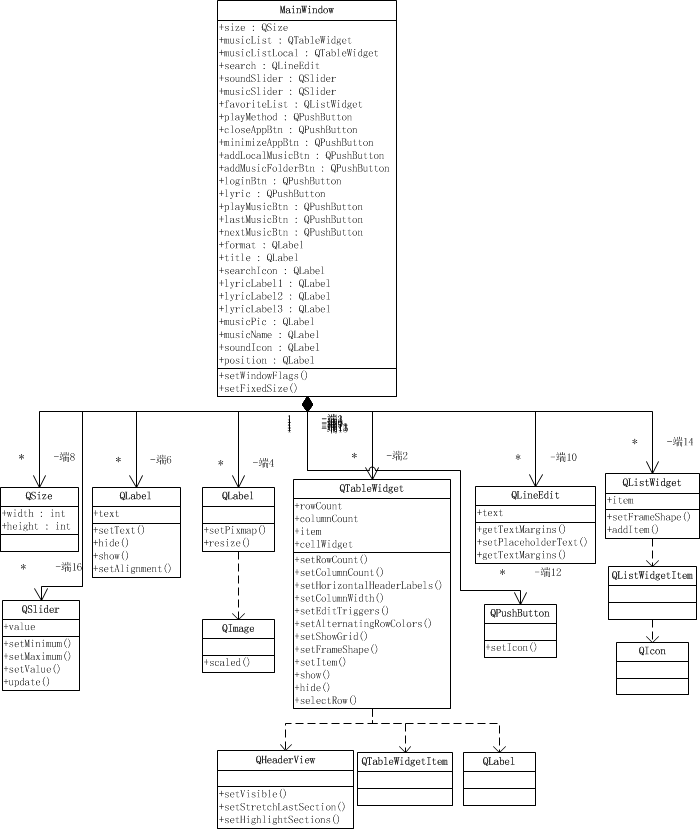
# 一、UI部分



由于qt的QLabel类既可以用来显示图片，也可以用来显示文字，所以图中出现了两个QLabel类。

主界面是一个MainWindow类，其中包含很多控件类——关闭按钮（closeAppBtn)、最小化按钮（minimizeAppBtn)、登录按钮（loginBtn）、播放按钮（playMusicBtn）、上一首按钮（lastMusicBtn）、下一首按钮（nextMusicBtn）、循环方式按钮（playMethod）、歌词显示切换按钮（lyric）、标题（title）、标题图片（format）、搜索图标（searchIcon）、歌词显示（lyricLabel1、lyricLabel2、lyricLabel3）、歌名显示（musicName）、播放时间显示（position）、音量图标（soundIcon）、歌单列表（favoriteList）、歌曲信息列表（musicList、musicListLocal）。

首先是qt的ui设计模式对这些控件进行排版，确定大致位置后开始进行逐一美化。背景颜色模式我重载了MainWindow的paintEvent函数利用QPainter等类进行底色的绘制。

对于界面中出现的图片和文字我基本都是用QLabel实现的，文字显示的QLabel较为容易，使用setText函数即可，歌词显示为了居中，采用setAlignment设置。而图片显示的QLabel一开始使用QPixmap载入图片再传入QLabel，发现原图缩放后会出现较为严重的锯齿问题，后来查文档发现，主要是由于Qt的图片缩放为了效率，多采用快速缩放模式，对于图片的破坏较大，后来查找文档发现可以使用QImage类，配合scale函数采用平滑缩放，最后获得的图片显示就没有什么严重的锯齿问题了。

按钮部分，playMusicBtn会随着歌曲的播放状态改变样式、音量图标在静音时也会改变图片，主要使用setIcon函数载入新图片，至于触发事件使用的是qt内的信号机制，一旦音量改变，则做相应判断和改变。

搜索栏（search）通过setPlacehoderText设置提示占位符为“SEARCH”，通过getTextMargins获取默认的上下左右边距，然后setTextMargins对其做出调整。

