文字编辑器开发文档

1 需求分析

1.1 文本编辑功能

- **输入：**用户可以通过键盘在文本编辑器中进行文本输入，可以输入各种字符，包括数字、字母、汉字，标点符号和特殊字符等，以适应不同用户的文本编辑需求。用户在键盘上长按某个键时，能够自动重复输入相同的字符或符号，从而提高输入效率。用户可以通过键盘或鼠标来移动光标的位置，以便精确输入或编辑文本。用户还可以用鼠标或键盘将文本选中，以便进行复制、剪切或删除等操作。并且，文本编辑器应该支持快捷键输入，即设置一些热键，如Ctrl+C、Ctrl+V等，以便更加快速地进行文本输入和编辑。
- **插入：**用户可以在文本中任意位置插入新的字符或文本，支持多种插入方式，例如手动输入，粘贴板复制、拖放等。这可以用于添加新的文本、更正拼写或改变文本样式等任务。
- **删除：**用户可以使用删除键（Backspace或Delete）将光标前后的字符或文本删除，支持多种删除方式，如删除单个字符、单词、行或段落等，还可以用鼠标选择性地删除一部分内容，方便用户快速地删除不需要的内容，提高文档准确性和效率。
- **复制：**用户可以在文本编辑器中选择需要复制的文本内容，复制功能应该提供多种复制方式，例如：复制选中区域、复制整行、复制整段等。复制操作应该将选中内容复制到系统的剪切板中，以支持用户在其他环境中直接粘贴。为了提高用户的编辑效率，复制功能应该提供相应的快捷键，例如 Ctrl+C。
- **粘贴：**程序应该支持多种粘贴方式，例如：粘贴到光标处、粘贴到选中区域、粘贴纯文本或格式化文本等。程序应该自动适应粘贴板中的内容格式，并进行相应的转换，避免因格式不匹配而产生其他问题。为了提高用户编辑效率，粘贴功能应该提供相应的快捷键，例如 Ctrl+V。
- **剪切：**用户可以在文本编辑器中选择需要剪切的文本内容，剪切功能应该提供多种剪切方式，例如：剪切选中区域、剪切整行、剪切整段等。剪切操作应该将选中内容保存到系统的剪切板中，以支持用户在其他环境中直接粘贴。为了提高用户的编辑效率，剪切功能应该提供相应的快捷键，例如 Ctrl+X。
- **全选：**用户可以在文本编辑器中使用全选功能，将编辑框中的所有文本进行选定操作，方便用户进行批量操作。为了提高用户的编辑效率，全选功能应该提供相应的快捷键，例如 Ctrl+A。如果用户不需要选择全部文本，程序应该允许用户取消全选状态，避免误操作。
- **搜索/替换：**考虑输入查找内容和替换内容、从光标位置逐个查找或替换相应内容、一次性全部替换掉相应内容等基本功能。同时，需要从工具栏或应用下拉菜单中提供该功能，以方便用户的查找和替换操作。为了进一步提高编辑器的易用性和效率，可以考虑增加一些优化功能，如自动选中匹配文本、范围设置、正则表达式和大小写敏感选项、替换结果的统计信息等。

- **撤销/重做：**通过"编辑"下拉菜单或右键单击菜单执行，并应记录所有编辑操作以供后续使用，可以作用于整个文档或选定的文本区域，还可以通过显示编辑历史记录让用户了解操作的顺序。快捷键等可快速执行撤销/重做操作的方式也应该被提供。

1.2 文件功能

- **新建：**创建一篇空白文档，在“工具栏”或“文件下拉菜单”点击“新建”按钮后，若原文档未保存，程序将弹出保存窗口提醒用户保存，用户保存已有文档后窗口内容将被清空。
- **打开：**能够打开文本（.txt）文件，点击“工具栏”或“文件下拉菜单”中的打开按钮，将弹出选择框，用户点击需要打开的文件就能在文本编辑器中打开该文件并显示文件内容。当用户打开一个文件时，程序可以显示该文件一些的属性，例如文件名、文件路径。
- **保存：**实现保存文档，点击“工具栏”或“文件下拉菜单”中的保存按钮，将弹出选择框，用户可设置文件名，文件类型和保存路径。
- **另存为：**用户可以在保存文件时，将修改后的文档保存成为一个新的文件名或新的路径，而不会影响原来的文件，另存为功能应该提供让用户选择保存路径和文件名称的选项，可以通过“工具栏”或“文件下拉菜单”中执行，并且用户可以保存不同格式的文档文件。

1.3 文本格式设置

- **字体大小：**应该支持多种字号设置，例如：8号字、10号字、12号字等。在设置字号的过程中，程序应该实时显示当前选择的效果，方便用户实时调整。为了满足用户个性化需求，程序应该允许用户自定义设置字号。
- **字体样式：**支持多种字体样式，例如：宋体、黑体、微软雅黑等，还应该支持多种字形设置，例如：粗体、斜体、下划线等。在设置字体样式的过程中，程序应该实时显示当前选择的效果，方便用户实时调整。为了提高用户的编辑效率，字体样式功能应该提供相应的快捷设置方式，例如在工具栏中提供选项。
- **颜色：**支持多种颜色设置，除了程序提供的预设颜色外，字体颜色功能还应该允许用户自定义设置颜色。在设置颜色的过程中，程序应该实时显示当前选择的效果，方便用户实时调整。字体颜色功能应该提供相应的快捷设置方式，例如在工具栏中提供选项。

1.4 排版设置

- **对齐方式：**支持多种对齐方式，例如左对齐、居中对齐、右对齐等。为了提高用户的编辑效率，对齐方式功能应该提供相应的快捷设置方式，例如在工具栏中提供选项。在设置对齐方式的过程中，程序应该实时显示当前选择的效果，方便用户实时调整。除了单独设置一行文本的对齐方式外，对齐方式功能还应该允许用户同时设置多行文本的对齐方式。

2 总体设计

2.1 设计思路

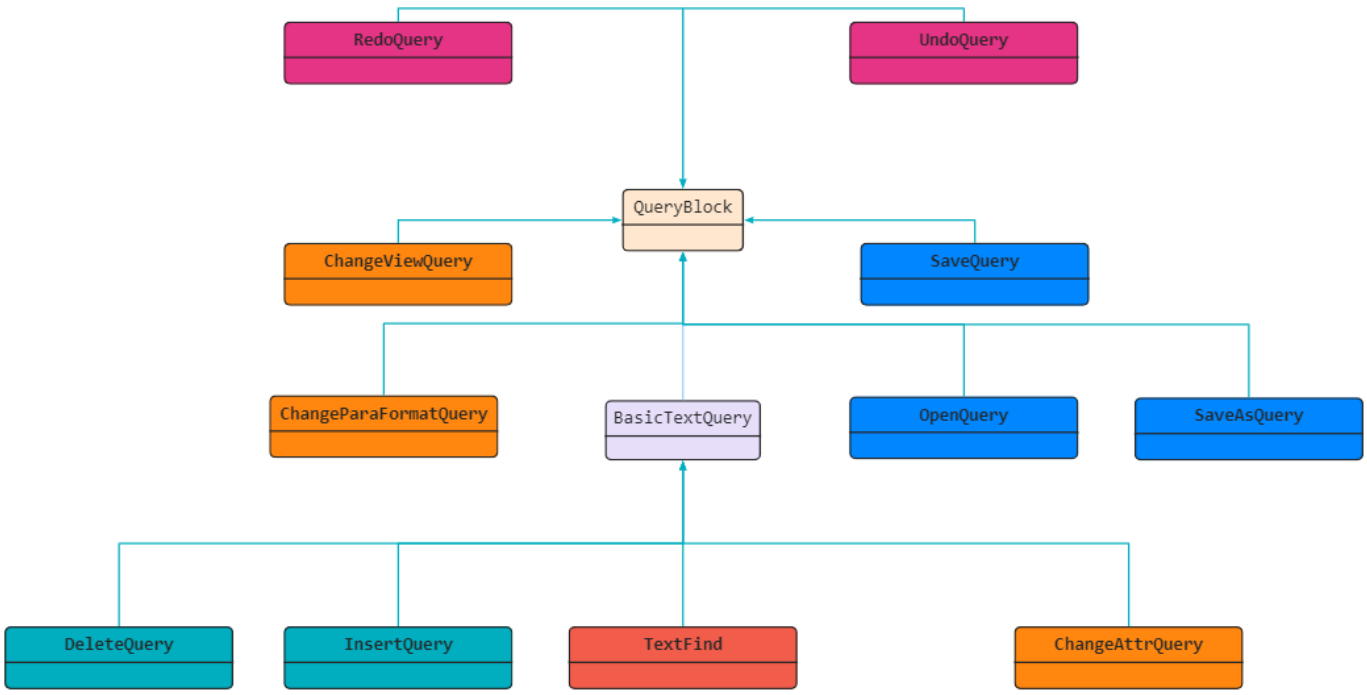
文本编辑器软件的实现分为前端和后端两部分。前端负责直接与用户交互的部分，采用Qt6实现可视化图形界面，以提供便于用户操作的界面。前端需要精确地获取并理解用户的需求，并根据需求调用相应的后端功能，以完成对文本的处理。而后端部分则主要使用自定义的函数及部分Qt接口完成对文字的处理，设计一个便于使用、功能强大的文字处理函数库接口，供前端调用。

具体来说，前端需要实现文本框、菜单栏、工具栏等组件，以及相应的事件处理函数。在用户进行操作时，前端会通过前后端接口向后端发送指令，后端接口根据接收到的指令调用相应的函数进行文本操作，并将处理结果返回给前端。后端需要实现文本处理函数，如获取文本、查找文本、更改属性、插入文本、删除文本、撤销和重做等。同时，还需要对文本进行格式化处理，比如段落格式、字体样式等。为了提高效率，后端还需要实现一些文本缓存、文件读写等功能。同时，为了确保数据的安全性，后端还需要实现类似版本控制的功能，比如记录编辑历史、恢复到指定版本等。

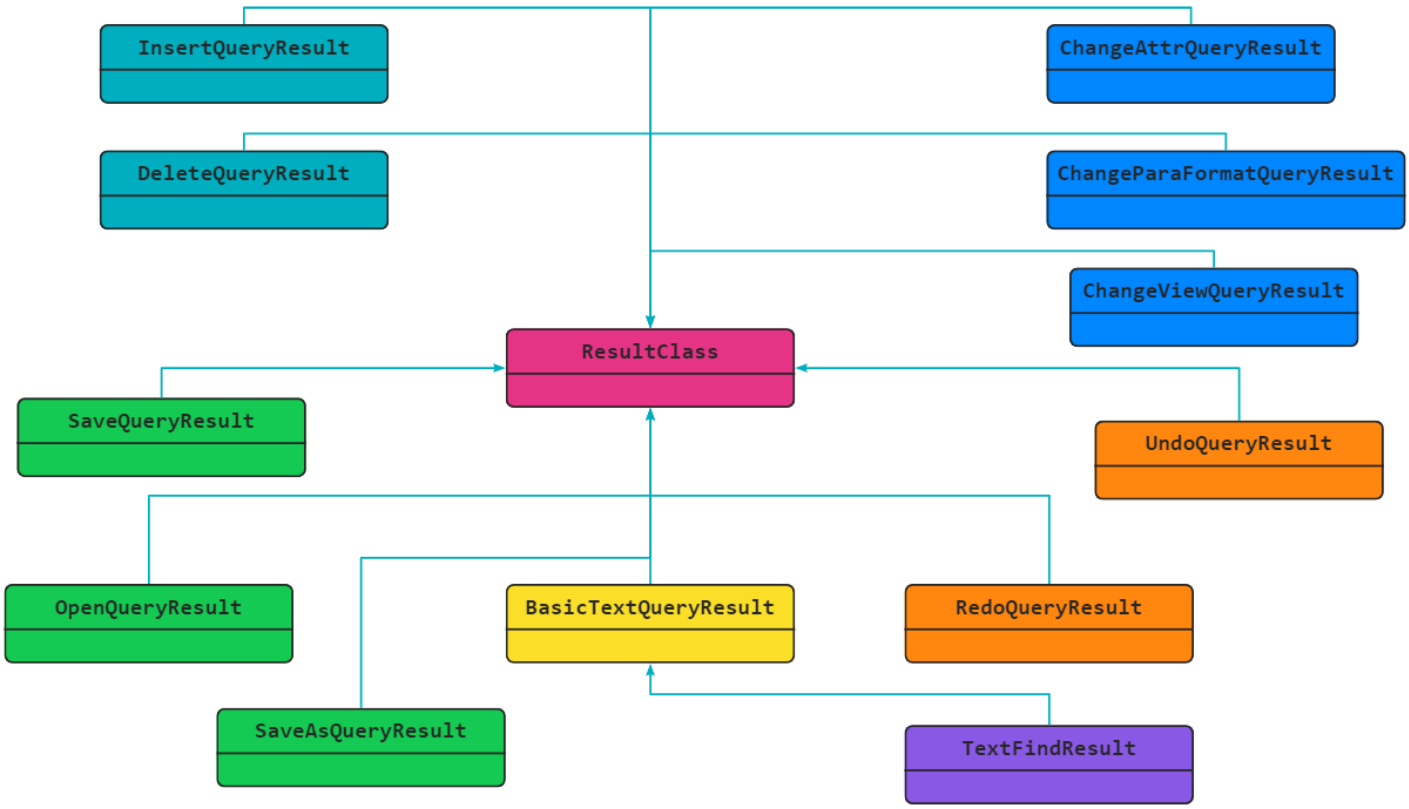
通过前后端接口建立联系，前端可以通过接口向后端发送指令，并获取后端返回的数据来实现相应的操作。整个编辑器的设计可以灵活扩展和优化，适应不同的需求和场景。

2.2 类关系图

2.2.1 查询类关系图



2.2.2 查询结果类关系图



3 系统模块说明

3.1 数据结构模块

3.1.1 WordData类

WordData类被用作存储文本单词的基本单元。用户在编辑文本时，输入的文本内容会被分割成一个个单词，每个单词都会被存储为一个WordData类的对象。

3.1.1.1 主要作用

（1）存储文本单词的内容和格式信息

WordData类中包含了单词的内容和格式信息，例如字体、字号、颜色等等，这些信息可以用于在文本编辑器中显示和处理文本。

（2）支持文本单词的插入、删除和修改

在文本编辑器中，用户可以对文本进行插入、删除和修改等操作，这些操作通常都是针对单个单词或者一段连续的单词进行的。WordData类提供了一系列方法，帮助用户实现对单词的快速插入、删除和修改。

（3）作为其他高级数据结构的基础

在文本编辑器中，WordData类通常会和其他高级数据结构一起使用，例如字符串、文本块、行、页等等。这些数据结构都是由多个WordData对象组成的，因此WordData类可以看作是这些高级数据结构的基础单元。

3.1.1.2 成员变量

```
//单词的颜色
uint32_t color;

//单词的风格，可以是BOLD、ITALIC、UNDERLINE
uint32_t style;

//单词的字体大小
int fontSize;

//单词所使用的字体
QString font;

//单词的长度
Index length;

//单词的具体内容
QChar data[MAXSIZE+10];
```

