- 开始之前，已经将正则表达式的 `[]` 转为或运算，`+`正闭包已转为：`aa*`运算
- 正则表达式转后缀表达式的算法：

```
set<string>regName;
// 5.将正则表达式转为后缀表达式:保证为：(xxx)yyy?.*
// ！！！不支持的命名规则：_或含有数字的命名串作为正则表达式的一部分。比如：_name123 = _digit200S | a* , (_digit200s)不支持
string latExp(string regex) {
    stack<char> op; //运算符栈
    string latAns; //存储后缀表达式
    string currentStr; //存储命名字符串

    //优先级函数
    auto priv = [](char ch) {
        if(ch == '*' || ch == '?') return 3;
        else if(ch == '&') return 2; //连字符
        else if(ch == '|') return 1;
        else return 0; //'(' 或 ')'
    }

    //开始转为后缀表达式
    for(int i = 0; i < regex.size(); i++) {
        if(isspace[regex[i]]) continue;

        //检查是否为转义字符
        if(ch=='\\') {
            latAns += '\\';
            latAns += regex[++i];
            continue;
        }

        //左括号，入栈
        if(regex[i] == '(')
            op.push(regex[i]);
        //右括号，出栈
        else if(regex[i] == ')') {
            while(!op.empty() && op.top() != '(') {
                latAns += op.top();op.pop();
            }
            op.pop(); //左括号弹出
        }
        //普通字符,是否为命名规则
        else if(isalpha(ch)) {
            currentStr.clear();
            while(i < regex.size() && isalpha(regex[i])) {
                currentStr += regex[i++];
            }
            i--;

            if(regName.find(currentStr) != regName.end()) {
                latAns += " " + currentStr + " "; //前后加上空格
```

```
            }
            else {
                for(char c: currentStr)
                    latAns += c;
            }
        }
    }

    //剩余未出栈的运算符依次出栈
    while(!op.empty()) {
        latAns += op.top();op.pop();
    }
}
```

- 后缀表达式转NFA算法：

```
//NFA节点结构体
struct NFANode {
    int id = -1;    //NFA节点编号
    string c = "#"; //NFA节点变换弧的值,可能是命名规则
    int to = -1;    //NFA节点通过该弧转到的状态号
    set<int> same;  //等价类，通过ϵ转换
};
//NFA结构体
struct NFA {
    NFANode * h = 0;//NFA的头指针
    NFANode * t = 0;//NFA的尾指针
};
stack<NFA> NFAStack; //NFA栈,用来进行后缀表达式的使用eg:aab|*
string latReg; //已经计算好结果的后缀表达式aab|*或:a digit b|*
int NNum = 0; //节点数

//从状态n1到状态n2添加一个弧边，值为st,默认为ϵ，为ϵ时添加等价类
void add(NFANode* n1, NFANode* n2, string ch="#") {
    if(ch == "#") {
        n1.same.insert(n2->id);
    }
    else {
        n1->c = ch;
        n1->to = n2->id;
    }
}

void regToNFA() {

    int i = 0;
    while(i < latReg.length()) {
        //是命名规则，命名规则与其他之间，前后用空格分割
        if(isspace(latReg[i])) {
            i++;
            string name = "";
            while(!isspace(latReg[i]))
                name += latReg[i++];
            i++;
            NFA n = createNFA(NNum);//建立新的NFA节点(包含指针new)，NNum+2
```

```
            add(n.h, n.t, name);     //头到尾的name连接。o ->(name) o
            NfaStack.push(n); //建立好节点就入栈
            continue;
        }
        //最后再处理普通字符。（即都不在这些里面的肯定都是ok的）
        //闭包
        else if(latReg[i] == '*') {
            NFA n = createNFA(NNum); //+2,是ϵ
            NFA n1 = NfaStack.top();NfaStack.pop();
            add(n.h, n.t); //o ->(ϵ) o
            add(n.h, n1.h); //都是等价节点，通过ϵ转换
            add(n1.t, n.t);
            add(n1.t, n1.h); //自环
            NfaStack.push(n); //新的大闭包。
        }
        //或
        else if(latReg[i] == '|') {
            NFA n = createNFA(NNum);
            NFA n1 = NfaStack.top();NfaStack.pop();
            NFA n2 = NfaStack.top();NfaStack.pop();
            add(n.h, n1.h); //NFA这个结构是:o -> o，h是前面o，t是后面o
            add(n.h, n2.h);
            add(n1.t, n.t);
            add(n2.t, n.t);
            NfaStack.push(n);
        }
        //与
        else if(latReg[i] == '&') {
            NFA n;
            //注意，压入栈后的顺序是相反的。比如：ab&压入栈后：栈顶:&ba，我们最开始取出来的是b，是后者。因为要:o->a->o->b->o
            NFA n2 = NfaStack.top();NfaStack.pop();
            NFA n1 = NfaStack.top();NfaStack.pop();
