第一章

1、**区别**：编译执行方式有两个独立的处理环节：编译和执行

解释器执行方式只有一个处理环节

编译执行的特点：是源程序被等价翻译成目标程序，输出可执行文件，可执行文件再在运行环境下独立运行。

解释执行的特点：源程序是解释器的输入，解释器的输出是源程序的执行结果。

用高级程序语言写的源程序通常用编译方式执行。

用脚本语言写的源程序通常用解释执行方式执行。

2、违背了语义法则，(2)(3)两行代码重复对变量 c 进行定义道之语义模糊，是语义规则的范畴。

3、经过本地运行代码测试

在前三种函数定义中，其中任意两种定义会导致函数重定义错误。因为这些定义之间的区别不足以让编译器区分它们对应的函数，它们都具有相同的函数签名，即函数名为 fun，参数类型为 classA 或其引用或其常量引用，返回类型为 int。

前 3 种的任何一个和（4）在一起编译都不会出错，因为第四种定义的参数类型为指向 classA 类型的指针，与前三种定义的参数类型不同，因此编译器可以做出区分。

4、第（6）行语句“i=6”是对第（2）行的变量 i 重新赋值为 6

第（7）行的 j 指代离它最近的第（5）行的“j=7“

所以第（7）行后的“i=i+j”，得到 i 的值为 6+7=13；

第（9）行的 j 指代代码块外的 j，即第（3）行 j=5；

所以 w 的值为 i+j，即 13+5 等于 18

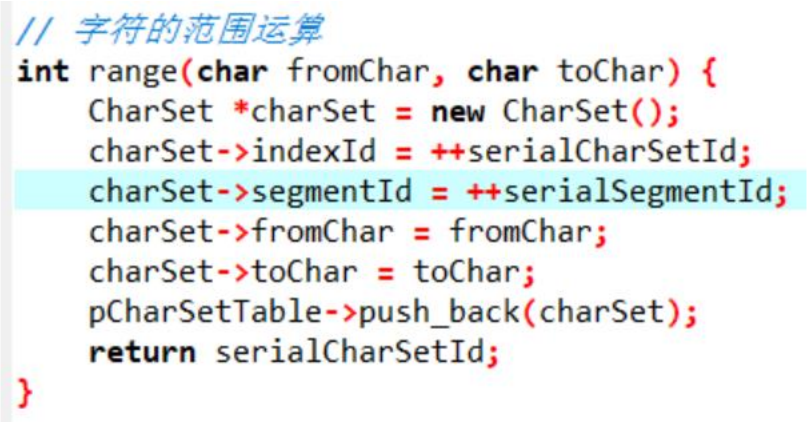
第 10-13 行的变量 i 是代码块里的，对代码块外的 i 值不产生影响

所以 z 的值为 i+j，也为 18

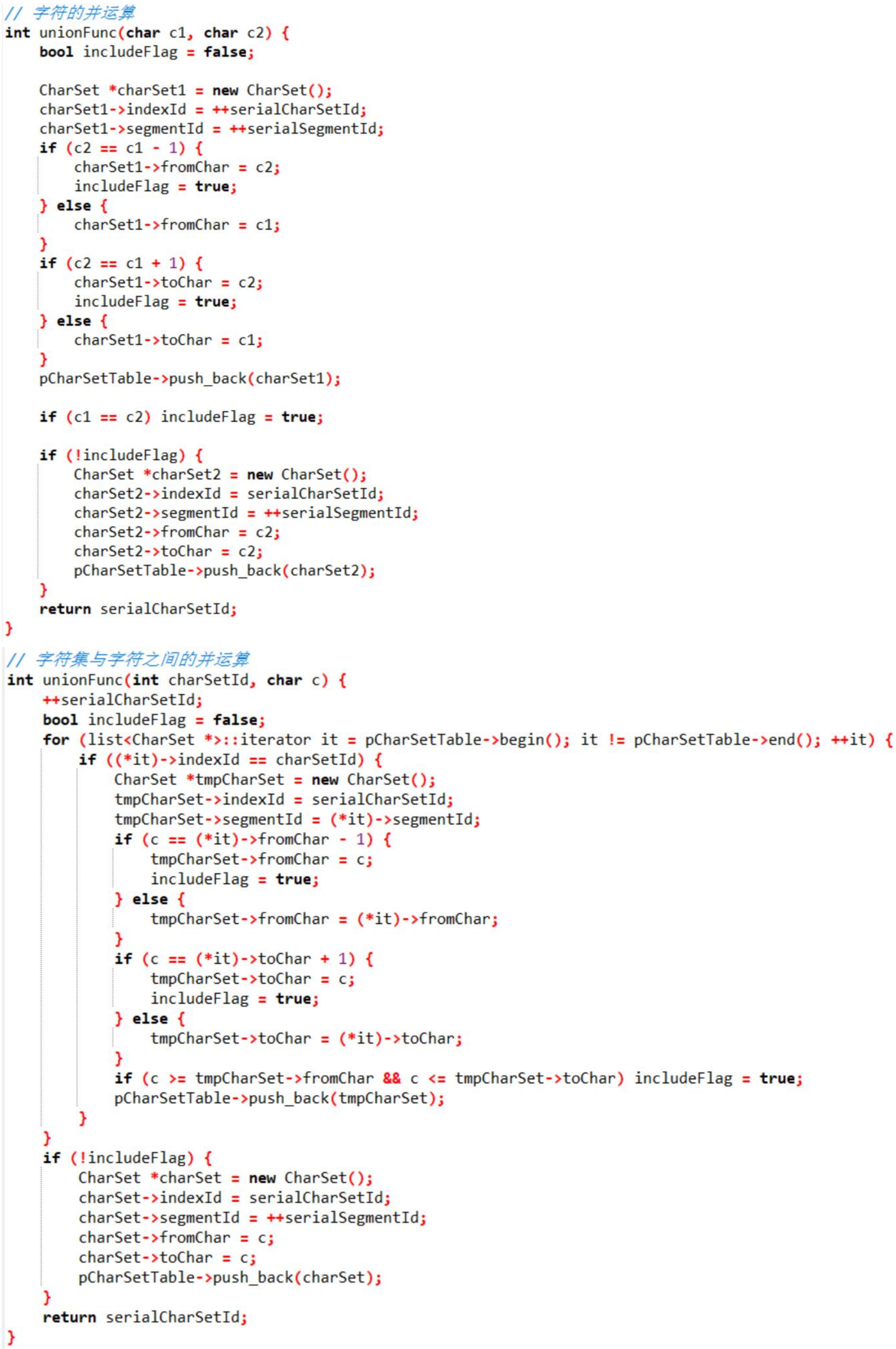
综上 w 和 z 的值均为 18。

第二章

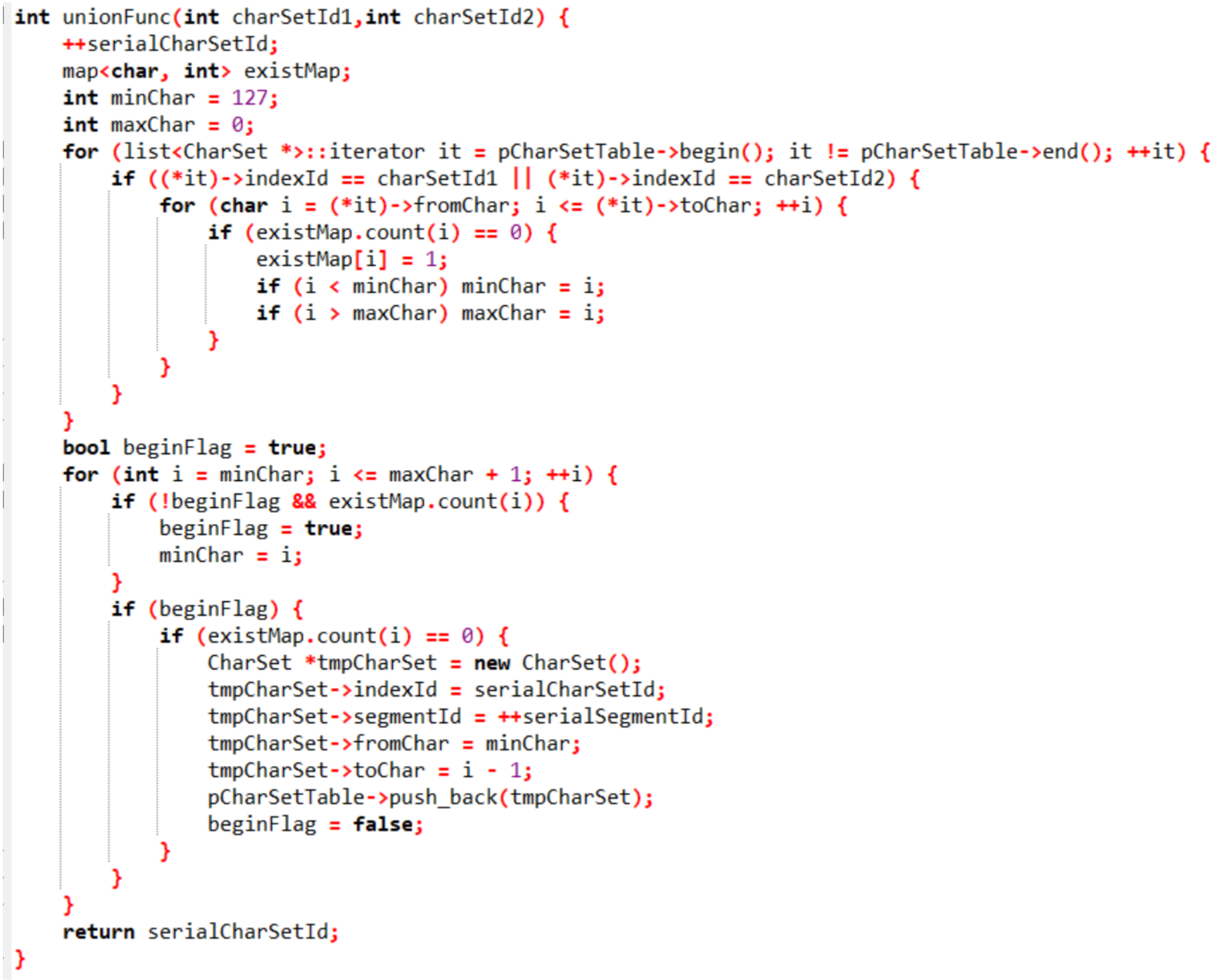
1、



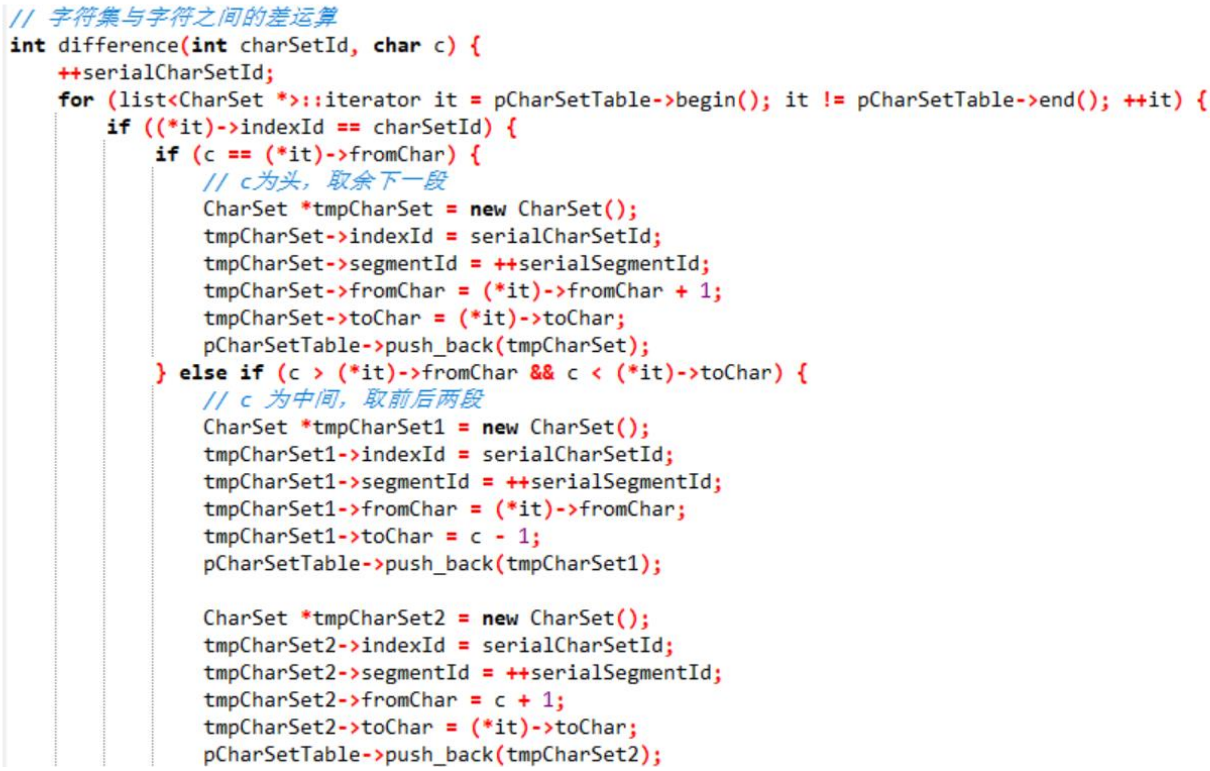
```
// 字符的范围运算
int range(char fromChar, char toChar) {
    CharSet *charSet = new CharSet();
    charSet->indexId = ++serialCharSetId;
    charSet->segmentId = ++serialSegmentId;
    charSet->fromChar = fromChar;
    charSet->toChar = toChar;
    pCharSetTable->push_back(charSet);
    return serialCharSetId;
}
```



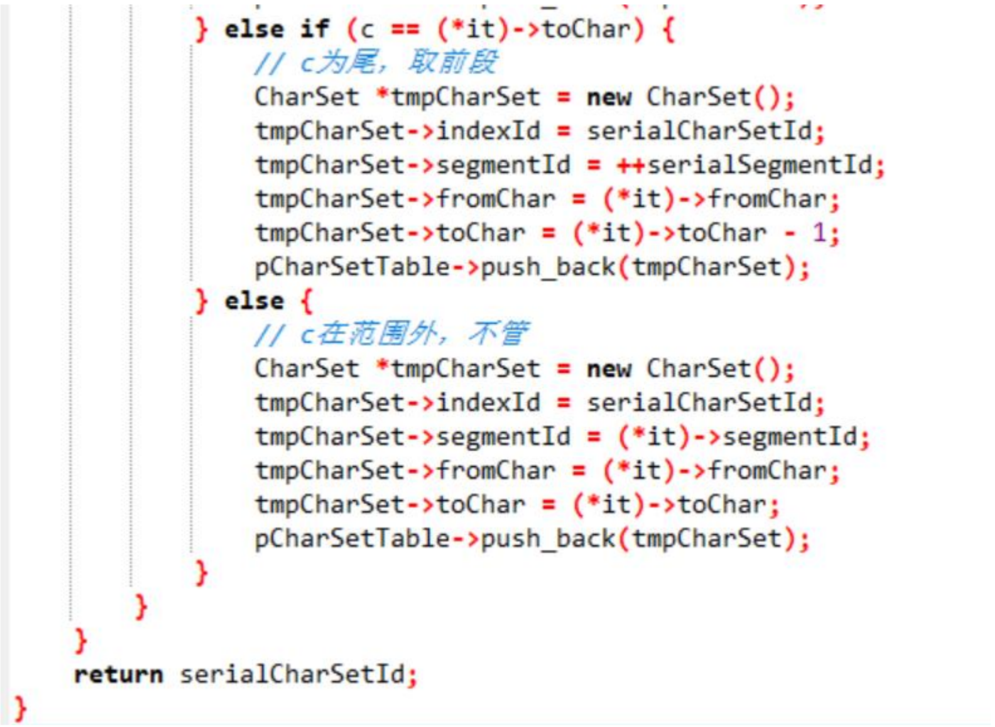
```
// 字符的并运算
int unionFunc(char c1, char c2) {
    bool includeFlag = false;

    CharSet *charSet1 = new CharSet();
    charSet1->indexId = ++serialCharSetId;
    charSet1->segmentId = ++serialSegmentId;
    if (c2 == c1 - 1) {
        charSet1->fromChar = c2;
        includeFlag = true;
    } else {
        charSet1->fromChar = c1;
    }
    if (c2 == c1 + 1) {
        charSet1->toChar = c2;
        includeFlag = true;
    } else {
        charSet1->toChar = c1;
    }
    pCharSetTable->push_back(charSet1);

    if (c1 == c2) includeFlag = true;

    if (!includeFlag) {
        CharSet *charSet2 = new CharSet();
        charSet2->indexId = serialCharSetId;
        charSet2->segmentId = ++serialSegmentId;
        charSet2->fromChar = c2;
        charSet2->toChar = c2;
        pCharSetTable->push_back(charSet2);
    }
    return serialCharSetId;
}
// 字符集与字符之间的并运算
int unionFunc(int charSetId, char c) {
    ++serialCharSetId;
    bool includeFlag = false;
    for (list<CharSet *>::iterator it = pCharSetTable->begin(); it != pCharSetTable->end(); ++it) {
        if ((*it)->indexId == charSetId) {