对于音乐信息列表（musicList、musicListLocal）用setRowCount和setColumnCount设置函数和列数、setHorizontalHeaderLabels用于设置表头的文字，表头是一个QStringList类，需先构造后传入。在用setColumnWidth设置每一栏的长度、在用setStretchLastSection让最后一列自行扩展恰好达到表的宽度。setSelectBehavior设置使表每次选择一行而不是一个单元格。setShowGrid可以隐藏起网格线，使其更美观。setAlternatingRowColors开启间隔色模式，相邻两行的颜色不同，在没有网格线的情况下采用这种方式非常合适，至于颜色的设置在qss中进行。同时表只负责显示信息，不能编辑，采用setEditTrggers使其只读。由于行表头没有必要，所以用verticalHeader的setVisable让其隐藏起来。信息列表的每个单元格会要用setItem或者setCellWidget绑定一个QTableWidgetItem类地址后才能进行设置内容等操作，所以我一开始就为表申请了空间，方便后续操作，也方便析构防止内存泄漏。

歌单（favoriteList）的每一行由一个图标和文字组成，先用QIcon和QString构造一个QListWidgetItem，然后使用addItem添加到表中。

进度条和音量条（musicSlider、soundSlider）采用setMinimum和setMaximum设置最大和最小值。setvValue用于设置当前值。同时进度条的valueChange信号将用于改变歌曲播放进度和歌词显示。

所有按钮都采用点击触发，qt有自带的槽函数，然后在其中调用对应接口函数，根据需要修改图标即可。

搜索栏采用回车按键触发。

歌单采用单击触发，选中指定行。

歌曲信息表单击第一列的删除图标可以删除本地歌曲，单击也会选中当前行，双击则播放当前行的歌曲，这两个槽函数都有提供，在其中调用接口函数。

本地歌曲和在线歌曲分开两张表存储方便管理。

播放进度条，当前时间，歌名会与播放模块对应信号绑定，实时修改相应内容。

歌词显示部分我使用3个QLabel分别显示上一条当前下一条歌词。通过解析lrc文件，获得歌词的时间轴和对应内容，根据播放进度信号，寻找复合要求的当前歌词，再将另外两条载入相应位置。一开始由于若将当前时间拖动到最后一条歌词后会找不到，导致崩溃，后来采用QMap对歌词进行管理很好地解决了这个问题。

以上主要是对控件格式及功能的实现，样式的美化则独立采用qss实现。具体不多加展开，主要就是针对每个控件的具体细节做定制，如搜索栏，设置其圆角半径、字体颜色等。

我将所有需要用到的图片及qss文件加入资源文件中，采用了相对路径，在不同平台都可直接编译。

# 二、Commander、Network部分

**Commaner简略UML图：**

软件的ViewModel层是通过一个个接口(interface)实现的。一个能够调用各种接口的Commander类与需要调用Model层数据的View层绑定，实现了View层与Model层的联系，而又消除了传统模式中存在大量View层直接调用接口或者Model层现象导致的高耦合问题，大大增强了代码独立修改和重用的能力。

Commander中每个函数对应一个具体功能，通过创建一个接口类指针，指向一个实现了接口的对象，通过这个对象完成所需的操作。一个接口可以由多个对象实现，Commander只需要利用接口指针指向具体对象调用方法。

这不同于传统的继承模式。继承模式是通过利用特殊化的类包含了相对一般化的父类的特征这一特性，实现了代码重用，然而当父类对象进行变更或者子类对象所需功能发生较大改变时，难免出现同时修改父类和子类来协调继承关系的问题。而实现接口的对象是具有某种规则的对象，虽然不能像继承一样重用部分变量和方法，但好处在于各个对象之间低耦合，增减功能时只需要在对具体的一个类进行修改即可，很好地满足了工程项目多人合作、各自独立开发并进行单元测试的需要。

**Network UML图：**



NetWork部分主要实现的是歌曲名称、歌手名称mp3链接、专辑图像链接、lrc链接等相关信息的抓取和分析。

一个Search类用来表示整个网络搜索的类，下面继承的MusicSearch和LrcSearch类分别实现在线歌曲搜索和歌词搜索的功能。

向网易云音乐提交http请求的api链接、post data、header等信息，由子类各自实现的MakePostData函数实现，其中调用了父类的SetApiFor…函数。搜索的执行统一由 Dosearch函数完成，期间具体实现：

用QNetworkManager类提交QNetworkRequest类的请求，调用QLoop执行循环，知道收到服务器返回的QReply才推出循环，表示请求已经结束并获得结果，再讲接收到的Json数据存到QJsonObject类型的searchObj成员变量中，留给不同的子类进行对应的分析。MusicSearch和LrcSearch中的GetMusicList和SearchLrc就进行了相应分析。（lrc的分析由于需要根据歌曲id再调用一次API进行搜索，所以需要在现有结果上再进行一次搜索分析。

