

QT4.7 使用教程

作者：向仔州

VS2013+QT 软件安装.....	10
创建一个 QT 工程.....	12
VS2013 破解.....	14
Windows 下 qt-creator 安装.....	15
Qtcreator 和 VS2013 QT 冲突的问题.....	25
Linux 下 QT 的安装与搭建.....	25
创建一个主程序顶层窗口.....	31
QWidget	
设置窗口名称.....	31
Qwidget :: setWindowTitle //设置窗口名称	
设置窗口的最大化最小化栏，变成问号栏。.....	32
Qt :: Dialog //窗口初始化属性设置	
设置窗口贴死在桌面	32
Qt :: SplashScreen //窗口贴死在桌面就是启动软件时显示的 log	
设置父窗口初始化大小	33
QWidget :: resize //设置窗口大小	
把父窗口卡死在顶层	33
Qt :: Window Qt::WindowStaysOnTopHint //设置窗口卡死在顶层的属性其余窗口无法覆盖	
生成标题栏.....	34
Qlabel //标题栏类	
Qlabel :: setText //设置标题栏名称	
Qt 程序父窗口坐标及窗口大小获取.....	35
按钮组件(QPushButton)	36
QPushButton :: move //设置按钮在窗口位置	
QPushButton :: resize //设置按钮大小	
QPushButton :: setText //设置按钮名称	
文本框组件(LineEdit)	37
QLineEdit :: move //设置可以输入字符的文本框位置	
QLineEdit :: resize //文本框大小	
QT 消息处理方法，信号与槽.....	38
QObject :: connect //信号与槽链接函数	
emit //信号发送函数	
QT 的字符串类.....	45
QString	
QString :: append //在已有的字符串后面追加字符	
QString :: prepend //在已有的字符串前面追加字符	
QString :: replace //指定字符串替换.....	46

Sprintf 输出字符串.....	46
QString 寻找某个字符在字符串里面的位置.....	47
QString :: indexOf //从字符串中找出指定字符的位置	
QString 字符串截取	47
QString :: mid //窃取前面的字符串	
QString :: trimmed //去掉字符串前后空格	
QString 字符串指定部分段截取	48
QString :: section	
QString:: length 字符串长度计算.....	49
QStringList 字符串(列表)/字符串数组使用.....	49
QStringList :: append //在字符串数组末尾加入新字符串	
QStringList :: insert //指定字符串数组某个位置插入字符串	
QStringList :: replace //指定字符串数组某一组进行替换	
QStringList :: removeAll //删除字符串数组里面的某一组字符串	
QStringList :: join //取消字符串数组里面的标点符号	
QString :: split //用标点符号分割字符串	
QStringList :: indexOf //获取指定字符串在字符串数组的位置	
QStringList :: replaceInStrings //替换字符串数组某一段字符串	
QT 中不依赖父窗口的对话框	
QWidget::setWindowTitle //设置 widget 窗口名字	
QDialog::setWindowTitle //设置模态对话框窗口名字	
用 QDialog 来实现模态对话框，模态对话框就点击按钮后再弹出的对话框.....	53
模态对话框有使用后的返回值，非模态对话框没有返回值.....	57
QDialog :: exec(); //接受子程序的 done 返回值	
void QDialog :: done(int i); //子程序直接用 done 返回数字给主程序	
QDialog :: Accepted //这就是返回的值	
QDialog :: Rejected //这就是返回的值	
登录对话框编程思想.....	59
QLineEdit :: setEchoMode //设置输入对话框输入密码时显示还是不显示密码	
setWindowTitle	
setFixedSize	
done(Accepted)	
done(Rejected)	
QloginDialog::Accepted	
文本框组件(QLineEdit)里面输入的字符串获取.....	67
QLineEdit :: text() //获取输入编辑框的字符串	
QT 消息对话框实现.....	68
QMessageBox	
QMessageBox :: setIcon(QMessageBox::Information) //设置对话框图标形状	
QMessageBox :: setStandardButtons() //给对话框建立指定数量按钮	

QMessageBox :: Ok //OK 按钮
QMessageBox :: Cancel //退出按钮
QMessageBox :: YesToAll //yes 按钮

颜色对话框，设置窗口颜色.....70

QColorDialog :: setWindowTitle
QColorDialog :: setCurrentColor //颜色对话框初始颜色
QColorDialog :: setCurrentColor(Qt::red) //对话框初始颜色为红色
QColorDialog :: Accepted
QColorDialog :: selectedColor() //获取用户按钮选择的颜色
QColor :: red
QColor :: green
.....

输入对话框，就是你给对话框输入字符，然后代码可以获取字符75

QInputDialog
QInputDialog :: setWindowTitle
QInputDialog :: setLabelText //输入对话框显示的字符
QInputDialog :: setInputMode //规定用户输入的字符类型
QInputDialog :: IntInput //用户必须输入整形数字
QInputDialog :: Accepted
QInputDialog :: IntMinimum
QInputDialog :: IntMaximum
QInputDialog :: exec()
QInputDialog :: TextInput

字体设置对话框77

QFontDialog
QFont
QFont :: Bold //设置初始字体颜色
QFontDialog :: Accepted

进度条对话框.....79

QProgressDialog
QProgressDialog :: setWindowFilePath //进度条对话框名
QProgressDialog :: setLabelText //进度条对话框里面的提示信息
QProgressDialog :: setMinimum //设置进度条最小值
QProgressDialog :: setMaximum //设置进度条最大值
QProgressDialog :: setValue //设置进度条当前进度值

打印对话框，让你的软件有使用打印机的功能，或者直接输出 pdf 功能.....81

QPrintDialog
QPrintDialog::printer()
QTextDocument //建立一个文本文档需要的类

QTextDocument :: print //将建立的文本对象放入打印机	
QPrinter :: setOutputFileName //打印的 PDF 要存放的路径	
布局管理器解决窗口无法自适应问题.....	83
QVBoxLayout	
QVBoxLayout :: addWidget //将自己创建的控件对象放入布局管理器	
Widget :: setLayout //布局管理器管理的是哪个窗口的控件	
QPushButton :: setSizePolicy //让按钮水平或者垂直自动调整	
QVBoxLayout :: setSpacing //控件之间间距	
设置按钮初始化大小.....	84
QPushButton :: setMinimumSize //设置按钮初始化大小	
QSizePolicy :: Expanding	
横向布局管理器.....	86
QHBoxLayout	
QHBoxLayout :: addWidget //水平布局管理器设置控件	
垂直水平布局管理器相互嵌套使用思路.....	87
设置布局管理区不同按钮放大缩小比例自定义.....	88
QVBoxLayout :: setStretchFactor	
如何自定义布局管理器里嵌入的布局管理器窗口比例.....	89
矩阵格子方式的布局管理器用 Vboxlayout 和 Hboxlayout 嵌套方式太麻烦	
现在用专用的格子布局管理器.....	90
QGridLayout	
QGridLayout :: setColumnStretch //设置格子布局管理器的每个格子大小	
QGridLayout :: setRowStretch //设置格子布局管理器的每个格子大小	
表单布局管理器.....	93
QFormLayout	
QFormLayout :: setRowWrapPolicy	
QFormLayout :: WrapAllRows	
QFormLayout :: setLabelAlignment	
Qt :: AlignRight	
栈式布局管理器.....	95
QStackedLayout	
QStackedLayout :: setCurrentIndex	
setParent(..)	

计时器.....	97
QTimer	
QTimer :: start(..) //启动计时器计数	
布局管理器综合实验(重点， 多复习)	98
菜单栏程序创建, 用 QMainWindow 顶层窗口.....	107
QMenuBar	
QMenuBar :: addMenu //将下拉菜单栏放入顶层窗口行	
QMenu	
QMenu :: addAction //将菜单项放入新建的下拉菜单栏	
QMenu :: addSeparator	
QAction //创建菜单项	
QAction :: setShortcut //给菜单项配上一个快捷键	
QkeySequence //要填入快捷键宏的函数	
菜单栏界面制作的时候, 要多加一些判断语句, 已保正式工程项目安全.....	111
工具栏创建.....	113
QToolBar	
QToolBar :: setFloatable //菜单栏对象取消浮动功能	
QToolBar :: addAction //菜单栏对象取消浮动功能	
QToolBar :: setMovable //禁止工具栏在窗口浮动	
addToolBar //工具栏类	
QAction :: setToolTip //菜单项提示符	
QAction :: setIcon //给菜单项添加图片	
菜单栏和工具栏综合开发.....	119
QString :: length //计算字符串长度	
QString :: section	
状态栏创建.....	123
QStatusBar	
statusBar()	
QStatusBar :: addPermanentWidget //给状态栏添加各种各样控件	
QStatusBar :: showMessage //在状态栏左下角显示字符串	
单行文本和多行文本, 富文本使用.....	126
QPlainTextEdit	
QPlainTextEdit :: insertPlainText //获取 windows 上的图片给文本框显示	
QTextEdit	
QLineEdit :: insert //使用 QML 语言加入图片	
setCentralWidget //设置对象在窗口中心区域	

文件创建, 打开, 写入, 读取, 操作方法, 131

QFile

```
QFile :: open //打开文件  
QFile :: write //写文件  
QFile :: read//读文件  
QFile :: close //关闭文件  
QByteArray //字节数组用法请看 QByteArray 章节
```

QFileInfo

```
QFileInfo :: exists //文件是否存在, 存在返回 true  
QFileInfo :: isFile //判断是文件夹还是文件  
QFileInfo :: isReadable //文件是否可读  
QFileInfo :: isWritable //文件是否可写  
QFileInfo :: created //文件什么时候创建的  
QFileInfo :: lastRead //文件被访问时间  
QFileInfo :: lastModified //文件最后一次被修改时间  
QFileInfo :: path //文件路径  
QFileInfo :: fileName //文件名字  
QFileInfo :: suffix //文件后缀  
QFileInfo :: size //文件大小
```

临时文件操作类进行进程之间通信, 传数据..... 134

QTemporaryFile

```
QTemporaryFile :: open  
QTemporaryFile :: write  
QTemporaryFile :: close
```

文本流数据流读写 IO 设备数据..... 135

QIODevice :: ReadOnly

```
QString :: toStdString().c_str() //将字符串转换成指定类型
```

QTextStream

```
QTextStream :: readLine //读取文本每行字符串  
QTextStream :: atEnd //判断是不是读取文本的最后一行  
QDataStream  
QDataStream :: setVersion//写入版本  
QDataStream :: Qt_4_7
```

QT 缓冲区变量, 用于多线程数据通信..... 140

QBuffer

QT 文件夹创建, 删除, 重命名操作..... 141

QDir

```
QDir :: exists //判断指定路径的文件夹是否存在  
QDir :: mkdir //创建目录  
QDir :: cd //进入目录
```

用于监控指定文件，目录内容变化的类 QFileSystemWatcher 144

QFileSystemWatcher :: addPath //监控指定路径文件是否变化

用界面窗口读取文本数据显示。写好的文本用界面进行存储..... 148

QFileDialog
QFileDialog :: setAcceptMode //选择什么方式打开文件
QFileDialog :: AcceptOpen
QFileDialog :: ExistingFile
QFileDialog :: set FileMode
QFileDialog :: setWindowTitle
QFileDialog :: selectedFiles
QFileDialog :: Accepted
toPlainText

QMap 使用，键值对概念..... 159

QMap
 QMap<类型 1, 类型 2> == QMap<QString,int> //定义一个 QMap 模板对象
 QList<QString>
 QMap.insert //向 QMap 对象写入键值
 QMapIterator<类型 1, 类型 2>

QHash 使用，哈希表使用..... 161

QHash<类型 1, 类型 2>
 QHash :: insert //向表里面放数据
 QHash :: keys //取出哈希表里面的 key
 QHash :: values //取出哈希表里面的值
 QHash :: begin
 QHash :: constEnd
 QFileDialog :: AcceptSave

QT 事件类型，就是信号触发的事件，只是我们用对象本身的事件函数..... 166

event(QEvent* e)
 QEvent :: KeyPress //返回当前按键状态
 拖放事件案例
 setAcceptDrops //让当前对象或窗口接受拖放事件
 QdragEnterEvent //拖拽事件发生自动执行的函数
 QDragEnterEvent :: acceptProposedAction //支持用户在窗口部件上拖放对象
 QDragEnterEvent :: ignore
 QDropEvent::mimeData :: hasUrls //过去 URL 列表
 QList :: toLocalFile //获取文件路径
 QDropEvent :: ignore

编辑器窗口点击按钮实现复制，粘贴，撤回.....171

槽函数功能 appendHtml

appendPlainText
centerCursor
clear
copy
cut
insertPlainText
paste
redo
selectAll
setPlainText
undo

文本编辑器编辑区域的打印功能实现.....172

QplainTextEdit

QTextDocument

QT 查找对话框实现.....174

QCheckBox 复选框

QRadioButton 单选按钮

QSharedPointer 智能指针类

关于对话框设计，就是现实版本信息和一些废话的对话框.....177

QLabel :: setPixmap 对话框里的 png 图片转换成位图

QPixmap

多窗口切换页面.....180

QTabWidget

QTabWidget :: addTab //多页面框增加 tab 项
QTabWidget :: setTabPosition //设置 tab 栏位置
QTabWidget :: South
QTabWidget :: East
QTabWidget :: West
QTabWidget :: North
QTabWidget :: setTabShape //设置 tab 样式
QTabWidget :: Triangular
QTabWidget :: setTabsClosable

QT 表格界面设计.....185

QTabWidget :: setRowCount //设置表格行
QTabWidget :: setColumnCount //设置表格列
QTabWidget :: setAlternatingRowColors //设置表格隔行变色

QTabWidget :: setSelectionBehavior //鼠标选择时表格状态	
QAbstractItemView :: SelectRows	
QTabWidget :: currentRow	
QTabWidget :: insertRow	
QTabWidget :: removeRow	
QTabWidget :: item	
QTabWidget :: showRow	
QTabWidget :: hideRow	
列表实现.....	192
QTreeWidget	
QTreeWidget :: setHeaderLabel	
QTreeWidgetItem	
有个重要的错误 collect2:ld returned 1 exit status 解决方法.....	196
模型视图设计模式方法.....	197
QFileSystemModel	
QFileSystemModel::setRootPath	
QTreeView :: setModel	
QTreeView :: setRootIndex	
用表格案例来解释模式视图的基本作用.....	199
QStandardItemModel	
QStandardItemModel :: setData	
QStandardItemModel :: invisibleRootItem	
QStandardItemModel :: setChild	
QStandardItemModel :: index	
模型视图委托机制，就是界面上修改的参数自动传入模型类.....	204
QModelIndex	
QStyledItemDelegate :: createEditor	
QStyledItemDelegate :: updateEditorGeometry	
QStyledItemDelegate :: setEditorData	
QStyledItemDelegate :: setModelData	
我们想在修改表格数据饿时候，自动就把表格对应底层的模型数据修改了，怎么做呢？那就要自定义委托类，其实就是要重写<QStyledItemDelegate>里面的几个函数.....	208
QByteArray 使用.....	208
QT Creator 制作可执行程序在其它 windows 电脑上运行.....	210
把 windows 的 QT 工程复制粘贴到 linux 下编译使用遇到的问题.....	214
Qt Creator 设置工程的一些字体大小和其它属性.....	216

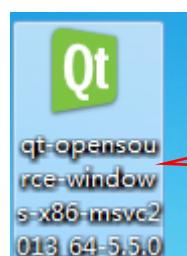
VS2013+QT 软件安装

下载 VS2013 不停的下一步下一步安装
然后还要下载另外两个软件



去下载 QT5 在 VS 的插件 http://download.qt.io/official_releases/vsaddin/qt-vs-addin-1.2.5.exe

这样你在 VS2013 打开后可以看到 QT5 的标志



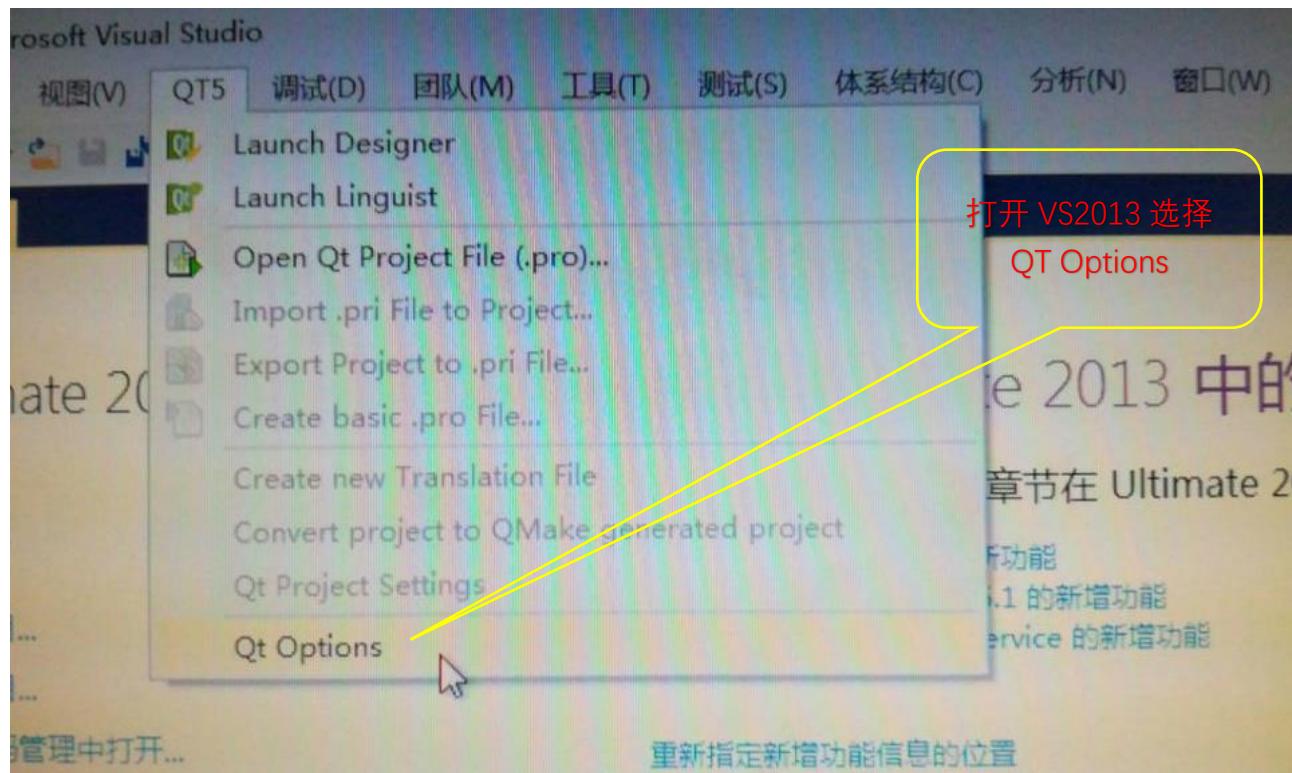
还要安装 MSVC2013 软件，否则在 VS 配置 QT 路径时
找不到 MSVC，建议安装 MSVC 时指定路径

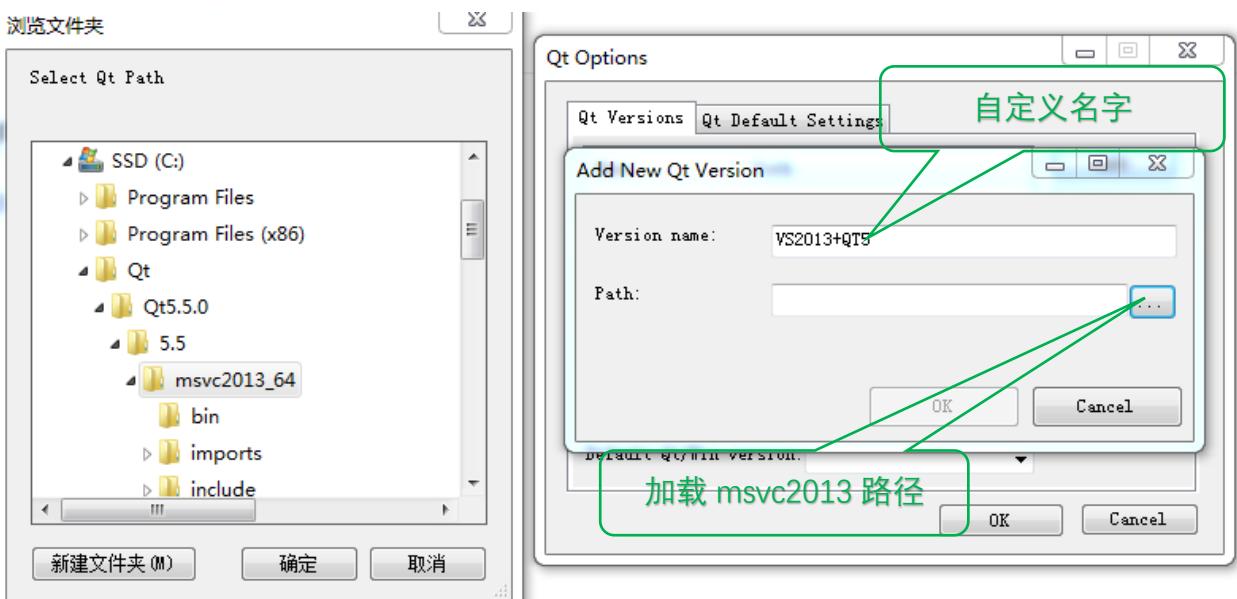
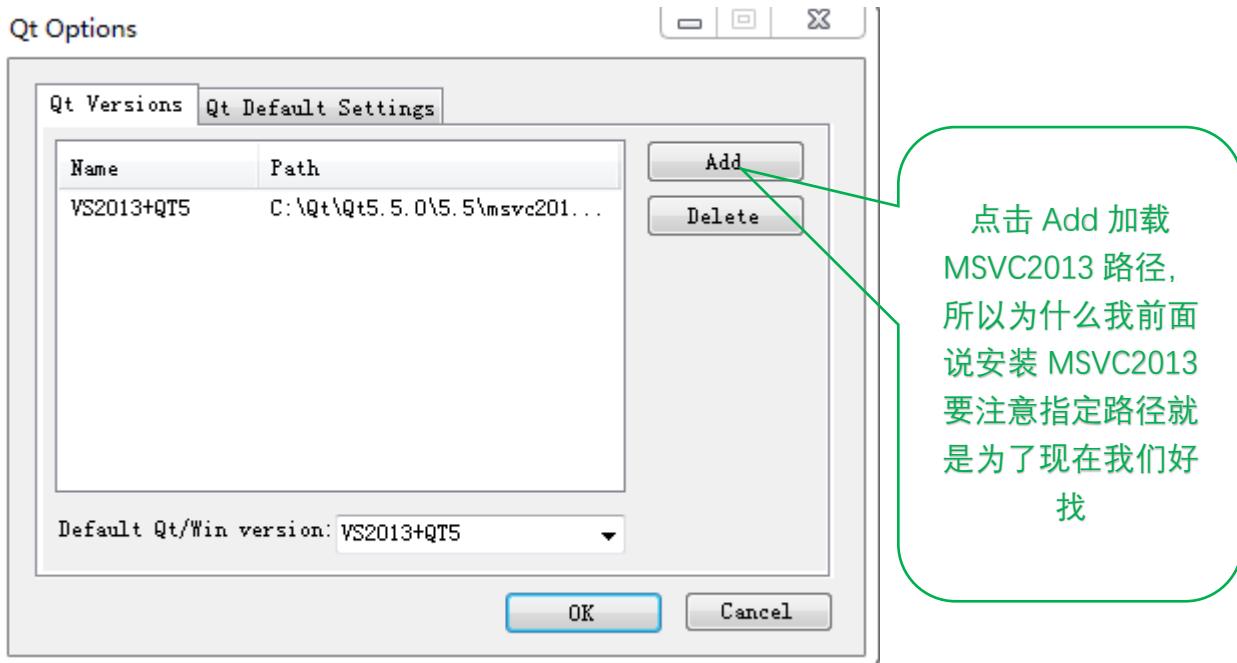
MSVC2013 软件下载地址 <http://download.qt.io/archive/qt/5.5/5.5.0/>

qt-opensource-windows-x86-msvc2013_64-5.5.0.exe 01-Jul-2015 09:26 650M Details

选择 X86-MSVC2013 64 位

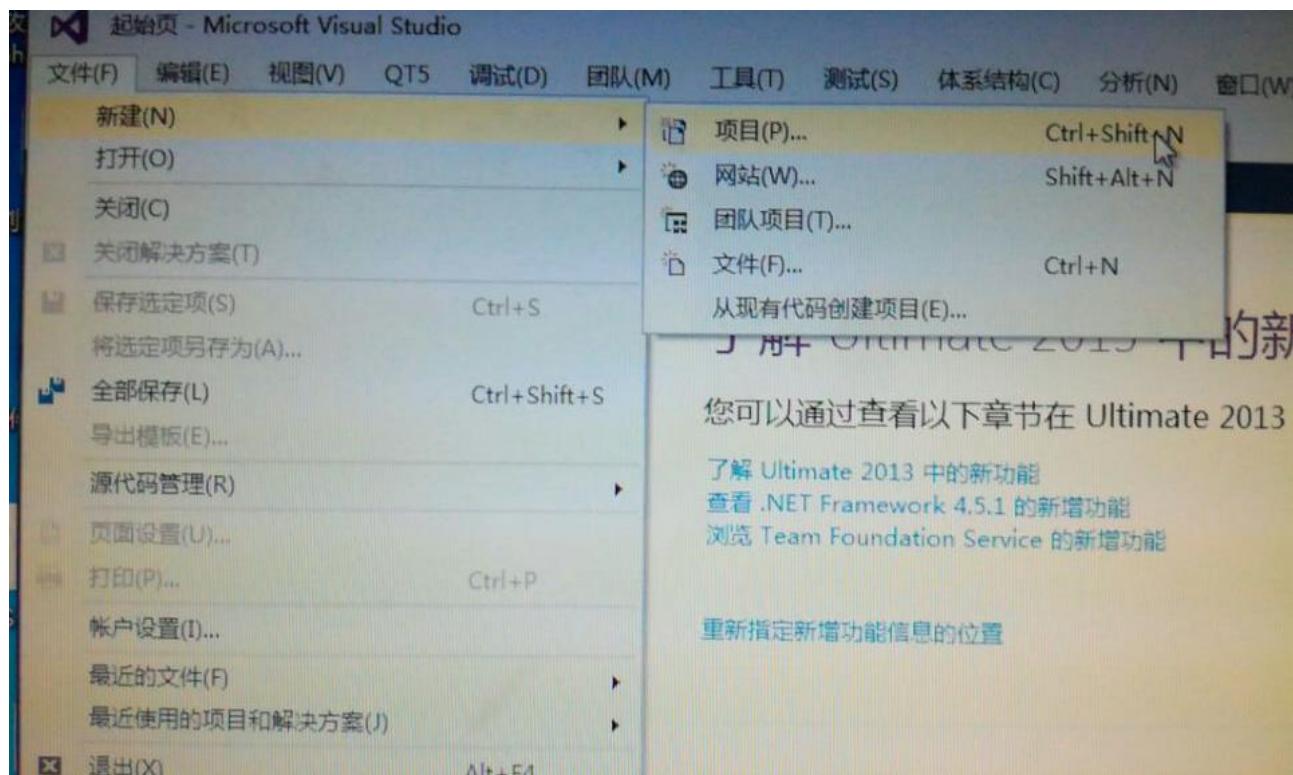
安装完成后打开 VS2013 软件





然后点击 OK, VS2013 的 QT 功能激活了。

创建一个 QT 工程



The screenshot shows the 'Create New Project' dialog in Visual Studio. The 'Qt Application' template is selected and highlighted with a green box. A callout bubble points to it with the text '选择第 1 个'. Below the dialog, a code editor window displays a simple 'Hello World' application. Red arrows point from the 'Qt Application' selection in the dialog to the corresponding code in the editor, specifically highlighting the Qt-related imports and main function calls.

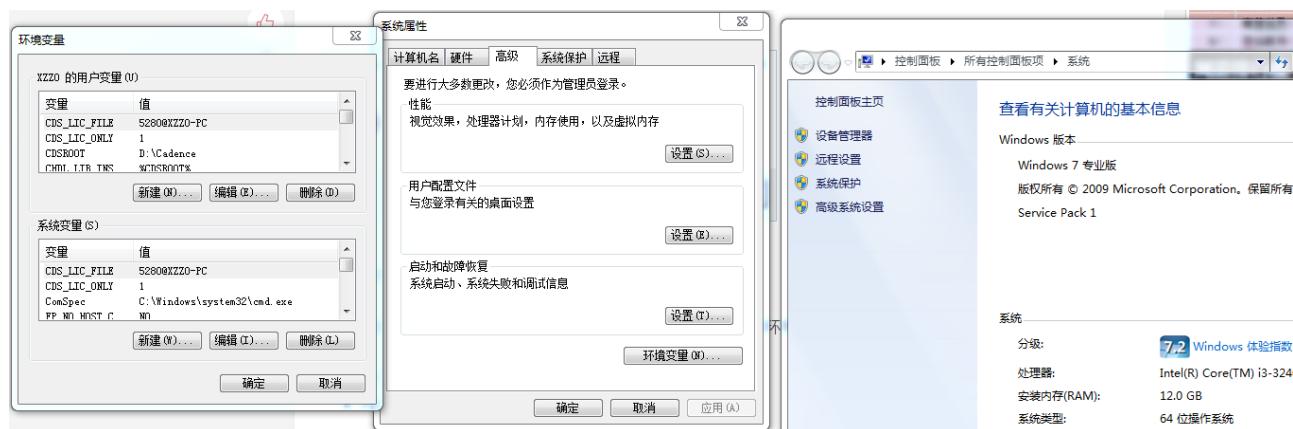
```
1 #include "demo.h"
2 #include <QtGui/QApplication>
3
4 int main(int argc, char *argv[])
5 {
6     QApplication a(argc, argv);
7     demo w;
8     w.show();
9     return a.exec();
10 }
11
```



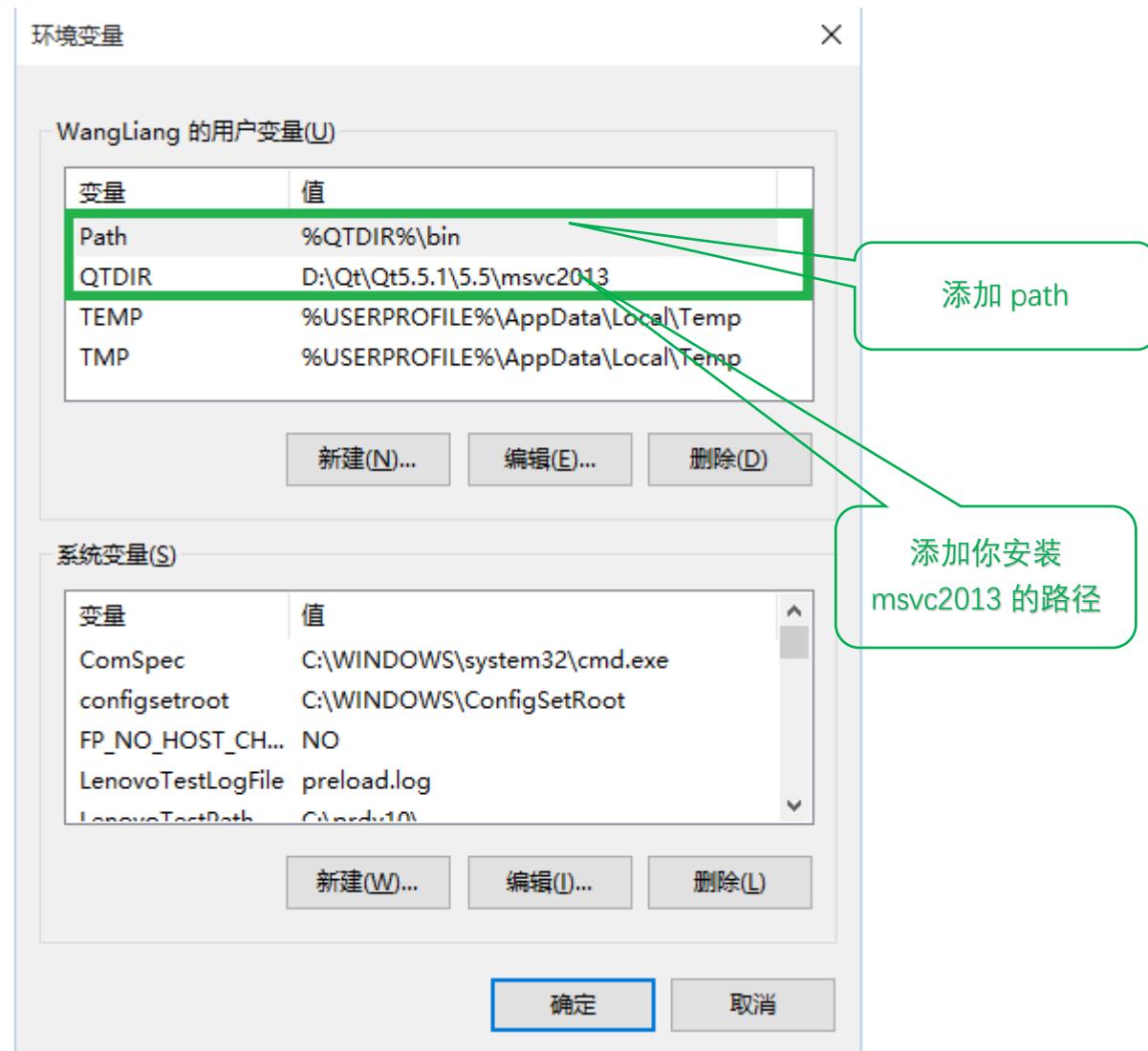
或者出现 dll 丢失情况

这是系统环境变量没有设置 QT 的库路径导致

所以为什么我要求 MSVC2013 安装时要指定路径，就是因为这里也要找到 MSVC2013 路径的一些库文件



打开计算机->高级设置->环境变量

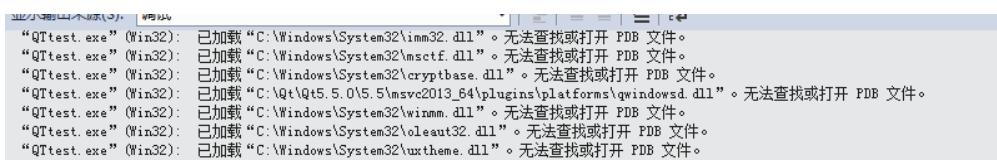


The screenshot shows the Qt Creator interface with the file 'main.cpp' open. The code editor displays the following code:

```
#include "qttest.h"
#include <QtWidgets/QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QTtest w;
    w.show();
    return a.exec();
}
```

这样你的代码就不会出现找不到类或者库的问题了。



听说编译工程的时候这些不用管

VS2013 破解

1 打开VS2013,找到注册产品



2 在下面框中输入密钥 : BWG7X-J98B3-W34RT-33B3R-JVYW9

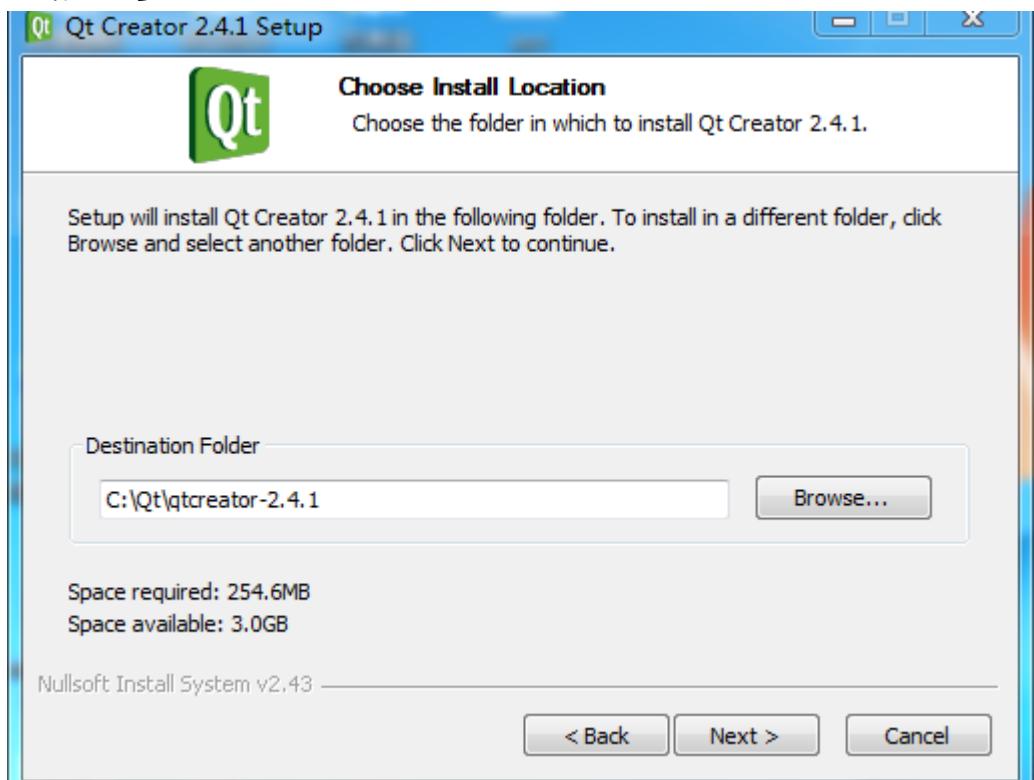
单击应用



Windows 下 qt-creator 安装



一路下一步



选择你要安装的路径，这个路径你要记住，后面装 QT SDK 的时候需要连接这个路径

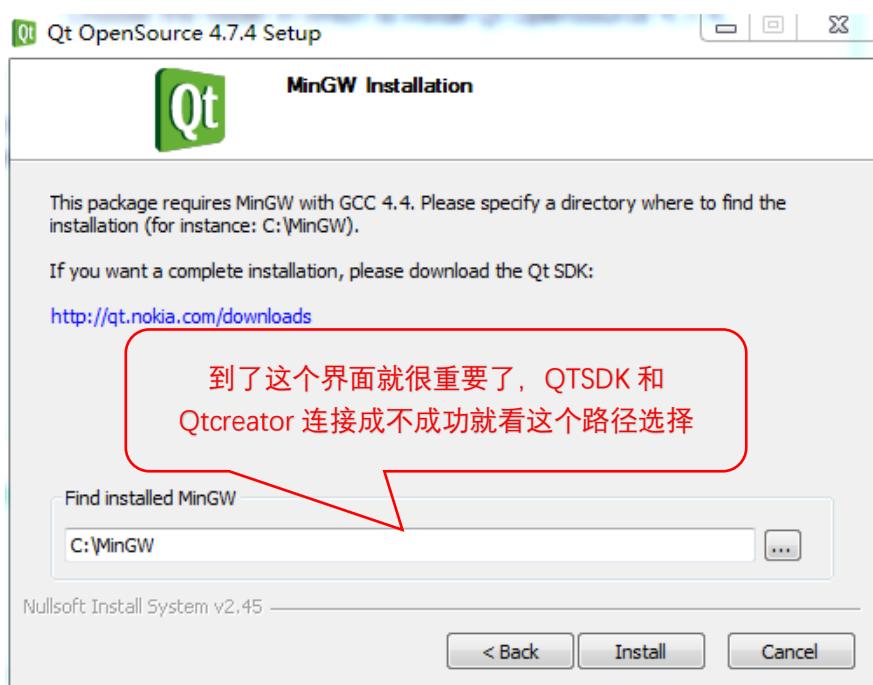


然后结束

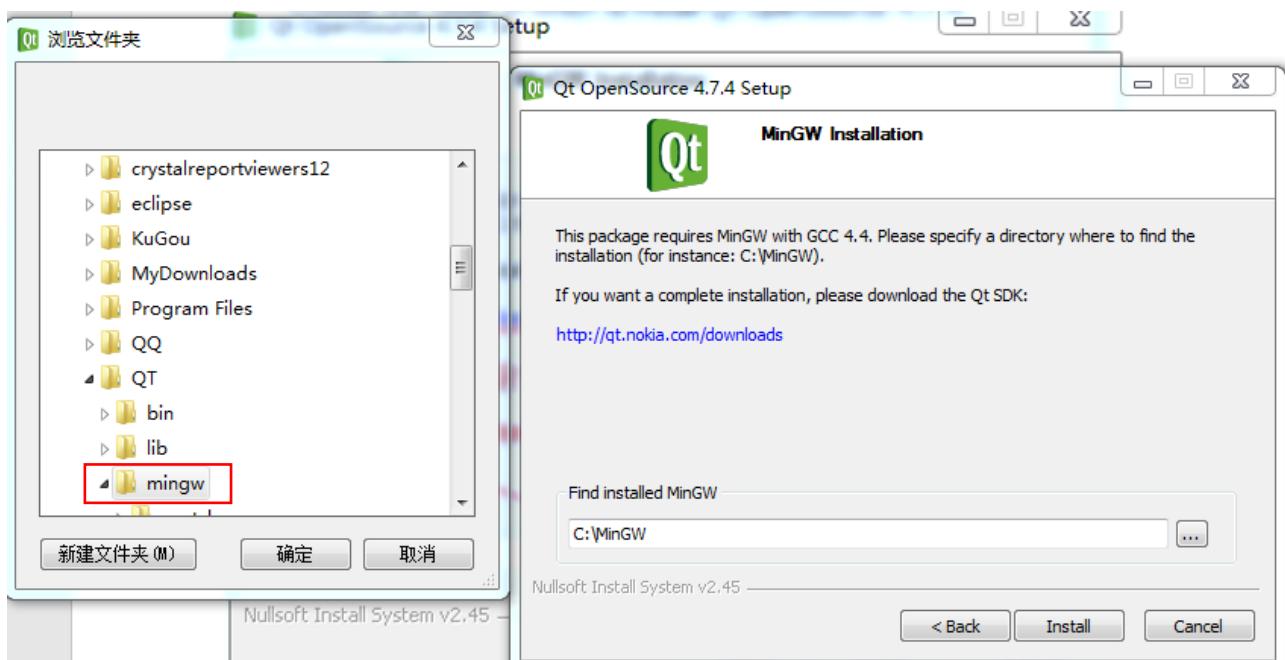


双击 qt-win..., 安装 QT SDK

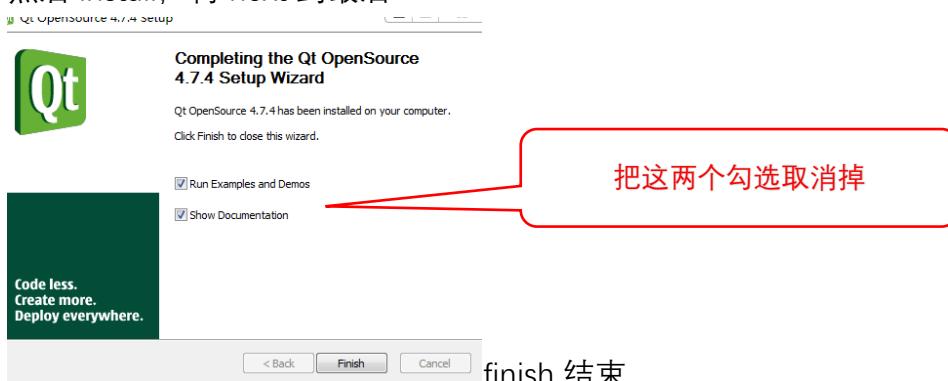
一路 next



一定要找到你最先安装 QT creator 时候自动生成的 mingw 文件夹路径

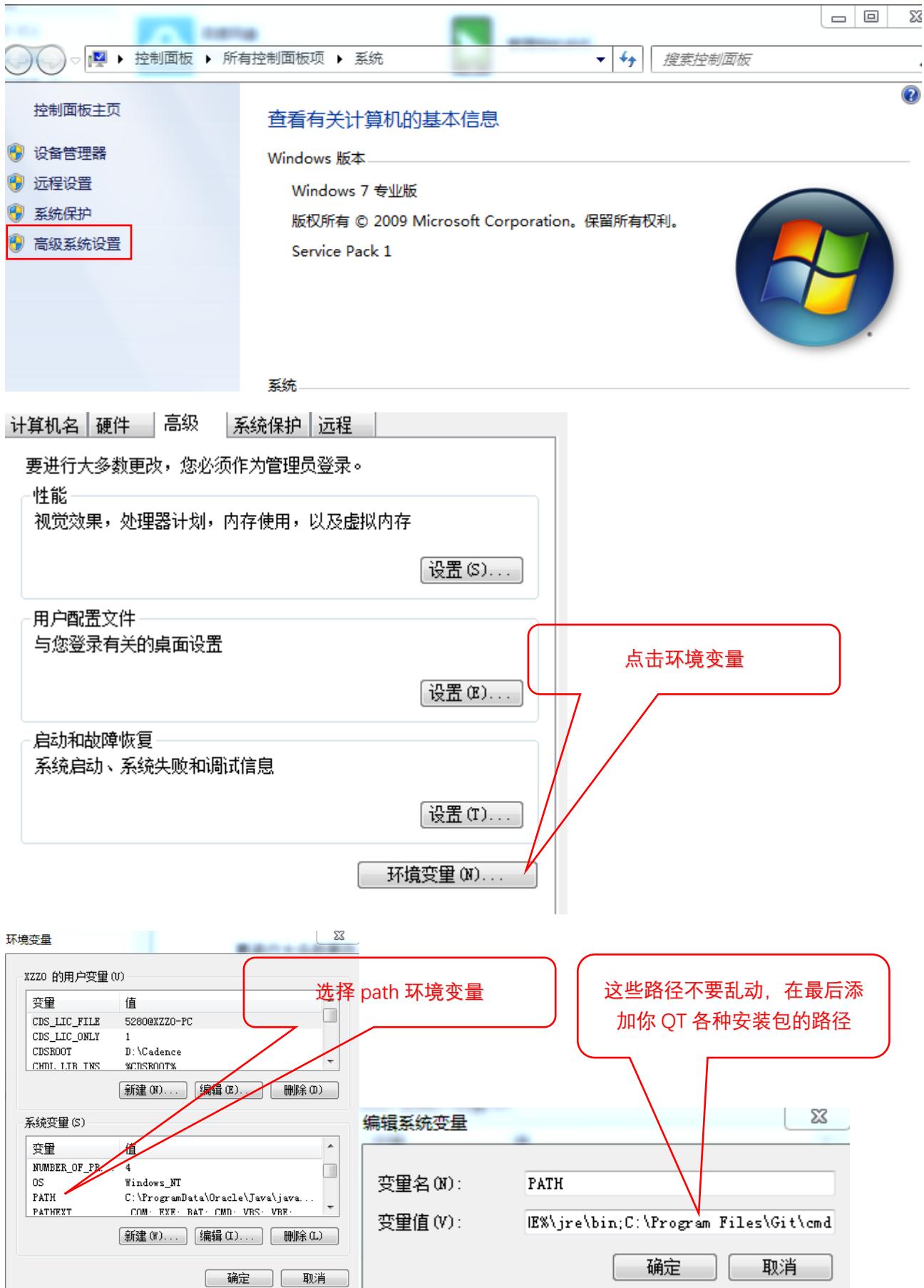


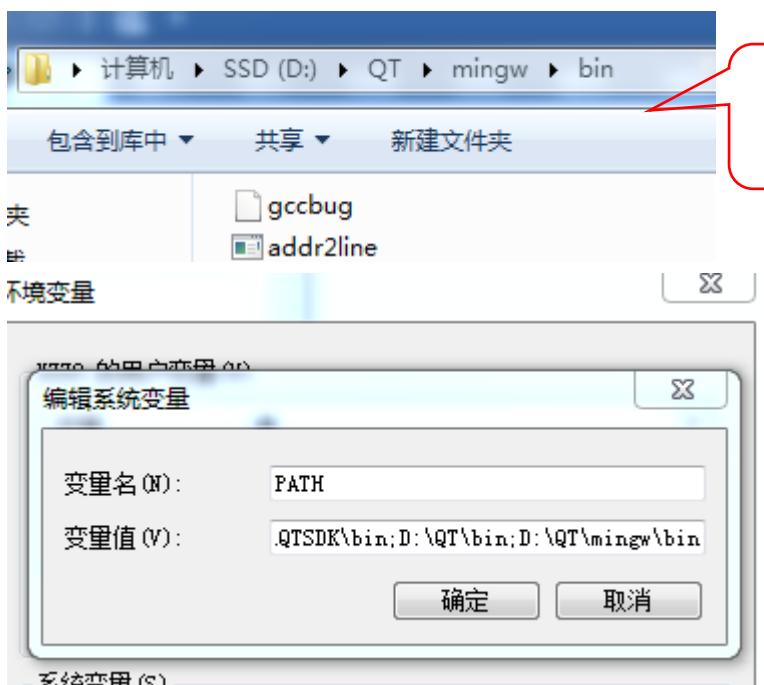
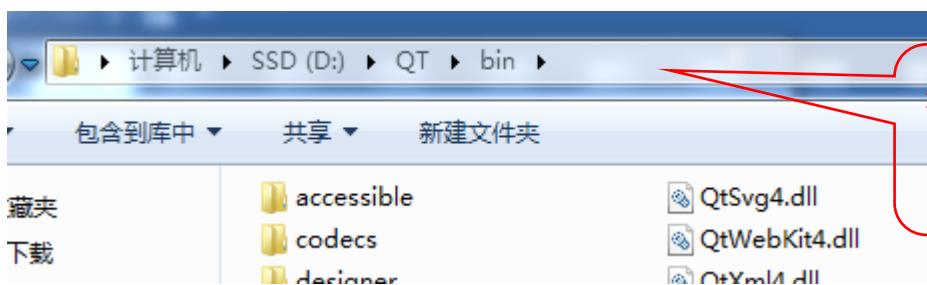
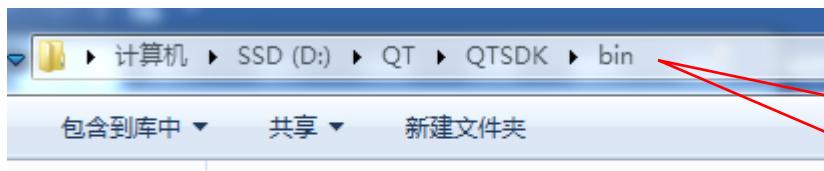
然后 install, 再 next 到最后



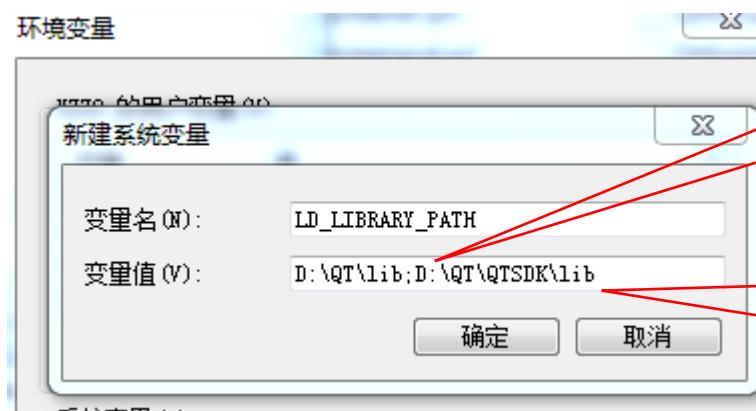
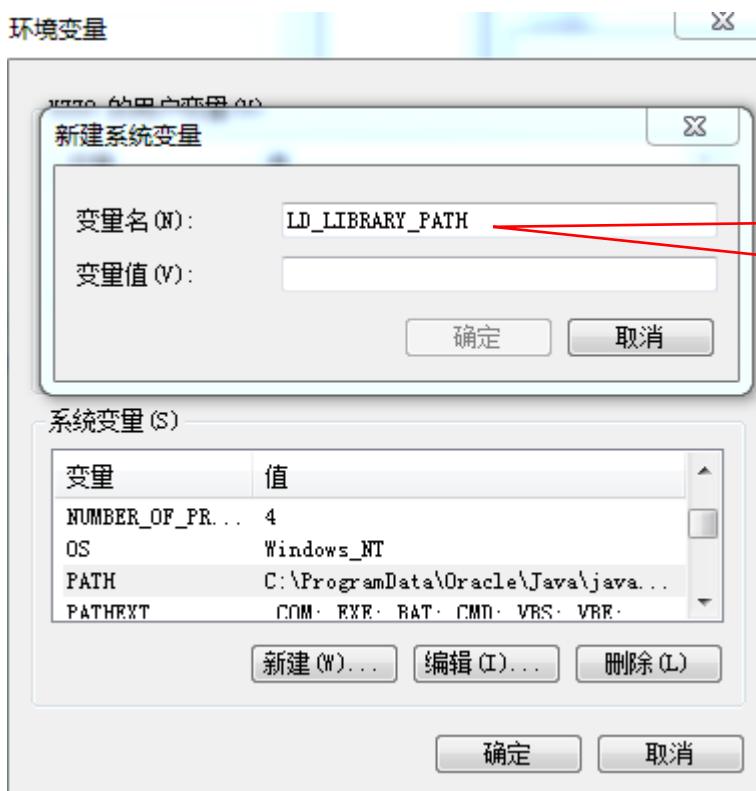
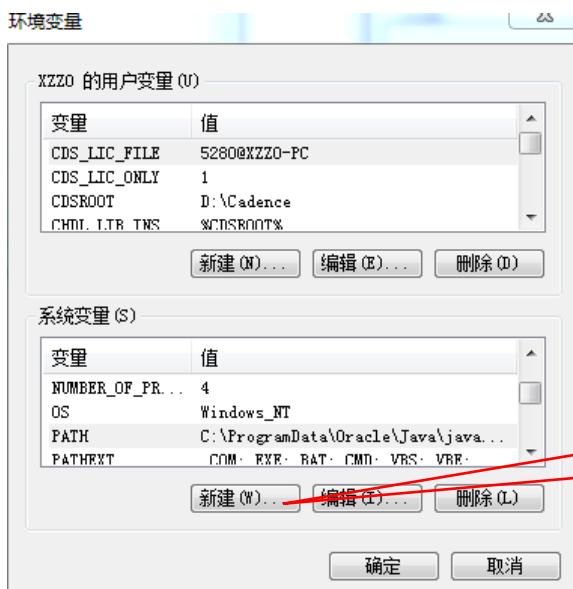
finish 结束

因为 Qtcreator 和 QT SDK 是分开安装的，所以我们还要进行最后的关联
设置环境变量





然后我们点击确定

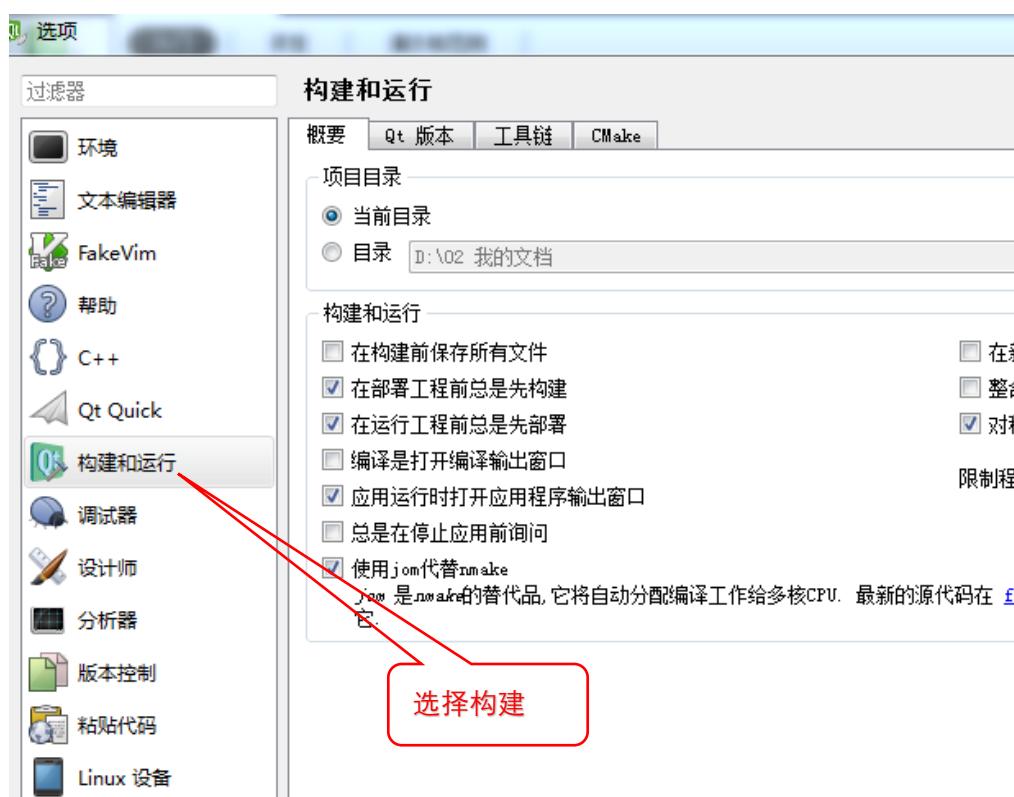
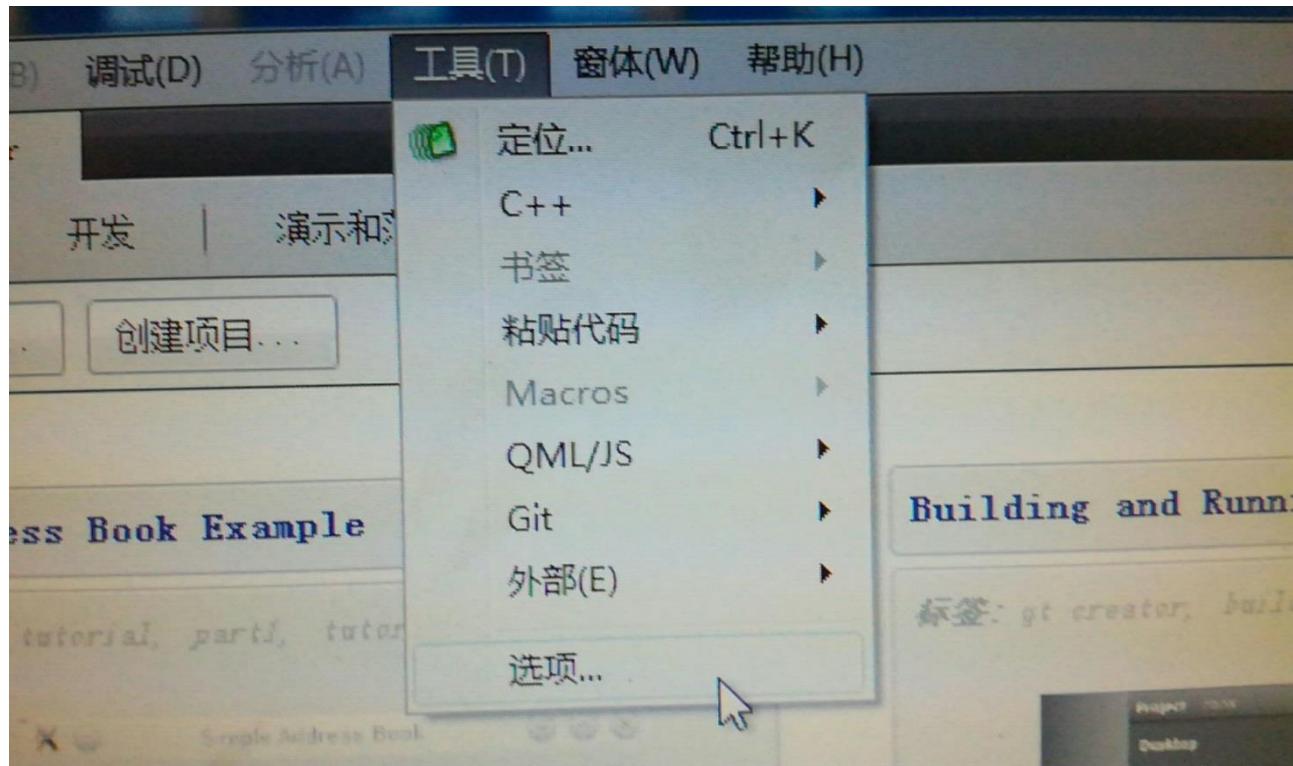


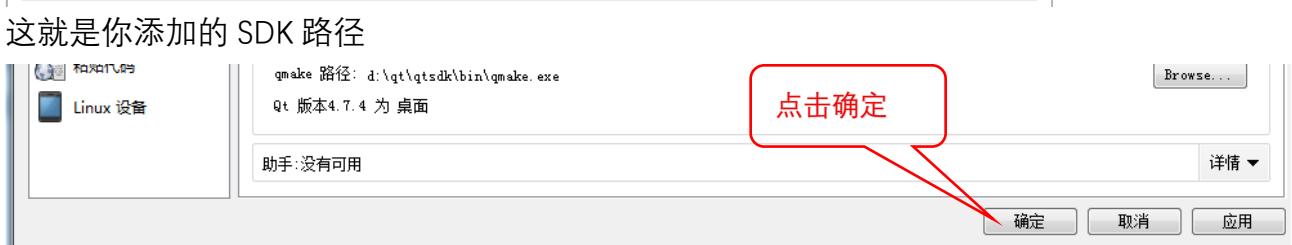
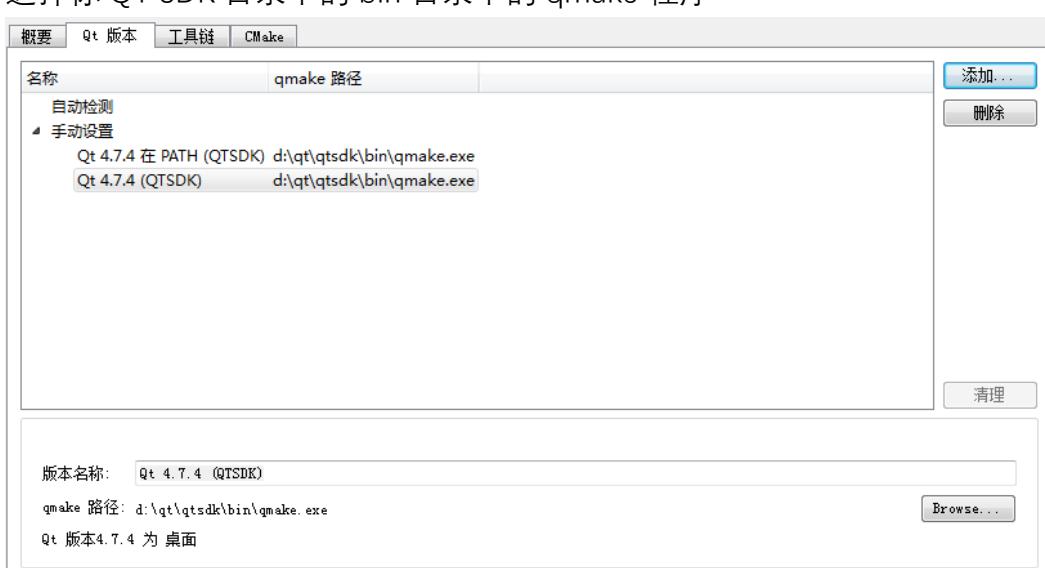
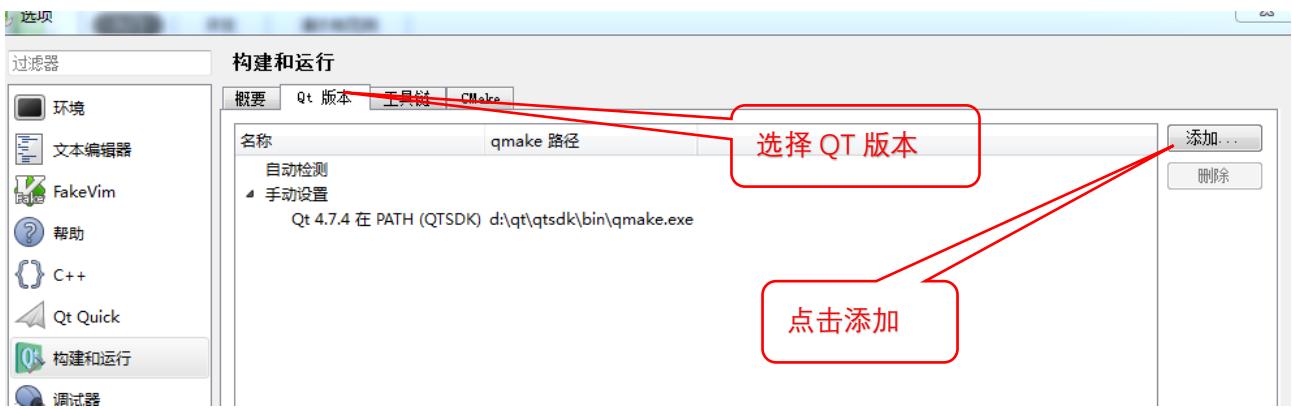
然后点击确定，环境变量设置完成



运行 QT creator

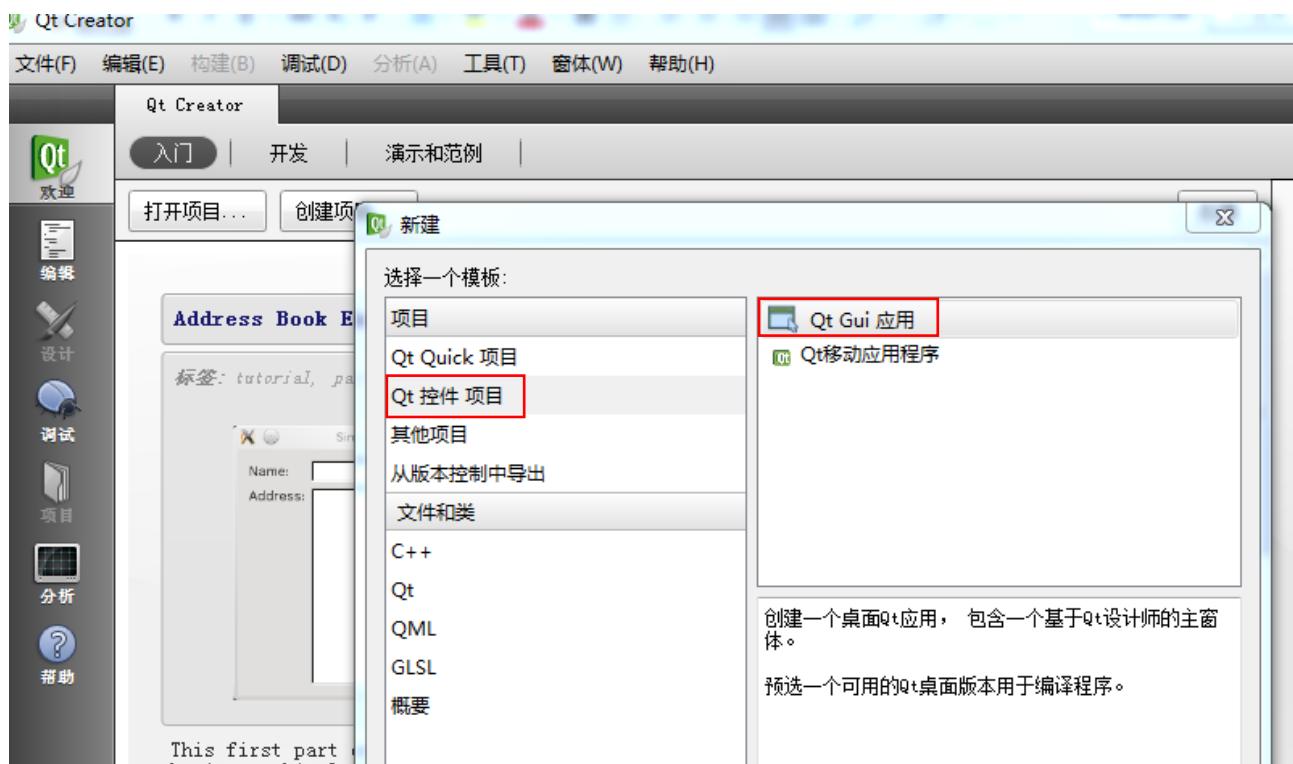
点击 工具->选项





然后点击确定完成 SDK 和 Qtcreator 的关联
然后点击新建工程

选择 QT 控件项目



然后点击选择



把工程名和工程保存的路径选择好，一路下一步，直到完成

QTtest

- QTTest.pro
- 头文件
- 源文件
 - main.cpp
 - mainwindow.cpp
- 界面文件

```
1 #include "mainwindow.h"
2 #include "ui_mainwindow.h"
3
4 MainWindow::MainWindow(QWidget *parent)
5     : QMainWindow(parent),
6     ui (new Ui::MainWindow)
7 {
8     ui->setupUi(this);
9 }
10
11 MainWindow::~MainWindow()
12 {
13     delete ui;
14 }
15
```

mainwindow.cpp

编译输出

```
17:32:21: 为项目QTtest执行构建步骤 ...
17:32:21: 正在启动 "d:\qt\qtsdk\bin\qmake.exe" "D:\02 我的文档\Desktop\QTtest\QTtest\QTtest.pro"
win32-g++
{1"?"} {2?}
Cannot find file: d:\02 我的文档\Desktop\QTtest\QTtest\QTtest.pro.
17:32:21: 进程"d:\qt\qtsdk\bin\qmake.exe"退出, 退出代码 2。
构建项目QTtest 时发生错误 (目标: 桌面)
当执行构建步骤 'qmake' 时
```

点击运行发现 qmake.exe 退出错误，这是因为我的桌面目录是中文路径。

mainwindow.cpp

```
6     ui (new Ui::MainWindow)
7 {
8     ui->setupUi(this);
9 }
10
11 MainWindow::~MainWindow()
12 {
13     delete ui;
14 }
15
```

程序输出

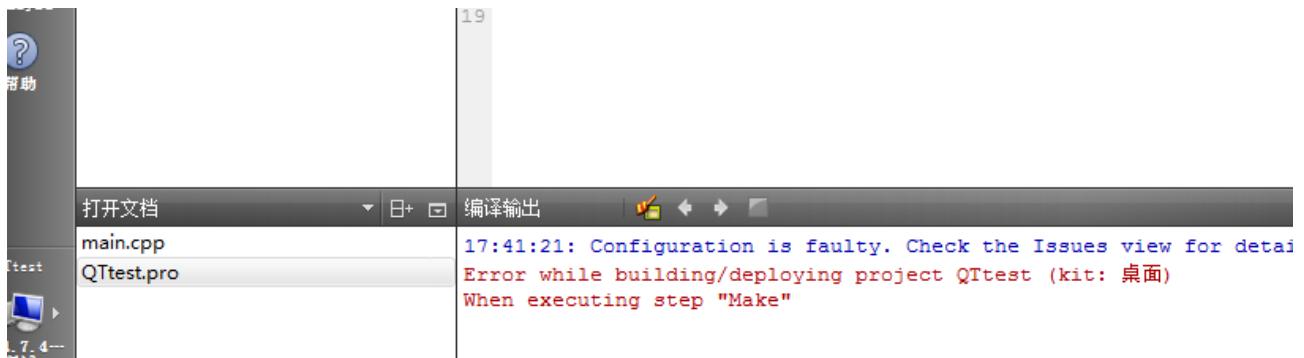
```
test x
\QTtest\QTtest-build-desktop-Qt_4_7_4_
\QTtest\QTtest-build-desktop-Qt_4_7_4_
代码: 0
{1 ?}
E:\QTtest\QTtest-build-desktop-Qt_4_7_4_
```

Qt ... 调试

这是在英文路径下建立的 QT 工程，一切正常
所以你建立的 QT 工程路径一定不能带有中文

Qtcreator 和 VS2013 QT 冲突的问题

你修改中文后创建第一次 QT 项目可以正常运行，然后你关闭 QT 项目，再重新打开这个 QT 项目就出现了如下错误



这是因为你用 VS2013 安装的 QT SDK 库存在，所以和 Qtcreator 的 SDK 库发生了冲突。解决办法就卸载电脑里面的 VS2013 的 QT SDK 库这样你的 Qtcreator 建立的工程就可以反复使用了，所以一山不能容二虎，用 VS2013 的 QT，就不能用 Qtcreator。

Linux 下 QT 的安装与搭建

```
root@ubuntu:/home/xiang/QTcreat# tar -vxf qt-everywhere-opensource-src-4.7.4.tar
```

qt-everywhere-opensource-src-4.7.4

先安装 `sudo apt-get install libxtst-dev`，不安装 libxtst-dev 库，`./configure` 会出错

```
root@ubuntu:/home/xiang/QTcreat# cd qt-everywhere-opensource-src-4.7.4
root@ubuntu:/home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4# ls
bin          config.tests  configure.exe  doc        imports  INSTALL    lib
changes-4.7.4  configure   demos       examples  include  LGPL_EXCEPTION.txt  LICEN
root@ubuntu:/home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4# ./configure
```

changes-4.7.4 configure demos examples include LGPL_EXCEPTION.TXT
root@ubuntu:/home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4# ./configure
Which edition of Qt do you want to use ?