            CharSet *tmpCharSet = new CharSet();
            tmpCharSet->indexId = serialCharSetId;
            tmpCharSet->segmentId = (*it)->segmentId;
            if (c == (*it)->fromChar - 1) {
                tmpCharSet->fromChar = c;
                includeFlag = true;
            } else {
                tmpCharSet->fromChar = (*it)->fromChar;
            }
            if (c == (*it)->toChar + 1) {
                tmpCharSet->toChar = c;
                includeFlag = true;
            } else {
                tmpCharSet->toChar = (*it)->toChar;
            }
            if (c >= tmpCharSet->fromChar && c <= tmpCharSet->toChar) includeFlag = true;
            pCharSetTable->push_back(tmpCharSet);
        }
    }
    if (!includeFlag) {
        CharSet *charSet = new CharSet();
        charSet->indexId = serialCharSetId;
        charSet->segmentId = ++serialSegmentId;
        charSet->fromChar = c;
        charSet->toChar = c;
        pCharSetTable->push_back(charSet);
    }
    return serialCharSetId;
}
```



```
int unionFunc(int charSetId1,int charSetId2) {
    ++serialCharSetId;
    map<char, int> existMap;
    int minChar = 127;
    int maxChar = 0;
    for (list<CharSet *>::iterator it = pCharSetTable->begin(); it != pCharSetTable->end(); ++it) {
        if ((*it)->indexId == charSetId1 || (*it)->indexId == charSetId2) {
            for (char i = (*it)->fromChar; i <= (*it)->toChar; ++i) {
                if (existMap.count(i) == 0) {
                    existMap[i] = 1;
                    if (i < minChar) minChar = i;
                    if (i > maxChar) maxChar = i;
                }
            }
        }
    }
    bool beginFlag = true;
    for (int i = minChar; i <= maxChar + 1; ++i) {
        if (!beginFlag && existMap.count(i)) {
            beginFlag = true;
            minChar = i;
        }
        if (beginFlag) {
            if (existMap.count(i) == 0) {
                CharSet *tmpCharSet = new CharSet();
                tmpCharSet->indexId = serialCharSetId;
                tmpCharSet->segmentId = ++serialSegmentId;
                tmpCharSet->fromChar = minChar;
                tmpCharSet->toChar = i - 1;
                pCharSetTable->push_back(tmpCharSet);
                beginFlag = false;
            }
        }
    }
    return serialCharSetId;
}
```

