简单的正则表达式引擎实现

本文介绍了一个简单的正则表达式引擎的实现，总共用了四百五十行左右的代码。

基本的数据结构定义

核心思路是读取正则表达式以后生成对应的NFA，NFA中有边和状态两个结构。边的结构记录了它的起点和终点，同时通过枚举类型记录匹配的其他需求。

```
1.  //用于处理'^'字符
2.  enum { NEXCLUDED = false, EXCLUDED = true };
3.  //用于处理预处理类型, 0-128以内ASCII字符直接匹配
4.  enum { LCASES=256, UCASES=257, NUM=258, EPSILON=259, ANY=260, WS=261 };
5.  class Edge
6.  {
7.  public:
8.      State *start;
9.      State *end;
10.     int type;
11.     int exclude;
12.     Edge(State *s, State *e, int t, bool ex = NEXCLUDED) :start(s), end
    (e), type(t), exclude(ex) {};
```

```
}
```

状态有预备，成功和失败三种，同时每个状态维护两个向量，向量存储了出边和入边的指针。

```
enum { READY = -1, SUCCESS = 1, FAIL = 0};
class State
{
public:
    int status;
    std::list<Edge *> InEdges;
    std::list<Edge *> OutEdges;
}
```

Nfa类会存储一个正则表达式，同时存储NFA的起点和终点，并使用了两个链表来维护NFA的边和状态，同时用一个链表来存储匹配成功的字符串。两个静态的字符串指针用于记录文件和正则表达式字符串的读取状态，静态常量，使得最终函数只会对文件内容和正则表达式扫描一次，避免在匹配成功的字符串中再匹配子串。

```
char *regex;
    State *Start;
    State *End;
    std::list<Edge *> edgeList;
    std::list<State *> stateList;
    std::list<char> matchedChar;
    static char *regRead;
    static char *fileRead;
}
```

生成NFA的过程中，通过**currentEnd**和**currentStart**两个指针分别指向当前字符读取完成后生成的最后一个状态和当前字符读取之前的开始状态，维护这两个指针的目的是为了记录NFA的生成过程，在处理‘*’、‘+’、‘?’等字符的时候起到了重要的作用。同时我们利用list内置的迭代器对链表进行遍历，这个方式在匹配过程中也用到了。

```
State *currentEnd, *currentStart;
State *alternate;
list<Edge *>::iterator itor;
```

NFA的生成

关键的部分在于匹配字符串时采取的思路，尤其是特殊字符的生成NFA的方式，这个不同于课本上最开始的NFA生成算法，而是基于读取字符串的过程，同时避免了字符串的回退等，读取一个字符就生成一个对应的边并压入链表中，对‘*’、‘+’，‘?’和特殊符号也是如此，使得处理更加简单的同时避免生成过于冗余的状态，兼顾了时间和空间效率。以下举例说明。

边和状态的生成

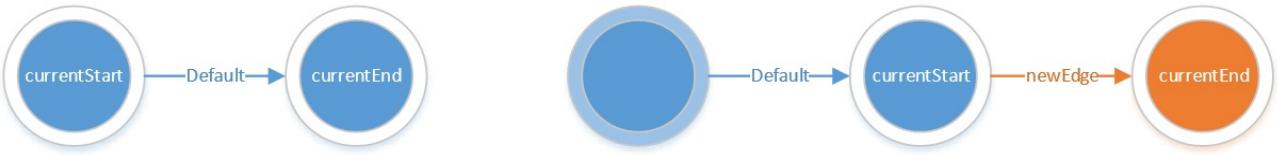
边的生成使用**newEdge**函数,需要记录起点和终点，以及类型，同时在生成边以后要用重载的两个**patch**函数将状态和边完全连接起来。

```
1.  void Nfa::newEdge(State * start, State * end, int type, int exclude = N
    EXCLUDED)
2.  {
3.      Edge *out = new Edge(start, end, type, exclude);
4.      end->patch(out, end);
5.      start->patch(start, out);
6.      edgeList.push_back(out);
7.  }
```

以普通字符的生成和‘.’字符的产生方式为例，他们都是生成一条边和一个新的状态。

```
1.  case '.':   /* any */
2.      currentStart = currentEnd;
3.      currentEnd = new State();
4.      newEdge(currentStart, currentEnd, ANY, NEXCLUDED);
5.      stateList.push_back(currentEnd);
6.  default:
7.      currentStart = currentEnd;
8.      currentEnd = new State();
9.      newEdge(currentStart, currentEnd, *regRead, NEXCLUDED);
10.     stateList.push_back(currentEnd);
11.     break;
```

