作业报告

——音乐播放器

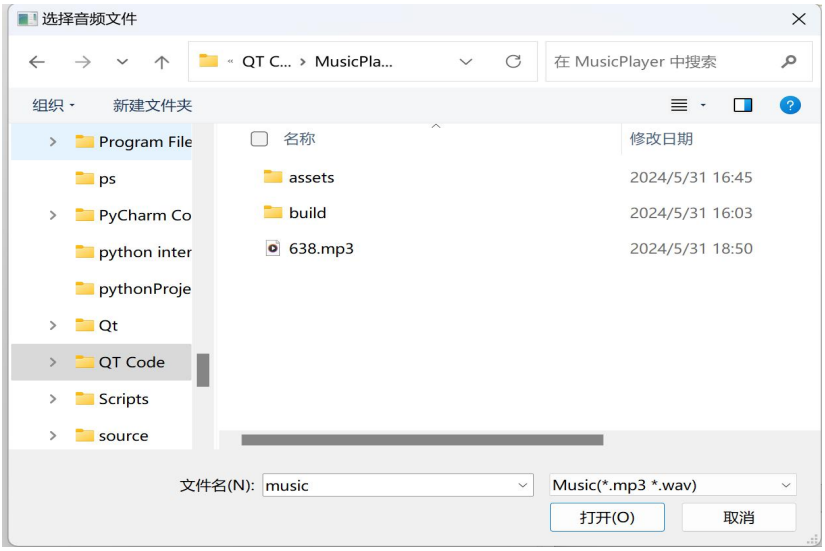
一、功能介绍

我们小组制作的音乐播放器主要有选择文件、数据库、单曲循环、上一首/下一首、暂停/播放、随机播放、音量调节、通过进度条改变音乐播放进度等功能。

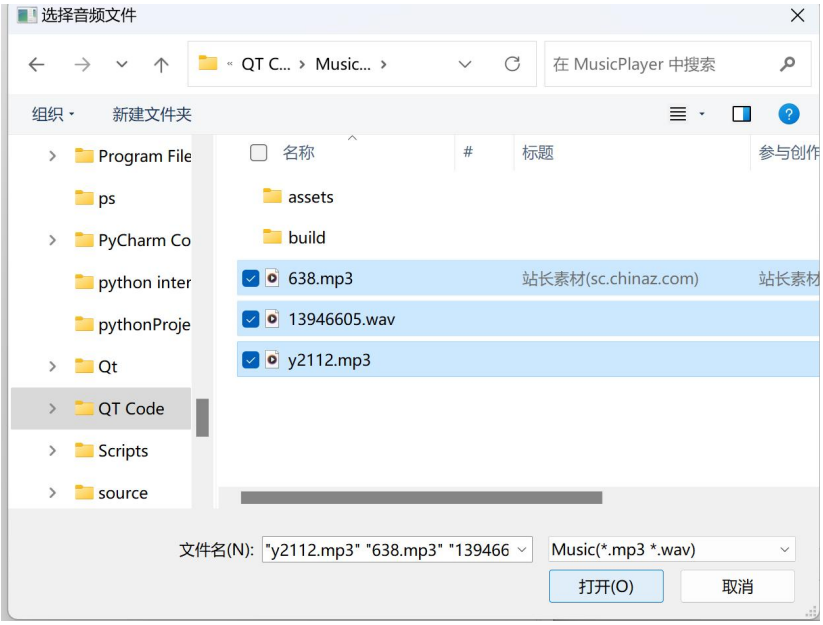
（一）选择文件

1、鼠标移动至按钮处时会有工具提示

2、点击按钮弹出选择音频文件的界面



3、支持同时选择多个文件



4、若选择文件夹则添加文件夹中所有音频文件

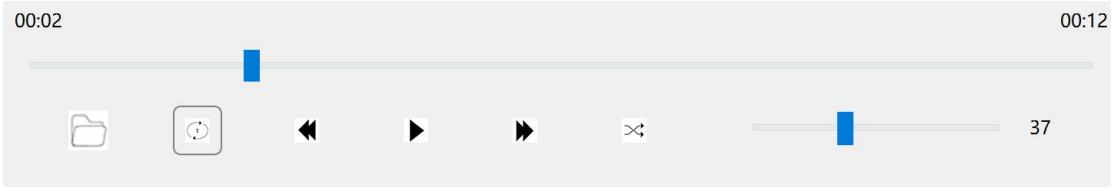
5、**数据库功能：**能记录添加过的歌曲，不必每次运行都重新添加

6、所选文件会在界面上显示，若已有列表则添加在已有列表末尾

（二）单曲循环/随机播放

1、通过点击相应按钮来实现

2、单曲循环或随机播放时，相应的按钮会产生灰色边框



（三）上一首/下一首

1、默认状态下自动播放下一首

2、通过点击对应按钮实现播放上一首/下一首

3、未选择文件时点击上一首/下一首不会导致崩溃

4、“上一首”：若当前是第一首则播放最后一首

5、“下一首”：若当前是最后一首则播放第一首

（四）暂停/播放

1、双击某一音频文件可以直接播放

2、选中某一音频文件后点击播放按钮也可以播放

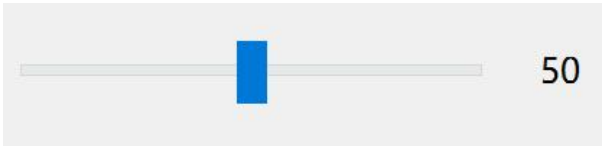
3、按下播放/暂停对应按钮后能够实现图标的切换

4、未选择文件时点击播放不会导致崩溃

（五）音量调节

1、能够显示当前音量，对应音量为系统音量的 100%

