面向对象程序设计课程大作业
——个人日记软件的设计实现

宋宇轩 2019K8009929042

一. 需求分析与进度安排

我想要实现的程序是一个日记软件，支持单篇日记的文字排版、存储和记录日期等功能，同时可以对多篇日记进行统一查看和检索。预想的界面如下图所示：

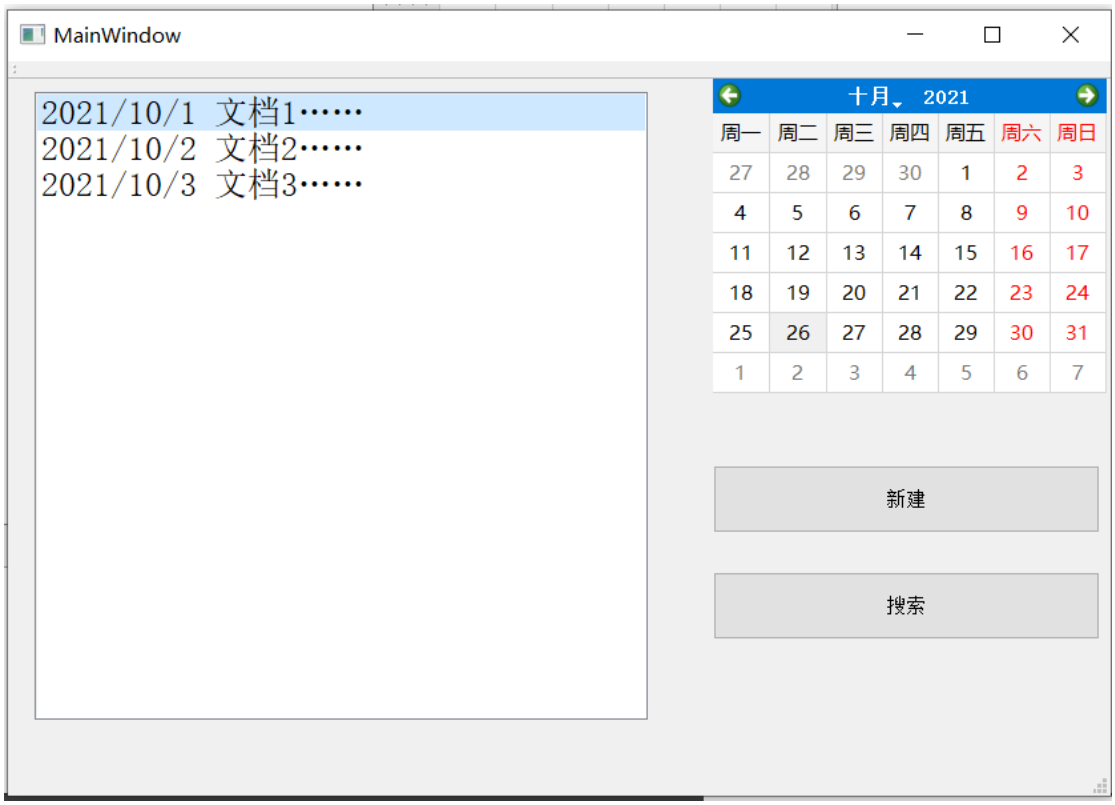

图 1 初始界面示意图

主界面主要可以分为 3 个部分：左侧按时间先后顺序显示目前已有的所有文档的简要信息，单击后即可打开对应文档。右上角是一个日历，右下角则为新建文档和按关键字搜索文档功能的按键。在单击日历中的某一天后，左侧的显示区会显示生成日期为当天的所有文档，如下图所示：

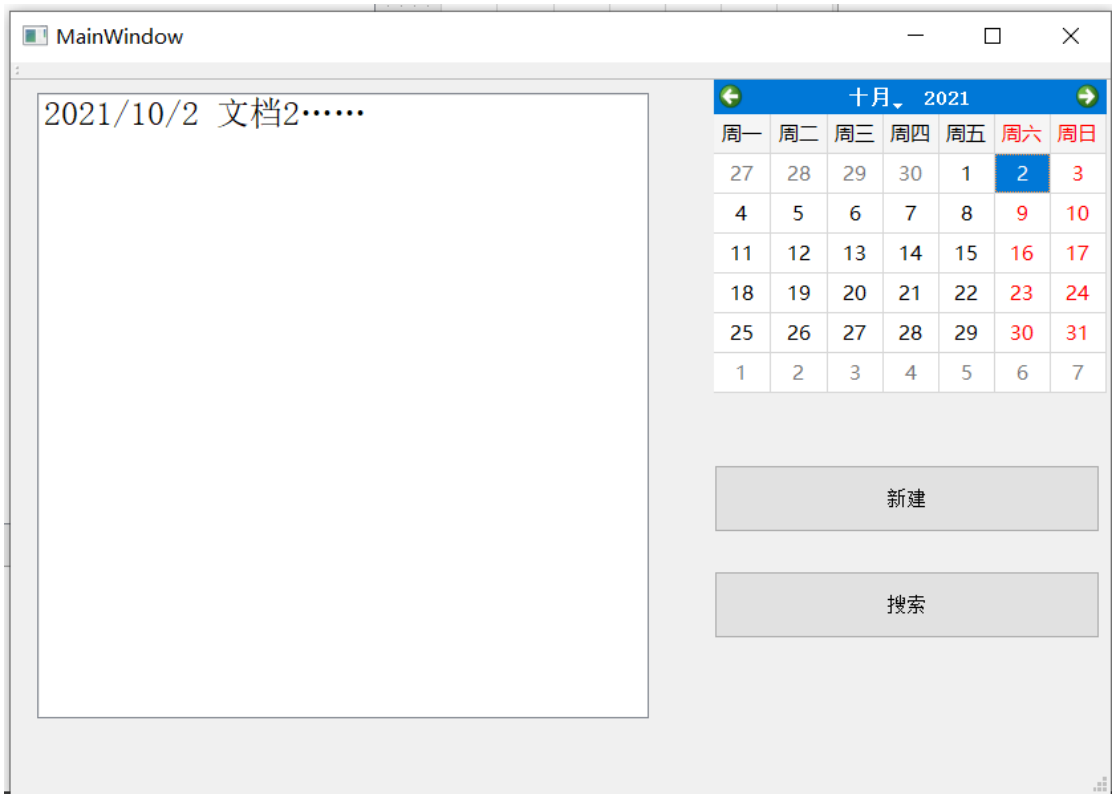

图 2 按日期检索文档示意图

新建文档或打开文档后的界面与 windows 的记事本应用界面类似，支持中英文（UTF-8 编码字符）的编辑、排版、复制粘贴和保存。同时删除文档也将在打开文档之后的子页面内实现。后续将根据时间和进度安排尝试增加实时自动保存，插入图片、视频和 emoji 表情图，支持云端存储，与其它设备互联同步等功能。

目前的进度安排为以三次提交课程作业的时限为节点，将整个学期划分为 3 个阶段。第一阶段确定主要设计目标和需求，借助网上的资料敲定基本目标的实现方法；第二阶段进行程序编写，完成基本功能的大致实现；第三阶段完善基本功能，确保基本功能正确实现，随后尝试加入更多高级功能。

二. 设计目标与思路

整个项目可以被分为两个模块：显示总体信息的主界面和提供文本编辑的子界面。主界面的目标状态如第一节所示，提供所有文档的总体展示，支持按日期检索和按关键字搜索，可以新建文档或打开文档。除此之外，还可以在初次打开文档时以只读模式展示，在选择编辑模式之后再启动文档的编辑。子界面将会是一个类似记事本的文本编辑器，用来书写、修改文档，可以支持中英文（UTF-8）的编辑，兼容鼠标操作（通过单击改变光标位置等），支持手动保存或自动保存，在关闭窗口时提示是否需要保存。除此之外，我还希望能支持撤销功能，实现复制粘贴功能（兼容系统本身的剪切板），为需要的功能增加快捷键（按下 ctrl+s 进行保存等），插入图片、视频或表情，统计字数，修改字体字号，实现文本内容搜索等功

能。

目前看来我需要定义三个类：文本类 Text 和目录类 Menu，分别用于对单个文档进行编辑和保存，以及对所有文档进行管理和检索；窗口控制类 Window 用于显示文本，接收键盘、鼠标的操作并调用对应的处理函数。

为了实现文档的编辑，文本类 Text 内需要用来记录文字的缓冲区 text_buff；我计划将每一篇生成的文档都使用.txt 类型文件存储在本地，因此 Text 内需要记录文档所在地址 text_addr；除此之外还需要记录文档的基本信息，例如生成日期 creat_date，总字数 tot_word；为了实现提醒保存，还需要变量 is_change 来记录是否进行了修改。

对文档的操作包含新建，打开，编辑，保存，删除，因此需要 5 个公共的成员函数来实现它们。新建函数 new_text()即是 Text 类的构造函数，负责设置存储地址，初始化生成日期、缓冲区及光标位置等基本信息。打开函数 open_text()根据类中的 text_addr 变量寻址并打开对应文件，将数据写入缓冲区 text_buff 以供编辑。编辑函数 edit_text()需要接收控制部件传来的指令码，并依据指令码对缓冲区做相应的修改（插入字符、删除字符等）。保存函数 save_text()根据原有地址或新输入的地址，将缓冲区中的内容输出到对应地址的文件中。对于删除操作 del_text()，我计划单独设置一个“已删除”文件夹，删除文档时将文档从原地址删除，将其放入这个文件夹之中并修改地址变量，再次删除时才调用析构函数。为此，类中还需要一个成员变量 is_delete 来判断是否已经删除过一次。同时为了支持目录类的管理和检索，还需要输出生成时间和所在地址的函数 get_date()和 get_addr()。

