设计文档

正则表达式转NFA

正则表达式转 `NFA` 是通过递归分解复杂问题，化简到可以处理的简单问题，然后回溯。类似于归并排序。

这里我计划的递归终点，也就是可以处理的简单形式的子字符串（ `token` ），是形如 `x*` 或者不含有 `|` 和 `*` 字符的字符串。其它的复杂字符串都可以化为这两种形式解决。**这也就是 `Thompson` 构造法。**

思路步骤：

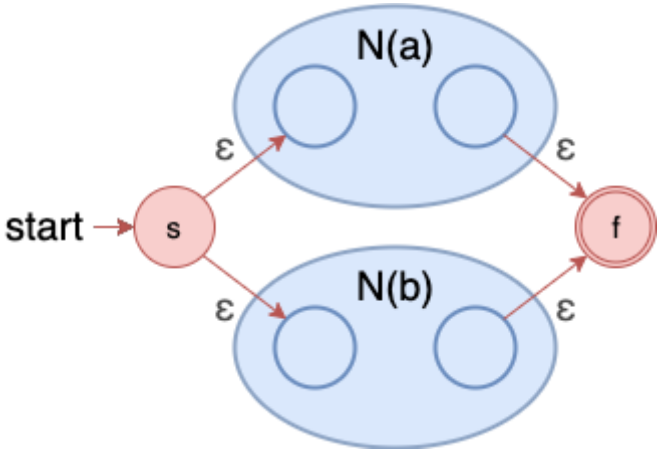
1. 首先对于一个字符串，如果其存在括号外部的 `|` ，也就是形如 `(xxx)|(yyy)` ，那么根据正则表达式的运算规则肯定优先考虑 `|` （不代表 `|` 的优先级最高）。因此专门写了一个函数 `vector<int> getOuterVerticalLinePosition(string token)` 来获取字符串括号外部的 `|` 字符下标。
2. `vector<string> splitToken(string token)` 将字符串分割为更简单的字符串来处理。这里举几个例子，`(xxx)|(yyy)` 分割为 `(xxx)` ， `|` ， `(yyy)` ； `aaax*yyy` 分割为 `aaa` ， `x*` ， `yyy` ； `(aaa)(bb)*yy` 分割为 `(aaa)` ， `(bb)*` ， `yy` 。
3. 在主核心运行函数 `NFAToken innerRun(string input, int state = 1)` 中，先判断递归是否到了可以处理的终点。

```
NFAToken innerRun(string input, int state = 1) {
            // 纯连接(不存在|和*)的Token可以直接解决
            if (input.find('|') == string::npos && input.find('*') == string::npos) {
                                // .....
                                return token
            }

            // x*可以直接解决
            if (input.length() == 2 && input[1] == '*') {
                // ....
                return token;
            }

            // 无法用最简单基本可处理的状态处理就分割为单元表达式，然后依次递归处理，这里的单元表达式可能存在单个字符"|"
            vector<string> tokenValueList = splitToken(input);
            // ...
```

如果没到达，就利用 `splitToken` 将字符串化为更容易处理的子字符串。

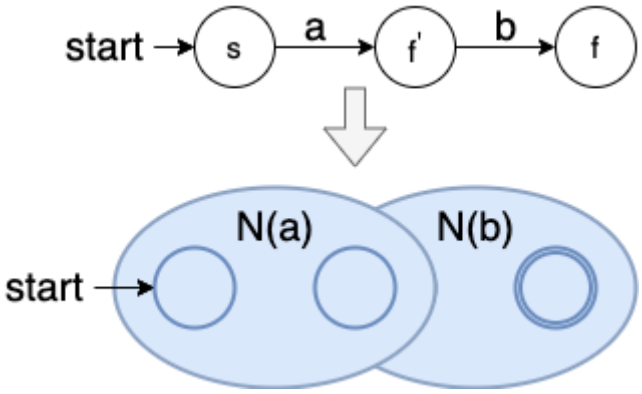
4. 在 `splitToken` 后，先判断有没有括号外部的 `|` ，因为这个需要优先处理。

对于选择 `|` ，递归的处理逻辑如下。我们知道 `i` 节点当前的状态 `state` 下标，然后处理的思路是递归处理 `s` 字符串，然后处理完 `s` 字符串后会返回 `s` 字符串的 `endState` 结束的状态下标，将其加一后作为处理 `t` 字符串起始 `state` 状态下标。因此我们可以知道我们需要用一个变量保存 `i` 节点的状态下标，免得回溯之后不知道 `i` 节点的状态下标为多少，就不能建立 `i` 和 `t` 的连线。