            add(n1.t, n2.h);
            //连接整体
            n.h = n1.h;
            n.t = h2.t;
            NfaStack.push(n);
        }
        //可选
        else if(latReg[i] == '?') {
            NFA n = createNFA(NNum);
            NFA n1 = NfaStack.top();NfaStack.pop();
            add(n.h, n1.h);
            add(n.h,  n.t);
            add(n1.t, n.t);
        }
        //剩下的都是普通字符
        else {
            string st = "";
            if(latReg[i] == '\\') {
                i++;
                st += "\\";
            }
            st += latReg[i];
```

```
            NFA n = createNFA(NNum);
            add(n.h, n.t, st); //o ->(st) o
            NfaStack.push(n);
        }
        i++;
    }
    return NfaStack.top();//最终的最大节点。
}
```

- NFA转DFA算法：

```
// NFA节点结构体
struct NFANode {
    int id = -1;      // NFA节点编号
    string c = "#"; // NFA节点变换弧的值，可能是命名规则
    int to = -1;      // NFA节点通过该弧转到的状态号
    set<int> same;    // ε转换的等价类
};

// NFA结构体
struct NFA {
    NFANode* h = nullptr; // NFA的头指针
    NFANode* t = nullptr; // NFA的尾指针
};

// DFA 转移边结构体
struct DfaEdge {
    string c = "#"; // DFA状态弧值，默认为 epsilon
    int to = -1;      // 转移的目标状态
};

// DFA 节点结构体
struct DfaNode {
    int id = -1;               // DFA节点状态号，-1为无效
    vector<DfaEdge> edges;     // DFA节点的转换弧列表
    set<int> closure;          // NFA的ε闭包
    bool isEnd = false;        // 是否为终态
};

// DFA结构体
struct DFA {
    int start = 0;                // DFA初态
    set<int> end;                 // DFA终态集合
    set<string> symbol;           // DFA的转移字符集
    map<int, vector<DfaEdge>> to; // DFA的转移映射，键为DFA状态，值为其转换边,构建转换表！！！
};

vector<NFANode> NFANodes; // 存储所有NFA节点
vector<DfaNode> DNS(300); // DFA节点数组，支持最多300个节点
int DNSNUM = 0;           // DFA节点总个数
DFA d;                    // DFA实例

// 计算ε闭包
// 函数：计算状态集的ε闭包
```

```
// 参数: 输入状态集s
// 返回: 计算得到的ε闭包状态集
set<int> epClosure(const set<int>& s) {
    stack<int> epStack;        // 存放状态集所有状态的栈
    set<int> result = s;       // 初始化结果集为输入状态集

    // 将目前状态集的所有元素先压入栈
    for (const auto& it : s) {
        epStack.push(it);
    }

    // 遍历状态集合通过ε能够到达的所有状态
    while (!epStack.empty()) {
        int temp = epStack.top(); epStack.pop();

        // 遍历temp能到达的所有状态
        for (const auto& it : NFANodes[temp].same) {
            if (!result.count(it)) { // 如果该状态不在集合中，加入集合
                result.insert(it);
                epStack.push(it);
            }
        }
    }
    return result;
}

// 计算通过某个符号的ε闭包
set<int> move(const set<int>& s, const string& symbol) {
    set<int> result;

    // 遍历状态集 s 中的所有状态
    for (const auto& state : s) {
        // 检查该状态的出边是否匹配给定符号
        if (NFANodes[state].c == symbol) {
            // 如果匹配，则将目标状态加入结果集
            result.insert(NFANodes[state].to);
        }
    }

    // 计算结果集的ε闭包
    return epClosure(result);
}


// 判断当前闭包是否包含终态
bool isEndState(const set<int>& closure, const set<int>& endStates) {
    for (const auto& state : closure) {
        if (endStates.count(state)) {
            return true;
        }
    }
    return false;
}

// NFA转DFA
```

```
void NFAtoDFA() {
    // 初始化DFA的初态
    DNS[0].closure = epClosure({d.start}); // 初态的ϵ闭包
    DNS[0].id = 0;
    DNS[0].isEnd = isEndState(DNS[0].closure, d.end);
    DNSNUM++;

    set<set<int>> states; // 存储已处理的闭包状态集
    states.insert(DNS[0].closure);

    stack<int> s;
    s.push(0); // 将初态压入栈

    while (!s.empty()) {
        int current = s.top(); s.pop();

        // 遍历DFA的转移字符集
        for (const auto& sy : d.symbol) {
            // 计算通过字符sy到达的闭包
            set<int> temp = move(DNS[current].closure, sy);

            if (!temp.empty() && states.find(temp) == states.end()) {
                // 新状态，未处理过
                DNS[DNSNUM].closure = temp;
                DNS[DNSNUM].id = DNSNUM;
