

冰乐

我亦无他，惟手熟尔！

博客园 首页 新随笔 联系 订阅 管理

Golang使用前缀树算法实现敏感词过滤

大家都知道游戏文字、文章等一些风控场景都实现了敏感词检测，一些敏感词会被屏蔽掉或者文章无法发布。打游戏的时候不要骂人，骂人是不对的，但有时候无奈化身为肖邦。

今天我就分享用Go实现敏感词前缀树来达到文本的敏感词检测，让我们一探究竟！

敏感词检测

实现敏感词检测都很多种方法，例如暴力、正则、前缀树等。例如一个游戏的文字交流的场景，敏感词会被和谐成*，该如何实现呢？首先我们先准备一些敏感词如下：

复制代码

```
sensitiveWords := []string{
    "傻逼",
    "傻叉",
    "垃圾",
    "妈的",
    "sb",
}
```

复制代码

由于文章审核原因敏感词就换成别的了，大家能理解意思就行。

当在游戏中输入 **什么垃圾打野，傻逼一样，叫你来开龙不来，sb**，该如何检测其中的敏感词并和谐掉

暴力匹配

复制代码

```
sensitiveWords := []string{
    "傻逼",
    "傻叉",
    "垃圾",
    "妈的",
    "sb",
}

text := "什么垃圾打野，傻逼一样，叫你来开龙不来，sb"

for _, word := range sensitiveWords {
    text = strings.Replace(text, word, "*", -1)
}
```

公告

昵称：冰乐
园龄：5年4个月
粉丝：74
关注：49
[+加关注](#)

<	2023年9月						>
日	一	二	三	四	五	六	
27	28	29	30	31	1	2	
3	4	5	6	7	8	9	
10	11	12	13	14	15	16	
17	18	19	20	21	22	23	
24	25	26	27	28	29	30	
1	2	3	4	5	6	7	

搜索

找找看

谷歌搜索

我的标签

dotnet core(27)

SOA(18)

Go(18)

C#(14)

```
println("text -> ", text)
```

复制代码

这样采用的Go的内置的字符串替换的方法来进行暴力替换结果如下：

```
text ->  什么*打野，*一样，叫你来开龙不来，*
```

但暴力替换的时间复杂度太高了O（N^2），不建议这样，而且和谐的字符只有一个*，感觉像屏蔽了一个字一样，因此改造一下并引出go中的 `rune` 类型。

复制代码

```
sensitiveWords := []string{
    "傻逼",
    "傻叉",
    "垃圾",
    "妈的",
    "sb",
}

text := "什么垃圾打野，傻逼一样，叫你来开龙不来，sb"

for _, word := range sensitiveWords {
    replaceChar := ""
    for i, wordLen := 0, len(word); i < wordLen; i++ {
        // 根据敏感词的长度构造和谐字符
        replaceChar += "*"
    }
    text = strings.Replace(text, word, replaceChar, -1)
}

println("text -> ", text)
```

复制代码

输出如下：

```
text ->  什么*****打野，*****一样，叫你来开龙不来，**
```

为什么中文的和谐字符多了这么？*

因为Go中默认采用utf-8来进行中文字符编码，因此一个中文字符要占3个字节

```
text := "什么垃圾打野，傻逼一样，叫你来开龙不来，sb"
```

```
for _, word := range sensitiveWords {
    replaceChar := ""
    wordBytes := []byte(word)
    println(wordBytes, len(wordBytes))
    for i, wordLen := 0, len(word); i < wordLen; i++ {
        // 根据敏感词的长度构造和谐字符
        replaceChar += "*"
    }
    text = strings.Replace(text, word, replaceChar, -1)
}
```

```
[6/32]0xc000033ee0 6
[6/32]0xc000033ee0 6
[6/32]0xc000033ee0 6
[6/32]0xc000033ee0 6
[2/32]0xc000033ee0 2
```

```
text ->  什么 *****打野， *****一样，叫你来开龙不来， **
DE D:\goproject\src\text\
```

设计模式(11)

dotnet mvc(9)

前端(6)

数据库(5)

数据结构与算法(5)

微服务(3)

Redis(2)

RabbitMQ(2)

ORM(2)

MYSQL(2)

docker(2)

代码管理(2)

OOP(1)

k8s(1)

gRPC(1)

GORM(1)

AOP(1)

架构设计(1)

工具小技巧(1)

工具(1)

积分与排名


积分 - 153630

排名 - 8142

因此引出 Go 中的 rune 类型，它可以代表一个字符编码的int32的表现形式，就是说一个字符用一个数字唯一标识。有点像 ASCII 码一样 a => 97, A => 65