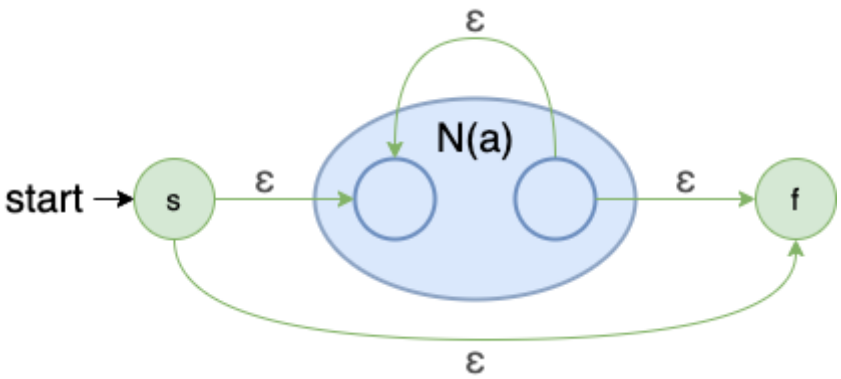


5. 如果不存在 `|` 就说明是连接，但是要注意 `(xxx)*` 。

连接的处理方式如下：



闭包的处理方式如下：



连接好处理，对于闭包的处理方式（例如 `(aaa)*`）就是递归处理 `aaa` ，然后也是要像 `|` 获取子字符串 `aaa` 的 `endState` ，然后再像上图一样连线。

