

Qt 教案

在此先感谢各位老师的无私奉献：

1. 爱编程的王廷胡，王老师的资料分享
2. 爱编程的贾立君，贾老师的资料分享
3. 爱编程的朱荣，朱老师的资料分享
4. 爱编程的苏丙榅，苏老师的资料分享

Qt 教案

一、Qt 入门

- 1.Qt 概述
 - 1.1 什么是 Qt
 - 1.2 Qt 的特点
 - 1.3 Qt 中的模块
 - 1.4 Qt 案例
2. 安装
 - 2.1 安装包下载
 - 2.2 安装 (windows)
 - 2.3 环境变量设置
 - 2.4 Qt Creator
3. 第一个 Qt 项目
 - 3.1 创建项目
 - 3.2 项目文件 (.pro)
 - 3.3 main.cpp
 - 3.4 mainwindow.ui
 - 3.5 mainwindow.h
 - 3.6 mainwindow.cpp
4. Qt 中的窗口类
 - 4.1 基础窗口类
 - 4.2 窗口的显示
5. 坐标体系
 - 5.1 窗口的坐标原点
 - 5.2 窗口的相对坐标
6. 内存回收

二、Qt中的基础数据类型

1. 基础类型
2. log 输出
 - 2.1 在调试窗口中输入日志
 - 2.2 在终端窗口中输出日志
3. 字符串类型
 - 3.1 QByteArray
 - 3.1.1 构造函数:
 - 3.1.2 数据操作:
 - 3.1.3 子字符串查找和判断:
 - 3.1.4 遍历:
 - 3.1.5 查看字节数:
 - 3.1.6 类型转换:
 - 3.2 QString
 - 3.2.1 构造函数:
 - 3.2.2 数据操作:
 - 3.2.3 子字符串查找和判断:
 - 3.2.4 遍历:
 - 3.2.5 查看字节数:

3.2.6 类型转换:

3.2.7 字符串格式:

4. 位置和尺寸

- 4.1 QPoint
- 4.2 QLine
- 4.3 QSize
- 4.4 QRect

三、Qt中的信号槽

1. 信号和槽概述

- 1.1 信号的本质
- 1.2 槽的本质
- 1.3 信号和槽的关系

2. 标准信号槽使用

- 2.1 标准信号 / 槽
- 2.2 使用

3. 自定义信号槽使用

- 3.1 自定义信号
- 3.2 自定义槽

4. 信号槽拓展

- 4.1 信号槽使用拓展
- 4.2 信号槽的连接方式
 - 1. Qt5 的连接方式
 - 2. Qt4 的连接方式
 - 3. 应用举例

5. Lambda 表达式

- 5.1 语法格式
- 5.2 定义和调用

四、Qt定时器类 QTimer

- 1. public/slot function
- 2. signals
- 3. static public function
- 4. 定时器使用举例

五、Qt中的基础窗口类

1. QWidget

- 1.1 设置父对象
- 1.2 窗口位置
- 1.3 窗口尺寸
- 1.4 窗口标题和图标
- 1.5 信号
- 1.6 槽函数

2. QDialog

- 2.1 常用 API
- 2.2 常用使用方法
 - 2.2.1 关于对话框窗口类的操作
 - 2.2.2 根据用户针对对话框窗口的按钮操作，进行相应的逻辑处理。

3. QDialog 的子类（标准对话框）

- 3.1 QMessageBox
 - 3.1.1 API - 静态函数
 - 3.1.2 测试代码
- 3.2 QFileDialog
 - 3.2.1 API - 静态函数
 - 3.2.2 测试代码
 - 3.2.2.1 打开一个已存在的本地目录
 - 3.2.2.2 打开一个本地文件
 - 3.2.2.3 打开多个本地文件
- 3.3 QFontDialog
 - 3.3.1 QFont 字体类
 - 3.3.2 QFontDialog 类的静态 API

- 3.3.3 测试代码
 - 3.4 QColorDialog
 - 3.4.1 颜色类 QColor
 - 3.4.2 QFontDialog 类的静态 API
 - 3.4.3 测试代码
 - 3.5 QInputDialog
 - 3.5.1 API - 静态函数
 - 3.5.2 测试代码
 - 3.5.2.1 整形输入框
 - 3.5.2.2 浮点型输入框
 - 3.5.2.3 带下拉菜单的输入框
 - 3.5.2.4 多行字符串输入框
 - 3.5.2.5 单行字符串输入框
 - 3.6 QProgressDialog
 - 3.6.1 常用 API
 - 3.6.2 测试代码
4. QMainWindow
- 4.1 菜单栏
 - 4.1.1 添加菜单项
 - 4.1.2 常用的添加方式
 - 4.1.3 通过代码的方式添加菜单或者菜单项
 - 4.1.4 菜单项 QAction 事件的处理
 - 4.2 工具栏
 - 4.2.1 添加工具按钮
 - 4.2.2 工具栏的属性设置
 - 4.3 状态栏
 - 4.4 停靠窗口
5. 资源文件 .qrc
- 1. 使用资源编辑器打开资源文件
 - 2. 给资源添加前缀
 - 3. 添加文件
 - 4. 如何在程序中使用资源文件中的图片

六、Qt窗口布局

- 1. 布局的样式
- 2. 在 UI 窗口中设置布局
 - 2.1 方式 1
 - 2.2 方式 2
 - 1. 首先需要从工具栏中拖拽一个容器类型的窗口到 UI 界面上
 - 2. 将要布局的子控件放到这个 QWidget 中
 - 3. 对这个 QWidget 进行布局
 - 2.3 弹簧的使用
 - 2.4 布局属性设置
 - 2.5 布局的注意事项
- 3. 通过 API 设置布局
 - 3.1 QLayout
 - 3.2 QHBoxLayout
 - 3.3 QVBoxLayout
 - 3.4 QGridLayout

七、在Qt窗口中添加右键菜单

- 1. 基于鼠标事件实现
 - 1.1 实现思路
 - 1.2 代码实现
- 2. 基于窗口的菜单策略实现
 - 2.1 Qt::DefaultContextMenu
 - 2.2 Qt::ActionsContextMenu
 - 2.3 Qt::CustomContextMenu

八、Qt中按钮类型的控件

- 1. 按钮基类 QPushButton

- 1.1 标题和图标
- 1.2 按钮的 Check 属性
- 1.3 信号
- 1.4 槽函数
- 2. QPushButton
 - 2.1 常用 API
 - 2.2 按钮的使用
- 3. QToolButton
 - 3.1 常用 API
 - 3.2 按钮的使用
- 4. QRadioButton
 - 4.1 常用 API
 - 4.2 按钮的使用
- 5. QCheckBox
 - 5.1 常用 API
 - 5.2 按钮的使用

九、Qt中容器类型的控件

- 1. QWidget
- 2. Frame
 - 2.1 相关 API
 - 2.2 属性设置
- 3. Group Box
 - 3.1 相关 API
 - 3.2 属性设置
- 4. Scroll Area
 - 4.1 相关 API
 - 4.2 属性设置
 - 4.3 窗口的动态添加和删除
- 5. Tool Box
 - 5.1 相关 API
 - 5.2 属性设置
- 6. Tab Widget
 - 6.1 相关 API
 - 6.2 属性设置
 - 6.3 控件使用
- 7. Stacked Widget
 - 7.1 相关 API
 - 7.2 属性设置
 - 7.3 控件使用

十、Qt中多线程的使用

- 1. 线程类 QThread
 - 1.1 常用共用成员函数
 - 1.2 信号槽
 - 1.3 静态函数
 - 1.4 任务处理函数
- 2. 使用方式 1
 - 2.1 操作步骤
 - 2.2 示例代码
- 3. 使用方式 2
 - 3.1 操作步骤
 - 3.2 示例代码

十一、Qt中线程池的使用

- 1.线程池的原理
- 2. QRunnable
- 3. QThreadPool

十二、基于TCP的Qt网络通信

- 1. QTcpServer
 - 1.1 公共成员函数

- 1.2 信号
- 2. QTcpSocket
 - 2.1 公共成员函数
 - 2.2 信号
- 3. 通信流程
 - 3.1 服务器端
 - 3.1.1 通信流程
 - 3.1.2 代码片段
 - 3.2 客户端
 - 3.2.1 通信流程
 - 3.2.2 代码片段

十二、Qt事件之事件处理器

- 1. 事件
- 2. 事件处理器函数
 - 2.1 鼠标事件
 - 2.2 键盘事件
 - 2.3 窗口重绘事件
 - 2.4 窗口关闭事件
 - 2.5 重置窗口大小事件
- 3. 重写事件处理器函数
 - 3.1 头文件
 - 3.2 源文件
 - 3.3 效果
- 4. 自定义按钮
 - 4.1 添加子类
 - 4.2 使用自定控件
 - 4.3 设置图片

十三、Qt事件之事件分发器

- 1. QEvent
- 2. 事件分发器

十四、Qt事件之事件过滤器

- 1. 事件过滤器
- 2. 事件过滤器的使用

十五、Qt程序的发布和打包

- 1. Qt 程序的发布
 - 1.1 生成 Release 版程序
 - 1.2 发布
- 2. Qt 程序打包
- 3. 安装

十六、Json的操作

- 1. json文件
 - 1.1 Json 数组
 - 1.2 Json 对象
 - 1.3 注意事项
- 2. Qt中Json的操作
 - 2.1 QJsonValue
 - 2.2 QJsonObject
 - 2.3. QJsonArray
 - 2.4. QJsonDocument
 - 2.5. 举例
 - 2.5.1 写文件
 - 2.5.2 读文件

一、Qt 入门

1.Qt 概述

1.1 什么是 Qt

不论我们学习什么样的知识点首先第一步都需要搞明白它是什么，这样才能明确当前学习的方向是否正确，下面给大家介绍一下什么是 Qt。

- 是一个跨平台的 C++ 应用程序开发框架
 - 具有短平快的优秀特质：投资少、周期短、见效快、效益高
 - 几乎支持所有的平台，可用于桌面程序开发以及嵌入式开发
 - 有属于自己的事件处理机制
 - 可以搞效率的开发基于窗口的应用程序。
- Qt 是标准 C++ 的扩展，C++ 的语法在 Qt 中都是支持的
 - 良好封装机制使得 Qt 的模块化程度非常高，可重用性较好，可以快速上手。
 - Qt 提供了一种称为 signals/slots 的安全类型来替代 callback（回调函数），这使得各个元件之间的协同工作变得十分简单

1.2 Qt 的特点

知道了 Qt 是什么之后，给大家介绍一下 Qt 这个框架的一些优点，就是因为具有了这些优秀的特质才使得现在很多企业都首选 Qt 进行基于窗口的应用程序开发，并且近年来市场对 Qt 程序猿的需求也在不断攀升。

- 广泛用于开发 GUI 程序，也可用于开发非 GUI 程序。
 - GUI = Graphical User Interface
 - 也就是基于窗口的应用程序开发。
- 有丰富的 API
 - Qt 包括多达 250 个以上的 C++ 类
 - 可以处理正则表达式。
- 支持 2D/3D 图形渲染，支持 OpenGL
- Qt 给程序员提供了非常详细的官方文档
- 支持 XML, JSON 框架底层模块化，使用者可以根据需求选择相应的模块来使用
- 可以轻松跨平台
 - 和 Java 的跨平台方式不同
 - 在不同的平台使用的是相同的上层接口，但是在底层封装了不同平台对应的 API（暗度陈仓）。

1.3 Qt 中的模块

Qt 类库里大量的类根据功能分为各种模块，这些模块又分为以下几大类：

- Qt 基本模块 (Qt Essentials)：提供了 Qt 在所有平台上的基本功能。
- Qt 附加模块 (Qt Add-Ons)：实现一些特定功能的提供附加价值的模块。
- 增值模块 (Value-Add Modules)：单独发布的提供额外价值的模块或工具。

- 技术预览模块 (Technology Preview Modules)：一些处于开发阶段，但是可以作为技术预览使用的模块。
- Qt 工具 (Qt Tools)：帮助应用程序开发的一些工具。

Qt 官网或者帮助文档的 “All Modules” 页面可以查看所有这些模块的信息。以下是官方对 Qt 基本模块的描述。关于其他模块感兴趣的话可以自行查阅。
于其他模块感兴趣的话可以自行查阅。

模块	描述
Qt Core	Qt 类库的核心，所有其他模块都依赖于此模块
Qt GUI	设计 GUI 界面的基础类，包括 OpenGL
Qt Multimedia	音频、视频、摄像头和广播功能的类
Qt Multimedia Widgets	实现多媒体功能的界面组件类
Qt Network	使网络编程更简单和轻便的类
Qt QML	用于 QML 和 JavaScript 语言的类
Qt Quick	用于构建具有定制用户界面的动态应用程序的声明框架
Qt Quick Controls	创建桌面样式用户界面，基于 Qt Quick 的用户界面控件
Qt Quick Dialogs	用于 Qt Quick 的系统对话框类型
Qt Quick Layouts	用于 Qt Quick 2 界面元素的布局项
Qt SQL	使用 SQL 用于数据库操作的类
Qt Test	用于应用程序和库进行单元测试的类
Qt Widgets	用于构建 GUI 界面的 C++ 图形组件类

1.4 Qt 案例

Qt 发展至今已经是一个非常成熟的框架，基于这个框架许多公司开发出了很多优秀的软件，下边给大家介绍几款我们常见的软件：

- VirtualBox：虚拟机软件。
- VLC 多媒体播放器：一个体积小巧、功能强大的开源媒体播放器。
- YY语音：又名“歪歪语音”，是一个可以进行在线多人语音聊天和语音会议的免费软件。
- 咪咕音乐：咪咕音乐是中国移动倾力打造的正版音乐播放器
- WPS Office：金山公司 (Kingsoft) 出品的办公软件，与微软 Office 兼容性良好，个人版免费。
- Skype：一个使用人数众多的基于 P2P 的 VOIP 聊天软件。

2. 安装

2.1 安装包下载

在我们学习 Qt 之前必须要安装开发环境，本小节基于 Qt 自带的集成开发环境（IDE）为大家讲解一下响应的开发步骤。

在这里我们基于 Window 平台 Qt 5.14.2 给大家讲解如何进行安装和相关配置。

- Qt官方下载地址：<https://download.qt.io/archive/qt/>

通过上边提供的地址打开 Qt 的下载页面，如下图：

Name	Last modified	Size	Metadata
↑ Parent Directory			
6.0/	07-Dec-2020 14:19	-	
5.15/	19-Nov-2020 13:11	-	
5.14/	31-Mar-2020 09:14	-	
5.13/	31-Oct-2019 07:17	-	
5.12/	09-Nov-2020 07:42	-	
5.9/	16-Dec-2019 15:00	-	
5.1/	06-May-2014 12:44	-	
5.0/	03-Jul-2013 11:57	-	
4.8/	04-Jun-2018 17:26	-	
4.7/	19-Feb-2013 16:14	-	
4.6/	19-Feb-2013 16:14	-	
4.5/	19-Feb-2013 16:13	-	
4.4/	19-Feb-2013 16:13	-	
4.3/	19-Feb-2013 16:13	-	
4.2/	19-Feb-2013 16:13	-	
4.1/	02-Mar-2020 11:44	-	
4.0/	02-Mar-2020 11:44	-	

QT的版本

选择需要的版本，通过链接进入下载页面

Name	Last modified
↑ Parent Directory	
submodules/	31-Mar-2020 09:27
single/	31-Mar-2020 10:10
qt-opensource-windows-x86-5.14.2.exe	Windows版 31-Mar-2020 10:18
qt-opensource-mac-x64-5.14.2.dmg	Mac版 31-Mar-2020 10:16
qt-opensource-linux-x64-5.14.2.run	Linux版 31-Mar-2020 10:14
md5sums.txt	31-Mar-2020 10:32

根据自己的需求下载不同平台对应的版本就可以了。

由于官方服务器在国外，不太稳定，有时候无法访问，可以通过国内几个著名的高校提供的 Qt 镜像网站进行下载：

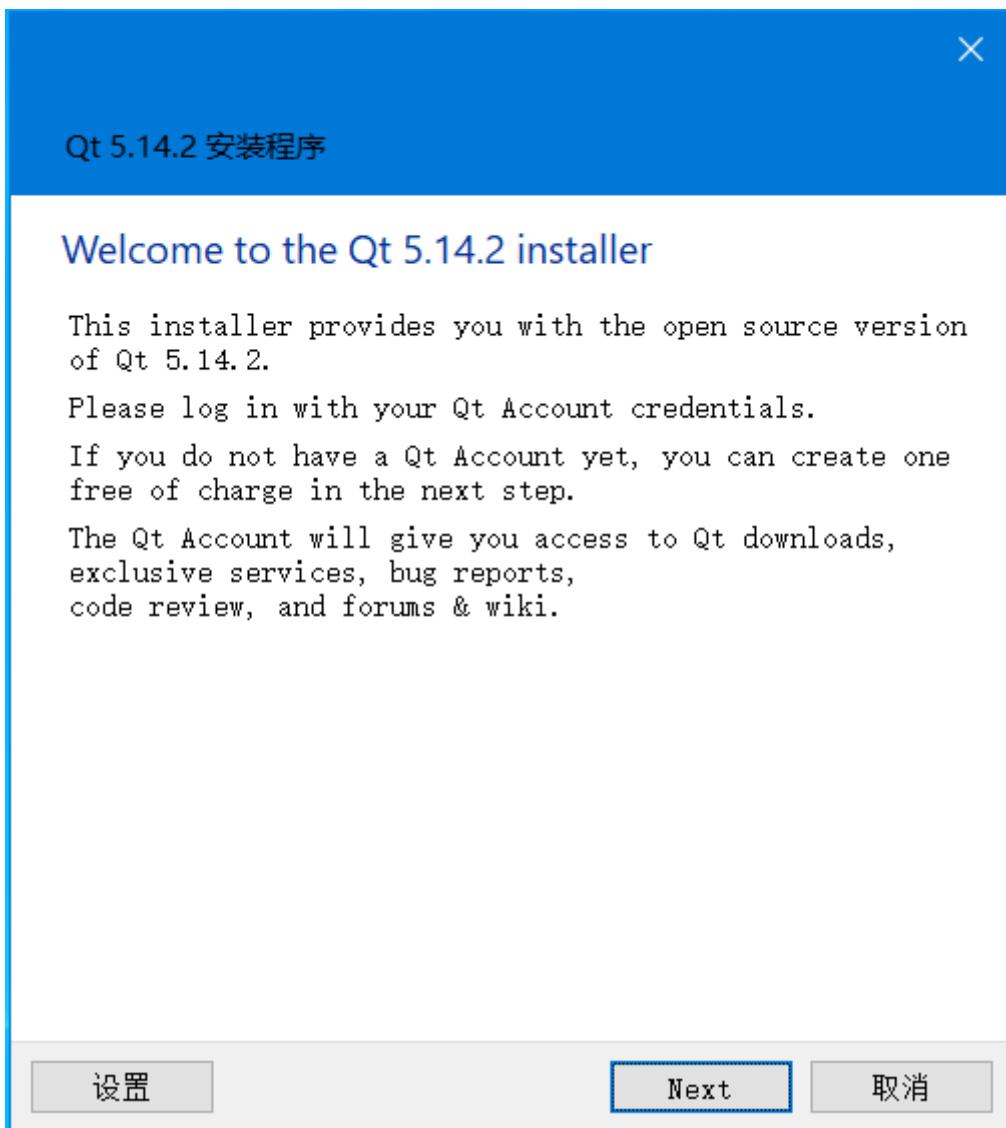
- 中国科学技术大学：<http://mirrors.ustc.edu.cn/qtproject/>
- 清华大学：<https://mirrors.tuna.tsinghua.edu.cn/qt/>
- 北京理工大学：<https://mirrors.bit.edu.cn/qtproject/>
- 中国互联网络信息中心：<https://mirrors.cnnic.cn/qt/>

找到 archive/ 目录下载离线安装包即可，以中国互联网络信息中心为例：

Index of /qt/archive/qt/		
File Name ↓	File Size ↓	Date ↓
Parent directory/	-	-
5.12/	-	2020-11-09 13:42
5.13/	-	2019-10-31 13:17
5.14/	-	2020-03-31 14:14
5.15/	-	2020-11-19 19:11
5.9/	-	2019-12-16 21:00
6.0/	-	2020-12-07 20:19
INSTALL	2.9 KiB	2012-12-13 20:58
LICENSE.GPL	17.9 KiB	2012-12-13 20:58
LICENSE.LGPL	26.2 KiB	2012-12-17 15:25
LICENSE.PREVIEW	21.7 KiB	2012-12-13 20:58
LICENSE.QPL	4.6 KiB	2012-12-13 20:58
copyrightnotice.txt	384 B	2012-12-13 20:58
md5sums.txt	49.6 KiB	2012-12-13 20:58

2.2 安装 (windows)

运行下载到本地的可执行程序，安装向导如下图：



填写 Qt 账号登录信息，如果没有就注册一个



Qt 5.14.2 安装程序

Qt Account – Your unified login to everything Qt

Please log in to Qt Account

Login

Email

Password

[Forgot password?](#)

Need a Qt Account?

Sign-up

Valid email address

没有账号在此填写账号信息

Confirm Password

I accept the [service terms](#).

设置

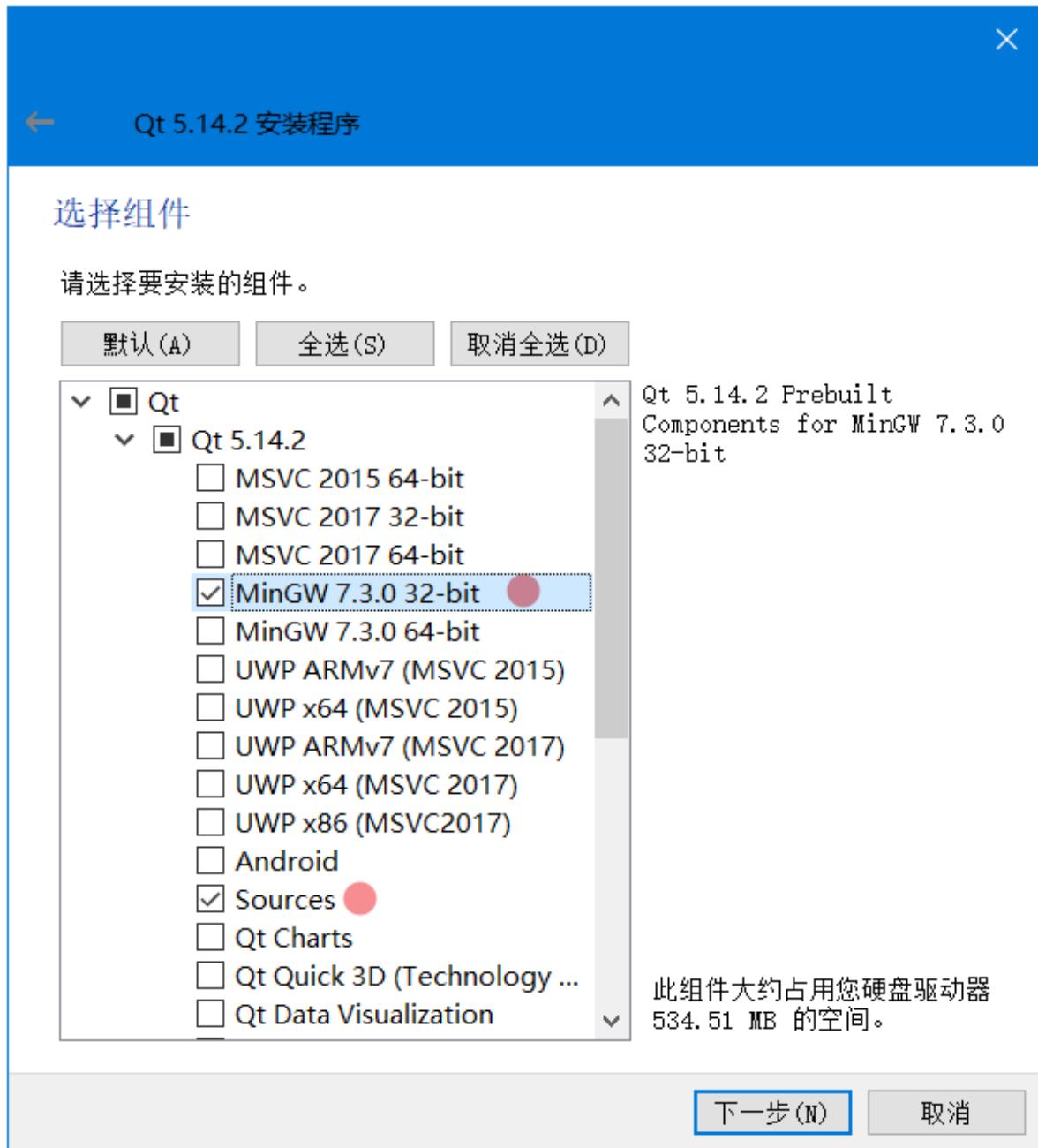
Next

取消

指定 Qt 的安装目录，安装文件需要占用比较大的磁盘空间



接下来需要选择 IDE 使用的编译套件的版本:



关于这些编译套件跟大家做一个介绍，安装过程中根据自己的情况酌情选择即可。

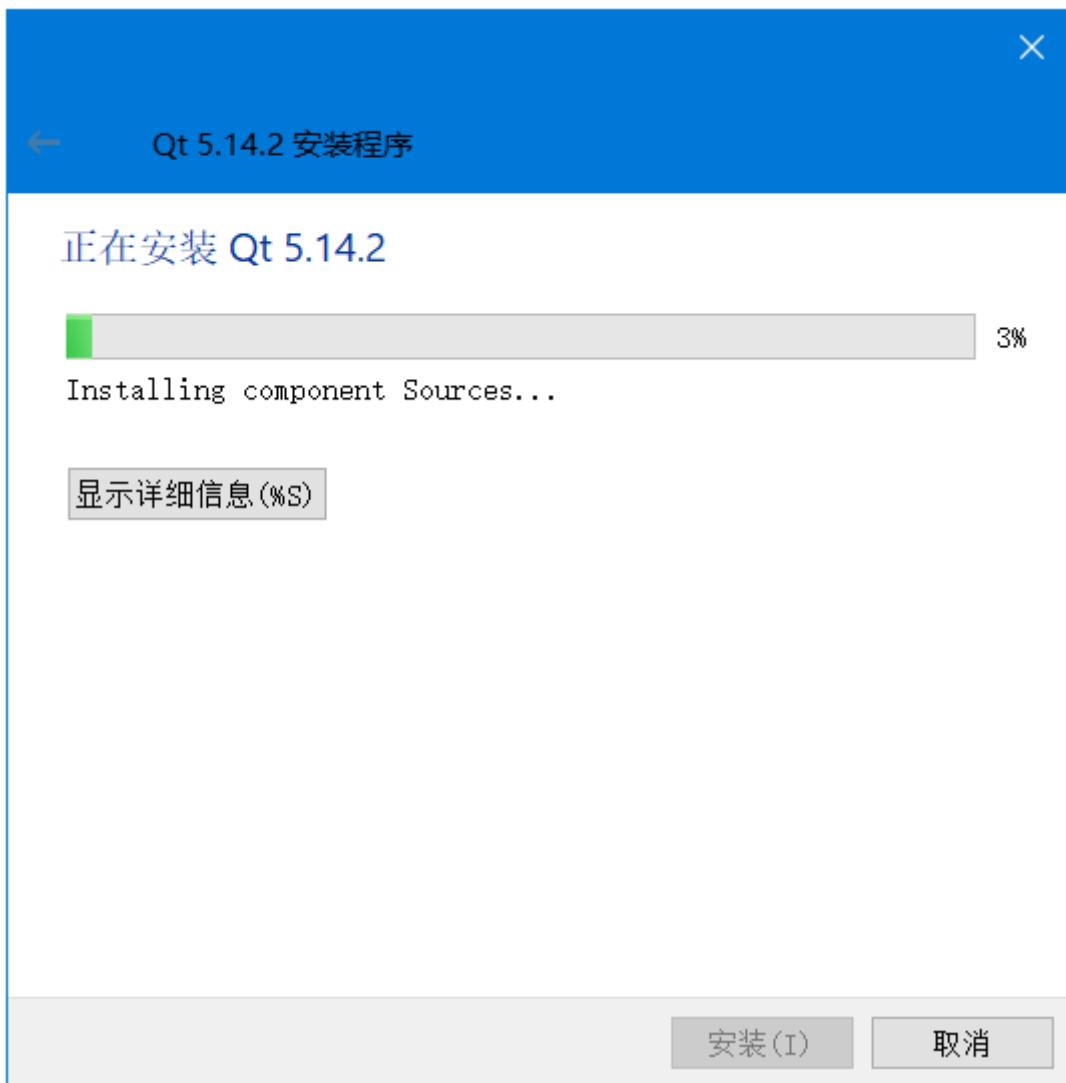
- MSVC2015 64-bit: Visual Studio 2015 使用的 64 位编译套件
- MSVC2017 32-bit: Visual Studio 2017 使用的 32 位编译套件
- MSVC2017 64-bit: Visual Studio 2017 使用的 64 位编译套件
- MinGW7.3.0 32-bit: QtCreator 使用的 32 位编译套件
- MinGW7.3.0 64-bit: QtCreator 使用的 64 位编译套件
- UWP -> Universal Windows Platform: 用于 window 平台应用程序开发的编译套件

UWP 即 Windows 通用应用平台，在 Windows 10 Mobile/Surface (Windows平板电脑) / PC/Xbox/HoloLens 等平台上运行，uwp 不同于传统 pc 上的 exe 应用，也跟只适用于手机端的 app 有本质区别。它并不是为某一个终端而设计，而是可以在所有 Windows10 设备上运行。

在这个窗口中除了选择必要的编译套件，还有一些非必要组件，常用的有以下两个：

- Source: Qt 源码，另外 Qt 的一些模块运行需要的驱动没有提供现成的动态库需要自己编译，建议安装
- Qt Charts: 用于绘制统计数据对应的图表，比如：折线图 / 曲线图等

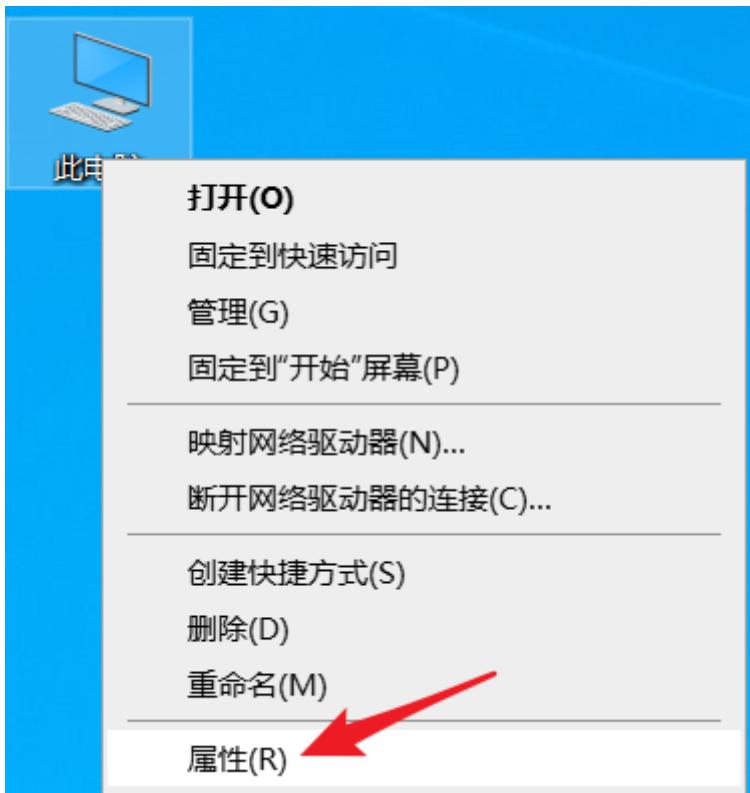
演示过程中选择安装了 MinGW7.3.0 32-bit 和 Source 两部分，接下来开始进行安装，这个过程需要漫长的等待...



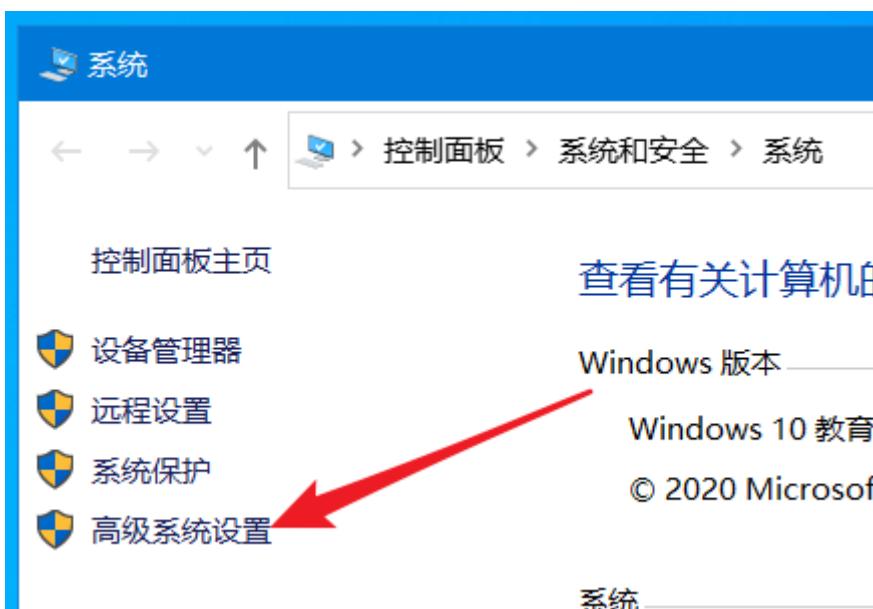
2.3 环境变量设置

当我们编写一个 Qt 程序，并且生成了可执行程序，这个可执行程序运行的时候默认需要加载相关的 Qt 动态库（因为默认是动态链接，静态链接则不需要）。为了保证可执行程序在任何目录执行都能链接到对应的动态库，我们可以将 Qt 模块对应的动态库目录设置到系统的环境变量中（这一点对于 Linux 系统也是一样的）。

1. 在桌面找到我的电脑（此电脑）图标，鼠标右键，打开属性窗口



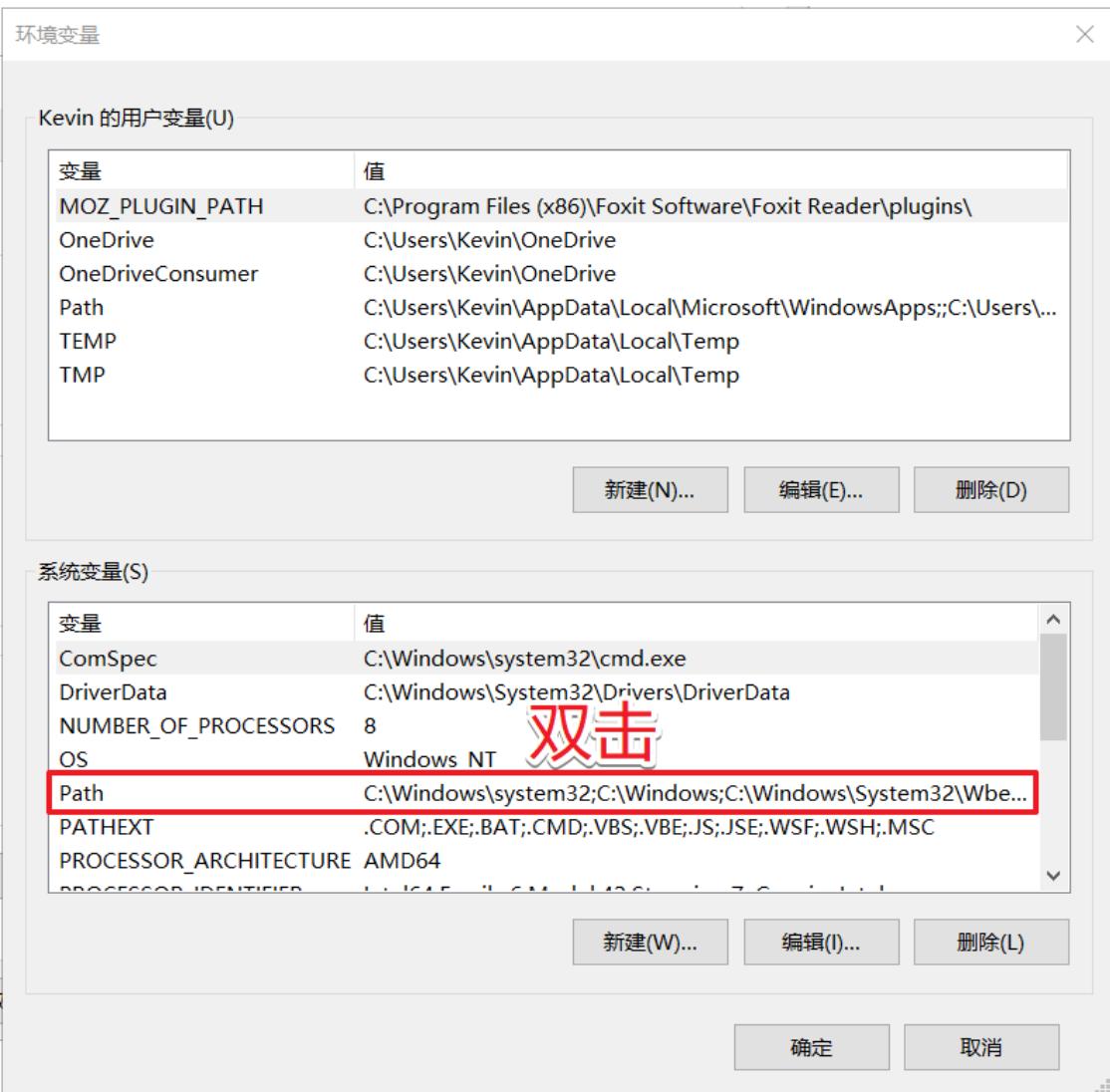
2. 在属性窗口中选择“高级系统设置”



3. 打开环境变量窗口

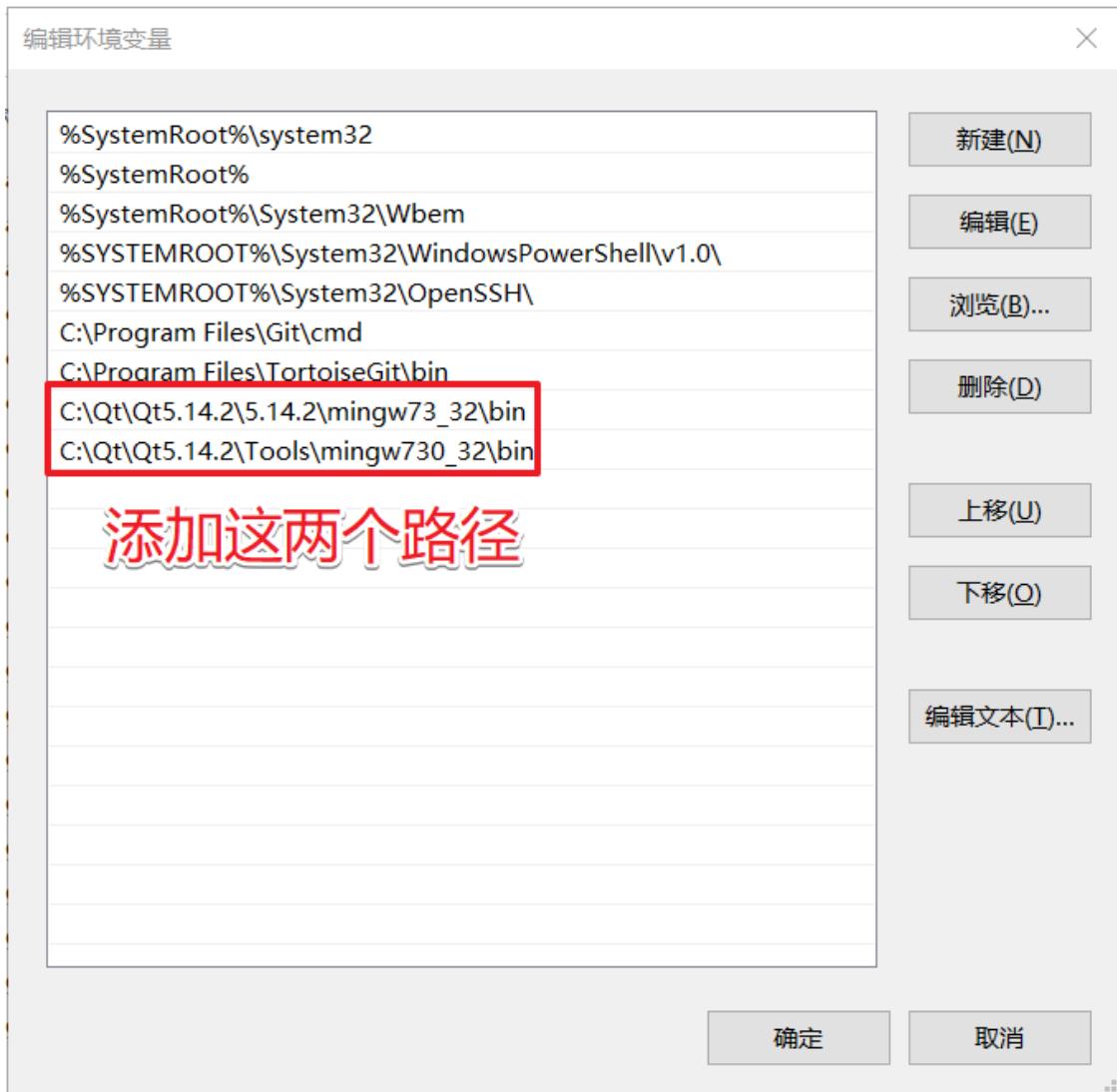


4. 新建环境变量



5. 将 Qt 的相关目录添加到系统环境变量中

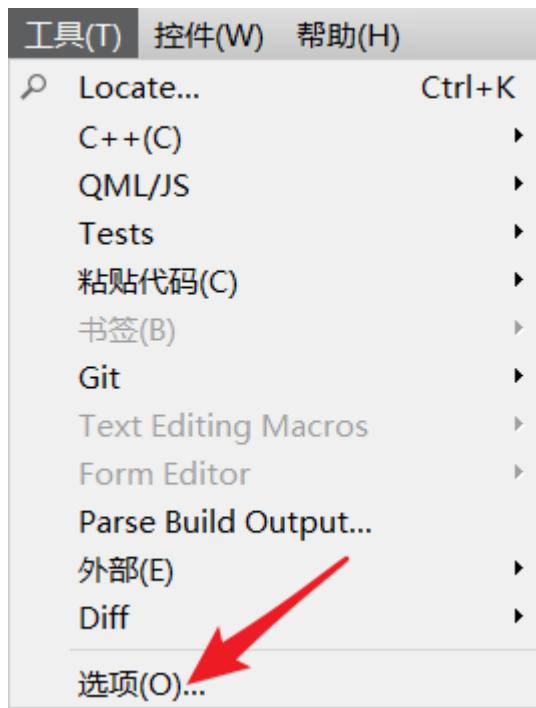
- 环境变量说明:
 - 找到 Qt 的安装目录: C:
 - 在安装目录中找到 Qt 库的 bin 目录: C:\Qt\Qt5.14.2\5.14.2\mingw73_32\bin
 - 在安装目录中找到编译套件的 bin 目录: C:\Qt\Qt5.14.2\Tools\mingw730_32\bin
- 以上目录为安装过程中的演示目录，各位小伙伴需要根据自己的实际情况，找到对应的本地路径。



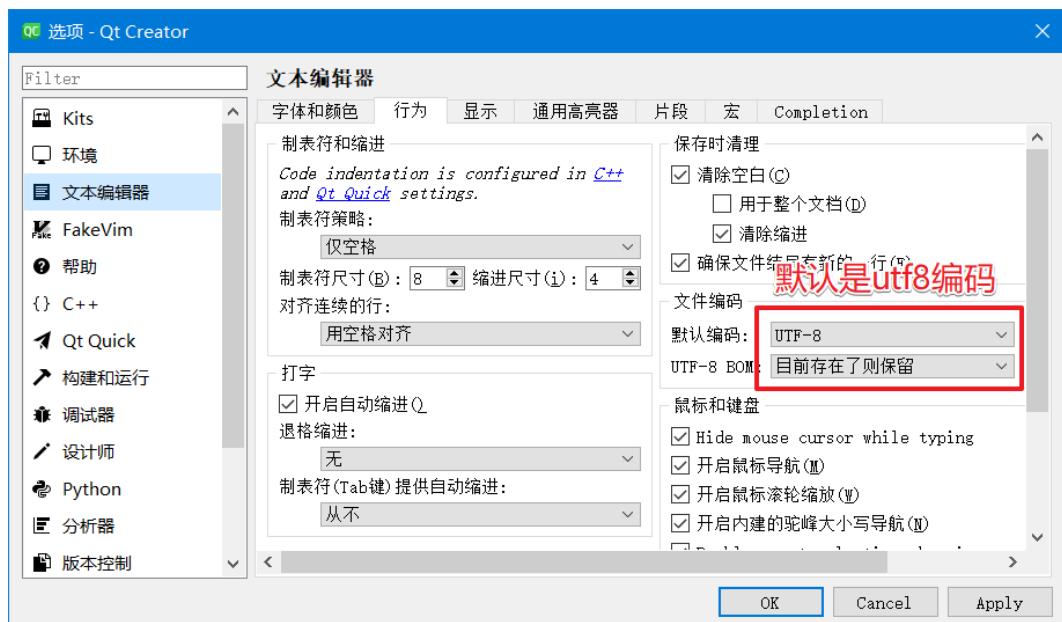
环境变量配置完毕之后，不会马上生效，需要注销或者重启计算机。

2.4 Qt Creator

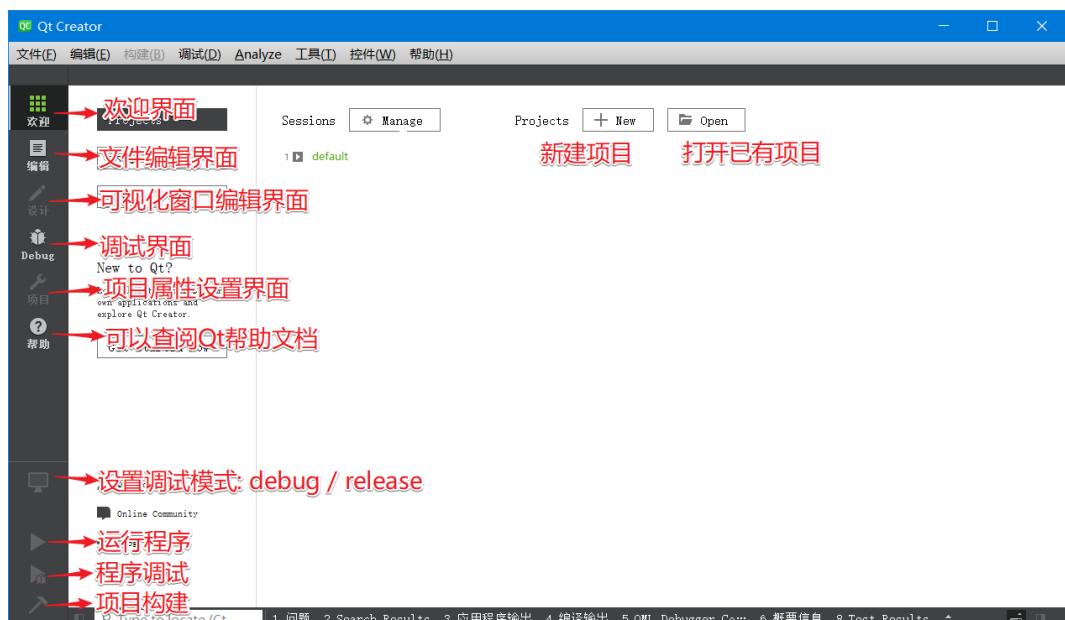
1. QtCreator 是编写 Qt 程序默认使用的一款 IDE，使用 VS 写 Qt 程序也是可以的，在此不做介绍。
2. 使用 QtCreator 创建的项目目录中不能包含中文
3. QtCreator 默认使用 Utf8 格式编码对文件字符进行编码
 - 字符必须编码后才能被计算机处理
 - 为了处理汉字，程序员设计了用于简体中文的 GB2312 和用于繁体中文的 big5。
 - GB2312 支持的汉字太少，1995 年的汉字扩展规范 GBK1.0，支持了更多的汉字。
 - 2000 年的 GB18030 取代了 GBK1.0 成为了正式的国家标准。
 - Unicode 也是一种字符编码方法，不过它是由国际组织设计，可以容纳全世界所有语言文字的编码方案
 - utf8
 - utf16
 - vs 写 Qt 程序默认使用的本地编码 -> gbk
 - 修改 QtCreator 的编码，菜单栏 -> 工具



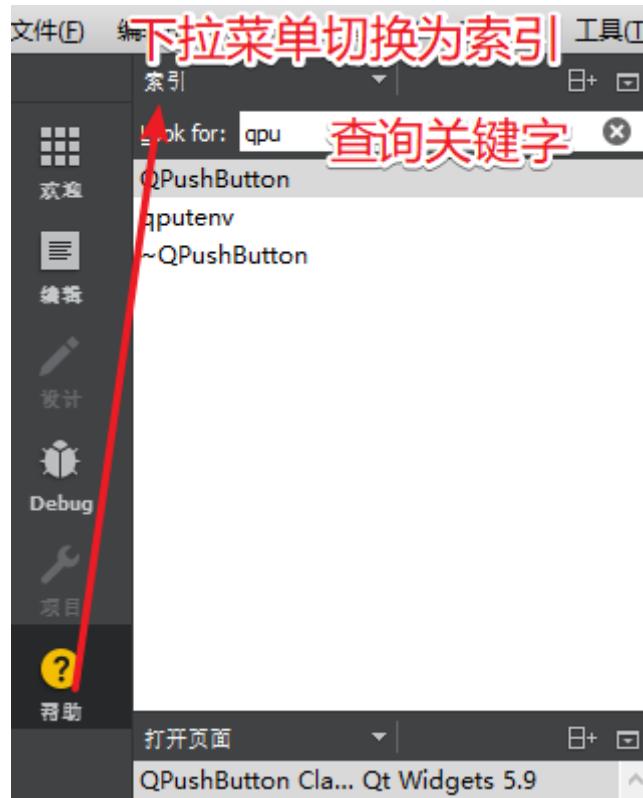
在弹出的选项窗口中设置文件编码，默认为 utf8



○ QtCreator 主界面介绍



通过 QtCreator 可以直接查阅 Qt 的帮助文档，里边对 Qt 框架提供的 API 做了非常详尽的介绍，查询方式如下图所示：



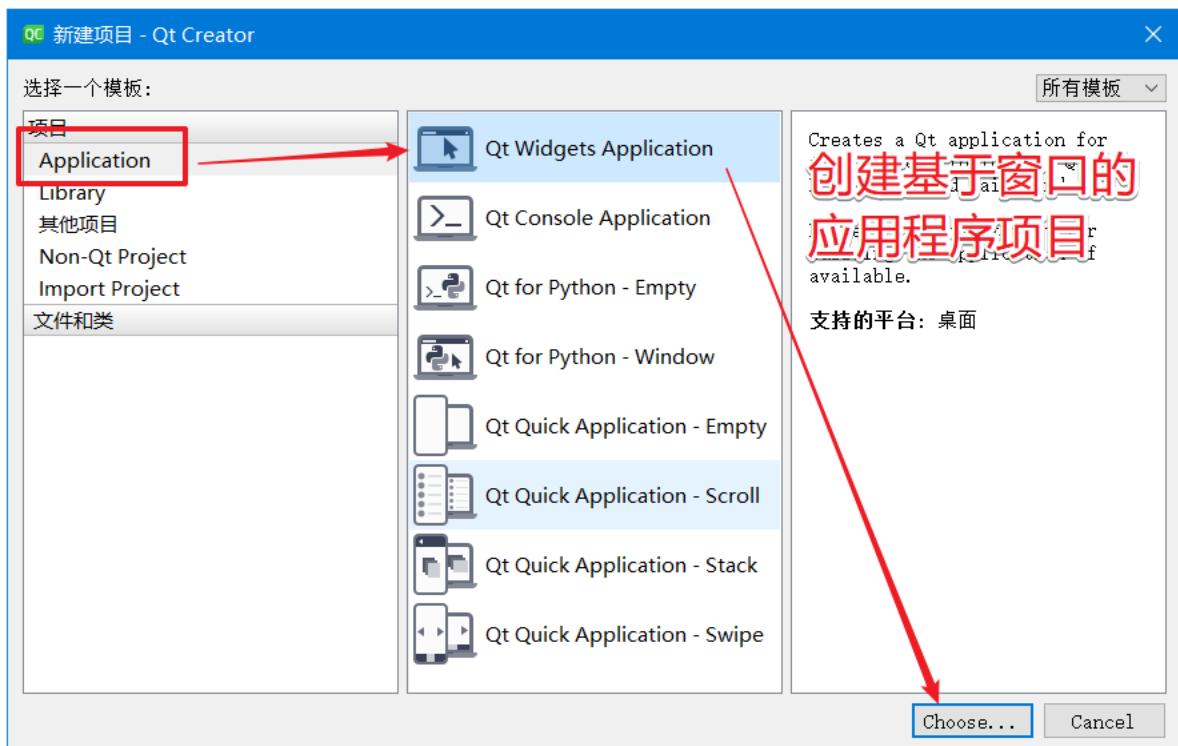
- 默认的编译套件：MinGW -> Minimalist GNU for Windows

- ◆MinGW 提供了一套简单方便的 Windows 下的基于 GCC 程序开发环境。MinGW 收集了一系列免费的 Windows 使用的头文件和库文件；
- ◆整合了 GNU 的工具集，特别是 GNU 程序开发工具，如经典 gcc, g++, make 等。
- ◆MinGW 是完全免费的自由软件，它在 Windows 平台上模拟了 Linux 下 GCC 的开发环境，为 C++ 的跨平台开发提供了良好基础支持，为了在 Windows 下工作的程序员熟悉 Linux 下的 C++ 工程组织提供了条件。

