# 正则表达式引擎设计文档 v1.0

## 1介绍

正则表达式通常被用在词法分析里面，例如用来表达token的规则。 但是它的应用场景远不止这些，例如字符串的查找，替换以及验证，特别是在当前大数据环境下，对数据进行的预处理例如清洗等都需要用到。当我们需要使用一些规则来定义和处理字符串时就会用到它。换句话说，本质上正则表达式提供了一种简洁的语法使得我们可以非常优雅的表达一系列的字符串。

## 2总体设计

本正则表达式引擎采用C++语言开发，

这是个命令行程序，输入一个正则表达式，以及一个文件（目前限定Ascii字符），程序运行结束后输出一个文件，该文件包含输入文件中与正则表达式匹配的所有字符。第一版只支持基本正则表达式。

整体的设计框架如下：

基本正则表达式的语法：

要定义正则表达式的语法，首先得定义字符串的基本操作，它们是∪，连接，Kleene闭包以及Positive闭包，它们分别定义如下：

* 并集：
* 连接：
* Kleene闭包：
* Positive闭包：

有了上面的字符串操作，基本正则表达式的语法可以定义如下：

1. **规则可以把一个特定的字符或者是空字符串认为是一种类型的记号的全部，也就是说，空字符串是正则表达式，以及任意一个来自字符集的字符a也是正则表达式；**
2. **假设r和s分别都是正则表达式，则(r)|(s)也是正则表达式，表达并集的意思；**
3. **假设r和s分别都是正则表达式，则(r)(s)也是正则表达式，表达字符串的连接操作；**
4. **假设r是正则表达式，则也是正则表达式，表达Kleene闭包操作；**
5. **如果r是正则表达式，则(r)也是，也就是说正则表达式周围加个括号不改变想要表达的字符串；**

正则表达式操作符的优先级：

* \*操作符（Kleene闭包）具有最高优先级，同时是从左开始匹配；
* 连接操作第二，同时是从左开始匹配；
* |具有最低优先级，也是从左开始匹配；

## 3细节设计

整个引擎的实现分为如下几步：

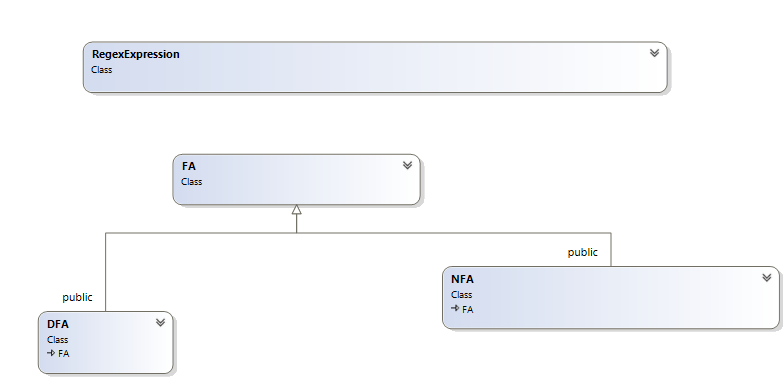
1. 将正则表达式转换为语法树；

利用双堆栈实现，类似四则运算，只是操作符和操作数不同；

1. 使用NFA算法抽取字符串：
   1. 将语法树转换为NFA；
   2. 使用NFA simulation算法解析，假如输入的是少量字符；
   3. 实现NFA转换为DFA算法；
2. 直接使用DFA抽取字符串：
   1. 实现算法；
3. 单元测试及自动化测试，性能测试；
4. 参考VCZH及各大神实现；
5. 根据大神实现修改并重构代码；
6. 根据最终版本详述实现算法及架构，并发布博客一篇，代码check in到Github；

## 3.1总体设计概要

总体设计的类图如下：



类FA是所有自动机的基类，它有两个子类，分别是NFA和DFA，其中DFA是NFA的友元，因为从NFA构造DFA需要访问NFA的状态转换表。类RegexExpression用来处理正则表达式，它知道NFA和DFA，并利用它们来实现正则表达式的解析。

## 3.2 正则表达式到语法树

这里首先对输入的正则表达式进行了一个预处理，最主要的操作是两个：

1. 末尾加了一个哨兵操作符‘#’， 用于结束判断；
2. 由于字符串连接是用两个字符紧挨着表示，并没有一个操作符可以查看，所以人为的加了一个’&‘用于表示连接，在下面几种情况下添加&：
   1. ab => a&b;
   2. \*b => \*&b
   3. )b => )&b;
   4. a( => a&(;

进行了基本预处理之后，可以使用双堆栈算法实现解析，伪代码如下：

其中：

1. operands: 操作数堆栈；
2. operators：操作符堆栈；
3. Eval，用于读入一个操作符和操作数堆栈，计算其值；
4. GetPrority，用于获得优先级，对于正则表达式，其优先级如之前所述；

foreach (char c in inputString)

{

if (c is Valid Operand)

{

operands.push(c)

}

else

{

if (c is Valid Operator)

{

if (c is '(')

{

operators.push(c)

}

else if (c is ')')

{

if (operators is empty)

{

fail;

}

tmpOperator = operators.top()

operators.pop();

while (tmpOperator != '('))

{

if (!eval (tmpOperator, operands))

{

fail;

}

if (operator is empty)

{

fail;

}

else

{

tmpOperator = operators.top();

operators.pop()；

}

}

}

else

{

while (operators is not empty && GetPriority(tmpOperator=operators.top()) > GetPriority(c))

{

if (!eval(tmpOperator, operands))

{

fail;

}

operators.pop();

}

operators.push(c);

}

}

else

{

fail;

}

}

}

## 3.3 正则表达式语法树到NFA

算法比较简单也就是Thompson’s construction，具体可见参考文献3 P64。

实现上来说，很直接，就是直接在三个正则表达式操作符里面写转换代码就行了。实现了NFA之后，可以直接用simulateNFA算法对输入字符进行判定，由于会产生很多NULL字符，所以效率不如DFA，但是对于某些只检测一次的字符串来说还是挺有用的，毕竟构造DFA也需要时间。

SimulateNFA算法伪代码 (详细细节见参考文献2 P156)：

// parse a string {strToParse} and return the accept string [0, index) in the string

// if no accept occues, return -1

currentStates.push\_back(startState);

Foreach (char c in the inputString)

{

// NULL closure operation

currentStates=findALLNULLPathStatesFromStates(currentStates);

foreach (State s in currentStates)

{

If (transitionTable has path from s to someStates using c)

{

tmpStates.insert (someStates);

}

}

If (tmpStates.empty())  
{

return i; //c’s index

}

else

{

currentStates = tmpStates;

tmpStates.clear();  
}

}

currentStates = findALLNULLPathStatesFromStates (currentStates);

if (containsEndStates(currentStates))

{

Return strToParse.size();  
}

Else

{

Return -1;  
}

## 3.4 NFA 到 DFA

NFA 到 DFA 主要利用subset Construction算法，具体可见参考文献2的P154：

## 4结论

D

## 5参考文献

1. VCZH, 《词法分析》文档
2. Vlfred V.Aho, Monica S.Lam, Ravi Sethi, Jeffery D.Ullman, Compilers: principle, techiniques & tools
3. Compiler Construction