**一. 基于单模式串和 Trie 树实现的敏感词过滤**

单模式串匹配算法，是在一个模式串和一个主串之间进行匹配，也就是说，在一个主串中查找一个模式串。

多模式串匹配算法，就是在多个模式串和一个主串之间做匹配，也就是说，在一个主串中查找多个模式串。

我们可以针对每个敏感词，通过单模式串匹配算法（比如 KMP 算法）与用户输入的文字内容进行匹配。但是，这样做的话，每个匹配过程都需要扫描一遍用户输入的内容。整个过程下来就要扫描很多遍用户输入的内容。如果敏感词很多，比如几千个，并且用户输入的内容很长，假如有上千个字符，那我们就需要扫描几千遍这样的输入内容。很显然，这种处理思路比较低效。

Trie 树就是一种多模式串匹配算法。我们可以对敏感词字典进行预处理，构建成 Trie 树结构。这个预处理的操作只需要做一次，如果敏感词字典动态更新了，比如删除、添加了一个敏感词，那我们只需要动态更新一下 Trie 树就可以了。

当用户输入一个文本内容后，我们把用户输入的内容作为主串，从第一个字符（假设是字符 C）开始，在 Trie 树中匹配。当匹配到 Trie 树的叶子节点，或者中途遇到不匹配字符的时候，我们将主串的开始匹配位置后移一位，也就是从字符 C 的下一个字符开始，重新在 Trie 树中匹配。

基于 Trie 树的这种处理方法，有点类似单模式串匹配的 BF 算法。我们知道，单模式串匹配算法中，KMP 算法对 BF 算法进行改进，引入了 next 数组，让匹配失败时，尽可能将模式串往后多滑动几位。借鉴单模式串的优化改进方法，能否对多模式串 Trie 树进行改进，进一步提高 Trie 树的效率呢？这就要用到 AC 自动机算法了。

**二. 经典的多模式串匹配算法: AC自动机**

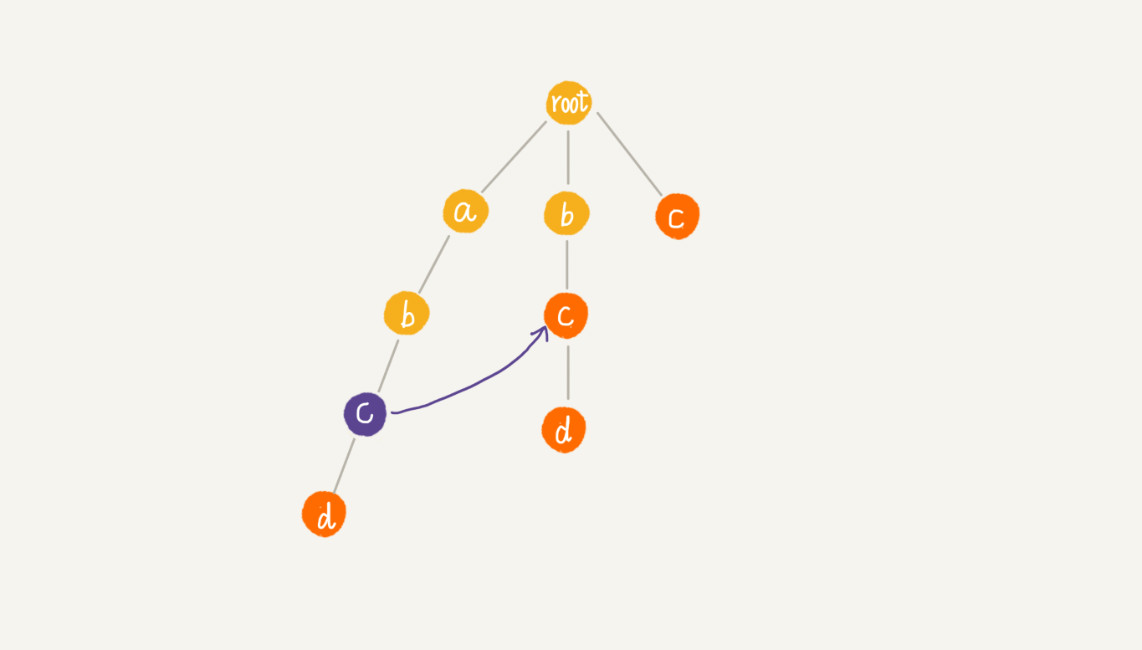
AC 自动机算法，全称是 Aho-Corasick 算法。其实，Trie 树跟 AC 自动机之间的关系，就像单串匹配中朴素的串匹配算法，跟 KMP 算法之间的关系一样，只不过前者针对的是多模式串而已。所以，AC 自动机实际上就是在 Trie 树之上，加了类似 KMP 的 next 数组，只不过此处的 next 数组是构建在树上罢了。

