

软件报告

第64组 PB19071472 王舒

一、需求分析

markdown 是一种方便快捷的文本格式，但很多网站只支持上传 html 格式的文件并显示。为了使用的方便，设计简单的软件，可以把现有的 markdown 文件生成对应的 html 文件，方便用户操作。

软件主要实现了 markdown 到 html 的基本语法实现，包括以下几个方面：

- text
 - 加粗
 - 斜体
 - 下划线
 - 删除线
- 标题 (1~6)
- 列表
 - 有序表
 - 无序表 (-, *两种)
- 图片
- 链接

生成好的 html 文件可以通过浏览器打开，显示与 markdown 预览相同的界面。用户通过输入需要转化的文件名以及输出文件名称，就可以通过运行该程序得到需要的结果。

二、概要设计

主要流程如下：

- 先把 markdown 文件读入，在读入的时候记录每个 \n 所在的位置。
- 然后把 \n 与 \n 之间的内容划分成一行，并对每行的特性（例如开头有 #、- 或 *）进行分析，然后把剩下的文本段部分储存起来。
- 对划分好的行，再把显示时没有换行的合并起来，划分成多个段，并对每一段的特征进行保存。
- 根据段的特征，为他们添加 html 语言中格式上的特性。对于文本内容，也通过某个函数对其进行处理，最终形成符合 html 语言规范的字符串。

- 把改好的内容写入新文件中，并通过系统调用在程序运行结束时用浏览器打开它。

三、详细设计

整个代码基于一个大的 class Trans，其中有很多函数以及其他的类。
对于概要中的几个主要流程，分别进行说明。

1. 读入

这里需要用到的函数和类有

```
string md_file; //存放原文
void input(string str); //读入原文并分析\n
line_breaks break_ins;
class line_breaks { //存放\n的信息
public:
    int cnt;
    int array[N];
};
```

- 读入的思路很简单，一个一个字符读入的过程中，如果遇到 \n，就在 break_ins.array[cnt] 中记录每一个 \n 的位置。
- 这里为了后续分析的方便，是从数组的[1]开始存的，而[0]处为-1。
代码如下：

```
void Trans::input(string file) {
    fstream fin(file.c_str(), ios::in);
    break_ins.cnt = 1;
    break_ins.array[0] = -1; //第一个\n是不存在的，从【1】开始存位置
    char c = fin.get();
    while (c != EOF) {
        md_file += c;
        if (c == '\n') {
            break_ins.array[break_ins.cnt] = md_file.length() - 1; //c在文本中的位置，\n占一个
            break_ins.cnt++;
        }
        c = fin.get();
    }
    md_file += "\n";
    break_ins.array[break_ins.cnt] = md_file.length() - 1;
    fin.close();
}
```

2. 行

用到的函数和类：

```
class md_lines {
public:
    string array[N]; // 除去\n和标识符的string
    int title[N]; // 数字表示#的个数
    int dash[N]; // 数字表示层级, 1, 2, 3, ...
    int number[N]; // 同上
    int star[N]; // 同上
    int blanks[N]; // 末尾是否有两个及以上的空格
    int lines_cnt;
};
md_lines md_lines;
void getRawLines();
```

函数对每一行进行分析，并填入 md_lines 中。本质上来说，这是一个字符串匹配的问题。

- 在这个函数中，首先需要去除所有的行首空格，并且记录其数量，方便列表时的缩进分析。
- 然后，对每一行来说，有这几种情况，分别进行讨论。（给出部分代码的实现）
 - 行尾有两个以上的空格

```
if (md_file[q] == ' ') {
    int cnt = 0;
    int rear = q;
    while ((rear >= p) && (md_file[rear] == ' ')) {
        cnt++;
        rear--;
    }
    if (cnt >= 2) {
        md_lines.blanks[md_lines.lines_cnt] = cnt;
    }
}
```