3.1.1.3 方法

```
//复制当前WordData对象的格式信息并创建一个新的WordData对象。
WordData* CloneFormat();

//将当前WordData对象拆分为两个新的WordData对象。
WordData* Split();

//在当前WordData对象之前合并other WordData对象。
bool MergeToFront(WordData &other);

//在给定索引处将当前WordData对象拆分为两个新的WordData对象。
WordData* Split(Index index);

//返回指向当前WordData对象的数据数组的指针
QChar* GetData();

//获取当前WordData对象的颜色信息
uint32_t GetColor();

//获取当前WordData对象的样式信息
uint32_t GetStyle();

//获取当前WordData对象的字号信息
```

```
23 int GetFontSize();
24
25 //获取当前WordData对象的字体信息
26 QString GetFont()
27
28 //获取当前WordData对象的长度。
29 Index GetLength();
30
31 //设置当前WordData对象的颜色。
32 void SetColor(uint32_t newColor);
33
34 //设置当前WordData对象的样式。
35 void SetStyle(Styles newStyle);
36
37 //设置当前WordData对象的字号。
38 void SetFontSize(int newFontSize);
39
40 //设置当前WordData对象的字体。
41 void SetFont(QString &newFont);
42
43 //设置当前WordData对象的长度。
44 void SetLength(int newLength);
45
46 //在给定索引处插入一个新的字符串，
47 //并将所有插入的单词的WordData对象存储到一个vector容器中，返回该容器的指针
48 std::shared_ptr<std::vector<WordData *>> InsertData(QString &&str, Index
   beginInd);
49
50 //判断另外一个WordData对象的格式是否和当前WordData对象的格式相同。
51 bool FormatEqual(const WordData &other) const;
```

3.1.2 TopData类

3.1.2.1 主要作用

TopData类扮演着存储文档内容及格式信息的角色。

在TopData类中，使用一个二维数组（data）来存储整个文档的字符数据，而每个元素（即每个位置）都对应着一个WordData对象。因此，TopData类中对文档进行插入、删除、查询等操作时，实际上就是对data数组中各个WordData对象进行操作。

同时，在TopData类中也会涉及到“块”的概念，即将文本分割为多个独立的块（通常基于段落或表格），并且每个块都由多个WordData对象组成。在这种情况下，TopData类可以通过indexOfBlocks数组来记录每个块在哪些位置上使用了哪些WordData对象。

TopData类还提供了插入、删除、查询、修改、格式化等操作，使得文本编辑器可以实现各种编辑操作。同时，TopData类也保证了编辑器的稳定性，支持了撤销/重做功能。

3.1.2.2 成员变量

```
//存储文档中每个字符的 WordData 对象。
std::vector<std::vector<WordData *>> data;

//记录每个块在哪些位置上使用了哪些WordData对象。这样可以快速定位到指定位置的字符。
std::vector<std::vector<Index>> indexOfBlocks;

//存储每个段落的格式信息。
std::vector<ParaFormat> paraFormats;
```

3.1.2.3 私有方法

```
// 检查给定的BlockPosition或Position是否在有效范围内。
bool CheckRange(const BlockPosition& position) const;

// 根据起始和结束位置，将一个块分裂成两个。
void SplitFrom(const Position& position, int index);

// 删除一个块，同时更新 indexOfBlocks 和 data 向量。
void DeleteBlock(int blockIndex);

// 与 DeleteBlock() 类似，但是不更新indexOfBlocks和data向量。
void SimpleDeleteBlock(int blockIndex);

// 根据给定位置，将当前块分裂成两个。
void SplitFrom(const Position& position);

// 根据块的行列号获取对应的WordData对象。
WordData* GetBlockFromBlockIndex(const BlockPosition& position);

// 查找给定Position所在的块的位置。
BlockPosition Find(const Position& position) const;

```

3.1.2.4 公有方法

```
// 将一段字符串插入到文档中指定位置，并返回插入的最后一个字符的位置。
Position Insert(const Position& position, const std::string& text, const
ChangeAttrQuery& query);

// 根据块的行列号获取对应的WordData对象。
```

```
5  WordData* GetBlockFromBlockIndex(const BlockPosition& position);
6
7  // 获取指定行（索引）的所有 WordData 对象。
8  std::vector<WordData*> Get(int lineIndex) const;
9
10 // 在指定的范围内查找第一个匹配给定文本的位置。
11 Position FindText(const std::string& text, const Position& startPosition, const
    Position& endPosition);
12
13 // 获取指定块的起始列
14 int GetBlockStartColumn(int blockIndex) const;
15
16 // 获取指定块的下一个块的位置。
17 BlockPosition GetNextBlockPosition(const BlockPosition& current) const;
18
19 // 获取指定块的前一个块的位置。
20 BlockPosition GetFrontBlockPosition(const BlockPosition& current) const;
21
22 // 获取指定位置的下一个位置。
23 Position GetNextPosition(const Position& current) const;
24
25 // 根据给定的 ChangeAttrQuery 对象，修改给定范围内所有字符的颜色、样式、字体等属性。
26 void ChangeAttr(const ChangeAttrQuery& query, const Position& start, const
   Position& end);
27
28 // 检查指定的Position或BlockPosition是否超出了合法范围。
29 bool CheckOutRange(const Position& position) const;
30
31 // 删除指定范围内的字符。
32 void Delete(const Position& start, const Position& end, const ChangeAttrQuery&
   query);
33
34 // 获取指定行（索引）的段落格式信息。
35 ParaFormat* GetParaFormat(int lineIndex);
36
37 // 修改指定行（索引）的段落格式信息。
38 void ChangeParaFormat(int lineIndex, const ParaFormat& format);
39
40 // 根据索引查找应该插入的位置。
41 Position findInsert(int index) const;
42
43 // 根据索引查找应该删除的位置。
44 Position findDelete(int index);
```

3.1.3 QueryBlock类

3.1.3.1 主要作用

提供一个通用的查询块类，方便其他查询块类进行继承和扩展。它包含一个查询 ID（queryID）作为标识符，并提供了获取查询 ID 的接口。此外，它还包含了一个 merge 函数，用于合并两个查询块，但在基类中它只做了基本的抛出异常操作，并不能真正实现合并功能。

在实际应用中，可以根据不同的需求创建不同的查询块类，并在其中实现具体的查询逻辑。通过继承 QueryBlock 类，可以使得具体的查询块类具备共同的属性和方法，避免了重复编写代码的问题。

3.1.3.2 成员变量

```
1 //表示查询块的 ID
2 Instruction queryID;
```

3.1.3.3 公有方法

```
1 //获取查询块的 ID
2 Instruction GetQueryID();
3
4 //销毁 QueryBlock 对象
5 virtual ~QueryBlock() = default;
6
7 //将两个查询块合并
8 virtual bool merge([[maybe_unused]]QueryBlock *other)
```

3.2 文本编辑模块

3.2.1 ResultClass类

3.2.1.1 主要作用

ResultClass类是一个查询结果基类，用于表示某个查询操作的返回结果。具体的查询操作可以继承该类，并通过实现其虚函数来自定义查询结果处理的过程。通过继承ResultClass类，可以避免重复编写相同的代码，并且提高了程序的可维护性和可扩展性。

3.2.1.2 成员变量

```
1 protected:
2     //表示查询结果是否合法
3     bool isOK;
```

```
    //表示错误信息
    std::string msg;
private:
    //表示查询块对象的指针
    std::shared_ptr<QueryBlock> query;
```

3.2.1.3 公有方法

```
    // 获取查询结果是否合法的标志位
    bool GetIsOK();

    // 设置查询结果是否合法的标志位
    void SetIsOK(bool newOK);

    // 虚析构函数
    virtual ~ResultClass() = default;
```

3.2.2 基本文本查询

3.2.2.1 BasicTextQuery类

3.2.2.1.1 主要作用

表示文本查询块，即可以定义在文本中的一个区域，并且它具有以下功能：

- 继承自 `QueryBlock`，因此可以使用 `QueryBlock` 中的成员函数。
- 记录查询区域的开始位置和结束位置，以及所在块的起始位置和终止位置。
- 提供获取查询区域位置和块位置的函数，方便后续操作。
- 提供设置查询区域位置和块位置的函数，方便对查询块进行更新。
- 提供 `merge()` 函数，用于将两个相邻的查询块合并成一个大的查询块。

3.2.2.1.2 成员变量

```
protected:
    Position begin, end;    // 查询块的起始位置和终止位置
    BlockPosition beginBlock, endBlock;    // 查询块所在的文本块的起始位置和终止位置
```

3.2.2.1.3 公有方法

```
    // 获取查询区域位置和块位置的函数
```

```
    Index GetBeginRow();                // 返回查询区域的起始行
    Index GetBeginColumn();             // 返回查询区域的起始列
    Index GetEndRow();                  // 返回查询区域的终止行
    Index GetEndColumn();               // 返回查询区域的终止列
    Position GetBeginPos();             // 返回查询区域的起始位置
    Position GetEndPos();               // 返回查询区域的终止位置
    BlockPosition GetBeginBlockPos();   // 返回查询块所在文本块的起始位置
    BlockPosition GetEndBlockPos();     // 返回查询块所在文本块的终止位置

    // 设置查询区域位置和块位置的函数
    void SetBeginPos(Position pos);     // 设置查询区域的起始位置
    void SetEndPos(Position pos);       // 设置查询区域的终止位置
    void SetBeginBlockPos(BlockPosition pos);   // 设置查询块所在文本块的起始位置
    void SetEndBlockPos(BlockPosition pos);     // 设置查询块所在文本块的终止位置

    // 将两个相邻的查询块合并成一个大的查询块
    bool merge(BasicTextQuery *other);
```

3.2.2.2 BasicTextQueryResult类

3.2.2.2.1 主要作用

`BasicTextQueryResult` 类是一个用于处理 `BasicTextQuery` 查询块的查询结果的类。它继承自 `ResultClass`，并包含了一些成员变量和成员函数，可以实现对查询结果的高亮、标记、分页等操作，以及便于用户定位和获取查询结果所在位置的功能。

3.2.2.2.2 成员变量

```
protected:
    WordData *now; // 当前查询结果所在的单词数据对象指针
    FrontBackInterface &interface = *(FrontBackInterface *)NULL;// 前后端接口对象的引用
```

3.2.2.2.3 公有方法

```
    // 将当前查询结果指针向下移动一位
    void Next();

    // 获取当前查询结果对应的单词数据对象指针
    WordData *Get();

    // 判断当前查询结果是否属于一个新段落
    bool IsNewPara();

```

```
    // 获取当前查询结果所在段落的索引
    Index GetParaIndex();

    // 获取查询块对象的指针
    virtual BasicTextQuery *GetQueryBlock();
```

3.2.3 文本插入

3.2.3.1 InsertQuery类

该类表示一个插入文本的查询块，它从 `BasicTextQuery` 类派生而来，继承了基础文本查询块类的成员函数和数据成员。

该类中包含以下成员：

- `data`：一个 `QString` 类型的成员变量，表示需要插入的字符串数据。
- 成员函数 `GetData()`：用于获取需要插入的字符串数据的引用。它返回一个 `QString&` 类型的引用，以便用户可以直接修改 `data` 成员变量的值。此函数是对外提供的一个访问该对象中私有字段的接口。

在使用该查询块对象时，用户可以通过调用 `GetData()` 函数获取需要插入的字符串内容，然后构造并发送给一个文本编辑器或其他程序，以便在指定的位置进行插入操作。

3.2.3.2 InsertQueryResult类

该类表示插入文本操作的查询结果。它继承了基础查询结果类 `ResultClass` 的成员函数和数据成员。

该类中包含以下成员：

- 成员函数 `GetQueryBlock()`：用于获取该查询结果所对应的查询块对象的指针。它返回一个 `InsertQuery*` 类型的指针，并使用 `dynamic_cast` 运算符将基类的指针类型强制转换为 `InsertQuery*` 类型。由于基类指针的实际类型可能不是 `InsertQuery` 类型，因此需要进行运行时类型检查。

在使用该查询结果类对象时，用户可以通过调用 `GetQueryBlock()` 函数获取与其对应的查询块对象指针，然后进一步分析查询结果。

3.2.4 文本删除

3.2.4.1 DeleteQuery类

该类表示一个删除文本的查询块。该类从 `BasicTextQuery` 类派生而来，继承了基础文本查询块类的成员函数和数据成员。

该类中包含以下成员：

- 构造函数 `DeleteQuery(Position begin, Position end)`：接受一个起始位置参数 `begin` 和一个结束位置参数 `end`，表示需要删除的文本范围。构造函数首先调用基类 `BasicTextQuery` 的构造函数，向其中传递三个参数：指令类型（`Instruction::DELETE`）、起始位置 `begin` 和结束位置 `end`。
- 成员函数 `GetBegin()` 和 `GetEnd()`：分别用于获取需要删除的文本的起始位置和结束位置。它们返回一个 `Position` 类型的值，用于直接访问 `BasicTextQuery` 基类中的相应成员变量。

3.2.4.2 DeleteQueryResult类

该类表示执行删除文本操作的查询结果。该类从 `ResultClass` 类派生而来，继承了基础查询结果类的成员函数和数据成员。

该类中包含以下成员：

- 构造函数 `DeleteQueryResult(bool isOK)`：接受一个布尔值参数 `isOK`，表示删除操作是否成功。构造函数首先调用 `ResultClass` 的构造函数，向其中传递两个参数：一个布尔值表示操作是否成功，以及一个空指针，表示删除操作所对应的查询块对象为空。
- 构造函数 `DeleteQueryResult(bool isOK, std::shared_ptr<DeleteQuery> query)`：接受一个布尔值参数 `isOK` 和一个查询块对象指针参数 `query`，表示删除操作是否成功和其所对应的查询块对象。构造函数首先使用 `std::reinterpret_pointer_cast` 函数将 `query` 指针转换为 `QueryBlock` 类型的指针，然后再调用 `ResultClass` 的构造函数，向其中传递两个参数：一个布尔值表示操作是否成功，以及转换后的指针对象。
- 成员函数 `GetQueryBlock()`：用于获取该查询结果所对应的查询块对象的指针。它返回一个 `DeleteQuery*` 类型的指针，并使用 `dynamic_cast` 运算符将基类指针的类型强制转换为 `DeleteQuery*` 类型。由于基类指针的实际类型可能不是 `DeleteQuery` 类型，因此需要进行运行时类型检查。

在使用该查询块对象和查询结果对象时，用户可以通过调用 `GetBegin()`、`GetEnd()`、`GetQueryBlock()` 等函数获取其相关信息，并根据具体情况执行删除操作或分析查询结果。

3.2.5 搜索替换

3.2.5.1 TextFind类

3.2.5.1.1 成员变量

```
1     const QString &m_query_word; // 输入的关键词
2     uint8_t control_bits; // 控制查找选项的值
3   //决定查找操作行为的三个控制属性：是否区分大小写、是否进行全字匹配、是否仅在选择区域内查找。每个枚举值使用2的幂次方表示，方便使用二进制位运算进行控制。
4     enum attr{m_is_case_sensitive=1,m_is_whole_word=2,m_is_find_in_selection=4};
```

3.2.5.1.2 公有方法

```
    // 获取输入的关键词
    const QString &GetWord();
    // 返回是否区分大小写的开关状态
    bool IsCaseSensitive();
    // 返回是否进行全字匹配的开关状态
    bool IsWholeWord();
    // 返回是否仅在选择区域内查找的开关状态
    bool IsFindInSel();
    // 返回存储控制选项的变量 control_bits 的值
    bool GetAttributes();
    // 静态函数：返回是否区分大小写的开关状态
    static bool IsCaseSensitive(uint8_t control_bits);
    // 静态函数：返回是否进行全字匹配的开关状态
    static bool IsWholeWord(uint8_t control_bits);
    // 静态函数：返回是否仅在选择区域内查找的开关状态
    static bool IsFindInSel(uint8_t control_bits);
```

3.2.5.2 TextFindResult类

该类继承自 `BasicTextQueryResult` 类，它用于表示文本查找操作的结果。该类包含以下两个主要成员：

- 构造函数 `TextFindResult()`：接受三个参数，分别为搜索操作是否成功的标识符、文本界面操作对象和查询操作对象。它使用这些参数来初始化基类 `BasicTextQueryResult` 中的数据成员。
- 成员函数 `GetNextIndex()`：该函数用于获取下一个匹配项在文本中的位置。

3.2.6 撤销和重做

定义了四个类，分别是 `UndoQuery`、`RedoQuery`、`UndoQueryResult` 和 `RedoQueryResult`。这些类用于表示撤销和重做操作的查询块和查询结果。