此外因为需要对抓取下来的图片链接进行处理，所以还设置了一个ImageDownload类来提供处理图片的功能。

关于测试：

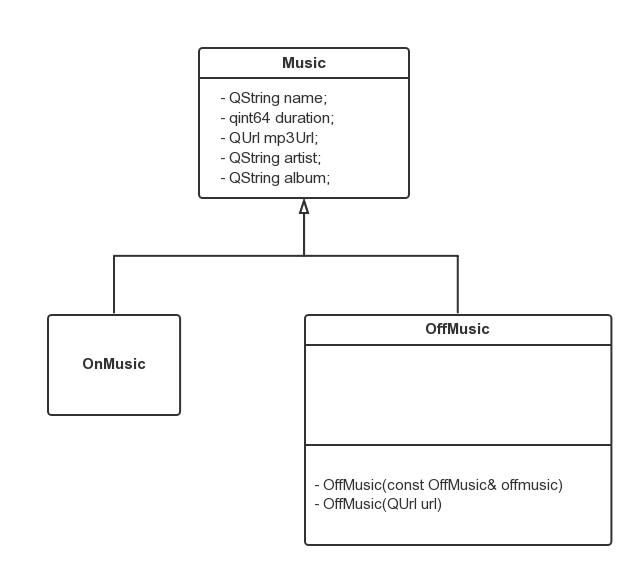
为MusicSearch和LrcSearch类都实现了测试模块，进行单元测试。对提交的request和返回的结果（code和json数据）进行检验，看是否符合预期，并对分析出来的歌曲和歌词结果与测试模块内的结果进行了比对，确保了类功能的正确性。

# 三、Music、PlayList、PlayListRecord

OffMusic内部实现

## 实现原理

Music类拥有数个需要提供给用户的信息，具体表现为交给protected属性，包括音乐名称，歌曲时长，歌曲路径，歌手名称以及专辑名称。而OffMusic类继承于Music类，自然继承了以上的protected属性。不同的是以url为参数进行构造，在构造函数中使用第三方库Taglib解析出需要的以上信息。



图表 1 OffMusic UML图

## 实现过程

1. 首先要下载Taglib的源代码，按照其中INSTALL文件的指示使用CMake和Make进行编译。为了以后方便他人对整个工程进行编译，我选择了编译出了对应的静态库libtag.a。使用Qt Creator添加工程需要用到的libtag.a，并且将第三方库的include文件添加到整个工程的include路径之中。
2. 撰写相关代码:

#ifndef OFFMUSIC

#define OFFMUSIC

#include "music.h"

#include "3rdpart/taglib/include/fileref.h"

//继承Music类

class OffMusic : public Music

{

public:

OffMusic() {}

OffMusic(const OffMusic& offmusic){

name = offmusic.name;

duration = offmusic.duration;

mp3Url = offmusic.mp3Url;

artist = offmusic.artist;

album = offmusic.album;

}

//使用url构造OffMusic类

OffMusic(QUrl url){

//先用url读入文件构造Taglib自己的文件解析类FileRef

TagLib::FileRef ref(url.path().toLatin1().data());

//检测是否成功

if(!ref.tag())

{

qDebug() << "NULL";

}

//调用FileRef的成员函数tag()获取标签，进而读取各项歌曲信息

name = QString(ref.tag()->*title*().toCString());

duration = ref.audioProperties()->*length*()\*1000;

mp3Url = url;

artist = QString(ref.tag()->*artist*().toCString());

album = QString(ref.tag()->*album*().toCString());

//如果使用taglib获取不了歌曲名称，则使用url中最后一段获取歌曲名称

if(name.isEmpty())

{

QString qstr = url.fileName();

QStringList list = qstr.split(".");

if(!list.isEmpty())

name = list.first();

else

name = "UNKNOWN";

}

}

virtual ~*OffMusic*(){}

};

