

# 议程管理系统 Agenda

面向对象设计 2017 期末课程设计



## 目录

1. 项目介绍.....	2
2. 操作指南.....	2
2.1. 安装方法.....	2
2.2. 运行截图.....	2
2.3. 基本功能.....	4
3. 系统设计.....	4
3.1. 三层架构（Three-Tier） .....	4
3.2. 单例模式.....	6
3.3. 面向对象编程.....	7
4. 系统实现.....	8
4.1. 开发依赖.....	8
4.2. 代码结构.....	8
4.3. 代码编译.....	9
4.4. 系统打包.....	9
5. 难点解析.....	10
5.1. C++11 新特性 .....	10
5.2. AgendaService 虚基类的设计.....	10
5.3. Qt 信号槽机制.....	10
5.4. Qt 自包含应用.....	11
6. 参考资料.....	12

## 1. 项目介绍

本应用名字为 **Agenda**，是一个简洁实用的议程管理系统。系统提供了用户登录，新用户注册，已注册用户登陆后用户可以删除当前用户账户，查询用户名单，也可以添加、删除、查询系统中记录的会议安排等基础管理功能。

**Agenda** 使用面向对象的思想进行设计，系统提供了 JSON 和 SQLite 两种数据存储方式，提供了终端界面以及基于 Qt 的图形化界面两种显示方式。整体的系统架构为“三层架构（3 Tier）”，有效地降低了各部件之间的耦合程度，使得系统可以非常方便地进行扩展。

**Agenda** 代码开源，项目地址为：<https://github.com/chenyvehtung/Agenda>，使用 MIT 许可证。Github 统计代码行数为 3613 行，其中 C++ 占比 94.5%。代码通过 make 和 qmake 实现自动化编译，通过 shell 脚本实现 Debian Package 的自动打包。

下面，文档将分为四部分讲述本项目，包括：

1. 操作指南。介绍项目的安装方法，运行截图，基本功能等；
2. 系统设计。介绍项目的整体架构，设计模式，项目对于面向对象思想的体现等，为本文档的重点介绍内容；
3. 系统实现。介绍项目的开发依赖，代码结构，代码编译，系统打包等。
4. 难点解析。介绍项目实施过程中的一些难点和技巧。
5. 参考文档。介绍项目的一些相关参考资料

## 2. 操作指南

本项目提供了 64 位 debian 安装包，原则上可以在 64 位的 Ubuntu 或者基于 Ubuntu 的 Linux 桌面发行版上进行安装运行。如果想要在其他机器上运行，请参考系统实现章节中的开发依赖并自行进行编译。在没有特殊声明的情况下，下面的操作默认在 64 位 Ubuntu 上进行。

### 2.1. 安装方法

找到项目提供的 `agendaqt_0.1.1_amd64.deb` 文件，在该文件所在目录下打开终端，使用如下命令进行安装

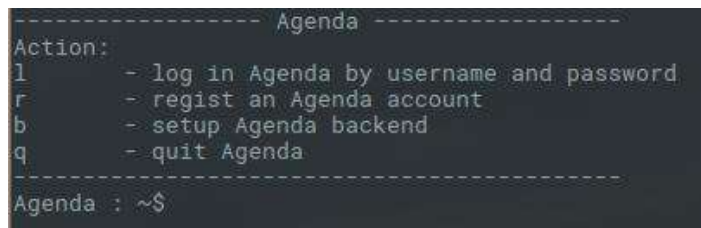
```
$ sudo dpkg -i agendaqt*.deb
$ sudo apt -f install
```

其中第一行命令用于安装 **Agenda**，第二行命令用于自动安装项目的相关依赖，本项目由于使用了 SQLite 做为数据存储方式的一种，所以运行该命令后会提示安装 `sqlite3`。

### 2.2. 运行截图

项目为了适应不同人对于 Linux 环境的偏好情况，提供了两种操作界面：终端界面和 GUI 界面。两种界面基于同一个后端，基本功能完全相同。

终端版本 **Agenda** 界面截图



```
----- Agenda -----
Action:
l      - log in Agenda by username and password
r      - regist an Agenda account
b      - setup Agenda backend
q      - quit Agenda
-----
Agenda : ~$
```

