

目录

- 目录
 - 模态对话框与非模块对话框
 - 非模块对话框
 - 模块对话框
 - 创建方法一
 - 创建方法二
 - 窗口置顶
 - open方法和exec方法的区别
 - exec()方法
 - open()方法
 - 总结
 - 信号和槽
 - 不同的连接方式
 - 实现界面的切换
 - 实现父窗口切换子窗口，再切换回去
 - 连接信号
 - 子界面代码：
 - 父界面代码
 - 模态对话框消息传递
 - 模态对话框接收和拒绝消息
 - 主界面和登陆界面切换
 - QT几种标准对话框
 - 颜色对话框
 - 文件对话框
 - 输入对话框
 - 条目输入对话框
 - 整形输入对话框
 - 浮点数输入对话框
 - 文本输入对话框
 - 消息对话框
 - QT进度对话框
 - QT向导对话框
 - QVBoxLayout
 - QButtonGroup
 - QRadioButton 与 QCheckBox
 - QRadioButton单选框
 - QCheckBox复选框
 - 总结1
 - QT QLineEdit介绍
 - 实战
 - Qt布局与页面切换
 - QT主窗口
 - 菜单栏

- 通过Ui来实现
- 通过代码实现
- 工具栏
- 状态栏
- QTextEdit 文本编辑器
 - 文本块
 - 遍历文本块
 - 只遍历根框架
 - 直接遍历文本块
 - 设置文本块样式
 - 插入表格列表图片
 - 实现查找功能

模态对话框与非模态对话框

非模态对话框

- 它是一种不会阻止用户与程序中其他部分进行交互的对话框，它们不会阻止用户与程序的其他部分进行交互。用户可以同时打开多个非模态对话框，而不必等待一个对话框关闭才能操作其他部分

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

    QDialog w();
    w.show();
}
```

- 我们在MainWindow的构造函数里创建了QDialog类型的变量w。然后调用w的show函数展示该对话框。运行程序后会看到对话框w一闪而逝，然后创建了主窗口。一闪而逝的原因是w在MainWindow的构造函数里，构造函数结束w就被释放了。所以窗口就会一闪而逝。即便是如下设置w的父窗口为mainwindow也无济于事

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

    QDialog w(this);
    w.show();
}
```

- 因为无论w的父窗口是谁，都会随着MainWindow构造函数的结束而释放。那么好用的办法就是通过new创建对话框，这样对话框的空间在堆上，就不会随着构造函数结束而被释放了。

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

    QDialog w = new QDialog(this);
    w.show();
}
```

- 用new创建QDialog 对象w，并且指明了this(MainWindow)为其父窗口，这样在父窗口关闭或者释放后，其子窗口也会释放。这个原理在之后会讲给大家，QT提供了**对象树的机制**，**保证了父类被释放后子类都会被回收**。所以这也是我们指明w的父窗口为MainWindow的意思，如果不指明就需要手动回收w。不回收就会造成内存泄漏。

模块对话框

- **模态对话框**是一种**会阻止用户与程序中其他部分进行交互的对话框**。也就是说，当模态对话框显示时，用户必须在对话框关闭之前处理该对话框，无法转到程序的其他部分进行操作，可以像如下设置模块

创建方法一

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    auto w = new QDialog(this);
    // 为true就说明是模块对话框
    w->setModal(true);
    w->show();
}
```

- 点击运行，弹出一个对话框和主窗口，点击主窗口没有任何反应，点击对话框关闭后才能点击主窗口，所以w就是一个模态对话框

创建方法二

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
```

```
    ui->setupUi(this);  
    QDialog w(this);  
    w.exec();  
}
```

- 主要原因是exec这个函数会阻塞其他界面响应事件。所以直到我们关闭这个对话框后，exec才返回，这样MainWindow的构造函数才继续往下执行。也就是说只有关闭这个对话框才能执行后续内容

窗口置顶

- 有时我们需要将对话框置顶，不论其是不是模态对话框我们都可以这么做

```
MainWindow::MainWindow(QWidget *parent) :  
    QMainWindow(parent),  
    ui(new Ui::MainWindow)  
{  
    ui->setupUi(this);  
    auto s = new QDialog(this);  
    s->setWindowFlag(Qt::WindowStaysOnTopHint);  
    s->show();  
}
```

open方法和exec方法的区别

exec()方法

- exec()方法是一个阻塞调用，它会阻塞当前线程，直到用户关闭对话框。当对话框关闭时，exec()方法会返回用户的操作结果（例如QDialog::Accepted或QDialog::Rejected）。通常情况下，你会将exec()方法用于模态对话框，因为它会阻塞用户与应用程序的交互，直到用户完成对话框的操作。

```
Copy code  
int result = dialog.exec();  
if (result == QDialog::Accepted) {  
    // 用户点击了确定按钮  
} else {  
    // 用户点击了取消按钮或者关闭按钮  
}
```

open()方法

- open()方法是一个非阻塞调用，它会将对话框显示在界面上，但不会阻塞当前线程。当使用open()方法时，你通常需要手动检查对话框的结果或者在对话框的槽函数中处理用户的操作。你可以将open()方法用于非模态对话框，因为它不会阻塞用户与应用程序的交互。

```
Copy code  
dialog.open();
```

```
// 继续执行其他代码
```

总结

- 总的来说，`exec()`方法适用于模态对话框，它会阻塞当前线程直到用户关闭对话框，而`open()`方法适用于非模态对话框，它会将对话框显示在界面上但不会阻塞当前线程。选择使用哪种方法取决于你的应用程序逻辑和用户体验的需求。

信号和槽

- 当我们需要一个界面通知另一个界面时，可以采用信号和槽机制。通过链接信号和槽，当一个界面发送信号时，链接该信号的槽会被响应，从而达到消息传递的目的。
- 所以我们先创建一个Qapplication Widgets 应用。Creator会为我们生成mainwindow类和其界面。我们在界面添加一个按钮，按钮的名字叫showChildButton, 按钮显示的文字改为“显示子界面”。同时为该界面添加一个label，显示的文字修改为“这是主界面”



- 现在实现点击按钮，在控制台打印一条日志 “show child dialog ”
- 我们先在MainWindow的构造函数中添加信号和槽的链接逻辑

```
//信号的发送者，信号的事件，信号的接收者，执行的函数
connect(ui->showChildButton, SIGNAL(clicked(bool)), this,
        SLOT(showChildDialog()));
```

- connect函数的参数：**信号的发送者，信号的事件，信号的接收者，执行的槽函数
- 然后我们为MainWindow添加showChildDialog槽函数，槽函数需要用**slots**声明，我们这里在mainwindow.h里用**public slots**的方式声明槽函数。

```
public slots:
    void showChildDialog();
```

- **public slots** 声明一个为信号槽，用来接收信号并作出对应响应，也就是说connect的最后一个参数必须为槽函数
- 接下来去mainwindow.cpp中完成该函数的实现，可以在头文件中右键该函数，在弹出菜单里选择Refactor(重构), 再选择在mainwindow.cpp中添加实现。

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "childdialog.h"
QT_BEGIN_NAMESPACE
namespace Ui {
class MainWindow;
}
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

public slots:
    void showChildWindow();
    void showMain();
private:
    Ui::MainWindow *ui;
    ChildDialog * child_dialog;
};
#endif // MAINWINDOW_H

```

```

#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "childdialog.h"
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow), child_dialog(new ChildDialog(this))
{
    ui->setupUi(this);
    child_dialog = new ChildDialog(this);
    connect(ui->showChildButton, &QPushButton::clicked, this,
    &MainWindow::showChildWindow);
    connect(child_dialog, &ChildDialog::showMainsig, this, &MainWindow::showMain);
}

```

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

void MainWindow::showChildWindow()
{
    this->hide();
    child_dialog->show();
}

void MainWindow::showMain(){
    this->show();
}
```

不同的连接方式

- 我们上边用来连接信号和槽的方式是qt4提供的方式，用SIGNAL和SLOT将信号和槽转化为字符串。但是这种方式会存在一定问题，Qt要求槽函数的参数不能超过信号定义的参数，比如我们用到的信号clicked(bool)参数就是bool，我们定义的槽函数showChildDialog()是不带参数的，可以连接成功，如果我们在连接的时候将showChildDialog的参数写为3个，也可以连接成功

```
//qt4 风格的Slot和Signal 只是宏转换，字符串定义不能检测编译错误
connect(ui->showChildButton, SIGNAL(clicked(bool)), this,
SLOT(showChildDialog(1,2,3)));
```

- 但是点击会没有反应，说明qt4 这种连接信号和槽的方式不做编译检查，只是将信号和槽函数转译成字符串。

所以我推荐使用qt5以上版本的连接方式

```
//推荐qt5 风格
connect(ui->showChildButton, &QPushButton::clicked, this,
&MainWindow::showChildDialog);
```

- 如果 showChildDialog 是 MainWindow 类的成员函数，那么使用 &MainWindow::showChildDialog 来表示该成员函数的指针。如果直接写 &showChildDialog，那么编译器会认为你要连接的是一个全局函数，而不是 MainWindow 类的成员函数，因此会导致编译错误。

实现界面的切换

- 主窗口转向子窗口
 - 方法一：直接new出一个子窗口对象

```
void MainWindow::showChildDialog()
{
    qDebug() << "show child dialog " << endl;
    auto _child_dialog = new ChildDialog(this);
    _child_dialog->show();
}
```

- 再次运行程序点击显示子界面的按钮，就会弹出子界面了。关闭子界面，再次点击主窗口的显示子窗口按钮，子窗口又显示出来。这么做有一个问题就是可能会重复创建子窗口，但是Qt的对象树机制会保证父窗口回收时才回收子窗口，所以关闭子窗口只是隐藏了。那么随着点击，久而久之窗口会越来越多。

2. 方法二：我们想到的一个避免重复创建的办法就是在MainWindow的构造函数里创建好子界面，在槽函数中只控制子界面的显示即可。但同时要注意在MainWindow的析构函数里回收子界面类对象。

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "childdialog.h"
QT_BEGIN_NAMESPACE
namespace Ui {
class MainWindow;
}
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

public slots:
    //接收信号并实现显示子窗口逻辑
    void showChildWindow();
    //接收信号并实现显示自己
    void showMain();
private:
    Ui::MainWindow *ui;
    //指向子窗口
    ChildDialog * child_dialog;
};
#endif // MAINWINDOW_H
```