AC 自动机的构建，包含两个操作：

* 将多个模式串构建成 Trie 树；
* 在 Trie 树上构建失败指针（相当于 KMP 中的失效函数 next 数组）。

**构建好 Trie 树之后，如何在它之上构建失败指针？**这里有 4 个模式串，分别是 c，bc，bcd，abcd；主串是 abcd。

假设我们沿 Trie 树走到 p 节点，也就是下图中的紫色节点，那 p 的失败指针就是从 root 走到紫色节点形成的字符串 abc，跟所有模式串前缀（或者整个模式串）匹配的最长可匹配后缀子串，就是箭头指的 bc 模式串。

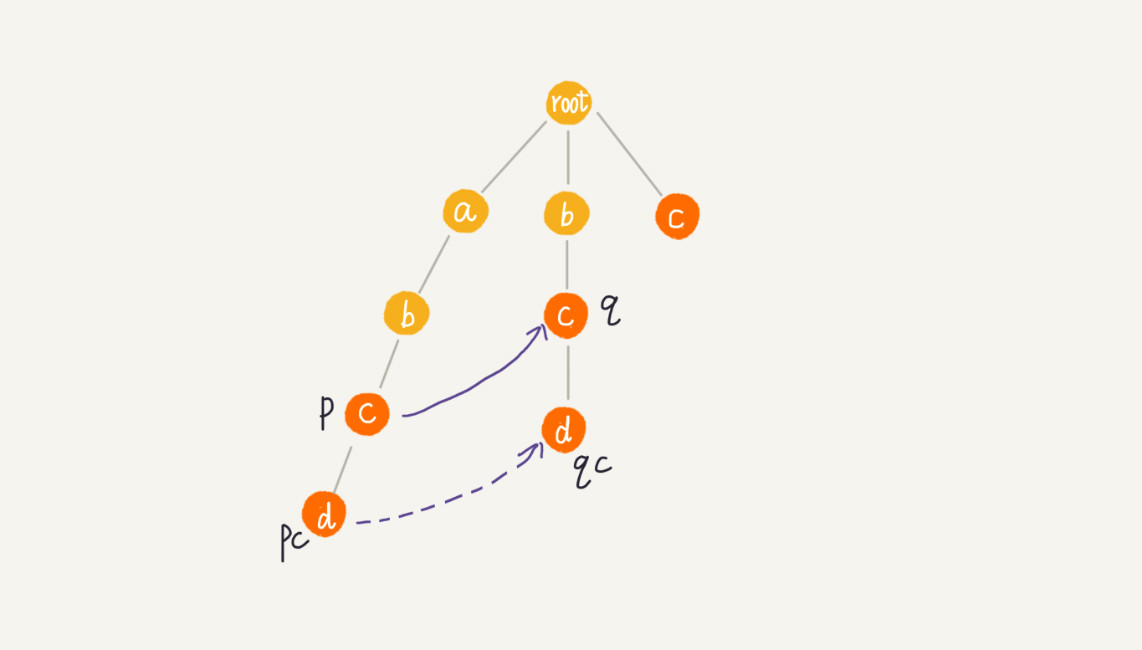


这里的最长可匹配后缀子串，我稍微解释一下。字符串 abc 的后缀子串有两个 bc，c，我们拿它们与其他模式串匹配，如果某个后缀子串可以匹配某个模式串的前缀（或者整个模式串），那我们就把这个后缀子串叫作可匹配后缀子串。

