

Day 1 (2020-07-18)

1. QT介绍

1.1 什么是QT?

Qt是一个跨平台的c++图形用户界面应用程序框架。

Qt是一个框架，这个框架是用来做用户图形界面编程，GUI编程。

其特点是跨平台

1.2 Qt历史

- 1991年 Qt最早由奇趣科技开发
- 1996年 进入商业领域，目前流行的Linux桌面环境KDE的基础
- 2008年 奇趣科技被诺基亚公司收购，Qt称为诺基亚旗下编程语言
- 2012年 Qt又被Digia公司收购
- 2014年 扩平台集成开发环境Qt Creator3.1.0发布，同年5月配发Qt5.3，至此实现对iOS、Android、WP等各平台全面支持。

1.3 支持平台

- Windows - XP、Vista、Win7、Win8、Win2008、Win10
- Unix/X11 - Linux、Sun Solaris、HP-UX、Compaq
- Macintosh - Mac OS X
- Embedded - 由帧缓冲支持的嵌入式Linux平台

2. Qt框架

2.1 新建项目

- 选择Qt Widgets Application



- 创建项目不可以包含中文

名称:

创建路径:


☒ 设为默认的项目路径

- 选择编译的套件，这里选择MinGW

Kit Selection

Qt Creator can use the following kits for project **01_QtTest**:

☒ Select all kits

☒  Desktop Qt 5.9.0 MinGW 32bit

- 在类信息中取消勾选窗口的创建

类信息

指定您要创建的源码文件的基本类信息。

类名(C):	<input type="text" value="MainWindow"/>
基类(B):	<input type="text" value="QMainWindow"/>
头文件(H):	<input type="text" value="mainwindow.h"/>
源文件(S):	<input type="text" value="mainwindow.cpp"/>
创建界面(G):	<input type="checkbox"/> ← 取消勾选
界面文件(F):	<input type="text" value="mainwindow.ui"/>

- 对于基类，Qt提供三个类，选择QWidget类

QMainWindow 在PC中使用，带菜单栏

QWidget 所有控件的基类

QDialog 对话框

类名(C):	<input type="text" value="MainWindow"/>
基类(B):	<input type="text" value="QMainWindow"/>
头文件(H):	<input type="text" value="QMainWindow"/>
	<input type="text" value="QWidget"/>
	<input type="text" value="QDialog"/>

图中的类名是子类的名字，取名为"MyWidget"，Qt会为我们自动生成一个类：

```
class Mywidget:public QWidget
```

- 界面说明



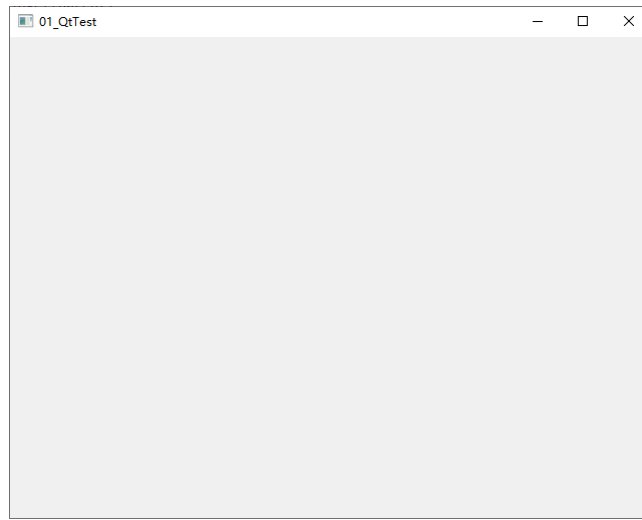
第一个是“编译-运行”

第二个是“编译-调试”

第三个是“编译-不运行”

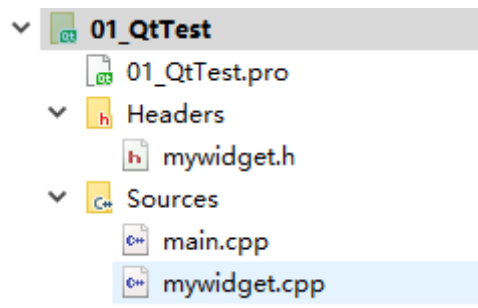
快捷键：Ctrl+R

- 结果



一个空窗口

2.2 目录结构



- 项目文件

每一个应用程序都对应一个项目文件 “01_QtTest.pro”

- main.cpp

主函数

- mywidget.h

该头文件自动生成“MyWidget”类

- mywidget.cpp

1. 从main.cpp文件开始看起：

```

#include "mywidget.h"

// QApplication应用程序类
// Qt头文件没有.h
// 头文件和类名一样
// 都是Q开头，前两个字母大写
#include <QApplication>

int main(int argc, char *argv[])
{
    // 有且只有一个应用程序类的对象
    QApplication a(argc, argv);

    // MyWidget继承QWidget          用户自己编写
    // QWidget是一个窗口基类
    // 所以MyWidget也是窗口类
    // w是一个窗口类对象
    MyWidget w;

    // 窗口创建默认是隐藏的，需要人为的显示
    w.show();

    // 让程序一直执行，等待用户操作
    // 等待事件的发生
    return a.exec();
}

```

2. 看一下.h文件:

```

#ifndef MYWIDGET_H
#define MYWIDGET_H

#include <QWidget>

class MyWidget : public QWidget
{
    // Q_OBJECT // 信号与槽的时候需要

public:
    MyWidget(QWidget *parent = 0);
    ~MyWidget();
};

#endif // MYWIDGET_H

```

其中“Q_OBJECT”是使用信号与槽的时候才用到的，这里可以对其进行屏蔽，不影响结果

快捷键说明，将光标锁定到文件中，按“F4”跳转至对应cpp文件

"Alt+0"切换左边的文件管理窗口

"F1"帮助文档

3. 看一下项目文件pro

```
# 代表模块
QT      += core gui

# 高于4版本，添加 QT += widgets，其目的是为了兼容Qt4
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

# 应用程序的名字
TARGET = NickTest
# 指定makefile的类型，app（还可以选lib）
TEMPLATE = app

# 源文件 .cpp文件
SOURCES += main.cpp \
           mywidget.cpp

# 头文件 .h文件
HEADERS += mywidget.h
```

- 首先将光标放到.h文件中QWidget上，按F1调到对应的帮助文档，如下图，其中可以看到头文件“QWidget”需要包含的“模块”

QWidget Class

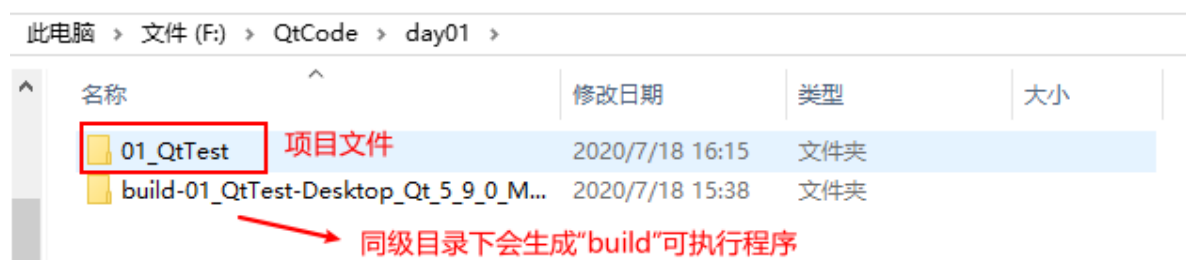
The `QWidget` class is the base class of all user interface objects.