2、可以通过拖动或点击操作来调节音量。其中每点击滑块所在位置右侧滚动条一次音量增加 10；每点击滑块所在位置左侧滚动条一次音量减小 10。

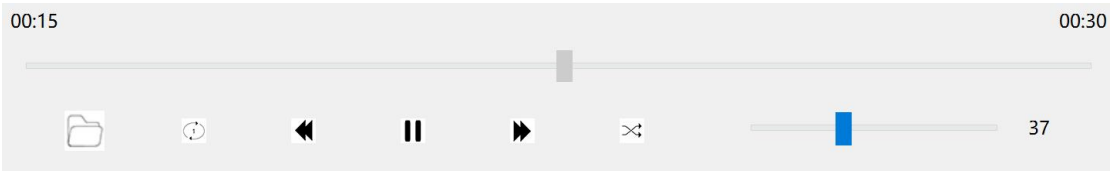


（六）进度条

1、会随音频播放进度移动

2、可以拖动使音乐播放器播放音频的对应部分（拖动时进度条上方块显示为灰色）

3、进度条上方显示音频总时长和当前播放时长



二、项目各模块与类设计细节

（一）模块

1. Widget 类：

该类继承自 QWidget，是整个播放器的核心类，负责初始化 UI 组件、媒体播放器、音频输出设备、数据库连接和信号槽的连接。

2. QMediaPlayer 和 QAudioOutput:

QMediaPlayer 用于媒体播放，QAudioOutput 用于音频输出，这两个对象用于控制音乐的播放和音量。

3. 信号和槽机制：

Qt 的信号和槽机制用于响应用户的交互，例如播放、暂停、拖动进度条等。通过 connect 函数将信号与槽函数连接起来。

4．数据库操作：

使用 Qt 的 SQL 模块管理音乐文件的历史记录，将音乐文件的信息存储在 SQLite 数据库中，以便下次启东市可以加载历史记录。

（二）类设计细节

1．Widget 类的构造函数

（1）UI 初始化：ui->setupUi(this);

（2）初始化媒体播放器和音频输出设备：

```
audioOutput = new QAudioOutput(this);
mediaPlayer = new QMediaPlayer(this);
mediaPlayer->setAudioOutput(audioOutput);
```

