在C++应用程序中使用设计器 UI 文件

Qt 设计器 UI 文件以 XML 格式表示表单的小部件树。表格可以处理:

- 在编译时，这意味着表单将转换为C++可以编译的代码。
- 在运行时，这意味着表单由 QUiLoader 类处理，该类在分析 XML 文件时动态构造小部件树。

编译时表单处理

您可以使用*Qt设计器*创建用户界面组件，并使用Qt的集成构建工具qmake和uic，在构建应用程序时为它们生成代码。生成的代码包含窗体的用户界面对象。它是一个C++结构，包含:

- 指向窗体的构件、布局、布局项、按钮组和操作的指针。
- 调用的成员函数，用于在父小部件上构建小部件树。setupUi()
- 调用的成员函数，用于处理窗体的字符串属性的转换。有关详细信息，请参阅响应语言更改。retranslateUi()

生成的代码可以包含在应用程序中，并直接从应用程序中使用。或者，您可以使用它来扩展标准小部件的子类。

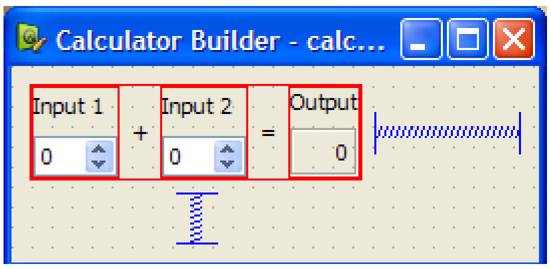
编译时处理的表单可以通过以下方法之一在应用程序中使用:

- 直接方法：构造一个小部件以用作组件的占位符，并在其中设置用户界面。
- 单一继承方法：对窗体的基类（例如 QWidget 或 QDialog）进行子类化，并包括窗体的用户界面对象的专用实例。
- 多重继承方法：对窗体的基类和窗体的用户界面对象进行子类化。这允许在表单中定义的小部件直接从子类的范围内使用。

为了进行演示，我们创建了一个简单的计算器表单应用程序。它基于原始的计算器表单示例。

该应用程序由一个源文件和一个 UI 文件组成。main.cpp

使用*Qt设计器*设计的文件如下所示：calculatorform.ui



```
HEADERS     = calculatorform.h
```

此文件的特殊功能是声明，它告诉要使用 处理哪些文件。在这种情况下，该文件用于创建可由声明中列出的任何文件使用的文件。FORMSqmakeuiccalculatorform.uiui_calculatorform.hSOURCES

注意： 您可以使用 Qt 创建器创建计算器窗体项目。它会自动生成主文件.cpp、UI 和 .pro 文件，然后您可以修改这些文件。

直接方法

要使用直接方法，我们将文件直接包含在：ui_calculatorform.hmain.cpp

```
#include "ui_calculatorform.h"
```

该函数通过构造一个标准 QWidget 来创建计算器小部件，我们用它来托管文件所描述的用户界面。maincalculatorform.ui

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget widget;
    Ui::CalculatorForm ui;
    ui.setupUi(&widget);

    widget.show();
    return app.exec();
}
```

在本例中，是文件中的接口描述对象，用于设置对话框的所有小部件及其信号和槽之间的连接。Ui::CalculatorFormui_calculatorform.h

直接方法提供了一种在应用程序中使用简单、独立的组件的快速简便的方法。但是，使用*Qt设计器*创建的组合通常需要与应用程序代码的其余部分紧密集成。例如，上面提供的代码将编译并运行，但QSpinBox对象不会与QLabel交互，因为我们需要一个自定义插槽来执行添加操作并在QLabel中显示结果。为了实现这一点，我们需要使用单一继承方法。CalculatorForm

The Single Inheritance Approach

To use the single inheritance approach, we subclass a standard Qt widget and include a private instance of the form's user interface object. This can take the form of:

- A member variable
- A pointer member variable

used in this way expose the widgets and layouts used in the form to the Qt widget subclass, and provide a standard system for making signal and slot connections between the user interface and other objects in your application. The generated structure is a member of the class.`Ui::CalculatorForm`

This approach is used in the Calculator Form example.

To ensure that we can use the user interface, we need to include the header file that generates before referring to :`uicUi::CalculatorForm`

```
#include "ui_calculatorform.h"
```

This means that the file must be updated to include :`.procalculatorform.h`

```
HEADERS      = calculatorform.h
```

The subclass is defined in the following way:

