哈爾濱工業大學（威海）
Harbin Institute of Technology, Weihai

Python 程序设计报告

学　　号：xxxxxxxxx

姓　　名：董成相

班　　级：xxxxxxx

任课教师：XXX

哈尔滨工业大学（威海）计算机科学与技术学院

总体要求

本次作业为《python 程序设计》课程的教学实践环节，要求同学们能够运用 python 语言进行程序设计，解决实际问题，以此来评价同学们对课程知识的掌握程度。同学们可以自由选题，独立完成。同学之间的作业，从选题、设计到报告书写都不能雷同，否则报告得 0 分，并上报学院进行作弊处罚。题目不限，但必须以 python 语言为主进行设计与开发。

作业评分主要从以下几个方面进行：

（1） 题材的选取。题目简单且易于完成的起平分低。严禁与其他课程实验、课设的题目及内容等方面相同。

（2） 功能。功能或模块少的起平分低。

（3） 设计功能完善且完成度高的起平分高。

（4） 报告书写的规范程度。

（5） 题目讲解的清晰程度。

报告内容八个组成部分：

（1） 题目

要求：一句简要的话概括或抽象出所做的实验内容

（2） 选题背景与意义

要求：描述选题的背景、针对该问题解决方案有多少种，发展历史及研究价值等。

（3） 需求分析

要求：可以使用用例图来描述所涉及的题目需求，并进行必要的分析。

（4） 软件体系结构或功能模块设计

要求：就需求分析中分析出来的功能进行设计及说明，包括功能模块设计、功能模块之间的关联与支撑等，可以用泳道图、流程图、数据流图、功能模块规划图等工具表明设计思路。

（5） 系统实现

要求：以类图、IDEF1x 图或 E-R 图等表明功能模块的实现方法。

（6） 主要代码说明

要求：挑选主要的、并与系统实现相对应的代码应进行必要的说明。

（7） 操作方法说明及系统测试

要求：给出每个模块的操作方法，并出据测试数据，证明软件的可用性。

（8） 总结

要求：如实撰写任务完成过程的收获和体会以及遇到问题的思考，严禁雷同。

所需提交的材料：

（1） 大作业报告 1 份

（2） 对于大作业内容讲解的 ppt 1 份

（3） 录制对大作业主要内容讲解的视频 1 份

目录

一、题目

在 Windows10 上基于 Python3.7.1、Pillow、MySql、PyQt5、PyMySql 等实现的 Windows 聚焦图片获取、查看、管理程序。

二、选题背景与意义

（一） 选题背景

首先当然是为了进行一次 Python 开发实践，其次是为了解决我在使用电脑过程中遇到的一个重复性劳动极高的行为，那就是从 Windows 聚焦图片保存文件夹（以下称源文件夹）将图片复制到我自己保存壁纸的文件夹，由于壁纸在源文件夹中的文件名是一串乱码，因此在复制之后为了预览图片我还要逐个重命名才能进行查看，重命名之后还要对壁纸进行编号，使用源文件名多少有点不妥和不优雅，因为源文件名是一串 64 位的字符串，我猜测是 Windows 为了避免重复，可以说是某种意义上的主键。

这之后还可能会面临着壁纸的筛选工作，因为不是所有获取的壁纸都是我想要保存的，而删除操作必然会导致文件编号的乱序，或者说浪费。因此我便萌生了编写一个 Python 脚本来替我完成这些重复性劳动的想法，最初加以实践应该是在 2019-2020 年度的秋季学期。

我在最初编写的是一个极其简单的脚本，仅仅 36 行代码，它完成的任务也很简单，就是把壁纸拷贝到存储文件夹并添加拓展名，这个时候仍然不够优雅。我又在寒假期间进行了一些完善，将代码扩充到了 168 行，这个时候实现的功能也不够完善，还不能帮助我处理因壁纸被删除而产生的未被占用的编号。

这之后我也尝试了将代码重构为完全面向对象的，并有了一点小进展，但是这个时候我在学习 Git 和 Github 的使用方法，由于一次误操作（rebase）将有一点小进展的面向对象版本给弄丢了。再之后就要开始上课了，时间也开始变少了，零零散散的就只能在原来面向过程和面向对象混杂的版本上做一些极小的调整，代码行数仍然是 100 多行的样子。

恰好本学期我们有 Python 这门课程，于是我便想趁着课设的机会，编写一个完整的程序，优雅地获取 Windows 聚焦壁纸，也顺便对我的 Python 开发技能进行一次检验。

（二） 解决方案

1. 人工处理

这个方法就是我最初使用的解决方案，但是所作的工作重复性劳动极高，效率低下，并且有点喧宾夺主，毕竟我只是想获得 Windows 聚焦壁纸，对他们的管理相比获取他们，获取他们的优先级是更高的。

2. Windows BAT 脚本

这个方法极为简单，但是提供的功能也很简单。这种解决方案需要我们先将图片复制到外界的一个文件夹（直接在文件夹内处理也是可以的，但是会污染源文件夹），然后新建一个文本文件，内容为：

```
ren * *.jpg
```

然后将文件扩展名重命名为 bat，双击执行即可。该方法与我最初编写的 36 行 Python 代码的效果几乎是一致的，只是这种方法多了一个手动复制的过程。这种方法也不能对源文件夹中的非壁纸文件进行过滤，就比说我在第一个版本遇到的一个 Windows 自家的图标，显然是没有必要获取的。

3. 程序

还记得某一次我在刷知乎时遇到了一个小程序，它也可以帮助获取壁纸，并按照版式建立子文件夹进行分类，这个功能也是相对比较简单的，没办法使用这个程序管理壁纸，而且每次提取都会新建文件夹，没法统一管理，为了统一管理还需要有人工操作。我印象中它也没有解决源文件名是一串 64 位的字符串的问题，这个功能点没能满足我自己的需求。

4. 我的解决方案

使用 Python 编写脚本，基于 Pillow 对壁纸的版式和格式进行分析，以便过滤掉非壁纸文件，使用 MySql 数据库作为一种存储文件名称的方式，以便对图片进行去重处理，以及统一管理获取的所有壁纸，最后使用 PyQt5 开发 GUI。

（三） 发展历史

我想大部分有这种需求的人，最开始都是人工处理的吧，然后因为对重复性劳动不满而编写了 Windows Bat 脚本或者简单的 Python 脚本，再进一步有人做成了小程序，从一定程度上摆脱对语言的依赖，然后就应该是我这种程度的了，当然有一些可能不在我的调查范围内，所以没有体现在这里。