目录类 Menu 将包含一个 Text 类型的优先队列，按照生成时间的先后顺序排序，同时记录目前文档的总数 tot_file。成员函数将包括增加文档（新文档入队），删除文档（从队列中删除对应文档），按时间检索和按文本检索。按时间检索只需要枚举队列里的文档，找到生成时间相符的即可。对于按文本内容检索，我目前的想法是将现有的所有文本依次打开，检查内容中有无检索关键词，之后再关闭文本。但这样需要多次访问内存，对于执行速度可能会有影响，未来可以进一步寻找更优秀的实现方法。

窗口控制类 Window 则包含光标位置 cursor_x 和 cursor_y，以及当前需要执行的指令码 instruction。成员函数有用来接收键盘、鼠标输入并确定目标操作的 check_inst()，根据操作调用相应处理函数的 handle_inst()，打印当前文档缓冲区的 print_buff()，以及移动光标的 move_cursor()。

综上所述，三个类的 UML 图如下所示：

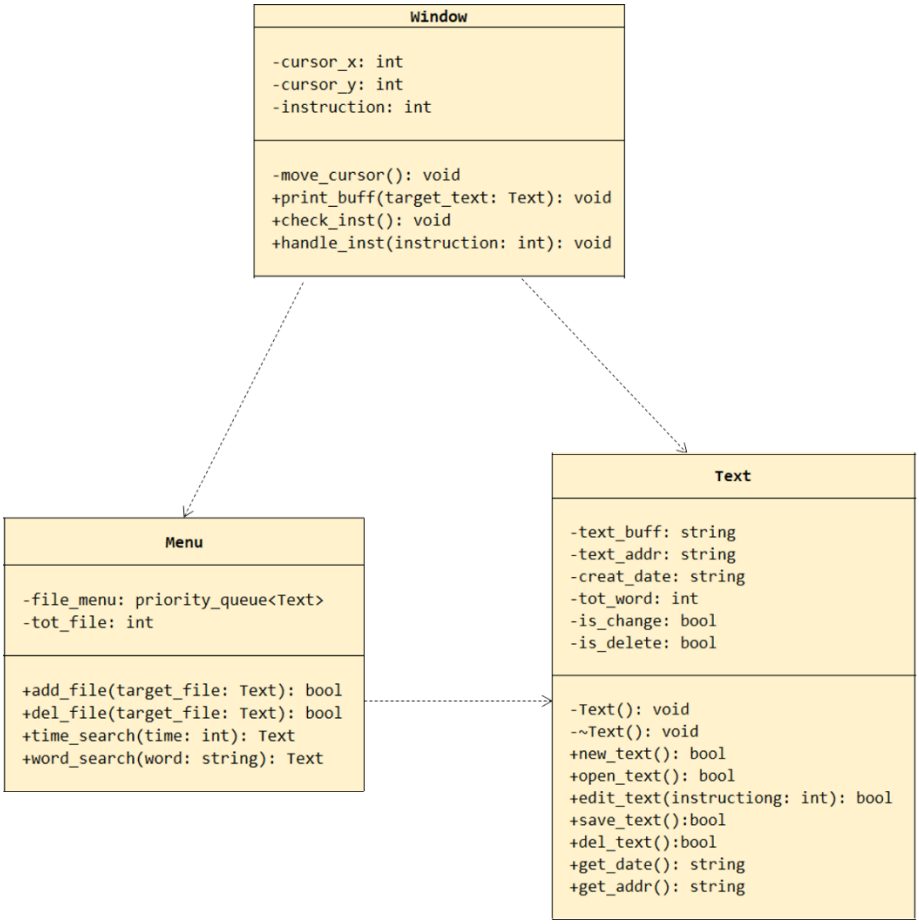

图 3 程序设计 UML 图

三. QT 简介与窗口实现分析

在介绍真正的程序设计之前，首先要说明的一点是我使用了 C++语言和 QT 开发库来进行设计。使用 C++语言而非 JAVA 语言主要是因为我对 C++语言相对更为熟悉，不需要重新学习一门新的编程语言，同时设计这样一个规模稍大一些的程序也可以加深我对 C++的理解。使用 QT 库则可以更方便的实现图形化界面，同时对鼠标、键盘操作的支持也更简单，免去了查询 window API 的麻烦。

QT 设计中可以直接拖动功能控件来图形化地实现弹出窗口的 UI 设计，如下图所示：

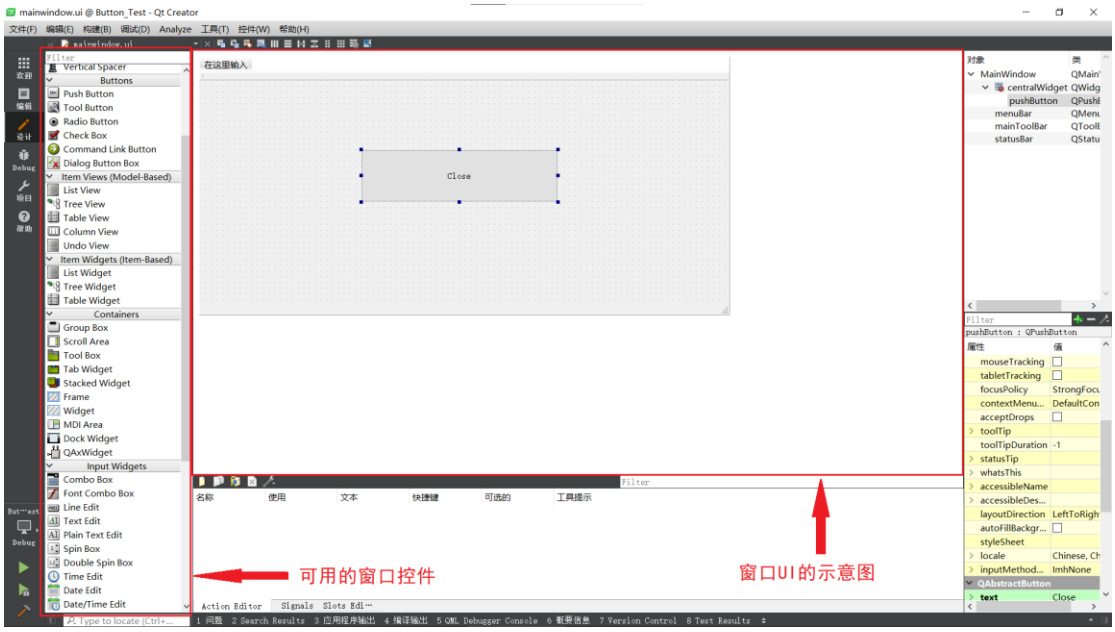

图 4 使用 QT 设计弹出窗口 UI 示意图

QT 软件会根据设计出的界面自动创建一个 UI 类，包含先前加入窗口设计的各类按键、菜单、图表等控件的实现。这些控件都是 QT 已经封装好的类，并提供了需要的接口供设计者使用。以单击窗口中的按钮来关闭窗口为例：从零开始设计需要支持输出图形化界面，检测鼠标指针位置，检测单击动作，设计关闭窗口函数来停止运行窗口等功能，而这些功能大多需要调用 windows API 实现，这是我非常陌生的一块内容；而使用 QT 进行设计只需要在已有的 on_pushButton_clicked()函数中调用 QT 封装好的 close()函数，即可实现单击按钮时关闭窗口的功能，设计者不需要再担心更底层的实现，从而节省时间和精力。事实上，在正式开始编写程序代码之前，我就可以利用 QT 设计生成图 1 和图 2 那样的窗口了，随后只需要对各类控件的功能进行编程实现即可，这也从侧面证明了使用 QT 对于设计效率的巨大提升。

不过尽管利用 QT 进行设计如此便捷，多了解一下窗口和控件的具体实现也是有帮助的。还是以单击窗口中的按钮来关闭窗口为例，这一项目最重要的代码文件有 4 个：main.cpp, ui_mainwindow.h, mainwindow.h 和 mainwindow.cpp。

main.cpp 实现的功能非常简单：定义一个 QApplication 类的对象 a 用于运行程序，一个 MainWindow 类的对象 w 用于显示图形化界面，具体代码如下：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

其中 QApplication 的父类为 QGuiApplication，而 QGuiApplication 则继承自基类 QCoreApplication，其主要作用为设置软件图标、接收鼠标和键盘的输入、弹出提示弹窗等

较为底层的操作实现。MainWindow 的父类为 QMainWindow，再往上的父类为 QWidget，QWidget 多继承自 QPaintDevice 类和 QObject 类，其中 QPaintDevice 控制弹出窗口的位置、大小、颜色等特征，QObject 则负责将一个信号（比如鼠标的单击）与对应的槽函数（类似处理函数）进行连接，保证对应操作的实现。