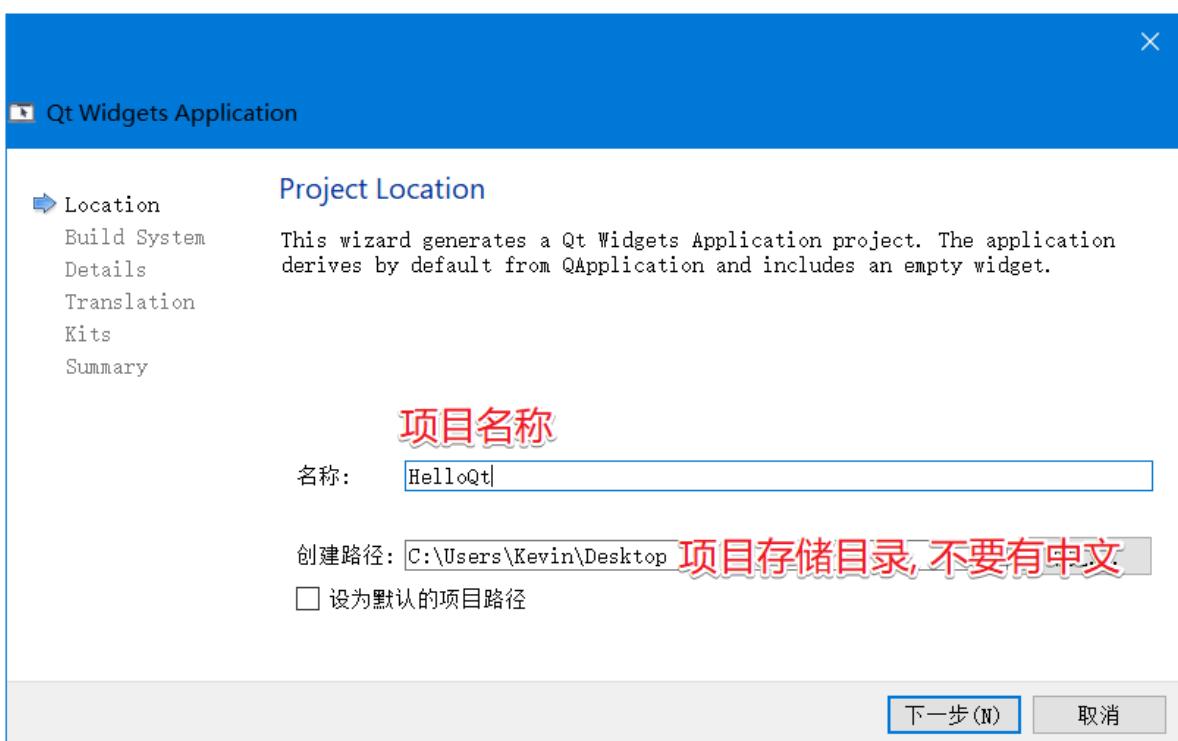
3. 第一个 Qt 项目

3.1 创建项目

创建基于窗口的 Qt 应用程序

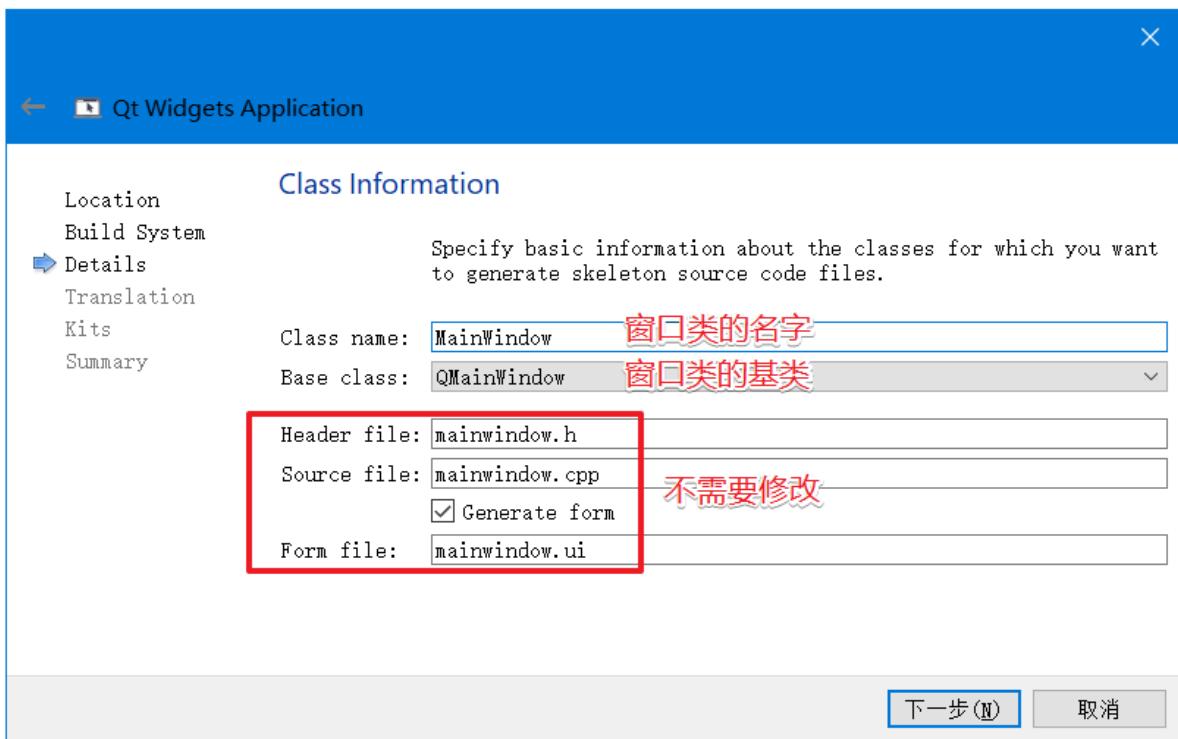


指定项目的存储路径

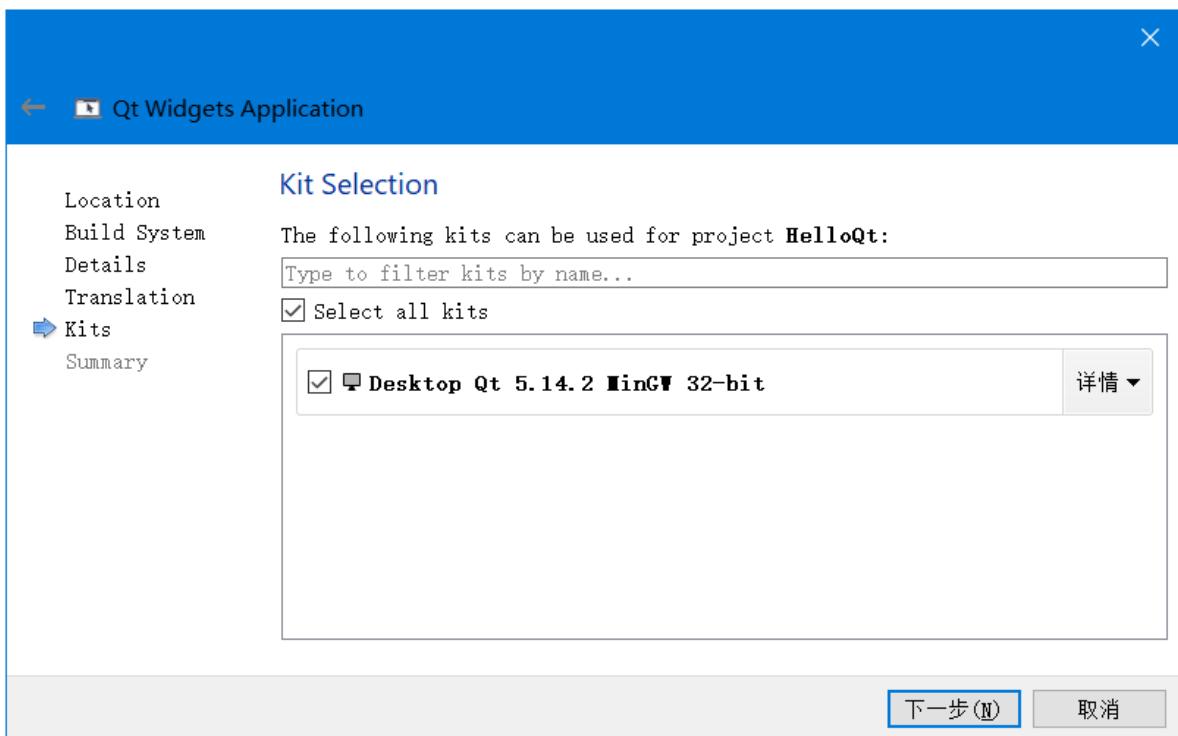


- 项目名称根据需求自己指定即可
- 在指定项目的存储路径的时候, 路径中不能包含中文, 不能包含中文, 不能包含中文

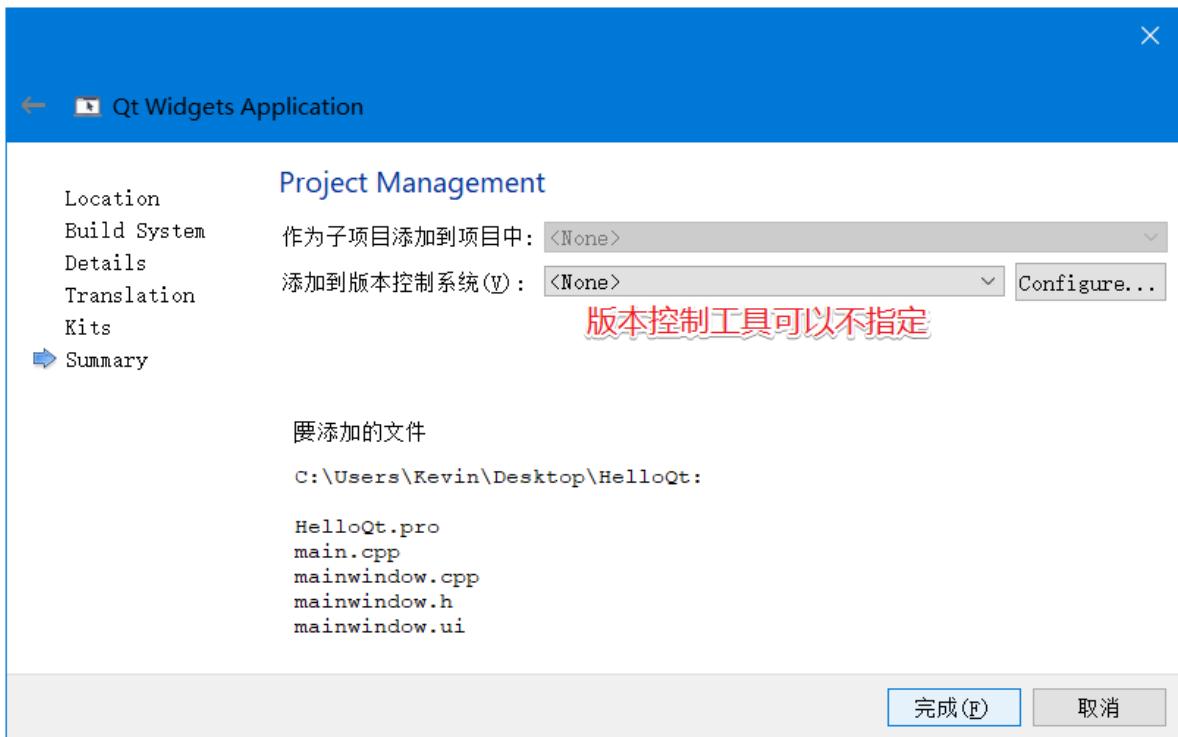
指定默认的窗口类的名字以及窗口的类型



选择编译套件，编译套件用于项目文件的编译，如果安装了多个编译套件，在这里选择其中一个就可以了。



选择版本控制工具，比如: git, svn 等，可以不指定。



3.2 项目文件 (.pro)

在创建的 Qt 项目中自动生成了一个后缀为 .pro 的项目文件，该文件中记录着项目的一些属性信息，具体信息如下：

```
1 # 在项目文件中，注释需要使用 井号(#)
2 # 项目编译的时候需要加载哪些底层模块
3 QT      += core gui
4
5 # 如果当前Qt版本大于4，会添加一个额外的模块：widgets
6 # Qt 5中对gui模块进行了拆分，将 widgets 独立出来了
7 greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
8
9 # 使用C++11新特性
10 CONFIG += c++11
11
12 #如果在项目中调用了废弃的函数，项目编译的时候会有警告的提示
13 DEFINES += QT_DEPRECATED_WARNINGS
14
15 # 项目中的源文件
16 SOURCES += \
17     main.cpp \
18     mainwindow.cpp
19
20 # 项目中的头文件
21 HEADERS += \
22     mainwindow.h
23
24 # 项目中的窗口界面文件
25 FORMS += \
26     mainwindow.ui
```

3.3 main.cpp

在这个源文件中有程序的入口函数 main()，下面给大家介绍下这个文件中自动生成的几行代码：

```
1 #include "mainwindow.h"      // 生成的窗口类头文件
2 #include <QApplication>     // 应用程序类头文件
3
4 int main(int argc, char *argv[])
{
    // 创建应用程序对象，在一个Qt项目中实例对象有且仅有一个
    // 类的作用：检测触发的事件，进行事件循环并处理
    QApplication a(argc, argv);
    // 创建窗口类对象
    MainWindow w;
    // 显示窗口
    w.show();
    // 应用程序对象开始事件循环，保证应用程序不退出
    return a.exec();
}
```

3.4 mainwindow.ui

在 Qt 中每一个窗口都对应一个可编辑的可视化界面 (*.ui)，这个界面对应的是一个 xml 格式的文件，一般情况下不需要在 xml 格式下对这个文件进行编辑，关于这个文件结构了解即可。

```
1 <!-- 双击这个文件看到的是一个窗口界面，如果使用文本编辑器打开看到的是一个XML格式的文件 --
2 -->
3 <!-- 看不懂这种格式没关系，我们不需要在这种模式下操作这个文件。 -->
4 <?xml version="1.0" encoding="UTF-8"?>
5 <ui version="4.0">
6   <class>MainWindow</class>
7   <widget class="QMainWindow" name="MainWindow">
8     <property name="geometry">
9       <rect>
10        <x>0</x>
11        <y>0</y>
12        <width>800</width>
13        <height>600</height>
14     </rect>
15   </property>
16   <property name="windowTitle">
17     <string>MainWindow</string>
18   </property>
19   <widget class="QWidget" name="centralwidget">
20     <widget class="QMenuBar" name="menubar">
21       <widget class="QStatusBar" name="statusbar"/>
22     </widget>
23     <resources/>
24     <connections/>
25   </ui>
```

3.5 mainwindow.h

这个文件是窗口界面对应的类的头文件。

```

1 #ifndef MAINWINDOW_H
2 #define MAINWINDOW_H
3
4 #include <QMainWindow>           // Qt标准窗口类头文件
5
6 QT_BEGIN_NAMESPACE
7 // mainwindow.ui 文件中也有一个类叫 MainWindow, 将这个类放到命名空间 ui 中
8 namespace ui { class MainWindow; }
9 QT_END_NAMESPACE
10
11 class MainWindow : public QMainWindow
12 {
13     Q_OBJECT // 这个宏是为了能够使用Qt中的信号槽机制
14
15 public:
16     MainWindow(QWidget *parent = nullptr);
17     ~MainWindow();
18
19 private:
20     Ui::MainWindow *ui; // 定义指针指向窗口的 UI 对象
21 };
22 #endif // MAINWINDOW_H

```

3.6 mainwindow.cpp

这个文件是窗口界面对应的类的源文件。

```

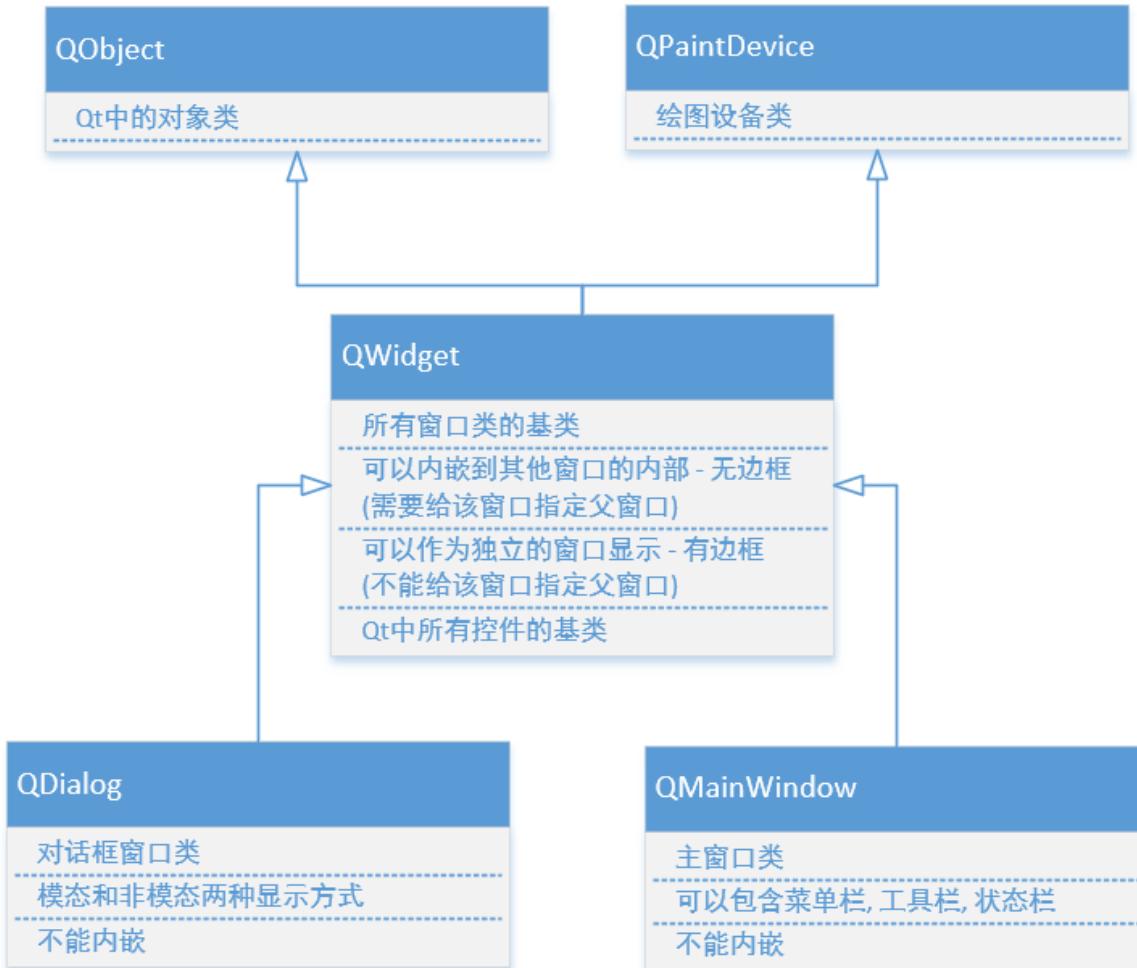
1 #include "mainwindow.h"
2 #include "ui_mainwindow.h"
3
4 MainWindow::MainWindow(QWidget *parent)
5     : QMainWindow(parent)
6     , ui(new Ui::MainWindow) // 基于mainwindow.ui创建一个实例对象
7 {
8     // 将 mainwindow.ui 的实例对象和 当前类的对象进行关联
9     // 这样同名的连个类对象就产生了关联，合二为一了
10    ui->setupUi(this);
11 }
12
13 MainWindow::~MainWindow()
14 {
15     delete ui;
16 }

```

4. Qt 中的窗口类

我们在通过 Qt 向导窗口基于窗口的应用程序的项目过程中倒数第二步让我们选择跟随项目创建的第一个窗口的基类，下拉菜单中有三个选项，分别为: QMainWindow、 QDialog、 QWidget 如下图：

4.1 基础窗口类



- 常用的窗口类有 3 个
 - 在创建 Qt 窗口的时候，需要让自己的窗口类继承上述三个窗口类的其中一个
- QWidget
 - 所有窗口类的基类
 - Qt 中的控件 (按钮, 输入框, 单选框...) 也属于窗口，基类都是 QWidget
 - 可以内嵌到其他窗口中：没有边框
 - 可以不内嵌单独显示：独立的窗口，有边框
- QDialog
 - 对话框类，后边的章节会具体介绍这个窗口
 - 不能内嵌到其他窗口中
- QMainWindow
 - 有工具栏，状态栏，菜单栏，后边的章节会具体介绍这个窗口
 - 不能内嵌到其他窗口中

4.2 窗口的显示

- 内嵌窗口
 - 依附于某一个大的窗口，作为大窗口的一部分
 - 大窗口就是这个内嵌窗口的父窗口
 - 父窗口显示的时候，内嵌的窗口也就被显示出来了
- 不内嵌窗口
 - 这类窗口有边框，有标题栏

- 需要调用函数才可以显示

```
1 // QWidget是所有窗口类的基类，调用这个提供的 show() 方法就可以显示将任何窗口显示出来
2 // 非模态显示
3 void QWidget::show(); // 显示当前窗口和它的子窗口
4
5 // 对话框窗口的非模态显示：还是调用show() 方法
6 // 对话框窗口的模态显示
7 [virtual slot] int QDialog::exec();
```

5. 坐标体系

在 Qt 关于窗口的显示是需要指定位置的，这个位置是通过坐标来确定的，所有坐标的选取又都是基于坐标原点来确定的，关于这些细节的确定，下面依次给大家进行讲解。

5.1 窗口的坐标原点

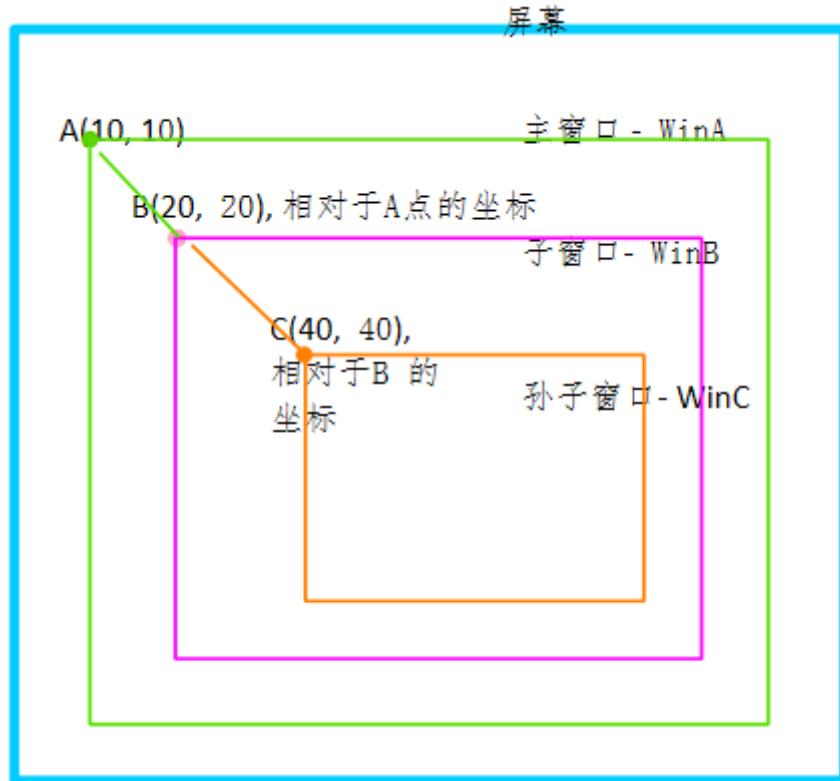
所有坐标的确定都需要先找到坐标原点，Qt的坐标原点在窗口的左上角

- x 轴向右递增
- y 轴向下递增



5.2 窗口的相对坐标

在一个 Qt 窗口中一般都有很多子窗口内嵌到这个父窗口中，其中每个窗口都有自己的坐标原点，子窗口的位置也就是其使用的坐标点就是它的父窗口坐标体系中的坐标点。



- 在 Qt 的某一个窗口中有可能有若干个控件，这个控件都是嵌套的关系
 - A 窗口包含 B 窗口，B 窗口包含 C 窗口
- 每个窗口都有坐标原点，在左上角
 - 子窗口的位置是基于父窗口的坐标体系来确定的，也就是说通过父窗口左上角的坐标点来确定自己的位置
- Qt 中窗口显示的时候使用的相对坐标，相对于自己的父窗口
 - 将子窗口移动到父窗口的某个位置

```

1 // 所有窗口类的基类: QWidget
2 // QWidget中提供了移动窗口的 API函数
3 // 参数 x, y是要移动的窗口的左上角的点, 窗口的左上角移动到这个坐标点
4 void QWidget::move(int x, int y);
5 void QWidget::move(const QPoint &);
```

6. 内存回收

在 Qt 中创建对象的时候会提供一个 Parent 对象指针（可以查看类的构造函数），下面来解释这个 parent 到底是干什么的。

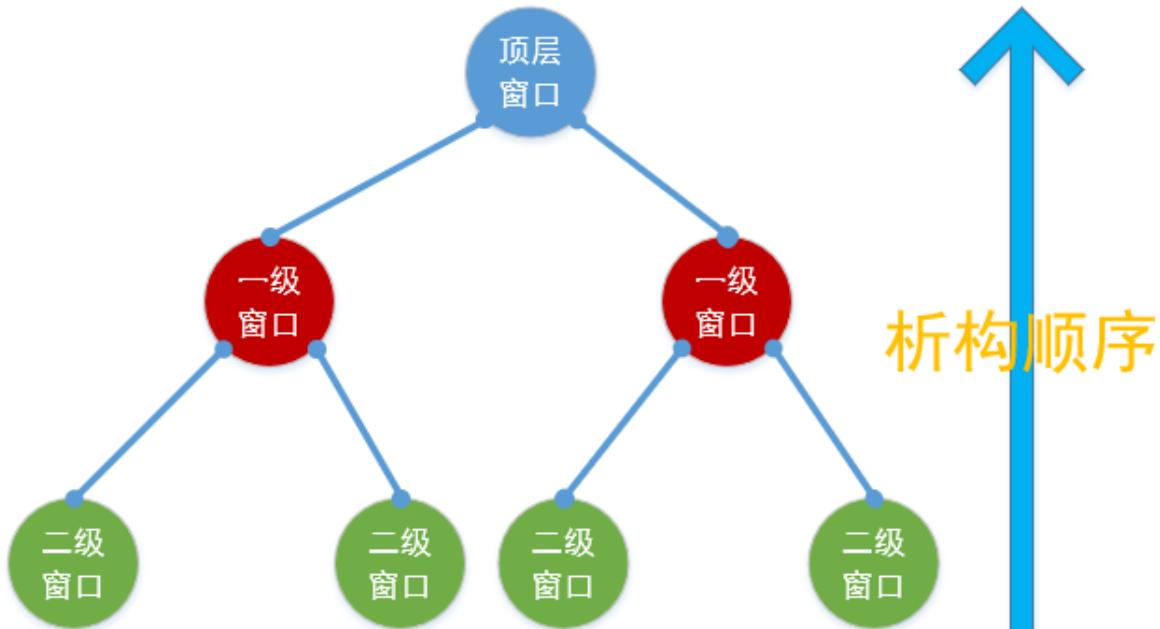
QObject 是以对象树的形式组织起来的。当你创建一个 QObject 对象时，会看到 QObject 的构造函数接收一个 QObject 指针作为参数，这个参数就是 parent，也就是父对象指针。这相当于，在创建 QObject 对象时，可以提供一个其父对象，我们创建的这个 QObject 对象会自动添加到其父对象的 children() 列表。当父对象析构的时候，这个列表中的所有对象也会被析构。（注意，这里的父对象并不是继承意义上的父类！）

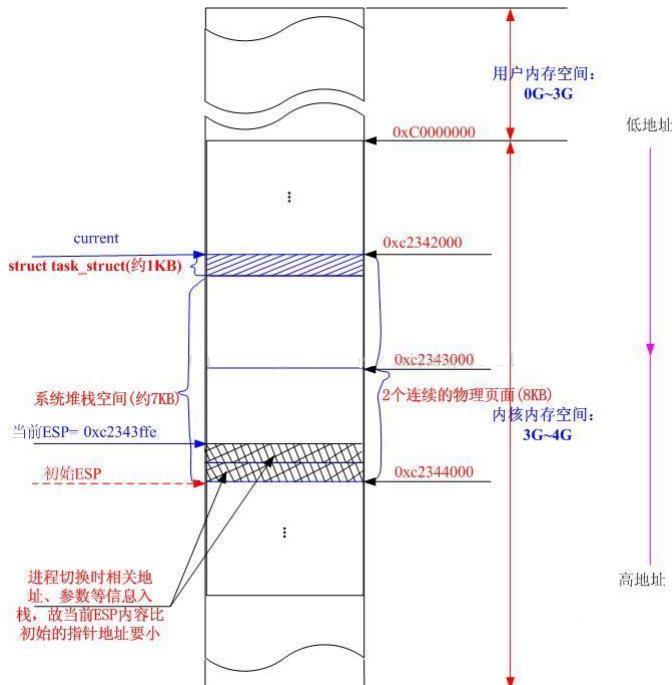
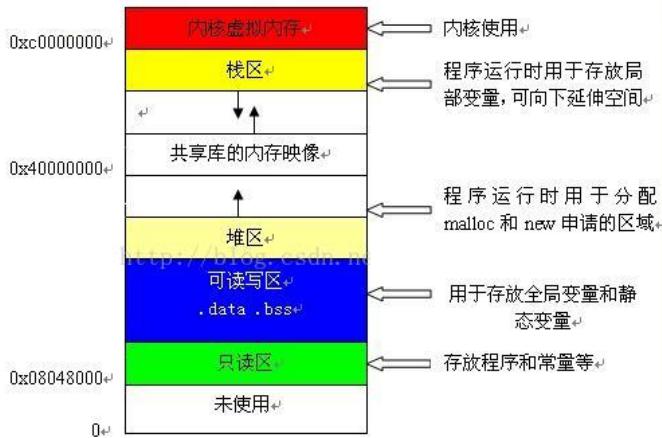
QWidget 是能够在屏幕上显示的一切组件的父类。QWidget 继承自 QObject，因此也继承了这种对象树关系。一个孩子自动地成为父组件的一个子组件。因此，它会显示在父组件的坐标系统中，被父组件的边界剪裁。例如，当用户关闭一个对话框的时候，应用程序将其删除，那么，我们希望属于这个对话框的按钮、图标等应该一起被删除。事实就是如此，因为这些都是对话框的子组件。

Qt 引入对象树的概念，在一定程度上解决了内存问题。

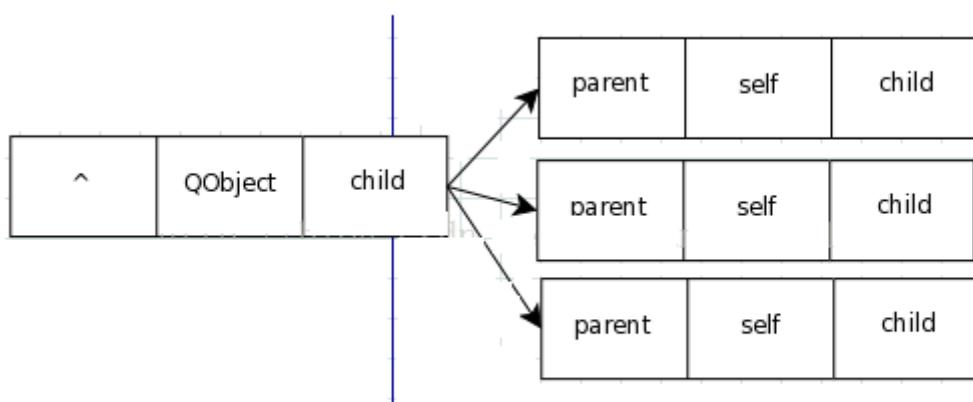
当一个 QObject 对象在堆上创建的时候，Qt 会同时为其创建一个对象树。不过，对象树中对象的顺序是没有定义的。这意味着，销毁这些对象的顺序也是未定义的。

任何对象树中的 QObject 对象 delete 的时候，如果这个对象有 parent，则自动将其从 parent 的 children () 列表中删除；如果有孩子，则自动 delete 每一个孩子。Qt 保证没有 QObject 会被 delete 两次，这是由析构顺序决定的。





在Qt中，最基础和核心的类是：QObject，QObject内部有一个list，会保存children，还有一个指针保存parent，当自己析构时，会自己从parent列表中删除并且析构所有的children



综上所述，我们可以得到一个结论：Qt中有内存回收机制，但是不是所有被new出的对象被自动回收，满足条件才可以回收，如果想要在 Qt 中实现内存的自动回收，需要满足以下两个条件：

- 创建的对象必须是 QObject 类的子类（间接子类也可以）
 - QObject 类是没有父类的，Qt 中有很大一部分类都是从这个类派生出去的
 - Qt 中使用频率很高的窗口类和控件都是 QObject 的直接或间接的子类

- 其他的类可以自己查阅 Qt 帮助文档
- 创建出的类对象，必须要指定其父对象是谁，一般情况下有两种操作方式：

```
1 // 方式1：通过构造函数
2 // parent: 当前窗口的父对象，找构造函数中的 parent 参数即可
3 QWidget::QWidget(QWidget *parent = Q_NULLPTR, Qt::WindowFlags f =
4     Qt::WindowFlags());
5
6 // 方式2：通过setParent()方法
7 // 假设这个控件没有在构造的时候指定父对象，可以调用QWidget的api指定父窗口对象
8 void QWidget::setParent(QWidget *parent);
9 void QObject::setParent(QObject *parent);
```

二、Qt中的基础数据类型

文章中主要介绍了 Qt 中常用的数据类型，主要内容包括：基础数据类型，Log日志输出，字符串类型，位置和尺寸相关类型，日期和时间相关类型。文章中除了关于知识点的文字描述。

1. 基础类型

因为 Qt 是一个 C++ 框架，因此 C++ 中所有的语法和数据类型在 Qt 中都是被支持的，但是 Qt 中也定义了一些属于自己的数据类型，下边给大家介绍一下这些基础的数据类型。

QT 基本数据类型定义在 #include 中，QT 基本数据类型有：

类型名称	注释	备注
qint8	signed char	有符号 8 位数据
qint16	signed short	16 位数据类型
qint32	signed int	32 位有符号数据类型
qint64	long long int 或 (_int64)	64 位有符号数据类型, Windows 中定义为_int64
qintptr	qint32 或 qint64	指针类型 根据系统类型不同而不同, 32 位系统为 qint32、64 位系统为 qint64
qlonglong	long long int 或 (_int64)	Windows 中定义为_int64
qptrdiff	qint32 或 qint64	根据系统类型不同而不同, 32 位系统为 qint32、64 位系统为 qint64
qreal	double 或 float	除非配置了 - qreal float 选项, 否则默认为 double
quint8	unsigned char	无符号 8 位数据类型
quint16	unsigned short	无符号 16 位数据类型
quint32	unsigned int	无符号 32 位数据类型
quint64	unsigned long long int 或 (unsigned _int64)	无符号 64 比特数据类型, Windows 中定义为 unsigned _int64
quintptr	quint32 或 quint64	根据系统类型不同而不同, 32 位系统为 quint32、64 位系统为 quint64
qulonglong	unsigned long long int 或 (unsigned _int64)	Windows 中定义为_int64
uchar	unsigned char	无符号字符类型
uint	unsigned int	无符号整型
ulong	unsigned long	无符号长整型
ushort	unsigned short	无符号短整型

虽然在Qt中有属于自己的整形或者浮点型,但是在变成过程中这些一般不用,常用的类型关键字还是C/C++中的 int, float, double 等。

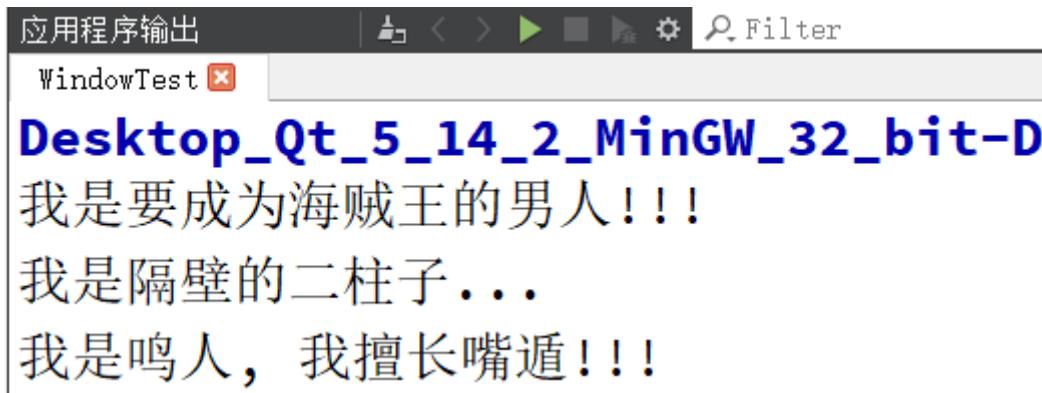
2. log 输出

2.1 在调试窗口中输入日志

在 Qt 中进行 log 输出，一般不使用 c 中的 printf，也不是使用 C++ 中的 cout，Qt 框架提供了专门用于日志输出的类，头文件名为 QDebug，使用方法如下：

```
1 // 包含了QDebug头文件，直接通过全局函数 qDebug() 就可以进行日志输出了
2 qDebug() << "Date:" << QDateTime::currentDate();
3 qDebug() << "Types:" << QString("String") << QChar('x') << QRect(0, 10, 50,
40);
4 qDebug() << "Custom coordinate type:" << coordinate;
5
6 // 和全局函数 qDebug() 类似的日志函数还有：qwarning(), qInfo(), qCritical()
7 int number = 666;
8 float i = 11.11;
9 qwarning() << "Number:" << number << "Other value:" << i;
10 qInfo() << "Number:" << number << "Other value:" << i;
11 qCritical() << "Number:" << number << "Other value:" << i;
12
13 qDebug() << "我是要成为海贼王的男人!!!";
14 qDebug() << "我是隔壁的二柱子...";
15 qDebug() << "我是鸣人，我擅长嘴遁!!!";
```

日志信息在 IDE 的调试窗口输出



2.2 在终端窗口中输出日志

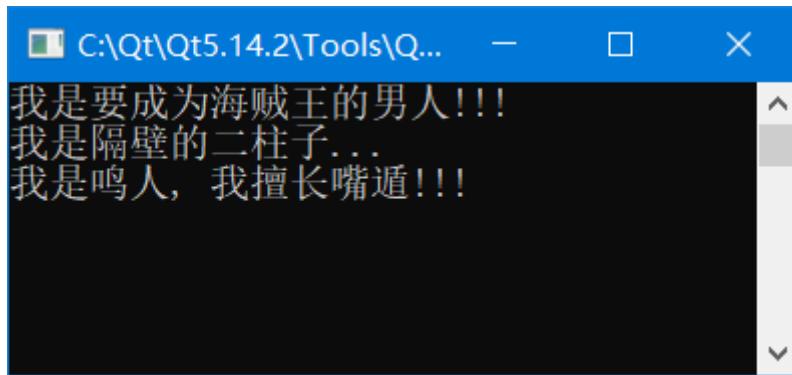
使用上面的方法只能在项目调试过程中进行日志输出，如果不是通过 IDE 进行程序调试，而是直接执行可执行程序在这种情况下是没有日志输出窗口的，因此也就看不到任何的日志输出。

默认情况下日志信息是不会打印到终端窗口的，如果想要实现这样的效果，必须在项目文件中添加相关的属性信息

打开项目文件 (*.pro) 找到配置项 config，添加 console 控制台属性：

```
1 | CONFIG += c++11 console
```

属性信息添加完毕，重新编译项目 日志信息就可以打印到终端窗口了



3. 字符串类型

在 Qt 中不仅支持 C, C++ 中的字符串类型，而且还在框架中定义了专属的字符串类型，我们必须掌握在 Qt 中关于这些类型的使用和相互之间的转换。

语言类型	字符串类型
C	char*
C++	std::string, char*
Qt	QByteArray, QString 等

3.1 QByteArray

在 Qt 中 QByteArray 可以看做是 c 语言中 char* 的升级版本。我们在使用这种类型的时候可通过这个类的构造函数申请一块动态内存，用于存储我们需要处理的字符串数据。

下面给大家介绍一下这个类中常用的一些 API 函数，大家要养成遇到问题主动查询帮助文档的好习惯

3.1.1 构造函数:

```
1 // 构造空对象，里边没有数据
2 QByteArray::QByteArray();
3 // 将data中的size个字符进行构造，得到一个字节数组对象
4 // 如果 size== -1 函数内部自动计算字符串长度，计算方式为: strlen(data)
5 QByteArray::QByteArray(const char *data, int size = -1);
6 // 构造一个长度为size个字节，并且每个字节值都为ch的字节数组
7 QByteArray::QByteArray(int size, char ch)
```

3.1.2 数据操作:

```
1 // 在尾部追加数据
2 // 其他重载的同名函数可参考Qt帮助文档，此处略
3 QByteArray &QByteArray::append(const QByteArray &ba);
4 void QByteArray::push_back(const QByteArray &other);
5
6 // 头部添加数据
7 // 其他重载的同名函数可参考Qt帮助文档，此处略
8 QByteArray &QByteArray::prepend(const QByteArray &ba);
```

```

9 void QByteArray::push_front(const QByteArray &other);
10
11 // 插入数据，将ba插入到数组第 i 个字节的位置(从0开始)
12 // 其他重载的同名函数可参考Qt帮助文档，此处略
13 QByteArray &QByteArray::insert(int i, const QByteArray &ba);
14
15 // 删除数据
16 // 从大字符串中删除len个字符，从第pos个字符的位置开始删除
17 QByteArray &QByteArray::remove(int pos, int len);
18 // 从字符数组的尾部删除 n 个字节
19 void QByteArray::chop(int n);
20 // 从字节数组的 pos 位置将数组截断 (前边部分留下，后边部分被删除)
21 void QByteArray::truncate(int pos);
22 // 将对象中的数据清空，使其为null
23 void QByteArray::clear();
24
25 // 字符串替换
26 // 将字节数组中的 子字符串 before 替换为 after
27 // 其他重载的同名函数可参考Qt帮助文档，此处略
28 QByteArray &QByteArray::replace(const QByteArray &before, const QByteArray
&after);

```

3.1.3 子字符串查找和判断:

```

1 // 判断字节数组中是否包含子字符串 ba，包含返回true，否则返回false
2 bool QByteArray::contains(const QByteArray &ba) const;
3 bool QByteArray::contains(const char *ba) const;
4 // 判断字节数组中是否包含子字符 ch，包含返回true，否则返回false
5 bool QByteArray::contains(char ch) const;
6
7 // 判断字节数组是否以字符串 ba 开始，是返回true，不是返回false
8 bool QByteArray::startswith(const QByteArray &ba) const;
9 bool QByteArray::startswith(const char *ba) const;
10 // 判断字节数组是否以字符 ch 开始，是返回true，不是返回false
11 bool QByteArray::startswith(char ch) const;
12
13 // 判断字节数组是否以字符串 ba 结尾，是返回true，不是返回false
14 bool QByteArray::endswith(const QByteArray &ba) const;
15 bool QByteArray::endswith(const char *ba) const;
16 // 判断字节数组是否以字符 ch 结尾，是返回true，不是返回false
17 bool QByteArray::endswith(char ch) const;

```

3.1.4 遍历:

```

1 // 使用迭代器
2 iterator QByteArray::begin();
3 iterator QByteArray::end();
4
5 // 使用数组的方式进行遍历
6 // i 的取值范围 0 <= i < size()
7 char QByteArray::at(int i) const;
8 char QByteArray::operator[](int i) const;

```

3.1.5 查看字节数:

```

1 // 返回字节数组对象中字符的个数
2 int QByteArray::length() const;
3 int QByteArray::size() const;
4 int QByteArray::count() const;
5
6 // 返回字节数组对象中 子字符串ba 出现的次数
7 int QByteArray::count(const QByteArray &ba) const;
8 int QByteArray::count(const char *ba) const;
9 // 返回字节数组对象中 字符串ch 出现的次数
10 int QByteArray::count(char ch) const;

```

3.1.6 类型转换:

```

1 // 将QByteArray类型的字符串 转换为 char* 类型
2 char *QByteArray::data();
3 const char *QByteArray::data() const;
4
5 // int, short, long, float, double -> QByteArray
6 // 其他重载的同名函数可参考Qt帮助文档, 此处略
7 QByteArray &QByteArray::setNum(int n, int base = 10);
8 QByteArray &QByteArray::setNum(short n, int base = 10);
9 QByteArray &QByteArray::setNum(qlonglong n, int base = 10);
10 QByteArray &QByteArray::setNum(float n, char f = 'g', int prec = 6);
11 QByteArray &QByteArray::setNum(double n, char f = 'g', int prec = 6);
12 [static] QByteArray QByteArray::number(int n, int base = 10);
13 [static] QByteArray QByteArray::number(qlonglong n, int base = 10);
14 [static] QByteArray QByteArray::number(double n, char f = 'g', int prec =
6);
15
16 // QByteArray -> int, short, long, float, double
17 int QByteArray::toInt(bool *ok = Q_NULLPTR, int base = 10) const;
18 short QByteArray::toShort(bool *ok = Q_NULLPTR, int base = 10) const;
19 long QByteArray::toLong(bool *ok = Q_NULLPTR, int base = 10) const;
20 float QByteArray::toFloat(bool *ok = Q_NULLPTR) const;
21 double QByteArray::toDouble(bool *ok = Q_NULLPTR) const;
22
23 // std::string -> QByteArray
24 [static] QByteArray QByteArray::fromStdString(const std::string &str);
25 // QByteArray -> std::string
26 std::string QByteArray::toStdString() const;
27
28 // 所有字符转换为大写
29 QByteArray QByteArray::toUpper() const;
30 // 所有字符转换为小写
31 QByteArray QByteArray::toLower() const;

```

3.2 QString

QString 也是封装了字符串, 但是内部的编码为 utf8, UTF-8 属于 Unicode 字符集, 它固定使用多个字节 (window为2字节, linux为3字节) 来表示一个字符, 这样可以将世界上几乎所有语言的常用字符收录其中。

下面给大家介绍一下这个类中常用的一些 API 函数。

3.2.1 构造函数:

```
1 // 构造一个空字符串对象
2 QString::QString();
3 // 将 char* 字符串 转换为 QString 类型
4 QString::QString(const char *str);
5 // 将 QByteArray 转换为 QString 类型
6 QString::QString(const QByteArray &ba);
7 // 其他重载的同名构造函数可参考Qt帮助文档，此处略
```

3.2.2 数据操作:

```
1 // 尾部追加数据
2 // 其他重载的同名函数可参考Qt帮助文档，此处略
3 QString &QString::append(const QString &str);
4 QString &QString::append(const char *str);
5 QString &QString::append(const QByteArray &ba);
6 void QString::push_back(const QString &other);
7
8 // 头部添加数据
9 // 其他重载的同名函数可参考Qt帮助文档，此处略
10 QString &QString::prepend(const QString &str);
11 QString &QString::prepend(const char *str);
12 QString &QString::prepend(const QByteArray &ba);
13 void QString::push_front(const QString &other);
14
15 // 插入数据，将 str 插入到字符串第 position 个字符的位置(从0开始)
16 // 其他重载的同名函数可参考Qt帮助文档，此处略
17 QString &QString::insert(int position, const QString &str);
18 QString &QString::insert(int position, const char *str);
19 QString &QString::insert(int position, const QByteArray &str);
20
21 // 删除数据
22 // 从大字符串中删除len个字符，从第pos个字符的位置开始删除
23 QString &QString::remove(int position, int n);
24
25 // 从字符串的尾部删除 n 个字符
26 void QString::chop(int n);
27 // 从字节串的 position 位置将字符串截断 (前边部分留下，后边部分被删除)
28 void QString::truncate(int position);
29 // 将对象中的数据清空，使其为null
30 void QString::clear();
31
32 // 字符串替换
33 // 将字节数组中的 子字符串 before 替换为 after
34 // 参数 cs 为是否区分大小写，默认区分大小写
35 // 其他重载的同名函数可参考Qt帮助文档，此处略
36 QString &QString::replace(const QString &before, const QString &after,
                           Qt::CaseSensitivity cs = Qt::CaseSensitive);
```

3.2.3 子字符串查找和判断:

```

1 // 参数 cs 为是否区分大小写， 默认区分大小写
2 // 其他重载的同名函数可参考Qt帮助文档， 此处略
3
4 // 判断字符串中是否包含子字符串 str， 包含返回true， 否则返回false
5 bool QString::contains(const QString &str, Qt::CaseSensitivity cs =
6 Qt::CaseSensitive) const;
7
8 // 判断字符串是否以字符串 ba 开始， 是返回true， 不是返回false
9 bool QString::startswith(const QString &s, Qt::CaseSensitivity cs =
10 Qt::CaseSensitive) const;
11
12 // 判断字符串是否以字符串 ba 结尾， 是返回true， 不是返回false
13 bool QString::endswith(const QString &s, Qt::CaseSensitivity cs =
14 Qt::CaseSensitive) const;

```

3.2.4 遍历:

```

1 // 使用迭代器
2 iterator QString::begin();
3 iterator QString::end();
4
5 // 使用数组的方式进行遍历
6 // i的取值范围 0 <= position < size()
7 const QChar QString::at(int position) const
8 const QChar QString::operator[](int position) const;

```

3.2.5 查看字节数:

```

1 // 返回字节数组对象中字符的个数 (字符个数和字节个数是不同的概念)
2 int QString::length() const;
3 int QString::size() const;
4 int QString::count() const;
5
6 // 返回字符串对象中 子字符串 str 出现的次数
7 // 参数 cs 为是否区分大小写， 默认区分大小写
8 int QString::count(const QStringRef &str, Qt::CaseSensitivity cs =
9 Qt::CaseSensitive) const;

```

3.2.6 类型转换:

```

1 // 将int, short, long, float, double 转换为 QString 类型
2 // 其他重载的同名函数可参考Qt帮助文档， 此处略
3 QString &QString::setNum(int n, int base = 10);
4 QString &QString::setNum(short n, int base = 10);
5 QString &QString::setNum(long n, int base = 10);
6 QString &QString::setNum(float n, char format = 'g', int precision = 6);
7 QString &QString::setNum(double n, char format = 'g', int precision = 6);
8 [static] QString QString::number(long n, int base = 10);
9 [static] QString QString::number(int n, int base = 10);
10 [static] QString QString::number(double n, char format = 'g', int precision
11 = 6);
12
13 // 将 QString 转换为 int, short, long, float, double 类型
14 int QString::toInt(bool *ok = Q_NULLPTR, int base = 10) const;
15 short QString::toShort(bool *ok = Q_NULLPTR, int base = 10) const;

```

```

15 long QString::toLong(bool *ok = Q_NULLPTR, int base = 10) const
16 float QString::toFloat(bool *ok = Q_NULLPTR) const;
17 double QString::toDouble(bool *ok = Q_NULLPTR) const;
18
19 // 将标准C++中的 std::string 类型 转换为 QString 类型
20 [static] QString QString::fromStdString(const std::string &str);
21 // 将 QString 转换为 标准C++中的 std::string 类型
22 std::string QString::toStdString() const;
23
24 // QString -> QByteArray
25 // 转换为本地编码，跟随操作系统
26 QByteArray QString::toLocal8Bit() const;
27 // 转换为 Latin-1 编码的字符串 不支持中文
28 QByteArray QString::toLatin1() const;
29 // 转换为 utf8 编码格式的字符串 (常用)
30 QByteArray QString::toUtf8() const;
31
32 // 所有字符转换为大写
33 QString QString::toUpperCase() const;
34 // 所有字符转换为小写
35 QString QString::toLowerCase() const;

```

3.2.7 字符串格式:

```

1 // 其他重载的同名函数可参考Qt帮助文档，此处略
2 QString QString::arg(const QString &a,
3                     int fieldwidth = 0,
4                     QChar fillchar = QLatin1Char(' ')) const;
5 QString QString::arg(int a, int fieldwidth = 0,
6                     int base = 10,
7                     QChar fillchar = QLatin1Char(' ')) const;
8
9 // 示例程序
10 int i; // 假设该变量表示当前文件的编号
11 int total; // 假设该变量表示文件的总个数
12 QString fileName; // 假设该变量表示当前文件的名字
13 // 使用以上三个变量拼接一个动态字符串
14 QString status = QString("Processing file %1 of %2: %3")
15 .arg(i).arg(total).arg(fileName);

```

4. 位置和尺寸

在 QT 中我们常见的 点，线，尺寸，矩形都被进行了封装，下边依次为大家介绍相关的类。

4.1 QPoint

QPoint 类封装了我们常用用到的坐标点 (x, y), 常用的 API 如下:

```

1 // 构造函数
2 // 构造一个坐标原点，即(0, 0)
3 QPoint::QPoint();
4 // 参数为 x轴坐标, y轴坐标
5 QPoint::QPoint(int xpos, int ypos);
6

```

```

7 // 设置x轴坐标
8 void QPoint::setX(int x);
9 // 设置y轴坐标
10 void QPoint::setY(int y);
11
12 // 得到x轴坐标
13 int QPoint::x() const;
14 // 得到x轴坐标的引用
15 int &QPoint::rx();
16 // 得到y轴坐标
17 int QPoint::y() const;
18 // 得到y轴坐标的引用
19 int &QPoint::ry();
20
21 // 直接通过坐标对象进行算术运算：加减乘除
22 QPoint &QPoint::operator*=(float factor);
23 QPoint &QPoint::operator*=(double factor);
24 QPoint &QPoint::operator*=(int factor);
25 QPoint &QPoint::operator+=(const QPoint &point);
26 QPoint &QPoint::operator-=(const QPoint &point);
27 QPoint &QPoint::operator/=(qreal divisor);

```

4.2 QLine

QLine 是一个直线类，封装了两个坐标点 (两点确定一条直线)

常用 API 如下：

```

1 // 构造函数
2 // 构造一个空对象
3 QLine::QLine();
4 // 构造一条直线，通过两个坐标点
5 QLine::QLine(const QPoint &p1, const QPoint &p2);
6 // 从点 (x1, y1) 到 (x2, y2)
7 QLine::QLine(int x1, int y1, int x2, int y2);
8
9 // 给直线对象设置坐标点
10 void QLine::setPoints(const QPoint &p1, const QPoint &p2);
11 // 起始点(x1, y1)，终点(x2, y2)
12 void QLine::setLine(int x1, int y1, int x2, int y2);
13 // 设置直线的起点坐标
14 void QLine::setP1(const QPoint &p1);
15 // 设置直线的终点坐标
16 void QLine::setP2(const QPoint &p2);
17
18 // 返回直线的起始点坐标
19 QPoint QLine::p1() const;
20 // 返回直线的终点坐标
21 QPoint QLine::p2() const;
22 // 返回值直线的中心点坐标, (p1() + p2()) / 2
23 QPoint QLine::center() const;
24
25 // 返回值直线起点的 x 坐标
26 int QLine::x1() const;
27 // 返回值直线终点的 x 坐标

```

```
28 int QLine::x2() const;
29 // 返回值直线起点的 y 坐标
30 int QLine::y1() const;
31 // 返回值直线终点的 y 坐标
32 int QLine::y2() const;
33
34 // 用给定的坐标点平移这条直线
35 void QLine::translate(const QPoint &offset);
36 void QLine::translate(int dx, int dy);
37 // 用给定的坐标点平移这条直线，返回平移之后的坐标点
38 QLine QLine::translated(const QPoint &offset) const;
39 QLine QLine::translated(int dx, int dy) const;
40
41 // 直线对象进行比较
42 bool QLine::operator!=(const QLine &line) const;
43 bool QLine::operator==(const QLine &line) const;
```

4.3 QSize

在 QT 中 QSize 类用来形容长度和宽度，常用的 API 如下：

```
1 // 构造函数
2 // 构造空对象，对象中的宽和高都是无效的
3 QSize::QSize();
4 // 使用宽和高构造一个有效对象
5 QSize::QSize(int width, int height);
6
7 // 设置宽度
8 void QSize::setWidth(int width)
9 // 设置高度
10 void QSize::setHeight(int height);
11
12 // 得到宽度
13 int QSize::width() const;
14 // 得到宽度的引用
15 int &QSize::rwidth();
16 // 得到高度
17 int QSize::height() const;
18 // 得到高度的引用
19 int &QSize::rheight();
20
21 // 交换高度和宽度的值
22 void QSize::transpose();
23 // 交换高度和宽度的值，返回交换之后的尺寸信息
24 QSize QSize::transposed() const;
25
26 // 进行算法运算：加减乘除
27 QSize &QSize::operator*=(qreal factor);
28 QSize &QSize::operator+=(const QSize &size);
29 QSize &QSize::operator-=(const QSize &size);
30 QSize &QSize::operator/=(qreal divisor);
```

4.4 QRect

在 Qt 中使用 QRect 类来描述一个矩形，常用的 API 如下：

```
1 // 构造函数
2 // 构造一个空对象
3 QRect::QRect();
4 // 基于左上角坐标，和右下角坐标构造一个矩形对象
5 QRect::QRect(const QPoint &topLeft, const QPoint &bottomRight);
6 // 基于左上角坐标，和 宽度，高度构造一个矩形对象
7 QRect::QRect(const QPoint &topLeft, const QSize &size);
8 // 通过 左上角坐标(x, y)，和 矩形尺寸(width, height) 构造一个矩形对象
9 QRect::QRect(int x, int y, int width, int height);
10
11 // 设置矩形的尺寸信息，左上角坐标不变
12 void QRect::setSize(const QSize &size);
13 // 设置矩形左上角坐标为(x,y)，大小为(width, height)
14 void QRect::setRect(int x, int y, int width, int height);
15 // 设置矩形宽度
16 void QRect::setWidth(int width);
17 // 设置矩形高度
18 void QRect::setHeight(int height);
19
20 // 返回值矩形左上角坐标
21 QPoint QRect::topLeft() const;
22 // 返回矩形右上角坐标
23 // 该坐标点值为: QPoint(left() + width() -1, top())
24 QPoint QRect::topRight() const;
25 // 返回矩形左下角坐标
26 // 该坐标点值为: QPoint(left(), top() + height() - 1)
27 QPoint QRect::bottomLeft() const;
28 // 返回矩形右下角坐标
29 // 该坐标点值为: QPoint(left() + width() -1, top() + height() - 1)
30 QPoint QRect::bottomRight() const;
31 // 返回矩形中心点坐标
32 QPoint QRect::center() const;
33
34 // 返回矩形上边缘y轴坐标
35 int QRect::top() const;
36 int QRect::y() const;
37 // 返回值矩形下边缘y轴坐标
38 int QRect::bottom() const;
39 // 返回矩形左边缘 x轴坐标
40 int QRect::x() const;
41 int QRect::left() const;
42 // 返回矩形右边缘x轴坐标
43 int QRect::right() const;
44
45 // 返回矩形的高度
46 int QRect::width() const;
47 // 返回矩形的宽度
48 int QRect::height() const;
49 // 返回矩形的尺寸信息
50 QSize QRect::size() const;
```

三、Qt中的信号槽

文章中主要介绍了 Qt 中的信号槽，主要内容包括：信号槽的本质，信号槽的关系，标准信号槽的使用，自定义信号槽的使用，信号槽的拓展，Lambda 表达式

1. 信号和槽概述

信号槽是 Qt 框架引以为豪的机制之一。所谓信号槽，实际就是观察者模式（发布 - 订阅模式）。当某个事件发生之后，比如，按钮检测到自己被点击了一下，它就会发出一个信号（signal）。这种发出是没有目的的，类似广播。如果有对象对这个信号感兴趣，它就会使用连接（connect）函数，意思是，将想要处理的信号和自己的一个函数（称为槽（slot））绑定来处理这个信号。也就是说，当信号发出时，被连接的槽函数会自动被回调。这就类似观察者模式：当发生了感兴趣的事件，某一个操作就会被自动触发。

1.1 信号的本质

信号是由于用户对窗口或控件进行了某些操作，导致窗口或控件产生了某个特定事件，这时候 Qt 对应的窗口类会发出某个信号，以此对用户的挑选做出反应。

因此根据上述的描述我们得到一个结论 - 信号的本质就是事件，比如：

- 按钮单击、双击
- 窗口刷新
- 鼠标移动、鼠标按下、鼠标释放
- 键盘输入

那么在 Qt 中信号是通过什么形式呈现给使用者的呢？

- 我们对哪个窗口进行操作，哪个窗口就可以捕捉到这些被触发的事件。
- 对于使用者来说触发了一个事件我们就可以得到 Qt 框架给我们发出的某个特定信号。
- 信号的呈现形式就是函数，也就是说某个事件产生了，Qt 框架就会调用某个对应的信号函数，通知使用者。

在 QT 中信号的发出者是某个实例化的类对象，对象内部可以进行相关事件的检测。

1.2 槽的本质

在 Qt 中槽函数是一类特殊的功能的函数，在编码过程中也可以作为类的普通成员函数来使用。之所以称之为槽函数是因为它们还有一个职责就是对 Qt 框架中产生的信号进行处理。

- | | |
|---|------------------------|
| 1 | 举个简单的例子： |
| 2 | |
| 3 | 女朋友说：“我肚子饿了！”，于是带她去吃饭。 |

上边例子中相当于女朋友发出了一个信号，我收到了信号并将其处理掉了。

实例对象	角色	描述
女朋友	信号发出者	信号携带的信息：我饿了
我	信号接收者	处理女朋友发射的信号：带她去吃饭

在 Qt 中槽函数的所有者也是某个类的实例对象。

1.3 信号和槽的关系

在 Qt 中信号和槽函数都是独立的个体，本身没有任何联系，但是由于某种特性需求我们可以将二者连接到一起，好比牛郎和织女想要相会必须要有喜鹊为他们搭桥一样。在 Qt 中我们需要使用 QObject类中的 connect 函数进二者的关联。



连接信号和槽的 connect() 函数原型如下，其中 PointerToMemberFunction 是一个指向函数地址的指针

```
1 QMetaObject::Connection QObject::connect(  
2     const QObject *sender, PointerToMemberFunction signal,  
3     const QObject *receiver, PointerToMemberFunction method,  
4     Qt::ConnectionType type = Qt::AutoConnection);  
5  
参数：  
6 - sender:    发出信号的对象  
7 - signal:    属于sender对象，信号是一个函数，这个参数的类型是函数  
8          指针，信号函数地址  
9 - receiver:  信号接收者  
10 - method:   属于receiver对象，当检测到sender发出了signal信号，  
11          receiver对象调用method方法，信号发出之后的处理动作  
12  
13 // 参数 signal 和 method 都是函数地址，因此简化之后的 connect() 如下：  
14 connect(const QObject *sender, &QObject::signal,  
15         const QObject *receiver, &QObject::method);
```

使用connect()进行信号槽连接的注意事项：

- connect函数相对于做了信号处理动作的注册
- 调用conenct函数的sender对象的信号并没有产生, 因此receiver对象的 method 也不会被调用
- method槽函数本质是一个回调函数, 调用的时机是信号产生之后, 调用是Qt框架来执行的
- connect中的sender和recever两个指针必须被实例化了, 否则conenct不会成功

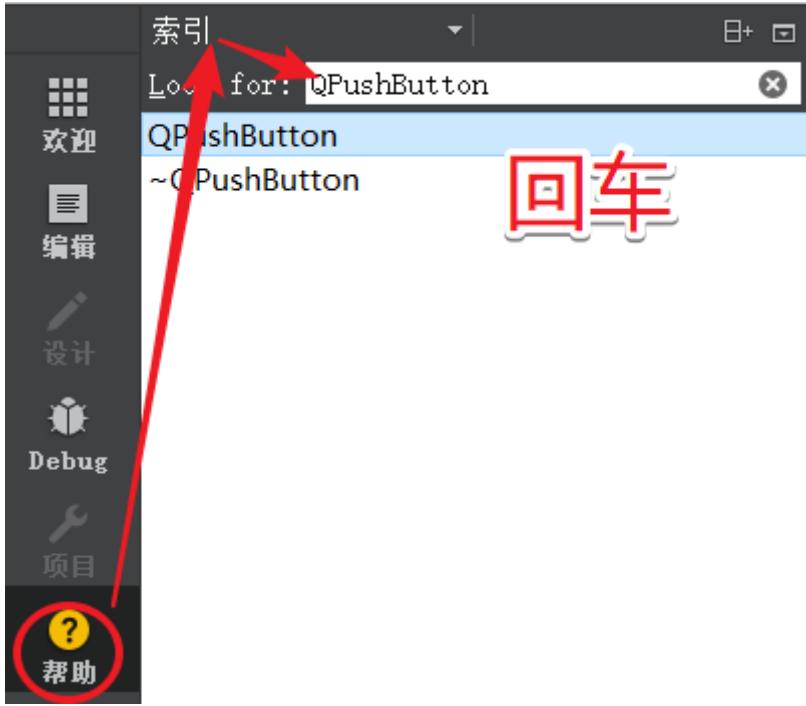
2. 标准信号槽使用

2.1 标准信号 / 槽

在 Qt 提供的很多标准类中都可以对用户触发的某些特定事件进行检测，因此当用户做了这些操作之后，事件被触发类的内部就会产生对应的信号，这些信号都是 Qt 类内部自带的，因此称之为标准信号。

同样的，在 Qt 的很多类内部为我提供了很多功能函数，并且这些函数也可以作为触发的信号的处理动作，有这类特性的函数在 Qt 中称之为标准槽函数。

系统自带的信号和槽通常如何查找呢，这个就需要利用帮助文档了，比如在帮助文档中查询按钮的点击信号，那么需要在帮助文档中输入 QPushButton



首先我们可以在 Contents 中寻找关键字 signals，信号的意思，但是我们发现并没有找到，这时候我们应该看当前类从父类继承下来了哪些信号

Contents 这个类中有哪些信息

Properties

Public Functions

Reimplemented Public Functions

Public Slots 槽函数

Protected Functions

Reimplemented Protected Functions

Detailed Description

没有信号，并不是没有信号，看当前类的父类

QPushButton Class

The QPushButton widget provides a command button

Header: #include <QPushButton>

qmake: QT += widgets

Inherits: QAbstractButton 父类

Inherited By: QCommandLinkButton

因此我们去他的父类 QAbstractButton 中就可以找到该关键字，点击 signals 索引到系统自带的信号有如下几个

Contents 先看信号和槽函数 再看公共成员

Properties

Public Functions 公共成员函数

Public Slots 槽函数

Signals 信号

Protected Functions

Reimplemented Protected Functions

Detailed Description

QAbstractButton Class

The `QAbstractButton` class is the abstract base class

More...

