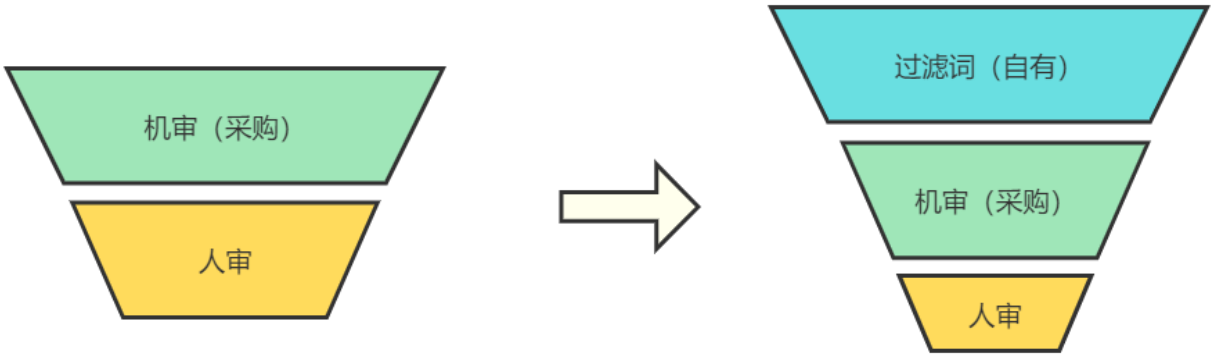
过滤词服务设计

2022 年是降本增效的一年。为了节省机审、人审成本，我们提出过滤词服务，一定程度的减少了流入机审跟人审环节的审核量。

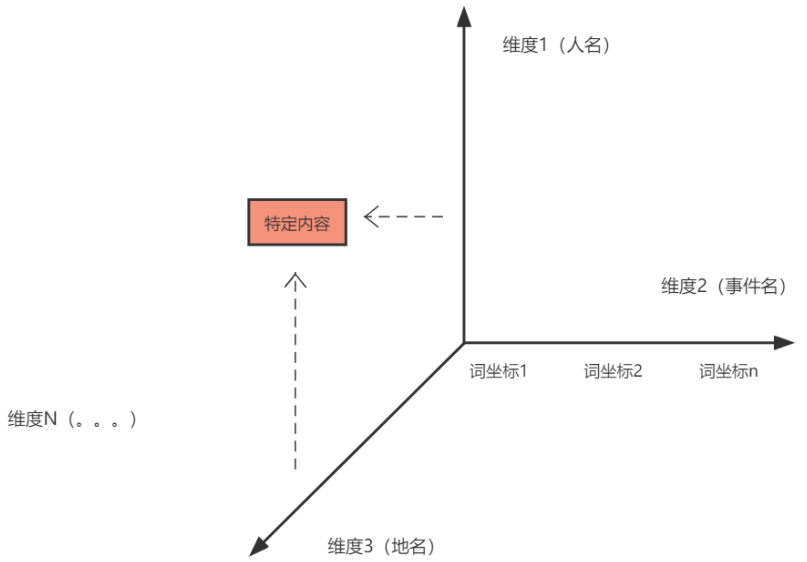


什么是过滤词服务

通过对业务场景分析，我们发现在一些环节，如果一条文本内容包含某些词条或词组，即可准确判定为违规内容，无需机审和人审。 特殊管控时期（国家公祭日、政治敏感时期等），处置劣迹艺人作品等环节。对于非强管控业务，过滤词可以将文本内容中出现的疑似违规词高亮，有助于提升人工审核的效率。当然，过滤词服务也可以在内容生成时，对用户做一些提示或限制，将审核前置，从而提升用户体验和审核效率。

审核方式	时效性	准确性	成本	适用场景
过滤词	毫秒级	高	低	特定
算法机审	秒级	中	中	大都数
人审	小时级	高	高	所有

业务模型



我们对一条文本内容的主成分进行分解，可以得到内容的关键词（特征值）。反过来说，当我们通过不同维度的关键词来定位某一违规性质的内容。维度越多，特征词条越多，关键词定位的精度越高。用来过滤违规内容的词条，我将其命名为【过滤词】。