[More...](#)

Header: `#include <QWidget>` → 所包含的头文件

qmake: `QT += widgets` 需要的哪一个模块

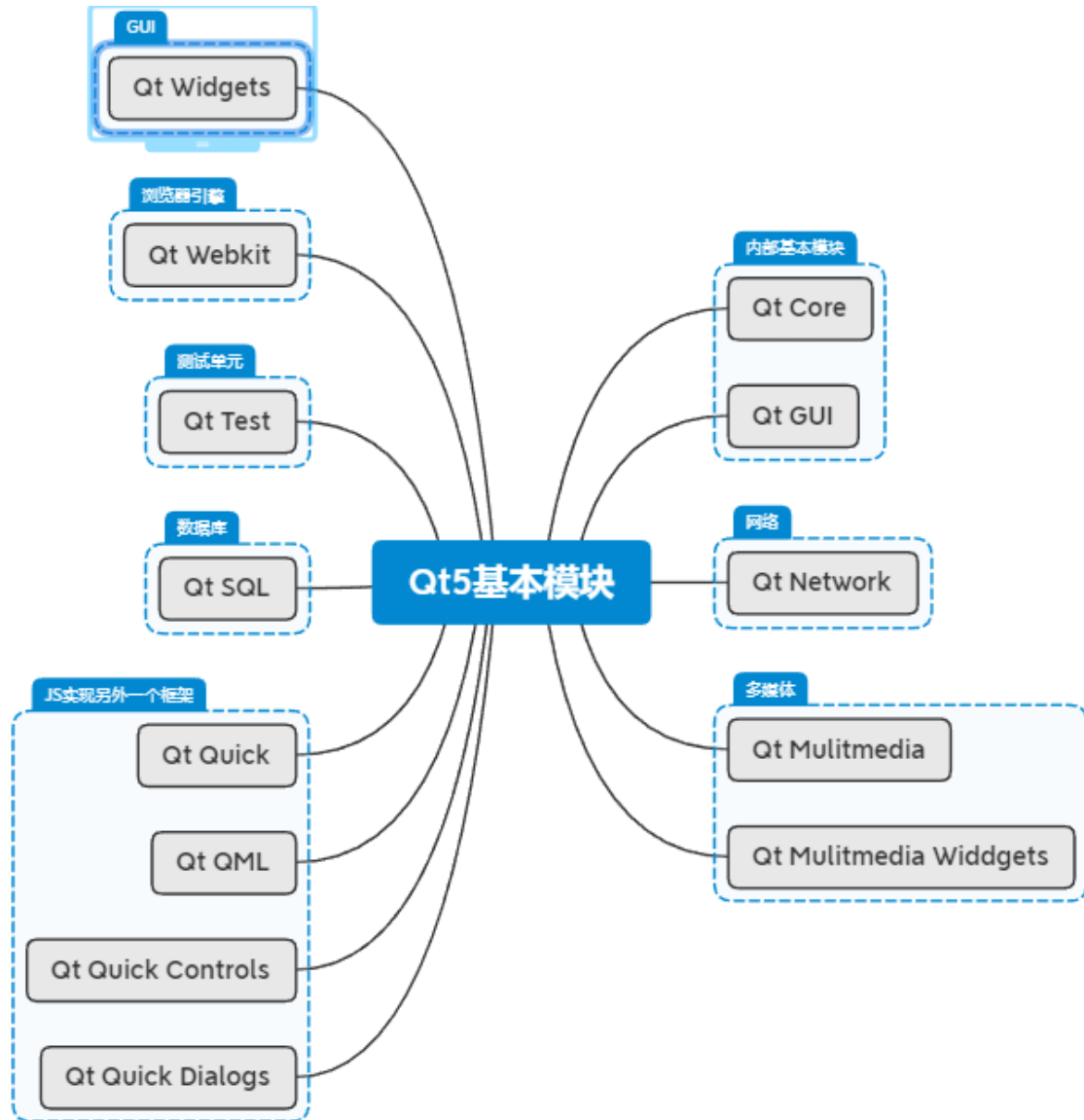
- TARGET后面对应的是应用程序的名字



可执行程序的文件夹可以删除，删除后，修改“TARGET”后的名字为“NickTest”，重新编译项目，可以看到应用程序的名字已经被改变了

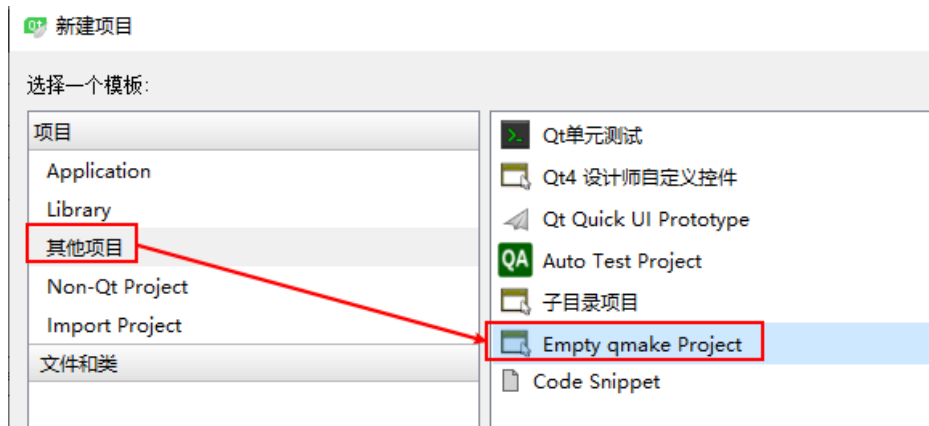
名称	修改日期	类型	大小
main.o	2020/7/18 16:19	O 文件	490 KB
mywidget.o	2020/7/18 16:19	O 文件	409 KB
NickTest	2020/7/18 16:19	应用程序	892 KB

2.3 QT基本模块

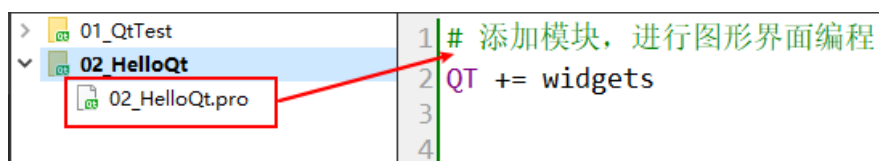


3. 第一个Qt程序

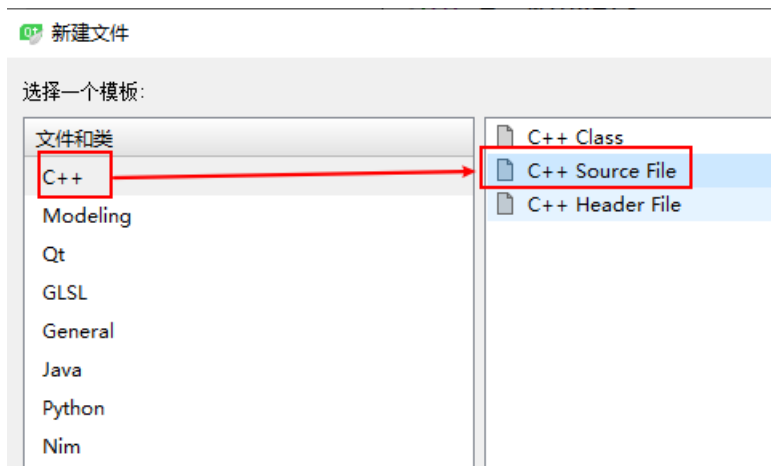
3.1 创建一个新的空项目



3.2 项目文件中添加模块



3.3 添加main文件



Location

名称:

路径:

1. 编写基本的框架

```
#include <QApplication>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    app.exec();
    return 0;
}
```

2. 创建一个窗口QWidget

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

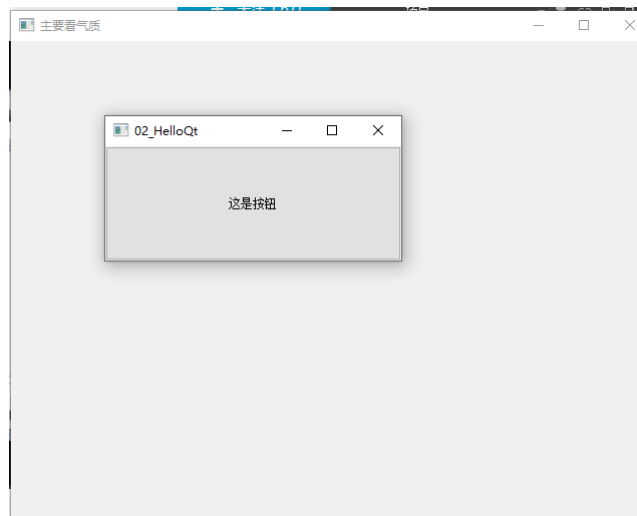
    // 创建窗口
    QWidget w;
    // 显示父对象
    w.show();

    app.exec();
    return 0;
}
```

3. 创建按钮QPushButton

```
// 窗口
QWidget w;
w.setWindowTitle("主要看气质"); // 设置标题
w.show();
// 按钮
QPushButton b;
b.setText("这是按钮");
b.show();
```

- 其结果为：



- 但是这并不是我们想要的效果，其原因是：

如果不指定**父对象**，对象和对象（窗口和窗口）之间是独立的。

指定父对象的方法有两种：

- (1) 通过setParent(指针)
- (2) 通过构造函数传参数

指定父对象的好处：

只需要显示父对象，上面的子对象自动显示

- 因此，修改代码为：

