QT creator用法

# 路径问题

在做Qt项目的时候，我们难免遇到到文件路径问题。 如QFile file("text.txt")加载不成功、QPixmap("../text.png") 加载图片不成功等等。今天就来做一个关于Qt路径问题的小结！

首先需要弄清楚两个概念：绝对路径与相对路径

**绝对路径**: 文件在硬盘上真正存在的路径。例如“text.jpg”这个图片是存放在硬盘  
       的“E:\book\目录下，那么 “text.jpg”这个图片的绝对路径就是“E:\book\text.jpg"  
  
**相对路径**:由这个文件所在的路径引起的跟其它文件(或文件夹)的路径关系  
      如当前file1的路径是/user/File/file1/file1.txt,   
      File2的路径是/user/File/file2/file2.txt  
      那么file2相对于file的路径就是“../file2/file2.txt”。  
      相对路径里常使用“../”来表示上一级目录  
  
弄清楚了上面两个概念，在来弄清楚我们常用到的“:/”与"./"：  
**./**  ：  表示当前路径 如“./log/log1.txt” 表示当前路径下的log目录下的log1.txt  
**:/**  ：  表示对资源的引用,引用资源文件路径 如“:/image/start.png” 表示资源文件里面定义的文件start.png  
  
现在我们来解释下QFile file("text.txt")加载不成功 ，如果text.txt是在可执行文件同级目录下，是可以打开的

的，否则则打不开

下面讲几个Qt常用的获取路径的函数

**[cpp]** [view plain](http://blog.csdn.net/Andy_93/article/details/52831175) [copy](http://blog.csdn.net/Andy_93/article/details/52831175) [print?](http://blog.csdn.net/Andy_93/article/details/52831175)

1. 1 .获取应用程序可执行文件所在的目录: QCoreApplication::applicationDirPath();
2. QString applicationDirPath;
3. applicationDirPath = QCoreApplication::applicationDirPath();
4. qDebug()<<"applicationDirPath"<<applicationDirPath;
6. 2.获取应用程序可执行文件的文件路径: QCoreApplication::applicationFilePath();
7. QString applicationFilePath;
8. applicationFilePath = QCoreApplication::applicationFilePath();
9. qDebug()<<"applicationFilePath"<<applicationFilePath;
11. 3.获取应用程序当前工作目录的绝对路径：QString QDir::currentPath()  (这个类似于“./”操作)
12. QString currentPath;
13. QDir dir;
14. currentPath=dir.currentPath();
15. qDebug()<<"path"<<currentPath;
17. 将相对路径转化为绝对路径：
18. QDir temDir("../../image.png");
19. QString filePath = temDir.absolutePath();
20. 这样就获得了一个文件的绝对路径了。

1 .获取应用程序可执行文件所在的目录: QCoreApplication::applicationDirPath();

QString applicationDirPath;

applicationDirPath = QCoreApplication::applicationDirPath();

qDebug()<<"applicationDirPath"<<applicationDirPath;

2.获取应用程序可执行文件的文件路径: QCoreApplication::applicationFilePath();

QString applicationFilePath;

applicationFilePath = QCoreApplication::applicationFilePath();

qDebug()<<"applicationFilePath"<<applicationFilePath;

3.获取应用程序当前工作目录的绝对路径：QString QDir::currentPath() (这个类似于“./”操作)

QString currentPath;

QDir dir;

currentPath=dir.currentPath();

qDebug()<<"path"<<currentPath;

将相对路径转化为绝对路径：

QDir temDir("../../image.png");

QString filePath = temDir.absolutePath();

这样就获得了一个文件的绝对路径了。

再来讲讲Qt资源系统：Qt 资源系统是一个跨平台的资源机制，用于将程序运行时所需要的资源以二进制的形式存储于可执行文件内部。 如果你的程序需要加载特定的资源（图标、文本翻译等）， 那么，将其放置在资源文件中，就再也不需要担心这些文件的丢失。也就是说，如果你将资源以资源文件形式存储，它是会编译到可执行文件内部。

所以再来讲讲Qt资源文件qrc文件的创建。  
方法1、使用qtdesinger创建.qrc文件  
  （1）在资源管理器窗口中选择新建一个资源文件.qrc，保存文件即可；  
  （2）在资源管理器窗口中选择添加资源和文件即可；

方法2、手工写文件的方法，.qrc可以用记事本打开，大概的形式是这样的：

**[html]** [view plain](http://blog.csdn.net/Andy_93/article/details/52831175) [copy](http://blog.csdn.net/Andy_93/article/details/52831175) [print?](http://blog.csdn.net/Andy_93/article/details/52831175)

1. **<RCC>**
2. **<qresource** prefix="/" **>**  //前缀
3. **<file>**image/text.png**</file>**
4. **</qresource>**
5. **</RCC>**

<RCC>

<qresource prefix="/" > //前缀

<file>image/text.png</file>

</qresource>

</RCC>

然后在pro文件中添加一句   
RESOURCES  = xxx.qrc  
  
这样在使用的时候QPixmap(":/image/text.png");就能加载成功了  
  
同时为了以后我们修改text.png路径的时候我们还需要一处处的修改源代码，所以在建立资源qrc的时候，给每个资源文件起一个别名

如 :

   <RCC>  
      <file alias="image\_copy.png">images/copy.png</file>  
  </RCC>  
  这时该文件可以通过QPixmap(":/image\_copy.png");能加载成功。  
  但通过QPixmap(":/images/copy.png");加载不成功，QT里“:/”表示对资源的引用，不是表示当前目录

==>使用别名是为了日后资源文件路径改变而去改动源代码。用别名后，只需要修改qrc文件即可！

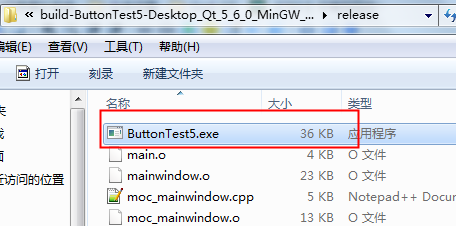
# Qt生成可执行文件（windows）

**环境：win7 32位 运行Qt5.6**

**所需外部软件：upx391w，Enigma virtual Box**

**以样例的形式说明如下：**

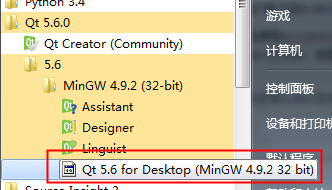
**1.利用QtCreator打开项目release生成项目可执行文件，注意这时生成的 \*.exe文件是不能执行的，需要很多外部.dll文件，debug方式生成的文件有1.8M大。**



