🚫敏感词“智能”检测🚫

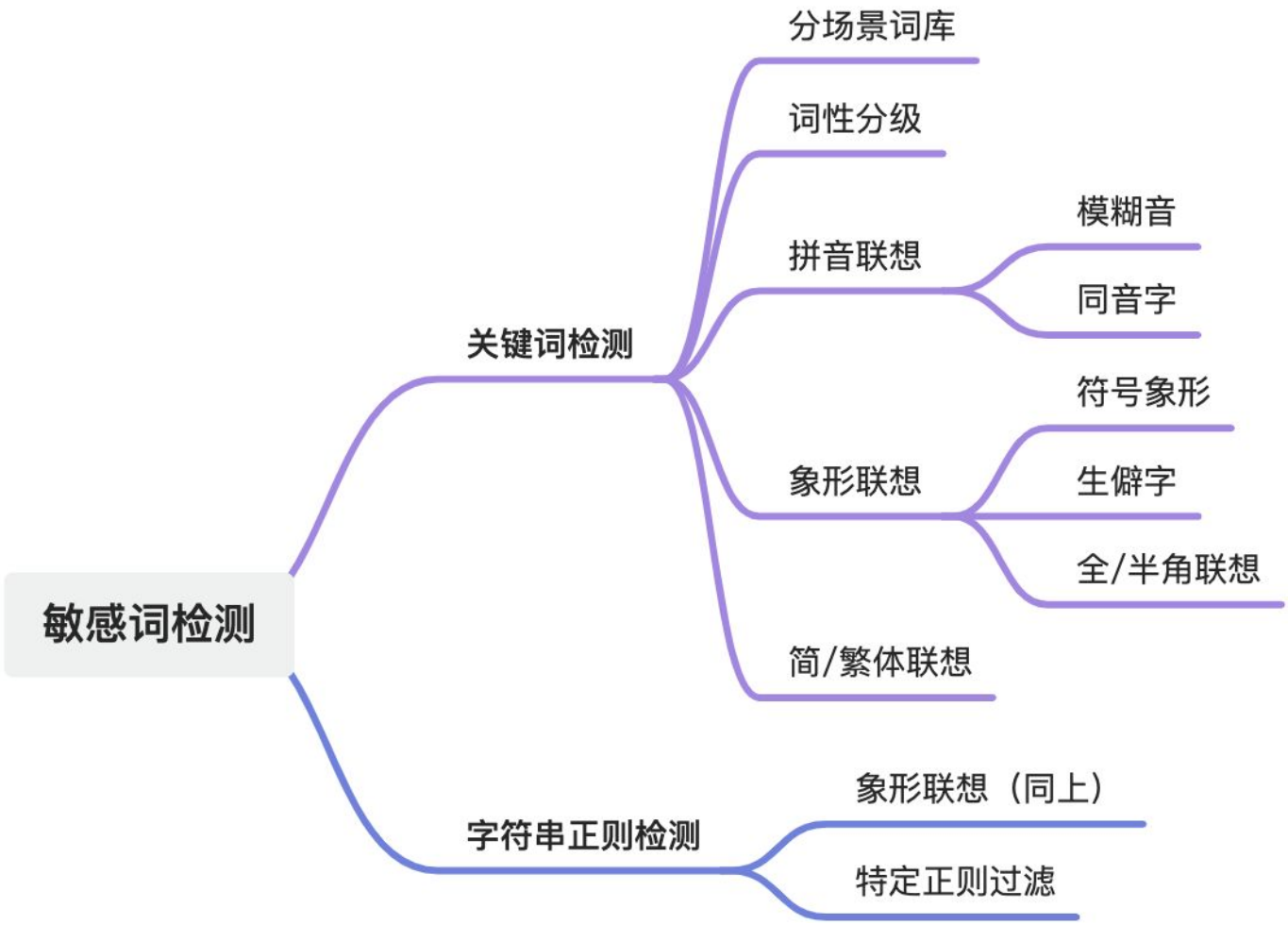
1. 需求背景

针对游戏中包括用户昵称、聊天内容等用户自定义文本内容提供一个高速且具有一定“联想”能力的敏感词检测系统。

> **注意：**这里所说的联想是表示对字形、拼音、简繁体的转换联想而非关联性联想。
>
> **案例：**目前词库中有 `畅唐` 这个词，当玩家输入 `Cang瑭` 时需要标注为敏感词。

2. 技术设计方案

2.1. 核心设计



2.2. 算法应用

显然使用 `for` 循环遍历词库再用 `contains` 判断包含关键词是一种dinner做法。亟需一种高效的字符串匹配算法来保证对用户输入字符串的处理速度。