- 行首有多个#及一个空格

```

if (md_file[p] == '#') {
    while (p <= q && md_file[p] == '#') {
        p++;
        md_lines.title[md_lines.lines_cnt]++;
    }
    if (md_file[p] == ' ') { //匹配到标题行，直接把该行后面的内容写进去
        md_lines.array[md_lines.lines_cnt] = md_file.substr(p + 1, q - p);
        md_lines.lines_cnt++;
        continue;
    }
    else { //匹配失败，清空记录的#个数
        p -= md_lines.title[md_lines.lines_cnt];
        md_lines.title[md_lines.lines_cnt] = 0;
        md_lines.array[md_lines.lines_cnt] = md_file.substr(p, q - p + 1);
        md_lines.lines_cnt++;
    }
    continue;
}
}

```

- 行首有 - (或 *) 及一个空格
- 行首是 x . 及一个空格 (x 代表自然数)
- 如果这几种情况都不满足，就说明是普普通通的一行，直接处理。
- 这里还需要注意到的是，如果匹配失败，一些参数需要被修改，例如#的计数。此时，也把这一行作为普普通通的一行来看待。

3. 段

根据已经划分好的行，逐行分析段的构成，并把每一段的内容连接起来，转化成 html 的格式。
用到的函数和类：

```

class paragraph { //markdown中的一段，相当于HTML中的一行
public:
    int start; //起始行
    int end; //结束行
    int title; //数字表示#个数
    int olist; //数字表示层级
    int uolist; //同上
    int blanks;
    string str; //去掉特殊字符的内容部分，并转换为HTML
    paragraph() {
        start = end = title = olist = uolist = blanks = 0;
    }
};

paragraph par[N];
int par_cnt; //类似于栈顶指针的计数器
void getParagraph(); //确定每段拥有的行
void getParagraphString(); //得到转换好的段内文本
string solve(string str); //文本转换

```

◦ void getParagraph()

思路大致上是扫描所有的行，用 pre 记录一段的起始行，找到可以结束的标志，算作找到了新的一段。

可能产生新的一段的情况有以下几种（其中展示部分代码）

- 标题行。此时它本身就是一段，同时划分清楚了上一段的结束和下一段的开始

```
if (md_lines.title[i] != 0) {
    //把上一段放进数组里
    if (pre <= i - 1) { //之前有多行，连接
        par[par_cnt].start = pre;
        par[par_cnt].end = i - 1;
        par_cnt++;
    }
    //else时，pre==i，即上面已经处理好了
    //把这一段也放进数组里，因为标题段是确定的
    par[par_cnt].start = i;
    par[par_cnt].end = i;
    par[par_cnt].title = md_lines.title[i];
    par_cnt++;
    pre = i + 1;
}
```

- 无序/有序列表。此时划分了上一段的结束，该行是下一段的开始
- 行尾空格。该行是上一段的结束，同时划分了下一段的开始。
- 空行。该行划分了前后两段。

◦ string solve(string str)

该函数对各种文本格式（text，链接，图片）进行转换。采用递归的思路，把一段文本可以划分为文本+标志字符+文本的形式，即 str1+"xx"+str1+"xx"+str3 ...。因此，只要识别出标志字符，对剩下的子串做同样的处理即可。下面是最复杂的图片类型的处理：

```
if ((pos1 = str.find("![")) != string::npos) {
    string substr1 = str.substr(0, pos1);
    string substr2 = str.substr(pos1 + 2);
    if ((pos2 = substr2.find("](")) != string::npos) {
        pos2 += pos1 + 1;
        string substr22 = str.substr(pos2 + 2);
        if ((pos3 = substr22.find(")")) != string::npos) { //匹配全成功
            pos3 += pos2 + 1;
            substr2 = str.substr(pos1 + 2, pos2 - pos1 - 1);
            string substr3 = str.substr(pos3 + 2);
            string link = str.substr(pos2 + 3, pos3 - pos2 - 2);
            return solve(substr1 + "<img src=\"" + link + "\"" alt=\"" + substr2 + "\">");
        }
    }
}
```

