找字符串是否与任何先前添加的字符串匹配」。

实现词典类 WordDictionary：

- `WordDictionary()` 初始化词典对象
- `void addWord(word)` 将 `word` 添加到数据结构中，之后可以对它进行匹配
- `bool search(word)` 如果数据结构中存在字符串与 `word` 匹配，则返回 `True` ；否则，返回 `False` 。 `word` 中可能包含一些 `.` ，每个 `.` 都可以表示任何一个字母。

说明：

- $1 \leq word.length \leq 25$。
- `addWord` 中的 `word` 由小写英文字母组成。
- `search` 中的 `word` 由 `'.'` 或小写英文字母组成。
- 最多调用 10^4 次 `addWord` 和 `search` 。

示例：

- 示例 1：

```py
输入：
["WordDictionary","addWord","addWord","addWord","search","search","search","search"]
[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
输出：
[null,null,null,null,false,true,true,true]
```

```
解释：
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // 返回 False
wordDictionary.search("bad"); // 返回 True
wordDictionary.search(".ad"); // 返回 True
wordDictionary.search("b.."); // 返回 True
```

解题思路

思路 1：字典树

使用前缀树（字典树）。具体做法如下：

- 初始化词典对象时，构造一棵字典树。
- 添加 `word` 时，将 `word` 插入到字典树中。
- 搜索 `word` 时：
  - 如果遇到 `.`，则递归匹配当前 所有子节点，并依次向下查找。匹配到了，则返回 `True`，否则返回 `False`。
  - 如果遇到其他小写字母，则按 `word` 顺序匹配节点。
  - 如果当前节点为 `word` 的结尾，则放回 `True`。

思路 1：代码

```py
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.children = dict()
        self.isEnd = False


    def insert(self, word: str) -> None:
        """
```

```
        Inserts a word into the trie.
        """
        cur = self
        for ch in word:
            if ch not in cur.children:
                cur.children[ch] = Trie()
            cur = cur.children[ch]
        cur.isEnd = True


    def search(self, word: str) -> bool:
        """
        Returns if the word is in the trie.
        """

        def dfs(index, node) -> bool:
            if index == len(word):
                return node.isEnd

            ch = word[index]
            if ch == '.':
                for child in node.children.values():
                    if child is not None and dfs(index + 1, child):
                        return T
            else:
                if ch not in node.children:
                    return False
                child = node.children[ch]
                if child is not None and dfs(index + 1, child):
                    return True
            return False

        return dfs(0, self)


class WordDictionary:

    def __init__(self):
        self.trie_tree = Trie()


    def addWord(self, word: str) -> None:
```

```
        self.trie_tree.insert(word)


    def search(self, word: str) -> bool:
        return self.trie_tree.search(word)
```

思路 1：复杂度分析

- **时间复杂度**：初始化操作为 $O(1)$。添加单词为 $O(|S|)$，搜索单词的平均时间复杂度为 $O(|S|)$，最坏情况下所有字符都是 `'.'` ，所以最坏时间复杂度为 O(|S|^\sum)。其中 $|S|$ 为单词长度，$\sum$ 为字符集的大小，此处为 26。
- **空间复杂度**：$O(|T| * n)$。其中 $|T|$ 为所有添加单词的最大长度，n 为添加字符串个数。