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
```



```

#include "childdialog.h"
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow), child_dialog(new ChildDialog(this))
{
    ui->setupUi(this);
    child_dialog = new ChildDialog(this);
    // UI界面的按钮为发送者, 信号函数为点击事件, 作用自己, 信号槽函数
    connect(ui->showChildButton, &QPushButton::clicked, this,
    &MainWindow::showChildWindow);
    // 发送者为子窗口, 发送的信号函数是showMainsig, 作用于自己, 信号槽响应函数
    connect(child_dialog, &ChildDialog::showMainsig, this, &MainWindow::showMain);
}

MainWindow::~MainWindow()
{
    delete ui;
    if(_child_dialog){
        delete _child_dialog;
        _child_dialog = nullptr;
    }
}

void MainWindow::showChildWindow()
{
    //实现隐藏
    this->hide();
    child_dialog->show();
}

void MainWindow::showMain(){
    this->show();
}

```

实现父窗口切换子窗口, 再切换回去

- 父窗口切换子窗口 可以用槽函数 加connect连接来实现 代码如上
- 子窗口切换父窗口 可以声明一个信号, 用来通知主界面显示

```

class ChildDialog : public QDialog
{
    Q_OBJECT
signals:
    void showMainSig();
public:
    explicit ChildDialog(QWidget *parent = nullptr);
    ~ChildDialog();

private:
    Ui::ChildDialog *ui;

```

```
QWidget *_parent;
public slots:
    void showMainWindow();
};
```

- showMainSig是一个信号，用来通知主界面，所以主界面MainWindow类要连接这个信号，我们先在主界面类中声明这个函数

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

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

private:
    Ui::MainWindow *ui;
public slots:
    void showChildDialog();
    void showMainDialog();
private:
    ChildDialog *_child_dialog;
};
```

- showMainDialog 是新增的槽函数，用来连接ChildDialog的showMainSig信号。
- 我们修改ChildDialog的showMainWindow函数

```
void ChildDialog::showMainWindow()
{
    qDebug() << "show main window" << endl;
    this->hide();
    //可以再次发送信号通知主窗口显示
    emit showMainSig();
}
```

- 然后再mainwindow里实现逻辑

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

    //推荐qt5 风格
    connect(ui->showChildButton, &QPushButton::clicked, this,
```

```

&MainWindow::showChildDialog);

    connect(_child_dialog, &ChildDialog::showMainSig, this,
&MainWindow::showMainDialog);
}

void MainWindow::showChildDialog()
{
    qDebug() << "show child dialog " << endl;
    _child_dialog->show();
    this->hide();
}

```

连接信号

- 上面的程序还可以进一步优化，因为Qt提供了信号连接信号的方式，也就是说我们可以把子界面的按钮点击信号和showMainSig信号连接起来。

```

ChildDialog::ChildDialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::ChildDialog), _parent(parent)
{
    ui->setupUi(this);
    connect(ui->showMainWindow, &QPushButton::clicked, this,
&ChildDialog::showMainSig);
}

```

- 将clicked和showMainSig两个信号连接起来，也可以实现消息的传递，让代码更简洁了。

子界面代码：

1. 头文件

```

#ifndef CHILDDIALOG_H
#define CHILDDIALOG_H

#include <QDialog>

namespace Ui {
class ChildDialog;
}

class ChildDialog : public QDialog
{
    Q_OBJECT
signals:
    void showMainsig();
public:
    explicit ChildDialog(QWidget *parent = nullptr);

```

```

    ~ChildDialog();
public slots:
    void showMainWindow();

private:
    Ui::ChildDialog *ui;
    QWidget *_parent;
};

#endif // CHILDDIALOG_H

```

2. cpp

```

#include "childdialog.h"
#include "ui_childdialog.h"

ChildDialog::ChildDialog(QWidget *parent)
    : QDialog(parent)
    , ui(new Ui::ChildDialog), _parent(parent)
{
    ui->setupUi(this);
    //信号的发送者, 信号的事件, 信号的接收者, 执行的函数
    connect(ui->pushButton, &QPushButton::clicked, this,
    &ChildDialog::showMainWindow);
}

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

void ChildDialog::showMainWindow(){
    this->hide();
    emit showMainsig();
}

```

父界面代码

- 头文件

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "childdialog.h"
QT_BEGIN_NAMESPACE
namespace Ui {

```

```

class MainWindow;
}
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

public slots:
    void showChildWindow();
    void showMain();
private:
    Ui::MainWindow *ui;
    ChildDialog * child_dialog;
};
#endif // MAINWINDOW_H

```

- 资源文件

```

#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "childdialog.h"
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow), child_dialog(new ChildDialog(this))
{
    ui->setupUi(this);
    connect(ui->showChildButton, &QPushButton::clicked, this,
    &MainWindow::showChildWindow);
    connect(child_dialog, &ChildDialog::showMainsig, this, &MainWindow::show);
}

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

void MainWindow::showChildWindow()
{
    this->hide();
    child_dialog->show();
}

void MainWindow::showMain(){
    this->show();
}

```

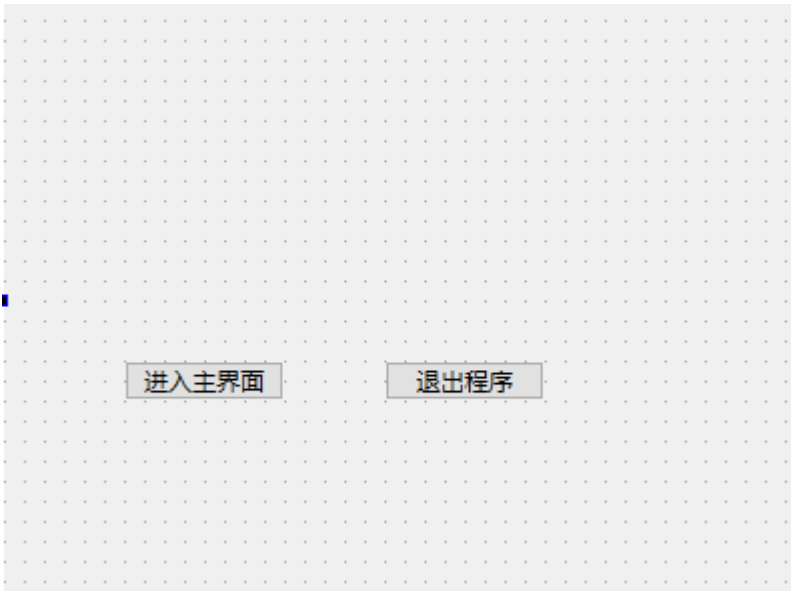
模态对话框消息传递

模态对话框接收和拒绝消息

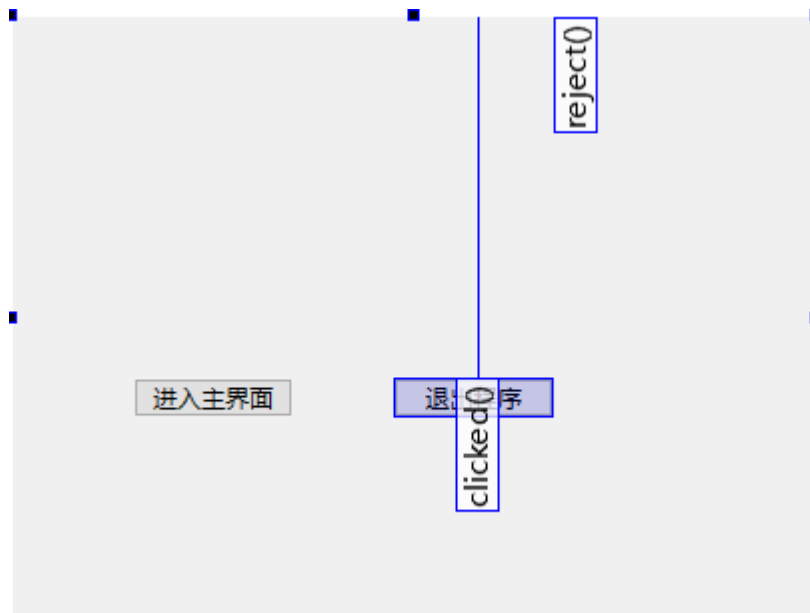
- 我们创建一个模态对话框，调用exec函数后可以根据其返回值进行不同的处理，exec的返回值有两种，Qt的官方文档记录的为

```
QDialog::Accepted  
QDialog::Rejected
```

- Accepted 表示接受消息，Rejected表示拒绝消息。
- 还是按照之前的操作，选择新建QT Application项目，然后创建类名为MainWindow, 基类选择QDialog, 点击创建生成Qt项目。然后我们添加设计师界面类，类名MyDialog, 基类选择QDialog。然后在这个mydialog.ui中添加两个按钮，一个是进入主界面，一个是退出程序。



- 在设计师界面点击Edit Signal/Slots 信号槽的按钮，进入信号槽编辑界面，鼠标按住退出程序按钮不松开拖动，将另一端连接到MyDialog对话框，QPushbutton这边信号选择clicked(), MyDialog信号选择reject, 这样就将两个信号连接起来了，我们点击退出程序按钮，会触发MyDialog发送reject信号，因为MyDialog调用exec后等待信号返回，此时收到reject信号，exec就会返回Rejected值。同样道理，MyDialog发送acceptp()信号后，exec返回值为Accepted。