2.2.1. DFA

> **确定有限状态自动机**或**确定有限自动机**（英语：deterministic finite automaton, DFA）

一个DFA包含以下部分

1. 非空有限的状态集合
2. 非空有限的输入字符集合
3. 转移函数
4. 开始状态（属于状态集合）
5. 接受状态集合（状态集合的真子集）

DFA中任何一个状态对应任何允许输入字符都能在转移函数中确定转移状态。即

$$f(p,x)=q,\quad p,q\in \text{状态集合}Q, x\in \text{输入字符串集合}U$$

当一个字符串按字符输入DFA能从开始状态最终转移到接受状态，就表示此DFA接受此字符串。

2.2.2. 基于DFA的变形

有了上述的理论支持，很容易将敏感词检查的背景套在DFA上，一个敏感词检测器包含以下部分

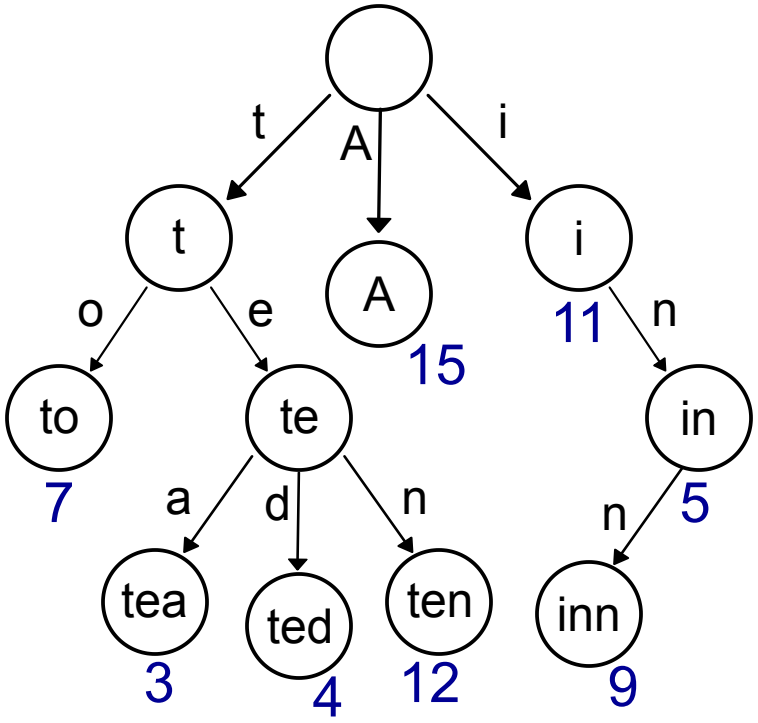
1. 状态集合：所有敏感词的字构成的集合
2. 输入字符集合：有效的用户输入字符集合（中英文、简繁体、全半角、字母数字、部分特殊符号）
3. 转移函数：一个敏感词就是一个转移公式
4. 开始状态：空白
5. 接收状态集合：所有敏感词的组成的集合

虽然说状态集合是所有敏感词的字构成的集合，但是 `中华人民共和国` 和 `中国` 里面的 `国` 并非同一个状态，而 `中` 却是同一个状态。

这就要提到一个和DFA及其相似的**Trie**

2.2.3. Trie

> trie，又称**前缀树**或**字典树**



前缀树在长用于搜索提示，字典树顾名思义思想参考了我们所使用的字典。从根节点到任意节点的路径共同组成了节点的前缀串，**故一个节点的值由它的前缀和自身的值共同决定**，Trie本质上也是一个DFA，特殊之处就是Trie根节点(开始状态)到达任何节点(状态)的路径都是唯一确定的。

2.2.4. 基于Trie的变形

只需要将敏感词库构建成一颗前缀树，然后通过用户的输入来搜索“这本字典”即可。

这种做法可以提高我们的检索效率，但是还是存在一个致命的缺陷：在词库大小相同的前提下，检测速率取决于关键词在输入字符串中的位置以及输入字符串的长度。

对于输入的字符串，需要逐字作为起始态进行检查。例如 `中华人民群众` ， `中` 字起始检查失败后，要回退到 `华` 字进行第二轮检查。

若 `中华人民共和国` 和 `人民群众` 是关键词，有没有办法让“ `中` 字轮”在 `共` 字处失败后继续 `人` 字检出 `人民群众` 关键词呢?

