ITN引擎数据结构设计

ITN主要对文本进行处理，将汉字形式的数字转写为阿拉伯，主要利用文法规则加权有限状态机实现。

# 1. 语法文件的构建

## 1.1 编写.grm文件

Thrax处理的文法文件.far是由.grm文件编译而成，接下来将举例介绍编写.grm文件步骤方法。

### 变量定义---“Optimize”

如下定义一个阿拉伯数字变量：

hanziDigit = Optimize["零"|"一"|"幺"|"二"|"两"|"三"|"四"|"五"|"六"|"七"|"八"|"九"];

### 变量叠加---“|”

如下实现两个变量叠加

measureUnit = Optimize["米" | "克" | "平" |"立"|"升"];

percentUnit = Optimize["之"];

kDictRemoveWithoutNumber = Optimize[percentUnit |measureUnit ];

那么kDictRemoveWithoutNumber = Optimize["米" | "克" | "平" |"立"|"升" |"之"];

### 变量剔除---“-”

Tmp = Optimize[kDictRemoveWithoutNumber – measureUnit]

那么 Tmp = Optimize["之"];

### 变量导出---export

export kDictRemoveWithoutNumber = Optimize["米" | "克" | "平" |"立"|"升" |"之"];

那么变量kDictRemoveWithoutNumber就可以在其它文件中使用；未加“export”则只能在本文件使用。

### 文件加载---stringFile

如果某个变量的内容非常多（如字典），可以通过该字符进行加载，如下：

dict = StringFile['dict.list','byte'];

那么就而已将文件dict.list的内容以“byte”格式加载赋给变量dict。这里需要说明的是thrax通常支持两种格式“utf8”和“byte”。

### 包含多次---“+”

Withoutnumber = Optimize[“ab”]

Tmp = Optimize[withoutnumber+];

表示Tmp可以匹配“ab”,“abab”,“ababab”等等。

### 可有可无---“\*”

匹配234556、234这种电话号码、身份证号等。

read\_digit\_by\_digit = (units | zero) (units | zero) (units | zero) ((units | zero))\*;

### 汉字替换阿拉伯---“：”

把零替换成0

delzero = "零" : "0";

## 1.2 .grm文件规则详解

本节对文法规则原理进行详细说明，主要实现变量组合，对涉及到的规则进行介绍；没有export的，但是比较复杂不易理解的变量也会进行说明，其他比较简单的变量这里不做赘述。

### numbers.grm

该文件主要实现对数字的处理，只举部分进行说明。

定义变量numbers\_to\_9999，数字范围为0-9999

export numbers\_to\_9999 = Optimize[numbers\_to\_99 | numbers\_100\_to\_999 | numbers\_1000\_to\_9999];

#其中：numbers\_to\_99：0-99

numbers\_to\_99 = Optimize[

(units|zero) | #匹配0-9

teens | #匹配10-19

decades | #匹配20、30、…、90

units ("十" : "") units ] #匹配22、35、99等这些数字

;

### number\_symbol.grm

该文件主要处理百分之、千分之问题，替换成%和‰，以及小数问题，替换成几点几。

import 'numbers.grm' as numberslib; #使用上面已经定义好的文法，import进来；

例如，

1）匹配小数

#匹配小数，并导出到变量kDecimal

export kDecimal = Optimize[

numbers decimal (Digit)+| #匹配九十九点二五->99.25

(Digit)+ decimal (Digit)+ #匹配三四点六六六->34.666

];

#其中，numbers为0-9999

numbers = numberslib.numbers\_to\_9999;

#小数点

decimal = ("点" : ".");

#Digit，小数点末尾数字

Digit =("零" : "0") | ("一" : "1") | ("幺" : "1") | ("二" : "2") | ("三" : "3") | ("四" : "4") | ("五" : "5") | ("六" : "6") | ("七" : "7") | ("八" : "8") | ("九" : "9");

2）匹配小数、百分之、千分之

#该变量涵盖了上面的kDecimal，同时包含百分之、千分之转为%，‰

export kNumberSymbol = Optimize[

percent (numbers|kDecimal) ("":"%") |

permill (numbers|kDecimal) ("":"‰") |

kDecimal

] ;