（3）连接媒体播放器的信号和槽：

```
// 获取当前媒体的时长，通过信号关联获取
connect(mediaPlayer,&QMediaPlayer::durationChanged,this,[=](qint64 duration)
{
    ui->totalLabel->setText(QString("%1:%2").arg(duration/1000/60,2,10,QChar('0')).arg(duration/1000%60,2,10,QChar('0')));
    ui->playCourseSlider->setRange(0,duration);
});
// 获取当前播放时长
connect(mediaPlayer,&QMediaPlayer::positionChanged,this,[=](qint64 pos)
{
    ui->curLabel->setText(QString("%1:%2").arg(pos/1000/60,2,10,QChar('0')).arg(pos/1000%60,2,10,QChar('0')));
    ui->playCourseSlider->setValue(pos);
});
//拖动滑块，让音乐播放的进度改变
connect(ui->playCourseSlider,&QSlider::sliderMoved,mediaPlayer,&QMediaPlayer::setPosition);
//
// audioOutput->setVolume(0.5);
connect(mediaPlayer,SIGNAL(playbackStateChanged(QMediaPlayer::PlaybackState)),this,SLOT(playNext()));
```

（4）设置音量滑块的范围和初始值：（分别为 1~100,37）

```
ui->horizontalSlider->setRange(0,100);
ui->horizontalSlider->setValue(37);
int val = ui->horizontalSlider->value();
ui->label->setText(QString::number(val));
```

（5）数据库连接和表创建：

```
// 数据库搭建
// 1、连接数据库，如果不存在则创建
if (QSqlDatabase::contains("Songs")) // 检查是否存在名为"song"的数据库连接
{
    db = QSqlDatabase::database("Songs"); // 如果存在，则获取该数据库连接
}
else
{
    db = QSqlDatabase::addDatabase("QSQLITE"); // 如果不存在，则添加一个SQLite数据库连接
    db.setDatabaseName("Songs.db"); // 设置数据库文件名为"song.db"
}

// 2、打开数据库，读取数据表
if (!db.open()) // 尝试打开数据库
{
    // 打开数据库失败，显示错误信息
    QMessageBox::critical(nullptr, "错误", db.lastError().text());
}
else
{
    // 3、定义查询对象，执行数据库操作

    QSqlQuery query; // 定义数据库查询对象
    // query.exec("DROP TABLE songlist;");
    // query.exec("DROP TABLE songhistory;");
 QString qstl; // 创建歌曲列表表格的SQL语句
 int ret;

//  // 创建歌曲记录表格
 qstl = "create table if not exists songhistory(id integer primary key autoincrement, fileName text , musicPath text);
 ret = query.exec(qstl);
 if (!ret)
 {
    QMessageBox::critical(nullptr, "create table songhistory", db.lastError().text());
 }

// // 查询歌曲历史记录表中的数据并显示
 qstl = "select * from songhistory";
 QStringList musicList;
 if (!query.exec(qstl)) // 执行查询操作
 {
    // 查询失败，显示错误信息
    QMessageBox::critical(nullptr, "错误", db.lastError().text());
 }
 while (query.next()) // 遍历查询结果
 {
    QString fileName = query.value(1).toString();
    QString musicPath = query.value(2).toString();
    musicList.append(fileName);

      playList.append(QUrl::fromLocalFile(musicPath));
      randomList.append(playList.size()-1);
      if(randomList.size()!=1)
      {
          int randomId=rand()%(randomList.size()-1);
          int tmp=randomList[randomId];
          randomList[randomId]=randomList[randomList.size()-1];
          randomList[randomList.size()-1]=tmp;
      }
   }
   ui->listWidget->addItems(musicList);
   if(playList.size()!=0)
      ui->listWidget->setCurrentRow(0);
}
```