源码解释如下


```
// rune is an alias for int32 and is equivalent to int32 in all ways. It is
used, by convention, to distinguish character values from integer values.
type rune = int32
```

复制代码

```
fmt.Println("a -> ", rune('a'))
fmt.Println("A -> ", rune('A'))

fmt.Println("晖 -> ", rune('晖'))
fmt.Println("霞 -> ", rune('霞'))


fmt.Println("晖霞 -> ", []rune("晖霞"))
```

复制代码

输出如下：

```
a -> 97
A -> 65
晖 -> 26198
霞 -> 38686
晖霞 -> [26198 38686]
```

因此将敏感词字符串转换成rune类型的数组然后来计算其字符个数

复制代码

```
sensitiveWords := []string{
    "傻逼",
    "傻叉",
    "垃圾",
    "妈的",
    "sb",
}


text := "什么垃圾打野，傻逼一样，叫你来开龙不来，sb"

for _, word := range sensitiveWords {
    replaceChar := ""

    for i, wordLen := 0, len([]rune(word)); i < wordLen; i++ {
        // 根据敏感词的长度构造和谐字符
        replaceChar += "*"
    }

    text = strings.Replace(text, word, replaceChar, -1)
}

println("text -> ", text)
```

复制代码

输出如下：

```
text -> 什么**打野，**一样，叫你来开龙不来，**
```

正则匹配

复制代码

```
sensitiveWords := []string{
    "傻逼",
    "傻叉",
    "垃圾",
    "妈的",
    "sb",
}
```

随笔档案 (121)

2023年5月(2)

2022年9月(2)

2022年2月(1)

2021年11月(2)

2021年7月(1)

2021年6月(1)

2021年5月(6)

2021年4月(8)

2021年3月(1)

2021年2月(2)

2021年1月(4)

2020年12月(1)

2020年11月(2)

2020年10月(17)

2020年8月(3)

更多

阅读排行榜

1. Vue+ElementUI的后台管理框架(186566)

2. WebApi简介(20044)

3. vue中使用echarts遇到的Error in v-on handler: "TypeError: Cannot read property 'getAttribute' of null"(17299)

4. MYSQL IN 一定走索引吗? (12725)

```
}

text := "什么垃圾打野，傻逼一样，叫你来开龙不来，sb"

// 构造正则匹配字符
regStr := strings.Join(sensitiveWords, "|")
println("regStr -> ", regStr)
wordReg := regexp.MustCompile(regStr)
text = wordReg.ReplaceAllString(text, "**")

println("text -> ", text)
```



输出如下：

```
regStr ->  傻逼|傻叉|垃圾|妈的|sb
text  ->  什么*打野，*一样，叫你来开龙不来，*
```

再优化下：



```
package main

import (
    "fmt"
    "regexp"
    "strings"
)

func main() {
    sensitiveWords := []string{
        "傻逼",
        "傻叉",
        "垃圾",
        "妈的",
        "sb",
    }

    matchContents := []string{
        "什么垃圾打野，傻逼一样，叫你来开龙不来，sb",
    }

    regDemo(sensitiveWords, matchContents)
}

// 正则匹配敏感词
func regDemo(sensitiveWords []string, matchContents []string) {

    banWords := make([]string, 0) // 收集匹配到的敏感词

    // 构造正则匹配字符
    regStr := strings.Join(sensitiveWords, "|")
    wordReg := regexp.MustCompile(regStr)
    println("regStr -> ", regStr)

    for _, text := range matchContents {
        textBytes := wordReg.ReplaceAllFunc([]byte(text), func(bytes []byte)
[]byte {
            banWords = append(banWords, string(bytes))
            textRunes := []rune(string(bytes))
            replaceBytes := make([]byte, 0)
            for i, runeLen := 0, len(textRunes); i < runeLen; i++ {
                replaceBytes = append(replaceBytes, byte('*'))
            }
            return replaceBytes
        })
        fmt.Println("srcText      -> ", text)
```

5. OAuth2、OpenID Connect简介(11198)

评论排行榜

- 1. .Net Core3.0 WebApi 四:JWT权限验证 (12)
- 2. ASP.NET Core 3.0 WebApi中使用Swagger生成API文档简介(9)
- 3. JavaScript中const,var,let区别与用法(6)
- 4. .Net Core3.0 WebApi 十五：使用Serilog替换掉Log4j(4)
- 5. OAuth2、OpenID Connect简介(4)