```
int main(int argc, char *argv[])
```



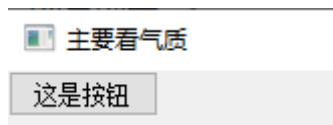
```

{
    QApplication app(argc, argv);
    QWidget w;
    w.setWindowTitle("主要看气质"); // 设置标题
    // 按钮
    QPushButton b;
    b.setText("这是按钮");
    b.setParent(&w);
    // 显示父对象
    w.show();

    app.exec();
    return 0;
}

```

- 修改后的效果为：



4. 移动button

```

QPushButton b;
b.setText("这是按钮");
b.setParent(&w); // 指定父对象
b.move(100, 100); // 移动坐标

```



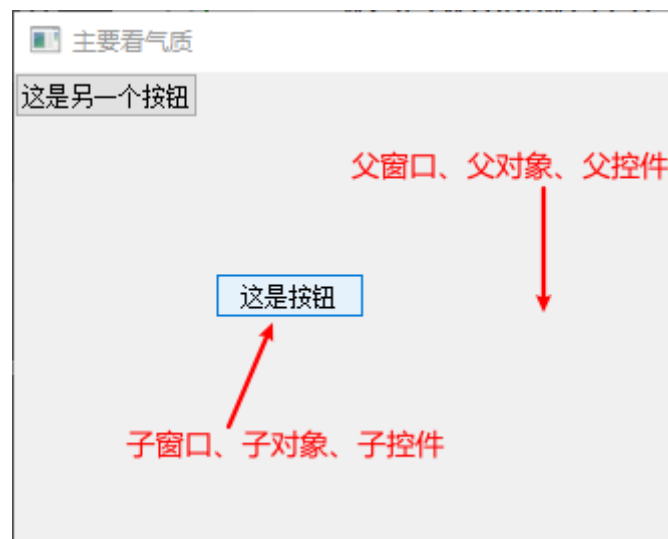
5. 通过构造函数创建子对象

```

// 按钮（通过构造函数传参指定父对象）
QPushButton b1(&w);
b1.setText("这是另一个按钮");

```

- 其结果为：



4. 标准信号和槽

4.1 一个错误的示范

不要将控件的创建放入构造函数中：

```
#include "mainwindow.h"
#include <QPushButton>

MainWindow::MainWindow(QWidget *parent)
    : QWidget(parent)
{
    // 实现相关控件 错误示范
    QPushButton b;
    b.setParent(this);
    b.setText("这是按钮");
}

MainWindow::~MainWindow()
{
}
```

构造函数结束后
对象会被析构
导致最后窗口显示
不出来控件

否则将无法显示控件。应该在.h文件中声明控件，在构造函数中创建。

- 头文件

```

#ifndef MAINWIDGET_H
#define MAINWIDGET_H

#include <QWidget>
#include <QPushButton>

class MainWidget : public QWidget
{
    // Q_OBJECT // 信号与槽相关，这里可以不要

public:
    MainWidget(QWidget *parent = 0);
    ~MainWidget();

private:
    QPushButton b1;
    QPushButton *b2;
};

#endif // MAINWIDGET_H

```

- cpp文件

```

MainWidget::MainWidget(QWidget *parent)
    : QWidget(parent)
{
    // 实现相关控件
    b1.setParent(this);
    b1.setText("这是按钮");
    b1.move(100, 100);

    // 另一个按钮
    b2 = new QPushButton(this);
    b2->setText("另一个按钮");
}

```

4.2 查看一个类的信号

- 通过类名找到对应的信号
- 首先对QPushButton点击F1

Contents

[Properties](#)
[Public Functions](#)
[Reimplemented Public Functions](#)
[Public Slots](#)
[Protected Functions](#)
[Reimplemented Protected Functions](#)
[Detailed Description](#)

- 在Contents中并没有找到Signal
- 看QPushButton类介绍

QPushButton Class

The [QPushButton](#) widget provides a command button. [More...](#)

Header: `#include <QPushButton>`

qmake: `QT += widgets`

Inherits: [QAbstractButton](#)

Inherited By: [QCommandLinkButton](#)

- 发现该类继承于QAbstractButton, 点击继续查看

Contents

[Properties](#)
[Public Functions](#)
[Public Slots](#)
[Signals](#)
[Protected Functions](#)
[Reimplemented Protected Functions](#)
[Detailed Description](#)

- 在QAbstractButton中看到了Signal, 点击查看

Signals

void [clicked](#)(bool *checked* = false)

void [pressed](#)()

void [released](#)()

void [toggled](#)(bool *checked*)

- 3 signals inherited from [QWidget](#)
- 2 signals inherited from [QObject](#)

- 继续点击“pressed”信号

```
[signal] void QAbstractButton::pressed()
```

This signal is emitted when the button is pressed down.

See also [released\(\)](#) and [clicked\(\)](#).

- 可以看到`pressed`信号是在按钮被按下时发送 (emitted)

4.3 connect函数

- 知道了信号后，通过`connect`函数来进行信号与槽的连接

```
/*
 * connect函数参数说明：
 * (1) &b1: 信号发出者，指针类型
 * (2) &QPushButton::pressed: 处理的信号，&发送者类名::信号名字
 * (3) this: 信号接收者
 * (4) &MainWidget::close: 槽函数，信号处理函数，&接收者类名::槽函数名字
 */
connect(&b1, &QPushButton::pressed, this, &MainWidget::close);
```

4.4 自定义槽函数

1. h文件中添加槽函数定义

```
class MainWidget : public QWidget
{
    Q_OBJECT // 信号与槽相关，这里可以不要

public:
    MainWidget(QWidget *parent = 0);
    ~MainWidget();

    void myslot();
}
```

2. cpp文件中实现函数

```
connect(b2, &QPushButton::released, this, &MainWidget::myslot);

// 槽函数，实现将按钮文本更换的功能
void MainWidget::myslot()
{
    b2->setText("123");
}
```

- 现在是窗口来接受信号
- 接下来使用另一个按钮来接受信号

```
// 按钮抬起的时候将另一个按钮进行隐藏
connect(b2, &QPushButton::released, &b1, &QPushButton::hide);
```

- 将信号可以理解为短信，而槽函数就是接受短信的手机，一条短信可以发送给不同的手机

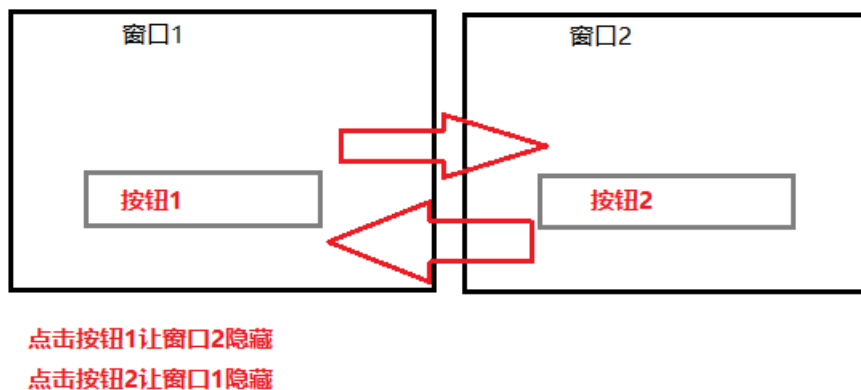
```
connect(b2, &QPushButton::pressed, this, &MainWidget::myslot);
// 按钮抬起的时候将另一个按钮进行隐藏
connect(b2, &QPushButton::released, &b1, &QPushButton::hide);