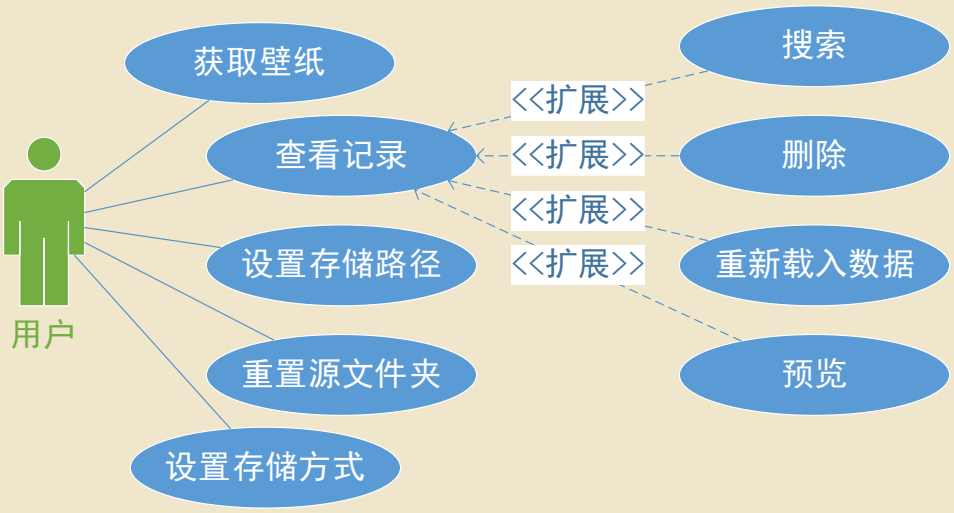
（四） 研究价值

对 Python 的部分内置库进行学习与理解，比如说本次课设中使用到的：os、shutil、json、enum、datetime、sys；对 Python 的包管理进行实践与理解；对 Pillow 库的 Image 模块进行学习与理解；对 MySql 数据库进行学习与理解；使用 PyQt5 对使用 Python 进行 GUI 开发有一定的体会；加深对控制器模式及 Python 面向对象开发的理解；接触 json 文

件的处理方式及使用他作为设置文件的好处。最最重要的当然是独自完整地开发一个完备程序的体验与实践。

三、需求分析

（一） 用例图



（二） 用例描述

1. 概述

经过分析可以发现该系统中的角色（Actor）只有一个用户，系统向用户暴露的功能分别是：获取壁纸、查看记录、设置存储路径、重置源文件夹、设置存储方式，其中搜索、删除、重新载入数据、预览是对查看记录的扩展（extend），所谓扩展就是，执行了查看记录用例，却不一定会执行扩展用例，具体表现在程序上就是打开了查看记录界面却什么也没有做就关闭了。

2. 获取壁纸

1) 用户点击获取壁纸按钮
2) 获取设置文件内容
3) 构造 ImageCopier 对象的实例
4) 筛选新的壁纸
5) 生成记录行
6) 复制壁纸
7) 重命名壁纸

8) 向记录文件/数据库中写入记录行
9) 更新设置文件中的图片编号（num）
10) 向用户反馈执行结果
11) 结束

3. 查看记录

1) 用户点击查看记录按钮
2) 获取设置文件内容
3) 构造 FilenameManger 子类的实例
4) 获取所有文件名称记录
5) 向用户反馈执行结果
6) 结束

4. 搜索

1) 用户选择搜索模式
2) 用户键入关键字
3) 用户点击搜索按钮
4) 从界面获取搜索关键字
5) 检查关键字合法性
6) 若合法则执行 4)，否则执行 7)
7) 获取设置文件内容
8) 构造 FilenameManger 子类的实例
9) 基于搜索模式进行搜索
10) 向用户反馈执行结果
11) 结束

5. 删除

1) 用户选择待删除壁纸
2) 用户点击删除按钮
3) 从界面获取待删除壁纸的 id
4) 获取设置文件内容
5) 构造 FilenameManager 子类的实例
6) 删除记录行
7) 删除壁纸
8) 更新未被使用到的 id 和编号（num）

9) 向用户反馈执行结果
10) 结束

6. 重新载入数据

1) 用户点击了重新载入数据按钮
2) 获取设置文件内容
3) 构造 FilenameManager 子类的实例
4) 获取所有记录行
5) 更新表格的数据来源
6) 刷新表格内容
7) 向用户反馈执行结果
8) 结束

7. 预览

1) 用户点击了预览按钮
2) 获取预览按钮所在行
3) 构造 PreviewWallaper 对象的实例
4) 显示壁纸
5) 用户在预览界面上单击了鼠标左键
6) 关闭预览界面
7) 结束

8. 设置存储路径

1) 用户点击了设置存储路径按钮
2) 程序弹出对话框让用户选择新的存储路径
3) 若用户选择了存储路径则执行 4)，否则执行 10)
4) 询问用户是否转移壁纸
5) 获取设置文件内容
6) 更新存储路径
7) 如果用户想要转移壁纸，则自行 6)，否则执行 7）
8) 转移壁纸
9) 向用户反馈执行结果
10) 结束

9. 重置源文件夹

注释：因为源文件夹的路径比较难记，因此将源文件夹的抽象路径硬编码进了程序中，方便用户误操作导致源文件夹路径错误时，可以方便的重置源文件夹路径

1) 用户点击了重置源文件夹按钮
2) 获取设置文件夹内容
3) 重置源文件路径
4) 向用户反馈执行结果
5) 结束

10. 设置存储方式

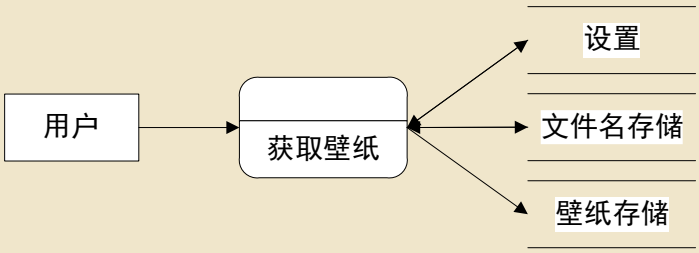
这里的存储方式是指原始文件名称的存储方式，一共有两种存储方式：数据库和纯文本文件。默认是纯文本文件，因为数据库配置对非程序员来讲可能有点复杂，所以默认设置为纯文本文件。所以设计了这个功能方便有需要的用户调整存储方式为数据库。

