

**课程设计**

**课 程： 算法与数据结构**

**题 目： 简单行编辑器**

**专 业： 软件工程**

**班 级： 2201班**

**学 号： 3221306105**

**姓 名： 张宏志**

**2024年 1月7日**

**答辩记录**

|  |
| --- |
| **题目** |
| **答辩问题：（至少3个问题）**  **1、**  **2、**  **3、** |
| **问题回答：**  **1、**  **2、**  **3、** |

# 题目：简单行编辑程序

## 一、要解决的问题

文本编辑程序是利用计算机进行文字加工的基本软件工具，实现对文本文件的插入、删除等修改操作。限制这些操作以行为单位进行的编辑程序称为行编辑程序。被编辑的文本文件可能很大，全部读入编辑程序的数据空间（内存）的做>法既不经济，也不总能实现。一种解决方法是逐段地编辑。任何时刻只把待编辑文件的一段放在内存，称为活区。试按照这种方法实现一个简单的行编辑程序。设文件每行不超过320个字符，很少超过80字符。

## 二、算法基本思想描述

字符串匹配算法：BF、KMP、Tire以及字符串哈希、AC自动机等。

1. **BF算法**：暴力枚举匹配起点和模式串的每一个字符，进行逐个匹配。**时间复杂度：O(NM)**

2. **KMP算法**：该算法的核心是利用匹配失败后的信息，尽量减少模式串与母串的匹配次数以达到快速匹配的目的。具体实现就是通过一个next[]函数实现，函数本身包含了模式串的局部匹配信息。**时间复杂度：O(N+M)**

3. **Trie算法**：Trie树(字典树)是一个可以高效地存储和查找字符串集合的一种多叉树的数据结构。每个节点都拥有若干个字符指针，若在插入或检索字符串时扫描到一个字符c，就沿着当前节点的c字符指针，走向该指针指向的节点>。**时间复杂度：O(M)，空间复杂度：O(NM)。**（其中N为节点个数，M为字符集大小）

## 三、程序设计思路

### 1. 功能描述

程序开始运行时需要键入输入和输出文件。其中输入文件可以为空，表示不从外部文件读取文本。之后程序会在屏幕上显示当前活区所展示的文本内容（最多100行），如果你是第一次使用该编辑器，可以键入help获取帮助菜单。

#### help menu

1. 行插入。格式：**i<行号><回车><文本><回车>** 将<文本>插入活区中第<行号>行之后。

**- Insert Line: Format - i<line\_number><Enter><text><Enter>.**

**Inserts <text> after line <line\_number> in the Active Area.**

2. 行删除。格式：**d<行号1>[ <行号2>]<回车>** 删除活区中第<行号1>行（到第<行号2>行）。

**- Delete Line: Format - d<line\_number1>[ <line\_number2>]<Enter>.**

**Deletes line <line\_number1> (to line <line\_number2>) in the Active Area.**

3. 活区切换。格式：**n<回车>** 将活区写入输出文件，并从输入文件中读入下一段，作为新的活区。

**- Switch Active Area: Format - n<Enter>**

**Write the Active Area to the output file, and read the next segment from the input file as the new Active Area.**

4. 活区显示。格式：**p<回车>** 逐页地（每页20行）显示活区内容。

**- Display Active Area: Format - p<Enter>**

**Display the Active Area content page by page (20 lines per page).**

**5**. 串替换。格式：**S<行号1>[<,行号2>]@<串1>@<串2><回车>** 将第<行号>行中的<串1>替换成<串2>。

**- Replace String:**

**Format - S<line\_number1><,line\_number2>@<string1>@<string2><Enter>**

**Replace <string1> with <string2> in line <line\_number1>( to line <line\_number2>) of the Active Area.**

6. 串匹配。格式：**m <串>[ <行号1>[ <行号2>]]<回车>** 匹配（第<行号1>行（到第<行号2>行））所有<串>，并打印匹配成功的位置。

**- Match String: Format - m <string>[ <line\_number1>[ <line\_number2>]]<Enter>**

**Matched <string>( in line <line\_number1>( to line <line\_number2>)) in the Active Area and print the position where the match is successful.**