Type 'c' if you want to use the Commercial Edition.
Type 'o' if you want to use the Open Source Edition.

Type 'c' if you want to use the Commercial Edition.
Type 'o' if you want to use the Open Source Edition.

这里点击 o 然后回车

```
Type 'yes' to accept this license offer.
Type 'no' to decline this license offer.
```

```
Do you accept the terms of either license?
```

这里写 yes

```
Do you accept the terms of either license? yes
```

然后 make

```
for /home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4/src/opengl/util/generator.pr  
Qt is now configured for building. Just run 'make'.  
Once everything is built, you must run 'make install'.  
Qt will be installed into /usr/local/Trolltech/Qt-4.7.4  
To reconfigure, run 'make confclean' and 'configure'.  
root@ubuntu:/home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4# make
```

然后 make install 安装

```
cp -f -r /home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4/mkspecs/wincewm60standard-msvc2008 /usr/local/Trolltech/Qt-4.7.4/mkspecs/  
cp -f -r /home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4/mkspecs/wincewm65professional-msvc2005 /usr/local/Trolltech/Qt-4.7.4/mkspecs/  
cp -f -r /home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4/mkspecs/wincewm65professional-msvc2008 /usr/local/Trolltech/Qt-4.7.4/mkspecs/  
root@ubuntu:/home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4#
```

没有报错，证明 QT SDK 安装完成

下面我们来安装 QT creator，因为没有安装各种图形库所以直接安装 qt-creator 是错误的

```
root@ubuntu:/home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4# ls  
root@ubuntu:/home/xiang/QTcreat# qt-creator-linux-x86-opensource-2.4.1.bin  qt-everywhere-opensource-src-4.7.4  qt-everywhere-opensource-src-4.7.4.tar  
root@ubuntu:/home/xiang/QTcreat# chmod 755 qt-creator-linux-x86-opensource-2.4.1.bin  
root@ubuntu:/home/xiang/QTcreat# ./qt-creator-linux-x86-opensource-2.4.1.bin
```

```
cp -f -r /home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4/mkspecs/wincewm60standard-msvc2008 /usr/local/Trolltech/Qt-4.7.4/mkspecs/  
cp -f -r /home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4/mkspecs/wincewm65professional-msvc2005 /usr/local/Trolltech/Qt-4.7.4/mkspecs/  
cp -f -r /home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4/mkspecs/wincewm65professional-msvc2008 /usr/local/Trolltech/Qt-4.7.4/mkspecs/  
root@ubuntu:/home/xiang/QTcreat/qt-everywhere-opensource-src-4.7.4#
```

这种显示 cp -f -r 是错误的

```
root@ubuntu:/home/xiang/67511476/bin# ./qtcreator  
./qtcreator: error while loading shared libraries: libgthread-2.0.so.0: cannot open shared object file: No such file or directory  
root@ubuntu:/home/xiang/67511476/bin# sudo apt-get install libglib2.0-0:i386
```

你看找不到库，所以我们下面先安装各种图形库

执行./qtcreator 发现无法找到库

```
sudo apt-get install libglib2.0-0:i386
```

```
root@ubuntu:/home/xiang/67511476/bin# ./qtcreator  
./qtcreator: error while loading shared libraries: libfreetype.so.6: cannot open shared object file: No such file or directory
```

执行./qtcreator 发现无法找到库

```
root@ubuntu:/home/xiang/67511476/bin# sudo apt-get install libfreetype6:i386  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following packages were automatically installed and are no longer required:  
Processing triggers for libc-bin (2.23-0ubuntu5) ...
```

```
root@ubuntu:/home/xiang/67511476/bin# ./qtcreator  
./qtcreator: error while loading shared libraries: libSM.so.6: cannot open shared object file: No such file or directory
```

执行./qtcreator 发现无法找到库

```
sudo apt-get install libgtk2.0-0:i386 libxxf86vm1:i386 libsm6:i386
```

```
lib32stdc++6
```

各种图形动态库安装完成

```
root@ubuntu:/usr/local/Trolltech# ls  
Qt-4.7.4  
root@ubuntu:/usr/local/Trolltech#
```

我们发现 QT SDK 默认安装在/usr/local/Trolltech 下面

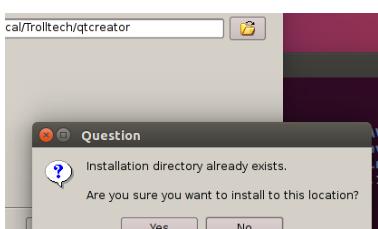
我们安装 qt-creator

 qt-creator-linux-x86-opensource-2.4.1.bin





然后点击 next



弹出这个对话框，我们点击 yes，然后再 next 就开始安装了
我们去修改.bash 脚本环境变量

```
root@ubuntu:~# ls -a
. . . .bash_history .bashrc .cache .gi
root@ubuntu:~# vim .bashrc
```

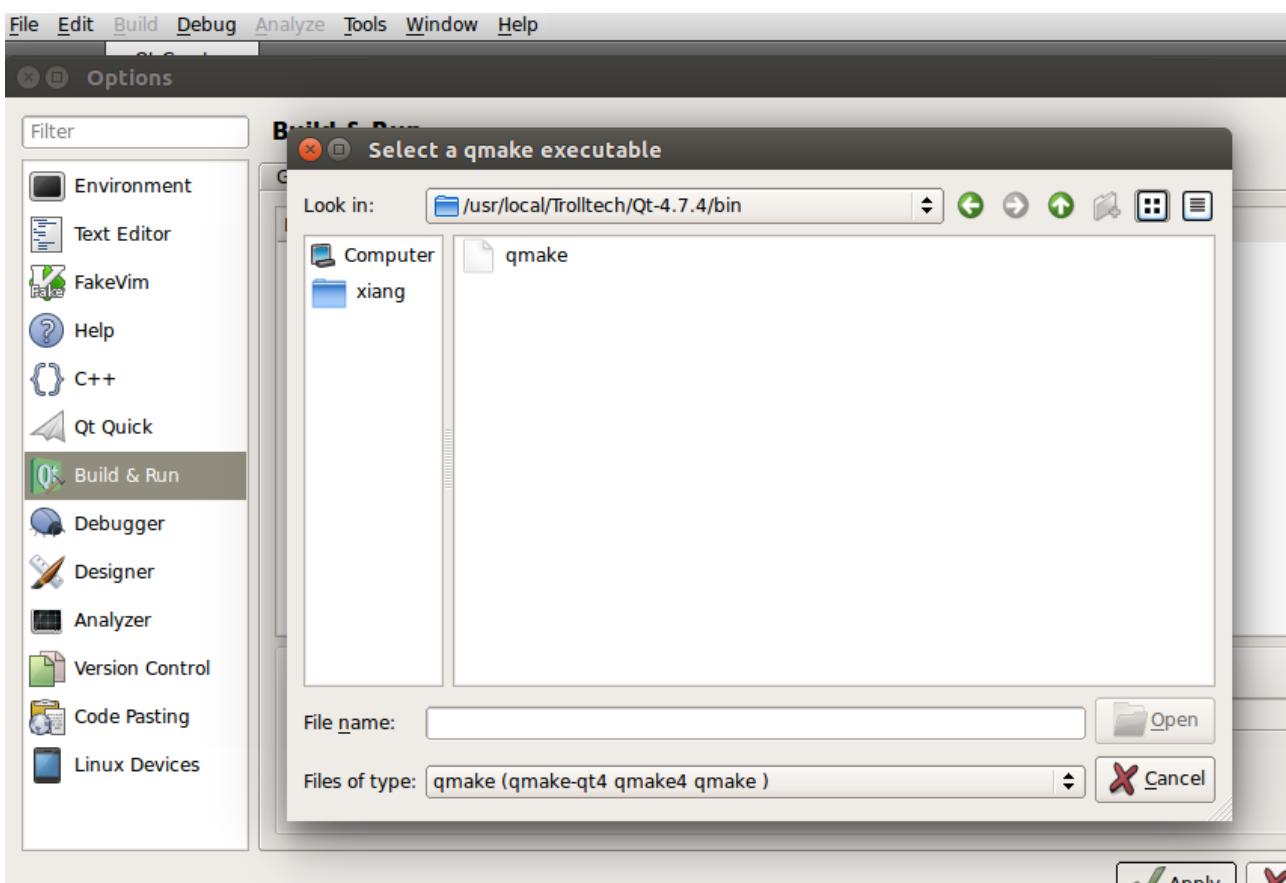
```
export QTDIR=/usr/local/Trolltech
export PATH=$QTDIR/Qt-4.7.4/bin:$QTDIR/qtcreator/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/Qt-4.7.4/lib:$QTDIR/qtcreator/lib:$LD_LIBRARY_PATH
1 export QTDIR=/usr/local/Trolltech
2 export PATH=$QTDIR/Qt-4.7.4/bin:$QTDIR/qtcreator/bin:$PATH
3 export LD_LIBRARY_PATH=$QTDIR/Qt-4.7.4/lib:$QTDIR/qtcreator/lib:$LD_LIBRARY_PATH
```

基本上和 windows7 设置环境变量方法一样。

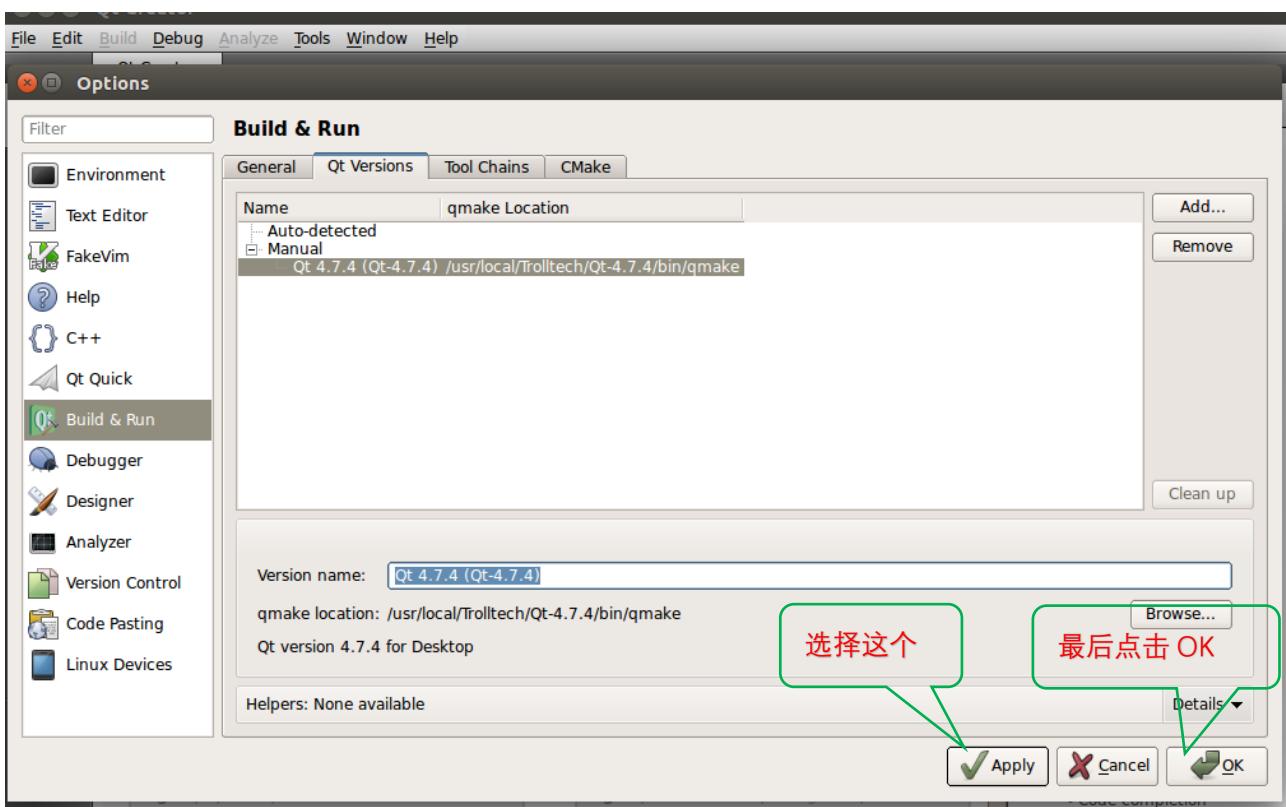
然后我们设置 QT SDK 和 qtcreator 的关联



打开 QT 软件



和 windows 一模一样，寻找 QT SDK 的 qmake 文件，添加上

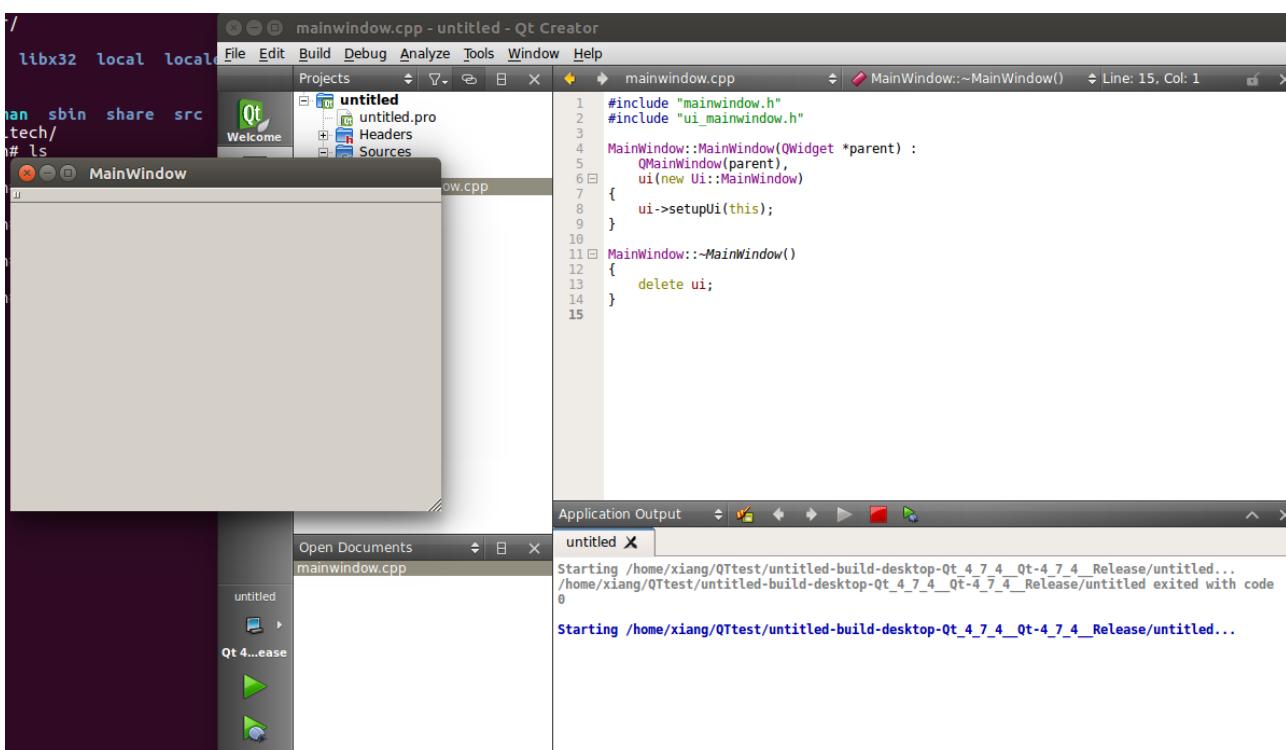


QTtest

QT 工程文件必须创建在没有权限的目录下

```
xiang@ubuntu:~$ ls
Alsa C++      C_code    Documents examples.desktop  IMX6  java_test   ls    Pictures QTtest  ubuntu16.04_plug-in vimpake
app  C-C_Grammar_Improve Desktop  Downloads githubfile   JAVA  linux_code_guide  Music  Public  templates  Videos  yellow_ht
xiang@ubuntu:~$
```

你看没有权限



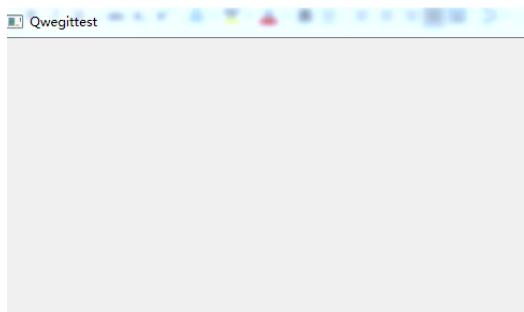
这样工程就顺利运行了。但是如果移植 windows 写好的 QT 程序到 linux，要注意编译器和文件工程输出的路径

创建一个主程序顶层窗口

所有的按钮，对话框，窗口等等都是基于 QWidget 这个类

```
#include <QtGui/QApplication>
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    w.show();
    return a.exec();
}
```

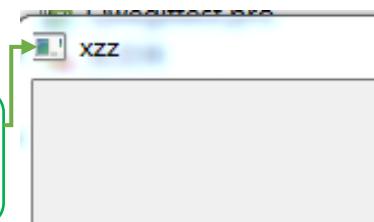


窗口创建成功

设置窗口名称

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    w.setWindowTitle("xzz");
    w.show();
    return a.exec();
}
```

用 setWindowTitle
设置窗口名称



设置窗口的最大化最小化栏，变成问号栏。

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w(NULL, Qt::Dialog);
    w.setWindowTitle("xzz");
    w.show();
    return a.exec();
}
```

那么最大化最小化栏变成问好

在窗口初始化对象上赋值 Qt::dialog

设置窗口贴死在桌面

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w(NULL, Qt::SplashScreen);
    w.setWindowTitle("xzz");
    w.show();
    return a.exec();
}
```

初始化父窗口设置
贴死在桌面

这种贴死在桌面可以用来
做打开软件的启动 log

设置父窗口初始化大小

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    w.setWindowTitle("xzz");
    w.resize(100,100);
    w.show();
    return a.exec();
}
```



用 resize 可以设置父窗口
打开时的大小

把父窗口卡死在顶层

```
#include <QtGui>
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w(NULL, Qt::Window|Qt::WindowStaysOnTopHint);
    w.setWindowTitle("xzz");
    w.resize(100,100);
    w.show();
    return a.exec();
}
```

在父窗口初始化的时候加这个
参数，窗口就始终在顶层



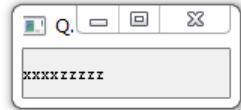
不管你打开什么软件，父窗口
始终不会被其它软件遮盖

生成标题栏

```
#include <QtGui>
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QLabel l;
    l.setText("xxxxzzzz");
    l.show();
    return a.exec();
}
```

标题栏要用 QLabel 这个类
标题栏显示
标题栏名称



但是我想把这个标题栏弄到父窗口去显示怎么办？

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    QLabel l(&w);
    l.setText("xxxxzzzz");
    w.setWindowTitle("xzz");
    w.show();
    return a.exec();
}
```

这是将标题栏放入窗口句柄

你看这是窗口名



这是标题栏

Qt 程序父窗口坐标及窗口大小获取

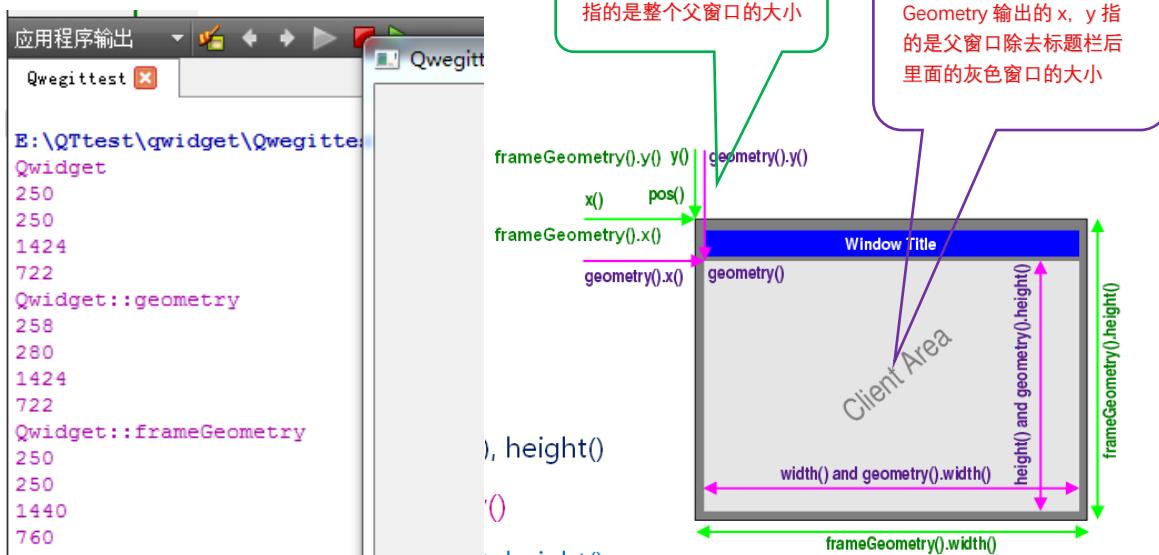
```
#include <QtGui>
#include <QDebug> // 调试用的log函数，你可以向qDebug() << "XXXXXX" 输入字符串
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;

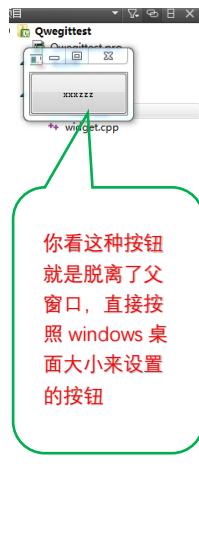
    w.show();
    qDebug() << "QWidget";
    qDebug() << w.x();
    qDebug() << w.y();
    qDebug() << w.width();
    qDebug() << w.height();

    qDebug() << "QWidget::geometry";
    qDebug() << w.geometry().x();
    qDebug() << w.geometry().y();
    qDebug() << w.geometry().width();
    qDebug() << w.geometry().height();

    qDebug() << "QWidget::frameGeometry";
    qDebug() << w.frameGeometry().x();
    qDebug() << w.frameGeometry().y();
    qDebug() << w.frameGeometry().width();
    qDebug() << w.frameGeometry().height();
    return a.exec();
}
```



按钮组件(QPushButton)



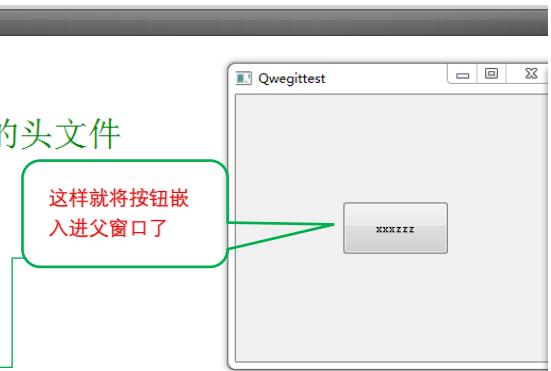
```
#include <QtGui>
#include <QDebug>
#include <QPushButton> //生成按钮的头文件
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QPushButton button;

    button.setText("xxxxxx"); //给按钮写名字
    button.move(100,100); //设置按钮创建后在windows界面的初始位置
    button.resize(100,50); //设置按钮大小
    button.show(); //显示按钮

    return a.exec();
}
```

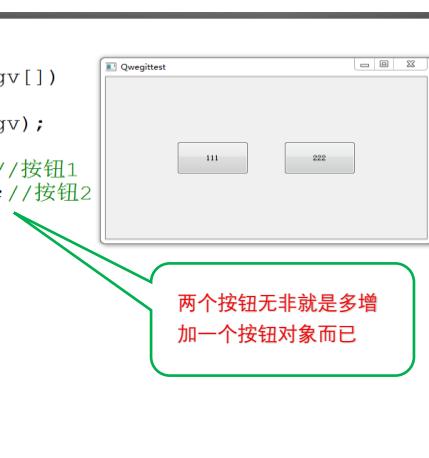
创建一个按钮，这种方式是创建类的静态对象



```
#include <QtGui>
#include <QDebug>
#include <QPushButton> //生成按钮的头文件
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    QPushButton button(&w); //将按钮嵌入进w对象的父窗口

    button.setText("xxxxxx"); //给按钮写名字
    button.move(100,100); //设置按钮创建父窗口界面的初始位置
    button.resize(100,50); //设置按钮大小
    w.show();
    return a.exec();
}
```



```
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    QPushButton button1(&w); //按钮1
    QPushButton button2(&w); //按钮2

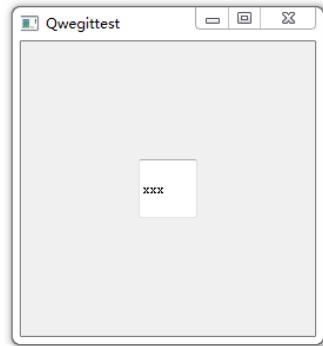
    button1.setText("111");
    button1.move(100,100);
    button1.resize(100,50);

    button2.setText("222");
    button2.move(250,100);
    button2.resize(100,50);
    w.show();
    return a.exec();
}
```

文本框组件(QLineEdit)

```
#include <QtGui>
#include <QDebug>
#include <QLineEdit> //文本框组件
#include "widget.h"

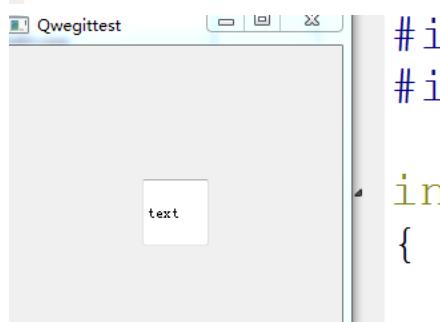
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    QLineEdit *l = new QLineEdit(&w); //将文本框嵌套进父窗口
    l->move(100, 100);
    l->resize(50, 50);
    w.show();
    return a.exec();
}
```



这里的字符是我手动敲入的，有没有办法让文本框运行就有字符

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    QLineEdit *l = new QLineEdit(&w); //将文本框嵌套
    l->move(100, 100);
    l->resize(50, 50);
    QString text = "text"; //将字符串写入text对象
    l->setText(text); //让文本框初始化运行时就有字符
    w.show();
    return a.exec();
}
```

加上这两行代码。
QLineEdit :: setText 就是设置
初始化文本框内容



这个文本框就有初始字符串了

QT 消息处理方法，信号与槽

- Qt 的核心 – `QObject::connect` 函数

```
bool connect(const QObject* sender,           // 发送对象
             const char* signal,            // 消息名
             const QObject* receiver,       // 接收对象
             const char* method,           // 接受对象的成员函数
             Qt::ConnectionType type = Qt::AutoConnection);
```

Note :

- 在 Qt 中，消息用字符串进行描述
- `connect` 函数在消息名和处理函数之间建立映射

我们先来实现按钮事件的接受程序

```
#include <QtGui/QApplication>
#include <QPushButton>
#include <QDebug>
#include "Qbutton.h"

class xzzfunc:public QWidget
{
    Q_OBJECT
private slots:
    void function();
```

首先要确认发送函数和接受函数实现的类都必须继承了 `QObject` 类，这里的 `QWidget` 继承了 `QObject` 类

```
};

void xzzfunc::function()
{
    qDebug ()<<"qt connect";
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QPushButton button;
    xzzfunc xfunc;

    button.setText ("button");
    button.show();
    QObject::connect (&button, SIGNAL(clicked()), &xfunc, SLOT(function()));
    return a.exec();
}
```

定义槽接收函数，前面必须加 `private slots:`

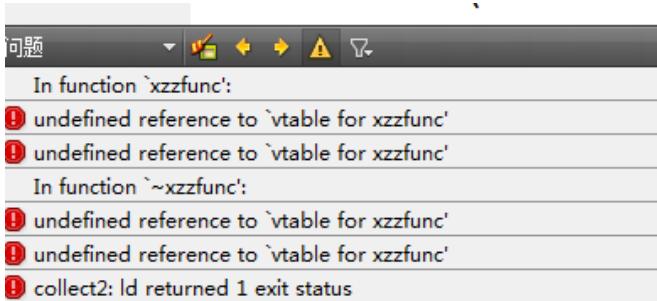
然后定义槽接收函数

一旦按钮按下就会执行按钮里面的信号函数发送事件消息给我的这个槽 `function` 函数。
我的这个槽 `function` 函数接受到这个信号消息就会执行

用 `QObject::connect` 函数，将按钮里面的信号函数放进来。`clicked` 是 QT 按钮控件自己实现的信号函数

将我自己实现的槽接受函数放进来

这样信号函数和槽函数就关联起来了
这样写代码没有错，但是编译会报错



这是因为在用信号与槽的功能下，必须在定义发送信号的函数，和负责接受槽函数类声明中加 Q_OBJECT，但是这个 Q_OBJECT 必须放在头文件下

```

#ifndef QBUTTON_H
#define QBUTTON_H

#include <QWidget>
class xzzfunc:public QWidget
{
    Q_OBJECT
private slots:
    void function();

};

namespace Ui {
    class Widget;
}

主函数

void xzzfunc::function()
{
    qDebug() << "qt connect";
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QPushButton button;
    xzzfunc xfunc;

    button.setText("button");
}


```

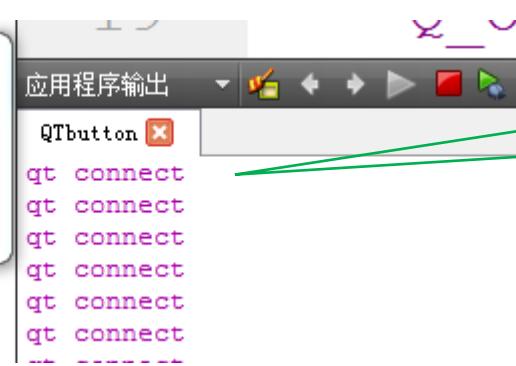
你看我将定义 Q_OBJECT 的 class 类放在了 h 文件下

你看我将 class 类定义的槽接受函数也放在了 h 文件下

主函数只负责实现 class 类里面的成员函数

这样编译就没有问题了，运行也很正常

所以说 QT 的信号与槽很扯。



下面我们完整的定义个信号发送函数，和槽接受函数

```
#ifndef SENDRECEVIE_H
#define SENDRECEVIE_H
#include<QObject>
// ----- send.cpp -----
class sender : public QObject
{
    Q_OBJECT
public:
    void doSend();
signals:
    void send();
};

// ----- receive.cpp -----
class receiver : public QObject
{
    Q_OBJECT
public slots:
    //带有参数的槽函数，需和绑定的信号的参数保持一致
    void recv();
};

#endif // SEND_H
```

第 2，在头文件定义一个无参信号发送函数，这个函数 QT 规定不需要实现，只需要定义

第 3，在接受类这里定义信号接受函数，该函数返回值和形参必须和发送函数 send 一致

在 C 文件实现头文件里面的函数

```
#include "sendreceiving.h"
#include<QDebug>
void sender::doSend()
{
    emit send();
}

void receiver :: recv()
{
    qDebug() << "recv number: " << endl;
```

在 cpp 文件中实现发送函数，用定义的外部调用去调用 send 发送函数，send 前面的 emit 是关键字，就是该函数执行发送功能，看到没有 send 是不需要实现的，send 发送是需要 dosend 外部调用函数来承载的，不能直接在主函数里面执行 send

接受函数就没有发送函数那么多限制了，直接实现接受函数即可

```

#include <QtGui/QApplication>
#include <QDebug>
#include "widget.h"
#include "sendreceiving.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    sender s;
    receiver r;
    QObject::connect(&s, SIGNAL(send()), &r, SLOT(recv()));
    s.doSend();
    return a.exec();
}

```

必须用 dosend 间接的去调用 emit 的发送函数，信号才会发送出去，调用一次发送一次

传入发送对象

信号接受对象和接受函数

执行 emit 的发送函数

```

{1 ?}
E:\QTtest\Qsendrcv\sendrcv-build-desktop-Qt_4_7_4__PATH_QTSRK__\debug\sendrcv.exe 启动中...
recv number:

```

程序执行成功

下面我们让接受端也能回传信号给发送端

```

#ifndef SENDRECEVIE_H
#define SENDRECEVIE_H
#include<QObject>
// ----- send.cpp -----
class sender : public QObject
{
    Q_OBJECT
public:
    void doSend();

signals:
    void send();
public slots:
    void sendrcv(); // 在发送端增加接受函数
};


```

```
// ----- receive.cpp -----
class receiver : public QObject
{
    Q_OBJECT

public:
void recvdoSend();
```

在接收端增加执行发送的函数

```
signals:
void rcvsend();
```

在接收端增加发送函数

```
public slots:
    //带有参数的槽函数，需和绑定的信号的参数保持一致
void recv();
```

```
};
```

```
#endif // SEND_H
```

在 Cpp 文件里实现头文件函数

```
#include<QDebug>
void sender::doSend()
{
    emit send();
}

void receiver :: recv()
{
    qDebug() << "recv number: " << endl;
}

void sender::sendrcv()
{
    qDebug() << "send number: " << endl;
}
void receiver::recvdoSend()
{
    emit rcvsend();
}
```

发送端接受函数实现

接受端发送函数实现

```

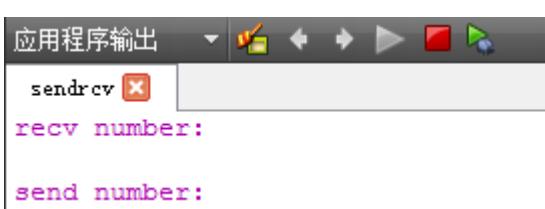
#include <QtGui/QApplication>
#include <QDebug>
#include "widget.h"
#include "sendrecevie.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    sender s;
    receiver r;
    QObject::connect(&s, SIGNAL(send()), &r, SLOT(recv()));
    QObject::connect(&r, SIGNAL(rcvsend()), &s, SLOT(sendrcv()));
    s.doSend();
    r.recvdoSend();
    return a.exec();
}

```

执行接受端发送信号

增加接受端发送函数，和发送端接受函数的信号槽链接



程序执行成功

信号和槽之间还可以传递数据

```

1 #ifndef SENDRECEVIE_H
2 #define SENDRECEVIE_H
3 #include<QObject>
4 // ----- send.cpp -----
5 class sender : public QObject
6 {
7     Q_OBJECT
8     public:
9         void doSend();
10    signals:
11        void send(int value);
12 };
13 // ----- receive.cpp -----
14 class receiver : public QObject
15 {
16     Q_OBJECT
17     public slots:
18         //带有参数的槽函数，需和绑定的信号的参数保持一致
19         void recv(int value);
20 };
21 #endif // SEND_H

```

发送信号变量类型必须和接受变量类型一致

接受信号变量类型必须和发送信号变量类型一致

在 CPP 文件里面实现头文件函数

```
#include "sendreceiving.h"
#include <QDebug>
void sender::doSend()
{
    int num = 50;
    emit send(num);
}

void receiver :: recv(int value)
{
    qDebug() << "recv number: " << value << endl;
}
```

虽然 send 发送函数不需要实现，但是可以向里面传入参数发送给接受端

接受端接受到 signal 的数据打印出来

```
#include <QtGui/QApplication>
#include <QDebug>
#include "widget.h"
#include "sendreceiving.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    sender s;
    receiver r;
    QObject::connect(&s, SIGNAL(send(int)), &r, SLOT(recv(int)));
    s.doSend();

    return a.exec();
}
```

因为 signal 是有参函数，所以这里要写入类型

因为 slot 和 signal 参数一致，所以这里要写入类型

The screenshot shows the Qt Creator application output window. The title bar says "应用程序输出". The window contains the following text:
sendrcv {1 ?}
E:\QTtest\Qsendrcv\sendrcv-build-debug
recv number: 50

执行成功

QT 的字符串类

因为 C++ 的 STL 库在跨平台上可能 STL 里面的代码还不是一样的，所以不管是 android, windows, linux 系统的 STL 可能都有差异，我们 QT 就用自己库里面的字符串类来开发

```
1 #include <QtGui/QApplication>
2 #include "widget.h"
3 #include <QDebug>
4
5 int main(int argc, char *argv[])
{
6     QApplication a(argc, argv);
7
8     QString s = "xxxxzzz";
9     qDebug() << s;
10    return a.exec();
11 }
12
13
```



```
5 int main(int argc, char *argv[])
6 {
7     QApplication a(argc, argv);
8
9     QString s = "xxxxzzz";
10    qDebug() << s;
11    s.append("ccc"); // 向s定义的字符串后面追加字符
12    qDebug() << s;
13    return a.exec();
14 }
```



```
5 int main(int argc, char *argv[])
6 {
7     QApplication a(argc, argv);
8
9     QString s = "xxxxzz";
10    qDebug() << s;
11    s.prepend("ccc"); //向s定义的字符串前面追加字符
12    qDebug() << s;
13    return a.exec();
14 }
```

应用程序输出 Qstr

E:\QTtest\Qstr\Qstr-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstr.exe 启动中...
"xxxxzz"
"cccxxxxxzz"

向前追加字符串

字符串替换

```
5 int main(int argc, char *argv[])
6 {
7     QApplication a(argc, argv);
8
9     QString s = "xxxxzz";
10    qDebug() << s;
11    s.replace("xxx", "uuu"); //替换字符串，将xxx替换成uuu
12    qDebug() << s;
13    return a.exec();
14 }
```

应用程序输出 str

E:\QTtest\Qstr\Qstr-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstr.exe 启动中...
"xxxxzz"
"uuuzzz"

xxx 被替换

Sprintf 输出字符串

```
5 int main(int argc, char *argv[])
6 {
7     QApplication a(argc, argv);
8
9     QString s = "";
10    int index = 10;
11    char *p = "abcdefg";
12    s.sprintf("numid = %d string = %s", index, p);
13    qDebug() << s;
14    return a.exec();
15 }
```

QString :: sprintf 类似 printf 字符串格式化输出

但是必须将格式后的字符串用 qDebug 才能显示出来

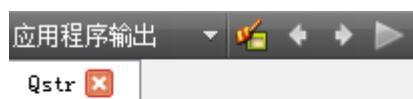
应用程序输出 Qstr

E:\QTtest\Qstr\Qstr-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstr.exe 启动中...
"numid = 10 string = abcdefg"

QString 寻找某个字符在字符串里面的位置

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

    QString s = "abcde,szaed,adsad";
    int index;
    index = s.indexOf(",", ",");
    qDebug() << index;
    return a.exec();
}
```



E:\QTtest\Qstr\Qstr-build

5

逗号在字符串第 6 的一个字符，从 0 开始计算就是 5

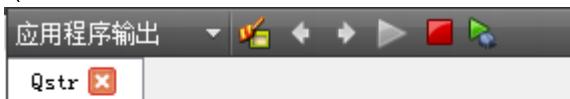
QString :: indexOf 就是根据输入的字符，寻找字符串里面第一个出现的该字符，然后记录该字符在字符串中的位置

位置从 0 开始计算

QString 字符串截取

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

    QString s = "abcde,szaed,adsad";
    qDebug() << s;
    int index;
    index = s.indexOf(",");
    s = s.mid(0, index); //从字符串0开始向后数,
                         //数到index的位置，窃取前面的字符串
    qDebug() << s;
    return a.exec();
}
```



E:\QTtest\Qstr\Qstr-build-desktop-Q
"abcde,szaed,adsad"
"abcde"

QString :: mid 字符串截取函数

我们截取的是以第一个逗号结束，前面的字符串

QString 去掉字符串前后空格

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

    QString s = " abcde ss ";
    qDebug() << s;
    s = s.trimmed(); //去掉字符串前后空格
    qDebug() << s;
    return a.exec();
}
```

QString :: trimmed 字符串前后空格
去掉

应用程序输出

```
Qstr x
E:\QTtest\Qstr\Qstr-build-desktop-Q
" abcde ss "
"abcde ss"
```

这就是去除字符串前后空格对比

QString 字符串指定部分段截取

```
QString str = "aaaa,bbbb,cccc,dddd";
str = str.section(',', ',', 1, 3);
qDebug() << str;
```

```
{1 ?}
E:\QTtest\qmenutool\qt
"bbbb,cccc,dddd"
```

第 3 段结束，第 3
段就是 dddd

写入逗号就是以逗号
为分隔符来分割段落

从第 1 段开始，逗号分割的第
1 段就是 bbbb，第 0 段才是
aaaa

```
QString str = "aaaa,bbbb,cccc,dddd";
```

```
str = str.section(',', ',', 2, 2);
qDebug() << str;
```

```
{1 ?}
E:\QTtest\qt
"cccc"
```

cccc 就是开始第 2 段，结
束也是第 2 段，后面的段
落就不会进来了

如果你想要其中的某一段，那么就直接将
开始段落和结束段落写相同的就是了

QString:: length 字符串长度计算

```
QString str = "aaaa,bbbb,cccc,dddd";  
  
int num = str.length();  
qDebug () << num;
```

E:\QT\

19

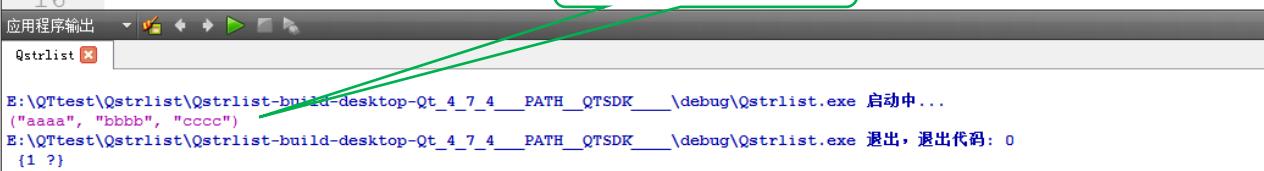
除去\0, str 字符串 19 个

QStringList 字符串(列表)/字符串数组使用

```
1 #include <QtGui/QApplication>  
2 #include "mainwindow.h"  
3 #include <QDebug>  
4 int main(int argc, char *argv[]){  
5     QApplication a(argc, argv);  
6     MainWindow w;  
7     QStringList list;  
8     list<<"aaaa"<<"bbbb"<<"cccc";//QStringList字符串数组只能这样初始化  
9  
10    qDebug () << list;  
11    w.show();  
12  
13    return a.exec();  
14 }  
15  
16 }
```

就是将多个单独" ... "的字符串放入 list

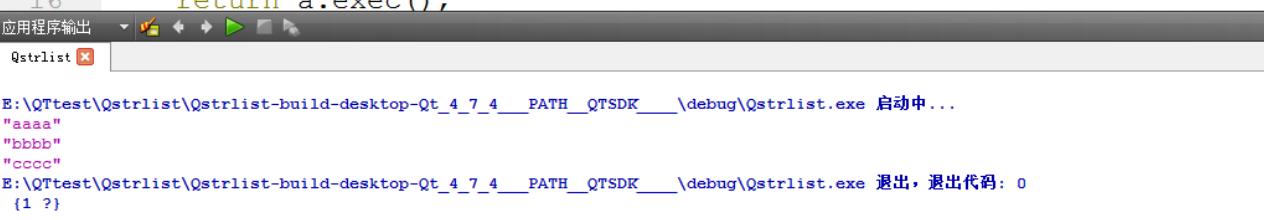
你看字符串数组就是这样



```
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 启动中...  
("aaaa", "bbbb", "cccc")  
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 退出, 退出代码: 0  
{1 ?}
```

```
4 int main(int argc, char *argv[]){  
5 {  
6     QApplication a(argc, argv);  
7     MainWindow w;  
8     QStringList list;  
9     list<<"aaaa"<<"bbbb"<<"cccc";//QStringList字符串数组只能这样初始化  
10  
11     qDebug () << list[0];  
12     qDebug () << list[1];  
13     qDebug () << list[2];  
14     w.show();  
15  
16     return a.exec();
```

如果想获取字符串数组中单组字符, 就使用数组下标



```
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 启动中...  
"aaaa"  
"bbbb"  
"cccc"  
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 退出, 退出代码: 0  
{1 ?}
```

```
6     QApplication a(argc, argv);
7     MainWindow w;
8     QStringList list;
9     list<<"aaaa"<<"bbbb"<<"cccc";
10    list.append("xiang");//append在字符串数组末尾插入新字符串
11    qDebug()<<list;
12
13    w.show();
14
15    return a.exec();
16 }
```

应用程序输出 Qstrlist

```
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 启动中...
("aaaa", "bbbb", "cccc", "xiang")
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 退出，退出代码：0
{1 ?}
```

```
6     QApplication a(argc, argv);
7     MainWindow w;
8     QStringList list;
9     list<<"aaaa"<<"bbbb"<<"cccc";
10    list.insert(0, "xiang");
11    list.insert(2, "zhi");
12    qDebug()<<list;
13
14    w.show();
15
16    return a.exec();
```

在字符串数组[0]位置插入字符串，以前的[0]位置的字符串 aaaa 向后移动 1 个元素，就是[1]

在字符串数组[2]位置插入字符串，以前的[2]位置的字符串 bbbb 向后移动 1 个元素，就是[3]

应用程序输出 Qstrlist

```
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 启动中...
("xiang", "aaaa", "zhi", "bbbb", "cccc")
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 退出，退出代码：0
{1 ?}
```

```
8     QStringList list;
9     list<<"aaaa"<<"bbbb"<<"cccc";
10    list.replace(1, "web");
11    qDebug()<<list;
12
13    w.show();
14
15    return a.exec();
16 }
```

替换字符串数组[1]位置 bbbb 为 web

应用程序输出 Qstrlist

```
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 启动中...
("aaaa", "web", "cccc")
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 退出，退出代码：0
{1 ?}
```

```

8
9
10 QStringList list;
11 list<<"aaaa"<<"bbbb"<<"cccc";
12 list.removeAll("bbbb"); //删除list里面的bbbb字符串
13 qDebug()<<list;
14
15 w.show();
16 return a.exec();
}

```

应用程序输出

Qstrlist

```

E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 启动中...
("aaaa", "cccc")
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 退出，退出代码：0
{1 ?}

```

就剩下 aaaa 和 bbbb 了

```

7
8
9
10 QStringList list;
11 QString str;
12 list<<"aaaa"<<"bbbb"<<"cccc";
13 str=list.join(",");
14 //取消字符串数组的 , 号分割符，将其合并成一个QString字符串
15 qDebug()<<str;
16
17 w.show();
}

```

应用程序输出

Qstrlist

```

E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 启动中...
"aaaa,bbbb,cccc"
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 退出，退出代码：0
{1 ?}

```

将 QString 的字符串拆分后合并成 QStringList 字符串数组

```

7
8
9
10
11
12
13
14
15
16
17

```

```

1 QStringList list;
2 QString str="xxxxx,zzzzz,bbbbbb";
3
4 list = str.split(",");
5 //将str里面的 , 号当做字符串的分割符，分割后各自组成带 "" 号的字符串
6 qDebug()<<list;
7 qDebug()<<list[0];
8 qDebug()<<list[1];
9 qDebug()<<list[2];
}

```

应用程序输出

Qstrlist

```

E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 启动中...
("xxxxx", "zzzzz", "bbbbbb")
"xxxxx"
"zzzzz"
"bbbbbb"

```

索引字符串在数组的位置

```
8     QTextEdit w,
9     int n;
10    QStringList list;
11    list<<"aaaa"<<"bbbb"<<"cccc";
12    n = list.indexOf("bbbb"); //list字符串数组的bbbb在1元素位置
13    qDebug()<<n;
14    n = list.indexOf("cccc"); //list字符串数组的cccc在2元素位置
15    qDebug()<<n;
16
17    w.show();
```

应用程序输出 Qstrlist

```
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 启动中...
1
2
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 退出，退出代码：0
{1 ?}
```

```
9     QStringList list;
10    list<<"aaaa"<<"bbbb"<<"cccc";
11
12    list.replaceInStrings("bbbb", "xxxx"); //将字符串数组bbbb替换成xxxx
13    qDebug()<<list;
14
15    w.show();
16
17    return a.exec();
```

应用程序输出 Qstrlist

```
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 启动中...
("aaaa", "xxxx", "cccc")
E:\QTtest\Qstrlist\Qstrlist-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qstrlist.exe 退出，退出代码：0
{1 ?}
```

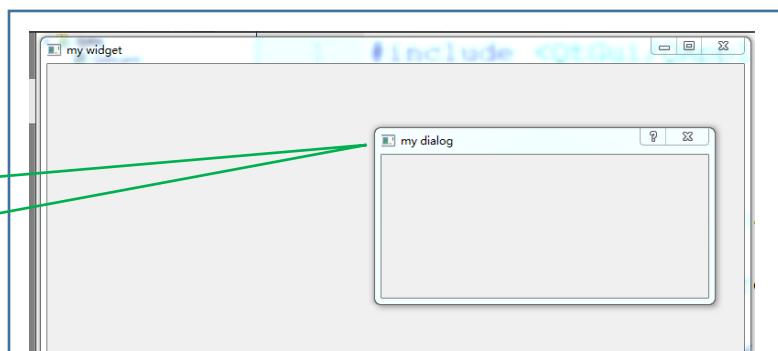
QT 中不依赖父窗口的对话框

```
#include <QtGui/QApplication>
#include<QWidget>
#include<QDialog>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget widget;
    QDialog dialog(&widget); //我们将dialog顶层对话框嵌入widget父窗口
    widget.setWindowTitle("my widget"); //设置widget窗口名字
    dialog.setWindowTitle("my dialog"); //设置dialog窗口名字
    widget.show();
    dialog.show();

    return a.exec();
}
```

这就是 QDialog 的作用，它创建的窗口随着父窗口打开而打开，但是它会独立出来，不会嵌入到父窗口上



用 QDialog 来实现模态对话框，模态对话框就点击按钮后再弹出的对话框

```
#include<QDialog>
#include<QPushButton>

class dio:public QDialog
{
    Q_OBJECT
protected:
    QPushButton button1;

protected slots:
    void button1send();

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

#endif // DIO_H
```

定义一个头文件来创建对话框类，但是这个类必须继承 QDialog，因为 QDialog 类里面才有对话框创建功能。我在主函数调用这个 dio 类创建对象的时候，这个 dio 子类会触发 QDialog 父类的构造函数被执行，这样我的按钮背景对话框就由 QDialog 创建出来了

在主函数执行 dio 子类创建对象时候 QDialog 父类对话框被创建，然后子类构造函数也被自动执行，创建了按钮，和信号与槽创建

```
#include<QPushButton>
#include<QWidget>
#include<QDebug>
#include "dio.h"
dio::dio(QWidget *parent):QDialog(parent),button1(this)
{
    button1.setText("button1");//创建按钮1
    button1.move(20,20);
    button1.resize(100,30);
    connect(&button1, SIGNAL(clicked()), this, SLOT(button1send()));
}

void dio::button1send()
{
    qDebug() << "button1";
}

dio::~dio()
{}
```

这里用列表方法把 button 对象嵌入进父窗口，this 就是指向父窗口，如果不懂，看看最前面是怎么把其余窗口嵌入进指定窗口的

```

#include <QtGui/QApplication>
#include<QWidget>
#include<QDialog>
#include "dio.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    dio dlg; //创建对话框
    dlg.show(); //调用 QDialog 类里面的显示函数显示
    return a.exec();
}

```



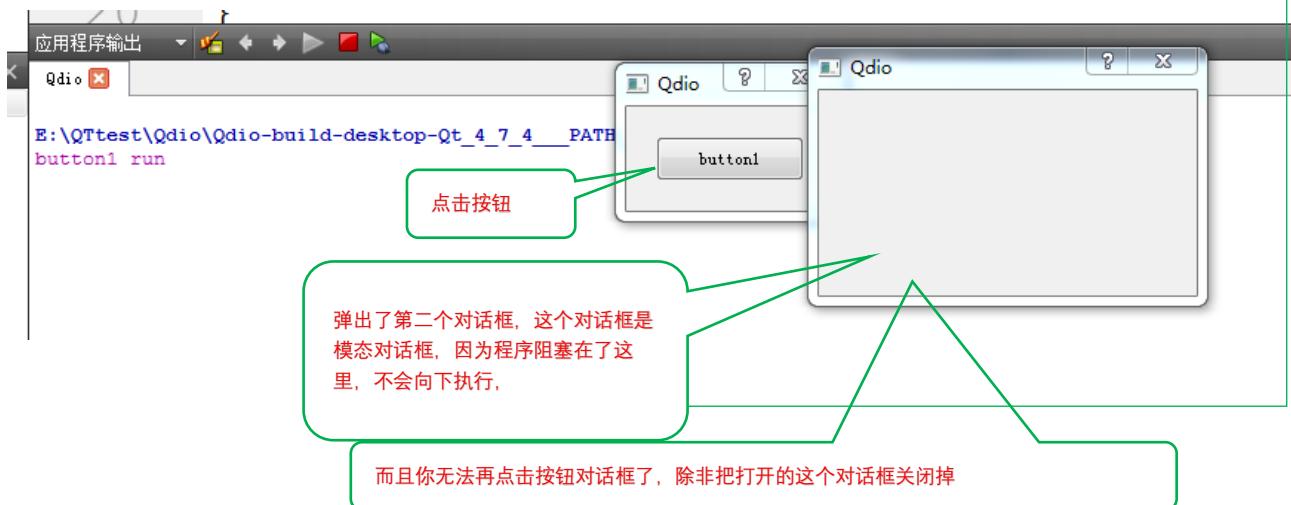
执行程序后我们 QDialog 对话框背景及按钮被创建。

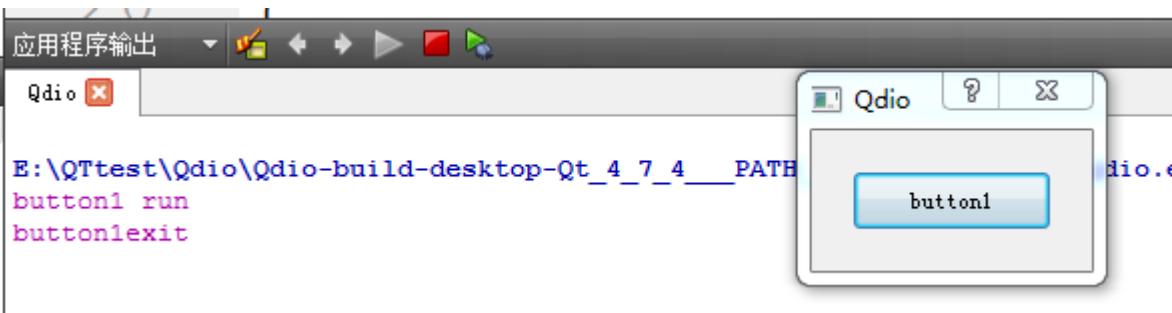
下面我们用阻塞方式点击按钮打开第二个 QDialog 对话框

```

void dio::button1send() //按钮点击后会执行这个函数
{
    qDebug() << "button1 run";
    QDialog dialog; //建立一个对话框类
    dialog.exec(); //阻塞对话框程序，程序就会卡在这里不动
    qDebug() << "button1exit";
}

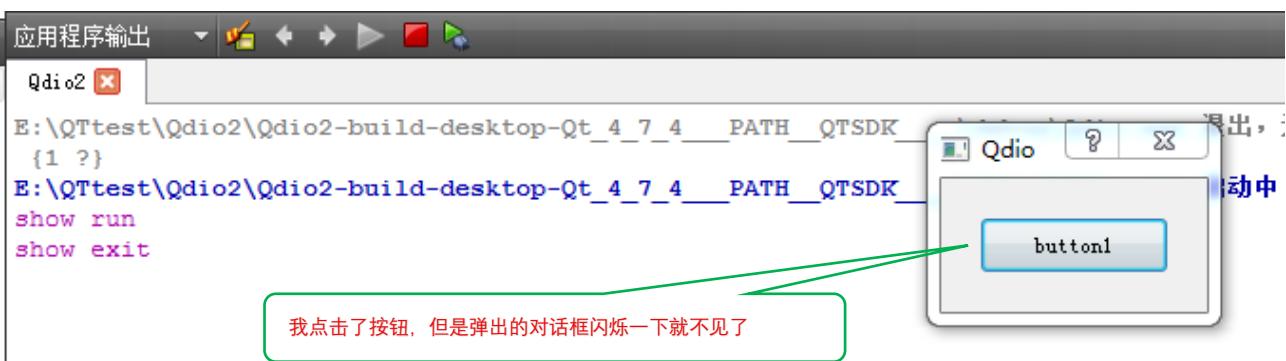
```





我将弹出的对话框关闭掉，才能再点击按钮对话框，这就是模态对话框让程序卡死在这里

```
void dio::button1send()
{
    qDebug() << "show run";
    QDialog dialog;
    dialog.show(); //非模态对话框，对话框打开后，程序继续向下执行
    qDebug() << "show exit";
}
```



这是因为 I 选择的是 show 非模态对话框，意思是程序不会阻塞在 show 哪里，它会受到按钮触发弹出第二个对话框，然后继续向下执行，button1send 函数执行完后，释放栈空间上的数据导致对话框消失

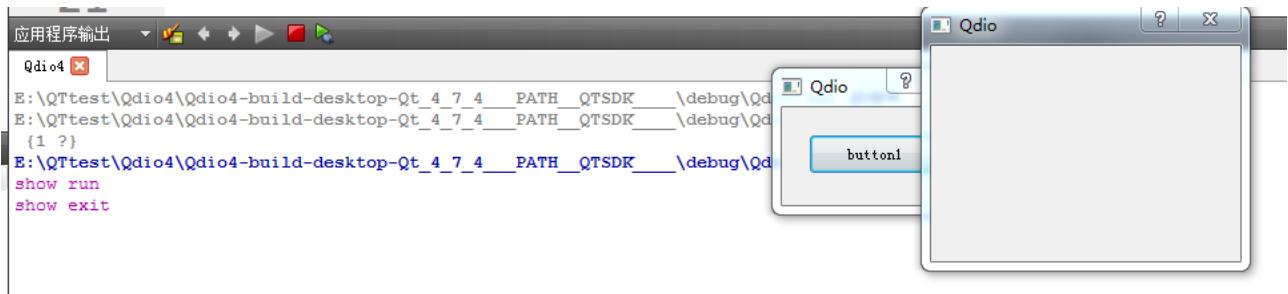
```
void dio::button1send()
{
    qDebug() << "show run";
    QDialog *dialog = new QDialog(); //非模态对话框建立在堆空间
    dialog->show();
    qDebug() << "show exit";
}
```





这是因为按钮对话框和弹出的非模态对话框不是父子关系。

```
void dio::button1send()
{
    qDebug() << "show run";
    QDialog *dialog = new QDialog(this); //将非模态对话框嵌入父对象,
                                         //也就是按钮背景对话框
    dialog->show();
    qDebug() << "show exit";
}
```



因为按钮对话框和非模态对话框是父子类关系，所以点击按钮对话框无法显示在非模态对话框前面。

现在还有个关键问题，就是 new 的 QDialog 对象没有跟随程序的结束自动释放堆内存

模态对话框有使用后的返回值，非模态对话框没有返回值

```
QDialog :: exec() ; //该函数就是 QDialog 类里面的对话框返回值函数  
void  QDialog::done(int i); //在对话框程序里面添加这个函数就可以返回值给 exec()函数  
QDialog :: Accepted //返回值表示用户操作成功  
QDialog :: Rejected //返回值表示用户操作失败
```

```
void dio::button1send()  
{  
    qDebug ()<<"show run";  
    done(Accepted);  
    qDebug ()<<"show exit";  
}
```

子函数向 QDialog::exec()函数的程序返回值

```
7 int main(int argc, char *argv[])  
8 {  
9     QApplication a(argc, argv);  
10    dio dlg;//创建对话框  
11    int ret = dlg.exec(); //返回QDialog::done函数的值  
12    if(ret == QDialog :: Accepted)  
13        qDebug ()<<"Accepted";  
14    else if(ret == QDialog :: Rejected)  
15        qDebug ()<<"Rejected";  
16    else  
17        qDebug ()<<ret;  
18  
19    return a.exec();  
20 }  
21
```



```
13 void dio::button1send()
14 {
15     qDebug() << "show run";
16     done(Rejected);
17     qDebug() << "show exit";
18 }
19
```

应用程序输出 Qdio6