1) 用户点击了设置存储方式按钮
2) 程序弹出选择存储方式对话框
3) 若用户点击了确定按钮则执行 4)，否则执行 5)
4) 向用户反馈执行结果
5) 结束

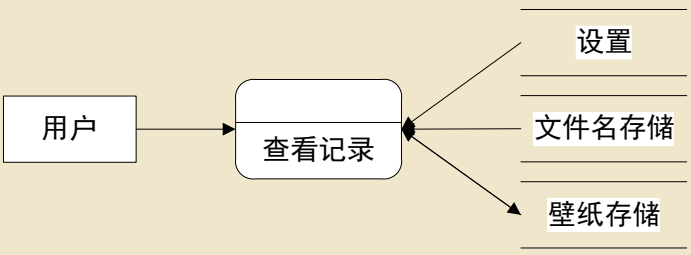
四、功能模块设计

（一） 数据流图

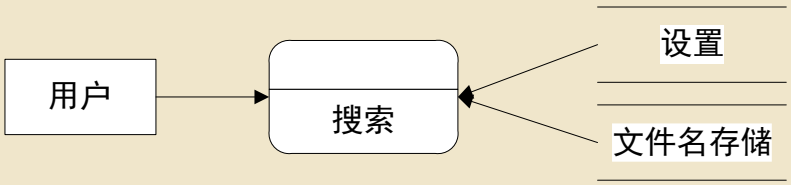
1. 获取壁纸



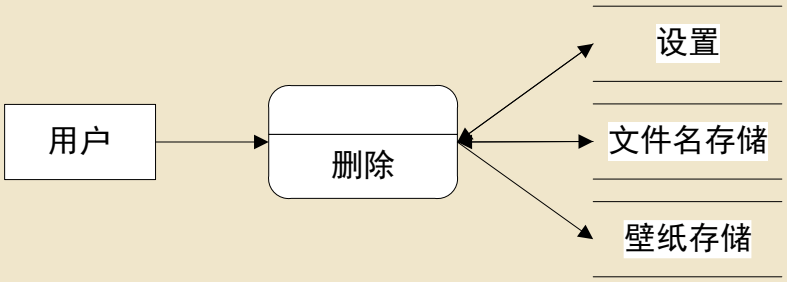
2. 查看记录



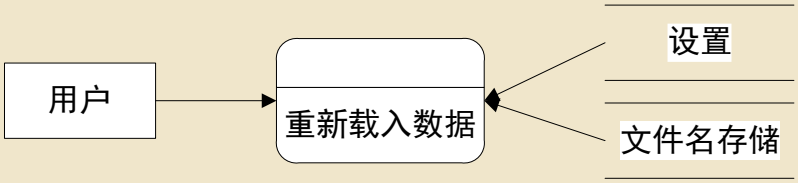
3. 搜索



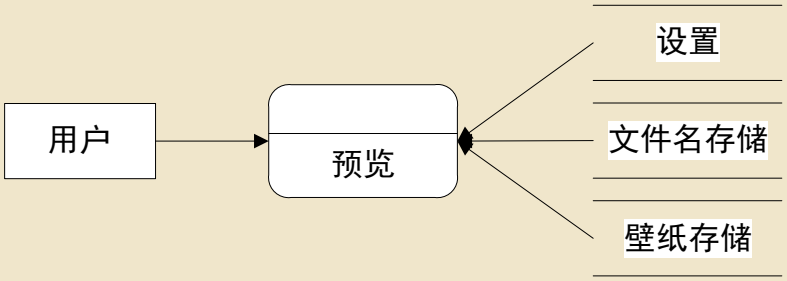
4. 删除



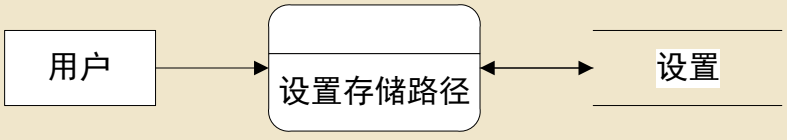
5. 重新载入数据



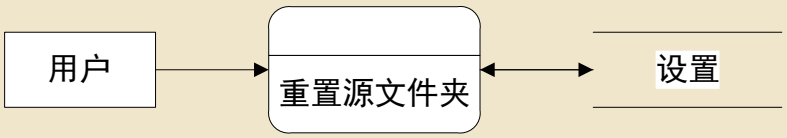
6. 预览



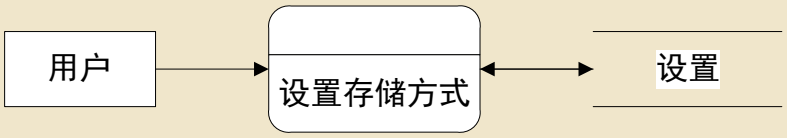
7. 设置存储路径



8. 重置源文件夹

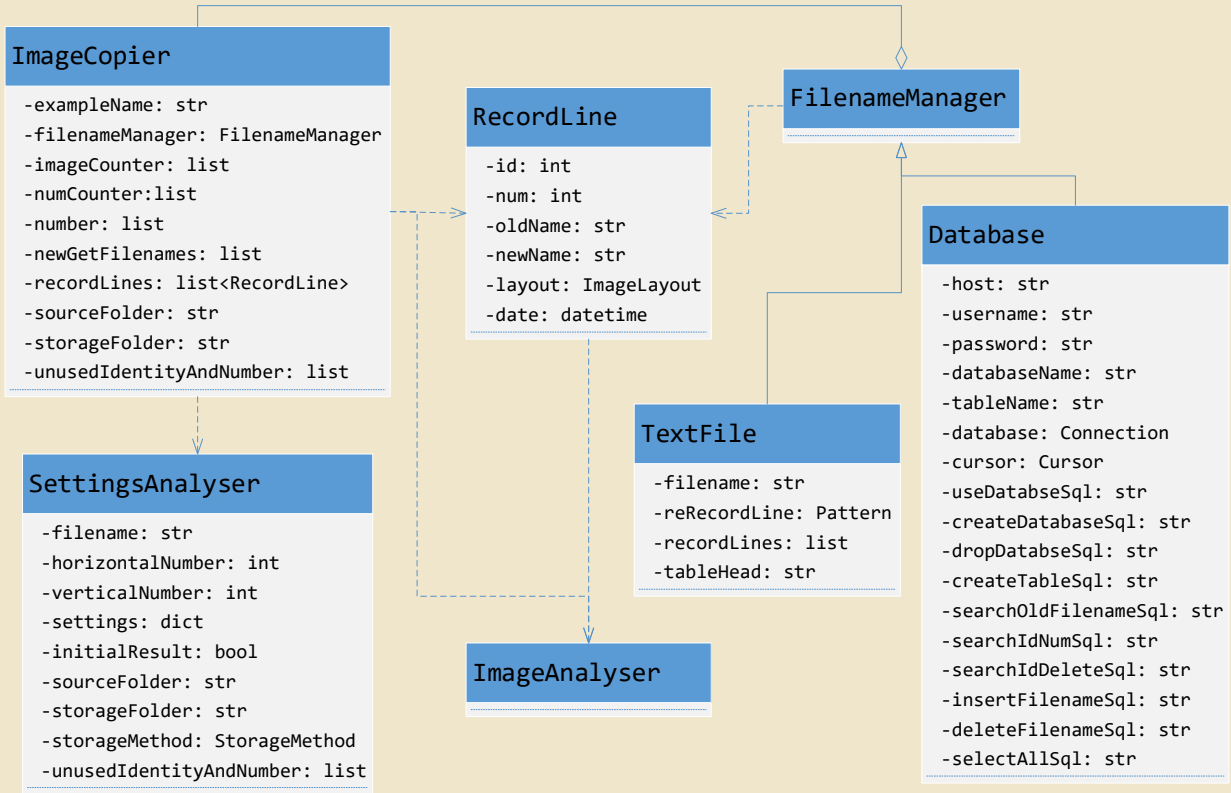


9. 设置存储方式



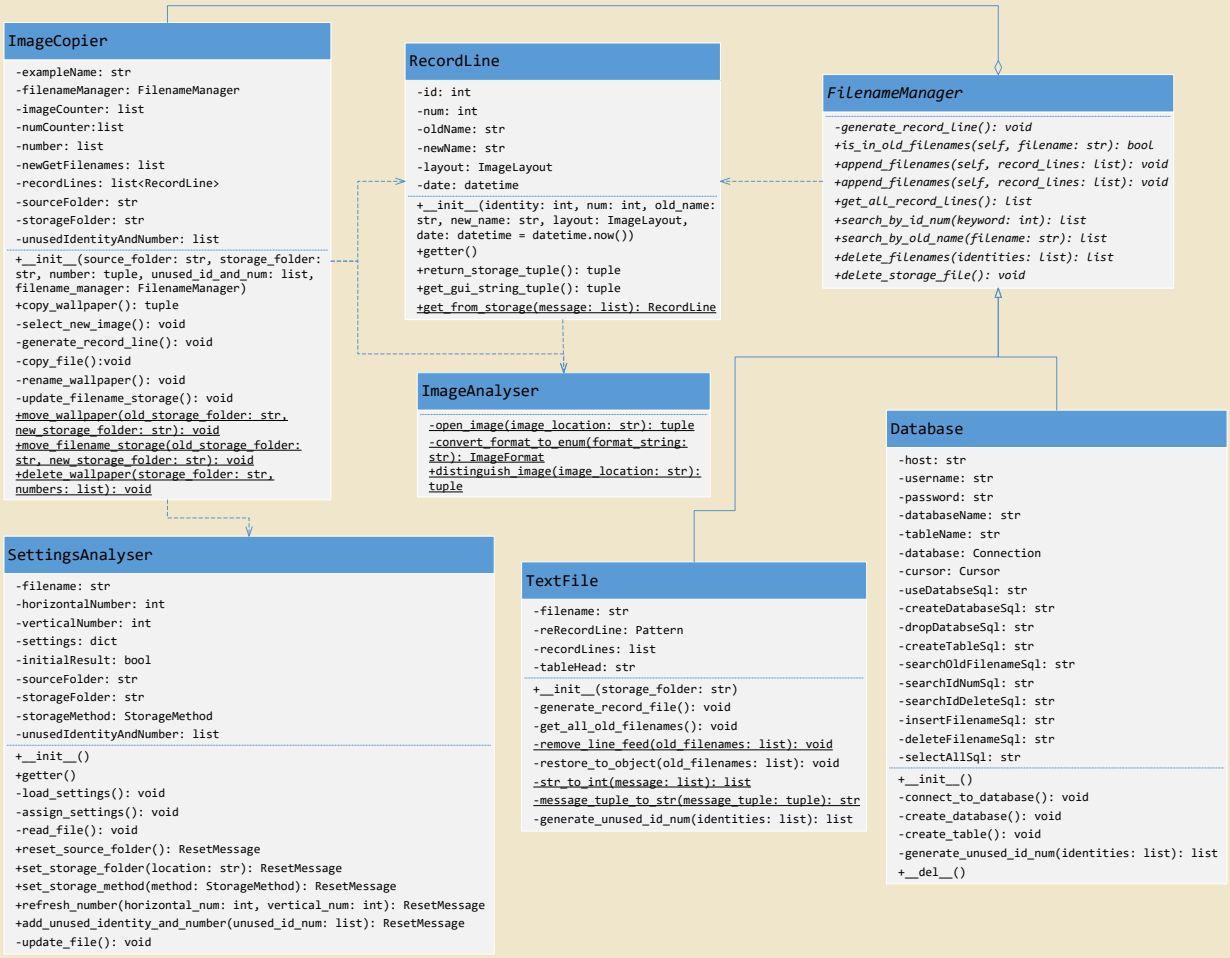
（二） 分析类图

我在需求分析中提到的功能，经过分析就不难发现，只有使用多个模块协同完成，才能比较好的处理，接下来就是按照我设计的几个模块所绘制的分析类图。

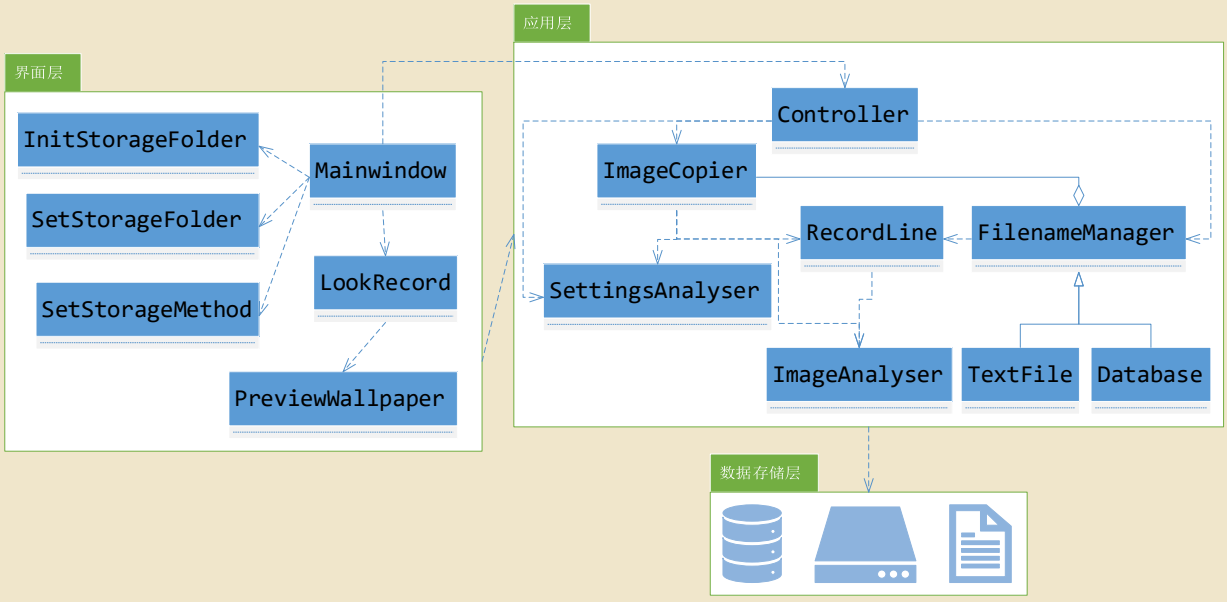


五、系统实现

（一） 设计类图



（二） 包图



六、主要代码说明

（一） ImageAnalyser 类

ImageAnalyser 类的代码如上所示，该类负责对图片的版式及图片的格式进行检测，并向调用者反馈一个包含图片版式和图片格式的元组，以便调用者进一步处理图片。这里还用到了两个枚举类，会在紧邻着的下一代码片段进行介绍。代码如下：

```
from enum import Enum
from PIL import Image


class ImageAnalyser:
    @staticmethod
    def __open_image(image_location: str):
        try:
            with Image.open(image_location) as image:
                width = image.width
                image_format = image.format
                return width, image_format
        except OSError:
            return 0, ''

    @staticmethod
    def __convert_format_to_enum(format_string: str):
        format_string = format_string.lower()
        if format_string == 'jpg' or format_string == 'jpeg':
            return ImageFormat.jpg
        if format_string == 'png':
            return ImageFormat.png
        if format_string == 'gif':
            return ImageFormat.gif
        if format_string == 'bmp':