/*
 * 信号：短信
 * 槽函数：接受短信的手机
 */
```

信号就好比短信
将短信可以发送给不同的人

4.5 两个独立窗口

1. 创建两个独立的窗口



2. 新建按钮，并设置相关属性

- 头文件添加一个按钮

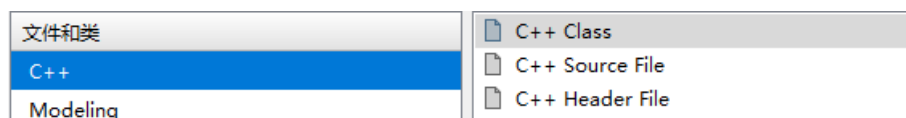
```
private:
    QPushButton b1;
    QPushButton *b2;
    QPushButton b3;
```

- 设置该按钮的相关属性

```
this->setWindowTitle("老大");
b3.setParent(this);
b3.setText("切换到子窗口");
b3.move(50, 50);
```

3. 添加新窗口

- 右击项目新建一个c++类



- 新建类名为“SubWidget”
- 然后在其头文件中添加按钮

```
class SubWidget : publi
{
    Q_OBJECT
public:
    explicit SubWidget(

signals:

public slots:

private:
    QPushButton b;
};
```

- 在其cpp文件构造函数中设置按钮父窗口

```
this->setWindowTitle("小弟");
b.setParent(this);
b.setText("切换到主窗口");
```



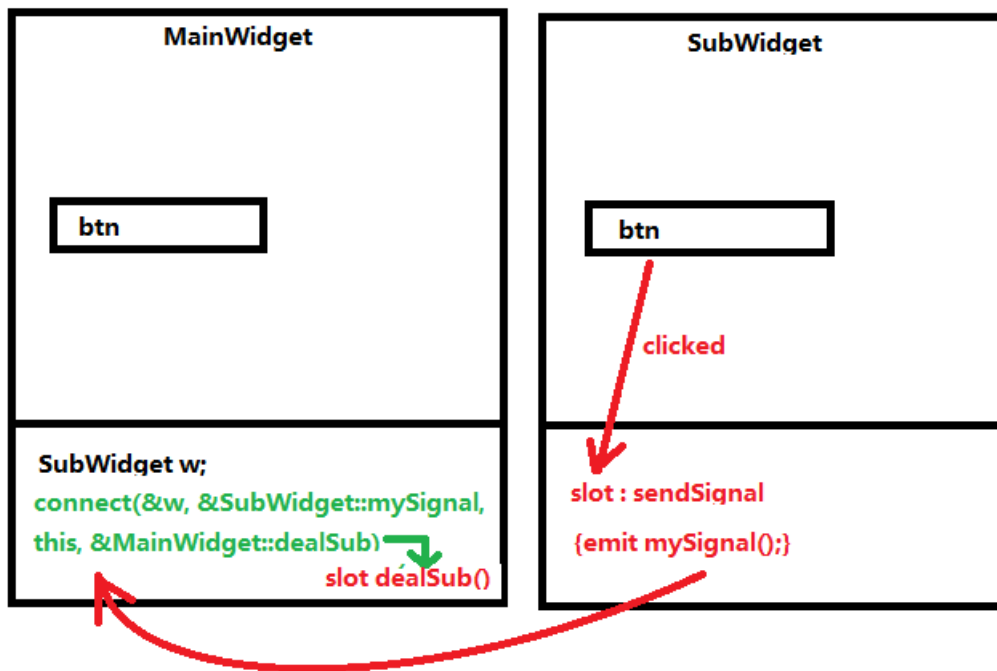
- 开始时显示主窗口隐藏子窗口
- 4. 点击主窗口按钮，隐藏自己，显示子窗口

```
connect(&b3, &QPushButton::released, this, &MainWidget::changeWin);
```

```
// 槽函数，实现点击按钮后隐藏主窗口，显示子窗口
void MainWidget::changeWin()
{
    // 子窗口显示
    w.show();

    // 本窗口隐藏
    this->hide();
}
```

- 5. 点击子窗口按钮，隐藏自己，显示父窗口
- 因为子窗口没有父窗口的指针，因此不能直接另父窗口显示
- 解决办法：



- 子窗口类定义

```
class SubWidget : public QWidget
{
    Q_OBJECT
public:
    explicit SubWidget(QWidget *parent = nullptr);

    /*
     * 信号必须有这个关键字来声明
     * 信号没有返回值，但可以有参数
     * 信号就是函数的声明，只需要声明，不需要定义
     * 使用: emit mySignal();
     */
signals:
    void mySignal();

public slots:
    void sendSlot();

private:
    QPushButton b;
};
```

- 主窗口创建连接

```
// 处理子窗口的信号
connect(&w, &SubWidget::mySignal, this, &MainWidget::dealSub);
```

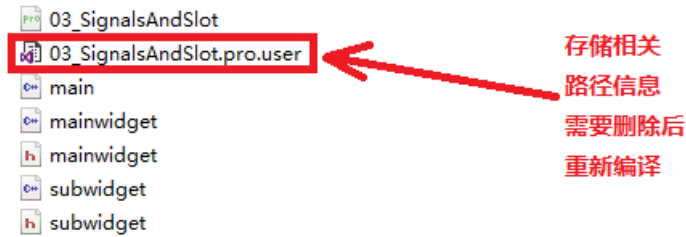
- 槽函数处理


```
void MainWidget::dealSub()
{
    // 隐藏子窗口
    w.hide();

    // 显示主窗口
    this->show();
}
```

5. 一个小问题

- 将项目用另一台机器打开或者放在别的路径打开后编译会出现问题



4.6 带参数的信号

1. 信号的重载

- 信号是可以重载的

```
signals:
    void mySignal();

    // 带参数的信号
    void mySignal(int, QString);
```

- 然后在之前的槽函数中发送两个信号，一个带参数，一个不带参数

```
// 槽函数
void SubWidget::sendSlot()
{
    emit mySignal();
    emit mySignal(250, "我是子窗口");
}
```

- 但是，信号重载之后会出现一个问题，之前的connect函数会出现编译错误

```
// 处理子窗口的信号
connect(&w, &SubWidget::mySignal, this, &MainWidget::dealSub);
```

指的是有参还是无参？

- 因此为了解决重载的信号无法区分，需要使用函数指针

函数：void fun(int) {//...}

函数指针：void (*p)(int) = fun;

```
void (SubWidget::*funSignal)() = &SubWidget::mySignal;
connect(&w, funSignal, this, &MainWidget::dealSub);
```

- 然后处理重载信号的槽函数

```
// 槽函数，处理带参数信号
void MainWindow::dealSlot(int num, QString str)
{
    qDebug() << num << " " << str;
}
// 需包含#include <QDebug>
```

- 问题：如果输出的int没问题，但是str乱码，还需要将其转成UTF-8

```
str.toUtf8().data();
```

↓
↓
 QByteArray char*

QByteArray QString::toUtf8() const

2. Qt4风格的信号和槽函数

- 信号使用宏SIGNAL

```
// Qt4信号连接
// Qt4槽函数必须要有slots关键字来修饰，否则不可使用
connect(&w, SIGNAL(my), this, SLOT())
// 使用宏
resize(400, 300);
// mySignal()
// mySignal(int,QString)
```

void mySignal()

可以直接选择是否带参数

- 槽函数使用宏SLOT

```
// Qt4信号连接
// Qt4槽函数必须要有slots关键字来修饰，否则不可使用
connect(&w, SIGNAL(mySignal()), this, SLOT(dea))
// dealSlot(int,QString)
// dealSub()
```

- 但是此处的槽函数必须要在类声明中使用关键字public || private slots:来修饰