Figure 1 Agenda 初始页面

```

----- Agenda(JSON) -----
Action:
o      - log out Agenda
dc     - delete Agenda account
lu     - list all Agenda user
cm     - create a meeting
la     - list all meetings
las    - list all sponsor meetings
lap    - list all participator meetings
qm     - query meeting by title
qt     - query meeting by time interval
dm     - delete meeting by title
da     - delete all meetings
-----

Agenda@a : #
    
```

Figure 2 Agenda 用户操作页面。顶部显示当前使用的是 JSON 存储结构  
Qt 版本 Agenda 界面截图



Figure 3 AgendaQt 用户登录页面

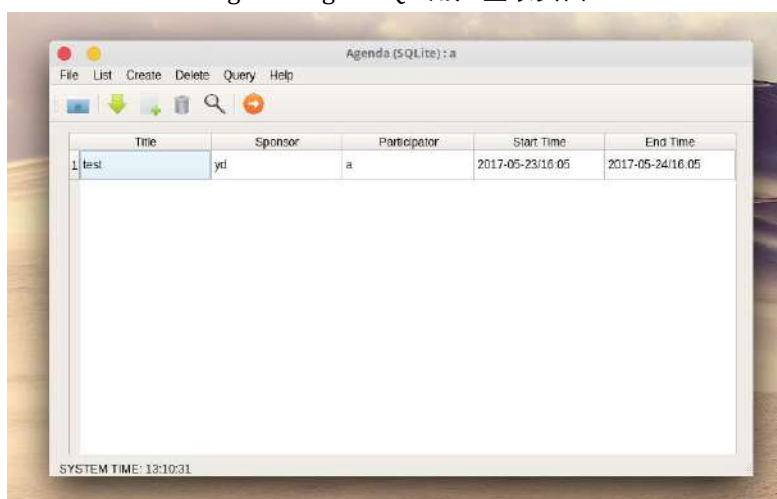


Figure 4 AgendaQt 列举用户所有会议。顶部显示当前使用的是 SQLite 存储方式

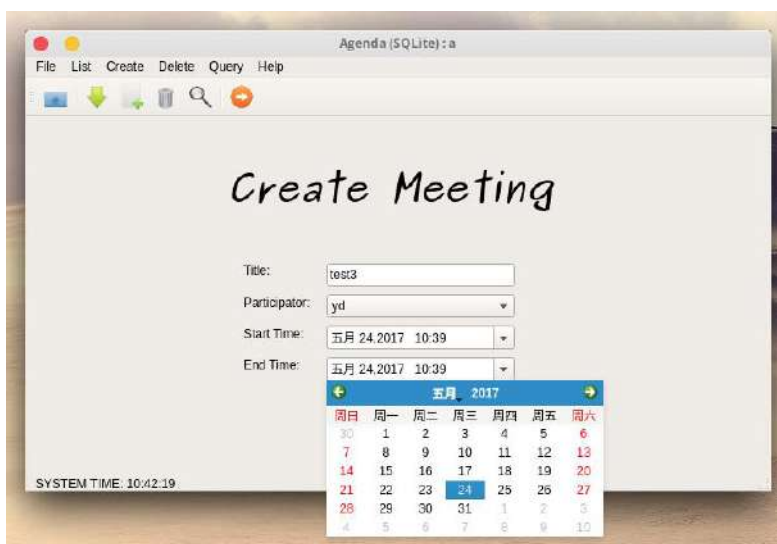


Figure 5 AgendaQt 用户创建新的会议

## 2.3. 基本功能

Agenda 的基本功能简单但完善，主要包括以下几种：

- 用户注册（提供姓名，密码，邮箱，电话），用户登录（提供姓名，密码），已注册的用户可在登录之后删除自己，被删除的用户资料（包括个人资料，会议资料等）将会被彻底地从系统中删除，保证用户信息安全。
- 在登入系统之前，可以选择使用 JSON/SQLite 作为后端数据存储。
- 已登录用户可以查看系统中所有的用户，查看自己的所有相关议程，包括用户自己创建的议程以及用户参与的议程。
- 已登录用户可以创建议程（需要提供议程标题，参与者，开始时间，结束时间等信息）。
- 已登录用户可以根据议程名字，删除自己创建的一个议程。也可以删除所有自己创建的议程
- 已登录用户可以根据名字或者时间区段查找和自己相关的议程，包括自己创建的议程以及用户参与的议程。
- .....