            return ImageFormat.bmp
        return ImageFormat.unknown

    @staticmethod
    def distinguish_image(image_location: str):
        width, image_format = ImageAnalyser.__open_image(image_location)
        if width != 0:
            image_format_enum =
                        ImageAnalyser.__convert_format_to_enum(image_format)
            if width == 1920 or width == 2000:
                return ImageLayout.horizontal, image_format_enum
```

使用枚举类的好处就是，可以使用 is 关键字对图片的版式和格式进行判断，代码在语义上更加符合实际，同时枚举成员的名称也是包含语义的，阅读代码时理解起来相比单纯的数字会更加容易，并且 Python 枚举类的成员的值还可以是字符串等，语义相比其它语言中的枚举类更加丰富。同时也可以限制程序处理的数据类型，在这里就是图片版式和图片格式，避免程序面临未知的数据类型时出错。程序中其他地方也有使用到枚举类，使用的好处和原因大体都是一样的。

```
class ImageLayout(Enum):
    horizontal = 0      # 横版图片
    vertical = 1        # 竖版图片
    unknown = 2         # 未知版式
    invalidFile = 3     # 非图片文件


class ImageFormat(Enum):
    jpg = 0     # jpg/jpeg 格式
    png = 1
    gif = 2
    bmp = 3
    other = 4
```

（二） SettingsAnalyser 类

这两个方法是对设置的读取和更新方法，采用 with 语句避免忘记关闭文件，更新文件时采用了"w"覆盖写模式，因为数据更新时，尤其是产生了未被占用的 id 和编号（num）时使用其他模式会在设置文件尾部产生多余的}和]，这些是更新前文件尾部的内容，出现这个问题是不能容忍或者交由程序处理的，因为出问题的情况没法确定，只有采用覆盖写可以解决这个问题，并且解决起来非常方便，根本不需要花大力气去分析究竟哪些字节会多出来，所以便采用覆盖写模式。代码如下：

```
import json
import os


class SettingsAnalyser:
    def __load_settings(self):
        with open(self.__filename, encoding="utf-8", mode="r") as file:
            self.__settings = json.load(file)

    def __update_file(self):
        with open(self.__filename, "w", encoding="utf-8") as file:
            json.dump(self.__settings, file, indent=4)
```

紧接着下一页的代码也是 SettingsAnalyser 类的方法，把这一段代码拿出来主要是为

了说明 SettingsAnalyser 更新设置文件的逻辑及异常处理。

可以发现 SettingsAnalyser 更新设置需要这么几步：

1. 获取新的数据；
2. 使用新的数据更新设置字典；
3. 尝试更新设置文件；