信号可以比作软件的中断，例如，按钮控件 A 被点击时就会广播“被单击”这一事件对应的信号。控件中可以有很多信号，每个信号代表不同的事件，而我们只需要向程序指定这一控件对哪些事件感兴趣即可。对于不感兴趣的事件的信号，程序就会忽略掉，不作响应。槽函数则是程序接收并响应某信号之后，需要执行的操作。QT 提供了一些已经设计好了的槽函数，设计者也可以自定义新的槽函数。

ui_mainwindow.h 中声明了 UI_MainWindow 类，包含了对于窗口界面的 UI 设计：

```
QWidget *centralWidget;
QPushButton *pushButton;
QMenuBar *menuBar;
QToolBar *mainToolBar;
QStatusBar *statusBar;
```

其中中心窗口、菜单栏、工具栏和状态栏是每个窗口 UI 都有的，按键控件*pushButton 则是依照我的设计而自动添加的。

此外这个类中还包含两个成员函数：setupUI 和 retranslateUI。retranslateUI 的作用是修改窗口与控件的显示文本（即窗口的标题、按键上显示的文字等）：

```
void retranslateUi(QMainWindow *MainWindow)
{
   MainWindow->setWindowTitle(QCoreApplication::translate("MainWindow", "MainWindow", nullptr));
   pushButton->setText(QCoreApplication::translate("MainWindow", "Close", nullptr));
}
```

setupUI 可以分为三个部分：第一部分是设置窗口与控件的大小、位置、命名等特征，第二部分是调用 retranslateUI 函数设置窗口与控件的显示文本，第三部分是调用 QMetaObject::connectSlotsByName(MainWindow);函数来将当前窗口中的控件与其对应的槽函数连接起来：

```
void setupUi(QMainWindow *MainWindow)
{
    if (MainWindow->objectName().isEmpty())
        MainWindow->setObjectName(QString::fromUtf8("MainWindow"));
    MainWindow->resize(918, 446);
    centralWidget = new QWidget(MainWindow);
    centralWidget->setObjectName(QString::fromUtf8("centralWidget"));
    pushButton = new QPushButton(centralWidget);
    pushButton->setObjectName(QString::fromUtf8("pushButton"));
    pushButton->setGeometry(QRect(280, 120, 341, 91));
    MainWindow->setCentralWidget(centralWidget);
    menuBar = new QMenuBar(MainWindow);
    menuBar->setObjectName(QString::fromUtf8("menuBar"));
    menuBar->setGeometry(QRect(0, 0, 918, 26));
    MainWindow->setMenuBar(menuBar);
    mainToolBar = new QToolBar(MainWindow);
    mainToolBar->setObjectName(QString::fromUtf8("mainToolBar"));
    MainWindow->addToolBar(Qt::TopToolBarArea, mainToolBar);
```

```
    statusBar = new QStatusBar(MainWindow);
    statusBar->setObjectName(QString::fromUtf8("statusBar"));
    MainWindow->setStatusBar(statusBar);

    retranslateUi(MainWindow);

    QMetaObject::connectSlotsByName(MainWindow);
}
```

mainwindow.h 文件中主要声明了表示图形化窗口的 MainWindow 类，其中除了构造函数和析构函数之外，还声明了需要用到的槽函数，并定义了对象*ui 来表示当前窗口：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void on_pushButton_clicked();

private:
    Ui::MainWindow *ui;
};
```

mainwindow.cpp 用于定义 MainWindow 类里的成员函数，这也是利用 QT 设计程序时主要需要编写的部分。除了自带的构造函数和析构函数以外，这个程序里增加了按键与一个槽函数 on_pushButton_clicked()，这个函数调用 QT 提供的 close()函数来关闭窗当前口，具体代码如下：

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}


void MainWindow::on_pushButton_clicked()
{
    close();
}
```

以上就是 QT 在设计好一个窗口后自动生成的主要代码了，接下来需要实现的就是根据需求安排控件和信号，并编写相应的槽函数来实现对应的功能。

四． 程序外观与功能实现

目前实现的程序外观与先前预想的程序界面基本一致：

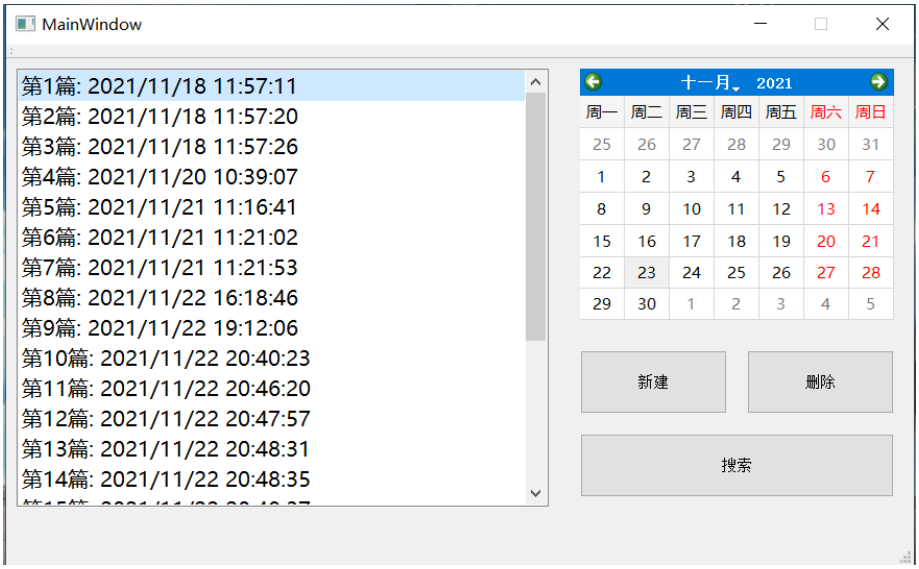

图 5 初始界面示意图

运行程序后立即弹出的初始界面（以下称为主界面）大小固定为 870×500（像素），主体分为 3 个部分：左侧为文档信息的显示区，右上角是一个日历，右下角有 3 个按键。

在新建或打开某个文档后，文档的编辑界面（以下称为子界面）就会弹出：

图 6 文档编辑界面示意图

子界面与 Windows 自带的记事本程序外观类似，大小固定为 600×620（像素）。子界面

的主体部分为文本编辑器，上方有 3 个下拉菜单，其中各自包含了一些操作或选项。

就目前的完成进度而言，主界面左侧的显示区会按时间先后顺序显示目前已有的所有文档的编号和生成时间，双击某一项后即可打开其对应的文档。右上角的日历目前仅能显示日期，计划在之后添加一项新功能：在单击日历中的某一天后，左侧的显示区会显示生成日期为当天的所有文档。右下角的 3 个按键中目前只完成了新建文档，删除文档和搜索文档的功能会在之后加以实现。文档的默认存储位置为程序所在目录的 diary_file 文件夹中，存储类型为.txt 格式。

子界面的文本编辑部分支持中英文等常见字符的编辑、删改和保存，可以通过鼠标或键盘改变光标位置、选中部分内容，并支持通过快捷键或右键菜单中的选项来实现全选、剪切、复制、粘贴或撤销等功能。新建文档时会在文本编辑区的最上方自动写入该文档的编号和创建时间。