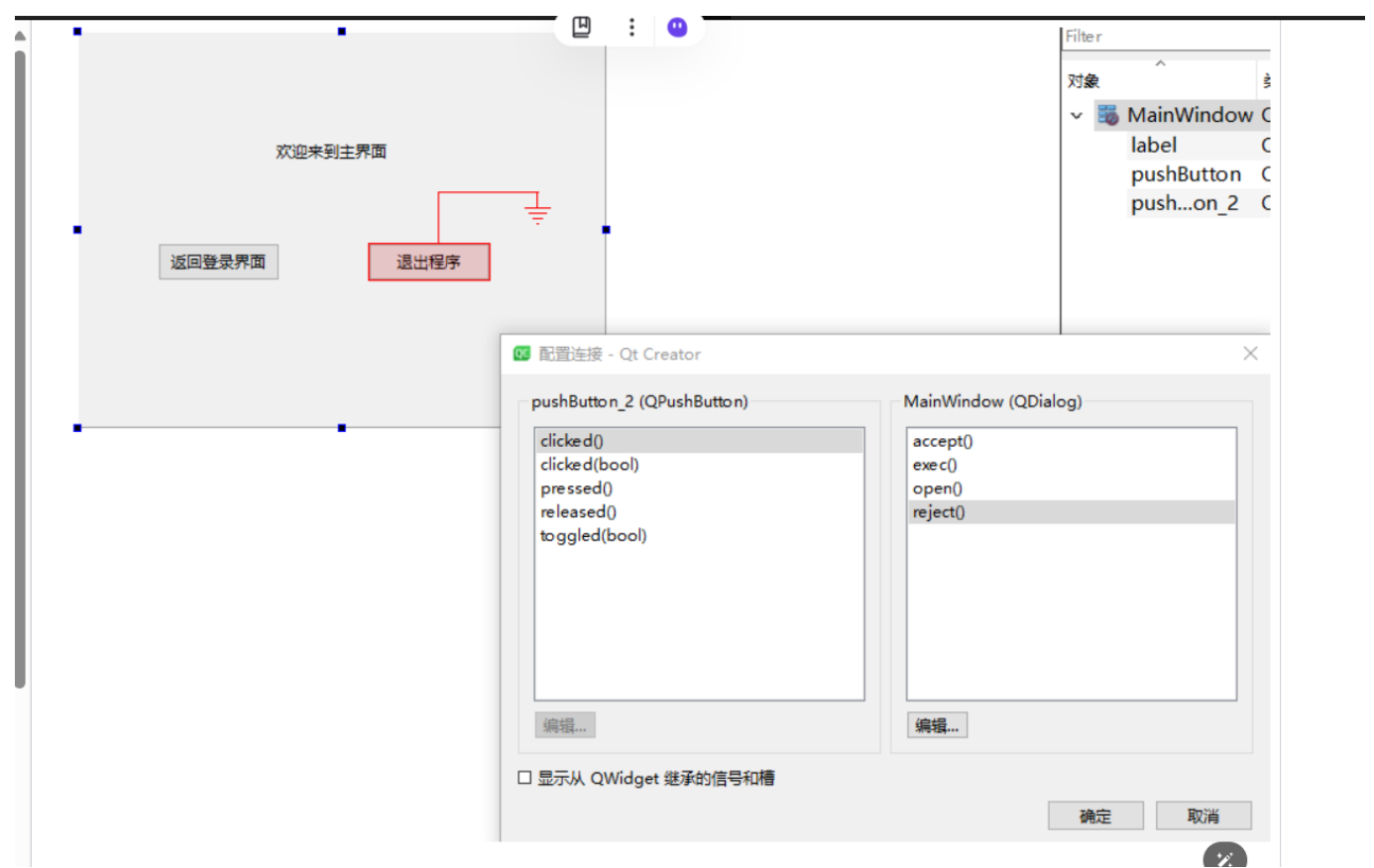
- 我们点击Edit Widget 按钮，然后右键点击键入主界面按钮，点击进入槽，在Qt 为我们生成的槽函数里添加acceptp()信号发送逻辑
- 他会生成一个槽函数，并且在内部实现了点击响应执行这个槽函数的代码

```
void MyDialog::on_pushButton_clicked()
{
    accept();
}
```

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    MyDialog dialog;
    if(dialog.exec() == QDialog::Accepted){
        w.show();
        return a.exec();
    }else{
        return 0;
    }
}
```

主界面和登陆界面切换

- 现在我们要实现主界面和登录界面的互相切换，我们刚才创建的对话框界面MyDialog界面是登录界面，主界面是MainWindow类。我们在mainwindow.ui添加两个按钮，返回登录按钮和退出程序按钮，再添加一个label表示这是主界面。然后点击Edit Signal/Slot 进入信号编辑界面，点击退出程序按钮不松开拖动到尾部连接MainWindow的主界面上，选择退出程序的clicked()信号，MainWindow选择reject信号,将两个信号连接起来。



- 然后右击登录按钮转到槽，在槽函数里添加这个逻辑

```
void MainWindow::on_pushButton_clicked()
{
    close();
    MyDialog mydialog;
    if(mydialog.exec() == QDialog::Accepted ){
        this->show();
    }else{
        return;
    }
}
```

- 点击返回登陆按钮就会close主窗口，但是并不是真的关闭，只是将主窗口隐藏，Qt回收机制是所有窗口都关闭后才回收。
- 再次运行程序，点击进入主界面按钮就可以进入主界面，点击返回登录按钮就可以返回登录界面，可以实现两个界面的切换了。

QT几种标准对话框

颜色对话框

- 颜色对话框用来选择颜色，创建后会显示各种颜色和透明度信息

```
void MainWindow::on_pushButton_clicked()
{
```



```
// QColorDialog colorDlg(Qt::blue, this);
// colorDlg.setOption(QColorDialog::ShowAlphaChannel);
// colorDlg.exec();
// QColor color = colorDlg.currentColor();
// qDebug() << "color is " << color;

    QColor color = QColorDialog::getColor(Qt::blue, this, tr("选择颜色"),
    QColorDialog::ShowAlphaChannel );
    qDebug() << "color is " << color;
}
```

- `setOption()`方法用于设置颜色对话框的选项，以控制对话框的行为和外观。通过这个方法，你可以根据自己的需求对对话框进行定制，例如设置初始颜色、显示模式、自定义按钮等。
- `QColorDialog::ColorDialogOption`:
- `QColorDialog::ShowAlphaChannel`: 设置是否显示颜色对话框中的Alpha通道（透明度），默认是不显示。
- `QColorDialog::NoButtons`: 设置是否隐藏颜色对话框中的按钮（确定和取消按钮），默认是显示。
- `QColorDialog::DontUseNativeDialog`: 设置是否使用平台本地的颜色对话框，而不是Qt内置的对话框。

文件对话框

- 它允许用户浏览文件系统并选择文件或目录，以便在应用程序中进行处理。文件对话框提供了多种选项和模式，包括：
 1. 打开文件对话框：允许用户选择一个或多个文件以供读取或处理。也就是**`getOpenFileName()`**：打开一个打开文件对话框，允许用户选择一个或多个文件以供读取或处理。
 2. 保存文件对话框：允许用户指定文件名和位置，以便保存文件。**`getSaveFileName()`**：打开一个保存文件对话框，允许用户指定文件名和位置以供保存文件。
 3. 选择文件夹对话框：允许用户选择一个文件夹或目录。**`getExistingDirectory()`**：打开一个选择文件夹对话框，允许用户选择一个文件夹或目录。
- 这些函数通常需要指定一些参数，包括父窗口、对话框的标题、初始路径、过滤器等，以及返回选择的文件路径或目录路径。
- 以下是简单示例

```
#include <QApplication>
#include <QFileDialog>
#include <QFile>
#include <QTextStream>
#include <QDebug>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    // 打开文件对话框，选择要打开的文件
    QString filePath = QFileDialog::getOpenFileName(nullptr, "选择文件", "", "文本
```

```
文件 (*.txt)");

    if (!filePath.isEmpty()) {
        // 如果用户选择了文件，则打开文件并读取内容
        QFile file(filePath);
        if (file.open(QIODevice::ReadOnly | QIODevice::Text)) {
            QTextStream in(&file);
            QString content = in.readAll();
            qDebug() << "文件内容: " << content;
            file.close();
        } else {
            qDebug() << "无法打开文件: " << file.errorString();
        }
    } else {
        qDebug() << "用户取消了选择文件";
    }

    return app.exec();
}
```

- 常常配合QFile 来对文件进行一系列操作

```
void MainWindow::on_pushButton_2_clicked()
{
    QString path = QDir::currentPath();
    QString title = tr("文件对话框");
    QString filter = tr("文本文件 (*.txt);;图片文件 (*.jpg *.gif *.png);;所有文件 (*.*)");
    QString aFileName=QFileDialog::getOpenFileName(this,title,path,filter);

    qDebug() << aFileName << endl;
}
```

输入对话框

- 输入对话框分几种，包括文本(字符串)输入对话框，整数输入对话框，浮点数输入对话框，条目输入对话框。

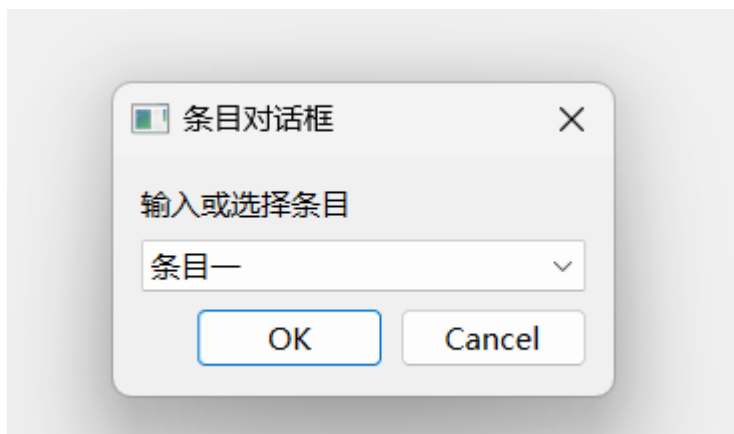
条目输入对话框

```
QString QInputDialog::getItem(QWidget *parent, const QString &title, const QString &label, const QStringList &items, int current = 0, bool editable = false, bool *ok = nullptr, Qt::WindowFlags flags = Qt::WindowFlags());
```

- 参数说明：
 1. parent: 指定对话框的父窗口，即在哪个窗口中显示对话框。

2. title: 指定对话框的标题。
3. label: 指定对话框中显示的文本标签, 用于提示用户输入或选择条目。
4. items: 一个字符串列表, 包含了用户可以选择的条目。
5. current: 指定默认选中的条目的索引。默认为0, 即默认选中列表中的第一个条目。
6. editable: 一个布尔值, 指示是否允许用户输入自定义的条目。如果设置为 true, 则在对话框中显示一个可编辑的文本框, 用户可以输入自己的条目; 如果设置为 false, 则只允许用户选择预定义的条目, 不允许输入自定义的条目。
7. ok: 一个指向布尔变量的指针, 用于接收用户是否点击了对话框的确认按钮。如果用户点击了确认按钮, 则 ok 为 true; 如果用户点击了取消按钮, 则 ok 为 false。这个参数是可选的。
8. flags: 指定对话框的标志, 例如 Qt::WindowStaysOnTopHint 表示对话框始终位于最顶层, Qt::WindowCloseButtonHint 表示显示关闭按钮等。这个参数也是可选的

```
QStringList items;  
items << tr("条目一") << tr("条目二");  
bool ok = false;  
auto itemData = QInputDialog::getItem(this, tr("条目对话框"), tr("输入或选择条  
目"), items, 0, false, &ok);  
if(ok){  
    qDebug() << "item is " << itemData;  
}
```



整形输入对话框