推荐排行榜

- 1. ASP.NET Core 3.0 WebApi中使用Swagger生成API文档简介(10)
- 2. Vue+ElementUI的后台管理框架(6)
- 3. MYSQL IN 一定走索引吗？ (4)
- 4. DotNet Core中使用dapper(4)
- 5. JavaScript中const,var,let区别与用法(3)

最新评论

1. Re:.Net Core3.0 WebApi 十三：自定义返回Json大小写格式

部分接口返回

--_York

2. Re:.Net Core3.0 WebApi 五:项目分层

老哥，能把整个项目的目录结构截图分享一下吗？谢谢

--喜欢写代码的小陈

3. Re:.Net Core3.0 WebApi 八:使用Redis做数据缓存

```
fmt.Println("replaceText -> ", string(textBytes))
fmt.Println("sensitiveWords -> ", banWords)
}
}
```

复制代码

输出如下：

```
regStr ->  傻逼|傻叉|垃圾|妈的|sb
srcText      ->  什么垃圾打野，傻逼一样，叫你来开龙不来，sb
replaceText  ->  什么**打野，**一样，叫你来开龙不来，**
sensitiveWords ->  [垃圾 傻逼 sb]
```

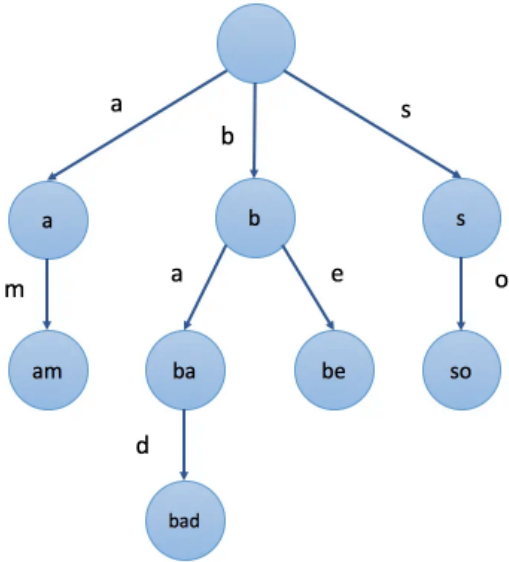
这里是通过敏感词去构造正则表达式然后再去匹配。

本文重点是使用Go实现前缀树完成敏感词的匹配，具体细节都在这里实现。

Golang 实现敏感词前缀树

前缀树结构

前缀树、也称字典树（Trie），是N叉树的一种特殊形式，前缀树的每一个节点代表一个字符串（前缀）。每一个节点会有多个子节点，通往不同子节点的路径上有着不同的字符。子节点代表的字符串是由节点本身的原始字符串，以及通往该子节点路径上所有的字符组成的。



如上图所示，就是一颗前缀树，注意前缀树的根节点不存数据。那么我们该如何表示一颗前缀树呢？

可以参考一下二叉树的节点结构

```
type BinTreeNode struct {
    Val      string
    LeftChild *BinTreeNode
    RightChild *BinTreeNode
}
```

二叉树，一个节点最多只能有两个孩子节点，非常明确，而前缀是一颗多叉树，一个节点不确定有多少子节点，因此可以用 切片 `Slice` 、 `Map` 来存储子节点，然后一般会设置标志位 `End` 来标识是否是字符串的最后一个节点。结构如下

复制代码

```
// TrieNode 敏感词前缀树节点
type TrieNode struct {
    childMap map[rune]*TrieNode // 本节点下的所有子节点
    Data     string                 // 在最后一个节点保存完整的一个内容
```

写的很不错，学习了，有一点不足的地方是接口代码忘记贴了。

--農碼一生

4. Re:.Net Core3.0 WebApi 八:使用Redis 做数据缓存

IredisCacheManager这个接口怎么编辑的


--缘—

5. Re:.Net Core3.0 WebApi 四:JWT权限 验证

老张的哲学

--AdmininAdmin

```
    End      bool           // 标识是否最后一个节点
}
```

 复制代码

这里采用 Map 来存储子节点，更方便找子节点。key是rune类型（字符），value是子节点。Data则是在最后一个节点保存完整的一个内容。


```
// SensitiveTrie 敏感词前缀树
type SensitiveTrie struct {
    replaceChar rune // 敏感词替换的字符
    root        *TrieNode
}
```

这里再用另一个结构体来代表整个敏感词前缀树。

添加敏感词

添加敏感词用于构造一颗敏感词前缀树。

相对每个节点来说 childMap 都是保存 相同前缀 字符的子节点

 复制代码


```
// AddChild 前缀树添加
func (tn *TrieNode) AddChild(c rune) *TrieNode {

    if tn.childMap == nil {
        tn.childMap = make(map[rune]*TrieNode)
    }

    if trieNode, ok := tn.childMap[c]; ok {
        // 存在不添加了
        return trieNode
    } else {
        // 不存在
        tn.childMap[c] = &TrieNode{
            childMap: nil,
            End:     false,
        }
        return tn.childMap[c]
    }
}
```