子界面上方的 3 个下拉菜单在单击后会展示其中包含的操作或选项，具体内容如下图所示：

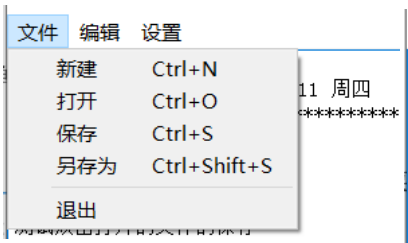

图 7 “文件”菜单示意图

图 8 “编辑”菜单示意图

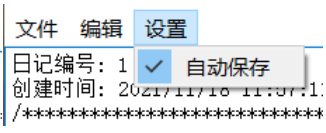

图 9 “设置”菜单示意图

“文件”菜单包含 5 个操作，其中新建操作和主界面的新建按键功能一致，都会新建一个空白文档并以子界面的形式呈现，文档开头会自动写入日记编号和创建时间：

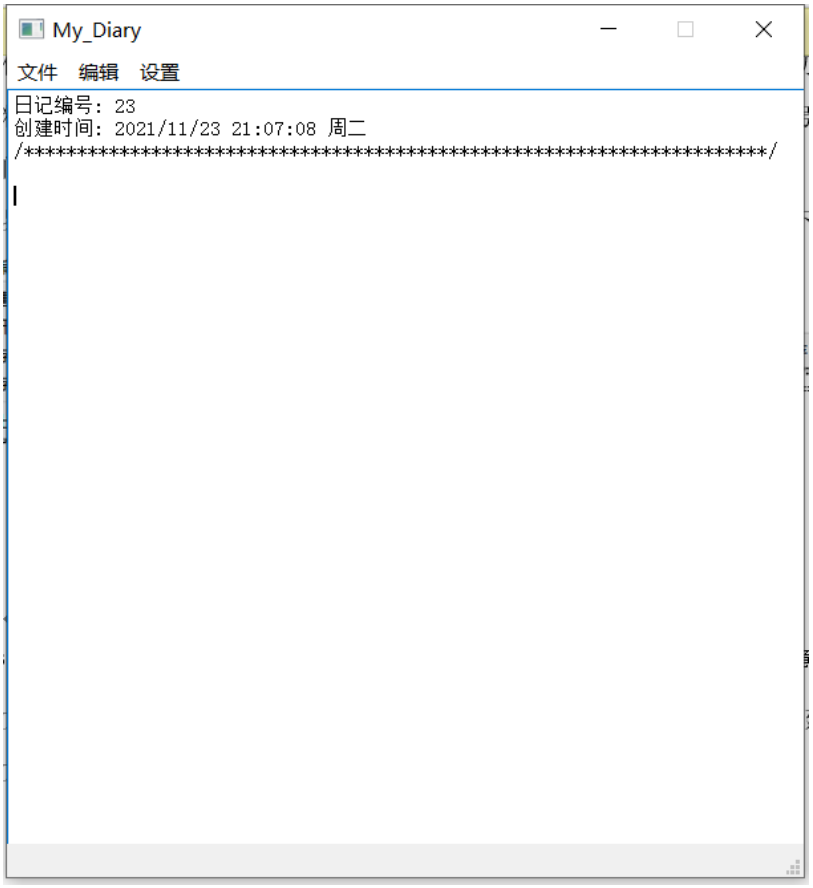

图 10 新建文档界面示意图

打开操作会弹出地址检索窗口，可以选定任意文档并打开：

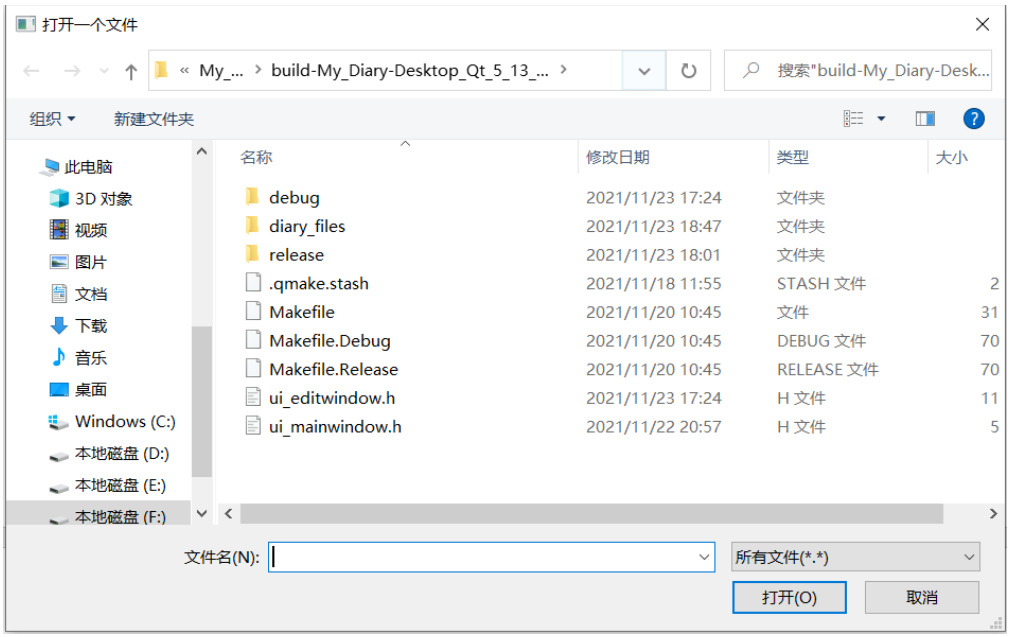

图 11 打开文档界面示意图

保存操作即为保存文档到当前地址；另存为操作还没有实现；退出操作与单击右上角的×按键功能一致，会在没有修改过文档或已经保存了当前文档后直接关闭子界面，在文档被修改过且没有保存时提示是否需要保存：

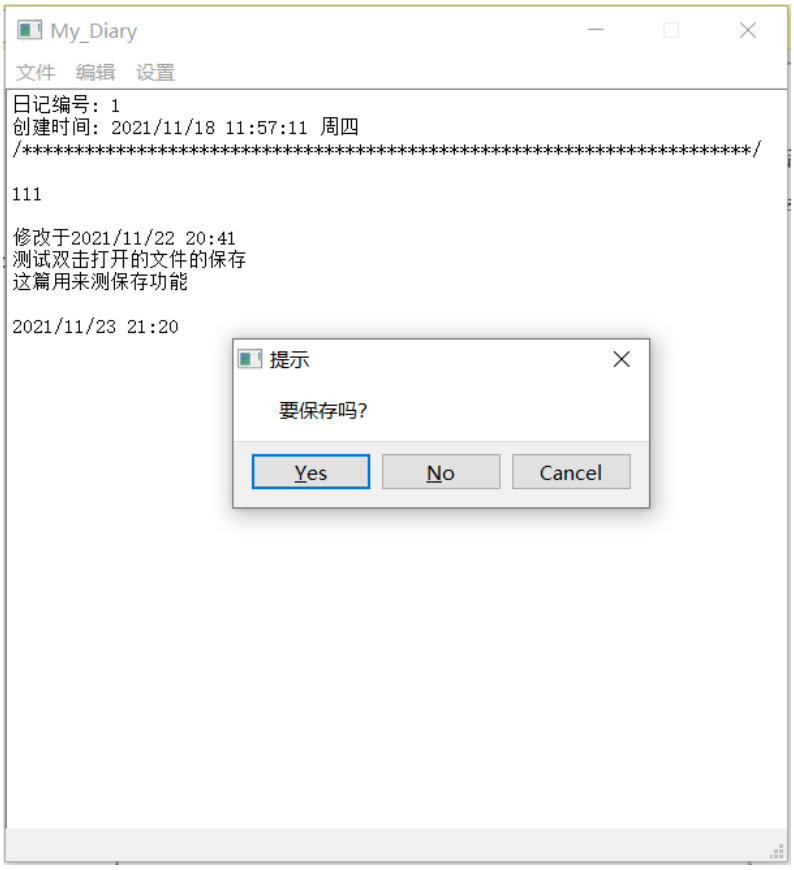

图 12 退出文档时的提示界面示意图

“编辑”菜单包含 9 个操作，目前这些操作在被单击时都是不起作用的，但撤销、剪切、复制、粘贴、删除和全选可以通过快捷键或右键菜单来完成，恢复、查找和替换操作目前还未实现。

“设置”菜单仅包含 1 个选项：自动保存。在单击勾选自动保存选项时，文档会自动执行一次保存操作，并在之后的每一次修改文档后都自动进行一次保存；在再次点击自动保存选项（取消勾选）之后，文档的修改将不再自动进行保存。