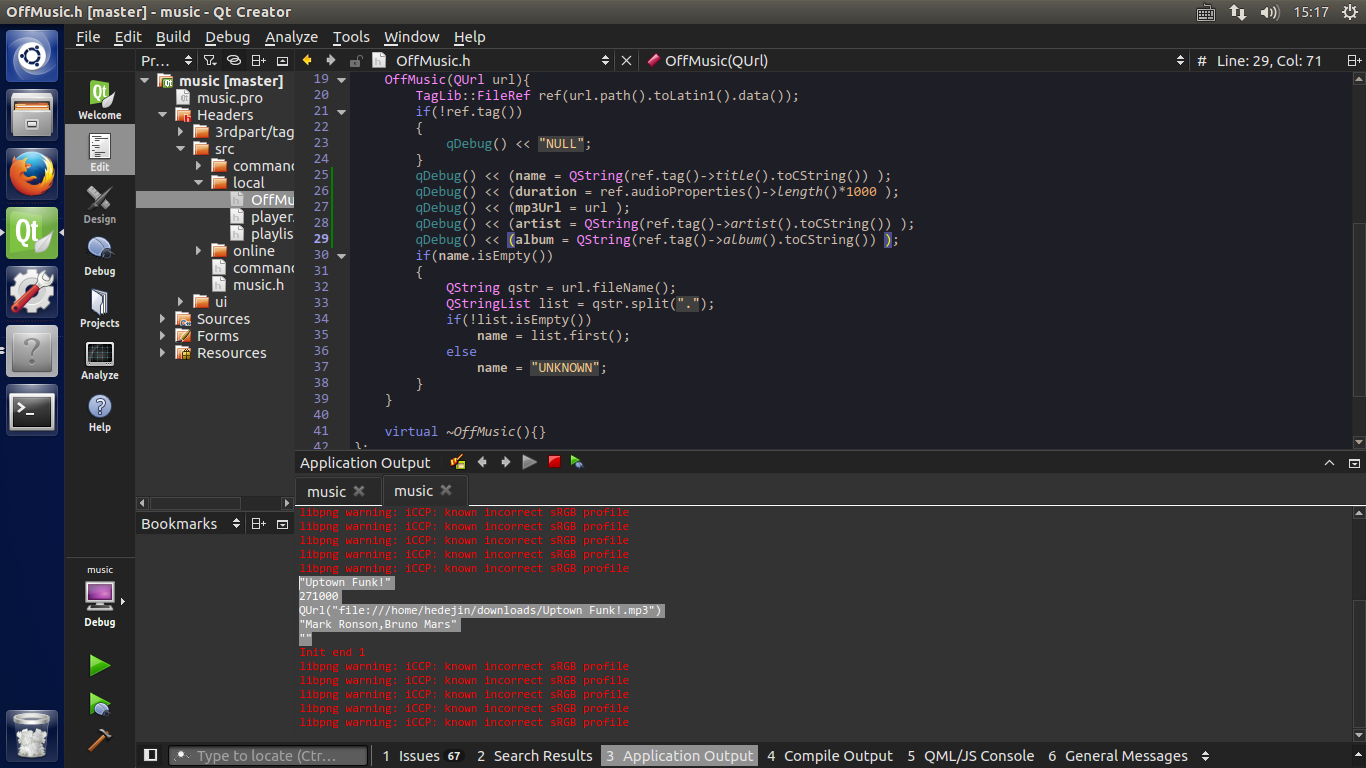
#endif

1. 编译时会提示需要源文件id3v2frame.cpp才能进行编译，再把需要的id3v2frame.cpp从Taglib源代码中找出，并添加到工程之中，qmake之后即可完成编译。

## 实现效果

当传入一个QUrl参数时，可以构造出一个含有歌曲信息的OffMusic对象。

OffMusic测试记录



从下方程序输出的debug()信息可以看到，在OffMusic构造的过程中成功读出并记录了歌曲名称、歌曲时长、url、专辑名称等信息。

关于url输入是否为空或是否为本地文件的判断，在Player类的加入文件阶段已经完成，故无需再次测试。

PlaylistRecord内部实现

## 实现原理

PlayerlistRecord.h中有两个函数，一个是void SyncMediaList(const QMediaPlaylist\* )，另外一个是void InitMediaList(QMediaPlaylist\* , QVector<OffMusic> &)。

每次程序结束，在MainWindow的析构函数中调用void SyncMediaList(const QMediaPlaylist\* )，将此时的QMediaPlaylist中的url列表转化为QJsonArray然后进一步转化为json文件存储下来。

在程序开始MainWindow初始化的时候调用void InitMediaList(QMediaPlaylist\* , QVector<OffMusic> &)。该函数将当前目录下的PlaylistRecord.json读入，并转化为QmeidaPlaylist中的ur表单，之后可以再由View部分读出QMediaPlaylist中的内容，显示出上一次退出程序时的歌单。