```
class CalculatorForm : public QWidget
{
    Q_OBJECT

public:
    explicit CalculatorForm(QWidget *parent = nullptr);

private slots:
    void on_inputSpinBox1_valueChanged(int value);
    void on_inputSpinBox2_valueChanged(int value);

private:
    Ui::CalculatorForm ui;
};
```

The important feature of the class is the private object which provides the code for setting up and managing the user interface.`ui`

The constructor for the subclass constructs and configures all the widgets and layouts for the dialog just by calling the object's function. Once this has been done, it is possible to modify the user interface as needed.`uisetupUi()`

```
CalculatorForm::CalculatorForm(QWidget *parent)
    : QWidget(parent)
{
    ui.setupUi(this);
}
```

We can connect signals and slots in user interface widgets in the usual way by adding the on_<object name> -

encapsulation of the user interface widget variables within the data member. We can use this method to define a number of user interfaces within the same widget, each of which is contained within its own namespace, and overlay (or compose) them. This approach can be used to create individual tabs from existing forms, for example.ui

Using a Pointer Member Variable

Alternatively, the structure can be made a pointer member of the class. The header then looks as follows:Ui::CalculatorForm

```
namespace Ui {
    class CalculatorForm;
}

class CalculatorForm : public QWidget
...
virtual ~CalculatorForm();
...
private:
    Ui::CalculatorForm *ui;
...
```

The corresponding source file looks as follows:

```
#include "ui_calculatorform.h"

CalculatorForm::CalculatorForm(QWidget *parent) :
    QWidget(parent), ui(new Ui::CalculatorForm)
{
    ui->setupUi(this);
}

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

The advantage of this approach is that the user interface object can be forward-declared, which means that we do not have to include the generated file in the header. The form can then be changed without recompiling the dependent source files. This is particularly important if the class is subject to binary compatibility restrictions.ui_calculatorform.h

We generally recommend this approach for libraries and large applications. For more information, see Creating Shared Libraries.

The Multiple Inheritance Approach

Forms created with *Qt Designer* can be subclassed together with a standard QWidget-based class. This approach makes all the user interface components defined in the form directly accessible within the scope of the subclass, and enables signal and slot connections to be made in the usual way with the connect() function.

We need to include the header file that generates from the file, as follows:ui_calculatorform.ui

```
#include "ui_calculatorform.h"
```

The class is defined in a similar way to the one used in the single inheritance approach, except that this time we inherit from *both* QWidget and , as follows:`Ui::CalculatorForm`

```
class CalculatorForm : public QWidget, private Ui::CalculatorForm
{
    Q_OBJECT

public:
    explicit CalculatorForm(QWidget *parent = nullptr);

private slots:
    void on_inputSpinBox1_valueChanged(int value);
    void on_inputSpinBox2_valueChanged(int value);
};
```

We inherit privately to ensure that the user interface objects are private in our subclass. We can also inherit it with the or keywords in the same way that we could have made public or protected in the previous case.`Ui::CalculatorFormpublicprotectedui`

The constructor for the subclass performs many of the same tasks as the constructor used in the single inheritance example:

```
CalculatorForm::CalculatorForm(QWidget *parent)
    : QWidget(parent)
{
    setupUi(this);
}
```

In this case, the widgets used in the user interface can be accessed in the same say as a widget created in code by hand. We no longer require the prefix to access them.`ui`

Reacting to Language Changes

Qt notifies applications if the user interface language changes by sending an event of the type QEvent::LanguageChange. To call the member function of the user interface object, we reimplement in the form class, as follows:`retranslateUi()QWidget::changeEvent()`

```
void CalculatorForm::changeEvent(QEvent *e)
{
    QWidget::changeEvent(e);
    switch (e->type()) {
    case QEvent::LanguageChange:
            break;
        }
    }
```

Run Time Form Processing

Alternatively, forms can be processed at run time, producing dynamically- generated user interfaces. This can be done using the QtUiTools module that provides the QUiLoader class to handle forms created with *Qt Designer*.

The UiTools Approach

A resource file containing a UI file is required to process forms at run time. Also, the application needs to be configured to use the QtUiTools module. This is done by including the following declaration in a project file, ensuring that the application is compiled and linked appropriately.qmake

```
QT += uitools
```

The QUiLoader class provides a form loader object to construct the user interface. This user interface can be retrieved from any QIODevice, e.g., a QFile object, to obtain a form stored in a project's resource file. The QUiLoader::load() function constructs the form widget using the user interface description contained in the file.