Header: `#include <QAbstractButton.h>`

qmake: `QT += widgets`

Inherits: `QWidget`

2.2 使用

掌握标准信号、槽的查找方式之后以及 `connect()` 函数的作用之后，下面通过一个简单的例子给大家讲解一下他们的使用方式。

- 1 功能实现：点击窗口上的按钮，关闭窗口
- 2 功能分析：
 - 按钮：信号发出者 \rightarrow `QPushButton` 类型
 - 窗口：信号的接收者和处理器 \rightarrow `QWidget` 类型

需要使用的标准信号槽函数

```
1 // 单击按钮发出的信号
2 [signal] void QAbstractButton::clicked(bool checked = false)
3 // 关闭窗口的槽函数
4 [slot] bool QWidget::close();
```

对于上边的需求只需要一句代码，只需要写一句代码就能实现了

```
1 // 单击按钮关闭窗口
2 connect(ui->closewindow, &QPushButton::clicked, this, &MainWindow::close);
```

`connect()`操作一般写在窗口的构造函数中，相当于在事件产生之前在qt框架中先进行注册，这样在程序运行过程中假设产生了按钮的点击事件，框架就会调用信号接收者对象对应的槽函数了，如果信号不产生，槽函数也就一直不会被调用。

3. 自定义信号槽使用

Qt 框架提供的信号槽在某些特定场景下是无法满足我们的项目需求的，因此我们还设计自己需要的信号和槽，同样还是使用 connect () 对自定义的信号槽进行连接。

如果想要在QT类中自定义信号槽，需要满足一些条件，并且有些事项也需要注意：

- 要编写新的类并且让其继承Qt的某些标准类
- 这个新的子类必须从QObject类或者是QObject子类进行派生
- 在定义类的头文件中加入 Q_OBJECT 宏

```
1 // 在头文件派生类的时候，首先像下面那样引入Q_OBJECT宏:  
2 class MyMainWindow : public QWidget  
3 {  
4     Q_OBJECT  
5     .....  
6 }
```

3.1 自定义信号

在 Qt 中信号的本质是事件，但是在框架中也是以函数的形式存在的，只不过信号对应的函数只有声明，没有定义。如果 Qt 中的标准信号不能满足我们的需求，可以在程序中进行信号的自定义，当自定义信号对应的事件产生之后，认为的将这个信号发射出去即可（其实质就是调用一下这个信号函数）。

1. 下边给大家阐述一下，自定义信号的要求和注意事项：
2. 信号是类的成员函数
3. 返回值必须是 void 类型
4. 信号的名字可以根据实际情况进行指定
5. 参数可以随意指定，信号也支持重载
6. 信号需要使用 signals 关键字进行声明，使用方法类似于public等关键字
7. 信号函数只需要声明，不需要定义(没有函数体实现)
8. 在程序中发射自定义信号：发送信号的本质就是调用信号函数
 - 习惯性在信号函数前加关键字：emit，但是可以省略不写
 - emit只是显示的声明一下信号要被发射了，没有特殊含义
 - 底层 emit == #define emit

```
1 // 举例：信号重载  
2 // Qt中的类想要使用信号槽机制必须要从QObject类派生(直接或间接派生都可以)  
3 class Test : public QObject  
4 {  
5     Q_OBJECT  
6     signals:  
7         void testsignal();  
8         // 参数的作用是数据传递，谁调用信号函数谁就指定实参  
9         // 实参最终会被传递给槽函数  
10        void testsignal(int a);  
11 }
```

3.2 自定义槽

槽函数就是信号的处理动作，在Qt中槽函数可以作为普通的成员函数来使用。如果标准槽函数提供的功能满足不了需求，可以自己定义槽函数进行某些特殊功能的实现。自定义槽函数和自定义的普通函数写法是一样的。

下边给大家阐述一下，自定义槽的要求和注意事项：

1. 返回值必须是 void 类型
2. 槽也是函数，因此也支持重载
3. 槽函数需要指定多少个参数，需要看连接的信号的参数个数
4. 槽函数的参数是用来接收信号传递的数据的，信号传递的数据就是信号的参数
 - 举例：
 - 信号函数: void testsig (int a, double b);
 - 槽函数: void testslot (int a, double b);
 - 总结：
 - 槽函数的参数应该和对应的信号的参数个数，从左到右类型依次对应
 - 信号的参数可以大于等于槽函数的参数个数 == 信号传递的数据被忽略了
 - 信号函数: void testsig (int a, double b);
 - 槽函数: void testslot (int a);
5. Qt中槽函数的类型是多样的

Qt中的槽函数可以是类的成员函数、全局函数、静态函数、Lambda表达式（匿名函数）
6. 槽函数可以使用关键字进行声明: slots (Qt5中slots可以省略不写)
 - public slots:
 - private slots: -> 这样的槽函数不能在类外部被调用
 - protected slots: -> 这样的槽函数不能在类外部被调用

```
1 // 槽函数书写格式举例
2 // 类中的这三个函数都可以作为槽函数来使用
3 class Test : public QObject
4 {
5     public:
6         void testSlot();
7         static void testFunc();
8
9     public slots:
10    void testSlot(int id);
11};
```

根据特定场景自定义信号槽：

```
1 还是上边的场景：
2 女朋友说：“我肚子饿了！”，于是我带她去吃饭。
```

```
1 // class Girlfriend
2 class Girlfriend : public QObject
3 {
4     Q_OBJECT
5     public:
6         explicit Girlfriend(QObject *parent = nullptr);
```

```

7 signals:
8     void hungry();           // 不能表达出想要吃什么
9     void hungry(QString msg); // 可以通过参数表达想要吃什么
10    };
11
12 // class Me
13 class Me : public QObject
14 {
15     Q_OBJECT
16 public:
17     explicit Me(QObject *parent = nullptr);
18
19 public slots:
20     // 槽函数
21     void eatMeal();           // 不能知道信号发出者要吃什么
22     void eatMeal(QString msg); // 可以知道信号发出者要吃什么
23
24 };

```

4. 信号槽拓展

4.1 信号槽使用拓展

- 一个信号可以连接多个槽函数，发送一个信号有多个处理动作
 - 需要写多个 connect () 连接
 - 槽函数的执行顺序是随机的，和 connect 函数的调用顺序没有关系
 - 信号的接收者可以是一个对象，也可以是多个对象
- 一个槽函数可以连接多个信号，多个不同的信号，处理动作是相同的
 - 需要写多个 connect () 连接
- 信号可以连接信号

信号接收者可以不处理接收的信号，而是继续发射新的信号，这相当于传递了数据，并没有对数据进行处理

```

1 connect(const QObject *sender, &QObject::signal,
2         const QObject *receiver, &QObject::signal->new);

```

- 信号槽是可以断开的

```

1 disconnect(const QObject *sender, &QObject::signal,
2            const QObject *receiver, &QObject::method);

```

4.2 信号槽的连接方式

1. Qt5 的连接方式

```

1 // 语法:
2 QMetaObject::Connection QObject::connect(
3     const QObject *sender, PointerToMemberFunction signal,
4     const QObject *receiver, PointerToMemberFunction method,
5     Qt::ConnectionType type = Qt::AutoConnection);
6
7 // 信号和槽函数也就是第2,4个参数传递的是地址, 编译器在编译过程中会对数据的正确性进行检测
8 connect(const QObject *sender, &QObject::signal,
9         const QObject *receiver, &QObject::method);

```

2. Qt4 的连接方式

这种旧的信号槽连接方式在 Qt5 中是支持的, 但是不推荐使用, 因为这种方式在进行信号槽连接的时候, 信号槽函数通过宏 SIGNAL 和 SLOT 转换为字符串类型。

因为信号槽函数的转换是通过宏来进行转换的, 因此传递到宏函数内部的数据不会被进行检测, 如果使用者传错了数据, 编译器也不会报错, 但实际上信号槽的连接已经不对了, 只有在程序运行起来之后才能发现问题, 而且问题不容易被定位。

```

1 // 语法:
2 [static] QMetaObject::Connection QObject::connect(
3     const QObject *sender, const char *signal,
4     const QObject *receiver, const char *method,
5     Qt::ConnectionType type = Qt::AutoConnection);
6
7 connect(const QObject *sender, SIGNAL(信号函数名(参数1, 参数2, ...)),
8         const QObject *receiver, SLOT(槽函数名(参数1, 参数2, ...)));

```

Qt4中声明槽函数必须要使用 slots 关键字, 不能省略。

3. 应用举例

```

1 场景描述:
2     - 我肚子饿了, 我要吃东西。
3 分析:
4     - 信号的发出者是我自己, 信号的接收者也是我自己

```

我们首先定义出一个 Qt 的类。

```

1 class Me : public QObject
2 {
3     Q_OBJECT
4     // Qt4中的槽函数必须这样声明, qt5中的关键字 slots 可以被省略
5 public slots:
6     void eat();
7     void eat(QString somthing);
8 signals:
9     void hungry();
10    void hungry(QString somthing);
11 };
12
13 // 基于上边的类写出解决方案
14 // 处理如下逻辑: 我饿了, 我要吃东西
15 // 分析: 信号的发出者是我自己, 信号的接收者也是我自己
16 Me m;

```

```
17 // Qt4处理方式
18 connect(&m, SIGNAL(eat()), &m, SLOT(hungury()));
19 connect(&m, SIGNAL(eat(QString)), &m, SLOT(hungury(QString)));
20
21 // Qt5处理方式
22 connect(&m, &Me::eat, &m, &Me::hungury); // error
```

Qt5 处理方式错误原因分析:

上边的写法之所以错误是因为这个类中信号槽都是重载过的, 信号和槽都是通过函数名去关联函数的地址, 但是这个同名函数对应两块不同的地址, 一个带参, 一个不带参, 因此编译器就不知道去关联哪块地址了, 所以如果我们在这种时候通过以上方式进行信号槽连接, 编译器就会报错。

解决方案:

可以通过定义函数指针的方式指定出函数的具体参数, 这样就可以确定函数的具体地址了。定义函数指针指向重载的某个信号或者槽函数, 在 connect () 函数中将函数指针名字作为实参就可以了。

```
1 // 举例:
2 void (Me::*func1)(QString) = &Me::eat; // func1指向带参的信号
3 void (Me::*func2)() = &Me::hungury; // func2指向不带参的槽函数
```

Qt 正确的处理方式:

```
1 // 定义函数指针指向重载的某一个具体的信号地址
2 void (Me::*mysignal)(QString) = &Me::eat;
3 // 定义函数指针指向重载的某一个具体的槽函数地址
4 void (Me::*myslot)() = &Me::hungury;
5 // 使用定义的函数指针完成信号槽的连接
6 connect(&m, mysignal, &m, myslot);
```

总结

- Qt4的信号槽连接方式因为使用了宏函数, 宏函数对用户传递的信号槽不会做错误检测, 容易出bug
- Qt5的信号槽连接方式, 传递的是信号槽函数的地址, 编译器会做错误检测, 减少了bug的产生
- 当信号槽函数被重载之后, Qt4的信号槽连接方式不受影响
- 当信号槽函数被重载之后, Qt5中需要给被重载的信号或者槽定义函数指针

5. Lambda 表达式

Lambda 表达式是 C++ 11 最重要也是最常用的特性之一, 是现代编程语言的一个特点, 简洁, 提高了代码的效率并且可以使程序更加灵活, Qt 是完全支持 c++ 语法的, 因此在 Qt 中也可以使用 Lambda 表达式。

5.1 语法格式

Lambda 表达式就是一个匿名函数, 语法格式如下:

```
1 [capture](params) opt -> ret {body;};
2   - capture: 捕获列表
3   - params: 参数列表
4   - opt: 函数选项
5   - ret: 返回值类型
6   - body: 函数体
```

关于 Lambda 表达式的细节介绍:

1. 捕获列表: 捕获一定范围内的变量

- [] - 不捕捉任何变量
- [&] - 捕获外部作用域中所有变量, 并作为引用在函数体内使用 (按引用捕获)
- [=] - 捕获外部作用域中所有变量, 并作为副本在函数体内使用 (按值捕获)
 - 拷贝的副本在匿名函数体内部是只读的
- [=, &foo] - 按值捕获外部作用域中所有变量, 并按照引用捕获外部变量 foo
- [bar] - 按值捕获 bar 变量, 同时不捕获其他变量
- [&bar] - 按引用捕获 bar 变量, 同时不捕获其他变量
- [this] - 捕获当前类中的 this 指针
 - 让 lambda 表达式拥有和当前类成员函数同样的访问权限
 - 如果已经使用了 & 或者 =, 默认添加此选项

2. 参数列表: 和普通函数的参数列表一样

3. opt 选项 -> 可以省略

- mutable: 可以修改按值传递进来的拷贝 (注意是能修改拷贝, 而不是值本身)
- exception: 指定函数抛出的异常, 如抛出整数类型的异常, 可以使用 throw();

4. 返回值类型:

- 标识函数返回值的类型, 当返回值为 void, 或者函数体中只有一处 return 的地方 (此时编译器可以自动推断出返回值类型) 时, 这部分可以省略

5. 函数体:

- 函数的实现, 这部分不能省略, 但函数体可以为空。

5.2 定义和调用

因为 Lambda 表达式是一个匿名函数, 因此是没有函数声明的, 直接在程序中进行代码的定义即可, 但是如果只定义匿名函数在程序执行过程中是不会被调用的。

```
1 // 匿名函数的定义, 程序执行这个匿名函数是不会被调用的
2 [](){
3     qDebug() << "hello, 我是一个lambda表达式...";
4 }
5
6 // 匿名函数的定义+调用:
7 int ret = [](int a) -> int
8 {
9     return a+1;
10 }(100); // 100是传递给匿名函数的参数
```

在 Lambda 表达式的捕获列表中也就是 [] 内部添加不同的关键字, 就可以在函数体中使用外部变量了。

```

1 // 在匿名函数外部定义变量
2 int a=100, b=200, c=300;
3 // 调用匿名函数
4 []() {
5     // 打印外部变量的值
6     qDebug() << "a:" << a << ", b: " << b << ", c:" << c; // error, 不能使用
7     // 任何外部变量
8 }
9 [&]() {
10    qDebug() << "hello, 我是一个lambda表达式...";
11    qDebug() << "使用引用的方式传递数据: ";
12    qDebug() << "a+1:" << a++ << ", b+c= " << b+c;
13 }()
14
15 // 值拷贝的方式使用外部数据
16 [=](int m, int n)mutable{
17     qDebug() << "hello, 我是一个lambda表达式...";
18     qDebug() << "使用拷贝的方式传递数据: ";
19     // 拷贝的外部数据在函数体内部是只读的，如果不添加 mutable 关键字是不能修改这些只读
20     // 数据的值的
21     // 添加 mutable 允许修改的数据是拷贝到函数内部的副本，对外部数据没有影响
22     qDebug() << "a+1:" << a++ << ", b+c= " << b+c;
23     qDebug() << "m+1: " << ++m << ", n: " << n;
24 }();

```

四、Qt定时器类 QTimer

在进行窗口程序的处理过程中，经常要周期性的执行某些操作，或者制作一些动画效果，看似比较复杂的问题使用定时器就可以完美的解决这些问题，Qt 中提供了两种定时器方式一种是使用 Qt 中的事件处理函数这个在后续章节会给大家做细致的讲解，本节主要给大家介绍一下 Qt 中的定时器类 QTimer 的使用方法。

要使用它，只需创建一个 QTimer 类对象，然后调用其 start() 函数开启定时器，此后 QTimer 对象就会周期性的发出 timeout() 信号。我们先来了解一下这个类的相关 API。

1. public/slot function

```

1 // 构造函数
2 // 如果指定了父对象，创建的堆内存可以自动析构
3 QTimer::QTimer(QObject *parent = nullptr);
4
5 // 设置定时器时间间隔为 msec 毫秒
6 // 默认值是0，一旦窗口系统事件队列中的所有事件都已经被处理完，一个时间间隔为0的QTimer就会
7 // 触发
8 void QTimer::setInterval(int msec);
9 // 获取定时器的时间间隔，返回值单位：毫秒
10 int QTimer::interval() const;
11
12 // 根据指定的时间间隔启动或者重启定时器，需要调用 setInterval() 设置时间间隔
13 [slot] void QTimer::start();
14 // 启动或重新启动定时器，超时间隔为msec毫秒。
15 [slot] void QTimer::start(int msec);

```

```

15 // 停止定时器。
16 [slot] void QTimer::stop();
17
18 // 设置定时器精度
19 /*
20 参数：
21 - Qt::PreciseTimer -> 精确的精度，毫秒级
22 - Qt::CoarseTimer -> 粗糙的精度，和1毫秒的误差在5%的范围内，默认精度
23 - Qt::VeryCoarseTimer -> 非常粗糙的精度，精度在1秒左右
24 */
25 void QTimer::setTimerType(Qt::TimerType atype);
26 Qt::TimerType QTimer::timerType() const; // 获取当前定时器的精度
27
28 // 如果定时器正在运行，返回true；否则返回false。
29 bool QTimer::isActive() const;
30
31 // 判断定时器是否只触发一次
32 bool QTimer::isSingleShot() const;
33 // 设置定时器是否只触发一次，参数为true定时器只触发一次，为false定时器重复触发，默认为
34 // false
35 void QTimer::setSingleShot(bool singleShot);

```

2. signals

这个类的信号只有一个，当定时器超时时，该信号就会被发射出来。给这个信号通过 `connect()` 关联一个槽函数，就可以在槽函数中处理超时事件了。

```
1 [signal] void QTimer::timeout();
```

3. static public function

```

1 // 其他同名重载函数可以自己查阅帮助文档
2 /*
3 功能：在msec毫秒后发射一次信号，并且只发射一次
4 参数：
5   - msec: 在msec毫秒后发射信号
6   - receiver: 接收信号的对象地址
7   - method: 槽函数地址
8 */
9 [static] void QTimer::singleShot(
10     int msec, const QObject *receiver,
11     PointerToMemberFunction method);

```

4. 定时器使用举例

- 周期性定时器

```

1 // 创建定时器对象
2 QTimer* timer = new QTimer(this);
3
4 // 修改定时器对象的精度
5 timer->setTimerType(Qt::PreciseTimer);
6
7 // 按钮 loopBtn 的点击事件
8 // 点击按钮启动或者关闭定时器，定时器启动，周期性得到当前时间

```

```

9 connect(ui->loopBtn, &QPushButton::clicked, this, [=]()
10 {
11     // 启动定时器
12     if(timer->isActive())
13     {
14         timer->stop(); // 关闭定时器
15         ui->loopBtn->setText("开始");
16     }
17     else
18     {
19         ui->loopBtn->setText("关闭");
20         timer->start(1000); // 1000ms == 1s
21     }
22 });
23
24 connect(timer, &QTimer::timeout, this, [=]()
25 {
26     QTime tm = QTime::currentTime();
27     // 格式化当前得到的系统时间
28     QString tmstr = tm.toString("hh:mm:ss.zzz");
29     // 设置要显示的时间
30     ui->curTime->setText(tmstr);
31 });

```

- 一次性定时器

```

1 // 点击按钮 onceBtn 只发射一次信号
2 // 点击按钮一次，发射一个信号，得到某一个时间点的时间
3 connect(ui->onceBtn, &QPushButton::clicked, this, [=]()
4 {
5     // 获取2s以后的系统时间，不创建定时器对象，直接使用类的静态方法
6     QTimer::singleShot(2000, this, [=]()
7     {
8         QTime tm = QTime::currentTime();
9         // 格式化当前得到的系统时间
10        QString tmstr = tm.toString("hh:mm:ss.zzz");
11        // 设置要显示的时间
12        ui->onceTime->setText(tmstr);
13    });
14 });

```

五、Qt中的基础窗口类

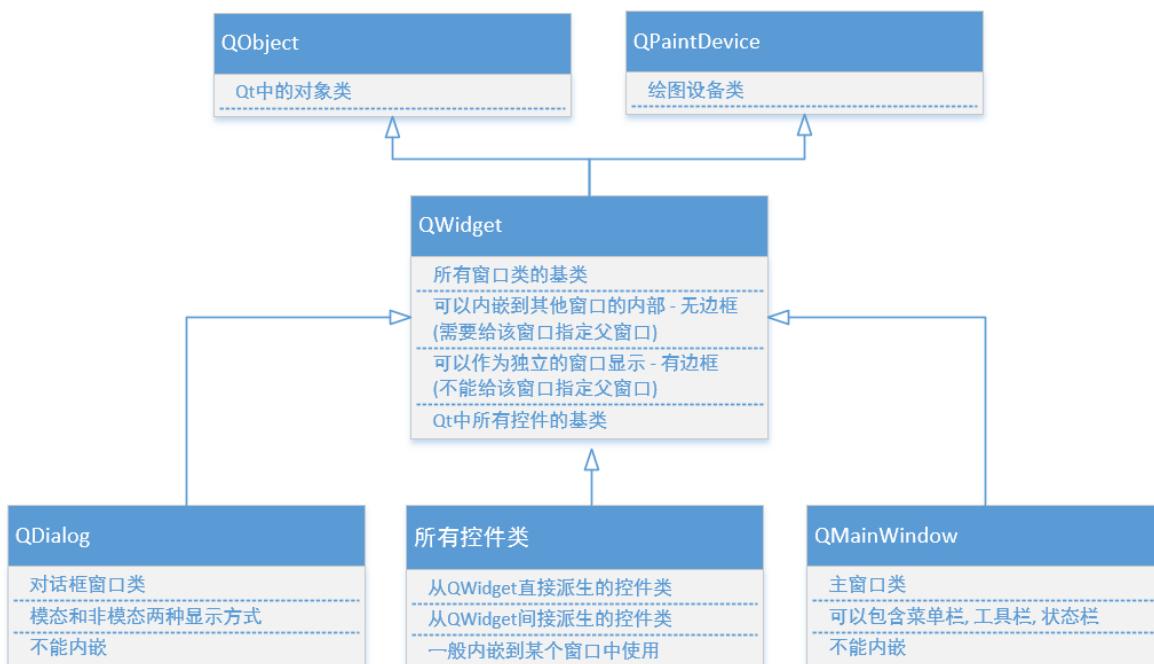
文章中主要介绍了 Qt 中常用的窗口类，主要内容包括：窗口类的基类QWidget, 对话框基类 QDialog, 带菜单栏工具栏状态栏的 QMainWindow, 消息对话框 QMessageBox, 文件对话框 QFileDialog, 字体对话框 QFontDialog, 颜色对话框 QColorDialog, 输入型对话框 QInputDialog, 进度条对话框 QProgressDialog, 资源文件

1. QWidget

QWidget 类是所有窗口类的父类 (控件类是也属于窗口类), 并且 QWidget 类的父类的 QObject, 也就意味着所有的窗口类对象只要指定了父对象, 都可以实现内存资源的自动回收。

在 1.1 Qt 入门章节中已经为大家介绍了 QWidget 的一些特点, 为了让大家能够对这个类有更深入的了解, 下面来说一说这个类常用的一些 API 函数。

关于这个窗口类的属性介绍, 请参考容器控件之 QWidget。



1.1 设置父对象

```
1 // 构造函数
2 QWidget::QWidget(QWidget *parent = nullptr, Qt::WindowFlags f =
3 Qt::WindowFlags());
4
5 // 公共成员函数
6 // 给当前窗口设置父对象
7 void QWidget::setParent(QWidget *parent);
8 void QWidget::setParent(QWidget *parent, Qt::WindowFlags f);
9 // 获取当前窗口的父对象, 没有父对象返回 nullptr
10 QWidget *QWidget::parentWidget() const;
```

1.2 窗口位置

```
1 //----- 窗口位置 -----
2 // 得到相对于当前窗口父窗口的几何信息, 边框也被计算在内
3 QRect QWidget::frameGeometry() const;
4 // 得到相对于当前窗口父窗口的几何信息, 不包括边框
5 const QRect &geometry() const;
6 // 设置当前窗口的几何信息(位置和尺寸信息), 不包括边框
7 void setGeometry(int x, int y, int w, int h);
8 void setGeometry(const QRect &);
9
10 // 移动窗口, 重新设置窗口的位置
11 void move(int x, int y);
12 void move(const QPoint &);
```

窗口位置设定和位置获取的测试代码如下:

```
1 // 获取当前窗口的位置信息
2 void MainWindow::on_positionBtn_clicked()
3 {
4     QRect rect = this->frameGeometry();
5     qDebug() << "左上角: " << rect.topLeft()
6             << "右上角: " << rect.topRight()
7             << "左下角: " << rect.bottomLeft()
8             << "右下角: " << rect.bottomRight()
9             << "宽度: " << rect.width()
10            << "高度: " << rect.height();
11 }
12
13 // 重新设置当前窗口的位置以及宽度，高度
14 void MainWindow::on_geometryBtn_clicked()
15 {
16     int x = 100 + rand() % 500;
17     int y = 100 + rand() % 500;
18     int width = this->width() + 10;
19     int height = this->height() + 10;
20     setGeometry(x, y, width, height);
21 }
22
23 // 通过 move() 方法移动窗口
24 void MainWindow::on_moveBtn_clicked()
25 {
26     QRect rect = this->frameGeometry();
27     move(rect.topLeft() + QPoint(10, 20));
28 }
```

1.3 窗口尺寸

```
1 //----- 窗口尺寸 -----
2 // 获取当前窗口的尺寸信息
3 QSize size() const
4 // 重新设置窗口的尺寸信息
5 void resize(int w, int h);
6 void resize(const QSize &);
7 // 获取当前窗口的最大尺寸信息
8 QSize maximumSize() const;
9 // 获取当前窗口的最小尺寸信息
10 QSize minimumSize() const;
11 // 设置当前窗口固定的尺寸信息
12 void QWidget::setFixedSize(const QSize &s);
13 void QWidget::setFixedSize(int w, int h);
14 // 设置当前窗口的最大尺寸信息
15 void setMaximumSize(const QSize &);
16 void setMaximumSize(int maxw, int maxh);
17 // 设置当前窗口的最小尺寸信息
18 void setMinimumSize(const QSize &);
19 void setMinimumSize(int minw, int minh);
20
21
22 // 获取当前窗口的高度
23 int height() const;
```

```
24 // 获取当前窗口的最小高度  
25 int minimumHeight() const;  
26 // 获取当前窗口的最大高度  
27 int maximumHeight() const;  
28 // 给窗口设置固定的高度  
29 void QWidget::setFixedHeight(int h);  
30 // 给窗口设置最大高度  
31 void setMaximumHeight(int maxh);  
32 // 给窗口设置最小高度  
33 void setMinimumHeight(int minh);  
34  
35 // 获取当前窗口的宽度  
36 int width() const;  
37 // 获取当前窗口的最小宽度  
38 int minimumWidth() const;  
39 // 获取当前窗口的最大宽度  
40 int maximumWidth() const;  
41 // 给窗口设置固定宽度  
42 void QWidget::setFixedwidth(int w);  
43 // 给窗口设置最大宽度  
44 void setMaximumwidth(int maxw);  
45 // 给窗口设置最小宽度  
46 void setMinimumwidth(int minw);
```

1.4 窗口标题和图标

```
1 //----- 窗口图标 -----  
2 // 得到当前窗口的图标  
3 QIcon windowIcon() const;  
4 // 构造图标对象，参数为图片的路径  
5 QIcon::(QIcon(const QString &fileName));  
6 // 设置当前窗口的图标  
7 void setwindowIcon(const QIcon &icon);  
8  
9 //----- 窗口标题 -----  
10 // 得到当前窗口的标题  
11 QString windowTitle() const;  
12 // 设置当前窗口的标题  
13 void setWindowTitle(const QString &);
```

1.5 信号

```
1 // QWidget::setContextMenuPolicy(Qt::ContextMenuPolicy policy);  
2 // 窗口的右键菜单策略 contextMenuPolicy() 参数设置为 Qt::CustomContextMenu，按下鼠标右键发射该信号  
3 [signal] void QWidget::customContextMenuRequested(const QPoint &pos);  
4 // 窗口图标发生变化，发射此信号  
5 [signal] void QWidget::windowIconChanged(const QIcon &icon);  
6 // 窗口标题发生变化，发射此信号  
7 [signal] void QWidget::windowTitleChanged(const QString &title);
```

基于窗口策略实现右键菜单具体操作请参考 Qt 右键菜单的添加和使用

1.6 槽函数

```

1 //----- 窗口显示 -----
2 // 关闭当前窗口
3 [slot] bool QWidget::close();
4 // 隐藏当前窗口
5 [slot] void QWidget::hide();
6 // 显示当前创建以及其子窗口
7 [slot] void QWidget::show();
8 // 全屏显示当前窗口，只对windows有效
9 [slot] void QWidget::showFullScreen();
10 // 窗口最大化显示，只对windows有效
11 [slot] void QWidget::showMaximized();
12 // 窗口最小化显示，只对windows有效
13 [slot] void QWidget::showMinimized();
14 // 将窗口回复为最大化/最小化之前的状态，只对windows有效
15 [slot] void QWidget::showNormal();
16
17 //----- 窗口状态 -----
18 // 判断窗口是否可用
19 bool QWidget::isEnabled() const; // 非槽函数
20 // 设置窗口是否可用，不可用窗口无法接收和处理窗口事件
21 // 参数true->可用，false->不可用
22 [slot] void QWidget::setEnabled(bool);
23 // 设置窗口是否可用，不可用窗口无法接收和处理窗口事件
24 // 参数true->不可用，false->可用
25 [slot] void QWidget::setDisabled(bool disable);
26 // 设置窗口是否可见，参数为true->可见，false->不可见
27 [slot] virtual void QWidget::setVisible(bool visible);

```

2. QDialog

2.1 常用 API

对话框类是 QWidget 类的子类，处理继承自父类的属性之外，还有一些自己所特有的属性，常用的一些 API 函数如下：

```

1 // 构造函数
2 QDialog::QDialog(QWidget *parent = nullptr, Qt::WindowFlags f =
Qt::WindowFlags());
3
4 // 模态显示窗口
5 [virtual slot] int QDialog::exec();
6 // 隐藏模态窗口，并且解除模态窗口的阻塞，将 exec() 的返回值设置为 QDialog::Accepted
7 [virtual slot] void QDialog::accept();
8 // 隐藏模态窗口，并且解除模态窗口的阻塞，将 exec() 的返回值设置为 QDialog::Rejected
9 [virtual slot] void QDialog::reject();
10 // 关闭对话框并将其结果代码设置为r。finished()信号将发出r;
11 // 如果r是QDialog::Accepted 或 QDialog::Rejected，则还将分别发出accept()或
Rejected()信号。
12 [virtual slot] void QDialog::done(int r);
13
14 [signal] void QDialog::accepted();
15 [signal] void QDialog::rejected();
16 [signal] void QDialog::finished(int result);

```

2.2 常用使用方法

```
1 | 场景介绍：  
2 | 1. 有两个窗口，主窗口和一个对话框子窗口  
3 | 2. 对话框窗口先显示，根据用户操作选择是否显示主窗口
```

2.2.1 关于对话框窗口类的操作

```
1 // 对话框窗口中三个普通按钮按下之后对应的槽函数  
2 void MyDialog::on_acceptBtn_clicked()  
3 {  
4     this->accept(); // exec()函数返回值为QDialog::Accepted  
5 }  
6  
7 void MyDialog::on_rejectBtn_clicked()  
8 {  
9     this->reject(); // exec()函数返回值为QDialog::Rejected  
10 }  
11  
12 void MyDialog::on_donBtn_clicked()  
13 {  
14     // exec()函数返回值为 done() 的参数，并根据参数发射出对应的信号  
15     this->done(666);  
16 }
```

2.2.2 根据用户针对对话框窗口的按钮操作，进行相应的逻辑处理。

```
1 // 创建对话框对象  
2 MyDialog dlg;  
3 int ret = dlg.exec();  
4 if(ret == QDialog::Accepted)  
5 {  
6     qDebug() << "accept button clicked...";  
7     // 显示主窗口  
8     MainWindow* w = new MainWindow;  
9     w->show();  
10 }  
11 else if(ret == QDialog::Rejected)  
12 {  
13     qDebug() << "reject button clicked...";  
14     // 不显示主窗口  
15     .....  
16     .....  
17 }  
18 else  
19 {  
20     // ret == 666  
21     qDebug() << "done button clicked...";  
22     // 根据需求进行逻辑处理  
23     .....  
24     .....  
25 }
```

3. QDialog 的子类 (标准对话框)

3.1 QMessageBox

QMessageBox 对话框类是 QDialog 类的子类，通过这个类可以显示一些简单的提示框，用于展示警告、错误、问题等信息。关于这个类我们只需要掌握一些静态方法的使用就可以了。

3.1.1 API - 静态函数

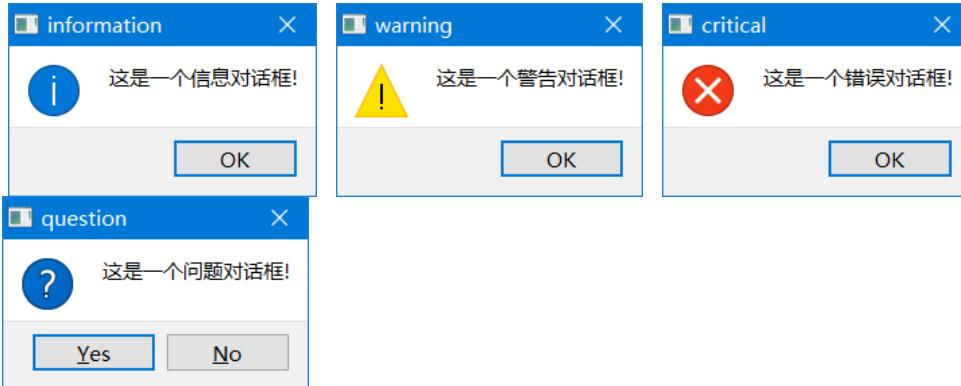
```
1 // 显示一个模态对话框，将参数 text 的信息展示到窗口中
2 [static] void QMessageBox::about(QWidget *parent, const QString &title,
3                                 const QString &text);
4
5 /*
6  * 参数：
7  * - parent: 对话框窗口的父窗口
8  * - title: 对话框窗口的标题
9  * - text: 对话框窗口中显示的提示信息
10 * - buttons: 对话框窗口中显示的按钮(一个或多个)
11 * - defaultButton
12     1. defaultButton指定按下Enter键时使用的按钮。
13     2. defaultButton必须引用在参数 buttons 中给定的按钮。
14     3. 如果defaultButton是QMessageBox::NoButton, QMessageBox会自动选择一个合适的
15        默认值。
16 */
17 // 显示一个信息模态对话框
18 [static] QMessageBox::StandardButton QMessageBox::information(
19     QWidget *parent, const QString &title,
20     const QString &text,
21     QMessageBox::StandardButtons buttons = Ok,
22     QMessageBox::StandardButton defaultButton = NoButton);
23
24 // 显示一个错误模态对话框
25 [static] QMessageBox::StandardButton QMessageBox::critical(
26     QWidget *parent, const QString &title,
27     const QString &text,
28     QMessageBox::StandardButtons buttons = Ok,
29     QMessageBox::StandardButton defaultButton = NoButton);
30
31 // 显示一个问题模态对话框
32 [static] QMessageBox::StandardButton QMessageBox::question(
33     QWidget *parent, const QString &title,
34     const QString &text,
35     QMessageBox::StandardButtons buttons = StandardButtons(Yes | No),
36     QMessageBox::StandardButton defaultButton = NoButton);
37
38 // 显示一个警告模态对话框
39 [static] QMessageBox::StandardButton QMessageBox::warning(
40     QWidget *parent, const QString &title,
41     const QString &text,
42     QMessageBox::StandardButtons buttons = Ok,
43     QMessageBox::StandardButton defaultButton = NoButton);
```

3.1.2 测试代码

```

1 void MainWindow::on_msgbox_clicked()
2 {
3     QMessageBox::about(this, "about", "这是一个简单的消息提示框!!!");
4     QMessageBox::critical(this, "critical", "这是一个错误对话框-critical..."); 
5     int ret = QMessageBox::question(this, "question",
6                                     "你要保存修改的文件内容吗???", 
7                                     QMessageBox::Save | QMessageBox::Cancel,
8                                     QMessageBox::Cancel);
9     if(ret == QMessageBox::Save)
10    {
11        QMessageBox::information(this, "information", "恭喜你保存成功了, o(*—*)o!!!");
12    }
13    else if(ret == QMessageBox::Cancel)
14    {
15        QMessageBox::warning(this, "warning", "你放弃了保存, TT—TT !!!");
16    }
17 }

```



3.2 QFileDialog

QFileDialog 对话框类是 QDialog 类的子类，通过这个类可以选择要打开 / 保存的文件或者目录。关于这个类我们只需要掌握一些静态方法的使用就可以了。

3.2.1 API - 静态函数

```

1 /*
2 通用参数:
3   - parent: 当前对话框窗口的父对象也就是父窗口
4   - caption: 当前对话框窗口的标题
5   - dir: 当前对话框窗口打开的默认目录
6   - options: 当前对话框窗口的一些可选项,枚举类型, 一般不需要进行设置, 使用默认值即可
7   - filter: 过滤器, 在对话框中只显示满足条件的文件, 可以指定多个过滤器, 使用 ;; 分隔
8   - 样式举例:
9   - Images (*.png *.jpg)
10  - Images (*.png *.jpg);;Text files (*.txt)
11  - selectedFilter: 如果指定了多个过滤器, 通过该参数指定默认使用哪一个, 不指定默认使用
第一个过滤器
12 */
13 // 打开一个目录, 得到这个目录的绝对路径
14 [static] QString QFileDialog::getExistingDirectory(
15             QWidget *parent = nullptr,
16             const QString &caption = QString(),

```

```

17         const QString &dir = QString(),
18         QFileDialog::options options = ShowDirsonly);
19
20 // 打开一个文件，得到这个文件的绝对路径
21 [static] QString QFileDialog::getOpenFileName(
22     QWidget *parent = nullptr,
23     const QString &caption = QString(),
24     const QString &dir = QString(),
25     const QString &filter = QString(),
26     QString *selectedFilter = nullptr,
27     QFileDialog::options options = Options());
28
29 // 打开多个文件，得到这多个文件的绝对路径
30 [static] QStringList QFileDialog::getOpenFileNames(
31     QWidget *parent = nullptr,
32     const QString &caption = QString(),
33     const QString &dir = QString(),
34     const QString &filter = QString(),
35     QString *selectedFilter = nullptr,
36     QFileDialog::options options = Options());
37
38 // 打开一个目录，使用这个目录来保存指定的文件
39 [static] QString QFileDialog::getSaveFileName(
40     QWidget *parent = nullptr,
41     const QString &caption = QString(),
42     const QString &dir = QString(),
43     const QString &filter = QString(),
44     QString *selectedFilter = nullptr,
45     QFileDialog::options options = Options());

```

3.2.2 测试代码

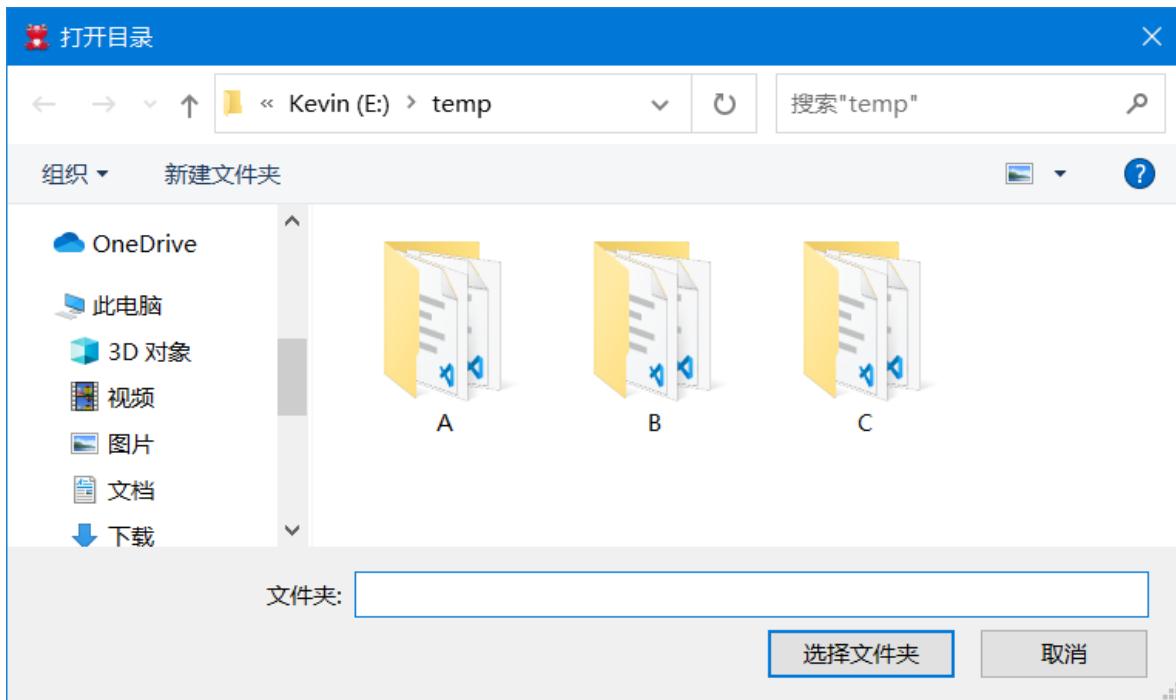
3.2.2.1 打开一个已存在的本地目录

```

1 void MainWindow::on_fileDlg_clicked()
2 {
3     QString dirName = QFileDialog::getExistingDirectory(this, "打开目录",
4 "e:\\temp");
5     QMessageBox::information(this, "打开目录", "您选择的目录是: " + dirName);
6 }

```

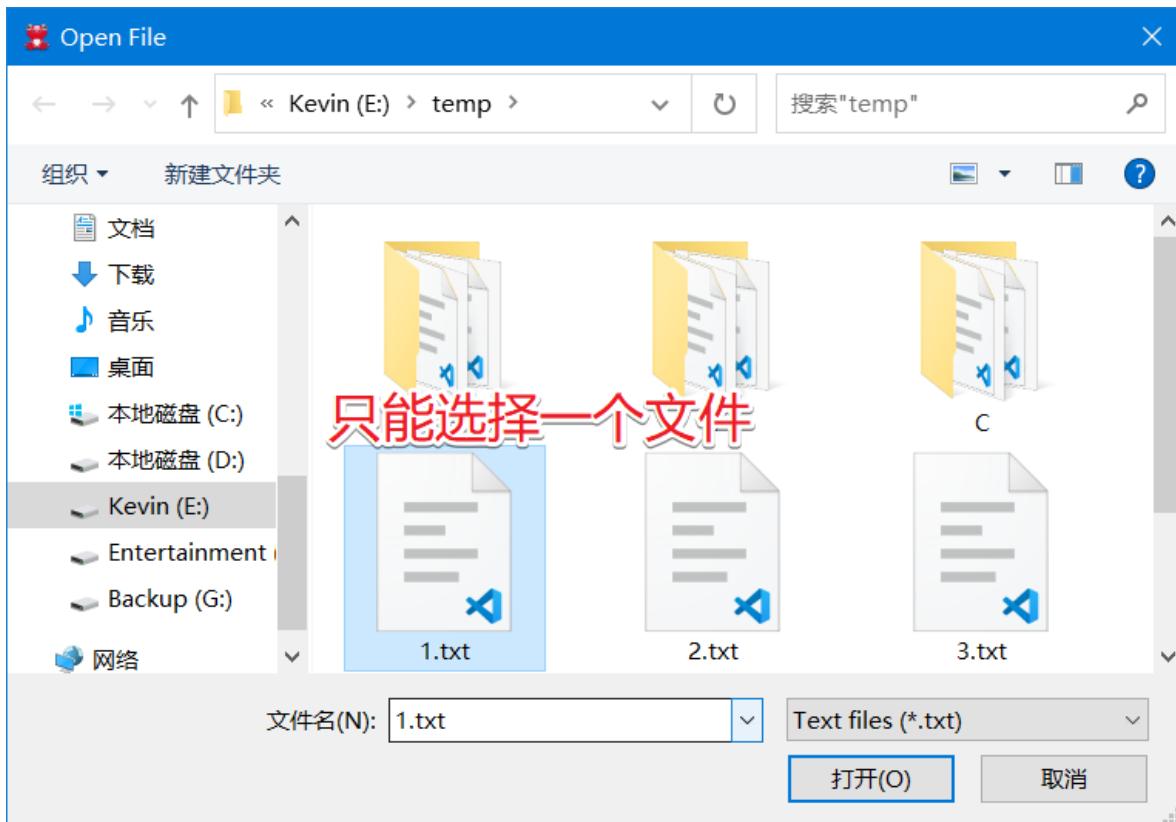
对话框效果如下:



3.2.2.2 打开一个本地文件

```
1 void MainWindow::on_fileDlg_clicked()
2 {
3     QString arg("Text files (*.txt)");
4     QString fileName = QFileDialog::getOpenFileName(
5         this, "Open File", "e:\\temp",
6         "Images (*.png *.jpg);;Text files (*.txt)", &arg);
7     QMessageBox::information(this, "打开文件", "您选择的文件是: " + fileName);
8 }
```

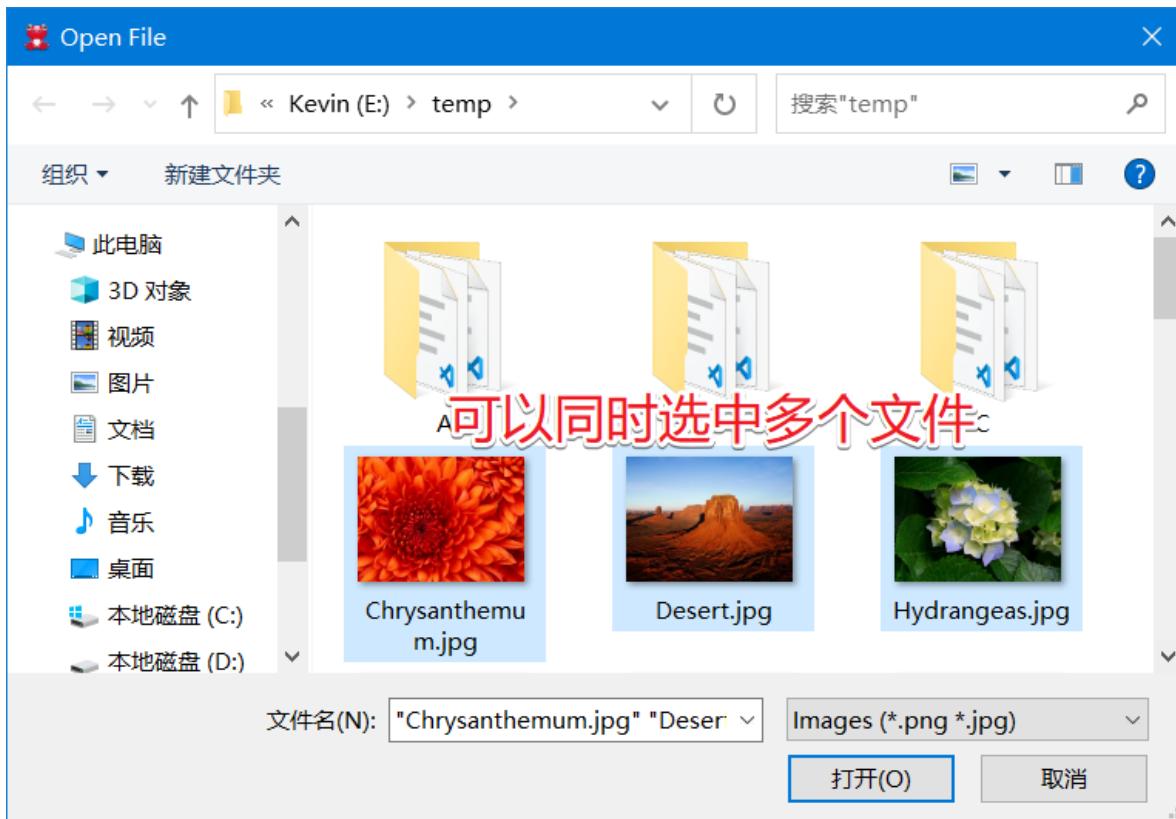
对话框效果如下:



3.2.2.3 打开多个本地文件

```
1 void MainWindow::on_fileDlg_clicked()
2 {
3     QStringList fileNames = QFileDialog::getOpenFileNames(
4         this, "Open File", "e:\\temp",
5         "Images (*.png *.jpg);;Text files (*.txt)");
6     QString names;
7     for(int i=0; i<fileNames.size(); ++i)
8     {
9         names += fileNames.at(i) + " ";
10    }
11    QMessageBox::information(this, "打开文件(s)", "您选择的文件是: " + names);
12 }
```

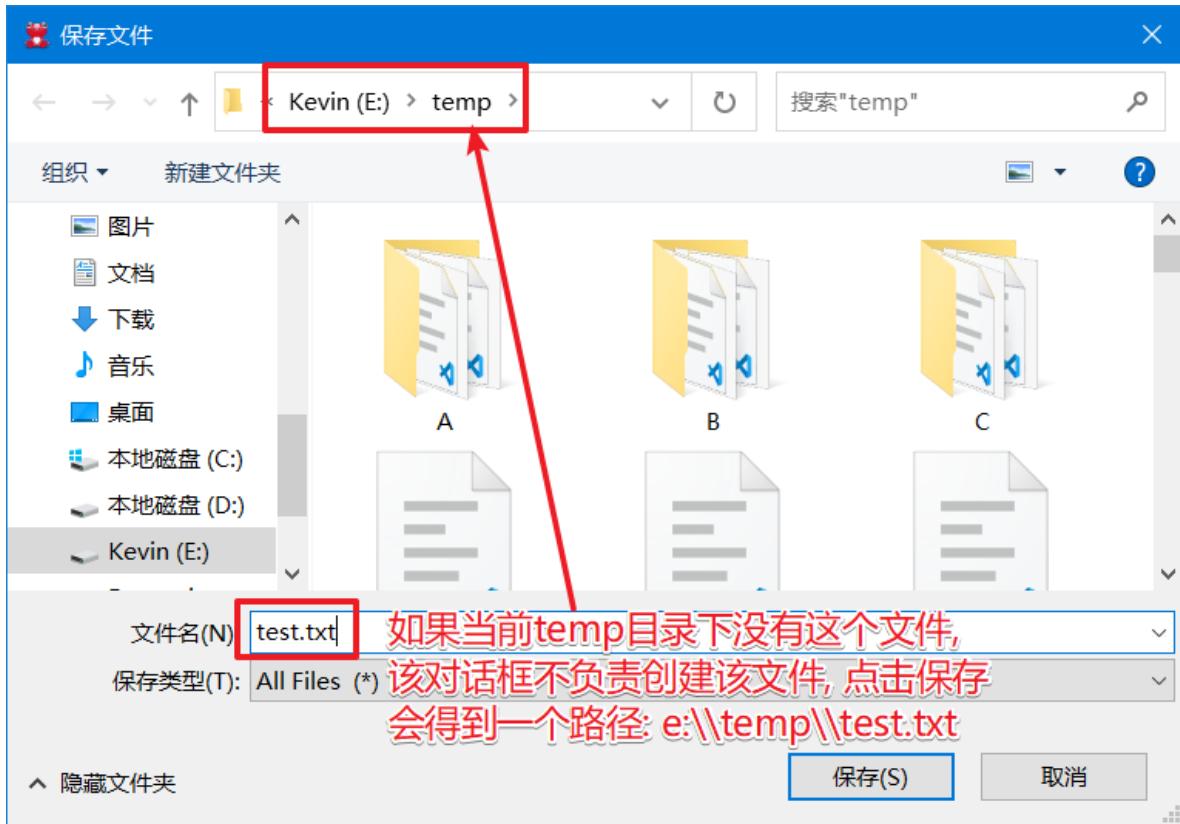
对话框效果如下:



3.2.2.4 打开保存文件对话框

```
1 void MainWindow::on_fileDlg_clicked()
2 {
3     QString fileName = QFileDialog::getSaveFileName(this, "保存文件",
4 "e:\\temp");
5     QMessageBox::information(this, "保存文件", "您指定的保存数据的文件是: " +
fileName);
```

对话框效果如下:



3.3 QFontDialog

QFontDialog 类是 QDialog 的子类，通过这个类我们可以得到一个进行字体属性设置的对话框窗口，和前边介绍的对话框类一样，我们只需要调用这个类的静态成员函数就可以得到想要的窗口了。

3.3.1 QFont 字体类

关于字体的属性信息，在 QT 框架中被封装到了一个叫 QFont 的类中，下边为大家介绍一下这个类的 API，了解一下关于这个类的使用。

```
1 // 构造函数
2 QFont::QFont();
3 /*
4  * 参数:
5  * - family: 本地字库中的字体名，通过 office 等文件软件可以查看
6  * - pointSize: 字体的字号
7  * - weight: 字体的粗细，有效范围为 0 ~ 99
8  * - italic: 字体是否倾斜显示，默认不倾斜
9 */
10 QFont::QFont(const QString &family, int pointSize = -1, int weight = -1,
11           bool italic = false);
12
13 // 设置字体
14 void QFont::setFamily(const QString &family);
15 // 根据字号设置字体大小
16 void QFont::setPointSize(int pointSize);
17 // 根据像素设置字体大小
18 void QFont::setPixelSize(int pixelsize);
19 // 设置字体的粗细程度，有效范围: 0 ~ 99
20 void QFont::setWeight(int weight);
```

```

20 // 设置字体是否加粗显示
21 void QFont::setBold(bool enable);
22 // 设置字体是否要倾斜显示
23 void QFont::setItalic(bool enable);
24
25 // 获取字体相关属性(一般规律: 去掉设置函数的 set 就是获取相关属性对应的函数名)
26 QString QFont::family() const;
27 bool QFont::italic() const;
28 int QFont::pixelSize() const;
29 int QFont::pointSize() const;
30 bool QFont::bold() const;
31 int QFont::weight() const;

```

如果一个 QFont 对象被创建，并且进行了初始化，我们可以将这个属性设置给某个窗口，或者设置给当前应用程序对象。

```

1 // QWidget 类
2 // 得到当前窗口使用的字体
3 const QWidget::QFont& QFont::font() const;
4 // 给当前窗口设置字体，只对当前窗口类生效
5 void QWidget::setFont(const QFont &);

6
7 // QApplication 类
8 // 得到当前应用程序对象使用的字体
9 [static] QFont QApplication::font();
10 // 给当前应用程序对象设置字体，作用于当前应用程序的所有窗口
11 [static] void QApplication::setFont(const QFont &font, const char
*className = nullptr);

```

3.3.2 QFontDialog 类的静态 API

```

1 /*
2 参数:
3   - ok: 传出参数，用于判断是否获得了有效字体信息，指定一个布尔类型变量地址
4   - initial: 字体对话框中默认选中并显示该字体信息，用于对话框的初始化
5   - parent: 字体对话框窗口的父对象
6   - title: 字体对话框的窗口标题
7   - options: 字体对话框选项，使用默认属性即可，一般不设置
8 */
9 [static] QFont QFontDialog::getFont(
10     bool *ok, const QFont &initial,
11     QWidget *parent = nullptr, const QString &title = QString(),
12     QFontDialog::FontDialogOptions options = FontDialogOptions());
13
14 [static] QFont QFontDialog::getFont(bool *ok, QWidget *parent = nullptr);

```

3.3.3 测试代码

通过字体对话框选择字体，并将选择的字体设置给当前窗口

```

1 void MainWindow::on_fontdlg_clicked()
2 {
3 #if 1
4     // 方式1
5     bool ok;

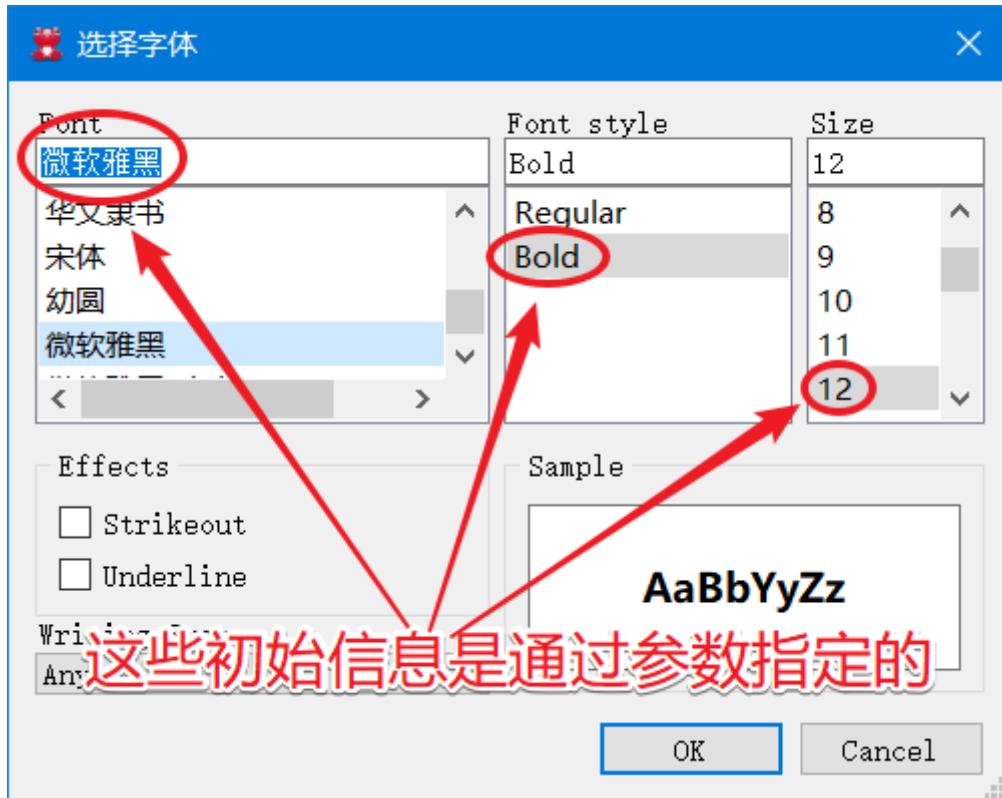
```

```

6     QFont ft = QFontDialog::getFont(
7             &ok, QFont("微软雅黑", 12, QFont::Bold), this, "选择字体");
8     qDebug() << "ok value is: " << ok;
9 #else
10    // 方式2
11    QFont ft = QFontDialog::getFont(NULL);
12 #endif
13    // 将选择的字体设置给当前窗口对象
14    this->setFont(ft);
15 }