 复制代码

敏感词前缀树则是一个完整的敏感词的粒度来添加

 复制代码

```
// AddWord 添加敏感词
func (st *SensitiveTrie) AddWord(sensitiveWord string) {
    // 将敏感词转换成rune类型(int32)
    tireNode := st.root
    sensitiveChars := []rune(sensitiveWord)
    for _, charInt := range sensitiveChars {
        // 添加敏感词到前缀树中
        tireNode = tireNode.AddChild(charInt)
    }
    tireNode.End = true
    tireNode.Data = sensitiveWord
}
```

 复制代码

具体是把敏感词转换成 []rune 类型来代表敏感词中的一个个 字符，添加完后再将最后一个字符节点的 End 设置True，Data为完整的敏感词数据。

可能这样还不好理解，举个例子：

 复制代码

```
package main

import "fmt"

// SensitiveTrie 敏感词前缀树
type SensitiveTrie struct {
    replaceChar rune // 敏感词替换的字符
    root        *TrieNode
}

// TrieNode 敏感词前缀树节点
type TrieNode struct {
    childMap map[rune]*TrieNode // 本节点下的所有子节点
    Data     string              // 在最后一个节点保存完整的一个内容
    End      bool               // 标识是否最后一个节点
}

// NewSensitiveTrie 构造敏感词前缀树实例
func NewSensitiveTrie() *SensitiveTrie {
    return &SensitiveTrie{
        replaceChar: '*',
        root:        &TrieNode{End: false},
    }
}

// AddWord 添加敏感词
func (st *SensitiveTrie) AddWord(sensitiveWord string) {

    // 将敏感词转换成utf-8编码后的rune类型(int32)
    tireNode := st.root
    sensitiveChars := []rune(sensitiveWord)
    for _, charInt := range sensitiveChars {
        // 添加敏感词到前缀树中
        tireNode = tireNode.AddChild(charInt)
    }
    tireNode.End = true
    tireNode.Data = sensitiveWord
}


// AddChild 前缀树添加子节点
func (tn *TrieNode) AddChild(c rune) *TrieNode {

    if tn.childMap == nil {
        tn.childMap = make(map[rune]*TrieNode)
    }

    if trieNode, ok := tn.childMap[c]; ok {
        // 存在不添加了
        return trieNode
    } else {
        // 不存在
        tn.childMap[c] = &TrieNode{
            childMap: nil,
            End:     false,
        }
        return tn.childMap[c]
    }
}

func main() {
    sensitiveWords := []string{
        "傻逼",
        "傻叉",
        "垃圾",
    }
}
```

```
st := NewSensitiveTrie()
for _, word := range sensitiveWords {
    fmt.Println(word, []rune(word))
    st.AddWord(word)
}
}
```

 复制代码

输出如下：

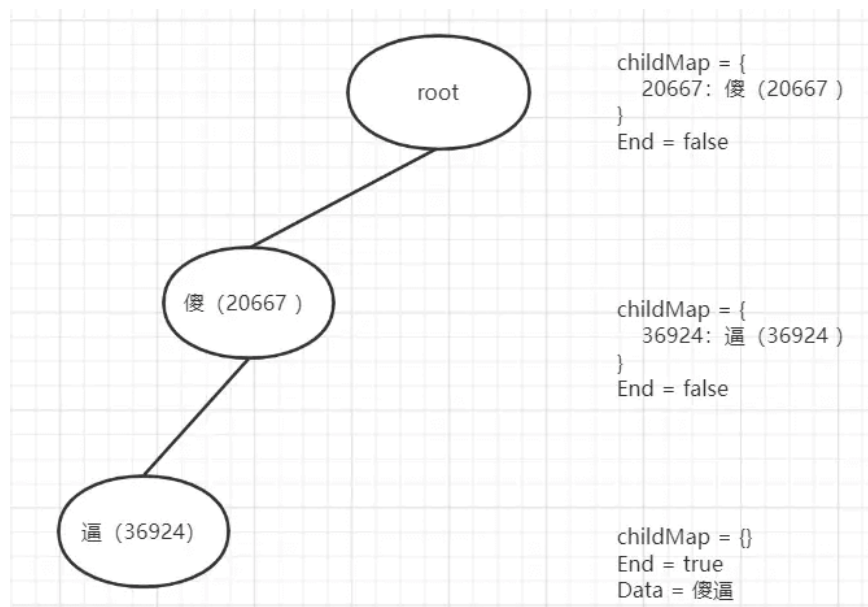
```
傻逼 [20667 36924]
傻叉 [20667 21449]
垃圾 [22403 22334]
```