**2.提取ButtonTest5.exe运行所需要的dll文件。**

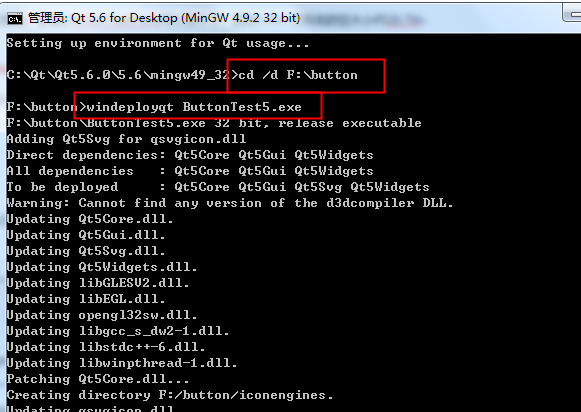
如在F盘新建一个button文件夹，将ButtonTest5.exe复制到button文件夹中，启用Qt自带的[windeployqt](http://blog.csdn.net/syrchina/article/details/50663337)，将ButtonTest5.exe运行需要关联的dll文件拷贝到F 盘的button文件夹中，方法如下（参考：http://blog.csdn.net/syrchina/article/details/50663337）：

在开始菜单找到Qt 5.6 for Desktop (MinGW 4.9.2 32 bit)，运行打开Qt命令行，如下图。

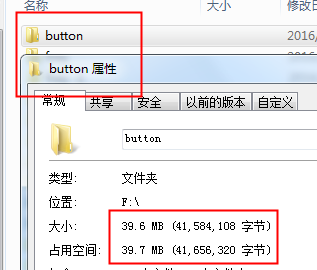


针对F盘下的button文件夹，输入命令行： cd /d F:\button，切换到button文件夹目录下，

然后使用windeployqt工具， 再次输入命令行;windeployqt  ButtonTest5.exe，就可以执行拷贝ButtonTest.exe文件关联的dll文件到button目录下面了，再看看F盘下的button文件夹，所需要的dll文件都已经拷贝进去了，如下图。

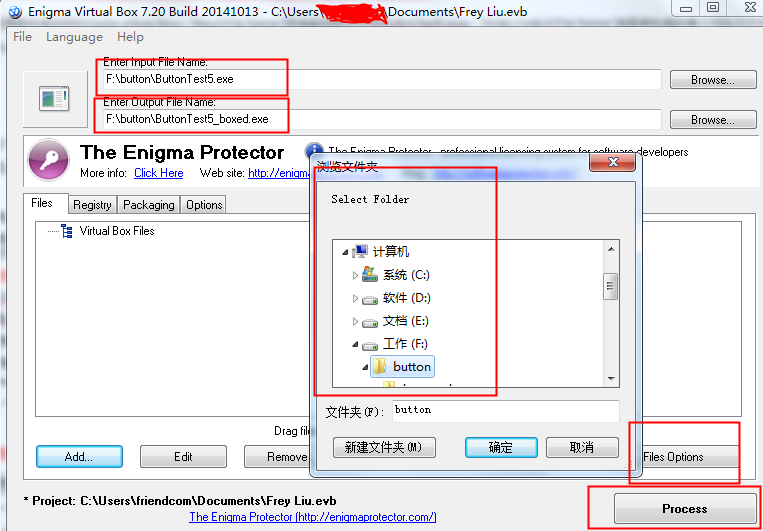


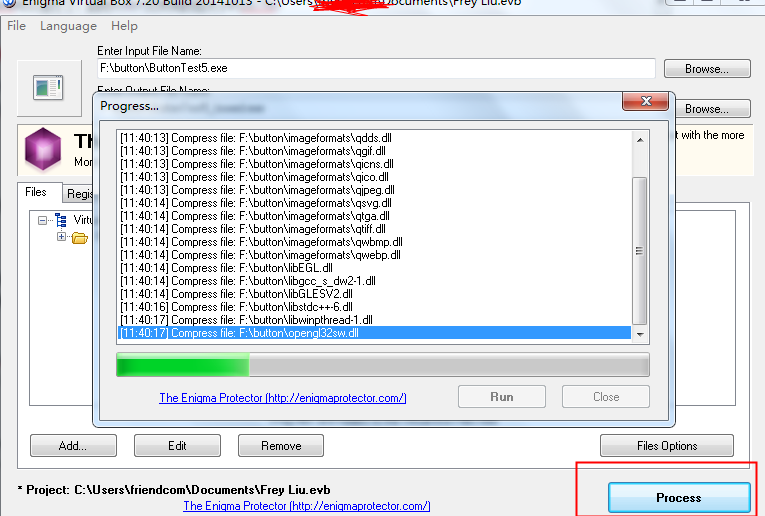
可以打开button文件夹，里面的ButtonTest5.exe可以运行了，此时button文件夹的总大小约39.7M，



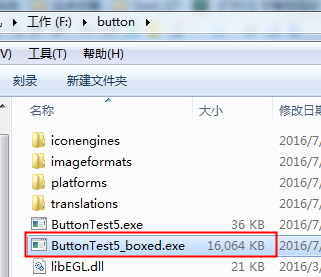
**3.使用****Enigma virtual Box将ButtonTest5.exe文件及需要关联的dll文件生成单个可执行文件**

运行提前安装好的Enigma virtual Box，"Input File Name"项选择可执行文件ButtonTest5.exe，“Enter Output File Name”选择待生成的单个可执行文件存放位置。点击“Add...”选择“Add Folder Recursive”添加dll文件所在的文件夹，选择F盘下的button文件夹。





点击"File Options"按钮，设置文件选项如上图，点击“OK”，最后一步是点击"Process"，即可自动生成所需的单个可执行文件。生成的单个可执行文件有15.6M大小，如下图。



如果对生成的exe大小比较满意，到此就可以结束了。

如果是不想制作成单个可执行文件，而只是想以一个文件夹的方式（即：dll文件+ButtonTest.exe），显然，39M的大小是不能令人满意的，我们需要upx391w对dll文件进行无损压缩。