7. 读取外部文件。格式：**r <文件名><回车>** 若当前非活区为空，则可以从外部文件中读入新文本到活区中作为待操作内容。

**- Read From File: Format - r <filename><Enter>**

**If Other Area is empty, new text can be read from <filename> to Active Area as the content to be operated.**

8. 写入外部文件。格式：**w <文件名>[ <行号1>[ <行号2>]]** 将活区中（第<行号1>行（到第<行号2>行））的内容写入外部文件中。

**- Write To File:**

**Format - w <filename>[ <line\_number1>[ <line\_number2>]]<Enter>**

**Write the contents of the Active Area (line <line number 1> (to line <line number 2>)) into <filename>.**

9. 退出编辑器。格式：**exit<回车>**

**- Exit the editor: Format - exit**

### 2. 活区与非活区

活区指的是用户当前可修改的文本区，活区最多可以存100行文本，每行文本的最大长度为320个字符。用户可以从输入文件中读取文本到活区中，若输入文件文本内容多于80行，则只读取前80行文本，留下20行给用户操作的空间，而输入文件中的其余文本会暂时存入非活区中。

用户可对文本进行自定义操作，操作结束后按下`n`写到输出文件中。此时若非活区中还存有文本，则将非活区中的文本继续按照上述方式继续读入活区中。

考虑到文本文件行长通常为正态分布，且峰值在60 ~ 70之间，用320×100大小的字符数组实现存储将造成大量浪费。因此本编辑器采用以标准行块为单位为各行分配存储，每个标准行块含80个字符，并且用动态链表将每一个行块连接起来，从而不会导致内存的浪费。

### 3. 活区的表示

每一行采用行块链表存储，每一块最多存储80个字符，若超出限制，则新增一个行块，并且将其于前面的行块连接起来，构成动态链表的形式。如下图所示：

图示

描述已自动生成

### 4. 字符串匹配算法的设计

在字符串匹配方面：用户可以自行选择采用哪种匹配算法对字符串进行匹配，目前仅提供4种算法：KMP、Trie、字符串哈希、BF。默认采用KMP算法进行字符串匹配。

1. **KMP算法**

该算法的核心为next数组，next数组的含义：next[i]表示以i为结尾的后缀最多有多少个字符与前缀相等。在匹配过程中，若未匹配成功，则跳到next所指向的地方继续匹配，而不是每次都跳到起点重新匹配。更详细的算法思路可以参考我之前写过的一篇博客：

<https://blog.csdn.net/m0_75186429/article/details/134505128?spm=1001.2014.3001.5501>

1. **Trie树**

Trie树又称字典树或前缀树，是一种用于实现字符串快速检索的多叉树结构。匹配效率和KMP算法不相上下。

1. **字符串哈希**

字符串哈希思路：1. 先预处理所有前缀的哈希，把字符串看成是一个p进制的数，每一位上的字符表示它p进制的数（当p取131或13331，Q取2^64时，99.99%情况下可以假定不会发生冲突）。由于最后要对Q取模，因此可以直接用`unsigned long long`来存储哈希值，溢出的时候会自动对2^64取模。

s = "ABCD"

-> 'A' 'B' 'C' 'D'

   (1   2   3   4)p // p进制为：1234

 = 1 \* p^3 + 2 \* p^2 + 3 \* p^1 + 4 \* p^0 // 十进制

// 这样就可以把一个字符串转换成一个数字

// 这个数字可能会非常大，因此要在最后模上一个非常小的数Q

// 这样就可以把很大的数映射到0 ~ Q - 1

1. **BF算法**

BF(Brute Force)算法。顾名思义，是一种最朴素也是最易理解的字符串匹配算法。在匹配过程中，若未匹配成功，则直接跳到起点进行下一次匹配，时间复杂度较高，效率不如KMP算法。

int n = strlen(str), m = strlen(match\_str);

// 暴力枚举每次匹配的起点

    for (int i = 0; i <= n - m; i ++ )

    {

        int j;

        for (j = 0; j < m; j ++ )  // 枚举匹配长度

            if (str[i + j] != match\_str[j])

                break;