```
E:\QTtest\Qdio6\Qdio6-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qdio.exe 启动中...
show run
show exit
Rejected
```

你看打印的返回值

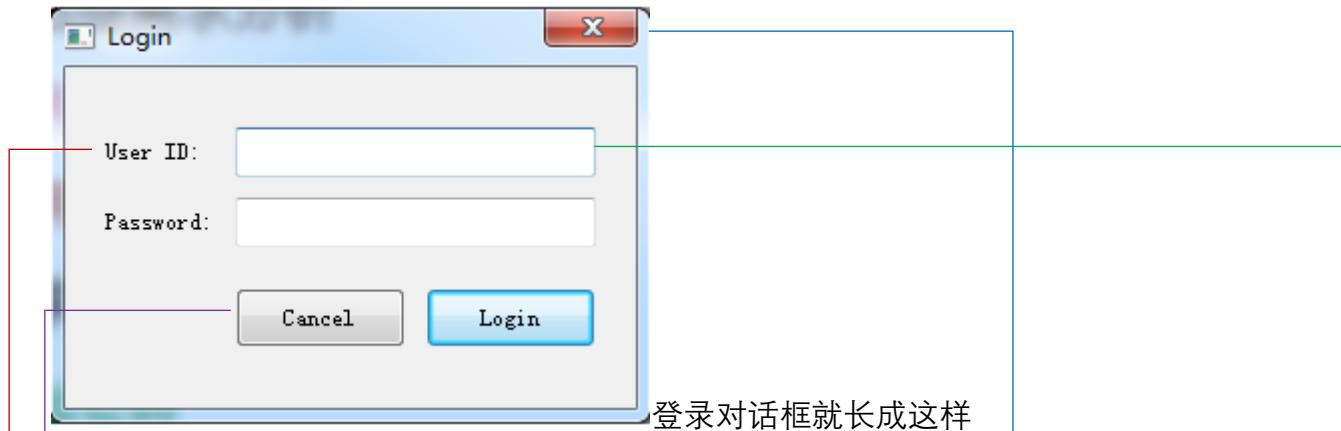
```
13 void dio::button1send()
14 {
15     qDebug() << "show run";
16     done(100);
17     qDebug() << "show exit";
18 }
19
```

应用程序输出 Qdio6

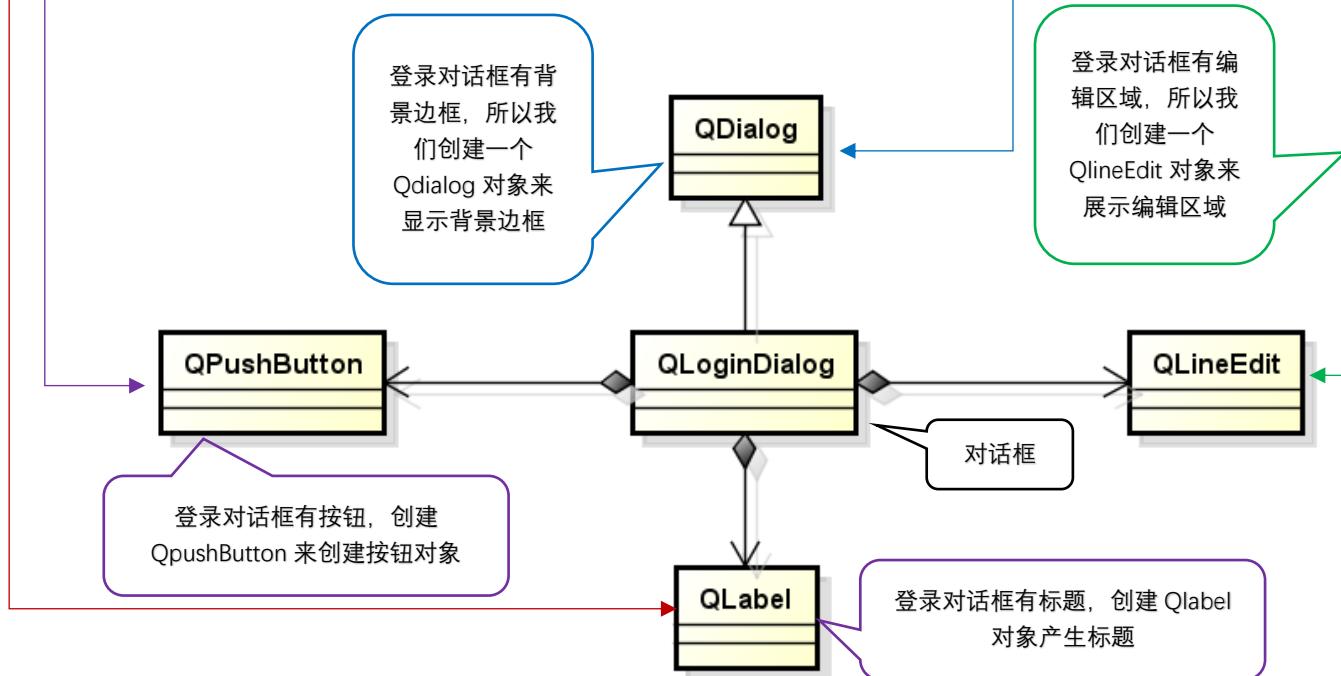
```
E:\QTtest\Qdio6\Qdio6-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\Qdio.exe 启动中...
show run
show exit
100
```

你看打印的返回值

登录对话框编程思想



登录对话框的设计与架构



```

#ifndef QLOGINDIALOG_H
#define QLOGINDIALOG_H
#include <QtGui/QDialog> //父窗口
#include <QLabel> //窗口标题
#include <QLineEdit> //编辑文本
#include <QPushButton> //按钮
class QloginDialog : public QDialog
{
    Q_OBJECT:
private:
    QLabel Username; //用户名
    QLabel Pasword; //密码
    QLineEdit UserEdit; //用户名编辑区域
    QLineEdit PaswEdit; //密码编辑区域
    QPushButton loginBtn; //登录
    QPushButton CancelBtn; //退出
};

```

所以我们要的窗口里面的按钮啊，这样框框，那样框框，只要是显示的部分，都可以在头文件先定义好对象。

```

#include <QLabel> //窗口标题
#include <QLineEdit> //编辑文本
#include <QPushButton> //按钮
#include "widget.h"
class QloginDialog : public QDialog
{

private:
    QLabel Username; //用户名
    QLabel Pasword; //密码
    QLineEdit UserEdit; //用户名编辑区域
    QLineEdit PaswEdit; //密码编辑区域
    QPushButton loginBtn; //登录
    QPushButton CancelBtn; //退出
public:
    QloginDialog(QWidget *parent = 0); //定义个回调函数
        /*这样以后想创建对话框就可以调用*/
        /*自己定义的QloginDialog*/
    ~QloginDialog();
};

#endif // QLOGINDIALOG_H

```

头文件把对话框界面的组件定义好后，现在在下面 CPP 文件实现头文件内容

```
#include "QLoginDialog.h"
#include <QDebug>
#include <QtGui/QDialog>
QloginDialog::QloginDialog(QWidget* parent) :
    QDialog(parent, Qt::WindowCloseButtonHint),
    Username(this),
    Pasword(this),
    UserEdit(this),
    PaswEdit(this),
    loginBtn(this),
    CancelBtn(this)
{
    Username.setText("User ID:");
    Username.move(20, 30);
    Username.resize(60, 25);

    UserEdit.move(85, 30);
    UserEdit.resize(180, 25);

    Pasword.setText("Password:");
    Pasword.move(20, 65);
    Pasword.resize(60, 25);
```

列表初始化对象，把对象组件嵌入进同一个父窗口

设置用户界面的标题栏

设置用户界面编辑框

设置密码界面标题栏

```

    PaswEdit.move(85, 65);           设置密码界面编辑框
    PaswEdit.resize(180, 25);
    PaswEdit.setEchoMode(QLineEdit::Password);

    CancelBtn.setText("Cancel");
    CancelBtn.move(85, 110);
    CancelBtn.resize(85, 30);          这句话是让密码输入时显示器
                                      用*号覆盖, 不要显示明文

    loginBtn.setText("Login");
    loginBtn.move(180, 110);          设置退出按钮
    loginBtn.resize(85, 30);          设置进入按钮

    setWindowTitle("Login");
    setFixedSize(285, 170);          设置窗口标题
}

QloginDialog::~QloginDialog()
{
}

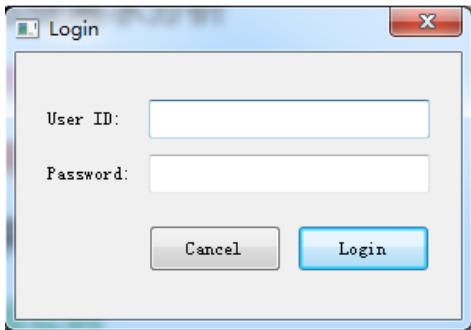
```

```

#include <QtGui/QApplication>
#include "widget.h"
#include "QLoginDialog.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QloginDialog dlg;             调用自己定义的类, 用构造函
                                数生成登录对话框
    dlg.show();

    return a.exec();
}

```



界面有了

为了让按钮按下有反应我们在头文件加入按钮的信号与槽函数

```
#include <QLabel>/>窗口标题
#include <QLineEdit>/>编辑文本
#include <QPushButton>/>按钮
#include <QWidget>
class QloginDialog:public QDialog
{
    Q_OBJECT
private:
    QLabel Username;//用户名
    QLabel Pasword;//密码
    QLineEdit UserEdit;//用户名编辑区域
    QLineEdit PaswEdit;//密码编辑区域
    QPushButton loginBtn;//登录
    QPushButton CancelBtn;//退出
private slots:
    void loginBtn_func();
    void CancelBtn_func();
public:
    QloginDialog(QWidget *parent = 0); //定义一个构造
    /*这样以后想
    /*自己定义的
    ~QloginDialog();
};

在原有的头文件里面加入 Q_OBJECT，这样才可以定义信号与槽
```

在原有的头文件里面加入 Q_OBJECT，这样才可以定义信号与槽

```
void loginBtn_func();
void CancelBtn_func();
```

定义信号接受函数

```
QloginDialog(QWidget *parent = 0); //定义一个构造
/*这样以后想
/*自己定义的
~QloginDialog();
};
```

在对应的 CPP 文件的构造函数下加入信号发送与信号接受的关联 api

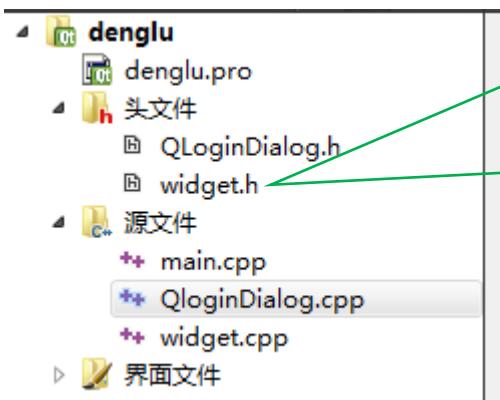
```
setWindowTitle("登录");
setFixedSize(285, 170);
connect(&loginBtn, SIGNAL(clicked()), this, SLOT(loginBtn_func()));
connect(&CancelBtn, SIGNAL(clicked()), this, SLOT(CancelBtn_func()));
}
```

在 CPP 文件下实现信号接受函数

```
void QloginDialog::loginBtn_func()
{
    qDebug() << "loginbtn";
}
void QloginDialog::CancelBtn_func()
{
    qDebug() << "cancelbtn";
}
```

然后编译运行

在编译运行的时候有可能发生错误，有可能不会发生错误



那是因为在创建工程的时候系统会自动帮你创建 widget 文件，但是这里面有 Q_OBJECT 类。
不知道是不是 widget 里面的 Q_OBJECT 和我的 QLoginDialog.h 里面的 Q_OBJECTY 类冲突了吗 ??
我把 widget 文件移除工程就正常了

```

42 -> QloginDialog::~QloginDialog()
43 {
44 }
45
46 -> void QloginDialog::loginBtn_func()
47 {
48     qDebug() << "loginbtn";
49 }
50 -> void QloginDialog::CancelBtn_func()
51 {
52     qDebug() << "cancelbtn";
53 }
54
55

```

应用程序输出

```

denglu
loginbtn
loginbtn
cancelbtn
cancelbtn

```

你看点击 login 或者 Cancel 就会在终端显示字符串
但是我不想点击按钮只产生一些字符串，我想来点实际的页面变化

```

void QloginDialog::loginBtn_func()
{
    qDebug() << "loginbtn";
    done(Accepted);
}
void QloginDialog::CancelBtn_func()
{
    qDebug() << "cancelbtn";
    done(Rejected);
}

```

在信号接受函数加入 done，你点击按钮对话框就会消失

```

void QloginDialog::loginBtn_func()
{
    qDebug() << "loginbtn";
    done(Accepted);
}
void QloginDialog::CancelBtn_func()
{
    qDebug() << "cancelbtn";
    done(Rejected);
}

```

而且这个 done 函数不仅可以关闭窗口，还返回值给 QloginDialog 创建的对象

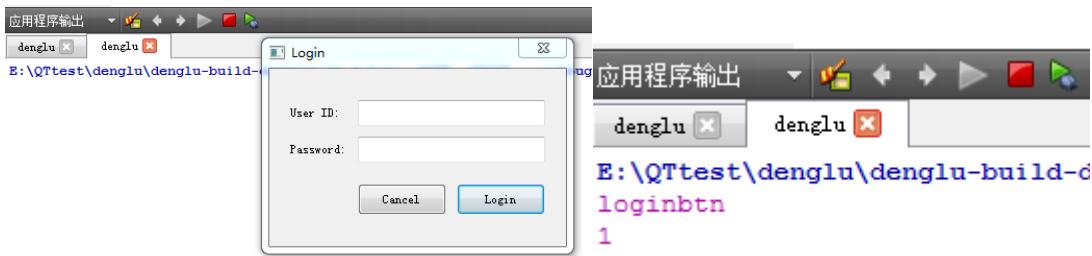
```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QloginDialog dlg;
    dlg.show();
    qDebug() << dlg.exec(); // 这里可以接受信号函数done的返回值

    return a.exec();
}

```

我们在主函数接受这个 dlg 对象信号槽函数的 done 返回值



按下 Login 按钮输出 1

其实这样的变化太单一了，我们看看点击按钮能不能生成模态对话框

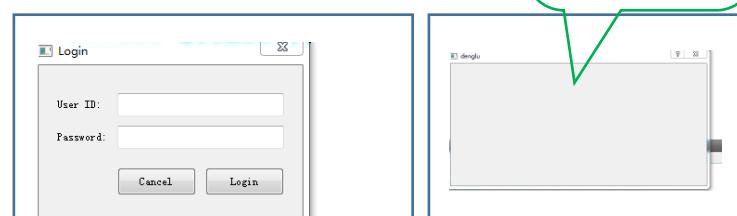
```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QloginDialog dlg;
    dlg.show();
    if(dlg.exec() == QloginDialog::Accepted)
    {
        QDialog mt;
        mt.exec();
    }
    return a.exec();
}

```

再定义一个模态对话框对象，
按钮点击就会生成模态对话框

但是有个问题，点击按钮
生成模态对话框，但是登录
对话框不见了？



```

void QloginDialog::loginBtn_func()
{
    qDebug() << "loginbtn";
    done(Accepted);
}
void QloginDialog::CancelBtn_func()
{
    qDebug() << "cancelbtn";
    done(Rejected);
}

```

所以只有把建立模态对话框的函数放入按钮消息接受函数

```

if (dlg.exec() == 
{
    QDialog mt;
    mt.exec();
}

```

```

void QloginDialog::loginBtn_func()
{
    qDebug() << "loginbtn";
    QDialog mt;
    mt.exec();
}

```

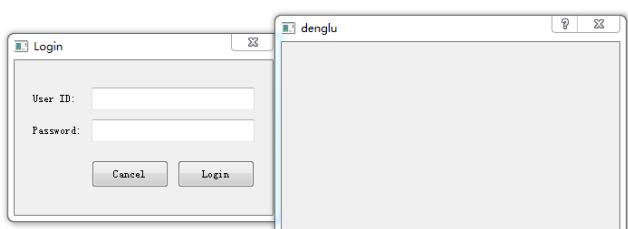
```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QloginDialog dlg;
    dlg.show();

    return a.exec();
}

```

主函数取消创建模态对话框函数，保持原函数不变



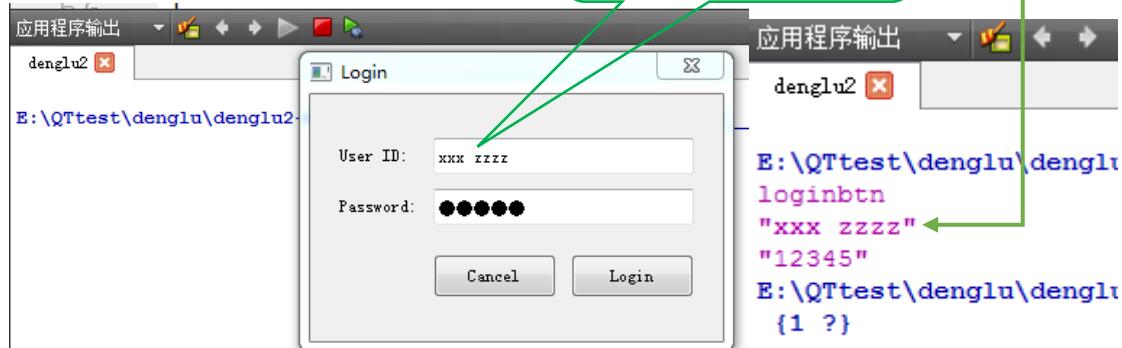
点击按钮产生了模态对话框，登录界面也在

这里有个问题，就是模态对话框静态对象的内存没有释放，注意！！注意！！

文本框组件(QLineEdit)里面输入的字符串获取

```
class QloginDialog : public QDialog
{
    Q_OBJECT
private:
    QLabel Username; //用户名
    QLabel Pasword; //密码
    QLineEdit UserEdit; //用户名编辑区域
    QLineEdit PaswEdit; //密码编辑区域
    QPushButton loginBtn; //登录
    QPushButton CancelBtn; //退出
    QString uservalue; //存放输入框的用户名字符串
    QString passvalue; //存放输入框的密码字符串
private slots:
    void loginBtn_func();
    void CancelBtn_func();
}

void QloginDialog::loginBtn_func()
{
    qDebug() << "loginbtn";
    uservalue = UserEdit.text(); //获取用户编辑框的字符串，空格也获取
    passvalue = PaswEdit.text(); //获取密码编辑框的字符串
    qDebug() << uservalue;
    qDebug() << passvalue;
    done(Accepted);
}
```

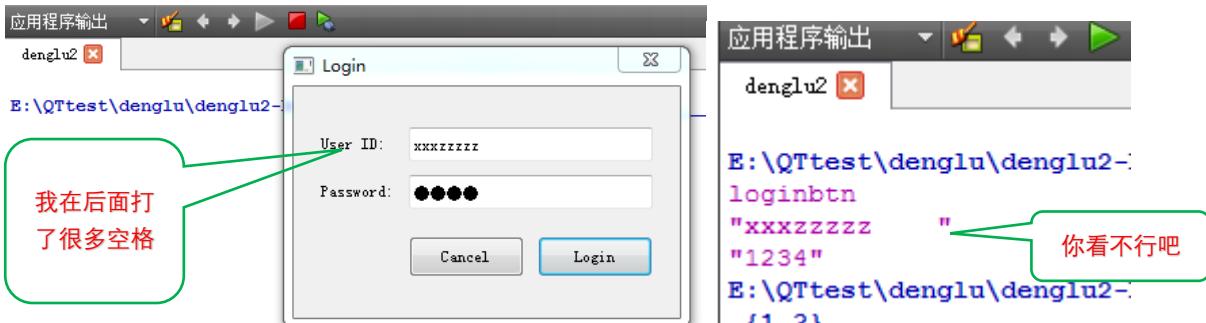


字符串中间的空格是去不掉的，只有去除字符串前后的空格

```

void QloginDialog::loginBtn_func()
{
    qDebug() << "loginbtn";
    uservalue = UserEdit.text();
    passvalue = PaswEdit.text(); //获取密码编辑框的字符串

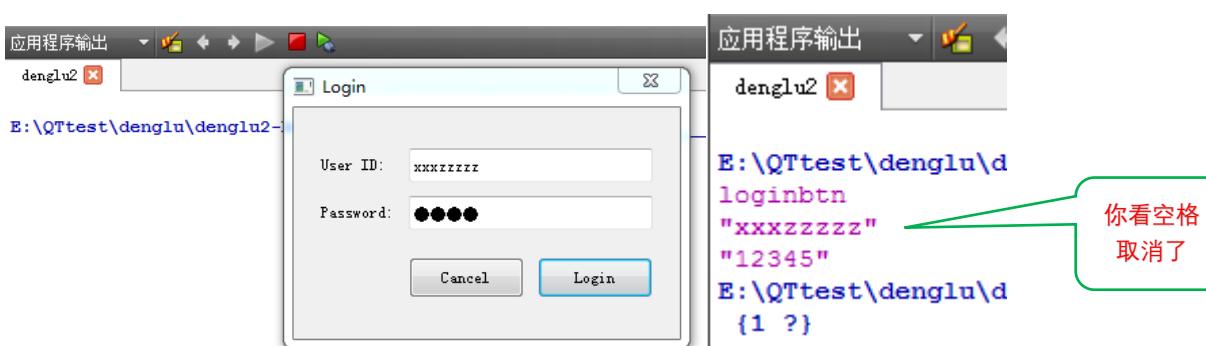
```



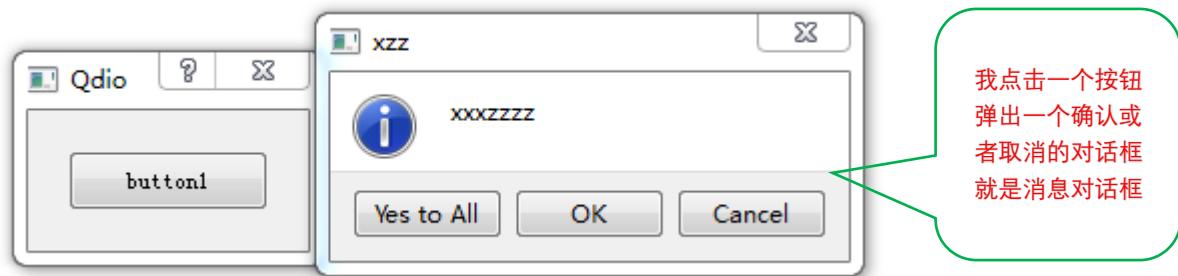
```

void QloginDialog::loginBtn_func()
{
    qDebug() << "loginbtn";
    uservalue = UserEdit.text().trimmed(); //trimmed是去掉字符串前后的空格API
    passvalue = PaswEdit.text(); //获取密码编辑框的字符串

```



QT 消息对话框实现



```

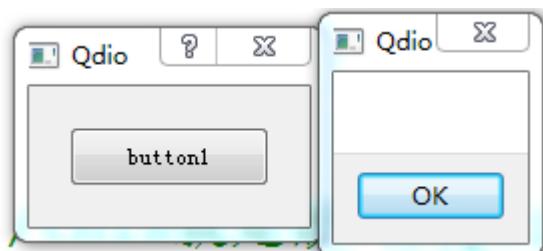
#ifndef DIO_H
#include<QWidget>
#include<QDebug>
#include<QMessageBox> //消息对话框头文件
#include "dio.h"

```

```

void dio::button1send()
{
    QMessageBox msg(this); //this就是嵌入到QWidget父窗口
    msg.exec();
}

```



就在已有的按钮信号接受函数里面建立消息对话框

```

connect(&button1, SIGNAL(clicked()), this, SLOT(button1send()));
}

void dio::button1send()
{
    QMessageBox msg(this); //this就是嵌入到QWidget父窗口
    msg.setWindowTitle("xzz");
    msg.setText("xxxxzzz");
    msg.setIcon(QMessageBox::Information);
    msg.exec();
}

```

按钮是有了，但是缺少标题名字

对话框标题

对话框文本

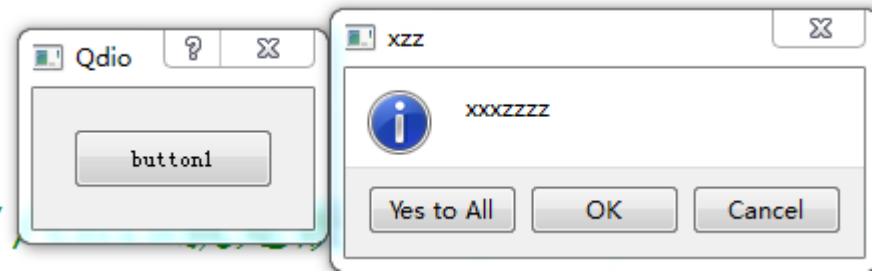
对话框图标

```

void dio::button1send()
{
    QMessageBox msg(this); //this就是嵌入到QWidget父窗口
    msg.setWindowTitle("xzz");
    msg.setText("xxxxzzz");
    msg.setIcon(QMessageBox::Information);
    msg.setStandardButtons(QMessageBox::Ok | QMessageBox::Cancel | QMessageBox::YesToAll);
    msg.exec();
}

```

多少个或符号就是给对话框建立多少个按钮



对话框按钮有了

按钮按下需要有处理程序

```
void dio::button1send()
{
    QMessageBox msg(this); //this就是嵌入到QWidget父窗口
    msg.setWindowTitle("xzz");
    msg.setText("xxxxzzz");
    msg.setIcon(QMessageBox::Information);
    msg.setStandardButtons(QMessageBox::Ok | QMessageBox::Cancel | QMessageBox::YesToAll);
    if(msg.exec() == QMessageBox::Ok)
    {
        qDebug() << "OK";
    }
    if(msg.exec() == QMessageBox::Cancel)
    {
        qDebug() << "Cancel";
    }
}
```

我们经过对象返回值来确定是对话框某个按钮按下，然后去执行对应的函数



我按下 OK 执行了打印 OK 的函数

颜色对话框，设置窗口颜色

QcolorDialog 这是颜色对话框类

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QPushButton>
#include <QColorDialog>
#include <QInputDialog>
namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();

private:
    Ui::Widget *ui;
    QPushButton ColorDialogBtn; // 定义个按钮
private slots:
    void ColorDialogBtn_Clicked(); // 定义按钮按下的信号接受函数
};

#endif // WIDGET_H
```

颜色设置函数头文件

定义个按钮

定义按钮按下的信号接受函数

头文件函数实现

```
#include "widget.h"
#include "ui_widget.h"

Widget::Widget(QWidget *parent) : QWidget(parent), ColorDialogBtn(this)
{
    ColorDialogBtn.setText("Color Dialog");
    ColorDialogBtn.move(20, 20);
    ColorDialogBtn.resize(160, 30);
    connect(&ColorDialogBtn, SIGNAL(clicked()), this, SLOT(ColorDialogBtn_Clicked()));
}

void Widget::ColorDialogBtn_Clicked()
{
    qDebug() << "enter color";
    QColorDialog dlg(this); //建立颜色对话框对象，嵌入父窗口
    dlg.setWindowTitle("xxxxzz"); //对话框名称
    dlg.setCurrentColor(Qt::red); //对话框初始颜色
    dlg.exec(); //要调用这个函数颜色对话框才会在屏幕显示出来
}

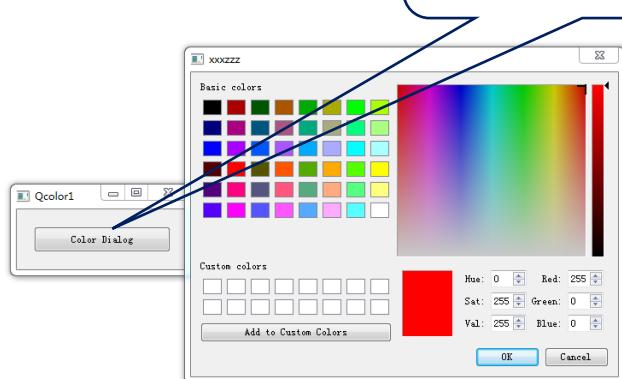
Widget::~Widget()
{
    delete ui;
}
```

按钮按下后建立颜色对话框

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();
    return a.exec();
}
```

因为这是模态对话框，所以对话框
打开后后面按钮就不能按下了

主函数完成，执行



现在有个问题，我点击 OK 后没有获取对话框颜色设置的值

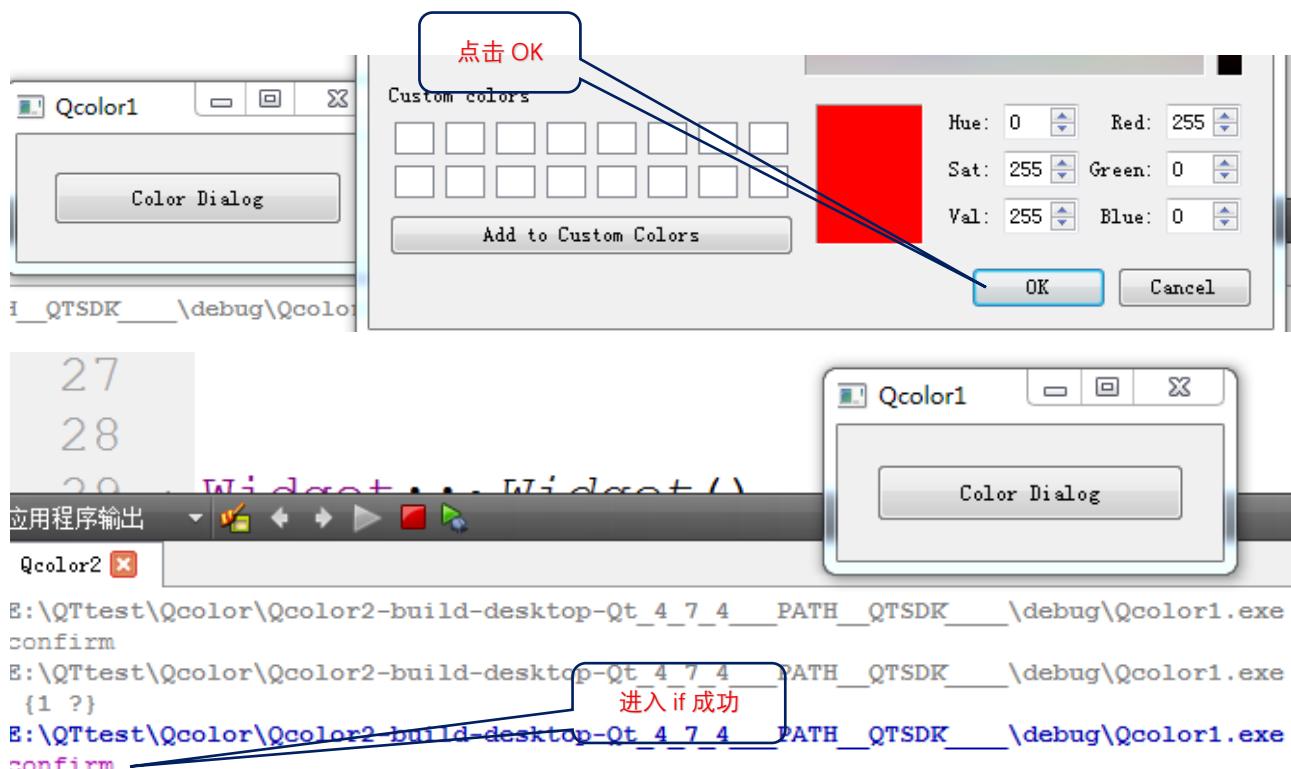
```

void Widget::ColorDialogBtn_Clicked()
{
    QColorDialog dlg(this);
    dlg.setWindowTitle("xxxzzz");
    dlg.setCurrentColor(Qt::red);

    if(dlg.exec() == QColorDialog::Accepted)
    {
        qDebug() << "confirm";
    }
}

```

修改信号接受函数，模态对话框是阻塞方式，所以我点击 OK 这里程序执行通过，然后会进入 if 函数



我们就使用进入 if 程序后的方法，点击确定时的颜色值

```

void Widget::ColorDialogBtn_Clicked()
{
    QColorDialog dlg(this);
    dlg.setWindowTitle("xxxxxx");
    dlg.setCurrentColor(Qt::red);

    if(dlg.exec() == QColorDialog::Accepted)
    {
        qDebug() << "confirm";
        QColor color = dlg.selectedColor();
        qDebug() << color; // 获取颜色分量 (ARGB)，才有0~1的范围表示颜色

        qDebug() << color.red(); // 获取十进制红色分量
        qDebug() << color.green(); // 获取十进制绿色分量
        qDebug() << color.blue(); // 获取十进制蓝色分量
        qDebug() << color.hue();
        qDebug() << color.saturation();
        qDebug() << color.value();
    }
}

```

修改信号接受函数，定义一个颜色类 color，该类用来获取颜色

将 RGB 用 10 进制打印出来

这是获取 6 角参数



出代码：0
出代码：0

```

confirm
QColor(ARGB 1, 1, 0, 0)
255
0
0
0
255
255

```

你看打印的和设置的参数一样

这就是颜色对话框使用方法。

输入对话框，就是你给对话框输入字符，然后代码可以获取字符

QInputDialog 类就是输入对话框类

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QPushButton>
#include <QColorDialog>
#include <QInputDialog>
namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();

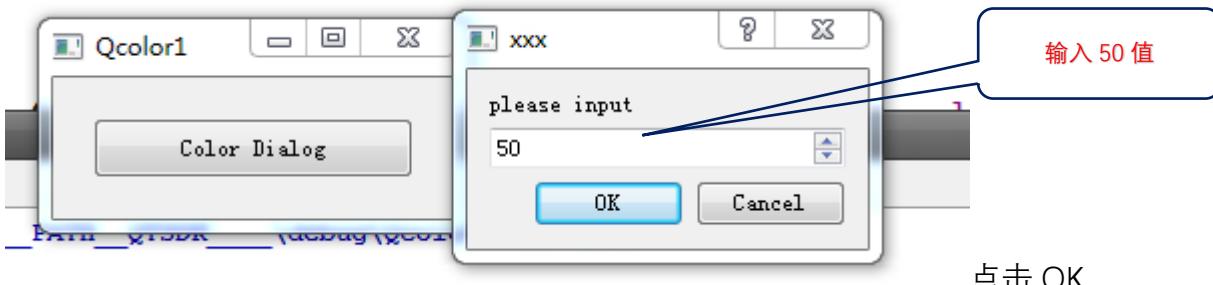
private:
    Ui::Widget *ui;
    QPushButton InputDialogBtn;//定义按钮
private slots:
    void InputDialogBtn_Clicked();//按钮信号接受函数
};

#endif // WIDGET_H
```

按钮函数自己实现，我下只给出在按钮信号接受函数里面创建输入对话框

```
void Widget::InputDialogBtn_Clicked()
{
    qDebug() << "input";
    QInputDialog input;
    input.setWindowTitle("xxx");//设置窗口标题
    input.setLabelText("please input");
    input.setInputMode(QInputDialog::IntInput);//IntInput就是设置输入必须是整形数
                                                //像字符串这些输入就不行

    if (input.exec() == QInputDialog::Accepted) //如果用户按下确定按钮
    {
        qDebug() << input.intValue(); //接受对话框输入的值
    }
}
```

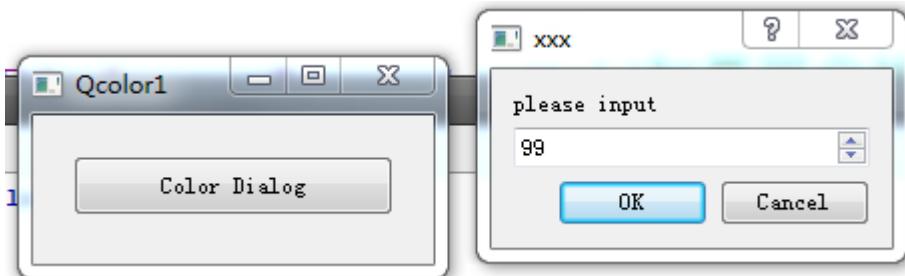


点击 OK

```
应用程序输出
Qcolor1
E:\QTtest\Qcolor
input
50
```

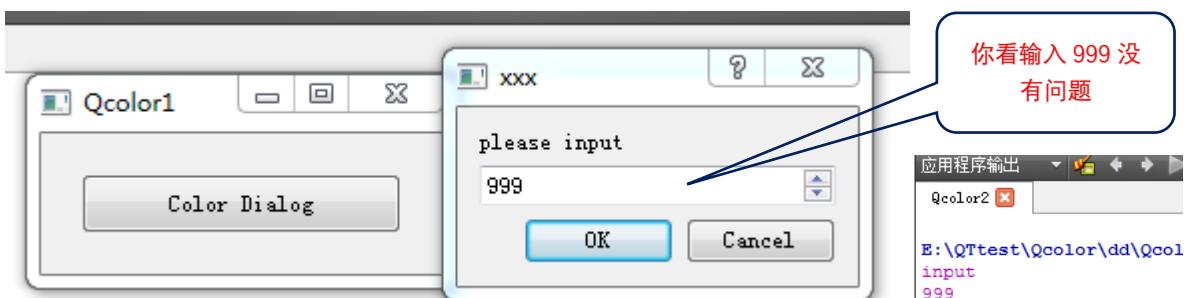
程序就接受了 50 的整形数据

这里有个问题……..



数字最多输入到 99，这是因为你没有设置输入整形数据的返回

```
void Widget::InputDialogBtn_Clicked()
{
    qDebug() << "input";
    QInputDialog input;
    input.setWindowTitle("xxx"); //设置窗口标题
    input.setLabelText("please input");
    input.setInputMode(QInputDialog::IntInput); //IntInput就是设置输入必须是整形数
                                                //像字符串这些输入就不行
    input.setIntMinimum(0); //设置允许输入的最小值
    input.setIntMaximum(1000); //设置允许输入的最大值
    if (input.exec() == QInputDialog::Accepted) //如果用户按下确定按钮
    {
        qDebug() << input.intValue();
    }
}
```



```
应用程序输出
Qcolor2
E:\QTtest\Qcolor\dd\Qcolor
input
999
```

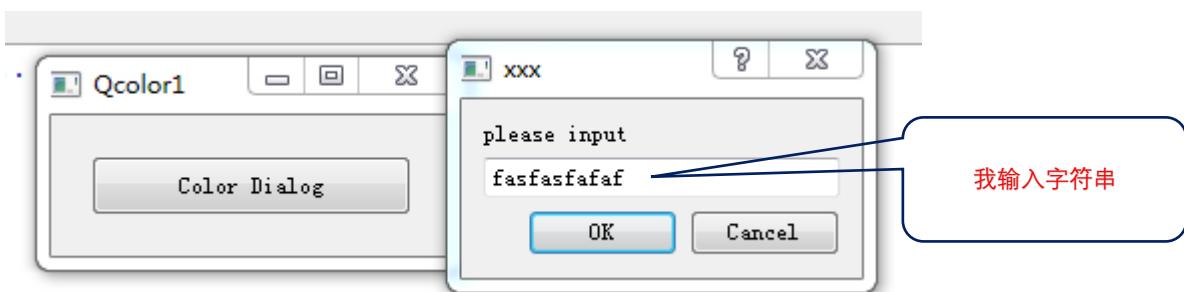
所以输入整形最大范围可以根据实际情况来修改，反正不要超过 int 范围就行。
我现在想输入字符串给程序怎么办？

```

void Widget::InputDialogBtn_Clicked()
{
    qDebug() << "input";
    QInputDialog input;
    input.setWindowTitle("xxx"); //设置窗口标题
    input.setLabelText("please input");
    input.setInputMode(QInputDialog::TextInput); //接受字符串输入模式

    if (input.exec() == QInputDialog::Accepted) //如果用户按下确定按钮
    {
        qDebug() << input.textValue();
    }
}

```



应用程序输出

```

Qcolor3
E:\QTtest\Qcolor\
input
"fasfasfafaf"

```

你看程序就接受字符串了

字体设置对话框

QFontDialog 字体大小设置对话框类

```

#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QPushButton>
#include <QFontDialog> //添加字体头文件
namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();

private:
    Ui::Widget *ui;
    QPushButton QFontbutton; //定义按钮
private slots:
    void QFontbutton_Clicked(); //按钮信号接受函数
};

#endif // WIDGET_H

```

定义字体设置类的头文件

定义按钮

定义消息处理函数

按钮显示自己实现就行了。

我现在直接写消息处理函数

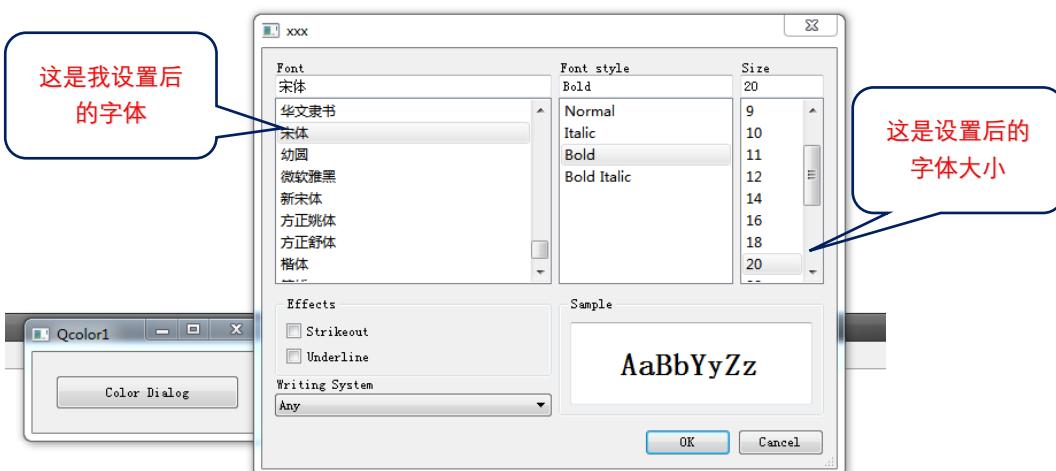
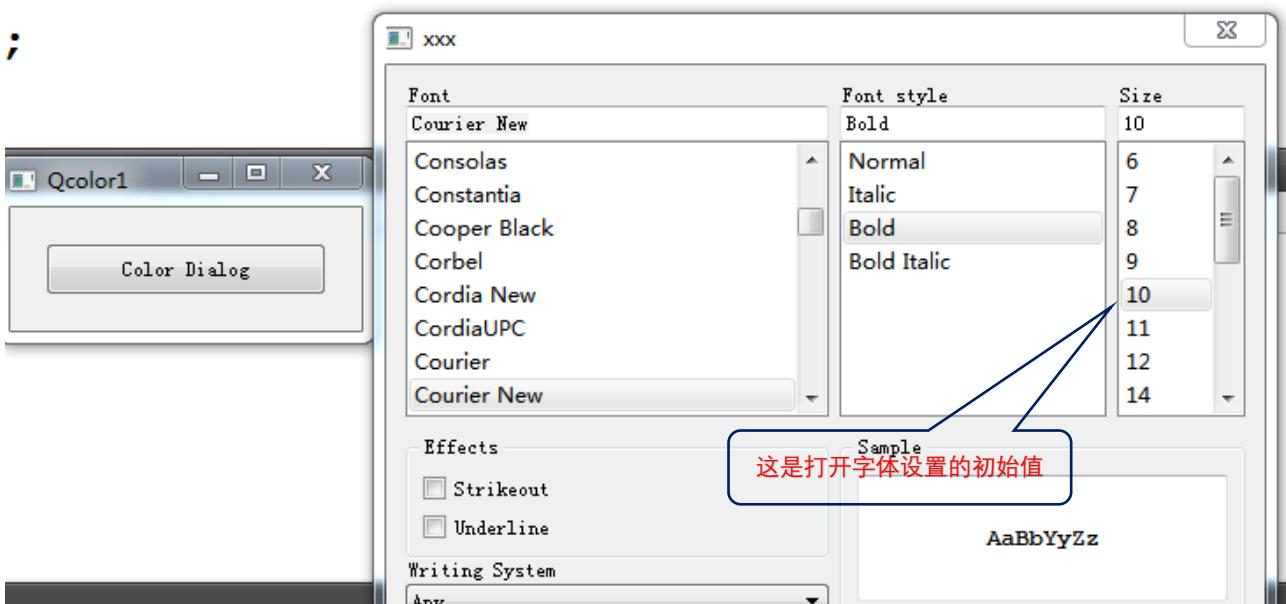
```

void Widget::Qfontbutton_Clicked()
{
    qDebug() << "Qfont";
    QFontDialog font;
    font.setWindowTitle("xxx"); //设置对话框名
    font.setCurrentFont(QFont("Courier New", 10, QFont::Bold)); //初始打开对话框的默认选择

    if(font.exec() == QFontDialog::Accepted) //如果用户按下确定按钮
    {
        qDebug() << font.selectedFont();
    }
}

```

设置初始字体
设置初始字体大小
设置初始字体颜色
获取你设置字体属性的参数



```

Qfont X
E:\QTtest\font\Qfont-build-desktop-Qt_4_
Qfont
QFont( "宋体,20,-1,5,75,0,0,0,0,0" )

```

你看获取的值和我设置的一模一样

进度条对话框

QProgressDialog 进度条对话框类

```
#include <QProgressDialog> //进度条对话框
namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();

private:
    Ui::Widget *ui;
    QPushButton Progress; //定义按钮
private slots:
    void Progress_Clicked(); //按钮信号接受函数
};

#endif // WIDGET_H
```

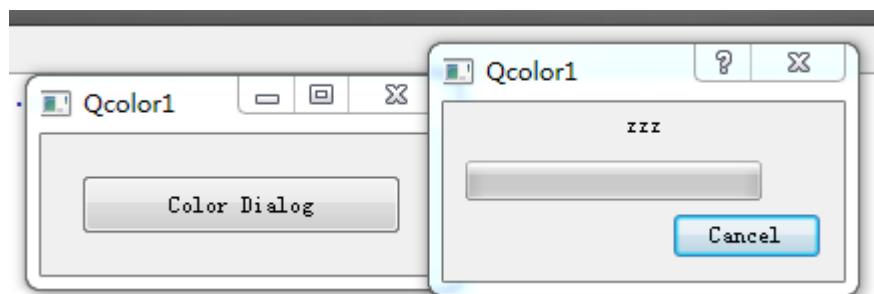
按钮显示函数自己去实现

```
void Widget::Progress_Clicked()
{
    qDebug() << "Progress";
    QProgressDialog prog;

    prog.setWindowTitle("xxx"); //进度条对话框名
    prog.setLabelText("zzz"); //进度条对话框里面的提示信息
    prog.setMinimum(0); //设置进度条最小值
    prog.setMaximum(1000); //设置进度条最大值

    prog.exec();
}
```

进度条对话框在按钮消息
处理函数中创建



这就是进度条对话框，怎么没有进度条进度值呢？

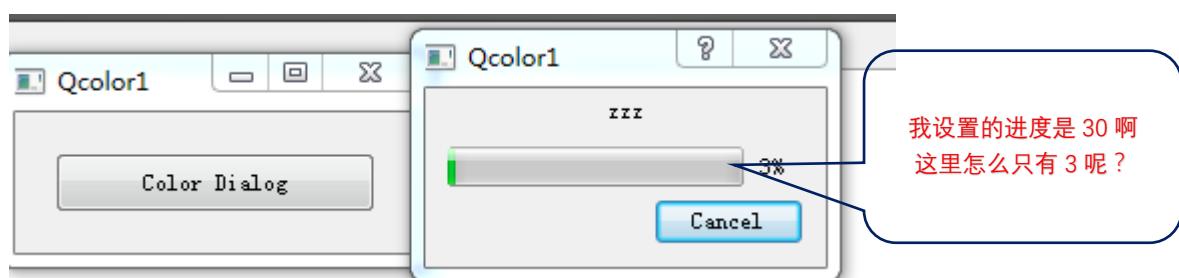
```

void Widget::Progress_Clicked()
{
    qDebug() << "Progress";
    QProgressDialog prog;

    prog.setWindowFilePath("xxx"); //进度条对话框名
    prog.setLabelText("zzz"); //进度条对话框里面的提示信息
    prog.setMinimum(0); //设置进度条最小值
    prog.setMaximum(1000); //设置进度条最大值
    prog.setValue(30); //设置进度条进度值
    prog.exec();
}

```

用 setValue 设置进度条值



```

prog.setMinimum(0); //设置进度条最小值
prog.setMaximum(1000); //设置进度条最大值
prog.setValue(30); //设置进度条进度值

```

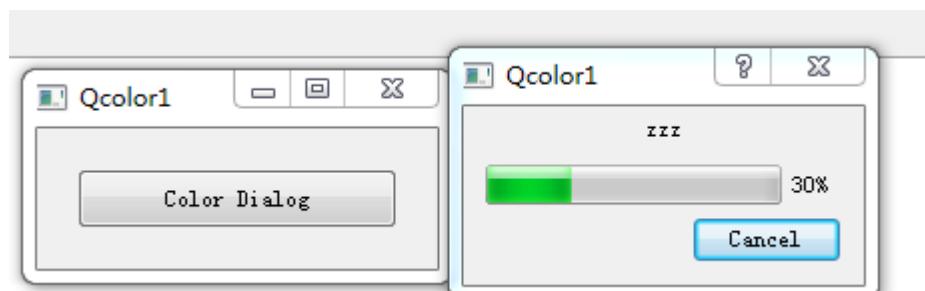
这是因为你进度条最大值设置的是 1000,
1000 的 30 就是 3%

```

prog.setWindowFilePath("xxx"); //进度条对话框名
prog.setLabelText("zzz"); //进度条对话框里面的提示信息
prog.setMinimum(0); //设置进度条最小值
prog.setMaximum(100); //设置进度条最大值
prog.setValue(30); //设置进度条进度值

```

修改到 100



你看是不是 30% 了

打印对话框，让你的软件有使用打印机的功能，或者直接输出 pdf 功能

QprintDialog 类就是打印对画框类

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QPushButton>
#include <QPrintDialog> // 打印对话框
namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();

private:
    Ui::Widget *ui;
    QPushButton Progress; // 定义按钮
private slots:
    void Progress_Clicked(); // 按钮信号接受函数
};

#include "ui_widget.h"

Widget::Widget(QWidget *parent) : QWidget(parent), Progress(this)
{
    Progress.setText("Color Dialog");
    Progress.move(20, 20);
    Progress.resize(160, 30);
    connect(&Progress, SIGNAL(clicked()), this, SLOT(Progress_
    ));

    void Widget::Progress_Clicked()
    {
        QPrintDialog prt(this);
        prt.setWindowTitle("xxxxxx"); // 打印对话框名字

        if(prt.exec() == QPrintDialog::Accepted) // 生成模态的打印对话框
        {
        }
    }
}
```

头文件包含打印对话
框类

点击按钮后出现打印
对话框



打印对话框生成成功，但是我点击打印什
么东西都没有输出，比如 pdf 文件

QPrinter 增加系统打印设备类

QTextDocument 增加文档类，一个对象表示一个文档

```
#include <QPrintDialog> // 打印对话框  
#include <QPrinter> // 这是系统打印设备的接口  
#include <QTextDocument> // 文档类接口
```

```
namespace Ui {  
    class Widget;  
}
```

```
class Widget : public QWidget  
{  
    Q_OBJECT
```

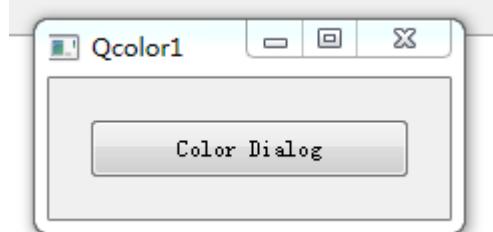
```
void Widget::Progress_Clicked()  
{  
    QPrintDialog prt(this);  
    prt.setWindowTitle("xxxxzz");  
  
    if(prt.exec()==QPrintDialog::Accepted)  
    {  
        QPrinter* p = prt.printer(); // 指定打印设备执行哪一个打印对话框的命令  
        QTextDocument td; // 建立一个文本文档  
        td.setPlainText("xxxxxxxxzzzzzzzz"); // 向这个文本文档写内容  
        p->setOutputFileName("D:\\text.pdf"); // p对象打印机打印的pdf文件存放路径  
        td.print(p); // 将文本文档放入p对象打印机，然后打印出来  
    }  
}
```

增加打印机与打印对话框关联，
然后打印创建的文本文档

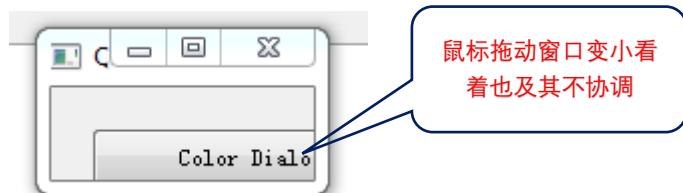


布局管理器解决窗口无法自适应问题

QVBoxLayout 这是布局管理器类



这是一个父窗口下创建的按钮



为了解决这个问题我们用布局管理器

```
#include <QPushButton>
#include <QDebug>
#include <QVBoxLayout> // 垂直布局管理器头文件
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;

    QPushButton button1(&w);
    QPushButton button2(&w);
    QPushButton button3(&w);
    QPushButton button4(&w);
    QVBoxLayout* vlayout = new QVBoxLayout(); // 创建一个垂直对称的布局管理器

    button1.setText("button1");
    button2.setText("button2");
    button3.setText("button3");
    button4.setText("button4");
    vlayout->addWidget(&button1); // 将创建的按钮放入布局管理器
    vlayout->addWidget(&button2);
    vlayout->addWidget(&button3);
    vlayout->addWidget(&button4);

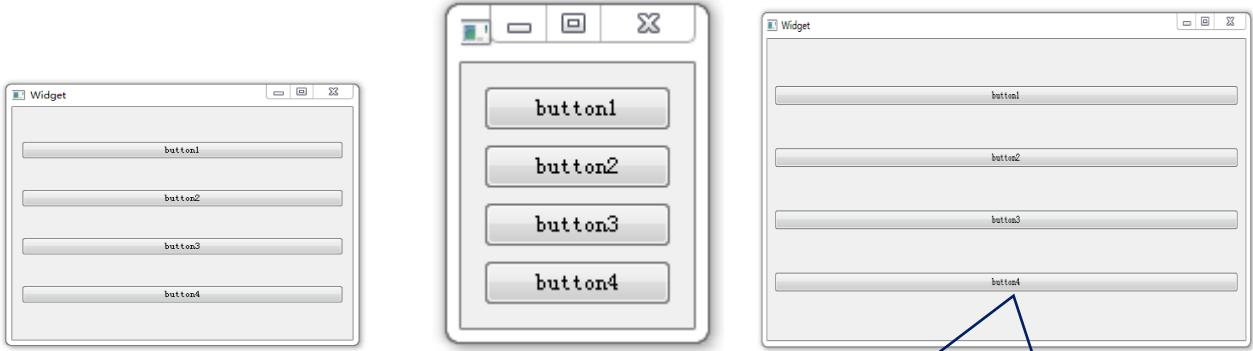
    w.setLayout(vlayout); // 因为按钮的父窗口是Widget，所以用Widget的布局功能设置布局管理器vlayout
    w.show();

    return a.exec();
}
```

第1步

第2步

第3步



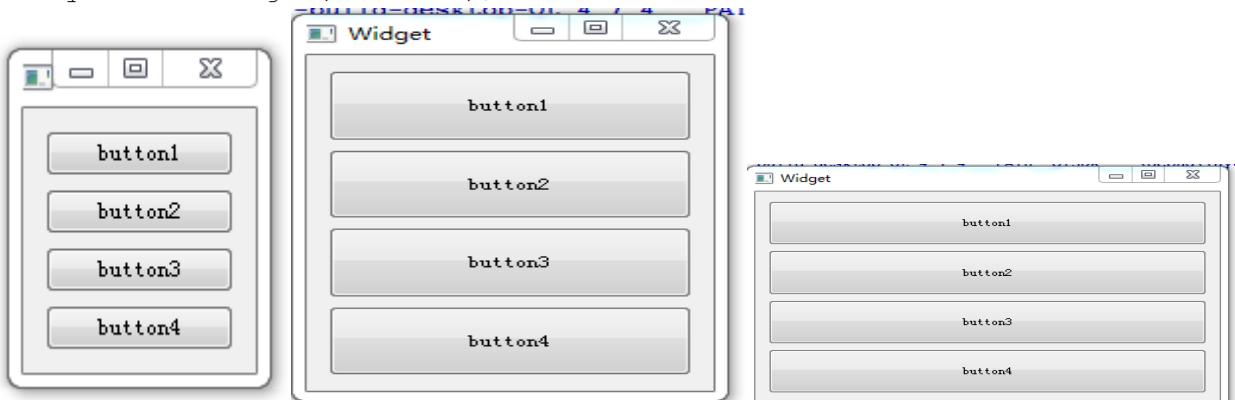
你看不管窗口缩小还是扩大都是正常对称的。

现在有个问题，就是我的按钮随着窗口横向对称变化，但是纵向，按钮宽度没有变化

我们要设置按钮本身属性来解决这个问题

`setSizePolicy` 设置按钮的自动调整属性

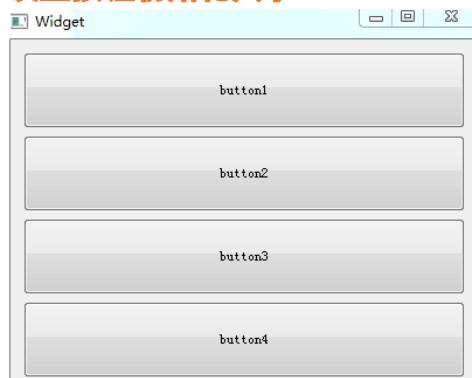
```
button1.setText("button1");
button1.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
//setSizePolicy按钮水平垂直自动调整
button2.setText("button2");
button2.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button3.setText("button3");
button3.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button4.setText("button4");
button4.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
vlayout->addWidget(&button1);
```



你看按钮随着窗口的变化，对称的变化

但是我想设置按钮初始化的大小怎么办？

设置按钮初始化大小



按钮初始化大小，最小默认是 160*30

不能设置比 160*30 更小的按钮大小

我现在设置 200*200

setMinimumSize 函数设置按钮初始化大小

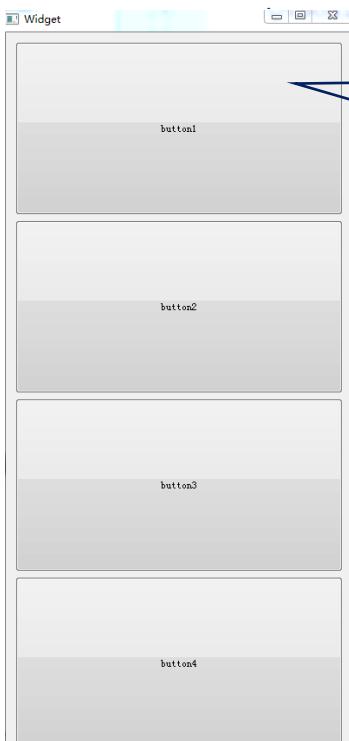
```
button1.setText("button1");
button1.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button1.setMinimumSize(200, 200); //设置按钮初始化大小

button2.setText("button2");
button2.setMinimumSize(200, 200);
button2.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

button3.setText("button3");
button3.setMinimumSize(200, 200);
button3.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

button4.setText("button4");
button4.setMinimumSize(200, 200);
button4.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
```

增加设置按钮大小函数



这就是按钮初
始化大小

```
Widget w;

QPushbutton button1(&w);
QPushbutton button2(&w);
QPushbutton button3(&w);
QPushbutton button4(&w);
QVBoxLayout* vlayout = new QVBoxLayout();

button1.setText("button1");
button1.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button2.setText("button2");
button2.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button3.setText("button3");
button3.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button4.setText("button4");
button4.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

vlayout->setSpacing(30); //设置布局管理器里按钮之间的间隔
vlayout->addWidget(&button1);
vlayout->addWidget(&button2);
vlayout->addWidget(&button3);
vlayout->addWidget(&button4);

w.setLayout(vlayout);
w.show();
```

setSpacing 设置布局管理器
模式下按钮之间的间隔



你看按钮间隔 30 个像素

横向布局管理器

QHBoxLayout 水平布局管理器类

```
#include <QPushButton>
#include <QDebug>
#include <QHBoxLayout>/>//水平布局管理器头文件
#include "widget.h"

QApplication a(argc, argv);
Widget w;

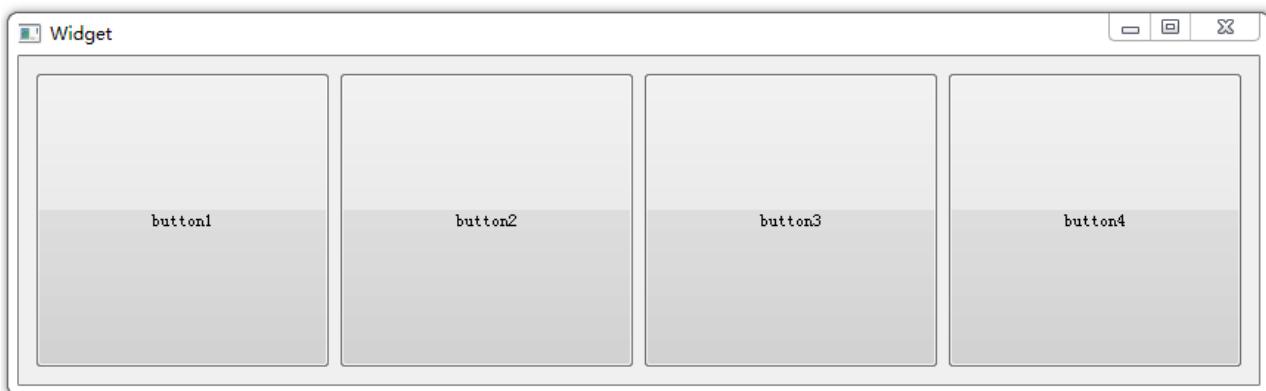
QPushButton button1(&w);
QPushButton button2(&w);
QPushButton button3(&w);
QPushButton button4(&w);
QHBoxLayout* hlayout = new QHBoxLayout(); //水平布局管理器类

button1.setText("button1");
button1.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button1.setMinimumSize(200, 200);
button2.setText("button2");
button2.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button2.setMinimumSize(200, 200);
button3.setText("button3");
button3.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button3.setMinimumSize(200, 200);
button4.setText("button4");
button4.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button4.setMinimumSize(200, 200);

hlayout->addWidget(&button1);
hlayout->addWidget(&button2);
hlayout->addWidget(&button3);
hlayout->addWidget(&button4);

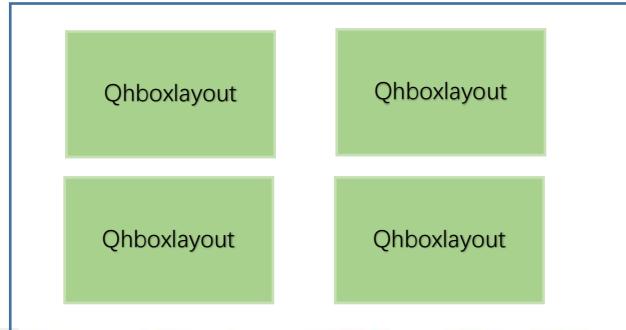
w.setLayout(hlayout);
w.show();
```

水平布局管理器



这就是水平布局管理器，按钮横着放

垂直水平布局管理器相互嵌套使用思路



设计个这样的按钮界面

```
#include <QPushButton>
#include <QDebug>
#include <QHBoxLayout>/>//水平布局管理器头文件
#include <QVBoxLayout>/>//垂直布局管理器头文件
#include "widget.h"
QApplication a(argc, argv);
Widget w;

QPushButton button1(&w);
QPushButton button2(&w);
QPushButton button3(&w);
QPushButton button4(&w);
QHBoxLayout* hlayout1 = new QHBoxLayout(); //建立1个水平布局管理器
QHBoxLayout* hlayout2 = new QHBoxLayout(); //建立2个水平布局管理器
QVBoxLayout* vlayout = new QVBoxLayout(); //建立垂直布局管理器

button1.setText("button1");
button1.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button2.setText("button2");
button2.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

hlayout1->addWidget(&button1); //用水平布局管理器来管理1,2个按钮
hlayout1->addWidget(&button2);
```

1. 建立3个布局管理器

```
button3.setText("button3");
button3.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button4.setText("button4");
button4.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
hlayout2->addWidget(&button3); //再用另外一个水平布局管理器管理3,4个按钮
hlayout2->addWidget(&button4);
```

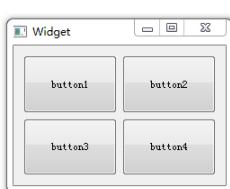
2. 得到横向按钮2个

```
vlayout->addLayout(hlayout1); //用垂直管理器layout函数管理水平布局管理器
vlayout->addLayout(hlayout2); //用垂直管理器layout函数管理水平布局管理器
```

```
w.setLayout(vlayout);
w.show();
return a.exec();
```

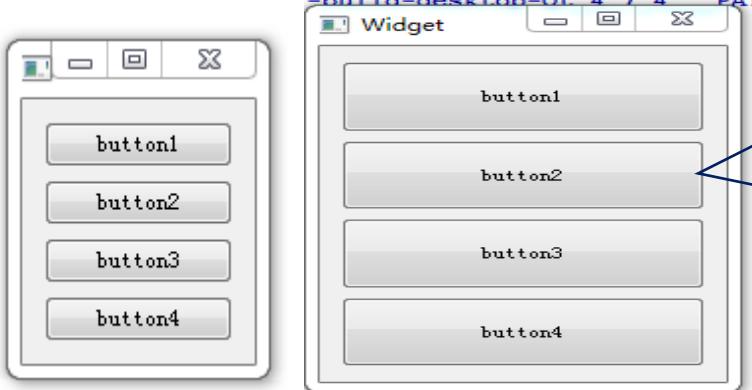
3. 得到第二排横向按钮2个

```
4. 将两排横向按钮垂直分布
```



这就是水平垂直管理器设计思路

设置布局管理区不同按钮放大缩小比例自定义



我们布局管理器管理的按钮窗口是成比例对称放大的，但是现在我想个别按钮放大缩小的比例和其它按钮不一样，怎么办？

```
void setStretch(int index, int stretch);
int index : 表示第几个按钮
int stretch : 表示放大缩小比例
QPushbutton button1(&w);
QPushbutton button2(&w);
QPushbutton button3(&w);
QPushbutton button4(&w);
QVBoxLayout* vlayout = new QVBoxLayout();

button1.setText("button1");
button1.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding); //setSizePolicy按钮水平垂直自动调整
button2.setText("button2");
button2.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button3.setText("button3");
button3.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button4.setText("button4");
button4.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
vlayout->addWidget(&button1);
vlayout->addWidget(&button2);
vlayout->addWidget(&button3);
vlayout->addWidget(&button4);

vlayout->setStretch(0, 1); //第0格按钮
vlayout->setStretch(1, 1); //第1格按钮
vlayout->setStretch(2, 2); //第2格按钮
vlayout->setStretch(3, 2); //第3格按钮
w.setLayout(vlayout);

w.show();
return a.exec();
```

你看按钮按照设置的比例来显示
这种用下标数格子的方法适用于 for 循环格子同一设置
但是不适用于指定某个格子来设置

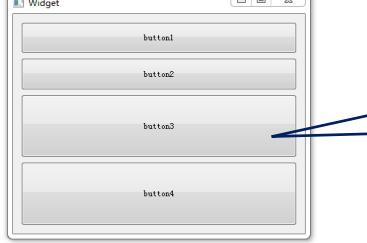
```
bool setStretchFactor(QLayout *l, int stretch);
QLayout *l : 填入按钮对象
```



int stretch : 设置按钮窗口比例

```
vlayout->addWidget(&button1);
vlayout->addWidget(&button2);
vlayout->addWidget(&button3);
vlayout->addWidget(&button4);
vlayout->setStretchFactor(&button1, 1);
vlayout->setStretchFactor(&button2, 2);
vlayout->setStretchFactor(&button3, 4);
vlayout->setStretchFactor(&button4, 4);
w.setLayout(vlayout);

w.show();
```



效果和数格子一样，只不过指定按钮对象更方便阅读

如何自定义布局管理器里嵌入的布局管理器窗口比例

```
button1.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button2.setText("button2");
button2.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