**4.使用upx391w对dll文件进行压缩**

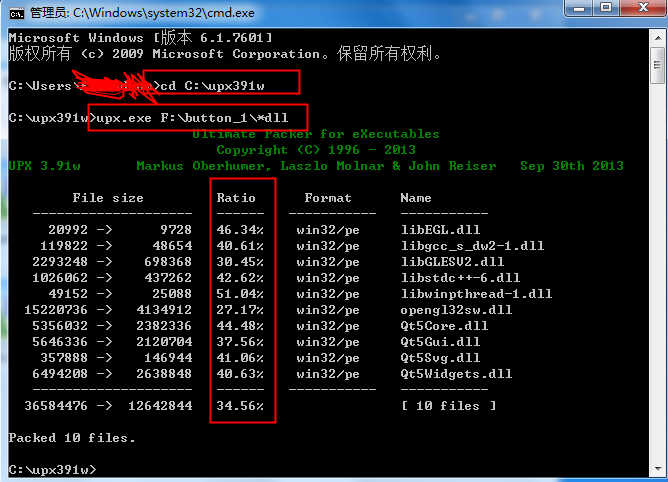
upx391w（下载地址：http://upx.sourceforge.net/），在使用upx压缩前我们复制一份button文件夹，取名为button\_1，方便对比。

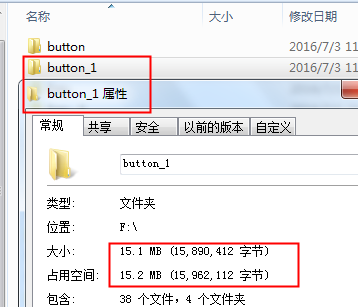
下载upx391w，并将其解压到C盘下面（注：不知道为什么我放到F盘下没法正常使用，可能是系统命令行的原因吧！）

运行电脑的命令行，输入 cd C:\upx391w

再次输入命令行： upx.exe   F:\button\_1\\*dll

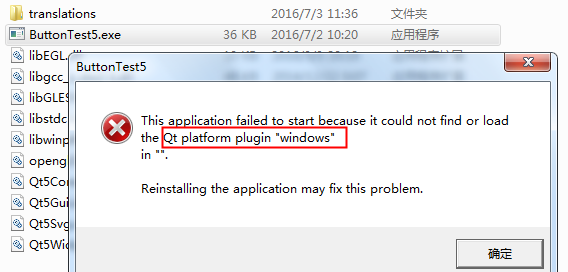
这样就能将button\_1下的所有dll文件进行压缩了，下图可以看出压缩后的文件比压缩前的文件小一倍，





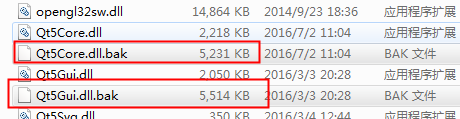
当然你也可以如法炮制对iconengines等文件夹下的dll文件进行压缩，当所有dll文件都压缩完后，整个button\_1文件夹的大小变为15.2M了，和上一步骤生成的单个可执行文件大小差不多了，比原来的39M小了一倍多。

但是，会发现不能对platforms文件夹中的dll文件进行压缩，否则button\_1文件夹下的ButtonTest5.exe不能正常运行了，出现如下图示错误。所以在压缩dll文件时不要对platforms文件夹内的dll文件进行压缩，将platforms文件夹内的dll文件换回来就可以正常运行了。



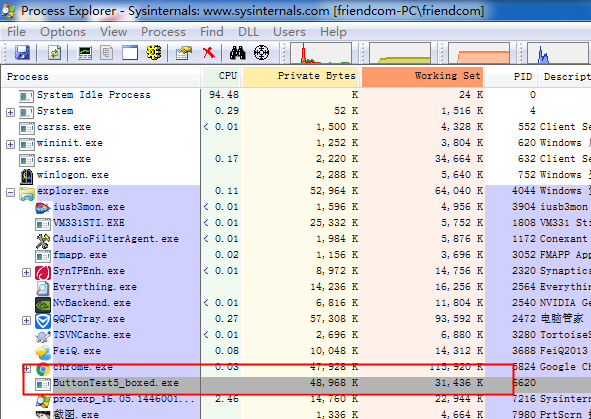
这时你还可以仿照步骤3，用Enigma virtual Box将button\_1文件夹内的ButtonTest5.exe进行制作成单个可执行文件。将生成的单个可执行文件命名为ButtonTest5\_boxed11.exe，发现制作后的可执行文件大小为14.1M，相比较未压缩dll文件制作的单个可执行文件15.6M小了一些。

另外，网上也有不少文章（如：http://blog.csdn.net/gzshun/article/details/7495488）介绍hap-depends + ASpack + filepack方法生成可执行文件的，自己尝试用ASpack对button文件内的dll文件进行压缩，结果生成的.bak文件反而更大，如下图示，不知道是使用方法不对还是下载的软件有问题。尝试filepack封包结果也是不行，当然，很可能是自己使用不当的原因。



写到这里，将自己写的Qt测试小程序生成单个或者是简单压缩以单个文件夹的形式使用基本可以满足需要了。

最后仍有很多不懂的东西，比如生成的单个可执行文件BottonTest5\_boxed11.exe运行起来，通过Process Explorer软件查看内存占用比程序本身还大，可能是Qt本身的原因吧！



# 程序图标

Qt4的时代里，为win下的Qt应用程序添加图标需要创建一个.rc文件，然后在里面输入一行代码，还要修改.pro文件。而在Qt5中变得十分简单：只需要将.ico图标文件放到源码目录，然后在.pro文件中添加代码：　RC\_ICONS=文件名即可。

# .pro文件

简述：

在QT中，有一个工具qmake可以生成一个makefile文件，它是由.pro文件生成而来的。.pro是qmake的工程文件（project）；.pri文件可以把 \*.pro 文件内的一部分单独放到一个 \*.pri 文件内(include)，然后包含进来；.prf文件（feature），和.pri文件类似，也是被包含进.pro文件，只是更隐秘，如CONFIG+=QT；.prl文件（link），主要和生成与使用静态库密切相关。

## 注释

以“#”开始，到这一行结束。

快捷键：Ctrl + /

## CONFIG

指定编译器选项和项目配置，值由qmake内部识别并具有特殊意义。可以指定是生成debug模式还是release模式，还是都生成。也可以用来打开编译器警告或者关闭。还可以用来配置要Qt加载库