```

字体对话框效果展示:



3.4 QColorDialog

QColorDialog 类是 QDialog 的子类，通过这个类我们可以得到一个选择颜色的对话框窗口，和前边介绍的对话框类一样，我们只需要调用这个类的静态成员函数就可以得到想要的窗口了。

3.4.1 颜色类 QColor

关于颜色的属性信息，在 QT 框架中被封装到了一个叫 QColor 的类中，下边为大家介绍一下这个类的 API，了解一下关于这个类的使用。

各种颜色都是基于红，绿，蓝这三种颜色调配而成的，并且颜色还可以进行透明度设置，默认是不透明的。

```

1 // 构造函数
2 QColor::QColor(Qt::GlobalColor color);
3 QColor::QColor(int r, int g, int b, int a = ...);
4 QColor::QColor();
5
6 // 参数设置 red, green, blue, alpha, 取值范围都是 0-255
7 void QColor::setRed(int red);           // 红色
8 void QColor::setGreen(int green);       // 绿色
9 void QColor::setBlue(int blue);         // 蓝色

```

```
10 void QColor::setAlpha(int alpha); // 透明度， 默认不透明(255)
11 void QColor::setRgb(int r, int g, int b, int a = 255);
12
13 int QColor::red() const;
14 int QColor::green() const;
15 int QColor::blue() const;
16 int QColor::alpha() const;
17 void QColor::getRgb(int *r, int *g, int *b, int *a = nullptr) const;
```

3.4.2 QFontDialog 类的静态 API

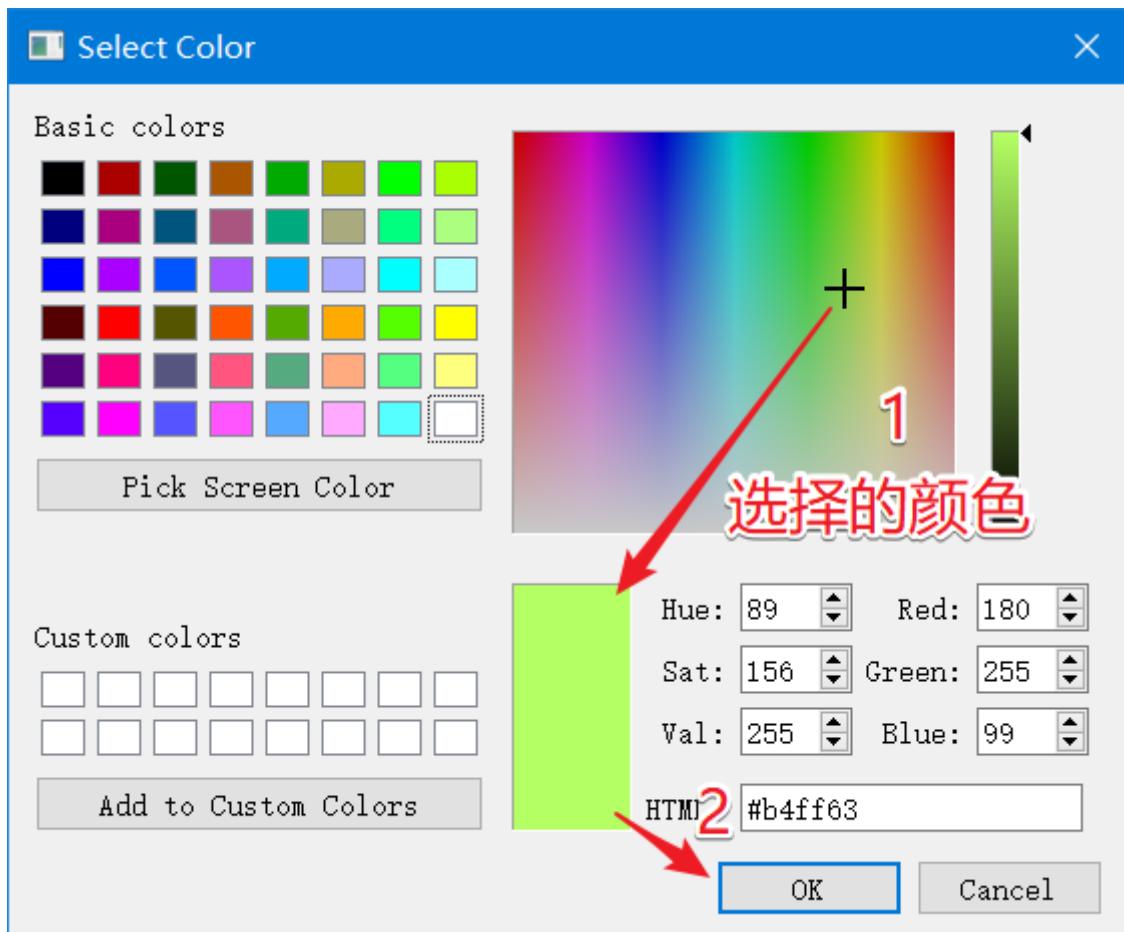
```
1 // 弹出颜色选择对话框，并返回选中的颜色信息
2 /*
3 参数：
4 - initial：对话框中默认选中的颜色，用于窗口初始化
5 - parent：给对话框窗口指定父对象
6 - title：对话框窗口的标题
7 - options：颜色对话框窗口选项，使用默认属性即可，一般不需要设置
8 */
9 [static] QColor QColorDialog::getColor(
10     const QColor &initial = Qt::white,
11     QWidget *parent = nullptr, const QString &title = QString(),
12     QColorDialog::ColorDialogOptions options = ColorDialogOptions());
```

3.4.3 测试代码

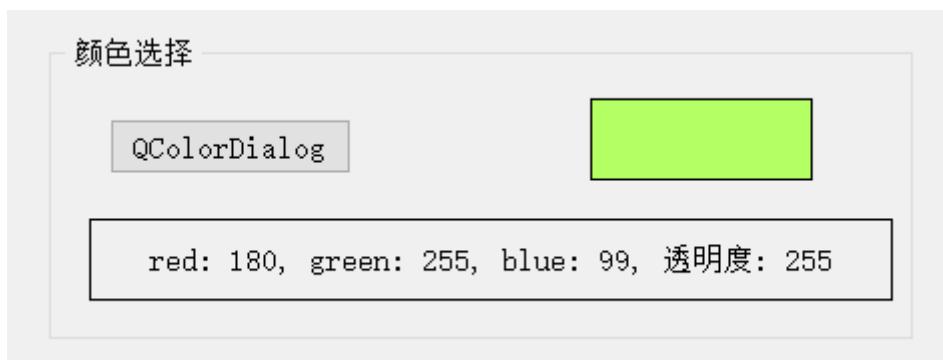
```
1 场景描述：
2 1. 在窗口上放一个标签控件
3 2. 通过颜色对话框选择一个颜色，将选中的颜色显示到标签控件上
4 3. 将选中的颜色的 RGBA 值分别显示出来
```

```
1 void MainWindow::on_colorDlg_clicked()
2 {
3     QColor color = QColorDialog::getColor();
4     QBrush brush(color);
5     QRect rect(0, 0, ui->color->width(), ui->color->height());
6     QPixmap pix(rect.width(), rect.height());
7     QPainter p(&pix);
8     p.fillRect(rect, brush);
9     ui->color->setPixmap(pix);
10    QString text = QString("red: %1, green: %2, blue: %3, 透明度: %4")
11    .arg(color.red()).arg(color.green()).arg(color.blue()).arg(color.alpha());
12    ui->colorLabel->setText(text);
13 }
```

颜色对话框窗口效果展示



测试代码效果展示



3.5 QInputDialog

QInputDialog 类是 QDialog 的子类，通过这个类我们可以得到一个输入对话框窗口，根据实际需求我们可以在这个输入窗口中输入整形，浮点型，字符串类型的数据，并且还可以显示下拉菜单供使用者选择。

和前边介绍的对话框类一样，我们只需要调用这个类的静态成员函数就可以得到想要的窗口了。

3.5.1 API - 静态函数

```

1 // 得到一个可以输入浮点数的对话框窗口，返回对话框窗口中输入的浮点数
2 /*
3 参数：
4 - parent: 对话框窗口的父窗口
5 - title: 对话框窗口显示的标题信息
6 - label: 对话框窗口中显示的文本信息(用于描述对话框的功能)
7 - value: 对话框窗口中显示的浮点值，默认为 0

```

```

8   - min: 对话框窗口支持显示的最小数值
9   - max: 对话框窗口支持显示的最大数值
10  - decimals: 浮点数的精度， 默认保留小数点以后1位
11  - ok: 传出参数， 用于判断是否得到了有效数据， 一般不会使用该参数
12  - flags: 对话框窗口的窗口属性， 使用默认值即可
13 */
14 [static] double QInputDialog::getDouble(
15     QWidget *parent, const QString &title,
16     const QString &label, double value = 0,
17     double min = -2147483647, double max = 2147483647,
18     int decimals = 1, bool *ok = nullptr,
19     Qt::WindowFlags flags = Qt::WindowFlags());
20
21 // 得到一个可以输入整形数的对话框窗口， 返回对话框窗口中输入的整形数
22 /*
23 参数：
24   - parent: 对话框窗口的父窗口
25   - title: 对话框窗口显示的标题信息
26   - label: 对话框窗口中显示的文本信息(用于描述对话框的功能)
27   - value: 对话框窗口中显示的整形值， 默认为 0
28   - min: 对话框窗口支持显示的最小数值
29   - max: 对话框窗口支持显示的最大数值
30   - step: 步长， 通过对话框提供的按钮调节数值每次增长/递减的量
31   - ok: 传出参数， 用于判断是否得到了有效数据， 一般不会使用该参数
32   - flags: 对话框窗口的窗口属性， 使用默认值即可
33 */
34 [static] int QInputDialog::getInt(
35     QWidget *parent, const QString &title,
36     const QString &label, int value = 0,
37     int min = -2147483647, int max = 2147483647,
38     int step = 1, bool *ok = nullptr,
39     Qt::WindowFlags flags = Qt::WindowFlags());
40
41 // 得到一个带下来菜单的对话框窗口， 返回选择的菜单项上边的文本信息
42 /*
43 参数：
44   - parent: 对话框窗口的父窗口
45   - title: 对话框窗口显示的标题信息
46   - label: 对话框窗口中显示的文本信息(用于描述对话框的功能)
47   - items: 字符串列表， 用于初始化窗口中的下拉菜单， 每个字符串对应一个菜单项
48   - current: 通过菜单项的索引指定显示下拉菜单中的哪个菜单项， 默认显示第一个(编号为0)
49   - editable: 设置菜单项上的文本信息是否可以进行编辑， 默认为true， 即可以编辑
50   - ok: 传出参数， 用于判断是否得到了有效数据， 一般不会使用该参数
51   - flags: 对话框窗口的窗口属性， 使用默认值即可
52   - inputMethodHints: 设置显示模式， 默认没有指定任何特殊显示格式， 显示普通文本字符串
53     - 如果有特殊需求， 可以参数帮助文档进行相关设置
54 */
55 [static] QString QInputDialog::getItem(
56     QWidget *parent, const QString &title,
57     const QString &label, const QStringList &items,
58     int current = 0, bool editable = true, bool *ok = nullptr,
59     Qt::WindowFlags flags = Qt::WindowFlags(),
60     Qt::InputMethodHints inputMethodHints = Qt::ImhNone);
61
62 // 得到一个可以输入多行数据的对话框窗口， 返回用户在窗口中输入的文本信息
63 /*
64 参数：
65   - parent: 对话框窗口的父窗口

```

```

66     - title: 对话框窗口显示的标题信息
67     - label: 对话框窗口中显示的文本信息(用于描述对话框的功能)
68     - text: 指定显示到多行输入框中的文本信息，默认是空字符串
69     - ok: 传出参数，用于判断是否得到了有效数据，一般不会使用该参数
70     - flags: 对话框窗口的窗口属性，使用默认值即可
71     - inputMethodHints: 设置显示模式， 默认没有指定任何特殊显示格式， 显示普通文本字符串
72         - 如果有特殊需求， 可以参数帮助文档进行相关设置
73 */
74 [static] QString QInputDialog::getMultiLineText(
75     QWidget *parent, const QString &title, const QString &label,
76     const QString &text = QString(), bool *ok = nullptr,
77     Qt::WindowFlags flags = Qt::WindowFlags(),
78     Qt::InputMethodHints inputMethodHints = Qt::ImhNone);
79
80 // 得到一个可以输入单行信息的对话框窗口， 返回用户在窗口中输入的文本信息
81 /*
82 参数：
83     - parent: 对话框窗口的父窗口
84     - title: 对话框窗口显示的标题信息
85     - label: 对话框窗口中显示的文本信息(用于描述对话框的功能)
86     - mode: 指定单行编辑框中数据的反馈模式， 是一个 QLineEdit::EchoMode 类型的枚举值
87         - QLineEdit::Normal: 显示输入的字符。这是默认值
88         - QLineEdit::NoEcho: 不要展示任何东西。这可能适用于连密码长度都应该保密的密码。
89         - QLineEdit::Password: 显示与平台相关的密码掩码字符，而不是实际输入的字符。
90         - QLineEdit::PasswordEchoOnEdit: 在编辑时按输入显示字符，否则按密码显示字符。
91     - text: 指定显示到单行输入框中的文本信息， 默认是空字符串
92     - ok: 传出参数，用于判断是否得到了有效数据，一般不会使用该参数
93     - flags: 对话框窗口的窗口属性， 使用默认值即可
94     - inputMethodHints: 设置显示模式， 默认没有指定任何特殊显示格式， 显示普通文本字符串
95         - 如果有特殊需求， 可以参数帮助文档进行相关设置
96 */
97 [static] QString QInputDialog::getText(
98     QWidget *parent, const QString &title, const QString &label,
99     QLineEdit::EchoMode mode = QLineEdit::Normal,
100    const QString &text = QString(), bool *ok = nullptr,
101    Qt::WindowFlags flags = Qt::WindowFlags(),
102    Qt::InputMethodHints inputMethodHints = Qt::ImhNone);

```

3.5.2 测试代码

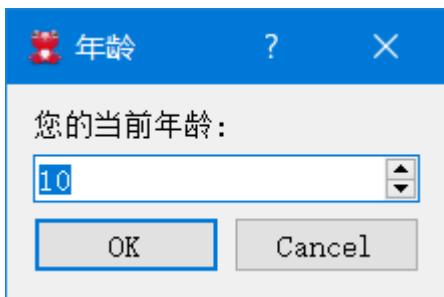
3.5.2.1 整形输入框

```

1 void MainWindow::on_inputdlg_clicked()
2 {
3     int ret = QInputDialog::getInt(this, "年龄", "您的当前年龄：", 10, 1, 100,
4                                     2);
5     QMessageBox::information(this, "年龄", "您的当前年龄：" +
6     QString::number(ret));
7 }

```

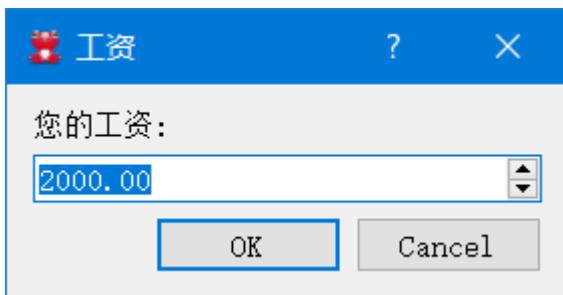
窗口效果展示:



3.5.2.2 浮点型输入框

```
1 void MainWindow::on_inputdlg_clicked()
2 {
3     double ret = QInputDialog::getDouble(this, "工资", "您的工资: ", 2000,
4     1000, 6000, 2);
5     QMessageBox::information(this, "工资", "您的当前工资: " +
6     QString::number(ret));
7 }
```

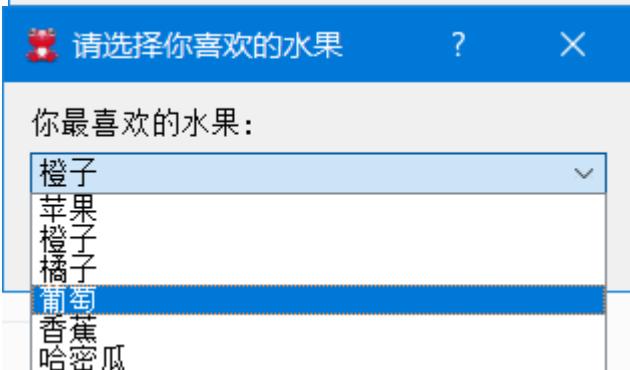
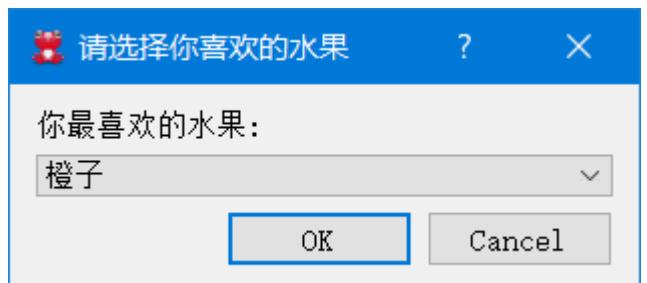
窗口效果展示:



3.5.2.3 带下拉菜单的输入框

```
1 void MainWindow::on_inputdlg_clicked()
2 {
3     QStringList items;
4     items << "苹果" << "橙子" << "橘子" << "葡萄" << "香蕉" << "哈密瓜";
5     QString item = QInputDialog::getItem(this, "请选择你喜欢的水果", "你最喜欢的水
6     果:", items, 1, false);
7     QMessageBox::information(this, "水果", "您最喜欢的水果是: " + item);
8 }
```

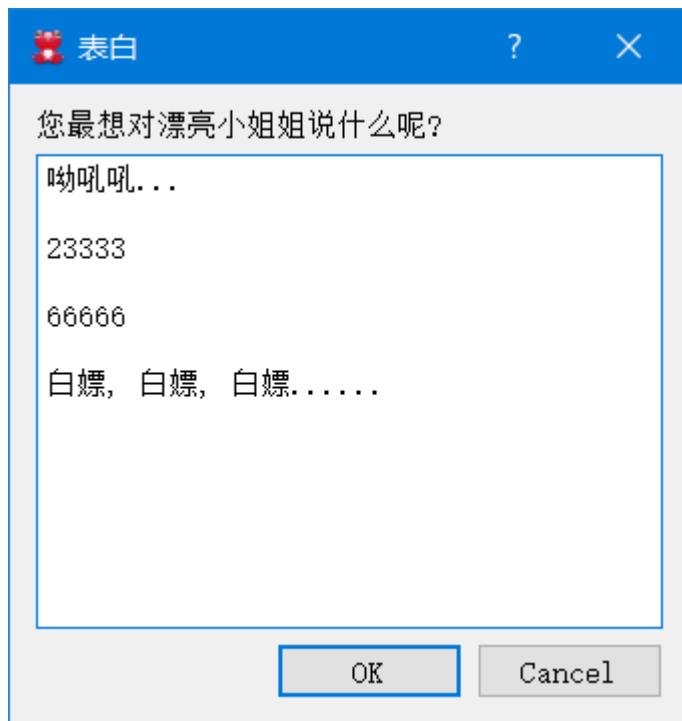
窗口效果展示:



3.5.2.4 多行字符串输入框

```
1 void MainWindow::on_inputdlg_clicked()
2 {
3     QString info = QInputDialog::getMultiLineText(this, "表白", "您最想对漂亮小
4         姐姐说什么呢?", "呦吼吼...");  
5     QMessageBox::information(this, "知心姐姐", "您最想对小姐姐说: " + info);  
6 }
```

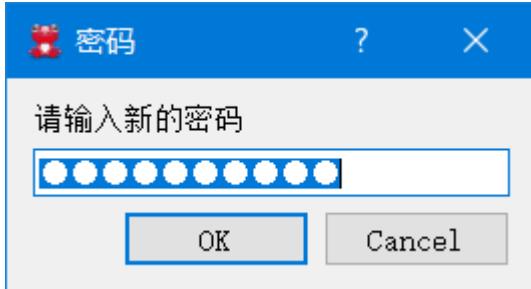
窗口效果展示:



3.5.2.5 单行字符串输入框

```
1 void MainWindow::on_inputDlg_clicked()
2 {
3     QString text = QInputDialog::getText(this, "密码", "请输入新的密码",
4                                         QLineEdit::Password, "helloworld");
5     QMessageBox::information(this, "密码", "您设置的密码是：" + text);
6 }
```

窗口效果展示:



3.6 QProgressDialog

QProgressDialog 类是 QDialog 的子类，通过这个类我们可以得到一个带进度条的对话框窗口，这种类型的对话框窗口一般常用于文件拷贝、数据传输等实时交互的场景中。

3.6.1 常用 API

```
1 // 构造函数
2 /*
3 参数:
4 - labelText: 对话框中显示的提示信息
5 - cancelButtonText: 取消按钮上显示的文本信息
6 - minimum: 进度条最小值
7 - maximum: 进度条最大值
8 - parent: 当前窗口的父对象
9 - f: 当前进度窗口的flag属性, 使用默认属性即可, 无需设置
10 */
11 QProgressDialog::QProgressDialog(
12     QWidget *parent = nullptr,
13     Qt::WindowFlags f = Qt::windowFlags());
14
15 QProgressDialog::QProgressDialog(
16     const QString &labelText, const QString &cancelButtonText,
17     int minimum, int maximum, QWidget *parent = nullptr,
18     Qt::WindowFlags f = Qt::windowFlags());
19
20
21 // 设置取消按钮显示的文本信息
22 [slot] void QProgressDialog::setCancelButtonText(const QString
&cancelButtonText);
23
24 // 公共成员函数和槽函数
25 QString QProgressDialog::labelText() const;
26 void QProgressDialog::setLabelText(const QString &text);
27
28 // 得到进度条最小值
29 int QProgressDialog::minimum() const;
```

```

30 // 设置进度条最小值
31 void QProgressDialog::setMinimum(int minimum);
32
33 // 得到进度条最大值
34 int QProgressDialog::maximum() const;
35 // 设置进度条最大值
36 void QProgressDialog::setMaximum(int maximum);
37
38 // 设置进度条范围(最大和最小值)
39 [slot] void QProgressDialog::setRange(int minimum, int maximum);
40
41 // 得到进度条当前的值
42 int QProgressDialog::value() const;
43 // 设置进度条当前的值
44 void QProgressDialog::setValue(int progress);
45
46
47 bool QProgressDialog::autoReset() const;
48 // 当value() = maximum()时, 进程对话框是否调用reset(), 此属性默认为true。
49 void QProgressDialog::setAutoReset(bool reset);
50
51
52 bool QProgressDialog::autoClose() const;
53 // 当value() = maximum()时, 进程对话框是否调用reset()并且隐藏, 此属性默认为true。
54 void QProgressDialog::setAutoClose(bool close);
55
56 // 判断用户是否按下了取消键, 按下了返回true, 否则返回false
57 bool wasCanceled() const;
58
59
60 // 重置进度条
61 // 重置进度对话框。wasCancelled()变为true, 直到进程对话框被重置。进度对话框被隐藏。
62 [slot] void QProgressDialog::cancel();
63 // 重置进度对话框。如果autoClose()为真, 进程对话框将隐藏。
64 [slot] void QProgressDialog::reset();
65
66 // 信号
67 // 当单击cancel按钮时, 将发出此信号。默认情况下, 它连接到cancel()槽。
68 [signal] void QProgressDialog::canceled();
69
70 // 设置窗口的显示状态(模态, 非模态)
71 /*
72 参数:
73     Qt::NonModal -> 非模态
74     Qt::WindowModal -> 模态, 阻塞父窗口
75     Qt::ApplicationModal -> 模态, 阻塞应用程序中的所有窗口
76 */
77 void QWidget::setWindowModality(Qt::WindowModality windowModality);

```

3.6.2 测试代码

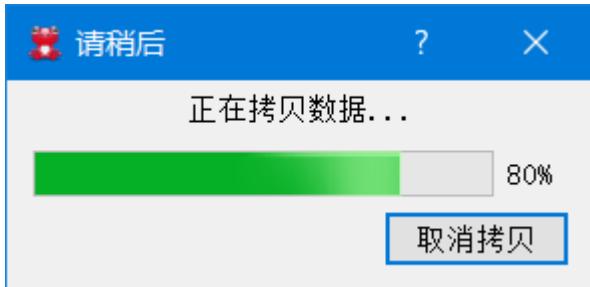
- 1 场景描述:
- 2 1. 基于定时器模拟文件拷贝的场景
- 3 2. 点击窗口按钮, 进度条窗口显示, 同时启动定时器
- 4 3. 通过定时器信号, 按照固定频率更新对话框窗口进度条
- 5 4. 当进度条当前值 == 最大值, 关闭定时器, 关闭并析构进度对话框

```

1 void MainWindow::on_progressDlg_clicked()
2 {
3     // 1. 创建进度条对话框窗口对象
4     QProgressDialog *progress = new QProgressDialog(
5             "正在拷贝数据...", "取消拷贝", 0, 100, this);
6     // 2. 初始化并显示进度条窗口
7     progress->setWindowTitle("请稍后");
8     progress->setWindowModality(Qt::WindowModal);
9     progress->show();
10
11    // 3. 更新进度条
12    static int value = 0;
13    QTimer *timer = new QTimer;
14    connect(timer, &QTimer::timeout, this, [=]()
15    {
16        progress->setValue(value);
17        value++;
18        // 当value > 最大值的时候
19        if(value > progress->maximum())
20        {
21            timer->stop();
22            value = 0;
23            delete progress;
24            delete timer;
25        }
26    });
27
28    connect(progress, &QProgressDialog::canceled, this, [=]()
29    {
30        timer->stop();
31        value = 0;
32        delete progress;
33        delete timer;
34    });
35
36    timer->start(50);
37 }

```

进度窗口效果展示:



4. QMainWindow

QMainWindow 是标准基础窗口中结构最复杂的窗口，其组成如下:

- 提供了菜单栏, 工具栏, 状态栏, 停靠窗口
- 菜单栏: 只能有一个, 位于窗口的最上方

- 工具栏：可以有多个，默认提供了一个，窗口的上下左右都可以停靠
- 状态栏：只能有一个，位于窗口最下方
- 停靠窗口：可以有多个，默认没有提供，窗口的上下左右都可以停靠



4.1 菜单栏

4.1.1 添加菜单项

关于顶级菜单可以直接在 UI 窗口中双击，直接输入文本信息即可，对应子菜单项也可以通过先双击在输入的方式完成添加，但是这种方式不支持中文的输入。



4.1.2 常用的添加方式

一般情况下，我们都是先在外面创建出 QAction 对象，然后再将其拖拽到某个菜单下边，这样子菜单项的添加就完成了。



4.1.3 通过代码的方式添加菜单或者菜单项

```
1 // 给菜单栏添加菜单
2 QAction *QMenuBar::addMenu(QMenu *menu);
3 QMenu *QMenuBar::addMenu(const QString &title);
4 QMenu *QMenuBar::addMenu(const QIcon &icon, const QString &title);
5
6 // 给菜单对象添加菜单项(QAction)
7 QAction *QMenu::addAction(const QString &text);
8 QAction *QMenu::addAction(const QIcon &icon, const QString &text);
9
10 // 添加分割线
11 QAction *QMenu::addSeparator();
```

4.1.4 菜单项 QAction 事件的处理

单击菜单项，该对象会发出一个信号

```
1 // 点击QAction对象发出该信号
2 [signal] void QAction::triggered(bool checked = false);
```

示例代码

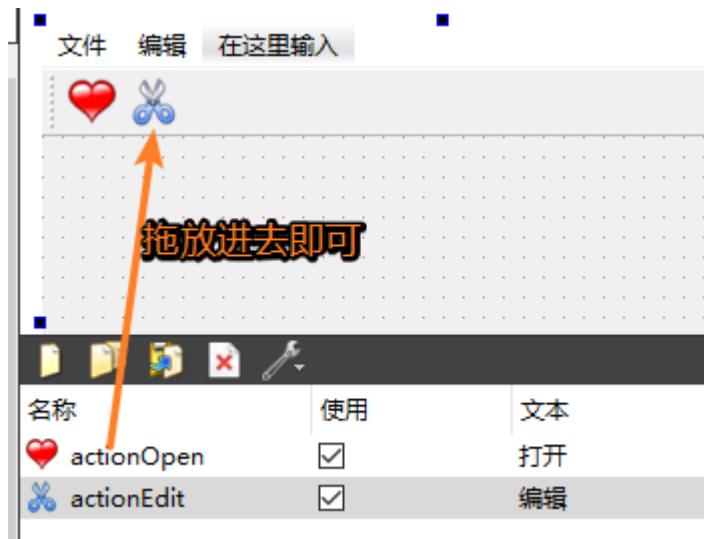
```
1 // save_action 是某个菜单项对象名，点击这个菜单项会弹出一个对话框
2 connect(ui->save_action, &QAction::triggered, this, [=]()
3 {
4     QMessageBox::information(this, "Triggered", "我是菜单项，你不要调戏
我...");
5});
```

4.2 工具栏

4.2.1 添加工具按钮

窗口中的工具栏我们经常见到，并不会为此感到陌生，那么如何往工具栏中添加工具按钮呢？一共有两种方式，这里依次为大家进行介绍。

方式 1：先创建 QAction 对象，然后拖拽到工具栏中，和添加菜单项的方式相同



方式 2：如果不通过 UI 界面直接操作，那么就需要调用相关的 API 函数了

```

1 // 在QMainWindow窗口中添加工具栏
2 void QMainWindow::addToolBar(Qt::ToolBarArea area, QToolBar *toolbar);
3 void QMainWindow::addToolBar(QToolBar *toolbar);
4 QToolBar *QMainWindow::addToolBar(const QString &title);
5
6 // 将Qt控件放到工具栏中
7 // 工具栏类: QToolBar
8 // 添加的对象只要是QWidget或者启子类都可以被添加
9 QAction *QToolBar::addWidget(QWidget *widget);
10
11 // 添加QAction对象
12 QAction *QToolBar::addAction(const QString &text);
13 QAction *QToolBar::addAction(const QIcon &icon, const QString &text);
14
15 // 添加分隔线
16 QAction *QToolBar::addSeparator()

```

通过代码的方式对工具栏进行操作

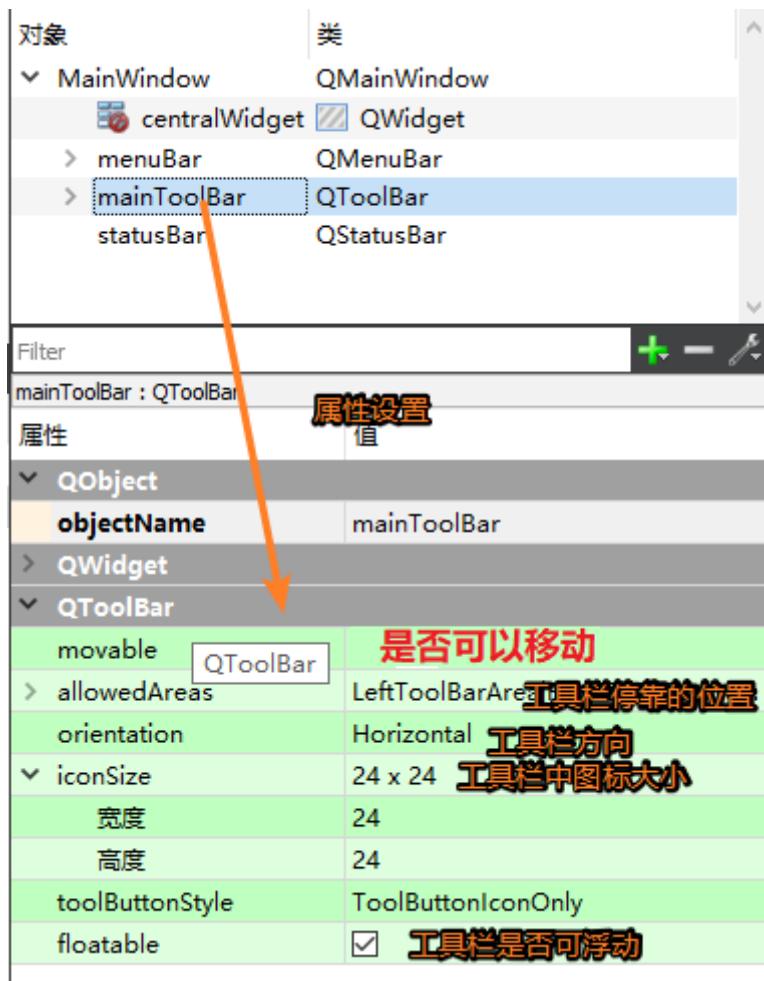
```

1 MainWindow::MainWindow(QWidget *parent)
2     : QMainWindow(parent)
3     , ui(new Ui::MainWindow)
4 {
5     ui->setupUi(this);
6
7     // 添加第二个工具栏
8     QToolBar* toolbar = new QToolBar("toolbar");
9     this->addToolBar(Qt::LeftToolBarArea, toolbar);
10
11    // 给工具栏添加按钮和单行输入框
12    ui->toolBar->addWidget(new QPushButton("搜索"));
13    QLineEdit* edit = new QLineEdit;
14    edit->setMaximumWidth(200);
15    edit->setFixedWidth(100);
16    ui->toolBar->addWidget(edit);
17    // 添加QAction类型的菜单项
18    ui->toolBar->addAction(QIcon(":/er-dog"), "二狗子");
19 }

```

4.2.2 工具栏的属性设置

在 UI 窗口的树状列表中，找到工具栏节点，就可以到的工具栏的属性设置面板了，这样就可以根据个人需求对工具栏的属性进行设置和修改了。



在 Qt 控件的属性窗口中对应了一些属性，这些属性大部分都应了一个设置函数

- 在对应的类中函数名叫什么?
 - 规律: set+属性名 == 函数名
- 某些属性没有对应的函数，只能在属性窗口中设置

4.3 状态栏

一般情况下，需要在状态栏中添加某些控件，显示某些属性，使用最多的就是添加标签 QLabel

```
1 // 类型: QStatusBar
2 void QStatusBar::addWidget(QWidget *widget, int stretch = 0);
3
4 [slot] void QStatusBar::clearMessage();
5 [slot] void QStatusBar::showMessage(const QString &message, int timeout =
0);
```

相关的操作代码

```
1 MainWindow::MainWindow(QWidget *parent)
2   : QMainWindow(parent)
3   , ui(new Ui::MainWindow)
```

```
4 {  
5     ui->setupUi(this);  
6  
7     // 状态栏添加子控件  
8     // 按钮  
9     QPushButton* button = new QPushButton("按钮");  
10    ui->statusBar->addWidget(button);  
11    // 标签  
12    QLabel* label = new QLabel("hello,world");  
13    ui->statusBar->addWidget(label);  
14 }
```

4.4 停靠窗口

停靠窗口可以通过鼠标拖动停靠到窗口的上、下、左、右，或者浮动在窗口上方。如果需要这种类型的窗口必须手动添加，如果在非QMainWindow类型的窗口中添加了停靠窗口，那么这个窗口是不能移动和浮动的。

浮动窗口在工具栏中，直接将其拖拽到 UI 界面上即可。



停靠窗口也有一个属性面板，我们可以在其对应属性面板中直接进行设置和修改相关属性。

属性	值
> QObject	
> QWidget	
✓ QDockWidget	
floating	<input type="checkbox"/> 窗口显示之后是否浮动在父窗口上方, 勾选==浮动显示
✓ features	DockWidgetClosable DockWidgetMovable DockWidgetFloatable A...
DockWidgetClosable	<input checked="" type="checkbox"/> 停靠窗口是否添加关闭按钮
DockWidgetMovable	<input checked="" type="checkbox"/> 停靠窗口是否可以移动
DockWidgetFloatable	<input checked="" type="checkbox"/> 停靠窗口是否有浮动属性
DockWidgetVerticalTitleBar	<input type="checkbox"/> 停靠窗口是否有垂直标题栏
DockWidgetFeatureMask	<input type="checkbox"/>
AllDockWidgetFeatures	<input checked="" type="checkbox"/>
NoDockWidgetFeatures	<input type="checkbox"/>
Reserved	<input type="checkbox"/>
✓ allowedAreas	LeftDockWidgetArea RightDockWidgetArea TopDockWidgetArea ...
LeftDockWidgetArea	<input checked="" type="checkbox"/> 可以停靠在窗口左侧
RightDockWidgetArea	<input checked="" type="checkbox"/> 可以停靠在窗口右侧
TopDockWidgetArea	<input checked="" type="checkbox"/> 可以停靠在窗口顶部
BottomDockWidgetArea	<input checked="" type="checkbox"/> 可以停靠在窗口底部
DockWidgetArea_Mask	<input checked="" type="checkbox"/>
AllDockWidgetAreas	<input checked="" type="checkbox"/>
NoDockWidgetArea	<input type="checkbox"/> 不能停靠在任何位置
> windowTitle	设置停靠窗口的标题
dockWidgetArea	LeftDockWidgetArea 设置窗口默认停靠的位置
docked	<input checked="" type="checkbox"/> 设置窗口是否可以停靠, 默认是可以的

5. 资源文件 .qrc

资源文件顾名思义就是一个存储资源的文件，在 Qt 中引入资源文件好处在于他能提高应用程序的部署效率并且减少一些错误的发生。

在程序编译过程中，添加到资源文件中的文件也会以二进制的形式被打包到可执行程序中，这样这些资源就永远和可执行程序捆绑到一起了，不会出现加载资源却找不到的问题。

虽然资源文件优势很明显，但是它也不是万能的，资源文件中一般添加的都是比较小的资源，比如：图片，配置文件，MP3 等，如果是类似视频这类比较大的文件就不适合放到资源文件中了。

比如我们需要给某个窗口设置图标，代码如下：

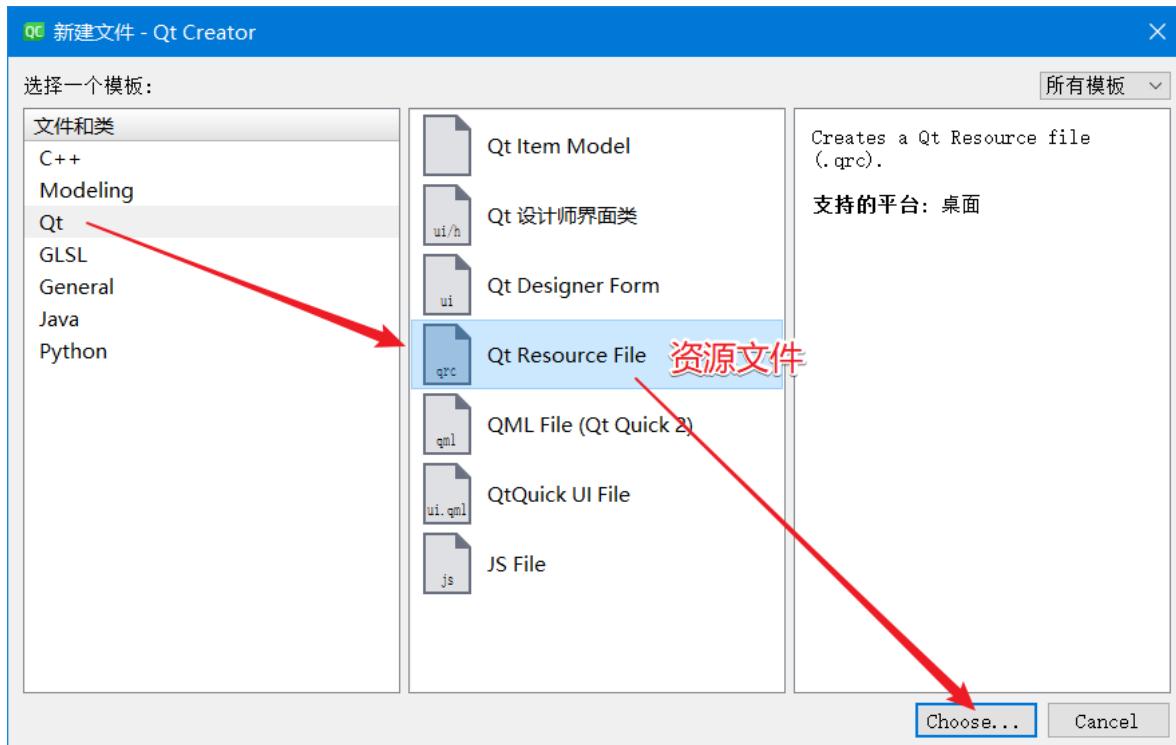
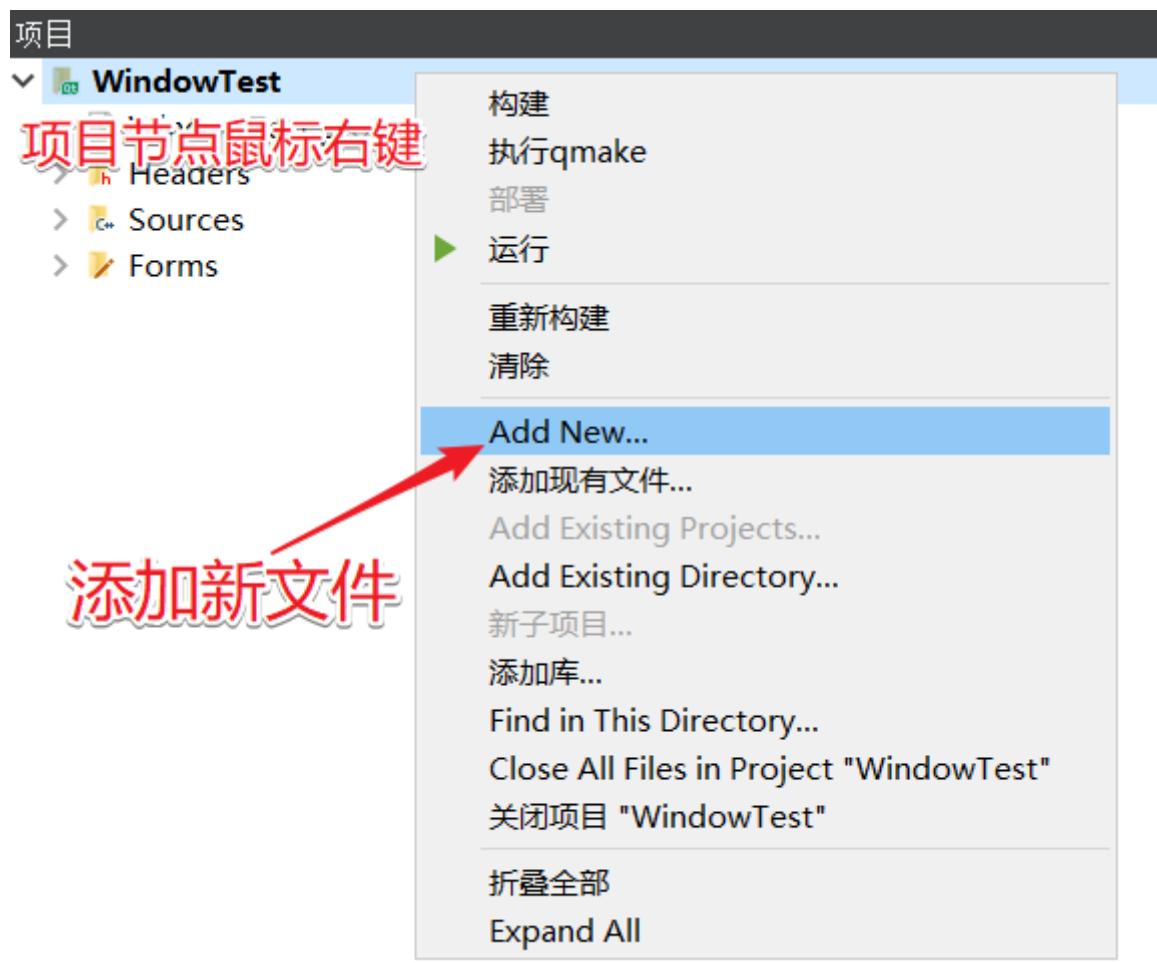
```

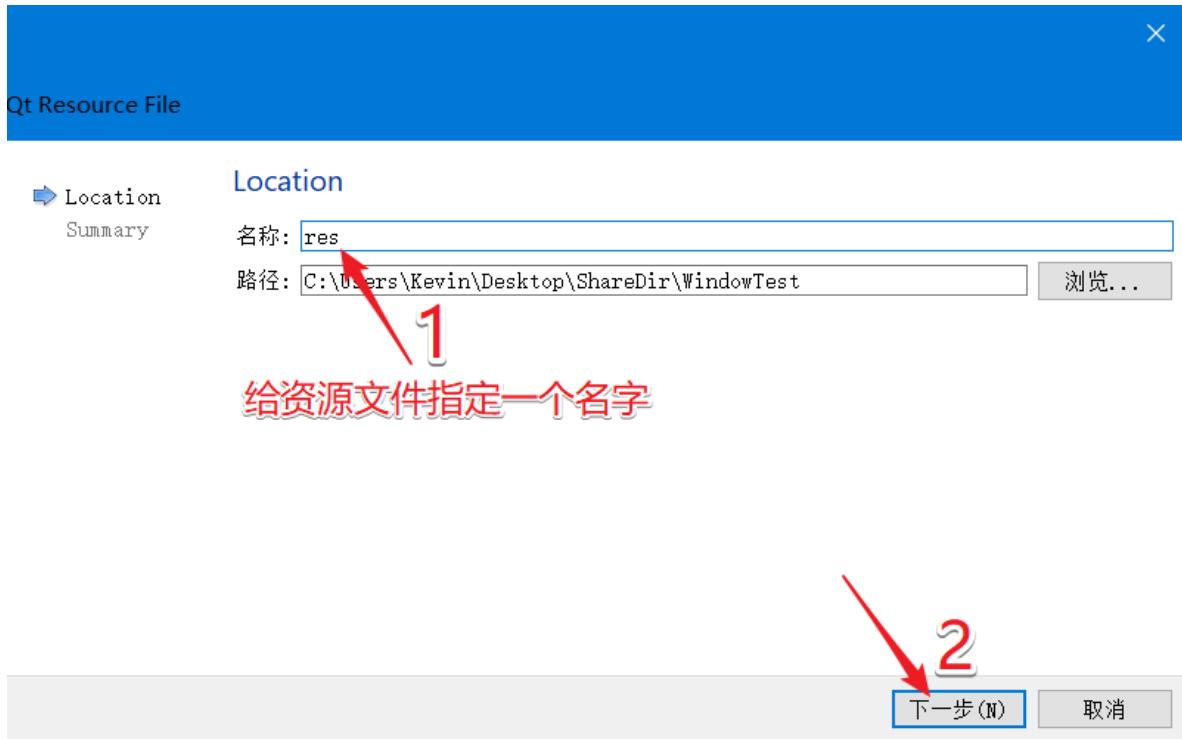
1 // 创建图标对象
2 QIcon::QIcon(const QString &fileName);
3 // QWidget类的 公共成员函数
4 void setwindowIcon(const QIcon &icon);
5
6 // 给窗口设置图标
7 // 弊端：发布的 exe 必须要加载 d:\\pic\\1.ico 如果当前主机对应的目录中没有图片，图标就无法被加载
8 // 发布 exe 需要额外发布图片，将其部署到某个目录中
9 setwindowIcon(QIcon("d:\\pic\\1.ico"));

```

我们可以使用资源文件解决上述的弊端，这样发布应用程序的时候直接发布 exe 就可以，不需要再额外提供图片了。

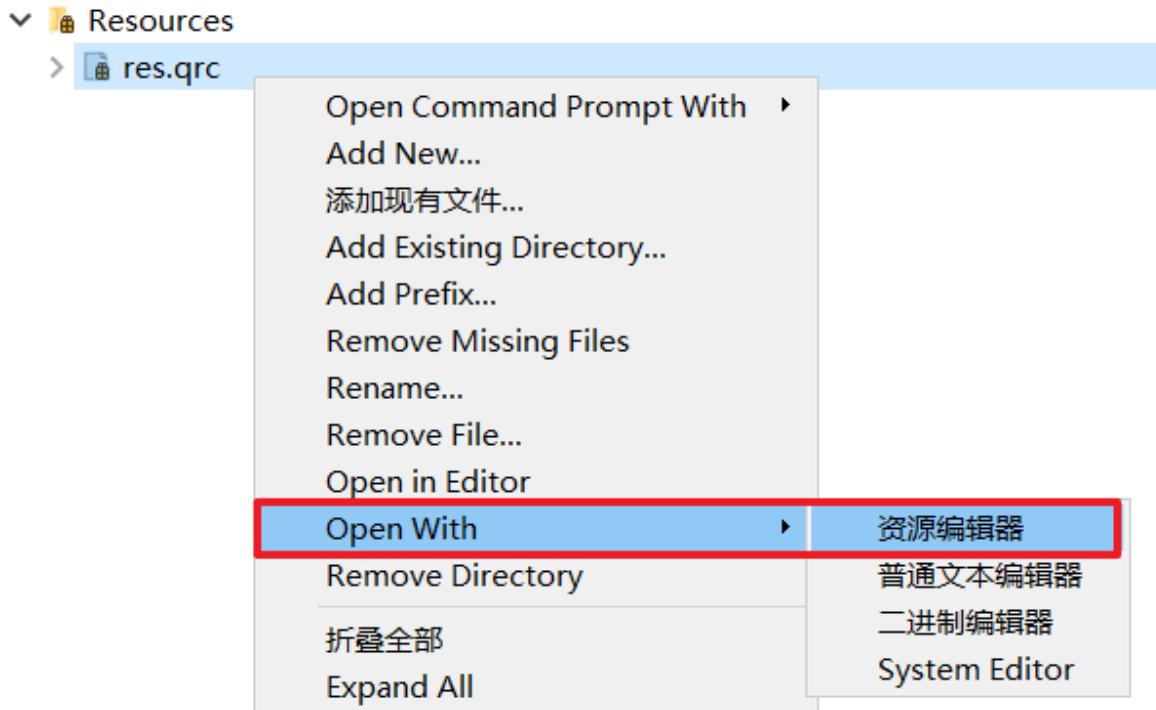
下面介绍一下关于资源文件的创建步骤：





资源文件添加完毕之后，继续给大家介绍资源文件的使用

1. 使用资源编辑器打开资源文件



2.给资源添加前缀

一个资源文件中可以添加多个前缀，前缀就是添加的资源在资源文件中的路径，前缀根据实际需求制定即可，路径以 / 开头

Add Prefix Add Files 删 除 Remove Missing Files

属性
别名：
前缀：
语言：

添加一个前缀

Add Prefix Add Files 删 除 Remove Missing Files

属性
别名：
前缀： /new/prefix1
语言：

根据实际需求, 修改前缀的名字

3.添加文件

前缀添加完毕，就可以在某个前缀下边添加相关的资源了。



- 弹出以文件选择对话框，选择资源文件
 - 资源文件放到什么地方?
 - 放到和项目文件.pro同一级目录或者更深的目录中
 - 错误的做法：将资源文件放到.pro文件的上级目录，这样资源文件无法被加载到
- 可以给添加的资源文件设置别名，设置别名之后原来的名字就不能使用了

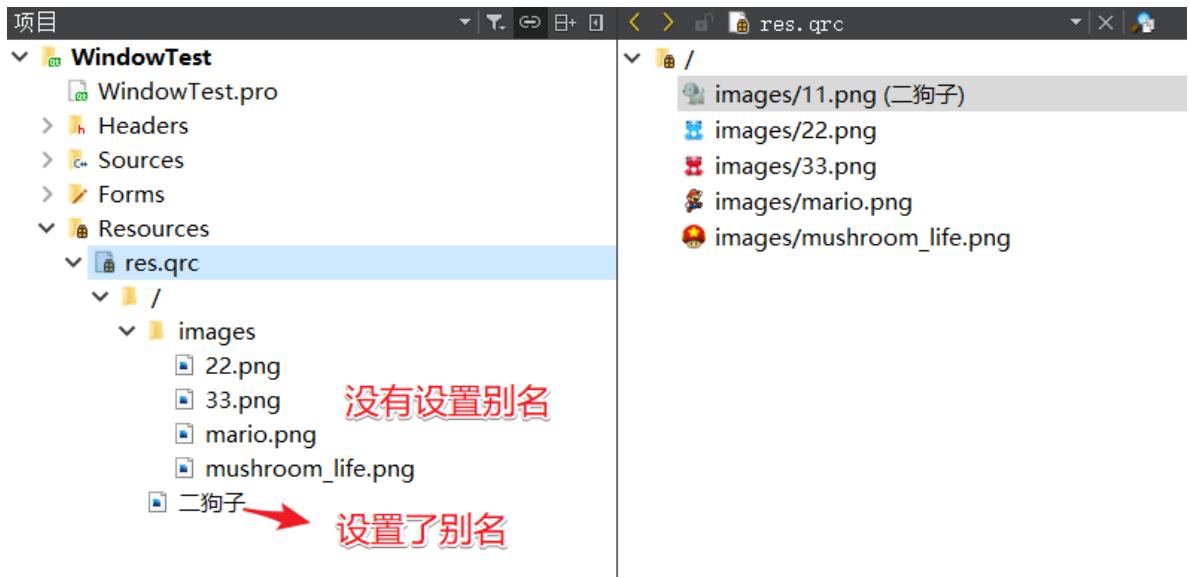
温馨提示：

1. 在高版本的QtCreator中，资源文件名字或者别名不支持中文
2. 如果设置了中文会出现编译会报错
3. 在此只是演示，使用过程中需要额外注意该问题

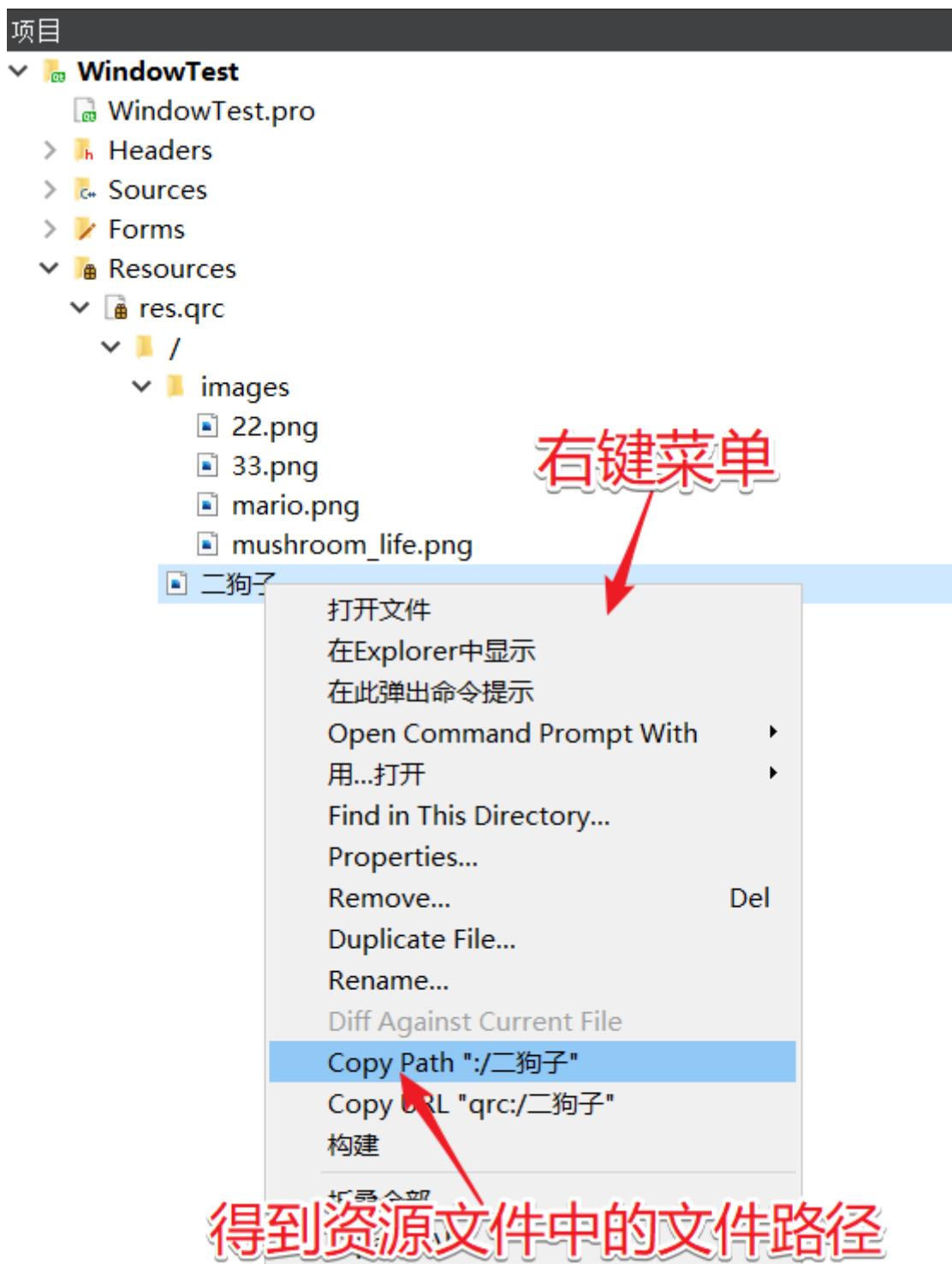


4. 如何在程序中使用资源文件中的图片

将项目树中的资源文件节点展开



找到需要使用的资源图片节点，鼠标右键，弹出的菜单中选择 Copy Path ...