既然这么说了那肯定是有———DFA进阶——Fail指针。

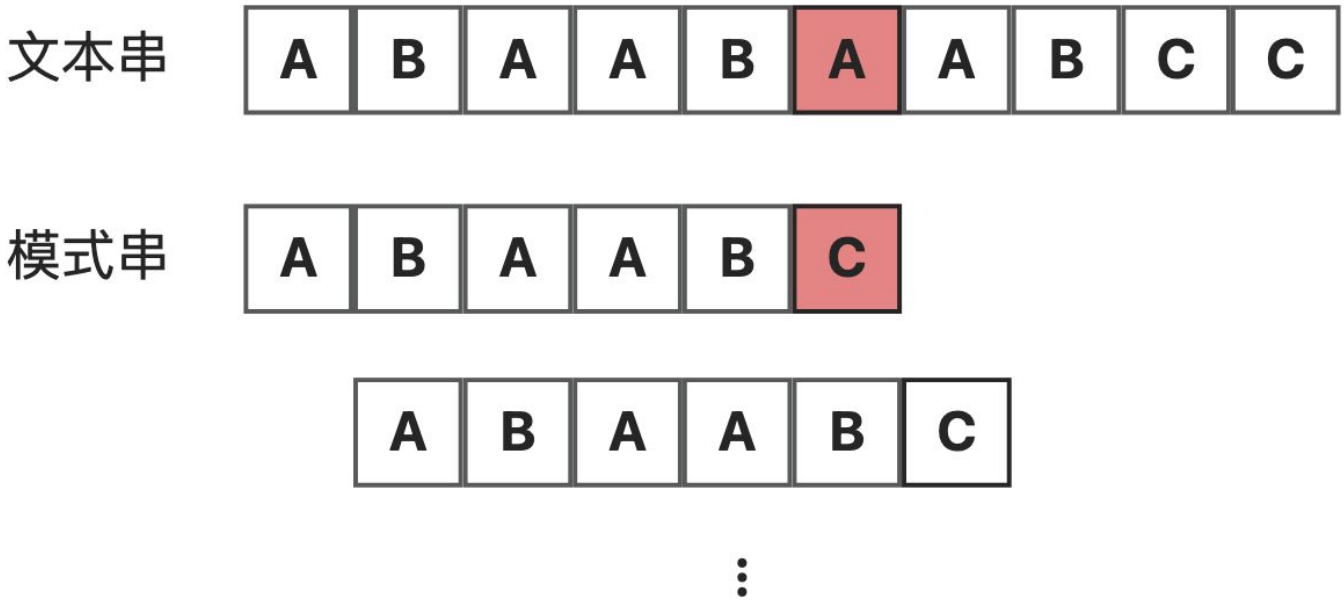
2.2.5. Fail指针 & KMP

在了解Fail指针前，先来看一个著名的字符串匹配算法——KMP算法。

2.2.5.1. KMP

KMP算法应该算是大学数据结构课程中的第一个真正意义上的算法，用于字符串匹配——检查模式串A是否为文本串B的子串。

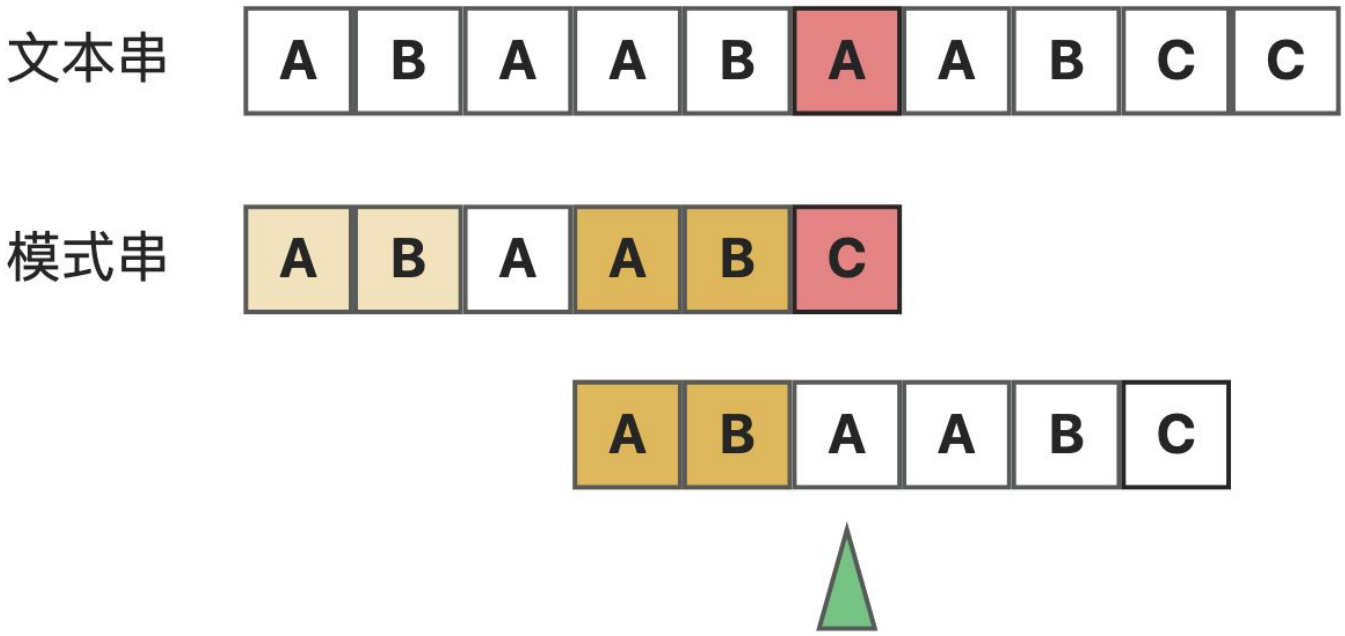
普通方法下，当文本串和模式串都比较长时，这种暴力解法就显得效率低下



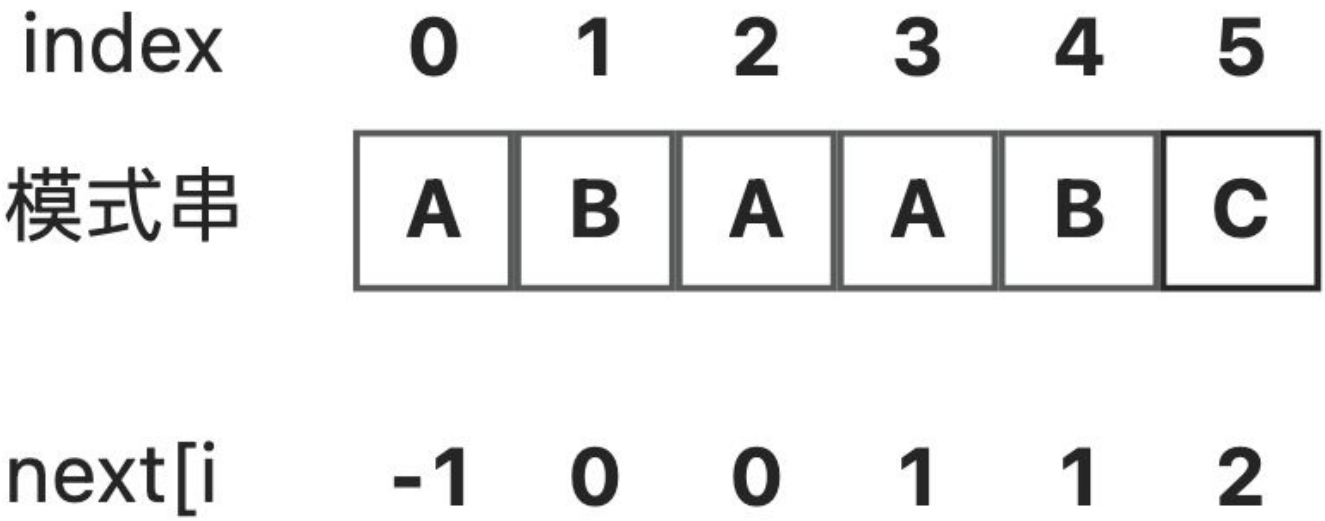
于是就有了KMP算法所提出的核心部分：**next数组**

next数组基于模式串生成，相当于为模式串提供了一份行动手册，让模式串知道在一轮失败匹配后，下一轮从何处开始，从而避开所有无效的移动操作。

有了next数组后，匹配的过程变成了这样



直接省去了两次无效的移动。此案例中的模式串的next数组为