首先检查是否存在名为“Songs”的数据库连接。如果存在，则获取该数据库连接，否则穿件一个新的 SQLite 数据库连接。

然后尝试打开数据库连接，如果失败则显示错误信息。

如果数据库打开成功，定义一个查询对象‘QSqlQuery’，并使用 SQL 语句创建存储音乐历史记录的表格‘songhistory’。

最后查询表‘songhistory’中的所有记录，并将结果加载到播放器的播放列表中。

2．Widget 类的槽函数

（1）添加音乐文件（on_pushButton_clicked 函数）：

```
void Widget::on_pushButton_clicked()
{
    QSqlQuery query;
    QString sql;

    QStringList musicList;
    QStringList musicPathList = QFileDialog::getOpenFileNames(this, tr("选择音频文件"), tr("F:\\music"), tr("Music(*.mp3 *.wav)"));
    for (int i = 0; i < musicPathList.size(); i++)
    {
        QString musicPath = musicPathList[i];
        QFileInfo file = QFileInfo(musicPath);
        QString fileName = file.fileName();
        musicList.append(fileName);
        qInfo()<<musicPath;
        qInfo()<<fileName;

        sql = QString("select musicPath from songhistory where musicPath = '%1';").arg(musicPath);
        if(!query.exec(sql))
        {
            QMessageBox::critical(nullptr,"select hash from songhistory where musicPath =",db.lastError().text());
        }
        if(query.next()==NULL)
        {
            ui->listWidget->addItem(fileName);
            sql = QString("insert into songhistory values(NULL,'%1','%2')").arg(fileName).arg(musicPath);
```

```
        if(query.next()==NULL)
        {
            ui->listWidget->addItem(fileName);
            sql = QString("insert into songhistory values(NULL,'%1','%2')").arg(fileName).arg(musicPath);
            if(!query.exec(sql))
            {
                QMessageBox::critical(nullptr,"insert error",db.lastError().text());
            }
            playList.append(QUrl::fromLocalFile(musicPath));
            randomList.append(playList.size()-1);
            if(randomList.size()!=1)
            {
                int randomId=rand()%(randomList.size()-1);
                int tmp=randomList[randomId];
                randomList[randomId]=randomList[randomList.size()-1];
                randomList[randomList.size()-1]=tmp;
            }
        }
    }
```

首先使用‘QFileDialog：：getOpenFileNames’打开文件对话框，允许用户选择一个或多个音频文件。并通过一个‘for’循环遍历用户选择的文件路径列表，逐个处理每个文件，使用 SQL 查询语句检查文件是否已经存在于数据库中的‘songhistory’表中，如果文件不存在于数据库中，则将文件信息插入‘songhistory’表中。最后将新添加的文件路径添加到播放列表和随机播放列表中，并随机打乱随机播放列表中的文件顺序。

（2）播放/暂停音乐（on_pushButton_4_clicked 函数）：

```
void Widget::on_pushButton_4_clicked()
{
    if(playList.empty())
    {
        return;
    }

    switch(mediaPlayer->playbackState())
    {
    case QMediaPlayer::PlaybackState::StoppedState: //停止状态
    {
        // 播放当前选中的音乐
        //1、获取选中的行号
        curPlayIndex = ui->listWidget->currentRow();
        //2、播放对应下标的音乐
        mediaPlayer->setSource(playList[curPlayIndex]);
        mediaPlayer->play();
        this->ui->pushButton_4->setStyleSheet("icon:url(:/Play.png)");
        break;
    }
    case QMediaPlayer::PlaybackState::PlayingState://播放状态
    {
        // 如果现在正在播放，暂停播放
        mediaPlayer->pause();
        this->ui->pushButton_4->setStyleSheet("icon:url(:/Pause.png)");
        break;
    }
    case QMediaPlayer::PlaybackState::PausedState://暂停状态
    {
        //如果现在是暂停的,继续播放音乐
        mediaPlayer->play();
        this->ui->pushButton_4->setStyleSheet("icon:url(:/Play.png)");
        break;
    }
    }
}
```