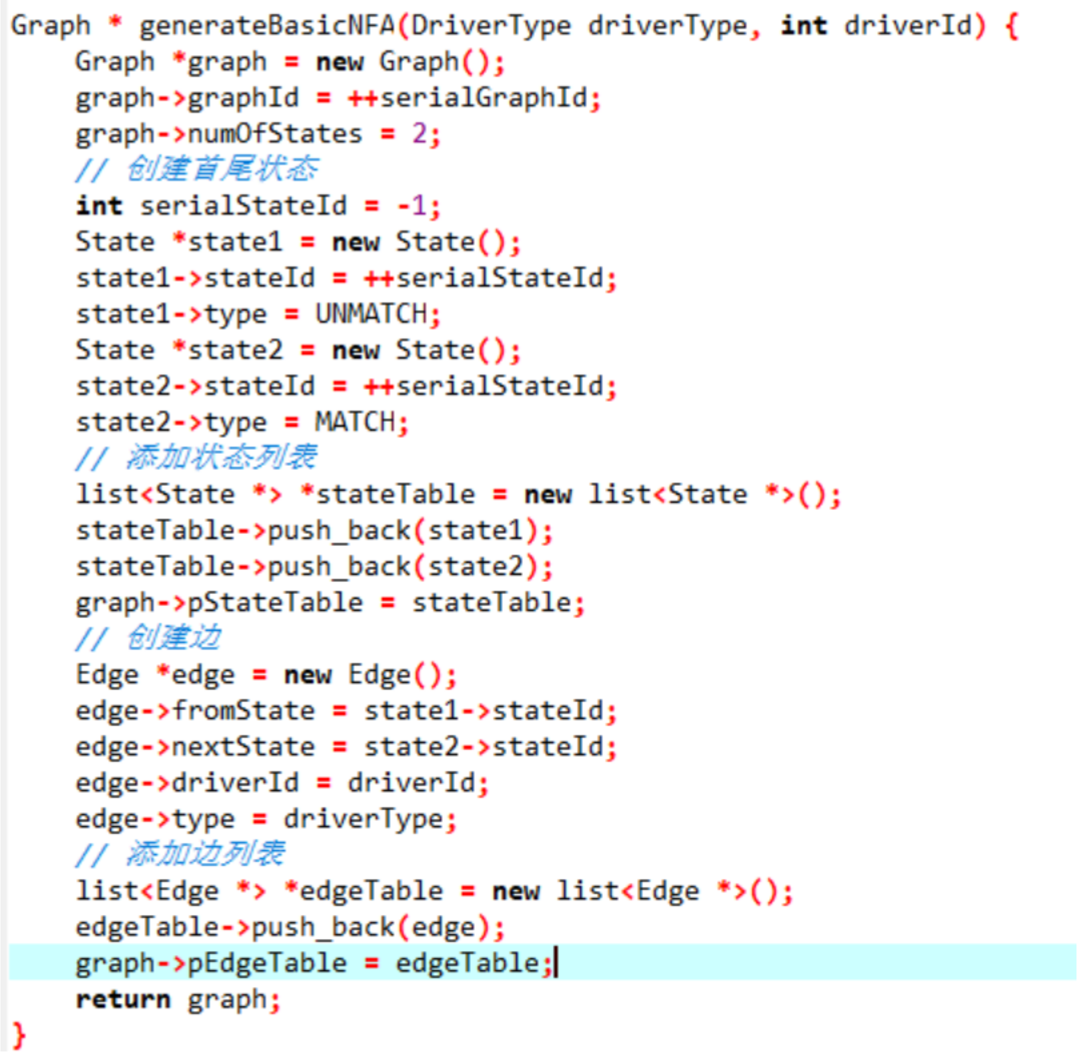


```
// 字符集与字符之间的差运算
int difference(int charSetId, char c) {
    ++serialCharSetId;
    for (list<CharSet *>::iterator it = pCharSetTable->begin(); it != pCharSetTable->end(); ++it) {
        if ((*it)->indexId == charSetId) {
            if (c == (*it)->fromChar) {
                // c为头，取余下一段
                CharSet *tmpCharSet = new CharSet();
                tmpCharSet->indexId = serialCharSetId;
                tmpCharSet->segmentId = ++serialSegmentId;
                tmpCharSet->fromChar = (*it)->fromChar + 1;
                tmpCharSet->toChar = (*it)->toChar;
                pCharSetTable->push_back(tmpCharSet);
            } else if (c > (*it)->fromChar && c < (*it)->toChar) {
                // c 为中间，取前后两段
                CharSet *tmpCharSet1 = new CharSet();
                tmpCharSet1->indexId = serialCharSetId;
                tmpCharSet1->segmentId = ++serialSegmentId;
                tmpCharSet1->fromChar = (*it)->fromChar;
                tmpCharSet1->toChar = c - 1;
                pCharSetTable->push_back(tmpCharSet1);

                CharSet *tmpCharSet2 = new CharSet();
                tmpCharSet2->indexId = serialCharSetId;
                tmpCharSet2->segmentId = ++serialSegmentId;
                tmpCharSet2->fromChar = c + 1;
                tmpCharSet2->toChar = (*it)->toChar;
                pCharSetTable->push_back(tmpCharSet2);
```

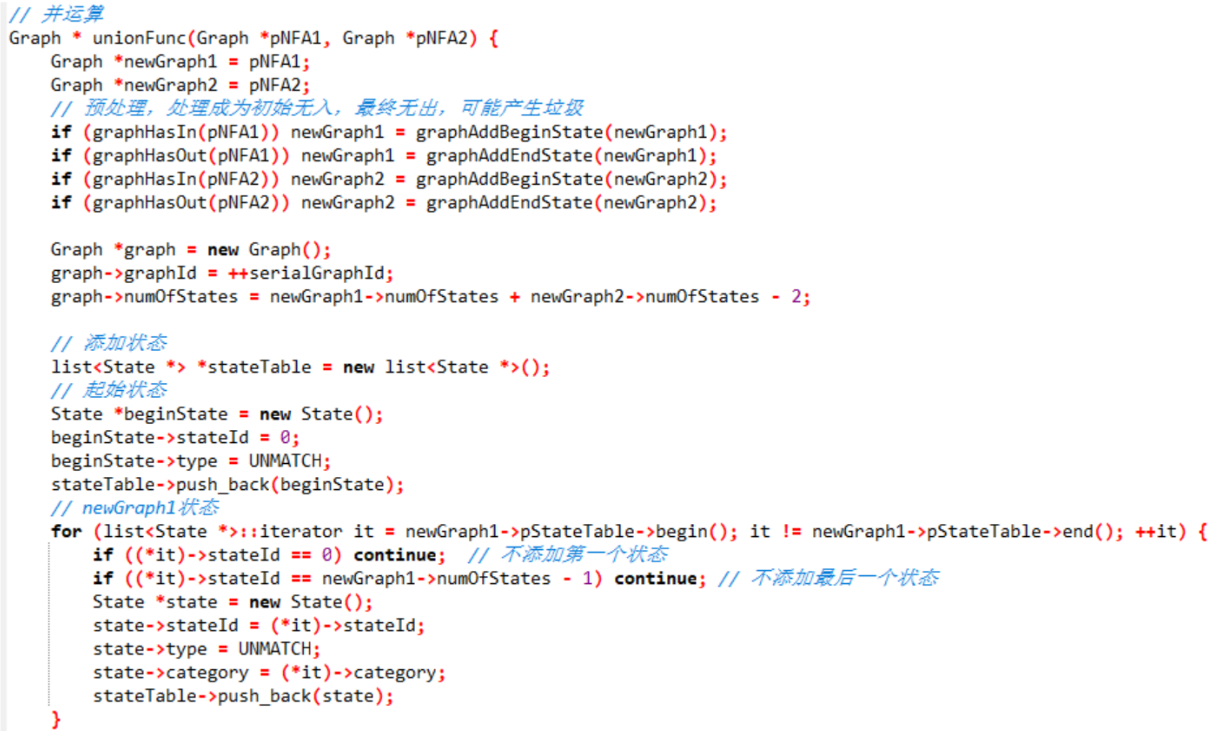


```
            } else if (c == (*it)->toChar) {
                // c为尾，取前段
                CharSet *tmpCharSet = new CharSet();
                tmpCharSet->indexId = serialCharSetId;
                tmpCharSet->segmentId = ++serialSegmentId;
                tmpCharSet->fromChar = (*it)->fromChar;
                tmpCharSet->toChar = (*it)->toChar - 1;
                pCharSetTable->push_back(tmpCharSet);
            } else {
                // c在范围外，不管
                CharSet *tmpCharSet = new CharSet();
                tmpCharSet->indexId = serialCharSetId;
                tmpCharSet->segmentId = (*it)->segmentId;
                tmpCharSet->fromChar = (*it)->fromChar;
                tmpCharSet->toChar = (*it)->toChar;
                pCharSetTable->push_back(tmpCharSet);
            }
        }
    }
    return serialCharSetId;
}
```