以下配置值控制编译标志：

|  |  |
| --- | --- |
| 选项 | 说明 |
| release | 项目以release模式构建。如果也指定了debug，那么最后一个生效。 |
| debug | 项目以debug模式构建。 |
| debug\_and\_release | 项目准备以debug和release两种模式构建。 |
| debug\_and\_release\_target | 此选项默认设置。如果也指定了debug\_and\_release，最终的debug和release构建在不同的目录。 |
| build\_all | 如果指定了debug\_and\_release，默认情况下，该项目会构建为debug和release模式。 |
| autogen\_precompile\_source | 自动生成一个.cpp文件，包含在.pro中指定的预编译头文件。 |
| ordered | 使用subdirs模板时，此选项指定应该按照目录列表的顺序处理它们。 |
| precompile\_header | 可以在项目中使用预编译头文件的支持。 |
| warn\_on | 编译器应该输出尽可能多的警告。如果也指定了warn\_off，最后一个生效。 |
| warn\_off | 编译器应该输出尽可能少的警告。 |
| exceptions | 启用异常支持。默认设置。 |
| exceptions\_off | 禁用异常支持。 |
| rtti | 启用RTTI支持。默认情况下，使用编译器默认。 |
| rtti\_off | 禁用RTTI支持。默认情况下，使用编译器默认。 |
| stl | 启用STL支持。默认情况下，使用编译器默认。 |
| stl\_off | 禁用STL支持。默认情况下，使用编译器默认。 |
| thread | 启用线程支持。当CONFIG包括qt时启用，这是缺省设置。 |
| c++11 | 启用c++11支持。如果编译器不支持c++11这个选项，没有影响。默认情况下，支持是禁用的。 |
| c++14 | 启用c++14支持。如果编译器不支持c++14这个选项，没有影响。默认情况下，支持是禁用的。 |

当使用debug和release选项时(Windows下默认的)，该项目将被处理三次：一次生成一个”meta”Makefile，另外两次生成Makefile.Debug和Makefile.Release。

## QT

指定项目中使用Qt的模块。默认情况下，QT包含core和gui，以确保标准的GUI应用程序无需进一步的配置就可以构建。如果想建立一个不包含Qt GUI模块的项目，可以使用“ -=”操作符。

下面一行将构建一个很小的Qt项目

QT -= gui # 仅仅使用core模块

如果要创建一个界面，里面用到XML及网络相关的类，那么需要包含如下模块

QT += core gui widgets xml network

## TARGET

指定目标文件的名称。默认情况下包含的项目文件的基本名称。

TARGET = LidarPlus

上面项目会生成一个可执行文件，Windows下为LidarPlus.exe，Unix下为LidarPlus

## TEMPLATE

模板变量告诉qmake为这个应用程序生成哪种makefile，可供使用的选项如下：

|  |  |
| --- | --- |
| 选项 | 说明 |
| app | 创建一个用于构建应用程序的Makefile（默认） |
| lib | 创建一个用于构建库的Makefile |
| subdirs | 创建一个用于构建目标子目录的Makefile，子目录使用SUBDIRS变量指定 |
| aux | 创建一个不建任何东西的Makefile。 |
| vcapp | 仅适用于Windows。创建一个Visual Studio应用程序项目 |
| vclib | 仅适用于Windows。创建一个Visual Studio库项目。 |

## SOURCES

指定项目中的所有源文件。

SOURCES += main.cpp\

mainwindow.cpp \

serialtab.cpp \

connectorsdlg.cpp \

serialdataset.cpp \

serialotherset.cpp \

## HEADERS

指定项目中的所有头文件。

HEADERS += mainwindow.h \

serialtab.h \

connectorsdlg.h \

serialdataset.h \

serialotherset.h \

## FROMS

指定项目中的UI文件，这些文件在编译前被uic处理。所有的构建这些UI文件所需的依赖、头文件和源文件都会自动被添加到项目中。

FORMS += \

$$PWD/LogConsoleWidget.ui

## DEFINES

qmake添加这个变量的值作为编译器C预处理器宏（-D选项）

例如：

DEFINES += USE\_MY\_CONNECT

然后在代码中就可以使用：

#ifdef USE\_MY\_CONNECT

// TODO

#else

// TODO

#endif

## RESOURCES

指定资源文件 (qrc) 的名称

RESOURCES += Resource/resource.qrc

## DEPENDPATH

程序编译时依赖的相关路径.

DEPENDPATH += .

## INCLUDEPATH

头文件的包含路径，即指定编译项目时应该被搜索的#include目录。如果路径包含空格，需要使用引号包含。

INCLUDEPATH = c:/include

INCLUDEPATH += "C:/extra headers"

## LIBS

指定链接到项目中的库列表。如果使用Unix -l (library) 和 -L (library path) 标志，在Windows上qmake正确处理库（也就是说，将库的完整路径传递给链接器），库必须存在，qmake会寻找-l指定的库所在的目录。如果路径包含空格，需要使用引号包含路径。

win32:LIBS += c:/mylibs/math.lib

unix:LIBS += -L/usr/local/lib -lmath

## DESTDIR

指定在何处放置目标文件。

CONFIG(release):DESTDIR = $$PWD/../bin

CONFIG(debug, debug|release):DESTDIR = $$PWD/../bin

## MOC\_DIR

指定来自moc的所有中间文件放置的目录（含Q\_OBJECT宏的头文件转换成标准.h文件的存放目录）

## OBJECTS\_DIR

指定所有中间文件.o（.obj）放置的目录。

## RCC\_DIR

指定Qt资源编译器输出文件的目录（.qrc文件转换成qrc\_\*.h文件的存放目录）。

## UI\_DIR

指定来自uic的所有中间文件放置的目录（.ui文件转化成ui\_\*.h文件的存放目录）。

unix:UI\_DIR = ../myproject/ui

win32:UI\_DIR = c:/myproject/ui

## RC\_FILE

指定应用程序资源文件的名称。这个变量的值通常是由qmake或qmake.conf处理，很少需要进行修改。

RC\_FILE += $$PWD/UrgBenri.rc

## RC\_ICONS

仅适用于Windows，指定的图标应该包含在一个生成的.rc文件里。如果RC\_FILE 和RES\_FILE变量都没有设置这才可利用。

RC\_ICONS = myapp.ico

## CODECFORSRC

源文件编码方式。

CODECFORSRC = GBK

## 平台相关处理

根据qmake所运行的平台来使用相应的作用域来进行处理。

win32:RC\_FILE += $$PWD/UrgBenri.rc

macx:ICON = $$PWD/icons/UrgBenri.icns

win32 {

CONFIG += embed\_manifest\_exe

}

Qt4.1.3之后的版本提供了CONFIG选项来提供内嵌manifest文件的功能，embed\_manifest\_dll和embed\_manifest\_exe， 用法是将下面的选项加入pro文件， 如下：