其中，用到了一些 string 类的函数，这样使得代码更简单了一点。

- void getParagraphString()

调用 solve 函数，根据得到的每段拥有的行，就可以拼接起来，存入 par[i] 中。

4. 组合

主要思路是按照标题行的分布，把每两个标题行之间的段拿出来单独处理，来进行列表的排版。用到的主要的函数和类如下

```
string getHTMLLines();//得到全文的html表示
string title(int i);//处理标题
string list(int m, int n); //处理标题间的段
```

- string list(int m, int n)
 - 在 html 中，list 也是可以嵌套的。这样，就可以根据 list 的层次深度，把相应的文本进行嵌套，输出相应的 html 代码。此外，还要考虑到在 markdown 中，有时会出现没有分段符号但同样缩进的情况。
 - 最终采用的算法思路大致上是：把每一行看成一个节点，用链表来维护。每次对比层次深度和段的类型，进行合并。具体来说，就是高层次的段先合并，看做一个大的段，但深度减少。然后，再与其左边的同层次段合并。
 - 该函数实现的结果是，混合类型的列表也可以处理。
 - 节点定义，其中每个节点都有三种可能的类型。

```

#define NODE_TYPE_ORDERED 1
#define NODE_TYPE_UNORDERED 2
#define NODE_TYPE_COMMON 3
#define NODE_TYPE_SUBLIST 4
struct llnode {
    llnode* next, * pre;
    string str;
    int level;
    int type;//NODE_TYPE_XXXXXX
    llnode(paragraph* input) {
        next = pre = NULL;
        str = input->str;
        if (input->olist != 0) {
            type = NODE_TYPE_ORDERED;
            level = input->olist;
        }
        else if(input->uolist!=0){
            type = NODE_TYPE_UNORDERED;
            level = input->uolist;
        }
        else {
            type = NODE_TYPE_COMMON;
            level = 0;
        }
    }
    llnode() {
        next = pre = NULL;
        str = "";
        level = 0;
        type = 0;
    }
};

```

- 链表类：其中需要实现添加节点、合并节点以及实现 list 嵌套。

```

class llist {
public:
    llnode head, tail;
    int count;
    int count_node() {
        llnode* ptr = head.next;
        int count = 0;
        while (ptr != &tail) {
            count++;
            ptr = ptr->next;
        }
        return count;
    }
    llist() {
        head.pre = NULL;
        head.next = &tail;

        tail.pre = &head;
        tail.next = NULL;

        count = 0;
    }
    void add_node(paragraph *input) {
        llnode* newnode = new llnode(input);
        newnode->pre = tail.pre;
        newnode->next = &tail;
        tail.pre->next = newnode;
        tail.pre = newnode;
        count++;
    }
    void merge(llnode* ptr1, llnode* ptr2, int type) {
        string ans;
        llnode* ptr = ptr1;
        while (ptr != ptr2) {
            if (ptr->type == NODE_TYPE_ORDERED || ptr->type == NODE_TYPE_UNORDERED) {
                ans += "<li>" + ptr->str + "</li>\n";
            }
            else if(ptr->type==NODE_TYPE_COMMON){
                ans += ptr->str+"<br>\n";
            }
            else {
                ans += ptr->str + "\n";
            }
            ptr = ptr->next;
        }
        if (type == NODE_TYPE_ORDERED) {
            ans = "<ol>\n" + ans + "\n</ol>\n";
        }
        else if(type==NODE_TYPE_UNORDERED){
            ans = "<ul>\n" + ans + "\n</ul>\n";
        }
    }
}