`UndoQuery` 类和 `RedoQuery` 类都继承自 `QueryBlock` 类，并分别代表撤销和重做操作。在构造函数内，它们都调用了基类 `QueryBlock` 的构造函数，并传递了一个参数 `Instruction::UNDO` 或 `Instruction::REDO` 表示具体的操作类型。

`UndoQueryResult` 类和 `RedoQueryResult` 类都继承自 `ResultClass` 类。它们都有两个构造函数：

- `UndoQueryResult(bool isOK, std::shared_ptr<UndoQuery> query)` 和 `RedoQueryResult(bool isOK, std::shared_ptr<RedoQuery> query)`：它们分别表示撤销和重做操作的查询结果。参数 `isOK` 表示操作是否成功，参数 `query` 则是一个指向相应查询块的指针。
- `UndoQueryResult(bool isOK)` 和 `RedoQueryResult(bool isOK)`：它们也表示撤销和重做操作的查询结果，但是不需要传递查询块对象，因为这些对象将使用 `std::make_shared<UndoQuery>` 或 `std::make_shared<RedoQuery>` 创建，并传递给基类 `ResultClass` 的构造函数进行初始化。

  这两个类都实现了虚函数 `GetQueryBlock()`，以返回相应的查询块对象的指针。

3.3 文件功能模块

3.3.1 文件打开

3.3.1.1 OpenQuery类

该类表示一个打开文件的查询块。它从 `QueryBlock` 类派生而来，继承了基础查询块类的成员函数和数据成员。

在 `OpenQuery` 类中，我们可以看到：

- 私有数据成员 `QString &filename`：表示需要打开的文件名。
- 构造函数 `OpenQuery(QString &filename)`：接受一个 `QString` 类型的引用参数 `filename`，表示需要打开的文件名。构造函数首先调用基类 `QueryBlock` 的构造函数，向其中传递一个参数：指令类型（`Instruction::OPENFILE`）。然后，在初始化列表中将 `filename` 引用赋值给相应的数据成员。

3.3.1.2 OpenQueryResult类

该类表示一个打开文件操作的查询结果。它从 `ResultClass` 类派生而来，继承了基础查询结果类的成员函数和数据成员。

在 `OpenQueryResult` 类中，我们可以看到：

- `protected` 成员函数 `OpenQuery *GetQueryBlock()`：用于获取该查询结果所对应的查询块对象的指针。该函数重载了基类函数，并首先使用 `assert()` 函数断言该函数永远不应该被调用。然后，该函数返回 `nullptr`，表示该查询结果所对应的查询块对象为空。
- 构造函数 `OpenQueryResult(bool isOK)`：接受一个布尔值参数 `isOK`，表示打开文件操作是否成功。构造函数首先调用基类 `ResultClass` 的构造函数，向其中传递两个参数：一个布尔值表示操作是否成功，以及一个空的指针，表示该查询结果所对应的查询块对象为空。

3.3.2 文件保存

3.3.2.1 OpenQuery类

该类表示一个文件保存操作的查询块。它从 `QueryBlock` 类派生而来，继承了基础查询块类的成员函数和数据成员。

在 `SaveQuery` 类中，我们可以看到：

- 构造函数 `SaveQuery()`：调用基类 `QueryBlock` 的构造函数，向其中传递一个参数：指令类型（`Instruction::SAVEFILE`），表示该查询块的动作是进行文件保存。

3.3.2.2 OpenQueryResult类

该类表示一个文件保存操作的查询结果。该类从 `ResultClass` 类派生而来，继承了基础查询结果类的成员函数和数据成员。

在 `SaveQueryResult` 类中，我们可以看到：

- 私有成员函数 `SaveQuery *GetQueryBlock()`：用于获取该查询结果所对应的查询块对象的指针。该函数重载了基类函数，并首先使用 `assert()` 函数断言该函数永远不应该被调用。然后，该函数返回通过 `reinterpret_cast` 转换后的 `SaveQuery` 对象指针，表示该查询结果所对应的查询块对象是一个 `SaveQuery` 对象。
- 两个构造函数：
  - `SaveQueryResult(bool isOK)`：接受一个布尔值参数 `isOK`，表示文件保存操作是否成功。构造函数首先调用基类 `ResultClass` 的构造函数，向其中传递两个参数：一个布尔值表示操作是否成功，以及一个空指针，表示该查询结果所对应的查询块对象为空。
  - `SaveQueryResult(bool isOK, std::shared_ptr<SaveQuery> query)`：接受两个参数：一个布尔值参数 `isOK`，表示文件保存操作是否成功，以及一个智能指针参数，表示该查询结果所对应的查询块对象是一个 `SaveQuery` 指针。该构造函数首先调用基类 `ResultClass` 的构造函数，向其中传递两个参数：一个布尔值表示操作是否成功，以及一个通过 `reinterpret_cast` 转换后的 `QueryBlock` 指针，表示该查询结果所对应的查询块对象是一个 `SaveQuery` 对象指针。

3.3.2 文件另存为

3.3.2.1 SaveAsQuery类

它继承自 `QueryBlock` 类，表示一个将特定数据保存为文件的查询块对象。

类的构造函数接受两个参数，分别是字符串类型的文件名和枚举类型的保存方式。在构造函数中，该类调用基类 `QueryBlock` 的构造函数，并将指令类型设为 `Instruction::SAVEFILE` 表示此查询块为文件保存操作。同时，构造函数还将文件名和保存方式属性初始化。

该类提供了两个公有成员函数：

- `QString &GetName()`：返回文件名属性值引用。
- `SaveType GetType()`：返回保存方式属性值。

3.3.2.2 SaveAsQueryResult类

它继承自 `ResultClass` 类，表示一个将特定数据保存为文件的查询结果。

类的构造函数接受两个参数，第一个是布尔类型的操作是否成功属性值，第二个是指向 `SaveAsQuery` 对象的智能指针。在构造函数中，该类调用基类 `ResultClass` 的构造函数，并传入这两个参数进行初始化。

该类还重载了基类 `GetQueryBlock()` 函数，该函数的作用是获取该查询结果所对应的查询块对象指针。该函数会首先使用 `assert()` 函数进行断言，抛出一个错误信息，表示不应该调用该函数。然后，该函数返回通过 `reinterpret_cast` 转换后的 `SaveAsQuery` 对象指针，表示该查询结果所对应的查询块对象是一个 `SaveAsQuery` 对象。

3.4 文本格式设置模块

3.4.1 字体大小，样式，颜色

3.4.1.1 ChangeAttrQuery类

3.4.1.1.1 公共成员类

- `Color` 类：这个类主要用于表示颜色，有一个 `uint32_t` 类型的数据成员 `RGB`，用于存储颜色信息。类中定义了一个构造函数，接受一个 `uint32_t` 类型的参数 `RGB`，并将其赋值给数据成员 `RGB`。此外，`Color` 类还重载了 `==` 运算符，以实现对两个 `Color` 对象进行比较操作。
- `Style` 类：这个类主要用于表示样式，有一个 `uint32_t` 类型的数据成员 `style`，用于存储样式信息。类中定义了一个构造函数，接受一个 `uint32_t` 类型的参数 `style`，并将其赋值给数据成员 `style`。同时，`Style` 类也重载了 `==` 运算符，以实现对两个 `Style` 对象进行比较操作。

3.4.1.1.2 私有成员变量和方法

```
    // 存储颜色信息的指针
    Color *color;
    // 存储样式信息的指针
    Style *style;
    // 存储字体信息的指针
    QFont *font;
    // 用于比较两个 ChangeAttrQuery 对象颜色和样式是否相同的私有函数
    bool EqualFormat(ChangeAttrQuery *other);
```

3.4.1.1.3 公有方法

```
     // 获取颜色信息指针
    Color *GetColor();
    // 获取样式信息指针
```

```
    Style *GetStyle();
    // 获取字体信息指针
    QFont *GetFont();
    // 重写 BasicTextQuery 类中的 merge 函数
    virtual bool merge(ChangeAttrQuery *other){
        // 如果当前对象和 other 对象的颜色和样式都相同,
        //则调用父类的 merge 函数并返回 true
        if(this->EqualFormat(other)){
            return this->BasicTextQuery::merge(other);
        }
        // 否则只返回 false
        return false;
    }
};
```

3.4.1.2 ChangeAttrQueryResult类

该类用来封装属性变更操作的结果，它继承了 `ResultClass` 类。该类具有两个构造函数，第一个构造函数接受一个 `bool` 类型的参数 `isOK`，用于表示操作是否成功；第二个构造函数则额外接受一个 `std::shared_ptr<ChangeAttrQuery>` 类型的参数 `query`，用于存储和表示属性变更操作的具体信息。

此外，这个类还重写了父类中的一个函数 `GetQueryBlock()`，但是在这个函数中调用了 `assert` 断言语句，限制了这个函数不应被子类使用。

3.4.2 排版设置模块

定义了一个枚举类型 `ParaFormat`，包含了四个成员： `LEFT_ALIGN`、 `MID_ALIGN`、 `RIGHT_ALIGN` 和 `JUSTIFY_ALIGN`。其中前三个枚举值的二进制表示的末两位分别为 00、01 和 10，用于表示对齐方式为左对齐、居中对齐和右对齐。最后一个枚举值的二进制表示的末两位为 11，用于表示两端对齐。

3.4.2.1 ChangeParaFormatQuery类

它从 `QueryBlock` 类继承而来。该类用于表示改变段落格式的操作。

类的私有成员变量包括：

- `std::vector<Index> para` ，用于存储需要被修改的段落编号。
- `ParaFormat newParaFormat`，表示新的段落格式。

类的构造函数有两个重载版本。第一个版本接受一个 `std::vector<Index>` 类型的参数 `para` 和一个 `ParaFormat` 类型的参数 `newParaFormat`。通过这些参数初始化了对象的成员变量，并调用了父类 `QueryBlock` 的构造函数，传入相应的参数。

第二个版本接受一个 `Index` 类型的参数 `index` 和一个 `ParaFormat` 类型的参数 `newParaFormat`。在构造函数中，使用一个花括号括起来的语法，将 `index` 转化为

`std::vector<Index>` 类型，并赋值给 `para`，然后也是调用了父类的构造函数。

该类声明了两个公有成员函数：

- `ParaFormat GetNewParaFormat()`，用以返回新的段落格式。
- `std::vector<Index>& GetParaImpacted()`，用以返回需要被修改的段落编号。

同时，该类实现了虚函数 `bool merge([[maybe_unused]]ChangeParaFormatQuery *other)`

3.4.2.1 ChangeParaFormatQueryResult类

它从 `ResultClass` 类继承而来。该类用于表示改变段落格式操作的结果。

该类的构造函数有两个参数。第一个参数 `isOK` 是一个 `bool` 类型，表示查询是否成功。第二个参数是一个指向 `QueryBlock` 类型的 `shared_ptr`，表示相关的查询块。在构造函数中，通过传入的参数调用父类 `ResultClass` 的构造函数。

该类还实现了函数 `ChangeParaFormatQuery *GetQueryBlock()override`，用于获取相关的查询块。

3.5 BackImplement类

该类为后端实现类，包含了文本编辑器的各种操作，例如打开、插入、查找、更改属性、保存等。它通过操作数据类TopData中的对象来实现文本编辑器的各种功能，同时还包括了撤销/重做栈的实现以及保存文件等功能。它的作用是为文本编辑器提供后端的操作实现。

3.5.1 私有成员变量和方法

```
    QString name; // 保存文件名
    SaveType type; // 文件保存类型
    UndoRedoStack undoStack; // 撤销操作栈
    UndoRedoStack redoStack; // 重做操作栈
    int savedUndoStackSize=0; // 已经保存的撤销操作数
    void ClearUndoRedoStack(); // 清空撤销和重做栈
    void ClearUndoRedoStack(UndoRedoStack &stack); // 清空指定的栈
    void ClearUndoStack(); // 清空撤销栈
    void ClearRedoStack(); // 清空重做栈
    bool SaveDocx(); // 文件保存为docx格式
    void AddUndo(UndoBlock *undoBlock);// 添加撤销操作
    void AddRedo(UndoBlock *undoBlock);// 添加重做操作

```

3.5.2 Protected 成员变量和方法

```
    TopData objData; // 存储文本数据的类TopData的对象

    std::shared_ptr<BasicTextQueryResult>
GetText(std::shared_ptr<BasicTextQuery>query,std::shared_ptr<BasicTextQueryResu
lt> result); // 获取查询结果并返回std::shared_ptr类型指针

    std::shared_ptr<OpenQueryResult> OpenFile(OpenQuery *query); //打开文件并返回
OpenQueryResult对象的指针

    std::shared_ptr<InsertQueryResult> Insert(InsertQuery *query,UndoRedo
choice); // 插入文本操作并返回InsertQueryResult对象的指针

    std::shared_ptr<TextFindResult>
Find(std::shared_ptr<TextFind>query,std::shared_ptr<TextFindResult> result);
// 查找文本操作并返回TextFindResult对象的指针

    std::shared_ptr<ChangeAttrQueryResult> ChangeAttr(ChangeAttrQuery
*query,UndoRedo choice); // 更改文本属性操作并返回ChangeAttrQueryResult对象的指针

    std::shared_ptr<SaveQueryResult> SaveFile(SaveQuery *query); // 保存文件操作
并返回SaveQueryResult对象的指针

    std::shared_ptr<SaveAsQueryResult> SaveAs(SaveAsQuery *query); // 另存为操作
并返回SaveAsQueryResult对象的指针

    std::shared_ptr<DeleteQueryResult> Delete(DeleteQuery *query,UndoRedo
choice); // 删除操作并返回DeleteQueryResult对象的指针

    std::shared_ptr<UndoQueryResult> Undo(); // 撤销操作并返回UndoQueryResult对象
的指针
    std::shared_ptr<RedoQueryResult> Redo(); // 重做操作并返回RedoQueryResult对象
的指针
    std::shared_ptr<ChangeParaFormatQueryResult>
ChangeParaFormat(ChangeParaFormatQuery *query,UndoRedo choice); // 更改段落属性操
作并返回ChangeParaFormatQueryResult对象的指针

    Position findInsert(Index index); // 查找插入点位置
    Position findDelete(Index index); // 查找删除点位置

    BlockPosition GetNextBlock(Index row,Index index); // 获取下一个块的位置
    BlockPosition GetNextBlock(BlockPosition pos); // 获取从pos开始的下一个块的位置
    virtual bool IsNewFile(); // 是否是新建文件
    virtual bool IsChanged(); // 文件是否被修改过
```

3.5.3 公有方法

```
    WordData *GetBlock(Index row,Index index); // 获取指定行和列的单词块
    WordData *GetBlock(BlockPosition pos); // 获取指定块位置的单词块
    virtual ~BackImplement() = default; // 析构函数
    bool GetIsChanged(); // 判断文件是否被修改过
```

3.6 用户界面模块

3.6.1 MainWindow类

该类是一个继承自 `QMainWindow` 的主窗口类，提供了一个完整的 UI 界面以及一些相关的事件处理函数。其主要作用是实现文本编辑器功能，能够创建、打开、保存、打印、复制、粘贴、剪切、撤销、重做、更改字体和颜色等操作。同时，该类还有一些私有槽函数，用于处理不同的事件触发响应。该类使用了前后端分离的设计思想，将前端与后端逻辑分开，通过 `Front_Back_Interface.hpp` 中定义的 API 借助 `interface` 变量实现交互。

3.6.1.1 私有成员变量和方法

```
    static MainWindow *singleInstance;  // MainWindow对象的单例模式实现
    Ui::MainWindow *ui;  // 指向MainWindow对象的UI界面的指针
    FrontBackInterface interface;  // 前后端接口API类
    // 获取当前选中区域的首尾游标位置
    std::pair<Position, Position> GetSel(bool isInsert);
    // 询问用户是否保存文件
    int AskIfSave();
```