系统的业务逻辑和用户界面是分离，终端 UI 和 Qt UI 使用了同一份业务逻辑代码，所以两者在功能上是完全相同的，就是如上所述的几个功能。

当然，命令行界面和 GUI 在使用上会有不同，但相关的操作提示较为明显，操作简单，此处不做详述。

## 3. 系统设计

### 3.1. 三层架构（Three-Tier）

对于一个相对完整的系统来说，良好的设计结构是非常重要的。由于本系统考虑到要有两种形式的前端显示页面以及两种形式的数据存储方式，所以系统的设计的重要性就更加突出。如果简单地使用两层模式，将显示代码和业务逻辑等混合在一起编写，这种分工十分不明确的组织形式，必将造成代码的冗余杂乱，大大降低代码的可读性，系统的可维护性和鲁棒性。经过仔细的分析，最终本人采用了“三层架构（Three-Tier）”来构建整个系统。**注意，不要将 Three Tier 和 BS 应用中常用的 MVC 混淆**，具体联系与区别可以参看参考资料链接 2。下面，描述一下本系统的结构设计。

1. UI 层（表现层）：主要是指与用户交互的界面。在本系统中，UI 层包含有 terminal 和 qt5 两种界面

2. BLL (Business Logic Layer 业务逻辑层): UI 层和 DAL 层之间的桥梁, 实现业务逻辑。在本系统中, 通过一个虚基类 AgendaService 来实现, JSONService 和 SqliteService 通过继承该基类来具体化相应的功能。
3. DAL (Data Access Layer 数据访问层): 直接与底层数据打交道。对于 JSON 存储, 系统通过一个 Storage 类来实现, 向上层 JSONService 提供相应的 API, 而对于 Sqlite 存储, 本系统直接调用 SQLite 的 API。
4. Entity (业务实体): 贯穿三层, 用于在三层之间传递数据。主要将相关的数据进行“封装”, 方便各层之间的传递。在本系统中, 包含 User, Meeting, Date 三类。