五． 代码分析与功能实现

项目的总体 UML 类图如下所示：

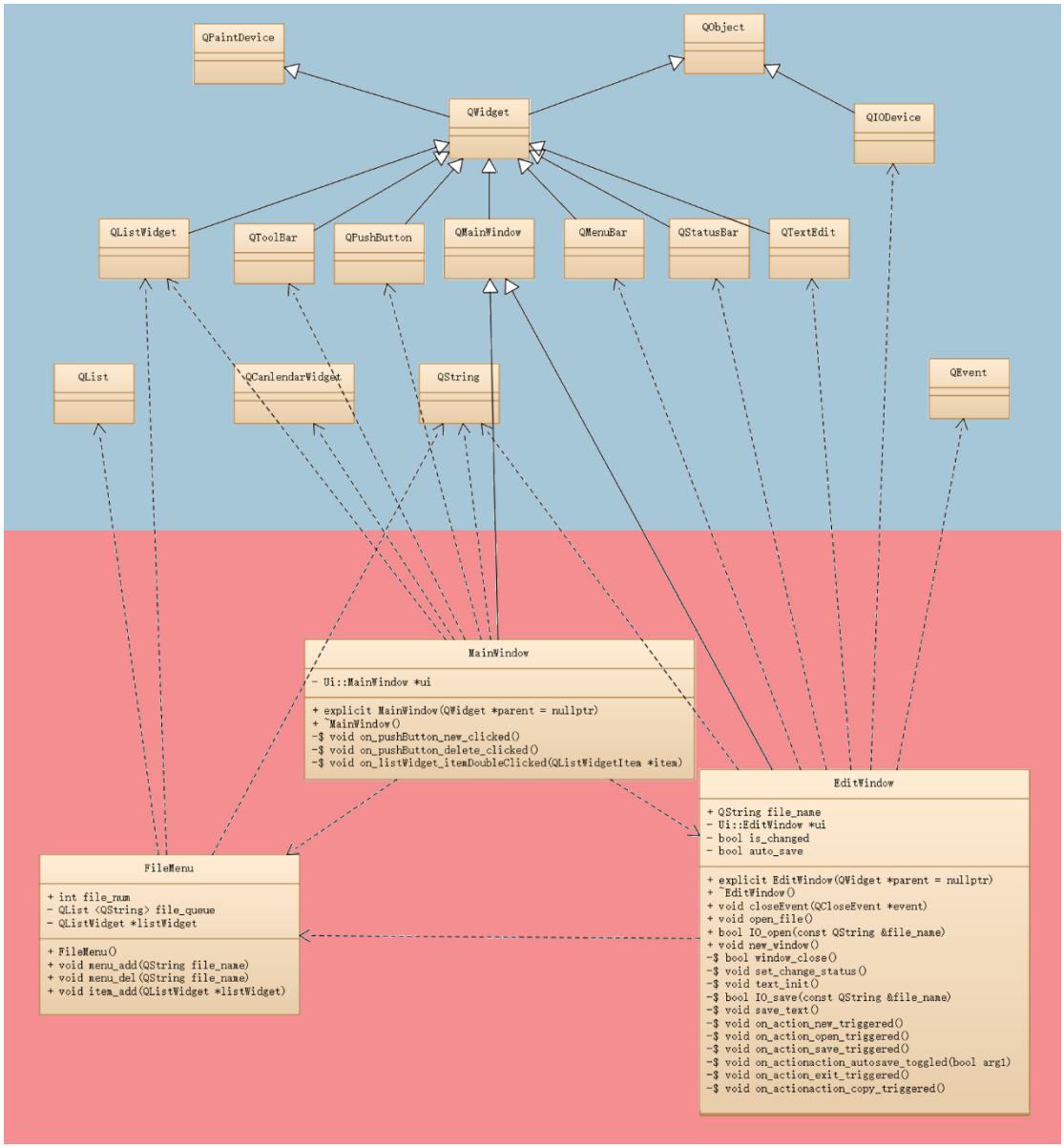

图 13 项目总体 UML 类图

得益于 QT 库中大量定义完整的类，我自己实际进行实现的类只有 3 个，它们之间的关系也很简单。如图 9 所示，上半部分展示了项目中使用的部分 QT 库自带的类的大致情况，下半部分则为我自己实现的 3 个类：MainWindow, EditWindow 和 FileMenu，分别用于控制主界面、控制子界面，以及管理文档与显示区。类图中的-$前缀表示这个成员函数是一个槽函数，它可以与信号连接起来，在对应信号发射时被自动调用。

MainWindow 类和 EditWindow 类中都包含*ui 成员变量，用于表示图形化的界面；EditWindow 类中的 file_name 变量记录目前打开的文档的名称，is_changed 变量记录文档是否发生了修改（是否需要保存），auto_save 变量记录目前是否需要执行自动保存。FileMenu 类中的 file_num 变量用于记录目前的日记文档总数。file_queue 是一个链表，按照创建时间的先后顺序存储所有文档的名称。listWidget 是 QListWidget 类的对象，用于控制主界面左侧的文档显示区。

目前该项目的程序所在目录下有一个命名为 dairy_file 的文件夹，里面存有用于记录文档数目和题目的 menu.txt，以及所有的日记文档。在整个项目开始运行时就会新建一个 FileMenu 类的全局对象 menu，用于管理所有的文档。menu 的构造函数会打开 menu.txt 文件，从中读取文档的数量和所有文档的名称并记录下来，具体代码如下所示：

```
FileMenu::FileMenu()
{
    this->listWidget = NULL;
    QFile fp("./diary_files/menu.txt");
    if(fp.open(QIODevice::ReadOnly | QIODevice::Text))
    {
        QTextStream stream_in(&fp);
        QString line;
        int line_num = 1;
        while(stream_in.readLineInto(&line))
        {
            if(line_num == 1)
            {
                this->file_num = line.toInt();    //menu.txt 的第一行写有文档总数
            }
            else
            {
                this->file_queue += line;         //此后每一行对应一个文档的名称
            }
            line_num++;
        }
        fp.close();
    }
}
```

项目运行时会新建 MainWindow 类的对象 w，它的构造函数会生成图形化的主界面，并将主界面的大小设置为固定的 870×500（像素）：

```
    ui->setupUi(this);
    setFixedSize(870,500);
```

随后会向全局对象 menu 中传入主界面左侧的文档显示区控件的地址，并向其中添加所有文档的信息：

```
    menu->item_add(ui->listWidget);
```

FileMenu::item_add(QListWidget *listWidget)函数会将 menu 中的成员变量 listWidget 指向主界面左侧的文档显示区，将其清空并输入所有已有文档的信息：

```
void FileMenu::item_add(QListWidget *listWidget)
{
    if(this->listWidget == NULL)
    {
        this->listWidget = listWidget;
    }
    this->listWidget->clear();
    for(int i = 0; i < this->file_num; i++)
    {
       QString string = this->file_queue.at(i);
       string.replace(4, 1, "/");
       string.replace(7, 1, "/");
       string.replace(10, 1, " ");
```

```
            string.replace(13, 1, ":");
            string.replace(16, 1, ":");
            QString file_name = string.left(19);    //获得目标格式的生成时间
            QString target_str = "第" + QString::number(i+1) + "篇: " + file_name;
            this->listWidget->addItem(target_str);
        }
    }
```

这样一来，主界面的所有初始化就都完成了。

主界面的新建按键和子界面“文件”菜单下的新建操作都可以实现新建文档，不同之处在于主界面需要新建一个 EditWindow 类的对象 New_window，用于生成一个子界面：

```
void MainWindow::on_pushButton_new_clicked()
{
    EditWindow *New_window = new EditWindow;
    New_window->new_window();
}
```

其中子界面的构造函数会生成图形化的界面，将其大小设置为固定的 600×600（像素），将文本编辑区自带的 textChanged()信号与槽函数 set_change_status()连接起来，并初始化成员变量：

```
EditWindow::EditWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::EditWindow)
{
    ui->setupUi(this);
    setFixedSize(600,620);
    connect(ui->textEdit,SIGNAL(textChanged()),this,SLOT(set_change_status()));
    is_changed = 0;
    auto_save = 0;
}
```

而子界面中不会生成新的子界面，而是直接修改当前界面的文本编辑区：

```
void EditWindow::on_action_new_triggered()
{
    new_window();
}
```

new_window()函数是一个公有成员函数，会调用私有成员函数 text_init()来初始化文本，然后生成图形化的子界面：

```
void EditWindow::new_window()
{
    text_init();
    show();
}
void EditWindow::text_init()
{
    QDateTime *DateTime = new QDateTime(QDateTime::currentDateTime());
    QString date_str = DateTime->toString("yyyy/MM/dd hh:mm:ss ddd");
    QString num_str = QString::number(menu->file_num+1);
    QString str = "日记编号: "
                + num_str
                + "\n 创建时间: "
                + date_str
                +
```

```
"\n/*********************************************************************/\n\n";
        ui->textEdit->setText(str);
        ui->textEdit->moveCursor(QTextCursor::End, QTextCursor::MoveAnchor);

        file_name = DateTime->toString("yyyy_MM_dd_hh_mm_ss") + ".txt";
        //文档名即为生成时间
        is_changed = 0;
    }
```

文本编辑通过 QT 已有的控件 QTextEdit 实现，支持中英文字符的编辑与复制、粘贴、撤销等功能。

打开文档同样有两种方式：双击主界面左侧显示的某个文档，或是在子界面单击“文件”菜单中的打开操作。主界面打开的会是已有的日记文档；子界面则会进行地址检索，打开选定的文档，因此可以打开可找到的任意文件，但编辑器不一定兼容目标文件的格式。

两种打开方式的核心函数都是 EditWindow::IO_open(const QString &file_name)，它会根据传入的文件名打开文件，并将文件中的所有内容写入文本编辑区：

```
bool EditWindow::IO_open(const QString &file_name)
{
    QFile fp(file_name);
    if (!fp.exists())
        return false;
    if (!fp.open(QIODevice::ReadOnly | QIODevice::Text))
        return false;
    QTextStream Stream(&fp);
    ui->textEdit->setPlainText(Stream.readAll());
    fp.close();
    is_changed = 0;
    show();

    return true;
}
```

不同之处在于，主界面需要进行一些操作来获取目标文档的正确目录和名称，并新建一个子界面：

```
void MainWindow::on_listWidget_itemDoubleClicked(QListWidgetItem *item)
{
    QString tmp = item->text();
    QString file_name = tmp.right(19);
    file_name.replace(4, 1, "_");
    file_name.replace(7, 1, "_");
    file_name.replace(10, 1, "_");
    file_name.replace(13, 1, "_");
    file_name.replace(16, 1, "_");
    file_name += ".txt";
    EditWindow *New_window = new EditWindow;
    New_window->file_name = file_name;
    New_window->IO_open("./diary_files/"+file_name);
}
```

子界面则要先弹出对话框进行地址检索，再利用得到的目录尝试打开文件：

```
void EditWindow::on_action_open_triggered()
{
    open_file();
```

```
}
void EditWindow::open_file()
{
    QString curPath=QDir::currentPath();
    QString dlgTitle="打开一个文件";
    QString filter="所有文件(*.*);;文本文件(*.txt)";
    file_name=QFileDialog::getOpenFileName(this,dlgTitle,curPath,filter);
    if (file_name.isEmpty())
        return;
    if(!IO_open(file_name))
    {
        QMessageBox::warning(NULL,"警告","文件打开失败!");
    }
}
```