3.6.1.2 私有槽函数

```
private slots:
    // 创建新文件事件处理函数
    void on_actionNew_triggered();
    // 打开文件事件处理函数
    void on_actionOpen_triggered();
    // 保存文件事件处理函数
    bool on_actionSave_triggered();
    // 打印文件事件处理函数
    void on_actionPrint_triggered();
    // 退出程序事件处理函数
    void on_actionExit_triggered();
    // 复制事件处理函数
    void on_actionCopy_triggered();
    // 粘贴事件处理函数
    void on_actionPaste_triggered();
```

```
    // 剪切事件处理函数
    void on_actionCut_triggered();
    // 撤销事件处理函数
    void on_actionUndo_triggered();
    // 重做事件处理函数
    void on_actionRedo_triggered();
    // 更改字体事件处理函数
    void on_actionFont_triggered();
    // 更改颜色事件处理函数
    void on_actionColor_triggered();
    // 另存为事件处理函数
    bool on_actionSave_as_triggered();
    // 设置为加粗文字事件处理函数
    void on_actionBold_triggered();
    // 设置为斜体文字事件处理函数
    void on_actionItalic_triggered();
    // 设置为下划线文字事件处理函数
    void on_actionUnderline_triggered();
    // 左对齐文本事件处理函数
    void on_actionLeftAlign_triggered();
    // 居中对齐文本事件处理函数
    void on_actionCenterAlign_triggered();
    // 右对齐文本事件处理函数
    void on_actionRightAlign_triggered();
    // 两端对齐文本事件处理函数
    void on_actionJustifyAlign_triggered();
```

3.6.1.3 公有成员变量和方法

```
    bool isFirst = true;  // 是否为第一次打开文件
    bool isActionByProgrammer = false;  // 是否为程序员操作
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    bool shouldModify() { return isActionByProgrammer; }  // 判断是否需要修改
    // 获取MainWindow对象
    static MainWindow *GetMainWindow() { return singleInstance; }
    // 删除字符事件处理函数
    void on_actionDelete_triggered(int position, int removedChars);
    // 添加字符事件处理函数
    void on_actionAdd_triggered(int position, int addChars);
    // 更改字符样式，如加粗、斜体、下划线等
    void ChangeStyle(bool Bold, bool Italic, bool Underline);
    // 获取当前选中区域的行数
    std::vector<Index> GetSelRow();
    // 关闭程序时触发的事件处理函数
```

```
17      void closeEvent(QCloseEvent *e) override;
```

3.6.2 mainwindow.ui

- 文件格式为XML，用于描述Qt框架中的用户界面(UI)。
- 主要包含主窗口(QMainWindow)和各种控件，例如文本编辑框(MyTextEdit)、水平滚动条(QScrollBar)、垂直滚动条(QScrollBar)、菜单栏(QMenuBar)、状态栏(QStatusBar)和工具栏(QToolBar)等。
- 包含一些动作(Action)描述，例如新建文件、打开文件、保存、剪切、复制、粘贴、撤消、重做、退出等，以及一些用于界面控件样式设计的动作描述，例如字体、颜色、加粗、倾斜、下划线、对齐方式等。
- 包括图标资源等描述。

3.7 资源管理模块

3.7.1 resource.qrc

该文件是Qt资源文件配置（.qrc）文件，它的作用是将应用程序需要使用的所有资源文件打包到一个二进制文件中，供程序调用。这样做的好处是可以对资源进行统一管理，便于程序的维护和发布，同时也能提高程序的运行效率和安全性。

该文件包含两个 `qresource` 标签，分别用于存放图片和翻译文件。

第一个 `qresource` 标签中存放了应用程序需要使用的图片资源文件，包括了多种操作所需的图标、界面元素等。这些文件被打包成二进制格式，编译进应用程序，应用程序启动时可以直接通过 `:/` 前缀访问这些资源文件，而无需再次读取磁盘上的文件，从而提高了程序的运行效率。

第二个 `qresource` 标签中则包含了多语言支持所需的翻译文件，包括Qt本身和应用程序特定的UI文件的翻译文件等。这些翻译文件同样被打包成二进制格式，编译进应用程序，应用程序启动时可以根据用户选择的语言动态加载相应的翻译文件，实现应用程序的多语言支持。

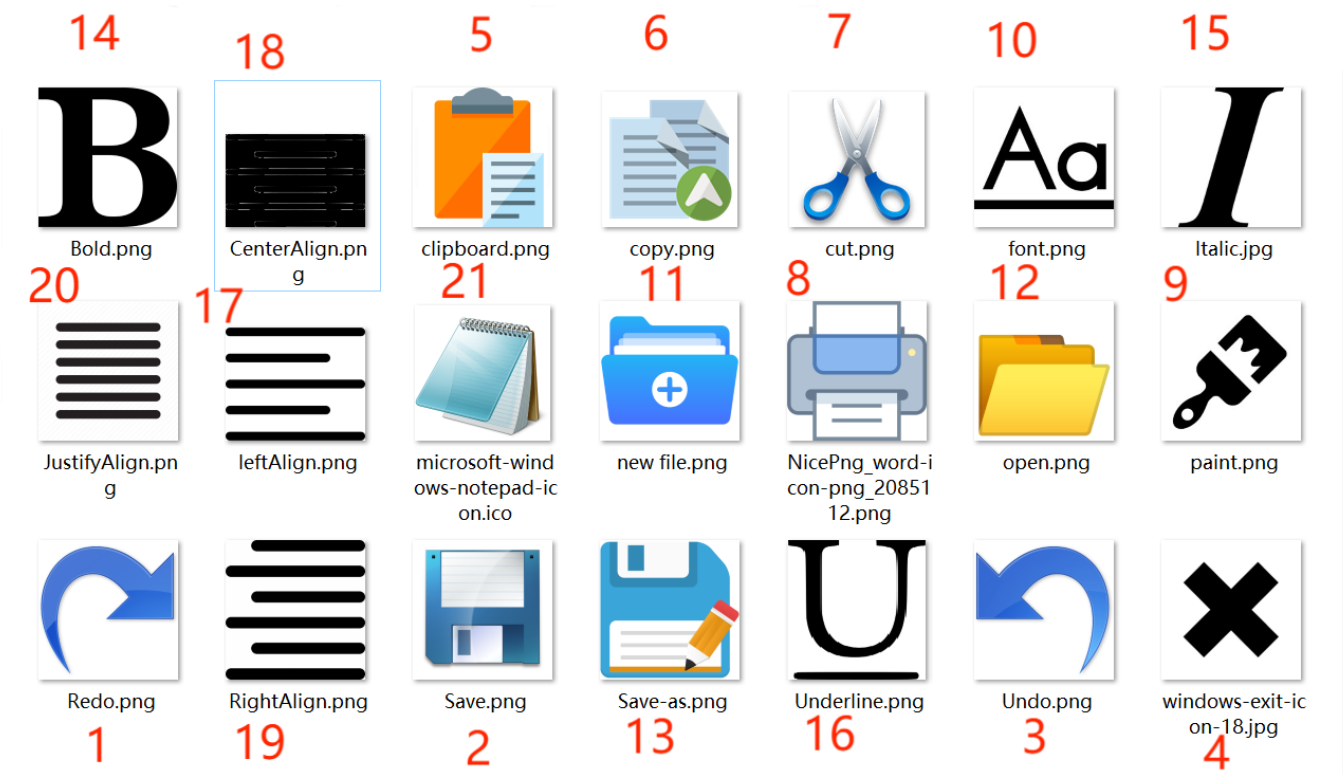
3.7.2 图片资源

图片资源主要用于美化和装饰应用程序的用户界面，这些图标用于表示应用程序的各种功能和操作，并且全部采用透明度较好、压缩率较高的 PNG 格式以获得更好的视觉效果。具体来说包括了保存、撤销、重做等操作的图标。通过使用这些图标，可以让应用程序界面更加清晰、直观、易用，并且提高整个应用程序的专业性和美观程度。

1. `imgs/Redo.png`：重做操作的图标。
2. `imgs/Save.png`：保存操作的图标。
3. `imgs/Undo.png`：撤销操作的图标。
4. `imgs/windows-exit-icon-18.jpg`：退出应用程序的图标。

5. `imgs/clipboard.png`：剪贴板操作的图标。
6. `imgs/copy.png`：复制操作的图标。
7. `imgs/cut.png`：剪切操作的图标。
8. `imgs/NicePng_word-icon-png_2085112.png`：文字编辑器的图标。
9. `imgs/paint.png`：画笔工具的图标。
10. `imgs/font.png`：字体选择工具的图标。
11. `imgs/new file.png`：新建文本文档的图标。
12. `imgs/open.png`：打开文件的图标。
13. `imgs/Save-as.png`：另存为操作的图标。
14. `imgs/Bold.png`：加粗文字的图标。
15. `imgs/Italic.jpg`：斜体文字的图标。
16. `imgs/Underline.png`：下划线文字的图标。
17. `imgs/LeftAlign.png`：左对齐的图标。
18. `imgs/CenterAlign.png`：居中对齐的图标。
19. `imgs/RightAlign.png`：右对齐的图标。
20. `imgs/JustifyAlign.png`：两端对齐的图标。
21. `imgs/microsoft-windows-notepad-icon.ico`：应用程序图标。

以上编号分别对应下图中的图标编号



3.7.3 语言资源

语言资源采用了 Qt 自带的翻译文件格式（.qm），位于"lang"文件夹下。这些语言资源的作用是支持应用程序在不同的语言和文化环境下本地化和国际化展示。在应用程序中使用 Qt 的翻译文件，可以通过轻松地修改语言环境，来实现应用程序界面文字、日期格式、货币单位等相应地更改。这样做可以让用户在使用应用程序时更加舒适和便捷，提高应用程序的易用性和受众范围。

- qt_zh_CN.qm：Qt 库的中文简体翻译文件。
- qtbase_zh_CN.qm：Qt 基础库的中文简体翻译文件。
- qtlocation_zh_CN.qm：Qt 定位库的中文简体翻译文件。
- qtmultimedia_zh_CN.qm：Qt 多媒体库的中文简体翻译文件。
- zh_CN.qm：应用程序的中文简体翻译文件。
- en_US.qm：应用程序的英文翻译文件。
- ui_zh_CN.qm：应用程序 UI 界面的中文简体翻译文件。
- ui_en_US.qm：应用程序 UI 界面的英文翻译文件。

3.8 docx格式文档解析

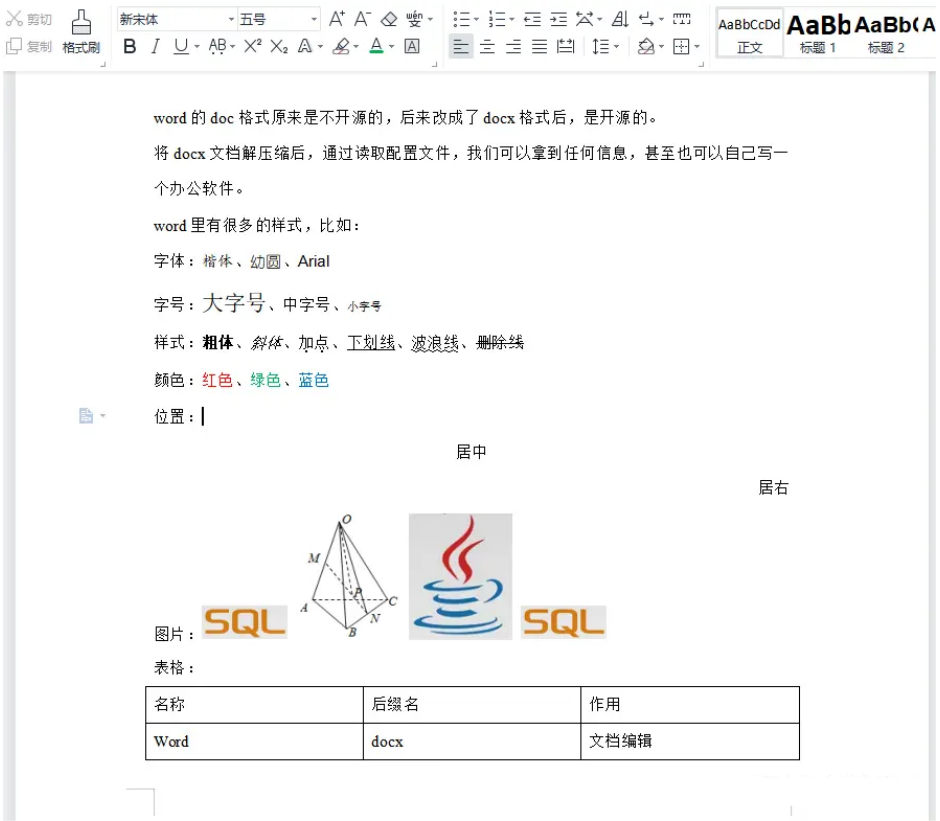
3.8.1 目的

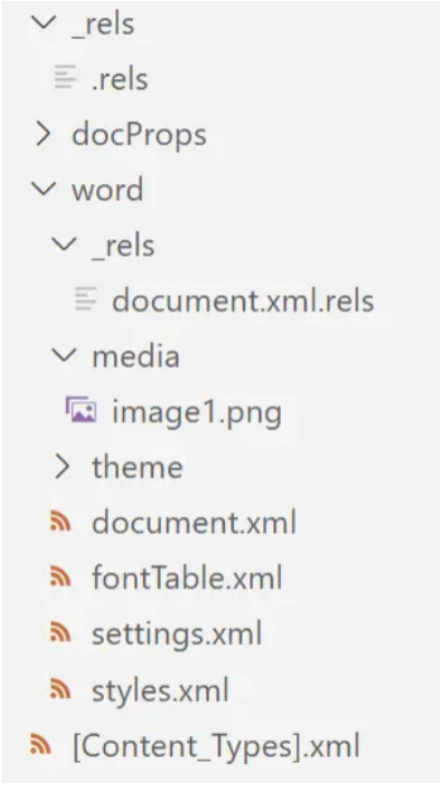
文本编辑器做docx文档解析的目的是为了从word文档中提取结构和内容，以便于后续的处理和应用。通过解析docx格式的word文档，我们可以获取文档中的文字、图片、表格等数据，并进行处理和分析。在不同的领域，例如教育信息化、金融、医疗等，都会涉及到对文档的解析和处理。例如，在教育信息化领域中，可以通过解析word文档来提取试卷名、题目、选项等信息，以便于后续的评分和统计分析。在金融领域中，可以通过解析word文档来提取合同、财务报表等信息，以便于实现自动化处理。因此，文本编辑器解析docx文档的目的是为了更好地应用word文档中的内容。

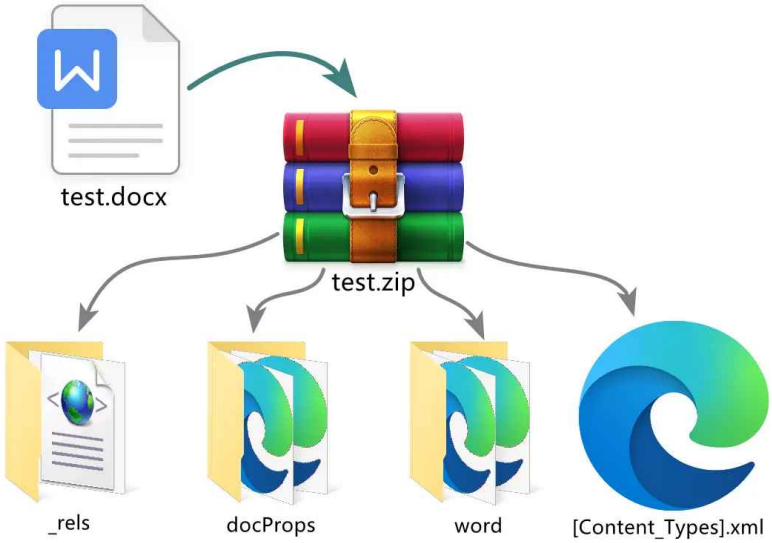
3.8.2 解析原理

docx文件实际上是一个基于XML和ZIP格式的复合文档格式，它把多个文件打包在一起形成一个zip文件，并使用XML文件描述不同部分（比如文字、图片、表格等）的结构。因此，在解析docx文件时，需要用类似于解析zip文件的方式来处理其内部数据，同时读取其XML文件来确定各个部分的内容和位置等信息。在处理完各个部分的数据后，可以将它们重新组合起来以生成文档的最终内容。