CONFIG += embed\_manifest\_exe

默认情况下embed\_manifest\_dll已经开启。

## 系统变量

QMAKE\_TARGET\_COMPANY：指定项目目标的公司名称，仅适用于Windows

QMAKE\_TARGET\_PRODUCT：指定项目目标的产品名称，仅适用于Windows

QMAKE\_TARGET\_DESCRIPTION：指定项目目标的描述信息，仅适用于Windows

QMAKE\_TARGET\_COPYRIGHT：指定项目目标的版权信息，仅适用于Windows

PACKAGE\_DOMAIN：

PACKAGE\_VERSION：

RC\_CODEPAGE：指定应该被包含进一个.rc文件中的代码页，仅适用于Windows

RC\_LANG：指定应该被包含进一个.rc文件中的语言，仅适用于Windows

RC\_ICONS：指定应该被包含进一个.rc文件中的图标，仅适用于Windows

VERSION：指定程序版本号

BUILD\_NUMBER：

APP\_REVISION：

APP\_VERSION\_DATE：

APP\_VERSION：

# QT Designer自定义控件的创建和使用

## 创建控件

方法1：

直接使用QT Creator的新建控件（建议）。

方法2：

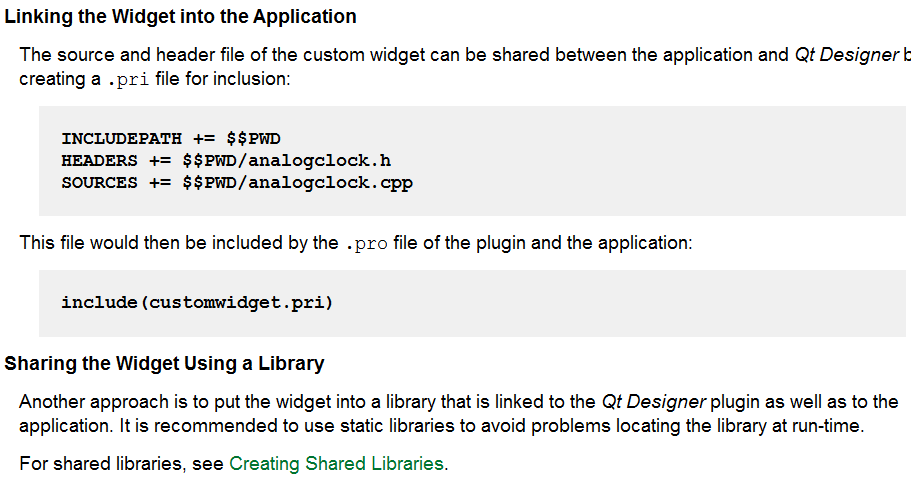
参照HELP中的“Custom Widget Plugin Example”这一例程。

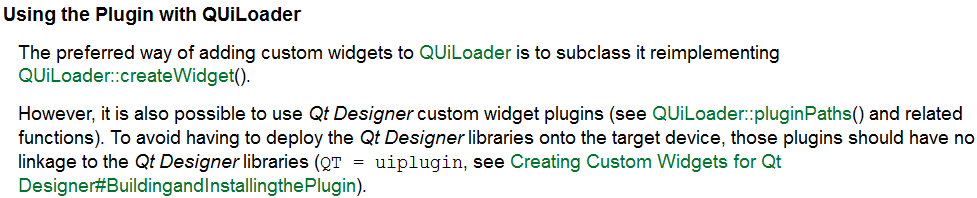
最后，都需要将编译后的dll文件（和Lib文件）放入qt安装目录下的plugins\designer。注意，应该不需要Lib文件。

此时，可在QT Designer中使用该控件，可是还是不能QT CREATOR中的Designer中使用，是因为Creator编译版本和Qt Designer不一致，具体可查阅相关文献。

## 使用控件

在QT Designer中使用了自定义控件后，发现在项目中无法通过编译。HELP中有如下解决办法：





仅尝试过第一个方法，说明：

需要将控件的源文件和头文件放入工程中。（不需要plugin的源文件和头文件，因为这两文件是用来做QT Designer插件的，而项目中不需要）

例如将控件源文件和头文件放入工程目录中，则.pro文档增加如下：

INCLUDEPATH += $$PWD （包含控件的头文件）

HEADERS += $$PWD/\*\*\*.h

SOURCES += $$PWD/\*\*\*.cpp

VS+QT用法

# 创建环境

1. 安装VisualStudio。建议安装较新版本的，比如本文用的版本为2019.
2. 安装qt。需要下载的是后缀为msvc的exe安装文件，MSVC指的是微软的VC编译器。
3. VS在扩展工具中安装QT工具。
4. 安装完成后重启VS，在扩展工具中有QT VS tool。
5. 在QT VS tool->qt opition中add进第二个步骤安装的qt版本。如

D:\Qt\Qt5.7.1\5.7\msvc2015\_64。

1. 此时可以新建QT项目。

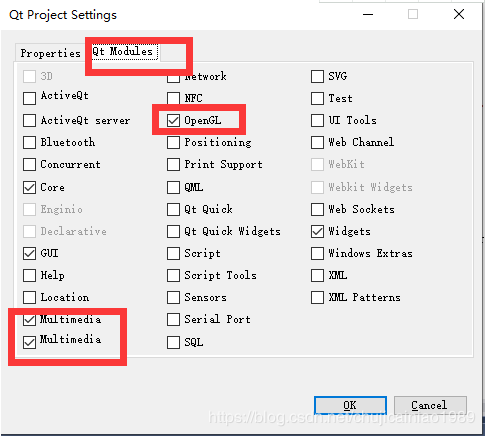
# 增加模块

功能类似在creator中的.pro文件中写入

Qt+=…

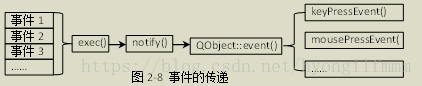
方法：

右键项目->QT project settings，选择QT modules，可以自行添加需要的模块



QT使用

# 事件处理



在Qt中有一个事件循环，该循环负责从可能产生事件的地方捕获各种事件，并把这些事件转换为带有事件信息的对象，然后由Qt的事件处理流程分发给需要处理事件的对象来处理事件。