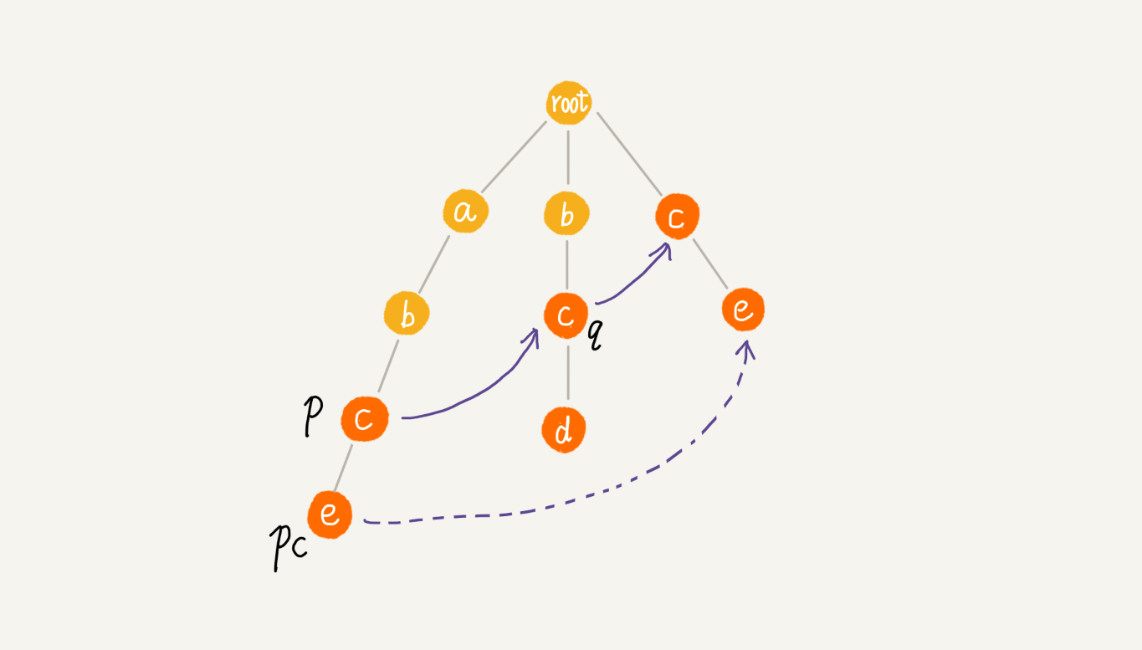
我们从可匹配后缀子串中，找出最长的一个，就是刚刚讲到的最长可匹配后缀子串。我们将 p 节点的失败指针指向那个最长匹配后缀子串对应的模式串的前缀（或模式串）的最后一个节点，就是下图中箭头指向的节点。

计算每个节点的失败指针这个过程看起来有些复杂。其实，如果我们把树中相同深度的节点放到同一层，那么某个节点的失败指针只有可能出现在它所在层的上一层。

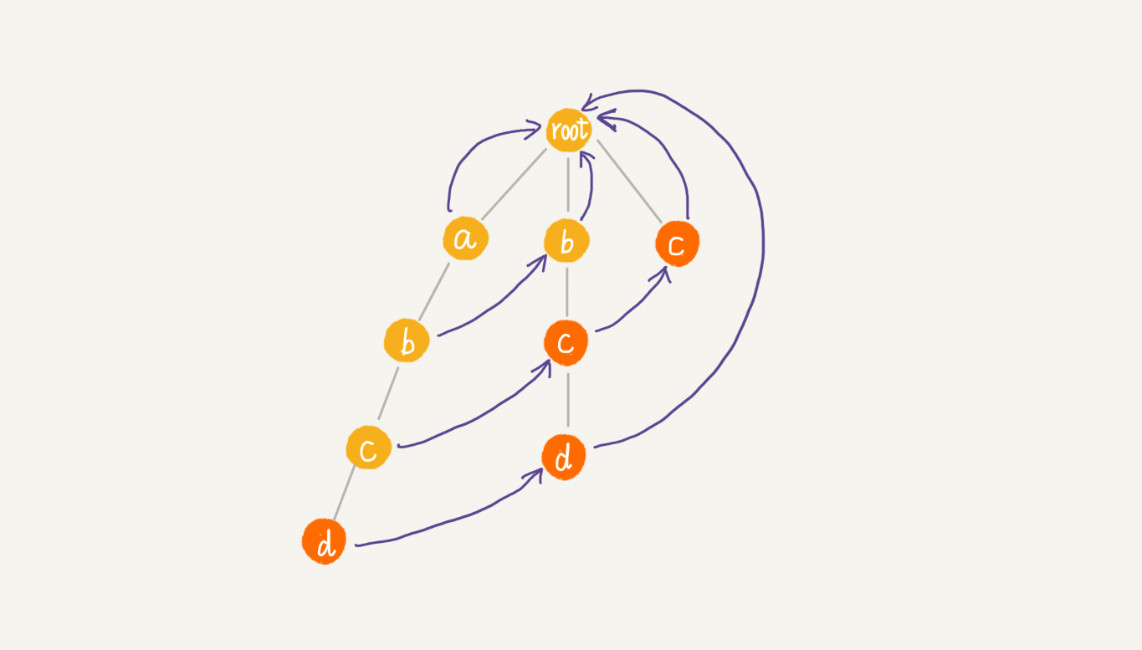
首先 root 的失败指针为 NULL，也就是指向自己。假设节点 p 的失败指针指向节点 q，我们看节点 p 的子节点 pc 对应的字符，是否也可以在节点 q 的子节点中找到。如果找到了节点 q 的一个子节点 qc，对应的字符跟节点 pc 对应的字符相同，则将节点 pc 的失败指针指向节点 qc。