例如将下面文档的后缀将.docx改为.zip，再解压，就能查看docx文件内部结构和内容。

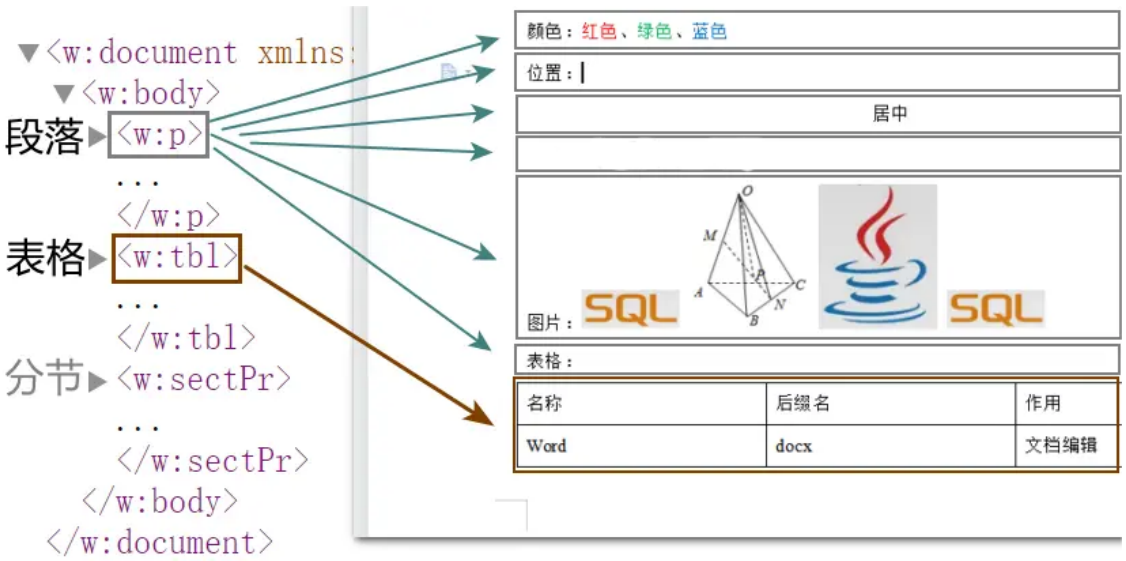






3.8.3 主文件document.xml

document.xml文件包含了文档结构和各个部分之间的关系，以及每个部分所包含的详细信息。例如，文档中的段落、标题、文本框、注释等，都被记录在document.xml文件中的对应位置。同时，document.xml文件还包含了与文档格式和样式相关的大量信息，例如字体、颜色、边框、页眉页脚、目录等。因此，通过解析document.xml文件，我们可以了解到整个文档的结构、内容和格式等方面的信息，从而更加方便地对文档进行处理和应用。



3.8.4 标签说明

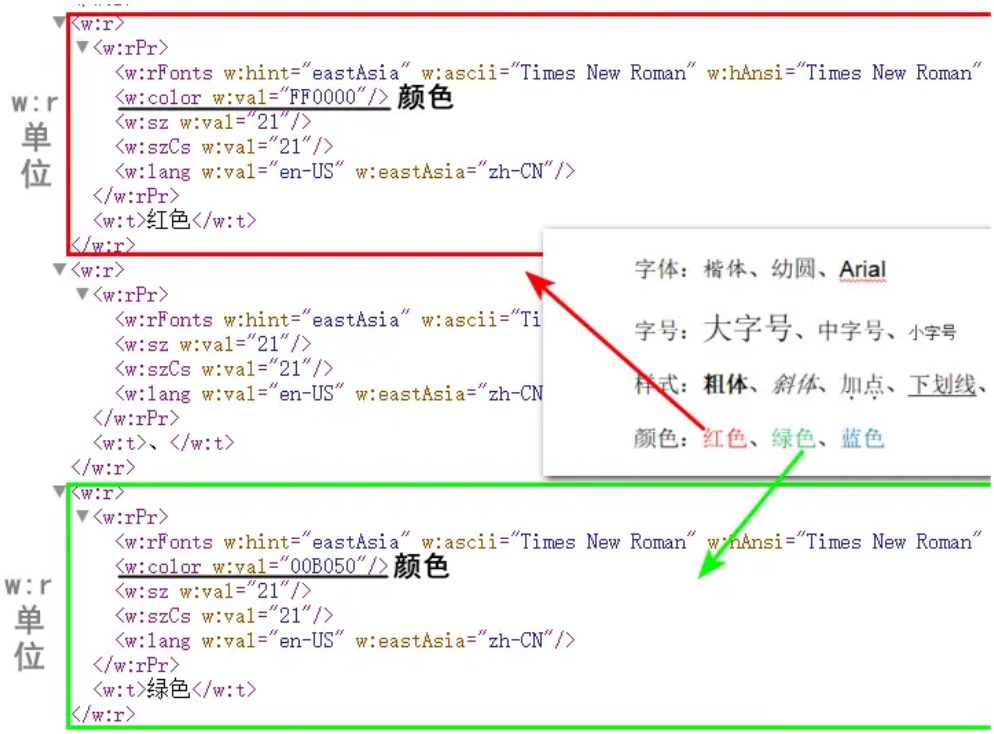
3.8.4.1 段落标签 w:p

每个段落都可以用<w:p></w:p>元素来表示。无论是空白行还是非空白行，只要没有换行符将其分隔开，都将被当做一个段落来处理。因此，对于连续的多个空白行，每个空白行都将被解析成一个独立的<w:p></w:p>元素，从而成为单独的段落。

3.8.4.2 连续块标签 w:r

<w:r>标签是Word文档中的一个XML标签，它表示一段具有相同格式和样式的连续文本块。通常情况下，每个<w:r>标签嵌套在<w:p>标签中，用来表示一个段落中的具有不同样式的文本。

<w:r>标签通常用于定义文本的样式和属性，例如字体、字体大小、颜色、加粗、斜体、下划线等。一个<w:r>标签可以包含多个XML元素，例如<w:t>、<w:rPr>和<w:tab>等。



3.8.4.3 文本标签 w:t

<w:t>标签表示<w:r>标签中的一段文字内容。每个<w:r>标签可以包含多个<w:t>元素，用来表示<w:r>标签中的不同片段的文本内容。例如，如果一段文字中含有下划线、加粗等格式，则这些文字将被拆分为多个<w:r>标签和<w:t>标签来表示，同时在<w:r>标签内通过<w:rPr>元素来定义该段文字的格式和样式。

3.8.5 第三方库

3.8.5.1 Minizip库

MiniZip是基于C语言的开源压缩库，提供了一组API函数，可以用于创建、读取、修改、关闭ZIP归档文件（.zip），支持标准的ZIP文件格式以及PKWare公司的增强压缩（Deflate64）格式。MiniZip在许多软件中都有广泛应用，例如WinZip、7-Zip等压缩软件就使用了MiniZip库。

MiniZip的主要特点是简单易用、跨平台、高效快速，并且开源免费。它可以轻松地实现ZIP文件的创建、读取、修改、删除、重命名、注释等操作，并支持LHA、CAB、gzip、bzip2等多种压缩格式。

3.8.5.2 TinyXML库

TinyXML是一个轻量级、简单易用的C++ XML解析库，适用于嵌入式设备等资源受限的场景。TinyXML的主要特点如下：

1. 体积小：整个库只有几个文件，不到100KB。

2. 易于使用：TinyXML提供了简单易用的API，使用者无需了解复杂的XML规范，即可快速解析XML文件。
3. 轻量级：TinyXML没有额外依赖库，只需要一个标准C++编译器即可编译通过。
4. 快速稳定：TinyXML使用的是SAX（Simple API for XML）风格的解析方式，可以快速解析大型XML文件，并且不会因为内存问题导致程序奔溃。

TinyXML支持对XML文件的读取、写入、修改操作，可以方便的获取XML文件中的节点、属性、文本等信息。

3.8.6 document.xml解析

首先，在myunzip类中定义了unzip方法用于解压缩zip文件。接着，在xml_read类中定义了readXmlFile()函数，用于解析名为"document.xml"的XML文件，并提取"w:p"和"w:r"标签内的属性信息。

3.8.6.1 myunzip类

在该类中定义了unzip方法用于解压缩zip文件。该方法接收一个参数，即待解压缩的zip文件路径。具体实现步骤如下：

1. 判断传入的文件路径是否为空或文件类型是否为zip格式；如果不是，则输出相应提示信息并返回。
2. 获取zip文件所在的父级目录。
3. 打开zip文件，并获取zip的全局信息。
4. 遍历zip文件内的所有文件：
    - 获取zip包内当前文件的文件名和文件信息；
    - 将父级目录和文件名拼接起来，得到文件的完整路径；
    - 判断当前文件是目录还是文件，如果是目录则创建该目录，如果是文件则将其写入到磁盘中。
5. 关闭zip文件。

3.8.6.2 xml_read类

该类定义了readXmlFile()函数，用于解析名为"document.xml"的XML文件，并提取其中"w:p"和"w:r"标签内的属性信息。

在readXmlFile()函数中，首先通过SetCurrentDirectoryA函数将工作目录切换到"c:*\\word"，即文档所在目录。然后创建一个TiXmlDocument对象，并使用LoadFile方法载入XML文件。如果载入失败，则输出错误信息并退出程序。

接着根据XML的结构，遍历根元素下所有"w:p"标签，并进一步遍历其中的所有"w:r"标签。如果标签名与"w:r"相同，则说明该标签包含所需属性信息。可以依次获取"w:rPr"、"w:rFonts"、"w:color"、

"w:b"、"w:i"、"w:u"和"w:t"标签，以提取Font对象、颜色、加粗、斜体、下划线和文本内容等属性信息。最后，将所有属性信息保存在一个xmlattr对象中，并将其添加到res向量中。

3.8.7 document.xml文件生成

3.8.7.1 xmlzip类

该类中定义了MyXmlZip()方法，实现了将当前目录及其所有子目录下的文件和文件夹压缩成一个新的zip文件的功能。具体实现过程如下：

1. 获取当前工作目录。
2. 打开一个名为"output.zip"的新zip文件，并返回一个zipFile类型的对象。
3. 使用C++17引入的std::filesystem库中的recursive_directory_iterator对当前目录及所有子目录进行遍历，获取各个文件和文件夹的路径和相对路径。
4. 判断每个路径对应的是文件还是文件夹，如果是文件，则打开该文件，并使用zipOpenNewFileInZip函数将该文件加入到zip压缩文件中；如果是文件夹，则使用zipOpenNewFileInZip函数创建该文件夹，并将其添加到zip压缩文件中。
5. 关闭zip压缩文件。

3.8.7.2 ToDocx类

该类中定义了XmlToDocx()方法，用于将文本数据转换为docx格式。首先，创建一个XML文档，并在文档中创建了根元素为"w:document"，子元素为"body"的基础结构。然后，它使用接口中的命令获取文本块，逐个识别文本块中的数据并将其转换为XML元素。对于每个文本块，如果是新段落，则创建一个"w:p"元素作为新的段落，并设置段落格式；否则直接将文本块数据生成"w:r"元素，并设置相应的字体、大小以及样式（如加粗、斜体、下划线）。最后，将生成的XML文档保存到文件"document.xml"中。同时，如果原有的XML文件存在，则先删除原有文件。

4 系统设计难点及解决

4.1 数据结构模块

4.1.1 设计难点

1. 如何存储文本单词的内容和格式信息，以及如何快速插入、删除和修改单词；
2. 如何支持文本块的分割和组合，并记录其对应的WordData对象；
3. 如何实现各种查询操作，如查找文本、修改颜色、样式等属性；
4. 如何实现撤销/重做功能，保证编辑器的稳定性；
5. 如何扩展查询块的功能，使得不同的查询块类可以共享基础属性和方法。

4.1.2 解决方案

1. WordData 类作为文本单词的基本单元，包含单词的内容和格式信息，并提供一系列方法用于快速插入、删除和修改单词，以及获取和设置单词的属性。同时，它还支持复制和拆分操作，方便其他高级数据结构进行组合和拆分。
2. TopData 类作为文档的总体存储单元，使用一个二维数组来存储整个文档的字符数据，而每个元素都对应着一个 WordData 对象。同时，使用 indexOfBlocks 数组记录文本块的位置信息，方便定位和查询。TopData 类还提供了插入、删除、查询、修改、格式化等操作，保证文本编辑器的稳定性，并支持了撤销/重做功能。
3. QueryBlock 类作为通用的查询块类，封装了查询块的 ID，并提供了获取查询块 ID 的接口和 merge 函数。具体的查询块类可以通过继承 QueryBlock 类并实现具体的查询逻辑，实现各种查询操作。
4. 为了支持撤销/重做功能，TopData 类引入了一个 ChangeAttrQuery 对象，用于记录每次修改单词属性的操作，方便进行回滚操作。
5. 考虑到查询块类的扩展性，实现了一个基础的 QueryBlock 类，并采用继承和多态的方式来扩展其功能。这样，不同的查询块类可以共享基础属性和方法，避免了重复编写代码的问题。

4.2 文本编辑模块

4.2.1 查询结果的处理

查询结果是指查询操作返回的结果，这些结果需要进行进一步的处理才能展示给用户。为了提高程序的可维护性和可扩展性，文本编辑模块采用了 ResultClass 类作为查询结果的基类，并允许用户通过继承该类来自定义查询结果的处理过程。例如，用户可以定义一个包含多个 ResultClass 对象的列表，并对其进行排序、筛选等操作，以便最终呈现给用户的结果更符合期望。

4.2.2 文本搜索替换的实现

搜索替换的实现需要考虑到搜索的关键词、搜索选项以及匹配的文本位置等问题。为了解决这些问题，文本编辑模块采用了 TextFind 类来表示搜索操作，其中包含了搜索的关键词、搜索选项等属性，以及执行搜索操作的方法。在执行搜索操作后，文本编辑模块将返回一个 TextFindResult 对象，其中包含了匹配的文本位置等信息。

4.2.3 撤销和重做操作的实现

撤销和重做操作的实现需要考虑到对历史操作的记录、数据恢复等问题。为了解决这些问题，文本编辑模块采用了 UndoQuery、RedoQuery、UndoQueryResult 和 RedoQueryResult 等类来表示撤销和重做操作。具体地，每次执行编辑操作时，文本编辑模块会将该操作记录到撤销栈中，并生成相应的 UndoQuery 对象。当用户请求撤销操作时，文本编辑模块会从撤销栈中取出最近的操作，并生成相应的 UndoQueryResult 对象，供用户确认是否执行撤销操作。类似地，重做操作也是通过 RedoQuery、RedoQueryResult 等类来实现的。

4.3 文件功能模块

4.3.1 文件打开

主要难点在于如何获取用户需要打开的文件名并且进行相应的操作。通过定义 OpenQuery 类来处理该问题，在构造函数中接收用户传入的文件名参数并将其赋值给私有数据成员 filename。同时，OpenQueryResult 类用于表示一个打开文件操作的查询结果，并且它的构造函数接受一个布尔值参数 isOK 表示操作是否成功。