4. 如果成功则使用新的数据更新旧的数据，如果失败则使用旧的数据回写设置字典；
5. 最后返回更新结果

从这里也可以看出枚举类的魅力所在。代码如下：

```
    def set_storage_folder(self, location: str):
        if os.path.exists(location) is True:
            try:
                self.__settings["StorageFolder"] = location
                self.__update_file()
                # 对程序中正在使用中的数据进行更新
                self.__storageFolder = self.__settings["StorageFolder"]
                return ResetMessage.success
            except FileNotFoundError:
                self.__settings["StorageFolder"] = self.__storageFolder
                return ResetMessage.fileNotFound
        else:
            return ResetMessage.locationNotExist
```

（三） TextFile 类

TextFile 是 FilenameManager 的子类，下面紧邻的文本框中的方法是将从文本文件中读取的数据还原为 RecordLine 对象，以便处理，该方法使用了 RecordLine 的静态方法 get_from_ storage，以及正则表达式切分从文本文件中读取的字符串。代码如下：

```
from FilenameManager.FilenameManager import FilenameManager
from FilenameManager.RecordLine import RecordLine
from ImageAnalyser import ImageLayout
import re
import os


class TextFile(FilenameManager):
    def __restore_to_object(self, old_filenames: list):
        for line in old_filenames:
            message = self.__re_record_line.split(line)
            message = self.__str_to_int(message)
            record_line = RecordLine.get_from_storage(message)
            self.__record_lines.append(record_line)
```

下面紧邻的文本框中的方法则是以原始名称为关键字进行查找的方法，这里为了实现模糊匹配，使用了 in 关键字。代码如下：

```
    def search_by_old_name(self, filename: str):
        tempRecordLines = list()
        for recordLine in self.__record_lines:
            if filename in recordLine.get_old_name():
                tempRecordLines.append(recordLine)
        return tempRecordLines
```