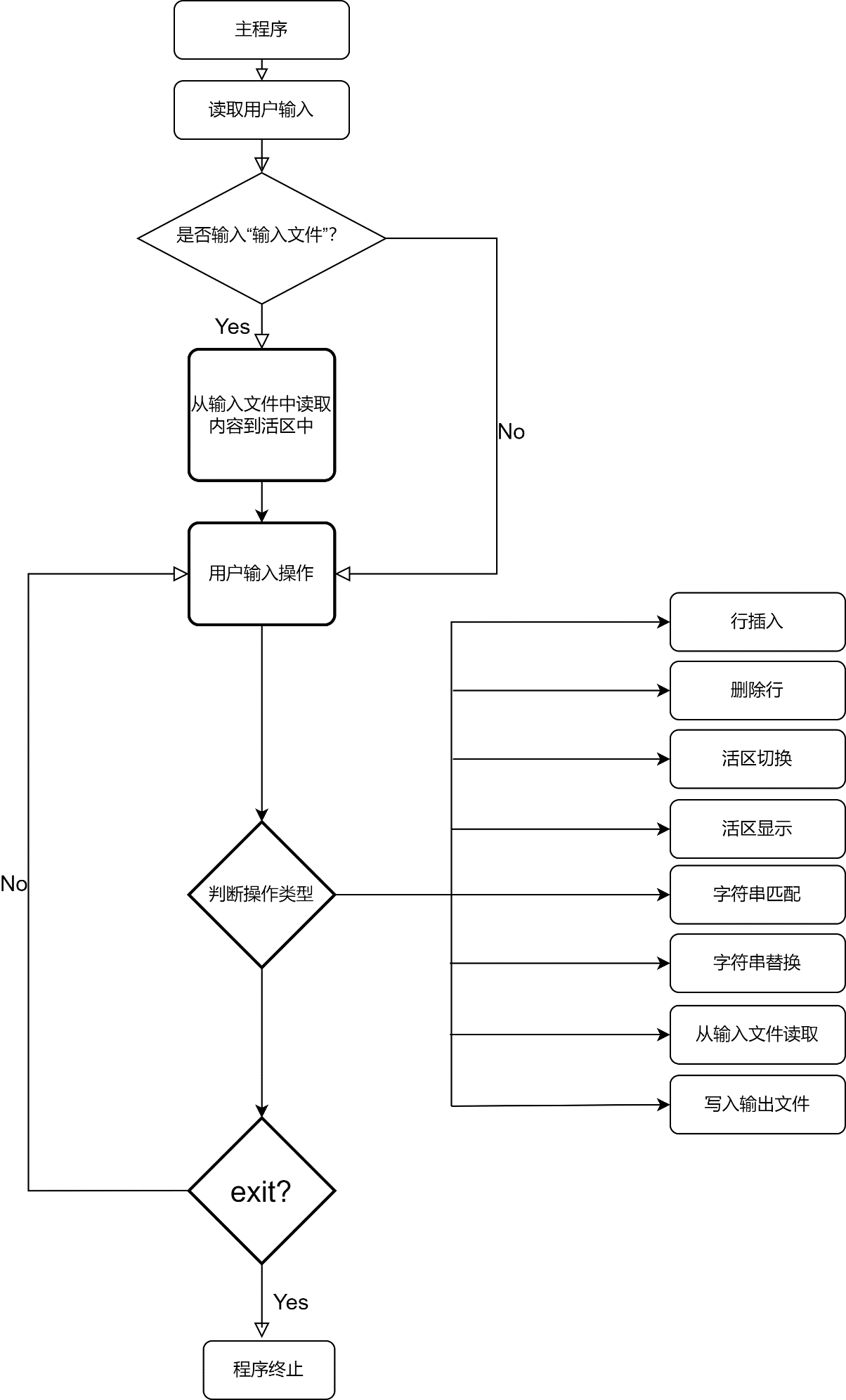
        // 若匹配成功，保存位置

        if (j == m) pos[cnt ++ ] = i;

    }

## 四、程序模块结构

### 1. 流程图

****

### 2. 结构体设计

// 存储每一行的文本，链式存储

typedef struct LineBlock {

    char data[MAXBLOCKSIZE];  // 行中每一块的字符内容

    LineBlock\* next;  // 指向下一个行块

    // 默认构造

    LineBlock(const char\* text, LineBlock\* ne = nullptr)

    {

        int length = strlen(text);

        memcpy(data, text, min(length, MAXLINESIZE - 1));  // 把text的内容拷贝给data

        data[length] = '\0';

        next = ne;

    }

} LineBlock;

// 存储活区中的每一行，链式存储

typedef struct Line {

    int line\_no;  // 行号

    LineBlock\* content;  // 指向行内容

    Line\* next;  // 指向下一行

    Line(int number, LineBlock\* line\_block, Line\* ne = nullptr) : line\_no(number), content(line\_block), next(ne) {}

} Line, \*ActiveArea;