首先检查播放列表是否为空。如果为空，则直接返回，不进行任何操作。

然后使用‘QMediaPlayer::playbackState()’获取当前播放器的状态，并根据不同的状态执行不同的操作。如果播放器处于停止状态，则获取当前选中的音乐并开始播放；如果播放器处于播放状态，则暂停播放，并更新按钮的样式为暂停图标；如果播放器处于暂停状态，则继续播放，并更新按钮的样式为播放图标。

（3）上一曲/下一曲（on_pushButton_3_clicked 和 on_pushButton_5_clicked 函数）：

```
void Widget::on_pushButton_3_clicked()
{
    if(playList.empty())
    {
        return;
    }
    if(curPlayIndex == 0)
        curPlayIndex = playList.size() - 1;
    else
        curPlayIndex = (curPlayIndex - 1) % playList.size();
    this->ui->pushButton_4->setStyleSheet("icon:url(:/Play.png)");
    playMode-=3;
    ui->listWidget->setCurrentRow(curPlayIndex);
    mediaPlayer->setSource(playList[curPlayIndex]);
    mediaPlayer->play();
}
```

上一曲：

首先检查播放列表是否为空，若播放列表为空，则直接返回，不进行任何操作。

然后确定当前播放索引： 如果当前播放索引为 0，即正在播放列表中的第一首歌曲，那么将当前播放索引设置为播放列表的最后一首歌的索引，即 playList.size() - 1；否则，将当前播放索引减 1，以获取上一首歌曲的索引。

最后更新 UI 和播放状态。

下一曲：

```
//下一曲
void Widget::on_pushButton_5_clicked()
{
    if(playList.empty())
    {
        return;
    }
    // curPlayIndex++;
    // if(curPlayIndex >= playList.size())
    // {
    //     curPlayIndex = 0;
    // }
    curPlayIndex = (curPlayIndex + 1) % playList.size();
    playMode-=3;
    this->ui->pushButton_4->setStyleSheet("icon:url(:/Play.png)");
    ui->listWidget->setCurrentRow(curPlayIndex);
    mediaPlayer->setSource(playList[curPlayIndex]);
    mediaPlayer->play();
}
```

首先检查播放列表是否为空，如果播放列表为空则直接返回，不进行任何操作。

然后确定当前按播放索引。curPlayIndex = (curPlayIndex + 1) % playList.size()：将当前播放索引加 1，以获取下一首歌曲的索引。如果当前播放索引是最后一首歌曲的索引，取模运算将使其回到播放列表的第一首歌曲的索引。

最后更新 UI 和播放状态。

（4）双击列表播放音乐（on_listWidget_dxoubleClicked 函数）：

```
void Widget::on_listWidget_doubleClicked(const QModelIndex &index)
{
    this->ui->pushButton_4->setStyleSheet("icon:url(:/Play.png)");
    playMode-=3;
    curPlayIndex = index.row();
    ui->listWidget->setCurrentRow(curPlayIndex);
    mediaPlayer->setSource(playList[curPlayIndex]);
    mediaPlayer->play();
}
```

首先更新播放按钮的图标为播放状态。

然后用 playMode-=3 更新播放模式，使其进入播放状态。

接着用 curPlayIndex = index.row()获取双击项在列表中的行号（索引）；用 index.row() 返回用户双击的列表项的索引。

最后更新播放列表中的当前选中项以反映在 UI 上并设置播放器的播放源为当前选中项对应的歌曲，开始播放。

（5）播放模式切换（on_pushButton_2_clicked 和 on_pushButton_6_clicked 函数）：

```
void Widget::on_pushButton_2_clicked()
{
    if(playMode==1)
    {
        playMode=0;
        this->ui->pushButton_2->setStyleSheet("border:none;icon:url(:/Cycle.png)");
    } else
    {
        playMode=1;
        this->ui->pushButton_2->setStyleSheet("border:1px solid rgb(128,128,128);icon:url(:/Cycle.png)");
        this->ui->pushButton_6->setStyleSheet("border:none;icon:url(:/Random.png)");
    }

}
```