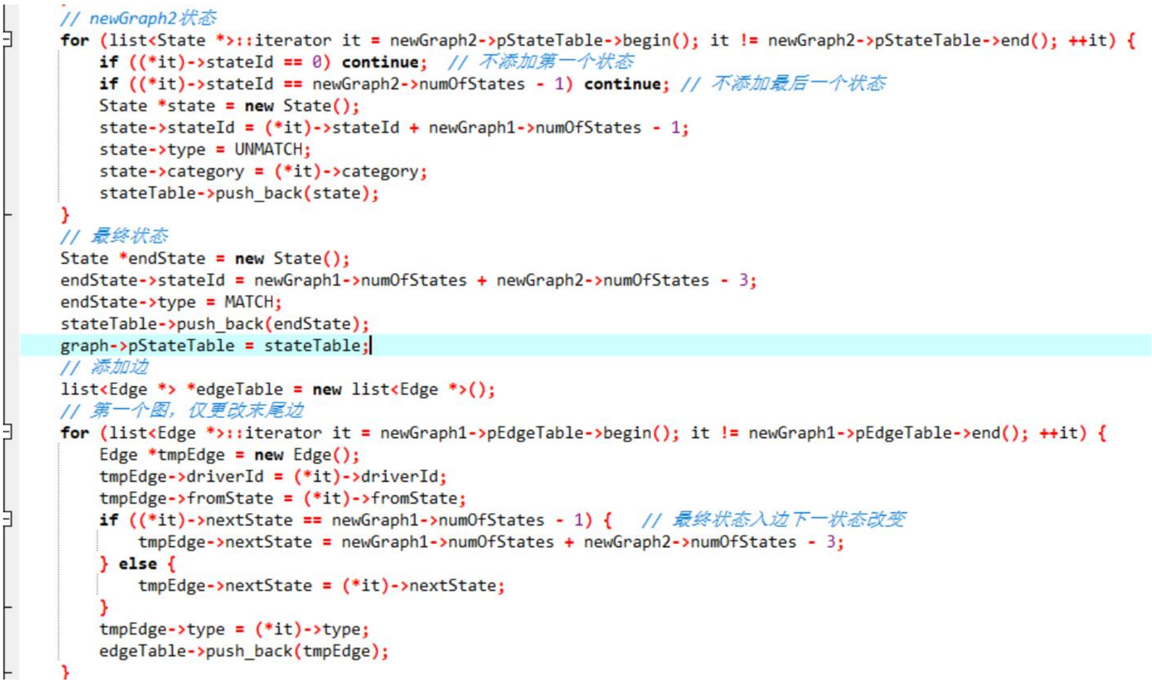
2、



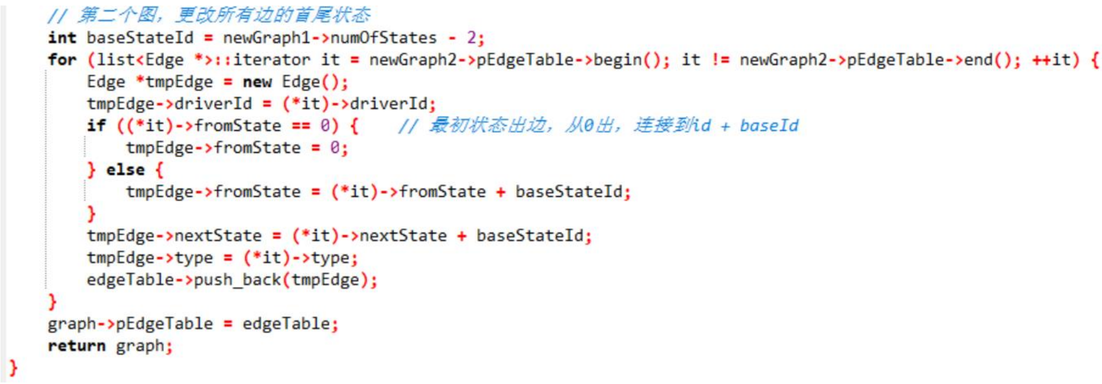
```
Graph * generateBasicNFA(DriverType driverType, int driverId) {
    Graph *graph = new Graph();
    graph->graphId = ++serialGraphId;
    graph->numOfStates = 2;
    // 创建首尾状态
    int serialStateId = -1;
    State *state1 = new State();
    state1->stateId = ++serialStateId;
    state1->type = UNMATCH;
    State *state2 = new State();
    state2->stateId = ++serialStateId;
    state2->type = MATCH;
    // 添加状态列表
    list<State *> *stateTable = new list<State *>();
    stateTable->push_back(state1);
    stateTable->push_back(state2);
    graph->pStateTable = stateTable;
    // 创建边
    Edge *edge = new Edge();
    edge->fromState = state1->stateId;
    edge->nextState = state2->stateId;
    edge->driverId = driverId;
    edge->type = driverType;
    // 添加边列表
    list<Edge *> *edgeTable = new list<Edge *>();
    edgeTable->push_back(edge);
    graph->pEdgeTable = edgeTable;
    return graph;
}
```



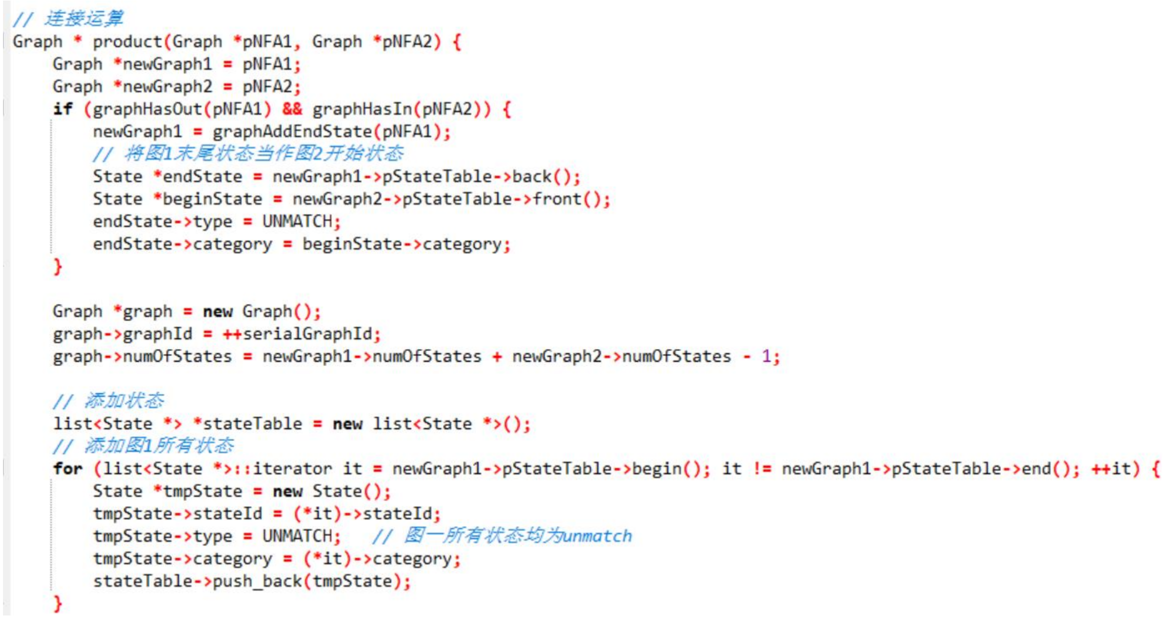
```
// 并运算
Graph * unionFunc(Graph *pNFA1, Graph *pNFA2) {
    Graph *newGraph1 = pNFA1;
    Graph *newGraph2 = pNFA2;
    // 预处理，处理成为初始无入，最终无出，可能产生垃圾
    if (graphHasIn(pNFA1)) newGraph1 = graphAddBeginState(newGraph1);
    if (graphHasOut(pNFA1)) newGraph1 = graphAddEndState(newGraph1);
    if (graphHasIn(pNFA2)) newGraph2 = graphAddBeginState(newGraph2);
    if (graphHasOut(pNFA2)) newGraph2 = graphAddEndState(newGraph2);

    Graph *graph = new Graph();
    graph->graphId = ++serialGraphId;
    graph->numOfStates = newGraph1->numOfStates + newGraph2->numOfStates - 2;

    // 添加状态
    list<State *> *stateTable = new list<State *>();
    // 起始状态
    State *beginState = new State();
    beginState->stateId = 0;
    beginState->type = UNMATCH;
    stateTable->push_back(beginState);
    // newGraph1状态
    for (list<State *>::iterator it = newGraph1->pStateTable->begin(); it != newGraph1->pStateTable->end(); ++it) {
        if ((*it)->stateId == 0) continue;  // 不添加第一个状态
        if ((*it)->stateId == newGraph1->numOfStates - 1) continue; // 不添加最后一个状态
        State *state = new State();
        state->stateId = (*it)->stateId;
        state->type = UNMATCH;
        state->category = (*it)->category;
        stateTable->push_back(state);
    }
```



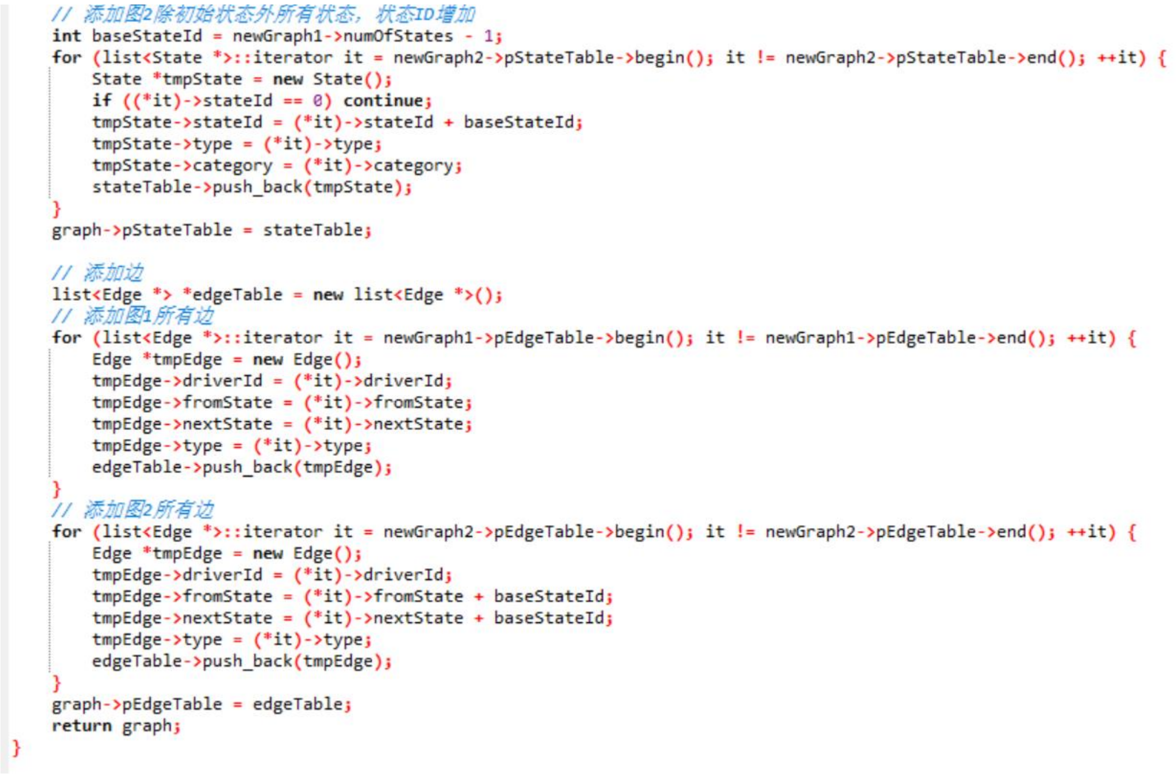
```
    // newGraph2状态
    for (list<State *>::iterator it = newGraph2->pStateTable->begin(); it != newGraph2->pStateTable->end(); ++it) {
        if ((*it)->stateId == 0) continue;  // 不添加第一个状态
        if ((*it)->stateId == newGraph2->numOfStates - 1) continue; // 不添加最后一个状态
        State *state = new State();
        state->stateId = (*it)->stateId + newGraph1->numOfStates - 1;
        state->type = UNMATCH;
        state->category = (*it)->category;
        stateTable->push_back(state);
    }
    // 最终状态
    State *endState = new State();
    endState->stateId = newGraph1->numOfStates + newGraph2->numOfStates - 3;
    endState->type = MATCH;
    stateTable->push_back(endState);
    graph->pStateTable = stateTable;
    // 添加边
    list<Edge *> *edgeTable = new list<Edge *>();
    // 第一个图，仅更改末尾边
    for (list<Edge *>::iterator it = newGraph1->pEdgeTable->begin(); it != newGraph1->pEdgeTable->end(); ++it) {
        Edge *tmpEdge = new Edge();
        tmpEdge->driverId = (*it)->driverId;
        tmpEdge->fromState = (*it)->fromState;
        if ((*it)->nextState == newGraph1->numOfStates - 1) {   // 最终状态入边下一状态改变
            tmpEdge->nextState = newGraph1->numOfStates + newGraph2->numOfStates - 3;
        } else {
            tmpEdge->nextState = (*it)->nextState;
        }
        tmpEdge->type = (*it)->type;
        edgeTable->push_back(tmpEdge);
    }
```