```
int QInputDialog::getInt(QWidget *parent, const QString &title, const QString  
&label, int value = 0, int minValue = -2147483647, int maxValue = 2147483647, int  
step = 1, bool *ok = nullptr, Qt::WindowFlags flags = Qt::WindowFlags());
```

- 参数说明:
 1. parent: 指定整形输入对话框的父窗口, 即在哪个窗口中显示对话框。
 2. title: 指定对话框的标题。
 3. label: 指定对话框中显示的文本标签, 用于提示用户输入。
 4. value: 指定整形输入框的初始值, 默认为0。
 5. minValue: 指定可输入的最小值, 默认为-2147483647。

6. `maxValue`: 指定可输入的最大值, 默认为2147483647。
7. `step`: 指定增加或减少的步长, 默认为1。
8. `ok`: 一个bool类型的指针, 用于指示用户是否点击了对话框中的确认按钮。如果用户点击了确认按钮, 则`ok`为true; 如果用户点击了取消按钮, 则`ok`为false。这个参数是可选的。
9. `flags`: 指定对话框的标志, 例如`Qt::WindowStaysOnTopHint`表示对话框始终位于最顶层, `Qt::WindowCloseButtonHint`表示显示关闭按钮等。

```
void MainWindow::on_pushButton_4_clicked()
{
    bool ok = false;
    auto intdata = QInputDialog::getInt(this, tr("数字输入对话框"), tr("请输入数字"), 200, -200, 400, 10, &ok);
    if(ok){
        qDebug() << intdata << endl;
    }
}
```

浮点数输入对话框

```
bool QInputDialog::getDouble(QWidget *parent, const QString &title, const QString &label, double value = 0, double minValue = -2147483647, double maxValue = 2147483647, int decimals = 1, bool *ok = nullptr, Qt::WindowFlags flags = Qt::WindowFlags());
```

- 与其他输入对话框不同的是 他的`decimals` 那位并不是表示步长, 而是表示浮点数的小数位数

文本输入对话框

```
QString QInputDialog::getText(QWidget *parent, const QString &title, const QString &label, QLineEdit::EchoMode echo = QLineEdit::Normal, const QString &text = QString(), bool *ok = nullptr, Qt::WindowFlags flags = Qt::WindowFlags());
```

- 参数说明:
 1. `parent`: 指定对话框的父窗口, 即在哪个窗口中显示对话框。
 2. `title`: 指定对话框的标题。
 3. `label`: 指定对话框中显示的文本标签, 用于提示用户输入。
 4. `echo`: 指定输入框的回显模式, 默认为 `QLineEdit::Normal`, 即显示输入的文本。
 5. `text`: 指定输入框的初始文本, 默认为空字符串。
 6. `ok`: 一个 bool 类型的指针, 用于指示用户是否点击了对话框中的确认按钮。如果用户点击了确认按钮, 则 `ok` 为 true; 如果用户点击了取消按钮, 则 `ok` 为 false。这个参数是可选的。
 7. `flags`: 指定对话框的标志, 例如 `Qt::WindowStaysOnTopHint` 表示对话框始终位于最顶层, `Qt::WindowCloseButtonHint` 表示显示关闭按钮等。

- echo的多种常见模式：
 1. QLineEdit::Normal: 默认模式, 显示用户输入的文本。这是最常用的回显模式
 2. QLineEdit::NoEcho: 不显示任何输入, 通常用于隐藏输入内容, 例如用于密码输入框。
 3. QLineEdit::Password: 显示密码字符 (通常是圆点或星号) 来隐藏用户输入的文本。这个模式常用于密码输入框, 以保护用户的隐私安全。
 4. QLineEdit::PasswordEchoOnEdit: 当编辑文本时显示密码字符, 否则显示原始文本。这种模式在用户输入密码时会显示圆点或星号, 但在编辑密码时会显示实际的文本。

```
void MainWindow::on_pushButton_3_clicked()
{
    bool ok = false;
    auto text = QInputDialog::getText(this, tr("文字输入对话框"), tr("请输入用户的姓名"), QLineEdit::Normal, tr("admin"), &ok);
    if(ok){
        qDebug() << text << endl;
    }
}
```

消息对话框

- 消息提示对话框用于向用户显示简单的信息、警告或错误消息, 通常是以模态 (阻塞) 的方式显示。Qt 提供了 QMessageBox 类来创建消息提示对话框, 并支持不同的消息类型, 包括信息、警告、错误、询问等
- 这里的 parent 参数指定了消息提示框的父窗口, 即在哪个窗口中显示对话框。title 参数指定了对话框的标题, text 参数指定了对话框中显示的文本内容。
- QMessageBox 还提供了更多的参数, 以支持更丰富的功能。例如, 你可以使用
- QMessageBox::Yes 和 QMessageBox::No 指定对话框中的按钮
- 或者使用 QMessageBox::Ok 指定只显示一个确定按钮。
- 你也可以使用 QMessageBox::setDetailedText() 方法设置更详细的文本信息。

```
void MainWindow::on_pushButton_clicked()
{
    auto ret = QMessageBox::question(this, tr("提问对话框"), tr("你是单身吗"), QMessageBox::Yes, QMessageBox::No);
    if(ret == QMessageBox::Yes || ret == QMessageBox::No){
        qDebug() << "ret is " << ret << Qt::endl;
    }

    auto ret2 = QMessageBox::information(this, tr("通知对话框"), tr("你好单身狗"), QMessageBox::Ok);
    if(ret2 == QMessageBox::Ok){
        qDebug() << "ret2 is " << ret2 << Qt::endl;
    }
}
```

```

    auto ret3 = QMessageBox::warning(this, tr("警告对话框"), tr("你最好找个地方发泄一下"), QMessageBox::Ok);
    if (ret3 == QMessageBox::Ok) {
        qDebug() << "ret3 is " << ret3 << Qt::endl;
    }

    auto ret4 = QMessageBox::critical(this, tr("关键提示对话框"), tr("我梦寐以求是真爱和自由"), QMessageBox::Ok);
    if (ret4 == QMessageBox::Ok) {
        qDebug() << "ret4 is " << ret4 << Qt::endl;
    }
}

```

QT进度对话框

- 用于显示任务的进度情况。进度对话框通常用于长时间运行的任务，以便向用户显示任务的完成进度，并让用户了解任务是否正在进行中。
- 你可以设置进度对话框的标题、文本、进度条的范围和当前值等属性，以及显示取消按钮以允许用户取消任务。
- 最基本的用法

```

#include <QProgressDialog>

QProgressDialog progressDialog("任务进行中...", "取消", 0, 100, parent);
progressDialog.setWindowTitle("进度对话框");
progressDialog.setWindowModality(Qt::WindowModal);

for (int i = 0; i < 100; ++i) {
    // 执行任务的代码

    // 更新进度对话框的进度值
    progressDialog.setValue(i);

    // 检查用户是否点击了取消按钮
    if (progressDialog.wasCanceled()) {
        // 用户点击了取消按钮，执行相应的操作
        break;
    }
}

```

- 我们可以通过定时器的方式定时更新进度，这样就可以更清楚的看到进度条对话框更新情况了

```

void MainWindow::on_pushButton_2_clicked()
{
    // QProgressDialog progressdialog(tr("正在复制"), tr("取消复制"), 0, 5000, this);
    // progressdialog.setWindowTitle(tr("文件复制进度对话框"));
    // progressdialog.setWindowModality(Qt::ApplicationModal);
}

```

```

        // progressdialog.show();
        // for(int i = 0; i < 5000; i++){
        //     progressdialog.setValue(i);
        //     QApplication::processEvents();
        //     if(progressdialog.wasCanceled()) break;
        // }
        // progressdialog.setValue(5000);
        progressdialog = new QProgressDialog(tr("正在复制"), tr("取消复制"), 0, 5000,
        this);
        progressdialog->setWindowTitle(tr("文件复制进度对话框"));
        progressdialog->setWindowModality(Qt::ApplicationModal);
        _timer = new QTimer(this);
        _count = 0;
        connect(_timer, &QTimer::timeout, this, &MainWindow::on_updateProgressDialog);
        connect(progressdialog, &QProgressDialog::canceled, this,
        &MainWindow::out_cancelProgressDialog);
        _timer->start(1);
    }

    void MainWindow::on_updateProgressDialog()
    {
        _count ++;
        if(_count > 5000){
            delete _timer;
            _timer = nullptr;
            delete progressdialog;
            progressdialog = nullptr;
            _count = 0;
            return;
        }
        progressdialog->setValue(_count);
    }

    void MainWindow::out_cancelProgressDialog()
    {
        delete _timer;
        _timer = nullptr;
        delete progressdialog;
        progressdialog = nullptr;
        _count = 0;
    }
}

```

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QProgressDialog>
#include <QTimer>
QT_BEGIN_NAMESPACE
namespace Ui {

```

```
class MainWindow;
}
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

private slots:
    void on_colorBtn_clicked();

    void on_textBtn_clicked();

    void on_intBtn_clicked();

    void on_floatBtn_clicked();

    void on_listBtn_clicked();

    void on_pushButton_clicked();

    void on_inputBtn_clicked();

    void on_pushButton_2_clicked();
    //以下才是进度对话框真正用到的
    void on_updateProgressDialog();

    void out_cancelProgressDialog();
private:
    Ui::MainWindow *ui;
    QProgressDialog *progressdialog;
    QTimer* _timer;
    int _count;
};
```