（四） Database 类

该类用于处理以数据库形式存储的文件名信息，也是 FilenameManager 的子类。该类编写了大量 SQL 语句用于实现相应的功能。代码如下：

```
from FilenameManager.FilenameManager import FilenameManager
from FilenameManager.RecordLine import RecordLine
from ImageAnalyser import ImageLayout
import pymysql


class Database(FilenameManager):
    def __init__(self):
        self.__host = "localhost"
        self.__username = "root"
        self.__password = "my,,,321"
        self.__database_name = "FilenameStorage"
        self.__table_name = "filename"
        self.__use_database_sql = "Use {}".format(self.__database_name)
        self.__create_database_sql = "Create Database If Not Exists {}"
                                     .format(self.__database_name)
        self.__drop_database_sql = """Drop Database {}"""
                                     .format(self.__database_name)
        self.__create_table_sql = """Create Table If Not Exists `{}` (
                                  `id` int Not Null,
                                  `num` int Not Null,
                                  `old_name` nchar(64) Not Null,
                                  `new_name` varchar(29) Not Null,
                                  `layout` int Not Null,
                                  `date` nchar(19) Not Null,
                                  Primary Key (`id`)
                                  )
                                """.format(self.__table_name)
```

转下一页。

续上一页。

```
        self.__search_old_filename_sql =
                            """Select id, num, old_name, new_name, layout, date
                               From {} Where old_name Like "%{}%" """
        self.__search_id_num_sql =
                            """Select id, num, old_name, new_name, layout, date
                              From {} Where id = {} Or num = {}"""
        # 用于在删除前获取 unused_id_num
        self.__search_id_delete_sql =
                             """Select id, num, layout From {} Where id = {}"""
        self.__insert_filename_sql =
                  """Insert Into {0} (id, num, old_name, new_name, layout, date)
                     Values ({1}, {2}, "{3}", "{4}", {5}, "{6}")
                  """
        self.__delete_filename_sql = """Delete From {} Where id = {}"""
        self.__select_all_sql =
                  """Select id, num, old_name, new_name, layout, date From {}"""
        self.__connect_to_database()
```

在该类的析构方法中我添加了关闭数据库的语句。因为有可能数据库创建失败，为了不使程序异常崩溃，再次也进行了异常处理。代码如下：

```
    def __del__(self):
        try:
            self.__database.close()
        except AttributeError:
            pass
```

我也考虑了数据库不存在的情况，在数据库不存在时，程序会连接服务器，而不指明具体的数据库，然后执行数据库创建 SQL 语句，同时还要 USE DATABSE，否则使用获取的连接对数据库无法进行操作，代码如下：

```
    def __create_database(self):
        self.__database = pymysql.connect(self.__host,
                                          self.__username,
                                          self.__password,
                                          charset='utf8mb4')
        self.__cursor = self.__database.cursor()
        self.__cursor.execute(self.__create_database_sql)
        self.__cursor.execute(self.__use_database_sql)
```

（五） ImageCopier 类

该类是核心的获取壁纸的类，它依赖于 ImageAnalyser、SettingsAnalyser 和 Filename Manager。我把 copy_wallpaper 这一过程拆分成了 5 个子过程，其中数筛选图片和生成记

录行的代码比较重要。代码如下：

```
import shutil
from FilenameManager.RecordLine import RecordLine
from FilenameManager.FilenameManager import *
from os import rename, listdir, path, mkdir, remove
from ImageAnalyser import *


class ImageCopier:

    def copy_wallpaper(self):
        self.__select_new_image()
        self.__generate_record_line()
        self.__copy_file()
        self.__rename_wallpaper()
        self.__update_filename_storage()
        return self.__imageCounter, self.__numCounter
```

generate_record_line 是先将之后会产生的数据一次性生成了，有点类似于预先制定目标，这样之后完成任务时就不必再进行设计，直接按照预定目标完成就好了，投射到程序中就是直接使用数据就好了。这里本来是准备在数据全部生成之后再生成记录行的，但是后来考虑了一下，因为我将 copy_wallpaper 方法拆分为了 5 个子过程，数据必然是在不同时刻产生的，这样就必定需要需要一个缓存，还需要重新从文件夹中读取数据，直接生成可以免去缓存，同时避免了不必要的重新读取。代码如下：

```
    def __generate_record_line(self):
        identityUp = self.__number[0] + self.__number[1]
        for old_name in self.__new_get_filenames:
            absoluteName = path.join(self.__source_folder, old_name)
            i_layout, i_format =
                                ImageAnalyser.distinguish_image(absoluteName)
            if i_layout not in [ImageLayout.unknown,
                                                    ImageLayout.invalidFile]:
                # 依据待处理图片版式选择要处理的数据
                flag = 0 if i_layout is ImageLayout.horizontal else 1
                # 记录行属性预生成
                try:
                    identity = self.__unused_id_and_num[0].pop(0)
                except IndexError:
                    identityUp += 1
                    identity = identityUp
                try:
                    num = self.__unused_id_and_num[flag + 1].pop(0)
```

转下一页。

续上一页。

```
            except IndexError:
                self.__numCounter[flag] += 1
                num = self.__number[flag] + self.__numCounter[flag]
            self.__imageCounter[flag] += 1
            new_name = self.__example_name +
                                            str(num) + '.' + i_format.name
```

因为一张壁纸会被 Windows 向用户多次推送，因此检测出重复推送的图片显得尤为重要。经过之前的观察，我发现源文件夹中的原始文件名称实际上就是壁纸的主键，这也在后来经过了验证。select_new_image 方法就是负责从源文件夹中筛选新的壁纸，筛选的规则是基于源文件夹中的原始文件名是壁纸的主键，把它放在 generate_record_line 之后不是因为它没有 generate_record_line 重要，而是因为排版需要才这样做的，毕竟 generate_record_line 需要基于 select_new_image 筛选出的新壁纸工作。代码如下：

```
    def __select_new_image(self):
        new_filenames = listdir(self.__source_folder)
        for filename in new_filenames:
            if self.__filename_manager.is_in_old_filenames(filename)
                                                                is False:
                self.__new_get_filenames.append(filename)
```

不难发现，以上介绍的都是应用层面的代码，接下来我将介绍界面层的代码。

（六） InitStorageFolderDialog 类

该类负责在程序首次运行或者存储路径未被设置时要求用户设置存储路径，它与 SetStorageFolderDialog 类的区别是：1. 当用户未选择路径时会发送一个 storageFolder NotSet 的信号通知主窗口关闭界面并结束程序，以限制用户的使用，同时也是为了保证程序的健壮性；2. 他不会询问用户是否转移已有壁纸，毕竟这个时候还没有获得任何壁纸。代码如下：

```
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
from Controller import Controller


class InitStorageFolderDialog(QFileDialog):
    storageFolderNotSet = pyqtSignal()
```

转下一页。

续前一页

```
    def __init__(self):
        super(InitStorageFolderDialog, self).__init__()

    def open_directory_dialog(self):
        directory = QFileDialog.getExistingDirectory(self,
                                                     "请选择存储文件夹", ".")
```