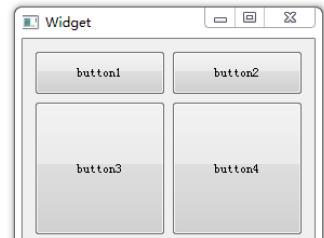
hlayout1->addWidget(&button1); //用水平布局管理器来管理1,2个按钮
hlayout1->addWidget(&button2);

button3.setText("button3");
button3.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button4.setText("button4");
button4.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
hlayout2->addWidget(&button3);
hlayout2->addWidget(&button4);

vlayout->addLayout(hlayout1);
vlayout->addLayout(hlayout2);
vlayout->setStretchFactor(hlayout1, 1);
vlayout->setStretchFactor(hlayout2, 3);

w.setLayout(vlayout);
w.show();
```

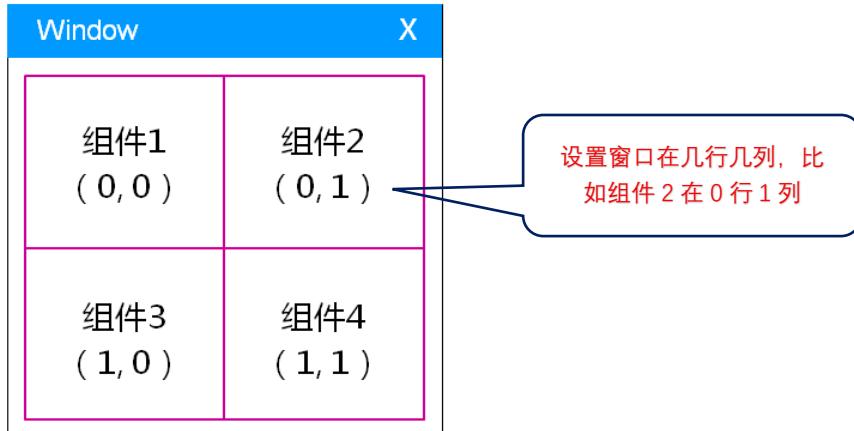
我们直接设置布局管理器里面的子布局管理器的比例就行了



矩阵格子方式的布局管理器用 Vboxlayout 和 Hboxlayout 嵌套方式太麻烦 现在用专用的格子布局管理器

QGridLayout 布局管理器

- 以网格（二维）的方式管理界面组件



```
QPushButton button1(&w);
QPushButton button2(&w);
QPushButton button3(&w);
QPushButton button4(&w);

QGridLayout *layout = new QGridLayout(); // 创建格子布局管理器
button1.setText("button1");
button1.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button2.setText("button2");
button2.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button3.setText("button3");
button3.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
button4.setText("button4");
button4.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
layout->setSpacing(10); // 按钮间距10px
```

```
layout->addWidget(&button1, 0, 0); // 设置按钮1在格子的0行0列
layout->addWidget(&button2, 0, 1); // 设置按钮2在格子的0行1列
layout->addWidget(&button3, 1, 0); // 设置按钮3在格子的1行0列
layout->addWidget(&button4, 1, 1); // 设置按钮4在格子的1行1列
w.setLayout(layout);
w.show();
return a.exec();
```



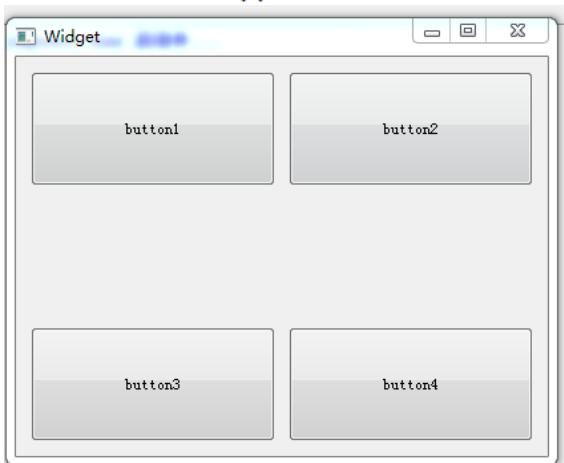
定义了格子布局管理器，然后直接对按钮进行布局，剩去了嵌套布局的麻烦

```
QHBoxLayout* hlayout1 = new QHBoxLayout(); // 建立1个水平
QHBoxLayout* hlayout2 = new QHBoxLayout(); // 建立2个水平
QVBoxLayout* vlayout = new QVBoxLayout(); // 建立垂直布局管
```

```
button4.setText("button4");
button4.setSizePolicy(QSizePolicy::Expand
layout->setSpacing(10); //按钮间距10px
```

```
layout->addWidget(&button1, 0, 0);
layout->addWidget(&button2, 0, 1);
layout->addWidget(&button1, 1, 0);
layout->addWidget(&button2, 1, 1);
layout->addWidget(&button3, 2, 0);
layout->addWidget(&button4, 2, 1);
w.setLayout(layout);
w.show();
```

这段代码只是想证明格子布局管理器是按照行列排列的，但是我这里代码有问题，按钮1重布局了



设置格子布局按钮的放大缩小比例

```
void setColumnStretch(int column, int stretch);
int column : 设置第几列
int stretch : 该列比例大小

layout->addWidget(&button1, 0, 0);
layout->addWidget(&button2, 0, 1);
layout->addWidget(&button3, 1, 0);
layout->addWidget(&button4, 1, 1);

layout->setColumnStretch(0, 1);
layout->setColumnStretch(1, 3);

w.setLayout(layout);
```

第0列大小1, 第1列大小3

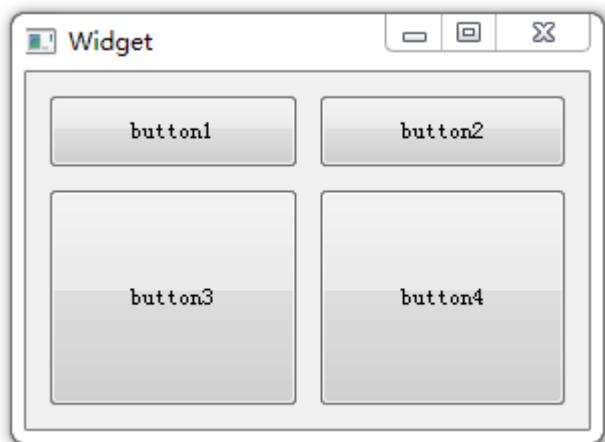
```
void setRowStretch(int row, int stretch);
int row : 设置第行列
```



int stretch : 该列比例大小

```
layout->addWidget(&button1, 0, 0);
layout->addWidget(&button2, 0, 1);
layout->addWidget(&button3, 1, 0);
layout->addWidget(&button4, 1, 1);

layout->setRowStretch(0, 1);
layout->setRowStretch(1, 3);
```



表单布局管理器

```
QApplication a(argc, argv);
Widget w;
QLabel namelab(&w);
QLabel maillab(&w);
QLabel addrlab(&w);
QLineEdit nameedit(&w);
QLineEdit mailedit(&w);
QLineEdit addredit(&w);
QGridLayout *layout = new QGridLayout();

namelab.setText("name");
maillab.setText("mail");
addrlab.setText("addr");
layout->setSpacing(5);
layout->addWidget(&namelab, 0, 0);
layout->addWidget(&nameedit, 0, 1);
layout->addWidget(&maillab, 1, 0);
layout->addWidget(&mailedit, 1, 1);
layout->addWidget(&addrlab, 2, 0);
layout->addWidget(&addredit, 2, 1);
layout->setColumnStretch(0, 1);
layout->setColumnStretch(1, 4);

w.setLayout(layout);
w.show();
return a.exec();
```



下面我们用表单布局管理来解决这个问题

QformLayout 建立表单布局管理器类

```
#include <QFormLayout> //表单布局管理器
```

```
QApplication a(argc, argv);
Widget w;
w.resize(200, 100); //设置窗口初始化大小

QLineEdit nameedit(&w);
QLineEdit mailedit(&w);
QLineEdit addredit(&w);
QFormLayout *layout = new QFormLayout(); //表单布局管理器类

layout->setSpacing(5);
layout->addRow("name", &nameedit); //直接在表单布局管理器写 QLabel 标签名, 编辑框 QLineEdit 地址
layout->addRow("mail", &mailedit);
layout->addRow("addr", &addredit);

w.setLayout(layout);
w.show();
return a.exec();
```

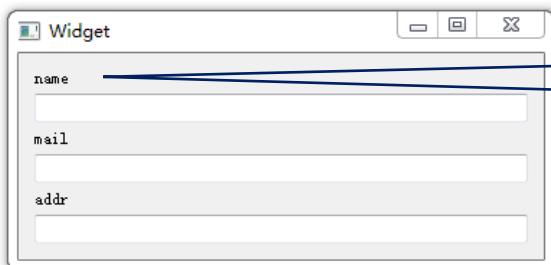
使用 addRow 链接标签和编辑框



对话框拉长中间不会有空白

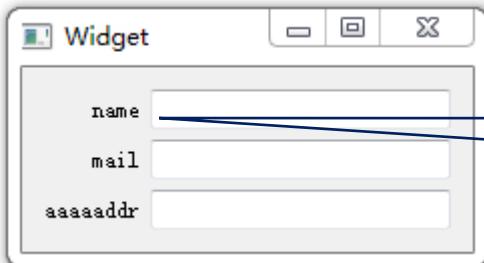


```
layout->setSpacing(5);
layout->addRow("name", &nameedit);
layout->addRow("mail", &mailedit);
layout->addRow("addr", &addredit);
layout->setRowWrapPolicy(QFormLayout::WrapAllRows); //可以设置标签和编辑框的排列
```



你看标签在编辑框上面

```
layout->setSpacing(5);
layout->addRow("name", &nameedit);
layout->addRow("mail", &mailedit);
layout->addRow("aaaaaddr", &addredit);
layout->setLabelAlignment(Qt::AlignRight); //标签字体向右对齐
```



你看标签右对齐了

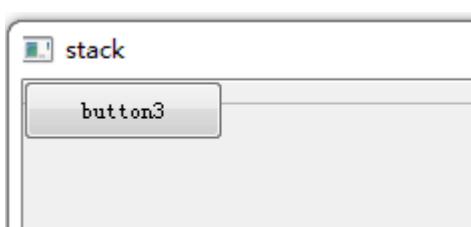
栈式布局管理器

```
#include <QStackedLayout> // 栈式布局管理器
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    stack w;
    QPushButton button1(&w);
    QPushButton button2(&w);
    QPushButton button3(&w);
    QPushButton button4(&w);

    QStackedLayout* slayout = new QStackedLayout();
    button1.setText("button1");
    button2.setText("button2");
    button3.setText("button3");
    button4.setText("button4");

    slayout->addWidget(&button1); // 第0层
    slayout->addWidget(&button2); // 第1层
    slayout->addWidget(&button3); // 第2层
    slayout->addWidget(&button4); // 第3层
    slayout->setcurrentIndex(2); // 我到底显示第几层按钮
    w.setLayout(slayout); // 将布局属性装入父窗口的布局变量
    w.show();

    return a.exec();
}
```



将按钮自动放入布局管理器函数，根据摆放顺序，决定层数

```
slayout->addWidget(&button1); // 第0层
slayout->addWidget(&button2); // 第1层
slayout->addWidget(&button3); // 第2层
slayout->addWidget(&button4); // 第3层
slayout->setcurrentIndex(0); // 我到底显示第几层按钮
w.setLayout(slayout); // 将布局属性装入父窗口的布局变量
w.show();
```



修改显示第0层

做一个嵌套的布局管理器窗口

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    stack w;
    QStackedLayout* slayout = new QStackedLayout();
    QHBoxLayout* hlayout = new QHBoxLayout();
    QWidget* w2 = new QWidget(); //建立另外一个父窗口，现在父窗口就有w和w2了
    QPushButton button1(&w);
    QPushButton button2(&w);
    QPushButton button3(&w);
    QPushButton button4(&w);

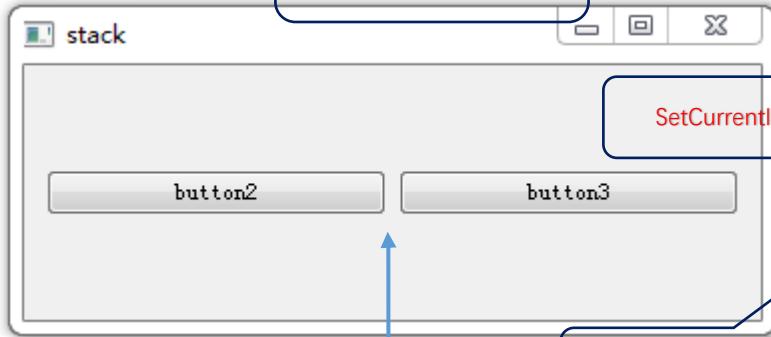
    button2.setParent(w2); //将button2的父窗口改成w2
    button3.setParent(w2); //将button3的父窗口改成w2

    button1.setText("button1");
    button2.setText("button2");
    button3.setText("button3");
    button4.setText("button4");

    hlayout->addWidget(&button2); //需要水平布局的是button2按钮
    hlayout->addWidget(&button3); //需要水平布局的是button3按钮
    /*但是button2和button3是在什么窗口下水平布局呢？*/
    w2->setLayout(hlayout); // button2和button3当然是在w2窗口里面水平布局

    slayout->addWidget(&button1); //第0层
    slayout->addWidget(w2); //把水平布局好的窗口放在第1层，这就是stack布局管理器嵌套
    slayout->addWidget(&button4); //第2层

    slayout->setCurrentIndex(1); //我到底显示第几层按钮
    w.setLayout(slayout); //将布局属性装入父窗口的布局变量
    // w.show();
    return a.exec();
```



将嵌入按钮的父窗口放入爷爷窗口

所以这个 w 没有意义，
会多显示出一个窗口

将按钮设置进 w2 父窗口

将按钮设置进 w2 父窗口

将按钮放入新建立的窗口

将按钮设置为水平布局

建立 4 个按钮

button2.setParent(w2); //将button2的父窗口改成w2
button3.setParent(w2); //将button3的父窗口改成w2

button1.setText("button1");
button2.setText("button2");
button3.setText("button3");
button4.setText("button4");

hlayout->addWidget(&button2); //需要水平布局的是button2按钮
hlayout->addWidget(&button3); //需要水平布局的是button3按钮

/*但是button2和button3是在什么窗口下水平布局呢？*/

w2->setLayout(hlayout); // button2和button3当然是在w2窗口里面水平布局

slayout->addWidget(&button1); //第0层

slayout->addWidget(w2); //把水平布局好的窗口放在第1层，这就是stack布局管理器嵌套

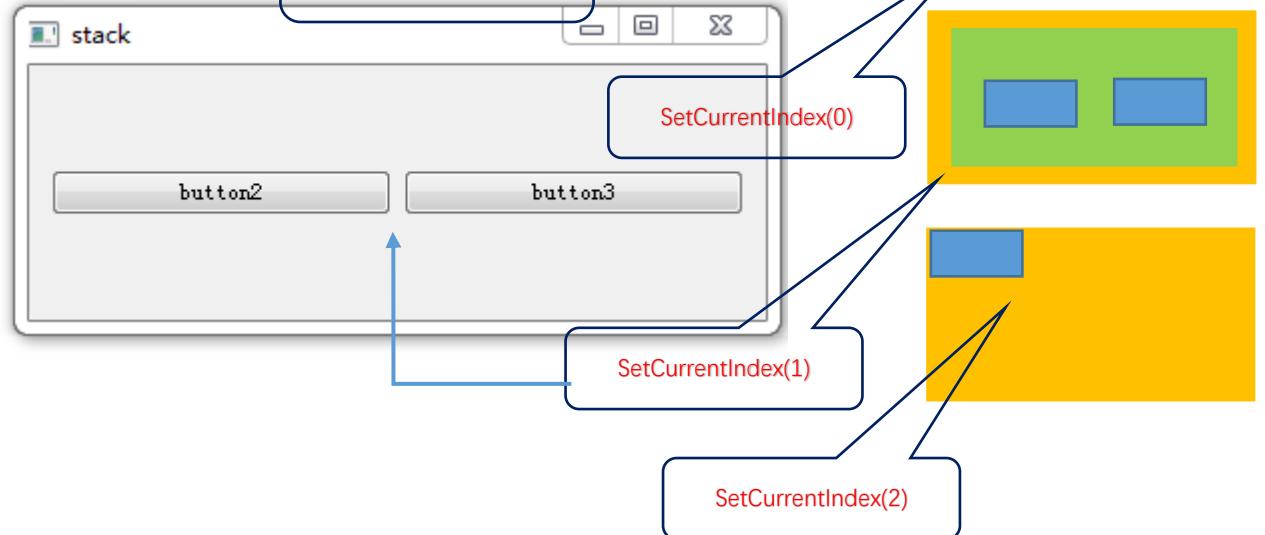
slayout->addWidget(&button4); //第2层

slayout->setCurrentIndex(1); //我到底显示第几层按钮

w.setLayout(slayout); //将布局属性装入父窗口的布局变量

// w.show();

return a.exec();



所以只需要执行 w.setLayout 就可以了，父窗口显示子窗口和按钮。

计时器

```
class stack : public QMainWindow
{
    Q_OBJECT

public:
    explicit stack(QWidget *parent = 0);
    ~stack();

private:
    Ui::stack *ui;
private slots:
    void timerout();
```

#include <Qtcore> //计时器
#include "stack.h"

```
void stack::timerout()
{
    qDebug() << "timer";
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    stack w;
    QTimer* timer = new QTimer();
    QObject::connect(timer, SIGNAL(timeout()), &w, SLOT(timerout()));
    timer->start(1000);
    return a.exec();
}
```

在 h 文件建立信号接受函数

定义一个计时器

这个计时器计时时间到就发送信号执行槽函数

单位为毫秒(ms), timer 计时器 1 秒发送一次信号

```
stack3 [x]
timer
timer
timer
timer
timer
timer
timer
```

布局管理器综合实验(重点，多复习)

在头文件定义初始化构造函数

```
class Widget : public QWidget
{
    Q_OBJECT
    void initcontrol();
public:
```

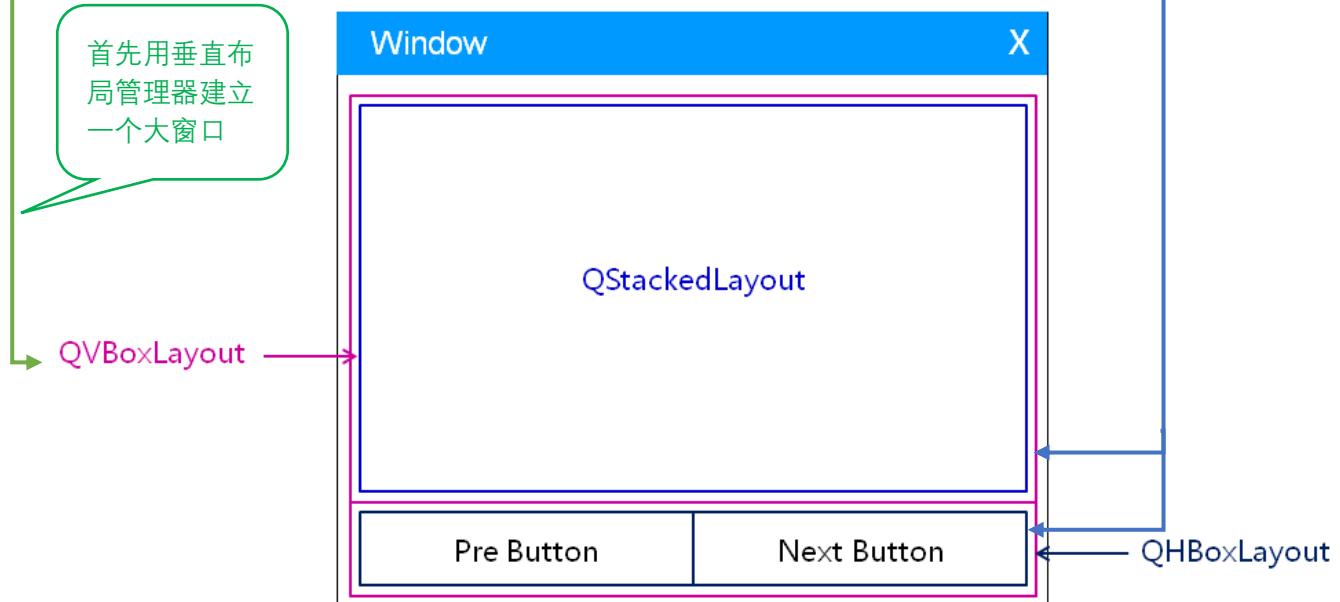
在C文件里面实现

```
void Widget::initcontrol()
{
    QVBoxLayout* vlayout = new QVBoxLayout();
    QHBoxLayout* hlayout = new QHBoxLayout();
    QStackedLayout* slayout = new QStackedLayout();

    vlayout->addLayout(slayout);
    vlayout->addLayout(hlayout);

}
```

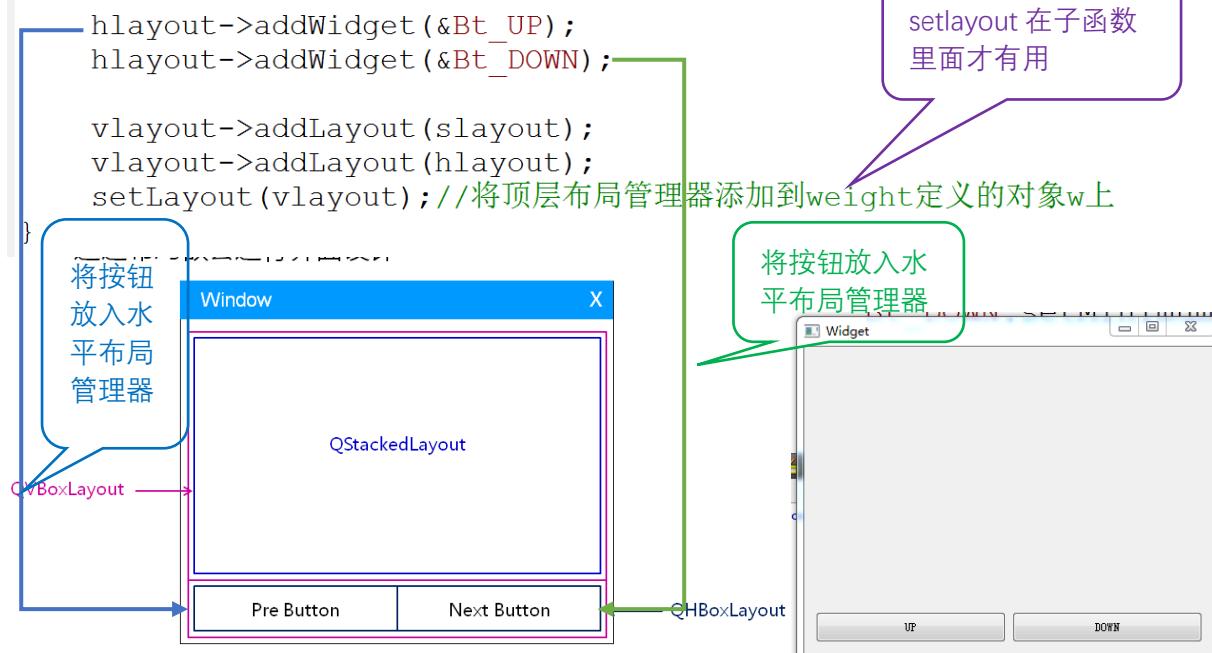
水平布局管理器和栈式布局管理器把垂直布局管理器分成了两部分



在头文件定义两个按钮

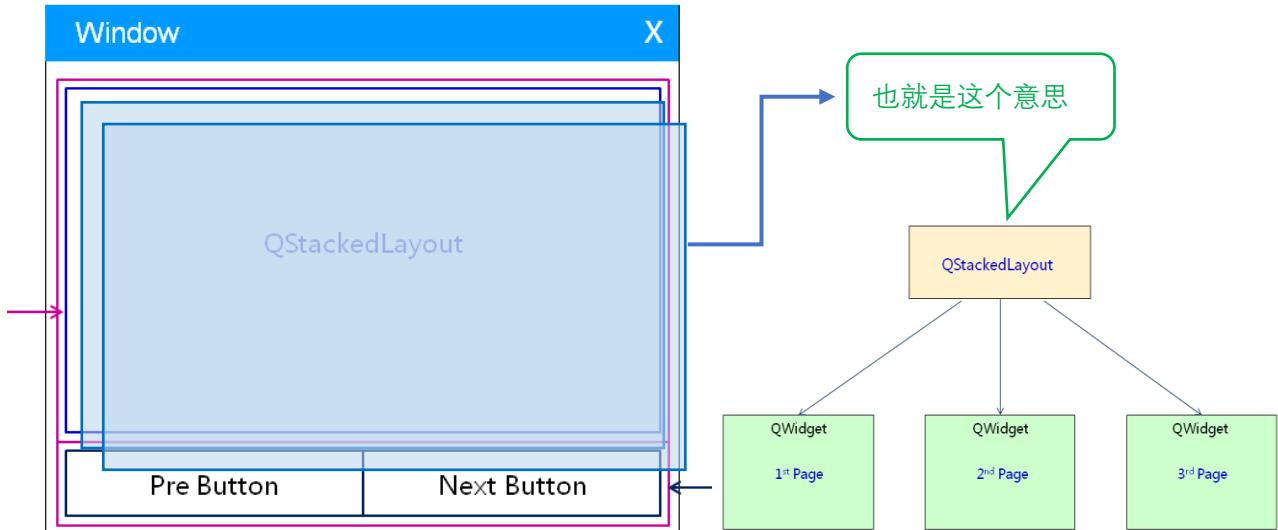
```
private:  
    Ui::Widget *ui;  
    QPushButton Bt_UP;  
    QPushButton Bt_DOWN;  
};
```

```
void Widget::initcontrol()  
{  
    QVBoxLayout* vlayout = new QVBoxLayout();  
    QHBoxLayout* hlayout = new QHBoxLayout();  
    QStackedLayout* slayout = new QStackedLayout();  
  
    Bt_UP.setText("UP"); //设置按钮名字  
    Bt_UP.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);  
    //Expandin窗口水平拉伸，按钮跟着拉伸，Fixed窗口垂直拉伸，按钮大小不动  
    Bt_UP.setMinimumSize(160, 30); //按钮最小大小  
  
    Bt_DOWN.setText("DOWN"); //设置按钮名字  
    Bt_DOWN.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);  
    Bt_DOWN.setMinimumSize(160, 30);
```



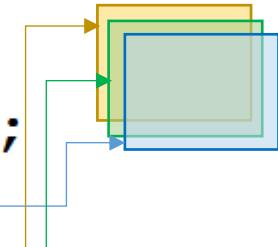
这样就形成了第一个页面

我们用 stack 布局管理器创建 3 层窗口



头文件先定义三个窗口

```
class Widget : public QWidget
{
    Q_OBJECT
    void initcontrol();
    QWidget* w1();
    QWidget* w2();
    QWidget* w3();
```



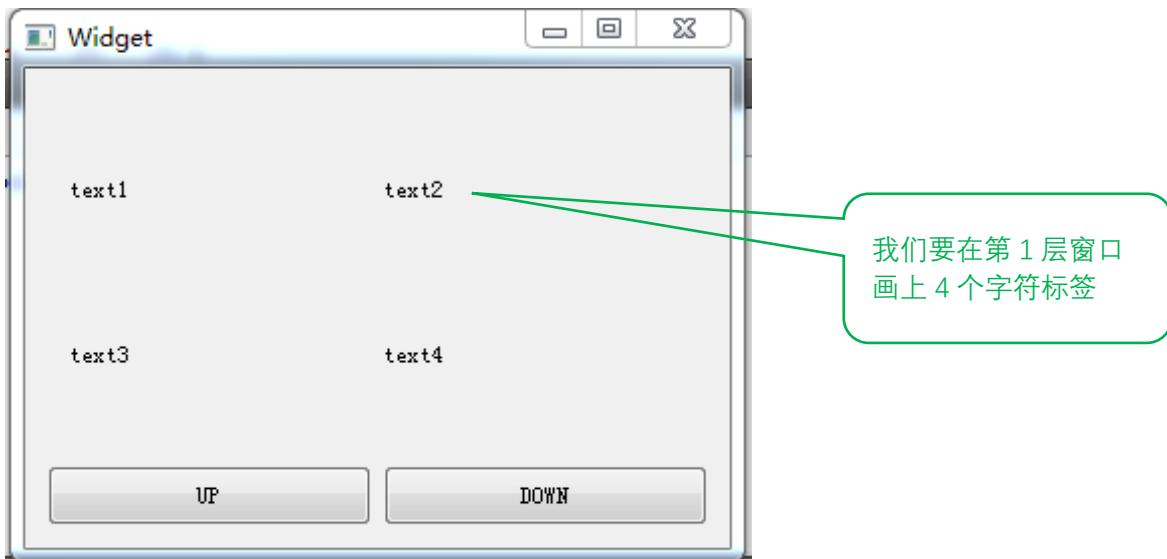
在 Cpp 文件实现这三个窗口函数，返回的是自己这层窗口的地址

```
QWidget* Widget::w1()
{
    QWidget* ret = new QWidget();
    return ret;
}

QWidget* Widget::w2()
{
    QWidget* ret = new QWidget();
    return ret;
}

QWidget* Widget::w3()
{
    QWidget* ret = new QWidget();
    return ret;
}
```





```
class Widget : public QWidget
{
    Q_OBJECT
    void initcontrol();
    QWidget* w1();
    QWidget* w2();
    QWidget* w3();
    QLabel text1;
    QLabel text2;
    QLabel text3;
    QLabel text4;

public:
    QWidget* Widget::w1()
    {
        QWidget* ret = new QWidget();
        QGridLayout* glayout = new QGridLayout();
        text1.setText("text1");
        text2.setText("text2");
        text3.setText("text3");
        text4.setText("text4");

        glayout->addWidget(&text1, 0, 0);
        glayout->addWidget(&text2, 0, 1);
        glayout->addWidget(&text3, 1, 0);
        glayout->addWidget(&text4, 1, 1);
        ret->setLayout(glayout);

        return ret;
    }
}
```

在头文件建立4个标签的对象

在cpp文件第1层窗口的函数实现头文件的4个标签对象

创建第1层窗口

创建表格布局管理器

给4个标签添加名字

表格布局管理器设置4标签放的位置

将表格布局管理器放在第一层窗口

```

void Widget::initcontrol()
{
    QVBoxLayout* vlayout = new QVBoxLayout();
    QHBoxLayout* hlayout = new QHBoxLayout();
    QStackedLayout* slayout = new QStackedLayout();

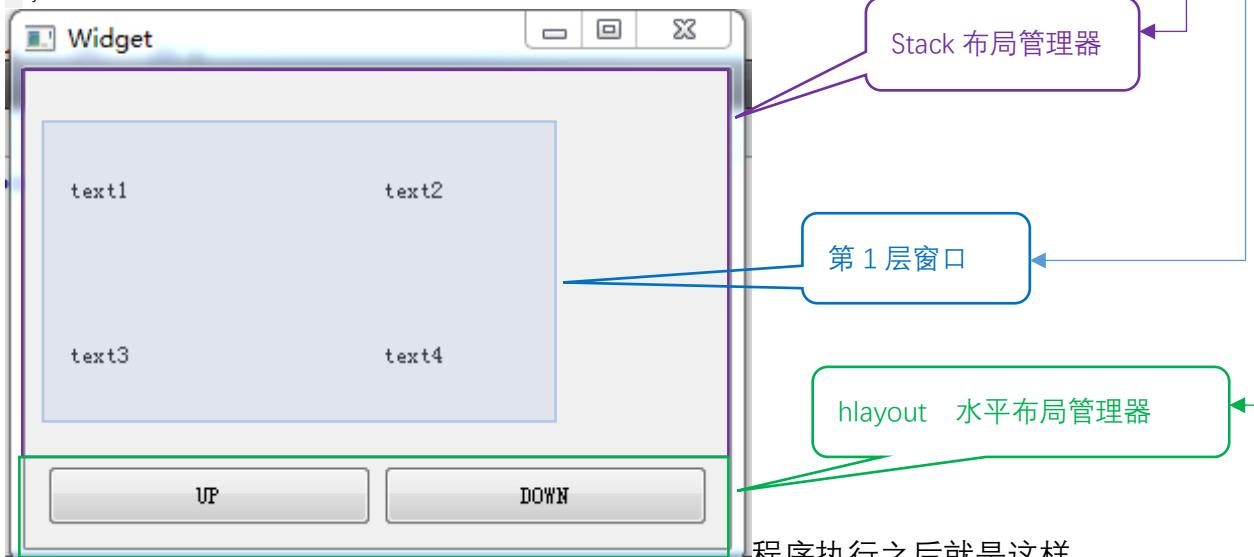
    Bt_UP.setText("UP"); //设置按钮名字
    Bt_UP.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    //Expandin窗口水平拉伸，按钮跟着拉伸，Fixed窗口垂直拉伸，按钮大小不动
    Bt_UP.setMinimumSize(160, 30); //按钮最小大小

    Bt_DOWN.setText("DOWN"); //设置按钮名字
    Bt_DOWN.setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    Bt_DOWN.setMinimumSize(160, 30);
    hlayout->addWidget(&Bt_UP);
    hlayout->addWidget(&Bt_DOWN);

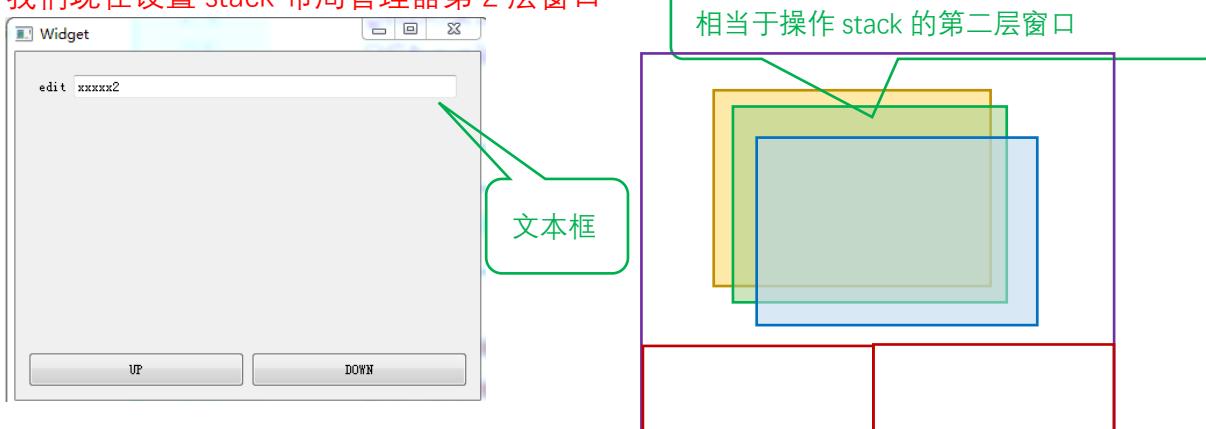
    slayout->addWidget(w1());
    slayout->addWidget(w2());
    slayout->addWidget(w3());

    vlayout->addLayout(slayout);
    vlayout->addLayout(hlayout);
    setLayout(vlayout); //将顶层布局管理器添加到weight定义的对象w上
}

```



我们现在设置 stack 布局管理器第 2 层窗口



```
class Widget : public QWidget
{
    Q_OBJECT
    void initcontrol();
    QWidget* w1();
    QWidget* w2();
    QWidget* w3();
    QLabel text1;
    QLabel text2;
    QLabel text3;
    QLabel text4;
    QLineEdit linedit;
```

第二层窗口有文本框，在头文件加入文本框对象

```
QWidget* Widget::w2()
{
    QWidget* ret = new QWidget();
    QFormLayout *flayout = new QFormLayout(); // 创建表单对象
    linedit.setText("xxxxx2"); // 给文本框初始化字符
    flayout->addRow("edit", &linedit); // 将表单和文本框链接在一起
    ret->setLayout(flayout); // 将表单对象嵌入第2层窗口
    return ret;
}
```

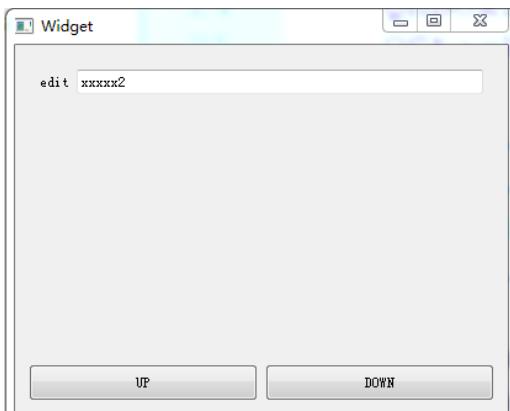
在第2层窗口里面实现内容

```
hlayout->addWidget(&Bt_UP);
hlayout->addWidget(&Bt_DOWN);

// slayout->addWidget(w1());
// 去掉stack布局管理器第1层，这样才能看到第2层的窗口内容
slayout->addWidget(w2());
slayout->addWidget(w3());

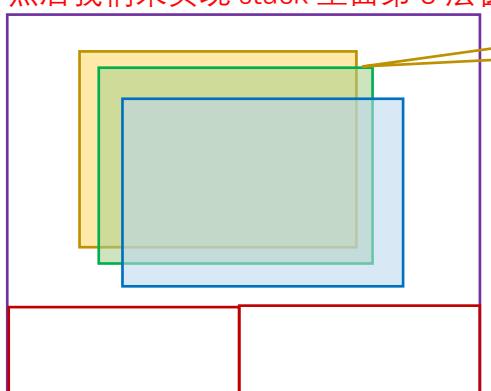
vlayout->addLayout(slayout);
vlayout->addLayout(hlayout);
```

在初始化函数里面
显示 stack 管理器
第二层窗口



这是实现结果

然后我们来实现 stack 里面第 3 层窗口



这是 stack 里面第 3 层窗口

```
class Widget : public QWidget
{
    Q_OBJECT
public:
    void initcontrol();
    QWidget* w1();
    QWidget* w2();
    QWidget* w3();
    QLabel text1;
    QLabel text2;
    QLabel text3;
    QLabel text4;
    QLineEdit linedit;
    QPushButton button1;
    QPushButton button2;
};
```

在头文件添加
第 3 层窗口按
钮

```
QWidget* Widget::w3()
{
    QWidget* ret = new QWidget();
    QVBoxLayout *w3vlayout = new QVBoxLayout();
    button1.setText("button1");
    button2.setText("button2");

    w3vlayout->addWidget(&button1);
    w3vlayout->addWidget(&button2);

    ret->setLayout(w3vlayout);
    return ret;
}
```

在第 3 层窗口添
加垂直布局管理器，
让按钮竖着
排列

```
//slayout->addWidget(w1()); //遮住stack布局管理器第1层
//slayout->addWidget(w2()); //遮住stack布局管理器第2层
slayout->addWidget(w3());
vlayout->addLayout(slayout);
```

在初始化函数打
开第 3 层窗口



在头文件增加函数

```

24     QLineEdit linedit;
25     QPushButton button1;
26     QPushButton button2;
27     QStackedLayout slayout;
28 //将stack布局对象放入类，这样继承该Widget类
29 //的各种函数都可以调用slayout布局管理器，slayout在类里面类似全局的
30 public:
31     explicit Widget(QWidget *parent = 0);
32     ~Widget();
33
34
35 private:
36     Ui::Widget *ui;
37     QPushButton Bt_UP;
38     QPushButton Bt_DOWN;
39 private slots:
40     void ON_UP();
41     void OFF_DOWN();
42 }

void Widget::initcontrol()
{
    QVBoxLayout* vlayout = new QVBoxLayout();
    QHBoxLayout* hlayout = new QHBoxLayout();

    Bt_UP.setText("UP");
    Bt_UP.setSizePolicy(QSizePolicy::Expanding,QSizePolicy::Fixed);

    Bt_UP.setMinimumSize(160,30);
    Bt_DOWN.setText("DOWN");
    Bt_DOWN.setSizePolicy(QSizePolicy::Expanding,QSizePolicy::Fixed);
    Bt_DOWN.setMinimumSize(160,30);

    hlayout->addWidget(&Bt_UP);
    hlayout->addWidget(&Bt_DOWN);

    slayout.addWidget(w1());
    slayout.addWidget(w2());
    slayout.addWidget(w3());

    vlayout->addLayout(&slayout);
    vlayout->addLayout(hlayout);
    setLayout(vlayout); //将顶层布局管理器添加到weight定义的对象w上

    connect(&Bt_UP,SIGNAL(clicked()),this,SLOT(ON_UP()));
    connect(&Bt_DOWN,SIGNAL(clicked()),this,SLOT(OFF_DOWN()));
}

```

因为 cpp 文件其它函数会调用到 Qstack, 所以我们把它写在类里面

增加两个信号，槽函数来响应按钮事件

CPP 文件初始化函数
增加信号与槽关联

```

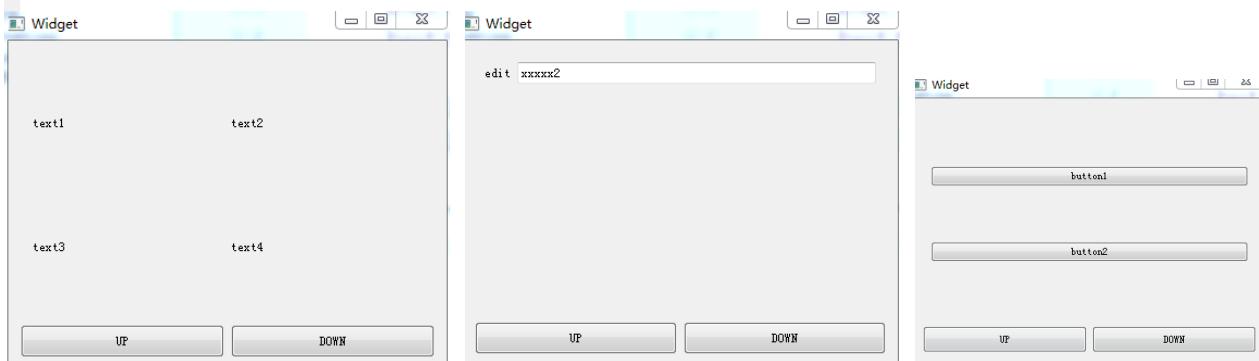
void Widget::ON_UP()
{
    int index = ((slayout.currentIndex() - 1) + 3) % 3;
    slayout.setCurrentIndex(index);
}

```

```

void Widget::OFF_DOWN()
{
    int index = (slayout.currentIndex() + 1) % 3;
    slayout.setCurrentIndex(index);
}

```



这样就是实现了界面按钮切换

这里有个重点问题说一下

```

void Widget::ON_UP()
{
    int index = ((slayout.currentIndex() - 1) + 3) % 3;
    slayout.setCurrentIndex(index);
}

```

获取我现在 stack 界面显示的哪一层 ?

```

void Widget::OFF_DOWN()
{
    int index = (slayout.currentIndex() + 1) % 3;
    slayout.setCurrentIndex(index);
}

```

设置我想显示哪一层

SetcurrentIndex 是设置 stack 区域显示第几层界面

SetcurrentIndex(0)

显示第 1 层

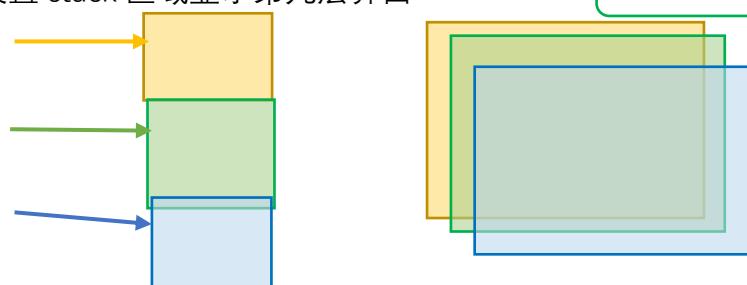
SetcurrentIndex(1)

显示第 2 层

SetcurrentIndex(2)

显示第 3 层

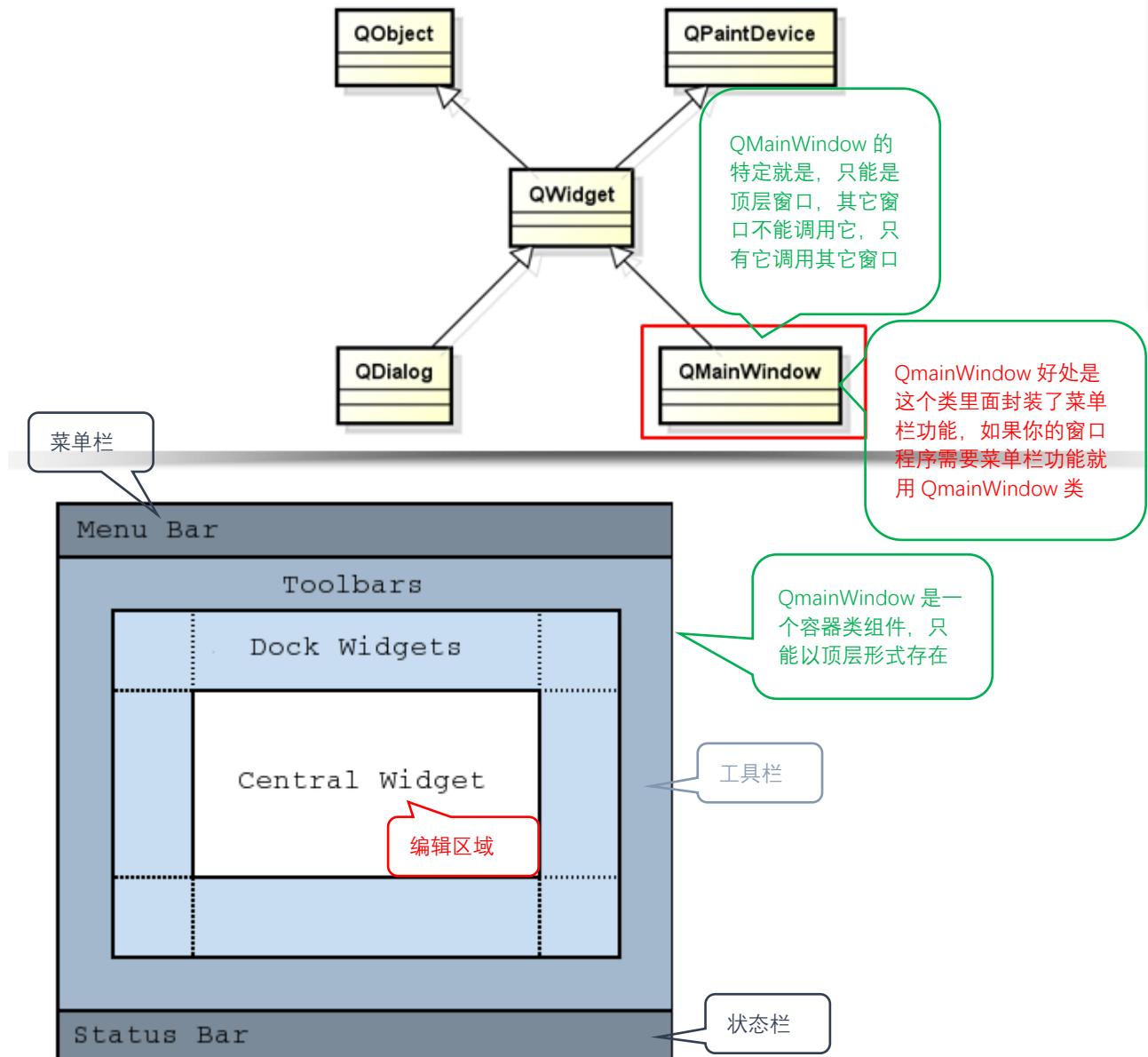
至于这个计算自己去想



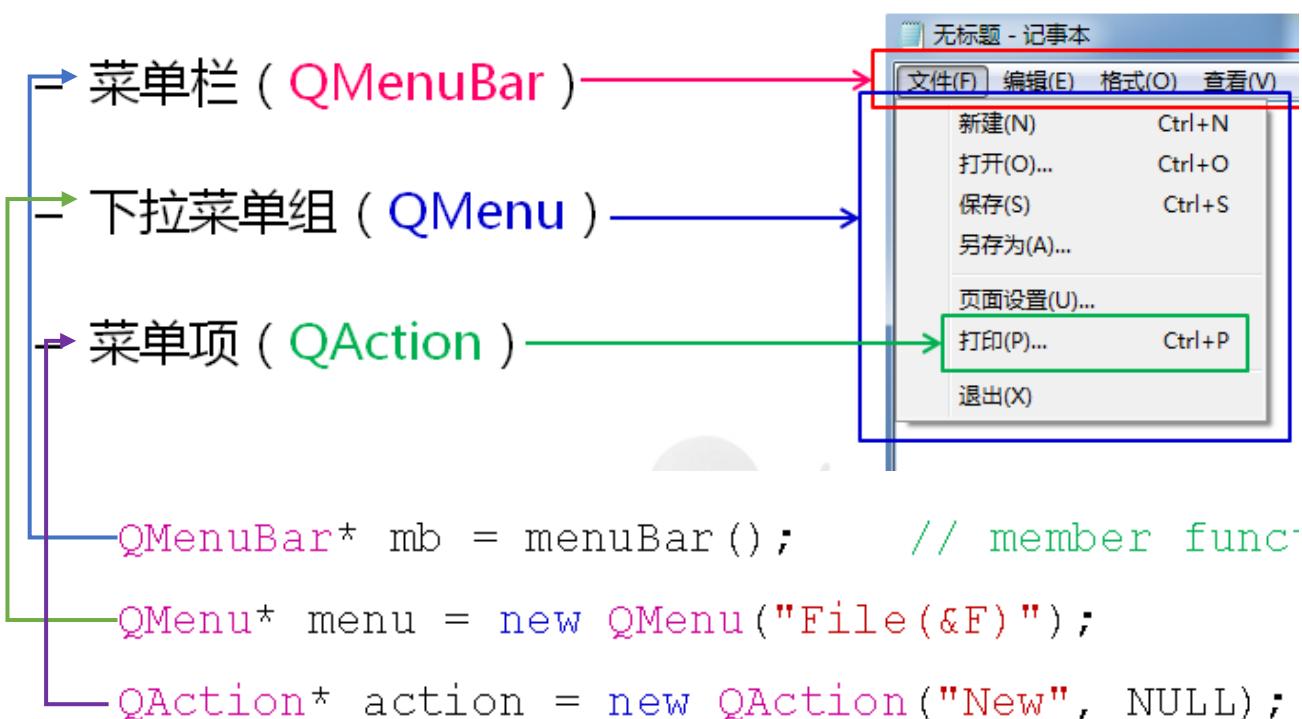
菜单栏程序创建，用 QMainWindow 顶层窗口

QMenuBar 类 QMenu 类 QAction 类

- QMainWindow 继承于 QWidget 是一种容器类型的组件



下面我们来说说设计下拉菜单栏的思路



menu->addAction(action);

下拉菜单栏里的
菜单项放在下
拉菜单组

mb->addMenu(menu);

将下拉菜单组
放入菜单栏

在头文件定义窗口需求

```

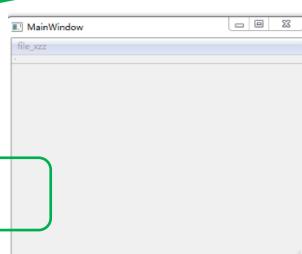
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
    static MainWindow* NewInstance();
private:
    Ui::MainWindow *ui;
    bool construct(); // 创建菜单栏
    bool init_menu(); // 创建下拉菜单栏
    bool init_menufile(QMenuBar* mb); // 在下拉菜单栏插入选择框
    bool makeaction(QAction*& action, QString text); // 将创建的action地址返回给外部指针
};

MainWindow* MainWindow::NewInstance()
{
    MainWindow *ret = new MainWindow();
    if(ret == NULL)
    {
        delete ret;
        ret = NULL;
    }
    ret->construct();
    return ret;
}

```

1.先创立顶层界面窗口



6.菜单栏创建完成

```

bool MainWindow::construct()
{
    bool ret = true;
    ret = init_meue();
    return ret;
}

bool MainWindow::init_meue() ←
{
    bool ret = true;
    QMenuBar *mb = menuBar(); // 创建菜单栏
    ret = init_meuefile(mb); // 创建下拉菜单组
    return ret;
}

bool MainWindow::init_meuefile(QMenuBar* mb)
{
    bool ret = true;
    QMenu *menu = new QMenu("file_xzz"); // 在行条上创建下拉菜单
    QAction* action = NULL;
    ret = makeaction(action, "new");
    menu->addAction(action);
    mb->addMenu(menu);
    return ret;
}

```

2. 创建菜单栏

3. 给菜单栏添加菜单项

4. 将菜单栏选项放入下拉菜单栏

5. 将下拉菜单栏放入顶层窗口

我发现这里没有快捷键提示，alt+F无法快捷打开下拉菜单

```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow* w = MainWindow::NewInstance();
    int ret = -1;

    w->show();
    ret = a.exec();
}

```

执行主程序我们看看最终效果

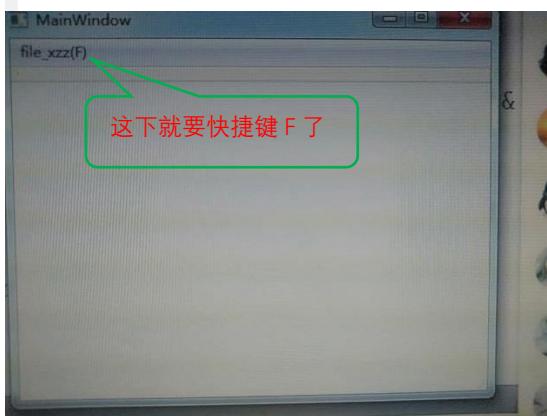
```

bool MainWindow::init_menufile(QMenuBar* mb)
{
    bool ret = true;
    QMenu *menu = new QMenu("file_xzz (&F)"); //在行条上创建下拉菜单

    QAction* action = NULL;
    ret = makeaction(action, "new");

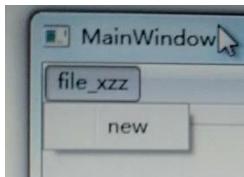
    menu->addAction(action);
    mb->addMenu(menu);
    return ret;
}

```



我们要用(&指定快捷符号)
才能显示快捷键字符

我们按 alt+F 就可以快捷打开下拉菜单



如何给 new 按钮设置快捷键

QkeySequence 类就是创建快捷按钮对象，setshortcut 将快捷按钮对象放入菜单项

```

bool MainWindow::makeaction(QAction*& action, QString text, int key)
{
    bool ret = true;
    action = new QAction(text, NULL); //QAction创建下拉菜单项

    action->setShortcut(QKeySequence(key)); //将快捷键加入进下拉菜单项
    return ret;
}

bool MainWindow::init_menufile(QMenuBar* mb)
{
    bool ret = true;
    QMenu *menu = new QMenu("file_xzz (&F)"); //在行条上创建下拉菜单

    QAction* action = NULL;
    ret = makeaction(action, "new (&N)", Qt::CTRL+Qt::Key_N); //传入设置的快捷按钮

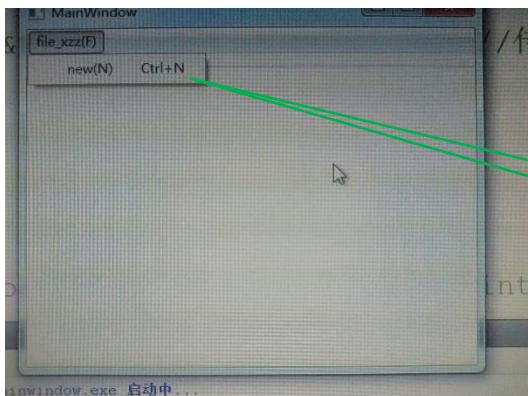
    menu->addAction(action);
    mb->addMenu(menu);
    return ret;
}

```

将快捷键对象放入指定的菜单项

Qkeysequence 根据 key 参数
创建快捷键类

我们要实现 CTRL+N 的快
捷键，所以这里就用 Qt 来设
置



菜单项的快捷键成功创建

如果我想在下拉菜单列表多创建几个菜单项怎么办？

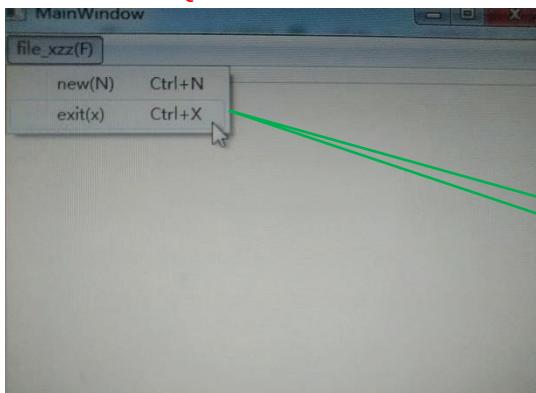
```
bool MainWindow::init_menufile(QMenuBar* mb)
{
    bool ret = true;
    QMenu *menu = new QMenu("file_xzz(&F)");//在行条上创建下拉菜单

    QAction* action = NULL;
    ret = makeaction(action, "new(&N)", Qt::CTRL+Qt::Key_N);//第1行菜单项创建
    menu->addAction(action);//第1行菜单项加入下拉菜单组

    ret = makeaction(action, "exit(&x)", Qt::CTRL+Qt::Key_X);//第2行菜单项创建
    menu->addAction(action);//第2行菜单项加入下拉菜单组

    mb->addMenu(menu); //将下拉菜单组赋值给菜单行
    return ret;
}
```

多创建几个 QAction 就可以了



多个菜单项已经实现

但是我发现菜单项之间没有横杠看起不舒服

```
bool MainWindow::init_menufile(QMenuBar* mb)
{
    bool ret = true;
    QMenu *menu = new QMenu("file_xzz(&F)");//在行条上创建下拉菜单

    QAction* action = NULL;
    ret = makeaction(action, "new(&N)", Qt::CTRL+Qt::Key_N);
    menu->addAction(action);//第1行菜单项加入下拉菜单组

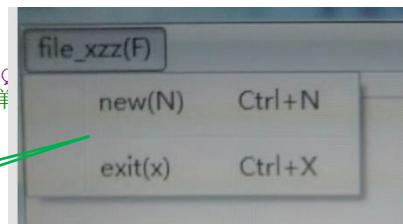
    menu->addSeparator(); //在两行菜单项之间加入横杠

    ret = makeaction(action, "exit(&x)", Qt::CTRL+Qt::Key_X);
    menu->addAction(action);//第2行菜单项加入下拉菜单组

    mb->addMenu(menu); //将下拉菜单组赋值给菜单行
    return ret;
}
```

用下拉菜单类在菜单项之间加入横杠

这就是横杠



菜单栏界面制作的时候，要多加一些判断语句，已保正式工程项目安全

```
MainWindow* MainWindow::NewInstance()
{
    MainWindow *ret = new MainWindow();
    if(ret == NULL) //判断顶层窗口是否创建成功
    {
        delete ret;
        ret = NULL;
    }
    ret->construct();
    return ret;
}

bool MainWindow::construct()
{
    bool ret = true;
    ret = ret && init_meue(); //判断菜单是否初始化成功
    return ret;
}

bool MainWindow::init_meue()
{
    bool ret = true;
    QMenuBar *mb = menuBar();
    ret = ret && init_meuefile(mb); //判断下拉菜单是否创建成功
    return ret;
}

bool MainWindow::init_meuefile(QMenuBar* mb)
{
    bool ret = true;
    QMenu *menu = new QMenu("file_xzz(&F)");
    ret = (menu != NULL); //获取菜单行对象创建是否成功

    if(ret) //如果菜单行创建成功我们才创立下拉菜单
    {
        QAction* action = NULL;
        ret = ret && makeaction(action, "new(&N)", Qt::CTRL+Q1);
        if(ret) //如果下拉菜单创建成功我们才创建菜单项
        {
            menu->addAction(action);
        }

        menu->addSeparator();
        ret = ret && makeaction(action, "exit(&x)", Qt::CTRL+Q1);
        if(ret) //如果下拉菜单创建成功我们才创建菜单项
        {
            menu->addAction(action);
        }
    }
}
```

说白了就是创建对象之后
多加 if...else 判断对象是否
创建成功

如果不加 if...else 判断，
你也要把对象创建后的状态
返回给上层函数取判断

```

    if(ret) //如果下拉菜单和两个菜单项创建成功，我才把下拉菜单放入菜单栏
    {
        mb->addMenu(menu);
    }

    return ret;
}

bool MainWindow::makeaction(QAction*& action,QString text,int
{
    bool ret = true;

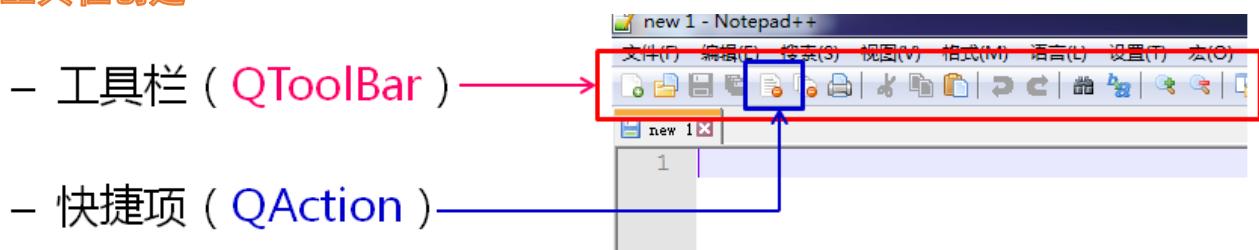
    action = new QAction(text,NULL);
    if( action != NULL )//菜单项创建成功，才允许创建快捷键
    {
        action->setShortcut(QKeySequence(key));
    }
    else
    {
        ret = false;
    }

    return ret;
}

```

还有就是记得动态释放内存，这点没有做，因为没有子窗口，所以就没做

工具栏创建



```
/* call member function */
```

```
QToolBar* tb = addToolBar("Tool Bar");
```

```
/* create item for Tool Bar */
```

```
QAction* action = new QAction("", NULL);
```

创建工具栏条

```
/* set action property */
```

```
action->setToolTip("Open");
```

工具栏里面的每格图标都是用 QAction 创建

```
/* add item to Tool Bar */
```

setToolTip 是设置鼠标移到图标显示什么字符

```
tb->addAction(action);
```

将图标放入工具栏

```

#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QToolBar> //工具栏头文件
#include <QAction>

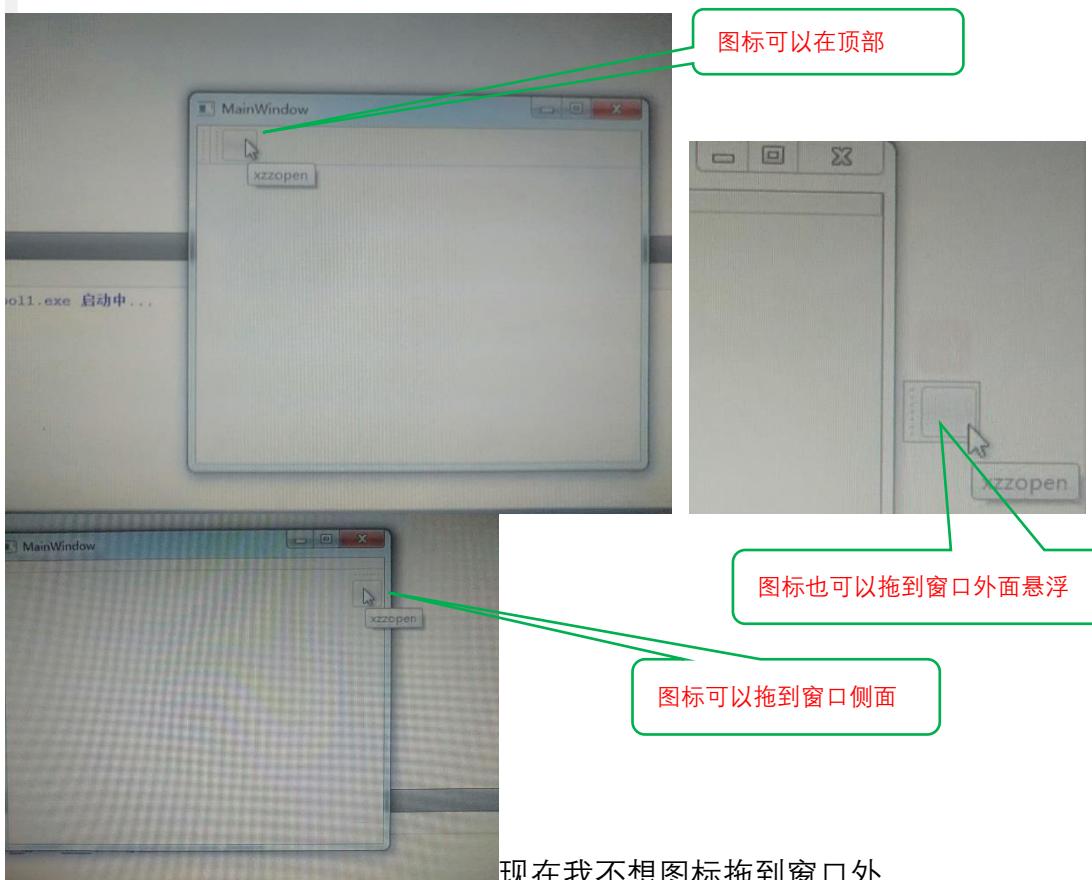
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
{
    ui->setupUi(this);
    QToolBar* tb = addToolBar("xzz_tool"); //创建工具栏对象
    QAction* action = new QAction("",NULL); //创建空白图标
    action->setToolTip("xzzopen"); //工具栏里面图标的提示字符

    tb->addAction(action); //将空白图标放入工具栏
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}

```

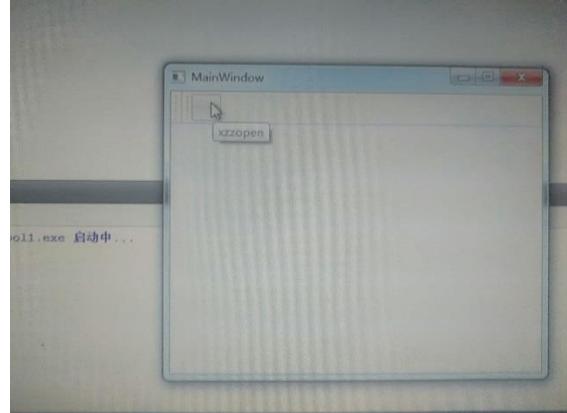
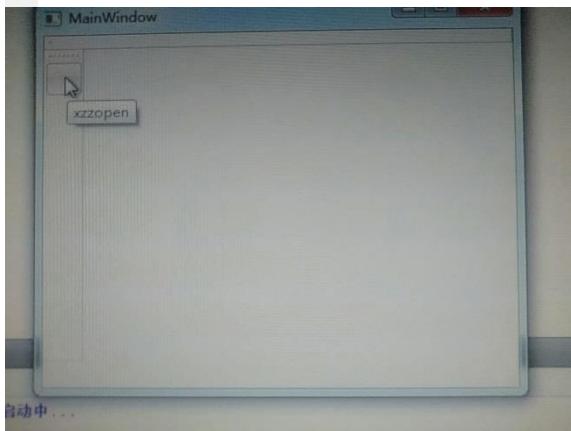


```

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
{
    ui->setupUi(this);
    QToolBar* tb = addToolBar("xzz_tool");//创建工具栏对象
    QAction* action = new QAction("",NULL);//创建
    action->setToolTip("xzzopen");//工具栏里面图标的提示字符

    tb->setFloatable(false); //禁用浮动功能
    //禁止工具栏向窗口外悬浮，只允许工具栏在窗口内悬浮
    tb->addAction(action);
}