QT向导对话框

- 所需头文件

```
#include <QWizard>
#include <QVBoxLayout>
#include <QLabel>
#include <QButtonGroup>
#include <QRadioButton>
```



```
void MainWindow::on_pushButton_10_clicked()
{
    QWizard wizard(this);
    wizard.setWindowTitle(tr("全城热恋"));
    QWizardPage* page1 = new QWizardPage();
    page1->setTitle(tr("婚恋介绍引导程序"));
    auto label1 = new QLabel();
    label1->setText(tr("该程序帮助您找到人生伴侣"));
    QVBoxLayout *layout = new QVBoxLayout();
    layout->addWidget(label1);
    page1->setLayout(layout);
    wizard.addPage(page1);
    QWizardPage* page2 = new QWizardPage();
    page2->setTitle("选择心动类型");

    QButtonGroup *group = new QButtonGroup(page2);
    QRadioButton * btn1 = new QRadioButton();
    btn1->setText("白富美");
    group->addButton(btn1);
    QRadioButton * btn2 = new QRadioButton();
    btn2->setText("萝莉");
    group->addButton(btn2);
    QRadioButton * btn3 = new QRadioButton();
    btn3->setText("御姐");
    group->addButton(btn3);
    QRadioButton * btn4 = new QRadioButton();
    btn4->setText("小家碧玉");
    group->addButton(btn4);
    QRadioButton * btn5 = new QRadioButton();
    btn5->setText("女汉子");
    group->addButton(btn5);

    QRadioButton * btn6 = new QRadioButton();
    btn6->setText("成年人不做选择，全选!");
    group->addButton(btn6);
    QVBoxLayout *vboxLayout2 = new QVBoxLayout();
    for(int i = 0; i < group->buttons().size(); i++){
        vboxLayout2->addWidget(group->buttons()[i]);
    }

    page2->setLayout(vboxLayout2);
    wizard.addPage(page2);

    QWizardPage* page3 = new QWizardPage();
    page3->setTitle(tr("你的缘分即将到来"));
    auto label3 = new QLabel();
    label3->setText(tr("感谢您的参与，接下来的一个月会遇到对的人"));
    QVBoxLayout *layout3 = new QVBoxLayout();
    layout3->addWidget(label3);
    page3->setLayout(layout3);
    wizard.addPage(page3);
    wizard.show();
}
```

```
wizard.exec();  
}
```

- 我们来讲讲以上代码中用到的类以及方法

QVBoxLayout

- 这是Qt 中的一个布局管理器，用于在垂直方向上排列子部件，它可以将子部件按照垂直方向依次排列，使它们在垂直方向上均匀分布。QVBoxLayout 通常与 QWidget 或其子类一起使用，用于管理窗口或部件中的子部件的布局。
- 使用时的基本案例

```
#include <QVBoxLayout>  
#include <QLabel>  
#include <QPushButton>  
#include <QWidget>  
  
// 创建父部件  
QWidget *parentWidget = new QWidget();  
  
// 创建 QVBoxLayout 布局管理器  
QVBoxLayout *layout = new QVBoxLayout();  
  
// 创建子部件  
QLabel *label1 = new QLabel("Label 1");  
QPushButton *button1 = new QPushButton("Button 1");  
QPushButton *button2 = new QPushButton("Button 2");  
  
// 将子部件添加到 QVBoxLayout 中  
layout->addWidget(label1);  
layout->addWidget(button1);  
layout->addWidget(button2);  
  
// 将 QVBoxLayout 设置为父部件的布局管理器  
parentWidget->setLayout(layout);  
  
// 显示父部件  
parentWidget->show();
```

QButtonGroup

- 这是 Qt 中的一个类，用于管理一组按钮。它可以将多个单选按钮或复选按钮组织在一起，形成一个逻辑上的单元，使它们能够相互排斥或者互斥。
- 使用 QButtonGroup 可以方便地管理一组按钮的状态和行为，特别是在需要对按钮进行组合选择或互斥选择时。例如，你可以将多个单选按钮添加到一个 QButtonGroup 中，使用户只能选择其中的一个按钮，而取消选择其他按钮；或者将多个复选按钮添加到一个 QButtonGroup 中，使它们在同一时间只能有一个或多个被选中。

- 以下未使用案例

```
#include <QButtonGroup>
#include <QRadioButton>

// 创建一个 QButtonGroup 对象
QButtonGroup *buttonGroup = new QButtonGroup(parent);

// 创建单选按钮并添加到按钮组中
QRadioButton *radioButton1 = new QRadioButton("Option 1");
QRadioButton *radioButton2 = new QRadioButton("Option 2");
QRadioButton *radioButton3 = new QRadioButton("Option 3");

buttonGroup->addButton(radioButton1);
buttonGroup->addButton(radioButton2);
buttonGroup->addButton(radioButton3);

// 设置按钮组中的默认选中按钮
radioButton1->setChecked(true);

// 监听按钮组中按钮的选择状态变化
connect(buttonGroup, SIGNAL(buttonClicked(QAbstractButton*)), this,
        SLOT(onButtonClicked(QAbstractButton*)));
```

QRadioButton 与 QCheckBox

QRadioButton 单选框

- 这是 Qt 中的一个类，用于创建单选按钮。单选按钮通常用于让用户在一组互斥的选项选择一个选项。
- 与普通的按钮类似，QRadioButton 也是 QAbstractButton 的子类，因此它继承了按钮的所有功能和特性。单选按钮的特点是它们可以成组出现，并且在同一组中只能选择一个按钮，当选择一个按钮时，其他按钮会自动取消选择。

QCheckBox 复选框

- 是 Qt 中的一个类，用于创建复选框。与单选按钮类似，复选框也是一种用户界面控件，但是它允许用户同时选择多个选项。

总结1

- new出来的需要手动释放，除非他被加载到了父对象上，只要他被加载到了一个界面上，就不需要手动释放了，只需要手动释放那个界面就行

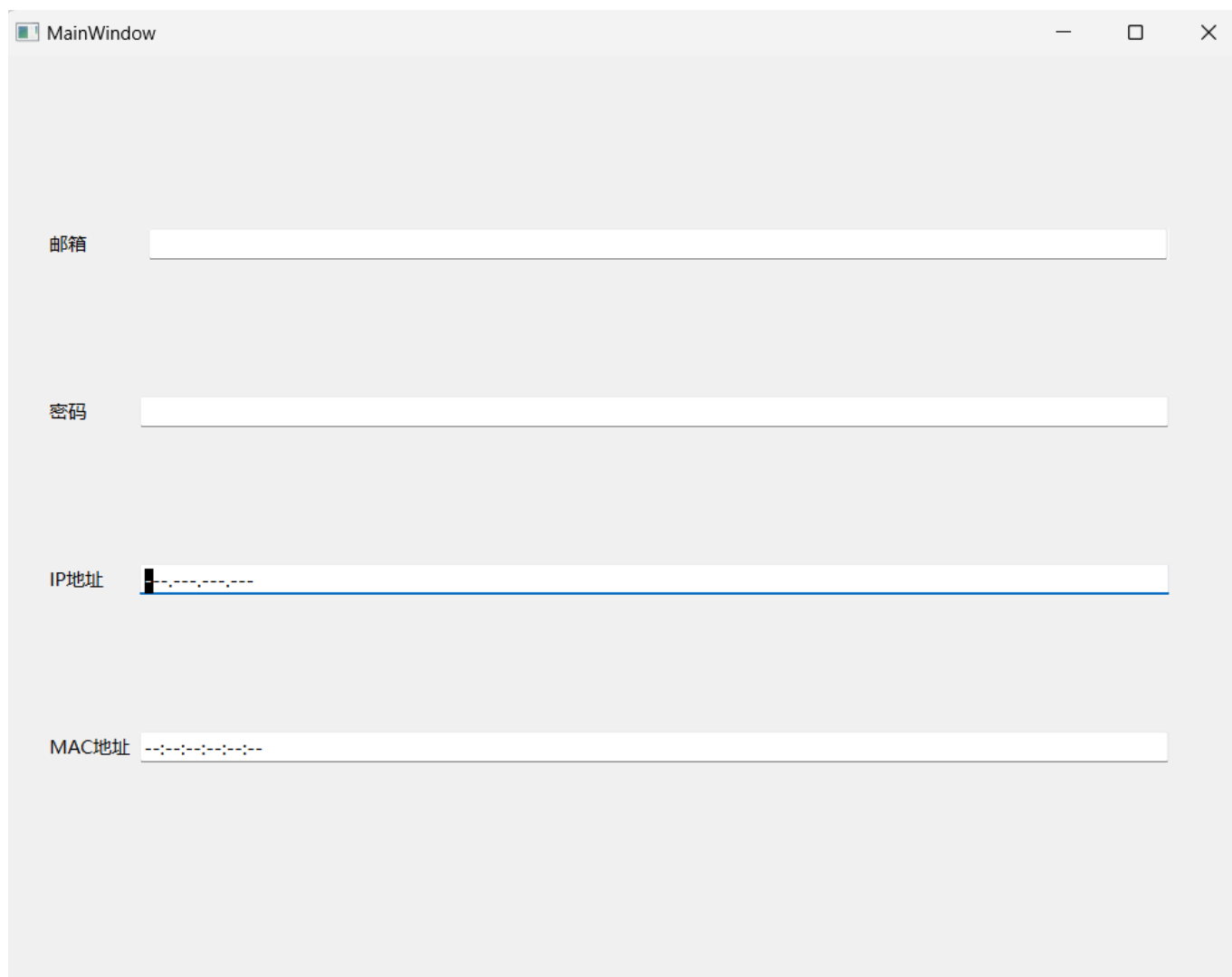
QT QLineEdit介绍

- QLineEdit属于输入插件，用来实现单行录入。支持几种录入模式。
- Normal表示正常录入,录入的信息会显示在QLineEdit上。

- Password表示密码录入的方式，录入的信息不显示QLineEdit，只是通过黑色圆点显示。
- NoEcho 表示不显示录入信息，类似于Linux输入密码时，显示的是一片空白。
- PasswordEchoOnEdit 表示在输入的一刹那可以看到字符，但是立刻变为不可见的黑色圆点显示。

实战

- 做一个如下页面



The image shows a Qt application window titled "MainWindow". Inside the window, there is a form with four input fields, each with a label to its left:

- The first field is labeled "邮箱" (Email) and is an empty QLineEdit.
- The second field is labeled "密码" (Password) and is an empty QLineEdit.
- The third field is labeled "IP地址" (IP Address) and contains the text "---.---.---.---".
- The fourth field is labeled "MAC地址" (MAC Address) and contains the text "--:--:--:--:--:--".

- QLineEdit 支持正则表达式，支持mask（定义一种输入规则）

The following table shows the mask and meta characters that can be used in an input mask.

Mask Character	Meaning
A	character of the Letter category required, such as A-Z, a-z
a	character of the Letter category permitted but not required.
N	character of the Letter or Number category required, such as A-Z, a-z, 0-9.
n	character of the Letter or Number category permitted but not required.
X	Any non-blank character required.
x	Any non-blank character permitted but not required.
9	character of the Number category required, e.g 0-9.
0	character of the Number category permitted but not required.
D	character of the Number category and larger than zero required, such as 1-9
d	character of the Number category and larger than zero permitted but not required, such as 1-9.
#	character of the Number category, or plus/minus sign permitted but not required.
H	Hexadecimal character required. A-F, a-f, 0-9.
h	Hexadecimal character permitted but not required.
B	Binary character required. 0-1.
b	Binary character permitted but not required.
Meta Character	Meaning
>	All following alphabetic characters are uppercased.
<	All following alphabetic characters are lowercased.
!	Switch off case conversion.
;c	Terminates the input mask and sets the <i>blank</i> character to c.
[] { }	Reserved.
\	Use \ to escape the special characters listed above to use them as separators.