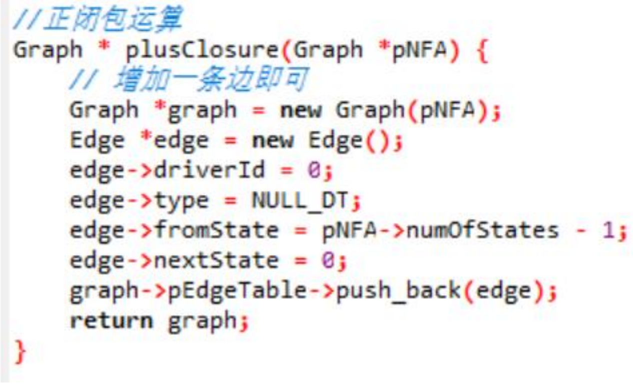
```
    // 第二个图，更改所有边的首尾状态
    int baseStateId = newGraph1->numOfStates - 2;
    for (list<Edge *>::iterator it = newGraph2->pEdgeTable->begin(); it != newGraph2->pEdgeTable->end(); ++it) {
        Edge *tmpEdge = new Edge();
        tmpEdge->driverId = (*it)->driverId;
        if ((*it)->fromState == 0) {    // 最初状态出边，从0出，连接到id + baseId
            tmpEdge->fromState = 0;
        } else {
            tmpEdge->fromState = (*it)->fromState + baseStateId;
        }
        tmpEdge->nextState = (*it)->nextState + baseStateId;
        tmpEdge->type = (*it)->type;
        edgeTable->push_back(tmpEdge);
    }
    graph->pEdgeTable = edgeTable;
    return graph;
}
```



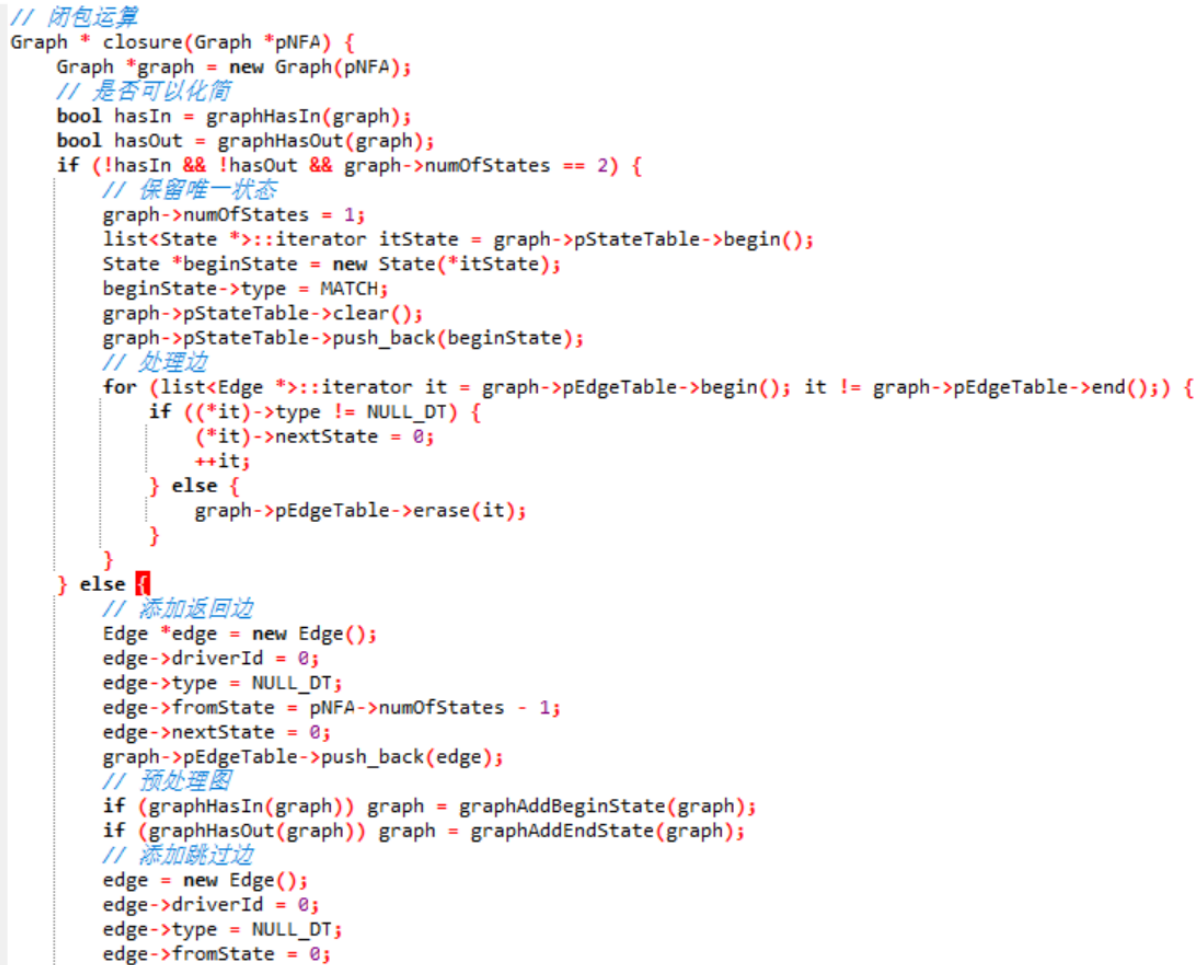
```
// 连接运算
Graph * product(Graph *pNFA1, Graph *pNFA2) {
    Graph *newGraph1 = pNFA1;
    Graph *newGraph2 = pNFA2;
    if (graphHasOut(pNFA1) && graphHasIn(pNFA2)) {
        newGraph1 = graphAddEndState(pNFA1);
        // 将图1末尾状态当作图2开始状态
        State *endState = newGraph1->pStateTable->back();
        State *beginState = newGraph2->pStateTable->front();
        endState->type = UNMATCH;
        endState->category = beginState->category;
    }

    Graph *graph = new Graph();
    graph->graphId = ++serialGraphId;
    graph->numOfStates = newGraph1->numOfStates + newGraph2->numOfStates - 1;

    // 添加状态
    list<State *> *stateTable = new list<State *>();
    // 添加图1所有状态
    for (list<State *>::iterator it = newGraph1->pStateTable->begin(); it != newGraph1->pStateTable->end(); ++it) {
        State *tmpState = new State();
        tmpState->stateId = (*it)->stateId;
        tmpState->type = UNMATCH;   // 图一所有状态均为unmatch
        tmpState->category = (*it)->category;
        stateTable->push_back(tmpState);
    }
```



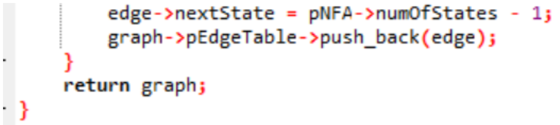
```
    // 添加图2除初始状态外所有状态，状态ID增加
    int baseStateId = newGraph1->numOfStates - 1;
    for (list<State *>::iterator it = newGraph2->pStateTable->begin(); it != newGraph2->pStateTable->end(); ++it) {
        State *tmpState = new State();
        if ((*it)->stateId == 0) continue;
        tmpState->stateId = (*it)->stateId + baseStateId;
        tmpState->type = (*it)->type;
        tmpState->category = (*it)->category;
        stateTable->push_back(tmpState);
    }
    graph->pStateTable = stateTable;

    // 添加边
    list<Edge *> *edgeTable = new list<Edge *>();
    // 添加图1所有边
    for (list<Edge *>::iterator it = newGraph1->pEdgeTable->begin(); it != newGraph1->pEdgeTable->end(); ++it) {
        Edge *tmpEdge = new Edge();
        tmpEdge->driverId = (*it)->driverId;
        tmpEdge->fromState = (*it)->fromState;
        tmpEdge->nextState = (*it)->nextState;
        tmpEdge->type = (*it)->type;
        edgeTable->push_back(tmpEdge);
    }
    // 添加图2所有边
    for (list<Edge *>::iterator it = newGraph2->pEdgeTable->begin(); it != newGraph2->pEdgeTable->end(); ++it) {
        Edge *tmpEdge = new Edge();
        tmpEdge->driverId = (*it)->driverId;
        tmpEdge->fromState = (*it)->fromState + baseStateId;
        tmpEdge->nextState = (*it)->nextState + baseStateId;
        tmpEdge->type = (*it)->type;
        edgeTable->push_back(tmpEdge);
    }
    graph->pEdgeTable = edgeTable;
    return graph;
}
```