4.3.2 文件保存

主要难点在于如何处理用户传入的文件名和保存方式不同的保存方式。定义了 SaveQuery 类，并且在构造函数中设置指令类型（Instruction::SAVEFILE），表示该查询块的动作是进行文件保存。同时，SaveQueryResult 类用于表示一个文件保存操作的查询结果，在构造函数中，通过建立指针与查询块对象的联系，来处理不同的保存方式。该类提供了两个构造函数重载，一个用于表示操作是否成功，一个用于表示操作是否成功并且包含查询块对象指针。

4.4 文本格式设置模块

4.4.1 对字体大小、样式和颜色进行统一管理

定义三个类来表示这三种信息，即 Color、Style 和 QFont（字体），然后将这三个类的指针作为私有成员变量存储在 ChangeAttrQuery 类中。同时，在 ChangeAttrQuery 类中实现了 EqualFormat() 函数，用于比较两个 ChangeAttrQuery 对象的颜色和样式是否相同，从而方便 merge() 函数的实现。

4.4.2 实现段落格式的改变。

定义一个枚举类型 ParaFormat，其中包含了四个成员，分别表示左对齐、居中对齐、右对齐和两端对齐。然后定义 ChangeParaFormatQuery 类表示段落格式的改变操作，其中包括需要被修改的段落编号和新的段落格式。在 ChangeParaFormatQuery 类中实现 merge() 函数以支持撤销/重做功能。最后，定义ChangeParaFormatQueryResult 类用于表示段落格式的改变结果，并实现 GetQueryBlock() 函数以返回相关的查询块。

4.5 用户界面模块

4.5.1 界面布局的设计

在界面布局设计时，需要考虑到用户需求、功能和使用场景。先绘制草图来确定控件的位置和大小，再利用Qt Designer工具将控件添加到界面中，并设置其属性和布局方式来实现具体的界面布局。

4.5.2 响应事件的处理

在UI界面中定义信号与槽函数，通过关联信号与槽函数的方式，实现控件的事件响应。例如，在MainWindow类中定义了一个槽函数on_actionNew_triggered()，用于响应新建文件动作的信号，实现新建文件的功能。具体实

4.5.3 前后端交互的实现

通过定义FrontBackInterface类对象interface，来实现前后端交互。在MainWindow类中，定义了该类对象，并在需要调用后端逻辑函数时，通过interface对象来实现。例如，在on_actionSave_triggered()中，调用后端逻辑中的SaveFile函数实现文件保存功能：

4.6 资源管理模块

4.6.1 设计难点

1. 如何将应用程序需要使用的资源文件打包进一个二进制文件中，并以最优化的方式提高程序的运行效率和安全性。
2. 如何支持多语言和文化环境的本地化和国际化展示。
3. 怎样能够使应用程序界面更加清晰、直观、易用，并增强整个应用程序的专业性和美观程度。

4.6.2 解决方案

1. 采用Qt资源文件配置（.qrc）文件格式来存储所有的资源文件，包括图片和翻译文件。这种解决方案能够将应用程序所需的所有资源文件打包到一个二进制文件中，提高了程序的运行效率和安全性。
2. 采用Qt自带的翻译文件格式（.qm），并将其放置于“lang”文件夹内。在应用程序中使用Qt的翻译文件，可以通过轻松地修改语言环境，更改应用程序界面文字、日期格式、货币单位等内容，实现应用程序的本地化和国际化展示。
3. 使用具有透明度较好、压缩率较高的PNG格式的图片资源，包括了多种操作图标，如保存、撤销、重做、剪贴板，剪切、复制等操作的图标。这些图片资源能够使应用程序界面更加清晰、直观、易用，并增强整个应用程序的专业性和美观程度。

4.7 document.xml解析

4.7.1 设计难点

1. docx文件的格式较为复杂，需要使用类似于解压缩zip文件的方式来处理其内部数据，并通过读取其XML文件获取文档信息。
2. document.xml文件是docx文件中的一个重要组成部分，需要解析其中的内容，包括段落、文本和样式等信息。在解析document.xml文件时，需要逐个识别其中的标签并提取相应的属性信息。
3. 在生成document.xml文件时，需要将文本数据转换为docx格式数据，并设置相应的段落格式和字符格式。

4.7.2 解决方案

1. 使用开源的Minizip库来处理docx文件。该库提供了一组API函数，可以用于创建、读取、修改、关闭ZIP归档文件，支持标准的ZIP文件格式以及PKWare公司的增强压缩（Deflate64）格式。通过使用该库，可以高效地处理docx文件，并读取其中的内容。
2. 使用TinyXML库来解析document.xml文件中的内容。该库是一个轻量级、简单易用的C++ XML解析库，适用于嵌入式设备等资源受限的场景。通过使用该库，可以方便地获取XML文件中的节点、属性、文本等信息。使用循环逐一解析document.xml文件中的标签，并提取其中的属性信息。在解析w:p和w:r标签时，需要识别各个子标签，并根据其属性信息构造相应的段落或文本对象。
3. 创建一个XML文档，并在文档中创建了根元素为"w:document"，子元素为"body"的基础结构。然后，使用接口中的命令获取文本块，逐个识别文本块中的数据并将其转换为XML元素。对于每个文本块，如果是新段落，则创建一个"w:p"元素作为新的段落，并设置段落格式；否则直接将文本块数据生成"w:r"元素，并设置相应的字体、大小以及样式（如加粗、斜体、下划线）。

5 总结

该文本编辑器是一个基于面向对象设计思想开发项目，旨在提高用户的编辑效率和体验，为用户提供一个功能完备、易用、稳定可靠的文本编辑环境。

在需求分析阶段，我们将文本编辑器的功能需求分解为多个小的模块，并将每个模块作为一个类来进行设计。通过这样的方式，每个类都具有独立的功能和特性，可以被其他类调用和复用，从而实现了代码的模块化、可维护性和可扩展性。同时，在设计过程中我们尽可能考虑到了用户的实际需求，注重实现正确、高效、易用的功能。

我们采用了前后端分离的架构，将前端负责用户交互和界面绘制，后端则负责文本处理和数据存储等功能。通过接口来实现前后端的通信，实现了功能的隔离和模块化，方便后期的维护和升级。对于前端的设计，我们使用了Qt6提供的图形界面组件，并编写相应的事件处理函数。同时，我们还通过实现菜单栏、工具栏、状态栏等组件，使得用户可以方便地进行编辑操作。对于后端的设计，我们主要是通过自定义函数库及部分Qt接口来实现文本处理功能，为了提高效率，我们还实现了文本缓存、文件读写等功能。同时，为了保证数据的安全性和版本控制，我们实现了文本编辑历史记录、恢复到指定版本等功能。

特别地，我们的文本编辑器在实现过程中，还注重解决了一些常见的编辑问题。针对查询结果处理，我们采用 ResultClass 类作为查询结果的基类，并允许用户通过继承该类来自定义查询结果的处理过程。为了解决文件打开和保存的问题，我们定义了 OpenQuery 和 SaveQuery 类，并通过这两个类来处理用户传入的文件名和保存方式。同时，我们还实现了相应的查询结果类 OpenQueryResult 和 SaveQueryResult，来表示打开文件和保存文件操作的查询结果。并且，我们定义了 Color、Style 和 QFont 类来表示字体大小、样式和颜色等信息，并将它们的指针存储在 ChangeAttrQuery 类中。针对界面布局的设计，我们先绘制草图来确定控件的位置和大小，再利用 Qt Designer 工具将控件添加到界面中，并设置其属性和布局方式。而对于事件响应的处理，我们在UI界面中定义信号与槽函数，并通过关联信号与槽函数的方式，实现控件的事件响应。除此之外，我们采用 Qt 资源文件配置文件格式来存储所有的资源文件，并使用 Qt 的翻译文件来实现多语言本地化展示。

面向对象思想是一种将事物拆分为许多类和对象的编程思想。它将现实世界的概念和关系映射到程序中，将代码组织成可重用、可维护的模块，让代码更加易于开发、扩展和维护。知行合一理念是指理论与实践相结合，通过不断地探索和实践，来增强对知识的理解和掌握。

在这个文本编辑器项目中，我们采用了面向对象设计思想来完成各个模块的开发和整合，使得代码结构清晰、易于管理，同时也满足了用户的需求。并且，在学习理论知识的同时，我们还积极地付出行动，将理论知识转化为实际应用。我们小组团结一致，分工明确，在开发过程中时刻保持密切的交流和讨论，我们细致地分析了用户需求，不断地进行调试、优化和完善，最终开发出一个功能完备、易用、稳定可靠的文本编辑器。通过这个过程，我们不仅掌握了更多的技能，而且积累了宝贵的经验和成长。

未来，我们应该继续努力，不断提高自己的技术和专业能力，追求更高效、更安全、更稳定的软件开发。我们也应该注意拓展眼界，了解新的技术和趋势，积极融入开源社区和技术圈，与同行们交流和合作，共同推进技术发展和产业进步。同时，我们也应该注重团队协作和管理能力的提升，加强沟通、合作和领导能力的训练，为未来的职业生涯打下坚实的基础，并为团队的成功贡献自己的力量。

6 程序使用说明

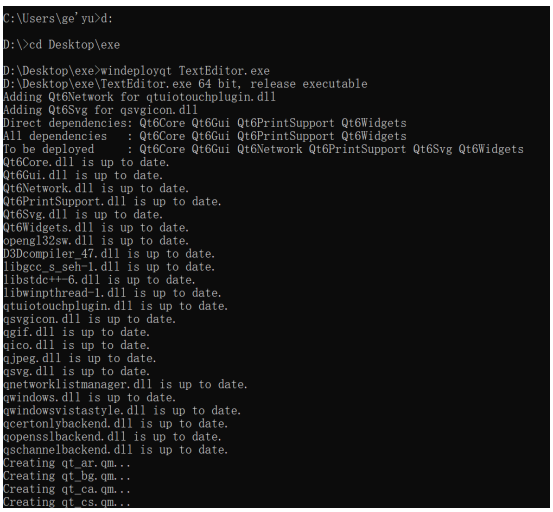
6.1 运行环境

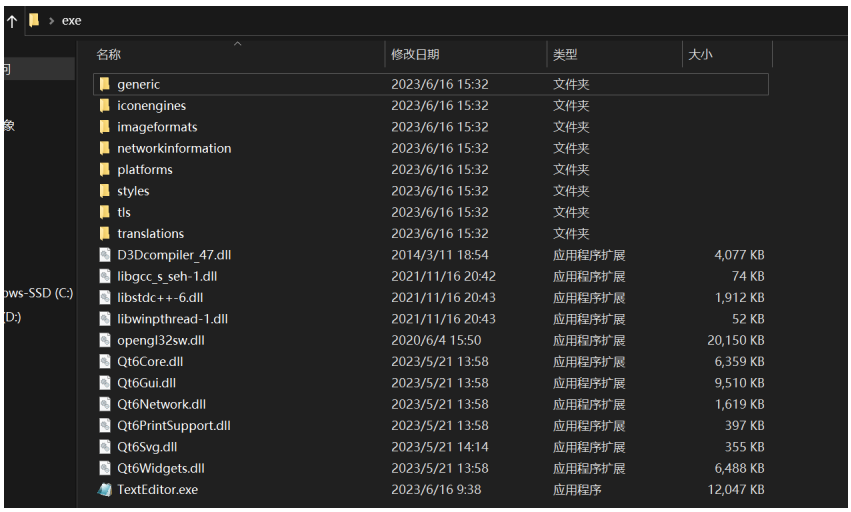
Windows操作系统，需要安装Qt6软件并配置相应的环境变量

6.2 打开软件

新建空文件夹，将TextEditor.exe移动至该文件夹，打开命令提示符窗口，进入到该文件夹中，输入命令windeployqt TextEditor.exe，将Qt程序所依赖的动态库文件和相关资源文件一同打包到可执行文件所在目录中。

鼠标双击TextEditor.exe，即可打开文本编辑器。





6.3 文本编辑

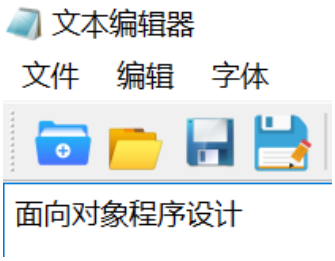
6.3.1 输入

通过键盘输入文本，可输入字母、数字、汉字、标点、特殊符号。按下"Enter"可实现换行。"Tab"键可实现缩进。

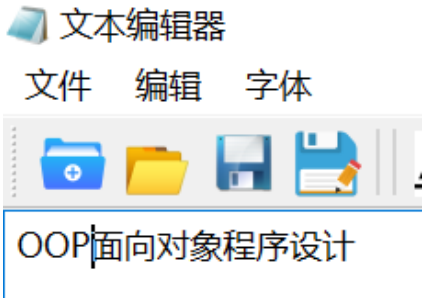
6.3.2 插入

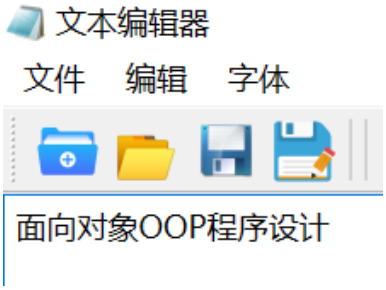
通过键盘左右键或鼠标点击可移动光标，在插入位置插入文本，可在文字前后实现插入

插入前：



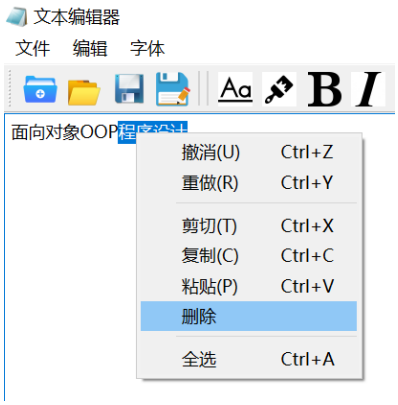
插入后：





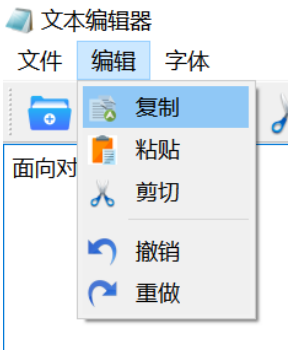
6.3.3 删除

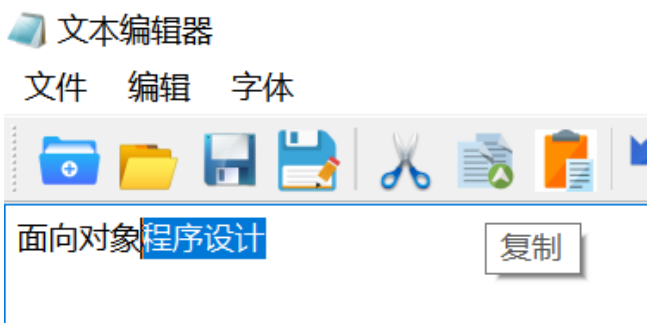
通过键盘"Backspace"键可删除光标左边内容，按下"Delete"键可删除光标右边内容。连续按住删除键不放可连续删除文本。还可以用鼠标左键将需要删除的文本选中，再右键点击"删除"实现指定文本的删除。

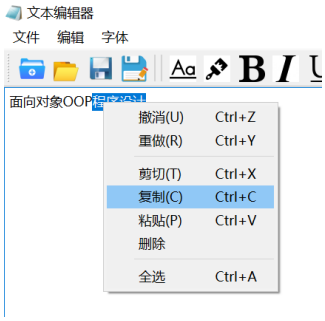


6.3.4 复制

用鼠标左键选中需要复制的内容，点击工具栏"复制"图标或者菜单栏"编辑"下的"复制"选项或者鼠标右键"复制"或者使用快捷键"Ctrl + C"将文本复制到系统剪贴板。