```
// Qt4风格，Qt5可以直接将函数作为public就行
public slots:
    void myslot();
    void changeWin();
    void dealSub();
    void dealSlot(int, QString);
```

3. 问题

- 为什么SIGNAL和SLOT宏这么好用，Qt5缺不再继续使用？

原因：

- (1) 宏是将函数名字转换成字符串，而不进行错误检查
- (2) 槽函数必须要使用slots关键字来修饰

例如：信号名字如果拼写错了，在Qt5的风格上直接就会显示编译错误

而Qt4的方式编译不会报错，只有在运行的时候才会提示，因为使用宏只要是字符串就行

4.7 Lamda表达式与信号的功能探究

1. 因为Lambda是c++11的新特性，Qt与信号结合使用很方便，需要在项目文件中添加CONFIG

```
QT += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = 03_SignalsAndSlot
TEMPLATE = app

SOURCES += \
    main.cpp \
    mainwidget.cpp \
    subwidget.cpp

HEADERS += \
    mainwidget.h \
    subwidget.h

CONFIG += c++11
```

2. 使用Lambda表达式

```
int testNum = 100;
connect(b4, &QPushButton::released,
    // lambda表达式
    // []表示把外部变量传进来，不是函数传参
    []()
    {
        qDebug() << "这是Lambda表达式测试";
    }
);
```

Lambda表达式

- 方便，不用写接收者和槽函数

3. 使用Lambda表达式修改按钮的文本

```
int testNum = 100;
connect(b4, &QPushButton::released,
    // lambda表达式
    // []表示把外部变量传进来，不是函数传参
    []()
    {
        b4->setText("修改按钮文本");
        qDebug() << "这是Lambda表达式测试";
    }
);
```

- 编译会报错，其原因是表达式不能够识别b4这个控件
- 解决办法是将b4通过[]进行值传递

```

connect(b4, &QPushButton::released,
        // lamda表达式
        // []表示把外部变量传进来，不是函数传参
        [b4]()
        {
            b4->setText("修改按钮文本");
            qDebug() << "这是Lamda表达式测试";
        }
);

```

- 需要传递多个值时使用逗号分隔，但是如果传入的值十分多，这种方式太过于复杂，使用 [=] 表示将所有的局部变量、类的成员函数以值传递的方式进行传递

```

int testNum = 100; // 局部变量
connect(b4, &QPushButton::released,
        // lamda表达式
        // []表示把外部变量传进来，不是函数传参
        [=]() {
            b4->setText("修改按钮文本");
            qDebug() << "这是Lamda表达式测试";
            qDebug() << "局部变量testNum值为:" << testNum;
        }
);

```

可以访问

- 但是，由于是使用值传递，因此系统认为这些传进来的值是 read-only 的，如果想要改变其值，需要使用 mutable

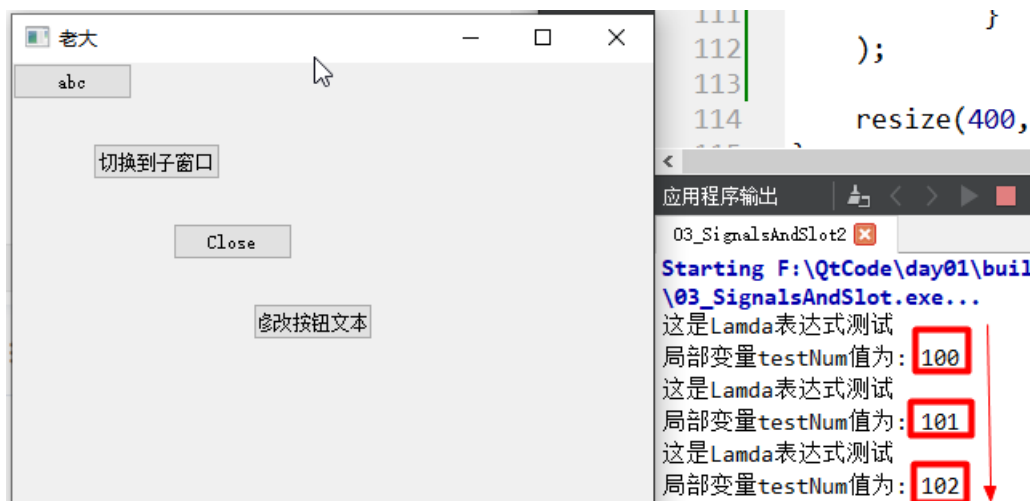
```

int testNum = 100;
connect(b4, &QPushButton::released,
        // lamda表达式
        // []表示把外部变量传进来，不是函数传参
        [=]() mutable {
            b4->setText("修改按钮文本");
            qDebug() << "这是Lamda表达式测试";
            qDebug() << "局部变量testNum值为:" << testNum;
            testNum++;
        }
);

```

可以对传入的局部变量进行修改

- 多次点击按钮，可以在控制台看到输出的局部变量值已经修改了

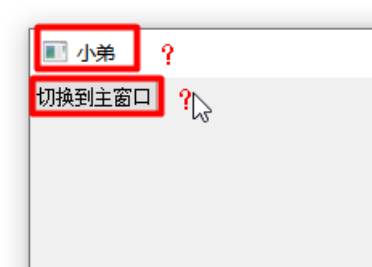


4. 如果信号带有参数

```
// 信号有参数
connect(b4, &QPushButton::clicked, 有一个bool型参数,
// lambda表达式 参数有默认值
// []表示把外部变量传进来, 不是函数传参
// [=]把外部所有局部变量
// 但因为是值传递, 这个参数是只读
// [this]:类中所有成员以值传递方式传进来
// [&]: 把外部所有局部变量以引用的方式传递进来
[=](bool isChecked) 需要同类型参数
{
    qDebug() << isChecked;
});
```

[signal] **void QAbstractButton::clicked(bool checked = false)**

5. 一个问题



- 当点击按钮“切换到主窗口”后，信号是窗口发的，还是按钮发的？
- 看具体代码可知，按钮只是出发槽函数，槽函数发送了信号，而槽函数是属于窗口的，因此是窗口发送了信号

```
connect(&b, &QPushButton::clicked, this, &SubWidget::sendSlot);
resize(400, 300);
```

Question:这个信号是谁发的? 是按钮还是窗口?

```
// 槽函数
void SubWidget::sendSlot()
{
    emit mySignal();
    emit mySignal(250, "我是子窗口");
}
```

槽函数是属于窗口的, 因此是窗口发送的信号, 而不是按钮

按钮只是触发了槽函数

槽函数最终发送了信号

- 另一个例子, 点击“Close”按钮后窗口会关闭, 关闭这个功能其实和按钮一点关系都没有。因为按钮被点击之后, 会调用一个回调函数 (软件中断), 至于回调函数做什么按钮就不知道了。

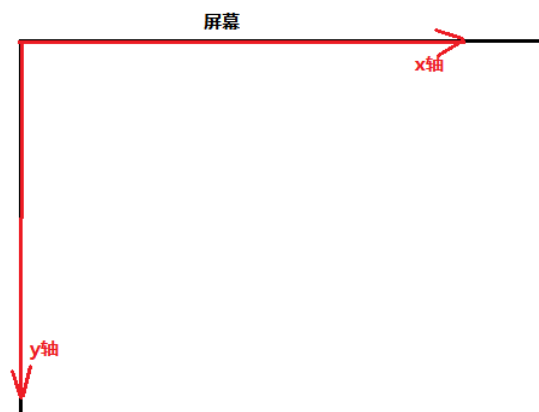
4.8 坐标系

1. 新创建一个项目

- 名字: 04_QtCoordinate
- 生成类: Mywidget

2. 在类的构造函数中使用move函数来观察坐标系

```
MyWidget::MyWidget(QWidget *parent)
: QWidget(parent)
{
    /*
     * 对于父窗口（主窗口），坐标系相对于屏幕
     * 原点：相对于屏幕左上角
     * x轴：往右递增
     * y轴：往下递增
     */
    move(0, 0);
}
```

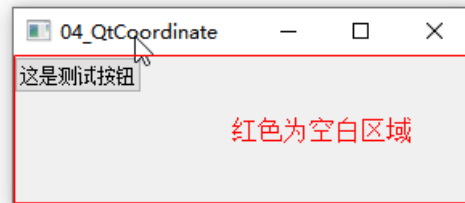


3. 子窗口的位置