#其中，百分之、千分之定义如下：

percent = ("百分之" : "");

permill = ("千分之" : "");

### time.grm

该文件主要匹配时间信息，包括年月日、时分秒。

同样引用numbers.grm文法，里面变量units、teens、teens\_withoutten、decades的定义同numbers.grm文件里的变量定义，同时定义了months、dates、hours\_safe、hours\_unsafe、minutes、seconds等变量。

export kTIMES = Optimize[

date\_digit (time\_digit | "") |

date\_digit (hours\_safe | hours\_unsafe) ("点"|"时") |

(date\_digit | "") time\_digit

] ;

其中，date\_digit和time\_digit需要重点说明；

date\_digit = Optimize[

months "月" dates ("日" |"号")| #五月二十四号/日->5月24日

];

time\_digit = Optimize[

hours\_safe ("点"|"时") ((minutes "分") | "") ((seconds "秒") | "") | #三点、三点十分、三点十分二十五秒 等都可以匹配

hours\_unsafe ("点"|"时") (minutes "分") ((seconds "秒") | "") | #三点十分、三点十分三十秒

(hours\_safe | hours\_unsafe) "点" number\_up\_ten | #三点二十

];

### measure.grm

匹配计量单位，引入numbers.grm和number\_symbol.grm文法

export kMEASURES = Optimize[

(numbers | decimal | special\_unit) measures

] ;

#其中：计量单位包括以下几种

measures =

("克" : "g")

| ("千克" : "kg")

| ("平方米" : "m^2")

| ("平米" : "m^2")

| ("立方米" : "m^3")

| ("平" : "平")

| ("厘米" : "cm")

| ("毫米" : "mm")

| ("米" : "m")

| ("升" : "L")

| ("千米" : "km")

| ("千米每小时" : "km/h")

;

currency.grm

匹配金钱相关，引入numbers.grm和number\_symbol.grm文法

首先定义金钱相关的单位：

MoneyUnitL1 = Optimize["块" | "元"];

MoneyUnitL2 = Optimize["毛" | "角"];

MoneyUnitL3 = Optimize["分"];

export kMoney = Optimize[

numbers MoneyUnitL1 Digit MoneyUnitL2 ((Digit (MoneyUnitL3|""))|"") | #0-99999999元1-9毛、0-9999元1-9毛1-9分

(numbers | Digit) MoneyUnitL1 Digit (MoneyUnitL3|"") | #0-99999999块1-9（分）Digit MoneyUnitL2 Digit (MoneyUnitL3|"") | #1-9毛1-9（分）

(numbers | Digit) MoneyUnitL1 Digit | #1-9块1-9

(numbers | Digit) MoneyUnitL1 zero Digit (MoneyUnitL2 | MoneyUnitL3) | #0-99999999元零1-9分/角

Digit MoneyUnitL2 | #1-9角

(numbers | Digit) MoneyUnitL1 | #0-99999999元

Digit MoneyUnitL3 #1-9分

];

### addword.grm

该文件主要存放不想转的文本

export kAddWords = Optimize[

"一五一十" |

"十一国庆" |

"一年" |

"九牛一毛" |

"冰山一角" |

"点钟" |

"第一点"

];

### dict.grm

dict.list存放所有的汉字及字符，单字，通过以下方法加载；

dict = StringFile['dict.list','byte'];

export kDict = Optimize[ dict ];

#0-9

export kDictNumber = Optimize[hanziDigit];

#除了数字之外的所有特殊的字符

export kDictRemoveWithoutNumber = Optimize[kDictSpace | kCalUnit | timeUnit | measureUnit | money | percentUnit | dataflow];

#从识别结果中挖去所有需要特殊处理的字符

export kDictNotSpace = Optimize[kDict - kDictSpace - kCalUnit - hanziDigit - timeUnit- measureUnit - money - percentUnit - dataflow];

以上export的全部用在exampl.grm中构建解码空间；

### data\_flow.grm

电信数据中较容易涉及到兆，g等，所以增加该文法

data\_flow =

("兆" : "M")

| ("g" : "G")