保存文档主要通过 EditWindow::save_text()函数实现，大体分为修改文档和修改目录两部分。EditWindow::IO_save(const QString &file_name)函数通过传入的文档名新建或打开文档，将当前子界面的文本编辑器内的内容写入其中；FileMenu::menu_add(QString file_name)函数检查当前文件队列中是否包含传入的文档，并对 menu.txt 进行相应修改；最后由 FileMenu::item_add(QListWidget *listWidget)函数对主界面的文档显示区进行修改：

```
void EditWindow::on_action_save_triggered()
{
    save_text();
}
void EditWindow::save_text()
{
    IO_save(file_name);
    menu->menu_add(file_name);
    menu->item_add(NULL);
    is_changed = 0;        //保存后认为文档没有被修改，可以直接关闭
}
bool EditWindow::IO_save(const QString &file_name)
{
    QFile fp("./diary_files/"+file_name);
    if (!fp.open(QIODevice::WriteOnly | QIODevice::Text))
        return false;
    QTextStream stream_out(&fp);
    QString str=ui->textEdit->toPlainText();
    stream_out<<str<<endl;
    fp.close();
    return true;
}
void FileMenu::menu_add(QString file_name)
{
    if(!this->file_queue.contains(file_name))
    {
        this->file_queue.append(file_name);
        this->file_num++;
    }       //如果文档队列中没有目标文件
    QFile fp("./diary_files/menu.txt");
    if(fp.open(QIODevice::WriteOnly | QIODevice::Text))
    {
        QTextStream stream_out(&fp);
        stream_out<<this->file_num<<endl;
```

```
        for(int i=0;i<this->file_num;i++)
        {
            QString str = this->file_queue[i];
            stream_out<<str<<endl;
        }
        fp.close();
    }
}
```

自动保存功能本质上就是在选项被勾选时执行一次保存操作，此后每当文档被修改时就进行一次保存，直到选项被取消勾选。textChanged()信号用于表示文本编辑器中的内容是否发生变化，EditWindow 的构造函数中将其与槽函数 set_change_status()连接了起来，每当内容被修改就会自动执行该函数，根据 auto_save 变量决定是否需要保存文档：

```
void EditWindow::on_action_autosave_toggled(bool arg1)
{
    if(arg1)    //arg1 == 1 表示选项被勾选
    {
        save_text();
    }
    auto_save = arg1;
}
void EditWindow::set_change_status()
{
    if(auto_save)
    {
        save_text();
    }
    else
    {
        is_changed = 1;
    }
}   //每当文本被修改时，如果需要自动保存则执行保存，否则记录文本已改变
```

为了在退出子界面时检查是否需要保存文档，项目中重写了关闭事件的处理函数，利用 EditWindow::window_close()判断是否需要保存，在需要保存时弹出提示窗口，并根据选择的结果来决定下一步的动作：

```
void EditWindow::closeEvent(QCloseEvent *event)
{
    bool need_close = window_close();//拦截关闭窗口的行为
    if(need_close) event->accept();
    else event->ignore();
}
bool EditWindow::window_close()
{
    if(is_changed==0)
    {
        return 1;    //文档没被修改则直接退出
    }
    else
    {
        QMessageBox box;
        box.setWindowTitle("提示");
        box.setText("要保存吗？");
        box.setStandardButtons(QMessageBox::Yes
```

```
                                | QMessageBox::No |QMessageBox::Cancel);
        int ret = box.exec();
        if(ret == QMessageBox::Yes)
        {
            save_text();
            return 1;       //保存并退出
        }
        else if(ret == QMessageBox::No)
        {
            return 1;       //直接退出
        }
        else
        {
            return 0;       //取消，不退出
        }
    }
}
```

六. 后续设计分析

这个项目目前已经完成了最基本的功能，但一方面目前仍有一些我想要添加的功能没有实现，另一方面也还存在一些 bug 或不够完善的地方，因此在这里也对接下来需要做的工作做一个总结和分析。

对于后续计划实现的功能，主要包含完成主界面删除和搜索两个按键的功能、根据日历中选中的日期显示文档、完成子界面“文件”菜单和“编辑”菜单中的所有操作。删除可以将选中的文档移至另一个“回收站”文件夹。搜索最简单的实现方法就是依次打开所有文档逐字比对，但可能会耗费很多时间，可以再寻找更好的策略。根据选中日期显示文档也不麻烦，只需要将选中的日期传入函数，根据日期检索文档，最后刷新显示区即可。另存为与打开操作类似，结合已有的打开和保存操作的代码即可。“编辑”菜单中的操作大都是 QTextEdit 类自带的，添加起来也不复杂。

除此之外，要是时间足够，我还想添加插入图片和表情、字数统计、修改字体字号、云端存储等功能。QTextEdit 类支持富文本编辑，应该是可以实现插入图片的。字数统计本身并不复杂，但我希望能像 Windows Word 一样输出在页面下方的菜单栏而不是文本之中，还不确定该怎么实现。QTextEdit 类应该可以修改整个文本的字体字号，但只修改选定内容我还不确定能否实现。云端存储大概就需要用脚本连接 github 或百度网盘之类的网站，还要实现自动上传，目前完全没有头绪。

关于问题和缺陷，首先还有一个 bug 等待修复：在子界面中点击打开操作，然后关闭弹出的地址检索窗口，此时再保存原有文档会导致 menu.txt 中的 file_num 错误+1，且没有新文档生成。此外在子界面新建文档会直接覆盖当前文档，不会检查当前文档是否需要保存，可能导致内容的丢失。所以新建也应该与退出操作类似，根据情况提示是否需要保存

另外在书写这次报告时把所有代码又重新梳理了一遍，发现有一些地方的实现存在不必

要的操作，比如 FileMenu::menu_add()函数中无论是否新增了文件，都会从头重写一遍 menu.txt，实际上只需要在真的新增了文件时修改第一行的 file_num，并在最后一行增加新增文件的名称即可。为了提高程序的运行效率，应该修改掉这种不必要的操作。

七. 程序功能补全与完善

最终实现的程序外观与先前预想的程序界面一致，此次主要是完成了先前还未实现的功能，包含主界面的删除文档和搜索文档功能，以及子界面下拉菜单中的另存为、查找和替换功能。其中主界面的搜索功能和子界面的查找功能共用一个 ui 界面，子界面的替换功能也设计了另一个 ui 界面。两个 ui 界面的示意图如下所示：

图 14 搜索界面示意图

图 15 替换界面示意图

除此之外，我还对原先的代码做了部分修改和重构，以提高整体的性能、稳定性和可扩展性。至此，这个项目的所有基本需求都已实现完毕，后续的一些设想详见**四. 总结与展望**部分。

八. 代码分析与功能实现

项目最终的 UML 类图如下所示，相较之前主要增加了用于控制搜索/查找功能的 SrchWindow 类、MainSrchWindow 类和 EditSrchWindow 类，以及用于控制替换功能的 RepWindow 类。此外其余类中的成员函数和成员变量也有一定的增减。