添加前两个敏感词傻逼、傻叉，有一个共同的前缀 傻、`rune-> 200667`

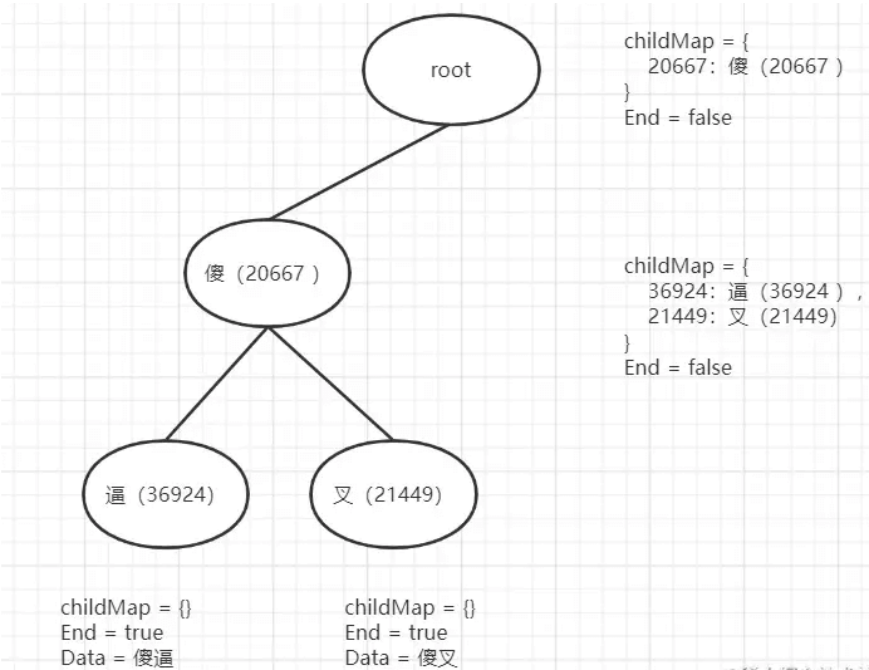
- 1.前缀的root是没有孩子节点，添加第一个敏感词时先转换成 `[]rune`（可以想象成字符数组）
- 2.遍历`rune`字符数组，先判断有没有孩子节点（一开始`root`是没有的），没有就先构造，然后把 傻（200667）存到 `childMap`中 key 为 傻(200667)，value 为 `TrieNode` 但没有任何数据然后返回当前新增的节点

```
TrieNode{
    childMap: nil
    End:      false,
}
```

- 3.此时添加 逼（36924），同样做2的步骤，傻逼这个敏感词添加完成走出for循环，然后将 `End=true`、`Data=傻逼`。

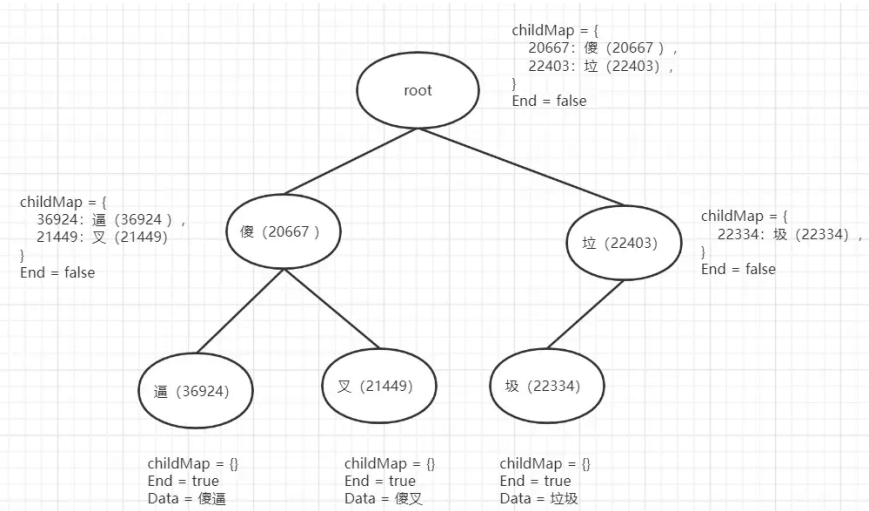


- 4.添加第二个敏感词傻叉的时候又是从根节点开始，此时`root`有`childMap`，也存在傻（20667）节点，则是直接不添加把傻（20667）节点返回，然后再此节点上继续添加 叉（21449），不存在添加到傻节点的`childMap`中。