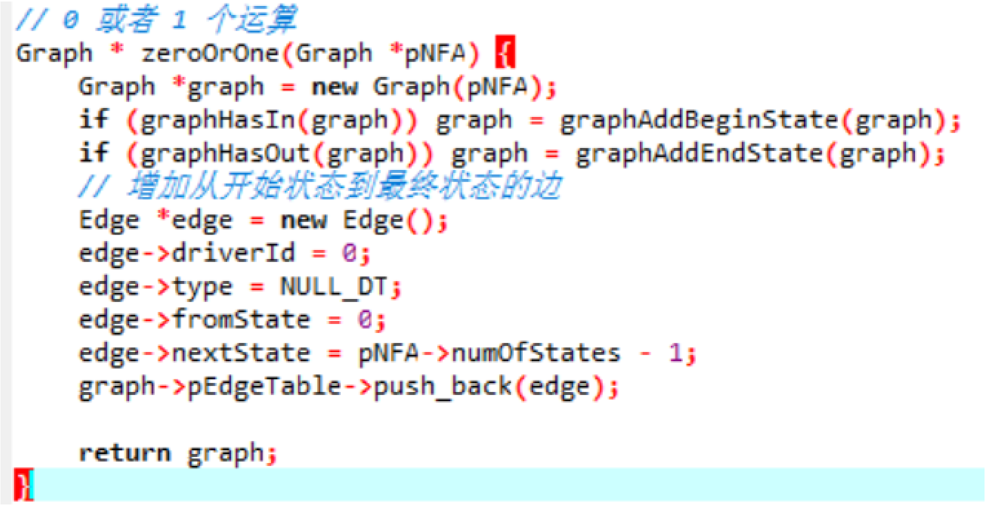
```
//正闭包运算
Graph * plusClosure(Graph *pNFA) {
    // 增加一条边即可
    Graph *graph = new Graph(pNFA);
    Edge *edge = new Edge();
    edge->driverId = 0;
    edge->type = NULL_DT;
    edge->fromState = pNFA->numOfStates - 1;
    edge->nextState = 0;
    graph->pEdgeTable->push_back(edge);
    return graph;
}
```



```
// 闭包运算
Graph * closure(Graph *pNFA) {
    Graph *graph = new Graph(pNFA);
    // 是否可以化简
    bool hasIn = graphHasIn(graph);
    bool hasOut = graphHasOut(graph);
    if (!hasIn && !hasOut && graph->numOfStates == 2) {
        // 保留唯一状态
        graph->numOfStates = 1;
        list<State *>::iterator itState = graph->pStateTable->begin();
        State *beginState = new State(*itState);
        beginState->type = MATCH;
        graph->pStateTable->clear();
        graph->pStateTable->push_back(beginState);
        // 处理边
        for (list<Edge *>::iterator it = graph->pEdgeTable->begin(); it != graph->pEdgeTable->end();) {
            if ((*it)->type != NULL_DT) {
                (*it)->nextState = 0;
                ++it;
            } else {
                graph->pEdgeTable->erase(it);
            }
        }
    } else {
        // 添加返回边
        Edge *edge = new Edge();
        edge->driverId = 0;
        edge->type = NULL_DT;
        edge->fromState = pNFA->numOfStates - 1;
        edge->nextState = 0;
        graph->pEdgeTable->push_back(edge);
        // 预处理图
        if (graphHasIn(graph)) graph = graphAddBeginState(graph);
        if (graphHasOut(graph)) graph = graphAddEndState(graph);
        // 添加跳过边
        edge = new Edge();
        edge->driverId = 0;
        edge->type = NULL_DT;
        edge->fromState = 0;
```

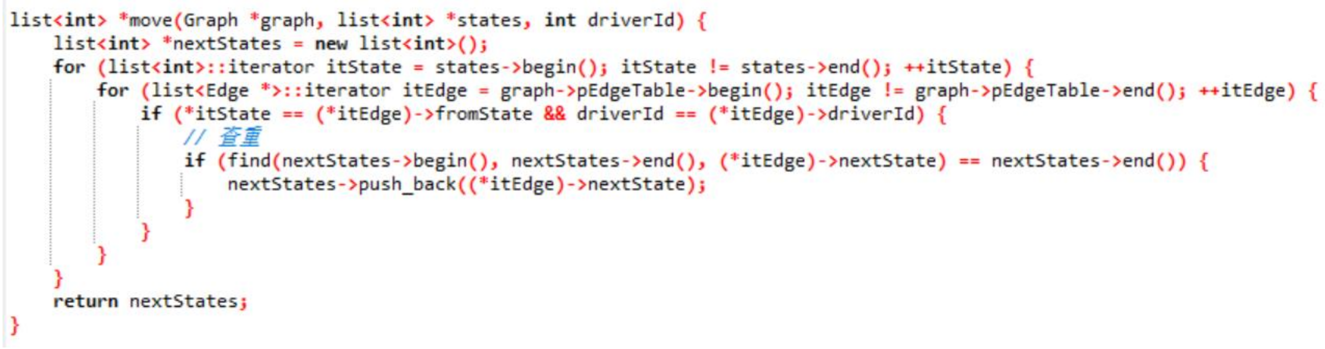


```
        edge->nextState = pNFA->numOfStates - 1;
        graph->pEdgeTable->push_back(edge);
    }
    return graph;
}
```

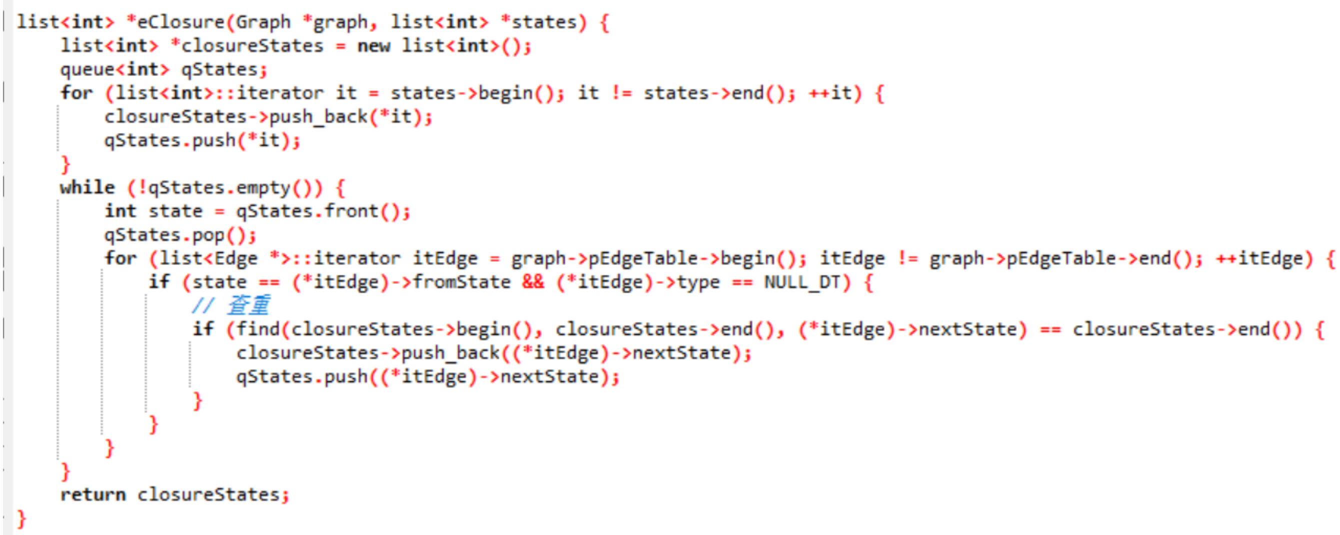


```
// 0 或者 1 个运算
Graph * zeroOrOne(Graph *pNFA) {
    Graph *graph = new Graph(pNFA);
    if (graphHasIn(graph)) graph = graphAddBeginState(graph);
    if (graphHasOut(graph)) graph = graphAddEndState(graph);
    // 增加从开始状态到最终状态的边
    Edge *edge = new Edge();
    edge->driverId = 0;
    edge->type = NULL_DT;
    edge->fromState = 0;
    edge->nextState = pNFA->numOfStates - 1;
    graph->pEdgeTable->push_back(edge);

    return graph;
}
```