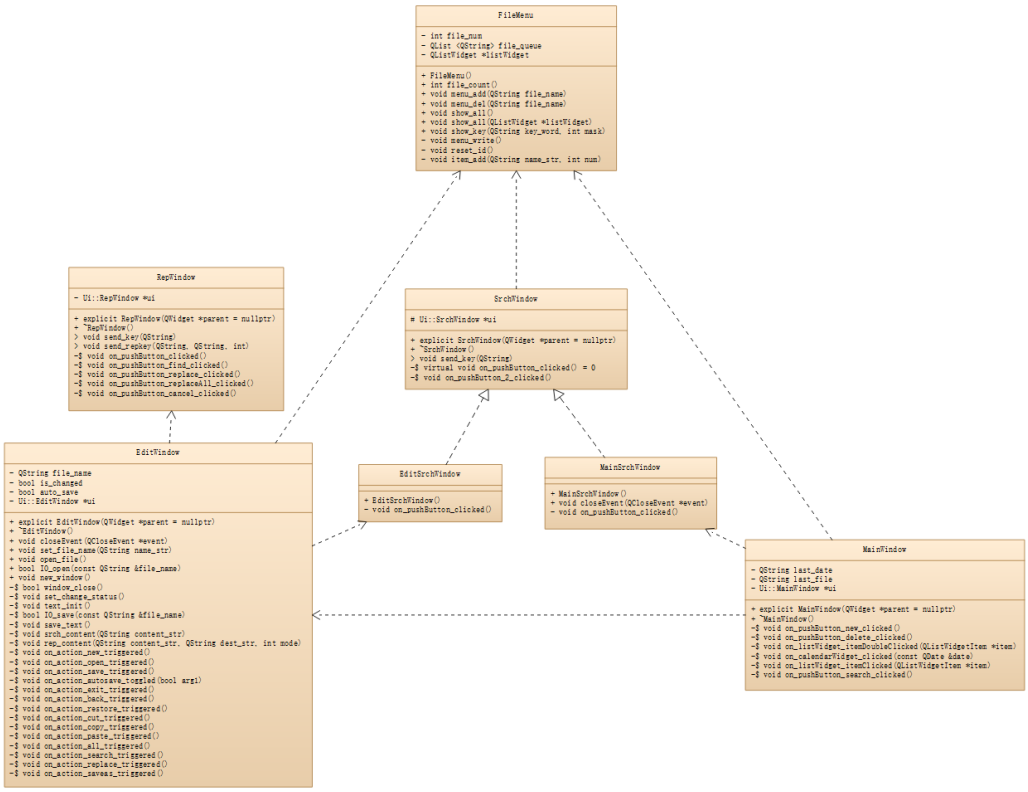

图 16 项目总体 UML 类图

主界面的删除文档功能需要先单击选中显示区中的一项文档，随后再单击删除按键即可删除选中文档。在单击“删除”按键后，主界面会弹出窗口请求确认删除行为，待用户确认后调用 FileMenu 类中的 menu_del 函数，将对应文档从文件目录中删去、重命名文档以标记已删除，并修改所有日记文档内的编号。各部分对应的具体代码如下：

```
void MainWindow::on_pushButton_delete_clicked()
{
    if(last_file == "Not a file")
    {
        QMessageBox::warning(NULL,"警告","请先选择一个文件！");
    }
    else
    {
        QString file_name = last_file.right(19);
        file_name.replace(4, 1, "_");
        file_name.replace(7, 1, "_");
        file_name.replace(10, 1, "_");
        file_name.replace(13, 1, "_");
        file_name.replace(16, 1, "_");
        file_name += ".txt";

        QMessageBox box;
        box.setWindowTitle("提示");
        box.setText("确定删除文件 \""+file_name+"\" 吗？");
        box.setStandardButtons(QMessageBox::Yes | QMessageBox::Cancel);
        int ret = box.exec();
        if(ret == QMessageBox::Yes)
        {
```

```
            //删除对应文件
            menu->menu_del(file_name);
            last_file = "Not a file";
        }
        else
        {
            //不执行操作
        }
    }
}

void FileMenu::menu_write()
{
    QFile fp("./diary_files/menu.txt");
    if(fp.open(QIODevice::WriteOnly | QIODevice::Text))
    {
        QTextStream stream_out(&fp);
        stream_out<<this->file_num<<endl;
        for(int i=0;i<this->file_num;i++)
        {
            QString str = this->file_queue[i];
            stream_out<<str<<endl;
        }
        fp.close();
    }
}

void FileMenu::reset_id()
{
    for(int i = 0; i < this->file_num; i++)
    {
       QString string = this->file_queue.at(i);
       QFile fp("./diary_files/" + string);
       QString tmp = "日记编号: " + QString::number(i+1) + "\r\n";
       if(fp.open(QIODevice::ReadOnly | QIODevice::Text))
       {
           QTextStream stream_in(&fp);
           QString line;
           int line_num = 1;
           while(stream_in.readLineInto(&line))
           {
               if(line_num != 1)
               {
                   tmp += line + "\r\n";
               }
               line_num++;
           }
           fp.close();
       }
       if(fp.open(QIODevice::WriteOnly | QIODevice::Text))
       {
           QTextStream stream_out(&fp);
           stream_out<<tmp<<endl;
           fp.close();
       }
    }
}
```

目前实现的操作并不是真正的从本地删除对应文档，只是将其从文档目录中删去，并在文档后加上已删除的标注，默认不再打开该文档而已。这样做的目的是为了便于以后添加类似“回收站”的功能，可以恢复已删除的文档。

主界面的搜索功能和子界面的查找功能使用的是同一个 ui 界面，仅在按键上的文字注释处有一些不同。不过由于二者实现的功能有所不同，该项目中设计了一个基类 SrchWindow 用于生成 ui、完成部分共有的功能；而最重要的搜索功能则实现为了一个接口，由两个子类 MainSrchWindow 和 EditSrchWindow 分别实现各自所需的操作。具体代码如下：

```
class SrchWindow : public QWidget
{
    Q_OBJECT

public:
    explicit SrchWindow(QWidget *parent = nullptr);
    ~SrchWindow();

signals:
    void send_key(QString);

private slots:
    virtual void on_pushButton_clicked() = 0;

    void on_pushButton_2_clicked();

protected:
    Ui::SrchWindow *ui;
};

class MainSrchWindow : public SrchWindow
{
public:
    MainSrchWindow();
    void closeEvent(QCloseEvent *event);
private:
    void on_pushButton_clicked();
};

class EditSrchWindow : public SrchWindow
{
public:
    EditSrchWindow();
private:
    void on_pushButton_clicked();
};
```

其中成员函数 on_pushButton_clicked()即为按下搜索按键后会触发的操作，在基类中是一个纯虚函数，由两个子类分别实现：

```
void MainSrchWindow::on_pushButton_clicked()
{
    QString srch_str = ui->lineEdit->text();
    menu->show_key(srch_str, SHOW_SRCH);
}
void EditSrchWindow::on_pushButton_clicked()
```

```
    {
        QString srch_str = ui->lineEdit->text();
        emit send_key(srch_str);
    }
```

主界面的搜索功能会在显示区中显示包含搜索内容的文档，子界面的查找则会先将查找内容发送给 EditWindow 类，再由 srch_content 函数将当前光标位置之后的第一个查找内容选中：

```
void EditWindow::srch_content(QString content_str)
{
    if(content_str == NULL)
    {

    }
    else if(ui->textEdit->find(content_str))//如果找到了
    {
        // 查找到后高亮显示
        QPalette palette = ui->textEdit->palette();
        palette.setColor(QPalette::Highlight,
                         palette.color(QPalette::Active,QPalette::Highlight));
        ui->textEdit->setPalette(palette);
    }
    else               //如果没找到，就把光标移到开头再找（循环），如果还找不到就是没有
    {
        ui->textEdit->moveCursor(QTextCursor::Start);
        if(ui->textEdit->find(content_str))
        {
            QPalette palette = ui->textEdit->palette();
            palette.setColor(QPalette::Highlight,
                             palette.color(QPalette::Active,QPalette::Highlight));
            ui->textEdit->setPalette(palette);
        }
        else
        {
            QMessageBox::information(this,tr("注意"),
                                     tr("没有找到内容"),QMessageBox::Ok);
        }
    }
}
```

替换功能仅在子界面中出现，实现原理与查找功能类似：在单击替换按键后，将查找内容和目标内容都发送给 EditWindow 类，由 rep_content 函数将查找内容修改为目标内容。由于包含了“替换”和“替换全部”两种功能，还需要将一个用于标识的参数 mode 也一同传给 EditWindow 类：

```
    void RepWindow::on_pushButton_replace_clicked()
    {
        QString srch_str = ui->srch_text->text();
        QString dest_str = ui->rep_text->text();
        emit send_repkey(srch_str, dest_str, 0);
    }
    void RepWindow::on_pushButton_replaceAll_clicked()
    {
        QString srch_str = ui->srch_text->text();
        QString dest_str = ui->rep_text->text();
```

```
        emit send_repkey(srch_str, dest_str, 1);
    }

    void EditWindow::rep_content(QString content_str, QString dest_str, int mode)
    {
        if(mode == 0)
        {
            //替换
            //单击替换后先检测当前光标选定的内容是不是 find 的内容
            //如果是，就直接替换，并尝试找下一个；否则类似 find 的功能，从头开始找一个
            QTextCursor now_cursor = ui->textEdit->textCursor();
            QString select_str = now_cursor.selectedText();
            if(select_str != content_str)
            {
                srch_content(content_str);
            }
            else
            {
                now_cursor.insertText(dest_str);
                srch_content(content_str);
            }
        }
        else
        {
            //替换全部
            QString tmp_str = ui->textEdit->toPlainText();
            tmp_str.replace(content_str, dest_str);
            ui->textEdit->setText(tmp_str);
        }
    }
```

另存为功能与保存功能类似，通过弹出窗口选择目标路径，然后将当前文件复制一份到目标路径即可：

```
void EditWindow::on_action_saveas_triggered()
{
    QString curPath = QDir::currentPath()+"/diary_files/"+file_name;
    QString dlgTitle = "另存为";
    QString filter = "所有文件(*.*);;文本文件(*.txt)";
    QString dirPath = QFileDialog::getSaveFileName(this, dlgTitle, curPath, filter, Q_NULLPTR,
                                QFileDialog::ShowDirsOnly | QFileDialog::DontResolveSymlinks);

    //如果选择的路径无效，则不保存
    if (!dirPath.isEmpty())
    {
       QFileInfo fileInfo(dirPath);
       if(fileInfo.exists())
           QFile::remove(dirPath);
       QMessageBox::warning(NULL,"测试",curPath);
       QFile::copy(curPath, dirPath);
    }
}
```

九. 面向对象特性分析

在实现这个项目的过程中，我也用到了不少面向对象的特性和方法，这里主要从“封装”、“继承”、“多态”、“重载”和“高内聚低耦合”几部分入手，对这一项目的面向对象特性做

一简单介绍和分析。

首先是封装：数据封装是面向对象最基础也是最核心的特性之一，也常常与数据隐藏和数据抽象联系在一起。它把数据和操作数据的函数捆绑在一起，通过隐藏对象实现的具体细节来减少耦合，避免受到外界的干扰和误用。这个项目实现的所有功能都是通过创建各种类及其成员来实现的，从而保证了良好的封装性。此外，每个类中的成员变量都是私有的，仅通过公有的成员函数向外提供交互的接口，从而实现数据隐藏的需求：

```
class FileMenu
{
public:
    int file_count();       //对外提供的接口
private:
    int file_num;           //对外隐藏的数据
    ...
};
int FileMenu::file_count()
{
    return file_num;
}
```

其次是继承：继承允许设计者依据一个类来定义另一个类，使得创建和维护一个应用程序变得更加容易，也达到了复用代码功能和提高执行效率的效果。不过继承是一种强耦合关系：父类的实现细节对于子类来说都是透明的，因此继承在一定程度上破坏了封装。继承的主要作用在于让若干个功能相近的类共享一般化的父类中的结构和行为，同时也扩展出独属于自己的特殊化的属性和方法。由于这个项目是基于 Qt 框架开发的，大多数类都继承自 Qt 提供的父类，从而拥有了很多父类中设计完善的结构和功能。