Mask	Notes
000.000.000.000;_	IP address; blanks are _.
HH:HH:HH:HH:HH:HH;_	MAC address
0000-00-00	ISO Date; blanks are space
>AAAAA-AAAAA-AAAAA-AAAAA-AAAAA;#	License number; blanks are # and all (alphabetic) characters are converted to uppercase.

- 正则表达式

字符	描述
\	将下一个字符标记为一个特殊字符、或一个原义字符、或一个向后引用、或一个八进制转义字符。例如，“n”匹配字符“n”。“\n”匹配一个换行符。串行“\\”匹配“\”而“\c”则匹配“c”。
^	匹配输入字符串的开始位置。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。
\$	匹配输入字符串的结束位置。如果设置了RegExp对象的Multiline属性，\$也匹配“\n”或“\r”之前的位置。
*	匹配前面的子表达式零次或多次。例如，“zo*”能匹配“z”以及“zoo”。“*”等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。“+”等价于{1,}。
?	匹配前面的子表达式零次或一次。例如，“do(ez)?”可以匹配“does”或“does”中的“do”。?等价于{0,1}。
{n}	n是一个非负整数。匹配确定的n次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个o。
{n,}	n是一个非负整数。至少匹配n次。例如，“o{2,}”不能匹配“Bob”中的“o”，但是能匹配“fooooo”中的所有o。“o{1,1}”等价于“o+”。“o{0,1}”则等价于“o?”。
{n,m}	m和n均为非负整数，其中n<=m。最少匹配n次且最多匹配m次。例如，“o{1,3}”将匹配“fooooo”中的前三个o。“o{0,1}”等价于“o?”。请注意在逗号前和两个数之间不能有空格。
?	当该字符紧跟在任何一个其他限制符“*、+、?、{n}、{n,}、{n,m}”后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的模式则尽可能多的匹配所搜索的字符串。例如，对于字符串“oooo”，“o+?”将匹配单个“o”，而“o*”将匹配所有“o”。
.	匹配除“\n”之外的任何单个字符。要匹配包括“\n”在内的任何字符，请使用像“(. \n)”的模式。
(pattern)	匹配pattern并获取这一匹配。所获取的匹配可以从产生的Matches集合得到，在VBScript中使用SubMatches集合，在JavaScript中则使用\$0...\$9属性。要匹配圆括号字符，请使用“\("或“\)”。
(?:pattern)	匹配pattern但不获取匹配结果。也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用或字符“ ”来组合一个模式的各个部分是很有用。例如“industr(?:y ies)”就是一个比“industry industries”更简略的表达式。
(?=pattern)	正向肯定预查，在任何匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如“Windows(?=95 98 NT 2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
(?!pattern)	正向否定预查，在任何不匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如“Windows(?!95 98 NT 2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
(?<=pattern)	反向肯定预查，与正向肯定预查类似，只是方向相反。例如，“(?:<95 98 NT 2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。
(?<!pattern)	反向否定预查，与正向否定预查类似，只是方向相反。例如“(?:<95 98 NT 2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。
x y	匹配x或y，例如，“x food”能匹配“x”或“food”。“x food”则匹配“xood”或“food”。
[xyz]	字符集合。匹配所包含的任意一个字符。例如，“[abc]”可以匹配“plain”中的“a”。
[^xyz]	负值字符集合。匹配未包含的任意字符。例如，“[!abc]”可以匹配“plain”中的“p”。
[a-z]	字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符。例如，“[!a-z]”可以匹配任何不在“a”到“z”范围内的任意字符。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”。
\B	匹配非单词边界。“er\b”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。
\cx	匹配由x指明的控制字符。例如，\cM匹配一个Control-M或回车符。x的值必须为A-Z或a-z之一。否则，将c视为一个原义的“c”字符。

• 源码

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QRegularExpressionValidator>
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QString ip_mask = "000.000.000.000;-";
    ui->IPLineEdit->setInputMask(ip_mask);
    QString mac_mask = "HH:HH:HH:HH:HH:HH;-";
    ui->MAClineEdit->setInputMask(mac_mask);
    ui->pwdlineEdit->setEchoMode(QLineEdit::Password);
    QRegularExpression reg("[a-zA-Z0-9-_]+@[A-Za-z0-9]+\\. [a-zA-Z0-9]+");
    QValidator * validator = new QRegularExpressionValidator(reg, ui->emailineEdit);
    ui->emailineEdit->setValidator(validator);
}

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

Qt布局与页面切换

- Qt 中的布局有三种方式，水平布局，垂直布局，栅格布局。
- 具体见如下链接

<https://llfc.club/category?catid=2DX72J8YkeAknI5uTeM7ysTkgHS#!aid/2FqCn3phpuy0qm5aHMHa3wBFGXY>

QT主窗口

- 任何界面应用都有一个主窗口，今天我们谈谈主窗口相关知识。一个主窗口包括菜单栏，工具栏，状态栏，以及中心区域等部分。我们先从菜单栏说起

菜单栏

- 主菜单通常包括以下几个部分：
 1. 菜单项 (Menu)： 顶级菜单栏上的每个项目称为一个菜单项，点击菜单项会弹出子菜单或执行相应的操作。
 2. 子菜单 (Submenu)： 菜单项可以包含子菜单，子菜单是下拉菜单，包含了一组相关的菜单命令或选项。
 3. 动作 (Action)： 菜单项或子菜单中的每个命令或选项称为一个动作，通常与具体的操作或功能相关联。
- 在Qt中，可以通过QMenu和QAction类来创建菜单和动作。主菜单通常是由多个QMenu对象组成的，而每个QMenu对象又包含多个QAction对象。

```
#include <QMainWindow>
#include <QMenu>
#include <QMenuBar>

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr)
        : QMainWindow(parent)
    {
        // 创建菜单
        QMenu *fileMenu = menuBar()->addMenu(tr("文件"));
        QMenu *editMenu = menuBar()->addMenu(tr("编辑"));

        // 创建动作
        QAction *newAction = new QAction(tr("新建"), this);
        QAction *openAction = new QAction(tr("打开"), this);
        QAction *saveAction = new QAction(tr("保存"), this);

        QAction *cutAction = new QAction(tr("剪切"), this);
        QAction *copyAction = new QAction(tr("复制"), this);
        QAction *pasteAction = new QAction(tr("粘贴"), this);

        // 将动作添加到菜单中
        fileMenu->addAction(newAction);
        fileMenu->addAction(openAction);
        fileMenu->addAction(saveAction);

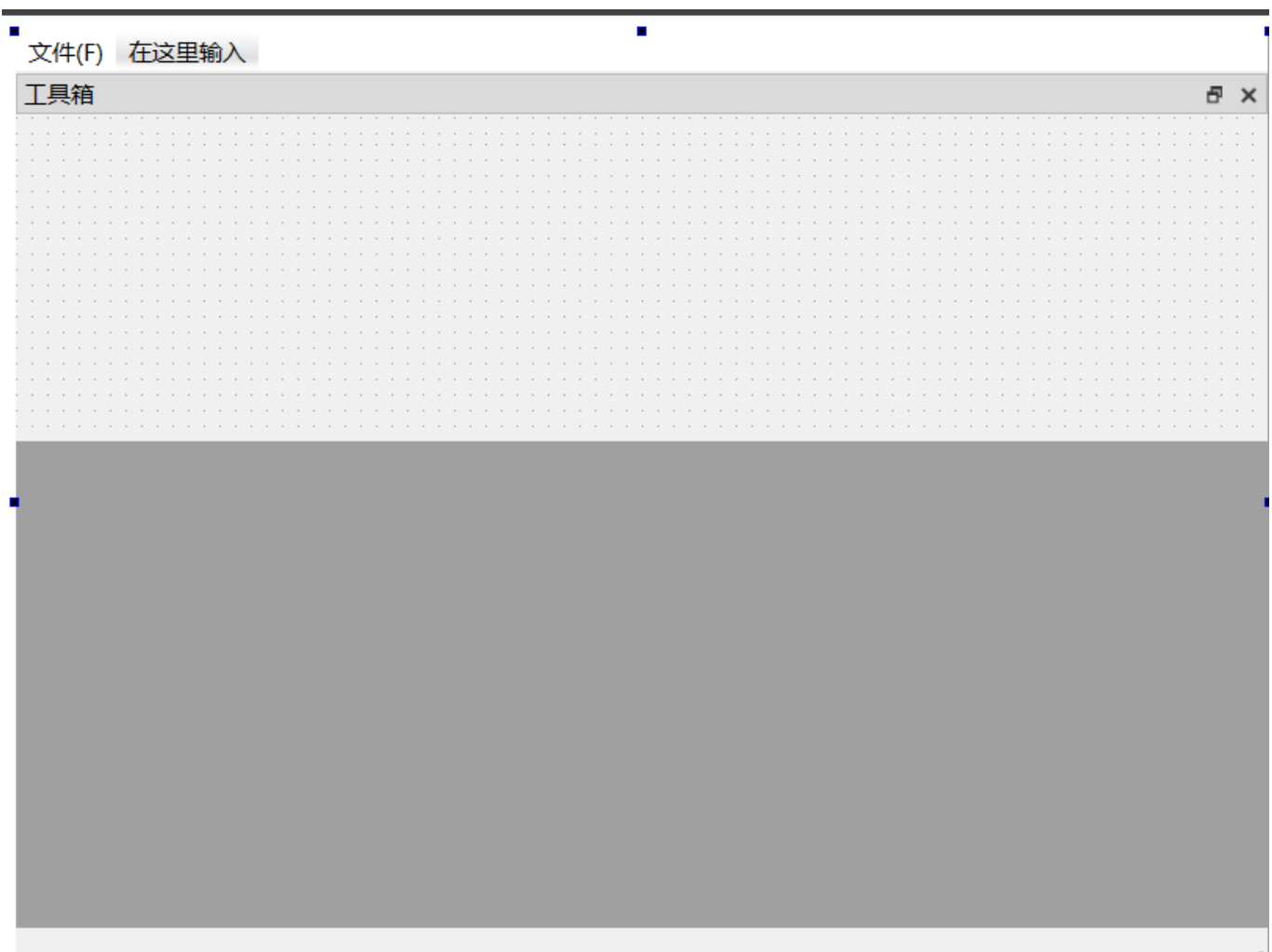
        editMenu->addAction(cutAction);
        editMenu->addAction(copyAction);
        editMenu->addAction(pasteAction);
    }
}
```

```
};
```