通过调用QCoreApplication::exec()函数启动事件主循环。主循环从事件队列中获取事件，然后创建一个合适的QEvent对象或其子类类型的对象来表示该事件，在此步骤中，事件循环首先处理所有发布的事件，直到队列为空，然后处理自发的事件，最后处理在自发事件期间产生的已发布事件。注意：发送的事件不由事件循环处理，该类事件会被直接传递给对象。

--------------------- --------------------------

注：1.自发事件：这是由窗口系统生成的，这些事件置于系统队列中，并由事件循环一个接一个地处理。

 2.发布的事件(Posted events)：该类事件由Qt或应用程序生成，这些事件由Qt排队，并由事件循环处理。

 3.发送的事件(Sent events)：该类事件由Qt或应用程序生成，这些事件直接发送到目标对象，不经过事件循环处理。

--------------------- --------------------------

然后，Qt会调用QCoreApplication::notify()函数对事件进行传递(或分发)。

最后，QObject对象调用QObject::event()函数接收事件，event()函数是一个虚函数，是QObject对象处理事件的入口，

在QObject的子类中通常会重写event()函数，比如QWidget类就重写了event()函数。

 event()函数与事件处理函数的关系：event()函数负责把事件传递给目标对象并调用对应的事件处理函数处理事件，比如调用QWidget::keyPressEvent()函数处理键盘按下事件等，注意，QObject中的event()函数并不能实现该功能，这是通过QObject的子类中重写的event()函数实现的，比如调用QWidget::keyPressEvent()函数，就是由在QWidget类中重写的event()函数来完成的。

 event()函数对大多数事件都调用了默认的处理函数，但并不能包括全部的事件，因此，有时我们需要重写该函数，这也是我们处理Qt事件的方式之一。

注意：event()函数不会处理事件，他只是根据事件的类型进行事件的传递(或分发)，若返回true则表示这个事件被接受并进行了处理，否则事件未被处理，需进行进一步传递或丢弃。

其实，在一个对象的event()之前(notify()之后)还会检测是否注册了事件过滤，若是，则现进入被注册对象的事件过滤函数eventFilter()函数。并且，如果此函数返回true，则表示这个事件已经被处理，不再返回原来的目标对象；否则发送给原来的目标对象。

# 事件父子传递规则

基本规则：若事件未被目标对象处理，则把事件传递给其父对象处理，若父对象仍未处理，则再传递给父对象的父对象处理，重复这个过程，直至这个事件被处理或到达顶级对象为止。注意：事件是在对象间传递的，这里是指对象的父子关系，而不是指类的父子关系。

使用QEvent::accept()函数表示接受一个事件，使用QEvent::ignore()函数表示忽略一个事件。也就是说若调用accept()，则事件不会传递给父对象，若调用ignore()则事件会向父对象传递。

Qt默认值是accept (接受事件)，但在QWidget的默认事件处理函数(比如keyPressEvent())中，默认值是ignore()，因为这样可实现事件的传递(即子对象未处理就传递给父对象处理)。因此对事件的接受和忽略，最好是明确的调用accept()和ignore函数。

# [Qt::ConnectionType（信号与槽的传递方式）](https://www.cnblogs.com/senior-engineer/p/6186226.html)

取决于connect第五个参数：

Qt::AutoConnection

自动连接：（默认值）如果信号在接收者所依附的线程内发射，则等同于直接连接。如果发射信号的线程和接受者所依附的线程不同，则等同于队列连接。

Qt::DirectConnection

直接连接：当信号发射时，槽函数将直接被调用。无论槽函数所属对象在哪个线程，槽函数都在发射信号的线程内执行。

Qt::QueuedConnection

队列连接：当控制权回到接受者所依附线程的事件循环时，槽函数被调用。槽函数在接收者所依附线程执行。也就是说：这种方式既可以在线程内传递消息，也可以跨线程传递消息

Qt::BlockingQueuedConnection

与Qt::QueuedConnection类似，但是会阻塞等到关联的slot都被执行。这里出现了阻塞这个词，说明它是专门用来多线程间传递消息的。

# Qt中多线程需要注意的事情

1. 对象在哪个线程中new，就属于哪个线程。Qthread中，只有run函数中创建的对象才属于子线程。
2. 所有父子object必须在同一个线程中
3. 所有事件驱动型的object必须自始至终使用在单线程中，比如qtimer，创建和使用都必须在同一个线程中。
4. GUI相关的object只能在主线程中
5. 在一个线程中delete掉不属于此线程的object是不安全的，除非你确定该object没有处理事务。最好使用[QObject::deleteLater](../qtcore/qobject.html#deleteLater)()。
6. 所有qobject都不是线程安全的。
7. 一定要注意多线程中使用信号槽相关的问题。参见“3 Qt::ConnectionType（信号与槽的传递方式）”

# Qt窗口关闭发生的事情

1. 点击关闭或者执行QWidget::close()时，First it sends the widget a [QCloseEvent](../qtgui/qcloseevent.html). The widget is [hidden](qwidget.html#hide) if it [accepts](../qtcore/qevent.html#accept) the close event. If it [ignores](../qtcore/qevent.html#ignore) the event, nothing happens. The default implementation of [QWidget::closeEvent](qwidget.html#closeEvent)() accepts the close event. If the widget has the [Qt::WA\_DeleteOnClose](../qtcore/qt.html#WidgetAttribute-enum) flag, the widget is also deleted. A close events is delivered to the widget no matter if the widget is visible or not.
2. 当最后一个拥有“[Qt::WA\_QuitOnClose](file:///D:\康驰学习笔记\qtcore\qt.html#WidgetAttribute-enum)”属性的visible primary window (i.e. window with no parent)被关闭时，将发生以下事情：

1）首先发送[QApplication::lastWindowClosed](file:///D:\康驰学习笔记\qtgui\qguiapplication.html#lastWindowClosed)()信号。

2）若QGuiApplication的quitOnLastWindowClosed属性为真时，应用程序quit()，退出主循环。

若假，则不做任何事了。

QT类与函数

# Qfile文件操作

## 清空文件内容的方法

file.*open*(QIODevice::*Truncate*);