```



```

else {
    ans =ans + "\n";
}

ptr1->next = ptr2;
ptr = ptr2->pre;
ptr1->level = ptr1->pre->level;
ptr1->str = ans;
ptr1->type = NODE_TYPE_SUBLIST;
while (ptr != ptr1) {
    llnode* tmp = ptr;
    ptr = ptr->pre;
    delete(tmp);
    count--;
}
ptr2->pre = ptr1;
}
string solve_list() {
    while (count_node() > 1) {
        llnode* p = head.next;
        //do merge

        //get maxlevel
        int maxlevel = 0;
        llnode* ptr = head.next;
        while (ptr != &tail) {
            if (ptr->level > maxlevel) {
                maxlevel = ptr->level;
            }
            ptr = ptr->next;
        }
        //find node with maxlevel

        ptr = head.next;
        while (ptr != &tail) {
            if (ptr->level == maxlevel) {
                llnode *ptr2 = ptr->next;

                int type = ptr->type;
                while ((ptr2->level == maxlevel && (ptr2->type==type || ptr2->type==N
                    ptr2 = ptr2->next;
                }
                merge(ptr, ptr2,type);
            }
            ptr = ptr->next;
        }

    }
    return head.next->str;
}

```

```
    }  
};
```

- 然后，在 `list` 函数中，就可以先把行作为节点添加到链表里，然后由 `solve_list()` 函数得到结果。

- `string getHTMLLines()`

这里的算法大致思路是，找到标题行并用 `title()` 进行处理，然后找到下一个标题行，对中间的段用 `list()` 处理。重复这个过程，最后拼接到一起。

考虑特殊情况，即第一个标题行不在文章开头；最后一个标题行之后还有内容。这两种情况单独处理。

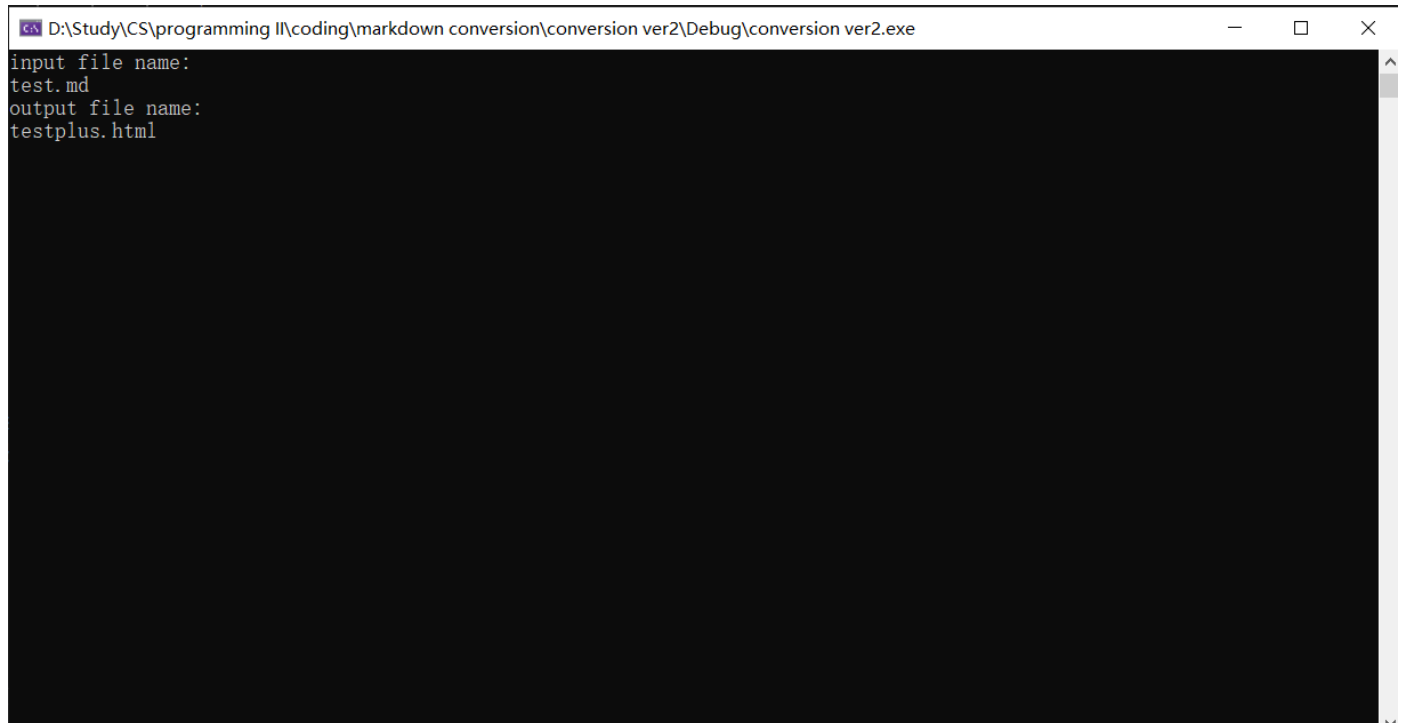
5. 输出