5.添加第三个敏感词 垃圾 ，又从根节点开始， 垃 (22403) ，根节点不存在该子节点，故添加到根节点的childMap中，然后返回新增的 垃 (22403) 节点

6.在垃节点基础上添加 圾 (22334) 节点，不存在子节点则添加并返回。



由此一颗敏感词前缀树就构造出来了。

小结：添加敏感词字符节点存在不添加返回存在的节点，不存在添加新字符节点并返回新添节点，当敏感词的所有字符都添加完毕后，让最后一个节点，End=true，存储一个完整的敏感词。

匹配敏感词

将待匹配的内容转换成 []rune 类型，然后遍历寻找前缀树种第一个匹配的前缀节点，然后从后一个位置继续，直到完整匹配到了敏感词，将匹配文本的敏感词替换成 *

复制代码

```
// FindChild 前缀树寻找字节节点
func (tn *TrieNode) FindChild(c rune) *TrieNode {
    if tn.childMap == nil {
        return nil
    }

    if trieNode, ok := tn.childMap[c]; ok {
```

```
        return trieNode
    }
    return nil
}

// replaceRune 字符替换
func (st *SensitiveTrie) replaceRune(chars []rune, begin int, end int) {
    for i := begin; i < end; i++ {
        chars[i] = st.replaceChar
    }
}

// Match 查找替换发现的敏感词
func (st *SensitiveTrie) Match(text string) (sensitiveWords []string,
replaceText string) {
    if st.root == nil {
        return nil, text
    }

    textChars := []rune(text)
    textCharsCopy := make([]rune, len(textChars))
    copy(textCharsCopy, textChars)
    for i, textLen := 0, len(textChars); i < textLen; i++ {
        trieNode := st.root.FindChild(textChars[i])
        if trieNode == nil {
            continue
        }

        // 匹配到了敏感词的前缀, 从后一个位置继续
        j := i + 1
        for ; j < textLen && trieNode != nil; j++ {
            if trieNode.End {
                // 完整匹配到了敏感词, 将匹配的文本的敏感词替换成 *
                st.replaceRune(textCharsCopy, i, j)
            }
            trieNode = trieNode.FindChild(textChars[j])
        }

        // 文本尾部命中敏感词情况
        if j == textLen && trieNode != nil && trieNode.End {
            if _, ok := sensitiveMap[trieNode.Data]; !ok {
                sensitiveWords = append(sensitiveWords, trieNode.Data)
            }
            sensitiveMap[trieNode.Data] = nil
            st.replaceRune(textCharsCopy, i, textLen)
        }
    }

    if len(sensitiveWords) > 0 {
        // 有敏感词
        replaceText = string(textCharsCopy)
    } else {
        // 没有则返回原来的文本
        replaceText = text
    }

    return sensitiveWords, replaceText
}
```

 复制代码

这样需要注意的是在内容的末尾匹配到了的敏感词处理, 因为j+1后, 会等于textLen的从而不进入for循环从而没有处理末尾, 因此需要特殊处理下末尾情况。具体测试如下

 复制代码

```
// AddWords 批量添加敏感词
func (st *SensitiveTrie) AddWords(sensitiveWords []string) {
    for _, sensitiveWord := range sensitiveWords {
        st.AddWord(sensitiveWord)
    }
}

// 前缀树匹配敏感词
func trieDemo(sensitiveWords []string, matchContents []string) {

    trie := NewSensitiveTrie()
    trie.AddWords(sensitiveWords)

    for _, srcText := range matchContents {
        matchSensitiveWords, replaceText := trie.Match(srcText)
        fmt.Println("srcText      -> ", srcText)
        fmt.Println("replaceText   -> ", replaceText)
        fmt.Println("sensitiveWords -> ", matchSensitiveWords)
        fmt.Println()
    }

    // 动态添加
    trie.AddWord("牛大大")
    content := "今天，牛大大去挑战灰大大了"
    matchSensitiveWords, replaceText := trie.Match(content)
    fmt.Println("srcText      -> ", content)
    fmt.Println("replaceText   -> ", replaceText)
    fmt.Println("sensitiveWords -> ", matchSensitiveWords)
}

func main() {
    sensitiveWords := []string{
        "傻逼",
        "傻叉",
        "垃圾",
        "妈的",
        "sb",
    }
    matchContents := []string{
        "你是一个大傻逼，大傻叉",
        "你是傻@叉",
        "shabi东西",
        "他made东西",
        "什么垃圾打野，傻逼一样，叫你来开龙不来，SB",
        "正常的内容@",
    }

    //fmt.Println("----- 普通暴力匹配敏感词 -----")
    //normalDemo(sensitiveWords, matchContents)
    //
    //fmt.Println("\n----- 正则匹配敏感词 -----")
    //regDemo(sensitiveWords, matchContents)

    fmt.Println("\n----- 前缀树匹配敏感词 -----")
    trieDemo(sensitiveWords, matchContents)
}
```