// 存储活区中匹配到的字符串所处的位置(row,col)

typedef struct Position {

    int row, col;

} Pos;

// Trie节点

struct TrieNode {

    bool isEnd;  // 标记是否是单词的结尾

    std::unordered\_map<char, TrieNode\*> son;  // 子节点集合

    TrieNode() : isEnd(false) {}

};

// Trie树

class Trie {

public:

    // 默认构造

    Trie() : root(new TrieNode()) {}

    // 插入字符串

    void insert(const char\* word)

    {

        TrieNode\* node = root;

        while (\*word)

        {

            if (node->son.find(\*word) == node->son.end())

                node->son[\*word] = new TrieNode();

            node = node->son[\*word];

            word ++ ;

        }

        node->isEnd = true;

    }

    // 搜索字符串

    bool search(const char\* src, const char\* match\_str, int pos[], int& cnt)

    {

        cnt = 0;

        insert(match\_str);

        int len = strlen(src);

        for (int i = 0; i < len; i ++ )

        {

            TrieNode\* node = this->root;

            for (int j = i; j < len; j ++ )

            {

                if (node->son.find(src[j]) == node->son.end()) break;

                node = node->son[src[j]];

                // 匹配成功，保存位置

                if (node->isEnd) pos[cnt ++ ] = i;

            }

        }

        return cnt > 0;

    }

private:

    TrieNode\* root;

};

### 3. 主要函数设计

/\* 显示函数 \*/

void showMainMenu();

void showHelpMenu();

void showActiveArea(const ActiveArea&, int, int, int);

void printLine(const LineBlock\*);

void printPositions(Pos[], int);

/\* 辅助函数 \*/

void readFile(char\*, char\*);

int getValidInput();

void getNumber(const char\*, int&, int, int);

void getString(const char\*, char\*, int, int);

bool CHECK\_AREA(const ActiveArea&);

bool CHECK\_BLOCK(const LineBlock\*);

bool CHECK\_BOUND(int, int, int, int low = 0);

/\* 对Line操作的一些函数 \*/

bool initArea(ActiveArea&);

int countLine(const ActiveArea&);

bool emptyArea(const ActiveArea&);

bool insertLine(ActiveArea&, char\*, int, ActiveArea&, char\* output\_file = nullptr);

void changeActiveArea(ActiveArea&, ActiveArea&, char\*);

bool deleteLine(ActiveArea&, ActiveArea&, int, int);

bool matchString(ActiveArea&, char\*, int, int, Pos[], int&, int);

bool replaceString(ActiveArea&, int, char\*, char\*, int&, int);

/\* 对LineBlock操作的一些函数 \*/

bool insertLineBlock(LineBlock\*&, char\*);

bool emptyLineBlock(const LineBlock\*);

bool clearLineBlock(LineBlock\*&);

void blocks\_to\_str(LineBlock\*, char\*);

void str\_to\_blocks(char\*, LineBlock\*&);

/\* 对文件操作的一些函数 \*/

bool readFromInputFile(char\*, ActiveArea&, ActiveArea&, char\* ouput\_file = nullptr);

bool writeToOutputFile(char\*, ActiveArea&, ActiveArea&, int, int);

bool readFromOtherArea(ActiveArea&, ActiveArea&, int);

/\* 对字符串操作的一些函数 \*/

bool insertString(char\*, int, const char\*);

bool eraseString(char\*, int, int);

bool kmp(char\*, char\*, int[], int&);

bool trie(char\*, char\*, int[], int&);

unsigned long long get(int, int, unsigned long long[], unsigned long long[]);

bool string\_hash(char\*, char\*, int[], int&);

bool bf(char\*, char\*, int[], int&);