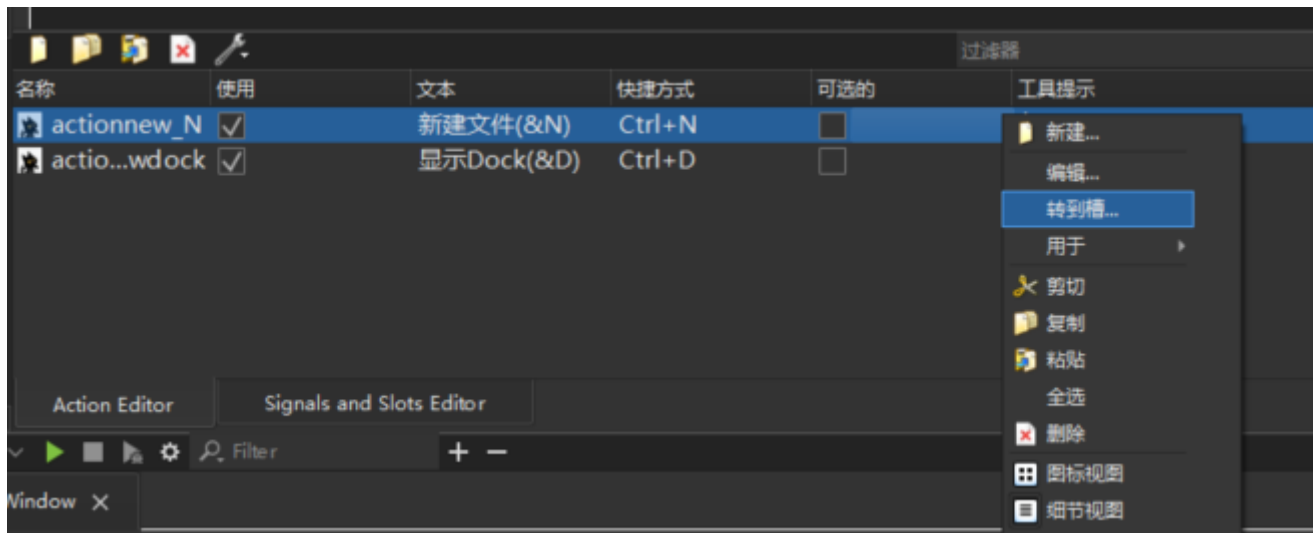
通过Ui来实现

- 我们创建一个主窗口应用程序, 在ui文件里的菜单栏里有“在这里输入”的一个菜单, 我们双击它输入“文件(&F)”, 这样通过点击alt + F 就可以弹出文件菜单。点击文件菜单, 同样会弹出“在这里输入”, 我们双击它编辑输入“新建文件(&N)”, 可以在右侧的属性栏里为其添加一个图标, 同样, 我们再添加一个“显示 Dock(&D)”的菜单。我们再从左侧的控件Containers里拖动Dock Widget到中心区域, 在dock widget中添加一个按钮, 一个fontComboBox, 一个QTextEdit。DockWidget添加的控件并不会影响主窗口的centralwidget, 我们在centralwidget中添加一个QMdiArea控件。

MDI Area



- 我们可以选择Action Editor中的两个action, 分别为其添加槽函数, 右击actionnew_N选择转到槽, 然后选择trigger信号, Qt会自动生成槽函数代码



```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QDebug>
#include <QTextEdit>
#include <QMdiSubWindow>
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

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

void MainWindow::on_actionNew_triggered()
{
    qDebug() << "新建文件...";
    QTextEdit *textedit = new QTextEdit(this);
    auto childWindow = ui->mdiArea->addSubWindow(textedit);
    childWindow->setWindowTitle(tr("文本编辑框"));
    childWindow->show();
}

void MainWindow::on_actionshowduck_S_triggered()
{
    qDebug() << "显示dock widget";
    ui->dockWidget_2->show();
}
```

- 因为QMdiArea是一个多窗口控件，这样我们每次点击新建菜单就会在窗口的中心部件中创建一个子窗口，多次点击会生成多个子窗口。点击显示dock菜单就会显示dockwidget。

通过代码实现

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QDebug>
#include <QTextEdit>
#include <QMdiSubWindow>
#include <QMenu>
#include <QActionGroup>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QMenu * editMenu = ui->menubar->addMenu(tr("编辑(&E)"));
    editMenu->addSeparator();
    QAction * action_Open = editMenu->addAction(QIcon(":/img/head.jpg"), tr("打开文件(&O)"));
    action_Open->setShortcut(QKeySequence("Ctrl+O"));
    //建立动作组
    QActionGroup * group = new QActionGroup(this);
    QAction * action_L = group->addAction(tr("左对(&L)"));
    action_L->setCheckable(true);
    QAction * action_R = group->addAction(tr("右对(&R)"));
    action_R->setCheckable(true);
    QAction * action_C = group->addAction(tr("居中(&C)"));
    action_C->setCheckable(true);
    editMenu->addSeparator();
    editMenu->addAction(action_L);
    editMenu->addAction(action_R);
    editMenu->addAction(action_C);
    connect(action_Open, &QAction::triggered, this,
    &MainWindow::on_action_open_triggered);
}

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

void MainWindow::on_actionNew_triggered()
{
    qDebug() << "新建文件...";
    QTextEdit *textedit = new QTextEdit(this);
    auto childWindow = ui->mdiArea->addSubWindow(textedit);
    childWindow->setWindowTitle(tr("文本编辑框"));
    childWindow->show();
}
```

```
void MainWindow::on_action_open_triggered()
{
    qDebug() << "打开文件...";
}

void MainWindow::on_actionshowduck_S_triggered()
{
    qDebug() << "显示dock Widget";
    ui->dockWidget_2->show();
}
```

- 除了可以为菜单栏添加动作外，还可以添加动作组
- 我们也可以自定义动作类，我们接来自定义一个动作类，动作类包含一个label和lineEdit。自定义的动作类叫MyAction，声明如下

```
#ifndef MYACTION_H
#define MYACTION_H

#include <QWidgetAction>
#include <QLineEdit>
#include <QObject>
class MyAction : public QWidgetAction
{
    Q_OBJECT
signals:
    void getText(const QString& string);

public:

    explicit MyAction(QObject * parent = 0);
    virtual ~MyAction();
protected:

    virtual QWidget *createWidget(QWidget *parent);

private slots:
    void sentText();
private:
    //声明行编辑器对象
    QLineEdit* lineEdit;
};
#endif // MYACTION_H
```

- createWidget 为一个虚函数，继承自QWidgetAction，将Action加入菜单或者工具栏就会调用createWidget函数。接下来我们实现这个类

```

#include "myaction.h"
#include <QSplitter>
#include <QLabel>

MyAction::MyAction(QObject * parent):QWidgetAction(parent){
    // 创建行编辑器
    lineEdit = new QLineEdit;
    // 将行编辑器的按下回车键信号和发送文本槽关联
    connect(lineEdit, &QLineEdit::returnPressed, this, &MyAction::sentText);
}

QWidget* MyAction::createWidget(QWidget * parent){
    if(parent->inherits("QMenu") || parent->inherits("QToolBar")){
        QSplitter * splitter = new QSplitter(parent);
        QLabel * label = new QLabel;
        label->setText(tr("插入文本:"));
        splitter->addWidget(label);
        splitter->addWidget(lineEdit);
        return splitter;
    }
    return 0;
}

void MyAction::sentText()
{
    emit getText(lineEdit->text());
    lineEdit->clear();
}

```

- 构造函数里创建了一个LineEdit，然后绑定了LineEdit的返回信号，在sentText槽函数里发送了getText信号，然后清除了lineEdit里的内容。createWidget里判断了父节点如果是QMenu或者QToolBar，就创建一个splitter，然后将label和lineEdit都加入splitter。在mainwindow的构造函数中创建MyAction，并且加入菜单里。然后将MyAction的getText的信号和MainWindow的setText函数绑定在一起。

```

MyAction * action = new MyAction(this);
editMenu->addAction(action);
connect(action, &MyAction::getText, this, &MainWindow::setText);
void MainWindow::setText(const QString &string){
    ui->textEdit->setText(string);
}

```

工具栏

- 工具栏相比菜单栏更容易操作，更加直观，添加方式和菜单的方式类似，可以添加label，按钮，以及spinbox等。可以通过ui添加，也可以通过代码添加，下面用代码添加工具栏菜单。

```
//工具栏添加元素
QToolButton * toolBtn = new QToolButton(this);
toolBtn->setText(tr("颜色"));
QMenu* colorMenu = new QMenu(this);
colorMenu->addAction(tr("红色"));
colorMenu->addAction(tr("绿色"));
toolBtn->setMenu(colorMenu);
toolBtn->setPopupMode(QToolButton::MenuButtonPopup);
ui->toolBar->addWidget(toolBtn);
QSpinBox* spinBox = new QSpinBox(this);
ui->toolBar->addSeparator();
ui->toolBar->addWidget(spinBox);
```

状态栏

- 状态栏在窗口的下方，一般在右下方，左下方的为临时的，右下方的为永久的