图表 2 PlaylistRecord UML图

## 

## 实现过程

代码如下：

#ifndef PLAYLISTRECORD

#define PLAYLISTRECORD

#include <QJsonDocument>

#include <QJsonArray>

#include <QFile>

#include <QMediaPlaylist>

#include <QException>

#include "OffMusic.h"

//当程序结束调用同步失败时，抛出SyncPlaylistFail异常

class SyncPlaylistFail : public QException

{

public:

void *raise*() const { throw \*this; }

SyncPlaylistFail \**clone*() const { return new SyncPlaylistFail(\*this); }

};

//从json文件中读取url，载入QMediaPlaylist中

inline void InitMediaList(QMediaPlaylist\* playlist, QVector<OffMusic> &qvec)

{

//打开PlaylistRecord.json文件并转化为QJsonDocument

QFile file("PlaylistRecord.json");

QJsonDocument json;

if(file.*open*(QIODevice::ReadOnly | QIODevice::Text))

json = QJsonDocument::fromBinaryData(file.readAll());

//从QjsonDocument中获取QJsonArray，遍历得到上次存入的所有url

QJsonArray jsonarr = json.array();

for(int index = 0; index < jsonarr.size(); index++ )

{

QUrl url(jsonarr[index].toString());

//同时构造OffMusic返回给上层，给用户显示歌曲信息

OffMusic omus(url);

qvec.append(omus);

playlist->addMedia(url);

}

qDebug() <<"Init end"<< playlist->mediaCount();

}

//遍历QMediaPlaylist将url存储为json文件

inline void SyncMediaList(const QMediaPlaylist\* playlist)

{

QJsonArray jsonarr;

QJsonDocument json;

//遍历QMediaPlaylist，将url放入QJsonArray中

int cnt = playlist->mediaCount();

for(int index = 0; index < cnt; index++)

{

QUrl url(playlist->media(index).canonicalUrl());

jsonarr.append(url.toString());

}

//将QJsonArray转化为QJsonDocument

json.setArray(jsonarr);

//将QJsonDocumemt转化为QByteArray并写入PlaylistRecord.json文件

QByteArray byteArr = json.toBinaryData();

QFile file("PlaylistRecord.json");

if(file.*open*(QIODevice::WriteOnly | QIODevice::Text))

{

file.write(byteArr);

}

else

{

throw SyncPlaylistFail();