```



菜单栏只能在窗口内部浮动，无法浮动到窗口外部。

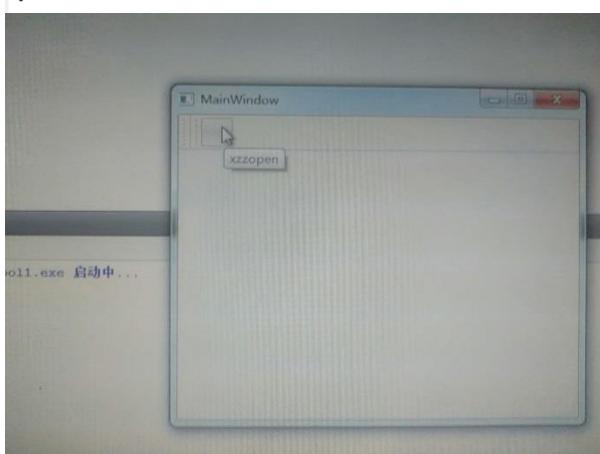
现在我不想菜单栏浮动

```

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),ui
{
    ui->setupUi(this);
    QToolBar* tb = addToolBar("xzz_tool");//创建工具栏对象
    QAction* action = new QAction("",NULL);//创建
    action->setToolTip("xzzopen");//工具栏里面图标的提示字符

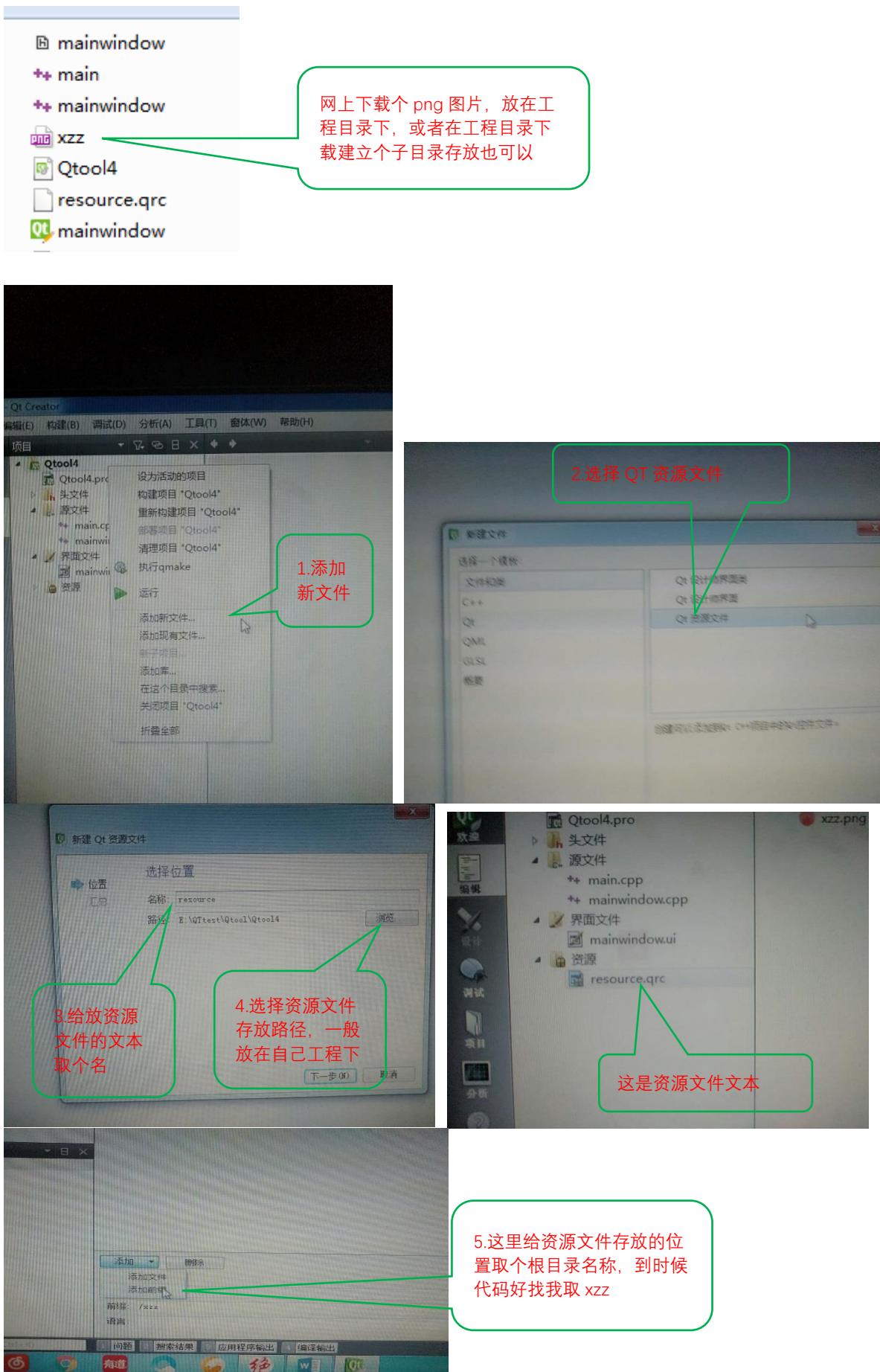
    tb->setFloatable(false);
    tb->setMovable(false); //禁止工具栏在窗口内悬浮，直接在窗口上方卡死
    tb->addAction(action);
}

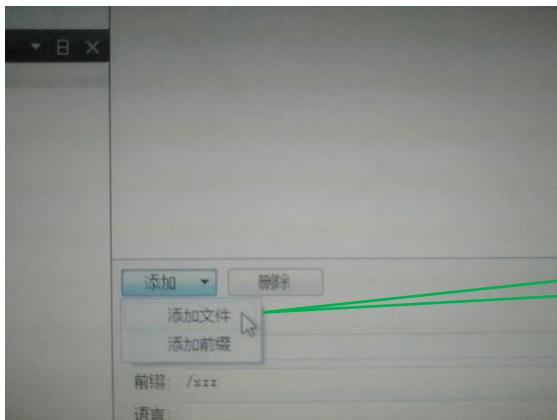
```



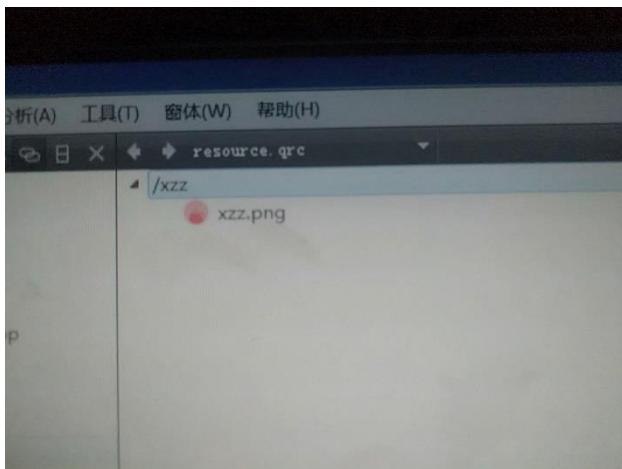
菜单栏卡死在窗口上边，无法浮动

我觉得菜单栏的图标框没有图形不好看，我们加点图形





6.在添加你存放的 png 图片



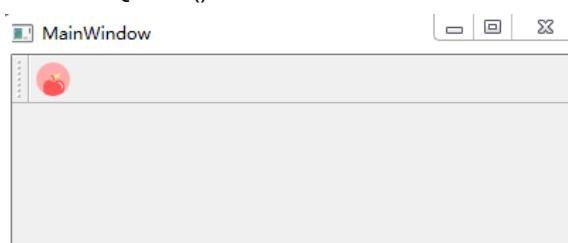
图片添加成功

```
#include <QToolBar> //工具栏头文件
#include <QAction>
#include <QIcon> //添加图标头文件
MainWindow::MainWindow(QWidget *parent) : QMainWindow(par
{
    ui->setupUi(this);
    QToolBar* tb = addToolBar("xzz_tool"); //创建工具栏对象
    QAction* action = new QAction("",NULL); //创建
    action->setToolTip("xzzopen"); //工具栏里面图标的提示字符
    action->setIcon(QIcon(":/xzz/xzz.png")); //用 QIcon 函数指定资源图片的位置，将其提取出来
    tb->setFloatable(false);
    tb->setMovable(false);
    tb->addAction(action);
}
```

用 QIcon 函数指定资源图片的位置，将其提取出来

这里的冒号指
路径，前缀名
xzz 添加的
xzz.png 图片

这里的 QIcon() 生成的对象用 setIcon 传递给 QAction 图标框



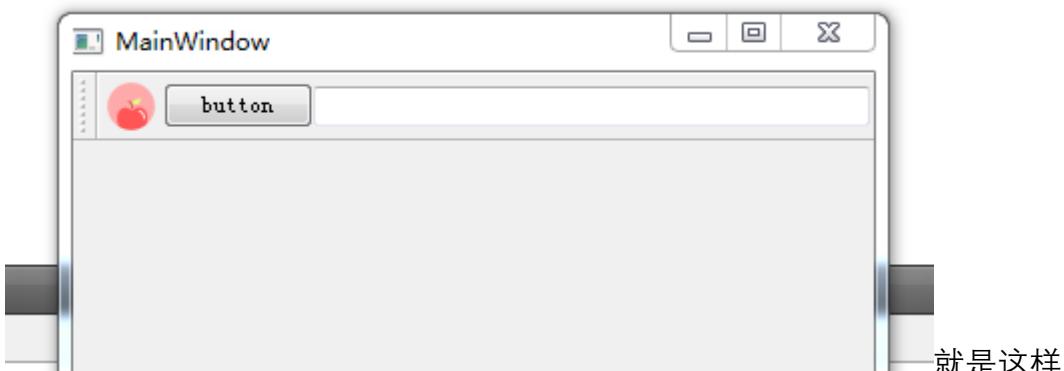
图标里面有图片了

工具栏除了加图标组件以外，还可以加按钮组件，QWidget 窗口组件等等…

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QToolBar* tb = addToolBar("xzz_tool");//创建工具栏对象
    QAction* action = new QAction("",NULL);//创建
    action->setToolTip("xzzopen");//工具栏里面图标的提示字符
    action->setIcon(QIcon(":/xzz/xzz.png"));

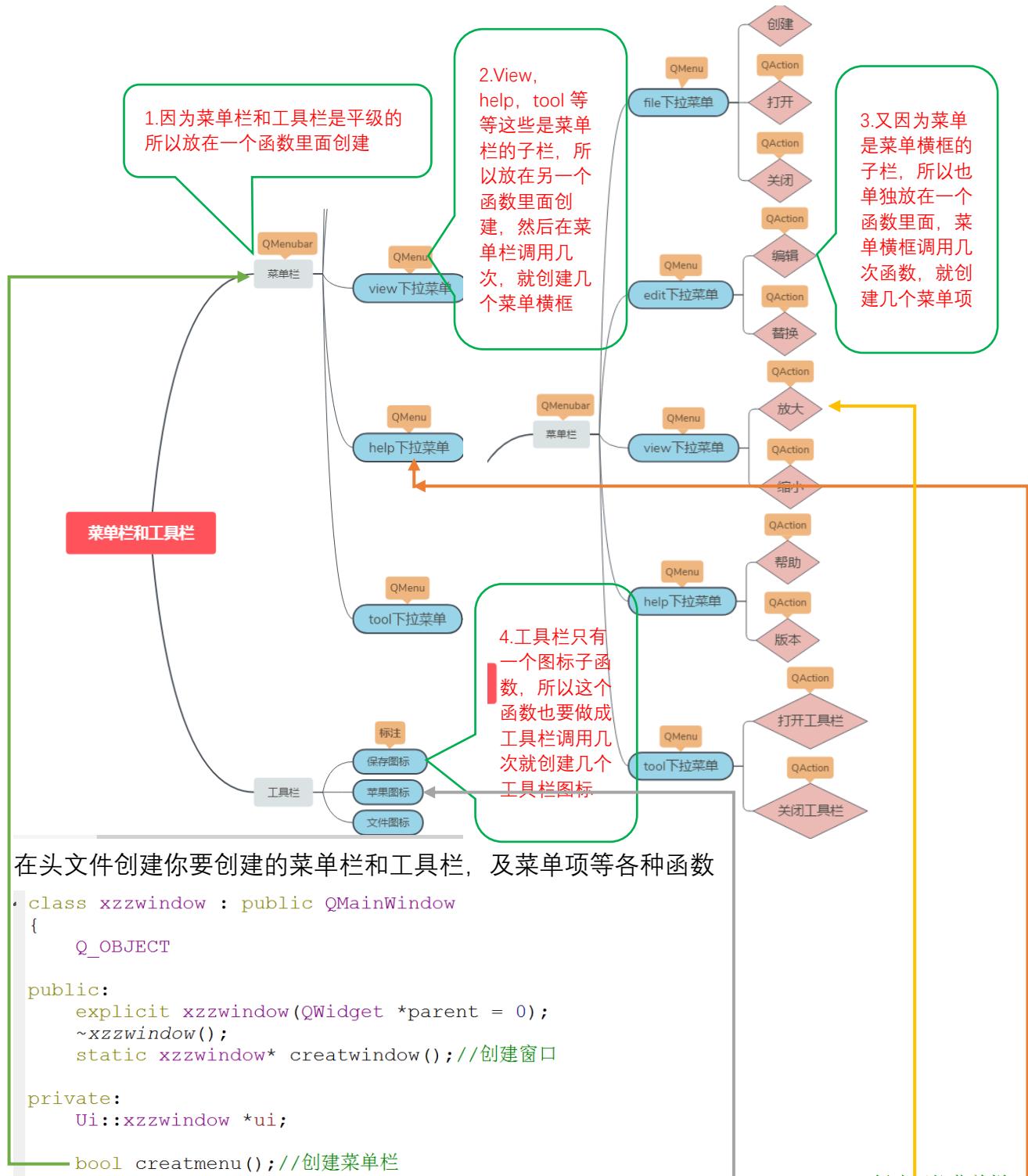
    tb->setFloatable(false);
    tb->setMovable(false);
    tb->addAction(action);//action对象的东西用QToolBar的addAction函数添加

    QPushButton* button = new QPushButton("button");
    QLineEdit* edit = new QLineEdit();
    tb->addWidget(button);//QPushButton, QLineEdit, QLabel这些东西用
    tb->addWidget(edit);//QToolBar的addWidget函数添加
}
```



就是这样

菜单栏和工具栏综合开发



下面是函数的实现

```

xzzwindow* xzzwindow::creatwindow()
{
    xzzwindow* ret = new xzzwindow();
    if(ret == NULL)
    {
        delete ret;
        ret = NULL;
    }
    ret->creatmenu();
    return ret;
}

bool xzzwindow::creatmenu()
{
    bool ret = true;
    QMenuBar* mb = menuBar(); // 创建菜单横框

    if(mb == NULL)
        ret = false;
    else
    {
        creatxialamenu(mb, "File (&F)", "New,Open,close,xzz1,xzz3,xzz5");
        creatxialamenu(mb, "Edit (&E)", "edit,exit,replace,xzz2,xzz4,xzz5");
        creatxialamenu(mb, "View (&V)", "amp,shrink,see,xzz2,xzz4,xzz5");
        creatxialamenu(mb, "Help (&H)", "doc,readme,read,xzz2,xzz4,xzz5");
        creatxialamenu(mb, "Tool (&T)", "bug,debuq,write,xzz2,xzz4,xzz5");
    }
    QToolBar* tb = addToolBar("xzz_tool"); // 创建工具栏

    if(tb == NULL)
        ret = false;
    else
    {
        ret = creattool(tb, "save",(":/res/hong.png"));
        ret = creattool(tb, "apple",(":/res/apple.png"));
        ret = creattool(tb, "file",(":/res/file.png"));
    }
    return ret;
}

```

1.我先创建个主窗口

2.在主窗口里面创建菜单栏

3.你需要几个下拉菜单就调用几次这个函数

4.因为下拉菜单是创建在菜单横框里面的，所以这里要写入菜单横框 mb

5.每个下拉菜单都有很多菜单项，你写多少段字符就有多少个菜单项，用逗号分隔

创建工具栏

鼠标移动到工具栏图标名称

给工具栏图标加图片

这些函数都是我自己封装的，至于封装细节看下面

获取字符串指定字符的个数

```
/*获取字符串指定字符个数*/
int xzzwindow::Stringchargetnum(QString str,QChar c)
{
    int len=0,ret=0;
    len = str.length();
    for(int i=0;i<len;i++)
    {
        if(str[i] == c)
        {
            ret++;
        }
    }
    return ret;
}
```

获取字符串里面用指定符号分割后得到的字符串段落

```
/*字符串段落自动提取函数，返回的字符串数组放在retstr里面，retstr[i]就是提取的段落*/
int xzzwindow::xzzStringcut(QString str,QString *retstr,QChar c)
{
    int num = Stringchargetnum(str,c);
    for(int i=0;i<num+1;i++)
    {
        retstr[i] = str.section(c,i,i); //section是QString的库函数
    }
    return num;
}
```

有了字符串处理函数后就好办了

创建工具栏图标的函数

```
bool xzzwindow::creattool(QToolBar* tb,QString str,QString icon)
{
    bool ret = true;
    QAction* action = new QAction("",NULL);
    if(action == NULL)
        ret = false;
    else
    {
        action->setToolTip(str);
        action->setIcon(QIcon(icon));
        tb->addAction(action);
    }
    return ret;
}
```

```
bool xzzwindow::creatxialamenu(QMenuBar *mb,QString menustr,QString actstr)
{
    bool ret = true;
    QMenu *menu = new QMenu(menustr);
    if(menu == NULL)
        ret = false;
    else
    {
        creatcaidanpro(menu,actstr);
        mb->addMenu(menu);
    }

    return ret;
}

bool xzzwindow::creatcaidanpro(QMenu* menu,QString actstr)
{
    bool ret = true;
    int num = Stringchargetnum(actstr,',');//获取字符串逗号个数
    QString strbuff[num+1];
    num = xzzStringcut(actstr,strbuff,',');
    for(int i=0;i<num+1;i++)
    {
        QAction *action = new QAction(strbuff[i],NULL);
        if(action == NULL)
            ret = false;
        else
            menu->addAction(action);

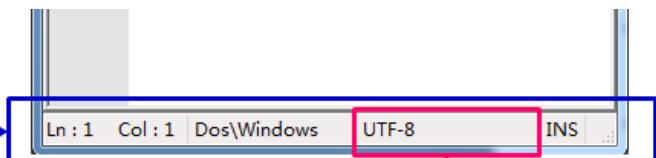
        menu->addSeparator();
    }
    return ret;
}
```

创建下拉菜单函数，同时调用了创建菜单项函数

至于细节分析自己思考

状态栏创建

- 状态栏 (`QStatusBar`)



- 任意组件 (`QWidget`)

状态栏 QT 规定是设置在窗口下边的

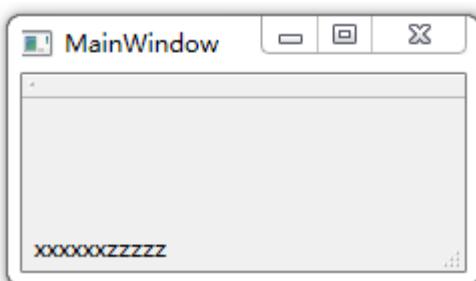
```
QStatusBar* sb = statusBar();  
  
/* create widget for Status Bar */  
  
QLabel* l = new QLabel("Label");  
QLineEdit* e = new QLineEdit();  
  
/* add widget to Status Bar */  
  
sb->addPermanentWidget(l);  
  
sb->addPermanentWidget(e);
```

1. 创建状态栏

2. 状态栏对象可以装各种控件和文本，比如按钮，编辑框，标签等等…

3. addpermanentwidget 函数是将各种传入的对象放在状态栏右边

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),  
ui(new Ui::MainWindow)  
{  
    ui->setupUi(this);  
    QStatusBar* sb = statusBar(); // 创建状态栏  
    sb->showMessage("xxxxxxxxzzzz"); // 在状态栏左下角显示消息  
}
```



Showmessage 函数就是规定在状态栏左边显示你的字符

```

#include <QStatusBar> //状态栏头文件
#include <QLabel> //标签
#include <QLineEdit> //文本框
#include <QPushButton> //按钮

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QStatusBar* sb = statusBar(); //创建状态栏
    QLabel* l = new QLabel("label"); //创建标签
    QLineEdit* le = new QLineEdit(); //创建文本框
    QPushButton* button = new QPushButton(); //创建按钮

    sb->addPermanentWidget(l); //将标签加入状态栏
    sb->addPermanentWidget(le); //将文本框加入状态栏
    sb->addPermanentWidget(button); //将按钮加入状态栏

    sb->showMessage("xxxxxxxxzzzz"); //在状态栏左下角显示消息
}

```



因为 addpermanentWidget 就是将对象放入状态栏右边，排列顺序按照创建顺序来，从左到右

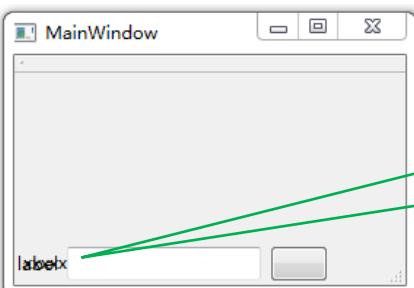
```

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QStatusBar* sb = statusBar(); //创建状态栏
    QLabel* l = new QLabel("label"); //创建标签
    QLineEdit* le = new QLineEdit(); //创建文本框
    QPushButton* button = new QPushButton(); //创建按钮

    sb->addWidget(l); //addWidget 将对象放在状态栏左边
    sb->addWidget(le);
    sb->addWidget(button);

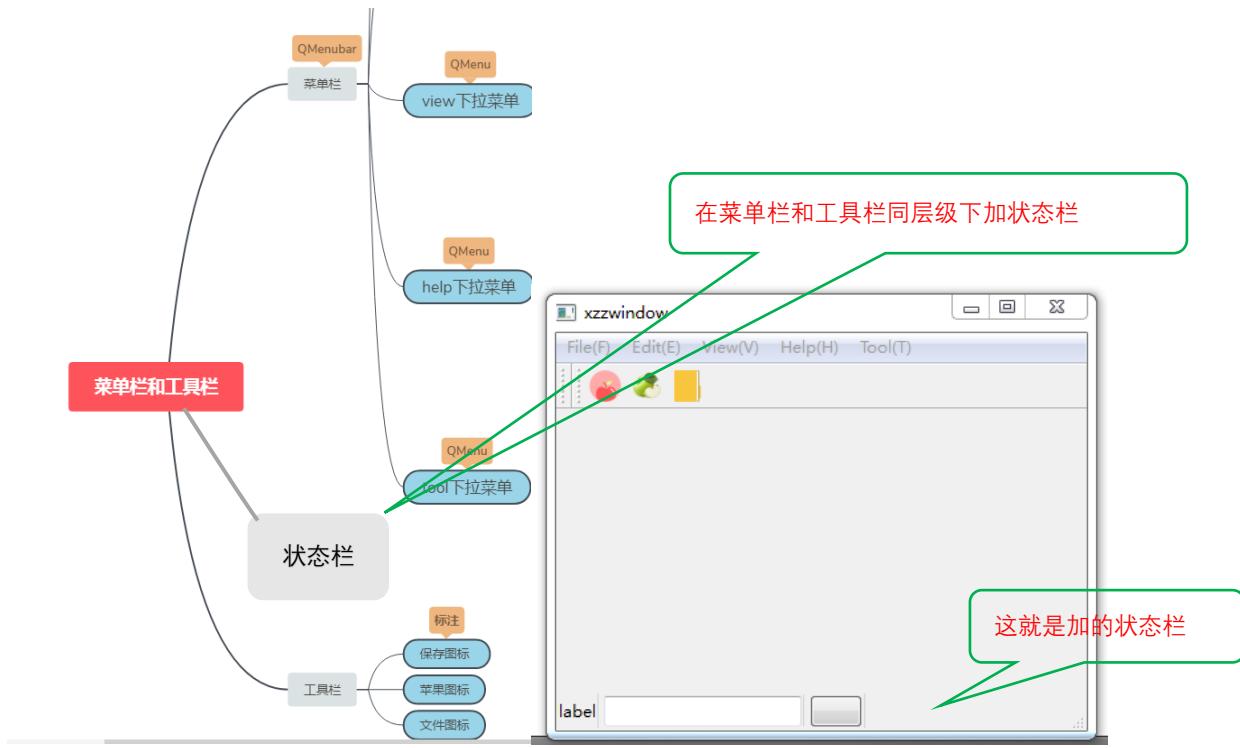
    sb->showMessage("xxxxxxxxzzzz"); //在状态栏左下角显示消息
}

```



对象放在左边了，但是标签和消息的字符重合了，你只需要把 showMessage 去掉就看得清楚了

在上面菜单栏和工具栏综合开发项目里加入状态栏



```
class xzzwindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit xzzwindow(QWidget *parent = 0);
    ~xzzwindow();
    static xzzwindow* creatwindow(); // 创建窗口

private:
    Ui::xzzwindow *ui;

    bool creatmenu(); // 创建菜单栏
    bool creatxialamenu(QMenuBar *mb, QString menustr, QString acts
    bool creatcaidanpro(QMenu* menu, QString actstr); // 创建菜单项
    int Stringchargetnum(QString str, QChar c);
    int xzzStringcut(QString str, QString *retstr, QChar c);
    bool creattool(QToolBar* tb, QString str, QString icon);
    bool initstatus(); // 创建状态栏
};

bool xzzwindow::initstatus() // 实现状态栏
{
    QStatusBar* sb = statusBar(); // 创建状态栏
    QLabel* l = new QLabel("label"); // 创建标签
    QLineEdit* le = new QLineEdit(); // 创建文本框
    QPushButton* button = new QPushButton(); // 创建按钮

    sb->addWidget(l);
    sb->addWidget(le);
    sb->addWidget(button);
}
```

初始化程序调用就是了

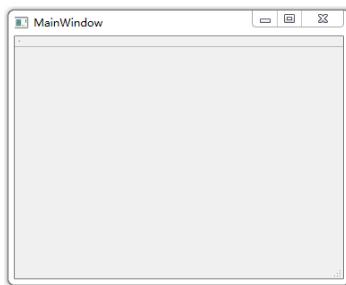
单行文本和多行文本，富文本使用

- QLineEdit
 - 单行文本编辑组件

- QTextEdit
 - 多行富文本编辑组件
- QPlainTextEdit
 - 多行普通文本编辑组件

	单行文本支持	多行文本支持	自定义格式支持	富文本支持
QLineEdit	Yes	No	No	No
QPlainTextEdit	Yes	Yes	No	No
QTextEdit	Yes	Yes	Yes	Yes

```
MainWindow::MainWindow(QWidget *parent) :  
    QMainWindow(parent),  
    ui(new Ui::MainWindow)  
{  
    ui->setupUi(this);  
    resize(600, 400); // 设置父窗口大小  
    Ledit.move(20, 20); // 设置单行文本的绝对定位坐标  
    Ledit.resize(560, 150); // 设置单行文本的大小  
  
}
```



怎么没有看到文本框？这个跟在主函数里面实现还是在子函数里面实现有关

```
int main(int argc, char *argv[])  
{  
    QApplication a(argc, argv);  
    QWidget w;  
    QLineEdit *l = new QLineEdit(&w); // 将文本框嵌套进父窗口  
    l->move(100, 100);  
    l->resize(50, 50);  
    w.show();  
    return a.exec();  
}
```

在 QLineEdit 章节，我们在创建单行文本时把父窗口对象指定进来了



但是我们在子文件创建的单行文本没法指定主文件定义的父窗口对象

那我们只有用 this 指针
在头文件静态定义三个文本

```
#include <QMainWindow>
#include <QLineEdit>/>单行文本
#include <QPlainTextEdit>/>多行文本
#include <QTextEdit>/>富文本
```

~~~~~

```
Ui::MainWindow *ui;
QLineEdit Ledit;
QPlainTextEdit ptedit;
QTextEdit qtedit;
```

单行文本

多行文本

富文本

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent), Ledit(this), ptedit(this), qtedit(this),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    resize(600, 400);
    Ledit.move(20, 20); //设置单行文本坐标
    Ledit.resize(560, 100); //设置单行文本大小

    ptedit.move(20, 130); //设置多行文本坐标
    ptedit.resize(560, 130); //设置多行文本大小

    qtedit.move(20, 270); //设置富文本坐标
    qtedit.resize(560, 130); //设置富文本大小
}
```

因为 MainWindow 是父窗口，我们在初始化的时候用 this 指定  
就相当于把&w 写入 ledit...，this 就代表上面案例的&w

单行文本只能写一  
行，不管你多大的  
框，你回车就是不换  
行



富文本下面来演示

```

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent), Ledit(this), ptedit(ui
ui(new Ui::MainWindow)

{
    ui->setupUi(this);
    resize(600, 400);
    Ledit.move(20, 20);
    Ledit.resize(560, 100);
    Ledit.insert("QLineEdit");

    ptedit.move(20, 130);
}

```

```

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent), Ledit(this), ptedit(this), qtedit(this),
ui(new Ui::MainWindow)

{
    ui->setupUi(this);
    resize(600, 400);
    Ledit.move(20, 20);
    Ledit.resize(560, 100);
    Ledit.insert("QLineEdit");
    Ledit.insert("\n");
    Ledit.insert("<img src=\"E:\\QTtest\\Qtext\\QText4\\apple.png\"/>");

    ptedit.move(20, 130);
    ptedit.resize(560, 130);
    ptedit.insertPlainText("ptedit");
    ptedit.insertPlainText("\n");
    ptedit.insertPlainText("<img src=\"E:\\QTtest\\Qtext\\QText4\\apple.png\"/>");

    qtedit.move(20, 270);
    qtedit.resize(560, 130);
    qtedit.insertPlainText("qtedit");
    qtedit.insertPlainText("\n");
    qtedit.insertPlainText("<img src=\"E:\\QTtest\\Qtext\\QText4\\apple.png\"/>");
}

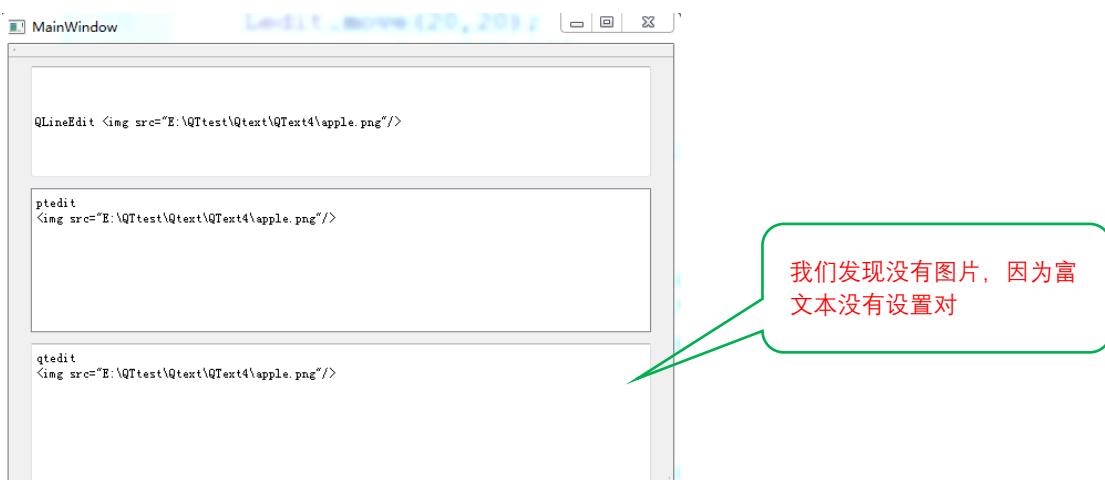
```

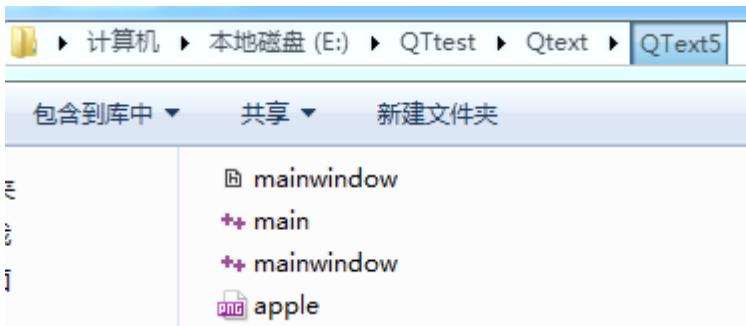
可以默认换行

可以使用 html 语言  
加入你的图片

这是图片在 windows 上的存放路径

其实图片的读取只有富文本有效

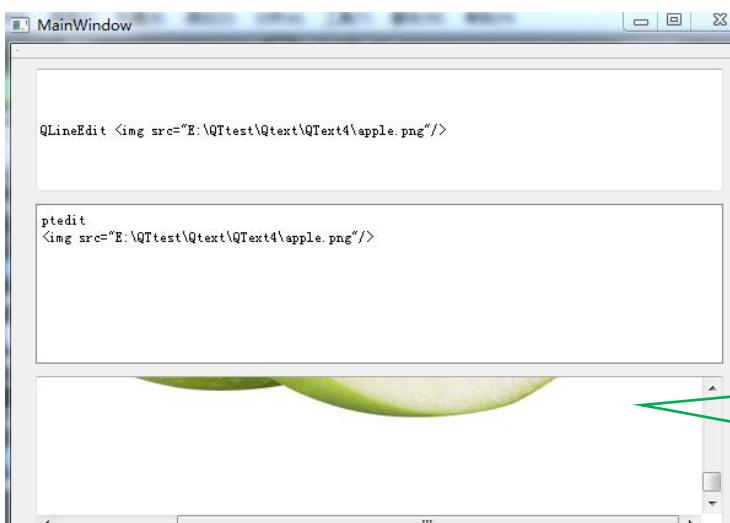




png 图片路径

```
ptedit.move(20,130);
ptedit.resize(560,130);
ptedit.insertPlainText("ptedit");
ptedit.insertPlainText("\n");
ptedit.insertPlainText("<img src=\"E:\\QTtest\\Qtext\\QText4\\apple.png\"");

qtedit.move(20,270);
qtedit.resize(560,130);
qtedit.insertPlainText("qtedit");
qtedit.insertPlainText("\n");
qtedit.insertHtml("<img src=\"E:\\QTtest\\Qtext\\QText4\\apple.png\"/>");
```

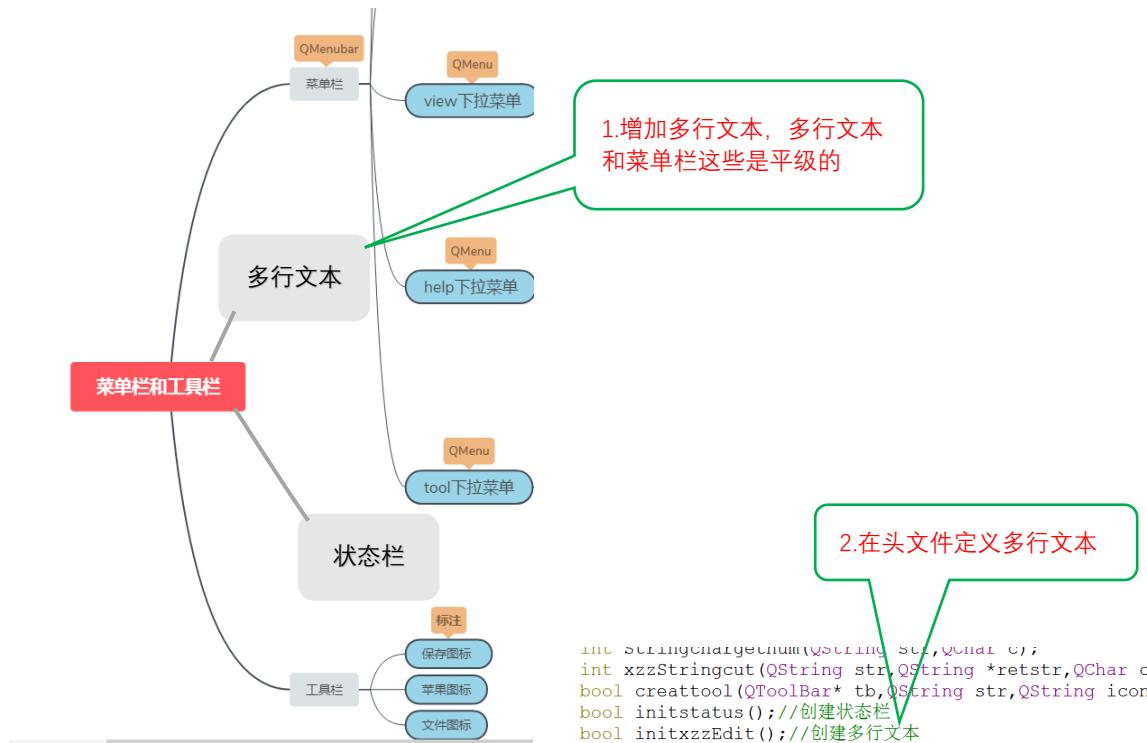


富文本应该用 insertHtml 函数来导入图片

因为图片太大，所以只能这样了

单行文本用于传输数据，  
多行文本用于编辑文字，写小说  
富文本用于做 word 软件

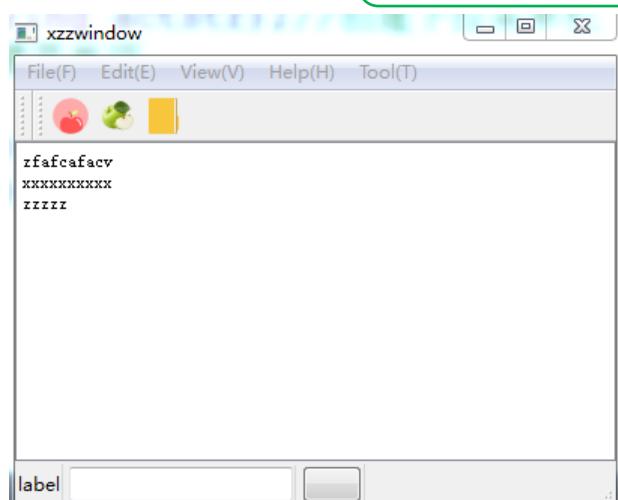
## 在上面菜单栏，工具栏，状态栏，综合开发项目里加入多行文本



```
bool xzzwindow::initxzzEdit()
{
    bool ret = true;
    QPlainTextEdit* ptedit = new QPlainTextEdit();
    if(ptedit == NULL)
        ret = false;
    else
    {
        ptedit->setParent(this); // 将多行文本嵌入进父窗口 this就是父对象
        setCentralWidget(ptedit); // 将多行文本设置在父窗口中央
    }
    return ret;
}
```

SetCentralWidget 是设置你的对象在窗口中心区域，而且还可以跟着窗口自动放大缩小，这样就不用设置文本大小了

```
int StringChangeNum(QString str,QChar c);
int xzzStringCut(QString str,QString *retstr,QChar c);
bool creattool(QToolBar* tb,QString str,QString icon);
bool initstatus(); // 创建状态栏
bool initxzzEdit(); // 创建多行文本
```



## 文件创建，打开，写入，读取，操作方法，

头文件 QFile

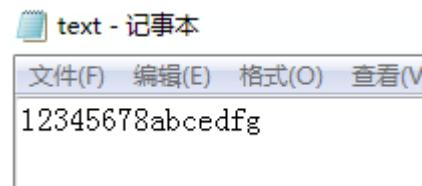
```
void write(QString f)
{
    QFile file(f);
    if(file.open(QIODevice::WriteOnly | QIODevice::Text))
    {
        file.write("12345678");//向文件里面写内容，然后换行
        file.write("abcdedfg");
        file.close();
    }
}
```

用 write 函数想文件写内容，最后写完记得关闭文件

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    write("E:/QTtest/text.txt");//注意必须是英文路径
}
```



文件创建成功



打开文件发现内容没错，为什么没有换行呢？

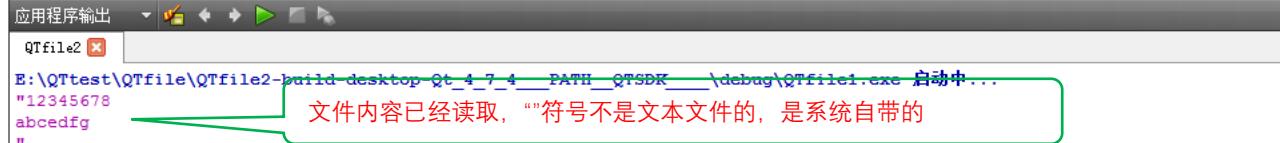
```
void write(QString f)
{
    QFile file(f);
    if(file.open(QIODevice::WriteOnly | QIODevice::Text))
    {
        file.write("12345678\n");//向文件里面写内容，然后换行
        file.write("abcdedfg\n");
        file.close();
    }
}
```



换行要在 write 写文件的时候加\n

执行程序后，新的 txt 文件会覆盖以前的 txt 文件

```
3 #include <QFile> //文件操作头文件
4 #include <QByteArray> //字节数组头文件，用来存放读取文件的数据流
5 #include <QDebug>
6
7 void read(QString f)
8 {
9     QFile file(f);
10    if(file.open(QIODevice::ReadOnly | QIODevice::Text))
11    {
12        QByteArray ba = file.readAll(); //读取文件里面所有内容
13        QString s(ba); //将数据流二进制转换成字符串
14        qDebug() << s;
15        file.close();
16    }
17 }
18
19 int main(int argc, char *argv[])
20 {
21
22     QApplication a(argc, argv);
23     MainWindow w;
24     w.show();
25     read("E:/QTtest/text.txt"); //注意必须是英文路径
26 }
```



```
6
7 void read(QString f)
8 {
9     QFile file(f);
10    if(file.open(QIODevice::ReadOnly | QIODevice::Text))
11    {
12        QByteArray ba = file.read(4); //读取文本指定4个字符长度的数据
13        QString s(ba); //将数据流二进制转换成字符串
14        qDebug() << s;
15        file.close();
16    }
17 }
18
```



文件读取一行

```

7 void read(QString f)
8 {
9     QFile file(f);
10    if(file.open(QIODevice::ReadOnly | QIODevice::Text))
11    {
12        QByteArray ba = file.readLine(); //读取文本一行数据
13        QString s(ba); //将数据流二进制转换成字符串
14        qDebug() << s;
15        file.close();
16    }
17 }

```

应用程序输出 QTfile4

```

E:\QTtest\QTfile\QTfile4-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\QTfile1.exe 启动中...
"12345678"
E:\QTtest\QTfile\QTfile4-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\QTfile1.exe 退出, 退出代码: 0

```

## 文件信息获取

```

#include <QtGui/QApplication>
#include "mainwindow.h"
#include <QFile> //文件操作头文件
#include <QDebug>
#include <QFileInfo> //读取文件属性的头文件
#include <QDateTime> //info里面的函数需要依赖DateTime头文件

```

```

8 void TextInfo(QString f)
9 {
10    QFile file(f); //先找到文件路径
11    QFileInfo info(file); //把找到路径的文件属性提取出来
12
13    qDebug() << info.exists(); //文件是否存在, 存在返回true
14    qDebug() << info.isFile(); //文件是文件夹还是文件, 是文件返回true
15    qDebug() << info.isReadable(); //文件是否只读, 是只读返回true
16    qDebug() << info.isWritable(); //文件是否可写, 是可写返回true
17    qDebug() << info.created(); //文件什么时候创建
18    qDebug() << info.lastRead(); //文件被访问的时间
19    qDebug() << info.lastModified(); //文件最后一次被修改时间
20    qDebug() << info.path(); //文件路径
21    qDebug() << info.fileName(); //文件名字
22    qDebug() << info.suffix(); //文件后缀
23    qDebug() << info.size(); //文件大小
24 }
25 int main(int argc, char *argv[])
26 {
27     QApplication a(argc, argv);
28     MainWindow w;
29     w.show();
30     TextInfo("E:/QTtest/text.txt"); //注意必须是英文路径

```

应用程序输出 QTfile5

```

E:\QTtest\QTfile\QTfile5-build-desktop-Qt_4_7_4__PATH__QTSDK__\debug\QTfile1.exe 启动中...
true
true
true
true
QDateTime("周日 七月 15 15:49:00 2018")
QDateTime("周日 七月 15 15:59:06 2018")
QDateTime("周日 七月 15 16:08:21 2018")
"E:/QTtest"
"text.txt"
"txt"
19

```

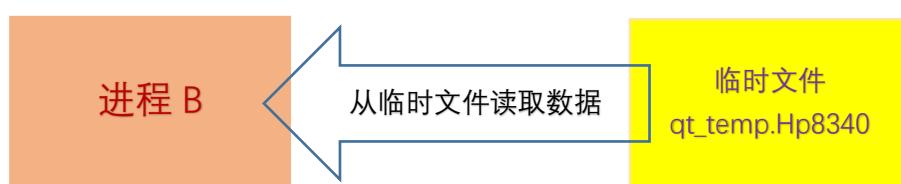
## 临时文件操作类进行进程之间通信，传数据

```
#include <QFile> //文件操作头文件
#include <QDebug>
#include <QTemporaryFile> //临时文件操作类
#include <QDebug>
#include <QFileInfo>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QTemporaryFile tempfile;
    //定义个临时文件，不需要写文件名，因为文件名系统会创建一个唯一的
    MainWindow w;
    w.show();

    if(tempfile.open()) //打开临时文件
    {
        tempfile.write("xxxxzzzz"); //打开成功向临时文件写数据
        tempfile.close();
    }
    QFileInfo info(tempfile); //因为临时文件名是唯一的，所以只有用info来查看
    qDebug() << info.isFile();
    qDebug() << info.path();
    qDebug() << info.fileName();
    return a.exec();
```

```
E:\QTtest\QTfile\QTfile6-build-desktop-
true
"C:/Users/XZZO/AppData/Local/Temp"
"qt_temp.Hp4980"
```

在打印的目录下找到了 qt\_temp.Hp8340 文件



所以临时文件是进程之间传递小数据，或者传递大数据用的

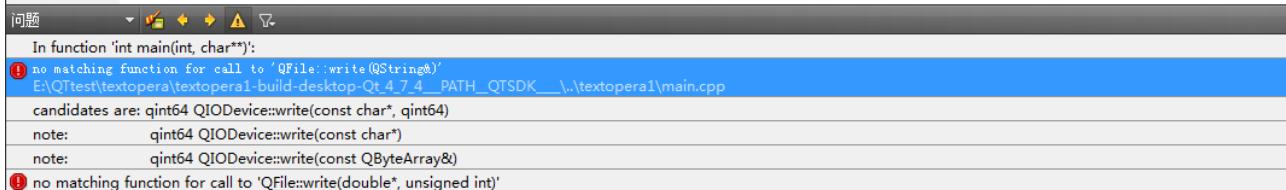
## 文本流数据流读写 IO 设备数据

创建一个二级制 hex 文件，向里面写数据

### QFile 直接支持文本文件和数据文件的读写

- qint64 `read(char* data, qint64 maxSize)`  
这个要求的是按照字节位来读取
- `QByteArray read(qint64 maxSize)`  
这个要求的是按照字节位来写
- qint64 `write(const char* data, qint64 maxSize)`
- qint64 `write(const QByteArray& byteArray)`

```
2 #include <QFile>
3 int main(int argc, char *argv[])
4 {
5     QApplication a(argc, argv);
6     QFile file("E:/QTtest/text.hex");//在指定路径下创建hex二进制文件
7
8     QString str = "xxxxzzstr";
9     double value = 3.14;
10    if(file.open(QIODevice::ReadOnly))
11    {
12
13
14        file.write(str);//将字符串写入二级制文件
15        file.write(&value, sizeof(value));//将小数写入二进制文件
16        file.close();
17    }
18    return a.exec();
19 }
20 }
```



所以我们要对变量进行转换，转换成字节类型

```

#include <QtCore/QCoreApplication>
#include <QFile>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile file("E:/QTtest/text.hex");//在指定路径下创建hex二进制文件

    if(file.open(QIODevice::WriteOnly))
    {
        QString str = "xxxxzzstr";
        double value = 3.14;

        file.write(str.toStdString().c_str());
        file.write(reinterpret_cast<char*>(&value), sizeof(value));
        file.close();
    }
    return a.exec();
}

```

用 toStdString 将 QString 字符串转换成 String 类型

c\_str 将 String 对象转换成 char\*类型

将 double 转换成 char\*类型的二级制，然后根据 sizeof 取的 double 大小去一个字节一个字节的写二进制数

QFile 直接支持文本文件和数据文件的读写

- qint64 read(char\* data, qint64 maxSize)
- QByteArray read(qint64 maxSize)
- qint64 write(const char\* data, qint64 maxSize)
- qint64 write(const QByteArray& byteArray)

就是为了满足 char\* data 的形参条件

### **std::string QString::toStdString () const**

toStdString 函数：  
成 String

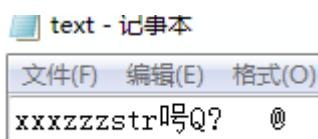
将 QString 转换

返回值 string

注意：使用 toStdString 或者使用 toStdWString 会出错，这个可能是有些版本的 qt 的 bug，编译是能编过，但是运行时会段错误。你改为 toLocal8bit().data() 这样获取到 const char\* 类型的数据就没问题了  
String 类成员函数 c\_str() 的原型：

const char \*c\_str() const; // 返回一个以 null 终止的 c 字符串

c\_str() 函数返回一个指向正规 c 字符串的指针，内容和 string 类的本身对象是一样的，通过 string 类的 c\_str() 函数能够把 string 对象转换成 c 中的字符串的样式；



程序执行后：

确实得到了变成二进制的字符串和 double 数据

```

#include <QFile>
#include <QDebug>
int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);
    QFile file("E:/QTtest/text.hex"); //读取指定路径的二进制文件

    QString str1;
    double value = 0;
    if(file.open(QIODevice::ReadOnly))
    {
        str1 = file.read(9); //二进制文本前9个字节是字符
        file.read(reinterpret_cast<char*>(&value), sizeof(value)); //读取接在后面的double数据
        file.close();
    }
    qDebug() << str1;
    qDebug() << value;
    return a.exec();
}

```

将 double 转换成 char\*类型的二级制，  
然后根据 sizeof 取的 double 大小去一个  
字节一个字节的读出来

```
D:\QT\bin\qtc
"xxxxzzzstr"
3.14
```

你看读出的 text.hex 文本和写入时的一样。

上面这种写用 QFile 类的 write, read 函数还需要数据转换太麻烦了，我们整个简单点的 QTextStream 类使用，读写 txt 文本文件

### - QTextStream

- 写入的数据全部转换为可读文本

```

#include <QFile>
#include <QDebug>
#include <QTextStream> //辅助输入输出类
int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);
    QFile file("E:/QTtest/text.txt"); //创建文本文件路径

    QString str = "xxxxxxxx";
    int value = 20;
    double pi = 3.14;
    if(file.open(QIODevice::WriteOnly | QIODevice::Text)) //文本方式打开
    {
        QTextStream out(&file); //创建辅助输入输出类
        out << str << endl; //我们可以直接将字符串用移位方式写入文本，类似C++的cout, cin
        out << value << endl; //将int数据写入文本， endl可以文本换行
        out << "pi = " << pi << endl; //也可以文本不换行，挨着写字符串和变量。
    }
    file.close();
}

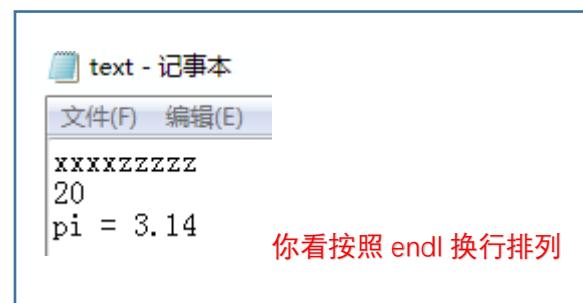
return a.exec();
}

```

 文本创建，打开文本

用 QTextStream 读出文本

主要就是这个



你看按照 endl 换行排列

```

#include <QFile>
#include <QTextStream> //辅助输入输出类
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile file("E:/QTtest/text.txt"); //创建文本文件路径

    QString rstr1;
    QString rstr2;
    QString rstr3;
    if(file.open(QIODevice::ReadOnly | QIODevice::Text)) //读文本字符
    {
        QTextStream in(&file); //创建辅助输入输出类
        rstr1 = in.readLine(); //读取字符串，用QString接受
        rstr2 = in.readLine(); //写入文本时是整形，但是在文本存放后就是字符串，用QString接受
        rstr3 = in.readLine(); //写入文本时是浮点型，但是在文本存放后就是字符串，用QString接受
        file.close();
    }

    qDebug() << rstr1;
    qDebug() << rstr2;
    qDebug() << rstr3;
    return a.exec();
}

```

执行一次 readLine 就读取文本一行，连着执行几次 readLine 就跟着向下读取几行

```

"xxxxzzzz"
"20"
"pi = 3.14"

```

输出正确

如果你觉得 readLine 写的太多太累，可以用循环

`bool atEnd() const;` 返回 false 没有读取完，返回 true 读取完毕

```

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile file("E:/QTtest/text.txt"); //创建文本文件路径

    QString line;

    if(file.open(QIODevice::ReadOnly | QIODevice::Text)) //读文本字符
    {
        QTextStream in(&file); //创建辅助输入输出类
        while(!in.atEnd()) //判断文本是不是读取到最后一行
        {
            line = in.readLine();
            qDebug() << line; //这种情况就只有读一行显示一行，因为line会被覆盖
        }
        file.close();
    }

    return a.exec();
}

```

atEnd 判断文本是不是读取到最后一行

```

"xxxxzzzz"
"20"
"pi = 3.14"

```

QDataStream 类使用，读写二级制文件，注意（QDataStream 和 QT 版本有兼容性问题）

## QDataStream

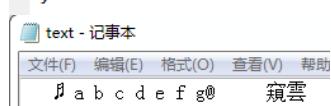
- 写入的数据根据类型转换为二进制数据

```
#include <QDataStream> //二级制文件读写类
int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);
    QFile file("E:/QTtest/text.bin"); //创建二级制文件路径

    QString str = "abcdefg";
    double f = 3.14;

    if(file.open(QIODevice::WriteOnly)) //写二进制文件
    {
        QDataStream data_out(&file); //输出二级制数据到二进制文件
        data_out<<str; //将字符串转换成二级制写入二级制文件
        //这里不需要换行，因为是二级制写入，所以这行加；号，自动换行
        data_out<<f; //将小数转换成二级制写入二进制文件

        file.close();
    }
    return a.exec();
}
```



这就是生成的二进制文件，一堆乱码

```
#include <QDataStream> //二级制文件读写类
int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);
    QFile file("E:/QTtest/text.bin"); //找到二级制文件路径

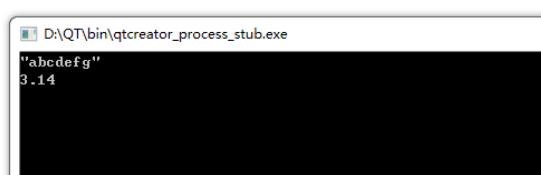
    QString str;
    double f=0;

    if(file.open(QIODevice::ReadOnly)) //读二进制文件
    {
        QDataStream data_in(&file); //创建读取二级制文件的类
        data_in>>str; //二进制文件数据写入str变量
        data_in>>f; //二进制文件数据写入f变量

        file.close();
    }
    qDebug()<<str;
    qDebug()<<f;
```

写二进制文件

读二进制文件



我们前面说了 QDataStream 在不同版本 QT 会出现不同兼容的问题

## setVersion 函数使用

```
#include <QDataStream> //二级制文件读写类
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile file("E:/QTtest/text.bin"); //创建二级制文件路径

    QString str = "abcdefg";
    double f = 3.14;

    if(file.open(QIODevice::WriteOnly)) //写二进制文件
    {
        QDataStream data_out(&file); //输出二级制数据到二进制文件
        data_out.setVersion(QDataStream::Qt_4_7); //在写入二进制数据之前先写入版本
        data_out<<str;
        data_out<<f;

        file.close();
    }
}
```

写二级制文件之前要先写入版本

```
#include <QDataStream> //二级制文件读写类
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile file("E:/QTtest/text.bin"); //找到二级制文件路径

    QString str;
    double f=0;

    if(file.open(QIODevice::ReadOnly)) //读二进制文件
    {
        QDataStream data_in(&file); //读二进制文件类
        data_in.setVersion(QDataStream::Qt_4_7); //版本读取
        data_in>>str;
        data_in>>f;

        file.close();
    }
    qDebug()<<str;
}
```

读二级制文件之前要读入版本

## QT 缓冲区变量，用于多线程数据通信



缓冲区就是申请一块连续的内存空间，用变量来指定这块空间区域。

所以我们申请一个字节数组对象 QByteArray，来当做这块连续内存

### - QBuffer 是 Qt 中缓冲区相关的类

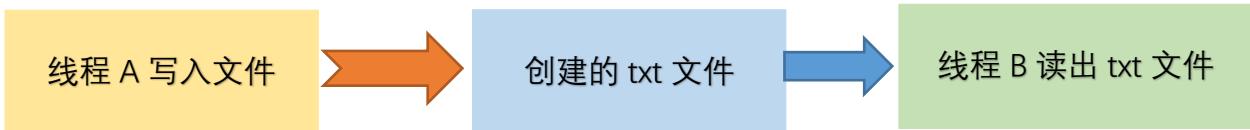
用 Qbuffer 将 QByteArray 关联在一起

```
2 #include <QBuffer> //缓冲区头文件
3 #include <QByteArray> //字节数组
4 #include <QDataStream>
5 #include <QDebug>
6 int main(int argc, char *argv[])
7 {
8     QApplication a(argc, argv);
9     QByteArray array;
10    QBuffer buffer(&array);
11
12    if (buffer.open(QIODevice::WriteOnly)) //将数据写入缓冲区buffer
13    {
14        QDataStream out(&buffer);
15        out<<QString("xxxxxxxx");
16        out<<3.145;
17        out<<65535;
18        buffer.close();
19    }
20
21
22    QString str;
23    float f;
24    int value;
25    if (buffer.open(QIODevice::ReadOnly)) //将数据从buffer缓冲区读出来
26    {
27        QDataStream in(&buffer);
28        in>>str;
29        in>>f;
30        in>>value;
31        buffer.close();
32    }
33
34    qDebug()<<str;
35    qDebug()<<f;
36    qDebug()<<value;
37    return a.exec();
38 }
```

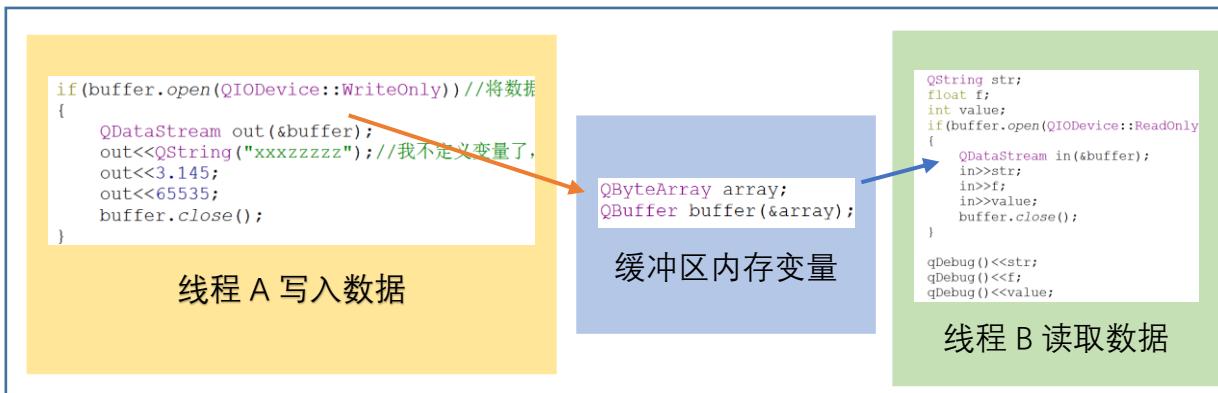
1.申请缓冲区对象  
2.关联  
3.打开缓冲区对象  
4.用数据流将字节数据写入 buffer 里面的 array 内存  
将 array 缓冲区内存的数据读出来

xxxxzzzz  
3.145  
65535

## 1. 缓冲区一般用于线程之间通信



这是传统的用文件来转换两个线程数据传递的方法



我这种就是不用文件，就用一个 buffer 缓冲区变量来传递线程之间的数据，程序执行完缓冲区就消失了，不用在硬盘上残留文件

## 2. 缓冲区可以用来读取硬件设备的数据



## 3. 缓冲区可以处理 A 进程写入数据太快，B 进程读取数据太慢的文本

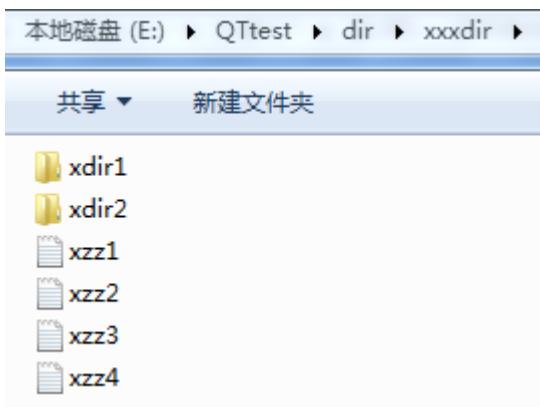
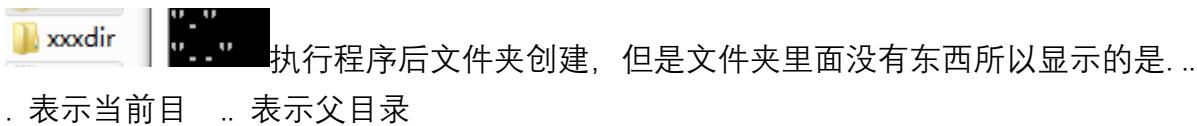
### QT 文件夹创建, 删除, 重命名操作

```
#include <QDir> //文件夹操作头文件
#include <QDebug>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    const char* PATH = "E:/QTtest/dir/xxxxdir"; //指定创建目录的路径
    QDir dir;

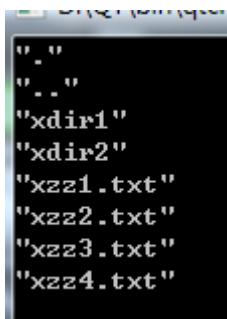
    //判断指定路径的文件/文件夹是否存在, 存在exists返回true, 不存在返回false
    if(!dir.exists(PATH))
    {
        dir.mkdir(PATH); //放入指定路径创建目录
    }

    if(dir.exists())
    {
        dir.cd(PATH); //进入已经创建的目录
        QStringList list = dir.entryList();
        //entryList返回当前目录下所有的目录或者文件夹列表给list
        for(int i = 0;i < list.count();i++) //list数组里面每一个元素就是一个文件/文件夹
        {
            qDebug()<<list[i];
        }
    }
    return a.exec();
}
```



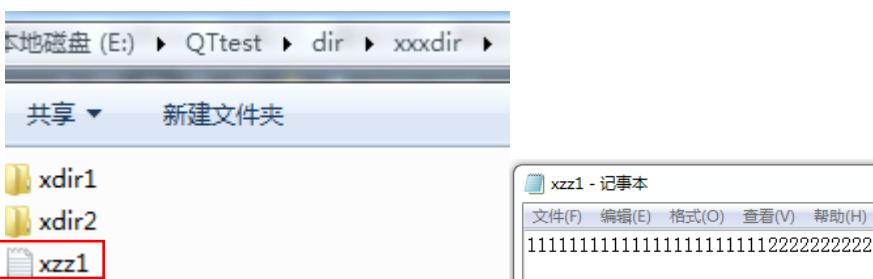
```
if(dir.exists())
{
    dir.cd(PATH); //进入已经创建的目录
    QStringList list = dir.entryList();
    //entryList返回当前目录下所有的目录或者文件夹列表给list
    for(int i = 0;i < list.count();i++) //list数组里面每一个元素就是一个文件/文件夹
    {
        qDebug()<<list[i];
    }
}
return a.exec();
```

就靠这段代码来获取目录信息



这就是拷贝到目录下的信息，一模一样

计算指定文件 txt 的大小



我给 xzz1 文本写了些内容

打开方式:  记事本  更改(C)...

位置: E:\QTtest\dir\xxxdir

大小: 88 字节 (88 字节)

window 给出的结果是 88 字节

```

#include <QDebug>
#include <QFileInfo>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    int ret = 0;
    QString path = "E:/QTtest/dir/xxxdir/xzz1.txt";
    QFileInfo info(path);

    if(info.isFile())//isFile是判断路径下的指定文件在不在
    {
        ret = info.size();//计算文件大小
    }
    else
    {
        qDebug()<<"file size failed";
    }
    qDebug()<<ret;

```



QFileInfo::exists 不管是目录还是文件都判断为 true, QFileInfo::isFile 只判断文件, 不判断目录

### 用于监控指定文件，目录内容变化的类 QFileSystemWatcher

QFileSystemWatcher 有两个信号, directoryChanged 是监控目录变化, fileChanged 是监控文件变化

在头文件声明信号接受函数和文件监控对象

```

#include <QObject>
#include <QFileSystemWatcher>

class wacher:QObject
{
    Q_OBJECT

private slots:
    void statuscao(const QString& path); //槽函数

public:
    QFileSystemWatcher watcher;
};


```

我们定义个信号接受槽函数。目录或者文件发生变化，就执行这个函数。这里的形参是根据信号来的

|                                                       |                             |
|-------------------------------------------------------|-----------------------------|
| void <u>directoryChanged</u> ( const QString & path ) | 发送信号的函数类 QT                 |
| void <u>fileChanged</u> ( const QString & path )      | 库已经实现好了，所以要根据信号的形参来设置槽函数的形参 |

目录/文件监控的发信号函数在这个对象里面

```

1 #include <QtCore/QCoreApplication>
2 #include "wacher.h"
3 #include <QDebug>
4
5 //监控文件状态的变化
6 void wacher::statuscao(const QString& path)
7 {
8     qDebug() << "file status change";
9 }
10
11 int main(int argc, char *argv[])
12 {
13     QCoreApplication a(argc, argv);
14     wacher xwacher;
15     QObject::connect(&xwacher.watcher, SIGNAL(fileChanged(const QString&)), &xwacher, SLOT(statuscao(const QString&)));
16     //先实现文件的监控
17     xwacher.watcher.addPath("E:/QTtest/filecheck/fille/xzz.txt");//监控文件变化
18     return a.exec();
19 }
20
21

```

信号槽链接指定信号发送的对象

文件被修改了，文件状态变化了执行槽函数

QObject is an inaccessible base of “...”这种错误是你使用了 QObject 里面的函数，但是你的 QObject 并不是 public 属性