这个next数组的计算过程，引入了一个**最长公共前后缀**的概念：即一个字符串中，即作为后缀又作为前缀的子串最大长度。通俗解释为“影子状态”。

字符串	最长公共前后缀	长度
A	无	0
AB	无	0
ABA	A	1
ABAA	A	1

ABAAB	AB	2

计算方式一目了然，用公式表述即为

$$next[i] = \begin{cases} -1 & i = 0 \\ len(\text{最长公共前后缀}(s, 0, i-1)) & i \in [1, len(s)) \end{cases}$$

理解所谓“影子状态”对于使用next值有很大的帮助

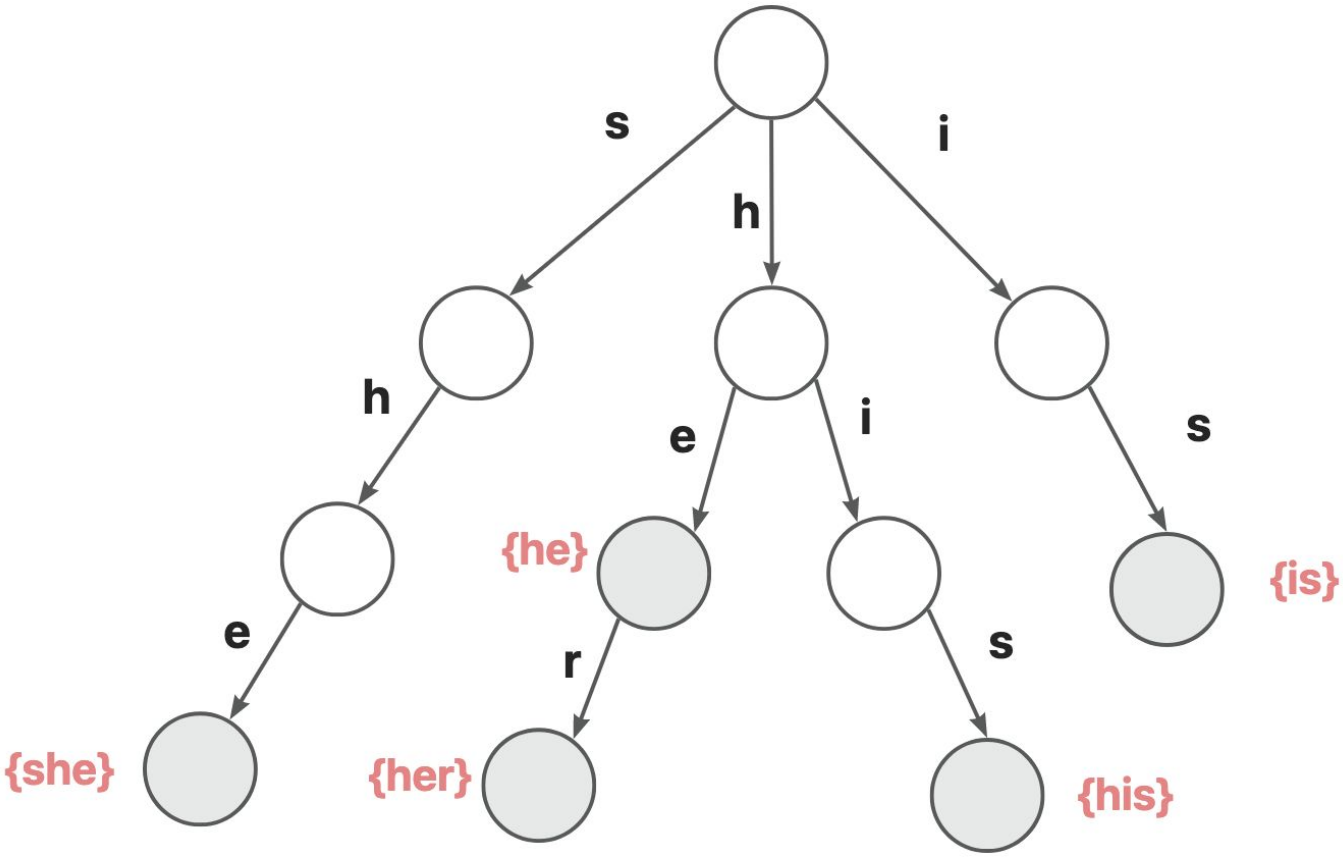
拿上面的例子来说，字符 `C` 处匹配失败时，我们匹配成功的串 `ABAAB` 中的最长公共前后缀是 `AB` 等价于 `C` 的影子状态是 `2` 对应原始字符串中 `index = 2` 的字符 `A` ，此时直接回溯到 `A` 位置继续比较即可，因为 `AB` 公共前后缀已经比对过了可以省略比较操作（即影子）。