根据上面的分析, 下面给出系统的架构图

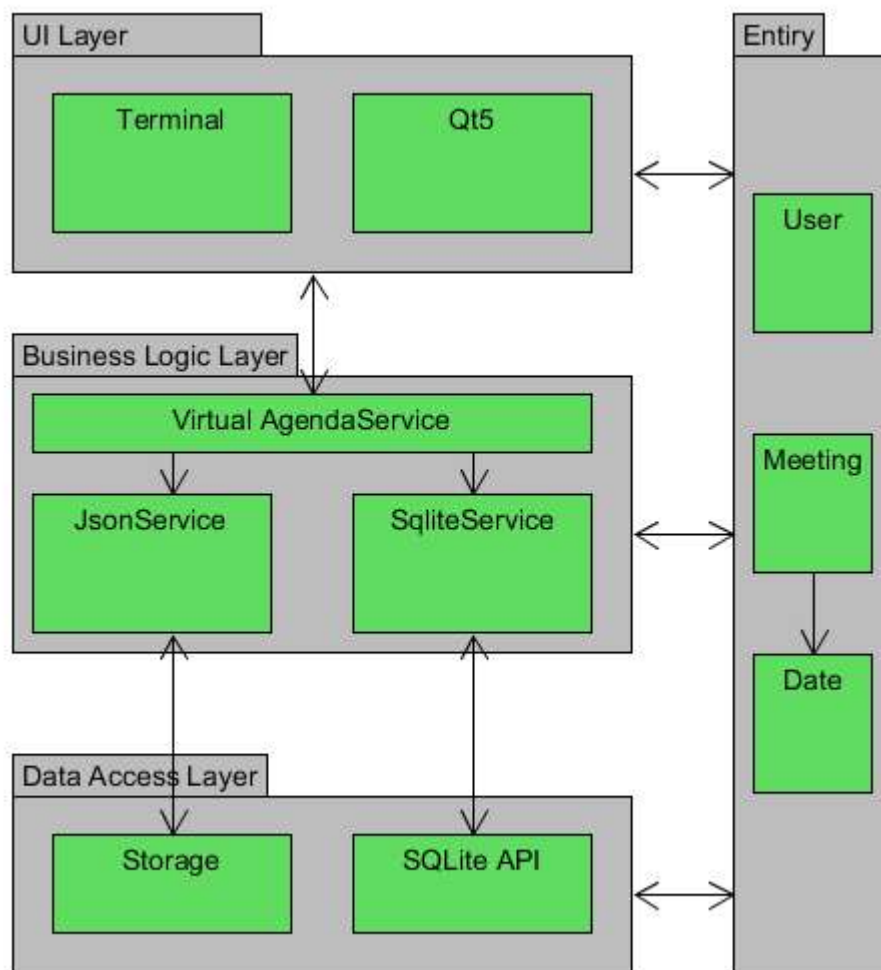


Figure 6 Agenda 系统架构图

基于这样的设计, 最后完成的系统类图如下

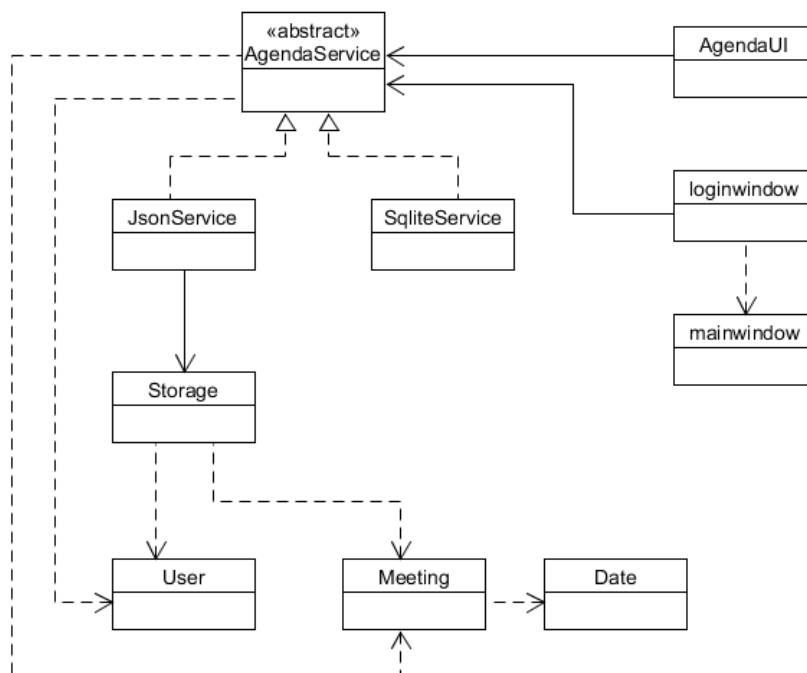


Figure 7 Agenda 系统类图

### 3.2. 单例模式

单例模式，也叫单子模式，是一种常用的软件设计模式。在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为。

实现单例模式的思路是：一个类能返回对象一个引用(永远是同一个)和一个获得该实例的方法（必须是静态方法，通常使用 `getInstance` 这个名称）；当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用；同时我们还将该类的构造函数定义为私有方法，这样其他处的代码就无法通过调用该类的构造函数来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例。

在本系统中，对于需要操作存储在 JSON 文件中数据的 `Storage` 类，就应用了以上的单例模式。这样可以确保读写 JSON 文件的一定是同一个 `Storage` 类的实例，一定程度上保证数据的统一性。核心代码如下

#### Storage.h

```

.....
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
    TypeName(const TypeName&); \
    void operator=(const TypeName&)

class Storage {
private:
    static Storage *instance_;
    DISALLOW_COPY_AND_ASSIGN(Storage);
    Storage();
    .....