六、Qt窗口布局

文章中主要介绍了 Qt 的窗口布局以及使用，主要内容包括：布局的样式，基于UI界面设置布局，基于API设置布局。

1. 布局的样式

Qt 窗口布局是指将多个子窗口按照某种排列方式将其全部展示到对应的父窗口中的一种处理方式。在 Qt 中常用的布局样式有三种，分别是：

布局样式	描述	行数	列数
水平布局	布局中的所有的控件水平排列	1 行	N 列 (N>=1)

垂直布局	描述中的所有的控件垂直排列	行数 (N≥1)	列数
网格 (栅格) 布局	布局中的所有的控件垂直 + 水平排列	N 行	N 列 (N≥1)

- 1 | 有问有答：
- 2 | 1. 控件的位置可以通过坐标指定，为什么还要使用布局？
- 3 | - 坐标指定的位置是固定的，当窗口大小发生改变，子窗口位置不会变化
- 4 | - 使用坐标指定子窗口位置，这个控件可能会被其他控件覆盖导致无法显示出来
- 5 | - 使用布局的方式可以完美解决以上的问题
- 6 | - 一般在制作窗口的过程中都是给子控件进行布局，而不是指定固定坐标位置
- 7 | 2. 布局有局限性吗，窗口结构复杂如何解决呢？
- 8 | - 没有局限性，并且布局的使用是非常灵活的
- 9 | - 各种布局是可以无限嵌套使用的，这样就可以制作成非常复杂的窗口了
- 10 | - 思路是这样的：给窗口设置布局，在布局中添加窗口，子窗口中再设置布局，
在子窗口布局中再次添加窗口，.....(无限循环)
- 11 |

2. 在 UI 窗口中设置布局

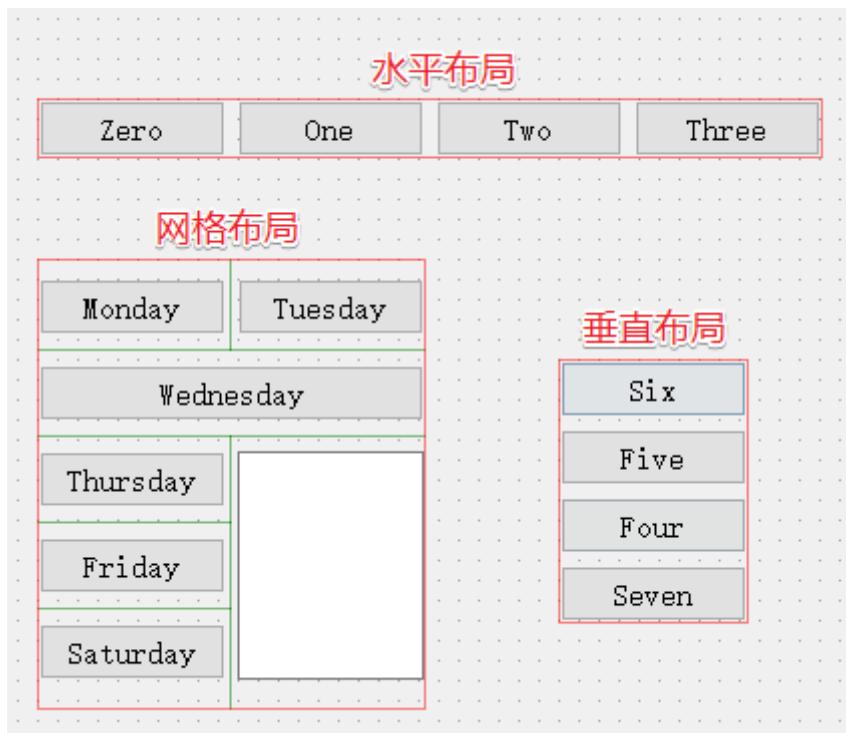
| 在 UI 窗口中进行布局的设置一共有两种处理方式，下面依次为大家进行讲解

2.1 方式 1

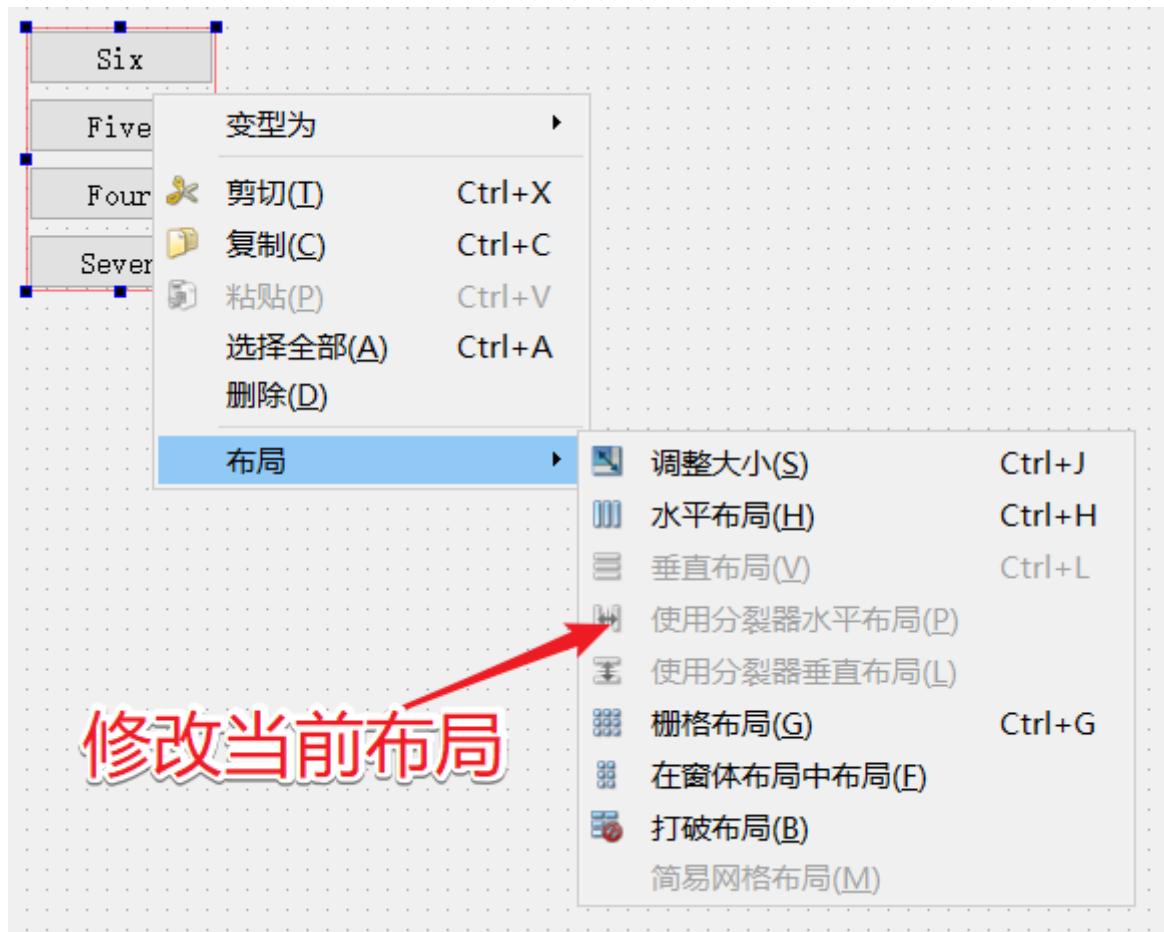
| 第一种方式是使用 Qt 提供的布局，从工具箱中找到相关的布局，然后将其拖拽到 UI 窗口中



| 将相应的控件放入到布局对应的红色框内部，这些控件就按照布局的样式自动排列到一起了，是不是很方便，赶紧自己操作起来感受一下吧。



除此之外，我们也可以修改当前布局，需要先选中当前布局，然后鼠标右键，在右键菜单中找布局在其子菜单项中选择其他布局即可

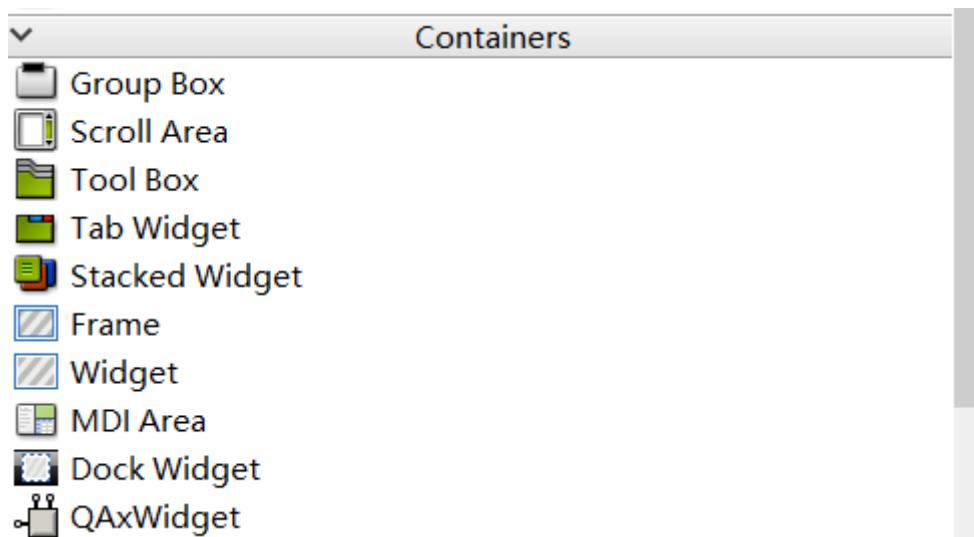


2.2 方式 2

第二种方式是直接在父窗口中对子部件进行布局，如果窗口结构很复杂需要嵌套，那么就需要先将这些子部件放到一个容器类型的窗口中，然后再对这个容器类型的窗口进行布局操作。步骤听起来布局复杂，下边依次为大家演示一下。

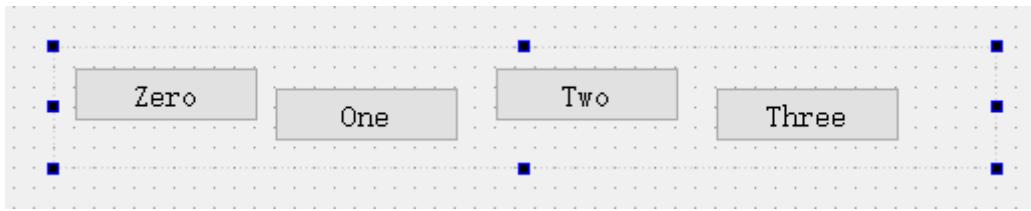
1.首先需要从工具栏中拖拽一个容器类型的窗口到 UI 界面上

一般首选 QWidget 原因是简单，并且窗口显示之后看不到任何痕迹



2.将要布局的子控件放到这个 QWidget 中

控件拖放过程中无需在意位置和是否对齐，没有布局之前显示杂乱无序是正常现象。



3. 对这个 QWidget 进行布局

首先选中这个存储子部件的父容器窗口，然后鼠标右键，在右键菜单中找布局，通过其子菜单就可以选择需要的布局方式了。布局之后所有的子部件就能够按照对应样式排列了(如果是网格布局，有时候需要使用鼠标调节一下)



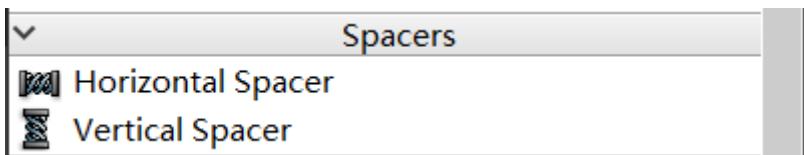
也可以通过窗口上方工具栏提供的布局按钮进行窗口布局

2.3 弹簧的使用

在进行窗口布局的时候为了让界面看起来更加美观，需要调整控件的位置，比如：靠左，靠右，居中，又或者我们需要调节两个控件之间的距离，以上这些需求使用弹簧都是可以实现的。

弹簧的样式有两种：

- 水平弹簧: 在水平方向起作用
- 垂直弹簧: 在垂直方向起作用



弹簧也有对应的属性可以设置，具体属性如下图所示：

属性	值
Spacer	
spacerName	horizontalSpacer_2 弹簧的名字
orientation	Horizontal 弹簧的方向: 水平 or 垂直
sizeType	Expanding 弹簧的类型
sizeHint	
宽度	122 弹簧的尺寸大小: 宽度 and 高度
高度	20

关于弹簧的 sizeType 属性，有很多选项，一般常用的只有两个：

- Fixed: 得到一个固定大小的弹簧
- Expanding: 得到一个可伸缩的弹簧，默认弹簧撑到最大

属性	值
Spacer	
spacerName	horizontalSpacer
orientation	Horizontal
sizeType	Expanding
sizeHint	
Fixed	弹簧大小固定, 没有弹性
Minimum	可以设置弹簧最小值, 有弹性
Maximum	可以设置弹簧最大值, 有弹性
Preferred	弹簧以合适大小显示, 有弹性
MinimumExpanding	可以设置最小值, 有弹性, 伸展到当前最大
Expanding	伸展到当前最大, 有弹性
Ignored	限制变少了, 有弹性

2.4 布局属性设置

当我们给窗口设置了布局之后，选中当前窗口，就可以看到在其对应的属性窗口中除了有窗口属性，还有一个布局属性，下面给大家介绍一下这些属性：

属性	值
> QObject	
> QWidget	
Layout	
layoutName	horizontalLayout 布局的名字
layoutLeftMargin	9 布局左侧预留的边距(留白), 单位像素
layoutTopMargin	9 布局上侧预留的边距(留白), 单位像素
layoutRightMargin	9 布局右侧预留的边距(留白), 单位像素
layoutBottomMargin	9 布局下侧预留的边距(留白), 单位像素
layoutSpacing	6 布局中控件之间的间距大小, 单位像素
layoutStretch	0,0,0,0,0
layoutSizeConstraint	SetDefaultConstraint

我们通过设置布局上下左右的边距, 或者是控件之间的距离也可以使界面看起来更加美观。

2.5 布局的注意事项

通过 UI 编辑窗口的树状列表我们可以对所有窗口的布局进行检查, 如果发现某个窗口没有布局, 一定要对其进行设置, 如果某个窗口没有进行布局, 那么当这个窗口显示出来之后里边的子部件就可能无法被显示出来, 尤其是初学者一定要注意这个问题。

对象	类	
MyDialog	QDialog	没有布局
acceptBtn	QPushButton	
donBtn	QPushButton	
gridLayout	QGridLayout	网格布局
horizontalWidget	QWidget	水平布局
rejectBtn	QPushButton	
verticalLayout_2	QVBoxLayout	垂直布局

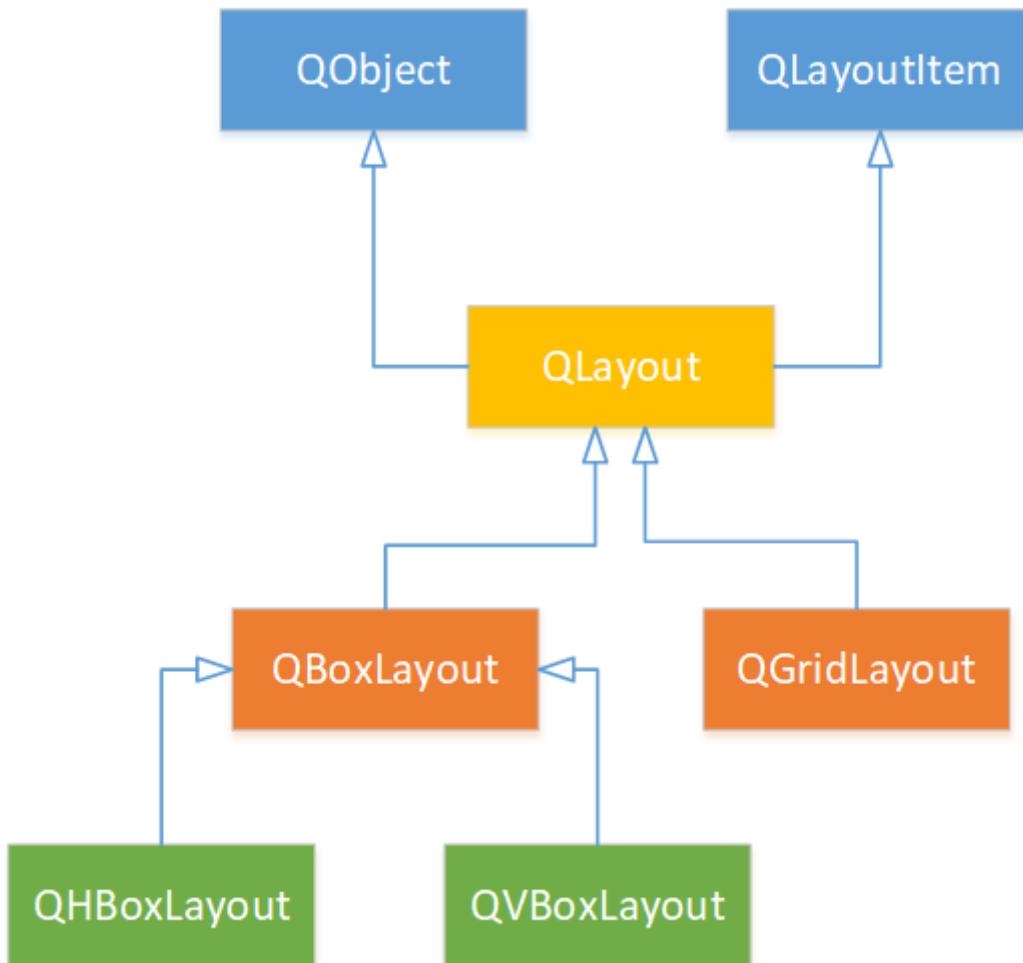
如果想要自己制作出一个比较美观的窗口需要反复对细节进行调节 (修改布局的属性) 并且要借助弹簧, 这样才能实现我们想要的效果。

下面是一个简单的登录窗口, 大家可以试着自己搞一搞



3. 通过 API 设置布局

在 QT 中，布局也有对应的类，布局类之间的关系如下：



在上图中的布局类虽然很多，常用的布局类有三个，就前边给大家介绍的三种布局，对应关系如下：

布局样式	类名
水平布局	QHBoxLayout
垂直布局	VBoxLayout
网格（栅格）布局	GridLayout

虽然一般我们不使用这些布局类对窗口进行布局，但是在这里还是给大家介绍一下这些类中常用的一些 API 函数

3.1 QLayout

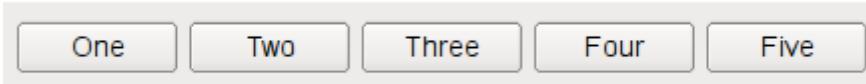
```
1 // 在布局最后面添加一个窗口
2 void QLayout::addWidget(QWidget *w);
3 // 将某个窗口对象从布局中移除，窗口对象如果不再使用需要自己析构
4 void QLayout::removeWidget(QWidget *widget);
5 // 设置布局的四个边界大小，即：左、上、右和下的边距。
6 void QLayout::setContentsMargins(int left, int top, int right, int bottom);
7 // 设置布局中各个窗口之间的间隙大小
8 void setSpacing(int);
```

3.2 QBoxLayout

这个类中的常用 API 都是从基类继承过来的，关于其使用，实例代码如下：

```
1 // 创建父窗口对象
2 QWidget *window = new QWidget;
3 // 创建若干个子窗口对象
4 QPushButton *button1 = new QPushButton("One");
5 QPushButton *button2 = new QPushButton("Two");
6 QPushButton *button3 = new QPushButton("Three");
7 QPushButton *button4 = new QPushButton("Four");
8 QPushButton *button5 = new QPushButton("Five");
9
10 // 创建水平布局对象
11 QHBoxLayout *layout = new QHBoxLayout;
12 // 将子窗口添加到布局中
13 layout->addWidget(button1);
14 layout->addWidget(button2);
15 layout->addWidget(button3);
16 layout->addWidget(button4);
17 layout->addWidget(button5);
18
19 // 将水平布局设置给父窗口对象
20 window->setLayout(layout);
21 // 显示父窗口
22 window->show();
```

代码效果展示：



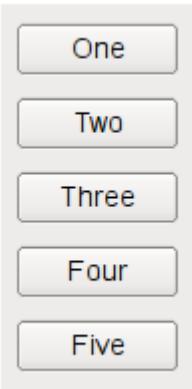
3.3 QVBoxLayout

这个类中的常用 API 都是从基类继承过来的，关于其使用，实例代码如下：

```
1 // 创建父窗口对象
2 QWidget *window = new QWidget;
3 // 创建若干个子窗口对象
4 QPushButton *button1 = new QPushButton("One");
5 QPushButton *button2 = new QPushButton("Two");
6 QPushButton *button3 = new QPushButton("Three");
7 QPushButton *button4 = new QPushButton("Four");
8 QPushButton *button5 = new QPushButton("Five");
9
10 // 创建垂直布局对象
11 QVBoxLayout *layout = new QVBoxLayout;
12 // 将子窗口添加到布局中
13 layout->addWidget(button1);
14 layout->addWidget(button2);
15 layout->addWidget(button3);
16 layout->addWidget(button4);
17 layout->addWidget(button5);
18
19 // 将水平布局设置给父窗口对象
```

```
20 window->setLayout(layout);  
21 // 显示父窗口  
22 window->show();
```

代码效果展示:



3.4 QGridLayout

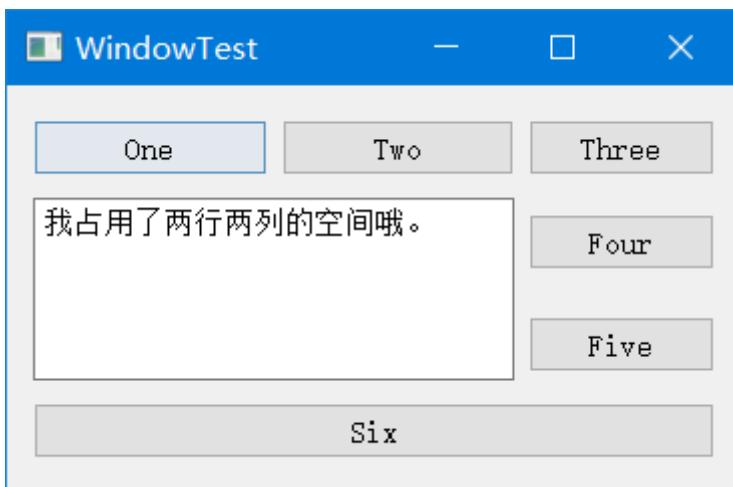
```
1 // 构造函数  
2 QGridLayout::QGridLayout();  
3 QGridLayout::QGridLayout(QWidget *parent);  
4  
5 // 添加窗口对象到网格布局中  
6 /*  
7 参数:  
8 - widget: 添加到布局中的窗口对象  
9 - row: 添加到布局中的窗口对象位于第几行 (从0开始)  
10 - column: 添加到布局中的窗口对象位于第几列 (从0开始)  
11 - alignment: 窗口在布局中的对齐方式, 没有特殊需求使用默认值即可  
12 */  
13 void QGridLayout::addWidget(  
14     QWidget *widget, int row, int column,  
15     Qt::Alignment alignment = Qt::Alignment());  
16  
17 /*  
18 参数:  
19 - widget: 添加到布局中的窗口对象  
20 - fromRow: 添加到布局中的窗口对象位于第几行 (从0开始)  
21 - fromColumn: 添加到布局中的窗口对象位于第几列 (从0开始)  
22 - rowSpan: 添加的窗口从 fromRow 行开始跨越的行数  
23 - columnSpan: 添加的窗口从 fromColumn 列开始跨越的列数  
24 - alignment: 窗口在布局中的对齐方式, 没有特殊需求使用默认值即可  
25 */  
26 void QGridLayout::addWidget(  
27     QWidget *widget, int fromRow, int fromColumn,  
28     int rowSpan, int columnSpan,  
29     Qt::Alignment alignment = Qt::Alignment());  
30  
31 // 设置 column 对应的列的最新宽度, 单位: 像素  
32 void QGridLayout::setColumnMinimumWidth(int column, int minsize);  
33  
34 // 设置布局中水平方向窗口之间间隔的宽度  
35 void QGridLayout::setHorizontalSpacing(int spacing);  
36
```

```
37 // 设置布局中垂直方向窗口之间间隔的宽度  
38 void QGridLayout::setVerticalSpacing(int spacing);
```

测试代码如下：

```
1 // 创建父窗口对象  
2 QWidget* window = new QWidget;  
3 // 创建子窗口对象  
4 QPushButton *button1 = new QPushButton("One");  
5 QPushButton *button2 = new QPushButton("Two");  
6 QPushButton *button3 = new QPushButton("Three");  
7 QPushButton *button4 = new QPushButton("Four");  
8 QPushButton *button5 = new QPushButton("Five");  
9 QPushButton *button6 = new QPushButton("Six");  
10 // 多行文本编辑框  
11 QTextEdit* txedit = new QTextEdit;  
12 txedit->setText("我占用了两行两列的空间哦。");  
13  
14 QGridLayout* layout = new QGridLayout;  
15 // 按钮起始位置：第1行，第1列，该按钮占用空间情况为1行1列  
16 layout->addWidget(button1, 0, 0);  
17 // 按钮起始位置：第1行，第2列，该按钮占用空间情况为1行1列  
18 layout->addWidget(button2, 0, 1);  
19 // 按钮起始位置：第1行，第3列，该按钮占用空间情况为1行1列  
20 layout->addWidget(button3, 0, 2);  
21 // 编辑框起始位置：第2行，第1列，该按钮占用空间情况为2行2列  
22 layout->addWidget(txedit, 1, 0, 2, 2);  
23 // 按钮起始位置：第2行，第3列，该按钮占用空间情况为1行1列  
24 layout->addWidget(button4, 1, 2);  
25 // 按钮起始位置：第3行，第3列，该按钮占用空间情况为1行1列  
26 layout->addWidget(button5, 2, 2);  
27 // 按钮起始位置：第4行，第1列，该按钮占用空间情况为1行3列  
28 layout->addWidget(button6, 3, 0, 1, 3);  
29  
30 // 网格布局设置给父窗口对象  
31 window->setLayout(layout);  
32 // 显示父窗口  
33 window->show();
```

测试代码效果展示：



七、在Qt窗口中添加右键菜单

如果想要在某一窗口中显示右键菜单，其处理方式大体上有两种，这两种方式分别为基于鼠标事件实现和基于窗口的菜单策略实现。其中第二种方式中又有三种不同的实现方式，因此如果想要在窗口中显示一个右键菜单一共四种实现方式。

1. 基于鼠标事件实现

1.1 实现思路

使用这种方式实现右键菜单的显示需要使用事件处理器函数，在 Qt 中这类函数都是回调函数，并且在自定义窗口类中我们还可以自定义事件处理器函数的行为（因为子类继承了父类的这个方法并且这类函数是虚函数）。

实现步骤如下：

1. 在当前窗口类中重写鼠标操作相关的事件处理器函数，有两个可以选择

```
1 // 以下两个事件二选一即可，只是事件函数被调用的时机不同罢了
2 // 这个时机对右键菜单的显示没有任何影响
3 [virtual protected] void QWidget::mousePressEvent(QMouseEvent *event);
4 [virtual protected] void QWidget::mouseReleaseEvent(QMouseEvent *event);
```

2. 在数据表事件处理器函数内部判断是否按下了鼠标右键

3. 如果按下了鼠标右键创建菜单对象（也可以提前先创建处理），并将其显示出来

```
1 // 关于QMenu类型的菜单显示需要调用的 API
2 // 参数 p 就是右键菜单需要显示的位置，这个坐标需要使用屏幕坐标
3 // 该位置坐标一般通过调用 QCursors::pos() 直接就可以得到了
4 QAction *QMenu::exec(const QPoint &p, QAction *action = nullptr);
```

1.2 代码实现

在头文件中添加要重写的鼠标事件处理器函数声明，这里使用的是 mousePressEvent()

```
1 // mainwindow.h
2 #include < QMainWindow>
3
4 namespace Ui {
5 class Mainwindow;
6 }
7
8 class Mainwindow : public QMainWindow
9 {
10     Q_OBJECT
11
12 public:
13     explicit Mainwindow(QWidget *parent = 0);
14     ~Mainwindow();
15
16 protected:
17     // 鼠标按下，该函数被Qt框架调用，需要重写该函数
18     void mousePressEvent(QMouseEvent *event);
```

```
20 | private:  
21 |     Ui::MainWindow *ui;  
22 | };
```

在源文件中重写从父类继承的虚函数 mousePressEvent()

```
1 // mainwindow.cpp  
2 void MainWindow::mousePressEvent(QMouseEvent *event)  
3 {  
4     // 判断用户按下的鼠标键  
5     if(event->button() == Qt::RightButton)  
6     {  
7         // 弹出一个菜单，菜单项是 QAction 类型  
8         QMenu menu;  
9         QAction* act = menu.addAction("C++");  
10        connect(act, &QAction::triggered, this, [=]()  
11        {  
12            QMessageBox::information(this, "title", "您选择的是C++...");  
13        });  
14        menu.addAction("Java");  
15        menu.addAction("Python");  
16        menu.exec(QCursor::pos()); // 右键菜单被模态显示出来了  
17    }  
18 }
```

2. 基于窗口的菜单策略实现

这种方式是使用 Qt 中 QWidget 类中的右键菜单函数

QWidget::setContextMenuPolicy(Qt::ContextMenuPolicy policy) 来实现，因为这个函数的参数可以指定不同的值，因此不同参数对应的具体的实现方式也不同。

这个函数的函数原型如下：

```
1 // 函数原型：  
2 void QWidget::setContextMenuPolicy(Qt::ContextMenuPolicy policy);  
3 参数：  
4 - Qt::NoContextMenu      --> 不能实现右键菜单  
5 - Qt::PreventContextMenu --> 不能实现右键菜单  
6 - Qt::DefaultContextMenu --> 基于事件处理器函数 QWidget::contextMenuEvent()  
实现  
7 - Qt::ActionsContextMenu --> 添加到当前窗口中所有 QAction 都会作为右键菜单项显示  
出来  
8 - Qt::CustomContextMenu   --> 基于 QWidget::customContextMenuRequested() 信号实现
```

2.1 Qt::DefaultContextMenu

使用这个策略实现右键菜单，需要借助窗口类从父类继承的虚函数 QWidget::contextMenuEvent() 并重写它来实现。

第一步是在窗口类的头文件中添加这个函数的声明

```
1 // mainwindow.h  
2 #include <QMainWindow>
```

```

3
4 namespace ui {
5 class Mainwindow;
6 }
7
8 class Mainwindow : public QMainWindow
9 {
10     Q_OBJECT
11
12 public:
13     explicit Mainwindow(QWidget *parent = 0);
14     ~Mainwindow();
15
16 protected:
17     // 如果窗口设置了 Qt::DefaultContextMenu 策略,
18     // 点击鼠标右键该函数被Qt框架调用
19     void contextMenuEvent(QContextMenuEvent *event);
20
21 private:
22     Ui::MainWindow *ui;
23 };

```

第二步在这个窗口类的构造函数设置右键菜单策略

```

1 // mainwindow.cpp
2 Mainwindow::Mainwindow(QWidget *parent) :
3     QMainWindow(parent),
4     ui(new Ui::MainWindow)
5 {
6     ui->setupUi(this);
7
8     // 给窗口设置策略: Qt::DefaultContextMenu
9     // 在窗口中按下鼠标右键, 这个事件处理器函数被Qt框架调用
10    QWidget::contextMenuEvent()
11    setContextMenuPolicy(Qt::DefaultContextMenu);
12 }

```

第三步在这个窗口类的源文件中重写事件处理器函数 contextMenuEvent()

```

1 // mainwindow.cpp
2 void Mainwindow::contextMenuEvent(QContextMenuEvent *event)
3 {
4     // 弹出一个菜单, 菜单项是 QAction 类型
5     QMenu menu;
6     QAction* act = menu.addAction("C++");
7     connect(act, &QAction::triggered, this, [=]()
8     {
9         QMessageBox::information(this, "title", "您选择的是C++..."); });
10    menu.addAction("Java");
11    menu.addAction("Python");
12    menu.exec(cursor->pos()); // 右键菜单被模态显示出来了
13 }
14 }

```

2.2 Qt::ActionsContextMenu

使用这个策略实现右键菜单，是最简单的一种，我们只需要创建一些 QAction 类型的对象并且将他们添加到当前的窗口中，当我们在窗口中点击鼠标右键这些 QAction 类型的菜单项就可以显示出来了。

虽然这种方法比较简单，但是它有一定的局限性，就在一个窗口中不能根据不同的需求制作不同的右键菜单，这种方式只能得到一个唯一的右键菜单。

相关的处理代码如下：

```
1 // mainwindow.cpp
2 MainWindow::MainWindow(QWidget *parent) :
3     QMainWindow(parent),
4     ui(new Ui::MainWindow)
5 {
6     ui->setupUi(this);
7
8     // 只要将某个QAction添加给对应的窗口，这个action就是这个窗口右键菜单中的一个菜单项
9     // 在窗口中点击鼠标右键，就可以显示这个菜单
10    setContextMenuPolicy(Qt::ActionsContextMenu);
11    // 给当前窗口添加QAction对象
12    QAction* act1 = new QAction("C++");
13    QAction* act2 = new QAction("Java");
14    QAction* act3 = new QAction("Python");
15    this->addAction(act1);
16    this->addAction(act2);
17    this->addAction(act3);
18    connect(act1, &QAction::triggered, this, [=]()
19    {
20        QMessageBox::information(this, "title", "您选择的是C++...");
```

2.3 Qt::CustomContextMenu

使用这个策略实现右键菜单，当点击鼠标右键，窗口会产生一个 QWidget::customContextMenuRequested() 信号，注意仅仅只是发射信号，意味着要自己写显示右键菜单的槽函数（slot），这个信号是 QWidget 唯一与右键菜单有关的信号。我们先来看一下这个信号的函数原型：

```
1 // 注意：信号中的参数pos为当前窗口的坐标，并非屏幕坐标，右键菜单显示需要使用屏幕坐标
2 [signal] void QWidget::customContextMenuRequested(const QPoint &pos)
```

代码实现也比较简单，如下所示：

```
1 // mainwindow.cpp
2 MainWindow::MainWindow(QWidget *parent) :
3     QMainWindow(parent),
4     ui(new Ui::MainWindow)
5 {
6     ui->setupUi(this);
7
8     // 策略 Qt::CustomContextMenu
9     // 当在窗口中点击鼠标右键，窗口会发出一个信号：
10    QWidget::customContextMenuRequested()
11    // 对应发射出的这个信号，需要添加一个槽函数，用来显示右键菜单
12    this->setContextMenuPolicy(Qt::CustomContextMenu);
```

```

12     connect(this, &MainWindow::customContextMenuRequested, this, [=](const
13         QPoint &pos)
14     {
15         // 参数 pos 是鼠标按下的位置，但是不能直接使用，这个坐标不是屏幕坐标，是当前窗
16         // 口的坐标
17         // 如果要使用这个坐标需要将其转换为屏幕坐标
18         QMenu menu;
19         QAction* act = menu.addAction("C++");
20         connect(act, &QAction::triggered, this, [=]()
21         {
22             QMessageBox::information(this, "title", "您选择的是C++...");  

23         });
24         menu.addAction("Java");
25         menu.addAction("Python");
26         // menu.exec(QCursor::pos());
27         // 将窗口坐标转换为屏幕坐标
28         QPoint newpt = this->mapToGlobal(pos);
29         menu.exec(newpt);
30     });
31 }

```

在上边的程序中，我们通过窗口发射的信号得到了一个坐标类型的参数，大家一定要注意这个坐标是当前窗口的窗口坐标，不是屏幕坐标，显示右键菜单需要使用屏幕坐标。
对应这个坐标的处理可以有两种方式：

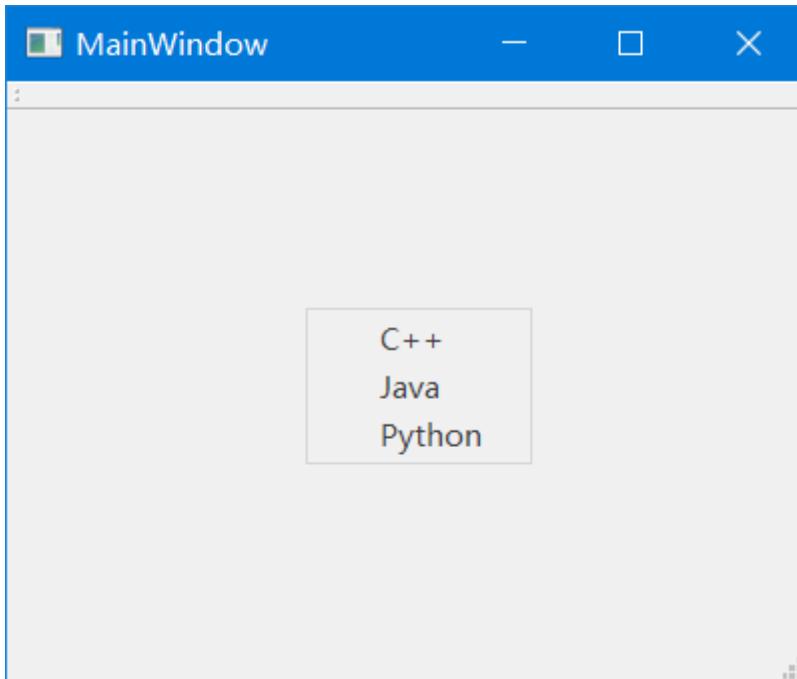
- 弃用，选择使用 `QCursor::pos()` 得到光标在屏幕的坐标位置
- 坐标转换，将窗口坐标转换为屏幕坐标，这里用到了一个函数 `mapToGlobal`

```

1 // 参数是当前窗口坐标，返回值为屏幕坐标
2 QPoint QWidget::mapToGlobal(const QPoint &pos) const;

```

不管使用以上哪种方式显示右键菜单，显示出来之后的效果是一样的



最后如果想要让自己的右键菜单项显示图标，可以调用这个函数

```
1 // 只显示文本字符串
2 QAction *QMenu::addAction(const QString &text);
3 // 可以显示图标 + 文本字符串
4 QAction *QMenu::addAction(const QIcon &icon, const QString &text);
```

八、Qt中按钮类型的控件

文章中主要介绍了 Qt 中常用的按钮控件，包括: QAbstractButton, QPushButton, QToolButton, QRadioButton, QCheckBox。

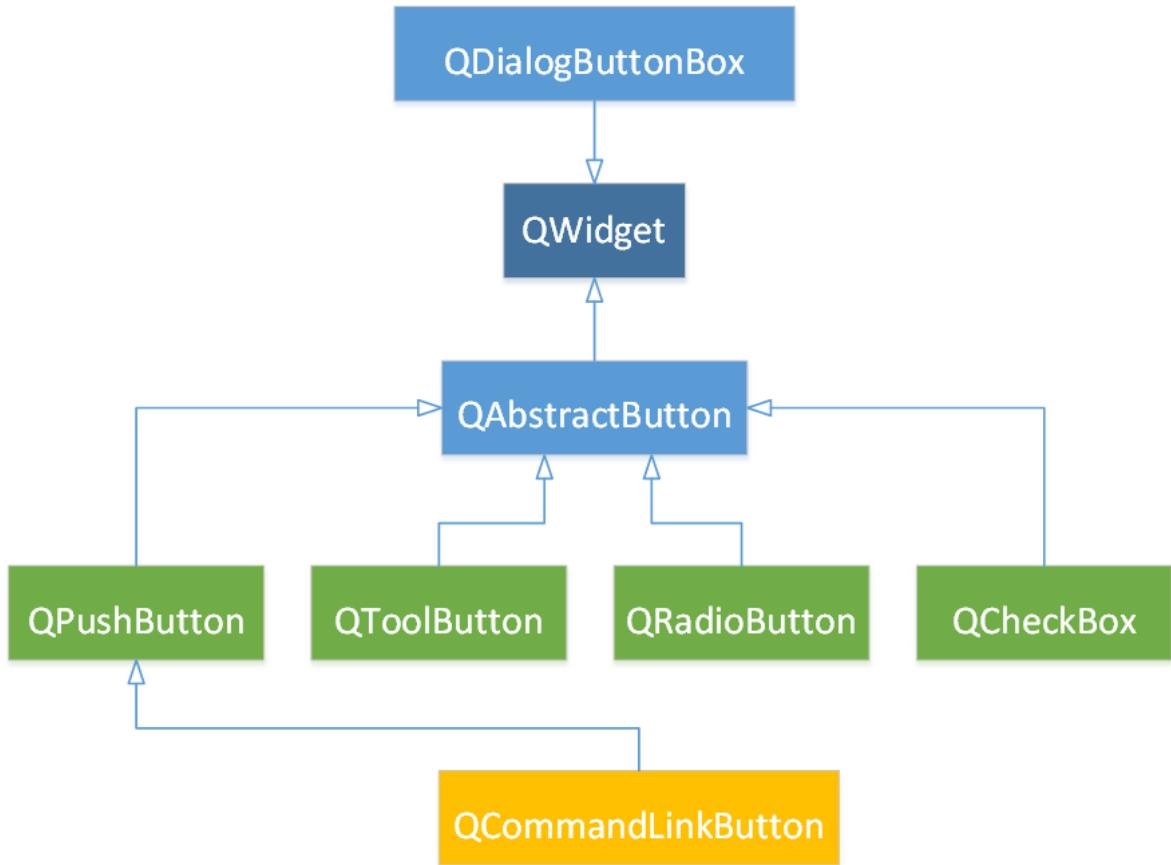
1. 按钮基类 QAbstractButton

在 QT 中为我们提供了可以直接使用的按钮控件，如下图。这些按钮种类虽然繁多，但是它们都拥有相同的父类 QAbstractButton。这些子类按钮的大部分属性都是从这个基类继承的，因此搞明白这个类为我们提供的相关功能还是非常重要的。

其中 Dialog Button Box 比较特殊不是一个单一控件，它是两个 QPushButton 的组合并且水平排列，这个不能作为一个新控件来研究。



这些按钮控件之间的继承关系如下图：



下边从功能的视角，给大家介绍一下 `QAbstractButton` 中的一些常用 API

1.1 标题和图标

```

1 // 参数text的内容显示到按钮上
2 void QAbstractButton::setText(const QString &text);
3 // 得到按钮上显示的文本内容，函数的返回就是
4 QString QAbstractButton::text() const;
5
6 // 得到按钮设置的图标
7 QIcon icon() const;
8 // 给按钮设置图标
9 void setIcon(const QIcon &icon);
10
11 // 得到按钮图标大小
12 QSize iconSize() const
13 // 设置按钮图标的大小
14 [slot]void setIconSize(const QSize &size);
  
```

1.2 按钮的 Check 属性

对应按钮来说，一般有三种常见状态，分别为: Normal, Hover, Pressed。

- Normal: 普通状态，没有和鼠标做任何接触
- Hover: 悬停状态，鼠标位于按钮之上，但是并未按下
- Pressed: 按压状态，鼠标键在按钮上处于按下状态

默认情况下，鼠标在按钮上按下，按钮从 Normal 切换到 Pressed 状态，鼠标释放，按钮从 Pressed 恢复到 Normal 状态。

当我们给按钮设置了 check 属性之后，情况就有所不同了，在按钮上释放鼠标键，按钮依然会处在 Pressed 状态，再次点击按钮，按钮才能恢复到 Normal 状态。具有 check 属性的按钮就相当

于一个开关，每点击一次才能实现一次状态的切换。

```
1 // 判断按钮是否设置了checkable属性，如果设置了点击按钮，按钮一直处于选中状态
2 // 默认这个属性是关闭的，not checkable
3 bool QAbstractButton::isCheckable() const;
4 // 设置按钮的checkable属性
5 // 参数为true：点击按钮，按钮被选中，松开鼠标，按钮不弹起
6 // 参数为false：点击按钮，按钮被选中，松开鼠标，按钮弹起
7 void QAbstractButton::setCheckable(bool);
8
9
10 // 判断按钮是不是被按下的选中状态
11 bool QAbstractButton::isChecked() const;
12 // 设置按钮的选中状态：true-选中，false-没选中
13 // 设置该属性前，必须先进行 checkable属性的设置
14 void QAbstractButton::setChecked(bool);
```

1.3 信号

这些信号都按钮被点击之后发射出来的，只是在细节上有细微的区别，其中最常用的是 clicked()，通过鼠标的不同瞬间状态可以发射出 pressed() 和 released() 信号，如果鼠标设置了 check 属性，一般通过 toggled() 信号判断当前按钮是选中状态还是非选中状态。

```
1 /*
2 当按钮被激活时(即，当鼠标光标在按钮内时按下然后释放)，当键入快捷键时，或者当click()或
3 animateClick()被调用时，这个信号被发出。值得注意的是，如果调用setDown()、
4 setChecked()或toggle()，则不会触发此信号。
5 */
6 [signal] void QAbstractButton::clicked(bool checked = false);
7 // 在按下按钮的时候发射这个信号
8 [signal] void QAbstractButton::pressed();
9 // 在释放这个按钮的时候发射直观信号
10 [signal] void QAbstractButton::released();
11 // 每当可检查按钮改变其状态时，就会发出此信号。checked在选中按钮时为true，在未选中按钮时
12 // 为false。
13 [signal] void QAbstractButton::toggled(bool checked);
```

1.4 槽函数

```
1 // 执行一个动画点击：按钮被立即按下，并在毫秒后释放(默认是100毫秒)。
2 [slot] void QAbstractButton::animateClick(int msec = 100);
3 // 执行一次按钮点击，相当于使用鼠标点击了按钮
4 [slot] void QAbstractButton::click();
5
6 // 参考 1.2 中的函数介绍
7 [slot] void QAbstractButton::setChecked(bool);
8 // 设置按钮上图标大小
9 [slot] void setIconSize(const QSize &size);
10 // 切换可检查按钮的状态。checked <==> unchecked
11 [slot] void QAbstractButton::toggle();
```

2. QPushButton

2.1 常用 API

这种类型的按钮是 Qt 按钮中使用频率最高的一个，对这个类进行操作，大部分时候都需要使用它从父类继承过来的那些 API。

在 QPushButton 类中，比较常用的一些 API 函数如下：

作者: 苏丙楓

链接: <https://subingwen.cn/qt/qt-buttons/#1-4-%E6%A7%BD%E5%87%BD%E6%95%B0>

来源: 爱编程的大丙

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

```
1 // 构造函数
2 /*
3 参数：
4   - icon: 按钮上显示的图标
5   - text: 按钮上显示的标题
6   - parent: 按钮的父对象，可以不指定
7 */
8 QPushButton::QPushButton(const QIcon &icon, const QString &text, QWidget
9 *parent = nullptr);
10 QPushButton::QPushButton(const QString &text, QWidget *parent = nullptr);
11 QPushButton::QPushButton(QWidget *parent = nullptr);
12
13 // 判断按钮是不是默认按钮
14 bool isDefault() const;
15 // 一般在对话框窗口中使用，将按钮设置为默认按钮，自动关联 Enter 键
16 void setDefault(bool);
17
18 /*
19 将弹出菜单与此按钮关联起来。这将把按钮变成一个菜单按钮，
20 在某些样式中会在按钮文本的右边产生一个小三角形。
21 */
22 void QPushButton::setMenu(QMenu *menu);
23
24 /*
25 显示(弹出)相关的弹出菜单。如果没有这样的菜单，这个函数什么也不做。
26 这个函数直到弹出菜单被用户关闭后才返回。
27 */
28 [slot] void QPushButton::showMenu();
```

2.2 按钮的使用

通过 API 的介绍，我们可以知道，使用 QPushButton 这种类型的按钮，有三种使用方式：

1. 作为普通按钮，可以显示文本信息和图标
2. 设置check属性，使其可以处于持续的被选中状态
3. 关联一个菜单，点击按钮菜单弹出

具体操作可以参考如下代码：

```
1 MainWindow::MainWindow(QWidget *parent)
2   : QMainWindow(parent)
3   , ui(new Ui::MainWindow)
4 {
5   ui->setupUi(this);
6
7   // 普通按钮，没有checked属性
8   ui->normalBtn->setText("我是小猪佩奇");
```

```

9  ui->normalBtn->setIcon(QIcon(":/Peppa-Pig.png"));
10 ui->normalBtn->setIconSize(QSize(30, 30));
11 connect(ui->normalBtn, &QPushButton::clicked, this, [=]()
12 {
13     qDebug() << "我是一个普通按钮， 图标是小猪佩奇..." ;
14 });
15
16 // 有checked属性的按钮
17 ui->checkedBtn->setCheckable(true);
18 connect(ui->checkedBtn, &QPushButton::toggled, this, [=](bool b1)
19 {
20     qDebug() << "我是一个checked按钮， 当前状态为:" << b1;
21 });
22
23 // 关联菜单
24 ui->menuBtn->setText("你喜欢哪种美女?");
25 QMenu* menu = new QMenu;
26 QAction* act = menu->addAction("可爱的");
27 menu->addAction("粘人的");
28 menu->addAction("胸大的");
29 menu->addAction("屁股翘的");
30 ui->menuBtn->setMenu(menu);
31 connect(act, &QAction::triggered, this, [=]{
32     qDebug() << "我是一个可爱的女人， 今晚约吗?";
33 });
34 }

```

3. QToolButton

3.1 常用 API

这个类也是一个常用按钮类，使用方法和功能跟 QPushButton 基本一致，只不过在对于关联菜单这个功能点上，QToolButton 类可以设置弹出的菜单的属性，以及在显示图标的时候可以设置更多的样式，可以理解为是一个增强版的 QPushButton。
和 QPushButton 类相同的是，操作这个按钮使用的大部分函数都是从父类 QAbstractButton 继承过来的。

```

1 ////////////////////////////////////////////////////////////////// 构造函数 //////////////////////////////////////////////////////////////////
2 QToolButton::QToolButton(QWidget *parent = nullptr);
3
4 ////////////////////////////////////////////////////////////////// 公共成员函数 //////////////////////////////////////////////////////////////////
5 /*
6 1. 将给定的菜单与此工具按钮相关联。
7 2. 菜单将根据按钮的弹出模式显示。
8 3. 菜单的所有权没有转移到“工具”按钮(不能建立父子关系)
9 */
10 void QToolButton::setMenu(QMenu *menu);
11 // 返回关联的菜单，如果没有定义菜单，则返回nullptr。
12 QMenu *QToolButton::menu() const;
13
14 /*
15 弹出菜单的弹出模式是一个枚举类型：QToolButton::ToolButtonPopupMode，取值如下：
16 - QToolButton::DelayedPopup:
17     - 延时弹出，按压工具按钮一段时间后才能弹出，比如：浏览器的返回按钮
18     - 长按按钮菜单弹出，但是按钮的 clicked 信号不会被发射
19 - QToolButton::MenuButtonPopup:

```

```

20      - 在这种模式下，工具按钮会显示一个特殊的箭头，表示有菜单。
21  - 当按下按钮的箭头部分时，将显示菜单。按下按钮部分发射 clicked 信号
22  - QToolButton::InstantPopup:
23      - 当按下工具按钮时，菜单立即显示出来。
24      - 在这种模式下，按钮本身的动作不会被触发(不会发射clicked信号
25 */
26 // 设置弹出菜单的弹出方式
27 void setPopupMode(QToolButton::ToolBarPopupMode mode);
28 // 获取弹出菜单的弹出方式
29 QToolButton::ToolBarPopupMode popupMode() const;
30
31 /*
32 QToolButton可以帮助我们在按钮上绘制箭头图标，是一个枚举类型，取值如下：
33     - Qt::NoArrow: 没有箭头
34     - Qt::UpArrow: 箭头向上
35     - Qt::DownArrow: 箭头向下
36     - Qt::LeftArrow: 箭头向左
37     - Qt::RightArrow: 箭头向右
38 */
39 // 显示一个箭头作为QToolButton的图标。默认情况下，这个属性被设置为Qt::NoArrow。
40 void setArrowType(Qt::ArrowType type);
41 // 获取工具按钮上显示的箭头图标样式
42 Qt::ArrowType arrowType() const;
43
44 //////////////////////////////////////////////////////////////////// 槽函数 ///////////////////////////////////////////////////////////////////
45 // 给按钮关联一个QAction对象，主要目的是美化按钮
46 [slot] void QToolButton::setDefaultAction(QAction *action);
47 // 返回给按钮设置的QAction对象
48 QAction *QToolButton::defaultAction() const;
49
50 /*
51 图标的显示样式是一个枚举类型->Qt::ToolButtonStyle，取值如下：
52     - Qt::ToolButtonIconOnly: 只有图标，不显示文本信息
53     - Qt::ToolButtonTextOnly: 不显示图标，只显示文本信息
54     - Qt::ToolButtonTextBesideIcon: 文本信息在图标的后边显示
55     - Qt::ToolButtonTextUnderIcon: 文本信息在图标的下边显示
56     - Qt::ToolButtonFollowStyle: 跟随默认样式(只显示图标)
57 */
58 // 设置的这个属性决定工具按钮是只显示一个图标、只显示文本，还是在图标旁边/下面显示文本。
59 [slot] void QToolButton::setToolButtonStyle(Qt::ToolButtonStyle style);
60 // 返回工具按钮设置的图标显示模式
61 Qt::ToolButtonStyle toolButtonStyle() const;
62
63
64 // 显示相关的弹出菜单。如果没有这样的菜单，这个函数将什么也不做。这个函数直到弹出菜单被用
65 // 户关闭才会返回。
66 [slot] void QToolButton::showMenu();

```

3.2 按钮的使用

通过 API 的介绍，我们可以知道，使用 `QToolButton` 这种类型的按钮，有三种使用方式：

1. 作为普通按钮，可以显示文本信息和图标
2. 按钮的图标可以使用不同的方式设置，并且制定图标和文本信息的显示模式
3. 设置`check`属性，使其可以处于持续的被选中状态
4. 关联一个菜单，点击按钮菜单弹出，并且可以设置菜单的弹出方式

具体操作可以参考如下代码:

```
1 MainWindow::MainWindow(QWidget *parent)
2     : QMainWindow(parent)
3     , ui(new Ui::MainWindow)
4 {
5     ui->setupUi(this);
6
7     // 普通按钮，没有checked属性
8     ui->normalBtn->setText("我是个屌丝");
9     ui->normalBtn->setIconSize(QSize(50, 50));
10    ui->normalBtn->setIcon(QIcon(":/mario.png"));
11    connect(ui->normalBtn, &QToolButton::clicked, this, [=]()
12    {
13        qDebug() << "我是一个普通按钮，是一个屌丝...";
14    });
15    // 设置图标和文本的显示模式
16    ui->normalBtn->setToolButtonStyle(Qt::ToolButtonTextBesideIcon);
17    // 基于 QAction 的方式给按钮设置图标和文本信息
18    QAction* actBtn = new QAction(QIcon(":/mushroom_life.png"), "奥利给");
19    ui->actionBtn->setDefaultAction(actBtn);
20    connect(ui->actionBtn, &QToolButton::triggered, this, [=](QAction* act)
21    {
22        act->setText("我是修改之后的马里奥...");
23        act->setIcon(QIcon(":/mario.png"));
24    });
25    // 设置图标和文本的显示模式
26    ui->actionBtn->setToolButtonStyle(Qt::ToolButtonTextUnderIcon);
27
28    // 基于自带样式，给按钮设置箭头图标
29    ui->arrowBtn->setArrowType(Qt::UpArrow);
30    ui->arrowBtn->setText("向上");
31    // 设置图标和文本的显示模式
32    ui->arrowBtn->setToolButtonStyle(Qt::ToolButtonTextUnderIcon);
33
34
35    // 有 checked 属性的按钮
36    ui->checkedBtn->setCheckable(true);
37    connect(ui->checkedBtn, &QToolButton::toggled, this, [=](bool b1)
38    {
39        qDebug() << "我是一个 checked 按钮，当前状态为:" << b1;
40    });
41
42    // 关联菜单
43    ui->menuBtn->setText("你喜欢哪种美女?");
44    QMenu* menu = new QMenu;
45    QAction* act = menu->addAction("可爱的");
46    menu->addAction("粘人的");
47    menu->addAction("胸大的");
48    menu->addAction("屁股翘的");
49    ui->menuBtn->setMenu(menu);
50    connect(act, &QAction::triggered, this, [=]{
51        qDebug() << "我是一个可爱的女人，今晚约吗?";
52    });
53
54    ui->popmenu->setMenu(menu);
55    /*
```

```

56     弹出菜单的弹出模式是一个枚举类型: QToolButton::ToolButtonPopupMode, 取值如下:
57     - QToolButton::DelayedPopup:
58         - 延时弹出, 按压工具按钮一段时间后才能弹出, 比如: 浏览器的返回按钮
59         - 长按按钮菜单弹出, 但是按钮的 clicked 信号不会被发射
60     - QToolButton::MenuButtonPopup:
61         - 在这种模式下, 工具按钮会显示一个特殊的箭头, 表示有菜单。
62         - 当按下按钮的箭头部分时, 将显示菜单。按下按钮部分发射 clicked 信号
63     - QToolButton::InstantPopup:
64         - 当按下工具按钮时, 菜单立即显示出来。
65         - 在这种模式下, 按钮本身的动作不会被触发(不会发射clicked信号
66 */
67 ui->popmenu->setPopupMode(QToolButton::MenuButtonPopup);
68 // 测试关联了菜单的按钮是否会发射clicked信号
69 connect(ui->popmenu, &QToolButton::clicked, this, [=]()
70 {
71     qDebug() << "我是popMenu按钮, 好痒呀...";
72 });
73 }

```

4. QRadioButton

`QRadioButton` 是 Qt 提供的单选按钮, 一般都是以组的方式来使用 (多个按钮中同时只能选中其中一个)。操作这个按钮使用的大部分函数都是从父类继承过来的, 它的父类是 `QAbstractButton`。

关于单选按钮的使用我们还需要注意一点, 如果单选按钮被选中, 再次点击这个按钮选中状态是不能被取消的。

4.1 常用 API

这个类混的很失败, 一直生活在父类的阴影之下, 也没有什么作为, 在官方的帮助文档中, 处理构造函数就没有再提供其他可用的 API 了

```

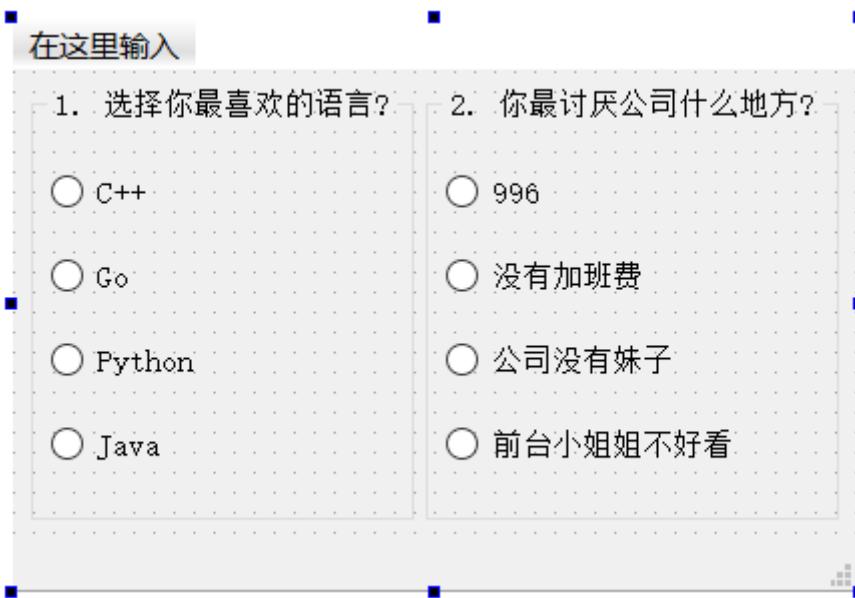
1 // 构造函数
2 /*
3 参数:
4     - text: 按钮上显示的标题
5     - parent: 按钮的父对象
6 */
7 QRadioButton::QRadioButton(const QString &text, QWidget *parent = nullptr);
8 QRadioButton::QRadioButton(QWidget *parent = nullptr);

```

4.2 按钮的使用

单选按钮一般是以组的形式来使用的, 如果在一个窗口中需要有多个单选按钮组, 应该如何处理呢?