NFA转DFA

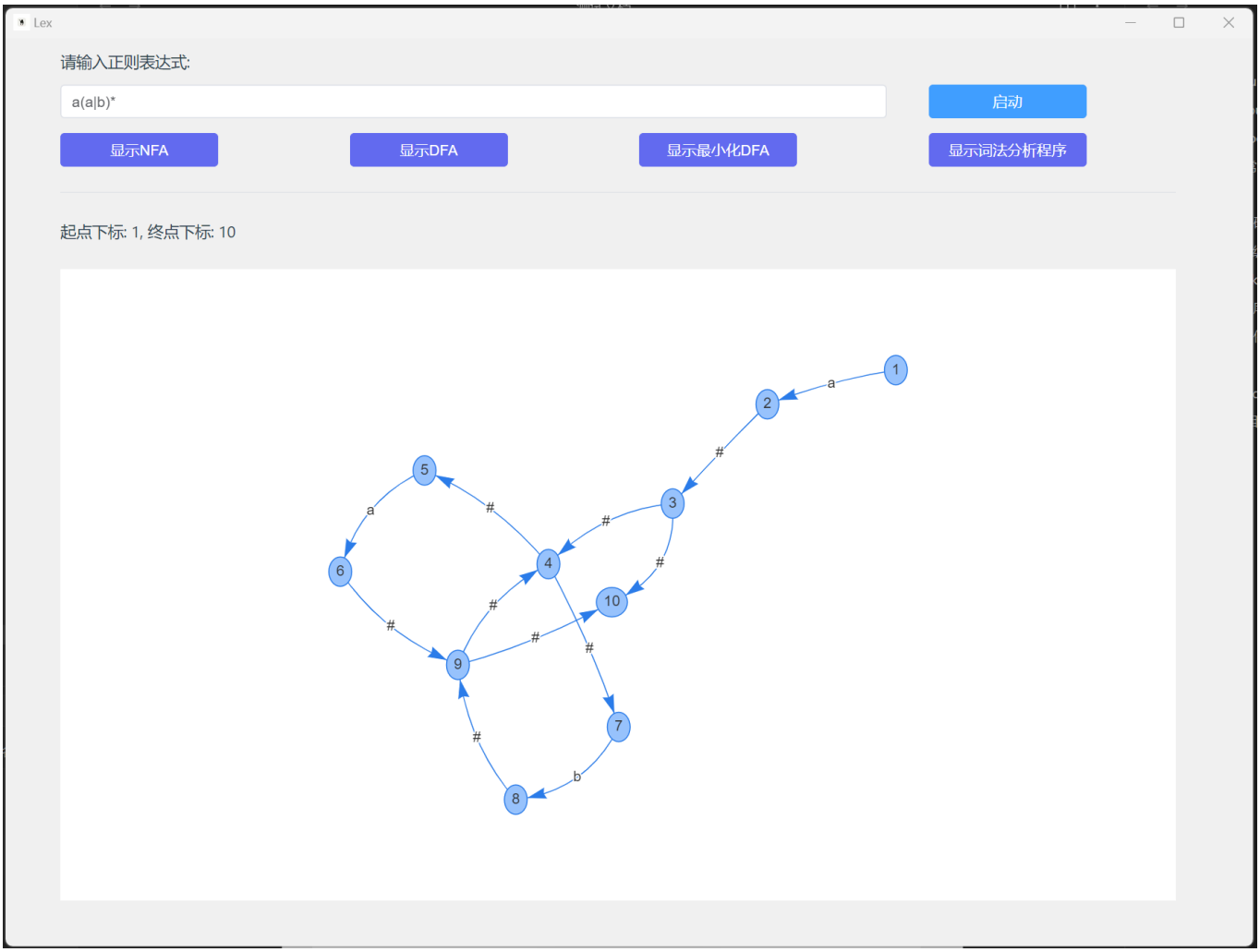
`NFA` 转 `DFA` (使用 **子集构造法**)。

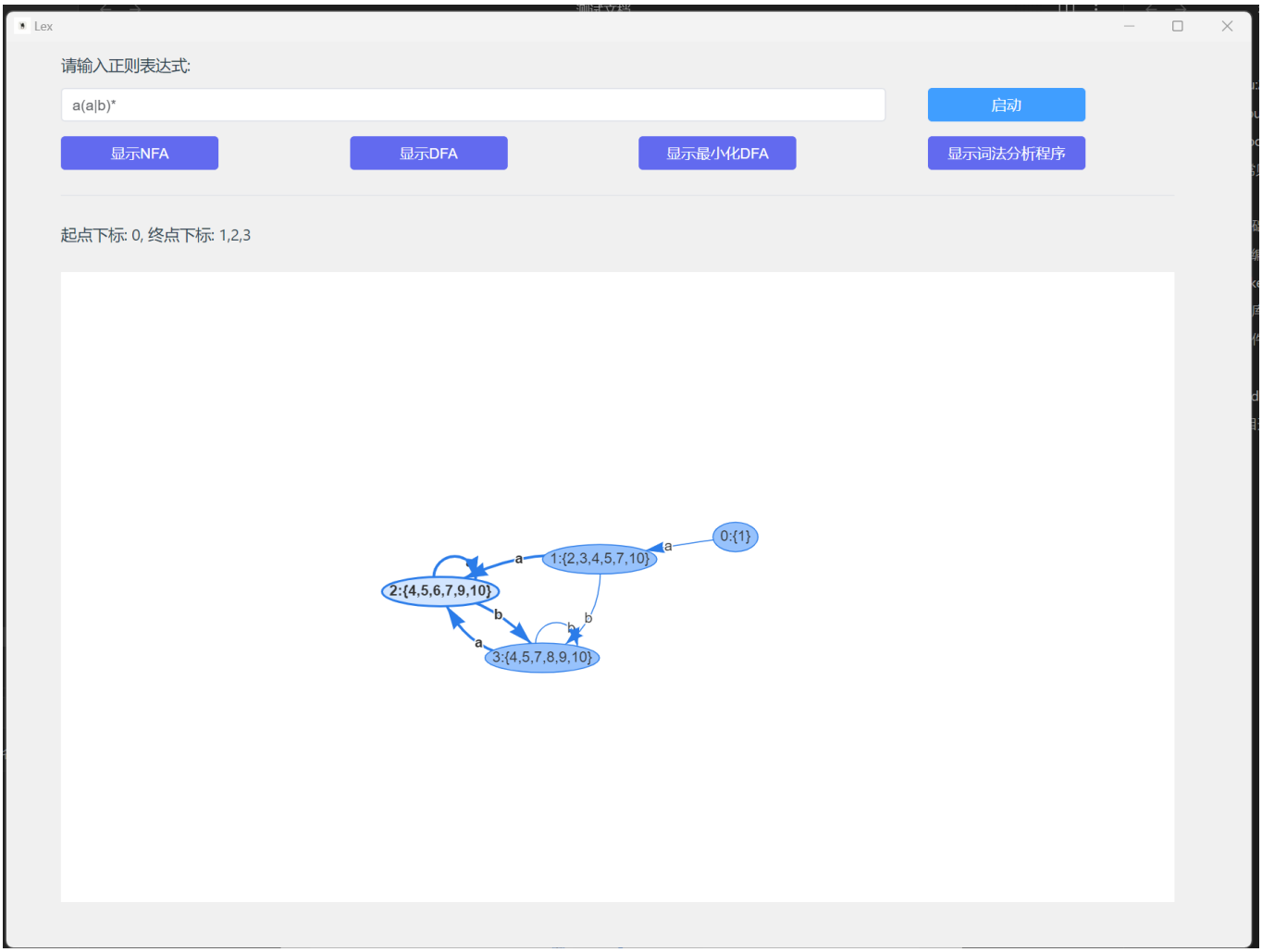
`NFA` 转 `DFA` 就是对一个节点进行 `dfs` （ `bfs` 也可以），遇到后续边为空串就加入那个后续的节点到当前节点的状态集合。这里需要注意的是 `dfs` 在遍历的时候要记录 `visited` 数组，遇到非空串且 `visited == true` 的