```

```

public:
    // singleton
    static Storage *getInstance(void);
    .....
}

```

#### Storage.cpp

```

.....
#include "Storage.h"
//definition, fix undefine reference
Storage* Storage::instance_;
.....
Storage* Storage::getInstance(void) {
    if (instance_ == NULL) {
        instance_ = new Storage();
    }
    return instance_;
}

```

### 3.3. 面向对象编程

面向对象编程，具有三个明显的特征，分别是：封装，继承，多态。下面就本系统对这三个特征的体现做一个描述。

#### 封装

C++的类的实现，大都有封装的思想。将多个属性，一些简单的操作封装到一个类中，其他类就只需要根据这个类提供的接口进行调用，而不需要知道这个类具体是如何实现这些功能。在本系统中，对这个思想体现较为明显的，就是前面提到的实体类，包括 **User**，**Meeting**，**Date** 类。比如 **Date** 类，它提供了将字符串转换为 **Date** 类对象的功能，**Date** 类对象之间的大小对比的功能。上层类调用时，只需要直接调用接口即可，不用关注细节，让代码更容易理解与维护。**Date** 类类图如下：

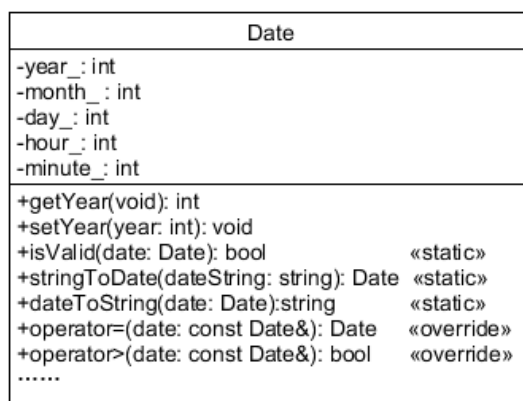


Figure 8 Date 类类图，用于说明封装

#### 继承

继承使得子类可以调用父类的一些操作，而不需要重复写冗余的代码。在本系统的业务逻辑层中，不同的控制类 **JsonService** 和 **SqliteService**，都继承于同一个父类 **AgendaService**。这样，他们都可以调用 **getServiceName** 来获取当前 **Service** 类的名称。



另外，在本系统中，AgendaService 的大多数函数都使用了虚继承的形式，把具体的实现交给继承与它的子类去实现，比如 createMeeting 等。这样，可以控制继承于它的子类，都必须去实现这些基本功能。

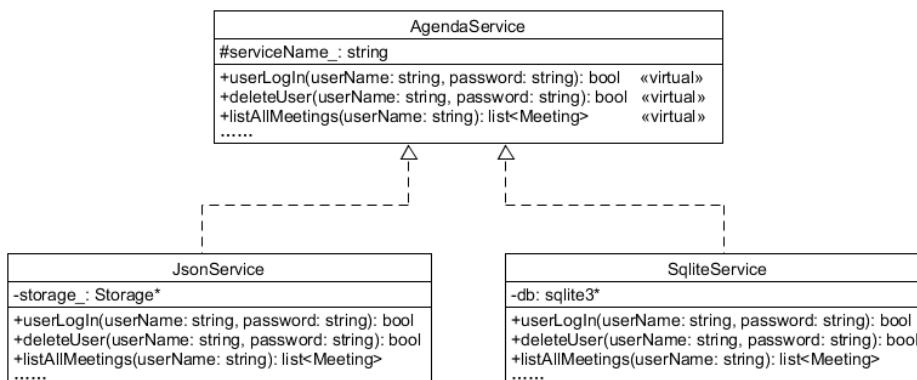


Figure 9 AgendaService 之间的继承关系

### 多态

多态是指计算机程序运行时，相同的消息可能会送给多个不同的类别之对象，而系统可依据对象所属类别，调用对应类别的方法，而有不同的行为。在本系统中，包括以下两种多态的形式：