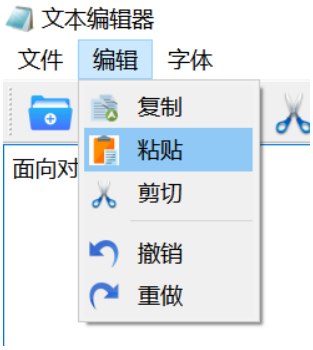


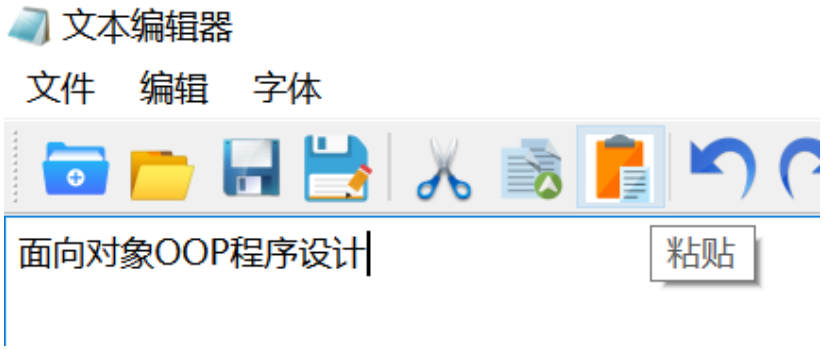


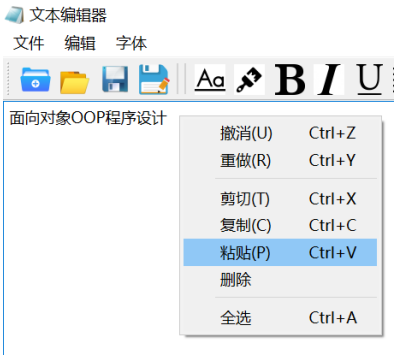


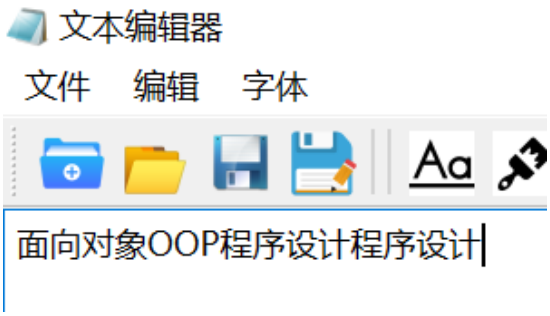
6.3.5 粘贴

点击工具栏"粘贴"图标或者菜单栏"编辑"下的"粘贴"选项或者鼠标右键"粘贴"或者使用快捷键"Ctrl + V"将已复制到系统剪贴板的内容粘贴到指定位置。



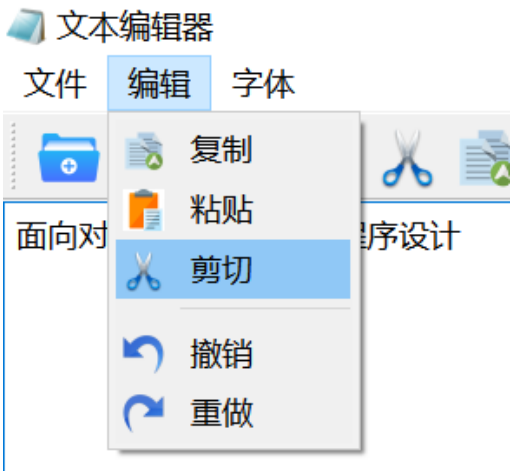


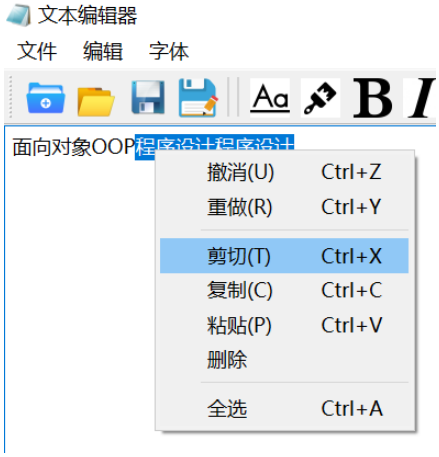


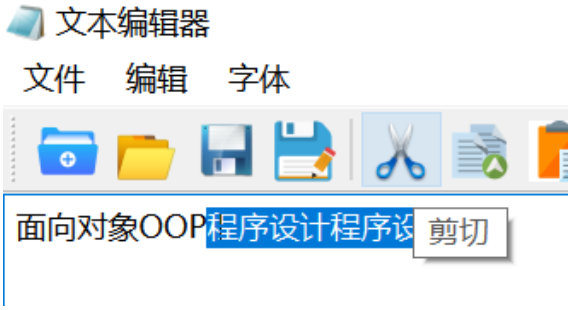


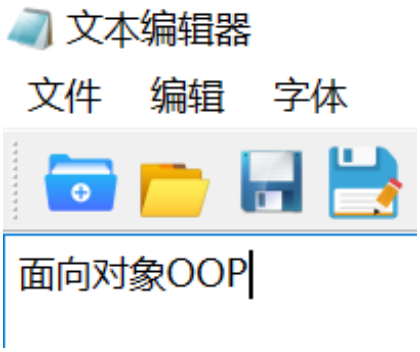
6.3.6 剪切

用鼠标左键选中需要剪切的内容，点击工具栏"剪切"图标或者菜单栏"编辑"下的"剪切"选项或者鼠标右键"剪切"或者使用快捷键"Ctrl + X"，被剪切的文本将被清除，并且被复制到系统剪贴板。



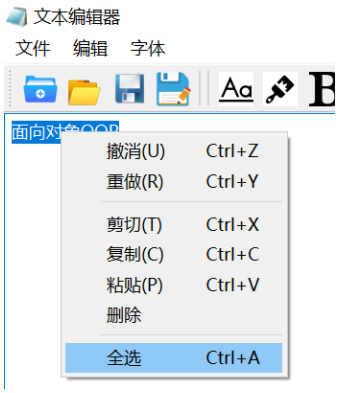






6.3.7 全选

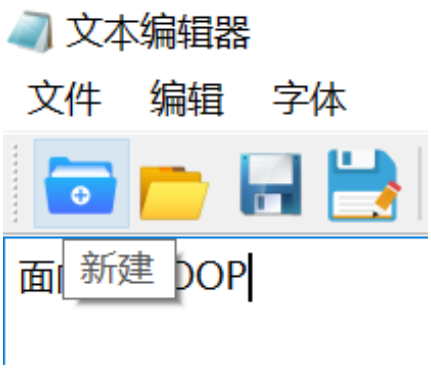
按下鼠标左键并拖动，选中所有文本内容，或使用快捷键"Ctrl + A"，所有文本背景为蓝色。全选后可进一步实现删除，复制，剪切等操作。

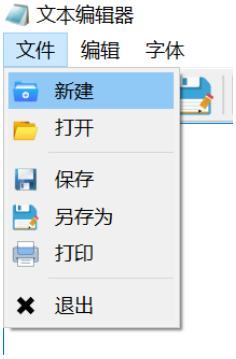


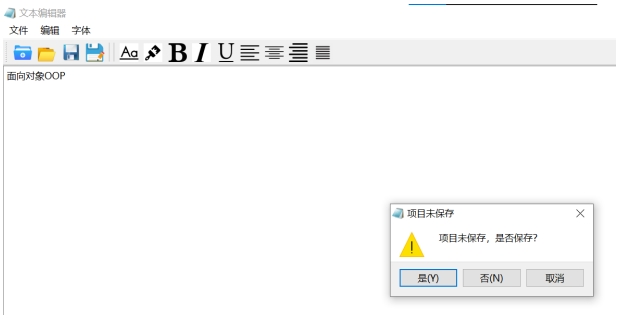
6.4 文件功能

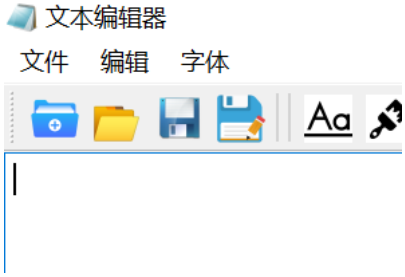
6.4.1 新建

点击工具栏的"新建"图标或菜单栏"文件"的下拉框中的"新建"后，弹出提醒框，提醒用户是否需要保存已有内容，若点击"是"则弹出保存框将原有文档保存后清空页面，若点击"否"将直接清空页面。