```

class wacher:public QObject
{
Q_OBJECT

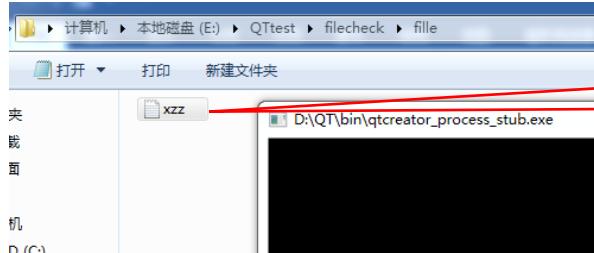
private slots:
    void statuscao(const QString& path); //槽函数

public:
    QFileSystemWatcher watcher;
};

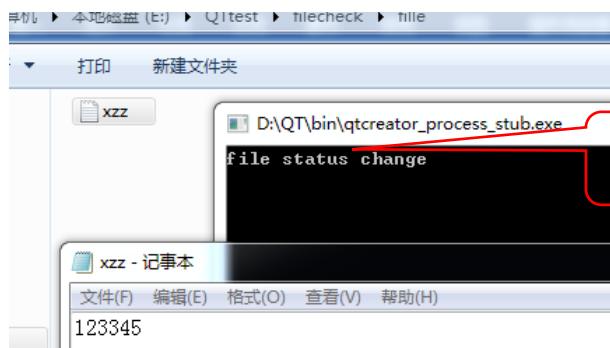
#endif // WACHER_H

```

头文件加上 public 问题解决



文件没有修改槽函数没有执行



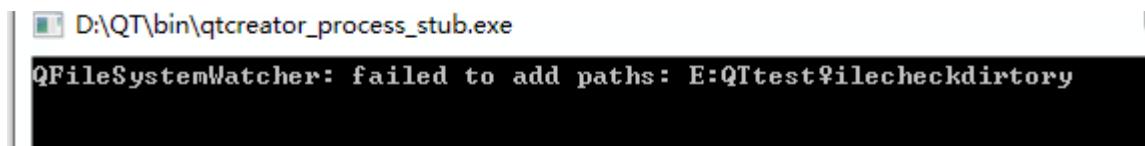
文件修改了槽函数执行了

下面我们取监控目录里面文件数量有没有变化



```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    wacher xwacher;
    QObject::connect(&xwacher.watcher, SIGNAL(fileChanged(const QString&));
    //先实现文件的监控
    xwacher.watcher.addPath("E:\QTtest\filecheck\dirtory"); //监控文件变化
    return a.exec();
}
```

我们监控指定路径的目录



程序执行就报错，为什么找不到目录呢？其实路径没有错，是监控目录的信号函数没有，我们说了 QFileSystemWatcher 有两个信号，一个监控文件，另一个监控目录

```
"include <QFileSystemWatcher>

class wacher:public QObject
{
Q_OBJECT

private slots:
    void statuscao(const QString& path); //文件槽函数
    void dirtorycao(const QString& path); //目录槽函数

public:
    QFileSystemWatcher watcher;
};

//监控文件状态的变化
void wacher::statuscao(const QString& path)
{
    qDebug()<<"file status change";
}
//监控目录变化的槽函数
void wacher::dirtorycao(const QString& path)
{
    qDebug()<<"dirtory status change";
}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    wacher xwacher;

    QObject::connect(&xwacher.watcher, SIGNAL(fileChanged(const QString&)), &xwacher, SLOT(statuscao(const QString&)));
    //将wacher类发送的信号，让对应的槽函数响应

    QObject::connect(&xwacher.watcher, SIGNAL(directoryChanged(const QString&)), &xwacher, SLOT(dirtorycao(const QString&)));

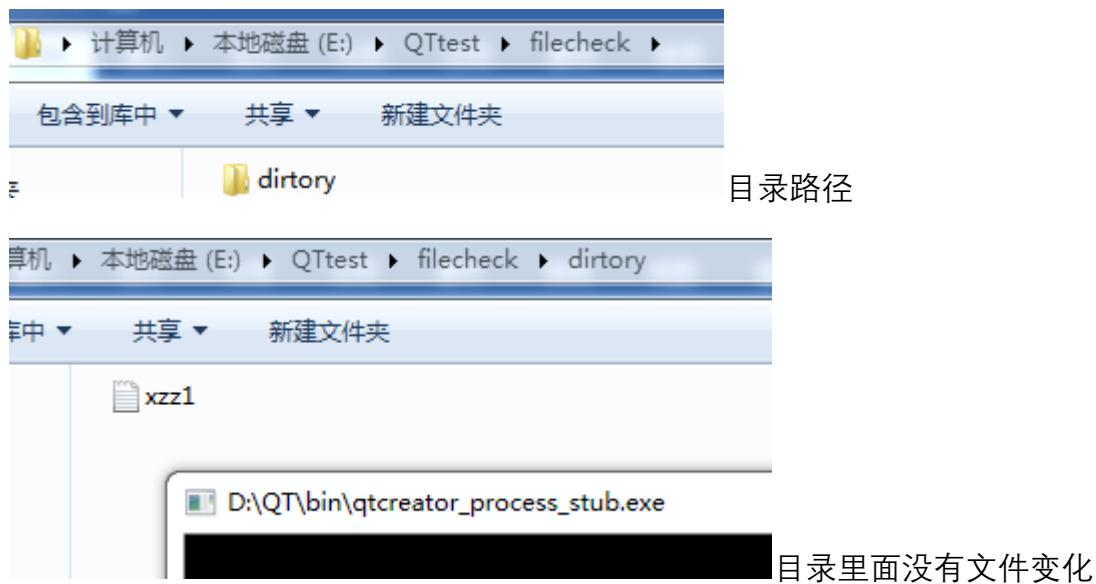
    xwacher.watcher.addPath("E:/QTtest/filecheck/fille/xzz.txt"); //监控指定路径文件变化
    xwacher.watcher.addPath("E:/QTtest/filecheck/dirtory"); //监控指定路径目录变化

    return a.exec();
}
```

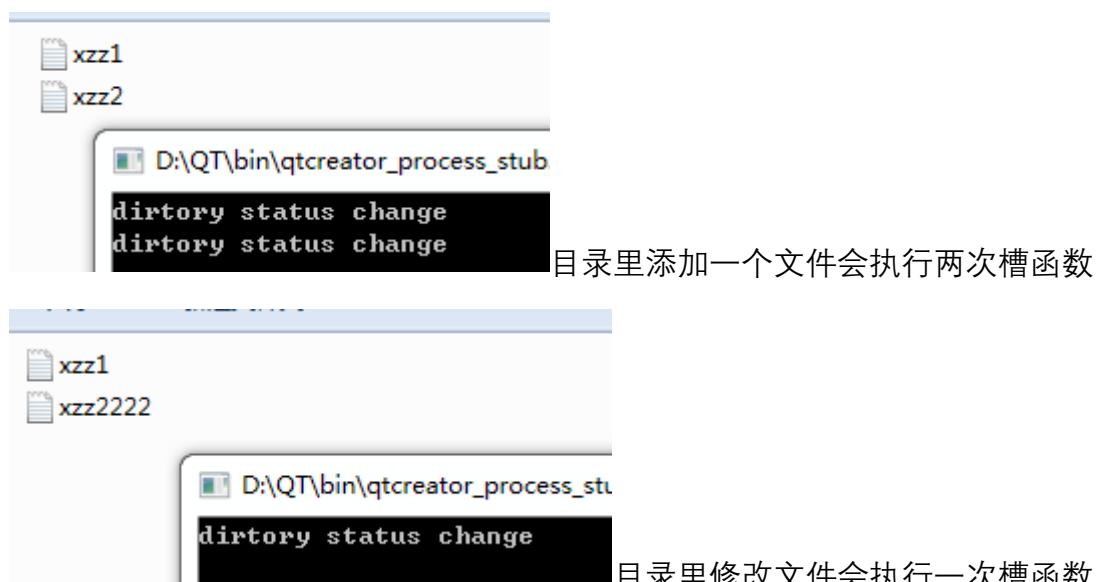
头文件增加个目录监控槽函数

目录里面文件数量/名字发生变化执行目录槽函数

使用目录监控信号函数链接槽



目录里面没有文件变化



目录里添加一个文件会执行两次槽函数



目录里修改文件会执行一次槽函数



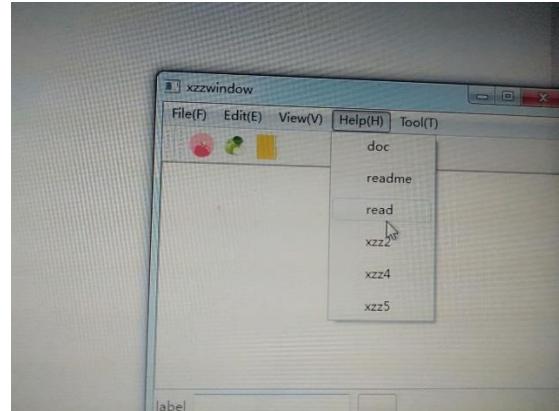
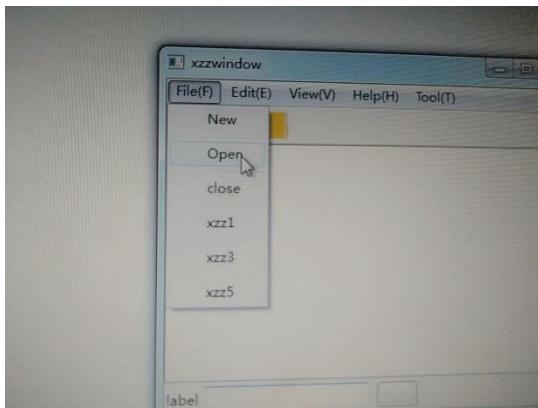
目录里删除一次文件会执行一次槽函数

目录里，文件自己里面内容修改，目录监控没有反应，只有用文件监控来解决

## 用界面窗口读取文本数据显示。写好的文本用界面进行存储

在上面菜单栏，工具栏，状态栏，多行文本综合开发项目代码下加入界面文本事件响应

先实现菜单按钮，按下之后响应函数



connect (**action**,

我们在创建下拉菜单按钮时，每一个下拉按钮都有对应的 action 对象，我们把这个 action 对象放入 connect 函数的信号位置

SIGNAL(**triggered()**) ,

**this**,

我们就要 QAction 自带的信号触发函数

SLOT(**slot\_function()**) );

在头文件实现一个槽函数

```
bool CreateStatus(QMainWindow * w, QToolBar * toolbar, QToolBar * toolBar);  
bool initstatus(); //创建状态栏  
bool initxzzEdit(); //创建多行文本  
private slots:  
    void onfileopen(void); //在这里增加一个槽函数  
};
```

单独建立一个 cpp 文件来处理下拉按钮信号事件的槽函数

```
#include "xzzwindow.h"  
#include <QDebug>  
  
void xzzwindow::onfileopen(void){  
    qDebug() << "onfileopen";  
}
```

在我以前创建界面的文件加入下拉菜单与槽函数链接机制

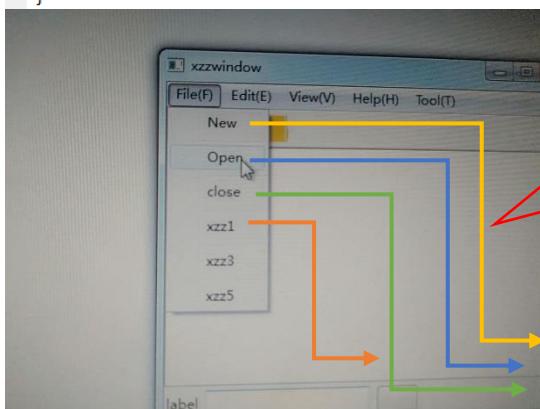
```

bool xzzwindow::creatcaidanpro(QMenu* menu,QString actstr)
{
    bool ret = true;
    int num = Stringchargegetnum(actstr,',');//获取字符串逗号个数
    QString strbuff[num+1];
    num = xzzStringcut(actstr,strbuff,',');
    for(int i=0;i<num+1;i++)
    {
        QAction *action = new QAction(strbuff[i],NULL);
        if(action == NULL)
            ret = false;
        else
            menu->addAction(action);

        menu->addSeparator();
        connect(action,SIGNAL(triggered()),this,SLOT(onfileopen()));
        //每创建一个下拉按钮，就将按钮对象的信号写入同一个槽函数
    }
    return ret;
}

```

就是为了验证按钮能触发槽函数



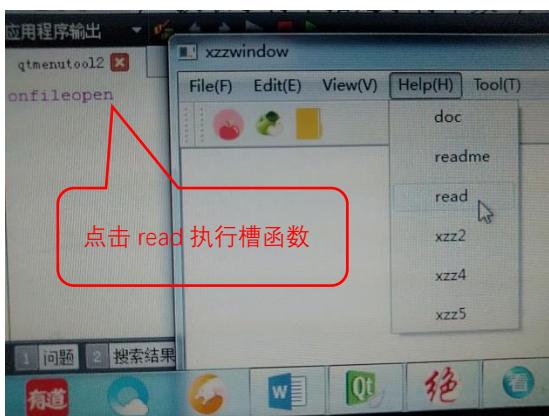
我现在点击下拉菜单任何按钮都执行一个槽函数

```

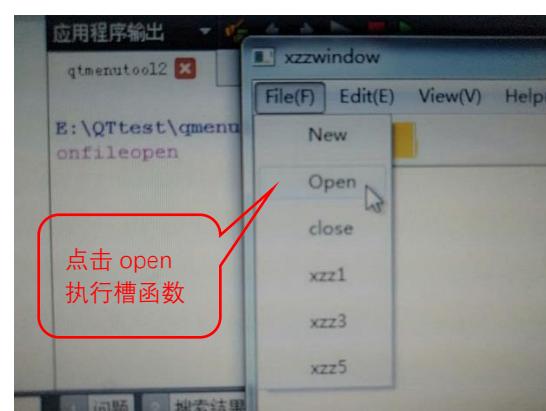
#include "xzzwindow.h"
#include <QDebug>

void xzzwindow::onfileopen(void)
{
    qDebug()<<"onfileopen";
}

```



点击 read 执行槽函数



点击 open 执行槽函数

我发现个问题



我发现点击工具栏，没有执行槽函数，  
这是因为工具栏的 action 没有加入进  
connect 信号与槽

```

130
131     bool xzzwindow::creattool(QToolBar* tb,QString str,QString icon)
132     {
133         bool ret = true;
134         QAction* action = new QAction("",NULL);
135         if(action == NULL)
136             ret = false;
137         else
138         {
139             action->setToolTip(str);
140             action->setIcon(QIcon(icon));
141             tb->addAction(action);
142             connect(action,SIGNAL(triggered()),this,SLOT(onfileopen()));
143             //为工具栏的action创建信号与槽链接函数
144         }
145         return ret;

```

应用程序输出 xzzwindow

E:\QTtest\qmenutool\qtm... onfileopen onfileopen onfileopen

g\qtmenutool.exe 启动中...

我们来实现指定路径文件内容读取操作

我们先看看 QDialog 类,窗口打开指定文件

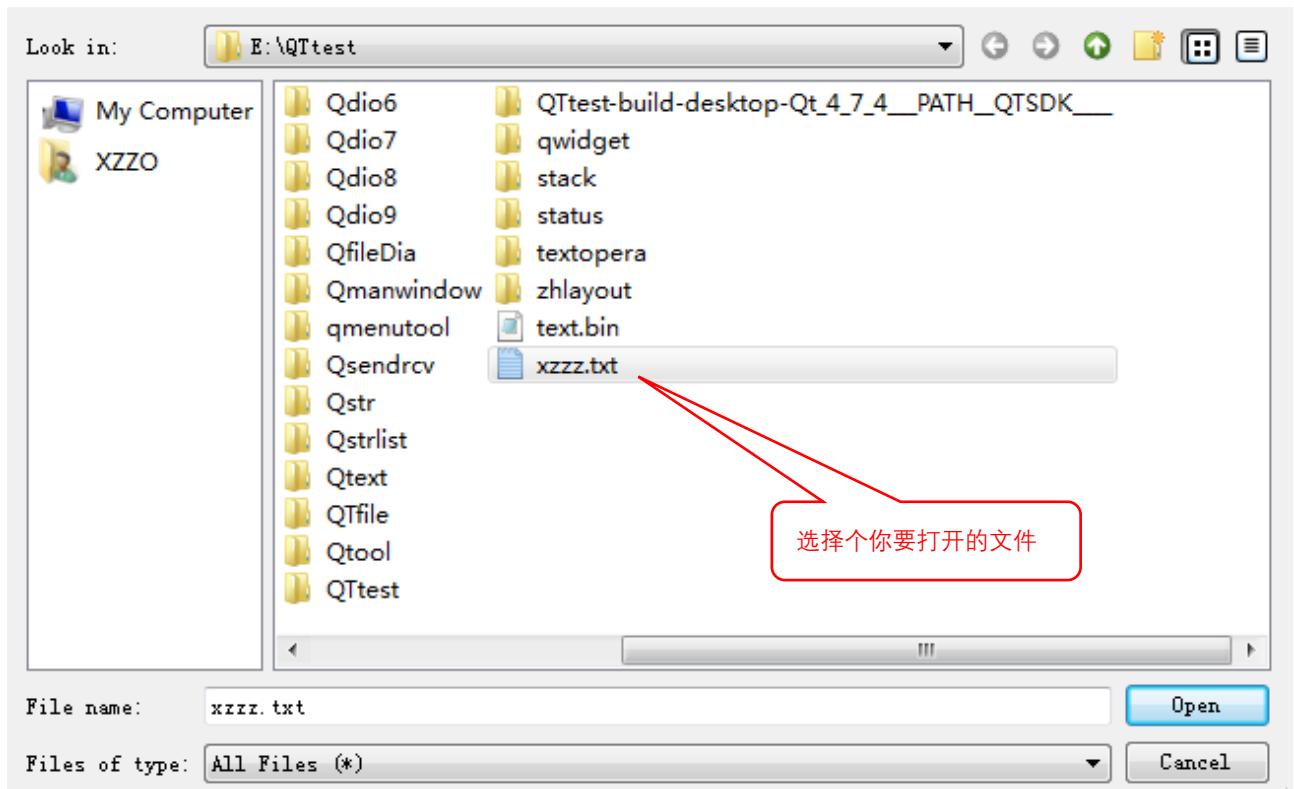
```

3 | #include "mainwindow.h"
4 | #include <QFileDialog> //窗口打开指定文件类
5 | #include <QDebug>
6 | #include <QString>
7 | int main(int argc, char *argv[])
8 | {
9 |     QApplication a(argc, argv);
10 |     QFileDialog fd;
11 |     fd.setAcceptMode(QFileDialog::AcceptOpen); //文件对话框为打开类型
12 |     fd.setFileMode(QFileDialog::ExistingFile); //存在的, 单个文件名
13 |     fd.setWindowTitle("标题");
14 |
15 |     if(fd.exec() == QFileDialog::Accepted)
16 |     {
17 |         QStringList fs = fd.selectedFiles();
18 |         //selectedFiles时候获取你打开后选择的文件名路径
19 |         qDebug() << fs;
20 |     }
21 |
22 |
23 |     return a.exec();
24 }

```

程序运行到这里就会弹出对话框, 阻塞程序

因为我们是用窗口去打开指定文本文件, 所以选择 AcceptOpen



```
if (fd.exec() == QFileDialog::Accepted)
{
    QStringList fs = fd.selectedFiles();
    //selectedFiles时候获取你打开后选择的文件名路径
    qDebug() << fs;
}
```

选择打开文件之后，跳过阻塞，执行下面的文件路径获取代码

```
E:\QTtest\QfileDia\QfileDia-build-de
{1 ?}
E:\QTtest\QfileDia\QfileDia-build-de
("E:/QTtest/xzzz.txt")
```

这就是文件名和路径，放入了 fs 变量，然后我们

用上面讲到的文件打开，read 读取方法就可以读取文件内容了  
我们接着讲文本编辑的数据获取

```

#include <QDebug>
#include <QFileDialog>

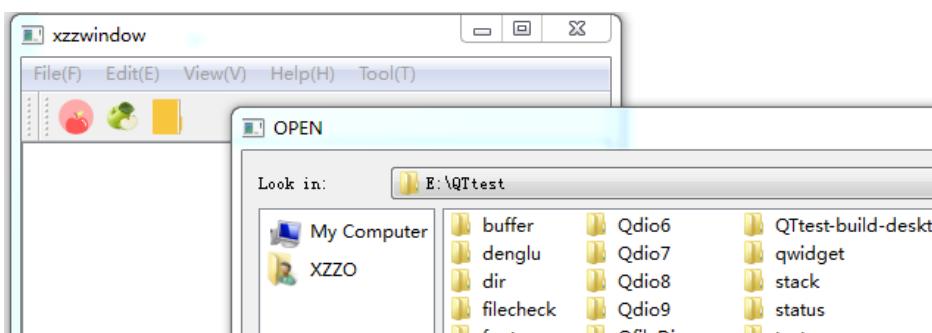
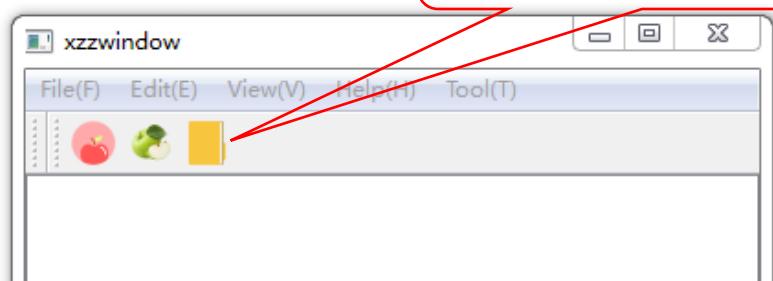
void xzzwindow::onfileopen(void) {
    qDebug () << "onfileopen";
    QFileDialog fd(this);

    fd.setWindowTitle ("OPEN");
    fd.setAcceptMode (QFileDialog::AcceptOpen);
    fd.setFileMode (QFileDialog::ExistingFile);

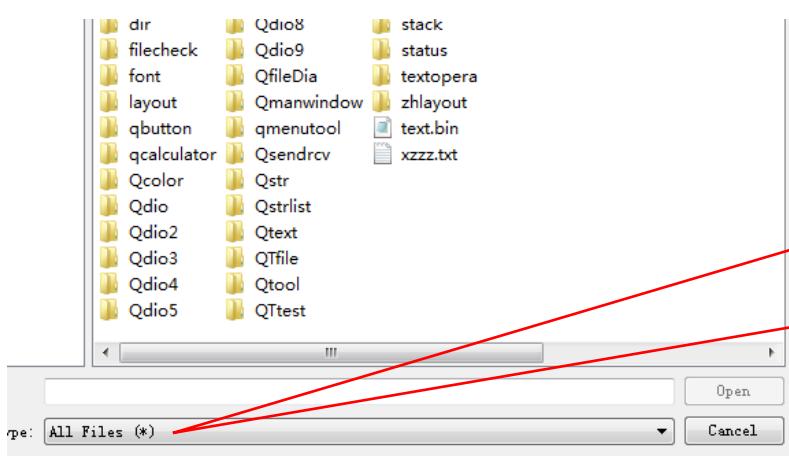
    if (fd.exec () == QFileDialog::Accepted)
    {
    }
}

```

当按下下拉菜单，或者状态栏按钮后，在槽函数用 QFileDialog 来实现窗口指定文件的操作



你看点击按钮，窗口指定文件程序就打开了



但是我们发现目录，  
文本文件，二进制文  
件都显示在窗口，当  
然这个问题可以不解决，  
但是我们希望有时能指  
定文本格式筛选显示，不要显示 ALL  
所有

```

1 #include "xzzwindow.h"
2 #include <QDebug>
3 #include <QFileDialog>
4
5 void xzzwindow::onfileopen(void){
6     qDebug() << "onfileopen";
7     QFileDialog fd(this);
8     QStringList filelist;
9
10    filelist.append("Text file (*.txt)");
11    filelist.append("All file (*)");
12
13    fd.setWindowTitle("OPEN");
14    fd.setAcceptMode(QFileDialog::AcceptOpen);
15    fd.setFileMode(QFileDialog::ExistingFile);
16    fd.setFilters(filelist); //设置过滤器，只显示ALL全部，或者指定文本格式
17
18    if(fd.exec() == QDialog::Accepted)
19    {
20    }
21
22 }
23
24

```

应用程序输出  
E:\QTtest\qmenutool\qmenutool4-build-desktop\qmenutool4>onfileopen

这里是指定要显示的文件格式，全部用字符串形式写入 setFilters

前面字符随便写，关键 是(\*)这个符号

你看我就可以不显示二进制文件 text.bin

```

void xzzwindow::onfileopen(void){
    qDebug() << "onfileopen";
    QFileDialog fd(this);
    QStringList filelist;

    filelist.append("Text file (*.txt)");
    filelist.append("All file (*)");

    fd.setWindowTitle("OPEN");
    fd.setAcceptMode(QFileDialog::AcceptOpen);
    fd.setFileMode(QFileDialog::ExistingFile);
    fd.setFilters(filelist);

    if(fd.exec() == QDialog::Accepted)
    {
        QStringList fs = fd.selectedFiles();
        QFile file(fs[0]);
        if(file.open(QIODevice::ReadOnly | QIODevice::Text))
        {
            QString str = QString(file.readAll());
            qDebug() << str;
            file.close();
        }
        else
        {
            qDebug() << "file read failed";
        }
    }
}

```

当你窗口选择文件后，跳过阻塞，我就靠 selectedFiles 来获取你选择的文件路径名

像我们上面操作文件那样，创建 QFile 类，来操作指定路径的文件

显示

将文件内容读出来

我们也可以将文本显示在中心编辑框

```

应用程序输出  
E:\QTtest\qmenutool\qmenutool\qmenutool5
E:\QTtest\qmenutool\qmenutool5
{1 ?}
E:\QTtest\qmenutool\qmenutool5
onfileopen
"1234567xxxxxxxxx"

```

读取文件内容

```

if(fd.exec() == QFileDialog::Accepted)
{
    QStringList fs = fd.selectedFiles();
    QFile file(fs[0]);

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

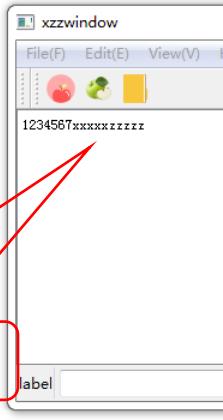
        ptedit.setPlainText(QString(file.readAll()));
        file.close();
    }
    else
    {
        qDebug() << "file read failed";
    }
}

if(fd.exec() == QFileDialog::Accepted)
{
    QStringList fs = fd.selectedFiles();
    QFile file(fs[0]);

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

        ptedit.setPlainText(QString(file.readAll()));
        file.close();
    }
    else
    {
        QMessageBox msg(this);
        msg.setWindowTitle("error");
        msg.setIcon(QMessageBox::Critical);
        msg.setStandardButtons(QMessageBox::Ok);
        msg.exec();
    }
}

```



上面 debug 输出的内容在这里显示了

我们将读取的文本文件内容显示在中心窗口编辑框内, ptedit 是定义的编辑框对象

如果打开文件失败我不想在 Qdebug 显示字符, 我就用弹出对话框的方式来提

## 文件编写后保存操作

### 修改界面槽函数的名字

```
bool xzzwindow::creatcaidanpro(QMenu* menu,QString actstr)
{
    bool ret = true;
    int num = StringchargegetNum(actstr,',');//获取字符串逗号个数
    QString strbuff[num+1];
    num = xzzStringcut(actstr,strbuff,',');
    for(int i=0;i<num+1;i++)
    {
        QAction *action = new QAction(strbuff[i],NULL);
        if(action == NULL)
            ret = false;
        else
            menu->addAction(action);

        menu->addSeparator();
        connect(action,SIGNAL(triggered()),this,SLOT(onfilesave()));
        //因为每个按钮都是执行一个槽函数，所以我就修改这个槽函数为存储功能
    }
    return ret;
}
```

```
#include <QMessageBox>
#include <QTextStream> //辅助类对象
void xzzwindow::onfilesave(void){
    qDebug()<<"onfileopen";
    QFileDialog fd(this);
    QStringList filelist;

    filelist.append("Text file (*.txt)");
    filelist.append("All file (*)");

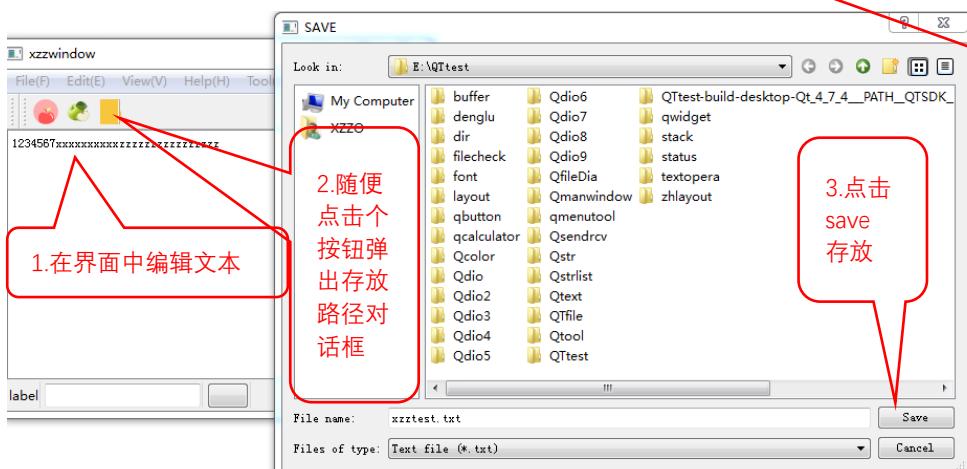
    fd.setWindowTitle("SAVE");
    fd.setAcceptMode(QFileDialog::AcceptSave); //以存储文件的方式打开文件对话框
    fd.setFilters(filelist);

    if(fd.exec() == QFileDialog::Accepted)
    {
        QStringList fs = fd.selectedFiles(); //获取指定的文件存放路径
        QFile file(fs[0]); //打开窗口得到的指定存放文件路径写入file

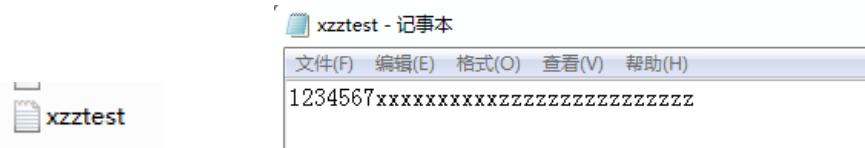
        if(file.open(QIODevice::WriteOnly | QIODevice::Text)) //数据写入文件WriteOnly，存放写入的东西是txt文本
        {

            QTextStream out(&file); //创建辅助类对象，至于辅助类对象有什么用，看我前面章节的内容
            out<<ptedit.toPlainText(); //你在中心编辑器写入的内容将传入到指定路径的文本
            file.close();
        }
    }
}
```

槽函数这里修改成窗口打开后指定为存储操作

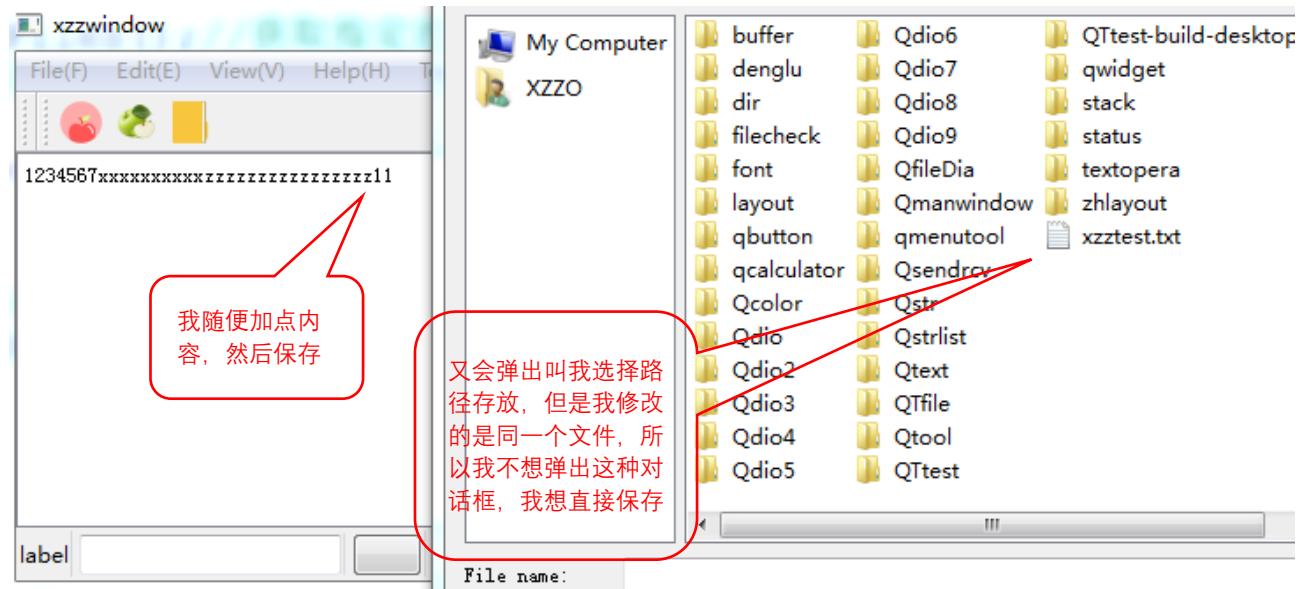


将内容写入文本缓冲区，最后要在窗口点击 Save 才能生成文本文档，这个文本文档就是接受 out 的数据



文件内容一致

但是现在有个问题



我的解决方法如下

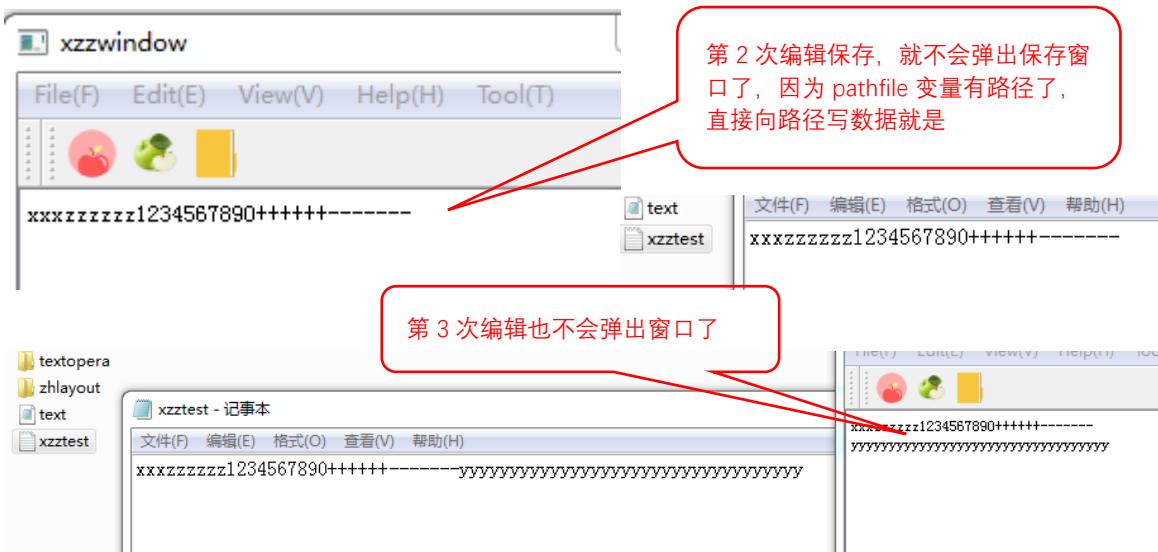
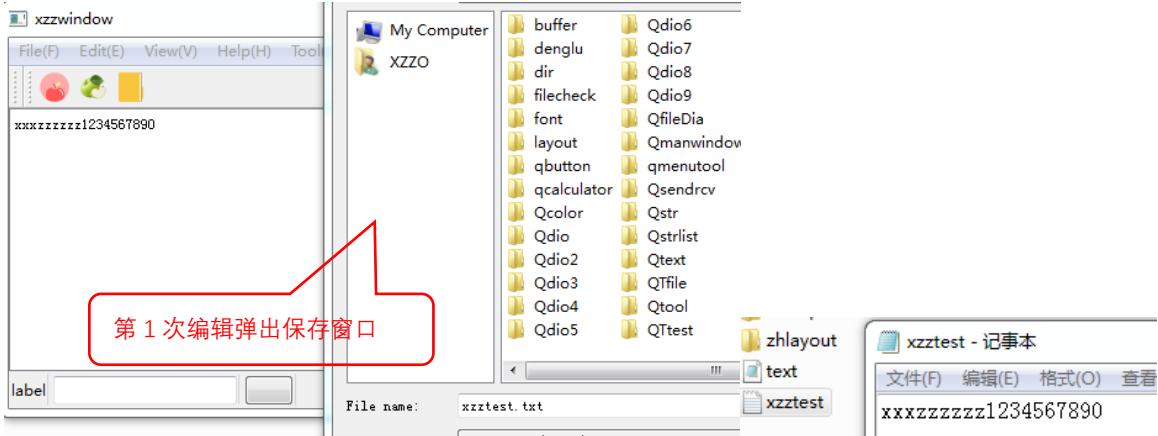
```
6 //include <QTextStream> //辅助类对象
7 QString filepath = "";//不管文件保存没有初始化文件路径为空
8 void xzzwindow::onFileSave(void){
9     qDebug()<<"onfileopen";
10    QFileDialog fd(this);
11    QStringList filelist;
12
13    filelist.append("Text file (*.txt)");
14    filelist.append("All file (*)");
15    fd.setFilters(filelist);
16
17    if(filepath == "") {
18        fd.setWindowTitle("SAVE");
19        fd.setAcceptMode(QFileDialog::AcceptSave);//以存储文件的方式打开文件对话框
20
21        if( fd.exec() == QFileDialog::Accepted )
22        {
23            filepath = fd.selectedFiles()[0];//获取文件路径
24        }
25    }
26
27    if(filepath != "") {
28        QFile file(filepath);//打开窗口得到的指定存放文件路径写入file
29
30        if(file.open(QIODevice::WriteOnly | QIODevice::Text))//数据写入文件WriteOnly，存放写入的东西是txt文本
31        {
32
33            QTextStream out(&file); //创建辅助类对象，至于辅助类对象有什么用，看我前面章节的内容
34            out<<ptedit.toPlainText(); //你在中心编辑器写入的内容将传入到指定路径的文本
35            file.close();
36        }
37    }
38 }
```

设置个全局变量，每次打开软件全局变量的路径为空

编辑文本按下按钮之后，发现第一次全局变量为空，弹出存储文件对话框

第二次按下按钮发现全局变量路径不为空，直接将第2次编辑的数据保存进文本

看看测试结果



所以这样就实现了固定路径文件自动保存，只有你关闭软件重新打开才会弹出路径窗口  
路径显示

```

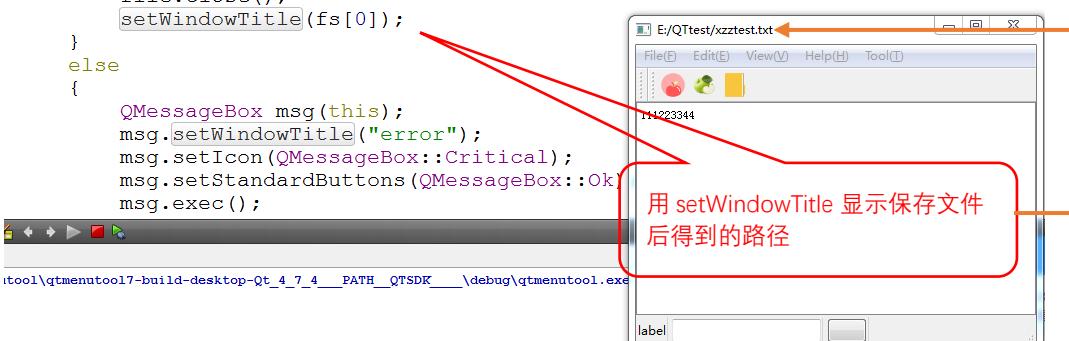
fd.setWindowTitle("Save As"); //另存为功能
fd.setAcceptMode(QFileDialog::AcceptSave); //以存储文件的方式打开文件对话框
fd.setFilters(filelist);

if(fd.exec() == QFileDialog::Accepted)
{
    QStringList fs = fd.selectedFiles(); //获取指定的文件存放路径
    QFile file(fs[0]); //打开窗口得到的指定存放文件路径写入file

    if(file.open(QIODevice::WriteOnly | QIODevice::Text)) //数据写入文件WriteOnly,
    {

        QTextStream out(&file); //创建辅助类对象，至于辅助类对象有什么用，看我前面章节的内容
        out<<pTextEdit.toPlainText(); //你在中心编辑器写入的内容将传入到指定路径的文本
        file.close();
        setWindowTitle(fs[0]);
    }
    else
    {
        QMessageBox msg(this);
        msg.setWindowTitle("error");
        msg.setIcon(QMessageBox::Critical);
        msg.setStandardButtons(QMessageBox::Ok);
        msg.exec();
    }
}

```



文本编辑器内容修改后状态提示 使用textChanged()

在头文件增加文本框内容变化触发的槽函数

```
QPLAINTEXTEDIT ptedit; //文本编辑框对象  
private slots:  
    void onfilesave(void); //在这里增加一个槽函数  
    void onTextchange(); //文本框内容变化的槽函数  
};
```

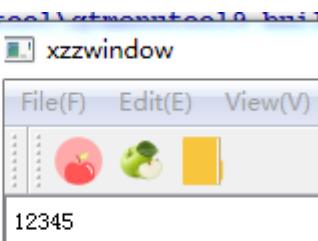
```
bool xzzwindow::initxzzEdit()  
{  
    bool ret = true;  
  
    if(&ptedit == NULL)  
        ret = false;  
    else  
    {  
        ptedit.setParent(this); //将多行文本嵌入进父窗口 this就是父对象  
        setCentralWidget(&ptedit); //将多行文本设置在父窗口中央  
        connect(&ptedit, SIGNAL(textChanged()), this, SLOT(onTextchange()));  
        //文本框对象里的内容发送变化发送信号textChanged()  
    }  
    return ret;  
}
```

在界面文件创建文本框对象的函数里增加信号函数

```
void xzzwindow::onTextchange()  
{  
    qDebug() << "file change";  
}
```

在槽函数文件里面增加文本框变化的槽函数

```
E:\QTtest\qmenu\mainwindow.h  
file change  
file change  
file change  
file change  
file change  
file change
```



你看我每写一个字符，槽函数就会执行一次

我们可以在槽函数里面随意加状态判断变量，来实现文本框是否修改，或者保存。

## QMap 使用，键值对概念

键值对 ("key = value") 字符串

每个键后面对应着相应的值，当按下相应的键时，就会输出相应结果

- **QMap 原型为 class QMap<K, T> 模板**  
K 是传入键的名称，QString 类型，T 是键名称对应的值，int 类型
- **QMap 中的键值对根据 Key 进行了排序**

```
1 #include <QtCore/QCoreApplication>
2 //存放键值对的头文件
3 #include<QDebug>
4 int main(int argc, char *argv[])
5 {
6     QApplication a(argc, argv);
7     QMap<QString,int> map; //定义一个QMap类模板对象
8
9     map.insert("key0",0); //向这个map对象写入键值，map会帮逆自动存放然后自动排序
0     map.insert("key1",1);
1     map.insert("key2",2);
2
3     QList<QString> Klist = map.keys();
4     //把map对象里面键值对的，键名读出来，键名是QString类型字符串
5
6     for(int i=0;i<Klist.count();i++) //count计算读出了多少个键名
7     {
8         qDebug()<<Klist[i]; //map存放的每个键名打印出来
9     }
0     return a.exec();
1 }
```



D:\QT\bin\qtcreator\_process  
"key0"  
"key1"  
"key2"

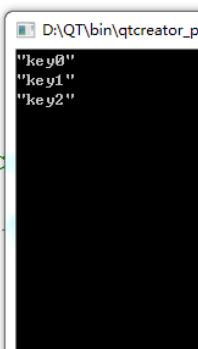
这个 key 的存放顺序到底是经过什么方法识别大小的。

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QMap<QString,int> map; //定义一个QMap类模板对象

    map.insert("key2",2); //我修改下键值存放顺序
    map.insert("key1",1);
    map.insert("key0",0);

    QList<QString> Klist = map.keys();

    for(int i=0;i<Klist.count();i++) //c
    {
        qDebug()<<Klist[i]; //map存放的每
    }
    return a.exec();
}
```



D:\QT\bin\qtcreator\_process  
"key0"  
"key1"  
"key2"

我把顺序翻过来了，还是能识别 key 大小

```

int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);
    QMap<QString, int> map; // 定义一个 QMap 类模板对象

    map.insert("kez", 2); // 我修改下键值存放顺序
    map.insert("kea", 1);
    map.insert("key", 0); // 原来 map 是根据字符串 ascii 码大小来排序的

    QList<QString> Klist = map.keys();

    for (int i=0; i<Klist.count(); i++) // 打印键名
    {
        qDebug() << Klist[i]; // map 存放的每个键名
    }
    return a.exec();
}

```

D:\QT\bin\qtcreator\_pro  
"kea"  
"key"  
"kez"

```

int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);
    QMap<QString, int> map; // 定义一个 QMap 类模板对象

    map.insert("key0", 0);
    map.insert("key1", 1);
    map.insert("key2", 2);

    QList<int> vlist = map.values(); // 获取 map 存放的键值对值的数据

    for (int i=0; i<vlist.count(); i++)
    {
        qDebug() << vlist[i]; // 打印值
    }
    return a.exec();
}

```

值的顺序是根据键名排序大小来对应的

D:\QT\bin\qtcreator\_process\_stub.exe  
0  
1  
2

## 迭代器

```
#include <QtCore/QCoreApplication>
#include<QMap>
#include<QDebug>
#include<QMapIterator>/迭代器
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QMap<QString,int> map; //定义一个 QMap类模板对象

    map.insert("key0",0);
    map.insert("key1",1);
    map.insert("key2",2);

    QMapIterator<QString,int> it(map);

    while(it.hasNext()) //判断it现在指向的map数组有没有下一个元素，现在的it指向的是第1个元素的上边NULL
    {
        it.next(); //it指针移动到map第1个元素
        qDebug()<<it.key()<<" :"<<it.value();
    }
    return a.exec();
}
```



## QHash 使用，哈希表使用

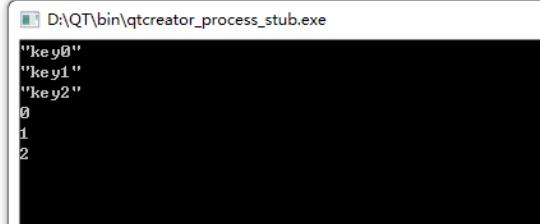
```
#include <QtCore/QCoreApplication>
#include<QHash>/哈希函数头文件
#include<QDebug>

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

    QHash<QString,int> hash; //定义个哈希表
    hash.insert("key0",0); //向表里面放数据和map一样的操作
    hash.insert("key1",1);
    hash.insert("key2",2);

    QList<QString> klist = hash.keys(); //取出哈希表里面存放的字符串
    for(int i=0; i<klist.count();i++)
    {
        qDebug()<<klist[i];
    }

    QList<int> vlist = hash.values(); //取出哈希表里面存放的值
    for(int i=0;i<vlist.count();i++)
    {
        qDebug()<<vlist[i];
    }
    return a.exec();
}
```



也可以用迭代器获取哈希表里面存放的数据

```

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    QHash<QString, int> hash;
    hash.insert("key0", 0);
    hash.insert("key1", 1);
    hash.insert("key2", 2);

    QList<QString> klist = hash.keys();
    for(int i=0; i<klist.count(); i++)
    {
        qDebug() << klist[i];
    }

    QList<int> vlist = hash.values();
    for(int i=0; i<vlist.count(); i++)
    {
        qDebug() << vlist[i];
    }

    QHash<QString, int>::const_iterator i;//定义哈希迭代器指针
    for(i=hash.begin(); i!=hash.constEnd(); ++i)
    {
        qDebug() << i.key() << ":" << i.value();
    }
    return a.exec();
}

```

```

D:\QT\bin\qtcreator_process
"key0"
"key1"
"key2"
0
1
2
"key0" : 0
"key1" : 1
"key2" : 2

```

```

QList<QString> klist = hash.keys();
for(int i=0; i<klist.count(); i++)
{
    qDebug() << klist[i];
}

QList<int> vlist = hash.values();
for(int i=0; i<vlist.count(); i++)
{
    qDebug() << vlist[i];
}

hash["key4"] = 4; //我这样也可以多添加键值对
QHash<QString, int>::const_iterator i;//定义哈希迭代器指针
for(i=hash.begin(); i!=hash.constEnd(); ++i)
{
    qDebug() << i.key() << ":" << i.value();
}

```

```

D:\QT\bin\qtcre
"key0"
"key1"
"key2"
0
1
2
"key4" : 4
"key5" : 5

```

可以在任何时候给哈希表添加值

我觉得 QMap 和 QHash 没啥区别啊，为什么要这么用呢

QHash 查找速度很快，只要写出要查找的 key 字符串，就能马上定位到 key 值  
但是 QMap 很占用内存空间

Qhash 可以随意在代码任何位置添加键值对

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QMap<QString, int> map; // 定义一个 QMap 类模板对象

    map.insert("key0", 0);
    map.insert("key1", 1);
    map.insert("key2", 2);

    QList<int> vlist = map.values(); // 获取 map 存放的键值对值的数据
    QList<QString> slist = map.keys(); // 获取 map 存放的字符串
    for (int i=0; i<vlist.count(); i++)
    {
        qDebug() << slist[i] << ":" << vlist[i]; // 打印值
    }
    map["key4"] = 4; // QMap 在任意位置添加 key 值对
    for (int i=0; i<vlist.count(); i++)
    {
        qDebug() << slist[i] << ":" << vlist[i]; // 打印值
    }
    return a.exec();
}
```

你看根据打印的结果  
Qmap 没有添加键值  
对成功

```
D:\QT\bin\qtcreator_process_stub.exe
"key0" : 0
"key1" : 1
"key2" : 2
"key0" : 0
"key1" : 1
"key2" : 2
```

```
QList<QString> klist = hash.keys();
for (int i=0; i<klist.count(); i++)
{
    qDebug() << klist[i];
}

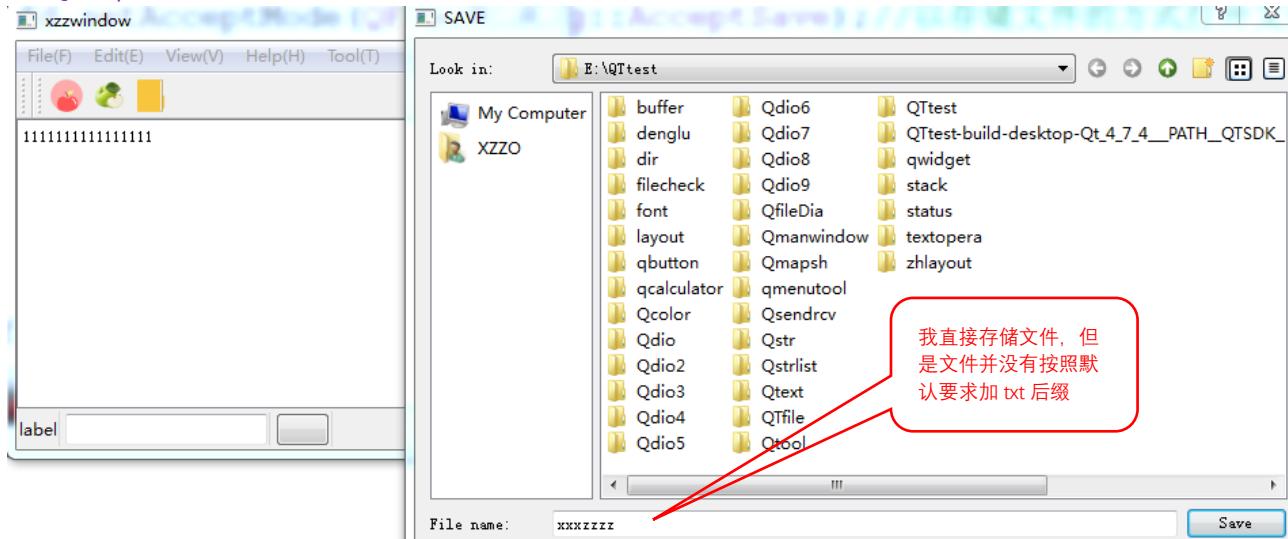
QList<int> vlist = hash.values();
for (int i=0; i<vlist.count(); i++)
{
    qDebug() << vlist[i];
}
hash["key4"] = 4; // 我这样也可以多添加键值对
QHash<QString, int>::const_iterator i; // 定义哈希迭代器
for (i=hash.begin(); i!=hash.constEnd(); ++i)
{
    qDebug() << i.key() << ":" << i.value();
}
```

你看根据打印的结果  
Qhash 就可以任意添  
加键值对

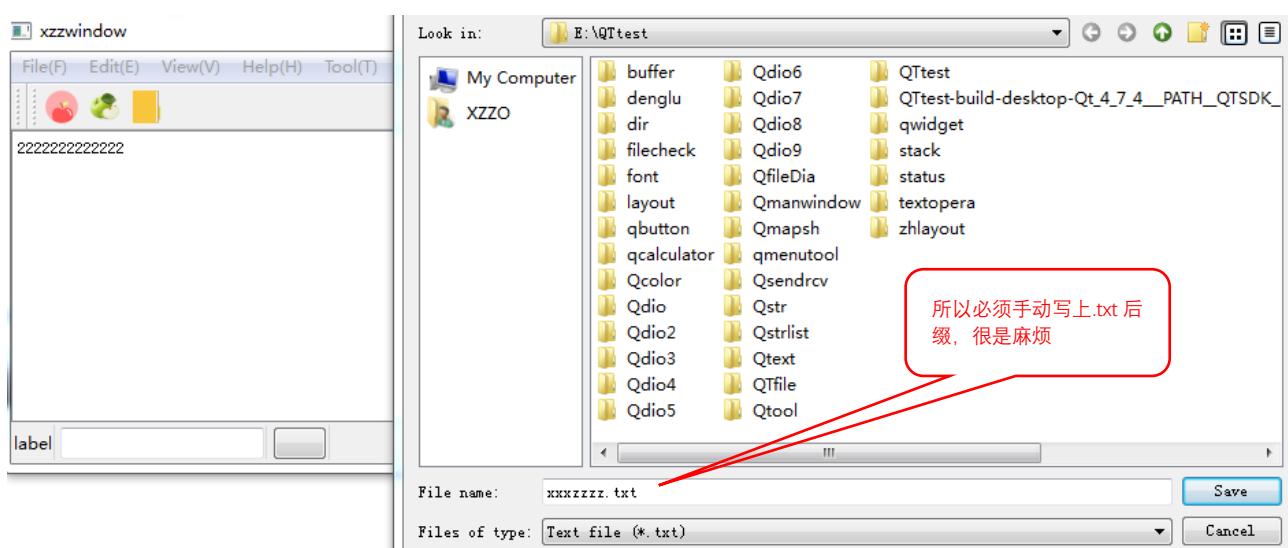
```
D:\QT\bin\qtcreator_process_stub.exe
"key0"
"key1"
"key2"
"key4"
0
1
2
"key0" : 0
"key1" : 1
"key2" : 2
"key4" : 4
```

Qmap 内存占用小，就这点优点完了，访问速度还慢。不能在代码任意位置添加键值对。  
QHash，访问速度快，可以在代码任意位置添加键值对。占用内存大而已。

## 用 QMap 键值对做一个案例



没有 txt 后缀



你看自己加的 txt 后缀，很麻烦

## 修改上面菜单栏，工具栏，状态栏，多行文本综合开发项目代码

```
QString filepath = "";//个首义什末仔沒有初始化什路径为空
void xzzwindow::onfilesave(void)
{
    qDebug()<<"onfileopen";
    QFileDialog fd(this);
    QStringList filelist;

    filelist.append("Text file (*.txt)");
    filelist.append("All file (*)");
    fd.setFilters(filelist);

    if(filepath == "")
    {
        fd.setWindowTitle("SAVE");
        fd.setAcceptMode(QFileDialog::AcceptSave);//以存储文件的方式打开文件对话

        if( fd.exec() == QFileDialog::Accepted )
        {
            filepath = fd.selectedFiles()[0];//获取文件路径
        }
    }
}
```

这两行代码改成存放二维数组

修改如下

```
void xzzwindow::onfilesave(void) {
    qDebug() << "onfileopen";
    QFileDialog fd(this);
    QStringList filelist;
    QMap<QString, QString> map; // 创建键值对对象

    const char* filterArray[] [2] =
    {
        {"Text Files (*.txt)", ".txt"}, // 二维数组指针，就是表示数组每行每列元素都是放字符串的，或者 char 类型的地址
        {"All Files (*)", "*"}, // 这里和 filelist.append("Text Files (*.txt)") 差不多，只是这里用的数组传递而已
        {NULL, NULL}
    };

    filelist.append(filterArray[0][0]); // 放入 "Text Files (*.txt)"
    map.insert(filterArray[0][0], filterArray[0][1]); // 将二维数组 txt 项放入键值对

    filelist.append(filterArray[1][0]); // 放入 "All Files (*)"
    map.insert(filterArray[0][0], filterArray[0][1]); // 将二维数组 All 项放入键值对

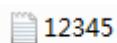
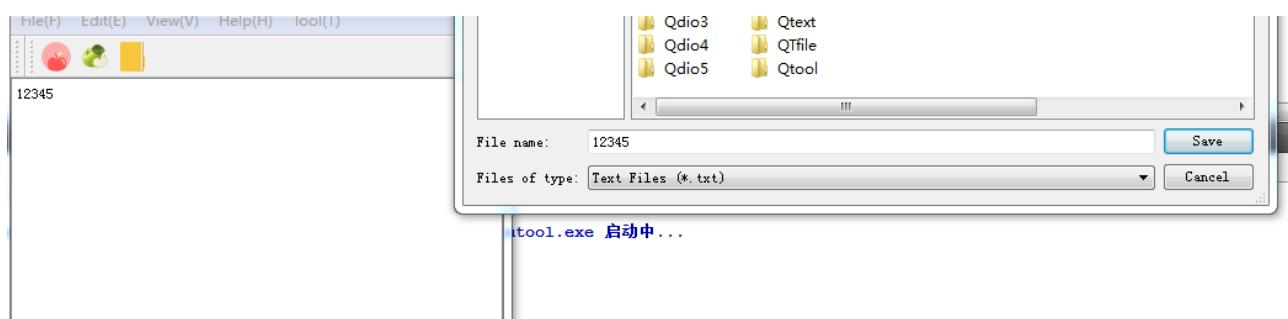
    fd.setFilters(filelist); // 主要是这个，这个是将 Text files..... 赋值了一个对应的 key 值，叫做.txt

    if(filepath == "") // filepath 得到的是 ("Text Files (*.txt)", "All Files (*)")
    {
        fd.setWindowTitle("SAVE");
        fd.setAcceptMode(QFileDialog::AcceptSave); // 以存储文件的方式打开文件对话框

        if(fd.exec() == QFileDialog::Accepted)
        {
            filepath = fd.selectedFiles()[0]; // 获取文件路径
            if(QFileDialog::AcceptSave == QFileDialog::AcceptSave)
            {
                QString postfix = map[fd.selectedFilter()];
                if(postfix != "*" && !filepath.endsWith(postfix))
                {
                    filepath = filepath + postfix;
                }
            }
        }
    }
}
```

Selectedfilter 的意思是设置当前选择的过滤器为文件 mask 中包含的第一个，所以就是 Text Files (\*.txt)，然后 map[Text Files (\*.txt)] 对应的 key 值是 .txt

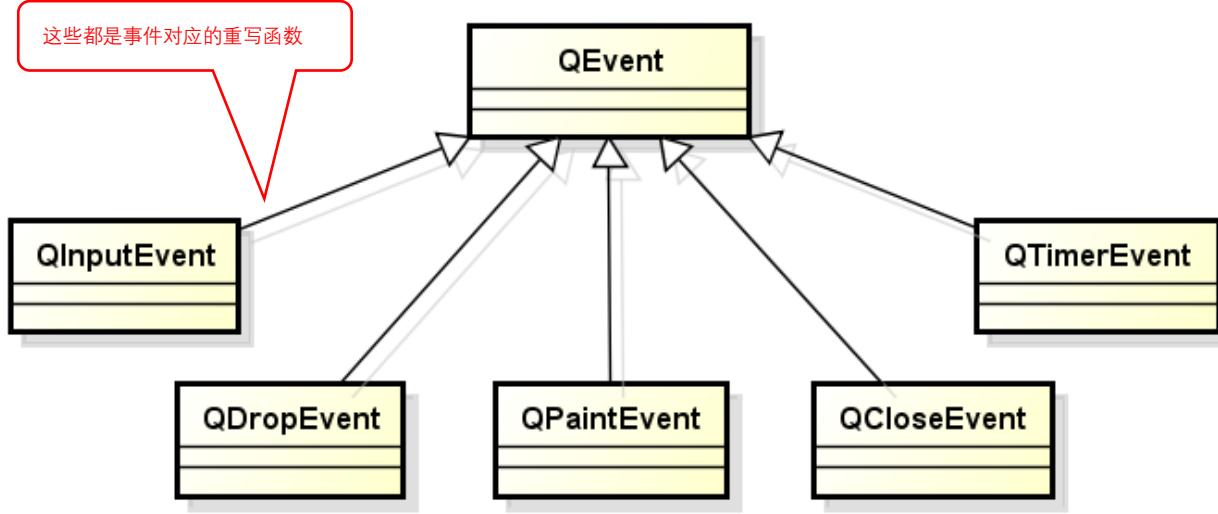
后面的判断自己分析了



你保存时不用加后缀默认都是存为 txt

## QT 事件类型，就是信号触发的事件，只是我们用对象本身的事件函数

- 任意的 **QObject** 对象都具备事件处理的能力



```
6  namespace Ui {  
7      class Widget;  
8  }  
9  
0  class Widget : public QWidget  
{  
1      Q_OBJECT  
2  
4  public:  
5      explicit Widget(QWidget *parent = 0);  
6      bool event(QEvent* e); //重写event处理函数  
7      ~Widget();  
8  
9  private:  
0      Ui::Widget *ui;  
1};
```

我们定义个重写函数，为什么是重写函数？是因为 event 函数不需要你手写调用，系统本来就有这个函数，只是你重写这个函数让系统实现自己想要的功能

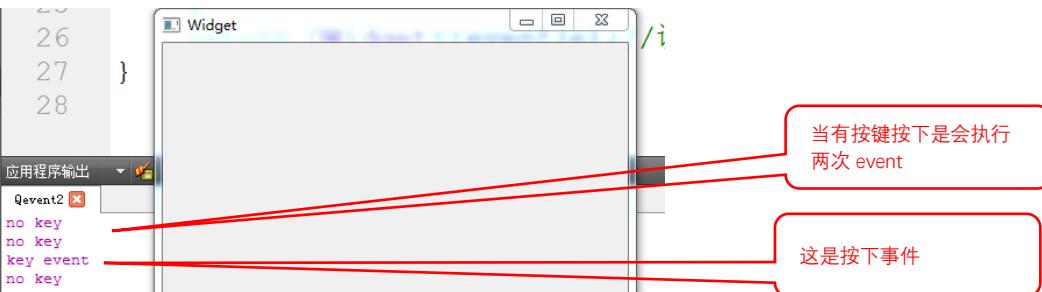
```
#include<QDebug>  
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}  
  
bool Widget::event(QEvent* e)  
{  
    qDebug() << "Widget::event";  
    return QWidget::event(e); //调用父类的事件处理函数  
}
```

既然系统会自动调用这个函数，那么在函数结束时，要给父类系统上报，我是自己实现了这个重写函数，要不然系统还会再去执行 QT 默认的 event 函数

我启动程序 event 就被执行了很多次，证明这个函数是启动时被系统多次执行的

```

1 bool Widget::event(QEvent* e)
2 {
3     if(e->type() == QEvent::KeyPress) //判断当前有没有按键按下，按下就执行
4     {
5         qDebug() << "key event";
6     }
7     else
8     {
9         qDebug() << "no key";
10    }
11   return QWidget::event(e); //调用父类的事件处理函数
12 }
```



```

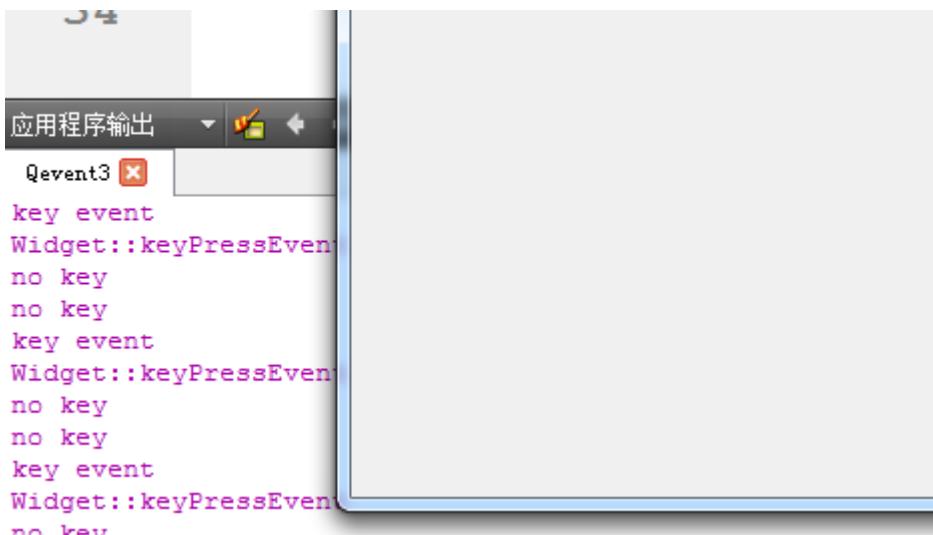
0 class Widget : public QWidget
1 {
2     Q_OBJECT
3
4 public:
5     explicit Widget(QWidget *parent = 0);
6     bool event(QEvent* e); //重写event处理函数
7     void keyPressEvent(QKeyEvent* e); //重写keyPressEvent函数
8     ~Widget();
9
10 private:
11     Ui::Widget *ui;
12 };
```

我头文件再定义一个 keyPressEvent 重写函数，这个函数是系统专门响应键盘按键事件的