- 动态多态。它一般通过**类继承**机制和**虚函数**机制在运行时来实现，面向对象编程中多态默认指的就是这种。比如上面所提到的 AgendaService，在 UI 层中，定义一个该对象的指针，根据需求设定其指向一个 JsonService 或者 SqliteService。这样，就可以在运行时，有不同的具体操作。本系统中，用户切换 Backend，就是通过这种方式来实现的。
- 静态多态。这种关联关系是在编译时确定的。在本系统中，主要体现在**函数重载**上。比如，在 AgendaService 中，meetingQuery 有两种参数输入形式，一种为提供 userName 和 title，另一种为提供 userName, startDate 和 endDate。程序会根据不同的参数输入，选择正确的调用函数。

```

AgendaService.h

class AgendaService {
public:
    virtual std::list<Meeting> meetingQuery(std::string userName, std::string title) const = 0;
    virtual std::list<Meeting> meetingQuery(std::string userName, std::string startDate,
                                             std::string endDate) const = 0;
};
  
```

## 4. 系统实现

### 4.1. 开发依赖

本系统在 Linux 下进行开发，使用 g++ 进行编译，需要支持 c++11 的标准。同时，项目的开发，需要依赖于 sqlite, qt5。

由于 bll 中，SqliteService 使用了 SQLite3 做为数据存储结构，所以需要先安装相关依赖。如果使用了 Ubuntu 或者基于 Ubuntu 的 Linux 发行版，可以使用如下命令来安装

```
$ sudo apt-get install sqlite3 libsqlite3-dev
```

由于 ui 中，GUI 使用了 Qt5 来开发，所以需要先安装依赖。建议到 Qt 官网下载 Qt Creator。安装好 Qt Creator 之后，Qt5 环境就可以了，同时，它也是一个非常不错的 IDE，极大的方便了开发。