```

/*
 * 子窗口，坐标系统相对于父窗口
 * 原点：窗口的空白区域，不包括边框
 * x轴：往右递增
 * y轴：往下递增
 */
QPushButton *b1 = new QPushButton(this);
b1->move(0, 0);
b1->setText("这是测试按钮");

```



4. 再创建一个按钮，并指定其父对象为b1

```

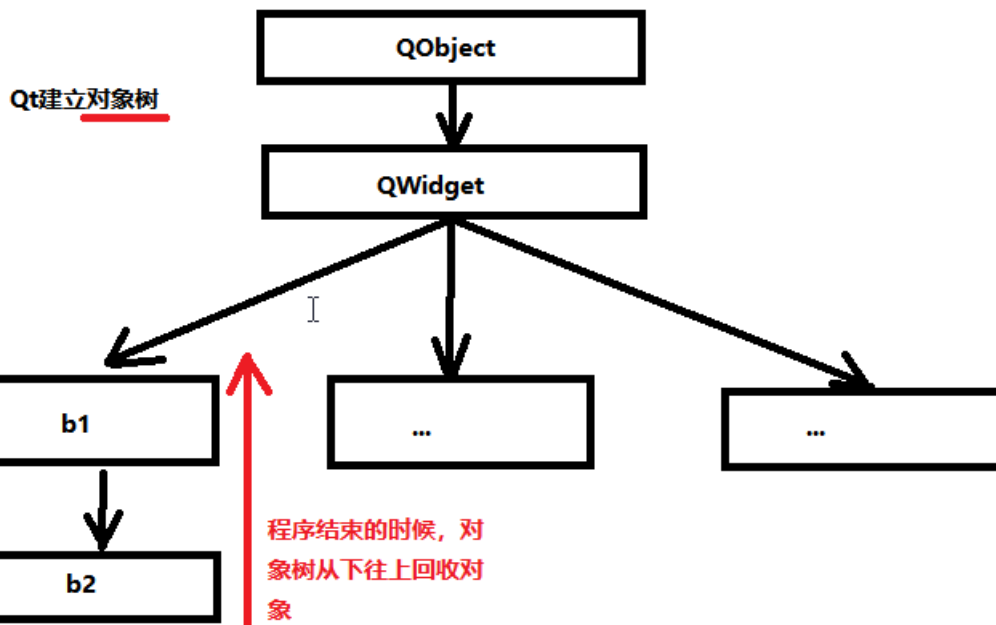
// 在新建一个按钮，指定其父对象为b1
QPushButton *b2 = new QPushButton(b1);
b2->move(10, 10);
b2->setText("按钮中的按钮");

```

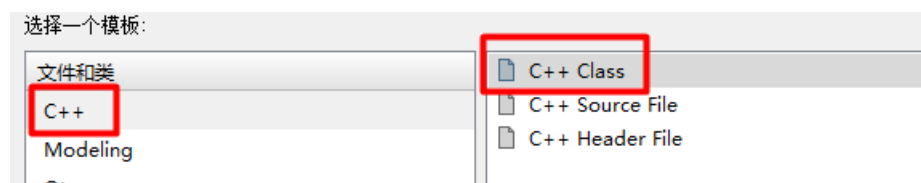
- 按钮b2的坐标系统是相对于父窗口（b1按钮）而言的



4.9 内存回收机制



1. 为了验证Qt对象树的机制，需要在析构函数中输出信息，看系统会不会自动调用析构函数
2. 但是QPushButton并不是我们自己写的，不能在其析构函数中输出，因此自己新建一个c++类



- 其名字为MyButton
- 3. 将其父类更改为QPushButton
- h头文件中

```

#include <QPushButton>

class MyButton : public QPushButton
{
    Q_OBJECT
public:
    explicit MyButton(QWidget *parent = nullptr);

```

- cpp构造函数中

```

MyButton::MyButton(QWidget *parent) : QPushButton(parent)
{
}

```

4. 为MyButton添加一个析构函数

```

class MyButton : public QPushButton
{
    Q_OBJECT
public:
    explicit MyButton(QWidget *parent = nullptr);
    ~MyButton();
}

```

- 在析构函数中进行信息输出


```
MyButton::~MyButton()
{
    qDebug() << "按钮被析构";
}
```

- 当窗口被关闭时，可以看到此条信息

Starting F:\QtCode\day01\build-04_QtCoordinate-Desktop_Qt_5_9_0_MinGW_32bit-Debug\debug\04_QtCoordinate.exe...

按钮被析构

F:\QtCode\day01\build-04_QtCoordinate-Desktop_Qt_5_9_0_MinGW_32bit-Debug\debug\04_QtCoordinate.exe exited

4.10 菜单栏和工具栏

1. 定义



2. 新建项目

类信息

指定您要创建的源码文件的基本类信息。

类名(C):	MainWindow
基类(B):	QMainWindow
头文件(H):	mainwindow.h
源文件(S):	mainwindow.cpp
创建界面(G):	<input type="checkbox"/> 取消
界面文件(F):	mainwindow.ui

PC端带菜单栏一般都用这个

3. 查看QMainWindow的帮助文档

- 文档中提供很多公有函数

Public Functions

```
QMainWindow(QWidget *parent = Q_NULLPTR,  
Qt::WindowFlags flags = Qt::WindowFlags())  
  
~QMainWindow()  
  
void addDockWidget(Qt::DockWidgetArea area, QDockWidget  
*dockwidget)  
void addDockWidget(Qt::DockWidgetArea area, QDockWidget  
*dockwidget, Qt::Orientation orientation)  
void addToolBar(Qt::ToolBarArea area, QToolBar *toolbar)  
void addToolBar(QToolBar *toolbar)  
QToolBar * addToolBar(const QString &title)  
void addToolBarBreak(Qt::ToolBarArea area =  
Qt::TopToolBarArea)  
  
QWidget * centralWidget() const  
Qt::DockWidgetArea corner(Qt::Corner corner) const  
virtual QMenu * createPopupMenu()  
DockOptions dockOptions() const  
Qt::DockWidgetArea dockWidgetArea(QDockWidget *dockwidget) const  
  
QMenuBar * menuBar() const  
QStatusBar * statusBar() const
```

添加浮动窗口 ←

添加工具栏 ←

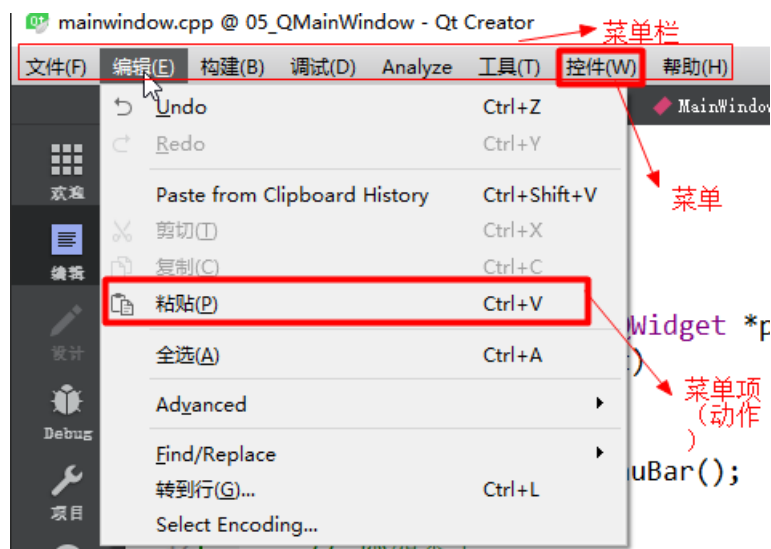
菜单栏

状态栏

4. 创建菜单栏

```
// 菜单栏  
QMenuBar *mBar = QMainWindow::menuBar();  
  
// 添加菜单  
QMenu *pFile = mBar->addMenu("文件");  
  
// 添加菜单项，添加动作  
// QAction* OMenu::addAction()  
QAction *pNew = pFile->addAction("新建");
```

注意区分：菜单栏、菜单
和菜单项（动作）



- 5. 查看帮助文档 `QAction` 的信号

Signals

```
void changed()
void hovered()
void toggled(bool checked)
void triggered(bool checked = false)
```

- 使用Lambda表达式来处理该信号