```

16 bool Widget::event(QEvent* e)
17 {
18     if(e->type() == QEvent::KeyPress)
19     {
20         qDebug() << "key event";
21     }
22     else
23     {
24         qDebug() << "no key";
25     }
26     return QWidget::event(e); //调用父类的事件处理函数
27 }
28
29 void Widget::keyPressEvent(QKeyEvent* e)
30 {
31     qDebug() << "Widget::keyPressEvent";
32     QWidget::keyPressEvent(e); //告诉父类我这里执行了keyPressEvent函数
33 }
34

```



知所以系统能重写 event, keypressEvent 函数, 是因为我们的类是 QWidgetL 类, 这个类继承至 QObject 对象, 所以 QWidget 类里面系统默认都定义了 event, keypressEvent 这些函数。

## 拖放事件案例

### ■ 常用 MIME 类型数据处理函数

| 测试函数       | 获取函数        | 设置函数           | MIME类型              |
|------------|-------------|----------------|---------------------|
| hasText()  | text()      | setText()      | text/plain          |
| hasHtml()  | html()      | setHtml()      | text/html           |
| hasUrls()  | urls()      | setUrls()      | text/url-list       |
| hasImage() | imageData() | setImageData() | image/*             |
| hasColor   | colorData() | setColorData   | application/x-color |

### ■ 自定义拖放事件的步骤

1. 对接收拖放事件的对象调用 `setAcceptDrops` 成员函数
2. 重写 `dragEnterEvent` 函数并判断 MIME 类型
  - 期望数据 : `e->acceptProposedAction();`
  - 其它数据 : `e->ignore();`
3. 重写 `dropEvent` 函数并判断 MIME 类型
  - 期望数据 : 从事件对象中获取 MIME 数据并处理
  - 其它数据 : `e->ignore();`

在头文件先定义拖上窗口的重写函数和放入窗口的重写函数

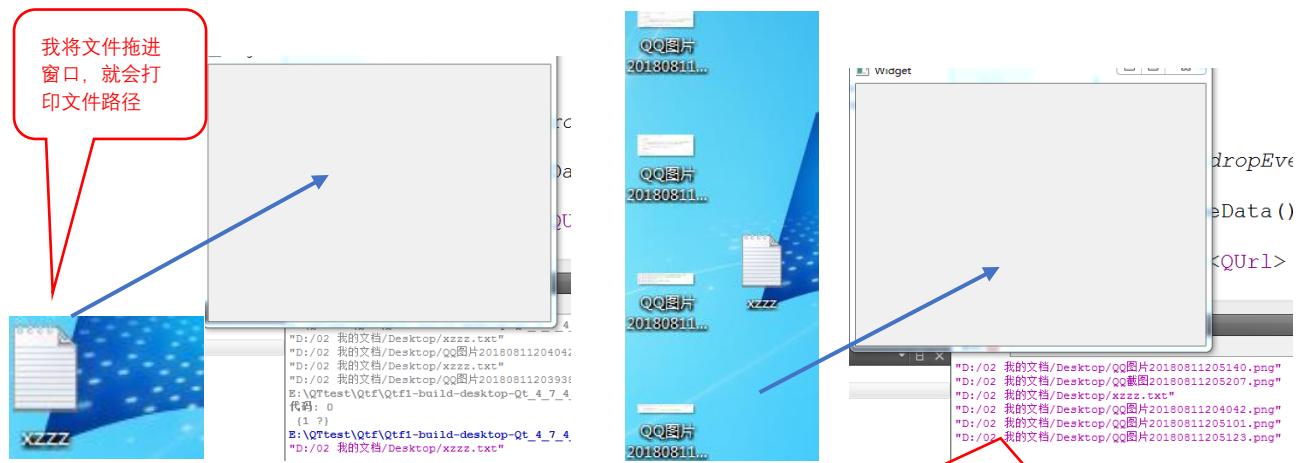
```
9
10 class Widget : public QWidget
11 {
12     Q_OBJECT
13 protected:
14     void dragEnterEvent(QDragEnterEvent * e); //重写拖的事件函数
15     void dropEvent(QDropEvent * e); //重写放的事件函数
16 public:
17     explicit Widget(QWidget *parent = 0);
18     ~Widget();
```

这些函数窗口建立的时候默认本身就有，我们主要是让它实现我们想要的功能。

```

1 //include "ui_widget.h"
2 #include<QDragEnterEvent> //拖在窗口的API函数
3 #include<QDropEvent> //放进窗口的API函数
4 #include<QList>
5 #include<QUrl> //获取文件路径
6 #include<QDebug>
7
8
9 Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
10 {
11     ui->setupUi(this);
12     setAcceptDrops(true); //当前的widget主窗口接受拖放事件
13 }
14
15 void Widget::dragEnterEvent(QDragEnterEvent * e)
16 {
17     if(e->mimeData()->hasUrls()) //hasUrls是判断拖放的文件是不是有路径，这种东西可以拖放任何文件
18     {
19         //因为windows有什么文件是没有路径的呢？
20
21         e->acceptProposedAction(); //如果对这个事件调用acceptProposedAction(),
22             //就表明用户可以在这个窗口部件上拖放对象
23     }
24     else
25     {
26         e->ignore();
27     }
28 }
29
30 void Widget::dropEvent(QDropEvent * e)
31 {
32     if(e->mimeData()->hasUrls()) //hasUrls是判断放入窗口的文件是不是有路径，这种东西可以放任何文件
33     {
34         QList<QUrl> list = e->mimeData()->urls(); //获取URL列表
35             //如果同时拖放多个文件给窗口就会排列每个文件的路径列表
36         for(int i=0;i<list.count();i++) //如果有多个文件被同时拖放进窗口，就要循环获取多个文件路径
37         {
38             qDebug()<<list[i].toLocalFile(); //获取文件的路径打印出来
39         }
40     }
41     else
42     {
43         e->ignore();
44     }
45 }

```



窗口支持的拖放功能已经实现

## 编辑器窗口点击按钮实现复制，粘贴，撤回

在上面菜单栏，工具栏，状态栏，多行文本综合开发项目代码下加入复制粘贴撤回功能

### Public Slots

```
void appendHtml ( const QString & html )
void appendPlainText ( const QString & text )
void centerCursor ()
void clear ()
void copy ()
void cut ()
void insertPlainText ( const QString & text )
void paste ()
void redo ()
void selectAll ()
void setPlainText ( const QString & text )
void undo ()
```

QT 中已经实现好了这些复制，粘贴，撤回，杂七杂八的槽函数，你只需要在按钮函数里面的 connect SLOT 填入对应的函数，按钮就触发了复制粘贴撤回的功能

```
131 bool xzzwindow::creattool(QToolBar* tb,QString str,QString icon)
132 {
133     bool ret = true;
134     QAction* action = new QAction("",NULL);
135     if(action == NULL)
136         ret = false;
137     else
138     {
139         action->setToolTip(str);
140         action->setIcon(QIcon(icon));
141         tb->addAction(action);
142         //connect(action,SIGNAL(triggered()),this,SLOT(onfilesave())); //工具栏按钮的保存功能取消
143         connect(action,SIGNAL(triggered()),&pTextEdit,SLOT(undo()));
144         //我们将工具栏按钮触发的事件链接到undo槽函数，这样撤回功能就是实现了。
145         //但是这个撤回功能是针对谁呢？，其实是针对文本编辑框，所以槽函数对应的文本编辑框对象是pTextEdit
146     }
147     return ret;
148 }
```

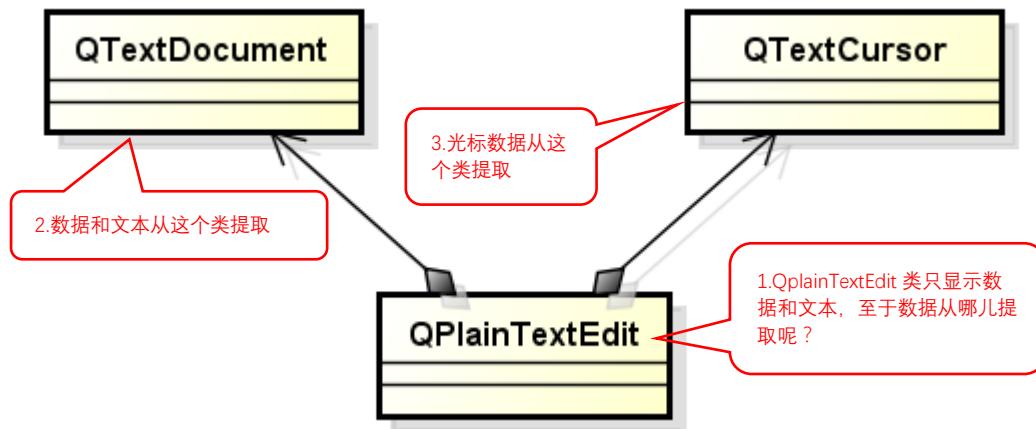
虽然按钮触发了复制粘贴撤回功能，但是谁来响应呢？让我们看到效果呢。这里就要填入文本编辑框对象，也可以是其它对象

```
connect(action,SIGNAL(triggered()),&pTextEdit,SLOT(redo()));
//如果需要撤回状态前的功能，用 redo 槽函数
connect(action,SIGNAL(triggered()),&pTextEdit,SLOT(cut()));
//剪贴功能用 cut
connect(action,SIGNAL(triggered()),&pTextEdit,SLOT(copy()));
//复制功能用 copy
connect(action,SIGNAL(triggered()),&pTextEdit,SLOT(paste()));
//粘贴功能 paste
```

## 文本编辑器编辑区域的打印功能实现

这一章和(打印对话框，让你的软件有使用打印机的功能，或者直接输出 pdf 功能)不一样  
打印对话框，让你的软件有使用打印机的功能，或者直接输出 pdf 功能是设置打印的内容  
这一章是直接选定编辑区，打印编辑区的内容

文本打印需要用到 `QplainTextEdit` 类



- **QTextDocument** 是表示文本以及文本属性的数据类
  - 设置文本的属性：排版，字体，标题，等
  - 获取文本参数：行数，文本宽度，文本信息，等
  - 实现标准操作：撤销，重做，查找，**打印**，等
  - . . .

(打印对话框，让你的软件有使用打印机的功能，或者直接输出 pdf 功能)章节就是还有了 `QtextDocument` 类

我们还是用上面章节制作的文本编辑器项目

先在头文件增加打印槽函数

```
QPLAINTEXTEDIT(ptedit); //文本编辑框对象
private slots:
    void onfilesave(void); //在这里增加一个槽函数
    void onTextchange(); //文本框内容变化的槽函数
    void onfileprint(); //定义一个打印功能的槽函数
};
```

```
#endif // XZZWTNDOW_H
```

建立一个打印槽函数

## 在界面 CPP 文件将工具栏图标按钮变成打印功能

```
bool xzzwindow::createtool(QToolBar* tb,QString str,QString icon)
{
    bool ret = true;
    QAction* action = new QAction("",NULL);
    if(action == NULL)
        ret = false;
    else
    {
        action->setToolTip(str);
        action->setIcon(QIcon(icon));
        tb->addAction(action);
        connect(action,SIGNAL(triggered()),this,SLOT(onfileprint())); //工具栏按钮点击后触发打印槽函数
    }
    return ret;
}
```

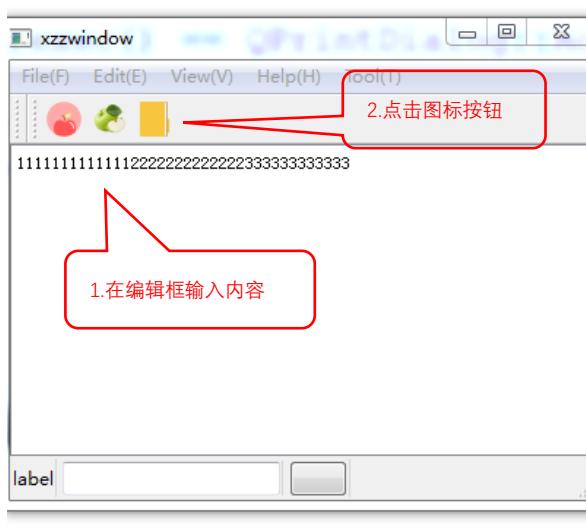
工具栏图标按钮触  
发打印功能槽函数

## 在槽函数文件实现打印槽函数

```
void xzzwindow::onfileprint() //打印槽函数实现
{
    QPrintDialog dlg(this);
    dlg.setWindowTitle("print"); //设置打印对话框名字

    if(dlg.exec() == QPrintDialog::Accepted) //弹出打印对话框
    {
        QPrinter *p = dlg.printer(); //这是创建打印缓冲区
        ptedit.document()->print(p); //将文本里面的数据传入给P缓冲区，自动打印出来
    }
}
```

最主要就是这句，ptedit 对象就是文本编辑器，将编辑器  
里面写的字符串传递给打印指针打印就是了



2.点击图标按钮

1.在编辑框输入内容



3.得到打印对话框

4.选择 PDF 点击打印

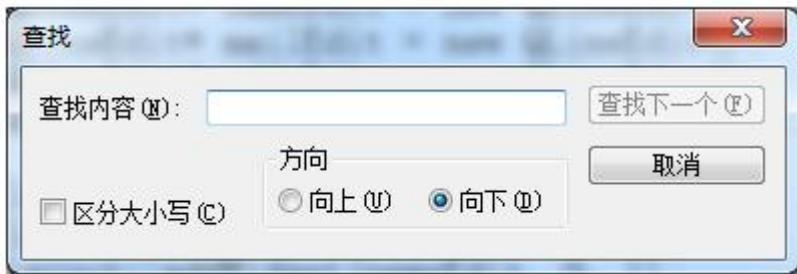


5.弹出 PDF 保存路径对  
话框，指定路径保存

6.pdf 内容出来了

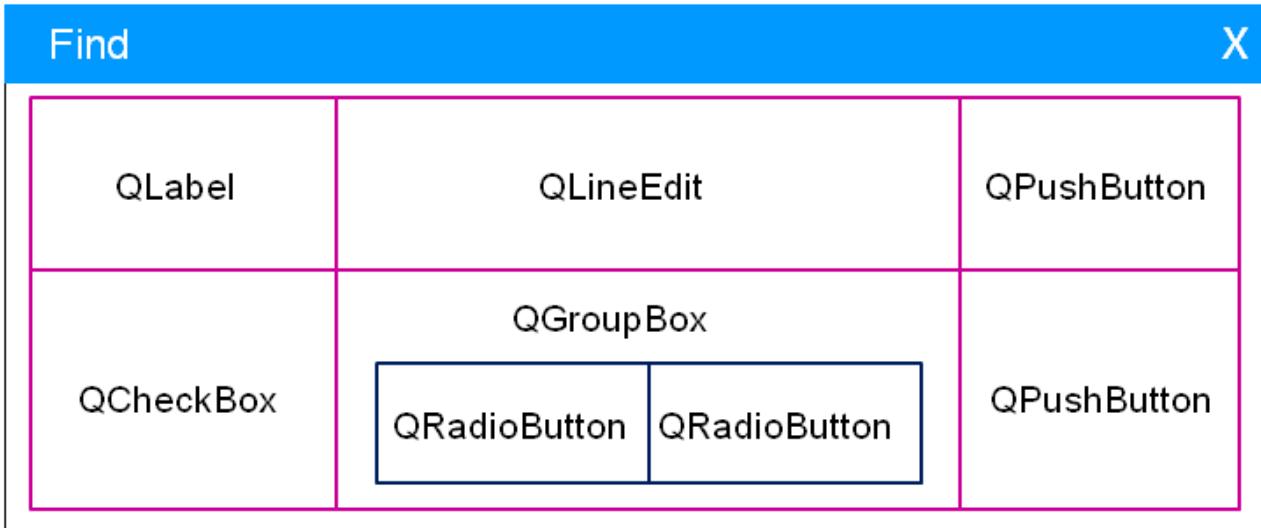
1111111111111222222222223333333333333

## QT 查找对话框实现



实现一个这样的查找窗口

我们要实现一个类，带 cpp 文件和 h 文件，以后其它程序使用这个类就可以建立查找对话框

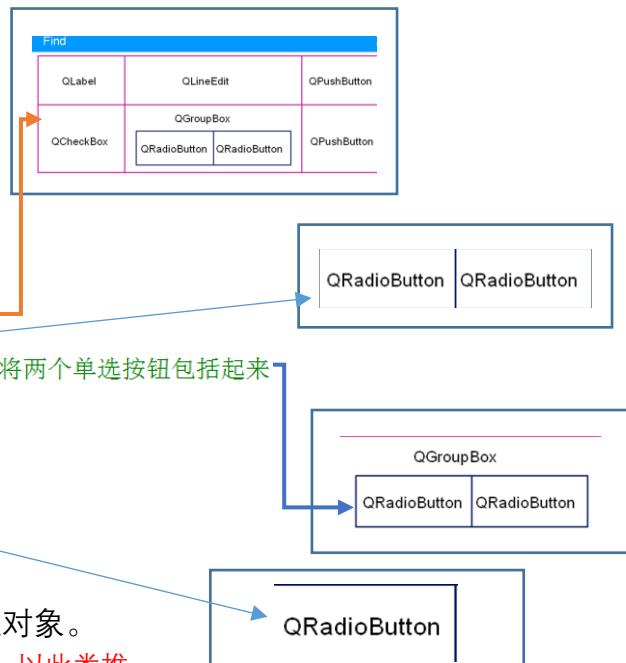


这是这个查找对话框的布局

```
#ifndef FINDDIALOG_H
#define FINDDIALOG_H

#include <QDialog>
#include <QGridLayout>/格子布局管理器
#include <QHBoxLayout>/横向布局管理器
#include <QLabel>/标签
#include <QLineEdit>/文本框
#include <QPushButton>/按钮
#include <QRadioButton>/单选按钮
#include <QCheckBox>/多选按钮
#include <QGroupBox>

class FindDialog : public QDialog
{
    Q_OBJECT
protected:
    QGridLayout m_gridlayout;/格子布局管理器
    QHBoxLayout m_hbox;/横向布局管理器
    QGroupBox m_groupbox;/groupBox的作用就是将两个单选按钮包括起来
    QLabel m_label;/标签
    QLineEdit m_lineEdit;/文本框
    QPushButton m_findbutton;/查找按钮
    QPushButton m_closebutton;/关闭按钮
    QCheckBox m_check;/复选框
    QRadioButton m_forwarBtn;/单选按钮
    QRadioButton m_backBtn;/单选按钮
```



先建立个头文件创建各种按钮，复选框，编辑框对象。

QT 的思路就是先做小窗口，然后用大窗口包含小窗口，以此类推

```

#include "FindDialog.h"

FindDialog::FindDialog(QWidget *parent) : QDialog(parent)
{
    m_label.setText("find what");//对标签进行文字设置
    m_findbutton.setText("find next");//设置按钮名字
    m_closebutton.setText("close");//设置按钮名字
    m_check.setText("math case");//复选框名字

    m_forwarBtn.setText("forwar");//向前查找单选按钮
    m_backBtn.setText("back");//向后查找单选按钮
    m_groupbox.setTitle("dir");//显示方向

    m_hbox.addWidget(&m_forwarBtn);//将按钮放入横向布局管理器
    m_hbox.addWidget(&m_backBtn);//将按钮放入横向布局管理器

    m_groupbox.setLayout(&m_hbox);//将横向的按钮用框框包起来

    m_gridlayout.setSpacing(10);//每个方框间隔10距离
    m_gridlayout.addWidget(&m_label, 0, 0);//标签在格子位置
    m_gridlayout.addWidget(&m_lineEdit, 0, 1);//文本框在格子位置
    m_gridlayout.addWidget(&m_findbutton, 0, 2);//向后查找按钮在格子位置

    m_gridlayout.addWidget(&m_check, 1, 0);
    m_gridlayout.addWidget(&m_groupbox, 1, 1);
    m_gridlayout.addWidget(&m_closebutton, 1, 2);

    setLayout(&m_gridlayout);
}

```

格子窗口设置好后，要用这句来启动

这下 FindDialog 的 cpp 文件和 h 文件就做好了。我们下一步用上面文本编辑器项目的槽函数来调用

```

xzzwindow.h
1 explicit xzzwindow(QWidget *parent = 0);
2 ~xzzwindow();
3 static xzzwindow* creatwindow();//创建窗口
4
5 private:
6     Ui::xzzwindow *ui;
7
8     bool creatmenu();//创建菜单栏
9     bool creatxialamenu(QMenuBar *mb,QString menustr,QString acts
10    bool creatcaidanpro(QMenu* menu,QString actstr);//创建菜单项
11    int Stringchargetnum(QString str,QChar c);
12    int xzzStringcut(QString str,QString *retstr,QChar c);
13    bool creattool(QToolBar* tb,QString str,QString icon);
14    bool initstatus();//创建状态栏
15    bool initxzzEdit();//创建多行文本
16
17    QPlainTextEdit ptedit;//文本编辑框对象
18    QSharedPointer<FindDialog> m_findpt; //定义栈区的智能指针对象
19
20 private slots:
21     void onfilesave(void); //在这里增加一个槽函数
22     void onTextchange(); //文本框内容变化的槽函数
23     void onfileprint(); //定义一个打印功能的槽函数
24     void onEditFind(); //添加一个查找对话框槽函数
25 };

```

在文本编辑器项目的 h 文件定义一个智能指针来指向对话框对象

建立一个槽函数，点击文本编辑器的工具图标就触发对话框显示

这是文本编辑器项目主头文件

下面是文本编辑器项目主 CPP 文件

```
xzzwindow::xzzwindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::xzzwindow), m_findpt(new FindDialog(this))
{
    ui->setupUi(this);
}

bool xzzwindow::creattool(QToolBar* tb,QString str,QString icon)
{
    bool ret = true;
    QAction* action = new QAction("",NULL);
    if(action == NULL)
        ret = false;
    else
    {
        action->setToolTip(str);
        action->setIcon(QIcon(icon));
        tb->addAction(action);
        connect(action,SIGNAL(triggered()),this,SLOT(onEditFind())); //工具栏按钮点击后触发查找对话框槽函数
    }
    return ret;
}

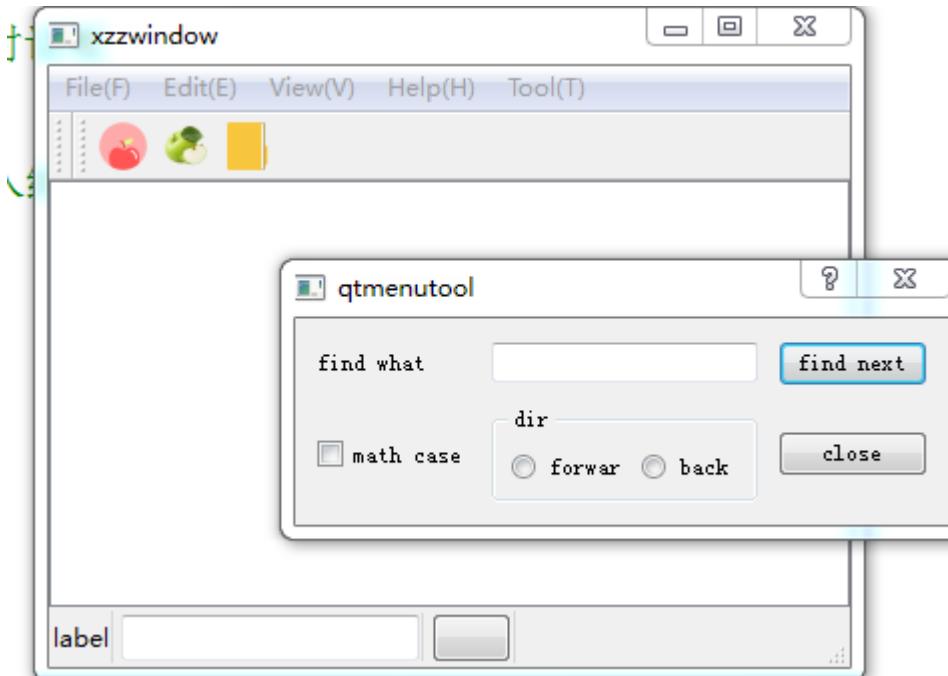
// ... (other code)
```

文本编辑器项目 CPP 文件创建文本编辑窗口的同时就在堆空间申请查找对话框对象，然后用智能指针 m\_findpt 指向查找对话框

创建工具栏图标的时候，给工具栏图标附上对话框触发信号函数

```
void xzzwindow::onEditFind() //查找对话框槽函数实现
{
    m_findpt->show();
}
```

如果工具栏图标的 connect 函数被按钮触发，自动执行槽函数文件的 onEditFind 槽函数，从而触发智能指针 m\_findpt 指定对象里面的对话框显示函数



这就是查找对话框触发结果，这里只是完成了查找对话框的界面程序

## 关于对话框设计，就是显示版本信息和一些废话的对话框



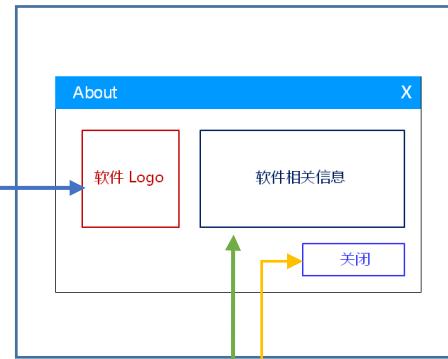
我们要设计一个这样的对话框

```
#include <QWidget>
#include <QLabel> //对话框LOG用Qlabel实现
#include <QPlainTextEdit> //文本编辑框
#include <QPushButton> //关闭按钮

class AboutDialog : public QWidget
{
    Q_OBJECT
    QLabel m_log;
    QPlainTextEdit m_info;
    QPushButton m_close;
public:
    explicit AboutDialog(QWidget *parent = 0);

signals:

public slots:
```



在头文件实现需要的对话框

组件

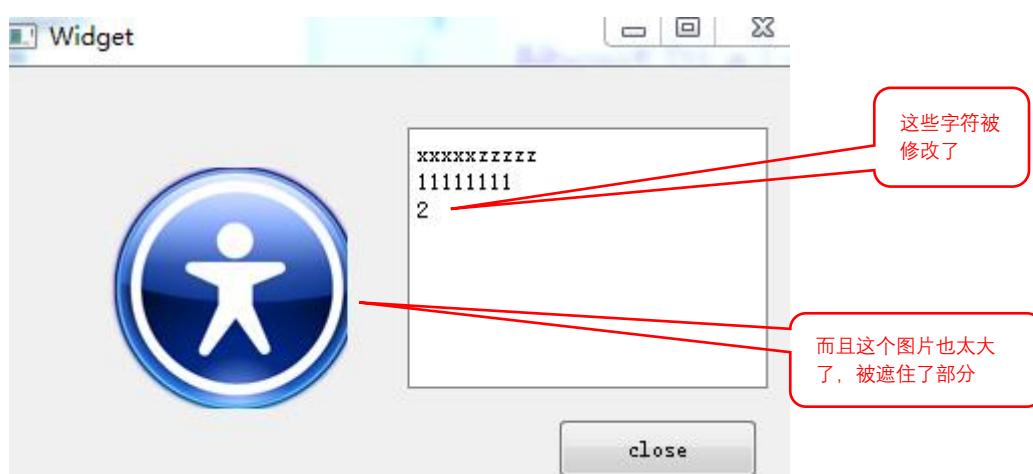
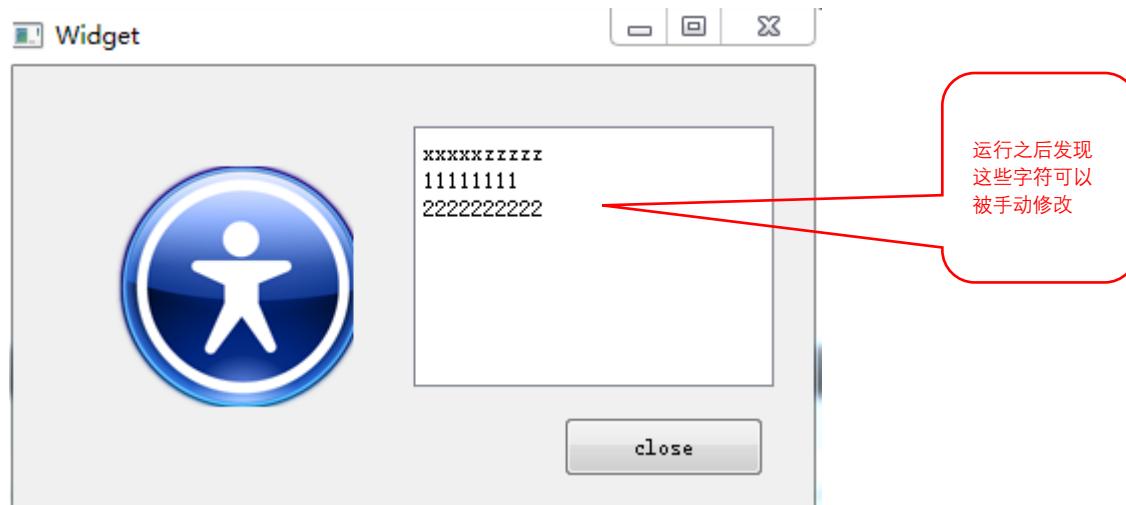
```
#include <QPixmap> //位图转换
#include "aboutdialog.h"

AboutDialog::AboutDialog(QWidget *parent) :
    QWidget(parent), m_log(this), m_info(this), m_close(this)
{
    m_log.move(50, 50);
    m_log.resize(120, 120);
    m_log.setPixmap(QPixmap(":/source/01.png")); //对话框log的png图片转换成位图显示

    m_info.move(200, 30);
    m_info.resize(180, 130);
    m_info.insertPlainText("xxxxxxxx\n11111111\n2222222222\nn"); //显示废话的字符串

    m_close.move(275, 175);
    m_close.resize(100, 30);
    m_close.setText("close"); //关闭按钮
}
```

在 C 文件实现具体内容



```

#include <QPixmap> //位图转换
#include "aboutdialog.h"

AboutDialog::AboutDialog(QWidget *parent) :
    QWidget(parent), m_log(this), m_info(this), m_close(this)
{
    QPixmap pm(":/source/01.png");
    pm = pm.scaled(120, 120, Qt::KeepAspectRatio); //在位图显示之前修改位图等比变化
    //位图缩小到120,120像素
    m_log.move(50, 50);
    m_log.resize(120, 120);
    m_log.setPixmap(QPixmap(pm)); //修改对话框log的png参数
    m_info.move(200, 30);
    m_info.resize(180, 130);
    m_info.insertPlainText("xxxxxxxx\n11111111\n2222222222\n"); //显示废话的字符
    m_info.setReadOnly(true); //这样你的字符就不会被修改了

    m_close.move(275, 175);
    m_close.resize(100, 30);
    m_close.setText("close"); //关闭按钮
}

```



1. 修改边框里面的白色为窗口背景色可以用 QSS，我这里用 QT 原生代码实现，比较复杂

2. 修改边框很简单

```

#include <QPixmap> //位图转换
#include <QPalette> //加入调色板类

```

修改背景色要加入调色板类

```

AboutDialog::AboutDialog(QWidget *parent) :
    QWidget(parent), m_log(this), m_info(this), m_close(this)

QPixmap pm(":/source/01.png");
pm = pm.scaled(120, 120, Qt::KeepAspectRatio); //在位图显示之前修改位图等比变化
//位图缩小到120,120像素
m_log.move(50, 50);
m_log.resize(120, 120);
m_log.setPixmap(QPixmap(pm)); //修改对话框log的png参数

QPalette p = m_info.palette(); //每个盒子框框形状的组件都有调色板函数，我们拿到文本框的调色板
p.setColor(QPalette::Active, QPalette::Base, palette().color(QPalette::Active, QPalette::Background));
p.setColor(QPalette::Inactive, QPalette::Base, palette().color(QPalette::Inactive, QPalette::Background))

m_info.move(200, 30);
m_info.resize(180, 130);
m_info.setPalette(p); //然后设置文本框为现在的得到的背景
m_info setFrameStyle(QFrame::NoFrame); //设置为无边框
m_info.insertPlainText("xxxxxxxx\n11111111\n2222222222\n"); //显示废话的字符

```

这是获取当前窗口的背景色

然后设置文本框为现在的得到的背景

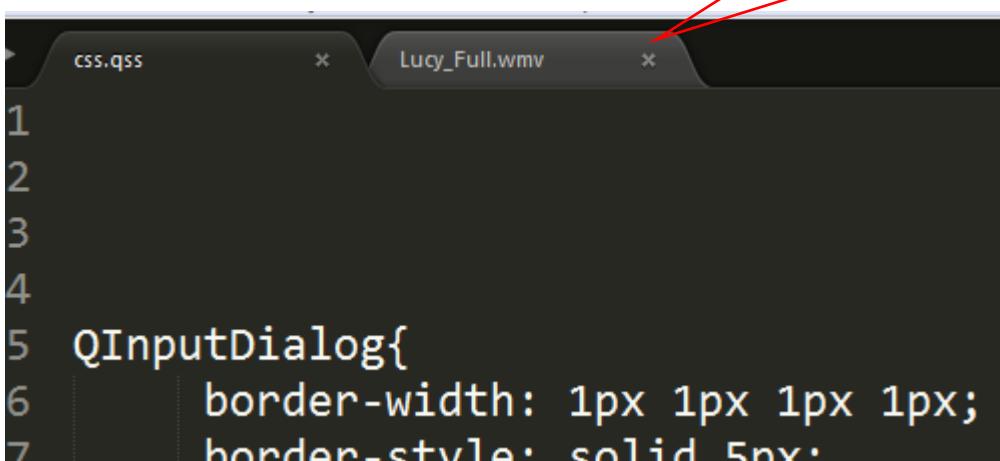
这是把当前窗口的背景色设置给文本框组件



你看文本框的背景色和边框都没有了

## 多窗口切换页面

多个窗体就像 sublime 这种，可以编辑多个文本框



QTabWidget 类实现多窗体切换

```
#include <QTabWidget> // 多页面(窗口)切换必须要有这个类
```

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    QTabWidget m_tabwidget;
```

定义一个多  
页面窗口

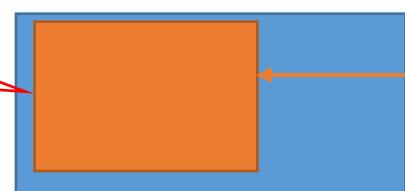
```
m_tabwidget.setParent(&w); // 将页面切换窗口放入父窗口
m_tabwidget.move(10, 10); // 页面切换窗口在父窗口什么位置
m_tabwidget.resize(200, 200);
```

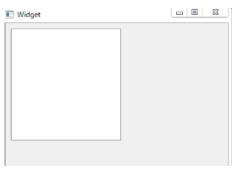
```
w.show();
```

将多页面窗口放入父窗口，设置多页  
面窗口大小，和在父窗口中的位置

```
return a.exec();
```

```
}
```





运行之后发现多页面窗口好 low

```
#include <QPlainTextEdit>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    QTabWidget m_tabwidget;
    QPlainTextEdit plaintext(&m_tabwidget);

    m_tabwidget.setParent(&w); //将页面切换窗口放入父窗口
    m_tabwidget.move(10,10); //页面切换窗口在父窗口什么位置
    m_tabwidget.resize(200,200);

    plaintext.insertPlainText("1st plan text");

    m_tabwidget.addTab(&plaintext, "1st");
    w.show();

    return a.exec();
}
```

这是在初始化文本编辑框时，把文本编辑框放入多页面切换框

这是给多页面切换框增加 tab 项

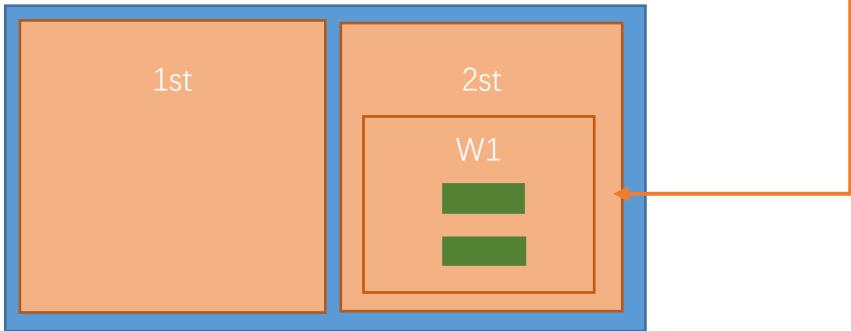
我如何增加多个 tab 窗口呢？

```
11
12     QTabWidget m_tabwidget;
13     QPlainTextEdit plaintext;
14
15     m_tabwidget.setParent(
16         m_tabwidget.move(10,10);
17         m_tabwidget.resize(200,200);
18
19         plaintext.insertPlainText("1st plan text");
20
21         m_tabwidget.addTab(&plaintext, "1st");
22
23         QWidget w1;
24         QPushButton b1(&w1);
25         QPushButton b2(&w1);
26         QVBoxLayout layout;
27
28         b1.setText("button1");//给按钮取名字
29         b2.setText("button2");
30
31         w1.setLayout(&layout);//把布局管理器放入w1窗口
32         layout.addWidget(&b1);//把按钮1放入布局管理器
33         layout.addWidget(&b2);//把按钮2放入布局管理器
34
35         m_tabwidget.addTab(&w1, "2st");
```

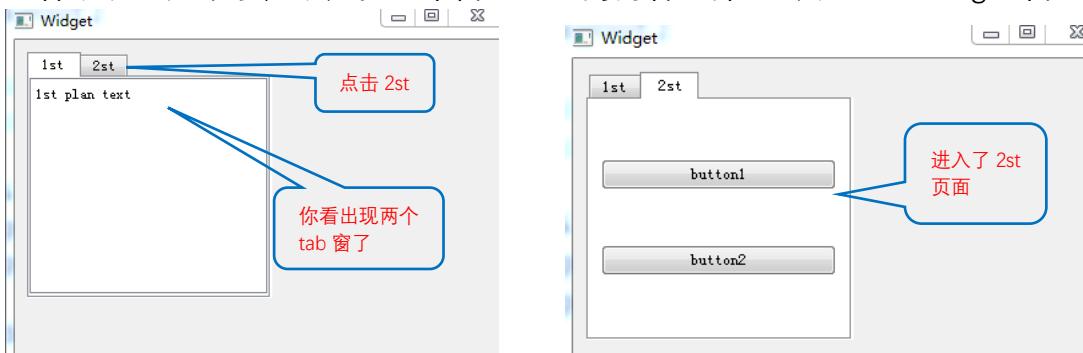
放入父窗口  
窗口什么位置

W1  
垂直布局管理器

W1

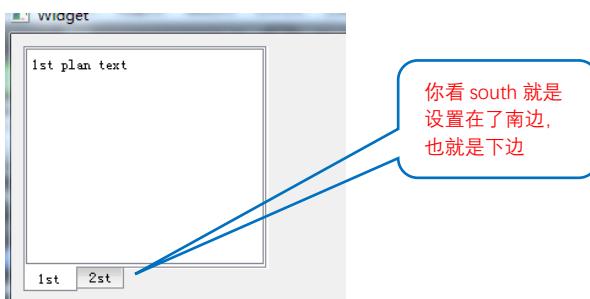


这样就设置完毕了，就是把一个窗口里面的内容全部一套放入 tabwidget 窗口



我现在想把 tab 框设置在上下左右任何一个位置

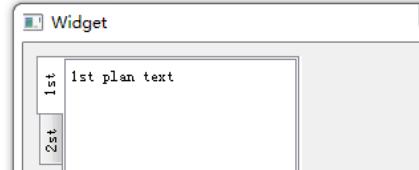
```
m_tabwidget.move(10,10); //页面切换窗口在父窗口什么位置  
m_tabwidget.resize(200,200);  
m_tabwidget.setTabPosition(QTabWidget::South); //将tab栏设置在下边
```



```
m_tabwidget.setTabPosition(QTabWidget::East); //将tab栏设置在东边
```



```
m_tabwidget.setTabPosition(QTabWidget::West); //将tab栏设置在西边
```



```

7     m_tabwidget.setTabPosition(QTabWidget::North); //将tab栏设置在北边
8     m_tabwidget.setTabShape(QTabWidget::Triangular); //设置tab框边框为梯形
9
10    plaintext.insertPlainText("1st plan text");

```

```

m_tabwidget.setTabShape(QTabWidget::Triangular); //设置tab框边框为梯形
m_tabwidget.setTabsClosable(true); //设置Tab框有关闭选项
plaintext.insertPlainText("1st plan text");

```

现在我们想切换窗口和点击关闭 xx 有事件反映

QTabWidget 这个类系统预定义了信号函数

```

9
10 class Widget : public QWidget
11 {
12     Q_OBJECT
13
14 public:
15     explicit Widget(QWidget *parent = 0);
16     ~Widget();
17
18 protected slots:
19     void ontab(int index); //页面切换窗口发生变化的槽函数响应
20     void closetab(int index); //关闭切换窗口某个窗口的槽函数响应
21
22 private:
23     Ui::Widget *ui;
24
25
26 };

```

所以我们只需要自定义槽函数就行了

我定义一个切换窗口会产生信号的接受信号槽函数

我在定义一个点击 xx 就会产生信号的接受信号槽函数

这是 h 文件

```

void Widget::ontab(int index)
{
    qDebug() << "tabwidget change";
}

void Widget::closetab(int index)
{
    qDebug() << "closetab";
}

```

我们将接受到的页面切换信号打印出来

我们将接受到的 xx 信号打印出来

这是 Cpp 文件

```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    QTabWidget m_tabwidget;
    QPlainTextEdit plaintext(&m_tabwidget);

    m_tabwidget.setParent(&w); //将页面切换窗口放入父窗口
    m_tabwidget.move(10,10); //页面切换窗口在父窗口什么位置
    m_tabwidget.resize(200,200);
    m_tabwidget.setTabPosition(QTabWidget::North); //将tab栏设置在北边
    m_tabwidget.setTabShape(QTabWidget::Triangular); //设置tab框边框为梯形

    QObject::connect(&m_tabwidget, SIGNAL(currentChanged(int)), &w, SLOT(ontab(int)));
    QObject::connect(&m_tabwidget, SIGNAL(tabCloseRequested(int)), &w, SLOT(closetab(int)));
    w.show();

    return a.exec();
}

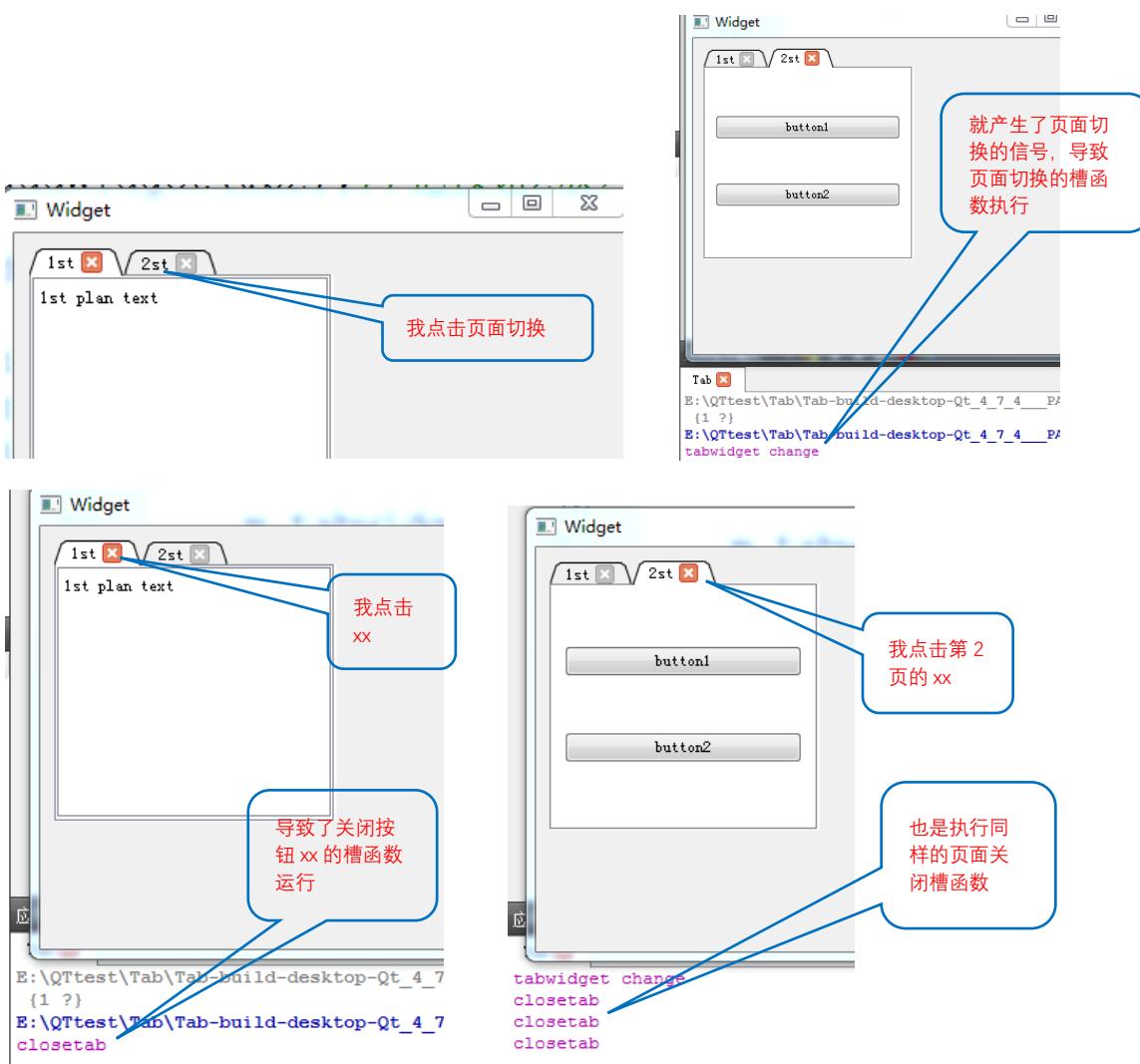
```

这是 QTabWidget 自带的点击 xx 信号发送函数

在主函数里面实现信号与槽的链接

这是 QTabWidget 自带的页面切换信号发送函数

这里就填写我自定义的页面切换接受槽函数就是了



我们现在可以利用这个功能来关闭不要的 tab 页面

```

m_tabwidget.addTab(&plaintext, "1st");
QWidget w1;
QPushButton b1(&w1);
QPushButton b2(&w1);
QVBoxLayout layout;

b1.setText("button1");//给按钮取名字
b2.setText("button2");

w1.setLayout(&layout); //把布局管理器放入w1窗口
layout.addWidget(&b1); //把按钮1放入布局管理器
layout.addWidget(&b2); //把按钮2放入布局管理器

m_tabwidget.addTab(&w1, "2st");

```

第 1 个 addTab 下标为 0 号

第 2 个 addTab 下标为 1 号

所以如果再加个 addTab 下标就为 2 号，以此类推

```

void Widget::closetab(int index)
{
    qDebug() << "closetab";
    qDebug() << index;
}

```



我们在关闭按钮的槽函数打印看看是不是有下标

点击 1st 的 xx 打印的 0 下标，点击 2st 的 xx 打印的 1 下标

那我们可以用 QTabWidget 的 removeTab 函数来输入下标，关闭对应的页面窗口

```

void Widget::onTabCloseRequested(int index)
{
    m_tabWidget.removeTab(index);
}

```

因为我的 `m_tabWidget` 不是全局变量，所以我的槽函数无法调用 `m_tabWidget`，如果要实现关闭 `xx`，就让 `m_tabWidget` 能被槽函数调用就是了。

## QT 表格界面设计

```

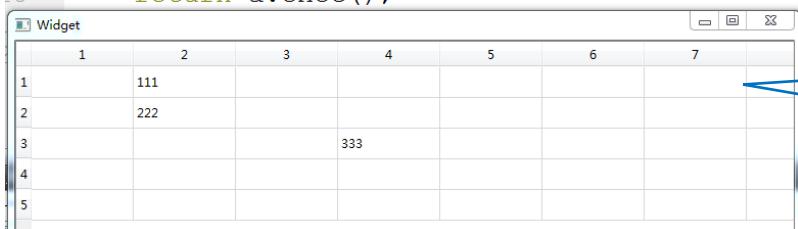
3 #include <QTableWidget> //表格头文件
4
5 int main(int argc, char *argv[])
6 {
7     QApplication a(argc, argv);
8     Widget w;
9
10    QTableWidget table(&w);
11
12    table.setRowCount(5); //设置表格5行
13    table.setColumnCount(10); //设置表格10列
14    table.resize(1000, 500);
15
16    w.show();
17
18    return a.exec();

```

表格放在父窗口里面

表格设置成 5 行 10 列

设置表格占用父窗口的大小

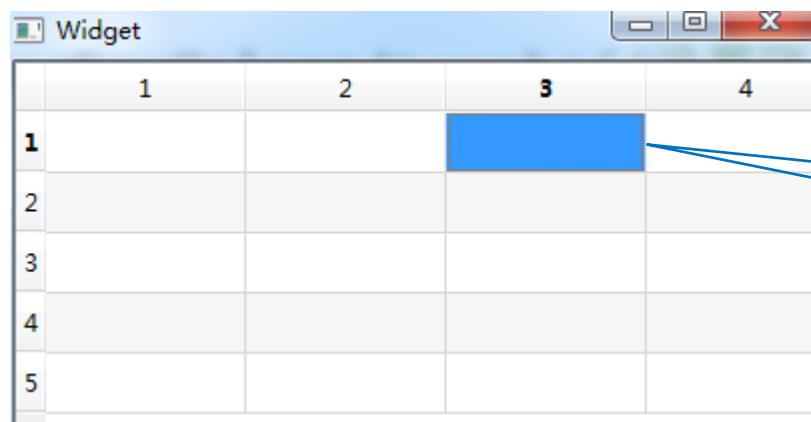
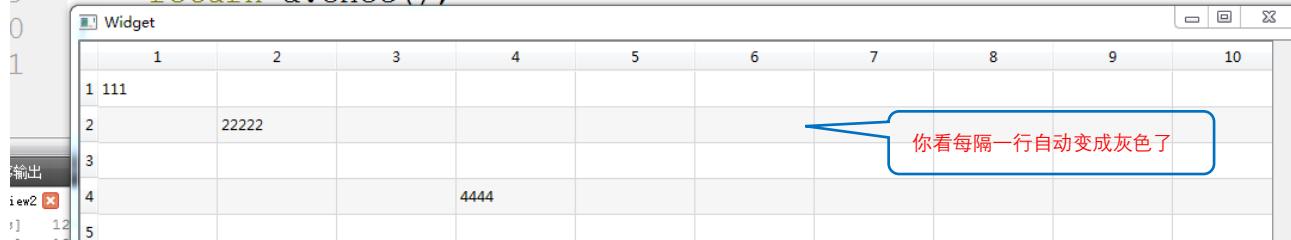


表格列数太多，拉进度条看

```

3 #include <QTableWidget> //表格头文件
4
5 int main(int argc, char *argv[])
6 {
7     QApplication a(argc, argv);
8     Widget w;
9
10    QTableWidget table(&w);
11
12    table.setRowCount(5); //设置表格5行
13    table.setColumnCount(10); //设置表格10列
14    table.resize(1000, 500);
15
16    table.setAlternatingRowColors(true); //设置隔行自动变色
17    w.show();
18
19    return a.exec();
20
21

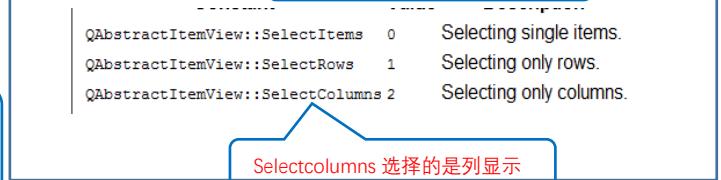
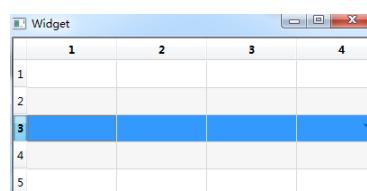
```



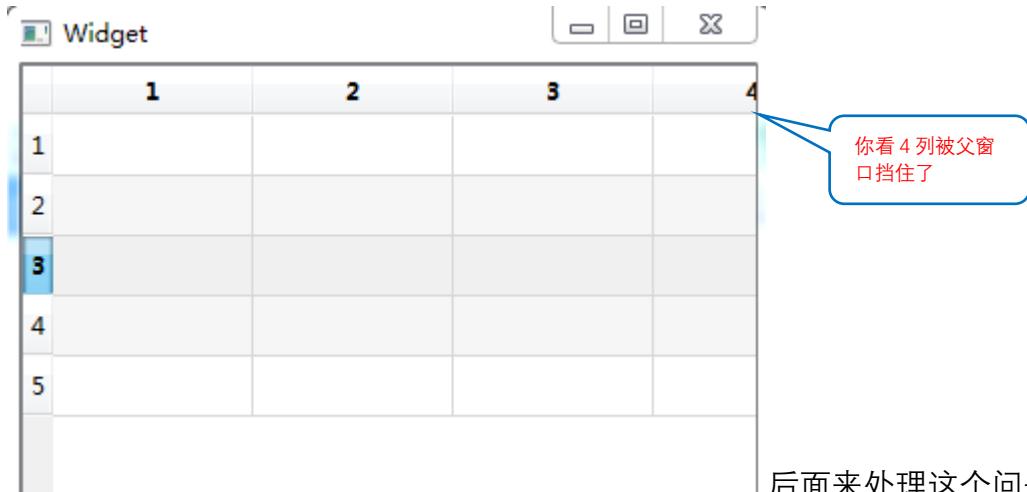
```

9
10   QTableWidget table(&w);
11
12   table.setRowCount(5); //设置表格5行
13   table.setColumnCount(10); //设置表格10列
14   table.resize(1000, 500);
15
16   table.setAlternatingRowColors(true); //设置隔行自动变色
17   table.setSelectionBehavior(QAbstractItemView::SelectRows); //让鼠标选择时显示一行
18   w.show();
19
20   return a.exec();
21

```



我发现表格窗口无法适应父窗口的大小变化



后面来处理这个问题

我现在想插入行,插入行就得加入按钮事件的信号

```
4  #include<QWidget>
5  #include<QDialog>
6  #include<QPushButton>
7  #include <QTableWidget>
8
9  class xzztableweight:public QWidget
10 {
11     Q_OBJECT
12 protected:
13
14 private slots:
15     void addrow();
16     void delrow();
```

这是插入行槽函数

这是删除行槽函数

```
17
18 public:
19     xzztableweight(QWidget *w);
20     ~xzztableweight();
21     QTableWidget *table;
22     QPushButton *button;
23 }
```

这是头文件

下面是 cpp 文件

```
6  xzztableweight::xzztableweight(QWidget *w) //传入父窗口进来, 让表格和按钮嵌入进父窗口
7  {
8
9      table = new QTableWidget(w); //创建的表格必须是堆空间,
10                                //因为直接定义对象, 构造函数执行完了会释放这些内存,
11                                //导致最后只显示父窗口
12      button = new QPushButton(w);
13
14      table->setRowCount(5); //设置表格5行
15      table->setColumnCount(10); //设置表格10列
16      table->resize(500, 300);
17
18      table->setAlternatingRowColors(true); //设置隔行自动变色
19      table->setSelectionBehavior(QAbstractItemView::SelectRows); //让鼠标选择时显示一行
20      button->move(100, 200);
21      button->resize(100, 50);
22      connect(button, SIGNAL(clicked()), this, SLOT(addrow()));
23
24 }
```

我们把表格和按钮创建起来

```

27 void xzztableweight::addrw()
28 {
29     int row;
30     row = table->currentRow(); // 获取当前行
31     table->insertRow(row);
32 }
33 void xzztableweight::delrow()
34 {
35
36 }

```

QTableWidgetItem::currentRow  
函数是获取你现在鼠标点击的某一行，如果你鼠标不点击，就默认最后一行

QTableWidgetItem::insertRow  
是插入行，传入你现在点击的行，在你现在点击的行后面插入 1 行

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 1 |   |   |   |   |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |
| 5 |   |   |   |   |

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 1 |   |   |   |   |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |
| 5 |   |   |   |   |
| 6 |   |   |   |   |

如果我要删除一行呢？

```

27 void xzztableweight::addrw()
28 {
29     int row;
30     row = table->currentRow(); // 获取当前行
31     table->removeRow(row); // 删除你现在指定的行
32 }

```

还是在槽函数里面获取现在你选择的行，或者没选择就是最后一行

然后删除你选择的行，或者默认的最后一行

| 1 | 2 | 3     | 4 | 5 |
|---|---|-------|---|---|
| 1 |   |       |   |   |
| 2 |   |       |   |   |
| 3 |   | 11111 |   |   |
| 4 |   |       |   |   |
| 5 |   |       |   |   |

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 1 |   |   |   |   |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |
| 5 |   |   |   |   |

下面我们实现表格的快速搜索

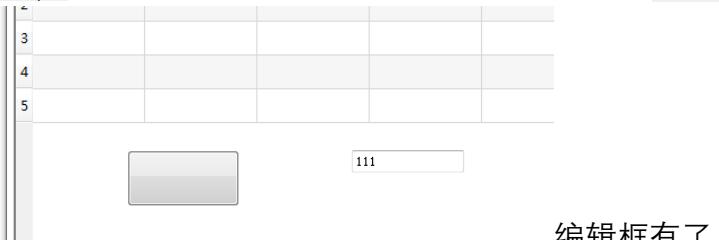
```
1 #include<QTableWidget>
2
3
4
5
6 class xzztableweight:public QWidget
7 {
8     Q_OBJECT
9 protected:
10
11     private slots:
12         void addrow();
13         void delrow();
14
15 public:
16     xzztableweight(QWidget *w);
17     ~xzztableweight();
18     QTableWidget *table;
19     QPushButton *button;
20     QLineEdit *line;
21
22
23
24
25
```

在头文件创造一个编辑框

```
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

在 Cpp 文件实现

```
line = new QLineEdit(w);
line->move(300, 200);
line->resize(100, 20);
```



编辑框有了。

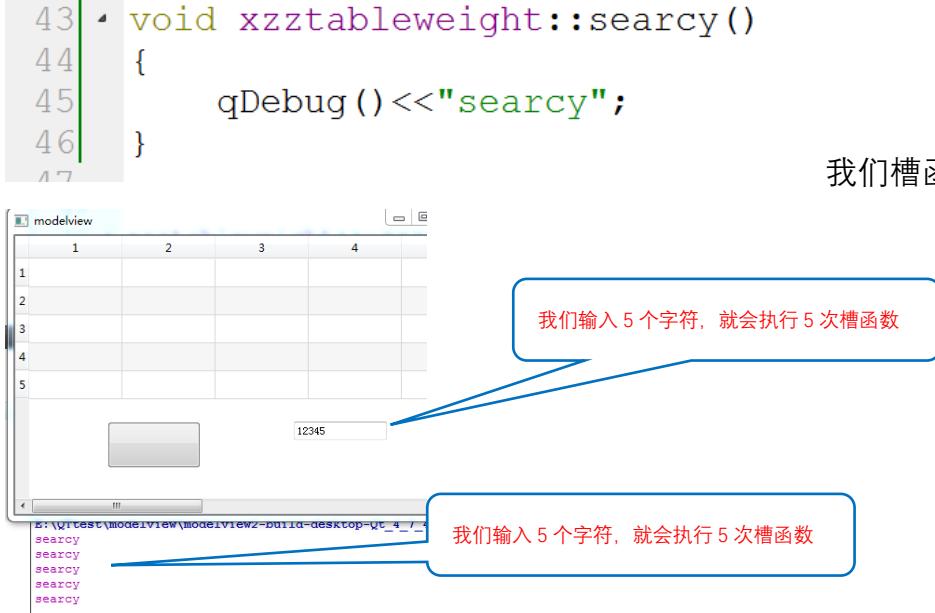
我们要在编辑框里面输入内容，就马上执行搜索函数，那么就必须对编辑框加信号槽函数

```
16 private slots:
17     void addrow();
18     void delrow();
19     void searcy();
```

在头文件建立 searcy 槽函数

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43 void xzztableweight::searcy()
44 {
45     qDebug() << "searcy";
46 }
```

在 Cpp 文件实现槽函数，这里一定要用 textChanged 信号，因为这个信号功能是编辑框被输入一个字符，槽函数就必须响应



我们槽函数打印字符，来实验下

所以懂了撒，就是要把字符搜索函数放在槽函数里面，每次编辑框输入字符，槽函数循环执行搜索

```

QString().isEmpty(); // returns true 字符串为空返回 true
QString("").isEmpty(); // returns true 字符串为空返回 true
QString("x").isEmpty(); // returns false 有字符串返回 false
QString("abc").isEmpty(); // returns false

```

我们来写个字符同名搜索算法

```

43 void xzztableweight::searcy()
44 {
45     QString arg="";
46     arg = line->text();
47     if(arg.isEmpty())
48     {
49         int rowCount = table->rowCount();
50         for(int row = 0;row<rowCount;row++)
51         {
52             table->showRow(row);
53         }
54         return ;
55     }
56
57     int rowCount = table->rowCount();
58     int columCount = table->columnCount();
59
60     qDebug()<<"row"<<rowCount;
61     for(int row = 0;row<rowCount;row++)
62     {
63         QString rowdata;
64         for(int colum = 0;colum < columCount;colum++)
65         {
66             if(table->item(row,colum))
67             {
68                 rowdata = rowdata + (table->item(row,colum)->text());
69             }
70             qDebug()<<rowdata;
71             if(!rowdata.isEmpty())//有数据
72             {
73                 if(rowdata.contains(arg))
74                 {
75                     table->showRow(row);
76                 }
77                 else
78                 {
79                     table->hideRow(row);
80                 }
81             }
82             else //无数据
83             {
84                 table->hideRow(row);
85             }
86         }
87     }
88 }
89
90 }
91

```

QString::contains 就实现了，你在 QLineEdit 编辑框内输入一个字符，也能匹配，输入两个字符也能再去执行槽函数，匹配两个字符，以此类推

modelview

| 1 | 2    | 3    | 4   | 5 |
|---|------|------|-----|---|
| 1 |      |      |     |   |
| 2 | 1234 |      |     |   |
| 3 |      | 8951 |     |   |
| 4 | bbbb |      |     |   |
| 5 |      |      | aaa |   |

Two rectangular input fields are present below the table: one with a gray background and one with a white background.

modelview

| 1 | 2 | 3    | 4 | 5 |
|---|---|------|---|---|
| 3 |   | 8951 |   |   |

Two rectangular input fields are present below the table: one with a gray background and one with a white background. A blue callout bubble points to the white input field containing the value "89".