### 4.2. 代码结构

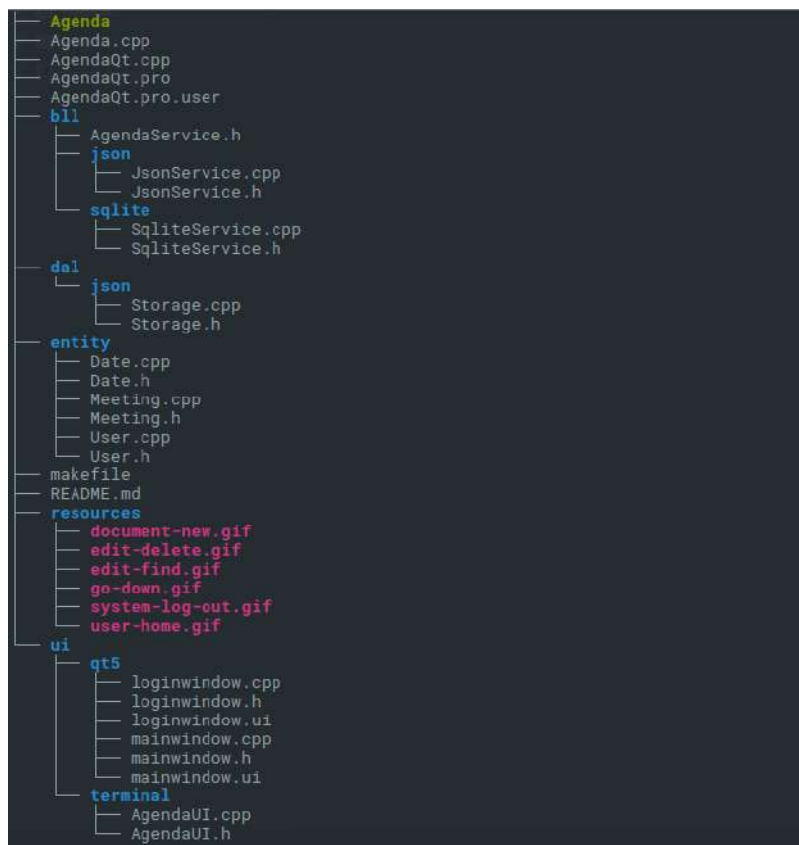


Figure 10 Agenda 系统代码结构图

结构如上所示，

- bll 里包含的是业务逻辑层代码，细分为 json 和 sqlite 的。
- dal 包含的是数据访问层的代码。
- entity 里包含的是业务实体类的代码。
- resources 里包含的是一些 Qt 前端用到的图片素材。
- ui 里包含的是展示层的代码，细分为 terminal 和 qt5 的。

#### 4.3. 代码编译

终端和 Qt 界面，这两种界面是分开编译的。

- 终端界面：在代码根目录下有一个 makefile 文件，用来自动化编译。在代码根目录下打开终端，运行 make 命令，成功之后就会生成 Agenda 的可执行文件。
- Qt 界面：
  - 如果使用 Qt Creator，那么可以直接通过根目录下的 AgendaQt.pro 文件，将整个代码导入到 Qt Creator 中，然后直接 Build & Run 即可；
  - 如果想直接在终端下编译，需要确保使用了 qt5（可以通过 qmake -v 来查看），然后运行如下命令

```
$ qmake AgendaQt.pro -r -spec linux-g++
$ make
```

成功之后，就会生成一个 AgendaQt 的可执行文件

#### 4.4. 系统打包

本系统由于是在 Ubuntu 下开发的，所以打包成较为主流的 debian package 是比较合理的。在 mkpkg 中，提供了一个自动打包的 shell 脚本 mkpkg.sh。另外，由于 Qt5 的运行时环境一般不是系统自带的，所以确保安装包的自包含（self-contain）就很关键，不然用户安装完，

很大可能会因为缺乏某些.so 文件而运行失败。本系统的自动打包脚本使用了开源软件 linuxdeployqt 来解决这个问题，具体参看难点解析中的问题 4。

打包方法，先将上述编译好的 Agenda 和 AgendaQt 两个可执行文件复制到 mkpkg/中，然后在 mkpkg/目录打开终端，运行如下命令

```
$ bash mkpkg.sh
```

成功之后，会生成一个名为 agendaqt\_x.x.x\_amd64.deb 的安装包文件。

## 5. 难点解析

### 5.1. C++11 新特性

C++11 提供了很多新的功能，合理的利用可以有效的提高代码的简洁性。比如在本系统中，对于 Storage 类，只需要提供一个函数 queryMeeting，就可以为上层代码 JsonService 提供多种方式的 Meeting 查询方法。

在 Storage 中，使用 **Function Template** 做为输入参数

#### Storage.h

```
.....
std::list<Meeting> queryMeeting(std::function<bool(const Meeting&)>
                                filter);
.....
```

在 JsonService, 使用 **Lambda Expression** 可以分别实现基于(userName, title)和基于(userName, startDate, endDate)的查询

#### JsonService.cpp

```
list<Meeting> JsonService::meetingQuery(string userName, string startDate,
                                         string endDate) const {
    return storage->getInstance()->queryMeeting(
        [&userName, &startDate, &endDate](const Meeting &meeting) {
            return ((meeting.getSponsor() == userName) || (meeting.getParticipator() == userName)) &&
                ((meeting.getStartDate() >= Date::stringToDate(startDate)) &&
                 (meeting.getEndDate() <= Date::stringToDate(endDate)));
        });
}

//the username here can be sponsor or participator
list<Meeting> JsonService::listAllMeetings(string userName) const {
    return storage->getInstance()->queryMeeting(
        [&userName](const Meeting &meeting) {
            return (meeting.getSponsor() == userName) ||
                (meeting.getParticipator() == userName);
        });
}
```