如果节点 q 中没有子节点的字符等于节点 pc 包含的字符，则令 q=q->fail继续上面的查找，直到 q 是 root 为止，如果还没有找到相同字符的子节点，就让节点 pc 的失败指针指向 root。



通过按层来计算每个节点的子节点的失效指针，刚刚举的那个例子，最后构建完成之后的 AC 自动机就是下面这个样子：



失败指针的作用是：

1. 假设主串中连续三个字母fuc已经在树中并且p指针已经指向了c，但是假设主串中下一个字母是h，而h不在节点c的子节点中。此时我们只能去找一个没有fuc这么长的但也从root开始的模式串或者模式串前缀，并且它的他的节点c的子结点中有h。比如我们q = p.fail得到了一个串的结尾。q的子节点有h，从root到q的串可能是c,uc,或者q = root。所以我们的匹配进度就变成了找到了连续的ch或uch或者只找到了h

2. 假设主串中连续三个字母fuc已经在树中并且p指针已经指向了c，并且主串的下一个字母k在节点c的子节点中。这时候我们就要看fuck是不是已经是一个完整的模式串，也就是k是不是一个模式串的结尾字符。是的话则保存一下fuck的坐标用来以后替换成\*

代码中无论k是不是结尾，我们都要看看fuck的后缀字串,uck,ck,k是不是一个模式串。就又用到了q=q.fail. 如果这个从root开始到q结束的后缀字串，结尾字符也是k。那么我们就把这个子串的坐标的也保存下以后替换成\*。

这里比如我有两个敏感词uck和fuck，就会重复保存了一些位置，但大多数情况下没有uck这种重复的敏感词。经过fuck判定完k是结尾之后的q=q.fail 一般会直接指向root。

**三. 时间复杂度**

**构建AC自动机**

首先，我们需要将敏感词构建成 AC 自动机，包括构建 Trie 树以及构建失败指针

Trie 树构建的时间复杂度是 O(m\*len)，其中 len 表示敏感词的平均长度，m 表示敏感词的个数。

假设 Trie 树中总的节点个数是 k，每个节点构建失败指针的时候，（你可以看下代码）最耗时的环节是 while 循环中的 q=q->fail，每运行一次这个语句，q 指向节点的深度都会减少 1，而树的高度最高也不会超过 len，所以每个节点构建失败指针的时间复杂度是 O(len)。整个失败指针的构建过程就是 O(k\*len)。

不过，AC 自动机的构建过程都是预先处理好的，构建好之后，并不会频繁地更新，所以不会影响到敏感词过滤的运行效率。

**用 AC 自动机做匹配**

跟刚刚构建失败指针的分析类似，for 循环依次遍历主串中的每个字符，for 循环内部最耗时的部分也是 while 循环，而这一部分的时间复杂度也是 O(len)，所以总的匹配的时间复杂度就是 O(n\*len)。因为敏感词并不会很长，而且这个时间复杂度只是一个非常宽泛的上限，实际情况下，可能近似于 O(n)，所以 AC 自动机做敏感词过滤，性能非常高。

你可以会说，从时间复杂度上看，AC 自动机匹配的效率跟 Trie 树一样啊。实际上，因为失效指针可能大部分情况下都指向 root 节点，所以绝大部分情况下，在 AC 自动机上做匹配的效率要远高于刚刚计算出的比较宽泛的时间复杂度。只有在极端情况下，如图所示，AC 自动机的性能才会退化的跟 Trie 树一样。