你看我输入我要的字符,  
就只显示我要的字符了

## 列表实现

使用 QTreeWidget 这个类

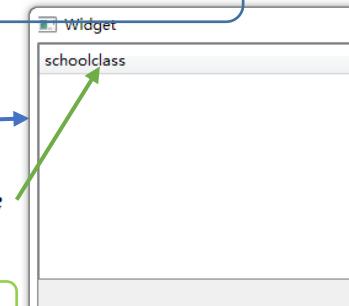
```
#include <QTreeWidgetItem> // 列表使用要包含该头文件

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    QTreeWidget treewidget(&w);

    treewidget.setHeaderLabel("schoolclass");

    w.show();
    return a.exec();
}
```

列表需要专用的框框，所以用 QTreeWidget 在父窗口里面建立个列表使用的框框



这是给列表头取个名字，否则默认名字是 1

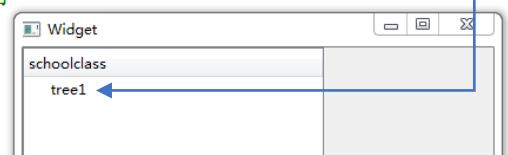
```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    QTreeWidget treewidget(&w);

    treewidget.setHeaderLabel("schoolclass");

    QTreeWidgetItem tree1(&treewidget); // 在列表头里面建立子列表
    tree1.setText(0, "tree1"); // 子列表名字

    w.show();
    return a.exec();
}
```

这个 tree1 列表就是在 treewidget 下面建立的



给 tree1 列表写上名字，这个 0 是让 tree1 字符显示在第 0 列，如果你写 1 就看不到字符了

```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    QTreeWidget treewidget(&w);

    treewidget.setHeaderLabel("schoolclass");

    QTreeWidgetItem tree1(&treewidget); //在列表头里面建立子列表
    tree1.setText(0, "tree1"); //子列表名字

    QTreeWidgetItem tree2(&treewidget); //在列表头里面建立更多的子列表
    tree2.setText(0, "tree2"); //子列表名字

    QTreeWidgetItem tree3(&treewidget); //在列表头里面建立更多的子列表
    tree3.setText(0, "tree3"); //子列表名字

```



其实还可以子列表里面再建立子列表，也就是子列表自身的子列表，在下面编写。

```

QApplication a(argc, argv);
Widget w;
QTreeWidget treewidget(&w);

treewidget.setHeaderLabel("schoolclass");

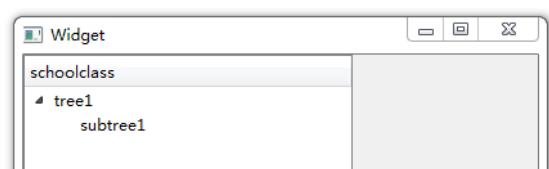
QTreeWidgetItem tree1(&treewidget); //在列表头里面建立子列表
tree1.setText(0, "tree1"); //子列表名字

QTreeWidgetItem subtree1(&tree1); //在子列表里面再建立子列表
subtree1.setText(0, "subtree1"); //子子列表名字

w.show();

```

要把新建的列表对象父类写成子列表，那么你这个新建的列表对象就是子子列表



```

treewidget.setHeaderLabel("schoolclass");

QTreeWidgetItem tree1(&treewidget); //在列表头里面建立子列表
tree1.setText(0, "tree1"); //子列表名字

QTreeWidgetItem subtree1(&tree1); //在子列表里面再建立子列表
subtree1.setText(0, "subtree1"); //子子列表名字

QTreeWidgetItem subtree2(&tree1); //在子列表里面再建立子列表2
subtree2.setText(0, "subtree2"); //子子列表2名字

```



可以建立多个子子列表，就是把新建的列表对象父类指定下就是了。

现在我们加入右键点击，增加列表和删除列表的功能

```

class xzzmenu : public QWidget
{
    Q_OBJECT
public:
    explicit xzzmenu(QWidget *parent = 0);
    xzzmenu();
    void creatmenu(void);
signals:
public slots:
private:
    QMenu *m_totalmenu;//总菜单
    QMenu *m_Add; //添加列表
    QMenu *m_del; //删除列表

    QAction *addschool; //菜单添加按钮下的添加学校选项需要action来承载
    QAction *addclass; //菜单添加班级选项
    QAction *del_action; //删除
};

void xzzmenu::creatmenu(void)
{
    m_Add = new QMenu; //建立添加列表菜单

    addschool = new QAction("add school", NULL); //在菜单选项显示建立学校列表
    addclass = new QAction("addclass", NULL); //在菜单选项显示建立教室列表

    m_Add->addAction(addschool); //将选项加入进行菜单
    m_Add->addAction(addclass); //将选项加入进菜单

    m_Add->exec(); //显示菜单
}

```

我们在头文件里面建立起我们需要的菜单和选项

整个框是菜单

框框里面的每一行选项是 action

执行 creatmenu 函数里面的 exec(), 就会显示出整个框

但是我发现这两个选项属于 Add 菜单, add 菜单在哪里呢? 看来 add 菜单是现在 m\_add 菜单的父菜单

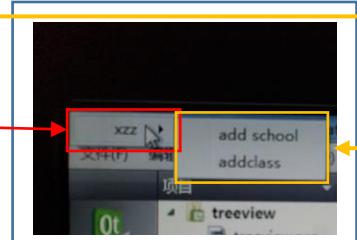
```
void xzzmenu::creatmenu(void)
{
    m_totalmenuAdd = new QMenu;//add总菜单
    m_Add = new QMenu("xzz");//建立添加列表菜单
    addschool = new QAction("add school",NULL);//在菜单选项显示建立学校列表
    addclass = new QAction("addclass",NULL);//在菜单选项显示建立教室列表

    m_Add->addAction(addschool);//将选项加入进行菜单
    m_Add->addAction(addclass);//将选项加入进菜单

    m_totalmenuAdd->addMenu(m_Add);
    m_totalmenuAdd->exec();
}
```

最顶层的父菜单框是无法设置名字的

只有顶层菜单的第一个子菜单开始才有名字



现在我想菜单根据鼠标的位置右键点击出现。

## 有个重要的错误 collect2:ld returned 1 exit status 解决方法

① collect2: ld returned 1 exit status

这是因为你 QT creator 工程里面加入了有类定义的 C 文件和 H 文件。  
但是 QT 很奇怪这种类定义的 C 文件和 H 文件必须用 Qtcreator 工程里面的类选项来创建

```
"-----"
class xzzmenu : public QWidget
{
    Q_OBJECT
public:
    explicit xzzmenu(QWidget *parent = 0);

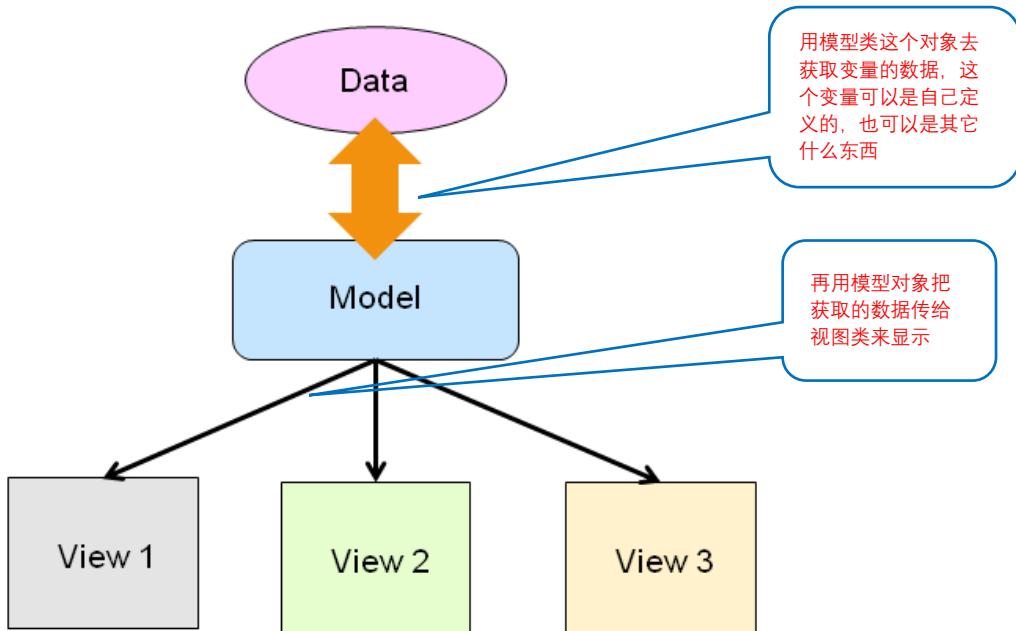
signals:
public slots:
};

-----"
```

因为用 QT creator 软件创建的类文件，在 h 文件里面有 signals: 和 slots 的定义，也许你在自己创建的 C 文件和 h 文件里面加入这个就可以了

反正这个问题现在时用 QT creator 里面的工程创建类文件解决的。

## 模型视图设计模式方法



为什么不直接把数据传给视图类来显示呢？我也很奇怪，后面就知道了。

```
3 | #include <QFileSystemModel> //文件系统模型类需要的头文件
4 | #include <QTreeView> //定义视图类头文件
```

```
5 | int main(int argc, char *argv[])
6 | {
```

```
7 |     QApplication a(argc, argv);
8 |     Widget w;
```

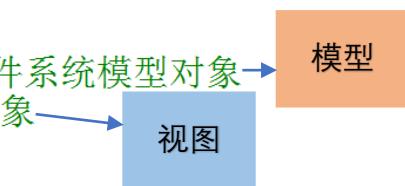
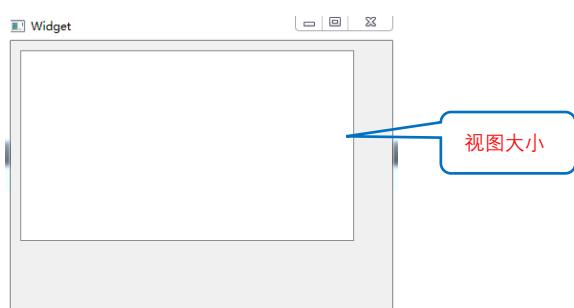
```
9 |     QFileSystemModel m_fsmodel; //文件系统模型对象 → 模型
10 |    QTreeView m_treeview; //视图模型对象 → 视图
```

```
11 |    m_treeview.setParent(&w); //将视图这个框框放到w窗口下
12 |    m_treeview.move(10, 10); //视图框放在w窗口的坐标10, 10位置
13 |    m_treeview.resize(350, 200); //视图框占用w窗口的大小
```

```
14 |    w.show();
```

```
15 |    return a.exec();
```

视图占用 w 窗口的大小



视图

w

```

#include <QDir>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    QFileSystemModel m_fsmode; //文件系统模型对象
    QTreeView m_treeview; //视图模型对象

    m_treeview.setParent(&w); //将视图这个框框放到w窗口下
    m_treeview.move(10, 10); //视图框放在w窗口的坐标10, 10位置
    m_treeview.resize(350, 200); //视图框占用w窗口的大小

    m_fsmode.setRootPath(QDir::currentPath()); //m_fsmode模型获取当前工作目录的数据
    m_treeview.setModel(&m_fsmode); //将m_fsmode模型和视图m_treeview链接起来

    m_treeview.setRootIndex(m_fsmode.index(QDir::currentPath())); //数据从视图的根节点开始向下显示

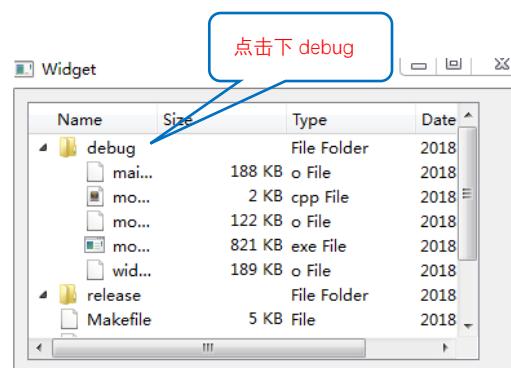
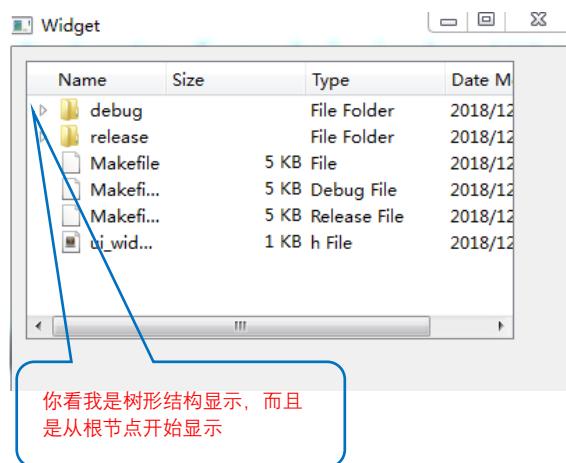
    w.show();
    return a.exec();
}

```

`m_fsmode.index` //是选择模型里面现有目录数据，是从目录中间的数据开始获取，还是从第1个根目录开始向下获取，`index`就是从根目录向下获取

`setRootIndex(m_fsmode)` //现在获取到的根目录数据放在了 `m_fsmode.index` 返回的变量中，

`setRootIndex` //这表示变量是从视图的根目录开始显示还是从视图的树状其它支叶开始显示



这就是树状显示，TreeView 视图类的功能

我用模式视图有什么用，感觉还是没有用啊！！！！

## 用表格案例来解释模式视图的基本作用

QStandardItemModel

在头文件定义模型和两个函数,一个函数交由一个工程师实现

```
#include <QWidget>
#include <QTableView> //用表格视图来显示数据
#include <QStandardItemModel> //模型来存放组织的数据

namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

    QStandardItemModel m_model; //模型对象,到时候用来接收组织的数据
    QTableView m_tableview; //显示模型里面的数据

    void initmodel(); //组织数据的函数
    void initview(); //显示数据的函数
}

void Widget::initmodel()
{
    QStandardItem *root = m_model.invisibleRootItem(); //invisibleRootItem返回模型根节点给变量

    QStandardItem *itemA = new QStandardItem(); //创建数据项A
    QStandardItem *itemB = new QStandardItem(); //创建数据项B
    QStandardItem *itemC = new QStandardItem(); //创建数据项C

    itemA->setData("Adata"); //向A数据项写3个数据
    itemA->setData(100);
    itemA->setData(1.75);

    itemB->setData("Adata"); //向B数据项写3个数据
    itemB->setData(200);
    itemB->setData(2.75);

    itemC->setData("Adata"); //向C数据项写3个数据
    itemC->setData(300);
    itemC->setData(3.75);

    root->setChild(0, 0, itemA);
    root->setChild(0, 1, itemB);
    root->setChild(1, 0, itemC);
}
```

这个 QStandardItemModel 类就是负责连接模型和视图的,也就是连接 A,B 工程师的数据

A 工程师采集数据,然后把数据做成表格类型的模型

B 工程师负责显示数据,把表格模型的数据用表格视图显示出来

每个数据项变量就是一个格子

向每个格子里面写几个不同类型的数据

虚拟一个表格排列方式,(0,0,数据项 A)就是数据项 A 排第 1 行第 1 格

|       |       |
|-------|-------|
| 数据项 A | 数据项 B |
| 数据项 C |       |

这样 A 工程把数据就组织好了,因为 B 工程师可能会用表格显示所以数据模型就是用的表格方式来排列

```
void Widget::initview()
{
    m_tableview.setParent(this); //这个表格视图窗口就嵌入在widget父窗口
    m_tableview.move(10, 10); //表格在父窗口的位置
    m_tableview.resize(300, 100); //表格大小
}
```

B 工程师只需要实现表格视图函数就是了。

```

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

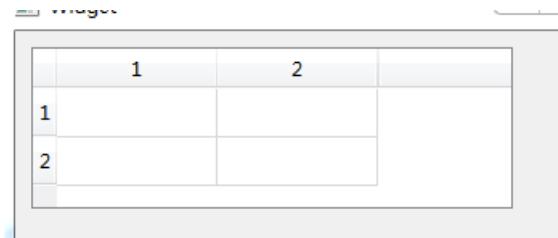
    initmodel();
    initview();

    m_tableview.setModel(&m_model); //用视图的setModel去连接模型，把模型的数据取出来显示
}

```

在构造函数里面，A 工程师的函数和 B 工程师的函数都必须先执行

然后用 B 工程师的表格类自带的模型函数 setModel 把 A 工程组织好数据的模型加载进来



我们运行程序发现表格里面没有数据？怎么回事？

```

itemA->setData("Adata"); //向A数据项写3个数据
itemA->setData(100);
itemA->setData(1.75);

```

```

itemB->setData("Adata"); //向B数据项写3个数据
itemB->setData(200);
itemB->setData(2.75);

```

```

itemC->setData("Adata"); //向B数据项写3个数据
itemC->setData(300);
itemC->setData(3.75);

```

这是因为这些格子里面的数据优先级都是一样的，所以放给表格视图时视图不知道该显示哪个？

所以要给这些数据加上它们该显示的位置

```

void Widget::initmodel()
{
    QStandardItem *root = m_model.invisibleRootItem(); //invisibleRootItem返

    QStandardItem *itemA = new QStandardItem(); //创建数据项A
    QStandardItem *itemB = new QStandardItem(); //创建数据项B
    QStandardItem *itemC = new QStandardItem(); //创建数据项C

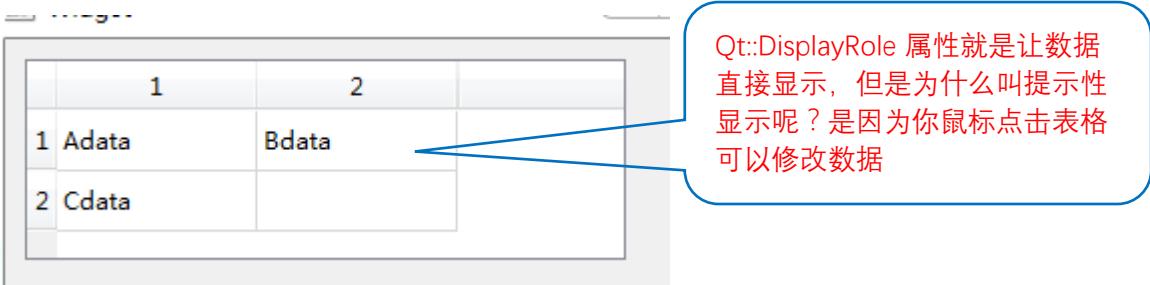
    itemA->setData("Adata", Qt::DisplayRole); //Qt::DisplayRole属性就是让Adata数据作为提示性质的字符串显示
    itemA->setData(100, Qt::ToolTipRole); //Qt::ToolTipRole属性是将数字放在悬浮框显示
    itemA->setData(1.75, Qt::WhatsThisRole);

    itemB->setData("Bdata", Qt::DisplayRole); //给每个数据项加数据显示角色(优先级)
    itemB->setData(200, Qt::ToolTipRole);
    itemB->setData(2.75, Qt::WhatsThisRole);

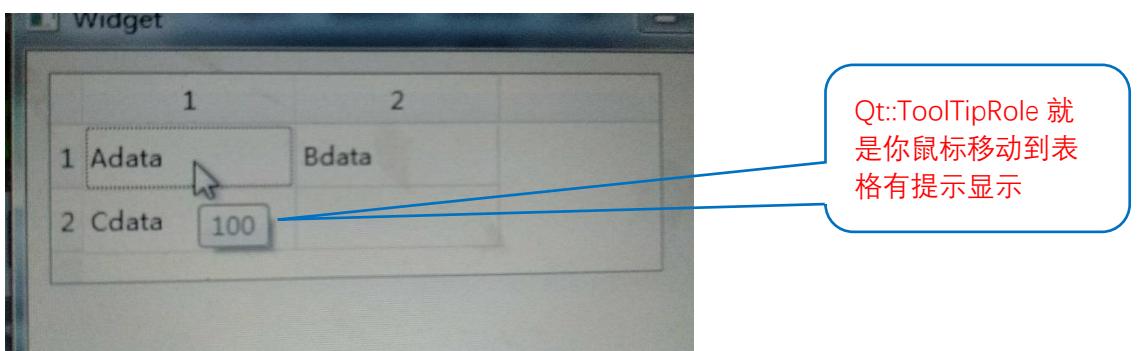
    itemC->setData("Cdata", Qt::DisplayRole); //给每个数据项加数据显示角色(优先级)
    itemC->setData(300, Qt::ToolTipRole);
    itemC->setData(3.75, Qt::WhatsThisRole);
}

这些(Qt::参数)就是给数据增加显示的属性(角色)优先级

```



Qt::DisplayRole 属性就是让数据直接显示，但是为什么叫提示性显示呢？是因为你鼠标点击表格可以修改数据



Qt::ToolTipRole 就是你鼠标移动到表格有提示显示

Qt::WhatsThisRole 这个属性必须你窗口有问号才能显示出来，我觉得没啥用  
这是表格模型组织数据成表格形式，用表格视图来显示数据。

我们用显示其它形状的视图来显示表格模型组织的数据

```
#include <QListView>
#include <QTreeView>
namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

    QStandardItemModel m_model;//模型对象，到时候用来接收组织的数据
    QTableView m_tableview;//显示模型里面的数据
    QListView m_listview;//列表视图
    QTreeView m_treeview;//树状视图
```

加入列表形状显示的视图和树状形状显示的视图

```
void Widget::initview()
{
    m_tableview.setParent(this);//这个表格
    m_tableview.move(10,10);//表格在父窗口的
    m_tableview.resize(300,100);//表格大小

    m_listview.setParent(this);
    m_listview.move(10,120);
    m_listview.resize(300,100);

    m_treeview.setParent(this);
    m_treeview.move(10,220);
    m_treeview.resize(300,100);
}
```

视图布局

```

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

    initmodel();
    initview();

    m_tableview.setModel(&m_model); //用视图的setModel去连接模型
    m_listview.setModel(&m_model); //用列表视图来连接表格模型
    m_treeview.setModel(&m_model); //用树状视图来连接表格模型
}

```

列表视图和树状视图连接表格模型组织的矩阵格子形态的数据

|       |       |
|-------|-------|
| 1     | 2     |
| Adata | Bdata |
| Cdata |       |

表格视图显示表格形式组织的数据没有问题

|       |       |
|-------|-------|
| Adata | Cdata |
| 1     | 2     |
| Adata | Bdata |

列表视图显示表格形式的数据就有问题了，因为列表值显示了表格数据的第 0 列

树状视图怎么能完整显示表格的数据呢？树状视图和表格视图差别不大？

我新增加一个数据项你就知道了

新增的数据项

```

QStandardItem *itemA = new QStandardItem(); //创建数据项A
QStandardItem *itemB = new QStandardItem(); //创建数据项B
QStandardItem *itemC = new QStandardItem(); //创建数据项C
QStandardItem *child = new QStandardItem(); //创建数据项的子数据项

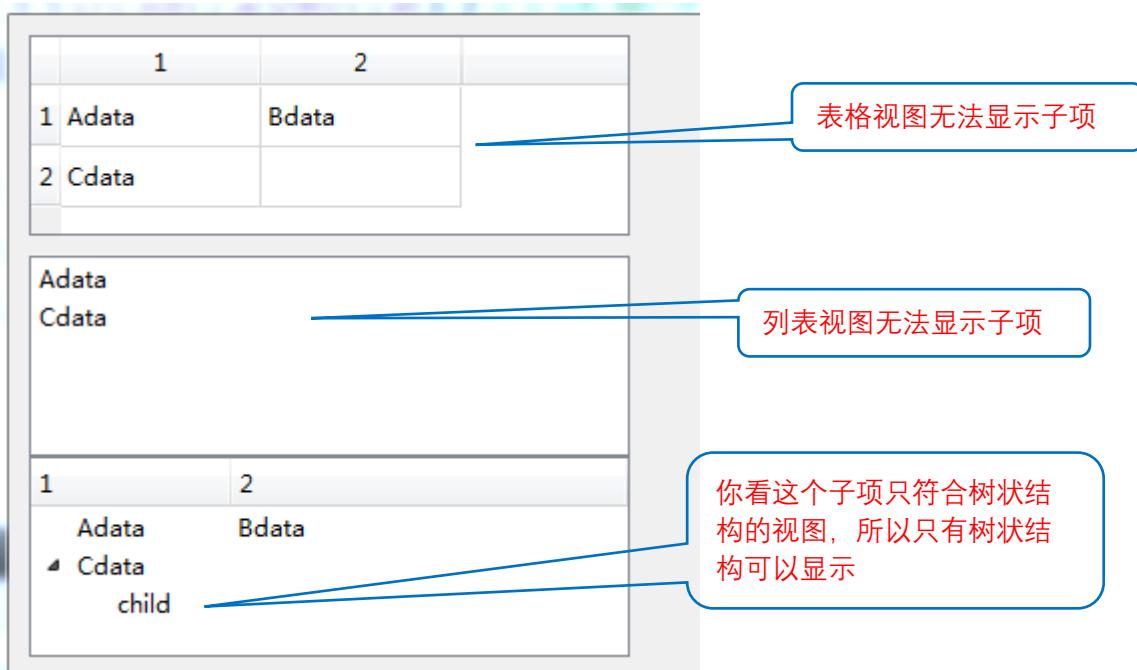
itemB->setData(200, Qt::ToolTipRole);
itemB->setData(2.75, Qt::WhatsThisRole);

itemC->setData("Cdata", Qt::DisplayRole); //给每个数据项加数据显示
itemC->setData(300, Qt::ToolTipRole); //把新增的数据项放在 itemC
itemC->setData(3.75, Qt::WhatsThisRole); //数据项下面，成为子项

child->setData("child", Qt::DisplayRole); //这是子数据项
itemC->setChild(0, 0, child); //child在itemC下面的第0行第0列

```

//给每个数据项加数据显示  
把新增的数据项放在 itemC  
数据项下面，成为子项



所以 QT 里面的默认数据模型只能实现少数的视图显示。

如果你是树状结构模型组织的数据就要用树状视图显示

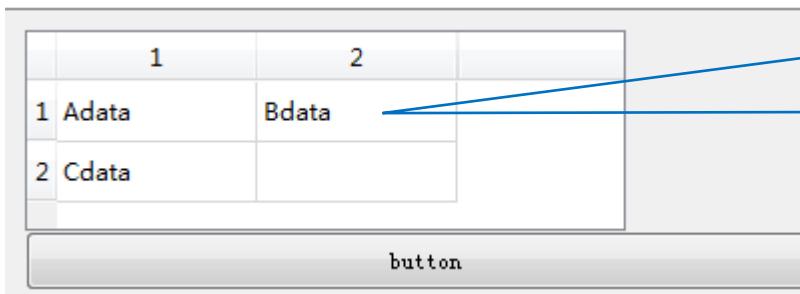
如果你是表格型结构组织的数据就要用表格视图显示

如果你是列表形状组织的数据就要用列表视图来显示

这样才符合模型和视图显示规范。

我们下面做自定模型视图类就不会被这些规范锁住。

## 模型视图委托机制，就是界面上修改的参数自动传入模型类



我们要实现的功能  
就是点击这个表格  
里面的编辑框就会  
触发模型里面的函  
数执行

我们就用上一节的模型和表格程序来修改

```
class Widget : public QWidget
{
    Q_OBJECT

    QStandardItemModel m_model; //模型对象，到时候用来接收组织的数据
    QTableView m_tableview; //显示模型里面的数据
    QPushButton button;

    void initmodel(); //组织数据的函数
    void initview(); //显示数据的函数
public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();

private:
    Ui::Widget *ui;
private slots:
    void buttonclicked();
```

在头文件里面加入按钮

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

    initmodel();
    initview();

    m_tableview.setModel(&m_model); //用视图的setModel去连接模型，把模型
    connect(&button, SIGNAL(clicked()), this, SLOT(buttonclicked()));
}
```

在头文件里面加入按钮触发槽函数

我们已经将视图和模型连接起来了

```

void Widget::initmodel()
{
    QStandardItem *root = m_model.invisibleRootItem(); // invisibleRootItem()方法返回一个空的QStandardItem对象，作为模型的根节点

    QStandardItem *itemA = new QStandardItem(); // 创建数据项A
    QStandardItem *itemB = new QStandardItem(); // 创建数据项B
    QStandardItem *itemC = new QStandardItem(); // 创建数据项C

    将 model 地址赋值给 root 变量，因为模型是 QStandardItemModel，是接收 QStandardItem 类对象数据的模型

    itemA->setData("Adata", Qt::DisplayRole);
    // Qt::DisplayRole 属性就是让 Adata 数据作为提示性质的字符串显示
    itemA->setData(100, Qt::ToolTipRole);
    // Qt::ToolTipRole 属性是将数字放在悬浮框显示
    itemA->setData(1.75, Qt::WhatsThisRole);

    itemB->setData("Bdata", Qt::DisplayRole); // 给每个数据项加数据显示
    itemB->setData(200, Qt::ToolTipRole);
    itemB->setData(2.75, Qt::WhatsThisRole);

    itemC->setData("Cdata", Qt::DisplayRole); // 给每个数据项加数据显示
    itemC->setData(300, Qt::ToolTipRole);
    itemC->setData(3.75, Qt::WhatsThisRole);

    root->setChild(0, 0, itemA);
    root->setChild(0, 1, itemB);
    root->setChild(1, 0, itemC);

    给StandardItem 初始化数据后，将StandardItem 对象赋值给 m_model 的 root 指针，这样 m_model 就有了StandardItem 对象的数据
}

```

```

void Widget::buttonclicked()
{
    qDebug() << "model data";

    QModelIndex index;
    index = m_model.index(0, 0, QModelIndex());
    QString text = index.data(Qt::DisplayRole).toString();
    qDebug() << text;

    index = m_model.index(0, 1, QModelIndex());
    text = index.data(Qt::DisplayRole).toString();
    qDebug() << text;

    index = m_model.index(1, 0, QModelIndex());
    text = index.data(Qt::DisplayRole).toString();
    qDebug() << text;
}

```

当界面表格数据修改后，会自动修改模型 m\_model 里面的数据，也就间接修改了 ItemA/B/C 对象的数据，你按下按钮，槽函数执行后，就可以获取修改的数据

然后调用最先定义的 m\_model 函数来实现，获取表格的数据，这里 index(参数 1, 参数 2, 参数 3)  
参数 1, 参数 2 都是填入获取几行几列数据

如果你不想一行一行的去写 `m_model.index` 获取表格的数据，就用循环获取

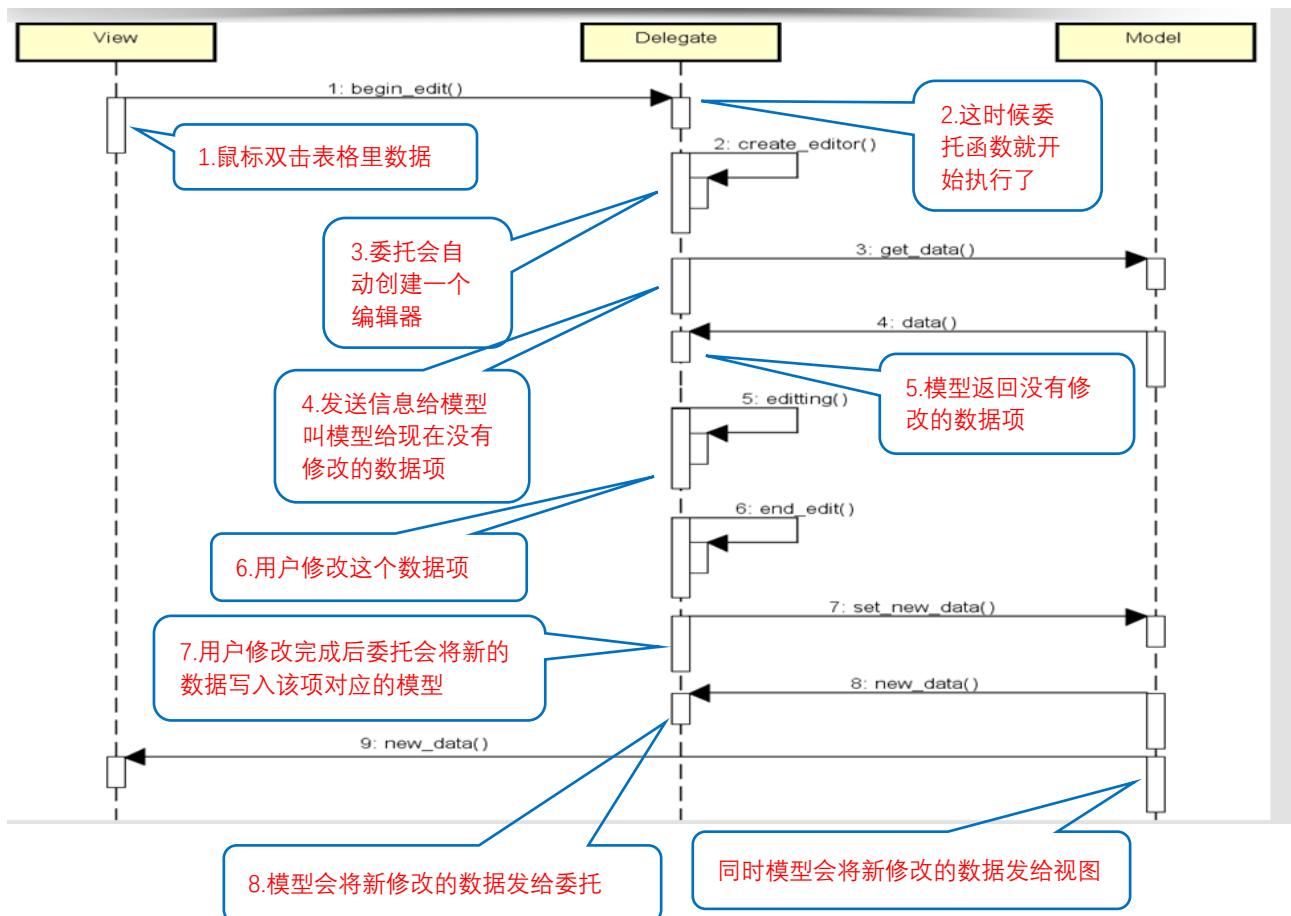
```
void Widget::buttonclicked()
{
    qDebug() << "model data";

    for(int i=0; i<m_model.rowCount(); i++)
    {
        for(int j=0; j<m_model.columnCount(); j++)
        {
            QModelIndex index = m_model.index(i, j, QModelIndex());
            QString text = index.data(Qt::DisplayRole).toString();

            qDebug() << text;
        }
    }
}
```

你发现没有，我们要点击按钮，才会去执行槽函数将视图修改的数据传递给模型。

下面我们要实现在视图表格里面点击的时候就迅速修改了模型里面的数据



```

1 #ifndef SUBWEITUO_H
2 #define SUBWEITUO_H
3
4 #include <QWidget>
5 #include <QStyledItemDelegate>/> //委托类
6
7 class subweituo : public QStyledItemDelegate
8 //这里要继承至QStyledItemDelegate, 这样这个类才能传入有QStyledItemDelegate形参的函数
9 {
10     Q_OBJECT
11 public:
12     explicit subweituo(QObject *parent = 0);
13     QWidget* createEditor(QWidget *parent, const QStyleOptionViewItem &option, const QModelIndex &index) const;
14     void updateEditorGeometry(QWidget *editor, const QStyleOptionViewItem &option, const QModelIndex &index) const;
15     void setEditorData(QWidget *editor, const QModelIndex &index) const;
16     void setModelData(QWidget *editor, QAbstractItemModel *model, const QModelIndex &index) const;
17 signals:
18 
```

我们新建一个委托类，来继承 QT  
库里面的 QStyledItemDelegate 类

这些委托类函数不是我们创建的，而是重写的 QStyledItemDelegate 类，也就  
是说这些函数在委托类里面本来就有，只是我们要修改它的实现

```

#include "subweituo.h"
#include <QDebug>

subweituo::subweituo(QObject *parent) : QStyledItemDelegate(parent)
{
}

QWidget* subweituo::createEditor(QWidget *parent, const QStyleOptionViewItem &option, const QModelIndex &index) const
{
    qDebug() << "createEditor";
    return QStyledItemDelegate::createEditor(parent, option, index);
}

void subweituo::updateEditorGeometry(QWidget *editor, const QStyleOptionViewItem &option, const QModelIndex &index) const
{
    qDebug() << "updateEditorGeometry";
    QStyledItemDelegate::updateEditorGeometry(editor, option, index);
}

void subweituo::setEditorData(QWidget *editor, const QModelIndex &index) const
{
    qDebug() << "setEditorData";
    QStyledItemDelegate::setEditorData(editor, index);
}

void subweituo::setModelData(QWidget *editor, QAbstractItemModel *model, const QModelIndex &index) const
{
    qDebug() << "setModelData";
    return QStyledItemDelegate::setModelData(editor, model, index);
}

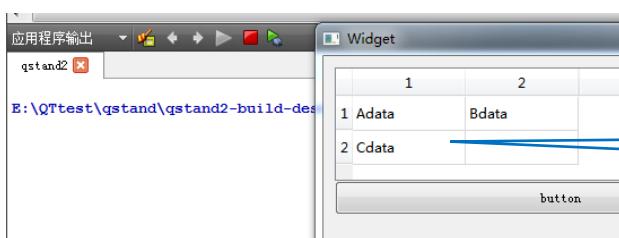
```

这个 createEditor 是在编辑表格  
的时候自动执行，创建个编辑器

这个函数是让创建  
的编辑器窗口自动  
适应表格大小

这个函数是在修改  
表格参数的时候就  
会执行，其实你双  
击表格的时候前面  
3 个函数都执行了

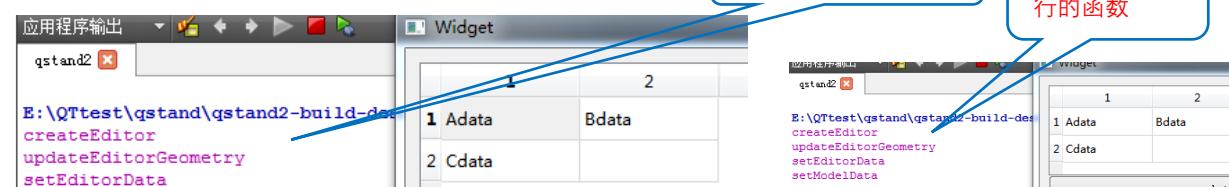
这个函数是修改完表  
格后鼠标点击其它地  
方会自动执行



这是表格什么都没有点

点击表格执行的函数

表格点击完执  
行的函数



我们想在修改表格数据时，自动就把表格对应底层的模型数据修改了，怎么做呢？那就  
要自定义委托类，其实就是要重写`<QStyledItemDelegate>`里面的几个函数

1. 重写 `createEditor` 函数
2. 重写 `updateEditorGeometry` 函数
3. 重写 `setEditorData` 函数
4. 重写 `setModelData` 函数
5. 重写 `paint` 函数(可选项)

还是用上一节模式视图委托机制的表格范例来修改  
未完待续……

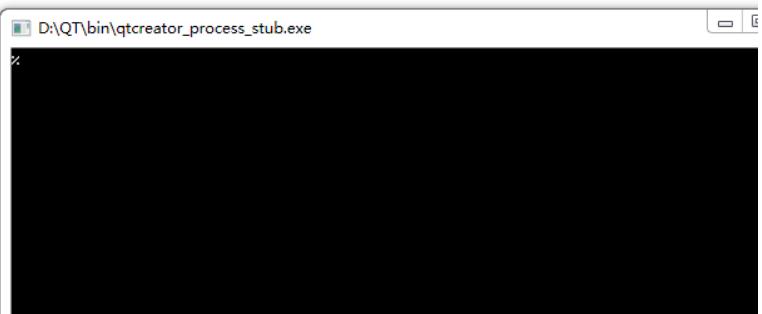
## QByteArray 的使用

QByteArray 常使用在串口接受数据应用当中，所以这里要重点讲解下

```
#include <QDebug>
#include <QByteArray>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QByteArray ba; // 定义 QByteArray 对象
    ba.resize(1); // 给 QByteArray 对象分配一个字节空间
    ba[0] = 0x25;

    qDebug() << ba[0];
    return a.exec();
}
```

如果想给 ba 赋值，必须先申请 ba 的大小



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

    QByteArray ba;
    ba.resize(1); // 给 QByteArray 对象分配一个字节空间
    ba[0] = 0x25;

    qDebug() << ba.toHex(); // 要求 QByteArray 的对象以16进制输出
    return a.exec();
}
```

必须用 `toHex` 来转换 QByteArray 里面的字符，转成 16 进制输出



```

int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);

    QBYTEARRAY ba;
    ba.resize(1); //给QByteArray对象分配一个字节空间
    ba[0] = 0x25;
    ba[1] = 0x1f; //如果定义了2个数组

    qDebug() << ba.toHex(); //要求QByteArray的对象以16进制输出
    //那么 tohex 就将两个数组的 16 进制拼接起来一起输出
    return a.exec();

```

QByteArray 没有办法用下标的方式指定某个数组元素输出 16 进制数吗？

```

int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);

    QBYTEARRAY ba;
    ba.resize(1); //给QByteArray对象分配一个字节空间
    ba[0] = 0x25;
    ba[1] = 0x1f; //只有用QString 的方式才能接受指定下标 QByteArray 转换出来的数据

    QString strhex = QString::number(ba.at(1), 16);
    //用at方式读取QByteArray里面的数据最快，省去了拷贝过程
    qDebug() << strhex; //这里指定数组元素转换出来以16进制数赋值给QString

    strhex = QString::number(ba.at(1), 10);
    qDebug() << strhex;

    strhex = QString::number(ba.at(0), 10);
    qDebug() << strhex;

```

但是还是有个问题 QString 虽然接受了数据，但是数据是以字符串方式显示的 16 进制。这种情况就要查找其它方式在转换一道，然后放给指定 int, long 变量。

```

int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);

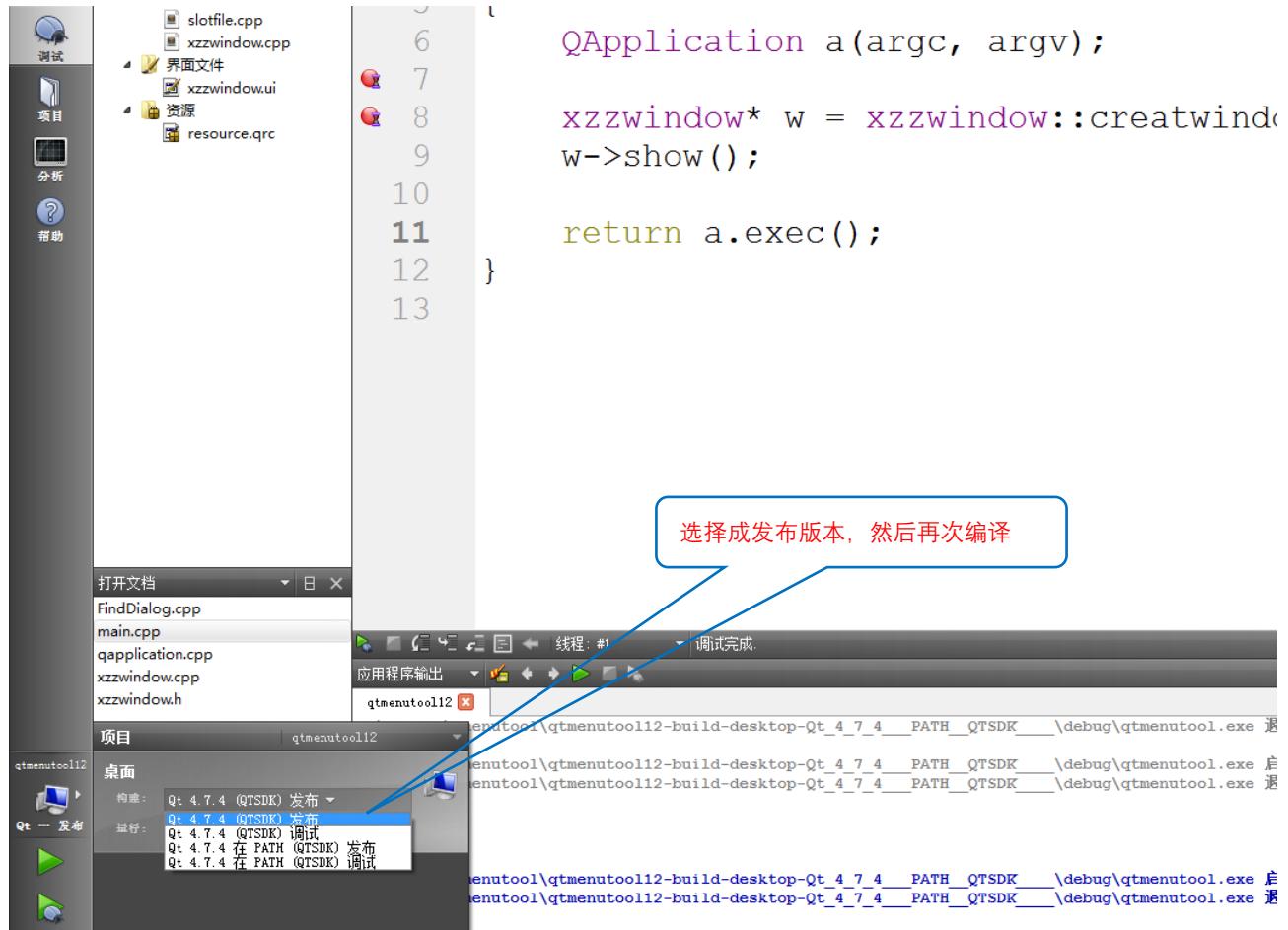
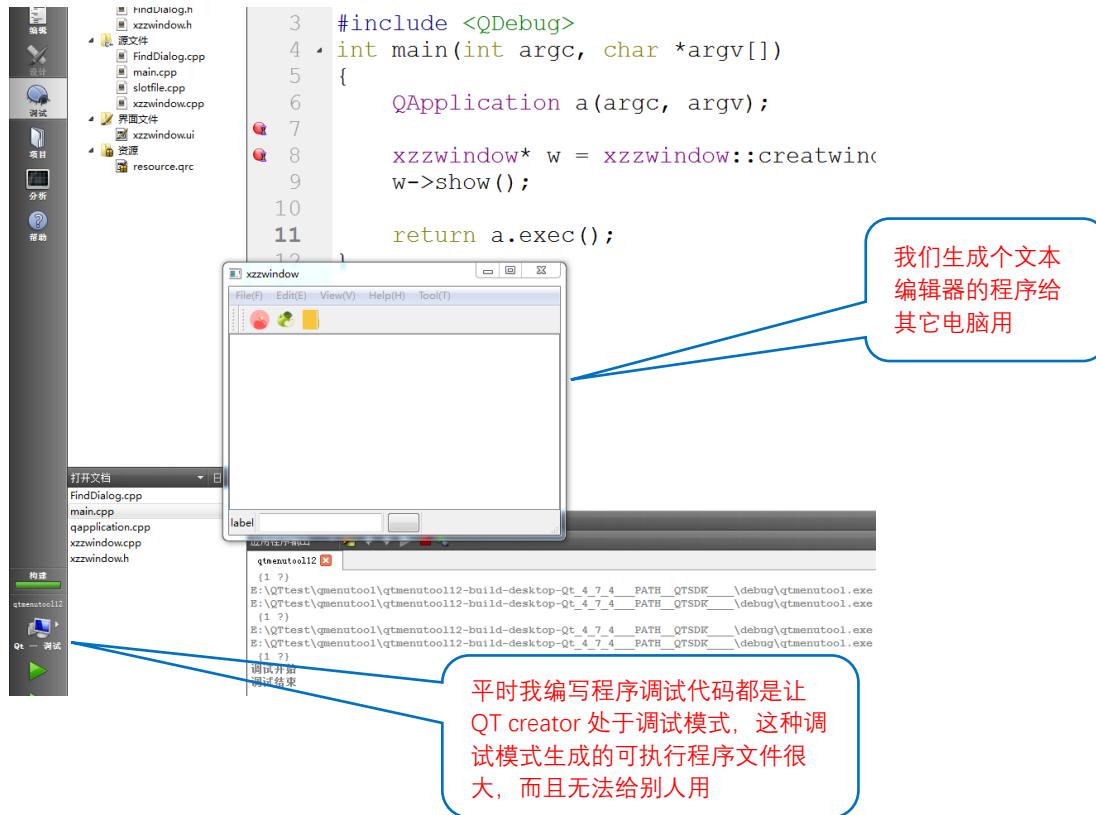
    QBYTEARRAY text = QBYTEARRAY::fromHex("517420697320677265617421");
    qDebug() << text.data();
    //QBYTEARRAY::data表示可以读取QBYTEARRAY里面的值，也可以向QBYTEARRAY里面写

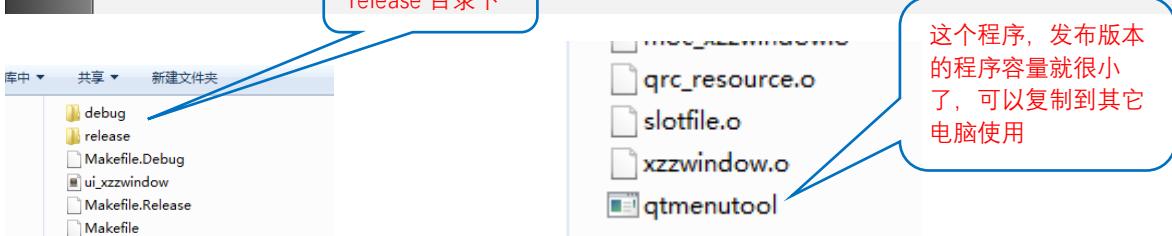
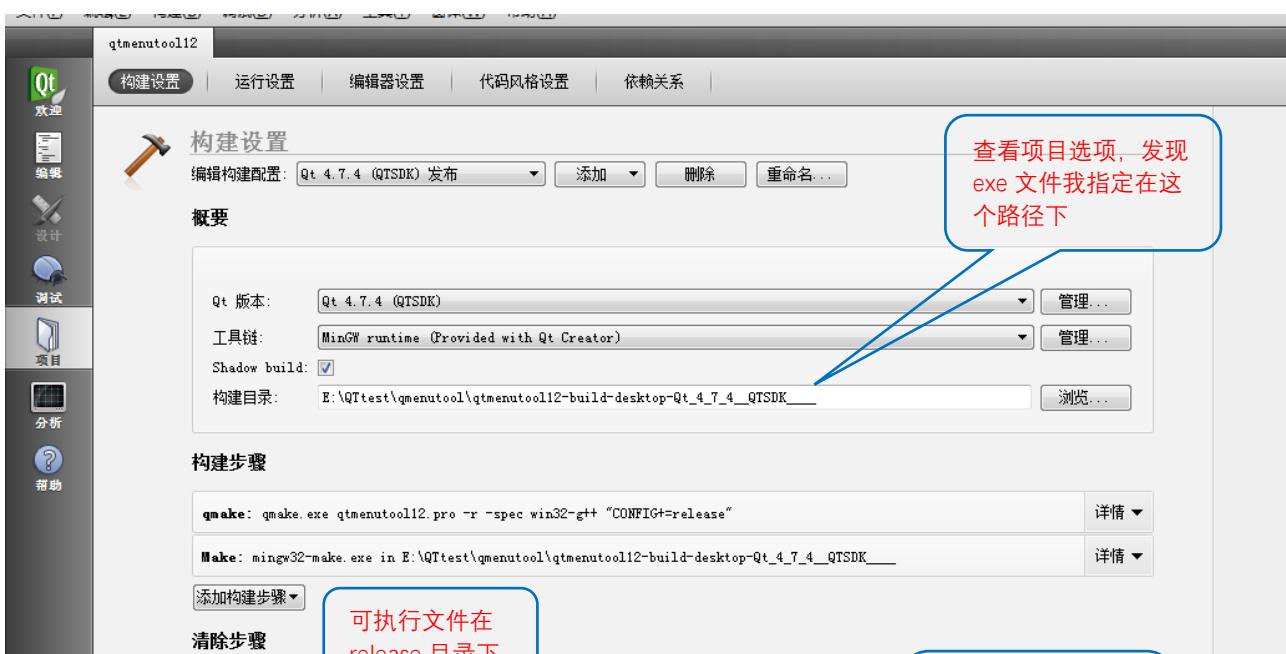
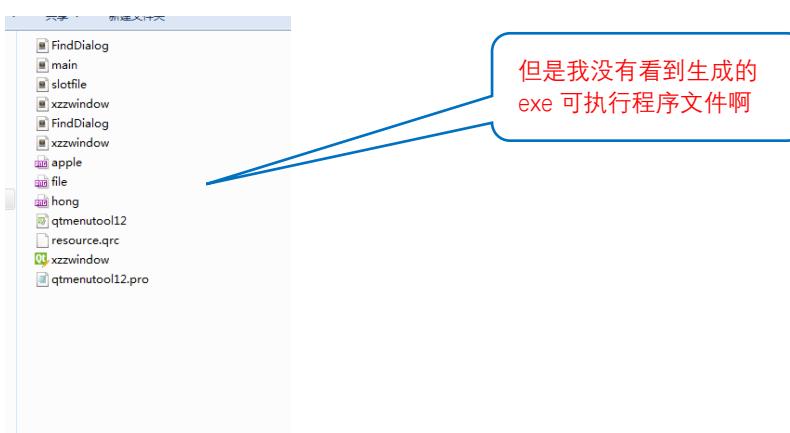
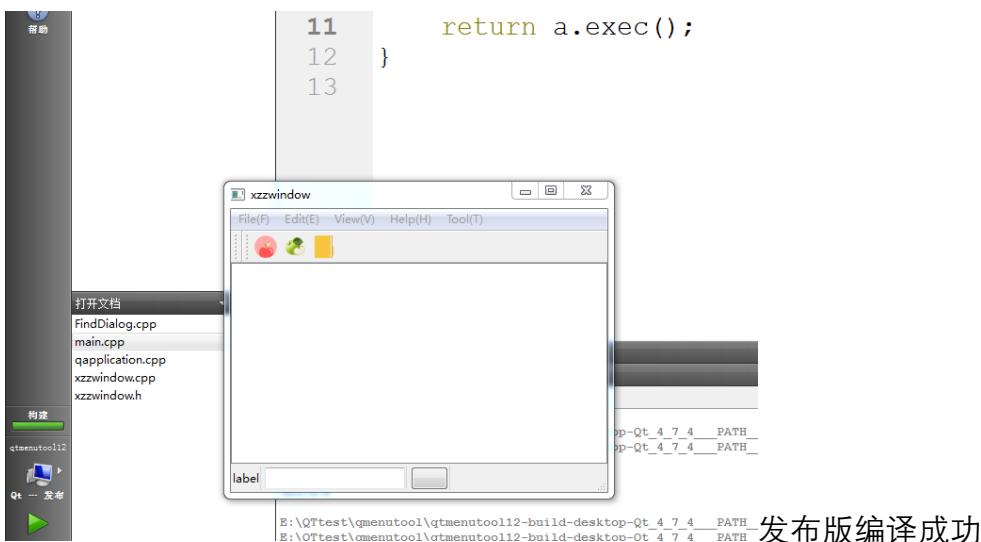
    return a.exec();
}

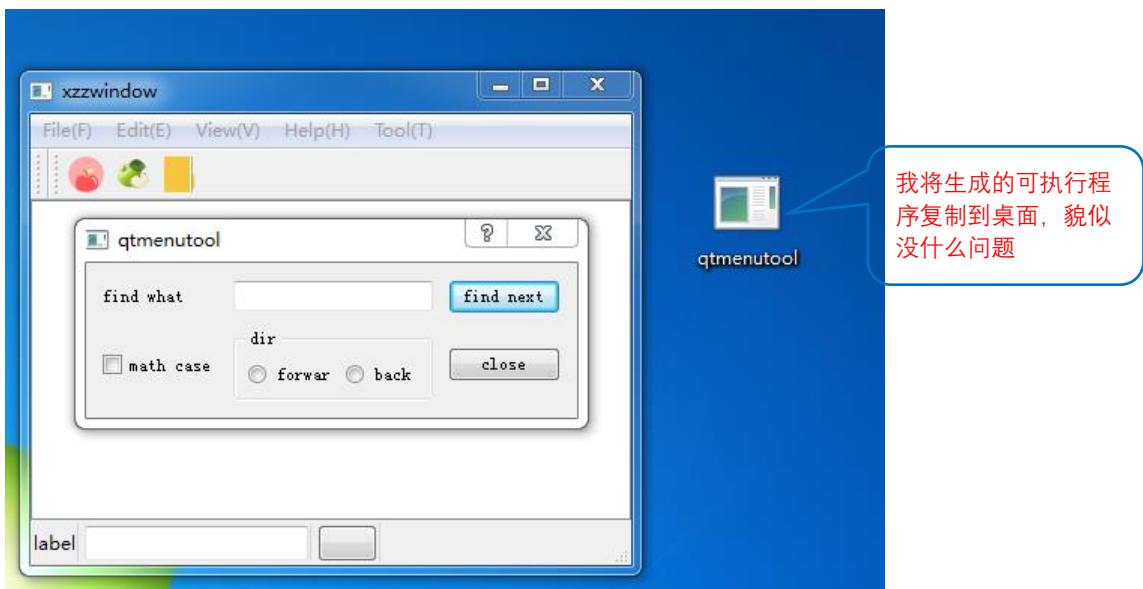
```

# QT Creator 制作可执行程序在其它 windows 电脑上运行

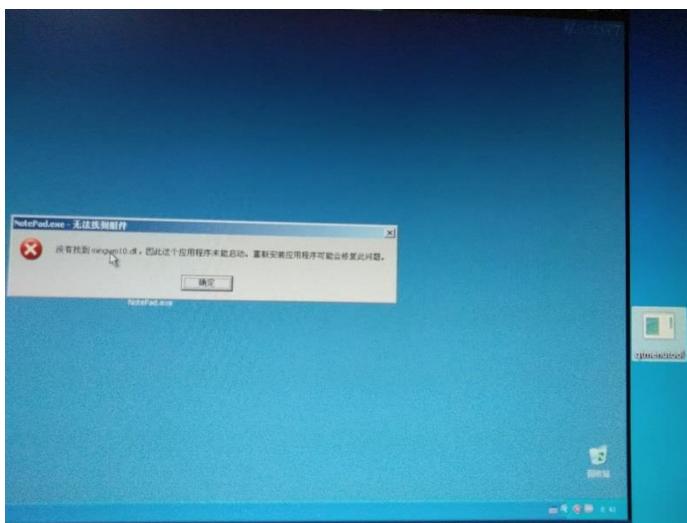
专业术语叫做应用程序打包与发布





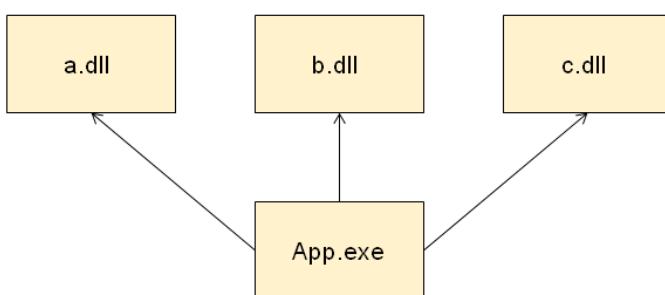


这是因为我的这台电脑有了 QT 程序运行时需要的各种依赖库和环境。  
如果换成其它电脑，你就复制一个 exe 文件去给别人，绝对运行不起，直接报错。



这是为什么呢？

- 发布程序时必须保证所有的依赖库都存在



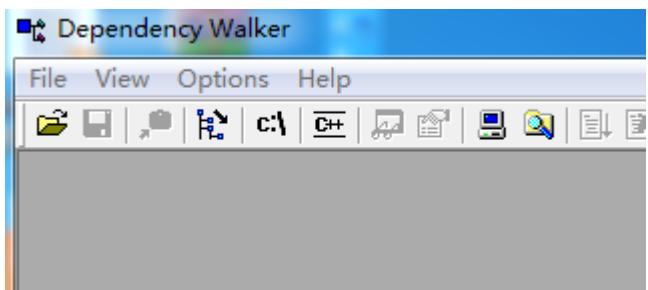
Windows 和 linux 一样，应用程序的运行需要依赖一些库里面的函数，windows 动态库是.dll，linux 动态库是.so

所以我们在把 exec 文件发给别人时，我们要用工具先查看下 exec 文件需要哪些库？

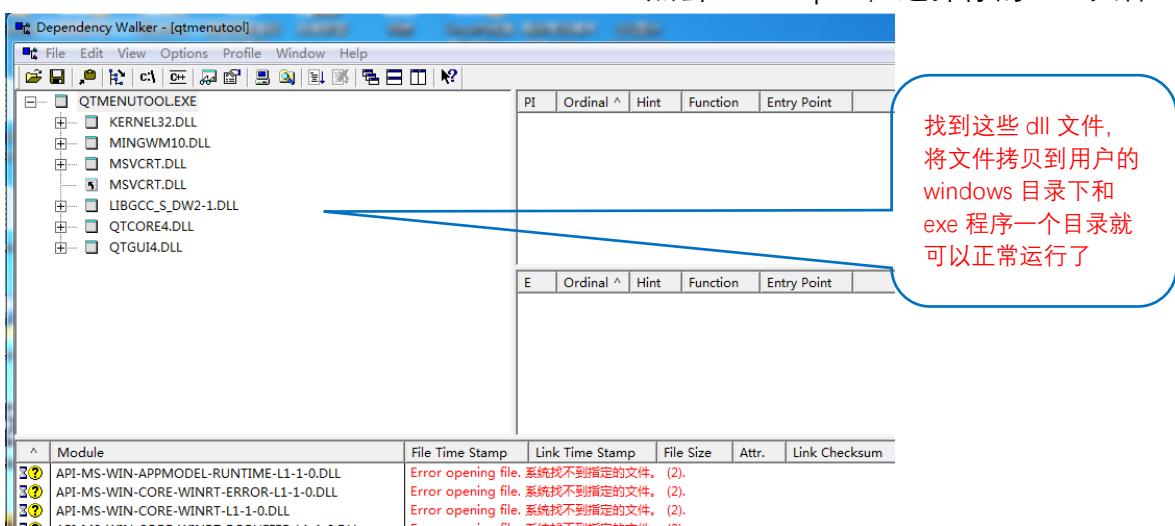
所以我要去下载个 windows 常用的应用程序分析工具 depends  
<http://www.dependencywalker.com/>

Download the latest version here:

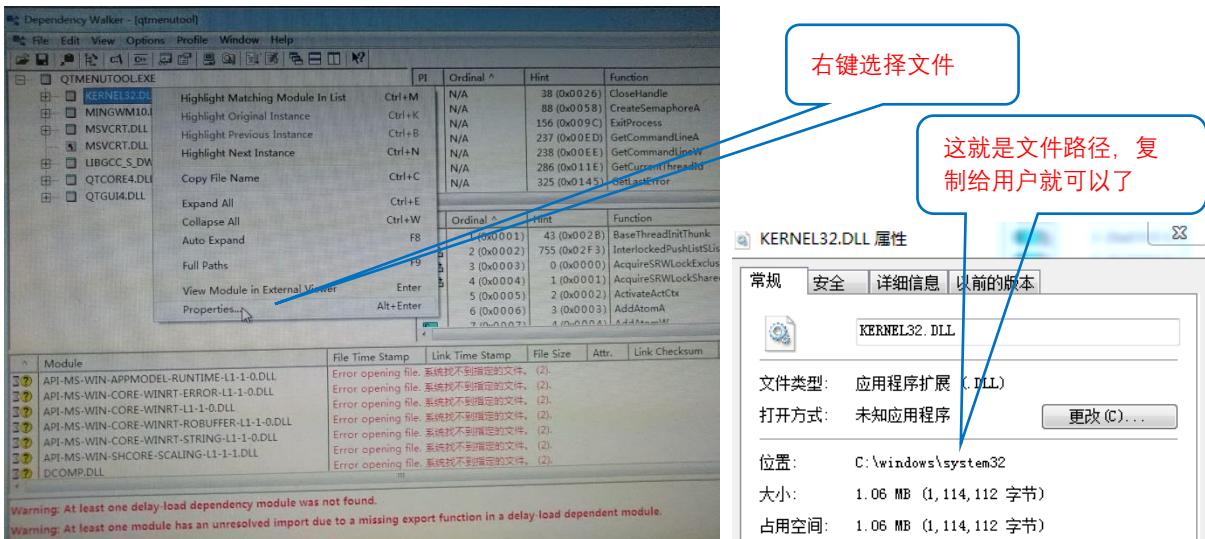
-  [Download Version 2.2.6000 for x86 \(Windows 95 / 98 / Me / NT / 2000 / XP / 2003 / Vista / 7 / 8\) \[610k\]](#)
-  [Download Version 2.2.6000 for x64 \[468k\]](#)



点击 file->open, 选择你的 exe 文件



如何找到这些 dll 文件呢 ?



以上就是 QT 开发打包程序给用户全过程。

再打包 exe 程序给用户前可以把工程复制到 linux 下编译，看看有没有警告之类的错误，或者潜在 BUG，因为 windows 下 QT 程序检查不是很严格。

## 把 windows 的 QT 工程复制粘贴到 linux 下编译使用遇到的问题

```

4 int main(int argc, char *argv[])
5 {
6     QApplication a(argc, argv);
7
8     xzzwindow* w = xzzwindow::creatwindow();
9     w->show();
10
11     return a.exec();
12 }
13

```

Compile Output

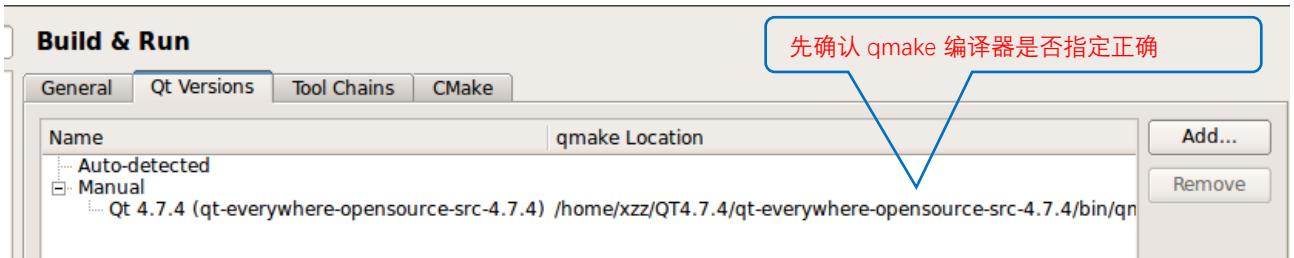
```

Error while building project qtmenutool12 (target: 桌面)
When executing build step 'qmake'
Error while building project qtmenutool12 (target: 桌面)
When executing build step 'qmake'
Error while building project qtmenutool12 (target: 桌面)
When executing build step 'qmake'
Error while building project qtmenutool12 (target: 桌面)
When executing build step 'qmake'
Error while building project qtmenutool12 (target: 桌面)
When executing build step 'qmake'
Error while building project qtmenutool12 (target: 桌面)
When executing build step 'qmake'

```

编译的时候发现 qmake 之类的错误

这是因为 windows 的 qtcreator 工程路径和 linux 的 qtcreator 工程路径不一样



因为 QT 是 linux 最先发行的，所以在 linux 下对 QT 的 C++ 语法编译检查要严格一些

```

bool xzzwindow::initstatus()
{
    QStatusBar* sb = statusBar(); // 'x' 'I-A,
    QLabel* l = new QLabel("label"); // 'x' '+eC
    QLineEdit* le = new QLineEdit(); // 'x' I-A+3j
    QPushButton* button = new QPushButton(); // 'x' "A

    sb->addWidget(l);
    sb->addWidget(le);
    sb->addWidget(button);
}

146 }
147
148 bool xzzwindow::initstatus()
149 {
150     QStatusBar* sb = statusBar(); // 'x' 'I-A,
151     QLabel* l = new QLabel("label"); // 'x' '+eC
152     QLineEdit* le = new QLineEdit(); // 'x' I-A+3j
153     QPushButton* button = new QPushButton(); // 'x' "A

154     sb->addWidget(l);
155     sb->addWidget(le);
156     sb->addWidget(button);
157     return true;
158 }

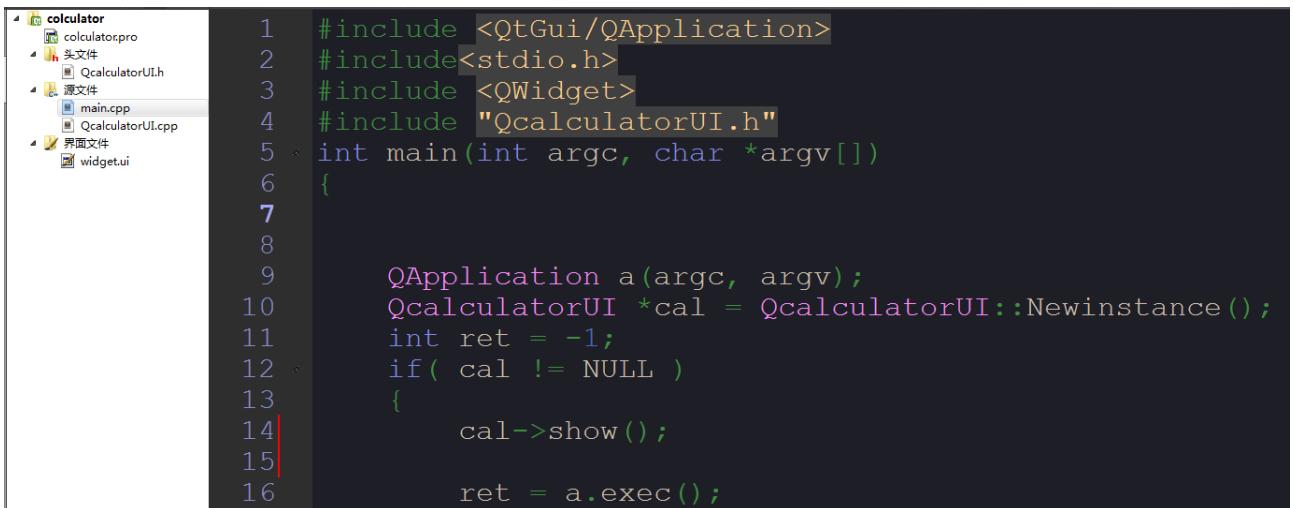
161 bool xzzwindow::initxzzEdit()
162 {
    bool ret = true;
163
    if(&ptedit == NULL)
        ret = false;
    else
    {
        ptedit.setParent(this); // 'x' 'I-A+3j EeEeEe ' 'U this iEeC.. ' 'O
        setCentralWidget(&ptedit); // 'x' 'I-A+3j EeOAOU .. ' 'U ODN
        connect(&ptedit, SIGNAL(textChanged()), this, SLOT(onTextChanged())
        // 'x' 'I-A+3j OiAiAeUEY.. ' 'EIDAOAtextChanged()
    }
}

```

## Qt Creator 设置工程的一些字体大小和其它属性

### 修改编辑框背景和字体颜色





The screenshot shows a Qt-based IDE interface. On the left, there is a tree view of the project structure:

- calculator
- calculator.pro
- 头文件
- QcalculatorUI.h
- 源文件
- main.cpp
- QcalculatorUI.cpp
- 界面文件
- widget.ui

The main window displays the content of main.cpp:

```
1 #include <QtGui/QApplication>
2 #include <stdio.h>
3 #include <QWidget>
4 #include "QcalculatorUI.h"
5 int main(int argc, char *argv[])
6 {
7
8
9     QApplication a(argc, argv);
10    QcalculatorUI *cal = QcalculatorUI::Newinstance();
11    int ret = -1;
12    if( cal != NULL )
13    {
14        cal->show();
15
16        ret = a.exec();
```

修改成功