### 5.2. AgendaService 虚基类的设计

上层 ui 代码在调用 AgendaService 时，由于使用不同的存储方式而要用不同的代码，本系统巧妙的应用了 C++ 的多态方法。将 AgendaService 做为 JsonService 和 SqliteService 的虚基类。这样，在 ui 层中，声明一个 AgendaService 类型的指针，同时声明一个 JsonService 和一个 SqliteService 的对象。这样，动态的将 AgendaService 指针指向两个不同的类对象，就可以动态改变具体操作。具体可以看 AgendaUI 类或者 loginwindow 类的实现代码。

### 5.3. Qt 信号槽机制

在本系统的 Qt 界面中，主要有用户登录窗口和已登录用户的主窗口两个窗口。由于用户的登录和注销操作，需要在这两个窗口之间进行切换。直觉情况下，打开新窗口可能会直接考虑如下的方式

```
MainWindow *mainWin = new MainWindow();
```

```
mainWin->show();
this->close();
```

但是，由于当前窗口类被关闭了，所以 new 出来的对象就无法手动删除，这样就会造成内存泄漏。如果用户不断登录，注销，后果比较严重。另外，由于每次创建一个窗口时，后端会进行数据的读写，这样有很大可能会造成数据的重复读写。

所以，可以在程序开始运行时，创建两个窗口对象，然后通过控制 hide 和 show 来实现窗口的交替出现。但是，如何从一个窗口 show 出另一个窗口呢？这里就需要应用到 Qt 的信号槽机制了。

信号槽是 QT 中用于对象间通信的一种机制，也是 QT 的核心机制。在本系统中，可以在 MainWindow 中创建一个 mainWinClose 的信号，每次要隐藏 MainWindow 时，就 emit 它。同时，在 LoginWindow 中创建一个 showLoginWin 的槽，用于接收 mainWinClose 的信号。部分核心代码如下：

#### loginwindow.h

```
class LoginWindow : public QMainWindow {
slots:
    void showLoginWin();
.....
}
```

#### loginwindow.cpp

```
if (mainWindow == NULL) {
    mainWindow = new MainWindow();
    connect(mainWindow, SIGNAL(mainWinClose()), this, SLOT(showLoginWin()));
}
mainWindow->updateWin(username, password, agendaService_);
mainWindow->show();
hide();
```

#### mainwindow.h

```
class MainWindow : public QMainWindow {
signals:
    void mainWinClose();
.....
}
```

#### mainwindow.cpp

```
if (ans == QMessageBox::Yes) {
    emit mainWinClose();
    hide();
}
```

## 5.4. Qt 自包含应用

如果只是将 Qt5 编译生成的可执行文件进行发布，那么，没有配置 Qt5 环境的用户，运行该文件时，一般会提示某些.so 文件缺失。比如，对于本系统，出现了如下的报错：

```
donald@EOS-LAB:~/Downloads/Telegram Desktop$ ./AgendaQt
./AgendaQt: /usr/lib/x86_64-linux-gnu/libQt5Gui.so.5: version 'Qt_5' not found (required by ./AgendaQt)
./AgendaQt: /usr/lib/x86_64-linux-gnu/libQt5Core.so.5: version 'Qt_5.7' not found (required by ./AgendaQt)
./AgendaQt: /usr/lib/x86_64-linux-gnu/libQt5Core.so.5: version 'Qt_5' not found (required by ./AgendaQt)
./AgendaQt: /usr/lib/x86_64-linux-gnu/libQt5Widgets.so.5: version 'Qt_5' not found (required by ./AgendaQt)
```

Figure 11 AgendaQt 运行报错

解决这个问题的办法，就是先找出所编写应用所需要依赖的 Qt 库文件和插件，然后手动加到安装包中，这样，生成的应用安装就是自包含(self-contained)的了。这里推荐使用开源软件 linuxdeployqt 来查找相关依赖，然后在 build 安装包时加入这些依赖文件，具体可以看参考资料 3。

## 6. 参考资料

1. [三层架构\(我的理解及详细分析\) - 韩学敏 专栏 - 博客频道 - CSDN.NET](#)
2. [mvc 与三层结构终极区别 - 曹胜欢 - 博客频道 - CSDN.NET](#)
3. <https://github.com/probonopd/linuxdeployqt>

[THE END]

2017-05-26