;

export kDATAFLOW = Optimize[

(numbers | decimal | special\_unit) data\_flow

] ;

### exampl.grm

总的grm文件，对上面的文法进行组合，生成exampl.far文件，可供程序使用；

里面主要包含rules/rules\_safe/word1/word2变量，最后组成word变量，进行匹配；

rules/rules\_safe，把前面定义好的文法组合，赋值给该变量，该变量主要负责进行替换；

word1和word2，和rules进行组合，除rules之外为不替换的特殊字符

最后word对word1和word2进行组合，实现数字转阿拉伯；

word1=Optimize[

((removeword\_number (removeword\_number | ""))<1.7>) |

(((removeword\_withoutnumber - "十")+)<1.2>) |

(((removeword\_withoutnumber - "十" )+ removeword\_number+)<1.5>) |

((("" | (removeword\_withoutnumber - "百" - "十" - "万")+) rules)<1.0>) |

((("" | (removeword\_withoutnumber - "十")+) rules\_close)<1.1>) |

rules\_safe (removeword\_number)+ |

addwords |

""

];

word2=Optimize[

rules |

""

];

word = Optimize [

(word1 (anyword | "之"))+ word1 (anyword | "") |

word1 |

word2

];

该文件为构建的解码空间，要保证你送进去转写的字符串在该空间中可以找到路径，其中道理也是根据出现问题进行规则修改总结出来的；

<1.5>此为权重，应该是值越小，越优先走该路径；

rule\_close是为了解决规则紧邻问题定义的特殊处理的方法；

# 2. 程序设计

## 2.1 TitTranslate

该类为接口类，该类实现对识别结果后处理，类定义存放于translate-api.h文件中。

class TitTranslate{

public:

//初始化：每个进程只需初始化一次

bool Init(const int nThread, const char \*grammarfile, const bool bMergeTime);

//处理：每个线程调用一次

bool DoTranslate(const TitText \*input, TitText \*\*output, const int sessionId);

//退出引擎：每个进程进行一次

bool Exit();

TitTranslate() {};

private:

//该类实现重写

void \*\*rewrite;

//该类实现时间合并，通过Init成员函数进行设置

void \*\*mergetime;

//是否需要进行时间合并，如果仅仅想进行文本转译，置为false

bool bmergetime;

//每个线程结果

TitText \*\*resultText;

//线程数，通过Init成员函数进行设置

int threadNum;

//简单的释放内存

void FreeText(TitText \*text);

};

## 2.2 MergeRewriteTime

该类实现时间点合并，类定义存放于merge-rewrite-time.h文件中。

class MergeRewriteTime{

public:

//删除所有空格，更新至srcstr

void DeleteAllSpace(std::string &srcstr, std::vector<PointNode> &segNodeVec);

//将原字符串strA和转译后的字符串strB进行比较

// bBackTrace=true，进行路径回溯；bBackTrace=false，不进行路径回溯

int EditDistance(const std::string &strA, const std::string &strB, const bool bBackTrace);

//根据EditDistance的回溯路径及原分词时间（sentTime息（newTime）

//这里需要注意的是sentTime为原始时间，newTime为合并后的时间

bool MergeTime(const std::string &sentTime, const std::vector<PointNode> segNodeVec, std::string &newTime);

MergeRewriteTime();

~MergeRewriteTime();

private:

//normal: row->base; col->rewrite

int \*\*operationPath; //保存路径

int \*\*operationDistance; //保存距离

int operationRow; //原字符串长度

int operationCol; //转译之后字符串长度

std::string strRow; //原字符串

std::string strCol; //转译后字符串

std::vector<PointNode> rewritePosRow; //原字符串进行转译的位置

std::vector<PointNode> rewritePosCol; //转译后字符串发生转译的位置

}

## 2.3 TitRewrite

class TitRewrite{

public:

////初始化，加载文法文件

bool Init(const char \*grammar);

//重写：根据grammar对input进行重写，保存至output

bool Rewrite(const std::string &input, std::string &output);

//退出

bool Exit();

private:

void \*utils;

};

3.2.4 PointNode

用于记录位置信息，如发生转译的开始位置和结束位置

typedef struct PointNode{

int start;

int end;

};