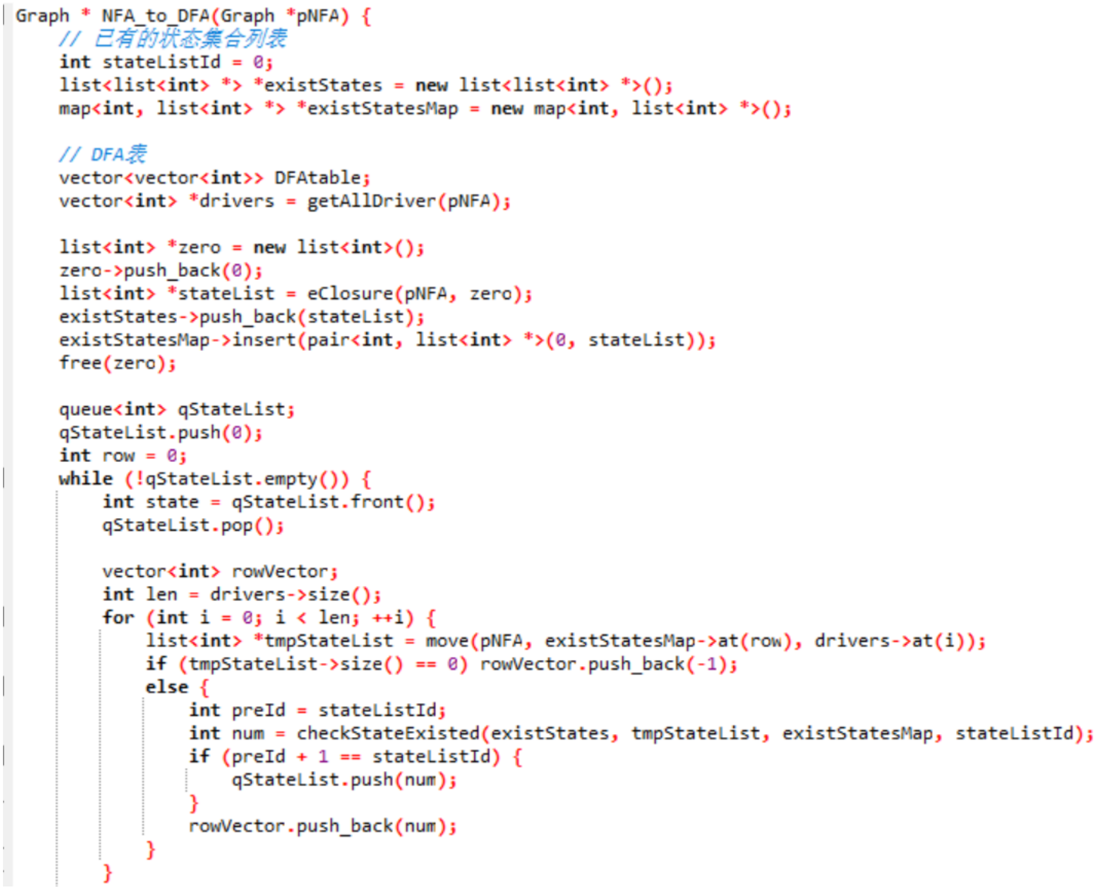
3、



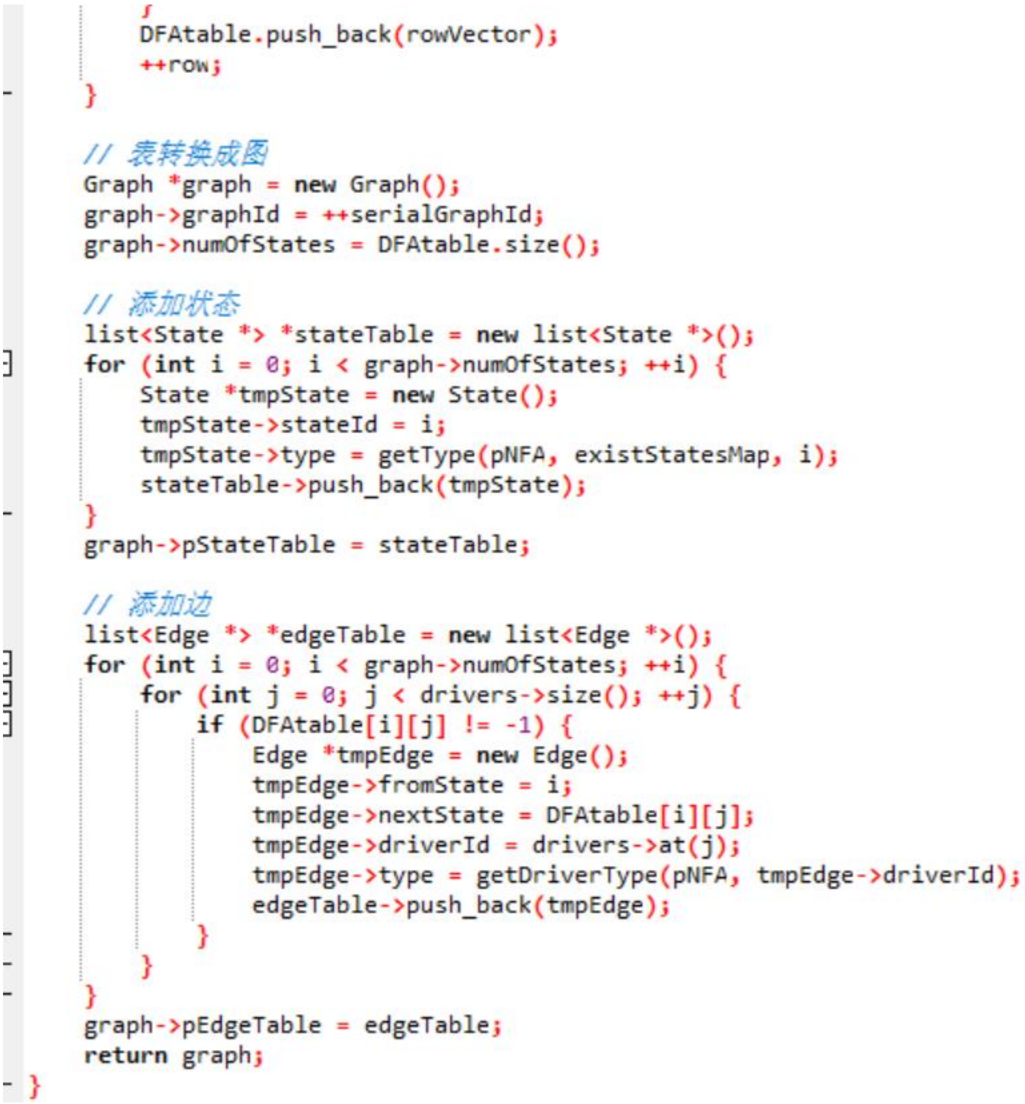
```
list<int> *move(Graph *graph, list<int> *states, int driverId) {
    list<int> *nextStates = new list<int>();
    for (list<int>::iterator itState = states->begin(); itState != states->end(); ++itState) {
        for (list<Edge *>::iterator itEdge = graph->pEdgeTable->begin(); itEdge != graph->pEdgeTable->end(); ++itEdge) {
            if (*itState == (*itEdge)->fromState && driverId == (*itEdge)->driverId) {
                // 查重
                if (find(nextStates->begin(), nextStates->end(), (*itEdge)->nextState) == nextStates->end()) {
                    nextStates->push_back((*itEdge)->nextState);
                }
            }
        }
    }
    return nextStates;
}
```



```
list<int> *eClosure(Graph *graph, list<int> *states) {
    list<int> *closureStates = new list<int>();
    queue<int> qStates;
    for (list<int>::iterator it = states->begin(); it != states->end(); ++it) {
        closureStates->push_back(*it);
        qStates.push(*it);
    }
    while (!qStates.empty()) {
        int state = qStates.front();
        qStates.pop();
        for (list<Edge *>::iterator itEdge = graph->pEdgeTable->begin(); itEdge != graph->pEdgeTable->end(); ++itEdge) {
            if (state == (*itEdge)->fromState && (*itEdge)->type == NULL_DT) {
                // 查重
                if (find(closureStates->begin(), closureStates->end(), (*itEdge)->nextState) == closureStates->end()) {
                    closureStates->push_back((*itEdge)->nextState);
                    qStates.push((*itEdge)->nextState);
                }
            }
        }
    }
    return closureStates;
}
```

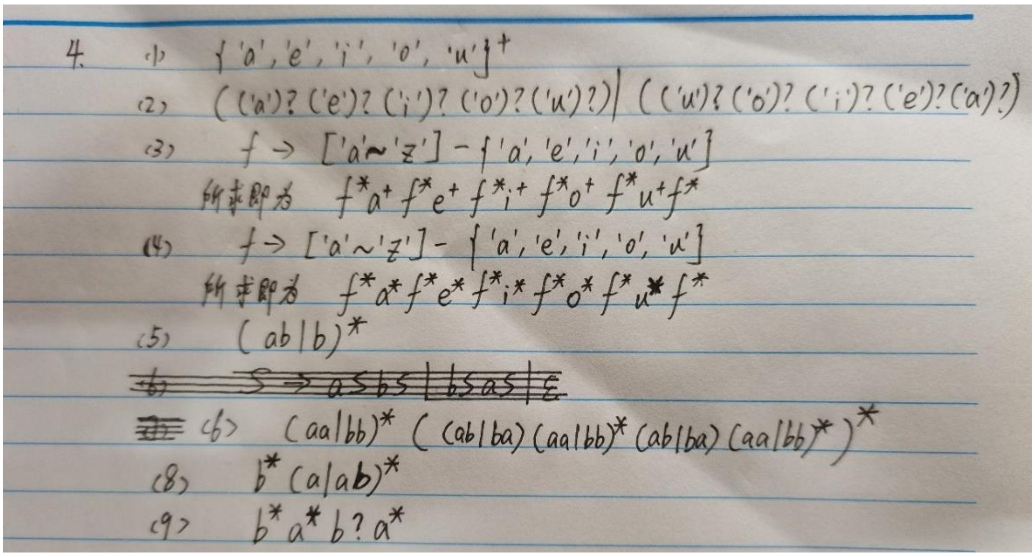


```
Graph * NFA_to_DFA(Graph *pNFA) {
    // 已有的状态集合列表
    int stateListId = 0;
    list<list<int> *> *existStates = new list<list<int> *>();
    map<int, list<int> *> *existStatesMap = new map<int, list<int> *>();

    // DFA表
    vector<vector<int>> DFAtable;
    vector<int> *drivers = getAllDriver(pNFA);

    list<int> *zero = new list<int>();
    zero->push_back(0);
    list<int> *stateList = eClosure(pNFA, zero);
    existStates->push_back(stateList);
    existStatesMap->insert(pair<int, list<int> *>(0, stateList));
    free(zero);

    queue<int> qStateList;
    qStateList.push(0);
    int row = 0;
    while (!qStateList.empty()) {
        int state = qStateList.front();
        qStateList.pop();

        vector<int> rowVector;
        int len = drivers->size();
        for (int i = 0; i < len; ++i) {
            list<int> *tmpStateList = move(pNFA, existStatesMap->at(row), drivers->at(i));
            if (tmpStateList->size() == 0) rowVector.push_back(-1);
            else {
                int preId = stateListId;
                int num = checkStateExisted(existStates, tmpStateList, existStatesMap, stateListId);
                if (preId + 1 == stateListId) {
                    qStateList.push(num);
                }
                rowVector.push_back(num);
            }
        }
```



```
        }
        DFAtable.push_back(rowVector);
        ++row;
    }

    // 表转换成图
    Graph *graph = new Graph();
    graph->graphId = ++serialGraphId;
    graph->numOfStates = DFAtable.size();

    // 添加状态
    list<State *> *stateTable = new list<State *>();
    for (int i = 0; i < graph->numOfStates; ++i) {
        State *tmpState = new State();
        tmpState->stateId = i;
        tmpState->type = getType(pNFA, existStatesMap, i);
        stateTable->push_back(tmpState);
    }
    graph->pStateTable = stateTable;

    // 添加边
    list<Edge *> *edgeTable = new list<Edge *>();
    for (int i = 0; i < graph->numOfStates; ++i) {
        for (int j = 0; j < drivers->size(); ++j) {
            if (DFAtable[i][j] != -1) {
                Edge *tmpEdge = new Edge();
                tmpEdge->fromState = i;
                tmpEdge->nextState = DFAtable[i][j];
                tmpEdge->driverId = drivers->at(j);
                tmpEdge->type = getDriverType(pNFA, tmpEdge->driverId);
                edgeTable->push_back(tmpEdge);
            }
        }
    }
    graph->pEdgeTable = edgeTable;
    return graph;
}
```