```
// 添加菜单项，添加动作
// QAction* QMenu::addAction()
QAction *pNew = pFile->addAction("新建");

connect(pNew, &QAction::triggered,
        [=]() mutable
        {
            qDebug() << "新建被按下";
        }
        );
```

- 点击“新建”后会看到输出信息

```
Starting F:\QtCode\day01\build-05_QMainWindow-Desktop_Qt_5_9_0_MinGW_32bit-Debug
\debug\05_QMainWindow.exe...
新建被按下
F:\QtCode\day01\build-05_QMainWindow-Desktop_Qt_5_9_0_MinGW_32bit-Debug\debug
```

- 添加分隔符

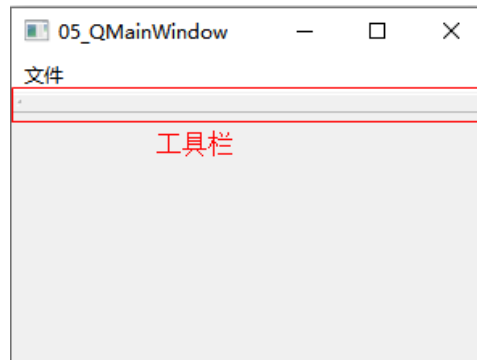
```
// 分割线
pFile->addSeparator();
```

- 5. 创建工具栏

- 工具栏实际是菜单栏的快捷键

```
// 工具栏，菜单项的快捷方式
QToolBar *toolBar = addToolBar("Tool Bar");
```

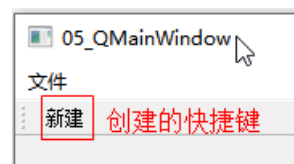
- 添加后会有一个空的工具栏



- 创建“新建”的工具栏，直接将菜单栏的指定按钮的Action指针作为参数

```
// 工具栏添加快捷键（动作）
// 把之前菜单栏的动作的指针直接拿过来就行
toolBar->addAction(pNew);
```

- 可以看到创建的工具栏已经出来了



- 点击该工具栏，会输出菜单栏新建的相应信息

```
Starting F:\QtCode\day01\build-05_QMainWindow-Desktop_Qt_5_9_0_MinGW_32bit-Debug
\debug\05_QMainWindow.exe...
新建被按下
F:\QtCode\day01\build-05_QMainWindow-Desktop_Qt_5_9_0_MinGW_32bit-Debug\debug
```

- 再添加一个按钮工具栏，点击后更改按钮的文本

```
// 添加一个按钮工具栏
QPushButton *b = new QPushButton(this);
b->setText("另一个工具栏选项");
toolBar->addWidget(b);

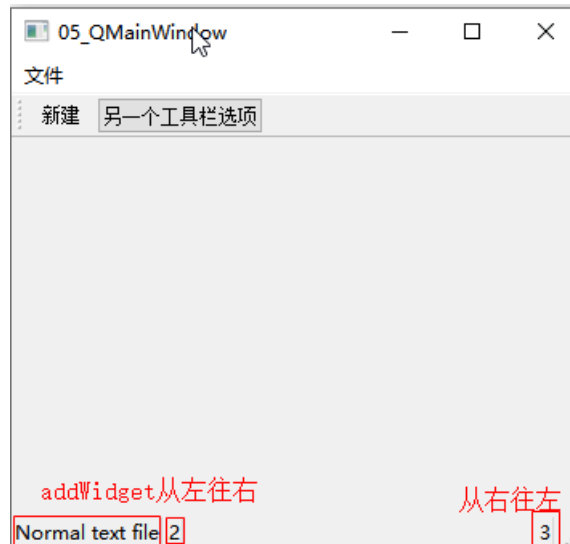
connect(b, &QPushButton::clicked,
        [=]() mutable
        {
            b->setText("工具栏被点击");
        }
        );
```

6. 添加状态栏

```

// 状态栏
QStatusBar *staBar = statusBar();
QLabel *label = new QLabel(this);
label->setText("Normal text file");
staBar->addWidget(label);
// addWidget从左往右依次添加
staBar->addWidget(new QLabel("2", this));
// addPermanentWidget从右往左添加
staBar->addPermanentWidget(new QLabel("3", this));

```



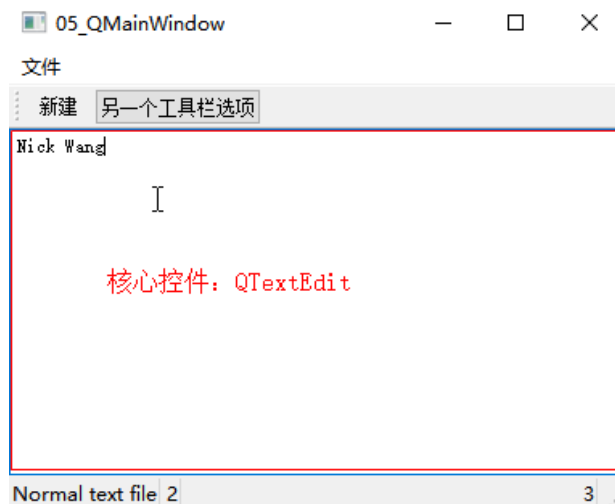
7. 核心控件

- 核心控件就是正中间的空间
- 核心控件添加一个文本编辑器QTextEdit

```

// 核心控件
QTextEdit *textEdit = new QTextEdit(this);
setCentralWidget(textEdit);

```



8. 浮动窗口

- 添加一个浮动窗口

```
// 浮动窗口
QDockWidget *dock = new QDockWidget(this);
addDockWidget(Qt::RightDockWidgetArea, dock);
QTextEdit *textEdit1 = new QTextEdit(this);
dock->setWidget(textEdit1);
```

- 分享一个小技巧

对于函数addDockWidget不知道参数的时候需要看帮助文档，但是对着函数按F1的时候会出现：

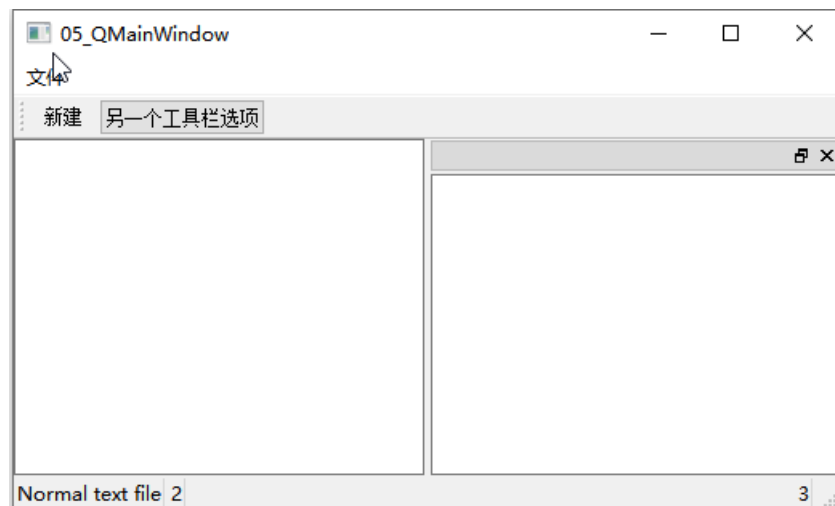
No documentation available.

此时，应该先将函数的参数补齐，例如addDockWidget(1, 2)，然后将光标指向函数名按F1

```
void QMainWindow::addDockWidget(Qt::DockWidgetArea area, QDockWidget *dockwidget)
```

可以看到第一个参数是表示dock的位置，这个参数是一个枚举类型，继续点击可以进行查看：

Constant	Value
Qt::LeftDockWidgetArea	0x1
Qt::RightDockWidgetArea	0x2
Qt::TopDockWidgetArea	0x4
Qt::BottomDockWidgetArea	0x8
Qt::AllDockWidgetAreas	DockWidgetArea_Mask
Qt::NoDockWidgetArea	0



4.11 模态和非模态对话框

1. 模态对话框

- 阻塞，执行的时候会一直等待用户操作，对话框关闭后才继续执行代码
- 使用exec调用