2.2.5.2. Fail指针

Fail指针与KMP算法中的next指针有异曲同工之妙，Fail指针建立在Tire树上，可以帮助在进行关键词匹配失配(Fail)时，快速转移到下一个树枝上，减少无效的回溯操作，并且最明显的优势就是对文本串进行一次遍历就能找出所有关键词。

> 模拟场景
>
> 词库：`{she, her, he, his, is}`
>
> 待检查词：`ishishe`

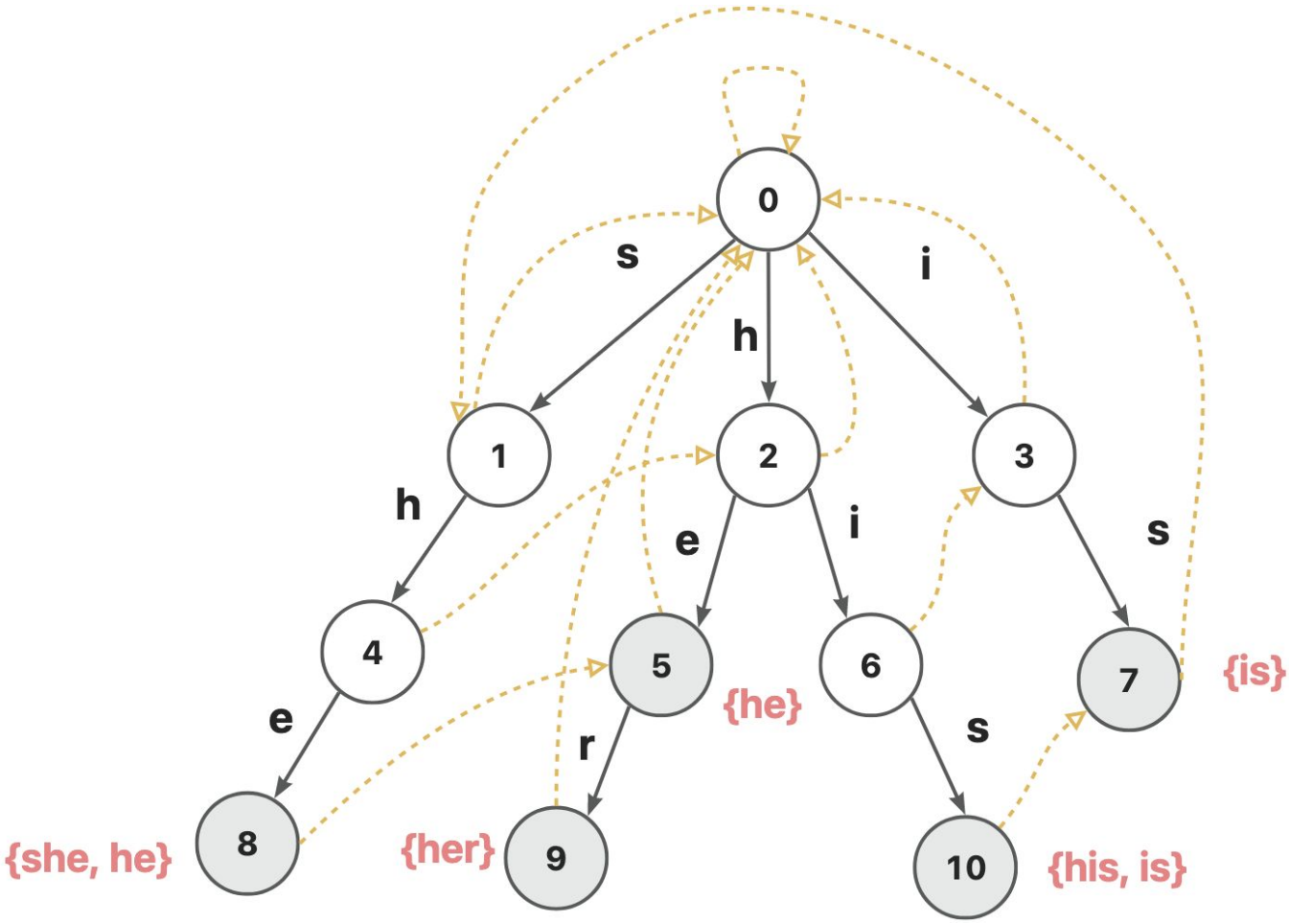
首先使用模拟词库生成一个Trie



匹配过程

1. start = `0, i` , match `is` break
2. start = `1, s` , mismatch break
3. start = `2, h` , match `his` break
4. start = `3, i` , match `is` break
5. start = `4, s` , match `she` break
6. start = `5, h` , match `he` break
7. start = `6, e` , mismatch break