节点就不用继续往后 `dfs` 了，不然会死循环。但是也要记录这个非空边。

分析一下下面的数据就清楚了。

测试数据1 `a(a|b)*` ：





DFA最小化

`DFA` 最小化使用 `Hopcroft` 算法。

1. 首先把整个状态划分为接收状态和非接受状态，方法是：包含了 `NFA` 终态节点的状态集就一定是 `DFA` 的非接收状态。
2. 遍历某个状态集中存在的字符，看当前状态集的每个节点接收到这个字符跳转到的下一个状态是否相同，如果相同就继续判断下一个字符，直到所有字符都判断成功就可以成功插入到结果。

- 如果存在不同，就把当前状态集的节点根据当前字符到的目标状态来划分，划分后的结果重新入栈处理。
- 要注意等价状态，如果两个状态 `s`，`t`，对于 **所有输入符号**，状态 `s` 和状态 `t` 都转换到等价的状态里，那么这两个状态就是等价的。
- 可区分：对于任何两个状态 `t` 和 `s`，若从一状态出发接受输入字符串 `ω`，而从另一状态出发不接受 `ω`，或者从 `t` 出发和从 `s` 出发到达不同的接受状态，则称 `ω` 对状态 `t` 和 `s` 是可区分的。
- 不可区分：设想任何输入序列 `ω` 对 `s` 和 `t` 均是不可区分的，则说明从 `s` 出发和从 `t` 出发，分析任何输入序列 `ω` 均得到相同结果。因此，`s` 和 `t` 可以合并成一个状态。

```
while (!workStack.empty()) { // 重复判断
        set<int> currentPartition = workStack.top();
        workStack.pop();
        // ....
```

```
        for (char ch = ' '; ch <= '~'; ++ch) { // 遍历可能字符
                map<int, set<int>> partitionMap; // 记录对于当前字符，每个节点经过该字符对应的下一个点属于哪个状态集，map<[状态集下标], [经过该字符对应的下一个点属于该状态集的节点下标]>
                for (int i: currentPartition) { // 遍历当前状态集的节点
                        获取当前节点通过ch字符到达的下一状态toState
                        如果当前节点的后续边的权值没有ch字符，就当作下一状态为-1
                        将i插入到partitionMap[ch]对应的set<int>中
                }
                判断partitionMap的长度，如果(大于2)||(等于1 && partitionMap[0].size()=0)就说明当前状态集可再分
                接着就把上面通过partitionMap再次分割的字状态集入workStack继续判断
                如果当前状态集经过了所有字符的判断依旧不可再分，就说明真的不可区分，可以记录到结果中了
        }
}
```