```
void MainWindow::init_status_bar(){
    //显示临时消息
    ui->statusbar->showMessage(tr("欢迎使用多文档编辑器"),2000);
    //创建标签
    QLabel* permanent = new QLabel(this);
    permanent->setFrameStyle(QFrame::Box | QFrame::Sunken);
    permanent->setText("llfc.club");
    ui->statusbar->addPermanentWidget(permanent);
}
```

QTextEdit 文本编辑器

- 它是一个富文本编辑器而QPlainTextEdit是纯文本编辑器

文本块

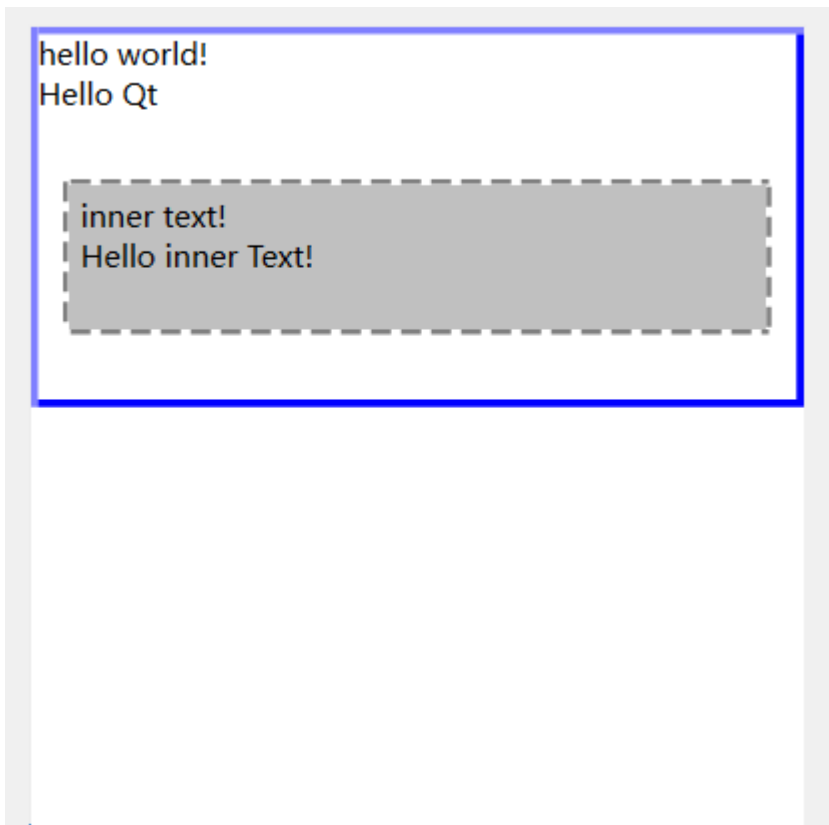
- 我们在MainWindow的ui文件中添加了textedit插件，然后在MainWindow的构造函数中写代码，修改文本框样式

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QTextDocument>
#include <QTextFrame>
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QTextDocument *doc = ui->textEdit->document();
    // 得到所有框架的根框架
    QTextFrame *root_frame = doc->rootFrame();
```

```
// 文本框的格式
QTextFrameFormat format;
//设置边框颜色
format.setBorderBrush(Qt::blue);
//设置边框宽度
format.setBorder(3);
//将样式加入道根框架
root_frame->setFrameFormat(format);
ui->textEdit->insertPlainText("hello world!\n");
ui->textEdit->insertPlainText("Hello Qt\n");
QTextFrameFormat frameFormat;
frameFormat.setBackground(Qt::lightGray);
//设置外边框
frameFormat.setMargin(10);
//设置内边框
frameFormat.setPadding(5);
//设置边框宽度
frameFormat.setBorder(2);
//设置边框样式为虚线
frameFormat.setBorderStyle(QTextFrameFormat::BorderStyle_Dashed);
//获取textedit的光标
QTextCursor cursor = ui->textEdit->textCursor();
//将上述定义好的文本框架格式用到光标所在位置，光标位置就是输入位置
cursor.insertFrame(frameFormat);
ui->textEdit->insertPlainText("inner text!\n");
ui->textEdit->insertPlainText("Hello inner Text!\n");
}

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

- 运行效果



遍历文本块

- 默认情况下，Qt 将文本按照换行符进行分割，每个文本块对应一行文字。但是，你也可以通过设置 QTextDocument 的文本格式来自定义文本块的边界，例如设置段落间距、文本对齐方式等，从而影响文本块的形成。
- 我们可以遍历文本块和框架节点。先在构造函数中添加一个菜单，用来打印Frame和TextBlock

只遍历根框架

```
//遍历根框架
QAction* action_frame = new QAction("Frame", this);
connect(action_frame, &QAction::triggered, this, &MainWindow::showTextFrame);
ui->mainToolBar->addAction(action_frame);

void MainWindow::showTextFrame()
{
    //得到文本文档
    auto doc = ui->textEdit->document();
    //从文档中得到根框架
    auto rootFrame = doc->rootFrame();
    //遍历根框架，根框架里面可能还有其他子框架所以要考虑其实不是样式(frame)
    for(auto iter = rootFrame->begin(); iter != rootFrame->end(); iter++){
        auto cur_frame = iter.currentFrame();
        auto cur_block = iter.currentBlock();
        //是样式
        if(cur_frame){
            qDebug() << "cur node is frame " ;
        }
        //不是样式
    }
```

```
        else if(cur_block.isValid()){
            qDebug() << "cur node is text block ,text is " << cur_block.text();
        }
    }
}
```

直接遍历文本块

- 遍历文本块不受子框架影响，即使文本块在子框架中也能遍历到

```
//遍历根框架及其子框架里的文本
    QAction* action_textBlock = new QAction(tr("文本块"),this);
    connect(action_textBlock, &QAction::triggered, this,
&MainWindow::showTextBlock);
    ui->mainToolBar->addAction(action_textBlock);

void MainWindow::showTextBlock()
{
    QTextDocument* document = ui->textEdit->document();
    //得到第一个文本块
    QTextBlock block = document->firstBlock();
    //遍历所有文本块
    for(int i = 0; i < document->blockCount(); i++){
        qDebug() << tr("文本块%1, 文本块首行行号%2, 长度%3, 内
容%4").arg(i).arg(block.firstLineNumber()).arg(block.length())
                << block.text();
        block = block.next();
    }
}
```

设置文本块样式

- 之前我们设置的都是文本框架的样式，这次我们设置文本块的样式.
- 在构造函数中添加字体菜单，用来设置文本块的字体样式

```
// 设置本文块样式
    QAction* action_font = new QAction(tr("字体"), this);
    action_font->setCheckable(true);
    connect(action_font, &QAction::toggled, this, &MainWindow::setTextFont);
    ui->mainToolBar->addAction(action_font);

void MainWindow::setTextFont(bool checked)
{
    //选中状态
    if(checked){
        //得到光标
        QTextCursor cursor = ui->textEdit->textCursor();
        //文本格式样式
```



```

        QTextBlockFormat blockFormat;
        //设置为居中对齐
        blockFormat.setAlignment(Qt::AlignCenter);
        //插入样式
        cursor.insertBlock(blockFormat);
        //字符样式
        QTextCharFormat charFormat;
        charFormat.setBackground(Qt::lightGray);
        charFormat.setForeground(Qt::blue);
        charFormat.setFont(QFont(tr("宋体"), 12, QFont::Bold, true));
        charFormat.setFontUnderline(true);
        //设置字符样式
        cursor.setCharFormat(charFormat);
        //插入字体
        cursor.insertText(tr("插入字体"));
    }
    //非选中状态
    else{
        QTextCursor cursor = ui->textEdit->textCursor();
        QTextBlockFormat blockFormat;
        //左对齐
        blockFormat.setAlignment(Qt::AlignLeft);
        cursor.insertBlock(blockFormat);
        QTextCharFormat charFormat;
        //          charFormat.setBackground(Qt::white);
        //          charFormat.setForeground(Qt::black);
        //          charFormat.setFont(QFont(tr("微软雅黑"), 12, QFont::Normal,
false));
        //          charFormat.setFontUnderline(false);
        cursor.setCharFormat(charFormat);
        cursor.insertText(tr("微软雅黑字体"));
    }
}

```

- 其中QTextBlockFormat 是 Qt 中用于描述文本块格式的类。文本块是 QTextDocument 中的基本单元，它通常对应于文本中的一行或一段。QTextBlockFormat 可以用于控制文本块的各种属性，例如对齐方式、缩进、间距、行高等。
- 以下为常用的方法与属性
 1. setAlignment(Qt::Alignment align)：设置文本块的对齐方式。
 2. setIndent(int indentation)：设置文本块的缩进。
 3. setLeadingMargin(int margin)：设置文本块的行首缩进。
 4. setTrailingMargin(int margin)：设置文本块的行尾缩进。
 5. setTopMargin(int margin)：设置文本块的顶部间距。
 6. setBottomMargin(int margin)：设置文本块的底部间距。
 7. setLineHeight(int height, QTextBlockFormat::LineHeightTypes heightType)：设置文本块的行高，可以指定行高的类型，如固定高度、行间距的倍数等。

插入表格列表图片

- QTextEdit也支持插入表格，列表，图片等资源。在MainWindow的构造函数里增加列表，图片，表格的信号和槽函数连接逻辑

```
QAction* action_textTable = new QAction(tr("表格"), this);
QAction* action_textList = new QAction(tr("列表"), this);
QAction* action_textImage = new QAction(tr("图片"), this);
connect(action_textTable, &QAction::triggered, this, &MainWindow::insertTable);
ui->mainToolBar->addAction(action_textTable);
connect(action_textList, &QAction::triggered, this, &MainWindow::insertList);
ui->mainToolBar->addAction(action_textList);
connect(action_textImage, &QAction::triggered, this, &MainWindow::insertImage);
ui->mainToolBar->addAction(action_textImage);

void MainWindow::insertTable()
{
    QTextCursor cursor = ui->textEdit->textCursor();
    QTextTableFormat format;
    //设置间距宽度
    format.setCellSpacing(2);
    //设置外边框宽度
    format.setCellPadding(10);
    //插入表格 宽度 高度 和样式
    cursor.insertTable(2,2,format);
}

void MainWindow::insertList()
{
    QTextListFormat format;
    //有序列表
    format.setStyle(QTextListFormat::ListDecimal);
    ui->textEdit->textCursor().insertList(format);
}

void MainWindow::insertImage()
{
    QTextImageFormat format;
    //得将图片资源加入同级目录
    format.setName(":/img/head.jpg");
    ui->textEdit->textCursor().insertImage(format);
}
```

实现查找功能

- 在构造函数里写

```
QAction* action_textFind = new QAction(tr("查找"), this);
connect(action_textFind, &QAction::triggered, this,
&MainWindow::textfind);
ui->mainToolBar->addAction(action_textFind);

//类的一个成员
findDialog = new QDialog(this);
```

```
//类的另一个成员
lineEdit = new QLineEdit(findDialog);
//加入一个按钮
QPushButton * btn = new QPushButton(findDialog);

btn->setText(tr("查找下一个"));
connect(btn, &QPushButton::clicked, this, &MainWindow::findnext);
QVBoxLayout* layout = new QVBoxLayout();
layout->addWidget(lineEdit);
layout->addWidget(btn);
findDialog->setLayout(layout);

void MainWindow::textfind()
{
    findDialog->show();
}

void MainWindow::findnext()
{
    QString string = lineEdit->text();
    bool isFind = ui->textEdit->find(string, QTextDocument::FindBackward);
    if(isFind){
        qDebug() << tr("行号%1, 列号%2, ").arg(ui->textEdit-
>textCursor().blockNumber()).arg(ui->textEdit->textCursor().columnNumber());
    }
}
```