加上了Fail指针后 `ishishe`



匹配过程

1. `N0` –> `i` –> `N3` –> `s` –> `N7` (match `is`)
2. `N7` –> `h` –fail–> `N1` (fail)
3. `N1` –> `h` –> `N4`
4. `N4` –> `i` –fail–> `N2` (fail)
5. `N2` –> `i` –> `N6` –> `s` –> `N10` (match `his`, `is`)
6. `N10` –> `h` –fail–> `N7` (fail)
7. `N7` –> `h` –fail–> `N1` (fail)
8. `N1` –> `h` –> `N4` –> `e` –> `N8` (match `she`, `he`)

两种匹配过程分别使用比较操作21次（普通）、11次（Fail指针），这个结果差距在输入串越长时越明显。

Fail指针的构建与next数组类似，就是寻找当前节点的“影子状态”，例如 `she` 词中 `e` 节点的影子状态就是 `he` 和 `her` 词中的 `e` 【she中后缀he同时作为her、he的前缀】，当输入词 `sher` 在 `N8` 处匹配关键词 `she` 后对继续输入的 `r` 失配时，通过fail指针跳到 `N5` 继续匹配 `r` 即可省略对影子 `he` 的检查，直接匹配关键词 `her` 。

根节点的Fail指针指向自己，其他节点的Fail指针指向父节点的Fail指针所指向节点的下值为当前节点值的节点。若父节点Fail指向节点下没有找到值相同的结点，就在父节点Fail指向节点的Fail指向节点上继续寻找，直到找到或者节点为null。

Fail指针构建伪代码 Python 复制代码

```python
# n为当前节点，  v为导向当前节点的路径值
def fail(n, v):
    if n == null:
        return root
    else:
        parent_fail = n.parent.p_fail
        p_fail = parent_fail.get_children(v)
        if p_fail:
            return p_fail
        else:
            return fail(parent_fail, v)
```

2.2.5.3. Fail指针的优劣势

优势

1. 全文本检查的情况下循环次数更少，速度更快

劣势

1. 构建Fail指针耗时，词库冗余的空间占用

2.3. 词形联想 & 词性分级

2.3.1. 词性联想

对于用户的输入字词，在“字形”的非语义表达上系统可以进行统一处理，例如

- 简繁体统一为简体
- 大小写统一为小写
- 全半角统一为半角
- 字似但不同音的生僻字统一为常见字

但是对于其所想的表述的内容（即语义）系统不可以做过度解读，可能存在歧义，例如

- 一个字符串中的字母 `o` ，可能会被误解为象形的数字 `0` ，反之亦然
- 一个单词 `如花似玉` 中的 `如花` 通过简单的拼音联想后 `ruhua` 就会与敏感词 `辱华` 匹配，这显然的不对的