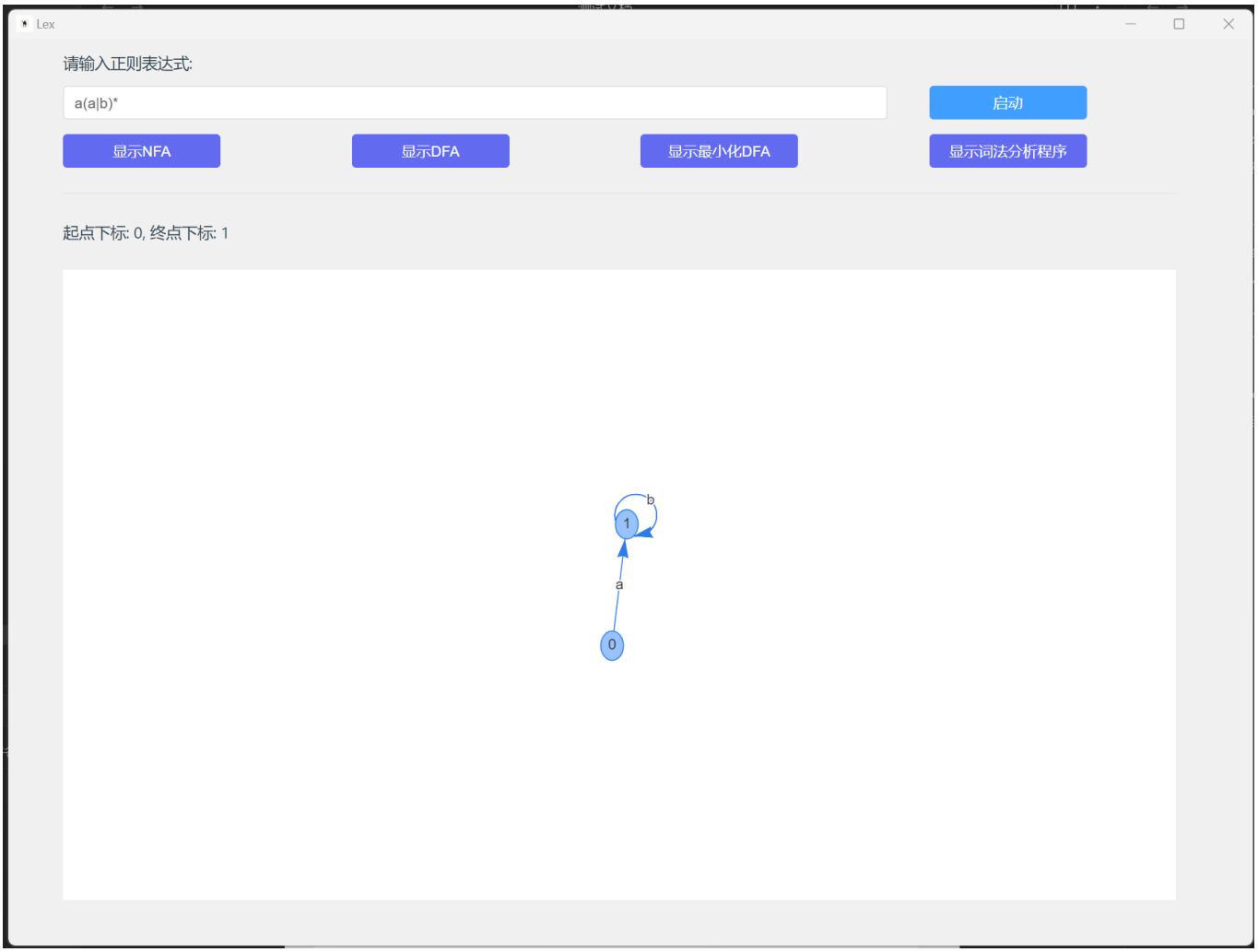
DFA转词法分析程序

这个步骤就稍微容易些了，关键是理解什么是词法分析程序。词法分析程序就是要生成一个可以判断某个输入字符串是否满足某个正则表达式的程序。我们根据前面生成的最小化 `DFA` ，确定起始状态和终止状态，当前遍历输入字符串结束的时候，如果字符串到达了终止状态，就说明某个正则表达式可以接受这个输入的字符串。如果在遍历字符串的途中遇到了字符串不可接受的状态，就直接可以说明某个正则表达式不能接受这个字符串了。

下面直接给个示例。

测试数据1 `a(a|b)*` ：

- 最小化 `DFA` 的拓扑图：



- 生成的词法分析程序：

```
#include <iostream>
#include <string>
using namespace std;

bool isAccepted(string str) {
    int currentState = 0;
    int endState = 1;
    for (int i = 0; i < str.size(); ++i) {
        switch (currentState) {
            case 0:
                if (str[i] == 'a') {
                    currentState = 1;
                }
                else {
                    return false;
                }
                break;
            case 1:
                if (str[i] == 'a') {
                    currentState = 1;
                }
```

```
                else if (str[i] == 'b') {
                    currentState = 1;
                }
                else {
                    return false;
                }
                break;
            default:
                return false;
        }
    }
    if (currentState == endState) {
        return true;
    }
    return false;
}

int main() {
    string str;
    cin >> str;
    if (isAccepted(str)) {
        cout << "accepted" << endl;
    }
    else {
        cout << "can not accepted" << endl;
    }
    return 0;
}
```

参考文章

```
# 正则表达式转NFA
https://blog.csdn.net/weixin_44691608/article/details/110195743
https://www.tr0y.wang/2021/04/01/%E7%BC%96%E8%AF%91%E5%8E%9F%E7%90%86%EF%BC%88%E4%B8%80%EF%BC%89%EF%BC%9A%E8%AF%8D%E6%B3%95%E5%88%86%E6%9E%90/#nfa-dfa
```