在同一窗口中, Qt 会认为所有的单选按钮都属于同一组, 如果需要多个单选按钮组, 应该将他们放到不同的子窗口中。



通过上图可以看到有两个单选按钮组，在制作的时候分别将单选按钮放到了不同的容器窗口（组框）中，这样就被人为分隔为两组了。

对象	类
MainWindow	QMainWindow
centralwidget	QWidget
groupBox	QGroupBox 第一组
radio_cpp	QRadioButton
radio_go	QRadioButton
radio_java	QRadioButton
radio_python	QRadioButton
groupBox_2	QGroupBox 第二组
radio_nogirl	QRadioButton
radio_nosalary	QRadioButton
radio_notbeautiful	QRadioButton
radio_996	QRadioButton

如果我们使用鼠标点击了某个单选按钮，按钮还是会发射出 clicked() 信号，简单的按钮测试代码如下所示：

```
1 void MainWindow::on_radio_996_clicked()
2 {
3     qDebug() << "996";
4 }
5
6 void MainWindow::on_radio_nosalary_clicked()
7 {
8     qDebug() << "没有加班费";
9 }
10
11 void MainWindow::on_radio_nogirl_clicked()
12 {
13     qDebug() << "公司没有妹子...";
14 }
```

```

16 // clicked 信号传递的参数可以接收，也可以不接收
17 // 这个参数对应这中类型的按钮没啥用
18 void MainWindow::on_radio_notbeautiful_clicked(bool checked)
19 {
20     qDebug() << "前台小姐姐不好看!!!";
21 }

```

5. QCheckBox

QCheckBox 是 Qt 中的复选框按钮，可以单独使用，也可以以组的方式使用 (同一组可以同时选中多个)，当复选按钮被选中，再次点击之后可以取消选中状态，这一点和单选按钮是不同的。操作这个按钮使用的大部分函数都是从父类继承过来的，它的父类是 QAbstractButton。

5.1 常用 API

我们对复选框按钮操作的时候，可以设置选中和未选中状态，并且还可以设置半选中状态，这种半选中状态一般需要当前复选框按钮下还有子节点，类似一树状结构。

- 公共成员函数

```

1 // 构造函数
2 /*
3 参数：
4     - text: 按钮上显示的文本信息
5     - parent: 按钮的父对象
6 */
7 QCheckBox::QCheckBox(const QString &text, QWidget *parent = nullptr);
8 QCheckBox::QCheckBox(QWidget *parent = nullptr);
9
10 // 判断当前复选框是否为三态复选框，默认情况下为两种状态：未选中，选中
11 bool isTristate() const;
12 // 设置当前复选框为三态复选框：未选中，选中，半选中
13 void setTristate(bool y = true);
14
15 /*
16 参数 state，枚举类型 Qt::CheckState：
17     - Qt::Unchecked      --> 当前复选框没有被选中
18     - Qt::PartiallyChecked    --> 当前复选框处于半选中状态，部分被选中(三态复
19     - Qt::Checked        --> 当前复选框处于选中状态
20 */
21 // 设置复选框按钮的状态
22 void QCheckBox::setCheckstate(Qt::Checkstate state);
23 // 获取当前复选框的状态
24 Qt::CheckState QCheckBox::checkstate() const;

```

- 信号

```

1 // 当复选框的状态改变时，即当用户选中或取消选中复选框时，他的信号就会发出。
2 // 参数 state 表示的是复选框的三种状态中某一种，可参考 Qt::Checkstate
3 [signal] void QCheckBox::stateChanged(int state);

```

5.2 按钮的使用

下面针对复选框按钮的三种状态，为大家展示一下对应的操作流程，首先第一步搭建一个有树状关系的界面：



这些复选框按钮的关系以及 objectName 如下：

对象	类
MainWindow	QMainWindow
centralwidget	QWidget
groupBox	QGroupBox
widget	QWidget
ake	QCheckBox
fangyi	QCheckBox
jianning	QCheckBox
longer	QCheckBox
mujianping	QCheckBox
shuanger	QCheckBox
zengrou	QCheckBox
wives	QCheckBox

第二步，在窗口类的头文件中添加槽函数，槽函数处理复选框按钮的状态变化：

```
1 //mainwindow.h
2 QT_BEGIN_NAMESPACE
3 namespace ui { class MainWindow; }
4 QT_END_NAMESPACE
5
6 class MainWindow : public QMainWindow
7 {
8     Q_OBJECT
9
10 public:
11     MainWindow(QWidget *parent = nullptr);
12     ~MainWindow();
13
14 private slots:
```

```

15     // 添加槽函数，处理复选框按钮状态变化
16     void statusChanged(int state);
17
18 private:
19     Ui::MainWindow *ui;
20     int m_number = 0;      // 添加一个计数器，记录有几个子节点被选中了
21 };

```

第三步，在源文件中添加处理逻辑

```

1 //mainwindow.cpp
2 // 窗口的构造函数
3 MainWindow::MainWindow(QWidget *parent)
4     : QMainWindow(parent)
5     , ui(new Ui::MainWindow)
6 {
7     ui->setupUi(this);
8
9     // 设置根节点的三态属性
10    ui->wives->setTristate(true);
11    // 处理根节点的鼠标点击事件
12    connect(ui->wives, &QCheckBox::clicked, this, [=](bool b1)
13    {
14        if(b1)
15        {
16            // 子节点全部设置为选中状态
17            ui->jianning->setChecked(true);
18            ui->fangyi->setChecked(true);
19            ui->longer->setChecked(true);
20            ui->zengrou->setChecked(true);
21            ui->mujianping->setChecked(true);
22            ui->shuanger->setChecked(true);
23            ui->ake->setChecked(true);
24        }
25        else
26        {
27            // 子节点全部设置为非选中状态
28            ui->jianning->setChecked(false);
29            ui->fangyi->setChecked(false);
30            ui->longer->setChecked(false);
31            ui->zengrou->setChecked(false);
32            ui->mujianping->setChecked(false);
33            ui->shuanger->setChecked(false);
34            ui->ake->setChecked(false);
35        }
36    });
37
38    // 处理子节点的状态变化，对应的槽函数相同
39    connect(ui->jianning, &QCheckBox::stateChanged, this,
40             &MainWindow::statusChanged);
41    connect(ui->fangyi, &QCheckBox::stateChanged, this,
42             &MainWindow::statusChanged);
43    connect(ui->longer, &QCheckBox::stateChanged, this,
44             &MainWindow::statusChanged);
45    connect(ui->zengrou, &QCheckBox::stateChanged, this,
46             &MainWindow::statusChanged);

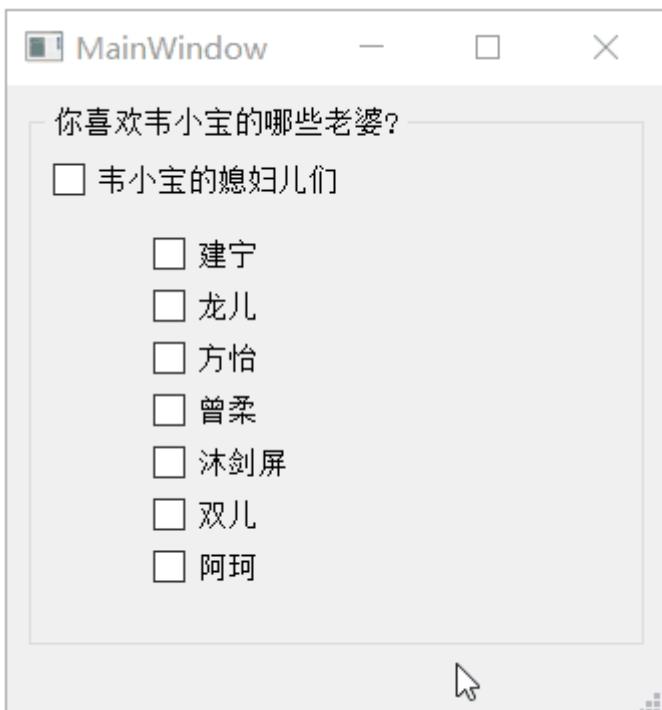
```

```

43     connect(ui->mujianping, &QCheckBox::stateChanged, this,
44         &MainWindow::statusChanged);
45     connect(ui->shuanger, &QCheckBox::stateChanged, this,
46         &MainWindow::statusChanged);
47     connect(ui->ake, &QCheckBox::stateChanged, this,
48         &MainWindow::statusChanged);
49 }
50
51 // 槽函数
52 void MainWindow::statusChanged(int state)
53 {
54     if(state == Qt::Checked)
55     {
56         m_number++;
57         // 选中一个子节点，计数器加1
58     }
59     else
60     {
61         m_number--;
62         // 取消选中一个子节点，计数器减1
63     }
64
65     // 根据计数器值判断根节点是否需要做状态的更新
66     if(m_number == 7)
67     {
68         ui->wives->setCheckState(Qt::Checked);
69     }
70     else if(m_number == 0)
71     {
72         ui->wives->setCheckState(Qt::Unchecked);
73     }
74 }

```

最终效果展示:



九、Qt中容器类型的控件

文章中主要介绍了 Qt 中常用的容器控件，包括: Widget, Frame, Group Box, Scroll Area, Tool Box, Tab Widget, Stacked Widget。

1. QWidget

关于QWidget 在前面的章节中已经介绍过了，这个类是所有窗口类的父类，可以作为独立窗口使用，也可以内嵌到其它窗口中使用。

Qt 中的所有控件都属于窗口类，因此这个类也是所有控件类的基类。

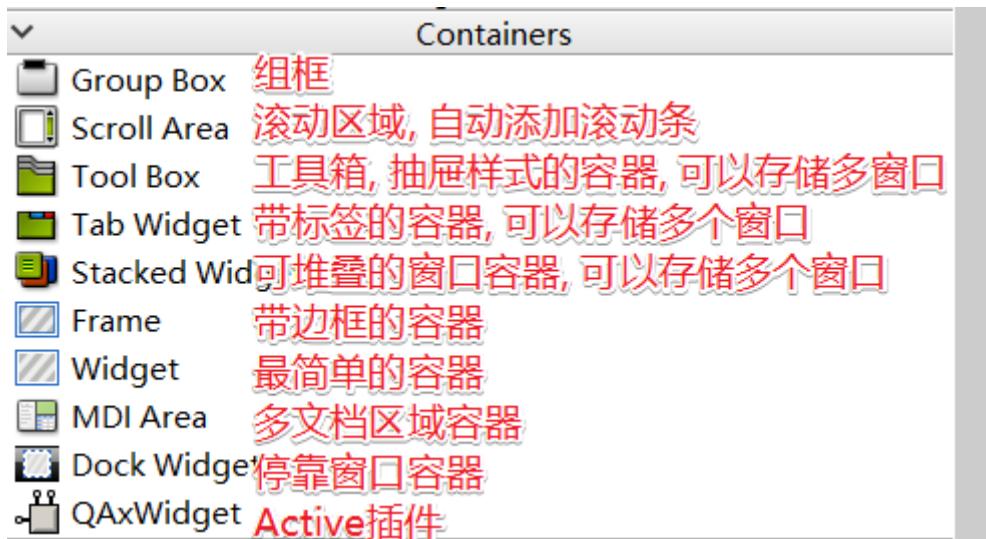
如果一个窗口中还有子窗口，为了让子窗口有序排列，这时候我们可以选择一个 QWidget 类型的容器，将子窗口放到里边，然后再给这个 QWidget 类型窗口进行布局操作。

在这里给大家介绍一下关于这个类的一些属性，因为这个类是所有窗口类的基类，因此相关属性比较多

属性	值	
> QObject		
QWidget		
windowModality	NonModal	窗口的显示模式: 模态, 非模态
enabled	<input checked="" type="checkbox"/>	窗口是否有效, 无效则无法处理触发的事件
> geometry	[0, 0, 329 x 316]	窗口当前的几何信息, 位置和宽度, 高度
> sizePolicy	[Preferred, Preferred, 0, 0]	窗口的尺寸策略, 默认自动缩放
> minimumSize	0 x 0	设置窗口的最新尺寸信息
> maximumSize	16777215 x 16777215	设置窗口的最大尺寸信息
> sizeIncrement	0 x 0	设置窗口的尺寸增量
> baseSize	0 x 0	
palette	继承	
> font	A [SimSun, 9]	设置窗口使用的字体
cursor	箭头	设置窗口中的光标样式
mouseTracking	<input type="checkbox"/>	设置窗口是否追踪鼠标移动事件
tabletTracking	<input type="checkbox"/>	设置窗口是否追踪键盘 Tab 事件
focusPolicy	NoFocus	设置窗口的焦点策略
contextMenuPolicy	DefaultContextMenu	设置窗口的右键菜单策略
acceptDrops	<input type="checkbox"/>	
> windowTitle	MainWindow	设置窗口的标题
> windowIcon		设置窗口的图标
windowOpacity	1.000000	设置窗口的透明度, 默认不透明
> toolTip		设置窗口的提示信息
toolTipDuration	-1	设置窗口提示信息持续的时长
> statusTip		
> whatsThis		
> accessibleName		
> accessibleDescription		
layoutDirection	LeftToRight	
autoFillBackground	<input type="checkbox"/>	
styleSheet		给窗口设置样式表, 进行窗口的美化
> locale	Chinese_China	

关于这些属性大部分都有对应的 API 函数，在属性名前加 set 即可，大家可以自己从 QWidget 这个类里边搜索，并仔细阅读关于这些函数的参数介绍。

在 Qt 中我们除了使用 QWidget 类型窗口作为容器使用，也可以根据实际需求选择其他类型的容器，下面看看具体都有哪些。



上述容器中，后边着重为大家介绍一些常用的，比如：Group Box, Scroll Area, Tool Box, Tab Widget, Stacked Widget, Frame, 关于 Dock Widget 在前边的章节中已经介绍过了，因此不在赘述。

2. Frame

QFrame 就是一个升级版的 QWidget, 它继承了 QWidget 的属性，并且做了拓展，这种类型的容器窗口可以提供边框，并且可以设置边框的样式、宽度以及边框的阴影。

2.1 相关 API

关于这个类的 API, 一般是不在程序中调用的，但是还是给大家介绍一下

```
1  /*
2   * 边框形状为布尔类型，可选项为：
3   * - QFrame::NoFrame: 没有边框
4   * - QFrame::Box: 绘制一个框
5   * - QFrame::Panel: 绘制一个面板，使内容显示为凸起或凹陷
6   * - QFrame::StyledPanel: 绘制一个外观取决于当前GUI样式的矩形面板。它可以上升也可以下沉。
7   * - QFrame::HLine: 画一条没有边框的水平线(用作分隔符)
8   * - QFrame::VLine: 画一条没有边框的垂直线(用作分隔符)
9   * - QFrame::WinPanel: 绘制一个矩形面板，可以像windows 2000那样向上或向下移动。
10  * 指定此形状将线宽设置为2像素。winPanel是为了兼容而提供的。
11  * 对于GUI风格的独立性，我们建议使用StyledPanel代替。
12 */
13 // 获取边框形状
14 Shape frameShape() const;
15 // 设置边框形状
16 void setFrameShape(Shape);
17 /*
18 Qt中关于边框的阴影(QFrame::Shadow)提供了3种样式，分别为：
19 - QFrame::Plain: 简单的，朴素的，框架和内容与周围环境显得水平；
20 使用调色板绘制QPalette::windowText颜色(没有任何3D效果)
21 - QFrame::Raised: 框架和内容出现凸起；使用当前颜色组的明暗颜色绘制3D凸起线
22 - QFrame::Sunken: 框架及内容物凹陷；使用当前颜色组的明暗颜色绘制3D凹线
23 */
24 // 获取边框阴影样式
25 Shadow frameShadow() const;
```

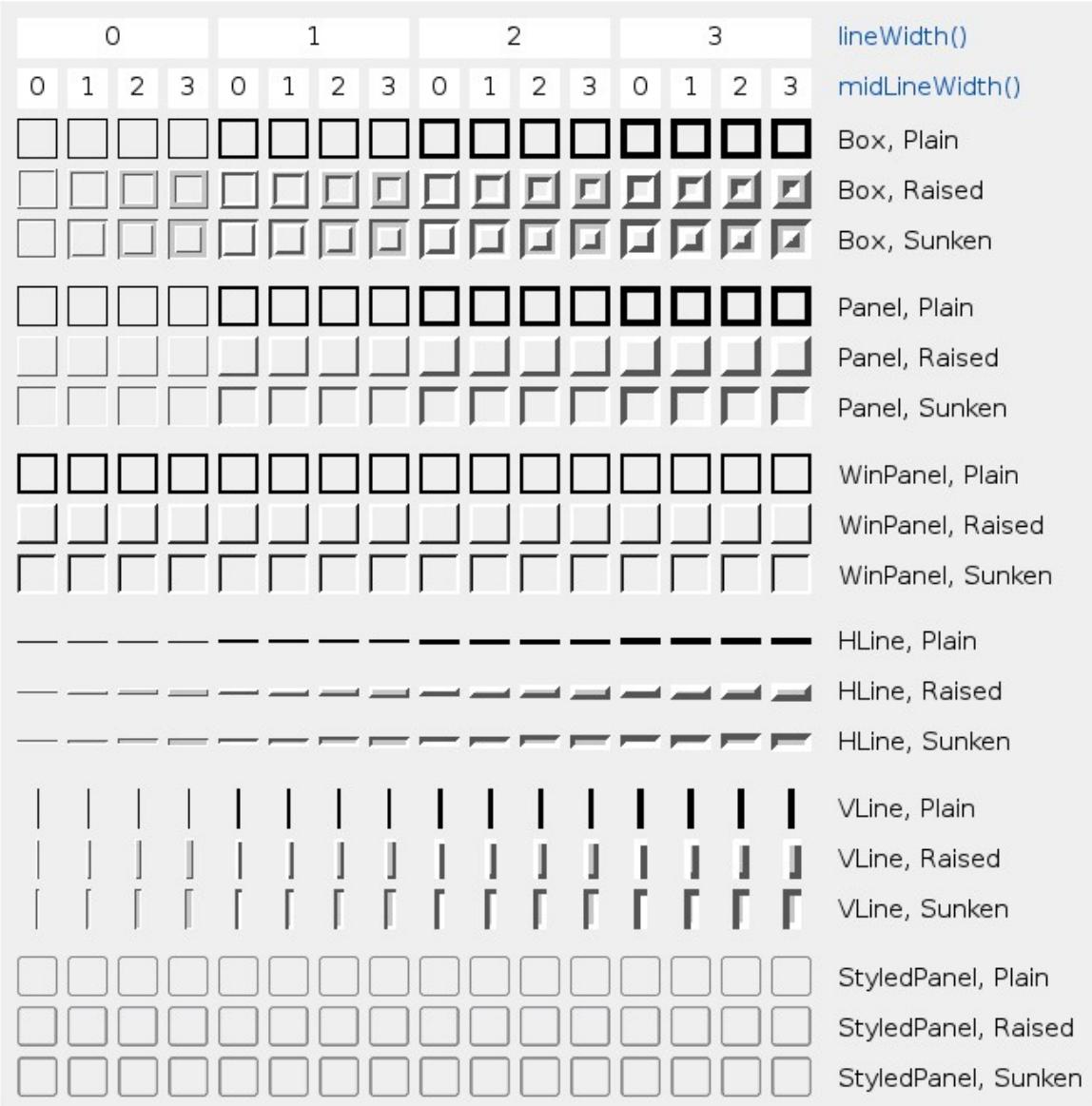
```
26 // 设置边框阴影样式
27 void setFrameShadow(Shadow);
28
29 // 得到边框线宽度
30 int lineWidth() const;
31 // 设置边框线宽度， 默认值为1
32 void setLineWidth(int);
33
34 // 得到中线的宽度
35 int midLineWidth() const;
36 // 设置中线宽度， 默认值为0， 这条线会影响边框阴影的显示
37 void setMidLineWidth(int);
```

2.2 属性设置

这个类的属性并不多，都是关于边框的设置的。

QFrame		
frameShape	Box	边框形状
frameShadow	Sunken	边框阴影样式
lineWidth	3	边框线宽
midLineWidth	1	中线宽度, 主要控制阴影

这个表格显示了一些边框样式和线宽以及阴影的组合：



3. Group Box

QGroupBox 类的基类是 QWidget, 在这种类型的窗口中可以绘制边框、给窗口指定标题，并且还支持显示复选框。

3.1 相关 API

关于这个类的 API 不常用，下面给大家介绍一下在编码过程中可能会用到的一些：

```

1 // 构造函数
2 QGroupBox::QGroupBox(QWidget *parent = Q_NULLPTR);
3 QGroupBox::QGroupBox(const QString &title, QWidget *parent = Q_NULLPTR);
4
5 // 公共成员函数
6 bool QGroupBox::isCheckable() const;
7 // 设置是否在组框中显示一个复选框
8 void QGroupBox::setCheckable(bool checkable);
9
10 /*
11  关于对齐方式需要使用枚举类型 Qt::Alignment, 其可选项为:
12   - Qt::AlignLeft: 左对齐(水平方向)

```

```

13     - Qt::AlignRight: 右对齐(水平方向)
14     - Qt::AlignHCenter: 水平居中
15     - Qt::AlignJustify: 在可用的空间内调整文本(水平方向)
16
17     - Qt::AlignTop: 上对齐(垂直方向)
18     - Qt::AlignBottom: 下对齐(垂直方向)
19     - Qt::AlignVCenter: 垂直居中
20 */
21 Qt::Alignment QGroupBox::alignment() const;
22 // 设置组框标题的对齐方式
23 void QGroupBox::setAlignment(int alignment);
24
25 QString QGroupBox::title() const;
26 // 设置组框的标题
27 void QGroupBox::setTitle(const QString &title);
28
29 bool QGroupBox::isChecked() const;
30 // 设置组框中复选框的选中状态
31 [slot] void QGroupBox::setChecked(bool checked);

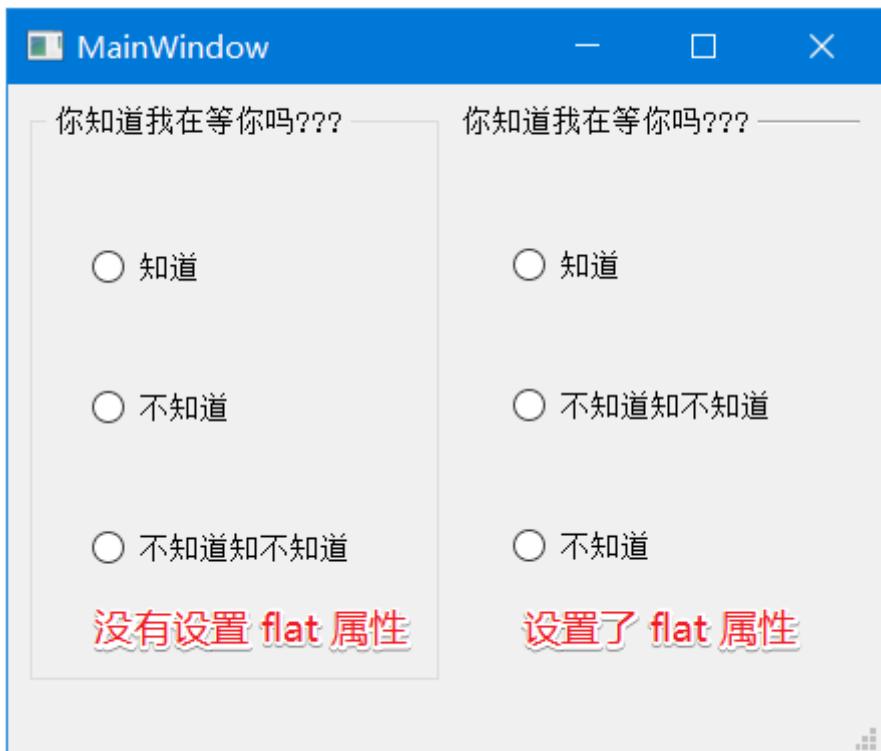
```

3.2 属性设置

关于组框的属性对应的就是上边介绍的那几个 API 函数，属性窗口如下：

QGroupBox	
title	
可翻译的	<input checked="" type="checkbox"/> 设置组框的标题
澄清	
注释	
alignment	
水平的	AlignJustify, AlignTop <input checked="" type="checkbox"/> 设置组框标题的对齐方式
垂直的	AlignTop (水平或者垂直方向)
flat	<input checked="" type="checkbox"/> 设置组框的绘制方式
checkable	<input checked="" type="checkbox"/> 设置是否给组框添加复选框
checked	<input type="checkbox"/> 设置添加的复选框的选择状态

组框中的 flat 属性没有对应的 API 函数，只能在属性窗口中设置，它控制的是窗口边框的绘制方式，如果打开该属性，组框的边框就消失了，效果如下：



4. Scroll Area

QScrollArea 这种类型的容器，里边可以放置一些窗口控件，当放置的窗口控件大于当前区域导致无法全部显示的时候，滚动区域容器会自动添加相应的滚动条(水平方向或者垂直方向)，保证放置到该区域中的所有窗口内容都可以正常显示出来。对于使用者不需要做太多事情，只需要把需要显示的窗口放到滚动区域中就行了。

4.1 相关 API

在某些特定环境下，我们需要动态的往滚动区域内部添加要显示的窗口，或者动态的将显示的窗口移除，这时候就必须要调用对应的 API 函数来完成这部分操作了。主要 API 有两个 添加 - setWidget(), 移除 - takeWidget()

```
1 // 构造函数
2 QScrollArea::QScrollArea(QWidget *parent = Q_NULLPTR);
3
4 // 公共成员函数
5 // 给滚动区域设置要显示的子窗口widget
6 void QScrollArea::setWidget(QWidget *widget);
7 // 删除滚动区域中的子窗口，并返回被删除的子窗口对象
8 QWidget *QScrollArea::takeWidget();
9
10 /*
11 关于显示位置的设定，是一个枚举类型，可选项为：
12     - Qt::AlignLeft: 左对齐
13     - Qt::AlignHCenter: 水平居中
14     - Qt::AlignRight: 右对齐
15     - Qt::AlignTop: 顶部对齐
16     - Qt::AlignVCenter: 垂直对其
17     - Qt::AlignBottom: 底部对其
18 */
19 // 获取子窗口在滚动区域中的显示位置
20 Qt::Alignment alignment() const;
21 // 设置滚动区域中子窗口的对其方式，默认显示的位置是右上
```

```

22 void setAlignment(Qt::Alignment);
23
24 // 判断滚动区域是否有自动调节小部件大小的属性
25 bool widgetResizable() const;
26 /*
27 1. 设置滚动区域是否应该调整视图小部件的大小，该属性默认为false，滚动区域按照小部件的默认
28 大小进行显示。
29 2. 如果该属性设置为true，滚动区域将自动调整小部件的大小，避免滚动条出现在本可以避免的地
30 方，或者利用额外的空间。
31 3. 不管这个属性是什么，我们都可以使用widget()->resize()以编程方式调整小部件的大小，滚动
32 区域将自动调整自己以适应新的大小。
33 */
34 void setWidgetResizable(bool resizable);

```

4.2 属性设置

关于滚动区域，其属性窗口提供的属性一般不需要设置，因为一般情况下即便是设置了也看不到效果，即便如此，还是看一下吧。至于具体原因在视频中有详细的说明，感兴趣的可以去看一下。

QScrollArea	
widgetResizable	<input checked="" type="checkbox"/> 是否自动调整滚动区域内部窗口大小
alignment	AlignLeft, AlignVCenter
水平的	AlignLeft 子窗口在滚动区域
垂直的	AlignVCenter 内部的对齐方式

4.3 窗口的动态添加和删除

关于窗口的滚动区域对象创建有两种方式，第一种比较简单在编辑页面直接拖拽一个控件到 UI 界面，然后布局即可。第二种方式是在程序中通过 new 操作创建一个实例对象，然后通过通过代码的方式将其添加到窗口的某个布局中，相对来说要麻烦一点。

下面通过第一种方式，演示一下如果往滚动区域中添加多个子窗口。

```

1 MainWindow::MainWindow(QWidget *parent)
2     : QMainWindow(parent)
3     , ui(new Ui::MainWindow)
4 {
5     ui->setupUi(this);
6
7     // 创建一个垂直布局对象
8     QVBoxLayout* vlayout = new QVBoxLayout;
9
10    for(int i=0; i<11; ++i)
11    {
12        // 创建标签对象
13        QLabel* pic = new QLabel;
14        // 拼接图片在资源文件中的路径
15        QString name = QString(":/images/%1.png").arg(i+1);
16        // 给标签对象设置显示的图片
17        pic->setPixmap(QPixmap(name));
18        // 设置图片在便签内部的对其方式
19        pic->setAlignment(Qt::AlignHCenter);
20        // 将标签添加到垂直布局中
21        vlayout->addWidget(pic);

```

```
22 }
23
24 // 创建一个窗口对象
25 QWidget* wg = new QWidget;
26 // 将垂直布局设置给窗口对象
27 wg->setLayout(vlayout);
28 // 将带有垂直布局的窗口设置到滚动区域中
29 ui->scrollArea->setWidget(wg);
30 }
```

关于以上代码做以下说明，调用 setWidget(wg)之后，wg会自动平铺填充整个滚动区域，因此：

- 在程序中调用 void setWidgetResizable(bool resizable); 不会有明显效果
- 在程序中调用 void setAlignment(Qt::Alignment); 不会看到任何效果
- 如果要设置显示的图片的对其方式要设置图片的载体对象即 标签签对象
- 如果要动态移除滚动区域中的窗口，直接使用滚动区域对象调用 takeWidget() 即可
- 滚动区域中只能通过 setWidget(wg) 添加一个子窗口，如果要添加多个可使用布局的方式来实现

代码效果展示：



5. Tool Box

QToolBox 工具箱控件，可以存储多个子窗口，该控件可以实现类似 QQ 的抽屉效果，每一个抽屉都可以设置图标和标题，并且对应一个子窗口，通过抽屉按钮就可以实现各个子窗口显示的切换。

5.1 相关 API

这个类对应的 API 函数相对较多，一部分是控件属性对应的属性设置函数，一部分是编程过程中可能会用的到的，怎么说呢，理解为主吧，知道有这么函数即可。

```
1 // 构造函数
2 QToolBox::QToolBox(QWidget *parent = Q_NULLPTR, Qt::WindowFlags f =
3 Qt::WindowFlags());
4
5 // 公共成员
6 /*
7 addItem(), insertItem() 函数相关参数：
8 - widget: 添加到工具箱中的选项卡对应的子窗口对象
9 - icon: 工具箱新的选项卡上显示的图标
10 - text: 工具箱新的选项卡上显示的标题
11 - index: 指定在工具箱中插入的新的选项卡的位置
12 */
13 // 给工具箱尾部添加一个选项卡，每个选项卡在工具箱中就是一个子窗口，即参数widget
14 int QToolBox::addItem(QWidget *widget, const QString &text);
15 int QToolBox::addItem(QWidget *widget, const QIcon &icon, const QString
&text);
16 // 在工具箱的指定位置添加一个选项卡，即添加一个子窗口
17 int QToolBox::insertItem(int index, QWidget *widget, const QString &text);
18 int QToolBox::insertItem(int index, QWidget *widget, const QIcon &icon,
const QString &text);
19 // 移除工具箱中索引index位置对应的选项卡，注意：只是移除对应的窗口对象并没有被销毁
20 void QToolBox::removeItem(int index);
21
22 // 设置索引index位置的选项卡是否可用，参数 enabled=true 为可用，enabled=false 为禁用
23 void QToolBox::setItemEnabled(int index, bool enabled);
24 // 设置工具箱中index位置选项卡的图标
25 void QToolBox::setItemIcon(int index, const QIcon &icon);
26 // 设置工具箱中index位置选项卡的标题
27 void QToolBox::setItemText(int index, const QString &text);
28 // 设置工具箱中index位置选项卡的提示信息(需要鼠标在选项卡上悬停一定时长才能显示)
29 void QToolBox::setItemToolTip(int index, const QString &toolTip);
30
31 // 如果位置索引的项已启用，则返回true；否则返回false。
32 bool QToolBox::isItemEnabled(int index) const;
33 // 返回位置索引处项目的图标，如果索引超出范围，则返回空图标。
34 QIcon QToolBox::itemIcon(int index) const;
35 // 返回位于位置索引处的项的文本，如果索引超出范围，则返回空字符串。
36 QString QToolBox::itemText(int index) const;
37 // 返回位于位置索引处的项的工具提示，如果索引超出范围，则返回空字符串。
38 QString QToolBox::itemToolTip(int index) const;
39
40 // 得到当前工具箱中显示的选项卡对应的索引
41 int QToolBox::currentIndex() const;
42 // 返回指向当前选项卡对应的子窗口的指针，如果没有这样的项，则返回0。
43 QWidget *QToolBox::currentWidget() const;
44 // 返回工具箱中子窗口的索引，如果widget对象不存在，则返回-1
45 int QToolBox::indexOf(QWidget *widget) const;
46 // 返回工具箱中包含的项的数量。
```

```

47 int QToolBox::count() const;
48
49 // 信号
50 // 工具箱中当前显示的选项卡发生变化，该信号被发射，index为当前显示的新的选项卡的对应的索引
51 [signal] void QToolBox::currentChanged(int index);
52
53 // 槽函数
54 // 通过工具箱中选项卡对应的索引设置当前要显示哪一个选项卡中的子窗口
55 [slot] void QToolBox::setCurrentIndex(int index);
56 // 通过工具箱中选项卡对应的子窗口对象设置当前要显示哪一个选项卡中的子窗口
57 [slot] void QToolBox::setCurrentWidget(QWidget *widget);

```

5.2 属性设置

关于这个容器控件的属性远比上边介绍的 API 要少，来看一看吧

▼ QToolBox	
currentIndex	1 工具箱中当前选项卡对应的索引, 从0开始
▶ currentItemText	Page 2 工具箱中当前选项卡上显示的标题
currentItemName	page_4 当前选项卡对应的子窗口的objectName
▶ currentItemIcon	当前选项卡上显示的图标, 默认没有, 可以设置
▶ currentItemToolTip	当前选项卡显示的提示信息
tabSpacing	20 工具箱中窗口折叠后, 选项卡之间的间隙

6. Tab Widget

QTabWidget 的一种带标签页的窗口，在这种类型的窗口中可以存储多个子窗口，每个子窗口的显示可以通过对应的标签进行切换。

6.1 相关 API

介绍的这些 API 大部分是进行属性设置的，因此我们可以完全不在程序中使用这些函数，通过属性窗口进行设置，但是 API 操作比较灵活，可以动态的设置相关属性。先来看公共成员函数：

```

1 // 构造函数
2 QTabWidget::QTabWidget(QWidget *parent = Q_NULLPTR);
3
4 // 公共成员函数
5 /*
6 添加选项卡addTab()或者插入选项卡insertTab()函数相关的参数如下：
7     - page: 添加或者插入的选项卡对应的窗口实例对象
8     - label: 添加或者插入的选项卡的标题
9     - icon: 添加或者插入的选项卡的图标
10    - index: 将新的选项卡插入到索引index的位置上
11 */
12 int QTabWidget::addTab(QWidget *page, const QString &label);
13 int QTabWidget::addTab(QWidget *page, const QIcon &icon, const QString
&label);
14 int QTabWidget::insertTab(int index, QWidget *page, const QString &label);
15 int QTabWidget::insertTab(int index, QWidget *page,
16                           const QIcon &icon, const QString &label);

```

```
17 // 删除index位置的选项卡
18 void QTabWidget::removeTab(int index);
19
20 // 得到选项卡栏中的选项卡的数量
21 int count() const;
22 // 从窗口中移除所有页面，但不删除它们。调用这个函数相当于调用removeTab(), 直到选项卡小部件为空为止。
23 void QTabWidget::clear();
24 // 获取当前选项卡对应的索引
25 int QTabWidget::currentIndex() const;
26 // 获取当前选项卡对应的窗口对象地址
27 QWidget *QTabWidget::currentWidget() const;
28 // 返回索引位置为index的选项卡页，如果索引超出范围则返回0。
29 QWidget *QTabWidget::widget(int index) const;
30
31 /*
32 标签上显示的文本样式为枚举类型 Qt::TextElideMode，可选项为：
33 - Qt::ElideLeft: 省略号应出现在课文的开头，例如：.....是的，我很帅。
34 - Qt::ElideRight: 省略号应出现在文本的末尾，例如：我帅吗.....。
35 - Qt::ElideMiddle: 省略号应出现在文本的中间，例如：我帅.....很帅。
36 - Qt::ElideNone: 省略号不应出现在文本中
37 */
38 // 获取标签上显示的文本模式
39 Qt::TextElideMode QTabWidget::elideMode() const;
40 // 如何省略标签栏中的文本，此属性控制在给定的选项卡栏大小没有足够的空间显示项时如何省略项。
41 void QTabWidget::setElideMode(Qt::TextElideMode);
42
43 // 得到选项卡上图标的尺寸信息
44 QSize QTabWidget::iconSize() const
45 // 设置选项卡上显示的图标大小
46 void QTabWidget::setIconSize(const QSize &size)
47
48 // 判断用户是否可以在选项卡区域内移动选项卡，可以返回true，否则返回false
49 bool QTabWidget::isMovable() const;
50 // 此属性用于设置用户是否可以在选项卡区域内移动选项卡。默认情况下，此属性为false；
51 void QTabWidget::setMovable(bool movable);
52
53 // 判断选项卡是否可以自动隐藏，如果可以自动隐藏返回true，否则返回false
54 bool QTabWidget::tabBarAutoHide() const;
55 // 如果为true，则当选项卡栏包含少于2个选项卡时，它将自动隐藏。默认情况下，此属性为false。
56 void QTabWidget::setTabBarAutoHide(bool enabled);
57
58 // 判断index对应的选项卡是否是被启用的，如果是被启用的返回true，否则返回false
59 bool QTabWidget::isTabEnabled(int index) const;
60 // 如果enable为true，则在索引位置的页面是启用的；否则，在位置索引处的页面将被禁用。
61 void QTabWidget::setTabEnabled(int index, bool enable);
62
63 // 得到index位置的标签对应的图标
64 QIcon QTabWidget::tabIcon(int index) const;
65 // 在位置索引处设置标签的图标。
66 void QTabWidget::setTabIcon(int index, const QIcon &icon);
67
68 /*
69 选项卡标签的位置通过枚举值进行指定，可使用的选项如下：
70 - QTabWidget::North: 北(上)，默认
71 - QTabWidget::South: 南(下)
```

```
72     - QTabWidget::West: 西(左)
73     - QTabWidget::East: 东(右)
74 */
75 // 得到选项卡中显示的标签的位置, 即: 东, 西, 南, 北
76 TabPosition QTabWidget::tabPosition() const;
77 // 设置选项卡中标签显示的位置, 默认情况下, 此属性设置为North。
78 void QTabWidget::setTabPosition(TabPosition);
79
80 /*
81 选项卡标签的形状通过枚举值进行指定, 可使用的选项如下:
82     - QTabWidget::Rounded: 标签以圆形的外观绘制。这是默认形状
83     - QTabWidget::Triangular: 选项卡以三角形外观绘制。
84 */
85 // 获得选项卡标签的形状
86 TabShape QTabWidget::tabShape() const;
87 // 设置选项卡标签的形状
88 void QTabWidget::setTabShape(TabShape s);
89
90 // 得到index位置的标签的标题
91 QString QTabWidget::tabText(int index) const;
92 // 设置选项卡index位置的标签的标题
93 void QTabWidget::setTabText(int index, const QString &label);
94
95 // 获取index对应的标签页上设置的提示信息
96 QString QTabWidget::tabToolTip(int index) const;
97 // 设置选项卡index位置的标签的提示信息(鼠标需要悬停在标签上一定时长才能显示)
98 void QTabWidget::setTabToolTip(int index, const QString &tip);
99
100
101 // 判断选项卡标签页上是否有关闭按钮, 如果有返回true, 否则返回false
102 bool QTabWidget::tabsClosable() const;
103 // 设置选项卡的标签页上是否显示关闭按钮, 该属性默认情况下为false
104 void QTabWidget::setTabsClosable(bool closeable);
105
106 // 判断选项卡栏中是否有滚动按钮, 如果有返回true, 否则返回false
107 bool QTabWidget::usesScrollButtons() const;
108 // 设置选项卡栏有许多标签时, 它是否应该使用按钮来滚动标签。
109 // 当一个选项卡栏有太多的标签时, 选项卡栏可以选择扩大它的大小, 或者添加按钮, 让标签在选项
110 // 卡栏中滚动。
111 void QTabWidget::setUsesScrollButtons(bool useButtons);
112
113 // 判断窗口是否设置了文档模式, 如果设置了返回true, 否则返回false
114 bool QTabWidget::documentMode() const;
115 // 此属性保存选项卡小部件是否以适合文档页面的模式呈现。这与macOS上的文档模式相同。
116 // 不设置该属性, QTabWidget窗口是带边框的, 如果设置了该属性边框就没有了。
117 void QTabWidget::setDocumentMode(bool set);
```

信号

```

1 // 每当当前页索引改变时，就会发出这个信号。参数是新的当前页索引位置，如果没有新的索引位置，则为-1
2 [signal] void QTabWidget::currentChanged(int index);
3 // 当用户单击索引处的选项卡时，就会发出这个信号。index指所单击的选项卡，如果光标下没有选项卡，则为-1。
4 [signal] void QTabWidget::tabBarClicked(int index)
5 // 当用户双击索引上的一个选项卡时，就会发出这个信号。
6 // index是单击的选项卡的索引，如果光标下没有选项卡，则为-1。
7 [signal] void QTabWidget::tabBarDoubleClicked(int index);
8 // 此信号在单击选项卡上的close按钮时发出。索引是应该被删除的索引。
9 [signal] void QTabWidget::tabCloseRequested(int index);

```

槽函数

```

1 // 设置当前窗口中显示选项卡index位置对应的标签页内容
2 [slot] void QTabWidget::setCurrentIndex(int index);
3 // 设置当前窗口中显示选项卡中子窗口widget中的内容
4 [slot] void QTabWidget::setCurrentWidget(QWidget *widget);

```

6.2 属性设置

容器类型的控件其大多数情况下都是直接在属性窗口中直接设置，因为这些属性设置完毕之后，就无需再做修改了，程序运行过程中无需做任何变化。下图为大家标注了每个属性对应的功能。

▶ QWidget	
▼ QTabWidget	
tabPosition	North 标签在窗口中的位置，上北下南，左西右东
tabShape	Rounded 标签的形状，有圆形和三角形可选择
currentIndex	0 当前选中的标签对应的索引
▶ iconSize	16 x 16 标签上显示的图标大小
elideMode	ElideNone 标签上的文本信息省略方式
usesScrollButtons	<input checked="" type="checkbox"/> 标签页太多无法全部显示时，是否添加滚动按钮
documentMode	<input type="checkbox"/> 文档模式是否开启，如果开启窗口边框会被去掉
tabsClosable	<input type="checkbox"/> 标签页上是否添加关闭按钮
movable	<input type="checkbox"/> 标签页是否可以移动
tabBarAutoHide	<input type="checkbox"/> 当标签 < 2 时，标签栏是否自动隐藏
▶ currentTabText	Tab 1 当前标签上显示的文本信息
currentTabName	tab 当前标签对应的窗口的objectName
▶ currentTabIcon	当前标签上显示的图标
▶ currentTabToolTip	当前标签的提示信息
▶ currentTabWhatsThis	

6.3 控件使用

关于这个控件的使用，主要是通过代码的方式演示一下相关信号发射的时机，再有就是当标签页添加了关闭按钮并点击了该按钮，如果移除该标签页已经如何将其再次添加到窗口中。

第一步，在头文件中添加存储已关闭的标签对应的窗口对象和标签标题的容器

```

1 // mainwindow.h

```

```

2 QT_BEGIN_NAMESPACE
3 namespace ui { class MainWindow; }
4 QT_END_NAMESPACE
5
6 class MainWindow : public QMainWindow
7 {
8     Q_OBJECT
9
10    public:
11        MainWindow(QWidget *parent = nullptr);
12        ~MainWindow();
13
14    private:
15        Ui::MainWindow *ui;
16        QQueue<QWidget*> m_widgets; // 存储标签对应的窗口对象
17        QQueue<QString> m_names; // 存储标签标题
18 };

```

第二步在源文件中添加处理动作

```

1 // mainwindow.cpp
2 MainWindow::MainWindow(QWidget *parent)
3     : QMainWindow(parent)
4     , ui(new Ui::MainWindow)
5 {
6     ui->setupUi(this);
7
8     // 点击了标签上的关闭按钮
9     connect(ui->tabWidget, &QTabWidget::tabCloseRequested, this, [=](int
index)
10    {
11        // 保存信息
12        QWidget* wg = ui->tabWidget->widget(index);
13        QString title = ui->tabWidget->tabText(index);
14        m_widgets.enqueue(wg);
15        m_names.enqueue(title);
16        // 移除tab页
17        ui->tabWidget->removeTab(index);
18        ui->addBtn->setEnabled(true);
19
20    });
21
22     // 当标签被点击了之后的处理动作
23     connect(ui->tabWidget, &QTabWidget:: tabBarClicked, this, [=](int index)
24     {
25         qDebug() << "我被点击了一下，我的标题是：" << ui->tabWidget-
>tabText(index);
26     });
27
28     // 切换标签之后的处理动作
29     connect(ui->tabWidget, &QTabWidget::currentChanged, this, [=](int
index)
30     {
31         qDebug() << "当前显示的tab页，我的标题是：" << ui->tabWidget-
>tabText(index);
32     });
33

```

```

34 // 点击添加标签按钮点击之后的处理动作
35 connect(ui->addBtn, &QPushButton::clicked, this, [=]()
36 {
37     // 将被删除的标签页添加到窗口中
38     // 1. 知道窗口对象，窗口的标题
39     // 2. 知道添加函数
40     ui->tabWidget->addTab(m_widgets.dequeue(), m_names.dequeue());
41     if(m_widgets.empty())
42     {
43         ui->addBtn->setDisabled(true);
44     }
45 });
46 }

```

测试代码效果演示:



7. Stacked Widget

QStackedWidget 栈类型窗口，在这种类型的窗口中可以存储多个子窗口，但是只有其中某一个可以被显示出来，至于是哪个子窗口被显示，需要在程序中进行控制，在这种类型的窗口中没有直接切换子窗口的按钮或者标签。

7.1 相关 API

先来了解一些这个类为我们提供的 API，在这些函数中最常用的就是它的槽函数，并且名字和 QToolBox, QTabWidget 两个类提供的槽函数名字相同 分别为 currentIndex(int), setCurrentWidget(QWidget*) 用来设置当前显示的窗口。

```

1 // 构造函数
2 QStackedWidget::QStackedWidget(QWidget *parent = Q_NULLPTR);
3
4 // 公共成员函数
5 // 在栈窗口中后边添加一个子窗口，返回这个子窗口在栈窗口中的索引值(从0开始计数)
6 int QStackedWidget::addWidget(QWidget *widget);
7 // 将子窗口widget插入到栈窗口的index位置
8 int QStackedWidget::insertWidget(int index, QWidget *widget);
9 // 将子窗口widget从栈窗口中删除
10 void QStackedWidget::removeWidget(QWidget *widget);
11
12 // 返回栈容器窗口中存储的子窗口的个数
13 int QStackedWidget::count() const;
14 // 得到当前栈窗口中显示的子窗口的索引
15 int QStackedWidget::currentIndex() const;
16 // 得到当前栈窗口中显示的子窗口的指针(窗口地址)
17 QWidget *QStackedWidget::currentWidget() const;
18 // 基于索引index得到栈窗口中对应的子窗口的指针
19 QWidget *QStackedWidget::widget(int index) const;
20 // 基于子窗口的指针(实例地址)得到其在栈窗口中的索引
21 int QStackedWidget::indexOf(QWidget *widget) const;
22
23 // 信号
24 // 切换栈窗口中显示子窗口，该信息被发射出来，index为新的当前窗口对应的索引值
25 [signal] void QStackedWidget::currentChanged(int index);
26 // 当栈窗口的子窗口被删除，该信号被发射出来，index为被删除的窗口对应的索引值
27 [signal] void QStackedWidget::widgetRemoved(int index);
28
29 // 槽函数
30 // 基于子窗口的index索引指定当前栈窗口中显示哪一个子窗口
31 [slot] void QStackedWidget::setCurrentIndex(int index);
32 [slot] void QStackedWidget::setCurrentWidget(QWidget *widget);

```

7.2 属性设置

因为栈类型的窗口容器很简单，所以对应的属性页很少，只有两个：

QStackedWidget		
currentIndex	0	当前显示的子窗口对应的索引
currentPageName	page_3	当前显示的窗口对应的objectName

7.3 控件使用

这里主要给大家演示一下 QStackedWidget 类型的容器中的子窗口如何切换，如下图所示，我们在一个栈窗口容器中添加了两个子窗口，通过两个按钮对这两个窗口进行切换

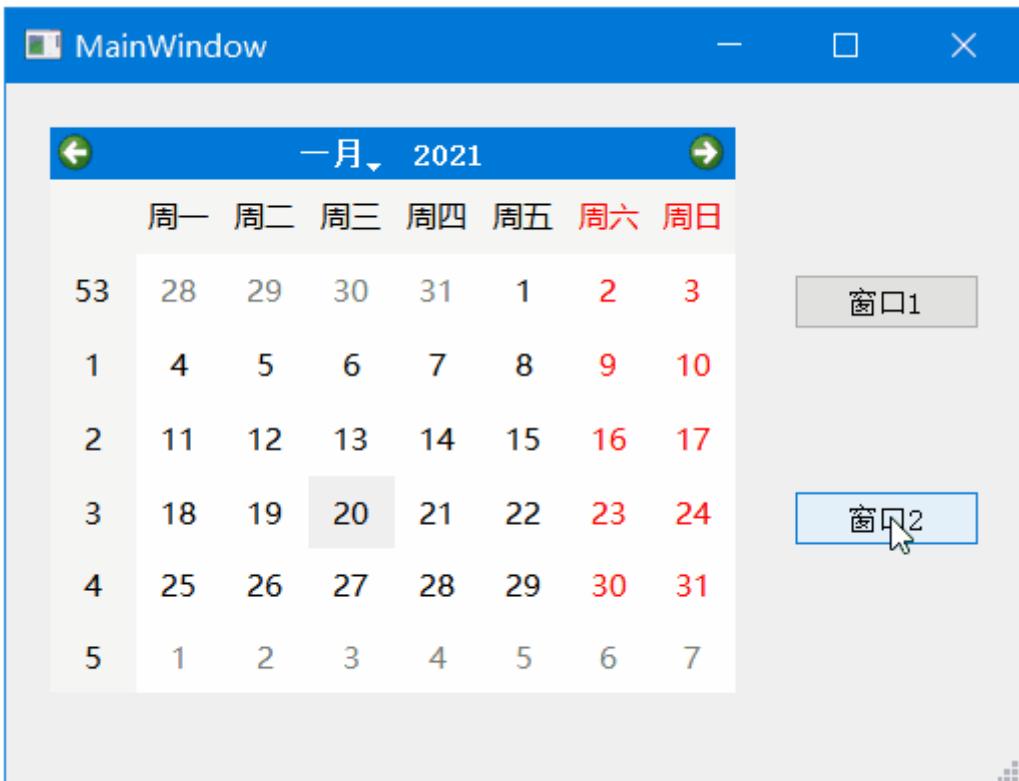
对象	类
MainWindow	QMainWindow
centralwidget	QWidget
stackedWidget	QStackedWidget
window1	QWidget
calendarWidget	QCalendarWidget
window2	QWidget
dial	QDial
widget	QWidget
showWin1	QPushButton
showWin2	QPushButton
menubar	QMenuBar
statusbar	QStatusBar

关于窗口的切换调用这个类的槽函数就可以了，代码如下：

```

1 Mainwindow::MainWindow(QWidget *parent)
2   : QMainWindow(parent)
3   , ui(new Ui::MainWindow)
4 {
5   ui->setupUi(this);
6   // 设置默认显示的窗口
7   ui->stackedWidget->setCurrentWidget(ui->window1);
8
9   connect(ui->showWin1, &QPushButton::clicked, this, [=]()
10  {
11    // 切换显示第一个子窗口
12    ui->stackedWidget->setcurrentIndex(0);
13  });
14
15  connect(ui->showWin2, &QPushButton::clicked, this, [=]()
16  {
17    // 切换显示第二个子窗口，调用这两个槽函数中的任何一个都可以
18    ui->stackedWidget->setCurrentWidget(ui->window2);
19  });
20 }
```

代码效果展示：



十、Qt中多线程的使用

在进行桌面应用程序开发的时候，假设应用程序在某些情况下需要处理比较复杂的逻辑，如果只有一个线程去处理，就会导致窗口卡顿，无法处理用户的相关操作。这种情况下就需要使用多线程，其中一个线程处理窗口事件，其他线程进行逻辑运算，多个线程各司其职，不仅可以提高用户体验还可以提升程序的执行效率。

在 qt 中使用了多线程，有些事项是需要额外注意的：

- 默认的线程在Qt中称之为窗口线程，也叫主线程，负责窗口事件处理或者窗口控件数据的更新
- 子线程负责后台的业务逻辑处理，子线程中不能对窗口对象做任何操作，这些事情需要交给窗口线程处理
- 主线程和子线程之间如果要进行数据的传递，需要使用Qt中的信号槽机制

1. 线程类 QThread

Qt 中提供了一个线程类，通过这个类就可以创建子线程了，Qt 中一共提供了两种创建子线程的方式，后边会依次介绍其使用方式。先来看一下这个类中提供的一些常用 API 函数：

1.1 常用共用成员函数

```
1 // QThread 类常用 API
2 // 构造函数
3 QThread::QThread(QObject *parent = Q_NULLPTR);
4 // 判断线程中的任务是不是处理完毕了
5 bool QThread::isFinished() const;
6 // 判断子线程是不是在执行任务
7 bool QThread::isRunning() const;
8
9 // Qt中的线程可以设置优先级
10 // 得到当前线程的优先级
```

```

11 Priority QThread::priority() const;
12 void QThread::setPriority(Priority priority);
13 优先级:
14     QThread::IdlePriority          --> 最低的优先级
15     QThread::LowestPriority
16     QThread::LowPriority
17     QThread::NormalPriority
18     QThread::HighPriority
19     QThread::HighestPriority
20     QThread::TimeCriticalPriority
21     QThread::InheritPriority      --> 最高的优先级, 默认是这个
22
23
24 // 退出线程, 停止底层的事件循环
25 // 退出线程的工作函数
26 void QThread::exit(int returnCode = 0);
27 // 调用线程退出函数之后, 线程不会马上退出因为当前任务有可能还没有完成, 调回用这个函数是
28 // 等待任务完成, 然后退出线程, 一般情况下会在 exit() 后边调用这个函数
29 bool QThread::wait(unsigned long time = ULONG_MAX);

```

1.2 信号槽

```

1 // 和调用 exit() 效果是一样的
2 // 代用这个函数之后, 再调用 wait() 函数
3 [slot] void QThread::quit();
4 // 启动子线程
5 [slot] void QThread::start(Priority priority = InheritPriority);
6 // 线程退出, 可能是会马上终止线程, 一般情况下不使用这个函数
7 [slot] void QThread::terminate();
8
9 // 线程中执行的任务完成了, 发出该信号
10 // 任务函数中的处理逻辑执行完毕了
11 [signal] void QThread::finished();
12 // 开始工作之前发出这个信号, 一般不使用
13 [signal] void QThread::started();

```

1.3 静态函数

```

1 // 返回一个指向管理当前执行线程的QThread的指针
2 [static] QThread *QThread::currentThread();
3 // 返回可以在系统上运行的理想线程数 == 和当前电脑的 CPU 核心数相同
4 [static] int QThread::idealThreadCount();
5 // 线程休眠函数
6 [static] void QThread::msleep(unsigned long msecs); // 单位: 毫秒
7 [static] void QThread::sleep(unsigned long secs);    // 单位: 秒
8 [static] void QThread::usleep(unsigned long usecs); // 单位: 微秒

```

1.4 任务处理函数

```

1 // 子线程要处理什么任务, 需要写到 run() 中
2 [virtual protected] void QThread::run();

```

这个 run() 是一个虚函数，如果想让创建的子线程执行某个任务，需要写一个子类让其继承 QThread，并且在子类中重写父类的 run() 方法，函数体就是对应的任务处理流程。另外，这个函数是一个受保护的成员函数，不能够在类的外部调用，如果想要让线程执行这个函数中的业务流程，需要通过当前线程对象调用槽函数 start() 启动子线程，当子线程被启动，这个 run() 函数也就在线程内部被调用了。

2. 使用方式 1

2.1 操作步骤

Qt 中提供的多线程的第一种使用方式的特点是：简单。操作步骤如下：

1. 需要创建一个线程类的子类，让其继承 QT 中的线程类 QThread，比如：

```
1 class MyThread:public QThread  
2 {  
3     .....  
4 }
```

2. 重写父类的 run () 方法，在该函数内部编写子线程要处理的具体的业务流程

```
1 class MyThread:public QThread  
2 {  
3     .....  
4     protected:  
5         void run()  
6         {  
7             .....  
8         }  
9 }
```

3. 在主线程中创建子线程对象，new 一个就可以了

```
1 MyThread * subThread = new MyThread;
```

4. 启动子线程，调用 start () 方法

```
1 subThread->start();
```

不能在类的外部调用 run () 方法启动子线程，在外部调用 start () 相当于让 run () 开始运行

当子线程别创建出来之后，父子线程之间的通信可以通过信号槽的方式，注意事项：

- 在 Qt 中在子线程中不要操作程序中的窗口类型对象，不允许，如果操作了程序就挂了
- 只有主线程才能操作程序中的窗口对象，默认的线程就是主线程，自己创建的就是子线程

2.2 示例代码

举一个简单的数数的例子，假如只有一个线程，让其一直数数，会发现数字并不会在窗口中时时更新，并且这时候如果用户使用鼠标拖动窗口，就会出现无响应的情况，使用多线程就不会出现这样的现象了。

在下面的窗口中，点击按钮开始在子线程中数数，让后通过信号槽机制将数据传递给 UI 线程，通过 UI 线程将数据更新到窗口中。



mythread.h

```
1 #ifndef MYTHREAD_H
2 #define MYTHREAD_H
3
4 #include <QThread>
5
6 class MyThread : public QThread
7 {
8     Q_OBJECT
9 public:
10     explicit MyThread(QObject *parent = nullptr);
11
12 protected:
13     void run();
14
15 signals:
16     // 自定义信号，传递数据
17     void curNumber(int num);
18
19 public slots:
20 };
21
22 #endif // MYTHREAD_H
```

mythread.cpp

```
1 #include "mythread.h"
2 #include <QDebug>
3
4 MyThread::MyThread(QObject *parent) : QThread(parent)
5 {
6
7 }
8
9 void MyThread::run()
10 {
11     qDebug() << "当前线程对象的地址：" << QThread::currentThread();
12 }
```

```

13     int num = 0;
14     while(1)
15     {
16         emit curNumber(num++);
17         if(num == 10000000)
18         {
19             break;
20         }
21         QThread::usleep(1);
22     }
23     qDebug() << "run() 执行完毕，子线程退出..." ;
24 }
```

mainwindow.cpp

```

1 #include "mainwindow.h"
2 #include "ui_mainwindow.h"
3 #include "mythread.h"
4 #include <QDebug>
5
6 MainWindow::MainWindow(QWidget *parent) :
7     QMainWindow(parent),
8     ui(new Ui::MainWindow)
9 {
10     ui->setupUi(this);
11
12     qDebug() << "主线程对象地址: " << QThread::currentThread();
13     // 创建子线程
14     MyThread* subThread = new MyThread;
15
16     connect(subThread, &MyThread::curNumber, this, [=](int num)
17     {
18         ui->label->setNum(num);
19     });
20
21     connect(ui->startBtn, &QPushButton::clicked, this, [=]()
22     {
23         // 启动子线程
24         subThread->start();
25     });
26 }
27
28 MainWindow::~MainWindow()
29 {
30     delete ui;
31 }
```