但是有一类高危词是绝对不允许出现的，即便是谐音模糊音。也就是说程序需要根据场景以及词条来设置变形的限度，于是就有了词性分级的背景。

2.3.2. 词性分级

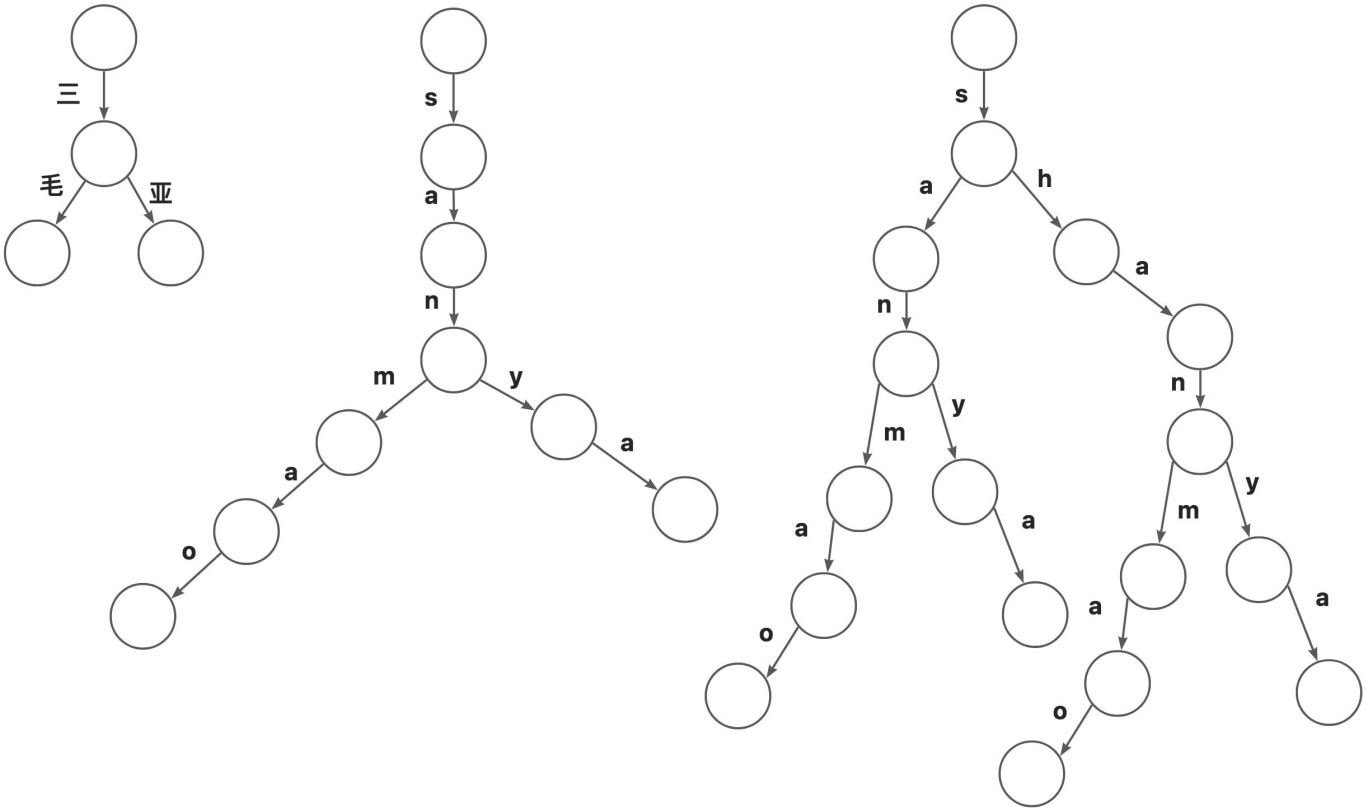
目前暂定将敏感词按危险程度划分了低中高三个等级

- 低：必须文字匹配
- 中：满足低级条件 or 拼音相同（不比较声调）
- 高：满足中级条件 or 读音相似（谐音，模糊音【卷翘舌，前后鼻音】）

进行分级后，就可以在词库上大做文章并且控制限度

低级词完全不做变换，中级词全部统一化为拼音，高级词同样也转换成拼音同时转换出相似读音，然后基于转换结果分别构建三颗ACTrie(Trie + Fail)。在进行检测时三颗ACTrie从低到高进行检测。

> 模拟场景
>
> 敏感词： `三毛` `三亚`
>
> 构建低中高ACTrie（省略fail指针）



不难看出即使词量相同，根据树的特性，等级越高的ACTrie联想结果所占的空间更多，构建耗时也更长。若没有词性分级限制联想，在词库较大的背景下前期构建所消耗资源是极其庞大的。

在进行词性分级后，系统对词库有了以下要求

1. 高级词性的词量必须严格控制
2. 对于弱敏感词（例如外国人名这类一旦变换形态就难以联想到的词），尽量使用低级词性
3. 对于正常敏感词（例如国家领导人名这类通过拼音或者生僻字可以联想到的词），可以选用中级词性
4. 对于高危词（例如先烈人名这类无论什么形态都不能出现的词），谨慎使用高级词性

3. 系统缺陷

3.1. 启动 & 构建（有趣的“坑”）

使用模拟词库：简中词库，词量17287词，平均词长5，均使用全量词库，构建耗时约65秒，内存占用约500MB

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
   193     34.4 MiB     34.4 MiB           1       @profile
   194                                             def __init__(self, keywords: dict[str, dict[WordLevel, list]]):
   195                                                 """
   196                                                 :param keywords dict 全量词库(分库分级) 结构为: {库1 -> { 等级1 -> [keywords, ...], 等级2 -> [keywords, ...] }, 库2 -> ...}
   197                                                 """
   198     34.4 MiB      0.0 MiB           1           low_level_words = {}
   199     34.4 MiB      0.0 MiB           1           medium_level_words = {}
   200     34.4 MiB      0.0 MiB           1           high_level_words = {}
   201     34.4 MiB      0.0 MiB           2           for lib_name, levels_word in keywords.items():
   202     34.4 MiB      0.0 MiB           1               low_level_words[lib_name] = levels_word.get(WordLevel.LOW, [])
   203     34.4 MiB      0.0 MiB           1               medium_level_words[lib_name] = levels_word.get(WordLevel.MEDIUM, [])
   204     34.4 MiB      0.0 MiB           1               high_level_words[lib_name] = levels_word.get(WordLevel.HIGH, [])
   205
   206     71.2 MiB     36.7 MiB           1           self._trie_low = Trie.build_low_level(low_level_words)
   207    177.2 MiB    106.0 MiB           1           self._trie_medium = Trie.build_medium_level(medium_level_words)
   208    523.7 MiB    346.4 MiB           1           self._trie_high = Trie.build_high_level(high_level_words)
```

总体

```
1678330583.37963
1678330583.73701
Filename: /Users/5akura/Documents/Code/ct108/PythonProject/dc.other/ml.sensitivewords.api/logic/algorithm/keyword_actrie_checker.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    85     34.0 MiB     34.0 MiB           1       @profile
    86                                             def __init__(self, keyword_libraries: dict, level: WordLevel) -> None:
    87     34.0 MiB      0.0 MiB           1           self._root = TrieNode()
    88     34.0 MiB      0.0 MiB           1           self._level = level
    89     70.7 MiB      0.0 MiB           2           for library, keywords in keyword_libraries.items():
    90     70.7 MiB      1.4 MiB       17288               for keyword in keywords:
    91     70.7 MiB     35.2 MiB       17287                   self._append_keyword(library, keyword)
    92     70.7 MiB      0.0 MiB           1           print(time.time())
    93     70.8 MiB      0.1 MiB           1           self._build_fail_point()
    94     70.8 MiB      0.0 MiB           1           print(time.time())


