Trie

简介

字典树也叫前缀树、Trie。它本身就是一个树型结构，也就是一颗多叉树，学过树的朋友应该非常容易理解，它的核心操作是插入，查找。删除很少使用，因此这个讲义不包含删除操作。

应用场景及分析（个人理解）：

它的核心思想是用空间换时间，利用字符串的公共前缀来降低查询的时间开销。

- 比如给你一个字符串query，问你这个字符串是否在字符串集合中出现过，这样我们就可以将字符串集合建树，建好之后来匹配query是否出现，那有的朋友肯定会问，之前讲过的hashmap岂不是更好？
- 我们想一下用百度搜索时候，打个“一语”，搜索栏中会给出“一语道破”，“一语成谶(四声的chen)”等推荐文本，这种叫模糊匹配，也就是给出一个模糊的query，希望给出一个相关推荐列表，很明显，hashmap并不能做到模糊匹配，而Trie可以完美实现。

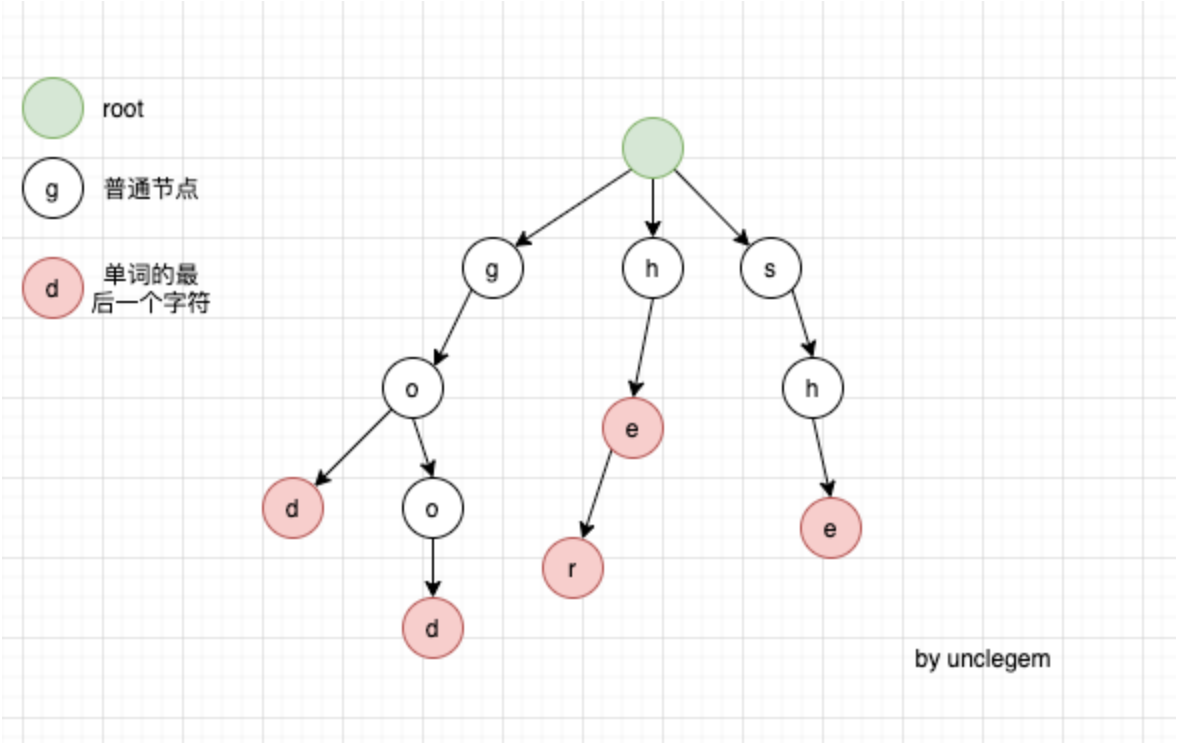
  因此，这里我的理解是：上述精确查找只是模糊查找一个特例，模糊查找hashmap显然做不到，并且如果在精确查找问题中，hashmap出现过多冲突，效率还不一定比Trie高，有兴趣的朋友可以做一下测试，看看哪个快。
- 给你一个长句和一堆敏感词，找出长句中所有敏感词出现的所有位置（想下，有时候我们口吐芬芳，结果发送出去却变成了****，懂了吧）
- 还有些其他场景，这里不过多讨论，有兴趣的可以google一下。