 复制代码

运行结果如下：

 复制代码

```
----- 前缀树匹配敏感词 -----
srcText      ->  你是一个大傻逼，大傻 叉
```

```
replaceText    -> 你是一个大傻&逼。大傻 叉
sensitiveWords -> []

srcText        -> 你是傻@叉
replaceText    -> 你是傻@叉
sensitiveWords -> []

srcText        -> shabi东西
replaceText    -> shabi东西
sensitiveWords -> []

srcText        -> 他made东西
replaceText    -> 他made东西
sensitiveWords -> []

srcText        -> 什么垃 圾打野，傻 逼一样，叫你来开龙不来，傻 逼东西，S B
replaceText    -> 什么**打野，**一样，叫你来开龙不来，**
sensitiveWords -> [垃圾 傻逼]

srcText        -> 正常的内容@
replaceText    -> 正常的内容@
sensitiveWords -> []
```



过滤特殊字符

可以发现在敏感词内容的中间添加一些空格、字符、表情都不能正确的在前缀树中匹配到。因此我们在进行匹配的时候应该过滤一些特殊的字符，只保留汉字、数字、字母，然后全部以小写来进行匹配。

```
// FilterSpecialChar 过滤特殊字符
func (st *SensitiveTrie) FilterSpecialChar(text string) string {
    text = strings.ToLower(text)
    text = strings.Replace(text, " ", "", -1) // 去除空格

    // 过滤除中英文及数字以外的其他字符
    otherCharReg := regexp.MustCompile("[^\u4e00-\u9fa5a-zA-Z0-9]")
    text = otherCharReg.ReplaceAllString(text, "")
    return text
}
```



感觉这里去除空格是多余的步骤，正则以已经帮你排除了。

- `\u4e00-\u9fa5a` 代表所有的中文
- `a-zA-Z` 代表大小写字母
- `0-9` 数字
- 连起来在最前面加上一个 `^` 就是进行一个取反

添加拼音检测

最后就是添加中文的拼音检测，让输入的拼音也能正确的匹配到，拼音检测是把我们的敏感词转换成拼音然后添加到前缀树中。

实现中文转拼音可以用别人造好的轮子

```
go get github.com/chain-zhang/pinyin
```

查看源码整体的思路就是用文件把文字的rune和拼音对应上，具体细节自行查看

测试一下

```
// HansCovertPinyin 中文汉字转拼音
func HansCovertPinyin(contents []string) []string {
```




```
pinyinContents := make([]string, 0)
for _, content := range contents {
    chineseReg := regexp.MustCompile("[\u4e00-\u9fa5]")
    if !chineseReg.Match([]byte(content)) {
        continue
    }

    // 只有中文才转
    pin := pinyin.New(content)
    pinStr, err := pin.Convert()
    println(content, "->", pinStr)
    if err == nil {
        pinyinContents = append(pinyinContents, pinStr)
    }
}
return pinyinContents
}

func main() {
    sensitiveWords := []string{
        "傻逼",
        "傻叉",
        "垃圾",
        "妈的",
        "sb",
    }

    // 汉字转拼音
    pinyinContents := HansCovertPinyin(sensitiveWords)
    fmt.Println(pinyinContents)
}
```

 复制代码

输出结果如下：

```
傻逼 -> sha bi
傻叉 -> sha cha
垃圾 -> la ji
妈的 -> ma de
[sha bi sha cha la ji ma de]
```