在 `output` 函数中，向文件输出必要的 `html` 头尾部以及转换好的内容即可。这之后采用了一个系统调用 `cmd`，可以直接打开浏览器，预览写好的文件。

四、用户使用说明和c++源码说明

- 用户可以通过控制台编译并运行核心代码。输入 `markdown` 文件名和 `html` 文件名后，就可以得到对应的文件，并自动打开浏览器预览。
- 文件结构比较简单，生成的文件位置与原文件在同一目录下，对用户的要求是可以用 `g++` 进行编译。
- 测试结果：

采用助教提供的测试文件，如下



```
D:\Study\CS\programming ll\coding\markdown conversion\conversion ver2\Debug\conversion ver2.exe  
input file name:  
test.md  
output file name:  
testplus.html
```

测试-综合

标题测试

一个3级标题

一个h3标题

文本测试

斜体1, 斜体2, **粗体1**, **粗体2**, 下划线, ~~删除线~~

粗体加斜体1, ~~删除线加上下划线1~~

列表测试

有序列表:

1. 有序1
2. 有序2
3. 有序3

无序列表:

- 无序1
- 无序2
- 无序3

还有一种无序列表:

- 无序2
- 无序3

混合列表:

- 层次1
 - 层次1.1
 - 层次1.1.1
 - 层次1.1.2
 - 层次1.1.3
 - 层次1.2
 - ~~层次1.3~~
- 层次2
- 层次3
 - **层次3.1**
 - 层次3.1.1
 - 层次3.1.1.1

1. 层次1
 1. 层次1.1
 2. 层次1.2
 3. 层次1.3
 1. 层次1.3.1
 2. 层次1.3.2
 1. 层次1.3.2.6
2. 层次2

图片与链接测试



[USTC导航](#) 和 [百度](#)

与原来的 markdown 文件预览结果相同。

五、分工及实验感想

- 本次实验是我一个人完成的，时间也比较紧张，虽然代码量不算特别大，但debug等时间加起来还是费了很多功夫。期间，也查阅了许多资料，对于如何下手完成这个任务做了很多思考。
- 开始的时候也在github上面看过别人写的代码，但是一方面是水平不足，另一方面是他写的东西没有注释，最后还是自己琢磨了一下，决定采用现在的流程来写。这可能不是效率最高的，也有着许多漏洞，但写完还是很有成就感。
- 在写的过程中，首先是许多函数根本不知道怎么用（甚至根本不知道有这么个函数），都是需要查阅很多东西，加上自己的一些实践，跌跌撞撞，竟然完成了。其次，对于字符串的分析其实是一个很痛苦的过程，它非常精确，一位都不能出错，情况又很多很复杂，debug经常令人头秃。还记得分段的时候，好不容易写完了，测出来完全不对，debug也找不到问题，最后只能换一种写法重来一次。不过，或许我的debug能力也得到了提高。
- 逻辑在写代码的过程中是非常关键的存在，如果逻辑比较混乱、不全面，或者混杂，都会导致巨大的痛苦，而且很难找出错误。此外，我也意识到了更好的解决方案往往是按照功能，把各种任务划分成小的函数，而不是混在一起。这样逻辑上清晰一点，也更利于模块化。