这种在程序中添加子线程的方式是非常简单的，但是也有弊端，假设要在一个子线程中处理多个任务，所有的处理逻辑都需要写到run()函数中，这样该函数中的处理逻辑就会变得非常混乱，不太容易维护。

3. 使用方式 2

3.1 操作步骤

Qt 提供的第二种线程的创建方式弥补了第一种方式的缺点，用起来更加灵活，但是这种方式写起来会相对复杂一些，其具体操作步骤如下：

1. 创建一个新的类，让这个类从 QObject 派生

```
1 class Mywork:public QObject
2 {
3     .....
4 }
```

2. 在这个类中添加一个公共的成员函数，函数体就是我们要子线程中执行的业务逻辑

```
1 class Mywork:public QObject
2 {
3     public:
4     .....
5     // 函数名自己指定，叫什么都可以，参数可以根据实际需求添加
6     void working();
7 }
```

3. 在主线程中创建一个 QThread 对象，这就是子线程的对象

```
1 QThread* sub = new QThread;
```

4. 在主线程中创建工作类对象（千万不要指定给创建的对象指定父对象）

```
1 Mywork* work = new MyWork(this);      // error
2 Mywork* work = new MyWork;             // ok
```

5. 将 MyWork 对象移动到创建的子线程对象中，需要调用 QObject 类提供的 moveToThread() 方法

```
1 // void QObject::moveToThread(QThread *targetThread);
2 // 如果给work指定了父对象，这个函数调用就失败了
3 // 提示： QObject::moveToThread: Cannot move objects with a parent
4 work->moveToThread(sub);      // 移动到子线程中工作
```

6. 启动子线程，调用 start(), 这时候线程启动了，但是移动到线程中的对象并没有工作

7. 调用 MyWork 类对象的工作函数，让这个函数开始执行，这时候是在移动到的那个子线程中运行的

3.2 示例代码

假设函数处理上面在程序中数数的这个需求，具体的处理代码如下：

mywork.h

```
1 ifndef MYWORK_H
2 define MYWORK_H
3
4 #include <QObject>
5
6 class MyWork : public QObject
```

```
7 {  
8     Q_OBJECT  
9 public:  
10    explicit Mywork(QObject *parent = nullptr);  
11  
12    // 工作函数  
13    void working();  
14  
15 signals:  
16    void curNumber(int num);  
17  
18 public slots:  
19 };  
20  
21 #endif // MYWORK_H
```

mywork.cpp

```
1 #include "mywork.h"  
2 #include <QDebug>  
3 #include <QThread>  
4  
5 Mywork::Mywork(QObject *parent) : QObject(parent)  
6 {  
7 }  
8  
10 void Mywork::working()  
11 {  
12     qDebug() << "当前线程对象的地址：" << QThread::currentThread();  
13  
14     int num = 0;  
15     while(1)  
16     {  
17         emit curNumber(num++);  
18         if(num == 10000000)  
19         {  
20             break;  
21         }  
22         QThread::usleep(1);  
23     }  
24     qDebug() << "run() 执行完毕，子线程退出...";  
25 }
```

mainwindow.cpp

```
1 #include "mainwindow.h"  
2 #include "ui_mainwindow.h"  
3 #include <QThread>  
4 #include "mywork.h"  
5 #include <QDebug>  
6  
7 MainWindow::MainWindow(QWidget *parent) :  
8     QMainWindow(parent),  
9     ui(new Ui::MainWindow)  
10 {
```

```

11     ui->setupUi(this);
12
13     qDebug() << "主线程对象的地址: " << QThread::currentThread();
14
15     // 创建线程对象
16     QThread* sub = new QThread;
17     // 创建工作的类对象
18     // 千万不要指定给创建的对象指定父对象
19     // 如果指定了: QObject::moveToThread: Cannot move objects with a parent
20     MyWork* work = new MyWork;
21     // 将工作的类对象移动到创建的子线程对象中
22     work->moveToThread(sub);
23     // 启动线程
24     sub->start();
25     // 让工作的对象开始工作, 点击开始按钮, 开始工作
26     connect(ui->startBtn, &QPushButton::clicked, work, &Mywork::working);
27     // 显示数据
28     connect(work, &Mywork::curNumber, this, [=](int num)
29     {
30         ui->label1->setNum(num);
31     });
32 }
33
34 Mainwindow::~Mainwindow()
35 {
36     delete ui;
37 }

```

使用这种多线程方式，假设有多个不相关的业务流程需要被处理，那么就可以创建多个类似于 MyWork 的类，将业务流程放多类的公共成员函数中，然后将这个业务类的实例对象移动到对应的子线程中 moveToThread() 就可以了，这样可以让编写的程序更加灵活，可读性更强，更易于维护。

十一、Qt中线程池的使用

1.线程池的原理

我们使用线程的时候就去创建一个线程，这样实现起来非常简便，但是就会有一个问题：如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。

那么有没有一种办法使得线程可以复用，就是执行完一个任务，并不被销毁，而是可以继续执行其他的任务呢？

线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。线程池线程都是后台线程。每个线程都使用默认的堆栈大小，以默认的优先级运行，并处于多线程单元中。如果某个线程在托管代码中空闲（如正在等待某个事件），则线程池将插入另一个辅助线程来使所有处理器保持繁忙。如果所有线程池线程都始终保持繁忙，但队列中包含挂起的工作，则线程池将在一段时间后创建另一个辅助线程但线程的数目永远不会超过最大值。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。

在各个编程语言的语种中都有线程池的概念，并且很多语言中直接提供了线程池，作为程序员直接使用就可以了，下面给大家介绍一下线程池的实现原理：

- 线程池的组成主要分为 3 个部分，这三部分配合工作就可以得到一个完整的线程池：

1. 任务队列，存储需要处理的任务，由工作的线程来处理这些任务

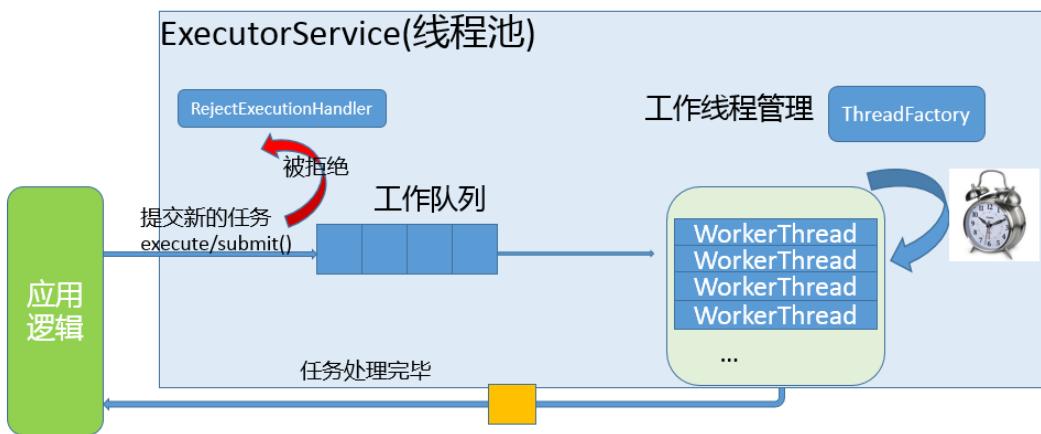
- 通过线程池提供的 API 函数，将一个待处理的任务添加到任务队列，或者从任务队列中删除
- 已处理的任务会被从任务队列中删除
- 线程池的使用者，也就是调用线程池函数往任务队列中添加任务的线程就是生产者线程

2. 工作的线程（任务队列任务的消费者），N 个

- 线程池中维护了一定数量的工作线程，他们的作用是不停的读任务队列，从里边取出任务并处理
- 工作的线程相当于是任务队列的消费者角色，
- 如果任务队列为空，工作的线程将会被阻塞（使用条件变量 / 信号量阻塞）
- 如果阻塞之后有了新的任务，由生产者将阻塞解除，工作线程开始工作

3. 管理者线程（不处理任务队列中的任务），1 个

- 它的任务是周期性的对任务队列中的任务数量以及处于忙状态的工作线程个数进行检测
- 当任务过多的时候，可以适当的创建一些新的工作线程
- 当任务过少的时候，可以适当的销毁一些工作的线程



2. QRunnable

在 Qt 中使用线程池需要先创建任务，添加到线程池中的每一个任务都需要是一个 QRunnable 类型，因此在程序中需要创建子类继承 QRunnable 这个类，然后重写 run() 方法，在这个函数中编写要在线程池中执行的任务，并将这个子类对象传递给线程池，这样任务就可以被线程池中的某个工作的线程处理掉了。

QRunnable 类 常用函数不多，主要是设置任务对象传给线程池后，是否需要自动析构。

```
1 // 在子类中必须要重写的函数，里边是任务的处理流程
2 [pure virtual] void QRunnable::run();
3
4 // 参数设置为 true：这个任务对象在线程池中的线程中处理完毕，这个任务对象就会自动销毁
5 // 参数设置为 false：这个任务对象在线程池中的线程中处理完毕，对象需要程序员手动销毁
6 void QRunnable::setAutoDelete(bool autoDelete);
7 // 获取当前任务对象的析构方式，返回true->自动析构，返回false->手动析构
8 bool QRunnable::autoDelete() const;
```

创建一个要添加到线程池中的任务类，处理方式如下：

```

1 class MyWork : public QObject, public QRunnable
2 {
3     Q_OBJECT
4 public:
5     explicit MyWork(QObject *parent = nullptr)
6     {
7         // 任务执行完毕,该对象自动销毁
8         setAutoDelete(true);
9     }
10    ~MyWork();
11
12    void run() override{}
13 }
```

在上面的示例中 MyWork 类是一个多重继承，如果需要在这个任务中使用 Qt 的信号槽机制进行数据的传递就必须继承 QObject 这个类，如果不使用信号槽传递数据就可以不继承了，只继承 QRunnable 即可。

```

1 class MyWork :public QRunnable
2 {
3     Q_OBJECT
4 public:
5     explicit MyWork()
6     {
7         // 任务执行完毕,该对象自动销毁
8         setAutoDelete(true);
9     }
10    ~MyWork();
11
12    void run() override{}
13 }
```

3. QThreadPool

Qt 中的 QThreadPool 类管理了一组 QThreads, 里边还维护了一个任务队列。QThreadPool 管理和回收各个 QThread 对象，以帮助减少使用线程的程序中的线程创建成本。每个 Qt 应用程序都有一个全局 QThreadPool 对象，可以通过调用 globalInstance() 来访问它。也可以单独创建一个 QThreadPool 对象使用。

线程池常用的 API 函数如下：

```

1 // 获取和设置线程中的最大线程个数
2 int maxThreadCount() const;
3 void setMaxThreadCount(int maxThreadCount);
4
5 // 给线程池添加任务，任务是一个 QRunnable 类型的对象
6 // 如果线程池中没有空闲的线程了，任务会放到任务队列中，等待线程处理
7 void QThreadPool::start(QRunnable * runnable, int priority = 0);
8 // 如果线程池中没有空闲的线程了，直接返回值，任务添加失败，任务不会添加到任务队列中
9 bool QThreadPool::tryStart(QRunnable * runnable);
10
11 // 线程池中被激活的线程的个数(正在工作的线程个数)
12 int QThreadPool::activeThreadCount() const;
```

```
13 // 尝试性的将某一个任务从线程池的任务队列中删除，如果任务已经开始执行就无法删除了
14 bool QThreadPool::tryTake(QRunnable *runnable);
15 // 将线程池中的任务队列里边没有开始处理的所有任务删除，如果已经开始处理了就无法通过该函数
16 // 删除了
17 void QThreadPool::clear();
18
19 // 在每个Qt应用程序中都有一个全局的线程池对象，通过这个函数直接访问这个对象
20 static QThreadPool * QThreadPool::globalInstance();
```

一般情况下，我们不需要在 Qt 程序中创建线程池对象，直接使用 Qt 为每个应用程序提供的线程池全局对象即可。得到线程池对象之后，调用 start() 方法就可以将一个任务添加到线程池中，这个任务就可以被线程池内部的线程池处理掉了，使用线程池比自己创建线程的这种多种多线程方式更加简单和易于维护。

具体的使用方式如下：

mywork.h

```
1 class Mywork :public QRunnable
2 {
3     Q_OBJECT
4 public:
5     explicit Mywork();
6     ~Mywork();
7
8     void run() override;
9 }
```

mywork.cpp

```
1 Mywork::Mywork() : QRunnable()
2 {
3     // 任务执行完毕，该对象自动销毁
4     setAutoDelete(true);
5 }
6 void Mywork::run()
7 {
8     // 业务处理代码
9     .....
10 }
```

mainwindow.cpp

```
1 MainWindow::MainWindow(QWidget *parent) :
2     QMainWindow(parent),
3     ui(new Ui::MainWindow)
4 {
5     ui->setupUi(this);
6
7     // 线程池初始化，设置最大线程池数
8     QThreadPool::globalInstance()->setMaxThreadCount(4);
9     // 添加任务
10    Mywork* task = new Mywork;
11    QThreadPool::globalInstance()->start(task);
12 }
```

十二、基于TCP的Qt网络通信

在标准 C++ 没有提供专门用于套接字通信的类，所以只能使用操作系统提供的基于 C 的 API 函数，基于这些 C 的 API 函数我们也可以封装自己的 C++ 类 C++ 套接字类的封装。但是 Qt 就不一样了，它是 C++ 的一个框架并且里边提供了用于套接字通信的类（TCP、UDP）这样就使得我们的操作变得更加简单了（当然，在Qt中使用标准C的API进行套接字通信也是完全没有问题的）。下面，给大家讲一下如果使用相关类的进行 TCP 通信。

使用 Qt 提供的类进行基于 TCP 的套接字通信需要用到两个类：

- QTcpServer：服务器类，用于监听客户端连接以及和客户端建立连接。
- QTcpSocket：通信的套接字类，客户端、服务器端都需要使用。

这两个套接字通信类都属于网络模块 network。

1. QTcpServer

QTcpServer 类用于监听客户端连接以及和客户端建立连接，在使用之前先介绍一下这个类提供的一些常用 API 函数：

1.1 公共成员函数

构造函数

```
1 | QTcpServer::QTcpServer(QObject *parent = Q_NULLPTR);
```

给监听的套接字设置监听

```
1 | bool QTcpServer::listen(const QHostAddress &address = QHostAddress::Any,  
2 |     quint16 port = 0);  
3 | // 判断当前对象是否在监听，是返回true，没有监听返回false  
4 | bool QTcpServer::isListening() const;  
5 | // 如果当前对象正在监听返回监听的服务器地址信息，否则返回 QHostAddress::Null  
6 | QHostAddress QTcpServer::serverAddress() const;  
7 | // 如果服务器正在侦听连接，则返回服务器的端口；否则返回0  
8 | quint16 QTcpServer::serverPort() const
```

参数：

address：通过类 QHostAddress 可以封装 IPv4、IPv6 格式的 IP 地址，QHostAddress::Any 表示自动绑定

port：如果指定为 0 表示随机绑定一个可用端口。

返回值：绑定成功返回 true，失败返回 false

得到和客户端建立连接之后用于通信的 QTcpSocket 套接字对象，它是 QTcpServer 的一个子对象，当 QTcpServer 对象析构的时候会自动析构这个子对象，当然也可自己手动析构，建议用完之后自己手动析构这个通信的 QTcpSocket 对象。

```
1 | QTcpSocket *QTcpServer::nextPendingConnection();
```

阻塞等待客户端发起的连接请求，不推荐在单线程程序中使用，建议使用非阻塞方式处理新连接，即使用信号 newConnection()。

```
1 | bool QTcpServer::waitForNewConnection(int msec = 0, bool *timedOut =
Q_NULLPTR);
```

参数:

msec: 指定阻塞的最大时长, 单位为毫秒 (ms)

timeout: 传出参数, 如果操作超时 timeout 为 true, 没有超时 timeout 为 false

1.2 信号

当接受新连接导致错误时, 将发射如下信号。socketError 参数描述了发生的错误相关的信息。

```
1 | [signal] void QTcpServer::acceptError(QAbstractSocket::SocketError
socketError);
```

每次有新连接可用时都会发出 newConnection () 信号

```
1 | [signal] void QTcpServer::newConnection();
```

2. QTcpSocket

QTcpSocket 是一个套接字通信类, 不管是客户端还是服务器端都需要使用。在 Qt 中发送和接收数据也属于 IO 操作 (网络 IO), 先来看一下这个类的继承关系:



2.1 公共成员函数

构造函数

```
1 | QTcpSocket::QTcpSocket(QObject *parent = Q_NULLPTR);
```

连接服务器, 需要指定服务器端绑定的IP和端口信息。

```
1 | [virtual] void QAbstractSocket::connectToHost(const QString &hostName,
quint16 port, OpenMode openMode = ReadWrite, NetworkLayerProtocol protocol =
AnyIPProtocol);
2 |
3 | [virtual] void QAbstractSocket::connectToHost(const QHostAddress &address,
quint16 port, OpenMode openMode = ReadWrite);
```

在 Qt 中不管调用读操作函数接收数据, 还是调用写函数发送数据, 操作的对象都是本地的由 Qt 框架维护的一块内存。因此, 调用了发送函数数据不一定会马上被发送到网络中, 调用了接收函数也不是直接从网络中接收数据, 关于底层的相关操作是不需要使用者来维护的。

接收数据

```
1 // 指定可接收的最大字节数 maxSize 的数据到指针 data 指向的内存中
2 qint64 QIODevice::read(char *data, qint64 maxSize);
3 // 指定可接收的最大字节数 maxSize, 返回接收的字符串
4 QByteArray QIODevice::read(qint64 maxSize);
5 // 将当前可用操作数据全部读出, 通过返回值返回读出的字符串
6 QByteArray QIODevice::readAll();
```

发送数据

```
1 // 发送指针 data 指向的内存中的 maxSize 个字节的数据
2 qint64 QIODevice::write(const char *data, qint64 maxSize);
3 // 发送指针 data 指向的内存中的数据, 字符串以 \0 作为结束标记
4 qint64 QIODevice::write(const char *data);
5 // 发送参数指定的字符串
6 qint64 QIODevice::write(const QByteArray &byteArray);
```

2.2 信号

在使用 QTcpSocket 进行套接字通信的过程中, 如果该类对象发射出 readyRead() 信号, 说明对端发送的数据达到了, 之后就可以调用 read 函数接收数据了。

```
1 [signal] void QIODevice::readyRead();
```

调用 connectToHost() 函数并成功建立连接之后发出 connected() 信号。

```
1 [signal] void QAbstractSocket::connected();
```

在套接字断开连接时发出 disconnected() 信号。

```
1 [signal] void QAbstractSocket::disconnected();
```

3. 通信流程

使用 Qt 提供的类进行套接字通信比使用标准 C API 进行网络通信要简单 (因为在内部进行了封装) 原始的 TCP 通信流程 Qt 中的套接字通信流程如下:

3.1 服务器端

3.1.1 通信流程

1. 创建套接字服务器 QTcpServer 对象
2. 通过 QTcpServer 对象设置监听, 即: QTcpServer::listen()
3. 基于 QTcpServer::newConnection() 信号检测是否有新的客户端连接
4. 如果有新的客户端连接调用 QTcpSocket *QTcpServer::nextPendingConnection() 得到通信的套接字对象
5. 使用通信的套接字对象 QTcpSocket 和客户端进行通信

3.1.2 代码片段

服务器端的窗口界面如下图所示:



头文件

```
1 class MainWindow : public QMainWindow
2 {
3     Q_OBJECT
4
5 public:
6     explicit MainWindow(QWidget *parent = 0);
7     ~MainWindow();
8
9 private slots:
10    void on_startServer_clicked();
11
12    void on_sendMsg_clicked();
13
14 private:
15    Ui::MainWindow *ui;
16    QTcpServer *m_server;
17    QTcpSocket *m_tcp;
18};
```

源文件

```
1 MainWindow::MainWindow(QWidget *parent) :
2     QMainWindow(parent),
3     ui(new Ui::MainWindow)
4 {
5     ui->setupUi(this);
```

```

6     setWindowTitle("TCP - 服务器");
7     // 创建 QTcpServer 对象
8     m_server = new QTcpServer(this);
9     // 检测是否有新的客户端连接
10    connect(m_server, &QTcpServer::newConnection, this, [=]()
11    {
12        m_tcp = m_server->nextPendingConnection();
13        ui->record->append("成功和客户端建立了新的连接...");
14        m_status->setPixmap(QPixmap(":/connect.png").scaled(20, 20));
15        // 检测是否有客户端数据
16        connect(m_tcp, &QTcpSocket::readyRead, this, [=]()
17        {
18            // 接收数据
19            QString recvMsg = m_tcp->readAll();
20            ui->record->append("客户端Say: " + recvMsg);
21        });
22        // 客户端断开了连接
23        connect(m_tcp, &QTcpSocket::disconnected, this, [=]()
24        {
25            ui->record->append("客户端已经断开了连接...");
26            m_tcp->deleteLater();
27            m_status->setPixmap(QPixmap(":/disconnect.png").scaled(20,
28));
28        });
29    });
30 }
31
32 MainWindow::~MainWindow()
33 {
34     delete ui;
35 }
36
37 // 启动服务器端的服务按钮
38 void MainWindow::on_startServer_clicked()
39 {
40     unsigned short port = ui->port->text().toInt();
41     // 设置服务器监听
42     m_server->listen(QHostAddress::Any, port);
43     ui->startServer->setEnabled(false);
44 }
45
46 // 点击发送数据按钮
47 void MainWindow::on_sendMsg_clicked()
48 {
49     QString sendMsg = ui->msg->toPlainText();
50     m_tcp->write(sendMsg.toUtf8());
51     ui->record->append("服务器Say: " + sendMsg);
52     ui->msg->clear();
53 }

```

3.2 客户端

3.2.1 通信流程

1. 创建通信的套接字类 QTcpSocket 对象
2. 使用服务器端绑定的 IP 和端口连接服务器 QAbstractSocket::connectToHost()
3. 使用 QTcpSocket 对象和服务器进行通信

3.2.2 代码片段

客户端的窗口界面如下图所示：



头文件

```
1 class MainWindow : public QMainWindow
2 {
3     Q_OBJECT
4
5 public:
6     explicit MainWindow(QWidget *parent = 0);
7     ~MainWindow();
8
9 private slots:
10    void on_connectServer_clicked();
11
12    void on_sendMsg_clicked();
13
14    void on_disconnect_clicked();
15
16 private:
17    Ui::MainWindow *ui;
18    QTcpSocket* m_tcp;
19};
```

源文件

```
1 MainWindow::MainWindow(QWidget *parent) :
2     QMainWindow(parent),
3     ui(new Ui::MainWindow)
4 {
5     ui->setupUi(this);
6     setWindowTitle("TCP - 客户端");
7
8     // 创建通信的套接字对象
9     m_tcp = new QTcpSocket(this);
10    // 检测服务器是否回复了数据
11    connect(m_tcp, &QTcpSocket::readyRead, [=]()
12    {
13        // 接收服务器发送的数据
14        QByteArray recvMsg = m_tcp->readAll();
15        ui->record->append("服务器Say: " + recvMsg);
16    });
17
18    // 检测是否和服务器是否连接成功了
19    connect(m_tcp, &QTcpSocket::connected, this, [=]()
20    {
21        ui->record->append("恭喜，连接服务器成功!!!!");
22        m_status->setPixmap(QPixmap(":/connect.png").scaled(20, 20));
23    });
24
25    // 检测服务器是否和客户端断开了连接
26    connect(m_tcp, &QTcpSocket::disconnected, this, [=]()
27    {
28        ui->record->append("服务器已经断开了连接， . . .");
29        ui->connectServer->setEnabled(true);
30        ui->disconnect->setEnabled(false);
31    });
32 }
33
34 MainWindow::~MainWindow()
35 {
36     delete ui;
37 }
38
39 // 连接服务器按钮按下之后的处理动作
40 void MainWindow::on_connectServer_clicked()
41 {
42     QString ip = ui->ip->text();
43     unsigned short port = ui->port->text().toInt();
44     // 连接服务器
45     m_tcp->connectToHost(QHostAddress(ip), port);
46     ui->connectServer->setEnabled(false);
47     ui->disconnect->setEnabled(true);
48 }
49
50 // 发送数据按钮按下之后的处理动作
51 void MainWindow::on_sendMsg_clicked()
52 {
53     QString sendMsg = ui->msg->toPlainText();
54     m_tcp->write(sendMsg.toUtf8());
55     ui->record->append("客户端Say: " + sendMsg);
56     ui->msg->clear();
57 }
58 }
```

```
59 // 断开连接按钮被按下之后的处理动作
60 void MainWindow::on_disconnect_clicked()
61 {
62     m_tcp->close();
63     ui->connectServer->setEnabled(true);
64     ui->disconnect->setEnabled(false);
65 }
```

十二、Qt事件之事件处理器

1. 事件

众所周知 Qt 是一个基于 C++ 的框架，主要用来开发带窗口的应用程序（不带窗口的也行，但不是主流）。我们使用的基于窗口的应用程序都是基于事件，其目的主要是用来实现回调（因为只有这样程序的效率才是最高的）。所以在 Qt 框架内部为我们提供了一些列的事件处理机制，当窗口事件产生之后，事件会经过：事件派发 -> 事件过滤->事件分发->事件处理几个阶段。Qt 窗口中对于产生的一系列事件都有默认的处理动作，如果我们有特殊需求就需要在合适的阶段重写事件的处理动作。

事件 (event) 是由系统或者 Qt 本身在不同的场景下发出的。当用户按下 / 移动鼠标、敲下键盘，或者是窗口关闭 / 大小发生变化 / 隐藏或显示都会发出一个相应的事件。一些事件在对用户操作做出响应时发出，如鼠标 / 键盘事件等；另一些事件则是由系统自动发出，如计时器事件。

每一个 Qt 应用程序都对应一个唯一的 QApplication 应用程序对象，然后调用这个对象的 exec() 函数，这样 Qt 框架内部的事件检测就开始了（程序将进入事件循环来监听应用程序的事件）。

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4     MainWindow* w = new MainWindow();
5     w.show();
6     return a.exec();
7 }
```

事件在 Qt 中产生之后，的分发过程是这样的：

1. 当事件产生之后，Qt 使用用应用程序对象调用 notify() 函数将事件发送到指定的窗口：

```
1 [override virtual] bool QApplication::notify(QObject *receiver, QEvent
*e);
```

2. 事件在发送过程中可以通过事件过滤器进行过滤，默认不对任何产生的事件进行过滤。

```
1 // 需要先给窗口安装过滤器，该事件才会触发
2 [virtual] bool QObject::eventFilter(QObject *watched, QEvent *event)
```

3. 当事件发送到指定窗口之后，窗口的事件分发器会对收到的事件进行分类：

```
1 [override virtual protected] bool QWidget::event(QEvent *event);
```

4. 事件分发器会将分类之后的事件（鼠标事件、键盘事件、绘图事件。。。）分发给对应的事件处理器函数进行处理，每个事件处理器函数都有默认的处理动作（我们也可以重写这些事件处理器函数），比如：鼠标事件：

```
1 // 鼠标按下
2 [virtual protected] void QWidget::mousePressEvent(QMouseEvent *event);
3 // 鼠标释放
4 [virtual protected] void QWidget::mouseReleaseEvent(QMouseEvent *event);
5 // 鼠标移动
6 [virtual protected] void QWidget::mouseMoveEvent(QMouseEvent *event);
```

2. 事件处理器函数

通过上面的描述可以得知：Qt 的事件处理器函数处于食物链的最末端，每个事件处理器函数都对应一个唯一的事件，这为我们重新定义事件的处理动作提供了便利。另外，Qt 提供的这些事件处理器函数都是回调函数，也就是说作为使用者我们只需要指定函数的处理动作，关于函数的调用是不需要操心的，当某个事件被触发，Qt 框架会调用对应的事件处理器函数。

QWidget 类是 Qt 中所有窗口类的基类，在这个类里边定义了很多事件处理器函数，它们都是受保护的虚函数。我们可以在 Qt 的任意一个窗口类中重写这些虚函数来重定义它们的行为。下面介绍一些常用的事件处理器函数：

2.1 鼠标事件

- 鼠标按下事件

当鼠标左键、鼠标右键、鼠标中键被按下，该函数被自动调用，通过参数可以得到当前按下的是哪个鼠标键

```
1 [virtual protected] void QWidget::mousePressEvent(QMouseEvent *event);
```

- 鼠标释放事件

当鼠标左键、鼠标右键、鼠标中键被释放，该函数被自动调用，通过参数可以得到当前释放的是哪个鼠标键

```
1 [virtual protected] void QWidget::mouseReleaseEvent(QMouseEvent *event);
```

- 鼠标移动事件

当鼠标移动（也可以按住一个或多个鼠标键移动），该函数被自动调用，通过参数可以得到在移动过程中哪些鼠标键被按下了。

```
1 [virtual protected] void QWidget::mouseMoveEvent(QMouseEvent *event);
```

- 鼠标双击事件

当鼠标双击该函数被调用，通过参数可以得到是通过哪个鼠标键进行了双击操作。

```
1 [virtual protected] void QWidget::mouseDoubleClickEvent(QMouseEvent *event);
```

- 鼠标进入事件

当鼠标进入窗口的一瞬间，触发该事件，注意：只在进入的瞬间触发一次该事件

```
1 | [virtual protected] void QWidget::enterEvent(QEvent *event);
```

- 鼠标离开事件

当鼠标离开窗口的一瞬间，触发该事件，注意：只在离开的瞬间触发一次该事件

```
1 | [virtual protected] void QWidget::leaveEvent(QEvent *event);
```

2.2 键盘事件

- 键盘按下事件

当键盘上的按键被按下了，该函数被自动调用，通过参数可以得知按下的是哪个键。

```
1 | [virtual protected] void QWidget::keyPressEvent(QKeyEvent *event);
```

- 键盘释放事件

当键盘上的按键被释放了，该函数被自动调用，通过参数可以得知释放的是哪个键。

```
1 | [virtual protected] void QWidget::keyReleaseEvent(QKeyEvent *event);
```

2.3 窗口重绘事件

当窗口需要刷新的时候，该函数就会自动被调用。窗口需要刷新的情景很多，比如：窗口大小发生变化，窗口显示等，另外我们还可以通过该函数给窗口绘制背景图，总之这是一个需要经常被重写的一个事件处理器函数。

```
1 | [virtual protected] void QWidget::paintEvent(QPaintEvent *event);
```

2.4 窗口关闭事件

当窗口标题栏的关闭按钮被按下并且在窗口关闭之前该函数被调用，可以通过该函数控制窗口是否被关闭。

```
1 | [virtual protected] void QWidget::closeEvent(QCloseEvent *event);
```

2.5 重置窗口大小事件

当窗口的大小发生变化，该函数被调用。

```
1 | [virtual protected] void QWidget::resizeEvent(QResizeEvent *event);
```

除此之外，关于 Qt 窗口提供的其他事件处理器函数还有很多，感兴趣的话可以仔细阅读 Qt 的帮助文档，窗口的事件处理器函数非常好找，规律是这样的：

1. 受保护的虚函数
2. 函数名分为两部分：事件描述+Event
3. 函数带一个事件类型的参数

3. 重写事件处理器函数

由于事件处理器函数都是虚函数，因此我们就可以添加一个标准窗口类的派生类，这样不仅使子类继承了父类的属性，还可以在这个子类中重写父类的虚函数，总起来说整个操作过程还是 so easy 的。

下面

举个栗子



1. 创建一个Qt项目，添加一个窗口类（让其从某个标准窗口类派生）
2. 在子类中重写从父类继承的虚函数（也就是事件处理器函数）

3.1 头文件

```
1 #include < QMainWindow>
2
3 QT_BEGIN_NAMESPACE
4 namespace Ui { class MainWindow; }
5 QT_END_NAMESPACE
6
7 class MainWindow : public QMainWindow
8 {
9     Q_OBJECT
10
11 public:
12     MainWindow(QWidget *parent = nullptr);
13     ~MainWindow();
14
15 protected:
16     // 重写事件处理器函数
17     void closeEvent(QCloseEvent* ev);
18     void resizeEvent(QResizeEvent* ev);
19
20 private:
21     Ui::MainWindow *ui;
22 };
```

3.2 源文件

```
1 #include "mainwindow.h"
2 #include "ui_mainwindow.h"
3 #include <QCloseEvent>
4 #include < QMessageBox>
5 #include <QResizeEvent>
6 #include <QDebug>
7
8 MainWindow::MainWindow(QWidget *parent)
9     : QMainWindow(parent)
10    , ui(new Ui::MainWindow)
11 {
12     ui->setupUi(this);
```

```

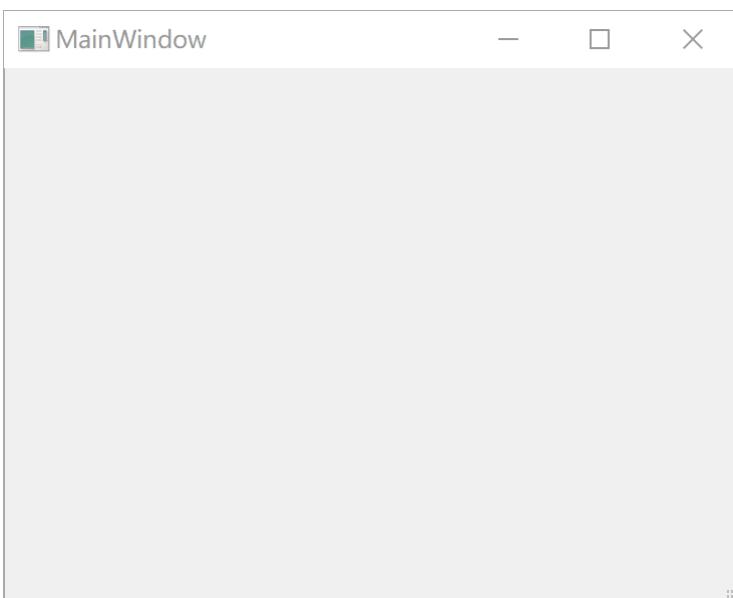
13 }
14
15 MainWindow::~MainWindow()
16 {
17     delete ui;
18 }
19
20 void MainWindow::closeEvent(QCloseEvent *ev)
21 {
22     QMessageBox::Button btn = QMessageBox::question(this, "关闭窗口", "您确定
要关闭窗口吗？");
23     if(btn == QMessageBox::Yes)
24     {
25         // 接收并处理这个事件
26         ev->accept();
27     }
28     else
29     {
30         // 忽略这个事件
31         ev->ignore();
32     }
33 }
34
35 void MainWindow::resizeEvent(QResizeEvent *ev)
36 {
37     qDebug() << "oldSize: " << ev->oldSize()
38             << "currentSize: " << ev->size();
39 }

```

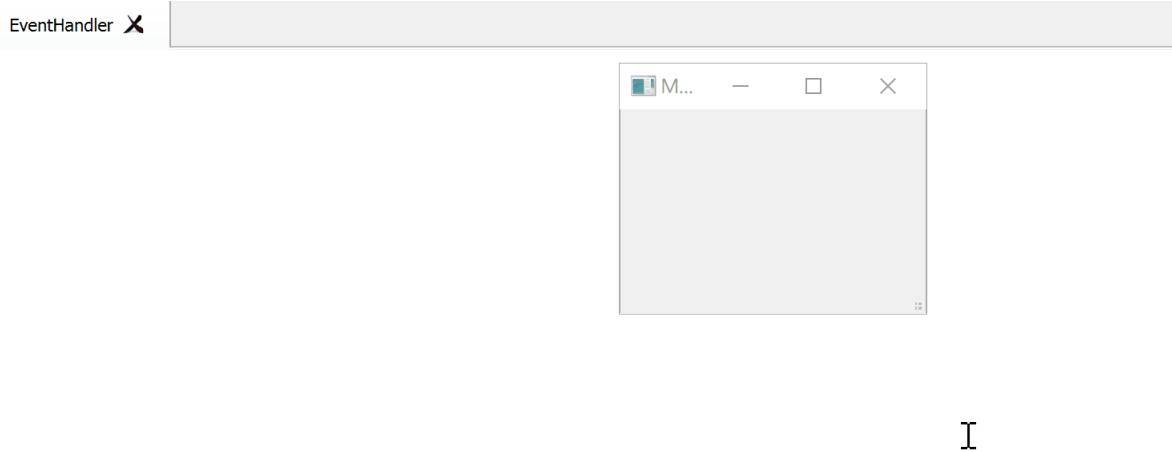
QCloseEvent 类是 QEvent 类的子类，程序中使用的 accept() 或者 ignore() 的作用请参考 QEvent 类

3.3 效果

在上面重写的 closeEvent 事件中添加了关闭窗口的判断，这样就可以避免误操作导致窗口被关闭了，效果如下：



如果想要时时检测窗口大小，就可以重写窗口的 resizeEvent 事件，这样就可以得到窗口的最新尺寸信息了：



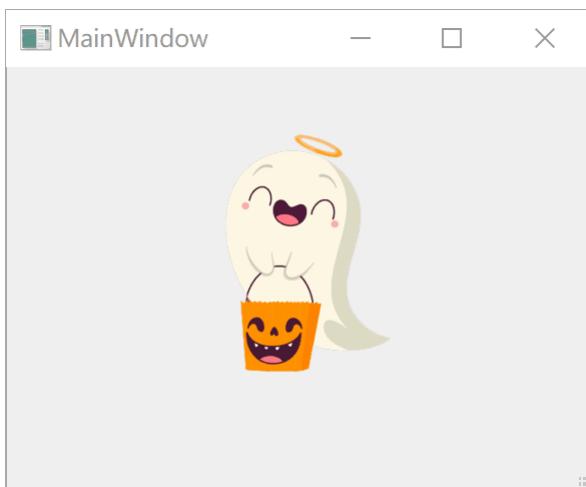
I

4.自定义按钮

基于 Qt 提供的事件处理器函数，我们可以非常轻松地按照自己的想法制作出一个按钮，按钮的要求如下：

1. 从视觉上看是一个不规则按钮（按钮实际上都是矩形的）
2. 按钮上需要显示指定的背景图片
3. 按钮在鼠标的不同操作阶段（无操作、鼠标悬停、鼠标按下）能够显示不同的背景图

按钮效果如下：



4.1 添加子类

新添加的按钮类可以让它继承 QPushButton，也可以让它继承其他的窗口类（代价是当鼠标点击事件触发之后需要自己发射自定义信号），这里让添加的子类从 QWidget 类派生。

自定义类头文件

```
1 #ifndef MYBUTTON_H
2 #define MYBUTTON_H
3
4 #include <QWidget>
```

```

5  class MyButton : public QWidget
6  {
7      Q_OBJECT
8  public:
9      explicit MyButton(QWidget *parent = nullptr);
10
11     void setImage(QString normal, QString hover, QString pressed);
12
13 protected:
14     void mousePressEvent(QMouseEvent* ev);
15     void mouseReleaseEvent(QMouseEvent* ev);
16     void enterEvent(QEvent* ev);
17     void leaveEvent(QEvent* ev);
18     void paintEvent(QPaintEvent* ev);
19
20 signals:
21     void clicked();
22
23 private:
24     QPixmap m_normal;
25     QPixmap m_press;
26     QPixmap m_hover;
27     QPixmap m_current;
28 };
29
30
31 #endif // MYBUTTON_H

```

自定义类源文件

```

1 #include "mybutton.h"
2
3 #include <QPainter>
4
5 MyButton::MyButton(QWidget *parent) : QWidget(parent)
6 {
7
8 }
9
10 void MyButton::setImage(QString normal, QString hover, QString pressed)
11 {
12     // 加载图片
13     m_normal.load(normal);
14     m_hover.load(hover);
15     m_press.load(pressed);
16     m_current = m_normal;
17     // 设置按钮和图片大小一致
18     setFixedSize(m_normal.size());
19 }
20
21 void MyButton::mousePressEvent(QMouseEvent *ev)
22 {
23     // 鼠标被按下，发射这个自定义信号
24     emit clicked();
25     m_current = m_press;
26     update();
27 }

```

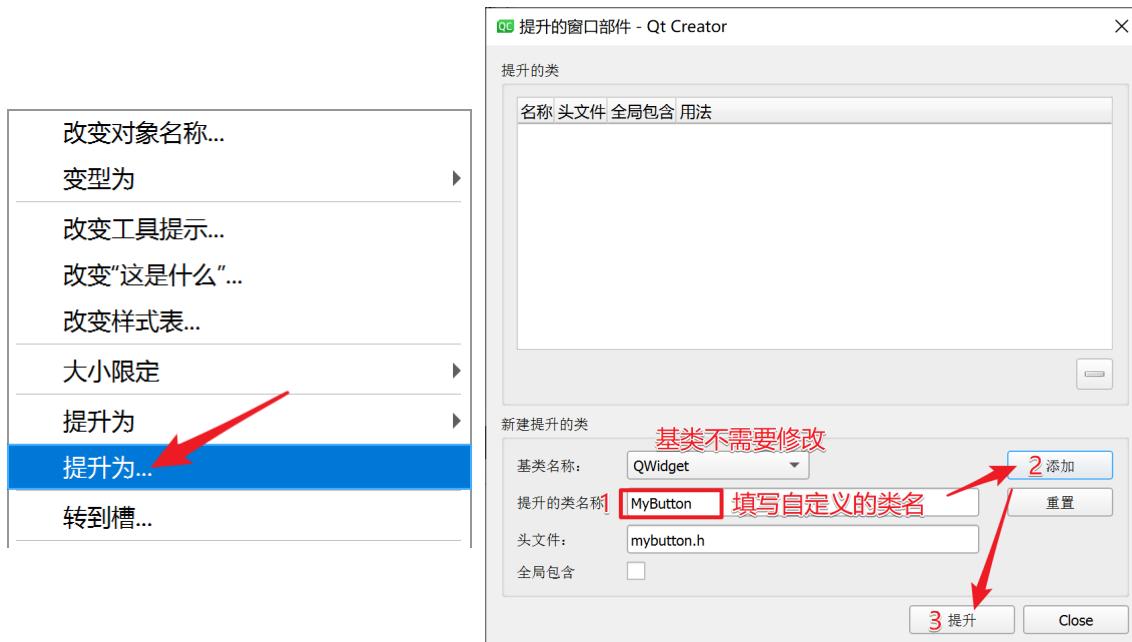
```

28 void MyButton::mouseReleaseEvent(QMouseEvent *ev)
29 {
30     m_current = m_normal;
31     update();
32 }
33
34 void MyButton::enterEvent(QEvent *ev)
35 {
36     m_current = m_hover;
37     update();
38 }
39
40 void MyButton::leaveEvent(QEvent *ev)
41 {
42     m_current = m_normal;
43     update();
44 }
45
46 void MyButton::paintEvent(QPaintEvent *ev)
47 {
48     QPainter p(this);
49     p.drawPixmap(rect(), m_current);
50 }
51

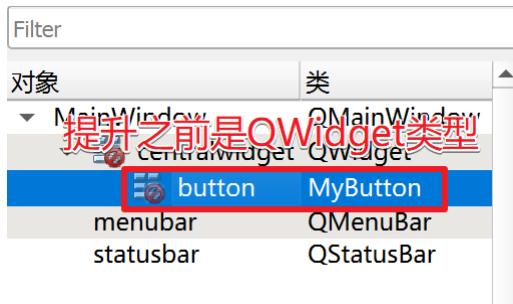
```

4.2 使用自定控件

由于 Qt 的 UI 工具箱中提供的都是标准控件，自定义的控件是不能直接拖拽到 UI 窗口中的，这时我们需要先看一下自定义控件的基类类型：上面自定义的 MyButton 的基类是 QWidget 类型，因此需要往窗口中拖拽一个 QWidget 类型的标准控件，然后在这个标准控件上鼠标右键：



这样添加的控件类型就变成了自定义的子类类型：



4.3 设置图片

在主窗口中通过添加的按钮的对象，调用子类的成员函数给其添加图片：

mainwindow.cpp

```
1 MainWindow::MainWindow(QWidget *parent)
2     : QMainWindow(parent)
3     , ui(new Ui::MainWindow)
4 {
5     ui->setupUi(this);
6
7     // 给自定义按钮设置图片
8     ui->button->setImage(":/ghost-1.png",(":/ghost-2.png",(":/ghost-
3.png"));
9     // 处理自定义按钮的鼠标点击事件
10    connect(ui->button, &MyButton::clicked, this, [=]()
11    {
12        QMessageBox::information(this, "按钮", "莫要调戏我...");  

13    });
14 }
15
16 MainWindow::~MainWindow()
17 {
18     delete ui;
19 }
```

十三、Qt事件之事件分发器

1. QEvent

当事件产生被发送到对应的窗口之后，窗口并不会直接处理这个事件，而是对这些事件进行细分，然后根据事件的类型再次进行分发（相当于公司接了个项目，对项目进行划分之后分发给各个职能部门，由各个部门进行模块的开发），对应的事件处理器函数得到这个分发的事件之后就开始处理这个事件。

关于窗口事件的分发，对应一个事件分发器，叫做 event

```
1 [override virtual protected] bool QWidget::event(QEvent *event);
```

通过事件分发器的函数原型可以得知，关于事件类型的判断是基于参数完成的，这个参数是一个 QEvent 类型的对象，下面来看一下这个类中常用的一些 API 函数：

```
1 void QEvent::accept();
```

- 该函数的作用是让窗口接受传递过来的事件，事件不会向上层窗口（父窗口）传递。

```
1 | void QEvent::ignore();
```

- 该函数的作用是让窗口忽略传递过来的事件，事件被传递给父窗口（向上传递）。

```
1 | bool QEvent::isAccepted() const;
2 | void QEvent::setAccepted(bool accepted);
```

- 设置传递过来的事件是被接受还是被忽略
 - setAccepted(true) == accept()
 - setAccepted(false) == ignore()
- 得到传递的窗口的事件的类型，返回值是一个枚举类型，内容很多可以自己查帮助文档，简单的贴个图：

enum QEvent::Type

This enum type defines the valid event types in Qt. The event types and the specialized classes for each type are as follows:

Constant 事件的类型	Value	Description	管理这个事件的子事件类
QEvent::None	0	Not an event.	
QEvent::ActionAdded	111	A new action has been added (QActionEvent).	
QEvent::ActionChanged	113	An action has been changed (QActionEvent).	
QEvent::ActionRemoved	115	An action has been removed (QActionEvent).	
QEvent::ActivationChange	99	A widget's top-level window activation state has changed.	
QEvent::ApplicationActivate	121	This enum has been deprecated. Use QApplication::stateChange instead.	
QEvent::ApplicationActivated	ApplicationActivate	This enum has been deprecated. Use QApplication::stateChange instead.	
QEvent::ApplicationDeactivate	122	This enum has been deprecated. Use QApplication::stateChange instead.	
QEvent::ApplicationFontChange	36	The default application font has changed.	
QEvent::ApplicationLayoutDirectionChange	37	The default application layout direction has changed.	
QEvent::ApplicationPaletteChange	38	The default application palette has changed.	
QEvent::ApplicationStateChange	214	The state of the application has changed.	
QEvent::ApplicationWindowIconChange	35	The application's icon has changed.	
QEvent::ChildAdded	68	An object gets a child (QChildEvent).	
QEvent::ChildPolished	69	A widget child gets polished (QChildEvent).	
QEvent::ChildRemoved	71	An object loses a child (QChildEvent).	
QEvent::Clipboard	40	The clipboard contents have changed.	

2. 事件分发器

在不需要人为干预的情况下，事件分发器会自主的完成相关事件的分发，下面来还原一下事件分发器的分发流程，以下是这个函数的部分源码展示：

```
1 | bool QWidget::event(QEvent *ev)
2 |
3 | {
4 |     switch(ev->type())
5 |     {
6 |         // 鼠标移动
7 |         case QEvent::MouseMove:
8 |             mouseMoveEvent((MouseEvent*)event);
9 |             break;
10 |            // 鼠标按下
11 |            case QEvent::MouseButtonDown:
12 |                mousePressEvent((MouseEvent*)event);
13 |                break;
```

```

13 // 鼠标释放
14 case QEvent::MouseButtonRelease:
15     mouseReleaseEvent((QMouseEvent*)event);
16     break;
17 // 鼠标双击
18 case QEvent::MouseButtonDoubleClick:
19     mouseDoubleClickEvent((QMouseEvent*)event);
20     break;
21 // 键盘按键被按下事件
22 case QEvent::KeyPress:
23     break;
24 ...
25 ...
26 ...
27 default:
28     break;
29 }
30 }
```

可以直观的看到事件分发器在对事件进行判定之后会调用相关的事件处理器函数，这样事件就被最终处理掉了。

如果我们不想让某些触发的事件进入到当前窗口中，可以在事件分发器中进行拦截，拦截之前先来了解一下事件分发器函数的返回值：

1. 如果传入的事件已被识别并且处理，则需要返回 true，否则返回 false。如果返回值是 true，那么 Qt 会认为这个事件已经处理完毕，不会再将这个事件发送给其它对象，而是会继续处理事件队列中的下一事件。
2. 在event()函数中，调用事件对象的 accept() 和 ignore() 函数是没有作用的，不会影响到事件的传播。

也就是说如果想过滤某个事件，只需要在判断出这个事件之后直接返回 true 就可以了。

下面来举个例子，在窗口中过滤掉鼠标按下的事件：

```

1 bool MainWindow::event(QEvent *ev)
2 {
3     if(ev->type() == QEvent::MouseButtonPress ||
4         ev->type() == QEvent::MouseButtonDoubleClick)
5     {
6         // 过滤掉鼠标按下的事件
7         return true;
8     }
9     return QWidget::event(ev);
10 }
```

这样窗口就再也收不到鼠标的单击和双击事件了，对于这两个事件以外的其他事件是没有任何影响的，因为在重写的事件分发器函数的最后调用了父类的事件分发器函数

```
1 | return QWidget::event(ev);
```

这样就能保证其他事件按照默认的分发流程进行分发，并最终被窗口处理掉。

十四、Qt事件之事件过滤器

1. 事件过滤器

除了使用事件分发器来过滤 Qt 窗口中产生的事件，还可以通过事件过滤器过滤相关的事件。当 Qt 的事件通过应用程序对象发送给相关窗口之后，窗口接收到数据之前这个期间可对事件进行过滤，过滤掉的事件就不能被继续处理了。QObject 有一个 eventFilter() 函数，用于建立事件过滤器。函数原型如下：

```
1 | [virtual] bool QObject::eventFilter(QObject *watched, QEvent *event);
```

参数：

- ◆ watched：要过滤的事件的所有者对象
- ◆ event：要过滤的具体的事件

返回值：如果想过滤掉这个事件，停止它被进一步处理，返回 true，否则返回 false

既然要过滤传递中的事件，首当其冲还是要搞明白如何通过事件过滤器进行事件的过滤，主要分为两步：

1. 给要被过滤事件的类对象安装事件过滤器

```
1 | void QObject::installEventFilter(QObject *filterObj);
```

假设调用 installEventFilter() 函数的对象为当前对象，那么就可以基于参数指定的 filterObj 对象来过滤当前对象中的指定的事件了。

2. 在要进行事件过滤的类中（filterObj 参数对应的类）重写从QObject类继承的虚函数 eventFilter()。

2. 事件过滤器的使用

根据上面的使用步骤，举一个例子：

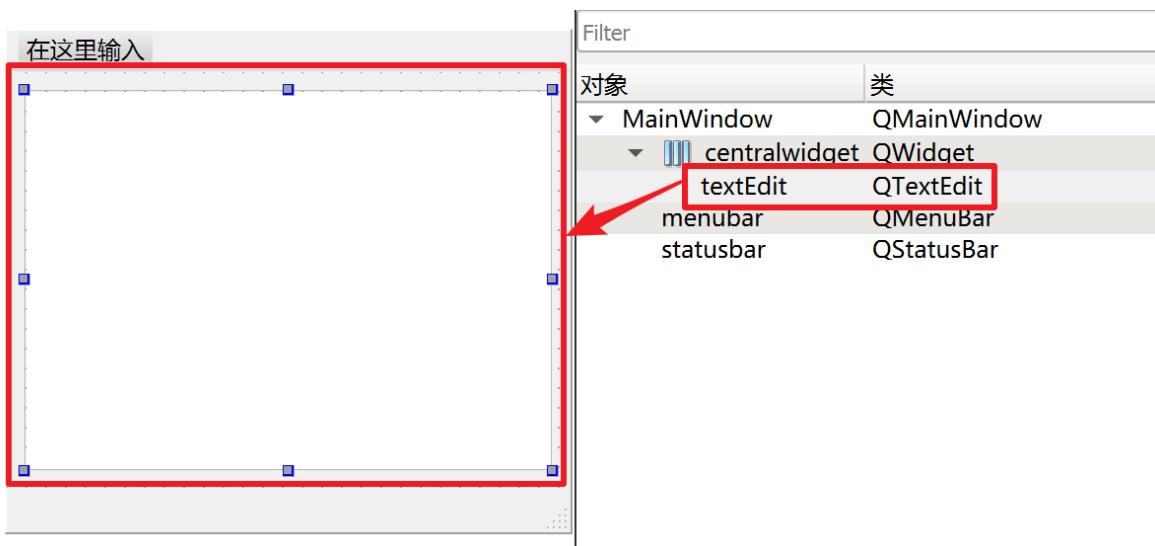
在一个窗口中有一个多行文本输入框 QTextEdit，需要让我们屏蔽掉键盘上的回车键，也就是按回车键之后在这个文本编辑框中再也不能换行了。

其实上面的需求有三种解决方案：

1. 自定义一个新的类让其继承 QTextEdit，在这个子类中重写键盘事件 keyPressEvent，在这个函数里边屏蔽掉回车键
2. 自定义一个新的类让其继承 QTextEdit，在这个子类中重写事件分发器 event，在这个函数里边屏蔽掉回车键
3. 给 QTextEdit 安装事件过滤器，基于 QTextEdit 的父窗口对这个控件的事件进行过滤

最简单的方式还是第三种，因为我们不需要再定义出一个子类就可以轻松的完成控件事件的过滤了。

准备工作：在主窗口中添加一个 QTextEdit 类型的控件，如下图：



主窗口头文件：mainwindow.h

```
1 QT_BEGIN_NAMESPACE
2 namespace Ui { class MainWindow; }
3 QT_END_NAMESPACE
4
5 class MainWindow : public QMainWindow
6 {
7     Q_OBJECT
8
9 public:
10     MainWindow(QWidget *parent = nullptr);
11     ~MainWindow();
12
13     bool eventFilter(QObject *watched, QEvent *event);
14
15 private:
16     Ui::MainWindow *ui;
17 };
```

主窗口源文件：mainwindow.cpp

```
1 MainWindow::MainWindow(QWidget *parent)
2     : QMainWindow(parent)
3     , ui(new Ui::MainWindow)
4 {
5     ui->setupUi(this);
6
7     ui->textEdit->installEventFilter(this);
8 }
9
10 MainWindow::~MainWindow()
11 {
12     delete ui;
13 }
14
15 bool MainWindow::eventFilter(QObject *watched, QEvent *event)
16 {
17     // 判断对象和事件
```

```

18     if(watched == ui->textEdit && event->type() == QEvent::KeyPress)
19     {
20         QKeyEvent* keyEv = (QKeyEvent*)event;
21         if(keyEv->key() == Qt::Key_Enter ||           // 小键盘确认
22             keyEv->key() == Qt::Key_Return)           // 大键盘回车
23         {
24             qDebug() << "我是回车， 被按下了...";
25             return true;
26         }
27     }
28     return false;
29 }
```

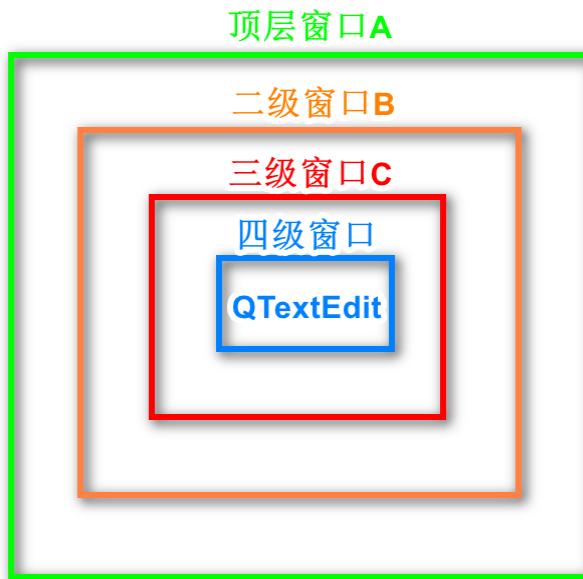
在示例代码的第 7 行：给多行编辑框控件安装了事件过滤器，由 this 对应的主窗口进行事件的过滤

在示例代码的第 15 行：主窗口通过重新事件过滤器函数，对多行编辑框控件进行事件的过滤，在函数体内部关于键盘事件的过滤需要判断按键是否是回车键，此处徐亚注意：

- Qt::Key_Enter 是小键盘上的回车（确认）键，有些键盘没有小键盘，因此也就没有该按键。
- Qt::Key_Return 是大键盘上的回车键

通过这样的处理，事件在被应用程序对象发送出去之后，进入到对应的窗口之前就被其父窗口过滤掉了。

如果在 Qt 的窗口中有多层嵌套的窗口，如下图：



先来描述一下这四层窗口的关系：

- 顶层窗口 A 的直接子窗口是 B，间接子窗口是 C, QTextEdit
- 二级窗口 B 的直接子窗口是 C，间接子窗口是 QTextEdit
- 三级窗口 C 的直接子窗口是 QTextEdit

在这种多层嵌套窗口中如果想要过滤掉 QTextEdit 的某些事件，可以交给 A 或者 B 或者 C 去处理，当然也可以给 QTextEdit 同时安装多个过滤器：