然后再测试敏感词匹配的效果

 复制代码

```
// Match 查找替换发现的敏感词
func (st *SensitiveTrie) Match(text string) (sensitiveWords []string,
replaceText string) {
    if st.root == nil {
        return nil, text
    }

    // 过滤特殊字符
    filteredText := st.FilterSpecialChar(text)
    sensitiveMap := make(map[string]*struct{}) // 利用map把相同的敏感词去重
    textChars := []rune(filteredText)
    textCharsCopy := make([]rune, len(textChars))
    copy(textCharsCopy, textChars)
    for i, textLen := 0, len(textChars); i < textLen; i++ {
        ...
    }

    if len(sensitiveWords) > 0 {
        // 有敏感词
        replaceText = string(textCharsCopy)
    } else {
```

```
// 没有则返回原来的文本
replaceText = text
}

return sensitiveWords, replaceText
}

// 前缀树匹配敏感词
func trieDemo(sensitiveWords []string, matchContents []string) {

    // 汉字转拼音
    pinyinContents := HansCovertPinyin(sensitiveWords)
    fmt.Println(pinyinContents)

    trie := NewSensitiveTrie()
    trie.AddWords(sensitiveWords)
    trie.AddWords(pinyinContents) // 添加拼音敏感词

    for _, srcText := range matchContents {
        matchSensitiveWords, replaceText := trie.Match(srcText)
        fmt.Println("srcText      -> ", srcText)
        fmt.Println("replaceText   -> ", replaceText)
        fmt.Println("sensitiveWords -> ", matchSensitiveWords)
        fmt.Println()
    }

    // 动态添加
    trie.AddWord("牛大大")
    content := "今天, 牛大大去挑战灰大大了"
    matchSensitiveWords, replaceText := trie.Match(content)
    fmt.Println("srcText      -> ", content)
    fmt.Println("replaceText   -> ", replaceText)
    fmt.Println("sensitiveWords -> ", matchSensitiveWords)
}

func main() {
    sensitiveWords := []string{
        "傻逼",
        "傻叉",
        "垃圾",
        "妈的",
        "sb",
    }

    matchContents := []string{
        "你是一个大傻逼, 大傻叉",
        "你是傻@叉",
        "shabi东西",
        "他made东西",
        "什么垃 圾打野, 傻逼一样, 叫你来开龙不来, SB",
        "正常的内容@",
    }

    fmt.Println("\n----- 前缀树匹配敏感词 -----")
    trieDemo(sensitiveWords, matchContents)
}


```



结果如下:



```
----- 前缀树匹配敏感词 -----
srcText      ->  你是一个大傻逼, 大傻叉
replaceText   ->  你是一个大**大**
```

```
sensitiveWords -> [傻逼 傻叉]

srcText      -> 你是傻@叉
replaceText  -> 你是**
sensitiveWords -> [傻叉]

srcText      -> shabi东西
replaceText  -> *****东西
sensitiveWords -> [shabi]

srcText      -> 他made东西
replaceText  -> 他****东西
sensitiveWords -> [made]

srcText      -> 什么垃圾打野，傻逼一样，叫你来开龙不来，SB
replaceText  -> 什么**打野**一样叫你来开龙不来**
sensitiveWords -> [垃圾 傻逼 sb]

srcText      -> 正常的内容@
replaceText  -> 正常的内容@
sensitiveWords -> []

srcText      -> 今天，牛大大挑战灰大大
replaceText  -> 今天***挑战灰大大
sensitiveWords -> [牛大大]
```



整体效果还是挺不错的，但是一些谐音或者全部英文句子时有空格还是不能去除空格不然可能会存在误判还是不能检测出，要想充分的进行敏感词检测，首先要有完善的敏感词库，其次就是特殊情况特殊处理，最后就是先进行敏感词匹配然后再进行自然语言处理NLP完善，训练风控模型等检测效果才更只能。

标签: Go

好文要顶

关注我

收藏该文



冰乐
粉丝 - 74 关注 - 49

00

+加关注

[升级成为会员](#)

« 上一篇: Golang开发过程中常遇到得一些小错误

» 下一篇: MQTT协议简介

posted @ 2022-09-13 12:07 冰乐 阅读(925) 评论(0) 编辑 收藏 举报

努力加载评论中...

[刷新评论](#) [刷新页面](#) [返回顶部](#)

[发表评论](#) [升级成为园子VIP会员](#)

编辑

预览

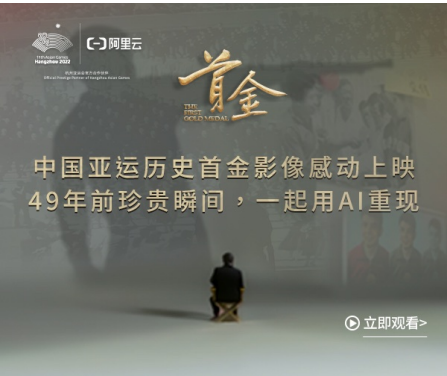
B

支持 Markdown

自动补全

提交评论 退出 订阅评论

[Ctrl+Enter快捷键提交]



Copyright © 2023 冰乐

Powered by .NET 7.0 on Kubernetes