（七） PreviewWallpaper 类

该类用于预览壁纸，秉承着一切只由一个工具完成的思想，我给系统添加了这个功能，毕竟使用一个程序的时候需要打开别的软件来辅助有点麻烦，尤其是当其他程序提供的辅助是很简单的功能时。该窗口使用了无边框的窗口模式，以便营造最佳的视觉效果。为了正常显示壁纸，我还给这个类添加了一个缩放图片的方法，因为按照原始大小显示图片的体验很差，横版的会铺满屏幕还算好一些，竖版的屏幕显示不下体验就会很差。我也重写了该窗口的鼠标释放事件，实现了我们使用其他软件预览图片时单击鼠标左键即可关闭图片的功能。代码如下：

```
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
import os
from ImageAnalyser import ImageLayout


class PreviewWallpaper(QWidget):

    def __zoom_image(self):
        if self.__imageLayout is ImageLayout.horizontal:
            size = QSize(*self.__horizontalSize)
        elif self.__imageLayout is ImageLayout.vertical:
            size = QSize(*self.__verticalSize)
        image = QImage(self.__filename)
        self.__pixmap = QPixmap.fromImage(
                                    image.scaled(size, Qt.IgnoreAspectRatio))

    def mouseReleaseEvent(self, e: QMouseEvent):
        if e.button() == Qt.LeftButton:
            self.close()
```

（八） LookRecord 类

我在这自定义了 MyQLabel 类和 MyQCheckBox 类，给这两个类增加行标，以便获取

预览按钮被单击时它所在的行以及复选框被选中时，获取被选中的复选框的行标。代码如下：

```
from GUI.PreviewWallpaper import *
from Controller import *


class MyQLabel(QLabel):
    def __init__(self, row):
        super(MyQLabel, self).__init__()
        self.__row = row

    def get_row(self):
        return self.__row

    def set_row(self, row):
        self.__row = row


class MyQCheckBox(QCheckBox):
    def __init__(self, row: int):
        super(MyQCheckBox, self).__init__()
        self.__row = row

    def get_row(self):
        return self.__row

    def set_row(self, row: int):
        self.__row = row
```

我在这里给压入单元格中的控件套了一层 QWidget，然后将 QWidget 压入单元格，直接压入单元格没法设置控件居中，可以通过这种方式间接实现居中。虽然针对复选框的效果并没有预期中的效果那么好，但是相比直接压入控件已经好太多了。将控件压入之后再结合自定义的两个类，成功解决了复选框和预览按钮的功能。代码如下：

```
class LookRecord(QWidget):
    def __set_widget_style(self, widget: QWidget):
        temp = QWidget()
        layout = QHBoxLayout()
        layout.addWidget(widget)
        temp.setLayout(layout)
        temp.setFont(self.__tableOperationFont)
        return temp
```

单元格数据的填充我设计成了一个单独的方法，方便更新数据时复用。在压入数据和控件时，还要注意绑定槽函数。代码如下：

```
    def __add_item_to_table(self):
        # 清空复选框和预览标签
        del self.__checkBoxes
        del self.__labels
        self.__checkBoxes = list()
        self.__labels = list()
        for i in range(len(self.__recordLines)):
            # 放入操作控件
            selectionCheckBox = MyQCheckBox(i)
            self.__checkBoxes.append(selectionCheckBox)
            selectionCheckBox = self.__set_widget_style(selectionCheckBox)

            previewLabel = MyQLabel(i)
            self.__labels.append(previewLabel)
            previewLabel.setText("<a href = '#'>预览</a>")
            previewLabel.linkActivated.connect(self.__preview_image)
            previewLabel = self.__set_widget_style(previewLabel)

            self.__table.setCellWidget(i, 0, selectionCheckBox)
            self.__table.setCellWidget(i, 1, previewLabel)

            # 放入记录项
            message = self.__recordLines[i].get_gui_string_tuple()
            for j in range(5):
                item = QTableWidgetItem(message[j])
                item.setTextAlignment(Qt.AlignCenter)
                item.setFont(self.__tableContentFont)
                self.__table.setItem(i, 2 + j, item)
```

七、操作方法说明及系统测试

（一） 操作方法说明

1. 确保开启 Windows 聚焦功能

使用 Windows + I 呼出设置，选择个性化，切到锁屏壁纸菜单项，将背景调整为 Windows 聚焦即可。刚调整为 Windows 聚焦时，有极大的可能源文件夹没有任何壁纸，需要等待一段时间（这一段时间可长可短），之后 Windows 便会向设备推送壁纸了。有的时候 Windows 聚焦功能也会失效，具体表现为长时间（天为单位）保持一张壁纸，这个是 Windows 聚焦的 Bug，当前不知道有没有修复。具体的解决办法可以考虑关闭一段时间（天为单位）的 Windows 聚焦，再次打开尝试即可。实在不行就要到网络上查找原因

了。

2. 确保程序所需依赖已安装

1) Windows10：1903 或以上，以前的版本应当也是可以使用的，但具体到哪一个版本不能使用因为硬件资源问题没法进行测试
2) Python3.7.1：Python3.x 应该都是支持的，未进行测试
3) Python 库 PyQt5
4) QT 所需动态库：如果装过 C++模式下的 QT 应该就不需要配置了；如果没装就需要装一下 PyQt5-tools，安装这个之后应该是可以使用的，由于我已经安装了 C++模式下的 QT，所以没对这种情况进行测试
5) MySQL：该模块是可选的，只要不使用数据库存储方式，应该就不会出现问题，但是 Python 库 PyMySql 应该还是要安装的，不然程序启动时可能会包 ImportError 而异常终止
6) 如果要使用数据库存储方式，务必修改 FilenameManager 包里的 DataBase.py 文件中硬编码的数据库用户名及密码
7) Python 库 Pillow

3. 将项目部署到本地的任何路径

只需将程序所有的代码在硬盘的任何一个目录保存即可，但是最好是一个独立的文件夹，当项目所在目录含有其他文件时，程序的行为未进行测试，但是理论上只要没有与程序脚本文件同名的文件就没有问题。

4. 如何运行程序

1) 直接使用命令行执行：在项目目录下执行 python MainWindow.py 即可
2) 使用打包程序将脚本打包为可执行程序：可以使用 PyInstaller 等打包模块进行打包，但是打包之后需要注意将项目目录下的 Images 文件夹和 settings.json 文件与可执行程序保存在同一文件夹下

5. 注意事项

1) 请勿随意修改源文件夹的内容，包括但不限于：重命名，移动（基于我之前的使用经验，移动文件可能会导致 Windows 聚焦停止更新）
2) 请确保存文件夹是独立的文件夹没有其他文件，否则程序可能会覆盖掉你的文件，并且在设置存储路径时如果选择了转移壁纸还可能会导致你的文件被一同转移
3) 不小心误修改源文件夹的路径可以使用重置源文件夹功能重置，这在做测试尤为有用，因为源文件夹内的图片通常不会超出 12 张，数据量较小，没法较好的体现程序的功