如下图所示



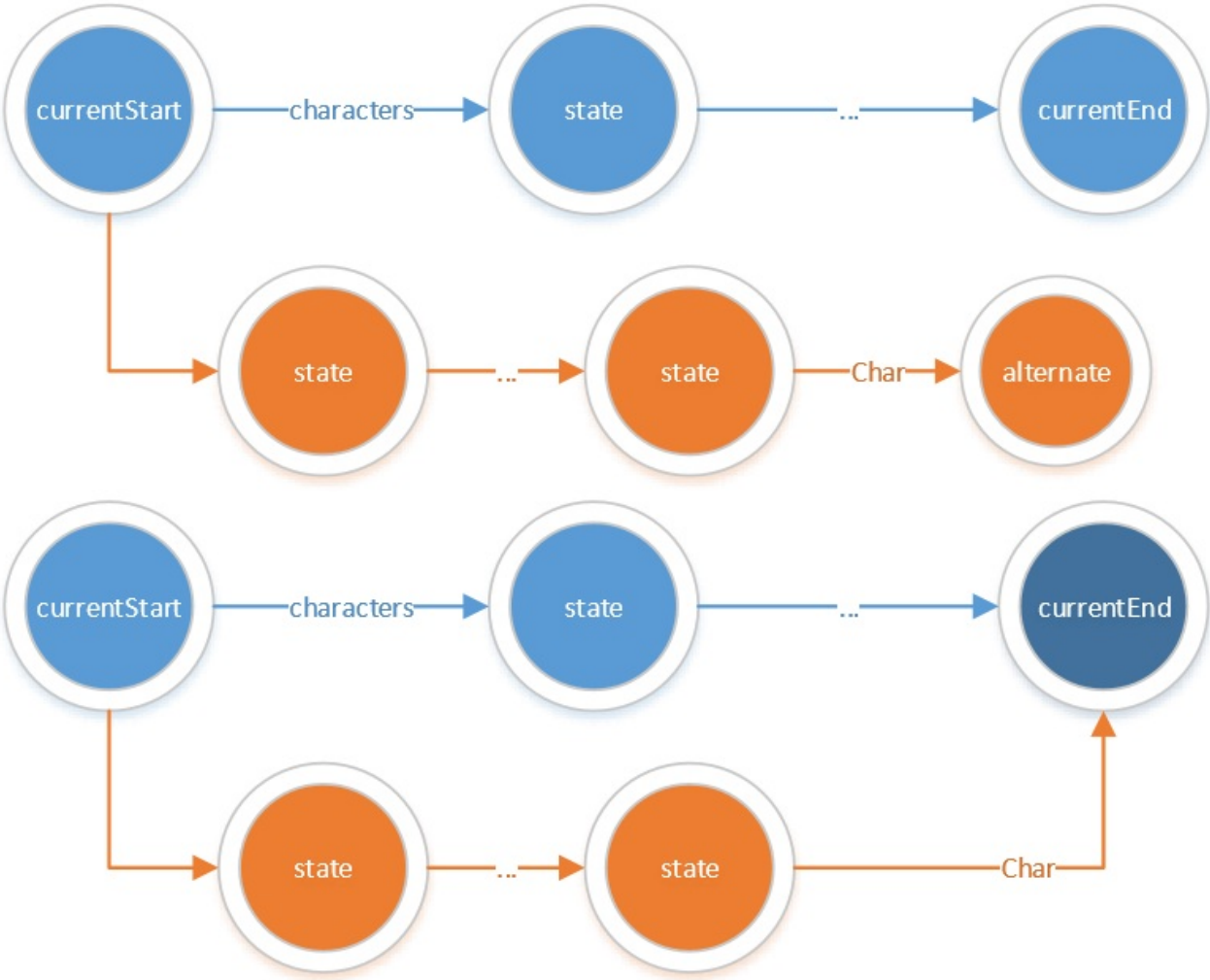
接下来的符号处理都假定初始状态如下图所示

'|'的处理

以**currentStart**指向的状态作为子NFA的起点，同时将子NFA的终点状态和原NFA的终点进行合并。

```
case '|':   // alternate
    regRead++;
    currentStart = start;
    alternate= regex2nfa(regRead, start);
    currentEnd->merge(alternate);
    stateList.remove(alternate);
    regRead--;
```

如下图所示



'?' & '*' & '+'的处理

读取到问号只需要在上一条边的基础上继续连接原有的边即可。

```
case '?':   // zero or one
    newEdge(currentStart, currentEnd, EPSILON, NEXCLUDED);
    break;
```