Truncate表示覆盖当前文件。

# Qmediaplay的用法

QString musicPath;

musicPath = QCoreApplication::applicationDirPath()+"/music";

QMediaPlaylist\* playlist = new QMediaPlaylist;

playlist->addMedia(QUrl::fromLocalFile(musicPath+"/纯音乐 - 天空之城.mp3"));

playlist->addMedia(QUrl::fromLocalFile(musicPath+"/昨日重现.mp3"));

playlist->setCurrentIndex(1);

playlist->setPlaybackMode(QMediaPlaylist::*Loop*);

QMediaPlayer\* player = new QMediaPlayer;

player->setPlaylist(playlist);

player->setVolume(7);

player->play();

# fontMetrics（）返回一串字符的长宽等信息

**[Q](../qtgui/qfontmetrics.html)FontMetrics** QWidget::**fontMetrics**() const

例如：fontMetrics().width(“hello world”)

会返回“hello world”在当前widge设置的font格式下的字符串的宽度。

fontMetrics().size(0, “hello world”);\

会返回“hello world”在当前widge设置的font格式下的字符串的Qsize.

# 时间和日期

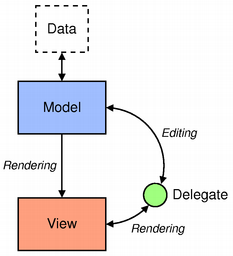
|  |  |
| --- | --- |
| [**QDate**](http://www.kuqin.com/qtdocument/qdate.html) | 日期功能 |
| [**QDateEdit**](http://www.kuqin.com/qtdocument/qdateedit.html) | 日期编辑器 |
| [**QDateTime**](http://www.kuqin.com/qtdocument/qdatetime.html) | 日期和时间功能 |
| [**QDateTimeEdit**](http://www.kuqin.com/qtdocument/qdatetimeedit.html) | 把一个QDataEdit和一个QTimeEdit窗口部件组合为一个单一的窗口部件来编辑日期时间 |
| [**QTime**](http://www.kuqin.com/qtdocument/qtime.html) | 时钟时间功能 |
| [**QTimeEdit**](http://www.kuqin.com/qtdocument/qtimeedit.html) | 时间编辑器 |
| [**QTimer**](http://www.kuqin.com/qtdocument/qtimer.html) | 定时器信号和单触发定时器 |

# [model/view 架构](http://www.qter.org/forum.php?mod=viewthread&tid=662&extra=page%3D1)

有时，我们的系统需要显示大量数据，比如从数据库中读取数据，以自己的方式显示在自己的应用程序的界面中。早期的 Qt 要实现这个功能，需要定义一个组件，在这个组件中保存一个数据对象，比如一个列表。我们对这个列表进行查找、插入等的操作，或者把修改的地方写回，然后刷新组件进行显示。这个思路很简单，也很清晰，但是对于大型程序，这种设计就显得苍白无力。比如，在一个大型系统中，你的数据可能很大，全部存入一个组件的数据对象中，效率会很低，并且这样的设计也很难在不同组件之间共享数据。如果你要几个组件共享一个数据对象，要么你就要用存取函数公开这个数据对象，要么你就必须把这个数据对象放进不同的组件分别进行维护。

Smalltalk 语言发明了一种崭新的实现，用来解决这个问题，这就是著名的 MVC 模型。对这个模型无需多言。MVC 是  Model-View-Controller 的简写，即模型-视图-控制器。在 MVC 中，模型负责获取需要显示的数据，并且存储这些数据的修改。每种数据类型都有它自己对应的模型，但是这些模型提供一个相同的 API，用于隐藏内部实现。视图用于将模型数据显示给用户。对于数量很大的数据，或许只显示一小部分，这样就能很好的提高性能。控制器是模型和视图之间的媒介，将用户的动作解析成对数据的操作，比如查找数据或者修改数据，然后转发给模型执行，最后再将模型中需要被显示的数据直接转发给视图进行显示。MVC 的核心思想是分层，不同的层应用不同的功能。

Qt 4 开始，引入了类似的 model/view 架构来处理数据和面向最终用户的显示之间的关系。当 MVC 的 V 和 C 结合在一起，我们就得到了 model/view 架构。这种架构依然将数据和界面分离，但是框架更为简单。同样，这种架构也允许使用不同界面显示同一数据，也能够在不改变数据的情况下添加新的显示界面。为了处理用户输入，我们还引入了委托（delegate）。引入委托的好处是，我们能够自定义数据项的渲染和编辑。

Model View 概览

如上图所示，模型与数据源进行交互，为框架中其它组件提供接口。这种交互的本质在于数据源的类型以及模型的实现方式。视图从模型获取模型索引，这种索引就是数据项的引用。通过将这个模型索引反向传给模型，视图又可以从数据源获取数据。在标准视图中，委托渲染数据项；在需要编辑数据时，委托使用直接模型索引直接与模型进行交互。

总的来说，model/view 架构将传统的 MV 模型分为三部分：模型、视图和委托。每一个组件都由一个抽象类定义，这个抽象类提供了基本的公共接口以及一些默认实现。模型、视图和委托则使用信号槽进行交互：

来自模型的信号通知视图，其底层维护的数据发生了改变；

来自视图的信号提供了有关用户与界面进行交互的信息；

来自委托的信号在用户编辑数据项时使用，用于告知模型和视图编辑器的状态。

所有的模型都是 QAbstractItemModel 的子类。这个类定义了供视图和委托访问数据的接口。模型并不存储数据本身。这意味着，你可以将数据存储在一个数据结构中、另外的类中、文件中、数据库中，或者其他你所能想到的东西中。我们将在后面再详细讨论这些内容。

QAbstractItemModel 提供的接口足够灵活，足以应付以表格、列表和树的形式显示的数据。但是，如果你需要为列表或者表格设计另外的模型，直接继承 QAbstractListModel 和 QAbstractTableModel 类可能更好一些，因为这两个类已经实现了很多通用函数。关于这部分内容，我们也会在后文中详述。

Qt 内置了许多标准模型：

QStringListModel：存储简单的字符串列表。

QStandardItemModel：可以用于树结构的存储，提供了层次数据。

QFileSystemModel：本地系统的文件和目录信息。

QSqlQueryModel、QSqlTableModel 和 QSqlRelationalTableModel：存取数据库数据。

正如上面所说，如果这些标准模型不能满足你的需要，就必须继承 QAbstractItemModel、QAbstractListModel 或者 QAbstractTableModel，创建自己的模型类。