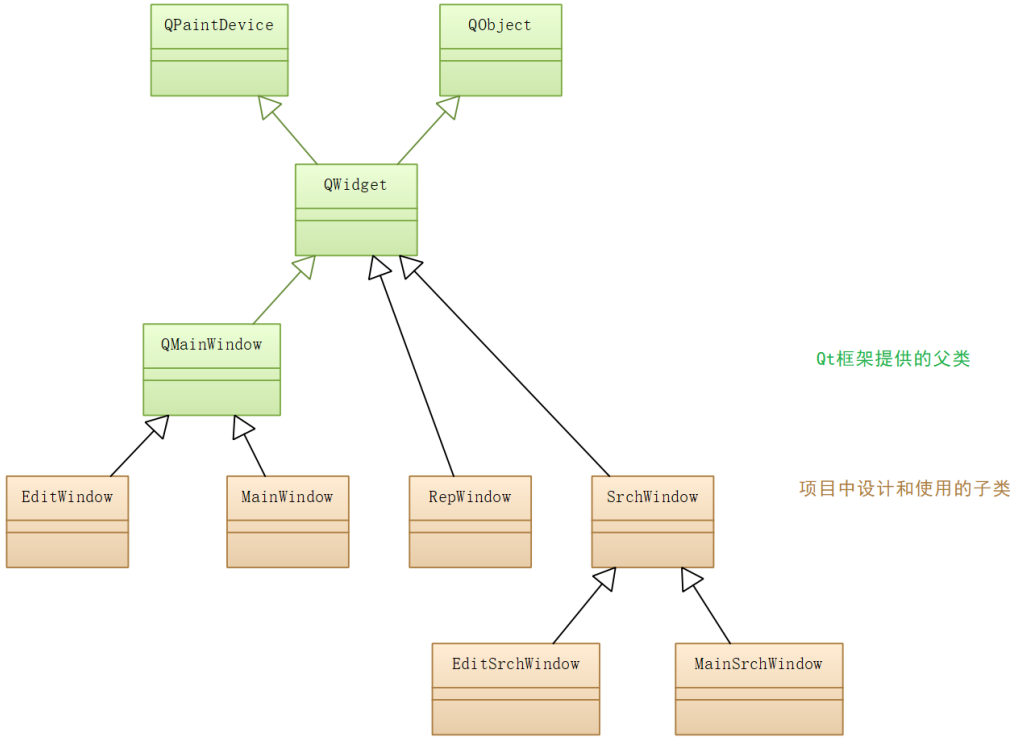

图 17 继承关系示意图

然后是多态：多态按字面的意思就是指多种形态，即一个操作可以被层次结构中的类以不同的方式实现。利用多态，子类可以扩展父类的能力，或者覆写父类的操作，根据调用函数的对象的类型来执行不同的函数。在有多种相似算法的情况下，多态可以有效避免 if-else 语句或 switch-case 语句带来的复杂和难以维护，减少设计的不稳定性。在这个项目中，主界面的搜索功能和子界面的查找功能共用同一个 ui 界面，其他很多功能也是相近的，只有核心的搜索功能根据调用来源的不同而有所区别。此时设计一个基类 SrchWindow 来控制 ui 和那些相似的功能，搜索功能则设计为接口；两个子类 EditSrchWindow 和 MainSrchWindow 分别重写搜索功能，即可根据调用函数的对象类型来选择执行相应的操作，也免去了 if-else 语句带来的种种问题。这种思想同时也属于设计模式中的策略模式：当一个系统需要动态地在几种算法中选择一种，而这些算法的区别仅在于它们的行为时，就可以将这些算法封装成单独的类并实现同一个接口，使它们可以相互替换：

```
class SrchWindow : public QWidget
{
    ...
private slots:
    virtual void on_pushButton_clicked() = 0;
};

class MainSrchWindow : public SrchWindow
{
    ...
private:
    void on_pushButton_clicked();
};

class EditSrchWindow : public SrchWindow
{
    ...
private:
    void on_pushButton_clicked();
};

void MainSrchWindow::on_pushButton_clicked()
{
    QString srch_str = ui->lineEdit->text();
    menu->show_key(srch_str, SHOW_SRCH);
}
void EditSrchWindow::on_pushButton_clicked()
{
    QString srch_str = ui->lineEdit->text();
    emit send_key(srch_str);
}
```

之后是重载：重载是 C++特有的一种概念，允许设计者用同样的名字来定义函数，只要它们的调用可以通过参数的个数或类型来进行区别。重载和多态一样，也是用来实现功能相似的函数，增加代码的可维护性和稳定性的。不过由于重载函数需要在参数个数或类型上有所区分，与多态中为同一个函数赋予不同的实现方式有所不同，分别有各自的使用场景：

```
class FileMenu
```

```
{
public:
    void show_all();
    void show_all(QListWidget *listWidget);
    ...
};

void FileMenu::show_all()
{
    this->listWidget->clear();
    for(int i = 0; i < this->file_num; i++)
    {
       QString string = this->file_queue.at(i);
       item_add(string, i+1);
    }
}
void FileMenu::show_all(QListWidget *listWidget)
{
    this->listWidget = listWidget;
    show_all();
}
```

最后是“高内聚低耦合”：“高内聚低耦合”是面向对象设计中评价一个系统设计好坏的标准，它希望一个模块中各个元素之间的联系程度尽量高，而模块之间相互联系的紧密程度尽量低。在代码设计的过程中，功能相近、联系紧密的方法被尽量归拢在同一个类中，避免一个类为它的内部属性提供直接的外界访问，减少其向外界提供实现其职责的方法，并降低这些方法受到类内部设计变化的影响，从而提高了内聚性；类之间的依赖关系尽量减少，避免互相依赖、循环依赖的出现，并保证类之间的影响只依赖于其他类提供的公开接口，而不依赖于它的内部工作原理，从而降低了耦合性。

十. 总结与展望

这个项目的代码设计与实现过程依然存在着一定的不足：在编写代码的过程中，我依然没有充分地运用面向对象的设计方法，总是面向“需求”编程，想到什么就写什么，经常要回头返工、重构代码。也正是由于没有在一开始就想好整体框架和布局，导致一部分函数的职责、功能虽然比较相近，却都被分开来单独实现，造成了一定的重复和冗余。此外，目前的几个类的功能依然比较复杂，没有很好地满足单一职责原则，这也需要再进一步地修改和细化。

另外，这一项目仍可以在一些方面进行补充和完善。首先是文档的持久化措施：目前这个项目需要记录下来的所有数据都是通过.txt 文档存储的，这样既不方便管理，在数据查找和读写的速度上也不尽人意。下学期的数据库系统课程或许可以带来一定的帮助，为这个项目增加一种更简单高效的文档持久化措施。此外，这个项目的界面还可以进一步美化，比如为各个窗口、选项添加一些图标，修改一下 ui 界面的样式等，这些内容又可以联系上人机交互课程的相关内容了。然后是之前计划添加的插入图片或表情、提供云端存储等功能，有

机会的话也可以尝试实现一下。

这次大作业是我第一次用 C++配合 Qt 框架编写代码，花费了不少时间学习、熟悉各种方法，但同样收获颇多。这也是我第一次接触面向对象的编程思想，对这种思想有了初步的了解和体验，也逐渐明白了面向对象的思想与设计方法对于整个项目的作用之大、好处之多。对我而言，这个项目不仅仅是一次课程大作业，同样也是为自己设计的一件趁手的工具。所以未来我还会继续修改、完善整个项目，相信面向对象的编程思想也会一直发挥出它的巨大价值。

参考文献

对 C++的简单介绍：https://www.runoob.com/cplusplus/cpp-inheritance.html

从 0 开始设计文本编辑器：https://www.catch22.net/tuts/neatpad#

UML 图箭头的意义：https://blog.csdn.net/wglla/article/details/52225571

对 QT 的简介和初步使用：http://c.biancheng.net/view/1804.html

在线查看 QT 设计源码：https://code.woboq.org/qt5/

利用 QT 设计文本编辑器：https://www.bilibili.com/video/BV1Lo4y1Z7hz?p=7

对信号和槽函数的介绍：https://blog.csdn.net/u014453898/article/details/70242786

重载运算符实现优先队列：https://www.cnblogs.com/huashanqingzhu/p/11040390.html

对 QT 各个控件的简介：https://www.cnblogs.com/zach0812/category/1524140.html?page=3

QList 控件的介绍：https://blog.csdn.net/qq_30725967/article/details/98205514

QListWidget 控件的介绍：https://www.pianshen.com/article/901542842/

拦截关闭事件：https://blog.csdn.net/vah101/article/details/6133728

QT 类继承关系图：
https://blog.csdn.net/qq_31036127/article/details/106890462?spm=1001.2101.3001.6650.6&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EOPENSEARCH%7Edefault-6.no_search_link&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EOPENSEARCH%7Edefault-6.no_search_link

QTextEdit 成员函数的介绍：https://blog.csdn.net/AAAA202012/article/details/120622526

两个窗口进行数据传输：https://www.cnblogs.com/zyore2013/p/4662911.html

获取 QTextEdit 选定的内容：https://blog.csdn.net/luochengguo/article/details/7520022

“另存为”功能的实现：https://blog.csdn.net/caoshangpa/article/details/78711743