```

1 ui->textEdit->installEventFilter(窗口A对象);
2 ui->textEdit->installEventFilter(窗口B对象);
3 ui->textEdit->installEventFilter(窗口C对象);
```

如果一个对象存在多个事件过滤器，那么，最后一个安装的会第一个执行，也就是说窗口C先进行事件过滤，然后窗口B，最后窗口A。

注意事项：

事件过滤器和被安装过滤器的组件必须在同一线程，否则，过滤器将不起作用。另外，如果在安装过滤器之后，这两个组件到了不同的线程，那么，只有等到二者重新回到同一线程的时候过滤器才会有效。

十五、Qt程序的发布和打包

1. Qt 程序的发布

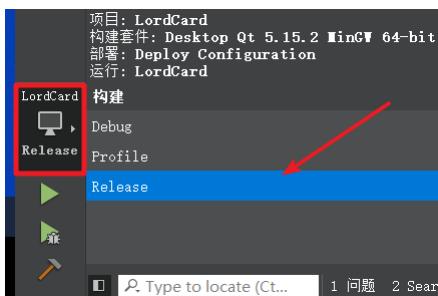
当 Qt 程序编写完成通过 IDE 编译就可以得到对应的可执行程序，这个可执行程序在本地运行是完全没有问题的（因为在本地有 Qt 环境，程序运行过程中可以加载到相关的动态库），但是如果我们要把这个 Qt 程序给到其他小伙伴使用可能就会出问题了，原因如下：

- 这个小伙伴本地根本没有 Qt 开发环境
- 这个小伙伴本地有 Qt 开发环境，但是和我们使用的版本不一致
- 这个小伙伴本地有 Qt 开发环境并且使用的版本与我们一致，但是没有配置环境变量

以上几种情况都会导致我们的小伙伴拿到可执行程序之后无法运行，下面来给大家讲一下解决方案。

1.1 生成 Release 版程序

在编写 Qt 程序的时候，不管我们使用的什么样的 IDE 都可以进行编译版本的切换，如果要发布程序需要切换为 Release 版本（Debug 为调试版本），编译器会对生成的 Release 版可执行程序进行优化，生成的可执行程序会更小。这里以 QtCreator 为例，截图如下：



模式选择完毕之后开始构建当前项目，最后找到生成的带 Release 后缀的构建目录，如下图所示：

build-LordCard-Desktop_Qt_5_15_2_MinGW_64_bit-Release >

名称	修改日期	类型	大小
debug	2021/6/14 10:50	文件夹	
release	2021/6/26 10:50	文件夹	
.qmake.stash	2021/6/14 10:50	STASH 文件	1 KB
LordCard_resource.rc	2021/6/14 10:50	Resource Script	1 KB
Makefile	2021/6/26 10:48	文件	28 KB
Makefile.Debug	2021/6/26 10:48	DEBUG 文件	257 KB
Makefile.Release	2021/6/26 10:48	RELEASE 文件	257 KB
object_script.LordCard.Debug	2021/6/26 10:48	DEBUG 文件	1 KB
object_script.LordCard.Release	2021/6/26 10:48	RELEASE 文件	1 KB
ui_buttongroup.h	2021/6/26 10:48	C Header 源文件	8 KB
ui_gamepanel.h	2021/6/26 10:48	C Header 源文件	3 KB
ui_scorepanel.h	2021/6/26 10:48	C Header 源文件	7 KB

进图到 release 目录中，在里面就能找到我们要的可执行程序了

build-LordCard-Desktop_Qt_5_15_2_MinGW_64_bit-Release > release

名称	修改日期	类型	大小
endingpanel.o	2021/6/26 10:49	O 文件	8 KB
gamecontrol.o	2021/6/26 10:49	O 文件	27 KB
gamepanel.o	2021/6/26 10:49	O 文件	125 KB
loading.o	2021/6/26 10:49	O 文件	7 KB
LordCard.exe	2021/6/26 10:50	应用程序	477 KB
LordCard_resource_res.o	2021/6/26 10:49	O 文件	242 KB
main.o	2021/6/26 10:49	O 文件	8 KB

1.2 发布

生成的可执行程序在运行的时候需要加载相关的 Qt 库文件，因此需要将这些动态库一并发布给使用者，Qt 官方给我们提供了相关的发布工具，通过这个工具就可以非常轻松的找出这些动态库文件了，这个工具叫做 `windeployqt.exe`，该文件位于 Qt 安装目录的编译套件目录的 `bin` 目录中，以我本地为例：`C:\Qt\5.15.2\mingw81_64\bin`。

- C:\Qt 是 Qt 的安装目录
- 5.15.2 是 Qt 的版本
- mingw81_64 是编译套件目录
- bin 存储 `windeployqt.exe` 文件的目录

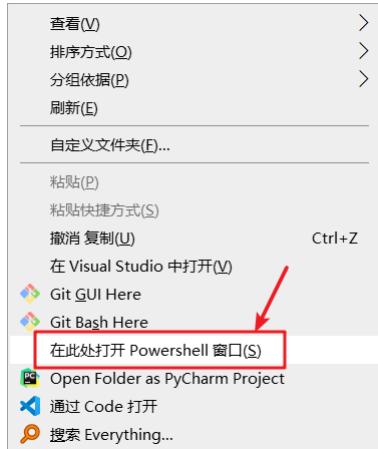
如果已经将这个路径设置到环境变量中了，那么在当前操作系统的任意目录下都可以访问 `windeployqt.exe`

Qt 环境变量设置

知道 Qt 提供的这个工具之后就可以继续向下进行了，首先将生成的 Release 版本的可执行程序放到一个新建的空目录中：

名称	修改日期	类型	大小
LordCard.exe	2021/6/26 10:50	应用程序	477 KB

进入到这个目录，按住键盘 shift键然后鼠标右键就可以弹出一个右键菜单



打开 Powershell窗口执行命令：

```

1 # LordCard.exe 是可执行程序的名字
2 # windeployqt.exe 的后缀 .exe 可以省略不写
3 windeployqt.exe LordCard.exe

```

```

PS C:\Users\Kevin\Desktop\LandLord> windeployqt.exe LordCard.exe
C:\Users\Kevin\Desktop\LandLord\LordCard.exe 64 bit, release executable
Adding Qt5Svg for qsvgicon.dll
Direct dependencies: Qt5Core Qt5Gui Qt5Multimedia Qt5Widgets
All dependencies : Qt5Core Qt5Gui Qt5Multimedia Qt5Network Qt5Widgets
To be deployed : Qt5Core Qt5Gui Qt5Multimedia Qt5Network Qt5Svg Qt5Widgets
Updating Qt5Core.dll.
Updating Qt5Gui.dll.
Updating Qt5Multimedia.dll.
Updating Qt5Network.dll.
Updating Qt5Svg.dll.
Updating Qt5Widgets.dll.
Updating libGLESv2.dll.
Updating libEGL.dll.
Updating D3Dcompiler_47.dll.
Updating opengl32sw.dll.
Updating libgcc_s_seh-1.dll.
Updating libstdc++-6.dll.
Updating libwinpthread-1.dll.

```

这样 LordCard.exe 需要的动态库会被全部拷贝到当前的目录中，如下图：

LandLord

名称	修改日期	类型	大小
audio	2021/6/26 11:16	文件夹	
bearer	2021/6/26 11:16	文件夹	
iconengines	2021/6/26 11:16	文件夹	
imageformats	2021/6/26 11:16	文件夹	
mediaservice	2021/6/26 11:16	文件夹	
platforms	2021/6/26 11:16	文件夹	
playlistformats	2021/6/26 11:16	文件夹	
styles	2021/6/26 11:16	文件夹	
translations	2021/6/26 11:16	文件夹	
D3Dcompiler_47.dll	2014/3/11 18:54	应用程序扩展	4,077 KB
libEGL.dll	2020/11/6 17:08	应用程序扩展	68 KB
libgcc_s_seh-1.dll	2018/5/12 14:11	应用程序扩展	75 KB
libGLESv2.dll	2020/11/6 17:08	应用程序扩展	6,152 KB
libstdc++-6.dll	2018/5/12 14:11	应用程序扩展	1,384 KB
libwinpthread-1.dll	2018/5/12 14:11	应用程序扩展	51 KB
LordCard.exe	2021/6/26 10:50	应用程序	477 KB
opengl32sw.dll	2016/6/14 20:00	应用程序扩展	20,433 KB
Qt5Core.dll	2020/11/6 17:08	应用程序扩展	7,995 KB

使用这种方式 Qt 会将一些用不到的动态库也拷贝到当前的目录中，如果确定用不到可以手动将其删除，如果不在意这些，完全可以不用理会，我选择后者。

现在一个绿色免安装版的程序就得到了，可以将这个目录打个压缩包发送给自己的小伙伴，但是这种方式终究比较 low，我们可以将这个目录中的文件制作成一个安装包，这样档次一下就上去了。

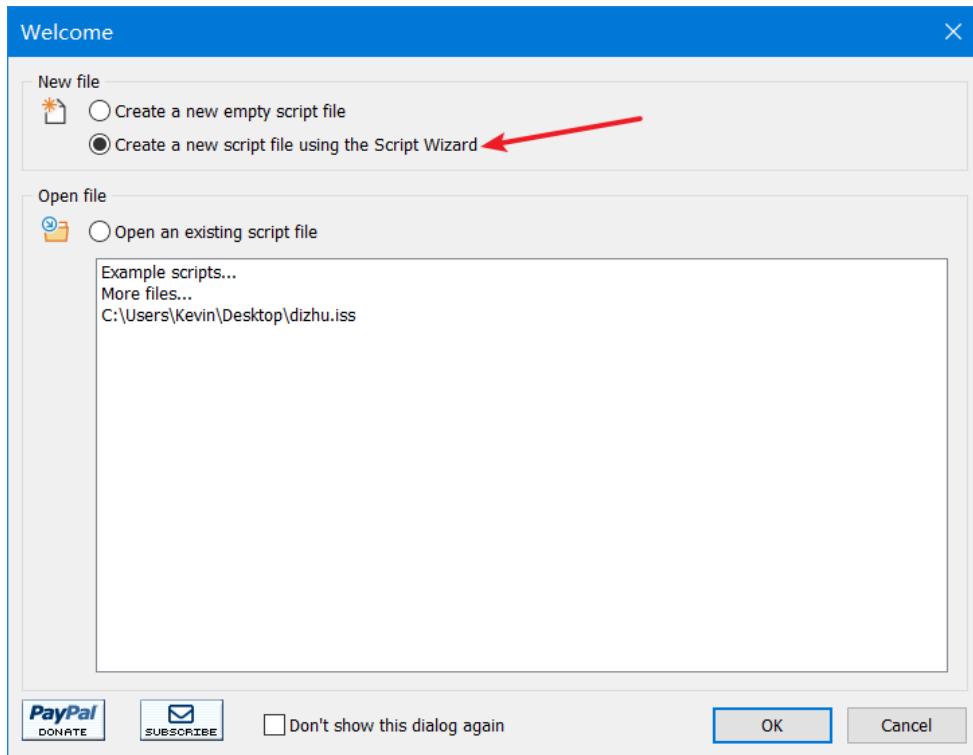
2.Qt 程序打包

将应用程序和相关的动态库打包成安装包的工具有很多，我自己用过两个一个是 NIS Edit，一个是 Inno Setup 这是一个免费的 Windows 安装程序制作软件，小巧、简便、精美。

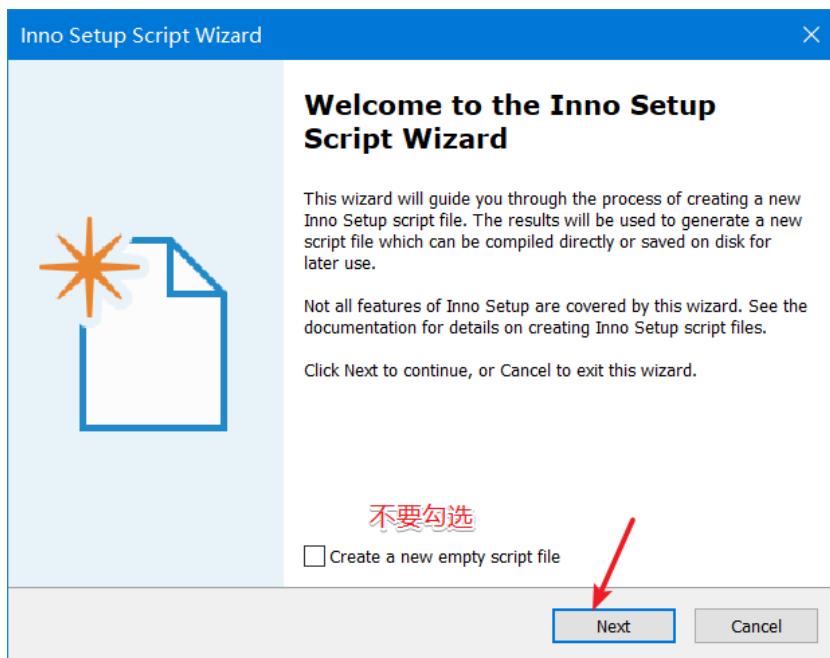
官方下载地址：<http://www.jrsoftware.org/isdl.php#stable>

其实这两个工具的使用方法是几乎一样的，下面拿 Inno Setup 使用举例。

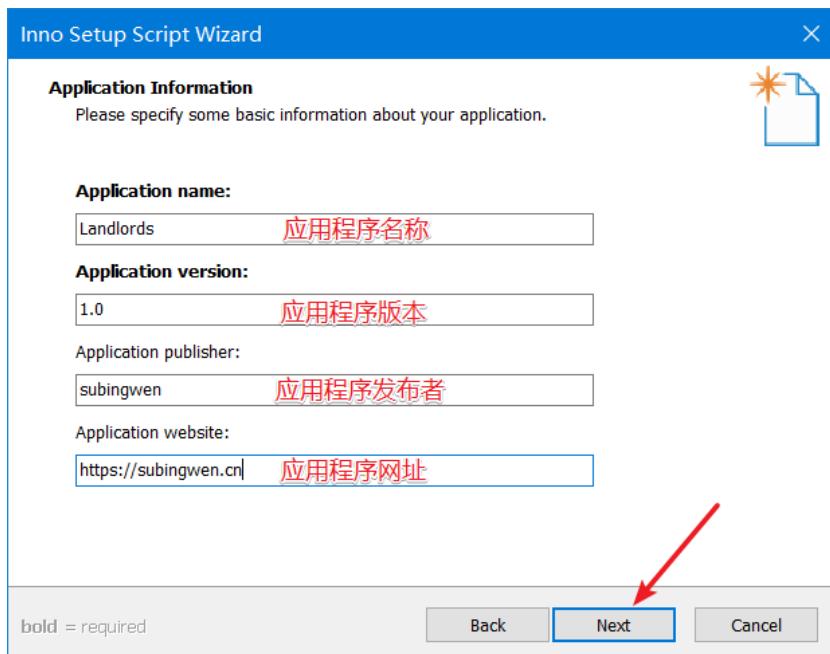
第一步：创建一个带向导的脚本文件



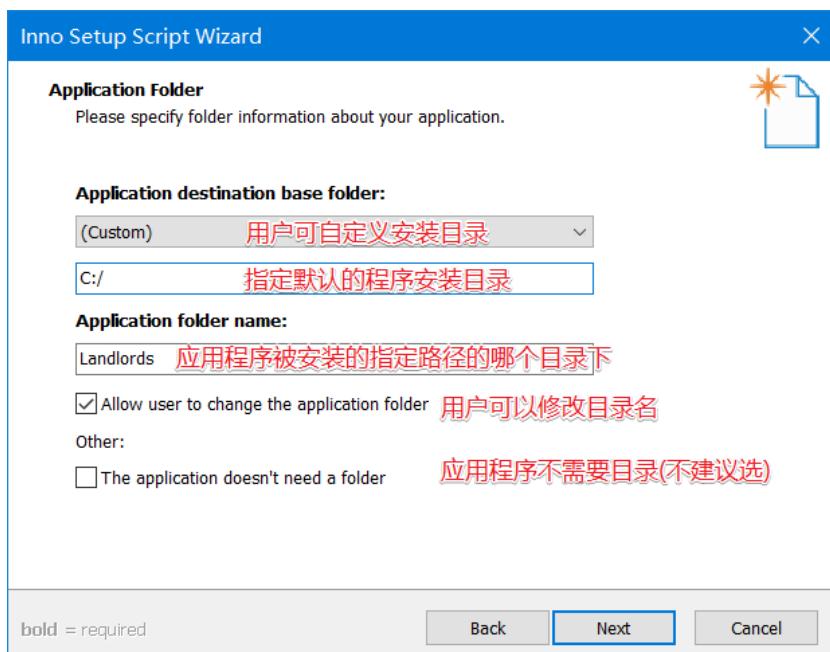
第二步：直接 Next，不要创建空的脚本文件



第三步：填写相关的应用程序信息



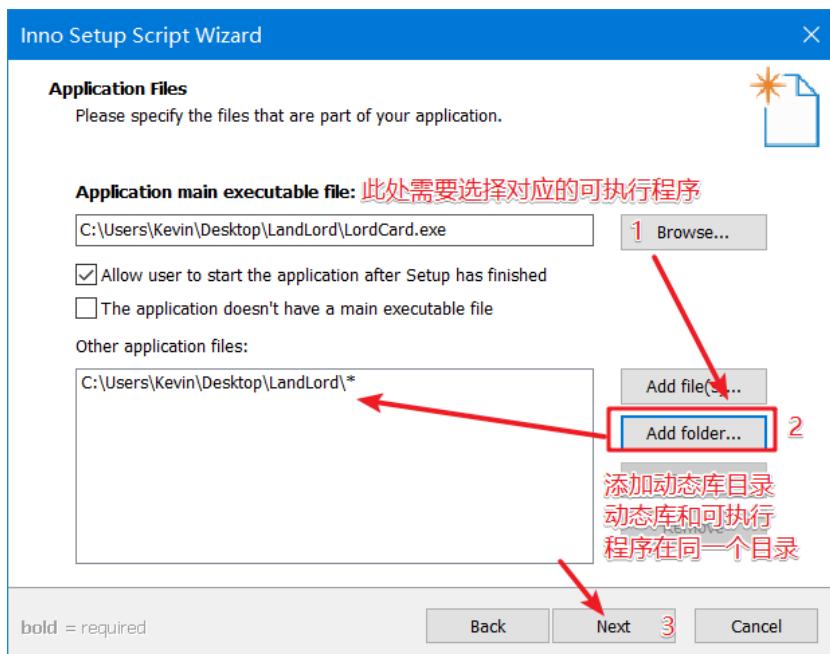
第四步：指定应用程序的安装目录相关的信息



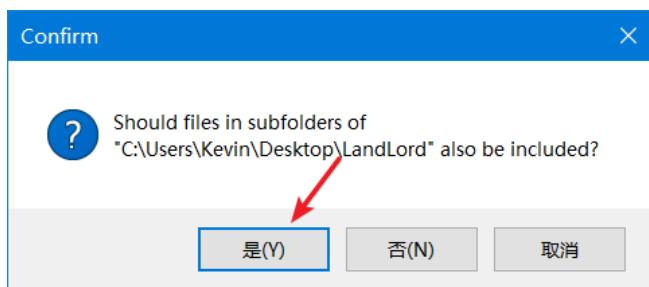
第五步：选择可执行程序和相关的动态库，此处参考的是前边的 1.2 章节中的目录

名称	修改日期	类型	大小
audio	2021/6/26 11:16	文件夹	
bearer	2021/6/26 11:16	文件夹	
iconengines	2021/6/26 11:16	文件夹	
imageformats	2021/6/26 11:16	文件夹	
mediaservice	2021/6/26 11:16	文件夹	
platforms	2021/6/26 11:16	文件夹	
playlistformats	2021/6/26 11:16	文件夹	
styles	2021/6/26 11:16	文件夹	
translations	2021/6/26 11:16	文件夹	
D3Dcompiler_47.dll	2014/3/11 18:54	应用程序扩展	4,077 KB
libEGL.dll	2020/11/6 17:08	应用程序扩展	68 KB
libgcc_s_seh-1.dll	2018/5/12 14:11	应用程序扩展	75 KB
libGLESv2.dll	2020/11/6 17:08	应用程序扩展	6,152 KB
libstdc++-6.dll	2018/5/12 14:11	应用程序扩展	1,384 KB
libwinpthread-1.dll	2018/5/12 14:11	应用程序扩展	51 KB
LordCard.exe	2021/6/26 10:50	应用程序	477 KB
opengl32sw.dll	2016/6/14 20:00	应用程序扩展	20,433 KB
Qt5Core.dll	2020/11/6 17:08	应用程序扩展	7,995 KB

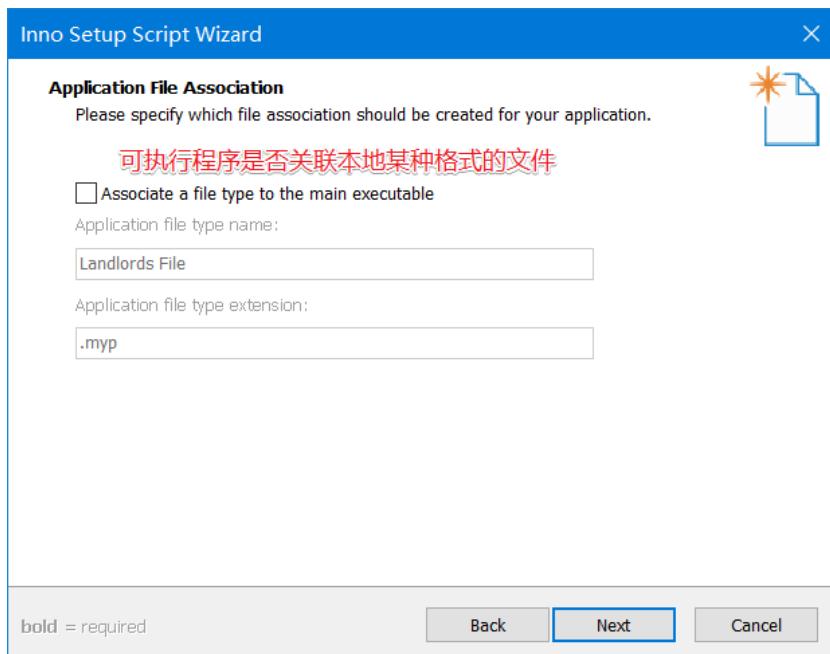
基于这个目录现在相关的文件和目录：



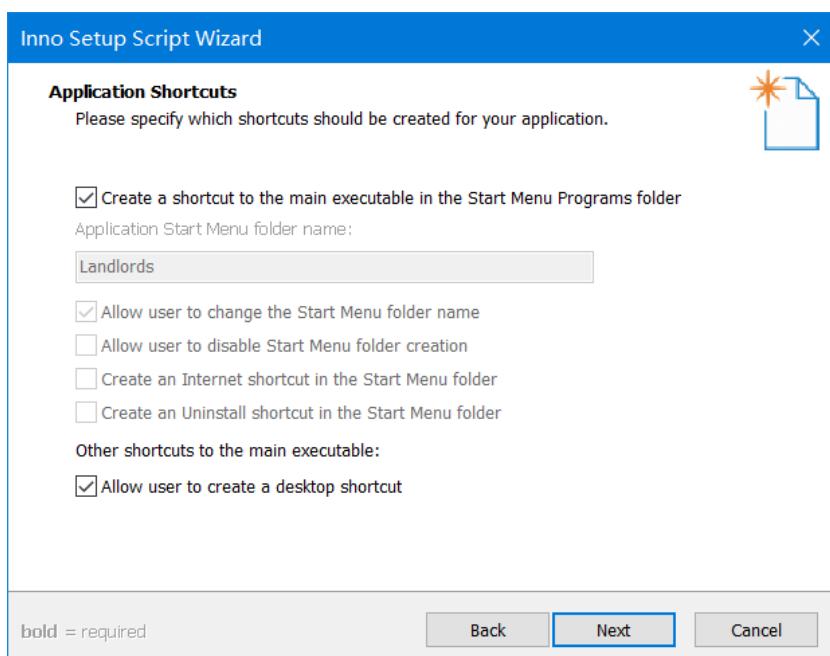
由于可执行程序关系的动态库有很多，所以可以直接添加动态库的目录，选中对应的目录之后，如果里边还有子目录会弹出如下对话框，选择是即可，需要包含这些子目录。



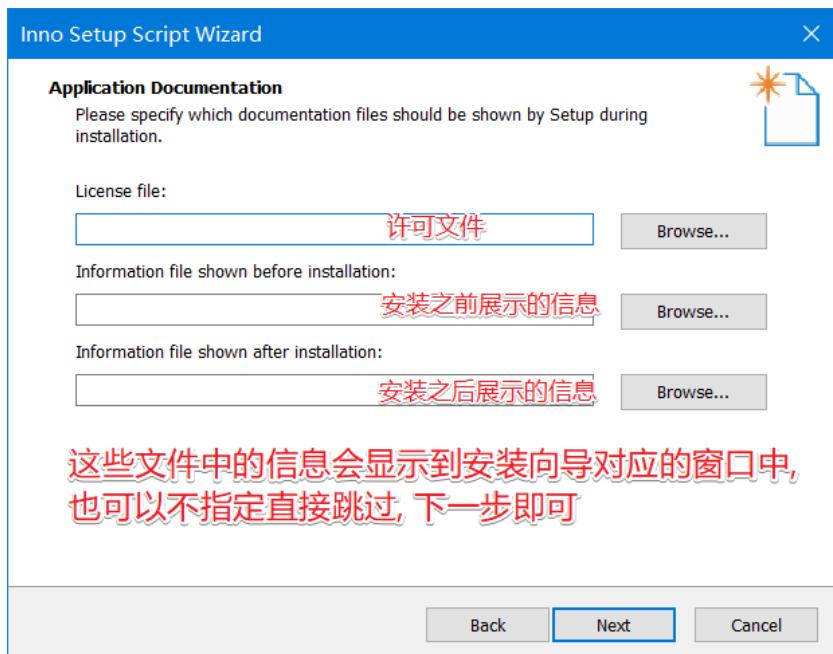
第六步：给可执行程序管理本地的某种格式的磁盘文件（比如记事本程序会自动关联本地的 .txt 文件），对于我的可执行程序来说无需关联，因此没有做任何设置，直接下一步



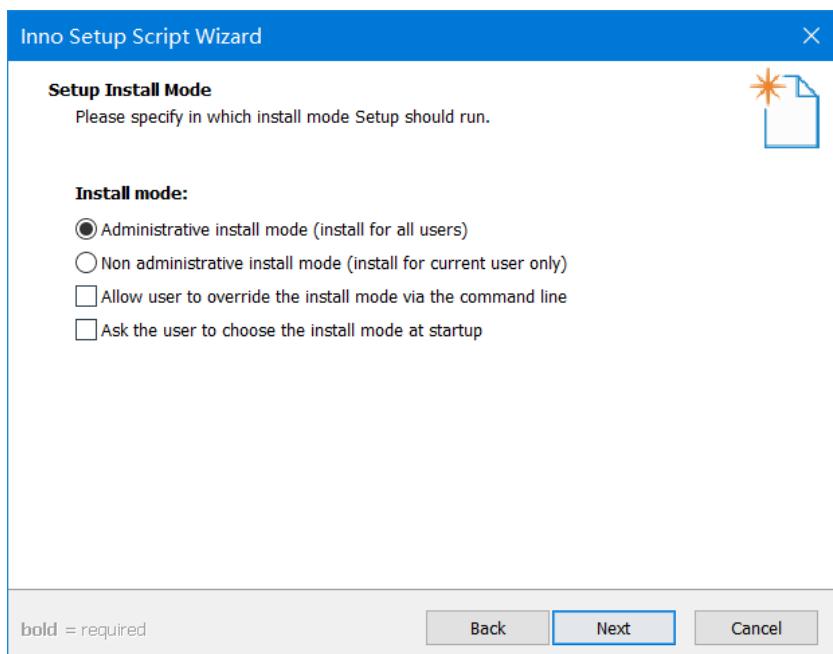
第七步：给应用程序创建快捷方式，此处没有进行任何设置，使用的默认选项



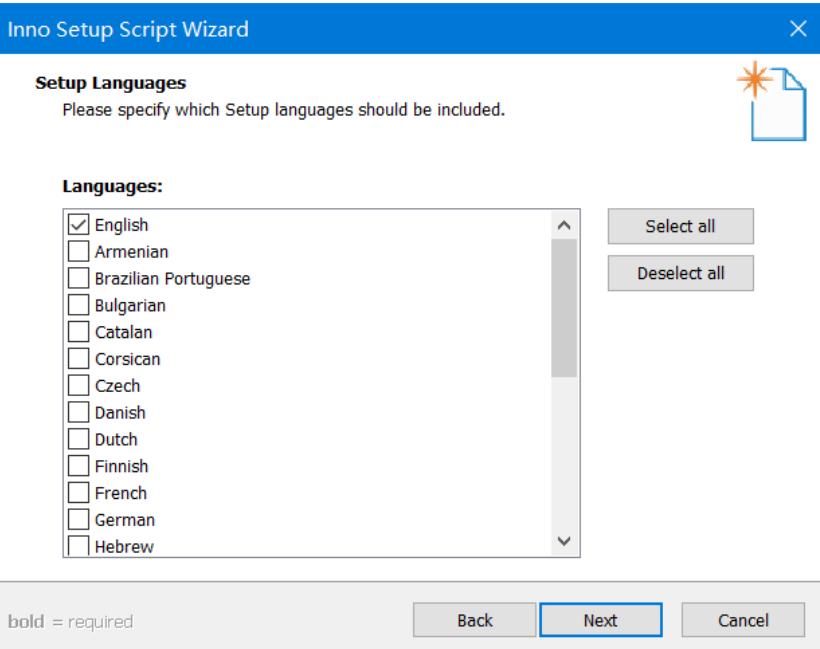
第八步：指定文档文件，文件中的内容会显示到安装向导的相关窗口中，可以选择不指定，直接跳过。



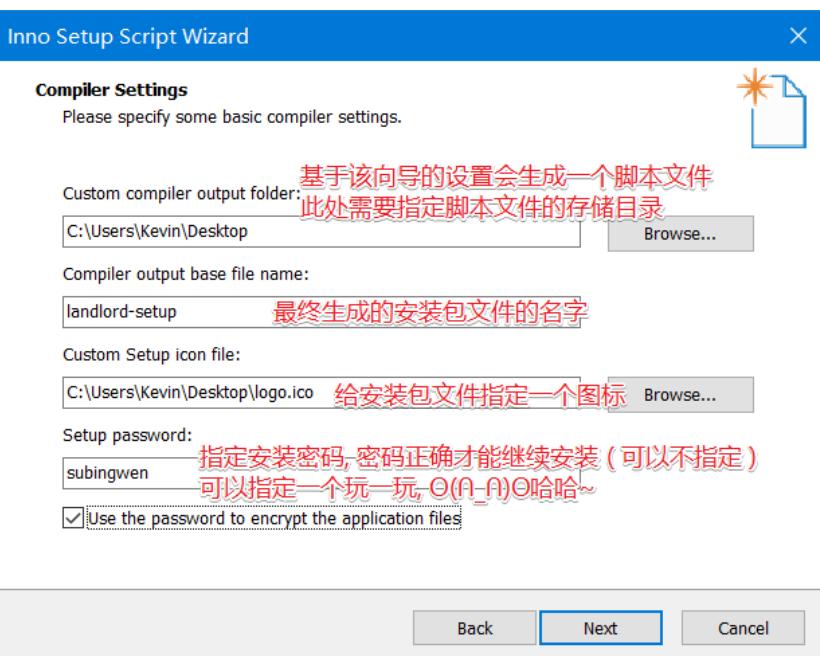
第九步：选择安装模式（给系统的当前用户安装还是给所有用户安装），根据自己喜好指定即可



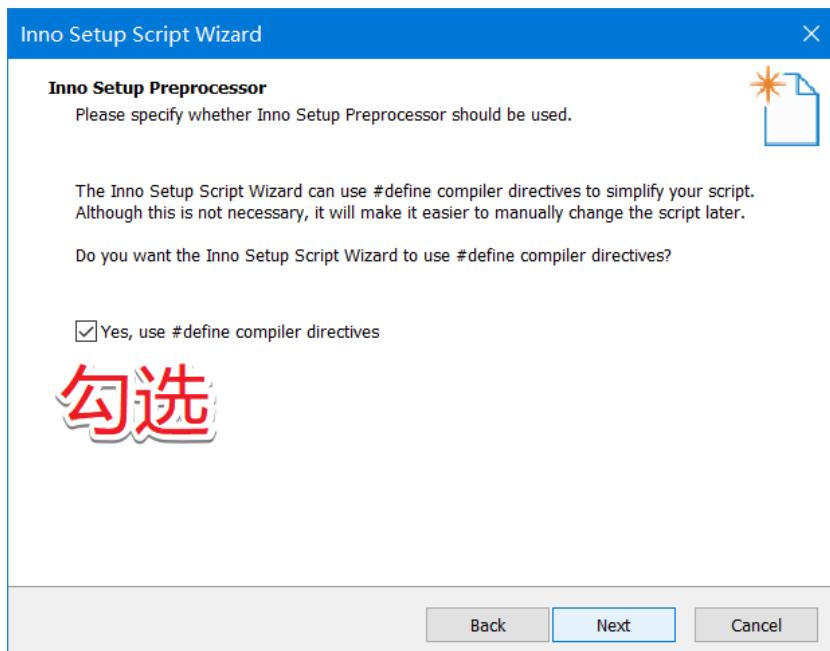
第十步：选择安装语言（这个工具没有提供中文，因此只能选择英文）



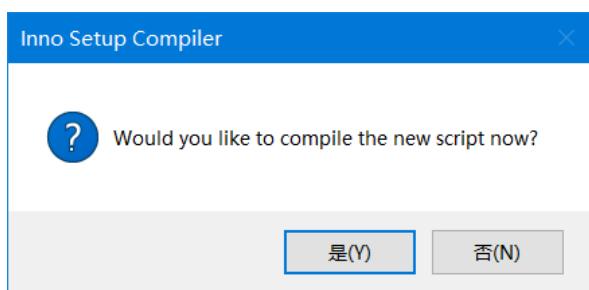
第十一步：指定安装包文件的相关信息



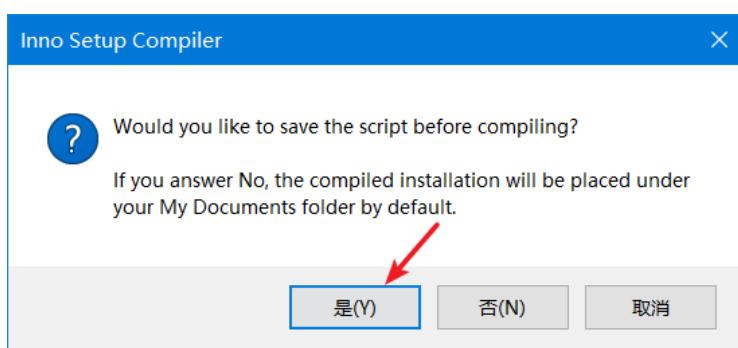
第十二步：向导结束，终于结束了。。。



第十三步：提示是否要编译生成的脚本文件，脚本编译完成之后，安装包就生成了。



之后弹出第二个对话框，建议通过向导生成的这个脚本文件，这样以后就可以直接基于这个脚本打包程序生成安装包了。



编译完成之后，就可以去保存脚本文件的目录找生成的安装文件了

1111.iss - Inno Setup Compiler 6.2.0

File Edit View Build Run Tools Help

Main Script Preprocessor Output

```
; Script generated by the Inno Setup Script Wizard.  
; SEE THE DOCUMENTATION FOR DETAILS ON CREATING INNO SETUP SCRIPT FILES.  
  
#define MyAppName "Landlords"  
#define MyAppVersion "1.0"  
#define MyAppPublisher "subingwen"  
#define MyAppURL "https://subingwen.cn"  
#define MyAppExeName "LordCard.exe"  
  
[Setup]  
; NOTE: The value of AppId uniquely identifies this application. Do not  
; (To generate a new GUID, click Tools | Generate GUID inside the IDE)  
AppId={{F7979D32-A963-4981-BAF3-48FE1EC039E8}  
AppName={#MyAppName}  
AppVersion={#MyAppVersion}  
;AppVerName={#MyAppName} {#MyAppVersion}  
AppPublisher={#MyAppPublisher}  
AppPublisherURL={#MyAppURL}  
AppSupportURL={#MyAppURL}  
AppUpdatesURL={#MyAppURL}  
DefaultDirName=C:/ {#MyAppName}  
DisableProgramGroupPage=yes  
  
Compressing: C:\Users\Kevin\Desktop\LandLord\translations\qt_pl.qm  
Compressing: C:\Users\Kevin\Desktop\LandLord\translations\qt_ru.qm  
Compressing: C:\Users\Kevin\Desktop\LandLord\translations\qt_sk.qm  
Compressing: C:\Users\Kevin\Desktop\LandLord\translations\qt_tr.qm  
Compressing: C:\Users\Kevin\Desktop\LandLord\translations\qt_uk.qm  
Compressing: C:\Users\Kevin\Desktop\LandLord\translations\qt_zh_TW.qm  
Compressing Setup program executable  
Updating version info (SETUP.EXE)  
*** Finished. [12:20:33, 00:30.500 elapsed]
```

脚本文件内容, 可以直接修改这个脚本文件

编译脚本文件过程中的日志输出

Compiler Output Debug Output Debug Call Stack Find Results

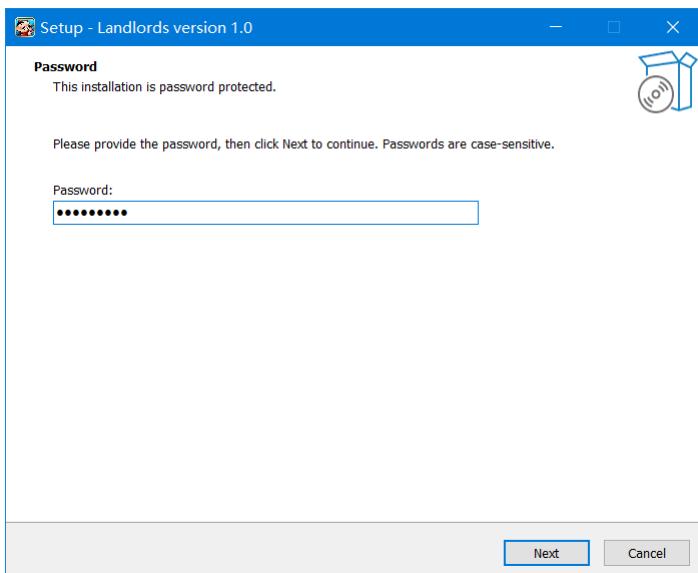
1: 1 Insert

3. 安装

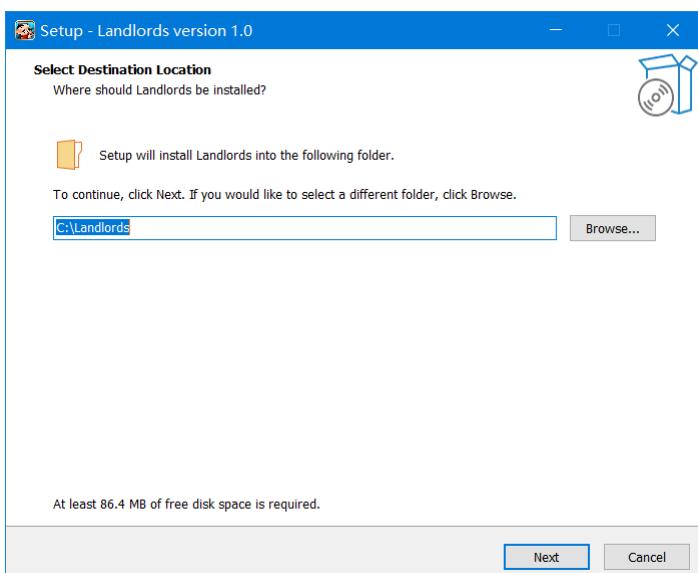
双击生成的安装包文件



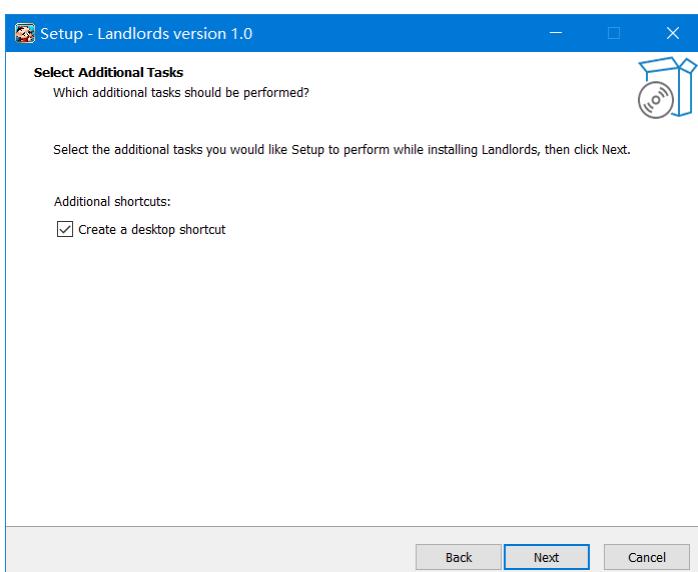
输入安装密码



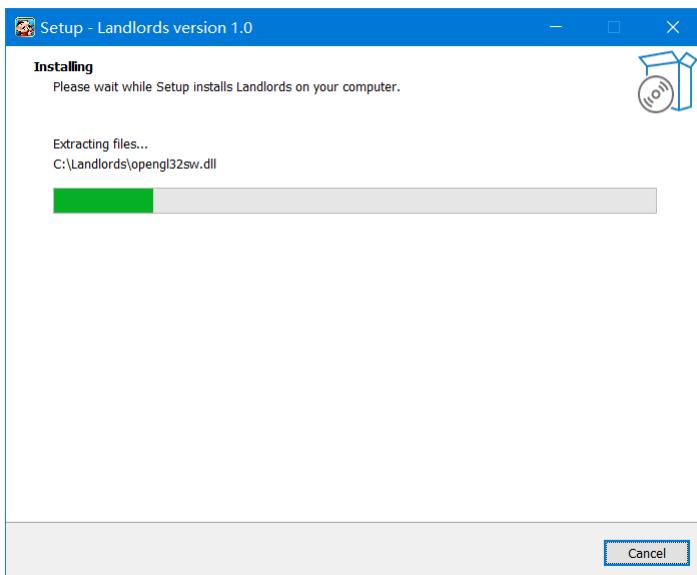
指定安装路径



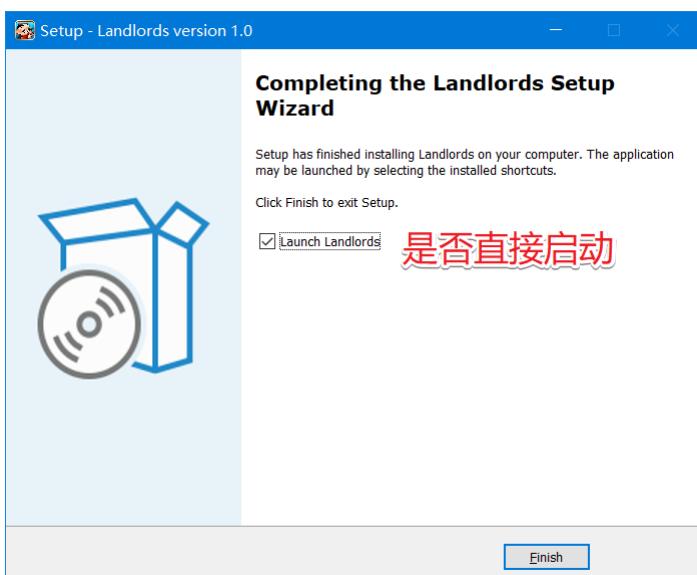
指定是否创建快捷方式



开始安装应用程序



安装完成，可以指定直接启动安装的应用程序



最后启动游戏测试下是否可以运行



十六、Json的操作

1. json文件

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于 ECMAScript (欧洲计算机协会制定的 js 规范) 的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。

关于上面的描述可以精简为一句话：Json 是一种数据格式，和语言无关，在什么语言中都可以使用 Json。基于这种通用的数据格式，一般处理两方面的任务：

- 组织数据（数据序列化），用于数据的网络传输
- 组织数据（数据序列化），写磁盘文件实现数据的持久化存储（一般以.json 作为文件后缀）

Json 中主要有两种数据格式：Json 数组和 Json 对象，并且这两种格式可以交叉嵌套使用，下面依次介绍下这两种数据格式

1.1 Json 数组

Json 数组使用 [] 表示，[] 里边是元素，元素和元素之间使用逗号间隔，最后一个元素后边没有逗号，一个 Json 数组中支持同时存在多种不同类型的成员，包括：整形、浮点、字符串、布尔类型、json 数组、json 对象、空值-null。由此可见 Json 数组比起 C/C++ 数组要灵活很多。

- Json 数组中的元素数据类型一致

```
1 // 整形
2 [1,2,3,4,5]
3 // 字符串
4 ["luffy", "sanji", "zoro", "nami", "robin"]
```

- Json 数组中的元素数据类型不一致

```
1 [12, 13.34, true, false, "hello,world", null]
```

- Json 数组中的数组嵌套使用

```
1 [
2   ["cat", "dog", "panda", "beer", "rabbit"],
3   ["北京", "上海", "天津", "重庆"],
4   ["luffy", "boy", 19]
5 ]
```

- Json 数组和对象嵌套使用

```
1 | [
2 |     "luffy": {
3 |         "age": 19,
4 |         "father": "Monkey · D · Dragon",
5 |         "grandpa": "Monkey D Garp",
6 |         "brother": "Portgas D Ace",
7 |         "brother": "Sabo"
8 |     }
9 | ]
```

1.2 Json 对象

Json 对象使用 {} 来描述，每个 Json 对象中可以存储若干个元素，每一个元素对应一个键值对（key: value 结构），元素和元素之间使用逗号间隔，最后一个元素后边没有逗号。对于每个元素中的键值对有以下细节需要注意：

1. 键值 (key) 必须是字符串，位于同一层级的键值不要重复（因为是通过键值取出对应的 value 值）
2. value 值的类型是可选的，可根据实际需求指定，可用类型包括：整形、浮点、字符串、布尔类型、json数组、json对象、空值-null。

使用 Json 对象描述一个人的信息：

```
1 | {
2 |     "Name": "Ace",
3 |     "Sex": "man",
4 |     "Age": 20,
5 |     "Family": {
6 |         "Father": "Goł·D·Roger",
7 |         "Mother": "Portgas·D·Rouge",
8 |         "Brother": ["Sabo", "Monkey D. Luffy"]
9 |     },
10 |     "IsAlive": false,
11 |     "Comment": "yyds"
12 | }
```

1.3 注意事项

通过上面的介绍可用看到，Json 的结构虽然简单，但是进行嵌套之后就可以描述很复杂的事情，在项目开发过程中往往需要我们根据实际需求自己定义 Json 格式用来存储项目数据。

另外，如果需要将 Json 数据持久化到磁盘文件中，需要注意一个问题：在一个 Json 文件中只能有一个 Json 数组或者 Json 对象的根节点，不允许同时存储多个并列的根节点。下面举例说明：

- 错误的写法

```

1 // test.json
2 {
3     "name": "luffy",
4     "age": 19
5 }
6 {
7     "user": "ace",
8     "passwd": "123456"
9 }
```

错误原因：在一个Json文件中有两个并列的Json根节点（并列包含Json对象和Json对象、Json对象和Json数组、Json数组和Json数组），根节点只能有一个。

- 正确的写法

```

1 // test.json
2 {
3     "Name": "Ace",
4     "Sex": "man",
5     "Age": 20,
6     "Family": {
7         "Father": "Gol·D·Roger",
8         "Mother": "Portgas·D·Rouge",
9         "Brother": ["Sabo", "Monkey D. Luffy"]
10    },
11    "IsAlive": false,
12    "Comment": "yyds"
13 }
```

在上面的例子中通过Json对象以及Json数组的嵌套描述了一个人的身份信息，并且根节点只有一个就是Json对象，如果还需要使用Json数组或者Json对象描述其他信息，需要将这些信息写入到其他文件中，不要和这个Json对象并列写入到同一个文件里边，切记！！！

2. Qt中Json的操作

在[Json的两种格式](#)中介绍了Json的格式以及应用场景。由于这种数据格式与语言无关，下面介绍一下Json在Qt中的使用。

从Qt 5.0开始提供了对Json的支持，我们可以直接使用Qt提供的Json类进行数据的组织和解析。相关的类常用的主要有四个，具体如下：

Json类	介绍
QJsonDocument	它封装了一个完整的JSON文档，并且可以从UTF-8编码的基于文本的表示以及Qt自己的二进制格式读取和写入该文档。
QJsonArray	JSON数组是一个值列表。可以通过从数组中插入和删除QJsonValue来操作该列表。
QJsonObject	JSON对象是键值对的列表，其中键是唯一的字符串，值由QJsonValue表示。
QJsonValue	该类封装了JSON支持的数据类型。

2.1 QJsonValue

在 Qt 中 QJsonValue 可以封装的基础数据类型有六种（和 Json 支持的类型一致），分别为：

- 布尔类型：QJsonValue::Bool
- 浮点类型（包括整形）：QJsonValue::Double
- 字符串类型：QJsonValue::String
- Json 数组类型：QJsonValue::Array
- Json 对象类型：QJsonValue::Object
- 空值类型：QJsonValue::Null

这个类型可以通过 QJsonValue 的构造函数被封装为一个类对象：

```
1 // Json对象
2 QJsonValue(const QJsonObject &o);
3 // Json数组
4 QJsonValue(const QJsonArray &a);
5 // 字符串
6 QJsonValue(const char *s);
7 QJsonValue(QLatin1String s);
8 QJsonValue(const QString &s);
9 // 整形 and 浮点型
10 QJsonValue(qint64 v);
11 QJsonValue(int v);
12 QJsonValue(double v);
13 // 布尔类型
14 QJsonValue(bool b);
15 // 空值类型
16 QJsonValue(QJsonValue::Type type = Null);
```

如果我们得到一个 QJsonValue 对象，如何判断内部封装的到底是什么类型的数据呢？这时候就需要调用相关的判断函数了，具体如下：

```
1 // 是否是Json数组
2 bool isArray() const;
3 // 是否是Json对象
4 bool isObject() const;
5 // 是否是布尔类型
6 bool isBool() const;
7 // 是否是浮点类型(整形也是通过该函数判断)
8 bool isDouble() const;
9 // 是否是空值类型
10 bool isNull() const;
11 // 是否是字符串类型
12 bool isString() const;
13 // 是否是未定义类型(无法识别的类型)
14 bool isUndefined() const;
```

通过判断函数得到对象内部数据的实际类型之后，如果有需求就可以再次将其转换为对应的基础数据类型，对应的 API 函数如下：

```
1 // 转换为Json数组
2 QJsonArray toArray(const QJsonArray &defaultValue) const;
3 QJsonArray toArray() const;
```

```
4 // 转换为布尔类型
5 bool toBool(bool defaultValue = false) const;
6 // 转换为浮点类型
7 double toDouble(double defaultValue = 0) const;
8 // 转换为整形
9 int toInt(int defaultValue = 0) const;
10 // 转换为Json对象
11 QObject toObject(const QObject &defaultValue) const;
12 QObject toObject() const;
13 // 转换为字符串类型
14 QString toString() const;
15 QString toString(const QString &defaultValue) const;
```

2.2 QJsonObject

QJsonObject 封装了 Json 中的对象，在里边可以存储多个键值对，为了方便操作，键值为字符串类型，值为 QJsonValue 类型。关于这个类的使用类似于 C++ 中的 STL 类，仔细阅读 API 文档即可熟练上手使用，下面介绍一些常用 API 函数：

- 如何创建空的 Json 对象

```
1 | QJsonObject::QJsonObject(); // 构造空对象
```

- 将键值对添加到空对象中

```
1 | iterator QJsonObject::insert(const QString &key, const QJsonValue
&value);
```

- 获取对象中键值对个数

```
1 | int QJsonObject::count() const;
2 | int QJsonObject::size() const;
3 | int QJsonObject::length() const;
```

- 通过 key 得到 value

```
1 | QJsonValue QJsonObject::value(const QString &key) const; // utf8
2 | QJsonValue QJsonObject::value(QLatin1String key) const; // 字符串不支
持中文
3 | QJsonValue QJsonObject::operator[](const QString &key) const;
4 | QJsonValue QJsonObject::operator[](QLatin1String key) const;
```

- 删除键值对

```
1 | void QJsonObject::remove(const QString &key);
2 | QJsonValue QJsonObject::take(const QString &key); // 返回key对应的value
值
```

- 通过 key 进行查找

```
1 | iterator QJsonObject::find(const QString &key);
2 | bool QJsonObject::contains(const QString &key) const;
```

- 遍历，方式有三种：
 - 使用相关的迭代器函数
 - 使用 [] 的方式遍历，类似于遍历数组，[] 中是键值
 - 先得到对象中所有的键值，在遍历键值列表，通过 key 得到 value 值

```
1 | QStringList QJsonObject::keys() const;
```

2.3. QJsonArray

QJsonArray 封装了 Json 中的数组，在里边可以存储多个元素，为了方便操作，所有的元素类统一为 QJsonValue 类型。关于这个类的使用类似于 C++ 中的 STL 类，仔细阅读 API 文档即可熟练上手使用，下面介绍一些常用 API 函数：

- 创建空的 Json 数组

```
1 | QJsonArray::QJsonArray();
```

- 添加数据

```
1 | void QJsonArray::append(const QJsonValue &value); // 在尾部追加
2 | void QJsonArray::insert(int i, const QJsonValue &value); // 插入到 i 的位置之前
3 | iterator QJsonArray::insert(iterator before, const QJsonValue &value);
4 | void QJsonArray::prepend(const QJsonValue &value); // 添加到数组头部
5 | void QJsonArray::push_back(const QJsonValue &value); // 添加到尾部
6 | void QJsonArray::push_front(const QJsonValue &value); // 添加到头部
```

- 计算数组元素的个数

```
1 | int QJsonArray::count() const;
2 | int QJsonArray::size() const;
```

- 从数组中取出某一个元素的值

```
1 | QJsonValue QJsonArray::at(int i) const;
2 | QJsonValue QJsonArray::first() const; // 头部元素
3 | QJsonValue QJsonArray::last() const; // 尾部元素
4 | QJsonValueRef QJsonArray::operator[](int i);
```

- 删除数组中的某一个元素

```
1 | iterator QJsonArray::erase(iterator it); // 基于迭代器删除
2 | void QJsonArray::pop_back(); // 删除尾部
3 | void QJsonArray::pop_front(); // 删除头部
4 | void QJsonArray::removeAt(int i); // 删除 i 位置的元素
5 | void QJsonArray::removeFirst(); // 删除头部
6 | void QJsonArray::removeLast(); // 删除尾部
7 | QJsonValue QJsonArray::takeAt(int i); // 删除 i 位置的原始，并返回删除的元素的值
```

- Json 数组的遍历，常用的方式有两种：

- 可以使用迭代器进行遍历（和使用迭代器遍历 STL 容器一样）
- 可以使用数组的方式遍历

2.4. QJsonDocument

它封装了一个完整的 JSON 文档，并且可以从 UTF-8 编码的基于文本的表示以及 Qt 自己的二进制格式读取和写入该文档。QJsonObject 和 QJsonArray 这两个对象中的数据是不能直接转换为字符串类型的，如果要进行数据传输或者数据的持久化，操作的都是字符串类型而不是 QJsonObject 或者 QJsonArray 类型，我们需要通过一个 Json 文档类进行二者之间的转换。

下面依次介绍一下这两个转换流程应该如何操作：

- QJsonObject 或者 QJsonArray ===> 字符串

1. 创建 QJsonDocument 对象

```
1 | QJsonDocument::QJsonDocument(const QJsonObject &object);
2 | QJsonDocument::QJsonDocument(const QJsonArray &array);
```

可以看出，通过构造函数就可以将实例化之后的 **QJsonObject** 或者 **QJsonArray** 转换为 QJsonDocument 对象了。

2. 将文件对象中的数据进行序列化

```
1 | // 二进制格式的json字符串
2 | QByteArray QJsonDocument::toBinaryData() const;
3 | // 文本格式
4 | QByteArray QJsonDocument::toJson(JsonFormat format = Indented)
  | const;
```

通过调用 toxxx() 方法就可以得到文本格式或者二进制格式的 Json 字符串了。

3. 使用得到的字符串进行数据传输，或者磁盘文件持久化

- 字符串 ===> QJsonObject 或者 QJsonArray

一般情况下，通过网络通信或者读磁盘文件就会得到一个 Json 格式的字符串，如果想要得到相关的原始数据就需要对字符串中的数据进行解析，具体解析流程如下：

1. 将得到的 Json 格式字符串通过 QJsonDocument 类的静态函数转换为 QJsonDocument 类对象

```
1 | [static] QJsonDocument QJsonDocument::fromBinaryData(const
  | QByteArray &data, DataValidation validation = validate);
2 | // 参数文件格式的json字符串
3 | [static] QJsonDocument QJsonDocument::fromJson(const QByteArray
  | &json, QJsonParseError *error = Q_NULLPTR);
```

2. 将文档对象转换为 json 数组 / 对象

```

1 // 判断文档对象中存储的数据是不是数组
2 bool QJsonDocument::isArray() const;
3 // 判断文档对象中存储的数据是不是json对象
4 bool QJsonDocument::isObject() const
5
6 // 文档对象中的数据转换为json对象
7 QJsonObject QJsonDocument::object() const;
8 // 文档对象中的数据转换为json数组
9 QJsonArray QJsonDocument::array() const;

```

3. 通过调用 `QJsonArray`, `QJsonObject` 类提供的 API 读出存储在对象中的数据。

关于 Qt 中 Json 数据对象以及字符串之间的转换的操作流程是固定的，我们在编码过程中只需要按照上述模板处理即可，相关的操作是没有太多的技术含量可言的。

2.5. 举例

2.5.1 写文件

```

1 void writeJson()
2 {
3     QJsonObject obj;
4     obj.insert("Name", "Ace");
5     obj.insert("Sex", "man");
6     obj.insert("Age", 20);
7
8     QJsonObject subObj;
9     subObj.insert("Father", "Gol·D·Roger");
10    subObj.insert("Mother", "Portgas·D·Rouge");
11    QJsonArray array;
12    array.append("Sabo");
13    array.append("Monkey D. Luffy");
14    subObj.insert("Brother", array);
15    obj.insert("Family", subObj);
16    obj.insert("IsAlive", false);
17    obj.insert("Comment", "yyds");
18
19    QJsonDocument doc(obj);
20    QByteArray json = doc.toJson();
21
22    QFile file("d:\\ace.json");
23    file.open(QFile::WriteOnly);
24    file.write(json);
25    file.close();
26 }

```

2.5.2 读文件

```

1 void MainWindow::readJson()
2 {
3     QFile file("d:\\ace.json");
4     file.open(QFile::ReadOnly);
5     QByteArray json = file.readAll();
6     file.close();
7
8     QJsonDocument doc = QJsonDocument::fromJson(json);

```

```

9     if(doc.isObject())
10    {
11        QJsonObject obj = doc.object();
12        QStringList keys = obj.keys();
13        for(int i=0; i<keys.size(); ++i)
14        {
15            QString key = keys.at(i);
16            QJsonValue value = obj.value(key);
17            if(value.isBool())
18            {
19                qDebug() << key << ":" << value.toBool();
20            }
21            if(value.isString())
22            {
23                qDebug() << key << ":" << value.toString();
24            }
25            if(value.isDouble())
26            {
27                qDebug() << key << ":" << value.toInt();
28            }
29            if(value.isObject())
30            {
31                qDebug()<< key << ":";
32                // 直接处理内部键值对，不再进行类型判断的演示
33                QJsonObject subobj = value.toObject();
34                QStringList ls = subobj.keys();
35                for(int i=0; i<ls.size(); ++i)
36                {
37                    QJsonValue subval = subobj.value(ls.at(i));
38                    if(subval.isString())
39                    {
40                        qDebug() << "    " << ls.at(i) << ":" <<
41                        subval.toString();
42                    }
43                    if(subval.isArray())
44                    {
45                        QJsonArray array = subval.toArray();
46                        qDebug() << "    " << ls.at(i) << ":";
47                        for(int j=0; j<array.size(); ++j)
48                        {
49                            // 因为知道数组内部全部为字符串，不再对元素类型进行判断
50                            qDebug() << "          " << array[j].toString();
51                        }
52                    }
53                }
54            }
55        }
56    }

```

一般情况下，对于 Json 字符串的解析函数都是有针对性的，因为需求不同设计的 Json 格式就会有所不同，所以不要试图写出一个通用的 Json 解析函数，这样只会使函数变得臃肿而且不易于维护，每个 Json 格式对应一个相应的解析函数即可。

上面的例子中为了给大家演示 Qt 中 Json 类相关 API 函数的使用将解析步骤写的复杂了，因为在解析的时候我们是知道 Json 对象中的所有 key 值的，可以直接通过 key 值将对应的 value 值取出来，因此上面程序中的一些判断和循环其实是可以省去的。