4、



4. (1) $\{'a', 'e', 'i', 'o', 'u'\}^+$

(2) $(('a')?('e')?('i')?('o')?('u')?) \mid (('u')?('o')?('i')?('e')?('a')?)$

(3) $f \rightarrow ['a' \sim 'z'] - \{'a', 'e', 'i', 'o', 'u'\}$

所求即为 $f^*a^+f^*e^+f^*i^+f^*o^+f^*u^+f^*$

(4) $f \rightarrow ['a' \sim 'z'] - \{'a', 'e', 'i', 'o', 'u'\}$

所求即为 $f^*a^*f^*e^*f^*i^*f^*o^*f^*u^*f^*$

(5) $(ab \mid b)^*$

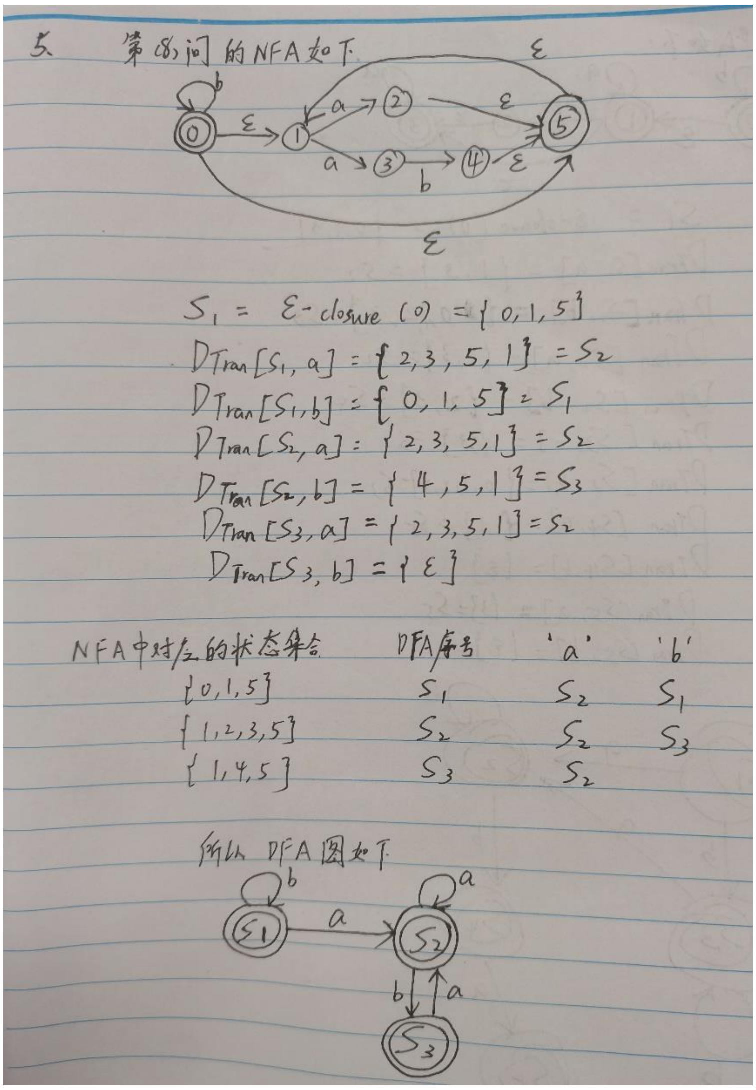
~~(6) $S \rightarrow aSbS \mid bSaS \mid \varepsilon$~~

(6) $(aa \mid bb)^*((ab \mid ba)(aa \mid bb)^*(ab \mid ba)(aa \mid bb)^*)^*$

(8) $b^*(a \mid ab)^*$

(9) $b^*a^*b?a^*$

5、



5. 第(8)问的NFA如下

b
0
ε
1
a
2
ε
5
a
3
b
4
ε
ε
ε

$S_1 = \varepsilon\text{-closure}(0) = \{0, 1, 5\}$

$DTran[S_1, a] = \{2, 3, 5, 1\} = S_2$

$DTran[S_1, b] = \{0, 1, 5\} = S_1$

$DTran[S_2, a] = \{2, 3, 5, 1\} = S_2$

$DTran[S_2, b] = \{4, 5, 1\} = S_3$

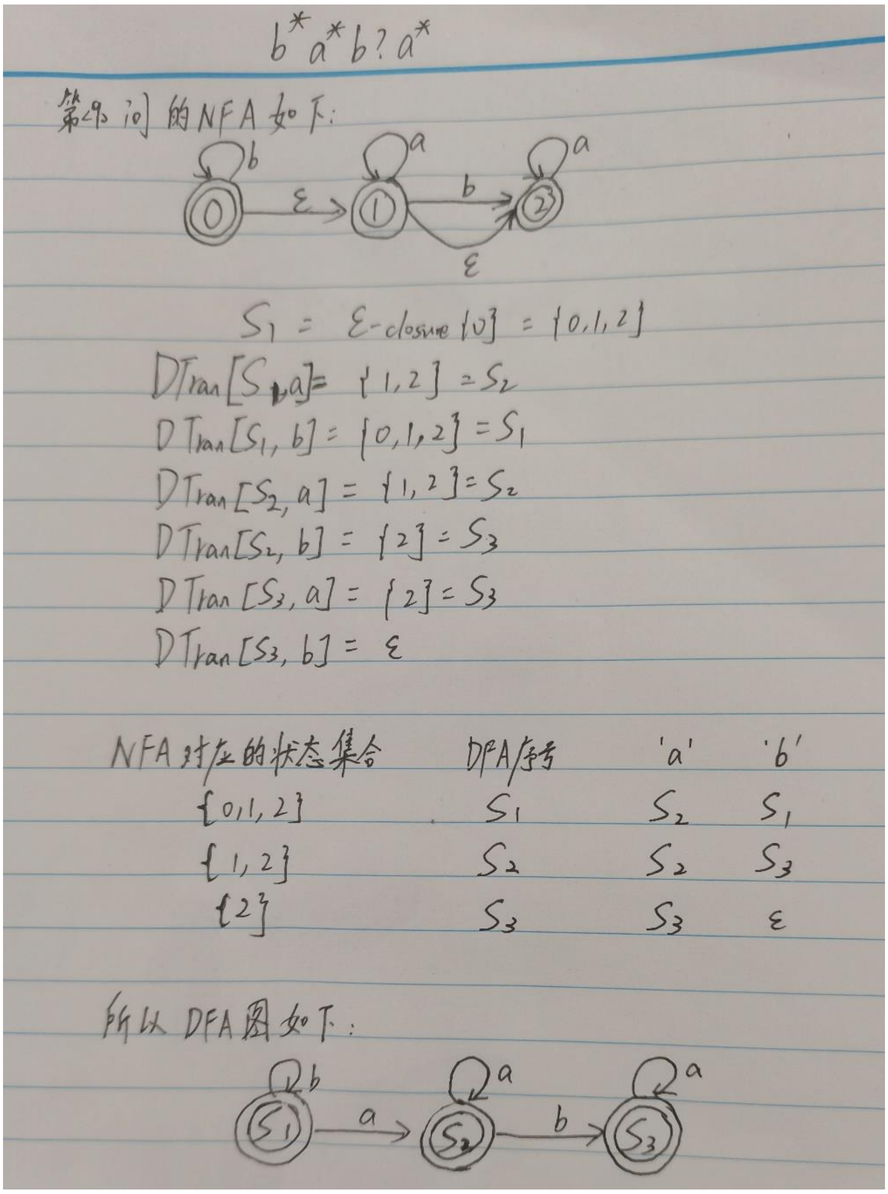
$DTran[S_3, a] = \{2, 3, 5, 1\} = S_2$

$DTran[S_3, b] = \{\varepsilon\}$

NFA中对应的状态集合	DFA序号	'a'	'b'
{0,1,5}	S_1	S_2	S_1
{1,2,3,5}	S_2	S_2	S_3
{1,4,5}	S_3	S_2	

所以 DFA图如下

b
S_1
a
a
S_2
b
a
S_3



$b^*a^*b?a^*$

第(9)问 的NFA 如下:

b
a
a
0
ε
1
b
2
ε

$S_1 = \varepsilon\text{-closure}\{0\} = \{0,1,2\}$

$DTran[S_1,a] = \{1,2\} = S_2$

$DTran[S_1,b] = \{0,1,2\} = S_1$

$DTran[S_2,a] = \{1,2\} = S_2$

$DTran[S_2,b] = \{2\} = S_3$

$DTran[S_3,a] = \{2\} = S_3$

$DTran[S_3,b] = \varepsilon$

NFA对应的状态集合	DFA序号	'a'	'b'
$\{0,1,2\}$	S_1	S_2	S_1
$\{1,2\}$	S_2	S_2	S_3
$\{2\}$	S_3	S_3	ε

所以 DFA图如下:

b
a
a
S1
a
S2
b
S3