能，因此便需要实现拷贝尽可能多的源文件放到一个文件夹下，并将这个文件夹设置为源文件夹

（二） 系统测试

系统测试数据请参阅项目目录下的 TestSourceFolder，里面存储有相对比较多的源文件，测试起来效果会比较明显。之所以专门建立了一个测试源文件夹是因为当获取一定量的壁纸之后就会出现每次只能获取 1-2 张壁纸的情况，有的时候甚至是 0 张壁纸。这一点受限于 Windows 在一个推送周期内只会推送 3 幅壁纸，如果不考虑版式问题，则是 6 幅壁纸，横版和竖版壁纸往往是成对出现。

八、总结

（一） 包内模块引入问题

Python 不像 Java 那样要求用户一个类只能占用一个文件（原则上），因此 Python 从包内导入模块时，也没法直接导入包内的类，必须使用“模块名.类名”的格式才能调用相应的类。

我遇到这个问题是因为我在导入包内模块并使用之后发现 Pycharm 一直提示我 RecordLine is not callable。我排查了好大一会儿也没找到在哪里出了问题，后来询问了一下我的室友才发现这个问题。

针对这个问题的解决方案大概有 3 种：

1. 使用“模块名.类名”的格式
2. 使用“from 包名.模块名 import 类名”
3. 使用“from 包名.模块名 import *”

（二） 数据库连接总是连接失败

这是一个低级错误，我把数据库的名称传递给了用户名，所以总是提示找不到用户，但因为抛出的异常包含有数据库名，我一直以为是数据库不存在的问题，到网上进行了大量检索也没找到解决办法，虽然有类似的问题，但是显然没办法解决我的问题。后来再次查看异常才发现了问题。

（三） 虚拟环境问题

老师在课程中向我们介绍了 pipenv 虚拟环境工具，我也在本次课设中进行了使用，最初一切并没有什么异常，但是就在我基于 PyQt5 编写了一个测试界面之后发现总是会弹出信息如下的警告窗口：“This application failed to start because no QT platform plugin

could be initialized. Reinstalling the application may fix this problem.”

我也在第一时间上网进行了检索，按照网上的解决方案进行了尝试，但是并没有解决这个问题。后来花费了好长时间排查问题，后来跟使用 PyQt5 做界面的同学沟通了一下，发现他们没有建虚拟环境。因为当时已经比较晚了，我就在第二天先用命令行使用系统环境测试了一下，发现是可以运行的。然后我在 Pycharm 中将 Python Interpreter 修改为了系统环境，并适当配置了环境，按下运行按钮的那一刻有一种忐忑的心情，最后发现果然是虚拟环境的问题。

针对这个问题的解决方案，于现在的我而言就是把 Python Interpreter 修改为系统环境，问题的根源是环境配置问题，可以等以后有机会详细了解一下 pipenv。

（四） 模块引入问题

因为抱持着将存储方式转换写在相关类里的想法，就在最初把转换方法卸载了 FilenameManager 类中，这就必然需要引入子类模块，运行之后程序报错 Cannot Import 子类模块，大概是这个意思，后来又发现了子类模块也不能互相引入，这些结果都是建立在单个类占用单个文件的情况，其他情况的效果不知道。

解铃还须系铃人。针对这个问题的解决方案当然是将转换方法移动到其他位置了，想来想去把她放到了控制器里，因为控制器统揽着全局，是第二适合做这件事的类。

（五） Json 文件更新问题

第一次写入设置正常，第二次写入设置正常，但是会在文件末多出之前文件末的字符，时而一个，时而多个。这个问题发生在我变更了文件更新时使用的文件打开方式为“r+”时，出现该问题的原因是更新文件时某些字节没有被覆盖，所以会产生该问题。

针对该问题的解决方案就是将文件更新时打开文件的方式修改为“w”覆盖写模式，保证更新后的设置没有任何多余字节。

（六） PyQt5 单元格放入控件的居中问题

该问题的表现就是，向 PyQt5 的表格的单元格中压入控件，没法设置控件居中对齐，在查阅了官方文档之后发现只提供了文本对齐的方法。但是控件不能居中对齐，观感上很差。

于是我便上网进行了检索，最后发现可以在控件外包裹一个控件，并给外层的控件设置布局方式，然后将外层控件压入单元格便可以从一定程度上解决该问题。

这个时候又带来了另外一个问题，如何获取用户究竟选择了那么多行中的哪一个控件，我采用了继承 PyQt5 提供的控件，给控件增添一个属性 row，在压入单元格时同时初始化该变量，这样便可以很好的掌握控件所在的行。

（七） 删除表格行产生的问题

我在删除表格的某一行时发现不能正确删除，后来意识到是因为从前往后删除，改变了索引，所以出去第一行，往后删除的全是错误的。

针对该问题的解决方案就是倒序删除，使用 Python 内置的 reversed 翻转获取的行标列表，然后再进行删除操作。

（八） 新发现的 Python 特性

在 Python 中如果一个方法返回了一个类的属性的值，如果返回的是属性本身，那么返回的就是属性本身，对返回值的修改也会影响到属性，即使这个属性是私有成员，效果上感觉像是返回了该属性的指针，我曾经在 C++中做过测试，C++的私有成员可以通过返回它的地址实现在外部更新该成员的值而不通过 setter 方法。

这个问题没有解决方案，硬说的话就是在返回数据时拷贝一个副本。我之所以会发现这个问题，是因为我的程序中写了很多的 getter，其中有一个 get 到的数据需要在使用后回写以更新源数据，但是在我还没有编写相应的方法时，突然发现已经在自动更新了。

（九） 我的感悟

最初我选择这个题目的时候是有点担心程序的规模不够大的，我完成应用层的编写仅用了五六百行代码，这是在除去给方法和类写的 doc 的情况下。但是在我编写了控制器和界面之后，发现控制器和界面的代码远比我想象的要多得多，即使除去 doc 仍然有将近八百行代码。是虽然说代码行数不能反应项目的复杂度，但我觉得是可以在一定程度上反应项目的规模的。如果算上 doc，该项目的代码总行数达到了 1790 行，而不算 doc 也达到了 1470 行，这与我之前编写的一百多行的代码完全不是一个量级的。

在编码期间我也因为时间紧张而有过几次疏忽的地方，导致浪费了更多的时间。但是最终的结果还是好的吧。我也算是及时完成了课设，成功地独立设计并实现了一个完整的项目，而且这个项目会在我以后的生活中大量使用到，想想就满满的成就感。