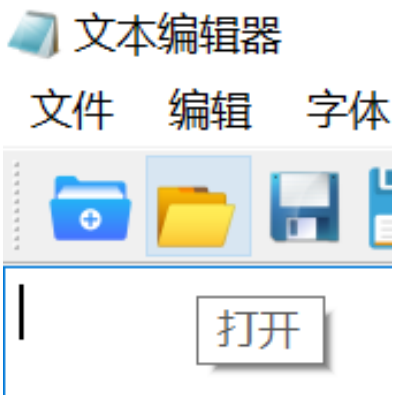


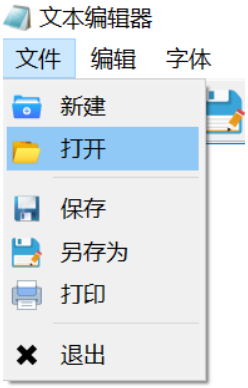


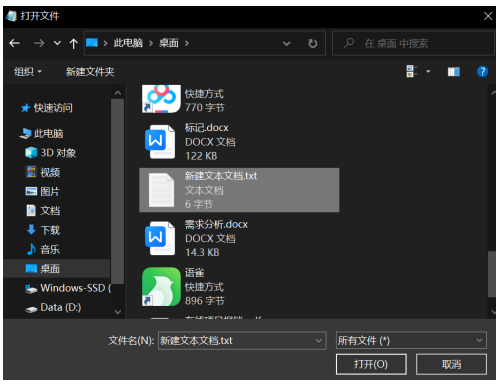


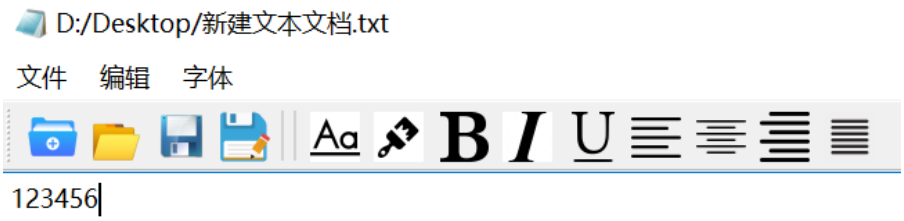
6.4.2 打开

点击"工具栏"的"打开"图标或"文件"下拉菜单中的"打开"选项，弹出选择框，选择本地文本文件，点击"打开"，即可打开本地文件，并显示文件内容和路径。



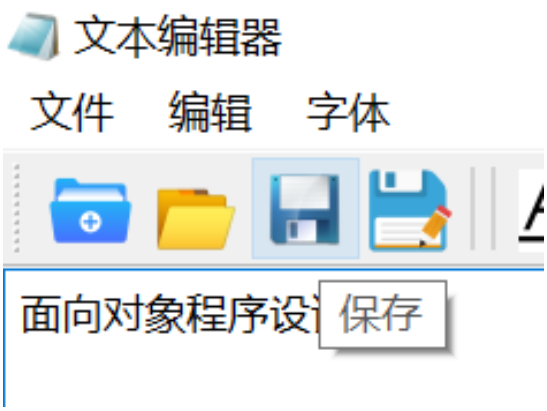


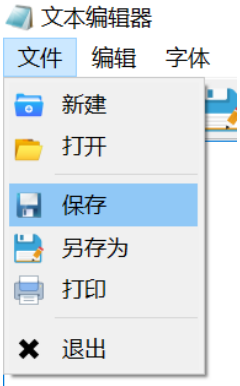


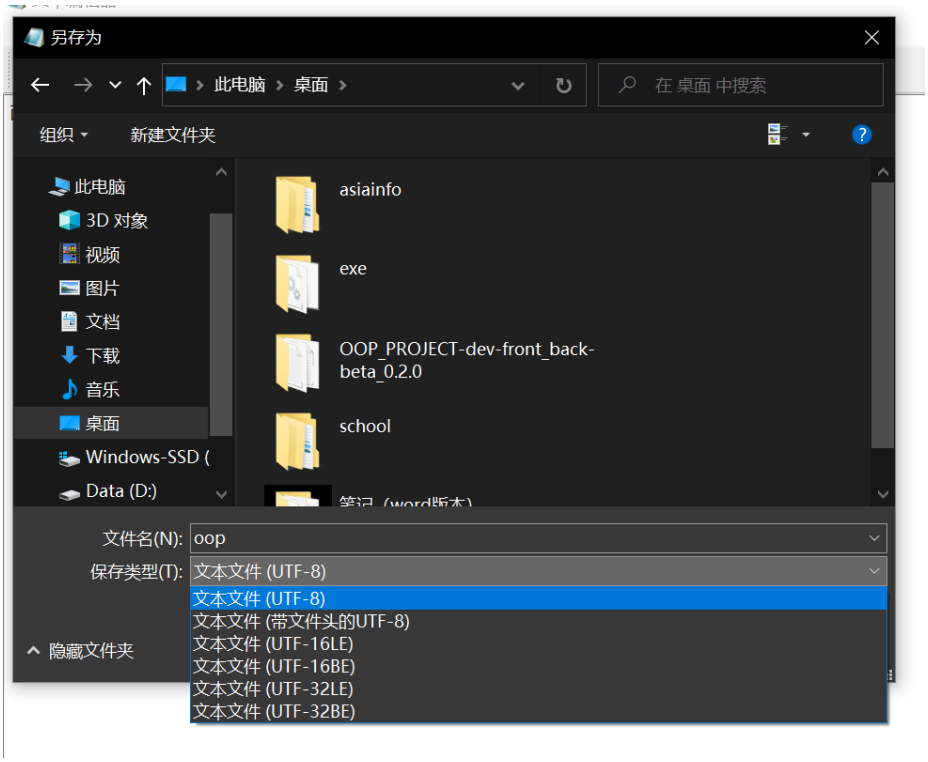


6.4.3 保存

点击"工具栏"的"保存"图标或"文件"下拉菜单中的"保存"选项，弹出选择框，输入文件名，选择保存路径和文件保存类型，点击"保存"，即可将文件保存至本地。

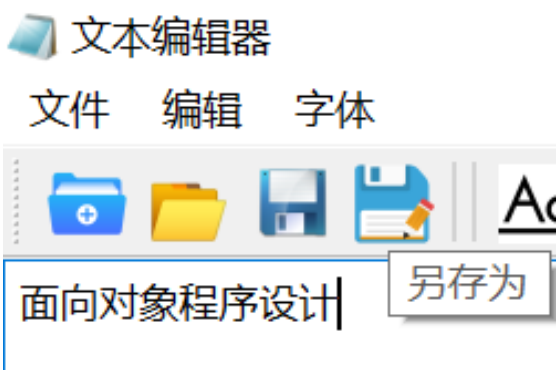


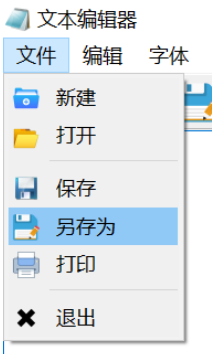


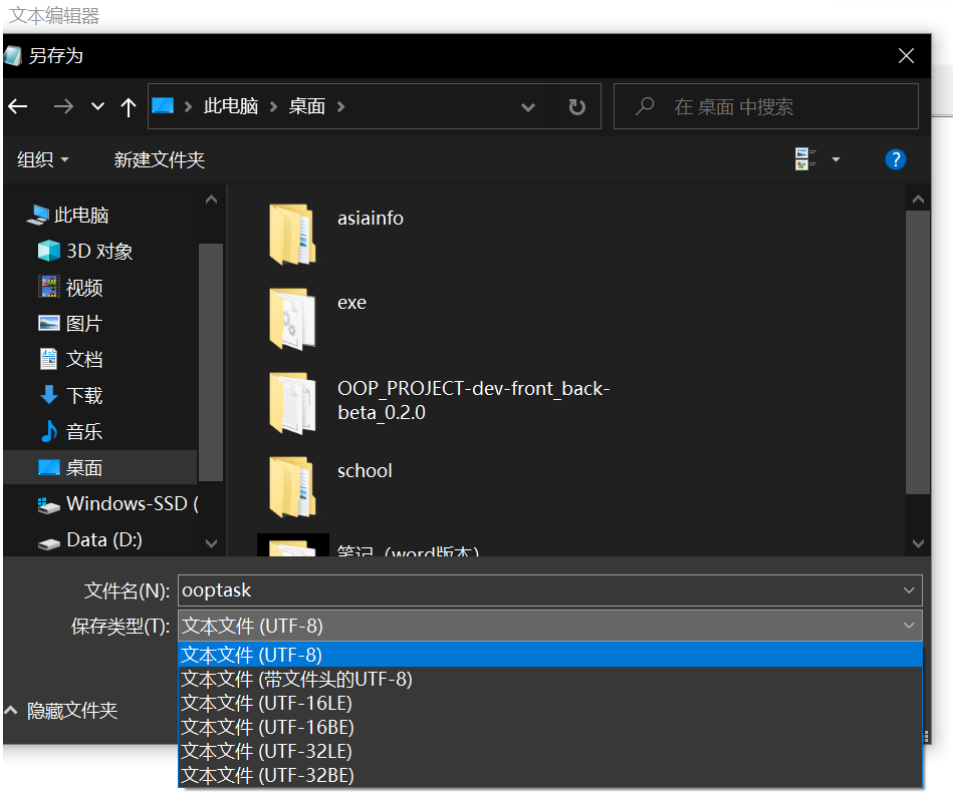


6.4.4 另存为

点击"工具栏"的"另存为"图标或"文件"下拉菜单中的"另存为"选项，弹出选择框，输入文件名，选择保存路径和文件保存类型，点击"保存"，即可将文件副本保存至本地。



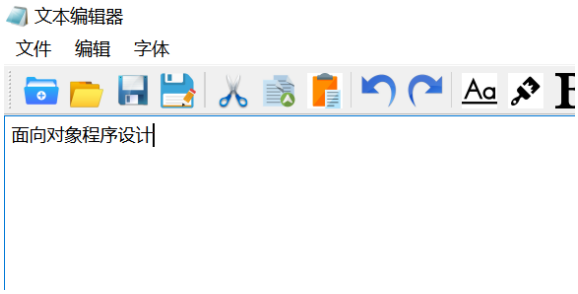




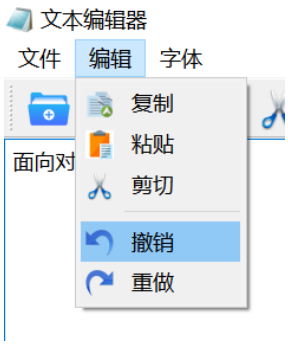
6.4.5 撤销和重做

点击"工具栏"的"撤销""重做"图标或"编辑"下拉菜单中的"撤销""重做"选项，可实现将操作恢复到上一步或者撤销前。

删除"程序设计"四个字：



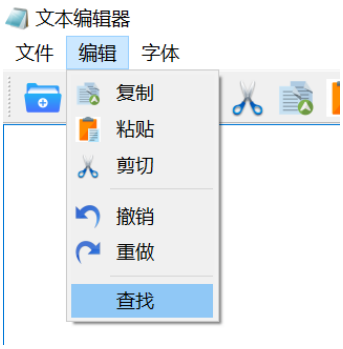
撤销删除：



重做：

6.4.6 搜索替换

点击菜单栏"编辑"下的"查找"选项，将弹出查找和替换窗口，在"查找"框中输入需要查找的文本，点击"查找下一个"，即可依次找出对应文本。在"替换"框中输入替换后的内容，点击"替换"，即可替换当前查找到的文本，点击"替换全部"，可替换所有与"查找"框中相同的文本。此外，还可勾选"在选中范围内查找"，实现在指定范围的文本内查找内容。

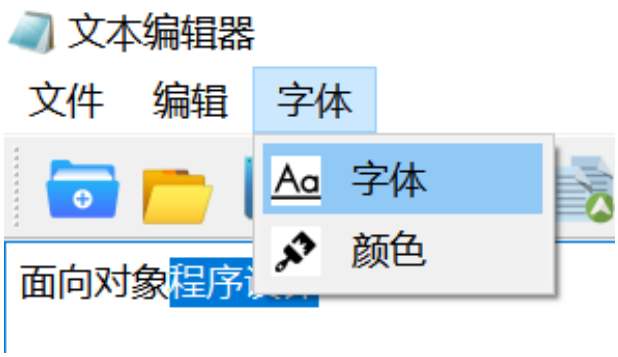


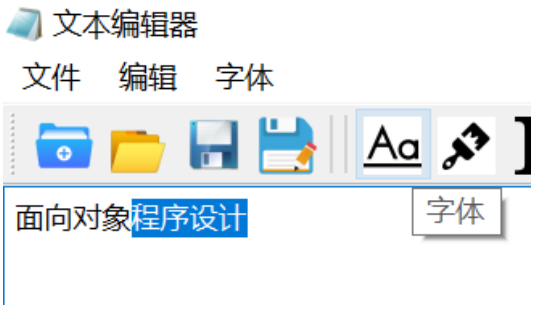
6.5 文本格式设置

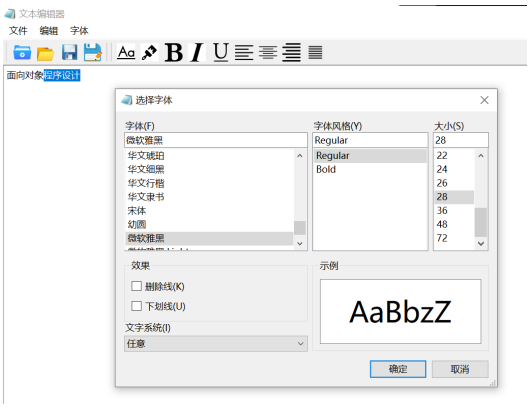
6.5.1 字体设置

6.5.1.1 字体大小和字体样式

鼠标选中需要调整的文本，点击"工具栏"的"字体"图标或者菜单栏"字体"下的"字体"，弹出窗口，可以选择宋体、微软雅黑等自定义字体样式，可以设置字体风格和字体大小，能够选择文字系统，还能添加下划线或删除线效果，最终效果图可在窗口右下角显示，点击确定即可调整字体大小和样式。







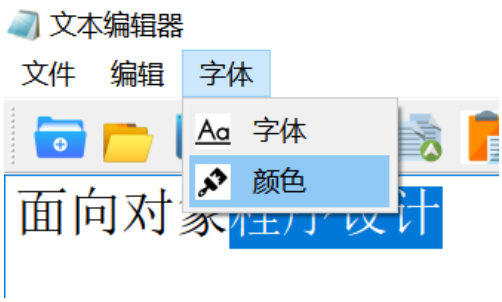
6.5.1.2 字形设置

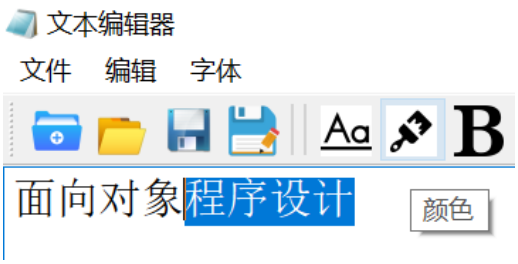
鼠标选中需要调整的文本，点击"工具栏"的"粗体"或"斜体"或"下划线"图标，可以使字体变粗，倾斜或添加下划线，再次点击图表可取消效果，多种字形效果可叠加。

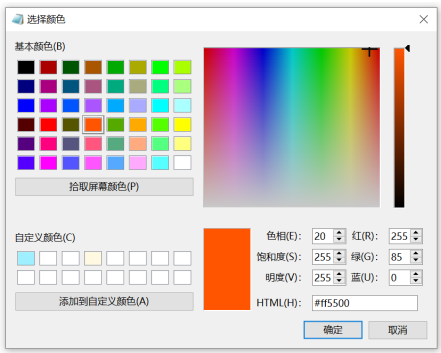
6.5.2 颜色设置

鼠标选中需要调整的文本，点击"工具栏"的"颜色"图标或者菜单栏"字体"下的"颜色"，弹出窗口，可以选择系统提供的基本颜色，拾取屏幕颜色，设置自定义颜色，还可以通过上下箭头微调，窗口右

下角显示预览效果和HTML（H）值，点击"确定"即可更改文字的颜色。

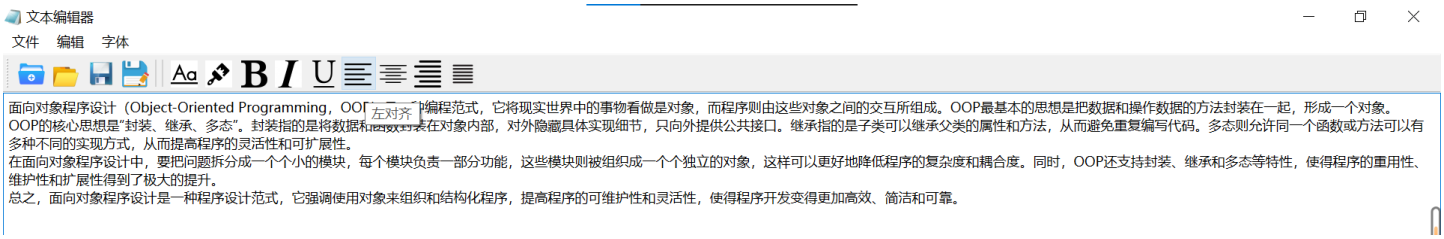


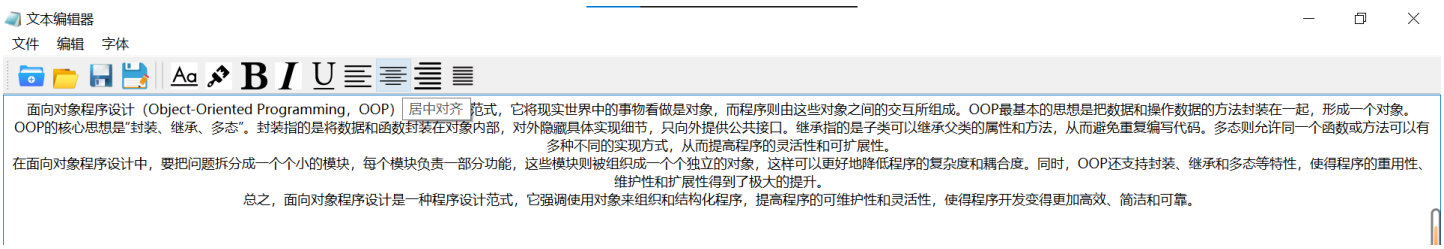


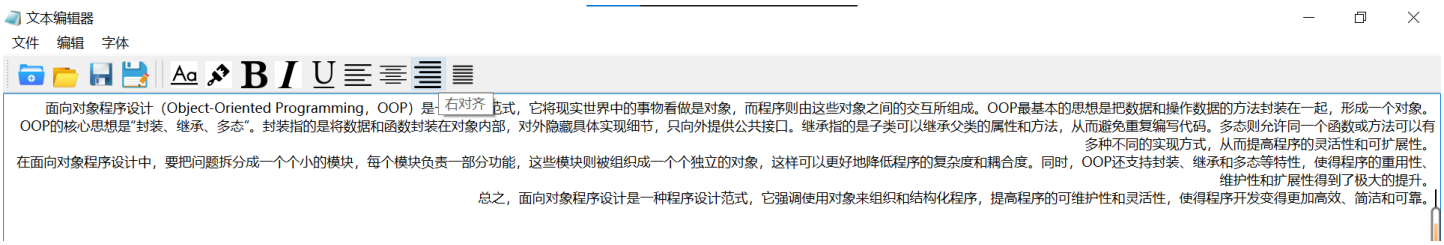


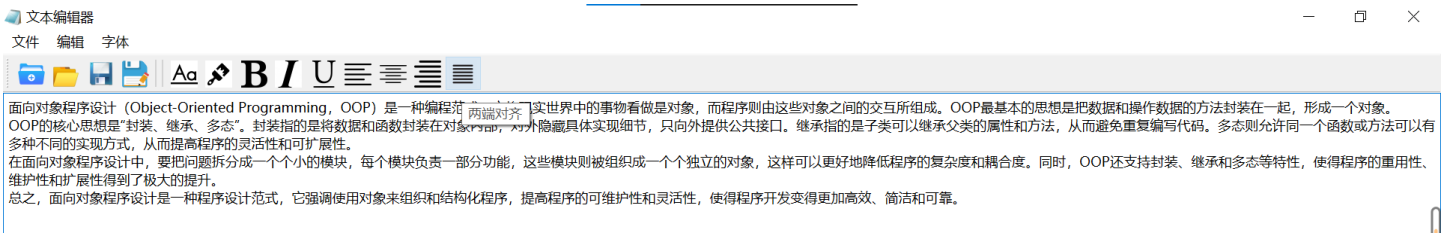
6.6 排版设置

鼠标选中需要调整的文本，点击"工具栏"的"左对齐"或"右对齐"或"居中对齐"或"两端对齐"图标，可设置对齐方式。可对部分文本或所有文本进行设置。



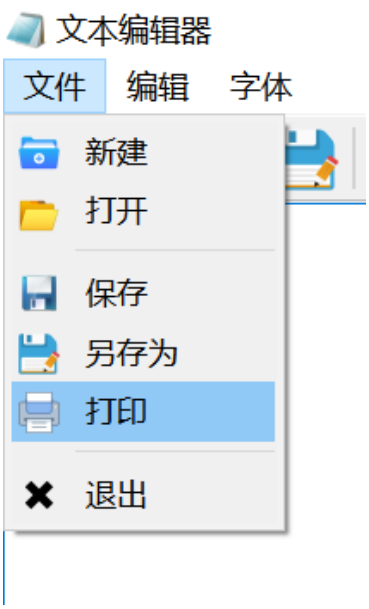


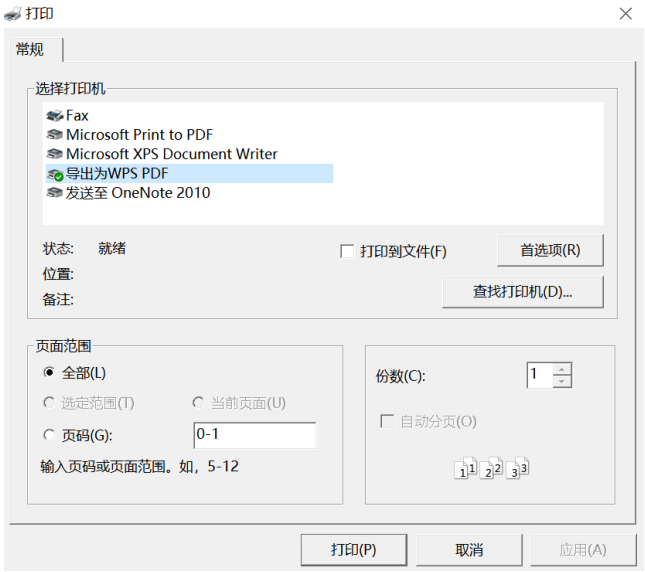




6.7 打印

点击菜单栏"文件"下的"打印，可将当前文档打印，也可导出为pdf文件。





6.8 退出软件

点击右上角的"关闭"图标或"文件"下拉菜单中的"退出"选项，将弹出窗口提醒用户保存文档，若点击"是"则弹出保存框将原有文档保存后退出软件，若点击"否"将直接退出软件。