```

QMenuBar *mBar = menuBar();
QMenu *menu = mBar->addMenu("对话框");

QAction *p1 = menu->addAction("模态对话框");
connect(p1, &QAction::triggered,
        [=]() mutable
        {
            QDialog dlg;
            // exec执行到这里就不会动，等着用户操作
            dlg.exec();
            qDebug() << "这是在exec后一行的输出";
        }
);

```

- 直到关闭该对话框，才能看到qDebug()输出

2. 非模态对话框

- 不阻塞，程序继续向后执行
- 不可以用局部变量，否则对话框一闪而过
- 可以将QDialog声明为QMainWindow的数据成员
- 后续的输出不受影响

```

// 非模态对话框
QAction *p2 = menu->addAction("非模态对话框");
connect(p2, &QAction::triggered,
        [=]() mutable
        {
            // show方法来调用，不会阻塞
            dlg.show();
            qDebug() << "这是在show后一行的输出";
        }
);

```

- 也可以动态分配空间，但是**此方法不好**，因为程序结束的时候才能够析构，如果这个按钮是经常按的，在程序结束之前，不停的new空间，但是不释放，会占用空间。
- 但是并不代表动态分配空间不可以使用，需要设置一个属性。创建的时候**不指定父对象**。

```
void QWidget::setAttribute(Qt::WidgetAttribute attribute, bool on = true)
```

enum Qt::WidgetAttribute

This enum type is used to specify various special convenience functions which are i

Constant

Qt::WA_AcceptDrops

Qt::WA_AlwaysShowToolTips

Qt::WA_ContentsPropagated

Qt::WA_CustomWhatsThis

Qt::WA_DeleteOnClose

Qt::WA_Disabled

```
[=]() mutable
{
    // show方法来调用，不会阻塞
    dlg.show();
    qDebug() << "这是在show后一行的输出";

    QDialog *dlg2 = new QDialog;
    dlg2->setAttribute(Qt::WA_DeleteOnClose);
    dlg2->show();
}
```

4.11 标准对话框和文件对话框

1. 标准对话框QMessageBox

- 查看帮助文档

Contents

Public Types

Properties

Public Functions

Public Slots

Signals

Static Public Members

Reimplemented Protected Functions

Detailed Description

The Property-based API

The Static Functions API

Advanced Usage

Default and Escape Keys

QMessageBox Class

- 查看Static Public Members

Static Public Members

```
void about(QWidget *parent, const QString &title, const QString &text)
void aboutQt(QWidget *parent, const QString &title = QString())

StandardButton critical(QWidget *parent, const QString &title, const QString &text, StandardButtons buttons = Ok,
StandardButton defaultButton = NoButton)

StandardButton information(QWidget *parent, const QString &title, const QString &text, StandardButtons buttons = Ok,
StandardButton defaultButton = NoButton)

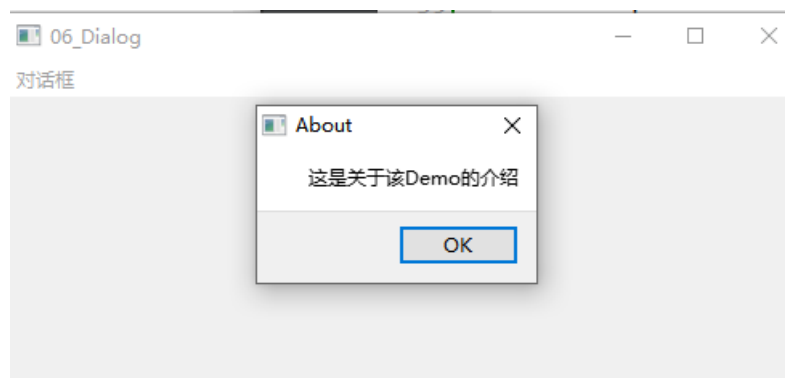
StandardButton question(QWidget *parent, const QString &title, const QString &text, StandardButtons buttons =
StandardButtons( Yes | No ), StandardButton defaultButton = NoButton)

StandardButton warning(QWidget *parent, const QString &title, const QString &text, StandardButtons buttons = Ok,
StandardButton defaultButton = NoButton)
```

2. 关于对话框

- 静态成员函数直接使用类名来调用

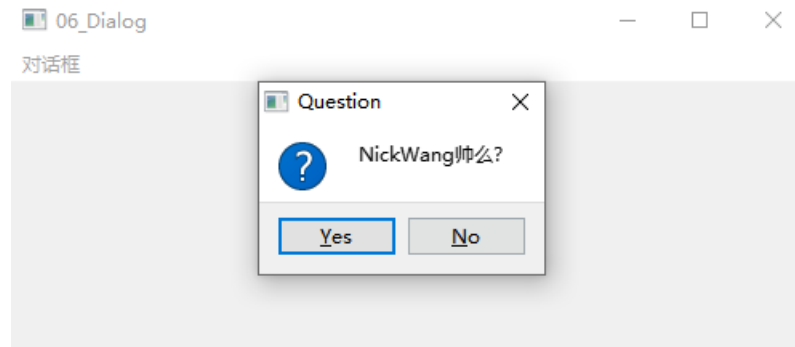
```
QAction *p3 = menu->addAction("关于对话框");
connect(p3, &QAction::triggered,
    [=]() mutable
    {
        QMessageBox::about(this, "About", "这是关于该Demo的介绍");
    }
);
```



3. 问题对话框

```
// 问题对话框
QAction *p4 = menu->addAction("问题对话框");
connect(p4, &QAction::triggered,
    [=]() mutable
    {
        int ret = QMessageBox::question(this, "Question", "NickWang帅么? ");

        switch (ret) {
        case QMessageBox::Yes:
            qDebug() << "有眼光呀兄弟";
            break;
        case QMessageBox::No:
            qDebug() << "亲, 这里建议您配一下眼镜";
            break;
        default:
            break;
        }
    }
);
```



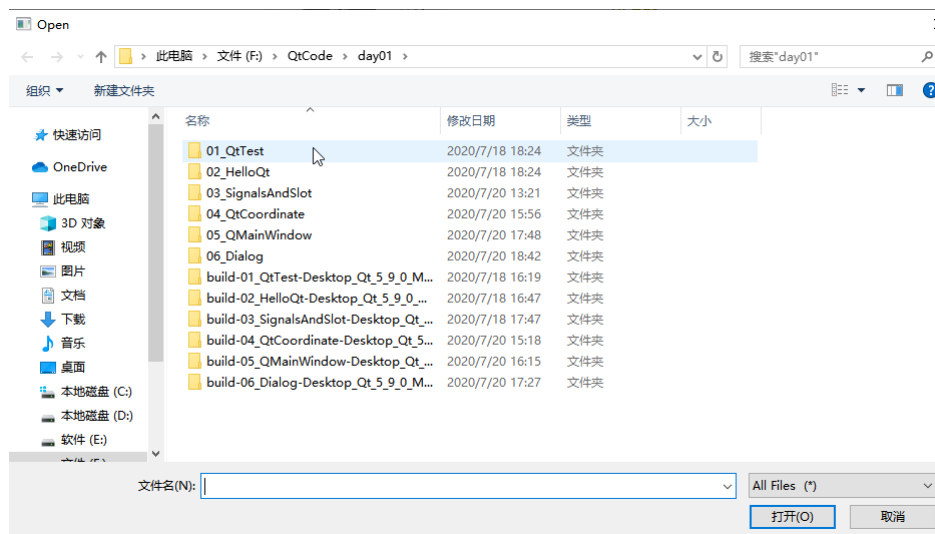
4. 文件对话框

- 代码

```
// 文件对话框
QAction *p5 = menu->addAction("文件对话框");
connect(p5, &QAction::triggered,
    [=]() mutable
    {
        QString path = QFileDialog::getOpenFileName(this, "Open", "../");
        qDebug() << path;
    }
);
```

父对象 标题 路径

- 结果



- 输出

Starting F:\QtCode\day01\build-06_Dialog-Desktop_Qt_5_9_0_MinGW_32bit-Debug\debug\06_Dialog.exe...
 "F:/QtCode/day01/06_Dialog/mainwindow.h"