}

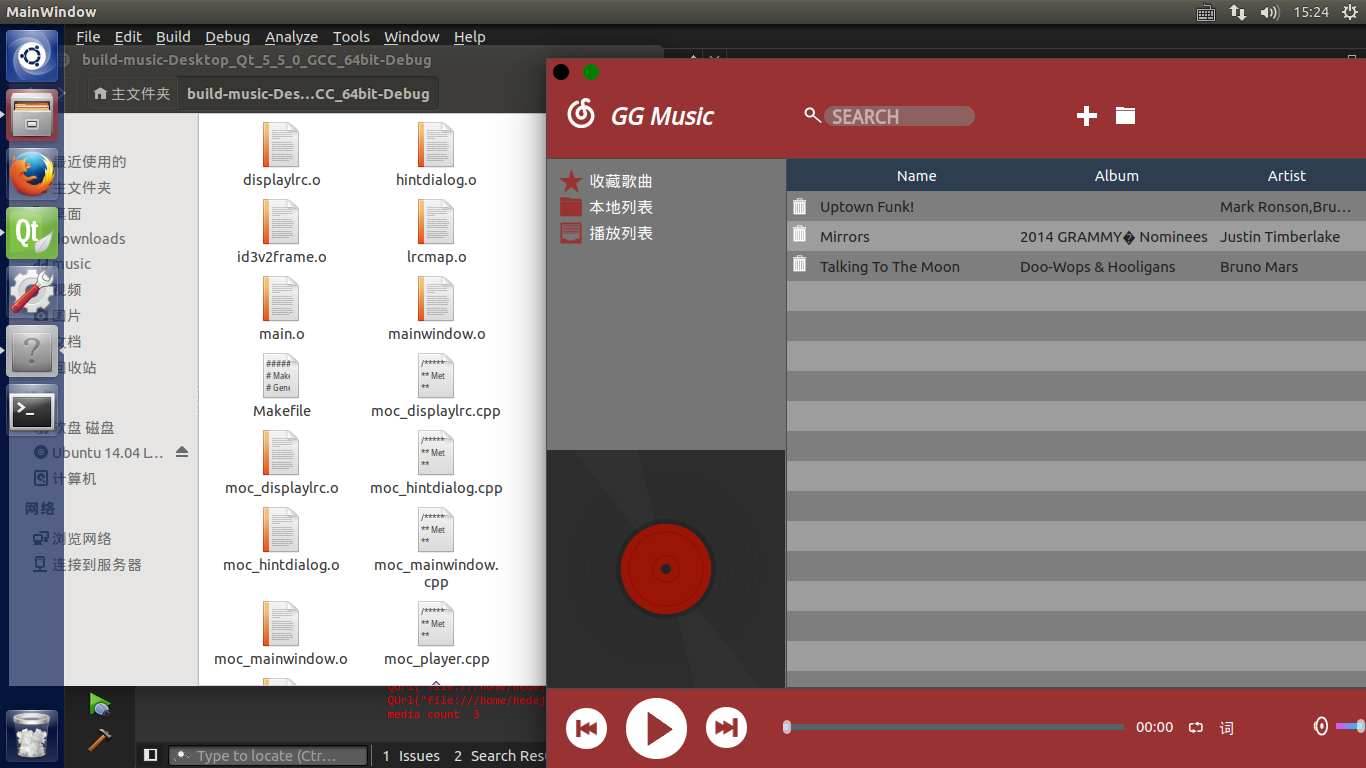
}

#endif // PLAYLISTRECORD

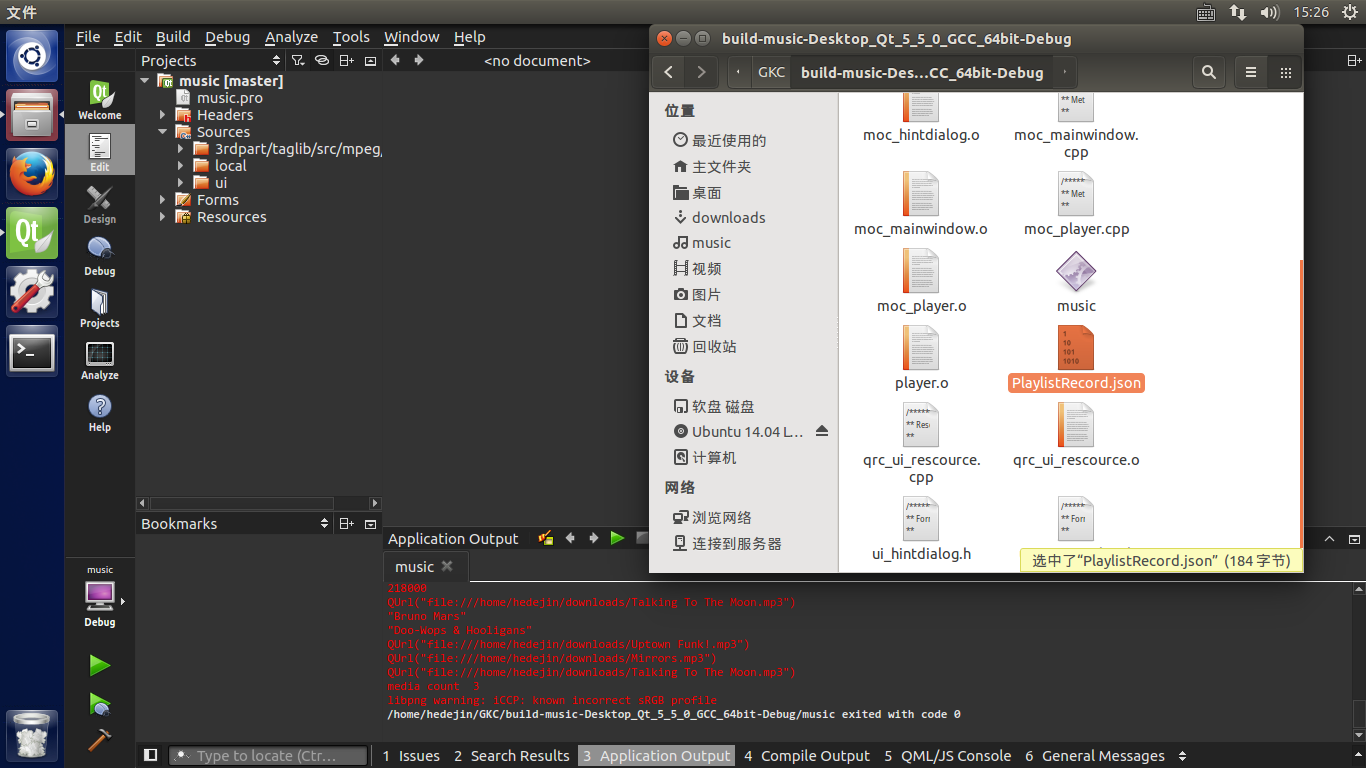
## 实现效果

每次点击窗口左上角的的退出标识进行安全退出，下一次打开程序时可以看到一模一样的歌单出现在本地列表之中。