维度	词条	审核标准解释	准确系数
动作	上门	色情内容高频词 但可能存在误伤，比如快递短信：上门取件	1
联系方式	+V 10086	获取联系方式	2
外链	www.baidu.com	违规外链	3
语气	cao	存在煽动倾向	4

以上示例解释了 4 个不同维度的关键词可以非常精准的过滤一些特定模式的违规内容。

算法设计

判断一个文本中是否包含某个字符串，是一个经典的字符串查找问题。

穷举法

常见的字符串穷举法遍历，其算法复杂度为 **O(n*m)**。如果要判断一个长文本中是否有多个词，那么时间复杂度就是 **O(n*m*k)**。随着词库的增长，查找时间会越来越长。随着业务管控的加强，词库的数量逐步增加，单次查找的耗时也会随之增加。

> n 为待检测文本的长度
> m 为违禁词的长度
> k 为词库中违禁词的总数

算法 1 用暴力搜索查找违禁词

输入: $Lexicon$词库（二维数组），$text$待检测字符串
输出: 是否包含违禁词

```
1:  function HasLegalText(Lexicon, text)
2:      for i = 0 → Lexicon.length do
3:          result ← true
4:          for j = 0 → Lexicon[i].length do
5:              if Lexicon[i][j] ∉ text then
6:                  result ← false
7:                  break
8:              end if
9:          end for
10:         if result = true then
11:             return true
12:         end if
13:     end for
14:     return false
15: end function
16:
```

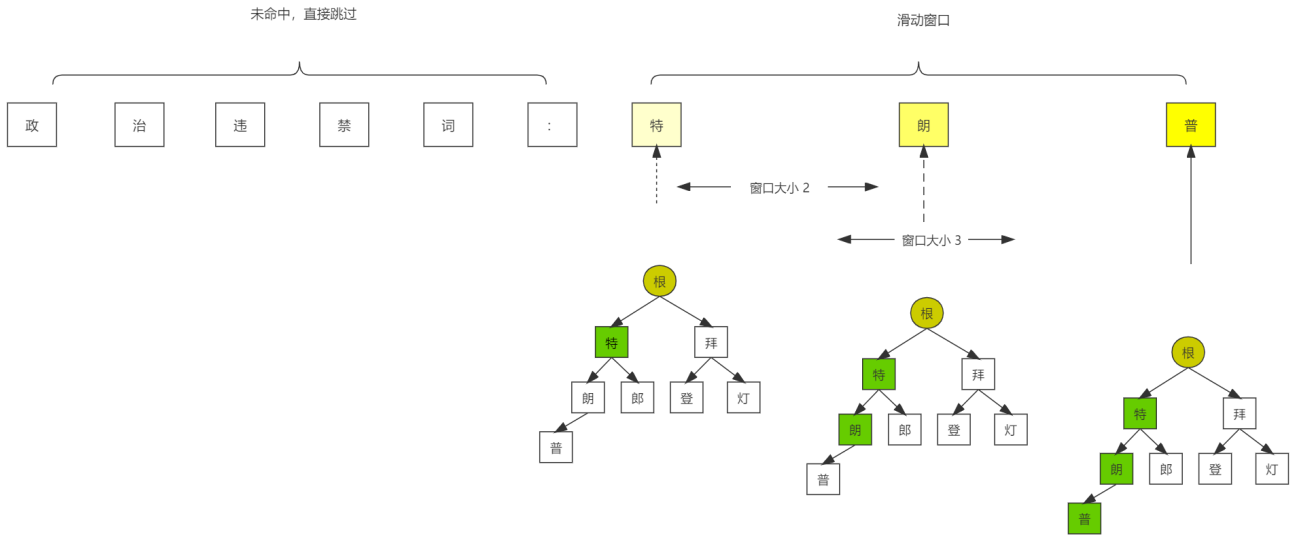
Trie 树查找

为了降低查找的算法复杂度，我选了 Trie 树查找算法。

- 首先，用词库中的词条构建一棵 Trie 树；
- 遍历待检测文本时，通过滑动窗口的方式对其进行分词；
- 然后在 Trie 树中查找是否包含分词结果，如果包含判断为命中；