1678330700.613921
1678330701.4578292
Filename: /Users/5akura/Documents/Code/ct108/PythonProject/dc.other/ml.sensitivewords.api/logic/algorithm/keyword_actrie_checker.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    85     70.8 MiB     70.8 MiB           1       @profile
    86                                             def __init__(self, keyword_libraries: dict, level: WordLevel) -> None:
    87     70.8 MiB      0.0 MiB           1           self._root = TrieNode()
    88     70.8 MiB      0.0 MiB           1           self._level = level
    89    178.5 MiB      0.0 MiB           2           for library, keywords in keyword_libraries.items():
    90    178.5 MiB   -428.1 MiB       17288               for keyword in keywords:
    91    178.5 MiB   -320.4 MiB       17287                   self._append_keyword(library, keyword)
    92    178.5 MiB      0.0 MiB           1           print(time.time())
    93    178.6 MiB      0.1 MiB           1           self._build_fail_point()
    94    178.6 MiB      0.0 MiB           1           print(time.time())


1678330823.630325
1678330827.097266
Filename: /Users/5akura/Documents/Code/ct108/PythonProject/dc.other/ml.sensitivewords.api/logic/algorithm/keyword_actrie_checker.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    85    178.6 MiB    178.6 MiB           1       @profile
    86                                             def __init__(self, keyword_libraries: dict, level: WordLevel) -> None:
    87    178.6 MiB      0.0 MiB           1           self._root = TrieNode()
    88    178.6 MiB      0.0 MiB           1           self._level = level
    89    527.1 MiB      0.0 MiB           2           for library, keywords in keyword_libraries.items():
    90    527.1 MiB   -307.5 MiB       17288               for keyword in keywords:
    91    527.1 MiB     40.9 MiB       17287                   self._append_keyword(library, keyword)
    92    527.1 MiB      0.0 MiB           1           print(time.time())
    93    527.4 MiB      0.3 MiB           1           self._build_fail_point()
    94    527.4 MiB      0.0 MiB           1           print(time.time())
```

分级

由此可以看出，Fail指针的构建对耗时以及内存占用都是较小的。

经过确认，主要构建时间都消耗在了对词进行拼音转换上。代码入下

汉字转拼音 Python 复制代码

```python
def han2pinyin(uchar: str, fuzzy: bool = False) -> list[str]:
    """
    汉字转换拼音
    :param uchar str 字符
    :param fuzzy bool [False] 是否使用模糊音
    :return list 所有的拼音，当使用了模糊音时列表有多个元素
    """
    uchar = uchar[0]

    def _get_pinyin() -> tuple[str, str]:
        pinyin: str = _Pinyin().get_pinyin(uchar, "")
        if pinyin != uchar:
            _initial: str = _Pinyin().get_initial(uchar, with_retroflex=True).lower()
            if _initial not in _HAN_INITIAL_LIST:
                return "", pinyin
            return _initial, pinyin.split(_initial, 1)[1]
        elif CharacterUtil.is_chinese(uchar):
            return "", ""
        else:
            return uchar, ""

    initial, vowel = _get_pinyin()

    if not initial and not vowel:
        return [""]
    if fuzzy:
        initials = {initial, _FUZZY_PINYIN_INITIAL.get(initial, initial)}
        vowels = {vowel, _FUZZY_PINYIN_VOWEL.get(vowel, vowel)}
        return [i + v for i, v in product(initials, vowels)]
    else:
        return [initial + vowel]
```

此方法的调用耗时极其严重！平均调用耗时10ms， `_Pinyin().get_pinyin()` 和 `_Pinyin().get_initial()` 分别耗时5ms，做个数学题

(17000 * 5 * 10) / 1000 = 850

理论耗时应该是850秒，实际上程序对汉字的拼音转换做了内存缓存，使得同一个字在同一个等级下字形变换只会做一次，程序统计这个方法实际调用次数为：6514，这样就非常接近实际耗时了。

也就是说如果可以优化这个方法的调用耗时，就可以大幅提高构建速度！

- 无敌巨坑

  重点来了朋友们，`_Pinyin().` 这个写法是不是有点奇怪！？！

  没错，它就是一个构造方法，但是构造方法里面的操作更是“违规”——读文件

  Pinyin 构造方法 Python 复制代码

```python
    def __init__(self, data_path: str = str(data_path)) -> None:
        lines = Path(data_path).read_text().splitlines()
        self.pinyins = dict(tuple(line.split('\t', maxsplit=1)) for l
ine in lines)
```

  于是将 `_Pinyin()` 抽取成公共变量，然后再次构建耗时为：

  2~3秒!!!!!!

3.2. 英文敏感词检测

当前设计思路中最小的状态切分是一个字母，即 `三` 字的最小切分粒度是：`s -> a -> n`，这种切分方式在进行中文同音字检查时提供了方便，但是坏处也显而易见——打破了英文单词分词规则

`cat` 单词会被拆分为 `c -> a -> t` 状态转移，就将导致任何包含了 `cat` 序列的词都会被视为敏感词！

目前采取的解决方案是对于用户的纯英文输入不做敏感词校验，听上去很荒唐，但是实际是当一串英文放在眼前，非母语者难辨别出敏感词。

4. 总结

本次会议主要分享DFA算法的演进、算法在项目中的套用以及项目优化过程。算法与数据结构、设计模式这些偏向理论的知识仿佛远在天边，实则我们每天都在接触，理解并合理使用它们对于我们的程序设计和性能优化的帮助是十分显著的。

敏感词也好风控也好，不可能是一劳永逸的，其所能达到的效果也并不是一触即得的，它是黑产玩家与运营人员之间的一场漫长的博弈，相互进化相互突破。