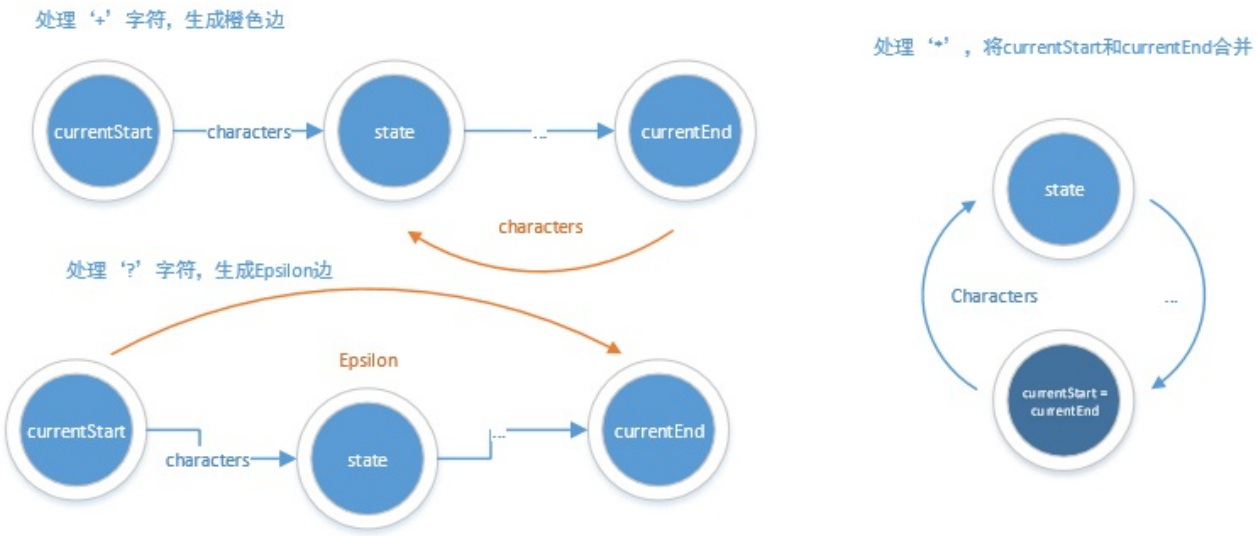
读取到'*'后，直接将**currentStart**和**currentEnd**进行合并成环

```
case '*':   // zero or more
    alternate = currentEnd;
    currentStart->merge(alternate);
    stateList.remove(alternate);
    currentEnd = currentStart;
    break;
```

读取到‘+’后，只需添加若干条边从**currentEnd**状态指向**currentStart**状态的下一个状态即可。

```
case '+':   /* one or more */
    itor = currentStart->OutEdges.begin();
    for (;itor != currentStart->OutEdges.end();itor++)
        newEdge(currentEnd, (*itor)->end, (*itor)->type, (*itor)->exclude);
    break;
```

如下图所示：



简单的分组支持

对于中括号和括号进行了一定的支持，括号直接递归调用NFA的生成函数，中括号和预定义字符都有其对应的函数进行支持。

NFA匹配

匹配过程采用了递归的方式，**step**函数调用**match**函数匹配边和文件字符，匹配成功后即递归调用进入下一个状态。

```
if (End->status == SUCCESS)
        return SUCCESS;

for(;itor != current->OutEdges.end();itor++)
{
    if ((*itor)->match(fileRead))
    {
        (*itor)->end->status = SUCCESS;
        matchedChar.push_back(*fileRead);
        ++fileRead;
        if (step((*itor)->end))
            return SUCCESS;
            --fileRead;
        matchedChar.pop_back();
    }
    if ((*itor)->type == EPSILON && step((*itor)->end))
        return SUCCESS;
}
return FAIL;
```

结果

较好的通过了测试用例，但是没有进一步拓展功能，只是一个简单的正则表达式，同时也有些取巧，都只对字符进行了一次扫描，没有进行完整的词法分析和语法分析，程序在四百五十行左右的情况下，其实是不够健壮的。

```
PS D:\reg> .\test.bat

D:\reg>type  test1.txt
bxz2333@gmail.com
Abcdef6G
aabbccdef
abbcdef
abcd2333
ghabcdefg

D:\reg>.\regEngine abcdef test1.txt
NFA has built successfully!
Matced characters: abcdef
Finished finding matched strings in test1.txt. Successed!

D:\reg>.\regEngine "(abcdef)|abc" test1.txt
NFA has built successfully!
Matced characters: abc
Matced characters: abcdef
Finished finding matched strings in test1.txt. Successed!

D:\reg>.\regEngine a*b+c?cdef test1.txt
NFA has built successfully!
Matced characters: bcdef
Matced characters: aabbccdef
Matced characters: abbcdef
Matced characters: abcdef
Finished finding matched strings in test1.txt. Successed!

D:\reg>.\regEngine abcd([0-9]+) test1.txt
NFA has built successfully!
Matced characters: abcd2
Finished finding matched strings in test1.txt. Successed!

D:\reg>.\regEngine [A-Z]bcd[\w][a-z][0-9]G test1.txt
NFA has built successfully!
Matced characters: Abcdef6G
Finished finding matched strings in test1.txt. Successed!

D:\reg>.\regEngine [\w]+@[a-z]+.com test1.txt
NFA has built successfully!
Matced characters: bxz2333@gmail.com
Finished finding matched strings in test1.txt. Successed!
PS D:\reg>
```