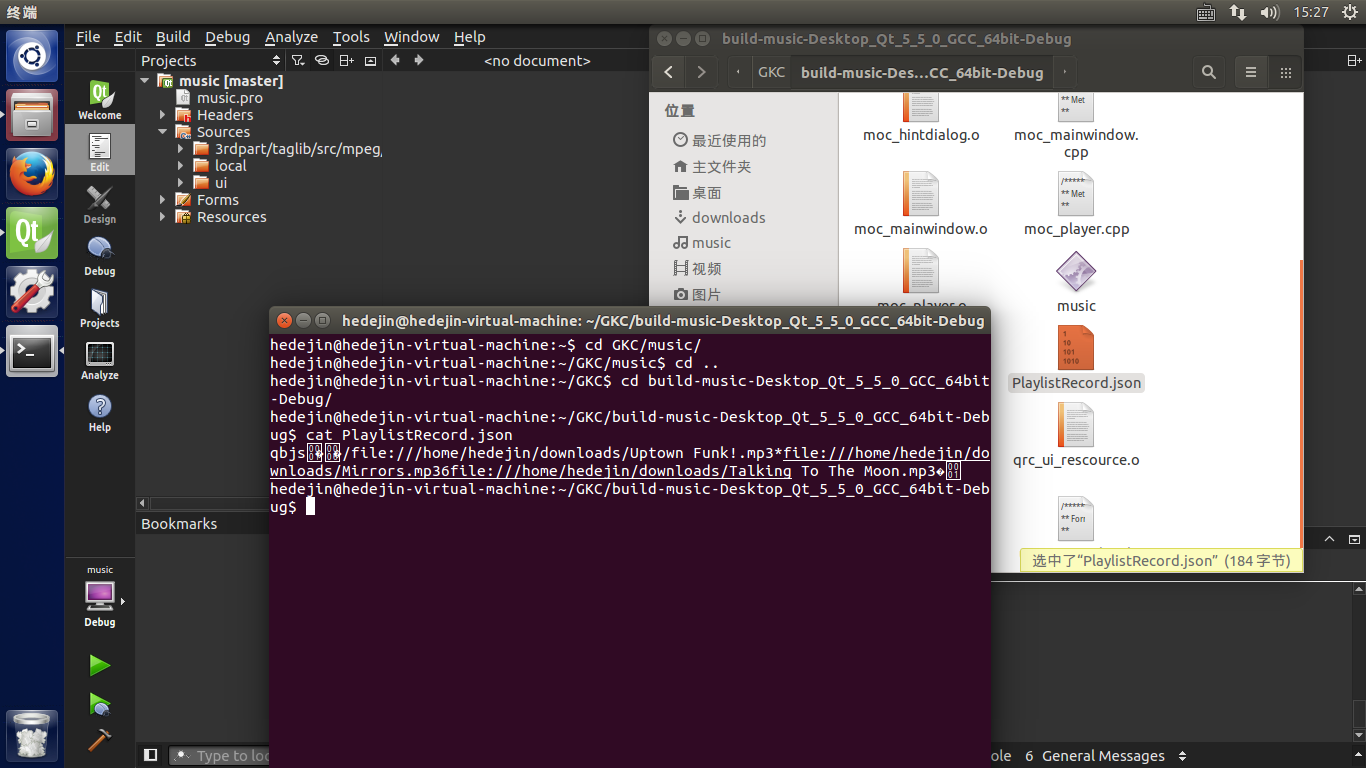
PlaylistRecord测试记录



假设程序第一次运行，对应的文件夹中不存在对应的PlaylistRecord.json文件。在添加三首歌后点击按钮退出程序。



则可以观察到生成的json文件。使用命令行可以观察到内部的部分信息。



再次打开程序。可以观察到三首歌已经被加载到本地列表之中。