当“循环”按钮被点击时，检查 playMode 的值，若其值为 1，则其将被置为 0（关闭循环模式），并修改按钮的样式（去掉 1 像素宽的灰色实线边框）；若值不为 1，则其将被设置为 1，并为按钮增加一个灰色边框，同时去掉随机播放按钮的边框。

```
void Widget::on_pushButton_6_clicked()
{
    if(playMode==2)
    {
        playMode=0;
        this->ui->pushButton_6->setStyleSheet("border:none;icon:url(:/Random.png)");
    } else
    {
        playMode=2;
        this->ui->pushButton_6->setStyleSheet("border:1px solid rgb(128,128,128);icon:url(:/Random.png)");
        this->ui->pushButton_2->setStyleSheet("border:none;icon:url(:/Cycle.png)");
    }
}
```

当“随机播放”按钮被点击时，检查 playMode 的值，若为 2，则将其设置为 0（关闭随机播放），并去掉按钮的灰色边框；若值不为 2，则将其设置为 2，并为随机播放按钮添加边框，同时将循环按钮的边框去掉。

（6）音量控制（on_horizonalSlider_valueChanged 函数）：

```
void Widget::on_horizontalSlider_valueChanged(int value)
{
    audioOutput->setVolume((float)value/100+0.005);
    ui->label->setText(QString::number(value));
}
```

value 是从水平滑块获取的当前值，其范围是 0 到滑块的最大值。(float)value/100 将 value 转换为浮点数并除以 100，将其归一化到 0.0 到 1.0 的范围内（滑块的最大值为 100）。+0.005 是在归一化后的值上添加了一个小的偏移量，以确保音量永远不会降到 0。最后，这个计算得到的音量值被传递给 audioOutput 对象的 setVolume 方法，用于设置音频输出的音量。然后更新标签文本，使用户可以通过查看标签上的数字来了解当前滑块的位置（即音量大小）。

三、小组成员分工情况

UI 设计和美工：董淑琦

前端展现：吴家炫、鲁程喆

功能：吴家炫、鲁程喆

测试：吴家炫、鲁程喆、董淑琦

PPT 和实验报告：董淑琦

录屏：吴家炫

四、项目总结与反思

（一）总结

本项目是一个基于 Qt 的音乐播放器应用，支持基本的音乐播放功能，包括播放、暂停、上一曲、下一曲、双击播放列表播放音乐等。项目使用了 Qt 的多媒体模块(QMediaPlayer, QAudioOutput)来处理音频播放，并使用 SQLite 数据库来管理和存储音乐文件信息。

（二）反思

1. 优点

（1）模块化设计：项目各功能模块设计较为清晰，界面、播放、数据库管理等模块分工明确，便于后续维护和功能扩展。

（2）功能完备：实现了基本的音乐播放功能，包括播放、暂停、上一曲、下一曲、音量控制等，满足了一个音乐播放器的基本需求。

（3）用户体验：通过进度条、音量控制、播放列表等 UI 元素，提高了用户的使用体验。

2. 不足

（1）异常处理：对于数据库连接失败、文件加载错误等情况的异常处理可能还不够完善，可以增加更多的错误处理机制。

（2）歌曲列表功能不够完善：未处理歌曲的删除功能。

（3）播放模式优化：播放模式（顺序播放、循环播放、随机播放）的逻辑较为简单，可以考虑引入更多的播放模式，并优化现有模式的实现。

（4）代码优化：部分代码可能存在冗余和不规范的地方，可以进一步优化代码结构，提高代码可读性和可维护性。

总之，通过本项目的开发，我们对 Qt 框架的多媒体模块和数据库模块有了更深入的理解和应用。在实现音乐播放器的过程中，我们锻炼了对项目的整体规划和模块化设计能力，提升了异常处理和用户体验优化的意识。尽管项目存在一些不足和改进空间，但总体上实现了预期的功能目标，为未来的开发工作积累了宝贵的经验。