最坏的算法复杂度为 **O(n*m*k)**。m 跟 k 都为常量（**O(1)**），词库再大都没有压力。

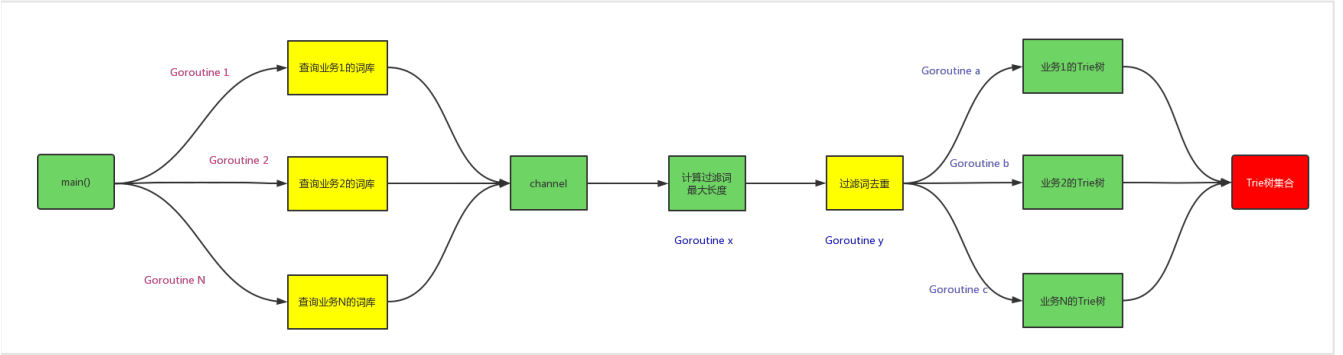
> n 为待检测文本的长度
> m 为词库中最大的词条长度。也就是分词的长度
> k 为 Trie 树一次查找的复杂度



系统设计

并发设计

目前（2023 年 5 月）生产环境有 117 个业务接入过滤词服务，单个业务最大的词条数为 1.63W。服务启动时就会拉取全部词库，然后根据词库中的词条构建 Trie 树。随着接入业务的增多，构建 Trie 树的时间越来越长，服务启动的耗时也越来越长。为了解决这个问题，我通过编排 Go 语言的协程，提升了词库的加载性能。查询词库、词条预处理、构建 Trie 树，作为一条流水线。不同业务间并发处理。结合 Go 语言并发模式中的【Pipeline】模式与【生产者 - 消费者】模式，将原本分钟级的耗时优化至秒级。



Go

```go
package main

var TrieMap = make(map[string]*Trie, 16)

type Trie struct{}

type Words []string

func loadWords(teanantId string) <-chan Words {
    ch := make(chan Words, 16)
    go func() {
        println("根据租户ID拉取对应的词库")
        ch <- Words{"foo"}
        close(ch)
    }()
    return ch
}

func processor(w <-chan Words) <-chan Words {
    ch := make(chan Words, 16)
    go func() {
        for item := range w {
            println("预处理")
            ch <- item
        }
        close(ch)
    }()
    return ch
}

func buildTrie(w <-chan Words) <-chan *Trie {
    ch := make(chan *Trie, 16)
    go func() {
        println("根据词库构建前缀树")
        ch <- &Trie{}
        close(ch)
    }()
    return ch
}
```

```
40
41 func buildTrieMap(t <-chan *Trie) {
42     go func() {
43         println("构建前缀树map")
44         for trie := range t {
45             TrieMap["租户ID"] = trie
46         }
47     }()
48 }
49
50 func main() {
51     buildTrieMap(buildTrie(processor(loadWords("12345"))))
52 }
```

扩展思路

将上述功能做成一个 Redis 插件。