The QtUiTools module classes can be included using the following directive:

```
#include <QtUiTools>
```

The QUiLoader::load() function is invoked as shown in this code from the Text Finder example:

```
static QWidget *loadUiFile(QWidget *parent)
{
    QFile file(":/forms/textfinder.ui");
    file.open(QIODevice::ReadOnly);

    QUiLoader loader;
    return loader.load(&file, parent);
}
```

In a class that uses QtUiTools to build its user interface at run time, we can locate objects in the form using QObject::findChild(). For example, in the following code, we locate some components based on their object names and widget types:

```
    ui_findButton = findChild<QPushButton*>("findButton");
    ui_textEdit = findChild<QTextEdit*>("textEdit");
```

Processing forms at run-time gives the developer the freedom to change a program's user interface, just by changing the UI file. This is useful when customizing programs to suit various user needs, such as extra large icons or a different colour scheme for accessibility support.

Automatic Connections

The signals and slots connections defined for compile time or run time forms can either be set up manually or automatically, using QMetaObject's ability to make connections between signals and suitably-named slots.

Generally, in a QDialog, if we want to process the information entered by the user before accepting it, we need to connect the clicked() signal from the **OK** button to a custom slot in our dialog. We will first show an example of the dialog in which the slot is connected by hand then compare it with a dialog that uses automatic connection.

A Dialog Without Auto-Connect

We define the dialog in the same way as before, but now include a slot in addition to the constructor:

```
class ImageDialog : public QDialog, private Ui::ImageDialog
{
    Q_OBJECT

public:
    ImageDialog(QWidget *parent = 0);

private slots:
    void checkValues();
};
```

The slot will be used to validate the values provided by the user.`checkValues()`

In the dialog's constructor we set up the widgets as before, and connect the **Cancel** button's clicked() signal to the dialog's reject() slot. We also disable the autoDefault property in both buttons to ensure that the dialog does not interfere with the way that the line edit handles return key events:

```
ImageDialog::ImageDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);
    okButton->setAutoDefault(false);
    cancelButton->setAutoDefault(false);
    ...
    connect(okButton, SIGNAL(clicked()), this, SLOT(checkValues()));
}
```

We connect the **OK** button's clicked() signal to the dialog's checkValues() slot which we implement as follows:

```
void ImageDialog::checkValues()
```

```
        (void) QMessageBox::information(this, tr("No Image Name"),
            tr("Please supply a name for the image."), QMessageBox::Cancel);
    else
        accept();
}
```

This custom slot does the minimum necessary to ensure that the data entered by the user is valid - it only accepts the input if a name was given for the image.

Widgets and Dialogs with Auto-Connect

Although it is easy to implement a custom slot in the dialog and connect it in the constructor, we could instead use QMetaObject's auto-connection facilities to connect the **OK** button's clicked() signal to a slot in our subclass. automatically generates code in the dialog's function to do this, so we only need to declare and implement a slot with a name that follows a standard convention:`uicsetupUi()`

```
void on_<object name>_<signal name>(<signal parameters>);
```

Using this convention, we can define and implement a slot that responds to mouse clicks on the **OK** button:

```
class ImageDialog : public QDialog, private Ui::ImageDialog
{
    Q_OBJECT

public:
    ImageDialog(QWidget *parent = 0);

private slots:
    void on_okButton_clicked();
};
```

Another example of automatic signal and slot connection would be the Text Finder with its slot.`on_findButton_clicked()`

We use QMetaObject's system to enable signal and slot connections:

```
    QMetaObject::connectSlotsByName(this);
```

This enables us to implement the slot, as shown below:

```
void TextFinder::on_findButton_clicked()
{
    QString searchString = ui_lineEdit->text();
    QTextDocument *document = ui_textEdit->document();
```

```
        // undo previous change (if any)
        document->undo();

        if (searchString.isEmpty()) {
            QMessageBox::information(this, tr("Empty Search Field"),
                                     tr("The search field is empty. "
                                        "Please enter a word and click Find."));
        } else {
            QTextCursor highlightCursor(document);
            QTextCursor cursor(document);

            cursor.beginEditBlock();
        ...
            cursor.endEditBlock();

            if (found == false) {
                QMessageBox::information(this, tr("Word Not Found"),
                                         tr("Sorry, the word cannot be found."));
            }
        }
    }
```

Automatic connection of signals and slots provides both a standard naming convention and an explicit interface for widget designers to work to. By providing source code that implements a given interface, user interface designers can check that their designs actually work without having to write code themselves.

‹ Using Stylesheets with Qt Designer

Using a Designer UI File in Your Qt for Python Application ›

Qt DOCUMENTATION

Support

Support Services
Professional Services
Partners
Training

For Customers

Support Center
Downloads
Qt Login
Contact Us

Customer Success

Community

Contribute to Qt
Forum
Wiki
Downloads

Marketplace

Feedback Sign In