Trie的节点：

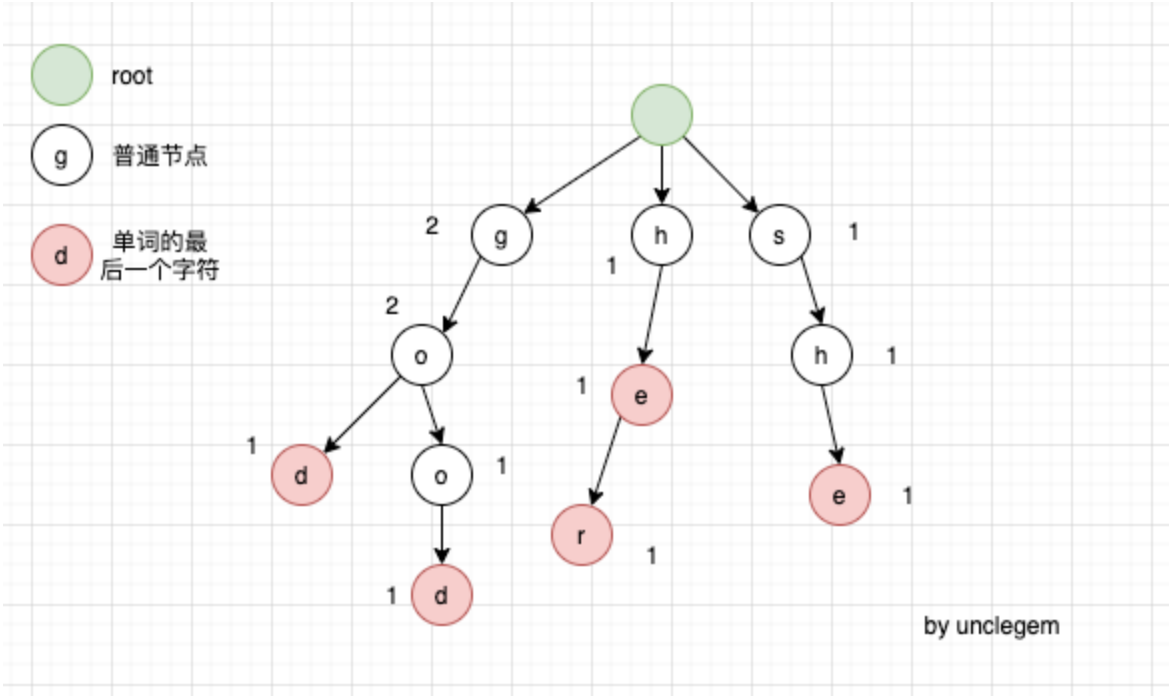
- 根结点无实际意义
- 每一个节点代表一个字符
- 每个节点中的数据结构可以自定义，如isWord(是否是单词)，count(该前缀出现的次数)等，需实际问题实际分析需要什么。

Trie的插入

- 假定给出几个单词如[she,he,her,good,god]构造出一个Trie如下图：



- 也就是说从根结点出发到某一粉色节点所经过的字符组成的单词，在单词列表中出现过，当然我们也可以给树的每个节点加个count属性，代表根结点到该节点所构成的字符串前缀出现的次数



可以看出树的构造非常简单，插入新单词的时候就从根结点出发一个字符一个字符插入，有对应的字符节点就更新对应的属性，没有就创建一个！

Trie的查询

查询更简单了，给定一个Trie和一个单词，和插入的过程类似，一个字符一个字符找

- 若中途有个字符没有对应节点→Trie不含该单词
- 若字符串遍历完了，都有对应节点，但最后一个字符对应的节点并不是粉色的，也就不是一个单词→Trie不含该单词

Trie的复杂度

插入和查询的时间复杂度自然是$O(len(key))$，key是待插入(查找)的字串。

建树的最坏空间复杂度是$O(m^{n})$, m是字符集中字符个数，n是字符串长度。