Qt 还提供了一系列预定义好的视图：QListView 用于显示列表，QTableView 用于显示表格，QTreeView 用于显示层次数据。这些类都是 QAbstractItemView 的子类。这意味着，如果你要创建新的视图类，则可以继承 QAbstractItemView。

QAbstractItemDelegate 则是所有委托的抽象基类。自 Qt 4.4 依赖，默认的委托实现是 QStyledItemDelegate。但是，QStyledItemDelegate 和 QItemDelegate 都可以作为视图的编辑器，二者的区别在于，QStyledItemDelegate 使用当前样式进行绘制。在实现自定义委托时，推荐使用 QStyledItemDelegate 作为基类，或者结合 Qt style sheets。

如果你觉得 model/view 模型过于复杂，或者有很多功能是用不到的，Qt 还有一系列方便使用的类。这些类都是继承自标准的视图类，并且继承了标准模型。这些类并不是为其他类继承而准备的，只是为了使用方便。它们包括 QListWidget、QTreeWidget 和 QTableWidget。这些类远不如视图类灵活，不能使用另外的模型，因此只适用于简单的情形。

QT的一些问题

# Qt 之 运行Qt Creator出现cannot find -lGL的问题的两种解决

一般是linux环境中出现。

有两种原因，

（1）一种是没有按照libGL库，那么就安装：

apt-get install libgl1-mesa-dev

或者yum install mesa-libGL

（2）一种是装了，但是路径不对，那么就配置一下路径：

$ locate libGL

/usr/lib/i386-linux-gnu/mesa/libGL.so.1

/usr/lib/i386-linux-gnu/mesa/libGL.so.1.2.0

/usr/lib/x86\_64-linux-gnu/libGLEW.so.1.10

/usr/lib/x86\_64-linux-gnu/libGLEW.so.1.10.0

/usr/lib/x86\_64-linux-gnu/libGLEWmx.so.1.10

/usr/lib/x86\_64-linux-gnu/libGLEWmx.so.1.10.0

/usr/lib/x86\_64-linux-gnu/libGLU.so.1

/usr/lib/x86\_64-linux-gnu/libGLU.so.1.3.1

/usr/lib/x86\_64-linux-gnu/mesa/libGL.so.1

/usr/lib/x86\_64-linux-gnu/mesa/libGL.so.1.2.0

/usr/lib/x86\_64-linux-gnu/mesa-egl/libGLESv2.so.2

/usr/lib/x86\_64-linux-gnu/mesa-egl/libGLESv2.so.2.0.0

$ sudo ln -s /usr/lib/x86\_64-linux-gnu/mesa/libGL.so.1 /usr/lib/libGL.so

（放着放入其它该工程可以找到库的地方）

QT一些未明白的问题

MainWindow::MainWindow(QWidget \*parent)  
        : QMainWindow(parent)  
{  
        ui.setupUi(this);  
        time = new QTimer();  
        time->start(1000);  
        time2 = new QTimer();  
        time2->start(5000);  
        connect(time,SIGNAL(timeout()),this,SLOT(slotTimeoutProcess()));  
        connect(time, SIGNAL(timeout()), this, SLOT(slotTimeoutProcess2()));  
}  
void MainWindow::slotTimeoutProcess(）{//槽1  
        //do something  
}  
  
void MainWindow::slotTimeoutProcess2(void) {//槽2  
        QDialog dialog;  
        dialog.exec();  
}  
  
  
（1）当我将time的超时信号绑定到槽1和槽2时，当槽2的dialog调用exec（）后，进入模态模式。此后永远无法进入槽1  
（2）当我将time超时绑定槽2，time2超时绑定槽1，则可以正常按时进入槽1。

一些分析：

Timeout（）信号是在timerEvent()事件处理函数中发射的，在Qt内部机制中，timeout()信号连接的槽函数2应该也在timeEvent()事件处理函数中处理，而因为exce()而被堵塞，估计因为此事件处理函数不可重入，所以无法再次进来。

判断依据：

自己写了一个KcTimer继承自QTimer，重写virtual void timerEvent(QTimerEvent\* e);并在此函数中发射自定义的void kcTimeout(void)。

void KcTimer::timerEvent(QTimerEvent\* e)

{

emit kcTimeout();

qDebug() << "kctimer进入超时事件";//在此处打断点。

}

MainWindow::MainWindow(QWidget \*parent)

: QMainWindow(parent)

{

ui.setupUi(this);

timeKc = new KcTimer(this);

timeKc->start(1000);

connect(timeKc, SIGNAL(kcTimeout()), this, SLOT(slotTimeoutProcess()));

}

void MainWindow::slotTimeoutProcess(void) {

QDialog dialog;

dialog.exec();

}

在上述代码注释“在此处打断点”处打断点。可以发现，一旦槽函数因为exec()而被堵塞，则再不会进入断点处。

但是感觉问题还不太清晰

1. 槽函数的确是在timeEvent()中调用的吗？
2. 事件处理函数的确不可重入吗？还是因为其他机制导致无法再进入槽函数？

20190804

1. 解决上述问题1：应该是的。connect的第五个参数采用Qt::DirectConnection时，是直接调用；采用Qt::QueuedConnection时，是在下一个事件循环中才调用。当发射者和接收者在同一个线程中时，默认采用Qt::DirectConnection；在不同线程中时，默认采用Qt::QueuedConnection。
2. 估计是此timerevent必须要对其处理完毕，才能产生新的timerevent。因为一直在event()函数（更细致点，是在timeEvent()中）中，所以系统认为一直正在处理此timerevent，所以不会产生新的timerevent.

# Qserialport与多线程

1. 主线程中使用QSerialPort::[waitForReadyRead](qiodevice.html#waitForReadyRead)()没有作用?
2. 次线程中必须先使用QSerialPort::[waitForReadyRead](qiodevice.html#waitForReadyRead)()，然后才能使用QSerialPort::readAll，否则读不了数据;另外，如果在次线程中无限循环write数据，只能write一次，必须在write的后面使用一次[waitForReadyRead](file:///D:\康驰学习笔记\xin\qiodevice.html#waitForReadyRead)()，方才能继续write数据.???