## 五、测试数据及测试结果

#### 测试数据

输入：

input.txt output.txt

help

i1<回车>hello world

d4 6

m hello

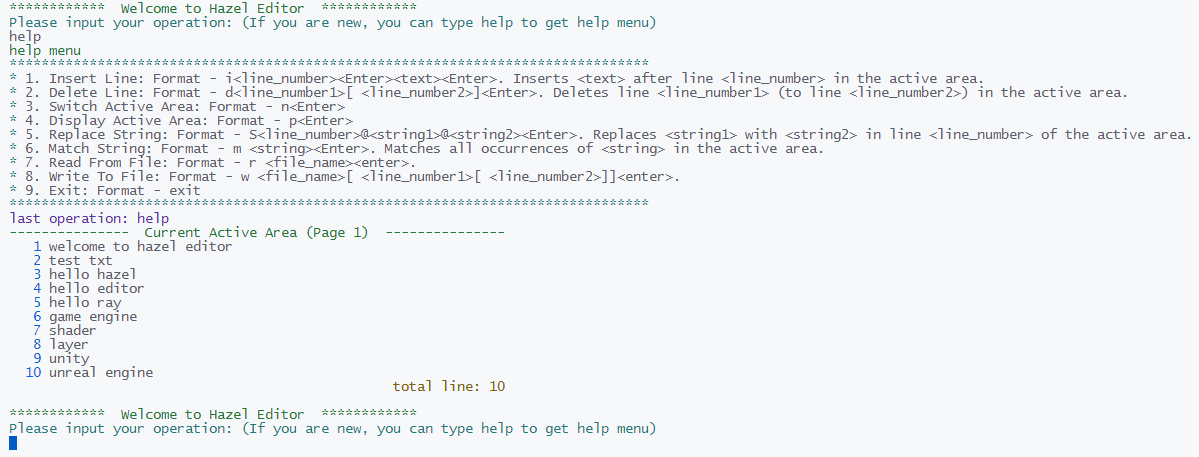
S2@world@hazel

p

exit

#### 测试结果





图片包含 文本

描述已自动生成

文本

描述已自动生成

图片包含 文本

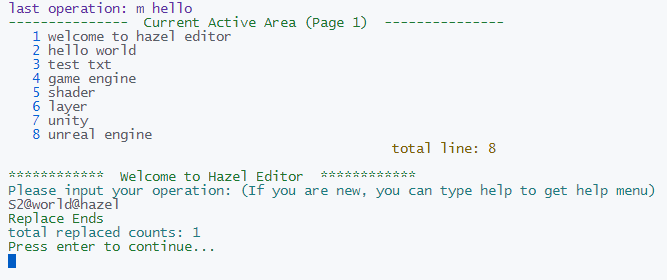
描述已自动生成

文本

描述已自动生成

图片包含 文本

描述已自动生成



文本

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

## 六、课程设计总结及心得体会

### 1. 心得体会

本次课设使我对链表的使用更加的娴熟，在看完这道题后的思路应该是这道题如果能用STL库，那么我会选择vector<string>来存储活区的文本内容，基于我之前自己实现的Vector和String，也就是使用动态数组来存储内容，就可以很轻易的写出这道题，但是我最终采用的是嵌套链表的形式来实现该编辑器。一来可以尝试新方式来实现这种vector<string>的结构，二来可以很好的复习链表的知识。

### 2. 遇到的问题

1. 在链表初始化过程中栽了不少跟头，于是写了一个CHECK\_AREA()函数用来判断是否已经初始化（然而用C++的类就不用担心这些小问题）。
2. 当输入文件中文本内容多于80行后，会把80行往后的文本读入非活区，在按下n将活区内容写入输出文件后，需要修正活区每一行的行号。在修正行号过程中，经过多次模拟尝试终于把修正错误的情况全部规避了。
3. 在将行块与字符数组的相互转换过程中，会用到清空行块的操作，而clearLineBlock函数由于传入参数没带上引用而导致只清空了函数内部的行块，而不是清空调用处的行块，从而导致没有正常删除该行，在调试的过程中发现该bug并及时修复。

### 3. 总结

经过这几天的不懈努力和调试，最终成果：该编辑器不仅能够支持需求中所有的功能，还额外新增了一些其他功能：

1. 新增帮助文档。键入help即可获取帮助文档。
2. 新增输入文件的切换。当一个输入文件中的内容全部处理完毕后，可以选择继续从其他文件中读入内容，而不用退出程序。命令：r <filename>。
3. 新增输出文件的切换。当需要把文本写入输出文件时，提供了更换输出文件的功能，用户可以随时更改输出文件；并且还增加了写入的行区间，用户可以自定义想要写到输出文件中的内容。命令：w <filename> <line\_number1> <line\_number2>。
4. 新增字符串匹配区间。用户可以自定义字符串匹配功能的匹配区间，而不是每次都匹配活区中的全部内容，命令：m <string> <line\_number1> <line\_number2>。
5. 新增字符串替换区间。用户可以自定义字符串替换功能的替换区间，而不是只能匹配某一行。命令：S<line\_number1>,<line\_number2>@<string1>@<string2>。
6. 新增退出编辑器功能。命令：exit。