                DNS[DNSNUM].isEnd = isEndState(temp, d.end);

                // 添加新状态到DFA
                states.insert(temp);
                s.push(DNSNUM);

                // 添加当前状态的转换边
                DNS[current].edges.push_back({sy, DNSNUM});

                DNSNUM++;
            } else {
                // 如果状态已存在，找到对应的目标ID
                for (int i = 0; i < DNSNUM; ++i) {
                    if (DNS[i].closure == temp) {
                        DNS[current].edges.push_back({sy, i});
                        break;
                    }
                }
            }
        }
    }

    // 构建DFA转移映射
    for (int i = 0; i < DNSNUM; ++i) {
        for (const auto& edge : DNS[i].edges) {
            d.to[DNS[i].id].push_back(edge);
        }
    }
}
```

- DFA最小化算法：

```
// DFA最小化节点结构体
struct Node {
    int id;        // 节点编号
    bool isEnd;    // 是否为终态
};

// DFA最小化转移边结构体
struct Edge {
    int to;        // 转移目标状态
    string c;      // 转移字符
};

// 字符集
vector<string> symbol; // 字符集

// DFA的节点集和转移边集
vector<Node> V;                  // DFA节点集
vector<vector<Edge>> edges;      // DFA转移边集，edges[idx]表示DFA的idx状态存储的所有状态转移

// 等价状态集合和辅助数组
vector<set<int>> eq;  // 等价状态集合
vector<int> p;        // 状态所属集合编号，即给DFA状态划分类别编号
int stateNum = 2;     // 初始分为两类：终态和非终态

// 分裂状态集合
void split(int x, const string& sy) {
    set<int> currentSet = eq[x]; // 获取当前需要分裂的等价类集合 eq[x]

    if(currentSet.size() == 1) return ; //大小为1，不需要再分裂

    map<int, set<int>> transitions; // 用于记录状态转移的分组结果，key 是目标集合编号，value 是状态集合

    // 遍历当前等价集合中的所有状态
    for (int state : currentSet) {
        bool found = false; // 标志位，记录当前状态是否找到匹配的转移

        // 遍历当前状态的所有出边，检查是否存在以 sy 为条件的转移
        for (const auto& edge : edges[state]) {
            if (edge.c == sy) { // 如果转移字符和当前字符 sy 匹配
                // 按目标集合编号（p[edge.to]）对当前状态进行分组，p[edge.to]这里就已经将该节点通过其对应边的到达状态属于那个集合在hop主函数操作了。
                transitions[p[edge.to]].insert(state);
                found = true; // 找到匹配的转移
            }
        }

        if (!found) {
            // 如果没有找到任何匹配的转移
            transitions[-1].insert(state); // 将当前状态归入特殊分组 -1 表示没有匹配的转移
        }
```

```
    }

    // 遍历 transitions 中的每个分组，检查是否需要分裂集合
    for (auto& [groupId, newSet] : transitions) {
        // 如果新的分组 newSet 的大小小于当前集合 currentSet 的大小

        // 说明当前集合需要分裂（通过符号 sy 的转移行为不同）
        // 每一次分裂出去一个集合，那么currentSet就会变小。直到不断分裂后，下一个
newSet的值和currentSet的值一样大的时候，说明已经分裂完毕。
        if (newSet.size() < currentSet.size()) {
            eq.push_back(newSet); // 将新的分组 newSet 添加到等价集合 eq

            // 更新分组中每个状态的集合编号
            for (int state : newSet) {
                p[state] = stateNum; // 为这些状态分配新的集合编号 stateNum
                currentSet.erase(state); // 将这些状态从原集合 currentSet 中移除
            }

            stateNum++; // 增加集合编号，用于表示新的分组
        }
    }

    eq[x] = currentSet; // 更新原集合 eq[x]，将其设置为剩余未分裂的状态部分
}


// DFA最小化算法（Hopcroft）
void Hopcroft() {
    // 初始化终态和非终态集合
    set<int> endStates, nonEndStates;
    //开始遍历每一个DFA节点，将其分为两组。
    for (const auto& node : V) {
        if (node.isEnd) {
            endStates.insert(node.id);
            p[node.id] = 1;
        } else {
            nonEndStates.insert(node.id);
            p[node.id] = 0;
        }
    }

    //划分等价类为两类
    if (!nonEndStates.empty()) eq.push_back(nonEndStates);
    if (!endStates.empty()) eq.push_back(endStates);

    // 主循环：逐步分裂集合
    while (true) {
        int previousStateNum = stateNum; //先计算旧的集合数目
        // 遍历字符集，通过该不同字符集，每个状态是否到达相同的集合。
        for (const auto& c : symbol) {
            //一开始只有旧的集合数目。一个个等价类集合内部去分裂。
            for (int i = 0; i < previousStateNum; ++i) {
                split(i, c); // 尝试分裂每个集合
            }
        }
```

```
            if (stateNum == previousStateNum) break; // 集合不再变化，结束
        }
    }
```